NGAIZA SERIES OF COMPUTER SCIENCE

ENJOY & MASTER COMPUTER SCIENCE

INSIDE THE MACHINE

Computer Architecture • Processor Performance • Advanced Networks • C++

Ngaiza Books App
Scan to download on Google Play

Scan for interactive solutions, extra questions, and all Ngaiza titles on your phone.

Enjoy & Master Computer Science

Topical Questions and Answers: Computer Architecture · Processor Performance · Advanced Networks · C++

Ngaiza Series of Advanced Computer Science

© 2026 Ngaiza Education Hub. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means: electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of the publisher, except for brief quotations in reviews or academic commentary.

First Edition, 2026

ISBN 978-9976-5530-9-3

Writers

Eng. Mwamvua Bakari, Eng. Najma Majid, and Eng. Robert Modestus

Editor

Dr. Isakwisa Tende

Series Creator and Director

Ngaiza Lusima

Published by

Ngaiza Education Hub

P.O. Box 60791

Mbezi Beach, Dar es Salaam, Tanzania

www.ngaizabooks.com

contact@ngaizabooks.com

Every effort has been made to ensure the accuracy of the content in this book. The publisher welcomes corrections and suggestions for future editions.

TABLE OF CONTENTS

Preface i
Student Survival Guide iii

1 Computer Architecture 1

Architecture versus organisation, and the von Neumann machine 1
Harvard, modified Harvard, and Flynn's taxonomy 6
The fetch-decode-execute cycle, the CPU registers, and RTL 15
Boolean laws and algebraic simplification 23
De Morgan's theorems, algebraically and at gate level 30
Truth tables both ways: minterms, maxterms, sum of products and product of sums 36
Karnaugh maps 43
The logic gates, XNOR, and the universality of NAND and NOR 51
Expression to circuit, and circuit to expression 57
Adders and subtractors 65
Multiplexers, decoders, encoders and comparators 73
Sequential logic: flip-flops, registers, counters and state tables 79
The memory hierarchy, locality, cache mechanics and average access time 86
Cache mapping, address decomposition, write policy and replacement 93
Memory capacity arithmetic, chip organisation and virtual memory 101
Instruction set architecture: formats, encoding, addressing modes and RISC against CISC 108
Programmed input and output, interrupts and direct memory access 116

2 Performance and Optimisation of the Processor 125

Pipelining: the idea, the five stages and the overlap 125
The numerical spine: clock, cycles, CPI, throughput and speed-up 131
Hazards, and the structural hazard 139
Data hazards, forwarding and the load-use stall 146
Control hazards, branch prediction and the branch penalty 153
Instruction level parallelism: multiple issue, dependences and out-of-order execution 159
Thread level parallelism: threads, simultaneous multithreading and multicore 167
Data level parallelism: SIMD lanes, vector width and SIMT 174
Amdahl's law and the limits of parallel speed-up 181

3 Advanced Computer Networks 188

Network models and the seven layers of OSI 188
What each layer does: protocol, PDU and device 194
The TCP/IP suite and its mapping onto OSI 201
Encapsulation, de-encapsulation and protocol overhead 207
The physical layer: signalling, encoding and impairment 215
Framing, error detection and the cyclic redundancy check 221

MAC addressing, ARP and the switched LAN 228
Media access control and the two domains 235
VLANs and network segmentation 242
The network layer and its design issues 249
Routing algorithms classified 255
Routing computed: Dijkstra, Bellman-Ford and the routing table 262
ICMP, congestion control and quality of service 269
IPv4 addressing: classes, masks and binary conversion 276
Subnetting I: borrowed bits, block size and host range 285
Subnetting II: VLSM, supernetting and summarisation 293
IPv6: notation, compression, the header and coexistence 300
The transport layer: TCP, UDP and reliable delivery 307
Ports, sockets and the application protocols 315
Guided transmission media and their limits 321
Unguided media, satellite and the spectrum 328
Structured cabling: T568-A and B, the toolkit and the test 335
LAN design, documentation and simulation 342
Network security: threats, the CIA triad and controls 349
Cryptography, wireless security and modern defence 356
PSTN, mobile networks and telephony 363

4 Object Oriented Programming with C++ 371

Paradigms, objects and the four pillars 371
From source code to a running program: the toolchain, the build pipeline and the four error classes 380
The anatomy of a C++ program, and console input and output 389
Data types, variables, constants, scope and lifetime 400
Operators, precedence, associativity and expression evaluation 412
Selection: if, else, nested tests and switch 425
Iteration: while, do-while, for, nested loops and early exit 439
Functions: parameters, overloading, default arguments and recursion 455
Arrays, one dimensional and two dimensional 470
Strings: char arrays, the C string library and std::string 485
Classes, objects, encapsulation, constructors and destructors 499
Inheritance, polymorphism and abstraction 518

Index 540

PREFACE

This book was written because you deserve better.

Advanced Level Computer Science rewards precision, yet many students arrive at the paper having learned to recite definitions they cannot apply and to copy diagrams they cannot follow. The subject looks approachable, because everybody has used a computer and most students have seen a line of code, and that familiarity is exactly the trap. Candidates memorise lists of features, sit an examination that asks them to define, to calculate, to draw and to judge, and leave believing that computer science is a matter of technical words confidently spelled. It is not. It is one of the few subjects in which almost every claim can be settled by working it out.

Computer science is the study of what a machine can be made to do, and of the cost of making it do so. It is built from parts you can name and rules you can check: a gate that gives one output for each pattern of inputs, a cache that either answers or does not, a mask that either leaves room for the machines a laboratory must connect or does not, a program that either compiles or does not. Nothing in it depends on opinion. That is what makes it learnable, and it is also why it is unforgiving: where an answer can be decided, it will be. Mastering it under examination conditions takes more than a book to read. It takes practice, attempting real questions, defining, calculating, drawing and judging, and learning, answer by answer, how to turn understanding into marks. That is what this book sets out to give you.

Why a book of questions and answers?

You do not learn to reason by reading someone else's reasoning, and you do not learn to draw a circuit by admiring a finished one. You learn by attempting a question, getting it wrong, finding out exactly why, and trying again. This book is built entirely around that truth. Every idea in this volume is turned into questions you must answer yourself, each followed at once by a full model answer that shows not only the right conclusion but the definition, the working, the drawing and the judgement that earn examination marks. Used honestly, with the model covered until you have written your own, it trains the four things the examination really tests: defining with precision, reasoning from cause to effect, calculating where a number is available, and judging how far a claim will hold.

How this book is organised

This volume covers the first half of the Form Five course in four chapters: computer architecture, the performance and optimisation of the processor, advanced computer networks, and object oriented programming with C++. Each rests on the one before, and the whole of it rests on the first, on the shape of the machine and the buses along which everything inside it travels. The second half of the course, which carries Java, Python, operating systems and databases, is the companion volume, COMMANDING THE MACHINE. Neither book is a summary of the other.

Every question is followed at once by its model answer, so that you can attempt it and check yourself immediately. Within each chapter the questions rise through four tiers.

BINDER questions anchor the core. They fix the definition, the mechanism and the arithmetic that must be right before anything else can be, and even here they ask you to reason rather than merely to name. If you can answer these, you own the idea.

REAL questions carry that core into the country itself, from a borehole pump at Sumbawanga to a weighbridge at Kiteto and an office block at Kinondoni, where the subject actually lives. Many of them hand you another candidate's working and ask you to find the error in it, which is the fastest way to stop making it yourself.

HOT questions push you to examination standard and beyond, drawing several ideas together and demanding a genuine judgement rather than a routine.

BUILD questions ask you to write the program, not to describe one. Half the marks in this subject are eventually won or lost at a keyboard, and the C++ chapter is where that begins.

Every answer in the book ends with a Master Key, a single sentence built to be carried into the hall. The figures, the tables and the program listings are there to be worked with rather than looked at: a diagram in this subject is a second answer, marked on its own, and a listing is worth nothing until you have traced it or run it.

Before the questions begin, a short Student Survival Guide shows you how to work this book, and how to beat the three habits that quietly cost good students their marks: the true fact attached to the wrong part of the

machine, the sentence written where a number was available, and the drawing nobody else can follow. Read it first. It is the shortest and most important part of the book. At the back, a full index takes you from any term to the pages where it is defined, drawn or applied.

The team behind this book

Enjoy and Master Computer Science is the work of the Ngaiza Education Hub, built by a team of experienced computer science educators and engineers who share one conviction: that Tanzanian students deserve world-class materials written for their context, their course, and their aspirations.

Writers: Eng. Mwamvua Bakari, Eng. Najma Majid and Eng. Robert Modestus.

Editor: Dr. Isakwisa Tende.

Designer: Mr. Frank Mwita.

I am grateful to every member of this team for their dedication, their expertise, and their refusal to settle for "good enough." Every program printed in these pages was compiled and run before it was set in type, and every figure was measured rather than eyeballed, which is slow work that no reader will ever see and every reader will benefit from. I thank too the students whose hard questions and honest struggles shaped every answer in this book. They reminded us, again and again, that a question a student cannot learn from has failed, no matter how correct its computer science may be.

To every student of Advanced Computer Science opening this book for the first time: the subject is not as hard as its reputation, and it is not as easy as its first pages feel. It is as exact as the definitions you will learn to trust, and as satisfying as an answer that you can prove. Give it your honest effort, with the model answer covered and your pencil moving, and this book will give you its honest best in return.

Ngaiza Lusima
Series Creator and Director

STUDENT SURVIVAL GUIDE

How to turn what you know into marks.

Why the Nearly Right Answer Earns Nothing

Computer science is the subject in which a candidate can know the material and still be marked down to a third of the paper. Nothing here is hidden. The gates are on the page, the formula for average memory access time is four symbols long, and a subnet mask is arithmetic a Form Two student could do. What the paper tests is not whether you have met these things but whether you can put each fact against the right part, reach for the number when a number is there, and draw so that somebody else can follow it. Three habits of imprecision cost more marks in this subject than ignorance ever does.

The first trap is the true fact attached to the wrong part. This book opens by warning you about it and repeats the warning at the head of every chapter, because it is the single commonest way a prepared candidate loses marks. Registers are architecture and cache is organisation. A hub sees signals, a switch sees frames and a router sees packets, and what a device cannot read it cannot act on. A port is not the connector on the back of the machine, and an answer describing that connector earns nothing at all, however accurately it describes it. Every one of those sentences is true of something; each is worthless attached to the wrong thing. Name the part, then state the property, in that order, and half of this danger disappears.

The second trap is the sentence written where a number was available. Questions in this subject are frequently answered in words when they could have been answered in arithmetic, and the examiner rewards the candidate who calculates. Given a hit time, a miss rate and a penalty, an answer explaining what caching is for scores a fraction of what one line of substitution scores. The working is not the route to the mark; the working is where the mark sits. Write the division on the page before you write the answer. A wrong figure with sound working still earns something; a bare right figure earns a fraction of what a derived one earns.

The third trap is the drawing nobody else can follow. A circuit, a Karnaugh map, a truth table and a network diagram are not illustrations of your answer. They are the answer, and they carry marks of their own. A drawing loses those marks not by being wrong but by being unreadable: wires that go nowhere named, gates in no particular column, a map whose groups were never ringed. Label the output of each gate as you draw it, and write beside it the expression it carries. Unlabelled wires are where a three gate circuit turns into a guess, and a labelled drawing can be checked without redrawing it.

The same question, answered by two candidates.

The question. Explain what average memory access time measures, state the formula, and work through an example, given a hit time of 4 nanoseconds, a miss rate of 5 percent and a miss penalty of 80 nanoseconds.

The answer that fails. Average memory access time is the average time the processor waits for memory. A cache is faster than main memory, so the higher the hit rate the lower the average time, which is why designers make caches large. Here the average would be about 42 nanoseconds. (Every sentence before the last is true, and the paper is not paying for it. The figure is the mean of 4 and 80, which is what the formula exists to prevent.)

The answer that scores. Average memory access time is the mean time an access takes once hits and misses are weighted by how often each happens, and it is found from AMAT = hit time + (miss rate x miss penalty). The hit time is paid on every access, hit or miss, because the cache is always searched first; the penalty is paid only by the accesses that miss, so it enters multiplied by the miss rate. Substituting, 4 + (0.05 x 80) = 4 + 4 = 8 nanoseconds. No single access takes 8 nanoseconds: a hit takes 4 and a miss takes 84, and 8 is the mean over a great many accesses.

Both candidates understood caching. The second wrote down a formula, said why it has that shape, substituted in one visible line and then said what the answer meant, and that is the whole distance between a pass and a distinction, repeated across a three hour paper. This book is built to close it.

How to Use This Book

This is not a book to read. It is a book to work, with a pen in your hand and paper beside you. A model answer you have only read feels learned and is gone by morning; one you have struggled toward, and then checked against your own, is learned for good. Work every question in four steps.

Cover the model answer. Read the question, then write your full answer on paper, circuit and all, exactly as you would in the hall, before you look at anything.

Uncover and compare. Read the model twice, once for the computer science and once for the way it is set down: its opening definition, its working, its drawing and its labels. Did you reach the same conclusion, and did you reach it the same way?

Mark yourself honestly. Award a mark only where your answer truly earned it. An unlabelled wire, a missing unit, a division you did in your head and never wrote down: each of those is a lost mark, not a near one.

Fix one thing, then move on. Correct your single biggest error in your own hand, and carry that lesson into the next question.

Climb the ladder, and do not skip the bottom rung. Every chapter rises through four tiers. BINDER fixes the core: the definition, the mechanism and the arithmetic that must be right before anything else can be. REAL carries that core into named Tanzanian settings, a borehole pump at Sumbawanga, a weighbridge at Kiteto, an office block at Kinondoni a hundred and forty metres from its server room. HOT asks for judgement: to weigh, to compare, and to decide how far a claim holds. BUILD asks for a working program. The tiers are cumulative, and a candidate who jumps to HOT because it looks like the real examination will find the ground missing beneath.

Work the chapters in order. The four chapters run from the ground up: computer architecture, the performance of the processor, advanced computer networks, and object oriented programming with C++. Everything else in the subject stands on the first. The programs you write are instructions for the machine described there, and the networks you build move data along its buses.

Read the Master Key last, never first. Every answer in this book closes with one sentence built to be carried into the hall. It is a summary, not a shortcut, and it is worth nothing to a student who has not first done the work it sums up.

Know that this volume is one of two. The Form Five course runs across INSIDE THE MACHINE, which you are holding, and COMMANDING THE MACHINE, which carries Java, Python, operating systems and databases. Neither volume is a summary of the other, and the syllabus asks for both.

How to Write Computer Science for Marks

Marks in this subject are won in sentences, and a small number of habits win most of them.

Name the part before you state the property. The register, the bus, the layer, the gate, the stage. Say which thing you are talking about, then say what is true of it. A true fact attached to the wrong stage earns nothing at all, and that is where careful students lose marks they had already done the thinking to win.

Define before you argue. Open by defining every technical term the question names, in one clean sentence each. The definition earns its own mark and it fixes the meaning you will use for the rest of the answer.

Distinguish by giving the difference, not two descriptions. Asked to distinguish a MAC address from an IP address, an answer that says only physical against logical has given one ground out of four. Set the two side by side and say what differs: the length, the scope, which one changes when the machine moves, and which layer reads it.

Say which layer, which stage, which level. Before you say what a device or a signal does, say where it operates. A router forwards on a layer 3 destination address; a hub works at layer 1 and holds no table in which an address could be stored. The location is half the answer.

Show the working, not only the answer. Write the formula, then the substitution, then the result with its unit. Stay in bytes until the arithmetic ends and convert once, by 1024, at the end. Dividing by 1000 in the middle is the commonest way to lose a capacity question.

Bracket when in doubt. Brackets cost nothing and end the argument. Far more marks are lost to an assumed precedence than to an unnecessary pair of brackets, in a Boolean expression and in a line of C++ alike.

Give both sides wherever judgement is asked for. Argue the case, then the case against, then decide and say why. An answer that reaches a condition scores where an answer that reaches a slogan does not, and an answer that argues only one side has answered a different question.

Scale your answer to the marks. A definition worth two marks wants two sentences, not a page. A question worth twenty wants a plan, an introduction that defines the terms, one developed point to a paragraph, and a conclusion that answers the question actually asked.

The Words That Cost Marks

This subject borrows ordinary words and gives them harder meanings, and more marks are lost here than anywhere else. Learn the right-hand column to the letter.

Word	In everyday speech	In this subject
Architecture	How something is built	What the machine looks like to the programmer: the promise it makes. How the hardware delivers it is organisation
Latency	Delay, slowness	The time one instruction takes from end to end. How many finish per second is throughput
Bandwidth	Speed of a connection	A capacity, fixed and advertised. What a link actually delivered is throughput
Hit rate	How often it works	The fraction of accesses the cache answers, between 0 and 1. The time a hit takes is the hit time
Miss penalty	The cost of a miss	The extra time beyond the hit time, not the whole cost of the miss
Block	A lump of something	The unit of transfer, a fixed run of bytes. The slot that holds it is a cache line
Protocol	A way of doing things	The rules a layer uses to converse with the same layer on the other host. What it offers upward is a service
Port	A socket on the case	A 16-bit number identifying a process on a machine, 0 to 65535
Packet	Anything sent	The network layer unit. With a data link header outermost it is a frame; the outermost header names the unit
Declaration	Saying what something is	Introducing a name and its type. Reserving the storage is a definition; giving the first value is initialisation

How to Draw a Circuit or a Map for Marks

A drawing is a second answer, marked on its own. Give every one of them the same furniture, in the same order, and the marks follow.

Run the inputs down the left as labelled rails. Then work left to right in columns: the gates that wait on nothing in the first column, the gates that wait on them in the next.

Label every intermediate wire. Name the output of each gate as you draw it, and write beside it the expression it carries. A marker who can follow the drawing never guesses your intention.

Let the brackets choose the gates. One gate for every bracket, working inside out, NOT before AND before OR. Every sum of products becomes a rank of AND gates feeding one OR gate, with inverters at the far left wherever a variable appears complemented.

State any assumption beside the count. A three input AND becomes two gates where only two input parts are available. Write the assumption down and either answer stands.

On a Karnaugh map, take the largest groups there are. Stopping at small loops gives an answer that is correct and needlessly expensive, and that is the commonest way marks are lost there.

Practise the physical act of drawing. Gates take far longer to draw than symbols take to write, and a five gate circuit with every wire labelled is several minutes gone. Candidates who have not practised it lose marks to the clock, not to the logic.

How to Handle the Numbers

The arithmetic in this subject is small and it is marked strictly. Six habits protect it.

State the formula before the figures. Average memory access time as hit time plus miss rate times penalty; speed-up as one time over another; capacity as locations times width. A stated formula earns its own mark and steadies everything that follows.

Substitute, then show every line. Put the figures in and work down the page. The examiner marks the method even when the final figure slips, but only if the method is on the paper.

Watch the form of the fraction. A miss rate of 5 per cent enters as 0.05, not as 5. Hit rate and miss rate sum to 1. Enter the wrong one and you have solved the wrong case perfectly.

Carry the unit, and convert on purpose. Nanoseconds, cycles, bytes, addresses per second. Most wrong answers are right numbers wearing the wrong unit.

Write the subtraction down. Counting usable host addresses, the subtraction of the two reserved addresses is forgotten far more often than the power is miscalculated. Put the count on one line and the subtraction beneath it, so both appear in the working.

Interpret the number in a sentence. Eight nanoseconds is not an answer until you have said that no single access takes it. The interpretation usually carries more marks than the arithmetic that produced it.

How to Fail Productively

You will get questions wrong. Every strong candidate does. What decides your grade is the five minutes afterwards.

The unproductive response is to rub out your attempt, read the model, nod, and move on. Your page looks tidy and your hand has learned nothing, because you have only watched someone else think.

The productive response is to stop, and not to erase. Set your answer beside the model and find the exact line where the two part company. Was it the definition, because you used the everyday word? The component, because the fact was true of something else? The arithmetic, because a percentage went in whole? The drawing, because a wire was never named? Or the command word, because you explained where you were asked to judge? Five faults need five different cures, and naming yours is half the cure. A wrong answer you have diagnosed is worth more than a right one you copied.

How to Revise on Your Own

Most revision feels productive and is not. Re-reading model answers and copying them out neatly let your eyes move while your memory rests. These methods work instead.

Rebuild every circuit from a blank page. Not traced and not copied, but drawn from the expression alone, with every rail labelled and every intermediate wire named, until your hand does it without prompting.

Work the arithmetic families until they are automatic. Average memory access time, cycles per instruction, pipeline speed-up, memory capacity, subnet ranges. Five of each, from figures you have made up yourself, marked against your own working.

Trace rather than guess. Take a short program and follow it by hand, line by line, writing down what each variable holds after each statement. Where you can predict but cannot trace, you have not yet understood it.

Learn the opening sentence of each model answer. It is written to be learned: one sentence, the whole definition, nothing wasted. Learn ten a week and the definition marks stop escaping.

Test yourself on the words that cost marks. Cover the right-hand column of the table above and define each term in full until the technical meaning arrives before the everyday one.

Revise your weakest chapter first, not your favourite. The topic you enjoy is the one you already own; the one you avoid is the one the paper will find.

How to Face the Examination

The paper does not test how clever you are. It tests whether you can define, explain, calculate, draw and judge, by the rules and against the clock. Those are skills, and skills can be practised.

Read every question twice before you write. The second reading catches the command word, the number of points asked for, and the instruction to draw that the first reading skimmed.

Obey the command word. Each asks for a particular kind of answer, and giving the wrong kind loses marks even when the computer science is sound. The table below sets out what each demands.

Plan every long answer before you write it. Three minutes with a pencil, listing the points and the drawing in order, buys more marks than three minutes of extra writing at the end.

Spend your time by the marks. A question worth twenty deserves roughly twice the minutes of one worth ten, whatever your instinct says while you are enjoying a topic.

Do the arithmetic questions first if you can choose. They are the fastest marks on the paper and the least dependent on how well the words are coming that morning.

Leave nothing blank. A definition, a labelled drawing, even one correct sentence earns marks that an empty space cannot. Reach every question you are required to answer.

Command word	**What it asks for**
Define; State	Give the exact meaning in one sentence. No explanation is wanted and none is paid for.
Explain; Account for	Give the reason and the mechanism behind it. Every such answer needs a because.
Describe; Outline	Give an orderly account of what happens, without having to justify it.
Distinguish; Compare	Set the two side by side and give the differences directly. Two separate descriptions do not count.
Calculate; Determine; Compute	Work the figure in full, formula first and every line shown, and state the answer with its unit.
Draw; Sketch; Construct	Produce the labelled drawing itself. It carries its own marks and cannot be replaced by prose.
Trace	Follow the process step by step and record the state after each step.
Convert; Derive; Simplify	Show the transformation line by line. The steps are the answer.
Evaluate; Assess; To what extent	Weigh both sides and commit to a judgement, saying how far the claim holds and why.
Write a program	Produce code that would compile and run, not a description of what it would do.

You now carry what the strongest candidates bring into the hall and most never receive: the discipline to name the part before you state the property, the habit of reaching for the number whenever a number is there, a drawing that earns its own marks, an honest way of being wrong, and a clear plan for the day itself. The machine in this book is described from its buses upward, and the first question is waiting a page away. Sharpen your pencil, cover the first model answer, and begin.

Chapter One

COMPUTER ARCHITECTURE

Everything else in this subject stands on this chapter. The programs you will write in C++, Java and Python are instructions for the machine described here; the databases you will design are stored in the memory described here; the networks you will build move data along buses described here. A candidate who is vague about computer architecture is vague about everything downstream of it, and it shows in every paper.

The chapter begins with the shape of the machine itself, the arrangement of processor, memory and buses that has survived essentially unchanged since the 1940s, and with the single narrow crossing inside it that has driven almost every clever idea in computing since. It then goes down a level, to the algebra of true and false on which every circuit is built, and works upward again through gates, truth tables, Karnaugh maps and the circuits that add, choose, decode and remember. From there it turns to memory, where the important skill is arithmetic rather than description, and finishes with the instruction set the programmer actually sees and the three ways a machine talks to the world outside itself.

Two habits will serve you throughout. The first is to name the part before you state the property, because most of the marks lost in this topic are lost by attaching a true fact to the wrong component. The second is to reach for a number wherever one is available. Questions in this area are frequently answered in words when they could have been answered in arithmetic, and an examiner rewards the candidate who calculates.

Architecture versus organisation, and the von Neumann machine

This section settles what a computer's architecture actually is and separates it from the organisation that implements it, draws the machine on which almost every computer you will ever meet is built, names the three buses along which everything inside it travels, and shows you the single narrow crossing that has shaped processor design for eighty years. Get the first distinction right and the rest of the chapter falls into place behind it.

BINDER Question 1

Two shops on Samora Avenue sell you the same phone at the same price, and one of them runs visibly faster than the other. The salesman calls it a better architecture. He has reached for the wrong word.

Define computer architecture and computer organisation, and distinguish clearly between them.

Model Answer

Computer architecture is what a machine looks like to the programmer, while computer organisation is how the hardware is built to deliver it.

Architecture: the attributes a programmer writing machine-level code can see and must obey. The instruction set, the number and names of the registers, the addressing modes, the word length, the way numbers and characters are represented, and the mechanism by which input and output happen. If changing something would force you to rewrite a machine-code program, that something is architecture.

Organisation: the operational units and their interconnection that realise the architecture. Whether a multiplication is performed by dedicated hardware or by repeated addition, how wide the internal data path is, how much cache sits beside the processor, how fast the clock ticks. Change any of these and the same program still runs, only faster or slower.

The test that settles it: ask whether a machine-code program written for the one machine would run unchanged on the other. If the answer is yes, you are looking at two organisations of one architecture. If the answer is no, the architecture itself has changed.

Why the distinction is worth money: a manufacturer can sell a whole family of machines at different prices, all sharing a single architecture so that software bought for the cheapest runs untouched on the most expensive, while the organisation inside each one is tuned to what its buyer can afford.

The two phones on Samora Avenue almost certainly share an architecture and differ only in organisation, which is exactly why the same applications install and run on both.

The Master Key: *Architecture is the promise the machine makes to the programmer; organisation is how it keeps it.*

BINDER Question 2

You save a letter, then open a photograph, and never tell the machine which is a program and which is data. It keeps both in one store, in one binary form. That single idea is why your desk holds a computer, not a calculator.

Describe the von Neumann architecture and explain the stored-program concept on which it rests.

Model Answer

The von Neumann architecture is a design in which one memory holds both instructions and data, and a single processing unit fetches and executes those instructions one at a time across a shared set of buses.

The stored-program concept: instructions are held in memory in exactly the same binary form as data, so the machine is given a new job by loading new instructions rather than by being rewired. This is what makes one machine general-purpose.

The central processing unit: holds the arithmetic and logic unit, which performs the calculations and comparisons; the control unit, which interprets each instruction and issues the signals that carry it out; and a small set of registers holding the values the processor is working on at this instant.

Main memory: a single addressable store holding the program and its data together, every location carrying a unique address by which the processor reaches it.

Input and output: the channels through which the machine receives data from the world outside and returns its results to it.

The buses: the address, data and control lines that join processor, memory and the outside world, and along which every transfer in the machine travels.

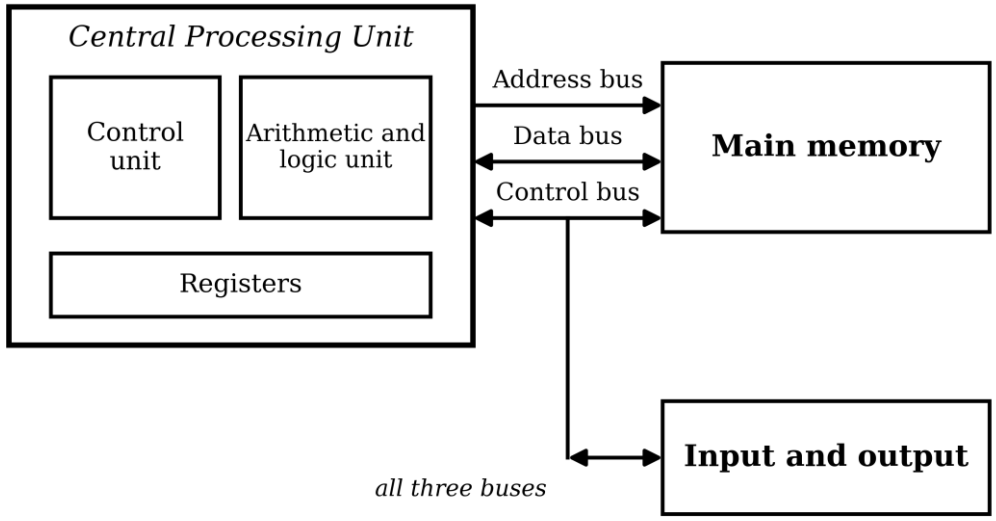


Figure 1.1: *The von Neumann architecture, showing the three system buses.*

Because instructions and data share one memory and one road to the processor, the machine is completely general in what it can be asked to do, and that same sharing is precisely what limits how fast it can do it.

The Master Key: *Store the program beside the data and the machine stops being a device that does one job and becomes a device that does any job you can write down.*

BINDER Question 3

Nothing leaves Ubungo until three things have happened. Someone says where the bus is going, the passengers board it, and a controller signals that it may pull out. Every byte a computer moves travels on three sets of wires doing exactly those three jobs.

Describe the three system buses, stating what each one carries and in which direction it operates.

Model Answer

The address bus says where, the data bus carries what, and the control bus decides when.

The address bus: carries the address of the memory location or device the processor wishes to reach. It is unidirectional, always outward from the processor, because it is always the processor that decides where a transfer is going. Its width fixes how many distinct locations the machine can name.

The data bus: carries the value itself, and is bidirectional, because the processor both reads values in and writes values out along the same lines. Its width fixes how many bits move in a single transfer, and therefore how much work each transfer accomplishes.

The control bus: carries the timing and command signals that say what is to happen and when. Read, write, the clock, interrupt request, bus grant. Its individual lines run in different directions, some out of the processor and some into it.

Why the widths matter: an address bus of n lines produces two to the power n distinct combinations, and each combination names one location, so widening it multiplies the memory the machine can use. Widening the data bus instead raises how much moves per transfer, which increases speed without changing the meaning of a single program.

$$\text{Number of addressable locations} = 2^{n}$$

Bus	What it carries	Direction	What its width fixes
Address	the location or device wanted	out of the processor only	how many locations can be named
Data	the value being transferred	both ways	how many bits move per transfer
Control	read, write, clock, interrupt, grant	line by line, both ways	how many distinct signals are available

Table 1.1: *The three system buses.*

Keep the three separate in your mind and most of the confusion around memory capacity, transfer rate and timing disappears at once.

The Master Key: *Widen the address bus and the machine can reach further; widen the data bus and it merely gets there faster.*

BINDER Question 4

Before the bridge to Kigamboni opened, everything crossing the creek queued for one ferry. It made no difference how many cars Dar es Salaam owned or how powerful their engines were, because the crossing set the pace. Every computer has a ferry like it.

Explain what is meant by the von Neumann bottleneck and state its consequence for processor performance.

Model Answer

The von Neumann bottleneck is the limit imposed by the single shared path between processor and memory, along which only one instruction or one item of data can travel at a time.

Where it comes from: instructions and data live in the same memory and reach the processor along the same buses, so a fetch of the next instruction and a fetch of an operand cannot happen together. They queue.

Why it grows worse rather than better: processor speed has risen far faster than memory speed for decades, so the distance between how fast the processor can work and how fast memory can feed it has widened with every generation rather than closing.

The visible consequence: the processor spends cycles idle, waiting for something to arrive. Beyond a certain point, raising the clock speed alone buys no extra performance at all, because the processor is not the thing that is slow.

How designers answer it: cache memory placed close to the processor, wider buses, separate paths for instructions and data, and prefetching, which guesses what will be wanted next and starts fetching it early. Every one of these exists to reduce how often the processor must wait on the single road.

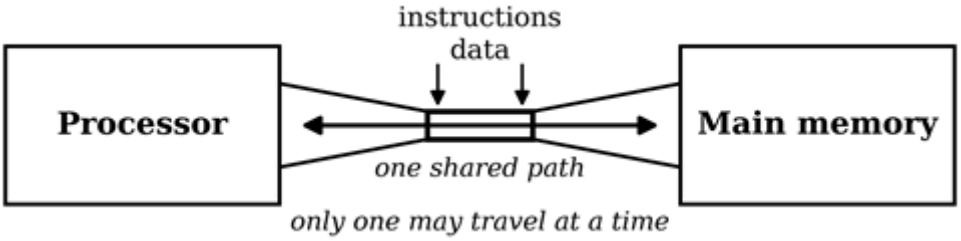


Figure 1.2: *The von Neumann bottleneck. Instructions and data share one path.*

Notice that none of those remedies removes the bottleneck. They widen it, hide it, or work around it, which is why it remains the central problem of computer design rather than a solved historical curiosity.

The Master Key: *A processor is only ever as fast as the road that feeds it.*

REAL Question 5

A secondary school in Njombe is offered refurbished machines at a price it can afford. The supplier's specification sheet is one page long, and the headteacher wants one thing before signing: will these machines run the science software the school already owns?

Calculate the maximum memory the machine can address, and advise the school whether the software will run.

Address bus: 24 lines

Data bus: 16 lines

Each memory location stores one byte

The science software requires 32 MB of memory

Model Answer

The machine can address only 16 MB, which is half of what the software requires, so the school should not buy these machines for this purpose.

Reading the address bus: each line carries one bit, so 24 lines produce two to the power 24 distinct combinations, and each combination names exactly one memory location.

The calculation:

$$\text{Addressable locations} = 2^{24} = 16\,777\,216$$

Turning locations into capacity: because each location stores one byte, 16,777,216 locations hold 16,777,216 bytes, and since one megabyte is 1,048,576 bytes, that is exactly 16 MB.

Why the one-byte assumption matters: it is the specification sheet, not the address bus, that tells you a location holds a single byte, and that is what turns a count of locations into a count of bytes. A machine storing one 16-bit word at every address would reach the same 16,777,216 locations and hold 32 MB. Never convert locations into capacity until you know what sits at each one.

What the data bus does and does not tell you: the 16-bit data bus fixes how much moves per transfer, which is two bytes at a time, so it governs speed and not capacity. Multiplying the two figures together is a common mistake and here it would produce 32 MB, the very answer the headteacher hopes for and the reason the mistake is so dangerous.

The advice: the software will not run, and no amount of extra disk or a faster processor will change that, because the limit is what the processor can name rather than what the school can attach to it. The school should either ask the supplier for machines with a wider address bus, or spend the same money on fewer newer machines rather than on more old ones.

What would have been enough: a single extra address line would have done it. Twenty-five lines reach two to the power 25 locations, which is 32 MB, because every additional line doubles the address space.

This is the arithmetic behind every specification sheet you will ever read, and it takes about thirty seconds once you know which bus to look at.

The Master Key: *Capacity comes from the address bus and speed from the data bus, and confusing the two can cost a school a laboratory.*

REAL Question 6

A technician in Mbeya leaves a handwritten note for the student who will take over the computer laboratory next term. He means well, and he is confident, and three of his sentences are wrong.

Identify the three errors in the technician's note and correct each one.

1. The von Neumann machine keeps programs in one memory and data in a separate memory.

2. The address bus is bidirectional, because the processor both reads and writes.

3. Doubling the width of the data bus doubles the amount of memory the machine can address.

Model Answer

All three statements are wrong, and each one makes the same underlying mistake of borrowing a property from one part of the machine and attaching it to another.

The first error, on the memories: separate memories for instructions and data describe the Harvard design, not the von Neumann one. The defining feature of von Neumann is precisely that a single memory holds both, which is what the stored-program concept requires.

The second error, on direction: the reasoning is the right reasoning applied to the wrong bus. It is true that the processor both reads and writes, and that is why the data bus is bidirectional. The address, however, travels outward in both cases, because the processor must say where it is reading from just as it must say where it is writing to. The address bus is unidirectional.

The third error, on capacity: doubling the data bus doubles how many bits cross per transfer, not how many locations can be named. Capacity is set by the address bus alone, as two to the power n. A machine whose data bus grows from 16 lines to 32 moves twice as much per transfer and reaches exactly the same memory as before. Put numbers to it: with 20 address lines the machine names 1,048,576 locations whether its data bus is 8 lines wide or 64.

What the three have in common: each one swaps an attribute of one bus or one design for another. Before answering any question of this kind, name the bus or the architecture the question is actually about, and only then state the property. Naming the part first costs a second and prevents the entire family of mistakes this note illustrates.

The note is worth keeping and correcting rather than throwing away, because a wrong statement you have explained is remembered far longer than a right one you have merely read.

The Master Key: *Most bus errors are swaps, so identify which line the question is asking about before you say anything about it.*

HOT Question 7

Every few years someone announces that the bottleneck is finally dead, and every few years a new class of workload arrives to prove otherwise. The interesting question is not whether the claim is true, but for whom it is true.

"The von Neumann bottleneck is no longer the main limit on computer performance." To what extent do you agree?

Model Answer

The bottleneck has been widened and hidden rather than removed, so the claim holds for ordinary predictable workloads and fails badly for data-intensive ones.

The case for agreeing, on cache: the great majority of memory accesses in ordinary work never reach main memory at all. They are satisfied by cache sitting beside the processor, so the shared road is used far less often than the plain model suggests.

The case for agreeing, on the road no longer being single: modern processors keep separate instruction and data caches, run several memory channels at once, and use buses far wider than anything von Neumann described. The path has been multiplied even though the principle has not changed.

The case for agreeing, on other limits now binding first: heat and power now cap clock speeds well before memory bandwidth does, and for many machines the true limit is what can be cooled rather than what can be fetched.

The case against, on the widening gap: processor speed has grown faster than memory latency for decades. The underlying disparity the bottleneck describes is larger today than when the term was coined, not smaller.

The case against, on where caches fail: caching works only where access is predictable. Workloads that stream through large volumes of data once, which is exactly what data analysis and machine learning do, break locality by their nature, and for them almost every access pays the full cost.

The case against, on the evidence of the remedies themselves: cache hierarchies, prefetching, wide buses and separate instruction paths are expensive and complicated, and they exist for one reason. A problem that has been solved does not require a permanent industry of workarounds.

The judgement: the claim is defensible for the ordinary desktop and indefensible as a general statement. The bottleneck is not the main limit for predictable work, and it is very much the main limit for the workloads growing fastest.

That distinction, rather than a simple yes or no, is what a question of this kind is testing.

The Master Key: *The bottleneck was never removed, only buried under cache, and it surfaces the moment a workload stops being predictable.*

HOT Question 8

A manufacturer takes a full page in a technology magazine to announce a completely new processor architecture. Further down the same page, in smaller type, it reassures buyers that all their existing software will run without modification. Those two sentences cannot both stand.

A manufacturer advertises a new processor as having a completely new architecture, yet existing programs run on it unchanged and simply run faster. Assess the accuracy of the claim.

Model Answer

The claim is almost certainly false as stated, because if existing programs run unchanged then what has been replaced is the organisation and not the architecture.

The decisive test: architecture is what the programmer can see. Software compatibility is therefore not a side benefit of an unchanged architecture, it is the definition of one. A genuinely new architecture would break old programs, and the advertisement itself promises that nothing breaks.

What has almost certainly changed instead: more or faster cache, a higher clock, better branch prediction, more execution units working in parallel, a smaller manufacturing process, wider internal paths. Every one of these is organisation, and every one makes the same instructions execute faster.

Why the word is used anyway: architecture sounds fundamental and organisation sounds like housekeeping, so the marketing value of the wrong word is considerable and the cost of using it is nothing.

The case for some sympathy: the industry does use microarchitecture for precisely this kind of change, and in casual speech that word is often shortened to architecture. The manufacturer may be using a loose trade convention rather than making a false claim.

Where the claim could be honestly true: if new instructions have been added, the architecture genuinely has been extended, since programs written to use them will not run on the older machine. Note the direction, though. Old software still runs on the new processor, but new software will not run on the old one, and it is that one-way failure that reveals a real architectural change.

The judgement: on the evidence given the claim is inaccurate, and inaccurate in a direction that flatters the seller. It would become accurate only if the manufacturer could point to instructions the previous machine was unable to execute, and the page offers no such evidence. Notice that a reader who knows the compatibility test can settle the question from the advertisement alone, without a single technical detail about the processor itself.

Read every specification this way and you will find the distinction from the first question of this section doing useful work for the rest of your life.

The Master Key: *If your old programs still run, the architecture did not change, and only the engineering underneath it did.*

Harvard, modified Harvard, and Flynn's taxonomy

Section one left you with one memory, one set of buses and a queue. This section shows you the two ways designers escaped it. The first splits the memory in two and pays handsomely for the privilege; the second splits

only the cache and keeps everything that matters. Then you meet Flynn, whose two questions about streams sort every parallel machine ever built into four boxes, and you learn to place a real machine in the right one.

BINDER Question 9

At the market in Arusha, produce comes in through one gate and buyers walk in through another. Force both streams through one doorway and the market still works, only slower. A computer can be built either way, and the choice is still argued over.

Describe the Harvard architecture and contrast it with the von Neumann machine.

Model Answer

The Harvard architecture keeps instructions and data in two separate memories reached by two separate sets of buses, so the processor can fetch an instruction and read or write a data word in the very same cycle.

The two memories: one store holds nothing but the program and the other nothing but the values it works on. Each carries its own address and data lines, so a transfer to one never waits for the other.

What the separation buys: genuine parallel access. On a von Neumann machine the next instruction fetch and an operand fetch queue behind one another on one shared road. Here they travel at the same instant on different roads, so the machine finishes sooner without its clock ticking any faster.

What the separation costs: two memory systems, two sets of buses, more pins and more copper, all of which raise the price. The division is fixed at manufacture, so space left unused in the instruction store cannot be lent to data.

The loss that really matters: a strict Harvard machine cannot easily treat a program as data. Instruction memory is closed to the ordinary data path, so there is no comfortable way to load a fresh program, and compilers, loaders and self-modifying code become awkward or impossible. That one limitation confines the design to fixed-purpose machines.

Where you actually meet it: in microcontrollers and digital signal processors, embedded in water pumps, electricity meters and engine management units. Their program is burned in once and their timing must be predictable, which is exactly what Harvard gives.

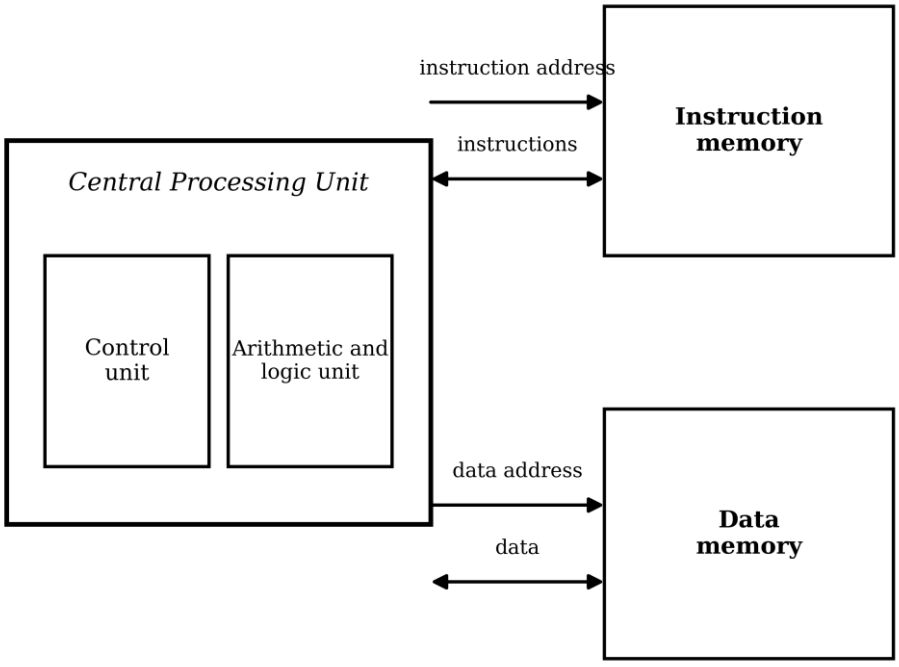


Figure 1.3: *The Harvard architecture. Instructions and data have separate memories and separate buses.*

Feature	Von Neumann	Harvard
Memories	one memory holding instructions and data together	two memories, one for instructions and one for data

Feature	Von Neumann	Harvard
Buses	one shared set of address, data and control lines	two independent sets, one serving each memory
Simultaneous access	impossible; a fetch and a data access must queue	routine; a fetch and a data access share one cycle
Cost and complexity	lower; less wiring, fewer pins, one memory system	higher; buses, pins and memory hardware are duplicated
Typical use	general-purpose computers	microcontrollers and digital signal processors
Code treated as data	natural; a program is loaded, written and altered as data	difficult; instruction memory is closed to the data path

Table 1.2: *Von Neumann and Harvard compared.*

The two designs are not rivals so much as two answers to one question about how much you will pay for parallel access, and how much freedom you will surrender to get it.

The Master Key: *Harvard buys speed with separate roads and pays for it with a program it can no longer easily change.*

BINDER Question 10

The road into Dodoma widens into two carriageways for the last stretch before the bus stand, then narrows again beyond it. The widening lies exactly where the queueing was, which is why the money went there. Almost every modern processor makes the same decision.

Explain the modified Harvard design and why nearly every modern processor uses it.

Model Answer

The modified Harvard design splits only the cache, into a separate instruction cache and data cache over one unified main memory, so the processor gains Harvard's parallel access without losing von Neumann's freedom to treat a program as data.

Where the split actually sits: not in main memory but in the small fast store beside the processor. The first level of cache is divided in two, an instruction cache feeding the fetch unit and a data cache serving loads and stores.

What the split buys: in one cycle the fetch unit pulls the next instruction from the instruction cache while the execution unit reads an operand from the data cache. Because most accesses never reach main memory, the machine behaves like a Harvard machine for almost all of its life.

What the unified memory preserves: one address space in which instructions and data are indistinguishable bytes. A program can be read from disk as data, written into memory as data, then jumped to and executed. Compilers, loaders and operating systems all depend on that.

Why the compromise is not a fudge: the bottleneck was never spread evenly. It bites hardest closest to the processor, where a fetch and an operand access collide every cycle. Splitting there removes the collision; splitting main memory too would cost more and gain almost nothing.

The price that is paid: coherence. One location can now sit in both caches, so a newly written instruction may live in the data cache while a stale copy sits in the instruction cache. One must be flushed before the new code is run.

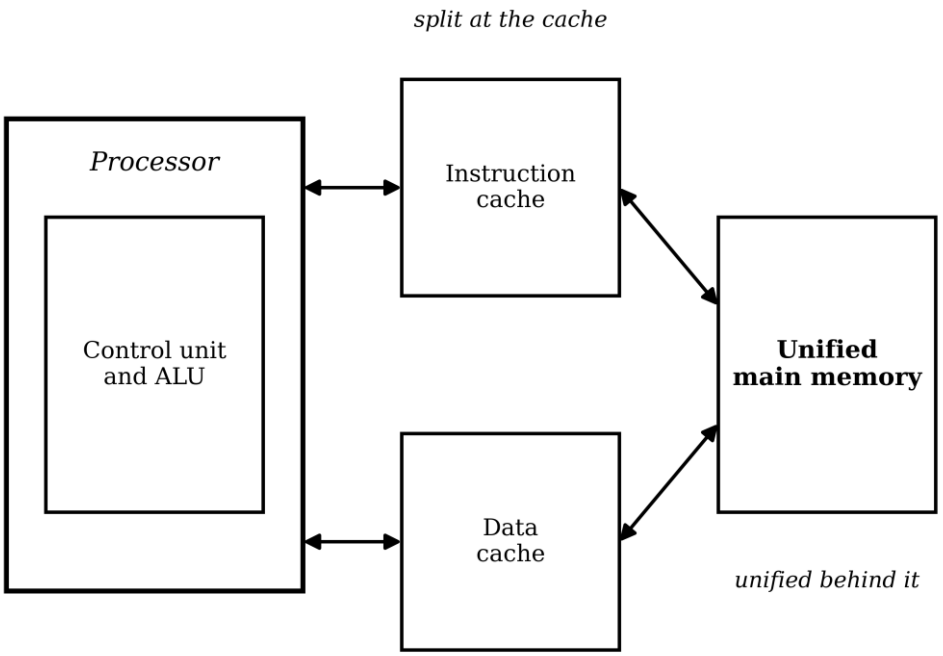


Figure 1.4: *The modified Harvard design. Split caches over a single unified memory.*

So your laptop is neither purely Harvard nor purely von Neumann. It is deliberately both, split where speed is won and unified where freedom is needed.

The Master Key: *Split the cache and you take Harvard's speed; keep one main memory and you keep the right to write your own programs.*

BINDER Question 11

Stand at a boatyard on the Mwanza shore at first light. One foreman may shout one order to twenty men, or each man may work to his own list. In 1966 Michael Flynn found that every parallel computer is one of four arrangements.

State the basis of Flynn's classification and describe its four classes.

Model Answer

Flynn classifies a machine by two things only, the number of instruction streams flowing through it at once and the number of data streams those instructions act upon, and crossing the two gives exactly four classes.

The basis of the classification: an instruction stream is a sequence of instructions being executed, and a data stream is a sequence of data items being operated on. Each may be single or multiple, and two choices with two values each produce four classes and no more. Notice what is deliberately ignored: clock speed, memory layout, how the parts are wired together. That severe abstraction is exactly why the scheme has lasted.

Single instruction, single data: one instruction stream working on one data stream, which is the classic sequential machine of the previous section. One instruction is fetched, decoded and executed on one set of operands before the next begins.

Single instruction, multiple data: one instruction stream applied simultaneously to many data items held in many processing elements. The control unit issues one instruction and every element performs it in the same cycle on its own value, marching in lockstep.

Multiple instruction, single data: several instruction streams operating on one data stream. This is the rare class. The standard example is a fault-tolerant system in which several processors run different algorithms over the same input and compare their answers, so that any disagreement announces a fault.

Multiple instruction, multiple data: many instruction streams on many data streams, each processor fetching its own instructions and working on its own data. Multi-core processors, server clusters and every ordinary computer you can buy today sit here.

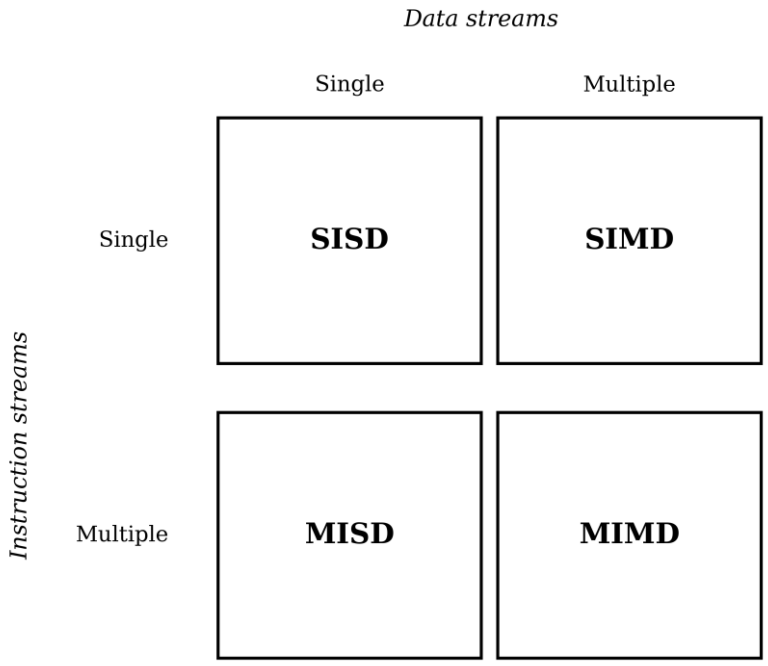


Figure 1.5: *Flynn's taxonomy, classified by instruction streams and data streams.*

Class	Instruction streams	Data streams	A typical example
SISD	One	One	A simple single-core controller taking one instruction at a time
SIMD	One	Many	A graphics processor or vector unit applying one operation to many pixels
MISD	Many	One	Rare; fault-tolerant systems running different algorithms on the same input and comparing results
MIMD	Many	Many	A multi-core processor, or a cluster of servers

Table 1.3: *Flynn's four classes.*

Two questions settle any machine put in front of you. How many different instruction streams are in flight, and how many different collections of data are they acting on.

The Master Key: *Count the streams and not the processors, and Flynn answers himself.*

BINDER Question 12

A photographer back from a dhow race off Zanzibar has four hundred frames to darken by the same amount before evening. A district hospital in Morogoro has forty clerks, each on a different patient. Both are parallel; only one obeys a single shouted order.

Explain SIMD and MIMD, and give a worked situation for each.

Model Answer

SIMD applies one instruction to many data items at the same instant, while MIMD lets independent processors execute entirely different instructions on entirely different data.

What SIMD actually does: one control unit decodes a single instruction and broadcasts it to many processing elements, each holding a data item of its own. Every element performs that operation in the same cycle, so there is one program counter in the whole arrangement and only the operands differ.

The worked SIMD situation: the photographer's four hundred frames. Every pixel needs the same constant subtracted and nothing else. A vector unit loads sixteen pixel values into one register and issues a single subtract acting on all sixteen at once, so the arithmetic costs a sixteenth of the instructions fetched and decoded.

Where SIMD collapses: the moment the items stop needing identical treatment. If some frames must be brightened and others darkened, the lockstep breaks. The hardware runs one branch with the wrong elements masked off, then the other with the rest masked off, so the work is done twice. SIMD rewards uniformity and punishes decisions.

What MIMD actually does: each processor carries its own control unit and its own program counter, fetches its own instructions and operates on its own data. The processors may cooperate and exchange results, but they are never synchronised instruction by instruction.

The worked MIMD situation: the hospital's forty clerks. One record needs a laboratory result attached, the next a discharge letter, the third a bill. Give each core a different patient and a different procedure and all forty advance at once. No single instruction would serve them all, which is why SIMD hardware would stand idle here.

Ask one question of any parallel workload. Is every item having the same thing done to it? Yes points to SIMD, and no points to MIMD.

The Master Key: *SIMD is one order given to many hands; MIMD is many hands each reading a list of their own.*

REAL Question 13

The controller in a maize mill at Kilombero closes its loop a thousand times a minute. The engineer is offered two versions at the same price, one with a shared bus, one with separate buses. She wants the difference as a number, not an adjective.

Calculate the bus cycles required per control pass on each architecture and the resulting speed-up, then advise the engineer which version to buy.

Instructions executed per control pass: 12,000

Instructions that also read or write one data word: 45 per cent

Shared bus: an instruction fetch and a data access each take one bus cycle and cannot overlap

Separate buses: a data access overlaps the next instruction fetch and so costs no extra cycle

Both versions run at the same clock speed

Model Answer

The shared-bus controller needs 17,400 bus cycles per pass and the separate-bus controller needs only 12,000, a speed-up of 1.45, so the engineer should buy the separate-bus version provided it costs less than about forty-five per cent more.

Counting the data accesses first: forty-five per cent of the 12,000 instructions in a pass also touch memory for a data word, which is 5,400 data accesses. The other 6,600 are fetched and executed without going back to memory, so they cost one cycle each.

The von Neumann total: every instruction must still be fetched, and on a shared bus a data access cannot overlap a fetch, so the two simply queue. Add the 12,000 fetch cycles to the 5,400 data cycles.

$$\text{Von Neumann bus cycles per pass} = 12\,000 + 5\,400 = 17\,400$$

The Harvard total: with separate buses the data access belonging to one instruction happens on the data bus while the instruction bus is already fetching the next instruction. All 5,400 data accesses hide completely inside the stream of fetches, so the pass costs only its 12,000 fetch cycles.

The speed-up: divide the larger figure by the smaller. 17,400 divided by 12,000 is 1.45, so on an identical clock the separate-bus controller completes forty-five per cent more control passes every second.

Why the forty-five per cent figure drives the whole answer: the entire gain is exactly the data traffic that no longer queues, so the speed-up is one plus the fraction of instructions that touch data. At twenty per cent it would be 1.2, at eighty per cent 1.8, and at zero the two machines would tie. Find that fraction first, because nothing else in the question can move the result.

The advice to the engineer: buy the separate-bus version. Forty-five per cent more passes per second on the same clock is a real margin in a mill where the gate must respond before the motor stalls, and a mill controller runs fixed firmware, so the flexibility Harvard gives up is flexibility this application never wanted.

What the number does not cover: cycles are not seconds. The comparison is honest only because the two versions are clocked alike; a Harvard part with slower memory could hand the whole advantage back. Compare cycles when the clock is equal and time when it is not.

The word faster is worth nothing to an engineer holding two quotations. A ratio of 1.45 is worth a decision, and four lines of arithmetic produced it.

The Master Key: *The Harvard gain is just the share of instructions that touch data, so find that fraction and the arithmetic is already finished.*

REAL Question 14

Three sentences go up on the board in a Tanga classroom, offered with complete confidence by a student who has plainly been reading. Each of the three reaches a conclusion that sounds right, and each reaches it for a reason that is wrong.

Identify each of the three errors in the note and correct it.

1. "The Harvard architecture is faster because it has a faster clock."

2. "A modified Harvard machine has two separate main memories."

3. "A multi-core laptop is an example of SIMD because it does many things at once."

Model Answer

All three statements are wrong, and all three fail in the same manner, by attaching a genuine advantage to the wrong cause.

The first error, on where the speed comes from: Harvard says nothing about clock rate. Two machines clocked identically will not finish together, because the Harvard one performs an instruction fetch and a data access in one cycle instead of two. Corrected: the Harvard architecture is faster because separate buses let a data access overlap the next instruction fetch.

Why that first error is so tempting: speed is learned as one quantity, when the time a program takes is instructions multiplied by cycles each and by cycle length. Harvard attacks the middle term; a faster clock attacks the last.

The second error, on what is actually split: the modified Harvard design splits the cache, not main memory. One unified main memory still holds instructions and data in a single address space, with two first-level caches above it. Corrected: a modified Harvard machine has separate instruction and data caches over one unified main memory.

Why the unified memory has to stay: if main memory really were divided, a program could not be loaded as ordinary data, and every loader, compiler and operating system would fail. The single memory is not an oversight; it is what the design exists to protect.

The third error, on Flynn: the cores each carry their own program counter and fetch their own instructions, so there are many instruction streams acting on many data streams. That is MIMD; SIMD means one instruction stream broadcast to many data items in lockstep. Corrected: a multi-core laptop is an example of MIMD, because each core runs a different instruction stream on different data.

Where SIMD really is in that laptop: inside each core sits a vector unit, and the graphics processor is SIMD throughout, so the machine contains SIMD hardware without being a SIMD machine. Flynn classifies the arrangement of streams in use, not every circuit on the board.

The habit that catches all three: state the mechanism before the effect. Separate buses cause overlap and overlap causes speed. Independent program counters cause multiple instruction streams, and multiple instruction streams mean MIMD. Each sentence on that board would have survived that chain said out loud.

A confident sentence with the right conclusion and the wrong reason earns very little, because the reason is the thing being examined.

The Master Key: *Each of these errors is a true effect married to a false cause, so name the mechanism before you claim the benefit.*

HOT Question 15

An aircraft on final approach to Kilimanjaro International is flown by several computers that are deliberately not identical. They read the same instruments and are expected to disagree, because disagreement is how a fault announces itself. One of Flynn's four boxes is called empty.

"MISD is a category with no real members." Assess this claim.

Model Answer

The claim is defensible as a statement about the marketplace and wrong as a statement about computing, because MISD has almost no products but several genuine and important members.

The case for the claim, on the absence of products: no manufacturer has ever sold a general-purpose MISD computer, and none advertises one now. Every commercial parallel machine of consequence is SIMD or MIMD, and a class with no product line behind it looks convincingly empty.

The case for the claim, on the strangeness of the arrangement: multiple instruction streams applied to one data stream is a difficult thing to want. Useful parallelism nearly always comes from dividing data among many workers rather than doing several unlike things to one item, so the class describes an arrangement few problems ask for.

The case for the claim, on how it is taught: because clean examples are awkward to give, MISD is usually introduced as the fourth box completed for symmetry, and a teaching shortcut repeated often enough hardens into a supposed fact.

The case against, on fault-tolerant systems: flight control computers and other safety-critical systems run several processors, often with independently written programs, over the same stream of sensor readings and compare the outputs. Different instruction streams, one data stream, exactly as defined. These machines are not thought experiments; they are in service and carrying passengers today.

The case against, on systolic arrays: in a systolic array a single stream of data is pumped through a chain of processing elements, each applying its own different operation as the data passes. Signal filters and matrix pipelines are built this way, placing many instruction streams against one data stream in real silicon.

The case against, on what should count as a member: the claim quietly assumes a class is empty unless somebody sells an entire computer of that kind. Apply that rule consistently and other classes begin to empty out too. Flynn was classifying arrangements of streams, not entries in a catalogue, and by his own criterion the box is occupied.

The judgement: MISD is the smallest and least commercially significant of the four, and it is fair to say no general-purpose MISD computer exists. It is not fair to call the category empty. Redundant fault-tolerant systems and systolic arrays are members in good standing, and that you meet them in aircraft rather than in shops tells you where the arrangement is useful, not whether it is real.

The honest version of the claim is that MISD is rare, and rare is a very different word from imaginary.

The Master Key: *A class with no shop window is not the same as a class with no members.*

HOT Question 16

A technician in Iringa reads separate instruction and data caches on the first page of a processor's specification and declares von Neumann retired. Further on he finds one flat address space where programs are rows of bytes. Both pages describe the same chip.

To what extent has the Harvard architecture displaced the von Neumann machine?

Model Answer

Harvard has displaced von Neumann in the hardware nearest the processor and almost nowhere else, because the single address space and the stored-program principle the programmer works with remain entirely intact.

The case for displacement, at the cache: practically every processor made today, from the controller in an electricity meter to a server chip, keeps a separate first-level instruction cache and data cache. Since most accesses never reach main memory, the machine spends almost all its working life behaving as a Harvard machine.

The case for displacement, by sheer numbers: counted as devices rather than as laptops, computing is dominated by microcontrollers buried in vehicles, meters, pumps and appliances, and many of those are strict Harvard machines with the program in flash and the variables in a small separate memory.

The case for displacement, in the direction of travel: every serious remedy for the bottleneck moves toward separation: split caches, separate prefetch paths, multiple memory channels. Nobody has designed toward a single narrow shared road in fifty years.

The case against, on what sits behind the caches: one memory and one address space. The split is a performance device laid on top of a von Neumann store, and the moment a cache misses, both streams draw on that same unified memory. Nothing architectural has been divided.

The case against, on the programmer's model: this is the decisive point. Architecture is what the programmer can see, and the programmer still sees one space in which instructions and data are indistinguishable bytes. Compilers write code as data, loaders place it as data, and interpreters build it at run time and jump into it. On the test that defines architecture, the machine is von Neumann.

The case against, on the stored-program principle: the idea that made a computer general-purpose is untouched. Harvard changes where an instruction is fetched from, not the fact that the program is stored and can be replaced by loading another. Giving that up would mean giving up software itself.

The judgement: the displacement is real, substantial and strictly partial, and it lies entirely below the line the programmer can see. Inside the chip Harvard has won almost completely; above it, in the model everyone actually programs to, von Neumann has lost nothing. The answer is a level rather than a verdict: Harvard at the cache, von Neumann at the memory and at the model.

Which is why the accurate name for a modern processor is neither of the two, but the compromise that took the speed of one and kept the freedom of the other.

The Master Key: *Harvard won the wiring; von Neumann still owns the meaning.*

HOT Question 17

A mapping agency in Dodoma has forty thousand aerial photographs of the Rufiji delta and a deadline set by the rains. The budget stretches to one large machine, and the two suppliers left offer completely different shapes of parallelism for nearly the same money.

The agency must apply the identical haze correction and contrast stretch to every pixel of all forty thousand photographs, and then run a separate flood-risk model for each of nine districts, every district using different rules. It can buy either a cluster of many independent processors or one processor with a very wide vector unit. Compare the two in Flynn's terms and justify a choice.

Model Answer

The image stage is pure SIMD work and the district models are pure MIMD work, and because the image stage dominates the schedule the agency should buy the vector machine and run the nine models on its ordinary cores.

Naming the two stages in Flynn's terms: stage one applies an identical operation to every pixel of every photograph: one instruction stream over many data streams, which is SIMD. Stage two runs nine unlike models over nine districts: nine instruction streams over nine data sets, which is MIMD. One workload, one class of each, in sequence.

What the vector machine offers the first stage: a wide vector unit loads many pixel values into one register and issues a single instruction acting on all of them. Both corrections are the same arithmetic on every pixel with no branching, so the lockstep never breaks and the unit runs near its theoretical rate.

What the cluster offers the first stage: it can do the same work by giving each processor a share of the photographs, but every processor still fetches and decodes an instruction for each pixel it touches. That parallelism costs far more in instructions, in electricity, and in moving the images across a network and back.

What the cluster offers the second stage: here the cluster is genuinely the better machine. Nine unrelated models with different rules and heavy branching are what independent processors exist for, and a vector unit handed nine different programs has nothing to broadcast.

Why the shape of the workload settles it: the pixel stage is thousands of millions of identical operations; the district stage is nine tasks. Nine tasks do not require a cluster, because any ordinary multi-core host already supplies nine instruction streams. Choose hardware for the stage that consumes the schedule.

The arguments that sit outside Flynn: one machine means no network to tune, one system to maintain and a smaller power draw, which matters where the supply is not steady. A cluster also needs a room, cooling, and staff who can debug a distributed job at midnight.

The judgement and the condition attached to it: buy the vector machine and treat the nine district models as MIMD work for its cores. State the condition plainly, because the condition rather than the conclusion shows understanding: this holds only while the SIMD stage dominates. Should the work shift toward many unlike simulations at once, the cluster wins.

Classify each stage before you look at either price. The classification decides the machine, and the quotation only decides which supplier sells it to you.

The Master Key: *Buy for the stage that eats the schedule, because the other stage will fit on whatever you bought.*

The fetch-decode-execute cycle, the CPU registers, and RTL

Section two described the machine from the outside, by the shape of its memories and the count of its streams. This section takes the lid off. Inside sit six small stores, each built for one job, and a cycle that moves words between them in an order that is not negotiable. You will learn to name every register, to trace a running program through them step by step, and to write down what you traced in the notation designers actually use.

BINDER Question 18

In the harbour office at Bagamoyo there are six pigeonholes above the desk, and the clerk has never once put a document in the wrong one. Each hole takes one kind of thing and nothing else, which is why he never pauses to think.

Describe the special-purpose registers of the processor and state the job of each.

Model Answer

Six special-purpose registers carry the machine, each built to hold one kind of thing and nothing else: the program counter, the memory address register, the memory data register, the instruction register, the accumulator and the status register.

The program counter: holds the address of the next instruction, never the instruction itself. It is advanced during the fetch, so while an instruction is being obeyed the counter already points past it. A jump changes the machine's whole path by overwriting it.

The address register and the data register: work as a pair on every memory access. The MAR holds the address about to be read or written and alone drives the address bus. The MDR holds the word in transit, either direction, and alone sits on the data bus.

The instruction register: holds the instruction just fetched and keeps it still while the work is done. The control unit reads the opcode from here to learn what to do, and the operand field beside it to learn what to do it to. Nothing else in the machine is ever decoded.

The accumulator: holds an operand going into the arithmetic and logic unit and receives the result coming back. Nearly every intermediate value on a simple machine passes through it.

The status register: holds single bits rather than a value: zero, negative, carry and overflow, set by the arithmetic and logic unit as a side effect of each operation. Nothing reads them until a conditional branch does, and that reading is how a machine decides.

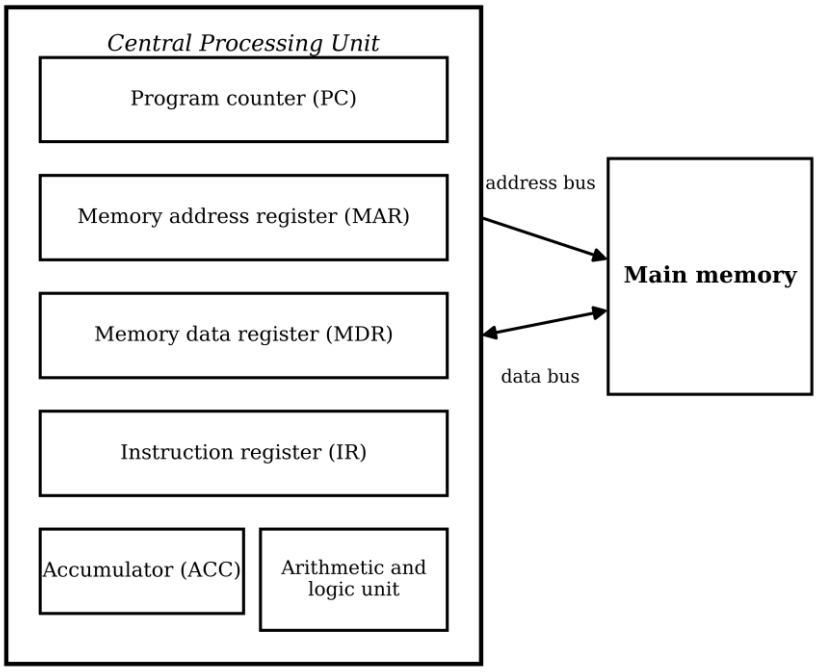


Figure 1.6: *The register set. Only the MAR drives the address bus and only the MDR sits on the data bus.*

Register	**What it holds**	**When it changes**
Program counter (PC)	The address of the next instruction to be fetched	Advanced during every fetch; overwritten by a jump or a taken branch
Memory address register (MAR)	The address of the location about to be read or written	At the start of every memory access, from the PC or from the instruction
Memory data register (MDR)	The word in transit to or from memory, either direction	On a read when memory answers; on a write before memory is told
Instruction register (IR)	The instruction now being decoded and obeyed	Once per instruction, at the end of the fetch phase
Accumulator (ACC)	An operand going into the ALU and the result coming out	Whenever an arithmetic, logic or load instruction completes
Status register (flags)	Single bits recording zero, negative, carry and overflow	As a side effect of nearly every operation the ALU performs

Table 1.4: *The special-purpose registers.*

Six stores, six jobs, no overlap. Learn what each may hold and half the instruction cycle explains itself, because the cycle is only words moving between these six in a fixed order.

The Master Key: *Name each register by the one thing it may hold, because a register that could hold anything would tell the control unit nothing.*

BINDER Question 19

A shopkeeper near the fish market in Musoma sends the boy for two crates from the shelf behind the counter, naming the shelf. She never mentions the store room at the back, though everything passes through it. Only one is ever named.

Distinguish a register from a cache.

Model Answer

A register is named by the programmer in the instruction set and is part of the machine's architecture, while a cache is managed entirely by hardware and is invisible to the program, so visibility rather than size is the real difference.

What visibility actually means: you can write a register into an instruction. An instruction that adds one register to another names both. No instruction anywhere names a line of cache, so the program cannot address it, cannot choose what is kept in it, and cannot tell where a value came from.

Who decides what is held: you do, for a register, or your compiler does for you, and the decision is fixed permanently in the instruction stream. The cache decides for itself, keeping what was just asked for and evicting what it judges least useful. Nothing in your program authorises it.

Which side of the line each falls: architecture is what the program can see; organisation is how the machine is built underneath. Registers are architecture and cache is organisation, which is exactly why adding cache never changes what a program means. The same compiled program runs unaltered and gives identical answers, only sooner.

The differences everybody quotes: a register holds a few bytes and answers inside the clock period, while a cache holds kilobytes and answers in a handful of cycles. Both are consequences of the first difference rather than the definition of it.

The sentence to stop writing: that a register is simply a very small, very fast cache. It makes the distinction a matter of degree when it is a matter of kind.

So the question to ask of any store inside a processor is not how big or how fast, but whether a program can name it. Registers can be named and caches cannot.

The Master Key: *Registers are architecture because you can name them; cache is organisation because you never can.*

BINDER Question 20

At the ginnery in Shinyanga a bale is weighed, then graded, then priced, and only then is a store opened. Nobody unlocks a store before the price is settled, because the price decides which store. Change that order and the work becomes impossible.

Describe the instruction execution cycle.

Model Answer

The instruction cycle runs in five stages in one fixed order, fetch, decode, execute, memory access and write back, and memory access follows execute because the address to be touched does not exist until execute has computed it.

Fetch: the counter supplies an address to the memory address register, memory answers into the memory data register, and the word is copied into the instruction register while the counter is advanced. Nothing has been understood; a word has merely arrived.

Decode: the control unit reads the opcode in the instruction register and works out what the instruction is, which registers it names and what the arithmetic and logic unit must do. Control signals are prepared; nothing has been done.

Execute: the arithmetic and logic unit performs the operation. For arithmetic that yields a result. For a load or a store it yields an address instead, formed by adding a base register to a displacement in the instruction. That effective address is why the stages run in this order.

Memory access: only now may a location be read or written, because only now is it known which location. That is why the stage stands alone and follows execute rather than preceding it. Instructions that never touch memory pass through doing nothing.

Write back: the result goes into the destination register, whether it came from the arithmetic and logic unit or from the memory just read. A store has nothing to write back and idles here too. The counter already points at the next instruction, so the machine begins again.

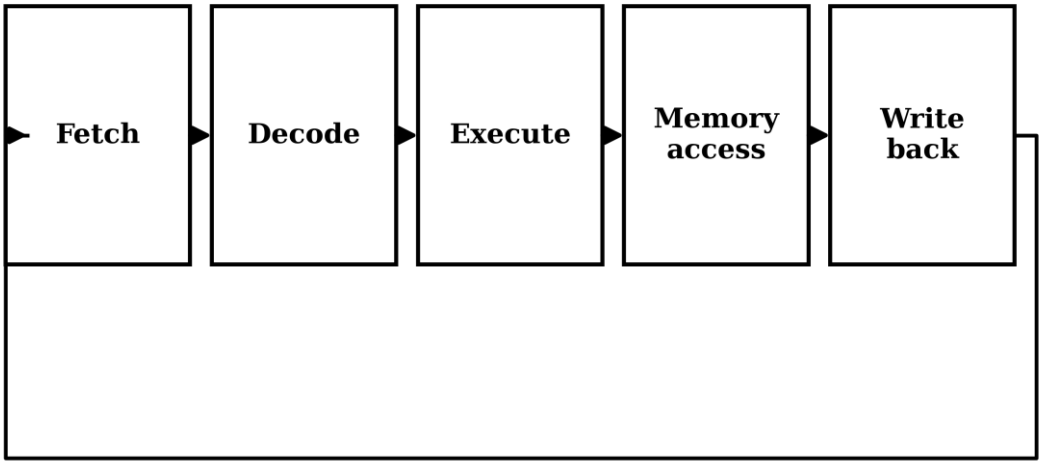


Figure 1.7: *The five stages of the instruction cycle.*

Five stages, then, but not five for every instruction. A register addition truly uses three and a store uses four, and the fixed order lets one set of hardware run them all without asking what it holds.

The Master Key: *The address must exist before memory can be touched, which is why memory access is a stage of its own and sits after execute.*

BINDER Question 21

The stores clerk at a Songea depot writes every movement the same way round: where it went first, then where it came from. It looks back to front until you see he never reads a line twice. A processor keeps the same discipline.

Explain register transfer language and write out the fetch phase in it.

Model Answer

Register transfer language writes each movement of data inside the processor as one short line, and one convention makes it readable: fix it once and never break it, because the arrow points at the destination, so the form is always destination ← source.

The notation itself: a bare name means the register itself. Square brackets mean the contents, so [PC] is the value held in the program counter. Brackets twice over mean the contents of the location whose address is held there, so [[MAR]] is the word in memory at the address in the address register.

The one convention: MAR ← [PC] means the contents of the program counter are copied into the memory address register. Copying does not empty the source, so the counter still holds what it held. Never write the reverse form on the next line; a trace with mixed directions cannot be read.

Step one of the fetch: MAR ← [PC]. The address of the next instruction is copied out of the program counter into the only register able to drive the address bus. The processor has said which location it wants and nothing more.

Step two: MDR ← [[MAR]] and PC ← [PC] + 1, both inside the same clock period, because transfers written on one line happen together. Memory is read and the word arrives in the data register, while the increment rides alongside because nothing still needs the old value.

Step three: IR ← [MDR]. The instruction is copied from the data register into the instruction register, where it sits still for as long as decoding and execution take. The data register is thereby freed for the operand this instruction may demand.

Step	Register transfer	In words
1	MAR ← [PC]	The address of the next instruction is copied from the program counter into the memory address register.

Step	Register transfer	In words
2	MDR ← [[MAR]]	The word at that address is read out of memory into the memory data register.
2	PC ← [PC] + 1	In the same period the program counter is advanced to the following location.
3	IR ← [MDR]	The instruction is copied from the memory data register into the instruction register.

Table 1.5: *The fetch phase in register transfer language.*

Three steps, and the fetch is finished for every instruction the machine will ever run, because the fetch neither knows nor cares what it has just fetched.

The Master Key: *Fix the arrow on the destination and a trace reads itself; mix the directions and nobody, yourself included, can follow it.*

REAL Question 22

In the stock office of a fish plant on the Mwanza shore, a small machine counts crates all day. It has three instructions in front of it and no idea what a crate is. Watch it read the first two, register by register.

Trace the contents of the PC, MAR, MDR, IR and accumulator through the first two fetch cycles of this program, showing the state of every register after each step.

Location 100 holds LDA 200, meaning load the accumulator from location 200

Location 101 holds ADD 201, meaning add the contents of location 201 to the accumulator

Location 102 holds STO 202, meaning store the accumulator into location 202

Location 200 holds 40, location 201 holds 12, and location 202 holds 0

The program counter starts at 100 and the accumulator starts at 0

Model Answer

After the first fetch the counter holds 101, the address register 100, the data and instruction registers both LDA 200, and the accumulator still 0; after the second the counter holds 102, the address register 101, those two both ADD 201, and the accumulator 40, loaded by the first instruction in between.

The state before anything happens: the counter holds 100 and nothing else holds anything useful. The accumulator reads 0. That is all the machine knows: not what it is about to do, only where to look for it.

The first fetch: MAR ← [PC] puts 100 into the address register. Memory answers, LDA 200 arrives in the data register, and the counter advances to 101 in the same period. Then IR ← [MDR]. The counter has moved on before the instruction is even decoded.

Why two registers briefly agree: the transfer copies, it does not move, so for a moment both hold LDA 200. The data register is a doorway and the instruction register a seat; within one step the doorway is wanted again while the seat stays occupied.

What execution does in between: LDA 200 sends 200 to the address register, memory answers 40, and the accumulator takes it. Those two now hold a data address and a data value rather than an instruction address and an instruction, so their contents mean nothing unless you know the step.

The second fetch: MAR ← [PC] now loads 101, because the counter advanced during the first fetch and not since. Memory answers ADD 201, the counter becomes 102, and the instruction register takes the new instruction over the old. Writing a stale counter value here is the commonest trace error.

The rule the trace reveals: the counter always runs one ahead. While ADD 201 is being decoded it already reads 102, which is not a fault but the thing that keeps the sequence continuous, and it is why a branch must overwrite the counter rather than adjust it.

How to check your own trace: after any completed fetch three values must agree. The address register holds the address of the instruction now in the instruction register, and the counter holds one more. Here that is 101, ADD 201 and 102. If they do not line up the trace is wrong.

Step	PC	MAR	MDR	IR	ACC
Before the first fetch	100	empty	empty	empty	0
Fetch 1 step 1: MAR ← [PC]	100	100	empty	empty	0
Fetch 1 step 2: MDR ← [[MAR]]; PC ← [PC] + 1	101	100	LDA 200	empty	0
Fetch 1 step 3: IR ← [MDR]	101	100	LDA 200	LDA 200	0
Execute of LDA 200: ACC ← 40	101	200	40	LDA 200	40
Fetch 2 step 1: MAR ← [PC]	101	101	40	LDA 200	40
Fetch 2 step 2: MDR ← [[MAR]]; PC ← [PC] + 1	102	101	ADD 201	LDA 200	40
Fetch 2 step 3: IR ← [MDR]	102	101	ADD 201	ADD 201	40

Table 1.6: *A traced fetch.*

The next execution would make the accumulator 52, and the instruction after that would put 52 into location 202, and the office would call the answer a stock count. The machine would call it five registers moved in order.

The Master Key: *Trace the registers and not the meaning, because the machine has no access to the meaning either.*

REAL Question 23

Three sentences go up on the board of a study room above a spice store in Zanzibar, and all three sound properly learned. Each is nearly right, which is the difficult kind. A wildly wrong sentence corrects itself the moment it is read aloud.

Identify the three mistaken statements and put each one right.

1. “The MDR holds the address of the next instruction.”

2. “Adding cache to a processor changes its architecture.”

3. “In the instruction cycle the operands are read from memory before the execute stage.”

Model Answer

All three are wrong, and each confuses a thing with its neighbour: the memory data register with the program counter, organisation with architecture, and the stage that computes an address with the stage that uses it.

The first error, on what the data register holds: whatever word is in transit between processor and memory, in either direction, and for one access only. The address of the next instruction lives in the program counter until the following fetch. Corrected: the program counter holds that address, and the data register holds the value in transit.

Why that error is easy to make: the data register really does hold instructions during a fetch, and the address register beside it really does hold addresses, so all three blur. Ask two questions of each instead. What does it hold, and for how long.

The second error, on what cache changes: a cache has no name in the instruction set and no instruction can address it, so no program can discover it except by running sooner. Architecture is precisely what the program can see. Corrected: adding cache changes organisation, not architecture.

The test that settles it: run the same compiled program with the cache and without. Identical results, different times, and nothing the program means has moved.

The third error, on where memory access sits: the operand’s address is not known until execute has produced it. A load computes its effective address during execute by adding a base register to a displacement in the instruction. Corrected: memory access follows execute, because execute computes the address memory access uses.

What that would require: reading a location before knowing which location. That is not a slow operation or an expensive one, it is not an operation at all. The order of the stages is forced by dependency, not chosen for neatness.

The habit that catches all three: before writing about a register or a stage, name what it needs and what it produces. The data register needs an address already in the address register. The memory stage needs an address already computed by execute. The cache needs nothing at all, which is why it stays invisible.

Each of these three would earn a nod in conversation and nothing on paper, because each names the right area of the machine and then the wrong member of it.

The Master Key: *Ask what a thing needs before it can act, and a sentence about the wrong register stops being possible to write.*

REAL Question 24

At a junction outside Morogoro a dispatcher waves three lorries north, then word comes that the convoy is going west. The new order takes a second to give. Turning three lorries round takes the morning. Everything expensive about a branch lives in that gap.

Explain what happens to the registers when a branch instruction is executed, and why a taken branch costs more than an ordinary instruction.

Model Answer

A branch executes by overwriting the program counter instead of letting the increment stand, and a taken branch costs more than an ordinary instruction because everything fetched behind it came from the road not taken and must be thrown away.

The registers on an unconditional branch: the instruction is fetched into the instruction register as usual and the counter has already advanced past it. Decode recognises a jump, execute computes the target, and the counter is written with it. The increment is discarded: the counter is not nudged, it is replaced.

What a conditional branch adds: the status register. Flags were set by whatever ran before, and the branch tests one of them. If the condition is false the counter keeps its incremented value; if true it is overwritten. Either way nothing is written back.

Why the fetch unit is already wrong: while the branch was being decoded and executed the fetch unit did not wait. It carried on from the addresses following the branch, because those were the only ones it had. Overwrite the counter and every one of them is from the wrong place.

What discarding involves: those instructions are removed before any can write to a register or to memory, and the stages holding them are emptied. Work already done cannot be saved, because the machine must look as though they were never fetched.

Where the cost falls: not in the branch, which is one addition on an address, but in the cycles that follow while the fetch unit starts again and the stages refill. The more instructions in flight, the more a taken branch destroys.

Why a branch not taken is nearly free: the counter keeps the value the fetch already gave it, so the instructions behind it were the right ones after all. Nothing is discarded. The same instruction testing the same flag costs almost nothing one way and several cycles the other.

What designers do about it: they guess. The fetch unit predicts which way the branch will go and fetches down that road before the condition is known. A correct guess costs nothing; a wrong one costs what an unpredicted taken branch costs.

The honest description of a taken branch is not that the jump is slow. The jump is one addition. What is slow is the work the machine had confidently begun on the assumption that none was coming.

The Master Key: *The program counter is overwritten rather than advanced, and the price is paid by everything already fetched behind it.*

HOT Question 25

A class in a workshop above the market in Iringa is asked which single register it would keep if the others were taken away. The room divides in seconds and neither half is being foolish. What settles it is not a fact about hardware.

"The program counter is the single most important register in the machine." To what extent do you agree?

Model Answer

The claim is half right: the program counter is the most important register for control, since without it there is neither sequence nor branch, but the instruction register decides what happens, the accumulator and flags carry every result, and a counter alone computes nothing.

The case for, on sequence: the program counter alone is concerned with the future. Every other register describes the instruction in hand; this one says what comes next. Remove it and the machine cannot find a second instruction at all.

The case for, on control flow: every jump, loop, procedure call and return is an operation on the program counter and nothing else. Selection and iteration, the structures from which all programs are built, exist in hardware only as overwrites of this one register.

The case for, on interrupts: an interrupt is handled by saving the counter, loading a handler address, and restoring the saved value afterwards. Procedure calls work the same way, and multitasking itself rests on that one trick.

The case against, on the instruction register: the counter says where and the instruction register says what. The decoder never looks at the counter, and every control signal comes from the bits in the instruction register. A counter without it would point faithfully at locations nobody would read.

The case against, on the accumulator and the flags: computation happens in the arithmetic and logic unit and its results live in the accumulator. With nowhere to hold an operand and receive a result, the machine fetches, decodes and produces nothing. Worse for the claim, a conditional branch cannot decide until the status register is set.

The case against, on the word important: it is doing two jobs at once. If it means indispensable, every register qualifies, since removing any one stops the machine. If it means central, the answer turns on whether a computer sequences or calculates.

The judgement: agree to a limited extent and state the limit exactly. The counter is the most important register for control flow, and control flow is what separates a computer from a calculator. But most important in the machine cannot stand, because the counter neither knows what an instruction is nor holds a result. Take it away and the machine cannot go anywhere; leave only it and there is nowhere worth going.

The better question is not which register matters most but what each makes possible that no other can, and answered that way the counter has a clear and modest claim: it owns the future.

The Master Key: *The program counter is necessary and nowhere near sufficient; it chooses the road and never once drives.*

HOT Question 26

The route map on the wall of the bus stand in Dodoma draws the road to Morogoro as one straight stroke. It leaves out every bend and village, and no traveller is ever misled, because nobody consults a wall map to learn about bends.

Assess the claim that the five-stage description of the instruction cycle is an oversimplification.

Model Answer

The claim is accurate about hardware and wrong as a criticism, because real processors do overlap the stages, skip them and subdivide them, and none of that damages a model built to show which step depends on which.

The case for, on overlapping: no modern processor runs five stages for one instruction and then starts afresh. They are pipelined, so five instructions occupy them at once, and instructions may be issued and completed out of order. The single file of the diagram exists nowhere in silicon.

The case for, on skipped stages: an addition between two registers never touches memory and a store writes nothing back. For most instructions at least one stage is an empty formality, so presenting all five as universal suggests a uniformity that is not there.

The case for, on microoperations: each stage is itself a sequence of register transfers clocked separately. The fetch alone is three, and on some machines one complex instruction is carried out by a small program in microcode. A stage is a bundle of steps, not an indivisible unit.

The case for, on the boundaries: five is not a fact about computation. Some accounts give three stages, some four, some six, and each is defensible. The number is a convenience for the learner.

The case against, on what the model gets right: the dependency order is real and no implementation escapes it. You cannot decode a word you have not fetched, address a location before the address is computed, or write back a value you have not obtained. Pipelining changes when the stages happen, never which precedes which.

The case against, on what a model is for: a model earns its place by what it lets you predict. These five explain why a load takes longer than a register addition, why a taken branch is expensive, and where a data hazard comes from. Accuracy that explained nothing would be the greater failure.

The judgement: the criticism is true and misdirected. Every clause of it is about implementation, while the five stages are about dependency, so the two never meet. Keep the five as the frame and hang the complications on it: overlapping is pipelining, skipped stages are why timings differ, microoperations are how one stage is built. The other way round, they have nothing to attach to.

So call it a simplification and go on using it. The wall map in Dodoma leaves out the bends and still puts every traveller in the right town in the right order.

The Master Key: *The five stages are not a picture of a processor but a picture of what must happen before what, and that ordering no design has ever escaped.*

Boolean laws and algebraic simplification

Every logic circuit you will ever design can be written as an expression, and every expression can be rewritten. This section gives you the rules that make rewriting legal, including the two that most lists forget, and a working order for applying them. You will simplify by hand, cost the saving in gates, catch a student misusing the distributive law, and learn the harder lesson of when a simplification should be refused.

BINDER Question 27

The coral stone doorways at Bagamoyo were cut by masons who never wrote their rules down, yet every arch obeys them. Boolean algebra is the opposite case. Its rules fit on one page, and most students read that page once and never go back.

State the laws of Boolean algebra and explain what each one lets you do.

Model Answer

The laws of Boolean algebra are the legal moves for rewriting a logic expression, and each one licenses a different kind of change: reordering, regrouping, factoring, expanding, or making a term vanish.

The laws that let you rearrange: commutative and associative. Order and grouping never affect an AND or an OR, so terms may be shuffled and rebracketed freely. They change nothing by themselves; they let you move a term next to the one that will cancel it.

The law that factors and expands: the distributive law, read either way. Left to right it opens a bracket; right to left it pulls out a common factor, which is the most productive move in any simplification. Its OR form has no counterpart in ordinary arithmetic, and it is the law that turns A + NOT A.B into A + B.

The laws that bring in a constant: identity and null. ANDing with 1 or ORing with 0 leaves a variable untouched; ANDing with 0 or ORing with 1 destroys it. These are the laws that finish a proof, because they are how a term finally becomes nothing or everything.

The laws that pair a variable with itself: idempotence deletes a duplicate. The complement law cancels a variable against its own negation, giving 0 under AND and 1 under OR. Double complement says two negations undo each other.

The two laws most lists leave out: absorption and consensus. Absorption throws away an entire product term because a variable standing alone has already covered it. Consensus deletes a term that the other two together already imply. These finish an answer when the obvious moves have run out.

Law	Statement	Statement
Commutative	A.B = B.A	A + B = B + A
Associative	(A.B).C = A.(B.C)	(A + B) + C = A + (B + C)
Distributive	A.(B + C) = A.B + A.C	A + B.C = (A + B).(A + C)
Identity	A.1 = A	A + 0 = A
Null (annulment)	A.0 = 0	A + 1 = 1
Idempotent	A.A = A	A + A = A
Complement	A.NOT A = 0	A + NOT A = 1
Double complement	NOT (NOT A) = A	One statement serves both forms
Absorption	A.(A + B) = A	A + A.B = A
Consensus	(A + B).(NOT A + C).(B + C) = (A + B).(NOT A + C)	A.B + NOT A.C + B.C = A.B + NOT A.C

Table 1.7: *The laws of Boolean algebra.*

Learn the page as ten laws rather than twenty statements, because each has an AND form and an OR form mirroring the other.

The Master Key: *Every law is a permission, so name the permission before you make the move.*

BINDER Question 28

A student in Musoma is three lines into a simplification and stuck. The term she cannot touch is in plain sight, and the law that removes it is not on the list she copied down. It is the shortest law of the ten.

Explain the absorption law and prove it, then confirm the proof with a truth table.

Model Answer

The absorption law says that a term already contained inside another adds nothing, so A + A.B = A and A.(A + B) = A, and it lets you delete a whole product term without disturbing the rest of the expression.

What the law actually claims: in A + A.B the second term can only be true when A is true, and whenever A is true the first term has already made the output true. The second term therefore never changes an answer anywhere, so it may be struck out. The larger term absorbs the smaller one whole.

The proof in four moves: start from A + A.B. The identity law lets you write A as A.1, giving A.1 + A.B. The distributive law pulls A out of both terms, giving A.(1 + B). The null law makes 1 + B into 1, leaving A.1. The identity law returns that to A.

The other form proved the same way: A.(A + B) expands by the distributive law into A.A + A.B. Idempotence turns A.A into A, leaving A + A.B, which the first proof has already reduced to A. The two statements are one law seen from opposite sides.

Reading the proof off the table: Table 1.8 sets out both variables. The last column reads 0, 0, 1, 1 down the page, which is column A exactly. The columns agree on every row, and four rows exhaust everything two variables can do, so the identity is proved again by a different method.

Why it earns its place: absorption removes work rather than rearranging it. Factoring only reshapes an expression; absorption shortens it. Look for it whenever a variable stands alone beside a product that contains that same variable.

A	B	A.B	A + A.B
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

Table 1.8: *Absorption proved.*

Two proofs of one line is not extravagance. The algebra shows you why the term disappears and the table shows you that it does, and either will be accepted.

The Master Key: *If a variable stands alone, every product that contains it has already been paid for.*

BINDER Question 29

A workshop in Shinyanga assembles alarm panels from loose gates bought by the tray. The designer who draws three gates and the designer who draws the same function with two are paid the same fee. Only one has cut the bill of materials.

Explain why simplification is worth doing, and describe a strategy for approaching one.

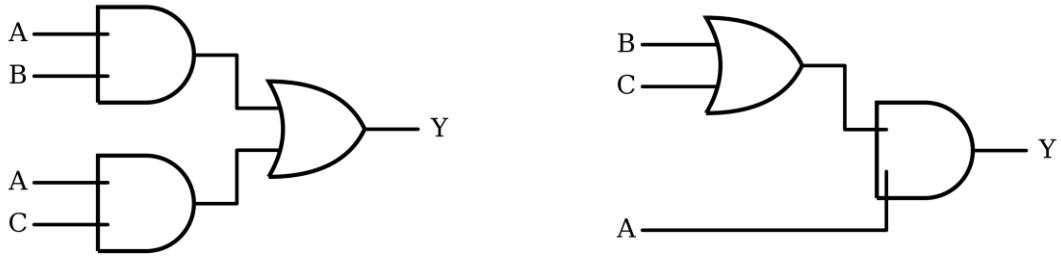
Model Answer

Simplification is worth doing because every gate removed takes cost, power, heat and delay away with it, and it is best approached in a fixed order: common factor first, then absorption, then complement pairs, then a check that the result still matches what you started with.

Fewer gates cost less money: a gate is silicon area on a chip or a package on a board. Cut three gates to two and you have cut a third of the area, a third of the soldered joints and a third of the parts somebody must buy and stock.

Fewer gates draw less power and make less heat: every gate draws current each time it switches, and every milliwatt becomes heat that must leave the enclosure. In battery equipment that is running time; in a sealed cabinet it is whether the thing survives an afternoon.

Fewer gates in a chain answer sooner: a signal takes real time to cross each gate, and the delays add along the path. Removing a layer removes its delay from every operation the circuit performs, which raises the clock speed the design can be trusted at.



three gates for A.B + A.C *two gates for A.(B + C)*

Figure 1.8: *The same function before and after simplification. Three gates become two.*

The strategy, in order: look for a common factor first, because factoring exposes everything else. Then absorption, since a lone variable beside a product containing it deletes that product outright. Then complement pairs, because a variable meeting its own negation collapses to a constant, and the constant then eats its neighbours.

The step nobody should skip: check the result against the original on every row. Simplification is a chain of small rewrites, and one mistyped negation ruins all of it silently, because a wrong expression still looks like an answer.

Simplify in that order and you will rarely stall, because each stage sets up the next. Skip the final check and you will hand in a tidy, confident, wrong circuit.

The Master Key: *Gates cost money, power, heat and time, so every one you delete is paid for four times over.*

BINDER Question 30

The working that convinces in Songea and the working that does not are the same five lines. The difference is that one says which law was used at each line and the other leaves the reader to guess. The algebra is the easy part.

Simplify A.B + A.NOT B + NOT A.B, naming the law used at each step.

Model Answer

A.B + A.NOT B + NOT A.B reduces to A + B in five steps, and every step is one named law.

Step one, factor A out of the first two terms: the first two terms share A, so the distributive law pulls it out. A.B + A.NOT B becomes A.(B + NOT B), and the whole expression is now A.(B + NOT B) + NOT A.B.

Step two, cancel the complement pair: B + NOT B is a variable ORed with its own negation, which the complement law makes 1. The expression becomes A.1 + NOT A.B.

Step three, drop the one: the identity law says A.1 = A. The expression is now A + NOT A.B, already shorter than the three product terms you began with.

Step four, distribute over the OR: the OR form of the distributive law turns A + NOT A.B into (A + NOT A).(A + B). This is the step students skip, because the OR form of the law looks wrong to anyone reading it as ordinary arithmetic.

Step five, cancel and finish: A + NOT A is 1 by the complement law, leaving 1.(A + B), and the identity law reduces that to A + B. Three product terms and two inverters have become a single OR gate.

The check that closes the answer: the original is true whenever A is true, true whenever B is true, and false only when both are 0. A + B behaves identically, so the two agree on all four rows.

Notice that step four did the real work and step one only made it visible. Factoring rarely finishes an answer, but nothing else starts one.

The Master Key: *Name the law on every line, because the naming is what is being marked and not the answer.*

REAL Question 31

The borehole pump at a girls' secondary school in Sumbawanga lifts water to a roof tank. A timer went into its control panel one term, an override button for the matron the next. The panel works. It also contains four gates that do nothing.

Translate the three control rules into a Boolean expression, simplify it, and state how many gates are saved.

Rule one: the pump runs whenever the roof tank is below its low mark.

Rule two: the pump runs when the roof tank is below its low mark and the morning timer contact is closed.

Rule three: the pump runs when the roof tank is not below its low mark and the override button is pressed.

Let L be the tank below its low mark, T the timer contact, and B the override button.

Count two-input gates only, and count each inverter as a gate.

Model Answer

The three rules give P = L + L.T + NOT L.B, which needs five gates as written and simplifies to P = L + B, a single OR gate, saving four gates out of five.

Turning the rules into an expression: the rules are alternatives, so each becomes a product term and the three are ORed. Rule one gives L alone, rule two gives L.T, rule three gives NOT L.B, so the panel implements P = L + L.T + NOT L.B.

Counting the gates before you touch anything: L.T needs one AND. NOT L needs an inverter and NOT L.B a second AND. Joining three terms with two-input ORs needs two more. That is five gates, all five soldered into the panel.

The first cut, absorption: L stands alone and L.T is a product containing L, which is exactly the pattern absorption removes, so L + L.T = L. Rule two was never a rule at all. It described a case rule one had already covered, so the AND gate serving the timer does nothing.

The second cut, distributing over the OR: what remains is L + NOT L.B. The OR form of the distributive law rewrites this as (L + NOT L).(L + B). The complement law makes L + NOT L into 1, and the identity law reduces 1.(L + B) to L + B. The inverter and the second AND fall away.

The gate count afterwards: P = L + B is one two-input OR gate. Five become one, four are saved, and the inverter goes with them.

Checking the matron still gets what she was promised: when the tank is low the pump runs whatever the button does, which was rule one. When it is not low the pump runs only if the button is pressed, which was rule three. Nothing the school relies on has changed, only the wiring behind it.

Why the redundancy was there at all: three people wrote three rules across six years and each was reasonable alone. Redundancy in real control logic rarely comes from bad thinking. It comes from good thinking done at separate times, which is why simplification belongs at the end of a design.

Four gates on one panel is a small saving. Four on every panel a workshop builds for a district is a different figure, and two laws found it.

The Master Key: *Rules written years apart overlap, so gather them into one expression before you believe any of them is needed.*

REAL Question 32

Three lines of working go up on a Morogoro blackboard, and the class copies them without a murmur. The answer is short, which everyone takes as a sign of health. The rot is in the first line, where a law is quoted backwards.

Identify the error in the student's working, state the correct law, show a row of the truth table on which the student's version fails, and complete the simplification correctly.

Simplify (A + B).C + B

Line 1: (A + B).C + B = (A + C).(B + C) + B

Line 2: = A.B + A.C + B.C + C + B

Line 3: = B + C

Model Answer

The error is in line one: the student has written (A + B).C = (A + C).(B + C), when the distributive law gives (A + B).C = A.C + B.C, and the correct simplification is A.C + B rather than B + C.

Where the error sits: line one, before any real work has been done. Lines two and three are competent manipulation of a false start, which is why the working looks healthy. No amount of good algebra afterwards can recover it.

The correct law: AND distributes over OR, so (A + B).C = A.C + B.C. Each term inside the bracket is ANDed with C separately and the two results are ORed.

Where the wrong pattern came from: the shape used is a real law, but the other one. OR distributes over AND as A + B.C = (A + B).(A + C), and that is the form with a repeated variable in two brackets. Taking the OR form and applying it to an AND is the commonest slip in this topic.

The row that convicts: take A = 0, B = 0, C = 1. The true left side is (0 + 0).1, which is 0. The student's right side is (0 + 1).(0 + 1), which is 1.1 and therefore 1. One side says 0 and the other says 1 on the same inputs, so the claim is dead.

Why one row was so easy to miss: rule all eight rows and the student's version agrees on six, failing only at A = 0, B = 0, C = 1 and at A = 1, B = 1, C = 0. Spot-check A = 1, B = 0, C = 0 and both sides give 0, which proves nothing. A law that survives most rows is more dangerous than one that fails everywhere.

The simplification done properly: start again from (A + B).C + B. The distributive law gives A.C + B.C + B. Now B stands alone beside B.C, so absorption deletes B.C, leaving A.C + B.

How far the answer was out: A.C + B and the student's B + C differ on exactly one of the eight rows, at A = 0, B = 0, C = 1, where the student's circuit switches on and the correct one stays off. One row is enough to flood a field.

The habit that would have caught it: test a suspect law on one row before you build three lines on it, and pick a row where the variables differ from each other. The easy rows are exactly the ones a false law survives.

The student was punished for quoting a real law in the wrong place, which is the most expensive error of all, because everything after it is fluent and wrong.

The Master Key: *A law you cannot test on a single row is a law you are borrowing rather than using.*

HOT Question 33

An engineer in Mwanza watches a synthesis tool turn four hundred lines of description into a gate netlist in eleven seconds, one no person could check. On the shelf above the machine is the page of laws she copied out in Form Five.

"Boolean algebra is only a mathematical curiosity; real engineers use software to minimise circuits." To what extent do you agree?

Model Answer

The claim is accurate about who does the labour and badly wrong about what the algebra is, because the software is these laws mechanised, and an engineer who cannot read them cannot specify, constrain, verify or debug anything the tool produces.

The case for the claim, on scale: a modern logic block holds millions of gates. Nobody factors that by hand, and nobody ever did. A synthesis tool performs in seconds a quantity of rewriting a career of paper could not approach, and repeats it on every recompilation.

The case for the claim, on quality: the tool searches a space no person can hold in mind, balancing area against delay against power at once. Hand algebra optimises one thing only, the count of terms, which is frequently not the thing that matters.

The case for the claim, on how designs are written now: the engineer describes behaviour in a hardware description language and never draws a gate. Gates are an output of the process rather than an input, so hand methods look like a stage the profession has passed.

The case against, on what the software is made of: this is the decisive point. Absorption, consensus, the distributive law and the complement law are the rewrite rules inside the tool. Calling the algebra a curiosity while running a program built entirely out of it is calling the foundation ornamental because you happen to stand on the floor above.

The case against, on the work the tool cannot do: a tool minimises whatever it is handed. Writing the conditions correctly, deciding which redundant term must be kept because it covers a hazard, and reading the report afterwards are all done in the algebra. Give a tool a wrong expression and it returns a beautifully minimal wrong circuit.

The case against, on the circuits still built by hand: interlocks, alarm chains, motor reversing logic and the glue on a board are small, and small logic is still reasoned out on paper by whoever understands the plant. Most of the logic in a district is of that kind.

The judgement: agree with the practice and reject the word curiosity. Software has taken the labour of minimisation completely and will not give it back, but it has taken none of the understanding. The algebra stopped being a technique for producing circuits and became the language for describing, constraining and checking them, which is a promotion rather than a retirement.

Ask who would notice if the tool returned a subtly wrong netlist. The answer is always somebody who knows the laws, which settles what they are worth.

The Master Key: *Software turns the handle, but the machine it is turning is built out of these laws.*

HOT Question 34

Two students in Iringa reach the same simplification question with eighteen minutes left. One starts factoring and is finished in four, or would be if the factor were there. The other rules eight rows and begins filling them in. Only one fails safely.

Compare simplification by algebra with simplification by inspection of a truth table, and assess which is more reliable under examination conditions.

Model Answer

Algebra is quicker and earns the method marks but depends on seeing a move, while a truth table is slower and blind to elegance but cannot deceive you, so the reliable practice is to reduce by algebra and confirm by table, falling back on the table alone when the move will not come.

What algebra does well: it works on the expression itself, so it is compact, it scales to any number of variables, and every line is justified by a named law. When the common factor is visible the whole thing collapses in three lines.

Where algebra fails you: it requires insight on demand. If no factor is apparent you stall, and stalling under time costs more than slow arithmetic. Worse, a misapplied law produces working that looks fluent all the way to a wrong answer, as a dropped negation will, without leaving a mark on the page.

What the truth table does well: it needs no insight at all. Rule the rows, evaluate the original, evaluate the candidate, and compare the two columns. It is mechanical, it can be done while tired, and it either agrees on every row or it does not.

Where the truth table fails you: rows double with each variable. Three give eight rows, four give sixteen, five give thirty-two, and by then the method is unusable under time pressure. It also does not hand you a minimal form; reading it off gives a sum of products, which needs the algebra anyway.

Reliability compared honestly: the table is the more reliable, because its only failure is a copied bit and that failure is visible on the page. An algebraic error hides inside correct-looking notation. Reliability is not about which method is cleverer but about which one tells you when it has gone wrong.

What the marks actually reward: this cuts the other way. A simplification question asks for the laws, so a table alone earns the answer and loses the method. A named law on each line earns credit even when the final expression is wrong, which is protection no table offers.

The judgement: use both, in that order. Simplify by algebra naming every law, then verify on the table when three variables leave you only eight rows. If the algebra will not start, read the answer off the table and reconstruct the laws backwards from it, because an answer with weak method beats no answer at all.

The two methods are not rivals but a producer and an auditor, and no serious answer has ever suffered from being audited.

The Master Key: *Algebra shows why an answer is right and a table shows that it is, so produce with one and check with the other.*

HOT Question 35

A firm in Tanga rebuilds a conveyor interlock, seven gates down to four, and signs off the drawing with satisfaction. Weeks later the line stops twice a shift for no reason anyone can reproduce. The new circuit is provably correct. It is also worse.

Suggest and evaluate the possible reasons why the smaller circuit performs worse, and reach a judgement about when simplification should stop.

Model Answer

Seven gates in two layers will beat four gates in four, so the candidates are a deeper chain adding delay, a signal pushed past its fan-out, a hazard uncovered by the very term that was deleted, and a design nobody can maintain; the symptoms point hardest at the hazard.

The first reason, a chain that got deeper: gate count is area and gate depth is time, and factoring trades one for the other. Pulling a common factor out puts a new gate in series with everything behind it. At five nanoseconds a gate, two layers answer in ten and four layers in twenty.

The second reason, fan-out: factoring concentrates load. A signal that fed two inputs may now feed eight, and a driver worked beyond its rated fan-out gives slow, sagging edges and a shrunken noise margin. The remedy is a buffer, which puts back a gate and a delay.

The third reason, glitches from unequal paths: if two paths to one output pass through different numbers of gates, the signals arrive at different moments, and for a few nanoseconds the output can take the wrong value though the logic is correct. A latch or counter downstream acts on that glitch as though it were real.

Why the deleted term matters most here: the term minimisation removes first is often the consensus term, and that term is exactly what covers a static hazard. It is logically redundant, which is why the algebra deletes it, but it was never electrically redundant.

The fourth reason, a circuit nobody can read: in the seven-gate version each product term matched one written rule, so a technician could trace a complaint to a gate. After factoring the terms match nothing anybody wrote down, and the next change to the plant will be guesswork.

Evaluating which explanation fits the symptoms: twice a shift, unreproducible, and only since the rebuild. Extra delay would show as a firm ceiling on speed, not random stops. Fan-out trouble drifts with temperature and load. An uncovered hazard gives exactly this signature, so the deleted term is where to look.

The judgement: stop when the next reduction adds depth to the critical path, pushes a signal past its fan-out, deletes a term that covers a hazard, or breaks the match between terms and written rules. Fewest gates was only ever a proxy for cheapest. The real target is the cheapest circuit that meets timing, drives its loads, cannot glitch, and can still be understood.

Which is why the old board went back in, and rightly. A circuit is judged not by the elegance of the expression behind it but by whether the conveyor runs.

The Master Key: *The cheapest circuit is not the one with fewest gates but the one that meets timing, drives its loads and can still be understood.*

De Morgan’s theorems, algebraically and at gate level

Two short lines change what you are allowed to do with a circuit. De Morgan’s theorems are the only rules in Boolean algebra that carry an expression across the boundary between AND and OR, and they turn a negation from an obstacle into a doorway. This section proves them on paper, draws them as gates, teaches the bubble pushing that follows, and then spends them on real decisions: an alarm on the Rufiji and a purchasing choice in Musoma.

BINDER Question 36

The ferry out of Kigoma casts off only when the fuel has come aboard and the manifest has been signed. A clerk who wants to know when it will not sail has two ways of saying so, and both describe the same delay.

State both of De Morgan’s theorems and prove one of them by truth table.

Model Answer

De Morgan’s two theorems say that NOT (A.B) = NOT A + NOT B and that NOT (A + B) = NOT A . NOT B, and a four row truth table proves the first by showing that its two sides agree on every input the circuit can ever see.

The first theorem: NOT (A.B) = NOT A + NOT B. Said in words, it claims only that the inverse of an AND is the OR of the inverses. Move the negation inward and the dot becomes a plus.

The second theorem: NOT (A + B) = NOT A . NOT B, the same sentence with the operators exchanged. The inverse of an OR is the AND of the inverses. Each theorem does two things at once, carrying the negation inward and swapping the operator on the way.

What a truth table can prove: everything, so long as the variables are few. Two inputs give four combinations and there is no fifth. If two expressions agree on all four rows they agree always, so exhaustion here is not a sample of the evidence but the whole of it.

Filling the columns: work from left to right and never guess a whole row. Compute A.B first, invert it to get NOT (A.B), write NOT A and NOT B in separate columns, and only then OR those two to produce the final column.

Reading the proof off the table: compare the fourth column with the seventh. Both read 1, 1, 1, 0 going down. Four rows, four agreements, no exception. The two expressions are the same function of A and B, which is what the theorem claims.

A	B	A.B	NOT (A.B)	NOT A	NOT B	NOT A + NOT B
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Table 1.9: *De Morgan's first theorem proved.*

Two sentences in words, one table in numbers, and a result that follows you through every circuit you ever draw.

The Master Key: *A negation over a product becomes the sum of the negations, and the operator always changes as the negation moves inward.*

BINDER Question 37

In a Morogoro workshop a technician who has run out of OR gates finishes the board anyway, using NAND gates and a handful of inverters, and the customer never learns the difference. He is not improvising. He is applying a theorem you already know.

Explain De Morgan's theorems at gate level and describe what bubble pushing means.

Model Answer

At gate level the first theorem says a NAND gate and an OR gate with both inputs inverted are one circuit, and bubble pushing is the technique that moves inversions across a gate by changing the gate as they go.

The bubble is the negation: on a diagram an inversion is a small circle at an input or an output, so a NAND is an AND with a bubble on its output. Every negation in the algebra is one of these circles.

The first theorem, drawn: NOT (A.B) is an AND gate with a bubble on its output, which is a NAND. NOT A + NOT B is an OR gate with a bubble on each input. Both give the same output for every input, so a NAND is an OR gate with both inputs inverted.

The bubble pushing rule: to push a bubble through a gate, change AND to OR or OR to AND, and invert every input and the output. Nothing else. Learn that one line and you can redraw any gate in its other form without touching the algebra.

Why the operator has to change: a bubble moved past a gate is a negation moved past an operator, which is what the theorem governs. Move the bubbles but leave the AND standing and you claim NOT (A.B) = NOT A . NOT B, false on two of the four rows.

What the technique is for: matching the gates you have to the logic you want. A designer draws in whichever form reads naturally, then pushes bubbles until every gate is one the board carries. The picture changes; the function never does.

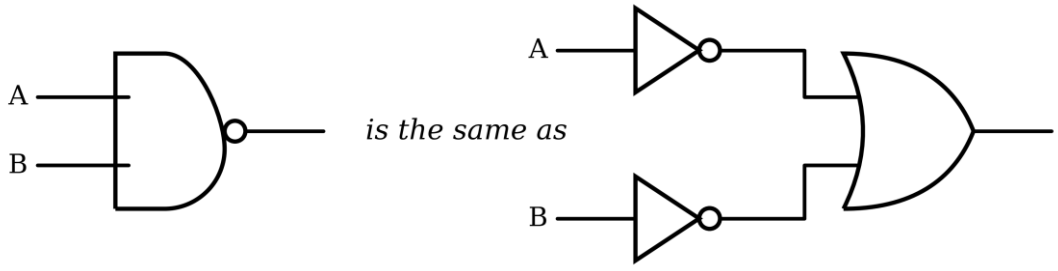


Figure 1.9: *De Morgan at gate level. A NAND gate is an OR gate with both inputs inverted.*

The algebra and the diagram are two views of one fact, and bubble pushing is the theorem done with a pencil.

The Master Key: *Push a bubble through a gate and the gate changes kind, AND becoming OR, while every input and the output flips.*

BINDER Question 38

The crane on the quay at Lindi lifts a load only when two conditions are both true, and it protests the moment either one fails. The fitter who wired that interlock had a single tube of chips on the bench and nothing else.

Explain De Morgan's second theorem at gate level and explain why the result matters to a manufacturer.

Model Answer

Drawn as gates, the second theorem says a NOR gate is an AND gate with both inputs inverted, and it matters industrially because this equivalence and its partner let a designer build an entire circuit from one kind of gate.

The second theorem, drawn: NOT (A + B) is an OR gate with a bubble on its output, which is a NOR. NOT A . NOT B is an AND gate with a bubble on each input. The two drawings are one circuit, so a NOR gate is an AND gate with both inputs inverted.

The same rule produced it: push the output bubble of the NOR backwards through the gate. OR becomes AND, the output bubble disappears, and one appears on each input instead. The same rule, run in the other direction, delivers the second theorem from the first.

Checking it on the awkward row: take A = 1 and B = 0. The NOR gives NOT (1 + 0), which is 0. The AND with inverted inputs gives NOT 1 . NOT 0, which is 0 . 1, also 0. They agree, and on the other rows too.

Why a manufacturer cares: the equivalence lets one gate type stand in for all the others. Any expression can be rewritten until every operation in it is a NOR, or every operation a NAND, so a board that needed three different chips is built from copies of one.

What that buys on the factory floor: one part number to purchase, one reel on the placement machine, one spare to carry to a breakdown, and a wafer laid out as an array of identical cells. Uniformity is worth money long before elegance is.

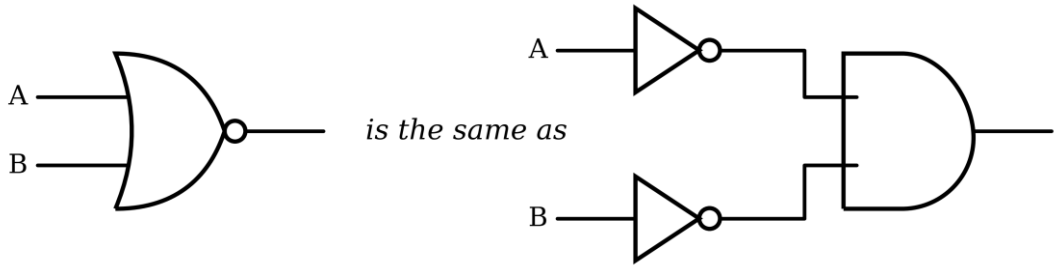


Figure 1.10: *A NOR gate is an AND gate with both inputs inverted.*

The theorem looks like a tidy piece of algebra and behaves like a purchasing decision, which is the usual fate of good mathematics.

The Master Key: *A NOR is an AND with inverted inputs, and that equivalence lets a whole board be built from one kind of gate.*

BINDER Question 39

A clerk in Mikindani is told the store may not be opened unless the manager is present and the register locked. He then writes when it may NOT be opened, and the second rule is not the first with a not in front.

Apply De Morgan's theorems to expressions of three or more variables, and state the rule for negating a whole expression.

Model Answer

De Morgan extends to any number of variables, and negating a whole expression means changing every operator and negating every term, not merely writing NOT in front of it.

The three-variable forms: the theorems do not stop at two. NOT (A.B.C) = NOT A + NOT B + NOT C, and NOT (A + B + C) = NOT A . NOT B . NOT C. The pattern is the same however many variables you have, because a product is only true when every term is true, so its negation is true whenever any one term fails.

The rule in one line: to negate an expression, swap every AND for an OR and every OR for an AND, then negate every individual term. Nothing else changes, and brackets keep the grouping they already had.

Work from the outside in: where negations are nested, deal with the outermost one first and leave the inside alone until you reach it. Trying to negate every level at once is where candidates lose the thread, because a term negated twice is simply itself again.

A worked case: take NOT (A.B + C). The outermost operator is the OR, so it becomes an AND and both terms are negated, giving NOT (A.B) . NOT C. Now apply the theorem again to the remaining bracket, and the answer is (NOT A + NOT B) . NOT C. Two steps, each one mechanical.

The commonest slip: changing the operators but forgetting to negate a term, or negating the terms and leaving the operators as they were. Half a transformation is worse than none, because the result looks finished. Check by testing one input combination in both forms before you move on.

The clerk's second sentence is therefore that the store may not be opened whenever the manager is absent or the register is unlocked, which is a different shape of rule entirely.

The Master Key: *To negate an expression, change every operator and negate every term, and do neither of those things by halves.*

REAL Question 40

High water on the Rufiji arrives faster than anyone would guess, and the siren above the spillway exists for those few minutes. A safety officer wrote its rule in plain words: they say when the alarm stays silent, not when it sounds.

Apply De Morgan's first theorem to the alarm condition below, and state the practical benefit of the transformed form.

The siren must sound unless both safe conditions hold at the same instant.

G = 1 when the spillway gate is confirmed at its safe position.

L = 1 when the water level is below the marked line.

Alarm = 1 when the siren sounds.

Each sensor already provides its own fault line, carrying NOT G and NOT L.

Model Answer

Written straight from the words the condition is Alarm = NOT (G.L), and De Morgan turns it into Alarm = NOT G + NOT L, a form needing no inverter after the logic and driven directly by the two fault lines the sensors already provide.

Putting the sentence into symbols: both safe conditions holding at once is G.L. The siren sounds unless that is true, so the alarm is its negation, giving Alarm = NOT (G.L). A faithful translation of the officer's sentence, and an awkward circuit.

Why the first form is awkward: it asks for an AND of two healthy signals followed by an inverter. The sensors do not offer healthy signals; they offer fault lines. Building this form means inverting each fault line back, ANDing the pair, then inverting the result, three inversions nobody wanted.

Applying the theorem: the negation sits over a product, so it may be carried inward provided the operator changes. Alarm = NOT (G.L) becomes Alarm = NOT G + NOT L. The alarm now reads as a plain OR of the two fault lines.

Checking it against the officer's words: suppose the gate is safe but the water is high, so G = 1 and L = 0. The first form gives NOT (1.0), which is 1. The second gives NOT 1 + NOT 0, which is 0 + 1, again 1. Both sound the siren, as the plain words demand.

The practical benefit: the transformed form maps onto hardware that already exists. Each fault line runs straight into one input of a single OR gate, so wiring, spares and fault tracing all become simpler in one stroke.

The benefit a safety engineer values most: each term in an OR is an independent cause. Tap NOT G to one lamp and NOT L to another and the panel names the fault instead of merely announcing one. The first form, hiding everything behind a single inverter, cannot.

What has not changed: the behaviour. De Morgan rearranges an expression and never alters the function it computes, so the alarm sounds in exactly the same instants as before. You have chosen a cheaper way of saying the same thing.

The safety officer described silence and the engineer needs noise, and one theorem carries the sentence across that gap without disturbing a condition.

The Master Key: *A negation left outside a bracket is a wiring problem, and De Morgan moves it inside so the signals you already have can drive the gate.*

REAL Question 41

Three claims are copied into a study group's shared notes in Iringa, each written out by a member who has plainly done the reading. Each one is close enough to the truth to pass a nod from the rest of the group.

Find the error in each of the three claims and state the correct position.

1. "NOT (A.B) = NOT A . NOT B."

2. "A NAND gate is the same as an AND gate with an inverted output, so it is also the same as an OR gate with an inverted output."

3. "Because NAND is universal, any circuit built from NAND gates uses fewer gates than the original."

Model Answer

All three are wrong: the first keeps the operator it was obliged to change, the second inverts the wrong side of the OR gate, and the third mistakes what a gate can do for what it costs.

The first error, the operator that did not change: De Morgan carries a negation inward and swaps the operator on the way, so the dot must become a plus. Corrected: NOT (A.B) = NOT A + NOT B. As written, the statement fails whenever exactly one input is high.

One row settles it: take A = 0 and B = 1. The true left side is NOT (0.1), which is 1. The student's right side is NOT 0 . NOT 1, which is 1 . 0, or 0. One disagreement is a disproof, and there are two among the four rows.

The second error, a right idea on the wrong side: the first half is correct, since a NAND is by definition an AND with its output inverted. The second half inverts the output of an OR, which gives a NOR. Corrected: a NAND also equals an OR gate with both inputs inverted.

Where that confusion comes from: bubble pushing does not leave a bubble where it found one. Move the output bubble of a NAND backwards and it lands on both inputs while the AND turns into an OR. The statement changed the gate and left the bubble behind.

The third error, possibility mistaken for economy: universality says every function can be built from NAND alone. It says nothing about how many gates that takes. Corrected: a NAND only version usually needs more gates than the mixed original, since one AND becomes two NANDs and one OR becomes three.

Why NAND is chosen in spite of that: not for gate count but for uniformity: one part to buy, one cell to lay out, one spare to stock. The saving lives in the storeroom and the fabrication mask, and it is paid for with extra gates on the board.

The habit that catches all three: test any claim on one row before believing it. Four rows exist for two variables, and one disagreement kills a statement outright, a cheaper check than any quantity of confident reasoning.

Each of these sentences is a true idea carried one step further than it can go, and one row of a truth table shows where the step went wrong.

The Master Key: *Change the operator when you move the negation, invert the inputs and not the output, and never confuse what is possible with what is cheap.*

HOT Question 42

The old customs house at Bagamoyo entered every transaction in its ledgers twice, as a receipt and as a debt, because two views of one event catch errors a single view hides. Boolean algebra owns a result that does the same for logic.

"De Morgan's theorems are the most useful result in the whole of Boolean algebra." To what extent do you agree?

Model Answer

The claim is largely defensible, because these theorems are the only rules carrying an expression across the boundary between AND and OR, but most useful is the wrong superlative for a result whose distinction is to enable the other tools rather than replace them.

For the claim, the only bridge between the operators: every other identity works inside one operator or holds a variable still. De Morgan alone converts a product into a sum. Without it AND and OR are separate countries, and no amount of factorising carries you from one to the other.

For the claim, universality depends on it: the proof that NAND and NOR can build any circuit runs entirely through these theorems. Since almost every chip ever fabricated rests on that fact, a result underwriting the manufacturing base of an industry has a strong claim to first place.

For the claim, it unblocks simplification: what usually stops an expression collapsing is a negation parked outside a bracket, where absorption and distribution cannot reach it. De Morgan gets that negation inside, after which the ordinary laws finally have something to bite on.

For the claim, its reach goes past hardware: the same two lines govern complements of sets, conditions inside a program, filters in a database query and the phrasing of a search. Few results you meet this year are used so widely by people who never learned the name.

Against the claim, transforming is not reducing: De Morgan almost never makes an expression shorter, because it rewrites rather than removes. The laws that delete terms are absorption, distribution and complementation, and a design closed with a Karnaugh map owes its gate count to them.

Against the claim, duality is the deeper idea: the two theorems are twins because the algebra itself is dual. Exchange AND with OR and 0 with 1 and every true statement stays true. Duality gives a second theorem free for every one you prove, a larger gift than either line alone.

The judgement: agree, with the superlative corrected. De Morgan is not the theorem that does the most work, and on raw simplification it is beaten. It is the theorem you cannot work around, since nothing else crosses between AND and OR, and a result every other technique leans on is better called the most enabling.

Rank results by what would fail in their absence rather than by how often they are written, and these two short lines come near the top.

The Master Key: *De Morgan is not the busiest law in Boolean algebra; it is the one with no substitute, which is a rarer thing.*

HOT Question 43

Mara Instruments assembles water meters in Musoma, and two quotations have landed on the manager's desk. One supplier offers chips of nothing but NAND gates cheaply; the other mixed AND, OR and NOT chips for more. The cheaper page is not the cheaper decision.

Evaluate the two options for Mara Instruments, considering gate count, inventory, propagation delay, board space and maintenance, and reach a judgement.

Model Answer

The NAND only chips are the better buy for a slow, field serviced product like a water meter, where inventory and maintenance savings are large and the timing penalty is invisible, but the choice would be wrong on a board whose longest chain must settle inside a tight clock.

For NAND only, one part number: purchasing, clearing paperwork, stores and the placement machine all shrink to one line item. Nothing halts a production run more reliably than the chip type that failed to arrive, and that risk falls by two thirds.

For NAND only, maintenance in the field: a technician sent to a pump house beyond Musoma carries one tube of chips and can replace any failed gate on the board. With mixed chips he carries three tubes or returns another day, and the journey costs more than the chips.

For NAND only, price and volume: one part bought in the factory's whole quantity attracts a better rate than three parts each bought in a third of it, so the quoted discount understates the real saving.

Against NAND only, gate count climbs: every OR becomes three NAND gates and every AND becomes two. A circuit using a dozen mixed gates may need twenty or more, so the cheaper chip is bought in greater numbers and part of the saving goes straight back.

Against NAND only, delay accumulates: propagation delay adds along the longest path, and replacing one OR with three NAND gates in series can treble the delay of that stage. In a meter, which samples slowly, this matters not at all. In a product with a fast clock it sets the ceiling.

Against NAND only, board space: more gates mean more packages, a larger board, more copper, more current and more solder joints. Board area is often a bigger line in the costing than the chips, and every joint is another candidate for failure.

The judgement: buy the NAND only chips for the meter, whose logic is shallow, whose clock is slow and whose lifetime cost sits mostly in service visits rather than silicon. Before signing, count the gates in the real design and measure the longest chain, and if a future product needs a fast clock keep one mixed part for those boards.

A unit price is the smallest number on a quotation and seldom the one that decides, because the true cost of a chip is settled in the storeroom and on the road long after the invoice.

The Master Key: *Universality lets one gate do every job, and whether it should depends on whether your cost lives in the silicon, the board or the service visit.*

Truth tables both ways: minterms, maxterms, sum of products and product of sums

A truth table and a Boolean expression carry the same information, and this section teaches the traffic in both directions. You will build a table from an expression by halving the input columns, then read an expression back off a table using minterms and maxterms. Sum of products comes from the rows that output one, product of sums from the rows that output zero, and the shorter form is whichever kind of row is rarer. Mechanical, unglamorous, and impossible to get wrong.

BINDER Question 44

A stock clerk on a sisal estate outside Korogwe rules her page into columns before she counts a single bale, and by the time she starts writing there is nowhere left for a bale to hide. A truth table is that page.

Explain how to construct a truth table from a Boolean expression, and build the table for Y = A.B + NOT C.

Model Answer

A truth table for n variables has two to the power n rows, the input columns are filled by halving so that no combination can go missing, and one working column is added for each part of the expression until the last column holds the output.

How many rows to rule: every variable is either 0 or 1, so each one you add doubles the count. One variable gives two rows, two give four, three give eight, four give sixteen. Rule two to the power n rows and stop, because there is no combination left over.

Filling the inputs by halving: the leftmost input column takes zeros for its top half and ones for its bottom half. The next column halves again, two zeros then two ones, repeated down the page. The rightmost alternates 0 and 1 on every line. Filled this way the rows count upwards in binary, so none can repeat and none can be lost.

One working column for each part: never try to hold a whole expression in your head. Y = A.B + NOT C has two parts, so give A.B a column of its own and NOT C another. Each column asks one small question that you can answer without thinking twice.

The last column is the answer: combine the working columns using the operator that joins them. Here the two parts are ORed, so Y is 1 on every row where either working column holds a 1, and 0 only where both hold zeros.

Checking your own table: read each column downwards rather than each row across. A.B must show a 1 in its last two rows and nowhere else. NOT C must alternate 1 and 0 the whole way down. A pattern down a column is visible at a glance; a mistake inside a single row is not.

A	B	C	A.B	NOT C	Y
0	0	0	0	1	1
0	0	1	0	0	0
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	1	0	1

Table 1.10: *Building a truth table column by column.*

The whole method is bookkeeping, and bookkeeping done in a fixed order is the one part of this course that cannot betray you.

The Master Key: *Halve each input column in turn and the rows write themselves, complete and in order, every single time.*

BINDER Question 45

A roll call at a boarding school near Singida can be taken two ways. The teacher can name the one student present, or the one student absent, and either sentence pins down the same room. Boolean algebra keeps both habits and names them.

Define minterm and maxterm, explain the difference between them, and write both out for all four combinations of two variables.

Model Answer

A minterm is a product term that takes the value 1 for exactly one input combination and 0 for every other, while a maxterm is a sum term that is 0 for exactly one combination and 1 for every other, so the two are opposite ways of singling out the same row.

What makes a minterm: an AND of every variable in the function, each appearing once, plain or negated. Because it is an AND, a single 0 anywhere kills the whole product, so exactly one combination survives and the term is 1 there alone.

What makes a maxterm: an OR of every variable, each appearing once, plain or negated. Because it is an OR, a single 1 anywhere rescues the whole sum, so exactly one combination fails and the term is 0 there alone.

The naming convention: number the rows from the top starting at 0, reading each row as a binary number. The minterm of row 5 is m5 and its maxterm is M5. Small m for the product, capital M for the sum, and the number is the decimal value of that row of inputs.

Writing a minterm from a row: take the row and write each variable plain if its input is 1 and negated if it is 0. Row A = 1, B = 0 gives A.NOT B, and feeding that row in makes both factors 1, so the product is 1 there and nowhere else.

Writing a maxterm, with the rule reversed: for a maxterm the convention flips. Write the variable negated if its input is 1 and plain if it is 0, then OR the lot. Row A = 1, B = 0 gives NOT A + B, which is 0 there and 1 everywhere else. Get these two rules the right way round and everything after them follows.

A	B	Minterm	Maxterm
0	0	NOT A.NOT B	A + B
0	1	NOT A.B	A + NOT B
1	0	A.NOT B	NOT A + B
1	1	A.B	NOT A + NOT B

Table 1.11: *Minterms and maxterms for two variables.*

Every row of every truth table therefore carries two names, and which you reach for depends only on whether that row holds a one or a zero.

The Master Key: *A minterm is true in one place and a maxterm is false in one place, and the negation rule runs opposite ways in the two.*

BINDER Question 46

A clerk at Ifakara skips the blank lines in the clinic register and copies out only the entries that carry a name, because the blanks describe nothing and the entries describe the day. A sum of products is that habit applied to logic.

Explain how to read a sum of products expression off a truth table, using the function that outputs 1 on the last three rows of a two variable table.

Model Answer

To read a sum of products off a truth table you take every row whose output is 1, write the minterm for that row, and OR those minterms together, which for the function that outputs 1 on the last three rows of a two variable table gives Y = NOT A.B + A.NOT B + A.B.

Why only the rows that output one matter: an OR is 1 as soon as any of its terms is 1. Give every 1 in the output column a minterm of its own and the expression switches on at exactly those rows, while every other row leaves all the terms at 0.

The three steps: mark every row where the output is 1, convert each marked row into its minterm by negating the inputs that are 0, then OR the minterms together. There is no fourth step and no judgement anywhere in it.

The worked case: the function is 1 on rows 1, 2 and 3. Row 1 has A = 0 and B = 1, so its minterm is NOT A.B. Row 2 gives A.NOT B and row 3 gives A.B. ORing the three produces Y = NOT A.B + A.NOT B + A.B, correct before anybody has had an idea.

What it looks like as gates: one AND gate for each minterm, inverters on the inputs that were 0, and a single OR gate collecting every AND output. The circuit is two layers deep whatever the function, which is why the form is called canonical.

Canonical is not minimal: this function simplifies. A.NOT B + A.B factors to A.(NOT B + B), which the complement and identity laws reduce to A. That leaves NOT A.B + A, which the distributive law turns into A + B. Three AND gates and two inverters become one OR gate.

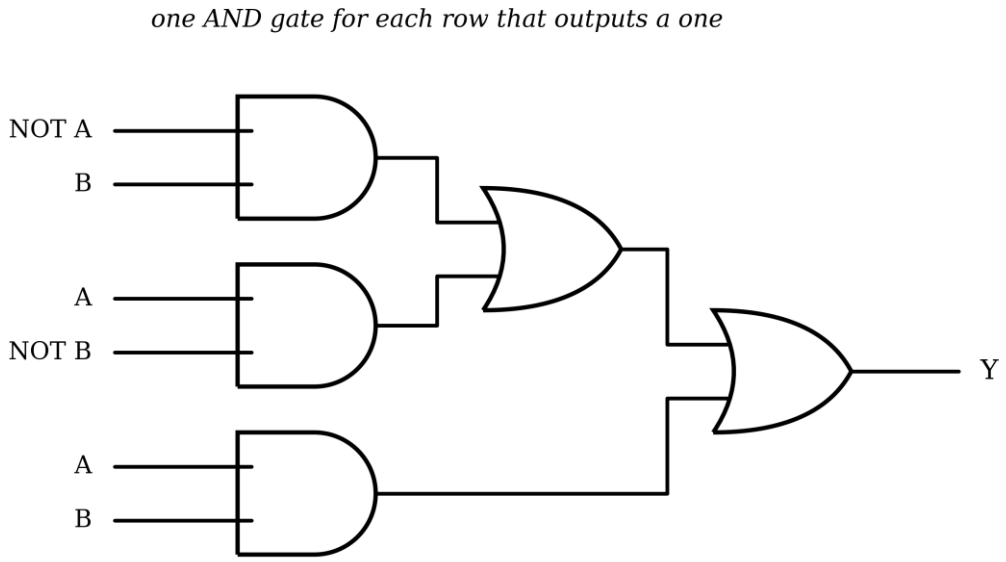


Figure 1.11: *A sum of products drawn straight from the truth table, one AND gate for each row that outputs a one.*

Reading the ones hands you an expression you can trust before you have had a single idea, and ideas are what the simplification afterwards is for.

The Master Key: *Every one in the output column earns a minterm, and the sum of those minterms is the function exactly.*

BINDER Question 47

The gate on the Mikumi road keeps a tally of the vehicles it turns back rather than the ones it waves through, because on an ordinary day the refusals are few and the passes are not. Product of sums is that same economy.

Explain how to read a product of sums expression off a truth table, and state when this form is shorter than a sum of products.

Model Answer

To read a product of sums you take every row whose output is 0, write the maxterm for that row, and AND those maxterms together, and the form is the shorter of the two whenever the output column holds fewer zeros than ones.

Why the zeros are the rows that matter: an AND is 0 as soon as any one of its terms is 0. Give every 0 in the output column a maxterm of its own and the expression switches off at exactly those rows, holding at 1 everywhere else, which is the whole of the function.

The three steps: mark every row where the output is 0, convert each marked row into its maxterm by negating the inputs that are 1, then AND the maxterms together, writing each one inside its own bracket with dots between.

The same function written both ways: the function that is 1 on the last three rows of a two variable table has exactly one zero, at A = 0 and B = 0. Its maxterm is A + B, so the whole product of sums is the single term A + B. Sum of products needed three terms for that identical function.

The rule for choosing: count the ones and count the zeros before you write anything. Few ones, use sum of products. Few zeros, use product of sums. A table of eight rows holding two zeros gives a product of two brackets against a sum of six product terms.

Both forms as gates: a product of sums is one OR gate for each maxterm feeding a single AND gate, still two layers deep. The choice between the forms therefore costs nothing structurally and saves gates whenever you have counted the rarer kind of row.

The two forms are the same instruction given from opposite ends, and the output column tells you at a glance which end you are standing nearer to.

The Master Key: *Read the ones for sum of products and the zeros for product of sums, and always read whichever list is shorter.*

REAL Question 48

A cold store at the fishing port in Mtwara holds prawns for export at minus eighteen, and the buyer overseas pays only for a consignment whose cold chain never broke. So its alarm is deliberately nervous. It stays quiet only when nothing is wrong.

Construct the truth table for the alarm, write both the sum of products and the product of sums, and state which form is shorter and why.

The alarm sounds whenever at least one of three sensors reports a fault.

Let D be 1 when the door contact reports the door standing open.

Let T be 1 when the probe reports the store above minus eighteen degrees.

Let P be 1 when the monitor reports that the mains supply has failed.

Let A be 1 when the alarm sounds.

Model Answer

The alarm is A = D + T + P, so its table holds seven ones and a single zero, which means the sum of products needs all seven minterms while the product of sums needs the one maxterm D + T + P and is finished in three symbols.

The condition in Boolean form: at least one means OR. Any fault on its own is enough to sound the alarm, and two faults together are still faults, so A = D + T + P before a table has been ruled. The table now has something definite to confirm.

Building the table: three variables give eight rows. Fill D, T and P by halving, then work along each row asking only whether any input is 1. Every row answers yes except the top one, where all three sensors report healthy.

The sum of products: seven rows output 1, so the canonical sum carries seven product terms, m1 through to m7. It opens NOT D.NOT T.P + NOT D.T.NOT P and closes D.T.NOT P + D.T.P, twenty-one literals in all, and every one of them correct.

The product of sums: one row outputs 0, the row where D, T and P are all 0. Its maxterm negates the inputs that are 1 and there are none of those, so the maxterm is D + T + P. The canonical product is that single sum term and nothing else.

Which is shorter, and why: the product of sums, by seven terms to one. The reason is the split in the output column. Seven ones against one zero means that counting the zeros is counting one thing while counting the ones is counting seven, and each form only ever writes out what it counted.

The check that matters: the product of sums came out identical to the expression the words gave in the first line. The seven term sum describes the same function, but recovering D + T + P from it would take several applications of the complement law to get back to where you already were.

What the difference costs on the wall: three sensor wires into one OR gate, against seven three-input AND gates, three inverters and a seven-input OR. The store gets exactly the same alarm from either panel, and one of the two fits inside a matchbox.

D	T	P	Alarm A
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Table 1.12: *The Mtwara cold store alarm, seven ones and one zero.*

A nervous alarm is almost always sounding, and an output that is almost always at one is a product of sums waiting to be read off in a single line.

The Master Key: *Count the zeros before you start writing, because a function with one zero has a one term product of sums and a monstrous sum of products.*

REAL Question 49

A trainee at a gold plant near Geita is handed eight rows of a table and told to reproduce the logic they describe, with no hint of what the circuit was for. The table is the whole of the evidence. It is also enough.

Derive the sum of products expression from the table, then simplify it algebraically, naming the law used at every step.

Model Answer

The five rows that output 1 give Y = NOT A.NOT B.C + NOT A.B.C + A.NOT B.C + A.B.NOT C + A.B.C, and six named steps reduce that to Y = C + A.B.

Reading the sum of products off: five rows hold a 1, at rows 1, 3, 5, 6 and 7. Each becomes a minterm with its 0 inputs negated, and ORing the five gives the canonical form, fifteen literals spread across five product terms.

Step one, the idempotent law: X + X = X, read backwards, allows a term to be written twice without changing the function. Write A.B.C a second time. This looks like moving the wrong way, and it is the move the whole simplification depends on, because that minterm has to pair with two different partners.

Step two, the distributive law: factor the six terms in pairs. NOT A.NOT B.C + NOT A.B.C becomes NOT A.C.(NOT B + B). A.NOT B.C + A.B.C becomes A.C.(NOT B + B). A.B.NOT C + A.B.C becomes A.B.(NOT C + C).

Step three, the complement law: X + NOT X = 1, so NOT B + B is 1 and NOT C + C is 1. Every bracket left standing after the factoring closes to a plain one.

Step four, the identity law: X.1 = X, which deletes each of those ones and leaves Y = NOT A.C + A.C + A.B. Fifteen literals have become six.

Steps five and six, the same two laws again: the distributive law factors NOT A.C + A.C into C.(NOT A + A), the complement law closes that bracket to 1, and the identity law drops it. What remains is Y = C + A.B.

Confirming against the table: C + A.B is 1 wherever C is 1, which accounts for rows 1, 3, 5 and 7, and it is also 1 when A and B are both 1, which adds row 6. Five ones, the same five as the table, so nothing has been lost on the way down.

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 1.13: *The table handed to the trainee at Geita.*

The trainee never learned what the circuit was for and never needed to, because a complete table already holds everything the original designer knew.

The Master Key: *Write a minterm twice when it must serve two pairings, because the idempotent law is the only way out of a five term sum.*

REAL Question 50

Three claims go up on a study group's board in Tabora, copied out of a friend's notes from another region and repeated with enough confidence that nobody tests them. Each is a real idea remembered slightly wrong, the most durable kind of error.

Identify the error in each of the three claims, correct it, and explain why the correction is right.

Claim one: A truth table for four variables has eight rows.

Claim two: To write the sum of products you take the rows where the output is 0.

Claim three: In a minterm, an input that is 1 appears negated.

Model Answer

All three are wrong: four variables give two to the power four, that is sixteen rows; sum of products is read from the rows where the output is 1 and it is product of sums that reads the zeros; and in a minterm it is the 0 inputs that appear negated, with the rule reversed only for maxterms.

Claim one, where the eight came from: eight is the row count for three variables, so the student has memorised a number in place of a rule. The rule is two to the power n. Two to the power four is sixteen, so the claim is out by half.

Why the count doubles: adding a variable takes every row you already had and splits it in two, one copy with the new input at 0 and one with it at 1. Sixteen rows for four variables, thirty-two for five, and that growth is why large tables stop being practical.

Claim two, the two forms swapped: sum of products reads the rows that output 1 and ORs their minterms. Product of sums reads the rows that output 0 and ANDs their maxterms. The claim has taken a true statement and attached it to the wrong form.

Why a sum has to be built from the ones: an OR is 1 as soon as one term is 1, so terms built from the zero rows would switch the circuit on at precisely the rows the table says off. What the claim actually produces is the complement of the function that was wanted.

Claim three, the negation inverted: in a minterm a 0 input appears negated and a 1 input appears plain, so the row A = 1, B = 0 gives A.NOT B. The claim has described the maxterm rule instead, where it really is the 1 inputs that are negated.

Why that rule must run the way it does: a minterm has to be 1 on its own row. Feed A = 1, B = 0 into A.NOT B and both factors come out at 1, so the product is 1. The claimed version gives NOT A.B, which is 0 on that very row and 1 on a different one.

One line that fixes all three: rows double with every variable, ones make products and zeros make sums, and the negation sits on whichever input would otherwise spoil the term. All three corrections fall out of that one sentence.

None of the three came from ignorance. Each is a true statement about the neighbouring idea, which is why saying them out loud is the only reliable way to catch them.

The Master Key: *Ones make minterms and zeros make maxterms, and the negation always sits where it must to make that one row the exception.*

HOT Question 51

A supervisor in Mpanda watches a student fill three quarters of a page with minterms, cross almost all of it out again, and arrive at an expression four symbols long. It is fair to ask whether that page was work or ceremony.

"Canonical forms are a waste of effort because every one of them simplifies." To what extent do you agree?

Model Answer

The claim is right that a canonical form is almost never the form you keep and wrong that this makes the effort wasted, because its value was never brevity but certainty: it is the only route from a table to an expression that asks for no insight at all and therefore cannot quietly fail.

The case for the claim, on sheer length: a canonical sum writes a full product term for every 1 in the output column, and each term names every variable whether it matters to that row or not. Five rows of three variables produce fifteen literals to describe something that may need three.

The case for the claim, on built-in redundancy: the form is built to be reducible. Any two rows differing in one variable collapse by the complement law, and a canonical sum is full of such pairs, so the writing out and the crossing out are two halves of one avoidable exercise.

The case for the claim, on hardware: nobody builds a canonical circuit. Seven three-input AND gates feeding a seven-input OR is not a design but a transcription, and any synthesis tool would discard it before its first pass finished. As a finished product the form has no defenders.

The case against, on what it actually is: a canonical form is not a good answer that fell short. It is a mechanical translation of a table, produced by a rule containing no choices, and it guarantees that the expression agrees with the table on every row.

The case against, on what the alternative costs: the alternative is to look at the table and see the pattern. When the pattern is there, that is quicker. When it is not, you stall, or write something shorter that is subtly wrong and leave no working to show where it went astray.

The case against, under pressure: a method that depends on insight fails hardest on the morning insight will not come. A canonical form is available while frightened, while tired, and in the last ten minutes of a long paper, and it earns credit even when the simplification after it is unfinished.

The judgement: agree with the description and reject the word waste. Every canonical form does simplify, and that is the plan rather than the flaw. You buy a guaranteed correct starting point cheaply, then spend skill reducing it. Effort spent making failure impossible is not waste, it is what insurance means.

Ask what the candidate would have put on the page instead of those minterms. Often enough the honest answer is nothing at all, and nothing does not simplify.

The Master Key: *A canonical form is not meant to be kept, it is meant to be certain, and certainty is the only thing worth owning before simplification starts.*

HOT Question 52

Two technicians on Pemba are given the same six sentences of rules for a clove drier. One rules sixteen rows and starts filling them in. The other writes an expression in a minute. What matters is which would notice a case the sentences missed.

Compare deriving a Boolean expression from a truth table with deriving it by reasoning about the problem directly, and judge which a candidate should use and when.

Model Answer

Direct reasoning is far quicker and produces a short expression at once, while a truth table is slow but exhaustive and cannot silently omit a case, so the choice turns on reliability rather than speed: reason directly when the conditions are few and plainly worded, and go through the table whenever the wording is tangled or a missed combination would be expensive.

How each method works: the table route lists every input combination, decides the output for each one, and reads a canonical form off the finished column. Direct reasoning turns each sentence of the requirement into a term and joins the terms with AND or OR according to what the words say.

Speed: direct reasoning wins outright and by a wide margin. Three variables mean eight rows to rule and fill against perhaps two lines of writing, and at four variables the table doubles to sixteen rows while the reasoning takes no longer than it did before.

Reliability: the table wins outright. Its answer is derived from a complete list of possibilities, so it is correct by construction rather than by judgement. Direct reasoning is correct only if the reasoning was, and nothing on the page tells you whether it was.

The risk of missing a case, which is the real difference: a table carries a row for every combination, including the awkward ones nobody thought to mention, and each row demands a decision before you may move on. Direct reasoning covers the cases the requirement discusses and says nothing about the rest, so an unstated combination leaves no visible gap.

How the two methods fail: a table fails visibly, through a copied bit or a mis-ruled column, and the fault sits on the page where a second reading will find it. Direct reasoning fails invisibly, producing a fluent short expression that behaves wrongly on one combination out of sixteen.

What the marks reward: a question that hands you a table is asking for minterms and named laws. A question that gives a scenario in words is asking for the conditions translated. Answer with the method the question is built around and the working earns credit even where the final expression slips.

The judgement: reason directly first, because it is fast and it gives you something to test, then rule the table whenever three variables or fewer make it cheap, and always when the requirement contains the word unless or any condition with an exception buried inside it. Where the two disagree, the table is right.

The technician who finished early was probably correct. Being probably correct is a different profession from engineering, and a different grade from the top one.

The Master Key: *Reasoning finds an answer quickly and a table proves there was nothing the answer forgot.*

Karnaugh maps

A truth table tells you everything and shows you nothing. Redraw its rows as a grid whose neighbouring squares differ in exactly one variable, and the pattern of the ones becomes visible; simplification stops being a search and becomes an act of looking. That grid is the Karnaugh map. This section builds it from nothing: why the columns run 00, 01, 11, 10, how groups are formed and read, how the edges wrap, what to do with combinations that can never occur, and where the method finally runs out.

BINDER Question 53

The signal box at Tabora keeps its rules in a ledger, one line for every lever setting, and the new signalman reads it downward like a shopping list. His trainer copies those lines onto a grid of eight squares, and the pattern is obvious.

Explain what a Karnaugh map is, and explain why its columns are labelled in the order 00, 01, 11, 10 rather than in counting order.

Model Answer

A Karnaugh map is a truth table redrawn as a grid of squares, one square for every input combination, arranged so that any two squares sharing an edge differ in exactly one variable, and the columns run 00, 01, 11, 10 because that Gray code ordering is the only ordering which keeps that promise all the way round.

The map is a truth table in a new shape: nothing is added and nothing is discarded. A three variable table has eight rows, so the map has eight squares, and each square holds the output that its own row carried. The information is identical. Only the geometry has changed.

Why geometry is worth the trouble: two product terms collapse into one only when they differ in a single variable, since X.Y + X.NOT Y = X. In a column of rows those partners may sit anywhere, and finding them is a hunt. On a map they have been seated side by side before you begin.

What the ordering has to guarantee: each step from one column label to the next must change one bit and no more. Counting order fails immediately, because the step from 01 to 10 changes both bits at once, and two squares differing in two variables have nothing whatever to combine.

Gray code ordering: 00, then 01, then 11, then 10. One bit moves at every step, and the step from the last label back to the first, 10 to 00, moves one bit as well, so the row of columns closes into a ring instead of stopping like a line.

What the ordering buys you: adjacency on paper now means adjacency in the algebra. Looping neighbouring ones by eye performs the collapsing theorem without your writing a line of it, and that single correspondence is the reason the map exists at all.

Karnaugh contributed no new theorem, only a seating plan, and the seating plan is what allows the eye to do the algebra.

The Master Key: *A Karnaugh map is a truth table seated in Gray code order, so that squares which touch hold terms that combine.*

BINDER Question 54

An apprentice in a Korogwe workshop is handed eight lines of a truth table and one empty grid, and told that the second is the first. She carries the numbers across, and three ones settle into a shape she had not suspected.

Describe how a three variable Karnaugh map is filled from a truth table, and explain how a group of ones is read as a simplified term.

Model Answer

Fill the map by taking each row of the table in turn, letting its A value choose the map row and its B and C values choose the column carrying that pair of bits, then read any group by keeping the variables that stay constant and discarding those that change.

Draw the frame before anything else: two rows, A = 0 above and A = 1 below, and four columns labelled BC = 00, 01, 11 and 10. Eight squares for eight rows, so nothing is left over and nothing is entered twice.

Placing one row: take the row A = 1, B = 0, C = 1, whose output is 1. Its A value sends you to the lower row of the map, its B and C values send you to the column headed 01, and the 1 is written in that square. Table 1.14 does this for all eight rows.

Never leave a square empty: a blank square is not a zero but an unanswered question, and it will later be grouped by accident or overlooked by accident. Write every zero out and the map becomes a complete statement of the function.

A	B	C	Y	The square it fills
0	0	0	0	row A = 0, column BC = 00
0	0	1	1	row A = 0, column BC = 01
0	1	0	0	row A = 0, column BC = 10
0	1	1	0	row A = 0, column BC = 11
1	0	0	1	row A = 1, column BC = 00
1	0	1	1	row A = 1, column BC = 01
1	1	0	0	row A = 1, column BC = 10
1	1	1	0	row A = 1, column BC = 11

Table 1.14: *The eight rows behind figure 1.12, and the square each one fills. The table counts upward; the map does not.*

Reading the first group: figure 1.12 loops the two squares at A = 1 with BC = 00 and BC = 01. A is 1 in both and B is 0 in both, while C is 0 in one and 1 in the other, so C is discarded and what remains reads A.NOT B.

Reading the second group and writing the answer: the other loop joins the square at A = 0, BC = 01 to the square at A = 1, BC = 01, where B = 0 and C = 1 throughout while A changes, giving NOT B.C. Now OR the groups: Y = A.NOT B + NOT B.C, which factors to Y = NOT B.(A + C), one inverter, one OR gate and one AND gate.

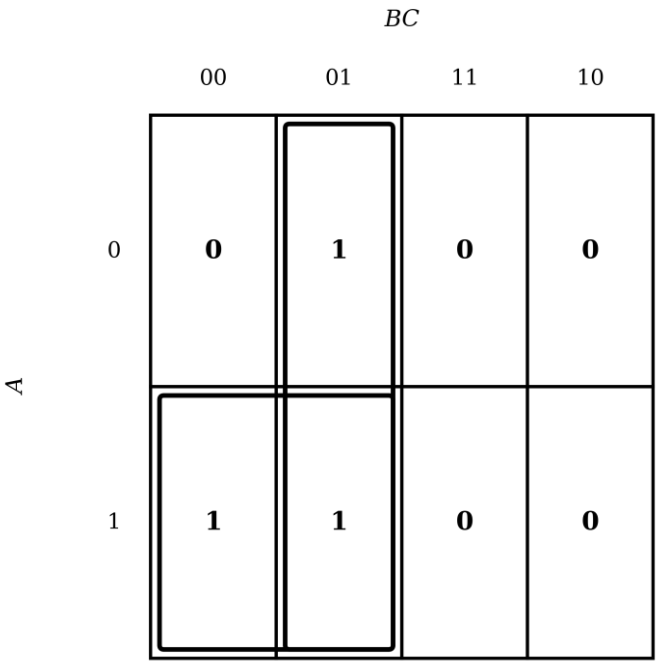


Figure 1.12: *A three-variable map. Two overlapping groups of two give Y = A.NOT B + NOT B.C.*

The map is filled by a clerk and read by an engineer, and the reading rule never grows longer than this: keep whatever stays the same.

The Master Key: *Every square is one row of the table, and every group is the product of the variables that do not change across it.*

BINDER Question 55

The lake at Babati keeps its own shoreline, and no argument moves it by a metre. Grouping on a Karnaugh map is governed as firmly. Six rules decide which loops may be drawn, and guessing produces an expression that looks simplified and is wrong.

State the rules that govern the forming of groups on a Karnaugh map, and explain what each rule is for.

Model Answer

A group may contain ones only; its size must be a power of two, that is 1, 2, 4, 8 or 16; it must be as large as the ones allow; groups may overlap; every one must lie in at least one group; and the map wraps at the left and right edges and at the top and bottom.

Ones only, and every one covered: these two keep the expression true. A group holding a zero claims a term is satisfied where the function is not; an uncovered one leaves an input where the circuit says zero and the function says one.

Sizes are powers of two: doubling a group deletes exactly one variable, because doubling is what lets that variable take both values while the rest stand still. Only sizes reached by doubling delete a whole variable, so three adjacent ones are a two and a one.

As large as the ones allow: every doubling removes another literal, so one group of four is cheaper than two groups of two. Stopping at small loops gives an answer that is correct and needlessly expensive, the commonest way marks are lost here.

Overlapping is free: a one already in one loop may be reused to enlarge a second, and nothing is spent, because groups are OR-ed and A + A = A. In figure 1.12 the square at A = 1, BC = 01 lies in both loops, which is what lets both reach full size.

The edges wrap: the left column touches the right and the top row touches the bottom, because Gray code labelling makes those outer pairs differ in one variable. A map is a ring in each direction, and a loop crossing an edge is as legitimate as any other.

Two of the six rules keep the expression true and four keep it short, and the two that keep it true are worth checking square by square before you leave.

The Master Key: *Group ones only, in powers of two, as largely as the ones allow, overlapping freely and across the edges, until every one is covered.*

BINDER Question 56

Sail east from Mafia for long enough and you arrive back at Mafia, which is a fact about the shape of the world. A Karnaugh map has that shape too, and its four corner squares, as far apart as squares can be, are neighbours.

Explain why the edges of a Karnaugh map wrap, and show how the four corners of a four variable map form a single group of four.

Model Answer

The edges wrap because Gray code labelling makes the first and last label on each axis differ in exactly one bit, so the outer columns and outer rows are neighbours in the algebra however far apart the printing sets them, and the four corners of a four variable map form one legitimate group of four.

Where the wrap comes from: the columns run CD = 00, 01, 11, 10. Set the last against the first, 10 against 00, and they differ in C alone. The rows run AB = 00, 01, 11, 10, and 10 against 00 differ in A alone. Adjacency belongs to the labels, never to the ink.

The map is a ring in each direction: roll the sheet until the right edge meets the left and you hold a cylinder; roll it again the other way and you hold a ring with no edges at all. Every square then has four neighbours, corners included.

The four corners: in figure 1.13 the ones sit at AB = 00 with CD = 00 and CD = 10, and at AB = 10 with CD = 00 and CD = 10. Wrapping left to right pairs them across each row; wrapping top to bottom joins the two pairs into one group of four.

Reading the group: look for what never changes across the four squares. B is 0 in all of them and D is 0 in all of them, while A changes and C changes. Both changing variables leave, and the group reads Y = NOT B.NOT D.

Why the size is worth chasing: four cells is two doublings, so two variables go. Refuse the vertical wrap and you are left with NOT A.NOT B.NOT D + A.NOT B.NOT D, six literals across two AND gates and an OR gate, where one group of four needs two inverters and a single AND gate.

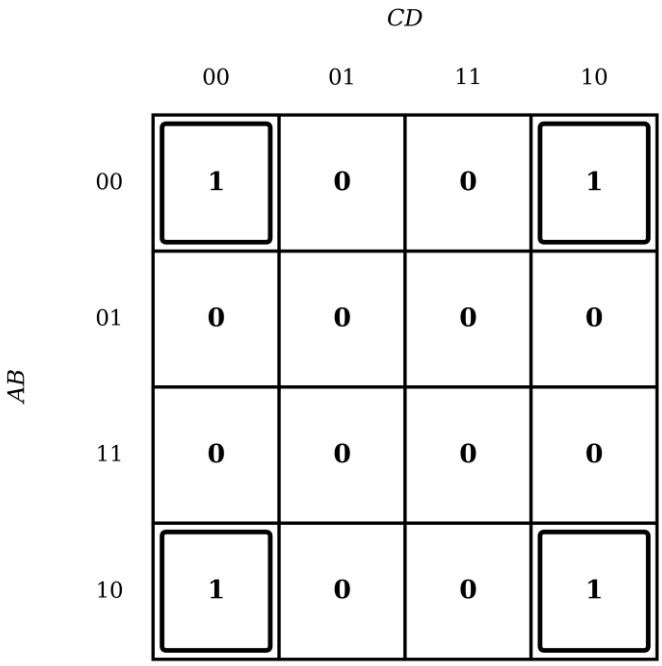


Figure 1.13: *The four corners form one group of four, because the map wraps at both edges. Y = NOT B.NOT D.*

Nothing about a Karnaugh map is rectangular except the printing, and a candidate who forgets it leaves the largest group on the page uncollected.

The Master Key: *The first and last labels on an axis differ in one bit, so the map has no edges and its corners are neighbours.*

REAL Question 57

Paddy arrives wet at the Ifakara drying shed, and the extraction fan is all that separates a good crop from a sour one. The engineer left a sentence saying when the fan may be left off. The controller needs the other sentence, in gates.

Build the truth table for the fan, plot and group the Karnaugh map, write the simplified expression, and state the saving in gates against the unsimplified form.

A rice drying bin at Ifakara, with three sensors feeding one fan controller.

A = 1 when the bin is filled to or above the working line.

B = 1 when the paddy is above target moisture.

C = 1 when the burner is alight.

Y = 1 when the extraction fan runs.

The fan may be left off in exactly three situations: the burner is out and the bin is below the working line, whether the paddy is wet or dry; and the burner is out with a full bin of dry paddy. In every other combination the fan must run.

Model Answer

The three permitted silences are A = 0, B = 0, C = 0, then A = 0, B = 1, C = 0, then A = 1, B = 0, C = 0, so the fan runs on the other five rows, and the map gathers those five into a block of four and a pair, giving Y = C + A.B, two gates where the raw form needs nine.

Turning the sentence into rows: the engineer described silence, so write the zeros first. Burner out with the bin below the line covers A = 0, B = 0, C = 0 and A = 0, B = 1, C = 0. Burner out with a full bin of dry paddy adds A = 1, B = 0, C = 0. The five rows not named carry a one.

Plotting the five ones: two rows for A, four columns for BC in the order 00, 01, 11, 10. The ones land at BC = 01 and BC = 11 in the upper row, and at BC = 01, 11 and 10 in the lower. The two middle columns are full, and the lower row runs one square further right.

The large group: those two full columns form a block of four. Across all four, C is 1, while A changes between rows and B between columns. Two variables leave and the group reads C, one literal covering four of the five ones.

The group that finishes the job: the one at A = 1, BC = 10 is still outside every loop. Take it with its neighbour at A = 1, BC = 11, where A = 1 and B = 1 while C changes, and the pair reads A.B. Every one is now covered.

The expression, read back into the shed: Y = C + A.B. The fan runs whenever the burner is alight, and whenever a full bin still holds wet paddy with the burner out. That is the engineer's sentence turned the right way round, and an operator can repeat it from memory.

Checking it by a second route: the off condition was NOT C.(NOT A + NOT B), so the fan runs when that is false: Y = NOT (NOT C.(NOT A + NOT B)), which De Morgan turns into C + A.B. Map and algebra arrive together, the only check worth the name.

The saving in gates: unsimplified, each of the five ones needs a three input AND gate, with an OR gate to collect them and three inverters for NOT A, NOT B and NOT C: nine gates. Y = C + A.B needs one AND gate, one OR gate and no inverter. Seven gates go, with the delay and solder joints they carried.

The engineer wrote down when the fan could rest; five rows, one grid and two loops later the shed has a circuit that knows when it must work.

The Master Key: *Translate the sentence into rows, plot every row, and take the largest block first, because it is often a single variable.*

REAL Question 58

The tower above Singida carries three float switches, low, middle and high, and the water inside has one surface. That fact rules out half the combinations the switches can report. A controller built for all eight does careful work for cases that never arrive.

Explain what a don't-care condition is, and use the don't-cares in this system to obtain a shorter expression for the pump.

A storage tower at Singida with three float switches on a single stem.

L = 1 when the water is above the low float.

M = 1 when the water is above the middle float.

H = 1 when the water is above the high float.

P = 1 when the fill pump runs.

The pump must run whenever the water has fallen below the middle float.

One stem in one column of water: a higher float can never be wet while a lower float is dry.

Model Answer

A don't-care is an input combination the hardware can never present, marked X on the map and free to be read as a one or a zero, whichever helps; four of these eight combinations are impossible, and reading two of them as ones turns P = NOT M.NOT H into the single literal P = NOT M.

What a don't-care is: an input combination the physical system cannot produce, so its output is never observed and cannot be wrong. It is written X rather than 0 or 1 because nobody will ever be able to check it.

Finding them in the tower: one surface of water means the floats go wet from the bottom upward, so LMH = 000, 100, 110 and 111 are the reachable states. The other four, 001, 010, 011 and 101, report a wet float sitting above a dry one, and no tower does that.

The map as it stands: rows for L, columns for MH in the order 00, 01, 11, 10. The upper row, L = 0, holds a one at MH = 00 and an X in its other three squares. The lower row, L = 1, holds a one at MH = 00, an X at MH = 01, and zeros at MH = 11 and MH = 10.

Grouping with every X read as zero: the timid reading treats every X as a zero. The two ones then form a pair in the column MH = 00, where M = 0 and H = 0 while L changes, giving P = NOT M.NOT H: an AND gate and two inverters.

Grouping with the useful X read as one: the squares at MH = 01 are X in both rows. Read them as ones and the loop extends across the two leftmost columns into a block of four. M is 0 throughout while L and H both change, so two variables leave and it reads P = NOT M: one inverter, and nothing else.

A don't-care is never obliged to be covered: the X at MH = 11 and the X at MH = 10 in the upper row stay outside every loop, and that is not an oversight. Ones must be covered and zeros excluded, but an X is only an offer: accept it when it enlarges a group, decline it when it does not.

Why the tower still needs three floats: the map shortened one output, not the instrument. L still drives the dry running cutout and H the overflow alarm, each with its own map. Simplification removes gates from an expression; it never removes a sensor from a tower.

Half of this map describes a tower that cannot exist, and the honest thing to do with an impossible square is to spend it.

The Master Key: *A don't-care costs nothing either way, so read it as a one when it grows a group and as a zero when it does not.*

REAL Question 59

Three sentences are read out in a Kahama study group, each by a member who has been drawing maps all week and is nearly right. Nearly right is the dangerous distance. It survives the nodding, and it is copied into three sets of notes.

Identify what is wrong with each of the three claims and correct it.

1. "A group of three adjacent ones is allowed if all three are next to each other."

2. "Groups may never overlap, because that would count a one twice."

3. "A four-variable map has no wrapping, only a three-variable map does."

Model Answer

All three are wrong: group sizes must be powers of two, so three is not a legal size at all; overlapping is not merely permitted but usually necessary; and every map wraps at both pairs of edges, the four variable map more visibly than any other.

The first error, a size that is not a power of two: each doubling of a group removes exactly one variable, so the legal sizes are 1, 2, 4, 8 and 16. No number of doublings produces three, and a loop drawn round three squares corresponds to no product term whatever. Corrected: three is not a group size.

What to do with three adjacent ones instead: cover them with a group of two and a group of one, or better, find two groups of two that overlap on the middle square. Figure 1.12 is exactly this case: three ones, two overlapping pairs, nothing stranded and nothing illegal drawn.

The second error, overlapping forbidden: corrected: groups may overlap freely, and overlapping is often the only way to make every loop as large as the ones permit. A worker who refuses to overlap will keep producing correct expressions that are longer than the map was offering.

Why a one may sit in two loops: the groups are OR-ed together, and A + A = A. A square inside two loops contributes to two terms, and wherever both are true the OR still returns one. Nothing is double counted, because at no point is anything being counted.

The third error, wrapping restricted to three variables: corrected: every Karnaugh map wraps at the left and right edges and at the top and bottom. Wrapping follows from Gray code labelling, and a four variable map uses Gray code on both axes, so if anything it wraps more than a three variable map, which wraps only sideways.

The proof already printed: figure 1.13 shows four ones in the four corners forming one group of four that reads NOT B.NOT D, two literals. Deny the wrapping and those become four isolated squares, four terms of four literals each, sixteen literals in place of two.

The habit that catches all three: before drawing a loop, count its squares and ask whether that count is 1, 2, 4, 8 or 16. After drawing them all, sweep the map twice: once for a one left uncovered, once for a loop that could have been larger. Every claim in the box fails one of those checks.

Each of these sentences guards its author against a danger that does not exist, and each one charges the circuit extra literals for the protection.

The Master Key: *Powers of two only, overlap freely, and remember that every map is a ring in both of its directions.*

HOT Question 60

Two answers to one question reach a teacher in Mpanda. The first is a grid, two loops and a single line. The second spends half a page on the distributive law and arrives one step longer. Only one could have known when to stop.

Compare the Karnaugh map with algebraic simplification, assessing speed, reliability, the risk of stopping too early, and the behaviour of each as the number of variables grows, and reach a judgement about which a candidate should reach for first.

Model Answer

For four variables or fewer the map wins on speed and wins decisively on reliability, because it carries a stopping rule that algebra does not have; algebra keeps its value for larger problems, for forms other than the sum of products, and for the final factoring the map cannot see.

For the map, speed at small sizes: eight or sixteen squares are filled in under a minute, and the groups are seen rather than searched for. There is no decision about which law to try next, because looking has replaced choosing.

For the map, a stopping rule: the map states when you are finished: every one covered, no loop that could have been larger. Algebra offers nothing of the kind. You stop when you cannot see a further move, which is a fact about you rather than the expression.

For the map, reliability under pressure: the procedure is mechanical, so two people working correctly reach the same answer by the same route. Algebraic paths diverge from the first line, and one that begins badly can run half a page before its author suspects anything.

Against the map, the ceiling: squares double with every variable added: eight, sixteen, thirty two, sixty four. At five variables the map becomes two layers and the adjacency between them must be held in the head; at six it becomes four; past that the eye gives up and the method with it.

For algebra, no ceiling and no fixed shape: an expression in ten variables is still an expression and the theorems apply unchanged. Algebra also reaches what the map cannot touch: a product of sums, a subexpression shared between two outputs, or a form chosen to suit the gates in the storeroom.

For algebra, the last step the map cannot take: a map returns a sum of products and stops. Turning A.NOT B + NOT B.C into NOT B.(A + C) removes a gate input, and no loop will ever suggest it. The two methods finish each other's work rather than compete for it.

The judgement: reach for the map first whenever there are four variables or fewer and a truth table is cheap to build, because it is quicker and, far more importantly, it tells you when to stop. Then spend a minute of algebra factoring the result. Use algebra alone once the variables pass four, when the target is not a sum of products, or when the expression arrives already written.

The map is a method with a boundary and algebra a method without one, and the sensible worker uses the bounded method inside its boundary, where it is the better of the two.

The Master Key: *Use the map where it fits, because it knows when it has finished, and hand the result to algebra, which never does.*

HOT Question 61

The claim is written across a shared page in Mtwara, in the hand of somebody who has just met a five variable map and did not enjoy the meeting. The first half is true. The second half is a conclusion drawn from it.

"The Karnaugh map is useless beyond four variables, so it is not worth learning." To what extent do you agree?

Model Answer

The premise is largely sound and the conclusion does not follow: the map does fail above four variables, but nearly every problem a candidate meets sits below that line, it teaches adjacency and redundancy far more plainly than algebra manages, and the reasoning it trains is what the tools apply to the cases it cannot reach.

For the claim, five variables strain the method: a five variable map is drawn as two four variable maps side by side, and squares in the same position on different sheets count as neighbours. That adjacency is real in the algebra and invisible on the page, and it is missed more often than caught.

For the claim, six variables break it: six variables need four sheets and adjacency in two directions between them, sixty four squares under an eye never built for it. Past six there is nothing left to try. The method does not degrade politely; it stops.

For the claim, serious designs are far larger: a single block inside a processor has hundreds of inputs, and nobody minimises that by hand. The tools used in practice run tabular and heuristic algorithms that never draw a grid.

Against the claim, the range you will actually be set: problems at this level almost always carry three or four variables, precisely where the map is strongest. A method that solves the problems you will be given is not useless, whatever happens above its ceiling.

Against the claim, what the map teaches that algebra hides: it makes adjacency visible. Algebra can tell you that X.Y + X.NOT Y = X; the map shows that two terms combine exactly when they touch. It shows redundancy as a shape too: a one covered twice costing nothing, a loop covering nothing new.

Against the claim, the intuition transfers upward: the algorithms inside the tools do what the map does, finding maximal groups of adjacent minterms and choosing a set that covers everything. A student who has grouped by hand can read what the tool reports; one who has not can only trust it.

The judgement: accept the premise and reject the conclusion. A tool with a ceiling is not a useless tool, in the way a spanner that does not fit every nut is still worth carrying. Learn the map for three and four variables, learn where and why it fails, keep algebra for the larger cases, and treat the ceiling as information rather than an argument against owning the tool.

Every method has a range, and the mark of somebody who understands a method is that they can say where it ends without concluding it should never have begun.

The Master Key: *The map fails above four variables and almost nothing you meet lives there, so learn it, learn its ceiling, and hand the rest to algebra.*

The logic gates, XNOR, and the universality of NAND and NOR

Seven gates are the entire alphabet of digital hardware, and two of them can spell every word alone. This section names all seven, gives each its Boolean expression, and settles the pair that costs the most marks: XOR, which reports difference, and XNOR, which reports agreement. It then proves why NAND and NOR are called universal, builds NOT, AND and OR from NAND alone, and spends the result on a maize dryer at Babati and a design argument at Geita.

BINDER Question 62

The signal cabin at Tabora stands above a junction where lines meet, and every lever answers with yes or no. Nothing in between, which is why trains arrive safely. A processor is built on the same refusal, and the pieces that answer are seven.

Describe the seven logic gates and give the Boolean expression of each.

Model Answer

Seven gates cover the whole of two input logic: AND gives A.B, OR gives A + B, NOT gives NOT A, NAND gives NOT (A.B), NOR gives NOT (A + B), XOR gives A.NOT B + NOT A.B, and XNOR gives NOT (A XOR B), the agreement of the two inputs.

AND, OR and NOT: AND outputs 1 only when both inputs are 1, written A.B. OR outputs 1 when at least one input is 1, written A + B. NOT inverts a single input, written NOT A. The other four gates are these three with an inversion added.

OR of 1 and 1 is 1: this is the row that costs marks. Speech uses “or” to mean one or the other but not both; the gate does not. OR asks only whether at least one input is high, so 1 + 1 = 1. The exclusive reading belongs to XOR.

NAND and NOR: a NAND is an AND with its output inverted, so NAND = NOT (A.B) and its column reads 1, 1, 1, 0 down the rows. A NOR is an OR with its output inverted, so NOR = NOT (A + B), high on the first row alone.

XOR and XNOR: XOR outputs 1 when the inputs differ, written A.NOT B + NOT A.B. XNOR outputs 1 when they agree and is the exact complement of XOR. Between them they ask the only question the other five never ask.

Reading the symbols: the shapes carry the logic, so memorise them. A flat back with a curved nose is AND, a curved back is OR, a small circle on an output means the result is inverted, and a second line across the back marks the exclusive pair.

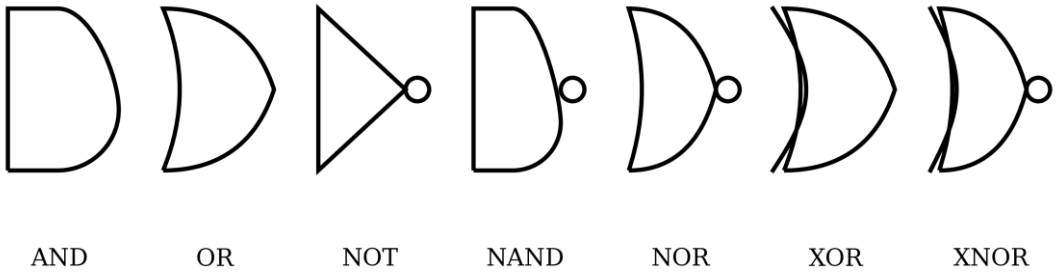


Figure 1.14: *The seven gates, in the order AND, OR, NOT, NAND, NOR, XOR and XNOR.*

A	B	AND	OR	NAND	NOR	XOR	XNOR
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1

Table 1.15: *The seven gates and their outputs.*

Seven gates, seven expressions and four rows each, and every circuit you will ever draw is an arrangement of them.

The Master Key: *Learn the seven expressions as one set, and remember that OR is inclusive while XOR is not.*

BINDER Question 63

At the cashew buying post outside Mtwara two scales weigh the same sack, one belonging to the buyer and one to the growers. Nobody at the table cares what either scale says alone. The only question worth asking is whether the two disagree.

Explain the XOR gate, state what makes it different from the OR gate, and give one real use.

Model Answer

XOR outputs 1 when its two inputs differ and 0 when they agree, which is where it parts company with OR, since OR outputs 1 whenever at least one input is 1, including the row where both are; its expression is XOR = A.NOT B + NOT A.B.

What XOR does: it reports difference. Feed it 0 and 1 in either order and the output is 1. Feed it 0 and 0, or 1 and 1, and the output is 0. Two of the four rows are high, and they are the unlike ones.

Where it separates from OR: on one row only, the row where both inputs are 1. OR calls that row 1 because at least one input is high. XOR calls it 0 because the inputs agree. The other three rows are identical, which is why the two are confused.

The expression read as a sentence: XOR = A.NOT B + NOT A.B says A is high while B is low, or B is high while A is low. Each product term names one disagreeing row and the plus collects them. The algebra is the definition written out.

Detecting a difference: give a gate the stored value and the incoming value and it raises a flag the instant they part. This is how a parity check catches a corrupted bit, and how a controller notices that a pair of sensors no longer agrees.

The sum bit in binary addition: add two bits and the sum column reads 0, 1, 1, 0 down the rows, with a carry generated on the last of them. That column is the XOR column exactly, which is why every adder ever built has an XOR gate at its heart.

One gate asks the question OR cannot be made to ask, and arithmetic depends on the answer.

The Master Key: *OR asks whether at least one input is high; XOR asks whether the two are unlike, and only the both high row separates them.*

BINDER Question 64

At the clove store on Pemba the same weight is written down twice, once by the weighman and once by the clerk, and the day is not closed until the two entries match. A machine given that job needs a gate of its own.

Explain the XNOR gate and describe its uses.

Model Answer

XNOR outputs 1 when its two inputs agree and 0 when they differ, making it the exact complement of XOR and the natural equality detector, which is why it sits at the heart of every circuit that compares two binary numbers.

What XNOR does: it reports agreement. Give it 0 and 0 and the output is 1. Give it 1 and 1 and the output is 1 again. Give it a mixed pair in either order and the output falls to 0. Its column reads 1, 0, 0, 1.

The complement of XOR: invert an XOR output and you have XNOR, so XNOR = NOT (A XOR B). Written in full it is A.B + NOT A . NOT B, one product term for the row where both are high and one for the row where both are low.

Why equality deserves a gate: comparison is among the commonest things a machine does, and the obvious method, subtracting and testing for zero, costs an arithmetic unit and several clock periods. One XNOR answers the same question for one bit in one gate delay.

Comparing two binary numbers: line the numbers up and give each bit position its own XNOR gate, A3 with B3, A2 with B2, and so on down. Every gate outputs 1 where its pair matches. AND all those outputs together and the result is 1 only when the numbers are equal throughout.

Where you meet it: in the equality output of a magnitude comparator, in address decoding where a device must answer to one address and no other, and in error checking where a received block is tested against the copy that was expected.

Agreement is a question no other gate answers cleanly, and machines ask it far more often than a diagram admits.

The Master Key: *XNOR is the equality gate, high when the inputs match, and a row of them under one AND compares whole numbers.*

BINDER Question 65

A drawer at a repair bench in Mpanda holds forty NAND chips and nothing else. The board needs an inverter, two AND gates and an OR. The technician orders nothing and waits for nobody, and the reason is a property with a name.

Explain why NAND and NOR are called universal gates, and show how NOT, AND and OR are each built from NAND gates alone.

Model Answer

NAND and NOR are called universal because each can build NOT, AND and OR on its own, and once you have those three you have every Boolean function; built from NAND, NOT costs one gate, AND costs two and OR costs three.

What universality means: that one gate type is enough for everything. Every Boolean function can be written with NOT, AND and OR alone, so any gate able to reproduce all three can reproduce any function whatever. NAND can. So can NOR. No other single gate can.

NOT from one NAND: connect both inputs of a NAND to the same signal A. The gate computes NOT (A.A), and A.A is simply A, so the output is NOT A. One gate, and the drawer no longer needs inverters in it.

AND from two NANDs: a NAND is an AND already inverted, so undo the inversion. Feed A and B into the first NAND to get NOT (A.B), then pass that result into a second NAND wired as an inverter. The output is A.B. Two gates.

OR from three NANDs: invert each input with its own NAND, giving NOT A and NOT B, then feed both into a third NAND. That gate computes NOT (NOT A . NOT B), which De Morgan turns into A + B. Three gates, and the theorem does the work.

Why anyone would bother: because one part type is cheaper to buy, to store and to fabricate than four. A line holding a single component has no wrong bin to reach into, and a silicon designer with one cell to lay out and verify saves that effort four times over.

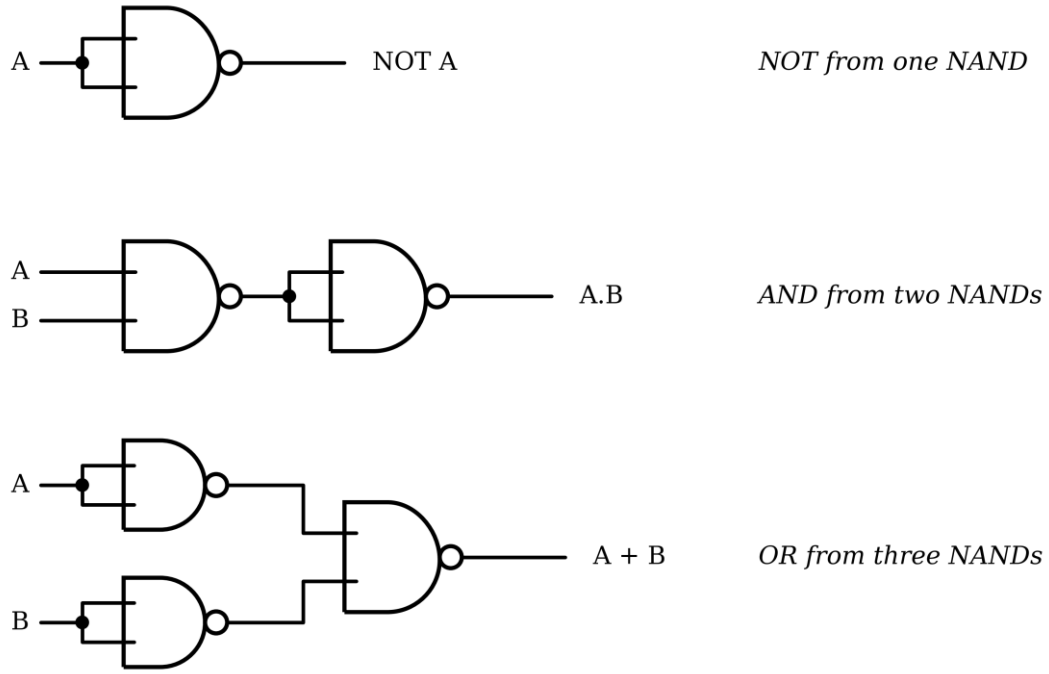


Figure 1.15: *NOT, AND and OR built from NAND gates only.*

One kind of gate in one drawer in Mpanda is a complete set of gates, and the proof is three small circuits.

The Master Key: *From NAND alone, NOT takes one gate, AND takes two and OR takes three, and those three build everything else.*

REAL Question 66

The maize dryer at a Babati mill forces hot air up through a column of grain, moving or not. Two sensors watch for the conditions that turn a dryer into a fire. The owner wants one gate between them and the burner relay.

Identify the gate that implements the condition below, justify the choice against the two nearest alternatives, give the Boolean expression, and verify your answer with a truth table.

The burner may fire only when neither fault is present.

H = 1 when the hopper sensor reports a blockage.

T = 1 when the outlet thermometer reports overheating.

Burn = 1 when the burner relay is allowed to close.

Both sensors go high on fault, and only one gate may sit between them and the relay.

Model Answer

The condition calls for a NOR gate, giving Burn = NOT (H + T), because NOR is the only two input gate whose output is high on the single row where both inputs are low, the one state in which this dryer is safe to fire.

Putting the sentence into symbols: a fault is present when H = 1 or T = 1, so any fault at all is H + T. The burner may fire only when that is false, which is Burn = NOT (H + T). The sentence has already named a NOR gate.

Why not NAND, the first near miss: NAND gives NOT (H.T), high unless both faults are live at once. A blocked hopper under a healthy thermometer would leave the relay closed and the burner roaring into a stationary column of grain, the exact case the owner is paying to prevent.

Why not OR, the second near miss: OR gives H + T, high whenever a fault exists. Wire that to the relay and the dryer fires only when something is wrong and sits idle when all is well, the correct logic running backwards. OR is the right sensing term and the wrong drive signal.

The row that decides it: H = 0 with T = 0 is the only safe state, so the output column must read 1 on that row and 0 on the other three. Exactly one two input gate carries that column down the table, and it is NOR.

Reading the verification: Table 1.16 sets NOR beside the NAND alternative on all four rows. They agree on the first and the last, and part on the two middle rows, which are the single fault cases. Those two rows are the whole argument.

The De Morgan reading of the same gate: NOT (H + T) is the same function as NOT H . NOT T, which says the hopper is clear and the temperature is safe. Both drive the same relay, so a bench with NOR chips builds the first and one with inverters and an AND builds the second.

Why one gate matters in a safety chain: every extra component between a sensor and a relay is another joint that can fail without announcing itself. One gate is not merely the cheaper answer; it is the shortest path along which a fault can hide.

H	T	H + T	Burn = NOT (H + T)	NAND, NOT (H.T)
0	0	0	1	1
0	1	1	0	1
1	0	1	0	1
1	1	1	0	0

Table 1.16: *The NOR condition checked against the NAND alternative.*

The owner asked for one gate, and the sentence he wrote on the delivery note had already chosen it.

The Master Key: *Only when neither input is high is the NOR condition, and the row where both inputs are 0 is where you check it.*

REAL Question 67

Three sentences are read aloud in a Korogwe study group and each one earns a nod. All three were written by people who had plainly done the work, which is what makes them dangerous. A claim that is almost right survives longest of all.

State the error in each of the three claims and give the correct version.

1. “The OR gate outputs 0 when both inputs are 1, because it means one or the other but not both.”

2. “XNOR is the same as NOR, since both start with N.”

3. “A NAND gate can replace an AND gate directly in a circuit without any other change.”

Model Answer

All three fail: OR is inclusive, so 1 + 1 = 1 and the exclusive reading belongs to XOR; XNOR and NOR differ where both inputs are high; and a NAND put in place of an AND inverts the result, so an inverter is needed to restore it.

The first error, an everyday word borrowed carelessly: in speech, “tea or coffee” quietly excludes taking both, and the gate does not work that way. OR is inclusive: it outputs 1 whenever at least one input is 1, and that includes the row where both are. Corrected, 1 + 1 = 1.

Where that meaning does live: in XOR, which is high only when the inputs differ and low when both are 1. The sentence accurately describes the wrong gate. Attach those same words to XOR and every part of the claim becomes true.

The second error, a letter mistaken for a family: the N in NOR negates an OR and the N in XNOR negates an XOR, and those are different starting gates. Corrected, XNOR outputs 1 whenever the two inputs agree, while NOR outputs 1 only when both inputs are 0.

One row settles the second claim: take A = 1 and B = 1. XNOR gives 1, because the inputs agree. NOR gives 0, because an input is high. One disagreement is a disproof. The gates do match on the other three rows, and that near miss made the slip possible.

The third error, a bubble left out of the account: a NAND is an AND with its output inverted, so dropping one into the socket flips the result on all four rows. Corrected, a NAND may replace an AND only with an inverter added after it, two gates where the circuit had one.

What the third claim is reaching for: universality, which is true and says something different. NAND can build an AND, but it builds it as a pair, not as a swap. Possible is not the same word as interchangeable, and here the difference costs a gate.

The habit that catches all three: test the sentence on the row where both inputs are 1. That one row exposes the inclusive OR, separates XNOR from NOR, and reveals the inverted NAND output. Almost all gate confusion is living on that row.

Three careful sentences, one row of a truth table, and it is the same row that breaks each of them.

The Master Key: *Check the both high row before believing any claim about gates, because that is where OR, XOR, NOR and XNOR all part company.*

HOT Question 68

A board designer at a Geita mine is told by a supplier that one NAND part will do everything, and the arithmetic on the quotation supports him. The claim is not false. It is simply narrower than the sentence it arrives in.

"Since NAND is universal, every other gate is redundant." To what extent do you agree?

Model Answer

The claim is sound in logic and misleading in practice: NAND really can build every function, so no other gate is logically necessary, but gate count, propagation delay and plain readability all worsen in a NAND only design, and redundant is the wrong word for a part working engineers keep reaching for.

For the claim, the mathematics is settled: NAND builds NOT, AND and OR, and every Boolean function can be written from those three. There is no function anywhere that a network of NAND gates cannot compute. On the narrow question of what is possible, the sentence is correct.

For the claim, one part is cheaper than four: a workshop stocking a single chip type has one order to place, one bin to check and one spare to carry. A line with one component cannot fit the wrong one. The saving is real, and it appears in the storeroom rather than on the board.

For the claim, uniformity pays inside the silicon: a chip designer lays out one cell, characterises it once for timing, power and yield, then repeats it across the die. Much of the industry does exactly this, so the argument is not a curiosity but ordinary manufacturing practice.

Against the claim, the gate count rises: one AND becomes two NANDs and one OR becomes three. A design of any size swells accordingly, and the extra gates eat board area, solder joints and current. Part of the storeroom saving is spent again on the board.

Against the claim, delay accumulates with depth: every gate a signal passes through costs propagation time, and NAND chains run deeper than the mixed circuits they replace. An OR that took one gate delay now takes two, and on a board that must settle inside a clock period, depth fails first.

Against the claim, a human has to read it: a diagram drawn in AND, OR and NOT states what the circuit means, because the shapes match the sentence the designer began from. The same function in NAND is a field of identical symbols with the intention rubbed out, and every repair is slower for it.

The judgement: agree on the logic and reject the word redundant. Nothing else is necessary, but necessity was never the test an engineer applies. The other gates survive because they are shallower, faster and legible, and a tool kept for those reasons is not redundant but chosen.

Universality tells you what can be built and stays silent on what should be, which is the question the designer at Geita is paid to answer.

The Master Key: *NAND makes every other gate unnecessary and none of them redundant, because depth, delay and readability still have to be bought.*

HOT Question 69

A small firm in Singida builds pulse counters for irrigation meters, and the same adder has been drawn twice on the bench, once with an XOR gate and once without. Both drawings work. Only one matches the parts the workshop can get this month.

Assess how a designer should choose between building a function from XOR gates and building the same function from AND, OR and NOT, using binary addition as the concrete case, and reach a judgement.

Model Answer

Use XOR wherever the function is genuinely a difference or a sum, since it says so in one gate instead of five, and fall back on AND, OR and NOT only when the part is unavailable or the exclusive structure has already been broken up.

The concrete case, a half adder: adding two bits produces a sum and a carry. Sum = A XOR B and Carry = A.B. Two gates in total, and the second is trivial. One whole bit position of binary addition is one XOR and one AND.

The same sum bit without XOR: written in basic gates, Sum = A.NOT B + NOT A.B. That is two inverters, two AND gates and an OR, five gates where there was one, with a signal path three levels deep instead of one.

Gate count is the obvious argument: one against five for a single bit, and an adder is repeated once per bit position. Across an eight bit machine that difference multiplies, then multiplies again for every adder in the design. Where a function is naturally exclusive, XOR is a compression, not a convenience.

Clarity favours XOR as well: Sum = A XOR B reads as a sentence: the sum bit is 1 when the inputs differ. The five gate version says the same thing while forcing the reader to rebuild the meaning from the terms. A diagram that states its intention is cheaper to maintain.

Availability argues the other way: XOR is not a primitive in silicon. It is assembled from simpler gates, and on a stocked shelf it is the package most likely to be missing. A bench with AND, OR and NOT in the drawer builds the five gate version and ships the meter.

The surrounding logic decides more often than either: if the rest of the board is mixed gates already, and a Karnaugh map has flattened the expression into products and sums, forcing an XOR back in may reduce nothing. Choose the form that suits the circuit around it, not the one that looks tidy alone.

The judgement: prefer XOR whenever the function truly is difference or sum, since one gate against five is a saving in count, in depth and in readability at the same time. Abandon it only for a hard reason: no part on the shelf, or a design in which the exclusive structure has already gone.

The firm in Singida should draw the adder with an XOR and keep the five gate version for the month the shelf is empty.

The Master Key: *XOR earns its place where the function is genuinely exclusive; where it is not, or where the part is not there, the three basic gates are no worse than honest.*

Expression to circuit, and circuit to expression

Every other skill in this chapter exists to serve this one. You are handed an expression and asked for the circuit, or handed a circuit and asked for the expression, and the marks sit in the crossing. This section gives a method for each direction, builds Y = (A + B).NOT C gate by gate, reads Y = NOT (A.B) + C back off a diagram, and spends both on a pump at Kyela and a weighbridge at Tunduma.

BINDER Question 70

At the Chalinze junction a fitter under a lorry tells you the same thing. You assemble from the middle outwards, because the part you can no longer reach is the part you should have fitted first. A Boolean expression is built the same way.

Explain the method for turning a Boolean expression into a logic circuit, and use it to build Y = (A + B).NOT C.

Model Answer

Work from the inside out. Find the innermost operation by precedence, which runs NOT first, then AND, then OR, give every bracketed part its own gate, label the wire it produces, and feed that wire outwards until a final gate delivers Y.

Precedence is the build order: the expression does not tell you where to begin by the order it is written in. It tells you by the strength of its operators. NOT binds tightest, AND comes next, OR is loosest, and the tightest operation is the first gate drawn.

A bracket is a gate: anything inside brackets must be finished before it can be used, which is exactly what a gate does to its inputs. Give each bracketed part a gate of its own and the drawing takes on the same nesting the algebra already has.

Label every intermediate wire: name the output of each gate as you draw it, P, Q, R, and write beside it the expression it carries. Unlabelled wires are where a three gate circuit turns into a guess, and a labelled drawing can be checked without redrawing it.

Building Y = (A + B).NOT C: two things are innermost here, the bracket and the inversion. Draw an OR gate fed by A and B and call its output P, so P = A + B. Draw an inverter fed by C and call its output Q, so Q = NOT C. Neither waits on the other.

Closing with the final gate: the dot between the bracket and NOT C is the last operation, so it takes the last gate. Feed P and Q into an AND gate and its output is Y. Three gates, two innermost and one final, exactly as figure 1.16 shows.

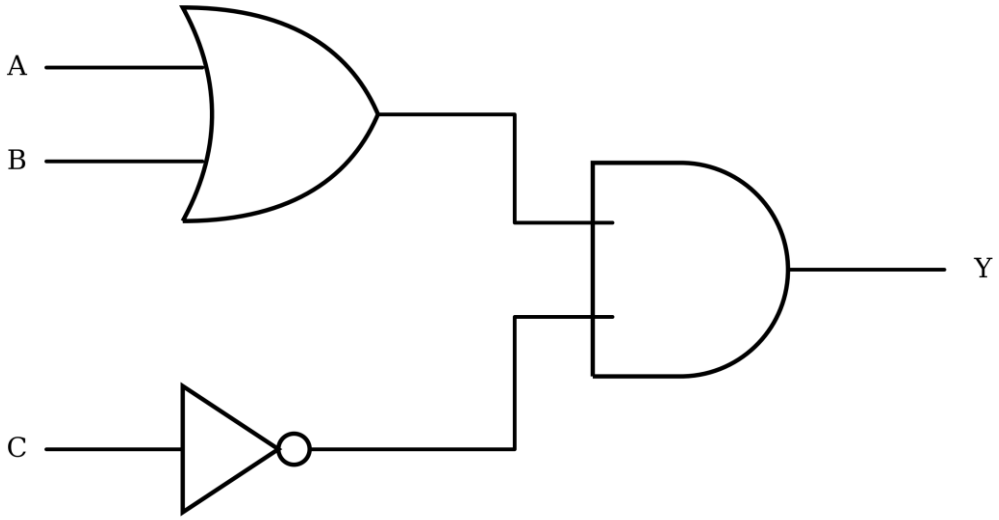


Figure 1.16: *Y = (A + B).NOT C, built from the inside out.*

Read the expression for its strongest operator, draw that gate, and work outwards until nothing is left to draw.

The Master Key: *Inside out, NOT before AND before OR, one gate for every bracket, and a label on every wire you make.*

BINDER Question 71

The cotton store at Nzega passes a lorry when any one of several conditions is met, each a bundle of facts that must all hold. All of this and that, or all of a third. Written down, that sentence is a sum of products.

Draw and explain the circuit for the three term sum of products Y = A.B + NOT A.C + B.C, describe the shape such circuits always take, and state the gate count.

Model Answer

Every sum of products becomes a rank of AND gates feeding one OR gate: one AND for each product term, a single OR collecting them, and inverters at the far left wherever a variable appears complemented. Here that is one inverter, three ANDs and one OR, five in all.

The three product terms: read the pluses as the joints. A.B is one term, NOT A.C is a second, B.C is a third. Each is a product, so each takes one AND gate, and no two terms mix before they reach the OR.

The inverter comes first: NOT A is the only complemented variable here, so a single inverter sits at the far left fed by A. Its output is a signal in its own right and may be tapped by as many AND gates as need it.

The rank of AND gates: stack three AND gates in a column. The first takes A and B, the second the inverter output and C, the third B and C. Label their outputs P, Q and R, and notice that A, B and C each feed more than one gate from a shared rail.

One OR gate collects them: P, Q and R enter a single three input OR gate whose output is Y. After the inverter the circuit is only two levels deep, which is why sum of products is the standard form. Its delay is short and stays the same whatever the terms say.

Counting the gates: one inverter, three ANDs and one OR make five. If the bench holds only two input parts, the three input OR becomes two and the count rises to six. State which parts you assumed and either count is defensible.

Sum of products always draws the same picture, and once you can see the picture you can draw any of them.

The Master Key: *One AND for every product term, one OR to collect them, inverters at the left, and five gates here.*

BINDER Question 72

A borehole pump outside Handeni arrived with nothing written down, only a control board with the wiring still on it. The fundi could not ask what it was meant to do. He could only read what it does, from copper back to sentence.

Explain the method for reading a Boolean expression back off a logic circuit, and read the expression off figure 1.17.

Model Answer

Start at the inputs and never at the output. Write on each gate output the expression that gate produces, then treat that wire as an input to the next gate along, and work forward until the final gate hands you the whole expression.

Start at the inputs: the inputs are the only wires whose expressions you already know, because they are A, B and C themselves. Every other wire has to be earned. Beginning at the output means carrying unknowns in your head, and that is where the thread is lost.

Label the output of every gate: the moment a gate is fed, write what comes out of it on the wire itself. A three gate circuit needs three labels and the last one you write is the answer. This costs seconds and saves the whole question.

Watch the bubble: a small circle on an output means the result is inverted, so that shape is NAND and not AND, NOR and not OR. The bubble is the smallest mark on the page and the one that changes the answer most.

Bracket whatever a gate produces: a gate output is one finished signal, so wrap it before feeding it onward. NOT (A.B) is the whole product inverted. NOT A.B reads as NOT A first and then ANDed with B, which is a different function. The brackets are the wire.

Reading figure 1.17: A and B enter a NAND gate, so its output wire is labelled NOT (A.B), the bubble included. That wire and the input C enter an OR gate, and an OR adds whatever reaches it, so Y = NOT (A.B) + C. The circuit has been read.

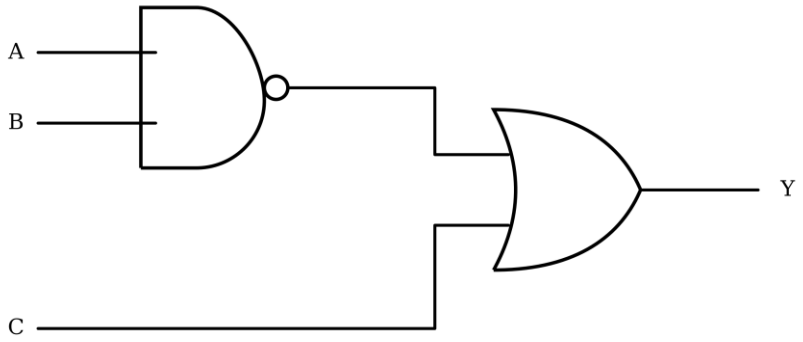


Figure 1.17: *Read this circuit back and it says Y = NOT (A.B) + C.*

Two gates, two labels, and the diagram has said out loud what it does.

The Master Key: *Inputs first, a written label on every gate output, brackets round anything a gate produces, and mind the bubble.*

BINDER Question 73

Two mechanics at Kilosa write the same three parts on the same order form and receive two different machines. The parts are identical. The order in which they are assembled is not. A.B + C and A.(B + C) are that order form.

Explain how the order of operations decides the shape of a circuit, using A.B + C and A.(B + C), and give an input combination on which the two disagree.

Model Answer

Precedence, not the order the symbols are written in, decides which gate is innermost, so A.B + C draws an AND feeding an OR while A.(B + C) draws an OR feeding an AND. Two gates each, and two rows where they part company.

A.B + C puts the AND innermost: the dot binds tighter than the plus, so A.B is finished first. Draw an AND gate fed by A and B, label its output P, then feed P and C into an OR gate. The AND is built first and the OR last.

A.(B + C) puts the OR innermost: the brackets overrule precedence and force B + C first. Draw an OR gate fed by B and C, label its output Q, then feed A and Q into an AND gate. The same two gate types, swapped in rank.

Where the two disagree: set A = 0, B = 0 and C = 1. Then A.B + C is 0 + 1, which is 1, because C alone can carry an OR. But A.(B + C) is 0.(0 + 1), which is 0, because A sits on the outside and A is low. One row proves the two are different functions.

Reading the two columns: table 1.18 sets them side by side. They agree on six rows and differ on the two where A is 0 while C is 1. A.B + C is high whenever C is high, while A.(B + C) can never be high with A low, because the final AND holds A on one input.

When in doubt, bracket: brackets cost nothing and end the argument. If you mean the OR to happen first, write it in brackets. Far more marks are lost to an assumed precedence than to an unnecessary pair of brackets.

A	B	C	A.B + C	A.(B + C)
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Table 1.17: *The same symbols, two different functions.*

Same symbols, different ranks, a different machine, and the truth table is where the difference shows.

The Master Key: *Precedence chooses the innermost gate, brackets overrule precedence, and A.B + C is not A.(B + C).*

REAL Question 74

The rice scheme below Kyela runs its pump off one relay, and the man wiring it must satisfy a station manager, a pump that burns out when dry, and a filter that blocks every wet season. He has one sentence and needs a circuit.

Translate the requirement below into a Boolean expression, describe the circuit gate by gate, draw it, and state the gate count.

S = 1 when the sluice gate is fully open.

L = 1 when the day tank reads low.

M = 1 when the operator is holding the manual override.

F = 1 when the intake filter alarm is raised.

P = 1 when the pump motor is allowed to run.

The pump may run when the sluice is open and either the tank is low or the override is held, and it may never run while the filter alarm stands.

Model Answer

The requirement is P = S.(L + M).NOT F. The sluice and the filter are both compulsory conditions, so they meet at an AND; the tank and the override are alternatives, so they meet at an OR; and the alarm enters inverted, because the pump runs when the alarm is absent.

Turning the sentence into symbols: and becomes a dot, either or becomes a plus, and never while becomes an inversion feeding an AND. Three joining words in the sentence, three operators in the expression, matched one for one before a gate is drawn.

The bracket is the safety case: L + M must be bracketed. Written as S.L + M the expression says the pump runs whenever the override is held, sluice open or shut, and a pump driving into a closed sluice is the accident the manager is paying to prevent.

The two innermost gates: draw an OR gate fed by L and M and label its output G, so G = L + M. Draw an inverter fed by F and label its output H, so H = NOT F. Neither waits on the other, so both belong in the first column.

The final AND: S, G and H must all be true together, so they enter one three input AND gate whose output is P. S runs from the input rail straight to that gate, which is correct, since the sentence asks nothing of the sluice signal except that it be high.

The gate count: one OR, one inverter and one AND make three gates. With two input parts only, the three input AND becomes two ANDs and the count is four. Write the assumption beside the count and either answer stands.

Checking two rows before you hand it in: take S = 1, L = 0, M = 1 and F = 0. Then P = 1.(0 + 1).NOT 0, which is 1, so the override works with the sluice open. Now raise the alarm to F = 1. The inverter delivers 0, the AND collapses, and P = 0 whatever else is happening.

Drawing discipline: run S, L, M and F down the left as four labelled rails, keep the OR and the inverter in one column, put the AND in the next, and write G, H and P on the wires. A marker who can follow the drawing never guesses your intention.

The sentence on the delivery note was already the circuit, and the drawing only made its shape visible.

The Master Key: *And is a dot, or is a plus, never is an inversion into an AND, and the bracket round L + M carries the whole safety case.*

REAL Question 75

A ginnery at Bariadi keeps a control card that has worked for eleven seasons, and the man who drew it is long gone. It carries six gates. The new engineer asks not whether it works, but whether it has any business being that big.

Read the expression off the circuit described below, simplify it, and state how many gates the simplified version needs.

Two inverters sit at the left of the card, one fed by A and one fed by B.

The first AND gate is fed by A and by B.

The second AND gate is fed by A and by the output of the inverter on B.

The third AND gate is fed by the output of the inverter on A and by B.

One three input OR gate collects the outputs of all three AND gates.

The output of that OR gate is Y.

Model Answer

Read forward and the card says Y = A.B + A.NOT B + NOT A.B. Simplified it says Y = A + B, which is one OR gate. Six gates are doing the work of one, and table 1.18 shows the two agreeing on all four rows.

Label the three AND outputs: the first AND is fed by A and B, so its wire carries A.B. The second carries A.NOT B. The third carries NOT A.B. Nothing has been simplified at this stage; the wires have only been given their names.

The OR gate gives the expression: an OR adds whatever reaches it, so Y = A.B + A.NOT B + NOT A.B. That is the card read back, three product terms collected into one sum, and it is worth writing down in full before any algebra is attempted.

Factor the first two terms: A.B + A.NOT B carries A in both, so take it outside: A.(B + NOT B). A variable ORed with its own complement is 1, and A.1 is A. Two AND gates and part of the OR have just collapsed into the bare input A.

Absorb the third term: what remains is A + NOT A.B. Where A is 1 the whole expression is 1 already, and where A is 0 the second term reduces to B, so the output is high whenever A or B is high. That is A + B, and the law A + NOT A.B = A + B says the same.

One gate instead of six: two inverters, three ANDs and a three input OR come to six gates. Y = A + B needs one OR gate fed by A and B. Five of the six were doing nothing the sixth was not already doing on its own.

What the table settles: table 1.18 carries every intermediate wire and both answers. The Y column reads 0, 1, 1, 1 and so does the A + B column. Agreement on all four rows is proof, and no run of algebra is worth as much to a marker as that pair of columns.

Why nobody noticed for eleven seasons: a bloated circuit is not a broken circuit. It works, and it costs: more delay between input and relay, more current drawn, more solder joints to corrode in a shed full of cotton dust. Simplification is bought with algebra and paid out in reliability.

A	B	A.B	A.NOT B	NOT A.B	Y	A + B
0	0	0	0	0	0	0
0	1	0	0	1	1	1
1	0	0	1	0	1	1
1	1	1	0	0	1	1

Table 1.18: *Six gates and one gate, agreeing on every row.*

The card was right for eleven seasons and expensive for all of them.

The Master Key: *Label every gate output first and simplify afterwards, and here six gates fall to one because Y = A + B.*

REAL Question 76

A wall chart in a study room at Masasi carries three rules about turning expressions into circuits, written in a confident hand and copied into thirty exercise books. Two of the three sound obvious, which is why they have survived. All three are wrong.

Identify the error in each of the three claims below, correct it, and state the test that settles the third.

“In the expression A + B.C you draw the OR gate first because it appears first.”

“A circuit with four gates always needs four terms in its expression.”

“If two circuits have the same number of gates they compute the same function.”

Model Answer

All three swap something visible for something true. Reading order does not decide the first gate, precedence does. Gate count and term count are unrelated quantities. And identical gate counts say nothing whatever about function, which only a truth table can settle.

The first claim, and why it fails: position on the page carries no authority at all. In A + B.C the dot binds tighter than the plus, so B.C is the innermost operation and its AND gate is built first. That gate output then feeds the OR gate along with A.

What the first claim would build instead: drawing the OR first gives (A + B).C, a different circuit and a different function. Set A = 1, B = 0 and C = 0. Then A + B.C is 1 while (A + B).C is 0. One row is enough to show that the chart has changed the answer.

The second claim, and why it fails: gate count and term count measure different things. A single term may need several gates, since NOT A.B.C takes one inverter and two AND gates, three gates for one term. Count gates on the drawing and terms in the expression, and expect no relation between the two.

The second claim from the other side: one gate may also serve several terms. A single inverter producing NOT A can be tapped by every product term that needs it, so three terms may share one gate between them. Shared wires are exactly what makes the two counts drift apart.

The third claim, and why it fails: size is not function. A.B + C and A.(B + C) each use two gates, and table 1.18 shows them parting on two of their eight rows. Equal gate counts prove nothing, and two circuits of very different size can compute the same thing.

The test that settles it: build the truth table of both and compare them row by row. Identical columns throughout means one function; a single differing row means two. Nothing else counts as proof, and nothing else is what the question is asking for.

What the three errors share: each trusts a surface feature, the order of the symbols, the number of them, the number of boxes on the page, in place of the logic underneath. The habit that defeats all three is to build inside out, label every wire, and test with a table.

Three confident sentences, thirty exercise books, and not one of the three survives a truth table.

The Master Key: *Precedence decides the first gate, gate count and term count are unrelated, and only a truth table proves two circuits agree.*

HOT Question 77

An instrument workshop at Moshi keeps two piles on a shelf. One holds expressions, tight and complete and quick to write. The other holds drawings, slow to make and easily smudged. A young technician proposes burning the second pile, and his argument is good.

"Drawing the circuit is unnecessary once you have the expression." To what extent do you agree?

Model Answer

The claim is true about information and false about engineering. An expression fixes the function completely, so nothing is lost in principle, yet almost everything that has to be decided next, gate count, depth, shared wires, fan out and cost, becomes visible only when the thing is drawn.

For the claim, the expression is complete: every row of the truth table can be recovered from the expression alone, so the function is fully specified without a single gate on paper. Two people working from the same expression must arrive at the same behaviour, whatever their drawings happen to look like.

For the claim, simplification happens in algebra: the work that actually shrinks a circuit is done with Boolean laws, on the expression, before anything is drawn. Drawing a bloated circuit and tidying it afterwards is the slow route, and it is what produces a card of six gates doing the work of one.

For the claim, machines take the expression: a modern design is written as expressions and handed to software that generates the layout. In parts of the industry the drawing has genuinely disappeared, which is why the technician cannot be dismissed as merely lazy.

Against the claim, cost lives in the drawing: the expression does not say how many gates, how many packages, or how deep the chain runs. Draw it and you can count: five gates, two levels, one shared inverter. Nobody buys an expression. They buy the parts it turns into.

Against the claim, the drawing exposes structure: a shared wire, an output asked to drive more gates than it can, an inverter that could serve three terms instead of one, none of these are legible in a line of algebra. They appear the moment the same statement is laid out in two dimensions.

Against the claim, somebody has to repair it: the man who opens the panel years later has a board in front of him, not a page of symbols. He works from a diagram, traces a wire, finds the bubble. Remove the drawing and you remove the only document that matches what he can actually see.

The judgement: agree that the expression is sufficient to define, and reject the word unnecessary. The circuit is not a decoration of the expression. It is the same statement priced, timed and made repairable, so keep the algebra for deciding what to build and the drawing for everything that follows.

The two piles at Moshi answer different questions, and only one of them can be held against a screwdriver.

The Master Key: *The expression defines the function while the drawing shows cost, depth and repair, so neither one replaces the other.*

HOT Question 78

Two students at Karatu work together. One draws circuits from expressions until his hand aches and has never read one back. The other reads diagrams all week and has not drawn a gate since April. Each thinks the other took the harder half.

Assess which is harder, going from expression to circuit or from circuit to expression, explain what makes each direction fail under pressure, and reach a judgement about how a candidate should practise.

Model Answer

Expression to circuit is the harder direction to execute and the easier one to learn, because it is a procedure. Circuit to expression is the easier direction to execute and the harder one to trust, because it depends on seeing what is on the page rather than what you expect to be there.

Expression to circuit is a procedure: there is an order and it never varies: NOT, then AND, then OR, brackets innermost, one gate at a time, labels as you go. A procedure can be drilled until pressure no longer touches it, which is why this direction rewards repetition.

How the drawing direction fails: on precedence and on space. A candidate draws the gates in reading order, loses a bracket, and builds (A + B).C where A + B.C was asked for. Or he starts too far left, runs out of page, and the last gate is crushed against the margin with its inputs crossing.

The drawing direction also costs time: gates take far longer to draw than symbols take to write, and a five gate circuit with every wire labelled is several minutes gone. Candidates who have not practised the physical act of drawing lose marks to the clock, not the logic.

Circuit to expression is perception, not procedure: there is no order to follow because the page decides it. You must see a bubble two millimetres across, notice that one wire feeds two gates, and tell a junction dot from a plain crossing. Nothing makes the eye more careful except practice at looking.

How the reading direction fails: in four ways, all of them silent. A missed bubble turns NAND into AND. A dropped bracket turns NOT (A.B) into NOT A.B. Working backwards from the output loses the thread halfway along. And a shared wire read as two inputs invents a variable that was never there.

Why the drawing direction still costs more marks: it is asked more often, it carries marks for the diagram itself on top of the marks for the logic, and its failures show as a wrong shape rather than a wrong letter. Reading is harder per question and cheaper across the paper.

The judgement and the practice: treat the two as one skill exercised in both directions on the same item. Build a circuit from an expression, cover the expression, read the circuit back, and set what you wrote against what you started with. Any mismatch is a fault you would have carried into the hall.

Both candidates at Karatu are half trained, and the training that finishes either of them is the round trip.

The Master Key: *Drawing fails on precedence and space, reading fails on bubbles, brackets and shared wires, so practise out and back on one item.*

HOT Question 79

A weighbridge at Tunduma stopped passing lorries the morning after a technician copied its control diagram onto a clean sheet. He changed nothing. He said so, and he believed it. The board is the same board, and the only thing that moved was ink.

Evaluate the likely causes of the failure in terms of the translation between diagram and expression, judge which is most likely, and explain how it would be found.

Model Answer

A neat copy is a translation, and every translation can drop something. Four losses account for almost all such failures: a bubble omitted, inputs swapped where the arrangement is not symmetric, a shared wire drawn as two inputs, and a junction dot lost at a crossing. The bubble is likeliest.

A bubble omitted: a small circle on an output is the whole difference between AND and NAND, and it is the smallest mark on the page. Redraw NOT (A.B) without it and the circuit computes A.B instead, high where it should be low. Every row inverts.

Inputs swapped where order matters: AND and OR do not care which input is which, so a swap on its own is harmless. It stops being harmless the moment one input passes through an inverter, because A.NOT B and NOT A.B are different functions, and a copy that feeds the inverter from the wrong rail has exchanged them.

A shared wire drawn as two inputs: one signal tapped by two gates is easily redrawn as two separate inputs bearing the same name, or worse, two different names. The copy then carries a variable the board never had, and the two can take different values on paper although only one wire exists.

A junction dot lost: where two wires cross, a dot means joined and no dot means passing. Lose the dot and a gate input is fed by nothing at all. A floating input never announces itself, it settles at whatever level the technology favours, so the output looks stuck rather than wrong.

Judging which is most likely: the bubble, and the reason lies in the motive. The copy was made for neatness, and neatness is a hand moving quickly over shapes it already recognises. A bubble is an ornament to the eye and a change of gate to the circuit, so it is the mark most easily dropped.

How it would be found: not by staring at the new drawing, which looks correct to the man who made it. Read both diagrams back to expressions independently, labelling every gate output, then lay the two side by side. The first wire whose labels differ is the fault, and its gate is the one to inspect.

The confirmation on the bench: build the truth table of both expressions, find the first row on which they disagree, set those inputs on the board and probe forward. A lost bubble shows as a column inverted from one gate onward, a lost dot as an input with nothing driving it.

Nothing on the board changed, which is exactly why the fault lies in the reading and not in the copper.

The Master Key: *Redrawn circuits fail at the bubble, the inverted input, the shared wire and the lost dot, and reading both drawings back finds it.*

Adders and subtractors

Addition is the only arithmetic a processor truly owns, and subtraction, multiplication and comparison are all built on top of it. This section builds the adder from the bottom up: the half adder and its two expressions, the full adder that accepts a carry from below, the ripple chain that turns single stages into a four-bit machine, and the two's complement trick that makes the same hardware subtract. It then spends the result on a breadboard at Kibaha, a weighbridge at Tunduma and a design argument at Karatu.

BINDER Question 80

The weighbridge at Chalinze counts axles on two lanes and wants one total. In binary one plus one is not a digit; it is 10, two characters in a column that was only ever allowed one. Every adder ever built begins with that embarrassment.

Describe the half adder, derive its two Boolean expressions, and give its truth table.

Model Answer

A half adder adds two single bits and produces two outputs, a sum and a carry, given by Sum = A XOR B and Carry = A.B; it is called half an adder because it has no input for a carry from the column below, so it can serve the least significant column of a wider addition and no other.

What one column of binary addition demands: four cases and one difficulty. 0 + 0 is 0, 0 + 1 is 1 and 1 + 0 is 1, but 1 + 1 is 10, which needs two digits. One output cannot report that row honestly, so the half adder is given two: a sum for this column and a carry for the one above.

Deriving the sum: read down the sum column. It is 1 on the two rows where the inputs differ and 0 where they agree, which is the definition of XOR and of nothing else. Hence Sum = A XOR B.

Deriving the carry: read down the carry column. It is 1 on one row only, where A and B are both 1, and that is the AND column exactly. Hence Carry = A.B. Two gates, and the circuit is finished.

Why the name says half: because a real column of addition holds three things, not two: the two digits and the carry pencilled in from the column to its right. The half adder has only two inputs, so it cannot accept an incoming carry.

Where a half adder still serves: at the least significant column, which has nothing to its right and so nothing to carry in. Bit zero of a wider adder may be a half adder; every column above it may not.

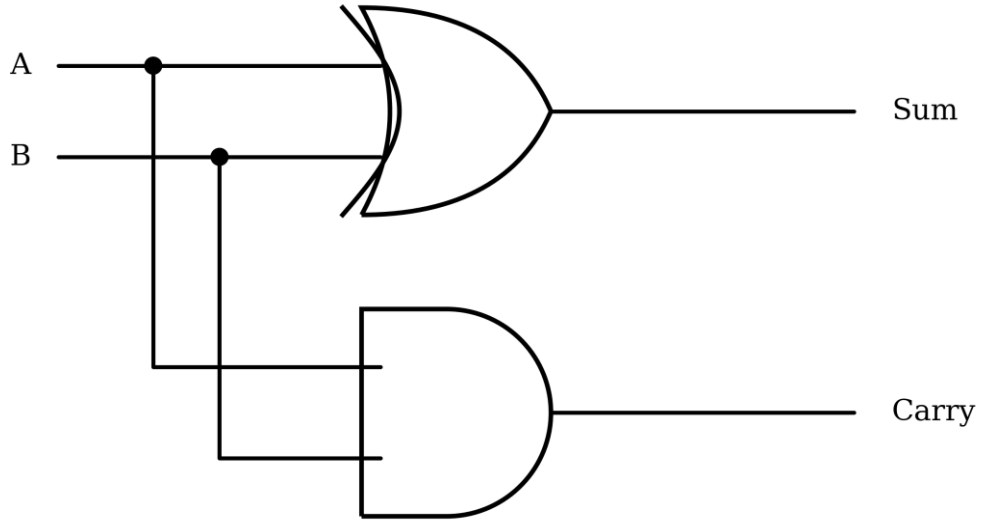


Figure 1.18: *The half adder. One XOR for the sum, one AND for the carry.*

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 1.19: *The half adder.*

Two gates and four rows, and the machine can add. What it cannot do is join that answer to the column beneath it, which is the business of the next question.

The Master Key: *Sum is XOR and carry is AND, and the word half means one missing input, not one missing output.*

BINDER Question 81

The clerk at the Handeni cotton store adds his columns by hand and never adds only two digits. There is always a third figure carried in from the column to its right. A circuit that cannot take it is not yet an adder.

Describe the full adder, give its truth table, and state its two Boolean expressions.

Model Answer

A full adder adds three bits, A, B and a carry in from the column below, and produces a sum and a carry out, with Sum = A XOR B XOR Cin and Carry out = A.B + Cin.(A XOR B); that third input is what the half adder lacks, and it is what lets adders be joined into a chain.

Three inputs, eight rows, two outputs: three bits total between 0 and 3, and 3 is binary 11, so two output bits are needed again. Three inputs give eight rows, and every row must agree with A + B + Cin written in binary.

Reading the sum column: the sum bit is 1 on the four rows where an odd number of inputs is 1, since only an odd total leaves a 1 in the ones place. Chained XOR reports that, so Sum = A XOR B XOR Cin.

Reading the carry out column: a carry appears whenever at least two inputs are 1, and there are two ways for that. A and B may both be 1, which generates a carry whatever comes from below, or exactly one may be 1, in which case an incoming carry is passed on. Hence Carry out = A.B + Cin.(A XOR B).

Built from two half adders: the first half adder adds A to B and hands its sum to the second, which adds Cin. An OR gate joins the two carries, since a carry from either stage is a carry out of the whole.

Rebuild the table, do not memorise it: add the three input bits as ordinary numbers, write the total in binary, then put the twos digit in the carry out column and the ones digit in the sum. Row 011 gives 0 + 1 + 1 = 10, so sum 0 and carry 1.

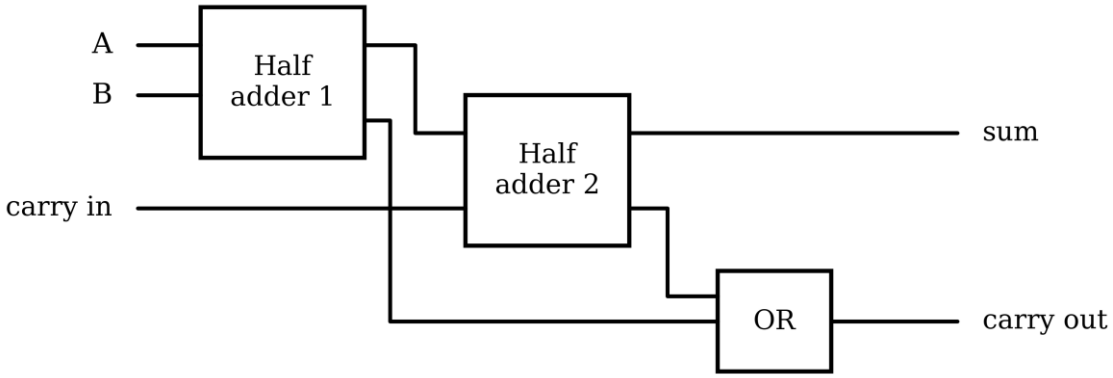


Figure 1.19: *A full adder made from two half adders and an OR gate.*

A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1.20: *The full adder.*

One extra input turns a curiosity into a component. With a carry in and a carry out, a full adder can take the hand of the one below it.

The Master Key: *Sum is the XOR of all three inputs, and the carry out has two terms because a carry can be generated or passed on.*

BINDER Question 82

The line through Kilosa is single track, and no train may enter a section until the train ahead has cleared it. The line then runs at the speed of the handover rather than the engines. A ripple carry adder inherits the same weakness.

Explain how full adders are chained into a ripple carry adder, and explain why the carry is the limiting factor.

Model Answer

A ripple carry adder joins n full adders so that the carry out of each stage becomes the carry in of the next, with the carry into stage 0 tied to 0; the chain is correct but slow, because no stage can settle until the carry beneath it arrives, so an n-bit adder needs roughly n gate delays and the worst case is a carry that travels from the least significant bit to the most significant.

How the chaining is done: one full adder per bit position. Stage i takes its operand bits from A and B, and its carry in from the stage below. Stage 0 is tied to 0, and the top carry out leaves the circuit.

Why it is called ripple: the operand bits all arrive at once, but the carries do not. Stage 0 must produce its carry before stage 1 can produce its own, so a wave travels upward after the inputs settle.

Why the carry is the problem: the operands are parallel and the carry is serial. Both outputs of a stage depend on its carry in, so a stage handed a late carry is late itself. The adder is only as quick as that single path.

A figure for the delay: each stage adds a fixed small delay to the carry path, so settling time grows in proportion to width: roughly n gate delays for n bits. Double the word and you double the wait.

The worst case crosses everything: add 1111 and 0001. The carry born at bit 0 is passed on by every stage and leaves only at the top, so each stage waited for all those below. The clock must cover that case every time.

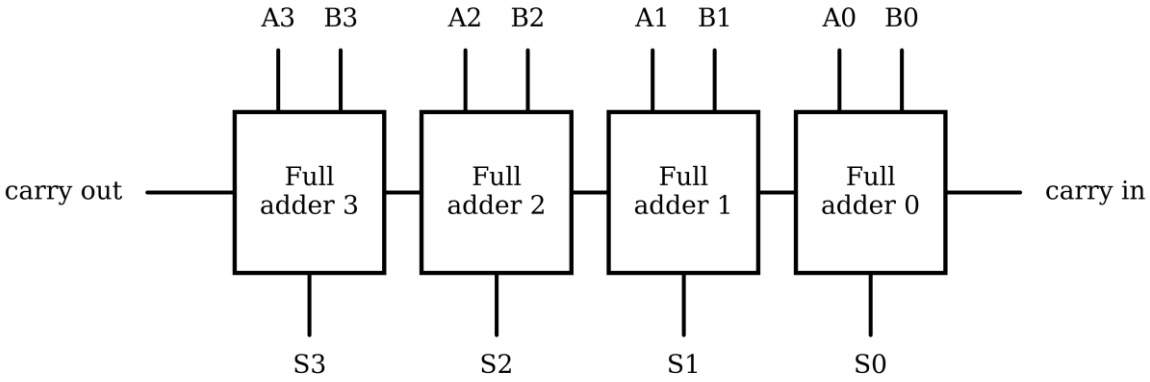


Figure 1.20: *A four-bit ripple carry adder. Each carry must settle before the next stage can finish.*

The ripple adder is the honest first answer, the smallest circuit that adds two whole numbers correctly. Everything faster is an attempt to stop the stages queueing.

The Master Key: *The bits arrive together and the carry arrives in single file, so delay grows with width and the clock pays for the longest carry.*

BINDER Question 83

A shopkeeper in Nzega owns one set of scales and weighs everything on it, whether goods are coming in or going out. A second machine that could only take weight away would be absurd. Processors own an adder and no subtractor at all.

Explain how a binary adder is made to subtract using two's complement, and state what the carry out means.

Model Answer

To compute A minus B, invert every bit of B, add 1, and add the result to A; in hardware the inversion is an XOR gate on each B input and the plus one is the carry into stage 0 set to 1, so one mode line turns the adder into a subtractor and the carry out reports whether a borrow was needed.

What the two's complement is: the negative of a binary number in the encoding processors actually use. Invert every bit and add one, and the pattern behaves as the negative of the original, because the two together give zero inside the register.

Inverting B costs one gate per bit: put an XOR gate on each B line and feed the mode line M to its other input. When M is 0 the XOR passes B through unchanged; when M is 1 it inverts every bit, because a bit XORed with 1 is its complement.

The plus one is free: there is no second adder and no increment step. Feed the same mode line into the carry in of stage 0. When M is 1 the chain adds an extra one at the bottom, exactly the plus one the complement demands.

Why one circuit does both: because the adder has no idea which operation it performs. It adds three bits per column and passes carries upward, and that is all. The meaning lives in the encoding, not in the hardware.

What the carry out means: for unsigned subtraction it is the inverse of a borrow. A carry out of 1 says A was not smaller than B and the sum lines hold the correct positive difference. A carry out of 0 says B was larger and the sum lines hold the complement of the true difference.

One mode line, one row of XOR gates and a carry in that is not tied down. That is the whole difference between adding and doing arithmetic.

The Master Key: *Subtraction is addition of the complement, the plus one enters as the carry in, and a carry out of 1 means no borrow.*

REAL Question 84

The electronics club at a Kibaha school has a four-bit adder on a breadboard, five lamps above it. Four are the sum and the fifth is the carry out. The argument is whether that fifth lamp belongs to the answer or warns about it.

Add the two operands stage by stage, showing the sum and carry produced at each stage, state the result and the carry out, and say whether overflow has occurred.

First operand A: 1011

Second operand B: 0110

Adder: four full adders in a ripple chain, stages numbered 0 at the least significant bit up to 3 at the most significant

Carry into stage 0: 0

Display: four lamps for the sum lines and one lamp for the carry out of stage 3

Model Answer

The sum lamps read 0001 and the carry lamp is lit, so the full answer is 10001, which is 17 in denary; seventeen will not fit in four bits, so read as unsigned numbers the addition has overflowed and the four sum lamps taken on their own are wrong.

Stage 0: A0 is 1, B0 is 0 and the carry in is 0. The three bits total 1, which is binary 01, so the sum line is 1 and the carry out of this stage is 0.

Stage 1: A1 is 1, B1 is 1 and the carry in is the 0 handed up from stage 0. The three bits total 2, which is binary 10, so the sum line is 0 and a carry of 1 goes upward.

Stage 2: A2 is 0, B2 is 1 and the carry in is 1. The three bits total 2 again, so the sum line is 0 and another carry of 1 goes up. This stage produced a carry only because one reached it from below, which is the propagate term at work.

Stage 3: A3 is 1, B3 is 0 and the carry in is 1. The total is 2 once more, so the sum line is 0 and the carry out of the adder is 1. There is no stage above to receive it, so it leaves on a lamp of its own.

Reading the result: the sum lines, from stage 3 down to stage 0, read 0001, and the carry out is 1, so the true total is 10001. Check it in denary: 1011 is 11, 0110 is 6, and 11 plus 6 is 17, which is 10001 in binary.

The unsigned verdict is overflow: seventeen needs five bits and the register holds four, so the result has run off the top of the word. For unsigned numbers the carry out of the most significant stage is precisely the overflow flag, and the lit lamp is the machine reporting its own failure honestly.

The signed verdict is no overflow: read the same patterns as two’s complement signed numbers and the story reverses. 1011 is minus 5, 0110 is plus 6, and the sum lines hold 0001, which is plus 1 and correct. Signed overflow is not the carry out; it is the case where the carry into the top stage differs from the carry out of it, and here both are 1.

Whether the fifth lamp belongs to the answer is settled by a convention agreed between people, not by anything inside the breadboard. The adder simply added, and it added correctly.

The Master Key: *Carry out is a wire, overflow is a reading of that wire, and the reading changes with the encoding you agreed to use.*

REAL Question 85

Every lorry at Tunduma is weighed twice, loaded and empty, and the driver is paid on the difference. The controller in the cabin has an adder and no subtractor, which sounds like an oversight until you watch what it does to the tare reading.

Compute A minus B by two's complement, showing the inversion, the added one and the stage by stage addition, then interpret the carry out.

Gross reading A: 1101

Tare reading B: 0101

Register width: four bits

The mode line is set to 1, so every bit of B is inverted by the XOR gates on the B inputs

The same mode line drives the carry into stage 0, so the carry in is 1

Model Answer

Inverting 0101 gives 1010, and adding that to 1101 with a carry in of 1 gives sum lines of 1000 and a carry out of 1; the carry out says that no borrow was needed, so the answer is the positive value 1000, which is 8, and 13 minus 5 is indeed 8.

Invert every bit of B: 0101 becomes 1010. The mode line is 1, so each XOR gate on the B inputs complements the bit passing through it, and the adder sees 1010 where the register still holds 0101. The inversion happens on the wire.

Add the one without adding it: the two's complement of 0101 is 1010 plus 1, which is 1011, but the adder never forms 1011 anywhere. The plus one enters as the carry into stage 0 and is absorbed into the ordinary rippling of the carry.

The lower two stages: stage 0 adds 1, 0 and the carry in of 1, a total of 2, so its sum line is 0 and it passes a carry of 1 upward. Stage 1 adds 0, 1 and that carry, again a total of 2, so its sum line is 0 and another carry goes up.

The upper two stages: stage 2 adds 1, 0 and the carry of 1, a total of 2, so its sum line is 0 and it too passes a carry up. Stage 3 adds 1, 1 and the carry of 1, a total of 3, which is binary 11, so its sum line is 1 and the carry out is 1.

$$1101 + 1010 + 1 = 11000, \text{so the sum lines read 1000 and the carry out is 1}$$

Interpreting the carry out: a carry out of 1 means no borrow was required, so the gross exceeded the tare and the sum lines hold the true difference as a positive number. That is 1000, which is 8, and in denary 1101 is 13, 0101 is 5, and 13 minus 5 is 8.

What a carry out of 0 would have meant: that the tare exceeded the gross, which at a weighbridge is a fault rather than a negative load. The sum lines would then hold the complement of the true difference and would need complementing back before a magnitude could be printed. A carry out of 0 is a borrow, and firmware should raise it rather than print it.

What the cabin did not have to buy: a subtractor. Four XOR gates, one mode line into the carry in, and the adder already in the controller does both jobs at the same speed for the price of one extra wire.

The driver is paid on a number the machine reached by adding. Nothing in the cabin ever took anything away from anything, and the ticket is still right.

The Master Key: *Invert B, let the carry in supply the plus one, and read a carry out of 1 as the guarantee that the difference is positive.*

REAL Question 86

Three sentences are pinned to a classroom wall at Masasi, copied out in good faith from borrowed notes. Each is a real idea remembered slightly wrong, the most durable kind of error there is, because it fails only when it is put to work.

Identify the error in each of the three claims and give the correct statement.

1. "The carry out of a full adder is simply A.B."

2. “A four-bit ripple carry adder is four times faster than a one-bit adder.”

3. “Overflow and carry out are the same thing.”

Model Answer

All three are wrong: the carry out of a full adder is A.B + Cin.(A XOR B), because a carry can also be born from the carry below; a four-bit ripple adder is about four times slower than a single stage, not faster, because its stages are serial in the carry; and carry out equals overflow for unsigned numbers only, while signed overflow means the carry into the top stage differs from the carry out.

Claim one, what has been dropped: A.B is the carry of a half adder, right only where the carry in is 0. The row A = 0, B = 1, Cin = 1 destroys it: A.B is 0, but 0 plus 1 plus 1 is binary 10, so the true carry out is 1.

Claim one corrected: Carry out = A.B + Cin.(A XOR B). A carry appears in two ways: generated when A and B are both 1, and propagated when exactly one of them is 1 and a carry comes from below. The missing term is the propagated one, and it is what makes chaining work.

Claim two, the direction of the error: width has been confused with speed. Four bits per addition is four times as much arithmetic in one operation, which says how much work is done, not how quickly. Speed is time, and time is what the extra stages cost.

Claim two corrected: it is about four times slower than a single stage, because the stages are serial in the carry: stage 3 cannot settle until stage 2 has. The wider adder wins on arithmetic per operation and loses on the delay of each one.

Claim three, why the two are confused: because they agree in the commonest case. Carry out is a physical signal leaving the top stage on a wire. Overflow is a judgement that the answer would not fit, and nobody can make it until the encoding is stated.

Claim three corrected for unsigned numbers: here the two genuinely coincide. If the operands are unsigned, a carry out of 1 says the result needed a bit the register does not have, and that is overflow exactly.

Claim three corrected for signed numbers: here they part. For two’s complement, overflow has occurred when the carry into the top stage differs from the carry out of it. Add 1011 and 0110: both are 1, so there is no signed overflow, and 0001 is the right answer to minus 5 plus 6.

Two of these errors cost a mark each. The third costs a working design, because a flag read under the wrong convention is worse than no flag.

The Master Key: *Keep the propagate term, remember a longer chain is slower, and never read carry out as overflow until the encoding is stated.*

HOT Question 87

A guest lecturer tells the technicians at a workshop in Newala that the ripple carry adder is a museum piece. On the bench are three boards with ripple adders inside them, all working, all built this year. Both are describing the same circuit correctly.

“The ripple carry adder is obsolete.” To what extent do you agree?

Model Answer

Obsolete is the wrong word: no processor built for speed uses a ripple carry adder, because lookahead and carry select designs settle in a time that grows with the logarithm of the width rather than the width itself, yet the ripple adder is still the smallest and cheapest arithmetic circuit there is, quick enough at small widths and modest clocks, and the design every faster adder is measured against.

For the claim, the delay grows the wrong way: the settling time of a ripple adder is proportional to its width, so doubling the word doubles the wait. At sixty four bits a carry may cross sixty four stages in sequence. A lookahead adder computes the carries for whole groups at once from generate and propagate terms.

For the claim, the adder sets the clock: a synchronous design must have a clock period longer than its slowest path, and in most datapaths the adder is that path. A wide ripple adder does not merely add slowly; it clocks down the whole chip.

For the claim, the performance market has voted: the arithmetic units of general purpose processors use lookahead, carry select, carry skip or hybrids. Nobody designing for speed chooses ripple, and a design abandoned by everyone competing on performance has a fair claim to being superseded there.

Against the claim, it costs the fewest gates: one full adder per bit and not one gate more. A lookahead adder must build a generate and a propagate signal for every bit and a tree of logic above them, which costs area, transistors, current and design time.

Against the claim, small and slow hardware is most hardware: an eight-bit controller in a water meter or a thermostat has a clock period many times longer than eight stage delays. The ripple adder finishes with time to spare, and speed bought beyond the clock period is thrown away.

Against the claim, it is the baseline: the merit of a lookahead adder is quoted as a saving over ripple and its cost as an overhead above it. It is the first adder taught, the first simulated and the first draft of most designs.

The judgement: agree about performance and reject the word. Obsolete means superseded everywhere and kept only from habit, and the ripple adder is superseded only where width and clock speed make its delay bite. Say instead that it is outclassed wherever speed is the binding constraint, and the sentence becomes true and useful.

The lecturer is right about processors, the technicians are right about their bench, and between them stands a word that claims more territory than the evidence supports.

The Master Key: *Ripple is outclassed on speed and unbeaten on cost, so it is retired from fast datapaths and still shipping everywhere else.*

HOT Question 88

A design house at Karatu is finishing a controller for the solar irrigation pumps nearby. Eight-bit datapath, four megahertz clock. Half the team wants ripple carry because it is small, the other half carry lookahead because it is fast. Neither has mentioned the workload.

Evaluate the choice between a ripple carry adder and a carry lookahead adder for this controller, weighing gate count and cost against speed, and reach a judgement.

Model Answer

Choose the ripple carry adder: eight stages settle in roughly sixteen gate delays, a small fraction of the 250 nanosecond period a four megahertz clock allows, so lookahead would buy speed this controller cannot spend while charging gates, area and current in every one of tens of thousands of units; the judgement reverses the moment the word grows wider or the clock grows faster.

The case for lookahead, speed that scales: a lookahead adder derives every carry in parallel from generate and propagate terms instead of waiting for the stage below, so its delay grows with the logarithm of the width. On a thirty two bit datapath that is the difference between a usable processor and an unusable one.

The case for lookahead, headroom: products change. A faster clock, a wider accumulator, one more term in the loop, and a design with no timing margin must be reopened. Speed bought early can be cheaper than a redesign under deadline.

The case for ripple, gate count: one full adder per bit at about five gates each, so roughly forty gates for the whole adder and nothing else. The lookahead version adds generate and propagate logic for every bit plus a tree above it, which roughly doubles the block at eight bits.

The case for ripple, cost multiplied by volume: the extra area is paid once per unit and the order is tens of thousands of units. It is paid in current too, and this product runs from a solar panel and a battery, where current is the tightest constraint.

Put the timing against the workload: four megahertz gives a period of 250 nanoseconds. Eight ripple stages at about two gate delays each is sixteen gate delays, and at two nanoseconds a gate that is about 32 nanoseconds. The adder finishes with more than 200 nanoseconds to spare.

Where the judgement reverses: change one number and the answer changes. Give the same team a thirty two bit processor at one hundred megahertz and the chain needs about sixty four gate delays against a ten nanosecond period, which no plausible gate speed can meet. Lookahead is then the only design that works.

The judgement: specify the ripple carry adder and write the reasoning into the design note. It is chosen not because it is the better circuit but because at eight bits and four megahertz the speed it lacks is speed this workload never asks for. Record the trigger: past sixteen bits or twenty megahertz, reopen it.

The two halves of the team were arguing about adders when the answer sat in the specification the whole time. Speed is worth buying up to the clock edge and no further.

The Master Key: *Match the adder to the clock period and the unit volume, and state the width and speed at which your answer would flip.*

Multiplexers, decoders, encoders and comparators

Four blocks sit between the gates and the machine, and not one of them is new logic. A multiplexer chooses, a decoder points, an encoder reports and a comparator judges, and each is only an arrangement of gates that has earned a name. This section derives all four from the gates you already have, fixes the counting rule that two to the power n governs every one of them, and spends the result on a water scheme at Handeni and a design bench at Kibaha.

BINDER Question 89

At Chalinze several roads come together and become one, and the officer standing at the mouth of the junction decides which line of traffic moves. Nothing about the vehicles changes as they pass him. Only the road is being handed out.

Explain what a multiplexer does, derive the expression for a two to one multiplexer, state the general rule for select lines, and give one real use.

Model Answer

A multiplexer is a selector: it takes several data inputs and passes exactly one of them to a single output, with select lines deciding which. For the two to one case Y = A.NOT S + B.S, so the output follows A when S is 0 and B when S is 1, and in general n select lines choose among two to the power n inputs.

What the block does: it chooses. Data arrives on several lines and one line leaves, while the select lines name the winner. A multiplexer never alters a bit; it only decides which bit travels.

Deriving the two to one expression: take inputs A and B and one select line S. The output must be A when S is 0, the term A.NOT S, and B when S is 1, the term B.S. Only one term can be high at once, so an OR collects them: Y = A.NOT S + B.S.

Reading the circuit: the inverter makes NOT S, one AND gate guards A with NOT S, the other guards B with S, and the OR passes whichever gate stands open. Hold S at 0 and the lower AND is dead, so Y follows A.

The counting rule: each select line doubles the choices, so n select lines address two to the power n inputs. One line chooses between two, two lines among four, three among eight. The input count is therefore always a power of two.

Where it earns its place: one line shared among several sources. Rather than run a wire from every sensor back to a controller, a multiplexer sits at the sensors while the controller drives the select lines, reading each in turn down one conductor.

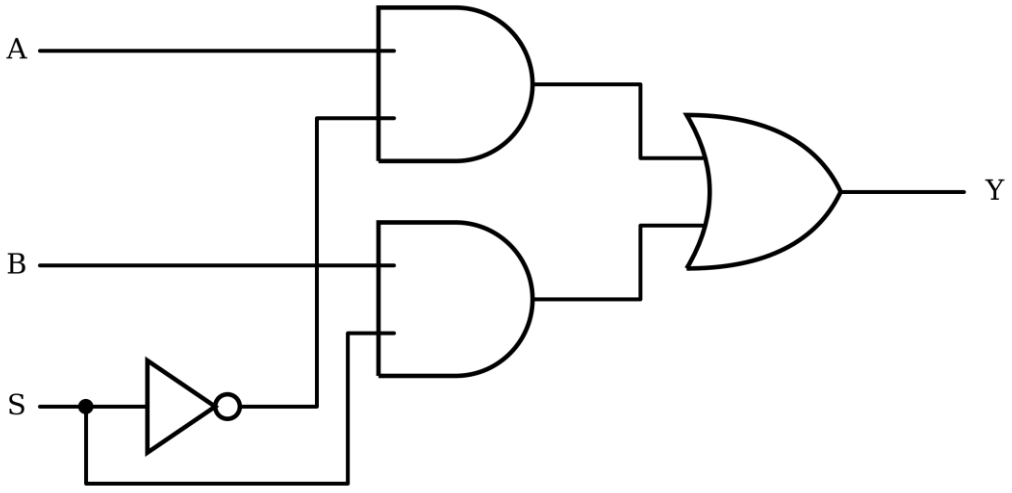


Figure 1.21: *A two to one multiplexer. Y = A.NOT S + B.S, so the output follows A when S is 0 and B when S is 1.*

One officer, several roads and one carriageway, and the whole of the algebra fits on a single line.

The Master Key: *Y = A.NOT S + B.S for the two to one case, and n select lines always choose among two to the power n inputs.*

BINDER Question 90

The auctioneer at a Bariadi cattle auction calls a pen number and one gate swings open. Not two, and never none: the number names a single pen and every other pen stays shut. That is the exact reverse of the officer at the junction.

Explain what a decoder does, state clearly how it differs from a multiplexer, and give one real use.

Model Answer

A decoder takes an n-bit code on its inputs and raises exactly one of its two to the power n outputs, the one whose number matches the code. That is the multiplexer read backwards: a multiplexer has many data inputs and one output, while a decoder has few inputs and many outputs, exactly one of them high.

Exactly one output, always: give a two to four decoder the code 10 and Y2 rises while the other three stay low. Every code lights one output and none lights two, because codes and outputs match one for one.

How each output is built: output Yi is the AND of the input bits in the pattern that names i, complemented wherever that bit is 0. So Y0 = NOT A.NOT B, Y1 = NOT A.B, Y2 = A.NOT B and Y3 = A.B. Two inverters and four AND gates build the block.

The difference from a multiplexer: a multiplexer answers which of these inputs shall I pass, and yields data. A decoder answers which output shall I point at, and yields a position. One narrows many lines to one, the other opens one out of many.

Reading Table 1.21: the A and B columns count up in binary. Read across any row and a single 1 appears among the four outputs, walking one place right as the code advances. That diagonal of ones is the decoder.

Where it earns its place: address decoding. A board with four memory chips hands the top two address bits to a decoder, and whichever chip owns that region is selected while the other three ignore the bus. The same block lights one display digit at a time.

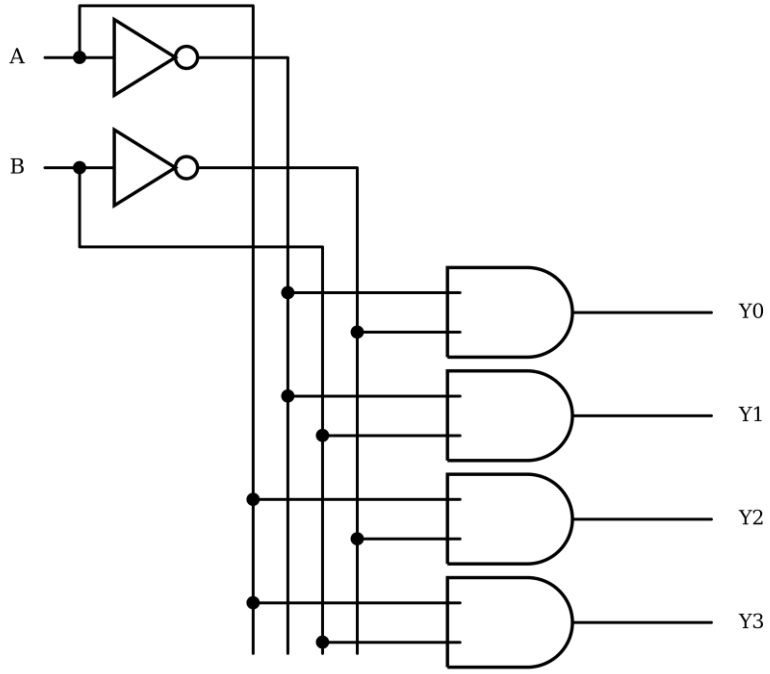


Figure 1.22: *A two to four decoder. Exactly one output is high for each input combination.*

A	B	Y0	Y1	Y2	Y3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Table 1.21: *A two to four decoder.*

The auctioneer names a number and one gate opens, and the decoder is that sentence written in gates.

The Master Key: *A decoder raises exactly one of two to the power n outputs, running opposite to a multiplexer: many outputs from few inputs, not one output from many.*

BINDER Question 91

A keypad at a Masasi fuel station has twelve keys, and the pump controller behind the panel has four wires waiting for them. Something between them must turn a pressed key into a number. Then a driver leans on two keys at once.

Explain what an encoder does, state the problem a plain encoder has, and explain how a priority encoder solves it.

Model Answer

An encoder is a decoder run backwards: it takes two to the power n input lines, expects exactly one to be high, and produces the n-bit code naming that line. Two simultaneous highs merge into a meaningless code, which a priority encoder cures by ranking the inputs and reporting only the highest ranked active one.

What the block does: eight lines in, three lines out. Raise input 5 and the outputs read 101. The encoder counts nothing and measures nothing; it names the active line in binary, which is why n bits serve two to the power n inputs.

How it is built: each output bit is an OR of every input whose number carries a 1 in that position. For an eight to three encoder the least significant output is D1 + D3 + D5 + D7. Three OR gates are the whole block.

Where the plain encoder fails: raise D3 and D5 together and the OR gates merge their codes, 011 with 101, so the output reads 111 and names line 7. Nothing is flagged; the circuit reports a line nobody touched, which is worse than an error because it looks like data.

How priority repairs it: rank the inputs and let the highest ranked active one win. Each input is ANDed with the complement of every input above it, so D5 may speak only while D6 and D7 are silent. Two keys together yield one truthful code.

The keypad, and one spare output: a pressed key drives one line, the encoder turns it into a code the controller stores, and a thumb across two keys gives the higher rather than a third key nobody pressed. A valid output separates line 0 from nothing at all.

Twelve keys, four wires and one thumb across two of them: the ranking is not a refinement, it is the design.

The Master Key: *A plain encoder assumes exactly one input is high; a priority encoder ranks the inputs so that two at once still give a truthful code.*

BINDER Question 92

The tank on the hill above Same fills from a borehole and empties into the town, and the pump beneath it needs one fact: whether the measured level has reached the level it was told to hold. Not by how much, and not why.

Explain what a magnitude comparator does, give the one-bit expressions for its three outputs, and explain how comparison is carried out on multi-bit numbers.

Model Answer

A magnitude comparator takes two binary numbers and produces three outputs, A equals B, A greater than B and A less than B, of which exactly one is high at any time. For a single bit, equality is A XNOR B, greater is A.NOT B and less is NOT A.B.

Three outputs, one of them true: the three answers are exhaustive and exclusive. Two numbers are equal or one is the larger, and there is no fourth case, so the three lines always carry exactly one 1.

The one-bit expressions: A XNOR B is high when the bits agree, which is equality. A.NOT B is high only where A is 1 and B is 0, which is A the greater, and NOT A.B is its mirror. Three expressions cover the four rows.

Reading the circuit: the XNOR sits across both inputs and reports agreement directly. Below it, B passes through an inverter before meeting A at an AND, so that gate fires only where A is 1 and B is 0. The mirror of that pair gives the third output.

Multi-bit comparison starts at the top: compare the most significant bits first. If they differ the answer is settled there, because that bit outranks every bit beneath it put together. Only when the top pair agree does comparison move down.

Why the chain runs on equality: each stage may speak only if every stage above reported agreement. So a four bit greater than output reads A3.NOT B3 + (A3 XNOR B3).A2.NOT B2 and onwards down the word, while equality is the AND of all four XNOR outputs.

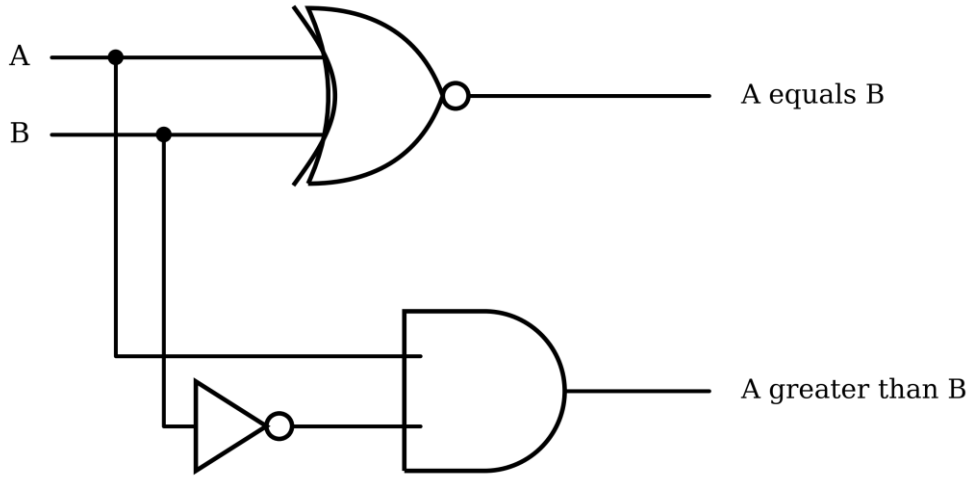


Figure 1.23: *A one-bit comparator. XNOR detects equality and A.NOT B detects that A is the greater.*

The pump on the hill needs one of the three lines, and the circuit hands over the other two for nothing.

The Master Key: *Equality is A XNOR B and greater is A.NOT B, and multi-bit comparison begins at the most significant bit and moves down only while the bits above agree.*

REAL Question 93

A rural water scheme outside Handeni draws from twelve boreholes scattered across kilometres of bush, each carrying a pressure sensor on its rising main. The telemetry unit at the pump house has one data input, and a second cable costs more than the sensors.

Identify the block this scheme requires, justify the choice, calculate how many select lines are needed, and give the expression for the output.

Twelve boreholes, each fitted with one pressure sensor.

Each sensor drives a single line: 1 while pressure is healthy, 0 once it has fallen.

The telemetry unit at the pump house has one data input line.

The unit can drive as many select lines as the chosen block requires.

The scheme expects to add up to three more boreholes in its next phase.

Model Answer

The scheme needs a multiplexer, because twelve sources must share one input line and only a multiplexer narrows many inputs to one under external control. Since two to the power 3 is 8 and two to the power 4 is 16, four select lines are required, which means a 16 to 1 multiplexer with four inputs left over.

Why a multiplexer and not the others: the direction of the problem decides it. Twelve lines must become one, which is narrowing, and narrowing is what a multiplexer does. A decoder widens, turning a code into one of many outputs, and an encoder would report only which sensor is active. The multiplexer alone carries the reading itself.

Sizing it, the calculation: n select lines address two to the power n inputs, so find the smallest n for which two to the power n is at least 12. Three lines give 8, which falls short. Four lines give 16, which covers it with room. Four select lines it is.

What the spare inputs are worth: 16 inputs for 12 sensors leaves four unused, and the next phase wants three more boreholes. The block already has room for them, so the expansion costs a wire instead of a redesign. Tie the remaining inputs low, so a stray select never reads a floating line.

The expression: Y = D0.(NOT S3.NOT S2.NOT S1.NOT S0) + D1.(NOT S3.NOT S2.NOT S1.S0) and so on, up to D15.(S3.S2.S1.S0). Every data input is guarded by the one select pattern that names it, and the terms are ORed together. Only one guard can stand open at a time.

How the unit uses it: the telemetry unit counts 0000 up to 1011 on the select lines and samples the data line after each step, so twelve boreholes are polled down one conductor in twelve steps. A sensor reporting 0 is the one the pump house acts on.

Why not simply run twelve wires: because the cost lives in the trench and not in the copper. Every run needs digging, ducting and a gland at both ends, and every one is a fault waiting for a hoe. One data line and four shared select lines make five conductors where twelve were proposed.

The block was named by the direction of the problem and sized by one inequality, and both took less time than the quotation for the trench.

The Master Key: *Many inputs into one output means a multiplexer, and the select count is the smallest n for which two to the power n is at least the number of sources.*

REAL Question 94

Three sentences come out of a Karatu study group, written in different hands and all three underlined for emphasis. Each is close enough to the truth to survive a whole term of reading. A claim that is nearly right gets copied forward.

Identify each of the three faulty claims and correct it.

1. “A multiplexer with three select lines can choose among six inputs.”

2. “A decoder and a demultiplexer are completely different circuits with nothing in common.”

3. “To compare two four-bit numbers you compare the least significant bits first.”

Model Answer

All three fail: three select lines give two to the power three, which is eight and not six, and the input count is always a power of two; a demultiplexer is a decoder fed with data on its enable, so the two share their whole structure; and magnitude comparison begins at the most significant bit, because that bit outranks every bit beneath it.

The first error, multiplying where the rule doubles: three has been multiplied by two to give six. Select lines do not add choices, they double them, so three lines give two times two times two, which is eight. Corrected, a multiplexer with three select lines is an eight to one.

Why the count is always a power of two: every combination of the select lines must name one input, and n lines have two to the power n combinations. Six is not such a number, so no multiplexer has six inputs. A design needing six takes an eight to one and ties two inputs low.

The second error, two names for one arrangement: a demultiplexer takes one data input and sends it to one of many outputs, chosen by select lines. Draw it and you have drawn a decoder: the same inverters, the same AND gates, one output high. Corrected, they share their whole structure.

Where their single difference lies: in the enable input. A decoder has an enable normally held high, and the block falls silent when it drops. Feed data into that enable and each AND gate passes that data on the output the code selects, so one pin changes role and the block changes name.

The third error, starting at the wrong end: in a four bit number the least significant bit is worth 1 and the most significant is worth 8, so nothing at the bottom can overturn what happens at the top. Corrected, comparison starts at the most significant bit and works down.

One example settles it: compare 1000 with 0111. The least significant bits are 0 and 1, so a comparison begun at the bottom calls B the greater, when A is 8 and B is 7. The top bit decided the answer before any other was read.

The habit all three share: each swaps a rule for something that feels like it. Doubling becomes multiplying, nearly identical becomes completely different, an ordered word becomes a loose bag of bits. Ask what a rule counts and in what order, and all three come apart.

Three underlined sentences, and not one survives a single worked example.

The Master Key: *Select lines double the choices, a demultiplexer is a decoder with data on its enable, and comparison always starts at the most significant bit.*

HOT Question 95

A workshop instructor at a Moshi technical college keeps a drawer of NAND chips and a habit of calling everything else decoration. His students can derive a decoder from first principles in ten minutes, which is his argument and also the flaw in it.

"These building blocks are just combinations of gates, so learning them separately is a waste of time." To what extent do you agree?

Model Answer

The claim is true about the hardware and false about the work: every block here is nothing but gates, and any of them can be rederived from a truth table, but the blocks are the units in which designers think, specify and buy, and rebuilding a decoder whenever you need one spends the effort that naming it exists to save.

For the claim, nothing new hides inside: open a decoder and you find inverters and AND gates; open a multiplexer and you find AND gates and an OR. No block introduces a function the gates cannot already express, so whoever holds the gates has met all four before.

For the claim, derivation outlasts recall: whoever can build a two to four decoder from its truth table can build a three to eight decoder that was never taught. Whoever memorised the block as a rectangle with pins is stranded the moment the question changes shape, and questions change shape.

For the claim, the list of blocks never closes: real designs also use priority encoders, parity trees, barrel shifters and a hundred arrangements with names of their own. If each must be learnt separately the learning has no end, whereas the gates underneath do not increase in number at all.

Against the claim, the block is how work is specified: nobody orders forty gates. A drawing calls for a 16 to 1 multiplexer, a supplier quotes a part number against it, and a technician at a pump house reads the same words on the board in front of him.

Against the claim, the name carries the intention: a diagram showing a decoder announces at a glance that one thing out of many is being chosen. The identical circuit in loose gates says nothing until it has been traced. Naming is not decoration, it is how a design stays readable to whoever inherits it.

Against the claim, rederiving is paid again and again: ten minutes on every occasion, plus the mistakes ten minutes attracts, against nothing at all for a block already understood and verified. Engineering economises on attention, and a solved problem deserves to be solved once.

The judgement: agree that these blocks are only gates, and reject the conclusion drawn from it. The claim confuses what a thing is made of with how it should be handled. Learn the derivations, because they answer questions the names do not cover, then learn the names, because that is the language every drawing is written in.

The instructor is right about the drawer and wrong about the drawing.

The Master Key: *Everything here is gates, yet the block is the unit of thought, purchase and repair, so derive it once and use the name thereafter.*

HOT Question 96

A bench at Kibaha has a function of three variables on a scrap of card and two ways to build it. One is a handful of gates from simplification; the other a single eight to one multiplexer wired straight from the truth table.

Evaluate how a designer should choose between implementing a function with a multiplexer and implementing it with gates, and reach a judgement.

Model Answer

The choice turns on which cost dominates. A multiplexer with n select lines implements any function of n variables directly, since each data input is tied to the value that row of the truth table demands, which removes simplification and its errors; gates are usually smaller, faster and cheaper once the function is reduced. The multiplexer buys certainty, the gates buy resources.

For the multiplexer, the table is the wiring: put the three variables on the select lines and tie each of the eight data inputs to the value that row requires. No algebra, no map, no dropped term. The function is transcribed rather than derived, and transcription is the safer operation.

For the multiplexer, a change costs one wire: revise the specification and the repair is to move a single data input from 0 to 1. A gate design must be simplified afresh and often redrawn, so wherever requirements are still moving the multiplexer is the cheaper thing to own.

For the multiplexer, one part replaces several: a single device instead of an assortment of AND, OR and NOT packages means fewer components, fewer solder joints and one propagation delay to characterise rather than a network of them.

Against it, simplification is thrown away: a great many functions collapse to two or three gates. Spending a whole eight to one block, its select wiring and its tied inputs on a function that reduces to A.B + C is hiring a lorry to carry a bucket.

Against it, the meaning is hidden: gates drawn from a simplified expression show the shape of the condition, so a reader sees that two faults are being ORed. A multiplexer shows a rectangle with a pattern of highs and lows on its inputs, which is correct and says nothing.

Against it, the block scales with variables: every extra variable doubles it, so six variables demand a 64 to 1 device. Gates grow with the complexity of the function, while the multiplexer grows with the count of variables whether the function is complicated or trivial.

The judgement: let the number of variables and the stability of the specification decide. Where a small function is settled and reduces well, build it from gates and keep its meaning visible. Where a function of few variables is irregular or still being argued over, take the multiplexer and buy certainty with silicon, the cheaper of the two things on that bench.

The scrap of paper does not say which to build, but it does say how many variables there are, and that is most of the answer.

The Master Key: *A multiplexer with n select lines implements any function of n variables straight from its truth table, so choose it when the function is irregular or unsettled and choose gates when it reduces small and must stay readable.*

Sequential logic: flip-flops, registers, counters and state tables

Every circuit in this chapter so far has forgotten everything the instant its inputs changed. This section gives logic a memory, and the whole cost is one wire: run an output back to an input and the circuit begins to depend on what it already holds. From that single bend come the latch, the flip-flop, the register and the counter, and from those comes a junction at Biharamulo whose lights know which colour they showed last.

BINDER Question 97

A clerk at a Mvomero ward office keeps two things. One is a rubber stamp, printing the same thing on whatever it meets and remembering nothing. The other is a tally sheet, where what he writes next depends on what is already there.

Distinguish combinational logic from sequential logic, and explain what makes memory possible in a circuit built from gates that have no memory of their own.

Model Answer

Combinational output depends only on the inputs present at this instant, while sequential output depends on those inputs and on a stored state as well, and the state is created by feedback: run one output back to an input and the circuit describes its own past.

Combinational logic has no yesterday: an adder given 1011 and 0110 produces the same sum today, tomorrow and on the ten thousandth attempt. Nothing inside it records what it did a moment ago, so nothing inside it can count, store or wait.

Sequential logic asks a second question: such a circuit asks what are my inputs and also what am I holding. The same inputs applied twice may give different outputs, because the stored state moved in between. That is not a defect. It is the entire capability.

Feedback is the trick, and the only trick: run an output back into an input and the circuit is solving an equation about itself. Two stable answers exist, the circuit can rest in either, and resting is remembering. No new component is bought. Only a wire that goes the wrong way.

Why one wire is enough: a gate holds its output for as long as its inputs hold. Arrange the feedback so the output sustains the very inputs producing it, and the pair props itself up. Take the driving signal away and nothing falls, because each half now holds the other.

What the memory costs: timing. A combinational circuit is read down the columns of a truth table, while a sequential one must be read as a sequence, with a present state and a next state, because the answer depends on the order events arrived in.

The stamp and the tally sheet are both made of ink. Only one of them is keeping records.

The Master Key: *Combinational output depends on the present inputs alone, sequential output depends on the present inputs and the stored state, and feedback is what creates the state.*

BINDER Question 98

Two watchmen at a Muheza citrus store share one lamp and one rule: whoever speaks last decides. Say set and the lamp stays lit long after you stop speaking. Say reset and it stays out. Say both and the lamp obeys neither man.

Describe the SR latch, name its inputs, state its behaviour for every combination of those inputs, and explain why one of the four combinations is forbidden.

Model Answer

The SR latch is two NOR gates with each output wired back into the other gate's input, which gives one bit of memory: S sets Q to 1, R resets Q to 0, both low holds the stored value, and both high is forbidden, because it drives Q and NOT Q to 0 and leaves the next state to a race.

What the figure shows: two NOR gates facing the same way. R enters the upper gate, whose output is Q; S enters the lower gate, whose output is NOT Q; and each output crosses over to become the second input of the other gate.

Holding, with both inputs low: a NOR outputs 1 only when both inputs are 0. Suppose Q is 1: the lower gate sees that 1 and outputs 0, so the upper gate sees two zeros and confirms Q. Suppose Q is 0 and the same reasoning confirms it. Both states are stable, so the latch holds.

Setting and resetting: raise S and the lower gate is forced to 0, so NOT Q falls, the upper gate sees two zeros, and Q rises to 1. Raise R instead and the mirror image drives Q to 0. Either way the latch keeps the new value after the input goes away.

Why both inputs high is forbidden: both gates are forced low at once, so Q and NOT Q are both 0 and the outputs have stopped being complements. Worse, when the inputs return to 0 the gates race to reassert themselves, and the winner is whichever input fell last, settled by gate delays rather than by any rule. The latch is not broken, only unpredictable, which in a design is worse.

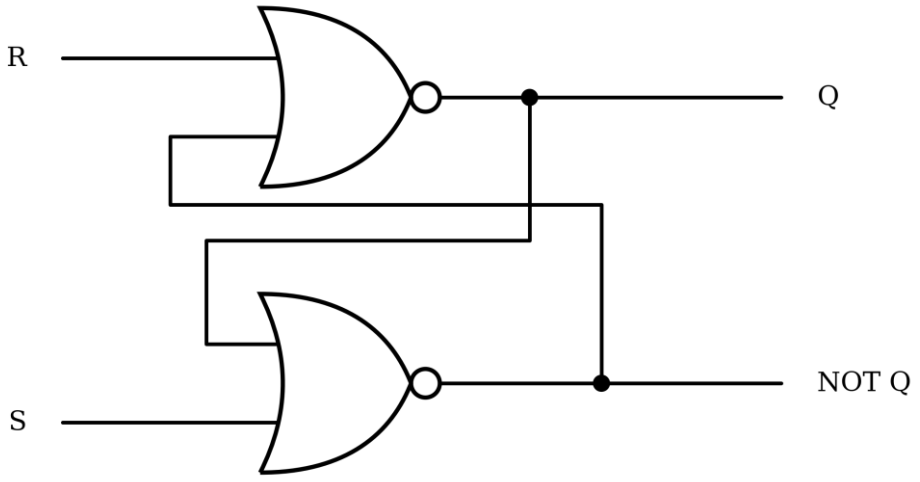


Figure 1.24: *An SR latch from two cross-coupled NOR gates. The output feeds back and becomes the memory.*

S	R	Q next	Comment
0	0	Q, unchanged	Holds the value already stored
1	0	1	Sets Q to 1
0	1	0	Resets Q to 0
1	1	Undefined	Forbidden: both outputs are driven to 0

Table 1.22: *The SR latch.*

Two gates, two wires crossed over, and the first thing in this chapter that can be said to remember.

The Master Key: *S sets, R resets, both low holds, and both high is forbidden because it drives Q and NOT Q to 0 and leaves the outcome to a race.*

BINDER Question 99

A tea factory at Rungwe runs on a bell. Nobody carries a crate when the mood takes them; they carry it when the bell sounds. Take the bell away and every station works at its own pace, until two crates reach the platform together.

Explain what a clock does in a sequential circuit, distinguish level triggering from edge triggering, and describe the D flip-flop, saying why it cannot reach a forbidden state.

Model Answer

A clock is a signal that alternates between 0 and 1 and tells the circuit when it may change; a level triggered device accepts its input the whole time the clock is high, an edge triggered device only in the instant the clock crosses from 0 to 1, and the D flip-flop copies D to Q at that instant and holds it.

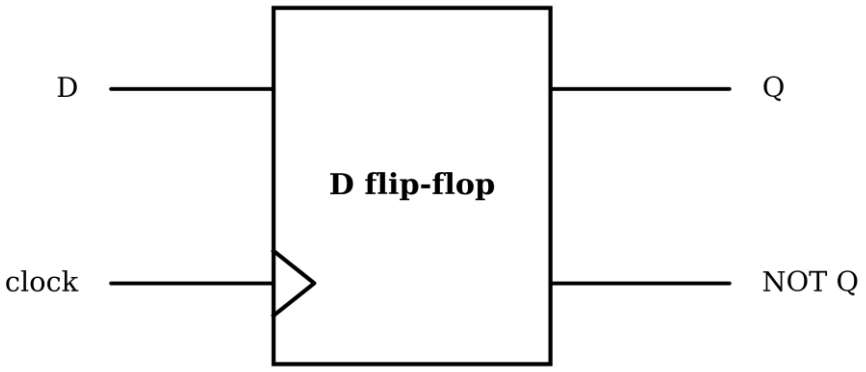
What the clock is for: it is not power and it carries no data. It is a square wave at a fixed rate whose only duty is to say now. Without it, changes arrive whenever the wires deliver them and circuits race one another to the answer.

Level triggering, open for a while: a level triggered latch is transparent while its enable is high. Whatever the input does inside that window, the output copies at once, so an input that wobbles three times gives an output that wobbles three times.

Edge triggering, open for an instant: an edge triggered flip-flop samples its input at the rising edge and ignores it for the rest of the cycle, so the output changes at most once per clock period. That restriction is what lets one stage feed the next safely.

The D flip-flop: one data input D, one clock input, two outputs. Q takes the value of D at the edge and NOT Q is its complement for free. Set D before the edge and Q becomes that value; leave the clock alone and Q does not move.

Why there is no forbidden state: the SR latch can be told two contradictory things because it has two independent inputs. The D flip-flop has one, so there is nothing to contradict and every value of D is lawful. That is why it is the workhorse of real designs.



Q takes the value of D at the clock edge, and holds it until the next one

Figure 1.25: *A D flip-flop. Q takes the value of D at the clock edge and holds it until the next one.*

One input, one bell, one answer per stroke, and no way to ask for something impossible.

The Master Key: *A clock says when, an edge narrows when to an instant, and the D flip-flop stores one bit through one input, so no illegal combination exists.*

BINDER Question 100

A coffee cooperative at Bukoba kept a switch on its drying shed wall, and somebody found a fourth thing to do with it: press both ends at once. On the old switch that broke it. On the new one it reverses the state.

Describe the JK flip-flop and say how it repairs the fault in the SR latch, then explain what a register is and state how many flip-flops an n-bit register needs.

Model Answer

The JK flip-flop carries the same two controls as an SR latch, renamed J and K, but the forbidden combination is given a meaning instead: J = 1 with K = 1 toggles, so Q becomes NOT Q at every clock edge; a register is then n such flip-flops sharing one clock and holding an n-bit word.

The four instructions: J = 0 with K = 0 holds. J = 1 with K = 0 sets Q to 1. J = 0 with K = 1 resets Q to 0. J = 1 with K = 1 toggles, so Q becomes its own opposite. The first three are the SR latch; the fourth is the repair.

Why toggling is a lawful answer: the trouble with S = 1 and R = 1 was that it asked for Q and NOT Q to be equal, which is impossible, so the circuit fell into nonsense. Toggle asks for something possible: swap the two outputs.

What toggling buys: hold a flip-flop with J = 1 and K = 1 and it changes state at every edge, completing one cycle of its own for every two clock cycles. That is division by two, the material a counter is made from.

A register is flip-flops in parallel: set eight D flip-flops side by side and tie all eight clock inputs to one line. Each owns one bit position, and the shared clock captures every bit in the same instant, so the word arrives whole rather than digit by digit.

The counting rule, with no discount: an n-bit register needs n flip-flops, one for each bit: sixteen bits need sixteen, and no cleverness reduces it. That is why register width is a genuine cost in silicon.

The move that broke the old switch is now the move that counts the sacks.

The Master Key: *JK is the SR latch with the forbidden combination redefined as toggle, and an n-bit register is n flip-flops sharing one clock.*

REAL Question 101

The fish landing at Kilwa counts crates onto the lorry with a man, a piece of chalk and a wall, and the count comes out wrong about once a morning. The cooperative wants a beam across the loading door and a display above it.

Explain how a ripple counter counts, set out the state sequence of a three-bit counter, and calculate how many flip-flops are needed to count at least 200 crates.

Model Answer

The counter is a chain of T flip-flops, each clocked by the output of the one before it, so Q0 changes at every pulse, Q1 at every second pulse and Q2 at every fourth, and read together Q2 Q1 Q0 count 000 up to 111; since n flip-flops reach 2 to the power n minus 1, counting at least 200 crates needs 8 flip-flops, because 7 reach only 127 while 8 reach 255.

Toggling is the whole mechanism: a T flip-flop with T held at 1 flips its output at every edge it receives. Nothing works out what the next value ought to be, because the next value is simply the opposite of the present one, which is exactly what adding one to a single bit does.

Why the chain divides: Q0 completes one full cycle for every two incoming pulses, so it is the clock divided by two. Feed Q0 into the second flip-flop and Q1 divides again, giving four, and Q2 gives eight. Halved rates are what place value looks like drawn in time.

Following one awkward pulse: at count 3 the state is 011. The next pulse toggles Q0 down to 0, that falling output toggles Q1 down to 0, and that in turn toggles Q2 up to 1, so 011 becomes 100. One pulse in, three changes out, and the carry has walked up the chain.

Reading Table 1.24: the Q0 column alternates every row, Q1 every two rows and Q2 every four. That is not an accident of this table; it is what binary counting is, and a counter of any width shows the same doubling in its columns.

Sizing the counter for the crates: n flip-flops hold 2 to the power n patterns, so they count from 0 up to 2 to the power n minus 1. Seven reach 127, which falls short of 200. Eight reach 255, which covers it with room over. Eight is the answer, and there is no seven and a half.

What the landing site ends up with: the beam delivers one pulse per crate, the eight flip-flops hold the running total, and a reset line clears them to 00000000 as each lorry backs up. The chalk is replaced by a chain of toggles, and the count stops depending on anybody's morning.

each stage toggles on its neighbour, so Q0 divides by two, Q1 by four, Q2 by eight

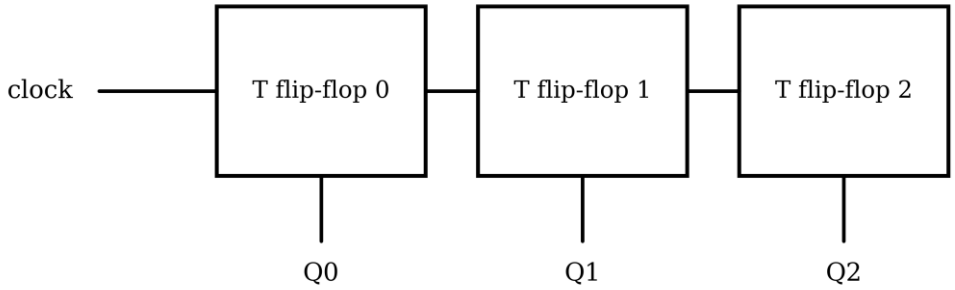


Figure 1.26: *A three-bit ripple counter. Q0 divides the clock by two, Q1 by four and Q2 by eight.*

Count	Q2	Q1	Q0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Table 1.23: *A three-bit counter counting from zero to seven.*

Add one, carry where you must, repeat until the lorry is full. The crates never knew they were doing arithmetic.

The Master Key: *Each stage of a ripple counter toggles on the stage before it, and n flip-flops count 0 to 2 to the power n minus 1, so at least 200 needs 8.*

REAL Question 102

A study group at Manyoni has three sentences on a whiteboard, copied out of three different places and now standing side by side, which is where the trouble started. Each one sounds like something a confident person would say. Each one is wrong.

Identify the error in each of the three claims below and give the correct statement.

1. "A latch and a flip-flop are the same thing."

2. "A four-bit register can store any number up to sixteen."

3. "In an SR latch, S=1 and R=1 sets the output to 1 because set wins."

Model Answer

All three are wrong. A latch is level sensitive and follows its input the whole time its enable is active, while a flip-flop changes only at a clock edge; four bits give sixteen patterns, so the range is 0 to 15 and not up to sixteen; and S = 1 with R = 1 is the forbidden state, which drives both outputs low and leaves the result to whichever input falls first.

Claim one, what is genuinely different: both store one bit, so the confusion is forgivable, but the timing is not the same. A latch is transparent: while its enable is high, the output follows the input continuously. A flip-flop is not: it samples once, at the edge, and is deaf for the rest of the cycle.

Claim one, why the difference decides designs: chain two transparent latches on one enable and a value can race through both inside a single window, arriving a whole stage early. Chain two edge triggered flip-flops and each sees only what the one before it held at the edge, so data advances exactly one stage per clock.

Claim two, count the patterns properly: four bits give 2 to the power 4 patterns, which is sixteen, and the first of them is 0000, standing for zero. Sixteen patterns beginning at zero must end at fifteen, so the register holds 0 to 15. Sixteen itself is 10000 and wants a fifth bit.

Claim two, the wording that avoids the slip: say sixteen values, or say 0 to 15, and never say up to sixteen. The error comes from confusing how many with how high, and it is the same off by one that ends a loop a single iteration too late.

Claim three, what actually happens: S = 1 with R = 1 forces both NOR gates low, so Q and NOT Q are both 0 and the outputs have stopped being complements of one another. Nothing has been set to 1. The pair has simply stopped carrying a meaning.

Claim three, why nothing wins: the harm arrives on release. When both inputs drop back to 0 the gates race to reassert themselves, and the winner is whichever input fell last, decided by track lengths and gate delays rather than by any rule. Neither set nor reset wins, which is why the combination is forbidden.

Three sentences, three quiet slips, and the third is the sort that leaves a technician tapping a board that works on Tuesdays.

The Master Key: *A latch is transparent while enabled while a flip-flop samples at an edge, four bits hold 0 to 15, and S = 1 with R = 1 is forbidden because a race settles the outcome.*

HOT Question 103

A guest lecturer at a secondary school at Tarime finishes his demonstration by holding up a jumper lead and announcing that everything the class has just watched came from bending one wire backwards. Half the room writes it down as a definition.

"Sequential circuits are just combinational circuits with a wire bent backwards." To what extent do you agree?

Model Answer

Agree about the construction and refuse the word just: the physical difference really is one feedback path, and every sequential circuit has combinational logic at its heart, but that single wire changes what may be asked of the circuit and how it must be analysed.

For the claim, the parts list does not change: open a D flip-flop and you find NAND gates. No new element, no new physics, nothing taken from a different shelf. Whoever already owns gates owns everything a sequential circuit is made of, which is why this section introduced no new components.

For the claim, feedback really is the difference: an SR latch is two NOR gates, and on their own two NOR gates are combinational. Cross the outputs into the inputs and memory appears out of nowhere. Cut the two feedback wires and it disappears on the spot, which is a fair test of where it lived.

For the claim, combinational logic still does the thinking: in any state machine the next state and the outputs are worked out by a block of perfectly ordinary gates. The flip-flops only hold the answer until the next edge. Take them out and what is left is a truth table like any other.

Against the claim, the behaviour changes kind: a combinational circuit is a function, so the same inputs always give the same output. A sequential circuit is not a function of its inputs at all, since identical inputs may give different outputs on different occasions. A truth table can no longer describe it and a state table must.

Against the claim, time joins the design: the moment feedback exists the circuit can race against itself, and questions appear that combinational logic never raises. How long must the input be steady before the edge, how long after, how fast may the clock run. None of these mean anything to a bare adder.

Against the claim, the wire is a boundary: one bent wire is also the whole difference between a road and a roundabout. Calling it just a wire measures the change by the price of the copper rather than by what the copper makes possible, which is every register, every counter and every stored program.

The judgement: the sentence is accurate about how sequential circuits are built and misleading about what they are, so accept the construction and reject the word just. One wire separates a machine that answers from a machine that remembers, and remembering is the whole of computing.

The lecturer held up the right piece of wire. He undersold it.

The Master Key: *The construction really is combinational logic plus feedback, but that feedback replaces the truth table with a state table and brings time into the design, so it is a change of kind.*

HOT Question 104

The junction at Biharamulo, where the trunk road meets the road into town, is governed by a police officer in the morning and by courage in the afternoon. The council costed a signal and shelved the file. Nobody knew how small the logic is.

Assess what sequential logic a traffic-light controller for this junction actually needs, covering the states, the state transition table, the flip-flops required to encode those states and the clock, and reach a judgement about the smallest sensible design.

Model Answer

Four states are enough, cycling in a fixed order; four states need two flip-flops, because two bits give four patterns; the next state is always the present plus one, so the transition logic is a two-bit counter; and the clock must be slow, because states last seconds while gates work in nanoseconds.

Listing the states: a state is a complete description of what the junction is doing. Trunk green with town red is one, trunk amber is the second, town green with trunk red is the third and town amber the fourth. Name the state and every lamp is determined.

The transition table is almost empty: read Table 1.24 and the next state column is simply the row below, wrapping from S3 back to S0. No input decides anything, so this is a free running machine and the only thing it answers to is the clock.

How many flip-flops, and why: two. Two flip-flops give 2 to the power 2 patterns, which is four, exactly the number of states. One would give two states and could not tell amber from green. The rule is the smallest n with 2 to the power n at least the number of states.

For the minimum, the lamp logic is built: Q1 and Q0 feed a small block that drives the lamps. Trunk green is NOT Q1 . NOT Q0, trunk amber is NOT Q1 . Q0, town green is Q1 . NOT Q0 and town amber is Q1 . Q0. That is a two to four decoder, already built.

Against the minimum, safety is elsewhere: the encoding uses every pattern, so a single stuck flip-flop lands the junction in a state that looks legal and may show green to both roads. Spend a flip-flop per state instead, one high at a time, and any pattern with two ones is visibly illegal.

The clock is the real work: the states are not equal in length. Green may hold for forty seconds and amber for three, so one steady clock advancing every state equally gives a three second green or a forty second amber. The usual answer is a fast clock, a counter and a comparator.

The judgement: two flip-flops, a two to four decoder and a timed clock is the smallest arrangement that is honestly a traffic light, and here it is the right one, because the sequence never varies. Spend what is saved on the timer and on an interlock that cannot permit two greens.

Present state	Q1	Q0	Next state	Trunk road	Town road
S0	0	0	S1	green	red
S1	0	1	S2	amber	red
S2	1	0	S3	red	green
S3	1	1	S0	red	amber

Table 1.24: *The state transition table for a four-phase controller at the junction.*

The file the council put away contains two flip-flops, a decoder and a clock. The pole was the expensive part.

The Master Key: *Four states, next state equals present plus one, two flip-flops because 2 to the power 2 is 4, and a decoder on Q1 and Q0 lights the lamps, so the difficulty is timing.*

HOT Question 105

A cashew warehouse at Nachingwea runs two counting machines on one conveyor, and one of them, whenever the belt is pushed hard, briefly shows a number that cannot be right and then corrects itself. The machine is not faulty. It is a ripple counter.

Evaluate the trade-off between a ripple counter and a synchronous counter.

Model Answer

A ripple counter is simpler and needs fewer connections, since each stage is clocked by the stage before it, but the change crawls along the chain, so for a short interval the output shows a count the counter never held; a synchronous counter clocks every stage from one line and pays in extra logic.

For ripple, the wiring is minimal: one clock connection reaches the first stage and no others are needed, because every later stage takes its clock from its neighbour's output. An eight-bit ripple counter needs no clock distribution at all, a real saving in track and in soldering.

For ripple, no extra logic is required: each stage is a plain toggle. Nothing has to decide whether a stage should flip, because its neighbour has already decided by changing. Fewer parts, less power drawn and less to get wrong at the bench.

Against ripple, the carry takes time: count from 0111 to 1000 and the first stage falls, which makes the second fall, which makes the third fall, which finally sets the fourth. Each step costs one propagation delay, so the output passes through 0110, 0100 and 0000 before it settles.

Against ripple, the delay grows with width: the worst case is the full width multiplied by the delay of one stage, so a sixteen-bit ripple counter needs sixteen delays to settle after a single pulse. Widen the counter and it gets slower, which is the opposite of what widening ought to cost.

For synchronous, everything moves at once: tie every clock input to the same line and add gates that work out, before the edge arrives, which stages ought to toggle: a stage flips when all stages below it are 1. Every bit then changes within one gate delay of every other, and settling time no longer grows with width.

Against synchronous, you pay in gates and discipline: each stage now needs an AND across the outputs beneath it, and the clock must reach every flip-flop with the same timing, which is a craft of its own. More parts, more power and a clock line routed with care.

The judgement: decide by asking whether anything reads the counter between edges. If the total is sampled long afterwards, as when a person reads a display, ripple is the honest choice. If the outputs feed other logic or run near the speed limit of the parts, the false counts become real errors and synchronous is the only defensible design.

The supervisor's machine was telling the truth about itself. It simply told it faster than the eye can catch.

The Master Key: *Ripple counters are simpler but settle stage by stage and show false intermediate counts, while synchronous counters clock every stage together at the cost of extra logic, so the choice turns on whether anything reads the outputs before they settle.*

The memory hierarchy, locality, cache mechanics and average access time

You have met registers, main memory and the buses between them, and so far every access has been treated as though it cost the same. It does not. A register answers inside the cycle that asks; a disk keeps the processor waiting for millions of them. This section explains how designers stack memories from fastest to largest, why the whole arrangement rests on two habits of ordinary programs, and how to put a single defensible number on the result.

BINDER Question 106

In the hospital records room at Bukoba a file lives in one of four places: the desk, the tray at the clerk's elbow, the shelf behind her chair, the store down the corridor. Nobody planned that. It grew because space near the desk is scarce.

Describe the levels of the memory hierarchy and explain why a computer is built with several kinds of memory rather than one.

Model Answer

A computer keeps data at several levels: registers, cache, main memory, solid state and disk, and finally backup and archive. Fast memory is dear per byte and cheap memory is slow, and no technology is both large and quick, so stacking them gives a store as large as the bottom and nearly as fast as the top.

Registers and cache at the top: answer fastest and cost most. A few dozen registers inside the processor answer within the cycle that asks; cache sits beside the core, kilobytes to a few megabytes, answering in a handful of cycles. Both are dear per byte, so there is little of either.

Main memory in the middle: holds everything currently running. Gigabytes of dynamic RAM answer in tens of nanoseconds, roughly a hundred times slower than a register and about a hundred times cheaper per byte. This is the level a program believes it is using.

Solid state and disk below: trade speed for size and permanence. Flash answers in microseconds and a spinning disk in milliseconds, yet both hold hundreds of gigabytes cheaply and keep their contents when the power goes.

Backup and archive at the base: are priced by what it costs to keep a copy for years, not to read one. Tape, external drives and remote copies are the cheapest storage per byte and the slowest to reach, sometimes needing a person and a journey.

Why stacking works: each level holds the part of the one below that is wanted now. Because a program touches little of its data at a time, most requests are met near the top, so the average cost drifts upwards while the capacity stays that of the base.

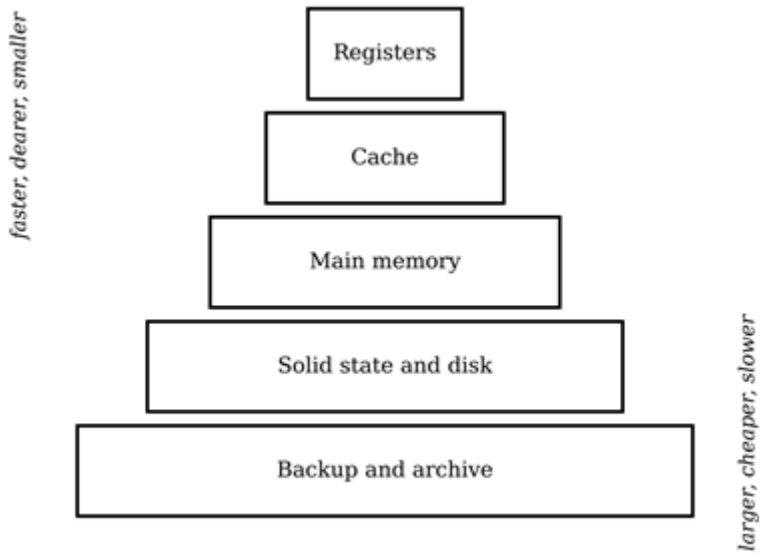


Figure 1.27: *The memory hierarchy. Speed and cost rise towards the top; capacity rises towards the bottom.*

No technology is cheap, fast and vast at once, so the designer refuses to choose and stacks them all.

The Master Key: *Speed and cost per byte rise together, so the hierarchy buys the size of the base at nearly the speed of the top.*

BINDER Question 107

The tailor at Tarime market goes to the same reel of blue thread two hundred times in an afternoon. When it finishes he does not walk to the shop; he reaches into the tin beside the machine. A running program has both habits.

Explain temporal locality and spatial locality, give a program example of each, and explain why the memory hierarchy depends on them entirely.

Model Answer

Temporal locality is the tendency of a location used once to be used again soon; spatial locality is the tendency of locations near one just used to be wanted next. The first is why a copy is worth keeping near the processor, the second why a whole block is fetched rather than a single word.

Temporal locality, and what it buys: shows itself in any loop. A counter is read and written on every one of ten thousand passes, and the loop body is fetched just as often, so a copy held near the processor is reused thousands of times. Hence a cache keeps what was used recently.

Spatial locality, and what it buys: shows itself in any array. Stepping through marks 0, marks 1, marks 2 touches consecutive addresses, and instructions run in sequence too. So memory hands over a whole block at once, because the rest of it is probably the next few requests.

One program showing both: a loop that sums a thousand marks. The counter, the total and the loop instructions come round again and again, which is temporal. The marks are walked in address order, which is spatial.

Why the hierarchy depends on them: the arrangement is a bet that a small copy will answer most requests, and the hit rate is the odds on it. A program addressing memory at random would hit almost never, and would pay the hit time and then the penalty on nearly every access.

Where locality fails: chasing pointers around a scattered list, probing a hash table at random, or striding so far that every step lands in a new block. The hardware still fetches whole blocks, so most of what it carries is wasted.

The hierarchy does not make memory faster. It bets that programs repeat themselves and stay near home, and collects when they do.

The Master Key: *Temporal locality justifies keeping a copy and spatial locality justifies fetching a block; without both, the hierarchy loses on every access.*

BINDER Question 108

The dispensary at Muheza keeps the twenty medicines it issues all day on a tray at the counter, and the rest behind the wall. Nine patients in ten are served without anyone leaving the counter. The tenth waits while somebody unlocks the store.

Explain how a cache works, and define block, cache line, hit, miss, hit rate, miss rate and miss penalty.

Model Answer

A cache is a small fast copy of part of main memory, sitting between the processor and memory. Every access goes to the cache first: a hit is answered in the hit time, while a miss must first bring the whole block containing that address into a cache line, and that extra time is the miss penalty.

Block and cache line: are not the same thing. A block is the unit of transfer, a fixed run of consecutive bytes, commonly thirty two or sixty four, moved as one item. A cache line is the slot holding one block, with a tag naming it and a valid bit.

Hit and miss: a hit is an access the cache can answer, because a valid line carries a tag matching the address. A miss is one it cannot. On a miss the block is fetched into a line, displacing whatever was there, and only then is the word handed over.

Hit rate and miss rate: are the fractions of accesses ending each way, the only two outcomes, so hit rate plus miss rate is 1: a hit rate of 0.95 is a miss rate of 0.05. Both belong to a program running on a cache, never to the cache alone.

Miss penalty: is the extra time a miss costs beyond the hit time, not the whole time it takes. The cache is searched first and fails before memory is troubled, so a miss costs the hit time and then the penalty.

What the program never sees: none of this changes an instruction. The same addresses go out and the same results come back; only the timing differs. Hardware manages the cache alone, so a programmer improves matters only indirectly, by laying data out for locality.

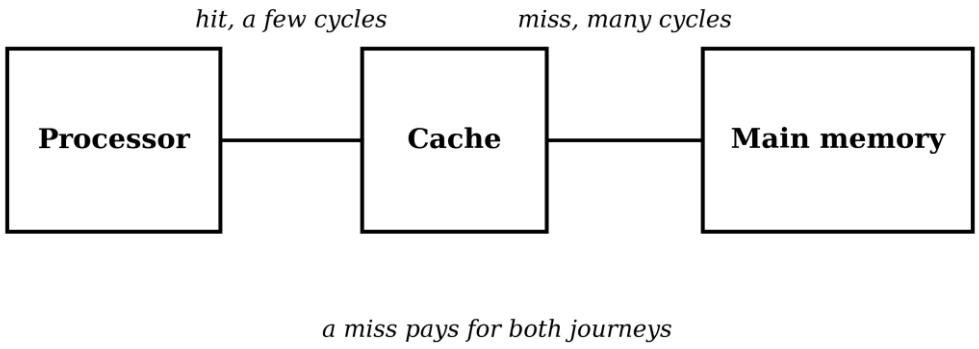


Figure 1.28: *A hit is served by the cache. A miss pays for both journeys.*

The tray, the store, and the walk between them: hit, miss and penalty are those three things given numbers.

The Master Key: *A hit is served by the cache in the hit time; a miss costs that hit time and then the penalty on top.*

BINDER Question 109

At the filling station on the Manyoni road most drivers pay at the pump in half a minute, but one in twenty walks to the office and loses four minutes. The manager wants the average wait. It is not the average of the two.

Explain what average memory access time measures, state the formula, and work through an example.

Model Answer

Average memory access time is the mean time an access takes once hits and misses are weighted by how often each happens, and it is found from AMAT = hit time + (miss rate x miss penalty). With a hit time of 4 nanoseconds, a miss rate of 5 per cent and a penalty of 80 nanoseconds, that is 4 + (0.05 x 80), which is 8 nanoseconds.

Why the formula has that shape: the hit time is paid on every access, hit or miss, because the cache is always searched first. The penalty is paid only by the accesses that miss, so it enters multiplied by the miss rate, which is its share of the traffic.

The worked example: hit time 4 nanoseconds, miss rate 0.05, penalty 80 nanoseconds. The penalty term is 0.05 times 80, which is 4. Add the hit time and the average access takes 8 nanoseconds.

What the answer means: no single access takes 8 nanoseconds. A hit takes 4 and a miss takes 84. Eight is the mean over a great many accesses, so a program making a million of them waits about 8 milliseconds on memory in total.

Where the leverage sits: half of the 8 nanoseconds comes from one access in twenty, because the penalty is twenty times the hit time. Halving the hit time to 2 saves 2 nanoseconds, and halving the miss rate to 0.025 saves the same 2. Small miss rates still matter when penalties are large.

Three ways it goes wrong: enter the miss rate as 5 instead of 0.05 and the answer is a hundredfold nonsense; forget that miss rate is 1 minus hit rate and you solve the wrong case; treat the penalty as the whole cost of a miss and you count the hit time twice.

One formula, two terms, and the habit of asking which term is doing the damage before spending anything on either.

The Master Key: *AMAT = hit time + (miss rate x miss penalty): the hit time is paid always, the penalty only by the fraction that misses.*

REAL Question 110

The tea factory at Rungwe logs every basket over the weighbridge on one control computer, and the engineer has been quoted a price for a cache upgrade that he thinks is high. Nobody in the room can say what the present cache is worth.

Calculate the miss rate and the average memory access time as the machine stands, then calculate what the upgrade would save, both per access and over one drying cycle.

The control computer has one cache between the processor and main memory.

A cache hit is served in 2 nanoseconds.

On the factory's logging program the cache achieves a hit rate of 90 per cent.

A miss costs a further 60 nanoseconds beyond the hit time.

The proposed upgrade would raise the hit rate to 96 per cent, leaving the hit time unchanged.

The program makes 500 million memory accesses during one drying cycle.

Model Answer

The miss rate is 10 per cent and the average memory access time is 8 nanoseconds. Raising the hit rate to 96 per cent takes it to 4.4 nanoseconds, an improvement of 3.6 nanoseconds per access, or 45 per cent, which over the 500 million accesses of a drying cycle saves 1.8 seconds of waiting.

The miss rate: hit rate and miss rate are the two outcomes of one event and sum to 1, so the miss rate is 1 minus 0.90, which is 0.10. One access in ten leaves the cache and goes out to main memory.

The average access time as it stands: AMAT = hit time + (miss rate x miss penalty) = 2 + (0.10 x 60) = 2 + 6 = 8 nanoseconds. Notice that 6 of those 8 nanoseconds are contributed by the single access in ten that misses.

The average access time after the upgrade: the miss rate falls to 1 minus 0.96, which is 0.04, so the penalty term becomes 0.04 x 60 = 2.4 and the average becomes 2 + 2.4 = 4.4 nanoseconds.

The improvement, stated honestly: 8 minus 4.4 is 3.6 nanoseconds saved on every access, which is 45 per cent of the old figure and a speed-up of 1.82 times on memory access. The hit rate rose by six points while the average fell by nearly half, so the two numbers are not the same kind of thing.

What it is worth on the shift: 500 million accesses at 8 nanoseconds is 4.0 seconds of waiting in every drying cycle, and at 4.4 nanoseconds it is 2.2 seconds. The upgrade buys back 1.8 seconds a cycle, and whether that beats the price is now a question with a number in it.

Quantity	As it stands	After the upgrade
Hit rate	90 per cent	96 per cent
Miss rate	0.10	0.04
Hit time (nanoseconds)	2	2
Miss penalty (nanoseconds)	60	60
Penalty term, miss rate x penalty (nanoseconds)	6.0	2.4
Average memory access time (nanoseconds)	8.0	4.4
Waiting per drying cycle (seconds)	4.0	2.2

Table 1.25: *The same machine at two hit rates, with hit time and penalty unchanged.*

The figure the quotation does not show: a larger cache takes longer to search. If the upgrade lifted the hit time from 2 to 3 nanoseconds, the new average would be 3 + 2.4 = 5.4 nanoseconds, still better than 8 but with two thirds of the gain gone. Ask for the hit time before believing the hit rate.

The engineer now holds two numbers instead of two opinions, and the only hard question left is whether the price is worth 1.8 seconds a cycle.

The Master Key: *Turn a hit rate into an average access time before valuing a cache, because percentage points of hit rate are not percentages of speed.*

REAL Question 111

Three sentences are on the board of the computer laboratory at Biharamulo, written in good faith by a student who has understood most of the topic. Each one sounds right when it is said quickly. Each one, in a purchasing decision, costs money.

Identify the error in each of the three claims and correct it.

Claim 1: "A cache with a 95 per cent hit rate makes the machine 95 per cent faster."

Claim 2: “Increasing the cache size always reduces the average access time.”

Claim 3: “The miss penalty is the total time taken by a miss.”

Model Answer

All three are wrong. A hit rate is a probability and not a speed-up, and any speed-up must be computed from access times, which is what AMAT exists for. A bigger cache is slower to search and useless without locality, so always is false. And the miss penalty is the extra time beyond the hit time, so a miss costs both together.

The first claim, why it cannot stand: hit rate is the fraction of accesses the cache answers, a probability between 0 and 1. A speed-up is a ratio of two times. Nothing converts one into the other without the hit time and the penalty, and the claim mentions neither.

The first claim, corrected with numbers: take a hit time of 2 nanoseconds and a penalty of 100. At a 95 per cent hit rate the average access is 2 + (0.05 x 100) = 7 nanoseconds, against roughly 102 with no cache at all. Memory access became about 14.6 times faster, not 95 per cent faster, and the whole machine gains less again.

The second claim, size costs hit time: more lines mean more tags to compare and longer wires to drive, so the hit time rises. The hit time is paid on every access and the penalty only on misses, so past some size the rising hit time swallows the falling miss rate and the average worsens.

The second claim, size cannot create locality: once the cache holds the working set, more lines catch nothing, and a program addressing memory at random gains nothing at any size. Compulsory misses, the first touch of a block, survive every enlargement. Replace always with often, and only up to a point.

The third claim, the penalty is the extra: the cache is searched first and fails before memory is asked, and that time is not refunded. A miss here costs 2 + 100 = 102 nanoseconds in total. The penalty is the 100, not the 102.

Why that definition decides the formula: AMAT = hit time + (miss rate x miss penalty) is right while the penalty is the extra time. The other correct form, (hit rate x hit time) + (miss rate x total miss time), gives (0.95 x 2) + (0.05 x 102) = 7 nanoseconds, the same answer. Mix them into 2 + (0.05 x 102) and you get 7.1, the hit time counted twice on every miss.

Every error here has one shape: a number lifted out of the definition that gave it meaning and used somewhere it does not fit.

The Master Key: *Hit rate is not speed-up, bigger is not always faster, and the penalty is extra, so a miss always costs the hit time as well.*

HOT Question 112

A lecturer at Mvomero opens the lesson with that sentence and half the room copies it down as a fact. It is a good sentence in the way that good sentences often are: it fits the evidence, then keeps walking past where the evidence stops.

“Cache is a confession that main memory is too slow.” To what extent do you agree?

Model Answer

Agree about the history and reject the diagnosis. Cache exists because processor speed pulled away from memory latency over three decades, so the gap it hides is real. But too slow implies an avoidable fault, when main memory is as quick as a store of that size and price can be, and cache is not an apology but the standard answer to a cost curve nobody can repeal.

For the claim, the gap is measurable: processor clock rates climbed far faster than dynamic RAM latency fell, so an access that once cost a few cycles now costs hundreds. Without a cache the processor would spend most of its cycles waiting on a memory that was, by its standards, standing still.

For the claim, the cache does no useful work: tags, comparators, valid bits and replacement logic take area and burn power without adding one instruction to the machine. If memory answered in one cycle, all of it would be waste, and hardware existing only to hide a delay is close to a confession.

For the claim, the gap shows through: on a program that walks memory at random, performance collapses to near raw memory speed. The cache conceals the problem for well behaved programs and cures it for nobody.

Against the claim, the slowness is bought: dynamic RAM uses one transistor and one capacitor a bit, which is why gigabytes are affordable, and must be refreshed, which is why it is slow. Static RAM is quick but needs six transistors a bit, so gigabytes of it could be neither paid for nor powered.

Against the claim, distance is not a fault: a store off the chip is centimetres away with capacitance and drivers between, and a signal takes time to cross. Even perfect technology would answer more slowly from across the board than a cache answers beside the pipeline.

Against the claim, caching is a principle: a small fast store fronting a large slow one appears in file buffers, browser caches, a spares shelf and a dispensary counter. If cache is a confession then so is every one of those, which empties the word of meaning.

The judgement: the sentence is right about the gap and wrong about the blame. Rewrite it as the admission that no one technology is fast, large and cheap at once, and it survives every objection. It also predicts well: memory twice as quick would still be worth caching once processors move again.

Confession is the wrong word because nothing went wrong. The hierarchy is not what designers build when memory disappoints; it is what they would build anyway.

The Master Key: *Cache answers a cost curve, not a mistake: memory is slow because it is cheap and large, and no technology yet is all three.*

HOT Question 113

The manager of a survey office at Pangani hands the engineer a target and no budget: the machine must wait half as long for memory. Three of the four ways cost money, two fight each other, and the cheapest is not inside the machine.

A designer is told to halve the average memory access time of a machine. Evaluate the options available and reach a judgement about which lever is worth pulling first.

Model Answer

AMAT has two terms, the hit time and the miss rate times the penalty, so every lever pulls on one of them: a faster hit time, a lower miss rate from a larger or smarter cache, a smaller effective penalty from a second level of cache, or a program rewritten for locality. Measure which term dominates first, because two of these levers fight each other.

Measure before choosing: with a hit time of 4 nanoseconds, a 5 per cent miss rate and a penalty of 80, both terms are 4 nanoseconds, so halving either alone cannot halve the total. At a 1 per cent miss rate only the hit time is worth attacking.

A faster hit time: attacks the term paid on every access, which makes it attractive. It is bought with a smaller, simpler cache closer to the pipeline, which is exactly the cache least able to hold a working set, and it is the hardest lever to move.

A lower miss rate: comes from more lines, higher associativity, better replacement or prefetching, and pays well while the working set is still spilling. Then it stops, because compulsory misses survive any size and the extra tags lengthen the hit time you were protecting.

A lower penalty from a second level: is usually the best value, because it sits off the critical path and may be large and slowish without touching the hit time. If a second level answers in 12 nanoseconds and a fifth of first level misses reach memory, the effective penalty is 12 + (0.20 x 80) = 28 rather than 80, and the average falls from 8 nanoseconds to 5.4.

Rewriting the program: costs no silicon. The hardware bets on locality and the program decides whether the bet pays: walk a matrix along rows rather than columns, block a loop so the working set fits, pack a structure so a fetched block is all useful. You must own the source.

The levers fight, and the fight matters: hit time and miss rate pull opposite ways, so buying one sells the other. A second level costs area and power but never hit time. A firm running one workload can rewrite it; a company selling chips to strangers cannot.

The judgement: add the second level first, because it attacks the larger term, leaves the smaller alone and is the only lever that does not fight the others. If you own the software too, profile it for locality before signing for hardware, since a badly laid out structure wastes a cache of any price. Keep a faster hit time for last.

Halving is rarely one lever's work. The engineer who measures first spends least, and the one who buys silicon before profiling usually buys the wrong silicon.

The Master Key: *Attack whichever term dominates: a second level cuts the penalty without raising the hit time, and better locality costs nothing but thought.*

Cache mapping, address decomposition, write policy and replacement

Knowing that a cache helps is not the same as knowing where a block lands, how the hardware finds it again, what happens when a write arrives, and which resident is thrown out to make room. This section answers those four questions in order. Two of the answers are commonly written down wrongly, and both are corrected here in full: the line is fixed by the block number and not by the address, and the second replacement policy is FIFO and not a term borrowed from scheduling.

BINDER Question 114

The tobacco floor at Urambo takes bales by the lorry load and the storeman has four bays. His rule never changes: bale number divided by four, and the remainder is the bay. Bale 4 goes where bale 0 went. One of them waits outside.

Explain direct mapping in a cache and state exactly how the line that a block occupies is determined, working through an example.

Model Answer

In direct mapping every block has exactly one line it may occupy, and that line is the block number modulo the number of lines. Write the rule with the block number, never the raw address: block 29 in an eight-line cache goes to line 5, since 29 modulo 8 leaves 5.

The rule, with its operand named: line equals block number modulo number of lines. The block number is the address divided by the block size, remainder discarded. Two steps: address to block, block to line.

Why it must be the block number, never the address: a block is fetched and held as one unit, so every byte of it sits in the same line. Use the address instead and consecutive bytes of one block go to different lines, which no cache can do. The two rules agree only for one-word blocks, which is why the error survives.

Block 29 in an eight-line cache: modulo 8 leaves 5, so that block occupies line 5 and no other. Blocks 5, 13, 21, 29 and 37 all queue for that one line, differing by multiples of 8, and only one can be resident at a time.

The tag is what tells them apart: because many blocks share a line, each line stores a tag beside the data and a valid bit saying whether anything is there. A hit needs the index to reach the line and the tag to match; either failing is a miss.

What the rule buys and what it costs: one index and one comparison, so this is the cheapest and fastest arrangement to build, and no replacement policy is needed since there is only one candidate. It pays in conflict misses: two busy blocks sharing a line evict each other endlessly while the cache stands mostly empty.

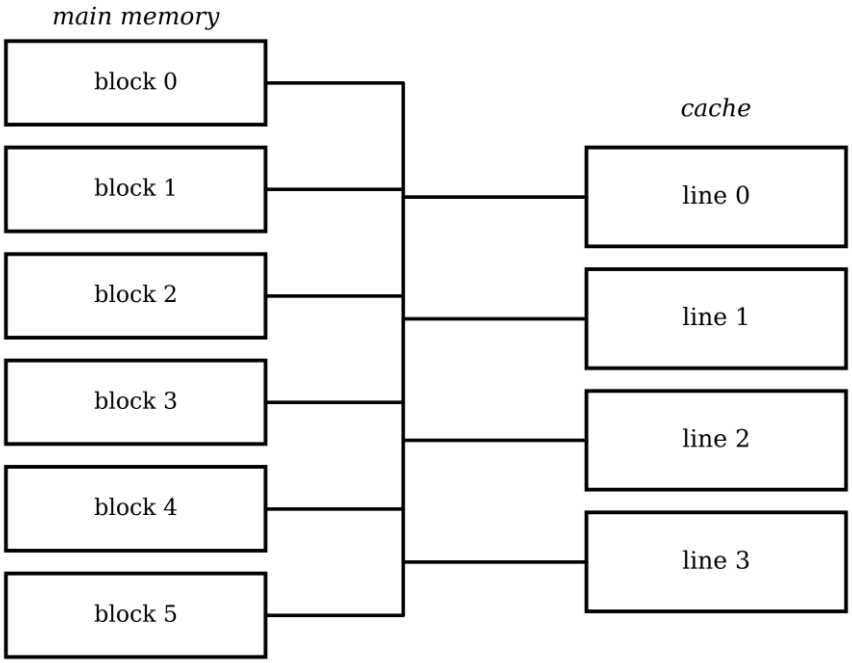


Figure 1.29: *Direct mapping. The line is the block number modulo the number of lines, so blocks 0 and 4 compete for the same line.*

One line per block, decided by arithmetic on the block number, is the whole of direct mapping.

The Master Key: *The line is the block number modulo the number of lines; write the address there and the rule breaks the moment a block holds more than one word.*

BINDER Question 115

The bus stand at Kondoa keeps two rules. At the main gate a coach may park in any free space, so the attendant walks the row to find one. In the goods yard each company has two bays. Every cache stands between those gates.

Explain fully associative mapping and set associative mapping, and explain how each of them relates to direct mapping.

Model Answer

A fully associative cache lets a block occupy any line; a set associative cache lets it occupy any line within one set fixed by the block number. These are not three schemes but one: one-way set associative is direct mapping, and a single set holding every line is fully associative, so associativity is a dial between the ends.

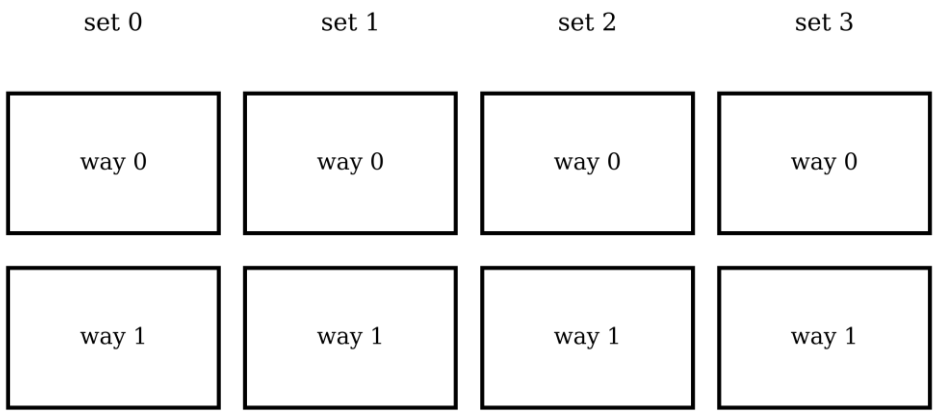
Fully associative mapping: a block may go in any line whatever, so the block number restricts nothing. Finding it again means comparing the wanted tag against every tag at once, a comparator on every line. Conflict misses vanish, leaving compulsory and capacity misses only.

Set associative mapping: the lines are grouped into equal sets, and the block number modulo the number of sets fixes the set. Inside it the block may take any way, so a four-way cache offers four homes not one, and a lookup compares four tags.

One way is direct mapping: put one line in each set and the sets equal the lines. The set index becomes the line index, the only way is the only choice, and the rule collapses into block number modulo number of lines.

One set is fully associative: go the other way and pour every line into one set. The index then selects nothing, there being only one set, and the whole cache must be searched. The two familiar schemes are ends of one scale, not rivals on it.

The trade-off along the dial: more ways means fewer conflict misses, since competing blocks can be resident together. It also means more comparators in parallel, more power on every access, a wider selection of the winning way, and a replacement policy to run. Hit time rises to buy the miss rate down.



a two-way set associative cache with four sets

Figure 1.30: *A two-way set associative cache with four sets. Direct mapping and fully associative are the two ends of this idea.*

Associativity is a setting rather than a third scheme, and the designer only chooses where along it to sit.

The Master Key: *One way is direct mapping, one set is fully associative, and between them ways are bought, conflict misses sold and hit time paid.*

BINDER Question 116

The postal clerk at Makete reads an address from the back. The last part tells her which pigeonhole, the middle part which rack, and the rest is the name she checks on arrival. Remove one of the three and the letter cannot be delivered.

Explain how a memory address is decomposed for a set associative cache, and state the method for finding the width of each field.

Model Answer

An address is cut into three fields: tag, set index and block offset. The offset width is the logarithm to base two of the block size in bytes, the set width that of the number of sets, and the tag whatever the address width has left. The three must sum to the address width, and that sum is the check.

What each field is asked to do: the offset picks the byte inside the block, the set index picks which set to look in, and the tag says which of the many blocks mapping there is the one resident. Three questions, three fields.

The offset comes from the block size alone: a block of 64 bytes needs 6 bits to name a byte within it, because 2 raised to the power 6 is 64. Neither cache size nor associativity enters here; a 32-byte block would need 5 bits.

The set width comes from the number of sets: divide the cache size by the block size for the lines, then the lines by the associativity for the sets. The width is the logarithm to base two of that count. Keep the order: dividing cache size by ways first yields bytes, not sets.

The tag is the remainder and nothing else: tag width equals address width minus set width minus offset width. It is never worked out on its own and never guessed. Whatever the address has left once the other two have taken their share is the tag.

The sum is the check that catches everything: offset plus set plus tag must equal the address width exactly. If they do not sum, one is wrong, and this addition catches a mis-divided cache size, a forgotten associativity and a slipped power of two.

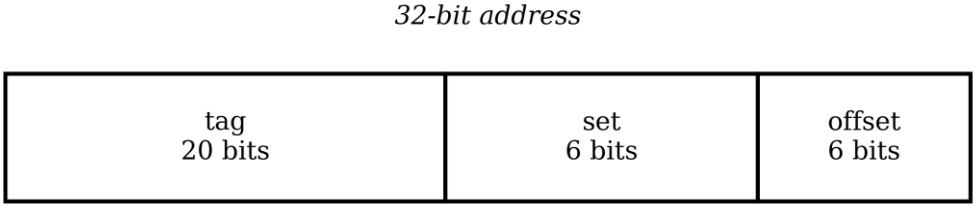


Figure 1.31: *An address cut into tag, set and offset. The three widths must add up to the address width.*

Three fields, two logarithms and a subtraction, then one addition that says whether to trust them.

The Master Key: *Offset from the block size, set from the number of sets, tag from what remains, and the three widths must sum to the address width.*

BINDER Question 117

The fish market clerk at Sengerema writes every sale twice, in the notebook at her hip and in the ledger under the counter. Her neighbour uses only the notebook and copies over when a page fills. One is never behind. The other is faster.

Explain write-through and write-back, and explain write-allocate and no-write-allocate on a write miss.

Model Answer

Write-through sends every write to the cache and to memory together, so memory is never behind, at the price of a memory write every time. Write-back writes only to the cache and marks the line dirty, sending it out only at eviction, which is faster but leaves memory temporarily stale.

Write-through, and what it charges: cache and memory are updated in one operation, so the two never disagree and nothing sits waiting to be saved. Every write crosses the bus, though, including all thousand updates to one counter, so traffic is heavy. A write buffer hides some waiting, none of the traffic.

Write-back, and what it charges: the write lands in the cache alone and a dirty bit is set on the line. Memory is corrected only at eviction, so a line written a thousand times costs one memory write, saving 999. Against

that, memory is stale while the line is resident, eviction costs a write before the fill, and a power cut takes every dirty line.

Write-allocate on a write miss: the block is fetched into the cache first and the write applied to the copy. That is a read before a write, and it pays whenever the program returns to the same block, which most do. It is the natural partner of write-back.

No-write-allocate on a write miss: the write goes straight to memory and the block is never brought in, so nothing is fetched and nothing resident displaced. This suits data written once and not read again, such as an appended log, and it partners write-through.

Choosing between them: the honest question is how soon a written block will be wanted again. Repeated writing to a few lines argues for write-back with write-allocate; scattered writes never read back argue for write-through with no-write-allocate.

Aspect	Write-through	Write-back
When main memory is updated	On every write	Only when the line is evicted
Agreement with the cache	Always in step	Stale while the line is dirty
Memory write traffic	One write per write	One write per dirty eviction
Extra state kept per line	None	A dirty bit
Cost of an eviction	Discard the line	Write the line out, then fill
Effect of a sudden power loss	At most the write in flight	Every dirty line is lost
Usual partner on a write miss	No-write-allocate	Write-allocate

Table 1.26: *Write-through and write-back compared.*

Two policies, two ways of meeting a write miss, one question underneath all four.

The Master Key: *Write-through keeps memory correct and pays on every write; write-back pays once at eviction and leaves memory stale until then.*

REAL Question 118

The cotton buying post at Ushetu has taken delivery of a second-hand control machine, and the fitter left a note saying only that the cache is 32 kilobytes and eight-way. The technician who must service it needs to know how it reads an address.

Calculate the number of lines and the number of sets, then find the widths of the offset, set and tag fields and verify that the three widths sum to the address width.

The processor issues 32-bit byte addresses.

The cache holds 32 kilobytes of data, not counting tags.

The cache is 8-way set associative.

A block, and therefore a cache line, holds 64 bytes.

One kilobyte means 1024 bytes throughout.

Model Answer

The cache holds 512 lines arranged in 64 sets. The offset is 6 bits, the set index is 6 bits and the tag is 20 bits, and 6 plus 6 plus 20 is 32, which is the address width, so the decomposition is sound.

The number of lines: cache size divided by block size. 32 times 1024 is 32768 bytes, and 32768 divided by 64 gives 512 lines. Note that the stated capacity counts data only, since the tags and the valid bits are extra silicon that nobody advertises.

The number of sets: lines divided by ways. 512 divided by 8 gives 64 sets. Take the two divisions in that order, because dividing the cache size by the associativity first produces a number of bytes rather than a number of sets, and that is the commonest slip in this whole calculation.

The offset width: a block of 64 bytes, and 2 raised to the power 6 is 64, so 6 bits name a byte inside the block. The associativity plays no part whatever in this field.

The set width: 64 sets, and 2 raised to the power 6 is 64 again, so 6 bits name the set. The two sixes are a coincidence of these particular numbers and not a rule. Make the cache four-way and the sets become 128, the set field becomes 7 bits and the tag falls to 19.

The tag width, then the check: 32 minus 6 minus 6 leaves 20 bits of tag. Now add the three back together: 6 plus 6 plus 20 is 32, the full address width. Nothing has been dropped and nothing counted twice, which is all the verification this calculation ever needs.

A second check that costs nothing: the whole address space holds 2 raised to the power 32 divided by 64, which is 67108864 blocks, and they share 64 sets, so 1048576 blocks compete for each set. That figure is 2 raised to the power 20, exactly the number of distinct tags, which is precisely what the tag field must be able to tell apart.

What the tags cost her: 512 lines carrying 20 bits each is 10240 bits, or 1280 bytes, sitting on top of the 32768 bytes of data. That is about 3.9 per cent of overhead before valid and dirty bits are counted, and it is why no designer makes a tag one bit longer than the arithmetic demands.

The technician now knows how her machine cuts an address, and she proved it with an addition that took two seconds.

The Master Key: *Lines from cache size over block size, sets from lines over ways, offset and set from powers of two, tag from what is left, and the three must sum.*

REAL Question 119

The clinic at Chunya keeps three files on the desk and the rest in the store. A fourth means somebody walks. The morning nurse returns whichever has lain untouched longest, the afternoon nurse whichever arrived first. By evening the two desks hold different files.

Trace the reference string through the cache under LRU, showing the contents after every reference and marking hits and misses, then count the hits and state where FIFO would have behaved differently.

The cache is fully associative and holds three lines.

The cache is empty at the start.

The replacement policy is least recently used, written LRU.

The blocks are referenced in this order: 2, 5, 2, 8, 5, 3, 2, 5.

A block is placed in a free line while one is free, and only then does replacement begin.

Model Answer

LRU gives three hits and five misses on this string, a hit rate of 37.5 per cent, and the cache ends holding blocks 3, 5 and 2. FIFO would give only two hits, because it evicts block 5 at the seventh reference for no reason but age and then misses on it at the eighth, where LRU still holds it.

How LRU decides: every hit and every fill moves that block to the most recently used end of an order kept across the lines. When room must be made, the block at the far end goes. The policy is a bet on temporal locality, that whatever has been touched lately will be touched again.

The first four references: block 2 misses into a free line, block 5 misses into the second, block 2 hits and becomes the most recent, block 8 misses into the third. Nothing has been evicted yet, because a free line is always taken before a resident block is disturbed.

The two evictions: at reference 6 the cache is full and block 3 arrives. Block 2 was last touched at reference 3 and is the least recently used, so block 2 goes. At reference 7 block 2 returns, and block 8, untouched since reference 4, is now the oldest by use, so block 8 goes.

The count: hits fall at references 3, 5 and 8, so three hits and five misses out of eight references, a hit rate of 37.5 per cent. Four of the five misses are compulsory, being the first touch of each distinct block, and the fifth is block 2 coming back at reference 7 after being evicted.

Where FIFO differs, and why: FIFO evicts by arrival and never by use, so a hit gives a block no protection at all. At reference 7 the oldest arrival is block 5, so FIFO throws it out although it was used at reference 5, and at reference 8, when block 5 is wanted again, FIFO misses where LRU hits. Two hits against three.

Name the policies correctly: the two policies here are LRU and FIFO. First Come First Served is scheduling vocabulary, describing the order in which a processor takes jobs from a queue, and it is not a cache replacement policy. Write FIFO when you mean the cache, and random and least frequently used are the other names worth knowing.

Reference	Hit or miss	Line 0	Line 1	Line 2	Least recently used now
2	Miss	2	empty	empty	2
5	Miss	2	5	empty	2
2	Hit	2	5	empty	5
8	Miss	2	5	8	5
5	Hit	2	5	8	2
3	Miss	3	5	8	8
2	Miss	3	5	2	5
5	Hit	3	5	2	3

Table 1.27: *The Chunya string traced under LRU. Contents are shown after the reference has been served.*

Three hits out of three lines, and the entire difference between the two policies turns on whether being used counts for anything.

The Master Key: *LRU evicts by last use and FIFO by arrival, so a hit protects a block under LRU and does nothing at all for it under FIFO.*

REAL Question 120

A sheet is going round a school in Ludewa, photocopied until its edges are grey. Three of its sentences about cache are now in a hundred notebooks. All three are wrong, and one shows itself only when a block holds more than one word.

Identify the error in each of the three claims and give the correct statement, with a worked example where one is needed.

Claim one: in direct mapping the line is found by taking the address modulo the number of lines.

Claim two: the replacement policies used in caches are LRU and First Come First Served.

Claim three: a fully associative cache needs no tag, because a block can go anywhere.

Where a worked example is needed, assume 32-bit addresses, 64-byte blocks and an eight-line direct-mapped cache.

Model Answer

All three are wrong. The modulo is taken on the block number and not on the address; the second policy is FIFO, while First Come First Served belongs to scheduling; and a fully associative cache needs the longest tag of any arrangement rather than none at all.

Claim one, and the example that kills it: the correct rule is line equals block number modulo number of lines. Address 128 has block number 128 divided by 64, which is 2, and 2 modulo 8 gives line 2. Address 129 lies one byte on, inside the same block, so its block number is still 2 and its line still 2. The wrong rule gives 0 for 128 and 1 for 129, sending two bytes of one block to different lines.

Why the wrong rule ever looked right: with a one-word block the block number and the address are the same number, so the two rules agree and nobody notices. Real caches use blocks of many bytes, and the agreement ends at once. Strip the offset bits away and the set field stands: the block number modulo the number of sets, wired rather than divided.

Claim two, a name taken from the wrong topic: the correct pair is LRU and FIFO, with random and least frequently used also in real use. First Come First Served is scheduling vocabulary for how a processor takes jobs from a queue, and it has been carried across here by mistake. The word belonging to cache replacement is FIFO.

Claim three, which is exactly backwards: a fully associative cache carries the longest tag of all. With no set field, nothing in the address says where the block sits, so every bit above the offset must be stored: 32 minus 6 is 26 bits, against 20 for the same cache cut into 64 sets. Six extra bits per line, and the tag grew because the index went.

And every one of those tags must be compared: since the block may be anywhere, a lookup compares the wanted tag against all resident tags at once, needing a comparator on every line. That is the price of the freedom the claim admired, and the reason fully associative caches stay small.

The habit behind all three corrections: say which quantity an operation acts on, take each name from the topic it belongs to, and remember the tag is only what the address has left, so removing a field lengthens it.

The sheet will keep circulating, but a candidate who can say why address modulo lines fails on a 64-byte block will not be copying from it.

The Master Key: *Modulo the block number, name the policy FIFO, and the tag grows as the index shrinks, which makes fully associative the longest tag of all.*

HOT Question 121

A supplier in Mbulu is selling a machine on one line of its specification sheet: sixteen-way associative, where the machine standing beside it is four-way. The buyer has no other numbers and the difference in price is real. The sheet prints none of what matters.

"Increasing associativity is always worth it." To what extent do you agree?

Model Answer

Only in part. Raising associativity does remove conflict misses, and the first doubling usually pays handsomely, but the gains fall away quickly while the costs do not, and the extra ways are paid for in hit time, which every access bears. The test is average access time, not miss rate, and by it more ways can leave a machine slower.

The case for, in one sentence: a direct-mapped cache can thrash on two busy blocks while most of it stands empty, and associativity is the only structural cure. Two ways lets the pair live together and a class of misses disappears.

The case for, with the usual rule of thumb: a two-way cache of a given size tends to miss about as often as a direct-mapped cache of twice the size, so buying a way is often cheaper than buying capacity. It also makes timing predictable, which matters wherever a deadline must be met.

The case against, the curve flattens fast: most of the benefit lies in the step from one way to two, and much of the rest in two to four. Beyond eight the surviving misses are largely compulsory and capacity misses, which no associativity touches, so the money buys almost nothing.

The case against, hit time is paid on everything: each way adds a tag comparator in parallel and widens the selection of the winning way, and no data is handed over until the winner is known. That lengthens every access, hit or miss, while the better miss rate helps only the few per cent that miss.

The arithmetic that settles it: with a 60 nanosecond penalty throughout: direct-mapped, hit time 2 and miss rate 6 per cent, gives 5.6 nanoseconds; two-way, hit time still 2 and miss rate 4 per cent, gives 4.4; eight-way, miss rate 3.5 per cent but hit time now 3, gives 5.1. The first doubling wins 1.2 nanoseconds and widening beyond it hands 0.7 back.

The case against, everything else it costs: all ways are energised on every access, so power climbs with the ways and area with them, and a policy must now choose a victim, true LRU across sixteen ways needing bookkeeping updated on every hit. None of it appears on the sheet.

The judgement: agree for the first step, disagree for the last. Two-way and four-way are nearly always worth having, most of all nearest the processor where conflicts do the greatest damage. Sixteen-way earns its place only off the critical path, as in a large last-level cache whose hit time can be spent freely. Ask for the hit time, then compute.

Associativity is a purchase, not a virtue, judged by what the whole machine does afterwards.

The Master Key: *Associativity buys conflict misses down and hit time up, so it pays early, flattens fast, and must be judged on average access time.*

HOT Question 122

The grain buying centre at Kilindi weighs and pays for maize on one machine. The line from the grid drops two or three times a week, sometimes for a second and sometimes until evening. Nobody has asked what happens to the last few sales.

Assess write-through against write-back for this machine. Weigh consistency, performance and the risk of losing dirty lines, say what a battery-backed buffer would change, and reach a judgement.

Model Answer

For a machine whose power fails without warning, write-through is the safer default and write-back the faster, and a battery bridges them. Write-back can hold tens of kilobytes of writes existing only inside the cache; write-through holds one. Behind both sits a harder truth: main memory is volatile too, so no write policy makes anything durable.

What a dirty line actually is: a write the program believes has happened and memory has never seen. Nothing marks it as at risk, no error is raised, and if the machine stops it is gone. The clerk watched the sale recorded.

The case for write-through here: memory never falls behind the cache, so the exposure at any instant is one write in flight rather than a cache full of them. Recovery is simpler with no dirty state to reason about, evictions are free since a clean line is merely discarded, and the loss is easy to account for.

What write-through costs on this machine: every write crosses the bus. If a fifth of accesses are writes, that is a hundred million memory writes in five hundred million accesses. Where one running total is updated two thousand times, write-through pays two thousand times and write-back once.

The case for write-back here: writes to a tally, a day total and a set of counters land on the same few lines again and again, exactly the pattern write-back was designed for. It gathers neighbouring writes so a whole line moves in one transfer, and leaves the bus free for reads.

Sizing the risk instead of fearing it: a 32 kilobyte cache of 64-byte lines holds 512 lines, so at worst 512 dirty lines, 32768 bytes, exist nowhere else. Far fewer are dirty at any moment, but for a machine that can stop mid-transaction the worst case is the planning figure.

What a battery-backed buffer changes: a battery or supercapacitor that keeps the machine alive long enough to flush every dirty line turns an unbounded worry into a bounded engineering task, and write-back then comes nearly free. The condition is that it be tested, since an untested battery is a belief, not a control.

The judgement: take write-back with a tested battery-backed flush and a supply large enough to reach the generator. Without a battery, take write-through and accept the traffic as the price of a machine that stops without warning. Either way the program must force each completed sale to non-volatile storage, since memory in step with the cache is still volatile.

The policy decides how much is lost when the lights go. Only the disk decides whether anything survives.

The Master Key: *Write-through bounds the loss to one write, write-back bounds it to the dirty lines, and only a flush to non-volatile storage bounds it to nothing.*

HOT Question 123

A student at a training centre in Ruangwa writes a loop that adds two arrays and reasons from her instruction count that the machine should finish in a blink. It takes seven times as long. The trouble is where the two arrays begin.

Evaluate why a program can run far slower than its instruction count suggests, in terms of conflict misses, and what a programmer can do about it without changing the hardware.

Model Answer

Instruction count sets a floor, not a time. Time is instructions times cycles per instruction times cycle time, and cycles per instruction is inflated by stalls waiting on memory. Conflict misses are the cruellest, since they can drive a nearly empty cache to a miss on every access, the classic case being two arrays whose addresses differ by exactly the cache size.

Where the time actually goes: an instruction that hits retires in a cycle or two; one that misses waits perhaps a hundred. A single miss per iteration outweighs the whole loop body, so counting instructions measures work, never waiting.

What a conflict miss is: a miss suffered because two blocks compete for one set, not because the cache is too small and not because the block was new. It is the only miss that can strike a cache standing almost empty.

The two-array case, worked: a 32 kilobyte direct-mapped cache of 64-byte lines has 512 lines, and an address indexes by its block number modulo 512. Put one array at 0 and the other at 32768, one cache size on: 32768 divided by 64 is 512, and 512 modulo 512 is 0, so element i of each lands on one line. Each read evicts what the other wants, and every access misses.

The cost in her own numbers: take 8-byte elements, eight to a line, and a hundred cycle penalty. Without the conflict each array misses once in eight iterations, near 25 cycles of waiting and 4 of work, about 29 an iteration. With it both miss every time: 204. Same instructions, seven times the time.

What the programmer can do without buying anything: pad. One extra block in the first array's declared size makes the second begin 64 bytes later, so the indexes differ by one. Or interleave the two into one array of pairs, so paired elements share a line. Or block the loop to fit the working set.

How you find it: the signature is a running time that changes by a large factor when an array size changes by a little, since a power-of-two size is what aligns the streams. Counters name the miss type, but the size experiment costs nothing.

The judgement, and whose responsibility this is: hardware cannot fix it in general: associativity helps, yet a two-way cache still fails where a loop touches three conflicting streams. Compilers and allocators can pad, and good ones do, but neither knows the loop. The programmer knows how many streams it touches and owns the layout, so the responsibility is practically hers, and the hardware owes only a fast common case.

The instruction count was right the whole time. It never claimed to be a time.

The Master Key: *Conflict misses can empty a cache of its usefulness while it stands nearly empty, and padding or interleaving fixes what no purchase can.*

Memory capacity arithmetic, chip organisation and virtual memory

Every claim in this section is a number you can check. The width of the address bus fixes how much memory a processor can reach, chips are combined by arithmetic and never by guesswork, and virtual memory is the arrangement that lets a program ignore both. Work each calculation on paper as you read it, because the marks here go to whoever can show the steps, not to whoever recognises the words.

BINDER Question 124

The market at Itigi has three-digit plot numbers stencilled on every gate, and the clerk who chose three digits chose the size of the market without meaning to. Build a thousandth stall and it has no address. An address bus does the same, silently.

Explain how the width of the address bus fixes the amount of memory a processor can address, and show how that capacity is calculated.

Model Answer

Each address line carries one bit, so n lines make two to the power n patterns and each pattern names one location. Multiply that count by the size of what is stored there and you have the capacity in bytes. Twenty lines over byte-sized locations reach 1 048 576 bytes, exactly 1 MB.

The rule, and why it is a power of two: each line is held low or high, so n lines produce two to the power n addresses and no others. Add a line and the count doubles; remove one and it halves. Sixteen lines reach 65 536 locations, twenty reach 1 048 576, twenty-four reach 16 777 216.

$$\text{capacity} = 2^{n} \text{ x word size}$$

Locations are counted, bytes are calculated: the address bus fixes how many locations exist; the width of a location decides how big each is. On a byte-addressable machine the two figures agree, which is why the second step gets forgotten. Where a location holds a sixteen-bit word, the same lines reach the same locations but twice the bytes.

One clean case, worked through: take twenty-four address lines over sixteen-bit words. 2 to the power 24 is 16 777 216 locations, each holding 2 bytes, giving 33 554 432 bytes. Divide by 1024 twice and that is 32 MB, not the 16 MB a hurried reading hands in.

$$\text{capacity} = 2^{24} \text{ x } 2 \text{ bytes} = 33\,554\,432 \text{ bytes} = 32 \text{ MB}$$

Convert last, and convert by 1024: one kilobyte is 2 to the power 10 bytes, one megabyte 2 to the power 20, one gigabyte 2 to the power 30. Stay in bytes until the arithmetic ends, then convert once. Dividing by 1000 in the middle is the commonest way to lose this question.

Reading the rule backwards: given a required capacity, divide by the size of a location for the number of locations, then ask what power of two that is. A store of 1 MB in bytes needs 20 lines, because 2 to the power 20 is 1 048 576.

Count the lines, raise two to that power, multiply by what sits at each address, then convert.

The Master Key: *n address lines reach two to the power n locations, and it is the data width, never the address bus, that decides how many bytes each holds.*

BINDER Question 125

The storeman at Chemba builds his grain store from sacks of one size, the only sack the mill sells. To widen a row he stands two side by side; to hold more he lays row on row. Memory chips arrive in one size too.

Explain how a memory chip is organised, and how several chips are combined to build a store of a required word size and capacity.

Model Answer

A chip labelled 1K x 8 holds 1024 locations of 8 bits each, which is 8192 bits, or 1 KB. To widen the word, place chips side by side and give them all the same address lines. To deepen the store, stack rows of chips and decode the high address bits so that only one row is enabled at a time.

Reading the label: the figure before the cross is the number of locations, the figure after it the width of one location in bits. 1K x 8 means 1024 locations of 8 bits, or 8192 bits in all. 1K x 4 holds the same locations but hands over only 4 bits at a time.

Chips across, for the width: every chip in a row receives the same address lines and the same enable, and each supplies its own slice of the word. Chips across equals word width divided by chip width, so two 1K x 4 chips make a 1K x 8 store.

Rows, for the depth: rows equal the locations required divided by the locations on one chip, so four rows of 1K chips reach 4096 locations. Every row shares the low address lines, and the row number decides which of them is listening.

The decoder chooses the row: the high address bits feed a decoder whose outputs drive the chip select pins, one output to a row. Choosing one of four rows takes 2 lines and one of eight takes 3. Lines to the chips plus lines to the decoder must equal the full address width.

The total, and the check: total chips equals chips across multiplied by rows. A 4K x 16 store built from 1K x 8 chips needs 2 across and 4 rows, so 8 chips. Multiply one chip's capacity by the count and compare against the target before ordering.

What you need	How the chips are placed	How many chips	What the address lines do
A wider word	Side by side, all enabled together	Word width divided by chip width	Every line reaches every chip
More locations	Stacked in rows, one row enabled	Locations needed divided by locations per chip	Low lines shared, high lines decoded
Both at once	A rectangle of rows and columns	Chips across multiplied by rows	Low lines shared, decoder drives chip select

Table 1.28: *Building memory from chips.*

Two moves only, side by side for width and stacked for depth, with one division behind each of them.

The Master Key: *Chips across for the width, rows for the depth, their product for the total, and the high address bits decoded to say which row answers.*

BINDER Question 126

A herdsman at Longido keeps more cattle than his boma will hold, so most are out at the grazing and only today's few stand inside. Anyone who counts the boma has counted the herd wrongly. A machine with too little memory does exactly this.

Explain what virtual memory is, how a virtual address becomes a physical one, and state the benefits the arrangement brings.

Model Answer

Virtual memory lets every program be written as though it owned one large continuous memory of its own. The addresses it issues are virtual; the memory management unit translates each into a physical address before any store is touched, and whatever is not resident is fetched from disk on demand.

What the program is allowed to believe: a program is compiled against one flat range of addresses beginning at zero, whatever else the machine is running and however little memory is fitted. It never learns where it truly sits, and nothing in it needs to.

The translation, done in hardware: the memory management unit stands between processor and memory and converts every virtual address into a physical one as the access is made. Software cannot do this work, because a translation is needed on every reference and any slower method would swallow the machine.

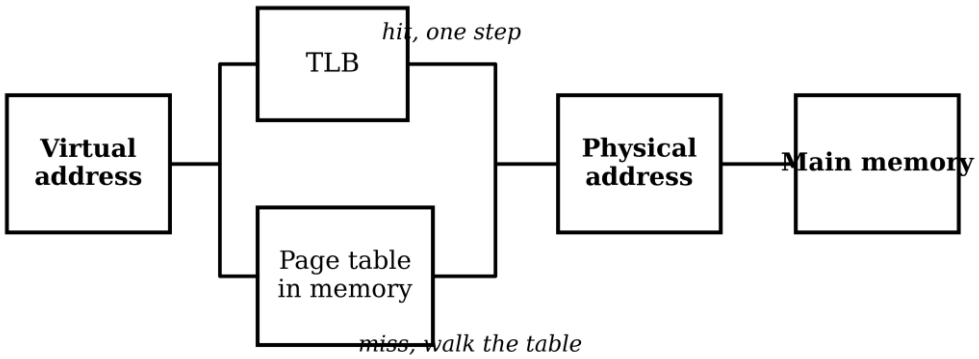


Figure 1.32: *Virtual to physical translation. The TLB is a cache of recent translations, and a miss means walking the page table.*

Not resident means on disk: the parts not currently wanted are held in a reserved area of the disk, the swap space. Touch one and the hardware raises a fault, the operating system brings the part in, and the instruction restarts as though nothing happened.

Benefit one, and benefit two: a program may be larger than the physical memory, since only its active parts need be resident, and a machine with 512 MB fitted can still offer a 32-bit program the whole 4 GB space. One program also cannot reach another's memory, because every translation goes through that program's own table.

Benefit three, relocation: a program can be loaded into any free frames, scattered anywhere in the store, and still see one continuous space. Without it, every program would have to be adjusted to wherever it landed, and free memory would have to be one unbroken block.

Virtual memory is a promise kept in hardware: address the machine as though it were large and let translation deal with the truth.

The Master Key: *Programs address a space, not a store; the unit translates, the disk holds the remainder, and size, protection and relocation all follow.*

BINDER Question 127

The land office at Kibondo keeps one heavy register of every plot in the district and a thin notebook on the counter holding the last few looked up. Most callers ask about a plot somebody asked about this morning, so the register stays shut.

Explain pages, page tables, page faults and the translation lookaside buffer, and show how a virtual address is split for translation.

Model Answer

A page is a fixed-size block of virtual address space and a frame is the same size block of physical memory. The page table maps page numbers onto frame numbers, a page fault is the exception raised when a page is not resident, and the TLB caches recent translations so the ordinary case costs one step.

Pages and frames: both are of one fixed size, commonly 4 KB, and a page is exactly the size of a frame. Fixing the size abolishes external fragmentation: any page fits any free frame, so the system never hunts for a hole of the right shape.

How the address is split: the low bits are the offset within the page and pass through untranslated, since page and frame have identical layout. Only the page number is looked up. With 4 KB pages the offset is 12 bits, because 2 to the power 12 is 4096, and a 32-bit address leaves 20 bits of page number.

What the page table holds: one entry per page, giving the frame number if the page is resident, plus present, protection and dirty bits. Those 20 bits of page number mean 1 048 576 entries for one program, which is why real tables are built in levels.

The page fault: the present bit is clear, so the hardware raises an exception. The operating system finds or frees a frame, reads the page from disk, writes the frame number into the entry and restarts the instruction. It is not an error and the program is never told.

The TLB, and why so little works: a small fully associative cache of recent page-to-frame mappings, perhaps 64 entries, inside the memory management unit. A hit yields the frame in one step; a miss means walking the table in memory, costing at least one further access. Locality saves it: a 4 KB page is touched thousands of times.

Offset straight through, page number translated, notebook first and register second.

The Master Key: *Only the page number is translated; the offset passes through, the table maps it, the fault fetches it, and the TLB makes it one step.*

REAL Question 128

The cold store at the Bunda fish landing runs on a small controller bought second hand, and the dealer selling an upgrade promises it will take 64 MB. The manager has the old board on the table and can count the address lines himself.

Calculate the number of addressable locations and the total capacity of the board as it stands, then determine how many further address lines the advertised upgrade would demand.

The controller has 22 address lines.

Each addressable location holds one 16-bit word, that is 2 bytes.

The upgrade is advertised as 64 MB of the same 16-bit memory.

One kilobyte means 1024 bytes and one megabyte means 1024 kilobytes throughout.

Model Answer

The board reaches 4 194 304 locations, which is 8 388 608 bytes, or 8 MB. The advertised 64 MB is eight times that, and 8 is 2 to the power 3, so the upgrade needs 3 further address lines, 25 in all. This board cannot be made to hold it, and the dealer should be told so.

The number of locations: 2 to the power 22, which is 4 194 304. Count address lines only. The data lines and the control lines take no part whatever in this figure, and a board with more pins is not thereby a board with more memory.

The capacity in bytes: multiply the locations by the size of one location. 4 194 304 times 2 gives 8 388 608 bytes. This is the step that gets skipped, and skipping it reports half the true capacity while looking entirely convincing.

The capacity in units anyone will read: divide by 1024 to get 8192 KB, and by 1024 again to get 8 MB. Convert at the end and never in the middle, and never by 1000, because the two conventions differ by enough to change an answer.

What the advertised figure demands: 64 times 1024 times 1024 is 67 108 864 bytes, and at 2 bytes to a location that is 33 554 432 locations. That figure is 2 to the power 25, so the processor would need 25 address lines to name them all.

The extra lines, and the check: 25 minus 22 leaves 3. Verify it from the other direction: each line doubles the store, three doublings multiply it by 8, and 8 MB times 8 is 64 MB. The two routes agree, so the answer is safe.

The advice to the manager: fitting larger chips will achieve nothing, because the processor has no way to name locations beyond its 22 lines and the surplus cannot be reached. Either the dealer supplies a board whose bus is 25 lines wide, or the upgrade is a fiction with a price on it.

One variation worth rehearsing: had those same 22 lines addressed single bytes, the capacity would be 4 194 304 bytes, that is 4 MB. Same bus, half the store, and nothing changed but the size of a location. It is the second step alone that separates the pair.

Twenty-two lines, two bytes at each, eight megabytes in total, and a dealer three address lines short of his promise.

The Master Key: *Locations come from the address lines, bytes from the size of a location, and extra lines from counting how many doublings the target lies away.*

REAL Question 129

The sawmill office at Mafinga wants a controller carrying 8 KB of memory, and the only chips on the dealer's shelf are 1K x 4. The technician neither argues nor guesses. He works out how many go across, how many rows, and orders once.

Calculate how many chips are needed across for the word width, how many rows are needed for the depth, the total chip count, and how many address lines must be decoded to select a row.

The store required is 8 KB, eight bits wide at every location.

The only chips available are 1K x 4.

1K means 1024 locations, and one kilobyte means 1024 bytes.

A decoder of any required size can be fitted.

Model Answer

Two chips go across, because 8 bits divided by 4 bits is 2. Eight rows are needed, because 8192 locations divided by 1024 is 8. That is 16 chips altogether. Thirteen address lines reach the board: the low 10 go to every chip on it, and the high 3 are decoded into eight chip selects.

Chips across, for the width: word width divided by chip width, so 8 divided by 4 is 2. One chip supplies the top 4 bits of every location and its partner the bottom 4. The pair share every address line and are enabled together, so they behave as one 1K x 8 chip.

Rows, for the depth: locations required divided by locations on one chip. 8 KB of eight-bit locations is 8192 locations, and 8192 divided by 1024 gives 8 rows. The depth calculation ignores the width entirely; the two divisions never touch.

The total, checked by capacity: 2 across times 8 rows is 16 chips. Check it: 16 chips of 1024 locations at 4 bits each hold 65 536 bits, and 65 536 divided by 8 is 8192 bytes, the 8 KB that was asked for. A chip count failing this check is wrong however tidy it looks.

How many address lines in all: 8192 locations must each have a name, and 2 to the power 13 is 8192, so the board takes 13 address lines. This figure comes from the finished store, never from the chip.

How those lines divide: one chip holds 1024 locations and so needs 10 lines of its own, and those same 10 are wired to every chip on the board. The remaining 3 are the row number, since 13 minus 10 is 3. Shared lines plus decoded lines equal the full width, which is the check worth doing every time.

The decoder: those 3 lines feed a 3-to-8 decoder giving 8 outputs, one per row, and each output drives the chip select of both chips in its row. Exactly one row is enabled for any address, so the shared data lines are never driven by two chips at once.

Reading the map: row 0 covers locations 0 to 1023 and row 7 covers 7168 to 8191, and the high three bits are the row number written in binary.

Row	High three lines	Locations covered	Chips in the row
0	000	0 to 1023	Two 1K x 4
1	001	1024 to 2047	Two 1K x 4
2	010	2048 to 3071	Two 1K x 4
3	011	3072 to 4095	Two 1K x 4
4	100	4096 to 5119	Two 1K x 4
5	101	5120 to 6143	Two 1K x 4
6	110	6144 to 7167	Two 1K x 4
7	111	7168 to 8191	Two 1K x 4

Table 1.29: *Sixteen 1K x 4 chips arranged as eight rows of two, giving 8 KB of eight-bit memory.*

Two across, eight down, sixteen chips, ten address lines shared by all of them and three decoded to pick the row.

The Master Key: *Divide the widths for chips across, divide the depths for rows, multiply for the total, and give the leftover address lines to the decoder.*

HOT Question 130

A secondary school at Kasulu bought twenty machines with very little memory, on the advice that virtual memory would make up the difference. In one sense it does. Everything opens, eventually. It also takes four minutes, and the disk light never goes out.

"Virtual memory makes a small machine behave like a large one, so buying more physical memory is a waste." To what extent do you agree?

Model Answer

Hardly at all. Virtual memory enlarges the address space, not the machine. It buys the ability to run at all, and pays for that by trading memory for disk at about a hundred thousand to one, so the moment the working set will not fit, the machine stops computing and starts paging.

The case for, which is real: without virtual memory a program larger than the store simply refuses to start. With it the machine runs everything, and the parts nobody is using cost only disk. For light work the difference is invisible, and those machines do open their files in the end.

The case for, the other two gifts: protection between programs and freedom of relocation come from the same machinery and are worth having at any memory size. Neither can be bought by fitting chips, so virtual memory is no substitute for memory and memory is none for it.

The case against, the ratio: a memory access takes roughly 100 nanoseconds and servicing a page fault roughly 10 milliseconds, which is 10 000 000 nanoseconds. One fault therefore costs what 100 000 memory accesses cost. Nothing else inside the machine has a cliff of that height.

The case against, the fault rate: at one fault per million accesses the effective access time is 110 nanoseconds, a slowdown of one tenth, which nobody notices. At one fault per thousand it is 10 099.9 nanoseconds, about 101 times slower. A rate a thousand times worse makes the machine a hundred times slower.

Thrashing, where the claim dies: when the resident frames are fewer than the working set, each page fetched evicts one about to be wanted. The fault rate climbs, the processor waits, the operating system sees an idle processor and admits more work, and the machine collapses into pure paging. That is the disk light nobody can switch off.

What the money actually buys: fitting more memory raises the number of frames, which lowers the fault rate, the only term in that calculation a school can move. It does not take much: enough frames to hold the working set and the cliff disappears.

The judgement: disagree, with one concession. Virtual memory is indispensable and is not an alternative to memory; it makes a shortage survivable rather than fatal. Judge a machine by whether its working set is resident

and buy chips until it is. Past that point further memory really is idle money, and that is the only sense in which the claim stands.

Virtual memory decides whether a program runs at all; physical memory decides whether it runs in time.

The Master Key: *Virtual memory widens the address space and not the store, the fault rate is the only term that matters, and physical memory alone moves it.*

HOT Question 131

The water board at Simanjiro delivers by tanker. One large tanker means fewer trips, but it reaches the smallest village half full and the surplus is poured away. A fleet of small tankers wastes almost nothing and spends the day on the road.

Assess the choice of page size in a virtual memory system.

Model Answer

Neither end wins outright, because the two costs move in opposite directions. Large pages shrink the page table and use a slow disk well but waste memory inside every page. Small pages waste almost nothing but multiply both the table and the faults. The right size follows the workload, and 4 KB survives as the compromise that offends fewest of them.

For large pages, the table shrinks: a 32-bit space cut into 4 KB pages has 1 048 576 entries, and at 4 bytes an entry that is a 4 MB table for every process. Cut the same space into 64 KB pages and there are 65 536 entries, a table of 256 KB, sixteen times smaller.

For large pages, the TLB reaches further: a 64-entry TLB with 4 KB pages covers 256 KB of memory and no more. Those same 64 entries with 64 KB pages cover 4 MB. A program with a wide footprint misses far less often, and not one extra transistor was bought.

For large pages, the disk prefers them: a disk transfer is dominated by seek and rotation rather than by the bytes moved, so fetching 64 KB costs barely more than fetching 4 KB. Measured per useful byte delivered, the large page is much the cheaper thing to bring in.

Against large pages, internal fragmentation: the last page of any region is on average half empty. At 4 KB that wastes 2048 bytes per region; at 64 KB it wastes 32 768 bytes. Across two hundred regions the loss is 400 KB in the first case and 6400 KB in the second, and none of it can be used.

Against large pages, the fetch is a gamble: a fault brings in the whole page whether the program will touch it or not. Under a scattered access pattern most of a 64 KB page is transferred, occupies a frame and is evicted unread, so bandwidth and memory are spent on nothing.

For small pages, and their bill: small pages give fine control: only what is genuinely wanted stays resident, and a tight loop needs very few frames. The price is a table too large to keep resident, forcing multi-level lookup, and many more faults, each carrying the same fixed disk cost.

The judgement: it turns on the footprint. Small pages suit many modest processes touching memory unpredictably, which describes a general-purpose machine, and 4 KB has been the settled answer for decades. Large pages suit one process walking a huge continuous structure, a database or a scientific array, where TLB reach dominates. Modern machines decline to choose and offer both at once.

There is no correct page size, only one that is correct for the working set in front of you.

The Master Key: *Large pages buy a small table and long TLB reach with wasted memory; small pages buy a tight fit with a huge table and more faults.*

HOT Question 132

The weighbridge on the road at Gairo stops every lorry without exception. It holds no cargo and costs each driver a minute, but nothing moves until it has done its work. A queue at that gate is a different problem from a slow warehouse.

Evaluate the claim that the translation lookaside buffer is the most important cache in the machine.

Model Answer

There is a strong case, and it fails on the word most. Every reference the processor makes is translated first, so the TLB stands on a path nothing avoids, and no data cache can say that. But it holds no data, its misses are usually cheap, and per event a data cache miss costs many times more.

For, it sits on every access: an instruction must be fetched and its operands read, so a simple load needs at least 2 translations before any value moves. The data cache is consulted for data alone; the TLB is consulted for everything, including every access the data cache will go on to hit.

For, a miss is not small: a TLB miss means walking the page table, and that table is itself in memory, so a two-level walk turns one load into 3 memory references. Remove the TLB and every access pays that walk, leaving a machine running at a fraction of its rated speed.

For, so much for so little: a few dozen entries, and locality supplies the rest, since a program touching a 4 KB page touches it thousands of times before moving on. Hit rates above 98 per cent come from a structure smaller than one cache line of data.

Against, it holds no data: a TLB hit produces an address, not a value, and the load still has to happen. The TLB can only save an extra trip; it can never supply the thing being asked for. Calling it the most important cache confuses the map with the goods.

Against, the arithmetic per access: at a 99 per cent hit rate with a 30-cycle miss, translation costs 0.3 cycles on an average access. A data cache missing 3 per cent of the time with a 100-cycle penalty costs 3.0 cycles, ten times as much. On figures that ordinary, the data cache is the larger problem.

Against, and the condition that reverses it: translation only catches the data cache when the TLB miss rate reaches 10 per cent, and that needs a footprint far beyond the reach of the TLB. A database walking gigabytes of index does exactly that; a loop over one array never comes close.

The judgement: the TLB is the most unavoidable cache, which is not the most important. It is indispensable and a machine without one is crippled, but importance is measured in cycles lost, and for most workloads the data cache loses more of them. Award the title to whichever structure the workload is missing in, and measure before awarding it.

Every lorry crosses the weighbridge, but the cargo is still what the journey was for.

The Master Key: *The TLB lies on every access and is unavoidable; the data cache lies off that path yet costs far more per miss, so importance is a measurement.*

Instruction set architecture: formats, encoding, addressing modes and RISC against CISC

A machine is bought and sold on what a programmer can see of it, and that visible surface is the instruction set architecture. This section takes it apart: what belongs to the architecture and what belongs to the organisation, how an instruction is cut into fields and what each field width can name, the addressing modes and the effective address each one computes, and the long argument between RISC and CISC. Two familiar errors are corrected in full along the way.

BINDER Question 133

The central line still stops at Kaliua, and the goods shed there has kept the same counter, the same three windows and the same green forms for forty years. The clerks have changed twice over. What the customer sees was held still on purpose.

Define instruction set architecture, state what belongs to it, distinguish it from computer organisation, and explain why a stable architecture is commercially valuable.

Model Answer

The instruction set architecture is the machine as the programmer can see it: everything a correct program is entitled to depend on. It is the contract between those who write software and those who build hardware, held still on purpose while the hardware underneath is rebuilt.

What belongs to it: the instruction set itself, the registers a program may name, the addressing modes, the data types and their sizes, the word length, and the memory model, meaning how memory is addressed and how bytes are ordered within a word.

What organisation is instead: how that visible machine is actually built: pipeline depth, cache size and mapping, bus widths, the number of execution units, and whether control is hardwired or microprogrammed. Change any of it and every program still runs to the same answer. Only the time moves.

The test that separates them: ask whether a correct program could tell. If a change can alter what the program computes, it is architecture. If it can only alter how long the program takes, it is organisation. Cache size is organisation; the count of visible registers is architecture.

One architecture, many machines: one architecture may be built as a cheap slow part and as an expensive fast one, so a family at several prices runs the same programs without recompilation. The programs were written against the architecture, never against the implementation.

Why stability is worth money: software costs more than hardware and lasts far longer. A buyer who can carry every program forward will buy the next machine; a buyer who must rewrite will shop elsewhere. An architecture that never breaks its promise keeps its customers, which is why awkward old instructions survive for decades instead of being tidied away.

Architecture is the promise about what the machine will do. Organisation is the private business of how it manages.

The Master Key: *If a correct program can tell the difference, it is architecture; if only the clock can tell, it is organisation.*

BINDER Question 134

The timber yard at Mufindi cuts every plank to one length before anything else happens. Then the sawyer takes his pencil and sets it out: so much for the tenon, so much for the shoulder, so much for waste. The plank never grows.

Explain instruction formats and encoding. Name the fields of a register to register format and of an immediate format, and explain what fixes the width of each field.

Model Answer

An instruction is a fixed number of bits cut into fields, and two rules govern the cutting. The field widths must sum to the instruction width, and the width of a field fixes what that field can name: five bits select one of thirty-two registers, and six bits allow sixty-four opcodes.

The register to register format: six bits of opcode, then source 1, source 2 and destination at five bits each, then a five-bit shift amount and a six-bit function code. Add them: 6 plus 5 plus 5 plus 5 plus 5 plus 6 is 32, exactly the instruction width.

The immediate format: the second source register is replaced by a constant carried inside the instruction: six bits of opcode, five for the source, five for the destination and 16 for the immediate, which is 32 again. One operand arrives free, with no register read and no memory access.

What a field width can name: a field of n bits names 2 raised to the power n things and not one more. Five bits reach 32 registers exactly, six bits reach 64 opcodes, and a 16-bit immediate holds 65536 values. Ask for a thirty-third register and the field must widen.

The budget is closed: every bit given to one field is taken from another, because the plank is not getting longer. Widen the opcode to seven bits and the immediate falls from 16 to 15, whose 32768 values are exactly half of what it held. That is why architects argue over single bits.

Why fixed length pays for itself: the decoder knows where every field lies before it knows what the instruction is, so registers can be read while the opcode is still being examined. Variable length buys smaller code and pays for it in decoding.

a 32-bit register-to-register instruction

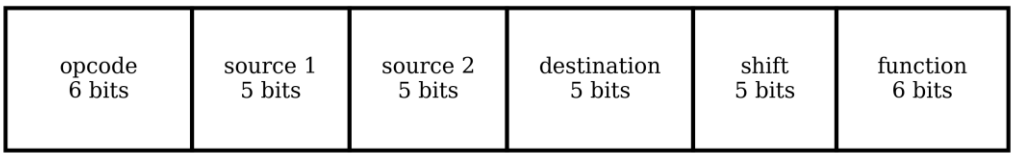


Figure 1.33: *A register to register instruction, cut into fields. The field widths must add up to the instruction width.*

Encoding is the division of a fixed resource, and the craft is deciding what each bit is worth.

The Master Key: *The fields must sum to the instruction width, and a field of n bits can name 2 raised to the power n things, never more.*

BINDER Question 135

A delivery driver in Lushoto is given his work in four ways. Sometimes they put the parcel into his hands. Sometimes they say it is already in the cab. Sometimes they write a house number. Sometimes they say three doors past where you stand.

Describe the addressing modes and give the effective address rule for each of them.

Model Answer

An addressing mode is the rule that turns the fields of an instruction into the operand wanted, and the effective address is what that rule produces. Two modes produce no address at all, because the operand is already in hand.

Immediate and register, which fetch nothing: in immediate mode the operand is a constant inside the instruction, so there is no effective address and no memory access, though the field width limits the value. In register mode the operand sits in a named register, and the field names the register, not an address.

Direct and register indirect: in direct mode the instruction carries the address, so the effective address is the address field. In register indirect mode a register holds it, so the effective address is the contents of that register, which is what a pointer is.

Base plus displacement and indexed: in base plus displacement the effective address is a base register's contents plus a constant from the instruction, which reaches a field of a record. Indexed adds a base to an index register multiplied by a scale, the index counting elements and the scale turning them into bytes.

Program counter relative, how branches work: the effective address is the program counter plus a signed offset, so a branch reaches backwards and forwards, and the code may be loaded anywhere because no absolute address is written down. The counter has already advanced to the next instruction before the offset is added.

What the choice costs: count the accesses: none for immediate and register, one for direct and for register indirect, one plus an addition for the computed modes. No architecture offers every mode here; a load and store machine may offer four and genuinely have no direct or indirect memory addressing at all.

Mode	Where the operand is	Effective address
Immediate	In the instruction itself	None. The operand is the field
Register	In a named register	None. The field names the register
Direct	In memory, at the stated address	The address field itself
Register indirect	In memory, at the address in a register	The contents of the register
Base plus displacement	In memory, near a base	Base register plus displacement
Indexed	In memory, in an array	Base plus index multiplied by scale
Program counter relative	In memory, near the current instruction	Program counter plus offset

Table 1.30: *The addressing modes and their effective addresses.*

Write the rule, substitute, then fetch. Every mode on the list yields to that order of work.

The Master Key: *The effective address is whatever the rule computes, and the modes differ only in the work and the accesses that computation costs.*

BINDER Question 136

Two workshops in Igunga repair the same lorries. One keeps plain spanners and a fitter who knows which three to reach for. The other keeps a wall of single-purpose tools and a boy told which to fetch. Both finish the work.

Explain what RISC and CISC are and what actually distinguishes them.

Model Answer

RISC and CISC are two answers to one question: where should the complexity live? RISC puts it in the compiler and keeps the hardware simple. CISC puts it in the hardware, in microcode, and keeps the program short. Everything else follows from that one choice.

What RISC decides, and what follows: few simple instructions, all of one fixed length, so decoding is trivial and the pipeline is easy to keep full. Many registers, because operands must be there. Load and store as the only instructions touching memory. Hardwired control. The compiler carries the scheduling and the allocation.

What CISC decides, and what follows: many instructions, some doing a great deal, of lengths that vary. Arithmetic may address memory directly, so fewer registers are needed. Control is microprogrammed, because microcode is cheaper than wiring for a wall of complicated instructions. The program comes out short, which mattered when memory was dear.

The load and store discipline: on a RISC machine only load and store reach memory, so adding two memory locations does not exist as an instruction. It must be written as load, load, add, store, four instructions where the CISC machine writes one. That rule alone explains the register count and much of the cycle count.

Why fixed length matters most: the decoder knows the field boundaries before it knows the instruction, so several can be examined at once. A variable-length stream must first discover where one instruction ends before it can begin the next, and that work is stubbornly serial.

What each side actually pays: RISC pays in instruction count and code size. CISC pays in cycles per instruction and in decoding. The only figure that settles anything is total time, which is instruction count multiplied by cycles per instruction multiplied by cycle time.

Criterion	RISC	CISC
Instruction count and complexity	Few, simple, each doing little	Many, some doing a great deal
Instruction length	Fixed, one width throughout	Variable, from one byte upwards
Addressing modes	Few, often three or four	Many, sometimes more than a dozen
Registers	Many, commonly thirty-two or more	Few, commonly eight to sixteen
Memory access	Load and store only	Arithmetic may address memory directly
Control unit	Hardwired	Microprogrammed
Cycles per instruction	Near one, and uniform	Several, and widely varying

Table 1.31: *RISC and CISC compared.*

Learn the one decision and the table writes itself. Learn the table alone and half of it is gone by morning.

The Master Key: *RISC hands the complexity to the compiler and CISC hands it to the microcode, and every other difference follows from that.*

REAL Question 137

The livestock market at Kiteto weighs cattle on a bridge whose controller was designed in a workshop rather than bought. The engineer has settled the instruction width and the register file, and listed the operations. She has not found out whether they fit.

Calculate the width of each field, decide whether the register to register format fits, find how many bits remain for a constant in the immediate format, and state the range that constant can express.

Every instruction is 32 bits wide, and all instructions are the same width.

The register file holds 64 registers, and any register may be named by any field.

The design calls for 90 distinct operations.

The register to register format names three registers: two sources and one destination.

The immediate format names one source register, one destination register and one constant, and the constant is signed.

Model Answer

Each register field needs 6 bits and the opcode needs 7, so the register to register format spends 25 of the 32 bits and fits with 7 to spare. The immediate format spends 19 bits on opcode and registers, leaving 13 bits of constant, which signed runs from minus 4096 to 4095.

The register field: 64 registers, and 2 raised to the power 6 is 64, so 6 bits name any of them exactly, with no pattern wasted and none short. Five bits would reach only 32, which is half the file, so the whole upper half would be unreachable by that field.

The opcode field: 90 operations. Six bits reach 64, and 64 is fewer than 90, so 6 bits will not do. Seven bits reach 128, which is enough. Always take the smallest width whose reach is at least the count; never round to the nearest, and never assume the count is a power of two.

Whether the register to register format fits: 7 bits of opcode and three register fields of 6 bits, which is 18, come to 25 bits of the 32 available. It fits, and 7 bits are left. Those spare bits are not waste: they can carry a shift amount, or a function code extending the opcode, or be reserved untouched for later.

The immediate format: the constant has taken the place of the second source register. The fixed part is 7 bits of opcode, 6 for the source and 6 for the destination, which is 19, and 32 minus 19 leaves 13 bits for the constant. Nothing else may be added without taking those bits back.

The range that constant can express: a field of 13 bits holds 8192 patterns, and two's complement splits them evenly about zero, so a signed field of n bits runs from minus 2 raised to the power n minus 1, up to 2 raised to the power n minus 1, minus 1. For 13 bits that is minus 4096 to 4095. Unsigned it would run from 0 to 8191.

What she must do about larger constants: a value outside that range cannot travel inside one instruction. It has to be assembled by two, the upper bits placed first and the lower bits added, or else loaded from memory. That is the standing charge for fixed-length encoding.

The reserve she is left holding: seven bits give 128 opcode patterns and 90 are spoken for, so 38 remain. That is her entire allowance for every operation anyone will ask for in the life of the design, because once machines are in the field the encoding cannot be widened.

The fields fit, the constant reaches four thousand either way, and she knows the price of every bit before a single wire is drawn.

The Master Key: *Take the smallest width that reaches the count, add the fields, subtract from the instruction width, and read the signed range off the bits that survive.*

REAL Question 138

A stock control machine at a gold buying office in Msalala is being traced instruction by instruction, because the totals it prints at closing are wrong. The engineer has the register contents, a dump of four memory words and the instruction, and nothing else.

For each addressing mode listed, compute the effective address and state the operand fetched, then say where the commonest slip lies.

Register R3 holds 4000. Register R5 holds 2.

The constant written in the instruction is 12, and the scale for one word is 4 bytes.

Memory holds: address 4000 contains 250; address 4004 contains 61; address 4008 contains 900; address 4012 contains 77.

The branch instruction sits at address 5000, instructions are 4 bytes wide, and the branch offset is 36.

The machine advances the program counter before the offset is added.

Model Answer

Immediate takes the operand 12 and computes no address at all. Register indirect gives 4000 and fetches 250. Base plus displacement gives 4012 and fetches 77. Indexed gives 4008 and fetches 900. Program counter relative gives 5040, and that is the one most often got wrong, by measuring from the branch instead of from the instruction after it.

Immediate, where there is no effective address: the operand is the constant 12, lying inside the instruction. No register is read and no memory is touched. Say that plainly rather than inventing an address for it, because an effective address written under this mode shows at once that the mode was not understood.

Register indirect: the effective address is the contents of R3, which is 4000, and the word at 4000 is 250. One register read and one memory access. R3 holds an address and not a value, which is the whole idea of a pointer, and the register itself is unchanged by the fetch.

Base plus displacement: the effective address is 4000 plus 12, which is 4012, and the word there is 77. R3 is the base and 12 the constant written into the instruction. The sum is formed in the address adder and used once; R3 still holds 4000 afterwards, which candidates often assume it does not.

Indexed, where the scale earns its place: the effective address is 4000 plus 2 multiplied by 4, which is 4008, and the word there is 900. R5 counts elements and the scale of 4 turns elements into bytes. Forget the scale and you compute 4002, which is not the address of any word at all, only a byte inside one.

Program counter relative, and the advance: by the time the offset is added the counter has already moved to the next instruction, so it holds 5000 plus 4, which is 5004, and the effective address is 5004 plus 36, which is 5040. Measure from 5000 and you get 5036, wrong by 4 bytes, which is exactly one instruction.

The pattern that covers all five: the effective address is always whatever the rule computes and never the field as written. Note also that the branch performs no data fetch: its effective address becomes the new program counter rather than a place to read from, which is why that row alone has no operand.

Five rules, five substitutions, five answers, and the only difficult one was difficult because of a counter that had already moved.

The Master Key: *Compute the address the rule demands, then fetch, and on a branch remember the counter has already advanced by one instruction before the offset is added.*

REAL Question 139

A sheet is going round a school in Mkalama with three sentences on it about architecture. Each has the shape of a fact and the confidence of one. Each has been copied into hundreds of notebooks. The third wears the appearance of fairness.

Identify the error in each of the three claims, give the correct statement, and supply figures where figures are needed.

Claim one: a processor with only register, immediate, base plus displacement and program counter relative addressing supports direct and indirect addressing too, since every processor does.

Claim two: RISC machines are always faster than CISC machines because their instructions are simpler.

Claim three: comparing a processor from 2005 with one from 2008 is a fair way to decide whether one architecture family is better than the other.

Where figures help, take a memory to memory addition: 7 cycles on the CISC machine at a 2 nanosecond cycle, against four instructions of 1 cycle each on the RISC machine at a 1 nanosecond cycle.

Model Answer

All three are wrong. A machine offers the modes its architect gave it and no others, so the machine described genuinely has no direct or indirect memory addressing at all. A simpler instruction does less work, so simplicity settles nothing. And two parts from 2005 and 2008 tell you about those two parts, not about two families.

Claim one, and why it is not a small slip: nothing obliges an architecture to offer every mode. The machine described reaches memory only through a base register and a displacement, so it has no instruction carrying an address and none following an address held in memory. Those four modes are all it has, and a question asking how it performs direct addressing is asking about a machine that does not exist.

How the effect is obtained without the mode: what direct addressing would have done is done in two steps: build the address into a register, then load through it with a displacement of zero. Register indirect is base plus displacement with the constant set to zero. The effect survives; say so rather than pretend the mode is there.

Claim two, and what simple actually means: a simple instruction does less, so more of them are needed. The RISC machine spends four instructions where the CISC machine spends one. Simplicity buys a shorter cycle

and a cycles-per-instruction figure near one; it is paid for in instruction count. Nothing guarantees which side wins.

The only honest comparison there is: total time is instruction count multiplied by cycles per instruction multiplied by cycle time. Here CISC gives 1 times 7 times 2, which is 14 nanoseconds, against RISC at 4 times 1 times 1, which is 4, so RISC wins easily. Now give the RISC machine a 4 nanosecond cycle: 4 times 1 times 4 is 16, and it loses. The same instruction counts, the opposite verdict.

Claim three, why those dates cannot decide anything: parts from 2005 and 2008 differ in fabrication process, clock speed, cache size, transistor budget and design effort, and not one of those is the architecture family. Whatever the comparison measures is a difference between two products, wrongly credited to two families.

And in any case the families have moved: the part from 2008 is now 18 years old, and the distinction has blurred throughout. CISC machines translate their instructions internally into simple fixed operations of exactly the RISC kind, and RISC machines have grown rich extensions. To compare the families today, compare machines of today.

Three sentences, three assumptions, and none of them survives being stated out loud.

The Master Key: *Not every machine has every mode, simplicity is only one of three factors in total time, and two old parts are a comparison of parts and never of families.*

HOT Question 140

A supplier at Mbinga is selling two machines off the same shelf and calling one of them RISC, as though the word decided which was quicker. Open both and the difference he charges for lives in the part that finds where one instruction stops.

"The distinction between RISC and CISC no longer means anything." To what extent do you agree?

Model Answer

Largely, but not entirely. As a prediction of speed the distinction died years ago, because both families use the same techniques on the same fabrication and a modern CISC machine is a RISC engine wearing a translator. As a description of the visible interface it is alive, since a program still sees fixed or variable length and whether arithmetic may touch memory.

The case for, the two families borrowed from each other: CISC machines break every instruction into simple internal operations and execute those, so the RISC engine won the argument inside the CISC machine. RISC machines meanwhile added extensions for vectors, floating point and encryption until their lists ran into hundreds.

The case for, what decides speed lies elsewhere: pipeline depth, superscalar width, out-of-order execution, branch prediction, the cache hierarchy and the fabrication process. Both families use all of them, and two machines within one family differ by more than the families do. The name predicts nothing a buyer cares about.

The case for, transistors made the decoder cheap: when the argument began, a complicated decoder took a serious fraction of a chip with very few transistors to spare. On a modern budget it is a rounding error in area, so the economy behind the quarrel has evaporated.

The case against, the encoding is still visible and still costs: a fixed-width instruction can be cut into fields before anyone knows what it is, so several can be decoded side by side. A variable-length stream must be scanned to find where each instruction ends, so widening the decoder is far harder and the extra logic burns power on every fetch.

The case against, load and store discipline is a real property: whether arithmetic may address memory changes the compiler, the exception model and how much work a partly executed instruction must abandon. It is written into the architecture and visible to every program, not a slogan on a sales sheet.

The case against, code density still buys something: variable length packs a program into fewer bytes, so less instruction cache is occupied and less bandwidth spent fetching. Where memory is measured in kilobytes, as in a water meter's controller, density decides.

The judgement: agree where the claim is aimed at performance: nobody can say which of two machines is faster from the family name. Disagree where it is aimed at the interface: fixed against variable encoding, and load and store against memory operands, remain real differences with consequences in decode width, power and code size. The distinction has stopped being a verdict and stayed a description.

The words still describe two ways of encoding a machine. They stopped saying which one wins long ago.

The Master Key: *Ask what the family name predicts: no longer the speed, but still the encoding and the memory discipline every program on it must obey.*

HOT Question 141

A design review at an instruments workshop in Iramba has stalled on one line. Somebody has proposed doubling the register file from 32 registers to 64, because the compiler will have more room. Nobody has asked what the extra registers are paid for with.

Assess the claim that a larger register set is always an improvement.

Model Answer

Only up to a point, and the point arrives earlier than the claim expects. More registers do cut memory traffic and ease the compiler's work, but every register must be nameable inside an instruction, saved on every context switch, and built into a structure whose access time grows with its size. The gains flatten once the working set fits; the costs do not.

The case for, fewer journeys to memory: every value kept in a register is a load and a store not performed. When the compiler runs out of registers it spills values to the stack, and each spill is memory traffic and cache pressure that need not exist.

The case for, the compiler can attempt more: with room to spare it keeps loop counters and repeated subexpressions in registers, unrolls loops, passes arguments in registers rather than on the stack, and moves a load ahead of the instruction needing it, covering the latency.

The case against, every register must be named: a register to register instruction names three registers. At 32 registers that is 5 bits each, 15 bits of a 32-bit instruction. At 64 it is 6 each, 18 bits, and the 3 extra come out of the immediate: a 16-bit constant falls to 13 and its range shrinks by a factor of 8.

The case against, state must be saved and restored: on every interrupt and every switch of process the registers are written out and read back. With 64 registers of 8 bytes that is 512 bytes each way, against 128 for a file of 16. On a machine that switches often, the extra registers must earn that time back.

The case against, the register file is a physical object: it is a multi-ported memory read two or three times and written once for every instruction in flight. Doubling it lengthens wires, enlarges decoders, adds area and power, and can push the read into a second cycle, lengthening the pipeline for everything.

The case against, the returns diminish and then stop: ordinary code seldom keeps more than about 16 to 32 values live at once, so beyond that the extra registers sit unused while every program pays their encoding and switching cost. Unrolled numerical loops are the honourable exception.

The judgement: a larger register set is an improvement up to roughly 32 and must be argued for beyond that. Where instructions are long enough to name registers cheaply, where switches are rare and where the code is numerical, more will pay. Where encoding is tight and switching frequent, more registers make the machine worse. And since the count is architecture, it cannot be corrected later.

The question is never whether more would help. It is what the bits and the microseconds are being taken from.

The Master Key: *Registers are bought with encoding space, context switch time and access latency, and the buying stops paying once the working set already fits.*

HOT Question 142

A metering company at Mbarali has been asked to add one instruction to a processor it already sells, for a checksum its meters compute a thousand times a second. The question is not whether the team can, but who maintains it in 2050.

Evaluate what an architect should do when asked to add a new instruction to an existing architecture, and reach a judgement.

Model Answer

Refuse by default and make the case be argued. A new instruction cannot be withdrawn later. It consumes encoding space that can never be recovered, must be implemented and verified in every machine the family ever ships, and is worth nothing unless something will actually emit it.

Encoding space cannot be renewed: the opcode field holds a fixed number of patterns, and every one spent is one a better idea will not have. Seven bits give 128 patterns, and where 90 are used only 38 remain for the design's whole life. Do not spend that reserve on a narrow operation.

Forever is the true cost: once the instruction ships and one customer's program uses it, it must keep working on every future implementation: in hardware, in the compiler, in the debugger, and in the trap handler that emulates it once the logic is dropped. Three weeks of design is decades of obligation.

The compatibility promise is the asset: an architecture is commercially valuable because old programs keep running on new machines, and anything added enters that promise on the day it ships. This is why long-lived architectures carry instructions nobody has needed for thirty years and cannot remove them.

Whether a compiler would ever emit it: an instruction a compiler cannot recognise is reached only by hand-written assembly and one library, and soon by nothing. Ask whether the pattern can be spotted in ordinary source, or whether programmers must be told to call an intrinsic by name. The second is far weaker.

The case for adding, when it is strong: where the pattern is frequent, measured in real programs and expensive done the long way. One instruction replacing twenty saves fetch bandwidth, cache footprint and energy. And some things cannot be expressed without new instructions: atomic operations, cache and ordering control, and access to new hardware.

How to add it if it must be added: prefer a general primitive to a special case, because the general one is still useful when the special case is forgotten. Put it in a block reserved for extensions rather than scarce main encoding space, and make it optional so a program may ask whether it is there.

The judgement: the burden of proof sits with the proposal. Add it only where measurement shows the pattern is hot in real work, a compiler or widely used library will emit it, and the encoding does not spend scarce space. For the checksum here, write the routine in software, measure what it costs, and add silicon only if the numbers survive.

An instruction takes three weeks to add and a generation to be rid of. Only one of those numbers is in the proposal.

The Master Key: *A new instruction spends non-renewable encoding space to buy a permanent obligation, so it must be justified by measured, compiler-reachable use or refused.*

Programmed input and output, interrupts and direct memory access

A processor that fetches every byte itself is a processor doing nothing else. This section puts the three ways a machine moves data between itself and the world side by side: programmed input and output, interrupt driven transfer, and direct memory access. They are usually met one at a time and never compared, so the comparison is made here in full. Memory mapping is then set against port mapping, an argument rarely met at all, and the chapter closes on what all of it was for.

BINDER Question 143

The clerk at the Ikungi weighbridge has one lorry due and no way of knowing when. So he walks to the gate, looks down the road, sees nothing, and walks back to his desk. Two hours of this, and the lorry comes no earlier.

Explain programmed input and output, also called polling, and explain exactly why it wastes processor time.

Model Answer

Under programmed input and output the processor does everything itself. It asks the device whether it is ready, asks again, and then moves the word by hand. Readiness is found by reading a status register in a loop and testing one bit, and every pass of that loop produces nothing.

The loop, written out: read the status register, test the ready bit, branch back if it is clear. When it is set at last, read the data register, store the word, advance the pointer, decrement the count and begin again for the next word.

Why this is waste and not merely slowness: the processor is fully occupied and completely unproductive. It holds the bus and retires instructions at full speed, and the output of all that effort is an answer it already has. Busy waiting is the proper name for it.

The slowest device sets the pace: the waiting is fixed by the device, not the processor, so a machine twice as fast finishes no sooner. It merely polls twice as often. A keypad yielding one character every 200 milliseconds holds a processor that could have retired a million instructions.

Two journeys for every word: even once the device is ready, the data runs through the processor. The word goes from the device register into a processor register, and only then into memory, crossing the bus twice and costing an instruction each time.

What it is nevertheless good for: it needs no interrupt line, no vector table and no controller. Its timing can be counted in cycles because nothing arrives unannounced, and it is short enough to write correctly first time. On a machine with one job, that is a fair bargain.

Polling buys simplicity with the one thing a processor cannot make more of.

The Master Key: *Programmed input and output costs the whole waiting time and not just the transfer time, because the processor learns that a device is ready by asking rather than by being told.*

BINDER Question 144

At the dispensary in Nkasi there is a hand bell on the counter. The clinical officer does not stand watching the door. He writes up his notes, and when somebody arrives they ring. He finishes his sentence, deals with the patient, returns to it.

Explain interrupt driven input and output, setting out the interrupt cycle step by step, and define maskable and non-maskable interrupts, interrupt priority and the interrupt vector table.

Model Answer

An interrupt reverses who asks. The device raises a signal when it is ready, and the processor, busy with other work, finishes the instruction in hand, saves enough state to return to, runs a short service routine, then restores it and resumes where it was.

The cycle, step by step: the device asserts the request line. The processor completes the current instruction, never abandoning it half done, and tests the line at the end of the execute phase. It acknowledges, pushes the program counter and status register, fetches the routine address, runs the routine, then pops what it pushed.

Maskable and non-maskable: a maskable interrupt can be refused by clearing the interrupt enable flag, which is how a critical section protects itself. A non-maskable interrupt cannot be refused, and is kept for what no program may ignore, such as power failure or a parity error.

Priority: several devices may call at once, and priority settles which is served first and whether a running routine may itself be interrupted. A daisy chain of acknowledge lines or a programmable controller enforces it, and whatever loses data when delayed ranks highest.

The interrupt vector table: a table of service routine addresses held in memory, indexed by interrupt number. The device supplies its number, the hardware turns that into an offset and loads the address found there, so a new device needs a table entry and no change to the processor.

What it buys and what it costs: the waiting disappears, since the processor works until spoken to. In its place come latency, the delay between the request and the routine's first instruction, and the fixed cost of saving and restoring state, paid once per interrupt and ruinous if a device interrupts once per word.

The bell does not make the patient arrive sooner. It makes the waiting somebody else's job.

The Master Key: *An interrupt lets the device announce itself, so the processor pays for the service and never for the wait.*

BINDER Question 145

The warehouse at Momba took forty tonnes of fertiliser, and the manager did not carry a bag. He told the head porter where the lorry stood and where the stack should go, then went back to his ledgers. One knock, two hours later.

Explain direct memory access, covering the three phases of a transfer and the modes burst, cycle stealing and transparent.

Model Answer

Direct memory access gives the whole block to a dedicated controller. The processor programs it with a source, a destination and a count, releases the bus and returns to other work. The controller moves every word untouched, and raises one interrupt when the count reaches zero.

Phase one, set-up: the processor writes the source address, the destination address, the word count and the direction into the controller registers, and starts it. Those instructions are paid once for the block and not once for each word, which is the entire argument for the method.

Phase two, the transfer: the controller requests the bus, the processor grants it and stops driving the address and control lines. The controller drives them itself, moving each word between device and memory, incrementing the address and decrementing the count, while the processor runs on out of its cache.

Phase three, completion: the count reaches zero, the controller releases the bus and raises one interrupt. The service routine reads the status register for errors and tells the waiting program its block has arrived. One interrupt, for thousands of words.

The three transfer modes: burst seizes the bus and holds it until the block is done, fastest of the three but locking the processor out. Cycle stealing takes the bus for one word and hands it back, slowing the processor rather than stopping it. Transparent uses only cycles the processor does not want, costing nothing and taking longest.

What it still costs: memory bandwidth is shared, so every stolen cycle slows other work. Memory changes underneath the processor, so a cached copy of the destination must be invalidated or the region marked uncacheable. And the controller is hardware, to be bought, wired and programmed correctly.

The manager paid two sentences at the start and one knock at the end.

The Master Key: *Direct memory access pays once for the block instead of once for each word, and buys that with a controller and a share of the bus.*

BINDER Question 146

Three offices in Rorya move the same forms. In the first, the officer walks to the registry after every page to see whether the next has come. In the second, the registry sends a boy. In the third, one clerk carries the whole file.

Compare programmed input and output, interrupt driven input and output and direct memory access, and state clearly which kind of transfer suits each.

Model Answer

Two questions separate the three methods: who moves each word, and who decides when. Programmed input and output answers both with the processor. Interrupts leave the moving with it but hand the timing to the device. Direct memory access gives both away, and that is why the ladder runs as it does.

Who moves the data: under the first two methods every word passes through a processor register on its way between device and memory, so the cost per word is instructions. Under the third the controller drives the bus and the processor never sees a word of the block.

Who decides when: polling makes the processor ask and keep asking, which is where all its waste lies. Interrupts and direct memory access let the device speak, so the processor spends the waiting time on other work. The two axes are independent, and these methods are three of their four combinations.

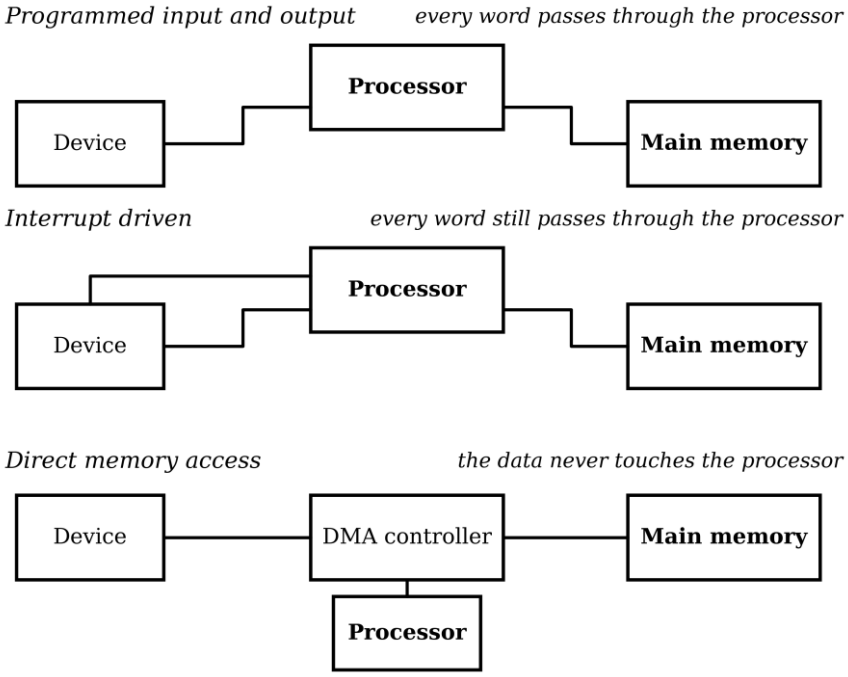


Figure 1.34: *The three input and output methods. Under the first two every word is carried through the processor; under direct memory access the data goes straight to memory and the processor only sets the transfer up.*

What a block costs the processor: polling costs the entire transfer time. Interrupts cost one entry and exit plus a few instructions per word. Direct memory access costs a set-up, one interrupt and stolen cycles, so its bill grows with the number of blocks and hardly at all with their size.

Nothing here is free: read the ladder downwards and processor time falls while hardware rises. Polling needs no line and no chip. Interrupts need a request line, priority logic and a vector table. Direct memory access needs those and a bus master. Time is bought with silicon.

Choosing, in one sentence: poll small, rare transfers on a machine with nothing better to do, interrupt for slow devices whose timing cannot be predicted, and use direct memory access for large blocks from fast devices. A real machine uses all three, one per device.

Method	Who moves the data	Processor cost	Extra hardware	Best suited to
Programmed input and output	The processor, word by word	The whole transfer time, waiting included	None beyond a status register	Small, rare transfers on a machine with nothing else to do
Interrupt driven	The processor, word by word	Entry and exit for each interrupt, plus the moving instructions	Request line, priority logic, vector table	Slow devices whose timing cannot be predicted
Direct memory access	A controller, alone	A set-up, one interrupt for the block, and stolen bus cycles	A controller able to master the bus	Large blocks from fast devices, such as disk, card and network

Table 1.32: *The three input and output methods compared.*

One machine, three methods, chosen device by device and never once and for all.

The Master Key: *Ask who moves each word and who decides when; the answers name the method, and time saved in the processor is paid for in hardware.*

BINDER Question 147

In Uvinza the health post has a number on its door like any house, so a letter addressed in the ordinary way reaches it. The freight depot has no street number at all. Both can be found, but only one by the postman.

Explain memory-mapped input and output and contrast it with isolated, or port-mapped, input and output, giving the price each one charges.

Model Answer

The question is whether device registers live in the memory address space or in one of their own. Memory mapping gives them ordinary addresses, so load and store instructions reach them. Port mapping gives them a separate space with its own instructions and its own control line, spending no memory address at all.

Memory mapping, and what makes it pleasant: a device register is given an address in the memory map, and every instruction that can touch memory can touch it. No new instructions are needed, pointers and structures describe a device naturally, and a driver is ordinary code.

What memory mapping costs: address space first, since every address given to a device is one no memory may use. Then care: those addresses must be uncacheable and must not be reordered or merged, because a register changes with nobody writing to it, and reading it twice differs from reading it once.

Port mapping, and what it protects: input and output live in a separate address space reached only by dedicated instructions, commonly written IN and OUT, and an extra control line tells the hardware which space the bus address belongs to. The whole memory space stays available for memory.

What port mapping costs: a second and poorer instruction set. Ports are usually reachable through one register and with few addressing modes, so no pointer arithmetic and no walking a structure. The processor needs an extra pin, and the programmer must remember which space a number names.

Which is used, and where: memory mapping dominates, because address space is no longer scarce and it asks nothing of the compiler. Port mapping is inherited from designs where sixty-four kilobytes was far too precious to give away, and machines descended from those still support both.

Point of comparison	Memory-mapped input and output	Port-mapped, or isolated, input and output
Address space	Device registers sit in the ordinary memory space	A separate space of its own
Instructions used	Ordinary load and store	Dedicated instructions, commonly IN and OUT
Control lines	The ordinary memory read and write lines	One extra line naming which space is meant
Price paid	Addresses lost to devices for ever	Fewer addressing modes and an extra pin
Caching	Those addresses must be marked uncacheable	Never cached, since the space is not memory
Where it is found	Most modern processors	Machines descended from early sixteen-bit designs

Table 1.33: *Memory-mapped and port-mapped input and output compared.*

One street, or two counters. Address space is the currency, and both prices are real.

The Master Key: *Memory mapping spends addresses to buy ordinary instructions; port mapping spends instructions and a pin to keep its addresses.*

REAL Question 148

The fisheries station on the lake at Nyasa writes a full day of sonar to a card, and the machine that reads it back also runs the pump timers. If reading takes the processor away for the whole transfer, the pumps miss their hour.

Calculate what the block costs the processor under programmed input and output, how much of that is busy waiting, what one interrupt per word would cost instead, and how much direct memory access saves. Show every step.

The block is 4096 words long.

The card delivers one word every 25 microseconds.

Moving one word under programmed input and output costs 2 microseconds of instructions.

One interrupt per word would cost 8 microseconds of entry, service and exit.

Programming the controller costs 40 microseconds, and the completion interrupt costs 60.

Cycle stealing costs the processor 0.5 microseconds for every word the controller moves.

Model Answer

Programmed input and output costs the processor the whole 102400 microseconds of the transfer, of which only 8192 are instructions and 94208 are busy waiting, exactly 92 per cent of the time thrown away. Direct memory access costs 2148 microseconds altogether and so returns 100252, near 98 per cent of the bill.

How long the block takes, whatever the method: 4096 words at 25 microseconds each is 102400 microseconds, which is 102.4 milliseconds. No method beats that, because it is the card that is slow and not the machine. What the methods differ over is what the processor may do meanwhile.

The polling bill: the processor cannot leave the loop until the last word lands, so its cost is that same 102400 microseconds. Notice that this figure is the transfer time itself and not a sum of instruction times, and that is the step most often got wrong.

How much of it was work: 4096 words at 2 microseconds is 8192 microseconds of real instructions. Take that away and 102400 less 8192 leaves 94208 microseconds spent in the loop, and 94208 divided by 102400 is exactly 0.92. So 92 per cent of the morning went on a question whose answer was already known.

Why an interrupt for every word is no rescue: 4096 interrupts at 8 microseconds is 32768 microseconds, four times the instruction cost of polling. The processor does get the other 69632 microseconds back for the pumps, so it is a genuine improvement, but paying entry and exit on every word of a block is the classic wrong use of interrupts.

The direct memory access bill: 40 microseconds to program the controller, 60 for the single completion interrupt, and 4096 stolen cycles at 0.5 microseconds, which is 2048. The total is 2148 microseconds, and of those three terms only the last grows with the length of the block.

The saving, stated properly: 102400 less 2148 is 100252 microseconds handed back to the pumps and the radio, about 97.9 per cent of what polling cost. The card still takes its 102.4 milliseconds, since nothing has made it quicker. What has changed is who spends them.

The station keeps its timers because the carrying was given to hardware with nothing else to do.

The Master Key: *Under polling the processor pays the device's own transfer time; under direct memory access it pays a set-up, one interrupt and a handful of stolen cycles.*

REAL Question 149

The gold buying office at Mbogwe weighs, records and prints, and the card reader interrupts the little machine two thousand times a second. None of that is visible to the clerk. It is visible in the printer queue, which is slower than it was.

Describe exactly what happens to the program counter, the status register, the general registers and the stack when the interrupt arrives, then calculate the overhead of one interrupt, the share of the whole event it represents, and what the reader costs the processor in every second.

Finishing the instruction in hand takes 4 clock cycles.

Pushing the program counter takes 2 cycles, and pushing the status register takes 2.

Reading the vector and loading the program counter takes 6 cycles.

The routine saves 8 general registers at 2 cycles each, and restores them at the same cost.

The body of the service routine is 40 cycles.

The return from interrupt pops the status register and the program counter and costs 6 cycles.

One clock cycle is 0.1 microseconds, and the reader interrupts 2000 times a second.

Model Answer

One interrupt costs 92 cycles end to end, of which only 40 do the work the device asked for. The other 52 are overhead, 56.5 per cent of the event. At 0.1 microseconds a cycle that is 9.2 microseconds each time, and 2000 of them take 18400 microseconds out of every second, or 1.84 per cent of the processor.

The instruction in hand is finished first: the request is a signal on a line and not an instruction, and the processor tests that line at the end of the execute phase. An instruction abandoned half done would leave registers and memory in a state nothing could restore, so 4 cycles go on completing it.

The program counter is saved before it is changed: the counter already points at the next instruction, so pushing it captures precisely the right return address. The status register follows it onto the stack, because the flags belong to the interrupted program and the routine is about to overwrite them. That pair costs 4 cycles.

The vector supplies the new program counter: the device number indexes the interrupt vector table, the hardware reads the routine address held there, and 6 cycles later the program counter holds it. Only now does the first instruction of the service routine execute; everything before this is machinery.

The general registers are saved by the routine itself: the hardware saves only what it must. Any general register the routine will use is pushed by the routine and popped before it returns, 8 registers at 2 cycles each, so 16 in and 16 out. The stack grows by the counter, the status register and those 8, and returns to its old height.

The arithmetic of one interrupt: entry is 4 plus 2 plus 2 plus 6 plus 16, which is 30 cycles. Exit is 16 plus 6, which is 22. Together they are 52 cycles of overhead wrapped around 40 cycles of service, so the whole event is 92 cycles, and 52 divided by 92 is 56.5 per cent of it.

What the reader costs every second: 92 cycles at 0.1 microseconds is 9.2 microseconds, and 2000 a second is 18400 microseconds, or 18.4 milliseconds. That is 1.84 per cent of every second, and it is the printer queue's missing time. Buffer the reader to interrupt once a line instead of once a character and almost all of it returns.

Nothing was changed except how often the machine is asked to put its work down and pick it up again.

The Master Key: *An interrupt costs a fixed entry and exit whatever it carries, so the overhead is set by how often it happens and not by how much it moves.*

REAL Question 150

Three answers came back from a class at Nyang'hwale, each written with confidence and each wrong in a way that costs data in a workshop. They are worth more than three correct answers, because the mistakes inside them are the ones almost everybody makes.

Identify what is wrong with each of the three claims and give the correction in full.

"Direct memory access removes the processor from the transfer entirely, so it costs the processor nothing at all."

"An interrupt and an exception are the same thing."

"Under memory-mapped input and output a device register can be cached like ordinary memory."

Model Answer

All three are wrong. Direct memory access is cheap and not free, an interrupt and an exception differ in origin and in timing, and a device register must never be cached. The three errors share one habit, which is believing that work done by hardware is work nobody pays for.

The first claim, at the start of the transfer: the processor programs the controller, writing a source, a destination, a count and a direction, and then starting the device. Those are instructions it executes. Begin a hundred small transfers and the set-up is paid a hundred times, which is why direct memory access is a poor way to move four words.

The first claim, during and at the end: controller and processor share one bus, so every stolen cycle is a cycle the processor waits on memory, and at the end there is one interrupt with its own entry and exit. Write the correction thus: the cost is small, fixed per block rather than per word, and never zero.

The second claim, where the two really differ: an interrupt arrives from outside the running program, raised by a device that had no idea what was executing, and it is asynchronous, since it could have landed one instruction earlier or later. An exception is raised by the instruction being executed, is synchronous, and falls at the same place every time the program runs on the same data.

The second claim, and why they are confused: a divide by zero, an illegal opcode, an overflow and a page fault are exceptions, and a system call is an exception raised deliberately. They are muddled with interrupts because the machinery is shared: both vector through the same table and both save state the same way. Same mechanism, opposite cause.

The third claim, and the stale value: a status register changes because the device changed it, with the processor writing nothing. Cache that address and the processor reads its own copy for ever, sees a ready bit cleared minutes ago, and either hangs waiting or takes a word that never arrived.

The third claim, and what the correction shows: such addresses must be marked uncacheable, and their accesses must not be merged or reordered either, since reading a register twice can do what reading it once does not. That restriction is the price memory mapping charges for the convenience of ordinary load and store.

Each of the three assumed that work moved elsewhere is work that stopped costing. It never is.

The Master Key: *Direct memory access is cheap and not free, an interrupt is asynchronous while an exception is raised by the instruction itself, and device registers must be uncacheable.*

HOT Question 151

The pump house above Busega has one job: read a float switch and start the motor when the level is low. The man who programmed it chose the crudest method there is. Ask why and he will ask what else the processor would do.

"Polling is obsolete." To what extent do you agree? Argue both sides and reach a judgement.

Model Answer

Largely true of general purpose machines, false about polling itself. Polling lost its place as the default because a processor with other work cannot afford the waiting time. It keeps one, and has lately won some back, wherever the wait is short, the timing must be provable, or events come too fast.

The case against, in one figure: the cost of polling is the whole waiting time, not the transfer time. A device ready once in 10000 cycles wastes the other 9999. On a machine running many programs that is not inefficiency but theft from every other program.

And it scales badly: polling several devices means a loop visiting each in turn, so the response owed to any one worsens as devices are added, and a device ignored while another is serviced can lose data. Interrupts cost nothing until something happens, and grow with events, not devices.

The case for, first: no latency: an interrupt spends entry and exit before the first useful instruction runs. A polling loop already watching the status register responds in the cycle after the bit is set. Where a deadline is microseconds long, the interrupt machinery is itself the delay you cannot afford.

Second: timing you can count: a polling loop has no arrivals to disturb it, so its worst case equals its average, and both can be counted from the instruction timings. Interrupts make execution time depend on what the outside world did, which a hard real time argument cannot tolerate.

Third: no hardware, and no real cost when idle: polling needs no request line, no priority logic and no vector table, which matters where pins and gates are the budget. And when the processor has nothing else to do, the wasted time has no other use, so a cost that looks total is zero in fact.

Fourth, and newest: at high rates interrupts cost more: a network interface taking hundreds of thousands of packets a second can spend the whole machine on entering and leaving service routines, a failure called an interrupt storm. The cure is one interrupt, then polling until the queue drains, then arming it again.

The judgement: obsolete is the wrong word for a technique the fastest drivers deliberately switch into. What is obsolete is polling as the general default on a machine with other work, and that much of the claim stands. It has been demoted from standard method to specialist tool, kept for tight loops, dedicated controllers and rates so high that being told costs more than looking.

The pump house programmer was not behind the times. He read the bill for interrupts and declined it.

The Master Key: *Interrupts win when the wait is long and unpredictable; polling wins when it is short, the timing must be provable, or events arrive faster than interrupts can be afforded.*

HOT Question 152

The maize drying plant at Butiama is buying one controller for three jobs: a keypad touched twice an hour, eight probes read on a strict two second tick, and a card that swallows the day's log at closing. The supplier offers one answer.

Assess how the designer of this plant controller should choose between programmed input and output, interrupt driven transfer and direct memory access, weighing transfer size, device speed, cost and the processor's other work, and reach a judgement.

Model Answer

There is no one right method here, and a designer who picks one for the whole machine has answered the wrong question. Four things settle it device by device: how much moves at a time, how fast the device is, what else the processor carries, and what the hardware costs.

Transfer size decides first: set-up is paid per block, instructions per word, so size tips the balance. The card moves a whole day of logging in one transfer, thousands of words, against which a set-up vanishes. The keypad delivers one byte, and a controller costs more than fetching it.

Device speed against the processor: keypad and probes are slower than the processor by orders of magnitude, so waiting on them is unthinkable and being told is cheap. The card runs near bus speed, where an interrupt per word costs more than the word is worth, and that is what direct memory access is for.

The processor's other work: this controller must hold a drying temperature, so its processor is never idle and every microsecond of waiting is taken from control. Were the machine only a card reader, polling would cost nothing real. It is not, so waiting is expensive.

The probes are the interesting case: they must be read on a fixed tick, so predictability matters more than efficiency. Poll them, but on the timer rather than in a free loop: the interrupt arrives, the routine reads all eight in turn, and the wait for each conversion is provably bounded. Polling used deliberately, inside an interrupt.

Cost, on a machine built in dozens: a bus mastering controller is silicon and pins, and on a design of this size that price is paid on every unit built, never spread over millions. Pay it for the card, where it saves most of a transfer, not for the keypad, where it saves nothing measurable.

What the wrong answers look like: direct memory access for the keypad is expensive nonsense, and an interrupt per word of the card would flood the processor at the hour the drying loop must not be disturbed. Polling the keypad would burn hours to catch two presses.

The judgement: this plant needs all three: an interrupt for the keypad, timed polling for the probes, direct memory access for the card. Change the case and the answer moves with it: make it a dedicated reader with nothing else to do and polling the card becomes defensible. The method belongs to the device and the workload, never to the machine.

The supplier wanted one answer because one answer is easier to sell. The plant needs three, each cheap for its own reason.

The Master Key: *Size, speed, cost and the processor's other work choose the method, device by device, which is why one machine properly uses all three.*

HOT Question 153

Stand at the falls at Kalambo and watch the water do in a second what a line of people with buckets could not do in a day. The work was not abolished, only handed to something built for it and nothing else.

Evaluate the claim that the whole history of computer architecture is the story of moving work away from the processor.

Model Answer

Largely true, and told the wrong way round. Everything here takes something off the processor: cache takes the waiting, direct memory access the copying, decoders and multiplexers the choosing, parallel machines whole instruction streams. Yet the processor also swallowed the units that once sat outside, so what moves is work off the general path onto a specialised one.

Cache moved the waiting: the processor once waited on memory at its own speed. A cache does not make memory faster; it takes the waiting off the critical path by keeping likely words nearer, on the evidence of locality. The processor lost a wait, not a task.

Direct memory access moved the copying: a block that once passed word by word through a processor register now passes between device and memory unseen. The processor kept the decision, what to move and where; it gave away the labour.

Multiplexers and decoders moved the choosing: selecting one line out of many, or turning a code into the line it names, is work a processor would do with tests and branches. Cast into gates it happens in one gate delay, with no instructions.

Adders, counters and controllers moved the routine: a carry chain adds without a program, a counter sequences without one, and a state machine walks a controller through its phases. Each is a job once done by instructions and now done by wire.

Parallelism moved whole instruction streams: the classes of machine met at the start of this chapter answer one question: what can be taken off a single processor and given to another. Not a task inside a stream, but the stream itself.

The other side: what the processor absorbed: floating point arithmetic was once a coprocessor and now sits on the die. Cache was once external chips and is now three levels deep inside. Address translation, the interrupt controller and the memory controller were outside and are now within. The boundary moved inwards; the processor grew to meet it.

The judgement, and the point of the chapter: the claim is right about the instruction stream and wrong about the chip. Work has left the instruction sequence permanently, but not the processor's address, because integration keeps drawing it back in. Architecture is the steady removal of work from the general path onto specialised hardware, wherever that hardware lives. Cache, direct memory access, a multiplexer and a second core answer one question.

That question is who should do this, and where the waiting should go. It has no final answer, and every machine you meet is somebody's attempt at it. You now know enough to read one and say what it cost.

The Master Key: *Work leaves the instruction stream, not the chip: architecture moves labour from the general path onto specialised hardware, and integration keeps pulling that hardware back inside.*

Chapter Two

PERFORMANCE AND OPTIMISATION OF THE PROCESSOR

Every program in this book, and every program you will ever write, runs on a processor whose speed is not a slogan but a measurement. Chapter One told you what the parts of the machine are and how an instruction travels through them. This chapter asks the harder question: how quickly does that journey happen, and what can a designer do to make more journeys happen at once? The answer decides whether a payroll runs overnight or over a weekend, whether a school laboratory feels usable or exhausting, and whether a program you have written well is served by hardware that can keep up with it. By the end of this chapter you will not say that a processor is fast. You will say how fast, in cycles, in nanoseconds, and in a speed-up figure you can defend.

The chapter runs as one line of argument. It opens with the pipeline: the idea of overlapping instructions, the five stages IF, ID, EX, MEM and WB, and the registers that separate them. It then turns that idea into arithmetic, with clock periods, cycle counts, the formula for the cycles a pipeline needs and the speed-up that follows from it. Next come the three things that break the smooth flow, structural hazards, data hazards and control hazards, each with the remedies a designer actually uses. From there the chapter widens: parallelism at the level of instructions, then at the level of threads, then at the level of data, each one lifting a ceiling the one before it could not reach. Last comes the law that limits all of them.

Two habits carry most of the marks in this topic. The first is to reach for a number. Where a question hands you a clock speed, a stage count or an instruction count, the answer that earns is arithmetic and not adjectives; a candidate who writes that the pipelined machine is faster collects very little, while one who writes that it finishes in seven cycles instead of fifteen, a speed-up of 2.14, has answered what was asked. The second habit is to name the stage or the unit before stating the property. Forwarding belongs to a path, a stall belongs to a stage, a branch resolves at a particular point. A true fact attached to the wrong stage earns nothing at all, and that is where careful students lose marks they had already done the thinking to win.

Pipelining: the idea, the five stages and the overlap

This section builds the pipeline from nothing. It defines what overlapping actually means, names the five stages IF, ID, EX, MEM and WB and the work each one does, explains why four registers must sit between them, and separates pipelining from the parallelism that arrives later in the chapter. It then counts cycles: three instructions in fifteen cycles without a pipeline, seven cycles with one, and a speed-up of 2.14 that falls a long way short of the ideal of five, for a reason worth understanding early.

BINDER Question 1

For years the maize mill at Urambo finished one sack before the next was tipped in: grind, sieve, bag. The morning the owner let a second sack begin while the first lay under the stone, nothing ground faster, yet the queue was gone by noon.

Define pipelining and explain how it raises the rate at which a processor completes instructions without making any single instruction faster.

Model Answer

Pipelining is the overlapping of the execution of several instructions on one set of hardware, each instruction occupying a different stage at the same moment, so that a finished result leaves the processor every cycle even though no single instruction takes less time than it did before.

What is actually overlapped: the stages, not the instructions. A five stage pipeline holds up to five instructions at once, one in each stage, so that while the first is in EX the second is in ID and the third is in IF. Each piece of hardware works on a different instruction in the same cycle instead of standing idle.

Latency does not change: one instruction still enters at IF and leaves at WB five cycles later, and that end to end time is its latency. Pipelining does not shorten the path. The register on each boundary adds a small overhead, so a single instruction is if anything very slightly slower than it would be on the unpipelined machine.

Throughput is what rises: throughput is the number of results completed per unit of time. Without a pipeline a result appears once every five cycles. Once the pipeline is full a result appears every cycle, so the completion rate climbs towards five times its old value while each individual instruction takes exactly as long as it always did.

The intuition to abandon: students are often taught, wrongly, that a five stage pipeline makes every instruction five times faster. It does not. The journey is the same length; what has changed is how many journeys are under way at once. Say rate rather than speed and the distinction will hold in every question that follows.

Pipelining buys you instructions per second, and it buys them without touching the seconds any one instruction needs.

The Master Key: *Pipelining overlaps stages on one set of hardware: latency stays the same, throughput rises towards one instruction per cycle.*

BINDER Question 2

Five sounds come from the printing shop at Kondoa, and a customer on the veranda can tell how far her order has travelled without looking through the door. The guillotine means one thing, the stapler another, and they never come in a different order.

State the five stages of a classic pipeline in order and describe the work done in each.

Model Answer

The five stages of a classic pipeline are IF, ID, EX, MEM and WB, always in that order: instruction fetch, instruction decode, execute, memory access and write back.

IF, instruction fetch: the processor reads the instruction from memory at the address held in the program counter, and advances the program counter so that it points at the next instruction. Nothing has yet been understood about the instruction; at the end of IF it is only a pattern of bits sitting inside the machine.

ID, instruction decode: the control unit works out what the instruction is and what it will need, and at the same time the operands are read out of the register file. Decoding and register reading can happen together because the register numbers sit in fixed fields of the instruction word.

EX, execute: the arithmetic and logic unit does the work. For an arithmetic instruction it computes the result. For a load or a store it computes the memory address by adding the offset to the base register. For a branch it evaluates the condition and works out the target address.

MEM, memory access: data memory is read or written. A load collects its value here and a store deposits one here. An instruction that touches no memory, such as an ordinary register to register addition, passes through MEM doing nothing at all, because every instruction must occupy every stage if the timing is to stay regular.

WB, write back: the result is written into the destination register in the register file. An arithmetic result arrives from EX and a loaded value arrives from MEM, but both are written here, and only when WB is finished is the instruction complete.

Name them in order and abbreviate them exactly, because a correct description attached to the wrong stage earns nothing.

The Master Key: *IF fetches, ID decodes and reads registers, EX computes, MEM touches data memory, WB writes the destination register.*

BINDER Question 3

A clerk at a mobile money kiosk in Ngara passed the slip across while still writing the last digit. His colleague keyed in what he read, and the customer lost nine thousand shillings to a number not yet finished. Every pipeline boundary faces that danger.

Explain why a pipeline needs registers between its stages, and state how many a five stage pipeline requires.

Model Answer

A pipeline needs a register on every boundary between one stage and the next to hold that stage's result steady while the following stage reads it, and a five stage pipeline therefore requires four of them.

Why a boundary is dangerous: the logic inside a stage does not produce its answer all at once. Signals ripple through the gates and settle only towards the end of the cycle. Wire the next stage straight to that output and it reads a value that is still changing, and acts on rubbish.

What the register does: at the clock edge it captures whatever the stage has produced and holds it unchanged for the whole of the next cycle. The following stage then reads a value that cannot move under it. One clock edge, one clean handover, repeated every cycle for the life of the machine.

How many, and where: four, one on each boundary: IF to ID, ID to EX, EX to MEM and MEM to WB. In general a pipeline of k stages needs k - 1 registers, because the boundaries lie between the stages and not at the two ends.

one clock cycle per stage; the latch holds the result until the next tick

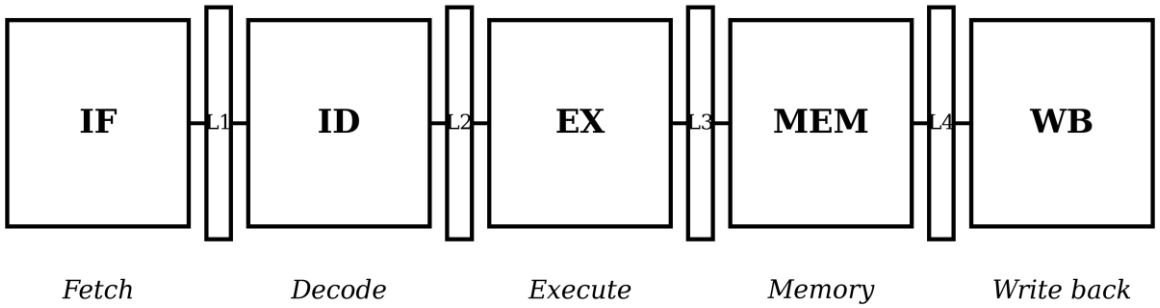


Figure 2.1: *Five stages and the four registers that separate them. Each register holds one stage's result steady while the next stage reads it.*

What travels inside it: more than the arithmetic result. The register also carries the destination register number and the control signals the later stages will need, so each instruction takes its own orders down the pipeline with it and no stage has to remember which instruction it is serving.

The price of the register: the clock period must cover the slowest stage plus the time the register needs to set up and settle. That overhead is fixed, so as stages are made shorter it becomes a larger share of every cycle, and it is one reason a pipeline cannot be deepened without limit.

Draw the four registers whenever you draw the pipeline, because five bare boxes in a row are not a pipeline.

The Master Key: *Five stages need four registers, one per boundary, each holding a result steady for exactly one cycle.*

BINDER Question 4

Two tailoring workshops in Same finish the same number of school uniforms each day. In one, four benches in a line take a uniform from cutting to pinning to sewing to pressing. In the other, four tailors each make a whole uniform alone.

Distinguish pipelining from parallelism.

Model Answer

Pipelining overlaps different stages of different instructions on one copy of the hardware, while parallelism replicates the hardware so that two instructions can occupy the same stage at the same moment.

Pipelining keeps one of everything: one fetch unit, one decoder, one execution unit, one memory port and one write back path. In any cycle exactly one instruction sits in each stage. Nothing has been duplicated. What pipelining exploits is that those five units used to stand idle four cycles out of every five.

Parallelism buys a second copy: two execution units mean two instructions can be in EX in the very same cycle, and two whole cores mean two instruction streams run side by side. The hardware bill rises, and so does the number of instructions in flight at any instant.

The ceiling pipelining cannot pass: one instruction per cycle. Only one instruction can be in WB in any cycle, and WB is where an instruction finishes, so a five stage pipeline completes at most one instruction per cycle however long the queue of waiting instructions grows. Hazards can push that rate down, never up.

Where the ceiling is broken: by parallelism, which is why Sections 6, 7 and 8 lift the same limit at the level of instructions, of threads and of data in turn. Pipelining is the foundation those techniques are built on rather than a rival to them, and a real processor is pipelined and parallel at once.

Ask one question of any design you are shown: was the hardware duplicated? If it was not, you are looking at a pipeline.

The Master Key: *Pipelining overlaps different stages on one set of hardware; parallelism duplicates hardware so two instructions share a stage.*

BINDER Question 5

Nobody on the rice threshing floor at Kyela knew why three of the four workers spent part of every round standing still. One beats, one winnows, one weighs, one bags. Then somebody timed the beating, and it took twice as long as anything else.

Explain why every stage of a pipeline must be given the same amount of time, and state the effect on the clock.

Model Answer

Every stage must be given the same amount of time because one clock tick has to be long enough for the slowest stage to finish its work, so the clock period is set by that slowest stage and every faster stage simply spends the remainder of the tick waiting.

One clock, one period, no exceptions: the pipeline advances on a single clock and every stage hands its result to the next register on the same edge. There is no way to give EX a long tick and IF a short one, so the period is chosen to satisfy the worst case, which is the slowest stage plus the register overhead.

The fast stages pay for the slow one: a stage that finishes early cannot begin the next instruction, because the instruction behind it has not arrived yet. It holds its output and idles. That idle time is never recovered later; it is spent again on the next cycle, and on every cycle the machine ever runs.

A concrete illustration: suppose the five stages need 2, 2, 4, 2 and 2 nanoseconds. The clock period must be 4 nanoseconds, so all five are clocked at 4. Five ticks of 4 nanoseconds is 20 nanoseconds of clock for 12 nanoseconds of real work, and 8 nanoseconds in every 20, two fifths of the machine, is thrown away.

Balance is worth more than depth: split that 4 nanosecond stage into two stages of 2 and the period falls from 4 to 2, doubling the rate at which results leave the pipeline without touching any other stage. A designer therefore hunts the bottleneck stage first, because depth added anywhere else changes nothing at all.

The clock is a promise made to the slowest stage, and every other stage has to keep it.

The Master Key: *The clock period equals the slowest stage plus register overhead, so an unbalanced pipeline wastes the difference on every cycle.*

REAL Question 6

Fifteen minutes for three lorries at the weighbridge outside Nachingwea. Then the operator changed nothing except the order in which the three checks were done, and the same three lorries cleared in seven. No lorry had been weighed any faster than before.

Trace three instructions through a five stage pipeline and through a machine with no pipeline, count the cycles each takes, and calculate the speed-up.

Model Answer

Three instructions take 15 cycles on a machine with no pipeline and 7 cycles on a five stage pipeline, so the speed-up is 15 divided by 7, which is 2.14.

With no pipeline each instruction owns the machine: the first instruction runs IF, ID, EX, MEM and WB, and only when it has left WB does the second begin its IF. Every instruction therefore costs a full five cycles of its own, and three instructions cost 3 x 5, which is 15 cycles.

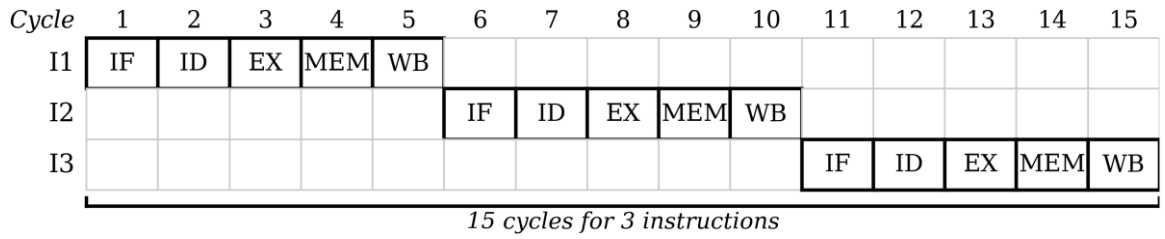
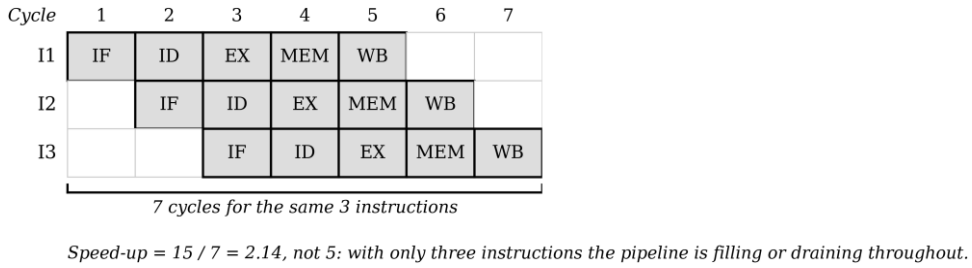


Figure 2.2: *Three instructions on a machine with no pipeline. Each waits for the one before it to finish, so the work takes fifteen cycles.*

With a pipeline a new instruction starts every cycle: the first instruction enters IF in cycle 1, the second in cycle 2 and the third in cycle 3. The first leaves WB at the end of cycle 5, the second at the end of cycle 6 and the third at the end of cycle 7. Seven cycles in total.



Speed-up = 15 / 7 = 2.14, not 5: with only three instructions the pipeline is filling or draining throughout.

Figure 2.3: *The same three instructions overlapped. Seven cycles, because the pipeline is filling or draining throughout.*

The formula behind the seven: cycles for N instructions in a k stage pipeline is N + k - 1. The first instruction pays the full k cycles to reach the end, and each instruction after it adds one further cycle. With N = 3 and k = 5 that gives 3 + 5 - 1 = 7, which is exactly what the trace shows.

The speed-up itself: divide the unpipelined time by the pipelined time. 15 divided by 7 is 2.14, so the pipelined machine finishes the same work in a little under half the time. Written in general, the speed-up is kN divided by N + k - 1.

$$\text{Speed} - \text{up} = \text{kN} / (\text{N} + \text{k} - 1) = 5 \times 3 / (3 + 5 - 1) = 15 / 7 = 2.14$$

Why 2.14 falls so far short of 5: because with only three instructions the pipeline is filling or draining in every one of the seven cycles and never has all five stages busy. At its fullest moment, cycle 5, only three stages hold an instruction. The ideal speed-up of 5 is a limit approached as the instruction count grows large, and three is not large.

What large would look like: run 1000 instructions instead. The pipelined time is 1000 + 5 - 1 = 1004 cycles against 5000 unpipelined, a speed-up of 4.98. The fill and drain cost of 4 cycles has not changed at all; it has simply been spread over a thousand instructions rather than three.

Quote both cycle counts and the ratio between them, and always say which cycles went on filling and draining.

The Master Key: *Cycles are N + k - 1 and speed-up is kN over N + k - 1, so three instructions on five stages give 15 over 7, or 2.14.*

REAL Question 7

Three sentences about pipelining were on the board of a Form Five class at Igunga, and the class was asked to choose the true one. Most of the room chose the first, because it says what everybody believes a pipeline is for.

Three statements about pipelining are given below. Identify the one that is correct, and explain precisely what is wrong with each of the other two.

Statement A: A five stage pipeline makes every instruction five times faster to execute.

Statement B: A pipeline with more stages always finishes a program sooner than one with fewer stages.

Statement C: A five stage pipeline can complete at most one instruction per clock cycle, however many instructions are waiting.

Model Answer

Statement C is correct: a five stage pipeline can complete at most one instruction per clock cycle, however many instructions are waiting.

Why C is correct: there is one copy of each stage, so only one instruction can be in WB in any cycle, and WB is where an instruction finishes. The length of the queue behind it makes no difference. A full pipeline retires one instruction per cycle and no more, and hazards can only push that rate below one.

Statement A confuses throughput with latency: it is false. The instruction still passes through IF, ID, EX, MEM and WB, and it still takes five cycles from beginning to end, or a shade longer once the register on each boundary is counted. Nothing whatever has been done to make one instruction execute faster.

What A should have said: that a five stage pipeline can complete up to five times as many instructions in a given time, once it is full. That sentence is about the rate at which results leave the machine, which is throughput. A attaches that rate to latency, the time one instruction spends inside the pipeline.

Statement B fails on fill and drain: it is false. Cycles are N + k - 1, and that figure grows with k. A twelve stage pipeline spends 11 cycles filling and draining against the 4 a five stage pipeline spends, so over a short run of instructions the deeper machine finishes later rather than sooner.

Statement B fails on branches as well: a branch is not resolved until part way down the pipeline, and everything fetched behind it must be discarded when the guess was wrong. The deeper the pipeline, the more work is thrown away each time. There is a depth beyond which a program runs slower, and the word always is what kills the statement.

One statement in three was true, and the two false ones failed for the two reasons this chapter keeps returning to: latency is not throughput, and depth is not free.

The Master Key: *A pipeline retires at most one instruction per cycle; deeper is not automatically sooner, and faster completion is not faster instructions.*

HOT Question 8

A proposal reached the chief engineer on the road works at Makambako: split every task into twelve smaller ones instead of five and the tarmac will go down faster. The front page arithmetic was persuasive. The back page, where somebody counted the handovers, was not.

A processor designer proposes to increase the pipeline from five stages to twelve. Assess whether the change is likely to be worthwhile, and state the condition under which you would approve it.

Model Answer

The change is worthwhile only for a workload of long, straight runs of independent instructions in which branches are rare or well predicted; on branch-heavy code the twelve stage pipeline loses more to fill, drain and misprediction than it gains from the shorter clock period, and I would refuse it.

The case for depth is the clock: each stage does less work, so the slowest stage shrinks and the clock period, which is set by that stage, shrinks with it. Since a full pipeline still retires one instruction per cycle, shorter cycles mean more instructions completed per second. That is a real and substantial gain.

The clock does not rise by the full factor: twelve over five is 2.4, but no designer collects 2.4. The work inside a stage does not divide neatly, and the register on each boundary costs a fixed setting up and settling time that does not shrink when the stage does. As stages get shorter that fixed overhead takes a larger share of every cycle.

Fill and drain becomes expensive: cycles are N + k - 1, so the overhead beyond N is k - 1: four cycles at five stages, eleven at twelve, which is 2.75 times as much. Code that runs in long uninterrupted streams pays this once. Code that is constantly called into, returned from and interrupted pays it over and over.

The branch penalty scales with depth: a branch resolves part way down the pipeline and everything fetched behind it on a wrong guess is discarded. Resolve in the third stage of five and two instructions are lost; resolve in the ninth stage of twelve and eight are lost. A misprediction every few hundred instructions makes that a continuous charge.

Hazards also cost more: producer and consumer instructions sit further apart in a deep pipeline, so a dependence that forwarding cannot cover stalls for longer before the value it needs is available.

The verdict: approve the twelve stage design for a machine that will run long, predictable, independent instruction streams, such as signal processing or bulk numerical work. There the pipeline stays full, the fill cost is spread over millions of instructions, and the shorter period is almost all profit.

The condition for refusing: branch-heavy interactive code with an ordinary predictor. There the misprediction penalty grows with depth faster than the clock gain arrives, and the twelve stage machine finishes real programs later while looking better on paper.

Depth is not an improvement in itself; it is a trade of fill, drain and branch cost against clock period, and the workload settles which side wins.

The Master Key: *Deepen a pipeline only where code runs long and branches rarely, because the misprediction penalty grows with depth while the clock gain does not.*

HOT Question 9

Two sealed cartons stood in the storeroom of a secondary school at Pangani, priced the same, with money for one. The label on the second promised 3.6 GHz against 2.4 on the first, and the head of department was ready to sign on that line alone.

Two processors are offered at the same price. One has a three stage pipeline and a 2.4 GHz clock. The other has an eight stage pipeline and a 3.6 GHz clock. Discuss which you would choose for a school computer laboratory, and justify your choice.

Processor A: three stage pipeline, clock 2.4 GHz.

Processor B: eight stage pipeline, clock 3.6 GHz.

Model Answer

I would buy Processor A, the three stage machine at 2.4 GHz, because laboratory work is short, interactive and thick with branches, and that is precisely the workload in which Processor B gives back its clock advantage in filling, draining and mispredicted branches.

What the clock alone says: 3.6 divided by 2.4 is 1.5, so B ticks half as fast again as A. If both machines retired one instruction on every tick without interruption, B would finish any program in 0.67 of the time A needs, and there would be nothing left to discuss.

The depth is why the clock is high: B is not fast in spite of its eight stages; it is clocked fast because of them. Each stage does less work, so the slowest stage is shorter and the period can be shorter. Depth and clock speed are one decision presented as two, and the costs of the depth arrive with the clock.

What a laboratory actually runs: editors, browsers, an interpreter, and short student programs that are compiled, run, corrected and run again. That code is dense with conditional branches, calls and returns, and it is interrupted constantly by keystrokes, screen updates and the switching of one program for another.

Every one of those events punishes depth: each interruption and each switch empties the pipeline and refills it, and filling costs k - 1 cycles: two on A and seven on B. Each mispredicted branch discards the instructions fetched behind it, and B, resolving branches further down, discards more of them than A ever does.

The honest case for B: half as much clock again is a genuine margin and should not be waved away. If B carries a good branch predictor, and if the students spend their waiting on compilation rather than on keystrokes, B wins, because a long compile is straight-line work and straight-line work is where depth pays.

The verdict: buy A. In an interactive laboratory the shorter penalty on every branch and every switch is worth more than half as much clock again, and A will feel quicker at the keyboard, which is what the students actually experience.

The measurement that would settle it: do not argue it, measure it. Put the real workload on both machines, an editor, a compile of a typical student program and a run of it, and compare completion time. Clock speed is an input to performance; completion time on the actual work is the performance.

Two machines, one price, and the deciding evidence is a stopwatch held over the work the school will really do.

The Master Key: *Clock speed is an input and not an answer: choose on measured completion time for the actual workload, and prefer the shallower pipeline where code branches often.*

The numerical spine: clock, cycles, CPI, throughput and speed-up

Section one drew the pipeline; this section makes it calculable. It defines the clock in both of the forms you will meet it in, then cycles per instruction, instructions per cycle, latency and throughput, and it puts a number on every one of them. From there it derives the two results the rest of the chapter leans on, that N instructions need $N + k - 1$ cycles and that the speed-up is $kN / (N + k - 1)$, and it works them at 4, 8, 10, 100 and 1000 instructions. By the last question you can be handed a clock speed, a CPI and an instruction count and say which of two processors finishes first, and why the clock speed was the least useful of the three.

BINDER Question 10

A ceiling fan in a Kilwa dispensary turns about one revolution a second, slow enough to count. The processor in the registration desk beneath it completes two and a half thousand million cycles in that same second, and nobody can count anything at that rate.

Define clock cycle time and clock frequency, state the relationship between them, and convert between the two.

Model Answer

Clock cycle time is the duration of a single tick of the processor clock, measured in seconds, and clock frequency is the number of those ticks in one second, measured in hertz; the two are reciprocals, so one divided by either gives the other.

Cycle time, and the units it is written in: the cycle time is how long one clock period lasts, and because it is such a small fraction of a second it is quoted in nanoseconds, thousand millionths of a second, or in picoseconds, a thousand times smaller again. Every stage of the pipeline must finish its work inside one cycle time, so the slowest stage decides how short that time may be.

Frequency, and what a hertz counts: one hertz is one cycle per second, a megahertz a million and a gigahertz a thousand million. Frequency is the figure printed on a specification sheet, because a large number reads better than a small one, though it carries what the cycle time carries.

$$f = 1 / T \quad \text{and} \quad T = 1 / f$$

Frequency into cycle time: a clock of 2.5 GHz ticks 2 500 000 000 times in one second, so one tick lasts 1 / 2 500 000 000 of a second. That works out at 0.4 nanoseconds, and it is the whole of the real time any one pipeline stage is given to do its work in.

Cycle time into frequency: a machine whose cycle time is 0.5 nanoseconds fits 1 / 0.0000000005 cycles into a second, which is 2 000 000 000 of them, so its clock runs at 2 GHz. The conversion works in both directions because nothing has been added by it; a reciprocal is a rearrangement, not a fresh measurement.

What neither number tells you: how much work one tick achieves. A clock says how often the machine is prodded, not what it manages between prods, and the two questions that follow supply the number that fills that gap.

Convert first and calculate second, because a performance question that begins in hertz almost always ends in seconds.

The Master Key: *Frequency and cycle time are one fact written two ways, and one divided by either gives you the other.*

BINDER Question 11

Twelve hundred instructions, fifteen hundred cycles. Those two counts, taken from a school laboratory machine in Kondoa running a sorting program, settle what the machine is worth, and the verdict is not the one its clock speed promised.

Define cycles per instruction and instructions per cycle, and explain what each tells you that the other does not.

Model Answer

Cycles per instruction is the total number of clock cycles a program consumes divided by the number of instructions it completes, and instructions per cycle is its reciprocal; CPI tells you what an instruction costs and IPC tells you what a cycle delivers.

How each one is measured: count the cycles a run takes, count the instructions it completed, and divide. A program that finishes 1200 instructions in 1500 cycles has a CPI of 1.25 and therefore an IPC of 0.8. Both are averages taken over a whole run, and neither is a property of any single instruction.

The ideal, and what lifts it: an in-order pipeline that never stalls completes one instruction in every cycle once it is full, so its CPI tends towards 1. Every stall adds cycles without adding instructions, which gives the rule that carries the rest of this chapter.

$$\text{CPI} = 1 + \text{stalls per instruction}$$

Why CPI cannot fall below one here: a single-issue pipeline has one issue slot per cycle, so it can never finish more than one instruction in a cycle. Its CPI can never be less than 1, its IPC never greater than 1, and a figure below 1 for such a machine is arithmetic gone wrong.

Where IPC is the better number: a machine that issues several instructions in each cycle can reach a CPI of 0.5, and half a cycle per instruction is an awkward thing to read. Its reciprocal, an IPC of 2, says plainly that two instructions leave in every cycle, which is why wider designs are quoted that way.

The error that produces a meaningless one: dividing five stages by five stages and announcing a CPI of 1. Stages are not cycles and instructions are not stages; CPI is measured from a run of a real program, never read off a drawing of the hardware.

Quote CPI when you are pricing an instruction and IPC when you are rating a cycle, and never quote either without saying which program produced it.

The Master Key: *CPI is one plus the stalls each instruction suffers, and IPC is nothing more than one divided by CPI.*

BINDER Question 12

The clerk at the Pangani ferry office was annoyed by his new ticket machine. A single receipt still took as long to print as it always had, and yet the queue behind the desk was clearing in half the time. Both observations were correct.

Define latency and throughput, and explain the effect of pipelining on each.

Model Answer

Latency is the time from the start of one instruction to the completion of that same instruction, k cycles in a k stage pipeline, and throughput is the number of instructions completed per unit of time, one every cycle in a full pipeline.

Latency is measured on one instruction: follow one instruction from IF through ID, EX and MEM to WB and time the journey. On five stages with a cycle time of 0.5 nanoseconds it takes 5 cycles, which is 2.5 nanoseconds, whatever else is in the pipeline.

Throughput is measured on the stream: stand at the end of the pipeline and count what comes out. Once it is full one instruction completes every cycle, so at 0.5 nanoseconds a cycle the machine finishes 2 000 000 000 instructions per second.

What pipelining does to each: it leaves latency where it was, or a little worse, and multiplies throughput by up to k. The work inside one instruction has not been shortened, only cut into pieces that proceed alongside those of its neighbours.

Why latency rises slightly: a latch sits between each pair of stages, and writing into it and reading out again costs a small setup time at every boundary. Five short stages and four latches take a little longer end to end than one undivided operation did.

The definition to refuse: latency is not the total execution time divided by the number of instructions. That quantity is the average time per instruction, the reciprocal of throughput; on the machine above it is 0.5 nanoseconds, while the latency is 2.5 nanoseconds, five times larger.

Why that error is expensive: it forces the two quantities into one number, so the trade-off that justifies pipelining cannot be stated. A design that lengthens each instruction slightly in order to finish far more of them each second is the ordinary case, and this error hides it.

Ask of any timing figure whether it came from one instruction or from a stream, because the two never collapse into one.

The Master Key: *Pipelining does not make an instruction faster, it makes instructions more frequent.*

REAL Question 13

A student in Mbinga wrote that eight instructions through five stages must take forty cycles, because eight times five is forty. The chart on the facing page of her exercise book showed twelve. She had counted the work correctly and the overlap not at all.

Derive the number of clock cycles a five stage pipeline needs to complete N instructions, and use it to find the cycles needed for 8, for 100 and for 1000 instructions.

Model Answer

A k stage pipeline needs N + k − 1 cycles to complete N instructions, so a five stage pipeline takes 12 cycles for 8 instructions, 104 cycles for 100 and 1004 cycles for 1000.

Getting the first instruction out: nothing can complete before it has passed through every stage, so the first instruction occupies cycles 1 to k and finishes at the end of cycle k. On five stages that is the end of cycle 5, and no amount of overlapping can bring it forward.

Getting all the others out: the second instruction entered one cycle behind the first and stays one cycle behind it the whole way, so it completes at cycle k + 1, the third at cycle k + 2, and so on to the end. There are N − 1 instructions after the first, and each of them adds exactly one cycle.

$$\text{cycles} = k + (N - 1) = N + k - 1$$

Reading the derivation off the chart: Figure 2.4 traces eight instructions through five stages. Four cycles pass while the pipeline fills, four more while every stage is busy and four while it drains, giving 12 cycles altogether, which is 8 + 5 − 1 exactly as the formula promises.

The same formula at scale: for a hundred instructions the count is 100 + 4, which is 104 cycles, and for a thousand it is 1000 + 4, which is 1004. The added term never grows, because it depends on the depth of the pipeline and not at all on the length of the program.

What the four extra cycles are worth: at 8 instructions they are 4 cycles out of 12, so a third of the run is spent filling and draining. At 100 instructions they are 3.8 per cent of the run and at 1000 they are 0.4 per cent, which means the same hardware looks mediocre on a short burst and excellent on a long one.

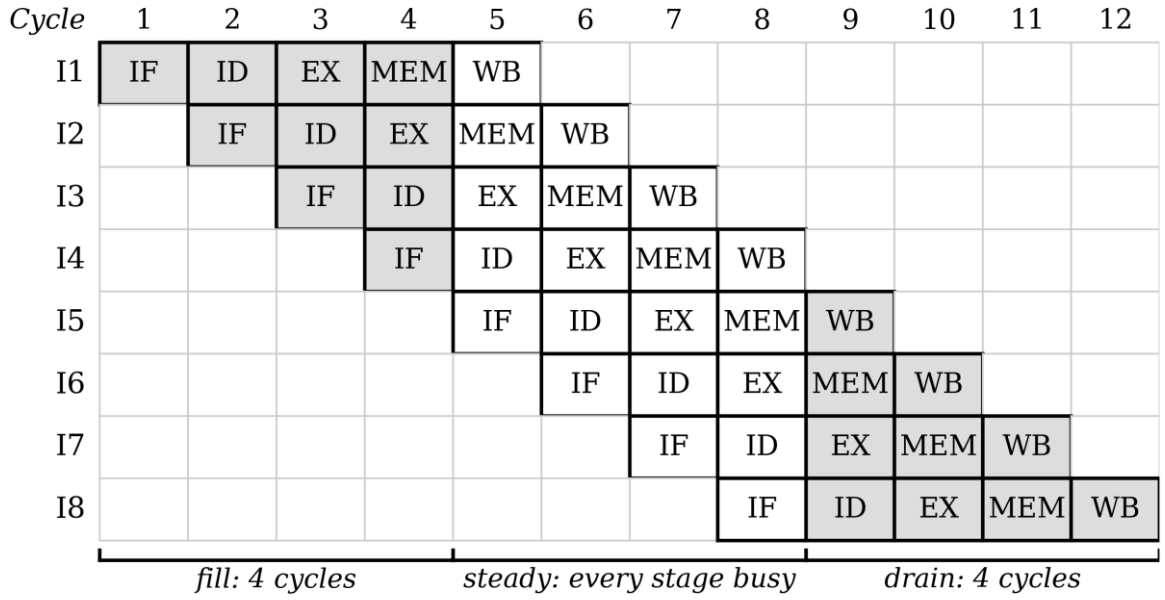


Figure 2.4: *Eight instructions through a five stage pipeline. The pipeline is filling for four cycles, full for four, and draining for four.*

Why this decides where pipelines are used: a pipeline is a machine for long, uninterrupted runs of work. Anything that empties it, a branch or an interrupt, makes the fill cost fall due a second time, which is why later sections spend so much effort on keeping it full.

Count the stages, count the instructions, add the two and subtract one, and you have the cycle count for any pipeline question of this shape.

The Master Key: *$N + k - 1$, in which the N is the work and the $k - 1$ is the price of the pipeline being a pipeline.*

REAL Question 14

On the board of a study group in Igunga are three sentences about latency, copied out of three different places, and two of them are wrong. All three sound reasonable read aloud, which is the difficulty.

Three definitions are given below. Identify the one that is correct, and explain precisely what is wrong with each of the other two.

Statement A: Latency is the total execution time divided by the number of instructions executed.

Statement B: Latency is the time taken for one instruction to pass from the first stage of the pipeline to the last.

Statement C: Pipelining reduces the latency of every instruction, which is why throughput rises.

Model Answer

Statement B is the correct one: latency is the time taken for a single instruction to pass from the first stage of the pipeline to the last, and on a five stage machine that is 5 cycles.

Why B is the one that survives: it fixes the measurement on one instruction and on the whole of that instruction, from IF to WB. Nothing in it depends on how many other instructions are present, which is what makes it usable whether the pipeline is empty, full or stalling.

What A actually defines: total execution time divided by the number of instructions is the average time per instruction, and that is the reciprocal of throughput rather than latency. In the steady state of a five stage pipeline it comes to 1 cycle per instruction, while the latency of every one of those instructions is 5 cycles.

Why A does real damage: it makes throughput and latency into the same measurement, so a machine with excellent throughput and poor latency cannot be described at all. That combination is the normal condition of every pipelined processor ever built, and a definition that cannot express it fails on any question asking what pipelining costs.

Where C runs backwards: throughput rises because instructions overlap, not because any single instruction was made shorter. The five stages of one instruction take precisely as long as they always did; what changed is that four other instructions are now inside the pipeline at the same time.

The second fault in C: latency does not fall at all, and in practice it rises a little. A latch between each pair of stages has to be written and read at every boundary crossing, and that overhead is charged to every instruction that passes through, so the end to end time grows rather than shrinks.

A test you can apply to any definition: ask whether the quantity is measured on one instruction or on a stream of them. Latency belongs to the instruction and throughput belongs to the stream, and a sentence that mixes the two is already wrong before you check its arithmetic.

Two of these three sentences would lose marks, and both fail in the same place, by measuring a stream of instructions and calling the answer latency.

The Master Key: *Latency is one instruction from end to end, and throughput is how many of them leave the pipeline each second.*

REAL Question 15

A five stage pipeline ought to be five times faster than a machine without one. Push four instructions through and it is two and a half times faster; push a hundred and it is not quite four and eight tenths. The missing amount never disappears.

Calculate the speed-up a five stage pipeline achieves over a non-pipelined machine for 4, 10 and 100 instructions, and explain why the speed-up approaches five without ever reaching it.

Model Answer

Speed-up is $kN / (N + k - 1)$, which for five stages comes to 2.5 on 4 instructions, 3.571 on 10 and 4.808 on 100, climbing towards five and never arriving.

The machine you are comparing against: a non-pipelined processor starts an instruction, finishes all five of its steps and only then starts the next, so N instructions cost N × k cycles. For four instructions that is 4 × 5, which is 20 cycles of work with no overlap anywhere in it.

The machine you are comparing: the pipelined version of the same processor completes those four instructions in N + k − 1 cycles, which is 4 + 4, so 8 cycles. Dividing the one time by the other gives 20 / 8, a speed-up of 2.5, and note that it is nowhere near the five stages the hardware has.

$$\text{speed} - \text{up} = \text{kN} / (\text{N} + \text{k} - 1)$$

Ten instructions, then a hundred: at ten the unpipelined machine needs 50 cycles and the pipelined one 14, giving 50 / 14, which is 3.571. At a hundred it is 500 cycles against 104, giving 4.808, and the gain has grown substantially without one wire of the hardware changing.

Why the ceiling is five: the numerator is 5N and the denominator is N plus a fixed 4. As N grows that 4 counts for less and less of the total, so the ratio tends towards 5, which is the number of stages and therefore the ideal speed-up.

Why five is never reached: the denominator always exceeds N by exactly k − 1, and those 4 cycles of fill and drain are paid whatever the length of the program. Figure 2.5 shows the curve still climbing at N = 30, where the speed-up is 4.41, close to the ideal line and still short of it.

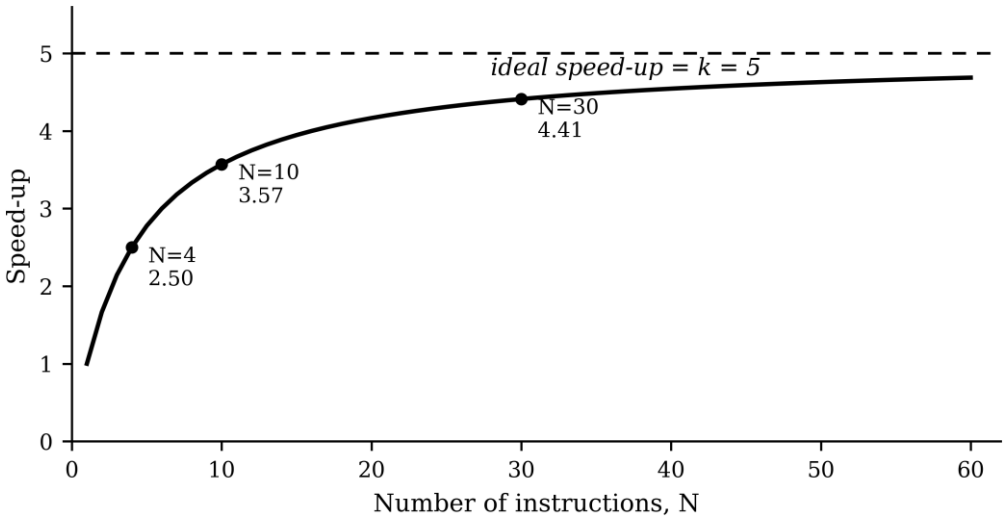


Figure 2.5: *Speed-up against the number of instructions for a five stage pipeline. The curve rises towards five but never reaches it.*

What this is worth in an answer: quote the limit and the reason for the shortfall together. A candidate who writes only that a five stage pipeline is five times faster has given the ideal and lost the fill and the drain, which is the part of the question actually being examined.

Ideal speed-up is the number of stages, and actual speed-up is that ideal discounted by the cycles spent filling and draining.

The Master Key: *The pipeline promises k and delivers kN over N + k − 1, and the whole of the difference is the fill and the drain.*

REAL Question 16

The mobile money kiosk beside the bus stand in Kyela serves one customer at a time, but every fifth has forgotten a PIN and holds the counter for two turns. The agent is never idle, and the queue still moves more slowly than her hands.

A pipeline runs a program in which 20 per cent of instructions cause a two cycle stall and 10 per cent cause a one cycle stall. Calculate the CPI, and calculate the throughput in instructions per second if the clock runs at 2 GHz.

Model Answer

The machine averages 1.5 cycles per instruction, and at 2 GHz that gives a throughput of 1.333 × 10^9 instructions per second.

Turn each proportion into cycles: one instruction in five, which is 0.20 of them, costs 2 extra cycles, contributing 0.20 × 2, which is 0.4 cycles per instruction on average. One in ten costs 1 extra cycle, contributing 0.10 × 1, which is 0.1 cycles per instruction. A percentage is of no use until it has been turned into cycles in this way.

Add them for the stall rate: 0.4 plus 0.1 gives 0.5 stall cycles per instruction. Seven instructions in every ten stall for nothing at all, but the waiting is spread evenly across the whole program to make an average, which is what allows it to be used in a single multiplication.

$$\text{CPI} = 1 + \text{stalls per instruction} = 1 + 0.5 = 1.5$$

What 1.5 means in words: the machine spends one cycle working on an instruction and half a cycle waiting, for every instruction it completes. Said without the fraction, it consumes 3 cycles for every 2 instructions it finishes, and the extra cycle buys nothing.

Throughput from the CPI and the clock: the clock supplies 2 000 000 000 cycles every second and each instruction consumes 1.5 of them, so the machine completes 2 000 000 000 divided by 1.5 instructions per second. That is 1.333×10^9, or about one and a third thousand million instructions a second.

What the stalls actually cost: the same clock with no stalls at all would run at a CPI of 1 and finish 2×10^9 instructions per second, so the waiting has cost 0.667×10^9 instructions every second. That is a third of the machine thrown away, and no increase in clock frequency recovers any of it.

The check that catches a slip: IPC here is 1 divided by 1.5, which is 0.667, comfortably below 1 as it has to be on a single-issue pipeline. If a CPI on a machine of this kind ever comes out below 1, it is the arithmetic that is wrong and not the processor.

Stalls per instruction, then CPI, then frequency divided by CPI; that order never changes, whatever the question dresses the stalls up as.

The Master Key: *Throughput is the clock frequency divided by the CPI, so a stall costs you instructions per second and never cycles per second.*

REAL Question 17

At a maize mill in Chemba the miller keeps four tallies: sacks ground, sacks weighed, sacks bagged, and the few put through again. Regrinding is one line in five, yet it eats more of the working day than any other line in the book.

A processor executes a program of 40 per cent arithmetic instructions, 20 per cent loads, 20 per cent stores and 20 per cent branches. Loads and stores each require one extra cycle in the memory stage, and branches cost a two cycle penalty. The clock period is 1 nanosecond. Calculate the CPI and the throughput.

Arithmetic instructions: 40 per cent of the program, no extra cycles

Loads: 20 per cent of the program, one extra cycle each in the MEM stage

Stores: 20 per cent of the program, one extra cycle each in the MEM stage

Branches: 20 per cent of the program, a two cycle penalty each

Clock period: 1 nanosecond

Model Answer

The processor averages 1.8 cycles per instruction, and with a clock period of 1 nanosecond that gives a throughput of about 5.56×10^8 instructions per second.

The method in one line: each class contributes its share of the program multiplied by the extra cycles it costs, and those contributions are added to the base of one cycle per instruction. Arithmetic instructions need no extra cycle at all, so the largest class in the program contributes nothing whatever to the excess.

The three contributions: loads give 0.20 × 1, which is 0.20, stores give 0.20 × 1, which is another 0.20, and branches give 0.20 × 2, which is 0.40. Added together the extra cycles come to 0.8 per instruction, and adding the base of 1 gives the answer.

$$\text{CPI} = 1 + 0.20 \times 1 + 0.20 \times 1 + 0.20 \times 2 = 1.8$$

From cycle time to throughput: a clock period of 1 nanosecond means the machine is given 1 000 000 000 cycles in every second, and each instruction takes 1.8 of them, so it completes 1 000 000 000 divided by 1.8 instructions per second. That is 5.56×10^8, a little over half a thousand million.

Instruction class	Share	Extra cycles	Contribution
Arithmetic	0.40	0	0.00
Loads	0.20	1	0.20
Stores	0.20	1	0.20
Branches	0.20	2	0.40
All classes	1.00		0.80

Table 2.1: *Each class weighted by its share. The contribution column is the share multiplied by the extra cycles, and the four contributions add to 0.80 above the base of one.*

What the table shows: branches are one instruction in five, the same share as loads and the same as stores, and yet they contribute 0.40 of the 0.80 excess, which is exactly half of it. A penalty of two cycles instead of one doubles the damage done by an identical share of the program.

What the CPI means in real time: each instruction consumes 1.8 cycles of 1 nanosecond, so one instruction is finished every 1.8 nanoseconds on average. The figure is an average and not a timetable: an arithmetic instruction still leaves after its own single cycle, while a branch occupies 3.

Where that sends the design effort: removing the branch penalty is worth as much as removing the load and the store penalties together. That single line of arithmetic is why so much hardware is spent on guessing where a branch will go before the answer is known.

Weight every class by its share before you add anything, because a small share carrying a large penalty can outweigh a large share carrying none.

The Master Key: *CPI is a weighted average, so the class that costs the most per instruction is rarely the class that appears most often.*

HOT Question 18

The procurement officer at a district hospital in Mufindi has two quotations and one program that matters, the patient register. One machine ticks half as fast again as the other, and the faster is dearer. Neither quotation says which will finish the register first.

Machine P runs at 3 GHz with a CPI of 2.0. Machine Q runs at 2 GHz with a CPI of 1.2. Both execute the same program of 10^9 instructions. Determine which machine finishes first, and discuss what the comparison shows about judging processors by clock speed.

Machine P: clock frequency 3 GHz, CPI 2.0

Machine Q: clock frequency 2 GHz, CPI 1.2

Both machines execute the same program of 1 000 000 000 instructions

Model Answer

Machine Q finishes first, in 0.600 seconds against machine P at 0.667 seconds, and it wins while running a clock a third slower than P's.

The only equation that settles it: execution time is the instruction count multiplied by the cycles each instruction needs, divided by the cycles the clock supplies each second. Three quantities go in, and dropping any one of them makes the comparison worthless rather than merely rough.

$$\text{time} = \text{instructions} \times \text{CPI} / \text{frequency}$$

What machine P costs: a thousand million instructions at 2.0 cycles each comes to 2 000 000 000 cycles, and the clock supplies 3 000 000 000 cycles per second, so P finishes the register in 0.667 seconds.

What machine Q costs: the same thousand million instructions at 1.2 cycles each comes to only 1 200 000 000 cycles, and the clock supplies 2 000 000 000 per second, so Q finishes in 0.600 seconds. Q is ahead by about 10 per cent on a clock that is slower by a third.

The case for quoting clock speed: it is not a worthless figure. Between two machines of the same design running the same instruction set, raising the frequency raises performance very nearly in proportion, and frequency is the one specification a buyer can compare without running a single program.

Why it fails on this pair: P and Q are not the same design. P needs twice the cycles an ideal pipeline would need for each instruction while Q needs only a fifth more, so P gives back at the pipeline everything it gained at the clock, and its extra gigahertz buys the hospital nothing at all.

The judgement: judge a processor on the product of the three quantities and never on one of them. Instruction count, CPI and clock frequency multiply together to give time, and a specification that quotes only the clock is quoting the least informative of the three, because either of the other two can move the answer further than it can.

The condition that would reverse it: P wins if its instruction set does more per instruction. Break-even is 900 000 000 instructions, since 900 000 000 at 2.0 cycles each is 1 800 000 000 cycles and 0.600 seconds at 3 GHz, so if P executes the same task in fewer than 0.9 of Q's instructions, P takes the register.

On this program the verdict is Q, by a tenth, and only a shorter instruction stream on P would overturn it.

The Master Key: *Time is instructions times CPI divided by frequency, and a comparison quoting fewer than all three is quoting the wrong one.*

Hazards, and the structural hazard

This section opens the hazard material for the chapter. It defines what a pipeline hazard is, names the three classes and gives an example of each, then takes the structural hazard as far as it will go: where it comes from, why a single memory sets the fetch stage and the memory stage against each other in every cycle once the pipeline is full, and what each remedy costs. It draws the collision and it draws the split cache that cures it, and it puts a number on the cure by costing a program of 500 instructions on a machine with one memory. Sections 4 and 5 take the other two classes and will assume the taxonomy set out here.

BINDER Question 19

The rice huller at Ulanga makes one steady sound when fed properly, a note the operator can hear from the road. When the feed falters the note drops for a beat and then recovers, and every dropped beat is rice that did not get hulled.

Define a pipeline hazard and name the three classes of hazard, giving one example of each.

Model Answer

A pipeline hazard is any condition that prevents the next instruction from entering its stage in the cycle it was due, so the pipeline can no longer sustain one completion per cycle.

Structural hazard: two instructions in different stages need the same piece of hardware in the same cycle, and that hardware can serve only one of them. The standard example is a single memory that must hand an instruction to the fetch stage and a data word to the memory stage at the same moment.

Data hazard: an instruction needs a result that an earlier instruction, still travelling through the pipeline, has not yet produced. An addition that reads a register the immediately preceding addition is still computing cannot proceed until that value exists.

Control hazard: the address of the next instruction is not yet known, because a branch has not resolved. The fetch stage has nothing certain to fetch, so it either waits or fetches on a guess that may have to be thrown away.

One currency, three remedies: all three classes are paid for in the same coin, wasted cycles, and every wasted cycle lifts CPI above one. What differs is the cure: a structural hazard is cured by more hardware, a data hazard by carrying the value backwards early, and a control hazard by predicting the branch, and Sections 4 and 5 take those last two in turn.

Name the class before you reach for a remedy, because the wrong name leads straight to the wrong cure.

The Master Key: *A hazard is a stage denied its cycle; structural means shared hardware, data means a missing value, control means an unknown address.*

BINDER Question 20

One adjustable spanner hangs on the workshop wall behind the bus stand at Makambako, and at ten o'clock two mechanics reach for it. Neither is waiting on the other's work and neither has done anything wrong; they want the same steel at the same moment.

Define a structural hazard and state two circumstances in which one arises.

Model Answer

A structural hazard arises when two instructions in different stages of the pipeline require the same hardware resource in the same clock cycle, and the hardware cannot serve both.

One memory, two hungry stages: the first circumstance. The fetch stage reads an instruction from memory in every cycle, and the memory stage reads or writes data memory for every load and every store. Where there is a single memory, or a single port onto it, the two demands arrive together and one of them must be refused.

One functional unit, two claimants: the second circumstance. A machine with one floating point divider, or a register file with a single write port, cannot serve two instructions that reach it in the same cycle. The later claimant is held back until the unit falls free.

A property of the hardware, not of the program: the same program on a machine with a second memory port, or a second divider, meets no such hazard at all. Not one instruction changed; the resource count did. That is why a structural hazard can be designed away, which is never true of a genuine dependence between values.

The confusion that costs marks: students are often taught, wrongly, that two instructions fighting over a port are data dependent. They are not. A data dependence means one instruction needs a value another produces, and two instructions contending for a port may share no data whatever, so the correct name is always structural.

Count the resources before you blame the program, because a structural hazard is a shortage of hardware and nothing else.

The Master Key: *Same cycle, same resource, one copy of it: that is a structural hazard, and it belongs to the machine rather than to the code.*

BINDER Question 21

Five stages, IF, ID, EX, MEM and WB, and one memory chip on the board of the teaching computer in the laboratory at Kondoa. For the first three cycles nothing goes wrong. In the fourth it asks that single chip for two things at once.

Explain, with reference to a five stage pipeline that has a single memory, when and why the fetch stage and the memory stage come into conflict.

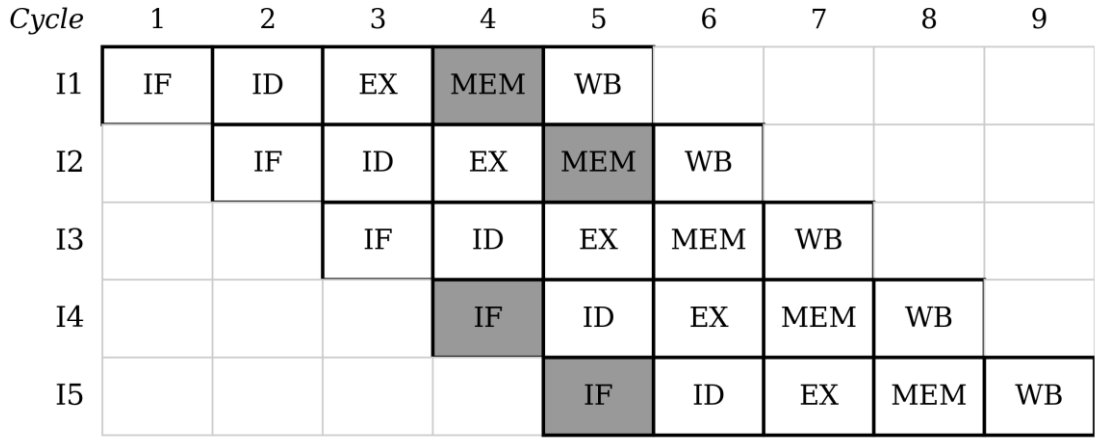
Model Answer

The conflict begins in cycle 4, the first cycle in which one instruction sits in MEM while another sits in IF, and it then repeats in every cycle for as long as the pipeline stays full.

What each stage wants: IF reads an instruction from memory in every single cycle, without exception, because every instruction has to be fetched before anything else can happen to it. MEM reads or writes data memory, but only for loads and stores. On a machine with one memory both requests go to the same place.

The first collision: the first instruction is fetched in cycle 1, decoded in cycle 2, executed in cycle 3 and reaches MEM in cycle 4. In that same cycle 4 the fourth instruction arrives at IF, because a new instruction enters the pipeline every cycle. One memory, two demands, and only one of them can be served.

Why this is the steady state: the clash is not an accident of a single cycle. In cycle 5 the second instruction is in MEM while the fifth is in IF, and the pattern recurs in every cycle after that. Figure 2.6 shades the contending cells in cycles 4 and 5.



One memory cannot serve IF and MEM in the same cycle. Shaded cells contend.

Figure 2.6: *A single memory serving both the fetch stage and the memory stage. Once the pipeline is full the two collide in every cycle.*

What the conflict costs: the hardware must hold the fetch back whenever the memory stage is using the memory, and a bubble enters the pipeline behind it. Were every instruction a load or a store, such a machine could never sustain one completion per cycle at all.

Once the pipeline is full, a single memory is asked for two things in every cycle in which a load or a store is in flight.

The Master Key: *With one memory, IF and MEM collide from cycle 4 onwards and keep colliding; the clash is the normal case, not the exception.*

BINDER Question 22

A second guillotine, or a longer queue: the printing shop at Nachingwea must choose one and cannot have both. The machine standing idle half the day is money that bought nothing, and the queue is money that leaks out an hour at a time.

Describe two ways of removing a structural hazard, and state the cost of each.

Model Answer

A structural hazard is removed by duplicating the contended resource, and the sharpest form of that duplication is splitting one memory into a separate instruction cache and data cache; stalling is a third option, which does not remove the hazard but merely pays for it as it occurs.

Duplicate the resource: add a second memory port, a second adder, a second write port on the register file. The hazard then cannot occur, because both stages are served in the same cycle and neither waits. The cost is silicon area, power and design time, and the duplicate earns nothing at all in any cycle when the conflict would not have arisen.

Split the memory: give the fetch stage an instruction cache and the memory stage a data cache, each with its own port, as Figure 2.7 shows. This is duplication aimed squarely at the one collision that happens in almost every cycle, which is why nearly every real processor is built with separate first level caches.

two ports, so both stages proceed in the same cycle

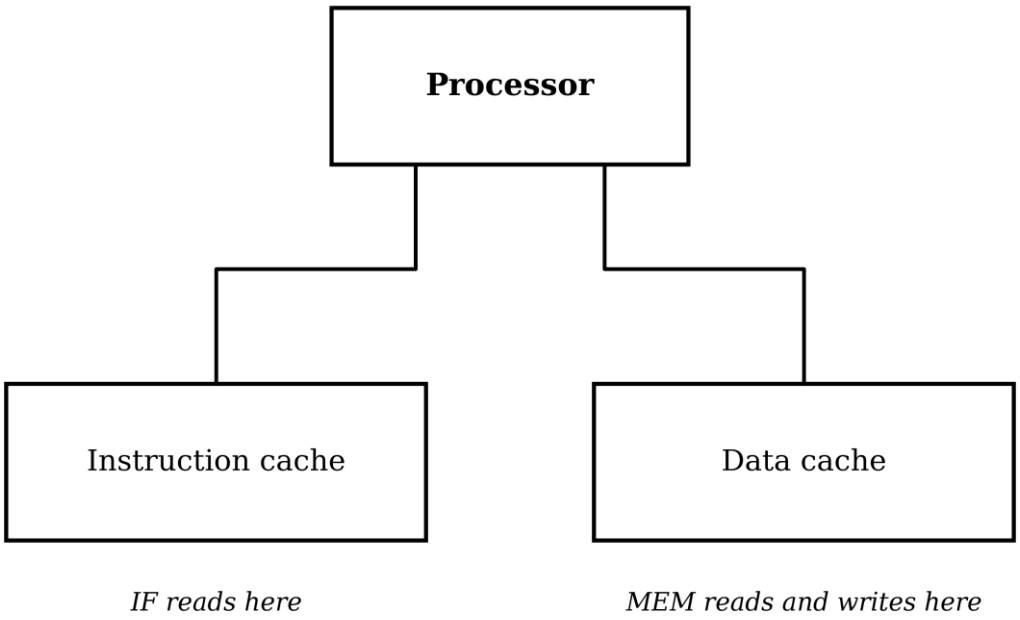


Figure 2.7: *Splitting the memory into an instruction cache and a data cache. Two ports, so the fetch stage and the memory stage no longer compete.*

Stall instead: hold the fetch for one cycle whenever the memory stage needs the memory, and let a bubble travel down the pipeline. This costs nothing to build and one cycle every time the conflict occurs, so the total bill is set entirely by how often loads and stores appear in the program.

Which to choose: duplicate when the conflict is frequent and stall when it is rare. Memory is touched in every cycle, so a second port repays its area at once; a unit contended a handful of times in a thousand instructions is not worth the silicon a spare would take.

The split cache is the standard answer because the memory conflict is the one structural hazard that never goes away on its own.

The Master Key: *Duplicate the resource to remove a structural hazard and stall to survive one; the split instruction and data cache is duplication placed where it pays best.*

BINDER Question 23

Watch the loading line at the maize mill in Kaliua. One man pauses to retie a sack, and the empty space he leaves does not stay with him; it travels down to the lorry. Nobody else stopped, yet every station after him stood idle.

Explain what a stall is, state what a bubble is, and describe how each appears on a space-time diagram.

Model Answer

A stall is the act of holding an instruction, and everything behind it, in place for one or more cycles; the bubble is the empty slot this creates, which then travels forward through the later stages doing no useful work.

The stall is the action: the control logic freezes the pipeline registers at the blocked stage and everywhere upstream of it, so those instructions hold their position instead of advancing. Nothing behind a stall can overtake it, because an in-order pipeline never allows a later instruction to pass an earlier one.

The bubble is the consequence: downstream of the stall the stages have nothing to receive, so an empty slot is inserted and moves on one stage each cycle until it leaves at WB. The bubble is a real slot in the schedule: the hardware is clocked and powered for it and produces nothing whatever.

How each looks on the chart: the stalled instruction shows the same stage twice in consecutive columns, or an empty cell just before the delayed stage, and every row below it shifts one column to the right. The emptiness then runs diagonally down and to the right, one cell per cycle, exactly as a working instruction would.

The arithmetic of a bubble: every bubble adds one cycle to the total, so the average cycles per instruction rises above one by exactly the number of stall cycles per instruction. A pipeline that never stalls has a CPI of one; one stall in every four instructions gives a CPI of 1.25.

$$\text{CPI} = 1 + \text{stalls per instruction}$$

Count bubbles rather than stalls when costing a program, because a single stall of two cycles puts two bubbles into the schedule.

The Master Key: *A stall holds instructions still, the bubble is the hole it leaves behind, and every hole costs exactly one cycle.*

REAL Question 24

A poster in the staffroom of a school in Igunga lists three statements about hazards, and the technician who read it ticked only one. The teacher who wrote all three had been sure that every one of them was sound.

Three statements about hazards are given below. Identify the one that is correct, and explain precisely what is wrong with each of the other two.

Statement A: A structural hazard is caused by a limitation of the hardware, and the same program would run without it on a machine with more resources.

Statement B: Two instructions competing for the same memory port is an example of a data dependence.

Statement C: Structural hazards can only be removed by stalling the pipeline.

Model Answer

Statement A is the correct one; Statement B gives the hazard the wrong name, and Statement C treats one remedy as the only remedy when it is merely the cheapest.

Why A is right: a structural hazard exists because the machine holds too few copies of some resource. Give that same machine a second memory port, or split its memory into two caches, and the identical program runs through without ever meeting the hazard. The fault therefore lies in the hardware, exactly as A states.

What B confuses: a data dependence exists when one instruction needs a value that another instruction produces. Two instructions competing for a memory port need share no data at all: the first may be storing a customer record while the second fetches an entirely unrelated instruction. Their only relationship is wanting one port in one cycle.

The correct name for B: contention for a memory port is a structural hazard, and one test settles it. Ask whether adding hardware would make the problem vanish. If it would, the hazard is structural. If the instruction must still wait because the value it needs does not yet exist, the hazard is a data hazard.

Why C is false: duplicating the resource removes the hazard outright rather than merely surviving it. B separate instruction cache and data cache is precisely that remedy applied to the commonest case of all, and once those two caches are in place the fetch stage and the memory stage never contend again.

What stalling actually is: stalling is the fallback that costs no silicon and one cycle for each conflict. It is the right choice for a resource contended rarely and the wrong choice for one contended every cycle, but it is never the only choice on offer.

Both failures come from the same root, which is forgetting that a structural hazard is a shortage of hardware and can therefore be bought away.

The Master Key: *Structural means the machine is short of a resource, so hardware cures it; only a data dependence is about values, and only there is the waiting unavoidable.*

REAL Question 25

The bursar at a secondary school in Mbinga asked a fair question. If the machine stalls on loads and stores, under a third of the program, the loss must be small. The technician worked the sum on a receipt: nearly a quarter of the machine.

A five stage pipeline has one memory, so every load and every store forces a one cycle stall. In a program of 500 instructions, 30 per cent are loads or stores. Calculate the total cycles, the CPI and the percentage of throughput lost against an ideal pipeline.

Model Answer

The program takes 654 cycles where an ideal pipeline would take 504, which gives a CPI of 1.308 and a throughput loss of about 22.9 per cent.

Count the stalling instructions: 30 per cent of the 500 instructions are loads or stores, and 0.30 multiplied by 500 gives 150 of them. Every one of those 150 reaches MEM while another instruction is being fetched, and each therefore forces exactly one stall, so the program suffers 150 stall cycles in total. The other 350 instructions never touch data memory and pass through untroubled.

The ideal pipeline: with no hazards at all, N instructions through a k stage pipeline take N plus k minus 1 cycles. Here that is 500 plus 5 minus 1, which is 504 cycles, and every later result in the question is measured against that one figure.

The actual run: add the stall cycles to the ideal count, so 504 plus 150 gives 654 cycles. Nothing else in the calculation moves, because a stall adds cycles to the schedule without adding any instructions to the program, and the instruction count is what the CPI is divided by.

Cycles per instruction: divide the actual cycles by the instruction count, so 654 divided by 500 is 1.308. That sits above the ideal of 1 by 0.308, which is 150 stalls spread across 500 instructions, or 0.3 stall cycles per instruction.

$$\text{CPI} = 654 / 500 = 1.308$$

Check it against the stall rule: an in-order pipeline has a CPI of 1 plus its stalls per instruction, which here is 1 plus 0.3, giving 1.3. The measured 1.308 is a shade higher because the 4 cycles spent filling and draining the pipeline are spread over the 500 instructions too, adding 0.008 to the average.

Throughput lost: throughput moves inversely with cycles, so achieved throughput as a fraction of the ideal is 504 divided by 654, which is 0.7706. The loss is 1 minus that, which is 0.2294, or about 22.9 per cent of the machine thrown away on a single shared memory.

Quantity	Value
Ideal cycles (500 + 5 - 1)	504
Stall cycles (0.30 x 500)	150
Actual cycles (504 + 150)	654
CPI (654 / 500)	1.308

Table 2.2: *The cost of one shared memory in a program of 500 instructions.*

What a designer should conclude: splitting the memory into an instruction cache and a data cache removes all 150 stalls and buys back nearly a quarter of the throughput. A gain that size is why the duplication is worth its silicon on any machine meant to run ordinary programs.

A hazard that fires on fewer than a third of the instructions still costs almost a quarter of the throughput, and that is the number that justifies the extra hardware.

The Master Key: *Stall cycles add straight onto N plus k minus 1, so 150 stalls over 500 instructions turn 504 cycles into 654 and a CPI of 1.308.*

HOT Question 26

Two technicians at the tea factory in Mufindi argued for a week about a second weighbridge. One held that no lorry should ever wait; the other that the queue forms twice a month and clears itself by noon. Both were right about their own facts.

Assess the claim that a structural hazard should always be removed by duplicating the hardware rather than by stalling.

Model Answer

The claim is sound for resources the machine uses in almost every cycle and false for resources contended rarely, so the word always is exactly what breaks it.

The case for duplicating: duplication removes the hazard instead of managing it. With two ports the pipeline goes on delivering one completion per cycle, CPI stays at one, and no detection logic is needed to spot the conflict and freeze the stages behind it.

Memory settles that case at once: the fetch stage reads memory in every cycle and the memory stage touches it on every load and every store, so contention there is continuous rather than occasional. Duplication in the form of a split instruction cache and data cache is not a luxury; without it the machine stalls on a large fraction of everything it runs.

The case against duplicating: every duplicate costs area on the die, power whether it is used or not, and engineering time that could have been spent elsewhere. A second copy that stands idle in 999 cycles out of 1000 has bought a thousandth of a cycle per instruction, and it has bought it at the full price.

Some units are simply too expensive: a floating point divider is large and slow, and a second one can cost more area than the whole of the rest of the execution hardware. A machine that divides rarely loses almost nothing by stalling on the second division, and it would lose a great deal by carrying a spare divider it seldom wakes.

The condition that decides it: duplicate when the rate at which the conflict occurs, multiplied by the cycles each conflict costs, exceeds the cost of the extra hardware measured in the same currency. Put in words, the stall cycles saved per instruction must be worth more than the area and the power that the duplicate consumes.

The verdict: the claim fails as stated. Duplicate the memory ports, because contention there is continuous and the fix is small and well understood; stall on rarely used functional units, because the saving is a fraction of a cycle while the hardware is not a fraction of anything. Real processors do exactly this.

Always is the wrong word: duplicate where contention is continuous and stall where it is rare.

The Master Key: *Duplicate when contention rate times stall cost exceeds the price of the hardware, which means split caches yes and a second divider no.*

HOT Question 27

One line item out of three came back approved. The team costing a small processor for a water metering project in Simanjiro wanted a split cache, a second arithmetic unit and three extra pipeline stages. Two of the three would have bought them almost nothing.

A design team has budget for exactly one of three changes: splitting the single memory into an instruction cache and a data cache, adding a second arithmetic unit, or deepening the pipeline from five stages to eight. Recommend one, and justify your recommendation against the other two.

Option A: Split the single memory into a separate instruction cache and data cache.

Option B: Add a second arithmetic unit.

Option C: Deepen the pipeline from five stages to eight.

Model Answer

Take Option A and split the memory, because the collision between fetch and memory access is the most frequent hazard the machine has and it is the only one of the three changes that removes a hazard this design actually suffers.

Why the split cache wins: the fetch stage needs memory in every cycle and the memory stage needs it on every load and every store, so the two collide in every cycle in which a load or a store is in flight. Nothing else in a single-issue five stage pipeline occurs anything like as often, and the remedy is small, well understood and cheap.

Against a second arithmetic unit: a second unit helps only when two arithmetic instructions are ready to execute in the same cycle. A single-issue pipeline starts exactly one instruction per cycle, so two can never be ready together, and the extra unit would stand idle in every cycle of every program. The gain is zero until the machine is made superscalar.

Against a deeper pipeline: moving from five stages to eight raises the fill and drain penalty from 4 cycles to 7 and lengthens the branch penalty in the same proportion. It also does nothing whatever about the shared memory, so the machine would still stall on every load and every store, and now with a longer pipeline to refill afterwards.

The verdict in one line: the split cache removes a hazard that fires constantly, the second arithmetic unit removes a hazard this machine cannot experience, and the deeper pipeline adds penalties while removing nothing. Only one of the three reduces the number of stall cycles the program suffers.

What would change the recommendation: were the machine already multiple-issue and already fitted with split first level caches, the second arithmetic unit would move to the top of the list, because two arithmetic instructions could then genuinely be ready in one cycle and a single unit would become the bottleneck. Greater depth would still come last.

Spend the budget where the hazard actually fires, and on this machine that is the single memory.

The Master Key: *Fix the hazard that occurs every cycle before buying parallelism the machine cannot use or depth that only raises the penalties.*

Data hazards, forwarding and the load-use stall

This section takes the data hazard from its definition through to a design decision. It settles which of the three data dependences a strictly in-order pipeline can actually produce, then measures the two remedies against each other: stalling, which costs two cycles on an arithmetic pair, and forwarding, which costs none but has to be paid for in hardware. It draws the stall, it draws the bypass, and it draws the single bubble that no bypass network can remove, before putting a CPI on a machine with forwarding and on the same machine without, which is the number that buys the wires.

BINDER Question 28

A dispensary clerk at Mbulu copied a patient total onto the return before the register beneath it was added up, and it went to the district office. His arithmetic was sound and so was the register; the figure he read did not yet exist.

Define a data hazard and explain the circumstances in which one arises in a five stage pipeline.

Model Answer

A data hazard arises when an instruction needs an operand that an earlier instruction, still travelling through the pipeline, has not yet made available.

Where the value is made available: the producing instruction writes its result into the register file in WB, the fifth cycle of its own passage. Until that write happens the register file still holds the old contents, and the new value exists only inside the pipeline latches.

Where the value is read: the consuming instruction reads its operands in ID, the second cycle of its own passage. Reading therefore happens three stages earlier in a passage than writing does, and a consumer that follows too closely reaches the register file before the producer has got there.

A concrete pair: an add that writes register r1, followed immediately by a subtract that reads r1. The add is fetched in cycle 1 and writes in cycle 5. The subtract is fetched in cycle 2 and would read in cycle 3, two cycles too early, taking the stale value that r1 held before the add ever ran.

The general rule: the hazard exists whenever the distance between producer and consumer is smaller than the distance between the producing stage and the consuming stage. Here WB stands three stages behind ID, so any consumer within three instructions of its producer is exposed.

Not a shortage of hardware: a second arithmetic unit changes nothing here. The consumer is not waiting for a machine to become free, it is waiting for a value to come into existence, and duplicated hardware cannot make a value appear sooner.

Every data hazard in this pipeline reduces to one arithmetic fact, that WB comes three stages after ID, so a value can be wanted before it has been written.

The Master Key: *A data hazard is a value wanted before it has been written, and in the five stage pipeline it bites whenever producer and consumer are within three instructions of each other.*

BINDER Question 29

Wherever this topic is taught, three names are given and two forgotten. A study group at Chunya spent an evening hunting through a five stage pipeline diagram for a write after write, found none, and decided it must be faulty. The diagram was correct.

Name the three kinds of data dependence, state which of them is a true dependence, and explain which can occur in an in-order five stage pipeline.

Model Answer

The three kinds are read after write, write after read and write after write; read after write is the true dependence, and it is the only one of the three that can arise in an in-order five stage pipeline.

Read after write, the true dependence: the consumer reads a register that an earlier instruction writes, and it genuinely needs that value to compute the right answer. No hardware can remove the requirement itself; forwarding and stalling only hide the delay. That is what the word true is doing in the name.

Write after read and write after write, the name dependences: in a write after read, a later instruction writes a register that an earlier one has still to read. In a write after write, two instructions write the same register and the later write must survive. Neither passes a value between the pair; they share only a register name, so giving one of them a different register removes the dependence outright.

Why only the true dependence can arise here: in an in-order single-issue five stage pipeline every instruction reads in ID and writes in WB, both in program order. An earlier instruction therefore always reads before a later one writes, and writes always land in order, so neither name dependence is possible. They become live only once instructions may execute out of order, which Section 6 takes up.

A common source of confusion: being asked to research write after read and write after write while studying a strictly in-order pipeline confuses many candidates; learn the two names now and meet the hazards later. Every stall counted in this chapter comes from a read after write.

Only one of the three costs this machine a cycle, and it is the one in which a value genuinely has to travel from a producer to a consumer.

The Master Key: *Read after write is a value, write after read and write after write are only shared names, and an in-order pipeline cannot produce either name dependence.*

BINDER Question 30

Stop the line and everyone behind you stops with you. That is the whole of stalling, and the only question is how long it lasts. At the lorry park in Ludewa the answer is however long the loader takes; in a pipeline, an exact number.

Explain how stalling resolves a data hazard, and calculate how many cycles are lost when an arithmetic instruction is immediately followed by an instruction that uses its result.

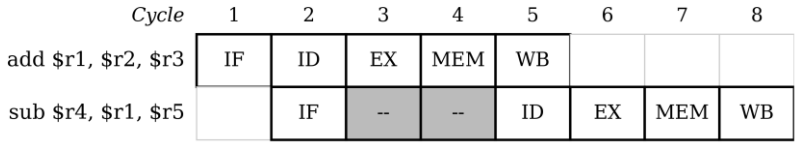
Model Answer

Stalling resolves a data hazard by holding the dependent instruction, and everything behind it, until the operand is available, and for an arithmetic result used by the very next instruction the cost is two cycles.

What the hardware actually does: the control logic compares the registers the instruction in ID wants to read against the destinations of the instructions ahead of it. On a match it freezes IF and ID, and pushes a bubble, an instruction that does nothing, into EX for each cycle of the wait.

State the assumption before you count: the register file writes in the first half of a cycle and reads in the second half. Say so explicitly, because on that assumption the answer is two cycles and without it the honest answer is three; the number two with no assumption stated cannot be reproduced.

The count: the add is fetched in cycle 1 and writes r1 in WB in cycle 5. The subtract is fetched in cycle 2 and would read r1 in ID in cycle 3. Because the write lands in the first half of cycle 5, the subtract may read in the second half of that same cycle.



Two bubbles, because the register file writes in the first half of a cycle and reads in the second.

Figure 2.8: *A data hazard resolved by stalling. The dependent instruction waits two cycles before it may read the register.*

The cycles lost: the subtract moves its ID from cycle 3 to cycle 5 and waits two cycles, the two grey bubbles in Figure 2.8. Everything behind it is held for the same two cycles, but the delay is charged once: the pair costs two cycles in total, not four.

What it costs a program: two cycles for every such pair. If one instruction in five consumes the result of the one before it, stalls per instruction is 0.2 multiplied by 2, which is 0.4, and the CPI rises to 1.4 on a machine sold as one cycle per instruction.

Two cycles is honest only because the register file is written and read within one cycle, so state that assumption first, then count.

The Master Key: *Stalling holds the consumer until the value exists, and with a half cycle register file an arithmetic result costs the next instruction two cycles, three without it.*

BINDER Question 31

At the fish landing on the lake at Bunda the ice is not carried into the store and fetched back out; the man packing the box takes it straight from the man breaking it. The saving is not in the ice but in the walking.

Explain what forwarding is, name the two bypass paths in a five stage pipeline, and state what forwarding costs.

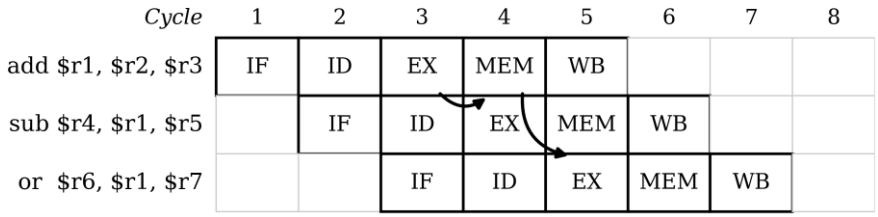
Model Answer

Forwarding, also called operand bypassing, takes a result from the pipeline register that already holds it and feeds it straight into the arithmetic unit that needs it, rather than waiting for it to reach the register file.

The idea: the value is finished at the end of EX, two cycles before WB writes it down. The register file is where a value is kept, not where it is made, so forwarding routes it from the latch straight to the input of the unit that wants it.

EX/MEM to EX: the first path serves a consumer one instruction behind its producer. The result sits in the EX/MEM latch at the end of the producer's EX, and the consumer needs it at the start of its own EX in the next cycle, cycle 3 to cycle 4 in Figure 2.9.

MEM/WB to EX: the second path serves a consumer two instructions behind. The result has moved on into the MEM/WB latch and is taken from there to the arithmetic unit, cycle 4 to cycle 5 in the figure. A consumer three behind needs no path, because the half cycle register file delivers in time.



No bubbles: the result is taken from the latch that holds it, not from the register file.

- *EX/MEM to EX bypass: cycle 3 to cycle 4.*
- *MEM/WB to EX bypass: cycle 4 to cycle 5.*

Figure 2.9: *The same dependence resolved by forwarding. The result is taken from the latch that already holds it, so no cycle is lost.*

What forwarding costs: comparators in every stage, to test the destination of each instruction in flight against the sources of the instruction entering EX; multiplexers on both inputs of the arithmetic unit, to select a bypassed

value instead of the register file output; and the wires, which lengthen the critical path and can force a slower clock.

Why stalling survives: forwarding cannot cover every case, so real processors use forwarding and stalling together. Where the value does not yet exist when the bypass would have to run, the hardware must still stall, and the next question works that case.

Forwarding buys back the cycles that stalling would have lost, and pays for them in comparators, multiplexers and critical path.

The Master Key: *Forwarding takes the value from the latch, never from the register file: EX/MEM to EX for the next instruction, MEM/WB to EX for the one after.*

REAL Question 32

One cycle. That is the load-use penalty, and it survives every wire a designer adds. A technician at Hai, his teaching board fitted with full forwarding, spent an afternoon hunting the fault behind one grey square on his timing diagram. There was no fault.

A load instruction is immediately followed by an instruction that uses the loaded value. Explain why full forwarding cannot remove the stall, and state how many cycles are lost.

Model Answer

Full forwarding cannot remove this stall because the loaded value does not exist until the end of the memory stage, while the consumer needs it at the start of an execute stage that would otherwise run in that very cycle, so exactly one cycle is lost.

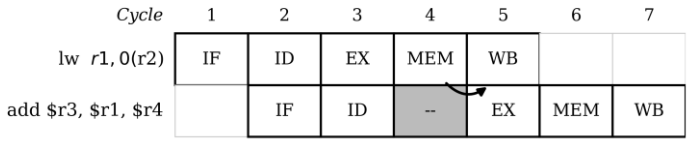
When the value appears: the load is fetched in cycle 1, so its MEM is cycle 4, and the data memory does not hand the value back until the end of cycle 4. Before that instant there is nothing anywhere in the machine to forward: not in a latch, not on a wire, not in the register file.

When the value is wanted: the dependent add is fetched in cycle 2, so unstalled its EX would also be cycle 4, and it needs both of its operands at the start of that cycle. A value that does not exist until the end of cycle 4 cannot possibly be an input to cycle 4.

Why more wires cannot help: forwarding moves a value across space, from a latch to an input, inside one cycle. It cannot move a value backwards through time. This single case separates the candidate who has understood forwarding from the one who has memorised the word.

$$\text{EX of the consumer} = \text{MEM of the load} + 1 = 4 + 1 = 5$$

The one bubble that survives: the hardware therefore holds the add for one cycle, its EX moves from cycle 4 to cycle 5, and the MEM/WB path then delivers the value into that cycle. Figure 2.10 shows the single grey bubble left behind and the bypass arrow that stops there being a second one.



One bubble survives full forwarding. This is the case the claim that forwarding removes every stall gets wrong.

● *MEM/WB to EX bypass: cycle 4 to cycle 5.*

Figure 2.10: *The load-use hazard. Full forwarding cannot remove this bubble, because the value does not exist until the end of the memory stage.*

What the compiler does about it: a compiler that knows the pipeline tries to place an unrelated instruction immediately after every load, so that the bubble is filled with useful work at no hardware cost whatever. Where no independent instruction can be found, the bubble stands and the cycle is simply lost.

Why one cycle matters: loads are commonly a fifth of the instructions a program executes, and many of them are followed at once by the instruction that consumes the value, so this is no curiosity at the edge of the design. Each such pair puts a whole cycle straight into the CPI.

One cycle, every time, and it is the work of the compiler rather than of the bypass network to fill it.

The Master Key: *A loaded value arrives at the end of MEM, so the dependent instruction cannot begin EX before the next cycle, because forwarding crosses space but never time.*

REAL Question 33

Somebody had typed the page neatly, and nobody in the Form Six class at Newala questioned its three sentences about forwarding. Two are wrong, and one would cost a designer money: a machine costed on it is quoted faster than it can possibly run.

Three statements about forwarding are given below. Identify the one that is correct, and explain precisely what is wrong with each of the other two.

Statement A: Forwarding eliminates the stall caused by every data hazard.

Statement B: Forwarding removes the stall for an arithmetic result, but a load followed immediately by a dependent instruction still costs one cycle.

Statement C: Forwarding works by writing the result into the register file earlier than usual.

Model Answer

Statement B is the correct one; Statement A is false in exactly one case, the load-use hazard, and Statement C describes a mechanism that forwarding does not use at all.

Why B is right: an arithmetic result is complete at the end of EX, a full cycle before the dependent instruction begins its own EX, so the EX/MEM path delivers it in time and nothing is lost. A loaded value is not complete until the end of MEM, one cycle too late, so a load followed at once by a dependent instruction still costs one cycle. B states both halves correctly.

What A gets wrong: A would be true if every producer finished its result in EX, and loads do not. The load-use hazard is the one case full forwarding cannot cover, and it is far from rare: loads are roughly a fifth of a typical instruction mix, so any program of real size contains a great many of them.

What believing A costs: a candidate who accepts A sets stalls per instruction to zero and reports a CPI of 1 for any program at all. On the mix used in the next question the true CPI is 1.12, so the error understates the cycles by 0.12 on every instruction and overstates the throughput of the machine by 12 per cent.

What C gets wrong: the register file write happens exactly when it always happened, in WB. Forwarding does not touch the register file at any point. It routes the value from a pipeline latch straight to the input of the arithmetic unit, and the register file is written later in the ordinary way, for the benefit of instructions far enough behind to read it normally.

Why C is tempting: the half cycle register file, which writes in the first half of a cycle and reads in the second, is a genuine trick and it does help. It belongs to the register file rather than to forwarding, and on its own it still leaves two cycles of stall on a dependent arithmetic pair rather than none.

Carry B exactly as it stands, because it names both the case that forwarding covers and the single case that it does not.

The Master Key: *Forwarding kills the arithmetic stall, leaves one cycle on a load-use pair, and does its work in the pipeline latches rather than in the register file.*

REAL Question 34

How much is a bundle of wires worth? A workshop at Kibondo has two boards on the bench, one with forwarding and one without, and a mix of instructions run through both settles it. The gap is wider than anybody in the room guessed.

A program contains 24 per cent load instructions, and half of all loads are immediately followed by an instruction that uses the loaded value. The machine has full forwarding. Calculate the CPI, and calculate the CPI the same program would suffer on a machine with no forwarding at all, where every one of the 40 per cent of instructions that depend on the immediately preceding result costs two cycles.

Loads are 24 per cent of the instructions executed.

Half of all loads are immediately followed by an instruction that uses the loaded value.

Without forwarding, 40 per cent of instructions depend on the immediately preceding result, and each costs two cycles.

Model Answer

With full forwarding the CPI is 1.12, and with no forwarding at all it is 1.8, so the bypass hardware buys about 61 per cent more throughput on this program.

Which instructions stall on the forwarding machine: only the loads whose value is used at once. Loads are 0.24 of the instructions and half of those are consumed immediately, so 0.24 multiplied by 0.5, which is 0.12 of all instructions, take a load-use stall. Every arithmetic dependence is covered by a bypass path and costs nothing.

CPI with forwarding: each of those stalls is exactly one cycle, because the value is late by one cycle and no more, so stalls per instruction is 0.12 multiplied by 1, which is 0.12, and the CPI is 1 plus 0.12, which is 1.12. Nothing else in the program adds a cycle.

$$\text{CPI} = 1 + 0.24 \times 0.5 \times 1 = 1.12$$

CPI without forwarding: now every dependent pair stalls and not merely the loads. Instructions that depend on the immediately preceding result are 0.40 of the program and each costs two cycles, so stalls per instruction is 0.40 multiplied by 2, which is 0.8, and the CPI is 1 plus 0.8, which is 1.8.

$$\text{CPI} = 1 + 0.40 \times 2 = 1.8$$

Machine	What stalls	Stall cycles per instruction	CPI
Full forwarding	Load-use pairs, 0.12 of instructions	0.12	1.12
No forwarding	All dependent pairs, 0.40 of instructions	0.80	1.80

Table 2.3: *The same program on two machines. Only the stall column differs, and it is the whole of the difference in CPI.*

Why the two totals differ so widely: the stall rate and the price of each stall both worsen at once. Stalling instructions rise from 0.12 to 0.40 of the program, which is 3.33 times as many, and each one costs two cycles instead of one, so the stall total rises from 0.12 to 0.8, which is 6.67 times larger.

The comparison: at one and the same clock, time per instruction is proportional to CPI, so the machine without forwarding takes 1.8 divided by 1.12, which is 1.607 times as long to run the identical program. Put the other way round, forwarding delivers about 61 per cent more instructions per second.

What the number is for: this is the figure that justifies the forwarding hardware. A design meeting does not vote on whether bypass wires are elegant; it asks what fraction of the machine they buy back, and 61 per cent of throughput for a few comparators and two multiplexers is a decision nobody argues with.

The mix decides everything here, so quote it alongside the answer: change the proportion of dependent pairs and both CPI figures move at once.

The Master Key: *Stalls per instruction is how often the stall happens multiplied by how many cycles it costs, which gives a CPI of 1.12 with forwarding and 1.8 without on this mix.*

HOT Question 35

A program compiled long before the machine on the desk was designed still runs on it, and nobody in the office at Bahi has ever wondered why. That single fact settles most of this argument before it starts, and it settles it narrowly.

Assess whether data hazards are better handled by the compiler, which can reorder instructions, or by the hardware, which can forward and stall.

Model Answer

Both belong in a serious machine, but the hardware must carry the main burden, because only the hardware is correct at run time and on code it has never seen before.

For the compiler, reordering is free: moving an independent instruction into the slot after a load costs no silicon at all. The bubble becomes useful work, the schedule is computed once at compile time, and the machine needs no comparator and no bypass wire for that case.

For the compiler, it sees the whole program: the hardware can only look at the instructions in flight, five of them in this pipeline. A compiler may search a whole loop or a whole function for something independent to move, and it can unroll a loop to manufacture independent work that did not exist in the source at all.

Against the compiler, the binary is tied to one pipeline: a schedule that is exactly right for a five stage machine is merely arbitrary on the next generation, which may have more stages and different bypass paths. The same binary then runs badly, and every program must be recompiled for each new machine.

Against the compiler, run time is invisible: whether a load hits or misses the cache cannot be known when the code is compiled, and a miss costs far more than the single cycle the schedule was built around. The compiler is scheduling against a delay that it can only estimate.

For the hardware, it adapts and it stays compatible: forwarding and interlocks act on whatever instructions actually arrive, and they work on code compiled years before the machine existed. They need no cooperation from anybody, which is why every commercial pipeline carries them.

Against the hardware, it costs silicon and clock: comparators in every stage, multiplexers on the inputs of the arithmetic unit, and extra wire on the critical path, which can force a slower clock and so tax every instruction in the program, dependent or not.

The verdict: use both, and if forced to choose one, choose hardware forwarding. Binary compatibility across generations is worth more than the last few per cent of throughput, and a machine with no interlocks at all is not merely slow when the compiler schedules badly, it is wrong. Reserve compiler scheduling for the load delay that forwarding provably cannot cover.

The division that real designers settle on follows capability exactly: hardware for what has to be right at run time, compiler for the one bubble the hardware cannot reach.

The Master Key: *Hardware first, because it is correct without cooperation and survives the next generation, with compiler scheduling kept for the load delay slot that no bypass path can fill.*

HOT Question 36

Everybody in the room liked the proposal, which is usually a warning. Take out the bypass wires, shorten the critical path, and the clock goes up twelve per cent. The meeting at Mafinga was almost over, and nobody had yet multiplied the CPI by anything.

A designer proposes removing all forwarding hardware from a processor in order to shorten the critical path and raise the clock frequency by 12 per cent. Using the figures from the program in question 34, determine whether the proposal is worthwhile.

Model Answer

The proposal should be rejected, because on this program the machine without forwarding takes about 43.5 per cent longer for every instruction: a CPI rising from 1.12 to 1.8 costs far more than a 12 per cent clock returns.

Set the comparison up properly: time per instruction is CPI divided by clock frequency, so a change that moves both must be judged on that ratio and never on either figure by itself. Quoting a clock speed on its own is the commonest way in which a slower machine is sold as a faster one.

The two machines: with forwarding, a CPI of 1.12 at frequency f, taken from the program in which 24 per cent of instructions are loads and half of those loads are consumed immediately. Without forwarding, a CPI of 1.8 at frequency 1.12f, because 40 per cent of instructions depend on the immediately preceding result and each then costs two cycles.

The arithmetic: the CPI penalty is 1.8 divided by 1.12, which is 1.607. The clock gain divides that penalty by 1.12, leaving 1.435. The stripped machine therefore takes about 43.5 per cent longer on the same work, which is about 30 per cent less delivered throughput, and it was offered to the meeting as an improvement.

$$\text{time ratio} = (1.8 / 1.12) / 1.12 = 1.435$$

The break-even condition: removing forwarding pays only when the clock gain exceeds the CPI penalty. The penalty here is a factor of 1.607, so the frequency would have to rise by more than 60.7 per cent before the two machines merely drew level, and taking a few wires off the critical path delivers nothing of that order.

What would reverse the verdict: the instruction mix, and nothing else. If only 6 per cent of instructions depended on the immediately preceding result, the machine without forwarding would also reach a CPI of 1.12, the two would tie on cycles and the faster clock would win outright. Real programs are nowhere near so independent, but that condition is the honest form of the answer.

The judgement: reject the proposal for this workload and keep the bypass network. Accept a clock-raising simplification only when the percentage gain in frequency exceeds the percentage penalty in CPI, and here it would need better than 60 against the 12 that is on offer.

A clock figure quoted without a CPI is not a performance claim at all, and this proposal is that trap dressed up as an economy.

The Master Key: *Compare CPI divided by frequency and never frequency alone, because 12 per cent of clock cannot buy back a CPI that has risen from 1.12 to 1.8.*

Control hazards, branch prediction and the branch penalty

This section takes the third class of hazard, the one branches cause, and works it from the definition through to the arithmetic. It defines a control hazard precisely, which matters because this is the definition most often learned in the wrong form, then builds the predictors in order: the static schemes, the one bit dynamic predictor and the weakness it shows on a loop, then the two bit saturating counter and the branch target buffer. It draws the state machine, draws the flush that follows a wrong guess, counts the instructions discarded, and puts CPI figures on predictors of different accuracy. It closes by asking whether prediction is worth more in a deep pipeline than in a shallow one, and finds that the honest answer is stronger than the usual claim.

BINDER Question 37

The fork outside Mbulu lost its signpost to a lorry two seasons ago, and every driver must decide, at speed, before the junction arrives. Slowing to think costs minutes; guessing wrong costs an hour. The difficulty is there whether the driver guesses or not.

Define a control hazard and explain precisely why one arises.

Model Answer

A control hazard is the condition in which the pipeline cannot know which instruction to fetch next, because a branch has entered the pipeline and its outcome has not yet been determined.

Fetch cannot wait: the fetch stage must start an instruction in every cycle, or the pipeline stops delivering one completion per cycle. A conditional branch fetched in cycle 1 leaves IF needing the address of its successor at the start of cycle 2, and there are two candidates for that address: the instruction lying next in memory, and the branch target.

Resolution comes late: the direction a conditional branch takes depends on a comparison of register values, and a comparison is arithmetic, so it is performed in EX. The answer therefore exists only at the end of cycle 3. Fetch needed it at the start of cycle 2, so two cycles of fetching are done in the dark.

A wrong prediction does not cause it: the hazard is created by fetch running ahead of resolution, and it is present the moment the branch enters IF, before the machine has decided what to do about it. A processor that predicted nothing at all would suffer exactly the same hazard; it would hold the fetch stage still until each branch resolved, and would never mispredict anything.

The hazard and its bill are different things: the hazard is the uncertainty, and it is fixed by where fetch and resolution sit in the pipeline. What the uncertainty costs depends on the remedy: stall, and pay on every branch, or predict, and pay only when the guess turns out wrong. Students are often taught, wrongly, that the misprediction is the hazard, when it is only one way of paying for it.

Put the hazard in the gap between fetching a branch and resolving it, and the rest of this topic follows without difficulty.

The Master Key: *A control hazard is fetch not knowing the next address while the branch is still in flight; the misprediction is the bill, never the hazard.*

BINDER Question 38

At the sisal estate weighbridge in Kiteto the rule never changes, whatever the load: every lorry turns left after the gate, because nearly every lorry is bound for the drying yard. The one wanting the workshop reverses out, and the estate accepts that cost.

Describe static branch prediction and state the accuracy each simple scheme can be expected to achieve.

Model Answer

Static branch prediction fixes the guess before the program runs and never revises it, so predict-not-taken assumes every branch falls through and predict-taken assumes every branch jumps; on a loop of ten iterations those two schemes score 10 per cent and 90 per cent on the very same branch.

Predict not taken: the fetch stage carries on down the sequential path as though the branch were not there, and if the branch turns out to be taken, the work behind it is discarded. This is the cheapest scheme there is: no table, no stored state and no lookup, because the machine does exactly what it would have done anyway.

Predict taken: this scheme needs the destination, and the destination is encoded inside the branch instruction, so it is not available until the branch has been decoded. The fetch stage therefore cannot follow the target in the cycle immediately after the branch, and predict-taken saves less than its accuracy suggests unless a target buffer supplies the address early.

Why the direction matters: a loop branch is taken on every iteration except the last, when the loop finally exits. Over 10 iterations it is taken 9 times and not taken once, so predict-taken is right 9 times out of 10 and predict-not-taken is right 1 time out of 10. The two schemes score 90 per cent and 10 per cent on identical code.

What the compiler can add: a compiler reads the shape of the code and marks the likely direction in the instruction itself, so a backward branch at the foot of a loop is marked taken and a forward branch guarding an error path is marked not taken. The scheme stays static, because the mark is fixed when the program is compiled and the same guess is made on every execution.

Static prediction is right about as often as the code is regular, and it costs almost nothing to provide.

The Master Key: *Static means one fixed guess for the life of the program, so predict-taken takes 9 of 10 on a loop while predict-not-taken takes 1 of 10 on the same branch.*

BINDER Question 39

Two mistakes a season, no more and no fewer, at the football ground in Bunda. The gatekeeper opens whichever gate the crowd used last time, so the week the fixture moves he is wrong, and wrong again the week after, his correction already stale.

Describe the one bit dynamic branch predictor, and explain the weakness it shows on a loop.

Model Answer

A one bit dynamic predictor holds a table of single bit entries, indexed by the low order bits of the branch address, each recording what that branch did the last time it executed, and it predicts that the branch will do the same again; on a loop of 10 iterations it is wrong twice, giving 8 correct out of 10 and an accuracy of 80 per cent.

The structure: the low order bits of the branch address select an entry, and the bit in that entry is written to taken or not taken each time the branch resolves. The predictor buffer and the branch history table are two names for this same structure, and how many bits an entry holds is a separate question from what the table is called.

Right for nine iterations, wrong at the exit: on iterations 1 to 9 the branch is taken and the bit already says taken, so all 9 of those predictions are correct. On the tenth iteration the loop finishes, the branch is not taken and the bit still says taken, which is the first misprediction; resolving it then flips the bit to not taken.

Wrong again on re-entry: when that loop is executed again, on the next pass through the routine containing it, the entry still says not taken, so the very first iteration is mispredicted as well. That is the second miss, and it was caused by the correction the predictor made at the exit of the previous run.

The arithmetic and the lesson: 2 misses in 10 outcomes leave 8 correct out of 10, an accuracy of 80 per cent, on a branch that behaves the same way 9 times in 10, which is 90 per cent regular. One bit cannot distinguish an unusual outcome from a change of habit, so a single odd result destroys the prediction outright.

The one bit predictor is beaten by a plain loop, which is why the second bit was added.

The Master Key: *One bit remembers only the last outcome, so a loop of ten costs it two misses and 80 per cent on a branch that is 90 per cent regular.*

BINDER Question 40

It took two bad harvests, not one, before the maize buyer at Mwanga moved his price. A single poor year he put down to the weather; the second changed his mind, and the traders who had shouted came to see the sense of it.

Describe the two bit saturating counter, explain how it repairs the weakness of the one bit predictor, and state what a branch target buffer adds.

Model Answer

A two bit saturating counter gives every entry 4 states, 00 strongly not taken, 01 weakly not taken, 10 weakly taken and 11 strongly taken, so two outcomes in the same direction are needed before the prediction changes; on the same loop of 10 iterations only the exit is mispredicted, and accuracy rises from 80 to 90 per cent.

How the counter moves: a taken outcome moves the counter one step towards strongly taken and a not-taken outcome moves it one step the other way, and both ends saturate, so a further taken outcome at 11 simply leaves it at 11. The prediction is which half of the range the counter sits in, as Figure 2.11 shows: 00 and 01 predict not taken, 10 and 11 predict taken.

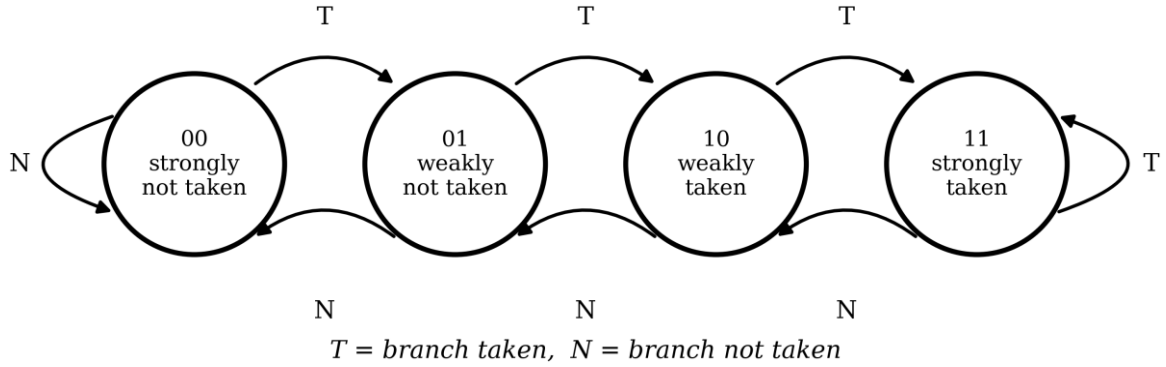


Figure 2.11: *The two bit saturating counter. Two outcomes in the same direction are needed to change the prediction, so a single unusual branch does not flip it.*

Why the loop improves: after 2 taken iterations the counter has reached strongly taken and stays there. The exit is mispredicted once and drops the counter to weakly taken, which still predicts taken, so the first iteration of the next run of the loop is predicted correctly. One miss in 10 rather than two lifts accuracy from 0.8 to 0.9.

What a branch target buffer adds: the direction is only half the answer, because a branch predicted taken also needs a destination, and that address is not computed until the branch is decoded. The branch target buffer is a small cache, indexed by the branch address, holding the address that branch jumped to last time, so the fetch stage can redirect itself in the same cycle in which it fetches the branch.

Direction and destination together: with a counter for the direction and a target buffer for the address, a taken branch can be followed without losing a single cycle, provided both tables are right. Miss in either one and the full penalty returns.

The second bit buys patience and the target buffer buys the address, and a predictor needs both before a taken branch can cost nothing.

The Master Key: *Two bits mean two wrong outcomes are needed to change the prediction, turning 80 per cent into 90 on a loop, while the target buffer supplies the address direction alone cannot.*

REAL Question 41

Everything already in the oven came out and went straight into the bin. The order was cancelled after the bread went in, at the bakery beside the bus stand in Tandahimba, and neither the flour nor the heat could be got back.

A branch is resolved at the end of the execute stage. Determine how many instructions must be discarded on a misprediction, explain what a flush is, and state what speculative execution means.

Model Answer

Two instructions must be discarded, because the branch is fetched in cycle 1 and resolved only at the end of cycle 3, so the instructions fetched in cycles 2 and 3 lie on the wrong path; flushing is the act of turning them into no-operations, and having run them at all is what speculative execution means.

Count the cycles: the branch occupies IF in cycle 1, ID in cycle 2 and EX in cycle 3, and its condition is known only at the end of cycle 3. In the meantime the fetch stage took one instruction in cycle 2 and another in cycle 3, both from the sequential path. If the branch is taken, neither belongs to the program, and the target is fetched in cycle 4. Figure 2.12 marks the two discarded slots.

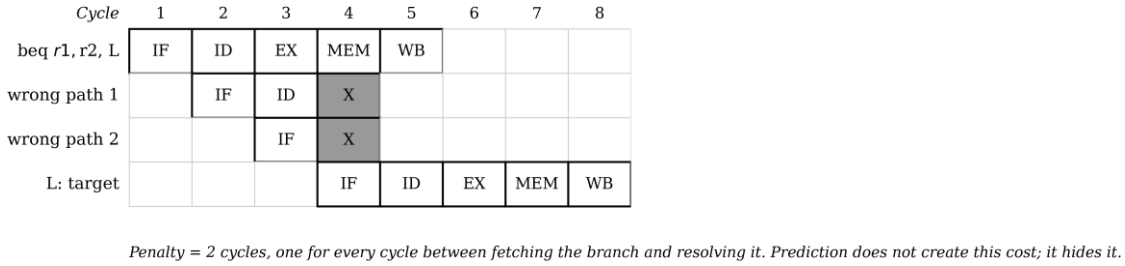


Figure 2.12: *A branch resolved at the end of the execute stage. The two instructions fetched behind it are wrong-path work and are discarded.*

The general rule: the penalty equals the number of cycles between fetching the branch and resolving it, which here is 3 minus 1, or 2. Move the resolution back to MEM and the penalty becomes 3. A twenty stage pipeline that resolves its branches in stage twelve discards 11 instructions on every wrong guess.

$$\text{penalty} = \text{resolution cycle} - \text{fetch cycle} = 3 - 1 = 2$$

What flushing is: the wrong-path instructions are converted into no-operations while they are still inside the pipeline, by clearing the control signals they carry. They may have been decoded and even executed, but they are stopped before any stage that changes state, so no register is written, no memory location is altered and no flag is set by work that should never have happened.

What speculative execution means: the machine executes down the predicted path before it knows the path is right. It can do so safely only because the results are held back rather than committed: they sit in pipeline registers until the branch resolves, and are released if the guess was right and destroyed if it was wrong.

The penalty belongs to the design: nothing the program does can alter the count, because it is set by the distance between the stage that fetches and the stage that resolves. That is why designers work hard to resolve branches early, and why deepening a pipeline raises the price of every wrong guess.

Count the cycles from fetch to resolution and you have the penalty, before you know anything at all about the predictor.

The Master Key: *The branch penalty is the number of cycles between fetching a branch and resolving it, and flushing is what makes those cycles harmless.*

REAL Question 42

Three sentences went up on the board in the laboratory at Kasulu and the class was asked to pick the sound one. Most hands went up for the first, a few for the third, and one for the second, which was the only true sentence.

Three statements about control hazards are given below. Identify the one that is correct, and explain precisely what is wrong with each of the other two.

Statement A: A control hazard occurs when the pipeline makes the wrong decision about a branch prediction.

Statement B: A processor with a perfect branch predictor would still lose two cycles on every conditional branch.

Statement C: A control hazard exists because the next address is unknown when the fetch stage needs it, whether or not the processor predicts.

Model Answer

Statement C is the correct one; Statement A mistakes the cost of a control hazard for the hazard itself, and Statement B charges a perfect predictor for a penalty that only a wrong guess can incur.

Why C is right: the fetch stage needs an address in every cycle, and after a conditional branch that address is genuinely unknown until the branch resolves several cycles later. The uncertainty appears the instant the branch is fetched, whatever the machine goes on to do about it, and that uncertainty is the whole of the hazard.

What A confuses: a misprediction is one possible consequence of the hazard, not its cause. Build a machine that never predicts and simply holds the fetch stage still until each branch resolves: it suffers the hazard on every branch and mispredicts nothing whatever, which on its own shows that the hazard and the wrong guess are separate things.

The test that settles A: ask whether the problem is already present before any decision has been taken. The uncertainty is; the misprediction cannot be, because at that moment there is nothing yet to be wrong about. A condition that comes before the guess cannot have been caused by the guess.

Why B is wrong in the other direction: when the prediction is correct the fetch stage was on the right path all along, so nothing is discarded and nothing has to be fetched again. A perfect predictor therefore pays nothing beyond looking up the target address, and the machine goes on completing one instruction per cycle straight through its branches.

Where B gets its two cycles: the penalty of 2 cycles is real, but it is paid only when the guess is wrong. Statement B has taken a cost that is conditional on being wrong and made it unconditional, which is the same confusion as Statement A running in the opposite direction.

The correction in one line: the hazard is unconditional and its penalty is conditional. A machine cannot escape the first, because fetch will always run ahead of resolution, and it escapes the second exactly as often as its predictor is right.

A and B fail together: one treats the penalty as the hazard, the other treats it as something even a perfect predictor must pay.

The Master Key: *The hazard is the unknown address and the penalty is what a wrong guess costs, so a right guess costs nothing at all.*

REAL Question 43

A question came from the back at the training centre in Mafinga: if the predictor is already right nine times out of ten, how much is left to gain? It sounds like a small remainder, and the sum on the board answered a different question.

A program contains 20 per cent conditional branches. The branch penalty is three cycles. Calculate the CPI when the predictor is 90 per cent accurate, when it is 95 per cent accurate, and when there is no prediction at all and every branch stalls for the full penalty.

Model Answer

The CPI is 1.06 with a 90 per cent predictor, 1.03 with a 95 per cent predictor and 1.6 with no prediction at all, so nearly the whole value of prediction lies in having one rather than in having a good one.

The shape of the calculation: the added cycles per instruction are the product of three things: how often branches occur, how often the prediction is wrong, and how many cycles each wrong guess costs. Add that product to 1 and the CPI follows, since an in-order pipeline runs at 1 plus its stalls per instruction.

$$\text{CPI} = 1 + 0.20 \times 0.10 \times 3 = 1.06$$

At 90 per cent accuracy: the misprediction rate is 0.10, so mispredictions per instruction are 0.20 multiplied by 0.10, which is 0.02, meaning one instruction in fifty sets off a flush. Each of those costs 3 cycles, so the added cycles per instruction are 0.06 and the CPI is 1.06. The 9 branches in every 10 that are guessed correctly cost nothing whatever.

At 95 per cent accuracy: the misprediction rate halves to 0.05, so 0.20 multiplied by 0.05 gives 0.01 mispredictions per instruction, the added cycles are 0.03 and the CPI is 1.03. Halving the errors of a good predictor has bought 0.03 cycles per instruction.

With no prediction at all: every branch stalls for the full penalty whether it is taken or not, so the cost is 0.20 multiplied by 3, which is 0.6 added cycles per instruction and a CPI of 1.6. That machine spends more than half as long again on the same program as the one with a 90 per cent predictor. Prediction is not a refinement here; it is most of the pipeline.

Accuracy	Mispredictions per instruction	Added cycles per instruction	CPI
None, every branch stalls	not applicable	0.6	1.6
90 per cent	0.02	0.06	1.06
95 per cent	0.01	0.03	1.03

Table 2.4: *Three predictors costed on a program of 20 per cent conditional branches with a three cycle penalty.*

What the numbers show: moving from no prediction to a 90 per cent predictor saves 0.54 cycles per instruction, while moving from 90 to 95 per cent saves 0.03. The first step is worth 18 times the second, so fitting a predictor at all is beyond argument, whereas fitting a better one must be justified on much finer margins.

Cost a predictor against having none before you cost it against a rival predictor, because that is where nearly all of the cycles actually are.

The Master Key: *Added cycles per instruction are branch frequency times misprediction rate times penalty, so 20 per cent branches and a three cycle penalty give 1.06 at 90 per cent accuracy and 1.6 with no prediction.*

HOT Question 44

There was money for one of two quotations at the design review for a controller destined for the water pumps around Chato. One bought a bigger prediction table, the other a shorter front end. The team expected the arithmetic to separate them, and it refused.

A design team can spend its remaining budget either on a larger branch predictor that raises accuracy from 92 to 96 per cent, or on halving the branch penalty from four cycles to two. The program contains 18 per cent conditional branches. Advise the team.

The program: 18 per cent of instructions are conditional branches.

Option A: raise predictor accuracy from 92 per cent to 96 per cent, with the penalty left at 4 cycles.

Option B: halve the branch penalty from 4 cycles to 2 cycles, with accuracy left at 92 per cent.

Model Answer

Both options give a CPI of 1.0288 against a baseline of 1.0576, so the arithmetic cannot separate them at all; take the shorter penalty, unless resolving the branch earlier would cost more than about 2.8 per cent of the clock frequency, in which case take the predictor.

The baseline: at 92 per cent accuracy the misprediction rate is 0.08, so mispredictions per instruction are 0.18 multiplied by 0.08, which is 0.0144. At 4 cycles each that is 0.0576 added cycles per instruction, and a CPI of 1.0576.

Option A, the larger predictor: raising accuracy from 92 to 96 per cent halves the misprediction rate from 0.08 to 0.04. The added cycles become 0.18 multiplied by 0.04 multiplied by 4, which is 0.0288, and the CPI falls to 1.0288.

Option B, the shorter penalty: halving the penalty leaves the misprediction rate at 0.08 and gives 0.18 multiplied by 0.08 multiplied by 2, which is also 0.0288, and the same CPI of 1.0288. The two options are not merely close; they are identical.

$$0.18 \times 0.04 \times 4 = 0.0288 = 0.18 \times 0.08 \times 2$$

Why they must be equal: the cost is a product of three terms, branch frequency, misprediction rate and penalty. Halving any one of them halves the product, and going from 8 per cent wrong to 4 per cent wrong is precisely a halving, so the two proposals are the same change wearing different clothes.

What the shorter penalty has that the predictor has not: it helps every branch in every program, including the branches no history can learn. A branch whose direction turns on unpredictable data, such as whether a meter reading crossed a threshold, defeats any table; on that code the bigger predictor buys nothing while the shorter penalty still halves the loss.

What it may cost in return: resolving a branch earlier means moving the comparison into an earlier stage, which lengthens the work that stage must finish inside one clock period. Should that force the clock down, the machine loses on every instruction rather than only on branches, and the entire prize here is 2.8 per cent.

The verdict: take Option B if the earlier resolution can be had without cutting the clock, because it is the more robust of the two and it protects exactly the code the predictor is worst at. If resolving earlier would cost more than about 2.8 per cent of clock frequency, buy Option A instead, since that figure is the whole margin either option delivers.

When two changes give the same CPI, the decision moves to the clock and to the branches no table can learn.

The Master Key: *Branch cost is frequency times misprediction rate times penalty, so halving either factor is worth the same, and the choice turns on clock frequency and on unlearnable branches.*

HOT Question 45

A claim made with great confidence at a teachers workshop in Ludewa: the deeper the pipeline, the more a good predictor is worth. It has the ring of a slogan, but it is arithmetic underneath, and the arithmetic turns out harsher than the slogan.

Assess the claim that branch prediction becomes more valuable as pipelines get deeper.

Model Answer

The claim is right in direction and too gentle in strength: the penalty grows with depth, so the same 90 per cent predictor costs seven and a half times as much on a twenty stage machine as on a five stage one, which makes prediction compulsory rather than merely valuable, and past a certain depth no achievable accuracy rescues the design.

Why depth raises the penalty: the penalty is the number of cycles between fetching a branch and resolving it, and deepening cuts the same work into more and shorter stages, pushing resolution further from fetch. A five stage machine resolving in EX loses 2 cycles; a twenty stage machine resolving in stage sixteen loses 15.

The comparison in numbers: take 15 per cent conditional branches and a predictor right 90 per cent of the time. On the short pipeline the cost is 0.15 multiplied by 0.10 multiplied by 2, or 0.03 cycles per instruction, a CPI of 1.03. On the deep one it is 0.15 multiplied by 0.10 multiplied by 15, or 0.225, a CPI of 1.225, which is seven and a half times as much for identical code.

The same accuracy is not the same accuracy: to hold the deep machine to the 0.03 cycles per instruction the short machine reaches, its misprediction rate would have to fall to about 0.013, an accuracy of about 98.7 per cent. Depth raises the standard a predictor must meet, not merely the reward for meeting it.

The bound on the other side: a predictor cannot be right more than 100 per cent of the time, so its returns are capped while depth is not. Once accuracy stands in the high nineties the headroom left is a fraction of a cycle per instruction, while every further stage adds directly to the penalty.

What prediction cannot touch: a deeper pipeline also takes longer to fill and to drain, and the clock frequency each added stage buys falls away as pipeline register overhead takes a larger share of a shorter cycle. No predictor hides either cost.

The verdict: the claim is true but understated. In a deep pipeline prediction is a precondition, not an optimisation, because without it the machine loses more to branches than the extra clock frequency ever gained, and there is a depth beyond which even excellent prediction cannot pay for the fill, the drain and the penalty.

The condition that decides it: add depth only while the clock frequency a new stage buys is worth more than the CPI that stage adds through the longer penalty. Past that crossing the machine is slower however good its predictor.

Depth makes prediction compulsory rather than valuable, and there is a depth at which no predictor is good enough.

The Master Key: *The branch penalty grows with depth while accuracy is capped at 100 per cent, so deep pipelines cannot run without prediction and eventually outrun what any predictor can repay.*

Instruction level parallelism: multiple issue, dependences and out-of-order execution

This section takes a single instruction stream as far as it will go. It defines instruction level parallelism and separates it from pipelining by one number, distinguishes scalar from superscalar and fixes what issue width

means, then completes the dependence taxonomy with the two name dependences and shows how register renaming disposes of both. It draws the issue slots and it draws the out of order engine, costs a loop of 32 multiplications three ways, and argues out whether the compiler or the hardware should be doing the finding. Sections 7 and 8 take the parallelism that lies outside one stream.

BINDER Question 46

Two lorries stand at the cashew store in Tandahimba, and both could be loaded at once if the sacks for one were not stacked behind those for the other. The bays are free; what sets the pace is how the sacks were piled last week.

Define instruction level parallelism and explain how it differs from pipelining.

Model Answer

Instruction level parallelism is the property of a program that instructions lying close together do not depend on one another and could therefore be executed at the same time, together with the compiler and hardware techniques that exploit that property.

The property first, then the machinery: the parallelism is in the code before any processor is built, because it is nothing more than the absence of dependences between neighbouring instructions. What the hardware and the compiler add is the ability to find that independence and act on it by starting several instructions in the same cycle.

The difference from pipelining, put as a number: a single issue pipeline overlaps different stages of different instructions, so five instructions can be in flight at once, and yet it can never complete more than one instruction per cycle, which caps its IPC at 1. Instruction level parallelism means issuing more than one instruction in the same cycle, and that is what lifts IPC above 1.

The contradiction to avoid: students are often taught, wrongly, that pipelining is itself a form of instruction level parallelism, while being told in the same breath that instruction level parallelism means several operations in one cycle. Both cannot hold. IPC settles which is meant, because a scalar pipeline can never pass 1.

One stream, and two other ideas handed on: all of this happens inside a single instruction stream. Running several streams at once, and applying one operation to many data elements at once, are different ideas altogether, and Sections 7 and 8 take them up in turn.

Ask what the IPC ceiling is, because that single number separates a pipeline from a machine with real instruction level parallelism.

The Master Key: *Instruction level parallelism is independence between neighbouring instructions turned into more than one instruction issued per cycle, so its signature is an IPC above 1.*

BINDER Question 47

The sales sheet said four way, and the technician at the internet shop in Bunda read that as four processors. What he had bought was one processor with four issue slots, and on his own workload it behaved like neither.

Distinguish a scalar processor from a superscalar processor, and explain what is meant by issue width.

Model Answer

A scalar processor issues at most one instruction per clock cycle however many execution units it holds, while a superscalar processor fetches, decodes and issues several instructions in the same cycle; the issue width is the largest number it can issue in one cycle.

What a scalar machine cannot do: a scalar machine may be deeply pipelined and may carry an adder, a multiplier and a divider, and it will still start exactly one instruction in each cycle. The spare units stand idle, because nothing is ever sent to fill them.

What a superscalar machine has to add: a wider path from the instruction cache, duplicated execution units, more ports on the register file, and logic that examines the candidate instructions in every single cycle to decide whether they are independent enough to travel together. That checking is the expensive part of the design.

Reading the three panels: the same eight independent instructions occupy 8 cycles at width one, 4 cycles at width two and 2 cycles at width four, as Figure 2.13 shows. Divide instructions by cycles and the IPC is 1, then 2, then 4, while the work performed is identical in all three.

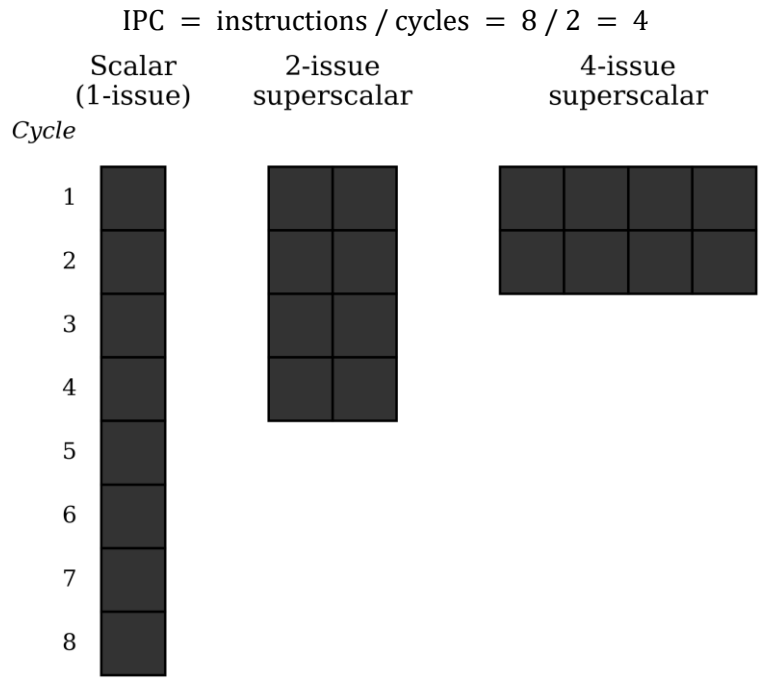


Figure 2.13: *The same eight instructions on a scalar machine, a two issue machine and a four issue machine. The work is identical; only the number of slots per cycle changes.*

The honest limit on those numbers: the panels assume every instruction is independent of every other. Real code is not like that, so a four issue machine rarely finds four independent instructions waiting in a cycle, and its achieved IPC sits well below its width. Width is a ceiling and never a promise.

Quote the width when you are asked what the machine can issue, and quote the achieved IPC when you are asked what it delivers.

The Master Key: *Scalar means one instruction issued per cycle and superscalar means several, while issue width is the maximum per cycle that the achieved IPC almost never reaches.*

BINDER Question 48

The ledger at the ward office in Mbulu has one column two clerks were filling at the same time, and neither needed anything the other had written. Give each a fresh sheet and both can write at once; the heading was all they shared.

Explain what a name dependence is, distinguish write after read from write after write, and describe how register renaming removes both.

Model Answer

A name dependence exists when two instructions use the same register name without passing any value between them, so the constraint on their order comes from the name alone and disappears the moment the two are given different registers.

Write after read: a later instruction writes a register that an earlier instruction has still to read. Reorder the pair and the earlier instruction reads the new value instead of the old one, which is wrong, and yet no value passes from the first to the second at all.

Write after write: two instructions write the same register. Reorder them and the wrong one is left holding the register at the end, so every later reader sees a stale value. Again nothing flows between them; they merely collide over a name.

What renaming does: the processor keeps a pool of physical registers far larger than the set of names the instruction set offers, and a rename table mapping each architectural name to one physical register. Every write allocates a fresh physical register and updates the map, so two writes to the same name land in two different places.

Instruction	Architectural register written	Physical register allocated	Dependence removed
I1: add r1, r2, r3	r1	p1	none, this is the first write to r1
I2: sub r4, r1, r5	r4	p2	none, the read after write on r1 from I1 is real and stays
I3: mul r5, r6, r7	r5	p3	write after read on r5 against I2 removed
I4: add r4, r8, r9	r4	p4	write after write on r4 against I2 removed

Table 2.5: *Four instructions passing through the rename table. Every write takes a fresh physical register, so both name dependences vanish while the genuine read after write survives.*

What renaming does not do: renaming does not change the instruction set and does not change the number of operands an instruction carries. It is a mapping performed inside the processor at issue time, invisible to the program, so the identical binary runs on a machine that has no rename table at all.

The dependence that survives: read after write cannot be renamed away, because there a value genuinely passes from one instruction to the next. Renaming clears the two false constraints and leaves the true one standing, which is the real limit on how far code can be reordered.

Test any pair by asking whether a value flows between them, because if none does the dependence is only a name.

The Master Key: *Write after read and write after write are collisions over a register name rather than over a value, so a fresh physical register for every write dissolves both.*

BINDER Question 49

Nobody at the fuel depot in Kasulu drives out of the gate before the lorry ahead is billed, yet the pumps are taken in whatever order they fall free. A driver blocked at a slow pump does not hold up the six behind him.

Describe dynamic scheduling, and explain why instructions are issued in order and committed in order but executed out of order.

Model Answer

Dynamic scheduling is the hardware technique of letting an instruction begin execution as soon as its operands are ready rather than when its turn arrives, so the machine issues in program order, executes in whatever order the data permits, and commits in program order once more.

Why issue stays in order: issuing in program order keeps dependence checking tractable, because each new instruction is compared against a known set of earlier ones, and it keeps the original order of the program available for later use. Reservation stations then hold each issued instruction until its operands arrive.

Why execution runs out of order: an instruction waiting on a load that missed the cache may be held up for dozens of cycles. Behind it sit instructions that need nothing from it, and letting those proceed keeps the execution units working instead of idle. That, and nothing else, is where the performance comes from.

Why commit must be in order: results wait in a reorder buffer and are made permanent strictly in program order. The reason, usually left out, is precise exceptions and recovery: if an instruction faults, or a speculated branch turns out to have gone the wrong way, the machine must show a state in which every earlier instruction has finished and no later one has changed anything, and only in order commit can guarantee it.

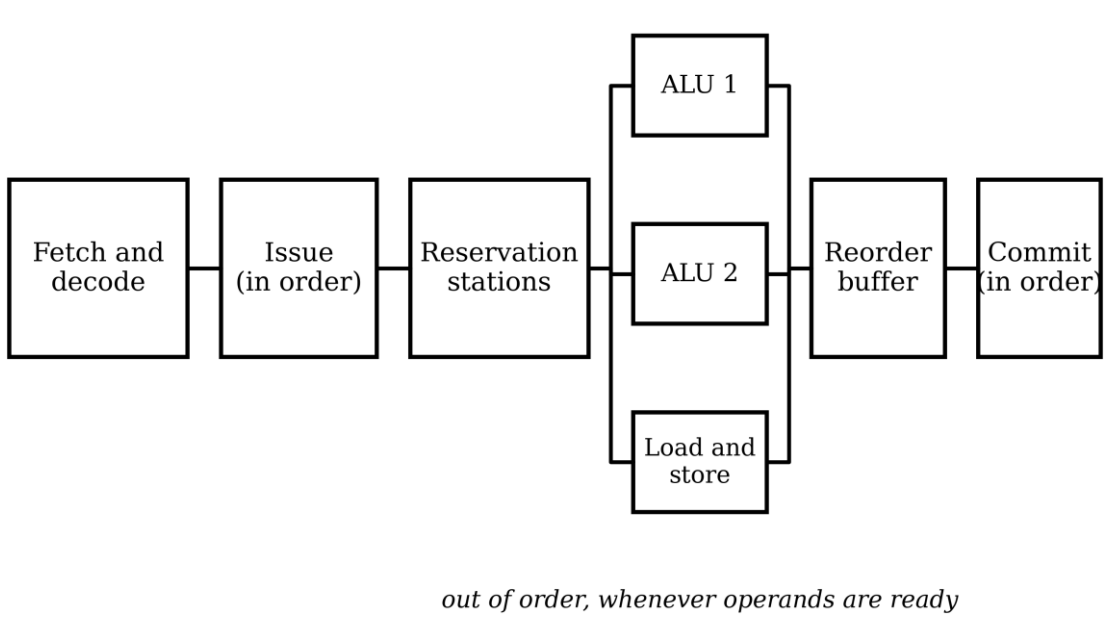


Figure 2.14: *In order issue, out of order execution, in order commit. The reorder buffer restores program order before anything is made permanent.*

What the buffer costs: every result sits in the buffer until all older instructions have committed, so the buffer must be deep enough to cover the longest delay the design will tolerate, and its entries, its comparators and its wakeup logic are clocked in every cycle.

Out of order execution buys the speed and in order commit keeps the machine correct, which is why any real design carries both.

The Master Key: *Issue in order to check dependences, execute out of order to hide waiting, and commit in order so that a fault or a misprediction always finds a clean program state.*

REAL Question 50

Ninety six instructions to perform 32 multiplications: that was the count a student at Kilolo obtained, and she could not see where the other 64 had come from. They were the counter and the branch, paid in every iteration, and most did not need to be there.

A loop performs 32 independent multiplications, one per iteration, with two instructions of loop overhead per iteration. Calculate the cycles required on a single issue machine, then after unrolling by four, then after unrolling by four on a two issue machine, and find the speed-up.

The loop runs 32 iterations and performs one multiplication in each.

Each iteration carries two further instructions of loop overhead.

The iterations are independent: no iteration uses a result produced by another.

Model Answer

The loop takes 96 cycles as written, 48 cycles unrolled by four, and 24 cycles unrolled by four on a two issue machine, so unrolling alone is worth a speed-up of 2.0 and unrolling with dual issue is worth 4.0.

Independence is the precondition: none of this works unless the iterations are genuinely independent, and here they are, since each multiplication reads its own data and writes its own result. A loop that adds every iteration into one running total is a different case: those additions form a chain of read after write dependences through the total, and no amount of unrolling lets two of them issue together.

The loop as written: each iteration costs one multiplication plus two overhead instructions, which is 3 instructions, and 32 iterations give 32 multiplied by 3, or 96 instructions. A single issue machine starts one instruction per cycle, so the loop occupies 96 issue cycles.

Unrolling by four: put four copies of the body inside one iteration and the counter and the branch are paid once for every four multiplications instead of once for each. The 32 multiplications remain, the number of groups is 32 divided by 4, which is 8, and the overhead becomes 8 multiplied by 2, which is 16. That gives 32 plus 16, or 48 instructions.

Adding a second issue slot: the unrolled body is a run of independent multiplications, so two can be issued in every cycle. Those 48 instructions at two per cycle need 48 divided by 2, which is 24 issue cycles. Unrolling has now paid twice, once for shedding overhead and again for supplying the independent instructions a wide machine needs.

$$\text{speed} - \text{up} \; = \; 96 \; / \; 24 \; = \; 4.0$$

The two speed-ups: measure both against the original 96 cycles. Unrolling alone gives 96 divided by 48, which is 2.0, and unrolling with dual issue gives 96 divided by 24, which is 4.0. No instruction became faster and the work is unchanged; the wasted cycles simply stopped being spent.

What unrolling costs: four results are live at once instead of one, so the demand for registers rises, and the loop body is four times larger, which may push it out of the instruction cache.

Unrolling removes overhead and exposes independence, and it is the second of those that a superscalar machine is really paying for.

The Master Key: *Unroll by four and 96 instructions become 48, then issue two per cycle and those 48 become 24 cycles, which is a speed-up of 4.0 on independent iterations only.*

REAL Question 51

Three sentences on a card taped above the counter of the computer repair shop in Newala, all written in the same confident hand, and only one of them survives a careful reading. The other two carry the two commonest errors in this whole topic.

Three statements about instruction level parallelism are given below. Identify the one that is correct, and explain precisely what is wrong with each of the other two.

Statement A: Processors that exploit instruction level parallelism need simpler hardware than processors that do not.

Statement B: A loop whose every iteration adds into the same running total cannot be issued two instructions at a time, however far it is unrolled.

Statement C: A four issue superscalar processor and a four core processor are two names for the same machine.

Model Answer

Statement B is the correct one; Statement A is exactly backwards, and Statement C confuses two entirely different kinds of parallelism.

Why B is right: every iteration reads the running total, adds to it and writes it back, so each addition needs the value the previous addition produced. That is a read after write chain, a genuine flow of data, and renaming cannot touch it. Unrolling copies the body but does not break the chain, so the additions still happen one after another.

Why A is backwards: a superscalar out of order processor needs a wide fetch path, logic that checks every pair of candidates in every cycle, a rename table, reservation stations, a reorder buffer and a register file with many ports. An in-order scalar machine has none of these. Such parallelism is bought with hardware, not by saving it.

Where A leads if it is believed: the error hides the reason the industry changed direction. It was precisely this rising cost, growing far faster than the performance it returned, that eventually persuaded designers to stop widening single cores.

Why C is false: a four issue superscalar has one instruction stream and one program counter, and it searches inside that single stream for instructions that happen to be independent. A four core processor has four streams and four program counters, running different work altogether.

The limit each one runs into: the superscalar is limited by how much independence exists inside one program, which is modest. The multicore machine is limited by whether the work can be split into separate threads at all, which is a question for the programmer rather than for the hardware.

Two of the three fall to the same pair of questions: is the dependence real, and does the parallelism lie inside one stream or across several.

The Master Key: *Instruction level parallelism costs more hardware rather than less, a serial accumulation cannot be widened by unrolling, and issue width is not a count of cores.*

HOT Question 52

A machine at the training centre in Chunya still runs, and a shelf of software beside it no longer does. Every program there was compiled for the exact timing of a processor since replaced; the new one will not keep to the old schedule.

Assess whether the compiler or the hardware should be responsible for finding instruction level parallelism, referring to very long instruction word and superscalar designs.

Model Answer

The hardware should find it in a general purpose processor and the compiler should find it in a fixed function one, because what decides the case is whether the machine must run binaries it was not compiled for and whether the delays it meets can be predicted in advance.

The case for the compiler: a compiler sees the whole program at once and may spend seconds on an analysis the hardware would have to complete in a fraction of a nanosecond. A very long instruction word machine takes that schedule as given, packs independent operations into fixed slots of one wide instruction, and drops the dependence checking altogether, saving area and power.

The case against the compiler: the schedule is only correct for the latencies the compiler assumed, so a binary built for one generation runs badly on the next or refuses to run at all. Worse, no compiler can know which loads will miss the cache, and a cache miss is the largest and least predictable delay a processor meets.

The case for the hardware: a superscalar processor decides at run time with the actual delays in front of it. Binary compatibility survives across generations because no schedule is baked into the code, and when a load misses the cache the machine simply runs whatever else is ready.

The case against the hardware: the checking grows roughly with the square of the issue width, since every candidate must be compared against every other, and all of that logic draws power in every cycle whether or not it finds anything worth issuing together.

The verdict and the condition that sets it: general purpose machines should leave the job to the hardware, which is what the market chose. Very long instruction word remains the better answer wherever the code is fixed, the latencies are known and the power budget is tight, which is signal processing and embedded control.

The names for the two approaches: these are the two forms of multiple issue, static where the compiler fixes what travels together and dynamic where the hardware settles it in the cycle.

Judge by the workload rather than by the technique, since fixed code with known timing favours the compiler and everything else favours the hardware.

The Master Key: *Give the work to the compiler when the code and the latencies are both fixed, and to the hardware whenever binaries must outlive the machine or cache misses cannot be foreseen.*

HOT Question 53

For a generation every performance question had one answer: a wider, faster single core. Midway through the 2000s that stopped working; the machines reaching the laboratory at Kongwa carried two cores, then four, at the same clock speed. Three costs had come due together.

Explain why processor manufacturers largely stopped increasing issue width in the mid 2000s and turned to multiple cores instead, and assess whether that was the right decision.

Model Answer

They stopped because the cost of widening a core rises far faster than the performance it returns, while duplicating a whole core is a linear cost for a potentially linear gain, and the decision was right for the hardware even as it moved the difficulty onto the people writing the software.

The checking cost grows about quadratically: every candidate must be compared against every other candidate in the same cycle, so the comparisons grow with the square of the width. At width two there is 1 pair

to check and at width four there are 6, and the rename ports, the wakeup logic and the register file ports all widen alongside. A four issue core is far more than twice a two issue core.

The returns fall away: the parallelism available inside one instruction stream is limited by the dependences the program actually contains, so each extra slot is filled less often than the slot before it. Achieved IPC climbs a little and then flattens, however wide the machine is built.

The clock could not be raised either: power was rising faster than performance, and the heat one fast core produced could no longer be carried away cheaply, so the other traditional route to speed closed at the same moment.

Why cores were the better bargain: a second core costs roughly the area of the first and can in principle deliver roughly the throughput of the first, which is a straight trade rather than a shrinking one. Each core also stays simple, and a simple core is easier to verify and easier to clock.

The case against the decision: the gain is only realised if the software is written as several cooperating threads, and writing correct threaded software is genuinely hard. Many everyday programs remain effectively single threaded, so they gain almost nothing from the extra cores they are running on.

The verdict: correct for the hardware, because no other route was open, but it bought a kind of performance the software industry has taken twenty years to learn how to spend. Single thread speed still decides how an interactive program feels, which is why designers kept improving it while adding cores, and Section 7 takes up what the software must now do.

The change was forced by physics and by economics rather than chosen freely, and it converted a hardware problem into a programming one.

The Master Key: *Widening a core costs about the square of the width for a shrinking return, so duplicating simple cores was the only bargain left and the bill was handed to the programmer.*

HOT Question 54

The tablets arrived at the secondary school in Liwale with one battery figure on the box and quite another once students had them open. The processor inside was working hard in cycles with nothing useful to do, and it was paid for in charge.

A team is designing a processor for a battery powered classroom tablet. Assess whether out of order execution should be included.

Model Answer

Include out of order execution but keep it narrow, at an issue width of about two, because cache misses on a small cache are frequent and unpredictable and no compiler can schedule around them, while the checking cost climbs much faster than the return.

What it buys on this machine: out of order execution hides the latency of a cache miss by running whatever else is ready while the miss is being serviced. A tablet has a small cache and runs a browser whose working set nobody can predict, so misses are frequent, and this is exactly the delay no static schedule can anticipate.

It asks nothing of the software: the benefit arrives without recompilation, so every application already written for the tablet gains from it on the day it is installed. That matters when the machine will run software the team did not write and cannot change.

What it costs on a battery: the reorder buffer, the rename table, the reservation stations and the wakeup logic are all clocked in every cycle, whether or not there is anything worth reordering. On mains power that is an area cost; on a battery it is a permanent drain paid for an occasional benefit.

The simpler alternative: the same silicon spent on an in-order core can buy a higher clock, a larger cache or a second core, and an in-order core spends far less energy per instruction because it is not maintaining a window of instructions in flight.

The verdict and its condition: include it, narrowly. The condition is the miss rate: where the cache is small and the workload is interactive and unpredictable, hiding misses is worth the standing power, but once the width passes two or three the checking cost outruns the parallelism the code actually contains.

What measurement would settle it: run the real workload on both designs and compare energy per task completed rather than peak performance. A core that finishes rendering the page sooner and returns to idle can use less charge overall than a slower core that never stops working.

On a battery the right measure is energy for the job done, and out of order execution earns its place only where the misses it hides are frequent.

The Master Key: *A narrow out of order engine is worth its standing power on a battery only where cache misses are frequent and unpredictable, and energy per task is the measurement that decides.*

Thread level parallelism: threads, simultaneous multithreading and multicore

This section leaves the single instruction stream behind and runs several streams at once. It separates the program from the process and the process from the thread, fixes exactly what threads of one process share, and then separates coarse grained, fine grained and simultaneous multithreading before separating simultaneous multithreading from a genuine multicore processor, which is the distinction most often got wrong. It traces a lost update on a shared counter, prices cache coherence and false sharing, and reaches a verdict on whether more cores always help. Section 8 takes the parallelism that lies in the data rather than in the streams.

BINDER Question 55

A laminated sheet of instructions hangs above the bench of the borehole repair store at Gairo. It has never repaired a pump. It becomes work when two fitters take it down, reaching into one tool tray while each keeps his own place on the sheet.

Distinguish a program, a process and a thread, and state what threads of the same process share and what they do not.

Model Answer

A program is a set of instructions stored on disk and doing nothing, a process is that program in execution together with everything the operating system gives it, and a thread is one flow of control inside a process, so one process may hold many threads.

The program is inert: the program is the file of instructions as it sits in storage, complete but not running. It consumes no processor time and owns no data. Copying it copies nothing that is executing.

The process is the program plus its world: a process is a program in execution. The operating system gives it its own address space, its own list of open files and its own entry in the process table, and it protects that space from every other process.

The thread is a flow of control: a thread is a single sequence of instructions being followed inside a process. Every process starts with one thread and may create more, and all of them run the same program text on the same data.

Shared and private, stated exactly: threads of one process share the code, the heap, the global data and the open files. Each thread keeps its own program counter, its own registers and its own stack, because each stands at a different point in the program and holds different local variables.

The consequence, which cuts both ways: because the heap is shared, one thread hands a large structure to another by passing an address instead of copying it, which is why threads communicate far more cheaply than processes. The same sharing lets two threads write one variable and destroy the work of the other.

Name the level before you state the property, because sharing the heap is true of threads and false of processes.

The Master Key: *A program is stored, a process is a program in execution with its own address space, and threads of one process share the code and the heap while each keeps its own program counter, registers and stack.*

BINDER Question 56

For a good part of every hour the press at the sunflower oil mill in Muleba stands waiting while one man fetches seed. A second man loading behind him costs nothing in press time. A processor with 4 issue slots stands waiting in the same way.

Explain why processors turned to executing several threads at once, and distinguish hardware multithreading from the multitasking an operating system performs.

Model Answer

Processors turned to several threads because one instruction stream runs out of independent instructions long before a wide machine runs out of slots, and hardware multithreading differs from operating system multitasking by keeping the state of several threads inside the processor, so a switch costs almost nothing.

The limit that forced it: within one instruction stream the supply of independent instructions is limited, so issue slots stay empty and execution units stand idle on a machine that is behaving perfectly. A second thread brings instructions that are independent of the first by construction, and they fit the empty slots.

What the operating system already does: multitasking is a software arrangement. The operating system runs one process for a few milliseconds, saves its registers to memory, loads another and resumes it, about 1000 times a second, which makes one core look like many. Nothing actually runs at the same instant.

What a switch costs each way: a software switch writes out one register set, reads in another and loses the cache contents of the old process, costing of the order of 500 cycles. Hardware multithreading duplicates the registers and the program counter per thread, so a switch costs 1 cycle or none, about 500 times cheaper.

The two are not alternatives: a machine with hardware multithreading still runs an operating system that multitasks, because there are always more runnable threads than the hardware can hold resident. The hardware makes the last step cheap; the software still decides who goes next.

What it does not do: filling empty slots raises throughput and leaves single thread latency alone, so a program with one thread gains nothing. How far the total can rise depends on how much of the work divides, which Section 9 makes exact.

Ask where the thread state is kept: in memory means the operating system is switching, in the processor means the hardware is.

The Master Key: *Hardware multithreading keeps several threads resident in duplicated registers so a switch costs about 1 cycle instead of about 500, while operating system multitasking swaps that state through memory.*

BINDER Question 57

It took the new hand at the livestock market in Longido a week to see 3 ways to work the weigh crush, not 2: finish one queue then the other; alternate, one from each; or, the crush having two gates, take one from each at once.

Describe coarse grained, fine grained and simultaneous multithreading, and state what distinguishes the last from the other two.

Model Answer

Coarse grained multithreading switches thread only when the running thread hits a long stall, fine grained multithreading switches thread every clock cycle in rotation, and simultaneous multithreading differs from both in kind rather than degree because it issues instructions from more than one thread within the same clock cycle.

Coarse grained: the processor runs one thread until it meets a stall long enough to be worth the change, typically a cache miss that goes out to main memory, and only then switches. It is the cheapest of the three to build and cannot hide a short stall, because the stall ends before a switch would pay.

Fine grained: the processor changes thread on every clock cycle, taking the ready threads in rotation, so a stall in one thread is covered by instructions from another. Short stalls disappear, but each thread advances only on the cycles allotted to it, so its own latency rises.

Simultaneous multithreading, the defining property: the other two place one thread in a cycle and merely choose which. Simultaneous multithreading issues instructions from two threads inside one cycle, which is possible only on a superscalar machine, because there must be slots a single thread was leaving empty.

Reading the four panels: each panel of Figure 2.15 covers 8 cycles at an issue width of 4, which is 32 issue slots. No row in the first three panels carries both shadings. Only the last mixes them, and that single visual difference is the whole distinction.

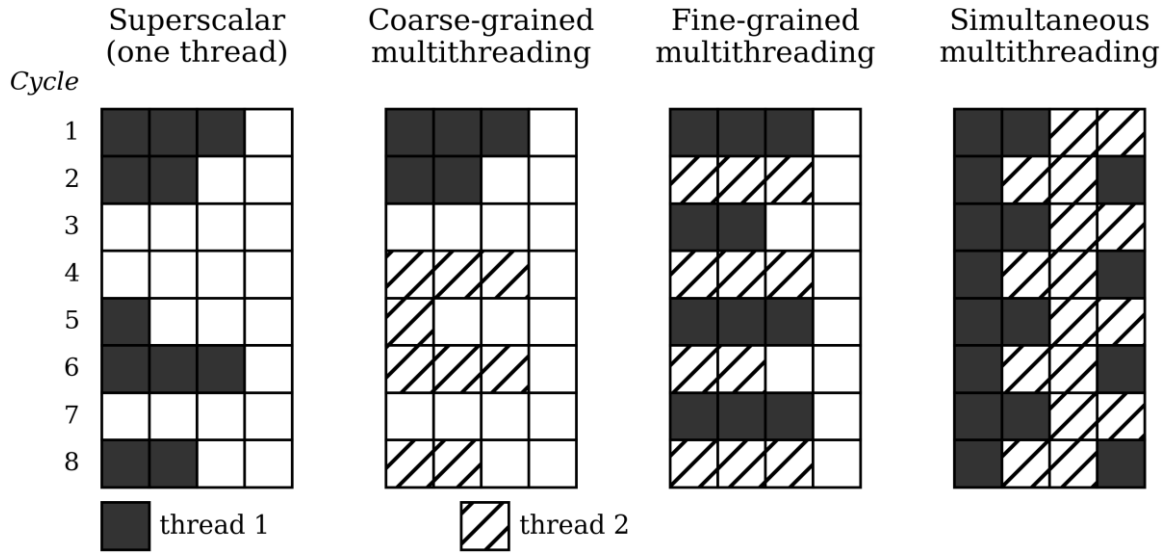


Figure 2.15: *Issue slots over eight cycles. Only simultaneous multithreading places instructions from two threads in the same cycle.*

The hardware each one needs: all three need a register set and a program counter per resident thread, so no state is saved on a switch. Simultaneous multithreading needs more: every instruction in flight carries a thread tag, so issue and completion keep the streams apart.

Say in the same clock cycle when you describe simultaneous multithreading, because that phrase is the whole difference.

The Master Key: *Coarse grained switches on a long stall and fine grained switches every cycle, but simultaneous multithreading does not switch at all: it issues from two threads inside one cycle.*

BINDER Question 58

A Form Six student in the computer laboratory at Ikungi opened the system monitor, counted 4 processor graphs on a machine the school bought as a dual core, and said the shop had cheated them. Two of those graphs were not what he took them for.

Distinguish simultaneous multithreading from a multicore processor, and explain what the operating system reports in each case.

Model Answer

A multicore processor holds several complete cores, each with its own fetch, decode, execution units and first level cache, while simultaneous multithreading gives one core two logical processors that share a single set of execution units, and the operating system counts logical processors in both cases and so cannot tell them apart.

What a multicore processor duplicates: every core is a complete processor with its own fetch and decode, its own execution units, its own registers and its own first level cache. Two threads on two cores execute on separate hardware, and neither slows the other except where they meet, at the shared outer cache and at memory.

What simultaneous multithreading duplicates: only the state. One core carries two register sets and two program counters and presents itself as two logical processors, but beneath them lie one set of execution units, one first level cache and one issue width, and the two logical processors compete for all of it.

Two logical processors are not two cores: state that plainly, because no specification sheet will. A dual core with simultaneous multithreading has 2 of everything that executes and 4 of everything that remembers where it is, while a quad core has 4 of both, and each reports 4 processors.

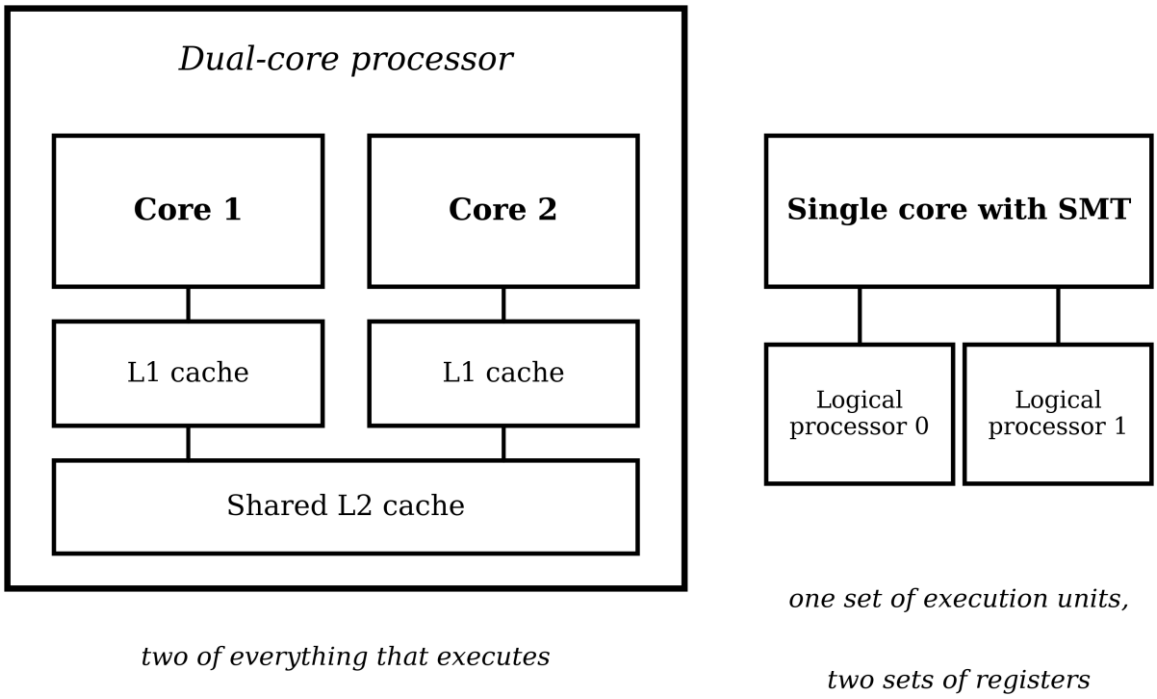


Figure 2.16: *Two physical cores beside one core with two logical processors. The operating system counts four processors, but only three of them can execute at the same moment.*

The size of the difference: run two compute bound threads on one core with simultaneous multithreading and the pair typically finishes about 20 to 30 per cent more work than one thread alone, not 100 per cent more, because they are dividing one set of units. A second real core gives close to 100 per cent.

Count the execution units, not the graphs in the system monitor.

The Master Key: *A core with simultaneous multithreading shows two logical processors over one set of execution units, so two threads gain about 20 to 30 per cent together where a second real core gives close to 100.*

REAL Question 59

Nothing was stolen, and the cashbook at the Maswa market was short. Two collectors each did what the book asks: read the running total, add your amount, write the new total. Both did it correctly, at once, and one collection had vanished from the page.

Explain what a race condition is, using two threads that each add one to a shared counter, and describe how a critical section prevents it.

The statement counter = counter + 1 looks like one operation. It is three.

First, read the value of counter from memory into a register.

Second, add one to the value now held in the register.

Third, write the register back to the memory holding counter.

Model Answer

A race condition is a defect in which the result of a program depends on the order in which two or more threads happen to interleave, and the shared counter shows it plainly, because the increment is three separate operations and two threads can both read the same old value and both write back the same new one, losing an update.

Why the counter is reachable at all: the counter is a global variable, and threads of one process share the global data and the heap. Both threads therefore hold the same address and neither has a copy of its own. The sharing that makes threads cheap is the same sharing that makes this defect possible.

The increment is three operations: read the counter into a register, add one to the register, write the register back. Between the read and the write the copy in memory is stale, and the thread has no way of noticing that somebody else changed it inside that window.

The lost update, traced: let the counter hold 10. Thread 1 reads 10. Before thread 1 writes, thread 2 also reads 10. Each adds one in its own register and each writes 11. Two increments were performed, the counter moved by 1, and it ends at 11 where it should be 12.

Time	Thread 1	Thread 2	counter in memory
1	read counter, gets 10		10
2		read counter, gets 10	10
3	add 1, register holds 11		10
4		add 1, register holds 11	10
5	write back 11		11
6		write back 11	11

Table 2.6: *Two threads each add one to a counter holding 10. Both read before either writes, so the counter ends at 11 instead of 12 and one increment is lost.*

The same fault at scale: let two threads each perform 1000 increments on a counter starting at 0. The correct total is 1000 plus 1000, which is 2000. Lose 107 updates and the answer is 1893, lose a different number on the next run and the answer changes again, so a test that passes once proves nothing.

$$\text{correct total} = 1000 + 1000 = 2000, \text{observed} = 2000 - 107 = 1893$$

The critical section and the lock: the critical section is the region of code that must not be occupied by two threads at once, here the three operations of the increment. A lock, also called a mutex, enforces it: a thread acquires the lock before entering, and any other thread that arrives waits until it is released.

What the lock costs: inside the critical section the threads run one at a time, so that region gains nothing from extra cores however many are fitted. The more of the running time the threads spend inside it, the less the extra cores buy, and Section 9 puts a figure on that limit.

Lock the smallest region that is genuinely shared, because everything inside the lock runs on one core no matter how many the machine has.

The Master Key: *An increment is read, add and write, so two unsynchronised threads can both read 10 and both write 11; a critical section guarded by a lock admits one thread at a time and the update is no longer lost.*

HOT Question 60

The same bar of soap went to two customers at different prices a minute apart, in a shop on the slopes at Rombo. The owner had raised the figure on the front counter list; at the back the old figure was still being read out.

Explain what cache coherence is, why it is needed in a multicore processor, and assess the cost it imposes.

Model Answer

Cache coherence is the guarantee that once one core has written a memory location, no other core can afterwards read a stale copy of it from its own cache, and it is needed because every core has a private first level cache in which the same location may sit at the same time.

How one location reaches two caches: a cache holds copies, and each core has its own first level cache. If two cores read the same shared variable, both caches now hold that line. Nothing is wrong yet, because the two copies agree with memory and with each other.

What a write breaks: core 1 writes its copy. Main memory may not have been updated, and the copy held by core 2 certainly has not, so core 2 reads a value that is no longer true anywhere else in the machine. The program is then wrong for a reason no line of its source explains.

What coherence does about it: the caches watch the traffic between themselves and memory. When one core writes a line, the hardware either invalidates every other copy, so the next reader must fetch the new value, or updates them in place. Invalidation is the usual choice, because a line written repeatedly and read rarely should not be broadcast.

The first cost, traffic that grows with cores: every write to a shared line must be announced to every other cache, so the messages grow with the core count. Coherence that is cheap across 4 cores is one of the reasons a machine of 64 cores is hard to build, and why large machines abandon a single shared bus.

The second cost, false sharing: coherence works on whole cache lines, typically 64 bytes, and not on variables. Two cores writing two different 8 byte variables that happen to lie in one line invalidate each other on every write and pay the full price of sharing while sharing nothing, and one line holds 8 such variables.

The judgement: coherence is not optional, because without it a shared memory program could not be trusted for a single instruction and no care in the source would repair that. It remains a real reason why doubling the cores does not double the speed, and a programmer can avoid the worst of it by padding the hot data of each thread onto its own cache line.

Coherence buys correctness with traffic, and false sharing is the case where the traffic is paid for and nothing is bought.

The Master Key: *Cache coherence makes one write visible to every other core by invalidating their copies, at a traffic cost that rises with the core count and is wasted entirely when two cores write different variables inside one 64 byte line.*

HOT Question 61

The box on the settings page was labelled worker threads and already held 8, the number the operating system had reported. The clerk at the district water office in Sikonge raised it to 16 to be safe, and the job took longer. The machine has 4 cores.

A processor has four physical cores, each supporting two threads by simultaneous multithreading. Assess how much work it can really do, and advise a programmer on how many threads to create.

Model Answer

The processor reports 8 logical processors but owns only 4 sets of execution units, so on compute bound work its real capacity is nearer 5 cores' worth than 8, and the right number of threads is 8 when the work waits often and 4 when it computes without pausing.

What the count means and does not mean: 4 cores each holding 2 threads gives 4 multiplied by 2, which is 8 logical processors, and the operating system will schedule work onto all 8. Behind them stand 4 sets of execution units. The count says how many threads may be resident, not how many can compute.

Eight compute bound threads: if every thread has work ready in every cycle, the partner thread on a core finds no empty slots to fill, and the pair gains only about 25 per cent over one thread. Four such pairs give 4 multiplied by 1.25, which is 5.0 cores' worth of throughput and not 8, so under a full compute load the machine behaves like five plain cores rather than eight.

Eight waiting threads: threads that miss in cache, wait on a disk or block on a network reply leave their execution units idle for long stretches, and those idle slots are exactly what the partner thread fills. Eight such threads can approach 8 threads' worth of useful progress on the same 4 cores, which is the case the design was built for.

The advice: create 8 threads when the work waits often and 4 when it computes continuously, and never go far beyond 8, because extra threads add switching and cache pressure without adding capacity. For mixed work start at the reported count, time the program, and halve the count if the time does not improve.

The honest ceiling: none of this matters if the program is largely serial, because the part that cannot be divided sets a limit that no thread count passes, and Section 9 gives that limit exactly. Measure the program itself rather than reading a verdict off the specification sheet.

Choose the thread count from the behaviour of the work rather than from the number the operating system prints, then confirm the choice with a stopwatch.

The Master Key: *Four cores with simultaneous multithreading report 8 processors over 4 sets of execution units, so compute bound work sees about 5 cores' worth and only work that waits justifies all 8 threads.*

HOT Question 62

Two cores gave 1.9 times the speed of one. Four gave 3.1. Eight gave 3.3, and the graph the students at the teacher training centre in Uvinza were drawing flattened along the top. Nothing in the program had changed, and no core had slowed down.

Assess the claim that adding more cores will always make a program run faster.

Model Answer

The claim is false, and it fails for four separate reasons that have nothing to do with the quality of the cores; more cores help only where the work divides into large independent pieces that write very little shared data.

The serial part: every program contains work that cannot be divided, such as reading the input file or writing the final answer, and that work takes as long on 64 cores as on 1. Shrink the divisible part far enough and the serial part is the whole running time, which Section 9 makes exact.

Communication and locks: threads that cooperate must exchange data and protect it, and a critical section runs one thread at a time by design. A program that spends a third of its time inside locks has handed a third of its work back to a single core, whatever the core count says.

The shared path to memory: cores share the outer cache, the memory controller and the road to main memory, and coherence messages add to that load. Threads that never touch the same variable still compete for the path, so every core can be busy and starved of data at the same time.

The cost of the threads themselves: creating and joining a thread costs of the order of 10000 cycles. Split a loop of 100 additions, which costs about 100 cycles, across 4 threads and you have paid 4 multiplied by 10000, or 40000 cycles, which is 400 times the cost of simply doing the work.

Where the claim does hold: work that divides into large independent pieces with little shared writing scales close to linearly over a useful range: rendering separate frames of an animation, serving separate web requests, marking separate examination scripts. Each piece touches its own data and finishes without asking anything of the others.

The verdict: more cores help when the work divides into large independent pieces with little shared writing, and not otherwise. The core count is never the first thing to optimise, because a better algorithm or a repaired cache miss usually returns more, and returns it on every core afterwards.

Ask what fraction of the work is genuinely independent before asking how many cores are available.

The Master Key: *More cores speed only the divisible part, and the serial section, the locks, the shared road to memory and the cost of the threads themselves are why the measured graph flattens long before the core count does.*

HOT Question 63

Both quotations came to the same price, and closed with the same line, four processors. That is what the head of computer studies at the secondary school in Namtumbo has to go on. Only one machine has four of the part that does the work.

A secondary school is choosing between a four core processor with no simultaneous multithreading and a two core processor with simultaneous multithreading, at the same price. The machines will run a browser, an office suite and a programming environment. Recommend one and justify the recommendation.

Machine A: 4 physical cores, no simultaneous multithreading. The operating system reports 4 processors.

Machine B: 2 physical cores, each with simultaneous multithreading. The operating system reports 4 processors.

The two machines cost the same, and both will run a browser, an office suite and a programming environment.

Model Answer

Buy machine A, the four core processor, because it holds 4 complete sets of execution units against 2, which is twice the hardware that actually computes, and the matching figure of 4 processors on the two sheets hides precisely that difference.

Count what executes: machine A computes on 4 instruction streams at full speed at once. Machine B holds 4 but computes on 2 sets of units, and the partner thread on each core adds about 25 per cent, so its useful capacity is 2 multiplied by 1.25, which is 2.5 cores' worth. Machine A does about 4 divided by 2.5, or 1.6 times as much, under load.

Why the sheet flatters machine B: both machines report 4 processors, so a buyer comparing those two lines sees no difference at all. The reported figure counts logical processors, and machine B reaches 4 by duplicating registers, which is cheap, rather than execution units, which are not.

The workload that decides it: a browser, an office suite and an editor are idle most of the day, and for that part either machine is ample. The moment that matters is the build, because a compiler divides cleanly and will use every core it is given, and there machine A is materially faster.

The honest concession: machine B may carry a higher clock, since two cores are easier to cool than four, and on lightly threaded interactive work, which is most of a school day, that higher clock may make it feel quicker. Simultaneous multithreading also copes well with a machine full of small waiting tasks.

The measurement that settles it: trust neither sheet. Compile a project of known size on each machine, or open the browser, the office suite and the programming environment together and time a realistic sequence of work. A stopwatch on the actual workload answers what a count of processors cannot.

What would reverse the verdict: recommend machine B instead if it runs materially faster per core and the school can show that its real work never keeps more than two threads busy at once. Neither condition is met here, so machine A stands.

Buy the execution units rather than the number printed on the sheet.

The Master Key: *Four real cores beat two cores with simultaneous multithreading at the same price, because 4 sets of execution units against 2 is about 1.6 times the capacity even after crediting the partner threads.*

Data level parallelism: SIMD lanes, vector width and SIMT

This section takes the parallelism that lies in the data rather than in the instruction stream. It defines data level parallelism, opens a SIMD unit to show what a lane is, and then supplies the number that is usually left out: the width of the register, which is what turns a drawing of four lanes into a calculation. It counts elements per register at 128, 256 and 512 bits, vectorises a loop, disposes of the remainder left when the count does not divide, and goes on to SIMT and the graphics processor before a district office decides whether wider registers are worth the money.

BINDER Question 64

Add these two columns of rainfall readings, said the note pinned above the desk at the water office in Namtumbo, and the clerk began at the top. There was no reason to begin at the top. No reading depended on the one before it.

Define data level parallelism and explain how it differs from dividing a task among threads.

Model Answer

Data level parallelism is applying the same operation to many data elements at the same time, so the parallelism lies in the data rather than in the control flow: there is one instruction stream, and no element has to wait for, signal to or synchronise with any other.

Where the division falls: thread level parallelism divides the task into pieces that must then be coordinated, while data level parallelism divides the data and drives every piece of it with one shared stream of instructions. Because each element is treated identically and touches nothing another element touches, there is nothing to coordinate.

Three workloads whose data divides itself: adding two arrays of measurements element by element, raising the brightness of every pixel in an image by the same amount, and applying the same interest calculation to every row of a ledger. In each of the three the operation is fixed and only the operands change, which is exactly the shape this kind of parallelism wants.

The property that decides whether it applies at all: every element must be independent of every other. A running total across the elements destroys that at once, because no element can be added in until the element before it has been added in, and a chain of that kind is worked one link at a time however many idle lanes are standing by.

Why it is the cheapest parallelism to build: dividing the data asks for no second program counter and no coordination hardware, only wider arithmetic driven by the instruction stream already there. That is why a processor can offer it for a fraction of what another core costs, and why nearly every processor sold carries it whether the programmer uses it or not.

Ask whether the elements are independent before anything else, because one running total in the loop settles the question before any hardware is considered.

The Master Key: *Data level parallelism applies one operation to many independent elements at once, and the moment a running total links the elements it does not apply.*

BINDER Question 65

The four grinding plates at the maize mill in Sikonge turn on one shaft, and the miller sets the gap for all of them. Four sacks are ground in the time one took, off a single setting. A processor performs the same trick with arithmetic.

Explain what a SIMD lane is and describe how one instruction produces several results.

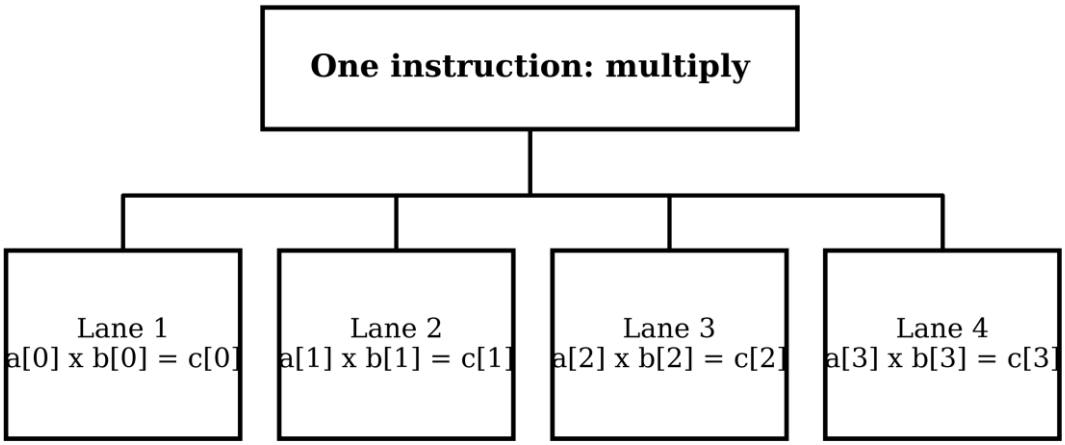
Model Answer

A SIMD lane is one of several identical arithmetic paths inside the processor, each holding its own operands and each producing its own result, so a single instruction fetched and decoded once yields as many results as there are lanes.

Paid once, performed many times: the instruction is fetched, decoded and issued a single time. Its operands sit in a wide register divided into slots, one slot for each lane, and in the same cycle every lane carries out that one operation on the slot it owns. Four lanes therefore return four results for one instruction.

The lanes hold different data, and that is the whole point: the words single instruction, multiple data name the two halves of the arrangement: the instruction is shared, the data are not. Figure 2.17 shows one lane computing one pair of array elements, the next lane a different pair, and so on across the unit. Lanes handed the same value would compute four copies of one answer and gain nothing.

one shared instruction, four different data values, four results in the same cycle



this is what Multiple Data means: the lanes never see the same operand

Figure 2.17: *One instruction driving four lanes. Each lane holds a different pair of operands, which is what the words multiple data mean.*

What is actually saved: the fetch, the decode and one round of loop control are paid for four results instead of four times over, so the instruction count for the same work falls to a quarter of what it was. Fetching and decoding account for much of what an instruction costs in energy, so the energy spent per result falls with it.

What a lane is not: a lane has no program counter and no control of its own, so it cannot branch away from what the shared instruction says or stop early. That restriction is the price of the saving, and it is the one a graphics processor later relaxes.

One instruction, several lanes, different operands in each: take away the different operands and the whole arrangement returns nothing.

The Master Key: *A lane is an arithmetic path with its own operands, and one instruction fetched once drives every lane in the same cycle, so four lanes return four results for the price of one fetch.*

BINDER Question 66

The same register held four numbers in the morning and sixteen in the afternoon, and nothing about the hardware had changed. The surveyor at the land office in Maswa had moved from 32 bit numbers to single bytes, and the register cut itself into more slots.

Explain what a vector register is, and calculate how many elements of a given type one holds.

Model Answer

A vector register is one wide register holding several data elements side by side in equal fixed slots, and the number it holds is its width in bits divided by the width of one element in bits.

The division, done four ways on one register: a 128 bit register divided by 32 bit single precision numbers holds 4 of them; divided by 64 bit double precision numbers it holds 2; divided by 16 bit values it holds 8; and divided by single bytes of 8 bits it holds 16. The register never changes width. Only the slot changes, and the count follows.

$$\text{elements per register} = \text{register width in bits} / \text{element width in bits}$$

Wider registers, counted the same way: 256 divided by 32 is 8 single precision numbers, and 512 divided by 32 is 16. Doubling the width doubles the elements one instruction handles and therefore doubles the results it produces, provided the memory system can deliver data fast enough to keep the lanes fed.

The extension families, by width rather than by name: on one widespread family of processors there is a 128 bit generation called SSE, a 256 bit generation called AVX and a 512 bit generation called AVX-512, and other families carry equivalents of their own. What is worth remembering is the width attached to each, because the width is the part you can calculate with.

Why the width is the number that matters: students are often shown four lanes drawn on a page with no width printed anywhere, and from such a drawing no calculation is possible at all. Give the width and everything follows: elements per register, the vector operations a loop needs, and the lanes wasted at the end of it.

Divide the register width by the element width and the lane count stops being something to memorise.

The Master Key: *Elements per register is register width divided by element width, so 128 bits holds four single precision numbers, 256 holds eight and 512 holds sixteen.*

BINDER Question 67

The loop looked simple enough, and the compiler refused to touch it. Nothing was wrong with the arithmetic. The programmer at the water board office in Muleba had passed in two arrays that, for all the compiler could tell, might be the same array.

Describe what vectorisation means, and state what prevents a loop from being vectorised.

Model Answer

Vectorisation is rewriting a loop that processes one element per iteration into a loop that processes a whole vector per iteration, by hand or by the compiler, so that the same arithmetic is performed with a fraction of the instructions.

The transformation, in numbers: a loop running 1024 times on single elements becomes a loop running 256 times on vectors of four, because 1024 divided by 4 is 256. Not one addition or multiplication has been removed; what has gone is three quarters of the counting, the branching and the instruction issue.

Dependence and branching, the two obstacles inside the loop: if each iteration needs the result the iteration before it produced, the elements form a chain and no vector can be formed at all. If a conditional sends different elements down different branches, the loop can still be vectorised, but only by masking: computing both sides in every lane and discarding what that lane does not want.

Memory and the leftover count, the two obstacles outside it: if the elements a vector needs are scattered through memory instead of lying next to each other, or if the array does not begin on the boundary a wide load

expects, one instruction cannot collect them cheaply. And if the loop count is not a multiple of the vector width, a remainder is left that must be handled on its own.

Why a compiler refuses: a compiler must prove independence rather than suspect it, and two array parameters that might refer to overlapping memory are enough to stop it, since it cannot rule out one element overwriting what a later element reads. It then emits correct scalar code and says nothing, so the loss is invisible unless it is looked for.

Vectorisation is a rewrite of the loop rather than a faster instruction, and it happens only where independence can be proved.

The Master Key: *Vectorisation turns 1024 single element iterations into 256 iterations on vectors of four, and it fails on dependence, on divergent branches, on scattered memory and on a count that will not divide.*

REAL Question 68

One thousand and two. Not one thousand, which divides cleanly into vectors of four, but one thousand and two, which does not. The clerk at the district office in Longido would never see the difference; the machine pays for it in idle lanes.

A program applies the same operation to 1000 single precision numbers on a machine with 128 bit vector registers. Calculate the number of vector operations required, the number of elements wasted, and the speed-up over scalar code if each operation takes one cycle.

Model Answer

Four elements fit in each register, so the 1000 element job needs 250 vector operations with nothing left over and runs 4.0 times faster than scalar code; change the count to 1002 and it needs 251 vector operations, wastes 2 lanes in the last of them, and the speed-up falls to 3.992.

Elements per register first: the registers are 128 bits wide and a single precision number is 32 bits, so each register holds 128 divided by 32, which is 4 elements. Every number that follows rests on this one, which is why a drawing of four lanes with no width printed on it cannot be calculated from.

The clean case, 1000 elements: 1000 divided by 4 is exactly 250, so 250 vector operations cover the array with no remainder whatever. Scalar code performs 1000 operations at 1 cycle each, which is 1000 cycles, and the vector code performs 250, so the speed-up is 1000 divided by 250, which is 4.0. That is the full width of the machine and the most that can be had.

The awkward case, 1002 elements: 1002 divided by 4 is 250.5, so 251 vector operations are needed and the last carries only 2 useful elements. The machine pushes 251 multiplied by 4, which is 1004 element slots, through the lanes for the 1002 that are wanted, so 2 slots do no useful work, a waste of 0.199 per cent. The speed-up is 1002 divided by 251, which is 3.992.

$$\text{vector operations} = \text{ceiling of } (n / \text{lanes})$$

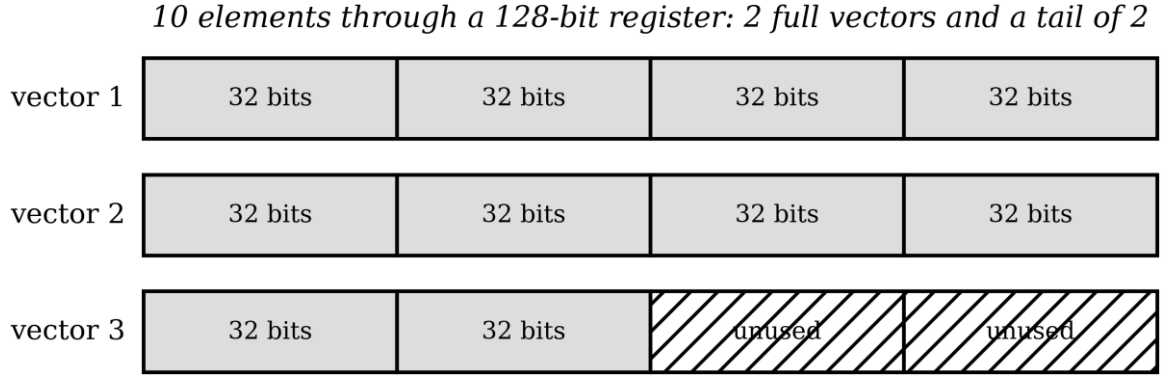


Figure 2.18: *Ten elements pushed through a 128 bit register. Two vectors are full and the third carries a tail of two, so two lanes do no useful work.*

Why the figure uses ten and not a thousand: Figure 2.18 sends 10 elements through the same register: 2 full vectors and a tail of 2, which is 3 vector operations covering 12 slots for 10 wanted elements. That is 2 slots wasted out of 12, or 16.7 per cent. The identical tail costs almost nothing on a long array and a great deal on a short one.

The general rule: the number of vector operations is n divided by the lane count, rounded up. The remainder is n modulo the lane count, and where that remainder is not zero the final operation carries only that many useful elements, leaving the lane count minus the remainder idle. A compiler cleans the tail up with a short scalar loop after the vector loop.

Round the division up, then ask what the last vector is carrying, because that is where the wasted lanes are.

The Master Key: *Vector operations are n divided by lanes rounded up, so 1002 single precision elements through a 128 bit register need 251 operations, waste 2 lanes and return 3.992 rather than 4.0.*

REAL Question 69

A message reached every phone in the class group at Mkalama, three sentences forwarded from somebody else's notes and never checked. All three were stated with the same confidence, and only one of them is true. The two that fail do so in different ways.

Three statements about data level parallelism are given below. Identify the one that is correct, and explain precisely what is wrong with each of the other two.

Statement A: Widening a vector register from 128 bits to 256 bits doubles the number of single precision elements processed per instruction, from four to eight.

Statement B: In a four lane SIMD unit, all four lanes receive the same data value and the same instruction.

Statement C: Threads on a graphics processor are distinguished from one another by having separate stack pointers.

Model Answer

Statement A is the correct one; Statement B contradicts the meaning of the term it is using, and Statement C attaches a real mechanism to the wrong piece of hardware.

Why A is right: the element count is the register width divided by the element width, so a 128 bit register holds 128 divided by 32, which is 4 single precision numbers, and a 256 bit register holds 256 divided by 32, which is 8. Doubling the width doubles the count, and one instruction then returns 8 results where it returned 4.

The one condition A leaves unstated: the doubling is delivered only where the memory system can supply twice as many bytes per cycle. The register widens on its own; the path that feeds it does not widen with it.

Why B is wrong: one instruction is shared across the lanes, but the data are not, and each lane holds a different element. Four lanes handed the same value would compute four copies of one answer, which is the work of a single lane repeated 3 times for nothing. Sharing the instruction is the saving; sharing the data would remove the reason for having lanes at all.

Why C is wrong: threads on a graphics processor are distinguished by a per thread identifier and by their own program counter, not by separate stack pointers. The identifier is the working part of the arrangement: it indexes each thread into its own portion of the array, so every thread reads and writes the element matching its own number.

What C hides: when threads in one group take different branches the hardware answers with predication and reconvergence, running both paths with the threads not on the current path switched off and then bringing them back together. No stack is involved, and a candidate who believes one is will look for the cost of divergence in the wrong place.

Two questions dispose of all three: is it the instruction that is shared or the data, and what actually tells one thread from another.

The Master Key: *Lanes share the instruction and never the data, doubling the register width doubles the elements, and threads on a graphics processor are told apart by an identifier and a program counter rather than by a stack pointer.*

HOT Question 70

The graphics card came out of its box at the computer laboratory in Turiani and the teacher expected everything on the machine to run faster. The image processing did, by a very wide margin. The attendance spreadsheet did not move at all.

Explain what SIMT is, how it differs from SIMD, and assess when a graphics processor is the right machine for a calculation.

Model Answer

SIMT, single instruction multiple threads, keeps the shared instruction stream of a SIMD unit but presents each lane as a thread with an identifier of its own, so the programmer writes the code for one element and the hardware runs thousands of copies of it at once.

What the programmer actually writes: one short piece of code describing what is to be done to one element, with the thread identifier standing in for the index. How many copies to run is given when the work is launched rather than written into a loop, so the same source serves a small array and a very large one.

The difference from SIMD that matters: a SIMD lane has no control flow of its own and cannot branch. A SIMT thread may branch, and when threads in one group branch different ways the hardware runs both paths in turn with the inactive threads switched off. Each thread still follows only one path, but the group pays for 2, so divergence costs real time.

Where the group boundary lies: threads are scheduled in fixed groups, commonly 32 of them, and divergence costs nothing unless it happens inside a group. If every thread in a group takes the same branch nothing is lost; if they split, the group takes as long as both paths together.

When a graphics processor is the right machine: when the work holds thousands of independent elements, has high arithmetic intensity, meaning many operations performed for each byte fetched, and branches little. Image processing, matrix arithmetic and physical simulation all have that shape, and on them such a machine returns many times what an ordinary processor can.

When it is the wrong machine: when the work branches heavily, so groups diverge and both paths are paid for; when the data set is small, so thousands of threads cannot be filled; and above all when data must be copied across the link to the card and the results copied back for every small piece of work.

The verdict and the deciding condition: use a graphics processor where the arithmetic saved on the card exceeds the time spent moving the data there and back. That transfer is the condition. A long calculation on data copied over once repays it easily, while a short calculation on data that crosses repeatedly finishes sooner if it never leaves the processor.

Count the transfers before counting the cores, because on a short job the copying is the calculation.

The Master Key: *SIMT gives every lane a thread identifier and lets it branch, at the cost of divergence, and a graphics processor is worth using only when the arithmetic saved exceeds the cost of moving the data across and back.*

HOT Question 71

The hand written vector code was fast. It was written against one register width, so when the next generation arrived, twice as wide, every loop had to be rewritten. The programmer at the cashew plant in Ruangwa must decide whether to pay that bill twice.

Assess whether a program should be vectorised by hand or left to the compiler.

Model Answer

Leave it to the compiler for almost every loop and vectorise by hand only in the few loops that measurement shows dominate the running time, because hand written vector code buys speed at the price of tying the program to one instruction set and one register width.

What the compiler does well: it costs nothing, it is applied to every loop it can prove safe, and it is redone from scratch at every compilation, so an edit made a year later does not silently lose the vectorisation. On straightforward loops with simple bounds and contiguous access a modern compiler vectorises reliably.

Where the compiler stops: it must prove independence rather than assume it, and two pointers that might refer to overlapping arrays are enough to make it refuse. So is a loop whose parallelism is hidden behind a function it cannot see into. It then emits correct scalar code without complaint, so nothing announces the loss.

What hand vectorisation buys: the programmer knows things the compiler cannot prove, most often that two arrays never overlap, and can act on that knowledge directly. More valuable still, the programmer can restructure the data so that the elements one instruction needs lie next to each other in memory, which often matters more than the instructions chosen.

What hand vectorisation costs: the loop is written against one register width, so when the 256 bit generation arrives it must be written again, and again for the 512 bit generation. It is much harder to read and much harder to test, and a mistake in it produces wrong numbers rather than a message from the compiler.

The verdict and its condition: write the loop so the compiler can vectorise it, which in practice means simple bounds, contiguous access and an assurance that the arrays do not overlap, and then measure. Reach for hand vectorisation only where measurement shows a loop dominating the running time; below that the cost of maintaining it exceeds the speed returned.

What to check before blaming the compiler: ask for the report it can produce on which loops it vectorised and why it declined the rest. Most refusals have a cause the programmer can remove in a line, and removing it is far cheaper than writing vector code by hand.

The best vector code in most programs is scalar source written so plainly that the compiler cannot refuse it.

The Master Key: *Write loops the compiler can prove safe and let it vectorise them, and hand vectorise only the measured few, because hand written vectors must be written again at every new register width.*

HOT Question 72

One line on the supplier's quotation mattered: vector registers of 512 bits in place of 128, for most of the year's equipment budget. The officer at the district education office in Mkinga wants to know what the extra bits will do to the results table.

A district office processes a large table of school examination results. Assess whether widening the processor's vector registers from 128 bits to 512 bits would be a good use of money.

Model Answer

Not for this job: widening from 128 bits to 512 bits raises the single precision count from 4 elements to 16, which is 4 times the arithmetic per instruction, but a job that reads records, compares text and writes output is held up by memory and by input and output, and none of those is made faster by a wider arithmetic register.

The gain, counted honestly: 128 divided by 32 is 4 elements per instruction and 512 divided by 32 is 16, so 16 divided by 4 gives 4 times as many results from each instruction. If arithmetic on long runs of numbers were the whole job, the affected work could run up to 4 times faster.

Ask what the job spends its time on: reading examination records from storage, comparing candidate names as text, sorting, matching subject codes and writing a report. Those are dominated by waiting on storage and memory and by branching that goes a different way on almost every record. Adding lanes lengthens a queue that was never the one holding the job up.

The second limit, named: memory bandwidth. 16 lanes need 4 times as many bytes per cycle as 4 lanes need, so unless the memory system delivers at that rate the wide unit waits with all 16 lanes idle. A register can be widened by buying a new processor; the path to memory cannot be widened so cheaply.

What the width would genuinely help: the parts that apply one operation to a long contiguous run of numbers: totalling every candidate's marks, standardising every score against a mean, or a statistical pass over the whole entry. Those parts are real and they will speed up, and what decides the purchase is the fraction of the running time they occupy, which is the calculation Section 9 sets out.

The verdict: do not buy it for record processing. Buy it where the work is numerical, the arrays are long and contiguous and the memory system can already feed the lanes. For this office the same money spent on faster storage or on more memory returns more.

The measurement that would decide: run the existing job and find whether the processor is waiting on memory or on arithmetic. If it stalls on memory most of the time the wider registers buy nothing; if it is busy computing and the loops are long and regular, the width is worth paying for.

A wider register is worth buying only when the arithmetic is the queue and the memory can keep the lanes fed.

The Master Key: *Widening 128 bits to 512 bits quadruples the elements per instruction from four to sixteen, and it buys nothing unless the job is arithmetic bound and the memory can deliver four times the data.*

Amdahl's law and the limits of parallel speed-up

This section puts a number on the limit that Sections 6, 7 and 8 kept pointing towards. It states Amdahl's law, defines every term in it, derives the ceiling that survives when the processor count grows without bound, and draws that ceiling as a graph. It then costs a 100 second job on 2, 4, 8 and 16 processors, decides between buying hardware and rewriting software, advises a data office on a 32 processor machine, and closes the chapter by putting every lever a designer has into an order of priority. A program described as parallel is not thereby fast, and the fraction that stays serial is what says how fast it can ever become.

BINDER Question 73

Four grinding machines now stand where one stood at the maize mill in Sikonge, and the queue is shorter, but nothing like four times shorter. The weighing, the ticket and the payment still pass through one counter and one clerk.

State Amdahl's law, define every term in it, and explain what it measures.

Model Answer

Amdahl's law states that if a fraction p of a program's running time can be made parallel while the remaining fraction 1 minus p cannot, then running the parallel part on N processors gives an overall speed-up of S equal to one divided by the quantity one minus p plus p over N.

$$S = 1 / ((1 - p) + p/N)$$

What p actually is: p is the proportion of the original running time spent in work that can be divided among processors, not the proportion of the code. A short routine may hold most of the running time, so p is measured in seconds from a profile and never estimated by reading the listing. That confusion is the commonest and most expensive error in this topic.

What N and S stand for: N is the number of processors the divisible work is spread across, and the law assumes the sharing is perfectly even. S is the ratio of the original running time to the improved running time, so an S of 2 means the job finishes in half the time.

What the law measures: It measures the whole-program effect of speeding up one part of it. The denominator holds the two pieces of the original time side by side: the serial piece, which no processor count touches, and the parallel piece divided by N.

One worked instance: Take p as 0.8 and N as 4. The serial term is one minus 0.8, which is 0.2, and the parallel term is 0.8 divided by 4, which is 0.2 as well. Their sum is 0.4, and one divided by 0.4 is 2.5. Four processors on a program four fifths parallel return two and a half times, not four.

Read p from a profile of the running time and never from the length of the code, because everything the law says afterwards rests on that single number.

The Master Key: *Amdahl's law is S equals one over the quantity one minus p plus p over N, where p is the share of the original running time that can be divided and N is the processor count.*

BINDER Question 74

A tender document at the district office in Ruangwa asked for a machine sixteen times faster and specified sixteen processors. The program it was written for spends a tenth of its time in work that cannot be shared. No supplier could have met that tender.

Derive the ceiling on speed-up as the number of processors grows without limit, and explain what it means in practice.

Model Answer

As N grows without limit the term p over N falls towards zero, so S tends to one divided by one minus p, and that value is a hard ceiling fixed by the serial fraction alone.

The derivation in one step: N appears in the formula only inside p over N, and p itself is fixed. Doubling N halves that term, doubling again halves it once more, and it approaches zero, while the serial term one minus p does not move at all. The denominator therefore approaches one minus p.

$$\text{ceiling of } S = 1 / (1 - p)$$

Five ceilings, and how fast the wall arrives: A parallel fraction of 0.50 stops at 2 and 0.75 stops at 4. Even 0.90 stops at 10, 0.95 stops at 20, and only 0.99 reaches 100. The ceiling climbs slowly until p is extremely close to one, then climbs violently, which is why the last few per cent of serial work matter more than all the rest.

The consequence, stated bluntly: A program that is 10 per cent serial can never run more than ten times faster, not on a hundred processors and not on a million. A hundred processors take it to 9.17 and a million take it to 9.9999, and no machine ever built or costed takes it past 10.

What a programmer should attack first: The serial fraction sets the ceiling and the processor count only decides how near the ceiling you come, so the serial part is the target. Cutting it from 10 per cent to 5 per cent raises the ceiling from 10 to 20, a factor of 2 bought with no hardware at all.

Ask what fraction of the time is serial before asking how many processors are on offer, because the first number caps everything the second can do.

The Master Key: *The ceiling on speed-up is one divided by the serial fraction, so code that is 10 per cent serial stops at ten times however many processors are bought.*

BINDER Question 75

The graph went up on the board at the teachers' college in Muleba, and the first question from the floor was why every line went flat. One student had expected the topmost curve to keep climbing, since its program was ninety five per cent parallel.

Using the graph provided, describe the shape of the speed-up curve and explain why adding processors gives less and less.

Model Answer

Every curve starts at a speed-up of 1 on one processor, rises steeply while the processor count is small, then bends over and flattens against a ceiling that the serial fraction alone decides.

Where every curve must start: Put N as 1 and the denominator becomes one minus p plus p, which is exactly 1, so S is 1. Every curve passes through that same point whatever its parallel fraction, because on one processor nothing has been divided yet.

Where each curve stops: The higher the parallel fraction, the later the bend and the higher the ceiling, but every curve bends. The four dashed lines in Figure 2.19 sit at 2, 4, 10 and 20 for parallel fractions of 0.50, 0.75, 0.90 and 0.95, and no curve ever reaches the line above it.

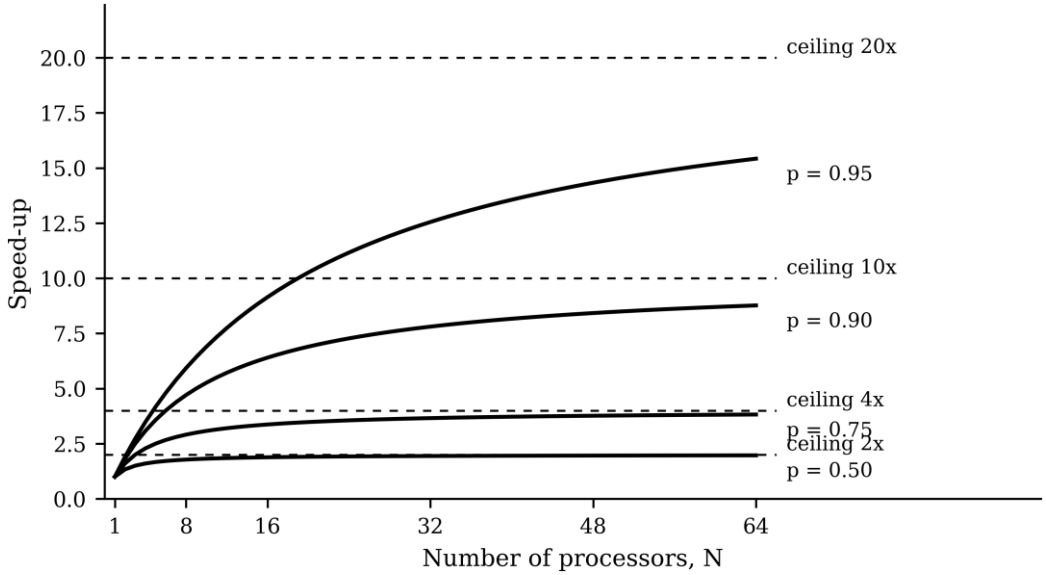


Figure 2.19: *Speed-up against processor count for four parallel fractions. Every curve flattens against a ceiling set by the serial part alone.*

Why the return keeps shrinking: Each time N doubles, the parallel term p over N halves, while the serial term one minus p is untouched. The serial term therefore comes to dominate the denominator, and every further processor is dividing an ever smaller share of the original time.

Two readings at a parallel fraction of 0.90: Eight processors give one divided by the sum of 0.1 and 0.1125, which is 4.71, already less than half of the ceiling of 10. Sixty four processors, eight times as many again, give only 8.77, a further factor of 1.86 for eight times the hardware.

The bend is not a fault in the machine; it is the serial part quietly becoming the whole of what is left to shorten.

The Master Key: *Doubling the processor count halves only the parallel term while the serial term stands still, so every speed-up curve bends and flattens against one divided by one minus p.*

REAL Question 76

One hundred seconds, of which seventy can be shared out and thirty cannot: that is the profile of the stock reconciliation program at a wholesaler in Kilindi. The owner has been offered processors in twos, fours, eights and sixteens, each dearer than the one before.

A program takes 100 seconds. 70 seconds of that is work that can be divided among processors and 30 seconds cannot. Calculate the running time and the speed-up on 2, 4, 8 and 16 processors, and state the shortest time the program can ever achieve.

Total running time on one processor: 100 seconds.

Work that can be divided among processors: 70 seconds.

Work that cannot be divided at all: 30 seconds.

Model Answer

The program takes 65, 47.5, 38.75 and 34.375 seconds on 2, 4, 8 and 16 processors, which are speed-ups of 1.538, 2.105, 2.581 and 2.909, and it can never finish in less than 30 seconds however many processors are bought.

The parallel fraction, read off the profile: The divisible work is 70 seconds out of 100, so p is 0.7 and the serial fraction is 0.3. Both come straight from the profile in seconds, which is the only honest way to obtain them, and everything below follows from those two numbers.

How each row is built: Divide the 70 seconds by the processor count and add the untouched 30 seconds back. On 2 processors that is 70 divided by 2, which is 35, plus 30, giving 65 seconds, and the speed-up is 100 divided by 65, which is 1.538. On 4 processors it is 17.5 plus 30, which is 47.5 seconds and a speed-up of 2.105, and the remaining rows are built the same way.

Processors, N	Divisible part (s)	Total time (s)	Speed-up
1	70	100	1.000
2	35	65	1.538
4	17.5	47.5	2.105
8	8.75	38.75	2.581
16	4.375	34.375	2.909
very large N	approaches 0	approaches 30	3.333

Table 2.7: *The same 100 second program on five machine sizes. The 30 serial seconds appear unchanged in every row, and they are what stops the last column growing.*

$$S = 100 / (30 + 70/16) = 2.909$$

What the table shows: Going from 1 processor to 2 saves 35 seconds. Going from 8 to 16 saves 4.375 seconds, and it costs eight further processors to do it. The saving per processor added collapses even though every processor is exactly as fast as the first one.

The floor and the ceiling: The 30 seconds are present in every row and no processor count removes them, so the running time approaches 30 seconds from above and never reaches it. The ceiling on speed-up is therefore 100 divided by 30, which is 3.333, and that is one divided by one minus 0.7.

What that means for the purchase: Sixteen processors have already delivered 2.909 of a possible 3.333, so a machine four times larger takes the job only to 31.094 seconds. The remaining money belongs somewhere other than in processors.

What every figure here assumes: The 70 seconds are taken to divide perfectly evenly with nothing spent on splitting the work up or gathering the results afterwards. A real machine spends something on both, so each running time in the table is the best case and every speed-up in it is an upper bound.

Divide the divisible seconds, add back the seconds that cannot be divided, and the sum gives both the new running time and the floor the program can never break.

The Master Key: *Seventy divisible seconds and thirty stubborn ones give 65, 47.5, 38.75 and 34.375 seconds on 2, 4, 8 and 16 processors, against a floor of 30 seconds and a ceiling of 3.333.*

REAL Question 77

Two quotations lay side by side on the desk at the printing shop in Gairo. One was for a great many more processors; the other was for a programmer's time and no hardware whatever. The cheaper of the two turned out to be the faster.

A team can either parallelise a program that is currently 60 per cent parallel so that it becomes 80 per cent parallel, or buy four times as many processors, going from 4 to 16. The program currently runs on 4 processors. Determine which change gives the greater speed-up.

Present state: parallel fraction 0.6, running on 4 processors.

Option one: raise the parallel fraction to 0.8 and keep 4 processors.

Option two: keep the parallel fraction at 0.6 and raise the processor count to 16.

Model Answer

Improving the program wins: raising the parallel fraction to 0.8 on the same 4 processors gives a speed-up of 2.5, while leaving the program alone and buying 16 processors gives only 2.286, both measured against the present 1.818.

The baseline as it stands: With p as 0.6 and N as 4, the serial term is 0.4 and the parallel term is 0.6 divided by 4, which is 0.15. The denominator is 0.55 and the speed-up is one divided by 0.55, which is 1.818.

Option one, rewrite the program: With p raised to 0.8 and N still 4, the serial term falls to 0.2 and the parallel term becomes 0.8 divided by 4, which is 0.2. The denominator is 0.4 and the speed-up is 2.5, a gain of 0.682 over the baseline and no hardware bought.

Option two, quadruple the hardware: With p still 0.6 and N raised to 16, the serial term is unchanged at 0.4 and the parallel term becomes 0.6 divided by 16, which is 0.0375. The denominator is 0.4375 and the speed-up is 2.286, a gain of only 0.468 for twelve extra processors.

$$1/((1-0.8)+0.8/4) = 2.5 \text{ and } 1/((1-0.6)+0.6/16) = 2.286$$

Why the software change wins: The processor count acts only on p over N, which is already the smaller term and is shrinking on its own. The parallel fraction acts on one minus p, which is the term that sets the ceiling. Buying processors can at best remove the whole 0.15 from the denominator, which is all that any number of them could ever do, while the rewrite removed 0.2 from a term no purchase whatever can touch.

The caveat the arithmetic does not carry: Rewriting costs engineering weeks and can introduce faults, and some work simply cannot be divided however long the team stares at it, whereas processors arrive in a crate and work on the first day. The comparison is therefore not purely arithmetical, and the verdict holds where the rewrite is already known to be achievable.

Attack the term that sets the ceiling before attacking the term that is already small and getting smaller.

The Master Key: *Raising p from 0.6 to 0.8 beats raising N from 4 to 16, because processors shrink a term that is shrinking anyway while the parallel fraction moves the ceiling itself.*

HOT Question 78

Twice the processors, twice the speed. A supervisor at the water board in Busokelo said it while signing for a machine, and for the first doubling he was nearly right. By the seventh he was not close, and he had paid for all of them.

Assess the claim that a program will run twice as fast on twice as many processors.

Model Answer

The claim is a serviceable rule of thumb only while the parallel fraction is very close to 1 and the processor count is still small, and it fails so badly elsewhere that as a general statement it is worthless.

Where it very nearly holds: Take an unusually good program with a parallel fraction of 0.99. Moving from 1 processor to 2 gives one divided by the sum of 0.01 and 0.495, which is 1.980. That is close enough to 2 that nobody would quarrel with the rule, and it is the case people see first and then remember for ever.

Where the same program collapses: On 64 processors that program reaches 39.26. Doubling to 128 takes it to 56.39, a factor of 1.436 rather than 2. Doubling again from 512 to 1024 moves it from 83.80 to 91.18, a factor of 1.09, and the ceiling it is pressing against is 100.

$$1/((1 - 0.99) + 0.99/128) = 56.39$$

Why the rule decays: Doubling the speed would require the denominator to halve, and only the p over N part of it halves. While that part is the larger of the two, halving it nearly halves the whole denominator; once the serial term dominates, halving p over N barely disturbs the total.

Grounds the law does not even include: Processors must exchange data, wait for one another at synchronisation points and compete for the same memory bandwidth, and all three worsen as the count rises. A real machine therefore falls below the curve rather than sitting on it, so the law is already the optimistic case.

The verdict and the condition that decides it: Accept the claim for the first doubling or two on a program whose parallel fraction is above about 0.95, and reject it otherwise, because for anything less parallel even the second doubling misses badly. Anyone who quotes the rule should be asked for p and for the current processor count before it is believed.

The rule is not so much false as true only in the corner of the graph where nobody has a performance problem.

The Master Key: *Twice the processors gives twice the speed near the start of the curve on an almost entirely parallel program, and nowhere else, so treat it as a special case and never as a general rule.*

HOT Question 79

The nightly job at the district data office in Nkasi begins when the last clerk locks the door and should end before the first returns. Lately it finishes with nothing to spare, and a supplier offers thirty two processors. Most of that offer is waste.

A regional data office runs a nightly job that takes 8 hours. Profiling shows 6 hours of the job can be parallelised and 2 hours cannot. The office is offered a machine with 32 processors. Advise whether to buy it, and state what else you would want to know.

Present running time on one processor: 8 hours.

Work that can be divided: 6 hours. Work that cannot: 2 hours.

The offer: one machine carrying 32 processors.

Model Answer

Do not buy the 32 processor machine: 8 processors already bring the job down to 2.75 hours and the full 32 bring it only to 2.19 hours, so the extra 24 processors purchase 33.75 minutes on a job that finishes inside the night either way.

The two numbers that settle it: The parallel fraction is 6 divided by 8, which is 0.75, so the serial term is 0.25 and the ceiling is one divided by 0.25, which is 4. The job can never be brought below the 2 hours that cannot be divided, whatever is bought.

What 32 processors actually deliver: The parallel term is 0.75 divided by 32, which is 0.0234, so the denominator is 0.2734 and the speed-up is 3.657. Eight hours becomes 2.19 hours, which is 91 per cent of the ceiling of 4 and about as close to that ceiling as any amount of money can come.

$$1/((1 - 0.75) + 0.75/32) = 3.657$$

What 8 processors deliver for a quarter of the outlay: The parallel term is 0.75 divided by 8, which is 0.09375, the denominator is 0.34375 and the speed-up is 2.909, so the job takes 2.75 hours. Four times fewer processors, and the job finishes 33.75 minutes later inside a window that is hours wide.

Where the difference should be spent: Spend it on the two serial hours. Cut them to 1 hour and the job becomes 6 parallel hours against 1 serial hour, so the parallel fraction rises to 0.857 and the ceiling from 4 to 7. That is a larger prize than any machine on the price list is offering, and the smaller machine would then be running against a ceiling nearly twice as high.

What else I would want to know: First, whether the 2 hours are genuinely serial or merely not yet parallelised, since the second is a programming task and the first is a fact about the problem. Second, whether the job is limited by the processor at all: if most of those hours are spent reading and writing data, no processor count helps and a faster disk is the purchase.

The verdict is to buy the smaller machine and to treat the two serial hours, rather than the processor count, as the real target.

The Master Key: *Work out the ceiling before reading the price list, because on a job that is three quarters parallel the extra twenty four processors are worth half an hour while the serial hours are worth everything.*

HOT Question 80

The weather model at the research station in Meatu was never meant to finish sooner on a larger machine. It was meant to cover a finer grid inside the same overnight window, which it now does. Amdahl's law has nothing discouraging to say about that.

Amdahl's law assumes the problem stays the same size while the machine grows. Assess whether that assumption holds in practice, and what follows if it does not.

Model Answer

The assumption holds whenever a fixed piece of work has to be finished sooner, and it fails whenever the point of the larger machine is a larger problem, so the law is exact for a deadline and misleadingly pessimistic for growth.

What the law holds fixed: The workload is held constant and the only question asked is how much sooner it finishes. The parallel fraction is a property of that fixed workload, so it does not change as processors are added, and the ceiling follows immediately from it.

Where that matches practice exactly: A payroll run, an examination results process, an overnight reconciliation: the work is what it is, the deadline is fixed at daybreak, and doing more of it would be of no use to anybody. For these, the law is not merely a guide but the correct arithmetic.

Where the assumption breaks: In scientific work and much commercial analysis the bigger machine is bought to attack a bigger problem in the same time, not the old problem faster. A finer grid, a longer simulation, a larger database: nobody wants yesterday's question answered before breakfast.

Why that changes the numbers so much: The serial part is usually reading configuration, setting up and writing results, and it grows far more slowly than the work itself. Double the problem and the divisible part doubles while the serial part barely moves, so 6 divisible hours against 2 serial become 12 against 2, the parallel fraction rises to 0.857 and the ceiling from 4 to 7.

The observation that follows: Scaling the problem along with the machine gives a far better return than holding the problem still, because the effective parallel fraction rises with problem size instead of standing fixed. That scaled view answers the pessimism without contradicting a single line of the derivation; it measures something else.

The verdict: Both are right about different questions. Apply Amdahl's law when the work is fixed and a deadline decides, and apply the scaled view when the machine is being bought to do more, but state which question is being asked first, since the same machine looks poor under one and excellent under the other.

Ask whether you want the same work sooner or more work in the same time, because that choice, and nothing about the hardware, decides which view applies.

The Master Key: *Amdahl's law is exact for a fixed workload and pessimistic for a growing one, so name the question before quoting the ceiling.*

HOT Question 81

The laboratory at the technical college in Mkalama keeps a shelf of processors, one from each generation it owned, and the newest does not have the highest clock speed. Every lever in this chapter was pulled on that shelf, in a particular order.

Drawing on the whole chapter, assess where a designer should spend effort to make a processor faster, and justify an order of priority.

Model Answer

Reduce CPI first, then balance and modestly deepen the pipeline, then widen issue only while the return holds, and add cores or vector lanes last, because that final gain depends on software the designer does not control.

Raising the clock: The cycle can be no shorter than the slowest pipeline stage, so the clock is bounded from below by the worst stage in the design, and it is bounded from above by power, which climbs faster than the performance it returns. It is the easiest lever to promise and the hardest to deliver.

Deepening the pipeline: More stages make each stage shorter and raise throughput, but they lengthen fill and drain and make every mispredicted branch dearer, because more instructions must be discarded. Depth pays until the branch penalty consumes what the shorter stage won.

Removing stalls, where I would begin: Forwarding, split instruction and data caches and branch prediction each attack CPI directly, and CPI is 1 plus stalls per instruction, so every stall removed is a cycle no longer wasted. These are modest pieces of hardware, they help every program that runs, and they ask nothing of the programmer.

Widening issue: The checking grows roughly with the square of the width, since every candidate must be compared against every other, while the independence available inside one instruction stream does not grow at all. Each added slot is filled less often than the one before, so widen narrowly and stop early.

Cores and lanes last: A second core or a wider vector lane offers the largest headline gain and the least certain one, because it is capped by the parallel fraction and by whether anyone writes software able to use it. One divided by one minus p is the honest limit on everything spent here.

The verdict, and what should decide it: That order holds for a general purpose processor, and the reason is certainty of return: stalls are wasted cycles on every workload, while parallel hardware pays only when the software cooperates. Profile the real workload before spending anything, because each of these levers is worth a different amount on a different program.

The strongest design decision in this chapter is a measurement, since the same silicon spent differently is worth more or less depending entirely on what the machine will be asked to run.

The Master Key: *Spend on stalls first because every program pays for them, and on parallel hardware last because only some programs can collect on it.*

Chapter Three

ADVANCED COMPUTER NETWORKS

Of everything in this subject, computer networks is the part you will meet again on the day you leave school. The office you join will have a switch in a cupboard and a router beside it; the mobile-money agent at the corner depends on a link you could describe; the hospital that treats you keeps its records on a network somebody designed and somebody must maintain. It is also the chapter in which an examiner can set arithmetic and mark it without argument. A mask either leaves room for the machines a laboratory must connect or it does not, and a candidate who can calculate cannot be talked out of the right answer.

The chapter opens with the two models that organise everything after them, the seven layers of one and the four of the other, together with the vocabulary they hand you. It then turns to addressing, where the work becomes arithmetic: classes, masks, subnetting and the number of hosts a design can actually carry. From there it moves up to the transport and application protocols that carry real traffic, and down to the copper, fibre and radio that carry the bits, with the connectors that terminate them. The last three movements are practical ones: designing a network for a building and a budget, securing it against people who want in, and the telephone and mobile networks lying underneath it all.

Four habits earn marks here and none of them costs time. Name the model before you name the layer, because a layer number means one thing in the seven-layer model and something quite different in the four-layer one. Write the mask before you write the address, since the mask is what makes an address mean anything at all. Show the working and not only the answer, because a bare figure earns a fraction of what a derived one earns, and a wrong figure with sound working still earns something. Say which layer a device operates at before you say what it does, and half the confusion in this topic never arises.

Network models and the seven layers of OSI

This section settles what a reference model is for, why seven layers are worth their overhead, and what the model can and cannot fairly be asked to do. It names the layers in order, fixes the protocol data unit each one handles, separates the service a layer offers upward from the protocol it speaks sideways, and shows where the model parts company with the products actually sold. Everything later in the chapter is described in the vocabulary established here, so a layer you cannot name now is a mark you will lose later.

BINDER Question 1

A shop in Kariakoo sells you a network card made in one country, cable made in another and a switch made in a third. You plug all three together that afternoon and they work. Nobody in that chain ever telephoned anybody.

Define a network reference model, and explain why network communication is organised into layers rather than written as one complete protocol.

Model Answer

A network reference model is a published framework that divides communication into ordered layers, each with a defined function and a defined interface to its neighbours, so that equipment from different makers can interwork.

The problem it solves: before an agreed model each maker wrote its own complete system, and two machines interworked only if they carried the same badge. A published model lets an engineer build one layer knowing what the layer below hands up and what the layer above expects.

What a layer is: a set of related functions treated as one unit. It takes work from the layer above, adds what its own job requires, and passes the result down. What happens inside is nobody else's concern.

Why not one protocol: a monolithic protocol must be written and revised as one object. Divided into layers, each part is developed and replaced independently, so moving a link from copper to fibre disturbs layer one and leaves the six above it untouched.

Fault isolation: one job per layer means a fault belongs to a layer rather than to the network at large. A technician who confirms that frames leave the interface correctly has cleared the bottom two layers.

A common vocabulary: the model gives engineers who have never met a shared language. To say that a device operates at layer three settles a great deal in one phrase, and every fault report leans on that shorthand.

Where it came from: the International Organisation for Standardisation published the Open Systems Interconnection reference model as ISO 7498 in 1984, when rival proprietary systems could not exchange a byte. Its purpose was commercial as much as technical.

Advantage	Disadvantage
A common vocabulary for engineers who have never met	A reference model only; no product implements it exactly
Modular design, each layer built and tested alone	Seven layers are heavy for a small network
A protocol can be replaced without disturbing its neighbours	Its boundaries do not match the suite actually in use
Faults can be attributed to one layer	It says nothing at all about security
Equipment from different makers interworks	It says nothing about network management

Table 3.1: *What layering buys and what it costs.*

A reference model is not a product. It is an agreement about how to divide the work, and that agreement is what makes the Kariakoo counter possible.

The Master Key: *Layer the job and you can replace any one part of a network without renegotiating the whole of it.*

BINDER Question 2

Two wall charts of the same seven layers hang in the laboratory of a secondary school in Bariadi, and they disagree. On one, physical sits at the top; on the other, at the bottom. The technician who pinned them up says neither is wrong.

Name the seven layers of the OSI model in order from the bottom upwards, and state in one clause what each layer is responsible for.

Model Answer

From the bottom upwards the layers are physical, data link, network, transport, session, presentation and application, and each one exists to solve a problem created by the layer beneath it.

Layer one, physical: moves raw bits across the medium as voltage, light or radio, and fixes the connectors, voltages and data rates. It attaches no address and recognises no structure.

Layer two, data link: delivers frames from one node to the next across a single link, using hardware addresses, and detects the errors the medium introduces so that they are not passed silently upward.

Layer three, network: delivers packets between hosts that are not on the same link, choosing a route through intermediate networks on the strength of a logical address rather than a hardware one.

Layer four, transport: delivers data end to end between two processes rather than two machines, and, where the protocol offers it, provides sequencing, acknowledgement, retransmission and flow control.

Layer five, session: establishes, manages and closes the dialogue between two applications, deciding whose turn it is to speak and where to resume after an interruption.

Layer six, presentation: settles how data is represented, so that character coding, compression and encryption are agreed before the two applications try to read each other’s bytes.

Layer seven, application: provides the network services the user’s program actually calls upon, such as file transfer, electronic mail and name lookup. It is not the program itself.

Many students climb the stack with the sentence Please Do Not Throw Sausage Pizza Away, whose initials run physical, data link, network, transport, session, presentation, application. Learn it upwards, since data leaving a host travels the other way and the two directions are easy to confuse.

The Master Key: *Read the stack upwards to remember it and downwards to understand it.*

BINDER Question 3

A letter posted from Mafia to Lindi is a letter, then a letter in an envelope, then one item in a mail sack, then cargo in a hold. Nothing on the page changes, yet the thing handled has a new name at every stage.

Using Figure 3.1, state the protocol data unit handled at each of layers 1 to 4, and explain why the unit changes its name as data descends the stack.

Model Answer

The unit is a segment at layer four, a packet at layer three, a frame at layer two and a bit at layer one, and the name changes because each layer adds its own header and hands on an object governed by different rules.

What a protocol data unit is: the complete object one layer hands to its peer: the data received from the layer above together with the control information that layer has added. The added part is a header, and at layer two a trailer as well.

Layer four, the segment: the transport layer divides the application's data stream into pieces small enough to travel and adds a header carrying port numbers and sequence information. The result is a segment; a connectionless transport calls its unit a datagram instead.

Layer three, the packet: the network layer takes the whole segment, header and all, treats it as data, and prefixes a header carrying logical source and destination addresses. The result is a packet, and it is the packet a router examines.

Layer two, the frame: the data link layer wraps the packet in a header carrying hardware addresses and a trailer carrying a frame check sequence, producing a frame. Only a frame can be placed on the medium.

Layer one, the bit: the physical layer has no header to add. It converts the finished frame into a stream of bits and then into signals, so its unit is simply the bit.

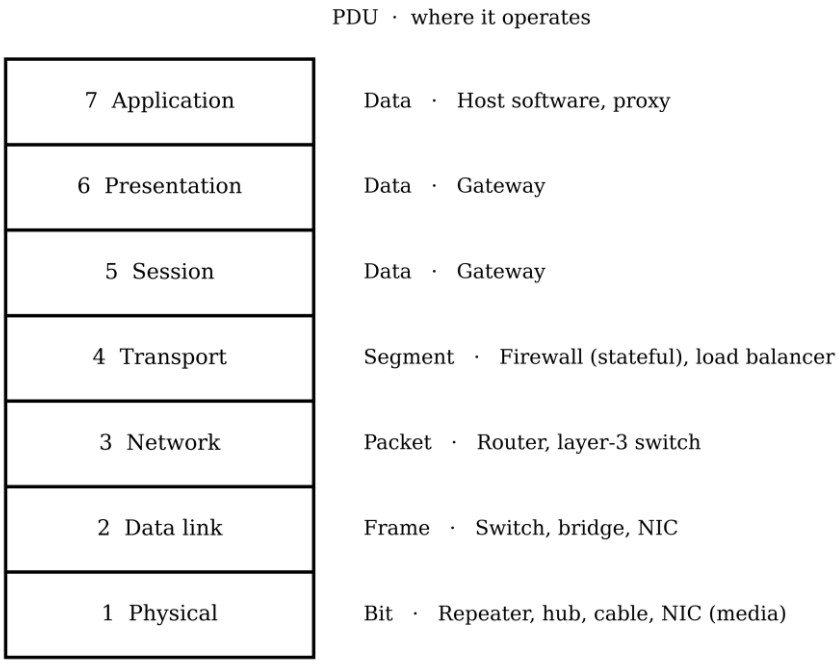


Figure 3.1: *The seven layers of the OSI model, with the protocol data unit each one handles and the equipment that operates there.*

Why the name must change: each name records how much control information has been added and therefore which rules apply. Calling a frame a packet loses the fact that hardware addresses are now present, and that is exactly what a fault report needs.

On the receiving host the sequence runs backwards, each layer stripping the header its peer added, so the application receives exactly the bytes that were sent.

The Master Key: *A new header means a new name, so name the unit and you have named the layer.*

BINDER Question 4

Across the counter at a bank branch in Msimbazi the clerk accepts your deposit slip and hands back a receipt. That exchange is a service. What she agrees with a clerk at the Mikocheni branch, in a language you never hear, is a protocol.

Distinguish between a service and a protocol as the OSI model uses the two terms, and explain why the distinction matters.

Model Answer

A service is what a layer offers vertically to the layer above it through a defined interface, while a protocol is the set of rules a layer uses horizontally to converse with the same layer on the other host.

A service is vertical: a layer offers a set of operations to the layer above at the boundary between them, and that layer calls the operations without knowing anything about how they are carried out. A service defines what is available, never how it is achieved.

A protocol is horizontal: the same layer on the two hosts agrees the format of the units they exchange, the meaning of every field, and the order in which units may be sent. That agreement is the protocol, and the layer above cannot see it.

Why the separation is useful: because the service is what is promised and the protocol is only how the promise is kept, a protocol can be replaced entirely without disturbing anything above it. The layer above notices nothing, since the interface it calls has not moved.

The peer conversation is logical: the horizontal lines in Figure 3.2 carry no bits. Every unit descends to the medium on the sending host and climbs again on the receiving one, and the peer relationship exists only because each layer removes exactly what its peer added.

Naming them correctly: a service is described by what it provides, such as reliable ordered delivery. A protocol is named and versioned. If you can put a name and a version number to it, you are talking about a protocol.

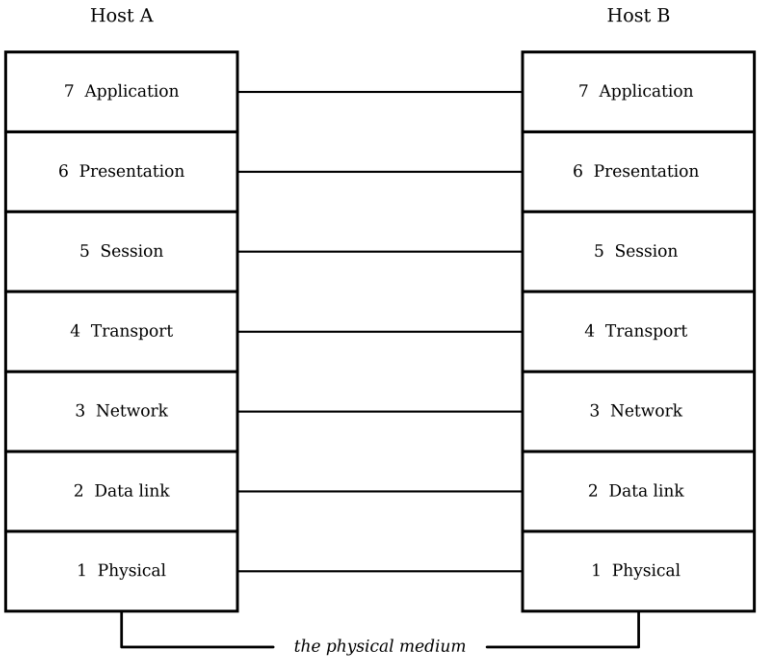


Figure 3.2: *Peer protocols. Each layer converses with its own number on the far host, although every real bit travels down to the medium and back up.*

Hold the two apart with a single question. Is this something a layer offers upward, or something it agrees sideways?

The Master Key: *A service is a promise made upward; a protocol is the conversation that keeps it.*

REAL Question 5

Two tenders go out from a regional office in Sengerema in one week. The eight switches come from one importer, the two routers from another, and neither company has heard of the other. The network works on the first afternoon it is powered up.

Explain how the OSI model makes such a purchase safe, and state the layers at which each of the two kinds of device operates.

Model Answer

The purchase is safe because both suppliers build to the same published layer boundaries, so the switch and the router meet only at defined interfaces and neither needs to know how the other is built inside.

What the model guarantees: a standard fixes what crosses the boundary between one layer and the next, and what one layer sends to its peer. Provided each product obeys the standard for the layers it implements, the two interoperate whatever their badge, and no private agreement between the importers is required.

Where the switch operates: a switch reads the destination hardware address in each frame, looks it up in a table it has learned from traffic, and forwards the frame out of the single port that leads to it. That is layer two. It also implements layer one, because it must receive and regenerate the signals themselves.

Where the router operates: a router reads the logical destination address in the packet and chooses the next network to send it towards, which is layer three. It also implements layer two, because to reach a packet at all it must first accept the frame that carried it and then build a fresh frame for the next hop.

Why each device names two layers: a device operates at the highest layer whose information it acts upon, and it must implement every layer below in order to reach that one. This is why a switch is a layer-two device and a router a layer-three device although both handle frames and bits.

The practical protection: if the routers later prove too slow, the office replaces them with routers from a third supplier and the switches, the cabling and the connectors all stay. Layering is what turns a network into something repaired one part at a time rather than replaced whole.

Where the model does not protect: interoperability is guaranteed only for what the standards cover. Management interfaces, configuration syntax and support arrangements differ between makers, so the procurement officer should still ask whether the technician who maintains both can configure both.

The tender is safe at the level of protocol and risky at the level of skills, and only the first of those two is the model’s business.

The Master Key: *A standard promises that two devices will talk to one another; it promises nothing about whether anyone in the building can configure them.*

REAL Question 6

An intern on attachment at a district hospital in Tarime pins a card above the server cupboard on her last day. Three statements, a neat hand, complete confidence. Two can be found in reputable diagrams. All three are wrong.

Identify what is wrong with each of the three statements on the card, and give the correct position with the reason in every case.

1. The peer unit exchanged between the session layers of two hosts is the SSDU.

2. Data at layer four on one host travels sideways to layer four on the other host, which is why the two are drawn side by side.

3. Every network product implements all seven layers of the OSI model.

Model Answer

All three are wrong, and each mistake comes from reading a picture of the model literally instead of reading what the picture stands for.

The first, on the session unit: the unit a session layer exchanges with its peer is the session protocol data unit, the SPDU. An SSDU is a session service data unit, a different object travelling in a different direction: it is what the session layer receives across the interface from the layer above, before any session header exists.

The pattern behind the letters: the same pair exists at every level. A service data unit is handed down across an interface; once the layer adds its own header it becomes a protocol data unit and is sent sideways to the peer. Learn the pair once and the letters stop being alphabet soup.

The second, on sideways travel: no data crosses horizontally. Layer four on the sending host passes its segment down to layer three, and the whole object eventually reaches the medium as signals; on the far host it climbs back up. The horizontal line means only that the receiving layer four reads exactly what the sending layer four wrote.

The third, on all seven layers: almost no product does. A repeater implements layer one alone, a switch layers one and two, a router layers one to three. Only a host running application software reaches layer seven, and even that host runs a suite which merges the top three layers rather than keeping them apart. Believing otherwise is what persuades a buyer to pay for a device advertised as covering all seven layers when the site needs a router.

What the three share: each treats the model as a description of equipment rather than as a framework for describing equipment. Before answering anything of this kind, ask what the diagram is a diagram of. The seven boxes are functions that must be performed somewhere, not seven components present in everything you buy.

The intern is not careless. All three statements can be found in print, which is why the habit of asking what a diagram represents is worth more than memorising the diagram.

The Master Key: *The seven layers are jobs that must be done, not parts that must be present.*

HOT Question 7

At a technicians' meeting in Makambako an engineer of fifteen years makes a case that sounds unanswerable. He has never configured a session layer, never bought a presentation layer, never met a product that implements all seven. Teach the four-layer model, he says.

Assess the argument that a model which no real product implements should no longer be taught, and reach a judgement.

Model Answer

The argument is right about the facts and wrong about what a reference model is for, so the model should be kept, but taught as a framework for describing networks rather than as a description of any network.

The strength of the argument: the factual claim is correct. No commercially significant stack implements the seven layers as separate entities. The suite that actually carries the world's traffic folds the top three into one and the bottom two into one, and its layers were defined before the model was published.

The cost in teaching hours: time spent on the session and presentation layers is time not spent on addressing, routing or security, where a student's errors cost real money later. Teaching hours are finite and the opportunity cost is genuine.

What the model omits: it says nothing about security and nothing about network management, which are two of the largest concerns a working administrator has. A framework silent on both is plainly not a complete account of the job.

Where the argument fails: a reference model is not a specification for a product; it is a vocabulary for talking about products. Nobody objects that a map of a district is not the district. The value lies in agreed names for the functions, and every one of the seven functions is performed somewhere in every real network.

The evidence from practice: engineers who have never heard of ISO 7498 say layer two switch, layer three routing and layer one fault every working day of their lives. The numbering has outlived the products it was written for, which is a strange fate for something useless.

What replacing it would cost: the four-layer suite gives no separate name to the functions of layers five and six, so a student taught only that model has no word for dialogue management or for negotiating character encoding, and must invent one when the problem arrives.

The judgement: keep the seven-layer model and change what is claimed for it. Teach it as the vocabulary and the four-layer suite as the implementation, naming which of the two is in use each time, and the engineer's complaint dissolves, because he was never objecting to the model but to a bad way of presenting it.

The test of a model is not whether anything has been built exactly like it, but whether people who must work together can describe what they have built.

The Master Key: *Judge a reference model by the arguments it lets people settle, not by the products it resembles.*

HOT Question 8

One week of training, a shelf of borrowed manuals and a laboratory she did not install: that is what a mathematics teacher in Njombe inherits when her school hands her the network. She asks the district technician which single layer to learn first.

Judge which single OSI layer she should learn first, and defend your choice against the two strongest alternatives.

Model Answer

She should learn layer three first, because almost every decision she will actually have to make, and almost every fault she cannot cure by swapping a cable, is a question of addressing.

What layer three gives her: an address plan for the school, the meaning of a subnet mask, the role of the default gateway, and the reason a machine with a sound cable and a live switch port still reaches nothing. Those four items account for most of what the manager of a small network decides.

Why the faults land there: a duplicated address, a wrong mask, a missing gateway or a laboratory that has quietly run out of addresses all produce the same complaint from the staff room, that the network is down, and not one of them is visible from the cable or the switch.

The first rival, layer one: the case is strong, since more faults in a school are physical than anything else: a crushed cable behind a bench, a badly crimped plug, a run made too long. Against it, physical faults announce themselves. A dead port shows no link light, and a colleague can be taught to test a cable in an afternoon.

The second rival, layer two: switching and virtual LANs matter, and separating student traffic from the office is a real need. Against it, a school of this size runs correctly for years on a flat switched network with factory settings, and cannot run for a week without a coherent address plan.

The order that follows: learn layer three properly, then layer one as a practical skill with a cable tester in hand, then layer two when the school grows enough to need segmentation. The four layers above can wait until something breaks in them.

The judgement: layer three, and without much hesitation. It is the layer where her decisions are recorded, where her mistakes persist invisibly, and where she must talk to the company selling the school its connection.

A different school would justify a different answer, and the defence is what is being marked rather than the choice, so name the rivals and say plainly why they lose.

The Master Key: *Learn first the layer whose mistakes are invisible, because the visible ones will teach themselves.*

What each layer does: protocol, PDU and device

This is the reference section of the chapter. Table 3.2 sets out, for every one of the seven layers, what the layer does, what its unit is called, which protocols operate there and which piece of equipment reads that far up. Learn that one table and a large family of questions becomes mechanical: name the layer, name its unit, name a protocol, name the box. The questions that follow build the table, test it from several directions, and settle the two arguments it provokes.

BINDER Question 9

In the ICT office of a district council in Kahama the fault goes round the table by layer. Layer 3, says one; layer 2, says another; presentation, says a third, because the file will not open. Nobody says what any layer does to what passes through.

State the function of each of the seven layers of the OSI model, in a single sentence for each.

Model Answer

Each layer performs one job on the unit it is handed and passes the result on: layer 7 offers services to programs, layer 6 fixes the form of the data, layer 5 manages the dialogue, layer 4 delivers between processes, layer 3 between networks, layer 2 across one link, and layer 1 delivers the bits themselves.

Layer 7, the application layer: offers network services to the program a person is using. It is not the browser or the mail client but the protocol those programs speak, so that a request is intelligible to a server that has never met them.

Layer 6, the presentation layer: settles the form the data takes on the wire, translating between character sets, compressing where that pays and applying encryption, so that both ends read the same bytes the same way.

Layer 5, the session layer: opens a conversation between two programs, keeps track of whose turn it is, marks places to resume from, and closes it cleanly when the work is done.

Layer 4, the transport layer: delivers between processes rather than between machines, cutting the data into numbered units, addressing each to a port, and deciding whether losses are repaired or simply accepted.

Layer 3, the network layer: delivers a packet from a host on one network to a host on another, choosing a path through routers that neither end can see or influence.

Layer 2, the data link layer: delivers a frame across a single link to the next device on it, marking where the frame begins and ends, addressing it physically and checking whether it arrived intact.

Layer 1, the physical layer: puts bits onto the medium as voltage, light or radio and recovers them at the far end, understanding nothing whatever about what they mean.

Layer	What it does	PDU	Protocols and devices
7 Application	Offers network services to the user's program	Data	HTTP, HTTPS, FTP, SMTP, POP3, IMAP, DNS, DHCP, SSH, Telnet; application gateway, proxy
6 Presentation	Settles character set, compression and encryption	Data	TLS, ASCII, JPEG, MPEG; host software only, no separate box
5 Session	Opens, maintains and closes the dialogue	Data	NetBIOS session service, RPC, PPTP; host software only
4 Transport	Delivers process to process, reliably or not, using ports	Segment (TCP) or datagram (UDP)	TCP, UDP; stateful firewall, load balancer
3 Network	Delivers host to host across networks, choosing a path	Packet	IP, ICMP, IGMP; router, layer-3 switch
2 Data link	Delivers across one link, framing, addressing and checking	Frame	Ethernet, PPP, 802.11 MAC, and ARP straddling the boundary above; switch, bridge, wireless access point
1 Physical	Puts bits on the medium as signals and recovers them	Bits	Ethernet PHY, DSL, Wi-Fi radio; hub, repeater, cable, connector, NIC transceiver

Table 3.2: *The seven layers: function, protocol data unit, example protocols and the equipment that operates there.*

Read the seven downward and each narrows the problem for the one beneath it. Read them upward and each hides a difficulty from the one above.

The Master Key: *Answer a layer question with a verb, because a layer is something done and not something that exists.*

BINDER Question 10

Painted along the corridor wall of a training centre in Lindi is the layer table, three protocol cells never filled in. A technician glances at it, says one question would fill all three, and leaves for his bus without saying what it is.

Name two protocols that operate at each of the three layers left blank below, and say in one clause what each of the six does.

1. Layer 7, application. Protocol data unit: data. Protocols: two required.

2. Layer 4, transport. Protocol data unit: segment or datagram. Protocols: two required.

3. Layer 3, network. Protocol data unit: packet. Protocols: two required.

Model Answer

The blanks are filled by HTTP and SMTP at layer 7, TCP and UDP at layer 4, and IP and ICMP at layer 3, and each pair is worth learning as a pair because its two members do the same layer's job in two opposite ways.

Layer 7, HTTP and SMTP: HTTP carries a request for a resource and the reply that satisfies it. SMTP carries a mail message from one server to the next. Each hands a finished message downward and neither knows or cares how it will travel.

Layer 4, TCP and UDP: TCP numbers what it sends, acknowledges it and retransmits what is lost, so a file arrives complete. UDP does none of that and is quicker for it, which suits a voice call, where a repair that arrives late is worse than the gap it was meant to fill.

Layer 3, IP and ICMP: IP carries the packet and holds the source and destination addresses on which every routing decision is taken. ICMP carries no user data at all, only reports about what became of packets, and it travels inside IP rather than beside it.

Others that would earn the mark: at layer 7, FTP, DNS, DHCP, POP3, IMAP, SSH and Telnet. At layer 3, IGMP. At layer 4 the pair is the whole answer, which is worth knowing, because a question demanding three transport protocols is asking for something that does not exist.

The test that places any protocol: ask what it addresses. Something that names a resource or a mailbox belongs at layer 7. Something that names a process by a port number belongs at layer 4. Something that names a host on a distant network belongs at layer 3.

Two examples are the minimum an examiner will accept, and a third costs nothing to add. Write them with the layer number beside each, never as a loose list.

The Master Key: *A protocol tells you its layer by what it addresses: a mailbox, a port, or a network.*

BINDER Question 11

A fault ticket raised at a college in Kinondoni reads: the packet never reached the switch. The engineer who picks it up writes back that no switch has ever seen a packet in its life. Both describe the same bytes leaving the same machine.

For each of layers 1 to 4, name the protocol data unit, one protocol that defines it and the device that reads that far, and state the rule that limits how deep any device reads.

Model Answer

Each layer pairs its protocol data unit with the protocol that defines it and the device that reads it: bits, the Ethernet PHY and a hub at layer 1; the frame, Ethernet and a switch at layer 2; the packet, IP and a router at layer 3; the segment, TCP and a firewall at layer 4.

Reading depth fixes the pairing: a device parses the unit at its own layer and treats whatever is sealed inside as payload. The three names travel together, because the protocol writes the fields and only a box built to parse them can act on what they say.

The bit and the hub: layer 1 has no address of any kind, so a physical standard such as the Ethernet PHY settles only how a bit reaches the medium. A hub regenerates the signal and never assembles the bits into anything it could read.

The frame and the switch: Ethernet defines the frame, its delimiters, the hardware addresses it carries and the check value at its tail. A switch reads those, takes the destination address, finds the port and sends the frame on with the packet inside unopened.

The packet and the router: IP defines the packet and the logical addresses it carries, and a router reads them. It opens the frame to reach the packet and builds a fresh one for the next link, but the transport header inside stays closed to it.

The segment and the firewall: TCP defines the segment and the port numbers that name a program rather than a machine, and the box reading that deep is a stateful firewall or a load balancer. Under UDP the unit is a datagram, so name the transport too.

The ticket and the reply describe the same bytes at two depths. The switch carried a frame, and the packet sealed inside it went past unread.

The Master Key: *One layer, one unit, one protocol, one box: fix any of the four and you have fixed the other three.*

BINDER Question 12

A school in Msimbazi ran two fibre pairs between its office and classroom blocks, because the committee was told two cables mean two paths. The switches use one pair and leave the other dark. Nothing is broken, and no configuration will change it.

Using Figure 3.3, distinguish the forwarding decision a switch makes from the routing decision a router makes, and explain why only one of the two can choose between paths to a distant network.

Model Answer

A switch reads only as far as layer 2, where the address is flat and says nothing about which network a host sits on, so it has no competing routes to compare; a router reads to layer 3, where the address carries a network number that its table may reach in two ways.

The rule the figure states: a device reads only as far up the stack as its own layer and no further. A hub sees signals, a switch sees frames, a router sees packets. What a device cannot read, it cannot act on.

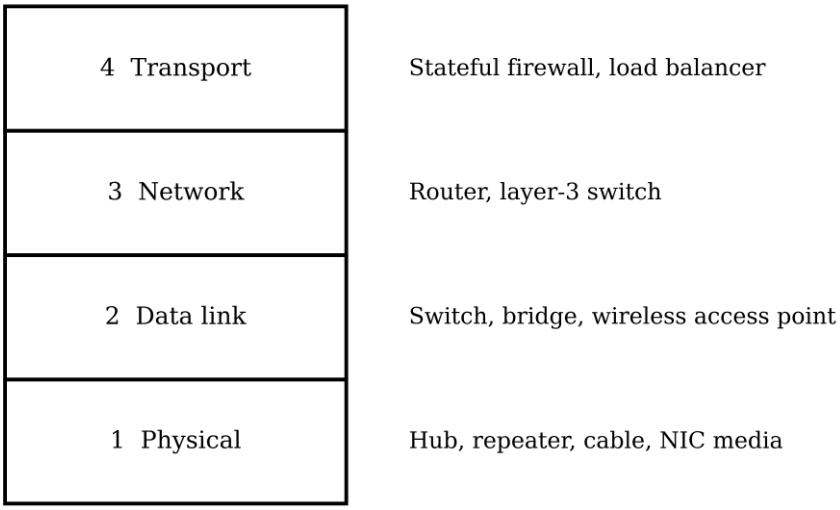


Figure 3.3: *Which device operates at which layer. A device can read only as far up the stack as its own layer, and no further.*

What a switch has to work with: a destination hardware address, which is flat. It identifies one interface and carries no clue about where that interface is, so there is no shorter or longer route to it. The address table holds one port per address, and a lookup is not a choice.

Why the second pair stays dark: two live paths between the same two switches would circulate a broadcast frame indefinitely, because a frame carries no field that counts down and kills it. The spanning tree protocol therefore blocks one link outright, which is loop prevention and not path selection.

What a router has instead: the destination logical address, which is hierarchical, having a network part and a host part. Two entries for the same destination network can sit in one routing table with different metrics, and the router compares them before it forwards anything.

Forwarding against routing: a switch forwards, which means looking an address up and sending the frame out of the single port the table names. A router routes, which means weighing candidates first and forwarding afterwards. The difference is not speed but reading depth.

The second fibre pair is worth having, because it becomes the live path the moment the first fails. It is redundancy, which layer 2 does provide, and not path selection, which it cannot.

The Master Key: *A switch looks an address up; a router weighs one address against another.*

REAL Question 13

The district hospital in Tarime keeps its wards in one block and its pharmacy across a 60 m yard. Since Monday ward staff open the patient register but never the pharmacy stock system, and the pharmacy cannot reach the wards. Nothing changed over the weekend.

Identify the layer at which the fault most probably sits, and justify the choice from the symptom alone, before any equipment is touched.

Model Answer

The symptom points to layer 3, the network layer, because the boundary between what still works and what fails is exactly the boundary between two networks, and that boundary exists at no other layer.

Read the partition, not the complaint: everything inside a block works and everything across the yard fails, for every user and every application, in both directions. A fault sorts the world into what still functions and what does not, and the shape of that sorting is the evidence. Here the sorting follows the buildings exactly.

Why layers 1 and 2 are largely cleared: inside each block frames are reaching switches and machines are answering, so the cabling, the ports and the switch address tables are doing their work there. A layer 2 fault confined to a block would take out the neighbouring machines first, which is the reverse of what is reported.

Why the upper layers are cleared: the register and the stock system are different applications on different servers, and both fail in the same direction at the same moment. A fault at layer 6 or layer 7 lives inside one application, so it would take one of them out and leave the other running.

What layer 3 alone explains: two blocks on two networks can only reach each other through a router. A wrong default gateway, a missing or withdrawn route, an interface administratively shut down, or two subnets that overlap will each produce this exact symmetry, and each of them is a layer 3 object.

The rival that must be named: the link across the yard may simply be dead, which is layer 1 and gives the same partition. The honest answer is layer 3 first with layer 1 immediately behind it, and one observation separates them: whether the router reports its interface towards the pharmacy as up.

Writing the justification: state what works, state what fails, name the one crossing point both groups have in common, and name the layer at which that crossing exists. That is the entire argument, and it can be made from the ward door.

A fault that respects a building boundary is telling you it respects a network boundary, because in a well built site the two were drawn on the same plan.

The Master Key: *Let the boundary between what works and what fails name the layer for you.*

REAL Question 14

A quotation reaches a secondary school in Kondoa for a twenty-four port device described in one line as a layer-4 switch, priced a third above the managed switch the school had budgeted for. The bursar wants to know what the extra third buys.

Explain what a device would have to do to deserve the description used in the quotation, and advise the school whether that description on its own is worth paying for.

Item 3. Managed layer-4 switch, 24 ports, gigabit, rack mounting.

Quantity 1. Price one third above the layer-2 managed switch quoted last term.

No further specification supplied.

Model Answer

To deserve the description, the box must read into the TCP or UDP header and take its forwarding, filtering or priority decisions on port numbers as well as on addresses; that is a real capability, but as one line on a quotation it is a label rather than a specification, so the school should not pay the premium until it is told which layer-4 decisions the box actually makes.

What reading to layer 4 costs the hardware: the destination port sits behind 36 bytes of headers, being 14 bytes of Ethernet header, 20 of IPv4 header and the 2 bytes of source port that precede it. A layer-2 switch stops after the first 14. Parsing three headers on every frame on every port at full speed is what the premium pays for.

What such a box does with that reading: it treats services differently. Traffic to port 443 can be given precedence over a backup job running elsewhere; a rule can drop everything addressed to port 23 while leaving port 22 open. Those are useful things to be able to do, and they are what the phrase ought to mean.

Why the phrase is loose: no standard defines it. In practice it is applied to anything from a layer-3 switch that can match a port number in one access list, to a load balancer that terminates connections on behalf of clients. Two boxes wearing the same label may have almost nothing in common.

The four questions to put to the supplier: does the box route between virtual networks, can its access lists match port numbers, is that matching done in hardware at full speed or in software, and how many such rules will it hold. Four answers turn a label into a specification the school can compare with a rival quotation.

The advice: a school with one internet link and a few dozen machines already concentrates its traffic at the router, which is the natural place to prioritise and to filter. Buy the managed switch that was budgeted for, put the difference into the router or into more ports, and revisit layer 4 when there are services worth protecting from one another.

Nothing here says the device is bad. It says the line describing it carries no information, and a bursar cannot buy a number.

The Master Key: *A layer number on a quotation is a claim and not a capability, so make the supplier spend it.*

REAL Question 15

The technician leaving a secondary school in Bariadi wrote three pages of notes for his replacement, and most is sound: which cupboard holds the patch leads, which port feeds the library, which machine may not be switched off. Three lines about equipment are wrong.

Correct each of the three statements, giving in each case the fact that should have been written and the reason it is so.

1. The hub in the staff room reads the MAC address on each frame and sends the frame only to the port that needs it.

2. The router works at the transport layer, because it is the device that deals with port numbers.

3. The presentation layer compresses a file before the user saves it, which is why the saved file is smaller than the original.

Model Answer

All three are wrong: a hub is a layer 1 device and can read no address of any kind, a router forwards on the layer 3 destination address rather than on port numbers, and the presentation layer acts on data leaving for the network and never on a file being written to a disk.

A hub reads nothing at all: a hub works at layer 1. It receives an electrical signal, regenerates it and repeats it out of every other port, and it holds no table in which an address could be stored. Reading a destination MAC address means interpreting a frame, and a frame is a layer 2 object. The device that does that is a switch, which is why replacing a hub with a switch cuts traffic on every port.

A router is a layer 3 device: it forwards on the destination logical address in the packet header and reaches its decision without opening the transport header at all. Some routers additionally filter or translate using port numbers, but that is a firewall or address translation function riding on the same chassis. Doing extra work higher up does not promote a device to a higher layer.

The presentation layer never touches the disk: layer 6 acts on data that is already on its way to the network, agreeing a character set, compressing for transmission and applying encryption. Saving a file is an operating system action on local storage, and no layer of a network model takes part in it. Compression that shrinks a stored file is done by the application itself.

What the three have in common: each gives a job to a device or a layer without first asking what that device or layer is able to read, and each is plausible because the job really is done by somebody. Ask what information the job requires, find the header that carries it, and the layer names itself.

Notes like these survive for years because they are written by someone competent and read by someone new. Correct them on the page, and date the correction.

The Master Key: *Before you give a device a job, ask how deep into the frame it is able to read.*

HOT Question 16

A capture from a laboratory machine in Magu shows an ARP request inside an Ethernet frame with nothing between the two. No IP header, and the type field reads 0x0806 rather than the 0x0800 that announces IPv4. Two teachers read it in opposite directions.

Argue whether ARP belongs to layer 2 or to layer 3, and commit to one answer.

Model Answer

ARP straddles the boundary, but forced to a single number the answer is layer 2, because the tests that decide the question are how a protocol travels and how far it reaches, and on both of those ARP behaves exactly like a data link protocol even though the addresses it resolves belong to layer 3.

The case for layer 3: ARP exists only to serve IP. Its message carries logical addresses in its own fields, it is useless on a link that runs no IP, and the general rule that a protocol sits one layer above whatever carries it would place anything travelling in an Ethernet frame at layer 3, which is precisely where IP itself sits.

The case for layer 2, on carriage: ARP is not inside IP. It is placed straight into an Ethernet frame with a type field of its own, and the whole message for IPv4 over Ethernet is 28 bytes with no network header in front of it. A protocol that IP does not carry, and that carries no IP header itself, is difficult to call a protocol of IP's layer.

The case for layer 2, on scope: an ARP request is broadcast to every port on the link and stops there. No router forwards one, and none ever has. Reaching exactly one link and no further is the definition of layer 2's reach, while crossing between links is the definition of layer 3's, and ARP does not cross.

Why the addresses do not settle it: naming an address is not the same as delivering to one. ARP quotes a logical address in order to ask a question about it, and the answer it receives is a hardware address. Nothing is ever delivered to the address ARP names, and delivery is the whole business of layer 3.

The judgement: call it layer 2, at the boundary with layer 3. It travels as a link protocol, it lives and dies on one link, and it hands a service upward exactly as layer 2 is supposed to. In an answer, write that ARP sits at the boundary of layers 2 and 3 and is carried directly in a frame rather than inside IP, then commit to layer 2 and give the reason. A marker looking for either number will find it, and the reason is what is being marked.

Charts that print ARP in the gap between two rows are not being evasive. They are recording the one protocol whose carriage and whose purpose point at different layers.

The Master Key: *Judge a protocol by what carries it and how far it reaches, not by the addresses it happens to mention.*

HOT Question 17

The ticketing office of a bus company in Ifakara loses its connection to the booking server for a minute or two, three or four times a day, always returning before anyone can check anything. The clerk now writes down the time of each failure.

Design an ordered layer-by-layer procedure for diagnosing this fault, and defend the order you choose.

Reported: the connection to the booking server drops for one to two minutes, three or four times a day.

Affects all four counter machines at once. The office printer stays reachable throughout.

Every test run after an incident passes. Nothing has been reconfigured for six months.

Model Answer

Work upward from layer 1, one layer at a time, and record counters and logs rather than single tests, because every layer depends on the service of the one below it and an intermittent fault will pass any snapshot test taken between incidents.

Layer 1 first, and by counters: inspect link lights, connectors and patch leads, then read the interface error counters at both ends of every hop, once before an incident and once after. A cracked lead or a failing transceiver shows itself as a rising count of errors rather than as a dead port, and a dead port is all a single test can see.

Layer 2 second: look for a duplex mismatch, a port flapping up and down, a spanning tree recalculation, or wireless clients reassociating. The switch timestamps all of these in its log, so lay the log beside the clerk's written times. A topology change recorded at the minute of an outage has answered the question.

Layer 3 third: leave a continuous ping running to the default gateway and a second one to the booking server. Two traces together show at once whether the local network or the path beyond it is dropping, and a duplicate address or a route withdrawing and returning betrays itself in the pattern of the losses.

Layer 4 and above, last: if the pings survive an incident and the application does not, the fault is above layer 3. Test whether the server port answers during an outage, then look at name resolution, the application log and the load on the server. These come last because they cost most to investigate and are easiest to blame.

Why the order runs upward: each layer is built on the service of the one below, so a fault at the bottom counterfeits a fault at the top. Clearing layer 1 removes every explanation that rests on it, and the lower tests are the cheapest and quickest. The printer staying reachable has already cleared much of the office wiring.

The judgement, with one concession: keep the upward order, but open with the single continuous ping to the gateway, because it exercises layers 1 to 3 together and, if it survives every incident, it licenses a jump to layer 4. Order by dependency rather than by suspicion, and for an intermittent fault trust counters and logs over any test you can run on demand.

Notice how much the clerk's list of times is worth. It converts an intermittent fault into moments that can be looked up in a log, which is what turns guessing into diagnosis.

The Master Key: *Diagnose upward, because a fault at the bottom wears the costume of every layer above it.*

The TCP/IP suite and its mapping onto OSI

The seven layers you have just learned are a reference model. The Internet is not built on them. It is built on four layers that were carrying traffic a decade before the seven were ever published, and the two vocabularies now sit side by side in the same questions. This section names the four, maps them onto the seven so that every OSI layer is accounted for exactly once, weighs the two models honestly against each other, and settles the naming problem that costs more marks in this topic than any other single thing.

BINDER Question 18

Every message leaving a mobile-money agent's counter in Kariakoo passes through four pieces of software before the wire, and the same four in reverse coming back. Those four have carried their names since the 1970s, and no manufacturer has sold equipment that ignores them.

Name the four layers of the TCP/IP suite, working from the bottom upwards, and state the responsibility of each.

Model Answer

The TCP/IP suite has four layers, network access at the bottom, then Internet, then transport, then application at the top, and between them they carry a message from a process running on one machine to a process running on another.

Network access: the bottom layer places bits on the medium and delivers a frame to the next device on the same physical network. Connectors, signalling, framing and hardware addressing all belong here, and its contents are replaced entirely when copper gives way to fibre or to radio.

The Internet layer: gives every host a logical address and moves a datagram out of one network and into another, choosing a route as it goes. IP defines the address and the header, ICMP reports what went wrong, and the router is the device that works here.

The transport layer: opens a conversation between two processes rather than two machines, each end identified by a port number. TCP numbers its segments, acknowledges them and resends whatever is lost; UDP adds almost nothing and is chosen where speed outweighs certainty.

The application layer: everything the user's program needs in order to be understood at the far end, which means naming, formatting and the dialogue itself. HTTP, DNS, SMTP and DHCP all live here, each defining its own message format and its own rules of exchange.

What holds the four together: each layer takes a unit from the layer above, adds a header of its own and passes the result downwards. The receiving host removes those headers in the reverse order, so a header written by one layer is read only by the matching layer at the other end.

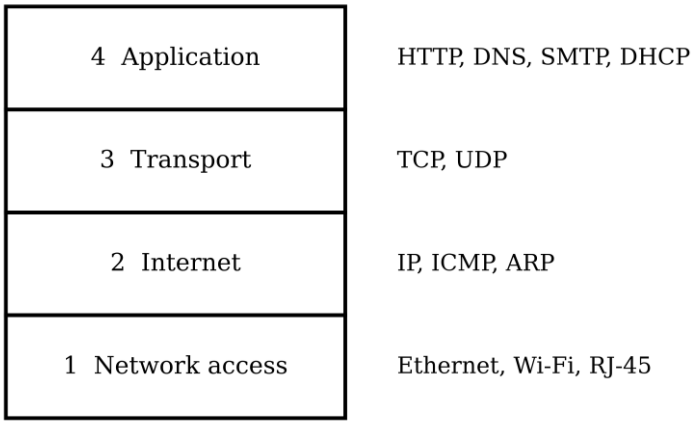


Figure 3.4: *The four layers of the TCP/IP suite, with the protocols that operate at each.*

Learn the four in order, and learn one protocol that lives at each, because a question asking for the layers nearly always asks for an example in the same sentence.

The Master Key: *Network access, Internet, transport, application, and a protocol you can name for every one of them.*

BINDER Question 19

Two maps of Stone Town, drawn by different surveyors, gather the same streets into differently named wards. Neither map is wrong. A visitor who takes the ward from one and the street from the other will give an address that exists on neither.

Using Figure 3.5, describe how the four layers of the TCP/IP suite correspond to the seven layers of OSI.

Model Answer

The correspondence is exact in the middle and compressed at both ends: transport answers to OSI layer 4 and Internet to OSI layer 3, while the TCP/IP application layer absorbs OSI layers 5, 6 and 7 together, and network access absorbs layers 1 and 2.

One to one in the middle: transport and OSI layer 4 hold the same responsibility under the same name, and the Internet layer holds the responsibility OSI gives to layer 3. These two need no translation beyond the change of name at the lower of them.

Three into one at the top: session, presentation and application become a single application layer, because the protocols that were actually built never divided those duties. A protocol such as HTTP conducts its own dialogue and settles its own encoding internally, leaving nothing to occupy two separate boxes beneath it.

Two into one at the bottom: physical and data link become network access. The designers declined to say how a frame crosses a single link, so that IP could ride over any technology already built or still to be invented, and one undivided box was the result.

Every OSI layer covered once: check the arithmetic before you leave the diagram, since three plus one plus one plus two is seven. Nothing is omitted, nothing is counted twice, and no TCP/IP layer stops halfway up an OSI layer. An answer showing Internet straddling layers 3 and 4 fails on the diagram alone.

Reading the bracket: each bracket on the right of Figure 3.5 spans whole boxes on the left and never part of one. Say so in an answer, because that property is what makes this a mapping rather than a loose resemblance.

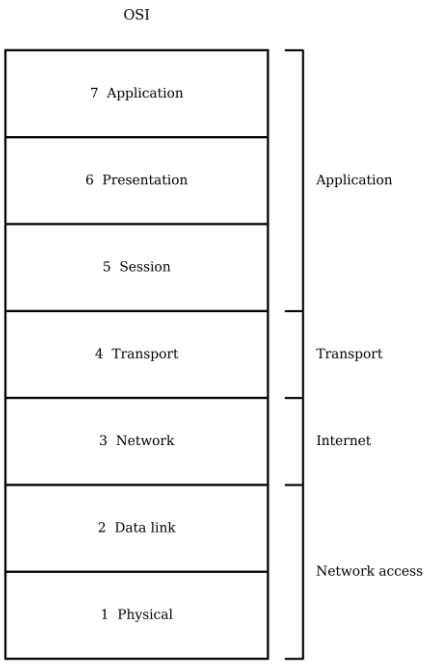


Figure 3.5: *The TCP/IP layers mapped onto OSI. Every OSI layer is covered exactly once, and no TCP/IP layer spans a gap.*

Draw the seven on the left, bracket them into four on the right, and you have answered every mapping question this topic can set.

The Master Key: *Three into one on top, one to one in the middle, two into one at the bottom.*

BINDER Question 20

Every bid for the district network at Bariadi answers the tender the same way: our equipment follows TCP/IP. A fortnight later the same engineers describe the same switches in the language of seven layers. The procurement officer cannot see why both are true.

Give two advantages and two disadvantages of the TCP/IP model when it is compared with OSI.

Model Answer

TCP/IP wins on realism and on simplicity, because every layer it names is defined by protocols in daily use and there are only four of them, and it loses on rigour, because it never separates service from protocol and its bottom layer is not really a layer at all.

Advantage, protocols that exist: the model was abstracted from software that was already carrying traffic, so each of its four layers can be filled with named, working protocols. The seven-layer model was standardised in the opposite order, and the protocols written to fill it never reached the market.

Advantage, simple and extensible: four layers are easier to teach and easier to implement than seven, and because everything meets at IP in the middle, a new medium can be added below or a new application above without disturbing anything else. The specifications are published openly, so no supplier owns them.

Disadvantage, service and protocol blurred: OSI keeps the service a layer offers, the interface through which it is requested and the protocol that delivers it as three separate ideas, so one may be changed without the others. TCP/IP blurs all three, which makes it a poor guide for designing anything new.

Disadvantage, descriptive not prescriptive: it faithfully records one protocol family and cannot properly describe any other, so it offers no standard against which a fresh design could be judged. OSI, whatever else is wrong with it, was written to apply to networks nobody had built yet.

Disadvantage, the bottom layer: network access folds two unrelated jobs into one box, an objection serious enough to have a question of its own next.

Feature	OSI	TCP/IP
Number of layers	Seven	Four
Origin	Published by ISO in 1984 as a general reference model	Abstracted from defence-funded research software of the 1970s
Prescriptive or descriptive	Prescriptive; it states what a conforming stack ought to look like	Descriptive; it records the shape of a suite already running
Service and protocol	Kept firmly apart, so either may be replaced alone	Run together, so the layer and its protocols are hard to separate
Real implementations	Its own protocols were never adopted in the market	Every layer names protocols in daily use
The lower layers	Physical and data link kept as two distinct layers	Both folded into a single network access layer

Table 3.3: *OSI and TCP/IP compared.*

Notice that the advantages and the disadvantages are the same two facts read twice: the model was copied from working software, which is why it is realistic, and why it is not a standard.

The Master Key: *TCP/IP describes what exists; OSI prescribes what ought to, and each is weak exactly where the other is strong.*

BINDER Question 21

Between two classroom blocks in Mbinga a copper run becomes a fibre pair and nothing else alters: the same switches, the same addresses, the same behaviour above. At a second school the wired links give way to Wi-Fi, and a great deal alters.

Explain why the network access layer of the TCP/IP model is criticised for combining two distinct responsibilities.

Model Answer

The network access layer covers both the physical transmission of bits and the logical assembly of those bits into addressed frames, and these are independent jobs with different standards, different faults and different rates of change, so folding them together hides the boundary a network manager most often needs to point at.

The two jobs inside one box: one job is signalling, that is the connector, the voltage or the light or the radio carrier, and the timing of each bit. The other is framing and hardware addressing, error detection and deciding who may transmit next. OSI gives each of these a layer of its own.

Why the boundary is real: the Mbinga upgrade proves it. An Ethernet frame keeps the same format whether it travels over copper or over fibre, so the medium was replaced and the frame was not. A model with one box cannot express why that substitution was safe.

Why fault-finding suffers: a crushed connector and a duplicated hardware address are both network access faults under this model, yet nothing about diagnosing them is shared. One is found with a cable tester and the other by reading an address table, and lumping them together loses that distinction.

Why the designers did it: the omission was deliberate. TCP/IP was built so that IP would run over any link technology whatever, and the surest way to guarantee that was to specify nothing below IP. What served the designers as freedom reaches the student as vagueness.

Stating the criticism precisely: the accurate objection is not that the layer is wrong but that it is not a layer at all. It is a place where the model stops and hands over to whatever technology is present, which is why the five-layer compromise splits it first.

That is why so many practical questions are set at the join: almost everything a technician touches lives inside that one undivided box.

The Master Key: *One box, two trades: the signal is not the frame, and only OSI keeps them apart.*

REAL Question 22

Ten pages of notes on the network layer sit before a Form Five student in Temeke: logical addressing, routing, ICMP, the lot. She is asked for two functions of the Internet layer, has never met the phrase, and assumes it concerns browsers.

Explain to her whether the Internet layer and the network layer are the same thing, and advise how to answer safely when a question does not say which model it means.

From her notes: "The network layer chooses a route across the internetwork and places the source and destination IP addresses in the header."

From the question paper: "State two functions of the Internet layer of the TCP/IP model."

From a classmate, helpfully: "The Internet layer is the one where the browser talks to the website."

Model Answer

They are one layer under two names: the Internet layer of the TCP/IP model is the network layer of OSI, layer 3, so every line of her notes answers the question in front of her exactly, and she may write them out as they stand.

Why there are two names: two models were developed separately and each named its own layers. TCP/IP called its third layer from the bottom after the job it does, joining separate networks into an internetwork. OSI called the equivalent layer the network layer. Neither name is a mistake, and neither is more correct.

The test that settles it: identify the layer by its protocol and its device rather than by its label. If logical addresses are being added, if datagrams are being routed from one network into another, and if the device concerned is a router, then it is layer 3 whatever the paper chooses to call it.

Correcting the classmate: the word Internet there means internetworking, the joining of separate networks, and has nothing to do with browsing. The conversation between a browser and a web server is HTTP, an application-layer protocol three layers higher. That single misreading turns a correct set of notes into a wasted answer.

The safe opening clause: when a question names a layer but not a model, name the model yourself in the opening line, thus: "In the TCP/IP model this is the Internet layer, which corresponds to layer 3, the network layer, of OSI." One clause, no risk, and it cannot be marked wrong under either vocabulary.

Where else the trap is laid: application means OSI layer 7 alone in one model and layers 5, 6 and 7 together in the other, so two functions of the application layer has two different correct answers. Sharpest of all, the network layer and the network access layer are three layers apart despite sharing a word.

One reliable signal: a number almost always means OSI, because TCP/IP layers are named and rarely numbered. If a question says layer 3, answer in OSI terms; if it says Internet layer, answer in TCP/IP terms and add the OSI equivalent in a clause.

She loses nothing by writing both names and everything by guessing which one is meant, so the habit to build is simply this: never write a layer name without first saying whose layer it is.

The Master Key: *Name the model before you name the layer, and the vocabulary can never catch you.*

REAL Question 23

The ICT committee of a secondary school in Makambako is shown two sets of training notes. One teaches four layers and matches the labels printed on the equipment in the room. The other teaches seven. Why learn the one that lost, a member asks.

Advise the committee why the model that lost the contest to become the working standard is still the one on which their students will be examined.

Model Answer

OSI lost as a set of products and won as a language, so the committee should have both taught: TCP/IP because it is what their equipment runs, and OSI because its seven layers are the vocabulary in which the questions, the fault reports and the suppliers' own descriptions of that equipment are all written.

How the contest actually ran: the TCP/IP design was published in 1974 and the whole of the research network it was written for moved onto it on the first day of 1983. ISO published the seven-layer model the following year, in 1984. Working software had a ten-year start, and committee papers could not close it.

Why the rival protocols failed: the OSI protocols were specified in full before much of anything had been implemented, and they arrived late, large and expensive. Several governments required them for a period and then quietly stopped, because free and working implementations of the other suite were already installed everywhere that mattered.

What OSI won instead: it became the industry's common vocabulary. A layer-2 switch, a layer-3 boundary, a layer-1 fault: these phrases are used every day by engineers whose equipment runs nothing but TCP/IP, and they are precise because seven layers give seven distinct places to stand.

Why four layers cannot supply that language: a model with fewer boxes can name fewer things. With physical and data link folded together there is no separate word for the broken cable and the wrong hardware address, and with the top three folded together there is no way to say that only the encoding is at fault.

The advice to the committee: teach the seven for naming and the four for what genuinely runs, and put both stacks on one poster in the laboratory with the mapping drawn between them. Their students will diagnose in the language of one model and be marked in the language of the other.

The general lesson worth stating: a standard can fail commercially and still succeed intellectually. OSI is examined not because anyone expects to meet its protocols, which they will not, but because thinking in layers is the skill, and seven layers teach that skill better than four.

Answer the committee member directly: the model did not lose an argument about how networks should be understood, it lost a race to be installed, and only the first of those two contests is what a school is preparing its students for.

The Master Key: *OSI lost the market and kept the dictionary, which is why you will be paid in one model and examined in the other.*

HOT Question 24

In under a minute an engineer in Ilala and a supplier abroad who has never seen her equipment agree that layer 1 is clean, the fault sits at layer 2, and nothing above it need be touched. Neither has run a line of that model's software.

Assess the claim that OSI is a purely theoretical model with no practical value, and reach a judgement.

Model Answer

The claim is true of the OSI protocols and false of the OSI model, and since specifying, purchasing and diagnosing equipment are practical activities that are carried out in its vocabulary every working day, the claim as stated must be rejected.

The case for, on the protocols: the suite designed to fill the seven layers was never adopted. No school, bank or ministry network in the country runs a single one of those protocols, and a specification that ended with no installed base is as fair a definition of theoretical as one could ask for.

The case for, on the boundaries: no operating system draws all seven lines. Inside a real host, layers 5, 6 and 7 are one body of code within the application itself, so three of the seven boundaries the model insists upon are not present in any machine the student will ever use.

The case for, on the fit: a model claiming generality ought to fit the dominant case comfortably, and this one does not. Mapping it onto the suite that actually won requires five of its seven layers to be collapsed into two, which is an awkward result for a general reference model.

The case against, on the vocabulary: equipment is sold, specified and escalated by layer number. A layer-3 switch, a layer-4 access rule, a layer-1 problem: these are commercial and operational terms, they appear on quotations and in support tickets, and the numbering they depend on comes from this model and from nowhere else.

The case against, on diagnosis: the model supplies an order of work. Test the cable, then the link, then the address, then the port, then the application, and stop at the first layer that fails. That procedure saves hours on a live fault, and the four-layer model is too coarse to generate it.

The case against, on real standards: the standards that do govern working equipment define their own scope in OSI terms, declaring themselves to cover the physical layer and the data link layer. A model being used to bound live standards is producing practical output, whatever became of its own protocols.

The judgement: reject the claim. It confuses a model with a product, and asks a reference framework to justify itself by shipping. The narrow version is true and should be conceded plainly, that the OSI protocol suite has no practical value; the wide version is refuted every time an engineer says the fault is at layer 2 and is understood.

Note the shape of the answer, because it recurs: concede the strongest version of the objection in one sentence, then show that it was aimed at something other than the thing being defended.

The Master Key: *A model earns its keep by what it lets people say to each other, not by what it ships.*

HOT Question 25

Neither seven nor four: a teacher in Sumbawanga draws five boxes on the board and calls them the layers. The student who learned seven counts them twice. The student who learned four counts them twice as well. Both are right to be uneasy.

Judge whether the five-layer hybrid model is a sensible compromise between OSI and TCP/IP or an unnecessary third standard.

Model Answer

The hybrid is the most honest picture of how a real stack is built and the most dangerous thing to write in an answer, so the judgement is to use it for understanding and never for wording: think in five, answer in seven, and translate into four when asked.

What the hybrid is: five layers: physical, data link, network, transport and application. It takes the lower split from OSI and the upper collapse from TCP/IP, keeping every distinction that real protocols observe and discarding the two that none of them does.

The case for it, below: separating physical from data link restores a boundary that exists in equipment. A fibre upgrade changes one and leaves the other untouched, and a technician needs a word for each. On this point the hybrid corrects the weakest feature of the four-layer model.

The case for it, above: no deployed protocol has a separate session entity or a separate presentation entity. Both jobs are done inside application protocols, so the hybrid removes two boxes that are permanently empty. Deleting an empty box is not a simplification; it is a correction.

The case against, on authority: it is nobody's standard. The seven layers were published by an international standards body and the four are described in the Internet's own specifications, but the five are a teaching convention with no document behind them, so an answer written in them can cite nothing.

The case against, on the numbers: this objection is decisive. Number the hybrid and its application layer becomes layer 5, whereas layer 5 in OSI is the session layer. A phrase such as a layer-5 problem would then mean two incompatible things, and ambiguity in a layer number is the one defect a layered model cannot afford.

The case against, on the cost: students already confuse the Internet layer with the network layer, as the previous question showed. Adding a third set of names to a topic whose commonest failure is already a naming failure makes the examinable problem worse, not better.

The judgement: adopt it as a mental model and refuse it as a vocabulary. As a description of a working stack the five layers are better than either standard, which is why so many teachers reach for them; as an answer they carry no authority and a colliding numbering. If you meet it, name it as a hybrid, state which model each layer came from, and never quote a hybrid layer by number.

The compromise is sensible about networks and unwise about language, and since a reference model exists to fix language, that division is what settles it.

The Master Key: *Understand in five, write in seven, translate into four, and number nothing that no standards body ever numbered.*

Encapsulation, de-encapsulation and protocol overhead

A message does not travel down a stack unchanged. Each layer wraps whatever it is handed, writes in front of it the control information its opposite number will need, and passes the result down; the receiving host undoes the wrapping in the reverse order. This section settles what each wrapper contains, what the whole set of them

costs in bytes, and how to turn that cost into a percentage. It assumes you can already name the seven layers and say what each is for. The arithmetic here is the chapter's first, so every division is written out before its answer.

BINDER Question 26

Coffee leaving a cooperative store at Mbinga is weighed into a sack, the sack tagged with the grower's number, the tagged sacks loaded under one manifest, the lorry sealed. Four wrappings, each read by a different person at the other end.

Define encapsulation and de-encapsulation, and name the protocol data unit that each layer of the stack produces.

Model Answer

Encapsulation is the process by which each layer on the sending host takes the unit handed down to it, treats all of it as data and prepends its own header, so that data becomes a segment, then a packet, then a frame, then bits; de-encapsulation runs the same process backwards on the receiver.

What a layer adds: a layer never alters what it is given. It places that unit whole inside its own and writes in front of it a header of control information for its opposite number at the far end. Only the data link layer also adds a trailer.

The five names in order: the application, presentation and session layers all handle data, often called a message. The transport layer produces a segment, or a datagram under UDP. The network layer produces a packet, the data link layer a frame, and the physical layer bits.

Why the name changes: the name records which header is outermost, and therefore which layer is in charge at that instant. Put an IP header in front and the object is a packet, whatever it was a moment before.

Unwrapping in reverse: the physical layer collects bits and hands up a frame. The data link layer checks the frame check sequence, strips its header and trailer and hands up a packet. The network layer strips the address header, and the transport layer strips the port header and delivers data to the waiting program.

Every unit is a PDU: protocol data unit is the general term and it applies at every layer without exception. The frame is the data link PDU and the packet the network PDU, so reserving the word for the upper layers alone costs marks.

The wrapping and the unwrapping are one operation seen from opposite ends of a link.

The Master Key: *Name the outermost header and you have named the unit, because those are always the same answer.*

BINDER Question 27

On a repair bench in Ilala a cut-away drawing is pinned above the tools: a frame opened out, the packet inside it, the segment inside that. Every apprentice asks why there are three headers at the front and one small block at the back.

Using Figure 3.6, state what the transport, network and data link headers each contribute, and identify the one layer that adds a trailer as well as a header.

Model Answer

The transport header carries port numbers naming the program at each end, the network header logical addresses that hold good for the whole journey, and the data link header hardware addresses good only across the next link; only the data link layer adds a trailer, the frame check sequence.

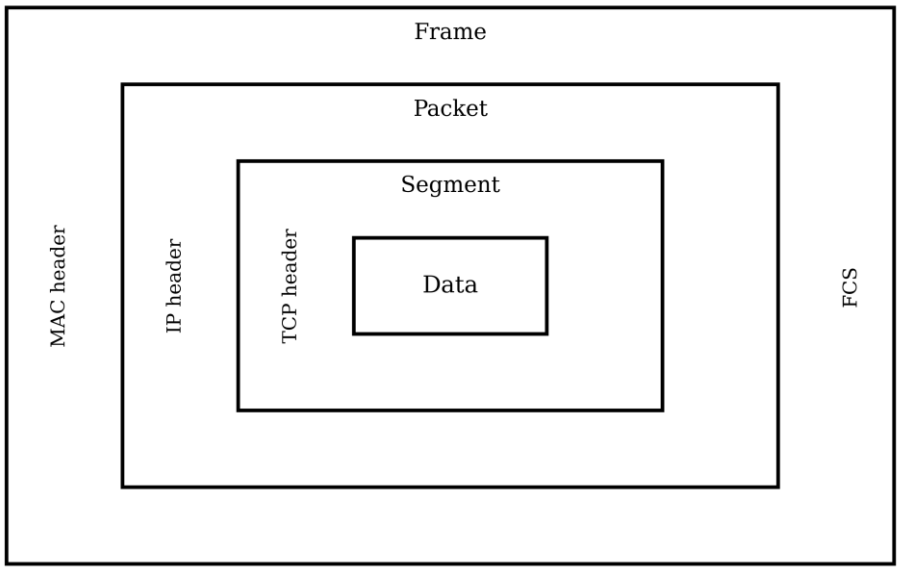
The transport header: twenty bytes for TCP. It holds the source and destination port numbers, which say which program on each host the data belongs to, and the sequence and acknowledgement numbers a receiver uses to restore order and ask again for what never arrived.

The network header: twenty bytes for IPv4 without options. It holds the source and destination logical addresses, and these do not change anywhere on the journey, which is what lets a packet cross networks that have never heard of one another.

The data link header: fourteen bytes for Ethernet. It holds hardware addresses, and unlike the logical ones these are strictly local: they name the two devices at the ends of this one link, and a router replaces them before the packet goes on.

The single trailer: only the data link layer appends anything behind the payload, and what it appends is the four-byte frame check sequence. No layer above has anything left to say once the data has passed.

Reading the figure outwards: start at the centre. Data with a TCP header in front is a segment; add an IP header and it is a packet; add a hardware header and a check sequence and it is a frame. Every box is the whole of the box inside it plus one layer's control information.



Each layer wraps the one above it. Only the data link layer adds a trailer.

Figure 3.6: *Encapsulation. Each layer wraps the unit above it, and only the data link layer adds a trailer as well as a header.*

Three headers and one trailer is not carelessness in the drawing; it is the only arrangement in which a check value can cover everything before it.

The Master Key: *Ports name the program, logical addresses name the host, hardware addresses name the next device, and only the last of the three adds a trailer.*

BINDER Question 28

At the fish market in Pangani a buyer counts a crate, writes the total on a slip, and pins it to the crate after the last fish is in. No total goes on a lid before packing. The frame check sequence obeys that rule.

Explain the purpose of the frame check sequence, and account for its position at the end of the frame rather than inside the header.

Model Answer

The frame check sequence is a four-byte value the sender computes over the contents of the frame and the receiver recomputes and compares, so that corruption picked up on the medium is detected and the damaged frame discarded; it sits at the end because a value computed from the data cannot exist before the data does.

What the four bytes hold: a cyclic redundancy check result: the frame treated as one long binary number and divided by an agreed divisor. The value depends on every bit it covers, so altering any one of them alters it.

How the receiver uses it: it repeats the calculation on the bytes that arrived and compares. Agreement means the frame is almost certainly intact; disagreement means at least one bit changed in flight, and the frame is dropped without repair and without complaint.

Why it cannot go in front: the value comes from the data, so the data must exist first. A check field in the header would force the sender to hold the whole frame in memory, compute, then go back and fill the gap it had left: slower, and costlier in storage on every network card built.

One pass at each end: with the field at the back the sender calculates as it pushes bytes onto the wire and writes the result last, while the receiver feeds each arriving bit into the same calculation and has its verdict ready the instant the last bit lands.

Why only layer two needs one: the layers above hand down units that nothing has yet threatened. Only the data link layer must certify what actually crossed a physical medium, and the medium is where corruption happens.

A trailer is not decoration on the back of a frame; it is the one field that could not have been written earlier.

The Master Key: *Anything computed from the whole must be written after the whole, which is why the check follows the data and never precedes it.*

BINDER Question 29

The inspector on the weighbridge outside Sumbawanga never asks what a lorry carries. He asks what it weighs, and a lorry over the axle limit shifts part of its load to a second vehicle or turns back. Every network link works the same way.

Define the maximum transmission unit, and describe what happens to a packet larger than the maximum transmission unit of the link it must cross.

Model Answer

The maximum transmission unit is the largest packet, in bytes, that a link will carry inside one frame, and it is 1,500 bytes on standard Ethernet; a packet larger than it must either be fragmented into pieces that each fit or be discarded and the sender told.

What the number measures: the unit counts the network-layer packet, not the frame wrapped round it. Add the fourteen-byte Ethernet header and the four-byte check sequence to the 1,500, that is 1,500 + 14 + 4 = 1,518 bytes, and you have the largest frame the standard permits.

Why a limit exists at all: a long frame holds the medium longer and delays everything queued behind it. It also offers more bytes for interference to spoil, and since the check covers the whole frame, one damaged bit costs the entire frame.

Fragmentation: where a packet is too large, an IPv4 router may cut its payload into pieces small enough to fit, giving each a copy of the header with a flag saying more follow and an offset saying where the piece belongs. Only the destination host reassembles them.

What fragmenting costs: every fragment pays a fresh twenty-byte header, so overhead rises. Worse, the pieces are all or nothing. Lose one fragment and the destination can never rebuild the original, so the entire packet has to be sent again.

Choosing a size that fits: a sender does better to fit from the start. On Ethernet, TCP takes the 1,500-byte limit and subtracts its own twenty bytes and the twenty of the IP header, so 1,500 − 20 − 20 = 1,460 bytes is the largest segment it will offer. Path discovery extends the idea across a whole route.

Fragmentation is a rescue rather than a feature, and a link that fragments steadily is a link whose senders have been told the wrong size.

The Master Key: *Subtract the headers from the limit before you send, because a packet cut up on the road arrives late or not at all.*

REAL Question 30

A college in Mikocheni buys its Internet capacity by the gigabyte, and the bursar asks why the invoice figure is always larger than the librarian's figure for files downloaded. Nobody is stealing anything. The difference can be worked out to two decimal places.

Calculate the percentage of a maximum-size Ethernet frame that is protocol overhead when it carries 1,460 bytes of application data, showing every step of the working.

Application data carried in the frame: 1,460 B

Ethernet header: 14 B

IP header: 20 B

TCP header: 20 B

Frame check sequence: 4 B

Model Answer

The overhead totals 14 + 20 + 20 + 4 = 58 B, the whole frame totals 1,460 + 58 = 1,518 B, and the overhead is therefore 58 / 1518 = 0.0382, which is 3.82 per cent to two decimals.

Add the headers first: the bytes that are not application data are the Ethernet header at 14 B, the IP header at 20 B, the TCP header at 20 B and the frame check sequence at 4 B. Their sum is 14 + 20 + 20 + 4 = 58 B, and that figure does not move when the payload does.

Then the whole frame: put the payload back in: 1,460 + 58 = 1,518 B. That is the largest frame standard Ethernet will carry, which is exactly why 1,460 B is the most application data one frame can hold once TCP and IP have taken their share.

Divide before you multiply: overhead as a percentage is the non-data bytes divided by the total bytes, times one hundred. So 58 / 1518 = 0.03821, and 0.03821 multiplied by 100 gives 3.82 per cent to two decimals. Write the division on the page before the answer, because the division is where the marks sit.

$$\text{Overhead} = 58 / 1518 = 0.0382 = 3.82 \text{ per cent}$$

Check from the other side: the useful fraction has to make up the remainder. 1460 / 1518 = 0.9618, that is 96.18 per cent, and 3.82 + 96.18 = 100 exactly. A pair of percentages that fails to complete to a hundred has an error in it somewhere.

Why this is the best case: the 58 B is fixed while the payload varies, so this is the smallest overhead Ethernet can produce. Every frame carrying less than 1,460 B pays the same 58 B on fewer useful bytes, and the percentage climbs.

What the bursar is buying: roughly four bytes in every hundred on that invoice are addresses, ports and a check value rather than anything a student asked for. No setting will remove them, and a college whose transfers are large is already paying the lowest rate the technology allows.

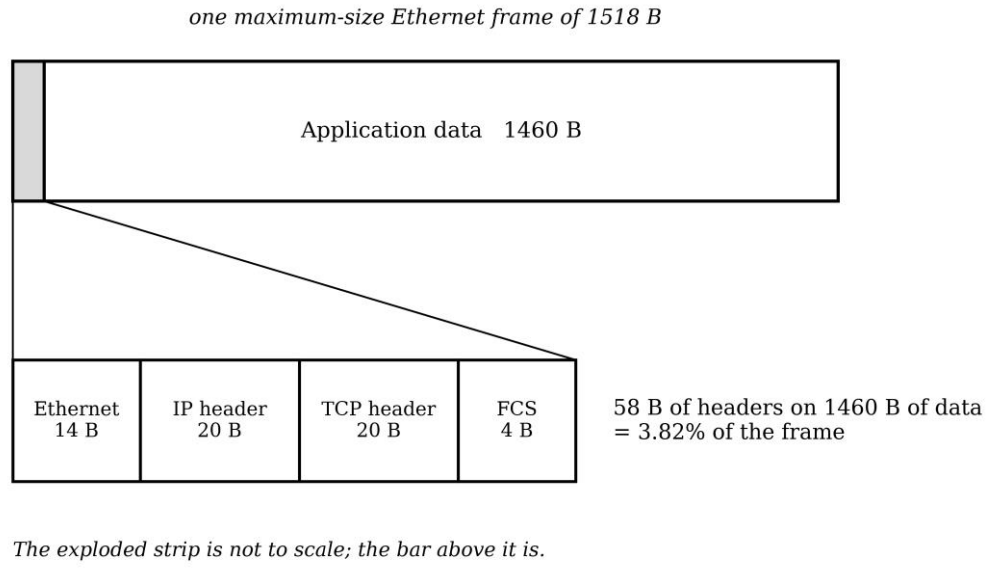


Figure 3.7: *Where the bytes go in a maximum-size Ethernet frame. The exploded strip is not to scale; the bar above it is.*

3.82 per cent is what it costs to have every layer keep its own books, and while the frames stay full it is remarkably cheap.

The Master Key: *Sum the headers, sum the frame, divide the first by the second, and never write a percentage without writing its division above it.*

REAL Question 31

A mobile-money agent off a side street in Kariakoo settles twelve hundred transactions on a busy Saturday, each leaving her terminal as a record of twenty bytes. Her bundle is metered by the byte, and she pays for a great deal more.

Calculate the percentage overhead carried by each twenty-byte record, and advise the agent on what she should do about it.

One transaction record: 20 B of application data

Headers on every record: Ethernet 14 B, IP 20 B, TCP 20 B, frame check sequence 4 B

The bundle is charged on every byte the terminal sends

Model Answer

The headers total 58 B against a payload of 20 B, so each record leaves in a frame of 20 + 58 = 78 B of which 58 / 78 = 0.7436, that is 74.4 per cent, is overhead; the cure is not a smaller record but fewer frames, and twenty records sent together bring the figure down to 12.7 per cent.

The same fifty-eight bytes: 14 + 20 + 20 + 4 = 58 B. Nothing about the headers shrinks because the payload is small. The addresses, the ports and the check value are all exactly the size they were on a full frame, because they have exactly the same work to do.

The frame each record makes: 20 + 58 = 78 B leave the terminal for every 20 B the agent actually sends. Twelve hundred records therefore cost 1200 × 78 = 93,600 B against the 1200 × 20 = 24,000 B she believes she is buying.

The division: 58 / 78 = 0.7436, so 74.4 per cent of every byte she is charged for is envelope rather than letter. Roughly three bytes in four carry no information about any transaction at all.

Why a shorter record would not help: Ethernet will not place a frame shorter than 64 B on the wire. On a frame sitting exactly on that floor the same 58 B of headers would be 58 / 64 = 0.9063, that is 90.6 per cent of everything sent. Trimming the payload makes the proportion worse rather than better.

The advice, which is to batch: hold records briefly and send twenty in one frame. The payload becomes 20 × 20 = 400 B, the frame becomes 400 + 58 = 458 B, and the overhead falls to 58 / 458 = 0.1266, that is 12.7 per cent. One set of headers now serves twenty transactions instead of one.

What batching costs her: a record held back is a record not yet at the server, so set a timer of two or three seconds, send whatever has gathered, and exempt anything a customer is standing there waiting for. Delay is the only currency batching spends, and a few seconds of it removes about seven bytes in every ten she was charged for.

Her instinct was sound and her arithmetic optimistic, because a metered link bills the envelope at the same rate as the letter inside it.

The Master Key: *When the payload is small the headers are the bill, so send fewer frames rather than smaller ones.*

REAL Question 32

A card of notes is passed round a Form Five class in Bariadi. It is neat, it is confident, and it is written in a hand the whole room trusts. Three of its lines are wrong, and each survives a quick reading.

Identify what is wrong with each of the three statements on the card, give the correct version, and explain why the correction is right.

1. Every layer adds a header and a trailer to the unit it receives from the layer above.

2. De-encapsulation happens at every router along the path, all the way up to layer 7 and back down again.

3. The receiving host reads the source port number first and then works down to the destination MAC address.

Model Answer

Only the data link layer adds a trailer and every other layer adds a header alone; a router de-encapsulates no further than layer 3 before building a fresh frame for the next link; and reception runs upwards, so the destination hardware address is read first and the port numbers last.

Statement one, the missing asymmetry: every layer above the data link layer prepends a header and nothing else. A trailer earns its place only when it carries a value computed over everything in front of it, and the frame check sequence, four bytes of it, is the only such field in the stack. Symmetrical boxes make a tidy diagram and a false one.

Statement two, how far a router opens: a router strips the frame header and trailer, reads the destination address in the IP header, decrements the time-to-live count, and then builds an entirely new frame with new hardware addresses and a newly computed check for the next link. It never touches the transport header. Only the destination host climbs to layer 7.

Why that limit is deliberate: a router reading layer 7 would have to understand every application protocol that exists or ever will, and would need replacing whenever a new one appeared. Forwarding is kept at layer 3 precisely so that the middle of the network can stay simple, fast and ignorant.

Statement three, the direction reversed: the network card first checks the destination hardware address to see whether the frame is even meant for it, then the check sequence, then the network layer reads the destination logical address, and only after all that does the transport layer look at ports. The source port is among the last fields read, not the first.

What the three have in common: each is the wrapping diagram read in one direction and then applied in the other. Repair the habit rather than the three facts: before stating what happens to a unit, name the layer holding it and the direction it is travelling, and the wrong answer stops being available.

Nothing on that card was invented. Every line is a true statement that has drifted one layer, or one direction, from where it belonged.

The Master Key: *State the layer and the direction before you state the action, and two thirds of these mistakes become impossible.*

HOT Question 33

The teacher in charge of ICT at a secondary school in Kondoa reads that 9,000-byte frames will make the network faster, and finds the setting in the switch menu, one click away. The saving is real, and so is the fault it would introduce.

Judge whether enabling 9,000-byte jumbo frames on the school network would be worthwhile, weighing the overhead recovered against the problems the change would create.

Model Answer

Jumbo frames would cut overhead from 3.82 per cent to 0.64 per cent, about three bytes in every hundred, but they work only if every device in the broadcast domain agrees, and on a school network of mixed switches, access points and an uplink fixed at 1,500 bytes that recovery is invisible while the fault it risks is not: do not enable them across the general network.

What a jumbo frame is: a maximum transmission unit of 9,000 B in place of the standard 1,500 B, so one frame carries six times the payload. It is the same Ethernet with the ceiling raised, and it lies outside the standard, which is why support for it varies.

The saving, computed: a 9,000 B payload gives a frame of 9,000 + 14 + 4 = 9,018 B carrying the same 58 B of headers, so overhead is 58 / 9018 = 0.0064, that is 0.64 per cent, against 3.82 per cent on a standard frame. The recovery is 3.82 − 0.64 = 3.18 percentage points.

The saving nobody computes: the larger gain is not bytes but interruptions. One frame in place of six means a sixth as many headers to parse and a sixth as many interrupts to service, which matters on a busy file server and hardly at all on a classroom workstation.

Everything must agree: a device expecting 1,500 B discards a 9,000 B frame. Configure some switches and not others, or overlook an access point, and short exchanges work perfectly while large transfers stall, because a connection is established with small packets long before it sends a big one. The symptoms then point everywhere except at the cause: nothing is down, no indicator is red, and whoever diagnoses it must already suspect the frame size to find it.

What the traffic actually is: a school sends web pages, mail and documents to a shared drive, and most of those packets never approach 1,500 B, so the raised ceiling would go largely unused. The path to the outside world is fixed at 1,500 B in any case, so every jumbo frame stops at the router.

The judgement: leave the setting alone on the general network. Recovering 3.18 percentage points that nobody could perceive is not worth a class of fault that can consume a week of term. If jumbo frames are wanted at all, confine them to an isolated segment between the file server and the backup machine, on managed switches, with every port on it under one person's control.

The click is easy, and that is the whole danger: the benefit arrives the same afternoon and the bill a month later, dressed as something else.

The Master Key: *Enable a setting only where you can name every device that must agree with it, and a school network is exactly where you cannot.*

HOT Question 34

A cargo agent on Mafia keeps two hundred vehicles and cold rooms reporting to a mainland server, and the only route off the island is metered satellite capacity. The messages are tiny, tens of thousands a day. The bill is not tiny.

Assess where the greatest reduction in traffic would come from on a link carrying mostly very small packets, and commit to a single recommendation.

Proposal A: raise the maximum transmission unit on the link.

Proposal B: compress the payload of each message.

Proposal C: compress the TCP and IP headers across the link.

Proposal D: collect several messages and send them in one packet.

Model Answer

Proposal D wins by a clear margin: each 20-byte message travels in a 78 B frame of which 58 B is header, so gathering ten messages into one frame turns 10 × 78 = 780 B into 200 + 58 = 258 B, a cut of 66.9 per cent against 44.9 per cent for header compression and nothing at all for the other two; aggregate in the application, on a short timer.

Proposal A saves nothing: the packets are small because the messages are small, not because a ceiling is cutting them short. A maximum transmission unit is an upper limit, not a target, and raising a limit nothing is reaching changes not one byte on the link.

Proposal B saves almost nothing: twenty bytes of a fixed-format status record hold very little repetition for a compressor to work on, and a compressor writes framing of its own around what it produces. On payloads this short the output is often longer than the input.

Where the bytes actually are: 58 B of headers against 20 B of message. Three quarters of the traffic is envelope, so any proposal worth the money must either shrink the envelope or cut the number of envelopes. The first two attack the letter, which was never the problem.

Proposal C, a smaller envelope: across a point-to-point link almost every field of the TCP and IP headers repeats from one packet to the next, so a compressor at each end can carry 40 B of header in roughly 5 B. That saves 35 B on each 78 B frame, 350 B in every 780 B, which is 44.9 per cent.

What proposal C costs: matching equipment at both ends, and a saving that ends at the first router, since the headers must be rebuilt in full before the packet can go further. On a satellite hop that is tolerable, but it is still hardware and a support contract.

Proposal D, fewer envelopes: ten messages in one packet carry 10 × 20 = 200 B of data under a single set of headers, so 200 + 58 = 258 B replaces 780 B. That is 522 B saved in every 780, or 66.9 per cent, won inside the application's own code with no new equipment at either end.

The judgement and its price: aggregate, on a timer of about five seconds so that a quiet period cannot strand a message, and exempt alarms so that a failing cold room still reports at once. Revisit header compression only if the link is still congested afterwards, because that costs money while this costs a fortnight of programming.

Two of the four proposals set out to shrink the message when the message was never the problem; the one that worked stopped repeating the address.

The Master Key: *Count the headers before the payload, because on a small-packet link the answer is nearly always fewer packets rather than shorter ones.*

The physical layer: signalling, encoding and impairment

Every layer above this one moves ideas: addresses, frames, sessions, files. The bottom layer moves electricity, light and radio, and it is the only place in the stack where a one and a zero stop being abstractions and become something an instrument could measure. This section settles what a physical-layer standard must fix, how a bit is turned into a signal that carries its own timing, and what the journey does to that signal on the way. It assumes you can already name the layers in order.

BINDER Question 35

A handcart man in Kariakoo moves whatever is loaded onto his cart, and has never opened a sack in his life. Ask what he carried this morning and he gives you the weight and the distance, nothing else. That refusal is not ignorance.

State what the physical layer is responsible for, and state what it deliberately knows nothing about.

Model Answer

The physical layer turns bits into a signal the medium can carry, puts that signal on at an agreed rate and recovers the bits at the far end, and it knows nothing of what the bits mean, so it recognises no address and no frame boundary.

What it puts on the medium: a physical representation of each bit: a voltage on a conductor, the presence or absence of light in a fibre, a change in a radio carrier. The layer fixes how a one is told from a zero and how long each bit lasts.

The unit it handles: the bit, and nothing larger. This layer never groups bits and never asks where one message ends and the next begins. A thousand bits and one bit are the same kind of work to it.

What it refuses to know: there is no addressing here, no error detection, no retransmission and no destination beyond the far end of this link. The signal reaches whoever is listening, and deciding whether it was meant for them belongs above.

Why the ignorance pays: the same signalling rules carry a voice call, a payroll file and a backup without alteration, and the traffic may change entirely without one change to the wire. A layer buys that freedom by refusing to look upward.

The confusion to avoid: the physical layer is not the cable. The cable is the medium; the layer is the agreement about how a bit becomes a signal on it, and one medium can carry several agreements at several speeds.

The honest picture is two translators facing each other across a link, one turning bits into a physical disturbance and the other turning it back into bits, neither curious about the cargo.

The Master Key: *Ask whether a question could be answered without knowing what the bits mean; if it could not, it is not a physical-layer question.*

BINDER Question 36

The consignment that reached a technical college in Kahama came with plugs that fitted the sockets and voltages that measured as the datasheet promised. Nothing worked. The two makers had assigned the pins differently. A connector that fits is one agreement out of four.

Describe the four characteristics that a physical-layer standard must define, and give one example of each.

Model Answer

A physical-layer standard is complete only when it fixes four separate things: the mechanical characteristics of the connector and the medium, the electrical characteristics of the signal, the functional meaning given to each circuit, and the procedural rules governing the order of events.

Mechanical: the physical shape of the interface: the dimensions of the plug, the number of pins, the way the socket latches and how the medium attaches to it. An eight-pin RJ-45 plug is the familiar example, and it is why equipment from two makers can be joined at all.

Electrical: the properties of the signal itself: the voltage or optical power representing each state, the duration of one bit, the signalling rate, and the threshold at which the receiver decides it has seen a one rather than a zero.

Functional: the meaning attached to each pin or circuit. Two pins are the transmit pair and two are the receive pair, and nothing electrical says so; it is purely an agreement. The Kahama consignment failed on this characteristic alone, with the other three satisfied.

Procedural: the sequence: what must happen before what. Two ports negotiate speed and duplex before either sends data, and each waits a defined time for the other to answer. Rules about order, not about volts.

Why four and not one: each can be satisfied while the others fail, and each failure has its own signature. A mechanical mismatch will not plug in, an electrical one gives no link, a functional one links and passes nothing, and a procedural one works at times and not at others.

Characteristic	What it specifies	Example
Mechanical	Shape, size and pin count of the connector, and how the medium attaches	The eight-pin RJ-45 plug and its socket
Electrical	Voltage levels, bit duration, signalling rate, the one and zero threshold	A stated voltage range meaning a one
Functional	The meaning assigned to each pin or circuit	Two pins for transmit, two for receive
Procedural	The order of events and what must precede what	Speed and duplex negotiated before data

Table 3.4: *The four characteristics the physical layer defines.*

Learn them as four questions asked of any interface. What does it look like, what does it measure, what does each wire mean, and in what order do the two ends speak.

The Master Key: *A connector that fits proves one characteristic out of four, which is why equipment can be plugged together and still refuse to talk.*

BINDER Question 37

Off Mafia a fishing boat answers the shore with a lamp. On or off, each flash carries one piece of news. Fit a shutter giving four distinguishable brightnesses and the same flashes in the same minute carry twice as much.

Distinguish between bit rate and baud rate, and give an encoding in which the two are not equal.

Model Answer

Bit rate counts the bits of information delivered each second and baud rate counts the signal changes made each second, so they are equal only when one change carries exactly one bit; Manchester differs, because it makes two signalling events for every bit.

Bit rate: the quantity of information crossing the link each second, measured in bit/s. It is what the user is promised, because it decides how long a file takes.

Baud rate: the number of signalling events, or symbols, the transmitter produces each second, measured in baud. It describes the transmitter's hand rather than the news it carries, and the medium must follow it.

The relation between them: bit rate equals baud rate multiplied by the number of bits each symbol carries. When a symbol carries one bit the two numbers are identical, which is exactly why they are so often treated as a single idea.

Where baud exceeds bit rate: Manchester spends two signalling events on each bit, so 10 Mbit/s of data requires 20 Mbaud of signalling. The baud rate is double the bit rate, and buys timing rather than information.

Where bit rate exceeds baud: give a symbol four distinguishable levels rather than two and it carries two bits, since two bits have four combinations. The bit rate is then double the baud rate, which is how fast links keep within what the medium allows.

Which one the medium limits: the medium limits the baud rate, because it limits how quickly a signal may change without smearing into the next. The bit rate is whatever the designer packs into each change, which is why the distinction matters.

State the units and the confusion dies: bit/s counts information and baud counts changes. A link described only in baud has told you the effort and not the result.

The Master Key: *Multiply baud by the bits each symbol carries to get bit rate, and never assume the multiplier is one.*

BINDER Question 38

A link between two machines in a laboratory at Kyela passes alternating patterns perfectly and mangles a file of zeros every time. The cable is new, the connectors sound, and swapping either machine changes nothing. On the oscilloscope the trace lies flat.

Using Figure 3.8, explain how Manchester encoding allows a receiver to recover the clock from the data, and state what the method costs.

Model Answer

Manchester places a transition at the centre of every bit period whatever the data, so the receiver locks its own clock to the signal and never loses count; the cost is two signalling events per bit, which doubles the baud rate.

The problem with NRZ: in the upper trace a one is held high and a zero low for the whole bit period, so at the pair of ones and the pair of zeros the line stays flat. The receiver samples on its own oscillator, never exactly the sender's, and across a flat run that instant drifts into the next bit period.

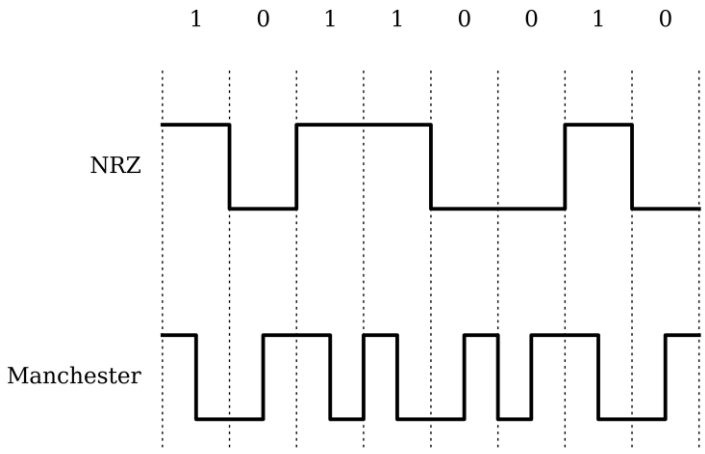


Figure 3.8: *The same eight bits in NRZ and in Manchester encoding. Manchester guarantees a transition in every bit period, so the receiver recovers the clock from the data itself.*

What the lower trace does: every bit period contains a transition at its midpoint. In the convention drawn here a one falls from high to low and a zero rises, so the direction of that transition is the data. Boundary transitions appear only where two like bits meet.

How the clock is recovered: the receiver watches for those mid-bit transitions and nudges its own oscillator to agree with them. One is guaranteed in every bit period, so the correction never stops arriving and the alignment cannot drift. The code is self-clocking.

What that saves: no separate clock conductor is needed, and equal time is spent high and low within each bit, so the signal carries no direct-current component and can pass through a transformer.

The price paid: one bit costs two signalling events, an efficiency of one half. Carrying 10 Mbit/s of data demands 20 Mbaud of signalling, so half the link's capacity buys timing rather than data.

Manchester therefore buys certainty with bandwidth. At ten megabits that was a bargain, because a receiver that never loses its place mattered more than the spectrum it spent.

The Master Key: *A code is self-clocking when the data cannot be arranged so as to hide the beat, and Manchester puts the beat inside every bit.*

REAL Question 39

Twenty-three machines in the laboratory of a secondary school in Chake Chake work all day. The twenty-fourth, at the desk furthest from the door, drops its connection whenever a class runs. Carried across the room to the switch, it behaved perfectly all afternoon.

Identify the impairment responsible for the fault at that desk, and explain the mechanism by which it produces these symptoms.

Model Answer

The impairment is attenuation: the signal loses power along the run to that desk and arrives too weak, against noise that has not weakened at all, for the receiver to tell a one from a zero reliably, and moving the machine beside the switch cures it by removing the length.

What the move proved: the computer, its card and its settings travelled across the room and kept working, so none of them is at fault. The only thing left behind was the run to that desk, and a test that moves one variable beats an afternoon of guessing.

Loss along the way: a signal loses power as it travels. Part turns to heat in the conductor and part leaks away, and the loss compounds along the whole length, so what arrives is a fraction of what was sent. Doubling the run does not add loss, it multiplies it.

Noise that does not fade: noise does not shrink with distance; it is added along the way. So as the signal falls and the noise holds steady, the gap between them closes from both directions at once, and that closing ratio, rather than weakness alone, is what defeats the receiver.

Where the decision fails: the receiver samples the line and compares it against a threshold. While the difference between a one and a zero stays comfortably larger than the noise, every decision is right. Once the noise swing approaches that shrinking difference, some samples fall the wrong side and bits are read wrongly.

The misleading symptom: a marginal link does not fail cleanly. It negotiates, passes light traffic, then loses frames under load, so the machine seems slow and unreliable rather than disconnected. A properly broken run is diagnosed in a minute; a run that is merely too long wastes a term.

Ruling out the neighbours: crosstalk or outside interference would track the route the run takes and the hours when the interfering equipment is on, and would trouble the desks beside it too. This fault tracks length alone and belongs to one desk, which points at attenuation.

The laboratory has not been sold a faulty computer. It has been sold a run whose far end sits outside the margin the standard leaves for loss, and the cure is a shorter path, fresh terminations or a regenerating device part of the way along.

The Master Key: *Attenuation is diagnosed by length, so move the equipment and keep the cable, or keep the equipment and change the cable, but never change both at once.*

REAL Question 40

Four complaints reached the technician at a secondary school in Sengerema in one week, from four different rooms, and he filed all four under a single heading. That is the mistake. Four separate things degrade a signal crossing a building, each cured differently.

Define attenuation, noise, distortion and crosstalk, separating each clearly from the other three, and name for each a place in a school building where it would be the dominant impairment.

Model Answer

Attenuation is loss of strength, noise is unwanted energy added from outside, distortion is a change in the shape of the signal because the medium treats different frequencies differently, and crosstalk is a signal leaking from one conductor into another beside it.

Attenuation: the signal grows weaker with distance and nothing is added to it. It is quoted in decibels, ten times the logarithm to base ten of the received power divided by the sent power, so a loss of 3 dB leaves 0.501 of the power and every further 3 dB halves it again. In the school it dominates on the longest run, from the office to the library.

Noise: energy that was never part of the signal arrives with it. Thermal noise is always present at a low level; impulse noise comes in bursts from motors, fluorescent fittings and switching contacts. It dominates on the run passing the workshop, where errors appear only while the welding set is running.

Distortion: the waveform changes shape because the components making it up do not travel at the same speed or lose the same fraction of their power. Sharp edges smear and one pulse spreads into the period of the next. It dominates on a long run driven at a high signalling rate, and unlike noise it is entirely repeatable.

Crosstalk: the changing current in one conductor induces a current in a neighbour, so a second circuit hears a faint copy of the first. It dominates in the ceiling void where thirty runs are pulled together in one bundle, and at any termination where the pairs were untwisted too far back.

The one-line separation: attenuation subtracts, noise adds, distortion reshapes, and crosstalk borrows from the circuit next door. Three are properties of the link itself and the fourth is a property of its neighbours, which is why crosstalk alone can be cured by moving a cable rather than changing it.

How each betrays itself: attenuation tracks length, noise tracks the clock on the wall and whatever machinery is running, distortion tracks the signalling rate and eases when the rate is lowered, and crosstalk tracks how busy the neighbouring circuit is.

Written down that way, the technician's four complaints are four different repairs: a shorter route, separation from the workshop, a lower rate or better cable, and a bundle loosened with its terminations redone.

The Master Key: *Name the variable the fault follows, because attenuation follows distance, noise follows time, distortion follows rate and crosstalk follows the traffic beside it.*

HOT Question 41

The contractor quoting for a new ward block at a district hospital in Tarime draws a small grey box halfway across the compound: a repeater, he says, and the distance is solved. A rival puts a switch there for four times the price.

Assess whether a repeater or a switch is the better answer to a run that is longer than the standard permits, and justify the choice you make.

Model Answer

Choose the switch. Both devices solve the distance problem the same way, by regenerating the signal rather than amplifying it, so the physical argument does not separate them; what separates them is that the switch also reads each frame's destination and forwards it only where it is wanted.

Why the run fails: beyond the permitted length the signal arrives too weak and too misshapen to be read. Putting a device part of the way along, where the bits are still recoverable, and starting the journey again from there, removes the problem rather than easing it.

Regeneration, not amplification: an amplifier multiplies whatever reaches it, so the noise picked up so far is magnified along with the signal and travels on, growing at every stage. A regenerator decides each bit, recovers the clock, discards the arriving waveform and builds a fresh one from its decisions. Its output is as clean as the original.

The case for the repeater: it is the cheapest device that regenerates, needs no configuration, adds almost no delay, and works on bits alone, so it neither knows nor cares which protocols run above it.

The case against it: it repeats everything. Traffic with no business at the far end still crosses the compound, every station beyond it contends with every station behind it, and it offers no counters and no way of telling where a problem lies.

What the switch adds free: every switch port already contains a receiver and a transmitter that regenerate, so a switch is a repeater that has also learnt to read. It forwards a frame only towards its destination, so the ward block's own traffic never leaves the ward block.

The practical objection: the grey box halfway across the compound needs mains power, an enclosure and somebody to find it in two years' time. A switch at the far end needs power in a building that has it anyway, which quietly reverses the cost comparison.

The judgement: take the switch. The repeater is defensible only where there is genuinely nothing beyond the far end but one device, and even there it buys reach at the cost of a shared and undiagnosable link. Where the distance is very great, converting to a medium with more reach is the better spend.

The distinction worth carrying away is not repeater against switch but regeneration against amplification. Amplify a tired signal and its errors travel on forever; regenerate it and they stop at that box.

The Master Key: *Amplification enlarges the noise with the signal, so any device worth putting in the middle of a link must decide the bits before it sends them on.*

HOT Question 42

In a design review at an internet provider's office in Mikocheni a junior engineer asks a fair question. The scheme that carried Ethernet at ten megabits guarantees a transition in every bit and needs no clock conductor. Why was it not carried forward?

Judge whether Manchester encoding would be a sensible choice for a gigabit link, and defend your answer.

Model Answer

It would not. Manchester spends two signalling events on every bit, so a gigabit of data would demand two gigabaud of signalling, and asking a link to change state that often surrenders on timing the very bandwidth the higher speed was bought to obtain.

The cost stated exactly: Manchester carries one bit for every two signalling events, an efficiency of one half. That is why 10 Mbit/s of data required 20 Mbaud of signalling, and by the same rule 1 Gbit/s would require 2 Gbaud. The overhead is not a fixed toll; it doubles whatever the bit rate becomes.

Why baud rate is what hurts: a medium limits how quickly a signal may change, not how many bits are attached to each change. Attenuation and distortion both worsen as the rate of change rises, so doubling the baud rate does not merely require a better link, it makes every impairment worse at the same moment.

The honest case in favour: the virtues are real: a transition in every bit period, no clock conductor, no direct-current component and a receiver simple enough for the electronics of the day. At ten megabits bandwidth was cheap and the receiver was expensive, so trading one for the other was correct.

What replaced it: codes that put several bits into each symbol, so the symbol rate stays a small fraction of the bit rate, with timing supplied by scrambling the data or by a block code that guarantees transitions while sacrificing a few per cent of capacity instead of half of it.

The principle underneath: the cost of buying timing with bandwidth rises in step with the bit rate, while the cost of buying it with a cleverer transceiver is paid once in silicon and then falls year by year. Two curves crossed, and the encoding chosen at each speed follows whichever is cheaper there.

The judgement: reject Manchester for a gigabit link, but reject it for the right reason. It is not obsolete because it is old, and it remains correct wherever rates are low and a trivial receiver matters, as on many sensor and industrial links. It fails at a gigabit because the exchange rate between bandwidth and cleverness turned against it.

So the junior engineer deserves the compliment of a real answer. Manchester was not superseded by something better in every way, but by something better at one particular price of bandwidth.

The Master Key: *Every encoding is a bargain struck between bandwidth and receiver complexity, and the bargain has to be struck again at each new speed.*

HOT Question 43

The accounts office at a district council in Kondoa copies a large file across its new link and watches the transfer rate settle well short of the figure on the invoice. The staff conclude they were sold less than they paid for.

Determine what could account for the gap between the advertised rate and the measured rate without any equipment being faulty, and reach a conclusion the office can act on.

The link is rated at 100 Mbit/s and both ends report a 100 Mbit/s connection.

One large file is copied from the server to a single workstation.

The measured transfer rate settles at 62 Mbit/s and holds there.

The run passes its cable test, and the error counters at both ports read zero.

Model Answer

Nothing is faulty. The 100 Mbit/s is a bandwidth, the rate at which the physical layer signals, while 62 Mbit/s is a throughput, the rate at which useful data arrives after every layer above has taken its share, and 0.62 of the signalling rate on one file copy is ordinary.

Bandwidth against throughput: bandwidth is a capacity, a fixed property of the link, and it is what a supplier can honestly advertise. Throughput is an outcome, measured, varying with the traffic, the endpoints and the hour. The first says what the link can do; the second says what it did.

The share taken above: the figure quoted counts bytes of the file, while the link also carries headers, a preamble and a gap between every frame. Several per cent of the advertised rate is spoken for before any other cause is considered.

A conversation, not a hose: the sender does not pour continuously. It sends an agreed amount, waits to be told it arrived, and sends more. Unless the quantity allowed in flight is at least the rate multiplied by the round-trip time, the sender idles part of every cycle on a healthy link.

The ends, not the link: a file copy involves a disk to be read, a disk to be written, a scanner inspecting what arrives and a file-sharing protocol taking its own turns. One transfer measures the slowest element in that chain, which is very often not the cable.

Everyone else: the capacity is the total for all users of that path. If two other offices are working at the same moment, the share reaching accounts is what is left. Capacity is shared; throughput is personal.

What would signal a real fault: error counters climbing rather than reading zero, a port settled at a tenth of its rated speed, a rate that collapses as load rises, or a rate that changes when a length of cable is swapped. None of those appears here.

The conclusion: conclude that the link is sound, and prove it before the supplier is blamed. Run several transfers at once and add the results; if the total climbs well above what one copy gave, the link was never the limit. Then measure between two machines on the same switch, which takes everything but the link out of the question.

The office was sold a capacity and measured a delivery, and the two were never the same quantity. What it should ask for is not more megabits but a written statement of the throughput one transfer should reach.

The Master Key: *Advertised rates are ceilings, so treat any measurement below one as normal until a counter, and not a feeling, says otherwise.*

Framing, error detection and the cyclic redundancy check

A physical medium delivers a run of voltage changes and nothing else: it cannot say where one message ends, and it cannot say whether what arrived is what was sent. The data link layer answers both questions with one structure. This section settles how a frame is marked out and counted, what every field of an Ethernet II frame is for and how large it is, and how the four bytes at the back are computed, checked and acted upon. It assumes you can already name the frame as the layer-2 unit.

BINDER Question 44

Eighteen machines in a secondary school laboratory in Kinondoni share one cable, and along it travels one unbroken run of voltage changes. Nothing announces itself. A receiver that cannot say where one message stops and the next starts has received orderly noise.

Explain what framing is, and account for the receiver's need to identify exactly where each frame begins and where it ends.

Model Answer

Framing is the data link layer's work of dividing a continuous stream of bits into discrete units and marking each one so that the receiver knows which bit is its first and which is its last; without those marks nothing can be located, computed over or handed upwards.

What a frame is: the data link protocol data unit: a header in front, a payload in the middle and a check value behind. It is the only form in which anything crosses a physical medium, so every bit on that medium belongs either to a frame or to the idle gap between two of them.

Bits carry no punctuation: the physical layer delivers ones and zeros with nothing between them. Begin reading the same run one bit late and every field lands in the wrong place: the address is rubbish, the check fails, and a perfect transmission is thrown away.

Marking the start: Ethernet opens with a preamble of 7 B of alternating ones and zeros, which lets the receiving circuit lock its clock to the sender's, and then 1 B, the start frame delimiter, whose ending breaks the alternation and declares that the frame proper begins with the very next bit.

Finding the end: the end is fixed by counting rather than by a second marker. The fields ahead of the payload are of known length, the standard fixes a smallest and a largest frame, and the last 4 B are the check sequence, after which the medium falls idle.

Why boundaries come first: every other service of the layer rests on this one. An address can be read only from a known position, a check value can be computed only over a known set of bytes, and nothing can be counted or discarded until it has edges.

A frame is not a convenience of the layer. It is the smallest thing about which a receiver can be certain.

The Master Key: *Find the start, count to the end, and only then read a single field.*

BINDER Question 45

A wall chart in the workshop of a technical college on Unguja draws the Ethernet frame as seven boxes of unequal width. Students copy it faithfully, then count the preamble inside the total and report a maximum of 1,526 B.

Using Figure 3.9, name each field of an Ethernet II frame in the order it is transmitted and state the size of each in bytes.

Model Answer

In order of transmission: preamble 7 B, start frame delimiter 1 B, destination address 6 B, source address 6 B, type 2 B, payload 46 to 1500 B and frame check sequence 4 B; the 64 B minimum and the 1,518 B maximum are both measured from the destination address to the check sequence, so the first two fields fall outside them.

Preamble and delimiter, 8 B: 7 B of alternating ones and zeros give the receiving circuit time to lock its clock to the sender's, and the single delimiter byte ends that alternation to announce the first bit of the frame. Neither is counted, because both are gone before the frame is held to begin.

The two addresses, 12 B: destination first at 6 B, then source at 6 B. Putting the destination first lets a station decide within the opening six bytes whether the frame concerns it, and stop reading if it does not.

The type field, 2 B: two bytes stating which protocol the payload belongs to, so that the receiver knows which process above should be handed it. It is the field that lets one medium carry several unrelated protocols at once.

The payload, 46 to 1500 B: the unit passed down from the layer above. Below 46 B the sender pads it; above 1500 B the frame is not legal, and something higher must divide the data before offering it.

The frame check sequence, 4 B: four bytes computed over everything from the destination address to the end of the payload. It is written last because it depends on all of that, and the receiver recomputes it to decide the frame's fate.

Adding the sizes up: 6 + 6 + 2 + 46 + 4 = 64 B at the floor and 6 + 6 + 2 + 1500 + 4 = 1518 B at the ceiling. Add the 8 B in front and you get 1,526 B on the wire, a true figure about the medium and the wrong answer about the frame.

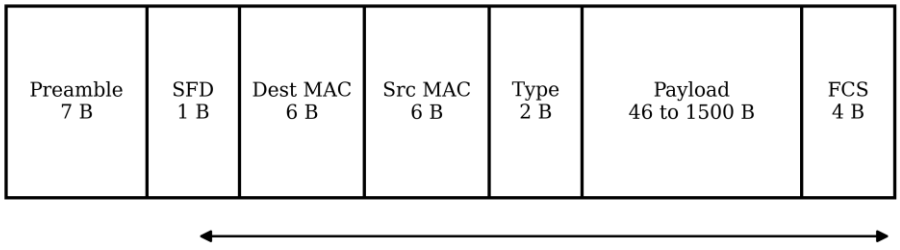


Figure 3.9: *The Ethernet II frame. The 64-byte minimum and the 1,518-byte maximum are both measured from the destination address to the frame check sequence.*

Seven fields, five of them counted, and the two left out arrive before the receiver is properly listening.

The Master Key: *Start counting at the destination address, because the preamble belongs to the wire rather than to the frame.*

BINDER Question 46

Twenty bytes of payload at a time, all day: a bank branch in Kyela sends short acknowledgements, and every one leaves the network card larger than it reached it. Nothing has been added to the message. Something has been added underneath it.

Explain why the Ethernet standard sets a minimum frame size of 64 bytes, and describe what a sending station does with a payload of fewer than 46 bytes.

Model Answer

The 64 B floor exists so that a station is still transmitting when news of a collision can reach it, since 6 + 6 + 2 + 46 + 4 = 64 B is 512 bit times, long enough for a signal to reach the far end of the longest permitted segment and return; a payload below 46 B is therefore padded with zero bytes up to 46 B.

Where 64 comes from: counting runs from the destination address to the check sequence, so the fixed fields take 6 + 6 + 2 + 4 = 18 B and the payload must supply the other 46 B. At eight bits to the byte, 64 × 8 = 512 bit times on the medium.

Collision detection needs time: a station that has already finished sending has nothing left to abandon and no reason to try again. It must still be putting bits onto the medium when the collision reaches it, or it will record as successful a frame that never arrived.

What a short frame would cost: a 20 B payload sent unpadded would make a frame of 18 + 20 = 38 B, and the sender would fall silent well before the round trip completed. Only the receiver would see the wreckage, and it would discard it without comment.

Padding, and who adds it: the sending station's data link layer appends zero bytes until the payload reaches the floor. For a 20 B payload that is 46 − 20 = 26 B of padding, and the check sequence is computed afterwards, so the padding is protected along with everything else.

Removing the padding: the padding carries no marker of its own. The receiver reads the true length from the header of the unit inside and ignores whatever lies beyond, so Ethernet never learns which bytes were real.

Runts: a frame arriving shorter than 64 B is a runt and is discarded unexamined. Runts are usually collision wreckage, so a rising count says the medium is contended rather than broken.

The floor is not about the message. It is about keeping the sender on the medium long enough to hear bad news.

The Master Key: *Pad up to the floor before you compute the check, because the check has to cover the padding too.*

BINDER Question 47

The referral desk at a district hospital in Ifakara sends each instruction twice, in writing and by radio, so a garbled word in one can be repaired from the other. The network under the same roof repairs nothing. It discards what is damaged.

Distinguish error detection from error correction, and state which of the two Ethernet performs.

Model Answer

Error detection adds only enough redundancy for a receiver to know that something in the unit is wrong, so its single remedy is to discard the unit, whereas error correction adds enough for the receiver to identify which bits changed and repair them without a resend; Ethernet detects and never corrects.

Detection, in outline: the sender computes a value over the data and sends it alongside. The receiver repeats the computation over what arrived and compares. Disagreement proves that at least one bit changed, and says nothing whatever about which one.

Correction, in outline: far more redundancy, arranged so that every fault the code claims to fix leaves a distinct signature. A Hamming code adding 4 check bits to 8 data bits repairs any single bit of the resulting 8 + 4 = 12, at a cost of 50 per cent more bits sent.

Three methods compared: the practical choices differ in how many bits they add and, more revealingly, in what they let through. Table 3.5 sets them side by side, and the pattern is that a longer check catches longer bursts of damage while none of the three repairs anything.

Method	Bits added	What it catches	What it misses
Parity bit	1	Any odd number of flipped bits	Any even number of flipped bits
Checksum	16	All single-bit errors, most short bursts	Errors that cancel within the sum
Cyclic redundancy check	32	Every burst of 32 bits or fewer	Roughly one longer burst in 4,294,967,296

Table 3.5: *Parity, checksum and cyclic redundancy check compared.*

Why Ethernet only detects: errors on a wired local link are rare and a resend across it is cheap, so paying correction overhead on every frame to save an occasional retransmission is a poor bargain. Ethernet spends 4 × 8 = 32 bits and stops there.

Where correction earns its cost: on links where asking again is impossible or ruinous: a probe beyond the planets, a scratched disc whose original is gone, a satellite hop with a long round trip. None of those describes a cable between two rooms.

Detection creates no obligation to repair. Ethernet finds the damage, discards the frame, tells nobody, and leaves the hole for a higher layer to notice.

The Master Key: *Ask what the receiver may do about the error, because that, and not the arithmetic, separates the two families.*

REAL Question 48

Four extra bits ride on the end of every catch total the fishing cooperative at Magu radios to its office in town, and nobody there reads them as data. They are the only reason the office trusts the weight it prints.

Calculate the cyclic redundancy check remainder for the data word and generator specified below, showing the modulo-2 division in full, and state the bit pattern that is actually transmitted.

Data word: 1101011011

Generator: 10011

The generator is 5 bits long, so the remainder is 4 bits.

Model Answer

Appending four zeros gives 11010110110000, and dividing that by 10011 modulo 2 leaves the remainder 1110, so the pattern actually transmitted is the original data word with that remainder behind it: 11010110111110.

Append four zeros: the generator 10011 is five bits long, so the check value is one bit shorter, four bits. Shift the data word left by four places, which means writing 11010110110000, and divide that. The zeros are reserving the space the remainder will occupy.

Modulo-2 arithmetic: there are no carries and no borrows, and subtraction is exclusive OR: 1 and 1 give 0, 0 and 0 give 0, 1 and 0 give 1. At each place, if the leading bit of the current five is 1 the generator is subtracted; if it is 0 nothing is subtracted and the next bit is brought down.

Step one: take the leading five bits, 11010. The leading bit is 1, so subtract: 11010 XOR 10011 = 01001. Drop the leading zero and bring down the next bit of the dividend, a 1, which gives 10011.

Step two: 10011 XOR 10011 = 00000. Now bring down the next four bits in turn, 1 then 0 then 1 then 1, and each time the leading bit of the five is 0, so nothing is subtracted. The working value becomes 01011, and one more bit, a 0, makes it 10110.

Step three: 10110 XOR 10011 = 00101. Bring down the next bit, a 0, giving 01010; its leading bit is 0, so bring down one more 0, which gives 10100.

Step four: 10100 XOR 10011 = 00111. Bring down the last zero of the dividend, giving 01110. The leading bit is 0, the dividend is exhausted, and the division stops there.

Reading off the remainder: the four bits remaining are 1110. The quotient, 1100001010, is thrown away: a cyclic redundancy check keeps only the remainder, because only the remainder is a value the receiver can reproduce from the bits it actually receives.

The pattern transmitted: put the remainder where the four appended zeros were. The data word 1101011011 followed by 1110 gives 11010110111110, fourteen bits, and that pattern divides by 10011 with nothing left over, which is exactly the property the receiver will test.

11010110110000 divided by 10011 leaves 1110

Each of those four bits was computed from all ten of the others, which is why altering any one of the ten alters at least one of the four.

The Master Key: *Append one zero fewer than the generator has bits, divide, and send the remainder in place of the zeros.*

REAL Question 49

A welding set two doors from the quay at Chake Chake inverts one bit of the fourteen a weighing terminal sends inland. The office computer knows within microseconds that the reading is spoiled. It can never say which bit spoiled it.

Show how the receiver detects a single inverted bit in the frame computed in the previous question, and explain why it cannot identify which bit was inverted.

Model Answer

The receiver divides the whole fourteen-bit word it received, check bits included, by the same generator 10011 and accepts only a remainder of 0000; the clean word 11010110111110 leaves 0000, while that word with its seventh bit inverted, 11010100111110, leaves 1011, so the frame is discarded, but 1,024 different error patterns leave that same remainder and nothing distinguishes them.

The test is one division: the receiver does not keep a copy of the check value and compare. It divides everything that arrived, the ten data bits and the four check bits together, by 10011, and asks one question: is the remainder zero.

Why a clean word divides: the sender chose the remainder precisely so that the transmitted word would be an exact multiple of the generator. Dividing 11010110111110 by 10011 therefore leaves 0000, and a remainder of 0000 is the only result that leads to acceptance.

The damaged word: inverting the seventh bit gives 11010100111110. Dividing produces, in turn, 11010 XOR 10011 = 01001, then 10011 XOR 10011 = 00000, then 11111 XOR 10011 = 01100, and finally 11000 XOR 10011 = 01011, so the remainder is 1011 and the frame is dropped.

A shortcut worth knowing: a received word is the sent word with an error pattern added modulo 2, and division distributes over that addition, so the remainder of the damaged word equals the remainder of the error pattern alone. Dividing 00000010000000 by 10011 also leaves 1011.

Why the bit cannot be named: the remainder is only four bits wide. A fourteen-bit word admits 16,384 possible error patterns while the remainder has only 16 possible values, so 16384 / 16 = 1,024 patterns share each one. Exactly one of the 1,024 flips a single bit, and nothing in the remainder says so.

What correction would need: a code that repairs damage must be built so that the nearest legal word to whatever arrived is always the word that was sent. Four check bits over fourteen fall far short of that, and Ethernet does not attempt it: the frame is dropped and no message goes back.

Detection returns a yes or a no, and the arithmetic that produces it was never asked for anything more.

The Master Key: *A non-zero remainder names the fact of damage and never its address.*

REAL Question 50

Three pages of handover notes wait on a desk at a district office in Lindi, left for whoever takes the network next. They are neatly written and admit no doubt anywhere. Three of the lines will send that person to the wrong place.

Identify the fault in each of the three statements below, give the correct version, and say why the correction is right.

1. The MAC sublayer controls the placement of packets onto the medium.

2. A frame that fails its check is retransmitted by the switch that noticed the failure.

3. A longer cyclic redundancy check will correct errors as well as detect them.

Model Answer

The unit the MAC sublayer places on the medium is a frame and never a packet; a switch that finds a bad check discards the frame in silence and asks nobody for a resend; and lengthening a check makes detection stronger without making correction possible at all.

Statement one, the wrong unit: at layer 2 the unit is a frame. A packet is the layer-3 unit, and it survives inside the frame as payload, unread by the sublayer carrying it. The MAC sublayer decides when a frame may go onto the medium and writes the hardware addresses onto it; it never handles a packet as a packet.

Why the name matters: the name of a unit records which header is outermost, and therefore which layer is in charge of it. Write packet where you mean frame and you have claimed that a device is reading a header it cannot see.

Statement two, nobody resends: a switch that finds a failed check discards the frame and advances a counter. It cannot ask for a resend: the frame is damaged, so even the source address in it may be wrong, and Ethernet has no acknowledgement to withhold. Nothing travels back at all.

Where recovery actually lives: the loss is noticed end to end, by a transport protocol that was expecting the missing bytes, and it is that protocol which asks again. Building the same machinery into every switch port would add timers, buffers and delay in order to repair something already repaired above.

Statement three, longer is not cleverer: a longer check detects longer bursts, and that is all it does. Correction requires a code in which every repairable fault leaves its own signature, whereas a remainder of any length is shared by an enormous number of different faults. A check of 64 bits would still be unable to name one wrong bit.

The habit that prevents all three: before writing what a layer does, name the unit it is holding and the single remedy it possesses. Layer 2 holds a frame and its only remedy is to discard it, and each of the three lines fails one half of that sentence.

None of the three statements is careless. Each is a true idea moved one layer, one device or one capability from where it belongs.

The Master Key: *Name the unit and name the remedy, and a claim that a layer does more than it can becomes impossible to write.*

HOT Question 51

A tea estate above Njombe runs its weighbridge, irrigation controllers and office on one cabled network. A supplier's engineer says the installation would be steadier if frames repaired themselves instead of being discarded. The proposal is not foolish. It is a bad bargain.

Assess whether Ethernet would be improved by correcting errors rather than merely detecting them, and commit to a judgement.

Model Answer

It would not be improved: correction has to be paid for on every frame while it would be called upon by almost none, and on a wired local network a retransmission is so cheap that detection and discard remain the right design, so keep the frame check sequence as it is and leave repair to the transport layer.

What correction would cost: enough redundancy that every fault the code claims to fix leaves a signature of its own. Four check bits added to eight data bits repair any single bit of the twelve and enlarge the transmission by 50 per cent, and covering the multi-bit bursts a real cable produces costs a great deal more again.

Paid always, used rarely: those bits travel in every frame, whether or not anything went wrong. On a switched copper or fibre link inside one building the overwhelming majority of frames arrive exactly as sent, so the premium buys insurance against an event that hardly ever happens.

What detection costs instead: 32 bits on a frame that may carry 1500 B of payload, and those 32 bits catch every burst up to 32 bits long. The comparison is not close: a small fixed charge that detects almost everything, against a large fixed charge that repairs a little.

Resending is cheap here: the two ends are metres apart, and the transport layer already keeps the sequence numbers, the timers and the copies needed to send again. Correction below would duplicate machinery that must exist above in any case, since no link-layer code can repair a frame that was never delivered.

Where the answer flips: correction earns its keep wherever a resend is impossible or ruinous: a spacecraft too far away to ask twice, an optical disc whose original no longer exists, a satellite hop whose round trip is long enough to stall a conversation. A cable between two rooms is none of these.

The judgement: keep detection. Advise the estate to spend nothing on correction and, if one run is producing failed checks, to repair that run, because correction on a bad cable hides the fault, keeps the link busy resending nothing and postpones the day somebody finds the cause.

The engineer's instinct, that damage should be repaired where it is found, is a good one everywhere except on a link where asking again costs almost no time at all.

The Master Key: *Buy correction only where you cannot buy a second attempt.*

HOT Question 52

A few dozen failed checks a month in January, 2,400 in the last hour, on one switch port at a district office in Nzega. Nothing is down. The manager must decide before lunch whether this is an emergency or a nuisance.

Determine what a network manager should conclude from a rising count of frames failing their check on a link that has not lost connectivity, and state what to do first.

Model Answer

The conclusion is that one physical link is degrading and that the network is not failing: the check is doing exactly what it exists to do, every damaged frame is being discarded and a transport protocol is quietly resending, so the cost shows as slowness rather than as an outage; the first action is to read the error counters port by port, find the single receiving port that owns the count, and replace the patch lead at that end.

What the counter records: a frame that arrived complete and whose recomputed check disagreed with the four bytes it carried. Bits changed somewhere between the two ends of one cable. Nothing above layer 2 was ever told, which is precisely why nothing looks broken.

Why connectivity survives: the discard leaves a gap that the transport layer notices and fills. Small exchanges succeed on the second attempt, and users see at worst a hesitation, so the fault reaches the manager as a counter rather than as a complaint.

Reading the rate honestly: 2,400 failures against roughly 3,000,000 frames in the hour is 0.08 per cent, which sounds like nothing and is not. Each failure costs a whole frame and a round trip, and the proportion is climbing, which matters far more than today's value.

The counter names the port: a failed check is recorded by the port that received the frame, so the port carrying the count sits at one end of the faulty run. Read the far end as well: damage in both directions points at the cable or its surroundings, damage in one at the transceiver feeding it.

Causes in order of cheapness: a crushed or badly terminated patch lead first, then a run longer than the standard permits, then a length of cable lying beside a motor, a welding bay or a fluorescent fitting, then a failing transceiver, and last a duplex mismatch, which brings a family of counted faults of its own.

What not to do: do not declare an outage, and do not begin altering configuration. No setting on a switch can make a check fail, because the arithmetic is done in hardware over the bits that actually arrived, so a rising count is always a statement about the physical world.

The judgement: treat it as maintenance with a deadline rather than an emergency. Swap the patch lead, watch the counter for an hour, and if it does not fall, move the run away from the interference and then replace the transceiver.

A link that fails quietly is more dangerous than one that fails loudly, because nobody reports it until the day it stops working altogether.

The Master Key: *When frames are dying quietly, follow the counter to the port and the port to the cable.*

MAC addressing, ARP and the switched LAN

Every frame that crosses a local network carries two kinds of address: one that names a machine anywhere in the world, and one that names the interface at the other end of this particular wire. This section settles the second of the two. It takes the hardware address apart, sets it beside the logical address so that neither can be mistaken for the other, follows one ARP exchange from the broadcast question to the unicast answer, and shows how a switch builds its address table out of frames it was never asked to study.

BINDER Question 53

A crate of replacement network cards reaches a technical college in Kyela, each card carrying a white sticker on its underside: twelve characters, printed in six pairs. The storekeeper wants to know whether two cards could ever carry the same twelve characters.

Describe the structure of a MAC address, stating how many bits the whole address occupies and how those bits are divided between its two parts.

Model Answer

A MAC address is 48 bits long, written as twelve hexadecimal digits in six pairs, and it divides into two halves of 24 bits: an organisationally unique identifier naming the manufacturer, and a device identifier naming that one interface.

Forty-eight bits, six bytes, twelve digits: one hexadecimal digit stands for four bits, so six bytes are written as twelve digits, usually separated in pairs by colons, as in 00:1b:44:11:3a:b7. The separators carry no meaning; the twelve digits carry all of it.

The first half names the maker: the leading 24 bits are the organisationally unique identifier, allocated to one manufacturer by the body that issues them. Twenty-four bits allow 16,777,216 different prefixes, so an address whose first six digits belong to nobody was issued by nobody.

The second half names the card: the remaining 24 bits are chosen by that manufacturer and need only be unique among the cards it ships under its own prefix. Uniqueness is therefore administered in two stages, and no central register hears about an individual card.

Burned in at manufacture: the address is written into memory on the interface when the card is made, which is why it is called the physical or hardware address. It belongs to the interface and not to the computer, so a machine with a wired card and a wireless card has two.

The one reserved value: an address of ff:ff:ff:ff:ff:ff, every one of the 48 bits set to one, is the broadcast address. Every interface on the link accepts a frame carrying it, and no manufacturer may issue it to a card.

$$\text{Total possible addresses} = 2^{48} = 281{,}474{,}976{,}710{,}656$$

Two raised to the power 48 is 281,474,976,710,656, which is why a scheme with no central register of individual cards can still promise the storekeeper that no two will collide.

The Master Key: *Split the twelve digits down the middle: the left half names the maker, the right half names the card.*

BINDER Question 54

One laptop travels with its teacher between the staff room and the computer laboratory in Njombe, two rooms on different networks. One of the machine's two addresses changes in the second room. The other has not altered since the card was made.

Distinguish a MAC address from an IP address on at least four grounds, and state which of the two changes when a machine is moved to a different network.

Model Answer

They differ in length, in structure, in the layer that reads them and in how far they reach: a MAC address is a flat 48-bit hardware identifier meaningful on one link only, while an IP address is a hierarchical logical address, 32 bits in version 4, that identifies a host end to end; the IP address is the one that changes.

Flat against hierarchical: nothing in a hardware address says where the interface is. Its two halves name a manufacturer and a card, and one manufacturer's cards are scattered across every country, so addresses cannot be grouped by place. A logical address splits into a network part and a host part, so one routing entry can stand for every host on a network.

Physical against logical: the hardware address is burned into the card and describes the thing itself. The logical address is issued by an administrator or by DHCP and describes where the thing currently sits.

One link against end to end: a router strips each arriving frame and writes fresh hardware addresses for the next hop, so they change at every hop of a journey. The logical addresses in the packet survive the whole path unaltered, which is why a frame reaching a distant server never carries the original sender's hardware address.

Why both are needed: the logical address answers which machine, anywhere in the world; the hardware address answers which interface, here on this wire. Neither can do the other's work, and the gap between them is what ARP exists to close.

Property	MAC address	IP address
Length and notation	48 bits, twelve hexadecimal digits	32 bits in version 4, four decimal octets
Structure	Flat: maker prefix plus card number	Hierarchical: network part plus host part
Layer that reads it	Layer 2, the data link layer	Layer 3, the network layer
How it is set	Burned into the card at manufacture	Configured by hand or issued by DHCP
How far it reaches	One link; rewritten at every hop	End to end, unchanged along the path
On moving to another network	Unchanged, it travels with the card	Changes, because it records a position

Table 3.6: *MAC address and IP address compared.*

The laptop is issued a new logical address in the laboratory and keeps the hardware address it was born with. An answer that says only physical against logical has given one ground out of four.

The Master Key: *One address says who the interface is, the other where it presently sits, and only the second changes when the machine moves.*

BINDER Question 55

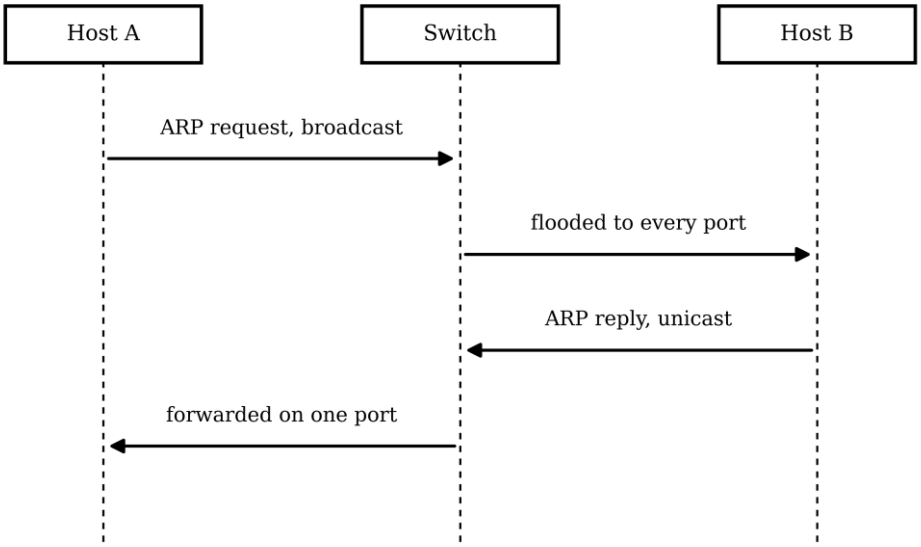
In a laboratory at Sengerema a machine has the IP address of the school server and nothing else. It cannot send a frame until it learns the six bytes naming the interface holding that address, and it can only ask the whole wire.

Using Figure 3.10, describe the ARP request and the ARP reply, saying which of the two is broadcast and which is not, and state what the sender does with the answer.

Model Answer

The sender knows a destination IP address and needs the hardware address that goes with it, so it broadcasts an ARP request to ff:ff:ff:ff:ff:ff; only the machine holding that address answers, with a unicast reply addressed to the asker alone, which the asker stores in its ARP cache.

What the request carries: the sender's own logical and hardware addresses, and the target logical address, with the target hardware field left empty. That blank is the question; the reply is the same message with it filled in.



The request is broadcast to every port and carries the sender's own address; the reply is a unicast frame that goes back only to the host that asked.

Figure 3.10: *An ARP exchange. The request is broadcast to every port; the reply is a unicast frame that goes back only to the host that asked.*

Why the request must be broadcast: the sender cannot address the frame to the machine it is hunting for, because the address it would need is the thing it is asking about. Every other interface reads the request, finds the target address is not its own, and discards it.

Why the reply need not be: the machine that answers already holds the asker's hardware address, because the request carried it, so it replies with an ordinary unicast frame that the switch forwards on one port only.

The cache and its lifetime: the answer goes into the ARP cache, so the next few hundred frames to that host need no broadcast at all. The entry expires after a few minutes, so that a replaced card is not believed for ever.

Where ARP sits in the stack: an ARP message travels directly inside an Ethernet frame with a type value of its own, not inside IP, which places it at the boundary of layers 2 and 3. No router forwards one, so ARP resolves addresses only on the link it was sent on.

Read the figure downwards and the asymmetry is the whole lesson: one question put to everybody, one answer returned to one.

The Master Key: *Broadcast the question, because you cannot address a frame to the address you are still looking for.*

BINDER Question 56

A four-port switch under the counter of a stationery shop in Kinondoni has run since the day it was plugged in, and nobody has ever configured it. It nevertheless knows which machine sits on which port. Nothing was ever told to it.

Using Figure 3.11, explain how a switch builds its MAC address table, and state what it does with a frame whose destination address is not yet in that table.

Model Answer

The switch reads the source address of every arriving frame and records it against the port the frame arrived on, because that is the one thing an arriving frame proves; a frame whose destination is not in the table is flooded out of every port except the one it came in on.

Learning is from the source: a frame arriving on Fa0/1 proves that whatever sent it can be reached through Fa0/1. It proves nothing whatever about where its destination lies. So the switch writes source address against

arrival port, and every one of the four entries under the figure was built that way, from traffic sent for other purposes.

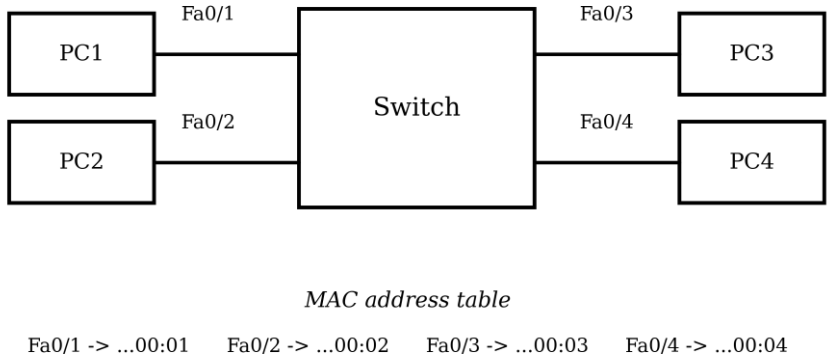


Figure 3.11: *A switch builds its address table from the source address of every frame that arrives, never from the destination.*

Forwarding and filtering a known address: if the destination is in the table against some other port, the frame goes out of that port and no other. If it is in the table against the arrival port, sender and receiver are already on the same side, and the switch discards the frame.

Flooding an unknown address: with no entry to consult, the switch sends the frame out of every port but the arrival port. This is wasteful and never wrong: the frame reaches the destination if it exists, and the answer supplies the entry that was missing. A broadcast frame is flooded always.

Entries do not last for ever: an entry that is not refreshed within its ageing timer, commonly five minutes, is deleted, so the table describes the network as it is rather than as it has ever been.

A switch with an empty table is not a faulty switch; it is a switch that has just been turned on, and the first frame to every silent machine is always flooded.

The Master Key: *A switch believes only what a frame demonstrates, and a frame demonstrates only where it came from.*

REAL Question 57

Two machines in a training laboratory in Mikocheni came from one disk image that carried a hardware address set in software. Either works while the other is idle; neither works when both are on. Replacing a patch lead and a card changed nothing.

Explain what a duplicate MAC address does to a switch's address table, and describe how the fault presents itself to the two users.

Both machines report the hardware address 00:1b:44:11:3a:b7.

Switch log, repeated many times an hour: host 00:1b:44:11:3a:b7 moving from port Fa0/6 to port Fa0/9.

Each machine has its own IP address, and neither IP address is duplicated.

Model Answer

One address may occupy only one row of the table, so each frame from either machine overwrites the row the other wrote; the address flaps between the two ports, and every frame sent to it goes to whichever machine transmitted most recently, so each user is served only while the other stays quiet.

Why the table cannot hold both: the table maps an address to a port, and an address appears once. Learning is unconditional: the switch does not weigh the new port against the old one, it simply writes what the latest frame proves. Two machines claiming one address produce one row rewritten several times a second.

What the users actually see: replies arrive at the wrong machine. A session holds for as long as its own machine is the more talkative and then stalls; a file transfer stops halfway; a ping answers ten times and then loses everything until the other machine falls silent. Nothing is consistently broken, and that is the signature.

Why the usual tests all pass: link lights are on, the cable tests good, both machines reach the switch, and each machine on its own is faultless. Testing one machine with the other switched off is precisely the test that hides this fault, which is how a technician comes to replace two innocent components.

How cloning can cause it at all: a burned-in address cannot clash, because the maker's own 24 bits are unique within its prefix. But an operating system can be told to present an address of its own choosing, and that setting travels inside a disk image. The card is innocent.

Reading the evidence: the log line names the address and both ports it is moving between, which is the diagnosis complete in one sentence. Without a log, display the address table twice a few seconds apart and watch the port change while nothing has been unplugged.

The repair: take the address override out of the image so that each interface presents its own burned-in value, or, where an override is wanted, give the second machine a different one. Then clear the table and let the switch learn the network again.

A duplicate hardware address is not a broken machine. It is two machines telling the truth about one identity, and the switch believing whichever spoke last.

The Master Key: *When a fault changes sides according to who spoke last, look for one identity claimed by two devices.*

REAL Question 58

A student in Chake Chake has written ARP out twice, on two cards made a fortnight apart, and neither card says which layer the protocol belongs to. Three statements taken from the two, all in the same confident hand, are wrong.

Identify what is wrong with each statement, give the fact that should have been written, and say in each case why it is so.

1. ARP resolves a computer's name to its address, which is how a machine finds a server it has only been given the name of.

2. A switch learns which port a machine is on by reading the destination address of each frame that arrives.

3. A laptop's MAC address changes when it joins a different network, which is how the new network knows where to deliver its traffic.

Model Answer

All three are wrong: ARP resolves a known IP address to a hardware address and never handles names at all, a switch learns from the source address because that is the only thing an arriving frame proves, and a MAC address is burned into the card and does not change when the machine is moved, the address that changes being the IP address.

ARP never sees a name: turning a name into an address is the work of DNS, up at layer 7. An ARP message has nowhere to put a name: it carries four address fields, the sender's logical and physical and the target's logical and physical, and the last of those is the blank the reply fills in. A machine given only a name resolves the name first, then runs ARP on the address it gets back.

A switch learns from the source: the destination address states where a frame is going, and the switch has no evidence at all that the destination lies in the direction the frame came from. The source address is different in kind: the frame demonstrably arrived on that port, so the sender is reachable through it. Learning from destinations would file every host against the port of whoever was looking for it.

A hardware address does not move house: the address is written into the card at manufacture and identifies that interface wherever it is taken. Joining another network changes the logical address, by configuration or by DHCP, precisely because a logical address records a position. An administrator can override a hardware address in software, but plugging in a cable does not do it.

What the three have in common: each hands one address the job of the other, or credits a device with evidence it does not hold. Two questions defeat the whole family: what does this address identify, and how far does it reach. Reconcile duplicated notes as well, because a protocol written out twice without its layer named is a protocol not yet understood.

Corrections of this kind cost one line each and are worth several marks, because a reader meeting the wrong claim cannot tell a slip of the pen from a settled misunderstanding.

The Master Key: *Ask of any address what it identifies and how far it reaches, and this whole family of mistakes becomes impossible.*

HOT Question 59

A Form Five class in Kahama is stuck on a fair question: a hierarchical address lets a router hold one entry for a whole distant network, so why must a switch hold one entry for every machine in a cupboard?

Judge whether MAC addresses could have been made hierarchical in the way IP addresses are, and defend your judgement.

Model Answer

They could have been, and hierarchy would indeed have bought aggregation; but they should not have been, because a hierarchical address must be handed out by whoever owns the hierarchy and reissued whenever the device moves, which is the cost the logical address already pays: layer 2 was given a permanent identity so that a card works anywhere with nothing configured.

The address is not shapeless: a hardware address is already structured; it is simply structured on the wrong axis. The first 24 bits group addresses by manufacturer, and one manufacturer's cards end up in every country there is. Hierarchy helps forwarding only when it follows the topology.

What hierarchy would have bought: aggregation. A switch could keep one entry for a whole cupboard, as a router keeps one route for a whole network, and its table would stop growing with the number of machines. That is a genuine gain, and it is the reason layer 2 does not scale beyond a building.

What it would have cost: assignment. Somebody must decide which branch of the hierarchy a card belongs to, and a card carried to another branch must be renumbered. That means configuring every device by hand, or a protocol to issue addresses, which must itself run before the machine has any address to speak with.

The property that would be lost: a stable identity. Because a burned-in address never changes, it can be used to recognise a particular machine: to reserve an address for it, to admit it to a wireless network, to tie a licence to it. An address reissued on every move could do none of that.

Two layers, two questions: the design keeps identity and location apart on purpose. The hardware address answers who this interface is; the logical address answers where it sits today. Making both hierarchical would answer the second question twice and leave the first unanswered.

The judgement: it was possible and it was rightly refused. Hierarchy earns its cost only where tables would otherwise be unmanageable, which is true of the whole world and false of one link, where a flat address costs a few thousand rows and saves every machine from being configured before it can speak. Write the answer as a trade and name both halves.

The class was not wrong about the benefit. They were weighing one side of a trade, and the marks are on the other side.

The Master Key: *Flat addressing buys identity that needs no configuration, and pays with a table that grows one row per machine.*

HOT Question 60

Anybody who can reach the wire can answer a question never put to them, and be believed. At a secondary school on Unguja the accounts machine shares a switch with thirty student machines, and one student has worked out what that means.

Assess the security consequences of ARP carrying no authentication, and recommend what a school of this size can realistically do about it.

Model Answer

Because a host accepts any reply that names the address it asked about, and most accept unsolicited replies too, anyone on the link can claim the gateway's address and receive every machine's traffic instead; the school cannot repair the protocol, so the realistic answer is encrypted services, the accounts machine off the laboratory link, and a fixed gateway entry where it matters.

Why the protocol cannot tell: an ARP reply carries no signature and no evidence of any kind. There is nowhere in the message to put either, and a new field would mean nothing unless every host on the link were changed on the same day. A host has no way to prefer a true answer to a quick one.

What the attack achieves: claiming the gateway's address places the attacker between every machine and the outside world, where traffic can be read, altered, or simply dropped. Claiming a server's address does the same for one service. Nothing has to be broken into, because the other machines cooperate.

The one comfort, and its limit: ARP does not cross a router, so an attacker must already be on the same link. That excludes the whole world and includes everybody in the room. In a laboratory where students sit at the machines and visitors plug into spare sockets, that is not much of a barrier.

What is not worth the money: buying managed switches for this reason alone. Dynamic ARP inspection and a limit on addresses learned per port both work, and are worth enabling on switches the school already owns, but new switches are not the cheapest protection and do nothing for a wireless client.

What is worth doing, in order of value: require that the accounting and results systems are reached over encrypted connections, so intercepted traffic yields nothing readable. Move the office machines onto a link the laboratory does not touch, since ARP reaches no further than one link. Enter a fixed ARP entry for the gateway on those machines. Disable unused ports.

The recommendation: spend nothing on defending ARP and everything on making it not matter. Encryption first, separation second, fixed entries on a handful of machines third, switch features only where the equipment already has them. Say plainly that ARP cannot be authenticated, then rank the remedies by cost.

The student on Unguja has found a weakness in a protocol written when everyone on a wire was trusted. The verdict is that it cannot be patched out, only planned around.

The Master Key: *You cannot authenticate ARP, so make the traffic it misdirects worth nothing to whoever collects it.*

HOT Question 61

A switch in the booking office of a ferry operator at Bagamoyo has run eight months without a restart. Its address table holds fewer rows this morning than at noon yesterday, and nothing has been removed. The switch is forgetting on purpose.

Determine why a switch must age entries out of its address table, and state what would go wrong if it never did.

Model Answer

Ageing is required because the table is a fixed-size store of guesses about a network that keeps changing, and the failure it prevents is not a frame sent to the wrong port but a table with no room left: without it, the store fills with machines long gone and the switch can no longer learn about those present.

Movement is not the main reason: a machine unplugged from one port and moved to another repairs its own entry the moment it sends anything, because learning is from the source and the newest frame wins. Ageing earns its keep on the machine that moves and stays silent, and on the one that never comes back.

The store is small because it is fast: entries live in special memory that can be searched in a single operation at the speed of the wire, and memory of that kind is expensive, so a small switch holds a few thousand rows. Nothing in the design lets it grow when it is full.

What a full table does: once no row can be added, the switch stops learning. Every address it has not already heard is flooded to every port from then on, because the entry that would have stopped the flooding can never be written. The office keeps working and gets steadily slower.

Forgetting is safe, remembering wrongly is not: flooding is the fallback for an address the switch does not know, and it always reaches the destination, merely wasting capacity. The fallback for an address it knows wrongly is to send the frame confidently to a port where nothing is listening, and that frame is lost.

Choosing the timer: five minutes, or 300 seconds, is the usual default. Too short and the switch floods constantly, because a host that pauses between jobs is forgotten. Too long and the table fills with the departed. Five minutes releases a machine that has gone within one lesson.

The judgement: the decisive argument is capacity rather than accuracy. Wrong entries correct themselves as soon as the machine speaks, but a table that never forgets cannot be enlarged, and the day it fills is the day the switch behaves as though it had learned nothing. State capacity first and the safety asymmetry second.

Eight months of uptime is exactly the condition that would expose a switch with no ageing timer, which is why the office at Bagamoyo has never noticed one exists.

The Master Key: *A switch must be allowed to forget, because a table that cannot forget will in the end be unable to learn.*

Media access control and the two domains

A medium that several machines share is a resource rationed in time, and the rules for rationing it are among the oldest ideas in networking. This section settles how wired Ethernet resolves a collision after it has happened, why a wireless station must instead work to prevent one, and what a hub, a bridge, a switch and a router each do to the two domains every network is divided into. It assumes you can already say what a frame is and what a switch is for.

BINDER Question 62

A squeal is all the controller hears when two drivers of a bus company in Nzega key their handsets in the same instant. Nobody blames the radios. The channel carries one voice, and the drivers had no rule of turns.

Explain why a medium shared by several stations needs a media access control method, and name the method used by wired Ethernet.

Model Answer

A shared medium carries one signal at a time, so two stations transmitting together destroy both frames; a media access control method is the agreed rule deciding which station may transmit and when, and wired Ethernet uses carrier sense multiple access with collision detection.

One signal at a time: copper and radio cannot keep two transmissions apart. Two overlapping signals add into a single meaningless waveform, so a medium is never divided among its users; at any instant it is occupied by one.

What a collision costs: neither frame survives its check sequence, so every receiver discards both. The medium has been busy for two frame times and delivered nothing, and both senders must begin again.

Why listening is not enough: a signal takes time to travel. Across 100 m of copper it arrives about 0.5 µs late, and inside that window a second station can test the medium, find it genuinely quiet and begin.

What the rule must settle: when a station may begin, how it learns that its attempt failed, how long it waits before retrying, and how two stations that have just collided avoid colliding again at once.

Three families of answer: contention lets stations compete and clears up afterwards; controlled access passes a token or polls each station in turn; channelisation splits the medium by time or by frequency. Ethernet chose contention, which needs no controller and costs nothing while the wire is quiet.

The name to give it: carrier sense multiple access with collision detection, CSMA/CD: listen first, share the medium, and notice the failure while it is still happening.

The drivers in Nzega need the same three habits: listen before speaking, know when two voices went out together, and wait different times before trying again.

The Master Key: *Every access method answers two questions, who may begin and what happens when two begin together, so learn a new one by finding its answer to each.*

BINDER Question 63

On an old hub in a laboratory at Mbinga two machines collide, wait, and collide again in the same instant. On the third attempt they do not. No cable was touched, and neither machine was told anything whatever about the other.

Using Figure 3.12, describe the CSMA/CD procedure from the moment a frame is ready to send until it has been sent successfully.

Model Answer

The station waits for an idle medium, transmits while still listening, and on detecting a collision sends a jam signal, abandons the frame, waits a random number of slot times set by binary exponential back-off, then senses the medium again before retransmitting.

Carrier sense: the station tests the medium first. While another transmission is present it neither queues nor interrupts but keeps testing, the small loop at the left of the flowchart, and starts only when the medium is idle.

Transmit, listen, jam: detection happens during transmission, not after it. The interface compares what is on the medium with what it is putting there, and a difference means somebody else is sending too. The station then stops the frame and puts out a jam sequence of 32 bits, so every station registers a collision rather than a short frame.

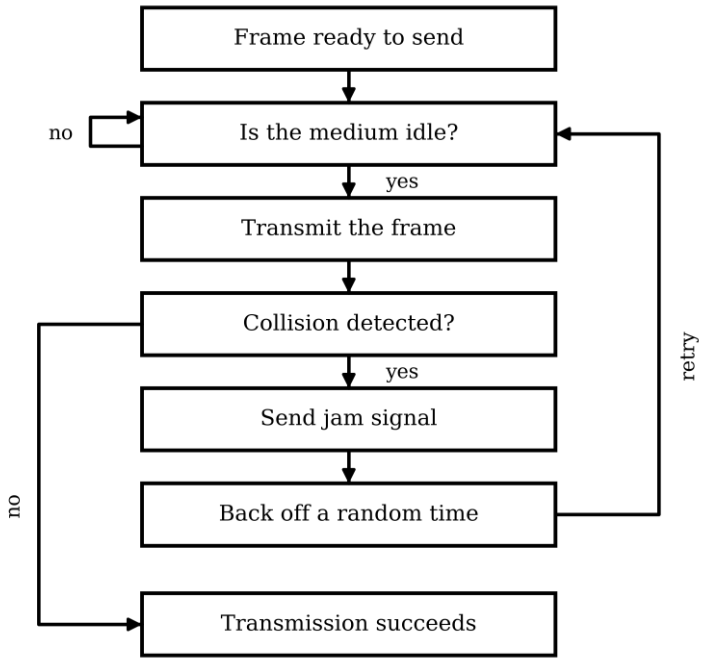


Figure 3.12: *CSMA/CD. Both loops return to the medium test, because a station that has backed off must sense the medium again before it retransmits.*

Binary exponential back-off: after a collision the station waits a whole number of slot times drawn at random from the range below. The first collision offers two waits and the third eight, from 0 to 7, so doubling the range makes an immediate repeat less likely.

$$\text{Wait} = \text{a random whole number of slot times from 0 to } 2^{n} - 1$$

The figures that bound it: a slot time is 512 bit times, 51.2 µs on a 10 Mbit/s segment, long enough for a signal to reach the far end and any collision to return. The exponent stops growing at 10, giving 1,024 slots, and the frame is discarded after 16 attempts.

Back to the top: the second loop matters as much as the first. A station whose wait has expired does not transmit at once; another may have claimed the medium meanwhile, so it senses again.

The procedure never promises that no collision will happen, only that one is noticed within a slot time, announced to everybody, and settled by two stations drawing different waits.

The Master Key: *Detection is only half of CSMA/CD, since it is the randomness in the back-off that stops the same two stations colliding for ever.*

BINDER Question 64

Two logs lie side by side at a secondary school in Chake Chake. The switch has counted collisions all week; the access point above the same corridor has counted none since it was installed, and yet frames are plainly being sent twice.

State why a wireless station cannot detect a collision, and outline what CSMA/CA does instead.

Model Answer

A wireless station transmits and receives on one channel, and its own output so far exceeds any distant signal at its antenna that it hears nothing else while sending, so 802.11 does not try to detect collisions; instead every station waits an inter-frame space and then a random back-off before transmitting, and confirms success with an acknowledgement.

Deafened by its own signal: transmitter and receiver share one antenna and one channel, and what the station puts out swamps whatever arrives, so the comparison a wired interface makes between what it sent and what it hears is impossible. In any case the collision happens at the receiver, which may be far from both senders.

Avoidance instead of detection: the station senses the channel, and once it falls idle waits a fixed inter-frame space and then counts down a random back-off in slot times. The counter freezes whenever the channel becomes busy, so a station that has already waited keeps its place.

Virtual carrier sense: every frame carries a duration value. A station overhearing one starts a timer, the network allocation vector, and treats the channel as busy for that long even if it hears nothing more, so deferral is decided from a header, not from energy alone.

Acknowledge or assume the worst: each unicast frame must be acknowledged after a short inter-frame space. Silence is the only evidence of failure available, so a sender treats a missing acknowledgement as a collision, doubles its contention window and tries again.

The optional handshake: before a long frame the sender may reserve the channel with a request-to-send and clear-to-send exchange.

Feature	CSMA/CD	CSMA/CA
Where it is used	Wired Ethernet on a shared half-duplex segment	Wireless 802.11
Collisions	Detected while transmitting	Cannot be detected, so avoided in advance
Before sending	Sense, then transmit as soon as the medium is idle	Sense, wait an inter-frame space, then count down a random back-off
On failure	Jam, then binary exponential back-off	No acknowledgement arrives; the contention window doubles
Acknowledgement	None at this layer	Every unicast frame is acknowledged

Table 3.7: *CSMA/CD and CSMA/CA compared.*

Detection reacts once the damage is done; avoidance spends a little airtime on waiting so that the damage happens less often. Wireless has no third option.

The Master Key: *Wired Ethernet learns that it failed and wireless learns only that it did not succeed, which explains every extra frame 802.11 sends.*

BINDER Question 65

Ask two questions of one room: which machines can spoil one another's transmissions, and which must break off to read a frame addressed to everybody? At a district office in Kinondoni the answers were nine and forty-one. One word will not serve for both.

Define a collision domain and a broadcast domain, and state precisely how the two differ.

Model Answer

A collision domain is the set of interfaces whose transmissions can overlap and destroy one another because they share one medium; a broadcast domain is the set of interfaces a frame addressed to every station will reach. The first concerns simultaneous sending, the second who must listen.

The collision domain: everything on one shared half-duplex segment: the stations, the cable and any hub joining them. Any device that takes in a whole frame before forwarding it, whether bridge, switch or router, ends the domain at its port.

The broadcast domain: every interface a frame addressed to all stations will reach. A hub repeats it, a bridge and a switch flood it out of every other port, and only a router refuses to pass it. It is the extent of one layer-2 network.

The two questions differ: one asks who can interfere with your transmission, a matter of contention for the medium. The other asks whose processor is interrupted by your broadcast, a matter of how far an address reaches. A silent segment still has a collision domain.

How they nest: many collision domains sit inside one broadcast domain, never the reverse. Adding switches multiplies the first and leaves the second untouched, which is why a fully switched building stays one broadcast domain until a router, or a logical division of the switch, is introduced.

Full duplex ends collisions: a switch port running full duplex gives each direction its own path, so the two interfaces on that link may transmit at once without interfering. Nothing is left to contend for, collision detection is switched off, and no collision is possible.

Count them separately. Collision domains tell you where machines slow one another down; broadcast domains tell you how far a storm travels before something stops it.

The Master Key: *Collisions are stopped by anything that stores a frame before sending it on, but broadcasts are stopped only by something that reads an address above layer two.*

REAL Question 66

The supplier's invoice for a small office in Ilala lists a router, two switches and a hub, and the manager wants to know what he has bought. The honest answer is not a count of boxes but a count of domains.

Determine how many collision domains and how many broadcast domains this office contains, and justify each figure.

1. The router has four interfaces, all in use: one to the service provider, one to switch A, one to switch B and one to the hub.

2. Switch A has eight ports: port 1 to the router, and ports 2 to 8 to seven computers.

3. Switch B has eight ports: port 1 to the router, and ports 2 to 8 to seven computers.

4. The hub has four ports: port 1 to the router, and ports 2 to 4 to three older machines.

5. Every switch port runs full duplex; the hub and its three machines run half duplex.

Model Answer

The office contains 18 collision domains and 4 broadcast domains: 8 from switch A, 8 from switch B, 1 for the whole hub segment and 1 for the link to the provider, while the router's four interfaces mark out four separate networks.

Read the rule off the table: a hub is 1 collision domain and 1 broadcast domain whatever its size, because it repeats every bit everywhere. A switch with 8 ports is 8 collision domains and 1 broadcast domain. A router with 4 interfaces is 4 collision domains and 4 broadcast domains, since it stops both.

The two switches: each port faces a segment of its own, and a frame leaving one port cannot collide with a frame leaving another, so each switch contributes eight. The link up to the router is one of those eight and must not be counted a second time at the router's end.

The hub contributes one: its four ports are electrically one wire. The link from the router and the three older machines all sit in the same collision domain, so the hub does not add four domains; it merges what would otherwise have been four into one.

The link to the provider: the router's fourth interface faces the provider's equipment, and that link is a collision domain in its own right. Adding the parts gives 8 plus 8 plus 1 plus 1, which is 18.

Why only four broadcast domains: a broadcast crosses a hub and is flooded by a switch, so it fills everything hanging off one router interface and stops there. Four interfaces give four broadcast domains, however many switches are cascaded beneath them.

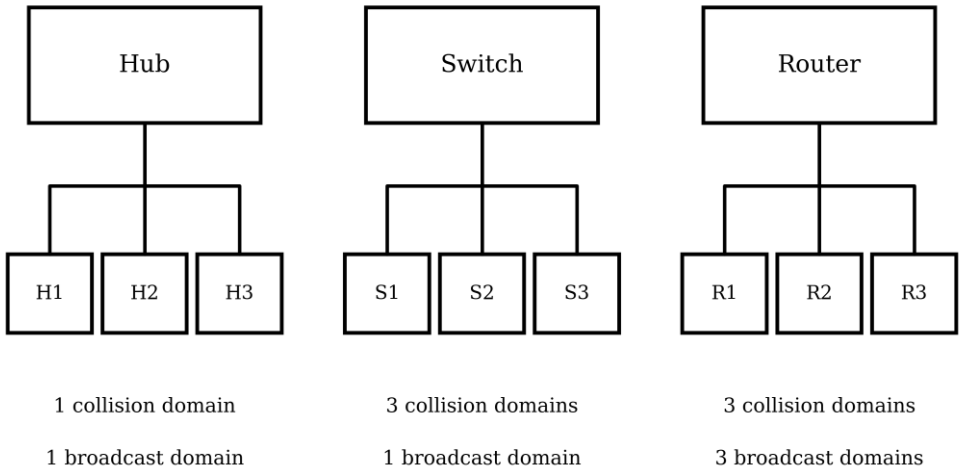


Figure 3.13: *What each device does to the two domains. A switch splits collision domains only; a router splits both.*

Device	Layer	Collision domains	Broadcast domains	Forwarding decision based on
Hub	1, physical	1 for the whole device	1	None; every bit is repeated to all other ports
Bridge	2, data link	1 per port	1	Destination MAC address, decided in software
Switch	2, data link	1 per port	1	Destination MAC address, decided in hardware
Router	3, network	1 per interface	1 per interface	Destination IP network, from the routing table

Table 3.8: *Hub, bridge, switch and router compared.*

What the count is for: the three machines on the hub share one medium and contend with one another all day, while the fourteen computers on the two switches never do. If those three matter, the four-port hub is the cheapest thing in the office to replace.

Two rules settle every question of this shape. Count one collision domain for each separate segment, and count one broadcast domain for each router interface.

The Master Key: *Count ports for collision domains and interfaces for broadcast domains, and never let one cable be counted at both of its ends.*

REAL Question 67

The access point at a secondary school in Kyela sits in the corridor between two classrooms. Machines in both rooms reach it comfortably. Machines in one room cannot hear the other room at all, and the wall between them is the whole difficulty.

Account for the collapse in throughput this arrangement produces, and identify the mechanism that addresses it.

Model Answer

This is the hidden-node problem: the two rooms lie within range of the access point but not of each other, so each senses an idle channel while the other is transmitting, the two frames overlap at the access point and both are lost; the mechanism that addresses it is the request-to-send and clear-to-send handshake, which has the access point announce the reservation to stations that cannot hear the sender.

Carrier sense in the wrong place: a station tests the channel where it stands. A frame from the far classroom never reaches that point, so the test reports idle and is not lying; it is answering a question about the sender's position when the collision will happen at the access point.

What the access point receives: two transmissions arrive together and neither can be decoded. Both fail their check sequence and are discarded, so from the access point's side the traffic of two whole classrooms simply disappears.

Why it does not correct itself: neither sender heard the other, so neither learns why it failed. Each waits for an acknowledgement, times out, doubles its contention window and retries, and since the two rooms remain deaf to each other the same overlap keeps occurring.

How it looks from the office: throughput falls far below what the signal strength promises, retry counters climb in both rooms at once, and the machines nearest the corridor work perfectly while the two far groups suffer. Replacing the access point changes nothing, which is the clearest sign of all.

The handshake that answers it: the sender asks with a short request-to-send frame and the access point replies with a clear-to-send that every station in its own range hears, including the hidden room. That reply carries a duration, and each station hearing it defers for exactly that long.

What it costs: two extra frames on every exchange, which is expensive for short ones, so it is normally enabled only above a threshold frame size. It cures the hidden node without curing the opposite case, in which a station defers although its transmission would have harmed nobody.

The lesson reaches past radio. A test made at the sender answers a question about the sender, while a collision is a fact about the receiver, and no amount of careful listening at one end will settle a matter decided at the other.

The Master Key: *When carrier sense fails, ask where the sensing was done, because a medium is only ever idle at one place at a time.*

HOT Question 68

An engineer visiting a teachers' college in Makambako tells the class not to trouble themselves with CSMA/CD, since no equipment they will ever buy performs it. Half the room is relieved. The claim is very nearly right.

Assess the claim that CSMA/CD is obsolete, and reach a judgement on whether it still deserves a place in the course.

Model Answer

The claim is right about operation and wrong about understanding: on a switched network every port runs full duplex and no collision can occur, so the procedure never runs, yet it should still be taught, because it is the clearest example of contention resolved without a controller and because the faults it explains are faults a technician genuinely meets.

Why it no longer runs: each switch port gives its station a segment of its own, with separate paths for the two directions. Traffic in opposite directions on separate paths cannot interfere, so auto-negotiation settles on full duplex, collision detection is disabled, and the whole procedure sits unused.

The standard agrees: half duplex was carried through 1 Gbit/s for compatibility and then dropped, and 10 Gigabit Ethernet has no half-duplex mode at all. A method that the standard no longer defines for new speeds is fairly described as historical.

Where a collision still appears: a surviving hub, a port forced to half duplex by hand, or a duplex mismatch in which one end negotiated full and the other half. The half-duplex end then reports late collisions, the full-duplex end reports almost nothing, and the link delivers a small fraction of its rate.

What the knowledge buys: that mismatch cannot be diagnosed without the procedure. A student who knows it reads a late-collision counter and understands at once that one end believes it is sharing a medium; a student who does not will change the cable twice and the card once.

The idea outlives the method: listen, transmit, detect failure, wait a random and growing time, retry is a shape that recurs wherever a resource is shared with no controller, most obviously in the wireless method that displaced it. CSMA/CD is where the shape is easiest to see whole and at once.

The judgement: teach it, but teach it in its place. Present it as the method that made cheap shared Ethernet possible, as the ancestor of CSMA/CA, and as the explanation of a late-collision counter, while saying plainly in the same breath that a full-duplex switched port never performs it. Teaching it as current practice is the real error.

Obsolete equipment and obsolete ideas belong to different categories. The hub has gone from the store cupboard; the argument it forced a generation of engineers to settle has not gone anywhere.

The Master Key: *Ask of any dead technology what problem its death solved, because that answer is what the modern design is quietly assuming.*

HOT Question 69

Four hubs come out and four switches go in at a secondary school in Ifakara, at the cost of a term's equipment budget. The same news page, from the same machine at the same hour, takes as long as ever. The staff feel cheated.

Judge what this result tells you about where the constraint lies, and advise the school on what to measure next.

Model Answer

It tells you that the bottleneck was never inside the building: the switches removed contention on the school's own network, while the measurement chosen depends almost entirely on the shared line to the outside world and the distance to a distant server, neither of which has changed.

What the switches did change: every machine now has a collision domain to itself and a full-duplex path, so traffic between two machines in the laboratory no longer waits for the rest of the room. That is a real improvement, and the test performed does not touch it.

What sets the page-load time: the page crosses perhaps 50 m of the school's cable and then several thousand kilometres of somebody else's. If the line is 4 Mbit/s shared by 40 machines, that is 0.1 Mbit/s each, about 100 kbit/s, and no change inside the building can move it.

Delay as well as capacity: a page is not one transfer but many small exchanges, each waiting a full round trip to a distant server. That waiting is set by the length of the path, and a switch shortens no path.

Improving the wrong part: an upgrade shows up only in a measurement that depended on the part upgraded. The share of the total time owned by the school's own network was small, so removing all of it was bound to be invisible here.

The measurements to make instead: copy a large file between two machines on the same switch and compare it with the figure kept from before, which isolates the local network and will show the gain. Then run one download while the line is otherwise quiet, and again at the busiest hour.

What each answer would mean: if the quiet-hour figure is close to the line rate, the line is too small, and the remedy is more capacity, a caching proxy, or large downloads barred during lessons. If that figure is poor as well, the fault lies at the router or with the provider.

The verdict: the school was not cheated and the money was not wasted. It bought something real and then measured something else, so the switches were the right purchase made in the wrong order, and the next expenditure belongs to the outside line.

A network has one slowest part at a time. Improve any other part and the gain is real, but nobody in the building will feel it.

The Master Key: *Before buying, name the quantity you expect to change and write down its present value, because an upgrade you cannot measure is an upgrade you cannot defend.*

HOT Question 70

Forty tablets arrive at a college in Temeke and are put to work in one classroom on one access point. Everything behaves until about the twentieth device joins, and by the end of the lesson the room is unusable.

Advise the college whether a second access point will help this classroom, and defend the advice you give.

Model Answer

A second access point will help only if it is placed on a non-overlapping channel, with the power of both radios reduced and the devices genuinely divided between them; on the same channel it will not help at all, because CSMA/CA shares airtime rather than bandwidth, and a second transmitter on one channel divides the same airtime among more contenders.

What is actually scarce: the channel, measured in time. Only one station in a contention group may transmit at any instant and the rest defer, so capacity is airtime divided among whoever competes for it. The access point is rarely the limit; the air above the desks is.

Why a second radio on one channel fails: both access points and all forty devices would sense the same channel and defer to one another, so the contention group has grown rather than split. Two sets of beacons are added on top, and the room ends slightly worse off than before.

Why a different channel works: on non-overlapping channels the two cells contend separately, and the room then has two independent shares of airtime instead of one. At 2.4 GHz only three such channels exist, 1, 6 and 11, so the plan must fit within them and around whatever the neighbours already occupy.

Getting the devices to move: clients choose for themselves and cling to a familiar access point long after a better one appears. Reducing the transmit power of both radios makes each cell smaller and the choice sharper. Twenty devices on each is the object; thirty-nine on one is not.

The slowest client sets the pace: a device in the far corner transmits at a low rate and so holds the channel far longer than a near one sending the same frame. Smaller cells help twice, by splitting the group and by keeping every device on a faster rate.

What will not improve: both access points share one cable back to the switch and one line to the outside. If the classroom's traffic comes mostly from the Internet, dividing the radio side changes nothing anyone can feel.

The judgement: buy the second access point, but only as part of a plan: a different non-overlapping channel, reduced power on both, one network name so that devices roam, and preferably one radio on 5 GHz where more channels exist. Without those conditions, decline the purchase.

The question was never how many access points the room contains. It is how many independent conversations the air above those desks can carry at the same moment.

The Master Key: *Count channels rather than boxes, because two radios on one channel are one network wearing two hats.*

VLANs and network segmentation

A switch leaves the factory as one network: every port in the same broadcast domain, every machine able to reach every other. A VLAN undoes that in configuration rather than in cable, cutting one switch into several logical networks that behave as though they had been built from separate hardware. This section settles what a VLAN is, how the four-byte tag that carries it between switches is laid out, what the arrangement buys a school or an office, and, just as firmly, what it does not buy. It assumes you already know what a broadcast domain is.

BINDER Question 71

One switch in the cabinet of a bank branch in Kinondoni serves three offices that must not see one another's traffic. There is one cabinet, one set of cables, and no budget for a second of anything. The separation still has to happen.

Define a virtual local area network and explain how it differs from a network separated by physical means.

Model Answer

A VLAN is a broadcast domain created in the configuration of a switch rather than in cable, so that a chosen group of ports behaves as one logical network and hears none of the broadcasts of any other group on the same hardware; physical separation reaches the same isolation with different switches and different wire.

A domain drawn in software: the switch holds a table of which port belongs to which VLAN, and forwards a frame only to ports carrying the same identifier as the one it arrived on. A broadcast entering VLAN 10 reaches the other VLAN 10 ports and stops there.

How membership is decided: usually by port. A technician assigns a socket to VLAN 20, and whatever is plugged into it joins VLAN 20. Membership can instead follow the hardware address, the protocol spoken, or an identity supplied at login, but port-based assignment is what school equipment uses.

The difference that matters: a physical boundary is made of things you can trip over, so changing it means moving cable. A VLAN is a line in a file, so it can be redrawn from a keyboard in another town, and redrawn by accident just as easily.

What the two still share: one switch, one power supply, one backplane. Physical separation gives independent failure as well as independent traffic; a VLAN gives only the second, so when the switch dies every VLAN on it dies with it. That is the honest price of the saving.

Grouping across a building: membership follows the function rather than the floor. In Figure 3.14 three departments share one switch, and those same VLANs can be carried to a switch in the next block, so a member of staff who moves upstairs stays in her own network.

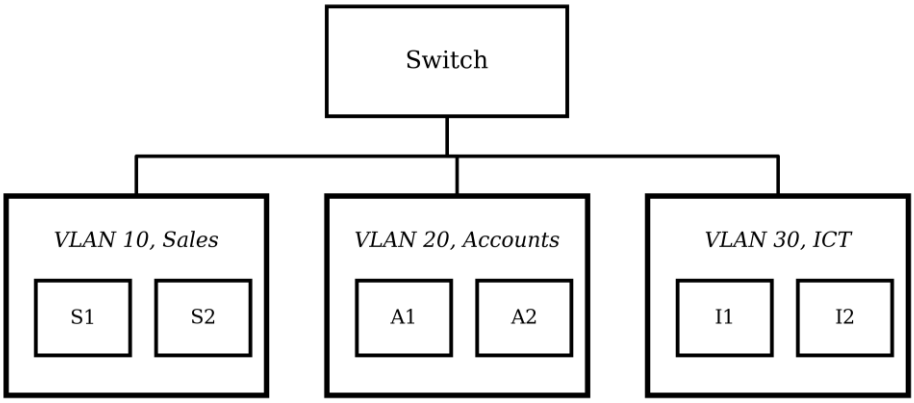


Figure 3.14: *One switch carrying three VLANs. Each is its own broadcast domain, and traffic between them must pass through a router.*

The copper has not moved; the boundary has, into a file that must now be kept as carefully as the cable plan.

The Master Key: *A VLAN is a broadcast domain you can edit; a physical network is one you must rewire.*

BINDER Question 72

A frame captured on the link between two switches at a college in Njombe is four bytes longer than the standard permits. The four extra bytes sit not at the front, not at the end, but immediately behind the source address.

Using Figure 3.15, describe the fields of the 802.1Q tag and determine how many usable VLAN identifiers it provides.

Model Answer

The tag is four bytes inserted after the source address, carrying a two-byte tag protocol identifier of 0x8100 that announces its presence, three bits of priority, one drop-eligible bit and a twelve-bit VLAN identifier; twelve bits give 2^{12} = 4,096 values, of which 0 and 4,095 are reserved, leaving 4,094 identifiers usable.

Where the four bytes go: the tag is inserted between the source address and the type field, so both hardware addresses stay exactly where every device expects to find them. Only equipment that reads beyond the addresses needs to understand tagging at all, which is why the standard chose that position.

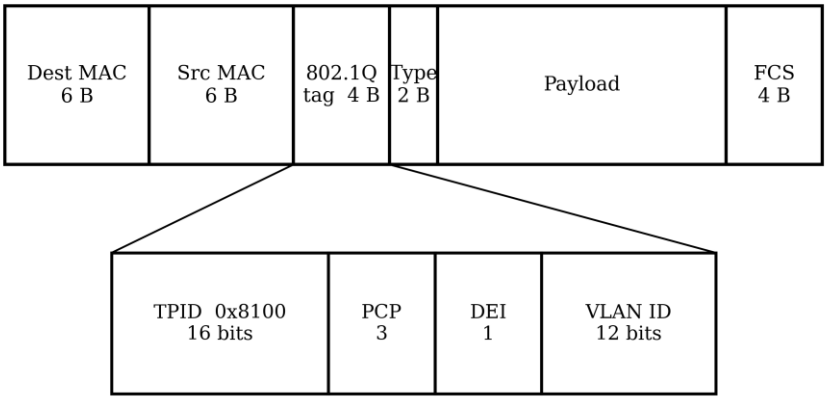
The tag protocol identifier: the first two bytes carry the fixed value 0x8100, sitting in the place an untagged frame uses for its type field. A receiver that finds 0x8100 there knows the next two bytes are tag control information and the real type field lies four bytes further on.

Priority and drop eligibility: three bits of priority code point give $2^3 = 8$ traffic classes, so a loaded switch can send a voice frame ahead of a queued file transfer. The single drop-eligible indicator bit marks a frame that may be discarded first when a queue overflows.

Counting the identifiers: twelve bits give 2^{12} = 4,096 combinations. Identifier 0 means the tag carries priority only and claims no membership, and 4,095 is reserved, so 4,096 − 2 = 4,094 and the usable range runs from 1 to 4,094. Quote the reservation with the number or the answer looks guessed.

$$\text{Usable VLAN IDs} = 2^{12} - 2 = 4{,}096 - 2 = 4{,}094$$

What the tag costs: the four fields total 16 + 3 + 1 + 12 = 32 bits, which is the four bytes the figure shows. Added to a frame already at its 1,518-byte maximum, that gives 1,518 + 4 = 1,522 B, so every device on the trunk must accept the longer frame or discard it as oversized.



The tag adds four bytes, so a tagged frame may reach 1522 B. Twelve ID bits give 4094 usable VLANs.

Figure 3.15: *The 802.1Q tag. Four bytes inserted after the source address, of which twelve bits carry the VLAN identifier.*

Four bytes, and all of them spent on one sentence: this frame belongs to that logical network, and the next switch should treat it accordingly.

The Master Key: *Twelve bits give 4,096 values and two are reserved, so write 4,094 with the reservation beside it.*

BINDER Question 73

Two cables leave the switch in a school office in Kyela. One runs to a classroom wall socket and carries a single VLAN; the other runs upstairs to a second switch and carries all of them. Swapping the two costs an afternoon.

Distinguish an access port from a trunk port, stating in each case what the switch does to the tag.

Model Answer

An access port belongs to one VLAN and faces an end device, tagging every frame that arrives from it and stripping the tag from every frame sent back out; a trunk port faces another switch or a router, carries the frames of many VLANs, and leaves the tag in place so the far end can tell them apart.

What an access port serves: an end device. A workstation, a printer or an access point knows nothing about VLANs and must never be shown a tag. The port is given one VLAN, and the tagging happens inside the switch.

What a trunk port serves: another switch, or the router that joins the VLANs. One cable carries them all, and the tag is the only thing telling the far end which is which. Without trunking a site would need one cable between cabinets per VLAN.

The native VLAN: a trunk keeps one VLAN untagged, the native VLAN, so an untagged frame arriving on the link still has somewhere to go. Both ends must be given the same one, because if they disagree the traffic of one network is quietly delivered into another.

Telling one from the other: capture on the link. Untagged frames of a single network mean an access port; frames carrying several identifiers mean a trunk. The tag, not the cable, settles which kind of port you are looking at.

The mistake to avoid: configuring an uplink between two switches as an access port. Everything in the one VLAN it carries works, everything else fails, and no indicator turns red. Whoever diagnoses it must already suspect the port type, so label uplink sockets.

An access port hides the VLAN from the device; a trunk advertises it to the switch. Each is wrong in the other's place.

The Master Key: *Decide what is at the far end of the cable, because that alone settles whether the tag should stay on or come off.*

BINDER Question 74

A tester pronounces both links perfect. Two machines in adjacent sockets of one switch at a district council office in Lindi pass not a single packet between them, and the technician is about to condemn the switch. It is doing what somebody configured.

Explain why two hosts placed in different VLANs on the same switch cannot reach one another without a router, and identify the equipment that resolves it.

Model Answer

A switch forwards a frame only to ports in the VLAN it arrived on, and takes that decision at layer 2 without reading an IP address, so a frame from VLAN 10 has no path to a VLAN 20 port; crossing between them needs a device that forwards on logical addresses, which means a router or a layer-3 switch.

The rule the switch obeys: each VLAN is a separate broadcast domain, so every layer-2 message stays inside it, including the address resolution request a host must send before it can build a frame at all. No reply comes back, and a frame never built cannot be forwarded.

Why the switch cannot help: its table maps hardware addresses to ports within one VLAN. It holds no routes and no notion that two logical networks might be joined, so it has nothing to decide with. Refusing the traffic is correct behaviour, not a fault to repair.

What a router adds: it reads the destination logical address, works out which network owns it, and builds a fresh frame on the interface serving that network. Two VLANs are two IP networks, so crossing between them is an ordinary routing decision.

Router on a stick: one physical link set as a trunk carries every VLAN to the router, which is given a logical subinterface per VLAN, each holding the address those hosts use as their gateway. One cable then serves every VLAN on the switch.

The layer-3 switch: the same routing built into the switch and done in hardware on its own backplane. Traffic between VLANs never leaves the chassis, so it is far faster than the trunked router, which is why a site with many VLANs buys one.

Nothing at Lindi was broken. The technician had found a boundary somebody drew deliberately and then forgot to write down.

The Master Key: *A VLAN is a wall built at layer 2, and only a layer-3 device can put a door in it.*

REAL Question 75

One managed switch of twenty-four ports at a secondary school in Magu carries the laboratory, four staff offices and the records office, whose file server holds this term's results. They all share that one switch, so any student machine can reach the server.

Design a VLAN scheme for the school, allocate the ports, and state what else must be bought or configured before staff can reach the examinations file server.

One managed switch of 24 ports, and no budget for a second.

12 student machines in the laboratory.

8 staff machines in four offices.

3 machines in the examinations office, one of them the file server.

Staff must reach the file server. Students must not.

Model Answer

Define three VLANs on the one switch, VLAN 10 for the students, VLAN 20 for the staff and VLAN 30 for the examinations office, allocating 12 + 8 + 3 = 23 access ports and keeping the twenty-fourth as a trunk; give each VLAN its own IP subnet, and then add the missing item, a router or a layer-3 switch, because the switch alone can never carry a staff request across into VLAN 30.

Group by rights, not rooms: the three groups differ in what they may do to the same server, so they become the three VLANs. Where a machine physically sits stops mattering, which is the point: a staff member who moves into the laboratory to teach keeps her own network as soon as her port is reassigned.

The port allocation: ports 1 to 12 to VLAN 10, ports 13 to 20 to VLAN 20, giving $20 - 13 + 1 = 8$, and ports 21 to 23 to VLAN 30, giving $23 - 21 + 1 = 3$. Port 24 becomes the trunk, so $12 + 8 + 3 + 1 = 24$ and the switch is exactly full.

One subnet for each: a VLAN is a broadcast domain, so it is also a separate IP network with its own address range and its own gateway on the router. Hand out each range from a scope of its own, and a machine then announces which VLAN it is in simply by the address it holds.

What must still be bought: the switch cannot move a frame from VLAN 20 to VLAN 30, so as it stands the design walls the examinations office off from the staff who need it exactly as thoroughly as from the students who must not have it. A router trunked on port 24, or a layer-3 switch in place of the present one, is the missing item.

The rule that isolates: once routing joins them, the isolation no longer rests on the VLAN but on the rule at the router. Permit VLAN 20 to that one server for file sharing alone, deny VLAN 10 to VLAN 30 outright, and record the denials so that an attempt is evidence rather than a rumour.

Label it before you leave: write the port allocation on the panel and in the change log. An undocumented VLAN plan survives until the first person moves a patch lead to a free socket, discovers the machine has lost the server, and concludes that the network is unreliable.

One switch, one afternoon and one router put the examinations office as far from the laboratory as another building would, and only the router had to be paid for.

The Master Key: *Settle the groups, then the ports, then the single device that lets the right group across.*

REAL Question 76

A student on industrial attachment at a contractor on Unguja hands in a report on six weeks of configuring an office network. It is neat, and his supervisor signs it without reading closely. Three of its sentences become faults long after he has gone.

Identify the error in each statement, give the correct version, and explain why the correction is right.

1. A VLAN encrypts the traffic inside it, so anything sent within one VLAN is private.

2. Two VLANs on the same switch cannot be reached from one another under any circumstances.

3. A trunk port belongs to no VLAN, so no VLAN can be assigned to it.

Model Answer

A VLAN changes where a frame may go and never touches the bytes inside it, so nothing is encrypted; VLANs are isolated only until a router or layer-3 switch is deliberately configured to join them, which is how almost every real network runs; and a trunk does have a VLAN of its own, the native VLAN, whose frames it carries untagged.

Statement one, the encryption: the tag is a plain twelve-bit number in the clear, and the payload behind it is exactly as readable as it was before. Anyone with a port in that VLAN, or a capture taken on the trunk carrying it, reads every byte. Confidentiality comes from encrypting the traffic itself, and from nothing a switch does.

Why that error is expensive: it is not merely inaccurate, it is a purchasing decision. A manager who believes the VLAN has already made the traffic private declines to buy the certificate or configure the encrypted service, and the network ships payroll across the site in the clear for years without anybody revisiting the sentence.

Statement two, the word never: isolation is the default, not a law of physics. A router or a layer-3 switch joins two VLANs whenever someone configures it to, and most sites depend on exactly that, because departments with nothing at all to exchange would not need to be on one network in the first place. What is true is the narrower claim: the switch by itself cannot do it.

Statement three, the trunk: a trunk carries many VLANs and it also has one of its own. Its native VLAN is the one whose frames travel untagged, and it is where the switch puts any untagged frame that arrives. Leave that at the factory default on both ends and a frame carrying two tags can be delivered into a VLAN its sender never belonged to.

The habit that prevents them: every sentence promotes a default into an absolute. Ask two questions of any boundary: what does it do on its own, and what has somebody configured on top of it? A VLAN on its own separates. The encryption, the routing and the native VLAN are all decisions, and every decision could have been made differently.

None of the three sentences is stupid, and that is what makes them dangerous. Each is a true statement about the default, written as though nobody would ever change it.

The Master Key: *Say what a thing does by itself before you say what it guarantees, because the guarantee always belongs to somebody's configuration.*

HOT Question 77

A supplier's quotation for a district hospital at Chake Chake sets out its work under four headings. Under the fourth, Security, there is a single line: configure VLANs. The price is fair and the work genuine. It is the heading that is wrong.

Assess whether a VLAN should be counted as a security measure, and commit to a judgement.

Model Answer

Count it as one, but as a security measure of exactly one kind: a VLAN decides which machines can reach which, and that is the whole of its contribution, since it supplies no confidentiality, no proof of who is at the keyboard and no defence against anything already inside the VLAN, so a quotation offering VLANs under Security and nothing else offers one control where a hospital needs several.

What it genuinely buys: reachability is the first question any attacker asks, and a VLAN answers it. A machine in the outpatient VLAN cannot open a connection to the records server without passing the router, where a rule can refuse it and a log can record the attempt. Cutting the number of machines reachable at all is a real reduction in exposure.

Containment of the broadcast attacks: the layer-2 attacks that matter most, address resolution poisoning among them, work by broadcast, and a broadcast stops at the edge of its domain. Splitting one flat hospital network into several confines each of them to the department where it started.

It does not encrypt: no field of the tag touches the payload. A VLAN is a direction written on an envelope, not a seal across the flap, and a captured patient record is as readable as it was on a flat network.

Nothing stops what is inside: malware on one machine spreads freely to every other machine in the same VLAN, and those are precisely the machines its user works with all day. Segmentation limits how far the damage travels; it does nothing to reduce the chance of the first infection.

The boundary can be defeated: a trunk left to negotiate itself, or a native VLAN left at the factory value, lets a frame carrying two tags be forwarded into a network its sender does not belong to. The wall holds only where trunks are set explicitly and the native VLAN is one that no department uses.

What it does	What it does not do
Cuts one switch into several broadcast domains	Encrypt a single byte of what it carries
Forces traffic between groups past a router, where a rule can refuse it	Stop malware spreading inside one VLAN
Confines broadcast-based attacks to one domain	Prove who the user at the keyboard is
Moves a user between groups without recabling	Survive the failure of the switch holding it
Groups users by function across several buildings	Configure itself safely, as a default native VLAN shows

Table 3.9: *What a VLAN gives you and what it does not.*

The judgement: put VLANs in the design and refuse to let them stand alone in the security section. Name the control for what it is, segmentation, then buy the encryption, the authentication and the patching it does not replace. Treat any quotation where the word security is followed only by the word VLAN as incomplete, and say so before signing.

The quotation is not dishonest. It is one line where four are wanted, and the four it omits are the ones that still protect a patient record after somebody has reached it.

The Master Key: *Ask what a control does to reachability and what it does to the bytes, and accept that a VLAN answers only the first.*

HOT Question 78

Twelve machines in the laboratory of a school in Ifakara are to be split into two networks: the teacher has read that segmentation improves performance. One proposal is a second switch. The other is two VLANs on the switch already in the cabinet.

Judge whether a twelve-machine laboratory is better served by VLANs or by a second switch, and defend the choice.

Model Answer

Configure the VLANs, and do it for isolation rather than speed: with 12 machines there is no performance problem to solve, a second switch simply uplinked to the first would not even produce a second broadcast domain, and the VLAN costs nothing to try and one command to undo, so it wins on every ground but one, that it needs a managed switch and a technician who documents it.

There is no performance case: twelve machines produce a broadcast load a switch handles without noticing. Splitting 12 // 2 = 6 machines from the other six halves a figure never large enough to measure, and no student will see a page load faster. Any argument for segmenting this room must be about who may reach whom.

The second switch misleads: plug switch two into switch one and the two remain one broadcast domain, so nothing is separated. The boundary appears only if the second switch is given its own router interface, which means buying the switch and then the routing as well. Hardware bought for a boundary that hardware does not provide is the most expensive way to stay where you were.

What the VLAN really costs: a managed switch, which the school either has or does not, and an hour at the console. Nothing is mounted, nothing is powered and nothing new can fail. If the split proves a mistake, one command puts the ports back and no equipment sits unused in a cupboard.

The condition attached: a VLAN needs somebody who can configure it, document it and still be traceable in a year. An undocumented VLAN turns into a fault nobody can explain, because the machine that has stopped reaching the printer is simply the one whose port was quietly reassigned.

When the switch would win: buy hardware when the ports have run out, when a second room needs a cabinet of its own, or when one group must keep working while the other cabinet is down. Those are physical requirements, and a VLAN answers none of them, because it shares the chassis whose failure is the risk.

The verdict: two VLANs on the existing switch, one holding the teacher's console and one the remaining 12 − 1 = 11 student positions, and buy nothing. Revisit it when the laboratory outgrows the room, not when a supplier calls. If the switch is unmanaged, change nothing at all: twelve machines in one room is not a network with a problem in it.

Segmentation is a treatment, and a treatment given to a healthy network leaves only its side effects behind.

The Master Key: *Segment for isolation, never for speed, and never buy a box to obtain a boundary that a configuration file already gives you.*

HOT Question 79

Forty VLANs have grown up in four years on a district office campus at Kahama, one for every department that asked. All forty route through one router on one trunk cable. Every complaint is the same: slow, but only across departments.

Determine what will fail first on this design, and recommend a remedy.

Model Answer

The trunk to the router fails first, and it fails as congestion rather than as an outage: every packet crossing between VLANs travels that one cable twice, so a 1,000 Mbit/s trunk delivers at most 1000 / 2 = 500 Mbit/s of inter-VLAN traffic, which is 500 / 40 = 12.5 Mbit/s a VLAN on average; move the routing into a layer-3 switch, and cut the amount of traffic that has to be routed at all.

Why the cable counts twice: a packet from one VLAN to another leaves the switch on the trunk, reaches the router, is routed, and comes straight back down the same cable into a different VLAN. The trunk therefore carries it in both directions, so halve the rated speed before comparing it with anything: 1000 / 2 = 500 Mbit/s.

What each VLAN is left: 500 / 40 = 12.5 Mbit/s each, and that average flatters the design badly, because the two or three departments that move files in bulk will take far more than their share and everybody else will feel it. This is the figure somebody should have worked out when the twentieth VLAN was created.

The router is next: a router forwarding in software has a ceiling in packets a second that sits well below the bit rate of its port, and forty subinterfaces mean forty sets of rules to consult. Its processor climbs and latency rises long before any interface reports a single error, which is exactly why nothing appears broken.

One failure ends all forty: one router, one trunk, one optic. Any of the three failing leaves forty VLANs perfectly healthy in isolation and unable to reach one another or the outside world. Four years of growth added forty broadcast domains and not one path of redundancy.

The verdict, and the remedy: route in a layer-3 switch. The same decision is taken in hardware on the backplane, so inter-VLAN traffic never leaves the chassis, the trunk drops out of the path entirely, and the ceiling rises by more than an order of magnitude. Keep the router for the link to the outside world, where its rule sets belong.

The remedy nobody proposes: route less. Put a shared server in the VLAN of the department that uses it most, and merge VLANs created for teams that exchange traffic constantly, since two departments talking all day were never really two broadcast domains. Every VLAN added is another set of journeys through the router.

Nothing here will fail with a red light. The design simply gets slower each time a department is added, until somebody remembers to count the trunk twice.

The Master Key: *Count the trunk twice, divide by the number of VLANs, and the answer tells you the day to stop routing on a stick.*

The network layer and its design issues

The two layers below this one carry a frame to the next device on the same wire and then stop. Layer three is the first with any business beyond the local network, and every hard question about joining networks together is settled here. This section states what the layer is for, separates the protocol that carries the data from the protocols that decide where it goes, sets out the design questions any network layer must answer, and explains why the service it finally offers promises so remarkably little.

BINDER Question 80

A payment at a shop counter in Kyela reaches a bank's computer in another region across a leased line, two provider networks and a length of fibre. Nothing in the shop sees past the router on the wall, yet the payment arrives.

State the functions of the network layer, and explain what separates it from the data link layer beneath it.

Model Answer

The network layer delivers a packet from a source host to a destination host across any number of networks in between, by giving every host a logical address, determining a path, forwarding each packet along it and fragmenting whatever exceeds the next link; the data link layer reaches only the next device on one wire.

End-to-end delivery: the layer answers for the whole journey, from the machine that composed the packet to the machine named in its destination field. The sender never learns which networks lie between the two, and does not need to.

Logical addressing: every host is given an address in software whose leading part names a network and whose remainder names a host inside it. The sender writes it once and it survives to the far end, unlike the hardware address, which is replaced at every hop.

Path determination and forwarding: determining the path means building a table of which networks are reachable through which neighbour, by hand or by a routing protocol. Forwarding is the separate, per-packet act of reading the destination, matching it against that table and sending the packet out of one line.

Fragmentation when needed: a packet larger than the next link will accept is divided into pieces that fit and marked so they can be put back together. Only the destination reassembles them.

Where layer two stops: the data link layer is bounded by one physical network. Its address is flat, it means nothing off that wire, and it is rewritten in every new frame. A switch works inside that boundary; a router is the device that crosses it.

Layer three delivers packets and layer two delivers frames, so an answer that has the network layer delivering frames has merged the two things it was asked to separate.

The Master Key: *The data link layer knows the next device; the network layer knows the destination.*

BINDER Question 81

Inside one router in a district office at Sengerema two different programs run at the same moment. One reads the address on every packet and pushes it onward. The other talks to the routers next door about which networks each can reach.

Distinguish between a routed protocol and a routing protocol, and name two examples of each.

Model Answer

A routed protocol defines the address and the packet in which a user's data travels between networks, and IPv4 and IPv6 are the two in current use; a routing protocol carries no user data but exchanges reachability information between routers so that each can build the table those packets are forwarded by, and RIP and OSPF are examples.

What a routed protocol defines: the format of the logical address, the layout of the header wrapped round the data, and the rules for delivering it. Anything that gives a host an address a distant sender can write down, and defines a packet to carry to it, is a routed protocol.

What a routing protocol defines: the conversation between routers. Its messages describe networks and the cost of reaching them, and they are meant for other routers rather than for any user. It fills the table; it never carries the traffic.

Which devices run each: every host and every router runs the routed protocol, because without it nothing can be addressed. Routing protocols run on routers, and a laptop in a laboratory runs none of them.

The dependence between them: a routing protocol sends its own messages inside routed-protocol packets, so it cannot work without one. The dependence does not run both ways: a small network whose tables are written by hand needs no routing protocol at all.

The trap in the wording: the two names differ by one syllable and the marker is looking for exactly that. IP is not a routing protocol, and RIP and OSPF are not routed protocols.

Question	Routed protocol	Routing protocol
What does it carry?	A user's data, packet by packet	Route information, and no user data
What does it define?	The address format and the packet header	The messages routers exchange about paths
Which devices run it?	Every host and every router	Routers alone; a host runs none
Which are the examples?	IPv4 and IPv6	RIP and OSPF

Table 3.10: *Routed protocols and routing protocols.*

Keep the two columns of Table 3.10 apart with one question: does the protocol carry somebody's data, or only news about the network?

The Master Key: *A routed protocol carries the load; a routing protocol draws the map the load is carried by.*

BINDER Question 82

A radio station in Chake Chake sends its morning bulletin over the network to listeners across the channel. Nothing was agreed in advance with any router, no share of any link was set aside, and no device along the way was warned.

Explain what is meant by connectionless delivery at the network layer, and state three things the layer therefore does not promise.

Model Answer

Connectionless delivery means that every packet is handled as an independent unit, carrying a full destination address and routed on its own merits with no arrangement made beforehand between the two ends, so the layer does not promise that a packet will arrive, that packets will arrive in the order sent, or that each will arrive only once.

No setup and no memory: no call is placed and nothing is agreed before the first packet leaves. A router records networks, not conversations, so it cannot tell whether the packet before it is the first of a million or the last.

Every packet self-contained: each carries the complete destination address, which is what allows it to be forwarded without reference to any other packet. That independence is the strength of the service and the source of all three failures below.

Arrival is not promised: a router whose outgoing queue is full discards packets, and a corrupted one is dropped. Nothing in the layer resends what was lost, and nothing obliges it to tell the sender.

Order is not promised: two packets sent a moment apart may leave by different lines if the tables change between them, and a longer path takes longer, so the second may overtake the first.

Single delivery is not promised: a momentary loop, or a lower layer retransmitting after a late acknowledgement, can put two copies of one packet on the network, and both may be delivered.

Who repairs the shortfall: the choice is deliberate, because a router that promises nothing stays simple and fast. Where the guarantees are needed they are supplied at the ends by TCP, and applications preferring speed choose UDP.

The usual phrase for this service is best effort, and it should be read strictly: the layer will try, and will promise nothing further.

The Master Key: *Assume nothing about a packet you have not received: not that it came, not that it came in turn, not that it came once.*

BINDER Question 83

Two hundred networks, run by different people, built from different equipment, must be joined so any machine can reach any other. The same questions must be answered before a single packet moves, whether the result serves one campus in Kinondoni or the world.

Describe four of the design issues that the designer of any network layer must settle, stating in each case what the problem actually is.

Model Answer

A network layer cannot be built until its designer has settled how hosts are addressed, how a path is chosen, how traffic is divided for carriage, what is done about errors and about overload, how competing traffic is ranked and how any of it is protected.

Addressing the whole internetwork: every host must have an address unique across the whole internetwork that also says where it is. The second requirement is the hard one: unless the address is hierarchical, a router would need one table entry per host rather than one per network.

Routing without the map: a router with several outgoing lines must choose one, knowing only what its neighbours have told it and never seeing the whole path. The design must make many independent hop-by-hop choices add up to a sensible route.

Packet switching: traffic is cut into packets switched one at a time rather than sent as an unbroken stream, so a link is shared between many conversations instead of held by one, and a loss costs one packet rather than a whole message.

Congestion control: more packets arrive at a router than its outgoing line can carry. The queue lengthens, delay rises and packets are discarded; worse, the discards provoke retransmissions, so the load grows when the network can least bear it.

The three that remain: error handling asks what becomes of a packet corrupted, looped or dropped; quality of service asks how traffic that cannot wait is ranked above traffic that can; and security asks how a packet is protected on a path its sender cannot see.

The eighth question: interoperability underlies the other seven. The layer exists to join networks of unlike technology, so every answer must hold whether the next link is fibre, copper or radio.

Issue	The problem	How it is addressed
Addressing	A distant host must be nameable by a sender that cannot see it	A hierarchical logical address whose leading part names the network
Routing	One outgoing line must be chosen without sight of the whole path	A table written by hand or built by a routing protocol
Packet switching	One conversation must not hold a shared link for itself	Traffic is cut into packets switched independently
Error handling	Packets are corrupted, looped or quietly dropped	A header check, a hop limit, and messages that report the failure
Congestion control	Arrivals exceed what the outgoing line can carry	Queue management, discard, and signals asking senders to slow
Quality of service	Traffic that cannot wait shares a queue with traffic that can	Classify and mark the traffic, then schedule the queues by class
Security	Any packet may be read, altered or forged in transit	Filtering at the boundary and encryption of the payload

Table 3.11: *The design issues a network layer must settle.*

The issues in Table 3.11 are not independent. Discarding packets is a congestion remedy that creates an error-handling problem, and ranking traffic creates a security one.

The Master Key: *Name the problem before the mechanism: address, route, switch, detect, throttle, rank, protect.*

REAL Question 84

Every night a backup copies to the regional server over the one link the district office in Nzega has to its provider, and has never failed. At eleven in the morning that link cannot hold a video meeting together for two minutes.

Identify the network-layer design issues implicated in this pattern of failure, and account for the difference between the two applications.

Model Answer

Nothing is broken: the issues implicated are congestion control and quality of service, with the best-effort nature of packet switching and the store-and-forward delay of every router behind them, and a file transfer survives delay, jitter and loss only because a higher layer repairs all three in a way a live call has no time for.

What a transfer tolerates: the backup asks only that every byte arrive eventually. A lost packet is sent again, a late one waits in a buffer, and order is restored before anything is written to disk. Its only measure is how much got through by morning.

What a call cannot tolerate: a packet of speech that arrives after its moment has passed is worthless, and asking for it again is worse than useless. The call is measured by delay, by the variation in that delay, and by loss, and fails on any of the three.

Congestion control, the first issue: at eleven in the morning the link is full. A router meeting more arrivals than it can send does not fail; it queues, and a queue turns spare capacity into delay long before it turns it into loss. At night the queue is empty.

Quality of service, the second: one queue serves both kinds of traffic, so every packet of the call waits behind whatever arrived first. Nothing in the layer ranks a conversation above a file copy unless somebody has classified the traffic and told it to.

Best effort, the third: each packet is independent, nothing was reserved for the meeting, and order was never promised. The layer is breaking no undertaking, because it gave none; the meeting simply needs guarantees from a service designed not to make any.

Store and forward, the fourth: a router may not send the first bit of a frame until it has received the last, because the check that proves it intact sits at the end. A 1,500-byte frame is 12,000 bits, which at 100 Mbps takes 120 microseconds, so eight routers add 960 microseconds, nearly a millisecond, before one queue is counted.

What to tell the office: the fault lies in the service the network layer was designed to give, not in the equipment. The two honest remedies are to classify and prioritise the call, or to buy enough capacity that no queue forms at eleven.

The pattern is worth memorising, because it recurs everywhere. When one application on a link works perfectly and another fails, the difference is almost never the cable.

The Master Key: *Delay is invisible to a file and fatal to a conversation, so ask what the traffic is for before you ask what is broken.*

REAL Question 85

The proposal for re-equipping the computer laboratory of a secondary school in Magu runs to eleven pages and is well presented. The head of department is no network specialist, and asks the ICT teacher to read the paragraph below before the school commits itself.

Advise the buyer what the mention of IPX tells them about this proposal, and state what they should require instead.

Protocol support: the equipment supports all routed protocols in current use, namely IP and IPX.

Routing: RIP and OSPF are both available, and either may be enabled on any interface.

Delivery: fourteen routers, configured on site, with thirty-six months of parts and labour.

Model Answer

The paragraph is out of date rather than dishonest: IPX belonged to Novell NetWare and has not been in general use since the early 2000s, so a document that offers it as a protocol in current use has been copied forward from an older document, and the buyer should require the proposal to name IPv4 and IPv6 instead.

What IPX actually was: the routed protocol of Novell NetWare, doing for a NetWare network what IP does for the Internet: its own address format, its own packet header, its own delivery rules. It really was a routed protocol, which is why the sentence reads so plausibly.

Why it is obsolete: Novell's own products moved onto IP, and the last networks running IPX were retired in the early 2000s. Nothing the school will buy, and nothing it will ever connect to, speaks it. Support for it is not a feature but a fossil.

The pair that is current: IPv4 and IPv6 are the routed protocols of any network built today. A proposal that does not mention IPv6 anywhere is quietly offering equipment with a short working life, and that omission matters far more to the school than the presence of IPX.

What the paragraph gets right: RIP and OSPF are correctly given as routing protocols and correctly kept apart from the routed ones, so the supplier has not confused the two categories. The fault is the currency of one example, not the classification into which it was put.

What it implies elsewhere: a paragraph unrevised for twenty years is evidence about the whole document. The buyer should check the model numbers against the supplier's present catalogue, and satisfy themselves that the warranty and the delivery terms are current commitments rather than inherited text.

The question to put back: ask the supplier to confirm in writing that the routers run IPv4 and IPv6 together on the same interfaces, and to name the routing protocols supported for each. A supplier who can answer that in a sentence is quoting from the equipment; one who cannot is quoting from a file.

Advise the head of department to proceed, but with that confirmation in hand. The proposal is probably still the right one; it simply has not been read by its author for a very long time.

The Master Key: *An obsolete example is never a small error, because it dates the document that contains it.*

HOT Question 86

The old telephone exchange in Ifakara worked the other way about. Before a word was spoken a path was reserved from end to end and held for the whole call, whether anyone spoke or not. The network layer's designers declined to copy that.

Assess the argument that the network layer ought to have been designed as a connection-oriented service, and reach a judgement.

Model Answer

The argument identifies real weaknesses and still fails: connection-oriented delivery would buy ordering and reservation at the price of state in every router, fragility at every failure and a setup delay imposed on traffic that never asked for one, so the original choice should stand.

What the alternative would be: a virtual circuit. A setup packet lays a path before any data moves, every router along it stores an entry for that conversation, and each later packet carries a short circuit number instead of a full address and follows the same route.

The case for, on ordering: packets that follow one path cannot overtake one another, so they arrive in the order sent. Two of the three things the connectionless layer refuses to promise would come free.

The case for, on reservation: because the path is known before the first packet, capacity along it can be set aside, and a request that cannot be honoured can be refused at setup rather than accepted and then ruined. That is what a live conversation needs.

The case against, on state: a router would hold an entry for every conversation crossing it rather than one for every network beyond it. Memory would then grow with users rather than with topology, and a restart would destroy every circuit at once instead of costing a few packets.

The case against, on fragility: a fixed path is a single point of failure. One cut fibre ends every circuit crossing it, and each must be set up again. With independent packets the next is simply forwarded another way, and most users never learn that anything happened.

The case against, on cost: reliability and ordering can be added at the two ends by TCP, at no cost to the routers between. Connections in the middle charge every user for guarantees, including the majority whose traffic would rather have the setup time back.

The judgement: keep the layer connectionless. Concede the narrow claim at once, that a reserved circuit serves a real-time call better than best effort ever will; but a service used by everything should offer the weakest guarantee that is still useful, because an application can add what it needs above and can never remove what the middle has imposed.

The same reasoning explains why the layer says nothing about the medium below or the application above. Each refusal to decide is what allowed everything else to be built on top.

The Master Key: *Design the layer that promises least, because a promise made in the middle can never be given back at the edge.*

HOT Question 87

A notebook in the ICT office of a secondary school in Njombe records every fault since the laboratory opened. It is not a record of broken equipment. Almost every entry argues about which machine is which, or what may reach what.

Judge which single network-layer design issue causes the most trouble in a school network of this kind, and defend the choice against the strongest rival to it.

Model Answer

Addressing causes the most trouble, because it is the one design issue a school must settle entirely for itself, it is settled once and then built upon, and its failures are intermittent and disguise themselves as faults at other layers, whereas congestion, the obvious rival, is a known quantity with a purchasable remedy.

Narrowing the field: of the issues at layer three, most are settled for the school by the protocols it buys; nobody in Njombe designs error handling or packet switching. What a school decides for itself is its addressing, its routing, how it copes with a shared link, and how it protects itself.

The rival at its strongest: congestion looks like the winner. Every complaint is that the network is slow, the whole laboratory competes for one modest link to the provider, and the worst hour of the week is the hour the results are uploaded.

Why the rival loses: congestion is understood by everybody, including the bursar who must pay for the cure. It is predictable, it worsens smoothly, and more capacity fixes it exactly. An issue whose remedy fits on a purchase order is not the one that consumes the year.

Why addressing wins: the address plan is normally invented by whoever installed the first switch, is never written down, and every later decision rests on it. A device given a fixed address inside the range the server issues automatically, a duplicate, a machine still pointed at a gateway replaced last term: none of these announces itself.

The damage is disguised: a duplicate address works perfectly until the other holder is switched on, so the fault appears at the wrong desk on the wrong morning and looks exactly like a bad cable. Half a day then goes on layer one for a problem created at layer three.

The cost that compounds: because the plan is undocumented, nobody dares alter it. The school cannot segment the laboratory from the office, cannot add a wireless network cleanly, and cannot admit a visitor's laptop without risk. One unsettled issue freezes every improvement that depends on it.

The judgement and the remedy: addressing, and the remedy costs nothing. Write the plan down, keep a printed copy in the laboratory, reserve one clearly bounded range for fixed addresses outside the range issued automatically, record every fixed address against the device that holds it, and let one named person allocate them.

Return to the notebook and read it again with that in mind. The entries stop being a list of failures and become a single failure, repeated, of a decision nobody remembers making.

The Master Key: *Write the address plan down before the second switch arrives, because every other layer-three trouble can be bought out of and this one cannot.*

Routing algorithms classified

Every routing algorithm answers the same two questions: who decides the path, and when is that decision revisited. The families in this section are the standard answers, and the difficulty is not learning their names but placing a protocol in the right family, since the classification most often copied puts two families in the wrong box. This section settles what separates static from dynamic routing, what a distance-vector router and a link-state router each send and each know, what a hybrid actually is, where the boundary between interior and exterior protocols falls, and which quantities may serve as a metric.

BINDER Question 88

Between the district council office in Korogwe and its store two kilometres away there is one fibre pair. A technician typed four lines into each router and neither has been touched since. Three more offices are joining, and every line must be typed again.

Distinguish static routing from dynamic routing, and state one situation in which each is the better choice.

Model Answer

In static routing every entry is typed in by an administrator and stays as typed until somebody edits it; in dynamic routing the routers run a protocol among themselves and rebuild their tables whenever the network changes. A site with one link and no alternative is best routed statically, a network with several paths and frequent change dynamically.

Where the entry comes from: a static entry has one author and one source, the configuration file. A dynamic entry is deduced from messages the routers exchange, so the table is the protocol's conclusion rather than anybody's instruction.

What happens when a link fails: this is the real difference. A static route goes on pointing at a path that no longer works, and traffic is discarded until a person edits the router. A dynamic protocol hears the silence, withdraws the route and installs the next best.

What each costs to run: static routing uses no bandwidth and almost no memory, and cannot itself misbehave. A dynamic protocol pays for its flexibility with hello messages, update messages and constant recomputation.

When static is right: a site with exactly one way out. A branch office joined to its provider by a single line has no decision to make, so one default route towards that line is the whole answer. Very small stable networks qualify too.

When dynamic is right: any network with alternative paths, more than a handful of routers, or changes arriving faster than an administrator can type. It pays for itself the first time a link fails at night and traffic is going the other way before anyone is telephoned.

Feature	Non-adaptive (static)	Adaptive (dynamic)
Source of the entry	Typed in by an administrator	Learned from a protocol between routers
Response to a failure	None until a person edits the router	Detected and routed around by the protocol
Traffic it generates	None at all	Hello and update messages, continually
Memory and processor	Negligible; the table is fixed	Higher; neighbours tracked and paths recomputed
Suited to	One link out, or a few stable routers	Several paths, many routers, frequent change

Table 3.12: *Adaptive and non-adaptive routing compared.*

Neither is a beginner's version of the other. A static table records what the administrator knows; a dynamic table records what the routers were left to work out.

The Master Key: *Ask how the network finds out that something has broken, because that one question separates the two families more sharply than any list of features.*

BINDER Question 89

Five names are written on the board of a Form Five classroom in Babati: flooding, random walk, RIP, OSPF and EIGRP. The class must sort them into three columns. Most can define all five and still put two in the wrong column.

Using Figure 3.16, classify flooding, random walk, RIP, OSPF and EIGRP, stating in each case the family the algorithm belongs to and what puts it there.

Model Answer

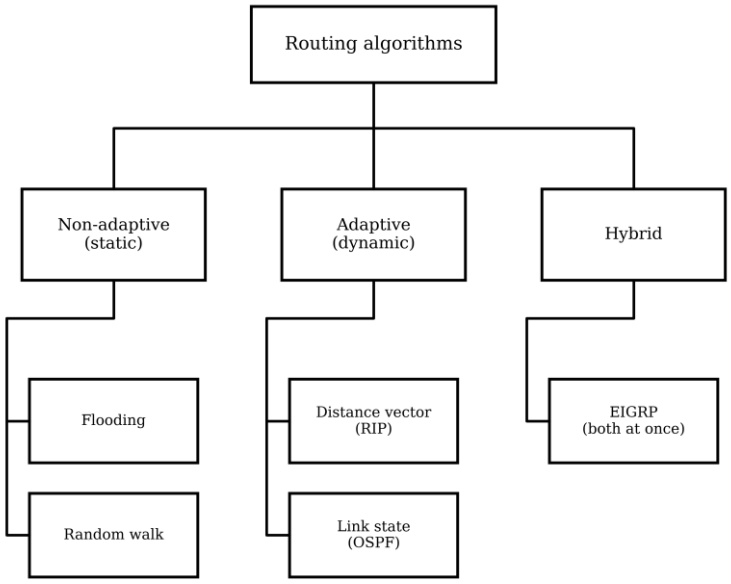
Flooding and random walk are non-adaptive, because neither consults the state of the network; RIP and OSPF are both adaptive, RIP in the distance-vector family and OSPF in the link-state family; and EIGRP alone is the hybrid, because it takes something from each adaptive family.

Flooding: every arriving packet is sent out of every line except the one it came in on. Nothing is measured and no table consulted, which makes it non-adaptive. It wastes bandwidth and reaches every reachable router, and that reliability is why link-state protocols use it to spread advertisements.

Random walk: the packet is forwarded on one line picked at random. Again no state is consulted, so it too is non-adaptive. It survives almost any damage and arrives eventually, but the path it takes has no claim to be a good one.

The two adaptive families: distance vector and link state both react to a failure or a new link with no administrator present, which is what adaptive means. RIP is the distance-vector example and OSPF the link-state one, and they differ in what a router tells its neighbours, not in whether it responds to change.

EIGRP, the one hybrid: it keeps the distance-vector habit of learning distances from neighbours instead of holding a map, and borrows link-state habits: neighbours found by hello messages, updates carrying only what changed, and a backup path held ready.



A hybrid protocol borrows from both families. A family is not itself a hybrid.

Figure 3.16: *Routing algorithms classified. Link state and distance vector are both adaptive families; the hybrid is EIGRP, which borrows from each.*

The test to apply: ask whether the decision can change with nobody touching it. If it cannot, the algorithm is non-adaptive. If it can, ask next what it exchanges, distances or descriptions of links. Only something doing both belongs in the third column.

Two of the five share the middle column, which is the point of the diagram: file the two adaptive families under hybrid and the adaptive column stands empty, with nothing left to call EIGRP.

The Master Key: *Name what a protocol sends before you name its family, since the family is defined by the message and not by the manufacturer.*

BINDER Question 90

Two people learn the streets of Karatu. One asks every neighbour how far each place is and adds the distance to their door. The other is handed every junction and draws the map. Only one can say what happens when a road is closed.

Explain how a distance-vector protocol builds its routing table, and how a link-state protocol builds its own.

Model Answer

A distance-vector router sends its whole table of destinations and distances to its neighbours at intervals, and each entry received is raised by the cost of reaching that neighbour and kept if it beats what was held; a link-state router floods a description of its own links to the whole area, so every router holds one map and computes its own paths.

What a distance-vector router knows: distances and next hops, nothing more. It never sees the network; its table is the sum of what neighbours have said. The method is called routing by rumour, because a router believes and cannot check.

How that table settles: each router begins with the networks it is attached to and learns the rest one exchange at a time, so knowledge spreads outwards at one router per update period, which is the origin of the family's slowness.

What a link-state router floods: a short description of itself, naming the routers it is adjacent to and what each of those links costs. It repeats nothing it has been told, only what it can see, and the description reaches every router unaltered.

One map, computed separately: when the flooding stops, every router holds the same collection of descriptions, and assembled they are a map. Each then works out its own shortest path to every destination and writes the first step into its table.

What the difference buys: distance vector is small, simple and slow to agree. Link state settles in seconds, because a change reaches everyone at once rather than being repeated onwards, but each router pays in memory and processor time.

Protocol	Family	Metric	Convergence	Best suited to
RIP	Distance vector, adaptive	Hop count, 15 usable	Slow, minutes on a deep network	A few routers, one administrator
OSPF	Link state, adaptive	Cost derived from bandwidth	Fast, seconds	Medium and large networks of any make
EIGRP	Hybrid	Bandwidth and delay combined	Fast, seconds	Networks built from one maker's routers

Table 3.13: *RIP, OSPF and EIGRP compared.*

Both families are distributed, in that every router takes part; the older alternatives, a router deciding in isolation and a central machine computing for all, survive only as names.

The Master Key: *One family exchanges conclusions and the other exchanges evidence, and every other difference between them is a consequence of that choice.*

BINDER Question 91

One link out of Tunduma runs 40 km of fibre to the nearest exchange. The other reaches a neighbour's building 300 m away, and on a map it wins outright. It is also 2 Mbit/s of tired copper shared with three other tenants.

List five metrics a routing protocol may use, and explain why the physical distance between two routers is not one of them.

Model Answer

Hop count, bandwidth, delay, load, reliability and cost are the quantities routing protocols use, singly or combined into one composite figure; physical distance is not among them, because a router cannot measure it and because it predicts almost nothing about how a packet will fare.

Hop count: the number of routers a packet must pass through. It treats every link as equal, so a path over two fast links loses to a path over one slow one. RIP uses it and nothing else.

Bandwidth and delay: bandwidth is the capacity of the slowest link on the path, usually turned into a cost so that a larger link yields a smaller number. Delay is how long a packet takes to cross, measured or taken from a table of typical values.

Load and reliability: load is how busy an interface is at present; reliability is the proportion of frames arriving undamaged, taken from the error counters. Both move minute by minute, so protocols damp them: a metric that follows the traffic exactly sets routes oscillating.

Cost: a number attached to an interface by hand, standing for whatever the organisation wishes to discourage: a metered circuit, or a path across somebody else's network. A metric because the router can read it, a policy because a person chose it.

Why kilometres fail: nothing at an interface reports where the far end of the cable is, and a router holds no map of the ground. What distance contributes is propagation delay: 40 km of fibre adds about 0.2 ms, against perhaps 30 ms from one congested queue, 150 times as much. Delay carries that already and can be measured.

Behind the list is one rule. A metric must be something the router can obtain for itself and something the packet will feel. Kilometres of cable fail both, and the first absolutely.

The Master Key: *A quantity earns the name of metric only where the router can measure it and the traffic can suffer from it.*

REAL Question 92

The best-kept notebook in a study group in Musoma has a page on routing families, copied in a firm hand and borrowed by everybody. Two pages on, the same notebook calls OSPF adaptive and EIGRP the hybrid. In three terms nobody has noticed.

Identify what is wrong with each of the three statements, give the correct position with the reason in each case, and state the habit that would have prevented all three.

1. Link state and distance vector are the two hybrid routing algorithms, which is why the diagram puts them both in the hybrid box.

2. The shortest route between two networks is the one with the fewest kilometres of cable, so a routing protocol works out the physical distance and takes the smallest.

3. A static route never needs changing, and that is what makes it static.

Model Answer

All three are wrong: link state and distance vector are the two adaptive families and EIGRP is the hybrid; routing metrics are quantities a router can measure, among them hop count, bandwidth, delay, load, reliability and cost, and never kilometres of cable; and a static route needs changing every time the network it describes changes, which is exactly its weakness.

Statement one, two families made one: both react to a failure with nobody present, which is what adaptive means. A hybrid is a protocol that borrows from two families, and EIGRP is the example: distances learned from neighbours, plus hello messages and updates carrying only what changed. File the two families under hybrid and the adaptive column stands empty.

The notebook against itself: two pages later the same hand has it right. Both cannot stand, and the diagram is not the senior partner here: a box drawn round two words does not make the classification inside it true. When two of your own notes disagree, the picture does not win because it is a picture. The definition decides.

Statement two, the kilometres: an interface reports its speed, its error counters and how busy it is. It reports nothing about where the far end of the fibre lies. A 300 m copper link at 2 Mbit/s is worse, in every respect a packet can feel, than a 40 km fibre at 1 Gbit/s, and delay, which is a proper metric, already carries whatever the distance contributes.

Statement three, the word static: static describes where the entry came from, not how long it stays true. The route outlives the path it names, and goes on feeding traffic into a link that has failed, so a statically routed network needs an administrator within reach of a telephone.

The habit that prevents all three: test a classification by asking what it predicts. If link state and distance vector were hybrids, what would the adaptive column hold, and what would EIGRP be? If distance were a metric, what instrument would report it? A statement that survives those questions is worth copying.

A diagram is the easiest thing in a set of notes to copy and the hardest to check, and this one has been copied for years. A figure makes a claim like any sentence, and it can be wrong while the prose two pages away from it is right.

The Master Key: *When a picture and a sentence contradict each other, go back to the definition and let it decide, because being drawn in a box is not evidence.*

REAL Question 93

A university campus on Unguja runs RIP across six routers. The link to the library block was moved one Saturday. Machines by the gate followed at once; those at the far end kept the old path for minutes, and a technician logged a fault.

Explain the mechanism that produces this delay, and recommend a protocol that would remove it, stating what the change would cost.

Model Answer

Nothing is faulty: RIP speaks only to its direct neighbours and only every 30 seconds, so news of a change moves outwards at about one router per update period and reaches the far end minutes later; OSPF would remove the delay, because a link-state router floods the change to the whole area at once.

Neighbours only, and on a timer: a RIP router sends its whole table to the routers directly attached to it, every 30 seconds. A router four hops away never hears from the source of the change. It hears its own neighbour's opinion, one period after that neighbour formed it.

One router per period: each further router adds another period, so on a campus six routers deep the news needs about six of them, roughly 180 seconds, to reach the halls. That figure belongs to the family, not to the equipment, and faster links will not move it.

Bad news is slower still: a failed link produces nothing to send. A router waits through six missed updates, 180 seconds, before treating that neighbour's routes as unusable, and only then does the withdrawal begin its own journey, giving a worst case of about 360 seconds, six minutes.

The safeguards add delay too: split horizon and hold-down exist to stop routers feeding stale information back to one another and counting upwards together, and they work by declining certain news for a time. Stability here is bought with delay, which is why 16 means unreachable and 15 hops is the longest usable path.

What OSPF does instead: the router whose link changed floods one advertisement to the whole area within a second, and every router recomputes its own paths from the corrected map. Convergence becomes a matter of seconds and stops depending on how deep the campus is.

What the change costs: each router must hold the database for its area and recompute on every change, so memory and processor use rise, and the campus needs an address plan and areas chosen deliberately. Against six minutes of misdirected traffic, on a campus growing towards the 15-hop ceiling, that is a small price.

The complaint was written up as a fault and was in fact a specification. RIP undertakes that the news will arrive, not that it will arrive soon, and no configuration makes a router tell a neighbour what it has not yet been told itself.

The Master Key: *When a network is slow to agree with itself, examine what each router is permitted to say and to whom, not the speed of the links between them.*

REAL Question 94

A district council in Mtwara buys its connection from a provider whose network reaches four regions. Inside the council's own buildings the routers settle among themselves which path is shortest. Where their fibre meets the provider's equipment that conversation stops and another begins.

Distinguish an interior gateway protocol from an exterior gateway protocol, using this provider and this customer as the example, and name the protocol that runs between them.

Model Answer

An interior gateway protocol runs between routers of one administration, an autonomous system, and chooses the path with the best metric; an exterior gateway protocol runs between autonomous systems and chooses on policy rather than on any shared metric. RIP, OSPF or EIGRP would run inside the council and, separately, inside the provider; BGP runs between the two.

What an autonomous system is: a set of networks under one administration following one routing policy, identified by a number allocated for the purpose. The council is one such system and the provider is another, and everything inside a boundary of that kind trusts everything else inside it.

Interior, one owner and one metric: inside the council every router is configured from the same office, so a neighbour may be believed without question and the only matter at issue is which path is shortest. OSPF, RIP, EIGRP and IS-IS all answer that question and differ only in how.

Exterior, two owners and no shared metric: the provider will not accept the council's idea of cost, and the council cannot see inside the provider. What crosses the boundary is reachability: which blocks of addresses can be reached, and through which sequence of autonomous systems. BGP carries exactly that.

Shortest is the wrong question: between organisations the shortest path may cross a competitor, a metered circuit or a party with whom nothing has been agreed. A protocol that always took the smallest number would send traffic through networks nobody had paid for, which is why BGP decides by rules about who may be preferred and what may be advertised.

What each side actually configures: a customer with one connection needs no BGP whatever: a default route towards the provider is the complete answer, and the provider advertises the customer's addresses for it. BGP becomes necessary when the council takes a second provider and must govern which is used for what.

The scale each is built for: an interior protocol holds every link inside one organisation and expects them to be few and known. BGP holds routes towards the whole Internet, expects them to be many, to be advertised by strangers and to be subject to policy at every step.

The boundary is administrative before it is technical. Were the two organisations to merge tomorrow, one interior protocol would run over the same routers and the same fibre that need BGP between them today.

The Master Key: *Inside a boundary the question is which path is shortest, and across a boundary the question is whose path it is.*

HOT Question 95

Four blocks, four routers and a ring of fibre at a secondary school in Iringa: the contractor found the trenches open. The technician configured every route by hand and left a two-page list. The head of ICT wonders whether a protocol should replace it.

Assess whether the school should run OSPF or keep its static routes, and commit to a recommendation.

1. Four routers, one in each block, each with its own user network.

2. The four are joined in a ring, so every pair of blocks has two paths between them.

3. The router in block A also carries the school's single link to the provider.

4. Nine networks in all: four user networks, four links round the ring, and the provider link.

Model Answer

Run OSPF. The ring gives every pair of blocks a second path, and a hand-written configuration cannot use that path without somebody editing routers during the failure, so the school has bought redundancy it is unable to spend; the one static entry worth keeping is the default route towards the provider.

What the list costs already: each router has to be told about every network it does not itself touch. Block A touches four of the nine and the other three touch three apiece, which comes to 23 statements, and every one of them is revisited when a network is added anywhere in the school.

Redundancy doubles it: to use the ring at all, each destination needs a second route the other way round, held in reserve behind the first, and something that notices when the first has failed. That is 46 statements and a tracking mechanism, and it is at this point that static routing stops being the simpler option.

What OSPF costs to configure: one routing process on each router and one line naming its interfaces in the area. That is less typing than the list it replaces, it does not grow when the fifth network appears, and it can be read by the next technician in a form that does not depend on his predecessor's handwriting.

The honest case for static: typed routes cannot flap, consume no bandwidth, cannot be corrupted by a misconfigured neighbour and are plain to whoever inherits them. On a school network wired as a chain, with no second path anywhere, they would be the right answer. The ring is what settles this one.

The failure that decides it: a trench is reopened during examinations and one side of the ring is cut. Under OSPF every router has recomputed within seconds and traffic goes round the other way. Under the list, the school is halved until somebody with the password arrives, and the second path lies idle throughout.

The judgement: the recommendation is not close. Run OSPF on all four routers in a single area, and keep one static default route on the block A router pointing at the provider, advertised into OSPF so the other three learn it. The two methods are not rivals: use the protocol where the network contains a choice, and a typed entry where it does not.

The question was never which method is the more advanced. It is whether the network contains a decision, and a ring contains one from the moment a fibre is cut.

The Master Key: *Choose a routing method by counting the paths and not the routers, since a protocol earns its keep only where an alternative exists.*

HOT Question 96

An internet provider serving Singida loses a link most weeks, more often in the rains. Its oldest routers are short of memory. One engineer wants the light protocol, one the fast one, and both argue from the same fact. The manager must choose.

Judge which matters more on a network that changes often, low memory use or fast convergence, and defend the judgement against the strongest objection to it.

Model Answer

Fast convergence matters more: memory is a fixed cost, paid once and purchasable, while convergence time is an outage paid again on every change, so on a network that changes weekly the arithmetic favours convergence heavily, and where memory genuinely will not stretch the answer is to divide the network into areas rather than to accept a slow protocol.

The two costs differ in shape: memory is a single payment, known in advance and bounded by the size of the network. Convergence time is a cost per event, and the number of events is set by the weather, the contractors and the age of the plant, none of which the design controls.

The recurring cost in figures: two changes a week at six minutes apiece is 624 minutes in a year, over ten hours in which traffic is discarded or circling. The memory saved is the same few megabytes on the last day of that year as on the first.

What convergence time really is: not slowness but disagreement. While the routers hold different views, packets go to a next hop that has no path, or pass back and forth between two routers each believing the other can reach the destination. Those parts of the network are not degraded; they are down.

The objection at its strongest: a router out of memory does not converge slowly, it fails to converge at all. It drops adjacencies, thrashes and may restart, and a restart costs more than any protocol timer. On genuinely small routers, preferring the light protocol is arithmetic rather than timidity.

Why the objection loses: it argues against an oversized database, not in favour of slow convergence. A link-state protocol divided into areas gives each router full detail of its own area and a summary of the rest, so the database is bounded by the area and not by the network. Hierarchy escapes the trade-off rather than taking a side in it.

The judgement: choose fast convergence. Where the memory will not stretch, buy memory, which is the cheapest thing in the rack, or cut the network into areas, and replace the routers that can do neither. Accept a slow protocol only where nothing changes, and this network changes most weeks of the year.

A trade-off offered as a choice between two evils is usually an invitation to find the third option. Between a protocol that cannot remember and a protocol that thinks too slowly, the engineering answer is to give it a smaller thing to think about.

The Master Key: *Cost a design by how often you pay rather than how much, because a small penalty on every failure will outrun a large one paid once.*

Routing computed: Dijkstra, Bellman-Ford and the routing table

Naming a routing algorithm is one thing; running one is another. This section does the arithmetic. A network is drawn as a weighted graph, Dijkstra is worked line by line until the least-cost route falls out of it, the same answer is reached again from the opposite end by Bellman-Ford, and a routing table is read column by column until you can say which line a given packet will match and why. It assumes you can already name the families of routing algorithm and say what a router does with a packet.

BINDER Question 97

A fibre contractor in Korogwe prices two ways to the same district office. One is eleven kilometres of buried cable in a single span. The other threads through three cabinets and covers thirty kilometres. The routing protocol will never be told a single kilometre.

Explain what a routing metric is, and show why the least-cost path between two routers need not be the path with the fewest hops.

Model Answer

A routing metric is a number a protocol attaches to each link so that a route's links can be added and whole routes compared as sums; because the numbers differ from link to link, the route crossing fewest links can carry the largest sum, and the protocol rejects it.

A number chosen to be added: the metric is not a measurement of the world but a figure assigned to a link so that arithmetic can be done on it. A route's cost is the sum of its links' costs, and the protocol keeps whichever route gives the smallest sum. Everything else about the link is discarded.

Where the number comes from: most often from bandwidth. A common rule divides a fixed reference speed by the link speed, so against a 100 Mbit/s reference a 2 Mbit/s line costs 100 / 2 = 50 and a 100 Mbit/s line costs 100 / 100 = 1. Delay, load and reliability serve equally well.

Two routes costed: a two-hop route over a pair of 2 Mbit/s lines costs 50 + 50 = 100. A three-hop route over three 100 Mbit/s lines costs 1 + 1 + 1 = 3. The longer route in hops is cheaper by 100 − 3 = 97, and a protocol adding costs takes it.

What hop count really is: counting hops is not the absence of a metric. It is a metric that has quietly set every link to 1, so the sum becomes the number of links. Where all the lines run at one speed that assumption holds; the moment one of them is slower, the fewest-hop route becomes the slowest.

The customer at Korogwe is measuring the ground while the protocol is costing the line, and since the numbers written on a network drawing are costs rather than kilometres, the longest line on the page may carry the smallest of them.

The Master Key: *Add the costs, never count the links, because a hop is only a cost of one wearing a disguise.*

BINDER Question 98

Two engineers at a provider's office in Makambako take the cheapest route from A to E off the whiteboard by eye, in under a minute, and hand in different answers. Neither has noticed that the true route turns away from E partway along.

Apply Dijkstra's algorithm to the network in Figure 3.17 to find the least-cost route from A to E, giving the tentative distance to every node at each iteration.

Model Answer

The least-cost route is A to C to B to D to E at 2 + 1 + 5 + 2 = 10, and Dijkstra reaches it in five iterations, each one settling the nearest unsettled node and then offering that node's neighbours a cheaper way in.

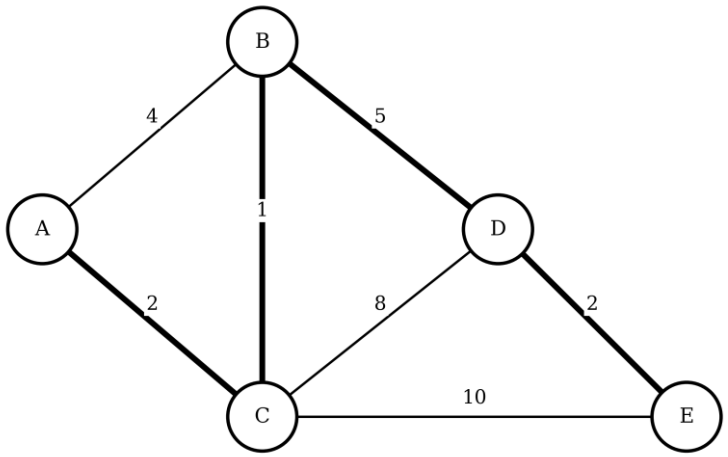
Iteration 1, settle A: set A to 0 and every other node to infinity, settle A, and offer its neighbours a route through it. B becomes 4 and C becomes 2. The table reads A 0 settled, B 4 via A, C 2 via A, D infinity, E infinity.

Iteration 2, settle C: the smallest unsettled figure is C at 2, so C is settled and can never improve again. Through C, B is offered 2 + 1 = 3, better than 4, D is offered 2 + 8 = 10 and E is offered 2 + 10 = 12. The table reads A 0 settled, C 2 settled, B 3 via C, D 10 via C, E 12 via C.

Iteration 3, settle B: B at 3 is now the smallest unsettled figure, so B is settled. Through B, D is offered 3 + 5 = 8, beating the 10 it held through C, and nothing else improves. The table reads A 0 settled, C 2 settled, B 3 settled, D 8 via B, E 12 via C.

Iteration 4, settle D: D at 8 is the smallest unsettled figure, so D is settled. Through D, E is offered 8 + 2 = 10, beating the 12 it held through C. The table reads A 0 settled, C 2 settled, B 3 settled, D 8 settled, E 10 via D.

Iteration 5, settle E: E at 10 is the only node left, so it is settled and the work stops. Read the via column backwards: E came from D, D from B, B from C, C from A. Turned round, that is A to C to B to D to E, costing 10.



Shortest path A to E is A - C - B - D - E at a total cost of 10. The direct-looking A-C-E costs 12 and A-B-D-E costs 11. Taking the direct link from C to E looks shorter and costs two more.

Figure 3.17: *A weighted network. The heavy path is the least-cost route from A to E, at a cost of 10. The direct-looking route through C to E costs 12.*

The route takes the cheap line to C and then turns away from E, back up to B, which is the move nobody makes by eye: A to C to E costs 12 and A to B to D to E costs 11, so both obvious answers are dearer.

The Master Key: *Settle the nearest unsettled node, offer its neighbours a cheaper way in, and repeat until the destination itself is settled.*

BINDER Question 99

The screen of the router at a district office in Mtwara shows five lines and will never show more. Three packets arrive in the same second for three different networks. One of those five lines serves each, and the printed order decides nothing.

State which entry of Table 3.14 the router uses for a packet addressed to 10.1.1.9, one addressed to 192.168.10.40 and one addressed to 10.2.7.4, and explain the rule that decides.

Model Answer

The packet for 10.1.1.9 leaves by Se0/1 to next hop 10.0.0.6 under the 10.1.1.0/24 entry, the packet for 192.168.10.40 is delivered straight out of Fa0/0 because that network is directly connected, and the packet for 10.2.7.4 matches nothing specific and takes the default route; the rule throughout is longest-prefix match.

One line as one sentence: for any address falling inside this destination under this mask, send the packet to this next hop, out of this interface, at this cost. The metric is used only to choose between two entries describing the very same network, never between entries describing different ones.

Two entries that both match: 10.1.1.9 lies inside 10.1.0.0 under 255.255.0.0, and it also lies inside 10.1.1.0 under 255.255.255.0. Both lines are true at once. The router takes neither the first nor the cheaper, but the more specific, and 24 mask bits beat 16.

Why the specific wins: a longer prefix names a smaller set of addresses, so whoever wrote it knew more about where that network actually is. The broad line still serves the rest of its range: a /16 spans 2 to the power 16, that is 65,536 addresses, and the /24 claims 2 to the power 8, that is 256, leaving 65,280 to the broad entry.

A directly connected network: 192.168.10.40 matches 192.168.10.0 under 255.255.255.0, and that line names no next hop because no next router exists. The host itself is at the far end of Fa0/0, so the packet is framed for the host and the metric is 0.

Nothing specific matches: masking 10.2.7.4 by 255.255.0.0 gives 10.2.0.0, which is not 10.1.0.0, and no other line covers it either. Only the default entry matches, so the packet goes to 203.0.113.1 out of Se0/2, and the router receiving it becomes responsible for knowing better.

Destination	Mask	Next hop	Interface	Metric
192.168.10.0	255.255.255.0	Directly connected	Fa0/0	0
10.1.0.0	255.255.0.0	10.0.0.2	Se0/0	2
10.1.1.0	255.255.255.0	10.0.0.6	Se0/1	3
172.16.5.0	255.255.255.0	10.0.0.2	Se0/0	4
0.0.0.0	0.0.0.0	203.0.113.1	Se0/2	1

Table 3.14: *A routing table, read line by line.*

Position on the screen decides nothing. The router tests every line, keeps every line that matches, and hands the packet to whichever of them carries the longest mask.

The Master Key: *Count the mask bits before you read the metric, because the longest prefix has already won before cost is consulted.*

BINDER Question 100

One router, one link to the world and no intention of a second: that is the network of a secondary school in Babati. A visiting technician proposes an entry for every network the school might wish to reach. A single line replaces the list.

Define the default route, describe how it is written, and state the circumstances under which a router uses it.

Model Answer

The default route is the entry 0.0.0.0 with a mask of 0.0.0.0, written 0.0.0.0/0, which matches every address in existence; its prefix is zero bits long, the shortest possible, so longest-prefix match reaches it only when nothing else in the table matches.

A mask of no bits: the mask 0.0.0.0 tests nothing at all. Every address masked by it yields 0.0.0.0, precisely the destination written on the line, so the comparison always succeeds. The entry matches everything and matches nothing specifically.

Why it is reached last: the router keeps every matching entry, then prefers the one with most mask bits set. The default has none, so any other match outranks it. It is called the gateway of last resort, and last is exactly when a router reaches it.

What it is for: a network with one way out need not know where anything is; it needs to know which door to use. One line sends everything unrecognised to the provider's router, which does hold the detail, and the school's table stays short enough to read at a glance.

When not to have one: a router at a junction with several exits is better without one, because a default makes it impossible to report that a destination does not exist. Every packet leaves somewhere, so a mistyped address is carried away and quietly discarded rather than refused on the spot.

The failure it hides: if a network inside the organisation loses its own entry, that traffic falls through to the default, travels to the provider and is dropped there. The symptom is one network unreachable while the router insists, correctly and uselessly, that it has a route to everything.

One line stands in for every network the school will never need to name, and it is read only after every other line has been offered the packet and declined it.

The Master Key: *A mask of zero bits matches all and outranks none, which is why the default route is the last line a router ever reaches.*

REAL Question 101

The five routers of a regional health network around Geita carry the seven links and costs of the drawing, which nobody has ever assembled. Each knows its own links and whatever its neighbours last said. There is no map here, and they still agree.

Apply Bellman-Ford to the network of Figure 3.17 to find the least-cost route from A to E, and identify one situation in which it and Dijkstra do not behave alike.

Model Answer

Working towards E rather than away from A, every node repeatedly takes the least value of a link's cost plus that neighbour's claimed cost to E; after four rounds the estimates settle at A 10, B 7, C 8, D 2 and E 0, so A reaches E for 10 by way of C, the route Dijkstra produced.

The rule every node applies: a node's cost to E is the smallest value of the link to a neighbour plus that neighbour's cost to E. E begins at 0 and everything else at infinity. No node sees more of the network than the neighbours it is joined to, and none is ever sent a map.

Round 1: only E has a real figure to give away. D, joined to E by 2, takes 2 + 0 = 2. C, joined to E by 10, takes 10 + 0 = 10. A and B have no neighbour with a figure and stay at infinity. The estimates read A infinity, B infinity, C 10, D 2, E 0.

Round 2: B now hears D at 2 and takes 5 + 2 = 7, which beats the 1 + 10 = 11 it could have had through C. A hears C at 10 and takes 2 + 10 = 12, its first real figure. The estimates read A 12, B 7, C 10, D 2, E 0.

Round 3: A hears B at 7 and improves to 4 + 7 = 11, beating its own 12. C hears B at 7 and improves to 1 + 7 = 8, beating the 10 it had direct. The estimates read A 11, B 7, C 8, D 2, E 0, the improvement spreading outward from E one link per round.

Round 4 and the answer: A hears C at 8 and improves once more to 2 + 8 = 10, beating the 11 it held through B. A fifth round moves nothing, so the work is finished. A hands to C, C to B, B to D, D to E, and the total is 10.

Same answer, different knowledge: Dijkstra needed the whole graph in one place and touched each node once. Bellman-Ford needed no graph, only repeated conversations, and touched every node in every round. One is a calculation on a drawing, the other a negotiation that eventually stops moving.

Where the two part company: when a link fails. Dijkstra is rerun on a corrected drawing and is right at once. Bellman-Ford has no drawing, so a node can be quoted a figure that is only an echo of its own from a round earlier, and the estimates then climb by ones instead of falling.

Both algorithms minimise the same sums over the same seven links; what separates them is who may see those links, which also decides how each behaves when one disappears.

The Master Key: *Ask every neighbour and repeat until nobody changes their answer, which is how a router with no map still finds the route and also how it can be misled.*

REAL Question 102

Every district office on one network was configured from the same five-line template, so the router at Sumbawanga holds the entries of Table 3.14. Staff reach the server room, the provider and the regional office, but nothing on 10.1.1.0 answers. No interface reports a fault.

Determine why a router holding a complete table cannot reach 10.1.1.0/24, and state what should be done about it.

Model Answer

The 10.1.1.0/24 entry points at next hop 10.0.0.6 out of Se0/1, and because longest-prefix match always prefers 24 mask bits to 16 that entry is chosen every time; if 10.0.0.6 is no longer a working path the packets die there, and the router never falls back to the 10.1.0.0/16 entry which would have carried them.

The symptom names the entry: one network unreachable while its neighbours are healthy is not an interface fault, because a dead interface takes down every network behind it at once. Look instead for an entry specific to the failing network alone, and this table holds exactly one such line.

Why the broad entry cannot rescue it: an address such as 10.1.1.9 matches the 10.1.1.0 line and the 10.1.0.0 line together. Longest-prefix match is not a preference that a metric can overturn; the more specific entry is taken whenever it is present, and the router never asks whether the path it names still works.

A written route cannot die: an entry put in by hand stays until it is taken out by hand. It carries no keepalive and puts no question to 10.0.0.6, so the router goes on believing it has a route long after the neighbour that made the route true was switched off, renumbered or moved to another interface.

How to prove it in five minutes: send a packet to 10.0.0.6 itself. If the next hop is silent while the rest of that side answers, the fault lies in front of the next hop rather than behind the destination. Then reach 10.1.1.9 from a router holding only the /16 entry; if that succeeds, the specific line is the entire problem.

What to do about it: remove the 10.1.1.0 line and let the traffic fall through to the /16 route, which reaches the same network at a metric of 2 rather than 3. If the specific route is genuinely wanted, correct its next hop and tie it to something that watches whether that next hop is alive, so that it withdraws itself when the path fails.

Nothing on that router was broken. The table did exactly what a table is built to do, and the most specific answer silenced the only working one.

The Master Key: *Suspect the narrowest entry first when a single network disappears, because a specific route outranks a general one even after it has stopped being true.*

HOT Question 103

Two routers share a rack at an exchange in Stone Town. One holds a description of every link and recomputes whenever anything changes. The other holds nothing but a column of totals its neighbours quoted. Both forward packets correctly, most of the time.

Assess why Dijkstra can be run inside a link-state router but not inside a distance-vector one, and judge which of the two designs is the sounder basis for a network that has to be trusted.

Model Answer

Dijkstra must have the whole graph in one memory before it can take a first step, and only a router sent everyone else's links has one; a router holding quoted totals has no graph to run it on and can only take part in a distributed Bellman-Ford, so the design that holds the graph is the one to build on.

What the algorithm demands: at every step Dijkstra selects the nearest node not yet settled, anywhere in the network. That selection cannot be made from local knowledge; it needs every node and edge in view at once. The algorithm cannot be distributed, and that one fact decides which family can run it.

How the graph arrives: each router describes only the links it can see for itself, and that description passes on unaltered until every router in the area holds it. Each assembles the same graph from first-hand statements and runs Dijkstra on it with itself as root, which is why all of them agree.

What the other family holds: a column of totals records destinations and costs but nothing about which links produced them. No graph can be recovered from totals, so no global nearest node can be chosen, and the only recourse is to apply the local rule until the figures stop moving.

What each design costs to run: the graph costs memory and a burst of processing on every change, and the flooding that maintains it consumes bandwidth just when the network is under strain. Totals cost almost nothing to keep and nothing to compute, which is why the cheaper design was sensible when a router had a megabyte of memory.

What each design costs when wrong: a router holding the graph can see that a claim contradicts the map and can rebuild the whole answer from first-hand descriptions in one pass. A router holding totals cannot tell a fresh figure from an echo of its own, so its errors take the shape of loops lasting minutes.

The judgement: build on the design that holds the graph. Memory has stopped being scarce, the arithmetic is trivial at any size a school or district will build, and the expense buys correctness within a second of a change rather than eventually. Keep the cheaper family only where there is nothing to compute, and there a few written routes are simpler still.

The real difference is not which algorithm runs. It is whether a router is allowed to see the network or must take somebody's word for it, and every other difference between the two follows from that.

The Master Key: *An algorithm that has to choose the nearest node in the whole network can only run where the whole network is known.*

HOT Question 104

Three routers stand in a line above Mpanda, and the network behind the first fails. Within a second its neighbour knows. Within six minutes all three agree on a meaningless number that rose by one every half minute, while packets shuttle between them.

Explain how count-to-infinity arises on a three-router network, and judge whether split horizon is a sufficient remedy for it.

Model Answer

Count-to-infinity occurs because a router accepts a total from a neighbour without being told the route behind it, so it can be quoted its own withdrawn route back and believe it; split horizon, which forbids advertising a route back out of the interface it was learned on, cures the case on a line but fails once three routers form a loop, so it is necessary and not sufficient.

The setting: routers X, Y and Z in a line, with network N attached to X. Y records N at 1 and Z records it at 2, learned from Y. Each sends its whole list to its neighbours every 30 seconds and accepts any figure improving on what it holds.

How the false route forms: the link from X to N fails and Y marks N unreachable. Before Y can announce it, Z's update arrives quoting N at 2. Z does not say its route runs through Y, and Y cannot ask, so Y accepts 2 + 1 = 3 and now believes Z can reach a network only Y could ever reach.

The climb: Y advertises 3, Z raises itself to 4, Y raises to 5, and the pair walk the figure upwards a step at a time. It halts only at the value agreed to mean unreachable, commonly 16. Climbing from 3 to 16 takes 16 − 3 = 13 exchanges, and at one update every 30 seconds that is 13 × 30 = 390 seconds, six and a half minutes.

What split horizon does: the rule is that a router never advertises a route back out of the interface on which it learned it. Z learned N from Y, so Z stops quoting N to Y. Y never hears the stale figure, marks N unreachable and says so. On this line of three the fault is cured completely.

Where it stops working: join the same three into a triangle, so each reaches the other two. Y and Z now learn N from X directly. When N fails and Y hears first, Z may still quote N to Y, since Z learned it from X and not from Y. The rule is unbroken and the count begins anyway.

The judgement: keep split horizon and never rely on it. Add poison reverse, so a withdrawal is stated rather than merely omitted, triggered updates, so bad news travels at once instead of waiting for a timer, and a hold-down, so a route just withdrawn is not believed again. Together they narrow the window; none shuts it.

Split horizon removes the loop with two routers in it and leaves every larger one standing: a fine return on one rule, and a poor foundation for any network with redundancy in it.

The Master Key: *A cure that works only while the loop has two routers in it is not a cure for loops.*

HOT Question 105

A contractor at Handeni cuts a duct while widening a road, and the link of cost 1 between two of the five routers goes dark. Nothing at A or E is broken, yet the route between them is wrong, and A has not noticed.

Determine the effect on the least-cost route from A to E in Figure 3.17 when the link between C and B fails, and judge how quickly the network can be expected to notice.

Model Answer

Without the C to B link the least-cost route becomes A to B to D to E at 4 + 5 + 2 = 11, one dearer than the 10 before, and A's first hop moves from C to B although the broken link is nowhere near A; a design that floods topology notices within a second where the interface drops, or within 40 seconds where it must wait for a timer.

Recost every candidate: delete the link of cost 1 and three routes from A to E remain worth costing. A to B to D to E is 4 + 5 + 2 = 11. A to C to D to E is 2 + 8 + 2 = 12. A to C to E is 2 + 10 = 12. The cheapest is now 11, so one broken link has cost the network 11 − 10 = 1.

The first hop moves: before the cut A sent towards E by way of C. Afterwards it must send by way of B, over a link costing twice as much. A cannot work this out for itself, because both of its own links are perfectly healthy and neither has changed in any way it could detect.

What has to happen next: the two routers at the ends of the broken link must detect the loss, and every router computing a route across it has to be told. Detection is either electrical, when the interface reports loss of carrier, or by timer, when a greeting that should have arrived does not.

How long detection takes: on a direct copper or fibre span the interface knows at once and the failure is reported within milliseconds. Where the loss is not electrically visible, a router waits out a dead interval, commonly four missed greetings at 10 seconds each, so $4 \times 10 = 40$ seconds pass before the link is declared down.

How long the rest takes: once the failure is described, flooding puts that description in front of every router within tens of milliseconds on a network this size, and each recomputes with Dijkstra in less time again. The recalculation is never the slow part. Detection is.

The judgement: budget under a second where the interface can see the fault and about 40 seconds where it cannot, and route the ducts so the links that matter are the electrically visible ones. Where routers exchange totals on a 30-second timer instead, the same cut leaves A using a route that no longer exists for minutes, and that is what to engineer against.

One severed duct at Handeni raises the cost of the journey by 1, and the cheapness of that answer is why the real cost goes unnoticed: it is measured in the seconds A spends confidently sending packets along a route that has ceased to exist.

The Master Key: *Ask how long the network stays wrong rather than how much dearer the new route is, because the delay is where the damage lives.*

ICMP, congestion control and quality of service

A network that is behaving perfectly still needs a way of saying that something has gone wrong, and a network carrying more than it can hold needs a way of deciding whose traffic waits. This section settles both. It covers the Internet Control Message Protocol and the five message types worth memorising, how ping and traceroute are built from them, what congestion is and why loss makes it worse, and the two shaping algorithms most often set as a calculation. It assumes you can already describe an IP datagram and the work of a router.

BINDER Question 106

A clerk at a district office in Ilala sends the same upload four times and watches it vanish four times. The router that discarded it has no field in an IP header to say so. Something explained the loss, and it was not IP.

Explain what the Internet Control Message Protocol is for, and state why it is never used to carry user data.

Model Answer

ICMP is the error reporting and diagnostic protocol of the network layer: it travels inside IP datagrams as protocol number 1, so that a router or host can tell a sender that a datagram could not be delivered, or answer a deliberate test; it carries no user data because it has no ports, no connection and no retransmission.

Carried inside IP, not beside it: an ICMP message sits in the data field of an ordinary IP datagram, marked with protocol number 1, and is routed like any other packet. It receives the same best effort treatment as the traffic it complains about, so a report of failure can itself fail to arrive.

What a report contains: a type, a code narrowing the type, a checksum, then the header of the datagram that provoked it and the first 8 bytes that followed. Those bytes let the sender match the complaint to the conversation that caused it.

Two kinds of message: errors are unsolicited, generated by a router at the moment it discards. Diagnostics are solicited: a host sends a request purely to see whether an answer returns.

Why no user data travels this way: an ICMP header holds no port numbers, so there is nothing to deliver a payload to, and there is no connection, no acknowledgement and no retransmission, so a lost message stays lost.

The restraint built into it: ICMP reports no error about an ICMP error, about a broadcast, or about any fragment but the first. One fault would otherwise provoke reports about reports, and the protocol meant to diagnose trouble would manufacture it.

Treat ICMP as the voice of the network rather than a service offered to users: it says what became of somebody else's packet, and that is all of its work.

The Master Key: *Ask of any ICMP message which packet it is about, because ICMP never speaks on its own behalf.*

BINDER Question 107

A capture running on a borrowed laptop at a technical college in Kinondoni fills with rows carrying no English at all: 8, then 0, then 3, then 11, then 3 again. The student can see which machine complained and when, but never what about.

Name four ICMP message types, give the type number of each, and state the circumstance that produces it.

Model Answer

Echo request, type 8, is sent by a host testing whether another host can be reached; echo reply, type 0, is that host's answer; destination unreachable, type 3, is sent by a router or host that cannot deliver the datagram; time exceeded, type 11, is sent where the time to live has reached zero; and redirect, type 5, is advice from a router that a different gateway serves that destination better.

Type and code together: the type names the family and the code narrows it. A router with no route, a host that is switched off and a port with nothing listening all produce type 3, and are told apart only by the code, so quoting the type alone answers half the question.

The pair that ping uses: an echo request, type 8, asks nothing of the receiver except a copy of itself. The echo reply, type 0, returns the same payload, so the sender can time the round trip and count what failed to come back. Only the two together prove a path in both directions.

The two complaints: destination unreachable, type 3, means delivery was refused and names the reason in its code. Time exceeded, type 11, means the datagram lived too long: either its time to live reached zero in transit, or the fragments of one datagram never all arrived.

The advice message: a redirect, type 5, is sent when a host is using one router for a destination that another router on the same network reaches better. Many networks discard redirects, because a forged one moves a victim's traffic wherever the forger pleases.

Message	Type	Sent when	What it tells you
Echo request	8	A host or a tool tests whether an address answers	Somebody is asking this host to reply
Echo reply	0	An echo request arrives and is answered	The host is alive and the path works both ways
Destination unreachable	3	A router has no route, or the host cannot accept the datagram	Delivery failed, and the code says why
Time exceeded	11	The time to live reaches zero, or fragments never reassemble	The datagram was discarded in transit for living too long
Redirect	5	A better first hop exists on the sender's own network	Use the other router for this destination

Table 3.15: *The ICMP messages worth knowing.*

Learn them as a group of pairs and singles: one question with its answer, two refusals, and one suggestion. That grouping survives pressure better than a flat list of five.

The Master Key: *Quote the number beside the name, since a capture will show you 3 and 11 and never a word of English.*

BINDER Question 108

The link from a secondary school at Tarime to the district office crawls. The head of ICT can name his own gateway, the clerk at the far end the last router, and nobody the machines between. One command prints all three in order.

Using Figure 3.18, describe how traceroute discovers the address of every router along a path.

Model Answer

Traceroute sends probes towards the destination with the time to live set to 1 on the first, 2 on the second and one higher each time, so every probe expires one hop further along and the router that discards it returns a time exceeded message, type 11, carrying its own address as the source; n hops therefore need n probes, and 3 routers are named by 3 probes.

The field it exploits: every IP datagram carries a time to live, and every router forwarding one reduces it by 1. A router that reduces it to 0 must discard the datagram and report the discard. The field exists to stop packets circling for ever in a loop; traceroute provokes it on purpose.

One probe for each hop: read the figure downwards. The first probe leaves with 1 and dies at R1. The second leaves with 2, passes R1 with 1 remaining and dies at R2. The third leaves with 3 and dies at R3.

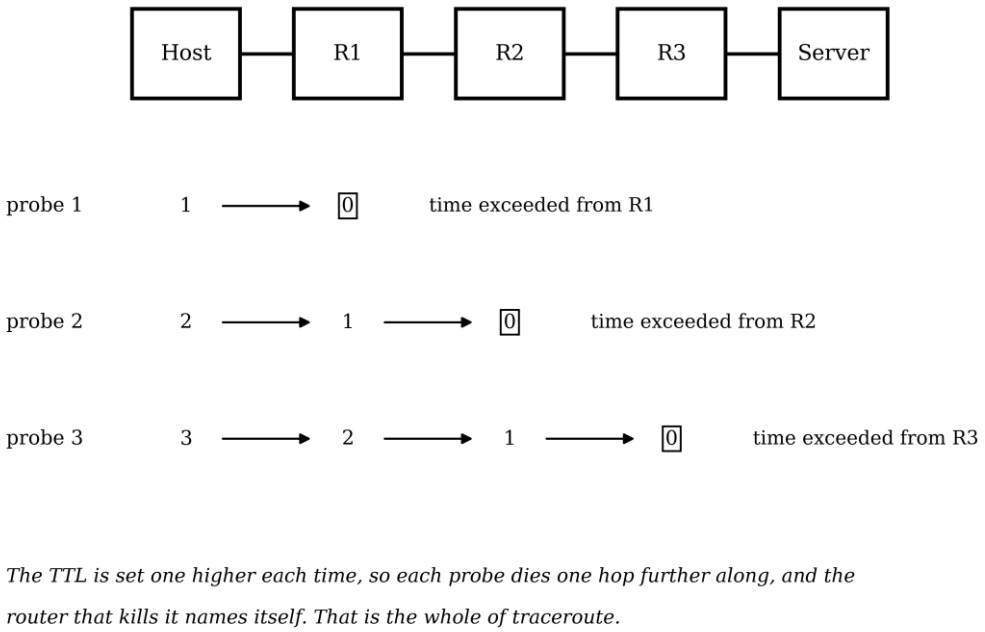


Figure 3.18: *How traceroute works. The time-to-live is raised by one on each probe, so each probe dies one hop further along and the router that discards it names itself.*

The router names itself: nothing in the probe requests an identity. The time exceeded message is generated on the router's own account, and its source address is that of the interface the dead probe arrived on, so a trace lists the addresses facing the sender.

How the tool knows to stop: the destination never sends time exceeded, since the probe arrives with life remaining. It answers with a port unreachable or an echo reply instead, and that different answer ends the trace.

Why asterisks appear: a router configured to generate no ICMP errors, or whose replies a firewall discards, contributes nothing to its row. The tool prints asterisks and continues, so a gap in the middle of a trace means one silent router rather than a broken path.

Every line of output is a separate small failure, deliberately caused and then read. Traceroute discovers nothing volunteered by the network; it collects error messages and puts them in order.

The Master Key: *When a tool reveals what no protocol records, look for the field it is abusing.*

BINDER Question 109

At the ferry ramp on Mafia the market-morning queue reaches the road, though nothing has broken. The vessel is the size it was yesterday. More lorries arrive each hour than can be carried away, and fewer cross than when the queue was short.

Define network congestion, and account for the way retransmission can turn it into congestion collapse.

Model Answer

Congestion is the condition in which the traffic offered to part of a network exceeds what that part can carry, so router queues lengthen, delay and jitter rise and packets are discarded once a buffer is full; retransmission worsens it because a sender that sees loss or long delay sends the packet again, adding load exactly when the network needs less.

Where congestion actually lives: not in a cable but in the queue before an output interface, wherever a fast side meets a slow one. A router passing a laboratory's traffic to one modest link is the classic place: the queue absorbs a short burst and fails on a sustained one.

Load, throughput and the cliff: as offered load rises, useful throughput rises with it, then flattens as queues fill, then falls. The falling part is the one to remember: beyond a certain load the network delivers less than it did at a lower load.

How retransmission feeds it: loss and delay both provoke a copy. A packet merely waiting in a long queue may be sent again before the first arrives, so two copies occupy the capacity one needed.

What collapse looks like: the links are fully occupied, the routers busy, and almost nothing useful arrives. Goodput falls towards nothing while utilisation reads 100 per cent, which is why a fully used link is not by itself good news.

Not a fault to be repaired: no device has failed and no setting is wrong. Demand has exceeded supply, so only three remedies fit: more capacity, less demand, or a rule deciding whose traffic goes first.

Congestion describes a network doing precisely what it was built to do under a load it was not built to carry, which is why a technician can find nothing wrong and still be looking at the fault.

The Master Key: *Congestion is a shortage, not a defect, so look for the queue that is growing rather than the device that is broken.*

REAL Question 110

The engineer at a district hospital in Bariadi is offered two settings on the router facing the provider, and no explanation of either. Both hold it to the same average rate. The hospital's traffic arrives in gusts, and each setting treats a gust differently.

Using Figure 3.19 and the arrivals specified below, compare what a leaky bucket and a token bucket do to the same six intervals of traffic, and state which of the two suits a video call.

Arrivals, in packets per interval: 9, 1, 0, 8, 2, 7.

Leaky bucket: depth 6 packets, drained at 3 packets per interval.

Token bucket: depth 6 tokens, refilled at 3 tokens per interval, and full at the start.

Model Answer

Both hold the flow to 3 packets per interval on average and both discard 8 of the 27 that arrive, but they differ in when traffic leaves: the leaky bucket emits a smooth 3 in every interval and queues the rest, while the token bucket releases 6 at once whenever 6 tokens have gathered, so a video call is far better served by the token bucket.

The two mechanisms in one line: the leaky bucket algorithm, correctly abbreviated LBA, is a queue drained at a fixed rate, so whatever the arrivals do the output is that rate and no more. A token bucket is a store of permits: tokens accumulate up to a limit, a packet may leave only by spending one, and an idle spell therefore buys a later burst.

The leaky bucket, interval by interval: 9 arrive on an empty bucket of depth 6, so 3 spill at once, 3 drain away and 3 remain. Then 1 joins those 3, 3 drain, 1 remains. Nothing arrives next, so the last 1 leaves. Then 8 arrive, 2 spill and 3 drain. Then 2 join, 3 drain. Then 7 arrive on the 2 left, 3 spill and 3 drain.

What the leaky bucket delivered: the output series is 3, 3, 1, 3, 3, 3, which is 16 packets forwarded, 8 discarded and 3 still queued when the sixth interval ends. The output never exceeds the drain rate, and the single interval below it is the one in which the bucket ran dry.

The token bucket, interval by interval: the bucket begins full with 6 tokens. 9 arrive, 6 are spent at once and 3 are discarded for want of a token. 3 tokens are added and the single arrival spends 1, leaving 2. Nothing arrives, so the store rises to 5. Refilled to 6, it passes 6 of the 8 and discards 2. Then 2 arrive against 3 tokens and both pass. Finally 4 tokens meet 7 packets, so 4 pass and 3 are discarded.

What the token bucket delivered: the output series is 6, 1, 0, 6, 2, 4, which is 19 forwarded and 8 discarded, with nothing left waiting. Both shapers therefore threw away the same 8; the token bucket simply delivered the survivors sooner, and in the shape they arrived.

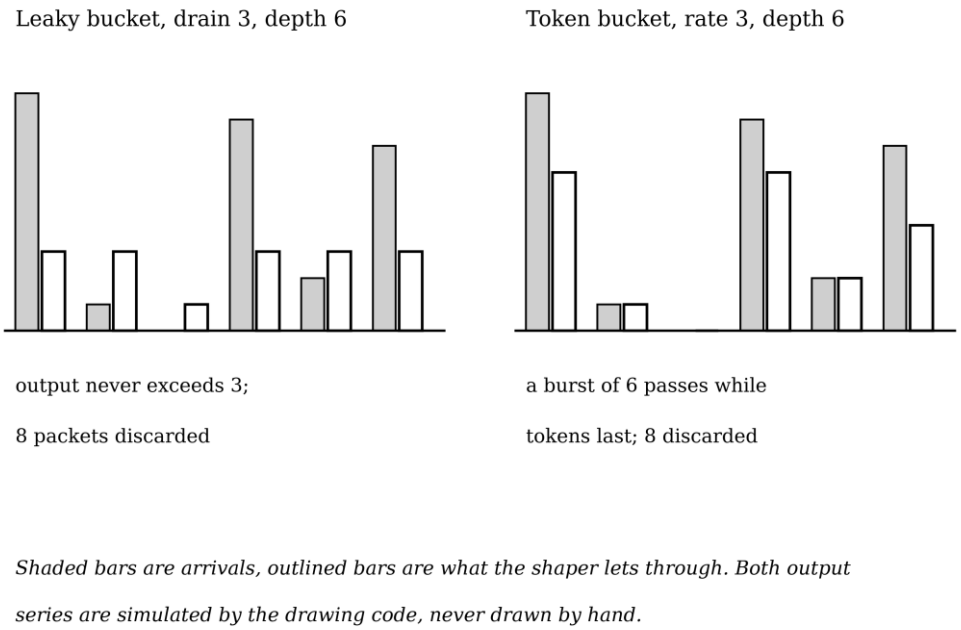


Figure 3.19: *A leaky bucket and a token bucket given the same arrivals. The leaky bucket never exceeds its drain rate; the token bucket lets a burst through while tokens last.*

Feature	Leaky bucket	Token bucket
Output rate	Fixed at the drain rate, here 3 per interval	Varies with the tokens held, up to 6 at once
Bursts	Removed, so the output is perfectly smooth	Allowed while tokens last, then held to the token rate
What the bucket holds	Packets waiting their turn	Permission to send, saved while the line is idle
Traffic in excess	Queued, then discarded once the bucket is full	Discarded on arrival when no token is left
Delay it adds	Whatever the queue imposes	None at all while tokens remain

Table 3.16: *Leaky bucket and token bucket compared.*

Which one suits a video call: the token bucket. A call is naturally bursty, because a complete picture is far larger than the small differences sent between pictures, and a packet held back in a queue arrives too late to be shown at all. The leaky bucket adds precisely the delay a call cannot absorb.

Choose the shaper by asking what the traffic cannot tolerate. Bulk transfer tolerates delay and should be smoothed; a conversation tolerates loss far better than lateness and should be allowed its burst.

The Master Key: *A leaky bucket protects the network from the sender, and a token bucket protects the sender from the network.*

REAL Question 111

Ping answers from every address the technician tries at a secondary school in Mbinga, and the replies come back promptly. No page will open, in any browser, on any machine in the laboratory. The head of department asks how both can be true.

Determine what this combination rules out and what it leaves, and set out the order in which the surviving possibilities should be tested.

Model Answer

A successful ping proves that the cabling, the addressing, the default gateway, the routing in both directions and the far host's network layer are all sound, so it eliminates a broken cable, a wrong mask, a missing gateway and a failed route; what remains lies above the network layer or outside ICMP altogether: name resolution, a filtered or proxied connection, a captive portal, a server refusing to serve, or a path that passes small packets and discards large ones.

What a reply actually proves: an echo request left the machine, crossed every router outbound, was accepted by a host that was switched on, and its reply crossed every router coming back. That is a complete round trip through the lower three layers, and no single command yields stronger evidence.

What is therefore eliminated: a disconnected cable, a dead switch port, an address in the wrong subnet, an absent or mistaken default gateway, and any failure of routing between the school and the address that answered. None of these can be true while replies are arriving.

Separate the name from the address: ping a public address known to answer, then ping a well known name. If the address replies and the name does not, the resolver is at fault: the machines point at a name server that has stopped answering, or the school's own server has lost its forwarder.

If the name resolves as well: the fault sits above the network layer. The provider may be permitting ICMP while blocking the ports a browser uses, a proxy may be set in the browser and no longer exist, a portal may be waiting for a login that nobody sees, or the server itself may be refusing connections.

The trap of the large packet: a path may pass a small echo request and discard a full-size data packet, because something along it reduced the largest size allowed and the message saying so is being filtered. Ping succeeds, the page begins and the transfer then stops dead. Test it by pinging with a large packet.

The order to work outwards: the gateway, then the provider's router, then a public address, then a name, then the page itself. The first step that fails names the layer, and a technician who stops at the first success discards the information the sequence was designed to give.

Ping tests reachability and nothing else. A path that answers is a path that exists, which is a considerable distance from a path that works.

The Master Key: *Read a successful test for what it removes, because the worth of a diagnostic lies in the possibilities it kills.*

REAL Question 112

An outgoing technician at a district council office in Pangani leaves a page of advice for his successor. Most of it is sound, learnt the hard way. The three lines below are offered in the same confident tone, and each points the wrong way.

Identify what is wrong with each of the three statements, and give the correct position with the reason for it.

1. The reply to a ping is called a Pong, and a machine that returns one is working normally.

2. The provider's firewall blocks ICMP, so anything beyond it is down and there is nothing further we can test.

3. Traceroute works by asking each router along the way to identify itself, and every router must answer.

Model Answer

All three are wrong: the reply to an echo request, type 8, is an echo reply, type 0, and no message called a Pong exists; a firewall that blocks ICMP is enforcing a policy and says nothing whatever about the

health of the network behind it; and traceroute asks no router for anything, since each one volunteers a time exceeded message of its own accord when a probe's time to live expires in its hands.

There is no Pong: the answer to echo request, type 8, is echo reply, type 0. The invented name matters because diagnostic output is read by number, and a successor hunting for a Pong will meet type 0 and fail to recognise it. The second half of the line is wrong too: an echo reply is produced by the network layer of the receiving machine and proves nothing about the services running above it.

Filtered is not down: discarding ICMP is a deliberate choice, made to render a network harder to scan, and it reports nothing about what lies behind the filter. Machines that answer no ping serve pages perfectly all day. Silence is an absence of evidence, not evidence of absence, so the next test should be of the service itself.

No router is asked anything: a probe is addressed to the destination, never to the routers. Each router discards it only because the time to live it happens to carry runs out there, and the time exceeded message that follows is sent under the ordinary rules of IP. A router told to generate no ICMP errors answers nothing and appears as asterisks, while the trace carries on past it.

The habit that prevents all three: each line treats a tool as an authority rather than as one particular kind of evidence. Before running any diagnostic, say what a success would prove and what a silence would prove. Ping proves a round trip at the network layer; traceroute collects errors that routers volunteer; neither was ever a report on a service.

The three share one shape. Each reads the behaviour of a tool as a statement about the network, when it is only a statement about that tool and one property of one path.

The Master Key: *Before you run a diagnostic, decide aloud what its silence will mean, because a test you cannot read when it fails is not a test.*

HOT Question 113

One link to the provider serves a secondary school in Sumbawanga: the office, the laboratory, and a video lesson from a partner college. It stutters every Thursday, at the hour the laboratory downloads its updates. The head of department wants it put first.

Assess whether the school should give video calls priority over file downloads on its single link, and commit to a recommendation.

Model Answer

Yes, but only as a complete scheme and never as a single setting: classify traffic at the school's own router, give video a strict priority capped at a stated share of the link, let bulk downloads have everything the lesson is not using, and shape the download class so that it cannot fill the queue in the first place.

What each flow really needs: a call needs its packets steadily and soon, and needs little capacity to do it. A download needs volume and does not care when any packet arrives. Requirements that different can be met together, which is why prioritisation is worth doing.

Priority creates no capacity: while the link is not full there is no queue to reorder and the scheme changes nothing. When it is full, every packet moved forward moves another back, so a gain to the lesson is paid for by the update, and the school must say so out loud.

The four parts of a scheme: quality of service is bandwidth management to fix the shares, classification and prioritisation to decide who belongs in which class, shaping and policing at the edge, and scheduling. A scheduler such as weighted fair queueing serves the queues in proportion, so the lesson gets its share whenever it wants it and the download gets the remainder rather than nothing.

Shaping and policing at the edge: a bucket on the download class holds it to an agreed rate so it never offers enough traffic to fill the queue, and a policer discards or demotes whatever exceeds that rate. Restraining the offender is gentler than promoting the victim.

Where strict priority fails: a class served first, always, and without a ceiling will take the whole link the moment it grows or the moment anything is misclassified into it. Cap it. A marking is only a claim, and a claim made by a student's machine must be rewritten at the school's router rather than believed.

The half the school cannot control: the congestion sits on the provider's link, and incoming traffic has already crossed it before the school's router sees a packet. The school can shape what it sends and ask the provider to police what it returns, but it cannot reorder what has arrived.

The judgement: put the scheme in, with two classes only: video first under a firm ceiling, everything else in one shared queue, and updates moved outside teaching hours. If the ceiling is refused, or the school cannot classify at its own edge, decline it and buy capacity instead.

Prioritisation is rationing, and rationing works only where the ration is written down. A scheme that names a favourite without naming its limit has not been designed at all.

The Master Key: *Give priority a ceiling in the same sentence that gives it a queue.*

HOT Question 114

The specification for a district network in Kondoa carries one line that reads like a guarantee: no packet shall be discarded. A supplier who signs must sell capacity the district cannot pay for, or fit buffers so deep that waiting is worse than loss.

Judge whether discarding packets can ever be the right response to congestion, and defend the position you take.

Model Answer

Yes, and more than that: once a queue is genuinely full, discarding is the only action left to a router, and discarding a few packets early and deliberately is better than discarding many late, because a small loss reaches senders as a signal to slow down while there is still room to recover.

There is no third option: a router holds finite memory and cannot order an upstream router to pause. Once memory is exhausted something goes, and refusing to choose merely means the newest arrival goes by default. The question was never whether to discard but which packet, and how early.

The cost of refusing to discard: a very large buffer converts loss into delay. Packets are kept and arrive a second late, which is worthless to a conversation, and because nothing was lost the senders receive no hint that they should send less. Delay then grows with the load and nothing acts to stop it.

Discard early, and at random: the sound practice watches the average queue length and, once it passes a threshold, discards a small proportion of arriving packets chosen at random. A few senders are asked to slow down before the queue is full, and choosing at random spreads that request across many flows rather than punishing whoever arrived at the instant of overflow.

Discard the right traffic: a discard is a decision, so make it. Drop from the class that is over its agreed rate rather than from the one that cannot recover, and drop early in the path rather than late, since a packet thrown away one hop from its destination has already consumed capacity on every link it crossed.

When discarding is the wrong answer: when the loss was never congestion, as on a poor radio link, discarding compounds a fault it did not cause. When the traffic will not respond, as with a live stream that sends at the same rate whatever happens, loss degrades the picture and leaves the load exactly where it was.

The judgement: discarding is legitimate and unavoidable, and a clause forbidding it should be refused rather than priced. What deserves criticism is not the discard but the manner of it: one chosen early, at random, from traffic that is over its share is a control action, while one caused by a buffer overflowing is a design failure that merely resembles a decision.

A network that never discards is not a network without loss. It is a network that has converted the loss into delay, where nobody can see it and nothing can respond to it.

The Master Key: *Choose your losses while you still can, because the packet a full buffer discards was chosen by nobody.*

IPv4 addressing: classes, masks and binary conversion

Every question in this part of the chapter rests on one thing: an IPv4 address is a single 32-bit number, and everything written about it is that number in a more convenient dress. This section settles how the 32 bits are written, how to move between binary and dotted decimal without guessing, where the boundary between the network part and the host part falls, and what the five classes were for. It assumes only that you can add. Later sections move that boundary; here it stays where the class puts it.

BINDER Question 115

A sticker on the underside of a switch in a school laboratory in Kinondoni carries four numbers separated by full stops. Nobody can say what they are, only that the network stops when they are changed. They are not four numbers at all.

Describe the structure of an IPv4 address, and explain what is meant by dotted decimal notation.

Model Answer

An IPv4 address is a single 32-bit binary number naming one interface on one network, and dotted decimal notation is its readable form: the 32 bits are cut into four groups of eight, each group becomes a value between 0 and 255, and the four are written in order separated by full stops.

Thirty-two bits in all: the address is one unbroken string of 32 binary digits and the machine treats it as one quantity. Written out in full it reads 11000000101010000000001100011000, which nobody could dictate over a telephone without error.

Four octets of eight bits: the 32 bits are divided into four fields, and a field of eight bits is an octet. Eight bits give $2^8 = 256$ patterns, from 00000000 to 11111111, so an octet begins at 0, ends at 255 and may take any whole value between.

Why 255 is the ceiling: an octet of 255 has all eight bits set and there is no ninth bit to carry into. An address containing 192.256.3.24 is not a large address but an impossible one, and equipment offered it rejects the entry outright.

The dots carry nothing: the full stops occupy no bits and mean nothing to the machine. They let the eye find the boundary between one octet and the next, and they go in only after the arithmetic is done.

What the address names: an address belongs to an interface, not to a machine. A router with three interfaces carries three addresses, and a laptop using a cable and a wireless card at once carries two. It answers where, never who.

How many there are: 32 bits give $2^{32} = 4{,}294{,}967{,}296$ distinct addresses in the whole of IPv4. That was ample when the scheme was drawn up and is not ample now, which drives much of what follows.

Four numbers on a sticker are one binary quantity dressed for human hands, and every rule about them follows from the eight bits behind each one.

The Master Key: *Read every octet as eight bits, and the limit of 255 stops being a rule you memorise and becomes something you can see.*

BINDER Question 116

The configuration sheet taped inside a wiring cabinet at a training centre in Unguja lists every address twice: once in the ordinary four-number form, once as thirty-two ones and zeros. Both columns are there for the reason a price list carries figures and words.

Convert 11000000 10101000 00000011 00011000 to dotted decimal, showing the place values used for every octet, and then convert your answer back to binary.

Model Answer

Writing the place values 128, 64, 32, 16, 8, 4, 2, 1 above each octet and adding only those values that stand above a 1 gives 128+64 = 192, then 128+32+8 = 168, then 2+1 = 3, then 16+8 = 24, so the address is 192.168.3.24, and subtracting the same place values from each decimal value in turn returns the original binary.

The eight place values: each position in an octet is worth twice the position on its right, so from the left they are 128, 64, 32, 16, 8, 4, 2, 1. Write that row above the bits before anything else. Every conversion is worked against it, and the arithmetic is addition.

The first octet: the bits are 11000000. Ones stand above 128 and above 64 and nowhere else, so the value is 128+64 = 192. The six zeros contribute nothing, because a zero is never added to the running total, only stepped over.

The second and the third: in 10101000 the ones stand above 128, 32 and 8, so the value is 128+32+8 = 168. In 00000011 only the last two places are set, giving 2+1 = 3. A long run of leading zeros always makes a small number, which checks your working.

The fourth octet: in 00011000 the ones lie above 16 and above 8, so the value is 16+8 = 24. Setting the four results side by side in the order they were converted gives 192.168.3.24, and the full stops are put in last.

Going back the other way: work along the same row from the left, asking at each place whether it fits inside what remains. Taking 168: at 128 yes, and 168 − 128 = 40 remains; at 64 no; at 32 yes, and 40 − 32 = 8 remains; at 16 no; at 8 yes, and nothing remains; the rest are zeros. That writes 10101000.

Checking by reversing: convert forwards, convert the answer straight back, and compare it with what you were given. Two conversions that disagree cost a minute to find and the whole question if they are never found.

The row of place values is the whole method, and a candidate who writes it down before touching the first bit rarely gets an octet wrong.

The Master Key: *Write 128 64 32 16 8 4 2 1 above the bits first, and conversion becomes addition going one way and subtraction going the other.*

BINDER Question 117

Nobody at the district office in Ifakara can agree about one address. The visiting engineer calls it a Class B, the router calls it a /22, and the wall diagram gives a mask that matches neither. All three are describing the same 32 bits.

Using Figure 3.20 and Table 3.17, state the leading bits, the first-octet range and the default mask of each assignable class, and say what Classes D and E were reserved for.

Model Answer

Class A opens with the bit 0, occupies first octets 1 to 126 and takes the mask 255.0.0.0; Class B opens 10, occupies 128 to 191 and takes 255.255.0.0; Class C opens 110, occupies 192 to 223 and takes 255.255.255.0; Class D opens 1110 across 224 to 239 and carries multicast, Class E opens 1111 across 240 to 255 and was kept for experiment, and neither is given to a host.

How the leading bits decide: the class is fixed by the first bits and by nothing else. A 0 in the leading position settles it as Class A; if that bit is 1 the second is examined, and if that is also 1 the third. The first-octet ranges are those patterns converted by place value, so an octet opening 10 runs from 128 to 191.

Where the split falls: Class A gives 8 bits to the network and 24 to the host, Class B divides them evenly, and Class C gives 24 to the network and 8 to the host. Figure 3.20 draws all three, and the boundary always lands on an octet.

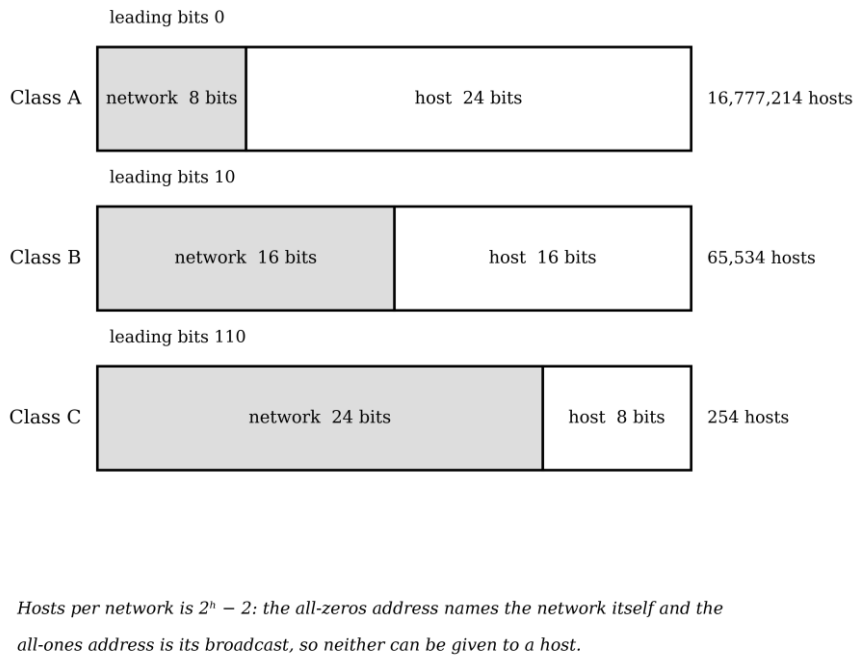


Figure 3.20: *How the 32 bits divide between network and host in each of the three assignable classes.*

What a default mask states: the mask is a second 32-bit number carrying a 1 in every network position and a 0 in every host position, so it announces where the boundary lies. The three defaults are 255.0.0.0, 255.255.0.0 and 255.255.255.0.

Prefix notation: the same mask may be written as a stroke and the count of leading ones, so the three become /8, /16 and /24. This is classless inter-domain routing notation. Both forms hold the same information, and the prefix is quicker to write.

Classes D and E: neither is assignable. Class D carries multicast group addresses, naming a set of listeners rather than one machine, and Class E was reserved for experiment and never released. Counting either among the networks available to an organisation is a common slip.

Class	Leading bits	First octet	Default mask	Networks	Hosts per network
A	0	1 to 126	255.0.0.0	126	16,777,214
B	10	128 to 191	255.255.0.0	16,384	65,534
C	110	192 to 223	255.255.255.0	2,097,152	254
D	1110	224 to 239	none defined	multicast	not assigned
E	1111	240 to 255	none defined	experimental	not assigned

Table 3.17: *The five address classes.*

The figure and the table state the same fact twice, once as a picture of the bits and once as numbers to quote from memory.

The Master Key: *Quote the leading bits before the first-octet range, because the bits are the definition and the range is only their decimal shadow.*

BINDER Question 118

A ward at a district hospital in Magu numbers its beds from one upward. Two numbers on the board are never given to a patient: one names the ward on every form leaving the building, the other is called when every bed must hear.

Explain why the number of usable host addresses on a network is two fewer than the number of addresses the network contains, naming both of the addresses that are lost.

Model Answer

The all-zeros host pattern is the network address, which names the network itself on plans and in routing tables, and the all-ones host pattern is the directed broadcast address, which reaches every host on that network at once; neither may be given to an interface, so the usable total is $2^h - 2$, where h is the number of host bits.

The formula and its terms: h is the count of bits left after the mask has claimed its share for the network. Those h bits can be arranged in 2^h patterns, and that is how many addresses the network contains. Two are already spoken for, so the number to hand out is $2^h - 2$.

$$\text{Usable hosts} = 2^h - 2$$

The all-zeros pattern: set every host bit to 0 and the result names the network as a whole. It appears in routing tables, on diagrams and in address plans, and it points at no single machine. A workstation given it would be indistinguishable from its own network.

The all-ones pattern: set every host bit to 1 and the result is the directed broadcast for that network. A packet sent to it reaches every interface on the network at once, which is what makes it useful and what makes it impossible to give to any one of them.

The three classes counted: Class C leaves 8 host bits, so $2^8 = 256$ addresses of which 254 may be used. Class B leaves 16, so $2^{16} = 65{,}536$ addresses and 65,534 usable. Class A leaves 24, so $2^{24} = 16{,}777{,}216$ addresses and 16,777,214 usable.

Where the marks are lost: the subtraction is forgotten far more often than the power is miscalculated. Write the number of addresses on one line and the subtraction beneath it, so that both reserved addresses appear in the working.

Two lost out of sixteen million is nothing; two lost out of 256 is a loss a small office feels, which is why the deduction is made every time.

The Master Key: *Count the addresses, take two away, and be ready to name both of the ones you took.*

REAL Question 119

An intern at a district council office in Sengerema is handed a page from an equipment log and must sort four addresses before the afternoon meeting: the class of each, its network, and the part that names the machine. The last is in binary.

Determine the class, the network identifier and the host identifier of each of the four addresses listed, using the default mask of the class in every case.

1. 10.42.7.19

2. 150.88.3.14

3. 200.45.11.6

4. 10000010 01100100 00000101 00001010

Model Answer

10.42.7.19 is Class A, network 10.0.0.0 with host 42.7.19; 150.88.3.14 is Class B, network 150.88.0.0 with host 3.14; 200.45.11.6 is Class C, network 200.45.11.0 with host 6; and the binary address converts to 130.100.5.10, which is Class B, network 130.100.0.0 with host 5.10.

The method in three steps: read the first octet and place the address in a class, take the default mask belonging to that class, then cut the address at the boundary the mask marks. Everything the mask covers is the network identifier, written out with the remaining octets as zeros, and everything it does not cover is the host identifier.

The first address: the leading octet is 10, which falls in the range 1 to 126, so this is Class A under the mask 255.0.0.0. Only the first octet belongs to the network, giving 10.0.0.0 as the network identifier and 42.7.19 as the host identifier. It is also a private address, so no registry issued it to anybody.

The second and the third: 150 lies between 128 and 191, so the second address is Class B under 255.255.0.0, giving network 150.88.0.0 and host 3.14. 200 lies between 192 and 223, so the third is Class C under 255.255.255.0, giving network 200.45.11.0 and host 6, the smallest host part of the four.

Converting the fourth: take the octets one at a time against the place values. In 10000010 the ones stand above 128 and 2, so 128+2 = 130. In 01100100 they stand above 64, 32 and 4, so 64+32+4 = 100. In 00000101 they stand above 4 and 1, so 4+1 = 5. In 00001010 they stand above 8 and 2, so 8+2 = 10. The address is 130.100.5.10.

Two routes to one class: the first octet is 130, which puts the address in Class B, and the binary confirms it without any conversion at all, because the address opens with the bits 10. Network 130.100.0.0 and host 5.10 then follow from the default mask. Two independent routes to one answer are worth taking when both are this cheap.

The trap in the fourth: an address written in binary invites a candidate to convert all 32 bits before thinking about anything else. The class needs only the leading bits and the split needs only the mask, so read the class off the front and convert because the answer has to be written in decimal.

All four were settled by the first octet and the default mask that follows from it, and nothing further inside any of the addresses was consulted at any point.

The Master Key: *First octet gives the class, class gives the mask, mask gives the cut, and the order never varies.*

REAL Question 120

A consultant quoting a secondary school in Kahama for an Internet connection prices one item as public address space registered to the school: 172.20.0.0/16. Beside it stands an annual charge. The head teacher, with 120 machines and no technician, asks whether it is worth paying.

Determine whether 172.20.0.0/16 is public address space, and advise the school on what to do about the offer.

Quotation item 4: public address block 172.20.0.0/16, registered to the school

Quotation item 5: annual charge for holding that block

The school runs 120 machines and wants all of them online

Model Answer

It is not public: 172.20.0.0 lies inside 172.16.0.0/12, one of the three ranges reserved by RFC 1918 for private use, so no registry issued it and no router on the public Internet will carry it; strike the item, take a private range free of charge behind network address translation, and pay only for the connection and the one public address on the router.

Reading the prefix: the /12 says that the first 12 bits of the address are fixed and the remaining 20 are free. Fixing 12 bits pins the whole first octet at 172 and the upper part of the second, which leaves the second octet free to run from 16 to 31. The reserved block therefore covers 172.16.0.0 to 172.31.255.255, and 172.20 sits well inside it.

What RFC 1918 set aside: three ranges were reserved for networks with no need to be reached from outside, and Table 3.18 lists all three with their sizes. Any organisation may use any of them at any time, without applying and without paying, which is exactly why none of them can be sold.

Why nobody owns it: a registry allocates public space and records the holder. Nobody holds 172.20.0.0, because it is in use in thousands of separate networks that never meet. Routers on the public Internet discard packets carrying such addresses, so a school given this block alone would reach nothing beyond its own gate.

What the school actually needs: 120 machines fit inside a private range many times over. The block on offer holds $2^{16} = 65{,}536$ addresses, of which 65,534 are usable, so the school could take it free of charge and still leave most of it idle. Every outgoing session is then translated at the router onto one public address.

The one thing to settle first: translation conceals the machines behind it, so anything the school later wants reached from outside needs a rule on the router pointing that traffic inward. Establish before signing whether the provider supplies a public address that stays the same.

Range	Prefix	Addresses
10.0.0.0 to 10.255.255.255	10.0.0.0/8	16,777,216
172.16.0.0 to 172.31.255.255	172.16.0.0/12	1,048,576
192.168.0.0 to 192.168.255.255	192.168.0.0/16	65,536

Table 3.18: *The private address ranges reserved by RFC 1918.*

How to answer the quotation: ask which registry issued the block and on what date. No answer will come, because Africa is served by AFRINIC and AFRINIC has never issued any part of that range. Strike item 4 and the charge beside it, and accept the rest on its merits.

Private addressing is not a cheap substitute for the real thing. It is the correct choice for a school of this size, and costing nothing follows from that rather than causing it.

The Master Key: *Before paying for an address, look for it among the private ranges, because nobody can sell you what everybody already has.*

REAL Question 121

Three pages of typed notes are left on the technician's desk at a district office in Pangani after a two-day visit by an outside adviser. Most of it is sound, which is the difficulty: the three statements below are already in a student's notebook.

Identify what is wrong with each of the three statements, give the correct version in each case, and explain why the correction is right.

1. There are 128 Class A networks, running from 0.0.0.0 to 127.255.255.255.

2. Public addresses are assigned to organisations by InterNIC.

3. 127.0.0.1 is a perfectly good address to give the server in the machine room.

Model Answer

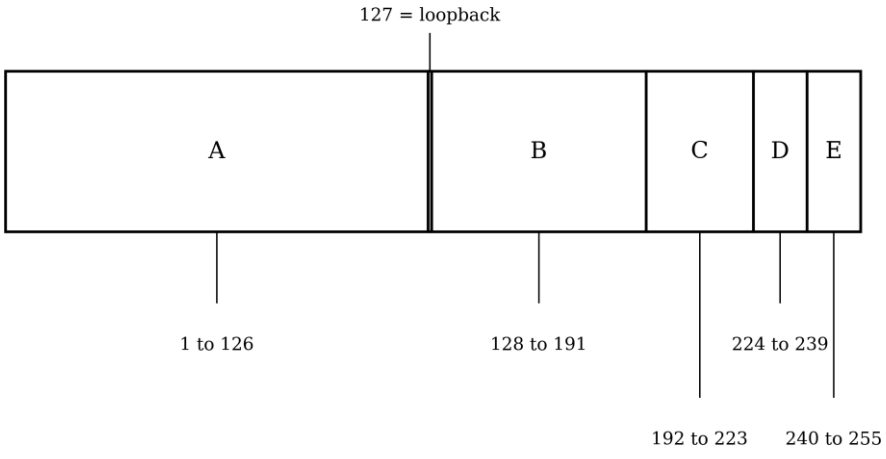
There are 126 usable Class A networks and not 128, because network 0 and network 127 are both reserved; public addresses come from IANA through the regional registries, and Africa is served by AFRINIC, InterNIC having been dissolved in 1998; and 127.0.0.1 is the loopback address, which no packet ever leaves the machine to reach, so a server given it can speak to nothing but itself.

Statement one, two networks short: seven bits are free in the network part of a Class A address, and $2^7 = 128$ is the number of patterns those bits can take. That is where the figure comes from, and it is where the counting stopped too soon. Network 0 means this network and network 127 is reserved for loopback, so $128 - 2 = 126$ remain and the first octet of a usable Class A address runs from 1 to 126.

Statement two, an office long closed: InterNIC ran registration services during the 1990s and was dissolved in 1998. Addresses are held by IANA, which delegates blocks to five regional registries, and the one serving Tanzania and the rest of Africa is AFRINIC. In practice a school or an office receives public addresses from its provider, who holds an allocation made by AFRINIC.

Statement three, an address going nowhere: the whole of 127.0.0.0/8 is loopback. A packet addressed there is turned round inside the sending machine and never reaches a cable, which is what makes 127.0.0.1 useful for testing that the software on a machine works at all. Configure it on a server and every other machine trying that address reaches its own loopback instead.

The reserved ranges to hold: keep four in mind and none of these three errors survives: 0.0.0.0 meaning this network, 127.0.0.0/8 for loopback, 169.254.0.0/16 for a host that asked for an address and was given none, and 255.255.255.255 as the limited broadcast, which reaches every host on the local network and is never forwarded.



There are 126 usable Class A networks, not 128: network 0 and network 127 are both reserved. Classes D and E are never assigned to hosts; D is multicast and E is experimental.

Figure 3.21: *The classes across the first octet. There are 126 usable Class A networks, not 128.*

What the three share: each was true, or nearly true, at some moment and was never checked again. Counting bit patterns without deducting the reservations, naming a body that has closed, and treating a reserved range as ordinary space are one habit at three removes, which is trusting a figure because it arrived looking like arithmetic.

Nothing in those notes was invented by the adviser. Every line is a fact that has outlived the arrangement it once described accurately.

The Master Key: *Deduct the reservations before quoting any total, and check that the organisation you are about to name still exists.*

HOT Question 122

Two teachers at a subject workshop in Kondoa disagree in front of the room. One says the classes are dead and the hours spent on them belong elsewhere. The other says a student who cannot state a default mask is lost inside a week.

Assess the reasons classful addressing was abandoned, and reach a judgement on whether it should still be taught.

Model Answer

Classful addressing was abandoned because three fixed sizes cannot fit organisations of every size, so address space was consumed far faster than need justified and routing tables grew with every allocation; it should still be taught, because the default masks, the reserved ranges and the whole vocabulary of the topic came out of it, and a mask is easiest to grasp where the boundary cannot move.

Three sizes and nothing between: an organisation could hold 254 usable addresses, 65,534 or 16,777,214, and there was no fourth option. A hospital needing 400 addresses was too large for one Class C, so it was given a Class B, and 65,534 − 400 = 65,134 addresses then sat idle where no other organisation could reach them.

Waste at the other end: the alternative was two Class C networks, giving 2 × 254 = 508 usable addresses split across two separate networks, which meant a router between them and two entries in every routing table that had to reach either one. Both answers were poor, and the choice between them was made thousands of times.

Tables that would not stop growing: every allocation added an entry to the tables carried by routers in the core of the Internet, and with 2,097,152 Class C networks available that growth had no natural ceiling. The tables were approaching the memory of the hardware holding them faster than the address pool itself was emptying.

What replaced it: classless inter-domain routing carries the prefix alongside the address, so the boundary is stated rather than inferred and may be placed at any bit rather than only at an octet. An organisation is then issued a block near the size it actually needs.

The case for dropping it: no router alive routes by class. A student who believes the first octet fixes the mask will misread every modern configuration, in which the mask is always stated and may be almost anything. Hours given to a scheme that decides nothing are hours taken from the scheme that decides everything.

The judgement: teach it, teach it early, and teach it as the origin of the vocabulary rather than as current practice. Default masks, the reserved ranges, the phrase Class C network and the habit of asking how many bits are left to the host all come from here, and a student who has never met a fixed boundary finds a movable one harder. Say plainly, once, that the class itself now decides nothing.

Classful addressing failed for a reason a student can state in a single sentence, and being able to state that reason is worth the hours by itself.

The Master Key: *Learn the classes as the question that classless addressing answers, never as a rule any router still obeys.*

HOT Question 123

A laptop in the staff room of a school in Msimbazi reaches neither the shared printer, nor the file store, nor anything outside, although the cable is seated and the port light is steady. Its address begins 169.254; every other machine begins 192.168.

Determine what has happened to produce an address beginning 169.254, and advise the technician what to check, in what order.

Model Answer

The machine asked for an address by DHCP, got no reply before its timeout expired and gave itself one from 169.254.0.0/16, the link-local range reserved for exactly that failure, so the address is a symptom and not the fault; check the path between the laptop and the DHCP server working outwards from the machine, taking the cable and the switch port first.

What the address means: 169.254.0.0/16 is set aside for automatic private addressing. A host that finds no DHCP server picks an address from that range, checks that no neighbour is already using it and carries on. It is a courtesy allowing two machines on an isolated segment to keep talking, not an address any router will forward.

Why nothing at all works: the machine has an address and a mask but no default gateway, because the gateway is one of the things DHCP would have supplied. The printer is unreachable because it sits in a different range, and the Internet because there is nowhere to send anything. A steady port light proves the cable carries a signal and nothing more.

Check the link first: reseat the cable at both ends, try a lead known to be good and try a different switch port. This is the commonest cause by a wide margin, it costs two minutes and it needs nobody's permission. A cable kicked under a desk all term is likelier than a server that failed silently.

Then the path to the server: confirm whether another machine on the same switch is still receiving addresses. If it is, the fault is confined to this laptop or its port. If it is not, either the DHCP service has stopped, usually on the router, or the link to it is broken, and one repair fixes every affected machine.

Then the pool: a DHCP server issues addresses from a fixed pool, and when the last is out on lease the next machine to ask is refused and falls back to 169.254. Look for a pool quietly filled by tablets and telephones, and shorten the lease before widening the range.

The order, and why: the judgement is to work outwards and stop at the first test that fails: cable, port, another machine on that switch, then the service and its pool. Once the path is repaired, release and renew the address, because the fallback will otherwise sit there until the laptop restarts.

An address the machine invented for itself is the network reporting, in the only way left open to it, that a question was asked and nothing answered.

The Master Key: *Treat 169.254 as a message rather than a fault: something asked, nothing replied, and your task is to find what stood between them.*

HOT Question 124

The board of a secondary school in Temeke has money for a proper network at last, and two proposals. One numbers every machine from a private range behind a single public address. The other buys a public address for every machine, and costs more.

Judge which of the two addressing schemes the school should adopt, and defend the choice.

Proposal A: private addressing from 192.168.0.0/16, translated to one public address at the router

Proposal B: a block of public addresses from the provider, one for every machine

The school runs 120 machines now and expects 200 within four years

Model Answer

Adopt Proposal A: private addressing behind network address translation costs nothing, offers 65,534 usable addresses where the school needs 200, leaves no machine reachable from outside unless somebody deliberately makes it so, and can be renumbered without consulting anyone, whereas a public block carries an annual charge, is drawn from a scarce pool and exposes 200 machines that no full-time technician is employed to defend.

What a private range costs: nothing at all. The three reserved ranges are free to any organisation that wants them, and the smallest of the three, 192.168.0.0/16, holds $2^{16} = 65{,}536$ addresses of which 65,534 are usable. A school planning for 200 machines has more room there than it can consume in the life of the buildings.

What a public block costs: an annual charge from the provider, and a wait, because public space is scarce and AFRINIC allocates it against demonstrated need. The school would also inherit the duty of recording which address sits on which machine, and of keeping that record current.

The security difference: a machine behind translation has no address anything outside can reach unless a rule is written to let it in. A machine holding a public address can be probed by anyone in the world, and 200 of them left unpatched across a long holiday is a serious exposure.

What translation costs: sessions started from outside fail unless the router is told where to send them, so a mail server, or a camera viewed from home, needs a rule and a public address that does not change. Tracing one machine from an external record is harder too.

The quiet advantage: a school that changes provider keeps its private numbering exactly as it stands and alters one address on the router. A school holding a provider's public block renumbers every machine, every printer and every written record on the day that contract ends, and does it under pressure.

The judgement and its exception: take Proposal A. Buy one public address for the router, add a second only if some service genuinely must be reachable from outside, and put everything else in a private range with room to grow. Proposal B suits an organisation that must publish services at scale, and a secondary school is not one.

The choice is not between a cheap arrangement and a proper one, because private addressing behind translation is what almost every organisation of this size runs.

The Master Key: *Ask what must be reachable from outside, and if the honest answer is nothing then no machine needs a public address.*

Subnetting I: borrowed bits, block size and host range

One network becomes four the moment the mask moves two places to the right, and the four are then fixed: their addresses, their host ranges and their broadcast addresses follow from the mask and from nothing else. This section settles that method and, just as importantly, the order in which it is written down, so that the same short sequence of lines answers every question of this kind. It assumes you can already convert an octet to binary and read a default mask off the first octet of an address. Every answer here checks itself before it stops.

BINDER Question 125

Two hundred machines in one district office at Sumbawanga share a single flat network, and each hears every broadcast the other hundred and ninety-nine send. Nothing is broken, no cable faulty. The wire carries a conversation that concerns almost nobody obliged to listen.

Explain what subnetting is, and give three reasons why an organisation divides one network into subnets.

Model Answer

Subnetting is the division of one IP network into several smaller logical networks by extending the mask, so that bits which were host bits become network bits; it is done to contain broadcast traffic, to separate groups whose traffic can then be controlled at a router, and to make the address plan follow the buildings it serves.

What the mask moves: the address stays thirty-two bits long; what moves is the boundary inside it. Bits taken from the front of the host part join the network part, so one network of many hosts becomes many networks of fewer hosts each, and n borrowed bits give 2^n subnets.

Containing broadcast traffic: a broadcast reaches every host in its own subnet and stops there, because a router does not forward it. Divide two hundred hosts into four subnets and a broadcast disturbs 200 / 4 = 50 machines rather than two hundred, sparing the rest a frame they would only discard.

Separation for control: traffic from one subnet to another must cross a router, and a router is a single place where a rule can be written. Accounts records can be kept unreachable from a student laboratory by filtering there. A flat network offers no such point at all.

Matching the plan to buildings: one subnet per block or department makes an address readable: the octet carrying the subnet number says where the machine stands. Fault reports, asset records and cable labels then agree, which shortens every diagnosis beginning with an address and a complaint.

What division costs: each subnet surrenders two addresses, one naming the subnet and one for broadcasting on it, and each needs a router interface to reach the others. Subnetting is a trade rather than an improvement, worth making only where there is something real to separate.

The office at Sumbawanga has no fault to repair. It has a boundary nobody drew, and drawing it changes the mask rather than the cabling.

The Master Key: *Move the boundary, not the address, and count what each new subnet costs before you draw it.*

BINDER Question 126

Every subnet gives up two of its addresses before a single machine is plugged in. A technician in Stone Town who orders one wireless access point for every address he counted will be two short on each subnet, whatever size he made the blocks.

State the formula for the number of usable hosts on a subnet, explain each of its terms, and use it to choose a prefix for a department of thirty machines.

Model Answer

The number of usable hosts on a subnet is two raised to the number of host bits, less two, where the power gives every address the block contains and the two subtracted are the pair no interface may hold: the all-zeros host field, which names the subnet, and the all-ones host field, which is its directed broadcast.

$$\text{Usable hosts} = 2^{h} - 2$$

Finding h: h is the number of host bits left after the mask, so h = 32 − prefix and nothing else. A /26 leaves 32 − 26 = 6 host bits and a /20 leaves 32 − 20 = 12. Count what the mask leaves, never what it takes, and the class the address came from does not enter the calculation at all.

Why exactly two are lost: an address whose host field is all zeros identifies the subnet itself and is what appears in a routing table. An address whose host field is all ones is the directed broadcast, which every host on that subnet must accept. Neither may be configured on an interface, whatever the size of the block.

The formula worked forwards: a /26 has 6 host bits, so $2^6 = 64$ addresses of which $2^6 - 2 = 62$ are usable. A /20 has 12 host bits, so $2^{12} = 4{,}096$ addresses of which $2^{12} - 2 = 4{,}094$ are usable. The subtraction is always two, however large the block.

The formula worked backwards: given a requirement, invert it. Thirty machines need $2^h - 2$ to be at least 30. Four host bits give $2^4 - 2 = 14$ and fail; five give $2^5 - 2 = 30$ and fit exactly. So h = 5, the prefix is 32 − 5 = /27, the mask is 255.255.255.224 and the block size is 256 − 224 = 32.

No subtraction for subnets: the deduction of two belongs to hosts alone. Borrowing bits for subnets loses nothing, so three borrowed bits give $2^3 = 8$ subnets and not six. Write 2^n for subnets and $2^h - 2$ for hosts, and keep the two formulae in separate columns of your working.

A department of thirty fits a /27 with nothing to spare, so one printer nobody counted forces the block up to a /26.

The Master Key: *Count the host bits the mask leaves, raise two to that number, then take two away and never take them anywhere else.*

BINDER Question 127

The wall of a small office in Msimbazi carries a printed strip showing one network cut into four, the ranges written underneath in a neat hand. Copying them off the wall is one thing. Producing them from the mask alone is another.

Using Figure 3.22, divide 192.168.1.0/24 into four equal subnets, and state for each of them the mask in both notations, the network address, the usable host range and the broadcast address.

Model Answer

Borrow two bits, because $2^2 = 4$; the mask becomes 255.255.255.192, which is /26; the block size is 256 − 192 = 64, so the four subnets begin at 192.168.1.0, .64, .128 and .192, and each holds $2^6 = 64$ addresses of which $2^6 - 2 = 62$ are usable.

How many bits to borrow: the requirement is four subnets and n borrowed bits give 2^n, so $2^2 = 4$ borrows exactly two. Those two come from the front of the host part, which on a /24 is the whole fourth octet, and the prefix therefore moves from /24 to 24 + 2 = /26. Always settle this number before writing anything else down.

The mask in both notations: a /26 is twenty-six leading ones. The first three octets are full at 255 each, and the fourth begins 11000000, which is 128 + 64 = 192. The mask is 255.255.255.192. The two notations are one statement written twice, and an examiner who asks for the mask expects to see both of them.

The block size: subtract the interesting octet of the mask from 256, so 256 − 192 = 64. Subnets then begin at every multiple of 64 in that octet, which is 0, then 0 + 64 = 64, then 64 + 64 = 128, then 128 + 64 = 192. Once the block size is known no further arithmetic is needed, only counting.

The four subnets in order: for each one give the network address, the first host, the last host and the broadcast, in that sequence. The first is 192.168.1.0, hosts .1 to .62, broadcast .63. The second is .64, hosts .65 to .126, broadcast .127. The third is .128, hosts .129 to .190, broadcast .191. The fourth is .192, hosts .193 to .254, broadcast .255.

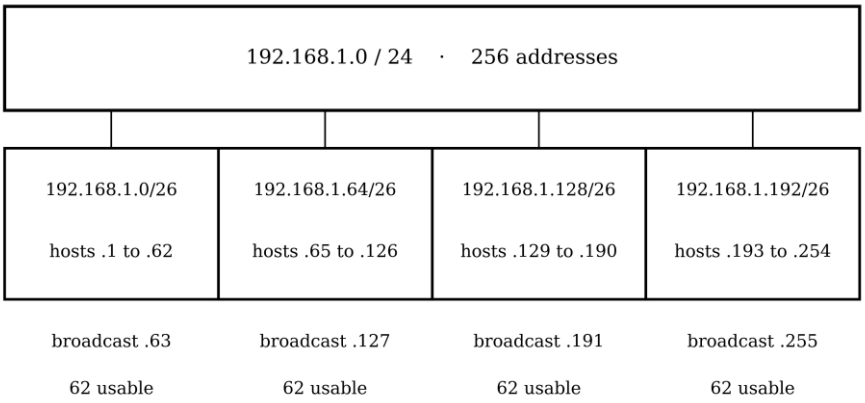


Figure 3.22: *One /24 divided into four /26 subnets, with the usable host range and the broadcast address of each.*

The check that catches errors: each broadcast address must be one below the next network address. Here 63 + 1 = 64, 127 + 1 = 128 and 191 + 1 = 192, and the last broadcast is .255, the top of the original block. Run that comparison down the whole list before you write the answer out fairly, because it finds a slipped block size at once.

The strip on the wall at Msimbazi is only a record of this working, and anyone who can produce the working can rebuild the strip for any mask at all.

The Master Key: *Bits, mask, block size, then four lines per subnet in the same order, and let the check close the list.*

BINDER Question 128

In a staffroom at Mbinga a teacher fills a whole page converting octets to binary and arrives, correctly, at eight subnet addresses. A colleague writes the same eight in under a minute from one subtraction. The second is not a shortcut around the binary.

Explain the block-size method, and use it to list the network addresses that a /27 mask produces on a Class C network.

Model Answer

The block size is 256 minus the interesting octet of the mask, so a /27 mask of 255.255.255.224 gives 256 − 224 = 32, and the subnets begin at every multiple of 32: .0, .32, .64, .96, .128, .160, .192 and .224, which is $2^3 = 8$ subnets of $2^5 − 2 = 30$ usable hosts each.

The interesting octet: it is the octet of the mask that is neither 255 nor 0, and there is never more than one. On a Class C network divided between /25 and /30 it is the fourth. All the arithmetic happens inside that octet and the others are copied down unchanged.

Why the subtraction works: the interesting octet holds some leading ones and then zeros, and the zeros are the host bits inside that octet. Subtracting from 256 recovers their weight: 256 − 224 = 32, and 32 = 2^5, so five host bits sit in that octet. The subtraction is raising two to a power without troubling to write the power.

Counting up in blocks: start at zero and add the block size, writing each value and stopping before 256 itself, since 224 + 32 = 256 ends the original network. A block of 32 gives eight values, which agrees with $2^3 = 8$ for three borrowed bits. That agreement is the first check available and it costs nothing.

Reading the rest off a block: the broadcast address of any subnet is the next network address less one, and the usable hosts lie between them. So the subnet at .32 has broadcast 64 − 1 = .63, first host .33 and last host .62. The same three lines follow from every entry in the list.

Prefix	Dotted-decimal mask	Block size	Usable hosts
/24	255.255.255.0	256	254
/25	255.255.255.128	128	126
/26	255.255.255.192	64	62
/27	255.255.255.224	32	30
/28	255.255.255.240	16	14
/29	255.255.255.248	8	6
/30	255.255.255.252	4	2

Table 3.19: *Prefix, mask, block size and hosts.*

Two rules the table obeys: shorten the prefix by one bit and the block size doubles; and the usable hosts of any row is its block size less two. The /26 row is the one to hold in memory, since a mask of 255.255.255.192, a block of 64 and 62 usable hosts is the pattern most questions are built on.

Converting to binary is never wrong, only slow, and a paper setting five subnetting questions will not let you be slow twice.

The Master Key: *Take the mask octet from 256 and count upwards in that block, because the binary has already been done for you.*

REAL Question 129

Six teaching blocks stand along one side of a school compound at Bariadi, and the contractor pulling fibre wants each to have its own network. The largest block will hold twenty-five machines. The school has one Class C address and no more.

Determine the mask the school must use, and list every subnet with its network address, usable host range and broadcast address.

Address allocated to the school: 211.77.20.0

Networks required, one for each teaching block: 6

Machines on the largest block: 25

Model Answer

Three bits must be borrowed, because $2^2 = 4$ is too few and $2^3 = 8$ is enough for six; the mask is /27, or 255.255.255.224; the block size is 256 − 224 = 32; and each subnet carries $2^5 - 2 = 30$ usable addresses, which covers the twenty-five the largest block needs with five to spare.

Start from the class: 211 lies between 192 and 223, so this is a Class C address and its default mask is /24. That leaves one octet of host bits, eight in all, and every bit borrowed for a subnet must come out of those eight. Fixing the starting prefix before anything else is what stops the rest of the working drifting.

Bits for the subnets: six networks are wanted and n borrowed bits give 2^n. Two bits give $2^2 = 4$, which is short by two. Three bits give $2^3 = 8$, which is enough with 8 − 6 = 2 subnets left over. Borrow three, and the prefix becomes 24 + 3 = /27. There is no arrangement that produces exactly six, so having spares is normal.

Prove the hosts still fit: three bits borrowed leave 8 − 3 = 5 host bits, and $2^5 - 2 = 30$ usable addresses. The largest block needs twenty-five, so 30 − 25 = 5 addresses remain for a printer or an access point. Had it needed thirty-five, /27 would have failed and six subnets could not have been cut from one Class C address at all.

The mask in both notations: a /27 is twenty-seven leading ones: three full octets, then 11100000 in the fourth, which is 128 + 64 + 32 = 224. So /27 and 255.255.255.224 are the same instruction, and the block size follows immediately from the second of them as 256 − 224 = 32.

The six subnets in order: network, first host, last host, broadcast, for each in turn. First 211.77.20.0, hosts .1 to .30, broadcast .31. Second .32, hosts .33 to .62, broadcast .63. Third .64, hosts .65 to .94, broadcast .95. Fourth .96, hosts .97 to .126, broadcast .127. Fifth .128, hosts .129 to .158, broadcast .159. Sixth .160, hosts .161 to .190, broadcast .191.

The spares and the check: the two unused subnets are .192 and .224, and they should be recorded as reserved rather than forgotten. Now check downwards, since every broadcast must be one below the next network: 31 + 1 = 32, 63 + 1 = 64, 95 + 1 = 96, 127 + 1 = 128, 159 + 1 = 160 and 191 + 1 = 192, which is the first spare.

Twenty-five machines and six networks pull against each other on the same eight bits, and three borrowed bits is the only setting that satisfies both of them at once.

The Master Key: *Borrow for the networks you must have, then prove the hosts still fit, and never carry out those two steps the other way round.*

REAL Question 130

An address written on a support ticket at a district office in Tarime reads 135.65.20.31, and the mask beside it is described only as the one giving sixteen subnets. Before anything is repaired, the engineer must decide whether a machine may hold it.

Determine the subnet address, the broadcast address and the valid host range for 135.65.20.31 when the mask is the one that divides its network into sixteen subnets.

Address on the ticket: 135.65.20.31

Subnets the mask must produce: 16

Model Answer

Sixteen subnets need four borrowed bits, because $2^4 = 16$, so the mask is /20, or 255.255.240.0; the block size is 256 − 240 = 16 in the third octet; the address falls in the subnet beginning 135.65.16.0, whose first host is 135.65.16.1, last host 135.65.31.254 and broadcast 135.65.31.255.

The class fixes the start: 135 lies between 128 and 191, so this is a Class B address with a default mask of /16 and sixteen host bits. The borrowing therefore starts from /16 and not from /24. Reading a Class B address as though it were Class C is the commonest way to lose every mark on a question of this shape.

Four bits for sixteen subnets: $2^4 = 16$ exactly, so borrow four and the prefix becomes 16 + 4 = /20. That leaves 32 − 20 = 12 host bits, so each subnet holds $2^{12} = 4{,}096$ addresses of which $2^{12} - 2 = 4{,}094$ are usable. A subnet on a Class B network is a large thing, and the size is a useful sanity check on the prefix.

The interesting octet has moved: a /20 mask is 255.255.240.0, and the octet that is neither 255 nor 0 is the third. Every boundary in the answer therefore lives in the third octet, and the fourth is free to run its whole range from 0 to 255 inside a single subnet without crossing anything.

The block it lies in: 256 − 240 = 16, so subnets begin at third-octet values 0, 16, 32, 48 and so on. The address carries 20 in that octet, and 20 lies at or above 16 but below 16 + 16 = 32, so the subnet it belongs to begins at 135.65.16.0. Round the interesting octet down to the nearest multiple of the block size.

The four answers in order: the network address is 135.65.16.0; the first host is 135.65.16.1; the broadcast is the address immediately below the next subnet, which starts at 135.65.32.0, giving 135.65.31.255; and the last host is one below the broadcast, 135.65.31.254. Write them in that sequence and none of the four can go missing.

What the ticket address is: 135.65.20.31 sits inside the usable range, so it is an ordinary host address and the machine may keep it. Its final octet of 31 looks like a broadcast only because 32 − 1 = 31 is a boundary on a /27, and there is no /27 here. On a /20 the fourth octet carries no boundary whatever.

The whole of this answer turns on which octet the mask stops in, and the fourth octet, which looks so suspicious, has nothing to do with it.

The Master Key: *Locate the interesting octet before you locate anything else, because every boundary in the answer is inside it.*

REAL Question 131

A page of subnetting working is handed in at a school in Kondoa. The layout is careful, the method broadly sound, and three lines are false. Each is the kind of statement a class repeats across a desk until it sounds like a rule.

Identify what is wrong with each of the three statements, give the correct version, and explain why the correction is right.

1. Borrowing three bits from the host part gives six subnets, because two of the eight are reserved.

2. The broadcast address of a subnet is its last usable host address.

3. A mask of 255.255.255.100 is valid, since any value from 0 to 255 may be written in the last octet of a mask.

Model Answer

Three borrowed bits give $2^3 = 8$ subnets and not six, because nothing at all is deducted from a subnet count; the broadcast address is one above the last usable host and is never the same address as it; and 255.255.255.100 is not a mask, because 100 is 01100100 in binary and a mask must be an unbroken run of ones followed by an unbroken run of zeros.

Statement one, a stray subtraction: the deduction of two belongs to hosts, where the all-zeros and all-ones host fields genuinely cannot be issued. Subnet numbers carry no such restriction in current practice, so three bits give $2^3 = 8$ usable subnets. Writing $8 - 2 = 6$ is taking a rule from the host column and applying it in the subnet column, where it has never been true.

Where the old rule came from: the earliest routing protocols carried no mask in their updates, so a router could not tell the first subnet from the whole network, nor the last subnet from a broadcast to all of it. Both were therefore barred. Every protocol now in service carries the mask, and the restriction went out with the equipment that needed it.

Statement two, one address apart: on 192.168.1.0/26 the last usable host is .62 and the broadcast is $64 - 1 =$ .63. They are adjacent, not identical, and the confusion costs twice over: a machine is configured on an address no interface may hold, and the range written into the plan is one address longer than it should be.

Statement three, masks are contiguous: 100 is 01100100, ones and zeros interleaved, so it marks no boundary. A mask exists to say where the network part stops, and a stopping place is a single point, not several. Only nine values may appear in an octet of a mask: 0, 128, 192, 224, 240, 248, 252, 254 and 255, each formed by adding the next power of two down to the one before.

The habit that prevents all three: every one of these is a correct idea applied one column, one address or one octet away from where it holds. Convert the doubtful octet to binary and look at it before trusting it. Eight bits take a few seconds to write out, and all three statements fall apart the moment they are written.

Nothing on that page was carelessness. Each line is a true statement carried a short distance from the place where it was true.

The Master Key: *Take two from hosts and never from subnets, and test any doubtful mask octet against the nine values a mask may hold.*

HOT Question 132

Four departments at a district hospital in Ifakara need a network each from one Class C block: sixty machines in records, twenty-eight in the laboratory, twelve in the pharmacy, twelve in administration. One mask must serve all four, and no second block is coming.

Determine whether four equal subnets cut from a single /24 can serve these four departments, and judge what the hospital should do.

Records: 60 machines

Laboratory: 28 machines

Pharmacy: 12 machines

Administration: 12 machines

Address space available: one /24, and no more

Model Answer

They can, but at one prefix only: /26 is the single equal-length mask that yields four subnets and still holds sixty hosts, giving $2^2 = 4$ subnets of $2^6 - 2 = 62$ usable addresses each, so adopt it, place records on the first subnet, and record 62 as a ceiling that department will reach long before any of the other three is half full.

Test the prefix above: a /25 borrows one bit and gives $2^1 = 2$ subnets of $2^7 - 2 = 126$ usable addresses. The hosts are ample but two subnets cannot serve four departments, so /25 fails on the count of networks and can be dismissed in a line.

Test the prefix below: a /27 borrows three bits and gives $2^3 = 8$ subnets of $2^5 - 2 = 30$ usable addresses. Eight networks is generous, but records needs sixty and 30 will not hold sixty. So /27 fails on hosts, /25 fails on networks, and /26 is what survives. The determination is arithmetic rather than preference.

What the chosen mask gives: the mask is 255.255.255.192 and the block size is 256 − 192 = 64, so the four subnets begin at fourth-octet values 0, 64, 128 and 192. Records takes the first, hosts .1 to .62 with broadcast .63; the laboratory the second, .65 to .126, broadcast .127; the pharmacy .129 to .190, broadcast .191; administration .193 to .254, broadcast .255.

The waste, counted honestly: the four departments need 60 + 28 + 12 + 12 = 112 addresses and the plan provides 4 × 62 = 248, so 248 − 112 = 136 addresses sit idle. On a private block that costs nothing at all: no one is billed for them and nothing else is waiting to use them.

Where the risk actually sits: records has 62 − 60 = 2 addresses of headroom. Three more machines and the subnet is full, and the remedy then is not an adjustment but a new mask across the whole block and a readdressing of all four departments at once. Count the printers, the access points and the router interface into the sixty first, because each of those holds an address too.

The judgement: adopt /26 on all four departments now and write the ceiling into the plan where the next technician will find it. Cutting the block into unequal pieces, which is a later refinement, would not rescue records either, since sixty hosts demand a /26 however the remainder is divided. Ask for a second block only if records is expected to pass sixty machines, and ask before the plan is built.

Equal subnets do serve this hospital, and they serve it with precisely two spare addresses in the one department that cannot afford to be surprised.

The Master Key: *Test the prefix above and the prefix below before you adopt either, because a mask that only just fits fails on the day a delivery arrives.*

HOT Question 133

A technician sizing subnets for a fish-landing station at Pangani knows each carries about thirty machines, and hesitates between two masks a single bit apart. One fits the requirement. The other leaves room to grow, and the room turns out not to be free.

Assess what it costs to choose a mask one bit more generous than the requirement, and reach a judgement on which way an address plan ought to err.

Model Answer

One bit of generosity doubles every subnet and halves their number, so a /24 cut at /26 instead of /27 gives $2^2 = 4$ subnets of 62 in place of $2^3 = 8$ of 30, leaves 4 × 32 = 128 addresses never issued and destroys four segments; err generously all the same, because idle addresses are recovered at leisure while a subnet that is too small must be rebuilt with people working on it.

What one bit changes: shortening the prefix by one moves a bit back from the network side to the host side. The block size doubles, usable hosts rise from 30 to 62, and the number of subnets falls from eight to four. One bit is never a small adjustment: it is a factor of two in both directions at once.

The cost in addresses: each subnet then holds 62 usable addresses for a department of thirty, so 62 − 30 = 32 are never issued, and across four that is 4 × 32 = 128. On a private range this is a paper loss. Nothing is bought or billed, and the addresses are reclaimed whenever the plan is next revised.

The cost in segments: this is the real one. The block now yields four networks where it would have yielded eight, since 8 / 2 = 4. A site with five buildings has just been made impossible to address, and no quantity of spare host addresses inside the four will conjure a fifth network out of them.

Erring the other way: a /28 offers $2^4 - 2 = 14$ usable addresses, so a department of thirty is $30 - 14 = 16$ machines short. Such a subnet does not slow down or fill gradually: it simply stops accepting machines, and the repair means a new mask on every host, every printer, every reservation held by the address server and every rule that named the old range.

The two failures are unequal: spare addresses cost nothing and are corrected on a quiet Saturday by whoever next opens the plan. A subnet that is short is discovered on the morning the machines arrive and corrected while the site is trying to work. Cost the disruption rather than the addresses, and the comparison stops being close.

The judgement, and where it turns: err generous by one bit on private address space, provided the segment count still leaves a subnet or two unused. The judgement reverses where segments are the binding constraint or the block is public and paid for; there, size to the requirement and take growth as a separate block. Name which of the two is scarcer before choosing, and the mask chooses itself.

Generosity buys a department room to grow and costs the site half its networks, so the question is never whether to be generous but which commodity the block is short of.

The Master Key: *Decide first whether subnets or hosts are scarcer, then give the spare bit to whichever of them is not.*

HOT Question 134

A visiting adviser recommends dividing the network of a secondary school at Makambako into subnets, on the ground that dividing networks is good practice. The school runs thirty computers, all in one laboratory, off one switch and one router. Here the advice is wrong.

Judge whether a school of thirty computers on a single switch should subnet its network at all, and defend the judgement.

Model Answer

It should not: a flat /24 offers $2^8 - 2 = 254$ usable addresses for thirty machines, so only 11.8 per cent of the block is in use, and subnetting would buy containment of a broadcast load that thirty hosts do not generate and a separation that one switch cannot enforce without a router interface for every subnet; keep the single network, and revisit the decision the day a second building, a records server or a visitors' network appears.

What the school has: thirty machines, one switch, one router and one room. Every host is one hop from every other and one hop from the gateway. There is no traffic passing between groups that anyone might wish to control, for the plain reason that there are no groups to pass between.

The benefit, measured: the case for dividing rests first on broadcast containment, and the broadcast load of thirty hosts is a handful of address resolution requests and some discovery messages. Splitting thirty into 30 / 2 = 15 and 15 halves a quantity that was never troubling anything. That argument is real at three hundred hosts and imaginary at thirty.

The cost in addresses: two /25 subnets give $2 \times 126 = 252$ usable addresses against 254 on the flat block, so $254 - 252 = 2$ addresses disappear into a second network and broadcast pair. The loss itself is trivial. What matters is the direction of travel, since every further division takes another pair.

The cost in equipment and error: traffic between subnets must be routed, so each subnet needs a router interface, or one interface configured to serve several. That is a configuration somebody must maintain and a technician must understand at midnight. A single mistyped mask then produces a fault that presents exactly like a broken cable and is not one.

When the answer changes: divide when there is a boundary worth drawing: a second building with a switch of its own, a server holding examination records that ought to be reachable only from the office, or a wireless network for visitors that must not see the school machines. Any one of those converts an abstract benefit into a concrete one that can be stated in a sentence.

The judgement and its record: leave the school on one subnet, and write down in the network file why the decision was taken, so the next adviser is answered from the record rather than from scratch. Design for the school that exists, and add the division on the day the second switch is bought.

Good practice is a summary of decisions taken in larger places, and a laboratory of thirty machines on one switch is not one of the places where those decisions were taken.

The Master Key: *Divide a network when there is a boundary to draw, and one room with one switch offers you none.*

Subnetting II: VLSM, supernetting and summarisation

One block, several subnets, and no two of them the same size: that is the second half of subnetting, and what settles it is an order of work rather than a new formula. This section allocates one /24 to four requirements that differ by a factor of twenty-five, shows the two ways such a plan fails, and then runs the whole operation backwards, collapsing four separate networks into a single advertisement. It assumes you can already choose a prefix from a host requirement and read a host range off a block size.

BINDER Question 135

The address plan on the wall at a secondary school in Kahama cuts one /24 into four quarters. Sixty-two addresses sit on a link holding two interfaces. A fifth department arrives in January to find no block left and more than half the addresses idle.

Explain what variable-length subnet masking is, and explain why subnets of equal size waste addresses when the requirements differ.

Model Answer

Variable-length subnet masking applies a different mask, and therefore a different block size, to each subnet cut from one address block, so that every subnet is sized to its own requirement; equal subnets waste addresses because the single mask must satisfy the largest requirement, and every smaller subnet then inherits a block it cannot fill.

One block, several masks: the block is divided for the largest requirement first, and what remains is divided again for the next, and so on down the list. Each subnet carries its own prefix, and since a router stores the prefix beside every route, blocks of four different sizes sit in one table without ambiguity.

Why one mask must be generous: a single mask fixes one block size for the whole plan, and that size has to hold the largest requirement. Usable addresses on a subnet are $2^h - 2$, where h is the count of host bits the mask leaves, so fifty hosts force a /26: $2^5 - 2 = 30$ is too few and $2^6 - 2 = 62$ fits.

The waste, counted: four equal /26 blocks take $4 \times 64 = 256$ addresses, which is the whole of a /24, and nothing is left over. The router link receives 62 usable addresses to hold two interfaces, so $62 - 2 = 60$ of them can never be issued to anything.

What sizing to fit returns: give the fifty a /26, the twenty-five a /27, the ten a /28 and the link a /30, and the four take $64 + 32 + 16 + 4 = 116$ addresses. The same requirements are met and $256 - 116 = 140$ addresses stay free for whatever the site asks for next.

Where the saving is real: spare addresses on a private block cost nothing while they lie unused, but they are the only source of the next subnet, and a plan with none left must be redrawn across every machine at once.

Equal subnets are not wrong. They are the special case in which every requirement happens to be about the same size, and fifty, twenty-five, ten and two is not that case.

The Master Key: *Size each subnet to its own requirement, because one mask serves everybody only when everybody is the same size.*

BINDER Question 136

Every column adds up on the subnet table sent to a district office in Tarime. The sizes and masks are right, and each of the four departments has its block. Two of those blocks begin where their own mask rounds them onto a neighbour.

State the two rules that a variable-length allocation must obey, and explain what goes wrong when either of them is broken.

Model Answer

The rules are, first, allocate the largest requirement first and work downwards to the smallest, and second, begin every block at an address that is a whole multiple of that block's own size; break the second and the mask rounds the address down onto a block already issued, so two subnets overlap, and break the first and the plan leaves holes that the larger blocks arriving later cannot use.

Rule two, its own boundary: a block of 32 addresses may begin only at 0, 32, 64 or 96 and upwards; a block of 64 only at 0, 64, 128 or 192. The test uses that block's own size and never a neighbour's, so the boundaries in one plan are of several different spacings at once.

$$\text{Block start} \bmod 2^{32-\text{p}} = 0$$

What a broken boundary does: write a /27 at 192.168.10.80 and the mask will not honour it. Since 80 is not a multiple of 32, applying 255.255.255.224 yields the network 192.168.10.64, so whatever was issued at .64 is claimed a second time and both groups of hosts compute the same network and broadcast addresses.

What a broken order does: allocate a small block first and the pointer stops at an address no larger block may legally use. The next block jumps forward to its own boundary, leaving a hole beneath it, and the largest requirement, which has the fewest legal starting places of all, arrives last to a block already cut about.

Why the two are one: place blocks in order of decreasing size and every later size divides the size just placed, so a pointer resting on a multiple of the larger already rests on a multiple of the smaller. Obeying the first rule makes the second look after itself, and the allocation comes out contiguous.

Both failures are silent on paper. An overlapping plan looks tidy in a table and announces itself only when two departments find they cannot reach one another.

The Master Key: *Largest first, and every block on a multiple of its own size, in that order and no other.*

BINDER Question 137

One /24, and four requirements that differ by a factor of twenty-five: fifty machines in administration, twenty-five in the library, ten in the laboratory and two router interfaces on the fibre to the district office. A school at Sengerema drew the plan in an afternoon.

Using Figure 3.23 and Table 3.20, describe how each of the four allocations was arrived at, taking the requirements in the order the method demands.

Model Answer

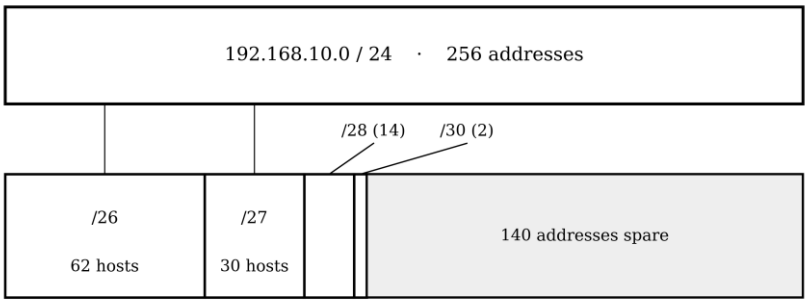
Sorted largest first, the four requirements take a /26, a /27, a /28 and a /30, and each begins where the last one ended: administration at 192.168.10.0, the library at .64, the laboratory at .96 and the router link at .112, consuming 116 of the 256 addresses between them.

Administration, fifty hosts: five host bits give $2^5 - 2 = 30$ and fail; six give $2^6 - 2 = 62$ and fit. The prefix is /26 and the block size 64. Being the largest requirement it is placed first, at the foot of the block, running from 192.168.10.0 to .63 with $62 - 50 = 12$ addresses spare.

Library, twenty-five hosts: four host bits give $2^4 - 2 = 14$, too few; five give $2^5 - 2 = 30$, which fits. The prefix is /27 and the block size 32. The pointer stands at 64, and 64 is a multiple of 32, so the library takes .64 to .95 and keeps $30 - 25 = 5$ addresses spare.

Laboratory, ten hosts: three host bits give $2^3 - 2 = 6$ and fail; four give $2^4 - 2 = 14$ and fit, so this is a /28 of 16 addresses. The pointer stands at 96, a multiple of 16, and the laboratory takes .96 to .111 with $14 - 10 = 4$ addresses spare.

The router link, two hosts: two host bits give $2^2 - 2 = 2$, exactly the pair of interfaces the link needs, so the prefix is /30 and the block size 4. The pointer stands at 112, a multiple of 4, and the link takes .112 to .115 with nothing spare at all.




```
Administration   50 needed   192.168.10.0/26    62 usable   12 spare
Library          25 needed   192.168.10.64/27   30 usable    5 spare
Laboratory       10 needed   192.168.10.96/28   14 usable    4 spare
Router link       2 needed   192.168.10.112/30   2 usable    0 spare
```

Four equal /26 subnets would have consumed all 256 addresses and still handed the router link 60 it can never use. VLSM takes 116 and keeps 140 for growth, which is what makes an unequal plan worth the extra arithmetic.

Figure 3.23: *A variable-length allocation from one /24. Equal subnets would have consumed the whole block; this plan uses 116 addresses and leaves 140 for growth.*

What the block has left: the four blocks take 64 + 32 + 16 + 4 = 116 addresses, so 256 − 116 = 140 remain in one unbroken run from 192.168.10.116 to .255. Four equal /26 subnets would have taken all 256 and left the school nothing whatever to grow into.

Department	Hosts needed	Network	Prefix	Usable	Spare
Administration	50	192.168.10.0	/26	62	12
Library	25	192.168.10.64	/27	30	5
Laboratory	10	192.168.10.96	/28	14	4
Router link	2	192.168.10.112	/30	2	0

Table 3.20: *The variable-length allocation, department by department.*

Read the plan in the order it was built and every line explains itself. Read it downwards from a finished table and it is only four rows of numbers to be memorised.

The Master Key: *Sort, then size, then place, letting the address at which one block ends choose where the next begins.*

BINDER Question 138

Between two routers on a fibre pair there is nothing to address but the two ends. No host will ever join that wire, no printer and no access point. A technician at Chake Chake gave the link a /26, the mask every other segment carried.

Justify the use of a /30 on a point-to-point link between two routers, and calculate how many addresses such a link consumes and how many of them it wastes.

Model Answer

A point-to-point link needs exactly two addresses, one for each router interface, and the smallest block supplying two usable addresses is a /30: it leaves 32 − 30 = 2 host bits, so $2^2 - 2 = 2$ addresses are usable out of the 4 the block contains, and the remaining 4 − 2 = 2 go to the network address and the broadcast address.

What the link needs: two interfaces face each other across the link and nothing else attaches to it. Every frame entering at one end leaves at the other, so there is no third party to address, and an address beyond the second would be unusable by construction rather than by choice.

Solving the requirement: set $2^h - 2$ to be at least 2 and take the smallest h that satisfies it. One host bit gives $2^1 - 2 = 0$ and supplies nothing; two give $2^2 - 2 = 2$ and supply exactly the pair wanted. So h = 2, the prefix is 32 − 2 = /30 and the mask is 255.255.255.252.

The waste inside the block: a /30 spends 4 addresses to deliver 2, so 2 are lost, which is 50 per cent of the block. No prefix loses a larger share, and the pair lost is the same pair every subnet surrenders, merely paid on the smallest block there is.

The waste avoided: a /26 on the same link delivers 62 usable addresses for two interfaces, stranding 62 − 2 = 60 on a wire nothing can join. Five such links at /26 would demand 5 × 64 = 320 addresses, more than a /24 holds, while five /30 links demand 5 × 4 = 20.

Where the links come from: every pair of routers joined directly needs one, so three routers in a triangle need three. Allocate them last, after the host subnets, where their small size makes them easy to place on a boundary.

A later standard permits a /31 on point-to-point links alone, treating both of its addresses as interfaces, but /30 is the answer expected of you and the one configured on most equipment.

The Master Key: *Two interfaces, two addresses, four consumed: the smallest subnet is also the dearest one per host.*

REAL Question 139

Nothing may be configured at a secondary school in Lindi until the ranges are settled: fifty machines in administration, twenty-five in the library, ten in the computer room and five in the gate office, all out of one private /24 and nothing more.

Draw up the complete variable-length plan for the school, giving for every subnet its prefix, its mask, its network address, its usable host range and its broadcast address.

Block available: 192.168.20.0/24

Administration: 50 hosts

Library: 25 hosts

Computer room: 10 hosts

Gate office: 5 hosts

Model Answer

Taken largest first, administration receives 192.168.20.0/26, the library 192.168.20.64/27, the computer room 192.168.20.96/28 and the gate office 192.168.20.112/29, which together consume 64 + 32 + 16 + 8 = 120 addresses and leave 136 free from 192.168.20.120 upwards.

Sort before allocating: write the requirements in descending order, 50, 25, 10 and 5, and turn each into the smallest prefix that holds it before any address is chosen. Sizing and placing are separate steps, and running them together is what puts a block on a boundary its own mask cannot express. Nothing here depends on the class of the address; the /24 given is where the working starts and the prefix is read from the requirement alone.

Administration: fifty hosts need $2^6 - 2 = 62$, since $2^5 - 2 = 30$ falls short, so the prefix is /26 and the mask 255.255.255.192. The network address is 192.168.20.0, the usable range runs from 192.168.20.1 to 192.168.20.62, and the broadcast address is 192.168.20.63.

Library: twenty-five hosts need $2^5 - 2 = 30$, since $2^4 - 2 = 14$ falls short, so this is a /27, mask 255.255.255.224, block size 32. It begins at 192.168.20.64, which is a multiple of 32; hosts run from .65 to .94 and the broadcast address is .95.

Computer room: ten hosts need $2^4 - 2 = 14$, since $2^3 - 2 = 6$ falls short, so a /28, mask 255.255.255.240, block size 16. It begins at 192.168.20.96, a multiple of 16; hosts run from .97 to .110 and the broadcast address is .111.

Gate office: five hosts need $2^3 - 2 = 6$, since $2^2 - 2 = 2$ falls short, so a /29, mask 255.255.255.248, block size 8. It begins at 192.168.20.112, a multiple of 8; hosts run from .113 to .118 and the broadcast address is .119.

Two checks before signing: every network address must divide by its own block size, and 0, 64, 96 and 112 divide by 64, 32, 16 and 8 in turn. Every broadcast address must sit one below the next network address, and 63, 95 and 111 do. The plan then takes 120 addresses and leaves 256 − 120 = 136.

The run left above the plan: the last block ends at 192.168.20.119, so everything from .120 to .255 is unissued, 136 addresses in one piece. That run will take a /29 at .120 and a /25 of $2^7 - 2 = 126$ hosts at .128, which is more room than the school has yet asked for, and it will stay in one piece for as long as later requirements are taken by the same rule.

Record where the plan is tight rather than where it is generous: the gate office holds 6 − 5 = 1 address in hand, and the free run above .119 is where its replacement block would come from.

The Master Key: *Turn every requirement into a prefix first, lay the prefixes down in one pass, and let the two checks close the plan.*

REAL Question 140

Four lines leave the router at a district council office in Magu, one for each network behind it, and every router beyond keeps four entries where one would serve. A note taped to the rack forbids anything shorter than a /24 against an address beginning 192.

Determine the single summary route that advertises all four of the networks listed, showing the working in binary, and say whether the note on the rack is correct.

Networks behind the district router, each a /24: 192.168.4.0, 192.168.5.0, 192.168.6.0 and 192.168.7.0

Required: one advertisement covering all four of them and nothing besides

Model Answer

The four agree in their first 22 bits, so the summary route is 192.168.4.0/22 with mask 255.255.252.0, covering the 2^{10} = 1,024 addresses from 192.168.4.0 to 192.168.7.255, which is exactly the 4 × 256 = 1,024 the four networks hold between them; the note is wrong.

The first line: 192.168.4.0 is 11000000.10101000.00000100.00000000. Keep the stops between the octets in place, so that the four lines can afterwards be read as columns rather than as strings.

The second line: 192.168.5.0 is 11000000.10101000.00000101.00000000, written directly beneath the first, and only the third octet has changed.

The third line: 192.168.6.0 is 11000000.10101000.00000110.00000000, beneath the second and aligned with it column for column.

The fourth line: 192.168.7.0 is 11000000.10101000.00000111.00000000, the last of the four and the highest of them.

The bits they hold in common: read down the four lines from the left and stop at the first column in which they disagree. The opening two octets are identical throughout, which is 16 bits. In the third octet every line begins 000001, six bits more; the next column reads 0, 0, 1 and 1, so agreement ends there.

The prefix from the count: count what agreed: 16 + 6 = 22 bits, so the summary prefix is /22, and a mask of 22 ones followed by ten zeros is 255.255.252.0. Set every bit after the twenty-second to zero and the address is 192.168.4.0, giving the single advertisement 192.168.4.0/22.

Proving it covers no more: a summary that is one bit too short attracts traffic for networks the office does not hold. This one spans 1,024 addresses against the 1,024 the four contain, so nothing is over-claimed, and it begins legally, since /22 blocks start at third-octet values 0, 4, 8 and 12, and the third octet here is 4.

What the note has wrong: a leading octet of 192 once implied a /24 by convention, but the prefix now travels beside the address in every routing message, so the boundary may fall at any bit. That is what makes a /22 across these four ordinary, and it is the same freedom by which a /24 may be written on 11.0.0.0.

Four routes become one, which is three entries fewer here and many thousands fewer across a provider's network, and the only price is that the summary must be recalculated whenever a network is added or removed at either end.

The Master Key: *Stack the addresses in binary, count the columns that agree all the way down, and that count is the prefix.*

HOT Question 141

An adviser at a school in Kyela argues that four different masks in one plan are three more than a technician can be expected to remember, and that the school should have cut four equal blocks instead. The argument is reasonable.

Assess whether a variable-length plan is worth its extra complexity for a school with four departments, and commit to a recommendation.

Model Answer

It is worth it and the school should keep the variable plan, because its complexity is one page of arithmetic done once, while the cost of equal subnets is permanent: four equal /26 blocks exhaust a /24 entirely and leave the school four networks and no fifth, whereas the variable plan meets the same four requirements and keeps 140 addresses, enough for a further /25 of $2^7 - 2 = 126$ hosts.

The complexity, stated fairly: four masks must be recorded instead of one, every subnet has a different block size, and nobody can read an address and infer its range without consulting the plan. A mistyped prefix then yields a machine that reaches part of the network and not the rest, which presents as an intermittent fault rather than a plain one.

What equal subnets cost here: four /26 blocks consume $4 \times 64 = 256$ addresses, the whole /24. There is no fifth network for a wireless segment, a server or a second building, and the router link sits on 62 usable addresses to hold two interfaces. The block is not full of machines; it is full of reservations.

What the variable plan returns: the same four requirements take 116 addresses, and what remains is a single unbroken run of 140. That run will accept a /25, or two /26 blocks, or any smaller mixture, without disturbing one address already issued or one machine already configured.

Where the complexity is paid: once, by whoever writes the plan, and then only if the plan is written down. Four rows giving department, network, prefix and range answer the objection outright, and they are the same four rows an examiner expects to see under a question of this kind.

When the answer reverses: were the requirements 50, 45, 40 and 40, one /26 would fit all four and the variable plan would return nothing but masks to remember. The test is the ratio of the largest requirement to the smallest, and a two-address router link carries any list containing one well past it.

The verdict: keep the variable plan, publish it as one table beside the switch, and require every change to be entered on that table. Adopt equal subnets only where the requirements lie within a factor of two of one another and no point-to-point link appears in the list.

Simplicity is worth paying for, but the price demanded here is the whole of the remaining block, and a school holding one /24 does not have that to spend.

The Master Key: *Price the simplicity of one mask in addresses before you buy it, and let the smallest requirement set the price.*

HOT Question 142

A trainee at a district council in Nzega allocated the four subnets in the order the departments asked, smallest first, packing each onto the end of the one before. The arithmetic is correct and the sizes adequate. The plan cannot be configured.

Diagnose what is wrong with the allocation below, and explain why a variable-length plan must take the largest requirement first.

1. Router link, 2 hosts: 192.168.10.0/30

2. Laboratory, 10 hosts: 192.168.10.4/28

3. Library, 25 hosts: 192.168.10.20/27

4. Administration, 50 hosts: 192.168.10.52/26

Model Answer

Only the first line is a legal network address: 4 is not a multiple of 16, 20 is not a multiple of 32 and 52 is not a multiple of 64, so each of the last three masks rounds its address down onto 192.168.10.0 and all

four blocks end up naming the same network; allocating the largest requirement first prevents it, because a pointer resting on a multiple of one block size rests automatically on a multiple of every smaller one.

Test each start: divide every network address by its own block size and demand no remainder. The /30 at 0 passes. The /28 at 4 fails, since its multiples are 0 and 16; the /27 at 20 fails, since its multiples are 0 and 32; and the /26 at 52 fails, since its multiples are 0 and 64.

What the mask does with 52: a mask keeps the network bits and clears the rest, so 255.255.255.192 applied to 192.168.10.52 yields 192.168.10.0. The line intended a block from .52 to .115; what it actually names is 192.168.10.0/26, from .0 to .63, which already contains the router link, the laboratory and most of the library.

How the fault presents: a machine configured 192.168.10.53/26 believes its own subnet runs from .0 to .63, so it treats the router link addresses as neighbours on its own wire, resolves them there instead of handing them to the gateway, and gets no reply. The complaint reaching the technician is that some addresses answer and others do not.

The same requirements, ordered: largest first gives the /26 at .0 to .63, the /27 at .64 to .95, the /28 at .96 to .111 and the /30 at .112 to .115. Every start divides by its own size, no block touches another, and the four occupy one contiguous run of 116 addresses ending at .115.

Why the order does the work: place blocks in decreasing size and each new size divides the size just placed, so the pointer left by one block is already a legal start for the next and no alignment step is ever required. Ascending order destroys that: each block must jump forward to its own boundary, and every jump is a hole.

Even the legal version loses: align the smallest-first plan properly and it still costs. The /30 ends at .3, but the /28 cannot open until .16, so $16 - 4 = 12$ addresses are stranded beneath it, the last block now ends at .127 rather than .115, and the stranded run will hold nothing larger than a /29.

Conclude that the trainee's mistake was not arithmetic but sequence, and that sequence is the one part of this method that checking each line on its own can never recover.

The Master Key: *Allocate downwards in size and the boundaries look after themselves; allocate upwards and every one of them must be forced.*

HOT Question 143

The office at Ilala has been unreachable since eight in the morning, and the route to it, seen from the provider's router, is up. It is up because three other networks behind the same summary are up. The summary never spoke for the fourth.

Judge whether route summarisation helps or hinders the diagnosis of faults, and defend the judgement.

Model Answer

On balance it helps and should be used, but only where the advertising router owns every network inside the summary and the specific routes stay visible within the region: summarisation conceals the state of any single component, so one failed /24 inside a /22 leaves a quarter of the summarised space silently unreachable, and what is bought in exchange is a table that no longer changes every time a distant link flaps.

What a summary conceals: one advertisement stands for several networks and is withdrawn only when the last of them goes. A neighbour cannot therefore distinguish three working networks from four, and the route it holds is a promise about a block of addresses rather than a report on any member of that block.

The black hole it creates: traffic for the failed network reaches the summarising router, which holds no specific route for it and discards it. Seen from outside, the path is healthy to that point, so the first hour of the diagnosis goes on proving that the fault does not lie in the provider's network.

What is bought in exchange: a region holding 64 contiguous networks of /24 can advertise one /18 instead, removing $64 - 1 = 63$ entries from every table beyond the boundary. Those routers also stop hearing about a link inside the region that keeps failing and recovering, so their tables stay still and reconvergence after a genuine failure is quicker.

Where diagnosis is actually done: inside the region nothing is hidden. The summarising router still holds the four specific routes and can be asked which of them has gone. Summarisation moves the detail rather than destroying it, and a technician who knows where the boundary lies knows which router to question first.

The condition that must hold: summarise only contiguous blocks you own entirely. Advertise 192.168.4.0/22 while another organisation holds 192.168.6.0/24 and you have offered to carry traffic that is not yours, a fault considerably worse than a slow diagnosis and one that somebody else will report before you notice it.

The judgement: summarise at the regional boundary, keep the specific routes inside it, and monitor each component network at the router doing the summarising, so a failure is caught where the detail still exists. Never let a monitoring system read a summary as evidence that the networks under it are reachable.

The office at Ilala was down for four hours, three of them spent trusting a route that had never claimed to describe it.

The Master Key: *Summarise only what you wholly own, and watch the components at the very router that hides them.*

IPv6: notation, compression, the header and coexistence

IPv4 ran out of addresses, and IPv6 is the replacement drawn up so that it can never happen again. This section settles four things: how a 128-bit address is written, how it is shortened and lengthened again without changing what it names, what the fixed 40-byte header carries, and how a network holding both protocols runs them side by side. It assumes you can already read a prefix such as /24. The arithmetic here is counting groups of sixteen bits, not borrowing bits from a host part.

BINDER Question 144

Eight groups of hexadecimal digits, separated by colons: the management address on the delivery note of a router bound for a training centre at Bagamoyo. The technician copies it twice and produces two different strings, and nothing in it resembles four decimal numbers.

State the length of an IPv6 address in bits, and describe how such an address is written down.

Model Answer

An IPv6 address is 128 bits long, four times the length of an IPv4 address, and it is written as eight groups of four hexadecimal digits separated by colons, each group standing for 16 bits, so that 8 × 16 = 128 accounts for every bit of the address.

Why 128 bits: the address is one binary number of 128 bits, and 2^{128} is about 3.4×10^{38} addresses, against 2^{32}, about 4.3 billion, for IPv4. The length was not chosen to be comfortable for a while. It was chosen to put exhaustion beyond reach.

Four bits to a digit: hexadecimal counts 0 to 9 and then a to f, which is 10 + 6 = 16 symbols, and $2^4 = 16$, so one digit carries exactly 4 bits and four digits carry 4 × 4 = 16 bits. Case is not significant, so ff and FF are one group.

Eight groups, seven colons: the groups run from the most significant to the least with a colon between each neighbouring pair, so a full address carries 8 − 1 = 7 colons. Only the colon separates, and it separates groups of sixteen bits rather than octets.

Where the prefix ends: a stroke and a number after the address state how many leading bits belong to the network, exactly as in IPv4. Almost every subnet is written /64, leaving 128 − 64 = 64 bits to identify the interface, so an IPv6 subnet is not sized to fit the hosts on it.

What has not changed: the address still names an interface rather than a machine, still splits into a network part and an interface part, and is still read from the left. Table 3.21 sets the two protocols side by side.

Feature	IPv4	IPv6
Address length	32 bits	128 bits
Notation	Dotted decimal, four octets	Hexadecimal, eight groups of 16 bits
Header size	20 bytes minimum, variable	40 bytes, never varies
Broadcast	Directed and limited broadcast	None; multicast does the work
Fragmentation	Source or any router on the path	Source only
Configuration	Manual or DHCP	Autoconfiguration or DHCPv6
Address space	2^{32}, about 4.3 billion	2^{128}, about 3.4×10^{38}

Table 3.21: *IPv4 and IPv6 compared.*

Two people copying one address must arrive at one string, and that follows from reading the groups rather than the characters.

The Master Key: *Count in groups of sixteen bits, never in characters, and an IPv6 address stops looking long.*

BINDER Question 145

Two students in a study group at Geita shorten the same address and produce different strings. One struck zeros from the end of a group instead of the front; the other left a group as an empty space between two colons already there.

Compress 2001:0db8:0000:0000:0000:ff00:0042:8329 to its shortest legal form, naming the rule you apply at each step, and then expand 2001:db8:acad::42 back to its eight full groups.

Model Answer

Dropping leading zeros gives 2001:db8:0:0:0:ff00:42:8329, and replacing the longest run of all-zero groups with a double colon gives 2001:db8::ff00:42:8329; the same rules run backwards on 2001:db8:acad::42 restore the zero groups it hides and pad each group to four digits, giving 2001:0db8:acad:0000:0000:0000:0000:0042.

Rule one, leading zeros: within a group, zeros standing in front of the first non-zero digit may be struck out, so 0db8 becomes db8 and 0042 becomes 42. But ff00 keeps both of its zeros, which are trailing, and 0000 becomes a single 0, because one digit must remain.

Rule two, the double colon: one run of consecutive all-zero groups may be replaced by a double colon. Here it is the third, fourth and fifth groups, so 2001:db8:0:0:0:ff00:42:8329 becomes 2001:db8::ff00:42:8329. Where two runs are equally long, compress the leftmost; where a run is one group, write the 0.

What may never be dropped: a trailing zero is significant, so ff00 may not be written ff. Struck out it leaves ff0, which pads back to 0ff0, and 0xff00 divided by 0x0ff0 is 16, so the group has been divided by sixteen and the address now names another machine.

Expansion is counting: to go the other way, count the groups written. In 2001:db8:acad::42 there are four, so the double colon stands for $8 - 4 = 4$ all-zero groups. Write those four groups of 0000 in its place, then pad every remaining group on the left to four digits.

Checking either direction: eight groups of 16 bits give $8 \times 16 = 128$ bits, so any other count is an error worth finding now. Then compress your own expansion and see whether it returns the string you were given.

Compression removes writing and never removes address, and a candidate who travels in both directions at will holds the whole of IPv6 notation.

The Master Key: *Leading zeros first, the longest run of zero groups second, and expand by counting the groups you can still see.*

BINDER Question 146

The interface report from a router at a district office in Musoma lists three addresses where an IPv4 router would list one. The first begins 2001, the second fe80, the third ff02. None of the three is a mistake, and each answers a different question.

Name the three kinds of IPv6 address, state what each delivers to, and say what IPv6 provides in place of the IPv4 broadcast.

Model Answer

IPv6 has unicast, delivered to one interface, multicast, which occupies ff00::/8 and reaches every interface that has joined the group named, and anycast, which reaches whichever of several interfaces sharing the address is nearest; there is no broadcast at all, and its work is done by multicast groups such as ff02::1.

Unicast and its two scopes: a unicast address names one interface and one delivery. A global unicast address is routable across the Internet, while a link-local address from fe80::/10 is configured automatically on every interface and never forwarded off its own link.

Multicast begins ff: every address in ff00::/8 is multicast, so a first group opening with ff identifies one at sight. It reaches each interface that has joined that group and no other. Two are worth holding: ff02::1 for all nodes on the link and ff02::2 for all routers.

Anycast, one of many: the same address is configured on several interfaces in different places, and routing carries the packet to the nearest by the metric in use. It suits a service offered identically in many locations, and the sender never learns which copy answered.

Why broadcast was removed: an IPv4 broadcast interrupts every host, and the processor is disturbed before the packet can be discarded. A host that has not joined a multicast group discards the frame in its adapter, so the cost of reaching many falls only on the machines concerned.

The /64 convention: a link is almost always given a /64, and 64 + 64 = 128, so the upper half addresses the subnet and the lower half the interface. That leaves 2^{64} identifiers on one link, and it lets a host build its own address rather than ask a server.

Three types and no fourth, and knowing which you hold from the opening group alone is most of what a router does with a destination field.

The Master Key: *Read the opening group before anything else, because ff means many, fe80 means only here, and anything else means one.*

BINDER Question 147

At a demonstration in Mikocheni a supplier claims his new box forwards IPv6 faster than IPv4, and a teacher objects that the IPv6 header is twice the size. The supplier agrees that it is, and repeats the claim. Nobody there can say why.

Using Figure 3.24, name the eight fields of the IPv6 header in order and show by adding their widths that the header comes to exactly 40 bytes.

Model Answer

The eight fields are version, traffic class, flow label, payload length, next header, hop limit, source address and destination address, of widths 4, 8, 20, 16, 8, 8, 128 and 128 bits; 4+8+20 = 32 bits fill the first row, 16+8+8 = 32 the second, 128+128 = 256 the eight rows of addresses, and 32+32+256 = 320 bits, which at eight bits to the byte is 40 bytes.

The first row: version holds 4 bits and carries the value 6. Traffic class holds 8 bits and marks the packet for priority handling. Flow label holds 20 bits and tags a stream of packets a router should treat alike without inspecting each one. Together they are 4+8+20 = 32 bits, one full row.

The second row: payload length holds 16 bits and counts the bytes that follow the header, not including it. Next header holds 8 bits and names what comes next, either an extension header or a transport protocol. Hop limit holds 8 bits. Again 16+8+8 = 32 bits.

Hop limit renamed: hop limit is what IPv4 calls time to live. Each router reduces it by one and discards the packet at zero, which is what stops a packet circling for ever in a routing loop. The older name described the field badly, because nothing about the count measures time.

Ten 32-bit rows, which is 40 bytes, and unlike IPv4 the length never varies. Hop limit is what IPv4 calls time-to-live.

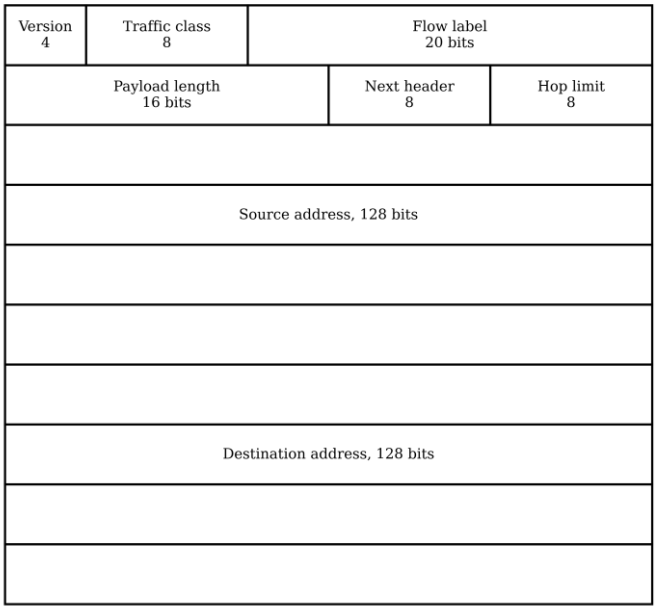


Figure 3.24: *The IPv6 header. Ten 32-bit rows, which is 40 bytes, and unlike IPv4 the length never varies.*

The two addresses: source and destination hold 128 bits each, so 128+128 = 256 bits, which is 256 ÷ 8 = 32 of the 40 bytes, and 128 ÷ 32 = 4 rows of the diagram are given to each. The addresses are the whole reason the header grew.

Why the length never varies: there is no header-length field because there is nothing to measure, no checksum because the layers above and below already check, and no options. Anything optional travels in an extension header placed after the fixed 40 bytes and named by the next header field.

Ten rows of 32 bits is 10 × 32 = 320 bits, and quoting 40 bytes with that working behind it is worth more than quoting it alone.

The Master Key: *Add the eight widths once, in order, and 40 bytes becomes a figure you can rebuild instead of one you must recall.*

REAL Question 148

A contractor commissioning a link for a district council at Morogoro types a destination address; the router rejects it without explanation. Every character is a legal hexadecimal digit and the length is right. The line reads as four different addresses, each 128 bits.

Explain why the double colon may appear only once in an IPv6 address, using 2001::25de::cade to show exactly what goes wrong when it appears twice.

Router entry: destination 2001::25de::cade, next hop over the new link

The router rejects the line and no route is added

Every group written in the address is a legal hexadecimal value

Model Answer

The double colon is not a separator but an instruction to insert as many all-zero groups as are needed to reach eight, and that instruction can be obeyed only if there is one place to put them; with two, the reader knows 8 − 3 = 5 groups are missing but not how they divide between the gaps, so 2001::25de::cade has four readings, each of eight groups and each of 128 bits.

What the symbol means: the double colon does not stand for a fixed number of groups. It stands for however many all-zero groups are needed, and the reader recovers that number by counting the groups written and subtracting from eight. The count is always recoverable; the position is recoverable only because there is a single gap.

Counting this address: three groups are written, 2001, 25de and cade, and eight are required, so 8 − 3 = 5 all-zero groups are hidden. One double colon would put all five in one place and settle the address. Two gaps must share the five, and nothing in the notation says how.

The four readings: the five may divide as one and four, giving 2001:0000:25de:0000:0000:0000:0000:cade; as two and three, giving 2001:0000:0000:25de:0000:0000:0000:cade; as three and two, giving 2001:0000:0000:0000:25de:0000:0000:cade; or as four and one, giving 2001:0000:0000:0000:0000:25de:0000:cade. Five groups across two gaps allow 5 − 1 = 4 splittings, and every one holds eight groups.

Why length is no defence: each of the four is a lawful 128-bit address, so a rule requiring only that the total reach 128 bits would admit all of them. The length was never in doubt. Where the zeros sit is, and they sit inside the network part, so the four point into four different networks.

The rule and its reach: one double colon at most, and it may stand at the beginning, in the middle or at the end. So ::1 is lawful, being 8 − 1 = 7 zero groups and then 1, and 2001:db8:: is lawful, being two groups and then 8 − 2 = 6 of zeros. Each has one gap and therefore one reading.

What the router did: equipment does not choose between candidates, because a guess would send traffic silently to a network nobody selected. The contractor must obtain the address in full and compress it once, correctly, before typing it again.

The router gave the only honest response available to it, which was that the operator had written down more than one place at the same time.

The Master Key: *Ask of any shortened address whether it can be lengthened in exactly one way, because that single test is the whole of the rule.*

REAL Question 149

A university in Stone Town replaces its core switches and routers, and the invoice describes every one of them as IPv6-ready. The provider selling the campus its Internet capacity issues one public IPv4 address and has no IPv6 service at any price.

Describe what running a dual stack means, distinguish it from tunnelling, and advise the university what to deploy now.

Every switch and router bought this year supports IPv4 and IPv6

The provider issues one public IPv4 address and no IPv6 prefix

The campus wants to be ready on the day IPv6 is offered, without buying again

Model Answer

A dual stack means every host and router runs both protocols at once, holding an IPv4 and an IPv6 address on the same interface and choosing between them for each destination, while tunnelling wraps an IPv6 packet inside an IPv4 packet to cross a network speaking only IPv4; deploy the dual stack across the campus now, keep IPv4 as the path outward, and hold the provider to a dated commitment rather than build a tunnel.

What a dual stack is: the two protocols run side by side and neither translates into the other. An interface holds an address of each kind, the name service answers with an IPv6 address where one exists and an IPv4 address otherwise, and the host uses whichever it was given.

What it costs to run: two address plans, two sets of routes and two sets of filters, because a firewall rule written for IPv4 protects nothing on the IPv6 side of the same interface. A campus that enables IPv6 and forgets its filters has opened a second door into every server, and nobody is watching it.

What tunnelling does instead: the tunnel endpoint places the whole IPv6 packet inside an IPv4 packet addressed to the far endpoint, which unwraps it, so the provider carries ordinary IPv4 and never sees the passenger. The outer header costs 20 bytes, so a path of 1500 bytes carries 1500 − 20 = 1480 bytes of IPv6 packet.

Why not a tunnel here: a tunnel earns its place when something must be reached over IPv6 before the provider is ready, and nothing here must be. It adds an endpoint to operate, a reduced packet size to explain whenever a transfer misbehaves, and a dependence on a third party.

What to deploy now: bring IPv6 up inside the campus, plan the addressing as though the prefix had arrived, and write both sets of filters in one sitting. A site prefix of /48 leaves 64 − 48 = 16 bits of subnetting, which is 65,536 subnets of /64, so the plan can be drawn now and the digits filled in later.

The clause for the contract: require the provider to state when native IPv6 will be offered and at what prefix length. Equipment described as IPv6-ready is worth nothing until a prefix arrives, so what the committee bought is readiness, and readiness becomes reach only when somebody else acts.

The switches were worth buying and the invoice was honest. What the money purchased is the ability to move on the day the provider moves.

The Master Key: *Turn both protocols on before you need the second one, because the costly part of this transition was never the hardware.*

REAL Question 150

An intern finishing a placement at an internet provider's office in Kibaha leaves a page of notes for the student who replaces her. Most of it is accurate and useful. The three statements below are not, and each reads IPv6 through IPv4.

Identify the fault in each of the three statements, give the correct version, and explain in each case why the correction holds.

1. To reach every host on a link in IPv6, send to the broadcast address, exactly as in IPv4.

2. The IPv6 header is larger than the IPv4 header because it carries more options.

3. The double colon may be used twice in one address, provided the result still comes to 128 bits.

Model Answer

IPv6 has no broadcast address of any kind, and anything meant for every host on a link goes to the all-nodes multicast group ff02::1; the header is larger only because its two addresses occupy 32 of its 40 bytes, and it carries no options at all, having fewer fields than IPv4 rather than more; and the double colon may appear once only, because arriving at 128 bits says nothing about where the hidden zero groups sit.

Statement one, a facility withdrawn: broadcast was taken out of IPv6 deliberately. Anything that must reach every host on a link is addressed to ff02::1, which every IPv6 interface joins. The difference is felt at the receiver: a host that has not joined a group rejects the frame in its adapter, whereas an IPv4 broadcast had to reach the processor before it could be discarded.

Statement two, larger and simpler: the header is 40 bytes against a minimum of 20 for IPv4, and the addresses alone account for 128 + 128 = 256 bits, which is 32 bytes. That leaves 40 − 32 = 8 bytes for the other 8 − 2 = 6 fields, where IPv4 spends 20 − 8 = 12 bytes on everything besides its addresses. The header grew because the addresses grew.

Statement three, a false safeguard: how many zero groups are hidden is fixed by the count of groups written, but where they sit is fixed only by there being one gap. Two gaps allow several splittings, every one of which reaches eight groups and 128 bits, so the length test the note relies on is passed by each of them while the address still names more than one destination.

What the three share: each carries an IPv4 habit across without asking whether the mechanism survived the redesign. Broadcast did not survive it, options were moved out of the header into a chain behind it, and shortened notation, which IPv4 never had, brings a rule IPv4 never needed.

Nothing in the note was careless. All three are what a competent IPv4 technician would expect to find, which is precisely why they have to be corrected in writing before the next student copies them.

The Master Key: *When a familiar mechanism is missing from a new protocol, find what replaced it before assuming it is still there under another name.*

HOT Question 151

A planning meeting at an internet provider serving Dodoma reaches the same item for the third year running. Every engineer agrees that IPv6 has to be deployed, and every engineer agrees that it cannot be this year. Nobody there is being unreasonable.

Assess why IPv6 has been adopted so slowly although IPv4 addresses ran out, and reach a judgement on what would change it.

Model Answer

Adoption is slow because IPv6 is not backward compatible, so whoever moves first pays the whole cost and gains nothing until the other end has moved too, and because address translation removed the visible symptom of exhaustion; the judgement is that argument will not shift this and price will, since deployment accelerates where holding IPv4 has become dearer than leaving it.

No compatibility between versions: an IPv6-only host cannot open a session with an IPv4-only server, because the two are different protocols in different packets. A network moving to IPv6 alone loses reach, so in practice it runs both, which is more work than running one. No gradual path exists along which each step pays for itself.

The first mover gains nothing: the value of a protocol lies in what it can reach. An organisation that deploys IPv6 finds everything it wanted was reachable over IPv4 already, so the cost falls on it and the benefit on whoever comes later. Every party sees this, and so every party waits.

Translation hid the shortage: an organisation of any size can put its machines behind private addresses and one public address, so exhaustion never arrived as a wall that stopped work. It arrived as a slowly rising price for addresses, which is easier to absorb than a redesign and easier still to defer.

The cost falls on people: the equipment mostly supports IPv6 already. What must change is the address plan, every filter, every monitoring rule, every application storing an address in a field sized for the $4 \times 3 + 3 = 15$ characters of dotted decimal, and the knowledge of every technician. None of that is bought.

Where it has already moved: mobile operators carrying millions of subscribers found large-scale translation expensive to buy and awkward to run, and they hold enough traffic to make the move pay. Providers with no IPv4 left to allocate move because there is nothing else to do.

The judgement: IPv6 will be taken up at the speed at which IPv4 becomes expensive and at no other speed. What would move it here is a provider issuing IPv6 by default to every new customer, so that a campus or a district office receives it without asking, and a public sector that requires it in tender documents. Volunteering has been tried since the specification was published and has not worked.

The engineers deferring the item are not mistaken about the technology or the priority. They are answering the question their budget asks, and it will be answered differently in the year IPv4 costs more than the move away from it.

The Master Key: *Explain slow adoption by asking who pays and who benefits, never by the merits of the protocol itself.*

HOT Question 152

After IPv6 is enabled at a teachers' college in Karatu, small transfers succeed and large ones stop. A page of text loads, a file of any size does not, and a ping of 64 bytes is answered while a ping of 1400 bytes vanishes.

Judge whether it was a good decision to forbid routers from fragmenting IPv6 packets, and defend your judgement.

Model Answer

It was a good decision, though a conditional one: moving fragmentation out of the routers and back to the source removes per-packet work from the devices least able to spare it and keeps the header at a fixed length, and it is safe because every IPv6 link must carry at least 1280 bytes; the condition is that the packet-too-big message must reach the source.

What the rule says: a router meeting a packet too large for the next link discards it and returns an ICMPv6 packet-too-big message carrying the size that link accepts. Only the source may fragment, and it does so by adding a fragment header. Collecting those messages is how a source learns the smallest size along the path.

The case from the router: fragmentation is per-packet work at the point least able to spare it, and on many platforms it drops the packet out of hardware forwarding into software. Removing it also removes identification, flags and fragment offset, which is 16 + 3 + 13 = 32 bits, or 32 ÷ 8 = 4 bytes, from the header.

The case from the destination: reassembly always happened at the destination anyway, so a router that fragments defers the cost rather than removing it, and meanwhile multiplies the damage done by loss. A datagram sent in four fragments across a path losing 1 per cent of packets fails 3.94 per cent of the time.

The case against: the mechanism depends on a message returning across networks the source does not control. An administrator who filters all ICMP produces exactly the failure at Karatu: small packets pass, large ones disappear, nothing is logged, and the symptom looks like a fault in the application. Tunnels make it common.

Why the floor matters: every IPv6 link must carry 1280 bytes, so a source that learns nothing can fall back to that size and still reach anywhere, and implementations do fall back after repeated silence. Without that floor the decision would have been reckless.

The judgement: keep the rule. The work is done once per path at the source instead of once per packet at every router, the header stays fixed, and the failure it exposes is a misconfiguration that ought to be corrected anyway. At Karatu, permit the packet-too-big message through the filter; where the filter belongs to somebody else, reduce the advertised segment size on the campus router.

A design that fails loudly when a network is misconfigured is easier to live with than one that quietly does expensive work on the operator's behalf.

The Master Key: *Judge a design by where it puts the work and what it asks in return, then check that the network is giving it.*

The transport layer: TCP, UDP and reliable delivery

IP delivers a datagram to a machine and stops there. Something must carry the last stretch, from the network card to the one program that asked for the data, and must decide whether losing a packet on the way matters. This section settles that layer: the two protocols you must be able to tell apart, the header each carries, the handshake that opens a connection, the acknowledgement that keeps a byte stream honest, and the windows that hold a sender back. It assumes you can already describe an IP datagram and what a router does with one.

BINDER Question 153

One laboratory machine at a college in Temeke runs a browser, a mail client and a backup together. Every packet for all three arrives at one address, on one cable, through one network card. Nothing in the IP header tells the three apart.

State what the transport layer provides that the network layer does not, and say why each addition is needed.

Model Answer

The network layer delivers a datagram from one machine to another and promises nothing more, so the transport layer adds four things: delivery to a process rather than to a machine, by way of port numbers; segmentation of a stream and its reassembly in order; a choice between connection and none; and, where asked for, reliability built from sequence numbers, acknowledgements, retransmission and a sliding window.

Host to host is not enough: IP carries an address for the machine and no way of naming the program inside it. A 16-bit source port and a 16-bit destination port turn delivery into a path between two processes, so all three programs share one address without confusion.

The unit of work changes: the application hands down a stream of bytes of any length. The transport layer cuts it into segments small enough to travel, numbers them and rebuilds the stream at the far end. IP does not know that the pieces belong together.

What best effort leaves undone: a datagram may be lost, delayed, duplicated or delivered out of order, and IP reports none of it. A sequence number exposes a gap or a duplicate, an acknowledgement confirms arrival, and a timer decides when to resend.

Two protocols, one layer: TCP supplies the whole apparatus and charges for it in delay and header space. UDP supplies ports and a checksum and nothing else. The layer offers a choice, and the application makes it.

Performed at the ends only: all of this work happens in the two end machines. Routers between them read the IP header and forward; they never acknowledge a segment and never notice a gap, which is why a connection survives a change of route.

Ask of any transport feature which machine performs it. If the answer is a router, it belongs to a lower layer.

The Master Key: *The network layer finds the machine; the transport layer finds the program and decides whether loss matters.*

BINDER Question 154

More than fifty bytes of frame for a single keystroke on a remote login at a laboratory in Sengerema, and the class can account for one. Twenty more are the same twenty every time, whether the segment carries one character or fifteen hundred.

Using Figure 3.25, name the fields of the TCP header and show by arithmetic that they occupy 20 bytes when no options are present.

Model Answer

The header is five rows of 32 bits: the source and destination ports at 16 bits each; the sequence number at 32 bits; the acknowledgement number at 32 bits; then the 4-bit data offset, 4 reserved bits, 8 flag bits and the 16-bit window size; and last the 16-bit checksum beside the 16-bit urgent pointer, which is 160 bits in all, and 160 divided by 8 is 20 bytes.

The two ports: 16 bits each, naming the sending process and the receiving one. Together with the two IP addresses they form the four values that distinguish one connection from every other on the machine.

Sequence and acknowledgement: both 32 bits, and both count bytes rather than segments. The sequence number is that of the first byte carried in this segment; the acknowledgement number is that of the next byte its sender expects to receive.

Offset, flags and window: the 4-bit data offset gives the header length in 32-bit words, which is how a receiver finds where the data begins once options are present. Four bits are reserved, eight carry the flags, and sixteen advertise the window.

The six flags to learn: URG marks urgent data, ACK says the acknowledgement field is meaningful, PSH asks that data be passed up at once, RST abandons a connection, SYN opens one and FIN closes one. Six names inside a field of 8 bits.

Checksum and urgent pointer: the checksum, 16 bits, covers the header, the data and a few fields borrowed from IP, so a segment delivered to the wrong place is detected. The urgent pointer, also 16 bits, means anything only when URG is set.

$$5 \times 32 \text{ bits} = 160 \text{ bits} = 20 \text{ bytes}$$

Five 32-bit rows, which is 20 bytes with no options. The eight flag bits are URG, ACK, PSH, RST, SYN and FIN, with two reserved.

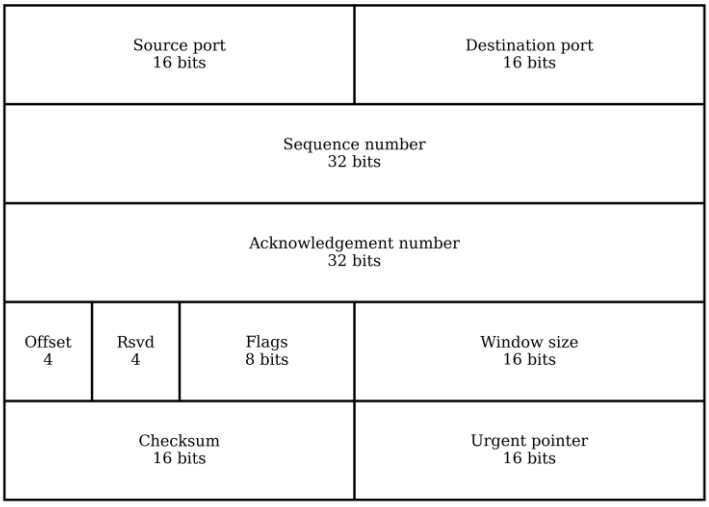


Figure 3.25: *The TCP header. Five 32-bit rows, which is 20 bytes with no options.*

Twenty bytes is the minimum and not a fixed size, since options such as a scaled window or selective acknowledgement lengthen the header, which is exactly why an offset field had to exist.

The Master Key: *Learn the header as five rows of four bytes and the arithmetic answers itself.*

BINDER Question 155

On the board at a college on Unguja the opening of a connection is drawn as four arrows: I am ready, I heard you, I am ready too, I heard you as well. The capture running beside it shows three segments and no fourth.

Using Figure 3.26, describe each segment of the three-way handshake and explain why the exchange needs three segments rather than four.

Model Answer

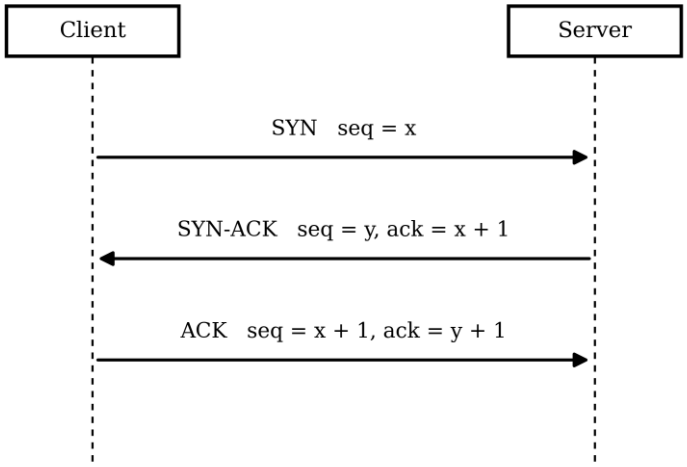
The client sends a segment with SYN set and its own starting sequence number x; the server replies with one segment carrying SYN and ACK together, its own number y and an acknowledgement of x + 1; the client answers with an ACK carrying seq = x + 1 and ack = y + 1, and three segments suffice because the server's acknowledgement and its own synchronise are separate fields of one header.

The first segment: the client chooses a starting sequence number, x, and sends a segment with SYN set and no data. The number is chosen unpredictably rather than from zero, so a stale segment of an old connection cannot be taken for a current one.

The second segment: the server sets SYN and ACK in one header. Its acknowledgement number is x + 1, saying that x is accounted for and x + 1 expected next; its sequence number is y, its own starting point. One segment does two jobs.

The third segment: the client acknowledges the server's number with ack = y + 1 and sends seq = x + 1. Both starting numbers are now agreed, both directions are open, and this last segment may carry the first data.

Why not four: nothing has been left out. The ACK and SYN bits are separate, and the header holds a sequence number and an acknowledgement number at once, so combining them costs nothing. Four counts logical events; a capture shows three segments.



Three segments, not four. The server's acknowledgement of the client and its own request to synchronise travel together in one segment.

Figure 3.26: *The three-way handshake. Three segments, not four.*

What the handshake buys: both ends have exchanged and confirmed a starting number, so every later segment can be placed exactly in the stream, and both have proved the path works in each direction before data is committed.

Closing is not the mirror of opening: each side sends a FIN of its own and has it acknowledged, so a connection may be shut in one direction while data still flows in the other.

The Master Key: *Count segments and not events whenever a protocol is set out as a numbered list of steps.*

BINDER Question 156

Two transfers crawl at a district office in Nzega. One goes to a laptop whose disk is thrashing, the other over a link every department is using. The technician reads the same figure for both, blames the provider, and is right about one.

Distinguish flow control from congestion control, and name the window that each of them uses.

Model Answer

Flow control protects the receiver from the sender: the receiver states in the window size field how many bytes of free buffer it has, and the sender may have no more than that outstanding. Congestion control protects the network from the sender: the sender keeps its own congestion window, opening it while acknowledgements return and cutting it when loss suggests a queue is filling.

Who sets the limit: the receive window is announced by the receiver in every segment it sends, so it is a fact the sender is told. The congestion window is announced by nobody, since no router will say anything; the sender infers it from the acknowledgements coming back.

Sliding, not stop and wait: a sender that waited for each acknowledgement would leave the link idle for a round trip per segment. A window allows several segments outstanding at once, and as acknowledgements arrive it slides forward so that much more may be sent.

Slow start in outline: a new connection knows nothing of the path, so it begins with a congestion window of one segment and doubles it each round trip: 1, 2, 4, 8, 16. Doubling stops at a threshold, after which the window grows by about one segment per round trip.

The smaller window wins: if the receiver advertises 64,000 bytes while congestion control permits 12,000, then 12,000 bytes may be outstanding. A transfer can be slow because the far machine is busy or because the path is full, and the two look alike from outside.

Names that date an answer: Tahoe, Reno and New Reno belong to the 1980s and 1990s and are quoted far more often than they are run. A machine bought today uses CUBIC or BBR, so treat the older names as history.

Ask which resource is scarce. A full receive buffer is one machine’s difficulty; a full router queue is everyone’s, and only the first is ever advertised.

The Master Key: *Flow control asks how much the other end can take; congestion control asks how much the path can take.*

REAL Question 157

A shipping agent in Stone Town uploads the day’s manifest to a mainland office every evening, and it takes seconds. On the evening one segment in the middle goes missing, the upload appears to stop dead, though three of the four segments have arrived.

Using Figure 3.27, trace the transfer segment by segment, stating the exact acknowledgement number the receiver returns at every step.

Model Answer

Four segments of 1,000 bytes are sent, starting at sequence numbers 1, 1001, 2001 and 3001; the receiver returns ack 1001 after the first and ack 2001 after the second, sends nothing at all for the lost third, returns ack 2001 a second time when the fourth arrives out of order, and returns ack 4001 only once the retransmitted third has arrived, because a cumulative acknowledgement names the first byte still missing and never the last byte received.

The first two segments: the segment with seq 1 carries bytes 1 to 1000, so the receiver returns ack 1001: everything below 1001 is held and 1001 is wanted next. The segment with seq 1001 carries bytes 1001 to 2000, and the acknowledgement moves on to 2001.

The loss itself: the segment with seq 2001 never arrives, and the receiver sends nothing, because no receiver can report a segment it has not seen. Only the sender’s timer, or a later arrival out of order, will reveal that anything is wrong.

An arrival out of order: the segment with seq 3001 arrives and is kept in the receive buffer, but the acknowledgement cannot move past the hole below it. The receiver therefore returns ack 2001 for a second time, and that repetition is the duplicate acknowledgement drawn in the figure.

What ack 2001 does not mean: it does not mean that 2001 arrived. It means the opposite: everything below 2001 is safely held, and 2001 is the first byte still missing. A candidate who reads an acknowledgement as the last byte received writes ack 2000 or ack 3000 here, and loses the mark.

The repair: the sender resends that one segment, either when its retransmission timer expires or, faster, as soon as three duplicate acknowledgements have reached it. The segment already sitting in the receiver's buffer is not sent again.

The acknowledgement jumps: with bytes 1 to 4000 now contiguous, the receiver returns ack 4001 in one step, acknowledging the retransmitted segment and the buffered one together. A single acknowledgement therefore clears two segments at once.

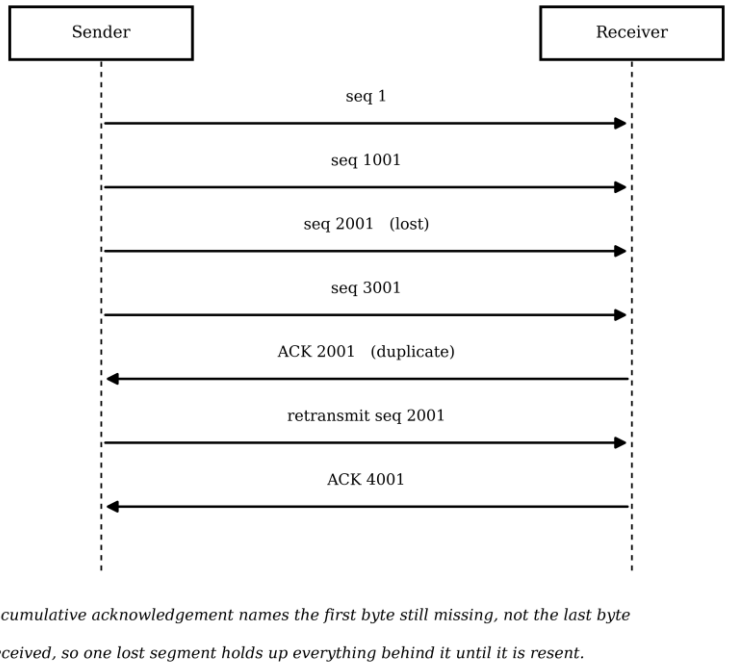


Figure 3.27: *A sliding window with one segment lost. The acknowledgement names the first byte still missing, so everything behind it waits.*

Read every acknowledgement in the figure as a request rather than a report. Each one states where the gap begins, which is why the number stands still while the data behind it piles up.

The Master Key: *An acknowledgement number is a request, not a receipt: it names the byte still wanted.*

REAL Question 158

During a video call between a district hospital in Kyela and a consultant in the city, the picture freezes and the voice breaks into fragments. On the same link, in the same minutes, a laboratory report reaches the same city complete and correct.

Explain why the call degrades while the file transfer succeeds, in terms of the two transport protocols and what each does about loss.

Model Answer

Both flows meet the same loss on the same link, and the difference lies entirely in what the protocol above does about it: the file moves over TCP, which detects every gap, retransmits it and hands up a stream that is complete and in order, paying for that in time; the call moves over UDP, which detects corruption with its 8-byte header's checksum but never asks for anything a second time, so a lost datagram is simply a missing fragment of sound or picture.

Loss is not the variable: the link discards packets from both flows at much the same rate, and nothing has singled out the call. What differs is that one protocol repairs its losses and the other has no machinery for doing so.

Why the call cannot wait: a retransmission costs at least one round trip. Sound that arrives a third of a second late cannot be played, because the moment it belongs to has gone and the receiver must discard it anyway. Repair is worthless where the data carries a deadline.

Why the file can wait: a file has no deadline of that kind. A segment resent a second later is worth exactly as much as one that arrived first time, so TCP trades time for correctness and a transfer that meets losses merely finishes later than it would have done.

The second effect: TCP reads loss as congestion and cuts its congestion window, so the transfer slows itself and survives. UDP slows down for nothing, so the call keeps pushing into a queue that is already full and adds to the very delay that is spoiling it.

What the application must supply: a call is not helpless. It hides small losses in a jitter buffer, conceals a missing fragment by repeating the last one, and lowers its own bit rate when quality falls. Those repairs sit in the application, because the transport protocol offers none.

Feature	**TCP**	**UDP**
Connection	Opened by a three-way handshake before any data	None; a datagram is sent without warning
Header	20 bytes with no options	8 bytes, always
Order of delivery	Bytes handed up in the order they were sent	Datagrams handed up in the order they arrive
Lost data	Detected from the gap and retransmitted	Not noticed by the protocol and never resent
Corrupted data	Checksum fails, so the segment is resent	Checksum fails, so the datagram is discarded
Flow control	Sliding window advertised by the receiver	None
Congestion control	Slow start and a congestion window	None; it sends at whatever rate is asked of it
Suits	Files, pages, mail, anything that must arrive whole	Calls, live video, short queries that can be repeated

Table 3.22: *TCP and UDP compared.*

The useful question is never which protocol is better but which failure the traffic can survive. A report cannot survive a missing byte; a conversation cannot survive a late one.

The Master Key: *Choose the transport protocol by what the data cannot forgive: loss, or lateness.*

REAL Question 159

Three lines from a classmate's card circulate in a study group at Tarime. They are neat, confident and quoted as settled. Each would mislead anybody who tried to reason from it about a network that was actually misbehaving.

Identify what is wrong with each of the three statements below, give the correct position with its reason, and name the habit that produces all three.

1. The three-way handshake takes four steps: the client sends SYN, the server sends ACK, the server then sends its own SYN, and the client sends the final ACK.

2. UDP is unreliable because it has no checksum, so corrupted data is passed straight up to the application.

3. TCP guarantees delivery, so a file transfer will still complete even if the fibre is cut halfway through it.

Model Answer

All three are wrong: the handshake is three segments, because the server's acknowledgement and its own synchronise are separate fields of a single header; UDP does carry a 16-bit checksum, and its unreliability lies in having no acknowledgement, no ordering and no retransmission rather than in failing to detect

corruption; and TCP guarantees that whatever is delivered is correct, complete and in order, or else that the application is told the connection has failed, which is not a promise to create a path where none exists.

Three segments, not four: the server sets SYN and ACK in the same header, so the second and third of those steps are one segment: SYN-ACK, seq = y, ack = x + 1. Four counts logical events. A capture of the exchange shows three segments, and an answer claiming four is marked wrong.

UDP does have a checksum: one of its four fields is a 16-bit checksum, and a datagram failing it is discarded. Detection is present and repair is absent. Nothing corrupted is passed up, but nothing lost is ever asked for again, and it is the second of those that makes UDP unreliable.

Guaranteed is not unconditional: with the fibre cut, retransmissions go unanswered, the timer expires again and again, and the connection is reset with an error reported to the application. The transfer fails, correctly. The guarantee is that nothing is delivered wrongly or silently, not that a broken path can be made to carry data.

The habit behind all three: each line turns a rough summary into a rule and then reasons from the rule instead of from the header. Ask what one segment can carry, which fields actually exist, and what a protocol can control, and none of the three could have been written.

Each of the three is close enough to the truth to sound learned in conversation and wrong in exactly the place where the mark sits, which is what makes a confident revision card more dangerous than an empty one.

The Master Key: *Test every confident sentence about a protocol against one segment of it.*

HOT Question 160

Forty laboratory machines at a secondary school in Makambako will send answers to a server in the next room. A supplier proposes UDP for the submissions: it is lighter, needs no handshake, and will stop forty machines overwhelming the server in the last minute.

Assess whether the examination system should be built on TCP or on UDP, and commit to a recommendation.

Model Answer

Build it on TCP without qualification for logging in, fetching questions and submitting answers, and keep UDP for the one part of the system where lateness would be worse than loss, such as a countdown clock or an invigilator's camera feed.

What this traffic cannot forgive: an answer lost in silence is a mark lost, and neither the candidate nor the invigilator would ever know. Examination traffic can forgive a second or two of delay. It cannot forgive silent loss, reordering, or a duplicate submission overwriting a later and better one.

The case for UDP as offered: it is real but small. There is no handshake, the header is 8 bytes rather than 20, and nothing retransmits during a rush. On a link inside one building, with a round trip well under a millisecond, none of those savings is measurable across a paper lasting two hours.

What building on UDP would cost: the application would have to number submissions, acknowledge them, time them out, resend them, discard duplicates and reassemble anything larger than one datagram. That is TCP, rewritten by whoever has least time to test it, and first tested during an examination.

The server load argument fails: forty connections are nothing to any server, and the handshake costs one round trip each. If the final minute is genuinely feared, save each answer as it is typed rather than all of them at the end, which flattens the peak far more than a change of protocol could.

Where UDP does earn a place: a clock ticking to every machine, or a camera feed an invigilator watches, is better carried on UDP, because a lost tick or a lost frame is replaced within a moment while a retransmitted one would arrive too late to be of any use.

The judgement: TCP for everything that carries a mark, UDP only for what is watched rather than recorded. On balance the handshake is the cheapest insurance the school will buy, and it is paid for in one round trip on a link where a round trip cannot be felt.

A system that has to produce a defensible result should not rest on a protocol that cannot tell it when something went missing.

The Master Key: *Where a lost packet becomes a lost mark, the argument for the lighter protocol is already over.*

HOT Question 161

A fishing cooperative on Mafia has no fibre and no line of sight to the mainland, so its office works over satellite. The link is respectable, yet a page that opens instantly in town takes several seconds here, and small transfers feel worst.

Judge whether the reliability TCP provides is worth its cost on a link with a round trip of nearly half a second, and defend the position you take.

Model Answer

Keep TCP, because the alternative is not a faster correct transfer but an incorrect one; recognise, though, that almost all of the cost here is round trips rather than reliability, and attack those instead by scaling the window, keeping connections open, caching locally and moving conversational traffic to UDP.

Where the half second comes from: a geostationary satellite sits about 36,000 km up, so a signal climbs and descends 72,000 km to reach the far ground station. At roughly 300,000 km per second that is 0.24 s each way, and 0.48 s for a round trip, before any equipment has done a thing.

What TCP spends it on: the handshake costs one round trip, so 0.48 s passes before the first byte of data may be sent, and every retransmission costs at least as much again. Slow start doubles the congestion window once per round trip, so 1, 2, 4, 8, 16 segments takes 4 round trips, which is 1.92 s of climbing.

Why short transfers suffer most: a long transfer pays those costs once and then runs at speed. A short request pays them in full and finishes before the window has properly opened, which is why a page made of many small objects can feel worse than one large file.

What reliability itself costs: very little. Acknowledgements are small and ride the return path, and the header is 20 bytes against 8. Abandoning reliability would save almost nothing and would forfeit the one property that makes a record worth sending at all.

The remedies that work: a window option large enough to hold a full round trip of data, so the sender never waits with nothing outstanding; connections held open instead of opened per request; a local cache and mail server, so most requests never cross the satellite; and voice on UDP, where a retransmission would arrive too late to play.

The judgement: TCP is worth its cost here, and most of that cost is avoidable. Replacing it would answer a delay problem by sacrificing correctness, which is the wrong trade for records that must be right.

Distance cannot be argued with, but the number of times a protocol crosses it can be. On a long link the design question is not how much to send but how often to wait.

The Master Key: *On a long path, count round trips rather than bytes, since each one costs the same half second whatever it carries.*

HOT Question 162

A wholesaler in Msimbazi copies each day's transactions to a branch office overnight. The branch reports the same thing every time: the transfer runs at full speed, pauses for a second or two, then finishes normally. One segment out of many thousands was lost.

Determine why the loss of a single segment can halt a transfer that is otherwise arriving perfectly, and assess what can be done to reduce the damage.

Model Answer

Three things stall at once: the stream must be handed up in order, so later segments sit undelivered in the receive buffer; the cumulative acknowledgement cannot advance past the gap, so the sending window stops sliding and fills; and the loss is read as congestion, so the congestion window is cut at the same instant.

Ordered delivery is the promise: TCP undertakes to hand up a byte stream and not a set of segments, so one missing byte blocks every byte behind it, however many arrived safely and however well they are stored. This is head-of-line blocking, and it follows from the guarantee rather than from any fault.

The window is anchored: a sender may not release a segment until it is acknowledged, and the acknowledgement will not move past the hole. Unacknowledged data therefore accumulates until the window is full, after which nothing new may be sent however healthy the path is.

The congestion window is cut: loss is the only congestion signal TCP has, so the event that opened the gap also reduces the congestion window sharply. The transfer resumes at a fraction of its former rate and has to climb again, which is why the pause is followed by a slow recovery.

Selective acknowledgement: the largest single improvement. With that option the receiver reports the blocks it does hold, so the sender learns that one segment is missing and resends only that. Without it a sender may repeat everything from the gap onwards and waste the capacity it is already short of.

Do not wait for the timer: three duplicate acknowledgements are taken as evidence of loss and trigger an immediate retransmission, far sooner than a timeout would. That mechanism works only while data is still flowing, which is a further argument for a window wide enough to keep it flowing.

The judgement: fit selective acknowledgement and fast retransmit and the stall shrinks to about one round trip, which is the honest floor. Beyond that, stop asking one ordered stream to carry independent objects, and give each its own stream or connection.

One lost segment is expensive because it is not treated as one lost segment. It is treated as a break in a promise about order, and every byte behind it waits for that promise to be kept.

The Master Key: *Head-of-line blocking is the price of ordered delivery, so either pay it deliberately or stop ordering the stream.*

Ports, sockets and the application protocols

A protocol name without a number beside it is only half learned. This section supplies the numbers: what a port identifies, how the port space is divided, and how an address and a port together name one end of one conversation. Table 3.23 sets out eleven application protocols with the port each uses, the transport it runs over and the work it actually does. Two figures then walk through the exchanges most often asked for, resolving a name and leasing an address, and the last three questions put the numbers to work.

BINDER Question 163

A single server in a district office at Lindi answers mail, serves the intranet page and accepts a remote login, all through one network card carrying one address. Something must tell the arriving traffic apart, and it cannot be the address.

Explain what a port number identifies, state the width of the field that carries it, and name the three ranges into which the port numbers are divided.

Model Answer

A port number identifies a process on a machine rather than the machine itself; it is carried in a 16-bit field, so it runs from 0 to 65535, and those 65,536 numbers fall into three ranges: well-known 0 to 1023, registered 1024 to 49151 and dynamic 49152 to 65535.

What the number addresses: the IP address brings the packet to the right machine, and the port number takes what is inside it to the right program. One server may run mail, web and remote login at once behind a single address, so the port is the only field that says which of the three a given segment belongs to.

Sixteen bits, and what they buy: the port sits in an unsigned 16-bit field in both the TCP and the UDP header. Sixteen bits give 65,536 distinct values, numbered 0 to 65535, and no service is ever assigned anything higher because nothing higher can be written in the field.

$$\text{Port numbers available} = 2^{16} = 65{,}536$$

Well-known ports, 0 to 1023: the lowest 1,024 numbers are assigned centrally to standard services, which is what lets a browser reach a web server on port 80 with nothing arranged in advance. Most operating systems require administrative privilege before a program may listen on one.

Registered ports, 1024 to 49151: 48,128 numbers recorded on request for particular applications, such as a database or a school management system. Registration is a convention that keeps two products from colliding; nothing in the protocol enforces it.

Dynamic ports, 49152 to 65535: the remaining 16,384, also called ephemeral or private. The operating system lends one to a client for the life of a single conversation and takes it back afterwards, which is why a client almost never chooses its own number.

The three ranges account for every number, since 1,024 and 48,128 and 16,384 come to 65,536 exactly. A port is not the connector on the back of the machine, and an answer describing that earns nothing.

The Master Key: *An address finds the machine, a port finds the program, and neither can do the other's work.*

BINDER Question 164

In a computer room at Pangani three machines open the results page together, and the student at the third opens it twice, in two windows. Four conversations are now aimed at one port on one server, and the server keeps all four apart. **Define a socket, and explain how a server tells apart several clients that are all connected to the same port.**

Model Answer

A socket is an IP address and a port number taken together, naming one end of one conversation, and a connection is identified by the pair of sockets at its two ends; since no two conversations can have both ends identical, none is ever confused with another.

The definition to write down: a socket is an IP address and a port number combined, written as the address, a colon and the number. The address names the host, the port names the process, and only the pair is precise enough to deliver to.

A connection is a pair of sockets: one socket is half a conversation. The transport layer identifies a connection by four values, source address, source port, destination address and destination port, with the protocol as a fifth. Two conversations may share four of the five and still be separate.

Why the server is never confused: the three machines have different addresses, so the source address alone separates them. The two windows on the third machine share an address, but the operating system lent each a different dynamic source port. Four distinct combinations, four distinct conversations.

Listening and connected sockets: the server holds one listening socket on port 443 and creates a separate connected socket for each client that arrives. All carry the same local port and differ in the remote address and remote port recorded against them, which is how each reply finds its way back.

The mistake this prevents: a port is not used up by the client using it. One busy server has a single port and thousands of simultaneous conversations, so the claim that a server can hold only 65,536 clients confuses a port with a connection.

Whenever a question says distinguish, give the four values, then give the two windows on one machine as the example that forces the source port into the answer.

The Master Key: *One socket names an endpoint; it takes two of them to name a conversation.*

BINDER Question 165

The support line at an internet provider will not act on the complaint from a secondary school in Ifakara until somebody says which port and which transport is failing. The head of ICT can name every service in the building and not one number.

Using Table 3.23, state the port number and the transport protocol used by HTTP, HTTPS, FTP, SSH, SMTP, DNS and DHCP, and say in one clause what each of them does.

Model Answer

HTTP uses TCP port 80 and HTTPS TCP port 443; FTP uses TCP port 21 for its commands and TCP port 20 for the file itself; SSH uses TCP port 22; SMTP uses TCP port 25; DNS uses port 53, over UDP for ordinary queries and over TCP for large ones; and DHCP uses UDP, port 67 at the server and port 68 at the client.

Protocol	Port	Transport	What it does
FTP data	20	TCP	Carries the file itself while a transfer runs
FTP control	21	TCP	Carries the commands and replies of a transfer session
SSH and SFTP	22	TCP	Encrypted remote login, and file transfer inside it
Telnet	23	TCP	Remote login in clear text, password included
SMTP	25	TCP	Sends mail out, and relays it between mail servers
DNS	53	UDP, TCP if large	Resolves a name into an address
DHCP	67 server, 68 client	UDP	Leases an address, mask, gateway and resolver
HTTP	80	TCP	Fetches web resources, in clear text
POP3	110	TCP	Downloads mail, normally clearing the server copy
IMAP	143	TCP	Reads mail that stays on the server, from any device
HTTPS	443	TCP	HTTP inside TLS, so request and reply are encrypted

Table 3.23: *Application protocols, their ports and their transports.*

The web pair, 80 and 443: HTTP fetches the resource a web address names and returns it in clear text. HTTPS is the same protocol carried inside TLS, so anyone watching the path learns which server is being visited but not which page was asked for. Both run over TCP, because half a page is worth nothing.

Transfer and login, 20, 21 and 22: FTP keeps commands and data apart: the control connection on port 21 lasts the whole session and port 20 carries the file. Neither is encrypted. SSH on port 22 gives an encrypted remote login, and SFTP is file transfer inside that same SSH session, which is why it shares port 22 and is not FTP at all.

Mail, 25 with 110 and 143: SMTP on port 25 pushes mail outward, from a client to its own server and on between servers. Collecting mail is a different job: POP3 on port 110 downloads it and normally clears the server copy, while IMAP on port 143 leaves it on the server so that every device sees one mailbox.

The two that serve the network itself: DNS on port 53 turns a name into an address, and DHCP on ports 67 and 68 hands a client its address, mask, gateway and resolver. Both use UDP for the ordinary case, because each exchanges one short message and one short reply.

Answer in the order number, transport, job, every time. Naming the protocol alone disposes of only the easiest third of the question.

The Master Key: *Learn a protocol as three facts and not one: the number, the transport and the work.*

BINDER Question 166

Typing a name into a browser in a Temeke classroom looks like a single action. Underneath, the machine asks one question of one server, that server asks three more, and only when the last answers does the browser know where to send its request.

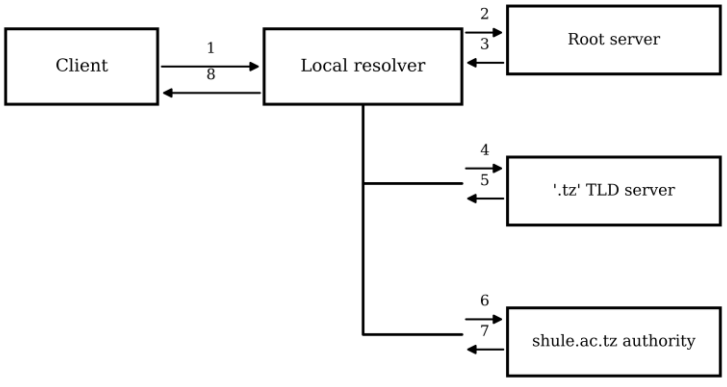
Using Figure 3.28, trace the resolution of the school's web address into an IP address, naming every server that is asked and stating what each one contributes.

Model Answer

The browser sends only the host part of the address to the local resolver, as one recursive query; the resolver then works iteratively down the hierarchy, asking a root server, then the .tz top-level domain server, then the school's own authoritative server, before answering the classroom machine and caching the result.

What is sent, and what is not: a web address divides into a scheme, a host, an optional port, a path and an optional query. Only the host needs resolving. The scheme supplies the port when none is written, so https implies 443 and http implies 80. The path and query are for the web server, not the resolver.

One recursive query from the client: the classroom machine asks its configured resolver for the address and expects a complete answer or a failure, never advice on where to ask next. That one question is recursive, and it is the only query the client itself makes.



1. A? www.shule.ac.tz
2. who serves .tz?
3. ask the .tz servers
4. who serves shule.ac.tz?
5. ask ns1.shule.ac.tz
6. A? www.shule.ac.tz
7. A = 41.86.x.x
8. 41.86.x.x, cached

The client asks once, recursively. The resolver then asks three times, iteratively, and caches what it learns for the record's time-to-live. DNS runs on port 53.

Figure 3.28: *Resolving a name. The client asks once, recursively; the resolver then asks three times, iteratively, and caches what it learns.*

Three iterative queries from the resolver: the resolver asks a root server, which names the servers for .tz. It asks one of those, which names the school's authoritative server. That server holds the zone and returns the address record. Each reply is a referral until the last, which is the answer.

Port and transport throughout: each exchange is a short UDP message to port 53. A query and its answer fit inside one datagram, and an unanswered query is simply asked again, or asked of another server.

What the resolver keeps: the answer is stored for the time to live carried on the record, so every other machine asking that name during the period is served from the cache, troubling neither the root nor the .tz servers again.

Name the parties in order, client, resolver, root, top-level domain and authority, and say which query is recursive and which are iterative; that distinction is what is tested.

The Master Key: *One recursive question from the client, three iterative ones from the resolver, and an answer kept until its time to live expires.*

REAL Question 167

A laboratory technician at a secondary school in Bariadi switches on thirty machines at eight o'clock on a Monday. Not one of them has an address. Within seconds each has an address, a mask, a gateway and a resolver, and nobody has typed anything.

Using Figure 3.29, describe the four DHCP messages in the order they occur, explain why the first of them must be broadcast, and account for the machines that are still without an address.

Model Answer

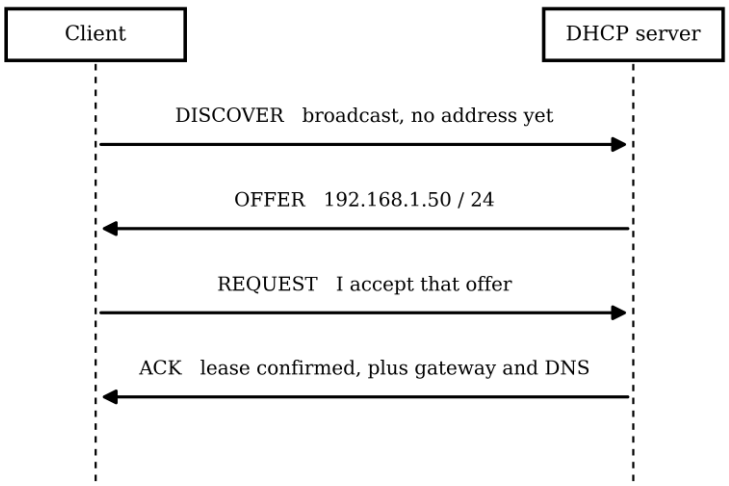
The four messages are Discover, Offer, Request and Acknowledge, carried over UDP to port 67 at the server and port 68 at the client; the first must be broadcast because a client with no address of its own cannot know the address of any server to ask; and the machines left out are those that arrived after the pool ran dry.

Discover, from nowhere to everywhere: the client broadcasts a request for configuration. Having no address yet, it sends from an all zeros source address to the all ones broadcast address, so every machine on the link receives the message although only a DHCP server acts on it.

Offer, a proposal and not a grant: a server holding a free address replies with a specific one, together with the subnet mask, the default gateway, the resolvers and the length of the lease. Nothing is allocated yet; the address is merely reserved while the server waits to hear whether the offer is taken.

Request, broadcast on purpose: the client accepts one offer, naming the server that made it, and broadcasts that acceptance rather than sending it directly. Broadcasting lets any second server that also made an offer see that it was refused, and return its reserved address to the pool instead of holding it idle.

Acknowledge, and the lease begins: the server confirms the lease, and only now may the client use the address. The client notes the lease time and renews well before it expires, this time directly to the server it now knows, so a working machine never repeats the broadcast.



Discover, Offer, Request, Acknowledge: server port 67, client port 68, both over UDP, because a client with no address yet cannot open a connection.

Figure 3.29: *The four DHCP messages: Discover, Offer, Request and Acknowledge.*

Why broadcast, and why UDP: a broadcast is the only message a machine can send when it knows neither its own address nor anyone else's. For the same reason the exchange runs over UDP: no connection can be established from an address that does not yet exist.

The six machines still waiting: the pool in this laboratory holds twenty-four addresses and thirty machines were switched on, so six of them receive no offer at all and fall back to a self-assigned address that reaches nothing. Widen the pool, or shorten the lease so that addresses come back when a machine is switched off.

Learn the order by its initials, D, O, R, A, and check the arrows on any diagram before trusting it: a figure has reached print still carrying the production note 'To redraw', and a wrong label is read as confidently as a right one.

The Master Key: *Discover, Offer, Request, Acknowledge: a client with no address must shout before it can speak.*

REAL Question 168

A new firewall goes in at a district office in Sumbawanga on a Friday afternoon, with one outbound rule: traffic to ports 80 and 443 is permitted and everything else is dropped. The complaints that arrive on Monday morning are not the ones anybody expected.

Determine which of the six services listed below will still work through the new rule and which will not, and state what must change before the office can work at all.

1. Staff browsing pages over HTTP, port 80.

2. The payments portal over HTTPS, port 443.

3. Resolving names by DNS, port 53.

4. Sending returns by SMTP to a mail server, port 25.

5. Collecting mail with IMAP, port 143.

6. A contractor's remote support session over SSH, port 22.

Model Answer

Only items 1 and 2 are permitted by the rule, and even those two will appear to fail, because item 3, name resolution on port 53, is blocked along with SMTP on 25, IMAP on 143 and SSH on 22; port 53 must be opened outbound, or a resolver must be provided inside the office, before anything works.

What the rule actually matches: a rule of this kind tests the destination port of traffic leaving the office. A request to a web server carries destination port 80 or 443 and is permitted; the reply returns to whatever dynamic port the client was lent, and a firewall that remembers the requests it has already allowed will admit that reply without a separate rule.

The two that pass: ordinary browsing on port 80 and the payments portal on port 443 both match the rule and leave the office. They are the only two of the six that the rule names, and both are TCP.

The three that are plainly blocked: SMTP to an outside mail server on port 25, IMAP to a mailbox on port 143 and the contractor's session on port 22 all carry destination ports the rule does not permit, so each is dropped. The applications report a timeout rather than a refusal, which is why a filtered port looks to a user like a dead link.

The one that takes the others with it: DNS on port 53 is blocked too, and it is not simply a sixth casualty. A browser given a name must resolve it before it can send anything to port 443 at all, so with 53 closed the two permitted services fail as well, and they fail complaining that the site cannot be found rather than that a port is shut.

What must change: permit UDP port 53 outbound, and better still to the two resolvers the office actually uses rather than to the whole internet. Then judge the other three on their merits: mail collection and remote support are genuinely needed, so open ports 143 and 22 to named destinations, and leave everything unnamed closed.

The lesson travels beyond this office. A service you permit may rest on a service you never thought about, and name resolution is the one that is nearly always forgotten.

The Master Key: *Open the port for the service you want, then open the port for the service that service depends on.*

HOT Question 169

Six years of switch configuration at a secondary school in Kondoa have been done over Telnet, from any machine on the network, with a password that crosses the link as plain characters. It has never failed. That is the argument for keeping it.

Assess whether the school should block port 23 across the whole network, and defend the recommendation you reach.

Model Answer

Block it, at the boundary and between the internal networks, and permit Telnet only from one named management machine to equipment too old to offer anything else; the reason is not that Telnet is old fashioned but that it has no protection of any kind that could be switched on, so the only control available is whether it runs.

What port 23 actually carries: Telnet is a remote login service on TCP port 23 which sends every character as it is typed, in clear text. The username, the password and every configuration command are readable by anyone able to capture a frame anywhere on the path, and there is no setting that changes this.

The case for leaving it open: some old managed switches offer no other way in. A teacher demonstrating a protocol may want Telnet to a service on some other port, because typing a request by hand and reading the raw reply is the clearest way to show what a protocol is. A blanket block can also strand a support visit.

Why that case does not survive: the replacement exists and costs nothing to use. SSH on port 22 provides the same login with the whole session encrypted, and equipment sold in the past fifteen years supports it. The network carries marks, staff records and student work, and the person most likely to be watching the wire is a curious student with a laptop.

What blocking means in practice: deny TCP 23 outbound at the router, so that nobody inside can be induced to log in to a remote device in clear text, and deny it from the student networks to the switches. Then disable the Telnet service on the switches themselves, because a rule protects only the paths it sits on.

The judgement, with its exception: block it. Where a switch is genuinely too old for SSH, permit port 23 from one management address only, write down which devices those are, and treat that list as a replacement programme rather than a standing arrangement. The cost of blocking is one documented inconvenience; the cost of not blocking is every credential on the network.

An answer recommending that Telnet be used carefully has missed the point. There is no careful way to send a password in clear text across a network other people can reach.

The Master Key: *Where a protocol offers nothing to secure, the only security decision left is whether it runs at all.*

HOT Question 170

Two entries in one monitoring report from a secondary school in Mbinga disagree. The daytime rows show DNS on UDP, thousands of them, one packet out and one back. A single row at two in the morning shows DNS on TCP for eleven seconds.

Justify the use of UDP for ordinary DNS queries and of TCP for large ones, and reach a conclusion about what the arrangement shows.

Model Answer

DNS uses UDP for the ordinary case because a query and its answer are each a single short message, so establishing a connection would cost more than the exchange it was carrying; it turns to TCP when an answer is too large for one datagram or when an entire zone is being copied, because those need an ordered stream of arbitrary length that UDP does not offer.

The ordinary query is tiny: a question is one name and one record type, and an answer is an address and a time to live. Both fit inside a single datagram sent to port 53. The whole transaction costs one packet out and one packet back, and a resolver serving a district may be handling thousands of them a second.

Loss is cheap to repair here: if a query or its reply disappears, the resolver waits briefly and asks again, or asks a different server. There is nothing to reassemble and nothing to acknowledge, because the transaction is one message. Reliability the application can supply this cheaply is not worth paying a transport for.

When one datagram is not enough: some answers are long: a name with many records behind it, a reply carrying signatures, or a zone transfer, in which a secondary server copies every record a domain holds. A server that cannot fit its answer into a datagram marks the reply as truncated, and the resolver asks the same question again over TCP, on the same port 53.

Why TCP suits those: a long answer must arrive complete and in order. TCP delivers a stream of any length, while UDP delivers separate datagrams that may arrive out of order, or not at all. Against the size of a zone transfer the cost of opening a connection is negligible, so the balance tips the other way.

The verdict on the design: this is not untidiness that survived into the standard; it is the design working. DNS chooses its transport for each message rather than once for the whole protocol, taking the cheap transport for the case that happens millions of times a day and the careful one for the case that must not fail.

Write the port once and the transport twice: DNS is port 53, over UDP by default and over TCP when the answer will not fit or a whole zone is moving.

The Master Key: *Choose a transport for the message in front of you, not for the protocol it belongs to.*

HOT Question 171

The results portal at a school in Stone Town takes several seconds to appear the first time it is opened each morning, then opens instantly for the rest of the day. Nothing is repaired in between, and a visiting salesman advises buying more bandwidth.

Identify what is responsible for the difference between the first request and the later ones, and judge whether it is a fault worth spending money on.

Model Answer

Caching is responsible, at two levels: the first request pays for a full name resolution and for fetching every object on the page, while later requests are served from the resolver's cache and the browser's cache; this is the system behaving correctly, provided the school knows what a cached answer does when something changes.

What the first request pays for: the name must be resolved from the root downwards before one byte of the page can be asked for. The browser then sends a GET for the page and another for every image, stylesheet and script it names. A GET asks for a resource; a POST, used when marks are submitted, sends data for the server to act on.

What the second request avoids: the address is already in the resolver's cache and stays there for the record's time to live, so the lookup disappears entirely. The objects are already in the browser's cache, so the browser either uses them outright or sends a conditional request and receives a short reply saying nothing has changed.

Reading the status codes: the first digit gives the class. A first load is a run of 200 replies, and 2xx means success. 3xx means redirection, or here 304, Not Modified, a cache being confirmed rather than a page sent. 4xx means the client asked wrongly and 5xx means the server failed.

Why bandwidth is the wrong purchase: the second load is fast over the same link as the first, so the link is not the constraint. More bandwidth shortens the transfer of large objects and does nothing whatever for the round trips spent resolving a name, which is where most of that first delay is going.

When caching does become a problem: a cache serves what it was given until it expires. If the portal moves to a different address, machines holding the old one keep failing for the length of the time to live, so lower that value a day before a planned move and raise it afterwards.

The judgement: this is not a fault, and no money should follow it. It is the behaviour that makes a shared school link usable at all. Explain it to the staff, confirm that the resolver in use is a local one, and reserve any spending for the case where the second load is slow as well.

A first visit and a second visit are not the same journey. Compare like with like before diagnosing, and clear the caches to measure a first visit again.

The Master Key: *A cache makes the second visit a different journey, so time both before you blame the link.*

Guided transmission media and their limits

Copper and glass carry almost every wired network in the country, and the choice between them is settled by three questions: what the cable is made of, how far it will reach, and what that reach costs. This section fixes

the construction of twisted pair, coaxial cable and optical fibre, prints the distance limit of each in metres and kilometres, and sets out the comparison a specification is written from. It assumes only that you know a signal grows weaker as it travels.

BINDER Question 172

Two quotations arrive for linking a district office in Lindi to its store across the compound. The first prices a trench, a duct and a drum of cable. The second prices two short masts, two radios and a morning spent aligning them.

Distinguish guided from unguided transmission media, and give two examples of each.

Model Answer

A guided medium confines the signal inside a path laid for it, a metal conductor or a glass core, so the route is fixed at installation; an unguided medium launches it into free space as a spreading wave, so twisted pair and fibre are guided, microwave and satellite unguided.

What guiding means: the energy travels inside something solid put there for the purpose. The route is fixed when the cable is laid, and the capacity of that path belongs to whoever owns it.

What unguided means: nothing directs the wave but the antenna that launched it. Free space is shared, so a transmitter must be licensed or keep inside a band opened for general use, and it hears every other user of that band.

Examples of each: the guided family is twisted pair, coaxial cable and optical fibre, so name a run of unshielded pair from a switch to a desk and a fibre span between two buildings. Against them set terrestrial microwave between two masts, and satellite.

Where they differ in use: a guided run must be reached physically before it can be listened to, and it ignores the weather. An unguided one reaches every receiver in range and fades in rain, but it crosses a river or a road without asking anybody.

The question that decides: ask whether a route exists that you may dig along or hang a cable from. Where there is one, guided media give more capacity, for longer, in private. Where there is none, the wireless link is the only link.

The two quotations answer different questions. One buys a permanent path and pays for the digging; the other buys the air and pays by keeping its line of sight clear.

The Master Key: *Guided media are chosen by what you may dig through, unguided media by what you can see across.*

BINDER Question 173

An installer in Kariakoo strips back a metre of cable, combs the eight conductors straight so the run into the patch panel will look tidy, and punches them down in the right order. The link now drops frames whenever the fluorescent tubes come on.

Explain why the two conductors of a twisted pair are wound about each other, and distinguish unshielded twisted pair from shielded twisted pair.

Model Answer

The pair carries its signal as the difference between two conductors, so twisting makes interference arrive almost equally on both, where the subtraction cancels it; unshielded pair relies on that balance alone, while shielded pair adds an earthed metal screen.

The balanced pair: one conductor carries the signal and the other its mirror image. The receiver reads the difference and discards whatever the two share, so anything added equally to both disappears at the far end.

Why twisting equalises: untwisted, one conductor would lie nearer an interfering source and pick up more of it. Twisting swaps their positions every few centimetres, so each spends the same distance nearest the source and the interference arrives on both alike, which is what the receiver discards.

The second benefit: each turn encloses a small loop and successive loops face opposite ways, so what one turn radiates the next opposes. The pair therefore disturbs its neighbours far less than a straight one.

Four different rates: the pairs in one jacket are wound at unequal lay lengths. Were they identical, two pairs would stay in step for the whole run and couple strongly; unequal rates keep them slipping past one another.

Unshielded against shielded: unshielded pair is four pairs and a jacket: thin, flexible, cheap, the ordinary indoor choice. Shielded pair adds foil round each pair or a braid round all four, intercepting what the balance cannot; it is thicker, dearer, stiffer, and useless unless the screen is earthed.

One connector, many grades: both terminate in the same eight-position RJ-45 plug, which takes all four pairs. The plug says nothing about what is inside, so read the grade from the jacket; Table 3.24 gives what each supports.

Category	Bandwidth	Supports	Maximum length
Cat 3	16 MHz	10 Mbit/s Ethernet and voice	100 m
Cat 4	20 MHz	16 Mbit/s Token Ring	100 m
Cat 5	100 MHz	100 Mbit/s Ethernet	100 m
Cat 5e	100 MHz	1 Gbit/s Ethernet	100 m
Cat 6	250 MHz	1 Gbit/s, and 10 Gbit/s on shortened runs	100 m
Cat 6a	500 MHz	10 Gbit/s over the full channel	100 m

Table 3.24: *The twisted-pair categories.*

Read the table across, not down: the bandwidth column and the supports column are different quantities, and the last column never changes.

The Master Key: *A screen keeps interference out; a twist makes what gets in arrive on both conductors at once, where it cancels.*

BINDER Question 174

A drum of black cable left from an aerial job sits in the store of a college in Sengerema, and somebody suggests it for the backbone. It has a single copper core, it is heavier than the blue cable, and it looks more serious.

Using Figure 3.30, describe the construction of coaxial cable and state the segment limit for thin and for thick coaxial Ethernet.

Model Answer

Coaxial cable is four concentric layers sharing one axis, a copper core, an insulating dielectric, a braided shield that is also the return conductor, and a protective jacket; thin coaxial Ethernet is limited to 185 m in one segment and thick coaxial Ethernet to 500 m.

The four layers: at the centre is the core, solid or stranded copper, carrying the signal. The dielectric insulates it and holds it central. A braid of fine wires screens the core and carries the return current, and the jacket protects the rest against damp and handling.

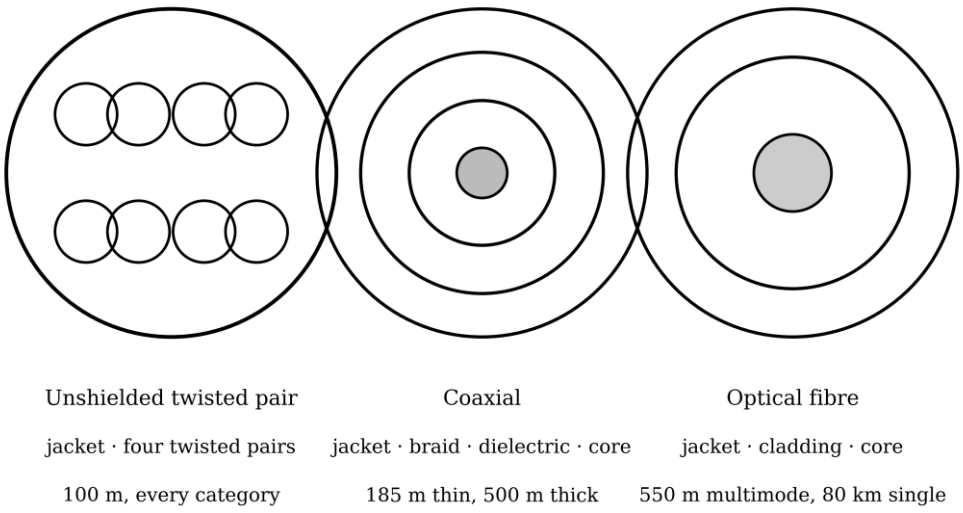


Figure 3.30: *The three guided media in cross-section, with the distance limit of each.*

Why the geometry works: in the middle drawing the braid completely surrounds the core and shares its axis, so outward and return currents are concentric and interference meets the braid first. That is a stronger defence than balance alone.

Two impedances: the spacing the dielectric fixes sets the characteristic impedance. Data cable is made to 50 ohms and television cable to 75 ohms. They look alike and take similar connectors, yet mixing them reflects part of the signal at every junction.

Thin and thick Ethernet: thin coaxial Ethernet, on flexible RG-58, runs 185 m in a segment. Thick coaxial Ethernet, on the rigid heavier-cored cable that named it, runs 500 m. Neither is installed today; both numbers are still asked for.

What coaxial is not: it is not unidirectional. Traffic passes both ways along one cable, in old Ethernet by taking turns and in cable television by giving each direction its own frequencies. Nor is it fixed at 10 Mbit/s: that belonged to the obsolete Ethernets, and cable systems now carry several gigabits.

The drum in the store is almost certainly 75 ohm aerial cable, so it belongs on a television and not a backbone. Read the jacket before anybody cuts it.

The Master Key: *Quote a coaxial cable by its impedance before its thickness, because 50 and 75 ohms are two different cables wearing one coat.*

BINDER Question 175

A contractor splicing fibre beside the Njombe road hands one end of a reel to an apprentice and shines a torch into the other. The reel is coiled twice round the apprentice's forearm; the light arrives undimmed. Bend it sharply and the glow dies.

Using Figure 3.31, state the condition for total internal reflection and distinguish single-mode from multimode fibre.

Model Answer

Light is held in the core by total internal reflection, because the cladding has a lower refractive index and any ray meeting the boundary beyond the critical angle is thrown wholly back; a wide core admits several such paths and a narrow one admits a single path.

The layers of a fibre: a glass core carries the light. A glass cladding of lower refractive index surrounds it and makes the reflection possible. A plastic buffer protects the glass, and the outer jacket, with its strength members, takes the pulling.

The reflection condition: a ray crossing from denser glass into less dense glass bends away from the normal, and past a certain angle of incidence, the critical angle, it cannot cross at all and is returned entirely to the core, where it stays for kilometres.

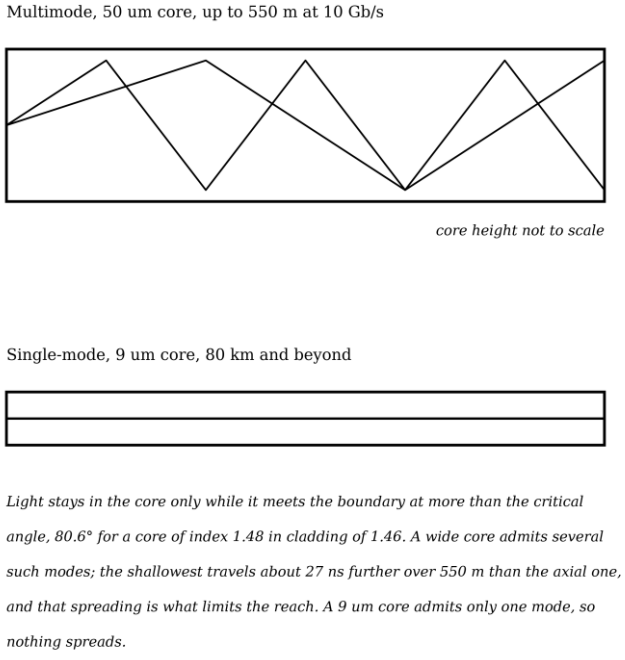


Figure 3.31: *Multimode and single-mode fibre. A wide core admits several paths of different length, the pulse spreads, and the reach collapses.*

What a wide core costs: in the upper drawing several rays satisfy that condition at once. Each is a mode, and a zig-zag path is longer than the straight one, so one pulse arrives smeared over several instants and finally overlaps its neighbour.

Multimode: a core 50 micrometres across admits those several modes. It is cheap to make, easy to couple light into, and it reaches 550 m at 10 Gbit/s, which covers a campus and nothing beyond one.

Single-mode: a core 9 micrometres across is too narrow to hold more than one path, so nothing spreads and distance is limited by loss instead. It reaches 80 km. The laser and the optics at each end are what cost.

The core diameter decides the reach: widen it and coupling is easy and the distance short, narrow it and you buy kilometres. The two are not interchangeable, and multimode fitted to a single-mode port will pass light, pass a casual test, and lose most of the signal.

The Master Key: *Fewer paths, further light: reach in fibre is bought by narrowing the core, never by turning up the power.*

REAL Question 176

The office block at a secondary school in Kinondoni stands 140 metres from the server room, once the route has gone up a wall, along a ceiling and down again. The supplier quotes one continuous length of Cat 6a, the highest category available.

Determine why a single copper run cannot serve that office, and recommend two workable options.

Model Answer

Every category of unshielded twisted pair carrying Ethernet is limited to a 100 m channel, so a 140 m route overruns the limit by 40 m and no grade of copper closes that gap; reach the office either with optical fibre for the whole distance or with an intermediate switch that divides the route into two compliant hops.

What the hundred metres includes: the limit is a channel, not a drum length: 90 m of fixed horizontal cable from the patch panel to the wall outlet, plus up to 10 m of patch leads at the two ends. Ninety and ten make the hundred, so the cords sit inside the budget.

Why 140 metres fails: the route is 40 m beyond the limit before one patch lead is counted. The figure is not a fuse: an over-length run commonly links, passes a few files and then loses frames as traffic rises, which buys a term of blaming the computers.

Category buys rate, not distance: a higher category raises the bandwidth the pair carries over those hundred metres. It does not extend them, and no category in the table reaches further, so the Cat 6a in this quotation buys nothing whatever at this distance.

The first option, fibre: run multimode fibre, which reaches 550 m at 10 Gbit/s and so covers 140 m with a great deal to spare, and fit a fibre port or a media converter at each end. It carries no metal, so a surge on one building cannot travel along it into the other.

The second option, a switch: put a switch in a room about halfway, where there is mains power and a lockable cupboard. Each hop is then comfortably under 100 m and each is a fresh link. The price is the power point, the enclosure and one more device to fail.

What will not work: joining two lengths with a coupler still leaves one 140 m channel and merely adds loss at the join. A longer patch lead at the far end does not help either, because patch leads come out of the same hundred metres.

Property	Twisted pair	Coaxial	Fibre
Bandwidth	500 MHz at Cat 6a	Wide, falling with length	Beyond present needs
Distance	100 m	185 m thin, 500 m thick	550 m multimode, 80 km single-mode
Interference	Resisted by the twist	Resisted by the braid	Immune
Cost	Lowest	Middling	Cheap glass, dear electronics
Installation	Simple and forgiving	Stiff and bulky	Skilled, and bends kept gentle
Security	Can be tapped quietly	Can be tapped quietly	A tap disturbs the light

Table 3.25: *The three guided media compared.*

Recommend the fibre where the route can be ducted, because it stays right when the school buys faster switches. Recommend the intermediate switch where there is a powered room and no money for civil work.

The Master Key: *When a run is too long, change the medium or break the run; no grade of copper will stretch.*

REAL Question 177

At the back of a quotation for cabling a new block at a district office in Stone Town is a neatly typed technical annex. Three of its statements are wrong in the same way, and the block will be cabled to whatever it says.

Identify what is wrong with each of the three statements below, give the correct position, and say why each mistake is an easy one to make.

1. Optical fibre is unaffected by attenuation, so no loss calculation is required on a fibre route.

2. Coaxial cable is resistant to attenuation, so segment length need not be checked where coaxial is used.

3. Cat 6 may be run to 150 m, since a higher category carries the signal further than a lower one.

Model Answer

All three swap a property the medium has for one it has not: fibre loses about 0.2 dB in every kilometre, coaxial loss climbs sharply with frequency and is the reason its segments are short, and every category of unshielded twisted pair carrying Ethernet stops at 100 m.

The first statement: fibre loses light. At 1550 nm a good single-mode fibre loses about 0.2 dB per kilometre, so a 40 km span arrives 8 dB down before one splice or connector is counted, and that budget is what decides where amplifiers go. Fibre is immune to electromagnetic interference, since light carries no charge, and the temptation is that the two claims sit side by side in every list of advantages, so a loss too small to matter over 100 m is rounded to no loss at all.

The second statement: coaxial loss is not small, and it worsens as frequency rises, which is precisely why thin coaxial Ethernet stops at 185 m and thick at 500 m. The braid is a screen, and a screen excludes what would be added; it does nothing about what is lost. The temptation is the word shielded doing two jobs at once, so a cable well protected against interference is assumed to be well protected against everything.

The third statement: category fixes bandwidth, not reach. Cat 6 is a 250 MHz cable and Cat 6a a 500 MHz cable, and both, like Cat 3 before them, are held to a 100 m channel. A higher category buys rate over those hundred metres and never buys extra metres. The temptation is that every other column of a category table improves as the number rises, and since most such tables print no length at all, nothing on the page contradicts the guess.

What prevents all three: each error trades one quantity for another: loss for interference, screening for loss, bandwidth for distance. Ask of any claim what quantity it concerns and in what unit that quantity is measured. A medium is honestly described only by loss per unit length at a stated frequency, together with a stated maximum length.

Mark the annex rather than reject it. The prices may well be good and the corrections run to three sentences; what would be expensive is accepting the distances that follow from the three sentences as written.

The Master Key: *Immunity to interference and freedom from loss are two separate claims, and no medium ever invented has supported the second.*

REAL Question 178

A sales representative visits a school at Ilala with three answers for the head of ICT, who had queried the cable on the order. They read like fact and are meant to settle it. If they do, the wrong drum arrives in a fortnight.

Correct each of these three statements, give the specification that should have been written, and say why each is a tempting thing to put on paper.

1. Category 4 cable carries 20 Mbit/s, which is ample for a 16 Mbit/s Token Ring.

2. RG-7 is the 50 ohm coaxial cable specified for thick Ethernet, and RG-7 will be supplied for the backbone.

3. The advantage of coaxial cable is that each of its pairs has its own earthed foil screen, which is what keeps crosstalk between the pairs low.

Model Answer

Cat 4 offers 20 MHz of bandwidth and carries 16 Mbit/s; RG-7 is 75 ohm television cable, while thick Ethernet ran on 50 ohm RG-8 or RG-11; and a screen around each pair is the definition of shielded twisted pair, not of coaxial cable, which has no pairs to screen.

The Cat 4 figure: twenty is the bandwidth in megahertz, not the rate in megabits per second. The category was made for 16 Mbit/s Token Ring, as the sentence half admits, and 16 Mbit/s is its rated rate, so the quotation overstates the cable by a quarter. The temptation is Cat 5, where the bandwidth is 100 MHz and the rate is 100 Mbit/s, so the two columns read alike; a habit learned there is carried back to Cat 4 and fails.

The RG number: RG-7 is 75 ohm cable made for distributing television. Thick Ethernet ran on 50 ohm RG-8 or RG-11 and thin Ethernet on 50 ohm RG-58, so RG-7 pushed into a 50 ohm system reflects part of every signal at every junction. The temptation is that RG-11 truly is a thicknet cable and RG-7 sits beside it in the same catalogue, one digit away, and an RG number carries no impedance of itself.

The misplaced advantage: that sentence is true of shielded twisted pair and has been written under the wrong heading. Coaxial cable is one centre conductor inside one shield, so it has no pairs, no crosstalk between pairs and nothing to screen individually. The temptation is that both cables are shielded, so the same list of advantages reads well under either name, and a heading that has slipped one section is invisible to anyone who already trusts the text.

What the three share: not one would survive reading a datasheet across its column headings. Quote a cable as a category with its bandwidth in megahertz, its rate in megabits or gigabits per second and its 100 m channel, or as an RG number with its impedance in ohms. A family name alone, coax or shielded pair, specifies nothing that can be ordered.

Return the page with the part numbers amended rather than with an argument. Cable bought to the wrong impedance is a cost that surfaces months later, when nobody connects the fault to the order.

The Master Key: *Numbers with units specify a cable; a family name only names its relatives.*

HOT Question 179

Eight hundred metres of planted slope separate the grading shed from the office of a tea estate above Mbinga, and one system must serve both. One quotation offers a wireless bridge working by Friday, the other fibre, a trench and a month of disruption.

Assess whether the two buildings should be linked with optical fibre or with a wireless bridge, and commit to a recommendation.

Model Answer

Link them with single-mode fibre in a duct. Copper is out eight times over at this distance, multimode is 250 m short of it, and though the wireless bridge could work on Friday, the estate owns the ground and will still be there in twenty years.

Copper is not a candidate: the 100 m channel settles that at once. Eight hundred metres is eight times the distance any category of twisted pair may be run, and no arrangement of couplers or higher grades alters the figure.

What the bridge offers: no trench, no wayleave, no damage to planted ground, and a working link within a day for a fraction of the capital. Two dishes in sight of one another carry a useful rate over 800 m, and they can be moved if a building changes its use.

What the bridge costs later: it needs clear line of sight, and a planted slope erodes that as trees grow. Heavy rain absorbs the higher frequencies, so the link is weakest in the months when the estate is busiest. The band is shared, the masts want earthing, and alignment drifts and must be checked.

What the fibre costs: almost all of it is civil work: 800 m of trench through planted ground, duct, a draw pit or two and a termination at each end. The glass is a small part of the bill, and nothing may be ploughed along the marked route again.

Which fibre to specify: not multimode. Its reach is 550 m at 10 Gbit/s, which leaves it 250 m short of this span, and a link that misses the specification is not a link. Single-mode reaches 80 km, costs a little more by the metre, and puts the difference into the transceivers at each end.

The argument that decides it: the trench is dug once, and the duct will take a second fibre without digging again. Fibre puts no metal between two buildings on separate earths, which matters where lightning is common, and the weather that threatens the bridge does not touch it.

The judgement and the fallback: install single-mode fibre in duct, and use the wireless bridge honestly: as the temporary link while the trench is dug, and afterwards as the standby for the day a digger cuts the cable.

The distance chose the type of fibre and the ownership chose the medium. An estate that owns its ground should not rent the air when it can dig once and be done.

The Master Key: *Where you own the route, buy the trench; the cable lying in it is the cheapest part and the longest lived.*

HOT Question 180

The conduit at a new secondary school in Pangani goes into the walls before the plaster does. The contractor offers Cat 6 at the price agreed, or Cat 6a for more on the cable and more on the outlets. The board decides at one meeting.

Judge whether the extra cost of Cat 6a over Cat 6 is justified in a school building being wired today, and defend your answer.

Model Answer

Buy Cat 6a for everything going into the walls and the buried ducts. The cable is a small share of an installed cost dominated by labour and builder's work, it will outlive three generations of switch, and Cat 6a is the lowest category that carries 10 Gbit/s over the full 100 m channel.

What the money buys: Cat 6a is specified to 500 MHz where Cat 6 stops at 250 MHz, twice the bandwidth, and it is tested for interference between one cable and its neighbours in the same bundle. That is what lets it carry 10 Gbit/s over a full channel, which Cat 6 manages only over a shortened run.

What the money costs: Cat 6a is thicker and stiffer and wants a larger bend radius, so trays and ducts fill sooner and corners must be planned more generously. The cable costs more by the metre, the outlets and panels cost more, and every termination takes a little longer to make well.

Where the real cost sits: cable is a minor part of a finished installation. The chasing, the conduit, the trunking and the labour are the greater part, and every one is paid again if the cable is replaced. The cheaper cable saves a fraction of the job and risks repeating all of it.

Power over the same pairs: access points and cameras now draw their power from the pairs that carry their data. That current warms a bundle, and warm copper loses more signal, so the heavier conductors of Cat 6a run cooler and keep more margin.

The case for Cat 6: Cat 6 carries a gigabit to every desk over the full 100 m, which is more than the school connection to the outside world will deliver for years. Where cable runs in surface trunking that opens with a screwdriver, replacing it later is cheap and the saving is real.

The judgement: take Cat 6a for the horizontal runs cast into the building and for every link between blocks, and allow Cat 6 only in surface trunking, and then only if the school keeps a record of which run is which. A site that cannot keep that record should buy Cat 6a throughout.

The question is not which cable is better this year but which will still be adequate when the plaster is twenty years old.

The Master Key: *Buy cable for the life of the wall it is buried in, not for the speed of this year's switch.*

Unguided media, satellite and the spectrum

Where no cable can be laid, the signal is launched into free space and must be caught by an antenna that may be in the next room or 35,786 km above the equator. This section fixes what unguided transmission is, where the four communication bands begin and end, how radio, microwave and infrared differ in the way each propagates, and what the air charges for the convenience: shared frequencies, weather, obstacles and delay. It assumes only that you know a signal weakens as it travels.

BINDER Question 181

Nothing can be inspected between the two ends of a wireless link: no drum, no duct, no trench, no join to check with a meter. What carries the traffic above a street in Kariakoo belongs to nobody and cannot be bought, only licensed.

Define unguided transmission, and name the four bands of the electromagnetic spectrum used for communication, giving the boundary frequencies of each.

Model Answer

Unguided transmission carries data as electromagnetic waves radiated through free space, with no conductor to confine them, and the four bands used for communication are radio from 3 kHz to 1 GHz, microwave from 1 GHz to 300 GHz, infrared from 300 GHz to 400 THz, and visible light from 400 to 900 THz.

What is radiated: an antenna turns the signal into an oscillating electric and magnetic field that travels away at the speed of light. Nothing material carries it, which is why one principle serves across a room and out to orbit.

Frequency settles the character: everything separating one band from the next follows from frequency. A long wave bends round obstacles and passes through walls; a short one carries far more information but is stopped by the first solid thing it meets.

The four bands: the boundaries are conventions rather than physical walls, but they are what a specification is written to, so learn them in the order Table 3.26 sets out, the unit climbing from kilohertz to gigahertz to terahertz.

Band	Frequency range	Propagation	Typical use
Radio	3 kHz to 1 GHz	Omnidirectional, passes through walls	AM and FM broadcasting, television, mobile telephony
Microwave	1 GHz to 300 GHz	Line of sight, must be aimed	Hilltop links, satellite, radar, Wi-Fi at 2.4 and 5 GHz
Infrared	300 GHz to 400 THz	Blocked by any opaque surface	IrDA, remote controls, short links inside one room
Visible light	400 to 900 THz	Straight line, blocked by anything solid	Free-space optical links; the same waves inside fibre

Table 3.26: *The four unguided bands.*

Where satellite belongs: satellite is not a fifth band but a microwave link whose repeater is in orbit, so a classification by medium lists radio, microwave, infrared and satellite, while one by frequency lists the four bands above.

What free space charges: the medium is shared and cannot be owned, so a transmitter works under licence or in a band opened to all. The wave weakens as it spreads, reaches every receiver in range rather than the intended one, and is at the mercy of rain and whatever grows into its path.

Nothing in a wireless link can be dug up and examined, so the specification must be exact: name the band, the frequency and the propagation before anybody climbs a mast.

The Master Key: *Name the frequency and you have named the behaviour, because everything a wireless link can and cannot do follows from it.*

BINDER Question 182

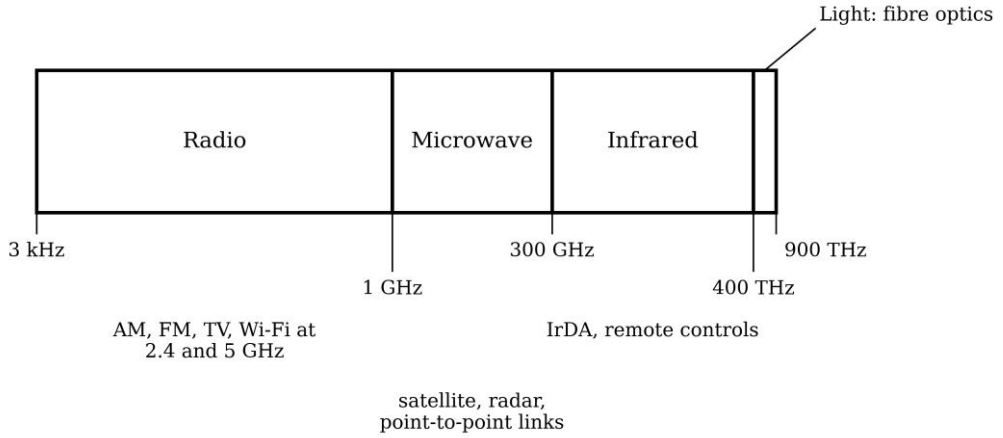
Three transmitters run at a secondary school in Makambako. The office one reaches every classroom without being pointed anywhere. The dish on the hill reaches one dish nine kilometres away. The third cannot pass a closed door. All three radiate the same wave.

Using Figure 3.32, distinguish radio, microwave and infrared by the way each propagates, and give one use of each.

Model Answer

Radio is omnidirectional and passes through walls, so one antenna serves everything in range; microwave is line-of-sight and must be aimed, so a link exists only between two antennas that see one another; and infrared is stopped by any opaque surface, so it never leaves the room.

Radio, spread in every direction: a whip or rod antenna radiates equally all round itself, so any receiver in range hears it and none need be aimed. That suits broadcasting and mobile telephony. The price is that everyone in range also interferes, and most power goes where nobody listens.



Radio is omnidirectional and passes through walls; microwave is line-of-sight and must be aimed; infrared will not leave the room. The boundaries are conventional and worth committing to memory.

Figure 3.32: *The electromagnetic spectrum, with the four bands used for communication and what each carries.*

Microwave, aimed and narrow: a dish or flat panel concentrates the power into a narrow beam, which is why a hilltop hop of 30 to 50 km works on modest power. Both ends must see one another, earth curvature and mast height set the limit, and an alignment that drifts by a degree can end the link.

Infrared, confined to one room: it passes through neither a wall nor a closed door, which is an advantage as well as a limit, since the same wavelengths then serve every classroom along a corridor without interfering, and nobody outside can listen.

Where Wi-Fi actually sits: 2.4 GHz and 5 GHz lie just above the 1 GHz boundary, so by frequency Wi-Fi is microwave, yet its antennas are unaimed and indoors it behaves like ordinary radio, which is why the drawing groups it there.

The trade in one line: spreading the wave buys coverage and costs reach; focusing it buys reach and costs coverage. Ask whether the job is to serve receivers that move or to join two fixed points, and the band chooses itself.

The three transmitters, in the office, on the hill and inside the room, are not three technologies but one, cut at two frequencies.

The Master Key: *Decide whether you are covering an area or joining two points, because that single choice divides the spectrum in half.*

BINDER Question 183

Nobody has realigned the dish behind a district hospital at Chake Chake since it was bolted to its plinth. It tracks nothing. It stares at a patch of sky above the equator where something has sat still since before the hospital owned a computer.

Describe how a satellite link carries traffic between two earth stations, naming the uplink, the transponder and the downlink.

Model Answer

The earth station transmits on the uplink; a transponder aboard receives, amplifies and shifts the signal to another frequency; and it retransmits on the downlink to every station inside its footprint, so a satellite is a repeater in orbit, 35,786 km up.

The uplink: a dish aimed at the satellite sends a narrow beam upward, in the C band near 6 GHz or the Ku band near 14 GHz. It needs real power and accurate pointing, because the signal must climb 35,786 km.

The transponder: it is the chain aboard that receives, amplifies, shifts and retransmits. One satellite carries many, each with its own slice of bandwidth, and capacity is sold by the transponder or a fraction of one.

Why the frequency changes: were the satellite to retransmit on the frequency it was listening to, its own output would deafen its own receiver. So the downlink is moved, to near 4 GHz in the C band and near 12 GHz in the Ku band.

Downlink and footprint: the downlink reaches every station in the footprint at once, which makes broadcasting to many sites almost free and point-to-point traffic wasteful. Three geostationary satellites spaced 120 degrees apart cover the populated surface.

Why geostationary: at 35,786 km above the equator one orbit takes about 24 hours, so the satellite keeps pace with the ground below and appears fixed. That is why a dish can be bolted down once and never touched.

The low orbit alternative: a satellite a few hundred kilometres up is far closer, so the delay is small, but it crosses the sky in minutes. That demands a constellation and a self-steering antenna, moving the cost from the dish to the fleet.

The dish at Chake Chake is aimed at a point rather than an object, and the point does not move.

The Master Key: *A satellite adds nothing to a network but height, and every advantage and every penalty of the link comes from that height.*

BINDER Question 184

Forty learners in one classroom at Mikocheni open the same page at the same moment, on equipment that behaves faultlessly when the room is empty. The access point is new and its standard is the newest sold. The room, not the equipment, has failed.

Using Table 3.27, compare the 2.4 GHz and 5 GHz Wi-Fi bands on range and on capacity, and state which you would put a crowded classroom on.

Model Answer

2.4 GHz has the longer wavelength, so it travels further and through more walls, but its band is only 83.5 MHz wide, holds three non-overlapping channels and is shared with Bluetooth and microwave ovens; 5 GHz reaches less than half as far indoors but offers more than twenty channels and far less interference, so put the crowded classroom on 5 GHz.

Wavelength decides the reach: the wave is 12.5 cm long at 2.4 GHz and 6 cm long at 5 GHz. Shorter waves are absorbed more heavily by brick, concrete and the water in a roomful of people, so the higher band fades sooner.

Room for channels: capacity comes from having somewhere to put the traffic. The 2.4 GHz band spans only 83.5 MHz, which holds three non-overlapping channels of 20 MHz, numbered 1, 6 and 11, so neighbouring classrooms must share. The 5 GHz band offers more than twenty.

Who else is there: 2.4 GHz is open to all, and Bluetooth, cordless telephones and microwave ovens are in it too, each raising the noise the access point must shout over. The 5 GHz band is comparatively quiet, which is worth as much as the extra channels.

Standard	Band	Nominal rate	Notes
802.11a	5 GHz	54 Mbit/s	Short reach, little interference; never common on cheap equipment
802.11b	2.4 GHz	11 Mbit/s	The longest reach of the early standards, and the slowest
802.11g	2.4 GHz	54 Mbit/s	The rate of a with the reach of b; works with b clients
802.11n	2.4 and 5 GHz	600 Mbit/s	Several aerials carrying separate streams at once
802.11ac	5 GHz	6.9 Gbit/s	Wider channels and more streams; 5 GHz only
802.11ax	2.4 and 5 GHz	9.6 Gbit/s	Designed for many clients in one room, not one fast client

Table 3.27: *The 802.11 wireless standards.*

What the standards add: read the table across. The nominal rate is what the radio manages in ideal conditions with every aerial working, and it is shared among all who associate with the access point. Only ax was designed for a full room, scheduling many small transmissions together.

The classroom decision: enable both bands, put the learners on 5 GHz and leave 2.4 GHz for the far corners and older machines. Fix each access point to channel 1, 6 or 11 by hand and lower the transmit power, so the cells stay small.

The newest standard on the box cannot invent channels the 2.4 GHz band does not have. Capacity in a crowded room is bought with spectrum and smaller cells.

The Master Key: *Reach and capacity sit at opposite ends of the same dial, and every Wi-Fi decision is a choice of where to set it.*

REAL Question 185

A head teacher on Mafia rings the district office and finds herself talking over the person at the other end, then waiting, then apologising, then doing it again. The line is clear. Every word arrives late, and late by the same amount.

Calculate the propagation delay for a signal travelling from an earth station up to a geostationary satellite and back down to a distant earth station, and explain what that delay does to a telephone conversation between Mafia and the mainland.

Model Answer

The signal climbs 35,786 km and descends the same distance, so it travels 71,572 km, and at 300,000 km per second that takes 0.2386 s, about 240 ms; a spoken word and the reply it prompts each make that journey, so the speaker waits nearly half a second, far beyond the 150 ms within which conversation feels natural.

The distance first: the satellite is 35,786 km above the equator. A signal leaving the earth station must climb that height to reach it and descend the same height to reach the far station, so the path is twice the orbital altitude.

$$\text{Path} = 2 \times 35{,}786\ \text{km} = 71{,}572\ \text{km}$$

Then the time: divide that path by the speed of the wave in free space, which is 300,000 km in every second. The journey takes just under a quarter of a second, and no equipment along the way has yet done a thing with the signal.

$$\text{Delay} = 71{,}572{,}000\ \text{m} \div (3 \times 10^8\ \text{m/s}) = 0.2386\ \text{s}, \text{about } 240\ \text{ms}$$

What it does to speech: the reply must make the same journey back, so about 0.48 s passes between the head teacher finishing a sentence and hearing the first word of the answer. Below 150 ms nobody notices a delay; from 150 to 400 ms one way a conversation is workable but stilted; past that, both speakers fill the silence at once and collide.

Echo returns twice over: any of her voice that leaks from earpiece to microphone at the far end comes back after the same 240 ms, late enough to be heard as a separate echo of herself rather than as part of her own speech. Echo cancellation at both ends is essential on a satellite circuit, not an optional refinement.

What actually helps: keep internal calls off the hop and on the local network; agree that one person speaks at a time; and if a low earth orbit service is available, take it, because 550 km up gives a path of 1,100 km and a delay near 3.7 ms, which removes the problem rather than managing it.

What does not help: buying more capacity. Rate and delay are different quantities, and a faster satellite service delivers more bits each second while every one of them still arrives 240 ms after it was sent, because neither the distance nor the speed of light is negotiable.

The head teacher is not on a bad line. She is on a long one, and 71,572 km of it lies above her head.

The Master Key: *Ask how far the signal must travel before blaming the equipment, because distance sets a delay that no amount of bandwidth will pay off.*

REAL Question 186

The page of notes left behind after a new wireless installation at a secondary school in Bariadi is neat, confident and mostly sound. Three of its statements are wrong, and because nothing on the page looks doubtful, they will be quoted for years.

Identify what is wrong in each of the three statements below, give the correct position with the reason, and name the habit that would have prevented all three.

1. Unguided media are of four kinds: radio, microwave, infrared and broadband, and broadband is the one to specify wherever high capacity is wanted.

2. Infrared is the technology behind Bluetooth, and the example to give in an answer is the video cassette recorder remote control.

3. Rain does not affect a microwave link, so the hilltop hop will hold its full rate right through the long rains.

Model Answer

Broadband is a way of using a medium and not a medium at all, so the four unguided categories are radio, microwave, infrared and satellite; Bluetooth is 2.4 GHz radio and the infrared example is IrDA or a television remote control; and rain absorbs and scatters microwaves badly above about 10 GHz, which is precisely why a wet-region link is engineered with a fade margin.

Broadband is not a medium: the word describes how a medium is used, dividing its capacity into several frequency channels that carry different signals at once, as against baseband, where one signal has the whole path to itself. Strike it from the list and the categories read radio, microwave, infrared and satellite. Nobody can specify broadband to a supplier, because there is nothing there to order.

Bluetooth is radio: Bluetooth works at 2.4 GHz in the industrial, scientific and medical band, hopping between narrow channels many times a second. It needs no line of sight and works through a pocket, a bag or a desk, none of which infrared could manage. The infrared example is IrDA, and the everyday one is a television or air-conditioner remote control; a video cassette recorder has not been sold in this country for many years, and an example nobody has seen teaches nothing.

Rain fade is real: above about 10 GHz a raindrop is a substantial fraction of the wavelength, so it absorbs and scatters the beam, and a heavy tropical downpour can take tens of decibels out of a link and drop it altogether. Below 10 GHz the effect is modest, which is why long hops in a wet region are built at 6 or 7 GHz and given a margin of spare signal for the worst hour of the year.

What the three share: each takes a term from one column and files it under another, a signalling method as a medium, a radio technology as an infrared one, a fair-weather figure as an all-weather promise. Ask of any claim what quantity it concerns, in what unit that quantity is measured, and how old the example is.

Correct the page and leave it on the wall rather than removing it. The next technician will read whatever is there, and a note with three corrections written on it is worth more than no note at all.

The Master Key: *Check the units and the date of every confident sentence, because most wireless errors are a true statement standing under the wrong heading.*

HOT Question 187

Two streams take computer studies at a school on Mafia, on forty machines with no connection. Mainland fibre is said to be coming, and has been for three years. A dish, a modem and a contract could carry traffic within a month.

Advise the school whether to buy a satellite service now or wait for terrestrial fibre to arrive, and commit to a recommendation.

Model Answer

Buy the satellite service now, on the shortest contract the provider will write, sized at about 10 Mbit/s and paired with a local content server, because a school cannot suspend two years of teaching on a project whose date it does not control, and a short contract is not a commitment to satellite for ever.

What waiting costs: two full cohorts pass through the school in the time an island fibre project takes to arrive, and the date belongs to a contractor and a budget line the school has no say in. Every year of waiting produces a set of learners examined on work they have never once done on a live connection.

What satellite gives at once: coverage is the thing it is good at. The footprint falls on the island exactly as it falls on the mainland, no channel has to be crossed, no wayleave signed and no third party consulted. The installation is a dish, a mount, a modem and one careful alignment.

What it charges: capacity is metered and expensive, so 10 Mbit/s across forty machines is 0.25 Mbit/s each if all of them work at once. The geostationary delay of 240 ms makes every page feel slow whatever the rate, heavy rain will take the link out for minutes at a time, and the equipment wants stable power behind it.

Spend on the ground: most of what a class needs never has to cross the channel. Put a server in the laboratory holding the teaching material, the reference library and the practice questions, and let the satellite carry mail, updates and the traffic that genuinely must leave the island. That turns a small metered link into an adequate one.

If a low orbit exists: prefer it where the coverage reaches Mafia and the ground equipment can be afforded. The delay falls from 240 ms to a few milliseconds, so the link behaves like a terrestrial one and the delay argument disappears from the decision altogether.

The judgement and the exit: buy now, keep the term short, and write the fibre into the plan rather than waiting on it. When it lands, the satellite service drops to standby for the days the fibre is cut, and the local server stays exactly where it is. Waiting is not a neutral option; it is a decision to teach two cohorts without a connection.

The fibre may well arrive next year, and it will still be the better link when it does. It is not, however, the link this year's Form Five will ever use.

The Master Key: *Buy the connection that exists today and design it so the better one can replace it tomorrow without wasting a shilling.*

HOT Question 188

Two hills near Njombe stand 12 km apart over a valley of planted pine. A college on one wants the office on the other inside its network. One quotation offers two dishes and two mornings; the other follows the road for 19 km with a trench.

Judge whether a microwave link between the two hills is a sound choice, and defend your answer against the alternative.

Model Answer

It is a sound choice and the link should be built, provided a path survey shows the first Fresnel zone at least 60 per cent clear of the plantation, the hop is engineered at about 6 GHz with a fade margin for highland rain, and tree growth is written into the maintenance plan.

Why the geography suits it: 12 km from hilltop to hilltop with the ground falling away between is close to an ideal microwave path. The hills supply the height, so the masts are short, and nothing need be negotiated with the owners of the land between, because the beam passes above them.

Sight is not enough: the beam is a volume rather than a line. The first Fresnel zone, the ellipse drawn about the direct ray, reaches roughly 12 m out from it at the midpoint of this hop, and 60 per cent of that must stay clear, so pine tops well below the sight line still take signal out of the link.

The rain of the highlands: the rains here are heavy and long. Above about 10 GHz raindrops absorb and scatter the beam severely, so specify 6 GHz, buy reach as antenna gain rather than transmitter power, and set the fade margin against a stated availability, remembering that 99.99 per cent still permits about 53 minutes of outage a year.

Multipath, the hidden fault: a second copy of the signal, reflected off the flat valley floor or off standing water, arrives slightly later than the direct one and can partly cancel it. The symptom is a link that fails at particular times for no visible reason; the cure is to shift one antenna height, or fit two and use whichever is stronger.

What the trench would buy: capacity that never fades, indifference to weather and no alignment to check. It would also buy 19 km of digging along a road the college does not own, a wayleave, a contractor and a cable a grader will eventually cut, all to join two points already in sight of one another.

The judgement, with conditions: build the microwave link. Make the survey a condition of payment, allow ten years of tree growth in the clearance figure, and book an annual visit to check alignment and cut back what has grown. If the survey cannot deliver that clearance, the answer reverses and the trench becomes the cheaper decision.

A microwave hop is not a cheap fibre but a different bargain: it crosses the valley in two mornings and charges for it in survey, margin and a view kept clear for as long as the link is wanted.

The Master Key: *A wireless hop is paid for in clearance rather than in distance, so survey the path before pricing the radios.*

Structured cabling: T568-A and B, the toolkit and the test

A wiring instruction that names a standard without printing it teaches nothing, so this section prints both colour orders in full, pin by pin, and again as a figure. It then settles which cable to make for which job, why the crossover cable has almost disappeared, what tools the work needs, and what a test does and does not prove. It assumes you know that a twisted pair carries its signal as the difference between two conductors.

BINDER Question 189

The practical sheet pinned above the bench at a secondary school in Bariadi runs to eleven steps. Step four says "arrange the conductors to T568-B", and step nine asks for T568-A. Nowhere is a single colour printed. The class cannot begin.

State the T568-A and the T568-B colour orders pin by pin, and identify the pins at which the two differ.

Model Answer

Reading from pin 1 to pin 8, T568-A is white-green, green, white-orange, blue, white-blue, orange, white-brown, brown, and T568-B is white-orange, orange, white-green, blue, white-blue, green, white-brown, brown; the two differ at pins 1, 2, 3 and 6, and nowhere else.

Four pairs, eight conductors: the jacket holds four twisted pairs, orange, green, blue and brown. Each pair is one solid conductor and its partner striped white in the same colour, which makes eight in all, and the RJ-45 plug has exactly eight positions to take them.

How the pins are numbered: hold the plug with the locking clip underneath, the contacts uppermost and the nose pointing away from you. Pin 1 is then the contact on your left and pin 8 the one on your right. Number them the other way and every conductor lands mirrored.

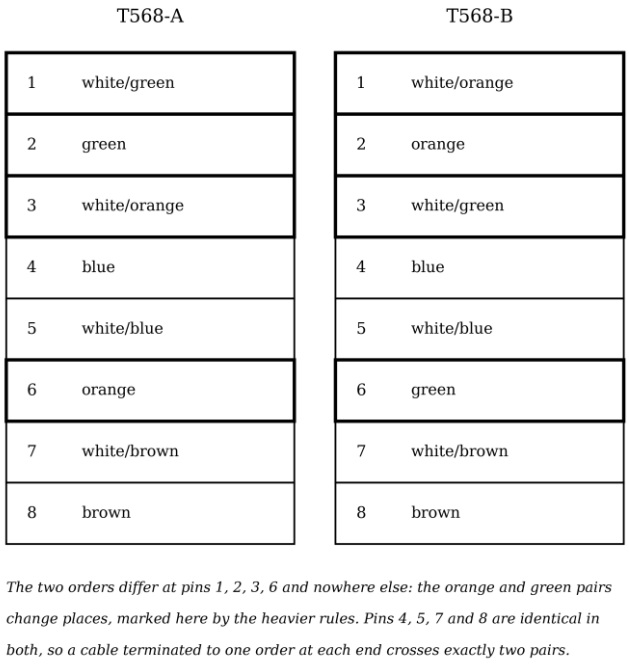


Figure 3.33: *The T568-A and T568-B colour orders, pin by pin. The heavier rules mark the four pins at which they differ.*

The four pins that differ: at pins 1, 2, 3 and 6 the orange and the green pair change places. T568-A gives green pins 1 and 2 and orange pins 3 and 6; T568-B gives orange pins 1 and 2 and green pins 3 and 6. That one exchange is the whole difference.

The four pins that agree: pins 4 and 5 carry the blue pair and pins 7 and 8 the brown pair in both orders, so half the plug is the same either way. Note that pins 3 and 6 are one pair although blue sits between them: a pair is not always two neighbours.

One order for one building: either standard makes a working cable, provided both ends of a straight-through match. Choose one, write it inside the cabinet door, and terminate every outlet and panel to it.

Learn one order and derive the other by swapping the orange and green pairs and leaving blue and brown where they are. That is far less to remember than sixteen colours, and it cannot be half remembered.

The Master Key: *Pins 4, 5, 7 and 8 never move; everything that separates T568-A from T568-B happens at pins 1, 2, 3 and 6.*

BINDER Question 190

Asked for a crossover cord, an apprentice at a district office in Kondoa reaches for a knife to swap two conductors. The technician stops him: nothing may be cut, each plug must follow a published standard, and the cord must still be a crossover.

Using Figure 3.34, explain how a crossover cable is produced, and state which pairs it crosses for 100BASE-T.

Model Answer

A crossover is made by wiring T568-B at one end and T568-A at the other; because the two orders differ at pins 1, 2, 3 and 6, that alone joins pin 1 to pin 3 and pin 2 to pin 6, which is the crossing that 10 and 100BASE-T require.

Nothing is crossed by hand: no conductor is swapped inside the jacket. Each plug is wired faithfully to its own standard, and the difference between the standards performs the crossing, which is why the two orders are worth knowing rather than the wiring of one special cable.

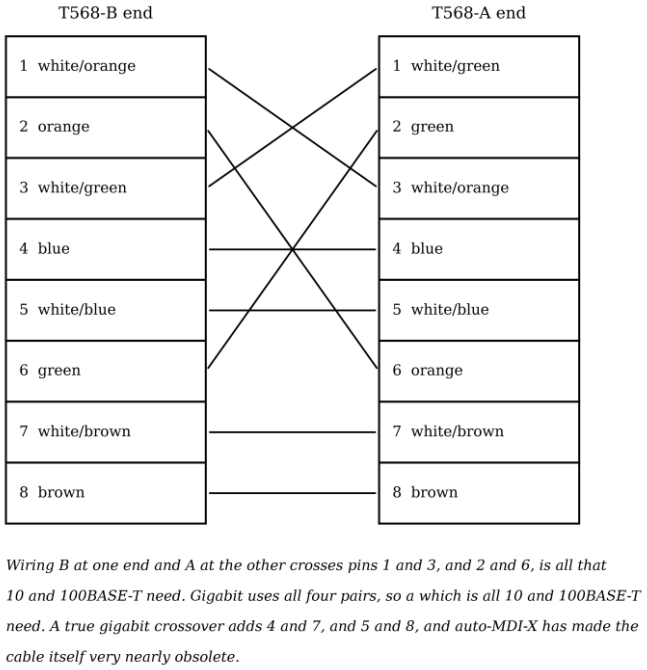


Figure 3.34: *A crossover cable, derived from the two colour orders. Wiring B at one end and A at the other crosses pins 1 and 3, and pins 2 and 6.*

What the crossing achieves: 10 and 100BASE-T transmit on pins 1 and 2 and receive on pins 3 and 6, using two of the four pairs. Bringing the transmit pair of one device to the receive pins of the other is the whole purpose; pins 4, 5, 7 and 8 pass straight through, unused.

Its own inverse: pin 1 goes to pin 3 and pin 3 comes back to pin 1, pin 2 to pin 6 and pin 6 to pin 2. The crossing is symmetrical, so neither plug is the near end and the cord works the same way round either way.

Gigabit crosses all four: 1000BASE-T uses all four pairs at once, each carrying in both directions, so a true gigabit crossover adds pin 4 to pin 7 and pin 5 to pin 8. A cord that crosses two pairs only is not a gigabit crossover, whatever the label says.

Why nobody makes one: auto-MDI-X, mandatory in 1000BASE-T and fitted to almost every port sold, senses which pins the far end transmits on and crosses inside the port. The crossover cable is now very nearly obsolete.

Tell the two apart by sight. Hold both plugs contacts up and clips down and compare pin 1: two white-orange means straight-through, and one white-orange against one white-green means crossover.

The Master Key: *A crossover is not a cable you cross; it is one standard at each end, and the standards do the crossing.*

BINDER Question 191

A box in the store cupboard of a school in Nzega holds thirty patch cords, four of them labelled crossover in pen. Nobody now working there made them, and nobody knows whether the pen came after a check or after a guess.

Distinguish a straight-through cable from a crossover cable, and state where each was traditionally used.

Model Answer

A straight-through cable is wired to the same standard at both ends, so every pin joins the pin of the same number; a crossover is wired T568-A at one end and T568-B at the other; straight-through cables traditionally joined unlike devices and crossovers joined like devices.

The straight-through: both plugs to B, or both to A; which one is chosen makes no difference so long as they match. Pin 1 joins pin 1 and so on to pin 8. This is the ordinary cable, and the one to make unless there is a reason not to.

The crossover: one plug to A and one to B, which joins 1 to 3 and 2 to 6, and for gigabit 4 to 7 and 5 to 8 as well. It is the same eight conductors and the same crimp; only the order at one end has changed.

Unlike against like: a computer transmits on pins 1 and 2 and a switch port receives on them, so between unlike devices a straight cable already brings transmit to receive. Two devices of the same kind transmit on the same pins, and something has to cross.

Cable	Wiring	Pins crossed	Traditionally used between
Straight-through	Same order at both ends, B to B or A to A	None; pin 1 to pin 1, and so on to pin 8	Unlike devices: computer to switch, switch to router
Crossover	T568-B at one end, T568-A at the other	1 with 3 and 2 with 6; gigabit adds 4 with 7 and 5 with 8	Like devices: switch to switch, computer to computer

Table 3.28: *Straight-through and crossover cables.*

Auto-MDI-X ends the rule: the port now senses which pins the far end transmits on and swaps internally. Since the feature is mandatory in 1000BASE-T, two modern devices of any kind link over a straight-through cord, so the like and unlike rule is history to recognise, not a rule to apply.

Treat those four marked cords as unknown until a wire-map tester has read them. A crossover between two modern switches works perfectly, which is exactly why it can sit in a cabinet for years and be found only when it meets old equipment.

The Master Key: *The same standard at both ends joined unlike devices, different standards joined like ones, and the port now does either for you.*

BINDER Question 192

The toolbox delivered with a new cabinet at a college in Tarime holds a crimper and a bag of plugs, nothing else. Half the faults that follow are made by a stripper set too deep, and the other half by a tester nobody bought.

Using Table 3.29, name six of the tools a cabling job needs and state what each one does.

Model Answer

Six that no installation can manage without are the jacket stripper, the flush cutter, the crimper, the punch-down tool, the tone generator with its probe, and the wire-map tester: the first two prepare the conductors, the next two make the two kinds of termination, and the last two find a cable and prove it.

Preparing the cable: the stripper scores the outer jacket only, at a depth set once and tried on a scrap end. The flush cutter takes the eight conductors off square in one stroke, so that all eight can reach the front of a plug together.

Two terminations, two tools: a crimper closes an RJ-45 plug onto the end of a cord. A punch-down tool seats a conductor into the sprung contact of a jack or a patch panel and trims the tail in the same movement. Neither will do the other's work.

Finding a cable: the tone generator puts an audio signal on one conductor and the probe sounds when held near it, which is how one run is picked out of a bundle of forty that nobody labelled. A fish tape does the earlier job of drawing that cable through conduit.

Proving a cable: a continuity tester says that something is joined end to end. A wire-map tester says which pin reaches which, and so catches a reversal, a transposition and an open circuit. The price difference is small; what you learn from it is not.

Tool	What it does	When you reach for it
Jacket stripper	Scores and removes the outer jacket without cutting insulation	Before every termination
Flush cutter	Cuts the cable and trims the eight conductors square	At the cut, and again before the plug
Crimper	Closes an RJ-45 plug onto a cord	Making a patch cord
Punch-down tool	Seats a conductor in a sprung contact and cuts the tail	Wiring a jack or a patch panel
Tone generator and probe	Puts a tone on one end and finds it at the other	Identifying a run in an unlabelled bundle
Continuity tester	Shows that each conductor is joined end to end	A first check on a finished cord
Wire-map tester	Shows which pin reaches which, and reports splits and reversals	Before any cable goes into service
Fish tape	Pushes through a conduit and pulls the cable back	Getting cable through a wall or a duct

Table 3.29: *The cabling toolkit.*

Where the cable ends: the tools serve hardware: a wall plate holding the jack at the desk, a patch panel holding the other end of every horizontal run, patch cords at both ends, and conduit, tray or raceway carrying the cable between them.

Buy the wire-map tester before the second crimper. It is the only item on the list that tells you the truth about work already finished by somebody who has left.

The Master Key: *A crimper finishes a plug and a punch-down tool finishes a socket, and reaching for the wrong one damages both.*

REAL Question 193

The counter at a mobile-money shop in Kariakoo has moved three metres from the wall, and no cord nearby is that long. On the shelf are a drum of stranded Cat 6, plugs for stranded conductors, and a crimper. The shop opens in an hour.

Describe, step by step, how to terminate one end of that cord to T568-B, naming the tool used at each step and the check made before the plug is crimped.

Model Answer

Cut generously with the flush cutter, strip about 25 mm of jacket without touching the insulation, remove the spline, fan the pairs into the B order, keep the untwist inside 13 mm, trim all eight square, seat them to the front of the plug with the jacket under the strain relief, read the colours back through the plastic, and only then crimp.

Measure and cut: allow the route the cord will take plus slack at each end, and cut with the flush cutter rather than with a blade, so that the end is square and the eight conductors start the same length.

Strip the jacket only: set the stripper to score the jacket, turn it once, and slide the jacket off whole. Bright copper anywhere means a nicked conductor, which becomes an open circuit the first time the cord is moved, so cut back and start again. Remove the ripcord and the plastic spline that separates the pairs in Cat 6.

Fan into the order: untwist each pair only as far as the plug demands and lay the eight flat, left to right, in the B order: white-orange, orange, white-green, blue, white-blue, green, white-brown, brown.

The 13 mm rule: no more than 13 mm of any pair may be left untwisted where it enters the plug. Past that the two conductors stop cancelling each other, crosstalk into the neighbouring pairs climbs, and a Cat 6 cord cannot spare it. Pull the fan back until the jacket almost touches the plug body.

Trim, seat and inspect: cut the eight square so that they reach the front of the plug together, hold the plug clip down and slide them home, then look through the clear body and check three things: pin 1 is white-orange, all eight touch the end, and the jacket has passed under the strain relief.

Crimp, then test: one firm squeeze drives the eight blades through the insulation and closes the strain relief onto the jacket. A crimped plug cannot be opened, so a wrong colour costs a plug and a centimetre of cable. With both ends made, a wire-map tester should read eight straight connections.

The inspection through the plastic is the most valuable step on the list, because it is the last moment at which the mistake is free. Every step after it is irreversible.

The Master Key: *Read the eight colours through the plug before you squeeze, because the crimp is the one step you cannot undo.*

REAL Question 194

A cord made at the bench in a laboratory in Msimbazi lights all eight lamps of the tester, in order, and is signed off. Plugged between two switches that both offer gigabit, the link comes up at 100 Mbit/s and will not go faster.

Determine the most likely cause of that behaviour and describe how to prove it.

Model Answer

The likeliest cause is a split pair: all eight conductors do reach the same pin at the far end, so the tester is satisfied, but two of them have been taken from different twisted pairs, so pins 3 and 6 no longer travel as a pair and the crosstalk that follows holds the link at 100 Mbit/s.

Why the lamps pass: a continuity tester asks one question of each conductor: does it reach the same pin at the other end? A split pair answers yes eight times. Even a plain wire map passes, because the map is correct; the fault is not in which pin but in which twist.

What a split pair costs: twisting is what makes interference arrive equally on both conductors, where the receiver subtracts it away. Two conductors from different pairs are not twisted together, so that cancellation is gone and near-end crosstalk rises. Gigabit, which drives all four pairs in both directions at once, is the first thing to fail.

The other candidates: check the cheap ones as well: more than 13 mm of untwist at a plug, a conductor nicked by the stripper, a channel longer than 100 m, a port left configured at 100 Mbit/s, or a plug meant for solid conductors crimped onto stranded cable.

How to prove it: substitute a factory-made cord of the same length. If the link trains at 1000 Mbit/s, the ports are innocent and the cord is guilty. Then read the suspect cord on a tester that reports split pairs, or on a certifier that measures near-end crosstalk against the Cat 6 limit and names the pair at fault.

The repair and the record: cut both plugs off and terminate again, taking each pair to its own two pins: 1 and 2, 3 and 6, 4 and 5, 7 and 8. Test before the cord goes back, and write the date on it, because an untested cord that has failed once will be suspected forever.

This is the commonest fault a bench produces, and the reason is that the tester on the bench measures the wrong quantity. Continuity is not performance, and eight lamps in a row prove only that eight conductors arrived.

The Master Key: *Continuity tells you the copper reached the far end; only crosstalk tells you it travelled as a pair.*

REAL Question 195

Written on the last day at a district hospital in Ilala, a handover note is left for the assistant taking over the network. Three of its instructions are set down with complete confidence, and the assistant, doubting none of them, will follow all three.

Identify what is wrong with each statement below, give the correct position with its reason, and say why each is a tempting thing to write.

1. A modern laptop will not link to a modern switch over a straight-through cable, so make a crossover for every laptop point.

2. If a cord passes on the continuity tester it is fit for gigabit, and no further test is needed.

3. Untwisting about 40 mm of each pair at the connector makes the plug much easier to load and does no harm.

Model Answer

A computer and a switch are unlike devices and have always taken a straight-through cable, and auto-MDI-X has since removed the question entirely; continuity shows only that each conductor arrived, not that it arrived as a twisted pair; and the untwist at a plug is limited to 13 mm, so 40 mm is more than three times over.

The first instruction: it is wrong twice. Under the old rule the crossover joined like devices, two switches or two computers, while a computer to a switch is the unlike case and takes a straight-through. The rule itself is then dead, because auto-MDI-X is mandatory in 1000BASE-T and fitted to essentially every port sold, so the switch crosses internally whatever cord it is given. The temptation is that crossover sounds like the special cable for the awkward job, and a laptop beside a switch looks like an awkward job.

The second instruction: the continuity tester lights a lamp when a conductor is joined end to end. It knows nothing of which twist that conductor came from, of how far the pairs were unwound at the plug, or of crosstalk at 250 MHz, so a split pair passes it and then fails at gigabit. Prove a cord on a wire-map tester and prove an installed link on a certifier. The temptation is that a test which can fail feels like a test which can pass.

The third instruction: 13 mm is the limit and 40 mm is more than three times it. The twist is what makes interference arrive on both conductors together, where the difference cancels it; unwind the pair and it both radiates into its neighbours and picks up from them. Loading really is easier with straight conductors, which is the trouble: the shortcut is rewarded at the bench and punished on the network months later.

The cure for all three: every one of these sentences treats a cable as either connected or not connected. Each is answered instead by a quantity: which pin reaches which pin, how much crosstalk at what frequency, how many millimetres left untwisted. Write a handover note in numbers and none of the three survives being written down.

Correct the note and leave it in the cabinet rather than throwing it away. The assistant will follow it exactly, which is what a handover note is for, and that is precisely what makes a confident wrong sentence expensive.

The Master Key: *Confidence in a handover note is not evidence; make every sentence in it produce a number.*

HOT Question 196

Two afternoons of practical time at a secondary school in Mikocheni are set aside for crossover cables. The kit is bought and the lesson is written. Every cable the class makes will be perfectly good, and not one of them will ever be needed.

Assess whether a school should still teach its students to make crossover cables, and commit to a recommendation.

Model Answer

Keep teaching what a crossover is and why the two colour orders differ, but stop teaching its manufacture as a live skill: auto-MDI-X has made the cable very nearly obsolete, and the two afternoons buy far more if they are spent on terminations the student will actually make.

What has changed: auto-MDI-X is mandatory in 1000BASE-T and is fitted to almost every switch, router and computer port now sold. The port senses which pins the far end transmits on and crosses inside itself, so two switches link happily over an ordinary straight-through cord.

Silence is worse than omission: a student taught to make a crossover and not told this leaves believing the cable is required. One day he will make one, label it, and leave it in a cabinet for the next person to puzzle over. Teaching the cable without the caveat teaches a rule that is false.

What is still worth knowing: the definition itself: A at one end and B at the other, crossing 1 with 3 and 2 with 6, with 4 to 7 and 5 to 8 added for gigabit. It explains what the port is doing on your behalf, it is asked for by examiners, and it costs one diagram and five minutes.

Where one is still needed: a few old switches and managed devices have a fixed port with no detection, and some point-to-point links on equipment made before the feature existed still want the cable. A school that owns such equipment should keep exactly one crossover, labelled, in the cabinet.

What the afternoons should carry: punching down a jack and a patch panel, making and testing a straight-through cord, reading a wire map, dressing and labelling both ends. Those are the skills the school needs the next time a room is rewired, and they produce cables that go into service.

The judgement: teach the concept in one lesson with the two colour orders on the board, make a single crossover as a demonstration, label it and keep it, and give the practical time to terminations. Do not set a class exercise whose product has no use.

The measure of a practical is whether the thing made goes into service. A cable made to satisfy a rule that no longer applies goes into a drawer, and teaches the class that cabling is a ritual.

The Master Key: *Teach the crossover as an idea and not as a product, because the ports have been doing the crossing themselves since gigabit arrived.*

HOT Question 197

Sixty patch cords are needed for a new laboratory at a secondary school in Kahama. A supplier quotes for sixty moulded cords. The school already owns a crimper and a drum of cable, and the head of department wants the students to make them.

Judge whether the school should buy factory-made patch cords or make its own, and defend the recommendation.

Model Answer

Buy the sixty cords. A patch cord is the one part of a cabling system that is bent, pulled and trodden on daily, the factory cord is moulded, strain-relieved and tested before it is boxed, and the school's crimper is better kept for terminating fixed runs and repairing what breaks.

Solid and stranded differ: the drum on the shelf is almost certainly solid-core cable, made to be punched down into a jack and never moved again. A cord flexed at a desk needs stranded conductors, and the plug for stranded has different contacts from the plug for solid. Crimp the wrong plug on and it passes the tester and fails in a month.

What the factory adds: a moulded boot that keeps the latch from snapping and stops the cable bending sharply where it leaves the plug, an untwist held well inside 13 mm on every cord, and a test at the end of the line. The rejects never reach the box.

What making them costs: plugs, of which a proportion will be wasted; time, at perhaps ten minutes a cord for a learner, which is ten hours for sixty; and the ones that fail their test and are made twice. None of that appears in the quotation the head of department is comparing against.

Where making them is right: an odd length no supplier stocks, a repair at five in the evening with the office still working, and the training itself, which is a genuine skill for the fixed cabling. Keep the crimper, a box of plugs and the wire-map tester for exactly those three.

The false saving: a practical class is paid for out of teaching time, so cords made in class are not free; they are bought with a lesson. Decide what goes into the laboratory on its merits, and buy the training separately, on its own.

The judgement: buy the sixty moulded cords, in two or three lengths so that none has to be coiled under a desk, and hold a kit of fifty plugs, a crimper and a tester for repairs and odd lengths. Let the class make cords, test them, and put the good ones into service.

The rule worth carrying away is one of movement. What is plugged and unplugged every week should be bought and tested; what is fixed inside a wall should be terminated on site by the person who put it there.

The Master Key: *Buy the cord that moves and terminate the cable that does not.*

HOT Question 198

Pacing the corridor gives 74 metres. The library at a district office on Mafia stands at the far end of a single-storey building, and the only cabinet is beside the entrance. The electrician measured the route, up into the ceiling and back down, at 96 metres.

Determine the greatest distance at which a workstation may sit from its floor switch under the structured-cabling rules, and assess what should be done when a room lies beyond it.

Model Answer

The horizontal cable from patch panel to outlet may be 90 m measured along the route it actually takes, with up to 10 m of patch cords at the two ends completing a channel of 100 m; a 96 m route therefore fails before a cord is plugged in, and the answer is a switch nearer the library rather than a longer cable.

What the hundred metres holds: 90 m of fixed horizontal cable plus up to 10 m of patch cords at the two ends, and 90 and 10 make the 100 m channel. The limit is a channel measured from equipment to equipment, not a length off a drum, and it holds for every category of twisted pair.

Measured along the route: cable does not travel in straight lines. Allow the rise into the ceiling and the drop to the outlet, perhaps 6 m in a single-storey building, and slack and a service loop at each end, perhaps 4 m more. The 90 m then leaves about 80 m of distance on the floor plan.

Why this route fails: 96 m is 6 m past the horizontal limit before the two cords are counted, and the cords are not optional. An over-length link often trains at a lower speed, or carries traffic until the day it stops, which costs more to find than an honest failure on the first morning.

What will not rescue it: a coupler joining two drums still leaves one channel of the same length and adds loss at the join. A repeater in a ceiling void needs power and maintenance nobody will give it. A longer patch cord comes out of the same 10 m allowance.

What does rescue it: put a second cabinet near the library and feed it from the first with fibre, which carries no such limit, then run short horizontal cable from there. Where a full cabinet is too much, a small switch in a locked wall box on a fibre uplink does the same work.

The judgement and the practice: extend the backbone, never the horizontal. Site the new cabinet so its furthest outlet is inside 90 m of route, keep data cable 50 mm clear of mains and cross the two at a right angle, respect a bend radius of four times the cable diameter, and label both ends of every run.

Ninety metres is not a distance on a plan. It is a length of cable that has already gone up, along and down, so measure the route before promising anybody the room.

The Master Key: *When the room lies beyond reach, move the switch to the room, because copper has one length and it does not negotiate.*

LAN design, documentation and simulation

A network designed on paper costs an afternoon; a network designed while it is being installed costs the budget twice. This section sets out the four steps that carry a school from a conversation with the head teacher to a folder a contractor can build from, the three-layer model that shapes almost every campus network put up today, the documents that must outlive the person who wrote them, and the one question a simulator answers honestly. It assumes you can already subnet a block, cut a switch into VLANs and read a distance limit off a cable specification.

BINDER Question 199

Cabling money and a fortnight of the holiday are in hand at a secondary school in Mikocheni, and the contractor begins on Monday. Nobody has written down the number of machines, the distance to the furthest building, or who stays off the office server.

Describe the four steps of local area network design, and state what each of them must produce.

Model Answer

The four steps are to establish what the client needs, to survey what already exists, to draw the logical design and then to document and present it; each ends in a written product, and the fourth hands over the folder the installation is built from.

Establish the requirement: turn what the users say into countable statements: how many machines now, how many in five years, which applications must work, what must not be reachable from where, and what may be spent. The product is a requirements list the client has signed.

Survey what exists: walk the site with a measuring wheel. Record the buildings, the distance from each to the cabinet, the cabling already in the walls, where power is reliable and where a cabinet can be locked. The product is a survey of measured lengths.

Draw the logical design: settle the topology, the address plan, the VLAN scheme, the routing and the naming convention. None of it mentions a room, and all of it can be corrected before anything is bought. The product is a logical diagram and an address table.

Document and present: turn the logical design into a physical one carrying cabinet positions and cable routes, add the labelling scheme, the bill of quantities and an empty change log, and put the set before the client. The product is the folder the contractor builds from.

Why the order holds: every step consumes the one before it. A design drawn before the survey commits the school to a copper run the distance forbids, and a survey made before the requirement measures the wrong buildings. The sequence is the first thing an impatient contractor abandons.

Step	What you do	What you hand over
1. Establish the requirement	Count users, list applications, fix the budget and the date	A requirements list the client has signed
2. Survey the site	Measure distances, record cabling, power and cabinet space	A survey with measured lengths
3. Logical design	Topology, addressing, VLANs, routing, naming	A logical diagram and an address plan
4. Document and present	Physical drawing, labelling, quantities, approval	The folder the installer builds from

Table 3.30: *The design steps and what each must produce.*

A fortnight at Mikocheni is time enough to install a network. It is not time enough to design one as well.

The Master Key: *End every step with something written, because a step that produced no document was never taken.*

BINDER Question 200

Both switches in a cabinet at a college near Makambako attach users, route between networks, hold the filtering rules and uplink to each other. When one is replaced next term, nobody can say which of its four jobs moves with it.

Using Figure 3.35, explain the hierarchical design model and state the single job of each of its three layers.

Model Answer

The model cuts a campus network into three layers and gives each exactly one job: the core carries traffic between distribution blocks as fast as it can, the distribution layer routes between VLANs and applies every policy the design calls for, and the access layer attaches the users, printers and access points.

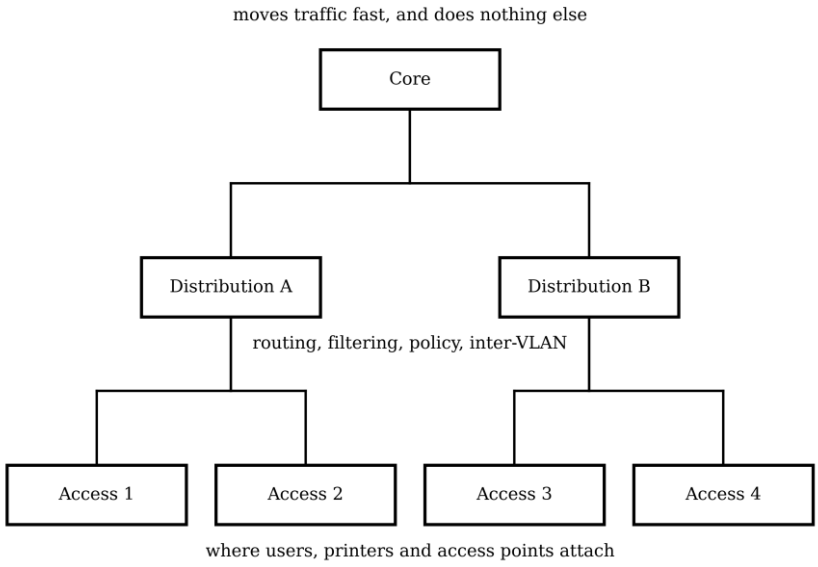
The core: forwarding, and nothing else. No filtering and no rule of any kind, because every rule added there delays all traffic at once. A core is built from few devices, few links and identical configurations, and the only sophistication it is allowed is redundancy, since everything crosses it.

The distribution layer: the boundary at which policy lives. Routing between VLANs happens here, so does filtering, so does the summarisation of routes towards the core, and so does the aggregation of the access switches below. A school keeping students out of the office network buys that rule here.

The access layer: port density, and everything that touches a user. Ports are assigned to their VLANs here, power over Ethernet feeds the access points here, and port security is applied here. It holds the most devices of the three and the cheapest.

Why the jobs are separated: so that each layer can be sized, replaced and diagnosed on its own. Adding a building adds one access switch and one uplink and changes nothing above it, and a fault at the access layer costs a room rather than a campus.

Reading the figure: four access switches feed two distribution switches, which feed a single core. Traffic between two rooms under the same distribution switch never reaches the core, and that is the property which keeps a core idle enough to stay fast.



Three layers, each with exactly one job. A network built without a model ends as a heap of switches nobody can reason about.

Figure 3.35: *The hierarchical design model. Three layers, each with exactly one job.*

The layers are jobs and not boxes, and a small site may run two of them on one device provided the drawing says which is which.

The Master Key: *Ask of every device which one of the three jobs it holds, and distrust any that has taken two.*

BINDER Question 201

Eight offices, and a proposal to run a cable from each one to every other: the district council in Rufiji is told that a network with no single point of failure is obviously the better network. The proposal is priced by the cable drum.

Using Figure 3.36, compare the star, ring and mesh topologies on cost, on resilience and on the number of links each of them requires.

Model Answer

A star of n nodes needs n links to a central switch and is the cheapest to build, at the price of a switch on which everything depends; a ring also needs n links and survives one break; a full mesh needs n(n − 1)/2 links and grows as the square, so eight offices want 8 × 7 / 2 = 28 links where a star wants 8 × 1 = 8.

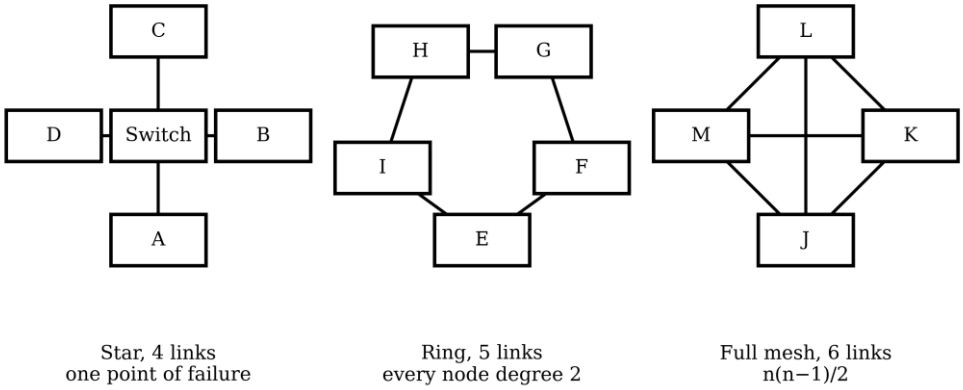
The star, as drawn: four nodes and four links, each running to the central switch. Cabling is simple, a fault is confined to one lead, and a fifth node is one more cable. Everything, though, depends on that switch.

The ring: five nodes and five links, every node of degree 2. One break leaves a path the other way round, so a ring tolerates a single failure where the star tolerates none, and a second break splits it. Adding a node means opening the ring.

The full mesh, counted: four nodes and six links in the figure, from n(n − 1)/2 with n = 4, so 4 × 3 / 2 = 6. Raise the count to eight and the formula gives 28, and every node then needs 8 − 1 = 7 interfaces. Cost climbs as the square.

The comparison that settles it: at eight nodes star and ring both want 8 links and the mesh 28, which is 28 − 8 = 20 more runs to pay for. The star is cheapest and least resilient, the mesh dearest and most resilient, and the ring buys one failure of tolerance at about the star's price.

What is actually built: an extended star, which is a star of stars: access switches home-run to a distribution switch, and that switch to the core. Meshing is kept for the few links between distribution and core, where a whole building rides on one cable.



The degree of every node is checked against the definition of the topology before the drawing is allowed to save.

Figure 3.36: *Star, ring and full mesh, with the number of links each needs.*

The Rufiji proposal is not wrong about resilience. It is wrong about where to buy it, which is on the three or four links that carry everything.

The Master Key: *Work out n(n − 1)/2 before anyone argues about resilience, because the drum is priced by the metre.*

BINDER Question 202

One sheet of A3, four boxes and three lines: the contractor left nothing else at a secondary school in Tarime. Eighteen months later a socket in the library stops working, and nobody can say which patch panel port it lands on.

State the documents that a completed network design should hand over, and explain what each of them is for.

Model Answer

Five documents, none of them optional: a logical diagram, a physical diagram, an address plan, a labelling scheme and a change log; and to those add a copy of every device configuration, so that the school could rebuild its network from paper if the equipment were destroyed tonight.

The logical diagram: what talks to what. Subnets, VLANs, gateway addresses, the routing between them and the link to the outside world. It carries no distances and no room names, because none of that alters what the network does.

The physical diagram: where everything is. Buildings, cabinet positions, the route each cable takes, its measured length and the port at either end. This is the drawing a technician carries up a ladder, and the one that decides whether a new run is possible.

The address plan: one row for every subnet, giving the network address, the prefix, the gateway, the range issued automatically and the range reserved for servers and printers that must never move. Without it a second technician invents a plan of his own, and both are then true.

The labelling scheme: a rule giving every socket, patch panel port and cable end one identifier, and putting that identifier on both diagrams. Building, floor, room and socket in a fixed order is enough. A label that cannot be worked out from the rule must be looked up.

The change log: dated lines recording who altered what and why. It is the only document that keeps the other four honest, because each of them begins to lie on the day of the first change made without being written down.

The library socket at Tarime is not a difficult fault. It is an easy fault made expensive by four documents that were never produced.

The Master Key: *Hand over the drawings and the log together, because a drawing without a log is a photograph of one afternoon.*

REAL Question 203

Four buildings, a room of new machines, a server the office paid for: a school at Nzega is to be networked over the holiday, with a promise that no student will reach that server. Nothing is ordered, and every choice falls in one week.

Design the network for the school specified below, choosing a topology, an address plan and a VLAN scheme, and justify each choice.

Users: 120 student positions, 45 teaching staff, 15 administration staff; 180 in all.

Buildings: administration block, holding the main cabinet and the server room; laboratory block; library and classroom block; staff block.

Measured from the main cabinet: laboratory 70 m, library 140 m, staff block 260 m.

Servers and management devices: 8, all of them in the administration block.

The school holds no registered address space and is not applying for any.

Model Answer

Build an extended star centred on a layer-3 switch in the administration cabinet, with one access stack in each building and three inter-building links, of which the 70 m run is copper and the 140 m and 260 m runs are multimode fibre; address the school from private space, taking the /16 at 172.16.0.0 and giving each of four function-based VLANs its own /24.

The topology, and why: an extended star. Every building keeps a cabinet and every cabinet home-runs to the distribution switch, so four buildings need $4 - 1 = 3$ inter-building links where a full mesh would want $4 \times 3 / 2 = 6$. A break then costs one building, and a fifth building is one more fibre rather than a redesign.

Media chosen by distance: the 100 m limit on a twisted-pair channel decides two of the three runs by itself. The laboratory at 70 m stays on copper with $100 - 70 = 30$ m in hand; the library and the staff block are $140 - 100 = 40$ m and $260 - 100 = 160$ m beyond it, so both take multimode fibre.

Address space, counted first: usable hosts on a subnet are $2^h - 2$, where h is the number of host bits. Left flat, $120 + 45 + 15 = 180$ users would need $2^8 - 2 = 254$ addresses, since a /25 offers $2^7 - 2 = 126$ and falls $180 - 126 = 54$ short. Nothing narrower than a /24 would serve this school.

$$\text{Usable hosts} = 2^h - 2, \text{so a /24 gives } 2^8 - 2 = 254 \text{ and a /25 gives } 2^7 - 2 = 126$$

One /24 to each VLAN: take the /16 at 172.16.0.0 out of the private range reserved at 172.16.0.0/12, and let the third octet carry the VLAN number, so that 172.16.10.0/24 is VLAN 10. Sizing each subnet to its requirement is what a scarce block demands; this one is not scarce, and a /25 would leave the students $126 - 120 = 6$ spare where the /24 leaves $254 - 120 = 134$.

Four VLANs by function: students on 10, staff on 20, administration on 30, servers and management on 40, each with its gateway at .1. The groups follow the rights and not the roofs, so the staff VLAN spans all four buildings; every access switch trunks the four upward, and the distribution switch routes between them and denies VLAN 10 any path into VLAN 40.

Ports, with growth counted: 180 users and 8 servers and management devices need 188 access ports today. Allow a quarter for growth, $188 \times 1.25 = 235$, and ten switches of 24 ports supply $10 \times 24 = 240$. Buy them now: the cabinet is being sized this week.

The promise made to the head of department is kept by one line of configuration at the distribution switch, and only because the VLAN scheme was drawn before the first cable was cut.

The Master Key: *Count the users, then the addresses, then the metres, and let each answer close off the choices the next one may make.*

REAL Question 204

Eleven pages of proposal for a computer laboratory at a school in Kariakoo are mostly sound. Three instructions were copied from an exercise sheet without being tried. A Form Five student who follows them will spend an afternoon hunting for a device nobody placed.

Take each of the three instructions below in turn, say precisely what is wrong with it, and give the version that should have been printed.

1. Drag a 2911 switch onto the workspace and join the four buildings to it.

2. Place three machines on the workspace, PC0, PC1 and PC2, then run a straight-through lead from PC2 to PC3.

3. Address the school network from 198.50.60.0, and once the machines answer one another in the simulator the cabling is proved sound.

Model Answer

A 2911 is an integrated services router and not a switch, so the exercise wants a 2950 or a 2960; a workspace holding PC0, PC1 and PC2 contains no PC3, and the lead intended runs from PC1 to PC2; and 198.50.60.0 is registered public space belonging to somebody else, while a simulator holding no copper cannot prove a cable of any length whatever.

The 2911 is a router: it is an integrated services router, so dragging it in gives a device with a handful of routed interfaces instead of a switch with twenty-four ports. The switch the exercise means is a 2950 or a 2960. Plug four buildings into a router and each becomes a separate network.

PC3 was never created: devices are numbered from zero, so three machines are PC0, PC1 and PC2 and the highest name in use is $3 - 1 = 2$. Cabling PC2 to PC3 asks for something the student has not placed. Either the lead runs PC1 to PC2 or a fourth machine must be added first.

198.50.60.0 belongs to somebody: it is registered public space held by a real organisation. A school local network uses the private ranges reserved for exactly this: 10.0.0.0/8, which by itself offers $2^{24} - 2 = 16{,}777{,}214$ usable addresses, or 172.16.0.0/12, or 192.168.0.0/16. None of the three is applied for, and none is carried across the Internet.

What the wrong block costs: on the day the school reaches the Internet through address translation, every packet sent towards the genuine holder of 198.50.60.0 is delivered instead to a machine in the laboratory. That organisation becomes unreachable from inside the school and from nowhere else, which nobody diagnoses quickly.

A simulator holds no copper: it models configuration and logic, and nothing physical. Attenuation, a crushed lead, a mis-punched pair, a run past the distance limit and a socket with no power all lie outside it. Machines answering on screen prove the addressing and the VLANs; only a tester proves the cabling.

The habit behind all three: read the part number, count what was actually placed, and ask of every claim what it was tested against. Each sentence is confident about something its writer never checked, and copied confidence is the cheapest ingredient in any proposal.

None of the three instructions is foolish, and each was written by somebody who had not carried it out. That is the whole distance between a proposal and a plan.

The Master Key: *Trust a part number, a device list and a test result only after you have seen each of them yourself.*

HOT Question 205

On a laptop in Bariadi, a supplier runs a whole school network. Every machine answers every other, the VLANs behave and the router forwards. The head teacher asks the only question that matters: does this mean it will work when it is built.

Assess what a network simulator can and cannot prove about a design, and reach a judgement on how much weight its results should be given.

Model Answer

Give the simulation full weight on the logical design and none whatever on the installation: it can prove that an address plan, a VLAN scheme, a trunk, a routing decision and a filtering rule do what the designer intended, and it can prove nothing about a metre of cable or a figure of throughput, so it belongs before the purchase order and never on the completion certificate.

What it genuinely settles: the faults that are dear on site and cheap on screen: a gateway configured into the wrong subnet, a mask one bit too generous, a VLAN missing from a trunk, a rule denying more than it was meant to. Each takes minutes to find in a simulator and a working day in a cabinet.

Arguing a design before purchase: a topology on a laptop can be shown to a head teacher, a bursar and a contractor, and altered in front of them at no cost. A design nobody can see is approved on trust; one that can be run is approved on evidence, and the objections arrive while they are still free to answer.

The physical layer is absent: the simulated cable has no length, no attenuation and no crushed section. A 260 m copper run works perfectly on screen and fails on the wall. Connectors, earthing, mains power and heat in a locked cabinet are not in the model, and those are what fail in a school building.

Performance is idealised: the simulated switch has unlimited buffers and the simulated router forwards as fast as the picture is drawn. Real ceilings in packets a second, real queueing when a class of thirty downloads at once, and real interference through concrete are all outside it, so quote no capacity figure from a simulation.

The device is a subset: a simulator implements part of the command set of part of a product family. A configuration accepted on screen may be refused by the equipment actually bought, and a feature the design leans on may be absent from the model quoted. Compare the two lists before the order goes out.

The judgement: require a simulation before anything is purchased and treat what it shows as evidence about the logical design alone. The physical design is proved by the survey and its measured distances, and the installation by a test certificate for every run. Not one of the three stands in for another.

The honest answer at Bariadi is half a yes. The laptop has proved the design, and the building will still have to prove itself.

The Master Key: *Let a simulator settle what a configuration does, and never what a cable does.*

HOT Question 206

One long building, ninety machines, and a quotation for a core switch, two distribution switches and four access switches: the school at Kahama is told that three layers are the professional way. The quotation is competent and the model behind it real.

Judge whether a school housed in a single building should build a two-layer or a three-layer hierarchy, and defend the answer.

Model Answer

Build two layers: collapse the core and distribution jobs onto one layer-3 switch and keep the access switches beneath it, because a single building holds one distribution block and a core exists to join distribution blocks to one another, so the third layer here would be a chassis and a second configuration bought to carry traffic that never leaves one cabinet.

What a core is for: joining distribution blocks at speed and doing nothing else. Give it one block to join and it joins that block to itself, adding a hop to every packet crossing the school and a device to every fault. The layer is not wrong in principle; in this building it is unemployed.

The model is not abandoned: the layers are jobs, not boxes. The single layer-3 switch still does the distribution job, routing between VLANs, filtering and terminating the uplink to the router, and the access switches still do theirs. A collapsed core is the three-layer model with two jobs on one chassis, and the drawing should say so.

What the third layer costs: a chassis, optics at both ends of two uplinks, a second configuration to be kept in step with the first, and one more thing able to fail at three in the morning. Against that stands a benefit appearing only once a second distribution block exists.

The condition that reverses it: add the core when three or more distribution blocks exist, when the campus outgrows what one chassis can aggregate, or when one building must keep working while another cabinet is down. Ninety machines under one roof ask none of those questions.

Design for the upgrade anyway: buy the distribution switch with spare optic slots, and number the VLANs and subnets so that a second building drops into the plan without a machine being renumbered. A saving is only a saving if taking it leaves the next step cheap.

The judgement: two layers, with the reason written in the design file beside the condition that would change it. A model applied where it does not fit is not professionalism, and the school should ask for a fresh quotation without the core and spend the difference on cabling.

The Kahama quotation is not padded out of dishonesty. It is padded because a model was followed instead of being asked whether it applied here.

The Master Key: *A layer with only one thing to join is a layer you are paying to have present.*

HOT Question 207

One distribution switch, one uplink to each building, one router, one line to the provider, no batteries anywhere: the design for a school at Mbinga. The bursar will fund one addition this year and wants to be told which. The mains fails most weeks.

Determine where the one available improvement should be spent on a design carrying no redundancy, and justify the choice against the alternatives.

Model Answer

Spend it on power: an uninterruptible supply holding the distribution switch, the router and the terminating equipment, because that cabinet already carries every user in every building and its most frequent failure by a wide margin is the mains, so the money removes a weekly total outage instead of guarding against a switch failure years away.

Rank the failures by loss: the distribution switch and its power take all 180 users at once; the router and the provider line take access to the outside while leaving the school network standing; one building's uplink takes that building only. Buy against the widest failure, not the nearest one.

Weight them by frequency: a switch of this class runs for years, while the mains at Mbinga fails weekly. Multiply users lost by hours lost by how often it happens, and power dominates every other line by an order of magnitude. Redundancy bought against the rare failure while the frequent one stands unprotected is money on the wrong risk.

Not a second distribution switch: it is the answer the model invites and it consumes the whole budget: a second chassis, a second uplink out of every access switch, and a spanning tree holding each standby link blocked until the live one fails. It defends against the rarest failure in the design.

Not a second fibre first: a standby fibre to one building, blocked until the working one fails, protects one building of four against a cable fault that may come once in a decade. It is worth having, and it is not worth having before the batteries.

What the battery must cover: the distribution switch, the router and the terminating equipment, and not the workstations, which have nothing to do without the servers anyway. Size it in minutes rather than hours, put the server cabinet behind it, and test it twice a term, because an untested battery is a document and not a defence.

The judgement, and what follows: buy the uninterruptible supply this year. Next year buy a cold spare switch, configured, labelled and left in the cupboard, which restores the school in an hour for a fraction of a live second chassis. The second fibre comes third and the second distribution switch fourth.

Redundancy is not a property a design has or lacks. It is a list of failures ordered by what each costs, and the first line of the Mbinga list is a power cut.

The Master Key: *Multiply the users a failure takes by how often it comes, and buy against the largest product on the list.*

Network security: threats, the CIA triad and controls

Security is not a device somebody buys; it is a set of properties an organisation keeps, and a set of layers at which it keeps them. This section settles the vocabulary an examiner expects: the three properties every attack must be measured against, the names of the attacks themselves, the four kinds of malicious program that students habitually call viruses, the firewalls that stand at a boundary, and the three layers, physical, technical and

administrative, at which every control lives. It assumes you already know what a router, a switch and a port number are.

BINDER Question 208

The office of a secondary school in Mikocheni keeps one file of results. Three things can go wrong: a student reads it, a student alters it, or nobody can open it on the morning the results are due. One file, failing in three ways.

State the three properties of the CIA triad, define each, and give one example of a failure of each.

Model Answer

Confidentiality means data is seen only by those authorised to see it, integrity means data is altered only by those authorised to alter it, and availability means the data is reachable when a legitimate user needs it; a mark sheet copied breaks the first, a mark quietly edited breaks the second, and a server that will not start on results morning breaks the third.

Confidentiality: the property that only authorised people see the data. It fails when a payroll file is copied to a memory stick, or when an open wireless network lets a stranger read traffic. Access control lists, least privilege and encryption in transit are its technical controls.

Integrity: the property that data is what its owner last made it. It fails when a mark is changed, or when a payment amount is rewritten in transit. An altered record is more dangerous than a deleted one, because it is believed.

Availability: the property that authorised users reach the data when they need it. It fails when a link is flooded, when ransomware locks the files, when a disk dies, or when the only person holding the password is away. To the user those are one event.

The three pull against one another: tighten confidentiality far enough and availability suffers, since every extra check is another way of being locked out of your own records. The question is which property this particular data cannot afford to lose.

Name the property first: an incident question is answered by naming the property that failed, then the attack, then the control. Begin with the attack and you write about hackers without saying what was lost.

One file, three questions: who saw it, who changed it, and could the right person open it.

The Master Key: *Ask of any incident which of the three properties broke, and the classification writes itself.*

BINDER Question 209

The firewall at a school in Kondoa works perfectly. The server stands on a desk in a room that is never locked, where the cleaner keeps the mop and the window has no grille. The loss that will happen walks in through the door.

Using Figure 3.37, describe the three layers at which a security control can operate, give two examples of each, and explain how the layers relate.

Model Answer

Every control belongs to one of three layers: physical controls keep people away from equipment, technical controls are enforced by the equipment itself, and administrative controls are the written rules that decide what the other two are set to; a door lock and a lockable cabinet are physical, a firewall rule and an access control list technical, an acceptable use policy and a leavers procedure administrative.

Physical controls: a locked server room and a lockable cabinet, with a visitor book and cable run in conduit as further examples. The reasoning is blunt: a machine an intruder can touch is one he can restart or carry away.

Technical controls: a firewall rule set and an access control list, with user accounts, logging and encryption as further examples. They are enforced without asking anybody, which is their strength; they also do exactly what somebody configured, which is their weakness.

Administrative controls: an acceptable use policy and a documented leavers procedure, with training, a change log and audit as further examples. They cost paper and habit rather than money, and they decide what the other two layers enforce.

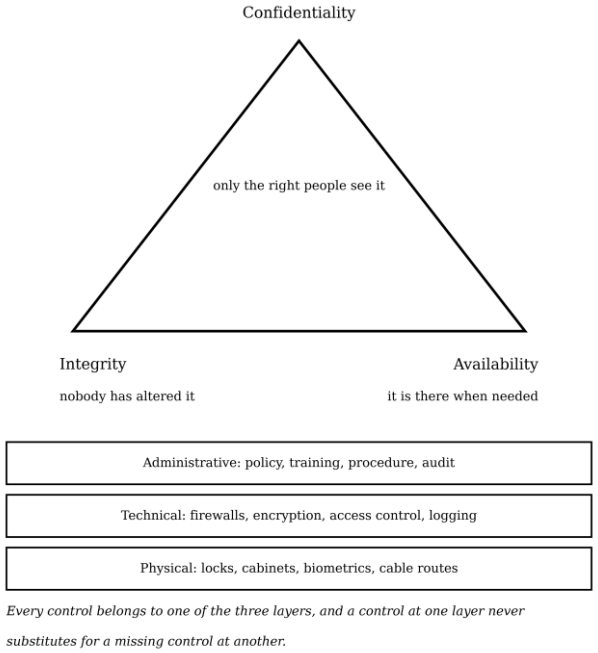


Figure 3.37: *The CIA triad and the three layers at which any control operates.*

A second axis, by purpose: cutting across the layers, a control is preventive, detective or corrective. A lock prevents, a log detects, a tested restore corrects. A design with no detective control never discovers that its preventive ones stopped working.

No layer substitutes for another: a firewall cannot lock a door, a policy cannot stop a packet, and a lock cannot decide who holds the account. Read the figure downwards: policy sets the rules, equipment enforces them, the lock protects the equipment.

The three layers are a checklist for a design review. Place each control the school owns into the figure, and the empty rows argue for the next purchase.

The Master Key: *Name the layer before you name the control, and the gap in a design shows itself.*

BINDER Question 210

The incident book in a laboratory in Kyela records the same diagnosis three times in one term: a virus. Yet the thing that filled the file server overnight attached itself to no file, and the free printer driver installed in March copied itself nowhere.

Distinguish a virus, a worm, a Trojan and ransomware, stating in each case how it spreads.

Model Answer

A virus attaches itself to a host file and replicates only when that host is run, a worm is a complete program that propagates across a network with no host and no user action, a Trojan does not replicate but relies on a user running it because it looks useful, and ransomware encrypts the files and demands payment.

A virus needs a host: it writes itself into a document, a program or a boot sector and runs when that carrier runs, copying itself into further carriers. It travels wherever the infected file travels.

A worm needs only a network: it is self-contained and self-propagating, reaching the next machine through a service listening on a port with nobody opening anything. Growth is multiplicative: ten doublings give 2^{10} = 1,024 machines, which is why a worm saturates a link before anyone reports a fault.

A Trojan does not replicate: it is a program that lies about what it is, and the user installs it willingly. It reaches a second machine only if a second user is deceived, and what it leaves behind is usually a back door.

Ransomware encrypts and extorts: it makes the files unreadable and demands payment. The property attacked is availability, since the records are still present and read by nobody. It is a payload rather than a way of travelling, and a worm or a Trojan delivers it.

Spyware observes: it records keystrokes or traffic and reports them elsewhere, and is written not to be noticed. It attacks confidentiality and succeeds by leaving everything working.

The habit that earns the mark: answer the propagation question before the damage question: host, network, or neither. Calling a Trojan a kind of virus concedes the only distinction being tested.

These are not four degrees of severity. They are four answers to one question: what does this program need to reach the next machine?

The Master Key: *Classify malware by how it travels, never by how much it destroys.*

BINDER Question 211

Two devices share a rack at a district office in Kahama, and the inventory calls both of them firewalls. One reads the addresses and port numbers on the front of each packet, nothing more. The other knows which request an arriving reply belongs to.

Describe four types of firewall, stating in each case what part of the traffic it inspects and where it is placed.

Model Answer

A packet filter inspects only the header of each packet in isolation, meaning address, protocol and port; a stateful inspection firewall also holds a table of open connections, so it tells a genuine reply from an unsolicited packet; an application gateway, or proxy, terminates the connection and inspects the application protocol itself; and a next-generation firewall adds recognition of the application and of the user.

Packet filter: permit and deny rules applied to header fields, one packet at a time with no memory of what came before. It is fast and cheap, and usually configured on the boundary router. Being stateless, it cannot tell whether an arriving packet answers anything the inside asked for.

Stateful inspection: the same header rules plus a table of every connection the inside has opened. Inbound traffic is permitted only where it matches an entry, so unsolicited packets are dropped without a rule being written. This is the ordinary perimeter device.

Application gateway or proxy: it accepts the client connection, opens its own to the server and reads the application protocol, so it can permit one request and refuse another within one service. It is slower, is written for each protocol, and stands in front of the service it guards.

Next-generation firewall: deep inspection of the payload, identification of the application whatever port it uses, identification of the user rather than the address, and intrusion prevention. It sits at the perimeter in place of the others.

Where an older taxonomy fails: dividing firewalls into packet filter, application gateway and hybrid omits stateful inspection and next-generation, the two types actually deployed. Hybrid describes how a product is packaged, not what it inspects, which is the only thing the classification is for.

Each type asks a different question of one packet: where is it from, does it answer something we began, what does it ask for, and who is asking.

The Master Key: *A firewall is defined by what it reads, so answer with what it reads.*

REAL Question 212

For one working day the records system at a district hospital in Tarime will not open. Clerks write by hand, the pharmacy queue reaches the door, two clinics run from memory. By morning it is back, every record present, nothing copied and nothing altered.

Determine which property of the CIA triad failed, classify the incident, and state what must be established before the report can be closed.

The patient records system was unreachable for one full working day.

When it returned, no record was missing and no record had been altered.

The logs show no copy of the database leaving the hospital.

The cause has not yet been established.

The hospital link to its provider is rated at 20 Mbit/s.

Model Answer

Availability failed, and only availability: nothing was disclosed and nothing was altered, so confidentiality and integrity held, and the incident is classified as a loss of availability, which is a security incident and not merely an operational one; before the report can close, the cause must be established, because a flooded link, a failed disk and ransomware present to the user as the same unreachable system.

Why availability is the property that failed: availability means reachability by an authorised user at the moment of need. A clinician who cannot open a record has been denied it as completely as if it were deleted. One lost day in a ninety-day quarter leaves 89 of 90 days, or 98.9 per cent availability, which sounds respectable and is unacceptable for a clinical system.

The other two are checked, not assumed: the claim that nothing was read and nothing changed rests entirely on the logs. Where there is no logging there is no evidence, and an honest report then records the two properties as unverified rather than intact.

Classify by cause: a denial of service, ransomware or an insider deleting a table are attacks; a full disk, a power failure or an undocumented configuration change are accidents. Both end in an availability failure, but only the first is reported as an attack and only the second is cured by housekeeping.

Why distributed changes the answer: one machine sending 1 Mbit/s of rubbish is a nuisance. Two hundred sending the same each deliver $200 \times 1 = 200$ Mbit/s, which is $200 \div 20 = 10$ times the capacity of the hospital link. No rule at the hospital firewall helps, because the link is full before the firewall sees anything; filtering must happen upstream at the provider.

The loss that arrives later: a day of paper records is a day of handwriting to be typed in afterwards, and that is where transcription errors enter the database. An availability failure becomes an integrity failure a week later, so recovery is part of the incident.

Threat	How it works	C, I or A	First defence
Unauthorised access	Uses another person's credentials, or a service left open, to reach data or a device	C and I	One account per person, least privilege, access control lists
Denial of service and DDoS	Floods a link or a service; distributed means many machines at once	A	Upstream filtering by the provider, rate limits, spare capacity
Man in the middle	Relays traffic between two parties who believe they are talking directly	C and I	Encryption in transit and authentication of the far end
SQL and code injection	Input crafted so the application runs it as a command instead of storing it as data	C, I and A	Validate all input, parameterised queries, a restricted database account
Privilege escalation	Gains rights not granted: horizontally to a peer's data, or vertically to administrator	C and I	Least privilege, separate administrator accounts, log every use
Insider threat	A trusted person misuses the access the job requires	C, I and A	Separation of duties, logging, a leavers procedure
DNS and routing-table poisoning	False entries fed to a resolver or a router divert traffic to the attacker	C, I and A	Authenticate routing updates, fix which resolvers may be used
Packet sniffing	Captures traffic on a shared or wireless medium, entirely passively	C	Encryption in transit, switched links, no open wireless
Social engineering and phishing	Persuades a person to give up credentials or run a program	C and I	Training, a route for reporting, rules no request may override

Table 3.31: *The threat catalogue.*

What the closed report must contain: the property that failed, the hours lost, the cause with its evidence, the grounds for saying the other two held, and the control that would have shortened the outage. A tested restore is that control, corrective in purpose, technical in layer.

Nothing was stolen from Tarime, and the hospital still suffered a security incident on a day when a clinician could not read a drug allergy.

The Master Key: *Count the hours nobody could work, because those hours are a security loss and nothing else records them.*

REAL Question 213

The wireless network in the staff room at Chake Chake is open, because a password was one more thing to explain to every visiting teacher. A laptop in a parked car advertises the same name and a stronger signal. Every screen still looks normal.

Explain how a man-in-the-middle attack works on an unencrypted wireless network, and state the two controls that would prevent it.

Model Answer

In a man-in-the-middle attack the attacker places himself so that two parties each send to him while believing they are sending to one another, letting him read everything and alter it if he chooses; on an open wireless network he takes that position by advertising the same network name with a stronger signal, because nothing obliges an access point to prove which network it is; the two controls that prevent it are encryption in transit and authentication of the far end.

How he reaches the middle: a client associates with the strongest signal carrying a familiar name, and an open network asks no proof of identity in either direction. The attacker forwards everything onward to the real access point, so pages load and nothing appears wrong. Invisibility is the whole design.

What the position gives him: all three properties at once. He reads what passes, which breaks confidentiality; he rewrites a bank account number or a mark before passing it on, which breaks integrity; and he delays or drops traffic, which breaks availability. One place in the path, three failures.

Why an open network invites it: with nothing encrypted, every frame is readable by any machine within radio range even without the middle position, so plain packet sniffing already succeeds and leaves no trace. The interposed laptop is the active version of a weakness the school has already accepted.

First control, encrypt in transit: a technical control. An interposed device then relays bytes it cannot read, and cannot rewrite them without the change being detected at the far end. Note what this leaves: an attacker still in the path can delay traffic, so availability stays exposed.

Second control, authenticate the far end: the client must refuse an access point that cannot prove which network it belongs to, and the network must equally establish which client is joining. Without this, encryption merely protects the conversation the user is having with the attacker.

The backstop at the other layers: a written rule that school business is never done over an open network, taught rather than filed, and a periodic walk round with a scanner looking for access points using the school's name. The first is preventive and administrative, the second detective and technical.

Nothing in the staff room looked wrong all morning, and that is precisely the point: this attack succeeds by working.

The Master Key: *Encryption hides the conversation; authentication settles who you are having it with.*

REAL Question 214

Each year the board of a secondary school in Kariakoo receives a one-page security report from the member of staff who looks after the computers. This year's is brisk, and the board approves it without discussion. Three of its sentences are wrong.

Identify the error in each statement, give the correct version, and explain why the correction matters.

1. The viruses to guard against are Melissa, the Trojan horse and BadTrans.

2. The firewall at the boundary protects the school against an attack launched by a member of staff at a desk inside the building.

3. Availability is an operational matter for the technician; security proper is about confidentiality and integrity.

Model Answer

A Trojan is not a virus at all, since it does not replicate, and Melissa and BadTrans were mass-mailing worms rather than viruses; a boundary firewall inspects only traffic that crosses the boundary it stands on, so it never sees a packet sent from one internal machine to another and can do nothing about an insider; and availability is one of the three properties security exists to protect, so an outage is a security incident whatever caused it.

Statement one, the list is mixed: a Trojan does not replicate, so it cannot be a virus, and Melissa and BadTrans mailed copies of themselves to addresses found on each infected machine, which makes them worms. The correction has teeth: a Trojan is answered by controlling what users may install and by least privilege, a worm by closing the service it travels through and segmenting the network.

Statement two, the boundary is the limit: a firewall filters what passes the boundary on which it is placed. A member of staff at an internal desk is already inside, and his traffic to the file server never reaches the firewall at all. The insider threat is answered by least privilege and access control lists technically, separation of duties and a leavers procedure administratively, and a locked cabinet physically.

Statement three, availability is a security property: ransomware and a distributed denial of service attack availability deliberately, a failed disk destroys it accidentally, and the clerk at the counter cannot tell the two apart. Calling it operational is how a school ends up with backups nobody has restored and an outage nobody owns.

What the three have in common: each shrinks the subject to the part its author already understood: malware to one word, security to one device, and the triad to two of its properties. Put two questions to any security sentence, which property does this protect and at which layer does it act, and all three collapse.

The report is not careless. It is confident about a smaller subject than the one it claims to report on, which is why nobody on the board noticed.

The Master Key: *Every security claim owes two answers: which property, and which layer.*

HOT Question 215

Three secondary schools around Mbinga buy the same firewall in the same year. In all three, the account used to log into the office computer is called admin, has no password, and is shared by every teacher who needs the printer.

Assess which of the three layers of control a Tanzanian secondary school most often neglects, and defend your judgement.

Model Answer

The administrative layer is the one most often neglected, and three things carry the judgement: it is the only layer that costs no money, so its absence cannot be blamed on the budget; it decides what the other two are set to, so its absence quietly disables them; and its failures are invisible until an incident, because no indicator turns red when a leaver's account is left enabled for two years.

What a school usually has: something at the technical layer, because technical controls can be quoted for and installed in a morning, and often something physical, because a cabinet arrives with a lock. Both are objects, which is exactly why they get bought.

What is usually missing: an acceptable use policy, an account for each person instead of one shared account, a leavers checklist, a change log, least privilege on the file server, a tested restore, and a named person answerable for security. Every item is paper and habit, and none appears on a quotation.

Least privilege is an administrative decision: somebody must decide what each role may do before the technical layer can enforce it as an access control list on a folder or an interface. Where that decision is never taken, the list is written to whatever was convenient, which means everybody reads everything.

The absence disables the layers below: an unowned firewall accumulates permit rules nobody can justify and nobody dares remove. A cabinet whose key hangs on a nail beside it is not locked. Both lower layers decay to the standard of the administrative layer above them, so the neglect compounds.

The strongest objection: in many schools the physical layer looks worse, with the server on an open shelf in a store room. The objection is fair on the evidence and loses on cause, because nobody was made responsible for where the server lives. Apply the honest test, which layer if repaired first improves the other two, and only the administrative answer passes.

The judgement and the first three actions: the neglect sits at the administrative layer. Retire the shared admin account and give each member of staff their own, write a one-page leavers checklist and use it when somebody next leaves, and restore one backup a month, recording the result. All three are free, all three fit inside a term, and each makes the firewall worth more than it was.

The firewall was never the wrong purchase. It was the second purchase made first, while the one that should have come before it cost nothing at all.

The Master Key: *Buy the layer that costs nothing before the layer that costs money, because the free one sets what the expensive one is worth.*

HOT Question 216

Parents will type an address into a browser and read their child's results. At a school in Temeke the server that answers can stand beside the file server, with the provider that hosts the school mail, or on a third interface of the firewall.

Judge whether the school should place its web server inside the network, outside it with a provider, or in a demilitarised zone, and justify the recommendation.

Model Answer

Put it in a demilitarised zone if the school has a firewall with a spare interface and somebody competent to maintain the rules on it; if it has neither, host it with the provider, which is the better of the two remaining choices; and never put it inside the office network, because a server the public must reach will eventually be reached by somebody unwelcome, who then arrives already past the boundary.

What a demilitarised zone is: a separate segment on its own firewall interface holding the services outsiders must reach. Three rules define it: the outside may reach the zone on the one port the service needs, the inside may reach the zone as required, and the zone may reach the inside on nothing at all.

Why inside is the worst of the three: a compromised web server on the office segment shares it with the file server, the accounts machine and every staff computer. The attacker's first foothold is also his last obstacle, and privilege escalation proceeds sideways while the firewall guards a boundary he is already behind.

What hosting outside buys and costs: the provider owns the machine and its upkeep, and a compromise touches nothing of the school's. The costs are real: dependence on the provider, a working link before anything can be published, and data sitting where the school cannot inspect it. For marks published to parents, that trade is acceptable.

The condition attached to the zone: a demilitarised zone is a rule set, not a socket. A third interface carrying a permit-any rule is the office network under a grander name. It needs rules that are reviewed and logs that are read, and a school without somebody to do both is safer buying hosting.

Size the decision by what the server holds: a portal publishing a copy of the marks written out each night carries little risk. One querying the live examinations database needs one narrow rule from the zone to one server on one port. Say that rule aloud before agreeing to it, because it is the hole in the middle rule.

The judgement: the demilitarised zone where the competence exists, provider hosting where it does not, and never the office network. Write the three rules into the handover document, so that the next technician must argue in writing before adding a fourth.

The question was never whether the results server will be attacked. It is which network the attacker is standing on at the moment it happens.

The Master Key: *Place a server by asking where its attacker would be left standing afterwards.*

Cryptography, wireless security and modern defence

A network is defended in the end by arithmetic, not by good intentions. This section settles what a key is, why there are two kinds of key and why a hash is neither kind; it puts the four wireless standards in order and names the two of them that are broken; and it finishes with the housekeeping, training and patching, that prevents more

incidents than any appliance ever bought. It assumes you can already name the common threats and the layers at which a control operates.

BINDER Question 217

Locking a file and delivering the key are two different problems. A results spreadsheet leaving a staff room at Mikocheni must be readable by one colleague and nobody else, so either a secret reaches that colleague first, or no secret travels at all.

Distinguish symmetric encryption from asymmetric encryption, and give one advantage and one disadvantage of each.

Model Answer

Symmetric encryption uses one shared key for both locking and unlocking, so it is fast, but the key must reach the other party in secret beforehand; asymmetric encryption uses a linked pair, a published public key and a withheld private key, so no secret travels first, but it is far slower.

The symmetric arrangement: one key performs both operations. The Advanced Encryption Standard, AES, is the cipher in general use, with keys of 128, 192 or 256 bits, and it is fast enough to encrypt a whole disk on ordinary hardware.

Its advantage and its cost: the advantage is speed. The disadvantage is distribution: the key must be agreed over a channel that is already private, which is the problem encryption was meant to solve. Every pair of parties needs its own key, so thirty machines conversing in pairs need 435 keys between them.

$$\text{Keys for n parties} = n(n-1)/2 = 30 \times 29/2 = 435$$

The asymmetric arrangement: two keys are generated together as a pair, commonly by RSA at 2,048 bits. What one half locks, only the other opens. The public half is published to anybody who asks and the private half never leaves the machine that made it.

Its advantage and its cost: the advantage is that two parties who have never met can correspond privately at once. The disadvantage is speed: RSA is orders of magnitude slower than AES on the same data, which rules it out for bulk traffic.

Why both are used together: the slow pair carries only a short session key and the fast cipher carries the traffic. Every secure web session works this way: the asymmetric step agrees one AES key, and AES encrypts the conversation.

Judge each family by the question it answers. Asymmetric encryption gets a key to the far end; symmetric encryption does the work once the key is there.

The Master Key: *Ask how the key gets there before you ask how strong it is; distribution, not strength, separates the two families.*

BINDER Question 218

The words public and private carry the whole scheme, and both keys belong to the same person. A supplier in Temeke prints one of them on its website, where any competitor may copy it, and gives away nothing whatever by doing so.

Using Figure 3.38, state which key encrypts and which key decrypts in an asymmetric scheme, and say to whom each of them belongs.

Model Answer

The sender encrypts with the recipient's public key and the recipient decrypts with the recipient's private key, so both halves belong to the recipient, and only the private half, which never leaves the recipient's machine, will open what the public half has locked.

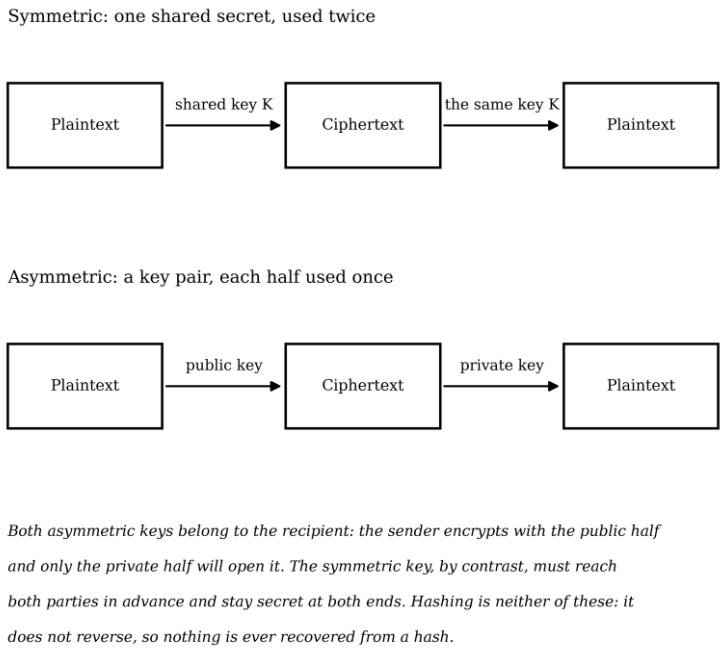


Figure 3.38: *Symmetric and asymmetric encryption. Both asymmetric keys belong to the recipient; only the private half opens what the public half has locked.*

Reading the two drawings: in the upper drawing one label, key K, appears on both arrows, and that key must already be held at both ends. In the lower drawing the arrows differ: the public key going in, the private key coming out. Two labels mean two keys, and they are not interchangeable.

Whose keys these are: both belong to the recipient. The sender supplies no key in this direction and needs nothing secret of their own, which is exactly why a stranger can send you a private message with no prior arrangement.

Why publishing is safe: the public half is meant to be given away. Recovering the private half from it means solving a problem that is infeasible at the sizes in use, so a competitor holding the public key gains one ability only: to write messages that only the owner can read.

Keeping the private half: it never travels. It is not emailed, not copied to a shared drive, not kept in the same backup as the data it protects. Its whole value lies in one machine holding it, and a copied private key opens every message ever sent to that public key.

The direction reversed: encrypting with the private half and opening with the public half conceals nothing, since everybody holds the public half. It proves instead that the owner of the pair performed the operation, which is the digital signature and not confidentiality.

Confidentiality always runs towards the owner of the key pair. Name whose key sits on each arrow and the rest of public-key cryptography follows without difficulty.

The Master Key: *Encrypt to somebody with what they published; only what they kept back will open it again.*

BINDER Question 219

A notice taped above the help desk at a district office in Pangani says staff can reset your password but cannot say what it was. An officer who has lost hers reads that as laziness, and the clerk cannot explain why it is true.

Explain what a hash function is, and explain why hashing is not a form of encryption.

Model Answer

A hash function converts an input of any length into a fixed-length digest by a one-way calculation, so SHA-256 reduces anything at all to 256 bits, and because there is no key and no inverse a hash cannot be undone: encryption exists to be reversed by whoever holds the key, and hashing is defined by nothing reversing it.

Fixed output, any input: SHA-256 returns 256 bits, that is 32 bytes, whether it is given one character or a file of one megabyte, which is 8,388,608 bits. The same input always gives the same digest, and altering one character changes about half the output bits.

Why no inverse exists: the output is far shorter than most inputs, so many different inputs must share one digest and nothing in a digest records which produced it. No key would help, because there is no key at all: a hash is not encryption with the key thrown away.

Checking integrity: publish the digest of a file beside the file. Anyone who downloads it hashes their copy and compares, and one altered byte gives a completely different value. That is how a tampered installer is caught, and why hashing is the integrity tool and not a secrecy tool.

Storing passwords: the system stores the digest and never the password. At each login it hashes what was typed and compares digests, so a stolen database yields digests only. A salt, a random value stored beside each digest and mixed into the calculation, defeats precomputed tables.

Property	**Symmetric**	**Asymmetric**	**Hashing**
Keys used	One shared key	A public and a private key	None at all
Reversible	Yes, with the key	Yes, with the private key	No, never
Speed	Fast	Slow	Fast
Usual algorithm	AES	RSA	SHA-256
What it is for	Bulk confidentiality	Key exchange and signatures	Integrity and stored passwords

Table 3.32: *Symmetric encryption, asymmetric encryption and hashing.*

The help desk was telling the truth. A system that verifies a password without holding it is behaving correctly; one that can send you your old password has stored it in a form a thief reads as easily as you do.

The Master Key: *A hash is a fingerprint, not a locked box: it identifies the thing and never gives it back.*

BINDER Question 220

The security menu on an access point at a guest house in Stone Town offers four settings, listed as four flavours of one thing. Two of them can be broken from recorded traffic in an afternoon, and the menu says nothing about which two.

Using Table 3.33, place the four wireless security standards in order of strength and state which of them should be configured today.

Model Answer

In ascending order of strength they are WEP, then WPA with TKIP, then WPA2 with AES, then WPA3: the first two are broken and must not be used, WPA2 with AES is the minimum acceptable setting today, and WPA3 is the current standard and should be chosen wherever the equipment offers it.

WEP, broken: Wired Equivalent Privacy arrived with the original wireless standard in 1997. It uses the RC4 cipher with a 24-bit initialisation vector sent in clear beside every frame, and 24 bits allow only 16,777,216 values, so a busy network repeats one within hours and the repeats give the key away.

WPA with TKIP, broken: WPA was an interim repair issued in 2003 that had to run on hardware built for WEP, so it kept RC4 and wrapped it in TKIP, which adds a fresh key per packet and an integrity check. It inherits too much of RC4, it is deprecated, and it holds a network down to older rates.

WPA2 with AES, the floor: WPA2 followed in 2004 and replaced RC4 altogether with AES in CCMP mode. Properly configured and with clients kept updated, it remains sound. Select WPA2 with AES specifically, because a mixed WPA and WPA2 setting readmits TKIP and quietly restores the weakness you were removing.

WPA3, current: WPA3 was published in 2018. It keeps AES and replaces the shared-key handshake with Simultaneous Authentication of Equals, so a recorded handshake can no longer be attacked offline by trying passphrases, and each session gets its own keys. Both personal and enterprise variants exist.

Standard	Encryption	Status	Advice
WEP	RC4, 24-bit IV	Broken; key recoverable from recorded traffic	Never use; replace equipment that offers nothing better
WPA	TKIP, still RC4	Broken and deprecated	Do not use; it was an interim repair for WEP-era hardware
WPA2	AES in CCMP mode	Sound when patched	The minimum acceptable setting today
WPA3	AES with the SAE handshake	Current	Choose it wherever the equipment supports it

Table 3.33: *The wireless security standards.*

Set WPA3 where every client can join it, WPA2 with AES where some cannot, and treat an access point that offers only WEP as equipment to be replaced rather than configured.

The Master Key: *Two of the four names are museum exhibits; configure only from the right-hand end of the table.*

REAL Question 221

Twenty-two characters, upper and lower case, two digits and a full stop: the wireless passphrase at a secondary school in Bariadi is changed every term. The encryption setting has never been touched. It stands at WEP, for four old laptops in the library.

Explain why that network is not secure in spite of the passphrase, and state what the school should change.

Model Answer

The passphrase is beside the point, because WEP's weakness lies in the protocol and not in the key: an attacker who records enough frames calculates the key instead of guessing it, so the school must move every access point to WPA2 with AES at the least, or WPA3 where the firmware allows, and deal with the old laptops separately.

How WEP fails: each frame is encrypted with RC4 using the shared key joined to a 24-bit initialisation vector that travels in clear. Only 16,777,216 vectors exist, so on a busy network one recurs within hours, and a statistical attack on the recurrences recovers the key itself in minutes.

Why the passphrase changes nothing: a WEP key is 40 or 104 bits, quoted in the menu as 64 and 128 because the 24-bit vector is counted in. The attack extracts those key bytes from captured traffic; it never tries a password. Twenty-two well-chosen characters therefore buy no time at all, and the termly change buys none either.

What is at risk meanwhile: once the key is out, everything crossing that air is readable to a listener in the car park, and any device that can associate is inside the school network rather than outside it. Traffic carried over HTTPS survives, but the network itself separates nobody from anything.

The change that matters: set WPA2 with AES on every access point, and WPA3 where the firmware offers it. Avoid the mixed WPA and WPA2 option, which readmits TKIP. Keep a long passphrase, because under WPA2 a recorded handshake can be guessed at offline, and there passphrase length genuinely does decide the outcome.

The four old laptops: do not hold an entire school at a broken standard for four machines. Fit each with a new wireless adapter, attach it by cable, or leave it on a separate access point reaching the internet and nothing else. Any of the three costs less than one incident.

The rest of the policy: change the administrator password on each access point from the factory default, disable WPS, give visitors a guest network kept apart from staff systems, and update firmware. Hiding the network name and filtering by MAC address are not security measures; both are defeated by listening for a few minutes.

The card in the safe protects the one thing WEP was never attacked through. Discipline about passphrases is worth keeping, but it is the encryption standard, and not the passphrase, that decides whether the traffic can be read.

The Master Key: *A passphrase guards an account; a protocol guards a network, and no passphrase repairs a protocol that is already broken.*

REAL Question 222

Three sentences from a two-page security report, written by an intern leaving the ICT office of a district council in Kondoa, are quoted below. Each is stated with total confidence and each would pass unchallenged in most staff rooms. All three are wrong.

Identify what is wrong with each statement, give the correct position, and say why each is a tempting thing to write.

1. A strong password makes a wireless network secure, whatever encryption standard the access point is set to.

2. SSL and TLS are two names for the same protocol, and both of them are current.

3. Hashing a password encrypts it, so a user who forgets a password can have the original recovered from the database.

Model Answer

A passphrase feeds the key but does not choose the cipher, and no passphrase rescues WEP; SSL and TLS are different protocols and SSL is dead, SSL 3.0 having been deprecated by RFC 7568 in 2015, so TLS 1.2 or TLS 1.3 is what should be running; and hashing is not encryption at all, because a hash has no inverse and nothing whatever can be recovered from it.

The first statement: the standard chooses the cipher and the handshake; the passphrase is only one input to the key. Under WEP the key falls to a statistical attack on recorded frames whatever the passphrase, under WPA2 the passphrase matters because a handshake can be guessed at offline, and under WPA3 even that is prevented. The temptation is that password strength is the one security instruction everybody has heard, and the access point asks for the password on the same screen as the standard.

The second statement: SSL was the original protocol; TLS succeeded it and the two differ on the wire. RFC 7568 deprecated SSL 3.0 in 2015, TLS 1.0 and TLS 1.1 have since been deprecated in their turn, and TLS 1.2 or TLS 1.3 is the standard. The temptation is that the older name survives everywhere it is no longer accurate, in the phrase SSL certificate, in the names of software libraries and in the price list of every supplier.

The third statement: hashing runs one way, with no key and no inverse, and SHA-256 returns 256 bits however long the input. Logging in works by hashing what was typed and comparing digests, so the system never holds the password. A forgotten password is reset, never recovered, and any system that can send you the old one has stored it reversibly, which is a fault. The temptation is that hashing and encryption both turn readable text into unreadable text, and both are taught in one lesson.

What the three share: each stops one step short of naming the mechanism. Put three questions to any security claim: which algorithm, which version of it, and does the operation reverse. None of these sentences can be written by anybody who has answered all three first.

Send the report back marked rather than rejected, because the intern has examined the right three areas. Each correction is one sentence long, and each turns a comfortable belief into something that can be configured and checked.

The Master Key: *In cryptography a family name settles nothing: only the version, and the direction the operation travels, can be marked right.*

HOT Question 223

A regional education office in Magu circulates timetables by electronic mail, and last term one reached the schools with a date altered. Nobody could establish whether the change came from the office, a fault in transmission, or somebody who had forged the head's address.

Assess how a digital signature proves both who sent a message and that the message was not altered, and reach a judgement on which of those two properties matters more.

Model Answer

A digital signature is the hash of the message encrypted with the sender's private key: the recipient decrypts it with the sender's public key, re-hashes the message, and the two digests match only if the

message is unaltered and the signer held the private key, so both properties arrive at once, but authentication carries the weight, because integrity alone is available from a bare hash that anybody can compute.

Making the signature: hash the timetable with SHA-256 to a 256-bit digest, encrypt that digest with the sender's private key, and send it alongside. The message itself is not concealed, since signing and encrypting are separate operations that use the pair in opposite directions.

Checking the signature: the receiving school decrypts the signature with the office's public key, which recovers the digest the office computed, and hashes the timetable it actually received. If the two digests are identical, one comparison has settled two separate questions.

Why it proves integrity: changing one date changes the recomputed digest completely, so it can no longer match. An attacker who alters the timetable must produce a new signature opening to the new digest, and that needs the private key, the one thing never sent.

Why it proves origin: only the holder of the private key can produce a value that the matching public key opens into the correct digest. That also gives non-repudiation: the office cannot later deny a timetable it signed, provided the private key was never shared with anybody.

Where the proof really rests: on the certificate. A public key by itself is only a number; a certificate is that number bound to a named identity and signed by a certificate authority the recipient already trusts. Without one, an attacker generates a pair, signs a forged timetable, supplies the matching public key, and every check passes.

The judgement: authentication matters more. Integrity is cheap on its own: publish a SHA-256 digest beside the file and any reader can verify it, so what the private key adds is that nobody else could have produced that digest. For the office at Magu, the property worth paying for is the binding of the timetable to a certified identity.

Notice where the judgement turns. Inside one organisation whose staff already hold each other's keys, identity is settled in advance and integrity is all that remains; on an open network no such assumption exists, and the certificate becomes the whole argument.

The Master Key: *Verify the key before you verify the signature, because a key nobody vouches for turns a proof of identity into a proof of nothing.*

HOT Question 224

One grant, two proposals, and a school board that must choose tonight. The first replaces the firewall at a secondary school in Ifakara with a next-generation appliance. The second buys two mornings of training for all staff, with a short drill each term.

Judge which of the two would prevent more incidents at that school, and defend the judgement.

Model Answer

Fund the training. The attacks that actually reach a Tanzanian secondary school arrive through people who are already authorised, and no firewall can refuse traffic that a legitimate user invited; the firewall the school already owns, kept configured and updated, covers what a firewall is able to cover.

What social engineering is: manipulating a person into performing an action or disclosing information, using authority, urgency, familiarity or fear in place of any technical flaw. Phishing is its commonest form: a message that imitates a trusted sender and asks for a credential, a payment or the opening of an attachment.

Why the firewall cannot help: the traffic is outbound, invited and encrypted. A teacher who types the school password into a convincing imitation of the results portal has made a request that is well formed and indistinguishable from legitimate work. There is nothing there for a filter to object to, and the attacker then arrives holding a valid credential.

Its forms at a school: a message claiming the results portal needs the password reconfirmed before the deadline; a caller presenting himself as the supplier's engineer and asking for the router password; a memory stick left in the staff room; an instruction from the head's address to change the bank details on a supplier payment.

What the training must contain: not general awareness but a rule and a route. No password is given to anybody by any channel, because no legitimate service asks for one. Any request for money or credentials is verified on a second channel using a number already known. Suspected messages go to one named person, and reporting is never punished.

The case for the firewall: it refuses unsolicited connections from outside, closes services nobody uses, keeps the student network away from the office network and records what left. A school with no firewall should buy one first. This school has one already and is offered a better one, which is a far smaller improvement.

The judgement, with one condition: buy the training, and spend the first hour of technician time confirming that the existing firewall is switched on, denies by default and carries current firmware. An appliance in place but misconfigured is the one case that would reverse this decision.

Technical controls raise the cost of attacks that need no permission. Training raises the cost of the attacks that ask for permission politely, and in a school it is overwhelmingly the polite ones that succeed.

The Master Key: *No appliance can refuse a request your own staff have already agreed to; train the people the attacker will telephone.*

HOT Question 225

No server at a school in Sumbawanga may be restarted between the first day of term and the last: a reboot during a results upload cost a week. Four security updates have appeared since term began. One concerns a flaw reachable from the internet.

Determine what compensating controls the school should adopt while patching is deferred, and justify each of them.

Model Answer

Defer the ordinary updates but hold one exception, that a remotely reachable flaw is patched out of hours within days rather than at the holiday, and carry the rest of the term on four compensating controls: an inventory checked by a vulnerability scan, exposure cut to the minimum, segmentation with least privilege, and tested offline backups whose logs somebody actually reads.

What deferring patches costs: patch management means knowing what software is installed, tracking the updates issued for it, testing them and applying them on a schedule. Deferring is a decision to run with known flaws, and known is the operative word: a published update tells an attacker exactly where the weakness lies.

Know what is exposed: list every server, service and version, then run a vulnerability scan, an automated check that compares what is running against a database of published flaws and reports which are present. Scanning finds and reports; patching fixes. Without the scan the school is guessing which of the four updates it can afford to postpone.

Cut the exposure: most flaws are exploitable only if the vulnerable service can be reached. Take the server off the public internet, close every unused port and disable services nobody needs. Where staff must reach it from home, put it behind a VPN, an encrypted tunnel across a public network built with IPSec or with TLS, so the service answers only authenticated tunnel traffic.

Segment and restrict: put the server on its own segment, unreachable from the student wireless network. Give administrators named individual accounts rather than one shared login, and run the application without full system rights, so a compromised service does not hand over the whole machine.

Assume something gets through: back up daily to media the server itself cannot overwrite, and restore one backup onto spare hardware each term so the restore is known to work. Switch logging on and give one person fifteen minutes a week to read what the server refused; an unread log has prevented nothing.

The judgement, and where it bends: the no-restart rule is defensible for ordinary updates and indefensible for a remotely exploitable flaw in an exposed service. Negotiate a written window of seven hours, from 22:00 to 05:00, with a snapshot taken first and a rollback rehearsed, and patch everything remaining in the first week of each holiday.

Compensating controls buy time; they never buy immunity. Every one of them costs less to run than a single term spent restoring a compromised server from backups that nobody has ever tested.

The Master Key: *When you cannot close the hole, move it out of reach and write down the date on which you will close it.*

PSTN, mobile networks and telephony

Long before any of it carried data, the country was wired for one purpose: to take a voice from one handset to another and hold a path open between them until somebody rang off. That network still stands, and the mobile

network that overtook it borrowed its shape and most of its vocabulary. This section fixes the fixed line and its local loop, the hierarchy of exchanges above it, the cellular equipment that replaced copper with a radio path, and the dates and rates that make the march from 1G to 5G mean something.

BINDER Question 226

The handset in a district council office in Nzega has no screen and no battery, yet works when the mains fail. Two copper wires leave the back, cross the compound on a pole and end in a cabinet nobody has seen inside.

Describe the public switched telephone network, and define the local loop.

Model Answer

The public switched telephone network is the worldwide circuit-switched voice network joining fixed telephones through a hierarchy of exchanges, and the local loop is the pair of copper wires from the subscriber's demarcation point to the local exchange.

Circuit-switched, not packet-switched: before a word is spoken the network reserves an unbroken path between the handsets and holds it at a fixed rate until somebody rings off. Nothing else may use it, and the speech itself carries no address.

The local loop: one pair of conductors, from the demarcation point where the operator's responsibility ends to the local exchange. It runs a few hundred metres in a town and several kilometres in a rural ward, carrying the speech and the current that powers the telephone.

The demarcation point: the boundary of ownership. Towards the exchange faults are the operator's; inside the premises the wiring, the sockets and the telephone are customer premises equipment and the subscriber's. Every fault report begins by settling which side of it the trouble lies.

Analogue outside, digital inside: the loop carries an analogue signal and draws power from the exchange, which is why a telephone rings during a blackout. The exchange digitises the speech on arrival, and everything above is digital.

Built for one job: speech of predictable quality, delayed little enough that two people may interrupt one another. It offers no addressing beyond the dialled number, no error correction and no way to share a path.

The successors: ISDN digitised the loop itself, giving the subscriber two 64 kbit/s channels on the same pair for voice or data. VoIP went further and abandoned the reserved circuit, carrying speech as packets across a data network, which is where telephony has since gone.

Two wires and a cabinet look primitive beside a base station, yet the loop gives speech that never breaks up and a telephone that works when the power does not.

The Master Key: *The local loop is the last copper the operator owns, and the demarcation point is where owning it stops.*

BINDER Question 227

Lifting the handset at a district hospital in Kahama, a nurse dials a supplier in Nairobi. Nothing she does distinguishes that call from one to the ward across the corridor. Between the two lie three levels of switching and a national border.

Using Figure 3.39, describe the path a call takes from the subscriber line up to the international gateway, and state where the mobile network joins it.

Model Answer

A call climbs only as far as it must: the subscriber line reaches the local exchange, which joins two of its own subscribers unaided; a call for another town goes up to the trunk exchange, and a call leaving the country up again to the international gateway.

The bottom two boxes: the subscriber line is the only part of the network belonging to one customer. It ends at the local exchange, or central office, which terminates every loop in its area, supplies line current and ringing voltage, digitises the speech, and switches its own subscribers unaided.

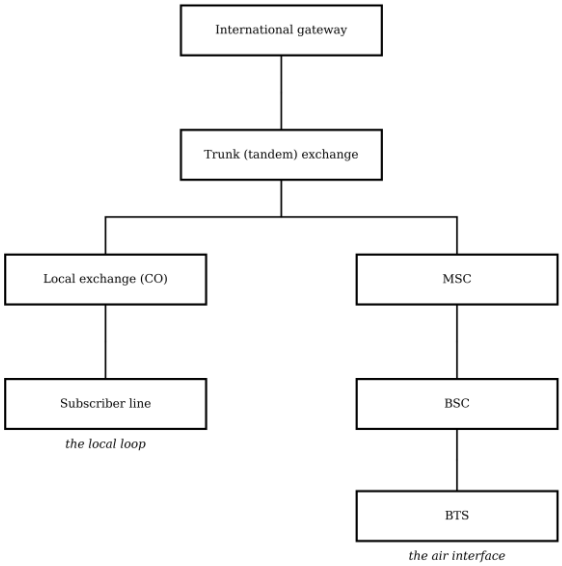


Figure 3.39: *The switching hierarchy. Fixed line on the left, mobile on the right, meeting at the trunk exchange.*

The trunk exchange: a tandem exchange switches between exchanges rather than between subscribers, so it holds no subscriber lines of its own. Traffic between towns crosses it, and it is the junction in the drawing: the fixed branch and the mobile branch both hang beneath it.

The international gateway: the single box at the top, where the national network meets others. It interprets the country code, translates between signalling systems where they differ, and records the traffic on which operators settle charges.

The mobile branch: on the right the climb has different names. A handset reaches a base transceiver station over the air interface, several base stations report to a base station controller, and that controller hands its traffic to a mobile switching centre, the mobile network's own exchange.

Read it as a tree: every box has exactly one parent, so from any subscriber there is one route upward and no argument about where a call goes next. That is what makes a number routable: read from the left, it selects branches downward.

The nurse reaches the top of the tree and comes down another operator's. Her call to the next ward never leaves the bottom box, and the two cost her the same effort and very different money.

The Master Key: *A call rises only as far as the lowest exchange that can see both ends, and never one level higher.*

BINDER Question 228

Forty calls a day at a bus company office in Kariakoo, and the clerk thinks of each as one action: dial, speak, hang up. The network divides that same event into four, and only the third carries any speech.

Describe the four stages of a telephone call, from initiation through to teardown, and state what the network is doing in each.

Model Answer

A call runs in four stages: initiation, in which the exchange sees the handset lifted, collects the digits and digitises the speech; routing, in which a path is chosen and a channel reserved on every link of it; the conversation, during which the circuit is held whether anyone speaks or not; and teardown, in which every channel is released.

Initiation: lifting the handset closes the loop and draws current, the exchange returns dial tone, and the dialled digits are collected. Speech is limited to the band from 300 Hz to 3,400 Hz, sampled 8,000 times a second and coded as 8 bits a sample.

$$8{,}000 \text{ samples/s} \times 8 \text{ bits/sample} = 64{,}000 \text{ bit/s}$$

Routing: the exchange reads the digits and decides how far up the hierarchy the call must go. Signalling runs ahead of the speech and reserves a channel on each link in turn. If any link is full the caller is given an engaged tone at once, and is not queued.

The held circuit: once the far end answers, 64 kbit/s in each direction belongs to that call and to nothing else. It is held through every pause, because the network has made a reservation and a reservation takes no notice of whether it is used.

Teardown: replacing the handset opens the loop. Each reserved channel is released back into its pool along the path in reverse, and the billing record is closed. A circuit never released is a circuit nobody else can have, so signalling that falls silent is timed out.

Three of the four are audible: dial tone says initiation succeeded, ringing tone says a path is reserved the whole way, and the click at the end is teardown. Only the held circuit makes no sound, and it is the stage that costs the network everything.

The Master Key: *The path is built by the signalling, held by the conversation, and dismantled by the hang-up.*

BINDER Question 229

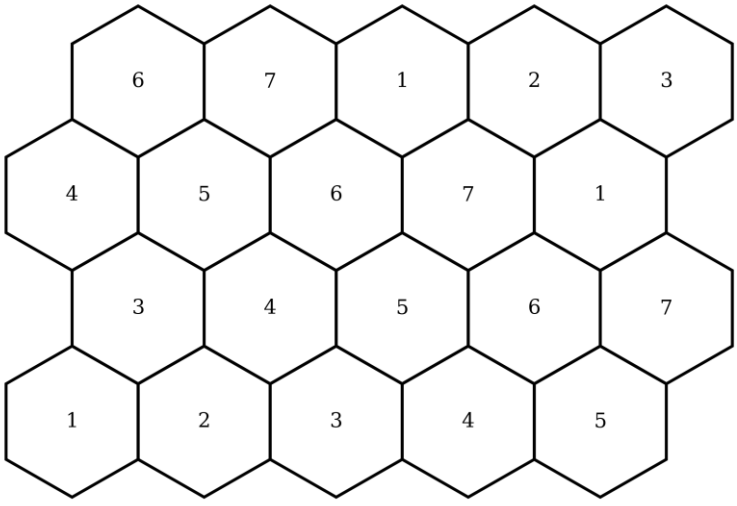
An operator holding one band of spectrum cannot buy a second the day Mikocheni fills up. The band it holds is the band it has, in Mikocheni and everywhere else it covers, and yet it serves millions of subscribers all talking at once.

Using Figure 3.40, explain frequency reuse, and state why no two neighbouring cells may be given the same channel group.

Model Answer

Frequency reuse divides the operator's spectrum into a fixed number of channel groups, seven here, gives each cell one group, and repeats the cluster across the coverage area, so one band serves the whole country many times over.

What a cell is: the area one base station serves. It is drawn as a hexagon because hexagons tile a plane with no gaps and no overlaps and come nearer a circle than squares do. On the ground the boundary is ragged and follows terrain and mast height.



A cluster of 7 channel groups, repeated across the coverage area. No two neighbouring cells ever carry the same group, which is what lets one band of spectrum serve a whole country.

Figure 3.40: *Frequency reuse. A cluster of seven channel groups repeats across the coverage area, and no two neighbouring cells ever share one.*

The cluster of seven: read the numbers across the drawing. Seven groups, each appearing once in a cluster, and then the pattern begins again. A cell carrying group 3 is ringed by 2, 4, 5, 6, 7 and 1, never by a second 3.

Why neighbours must differ: co-channel interference. A handset near a boundary would hear both base stations at similar strength and could not tell the wanted signal from the unwanted, so the two would add as noise. Distance and terrain between cells sharing a group drop the unwanted one far below the wanted.

What reuse buys: capacity without more spectrum. The whole band is spent seven cells at a time, and the eighth cell begins spending it again, so capacity grows with the number of cells rather than with the width of the band.

Adding capacity later: by cell splitting. Halve a cell radius and its area falls to a quarter, so four cells stand where one stood and the band is reused four times as often. The price is four masts, four backhaul links and many more handovers.

Each cell gets a seventh of the channels, which sounds like a loss until you count the cells. Fewer groups in a cluster would give every cell more channels and more interference, and weighing those two against each other is the whole art of planning.

The Master Key: *Spectrum is not used up by distance, so serve more people by making the cells smaller, never by asking for more band.*

REAL Question 230

Three handsets lie on a cloth at a market stall in Tarime, sold as 2G, 3G and 4G, their prices in the ratio one to two to six. The seller can name all three generations and not one number behind them.

Using Table 3.34, compare the mobile generations by date and by typical delivered data rate, and state what each made possible that the generation before it did not.

Model Answer

The generations arrive roughly a decade apart and each lifts the delivered rate by one or two orders of magnitude: analogue voice from about 1980, digital voice with 9.6 kbit/s of data from 1991, packet data of tens to hundreds of kbit/s from 2000, megabits from 2001, tens of megabits from 2009 and hundreds of megabits from 2019.

1G, voice and nothing else: analogue radio from about 1980, carrying speech on a channel much as a broadcast station does. There is no data service to quote a rate for and no encryption, so a conversation could be followed on an ordinary scanner. What it gave was a call made away from a fixed line.

2G, digital and the SIM: GSM from 1991 digitised the air interface, which brought encryption, more calls in the same spectrum, and the SIM that made the subscription separable from the handset. Circuit-switched data reached 9.6 kbit/s, and spare signalling capacity was sold as SMS.

Generation	From	Technology	Typical rate	What it made possible
1G	1980	AMPS and TACS, analogue	None: voice only	A call made away from a fixed line
2G	1991	GSM and IS-95, digital	9.6 kbit/s	SMS, encryption and the SIM
2.5G	2000	GPRS, then EDGE from 2003	40 to 150 kbit/s	Always-on data, billed by volume
3G	2001	UMTS, then HSPA	384 kbit/s to 4 Mbit/s	The mobile web and mobile money
4G	2009	LTE, all packet	15 to 30 Mbit/s	Video, and voice carried as packets
5G	2019	New radio, old and new bands	100 to 300 Mbit/s	Fixed wireless in place of a cable

Table 3.34: *The mobile generations.*

2.5G, the packet step: GPRS from about 2000 put packet switching on the GSM air interface, so a handset could stay attached without holding a circuit and could be charged by the megabyte rather than the minute. EDGE from 2003 changed the modulation and roughly tripled the rate again.

3G and then 4G: UMTS from 2001 carried 384 kbit/s, and HSPA lifted that to several megabits, which is where mobile money and the mobile web became ordinary rather than remarkable. LTE from 2009 threw the circuit away entirely: voice included, everything became packets on an IP network.

5G: from 2019, hundreds of megabits and a radio designed for short delay. Its air interface can approach one millisecond, although what a subscriber measures end to end is nearer twenty. It is the first generation whose fixed wireless service seriously rivals a cable into the building.

Peak against typical: an advertised rate is the peak the technology reaches on an empty cell, close to the mast, in good conditions. What arrives depends on distance, on how many others share the cell, and on the link behind it. The table gives the typical figure; the advertised one is commonly five or ten times higher.

The stall prices the handsets six to one while the rates stand at more than two thousand to one: the 4G handset moves in one second what the 2G handset would have needed about thirty-five minutes to move. That, and not the number in the name, is what the money buys.

The Master Key: *Quote a generation by three things or by none: the year it arrived, the rate it truly delivers, and the one service it made possible.*

REAL Question 231

The bus leaves Makambako at seven and is in Njombe before nine. A passenger spends twenty minutes on a call, and four base stations carry it in turn without the call dropping. Nobody on the bus does anything to bring that about.

Explain what happens during a handover as the bus travels from Makambako towards Njombe, naming the equipment responsible at each stage.

Model Answer

The mobile station measures its serving base transceiver station and its listed neighbours and reports continually; the base station controller watches the serving signal fall and a neighbour's rise, reserves a channel in the better cell and orders the handset to retune; and the mobile switching centre re-routes the call whenever the target cell hangs below a different controller.

The mobile station measures: the handset is not a passive receiver. Between speech frames it measures the strength and quality of its own cell and of the neighbours it has been told to watch, and sends that list back up the air interface several times a second. It reports, and decides nothing.

The base transceiver station carries: the equipment at the mast: the radios, the antennas and the amplifiers for one cell. It passes measurement reports upward and carries traffic in both directions, and it can see no cell but its own.

The base station controller decides: one controller manages many base stations, so it alone can compare the reports and see the bus approaching an edge. It applies a margin: a neighbour must be better by a set amount for a set time before anything happens, which stops a call bouncing between two cells of nearly equal strength.

The order and the retune: the controller reserves a channel in the target cell, then sends the handset a message naming it. In GSM the handset leaves the old channel before taking the new, a break of a few tens of milliseconds that no listener notices, and the old channel is released only once the new one is confirmed.

When the switching centre acts: if the target cell belongs to the same controller, that controller finishes the handover by itself. If it belongs to another, the mobile switching centre moves the call across and updates the visitor location register, the roaming copy of a subscriber record whose master is the home location register, so that an incoming call still finds her; the authentication centre alongside holds the key her SIM is challenged against.

What the passenger notices: nothing, where cells overlap and margins are set sensibly. Where coverage between the towns is thin there is no neighbour good enough to receive the call, and it is not handed over at all: it is dropped, which is the same event with nowhere to go.

Handover is what makes a small cell tolerable. Without it each region would need one enormous cell, frequency reuse would be pointless, and the capacity that reuse buys would go with it.

The Master Key: *The handset measures, the controller decides, the switching centre re-routes; name those three in order and the mechanism explains itself.*

REAL Question 232

A handwritten card goes round a study group at a secondary school in Bariadi. It is neat, it has been copied out twice already, and three of its lines are wrong in a way that would survive any casual reading.

Identify what is wrong with each of the three lines below, give the correct position with the reason, and say why each is an easy thing to write.

1. In an emergency the telephone network connects the caller to 911, free of charge and from any handset.

2. 2G means GSM, so any handset described as 2G, anywhere in the world, is a GSM handset.

3. A circuit-switched call uses the network only while somebody is actually speaking, so the pauses in a conversation cost nothing.

Model Answer

911 is the North American emergency number and reaches nothing here, where the number is 112; 2G names a generation and GSM is only one of the systems within it, IS-95 being the other; and a circuit-switched call holds its channel from the moment it is answered until it is cleared, silences included.

The emergency number: 911 belongs to the North American numbering plan and means nothing on a Tanzanian network, where the number is 112. That 112 is also the number every GSM handset recognises anywhere, with or without a SIM fitted and whatever the credit balance. The temptation is that 911 appears in every imported film and much imported teaching material; an emergency number is a national arrangement, not a property of telephony.

2G was never only GSM: GSM was the dominant 2G system and divided its channels by time. IS-95, sold as cdmaOne, was just as much 2G and divided them by code. What defines the generation is the digital air interface, not the standard that wins a particular market. The temptation is that GSM genuinely did win here, so locally the two words do point at the same handsets, and the habit travels badly.

The silent circuit: the reservation takes no notice of use. Once the far end answers, 64 kbit/s in each direction belongs to that call until teardown, and a two-way conversation is silent for roughly half its duration, so about 32 kbit/s of what is reserved carries nothing. That waste is exactly what packet switching removes, and it is why data never ran well over a voice network.

What the three share: each states a local habit as a universal fact: one continent's emergency number as the emergency number, one market's dominant standard as the whole generation, and the behaviour of a shared network as the behaviour of a reserved one. Ask of any such sentence whose country it is true in, which standard it names, and what is reserved as against what is used.

Correct the card rather than discard it, because a card copied twice will be copied again. Fix the 112 line first: the other two would cost marks, and that one could cost more than marks.

The Master Key: *Every confident sentence about telephony hides a country, a standard and a year; find all three before believing it.*

HOT Question 233

Thirty computers, a small budget and no telephone line at a school at Chake Chake. The nearest fixed-line cabinet stands in the town centre, and the copper was never laid. On the roof of the science block, a handset shows four bars of 4G.

Assess why a school in that position is far likelier to reach the Internet over a mobile network than over a fixed line, and reach a judgement about what that implies for the way its own network is designed.

Model Answer

Fixed access needs a physical path built to one building and repaid by one subscription, while mobile access needs only that the school stand inside a cell already built for everybody; the school will therefore connect over mobile, and its own network must then be designed around a single metered uplink of varying quality.

Why the fixed line is absent: a local loop serves exactly one subscriber, so every metre of it is repaid by one rental. An operator builds it where the density of paying lines covers the trench, and a school two kilometres past the last cabinet is one line at the end of two kilometres of work nothing else will use.

Why the cell is present: a base station serves everyone inside its cell, so its cost is spread across thousands of subscriptions, and it went up for the market and the bus stand rather than for the school. Connecting the school costs the operator one more subscription on equipment already standing and already powered.

What the link really gives: shared capacity, not a private wire. Expect 15 to 30 Mbit/s while the cell is quiet and appreciably less at four in the afternoon when the town is on it, latency higher than a wire's and, worse, less steady, and every byte counted and charged.

The first consequence: one public address, and it changes. Everything inside runs on private addressing behind network address translation, so nothing in the school can be reached from outside unless the operator is paid to arrange it. Plan the school as a client of the Internet and not as a server on it.

The sharper consequence: metering decides what belongs on the link. Operating system updates, antivirus signatures and video are the three heaviest consumers, so a caching server that fetches an update once and serves it thirty times repays itself within a term, and anything bulky is scheduled for the small hours.

The judgement: connect over mobile, with a roof antenna aimed at the serving mast and a router that will take a second operator's SIM, and then build inwards. The file store, the printing, the accounts and the teaching material belong on a machine in the school, because the uplink is both the narrowest link and the single point of failure. Revisit when fibre reaches the town, and not before.

The mobile network was not built for this school, which is exactly why it is available to it. Design for what a shared, metered, variable link is good at, and the thirty machines will still teach on the days the link does not.

The Master Key: *Connect to the network somebody else has already paid for, then build inwards until an outage is an inconvenience rather than a holiday.*

HOT Question 234

Two quotations reach a bank branch on Mafia. The first is a leased 2 Mbit/s circuit at a fixed fee, reserved whether the branch uses it or not. The second is a shared link ten times as fast for two thirds of the money.

Judge whether circuit switching retains any real advantage over packet switching, and defend the answer you reach.

Model Answer

It retains exactly one: a reservation made before the first bit moves, which yields a constant rate, a fixed delay and no jitter for as long as the circuit is held. That advantage is genuine and worth paying for in a few cases, and nowhere near enough to keep circuit switching as the way a public network is built.

What the reservation delivers: capacity settled at set-up rather than moment by moment. The delay is propagation plus coding and does not vary, there is no queue along the path so there is no jitter, and no other traffic can crowd the circuit because none is admitted. Either it is established or the call is refused.

Why that virtue is narrow: it earns its price only where traffic is a steady stream running for hours and a lapse is expensive. Almost nothing is. What a school or a branch office sends is bursty: a request, a silence, a file, a longer silence. Reserving a constant rate for that buys idleness.

The arithmetic of the waste: even voice, the traffic it was designed around, is silent for about half a conversation, yet its 64 kbit/s in each direction is held throughout, so some 32 kbit/s carries nothing. A data link is worse: idle most of the day, saturated for twenty minutes at closing.

What packet switching gave up: the guarantee. Packets queue, and a queue that fills discards. The honest comparison is therefore not certainty against chaos but certainty always against certainty nearly always, at two thirds of the price on a link ten times as fast, and the second has won that argument.

How the advantage survived: the guarantee did not die; it moved. It is bought inside packet networks now as classification and priority queuing, as a committed rate in a contract, and, in the optical transport beneath, as a wavelength reserved end to end. Every one is a reservation, and not one is a telephone exchange.

The judgement: circuit switching has no future as a way of building a network and a permanent place as a way of buying an assurance. Take the shared link at the branch, and keep a small guaranteed circuit only where the assurance is itself the product, as on a broadcaster's feed or a settlement line that cannot be late. Even there, what is bought is not the switching but the promise it happened to make easy.

The century of the held circuit ended, not because reserving capacity was a poor idea, but because reserving it one conversation at a time was. The reservation was the valuable part; the exchange that enforced it was only the machinery of its day.

The Master Key: *Buy the guarantee where losing it would cost more than the idleness, and never buy the circuit for its own sake.*

Chapter Four

OBJECT ORIENTED PROGRAMMING WITH C++

Everything in the three chapters before this one described a machine. This chapter asks you to instruct one, and that is a different kind of knowing. A candidate cannot describe a class into existence. They must type it, compile it, read what the compiler says back, and make it run. Half the marks in this subject are eventually won or lost at a keyboard, and this is the chapter where that begins.

The chapter opens by settling what object oriented programming actually is, and it settles it in four words that must be kept apart: encapsulation, abstraction, inheritance and polymorphism. It then builds the language from the ground, one floor at a time. How a program becomes a running process. How it is written and how it talks to a screen. Types, variables and the places a name is visible. Operators and the order they are applied in. Choosing between paths, and repeating work. Functions, including the kind that call themselves. Arrays and strings, and the exact point at which a string overruns the space it was given. Then classes, the objects made from them, and the three ideas that let one class stand on another.

Two habits earn marks in this topic, and both are habits of doubt. The first is to trace rather than to guess: work through a loop with a pen and a table of values, and the answer stops being an opinion. The second is to compile: a program that looks correct and a program that is correct are different objects, and only one of them is worth marks. Every program printed in this chapter has been compiled and run, and every output beside it is what the machine actually printed.

Paradigms, objects and the four pillars

This section settles the vocabulary the rest of the chapter stands on. It separates the two ways a program can be organised, names the four principles of object orientation and keeps abstraction and encapsulation apart, draws the five shapes inheritance can take, and shows the same task written twice, once as a list of instructions and once as a set of objects.

BINDER Question 1

A class at Kahama wrote the same fee calculator twice in one term, the second time starting by naming what the school deals with: a student, a fee, a receipt. It came out half the length.

Define a programming paradigm. Distinguish procedural programming from object oriented programming, and state which of the two C++ supports.

Model Answer

A programming paradigm is a way of organising a program, a decision about what the program is made of, taken before any line of it is written.

Procedural programming: the program is built from procedures, which in C++ are functions, and those functions act on data handed to them. The data and the operations on it are declared separately, so any function that can see a variable is free to change it.

Object oriented programming: the program is built from objects, and each object holds its own data together with the functions that act on that data. Work is requested of an object rather than performed on loose data lying somewhere else in the program.

Where the data lives: this is the difference that actually matters, and the one to write down first. In the procedural style a fee total is a variable many functions may touch. In the object oriented style it belongs to one object, and only that object can reach it directly.

What C++ supports: both of them, and more besides. C++ is a multi paradigm language, so a valid C++ program may be wholly procedural, wholly object oriented, or a mixture of the two. Writing in C++ does not by itself make a program object oriented.

Question asked of the program	Procedural	Object oriented
What is the program made of	Functions	Objects
Where does the data sit	Apart from the functions	Inside the object with its functions
Who may change a value	Any function that can see it	The object that owns it
What grows when the program grows	The number of functions and shared variables	The number of classes
How is a new kind of thing added	New functions, and edits to old ones	A new class, often built on an old one

Table 4.1: *The two paradigms compared by the questions that separate them. Only the second row is a difference of principle; the rest follow from it.*

Name the paradigm by where the data sits, never by the language printed on the front of the book.

The Master Key: *A paradigm decides what a program is made of; procedural code hands data to functions, object oriented code keeps data and its functions together, and C++ permits both.*

BINDER Question 2

The carpenter at Sengerema keeps one drawing of a school desk above his bench, and has built forty desks from it. Not one of the forty is the drawing.

Define a class, an object, an attribute and a method, and explain the relationship between a class and an object.

Model Answer

A class is a description of a kind of thing, and an object is one actual thing built to that description.

Class: a user defined type that states what data every thing of its kind will hold and what operations may be performed on that data. A class occupies no memory for those values, because it describes them rather than storing any.

Object: one instance of a class, created when the program runs, holding its own copy of every value the class declared. Two objects of one class share their structure completely and their values not at all.

Attribute: a data item declared inside the class, also called a data member or a field. In Program 4.1 the attributes are the name and the average mark, and each Student object carries its own pair of them.

Method: a function declared inside the class, also called a member function, which acts on that object's own attributes. It is called on an object using the dot operator, as juma.report() calls report on the object juma.

The dot operator selects the object: the same method text serves every object, so the call must say which object it is about. That is the work the dot does: juma.report() and asha.report() run identical code and print different lines, because each call reaches the attributes of the object named before the dot.

What each of them costs: the class is written once and occupies no memory while the program runs. Every object costs memory for its own attributes, and none for its methods, because the methods exist once and are shared by all objects of the class.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  class Student
 6  {
 7  private:
 8      string name;
 9      int averageMark;
10  public:
11      void setDetails(string n, int mark)
12      {
13          name = n;
14          averageMark = mark;
15      }
16      void report()
17      {
18          cout << name << " scored " << averageMark << endl;
19      }
20  };
21
22  int main()
23  {
24      Student juma;
25      Student asha;
26      juma.setDetails("Juma", 71);
27      asha.setDetails("Asha", 84);
28      juma.report();
29      asha.report();
30      return 0;
31  }
```

Output

```
Juma scored 71
Asha scored 84
```

Program 4.1: *One class, two objects. Each object answers with its own values, although both run the same report function.*

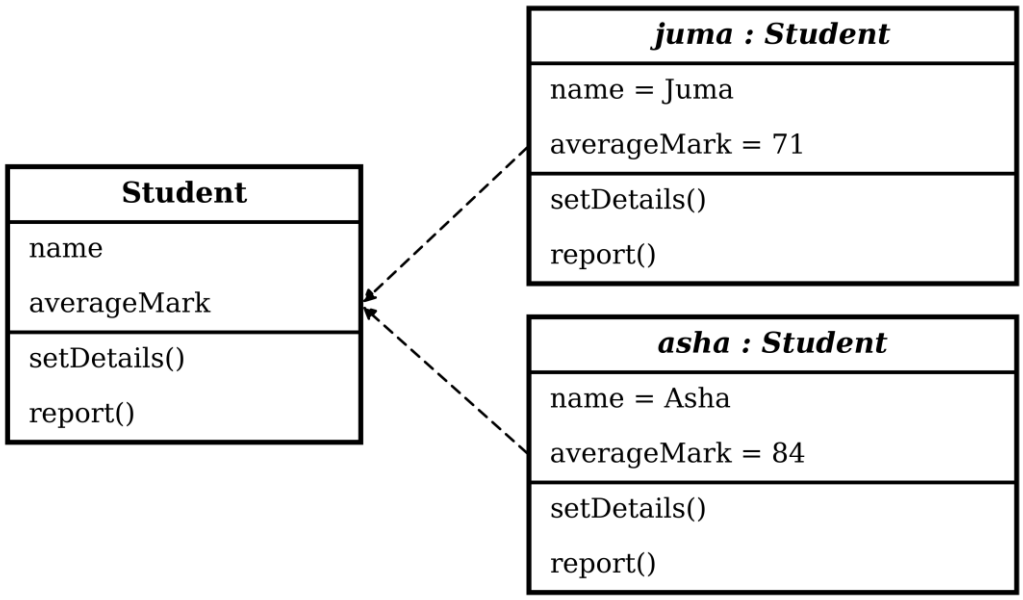


Figure 4.1: *One class and two objects made from it. The attribute names come from the class and are the same in both objects; the values belong to the objects and differ.*

The class is written once and costs nothing at run time; the objects are made many times and cost memory each.

The Master Key: *A class describes and an object exists, so attributes are the data the class declares, methods are the functions it declares, and every object gets its own attributes while sharing the methods.*

BINDER Question 3

Ask ten students to name the principles of object oriented programming and the answers come back four, five and seven, with reusability somewhere in the list. Reusability is not a principle.

Name and define the four principles of object oriented programming, and state the C++ construct that expresses each one.

Model Answer

There are exactly four principles: encapsulation, abstraction, inheritance and polymorphism.

Encapsulation: binding data together with the functions that act on it inside one unit, and controlling which parts of the program may reach that data. In C++ the class is the unit and the access specifiers private, protected and public do the controlling.

Abstraction: exposing only what a user of a class needs in order to use it, and keeping the manner of doing it out of sight. In C++ this is the public interface, the set of functions a caller may use without ever seeing how they work.

Inheritance: defining a new class in terms of an existing one, so that it acquires the members of the older class and adds or changes what it needs. In C++ this is written with a colon after the new class name, as in class Dog : public Animal.

Polymorphism: one name standing for several behaviours, with the right one chosen for the object in hand. In C++ it takes two forms: overloading, settled by the compiler, and virtual functions called through a base pointer, settled while the program runs.

Why the number itself matters: reusability, maintainability and efficiency are often listed alongside these four, and they do not belong. They are benefits a program collects when the four are observed. A question asking for the principles of object orientation is asking for four things, and a list of seven loses marks.

Principle	What it does	The C++ construct that carries it
Encapsulation	Keeps data and its functions together and restricts reach	class, and private, protected, public
Abstraction	Shows what a class is for and hides how it works	The public interface, and abstract classes
Inheritance	Builds a new class on an existing one	class Derived : public Base
Polymorphism	Lets one name select the behaviour that fits the object	Overloading, and virtual functions

Table 4.2: *The four principles and the construct each one is written with. Nothing else belongs in this list; reusability and maintainability are consequences, not principles.*

Four, and only four. Anything else on the list is a benefit that follows from observing them.

The Master Key: *Encapsulation, abstraction, inheritance and polymorphism, expressed in C++ by access specifiers, the public interface, the inheritance colon, and overloading with virtual functions.*

BINDER Question 4

Two answers came back from the same class at Ukerewe: encapsulation hides data, and abstraction hides data. If two words mean one thing, one of them is being wasted.

Distinguish encapsulation from abstraction, and explain why defining both as hiding is not enough to tell them apart.

Model Answer

Encapsulation is the mechanism and abstraction is the result: encapsulation is how a class controls reach into its data, and abstraction is what the class chooses to show of itself.

Encapsulation is about access: it binds data and the functions that act on it into one unit and then says who may touch what. It is a property of the implementation, and it is enforced by the compiler through the access specifiers, which reject the offending line outright.

Abstraction is about attention: it decides what a user of the class needs to think about and removes everything else from view. A person withdrawing money needs a withdraw operation and a balance; they do not need to know whether the balance is held as a long or a double.

One is used to achieve the other: they are related, which is exactly why they blur. Marking data private is encapsulation; the fact that a caller can now use the class without knowing what is behind that wall is abstraction. The first is the tool and the second is what the tool buys you.

The test that separates them: ask what would break if the rule were dropped. Drop encapsulation and any code anywhere may corrupt the data. Drop abstraction and the data stays safe, but every caller must now understand the internals, so any change inside the class breaks the callers.

Encapsulation protects the data from the caller; abstraction protects the caller from the data.

The Master Key: *Encapsulation is the mechanism that restricts access, abstraction is the simplified view that results; if you can only say hiding, you have not distinguished them.*

BINDER Question 5

A dispensary at Mkuranga runs three record cards, and the last two copy out the first in full before adding four lines of their own. The clerk has written the same eleven headings by hand for years.

Define inheritance, name the five forms it can take, and state which of them C++ supports directly.

Model Answer

Inheritance is the definition of a new class in terms of an existing one, so that the new class acquires the members of the old and then adds or replaces what it needs.

The vocabulary: the existing class is the base class, sometimes called the parent or superclass, and the new one is the derived class, the child or subclass. The relationship is read as is a kind of: a derived class should be a kind of its base, or it should not inherit from it.

Single and multilevel: single inheritance is one derived class built on one base. Multilevel is a chain, where a class is derived from a class that was itself derived from another, so a member declared at the top is reached through two steps.

Hierarchical and multiple: hierarchical inheritance is several classes derived from one common base, which is the dispensary's three cards. Multiple inheritance is the opposite arrangement, one class derived from two or more independent bases at once.

Hybrid, and what C++ allows: hybrid is any combination of the others, most often hierarchical joined to multiple, which produces the diamond in Figure 4.2. C++ supports all five directly, multiple inheritance included. Students are often taught, wrongly, that an interface must be used instead; that is a rule from another language.

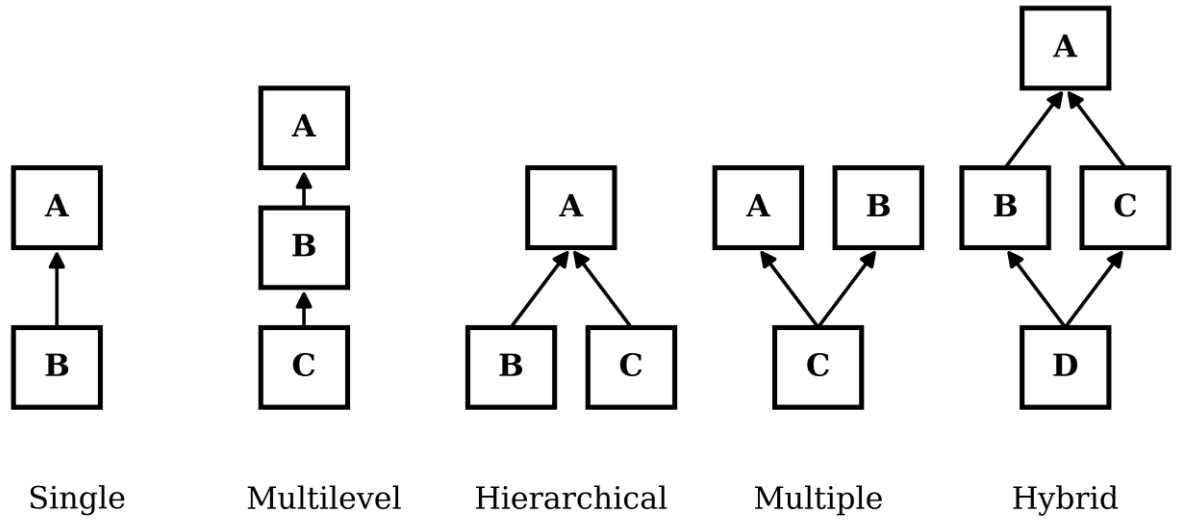


Figure 4.2: *The five shapes of inheritance. Every arrow runs from the derived class to its base, because that is the direction the dependency actually runs.*

Draw the arrows before you write the classes, and let the shape tell you whether the design is honest.

The Master Key: *Inheritance builds a new class on an old one in five shapes, single, multilevel, hierarchical, multiple and hybrid, and C++ supports every one of them directly.*

REAL Question 6

One exercise book at Kalambo Secondary holds every borrowed title as a line, ruled through when a copy is lost. Asked how many biology books remain, the librarian counted wrong twice last term.

Design the classes for the library described below. Name each class, list its attributes and its methods, and justify why you split the design where you did.

The library holds books and periodicals.

Every item has a title, an accession number and a shelf location.

A book also has an author and an edition; a periodical has a volume and an issue date.

Any item may be borrowed by a student, who has a name, a class and an admission number.

A loan records which item, which student, the date out and the date due.

The library must be able to report which items a given student has out.

Model Answer

Four classes carry this design: Item, with Book and Periodical derived from it, Student, and Loan.

Item, Book and Periodical: Item holds the title, the accession number and the shelf location, because every item has all three. Book adds the author and the edition; Periodical adds the volume and the issue date. Both are derived from Item, which is hierarchical inheritance.

Student: holds the name, the class and the admission number, with a method to report the items currently out. A student is not a kind of item and holds none of an item's data, so it is a separate class and not a derived one.

Loan: holds the date out and the date due, and identifies one item and one student. A loan is the relationship itself, which is why it is neither a student nor an item, and why it is the class that answers the librarian's question.

Why the split falls there: the test is whether a derived class is a kind of its base. A book is a kind of item, so it inherits. A loan is not a kind of student and not a kind of item, so it does not; instead it refers to both, which is composition rather than inheritance.

What is deliberately not a class: the date out and the date due are attributes of a loan, not classes of their own, because nothing in the specification asks a date to do anything. Make a class when a thing has both data and behaviour; a value that is only ever stored and compared stays an attribute.

How the librarian's question is answered: the report of what a student has out is a search of the loans for one student, so it belongs to Loan or to a class holding all the loans, not to Student. Putting it in Student would force every student object to know about every loan in the library.

What the design buys: the title and the accession number are declared once rather than twice, so they cannot drift apart. Adding a third kind of item later, a map or a wall chart, means one new derived class and no edit at all to anything already working and tested.

Ask is a kind of before you write a colon, and the design will usually settle itself.

The Master Key: *Derive only where the derived thing is genuinely a kind of the base thing; where two classes merely refer to each other, use composition and give the relationship its own class.*

REAL Question 7

The two programs below print the same three lines. Adding a second mark to one of them took a whole evening, and the printout came back with Asha holding Neema's score.

Compare Programs 4.2 and 4.3, which produce identical output. Explain what changes structurally between them, and state which one is safer to extend and why.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      string name[3] = {"Juma", "Asha", "Neema"};
 8      int mark[3] = {71, 84, 66};
 9      for (int i = 0; i < 3; i++)
10      {
11          cout << name[i] << " scored " << mark[i] << endl;
12      }
13      return 0;
14  }
```

Output

```
Juma scored 71
Asha scored 84
Neema scored 66
```

Program 4.2: *Three students held in two parallel arrays, printed in one loop.*

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  class Student
 6  {
 7  private:
 8      string name;
 9      int mark;
10  public:
11      void setDetails(string n, int m)
12      {
13          name = n;
14          mark = m;
15      }
16      void report()
17      {
18          cout << name << " scored " << mark << endl;
19      }
20  };
21
22  int main()
23  {
24      Student roll[3];
25      roll[0].setDetails("Juma", 71);
26      roll[1].setDetails("Asha", 84);
27      roll[2].setDetails("Neema", 66);
28      for (int i = 0; i < 3; i++)
29      {
30          roll[i].report();
31      }
32      return 0;
33  }
```

Output

```
Juma scored 71
Asha scored 84
Neema scored 66
```

Program 4.3: *The same three students held as objects of one class, printed in one loop.*

Model Answer

Both programs hold three students and print three lines, but Program 4.2 keeps the names and the marks in two separate arrays, while Program 4.3 keeps each student's name and mark together inside one object.

What holds the data together: in Program 4.2 the only thing joining the name Asha to the mark 84 is that both sit at index one. That agreement lives in the programmer's head. In Program 4.3 the two values are members of the same object and cannot be separated by an indexing mistake.

How the reported failure happened: an evening's edit added a second array of marks and inserted a student into the names array without inserting into both mark arrays at the same position. Every later name was then one place out of step with its mark, and the program printed happily, because nothing it does could detect that.

What happens when a field is added: adding a second mark to Program 4.2 means a third array, a third initialiser and a third subscript at every point of use, each of which is an opportunity to fall out of step. In Program 4.3 it means one more data member and one more line in setDetails.

Where the printing rule lives: Program 4.2 states the layout of a report inside main, so a second report elsewhere would repeat it. Program 4.3 states it once inside report, and every caller gets the same layout, because there is only one place it is written.

What it costs: Program 4.3 is longer and does more work to reach the same three lines, which is the honest objection to it. The cost is paid once, at the class; the saving is collected every time the data is used or extended.

Program 4.3 is safer to extend, because the relationship between a name and a mark is enforced by the compiler rather than remembered by the programmer.

The Master Key: *Parallel arrays keep related values apart and rely on the index to rejoin them; an object keeps them together, so a change of shape touches one class instead of every use.*

HOT Question 8

A teacher at Karagwe asks for the mean of forty marks. One student writes nine lines; another writes a class with a constructor, two accessors and a private vector, and hands in fifty.

A candidate argues that because object orientation is the better paradigm, every program should be written with classes. Evaluate this claim and reach a judgement.

Model Answer

The claim is wrong as stated, because object orientation is not better in the abstract; it is better under conditions, and a program that reads forty marks and prints their mean does not meet them.

The case for always using classes: a habit applied everywhere is a habit that does not have to be decided each time, and small programs grow into large ones without announcing it. Code written with classes from the start absorbs a new requirement without being rebuilt around one.

The case against: a class costs lines, and lines are places for errors to live. Forty marks and a mean need one array and one loop. Wrapping that in a class adds a constructor, accessors and a file of declarations, and none of it makes the mean more correct or the program easier to read.

What actually decides it: the number of distinct kinds of thing the program deals with, and how long it must live. One kind of thing and one run means a procedure. Several kinds, each with its own data and rules, and a program that will be edited by someone else next year, means classes.

A second condition, often forgotten: whether the data has rules attached to it. A balance that must never go negative wants a class, because the rule can then be enforced in one place. A running total with no rule attached gains nothing from being wrapped.

The verdict: use classes when the program models more than one kind of thing, or when data carries rules that must hold everywhere, or when the program will outlive the person who wrote it. Otherwise write the procedure, and be willing to convert later.

Why the conversion is cheap: moving a well written procedural program to classes is mostly a matter of gathering variables that always travel together, which is a mechanical change. That is why starting simple is defensible, and why starting complicated rarely is.

The paradigm is chosen for the problem, and the mean of forty marks is not a problem that asks for one.

The Master Key: *Reach for classes when the program has several kinds of thing, or data with rules, or a life longer than its author; otherwise the procedure is the right answer and can be converted later.*

HOT Question 9

A note on a laboratory wall at Lushoto reads: this room teaches object oriented programming, therefore all work here is object oriented. Underneath it somebody has written the word therefore twice and put a question mark after it.

Evaluate the claim that any program written in C++ is an object oriented program. Reach a judgement, and say what would have to be true for the claim to hold.

Model Answer

The claim is false, because a paradigm is a property of how a program is organised and not of the language it is written in, and C++ places no requirement on how a program is organised.

What is true in the claim: C++ does provide every construct object orientation needs, and provides them as part of the language rather than as a library. A C++ program can be object oriented without help, which is more than can be said for the language C++ was built from.

Why it still fails: a language offering a construct does not oblige anyone to use it. Program 4.2 is valid C++ and contains no class at all; it is a procedural program that happens to be compiled by a C++ compiler, and calling it object oriented empties the term of meaning.

The stronger version of the claim: someone might argue that using a class is enough. It is not. A program with one class holding public data that main reaches into directly has borrowed the syntax and refused the discipline, and it will fail in exactly the way a procedural program fails.

What would make it true: the claim would hold only in a language where organising a program any other way is impossible, so that every program must be a set of classes. C++ is deliberately not such a language, and its usefulness for systems work depends on that.

Why the distinction is worth keeping: if the language decided the paradigm, then a design fault could be fixed by changing compiler, which is plainly absurd. Keeping the two apart is what makes it possible to say that a particular C++ program is badly organised, and to say what would improve it.

The verdict: a program is object oriented when its data is held inside objects that control access to it, and its behaviour is requested of those objects. Judge a program by that description, never by the compiler used, and never by the presence of the keyword class.

The language supplies the tools; the programmer supplies the paradigm, and only one of those two is decided by the compiler.

The Master Key: *C++ permits object orientation but never requires it, so a program is object oriented only when its data lives in objects that control access to it and its work is asked of those objects.*

From source code to a running program: the toolchain, the build pipeline and the four error classes

This section follows a source file from the moment it is saved to the moment it is a running process, and names the artefact every stage of the toolchain produces. It gives the linker the place it is usually denied, because the message a missing definition produces cannot be read without it. It then settles the four classes of programming error by the only test that separates them, the stage at which each one is caught, and takes apart two faulty programs that are commonly filed under the wrong class. It closes by asking what a clean build and a clean run are actually evidence of.

BINDER Question 10

The folder held four files where the class at Msalala had typed one: the source, a file tens of thousands of lines long, a page of something that was not C++, and a file the editor refused to open.

Describe the stages by which a C++ source file becomes a running process, naming what each stage produces.

Model Answer

A source file becomes a running process in six stages, each one consuming what the stage before it produced: preprocessing, compilation, assembly, linking, loading and running.

Preprocessing: the preprocessor carries out the directives, the lines beginning with a hash, and hands on one file with no directives left in it. Every included header has been pasted in place of the line that asked for it, so eight lines of source become 36522 lines on the compiler used here. That file is the translation unit.

Compilation: the compiler checks the translation unit against the rules of the language and translates it into assembly, the processor's own instructions written as text. Those 36522 lines leave this stage as 67. Syntax and type errors are reported here, and nothing further runs if either is found.

Assembly: the assembler turns that text into machine code and writes it to an object file, whose name ends in .o. The object file holds real instructions and is still not a program, because every address belonging to something defined elsewhere is left blank.

Linking: the linker takes one or more object files with the already compiled library code they use, fills in every address left open, and writes one executable file. This is the stage most often left out of an account of compilation, and the next answer is about it.

Loading and running: the executable sits on disk doing nothing until it is started. The loader copies it into memory, arranges its stack and passes control to the entry point. From that moment there is a process, the program in memory with a state of its own, and the process is what produces output.

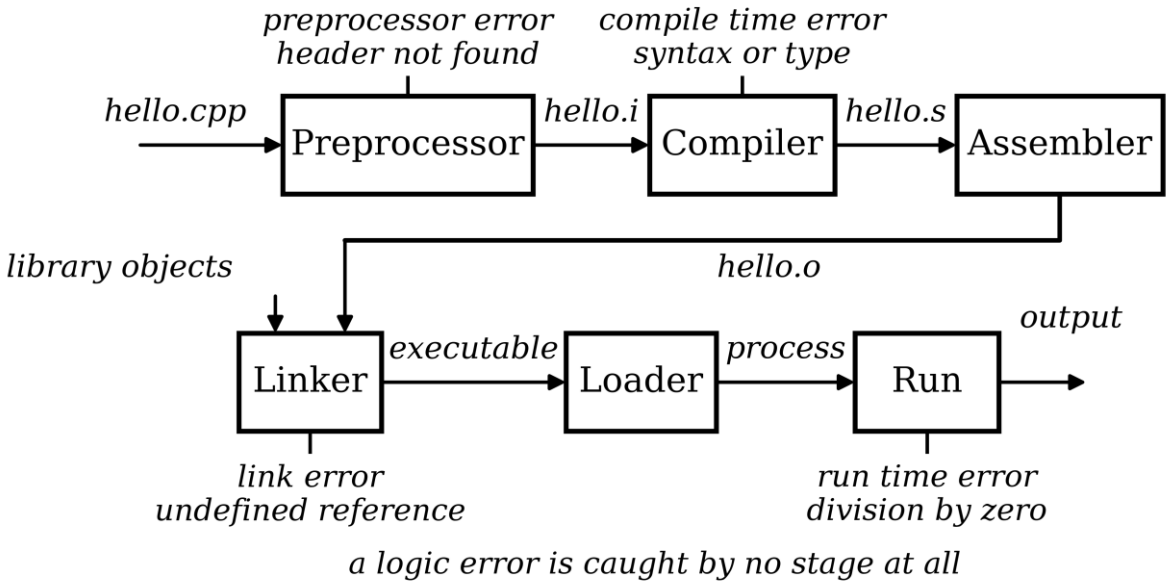


Figure 4.3: *From source file to running process. Each stage names what it consumes and what it produces, and the linking stage, the one most often left out, is what turns object code into something that can run.*

One command runs the first four stages and then deletes the evidence, which is why compiling looks like a single step; name the artefact on every arrow and it stops looking like one.

The Master Key: *Preprocessing yields the translation unit, compilation yields assembly, assembly yields an object file, linking yields the executable, and loading yields the process that runs.*

BINDER Question 11

The compiler had said nothing at all, and then the screen filled with a message naming something called ld and the words undefined reference to. The student spent an hour hunting a missing semicolon that was never there.

Explain what the linker does, and why an object file cannot be run directly.

Model Answer

The linker joins object files to one another and to the library code they use, filling in the address of every name used in one file and defined in another, and it alone produces the executable.

What an object file holds: the machine code of the functions defined in that one file, and a list of the names it uses without defining. Meeting a call to a function declared but not defined here, the compiler emits the call and leaves the address blank, because the definition may sit in a file it will never see.

Why separate compilation needs a linker: each source file is compiled on its own into its own object file, and no compilation ever sees another. The linker finds each missing definition and patches the address in, so it is the first moment at which a whole program exists.

The standard library: an include line brings in declarations only. The code behind cout was compiled long ago and sits in a library, and the linker attaches it. Nothing you write defines it and nothing you compile contains it.

The message a missing definition gives: Program 4.4 declares a function on line 4 and never defines it. The compiler is satisfied, because a declaration is a promise; the linker calls the promise in and reports undefined reference to total, then ld returned 1 exit status. The name ld is the linker itself.

Why the correction matters: students are commonly taught that compiling produces the program and that running copies it into memory, with linking never mentioned. On that account the message above cannot be explained at all, and the hour goes into the source file hunting a fault that is not there.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int total(int a, int b);
 5
 6  int main()
 7  {
 8      cout << total(60, 70) << endl;
 9      return 0;
10  }
```

Linker says

```
/usr/bin/ld: total.o: in function `main':
total.cpp:(.text+0x13): undefined reference to `total(int, int)'
collect2: error: ld returned 1 exit status
```

Program 4.4: *A function declared on line 4 and never defined. The compiler accepts the file and produces total.o; the linker refuses to turn it into a program.*

Compiling checks one file; linking builds one program, and only the second produces something that can run.

The Master Key: *The linker resolves every name used in one place and defined in another, joining object files and library code into a single executable; an object file still has holes where those addresses belong.*

BINDER Question 12

Syntax, type, run time, logic: four faults, and no two of them are found at the same moment. Three announce themselves; the fourth never does.

Define the four classes of programming error and state, for each, the stage at which it is caught.

Model Answer

There are four classes: syntax and type errors, both caught by the compiler before any program exists; run-time errors, which stop a program that has already started; and logic errors, which no tool catches at any stage.

Syntax error: the text breaks a rule of grammar, so it is not C++ at all: a missing semicolon, an unclosed brace, a keyword misspelt. The compiler reports it and refuses to produce an object file, so nothing is linked, loaded or run.

Type and constraint error: the text is grammatically well formed and breaks a rule about meaning: storing a string in an int, calling a function with two arguments where it takes three, using a name never declared. The compiler catches these too, and the build stops as before.

Be consistent about the middle class: it is also called a semantic error, and students are often given two definitions of it a page apart, one saying it stops compilation and one saying the compiler cannot catch it. In C++ these faults are diagnosed and the build stops. A fault that reaches the run was never in this class.

Run-time error: the build succeeded and the program fails while running: a division by a value that turns out to be zero, an index outside an array, a file that is not there. No tool saw it coming, because the fault depends on values that exist only while the program runs.

Logic error: the program builds, runs, ends normally and gives the wrong answer. Nothing is broken as far as any tool can tell, because the program did exactly what it was told and what it was told was wrong.

Class of error	The rule it breaks	Caught at	What is seen
Syntax	A rule of grammar, so the text is not C++	Compilation	A message and a line number, and no executable
Type or constraint	A rule about meaning, on text that is grammatically legal	Compilation	A message and a line number, and no executable
Run time	A step the machine cannot carry out on the values it was given	While the program runs	The program stops before its last line, exit status not zero
Logic	No rule at all: the instructions are legal and wrong	No stage catches it	A clean build, a normal exit, and a wrong answer

Table 4.3: *The four classes separated by the stage that catches each. Only the last two rows describe a program that ever runs, and only the last describes one that finishes.*

Ask when a fault is caught before you ask what it is called, and the four classes stop overlapping.

The Master Key: *Syntax and type errors stop the compiler, run-time errors stop the running program, and a logic error stops nothing at all, which is why only a reader can find it.*

BINDER Question 13

One file, carried home from Ushetu on a memory stick, built cleanly at school and produced a warning on the machine at home. The student decided the stick had damaged it and typed the whole program again.

Explain what a language standard is, why a compiler must be told which one to use, and what happens when it is not told.

Model Answer

A language standard is the published specification that fixes what C++ is, which constructs are legal, what each means and what the library must provide; a compiler must be told which one to apply, because it can apply several and will otherwise choose for you.

What a standard is: an international specification, revised on a schedule. C++98 was the first, and C++03, C++11, C++14 and C++17 followed it, each adding constructs and occasionally withdrawing one. A construct is therefore not simply legal C++; it is legal under a named standard.

Why the compiler must be told: one compiler implements several standards, so a single file has more than one lawful reading. The flag -std= chooses, as in -std=c++17. Programs 4.5 and 4.3 are the same eight lines built twice, and the number each prints is the standard it was given.

What happens when it is not told: the compiler falls back on its own default, which is a property of the compiler's age and not of the code. An environment that hides the flag in a project setting lets a class work for a year without learning that the setting exists.

The case that shows the cost: the keyword override was added in C++11. Under C++17 it is accepted without comment; under the C++98 default that older toolchains still ship with, the same file builds and runs but warns that override is available only with C++11 or later. The program appears to work while the check it asked for is granted as an extension.

Portability of source only: a standard settles what may be written, not what runs. Recompiled elsewhere the same source behaves alike, but the executable never moves between machines.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      cout << __cplusplus << endl;
7      return 0;
8  }
```

Output

```
201703
```

Program 4.5: *The predefined macro __cplusplus reports the standard in force. Built with -std=c++17.*

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      cout << __cplusplus << endl;
7      return 0;
8  }
```

Output

```
199711
```

Program 4.6: *The same eight lines, built with -std=c++98. Not one character of the file has changed.*

Name the standard on every build, and use the one everybody else in the room is using; leaving it unnamed lets somebody else choose.

The Master Key: *A standard fixes what C++ means, so select it with -std= on every build; the default belongs to the compiler and may be older than the code being taught.*

REAL Question 14

Program 4.7 is offered, in more than one place, as a demonstration of a logic error. Type it in exactly as printed and no answer arrives at all.

Program 4.7 below is faulty. Name the fault or faults, state exactly what the compiler does with the program, and correct it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int i;
 7      for (i = 0; i < 3; i++);
 8      {
 9          cout << "loop" << endl;
10          continue;
11      }
12      return 0;
13  }
```

Compiler says

```
loop.cpp:7:5: warning: this 'for' clause does not guard... [-Wmisleading-indentation]
loop.cpp:10:9: error: continue statement not within a loop
```

Program 4.7: *The listing as it is usually printed, offered as an example of a logic error.*

Model Answer

The program does not compile, so whatever else it demonstrates it cannot be a logic error, and it carries two faults: a stray semicolon at the end of line 7, and a continue on line 10 that is inside no loop.

The first fault, the semicolon on line 7: a semicolon standing alone is an empty statement, and the body of a for loop is the single statement that follows the header. The body here is that empty statement. The loop runs three times doing nothing at all, and the block on lines 8 to 11 merely follows the loop instead of belonging to it.

The second fault, the continue on line 10: continue means abandon the rest of this pass and begin the next pass of the loop enclosing it. After the first fault there is no enclosing loop, so there is no next pass to begin. The language has no meaning to give the statement and the compiler rejects it.

Exactly what the compiler does: it reports error: continue statement not within a loop against line 10, and stops. No object file is written, so there is nothing to link, nothing to load and nothing to run. It also warns first, at line 7, that the for clause does not guard line 8, which is the compiler pointing straight at the semicolon.

The correction: delete the semicolon at the end of line 7, which attaches the block to the loop. That single deletion makes the file compile, and it then prints three lines. The continue is now legal but idle, since nothing follows it in the body, so remove it as well and the result is Program 4.8.

Why it cannot be a logic error: a logic error is defined by what it survives: it compiles, links, loads, runs to a normal exit, and only then turns out to be wrong. This file stops at the second of six stages. An example that never reaches the run cannot illustrate a class of fault whose definition begins with running.

What the pair of faults teaches: a semicolon is the easiest character on a page to overlook, and this one changes the meaning of the four lines beneath it without changing how they look. Indentation is a message to the reader, and the compiler ignores it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int i;
 7      for (i = 0; i < 3; i++)
 8      {
 9          cout << "loop" << endl;
10      }
11      return 0;
12  }
```

Output

```
loop
loop
loop
```

Program 4.8: *Both faults corrected. The semicolon is gone, the block is now the loop body, and the idle continue has been removed.*

Remove the semicolon and the error the compiler complained about disappears with it, which is the usual shape of this fault.

The Master Key: *A stray semicolon after a for header makes the empty statement the loop body and detaches the block below it; the continue is then outside every loop and the file fails to compile, which is precisely what a logic error never does.*

REAL Question 15

A student wrote a + b = c and defended it: the line is true in arithmetic. He was told that a compiler cannot catch a semantic error, and a compiler had just caught this one.

A student writes the statement a + b = c; and is told that it is a semantic error which the compiler will not catch. Explain what the compiler actually does with the statement, and classify the error correctly.

Model Answer

The compiler catches it and refuses to build: it reports error: lvalue required as left operand of assignment and stops, so this is a compile-time error and not a fault that survives as far as the run.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int a = 5;
 7      int b = 3;
 8      int c = 8;
 9      a + b = c;
10      cout << a << endl;
11      return 0;
12  }
```

Compiler says

```
sum.cpp: In function 'int main()':
sum.cpp:9:7: error: lvalue required as left operand of assignment
```

Program 4.9: *The statement as the student wrote it, saved as sum.cpp. The build stops at the compilation stage.*

What assignment requires: the left operand of an assignment must name a place that can be stored into. The name a is such a place and so is b, but a + b is not a place; it is a value that the machine works out and then has nowhere to keep.

Lvalue and rvalue in plain terms: an lvalue names a place, an rvalue names a value. A place has an address and goes on existing; a value is produced, used and gone. Storing into the sum of a and b is not a forbidden instruction so much as an undescribable one, since there is no address to store into.

What the compiler actually does: it rejects Program 4.9 at the compilation stage, before the assembler and long before the linker. There is no assembly, no object file, no executable and nothing to run. The message names the file, the line, the column and the operator at fault.

Classifying it correctly: this is a constraint violation, which in the four-class scheme is a type and constraint error, the second class. The statement is grammatically well formed, so it is not a syntax error, and it is refused on meaning, so it is caught by the compiler and never by the run.

Why the advice given was self-contradictory: calling it a semantic error is an acceptable name, but not alongside a definition that says the compiler cannot catch it. Both claims cannot be held at once, and a candidate holding both will classify by whichever half is nearer to hand.

What the student meant, and where the uncatchable error lives: the intended line is c = a + b, which stores the sum in c: assignment is an instruction with a direction, right into left, and not the equals sign of arithmetic. For a fault the compiler truly cannot catch, write c = a - b where a + b was meant. That one compiles, runs, prints and is wrong.

The compiler decides what can be stored into, and a sum is not somewhere to store.

The Master Key: *The left operand of an assignment must be an lvalue, a named place in memory; a + b is an rvalue, so a + b = c is a constraint violation diagnosed at compile time, not an error that escapes to the run.*

REAL Question 16

Nothing came out of the first two programs at Nyang'hwale, and each showed a different message instead. The third stopped in the middle with the page still blank, and only the fourth printed an answer.

Four short programs are described below. Classify the error in each, and state the stage at which it is caught.

Program A prints one greeting, and the line that does the printing has no semicolon at its end.

Program B declares an int called count and then assigns to it a string variable holding a name.

Program C reads two whole numbers and prints the first divided by the second. The person at the keyboard types 8 and then 0.

Program D adds three marks, divides the total by two, and prints the result as the mean. The marks are 60, 70 and 80.

Model Answer

One of each: A is a syntax error, B a type error, C a run-time error and D a logic error, caught in that order by the compiler, by the compiler, by the running program itself, and by nobody.

Program A, syntax, caught at compilation: the missing semicolon leaves the text ungrammatical, so it is not C++ and no meaning is even looked for. The compiler names the line and stops. Marks are lost here for answering that the program runs and prints nothing; it never runs at all.

Program B, type, caught at compilation: the text is grammatically perfect, and the fault is that no meaning can be given to storing a name in an int. The compiler says it cannot convert a string to an int in an assignment, and the build stops exactly where A stopped.

Program C, run time, caught while running: the build is clean, because nothing in the text is wrong. The fault arrives with the values: dividing by zero is a step the processor cannot carry out, and the process is stopped part way with a non-zero exit status. Type 8 and 4 at the same program and it prints 2 and exits normally.

Program D, logic, caught by nobody: Program 4.10 compiles without a single warning, runs, exits with status zero and prints a mean of 105 for three marks whose mean is 70. Nothing is broken. The program divided by two because it was told to divide by two.

Why the fourth is the one that is argued about: nothing on the screen distinguishes a wrong answer from a right one. The machine has no idea what a mean is, so no tool that remains a compiler could ever catch this. Only a reader who already knows the right answer will find it.

The ladder, read backwards: if nothing was produced, the fault is in the text. If the program stopped part way, the fault is in the values it met. If it finished and the answer is wrong, the fault is in what it was told to do. That is a diagnostic procedure, not a definition to recite.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int first = 60;
 7      int second = 70;
 8      int third = 80;
 9      int total = first + second + third;
10      int mean = total / 2;
11      cout << "Mean mark: " << mean << endl;
12      return 0;
13  }
```

Output

```
Mean mark: 105
```

Program 4.10: *The logic error, run. It builds without a warning, exits with status zero, and prints a mean that is larger than any of the three marks.*

Program	Class of error	Stage at which it is caught	What is seen
A, missing semicolon	Syntax	Compilation	An error and a line number, and no executable
B, string stored in an int	Type or constraint	Compilation	A message about converting a string to an int
C, division by an input of zero	Run time	While the program runs	The process stops part way, exit not zero
D, total divided by two	Logic	No stage catches it	A clean run, a normal exit, and the wrong mean

Table 4.4: *The four programs classified by the stage that catches them. Two never produce a program, one produces a program that stops, and one produces a program that finishes and is wrong.*

The stage that catches a fault is a fact about the fault and not about how serious it is; the one that no stage catches is the dangerous one.

The Master Key: *Syntax and type errors stop the build, a run-time error stops the run, and a logic error stops nothing at all, which is why it survives as far as the printed answer.*

HOT Question 17

A build finished with fourteen warnings and the class cheered, because no line said error. Four days later a customer who had drawn three units of water was billed a surcharge, and nobody had read the seventh warning.

Evaluate the claim that compiler warnings can be ignored because the program compiled. Reach a judgement.

Model Answer

The claim rests on a false premise: compiling is a test of legality and not of correctness, so the fact that the build finished is precisely what the warning already assumed and never what it was disputing.

What is true in the claim: a warning does not stop a build, and not every warning is a defect. A parameter a function never uses is often deliberate, and a project carrying a thousand warnings is one where nobody reads any of them.

What -Wall and -Wextra actually report: not style and not opinion. They turn on the diagnostics for constructs that are legal and almost always mistakes: a variable used before it is given a value, a function reaching its end without returning one, a comparison between a signed and an unsigned value, a line indented as though an if guarded it when it does not.

The class of defect they catch: exactly the class the four-error scheme says nothing catches. Program 4.11 compiles, runs, exits normally and charges a surcharge to a customer who drew three units. No error message is possible, because the text is legal, and -Wall names the line anyway. That is a compiler catching a logic error.

The warning that came before the error: the misleading indentation warning fired in Program 4.7 as well, at the stray semicolon, before the error further down was reached. A warning is frequently the earliest and plainest statement of what has gone wrong, even when a hard error follows it.

The cost of noise, taken seriously: warnings are useful only while they are rare, because attention is the resource being spent. The discipline that works is to hold the count at zero, so that a new warning is an event. Where one is genuinely unwanted, remove its cause rather than the warning.

The condition that decides it: a warning may be ignored when you can state why the compiler is mistaken about that line. The condition is met rarely, and stating it takes about as long as fixing the code. It is never met by the observation that the program compiled.

The verdict: build with -Wall -Wextra, treat the warning count as a defect count, and clear it to zero or justify each survivor in writing. The claim is rejected: it reasons from legality to correctness, and a warning exists because those two are not the same thing.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int units = 3;
 7      if (units > 20)
 8          cout << "Large user rate" << endl;
 9          cout << "Surcharge added" << endl;
10      return 0;
11  }
```

Output

```
Surcharge added
```

Program 4.11: *Legal, well defined, and wrong. Only the second line of output should appear for a large user, and it appears for everybody.*

The compiler has already told you that the program is legal; a warning is the compiler telling you that legal was not the question.

The Master Key: *A warning reports a construct that is legal and probably not what was meant, which is the one class of fault a clean run will not reveal; keep the count at zero so that a new warning still means something.*

HOT Question 18

A stock program at an Mbogwe pharmacy had printed correct totals for a year. The morning after the computer was replaced it printed a total larger than the whole stock, from the same unedited file.

Evaluate the claim that a program which runs and produces output is a correct program. Reach a judgement, and say what would have to be true for running to be evidence of correctness.

Model Answer

The claim is false: a successful run is evidence that the program is legal and that it did not fail on the one input it was given, while correctness is a claim about every input the program will ever meet.

What the claim gets right: a run is real evidence and worth having. It rules out three of the four classes of error for that one path: the program compiled, so it holds no syntax or type error, and it ended normally, so it met no run-time error on the values given.

The fourth class defeats it outright: Program 4.10 compiles, runs, exits with status zero and prints a mean of 105 for marks whose mean is 70. Every observable sign of health is present, and the wrongness is visible only to somebody who already holds the right answer.

Undefined behaviour that happens to work today: a program that writes one element past the end of an array, or reads a variable before setting it, has no defined behaviour. It may print the expected answer for a year and print something else once the compiler is upgraded. It did not become wrong; it was always wrong, and every run was silent about that.

The input that was never tried: one run exercises one path with one set of values. A program that divides by a number typed at it is faultless until somebody types zero, and a loop that is right for ten items may be wrong for one or for none.

What would have to be true: for running to establish correctness you would need every path exercised with every class of input, every construct in the program to have defined behaviour, and somebody who independently knows the right answer in each case. The third condition is the one that is forgotten.

What to do instead: write the expected answer down before running, choose inputs at the boundaries as well as in the middle, the zero, the empty, the largest and the negative, and build with the warnings turned on.

The verdict: a program that runs and prints output is a program that has passed one test. Correctness is a property of the program over all its inputs, so a run supports the claim and cannot establish it. Treat a clean run as the beginning of checking, never the end of it.

The machine can tell you that it finished; whether it was right is a question the machine was never asked.

The Master Key: *Running proves legality and survival on one input, while correctness is a claim about all of them, so a clean run is evidence and never proof, and the fault it hides best is the plausible wrong answer.*

The anatomy of a C++ program, and console input and output

This section takes the smallest working C++ program apart line by line and then puts the console back together. It accounts for every line of that program, settles what a namespace is and what the using directive at the top of every listing in this book actually does, and works through the two console streams: how cin takes values in, how cout sends them out, and why a full name read with the extraction operator arrives with only its first word. It closes by building two programs whose output has to be laid out properly, and by weighing two habits that a great many programs are written with and few of them can defend.

BINDER Question 19

The same seven lines go onto the blackboard at Bukombe every February, copied down without a word about the first two. Ask what the include directive is for and the answer comes back: it is always there.

Explain the purpose of every line of the minimal C++ program shown in Figure 4.4.

Model Answer

Each of the seven lines does a separate job: one brings in the stream facilities, one shortens the standard library names, one names the function the run begins at, two bound its body, one does the visible work, and one reports the outcome.

Line 1, the include directive: #include <iostream> asks the preprocessor to fetch the declarations of the console stream facilities and put them here. Without it the compiler has never heard of cout. The angle brackets say the file belongs to the implementation and not to you.

Line 2, the using directive: using namespace std; makes the names the standard library declares usable without their std qualification, so cout may be written where std::cout is meant. It is a convenience and not a requirement, as Question 20 sets out.

Line 3, int main, and the braces on lines 4 and 7: int main() names the function the run begins at, and a program has exactly one of it. The int is the type of the value main hands back. The braces open and close its body; neither is a statement and neither takes a semicolon.

Line 5, the statement: the only one of the seven that puts anything on the screen. It sends the characters Karibu to the output stream and then ends the line with endl. The semicolon terminates the statement, not the line break.

Line 6, return 0: the value main hands back to the operating system, its exit status, where zero means the run succeeded and anything else means it did not. It prints nothing, and the program never reads it; whatever launched the program can test it.

the program	*what each line is for*
`1 #include <iostream>`	— brings in cout and cin
`2 using namespace std;`	— so cout, not std::cout
`3 int main()`	— the run starts here
`4 {`	— the body of main opens
`5  cout << "Karibu" << endl;`	— text out, then a new line
`6  return 0;`	— 0 reports success
`7 }`	— the body ends, main returns

Figure 4.4: *A complete program with every line accounted for, including the three that are most often used without being explained.*

Main is the one function permitted to fall off its end, and zero is handed back anyway, but write the return and let a reader see the intention.

The Master Key: *Seven lines, seven jobs: the include brings in the streams, the using directive shortens the names, int main says where the run begins, the braces bound the body, the statement does the work, and return 0 reports success.*

BINDER Question 20

Two girls in one Form Five class at Misungwi are both called Neema Joseph. The register now reads Neema Joseph M and Neema Joseph S, and nobody has been marked absent by mistake since.

Explain what a namespace is and what the directive using namespace std; does. State when the qualified form std::cout is to be preferred instead.

Model Answer

A namespace is a named region that the names declared inside it belong to, so that two libraries may each declare something called count and neither of them has to give the word up.

What a namespace is: a named scope. The full name of anything declared inside one is the namespace, then two colons, then the name, as in std::cout. Those two colons are the scope resolution operator, and a name written that way can never be mistaken for another.

Why the standard library has one: everything the standard library declares lives in the namespace std. Many of those names are ordinary words: count, distance, left, swap, data. Without a namespace to hold them, including a header could break a program that had already used one of those words for itself.

What the using directive does: it tells the compiler that an unqualified name it cannot otherwise find may also be looked for in std. It copies nothing, adds no code and costs nothing while the program runs. It changes where names are looked up, and it changes nothing else.

When the qualified form is preferred: in a header, always, because a using directive written there is forced on every file that includes the header and cannot be switched off by the author of that file. In a program of several source files, because a collision then surfaces in code somebody else wrote.

The house choice: listings in this book use the using directive and a candidate should too, because a practical answer is one short file, written once, under a clock. Know why the other form exists, and reach for it as soon as a program outgrows a single file.

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Karibu Misungwi" << std::endl;
6      return 0;
7  }
```

Output

```
Karibu Misungwi
```

Program 4.12: *The same greeting with no using directive. Every standard name is written out in full, and the program is longer to read and impossible to misread.*

The directive is a convenience of lookup, not a part of the language anybody is obliged to use.

The Master Key: *A namespace is a named region owning the names inside it; using namespace std; lets those names be written unqualified, and the qualified form is preferred in a header and in any program of more than one file.*

BINDER Question 21

An answer came back last year with cin << marks written twice, and the student could not see it even when it was pointed at. Both symbols look like arrows.

Explain what cin and cout are, what the extraction and insertion operators do, what endl does, and why both operators can be chained.

Model Answer

cout and cin are objects rather than keywords: cout is the standard output stream, attached to the screen, cin is the standard input stream, attached to the keyboard, and both are declared in <iostream> inside the namespace std.

Insertion, out of the program: the left shift symbols take the value on their right and insert it into the stream on their left, so cout << total sends the characters of total to the screen. In Figure 4.5 the arrows on that side of the program point away from it.

Extraction, into the program: the right shift symbols take characters from the stream on their left and extract a value into the variable on their right, converting the characters to that variable's type. In Figure 4.5 those arrows point into the program, and they never point both ways.

The direction is in the symbols: each pair points the way the data travels, which is the whole reason cin << marks is wrong. They are the same symbols that shift bits when both operands are integers, given a second meaning when the left operand is a stream.

What endl does: it ends the line and flushes the stream, so that what has been written is pushed out to the screen instead of waiting in a buffer. A newline alone can be written as a backslash and an n, which ends the line without flushing.

Why chaining works: an insertion returns the stream itself, so cout << a << b means the same as putting a into cout and then putting b into the stream that came back. Extraction returns the stream in the same way, which is why cin >> a >> b reads two values.

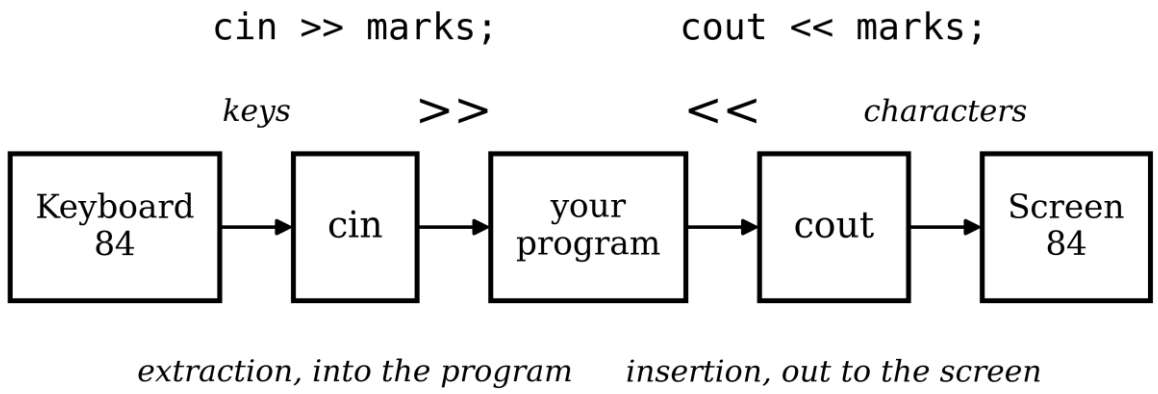


Figure 4.5: *The two console streams. Extraction with the right shift symbols moves data into the program, insertion with the left shift symbols moves it out, and the arrows never run both ways.*

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int books;
 7      int pens;
 8      cout << "Books and pens: ";
 9      cin >> books >> pens;
10      cout << "You have " << books + pens << " items" << endl;
11      return 0;
12  }
```

Typed at the program

```
12 5
```

Output

```
Books and pens: You have 17 items
```

Program 4.13: *One chained extraction and one chained insertion. The input typed at it was 12, a space, and 5. The prompt and the answer share a line because the prompt does not end one.*

Read each operator as an arrow and the statement itself tells you which way the data is going.

The Master Key: *cout and cin are stream objects; insertion sends data towards the screen, extraction pulls it in from the keyboard, endl ends a line and flushes it, and both operators hand the stream back, which is why they chain.*

REAL Question 22

The program asks for the first mark, then the second, then the grade, on three separate lines. A student typed all three answers on one line, pressed Enter once, and every value arrived where it belonged.

Trace exactly what Program 4.14 prints for the input given below, and account for every character of that input.

The input is two lines.

The first line is two spaces, then 74, then three spaces, then 58.

The second line is three spaces, then the letter B.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int first;
 7      int second;
 8      char grade;
 9      cout << "First mark: ";
10      cin >> first;
11      cout << "Second mark: ";
12      cin >> second;
13      cout << "Grade: ";
14      cin >> grade;
15      cout << "first = " << first << endl;
16      cout << "second = " << second << endl;
17      cout << "grade = " << grade << endl;
18      return 0;
19  }
```

Typed at the program

```
  74   58
   B
```

Output

```
First mark: Second mark: Grade: first = 74
second = 58
grade = B
```

Program 4.14: *Two integers and a character, read one at a time and printed back with labels. The prompts ask for three separate entries.*

Model Answer

It prints the three prompts run together on one line, then first = 74, second = 58 and grade = B on three lines of their own, because no prompt ends a line and the layout of the input never mattered.

The first extraction: cin >> first skips the two leading spaces, takes the characters 7 and 4, and stops at the space after them, because a space cannot belong to an int. It converts what it took to 74 and stores it. The space it stopped at is still waiting.

The second and the third: cin >> second skips the three spaces still waiting, takes 58 and stops at the newline. cin >> grade skips that newline and the three spaces after it, then takes exactly one character, B, because one character is the whole of a char.

Why the three prompts run together: not one of the prompt statements ends with endl, so the output stream stays on the same line, and cin itself prints nothing whatever. At a terminal the digits that appear between the prompts are echoed by the terminal, not written by the program.

The layout of the input did not matter: three prompts on three lines suggest three lines of input, and the program never asked for that. Any arrangement of spaces, tabs and newlines that keeps the three values apart is read the same way and gives this same output. A prompt is a courtesy to the reader and places no requirement at all on what is typed.

What is left at the end: one newline, the one typed after B, is still in the buffer when the program finishes. Nothing here reads it, so nothing goes wrong. Question 23 shows what happens in a program where something does read it.

Statement	What it skips first	What it stores	What is left waiting
cin >> first	Two spaces	74	Three spaces, 58, a newline, three spaces, B
cin >> second	Three spaces	58	A newline, three spaces, B
cin >> grade	A newline and three spaces	B	A newline

Table 4.5: *The three extractions traced. Each one skips whitespace, takes what suits its type, and leaves the character it stopped at exactly where it was.*

Trace an extraction by asking three questions in this order: what does it skip, what does it take, and what does it leave behind.

The Master Key: *cin >> skips any leading whitespace, converts as many characters as suit the type, stops at the first character that does not, and leaves that character in the buffer for whatever reads next.*

REAL Question 23

The registration slip printed at Buchosa on the first morning of term said, in full: Welcome, Asha. Her name is Asha Mwanjelwa, and it was typed into the program correctly.

Program 4.15 asks for a full name and prints only the first word of it. Explain why that happens, then correct the program.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      string name;
 8      cout << "Full name: ";
 9      cin >> name;
10      cout << "Welcome, " << name << endl;
11      return 0;
12  }
```

Typed at the program

```
Asha Mwanjelwa
```

Output

```
Full name: Welcome, Asha
```

Program 4.15: *A full name read with an extraction. The line typed at it was Asha Mwanjelwa.*

Model Answer

The extraction operator ends a value at the first whitespace character, so it took Asha and left the space and Mwanjelwa in the buffer, and the correction is getline, which reads to the end of the line instead.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      string name;
 8      cout << "Full name: ";
 9      getline(cin, name);
10      cout << "Welcome, " << name << endl;
11      return 0;
12  }
```

Typed at the program

```
Asha Mwanjelwa
```

Output

```
Full name: Welcome, Asha Mwanjelwa
```

Program 4.16: *The same line read with getline. Everything before the newline arrives, spaces included, and only line 9 has changed.*

The extraction is not broken: it did exactly what it is defined to do. It skipped any leading whitespace, took characters until it met whitespace, and stopped at the space after Asha. The rest of the line, Mwanjelwa and the newline, is still sitting in the buffer unread.

Why a name is not like a mark: whitespace is the separator the extraction operator works by, so no value containing whitespace can be read with it. A mark, a price and a single letter contain none, which is why they read correctly. A full name contains at least one space.

What getline does: getline(cin, name) reads characters into name until it meets a newline, stores everything before it with the spaces intact, and takes the newline off the buffer without storing it. It is a free function, the stream first and the string second, and it needs the string header.

Mixing the two needs care: an extraction stops at the newline and leaves it. A getline called after it meets that newline at once, stores an empty string and returns, and the program looks as though it skipped its own question. The fault lies in the read before it.

The fix has a name: discard the rest of the line before calling getline. That is cin.ignore, given the largest count a stream can hold and a newline as its two arguments, which throws away everything up to and including the next newline. Program 4.17 has that call in it.

What is not the fix: cin.ignore with no arguments discards exactly one character, which serves only while the newline is the only thing left; type 18 years at a prompt expecting a number and the years survive. An extra getline to soak up the line works too, and explains nothing.

Use the extraction operator for a value that cannot contain a space, getline for one that can, and clear the line between the two.

The Master Key: *cin >> ends a value at whitespace, so it cannot read a full name; getline reads to the end of the line, and after any extraction the newline left behind must be discarded before getline is called.*

BUILD Question 24

Three marks, one name, and a division by three. The class at Kwimba wrote this program four times and it failed the same way each time, with the name coming out empty and every number right.

Write a program that reads a student's full name and three marks, then prints the name, the marks, the total and the average to one decimal place.

Read a student's full name, then three marks out of 100.

Print the name, the three marks, the total, and the average to one decimal place.

Model Answer

The program reads the name with getline, the three marks with one chained extraction, adds them, divides by 3.0 so the division is not done in whole numbers, and prints the average under fixed with setprecision(1).

```
 1  #include <iostream>
 2  #include <iomanip>
 3  #include <limits>
 4  #include <string>
 5  using namespace std;
 6
 7  int main()
 8  {
 9      string name;
10      int first = 0;
11      int second = 0;
12      int third = 0;
13      cout << "Full name: ";
14      getline(cin, name);
15      cout << "Three marks: ";
16      cin >> first >> second >> third;
17      cin.ignore(numeric_limits<streamsize>::max(), '\n');
18      int total = first + second + third;
19      double average = total / 3.0;
20      cout << "Student: " << name << endl;
21      cout << "Marks: " << first << " " << second
22           << " " << third << endl;
23      cout << "Total: " << total << " out of 300" << endl;
24      cout << "Average: " << fixed << setprecision(1)
25           << average << endl;
26      return 0;
27  }
```

Typed at the program

```
Asha Mwanjelwa
72 65 80
```

Output

```
Full name: Three marks: Student: Asha Mwanjelwa
Marks: 72 65 80
Total: 217 out of 300
Average: 72.3
```

Program 4.17: *The specification satisfied. The input was the line Asha Mwanjelwa, then the line 72 65 80.*

What the specification asked for: four facts printed: the name as it was typed, the three marks, the total against the 300 that three papers of 100 can carry, and the average to exactly one decimal place. Each has a line of its own in the output beside the listing.

The one decision worth naming: the order of the two kinds of read. getline runs first, so no extraction has yet left a newline for it to trip over. Had the marks been read first, getline would have taken the newline they left and stored an empty name, which is the failure the class kept meeting.

One decimal place is an instruction about printing: fixed with setprecision(1) changes how the value is written and never what it holds. The average stored is 72.333 recurring and any further calculation would use all of it. Line 17 then clears the newline the marks left, so the stream is at the start of a line.

Read the line first and the numbers second, and the two styles of input never collide.

The Master Key: *getline before extraction, never after it without clearing the line, and setprecision fixes how a number is printed rather than what it holds.*

BUILD Question 25

A receipt from a stationery shop had two of its three prices sitting under the quantities instead of under each other. Every figure on it was correct, and the shopkeeper went back to a carbon book.

Write a program that reads three items and prints a receipt as a neat table with column headings and a total.

Read the name, the unit price and the quantity of each of three items.

Print a receipt as a neat table with column headings, one line for each item, and a total.

Model Answer

Five manipulators make the table: setw and setprecision come from the iomanip header, left, right and fixed arrive with iostream, and between them they fix the width of every field, its alignment, and the number of decimals shown.

```
 1  #include <iostream>
 2  #include <iomanip>
 3  #include <string>
 4  using namespace std;
 5
 6  int main()
 7  {
 8      string name;
 9      double price = 0.0;
10      int quantity = 0;
11      double total = 0.0;
12      cout << "Name, price and quantity for three items:" << endl;
13      cout << fixed << setprecision(2);
14      cout << left << setw(12) << "Item" << right
15           << setw(9) << "Price" << setw(5) << "Qty"
16           << setw(10) << "Amount" << endl;
17      for (int i = 0; i < 3; i++)
18      {
19          cin >> name >> price >> quantity;
20          double amount = price * quantity;
21          total = total + amount;
22          cout << left << setw(12) << name << right
23               << setw(9) << price << setw(5) << quantity
24               << setw(10) << amount << endl;
25      }
26      cout << left << setw(12) << "Total" << right
27           << setw(24) << total << endl;
28      return 0;
29  }
```

Typed at the program

```
Daftari 1200.00 4
Kalamu 500.00 10
Rula 800.00 2
```

Output

```
Name, price and quantity for three items:
Item            Price  Qty    Amount
Daftari       1200.00    4   4800.00
Kalamu         500.00   10   5000.00
Rula           800.00    2   1600.00
Total                     11400.00
```

Program 4.18: *The receipt. The three lines typed at it were Daftari 1200.00 4, then Kalamu 500.00 10, then Rula 800.00 2.*

What the specification asked for: headings, a line for each item and a total, all of them aligned. Alignment is the whole of the requirement, since a receipt whose figures do not sit under each other is a list. The four columns are 12, 9, 5 and 10 characters wide.

Sticky and one shot: setw sets the width of the very next item written and is then forgotten, which is why it appears again on every field. fixed, setprecision, left and right stay in force until something changes them, which is why they are set once. Confusing the two is what ragged tables are made of.

The decision worth naming: the item name is read with an extraction, so it has to be a single word. Daftari and Kalamu are fine; exercise book would be split and only the first half would arrive. Reading it with getline instead would put the name on a line of its own.

Manipulator	What it does	How long it lasts
setw(n)	Sets the width of the field written next	That one item only
setprecision(n)	Under fixed, sets the digits after the point	Until changed
fixed	Writes a double in plain decimal, not scientific	Until changed
left	Pads on the right, so the field starts at the left	Until changed
right	Pads on the left, so the field ends at the right	Until changed

Table 4.6: *The five manipulators used in Program 4.18. Only setw is forgotten after one item, which is why it is written again for every field. setw and setprecision need the iomanip header; the other three arrive with iostream.*

Give every field a width and the receipt aligns itself; the total is right aligned across the 24 characters the last three columns occupy, so it lands under the amounts.

The Master Key: *setw sets the width of the next item only, while fixed, setprecision, left and right stay in force until they are changed.*

HOT Question 26

Nearly every program in this chapter begins with the same two lines, and the second is a convenience rather than a necessity. Programmers with twenty years behind them disagree in public about whether to write it at all.

Evaluate the practice of writing using namespace std; at the top of every program. Reach a judgement and state the condition it turns on.

Model Answer

The directive is defensible in a single source file that one person writes, reads once and hands in, and indefensible in a header or in a program of several files, and what decides it is whose code will be compiled together with yours.

What it buys: brevity where brevity is read. Writing std::cout, std::fixed and std::setprecision is three qualifications inside a single statement, and a candidate writing by hand in an examination room has no autocomplete and a fixed number of minutes.

What it actually costs: it does not import a short list of names. It makes every name in std a candidate for every unqualified lookup below it, including the names a later version of the library will add. Code that compiled last year can stop compiling when the library grows.

The collision is not hypothetical: Program 4.19 declares a global called count and includes the algorithm header, which declares std::count. The compiler cannot tell which of the two the program means and refuses the line. The complaint arrives at the use, not at the directive that caused it.

Where it is always wrong: in a header. A using directive written in a header is inherited by every file that includes that header, and the authors of those files never asked for it and cannot switch it off. That one is not a matter of taste.

What changes as a program grows: with one file and one author, a collision is yours and you meet it at once. With ten files and three libraries it surfaces in code you did not write, in a build you did not run, over a name you have never used.

The middle course: a using declaration names only what is wanted, using std::cout; and using std::string;, and leaves everything else qualified. Written inside a function rather than at file scope, even the full directive is confined to that function and expires with it.

The judgement: write it at the top of a single file program that will be handed in and never maintained, which is exactly what a practical answer is; never write it in a header; and qualify from the second source file onwards. The condition is the reach of the program, not the taste of the programmer.

```
 1  #include <iostream>
 2  #include <algorithm>
 3  using namespace std;
 4
 5  int count = 0;
 6
 7  int main()
 8  {
 9      count = count + 1;
10      cout << count << endl;
11      return 0;
12  }
```

Compiler says

```
p.cpp:9:5: error: reference to ‘count’ is ambiguous
    9 |     count = count + 1;
      |     ^~~~~
note: candidates are: ... std::count(_IIter, _IIter, const _Tp&)
p.cpp:5:5: note:                 ‘int count’
```

Program 4.19: *A global called count, a using directive, and a standard library that also declares count. The program is rejected, and the message, abridged here, names both candidates.*

A convenience that reaches no further than your own file is a convenience; the same line in a header is a decision taken on somebody else’s behalf.

The Master Key: *using namespace std; changes name lookup for everything below it, which is harmless in one short file, wrong in a header, and a liability once a program has several files, so qualify from the second file onwards.*

HOT Question 27

A console window opens, black, prints six lines and vanishes before anybody can read one. The cure passed around that laboratory was a single line pasted above the closing brace, and it worked on every machine there.

A program ends with a call to system whose argument is the word PAUSE, put there so that the console window stays open long enough to read the output. Evaluate this practice and reach a judgement.

Model Answer

The practice should be abandoned: it ties the program to one operating system, it starts an entire command interpreter in order to do nothing, and on a machine whose shell has no such command it prints an error and does not pause at all.

What the call actually does: system hands a string to the host command processor and waits for it to finish. On Windows that processor has a built in PAUSE which prints a message and waits for a key. None of that is C++: the standard says only that the string is handed to the environment, and that the effect is implementation defined.

What it does on a machine that has no such command: on a machine running Linux a shell is started, it looks for a command called PAUSE, does not find it, writes a not found message to the error stream and reports failure. The window does not pause, and the program ends immediately afterwards.

The cost when it does work: starting a shell means creating a whole new process, loading it and waiting for it, all so the program can wait for one key. It is the most expensive way the language offers of doing nothing.

Where the habit comes from: running the program by double clicking it, so that the console window belongs to the program and is destroyed the moment it exits. The output does not vanish because of a fault in the program. It vanishes because of the way the program was started.

The alternatives, best first: start the program from a command prompt that is already open, so the window belongs to the shell and outlives the program. Failing that, use whatever setting the development environment offers for holding its console open. Both leave the program itself unaltered and portable.

If a program truly must wait: wait on input. Program 4.20 prints a prompt and calls cin.get(), which is standard C++ and behaves the same everywhere. It carries one honest cost: after an extraction it takes the newline left behind and does not wait, which is the newline of Question 23 again.

The judgement: do not write it. It is not portable, it is not free, and it answers a question about how the program was launched as though it were a question about the program. Where a wait is genuinely wanted, wait on input and say so in a prompt.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      cout << "Total: 11400" << endl;
 7      cout << "Press Enter to finish." << endl;
 8      cin.get();
 9      return 0;
10  }
```

Typed at the program

Output

```
Total: 11400
Press Enter to finish.
```

Program 4.20: *The portable wait. It prompts, then waits for a line, and the same source behaves identically on every system that has a console.*

A program made to behave differently on another machine for no gain has been made worse, and a window that closes too quickly was never the fault of the program.

The Master Key: *A PAUSE handed to system is a Windows shell command borrowed into C++: it starts a whole process, does nothing elsewhere, and the real answer is to run the program from a console that stays open by itself.*

Data types, variables, constants, scope and lifetime

This section settles what a value is made of, and where the name given to it may be used. It classifies the types, shows why the size of a built-in type is measured rather than memorised, separates declaration from definition from initialisation, gives the real rules for naming, and keeps scope, lifetime and linkage apart as the three different properties they are. Two of its programs interrogate this machine directly, one printing every built-in type beside its width and its largest value, the other holding a count inside a function across five calls without ever letting the name out.

BINDER Question 28

A chart taped above the machines in the laboratory at Magu gives int 4, long 4 and long double 8. Two of those numbers are wrong on the machines sitting underneath it.

Classify the C++ data types, and explain why the size of a built-in type is not fixed by the language.

Model Answer

C++ types fall into three groups, built-in, derived and user defined, and the size of a built-in type is chosen by the implementation rather than by the language, so it must be measured with sizeof.

The three groups: the built-in types arrive with the language: the integral types char, short, int, long and long long, the boolean bool, the floating types float, double and long double, and void. Derived types are assembled from those, as arrays, pointers and functions are. User defined types are the ones you write, chiefly the class, the struct and the enumeration.

What the language actually fixes: three things, and no more. Every implementation must meet the minimum ranges in the middle column of Table 4.7, sizeof(char) is exactly 1 by definition, and the widths must never decrease along char, short, int, long, long long. Not one item on that list names a byte count.

Why the sizes are left open: so that one language can serve a card reader with a small processor and a server with a large one. An implementation picks the width its own hardware handles fastest, which is why C++ compiles into efficient code on machines with nothing else in common.

What that does to the familiar chart: on this machine a long measures 8 bytes and a long double 16, so a chart giving them as 4 and 8 is wrong here and correct elsewhere. Figure 4.6 draws the measured widths to scale, and the bracket at its left marks the one ordering the language guarantees.

How to answer this honestly: quote the guarantee, then measure. Print sizeof(int) as Program 4.25 does, and the answer is true of the machine in front of you. A size written from memory is true of nothing in particular.

Type	Guaranteed by the language	Measured here with sizeof
bool	Holds true or false	1 byte
char	At least 8 bits, and every basic character fits	1 byte, by definition
short	At least -32 767 to 32 767	2 bytes
int	At least -32 767 to 32 767	4 bytes
long	At least -2 147 483 647 to 2 147 483 647	8 bytes
long long	At least -9 223 372 036 854 775 807 to 9 223 372 036 854 775 807	8 bytes
float	At least 6 decimal digits of precision	4 bytes
double	At least 10 decimal digits of precision	8 bytes
long double	At least as precise as double	16 bytes

Table 4.7: *The middle and right columns say different kinds of thing. The middle one holds on every machine and may be quoted from memory; the right one was measured by Program 4.25 on the machine that built this book, and may be quoted only of that machine.*

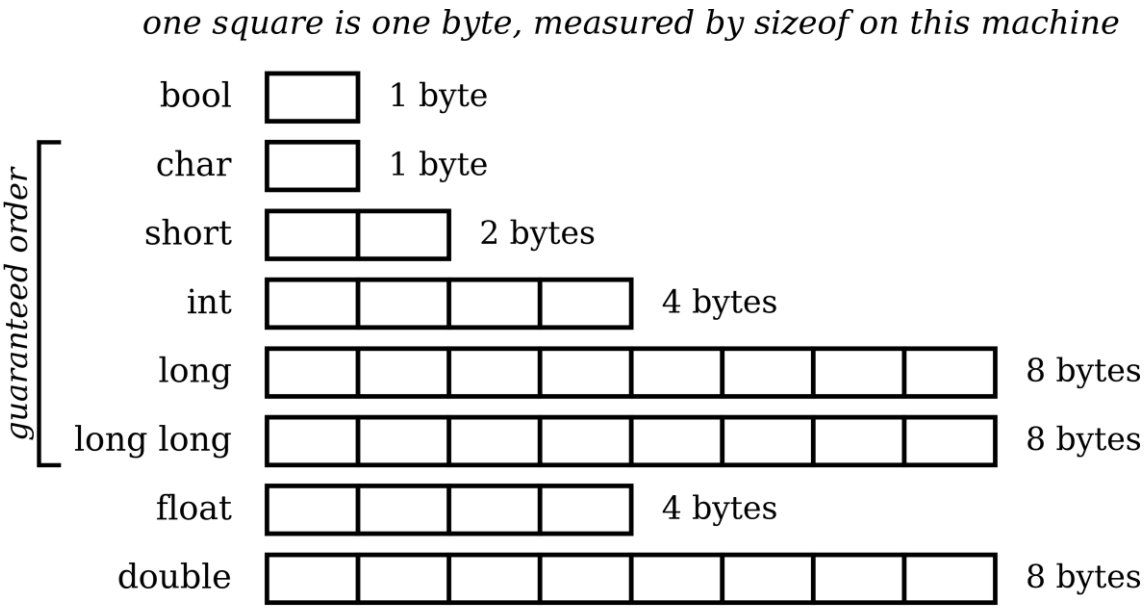


Figure 4.6: *The built-in types drawn to scale, with every width measured on the machine that built this book rather than assumed. The ordering is guaranteed by the language; the exact numbers are not.*

Take the guarantee from the language and the number from sizeof, and never the other way round.

The Master Key: *The language fixes minimum ranges, sizeof(char) as 1, and the non decreasing order char, short, int, long, long long; every actual size is implementation defined and is found by measuring it.*

BINDER Question 29

The same short program was run twice in one afternoon, printing 0 and then a five figure number. The student concluded that the machine was faulty.

Distinguish declaration, definition and initialisation, and state what an uninitialised variable holds.

Model Answer

A declaration introduces a name and its type, a definition is the declaration that also reserves the storage, and initialisation is the giving of a value at the moment that storage is created.

Declaration: introduces a name and tells the compiler what type it has, so that later lines mentioning it can be checked. Line 4 of Program 4.21 is a pure declaration: the word extern promises that bagsWeighed exists somewhere and reserves no memory for it.

Definition: the declaration that also sets the memory aside. Line 5 defines bagsWeighed. Almost every declaration a candidate writes is a definition as well, which is why the two words get swapped, but a program may declare a name many times and must define it exactly once.

Initialisation: supplying a value as the object is created, as line 10 does. Assignment is a different act: it replaces a value the object already holds, which is what line 12 does to rejected. An object is initialised at most once and may be assigned as often as you like.

What an uninitialised local holds: an indeterminate value, and reading it before anything has been written to it is undefined behaviour. Such a program may print anything at all, may print something different on the next run, and is under no obligation to keep behaving sensibly afterwards.

The one guaranteed case, and the false test: an object defined at file scope, or declared static, is zero initialised before the program starts, which is why line 9 prints 0. A local is promised nothing, and a run in which one happens to print 0 proves only that those bytes happened to hold 0 that afternoon.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  extern int bagsWeighed;
 5  int bagsWeighed;
 6
 7  int main()
 8  {
 9      cout << "bagsWeighed starts at " << bagsWeighed << endl;
10      int accepted = 45;
11      int rejected;
12      rejected = 3;
13      bagsWeighed = accepted + rejected;
14      cout << "bagsWeighed is now " << bagsWeighed << endl;
15      return 0;
16  }
```

Output

```
bagsWeighed starts at 0
bagsWeighed is now 48
```

Program 4.21: *A declaration that is not a definition on line 4, a definition on line 5, an initialisation on line 10 and an assignment on line 12. The zero on the first output line is guaranteed only because bagsWeighed sits at file scope.*

Say declared when only the name arrives, defined when the storage arrives, and initialised when the first value arrives with it.

The Master Key: *A declaration names, a definition reserves, initialisation supplies the first value; objects at file scope and static objects are zero initialised, locals are not, and reading one before writing to it is undefined behaviour.*

BINDER Question 30

A red line runs through every variable name in an exercise book at Rorya, each one clear and each one legal. The comment beside them reads: too long, eight characters maximum.

State the rules for a valid C++ identifier, and explain which commonly taught rule is not a rule at all.

Model Answer

An identifier may be made of letters, digits and the underscore, must not begin with a digit, must not be a keyword, and is case sensitive; the language sets no limit whatever on its length.

What a name may be made of: the letters A to Z and a to z, the digits 0 to 9, and the underscore. Nothing else, so a space, a hyphen and a full stop are all excluded. The first character must be a letter or an underscore, never a digit.

Keywords and case: the words the language reserves for itself, among them int, long, class and return, cannot be reused as names. Case matters everywhere, so total, Total and TOTAL are three separate variables that the compiler will never connect.

The names you may form but should not: two shapes are reserved to the implementation and its headers. A name beginning with an underscore followed by a capital, such as _Total, and any name containing a double underscore, such as bag__count, are reserved everywhere in a program. They compile, and then collide with something you cannot see.

The rule that does not exist: there is no eight character limit in C++ and there never has been. The restriction came from an early linker that kept only the first few characters of an external name. The standard sets no maximum and asks implementations to accept identifiers of at least 1024 characters.

What that permits: Program 4.24 defines a variable whose name runs to 46 characters, and it compiles without a murmur. A long name costs only the width of the line it sits on, and it buys a reader who need not guess what the value means.

Name as written	Verdict	Why
totalMarksAwarded	Valid	Letters only, and it begins with a letter
mark_2	Valid	Digits and underscores are allowed after the first character
2ndWeekDelivery	Invalid	It begins with a digit
total marks	Invalid	A space is not one of the permitted characters
long	Invalid	It is a keyword of the language
_Total	Reserved	An underscore then a capital is reserved to the implementation
bag__count	Reserved	It contains a double underscore
numberOfBagsDeliveredThisWeek	Valid	It is 29 characters long, and length has never been a rule

Table 4.8: *Eight names tested against the rules. Only the first character, the character set and the keyword list can make a name invalid; the two marked reserved compile and should still be avoided.*

Judge a name by whether it says what the thing is, because the language stops caring after the first character.

The Master Key: *Letters, digits and underscores, never beginning with a digit, never a keyword, always case sensitive, with the underscore then capital and double underscore forms reserved; the eight character limit is not a rule of C++.*

BINDER Question 31

The water kiosk at Itilima keeps two records of the same water: a tally sheet thrown away each shift, and a meter never reset. Only one can say what the village drew last month.

Explain scope, lifetime and linkage, and distinguish local, global and static variables by all three.

Model Answer

Scope is where in the source text a name is visible, lifetime is when during the run the object exists, and linkage is whether the same name in another file reaches the same object; a variable has one of each, and they are set separately.

Scope is a fact about the text: a name declared inside a block has block scope and cannot be mentioned outside the braces. A name declared outside every function has file scope and is visible from its declaration to the end of that file.

Lifetime is a fact about the run: automatic lifetime begins where the definition is reached and ends at the closing brace of the enclosing block. Static lifetime begins before main is entered and ends only when the program does.

Linkage is a fact about other files: a global has external linkage by default, so a declaration in a second file reaches the same object. A local has no linkage at all. Marking a file scope name static gives it internal linkage, confining it to the one file.

The combination that surprises people: a static local, as on line 6 of Program 4.26, has block scope with static lifetime. Its name is visible only inside the function while its value survives every return, which is why Figure 4.7 draws its bar unbroken across two calls when the automatic local below it vanishes between them.

The error to avoid: static does not mean available to the whole program. That claim muddles lifetime with linkage, and it gets the linkage backwards, since static at file scope removes reach that a global would otherwise have had. What static always changes is lifetime; where it changes linkage, it narrows it.

Kind of variable	Scope	Lifetime	Linkage
Local	The block it is defined in	Until that block closes	None
Static local	The block it is defined in	The whole run	None
Global	To the end of the file	The whole run	External
Global marked static	To the end of the file	The whole run	Internal

Table 4.9: *The same four variables answered against all three properties. Reading down the linkage column shows that static means one thing on a local and something else again on a global.*

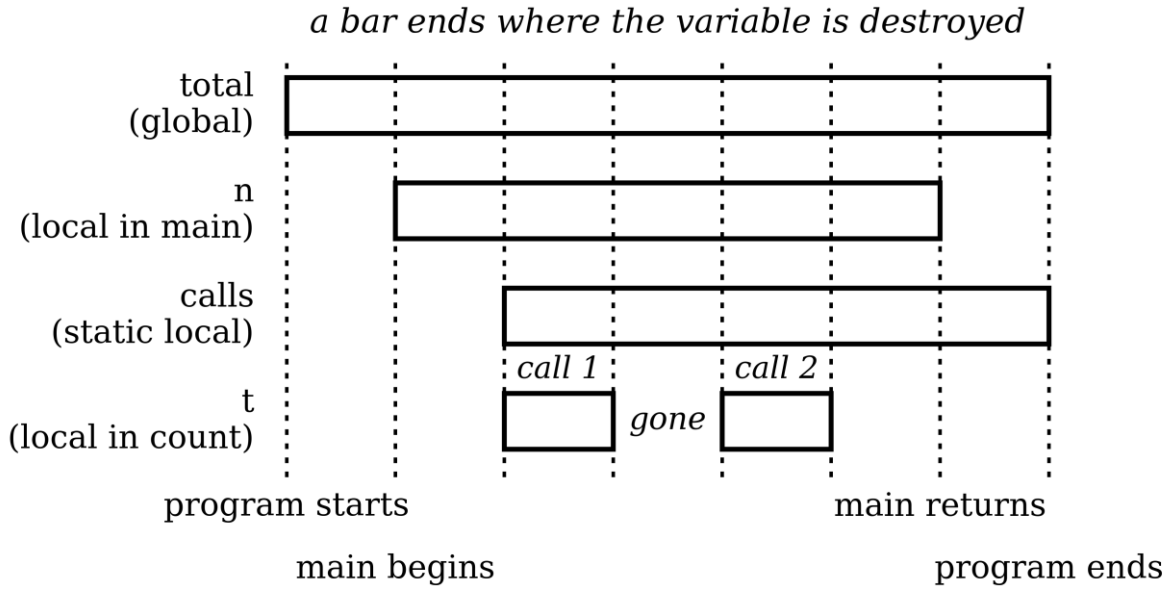


Figure 4.7: *When each kind of variable exists during one run. The static local is the one to look at: its bar is unbroken across two calls, although its name is visible only inside the function.*

Answer all three separately, because one variable may be narrow in one property and wide in another.

The Master Key: *Scope is where a name is visible, lifetime is when the object exists, linkage is whether other files can reach it; a static local is narrow in scope, long in life, and has no linkage at all.*

REAL Question 32

The tally program at the Busega fish landing site worked out for itself how many readings fit in a thousand bytes, and was right for four seasons. Then the office replaced the computer.

Program 4.22 below compiles and runs. Name the fault, state exactly what happens when it runs on this machine, and correct it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      const int BUFFER_BYTES = 1000;
 7      const int ASSUMED_INT_BYTES = 2;
 8      const int SLOTS = BUFFER_BYTES / ASSUMED_INT_BYTES;
 9      int reading[SLOTS];
10      for (int i = 0; i < SLOTS; i++)
11      {
12          reading[i] = 0;
13      }
14      cout << "Assuming an int is " << ASSUMED_INT_BYTES
15           << " bytes," << endl;
16      cout << SLOTS << " readings fit in " << BUFFER_BYTES
17           << " bytes." << endl;
18      cout << "The array really occupies " << sizeof(reading)
19           << " bytes." << endl;
20      return 0;
21  }
```

Output

```
Assuming an int is 2 bytes,
500 readings fit in 1000 bytes.
The array really occupies 2000 bytes.
```

Program 4.22: *A program that writes a type size into itself as a literal. It compiles without a warning and runs to completion.*

Model Answer

The fault is on line 7, which asserts that an int occupies 2 bytes; an int occupies 4 bytes on this machine, so the array sized from that assertion is twice as large as the buffer it was meant to fit inside.

Naming the fault: line 7 records a byte count as though it were a property of the language, and lines 8 and 9 then size an array from it. The size of a built-in type is implementation defined, so a literal byte count is a belief about one machine written permanently into a program that will be compiled on others.

What actually happens here: it compiles with no diagnostic at all and runs to a normal exit, which is the dangerous part. SLOTS becomes 500, the array is declared with 500 elements, and sizeof measures it as 2000 bytes, so the program has quietly reserved twice the thousand bytes its own second line claims are enough.

The contradiction is in its own output: the third line measures what the first line assumed, and they disagree by a factor of two. On a machine where an int really is 2 bytes all three lines would agree, which is exactly what let the error survive four seasons at the landing site without being noticed.

Why the compiler says nothing: nothing here is ill formed. A thousand divided by two is five hundred, and an array of five hundred ints is legal. The mistake lies in what the numbers are taken to mean and not in what they do, and no compiler diagnoses a wrong belief about the hardware.

The correction: replace the assumed constant with a measurement, as line 7 of Program 4.23 does. The expression BUFFER_BYTES divided by sizeof(int) is worked out with the real width, so SLOTS becomes 250

here and the array measures exactly the thousand bytes budgeted. On a 2 byte machine the same line yields 500 and still fits.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      const int BUFFER_BYTES = 1000;
 7      const int SLOTS = BUFFER_BYTES / sizeof(int);
 8      int reading[SLOTS];
 9      for (int i = 0; i < SLOTS; i++)
10      {
11          reading[i] = 0;
12      }
13      cout << "One int measures " << sizeof(int)
14           << " bytes here." << endl;
15      cout << SLOTS << " readings fit in " << BUFFER_BYTES
16           << " bytes." << endl;
17      cout << "The array really occupies " << sizeof(reading)
18           << " bytes." << endl;
19      return 0;
20  }
```

Output

```
One int measures 4 bytes here.
250 readings fit in 1000 bytes.
The array really occupies 1000 bytes.
```

Program 4.23: *The same task with the width measured instead of assumed. One line changed, and the third output line now agrees with the second on this machine and on any other.*

The habit to take away: wherever a program divides by a size, multiplies by a size, or reserves a block of bytes, that size must come from sizeof. Writing the number instead saves a few keystrokes and buys a fault that surfaces years later on somebody else's computer.

The program was never wrong about arithmetic; it was wrong about the machine, and only sizeof can settle that question.

The Master Key: *Never write the size of a type as a literal; ask sizeof at the point of use, and one piece of source stays correct on every machine it is compiled for.*

REAL Question 33

An exercise book came back from marking at Ilemela with every variable name struck through in red. The program compiles, runs, prints the right figures, and hides one name that really would stop the compiler.

Program 4.24 was rejected on the grounds set out below. Explain why the rejection is wrong, state the real rules for a C++ identifier, and identify the one name in the listing that genuinely could not be used.

The note on the work reads: rejected, no identifier may exceed eight characters.

The replacements suggested are bags, price, rej, acc and tot.

Program 4.24 as printed compiles with no warnings and prints the two lines shown.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int numberOfBagsDeliveredThisWeek = 48;
 7      int numberOfBagsRejectedAtWeighing = 3;
 8      long priceInShillingsPerBagOfMaize = 62000;
 9      // still to record: 2ndWeekDelivery and a running mean
10      int acceptedBags = numberOfBagsDeliveredThisWeek
11                       - numberOfBagsRejectedAtWeighing;
12      long totalShillingsPayableForTheAcceptedBagsOfMaize =
13          acceptedBags * priceInShillingsPerBagOfMaize;
14      cout << "Bags accepted: " << acceptedBags << endl;
15      cout << "Shillings payable: "
16           << totalShillingsPayableForTheAcceptedBagsOfMaize
17           << endl;
18      return 0;
19  }
```

Output

```
Bags accepted: 45
Shillings payable: 2790000
```

Program 4.24: *A maize ledger written with names that say what they hold. It is the listing the note was written on.*

Model Answer

The rejection is wrong because C++ sets no limit at all on the length of an identifier; the eight character rule came from an early linker for another language and has never been part of this one.

Why the rejection fails: no version of C++ has ever limited the length of a name. Early linkers kept only the first few characters of an external symbol, and the rule outlived the machines that needed it. The standard sets no maximum and asks implementations to accept at least 1024 characters in a name.

The rules that do apply: letters, digits and the underscore only; a first character that is a letter or an underscore and never a digit; no keyword; and case sensitivity throughout. Program 4.24 satisfies every one of them, which is why it compiles cleanly and prints its two lines.

The name that really would fail: 2ndWeekDelivery, in the comment on line 9. It begins with a digit, so it could never be an identifier in C++, and only the two slashes in front of it keep the listing valid. Its length was never the difficulty; its first character is.

What the suggested replacements cost: bags, price, rej, acc and tot all compile perfectly well. They also stop saying which bags, in what currency, and rejected by whom. Shortening a name does not remove the meaning; it moves the meaning out of the program and into somebody's memory, where it cannot be checked.

Where a genuine limit does live: in reading rather than in compiling. A name long enough to push a statement across two printed lines is harder to follow, which is why the assignment on lines 12 and 13 is broken in two. That is a judgement about clarity and has nothing to do with the language.

What the program computes: 48 bags delivered less 3 rejected leaves 45 accepted, and 45 bags at 62 000 shillings each gives 2 790 000 shillings. Line 8 declares the price a long deliberately, because the total exceeds the range the language guarantees for an int.

The compiler cares about the first character and the list of keywords, and everything else about a name is a decision taken for the reader.

The Master Key: *A C++ identifier is letters, digits and underscores, never starting with a digit and never a keyword, with no maximum length at all; 2ndWeekDelivery fails on its first character, not on its size.*

BUILD Question 34

Four, four and eight. Those three numbers are copied into more exercise books than any others in this subject, and on the machine that compiled this book two of them are wrong.

Write a complete C++ program to the specification below, and state what it reports on this machine.

Print one row for each of char, short, int, long, long long, float, double and long double.

Each row gives the type name, the bytes it occupies on this machine, and the largest value it can hold.

Take every byte count from sizeof and every limit from <climits> and <cfloat>.

Align the three columns with <iomanip> so that the result reads as a table.

Model Answer

Program 4.25 prints one heading and eight rows, and every number in it is measured by the program at the moment it runs rather than written into it beforehand.

Where each number comes from: the middle column is sizeof applied to the type itself, which costs nothing while the program runs because the compiler has already replaced it with a constant. The right hand column is the limit macro the library defines for that type, CHAR_MAX and its relatives from climits, FLT_MAX and DBL_MAX and LDBL_MAX from cfloat.

What it reports here: char 1 byte, short 2, int 4, long 8, long long 8, float 4, double 8 and long double 16. The long and the long double are precisely the two the familiar chart gets wrong, and long and long long come out equal here although the language only requires that the second is not narrower.

The one decision worth naming: the rows are written out one at a time instead of being driven by a loop, because C++ has no container of types to loop over. Repetition is the honest answer here, and it lets each row be read on its own without working out what a loop would have put there.

```
 1  #include <iostream>
 2  #include <iomanip>
 3  #include <climits>
 4  #include <cfloat>
 5  using namespace std;
 6
 7  int main()
 8  {
 9      cout << left << setw(13) << "Type" << right << setw(6)
10           << "Bytes" << setw(22) << "Largest value" << endl;
11      cout << left << setw(13) << "char" << right << setw(6)
12           << sizeof(char) << setw(22) << CHAR_MAX << endl;
13      cout << left << setw(13) << "short" << right << setw(6)
14           << sizeof(short) << setw(22) << SHRT_MAX << endl;
15      cout << left << setw(13) << "int" << right << setw(6)
16           << sizeof(int) << setw(22) << INT_MAX << endl;
17      cout << left << setw(13) << "long" << right << setw(6)
18           << sizeof(long) << setw(22) << LONG_MAX << endl;
19      cout << left << setw(13) << "long long" << right << setw(6)
20           << sizeof(long long) << setw(22) << LLONG_MAX << endl;
21      cout << left << setw(13) << "float" << right << setw(6)
22           << sizeof(float) << setw(22) << FLT_MAX << endl;
23      cout << left << setw(13) << "double" << right << setw(6)
24           << sizeof(double) << setw(22) << DBL_MAX << endl;
25      cout << left << setw(13) << "long double" << right << setw(6)
26           << sizeof(long double) << setw(22) << LDBL_MAX << endl;
27      return 0;
28  }
```

Output

```
Type          Bytes         Largest value
char              1                   127
short             2                 32767
int               4            2147483647
long              8   9223372036854775807
long long         8   9223372036854775807
float             4           3.40282e+38
double            8          1.79769e+308
long double      16         1.18973e+4932
```

Program 4.25: *Every built-in type interrogated on the machine that built this book. Not one number in the output was typed by the author; each was measured or read from a library limit as the program ran.*

Print the measurement and the answer is true of one machine on one day, which is more than any remembered figure can claim.

The Master Key: *sizeof gives the width and climits and cfloat give the limits, and a table built from both is the only honest answer to the question of how large an int is.*

BUILD Question 35

Click. The turnstile counts once for each person who walks through it, and the number standing on its dial at closing time is kept by none of them; it belongs to the gate.

Write a complete C++ program to the specification below, and state what it prints.

Write a function that reports how many times it has been called.

The count must live in a variable whose name is visible only inside that function and whose value survives from one call to the next.

Call the function five times from main, printing the count on each call.

Print the final total from main once the loop has finished.

Model Answer

Program 4.26 meets the specification with one word, static on line 6, which gives visitCount the lifetime of the whole program while leaving its name shut inside recordVisit.

How the count survives: a static local is created once, before the first call, and destroyed only when the program ends. The initialiser on line 6 runs once and never again, so visitCount is not reset on the second entry: it carries 1 into the call that turns it into 2.

The twin that proves it: visitsSinceEntry on line 7 is an ordinary local. It is built and destroyed on every call, so it is initialised to 0 five times and printed as 1 five times. The two columns of output are the same increment applied to variables that differ in nothing but lifetime.

Why the name stays private: static changes the lifetime and not the scope. Main cannot mention visitCount and would fail to compile if it tried, which is the whole difference between this and a global. The count is owned by the one function with any business changing it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int recordVisit()
 5  {
 6      static int visitCount = 0;
 7      int visitsSinceEntry = 0;
 8      visitCount++;
 9      visitsSinceEntry++;
10      cout << "static " << visitCount << "   automatic "
11           << visitsSinceEntry << endl;
12      return visitCount;
13  }
14
15  int main()
16  {
17      int total = 0;
18      for (int i = 0; i < 5; i++)
19      {
20          total = recordVisit();
21      }
22      cout << "Calls in total: " << total << endl;
23      return 0;
24  }
```

Output

```
static 1   automatic 1
static 2   automatic 1
static 3   automatic 1
static 4   automatic 1
static 5   automatic 1
Calls in total: 5
```

Program 4.26: *A static local counting its own calls. The left column climbs from 1 to 5 across five calls; the automatic variable beside it is created fresh each time and never gets past 1.*

A long life and a small scope are not a contradiction, because they are two different properties and static sets only the first.

The Master Key: *A static local is created once and lives until the program ends, but its name never leaves the function, so it can count across calls without becoming a global.*

HOT Question 36

Only one of the two corrections took effect. The price had been named twice, once with const and once as a macro, and the debugger insisted that one of those names did not exist.

Evaluate the use of const against #define for naming a constant. Reach a judgement.

Model Answer

Use const for a named value, and constexpr where the value must be settled while the program is compiled, keeping the macro for the one task nothing else can do, which is conditional compilation; a macro is not part of the language the compiler ever sees.

What each of them actually is: a const definition creates an object with a type, a scope and an address that the compiler knows about. A define instructs the preprocessor to replace one piece of text with another everywhere it appears, before compilation begins. The first is a declaration; the second is an edit to the source.

Type safety: a const constant has a type, so a wrong use is reported against the name on the line using it. A macro has no type of its own and takes whatever the surrounding expression forces on its text, so the diagnostic arrives against a number the programmer never typed.

Scope: a const obeys the ordinary rules: declared inside a block it ends at the closing brace, and it can be a member of a class. A macro ignores blocks, functions, classes and namespaces alike, holding from its own line to the end of the file, and will cheerfully rewrite a name inside somebody else's class.

Substitution is not evaluation: Program 4.27 carries the whole objection in two lines of output. BOOKS_PER_SHELF is defined as the text 3 + 1, so the expression on line 9 becomes 3 + 1 * 2 and prints 5, while the const version on line 10 prints 8. Brackets round the macro body cure this instance and not the disease.

Debugging: the compiler and the debugger see only what the preprocessor left behind. Ask a debugger for the value of a macro and it reports no such name, because the program now holds the number wherever the name once stood. A const keeps its name and can be inspected.

What constexpr adds: constexpr requires that the value be computable while the program is compiled, which is the one genuine advantage a macro ever offered. A constexpr constant can be used wherever the language demands a constant expression, and keeps its type and its scope while doing it.

The judgement: const for a named value, constexpr where a constant expression is required, and define only for conditional compilation, where the decision must be taken before the compiler starts. A macro used merely to name a number gains nothing and gives up the type, the scope and the name.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  #define BOOKS_PER_SHELF 3 + 1
 5  const int SAFE_BOOKS_PER_SHELF = 3 + 1;
 6
 7  int main()
 8  {
 9      cout << "macro gives " << BOOKS_PER_SHELF * 2 << endl;
10      cout << "const gives " << SAFE_BOOKS_PER_SHELF * 2 << endl;
11      return 0;
12  }
```

Output

```
macro gives 5
const gives 8
```

Program 4.27: *The same four written twice. The macro is pasted in as text and the multiplication reaches part of it; the const is a value and the multiplication reaches all of it.*

Prefer the tool the compiler can see, and keep the preprocessor for decisions that must be taken before the compiler starts work.

The Master Key: *const and constexpr give a constant a type, a scope and a name that survives into the compiler; a macro gives text substitution with none of the three, and belongs to conditional compilation.*

HOT Question 37

The stock report printed a negative number of bags, and the hunt took two days. A function nobody had opened for a year had been quietly subtracting from the total.

Evaluate the use of global variables. Reach a judgement with a stated condition.

Model Answer

Use a global only when it is const, or when it is genuinely one item of state the whole program shares and of which there can only ever be one; otherwise pass it as a parameter, because a global turns a local question into a search of the entire program.

The honest case for them: one value wanted by eight functions costs eight parameters and an argument at every call. A global removes all of that, and where the value is genuinely program wide, a setting read once at start up, this is accuracy rather than laziness.

What it costs when a bug is traced: the question of who changed this value has a bounded answer for a local and none for a global. For a local the suspects are the lines of one block; for a global, every function that can see the name, so the search grows with the program rather than with the bug.

What happens as the program grows: names at file scope all compete for one space, so a second programmer adding a global called count collides with yours. External linkage lets that collision reach across files, and objects at file scope in different files are built in no guaranteed order.

Testing and reuse: a function that reads a global cannot be judged by its arguments. Call it twice with identical arguments and it may answer differently, because something elsewhere moved the state in between. Nor can it be lifted into another program without dragging the global behind it.

The exception that is not the problem: a global that cannot be written is not what the warning is about. A const at file scope has no writer, so none of the tracing difficulties apply, and in C++ it has internal linkage by default, so it does not even compete for the shared name space.

What to reach for instead: a parameter and a return value when a value passes between two functions; a static local when the state belongs to one function, as in Program 4.26; and a class holding the data with the functions that act on it when several share it. The last of those is encapsulation.

The judgement: write a global for constants and for a single genuinely program wide item, and everything else as a parameter. The test before adding one is short: if you cannot name every function permitted to change it, you cannot debug it either.

A global is a standing permission for anything to write to it, and it will be used on the day you would least choose.

The Master Key: *Globals are defensible for constants and for one genuinely shared item; otherwise pass parameters, keep private state in a static local, and put shared state inside a class.*

Operators, precedence, associativity and expression evaluation

This section settles the order in which a C++ expression is evaluated. It replaces the precedence table students are commonly given, which ranks operators C++ does not possess and omits the logical, conditional and assignment operators altogether, with a complete table carrying the associativity column that the common one has never had. It then works integer division and the modulus until neither can surprise anyone, shows that the logical operators stop as soon as the answer is settled, and evaluates two expressions by hand against the numbers the machine actually printed.

BINDER Question 38

A chart above the door of the computer room at Kishapu ranks the C++ operators from highest precedence down to lowest. Nobody has asked what the blank strip down its right hand side is for.

Classify the C++ operators and state the rules of precedence and associativity that decide the order an expression is evaluated in.

Model Answer

C++ operators are classified by the work they do, into arithmetic, relational, logical, bitwise, assignment, increment and decrement, and the conditional and comma operators; and the order in which they are applied is fixed by two rules, precedence, which ranks operators of different kinds, and associativity, which settles operators of the same rank.

Precedence ranks unlike operators: an operator of higher precedence takes its operands before one of lower precedence takes them. In 2 + 3 * 4 the multiplication goes first, so the addition receives 12 rather than a 3 it never sees, and the whole is 14 and not 20.

Associativity settles equal operators: when two operators of the same rank compete for one operand between them, precedence has nothing left to say and associativity decides. Left to right grouping gives the operator on the left first claim, right to left gives it to the one on the right.

Only three rows group from the right: the unary operators, the conditional operator and every assignment operator. Every other row of Table 4.10 groups from the left. Those three are worth learning as a set, because they are the only exceptions in the whole table.

Arity is the second classification: an operator is unary if it takes one operand, binary if it takes two, ternary if it takes three. C++ has exactly one ternary operator, the conditional, which is why it is often named by its shape alone.

What the two rules do not decide: they fix which operator takes which operands, and nothing more. They do not fix the order in which the operands themselves are worked out, so an expression that alters one variable twice has no answer the language will stand behind.

Level	Operators	What they are	Associativity
1	::	Scope resolution	Left to right
2	a++ a-- f() a[i] . ->	Postfix increment and decrement, call, subscript, member	Left to right
3	++a --a +a -a ! ~ (type) sizeof	Prefix increment and decrement, unary sign, logical not, bitwise not, cast	Right to left
4	* / %	Multiply, divide, remainder	Left to right
5	+ -	Add, subtract	Left to right
6	<< >>	Bitwise shift left and right	Left to right
7	< <= > >=	Relational	Left to right
8	== !=	Equality and inequality	Left to right
9	&	Bitwise and	Left to right
10	^	Bitwise exclusive or	Left to right
11	|	Bitwise or	Left to right
12	&&	Logical and	Left to right
13	||	Logical or	Left to right
14	?:	Conditional, the one ternary operator	Right to left
15	= += -= *= /= %=	Assignment and compound assignment	Right to left
16	,	Comma	Left to right

Table 4.10: *The C++ operators in precedence order, highest first, with the associativity of every row. The three rows that group from the right are the unary operators, the conditional and the assignments; every other row groups from the left.*

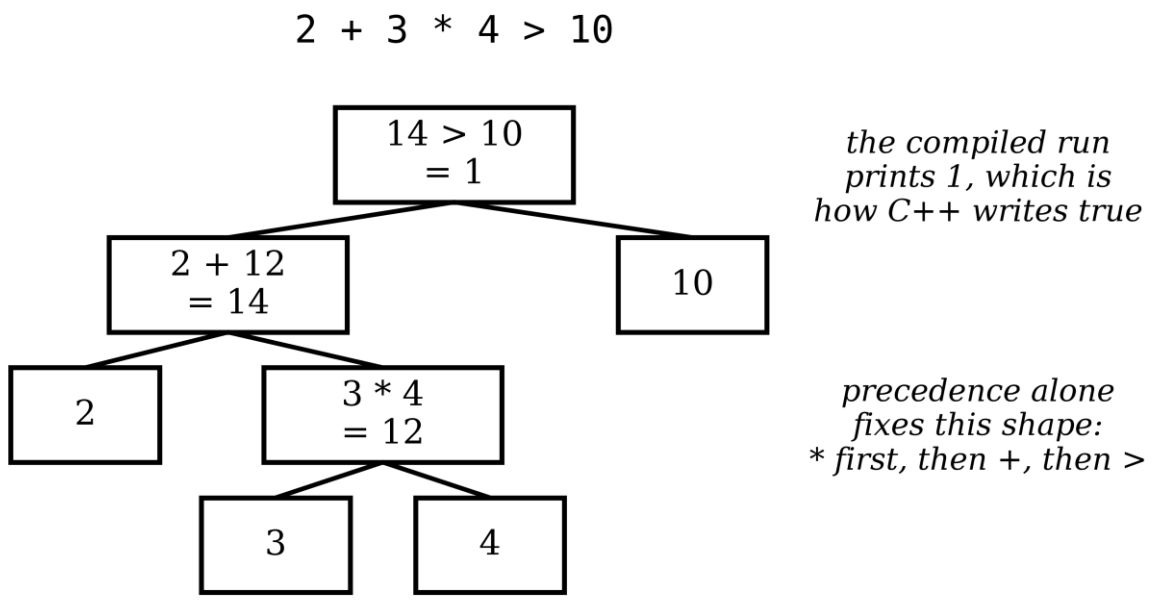


Figure 4.8: *An expression resolved by precedence alone, with the value carried up from every node. Precedence decides which operator takes its operands first, and nothing else is needed here.*

Read Table 4.10 twice, once down for precedence and once across for associativity, because an answer built from the first column alone stops halfway.

The Master Key: *Precedence ranks operators of different kinds and associativity settles operators of the same rank; only the unary operators, the conditional and the assignments group from the right.*

BINDER Question 39

Nothing looked wrong with the program a student at Monduli wrote to work out a pass rate. It printed zero for a class of forty and zero again for a class of one.

Explain integer division, the modulus operator and what happens when operands of different types are combined.

Model Answer

When both operands of the division operator are integers, C++ performs integer division, in which the fractional part of the answer is discarded rather than rounded, so 5 / 2 is 2.

Why the fraction disappears: an operation on two values of integer type produces a value of integer type, and an integer cannot hold a fraction, so the part after the point is thrown away. Discarding is not rounding: 5 / 2 is 2, and 9 / 10 is 0 rather than 1.

One floating point operand changes it: if either operand is a floating point value, the other is converted to floating point before the division and the answer keeps its fraction. Program 4.28 prints 2.5 for both 5.0 / 2 and 5 / 2.0. The conversion is automatic and needs no cast written by hand.

The modulus gives what is left: a % b is the remainder after a is divided by b, so 5 % 2 is 1, because 5 is two twos and one over. Division and modulus are a pair, and between them they recover the original value exactly.

The remainder takes the left sign: the sign of the answer follows the value being divided and never the divisor. Program 4.28 prints -3 for -7 % 4 and 3 for 7 % -4. Integer division truncates towards zero for the same reason, which is why -7 / 4 prints -1.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      cout << "5 / 2   is " << 5 / 2 << endl;
 7      cout << "5 % 2   is " << 5 % 2 << endl;
 8      cout << "9 / 10  is " << 9 / 10 << endl;
 9      cout << "5.0 / 2 is " << 5.0 / 2 << endl;
10      cout << "5 / 2.0 is " << 5 / 2.0 << endl;
11      cout << "-7 % 4  is " << -7 % 4 << endl;
12      cout << "7 % -4  is " << 7 % -4 << endl;
13      cout << "-7 / 4  is " << -7 / 4 << endl;
14      return 0;
15  }
```

Output

```
5 / 2   is 2
5 % 2   is 1
9 / 10  is 0
5.0 / 2 is 2.5
5 / 2.0 is 2.5
-7 % 4  is -3
7 % -4  is 3
-7 / 4  is -1
```

Program 4.28: *Integer division, the modulus and one floating point operand, every claim of this answer printed by the machine that made it.*

Both operands must be integers: the modulus is defined only for integer operands. Applying it to a double is not a fault to be discovered when the program runs; Program 4.29 shows the compiler refusing the line outright and naming the two types it was offered.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      double price = 5.0;
7      cout << price % 2 << endl;
8      return 0;
9  }
```

Compiler says

```
error: invalid operands of types 'double' and 'int'
to binary 'operator%'
```

Program 4.29: *The modulus applied to a double. The fault is caught before the program can be built, not while it runs.*

Look at the types before you look at the numbers, because a division that keeps printing zero is nearly always a division between two integers.

The Master Key: *Integer divided by integer gives an integer with the fraction discarded, one floating point operand converts the other and keeps the fraction, and the remainder from the modulus carries the sign of the left operand.*

BINDER Question 40

Twenty five was the number the class began from. The first two answers came back with no argument at all, and the third and the fourth split the room in half.

Distinguish the prefix and postfix forms of the increment and decrement operators, and state the value each form gives to the expression it appears in.

Model Answer

Both forms change the variable by exactly one, and they differ only in what the expression itself is worth: the prefix form gives the new value and the postfix form gives the old one.

The prefix form: ++x adds one to x and then hands the new value over to whatever the expression is doing with it. Written the other way round, x is updated first and the expression is worth the updated value.

The postfix form: x++ hands over the value x had, and only then adds one to x. The expression is worth the old value, and x has been changed by the time the statement finishes, so both things are true at once.

The variable ends the same either way: this is the part that is missed. Where the value of the expression is thrown away, as in a statement that reads only x++; on its own, the two forms cannot be told apart by their effect, which is exactly why the difference is so easily forgotten.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int x = 25;
 7      cout << "x at the start is " << x << endl;
 8      cout << "x++ is worth     " << x++ << endl;
 9      cout << "x-- is worth     " << x-- << endl;
10      cout << "++x is worth     " << ++x << endl;
11      cout << "--x is worth     " << --x << endl;
12      cout << "x at the end is  " << x << endl;
13      return 0;
14  }
```

Output

```
x at the start is 25
x++ is worth     25
x-- is worth     26
++x is worth     26
--x is worth     25
x at the end is  25
```

Program 4.30: *The four forms in order, from one starting value. The second and third lines print the same number for opposite reasons.*

Reading the trace: in Program 4.30 x starts at 25. Then x++ prints 25 and leaves 26 behind; x-- prints 26 and leaves 25; ++x raises x to 26 and prints 26; --x lowers it to 25 and prints 25. The two middle lines both print 26 for opposite reasons, and x ends where it began.

What must never be written: an expression that changes one variable twice, such as i = i++ + ++i, has no defined answer in C++. Compilers disagree about it and none of them is wrong, so it must not be written, traced, or set as an exercise with an answer attached.

Ask what the expression is worth rather than what the variable becomes, because the variable becomes the same thing either way.

The Master Key: *Prefix and postfix both add one to the variable; prefix gives the expression the new value and postfix gives it the old one, and an expression that alters one variable twice has no defined answer at all.*

BINDER Question 41

A stock program at Kisarawe divided the value of a delivery by the number of crates, and aborted the morning one arrived with none recorded. The repair was a single test placed to the left of the division.

Explain short-circuit evaluation of the logical operators, and state why it matters.

Model Answer

The logical operators evaluate their left operand first and stop the moment the answer is settled, so the right operand is not evaluated at all when the left one has already decided the result.

And stops on a false left operand: if the left operand of && is false, the whole expression is false whatever the right operand might have turned out to be. C++ therefore does not work the right operand out, and the result is false.

Or stops on a true left operand: if the left operand of || is true, the whole expression is true regardless of the right, and the right is skipped in the same way. Only when the left operand fails to settle the question is the right one reached at all.

The guard idiom depends on it: this is what makes crates != 0 && total / crates > 3 safe. When crates holds zero the left operand is false, the division to its right is never performed, and the program carries on instead of aborting. Reverse the two operands and the guard is worthless.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  bool report(int tag)
 5  {
 6      cout << "    right operand " << tag << " really ran" << endl;
 7      return true;
 8  }
 9
10  int main()
11  {
12      bool no = false;
13      bool yes = true;
14      cout << "no && report(1)" << endl;
15      bool r1 = no && report(1);
16      cout << "    value " << r1 << endl;
17      cout << "yes || report(2)" << endl;
18      bool r2 = yes || report(2);
19      cout << "    value " << r2 << endl;
20      cout << "no || report(3)" << endl;
21      bool r3 = no || report(3);
22      cout << "    value " << r3 << endl;
23      int total = 20;
24      int crates = 0;
25      if (crates != 0 && total / crates > 3)
26      {
27          cout << "above three per crate" << endl;
28      }
29      else
30      {
31          cout << "guard held, no division attempted" << endl;
32      }
33      return 0;
34  }
```

Output

```
no && report(1)
    value 0
yes || report(2)
    value 1
no || report(3)
    right operand 3 really ran
    value 1
guard held, no division attempted
```

Program 4.31: *A function that prints a line, placed on the right of each operator. It runs once, and the guard at the end divides by nothing.*

The skipping is visible, not argued: Program 4.31 puts a function that prints a line on the right of each operator. That line appears once only, for the one expression whose left operand fails to settle the answer, and the guard at the end reports that no division was attempted.

What follows for the programmer: never put work on the right of && or || that the program actually needs done. A value read in, an assignment, or any call that changes something belongs in a statement of its own, because whether it runs at all now depends on a value computed to its left.

Put the cheap test and the safe test on the left, and the operator does the protecting for you.

The Master Key: *The and operator stops at a false left operand and the or operator stops at a true one, the right operand is then never evaluated, and that is precisely what lets a test guard the operation written to its right.*

REAL Question 42

The photocopy has been round the class twice and its heading has nearly gone. It is a precedence table of ten levels, and three of the operators printed on it cannot be compiled at all.

Identify everything that is wrong with the precedence table below, and say what a student cannot do with it.

A precedence table of the kind commonly handed out, highest level first:

Level 1 ( ) [ ] . ->

Level 2 ++ -- ! ~ typeof is as

Level 3 * / %

Level 4 + - and + again, for joining two pieces of text

Level 5 << >>

Level 6 < <= > >=

Level 7 == !=

Level 8 &

Level 9 ^

Level 10 |

The table ends at level 10. There is no column headed associativity.

Model Answer

The table is not a C++ table at all: three of the operators on it belong to other languages, four whole families of C++ operator are absent, and there is no associativity column, so neither a compound condition nor a repeated subtraction can be resolved from it.

Operators C++ does not have: typeof is not part of standard C++ and survives only as a compiler extension, while is and as are C# operators for testing and converting types while a program runs. None of the three may appear in a standard C++ expression, so the table has been lifted from a source written for another language.

A level for joining text is not a level: C++ has no built in operator that joins two pieces of text. Joining is a library facility supplied by the string type, not a rank in the operator table, and a plain array of characters cannot be joined with a plus sign at all. Level 4 as printed invites an attempt the language will refuse.

The four missing families: the logical operators && and ||, the conditional ?:, every assignment operator including the compound forms, and the comma are all absent. Those are the lowest ranked operators in the real table, which is exactly why their absence matters: their place is the one a reader cannot guess.

No associativity column anywhere: the table ranks operators against one another and then says nothing at all about two operators of equal rank. Rank by itself cannot tell you whether a - b - c groups from the left or from the right, and the two groupings give different numbers.

What a student cannot do with it: a < b && c < d cannot be evaluated, because && is not on the table and its rank against the relational operators is therefore unknown. a - b - c cannot be evaluated either, because both operators sit on level 4 and nothing printed anywhere says which of them takes the middle value first.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      bool p = true;
 7      bool q = false;
 8      bool r = false;
 9      cout << "p || q && r   is " << (p || q && r) << endl;
10      cout << "(p || q) && r is " << ((p || q) && r) << endl;
11      cout << "p || (q && r) is " << (p || (q && r)) << endl;
12      return 0;
13  }
```

Output

```
p || q && r   is 1
(p || q) && r is 0
p || (q && r) is 1
```

Program 4.32: *The two groupings of one expression, printed side by side. Only the first is what C++ does with the unbracketed line.*

The correction, and the proof: Table 4.10 supplies both missing things. Program 4.32 confirms the part most often misread, that && binds more tightly than ||, so p || q && r means p || (q && r) and prints 1 where the other grouping prints 0. The compiler even offers a warning on that line, which measures how often it is got wrong.

A precedence table with no associativity column is not half a table; it is a table that cannot answer the questions it is consulted for.

The Master Key: *Test any precedence table for three things before trusting it: that every operator on it is a C++ operator, that the logical, conditional, assignment and comma operators are on it, and that it carries an associativity column.*

REAL Question 43

One hundred kilograms of fish came ashore at Ikwiriri, thirty were sold on the beach and twenty went away on the lorry. Two clerks worked out what was left, disagreed, and neither had made a mistake.

Evaluate the expression 100 - 30 - 20 by hand, then say which rule decides it.

Model Answer

The expression is worth 50, and it is associativity rather than precedence that makes it so, because both operators are subtractions of equal rank and subtraction groups from the left, so the leftmost subtraction takes the 100 first.

Why precedence cannot settle it: precedence ranks operators of different kinds against each other. Here both operators are the same operator at the same rank, competing for the same middle value, and a rule about rank has nothing to say when the ranks are equal. The first column of Table 4.10 is silent.

The rule that does settle it: subtraction associates from left to right, so the expression groups as (100 - 30) - 20. The other grouping, 100 - (30 - 20), is what right to left association would produce, and C++ never applies it to a subtraction.

The hand evaluation: take the leftmost pair first, so 100 - 30 is 70. That 70 now becomes the left operand of the second subtraction, and 70 - 20 is 50. Written out in two lines like that, the grouping is on the page as evidence rather than being carried in the head.

The two groupings disagree: grouped from the right instead, 30 - 20 is 10 and 100 - 10 is 90. Fifty and ninety are both honest arithmetic and only one of them is C++, which is why associativity is a rule and not a matter of taste. Figure 4.9 draws both trees side by side.

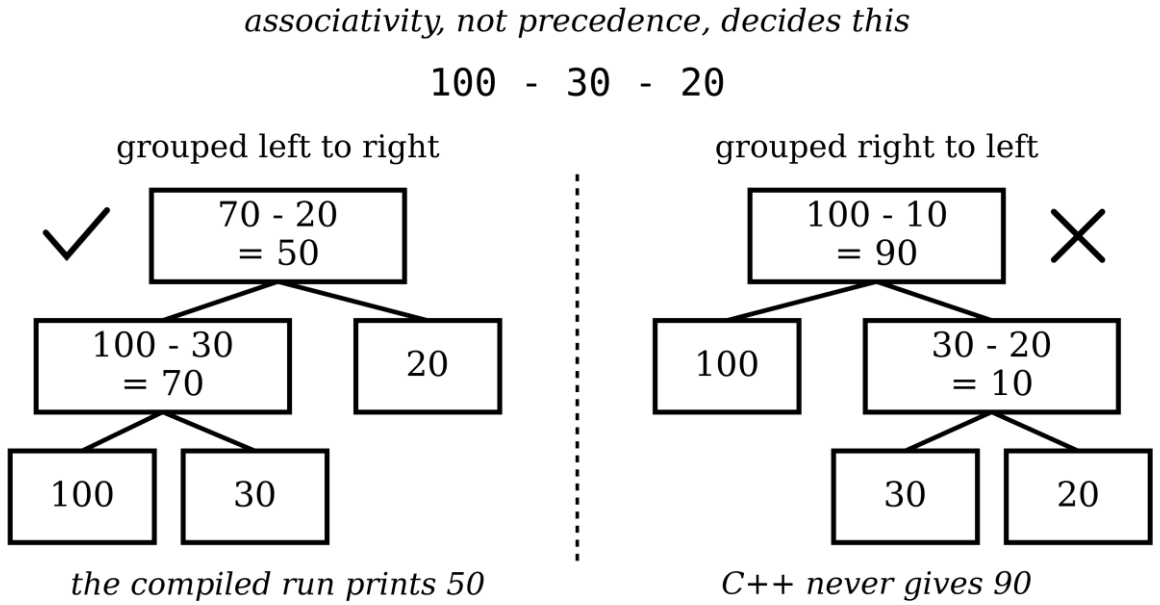


Figure 4.9: *An expression precedence cannot settle, drawn both ways. Only one of the two answers is what the machine produces, and associativity is the rule that says which.*

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      cout << "100 - 30 - 20   is " << 100 - 30 - 20 << endl;
 7      cout << "(100 - 30) - 20 is " << (100 - 30) - 20 << endl;
 8      cout << "100 - (30 - 20) is " << 100 - (30 - 20) << endl;
 9      return 0;
10  }
```

Output

```
100 - 30 - 20   is 50
(100 - 30) - 20 is 50
100 - (30 - 20) is 90
```

Program 4.33: *The bare expression and both bracketings. Two of the three agree, and the one that disagrees is the grouping C++ does not use.*

Confirmed by the machine: Program 4.33 prints all three forms. The bare expression and the left grouping both print 50, and the right grouping prints 90. The clerk who wrote 50 was reading the tally the way a compiler reads the line.

Where else the same question bites: it decides a / b / c, which means (a / b) / c and not a / (b / c). It decides a = b = c in the other direction, because assignment is one of the three families that group from the right, so that line assigns c to b and then to a.

When two operators of equal rank meet over one operand, stop reading the precedence column and read the one beside it.

The Master Key: *Precedence separates operators of different rank, and where the ranks are equal associativity decides; C++ arithmetic groups from the left, so 100 - 30 - 20 is 50 and never 90.*

REAL Question 44

An exercise at Utete asked for a program converting Fahrenheit to Celsius, five ninths of the difference from thirty two. Every program the class wrote compiled cleanly and gave boiling water the same answer as ice.

Program 4.34 below converts Fahrenheit to Celsius. Explain why it prints what it prints, name the rule responsible, and correct it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int fahrenheit = 0;
 7      cout << "Enter degrees Fahrenheit: ";
 8      cin >> fahrenheit;
 9      double celsius = 5 / 9 * (fahrenheit - 32);
10      cout << "Celsius is " << celsius << endl;
11      return 0;
12  }
```

Typed at the program

```
212
```

Output

```
Enter degrees Fahrenheit: Celsius is 0
```

Program 4.34: *The Fahrenheit conversion as it is commonly written. The input was 212, the temperature of boiling water.*

Model Answer

The program prints 0 for every input, because 5 / 9 is a division of one integer by another, integer division discards the fraction, and 5 / 9 is therefore 0, after which the whole product is 0 whatever temperature was typed in.

What the machine actually does: with 212 typed in, the bracketed difference is 180, which is right. The trouble sits to its left. 5 / 9 is worked out as an integer division, and 5 divided by 9 is 0 with 5 left over, so the expression has already become 0 times 180.

Why the multiplication cannot rescue it: multiplication and division share a rank and group from the left, so 5 / 9 * (f - 32) means (5 / 9) * (f - 32). The division is performed first and the fraction is gone before the difference ever arrives to be multiplied by it.

Why nothing warns you: the program is entirely legal. Both operands of the division are of integer type, integer division is exactly what the language defines for that case, and storing the result in a double afterwards converts a zero into 0.0. There is no diagnostic to notice, which is what makes the trap durable.

The rule named: integer division. Where both operands of the division operator are of integer type the result is of integer type and the fractional part is discarded, not rounded, and no warning is issued because nothing improper has occurred.

The correction: make one operand floating point. Program 4.35 writes 5.0 / 9.0, so the division is a floating point division worth about 0.5556, and 212 now gives 100 exactly. Writing static_cast<double>(5) / 9 does the same job with the conversion visible, which is the form later sections use.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int fahrenheit = 0;
 7      cout << "Enter degrees Fahrenheit: ";
 8      cin >> fahrenheit;
 9      double celsius = 5.0 / 9.0 * (fahrenheit - 32);
10      cout << "Celsius is " << celsius << endl;
11      return 0;
12  }
```

Typed at the program

```
212
```

Output

```
Enter degrees Fahrenheit: Celsius is 100
```

Program 4.35: *The same program with one operand made floating point. The same input now gives the answer it should always have given.*

How to find it in your own work: look at every division and ask what type each operand has. If both are integers and the answer was meant to carry a fraction, the fraction has already gone, and no amount of care further down the program will bring it back.

A program that prints zero for every input is rarely broken arithmetic; it is nearly always a division that never left the integers.

The Master Key: *5 / 9 is 0, so 5 / 9 * (f - 32) is 0 for every f; write 5.0 / 9.0 and the division keeps its fraction.*

BUILD Question 45

There is a particular silence in a computer room when two students have hand evaluated the same expression and arrived at different numbers. It lasts until somebody runs the expression.

Write a program that prints the value of each expression in a given list, labelled with the expression exactly as it is written.

Print the value of each expression in a given list, one to a line.

Label every line with the expression exactly as it is written, so a reader can lay a hand evaluation beside the output and see which line disagrees.

The list must include an integer division, a modulus with a negative left operand, a division mixing an integer with a double, a compound condition using both the and operator and the or operator, and a pair of expressions differing only in a bracket.

Every value printed must be the value C++ computes, not the value a reader expects.

Model Answer

Program 4.36 declares three variables and prints eight labelled lines, and every value on those lines is what C++ computed rather than what a reader assumed.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int a = 17;
 7      int b = 5;
 8      double d = 2.0;
 9      cout << "a / b           is " << a / b << endl;
10      cout << "a % b           is " << a % b << endl;
11      cout << "-a % b          is " << -a % b << endl;
12      cout << "a / d           is " << a / d << endl;
13      cout << "a > b && b > 0  is " << (a > b && b > 0) << endl;
14      cout << "a > b || (b == 5 && b > 9) is "
15           << (a > b || (b == 5 && b > 9)) << endl;
16      cout << "a - b - 2       is " << a - b - 2 << endl;
17      cout << "a - (b - 2)     is " << a - (b - 2) << endl;
18      return 0;
19  }
```

Output

```
a / b           is 3
a % b           is 2
-a % b          is -2
a / d           is 8.5
a > b && b > 0  is 1
a > b || (b == 5 && b > 9) is 1
a - b - 2       is 10
a - (b - 2)     is 14
```

Program 4.36: *Eight expressions, each labelled with itself, so a hand evaluation can be checked line by line.*

What each line is there for: a / b is the integer division and prints 3. a % b and -a % b are the remainders with a positive and a negative left operand, and the second prints -2. a / d keeps its fraction, and the last two lines differ only by a bracket.

The label is the expression: each label repeats the expression as written rather than describing it, so a reader can put their own working alongside and see at once which line they disagree with. A label reading first result would be no use.

The one design decision: the relational and logical expressions are wrapped in brackets before being sent to the output stream. The stream operator outranks both families, so without those brackets the compiler reads the line as a shift and rejects it, which is a precedence fault inside a program about precedence.

What the output settles: the last two lines print 10 and 14, which is the associativity of subtraction stated in numbers rather than words, and both conditions print 1, because C++ writes true as 1.

Keep the program and add a line whenever an expression surprises you, and it becomes a record of what you have got wrong once.

The Master Key: *A labelled expression printer turns an argument about precedence into a run, and the label has to be the expression itself for the comparison to be worth anything.*

BUILD Question 46

The clock at the bus desk counts in seconds and nothing else, so a journey shows as 7265 and the clerk does the rest in her head. Two divisions and two remainders would do it for her.

Write one program that converts a Fahrenheit temperature to Celsius and converts a count of seconds into hours, minutes and seconds.

Read a temperature in degrees Fahrenheit and print it in degrees Celsius to one decimal place.

Then read a whole number of seconds and print it as hours, minutes and seconds.

One program is to do both.

Model Answer

Program 4.37 does both jobs in one run, and the point of putting them together is that they need different divisions: the temperature must keep its fraction and the seconds must lose theirs.

```
 1  #include <iostream>
 2  #include <iomanip>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      double fahrenheit = 0.0;
 8      cout << "Enter degrees Fahrenheit: ";
 9      cin >> fahrenheit;
10      double celsius = 5.0 / 9.0 * (fahrenheit - 32.0);
11      cout << fixed << setprecision(1);
12      cout << "Celsius is " << celsius << endl;
13      int seconds = 0;
14      cout << "Enter a whole number of seconds: ";
15      cin >> seconds;
16      int hours = seconds / 3600;
17      int minutes = seconds % 3600 / 60;
18      int rest = seconds % 60;
19      cout << seconds << " seconds is " << hours << " h "
20           << minutes << " min " << rest << " s" << endl;
21      return 0;
22  }
```

Typed at the program

```
98.6
7265
```

Output

```
Enter degrees Fahrenheit: Celsius is 37.0
Enter a whole number of seconds: 7265 seconds is 2 h 1 min 5 s
```

Program 4.37: *One program, two divisions of opposite kinds. The inputs were 98.6 degrees Fahrenheit and 7265 seconds.*

The Celsius half: writing 5.0 / 9.0 makes the division a floating point division, so the fraction survives to be multiplied by the difference. The printing is fixed at one decimal place by setprecision, which is why iomanip is included.

The seconds half: hours is seconds / 3600, an integer division whose discarded fraction is exactly what was wanted. Minutes is seconds % 3600 / 60, which takes the remainder left after the whole hours and divides it down. The leftover seconds are seconds % 60.

Why the two operators travel together: division answers how many whole units fit and the modulus answers what is left over, so every conversion of one unit into several is these two applied in turn. For 7265 seconds that is 2 hours, then 65 seconds remaining, which is 1 minute and 5 seconds.

Printing is not rounding: setprecision changes the text that appears and does not change the value held in celsius. The distinction matters the moment a printed value is read back in and compared with the one that produced it.

Divide for the whole units and take the remainder for what is left, and any conversion between mixed units falls out in two lines.

The Master Key: *Floating point division keeps the fraction while integer division with the modulus splits a quantity into whole units and a remainder, and the types of the operands decide which a program gets.*

HOT Question 47

Somebody has written a rule on the board and underlined it twice: always use brackets. Underneath, in another hand, is a line so thick with brackets that nobody in the room can read it.

Evaluate the practice of relying on operator precedence rather than writing brackets. Reach a judgement with a stated condition.

Model Answer

Rely on precedence only where the ranking is one that every competent C++ programmer holds without checking, which in practice means arithmetic against arithmetic and arithmetic against a comparison; bracket wherever two different families meet.

The case for relying on precedence: the rules belong to the language and do not vary between compilers, so a correct unbracketed expression is not fragile in the least. Brackets cost characters, and past a certain density they cost the reader the shape of the expression, which is the very thing they were added to reveal.

What every programmer genuinely knows: that multiplication and division bind more tightly than addition and subtraction, that all of the arithmetic binds more tightly than the comparisons, and that assignment happens last of all. These are held reliably, and bracketing them tells the reader nothing they did not have.

What is routinely misremembered: that && binds more tightly than ||; that the bitwise operators bind less tightly than equality, so a mask tested with & and == does not mean what it appears to; that the shifts bind less tightly than addition; and that the conditional binds very nearly last. Bracket all four.

What brackets cost when overused: every pair a reader meets has to be matched by eye before the line can be understood, so a line carrying six pairs is slower to check than the same line carrying two. Heavy bracketing also hides a genuine error, because a reader who has stopped counting will not notice the pair that is missing.

The compiler has already decided part of it: a warning is issued where && appears inside || without brackets, and another where an assignment sits where a comparison was probably meant. Those two warnings mark the places the profession has agreed are dangerous, and they are a better guide than any private rule.

What to do under examination conditions: brackets never change a correct answer and cost only ink, so write them wherever an expression mixes families. Where a question asks you to evaluate an expression as printed, do not add any: the question is testing the table, and bracketing it answers a different question from the one asked.

The judgement: the practice is defensible within one family of operators and indefensible across families. Write a * b + c and write c > a + b without hesitation, and never write a && b || c or a mask test unbracketed, because those are the expressions on which most readers give the wrong answer.

Brackets are documentation rather than correction: put them where a reader would otherwise have to look something up, and nowhere else.

The Master Key: *Rely on precedence inside one family of operators, where the ranking is common knowledge, and bracket wherever families meet, because that is exactly where the ranking is misremembered.*

Selection: if, else, nested tests and switch

This section settles how a program chooses between paths. It gives the three shapes a conditional test can take, states the one rule that decides which if an else belongs to, and answers the question about the types a switch may accept that is asked far more often than it is answered. Several of its programs are wrong on purpose: one awards the wrong grade to almost every candidate without producing a single diagnostic, and one silently refuses to print the line its own indentation promised.

BINDER Question 48

The barrier at the weighbridge outside Kibiti has two positions and no others: under the limit and the arm lifts, over it and the arm stays down. Every test a program makes has that shape.

Explain the three forms of conditional statement in C++, and state the shape of each.

Model Answer

C++ has three forms of conditional statement: the plain if, which runs a body or does not; the if with an else, which runs exactly one of two; and the else-if ladder, in which at most one body runs and the first true test claims it.

The plain if: written as the keyword if, a condition in brackets, then one statement or one braced block. When the condition is true the body runs; when it is false it is skipped and control passes to whatever follows. There is no second path in it.

The if with an else: the same header and its body, then the keyword else and a second body. Exactly one of the two runs, never both and never neither, so the else is the whole of what is left over rather than a second test.

The else-if ladder: each else is followed at once by another if, so the tests are tried in written order until one is true. That body runs, every test below it is skipped, and control leaves the ladder. A closing else with no condition takes whatever reached it.

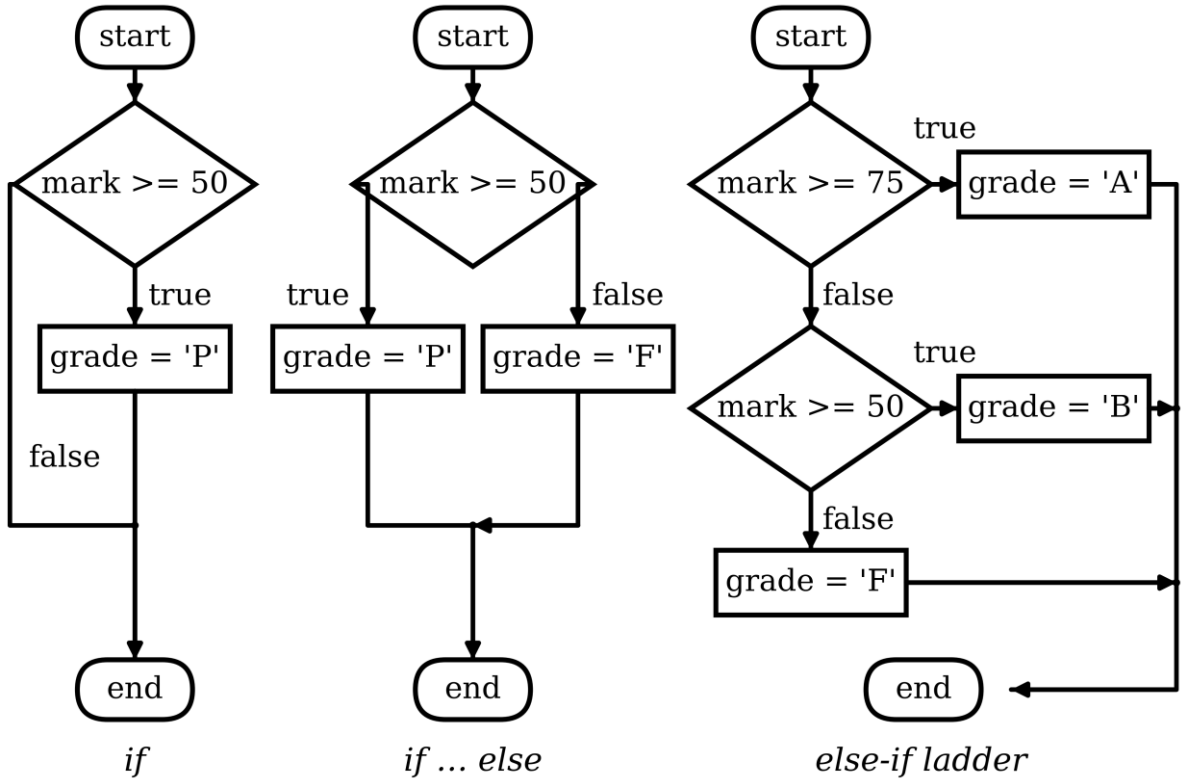


Figure 4.10: *The three shapes of a conditional test. Every decision has exactly two ways out, and in the else-if ladder only one arm can ever be taken.*

When no arm matches: a plain if does nothing, silently and correctly. A ladder with no closing else also does nothing, which is the shape that loses marks, because a variable it was meant to set keeps whatever it held before.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int mark = 82;
 7      if (mark >= 50)
 8      {
 9          cout << "if: the body ran" << endl;
10      }
11      if (mark >= 90)
12      {
13          cout << "if else: the true arm ran" << endl;
14      }
15      else
16      {
17          cout << "if else: the false arm ran" << endl;
18      }
19      if (mark >= 75)
20      {
21          cout << "ladder: grade A" << endl;
22      }
23      else if (mark >= 60)
24      {
25          cout << "ladder: grade B" << endl;
26      }
27      else
28      {
29          cout << "ladder: grade F" << endl;
30      }
31      return 0;
32  }
```

Output

```
if: the body ran
if else: the false arm ran
ladder: grade A
```

Program 4.38: *The three forms in one run, from a mark of 82. The ladder awards one grade and stops, although both of its tests would have been true.*

What may sit in the brackets: any expression that converts to bool. A comparison is of type bool already; an int or a char is converted, and the rule is that zero is false and every other value true. Minus four is true, and only zero is false.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int count = 0;
 7      int balance = -4;
 8      if (count)
 9      {
10          cout << "zero counted as true" << endl;
11      }
12      else
13      {
14          cout << "zero counted as false" << endl;
15      }
16      if (balance)
17      {
18          cout << "minus four counted as true" << endl;
19      }
20      return 0;
21  }
```

Output

```
zero counted as false
minus four counted as true
```

Program 4.39: *A condition of integer type. Zero is the one value that counts as false, and a negative value counts as true like any other.*

Count the ways out before writing the tests: one for a plain if, two for an if with an else, and one more than the number of tests for a ladder.

The Master Key: *The plain if runs a body or skips it, the if with an else runs one of two, and the ladder runs at most one body, chosen by the first true test.*

BINDER Question 49

A card taped to the counter of a mobile money kiosk lists five services against the keys 1 to 5. One statement can serve that counter in place of five separate tests, and it refuses two common types.

Explain the switch statement: what it can test, how control enters a case, what break does, and what happens when break is omitted.

Model Answer

A switch takes one expression of integral or enumeration type, compares it against the constant labels on its cases and transfers control to the label that matches; break ends a case, and without it control carries on into the case below.

What a switch may test: the expression in the brackets must be of integral or enumeration type: bool, char, short, int, long, long long and any enumeration. A double is not integral and neither is a std::string, so the compiler rejects both, as Program 4.40 shows.

How control enters a case: the labels must be constant expressions known at compile time, and no two may carry the same value. The value is compared against each until one matches, and control jumps to that point inside the block rather than into a body of its own.

What break does: break ends the switch at once and passes control to the first statement after the closing brace. Leave it out and control falls out of the matching case and runs through every case beneath it, labels ignored. The same word also leaves a loop, which the next section covers.

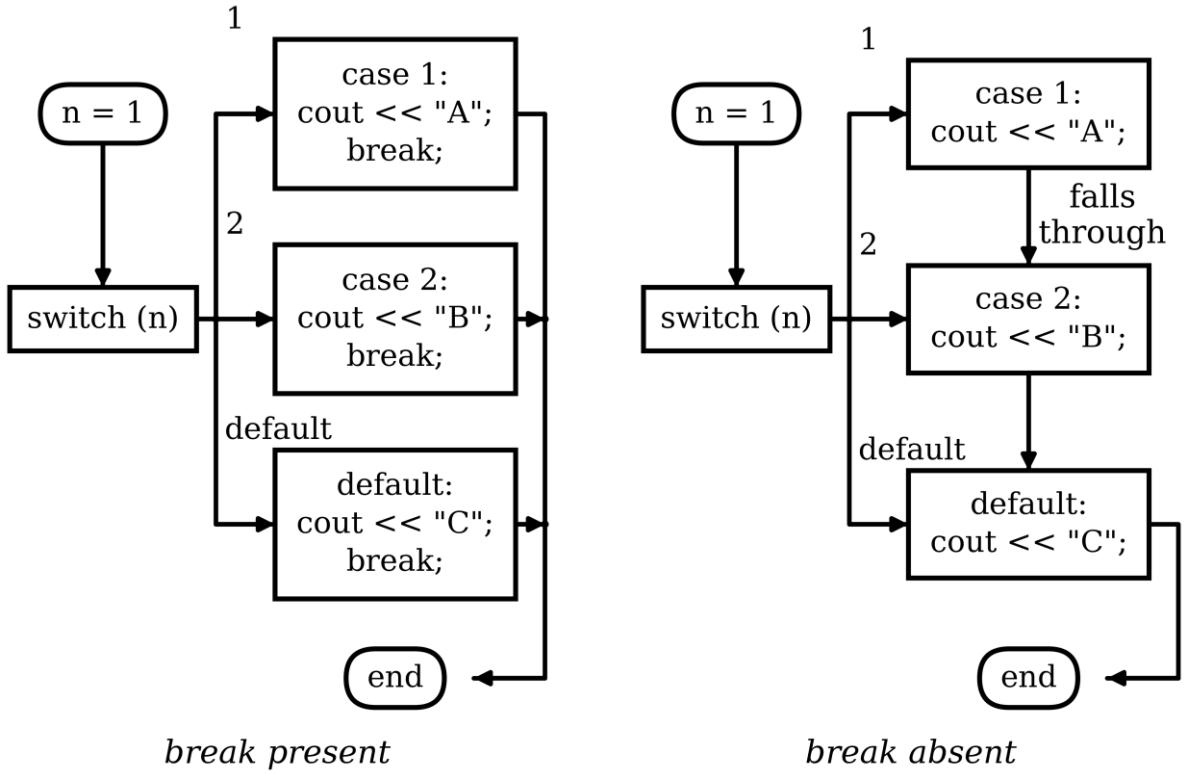


Figure 4.11: *A switch with and without break. Without it, control falls out of the matching case and straight through every case below it.*

The default label: control jumps to default when no case matched. It may be written anywhere among the cases and is conventionally last, and a switch with no default and no matching case does nothing at all.

Falling through on purpose: labels stacked one above another with nothing between them share a single body, which is a technique and not a mistake; Program 4.46 answers five vowels with one statement that way. Fall-through is a fault only when a case does work and leaks into the next.

The expression written in the brackets	Allowed	Why
int, short, long, long long	Yes	integral types, which is exactly what a switch requires
char	Yes	a character type is an integral type in C++
bool	Yes	also integral, although two values are better served by an if
an enumeration	Yes	an enumeration converts to an integer, and its enumerators make sound labels
float or double	No	not an integral type, so the switch itself is rejected
std::string	No	a class type with no conversion to an integer, refused the same way
a pointer, or an array of char	No	a pointer is not an integral type either

Table 4.11: *What may and may not be written in the brackets of a switch. Everything in the accepted rows is an integral or an enumeration type, and everything else belongs in a ladder.*

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      double price = 2.50;
 8      string service = "binding";
 9      switch (price)
10      {
11          case 2:
12              cout << "two shillings" << endl;
13              break;
14      }
15      switch (service)
16      {
17          case 1:
18              cout << "one shilling" << endl;
19              break;
20      }
21      return 0;
22  }
```

Compiler says

```
error: switch quantity not an integer
error: switch quantity not an integer
```

Program 4.40: *A switch on a double and a switch on a string. Neither reaches the running stage, and the compiler gives the identical message for both.*

A switch answers one question about one integral value; where the question is a range or the value a string, it is the wrong statement to reach for.

The Master Key: *A switch tests one expression of integral or enumeration type against constant labels, jumps to the one that matches, and runs on through the cases below it until a break or the closing brace stops it.*

BINDER Question 50

A student at Madaba printed a program on paper, ruled a pencil line down each level of indentation, and read the answer straight off the page. The reading was careful and it was wrong.

Explain the dangling else problem and state the rule that resolves it.

Model Answer

Where an else follows two ifs and only one of them can have it, the else belongs to the nearest preceding if that has not already been given an else, however the lines happen to be indented.

Where the ambiguity comes from: an if may have an else and may equally do without one, so a text holding two ifs and one else has two possible readings. The language settles it by rule and not by layout: the else joins the nearest unmatched if.

What indentation is worth: nothing whatever to the compiler. Spaces, tabs and alignment are stripped out before the program is parsed, so moving a line to the left changes what a reader believes and leaves what runs exactly as it was.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int mark = 30;
 7      bool registered = false;
 8      cout << "checking one candidate" << endl;
 9      if (registered)
10          if (mark >= 50)
11              cout << "candidate has passed" << endl;
12      else
13          cout << "candidate is not registered" << endl;
14      cout << "check finished" << endl;
15      return 0;
16  }
```

Output

```
checking one candidate
check finished
```

Program 4.41: *The else is indented to line up with the outer if. The rule attaches it to the inner one, and the middle line never appears.*

What the layout promises: that a candidate who is not registered will be told so, and the opposite is what runs. The else belongs to the inner if, so the whole inner statement is the body of the outer one, and because registered is false the outer test skips all of it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int mark = 30;
 7      bool registered = false;
 8      cout << "checking one candidate" << endl;
 9      if (registered)
10      {
11          if (mark >= 50)
12          {
13              cout << "candidate has passed" << endl;
14          }
15      }
16      else
17      {
18          cout << "candidate is not registered" << endl;
19      }
20      cout << "check finished" << endl;
21      return 0;
22  }
```

Output

```
checking one candidate
candidate is not registered
check finished
```

Program 4.42: *The same program with braces on the outer arm. Nothing else changed, and the missing line appears.*

How the braces settle it: the braces close the body of the outer if before the else is reached, so the inner if is complete and can no longer collect one. The nearest unmatched if is now the outer if, which is what the layout said all along.

The habit that follows: students are told, correctly, that braces are optional around a single statement, and the advice is true and reckless together. Brace every arm and the ambiguity cannot arise. The compiler will suggest as much, warning that the else is ambiguous, but a warning is not an error.

Indentation is a promise made to the reader and braces are a promise made to the compiler, and only one of the two is binding.

The Master Key: *An else binds to the nearest if that has not already been given one, whatever the indentation says, so brace every arm and the question cannot be asked.*

REAL Question 51

Eighty two out of a hundred, and the printout said D. The program had compiled without one complaint and had given the wrong grade to everyone in the room except those who failed.

The grade program below compiles, runs, and awards the wrong grade to almost every candidate. Explain why, and correct it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int mark = 0;
 7      char grade = 'F';
 8      cout << "Enter the mark: ";
 9      cin >> mark;
10      if (mark >= 40)
11      {
12          grade = 'D';
13      }
14      else if (mark >= 50)
15      {
16          grade = 'C';
17      }
18      else if (mark >= 60)
19      {
20          grade = 'B';
21      }
22      else if (mark >= 75)
23      {
24          grade = 'A';
25      }
26      else
27      {
28          grade = 'F';
29      }
30      cout << "Grade is " << grade << endl;
31      return 0;
32  }
```

Typed at the program

```
82
```

Output

```
Enter the mark: Grade is D
```

Program 4.43: *The grade ladder as it is commonly written. The mark typed in was 82.*

Model Answer

The bands are tested from the bottom upwards, so the first test, mark at least 40, is true for every mark of 40 and above and gives all of them D; no test beneath it is ever reached, and the three higher grades are unreachable.

What the machine actually did: with 82 typed in, the first condition asks whether the mark is at least 40 and it is. That arm sets the grade to D, and the whole of the rest of the ladder is skipped, including the test for 75 that 82 also satisfies. The program prints Grade is D.

Why the first true test wins: an else-if ladder is an ordered chain and not a set of independent tests. Each test is reached only when every test above it has been false, so the arms are tried in written order, and the first arm to be true takes control and ends the ladder for good.

Which candidates are affected: every mark of 40 and above, which is most of an entry. Marks below 40 are graded correctly, because they alone fall past every test to the closing else. A candidate on 39 gets the right answer and a candidate on 90 does not, which is why the fault survives a careless test run.

Why nothing warned anybody: every line is valid C++. The conditions are well formed, the assignments are legal and the ladder is complete, so no rule of the language has been broken. This is a logic error, which the compiler cannot see: the program does exactly what it says, and what it says is wrong.

The correction: test the bands from the top downwards, 75 first, then 60, then 50, then 40, with the closing else for the rest. Program 4.44 is the same program with the four tests reversed and nothing else touched, and the same input now prints Grade is A.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int mark = 0;
 7      char grade = 'F';
 8      cout << "Enter the mark: ";
 9      cin >> mark;
10      if (mark >= 75)
11      {
12          grade = 'A';
13      }
14      else if (mark >= 60)
15      {
16          grade = 'B';
17      }
18      else if (mark >= 50)
19      {
20          grade = 'C';
21      }
22      else if (mark >= 40)
23      {
24          grade = 'D';
25      }
26      else
27      {
28          grade = 'F';
29      }
30      cout << "Grade is " << grade << endl;
31      return 0;
32  }
```

Typed at the program

```
82
```

Output

```
Enter the mark: Grade is A
```

Program 4.44: *The same ladder with the bands ordered from the highest downwards. The input was 82 again.*

How to check any ladder of bands: read it downwards and ask what the first test lets through. If a value belonging to a lower arm satisfies a test written above it, that lower arm can never be reached. Then try the boundary of every band, because a band tested at 50 and a band tested at 49 are different programs.

Order the bands before you write them, because a ladder is read in the order it was written and not in the order it was meant.

The Master Key: *In an else-if ladder the first true test wins and the rest are never reached, so overlapping bands must be tested from the highest downwards or the high grades become unreachable.*

REAL Question 52

Two jobs came out on the receipt at the printing shop at Nanyamba, binding and lamination, and only one key had been pressed. The fault was one word long, and the word was missing.

Trace the program below and state exactly what it prints, then explain why.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int choice = 0;
 7      cout << "Choose a service: ";
 8      cin >> choice;
 9      switch (choice)
10      {
11          case 1:
12              cout << "photocopying" << endl;
13              break;
14          case 2:
15              cout << "binding" << endl;
16          case 3:
17              cout << "lamination" << endl;
18              break;
19          default:
20              cout << "no such service" << endl;
21              break;
22      }
23      cout << "order taken" << endl;
24      return 0;
25  }
```

Typed at the program

```
2
```

Output

```
Choose a service: binding
lamination
order taken
```

Program 4.45: *The service menu from the printing shop. The key pressed was 2.*

Model Answer

It prints binding, then lamination, then order taken, because the case for 2 has no break at its end, so control runs out of it and straight through the body of case 3 before the break there finally leaves the switch.

Where control enters: the value 2 is compared against the labels in turn. Case 1 does not match, case 2 does, and control jumps to the first statement below that label. The body of case 1 is not run at all, which is the whole purpose of the labels.

What happens at the end of case 2: nothing ends it. A case is not a block with a boundary of its own, it is a labelled point inside one block, so having printed binding the program simply continues to the next statement in that block, which is the body of case 3.

Why case 3 runs although 3 was never keyed: the labels are inspected only during the search for a match. Once control is inside the switch they are ignored completely, so case 3 is not a test that has to be passed but a name for a place control has already arrived at.

Where control is	What happens there	Printed so far
case 1	the label does not match the value 2, so the search moves on	nothing yet
case 2	the label matches, and control jumps to the statement below it	binding
case 3	no break ended case 2, so control arrives without the label being tested	binding, lamination
the break in case 3	the switch ends and control passes the closing brace	binding, lamination
after the switch	the next statement of the program runs	binding, lamination, order taken

Table 4.12: *The path control takes when 2 is keyed. The default is never reached, and the label on case 3 is never tested at all.*

Where it stops, and what is never reached: at the break written at the end of case 3, which ends the switch and passes control to the statement after the closing brace. The default is therefore never reached, and order taken prints once, from the line below the switch.

What the compiler says about it: it offers a warning that the statement may fall through, which is a warning and not an error. The program is built, it runs to the end, it returns zero, and a customer is charged for a lamination nobody asked for.

The fix: break. One word at the end of case 2, which turns that case from a doorway into an alternative.

Read a switch by asking where control enters and where it leaves, because those are two separate questions and only the second one is answered by break.

The Master Key: *Control enters a switch at the label that matches and then ignores every label beneath it, so a case with no break runs the cases below it until a break or the closing brace stops it.*

BUILD Question 53

Say the alphabet aloud: five of its twenty six letters want one answer, the other twenty one want another. Writing the same two lines five times over is one way to say that.

Write a program that reads one character and reports whether it is a vowel, a consonant, a digit or none of these.

Read one character typed at the keyboard.

Report whether it is a vowel, a consonant, a digit, or none of these.

Use a switch for the vowels, with the five cases sharing one body.

Letters are typed in lower case.

Model Answer

Program 4.46 stacks the five vowel labels one above another with no statement between them, so a match on any of the five arrives at the same body, and everything the labels do not name is sorted under default by a short ladder of ranges.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      char ch = ' ';
 7      cout << "Enter one character: ";
 8      cin >> ch;
 9      switch (ch)
10      {
11          case 'a':
12          case 'e':
13          case 'i':
14          case 'o':
15          case 'u':
16              cout << ch << " is a vowel" << endl;
17              break;
18          default:
19              if (ch >= 'a' && ch <= 'z')
20              {
21                  cout << ch << " is a consonant" << endl;
22              }
23              else if (ch >= '0' && ch <= '9')
24              {
25                  cout << ch << " is a digit" << endl;
26              }
27              else
28              {
29                  cout << ch << " is neither" << endl;
30              }
31              break;
32      }
33      return 0;
34  }
```

Typed at the program

```
i
```

Output

```
Enter one character: i is a vowel
```

Program 4.46: *Five labels over one body, and a ladder under default for everything else. The character typed was the letter i.*

Why the stacked labels are not a fault: the first four cases carry no statements at all, so there is nothing for control to fall through. They are five names for one entry point, and the single break at the end of the shared body ends the case for all five of them at once.

What default is carrying: three answers rather than one. A ladder inside default sorts them by range, testing the letters first and the digits second, and leaving everything else to the closing else, so that a full stop or a space is reported rather than ignored.

The one design decision: values named individually go in the switch and values forming a range go in the ladder. Twenty one consonant labels and ten digit labels would compile and would be unreadable. Extending the program to capitals is five more labels above the same body and no new logic at all.

Stacked labels with nothing between them are the one use of fall-through that needs no comment beside it.

The Master Key: *Labels with no statements between them share one body, which is fall-through used deliberately, and a range of values belongs in a ladder rather than in a column of labels.*

BUILD Question 54

Two of the three offcuts on the bench at Nyasa measure five centimetres and the third measures ten. A program that looks for equal sides before asking whether they close at all reports isosceles.

Write a program that reads three side lengths and reports no triangle, equilateral, isosceles or scalene.

Read three whole side lengths typed at the keyboard.

Report no triangle where the three lengths cannot form one.

Otherwise report equilateral, isosceles or scalene.

The test for a triangle must be made before any classification.

Model Answer

Program 4.47 tests the triangle inequality first and classifies only what survives it, so the reading of five, five and ten is thrown out before the two equal sides are ever looked at.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int a = 0;
 7      int b = 0;
 8      int c = 0;
 9      cout << "Enter three side lengths: ";
10      cin >> a >> b >> c;
11      if (a + b <= c || a + c <= b || b + c <= a)
12      {
13          cout << "no triangle" << endl;
14      }
15      else if (a == b && b == c)
16      {
17          cout << "equilateral" << endl;
18      }
19      else if (a == b || b == c || a == c)
20      {
21          cout << "isosceles" << endl;
22      }
23      else
24      {
25          cout << "scalene" << endl;
26      }
27      return 0;
28  }
```

Typed at the program

```
5 5 10
```

Output

```
Enter three side lengths: no triangle
```

Program 4.47: *The inequality tested first, then the classification. The lengths typed were 5, 5 and 10.*

The guard, and why it comes first: three lengths form a triangle only when every pair of them adds to more than the third. Five and five make ten, which is not more than ten, so the first arm is taken and the program stops there. Classify first and the same three lengths are reported as isosceles.

Why the classification is ordered: an equilateral triangle has two equal sides as well as three, so it satisfies the isosceles test too. The test for three equal sides is therefore written above it, and since the first true test wins, the order of those two arms is not free.

Why one test does the work of two: the inequality written with less than or equal to also throws out a zero or a negative length, because either one makes one of the three sums fail. No separate check that the lengths are positive is needed, and none is written.

Ask whether the thing exists before asking what kind of thing it is, and the classification never has to defend itself.

The Master Key: *Test the triangle inequality before classifying, and place equilateral above isosceles, because the first true test wins and every equilateral triangle passes the isosceles test as well.*

HOT Question 55

The line that broke the fee program was one output statement, indented to match the line above it, under an if that had no braces. It printed a receipt for every customer, paid or unpaid.

Evaluate the practice of omitting braces around a single-statement body. Reach a judgement, with the condition that decides it.

Model Answer

Drop the braces only where the whole conditional is a guard written on one line, and brace everything else, because all three of the classic failures need a body that spans a line to do their damage.

What the language allows: the body of an if or an else is one statement, and a braced block counts as one statement. Braces are therefore optional wherever the body is single, and leaving them out is legal, ordinary and taught everywhere, which is why the practice needs judging rather than banning.

The honest case for leaving them out: vertical space, and with it readability. A guard that returns at once, written on one line, says the whole of what it does in one line. Give it three and a function of four such guards is twelve lines instead of four, with not one word of meaning added.

The dangling else: the first failure, and the one that cannot be argued with. Without braces an else attaches to the nearest unmatched if, which need not be the if the layout points at. Programs 4.41 and 4.5 differ only in their braces and print different things.

The statement added later: the second failure, and the commonest. A line added under an unbraced if, indented to agree with the line above it, is not part of the if at all and runs every time. The edit passes review because the indentation agrees with the reader and disagrees with the machine.

The stray semicolon: the third. A semicolon after the closing bracket ends the statement there and gives the if an empty body, so the line beneath it runs whatever the mark. Program 4.48 prints the pass message for a mark of 30. Braces do not stop the semicolon; they make what it did visible.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int mark = 0;
 7      cout << "Enter the mark: ";
 8      cin >> mark;
 9      if (mark >= 50);
10          cout << "candidate has passed" << endl;
11      cout << "marking finished" << endl;
12      return 0;
13  }
```

Typed at the program

```
30
```

Output

```
Enter the mark: candidate has passed
marking finished
```

Program 4.48: *One semicolon after the condition, and a mark of 30 that passes. The if is complete before the message is reached.*

What the compiler will and will not do: it warns about an empty body, about an ambiguous else, and about a line indented as though it were guarded when it is not. All three warnings have to be asked for, none of them is an error, and the program is built and shipped either way.

The judgement: brace every body that occupies a line of its own, without exception, and allow the short form only for a complete guard whose if, condition and statement are all in view at once. The condition that decides it is whether the whole conditional can be read without the eye moving down the page.

A pair of braces costs two characters and buys the guarantee that the next person to edit the line cannot get it wrong.

The Master Key: *Braces are optional and worth writing anyway: keep the short form only for a complete one-line guard, because the dangling else, the line added later and the stray semicolon all need a body spanning a line to bite.*

HOT Question 56

A menu at a bus ticketing desk had four choices when it was written and nineteen by its third year. Written as a ladder, choice nineteen is reached only after eighteen tests have failed.

A candidate argues that any switch can be rewritten as an else-if ladder, so switch is unnecessary. Evaluate this and reach a judgement.

Model Answer

The premise is true and the conclusion does not follow: every switch can be rewritten as a ladder, but the two say different things to a reader and offer different things to a compiler, and the switch fits exactly the case where both differences are worth having.

What is true in the claim: a switch tests one expression for equality against constants, and a ladder writes that easily. Any switch can be turned into a ladder mechanically, one case at a time, with no change to what the program does.

What the claim leaves out: the reverse does not hold. A ladder can test a range, compare two variables, test a string or a floating point value, and ask a different question on every arm. A switch does none of these, so the claim shows the switch is dispensable, not useless.

What a switch expresses that a ladder does not: that one value is under examination and the arms are alternatives on it, which a reader sees from the first line, before reading a single case. A ladder must be read to its end before anyone can be certain every arm tests the same variable.

The constraint on the types: the expression in a switch must be of integral or enumeration type. A band on a double, a menu keyed by a std::string, or any test on a range must be a ladder, and this is not a preference: Program 4.40 shows the compiler refusing it outright.

What a dense switch buys: where the labels are many and close together a compiler can build a jump table, in which the value indexes a table of addresses and control transfers in one step, so the cost does not grow with the number of cases. A ladder compares in turn, and its last arm costs the most.

What that is worth, honestly: very little where the program runs once for a keypress, and a good deal where it runs thousands of times a second. It is a reason to prefer a switch where both forms fit, never a reason to force a range test into one.

The judgement: use a switch when one integral or enumeration value is compared for equality against a set of named constants, and a ladder in every other case. What decides it is not speed and not taste, but whether every arm asks the same question of the same value.

Choose the statement that has the shape of the decision, and a reader will see the shape without being told it.

The Master Key: *Any switch can be written as a ladder and not the reverse, so use a switch only where one integral or enumeration value is tested for equality against named constants, and a ladder wherever ranges or strings appear.*

Iteration: while, do-while, for, nested loops and early exit

This section settles how a program repeats work. It gives the three loops, says which one of them is certain to run its body even once, and then answers three questions the common teaching leaves out altogether: what break and continue do inside a loop, how many times the inner body of a nested loop runs, and how a program refuses input it cannot use. One of its programs prints a greatest common factor it never worked out, and another goes on asking until the number it was promised actually arrives.

BINDER Question 57

The hand pump at the water kiosk in Mbarali gives water on the first stroke, before anybody looks at the tank. The gauge beside it is read first, and a zero means nobody pumps.

Explain the three loop constructs of C++ and state precisely how they differ.

Model Answer

C++ has three loops: while tests its condition before every pass, do while runs its body once and tests afterwards, and for gathers the initialisation, the condition and the update into one header; only the do while is certain to run its body, and the other two may run no passes at all.

The while loop: the keyword while, a condition in brackets, then one statement or a braced block. The condition is tested before every pass, the first one included, so a condition false at the start leaves the body unvisited.

The do while loop: the keyword do, then the body, then while with the condition and a closing semicolon. The body runs before the condition is looked at, so it always runs at least once. That semicolon is required and is easily forgotten.

The for loop and its header: the header carries three parts between two semicolons and the body is the fourth. Initialisation is performed once before anything else, the condition is tested before every pass, the body runs when the condition is true, and the update is performed at the end of every pass, before the next test.

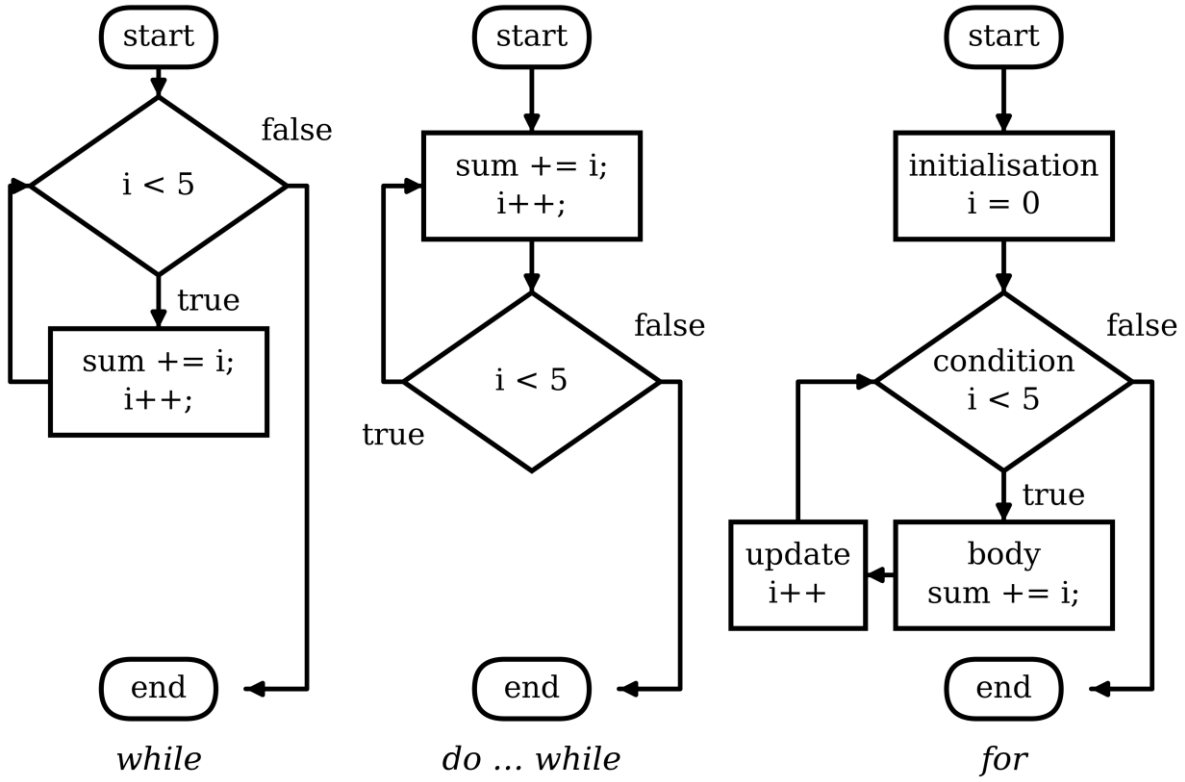


Figure 4.12: *The three loops compared. Only the do-while enters its body before testing anything, which is the whole of the difference between it and the while.*

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int limit = 5;
 7      int sum = 0;
 8      int i = 1;
 9      while (i <= limit)
10      {
11          sum += i;
12          i++;
13      }
14      cout << "while gave " << sum << endl;
15      sum = 0;
16      i = 1;
17      do
18      {
19          sum += i;
20          i++;
21      } while (i <= limit);
22      cout << "do while gave " << sum << endl;
23      sum = 0;
24      for (int k = 1; k <= limit; k++)
25      {
26          sum += k;
27      }
28      cout << "for gave " << sum << endl;
29      return 0;
30  }
```

Output

```
while gave 15
do while gave 15
for gave 15
```

Program 4.49: *The same total reached three ways. Each loop adds the numbers 1 to 5, and all three report 15.*

All three header parts are optional: the two semicolons are not. Leave out the initialisation and nothing is set up, leave out the update and nothing steps, and leave out the condition and it is taken to be true for ever.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int limit = 0;
 7      int i = 1;
 8      while (i <= limit)
 9      {
10          cout << "the while body ran" << endl;
11          i++;
12      }
13      i = 1;
14      do
15      {
16          cout << "the do while body ran" << endl;
17          i++;
18      } while (i <= limit);
19      for (int k = 1; k <= limit; k++)
20      {
21          cout << "the for body ran" << endl;
22      }
23      cout << "all three loops have finished" << endl;
24      return 0;
25  }
```

Output

```
the do while body ran
all three loops have finished
```

Program 4.50: *The same three loops given a limit of zero. Two of them print nothing whatever, and the third prints once before it discovers there was nothing to do.*

The one difference that matters: everything else is arrangement. A while and a for may be given a condition false from the start and then run no passes, which is often exactly right. A do while cannot, because it has acted before it asks.

Choose the loop by asking one question before any other: must the body run before the program knows anything at all?

The Master Key: *A while and a for test before the first pass and may run no passes at all, while a do while tests after the pass and therefore always runs its body at least once.*

BINDER Question 58

Two clerks counted sacks in the maize store at Ileje: one skipped a torn sack and carried on, the other found a rat and walked out. Both abandoned a sack, only one abandoned the count.

Explain break and continue inside a loop, and distinguish them.

Model Answer

break ends the loop at once and hands control to the first statement after it, while continue abandons the rest of the current pass only and goes on to the next one, which in a for loop still performs the update; each acts on the innermost loop that contains it and on nothing further out.

What break does: control leaves the loop immediately, whatever the condition would have said next. Nothing remaining in the body is run, no update is performed, and execution resumes at the first statement after the loop. The loop is finished for good and not merely paused.

What continue does: control abandons the rest of the body and goes back to the loop itself. In a while the next thing reached is the condition; in a for the update is performed first and the condition tested after it. Only this pass is over.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      for (int i = 1; i <= 8; i++)
 7      {
 8          if (i == 3)
 9          {
10              continue;
11          }
12          if (i == 6)
13          {
14              break;
15          }
16          cout << i << " ";
17      }
18      cout << endl;
19      cout << "the loop is over" << endl;
20      return 0;
21  }
```

Output

```
1 2 4 5
the loop is over
```

Program 4.51: *One loop carrying both jumps. The counter runs from 1 to 8 and four numbers reach the screen.*

Reading the two jumps: at 3 the continue skips the printing, the update still moves the counter to 4, and 3 is simply missing from the line. At 6 the break ends the loop outright, so 6, 7 and 8 never appear and control drops to the statement below.

Each acts on the innermost loop: a break inside an inner loop ends that inner loop and no other. The outer loop keeps its counter, takes its update and starts the inner loop again from the beginning, which is the part most often got wrong. Nothing leaves two loops at one word.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      for (int i = 1; i <= 3; i++)
 7      {
 8          for (int j = 1; j <= 4; j++)
 9          {
10              if (j == 3)
11              {
12                  break;
13              }
14              cout << i << j << " ";
15          }
16      }
17      cout << endl;
18      cout << "the outer loop still ran three times" << endl;
19      return 0;
20  }
```

Output

```
11 12 21 22 31 32
the outer loop still ran three times
```

Program 4.52: *A break inside the inner loop of a nest. The inner loop is cut short three separate times and the outer loop never notices.*

The same word inside a switch: break also ends a case of a switch, which the previous section covers. Where a switch sits inside a loop, a break in the switch leaves the switch and the loop carries on.

Ask what the word is leaving, not what it is skipping: continue leaves a pass and break leaves the loop.

The Master Key: *break ends the whole loop and continue ends only the current pass, still taking the update in a for, and each of them acts on the innermost loop that encloses it.*

BINDER Question 59

Fifteen. That is how many times the inner line runs when a loop counting to three carries a loop counting to five, and adding the two bounds gives eight.

Explain nested loops and state how many times the inner body runs.

Model Answer

A nested loop is a loop written inside the body of another, and the inner loop runs from its own beginning to its own end on every single pass of the outer one, so the inner body runs the product of the two counts and never their sum.

How control moves through a nest: the outer loop begins a pass, the inner loop runs to completion, and only then does the outer loop take its update and test again. The inner counter is set up afresh at the start of every outer pass, which is why the inner loop can run its full course over and over.

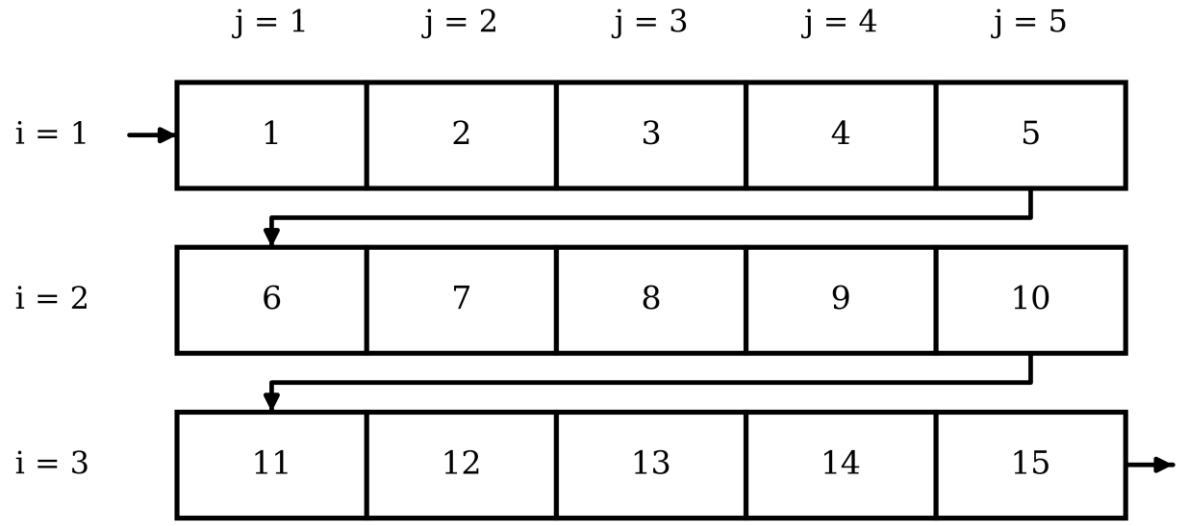


Figure 4.13: *A nested loop, visit by visit. The inner loop runs to completion for every single pass of the outer one, which is why the total is a product and not a sum.*

Counting the visits: three outer passes each carrying five inner passes give fifteen visits to the inner body. Multiply, never add. The outer body runs three times and the inner body fifteen, and those are two different counts with two different meanings.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int rows = 3;
 7      int columns = 5;
 8      int visits = 0;
 9      for (int i = 1; i <= rows; i++)
10      {
11          for (int j = 1; j <= columns; j++)
12          {
13              cout << i * j << " ";
14              visits++;
15          }
16          cout << endl;
17      }
18      cout << "the inner body ran " << visits << " times" << endl;
19      return 0;
20  }
```

Output

```
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
the inner body ran 15 times
```

Program 4.53: *A small multiplication table, three rows of five, with the inner body keeping its own tally.*

Reading the shape of the output: each printed row is one pass of the outer loop and each number in a row is one pass of the inner loop, so the shape on the screen is the shape of the loops. The tally of 15 at the foot is the inner body counting itself.

Why depth is expensive: three levels each counting to five give a hundred and twenty five visits, because depth multiplies the work instead of adding to it. A nest is therefore the first place to look when a program that used to answer at once begins to keep the user waiting.

Read a nest from the inside out: the inner bound tells you the length of a row and the outer bound tells you how many rows there are.

The Master Key: *The inner loop of a nest restarts on every outer pass, so the inner body runs the product of the two counts, and a nest three levels deep multiplies again.*

REAL Question 60

The register in the school library was ruled for four columns and the program filled five. One loop counted one to four, another zero to four, and whoever wrote them was certain they were the same loop.

Trace the three loops below and state exactly what each of them prints.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int n = 4;
 7      int passes = 0;
 8      cout << "one to n:";
 9      for (int i = 1; i <= n; i++)
10      {
11          cout << " " << i;
12          passes++;
13      }
14      cout << ", passes: " << passes << endl;
15      passes = 0;
16      cout << "zero to n minus one:";
17      for (int i = 0; i < n; i++)
18      {
19          cout << " " << i;
20          passes++;
21      }
22      cout << ", passes: " << passes << endl;
23      passes = 0;
24      cout << "the two idioms mixed:";
25      for (int i = 0; i <= n; i++)
26      {
27          cout << " " << i;
28          passes++;
29      }
30      cout << ", passes: " << passes << endl;
31      return 0;
32  }
```

Output

```
one to n: 1 2 3 4, passes: 4
zero to n minus one: 0 1 2 3, passes: 4
the two idioms mixed: 0 1 2 3 4, passes: 5
```

Program 4.54: *Three counting loops, each given the same value of n.*

Model Answer

The first loop prints 1 2 3 4 and the second prints 0 1 2 3, four passes each, while the third prints 0 1 2 3 4 in five passes, because it takes the starting value of the second idiom and the test of the first and so runs one pass too many.

What the header says, part by part: initialisation gives the counter its first value, and it is performed once. The condition is tested before every pass, so it decides whether the pass happens at all. The update is performed after the body of each pass and before the next test, so it never runs on a pass that did not happen.

Pass	Counter in the first loop	Is it at most 4	Counter in the second loop	Is it below 4	What the two bodies print
1	1	yes	0	yes	1 and then 0
2	2	yes	1	yes	2 and then 1
3	3	yes	2	yes	3 and then 2
4	4	yes	3	yes	4 and then 3
none	5	no	4	no	nothing, and both loops end

Table 4.13: *The two sound idioms traced side by side. Each makes four passes, each ends on a test that fails, and no two of the values they visit are the same.*

Why both sound loops make four passes: the count of passes is what the loop is for, and both idioms deliver four of them. The first visits the values 1 to 4 and the second visits 0 to 3, so the numbers on the screen differ while the amount of work does not. Neither is more correct than the other.

Where each counter stops: on the value that fails the test, not on the last value used. The first counter reaches 5 and the second reaches 4, and in both cases the update that produced that value is the last thing the loop does. The failing test is a pass that did not happen.

The third loop and its extra pass: it starts at 0 like the second and tests at most 4 like the first, so it visits 0, 1, 2, 3 and 4, which is five values and five passes. Nothing in it is illegal and nothing warns about it. This is the off-by-one error, and it is one character wide.

Checking any counting loop in two steps: name the first value the counter takes and the last value that passes the test, then count the values between them inclusive. If that count is not the number of passes wanted, the header is wrong, and the boundary is where to look first.

Fix the two idioms in mind as whole shapes and never assemble a header out of halves of both.

The Master Key: *A loop from 1 while at most n and a loop from 0 while below n both make n passes over different values, and mixing the start of one with the test of the other makes n plus one.*

REAL Question 61

The printout stapled to the front of the assignment said the greatest common factor was zero. It had compiled without a complaint and run to its last line, and zero is a common factor of nothing.

The program below is meant to find the greatest common factor of two numbers. It compiles cleanly, runs, and for some inputs prints a value it never computed. Explain why, and correct it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int first = 0;
 7      int second = 0;
 8      cout << "Enter two numbers: ";
 9      cin >> first >> second;
10      int limit = first;
11      if (second < first)
12      {
13          limit = second;
14      }
15      int gcf;
16      for (int i = 2; i <= limit; i++)
17      {
18          if (first % i == 0 && second % i == 0)
19          {
20              gcf = i;
21          }
22      }
23      cout << "The greatest common factor is " << gcf << endl;
24      return 0;
25  }
```

Typed at the program

```
12 1
```

Output

```
Enter two numbers: The greatest common factor is 0
```

Program 4.55: *The greatest common factor as the assignment was written. The numbers typed were 12 and 1.*

Model Answer

gcf is declared without a value and is assigned only inside the loop body, so on any input where the loop makes no passes at all, which is what happens as soon as either number is 1, the printing line reads a variable that was never written; that is undefined behaviour and not simply an unpredictable answer.

Why the loop makes no passes: limit is set to the smaller of the two numbers, which is 1, and the counter starts at 2. The condition asks whether 2 is at most 1, it is false at the very first test, and the body is never entered.

What is then printed: whatever that piece of memory happened to hold. A local variable is not cleared when it is created, so gcf holds the leftovers of whatever used that part of the stack before. On the machine this book was built on the leftover was zero, on forty runs out of forty.

Why that is worse than a wrong answer: reading an object before it has been given a value is undefined behaviour, so the language promises nothing at all about what follows. It is no guarantee of zero, or of the same value twice. Another compiler, another machine or the optimiser turned on may each differ, and this program built here with optimisation does warn that gcf may be used before it is set.

Why nothing was reported at the ordinary settings: the analysis that finds this fault is done during optimisation, and at the settings a student normally builds with the compiler does not attempt it. Every line is legal C++, the loop is well formed and the declaration is valid, so the build is silent and the program runs.

The correction: give gcf a value before the loop, and give it the right one. Every pair of whole numbers has 1 as a common factor, so 1 is the correct starting answer and any larger factor the loop finds replaces it. Program 4.56 differs in four characters and prints 1.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int first = 0;
 7      int second = 0;
 8      cout << "Enter two numbers: ";
 9      cin >> first >> second;
10      int limit = first;
11      if (second < first)
12      {
13          limit = second;
14      }
15      int gcf = 1;
16      for (int i = 2; i <= limit; i++)
17      {
18          if (first % i == 0 && second % i == 0)
19          {
20              gcf = i;
21          }
22      }
23      cout << "The greatest common factor is " << gcf << endl;
24      return 0;
25  }
```

Typed at the program

```
12 1
```

Output

```
Enter two numbers: The greatest common factor is 1
```

Program 4.56: *The result initialised before the loop rather than inside it. The same two numbers now produce the correct answer.*

The general rule this leaves: any variable a loop is meant to fill must be given a sound value before the loop starts, because a loop that may run no passes will leave it exactly as it found it. The starting value is part of the algorithm and not a formality.

A loop that can run no passes is not an unusual case; it is the case the initialisation of the result exists to answer.

The Master Key: *Initialise a result before the loop that computes it, because a loop making no passes leaves it untouched, and reading an uninitialised local is undefined behaviour rather than a merely unpredictable value.*

BUILD Question 62

The tally sheet at the fish landing is a column of weights with a line ruled under the last one. Nobody knows at dawn how many boats will come in; the ruled line says the day is finished.

Write a program that reads numbers until the user enters zero, then reports how many were entered, their total and their mean.

Read whole numbers typed at the keyboard until the user enters 0.

The 0 marks the end and is not one of the numbers.

Print how many numbers were entered, their total and their mean.

The program must not divide by zero if the first entry is 0.

Model Answer

Program 4.57 is a sentinel controlled loop: it reads one number before the loop, tests that number as the condition, and reads the next one as the last act of the body, so the sentinel is examined before it can be counted and never joins the total.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int value = 0;
 7      int count = 0;
 8      int total = 0;
 9      cout << "Enter numbers, and 0 to finish:" << endl;
10      cin >> value;
11      while (value != 0)
12      {
13          count++;
14          total += value;
15          cin >> value;
16      }
17      cout << "Numbers entered: " << count << endl;
18      cout << "Total: " << total << endl;
19      if (count > 0)
20      {
21          cout << "Mean: " << static_cast<double>(total) / count << endl;
22      }
23      else
24      {
25          cout << "Mean: none, because nothing was entered" << endl;
26      }
27      return 0;
28  }
```

Typed at the program

```
14 22 9 36 0
```

Output

```
Enter numbers, and 0 to finish:
Numbers entered: 4
Total: 81
Mean: 20.25
```

Program 4.57: *A sentinel controlled loop. Four weights were typed, then the 0 that ends the sheet.*

Why one read comes before the loop: the condition tests value, so value must already hold something when the condition is first reached. The read before the loop supplies it and the read at the foot of the body supplies every later one, which is the shape that keeps the sentinel out of the count and out of the total.

The guard on the mean: a sentinel loop is entitled to run no passes, and it does exactly that when 0 is the first thing typed. Dividing the total by a count of zero would then be an error the program need not commit, so the division is written under a test that the count is above zero.

The one design decision: the mean is worked out in floating point rather than whole numbers. Four weights totalling 81 have a mean of 20.25, and an integer division would have reported 20 and lost the quarter without saying so.

A sentinel is a value the data can never take, examined before it is used, and read in two places so that it is counted in neither.

The Master Key: *A sentinel controlled loop reads once before the loop and once at the foot of the body, so the value that ends the data is tested and then discarded rather than counted.*

BUILD Question 63

The word seven, spelled out in letters, was typed into the dispensary program at Mlele where a digit was wanted. One wrong keystroke stops the stream, not the program, and every later read is answered with silence.

Write a program that keeps asking until a positive integer is entered, rejecting everything else.

Read a positive integer typed at the keyboard.

Reject anything that is not a positive integer and ask again.

Keep asking until a positive integer is entered.

Then print the factorial of that number.

Model Answer

Program 4.58 puts the reading itself inside the loop condition, so a read that fails and a number that is not positive are both refused by the same test, and the body then clears the failed state, throws away the characters that caused it and asks again.

```
 1  #include <iostream>
 2  #include <limits>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      int n = 0;
 8      cout << "Enter a positive integer:" << endl;
 9      while (!(cin >> n) || n < 1)
10      {
11          cin.clear();
12          cin.ignore(numeric_limits<streamsize>::max(), '\n');
13          cout << "Not a positive integer. Try again:" << endl;
14      }
15      long long factorial = 1;
16      for (int i = 2; i <= n; i++)
17      {
18          factorial *= i;
19      }
20      cout << "The factorial of " << n << " is " << factorial << endl;
21      return 0;
22  }
```

Typed at the program

```
seven
-4
6
```

Output

```
Enter a positive integer:
Not a positive integer. Try again:
Not a positive integer. Try again:
The factorial of 6 is 720
```

Program 4.58: *Input validated in the loop condition. The word seven was typed, then minus four, then 6.*

What the condition is testing: two things at once. The read is attempted inside the brackets and gives a value that is false when it fails, so the first half catches a word offered where a number was wanted and the second half catches a number that is not positive.

Why clearing is not enough on its own: cin.clear puts the stream back into a working state but leaves the offending characters exactly where they were, so the next read meets them again and fails again. The call to ignore discards everything up to and including the end of the line, which is what actually removes them.

The one design decision: the loop guards the whole of the rest of the program, so nothing below it needs to test n again. By the time control reaches the factorial, n is a positive whole number and the counting loop below can be written without a single defence of its own.

Validation belongs in a loop because one refusal is not enough; the user may be wrong twice.

The Master Key: *Read inside the loop condition, and on failure clear the stream and discard the rest of the line before asking again, or the same bad characters will fail every read that follows.*

BUILD Question 64

The order book on the counter at Nsimbo is ruled into a triangle of ticks, one against the first order, two against the second, three against the third. No pair of fixed bounds can draw it.

Write a program that reads a height and prints a triangle of stars of that height.

Read the height of the triangle from the keyboard.

Print that many rows of stars.

Row one carries one star, row two carries two, and so on.

A height of zero or less must print no rows at all.

Model Answer

Program 4.59 nests one loop inside another and writes the outer counter into the inner condition, so the inner loop runs once on the first row, twice on the second and row times on row number row, which is what makes the output a triangle instead of a rectangle.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int height = 0;
 7      cout << "Enter the height of the triangle:" << endl;
 8      cin >> height;
 9      for (int row = 1; row <= height; row++)
10      {
11          for (int star = 1; star <= row; star++)
12          {
13              cout << "*";
14          }
15          cout << endl;
16      }
17      cout << "Rows printed: " << height << endl;
18      return 0;
19  }
```

Typed at the program

```
5
```

Output

```
Enter the height of the triangle:
*
**
***
****
*****
Rows printed: 5
```

Program 4.59: *The inner bound taken from the outer counter. The height typed was 5, and the five rows hold 15 stars between them.*

Where the triangle comes from: the inner condition is written against row rather than against a fixed number, so the length of a row is decided by the row it is on. Replace row with the height and the same nest prints a rectangle of 25 stars, which is the difference one name makes.

Why the line break sits where it does: the statement that ends a line belongs to the outer body, after the inner loop has finished and before the outer update. Put it inside the inner loop and every star takes a line of its own; leave it out and all 15 stars run together on one line.

The one design decision: no separate guard is written against a height of zero or less, because the outer condition already is one. The test is made before the first pass, so a height of zero fails it at once and the program prints no rows, which is what the specification asked for.

When the length of a row changes, the inner bound is not a constant; it is the outer counter.

The Master Key: *A nested loop whose inner bound is the outer counter prints a triangle rather than a rectangle, and the outer condition doubles as the guard against a height of zero.*

HOT Question 65

Anything a for loop does, a while loop can do as well, so why learn two of them? The claim is made in every classroom and half of it is exactly right.

Evaluate the claim that any for loop can be written as a while loop, so for is unnecessary. Reach a judgement, with the condition that decides it.

Model Answer

The premise is true and the conclusion does not follow: every for can be rewritten as a while by moving the initialisation above the loop and the update to the foot of the body, but that scatters across three places what the header held in one, widens the life of the counter, and changes what continue means.

What is true in the claim: the rewriting is real and mechanical. Put the initialisation on the line above, write the condition in the brackets of the while, and put the update as the last statement of the body. The result runs identically.

What the header gathers in one place: where the counter starts, when the loop stops and how the counter moves, on one line. A reader deciding whether a loop terminates reads that line and nothing else. The while version answers the same three questions from three separate places.

What is easiest to lose in the rewriting: the update. It is now an ordinary statement in the body, so a branch that skips the foot of the body can leave the counter unchanged, and a loop whose counter never moves never ends.

The change continue makes: in a for, continue takes the update on its way past, so the loop still advances. In the rewritten while the update sits at the foot of the body and continue jumps clean over it, so the counter is stuck and the loop runs for ever. The two forms are not interchangeable here.

The life of the counter: a counter declared in a for header exists only for the loop and cannot be named after it, so the same name is free for the next loop. A counter declared above a while stays visible for the rest of the enclosing block.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int total = 0;
 7      for (int i = 1; i <= 5; i++)
 8      {
 9          total += i;
10      }
11      cout << "the total is " << total << endl;
12      cout << "the counter stopped at " << i << endl;
13      return 0;
14  }
```

Compiler says

```
error: ‘i’ was not declared in this scope
```

Program 4.60: *The counter of a for loop named on the line after the loop. The build stops there, and the arithmetic above it is never reached.*

Where a while genuinely reads better: wherever there is no counter and the number of passes cannot be known in advance. Program 4.57 stops on a sentinel and Program 4.58 stops when the user types something usable, and neither has an initialisation or an update to put in a header. Forcing them into one leaves it mostly empty.

The judgement: use a for where the passes are governed by a counter whose start, limit and step are all known when the loop is written, and a while where continuation depends on what the body discovers. The condition that decides it is whether those three parts exist at all.

A language keeps two shapes for one job when they say different things to the reader.

The Master Key: *Every for can be written as a while, but the header holds the start, the test and the step where a reader sees all three at once, so use a for for counted passes and a while for everything else.*

HOT Question 66

A student at Mpimbwe, taught that a loop must have one way in and one way out, would not write break. His searching loop carried a flag and went on turning over a list it had finished.

A candidate argues that break and continue should never be used because they make a loop harder to follow. Evaluate this and reach a judgement.

Model Answer

The argument is sound in its reasoning and too strong in its conclusion: a break does hide an exit the loop header never mentions, but the flag that replaces it hides the same exit inside a variable, and in a search that has found what it wanted, break is the honest statement of what happened.

The case against, put fairly: structured programming asks that a block have one entry and one exit, so that a reader who has understood the header knows when the loop ends. break and continue add ways out the header does not mention.

What the alternative actually is: a flag, which is a variable set inside the body and tested in the condition. It keeps the single exit in the header, and it is the form every argument against break assumes without writing out.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int target = 91;
 7      int firstFactor = 0;
 8      for (int i = 2; i <= target; i++)
 9      {
10          if (target % i == 0)
11          {
12              firstFactor = i;
13              break;
14          }
15      }
16      cout << "found with break: " << firstFactor << endl;
17      int flagged = 0;
18      bool searching = true;
19      int j = 2;
20      while (searching && j <= target)
21      {
22          if (target % j == 0)
23          {
24              flagged = j;
25              searching = false;
26          }
27          j++;
28      }
29      cout << "found with a flag: " << flagged << endl;
30      return 0;
31  }
```

Output

```
found with break: 7
found with a flag: 7
```

Program 4.61: *The same search written twice, once ending with break and once ending with a flag. Both report the smallest factor of 91 above 1.*

What the flag costs: one more variable, a condition of two parts instead of one, and a counter declared above the loop that outlives it. In any longer body every statement after the discovery still runs unless another if is wrapped round it, which is the nesting the flag was to spare us.

The search that has nothing left to do: this is where break earns its place. The first factor of 91 is 7, so the loop has its answer after six passes out of a possible ninety, and break says plainly that the purpose of the loop is complete.

Where continue helps and where it bites: a continue at the top of a body, skipping a pass before any work is done, flattens a body that would otherwise sit inside one large if. Written lower down, or in a while whose counter is stepped at the foot, it jumps over the step and the loop never ends.

What the argument gets right: a break buried deep in a long body, or a body carrying several of them, is genuinely harder to follow than a flag, because the reader must hold every exit in mind at once. The objection is about distance and number.

The judgement: use break where the work of the loop is finished and nothing after it applies, and use continue only as a guard at the head of the body. What decides it is visibility: where every way out can be seen from the header without leaving the screen, the jump is clearer than the flag, and otherwise the flag is.

A rule made to protect the reader stops being a rule when it costs the reader more than it saves.

The Master Key: *break and continue are worth using exactly where every exit stays visible from the loop header, because a flag hides the same exit in a variable and pays for it in an extra condition and extra nesting.*

Functions: parameters, overloading, default arguments and recursion

This section is where a program stops being one long block. It defines the function, separates the declaration from the definition and the parameter from the argument, and settles what passing by value and passing by reference each do to the caller's variable. It then supplies two things the common teaching leaves out altogether, although exercises set at this level demand both: function overloading, in which one name serves several parameter lists, and recursion, in which a function calls itself and is stopped by a base case. One program here fails to compile because the function it calls does not exist, and the last one counts its own calls, so the price of a naive recursion arrives as a number rather than as an adjective.

BINDER Question 67

Pinned above the till at a mobile money kiosk in Kakonko is a card naming what the agent must be handed and what comes back. It went up months before anybody was hired.

Explain what a function is, and distinguish a declaration from a definition and a parameter from an argument.

Model Answer

A function is a named block of statements written once and called by name, given values to work on and asked for a value back; its declaration, or prototype, announces the return type, the name and the parameter types and ends at a semicolon, while its definition repeats that header and supplies the braced body.

What a function is for: one piece of work, given a name. A call passes control to the body, the body runs as far as its return, and control comes back to the calling statement carrying the returned value.

Declaration against definition: the declaration promises that such a function exists somewhere, and parameter names may be left out of it because only the types are promised. The definition is the body, and a function may be declared many times but defined exactly once.

Why the prototype lets the definition come after main: the compiler reads downwards and must know the shape of a function before it meets a call to it. Lines 4 and 5 supply that shape, so both definitions may sit below main.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int totalMarks(int paperOne, int paperTwo);
 5  void announce(int total);
 6
 7  int main()
 8  {
 9      int first = 34;
10      int second = 41;
11      int sum = totalMarks(first, second);
12      announce(sum);
13      return 0;
14  }
15
16  int totalMarks(int paperOne, int paperTwo)
17  {
18      cout << "parameters hold " << paperOne << " and "
19           << paperTwo << endl;
20      return paperOne + paperTwo;
21  }
22
23  void announce(int total)
24  {
25      cout << "the total is " << total << endl;
26  }
```

Output

```
parameters hold 34 and 41
the total is 75
```

Program 4.62: *One prototype, one definition and one call each. Both functions are defined after main, which is legal only because lines 4 and 5 declared them first.*

Formal parameter and actual argument: the parameter is the name in the header, paperOne here, and the argument is the value in the call, 34 here. In a call by value the copying runs from the argument into the parameter and never back; in a call by reference nothing is copied at all, and the parameter is bound to the caller's variable. A common error reverses that direction and expects the caller to see whatever the function assigned to its parameter.

The return type, and void: the type written before the name is the type of the value handed back. A function that hands nothing back is declared void, as announce is.

Read the header first: it says what goes in and what comes back.

The Master Key: *A declaration promises a name, a return type and a list of parameter types, a definition supplies the body, and the parameter lives inside the function while the argument is what the caller hands over.*

BINDER Question 68

Five went into the function and five came out, although the line inside it plainly wrote 99. Nothing was wrong with the compiler, and nothing was wrong with the arithmetic.

Distinguish passing by value from passing by reference, and state when each is appropriate.

Model Answer

Passing by value copies the argument into a parameter that is a separate variable with an address of its own, so writing to it cannot reach the caller, while passing by reference binds the parameter to the caller's variable itself, so writing to it is writing to the caller; small values a function only reads go by value, and anything the caller must see changed goes by reference.

What passing by value does: a fresh variable is created for the call, the argument is copied into it, and it dies when the call returns. Program 4.63 writes 99 into that copy and prints it, and x in main is still 5 afterwards.

What passing by reference does: the ampersand in the parameter list, as in int &n, makes the parameter a reference. Nothing is copied and no new object is made; the parameter is a second name for the caller's variable, fixed to it for the whole call.

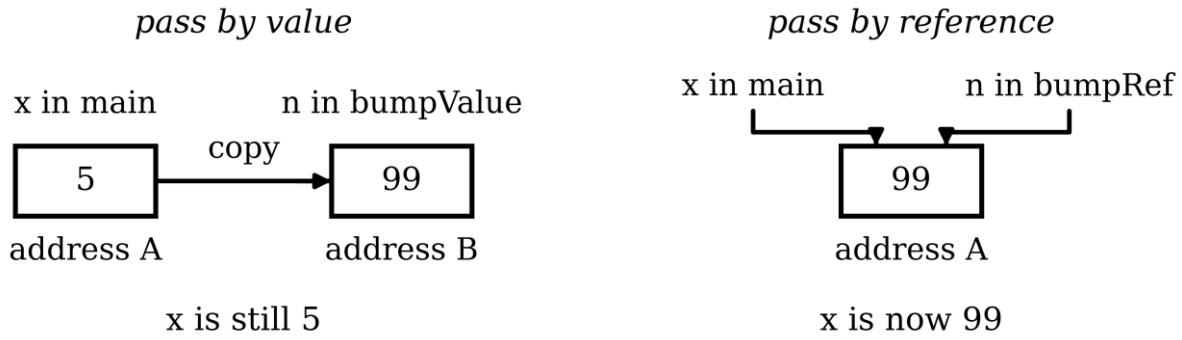


Figure 4.14: *Passing by value and by reference. The copy has an address of its own, so writing to it cannot reach the caller; the reference is a second name for one address, so writing to it does.*

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  void bumpValue(int n)
 5  {
 6      n = 99;
 7      cout << "inside bumpValue n is " << n << endl;
 8  }
 9
10  void bumpRef(int &n)
11  {
12      n = 99;
13      cout << "inside bumpRef n is " << n << endl;
14  }
15
16  void report(int copy, int &alias, int &twin)
17  {
18      cout << "copy at the address of x: "
19           << (&copy == &alias) << endl;
20      cout << "reference at the address of x: "
21           << (&twin == &alias) << endl;
22  }
23
24  int main()
25  {
26      int x = 5;
27      bumpValue(x);
28      cout << "after bumpValue x is " << x << endl;
29      bumpRef(x);
30      cout << "after bumpRef x is " << x << endl;
31      report(x, x, x);
32      return 0;
33  }
```

Output

```
inside bumpValue n is 99
after bumpValue x is 5
inside bumpRef n is 99
after bumpRef x is 99
copy at the address of x: 0
reference at the address of x: 1
```

Program 4.63: *The same variable passed both ways, and the addresses compared rather than described. report is handed x three times over, once by value and twice by reference.*

Reading the last two lines of output: 0 stands for false and 1 for true. The copy does not share the address of x, so the first comparison prints 0; the two references share one address, and it is the address of x, so the second prints 1.

A const reference for a large object: a large object passed by value is copied byte for byte on every call. Writing const before the reference type gives the saving without the licence: no copy is made, and the compiler refuses any write to it.

Choosing between them: small and read only goes by value, because the copy costs almost nothing and the caller is safe. Anything the caller must see changed goes by reference, and anything large that must not change goes by const reference.

Ask what the function is holding: a copy of the value, or the caller's variable under another name.

The Master Key: *A value parameter is a separate variable at its own address and a reference parameter is a second name for the caller's variable, so only the reference can change what the caller holds.*

BINDER Question 69

A student at Kyerwa gave three functions the same name, and the compiler said nothing. A fourth function with that name broke the build, and the only thing different about it was the type it gave back.

Explain function overloading and state the rule that decides whether two functions are a valid overload.

Model Answer

Function overloading is the writing of two or more functions with one name in one scope, and the compiler decides which of them a call means from the number and the types of the arguments; two functions are a valid overload only when their parameter lists differ in number or in type, and two that differ only in return type are an error at the second declaration.

What the compiler matches on: the signature, which is the name together with the number, the order and the types of the parameters. The return type is no part of it. Each call is matched against every function of that name, and the one whose parameter list fits is compiled in.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  const double PI = 3.14159;
 5
 6  int area(int side)
 7  {
 8      return side * side;
 9  }
10
11  int area(int length, int width)
12  {
13      return length * width;
14  }
15
16  double area(double radius)
17  {
18      return PI * radius * radius;
19  }
20
21  int main()
22  {
23      cout << "square of side 5: " << area(5) << endl;
24      cout << "rectangle 4 by 6: " << area(4, 6) << endl;
25      cout << "circle of radius 2: " << area(2.0) << endl;
26      return 0;
27  }
```

Output

```
square of side 5: 25
rectangle 4 by 6: 24
circle of radius 2: 12.5664
```

Program 4.64: *Three functions called area in one program. The first two differ in the number of parameters, the first and third in the type of one parameter, and every call is settled while the program is being compiled.*

Reading the three calls: area(5) has one whole number argument and reaches the first, printing 25. area(4, 6) has two and reaches the second, printing 24. area(2.0) has one argument written with a point, so its type is double and the third is chosen, printing 12.5664.

Why the return type alone cannot decide: a call carries its arguments and nothing else, so nothing in it says which return type was wanted and no rule could choose. The language settles the matter by refusing the second declaration rather than the call.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int average(int first, int second)
 5  {
 6      return (first + second) / 2;
 7  }
 8
 9  double average(int first, int second)
10  {
11      return (first + second) / 2.0;
12  }
13
14  int main()
15  {
16      cout << average(7, 4) << endl;
17      return 0;
18  }
```

Program 4.65: *Two functions differing only in what they return. The compiler rejects the second declaration at line 9 and never reaches the call.*

One name, several behaviours, chosen early: this is compile-time polymorphism, meaning that the choice is fixed before the program runs and the finished program holds three separate functions. The run-time kind, decided during execution, belongs to Section 12.

What overloading is worth: one idea keeps one name. A reader who knows what area means needs no second word for a rectangle, and a new shape adds a function rather than editing an old one.

Change the parameter list and you have an overload; change the return type alone and you have an error.

The Master Key: *Overloads are told apart by the number and types of their parameters, never by their return type, and the compiler picks one at every call before the program runs.*

BINDER Question 70

Two hundred copies, ordinary paper, no binding. The clerk at the printing shop in Missenyi writes down the number and nothing else, because the other two answers are the same for nearly everyone.

Explain default arguments and state the rules that govern them.

Model Answer

A default argument is a value written into a parameter so that a caller may leave that argument out, and three rules govern it: the default is written in the declaration if there is one and never repeated in the definition, defaulted parameters must be the trailing ones, and a call may omit any tail of them but never leave a gap in the middle.

Where the default is written: in the declaration, which is line 4 of Program 4.66. The definition at line 14 repeats the header without them. Writing the same default in both places is an error, and the rule is simply that each default is stated once.

Why they must be the trailing parameters: arguments are matched to parameters by position, first to first and second to second. A defaulted parameter in the middle could never be skipped, because the next argument written would slide into its place, so the language requires that everything after a defaulted parameter is defaulted too.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int jobCost(int copies, int pricePerCopy = 100, int binding = 0);
 5
 6  int main()
 7  {
 8      cout << "plain copies: " << jobCost(200) << endl;
 9      cout << "thick paper: " << jobCost(200, 150) << endl;
10      cout << "thick and bound: " << jobCost(200, 150, 2500) << endl;
11      return 0;
12  }
13
14  int jobCost(int copies, int pricePerCopy, int binding)
15  {
16      return copies * pricePerCopy + binding;
17  }
```

Output

```
plain copies: 20000
thick paper: 30000
thick and bound: 32500
```

Program 4.66: *One function, three calls. The same jobCost is called with one argument, then two, then three, and it charges 100 a copy and no binding unless it is told otherwise.*

What a call may leave out: any tail of the defaulted arguments. jobCost(200) takes both defaults and prints 20000, jobCost(200, 150) takes one and prints 30000, and jobCost(200, 150, 2500) supplies all three and prints 32500. There is no way to give the third and omit the second.

A default is not an overload: one function is compiled, not three, and it gains several call shapes rather than several bodies. Where a default and an overload of the same name would both fit a call, the compiler cannot choose and reports the call as ambiguous, so the two are best not mixed on one name.

Put the arguments a caller nearly always wants to leave out at the end of the list, and give them their values once.

The Master Key: *Default arguments are written once, in the declaration, on the trailing parameters only, and a call may drop any tail of them but never a gap in the middle.*

REAL Question 71

A table of built-in functions, copied by hand into a great many exercise books, lists isPrime beside sqrt and pow and gives cmath as its header. The student who believed it never got a program at all.

A student is told that isPrime is a standard library function available from cmath, and writes the program below. Explain what happens and correct it.

```
 1  #include <iostream>
 2  #include <cmath>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      int number = 0;
 8      cout << "Enter a whole number: ";
 9      cin >> number;
10      if (isPrime(number))
11      {
12          cout << number << " is prime" << endl;
13      }
14      else
15      {
16          cout << number << " is not prime" << endl;
17      }
18      return 0;
19  }
```

Program 4.67: *The program as the student wrote it. The header is included and the call is made.*

Model Answer

Nothing runs, because there is no isPrime in cmath or anywhere else in the standard library: the compiler reaches the call at line 10, finds a name it has never been given, reports that isPrime was not declared in this scope, and stops, so no program is produced at all; the correction is to write the function, which takes six statements.

What the header actually holds: cmath declares sqrt, pow, fabs, floor and their relatives, which are arithmetic operations on numbers. Whether a number is prime is a question about its divisors, decided by a search, and no standard header answers it. Including cmath here changes nothing and the program never uses it.

Exactly what the compiler does about it: it reads downwards, meets isPrime at line 10 with no declaration anywhere above, and reports that isPrime was not declared in this scope. It names the line and the identifier. Compilation ends there, so there is no object file, no linking and nothing to run.

Why this is not a fault found later: a function declared and never defined passes the compiler and fails at the linking step instead, with a different message about an undefined reference. Here the name is not declared at all, so the compiler settles it first. The two stages, and the errors that belong to each, are Section 2.

The correction: write the function. Program 4.68 refuses anything below 2, then tries divisors from 2 upwards while the square of the divisor has not passed the number, returns false at the first divisor that divides exactly, and returns true if none does. That is six statements, and the include of cmath is deleted with them, since nothing now needs it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  bool isPrime(int n)
 5  {
 6      if (n < 2)
 7      {
 8          return false;
 9      }
10      for (int d = 2; d * d <= n; d++)
11      {
12          if (n % d == 0)
13          {
14              return false;
15          }
16      }
17      return true;
18  }
19
20  int main()
21  {
22      int number = 0;
23      cout << "Enter a whole number: ";
24      cin >> number;
25      if (isPrime(number))
26      {
27          cout << number << " is prime" << endl;
28      }
29      else
30      {
31          cout << number << " is not prime" << endl;
32      }
33      return 0;
34  }
```

Typed at the program

```
97
```

Output

```
Enter a whole number: 97 is prime
```

Program 4.68: *The correction, with isPrime written out. The number 97 was typed at the prompt.*

Why the loop may stop so early: if a number has a divisor larger than its square root it has a matching one smaller, so a search that reaches the square root has settled the question. Writing the test as d * d <= n keeps the arithmetic in whole numbers, and for 97 the largest divisor tried is 9.

The lesson under the error: a list of built-in functions is worth no more than the compiler that accepts it. Where a name is claimed for the library and the build stops on it, the name is not there, and writing the function yourself is usually a matter of a few lines.

A function that does not exist is caught at the call, named and numbered, before a single instruction is generated.

The Master Key: *There is no isPrime in the standard library; the compiler reports that the name was not declared in this scope, at the line of the call, and the repair is to write the six statements yourself.*

REAL Question 72

The printout pinned to the wall of the computer room at Chamwino counted downwards, four, three, two and one, and then counted upwards again. No line anywhere in that program counts upwards.

Trace the recursion below and state exactly what it prints, and in what order.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int factorial(int n)
 5  {
 6      cout << "call with n = " << n << endl;
 7      if (n == 1)
 8      {
 9          cout << "base case reached, returning 1" << endl;
10          return 1;
11      }
12      int smaller = factorial(n - 1);
13      int result = n * smaller;
14      cout << "returning " << result << " for n = " << n << endl;
15      return result;
16  }
17
18  int main()
19  {
20      int answer = factorial(4);
21      cout << "answer: " << answer << endl;
22      return 0;
23  }
```

Output

```
call with n = 4
call with n = 3
call with n = 2
call with n = 1
base case reached, returning 1
returning 2 for n = 2
returning 6 for n = 3
returning 24 for n = 4
answer: 24
```

Program 4.69: *A recursive factorial that reports itself twice, once as it is entered and once as it returns.*

Model Answer

It prints nine lines: four call lines counting down from n equal to 4 to n equal to 1, the base case line, three returning lines carrying 2, 6 and 24 in that order, and the answer 24 from main; the call lines appear on the way down because each call prints before making the next, and the returning lines appear in the reverse order because each call must wait for the call it made.

The way down: each call prints its own n at line 6 and then reaches line 12, which calls factorial again with n minus 1 before any multiplication is attempted. Four calls are therefore made and none of them has multiplied anything yet.

The base case, which stops it: at n equal to 1 the test at line 7 succeeds, the function returns 1 and makes no further call. Without that case the argument would fall past zero and the calls would go on until the stack was exhausted, which is a crash and not an error message.

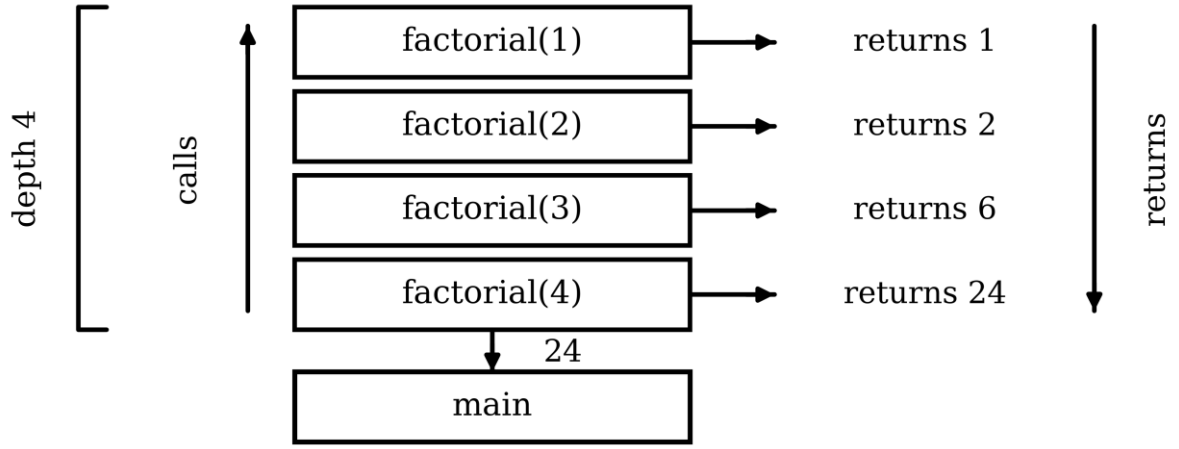


Figure 4.15: *The call stack for a factorial of four. Each call waits for the one it made, which is why the multiplications happen on the way back and not on the way down.*

The way back: line 13 cannot run until line 12 has produced a value, so the innermost call finishes first. The multiplications come out 2, then 6, then 24, which is the reverse of the order in which the calls were made, and each of them belongs to a call that had been waiting.

Call	**n**	**Depth**	**Returns**
factorial(4)	4	1	24
factorial(3)	3	2	6
factorial(2)	2	3	2
factorial(1)	1	4	1

Table 4.14: *The four calls, in the order they were made. The depth grows downwards and the returned values are produced upwards, the last call being the first to return.*

Reading the trace in the right direction: the table is made top to bottom and read bottom to top. Each row keeps its own n and its own smaller, because every call has its own copies of both, and the deepest row is the only one that returns without waiting for anything.

What the depth costs: an argument of 4 reaches a depth of 4, and one stack frame is held open for each of those calls until the last of them returns. Depth rises with the argument, so a recursion whose depth follows the size of the data is a recursion that can run out of stack.

Anything a recursive function does after its recursive call happens in the reverse of the order the calls were made.

The Master Key: *A recursive call prints and computes on the way down in the order of calling, and everything written after the recursive call happens on the way back, in the reverse of that order.*

BUILD Question 73

There is a definition of factorial that uses the word factorial: n times the factorial of n minus 1, held down at the bottom by the factorial of 1 being 1. A student at Mpwapwa called it circular.

Write a program that computes a factorial by recursion, and states the base case the recursion stops at.

Read one whole number from the keyboard.

Compute its factorial using a recursive function.

Print the base case the recursion stops at, in words, before asking.

Refuse a negative number instead of answering it.

Model Answer

Program 4.70 writes the definition down as it stands: factorial returns 1 when n has come down to 1 or 0, and otherwise returns n times factorial of n minus 1, so the calls run downwards to the base case and the multiplications happen on the way back, giving 720 for the 6 that was typed.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  long long factorial(int n)
 5  {
 6      if (n <= 1)
 7      {
 8          return 1;
 9      }
10      return n * factorial(n - 1);
11  }
12
13  int main()
14  {
15      int number = 0;
16      cout << "Base case: the factorial of 0 and of 1 is 1" << endl;
17      cout << "Enter a whole number: ";
18      cin >> number;
19      if (number < 0)
20      {
21          cout << "No factorial exists for a negative number" << endl;
22          return 0;
23      }
24      cout << number << " factorial is " << factorial(number) << endl;
25      return 0;
26  }
```

Typed at the program

```
6
```

Output

```
Base case: the factorial of 0 and of 1 is 1
Enter a whole number: 6 factorial is 720
```

Program 4.70: *A recursive factorial with its base case announced and negative input refused. The number 6 was typed at the prompt.*

The base case is the whole safety of it: a recursive function needs one case that returns without calling itself, and every call must move towards that case. Here n falls by exactly one each time and n of 1 or less returns 1, so the descent is finite and its end is certain.

Why the guard is not decoration: a negative number satisfies n <= 1 at once, so without the test at line 19 the function would return 1 for an entry of minus 4. That is not a crash but something worse: a confident wrong answer. The guard refuses before any call is made.

The one design decision: the return type is long long rather than int, because a factorial outgrows a small whole number type after a very few steps, while the parameter stays int because it counts steps and never grows.

Write the base case first, then the step that moves towards it, and the recursion is finished.

The Master Key: *A recursive function is a base case that returns without calling, plus a step that calls itself on a strictly smaller argument, and a guard on input the base case would answer wrongly.*

BUILD Question 74

Two lengths of cloth lie on the bench at the tailoring shop in Hanang, 48 centimetres and 36. The tailor wants the widest strip that cuts from both with nothing left over, and subtraction gets there slowly.

Write a program that computes the greatest common divisor of two numbers by recursion, and their lowest common multiple from it.

Read two whole numbers from the keyboard.

Compute their greatest common divisor with a recursive function.

Compute their lowest common multiple from that divisor.

Refuse zero and negative input.

Model Answer

Program 4.71 uses Euclid's rule, that the greatest common divisor of two numbers is the greatest common divisor of the second and the remainder of the first divided by it, until the remainder reaches 0 and the other number is the answer; the lowest common multiple then follows by division and multiplication, and for 48 and 36 the program prints 12 and 144.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int gcd(int first, int second)
 5  {
 6      if (second == 0)
 7      {
 8          return first;
 9      }
10      return gcd(second, first % second);
11  }
12
13  int main()
14  {
15      int first = 0;
16      int second = 0;
17      cout << "Enter two whole numbers: ";
18      cin >> first >> second;
19      if (first <= 0 || second <= 0)
20      {
21          cout << "Both numbers must be above zero" << endl;
22          return 0;
23      }
24      int factor = gcd(first, second);
25      cout << "GCD: " << factor << endl;
26      cout << "LCM: " << first / factor * second << endl;
27      return 0;
28  }
```

Typed at the program

```
48 36
```

Output

```
Enter two whole numbers: GCD: 12
LCM: 144
```

Program 4.71: *A recursive greatest common divisor, and the lowest common multiple taken from it. The numbers 48 and 36 were typed at the prompt.*

How the recursion shrinks: gcd of 48 and 36 becomes gcd of 36 and 12, which becomes gcd of 12 and 0, which returns 12. A remainder is always smaller than the number it was taken against, so the second argument falls strictly and the base case of a second argument of 0 is always reached.

The lowest common multiple from the divisor: the product of two numbers divided by their greatest common divisor. Line 26 writes it as first divided by the divisor and then multiplied by second, dividing before multiplying so that no intermediate value grows larger than it needs to.

The one design decision: the recursion is written with the remainder rather than with repeated subtraction. Both are Euclid and both are correct, but 48 and 36 take two steps by remainder against several by subtraction, and the gap widens as the numbers grow apart.

The rule and the program say the same thing here, which is what makes this recursion worth writing.

The Master Key: *Euclid's rule states that the greatest common divisor of two numbers equals that of the second and the remainder, stopping when the remainder is 0, and the lowest common multiple is their product divided by it.*

BUILD Question 75

Two routes to one answer, and one of them made a quarter of a million function calls to arrive where the other arrived in twenty five steps. Both were correct C++, written from the same definition.

Write a program that computes one Fibonacci number twice, by recursion and by a loop, and counts the work each one does.

Compute one Fibonacci number twice, by recursion and by a loop.

Count the calls the recursion makes and the passes the loop makes.

Print both answers and both counts.

Model Answer

Program 4.72 computes the twenty fifth Fibonacci number both ways and reports what each way cost: the recursion returns 75025 after 242785 calls, and the loop returns the same 75025 after 25 passes, which is the same answer for rather less than a thousandth of the work.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int fibRecursive(int n, int &calls)
 5  {
 6      calls++;
 7      if (n < 2)
 8      {
 9          return n;
10      }
11      return fibRecursive(n - 1, calls) + fibRecursive(n - 2, calls);
12  }
13
14  int main()
15  {
16      int n = 25;
17      int calls = 0;
18      int steps = 0;
19      int byRecursion = fibRecursive(n, calls);
20      int current = 0;
21      int next = 1;
22      for (int i = 0; i < n; i++)
23      {
24          int following = current + next;
25          current = next;
26          next = following;
27          steps++;
28      }
29      cout << "recursion gave " << byRecursion << " after "
30           << calls << " calls" << endl;
31      cout << "the loop gave " << current << " after "
32           << steps << " passes" << endl;
33      return 0;
34  }
```

Output

```
recursion gave 75025 after 242785 calls
the loop gave 75025 after 25 passes
```

Program 4.72: *The same Fibonacci number by recursion and by a loop, each counting what it did. The counter is a reference parameter, so every call adds to the one counter in main.*

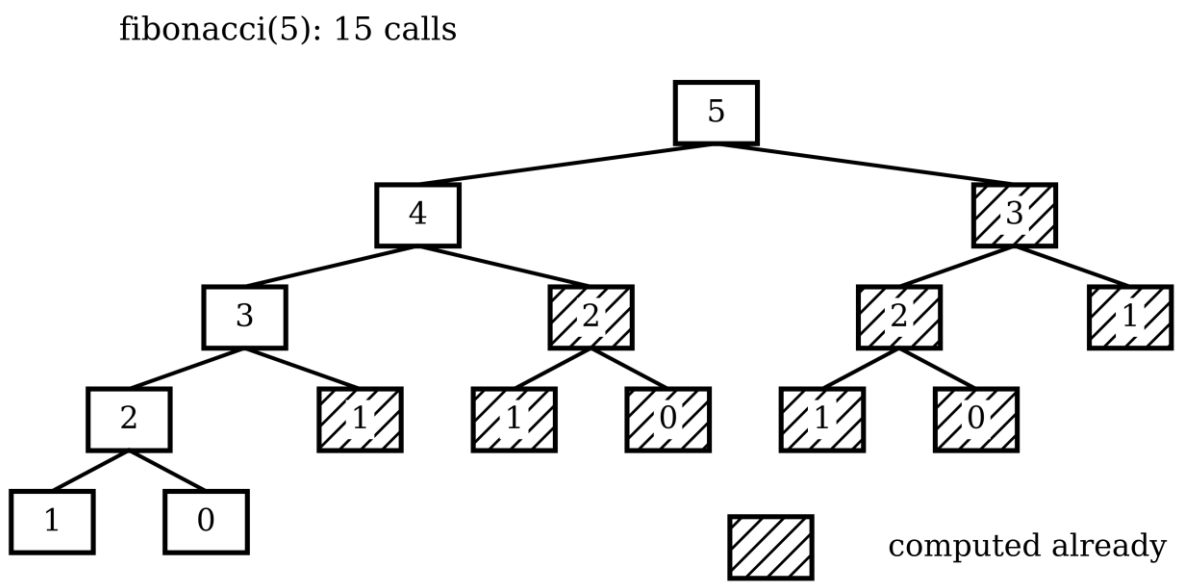


Figure 4.16: *The calls made by a naive Fibonacci of five. The shaded calls compute a value that has already been computed, and they are more than half the tree.*

Where the calls go: each call above the base makes two more, so the calls form a tree that nearly doubles at every level. Figure 4.16 draws the whole of it for an argument of 5, which costs 15 calls to answer a question about 6 different arguments, so 9 of the 15 recompute something already computed.

How the counting is done: calls is a reference parameter, passed down through every recursive call, so all of them increment one variable that lives in main. A parameter passed by value would have given each call a private counter and main would have read 0.

The one design decision: the loop carries only the two numbers it needs and adds them once per pass, so its cost rises in step with n while the cost of the recursion rises with the answer itself. Nothing in the loop is ever computed twice.

The recursion and the loop agree on the answer and disagree by four orders of magnitude on the price.

The Master Key: *A naive recursive Fibonacci recomputes the same values over and over, costing 242785 calls where a loop carrying two values costs 25 passes for the same 75025.*

HOT Question 76

Elegant is the word that follows recursion about. The same short function that makes one program clear is why another dies without a message halfway through a large file.

Evaluate the claim that recursion should be preferred because it is elegant.

Model Answer

The claim names a real advantage and stops short of a reason: prefer recursion where the problem is itself defined recursively or branches into subproblems, and where the depth stays bounded and small, and prefer a loop everywhere else, because elegance buys nothing that survives 242785 calls for an answer a loop reaches in 25 passes.

What recursion expresses that a loop does not: a definition that contains itself. The rule for a factorial, or for a greatest common divisor, is recursive when it is written in words, and the function is then a transcription that can be checked against the rule line by line.

What it costs in memory: one stack frame per call that has not yet returned, holding its parameters, its local variables and the address to come back to. A loop keeps one set of variables however many passes it makes, and depth that grows with the data can exhaust the stack without a diagnostic.

What it costs in repeated work: nothing, where each call makes one further call, and a great deal where each call makes two. The figures from Program 4.72 were counted by the program rather than estimated.

The twenty fifth Fibonacci number, two ways	Work done	Answer
Recursion, two calls at every step	242785 calls	75025
A loop carrying the last two values	25 passes	75025

Table 4.15: *The same answer at two prices, both counted by Program 4.72 while it ran.*

Where the repetition can be removed: keeping the answers already worked out, and consulting them before making a call, removes the second computation of every value and leaves the recursive shape standing.

Tail-shaped recursion: a call is tail shaped when the recursive call is the last act of the function, with nothing left to do after it returns, as in Program 4.73, which carries the running product down as an argument. A compiler may then reuse the frame rather than stack a new one, but C++ nowhere requires it, so tail shape is a hope and not a guarantee.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  long long factorialTail(int n, long long carried)
 5  {
 6      if (n <= 1)
 7      {
 8          return carried;
 9      }
10      return factorialTail(n - 1, n * carried);
11  }
12
13  int main()
14  {
15      cout << "10 factorial: " << factorialTail(10, 1) << endl;
16      return 0;
17  }
```

Output

```
10 factorial: 3628800
```

Program 4.73: *A factorial in tail shape. The multiplication is done before the call rather than after it, so nothing is left waiting when the call returns.*

Where a loop is genuinely harder to get right: wherever the work branches. Walking a folder that holds folders, or sorting by splitting a list in two, leaves unfinished work waiting. A loop manages it only by keeping a stack of its own by hand, which is what the recursion kept for you, and that hand-kept stack is where the mistakes are made.

The judgement: elegance is a reason to prefer recursion only once the depth is bounded and the work is not repeated. Where the problem branches and the depth grows with the logarithm of the data, recursion is both clearer and cheap. Where one accumulator would serve, the loop is the honest choice.

Elegance that cannot be paid for describes the code and not the program.

The Master Key: *Prefer recursion where the definition is recursive or the problem branches and the depth stays bounded, and prefer a loop wherever one accumulator would do, because frames and repeated calls are paid for while the program runs.*

HOT Question 77

Copying is waste, said the note in the margin, so pass everything by reference. Applied to a single whole number it saves nothing, and applied without const it hands the called function the caller's own paper.

A candidate argues that a function should always pass by reference, because copying is wasteful. Evaluate this and reach a judgement.

Model Answer

The candidate has the right worry and the wrong rule: pass by value anything small enough to sit in a machine word, where the function only reads it, pass by const reference anything large that the function only reads, and pass by non-const reference only where changing the caller's variable is the point, because a reference is not merely a cheaper copy, it is permission to write.

What copying a small value actually costs: one value moved into a register or a slot on the stack, which is about as cheap as an operation gets. A reference is itself an address, no smaller, and must then be followed to reach the value. For an int it is never meaningfully faster.

What copying a large object actually costs: every byte of it, on entry, and again on every call. Here the candidate is right and the saving is real, particularly for a call made inside a loop, where the copying is repeated as often as the loop runs.

What a non-const reference permits: the called function may write into the caller's variable, and nothing at the call site says so. A reader looking at the call alone cannot tell whether the variable survives it unchanged, which is the cost the candidate never counted.

Why const reference answers most of it: no copy is made, and the compiler refuses any statement that would write through the reference. It gives the whole saving and none of the permission, so a function that only reads a large argument should take it that way.

How the argument is passed	What it costs	What the function may do
A small value, by value	one word copied	nothing the caller can see
A small value, by reference	an address, then a step to follow it	write into the caller's variable
A large object, by value	every byte copied, on every call	nothing the caller can see
A large object, by const reference	nothing copied	read it, and nothing else

Table 4.16: *The cost and the permission are separate questions, and only the second of them is settled by const.*

Where a non-const reference is exactly right: where changing the caller's variable is the purpose of the call and not a side effect of an optimisation. Exchanging two values, adding to a running total, and the call counter in Program 4.72 are all of that kind, and each of them would be impossible by value.

What the argument gets right: copying a large object needlessly is genuine waste, and a candidate who passes everything by value will write slow programs once the objects grow. The error is the leap from a true statement about large objects to a rule about every parameter.

The judgement: decide by size first and by intent second. Small and read only goes by value; large and read only goes by const reference; anything the function must change goes by non-const reference, which is the only case where the reference is chosen for what it permits.

The question is not whether a copy is expensive, but whether the caller has agreed to be written to.

The Master Key: *Pass small values by value and large read-only objects by const reference, and use a non-const reference only where the caller is meant to see the change, because a reference saves a copy and grants permission at the same time.*

Arrays, one dimensional and two dimensional

This section is where a program stops naming its variables one at a time. It defines the array, gives the two routes by which values get into one, and states the rule that the common teaching leaves to inference, that the valid indices run from 0 to one less than the number of elements. It then describes exactly what a program does when an index leaves that range, which is not what most candidates expect, and it does the thing this topic is almost never shown doing: it passes an array to a function and watches the length disappear on the way. One listing here is killed by the operating system while it runs, and the declaration that kills it is lifted from an exercise that asks only for the multiplication.

BINDER Question 78

A strip of card nailed above the counter at the water kiosk in Bumbuli is ruled into five boxes for five meters, and the first is not numbered one. Nobody is to rule a sixth.

Define an array, state how it is declared and initialised, and state which indices are valid.

Model Answer

An array is a fixed number of elements of one type, laid out one after another in a single unbroken block of memory and reached by one name with an index in square brackets; it is declared by writing the element type, the name and the number of elements, initialised either by a braced list at the declaration or element by element afterwards, and its valid indices run from 0 to one less than that count.

What the declaration fixes: three things at once: the type of every element, the name of the whole, and how many elements there are. Line 7 reserves five int sized cells side by side with nothing between them, and that contiguity is what makes an index cheap, since element i is reached by adding i element widths to the address of the first.

The two routes to a value: a braced list at the declaration, as on line 7, fills the elements in order. Assignments by subscript, as on lines 13 and 14, fill them one at a time afterwards. A list shorter than the array sets the rest to zero, so empty braces zero every element.

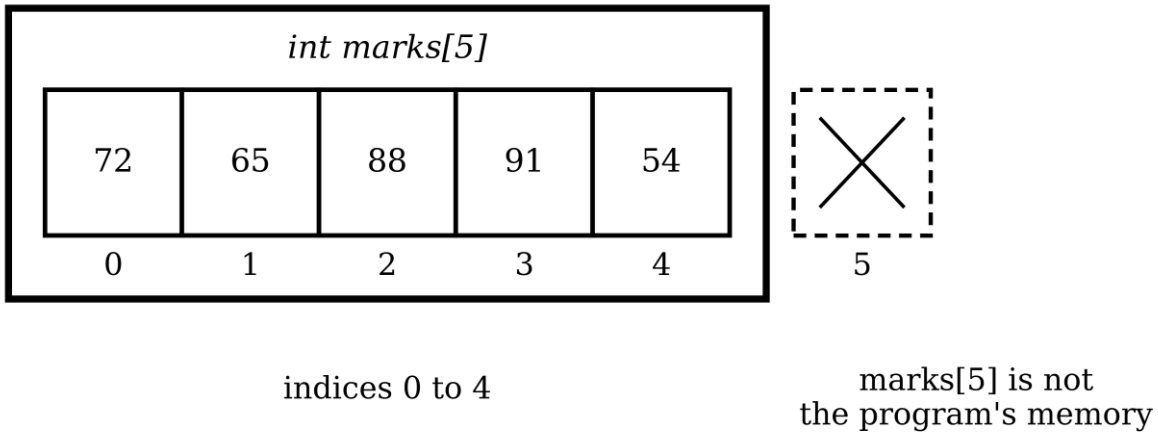


Figure 4.17: *A five element array. The valid indices run from zero to four, and the cell drawn outside the boundary is not the array's memory, although nothing in the language stops a program writing to it.*

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      const int SIZE = 5;
 7      int marks[SIZE] = {72, 65, 88, 91, 54};
 8      int fees[SIZE];
 9      for (int i = 0; i < SIZE; i++)
10      {
11          fees[i] = 0;
12      }
13      fees[0] = 12000;
14      fees[3] = 9000;
15      cout << "marks holds";
16      for (int i = 0; i < SIZE; i++)
17      {
18          cout << " " << marks[i];
19      }
20      cout << endl;
21      cout << "the first is " << marks[0] << " and the last is "
22           << marks[SIZE - 1] << endl;
23      cout << "fees holds";
24      for (int i = 0; i < SIZE; i++)
25      {
26          cout << " " << fees[i];
27      }
28      cout << endl;
29      return 0;
30  }
```

Output

```
marks holds 72 65 88 91 54
the first is 72 and the last is 54
fees holds 12000 0 0 9000 0
```

Program 4.74: *One array initialised by a braced list, another filled element by element. Line 22 reaches the last element as marks[SIZE - 1], which stays correct if the size is changed.*

Which indices are valid: the first element sits at index 0 and the last at one less than the count, so a five element array has exactly the indices 0, 1, 2, 3 and 4. Figure 4.17 draws the cell at index 5 outside the array, which is where it belongs.

The size is settled before the program runs: the number in the brackets must be a constant the compiler can work out for itself, so a size read from the user is not a legal bound for an ordinary array. Program 4.75 asks for one anyway, and this compiler takes it as an extension while reporting that ISO C++ forbids it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main()
 5  {
 6      int wanted = 0;
 7      cout << "How many marks? ";
 8      cin >> wanted;
 9      int marks[wanted];
10      for (int i = 0; i < wanted; i++)
11      {
12          marks[i] = 0;
13      }
14      marks[0] = 72;
15      cout << "an array of " << wanted << " elements was made, and the"
16           << " first holds " << marks[0] << endl;
17      return 0;
18  }
```

Typed at the program

```
3
```

Output

```
How many marks? an array of 3 elements was made, and the first holds 72
```

Program 4.75: *A size taken from the keyboard. The bound on line 9 is not a constant expression, and the compiler warns that ISO C++ forbids it before building the program anyway.*

Count the elements, subtract one, and that is the last index you may write.

The Master Key: *An array is a fixed number of elements of one type in one unbroken block, its size settled when the program is compiled and its valid indices running from 0 to n minus 1.*

BINDER Question 79

There was a sixth mark on the screen and the array held five. Nothing had turned red, and the extra number sat in the column looking as reasonable as the five above it.

Explain what happens when a program uses an index outside the valid range.

Model Answer

Nothing intervenes: an index outside 0 to n minus 1 is undefined behaviour, which means the standard places no requirement whatever on what follows, so the compiler is not obliged to diagnose it and here says nothing at all, the program does not halt at it, and the subscript reaches whatever bytes lie at that position and hands them back as though they were data.

What the run actually printed: Program 4.76 reads element 5 of two different five element arrays. One printed 0 and the other printed 40, which is the value of classSize, the variable that happens to sit next to it in memory. Neither number came out of an array.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int marks[5] = {72, 65, 88, 91, 54};
 5  int classSize = 40;
 6
 7  int main()
 8  {
 9      int scores[5] = {72, 65, 88, 91, 54};
10      cout << "scores[4] holds " << scores[4] << endl;
11      cout << "scores[5] reads " << scores[5] << endl;
12      cout << "marks[4] holds " << marks[4] << endl;
13      cout << "marks[5] reads " << marks[5] << endl;
14      cout << "classSize holds " << classSize << endl;
15      cout << "the program did not stop" << endl;
16      return 0;
17  }
```

Output

```
scores[4] holds 54
scores[5] reads 0
marks[4] holds 54
marks[5] reads 40
classSize holds 40
the program did not stop
```

Program 4.76: *The same mistake made twice on the same machine, giving two different answers and no complaint from either the compiler or the run. Lines 11 and 13 are both outside the array.*

Why the two answers differ: because neither is an answer. The bytes at that position belong to whatever the compiler put there, and the two arrays have different neighbours. That repeated runs give the same numbers is an accident of this machine and this build, guaranteed by nothing.

What a write out of range does: the same thing in the other direction, and worse. It alters a neighbouring variable, or a saved return address, without a word. The program that then misbehaves may do so far from the line that broke it.

Why the language does not check: a check would cost a comparison against the bound at every subscript, in every loop, on every machine that ran the program, including the great majority of accesses never in any danger. C++ refuses costs nobody asked for and leaves the duty with the programmer.

What the programmer must do instead: keep the count beside the array and write every loop to stop at one less than it, and test any index that arrives from a user or out of a calculation before using it as a subscript.

Silence from the compiler is not evidence that the access was legal.

The Master Key: *An index outside 0 to n minus 1 is undefined behaviour: nothing is required to diagnose it, the program does not stop, and the value read is only whatever happened to be lying there.*

BINDER Question 80

The ledger at the maize cooperative in Mlimba is ruled into twelve boxes, three rows of four, and anybody can see the rows. The memory that holds the same table has no rows in it whatever.

Explain two dimensional arrays: how they are declared, initialised and traversed, and how they are laid out in memory.

Model Answer

A two dimensional array is declared with two sizes in two separate pairs of square brackets, initialised either by a braced list of braced rows or by a pair of nested loops, traversed with the outer loop over the rows and the inner over the columns, and stored in row major order, which means the rows are laid down one after another in a single strip.

Declaring it and filling it: int litres[3][4] reserves 12 elements, three rows of four, in one block. The nested loops on lines 23 to 29 fill it with the outer loop over rows, which walks the memory in the order it is stored.

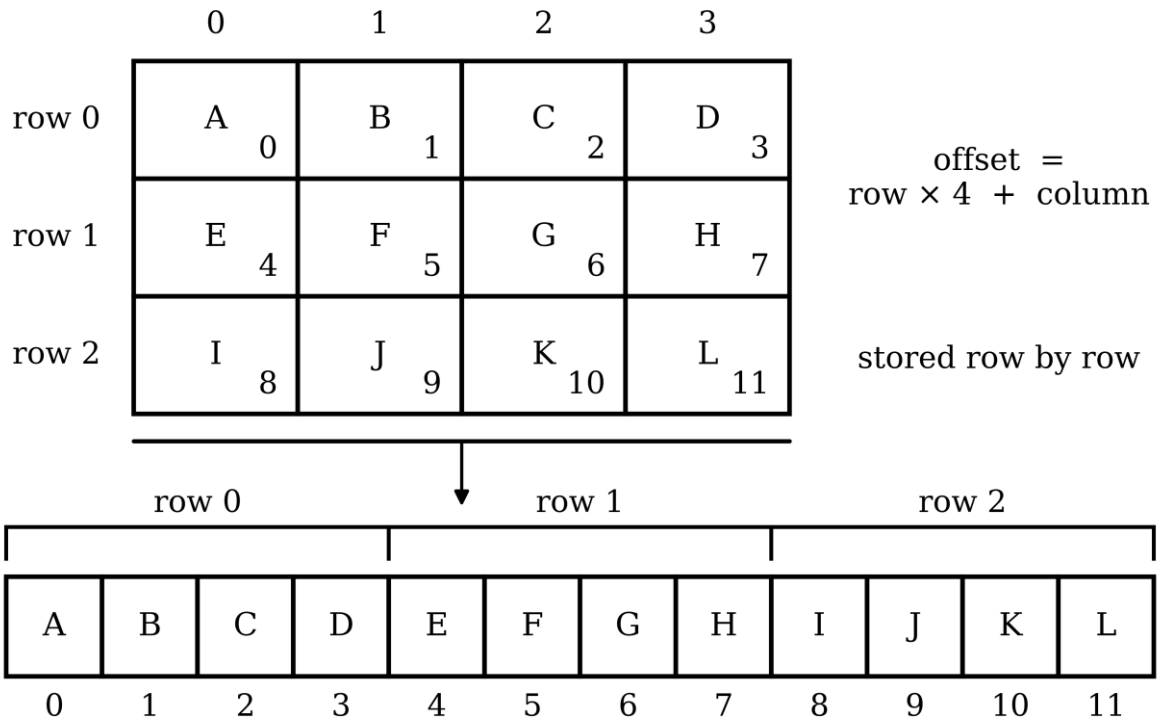


Figure 4.18: *A two dimensional array and the single strip of memory it actually occupies. The rows are laid down one after another, so the offset of any element is its row number times the column count, plus its column number.*

The offset rule: the row number times the number of columns, plus the column number. Element [1][2] of a four column grid sits at offset 6, which is 1 times 4 plus 2, and the last element sits at offset 11. Program 4.77 stores each offset in its own cell so the printed grid is the strip.

```
 1  #include <iostream>
 2  #include <iomanip>
 3  using namespace std;
 4
 5  const int ROWS = 3;
 6  const int COLS = 4;
 7
 8  void printGrid(int grid[][COLS], int rows)
 9  {
10      for (int r = 0; r < rows; r++)
11      {
12          for (int c = 0; c < COLS; c++)
13          {
14              cout << setw(4) << grid[r][c];
15          }
16          cout << endl;
17      }
18  }
19
20  int main()
21  {
22      int litres[ROWS][COLS];
23      for (int r = 0; r < ROWS; r++)
24      {
25          for (int c = 0; c < COLS; c++)
26          {
27              litres[r][c] = r * COLS + c;
28          }
29      }
30      printGrid(litres, ROWS);
31      cout << "litres[1][2] holds " << litres[1][2]
32           << " at offset " << 1 * COLS + 2 << endl;
33      return 0;
34  }
```

Output

```
   0   1   2   3
   4   5   6   7
   8   9  10  11
litres[1][2] holds 6 at offset 6
```

Program 4.77: *A three by four grid filled with the offset of each cell and printed as a grid. The function is told the column count on line 8 and the row count as an ordinary parameter.*

Why a function must be told the column count: because the offset cannot be worked out without it. The parameter is written int grid[][COLS], and the compiler needs that COLS to turn two subscripts into one address. The row count is not needed for the arithmetic and travels separately.

The rows are not separate objects: there are not three arrays here, there is one array of 12 elements. No row has an address of its own to be exchanged with another, and nothing separates the last element of one row from the first element of the next.

Read the two subscripts as arithmetic, and the single strip stops being a surprise.

The Master Key: *A two dimensional array is one block held row after row, so the offset of an element is its row times the column count plus its column, and a function must be given the column count.*

REAL Question 81

The function was handed an array of six readings and reported, with complete confidence, that it held two. The same sizeof that gave the right answer three lines earlier gave the wrong one three lines later.

The program below passes an array to a function and tries to find its length inside that function. Explain what goes wrong, and correct it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  void reportSize(int readings[])
 5  {
 6      cout << "inside the function sizeof gives "
 7           << sizeof(readings) << endl;
 8      cout << "and the division gives "
 9           << sizeof(readings) / sizeof(readings[0]) << endl;
10  }
11
12  int main()
13  {
14      int readings[6] = {21, 24, 22, 27, 25, 23};
15      cout << "in main sizeof gives " << sizeof(readings) << endl;
16      cout << "and the division gives "
17           << sizeof(readings) / sizeof(readings[0]) << endl;
18      reportSize(readings);
19      return 0;
20  }
```

Output

```
in main sizeof gives 24
and the division gives 6
inside the function sizeof gives 8
and the division gives 2
```

Program 4.78: *The same two expressions written in main and again inside a function. Nothing about the array changed between them.*

Model Answer

The array never arrives: an array passed to a function decays to a pointer to its first element, so inside the function the name is a pointer and sizeof measures a pointer, which on this machine is 8 bytes against the 24 bytes the same expression measured in main, and the division that counts elements therefore reports 2 where there are 6.

What is actually passed: one address, and not 24 bytes of data. A parameter written int readings[] is another spelling of int *readings, and the compiler treats the two as the same thing. The brackets in a parameter list carry no size and reserve nothing.

What each sizeof measured: in main the name is still an array, so sizeof reports the whole block. In reportSize the name is a pointer, so sizeof reports the pointer. Both are correct measurements of different things, which is exactly why the fault survives being read.

Where sizeof was written	What it reported	What it measured
In main, on the array	24	the whole block of 6 int elements
In main, divided by the size of one element	6	the number of elements, correctly
In the function, on the parameter	8	one pointer, which is what the parameter is
In the function, divided by the size of one element	2	how many int values fit inside a pointer, which answers nothing

Table 4.17: *Both numbers are honest measurements and only one of them measures the array. Every size here was measured by sizeof on this machine, where an int is 4 bytes and a pointer is 8.*

The compiler does say something: it raises the warning that sizeof on an array function parameter returns the size of a pointer, once for each of the two lines. It is a warning and not an error, so the program still builds and still runs, and a candidate who compiles without the warning flags turned on never sees it.

Why an array is passed this way: copying every element into the call would cost the whole block on every call, and C++ does not do it. The address goes instead, which is cheap and carries a second consequence worth knowing: the function is working on the caller's own array and can change it.

The correction: pass the length as a second parameter, as Program 4.79 does. main can still work the count out with sizeof, because there the name is still an array, and the function then has a bound it may trust and a loop that stops at it.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int totalOf(const int readings[], int count)
 5  {
 6      int total = 0;
 7      for (int i = 0; i < count; i++)
 8      {
 9          total += readings[i];
10      }
11      return total;
12  }
13
14  int main()
15  {
16      int readings[6] = {21, 24, 22, 27, 25, 23};
17      int count = sizeof(readings) / sizeof(readings[0]);
18      cout << "main counted " << count << " elements" << endl;
19      cout << "their total is " << totalOf(readings, count) << endl;
20      return 0;
21  }
```

Output

```
main counted 6 elements
their total is 142
```

Program 4.79: *The corrected form. The length is computed where the array still is an array, and handed over with it.*

Once the array has crossed the parameter list, its length is a fact only the caller still holds.

The Master Key: *An array passed to a function decays to a pointer to its first element, so sizeof inside the function measures the pointer and not the array; pass the length as a second parameter.*

REAL Question 82

Fifty, times a hundred, times a hundred, times twenty five. The multiplication is a minute's work and the answer is right, and the hard half is whether the declaration behind it can exist at all.

An exercise sets a student to count the elements of the declaration below and say whether it is reasonable. Do both.

int mayah[50][100][100][25];

The declaration appears as a local variable inside a function.

Model Answer

The element count is 12500000, the product of the four bounds, and that arithmetic is correct; the declaration is nevertheless unreasonable, because at the 4 bytes an int measures on this machine those elements ask for 50000000 bytes, about 47.7 mebibytes, and a local variable is placed on the stack, which here is 8 mebibytes, so the program is killed the moment it touches the array.

Counting the elements: multiply the bounds together, however many dimensions there are: 50 times 100 times 100 times 25 gives 12500000. The count does not depend on the element type, and every one of those elements is reserved by the declaration whether the program ever uses it or not.

Turning elements into bytes: sizeof measures an int at 4 bytes here, so the block wants 50000000 bytes, which is about 47.7 mebibytes. The stack this machine gives a program is 8 mebibytes, so the declaration asks for nearly six times what exists.

What the run actually does: Program 4.80 prints its first line, enters the function, and is stopped from outside by the operating system. There is no exception to catch and no error code to test, because nothing inside the program detected anything; the process was simply killed.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  void fillGrid()
 5  {
 6      int mayah[50][100][100][25];
 7      mayah[0][0][0][0] = 1;
 8      mayah[49][99][99][24] = 2;
 9      cout << "the last element holds " << mayah[49][99][99][24] << endl;
10  }
11
12  int main()
13  {
14      cout << "about to declare 12500000 local elements" << endl;
15      fillGrid();
16      cout << "the function returned" << endl;
17      return 0;
18  }
```

What the run printed before the process was killed

```
about to declare 12500000 local elements
```

Program 4.80: *A local array of 12500000 elements. The first line reaches the screen and the function does not return, because 47.7 mebibytes will not fit on an 8 mebibyte stack.*

Why the compiler allowed it: the declaration is legal C++ and its size is a constant expression, which is all the compiler is asked to check. How much stack this program will be given is not known until it starts, so there is nothing for the compiler to object to.

The two remedies: declare it static, which moves the block off the stack and into the program's own storage, or allocate it while the program runs, which Section 11 covers. Neither reduces the 47.7 mebibytes, so the question worth asking first is whether 12500000 elements are needed at all.

A declaration can be arithmetically correct and still describe a program that cannot run.

The Master Key: *Multiply the bounds for the count and multiply again by the element size for the bytes, and remember that a local array of that size goes on a stack which here is only 8 mebibytes.*

BUILD Question 83

Ten marks on a slip of paper, and a teacher at a secondary school in Makete reads the best, the worst and the average in one pass. A program cannot look at ten numbers at once.

Write a program that reads ten marks into an array and reports the highest, the lowest and the mean.

Read ten marks from the keyboard into an array.

Report the highest mark, the lowest mark and the mean.

Print the mean to two decimal places.

Model Answer

Program 4.81 reads the ten marks into an array of ten elements and then walks that array once, keeping the largest seen so far, the smallest seen so far and a running total, and for the marks typed it reports a highest of 91, a lowest of 39 and a mean of 67.50.

```
 1  #include <iostream>
 2  #include <iomanip>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      const int COUNT = 10;
 8      int marks[COUNT];
 9      cout << "Enter " << COUNT << " marks: ";
10      for (int i = 0; i < COUNT; i++)
11      {
12          cin >> marks[i];
13      }
14      int highest = marks[0];
15      int lowest = marks[0];
16      int total = 0;
17      for (int i = 0; i < COUNT; i++)
18      {
19          if (marks[i] > highest)
20          {
21              highest = marks[i];
22          }
23          if (marks[i] < lowest)
24          {
25              lowest = marks[i];
26          }
27          total += marks[i];
28      }
29      cout << endl << "Highest: " << highest << endl;
30      cout << "Lowest: " << lowest << endl;
31      cout << "Mean: " << fixed << setprecision(2)
32           << static_cast<double>(total) / COUNT << endl;
33      return 0;
34  }
```

Typed at the program

```
72 65 88 91 54 47 80 63 39 76
```

Output

```
Enter 10 marks:
Highest: 91
Lowest: 39
Mean: 67.50
```

Program 4.81: *Ten marks read into an array, then one pass for the highest, the lowest and the total. The marks shown were typed at the prompt.*

The one design decision: highest and lowest are both seeded from marks[0] on lines 14 and 15, and not from 0 or from 100. A lowest seeded at 0 reports 0 for every class that ever runs, and a highest seeded at 100 reports 100; only a real element cannot be wrong.

Where the division would go wrong: total and COUNT are both int, so total / COUNT would throw the fraction away and print 67 for a mean of 67.5. Line 32 casts the total to double before dividing, and fixed with setprecision fixes the two decimal places.

What the array earns: all three answers could be reached without one, since each is carried forward as the marks arrive. The array earns its place the moment the marks are wanted a second time, to print those above the mean, and a program that kept only a total could not go back.

One pass answers all three questions, and the marks are still there for whatever is asked next.

The Master Key: *Seed the highest and the lowest from a real element rather than from a guessed extreme, and cast before dividing when the mean must carry a fraction.*

BUILD Question 84

The same nine numbers, read down the columns instead of along the rows. Nothing is added and nothing is thrown away, and in C++ it is one change to one subscript, and not the obvious one.

Write a program that reads a three by three matrix and prints it beside its transpose.

Read nine whole numbers from the keyboard into a three by three array.

Print the matrix and its transpose.

Line the columns up with setw.

Model Answer

Program 4.82 reads the nine numbers row by row into a three by three array and prints the matrix beside its transpose, the transpose obtained by exchanging the two subscripts at the moment of printing, so the row 4, 12, 9 comes back as the column 4, 12, 9 and no second array is built anywhere.

```
 1  #include <iostream>
 2  #include <iomanip>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      const int SIZE = 3;
 8      int table[SIZE][SIZE];
 9      cout << "Enter nine whole numbers, row by row: ";
10      for (int r = 0; r < SIZE; r++)
11      {
12          for (int c = 0; c < SIZE; c++)
13          {
14              cin >> table[r][c];
15          }
16      }
17      cout << endl << setw(15) << "the matrix" << setw(18)
18           << "its transpose" << endl;
19      for (int r = 0; r < SIZE; r++)
20      {
21          for (int c = 0; c < SIZE; c++)
22          {
23              cout << setw(5) << table[r][c];
24          }
25          cout << "   ";
26          for (int c = 0; c < SIZE; c++)
27          {
28              cout << setw(5) << table[c][r];
29          }
30          cout << endl;
31      }
32      return 0;
33  }
```

Typed at the program

```
4 12 9
7 3 18
25 6 11
```

Output

```
Enter nine whole numbers, row by row:
     the matrix     its transpose
    4   12    9       4    7   25
    7    3   18      12    3    6
   25    6   11       9   18   11
```

Program 4.82: *A three by three matrix and its transpose, printed side by side from one array. The nine numbers shown were typed at the prompt.*

The one design decision: the transpose is not stored, it is read. Line 23 prints table[r][c] and line 28 prints table[c][r] out of that same array, so the flip costs nothing but the order in which the elements are visited.

Why setw and not spaces: setw sets the width of the next item only, so 4 and 25 are both padded out to five columns and the columns line up. A fixed run of spaces between the numbers would not line up, because the numbers are of different widths.

Reading and storing in the same order: the reading loops fill the array row by row, which is the order the elements sit in memory, so the numbers as typed and the block as stored are in step. The transpose is the only place where that order is deliberately broken.

Exchange the subscripts rather than the data, and the transpose costs nothing at all.

The Master Key: *The transpose of a square matrix is the same array read with its two subscripts exchanged, so it needs no second array and no copying.*

HOT Question 85

Some languages stop the program the instant an index leaves the array, and name the line that did it. C++ prints something and carries on, and that is a decision rather than a gap nobody noticed.

Evaluate the claim that C++ should check array bounds automatically, as some other languages do.

Model Answer

The claim is right about the harm and wrong about the price, and the condition that decides it is where the cost falls: a check on every subscript belongs in any program whose cost of a wrong answer exceeds a comparison per access, which is nearly every program a candidate will write, and it does not belong in the language itself, because C++ is chosen precisely by people who will not pay for what they did not ask for.

What an automatic check would cost: two comparisons and a branch at every subscript, one against zero and one against the bound, run on every pass of every loop whether or not the index could have been wrong. In a loop already written to stop at the bound, every one of them is waste repeated as often as the loop runs.

Why C++ refused it: the language is built on the rule that a program does not pay for what it does not use. It is the language of operating systems, device drivers, browsers and the controllers inside machinery, where a fixed charge on every memory access is paid again by every user of every copy, for as long as the program exists.

What the claim gets right: the cost of not checking is real and it is not paid in speed. It is paid in wrong answers that look plausible, in faults that surface far from the line that caused them, and in security holes, most of which are writes past the end of an array. The great majority of programs would never feel the check.

Where the check is made	What it costs	What it catches
By the language, at every access	a comparison against each bound on every subscript ever written	every access outside the array, at the moment it happens
By the programmer, where it is asked for	the same comparison, only where it is written	the accesses the programmer thought to guard
Nowhere at all	nothing	nothing, until somebody notices the wrong answer

Table 4.18: *The three positions on bounds checking. C++ takes the middle one, and leaves the first to languages that made the other trade.*

What the language offers instead: the check where you ask for it. The standard library gives element access through an at function that tests the index and refuses a bad one, a build made with the address sanitiser reports the exact line of an out of range access, and at this level a guard of four lines does the same for an ordinary array.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  bool fetch(const int data[], int size, int index, int &value)
 5  {
 6      if (index < 0 || index >= size)
 7      {
 8          return false;
 9      }
10      value = data[index];
11      return true;
12  }
13
14  int main()
15  {
16      int marks[5] = {72, 65, 88, 91, 54};
17      int value = 0;
18      for (int wanted = 3; wanted <= 5; wanted++)
19      {
20          if (fetch(marks, 5, wanted, value))
21          {
22              cout << "element " << wanted << " holds " << value << endl;
23          }
24          else
25          {
26              cout << "element " << wanted << " does not exist" << endl;
27          }
28      }
29      return 0;
30  }
```

Output

```
element 3 holds 91
element 4 holds 54
element 5 does not exist
```

Program 4.83: *A checked access written by hand. The index is tested against both ends before the subscript happens, and the caller is told rather than surprised.*

The judgement: accept the claim for programs and reject it for the language. Write as though the check were there, by keeping the count beside the array, stopping every loop one short of it and testing every index that came from outside, and do not ask the language to do it, because that same absence is why C++ is used for the work it is used for.

The language declines to check on your behalf, which leaves the checking to you rather than making it unnecessary.

The Master Key: *C++ leaves out automatic bounds checking to avoid a cost on every access in every program, so the check belongs in your own code, at the points where an index could be wrong.*

HOT Question 86

One meter was taken out of service at the kiosk in Serengeti, and a loop closed the gap. The register printed afterwards was neat, complete and wrong: meter 104 was credited with twenty litres it had never sold.

A candidate argues that parallel arrays are a perfectly good way to keep related data together. Evaluate this and reach a judgement.

Model Answer

Parallel arrays are sound exactly while nothing is inserted, removed or reordered, and they should be given up the moment any of those three becomes possible, because the only thing tying index i of one array to index i of another is the programmer's own discipline, and there is nothing written down for the compiler to check.

The case for them: they need no new syntax, they cost nothing at run time, and one loop counter serves every array at once. For a table that is read and never edited, a fixed list of subjects beside a fixed list of pass marks, they are entirely adequate and perfectly readable.

What actually holds them together: a convention, and nothing else. The compiler sees two independent arrays that happen to be the same length. It cannot know they are meant to be read in step, so it cannot complain on the day they stop being read in step.

The failure mode: Program 4.84 removes one meter from a register of four. The loop that closes the gap was written for the meter array and not for the litres array, so the register printed afterwards credits meter 104 with the 20 litres that belonged to 103, and the 60 litres it did sell disappear.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  const int COUNT = 4;
 5
 6  void showRegister(const int meter[], const int litres[], int used)
 7  {
 8      for (int i = 0; i < used; i++)
 9      {
10          cout << "meter " << meter[i] << " sold " << litres[i]
11               << " litres" << endl;
12      }
13  }
14
15  int main()
16  {
17      int meter[COUNT] = {101, 102, 103, 104};
18      int litres[COUNT] = {30, 45, 20, 60};
19      int used = COUNT;
20      cout << "before meter 103 is taken out of service" << endl;
21      showRegister(meter, litres, used);
22      for (int i = 2; i < used - 1; i++)
23      {
24          meter[i] = meter[i + 1];
25      }
26      used--;
27      cout << "after meter 103 is taken out of service" << endl;
28      showRegister(meter, litres, used);
29      return 0;
30  }
```

Output

```
before meter 103 is taken out of service
meter 101 sold 30 litres
meter 102 sold 45 litres
meter 103 sold 20 litres
meter 104 sold 60 litres
after meter 103 is taken out of service
meter 101 sold 30 litres
meter 102 sold 45 litres
meter 104 sold 20 litres
```

Program 4.84: *Two parallel arrays and one edit applied to only one of them. Nothing warns, nothing crashes, and the second register is the right shape and the wrong content.*

Why the fault survives testing: there is no crash, no diagnostic and no impossible value. Every line of that output is a plausible reading of a plausible meter. A fault of this kind is caught by somebody who knows what meter 104 actually sold, which is usually much later than the afternoon it was introduced.

What the alternative costs: one array whose elements each carry both fields, so an entry moves as a single thing and cannot be half moved. It costs a declaration and a little more writing at each access, and the type that keeps related data in one object is what Section 11 is for.

The judgement: keep parallel arrays only where the data is small, fixed and read together, and abandon them wherever entries are added, removed or sorted. The deciding condition is not how many arrays there are but whether any operation must change them all at once, because that is the operation the compiler cannot watch you getting wrong.

Two arrays that must agree are one structure the program has not been told about.

The Master Key: *Parallel arrays hold together only by discipline, so they are safe for fixed data read side by side and unsafe wherever an entry can be added, removed or moved.*

Strings: char arrays, the C string library and std::string

This section is where a program stops holding numbers and starts holding text. It sets out the two ways C++ represents a string, the character array inherited from C and the std::string of the standard library, gives the small family of functions that work on the first and the single precondition every one of them carries, and shows what a copy that does not fit actually does to a running program. Three of its listings are wrong on purpose, and one of those three is the copy most often printed as a model answer, complete with a picture of it apparently working.

BINDER Question 87

Eight pigeonholes hang behind the counter of a printing shop, and the word chalked above them has only seven letters. The eighth holds a blank card, the only thing telling the boy where to stop reading.

Explain how a string is represented as a C-style character array, and why the terminator matters.

Model Answer

A C-style string is an ordinary array of char holding the characters of the text in order, followed by one further character whose value is zero, called the null terminator and written \0 with a backslash; the array itself carries no length, so that terminator is the only mark of where the text ends.

What the declaration reserves: an array of char with a stated size, and it must be at least one larger than the text it will hold. Malinyi is seven letters, so char town[8] is the smallest array that can hold it, seven cells for the letters and the eighth for the terminator.

Two different counts: sizeof reports the bytes the array occupies and strlen reports the characters before the terminator, so the two never agree for a full string. A candidate who treats them as the same number has already made the mistake this whole section is about.

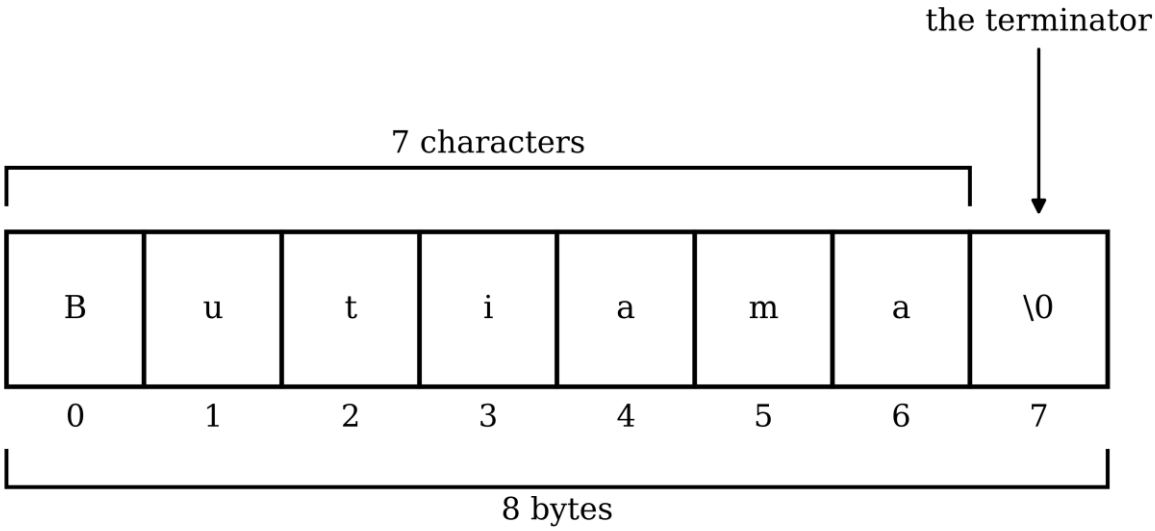


Figure 4.19: *A C-style string in memory. Seven characters need eight bytes, because the terminator is a character too and every library function depends on finding it.*

```
 1  #include <iostream>
 2  #include <cstring>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      char town[8] = "Malinyi";
 8      cout << town << endl;
 9      cout << "characters counted by strlen: " << strlen(town) << endl;
10      cout << "bytes occupied by the array: " << sizeof(town) << endl;
11      cout << "value held in the last byte: " << int(town[7]) << endl;
12      town[3] = '\0';
13      cout << "after a terminator is put at index 3: " << town << endl;
14      cout << "strlen now reports: " << strlen(town) << endl;
15      return 0;
16  }
```

Output

```
Malinyi
characters counted by strlen: 7
bytes occupied by the array: 8
value held in the last byte: 0
after a terminator is put at index 3: Mal
strlen now reports: 3
```

Program 4.85: *The terminator is what fixes the end of the text. Writing one at index 3 leaves the other letters untouched in memory and shortens the string to three characters.*

Why every function needs it: strlen walks forward from the first byte counting until it meets a zero, and strcpy, strcat and strcmp all find the end of their work the same way. Remove the terminator and none of them can tell where the text stops, so each reads on into memory that was never part of the string.

The mark that is not a terminator: students are very commonly shown the terminator written /0, with a forward slash. That is not a null character. It is a literal of two characters whose value is 12080, and a modern compiler rejects it outright rather than quietly storing something wrong.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      char town[5] = {'S', 'i', 'h', 'a', '/0'};
7      cout << town << endl;
8      return 0;
9  }
```

Compiler says

```
warning: multi-character character constant [-Wmultichar]
error: narrowing conversion of '12080' from 'int' to 'char'
```

Program 4.86: *The forward slash written where a backslash belongs. The message names the fault exactly, and the value 12080 is the two characters packed into one integer.*

Count the letters, add one for the terminator, and only then write the size in the brackets.

The Master Key: *A C-style string is a char array whose text is ended by the single character \0, written with a backslash, so the array must always be one byte longer than the text it holds.*

BINDER Question 88

A student wrote strcopy, compiled, and stared at a message about an undeclared identifier for twenty minutes before somebody pointed at the missing letter. The four functions it belongs to are all dangerous for one reason.

Describe the functions of the C string library and state the precondition every one of them carries.

Model Answer

The C string library, declared in the header <cstring>, provides strlen to count the characters before the terminator, strcpy to copy one string over another, strcat to join a second string onto the end of the first and strcmp to compare two strings, and every one of them requires that the destination array is already large enough, a condition none of them checks and none of them reports.

The header, and the one it is confused with: <cstring> holds the C functions that work on char arrays. <string> is a different header holding the class std::string, which is a different thing altogether. Include the one whose facilities you actually use, and never assume that either implies the other.

What each of the four does: strlen returns the character count, not the array size. strcpy replaces the whole contents of the destination. strcat begins writing at the destination terminator, so it appends. strcmp returns a value whose sign orders the two strings, and it changes neither of them.

Function	What it does	What must already be true
strlen(s)	returns the number of characters before the terminator	s ends in a terminator, or the count runs on past the array
strcpy(dest, src)	copies src, terminator included, over the start of dest	dest holds at least strlen(src) + 1 bytes
strcat(dest, src)	copies src onto the end of dest, starting at the terminator of dest	dest holds at least strlen(dest) + strlen(src) + 1 bytes
strcmp(a, b)	compares character by character and returns a signed value	both a and b end in a terminator

Table 4.19: *The four functions in daily use and the condition each one assumes. Nothing in the right-hand column is checked at run time, and nothing in it is reported when it is false.*

The precondition nobody states: the destination must be large enough before the call is made. These functions receive the address of the first byte and nothing else, so they cannot know how much room follows it. They write what they were given and stop only when the source terminator arrives.

```
 1  #include <iostream>
 2  #include <cstring>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      char line[32] = "";
 8      char first[] = "Mbozi";
 9      char second[] = "Momba";
10      strcpy(line, first);
11      strcat(line, " and ");
12      strcat(line, second);
13      cout << line << endl;
14      cout << "characters now in line: " << strlen(line) << endl;
15      cout << "bytes the array can hold: " << sizeof(line) << endl;
16      cout << "strcmp on the two towns: " << strcmp(first, second)
17           << endl;
18      return 0;
19  }
```

Output

```
Mbozi and Momba
characters now in line: 15
bytes the array can hold: 32
strcmp on the two towns: -13
```

Program 4.87: *The four functions used on an array with room to spare. The working array holds thirty two bytes and the finished text needs sixteen, so every call fits.*

A name that does not exist: strcopy is not a function in any C or C++ library, although it is often set as an exercise and written on boards. The compiler answers with an undeclared identifier, which is one of the few string mistakes that cannot reach a running program.

Learn the four names, then learn the sentence that belongs to all four: the destination must already be big enough, and nobody will tell you when it is not.

The Master Key: *strlen, strcpy, strcat and strcmp come from <cstring> and all of them assume without checking that the destination array is already large enough.*

BINDER Question 89

The borrower register at a secondary school library printed the first name of every reader and nothing else. The loop was sound, the array large enough; one operator was throwing away everything after the first space.

Explain what std::string provides that a character array does not.

Model Answer

A std::string stores its own length and obtains more memory whenever it needs it, so it can be constructed from a literal, assigned to with the ordinary assignment operator, joined with +, compared with ==, measured with length() or size(), cut with substr() and searched with find(), and it cannot overflow the way a fixed array can.

Construction and assignment: a string can be declared empty or given a value at once, and it can be assigned a longer value later. An array of char accepts a value only in its declaration; after that the array name cannot appear on the left of an assignment at all, which is why the C library needs a copying function at all.

Operators that do the obvious thing: the + operator joins two strings and produces a third, and == compares the characters and yields true or false. Both of these applied to char arrays would compare or add addresses instead of text, which is a fault that compiles quietly.

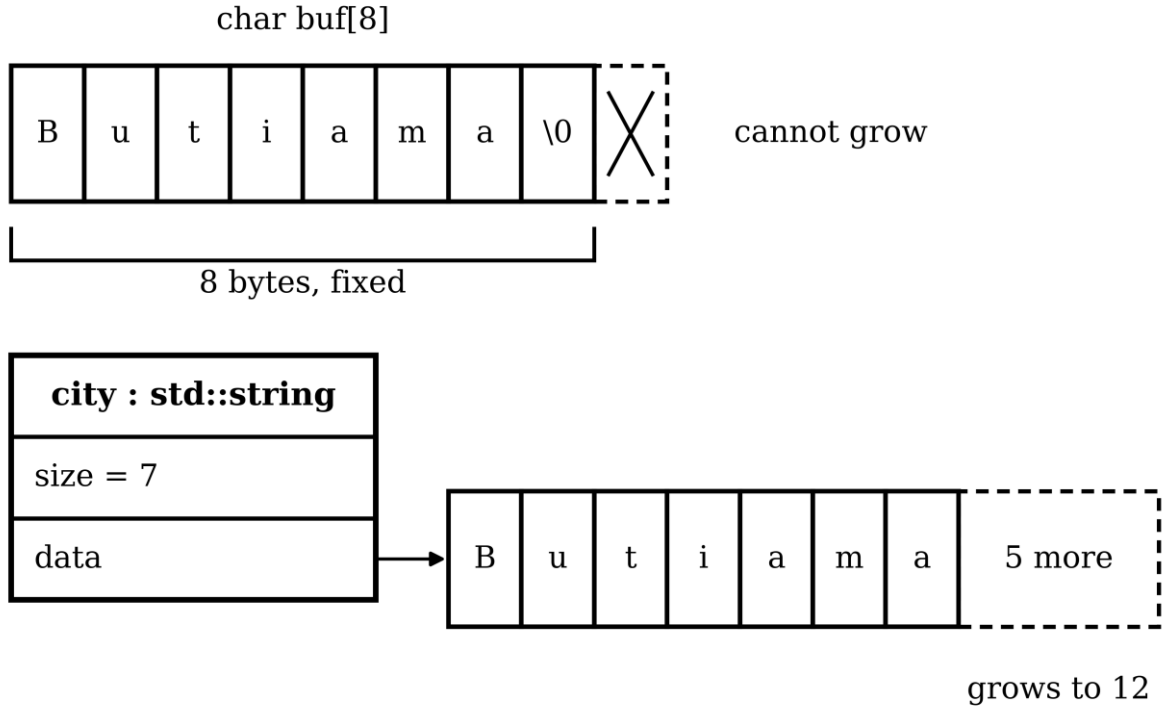


Figure 4.20: *A std::string beside a char array. The string stores its own length and obtains more room when it needs it; the array knows neither its length nor its limit.*

Measuring, cutting and searching: length() and size() are two names for the same count and either may be used. substr(start, count) returns a fresh string cut from the middle, and find(text) returns the position at which the text begins. Because the string grows on demand, an assignment that would have overrun an array simply succeeds.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      string town = "Mbozi";
 8      string other = "Momba";
 9      string both = town + " and " + other;
10      cout << both << endl;
11      cout << "length(): " << both.length() << endl;
12      cout << "size(): " << both.size() << endl;
13      cout << "substr(0, 5): " << both.substr(0, 5) << endl;
14      cout << "find(and): " << both.find("and") << endl;
15      cout << "town == other: " << (town == other) << endl;
16      town = other;
17      cout << "town after assignment: " << town << endl;
18      town = town + " Malinyi";
19      cout << "town after concatenation: " << town << endl;
20      cout << "its length now: " << town.length() << endl;
21      return 0;
22  }
```

Output

```
Mbozi and Momba
length(): 15
size(): 15
substr(0, 5): Mbozi
find(and): 6
town == other: 0
town after assignment: Momba
town after concatenation: Momba Malinyi
its length now: 13
```

Program 4.88: *The everyday operations on a std::string, with the value after each one. Nothing here declares a size, and nothing here can run past the end.*

Reading a whole line: the extraction operator stops at the first whitespace, so a full name typed at it yields only the first word and leaves the rest waiting in the stream. getline(cin, text) reads to the end of the line and keeps the spaces, which is what a program asking for a name or an address actually wants.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      string first;
 8      string leftOver;
 9      string whole;
10      cout << "Type the same full name on each of two lines." << endl;
11      cin >> first;
12      getline(cin, leftOver);
13      getline(cin, whole);
14      cout << "the extraction operator took: " << first << endl;
15      cout << "what it left in the stream:" << leftOver << endl;
16      cout << "getline took: " << whole << endl;
17      return 0;
18  }
```

Typed at the program

```
Neema Joseph Mbozi
Neema Joseph Mbozi
```

Output

```
Type the same full name on each of two lines.
the extraction operator took: Neema
what it left in the stream: Joseph Mbozi
getline took: Neema Joseph Mbozi
```

Program 4.89: *The same full name read twice. The extraction operator takes one word and leaves the remainder in the stream; getline takes the line entire.*

Where the text can change length, and text usually can, the string is the type built for it.

The Master Key: *A std::string knows its own length, grows when it must and supports assignment, +, == and the member functions, none of which a char array can offer; and getline reads a whole line where the extraction operator reads one word.*

REAL Question 90

There is a page in wide circulation showing one town name copied over another, beside a picture of it working. Compile it with the optimiser on and it prints a line about a buffer, then stops dead.

The listing below is faulty. Name the fault, state exactly what the compiler and the running program each do about it, and correct it.

```
 1  #include <iostream>
 2  #include <cstring>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      char town[8] = "Butiama";
 8      char longer[] = "Ngorongoro";
 9      strcpy(town, longer);
10      cout << town << endl;
11      return 0;
12  }
```

The program says

```
*** buffer overflow detected ***: terminated
```

Program 4.90: *The copy as it is commonly printed, built with the optimiser on.*

Model Answer

The fault is a buffer overflow in the call to strcpy: the destination town holds eight bytes and the source Ngorongoro needs eleven, so three bytes are written past the end of the array, and the program is killed by the C library before it prints anything at all.

Naming the fault: Butiama is seven characters and fits an array of eight exactly, terminator included. Ngorongoro is ten characters and needs eleven bytes. strcpy copies until it reaches the source terminator, so it writes eleven bytes into eight and the last three land on memory belonging to something else.

What the compiler does: with the optimiser on it raises two warnings and no error. One says that eleven bytes are being written into a region of size eight; the other reports the offsets that fall outside the array. Both are warnings, so the program is still built and can be run.

What the running program does: it prints nothing and is stopped. The C library keeps a record of the size of the destination and finds the copy running past it, prints a line about a buffer overflow, and raises the abort signal. The line that would have printed the town is never reached.

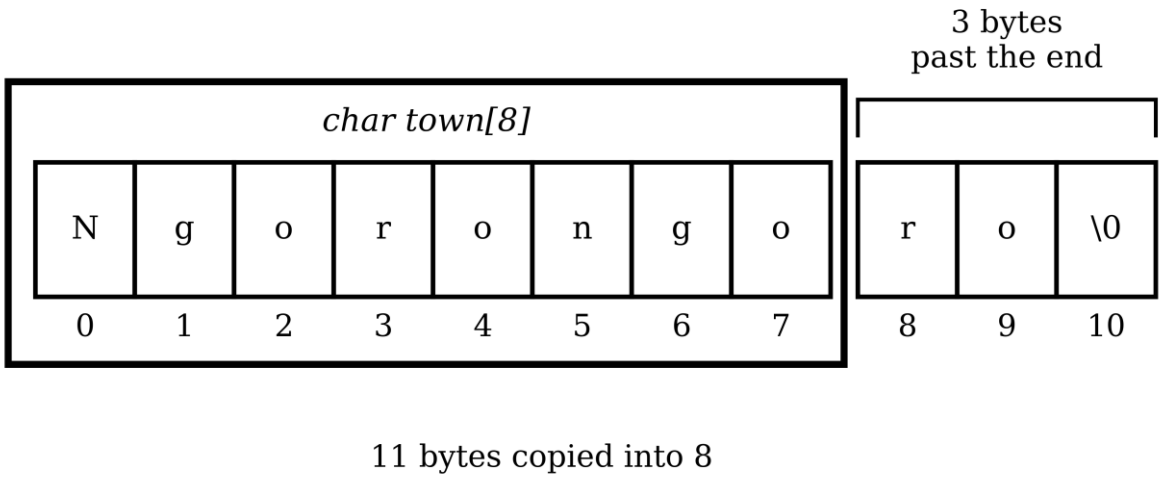


Figure 4.21: *What a copy that does not fit actually does. The bytes past the end of the array are not the program's to write, and nothing in the language stops it writing them.*

Three things to be clear about: the destination holds eight bytes and the source needs eleven, a fact fixed before the program runs. Nothing in the language checks it, because strcpy is handed one address and no size. And the result is not odd output; it is a program that does not finish.

Correcting it: either give the destination room for the longest text it will ever hold and guard the copy with a length test of your own, or use a std::string, which obtains the room it needs when it is assigned. The second removes the whole class of fault rather than this instance of it.

```
 1  #include <iostream>
 2  #include <cstring>
 3  #include <string>
 4  using namespace std;
 5
 6  int main()
 7  {
 8      char town[16] = "Butiama";
 9      char longer[] = "Ngorongoro";
10      if (strlen(longer) + 1 <= sizeof(town))
11      {
12          strcpy(town, longer);
13      }
14      cout << town << endl;
15      string safe = "Butiama";
16      safe = "Ngorongoro";
17      cout << safe << endl;
18      return 0;
19  }
```

Output

```
Ngorongoro
Ngorongoro
```

Program 4.91: *The same copy made safe twice over, first by sizing the array and testing before the call, then by using a std::string that sizes itself.*

Why the same fault in strcat is worse: strcat starts writing at the terminator of the destination rather than at its first byte, so the room it needs is the length already there plus the length of the source plus one. An array that had just enough space for one of them overflows by more, and the arithmetic is easier to get wrong.

Before any copy, ask how many bytes the destination holds and how many the source needs, because no part of the language will ask it for you.

The Master Key: *strcpy writes source length plus one bytes into whatever address it is given, so a destination of eight bytes and a source of eleven overflows by three, raises warnings rather than errors, and is stopped at run time by the library rather than by the language.*

REAL Question 91

Written on the board of a computer laboratory, in a hand that plainly believed it: strcmp returns minus one, zero or one. A program that trusts the board will one day be handed minus twelve.

A candidate is taught that strcmp returns minus one, zero or one. Test the claim and state what the standard actually guarantees.

Model Answer

The claim is false: the standard guarantees only the sign of the value, negative when the first string orders before the second, zero when they are equal and positive when it orders after, and on this compiler comparing Butiama with Ngorongoro returns minus twelve, which is neither of the two non-zero values the claim allows.

```
 1  #include <iostream>
 2  #include <cstring>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      char butiama[] = "Butiama";
 8      char ngorongoro[] = "Ngorongoro";
 9      char mbozi[] = "Mbozi";
10      char momba[] = "Momba";
11      char siha[] = "Siha";
12      char sihaAgain[] = "Siha";
13      char mal[] = "Mal";
14      char malinyi[] = "Malinyi";
15      cout << "Butiama, Ngorongoro: "
16           << strcmp(butiama, ngorongoro) << endl;
17      cout << "Ngorongoro, Butiama: "
18           << strcmp(ngorongoro, butiama) << endl;
19      cout << "Mbozi, Momba: " << strcmp(mbozi, momba) << endl;
20      cout << "Siha, Siha: " << strcmp(siha, sihaAgain) << endl;
21      cout << "Mal, Malinyi: " << strcmp(mal, malinyi) << endl;
22      cout << "does it equal -1? "
23           << (strcmp(butiama, ngorongoro) == -1) << endl;
24      cout << "is it less than zero? "
25           << (strcmp(butiama, ngorongoro) < 0) << endl;
26      return 0;
27  }
```

Output

```
Butiama, Ngorongoro: -12
Ngorongoro, Butiama: 12
Mbozi, Momba: -13
Siha, Siha: 0
Mal, Malinyi: -105
does it equal -1? 0
is it less than zero? 1
```

Program 4.92: *Five comparisons and the values that actually came back, followed by the wrong test and the right one on the same pair of towns.*

Where the numbers come from: this library returns the difference between the first pair of characters that differ. B is 66 and N is 78, so Butiama against Ngorongoro gives minus twelve. Reverse the arguments and the sign reverses with them. Mal against Malinyi differs where one string has run out, so the terminator, value zero, is compared against the letter i, value 105.

First string	Second string	Value returned	Sign
Butiama	Ngorongoro	-12	negative
Ngorongoro	Butiama	12	positive
Mbozi	Momba	-13	negative
Siha	Siha	0	zero
Mal	Malinyi	-105	negative

Table 4.20: *The pairs compared by Program 4.92, the value each comparison returned and the sign of that value. Only the last column is guaranteed by the standard.*

What the standard actually guarantees: it fixes the sign and nothing else. An implementation may return the character difference, as this one does, or minus one and one, or any other pair of values with the right signs, and it may return different values for the same pair on a different optimisation setting.

Why the wrong test survives so long: it does not fail every time. A pair of strings differing by one in their first differing character returns minus one, and the test passes. The program then runs correctly for months and takes the wrong branch the first time it meets a pair that differs by more.

The test to write instead: compare against zero and never against a value. strcmp(a, b) < 0 asks whether a orders before b, strcmp(a, b) == 0 asks whether they are equal, and strcmp(a, b) > 0 asks the reverse of the first. All three are guaranteed; strcmp(a, b) == -1 is a bug that compiles.

The reading of the sign: negative means the first argument comes first in order, which is the reading candidates most often get backwards. Read it as a subtraction: the value is what you get by taking the second from the first, so a negative answer means the first was the smaller.

Test the sign of what strcmp returns, never the value, because the value is the one thing the standard declines to promise.

The Master Key: *strcmp guarantees only the sign of its result, so strcmp(a, b) < 0 is correct and strcmp(a, b) == -1 is a bug that compiles and passes its first few tests.*

BUILD Question 92

A tailoring order book has a column for the customer name and a column for the number of letters in it, filled in by hand and wrong on one line in six. Counting words is harder still.

Write a program that reads a whole line of text and reports its length in characters, its length in words, and the line in upper case.

Read one whole line of text typed at the keyboard, spaces included.

Report the number of characters in the line.

Report the number of words in the line.

Report the whole line converted to upper case.

Use std::string and getline.

Model Answer

Program 4.93 reads the line with getline so that the spaces survive, walks it once with a subscript, and does the counting and the conversion in that single pass, treating a word as a run of characters beginning wherever a non-space follows a space or the start of the line.

```
 1  #include <iostream>
 2  #include <string>
 3  #include <cctype>
 4  using namespace std;
 5
 6  int main()
 7  {
 8      string line;
 9      int words = 0;
10      bool inWord = false;
11      cout << "Enter a line of text: ";
12      getline(cin, line);
13      for (size_t i = 0; i < line.length(); i = i + 1)
14      {
15          unsigned char ch = line[i];
16          if (isspace(ch))
17          {
18              inWord = false;
19          }
20          else if (!inWord)
21          {
22              inWord = true;
23              words = words + 1;
24          }
25          line[i] = toupper(ch);
26      }
27      cout << "characters: " << line.length() << endl;
28      cout << "words: " << words << endl;
29      cout << "upper case: " << line << endl;
30      return 0;
31  }
```

Typed at the program

```
Habari za asubuhi kutoka Siha
```

Output

```
Enter a line of text: characters: 29
words: 5
upper case: HABARI ZA ASUBUHI KUTOKA SIHA
```

Program 4.93: *One pass over the line counts the words and converts the characters, and length() supplies the character count without any counting at all.*

Why the reading uses getline: the extraction operator would have stopped at the first space and reported on the single word Habari. getline takes everything up to the newline, so the twenty nine characters and five words are all present to be counted.

The one design decision: words are counted by the transitions into a word rather than by the spaces. A flag records whether the walk is currently inside a word, and the count rises only where a non-space follows a space. Counting spaces and adding one would give the wrong answer for a line that begins or ends with a space, or holds two spaces together.

Why the conversion is safe: toupper is given the character as an unsigned char, which is the form it is defined for, and the result is written back into the same position. The string neither grows nor shrinks, and length() is asked afresh at the top of the loop, so no separate count is kept.

One walk of the line answers all three parts of the specification, which beats three tidy loops.

The Master Key: *getline preserves the spaces that make a line a line, and a word count is a count of entries into words, not a count of the spaces between them.*

BUILD Question 93

Two cards drop out of a dispensary record box and have to go back in alphabetical order. Compare first letters, then second, and the rule falls silent when one card runs out of letters.

Write a program that decides which of two words comes first in a dictionary, without using any library comparison.

Read two words typed on one line.

Decide which of them would come first in a dictionary.

Do not use strcmp, compare or any other library comparison.

Compare the words character by character.

Report how many characters matched before the words parted.

Model Answer

Program 4.94 walks both words together while the characters agree and neither has run out, then decides on whatever stopped the walk: differing characters are settled by the smaller one, and a word that ran out first is the shorter of two words sharing an opening and comes first.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      string one;
 8      string two;
 9      cout << "Enter two words: ";
10      cin >> one >> two;
11      size_t i = 0;
12      while (i < one.length() && i < two.length()
13             && one[i] == two[i])
14      {
15          i = i + 1;
16      }
17      string verdict = "the two words are the same";
18      if (i < one.length() && i < two.length())
19      {
20          verdict = (one[i] < two[i] ? one : two)
21                    + " comes first";
22      }
23      else if (one.length() < two.length())
24      {
25          verdict = one + " comes first";
26      }
27      else if (two.length() < one.length())
28      {
29          verdict = two + " comes first";
30      }
31      cout << "characters matched before they parted: " << i << endl;
32      cout << verdict << endl;
33      return 0;
34  }
```

Typed at the program

```
Mbozi Momba
```

Output

```
Enter two words: characters matched before they parted: 1
Mbozi comes first
```

Program 4.94: *Dictionary order worked out by hand. The loop stops at the first position where the two words differ, and the branch taken afterwards names which of the three cases occurred.*

The loop condition in three parts: the walk continues only while both words still have a character at that position and the two characters are equal. All three are needed: dropping either length test would read past the end of the shorter word.

The tie rule when one word is a prefix: if the walk ends because a word ran out, the two words agreed on every character the shorter one had. The shorter word comes first, which is why Mal precedes Malinyi in every dictionary ever printed. Only when both run out at the same position are the words the same word.

Why the comparison is a subtraction in disguise: comparing one[i] with two[i] compares the numeric codes of the two characters, which for the letters of the alphabet run in alphabetical order. This is the rule strcmp applies, and writing it out once shows why that function returns a signed value rather than a yes or a no.

Order is decided at the first position where two words disagree, and where there is none, by length.

The Master Key: *Dictionary order compares character by character to the first difference, and where one word is a prefix of the other the shorter word comes first.*

HOT Question 94

Should a student learn the hard way first? The question is asked about character arrays more often than about anything else here, usually by a teacher who learned that way and turned out perfectly capable.

Evaluate the claim that a student should learn C-style strings before std::string.

Model Answer

The claim is half right and the half that is wrong is expensive: char arrays should be learned, because nothing else shows what a string is made of, but they should be learned second and used almost never, because every program a candidate writes for marks is safer, shorter and quicker to write with std::string.

What char arrays teach that nothing else does: that a string is a run of bytes with a mark at the end, that the mark costs a byte, and that a function handed the address of the first byte knows nothing about the last. A candidate who has never met the terminator has no model of what a string is; the string class hides all of it, and hides it well.

The cost of teaching them first: the first exercises a class writes then involve fixed sizes, unchecked copies and comparisons that compile but compare addresses. Every one of those faults is silent, and a beginner cannot yet tell a silent fault from a working program. The teaching order manufactures the very defects it later warns about.

What a written answer actually asks for: a program that reads text, measures it, joins it and compares it, under time pressure and by hand on paper. Every one of those tasks is a single operator or a single member call on a std::string and several lines of careful arithmetic on a char array, and marks are lost in the arithmetic.

The job in hand	What to reach for	Why
text of unknown or changing length	std::string	it obtains the room it needs; an array cannot
joining, comparing, searching, cutting	std::string	one operator or one member call, with no size to work out
reading a whole line from the keyboard	std::string with getline	the line is kept entire and no buffer size is chosen
a fixed field of known width, as in a record layout	char array	the width is part of the format and never changes
passing text to a C library or an operating system call	char array, or c_str() on a string	those interfaces are defined in terms of char pointers
learning what a string is made of	char array	it is the only form in which the terminator is visible

Table 4.21: *Where each of the two forms belongs. The right-hand column is the reason in each case, and no row is decided by which form is older.*

Where the char array still belongs: in a fixed field whose width is part of a record format, and at the boundary where a C interface must be handed a char pointer. A std::string reaches that boundary with c_str(), so even there the array is a detail at the edge of a program rather than its ordinary way of holding text.

The judgement: teach std::string first and reach for it by default, then teach the char array as the representation underneath it, using the copy that overflows as the lesson rather than as an exercise. The condition that decides it is whether the text has a fixed width fixed by something outside the program; where it has not, the array is the wrong tool and the marks and the defects both follow.

Understand the array, then write the string, and keep the array for the places where a fixed width is genuinely part of the problem.

The Master Key: *Char arrays must be understood because they show what a string is made of, but std::string should be written by default, and the char array kept for fixed-width fields and C interfaces.*

HOT Question 95

A compiler that warns is not a language that forbids. The warning from Program 4.90 about writing eleven bytes into a region of size eight disappears when one flag is removed.

Evaluate the claim that buffer overflow is a historical problem which modern compilers have solved.

Model Answer

The claim is false, and the evidence is in this section: the diagnostic that caught the faulty copy appeared only because the optimiser was switched on, the abort that stopped the program came from the C library rather than from the language, and C++ itself still guarantees nothing about writing past the end of an array.

What the warning did catch: a copy whose source and destination were both visible in one function and both of known size. That is the easiest case there is, and the compiler handled it well, naming the byte counts, naming the object and pointing at the line.

What it did not catch: anything the compiler cannot see. Give strcpy a destination passed in as a parameter, or a source read from the keyboard, and the sizes are unknown until run time. Most real overflows have that shape, which is why they are still being found.

That it needed the optimiser: the same source compiled at the default level raises no diagnostic of any kind. The check depends on inlining and value tracking, neither of which happens until the optimiser is on, so a student building in the ordinary debug configuration sees nothing.

```
 1  #include <iostream>
 2  #include <cstring>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      char town[8] = "Butiama";
 8      char longer[] = "Ngorongoro";
 9      strcpy(town, longer);
10      cout << town << endl;
11      return 0;
12  }
```

Output

```
Ngorongoro
```

Program 4.95: *The same eleven lines as Program 4.90, compiled at the default optimisation level. No diagnostic is raised, the copy runs past the end of the array, and the program exits normally.*

Reading that output honestly: the program did not work. It wrote three bytes into memory it did not own, disturbed nothing the run needed, printed a town and returned zero. That is what a picture of the faulty program apparently working actually shows, and it is the most dangerous of the three outcomes.

The abort was a library guard, not a rule: the message came from the C library, which had been given the destination size by the compiler and checked the copy against it. That guard is one implementation adding a check of its own, switched on by build settings and switched off as easily. It is not part of C++.

What the language guarantees: nothing. Writing past the end of an array is undefined behaviour, which means the standard says nothing about what follows: the program may abort, may print, may corrupt an unrelated value, or may differ between two runs of one binary. No diagnostic is required.

The judgement: the claim fails, because what improved was the warnings and the guards around the language, not the language itself. What decides the risk is not the age of the compiler but whether a length can be decided at run time and copied without a check; where it can, the fault is as available now as ever.

Treat a diagnostic about a size as a gift and never as a guarantee, and write code in which the question of room cannot arise.

The Master Key: *Compilers and libraries have added warnings and run-time guards, but writing past the end of an array remains undefined behaviour that the language neither prevents nor reports.*

Classes, objects, encapsulation, constructors and destructors

This section is where a class stops being a diagram and becomes something that compiles. It sets out how a class is written and how much of it the outside world may touch, defines the three access specifiers including the one that is printed in every syntax box and explained in none, and then supplies the two member functions that run without ever being called by name: the constructor, which the common teaching uses in every worked example without once defining it, and the destructor, which does not appear in that teaching at all. The order in which those two run is not asserted here; a program prints it. Two listings are wrong on purpose, and one of them compiles cleanly, runs to the end and reports a stock of minus forty tablets.

BINDER Question 96

An afternoon at a computer laboratory in Itigi was lost to a punctuation mark that was not there. The class looked perfect, and the compiler complained about a function well below it that nobody had touched.

Explain how a class is defined in C++, and state the default access of a class and of a struct.

Model Answer

A class is defined by the keyword class, then the class name, then a body in braces holding the data members and the member functions, then a semicolon after the closing brace; members written before any specifier are private in a class, public in a struct.

The head and the body: the head is the word class and the name, written in PascalCase. The body holds the data members, which are the attributes, and the member functions, which are the operations. Nothing here reserves memory; it describes a kind of thing.

The semicolon that is not optional: a class definition is a declaration and ends with a semicolon after the closing brace, at line 16. Leave it out and the compiler reads what follows as part of the same declaration, then complains about a line further down in correct code.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  class Ticket
 6  {
 7  private:
 8      string route;
 9      int fare;
10  public:
11      void setDetails(string r, int f);
12      void report()
13      {
14          cout << route << " costs " << fare << endl;
15      }
16  };
17
18  void Ticket::setDetails(string r, int f)
19  {
20      route = r;
21      fare = f;
22  }
23
24  int main()
25  {
26      Ticket morning;
27      morning.setDetails("Itigi to Iramba", 4500);
28      morning.report();
29      return 0;
30  }
```

Output

```
Itigi to Iramba costs 4500
```

Program 4.96: *One class, two members and two member functions, with report defined inside the body and setDetails defined outside it. Line 11 declares that function and line 18 supplies it.*

Defining a member function outside the class: declare it in the body, as line 11 does, and define it later with the class name and the scope resolution operator before the function name, as line 18 does. Without that prefix it is a global function with no access to route or fare.

The default that is never stated: members of a class are private until a specifier says otherwise, and members of a struct are public until one does. That is the whole difference: a struct may carry member functions, constructors and destructors exactly as a class may, and exercises turn on this default constantly.

What the default refuses: fare in Program 4.97 sits after the word private, so main may not read it. The compiler rejects the line, names the member, points at the declaration that made it private, and produces no program.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  class Ticket
 5  {
 6  private:
 7      int fare;
 8  public:
 9      void setFare(int f)
10      {
11          fare = f;
12      }
13  };
14
15  int main()
16  {
17      Ticket morning;
18      morning.setFare(4500);
19      cout << morning.fare << endl;
20      return 0;
21  }
```

Compiler says

```
error: 'int Ticket::fare' is private within this context
note: declared private here
```

Program 4.97: *The same class with the fare read directly from main. The member is private, so this listing does not compile.*

Head, body, members, closing brace, semicolon; and with no specifier written, a class keeps everything private.

The Master Key: *A class definition is the keyword, the name, a braced body and a semicolon; a member function may be defined outside with the scope resolution operator; and members are private by default in a class, public by default in a struct.*

BINDER Question 97

Copy out any syntax box for a class and three words come with it: public, private and protected. Two of them are then explained, and the third is left undefined.

Explain the three access specifiers and state exactly what each one permits.

Model Answer

A public member may be used anywhere the object is visible, a private member only inside the member functions of its own class, and a protected member inside those and inside the member functions of any class derived from it, but never from outside.

How far a specifier reaches: the word is followed by a colon on its own line and governs every member from there until the next specifier or the closing brace. A class may carry as many as it likes in any order, and Program 4.98 carries four.

What public permits: everything. It is the interface, the part a caller is invited to use. Line 30 reads serial from main because serial is public, and any code holding a Meter may do the same.

What private permits: the member functions of the same class and nothing else. report reads calibration at line 22 because report is a member of Meter; main cannot read it at all. Access is granted to the class rather than the object, so a member function may reach the private members of any object of its own class.

What protected permits: exactly what private permits, plus the member functions of any class derived from this one. From outside it is as closed as private, and in Figure 4.22 the protected row differs from the private row in one cell only. Use it only where a class you mean to derive from must hand something down.

declared	outside the class	in a member function	in a derived class
public	✓	✓	✓
protected	✗	✓	✓
private	✗	✓	✗

✓ compiles ✗ rejected by the compiler

Figure 4.22: *What each access specifier permits, seen from three places. Every cell was settled by compiling that exact access, not by recollection.*

Access is settled before the program runs: the compiler checks every reach and rejects the offending line, so a private member costs nothing at run time: the machine code that reads it is the code that would have read a public one. Access control protects the program from the programmer.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  class Meter
 5  {
 6  public:
 7      int serial;
 8  protected:
 9      int lastReading;
10  private:
11      int calibration;
12  public:
13      void setAll(int s, int r, int c)
14      {
15          serial = s;
16          lastReading = r;
17          calibration = c;
18      }
19      void report()
20      {
21          cout << "from a member function: " << serial << ", "
22               << lastReading << ", " << calibration << endl;
23      }
24  };
25
26  int main()
27  {
28      Meter kiosk;
29      kiosk.setAll(4417, 6200, 3);
30      cout << "from main, serial: " << kiosk.serial << endl;
31      kiosk.report();
32      return 0;
33  }
```

Output

```
from main, serial: 4417
from a member function: 4417, 6200, 3
```

Program 4.98: *One class carrying all three specifiers. The member function reads all three members; main reads only the public one.*

Make the data private and the operations public: the rest is the class's own business.

The Master Key: *public is open to everything, private only to the member functions of its own class, and protected to those and to derived classes, all three enforced by the compiler at no cost at run time.*

BINDER Question 98

A savings cooperative at Iramba fills in the number, date and opening balance before a new card leaves the clerk's desk. C++ has a member function that does the same, and your program never calls it by name.

Explain what a constructor is, name the kinds, and state the rules that govern them.

Model Answer

A constructor is a member function carrying the same name as the class and no return type at all, not even void, which the compiler calls automatically at the moment an object is created, so that the object begins its life with its members already set.

The rules that fix its form: the name must match the class exactly, and there is no return type, so writing void in front of it is an error rather than a matter of taste. It may take parameters and may be overloaded, which Section 8 settles.

The default constructor: the one that takes no arguments. Line 29 declares an object with no argument list, so that constructor runs and leaves the object holding the values it chose. A default constructor is any constructor callable with no arguments, yours or the compiler's.

The parameterised constructor: the one that takes the opening values as arguments, written in brackets after the object name at line 30. Two constructors are told apart by their parameter lists, so a class may offer several ways of being started.

The member initialiser list: a colon after the header, then each member with its value in brackets, separated by commas, as at line 17. It gives a member its value as it comes into existence, where the body can only assign to one that already exists. For a constant member it is the only route.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  class Account
 6  {
 7  private:
 8      string holder;
 9      int balance;
10  public:
11      Account()
12      {
13          holder = "unnamed";
14          balance = 0;
15          cout << "default constructor ran" << endl;
16      }
17      Account(string h, int opening) : holder(h), balance(opening)
18      {
19          cout << "parameterised constructor ran" << endl;
20      }
21      void report()
22      {
23          cout << holder << " holds " << balance << endl;
24      }
25  };
26
27  int main()
28  {
29      Account blank;
30      Account named("Asha", 30000);
31      blank.report();
32      named.report();
33      return 0;
34  }
```

Output

```
default constructor ran
parameterised constructor ran
unnamed holds 0
Asha holds 30000
```

Program 4.99: *Two constructors in one class, the first taking nothing and the second taking a name and an opening balance through a member initialiser list.*

Declare one and the free one disappears: the compiler supplies a default constructor only while the class declares none at all. Program 4.100 declares one taking an int, so the free one is gone, and an object declared with no arguments draws a complaint that there is no matching function.

The kind	What it takes	When it runs
Default constructor	Nothing at all	Whenever an object is created with no argument list
Parameterised constructor	The opening values	Whenever an object is created with a matching argument list
The one the compiler supplies	Nothing at all	Only while the class declares no constructor of its own

Table 4.22: *The kinds of constructor, what each one takes and the moment each one runs. The third row stops applying the instant any constructor of your own is declared.*

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  class Fare
 5  {
 6  private:
 7      int amount;
 8  public:
 9      Fare(int given)
10      {
11          amount = given;
12      }
13      void report()
14      {
15          cout << amount << endl;
16      }
17  };
18
19  int main()
20  {
21      Fare item;
22      item.report();
23      return 0;
24  }
```

Compiler says

```
error: no matching function for call to 'Fare::Fare()'
note: candidate: 'Fare::Fare(int)'
note:   candidate expects 1 argument, 0 provided
```

Program 4.100: *A class with one constructor taking an argument, and an object declared with none. The compiler no longer supplies the missing one.*

Name it after the class, give it no return type, and let it make the object sensible before use.

The Master Key: *A constructor shares the name of its class, has no return type, runs automatically when an object is created, comes in a default and a parameterised form, and is no longer supplied by the compiler once you declare one of your own.*

BINDER Question 99

Two objects are made in a block, one after the other, and by the closing brace both are gone. Which one goes first, and does it matter when there is a file to close?

Explain what a destructor is, when it runs, and in what order.

Model Answer

A destructor is a member function whose name is the class name with a tilde in front, taking no parameters and returning nothing, of which a class has at most one, and which the compiler calls when the object is destroyed; objects in a scope are destroyed in the exact reverse of their construction.

The form, and why there can be only one: no parameters means no parameter list to tell two apart, so a destructor cannot be overloaded and a class carries at most one. It has no return type either, and like a constructor it is called for you.

The moment it runs: for a local object, at the closing brace of the block it was declared in. The inner block of Program 4.101 ends at line 26 and beans is destroyed there, before the next line runs. The object maize lasts until main ends.

Reverse order, and the reason for it: the last object built is the first destroyed. It is the only safe order, because an object built later may hold something belonging to one built earlier, and taking the earlier apart first would leave the later holding nothing.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  class Ledger
 6  {
 7  private:
 8      string name;
 9  public:
10      Ledger(string n) : name(n)
11      {
12          cout << name << " constructed" << endl;
13      }
14      ~Ledger()
15      {
16          cout << name << " destroyed" << endl;
17      }
18  };
19
20  int main()
21  {
22      Ledger maize("maize");
23      {
24          Ledger beans("beans");
25          cout << "inner block is about to end" << endl;
26      }
27      cout << "main is about to end" << endl;
28      return 0;
29  }
```

Output

```
maize constructed
beans constructed
inner block is about to end
beans destroyed
main is about to end
maize destroyed
```

Program 4.101: *Two objects, one of them inside an inner block, each printing from its constructor and from its destructor. The order in the output is evidence, not assertion.*

Where a member object fits: a member object is fully constructed before the body of its container's constructor begins, and destroyed after the container's destructor has finished. The reverse rule therefore holds twice: between objects in a scope, and between an object and its parts.

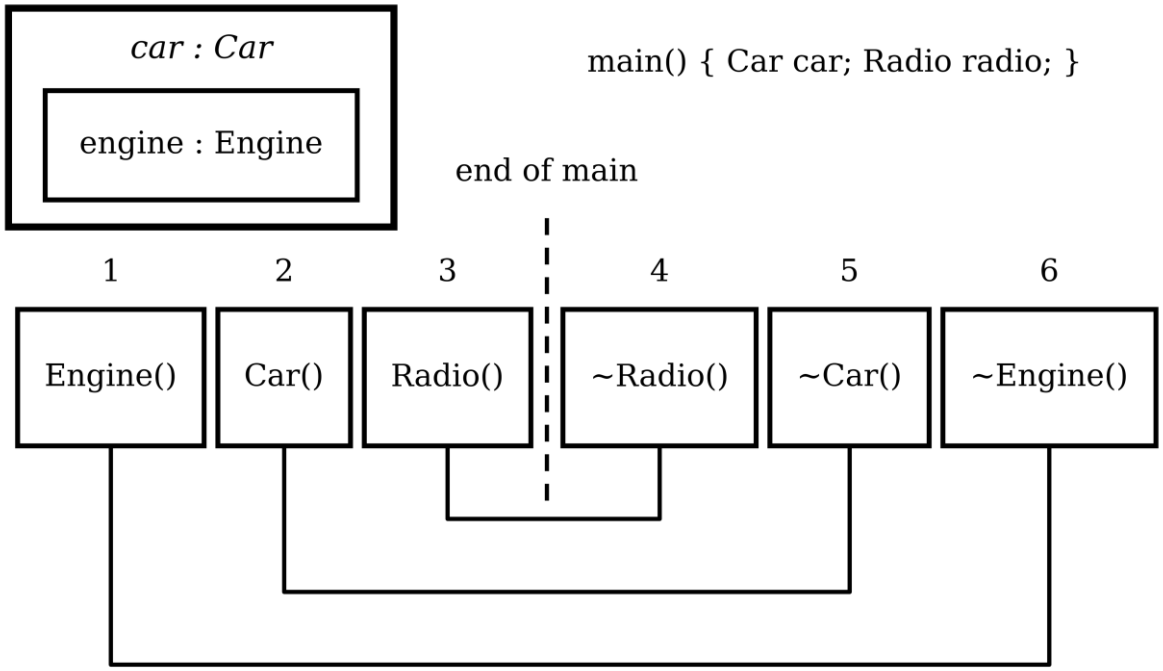


Figure 4.23: *Construction and destruction across one scope. Objects are destroyed in exactly the reverse of the order in which they were built, and a member is built before the object that contains it.*

When to write one at all: write one where the class holds something that must be given back, such as memory obtained with new, an open file or a lock. A class holding only ordinary members needs none, because the compiler's destructor already destroys every member.

Objects close in the reverse of the order they opened, and the destructor is where that closing is written.

The Master Key: *A destructor carries a tilde before the class name, takes nothing, returns nothing, occurs at most once per class, runs at the closing brace of the scope holding the object, and runs in the exact reverse of construction order.*

REAL Question 100

Not once has the right-hand column of the stock book at a pharmacy in Tunduru held a negative number. The program written to replace that book compiles without a warning and reports a stock of minus forty tablets.

The class below is offered as an example of encapsulation. Explain why it is not, and correct it. Name the fault, state exactly what the compiler does about it, and give a corrected version.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  class StockItem
 6  {
 7  public:
 8      string name;
 9      int quantity;
10      void report()
11      {
12          cout << name << ": " << quantity << " in stock" << endl;
13      }
14  };
15
16  int main()
17  {
18      StockItem tablets;
19      tablets.name = "paracetamol";
20      tablets.quantity = 120;
21      tablets.report();
22      tablets.quantity = -40;
23      tablets.report();
24      return 0;
25  }
```

Output

```
paracetamol: 120 in stock
paracetamol: -40 in stock
```

Program 4.102: *The class offered as an example of encapsulation, with the run it produced.*

Model Answer

The class is not encapsulated, because its data members are declared public: main reaches straight past the report function and writes minus forty into quantity at line 22, and the compiler accepts it without comment, because the keyword class was written but no access was ever controlled.

Naming the fault: line 7 says public and lines 8 and 9 are the data. Every StockItem in the program therefore offers its name and its quantity to any code that can see the object. The class has bundled the data with a function that acts on it and has then controlled nothing, and control is the half that was missing.

What the compiler does about it: nothing whatsoever. Not an error, not a warning, not a note. The assignment on line 22 is a perfectly legal assignment of an int to a public int, so the fault passes compilation, passes the run, and shows up only in a printed line that a pharmacist would query.

Why writing the keyword is not the thing: encapsulation is bundling data with the functions that act on it and controlling reach into that data. Program 4.102 does the first and skips the second, so what it demonstrates is a record with a function attached. A common error is to treat the keyword as the whole of the idea.

The correction: put the data behind private, give the class a constructor so that no StockItem can exist without an opening quantity, and put the rule inside the operation: issue tests the request against the stock before it changes anything, and returns without touching the stock when the test fails.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  class StockItem
 6  {
 7  private:
 8      string name;
 9      int quantity;
10  public:
11      StockItem(string n, int opening) : name(n), quantity(opening)
12      {
13          cout << name << " opens at " << quantity << endl;
14      }
15      void issue(int wanted)
16      {
17          if (wanted > quantity)
18          {
19              cout << "refused, only " << quantity << " left" << endl;
20              return;
21          }
22          quantity = quantity - wanted;
23          cout << wanted << " issued, " << quantity << " left" << endl;
24      }
25  };
26
27  int main()
28  {
29      StockItem tablets("paracetamol", 120);
30      tablets.issue(160);
31      tablets.issue(20);
32      return 0;
33  }
```

Output

```
paracetamol opens at 120
refused, only 120 left
20 issued, 100 left
```

Program 4.103: *The same model with the data private, a constructor setting the opening stock, and a guard inside issue. The request for 160 is refused and the stock is left where it was.*

Reading the corrected output: the request for 160 is refused and the stock stays at 120; the following request for 20 succeeds and leaves 100. The impossible value is not merely discouraged now. There is no longer any line that could be written, anywhere in the program, that would produce it.

Why the guard needs the wall: you could have written the same test into Program 4.102, and any caller could have ignored it and assigned to quantity as before. A rule inside a function is worth having only when the direct route around it has been closed, which is exactly what the word private does.

Writing the keyword class is not encapsulation; deciding who may reach the data, and refusing everybody else, is.

The Master Key: *Public data members leave a class open to any assignment, and the compiler reports nothing, so encapsulation is achieved by making the data private and putting every rule about it inside the member functions that are now the only way in.*

REAL Question 101

The water kiosk at Buhigwe has its meter bolted inside the cabin, fitted before the kiosk opened and taken out after it closed for good. A program holding one object inside another keeps the same order.

Trace the program below and state what is printed and in what order.

```
 1  #include <iostream>
 2  using namespace std;
 3  struct Meter
 4  {
 5      Meter()
 6      {
 7          cout << "Meter built" << endl;
 8      }
 9      ~Meter()
10      {
11          cout << "Meter destroyed" << endl;
12      }
13  };
14  class Kiosk
15  {
16  private:
17      int number;
18      Meter meter;
19  public:
20      Kiosk(int n) : number(n)
21      {
22          cout << "Kiosk " << number << " built" << endl;
23      }
24      ~Kiosk()
25      {
26          cout << "Kiosk " << number << " destroyed" << endl;
27      }
28  };
29  int main()
30  {
31      Kiosk first(1);
32      Kiosk second(2);
33      return 0;
34  }
```

Output

```
Meter built
Kiosk 1 built
Meter built
Kiosk 2 built
Kiosk 2 destroyed
Meter destroyed
Kiosk 1 destroyed
Meter destroyed
```

Program 4.104: *A Meter held as a member of a Kiosk, and two Kiosk objects in one scope. No constructor or destructor is called by name.*

Model Answer

Eight lines are printed: the meter belonging to first, then first itself, then the meter belonging to second, then second; and at the closing brace of main, second is destroyed, then the meter inside it, then first, then the meter inside first.

Why the meter is built first: an object cannot be complete until its parts exist, so every member object is constructed before the body of the container's constructor starts running. The line printed by the constructor of Meter therefore arrives before the line printed by the constructor of Kiosk, every time.

Why the second kiosk is destroyed first: the objects in a scope are destroyed in the reverse of the order in which they were constructed, with no exceptions. The object second was declared at line 32 and first at line 31, so second goes at the closing brace of main before first does.

Why a meter is destroyed after its own kiosk: the container's destructor body runs first and its members are destroyed afterwards, which is the mirror image of construction. So the reverse rule holds at both levels at once: between the two kiosks, and between each kiosk and the meter inside it.

Order	The event	The object
1	Constructor runs	the meter inside first
2	Constructor runs	first
3	Constructor runs	the meter inside second
4	Constructor runs	second
5	Destructor runs	second
6	Destructor runs	the meter inside second
7	Destructor runs	first
8	Destructor runs	the meter inside first

Table 4.23: *The eight events of Program 4.104 in the order they happen. Read the halves against each other: the fifth event undoes the fourth, the sixth undoes the third, and so on to the last.*

What the initialiser list is doing: line 20 sets number in the initialiser list, and says nothing about meter, so meter is built by the constructor of Meter that takes no arguments. Had Meter offered no such constructor, line 20 would have been obliged to name it and hand it something, which is one of the jobs an initialiser list exists for.

The trap in reading this trace: nothing in main calls anything. The last statement anybody wrote is on line 33, and four of the eight printed lines arrive after it. A common error in a trace of this kind is to stop writing at the end of the visible statements and lose the four lines that were never hidden.

Parts before wholes on the way in, wholes before parts on the way out, and the whole scope unwound backwards.

The Master Key: *A member object is constructed before its container and destroyed after it, and objects in one scope are destroyed in the exact reverse of their construction, so the last four lines of this trace are the first four read backwards.*

BUILD Question 102

At the top of a savings group ledger in Wanging'ombe, one rule: money paid out may never carry the total below zero. A program has nobody reading over its shoulder before it writes.

Write a class that models a savings account, with a private balance, a constructor that sets the opening balance, and deposit and withdraw operations that refuse an impossible request.

Hold the balance as a private data member.

A constructor sets the opening balance when the object is declared.

deposit adds an amount to the balance and reports the new balance.

withdraw pays out only when the balance is large enough for the request.

A refused withdrawal must leave the balance exactly as it was.

Model Answer

Program 4.105 keeps balance private at line 6, sets it from the constructor at line 8 so that no account exists without an opening figure, and puts the whole rule inside withdraw, which tests the request against the balance at line 18 before it changes anything and returns untouched when the test fails.

```
 1  #include <iostream>
 2  using namespace std;
 3  class SavingsAccount
 4  {
 5  private:
 6      int balance;
 7  public:
 8      SavingsAccount(int opening) : balance(opening)
 9      {
10      }
11      void deposit(int amount)
12      {
13          balance = balance + amount;
14          cout << "deposit, balance now " << balance << endl;
15      }
16      void withdraw(int amount)
17      {
18          if (amount > balance)
19          {
20              cout << "refused, balance is " << balance << endl;
21              return;
22          }
23          balance = balance - amount;
24          cout << "paid out, balance now " << balance << endl;
25      }
26  };
27  int main()
28  {
29      SavingsAccount asha(50000);
30      asha.deposit(20000);
31      asha.withdraw(90000);
32      asha.withdraw(30000);
33      return 0;
34  }
```

Output

```
deposit, balance now 70000
refused, balance is 70000
paid out, balance now 40000
```

Program 4.105: *A savings account whose rule about withdrawals is written once, inside the class, where no caller can go around it.*

Where the rule lives: the test sits inside the class, so it applies to every call from anywhere: from main, from another function, from inside a loop. Had the balance been public, the rule would have had to be repeated at every place a withdrawal was made, and forgotten at one of them.

Refusing without changing anything: the guard prints its refusal and returns before the subtraction is reached, so the balance is not touched on a failed request. The refused withdrawal of 90000 leaves the balance at 70000, and the request for 30000 that follows is paid out of that same 70000.

The one thing left to add: deposit takes whatever it is handed, including a negative amount, which would quietly work as a withdrawal without a guard. A real account tests it with an identical if, and that is the only gap between this listing and the account a bank would accept.

The balance is private and the only doors into it are two functions that know the rule, which is why the rule cannot be avoided.

The Master Key: *Put the data behind private and the rule inside the operation, and refuse by returning before anything is changed, so that a rejected request leaves the object exactly as it was.*

BUILD Question 103

Three marks, one name, one grade: the smallest report a school produces. Written as three loose variables it serves one student and falls apart at the second.

Write a class that models a student, holding a name and three marks, with a constructor and a method that reports the total, the mean and the grade.

Hold the name and the three marks as private data members.

Use std::string for the name.

A constructor takes the name and all three marks.

One member function reports the total, the mean and the grade.

Grade bands: 80 and above A, 60 to 79 B, 50 to 59 C, 40 to 49 D, below 40 F.

Model Answer

Program 4.106 holds the name and an array of three marks privately, takes all four values through the constructor at line 11 so that no half filled Student can exist, and works out the total, the mean and the grade inside report, where the grade rule is written once for every Student ever made.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  class Student
 6  {
 7  private:
 8      string name;
 9      int marks[3];
10  public:
11      Student(string n, int a, int b, int c) : name(n)
12      {
13          marks[0] = a;
14          marks[1] = b;
15          marks[2] = c;
16      }
17      void report()
18      {
19          int total = marks[0] + marks[1] + marks[2];
20          string letters = "FFFFDCBBAAA";
21          char grade = letters[total / 30];
22          cout << name << ": total " << total << ", mean "
23               << total / 3.0 << ", grade " << grade << endl;
24      }
25  };
26  int main()
27  {
28      Student asha("Asha Mkumbo", 78, 64, 83);
29      Student juma("Juma Kilonzo", 41, 38, 44);
30      asha.report();
31      juma.report();
32      return 0;
33  }
```

Output

```
Asha Mkumbo: total 225, mean 75, grade B
Juma Kilonzo: total 123, mean 41, grade D
```

Program 4.106: *A student class with the grade rule inside it. Two objects are declared, and each is asked for its own report.*

Why the constructor takes all four values: there is then no moment between the declaration and the first use at which a Student has a name and no marks. The object is correct from its first instant, which is the whole reason a constructor exists rather than four separate setters.

The one design decision worth naming: the grade is found by indexing a string of eleven letters with the total divided by thirty, because the tens digit of the mean is the total divided by thirty. For Asha the total is 225, so the index is 7 and the letter is B. Five bands, one line, and no ladder of else if to write out and get wrong.

Reading the second line of output: Juma totals 123, so the mean is 41, the index is 4 and the grade is D. Nothing in main knows any of that. It declares two students and asks each object for its report, which is the abstraction the class was made for.

One class, one grade rule, and as many students as the program needs.

The Master Key: *A constructor that takes every value the object needs leaves no half filled object, and a rule written once inside a member function applies to every object of that class.*

BUILD Question 104

Maize weighed in at a cooperative store near Nanyumbu arrives grower by grower, and at the end of the season somebody has to say who brought the most. Nobody reads the ledger twice.

Write a program that keeps a small collection of objects and reports over it.

Hold a grower's name and the sacks delivered as private data members.

A constructor sets both when the object is made.

Keep three growers in one array of objects.

Walk the array with a loop and report the grower who delivered the most.

Model Answer

Program 4.107 declares an array of three Grower objects and hands each one its name and its sacks through the constructor in the very braces that declare the array, then walks it once from the second element, keeping the index of the largest delivery so far, and asks that object to report itself.

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4  class Grower
 5  {
 6      string name;
 7      int sacks;
 8  public:
 9      Grower(string n, int brought) : name(n), sacks(brought)
10      {
11      }
12      int getSacks()
13      {
14          return sacks;
15      }
16      void report()
17      {
18          cout << name << " brought " << sacks << " sacks" << endl;
19      }
20  };
21  int main()
22  {
23      Grower book[3] = {{"Baraka", 148}, {"Neema", 205}, {"Anna", 96}};
24      int best = 0;
25      for (int i = 1; i < 3; i++)
26      {
27          if (book[i].getSacks() > book[best].getSacks())
28          {
29              best = i;
30          }
31      }
32      book[best].report();
33      return 0;
34  }
```

Output

```
Neema brought 205 sacks
```

Program 4.107: *Three objects in one array, filled through the constructor and searched with a single loop for the largest delivery.*

How an array of objects is filled: the inner braces at line 23 hand each pair of values to the constructor taking two arguments, so all three objects are complete before the array exists. Without such a constructor each element would have to be filled afterwards through a setter, and there would be a moment when every element was blank.

The loop that finds the highest: take element 0 as the best so far and compare from element 1 onwards, replacing best whenever a larger delivery appears. Keep the index rather than the number: keeping 205 alone would tell you the size of the largest delivery and not which object it belongs to, and the report is wanted from the object.

A detail on lines 6 and 7: those two members carry no access label and are private all the same, because that is what a class does before any specifier appears. The loop never touches them directly; it asks through getSacks, which is the only reason it compiles.

One array, one pass, and the winning object answers for itself.

The Master Key: *An array of objects is filled by handing each element its values through the constructor, and a single pass keeping the index of the best so far lets the winning object report itself.*

HOT Question 105

Usually it is the sharper student who objects: if setBalance only assigns its argument to balance, it is a public member with a longer name to type. That holds until the function is asked to do anything else.

Evaluate the claim that accessor and mutator functions are pointless, because a mutator that simply assigns to a private member is the same as making the member public. Reach a judgement.

Model Answer

The claim is right about the case it describes and wrong as a general rule: a mutator that does nothing but assign is no better than a public member, but the moment it can refuse a value, or the representation might change, or a member is to be read and never written, the private member does what no public member can.

The guard a mutator can carry: setReading in Program 4.108 refuses any reading below the one already recorded, because a water meter cannot run backwards. The refusal sits inside the class, so it holds for every caller that will ever be written. Assignment to a public member offers nowhere at all to put such a test.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Meter
 4  {
 5  private:
 6      int serial;
 7      int reading;
 8  public:
 9      Meter(int s, int start) : serial(s), reading(start)
10      {
11      }
12      int getSerial()
13      {
14          return serial;
15      }
16      void setReading(int n)
17      {
18          if (n < reading)
19          {
20              cout << "refused, a meter cannot fall" << endl;
21              return;
22          }
23          reading = n;
24          cout << "reading now " << reading << endl;
25      }
26  };
27  int main()
28  {
29      Meter kiosk(4417, 6200);
30      cout << "serial " << kiosk.getSerial() << endl;
31      kiosk.setReading(6350);
32      kiosk.setReading(6100);
33      return 0;
34  }
```

Output

```
serial 4417
reading now 6350
refused, a meter cannot fall
```

Program 4.108: *A mutator that can refuse, and an accessor with no mutator beside it. The serial can be read and never written; the reading can be raised and never lowered.*

Freedom to change what is behind the wall: a class reporting a total may hold three marks and add them, or hold a running total and return it. Both answer the same call. With the data public every caller names the members, so changing that choice means editing every caller, and one will be missed.

The member that is read and never written: serial has an accessor and no mutator, so it is readable from outside and fixed once the constructor has run. Making a member public grants reading and writing together, with no way to hand over only one.

The case the objection gets right: a class in which every member has a one line getter and a one line setter, none of them testing anything, has bought nothing and made the class longer to read. That pattern is common in submitted work, and the objection describes it exactly.

The member function	What it does beyond assigning or returning	Does it earn its place
setReading	Refuses a value that would be nonsense	Yes, and it is the only place that rule can live
getSerial	Offers a member for reading and never for writing	Yes, a public member cannot be made read only
A total that may later be stored rather than added up	Hides which of the two the class actually does	Yes, no caller changes when the choice changes
A setter that only assigns to its member	Nothing at all	No, and this is the case the objection describes

Table 4.24: *Four member functions of the same outward shape, and the test that separates them. Only the last could be replaced by a public data member without anything being lost.*

The test that separates the two cases: ask what the class would refuse. If there is no value the mutator would reject, no other member the assignment has to be kept in step with, and no chance of the representation changing, then the function is a pass through and the objection stands for it.

The judgement: keep the data private always, write an accessor only where something outside needs the value, and write a mutator only where it guards a value, keeps two members in step or hides a representation. The condition that decides it is whether the class has any rule about the value: where it has one, the mutator is the only place that rule can be enforced; where it has none, you are looking at a plain bundle of values and should write it as one.

The question is never whether to write a setter, but whether the setter has anything to say when it is called.

The Master Key: *A mutator that only assigns is a public member in disguise, but one that refuses a value, keeps members in step or hides a representation, and an accessor with no mutator beside it, do work no public data member can do.*

HOT Question 106

A small type with two ints, written to carry a pair of numbers between functions, came back marked with one comment: this needs a constructor. The rule is very nearly right.

A candidate argues that every class should have a constructor, whatever the class. Evaluate this and reach a judgement.

Model Answer

The claim is nearly always sound advice and wrong as a universal rule: a constructor belongs wherever the class has a rule about its members that must hold from the first instant, which is almost every class worth writing, but a plain public bundle of values needs none, because braces do that work.

What an uninitialised member actually holds: nothing was written into it, so it holds whatever bytes that memory already carried. Its value is indeterminate and reading it is undefined behaviour: the program may print a plausible number, a wild one, or a different one on each run. No listing here prints one.

The constructor the compiler supplies, and when it vanishes: declare no constructor and the compiler supplies one. It is easy to assume that this one clears the members, and it does not: for members of built-in type it does nothing, leaving them indeterminate. Declare one of your own and even that is gone, as Program 4.100 showed.

The aggregate that genuinely does not need one: the Reading in Program 4.109 is two public ints with no rule tying them together. The braces at line 26 set both at the declaration, and the empty braces at line 27 set both to zero. A constructor here would only repeat what the braces already say.

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  struct Reading
 5  {
 6      int hour;
 7      int litres;
 8  };
 9
10  class Tank
11  {
12  private:
13      int litres;
14  public:
15      Tank(int start) : litres(start)
16      {
17      }
18      void report()
19      {
20          cout << "tank holds " << litres << endl;
21      }
22  };
23
24  int main()
25  {
26      Reading morning = {7, 340};
27      Reading blank = {};
28      Tank tank(1200);
29      cout << morning.hour << " " << morning.litres << endl;
30      cout << blank.hour << " " << blank.litres << endl;
31      tank.report();
32      return 0;
33  }
```

Output

```
7 340
0 0
tank holds 1200
```

Program 4.109: *A plain aggregate initialised at its declaration, beside a class whose private member is set by a constructor. Both objects are fully determined; only one of them needed a constructor to get there.*

What the constructor buys where there is a rule: the litres in Tank is private and must always be a known figure. The constructor is the only door in, so no Tank exists that has not passed through it. That guarantee is the thing being bought, and careful initialising at each declaration cannot supply it.

The honest cost of the rule as stated: applied without thought, it produces constructors that copy a parameter into a member and do nothing else, the same trivial pass throughs the previous question refused to admire. A rule that produces ceremony teaches students to write ceremony.

The judgement, and the condition that decides it: write a constructor wherever the class has private data or a rule about its members that must hold before it is first used, which is very nearly every class; omit it only for a public aggregate of independent values, initialised with braces where it is declared so that no member is left indeterminate. The deciding question is not whether the thing is called a class but whether a rule has to hold from the first instant.

Follow the rule in practice, but for its reason, because the reason also names the case where you may drop it.

The Master Key: *A constructor exists to make an object correct before anyone can see it, so write one wherever the class has private data or a rule to enforce, and drop it only for a public aggregate initialised with braces where it is declared.*

Inheritance, polymorphism and abstraction

This section closes the chapter with the two principles that cannot exist until there is a base class to build on. It sets out how a derived class is written and what public, protected and private inheritance each do to the members

that come down, settles how a base class is initialised and in what order constructors and destructors run, and then defines polymorphism in prose before showing it in code, which is the reverse of the usual order and the reason so many candidates recognise a virtual function without being able to say what it is for. Multiple inheritance is shown working, because C++ has it; the diamond that two shared bases create is shown failing to compile, and then repaired twice over. Four listings are wrong on purpose, and the worst of them compiles without a single diagnostic, runs to the end, and quietly abandons half of the object it was told to destroy.

BINDER Question 107

Two shelves of accession cards in a secondary school library in Kibara, one for books and one for periodicals, carry the same first four lines on every card. A volunteer typed them twice for a whole term.

Explain how a derived class is written in C++, and state what public, protected and private inheritance each do to the inherited members.

Model Answer

A derived class is written by putting a colon after its name, then an access specifier, then the name of the base, as in class Book : public Item, and it acquires every member of the base except the constructors and the destructor, which are never inherited.

The form of the declaration: line 17 of Program 4.110 reads class Book : public Item, and that is the whole of it. Leave the specifier out and a class inherits privately, a struct publicly, so write the word every time rather than trusting a default nobody remembers.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Item
 4  {
 5  private:
 6      int accession;
 7  protected:
 8      int shelf;
 9  public:
10      void receive(int a, int s)
11      {
12          accession = a;
13          shelf = s;
14          cout << "item " << accession << " received" << endl;
15      }
16  };
17  class Book : public Item
18  {
19  public:
20      void card()
21      {
22          cout << "book stands on shelf " << shelf << endl;
23      }
24  };
25  int main()
26  {
27      Book atlas;
28      atlas.receive(3140, 7);
29      atlas.card();
30      return 0;
31  }
```

Output

```
item 3140 received
book stands on shelf 7
```

Program 4.110: *A derived class written in one line. Book declares no members of its own and already has the function on line 10 and the member on line 8.*

What comes down and what does not: every member of the base becomes a member of the derived class, so line 28 calls receive although Book never declared it. Constructors and destructors stay behind: the base constructor still runs, but a derived class must write its own.

What the specifier changes: it changes the access those inherited members have inside the derived class, and nothing else. Nothing is ever promoted: a protected member cannot become public by being inherited publicly, and Figure 4.24 has no cell in which access widens.

member in the base	public inheritance	protected inheritance	private inheritance
public	public	protected	private
protected	protected	protected	private
private	inaccessible	inaccessible	inaccessible

what the derived class ends up with

Figure 4.24: *What each kind of inheritance does to the members it inherits. Every cell was settled by compiling that exact access, not by recollection.*

The row that never moves: a private member of the base is inaccessible in the derived class under all three, which is the bottom row of Figure 4.24 and the reason Program 4.111 does not compile. The compiler names accession, points at line 6 and stops.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Item
 4  {
 5  private:
 6      int accession;
 7  protected:
 8      int shelf;
 9  public:
10      void receive(int a, int s)
11      {
12          accession = a;
13          shelf = s;
14      }
15  };
16  class Book : public Item
17  {
18  public:
19      void card()
20      {
21          cout << accession << " on shelf " << shelf << endl;
22      }
23  };
24  int main()
25  {
26      Book atlas;
27      atlas.receive(3140, 7);
28      atlas.card();
29      return 0;
30  }
```

Compiler says

```
error: 'int Item::accession' is private within this context
note: declared private here
```

Program 4.111: *The same pair with card reading the private member of the base at line 21. The listing does not compile.*

What protected exists for: shelf on line 8 is protected, so line 22 reads it and no code outside the family can. That is the purpose of the third specifier: open to derived classes, shut to everyone else, and it is why a base meant to be built on rarely makes everything private.

Write the colon, write public unless you have a reason not to, and put behind protected whatever a derived class will need.

The Master Key: *A derived class is written class Derived : public Base; constructors and destructors are not inherited; public inheritance leaves the access of each member as it was, protected and private inheritance lower it, and a private base member is out of reach under all three.*

BINDER Question 108

The four lines came out in the wrong order, and a class in Katesh spent twenty minutes arguing about which one was wrong. Nobody had written a single call to a constructor or a destructor.

Explain how a base class is initialised when a derived object is created, and state the order of construction and destruction.

Model Answer

The base class is initialised by naming it in the member initialiser list of the derived constructor; if it is not named there the compiler calls the default constructor of the base, and if the base has none the listing does not compile. The base part is built first and destroyed last.

Naming the base in the list: line 18 of Program 4.112 reads Book(int a, int e) : Item(a), so the base name stands in the list with its arguments in brackets. That list is the only place a base constructor may be called; there is no way to call it from the body.

The call the compiler writes for you: leave the base unnamed and the compiler still calls a base constructor, the one taking no arguments. The base part of an object is never left unbuilt, so the only question is which constructor builds it, never whether one runs.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Item
 4  {
 5  public:
 6      Item(int a)
 7      {
 8          cout << "Item constructor, accession " << a << endl;
 9      }
10      ~Item()
11      {
12          cout << "Item destructor" << endl;
13      }
14  };
15  class Book : public Item
16  {
17  public:
18      Book(int a, int e) : Item(a)
19      {
20          cout << "Book constructor, edition " << e << endl;
21      }
22      ~Book()
23      {
24          cout << "Book destructor" << endl;
25      }
26  };
27  int main()
28  {
29      Book atlas(3140, 4);
30      return 0;
31  }
```

Output

```
Item constructor, accession 3140
Book constructor, edition 4
Book destructor
Item destructor
```

Program 4.112: *One base, one derived class, and a line printed from all four functions. Nothing in main calls any of them by name.*

Reading the four lines: Item is constructed first because Book is built on it and may use what the base has already set. At the closing brace the destructors run in reverse, so the last thing destroyed is the first thing built.

When the silence becomes an error: Item in Program 4.113 declares a constructor taking an int, so the compiler no longer supplies one taking nothing. Line 14 names no base, the call the compiler inserts has nothing to reach, and the diagnostic lands on the derived constructor rather than on the base.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Item
 4  {
 5  public:
 6      Item(int a)
 7      {
 8          cout << "Item constructor, accession " << a << endl;
 9      }
10  };
11  class Book : public Item
12  {
13  public:
14      Book(int e)
15      {
16          cout << "Book constructor, edition " << e << endl;
17      }
18  };
19  int main()
20  {
21      Book atlas(4);
22      return 0;
23  }
```

Compiler says

```
error: no matching function for call to 'Item::Item()'
note: candidate: 'Item::Item(int)'
note:   candidate expects 1 argument, 0 provided
```

Program 4.113: *The derived constructor at line 14 names no base, and the base has no constructor taking nothing. The listing does not compile.*

The habit that avoids all of this: name the base explicitly in every derived constructor, even where a default constructor exists. It costs seven characters, it survives somebody later giving the base a constructor with arguments, and it says which one runs.

Base first, derived second, and every destructor in the exact reverse of that.

The Master Key: *Name the base in the member initialiser list of the derived constructor; otherwise the compiler calls the default constructor of the base and fails if there is none, and construction runs base before derived while destruction runs derived before base.*

BINDER Question 109

One word does all the work in the definition, and the common teaching leaves it out. Polymorphism is drawn as animals making noises and never defined, so nobody can say when the version is chosen.

Define polymorphism, distinguish its two kinds, and state when each is resolved.

Model Answer

Polymorphism is one name standing for more than one behaviour, with the behaviour that suits the object in hand chosen when the name is used; C++ has two kinds, and they differ in when that choice is made.

Compile-time polymorphism: this is function and operator overloading, and the compiler settles it from the arguments written at the call, before the program runs. Section 8 sets out how the choice is made and it is not repeated here. It is also called static binding or early binding.

Run-time polymorphism: this is a virtual function called through a base class pointer or reference, and it is settled while the program runs, from the object that is actually there. Nothing at the call says which version will run. It is also called dynamic binding or late binding.

What virtual does: the keyword goes on the declaration in the base, as at line 6 of Program 4.114, and it sends the call to the version belonging to the object rather than to the type of the pointer. A function overriding a virtual function is itself virtual, written or not.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Ticket
 4  {
 5  public:
 6      virtual void show()
 7      {
 8          cout << "ordinary ticket, 4500" << endl;
 9      }
10      virtual ~Ticket()
11      {
12      }
13  };
14  class StudentTicket : public Ticket
15  {
16  public:
17      void show() override
18      {
19          cout << "student ticket, 2700" << endl;
20      }
21  };
22  int main()
23  {
24      StudentTicket half;
25      Ticket* issued = &half;
26      issued->show();
27      return 0;
28  }
```

Output

```
student ticket, 2700
```

Program 4.114: *A base pointer at line 25 holding a derived object declared at line 24. The call on line 26 reaches the derived version.*

What override does: written after the parameter list, as at line 17, it asks the compiler to confirm that this function really does override a virtual function of the base. Misspell the name or change a parameter and the compiler reports it instead of silently making a new function.

Without virtual there is no choice to make: the call is settled from the type of the pointer, so a base pointer reaches the base version even when the object is derived. Figure 4.26 draws the call that virtual makes possible, and the next question prints the difference.

Name the kind, then name the moment: overloading is chosen by the compiler, a virtual call is chosen by the object.

The Master Key: *Polymorphism is one name with several behaviours; compile-time polymorphism is overloading, settled by the compiler from the arguments, and run-time polymorphism is a virtual function through a base pointer or reference, settled while the program runs by the object that is there.*

REAL Question 110

A guide sold outside a school in Mugumu says a derived class always overrides the version in its base. The listing printed beside the claim compiles, runs, and prints the wrong line.

The hierarchy below is offered with the claim that a derived class always overrides the base version. Explain what actually happens, say exactly what the compiler and the running program do, and correct it.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Account
 4  {
 5  public:
 6      void charge()
 7      {
 8          cout << "base charge 2000" << endl;
 9      }
10  };
11  class SavingsAccount : public Account
12  {
13  public:
14      void charge()
15      {
16          cout << "savings charge 500" << endl;
17      }
18  };
19  int main()
20  {
21      SavingsAccount saver;
22      Account* held = &saver;
23      held->charge();
24      return 0;
25  }
```

Output

```
base charge 2000
```

Program 4.115: *The listing printed beside the claim, with the line it produced.*

Model Answer

The claim is false, and Program 4.115 prints base charge 2000 to prove it: SavingsAccount declares a function with the same name and the same parameters as the one in Account, but the base version is not virtual, so the call on line 23 is settled from the type of the pointer and never reaches the derived version at all.

Naming the fault: this is hiding, not overriding. Without virtual on line 6 there is nothing to override, so SavingsAccount merely declares a second function of the same name that hides the first for calls made through a SavingsAccount handle, and for no others.

What the compiler does about it: nothing whatever. Program 4.115 was built with all warnings turned on and produced not one diagnostic, because every line of it is legal. This is what makes the fault expensive: it is invisible at build time and visible only in a line of output somebody has to notice.

What the running program does: line 22 makes a pointer of type Account and points it at a SavingsAccount. The call on line 23 is resolved from the pointer type, so Account::charge runs, 2000 is printed, and the object being pointed at has no say in the matter.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Account
 4  {
 5  public:
 6      virtual void charge()
 7      {
 8          cout << "base charge 2000" << endl;
 9      }
10      virtual ~Account()
11      {
12      }
13  };
14  class SavingsAccount : public Account
15  {
16  public:
17      void charge() override
18      {
19          cout << "savings charge 500" << endl;
20      }
21  };
22  int main()
23  {
24      SavingsAccount saver;
25      Account* held = &saver;
26      held->charge();
27      return 0;
28  }
```

Output

```
savings charge 500
```

Program 4.116: *The correction. virtual on line 6, override on line 17, and nothing else in the listing changed.*

The correction, and why it is two words: virtual on the base declaration moves the decision to run time. override on the derived one adds no behaviour; it asks the compiler to check that a function of that exact signature really is virtual in the base, so a mistyped correction is rejected rather than accepted.

Reading the pair: the two listings differ by two keywords and their outputs differ by one line: base charge 2000 becomes savings charge 500. The object was a SavingsAccount in both, so the pair is evidence about the call, not about the object.

The habit worth taking from this: a base class that is going to be derived from should declare every function meant to be replaced as virtual, and a derived class should write override on every function it replaces. Then a fault of this kind stops the build instead of reaching the output.

Without virtual the pointer decides; with virtual the object decides, and only the second is what the claim described.

The Master Key: *A function of the same name in a derived class hides the base version rather than overriding it unless the base version is declared virtual, the compiler reports nothing, and the call through a base pointer prints the base line until virtual is written.*

REAL Question 111

The photocopier in a printing shop in Loliondo is a printer and it is a scanner, and nobody there finds that confusing. A note in the margin of a borrowed book insists that C++ cannot say so.

Explain what is wrong with the claim that hybrid inheritance must be built from interfaces because multiple inheritance is not available. Correct the claim, and state what genuine difficulty multiple inheritance does bring.

Model Answer

The claim describes a different language: C++ supports multiple inheritance directly, and Program 4.117 compiles and runs with Photocopier derived from both Printer and Scanner and using a member of each, so no interface is needed to build a hybrid hierarchy.

Where the rule actually comes from: Java and C sharp permit one base class only and offer interfaces in place of the rest. That restriction belongs to those languages. It is not a rule of object orientation, and repeating it about C++ turns a design choice made elsewhere into a false statement about this language.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Printer
 4  {
 5  public:
 6      void print()
 7      {
 8          cout << "page printed" << endl;
 9      }
10  };
11  class Scanner
12  {
13  public:
14      void scan()
15      {
16          cout << "page scanned" << endl;
17      }
18  };
19  class Photocopier : public Printer, public Scanner
20  {
21  public:
22      void copy()
23      {
24          scan();
25          print();
26      }
27  };
28  int main()
29  {
30      Photocopier machine;
31      machine.copy();
32      return 0;
33  }
```

Output

```
page scanned
page printed
```

Program 4.117: *Two independent bases, named on line 19 with an access specifier each. copy calls a member of one on line 24 and a member of the other on line 25.*

How the declaration is written: the bases are listed after the colon, separated by a comma, and each one carries its own access specifier: class Photocopier : public Printer, public Scanner. Omit the second public and that base is inherited privately, which is a common slip.

The difficulty that is real: it arrives when the two bases share a base of their own. Photocopier then inherits Device twice, so the object carries two Device sub-objects, each with its own copy of every Device member, and power no longer names one function.

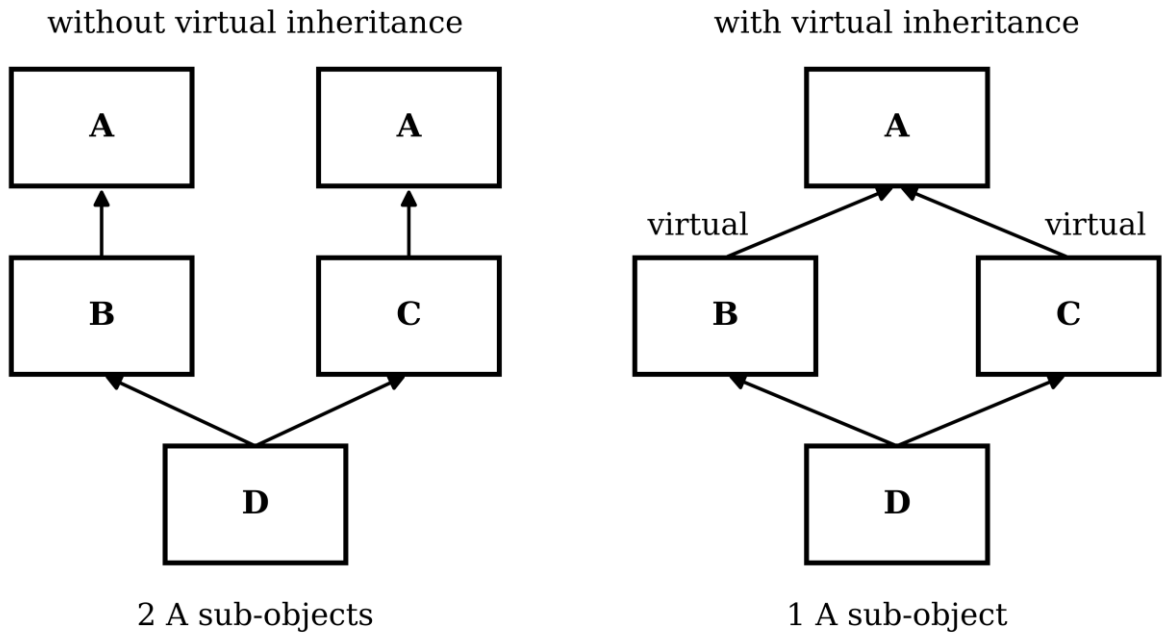


Figure 4.25: *The diamond. Without virtual inheritance the class at the bottom carries two copies of the class at the top; with it, one.*

```
 1  #include <iostream>
 2  using namespace std;
 3  class Device
 4  {
 5  public:
 6      void power()
 7      {
 8          cout << "power on" << endl;
 9      }
10  };
11  class Printer : public Device
12  {
13  };
14  class Scanner : public Device
15  {
16  };
17  class Photocopier : public Printer, public Scanner
18  {
19  };
20  int main()
21  {
22      Photocopier machine;
23      machine.power();
24      return 0;
25  }
```

Compiler says

```
error: request for member 'power' is ambiguous
note: candidates are: 'void Device::power()'
note:                 'void Device::power()'
```

Program 4.118: *The diamond of Figure 4.25 written out. Line 23 asks for power and there are two of them, so the listing does not compile.*

The first resolution: write virtual before the base in both middle classes, as Program 4.119 does on lines 11 and 14, and the two paths share one Device sub-object instead of owning one each. The ambiguity goes because there is now only one member to name.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Device
 4  {
 5  public:
 6      void power()
 7      {
 8          cout << "power on" << endl;
 9      }
10  };
11  class Printer : virtual public Device
12  {
13  };
14  class Scanner : virtual public Device
15  {
16  };
17  class Photocopier : public Printer, public Scanner
18  {
19  };
20  int main()
21  {
22      Photocopier machine;
23      machine.power();
24      return 0;
25  }
```

Output

```
power on
```

Program 4.119: *The same four classes with virtual inheritance on lines 11 and 14. One Device sub-object, and the call on line 23 now compiles.*

The second resolution: qualify the call with the scope resolution operator, writing machine.Printer::power(), which chooses one of the two copies. It compiles, but the duplication is still there and the two copies can hold different values, so it treats the symptom and not the cause.

C++ has multiple inheritance; what it does not have is a way of guessing which of two inherited copies you meant.

The Master Key: *Multiple inheritance is available in C++ and needs no interface; a shared base inherited along two paths gives an object two copies of that base and makes the member ambiguous, and the cure is virtual inheritance, or explicit qualification if you accept the duplication.*

REAL Question 112

Four readings were written into a dispensary record card in Kiomboi, and when the card was closed only one line was printed where two were expected. The program ended cleanly and returned zero.

A base class pointer is deleted after being used to hold a derived object. Trace what runs and what does not, and correct the listing.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Record
 4  {
 5  public:
 6      ~Record()
 7      {
 8          cout << "Record destructor runs" << endl;
 9      }
10  };
11  class PatientRecord : public Record
12  {
13  private:
14      int* readings;
15  public:
16      PatientRecord()
17      {
18          readings = new int[4];
19      }
20      ~PatientRecord()
21      {
22          delete[] readings;
23          cout << "PatientRecord destructor runs" << endl;
24      }
25  };
26  int main()
27  {
28      Record* card = new PatientRecord();
29      delete card;
30      return 0;
31  }
```

Output

```
Record destructor runs
```

Program 4.120: *A derived object destroyed through a base pointer at line 29. Both destructors print a line.*

Model Answer

Only the base destructor runs: Program 4.120 prints Record destructor runs and stops there, so the four integers PatientRecord allocated on line 18 are never released, and the delete on line 29 leaves half the object standing.

Why the derived destructor is skipped: the destructor on line 6 is not virtual, so delete is settled exactly as any other non-virtual call is: from the type of the pointer. The pointer is of type Record, Record::~Record is what runs, and the derived part is never told that its object is ending.

What is lost: line 22 is the only line that releases the array, and it is never reached, so those four integers are leaked. Anything else a derived class owns is lost with them: a file left open, a connection left standing, a total never written back.

Line	What it does	Did it run
18	allocates four integers for the derived part	Yes
29	deletes the object through a base pointer	Yes
8	prints from the base destructor	Yes
22	releases the four integers	No
23	prints from the derived destructor	No

Table 4.25: *What ran and what did not when Program 4.120 destroyed its object. The two lines that did not run are both inside the derived destructor, and they were skipped together.*

What the compiler said about it: nothing, with every warning switched on. Record declares no virtual function, so it is not a polymorphic type and the warning about deleting through a non-virtual destructor does not apply. The standard calls this undefined behaviour, and what it does in practice is what the output shows.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Record
 4  {
 5  public:
 6      virtual ~Record()
 7      {
 8          cout << "Record destructor runs" << endl;
 9      }
10  };
11  class PatientRecord : public Record
12  {
13  private:
14      int* readings;
15  public:
16      PatientRecord()
17      {
18          readings = new int[4];
19      }
20      ~PatientRecord() override
21      {
22          delete[] readings;
23          cout << "PatientRecord destructor runs" << endl;
24      }
25  };
26  int main()
27  {
28      Record* card = new PatientRecord();
29      delete card;
30      return 0;
31  }
```

Output

```
PatientRecord destructor runs
Record destructor runs
```

Program 4.121: *The correction. virtual on line 6 and override on line 20, and both destructors now run.*

Reading the corrected output: two lines now, derived first and base second, which is the order Question 108 established for every object and which delete has at last been made to follow. The array is released on the way past, so nothing is left owned by nobody.

The rule to carry: a class with any virtual function needs a virtual destructor. The cost is one keyword; the cost of leaving it out is a leak that no warning reports, and it grows with every object the program creates.

One keyword decides whether delete destroys the whole object or only the part the pointer knows about.

The Master Key: *Deleting a derived object through a base pointer runs only the base destructor unless that destructor is declared virtual, so any class with a virtual function must have a virtual destructor and every new must still be matched by a delete.*

BUILD Question 113

The mobile money kiosk in Mugumu adds a charge and the desk beside it does not. The ledger only wants to know that the twelve thousand was settled.

Write a program with an abstract base class and two derived classes, and call the same operation on each through a base class pointer.

The base class declares the operation and provides no body for it.

Two derived classes each supply their own version of that operation.

One array of base class pointers holds one object of each kind.

One loop calls the operation on every element.

The base class has a virtual destructor and every new is deleted.

Model Answer

Program 4.122 writes settle as a pure virtual function on line 6, which makes Payment an abstract class, and MobileMoney and CashDesk each supply their own, so the loop on line 30 calls one operation on two kinds of payment through one Payment pointer and gets two different answers.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Payment
 4  {
 5  public:
 6      virtual void settle(int amount) = 0;
 7      virtual ~Payment()
 8      {
 9      }
10  };
11  class MobileMoney : public Payment
12  {
13  public:
14      void settle(int amount) override
15      {
16          cout << "mobile money pays " << amount + 500 << endl;
17      }
18  };
19  class CashDesk : public Payment
20  {
21  public:
22      void settle(int amount) override
23      {
24          cout << "cash desk pays " << amount << endl;
25      }
26  };
27  int main()
28  {
29      Payment* till[2] = {new MobileMoney(), new CashDesk()};
30      for (int i = 0; i < 2; i++)
31      {
32          till[i]->settle(12000);
33          delete till[i];
34      }
35      return 0;
36  }
```

Output

```
mobile money pays 12500
cash desk pays 12000
```

Program 4.122: *An abstract base class and two derived classes. The loop calls settle twice and never names either derived class.*

Writing the pure virtual function: put = 0 after the declaration and give it no body. It says that every usable class in this family has a settle, and that Payment itself does not know how to perform one. That is abstraction written as a declaration.

What abstract means in practice: no object of the class can be made. Program 4.123 tries to declare one and the compiler refuses, naming the pure function that is missing. A derived class that fails to supply settle is abstract in its turn, and objects of it cannot be made either.

```
 1  #include <iostream>
 2  using namespace std;
 3  class Payment
 4  {
 5  public:
 6      virtual void settle(int amount) = 0;
 7      virtual ~Payment()
 8      {
 9      }
10  };
11  int main()
12  {
13      Payment direct;
14      direct.settle(12000);
15      return 0;
16  }
```

Compiler says

```
error: cannot declare variable 'direct' to be of abstract type 'Payment'
note:   because the following virtual functions are pure within 'Payment':
note:     'virtual void Payment::settle(int)'
```

Program 4.123: *Line 13 tries to declare an object of the abstract class. The listing does not compile.*

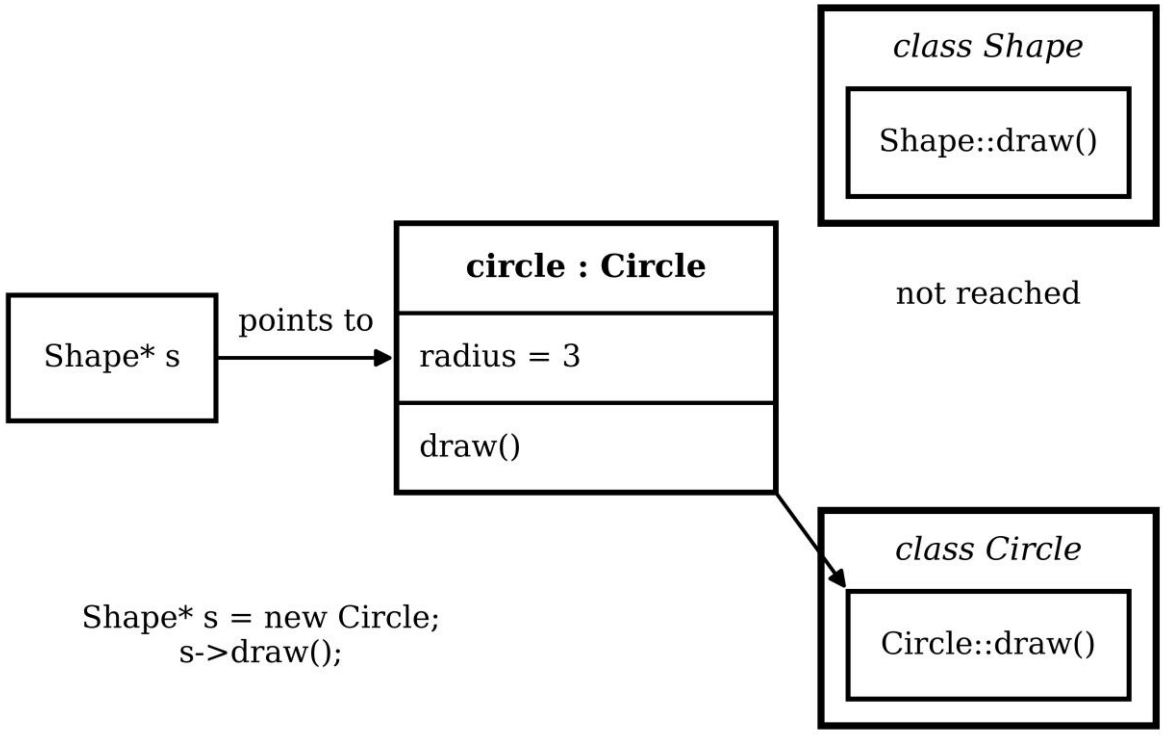


Figure 4.26: *One call, two candidates. The pointer is of the base type and the object is not, so a virtual call reaches the version belonging to the object that is actually there and leaves the base version unreached.*

The destructor and the deletes: line 7 makes the base destructor virtual, so the delete on line 33 reaches the destructor of the object it is actually holding. Each new in the array on line 29 is matched by that delete, once per element, inside the same loop.

Declare the operation in the base, refuse to define it, and let each derived class answer in its own way.

The Master Key: *A pure virtual function written = 0 makes its class abstract, so no object of it can be created and every derived class must supply the function or stay abstract, and a base pointer then calls one operation across the whole family.*

BUILD Question 114

A surveyor near Katesh measured three things in one morning: a rectangular plot, a triangular signboard and a circular water tank. One notebook, one column for area and one for perimeter.

Write a program that computes the area and the perimeter of three different shapes through one interface.

An abstract base class Shape declares the reporting operation.

Three derived classes compute the area and the perimeter of their own shape.

Rectangle 40 by 25; triangle with legs 9 and 12 and hypotenuse 15; circle of radius 7.

Take pi as 22 divided by 7.

One array of Shape pointers and one loop that reports all three.

Model Answer

Program 4.124 declares report as a pure virtual function on line 7, gives each of the three shapes its own version, and then holds all three in one array of Shape pointers on line 38 so that a single loop reports them without knowing which shape it has.

```
 1  #include <iostream>
 2  using namespace std;
 3  const double PI = 22.0 / 7;
 4  class Shape
 5  {
 6  public:
 7      virtual void report() = 0;
 8      virtual ~Shape()
 9      {
10      }
11  };
12  class Rectangle : public Shape
13  {
14  public:
15      void report() override
16      {
17          cout << "rectangle " << 40 * 25 << " " << 2 * (40 + 25) << endl;
18      }
19  };
20  class Triangle : public Shape
21  {
22  public:
23      void report() override
24      {
25          cout << "triangle " << 9 * 12 / 2 << " " << 9 + 12 + 15 << endl;
26      }
27  };
28  class Circle : public Shape
29  {
30  public:
31      void report() override
32      {
33          cout << "circle " << PI * 7 * 7 << " " << 2 * PI * 7 << endl;
34      }
35  };
36  int main()
37  {
38      Shape* site[3] = {new Rectangle(), new Triangle(), new Circle()};
39      for (int i = 0; i < 3; i++)
40      {
41          site[i]->report();
42          delete site[i];
43      }
44      return 0;
45  }
```

Output

```
rectangle 1000 130
triangle 54 36
circle 154 44
```

Program 4.124: *Three shapes behind one interface. The loop on line 39 calls report three times and mentions no shape by name.*

One interface, three sets of arithmetic: the rectangle multiplies 40 by 25 for 1000 and doubles 40 plus 25 for 130. The triangle halves 9 times 12 for 54 and adds 9, 12 and 15 for 36. The circle takes pi times 7 times 7 for 154 and twice pi times 7 for 44.

The loop that does not know what it holds: site is an array of pointers to Shape, so nothing on lines 39 to 43 mentions Rectangle, Triangle or Circle. Add a fourth shape and this loop is not edited; that is the whole return on writing the base class.

The one design decision worth naming: report is one operation printing both figures rather than two operations returning numbers, which keeps the listing inside a page. Where the numbers are wanted for further work, declare area and perimeter separately and let the caller do the printing.

Three shapes, three sets of arithmetic, one loop, and no shape named anywhere in it.

The Master Key: *Put the operation in an abstract base class, let each shape implement it, and hold the objects as base class pointers, and a loop written once reports every shape you will ever add.*

BUILD Question 115

The library in Kibara asked, at the start of this chapter, for an item, a book, a periodical and one list holding all of them. The last two lines of the output prove nothing was left behind.

Write a program that models a library item, with a book and a periodical derived from it, and reports a mixed collection through one base class pointer.

The base class holds what every item has, and declares the reporting operation.

Book adds the author; Periodical adds the issue number.

The base class is initialised from each derived constructor.

One array of base class pointers holds a book and a periodical.

The destructor is virtual, and every allocation is released.

Model Answer

Program 4.125 puts the title in Item because both kinds have one, declares card as a pure virtual function on line 12 so that each kind prints its own, and holds a book and a periodical in one array of Item pointers on line 44, which the loop reports without knowing which is which.

```
#include <iostream>
#include <string>
using namespace std;
class Item
{
protected:
    string title;
public:
    Item(string t) : title(t)
    {
    }
    virtual void card() = 0;
    virtual ~Item()
    {
        cout << "released " << title << endl;
    }
};
class Book : public Item
{
    string author;
public:
    Book(string t, string a) : Item(t), author(a)
    {
    }
    void card() override
    {
        cout << "Book: " << title << ", by " << author << endl;
    }
};
class Periodical : public Item
{
    int issue;
public:
    Periodical(string t, int i) : Item(t), issue(i)
    {
    }
    void card() override
    {
        cout << "Periodical: " << title << ", issue " << issue << endl;
    }
};
int main()
{
    Item* shelf[2] = {new Book("Coast Atlas", "Mnyika"),
                      new Periodical("Maize Prices", 58)};
    for (int i = 0; i < 2; i++)
    {
        shelf[i]->card();
        delete shelf[i];
    }
    return 0;
}
```

Output

```
Book: Coast Atlas, by Mnyika
released Coast Atlas
Periodical: Maize Prices, issue 58
released Maize Prices
```

Program 4.125: *The library design of Section 1, compiled. One array, two kinds of item, and one loop that names neither.*

What belongs in the base: the title, because every item has one and it should be declared once. The author belongs to Book and the issue number to Periodical, because neither is true of the other kind, and putting either in Item would leave one class carrying a member it can never fill.

How the base is initialised: line 22 and line 34 each name Item in the member initialiser list, passing the title up before setting their own member. Every Item therefore has a title from the instant it exists, and neither derived class can forget to give it one.

Released, and proved released: the destructor on line 13 is virtual, so the delete on line 49 reaches the derived destructor and then the base one. It prints as it goes, which is why released appears twice in the output: every new made on lines 44 and 45 is accounted for.

One base class, two derived classes, one array, one loop, and every object released by name.

The Master Key: *Declare in the base what every kind of item has, let each derived class add what only it has and supply the operation, hold the collection as base class pointers, and make the base destructor virtual so the whole object is released.*

HOT Question 116

The question came from the back of the room and it was the right one: if two classes already share half their code, why is that not reason enough to make one the base of the other?

Evaluate the claim that inheritance is the right tool whenever two classes share code. Reach a judgement.

Model Answer

The claim is wrong, and what decides the matter is not how much code two classes share but whether the derived thing is genuinely a kind of the base thing: inherit when it is, compose when it is not, and treat shared code alone as a reason for a function rather than a colon.

What is right in the claim: inheritance does remove duplication, and duplication is expensive. Item in Program 4.125 declares the title once for both kinds, so the two cannot drift apart, and a third kind of item costs one class and no edit to anything already working.

But reuse is the smaller half: the larger half is that one base class pointer can hold any of the derived kinds, which is what the loops in Program 4.122, Program 4.124 and Program 4.125 depend on. A hierarchy built only to save typing collects the cost of inheritance and none of that return.

The test that decides it: read the relation aloud as is a kind of. A book is a kind of item, and a photocopier is a kind of printer. Now try the other way: a loan is not a kind of student, however much code the two share.

Composition, the alternative: hold an object of the other class as a member and call it. A Loan holds an Item and a Student rather than inheriting from either. The relation is then has a, the coupling is limited to the functions actually called, and nothing is inherited that does not fit.

What inheritance costs: the derived class is tied to the protected members of the base and to how the base behaves, not merely to what it promises. Every public member of the base becomes public in the derived class, whether it makes sense there or not.

The fragile base class: a change inside the base can break derived classes that were correct yesterday and were never edited. Change what a base function does with a member and every class below may be quietly wrong, and the wider the hierarchy the worse that is.

The question to ask	If the answer is yes	If the answer is no
Is it a kind of the other thing	Inheritance is available	Use composition
Must it stand where the base stands	Inheritance is needed	Composition is enough
Does every public member of the base fit it	Inherit publicly	Compose, or inherit privately
Do they only share code	Make a third class or a function	The question does not arise

Table 4.26: *Four questions asked in order, and what each answer settles. Only the first two are about inheritance at all; the third asks whether the inheritance is honest, and the fourth catches the case in the claim.*

The verdict: use inheritance only where the derived class is a kind of its base and can stand wherever the base stands, because that is where the coupling buys something. Where two classes merely share code, put the shared part in a function they both call.

Shared code is a reason to write something once; it is not a reason to say that one thing is a kind of another.

The Master Key: *Inheritance is a statement about what a thing is, not a device for saving typing, so derive only where the derived class is genuinely a kind of its base and can stand in its place; where all that is shared is code, share it through a function or a third class instead.*

INDEX

A

Abstraction, 374, 375, 518. *see also* Encapsulation (object orientation)
  abstract class and pure virtual function, 532-536
Access point, 239-241, 360. *see also* Wireless networking
Accumulator, 15, 19, 22
Adder, 65-72
  half adder, 65-67
  full adder, 66, 67, 71
  ripple carry, 67, 71, 72
  carry look-ahead, 72
Address bus, 3, 4, 101. *see also* Buses
Address decomposition, 93-100. *see also* Cache memory
Addressing modes, 108-115
  immediate, 109-112
  direct and indirect, 113
  register and register indirect, 110-113
  indexed and base-offset, 110-113
ALU (arithmetic and logic unit), 17
Amdahl's law, 181-187. *see also* Speed-up
  parallel fraction, 182-185
  limits of speed-up, 181
Analogue and digital signalling, 215-220
Application layer, 195, 201, 202, 315-320
Argument, *see* Parameter
Arithmetic operators, 412-424
ARP (Address Resolution Protocol), 200, 228-234
Array, 470-484. *see also* String
  declaration and initialisation, 472
  index and bounds, 473, 481, 482
  two dimensional, 474-476
  as a function parameter, 478
  traversal, 474
Assembly language, 380-388
Associativity of operators, 412-424. *see also* Precedence of operators
Attenuation, 215-220
Authentication, 233, 349-355, 362
Average memory access time (AMAT), 86-92. *see also* Cache memory

B

Bandwidth, 220, 221, 258, 321
Base class, 518-539. *see also* Inheritance
  initialising from the derived constructor, 521-523
  virtual destructor, 531, 532
Bellman-Ford algorithm, 262-268. *see also* Distance vector routing
Binary arithmetic, 68-70
  two's complement, 68-70
  overflow, 69-71
Binary conversion, 276-284, 287
Bit and byte, 87, 212-214, 310, 311
Block size, cache, 93-100. *see also* Cache memory
Boolean algebra, 23-29. *see also* De Morgan's theorems
  absorption, 24-27
  commutative, associative and distributive laws, 23-27
  complement and identity laws, 24-27
  simplification, 23-25, 29
Boolean type, 400-411, 417
Branch prediction, 153-159. *see also* Control hazard
  static and dynamic, 153, 154
  branch penalty, 153, 157, 158
Bridge, 235-241, 327. *see also* Switch (network device)
Broadcast address, 279, 285-299, 305
Broadcast domain, 213, 235-242. *see also* Collision domain
Buffer overflow, 491, 498. *see also* Network security
Buses, 1-5, 11
  address, data and control, 2-4
  bus width and its effect, 101

C

Cabling, structured, 335-342. *see also* Twisted pair
  T568-A and T568-B, 335-337
  straight-through and crossover, 336, 337, 340
  crimping and testing, 338-341
Cache coherence, 167-173. *see also* Multicore
  false sharing, 172
Cache memory, 86-100. *see also* Locality of reference
  hit and miss, 89-91
  direct mapped, 99
  fully associative, 94, 98, 99
  set associative, 94, 95
  replacement policy, 93, 97-99
  write policy, 95, 96, 100
  split instruction and data, 141, 142, 145
Carrier sense multiple access, 235-241. *see also* Media access control
  CSMA/CD, 235-237, 240
  CSMA/CA, 236, 237, 241
Character type, 400-411, 485, 497, 498. *see also* String
Checksum, 308, 312, 313. *see also* Error detection
CIA triad, 349-355
CIDR notation, 285-299
CISC, 108-115. *see also* RISC
Class, 499-517. *see also* Object
  data members and member functions, 500, 503
  declaration and definition, 501
  object as an instance, 503
  scope resolution operator, 501
Clock, 128-138. *see also* CPI (cycles per instruction)
  clock period and clock rate, 132, 137, 138
  cycles per instruction, 133, 137, 138
Coaxial cable, 321-327
Collision domain, 235-241. *see also* Broadcast domain
Combinational logic, 79, 84. *see also* Sequential logic
Comment, 389-399
Comparator, 73-78
Compiler, 380-388. *see also* Error, classes of program
  compile-time error, 382, 385, 386
  preprocessor, 380, 381
  linker, 380-382
Congestion control, 252, 269-275, 310, 314
Constant, 400-411
Constructor, 499-518, 521-523. *see also* Destructor
  default and parameterised, 503, 504
Control hazard, 139, 153-158. *see also* Hazard
Control unit, 16
Counter, 79-85
  synchronous and asynchronous, 82, 83
CPI (cycles per instruction), 131-138, 144, 151, 152
  effect of stalls, 133, 137
CRC (cyclic redundancy check), 221-227. *see also* Error detection
  the generator and the division, 224, 225
  frame check sequence, 222, 223, 227

Cryptography, 246, 247, 356-362
symmetric and asymmetric, 357-359, 362
AES, 357-360
digital signature, 361

D

Data hazard, 139, 146-152. *see also* Hazard
forwarding, 149-151
load-use stall, 146, 149, 150
Data level parallelism, 174-180
Data link layer, 194-200, 208, 212, 221-227
Data types, 400-411
built-in and derived, 400, 401, 409
size and range, 405, 406
type conversion and casting, 412-424
De Morgan's theorems, 30-35, 55
at gate level, 30-32
proof by truth table, 30
Decoder, 73-78, 102, 105, 114
Default gateway, 274
Dependence, 143, 147, 159-166, 176, 177
true, anti and output, 161, 162
Destructor, 499-517, 530-533. *see also* Constructor
DHCP, 284, 316-318
Dijkstra's algorithm, 262-268. *see also* Link state routing
Direct memory access, 116-124. *see also* Interrupt
cycle stealing, 117
compared with programmed I/O and interrupts, 118
Distance vector routing, 255-261. *see also* Bellman-Ford algorithm
RIP, 256, 257, 260
DNS (Domain Name System), 315-320
Do-while loop, *see* Loop
Duplex, half and full, 238-240

E

Encapsulation (networking), 207-214
de-encapsulation, 207, 208, 212
protocol overhead, 207, 211-213
protocol data unit, 208
Encapsulation (object orientation), 371-379, 499-517. *see also* Information hiding
access specifiers, 501, 515-517
getters and setters, 515-517
Encoder, 73-78
Encoding, line, 215-220. *see also* Analogue and digital signalling
Manchester encoding, 216, 217, 220
bit rate and baud rate, 216, 217, 220
Error, classes of program, 380-388
syntax, 382, 386
logic, 382-386
run-time, 382, 383, 386, 387
linker, 381
Error correction, 221-227. *see also* Error detection
compared with retransmission, 227
Error detection, 221-227. *see also* CRC (cyclic redundancy check)
parity, 224
checksum, 224
Ethernet, 222, 223, 323, 324
Execution unit, 140, 159-166, 169, 172, 173
Expression, logic, 57-66
to circuit, 57, 63, 64
from circuit, 57, 64

F

Fetch-decode-execute cycle, 15-22. *see also* Registers
register transfer language, 15, 18, 19
the three phases, 15-17
Fibre optic cable, 321-327, 333, 346, 349
single mode and multimode, 324-327
FIFO, 97-99
Firewall, 196, 349-356, 362, 363
Flip-flop, 79-85. *see also* Sequential logic
SR, 80-82
D, 81, 82
JK, 82
T and toggle, 82, 83
Floating point, 400-411, 414, 415, 422-424
Flynn's taxonomy, 6-15. *see also* SIMD
SISD, SIMD, MISD, MIMD, 10-13
For loop, *see* Loop
Forwarding (bypassing), 146-152. *see also* Data hazard
Frame, 208-213, 221-227, 230, 231, 237
framing methods, 222
Full adder, *see* Adder
Function, 455-469. *see also* Parameter
declaration, definition and call, 455, 456
return type and value, 456-459
overloading, 455, 458, 459
recursion, 464, 467, 468
the base case in recursion, 463-465

G

Gates, logic, 51-56. *see also* Truth table
AND, 52-54
OR, 51-53
NOT and the inverter, 51-54
NAND, 51-53, 56
NOR, 51, 52, 55
XOR and XNOR, 51-53
universality of NAND and NOR, 51-53, 56
GSM and mobile generations, 363-370

H

Half adder, *see* Adder
Handshake, three-way, 307-314. *see also* TCP
Hardware address, *see* MAC address
Harvard architecture, 6-14. *see also* Von Neumann architecture
modified Harvard, 6-9
compared with von Neumann, 7, 8, 13
Hash function, 356-362. *see also* Cryptography
Hazard, 139-145. *see also* Pipelining
structural, 139-141
data, 139
control, 139
stall and bubble, 142-144
Header, 190, 207-214, 300-302, 308, 309
Hit rate and miss rate, 86-92, 97-99. *see also* Cache memory
Hop count, 258, 259, 263
Host bits and host range, 279, 285-299. *see also* Subnetting
HTTP, 196, 315-320
Hub, 196, 199, 238. *see also* Switch (network device)

I

ICMP, 196, 269-275
If statement, 425-438. *see also* Selection
nested, 425
dangling else, 430, 438
Information hiding, *see* Encapsulation (object orientation)
Inheritance, 518-539. *see also* Base class
single, multiple and multilevel, 526, 529

public, private and protected, 519-521
is-a and has-a, 518, 538, 539
compared with composition, 538
fragile base class, 538
Input and output, machine, 2, 116-124. *see also* Interrupt
programmed I/O and polling, 120, 123
memory mapped and isolated, 119, 120
Input and output, program, 389-399, 415-417
insertion and extraction operators, 393, 396, 397
endl and the newline, 391-395
Instruction format, 108-115
opcode and operand, 109-112
Instruction level parallelism, 159-166
multiple issue and superscalar, 159, 160, 165
out-of-order execution, 159, 166
speculative execution, 155, 156
Instruction register, 19-22. *see also* Registers
Instruction set architecture, 108-115. *see also* RISC; CISC
Integer types, 400-411
Interrupt, 22, 116-124. *see also* Direct memory access
maskable and non-maskable, 117
priority and vectored, 117-121
IP address, 276-284. *see also* Subnetting
classes A to E, 278-283
network and host portions, 276, 279, 280
private and public, 281-284
loopback and reserved, 282
IPv6, 253, 300-306
notation and compression, 300-303
header, 302, 303
coexistence with IPv4, 304-306
Iteration, *see* Loop

J

Jitter, 252, 272, 370

K

Karnaugh map, 43-50. *see also* Boolean algebra
grouping and adjacency, 45, 46, 49
don't care conditions, 48, 49
gray code ordering, 44-46, 49
four variable, 46, 49, 50
Keyword, 389-399, 403, 499, 508

L

LAN (local area network), 228, 342-348
design and documentation, 342, 343, 348
simulation, 342, 347, 348
Latency, 91, 125, 133-135. *see also* Throughput
Leaky bucket and token bucket, 269-275. *see also* Congestion control
Library, standard, 485-487
iostream, 486
cstring and string.h, 486-488
cmath and the maths functions, 460, 461
Link state routing, 255-261, 267. *see also* Dijkstra's algorithm
OSPF, 256, 257, 260, 261
Locality of reference, 86-92. *see also* Cache memory
temporal, 88
spatial, 88
Logic error, *see* Error, classes of program
Loop, 439-454
while, 440-442, 452
do-while, 439-442
for, 440-442, 452
nested, 439, 444
break and continue, 442, 453, 454
Loop unrolling, 159-166. *see also* Instruction level parallelism

M

MAC address, 228-234. *see also* ARP (Address Resolution Protocol)
OUI and vendor portion, 228, 229, 233
Media access control, 235-241. *see also* Carrier sense multiple access
contention and controlled access, 235, 240, 241
Memory address register, 15, 19, 20. *see also* Registers
Memory capacity arithmetic, 101-107
address lines and addressable locations, 102-105
chips across and chips down, 102-105
Memory data register, 15, 19, 20. *see also* Registers
Memory hierarchy, 86-92. *see also* Cache memory
registers, cache, main and secondary, 87, 88, 91
cost, speed and capacity trade-off, 86, 87
Minterm and maxterm, 36-42. *see also* Sum of products
Multicore, 167-173
Multiplexer, 73-79
select lines, 73, 76-78
demultiplexer, 77, 78
Multithreading, simultaneous, 167-174. *see also* Thread

N

Namespace, and using namespace std, 389-399
NAND gate, 31, 34, 35, 51-56. *see also* Gates, logic
Network layer, 205, 249-254, 274, 307
design issues, 249-252
Network security, 349-355, 360-362. *see also* Cryptography
threats and attacks, 349, 353, 354
controls and countermeasures, 349, 350, 354, 355
Node, network, 263, 266, 344
NOR gate, 32, 51-56, 80. *see also* Gates, logic

O

Object, 371-381, 499-517. *see also* Class
state and behaviour, 372-376
creating and using, 503
Object oriented programming, 371-380
the four pillars, 371
compared with procedural, 371, 372
Opcode, 108-115
Operator, 412-424
arithmetic, 414, 415, 424
relational and logical, 417-419, 422
assignment and compound assignment, 413
increment and decrement, 412, 416-418
precedence and associativity, 412, 413, 418, 419
OSI model, 188-193, 202-206. *see also* TCP/IP model
the seven layers, 188, 189, 193
device and PDU at each layer, 188-192
compared with TCP/IP, 201-206
Out-of-order execution, 159-166
Overflow, 65-72. *see also* Binary arithmetic
Overloading, 455-469. *see also* Polymorphism
function, 458
compared with overriding, 524-526, 531
Overriding, 518-539. *see also* Polymorphism

P

Packet, 196, 208-210, 250-252
Parallelism, 159, 160. *see also* Amdahl's law
instruction level, 159, 160
thread level, 167-173
data level, 174-180
Parameter, 455-469
formal and actual, 456

default, 455, 459, 460
by value and by reference, 456, 457, 469
const reference, 458
Parity, 221-227. *see also* Error detection
Physical layer, 195, 215-221, 348
Pipelining, 125-130. *see also* Hazard
the five stages, 125, 126, 129
speed-up and throughput, 125, 128, 129
filling and draining, 129, 130
hazards in, 125, 130
Pointer, 518-539
base class pointer, 529-531, 536
new and delete, 499-517
Polymorphism, 374, 518-539. *see also* Overriding; Virtual function
compile-time and run-time, 523, 524
virtual function, 532-536
pure virtual and abstract class, 532-536
Port number, 199, 208, 212, 315-320. *see also* Socket
well-known, registered and dynamic, 315
Precedence of operators, 58-60, 412-425. *see also* Associativity of operators
Presentation layer, 193-200. *see also* OSI model
Product of sums, 36-42. *see also* Sum of products
Program counter, 19-22, 121. *see also* Registers
Programming paradigm, 371-379
Protocol, 194-200, 250, 262, 263, 315, 316
PSTN, 363-370

Q

Quality of service, 252, 253, 269-275

R

Race condition, 167-173. *see also* Thread
critical section, 170, 171
lock and mutex, 171-173
Recursion, 464, 467, 468
base case, 463-465
Register file, 146-152. *see also* Registers
Register transfer language, 15-22
Registers, 15-22. *see also* Fetch-decode-execute cycle
program counter, 20-22
instruction register, 19, 22
MAR and MDR, 16-20
general purpose and special purpose, 15, 16
Repeater, 219, 330
Replacement policy, 93-100. *see also* Cache memory
RISC, 108-115. *see also* CISC
compared with CISC, 108, 111
Router, 249, 250, 256-261, 265-271. *see also* Routing
Routing, 255-268
static and dynamic, 256
distance vector, 257-259
link state, 257-259
routing table, 257, 262
metric and administrative distance, 259, 263-265
Run-time error, *see* Error, classes of program

S

Satellite communication, 314, 322, 328-334. *see also* Unguided media
GEO, MEO and LEO, 331, 332
Scope and lifetime, 400-411, 461
local and global, 411
static, 402-404, 410
Segmentation, network, 242-248. *see also* VLAN
Selection, 425-438
if and if-else, 427
switch, 425, 428-430
conditional operator, 413
Semicolon, as statement terminator, 384, 389-399, 438, 499
Sequential logic, 79-85. *see also* Combinational logic; Flip-flop
state table and state diagram, 79, 85
clocked and level triggered, 81, 82
Session layer, 192-200. *see also* OSI model
SIMD, 174-180. *see also* Flynn's taxonomy
lanes and vector width, 175-178
SIMT, 174, 179
SMT, *see* Multithreading, simultaneous
Socket, 315-320, 345. *see also* Port number
Speed-up, 131-138, 181-183. *see also* Amdahl's law
from pipelining, 133-135
Amdahl's limit, 181-187
Stall, 142-152. *see also* Hazard
State table, 79-85
String, 485-498. *see also* Array
char array and null terminator, 485, 497, 498
C string library, 486, 487, 492-494
std::string, 485, 497
getline, 490, 494, 495
Structural hazard, 139-145. *see also* Hazard
Subnetting, 285-299. *see also* CIDR notation
subnet mask, 293
borrowed bits and block size, 286-289
host range and usable hosts, 285-289, 296
VLSM, 293
supernetting and summarisation, 293, 299
Subtractor, 65-72. *see also* Adder
Sum of products, 36-42, 50, 58. *see also* Product of sums
Superscalar, 159-166
Switch (network device), 196, 197, 228-234, 241-248, 348, 349. *see also* Bridge; Hub; VLAN
MAC address table, 230, 231, 234
Switch statement, 425-438. *see also* Selection
fall-through and break, 428, 434, 435
default label, 429, 435, 436
Syntax error, *see* Error, classes of program

T

TCP, 307-314. *see also* UDP
three-way handshake, 309, 312, 313
reliable delivery and acknowledgement, 309-311
flow control and windowing, 308-310, 314
compared with UDP, 307, 312, 313
TCP/IP model, 201-207. *see also* OSI model
mapping onto OSI, 203-206
Thread, 167-173, 178, 179. *see also* Multicore
thread level parallelism, 167
simultaneous multithreading, 167-169, 172
Thread level parallelism, 167-173. *see also* Thread
Throughput, 125, 131-138, 144, 151. *see also* Latency
TLB (translation lookaside buffer), 104, 107, 108. *see also* Virtual memory
Topology, network, 268, 342-348
star, bus, ring and mesh, 344-346
Transport layer, 307-314
Truth table, 29, 30, 36-44, 78. *see also* Karnaugh map
minterms and maxterms, 37-39
Twisted pair, 321-327, 339, 340. *see also* Cabling, structured

U

UDP, 196, 307-316, 320. *see also* TCP
Unguided media, 328-334. *see also* Satellite communication
the spectrum, 328-330

Universality of NAND and NOR, 51-53, 56. *see also* Gates, logic

V

Variable, 400-411, 454-456. *see also* Scope and lifetime
  declaration and initialisation, 400-404, 410, 411
  naming rules, 403, 407
Vector width, 174-180. *see also* SIMD
Virtual function, 532-536. *see also* Polymorphism
Virtual memory, 101-107. *see also* TLB (translation lookaside buffer)
  page table and page fault, 103, 104
VLAN, 242-249. *see also* Segmentation, network
  trunking and tagging, 243-246
  inter-VLAN routing, 248
VLSM, 293-299. *see also* Subnetting
Von Neumann architecture, 1-8. *see also* Harvard architecture
  the bottleneck, 3-5
  stored program concept, 2
  compared with Harvard, 6-14

W

While loop, *see* Loop
Wireless networking, 237, 327, 330, 331, 356-362. *see also* Access point
  WEP, WPA and WPA2, 359-361
Write policy, 93-100. *see also* Cache memory

X

XNOR gate, 51-56, 75, 76. *see also* Gates, logic
XOR gate, 51-57, 65-68. *see also* Gates, logic