
Computer Systems

Consolidated Reference Notes

- Part I** Data Representation
- Part II** Boolean Algebra and Logic Minimisation
- Part III** Combinational Logic Circuits
- Part IV** Sequential Logic Circuits
- Part V** Computer Organisation and Architecture
- Part VI** Memory Systems

Consolidated from lecture series **2.1–4.1** and **Tutorials 01–08**.
Overlapping material across lectures and tutorials has been merged
into a single treatment of each topic; topics left as “search and
find” in the slides have been written up and marked accordingly.

August 5, 2026

Contents

How to use these notes	ix
I Data Representation	1
1 Bits, Bytes and Words	2
1.1 The bit	2
1.2 The byte and the storage hierarchy	2
1.3 Word size	3
1.4 Memory organisation and addressability	3
1.5 Summary	5
2 Number Systems and Conversion	6
2.1 The four bases	6
2.2 Positional value: any base to decimal	6
2.3 Decimal to any base: repeated division	7
2.4 Binary $\leftrightarrow$ hexadecimal: group in fours	8
2.5 Binary $\leftrightarrow$ octal: group in threes	8
2.6 Octal $\leftrightarrow$ hexadecimal: go via binary	9
2.7 Conversion map	9
2.8 Tutorial 01 exercises, with answers	10
3 Binary Arithmetic	12
3.1 Binary addition	12
3.2 Binary subtraction	13
3.3 Binary multiplication	13
3.4 Binary division	14
3.5 Summary of operator–gate correspondence	15
4 Integer Representation	16
4.1 Unsigned representation	16
4.2 Signed magnitude	16
4.2.1 Two problems with signed magnitude	17
4.3 One’s complement	17
4.3.1 Arithmetic: the end-around carry	17
4.4 Two’s complement	18
4.4.1 The shortcut	19
4.4.2 Why there is only one zero	19
4.5 Range comparison	20
4.6 Reading a two’s complement number	20
5 Fractional and Floating Point Representation	22
5.1 Binary fractions	22
5.1.1 Decimal fraction to binary: repeated multiplication	22
5.2 Normalised scientific notation	23
5.3 The 8-bit teaching format	23
5.4 Excess- K (offset binary) encoding	24
5.4.1 Ranges	24

5.4.2	Excess- K compared with two's complement	24
5.5	Encoding: decimal to floating point	25
5.6	Decoding: floating point to decimal	25
5.7	Round-off error	26
5.8	Range of the 8-bit format	27
5.9	Overflow and underflow	27
5.10	The IEEE 754 standard	28
5.10.1	Exponent encoding and reserved patterns	28
5.10.2	Reserved bit patterns	28
5.10.3	IEEE 754 single precision worked example	29
5.10.4	IEEE 754 single precision range	30
5.11	Summary	30
II	Boolean Algebra and Logic Minimisation	31
6	Boolean Algebra and Logic Gates	32
6.1	Origins and the two states	32
6.2	The basic operations	32
6.3	Truth tables	33
6.4	From algebra to hardware	33
6.5	The gate catalogue	33
6.5.1	AND	33
6.5.2	OR	34
6.5.3	NOT (inverter)	34
6.5.4	XOR (exclusive OR)	34
6.5.5	NAND (NOT-AND)	35
6.5.6	NOR (NOT-OR)	35
6.5.7	XNOR (exclusive NOR)	35
6.5.8	Buffer	35
6.5.9	Complete reference	36
6.6	Multi-input gates	36
6.7	Operator precedence	36
6.8	Drawing a circuit from an expression	37
6.9	Reading a circuit back into an expression	38
6.10	Summary	39
7	Canonical and Standard Forms	40
7.1	One behaviour, many expressions	40
7.2	Expression to truth table	40
7.3	Minterms and maxterms	41
7.3.1	Minterm	41
7.3.2	Maxterm	42
7.4	Truth table to canonical form	42
7.4.1	Sum of minterms	42
7.4.2	Product of maxterms	43
7.4.3	Deriving the product of maxterms properly	43
7.5	Canonical form to truth table	44
7.5.1	Sum of minterms to truth table	44
7.5.2	Product of maxterms to truth table	45
7.6	Sigma and Pi notation	45
7.7	Standard forms	46

7.7.1	Sum of Products (SoP)	46
7.7.2	Product of Sums (PoS)	46
7.8	Choosing between SoM and PoM	46
7.9	Worked exercise	47
7.10	Summary	47
8	Algebraic Simplification	48
8.1	Postulates (axioms)	48
8.2	The laws	48
8.2.1	Annulment law	48
8.2.2	Identity law	48
8.2.3	Idempotent law	48
8.2.4	Complement law	49
8.2.5	Commutative law	49
8.2.6	Associative law	49
8.2.7	Double negation law	49
8.2.8	Distributive law	49
8.2.9	Redundancy law (absorption)	50
8.2.10	Absorption variants	50
8.2.11	Combining (adjacency) law	50
8.2.12	De Morgan's theorem	50
8.3	Complete law reference	51
8.4	Simplification strategy	51
8.5	Worked examples	52
8.6	Tutorial 04 examples	54
8.7	Summary	55
9	Karnaugh Maps	56
9.1	What a K-map is	56
9.1.1	The Gray-code ordering	56
9.2	The three steps	57
9.3	A first example, step by step	57
9.4	Grouping rules	58
9.5	SoP versus PoS	60
9.6	Worked example: four variables, both forms	60
9.7	Second worked example	62
9.8	Don't-care conditions	63
9.9	Why the K-map works	65
9.10	Tutorial 06 exercise: the fire alarm	66
9.11	Summary	67
10	Single Gate Type Circuits	68
10.1	Why one gate type?	68
10.2	Functional completeness	68
10.3	{AND, NOT} is complete	68
10.4	{OR, NOT} is complete	69
10.5	NAND as a universal gate	69
10.5.1	NOT from NAND	69
10.5.2	AND from NAND	69
10.5.3	OR from NAND	70
10.6	NOR as a universal gate	70
10.6.1	NOT from NOR	70

10.6.2	OR from NOR	70
10.6.3	AND from NOR	70
10.7	Converting a circuit to NAND-only	71
10.8	The fire alarm in NAND-only and NOR-only form	72
10.9	Summary	73
III	Combinational Logic Circuits	74
11	Designing Combinational Circuits	75
11.1	Two kinds of digital circuit	75
11.2	Standard building blocks	75
11.3	The design procedure	76
11.4	Worked design: BCD to Excess-3 converter	76
11.5	A note on BCD	79
11.6	Summary	79
12	Adders, Subtractors and the ALU	80
12.1	Half adder	80
12.2	Full adder	81
12.2.1	Deriving the expressions	81
12.2.2	Full adder from two half adders	82
12.3	Ripple carry adder	83
12.4	Half subtractor	84
12.5	Full subtractor	84
12.5.1	Full subtractor from half subtractors	85
12.5.2	Multi-bit subtraction	85
12.6	The adder-subtractor	86
12.7	Multipliers	86
12.8	The Arithmetic Logic Unit	87
12.9	Summary	88
13	Decoders, Encoders and Multiplexers	89
13.1	Decoders	89
13.1.1	The 2×4 decoder	89
13.1.2	Decoders with an enable input	90
13.1.3	Building bigger decoders from smaller ones	90
13.1.4	Decoder as a general function generator	91
13.2	Encoders	92
13.2.1	4×2 encoder	92
13.2.2	8×3 encoder	92
13.3	Multiplexers	93
13.3.1	Internal structure	94
13.3.2	Multiplexer as a function generator	94
13.4	Demultiplexers	94
13.5	Where these blocks are used	95
13.6	Summary	96
IV	Sequential Logic Circuits	97
14	Sequential Logic Fundamentals	98
14.1	Combinational versus sequential	98

14.2	State and sequence	98
14.3	Asynchronous and synchronous circuits	99
14.4	The clock signal	99
14.5	Clocked synchronous sequential circuits	100
14.6	Level sensitive versus edge sensitive	100
14.7	Storage element family	101
14.8	Summary	102
15	Latches and Flip-Flops	103
15.1	The S-R latch	103
15.1.1	Characteristic equation	104
15.2	S-R latch with a control signal	104
15.3	The D latch	104
15.3.1	Timing behaviour	105
15.4	Edge triggering and the D flip-flop	105
15.4.1	Latch versus flip-flop timing, side by side	106
15.5	The J-K flip-flop	107
15.6	The T flip-flop	107
15.7	Characteristic and excitation tables	107
15.7.1	Characteristic tables — “given the inputs, what happens?”	107
15.7.2	Excitation tables — “I want this transition; what inputs?”	108
15.8	Summary	108
16	Registers, Counters and Memory	109
16.1	Registers	109
16.1.1	Storage register	109
16.1.2	Register with parallel load	110
16.2	Shift registers	110
16.3	Counters	111
16.3.1	Binary ripple counter	112
16.3.2	Synchronous binary counter	113
16.4	Random Access Memory	114
16.5	Summary	115
V	Computer Organisation and Architecture	116
17	CPU Organisation	117
17.1	The Central Processing Unit	117
17.1.1	Key components	117
17.1.2	Two conceptual halves	117
17.1.3	Hardware block diagram	118
17.2	The Control Unit	118
17.3	The Arithmetic and Logic Unit	118
17.4	Main memory	118
17.5	Buses	119
17.6	The memory controller	120
17.7	The clock	120
17.8	Summary	120

18 CPU Performance	121
18.1 Time and frequency units	121
18.2 Clock speed and clock cycle time	121
18.3 CPU execution time	122
18.3.1 Bringing in the instruction count	122
18.4 Cycles per instruction	123
18.5 Comparing two machines	123
18.6 Instruction count: static versus dynamic	124
18.7 Comparing code sequences	126
18.8 What limits throughput	126
18.9 Summary	127
19 Von Neumann Architecture and the FDE Cycle	128
19.1 Computer architecture	128
19.1.1 Notable early architectures	128
19.2 The stored program concept	128
19.3 Von Neumann architecture	129
19.3.1 The six defining principles	129
19.3.2 Components of a von Neumann machine	130
19.4 Registers	130
19.4.1 Key registers	131
19.4.2 The register sequence during execution	131
19.5 The Fetch–Decode–Execute cycle	131
19.5.1 Instruction processing in detail	132
19.5.2 The fetch phase in register transfer notation	132
19.5.3 Decode and execute	132
19.6 Interrupts	132
19.7 Summary	133
20 Instruction Set Architecture	134
20.1 What an ISA is	134
20.2 Levels of a modern computing system	134
20.3 Types of machine instruction	134
20.4 The example machine	134
20.5 Data transfer instructions	135
20.6 Arithmetic and logic instructions	136
20.7 Control instructions	137
20.8 Instruction encoding	138
20.8.1 Type 1 — register and immediate value	138
20.8.2 Type 2 — register and memory address	138
20.8.3 Type 3 — three registers	138
20.8.4 Type 4 — two registers	139
20.9 The complete instruction set	139
20.10 Assembling a program	139
20.11 Case study: sizing an architecture	141
20.12 Summary	142
21 Addressing Modes	143
21.1 The idea	143
21.2 Effective and physical addresses	143
21.3 The common addressing modes	144
21.3.1 Immediate addressing	144

21.3.2	Direct addressing	144
21.3.3	Indirect addressing	145
21.3.4	Register addressing (register direct)	146
21.3.5	Register indirect addressing	146
21.3.6	Displacement addressing	146
21.3.7	Stack addressing	147
21.4	Worked comparison	147
21.5	Summary	148
22	Instruction Level Pipelining	149
22.1	The idea	149
22.2	A typical set of stages	149
22.3	Overlapping execution	149
22.4	Pipeline efficiency	150
22.5	Hazards — why the ideal is not reached	151
22.6	Summary	152
VI	Memory Systems	154
23	Memory Technologies and the Hierarchy	155
23.1	Primary and secondary memory	155
23.2	Types of memory	155
23.2.1	Read Only Memory (ROM)	155
23.2.2	Random Access Memory (RAM)	156
23.2.3	SRAM versus DRAM	156
23.2.4	Flash memory	156
23.2.5	Solid state devices (SSD)	157
23.3	The memory hierarchy	158
23.4	The typical data access pattern	159
23.5	The CPU–memory gap	159
23.6	The principle of locality	160
23.7	Summary	161
24	Cache Memory	162
24.1	What a cache is	162
24.2	Cache hierarchy	162
24.3	Cache lines and cache entries	163
24.4	Cache performance	164
24.4.1	Effective access time	164
24.5	The three key design decisions	166
24.6	Write policy	166
24.6.1	The consistency problem	166
24.6.2	Write through	167
24.6.3	Write back	167
24.7	Summary	169
25	Cache Mapping Schemes	170
25.1	Why a mapping scheme is needed	170
25.2	Direct mapped cache	171
25.2.1	Splitting the address	171
25.2.2	A worked layout	172
25.2.3	Worked example: a conflict	173

25.2.4	How a lookup works	174
25.3	Field-width exercises	174
25.3.1	Advantages and disadvantages of direct mapping	176
25.4	Fully associative cache	176
25.5	Set associative cache	177
25.5.1	The associativity spectrum	177
25.5.2	Address split for set associative	178
25.5.3	Worked example: 2-way set associative	178
25.6	Cache replacement policies	180
25.6.1	The common policies	180
25.6.2	Worked comparison	180
25.6.3	Advantages and disadvantages	181
25.7	Comparing the three mapping schemes	182
25.8	Summary	183
VII	Appendices	184
A	Formula and Rule Sheet	185
A.1	Counting and sizing	185
A.2	Number representation	185
A.3	Floating point	186
A.4	Boolean algebra	186
A.5	Gate truth tables	187
A.6	Canonical forms	187
A.7	Karnaugh map rules	187
A.8	Combinational building blocks	187
A.9	Flip-flops	188
A.10	Performance	188
A.11	Fetch cycle and pipelining	188
A.12	Memory and cache	189
A.13	Addressing modes	189
B	Reference Tables	190
B.1	Powers of two	190
B.2	Base conversion table	191
B.3	Binary fractions	191
B.4	Excess- K reference (4-bit fields)	192
B.5	BCD and Excess-3	192
B.6	Minterms and maxterms (three variables)	193
B.7	K-map cell indices	193
B.8	Example machine instruction set	194
C	Remaining Tutorial Exercises, Solved	195
C.1	Tutorial 03 — truth tables	195
C.2	Tutorial 05 — K-map simplification, validated algebraically	197
C.3	Further practice	199
C.4	Answers to the further practice	200

How to use these notes

What was consolidated

The source material contains a good deal of repetition: the lecture slides and the tutorial slides cover many of the same topics, and several topics recur across lectures. In these notes each topic is treated *once*, in the place where it fits best, drawing on every source that covers it. The main merges are:

Topic	Sources merged	Now in
Logic gates and truth tables	Lec 2.1.1, Tutorial 03	Chapter 6
Boolean laws and axioms	Lec 2.2.1, Tutorial 04	Chapter 8
Karnaugh maps	Lec 2.2.1, Tutorials 05, 06	Chapter 9
Functional completeness	Lec 2.2.1, Tutorial 05	Chapter 10
Adders and subtractors	Lec 2.3.1, Tutorial 07	Chapter 12
Decoders, encoders, MUX	Lec 2.3.1, Tutorial 07	Chapter 13
Latches and flip-flops	Lec 2.4.1, Tutorial 08	Chapter 15
Number systems	Tutorial 01	Chapter 2
Complements and arithmetic	Tutorial 02	Chapters 3, 4
Floating point	Lec 3.2.1, Tutorial 02	Chapter 5
RAM organisation	Lec 2.4.1, Lec 4.1.1	Chapters 16, 23
Memory and caching	Lec 4.1.1	Chapters 23–25

How the parts map onto the course

Lecture 4.1.1 notes that the exam is a **two-hour paper with four compulsory questions, one from each section of the course**. The lecture numbering suggests the following correspondence, which is worth keeping in mind when revising:

Course section	Sources	Parts of these notes
1 — Data representation	Tutorials 01–02, Lec 3.2.1	Part I
2 — Digital logic	Lec 2.1–2.4, Tutorials 03–08	Parts II, III, IV
3 — Computer organisation	Lec 3.1, 3.2, 3.3	Part V
4 — Memory systems	Lec 4.1.1	Part VI

Confirm this mapping

The grouping above is inferred from the lecture numbering, not stated explicitly in the slides. Check it against your course outline before relying on it to allocate revision time.

Reading the boxes

Definition

Formal statements of terms. If a definition appears in a box, it is the version worth memorising — exam questions frequently ask for exactly this wording.

Key idea

The intuition behind a rule or technique — why it works, not just what it says.

Watch out

Common mistakes, sign errors, off-by-one traps, and places where the slides use loose or inconsistent notation.

Beyond the slides

Material the lecture slides left as “search and find”, “homework”, or “explore”. It is written up here for completeness; you should confirm with your lecturer whether it is examinable.

Worked examples appear in shaded boxes and are numbered per chapter. Nearly all of them are taken directly from the lectures or tutorials, so the numbers should match what you saw in class.

Notation used throughout

Symbol	Meaning	Symbol	Meaning
$A \cdot B$, AB , $A \wedge B$	AND	$\overline{A}$, A' , $\neg A$	NOT
$A + B$, $A \vee B$	OR	$A \oplus B$	XOR
$\overline{A \cdot B}$	NAND	$\overline{A + B}$	NOR
$\sum m(\dots)$	sum of minterms	$\prod M(\dots)$	product of maxterms
N_2 , N_8 , N_{10} , N_{16}	number in base 2/8/10/16	$\times$	don't-care cell
$Q(t)$	present state	$Q(t+1)$	next state

A note on the slides' notation

The lecture slides mix notations freely — $\overline{A}$, A' and $\neg A$ all appear for NOT, and both $\cdot$ and juxtaposition are used for AND. These notes use $\overline{A}$ and $\cdot$ as the primary forms but keep the alternatives visible, because your exam paper may use any of them.

PART I

Data Representation

Chapter 1

Bits, Bytes and Words

Where this fits

Everything a computer does is ultimately the manipulation of two voltage levels. This chapter fixes the vocabulary — bit, byte, word, addressability — that the rest of the notes depends on. Source: Tutorial 01, with the addressing material from Lecture 3.1 brought forward.

1.1 The bit

Bit

A **bit** (binary digit) is the smallest unit of data in a computer. It takes exactly one of two values, 0 or 1.

Physically a bit is a voltage level, a magnetic orientation, a charge in a capacitor, or a pit on an optical disc. Logically none of that matters: the abstraction is simply “one of two states”. This two-state abstraction is what makes Boolean algebra (Part II) the natural mathematics of computing.

Two states may seem impoverished, but n bits give 2^n distinct patterns, and 2^n grows very quickly:

n	1	4	8	16	32
2^n	2	16	256	65 536	4 294 967 296

The counting rule you will use constantly

n bits $\Rightarrow 2^n$ different patterns.

Conversely, to represent k distinct things you need $\lceil \log_2 k \rceil$ bits.

This single rule answers: how many rows in a truth table, how many cells in a K-map, how many memory addresses, how many decoder outputs, how many registers an instruction field can name.

1.2 The byte and the storage hierarchy

Byte

A **byte** (binary term) is a group of 8 bits. It is the standard unit in which memory capacity is quoted and, on most modern machines, the smallest individually addressable unit of memory.

One byte holds $2^8 = 256$ patterns — enough for the original ASCII character set with room to spare, which is the historical reason for the choice.

Unit	Definition	Bytes
1 bit	0 or 1	—
1 byte (B)	8 bits	2^0
1 kilobyte (kB)	1024 bytes	2^{10}
1 megabyte (MB)	1024 kB	2^{20}
1 gigabyte (GB)	1024 MB	2^{30}
1 terabyte (TB)	1024 GB	2^{40}
1 petabyte (PB)	1024 TB	2^{50}
1 exabyte (EB)	1024 PB	2^{60}

1024 or 1000?

The tutorial uses the binary convention: 1 kB = 1024 bytes. Storage manufacturers use the decimal convention: 1 kB = 1000 bytes. This is why a “1 TB” drive shows up as roughly 931 GB in an operating system that uses the binary convention — $10^{12}/2^{40} \approx 0.909$.

The IEC introduced unambiguous names for the binary units — kibibyte (KiB), mebibyte (MiB), gibibyte (GiB) — but the course material uses the older convention throughout, so **use 1024 unless told otherwise**.

1.3 Word size

Word

A **word** is the natural unit of data for a particular processor: the number of bits it moves and processes in one go. Typical word sizes are 8, 16, 32 and 64 bits.

Word size is not an arbitrary label. It usually determines, simultaneously:

- the width of the general-purpose registers;
- the width of the ALU, and therefore how many bits can be added in one operation;
- the width of the data bus (often, though not always);
- the size of the integers the machine handles “naturally”.

A 64-bit processor can add two 64-bit integers in a single instruction; a 32-bit processor needs several instructions to do the same job.

1.4 Memory organisation and addressability

Main memory is a linear array of storage cells. Each cell has a unique **address**, and the question of what one address refers to is the question of *addressability*.

Byte addressable		Word addressable	
Address	0	0	32 bits
	1	1	32 bits
	2	2	32 bits
	3	3	32 bits
	4	4	32 bits
	5	5	32 bits
one address = one byte		one address = one word (here 4 bytes)	

Byte addressable vs. word addressable

In a **byte-addressable** memory every individual byte has its own address. In a **word-addressable** memory each address refers to a whole word, so consecutive addresses are one word apart.

The three quantities you will be asked to compute are:

Memory sizing formulas

For an address length of n bits:

$$\text{number of addresses} = 2^n$$

$$\text{address range} = 0 \text{ to } 2^n - 1$$

$$\text{total memory size} = 2^n \times (\text{size of one addressable unit})$$

Worked Example 1 *Byte-addressable memory — Lecture 3.1*

Q. The address length of a memory system is 10 bits and it is byte addressable. Find (a) the maximum number of addresses, (b) the maximum memory size, (c) the address range.

(a) Each address is a distinct 10-bit pattern, so

$$2^{10} = 1024 \text{ addresses.}$$

(b) Each address holds one byte, so

$$2^{10} \times 1 \text{ byte} = 1024 \text{ bytes} = 1 \text{ kB.}$$

(c) Addresses are numbered from zero:

$$0 \text{ to } 2^{10} - 1 = 0 \text{ to } 1023.$$

Worked Example 2 *Word-addressable memory — Lecture 3.1*

Q. The address length is 10 bits, the memory is word addressable, and the word size is 32 bits. Find the same three quantities.

(a) The number of addresses depends *only* on the address length:

$$2^{10} = 1024 \text{ addresses.}$$

(b) Each address now holds a 32-bit word, i.e. 4 bytes:

$$2^{10} \times 4 \text{ bytes} = 4096 \text{ bytes} = 4 \text{ kB.}$$

(c)

$$0 \text{ to } 2^{10} - 1 = 0 \text{ to } 1023.$$

The trap in these questions

Notice that (a) and (c) are *identical* in both worked examples. The address length alone fixes how many addresses exist and what the range is; the addressable unit only affects the total capacity. Students routinely multiply the address count by the word size when asked for the number of addresses.

Why addressability matters later

Addressability reappears in the fetch cycle (Chapter 19). After fetching an instruction the CPU must advance the program counter to the next instruction:

$$PC \leftarrow PC + \begin{cases} 1 & \text{word addressable} \\ 2 & \text{byte addressable, 16-bit instructions} \\ 4 & \text{byte addressable, 32-bit instructions} \end{cases}$$

The increment is “however many addressable units one instruction occupies”.

1.5 Summary

Term	One-line definition
Bit	Smallest data unit; one of two states, 0 or 1.
Byte	8 bits; usual smallest addressable unit.
Word	Number of bits the CPU processes at once; sets register and ALU width.
Address	Unique identifier of a memory location.
Address length	Number of bits in an address; n bits $\Rightarrow 2^n$ locations.
Byte addressable	One address names one byte.
Word addressable	One address names one word.

Chapter 2

Number Systems and Conversion

Where this fits

Four bases matter in computing: binary (what the hardware uses), decimal (what humans use), and octal and hexadecimal (compact shorthands for binary). This chapter gives a single method for every conversion between them. Source: Tutorial 01.

2.1 The four bases

System	Base	Digits	Bits per digit
Binary	2	0, 1	1
Octal	8	0–7	3
Decimal	10	0–9	—
Hexadecimal	16	0–9, <i>A–F</i>	4

Hexadecimal digits *A* through *F* stand for 10 through 15:

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bin	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Why octal and hexadecimal exist

Because $8 = 2^3$ and $16 = 2^4$, each octal digit corresponds to exactly three bits and each hex digit to exactly four bits. Conversion is therefore *pure regrouping* — no arithmetic at all. Long binary strings become readable: 11010110_2 is $D6_{16}$.

Decimal enjoys no such relationship with 2, which is exactly why decimal↔binary conversions require real division or addition.

2.2 Positional value: any base to decimal

Every positional number system works the same way. A number $d_k d_{k-1} \dots d_1 d_0$ in base b has value

$$\sum_{i=0}^k d_i \times b^i$$

That single formula handles binary→decimal, octal→decimal and hexadecimal→decimal.

Worked Example 1 Binary to decimal

$101010_2 = ?$

Bit	1	0	1	0	1	0
Weight	2^5	2^4	2^3	2^2	2^1	2^0
	32	16	8	4	2	1
Contributes	32	0	8	0	2	0

$$32 + 8 + 2 = 42_{10}$$

Similarly $111_2 = 4 + 2 + 1 = 7_{10}$.

Worked Example 2 *Hexadecimal to decimal*

$1C2_{16} = ?$

$$1 \times 16^2 + C \times 16^1 + 2 \times 16^0 = 256 + 12(16) + 2 = 256 + 192 + 2 = 450_{10}$$

And $FF_{16} = 15(16) + 15 = 255_{10}$ — the largest value in one byte.

Worked Example 3 *Octal to decimal*

$$45_8 = 4 \times 8 + 5 = 37_{10} \quad 102_8 = 1 \times 64 + 0 \times 8 + 2 = 66_{10}$$

2.3 Decimal to any base: repeated division

The method

Divide the decimal number repeatedly by the target base, recording the **remainder** at each step. Stop when the quotient reaches 0. Then read the remainders **bottom to top**.

Worked Example 4 *Decimal to binary*

$20_{10} = ?_2$

$$\begin{array}{ll} 20 \div 2 = 10 & \text{remainder } \mathbf{0} \\ 10 \div 2 = 5 & \text{remainder } \mathbf{0} \\ 5 \div 2 = 2 & \text{remainder } \mathbf{1} \\ 2 \div 2 = 1 & \text{remainder } \mathbf{0} \\ 1 \div 2 = 0 & \text{remainder } \mathbf{1} \quad \uparrow \text{ read upwards} \end{array}$$

$$20_{10} = 10100_2$$

Check: $16 + 4 = 20$. ✓

Other examples from the tutorial: $5_{10} = 101_2$, $128_{10} = 10000000_2$.

Worked Example 5 *Decimal to octal and hexadecimal*

$342_{10} = ?_8$

$$342 \div 8 = 42 \text{ r } \mathbf{6}, \quad 42 \div 8 = 5 \text{ r } \mathbf{2}, \quad 5 \div 8 = 0 \text{ r } \mathbf{5} \Rightarrow 526_8$$

Check: $5(64) + 2(8) + 6 = 320 + 16 + 6 = 342$. ✓

$2000_{10} = ?_{16}$

$$2000 \div 16 = 125 \text{ r } \mathbf{0}, \quad 125 \div 16 = 7 \text{ r } \mathbf{13} = \mathbf{D}, \quad 7 \div 16 = 0 \text{ r } \mathbf{7} \Rightarrow 7D0_{16}$$

Check: $7(256) + 13(16) + 0 = 1792 + 208 = 2000$. ✓

A shortcut for decimal to binary

Rather than dividing, subtract the largest power of two that fits, repeatedly. For 452_{10} :

$$452 - 256 = 196, \quad 196 - 128 = 68, \quad 68 - 64 = 4, \quad 4 - 4 = 0$$

So the bits set are 256, 128, 64, 4, giving 111000100_2 . Many people find this faster and less error-prone than long division, and it doubles as a check on the division method.

2.4 Binary $\leftrightarrow$ hexadecimal: group in fours

The method

Binary $\rightarrow$ hex: group the bits into fours *starting from the right*, padding the leftmost group with leading zeros; convert each group to one hex digit.

Hex $\rightarrow$ binary: replace each hex digit by its four-bit pattern.

Worked Example 6 Binary to hexadecimal

$11001000_2 = ?_{16}$

$$\begin{array}{cc} 1100 & 1000 \\ \downarrow & \downarrow \\ \text{C} & 8 \end{array}$$

$$11001000_2 = C8_{16}$$

1010011010_2 : grouping from the right gives 10 1001 1010, pad to 0010 1001 1010 $\Rightarrow 29A_{16}$.

Always group from the right

Grouping from the left silently changes the value. 1110110_2 has seven bits; grouped from the right it is $0111\ 0110 = 76_{16}$. Grouped from the left it would be $1110\ 1100 = EC_{16}$, which is wrong.

Worked Example 7 Hexadecimal to binary

$7A_{16}$: $7 \rightarrow 0111_2$, $A \rightarrow 1010_2$

$$7A_{16} = 01111010_2 = 1111010_2$$

230_{16} : $2 \rightarrow 0010$, $3 \rightarrow 0011$, $0 \rightarrow 0000 \Rightarrow 1000110000_2$.

2.5 Binary $\leftrightarrow$ octal: group in threes

Identical in spirit, with groups of three instead of four.

Worked Example 8 Binary to octal

101110_2 : groups 101 110 $\Rightarrow 5, 6 \Rightarrow 56_8$.

11101001_2 : pad to $011\ 101\ 001 \Rightarrow 3, 5, 1 \Rightarrow 351_8$.

Worked Example 9 *Octal to binary*

21_8 : $2 \rightarrow 010$, $1 \rightarrow 001 \Rightarrow 010001_2 = 10001_2$.

317_8 : $3 \rightarrow 011$, $1 \rightarrow 001$, $7 \rightarrow 111 \Rightarrow 011001111_2 = 11001111_2$.

2.6 Octal $\leftrightarrow$ hexadecimal: go via binary

There is no direct digit-to-digit correspondence between octal and hex, because 3 and 4 do not divide each other. The reliable route is always

octal $\rightarrow$ binary (groups of 3) $\rightarrow$ regroup into 4s $\rightarrow$ hex

Worked Example 10 *Octal to hexadecimal*

$1750_8 = ?_{16}$

Octal	1	7	5	0
Binary	001	111	101	000
Regroup	0011	1110	1000	
Hex	3	E	8	

$$1750_8 = 3E8_{16}$$

Check: both equal 1000_{10} . ✓

Similarly $30_8 = 011000_2 \rightarrow 00011000_2 = 18_{16}$.

Worked Example 11 *Hexadecimal to octal*

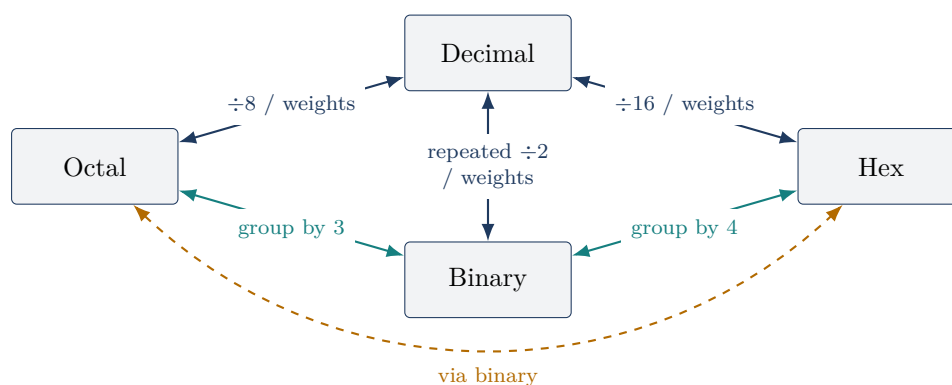
$1DA_{16} = ?_8$

$1 \rightarrow 0001$, $D \rightarrow 1101$, $A \rightarrow 1010 \Rightarrow 000111011010_2$

Regroup in threes from the right: $000\ 111\ 011\ 010 \Rightarrow 0, 7, 3, 2 \Rightarrow 732_8$.

345_{16} : $001101000101_2 \rightarrow 001\ 101\ 000\ 101 \Rightarrow 1505_8$.

2.7 Conversion map



Strategy

Binary is the hub. If you are ever unsure of a route, convert to binary first and then out again — it costs one extra step and never goes wrong.

2.8 Tutorial 01 exercises, with answers**Decimal $\rightarrow$ binary**

	Given	Answer	Check
(a)	63_{10}	111111_2	$32+16+8+4+2+1 = 63$
(b)	75_{10}	1001011_2	$64+8+2+1 = 75$
(c)	452_{10}	111000100_2	$256+128+64+4 = 452$

Binary $\rightarrow$ decimal

	Given	Answer	Working
(a)	110110110_2	438_{10}	$256+128+32+16+4+2$
(b)	11010011_2	211_{10}	$128+64+16+2+1$
(c)	1000110000_2	560_{10}	$512+32+16$

Hex $\rightarrow$ binary, binary $\rightarrow$ hex

Given	Answer	Given	Answer
$41F_{16}$	10000011111_2	100101001_2	129_{16}
$7D0_{16}$	11111010000_2	100111011_2	$13B_{16}$
		1110110_2	76_{16}

Decimal $\leftrightarrow$ hex

Given	Answer	Given	Answer
2000_{10}	$7D0_{16}$	$1F4_{16}$	500_{10}
3333_{10}	$D05_{16}$	457_{16}	1111_{10}

Binary $\rightarrow$ octal, octal $\rightarrow$ hex

Given	Answer	Given	Answer
11101001_2	351_8	3720_8	$7D0_{16}$
10010110_2	226_8	11610_8	1388_{16}
1001011000_2	1130_8		

Verify by a second route

Every one of these can be checked by converting the answer back, or by taking a different path through the conversion map. In an exam this costs seconds and catches the overwhelming majority of slips.

Chapter 3

Binary Arithmetic

Where this fits

The four basic operations in base 2. Addition is the one that matters most: subtraction is implemented as addition of a complement (Chapter 4), multiplication as repeated shift-and-add, and the hardware that performs it is built in Chapter 12. Source: Tutorial 02.

3.1 Binary addition

There are only four single-bit cases, plus the three-input case that arises once carries appear.

Two operands		
Sum	Result	Carry
$0 + 0$	0	0
$0 + 1$	1	0
$1 + 0$	1	0
$1 + 1$	0	1

With a carry in		
Sum	Result	Carry
$1 + 1 + 0$	0	1
$1 + 1 + 1$	1	1

Reading the table as logic

Look at the two-operand table. The *Result* column is exactly the XOR truth table and the *Carry* column is exactly the AND truth table:

$$\text{Sum} = a \oplus b, \quad \text{Carry} = a \cdot b$$

That observation *is* the half adder (Section 12.1). Binary arithmetic and Boolean logic are the same subject seen from two directions.

Worked Example 1 *Multi-bit addition*

Add 1011_2 and 0110_2 .

$$\begin{array}{r} \text{carry } 1\ 1\ 1 \\ 1\ 0\ 1\ 1 \\ +\ 0\ 1\ 1\ 0 \\ \hline 1\ 0\ 0\ 0\ 1 \end{array}$$

Column by column, right to left: $1+0 = 1$; $1+1 = 0$ carry 1; $0+1+1 = 0$ carry 1; $1+0+1 = 0$ carry 1; final carry out = 1.

Check in decimal: $11 + 6 = 17 = 10001_2$. ✓

3.2 Binary subtraction

Difference	Result	Note
$0 - 0$	0	
$0 - 1$	1	with a borrow from the next column
$1 - 0$	1	
$1 - 1$	0	

The only interesting case is $0 - 1$: you must borrow 1 from the column to the left, which turns the 0 into 10_2 ($= 2$), and $2 - 1 = 1$.

Worked Example 2 *Multi-bit subtraction with borrow*

$1100_2 - 0001_2$.

$$\begin{array}{r}
 \text{borrow} \quad 1 \quad 1 \\
 1 \quad 1 \quad 0 \quad 0 \\
 - 0 \quad 0 \quad 0 \quad 1 \\
 \hline
 1 \quad 0 \quad 1 \quad 1
 \end{array}$$

Rightmost column: $0 - 1$ needs a borrow. The next column is also 0, so the borrow propagates two places to the 1 in position 2. Result 1011_2 .

Check: $12 - 1 = 11 = 1011_2$. ✓

Direct subtraction is not what hardware does

Borrow-propagation logic is awkward to build and does not handle negative results gracefully. Real machines subtract by *adding the two's complement* of the subtrahend (Chapter 4). Dedicated subtractor circuits exist (Section 12.5) and are examinable, but a general-purpose ALU usually contains only an adder.

3.3 Binary multiplication

Product	Result
0×0	0
0×1	0
1×0	0
1×1	1

Single-bit multiplication *is* the AND operation. Multi-bit multiplication uses the same long-multiplication algorithm as decimal, but it is far easier, because each partial product is either a copy of the multiplicand or zero — there is no multiplication table to remember.

Worked Example 3 *Long multiplication*

$1011_2 \times 101_2$ (11×5).

$$\begin{array}{r}
 \begin{array}{r}
 101 \\
 \times 10 \\
 \hline
 101 \\
 0000 \\
 + 1011 \\
 \hline
 110111
 \end{array}
 \qquad
 \begin{array}{r}
 1 \\
 1 \\
 \times 1 \\
 \times 0, \text{ shifted} \\
 \times 1, \text{ shifted twice}
 \end{array}
 \end{array}$$

$$110111_2 = 32 + 16 + 4 + 2 + 1 = 55_{10} = 11 \times 5. \checkmark$$

Shift-and-add

Because multiplying by 2 in binary is a left shift, multiplication reduces to: for each 1 bit in the multiplier at position i , add the multiplicand shifted left by i . This is why a hardware multiplier is built from adders and shifters (Section 12.7) and why multiplication takes more cycles than addition (Chapter 18).

3.4 Binary division

Long division in base 2 is simpler than in decimal because at each step the quotient digit can only be 0 or 1 — you are only asking “does the divisor fit?”

Worked Example 4 Long division

$$110111_2 \div 101_2 \quad (55 \div 5).$$

$$\begin{array}{r}
 1011 \quad (\text{quotient}) \\
 101 \overline{) 110111} \\
 \underline{101} \\
 111 \\
 \underline{101} \\
 10 \\
 \underline{101} \\
 101 \\
 \underline{101} \\
 0 \quad (\text{remainder})
 \end{array}$$

$$\text{Quotient } 1011_2 = 11_{10}, \text{ remainder } 0. \text{ Check: } 55 \div 5 = 11. \checkmark$$

The algorithm

1. Bring down bits from the dividend until the partial remainder is $\geq$ the divisor.
2. Write 1 in the quotient and subtract; otherwise write 0.
3. Repeat until all dividend bits are consumed. What is left is the remainder.

3.5 Summary of operator–gate correspondence

Arithmetic operation	Bit-level behaviour	Gate
Sum bit of $a + b$	1 when exactly one input is 1	XOR
Carry bit of $a + b$	1 only when both inputs are 1	AND
Product $a \times b$	1 only when both inputs are 1	AND
Borrow of $a - b$	1 only when $a = 0, b = 1$	$\bar{a} \cdot b$

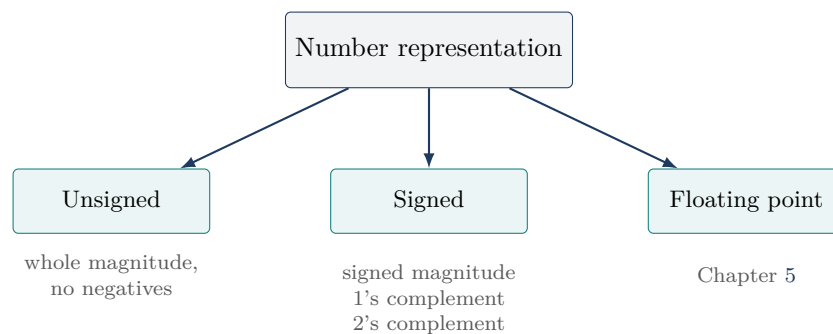
These four rows are the bridge between Part I and Part III: they are literally the specifications of the half adder and half subtractor.

Chapter 4

Integer Representation

Where this fits

A bit pattern has no intrinsic meaning. 10000100_2 is 132, or -4 , or -123 , or -124 , depending entirely on which *convention* we agree to use. This chapter covers the four conventions and shows why two's complement won. Source: Tutorial 02, extended with range and overflow analysis.



4.1 Unsigned representation

Every bit contributes magnitude; there is no sign.

Largest	1	1	1	1	1	1	1	1	= 255 ₁₀ (largest)
	128	64	32	16	8	4	2	1	
Smallest	0	0	0	0	0	0	0	0	= 0 ₁₀ (smallest)

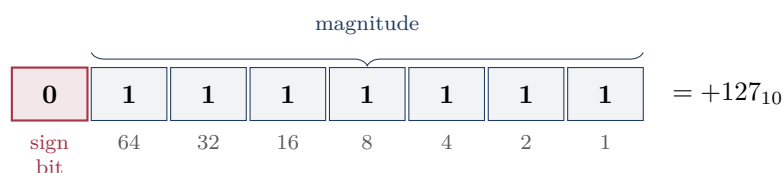
Range

For n bits, unsigned range is 0 to $2^n - 1$. For $n = 8$: 0 to 255.

Unsigned is the right choice for things that cannot be negative: memory addresses, array indices, counters, character codes.

4.2 Signed magnitude

The most obvious scheme: reserve the leftmost bit as a **sign bit** (0 = positive, 1 = negative) and let the remaining $n - 1$ bits carry the magnitude exactly as before.



Range

For n bits: $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$. For $n = 8$: -127 to $+127$ — roughly half the unsigned range, as one bit has been given up to the sign.

4.2.1 Two problems with signed magnitude

Problem 1: two zeros

$$00000000_2 = +0 \qquad 10000000_2 = -0$$

Two distinct bit patterns for the same value. This wastes an encoding and, worse, means a hardware equality test cannot simply compare bit patterns.

Problem 2: arithmetic does not work

Worked Example 1 *Why naive signed-magnitude addition fails*

Compute $7 - 4$ as $7 + (-4)$ using 8-bit signed magnitude.

$$\begin{array}{rcl} 00000111 & = & +7 \\ + 10000100 & = & -4 \\ \hline 10001011 & = & -11 \quad \text{wrong} \end{array}$$

The adder blindly added the sign bits as if they were magnitude. To get the right answer, hardware would have to: inspect both signs, compare magnitudes, decide whether to add or subtract, and work out the sign of the result. That is a lot of extra circuitry for a very common operation.

The design goal

We want a representation in which **subtraction is just addition** and the adder never needs to know the signs of its operands. Complement representations achieve exactly this.

4.3 One's complement

One's complement

The one's complement of a binary number is obtained by **inverting every bit**: every 0 becomes 1 and every 1 becomes 0. A negative number $-x$ is represented as the one's complement of x .

$$10110101 \xrightarrow{\text{invert}} 01001010$$

4.3.1 Arithmetic: the end-around carry

The rule

Add normally. **If a carry comes out of the most significant bit, add that carry back into the least significant bit** (the “end-around carry”).

Worked Example 2 $7 - 4$ in one's complementStep 1 — form -4 :

$$\begin{array}{rcl}
 4 & = & 00000100 \\
 & & 11111011 \quad \leftarrow \text{one's complement} = -4
 \end{array}$$

Step 2 — add:

$$\begin{array}{rcl}
 & 1111111 & \text{carries} \\
 & 00000111 & = 7 \\
 + & 11111011 & = -4 \\
 \hline
 & 1\,00000010 &
 \end{array}$$

Step 3 — end-around carry: a 1 has emerged from the top, so add it back:

$$\begin{array}{rcl}
 & 00000010 & \\
 + & 00000001 & \\
 \hline
 & 00000011 & = 3_{10} \quad \checkmark
 \end{array}$$

Worked Example 3 $115 - 27$ in one's complement $27 = 00011011_2$, so $-27 = 11100100_2$.

$$\begin{array}{rcl}
 & 11 & \text{carries} \\
 & 01110011 & = 115 \\
 + & 11100100 & = -27 \\
 \hline
 & 1\,01010111 & \\
 + & 00000001 & \text{end-around carry} \\
 \hline
 & 01011000 & = 88_{10} \quad \checkmark
 \end{array}$$

One's complement still has two zeros

$00000000_2 = +0$ and $11111111_2 = -0$. Range for n bits is $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$ — no better than signed magnitude. The end-around carry also costs an extra addition step. This is why one's complement is a historical curiosity rather than a live standard.

4.4 Two's complement

Two's complementThe two's complement of a binary number is its **one's complement plus one**:

$$2\text{'s comp}(x) = 1\text{'s comp}(x) + 1 = \bar{x} + 1$$

$$\begin{array}{rcl}
 00000111 & \leftarrow & \text{binary number } (7) \\
 11111000 & \leftarrow & \text{one's complement} \\
 + & 1 & \\
 \hline
 11111001 & \leftarrow & \text{two's complement } (-7)
 \end{array}$$

The rule — note how much simpler it is

Add normally and **discard any carry out of the most significant bit**. No end-around carry, no sign inspection, no magnitude comparison. One adder handles both addition and subtraction.

Worked Example 4 $7 - 4$ in two's complement

Form -4 :

$$\begin{array}{rcl} \text{binary} & 00000100 & = 4 \\ \text{1's complement} & 11111011 & \\ +1 & 11111100 & = -4 \end{array}$$

Add:

$$\begin{array}{rcl} & \textcolor{red}{11111} & \text{carries} \\ & 00000111 & = 7 \\ + & 11111100 & = -4 \\ \hline & 1\ 00000011 & = 3_{10} \quad \checkmark \end{array}$$

The leading 1 is discarded — it is not part of an 8-bit result.

Worked Example 5 $115 - 27$ in two's complement

$27 = 00011011_2$; 1's complement 11100100_2 ; +1 gives $-27 = 11100101_2$.

$$\begin{array}{rcl} & \textcolor{red}{11} \ \textcolor{red}{111} & \text{carries} \\ & 01110011 & = 115 \\ + & 11100101 & = -27 \\ \hline & 1\ 01011000 & = 88_{10} \quad \checkmark \end{array}$$

4.4.1 The shortcut

Two's complement by inspection

Scanning from the **right**: copy bits up to and including the first 1, then invert everything to the left of it.

$$\begin{array}{ccccccc} \underbrace{0011} & \underbrace{0} & \underbrace{100} & & 00110100_2 & \rightarrow & 11001100_2 \\ \text{invert} \rightarrow 1100 & \text{invert} \rightarrow 1 & \text{copy} & & & & \end{array}$$

Faster and less error-prone than inverting and adding under exam pressure.

4.4.2 Why there is only one zero

	Bit pattern	Result
+0	00000000	—
negate it	11111111 (invert)	
	00000000 (+1, carry discarded)	back to +0

Negating zero gives zero. The pattern 10000000_2 , which was -0 in the other schemes, is now free and is used for -128 . This is why two's complement has one extra negative value.

4.5 Range comparison

Scheme	Range (n bits)	$n = 8$	Zeros
Unsigned	$0 \dots 2^n - 1$	$0 \dots 255$	1
Signed magnitude	$-(2^{n-1}-1) \dots + (2^{n-1}-1)$	$-127 \dots + 127$	2
One's complement	$-(2^{n-1}-1) \dots + (2^{n-1}-1)$	$-127 \dots + 127$	2
Two's complement	$-2^{n-1} \dots + (2^{n-1}-1)$	$-128 \dots + 127$	1

Property	Unsigned	Sign-mag.	1's comp.	2's comp.
Single zero	yes	no	no	yes
Subtract via adder	n/a	no	yes	yes
Extra carry step	no	n/a	yes	no
Sign from MSB alone	n/a	yes	yes	yes
Used in modern hardware	yes	rarely	no	yes

Verdict

Two's complement wins on every axis that matters: one zero, subtraction by the same adder used for addition, no correction step, and the sign readable from the leading bit. It is what essentially every processor uses for integers, and it is what the **ADD** instruction in the example machine of Chapter 20 assumes.

4.6 Reading a two's complement number

Two equivalent methods

Method A — negate and read. If the sign bit is 1, take the two's complement of the whole pattern, read the result as unsigned, and attach a minus sign.

Method B — negative weight on the MSB. Treat the leading bit as carrying weight -2^{n-1} and all others as usual:

$$11111100_2 = -128 + 64 + 32 + 16 + 8 + 4 = -4 \quad \checkmark$$

Worked Example 6 *Interpreting one pattern four ways*

What is 10000100_2 ?

Interpretation	Value	Reasoning
Unsigned	132_{10}	$128 + 4$
Signed magnitude	-4_{10}	sign = 1, magnitude $0000100_2 = 4$
One's complement	-123_{10}	invert $\rightarrow 01111011_2 = 123$
Two's complement	-124_{10}	$-128 + 4$

The bit pattern alone tells you nothing. You must know the convention.

Overflow detection in two's complement

Overflow means the true result does not fit in n bits. Two equivalent tests:

Test 1 (sign rule). Overflow occurs only when the two operands have the *same* sign and the result has the *opposite* sign. Adding numbers of opposite signs can never overflow.

Test 2 (carry rule). Overflow occurs when the carry *into* the most significant bit differs from the carry *out* of it:

$$V = C_{n-1} \oplus C_n$$

Example in 8-bit two's complement: $100 + 50 = 150$, but the maximum is 127.

$$\begin{array}{rcl} 01100100 & = & +100 \\ + 00110010 & = & +50 \\ \hline 10010110 & = & -106 \quad \text{overflow} \end{array}$$

Two positives produced a negative — Test 1 fires. Carry into the MSB is 1, carry out is 0 — Test 2 fires too.

Do not confuse overflow with carry-out. In two's complement arithmetic a carry out of the MSB is normal and is discarded; it is the *mismatch* between the two carries that signals overflow. In *unsigned* arithmetic it is the opposite: carry-out *is* the overflow indicator.

Sign extension

To widen a two's complement number from n bits to $m > n$ bits, replicate the sign bit into all the new positions:

$$1100_2 (-4, 4 \text{ bits}) \longrightarrow 11111100_2 (-4, 8 \text{ bits})$$

$$0100_2 (+4, 4 \text{ bits}) \longrightarrow 00000100_2 (+4, 8 \text{ bits})$$

Padding with zeros instead would turn -4 into $+12$. This is what a sign-extending load instruction does, and why processors distinguish “load byte” from “load byte unsigned”.

Fractional and Floating Point Representation

Where this fits

Integers are not enough — 1049835.647497642 has to be storable too. This chapter builds the floating point representation from binary fractions upward, covers the Excess- K encoding used for exponents, and analyses the two unavoidable consequences: finite range and finite precision. Sources: Lecture 3.2.1 and Tutorial 02, merged.

5.1 Binary fractions

Positional notation extends to the right of the point exactly as you would expect, with *negative* powers of the base.

1	0	1	.	0	1	1	=	
2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}		
4	0	1		0	$\frac{1}{4}$	$\frac{1}{8}$		$= 4 + 0 + 1 + \frac{1}{4} + \frac{1}{8} = 5\frac{3}{8} = 5.375_{10}$

$$101.011_2 = 5.375_{10}$$

5.1.1 Decimal fraction to binary: repeated multiplication

The method (mirror image of integer conversion)

Multiply the fractional part by 2 repeatedly. At each step the **integer part** of the product (always 0 or 1) is the next bit; carry the remaining fraction forward. Read the bits **top to bottom**. Stop when the fraction reaches 0 — or when you run out of mantissa bits.

Worked Example 1 Converting 0.75_{10}

$$0.750 \times 2 = 1.500 \quad \text{bit} = 1$$

$$0.500 \times 2 = 1.000 \quad \text{bit} = 1$$

fraction is 0 — stop

$$0.75_{10} = 0.11_2$$

Check: $\frac{1}{2} + \frac{1}{4} = 0.75$. ✓

Similarly 0.625_{10} : $0.625 \times 2 = 1.25$, $0.25 \times 2 = 0.5$, $0.5 \times 2 = 1.0 \Rightarrow 0.101_2$.

Some fractions never terminate

0.1_{10} in binary is $0.0001100110011\dots_2$ — it repeats forever, exactly as $\frac{1}{3}$ does in decimal. A fraction terminates in binary only if its denominator (in lowest terms) is a power of two.

This is the root cause of round-off error (Section 5.7) and of the famous $0.1 + 0.2 \neq 0.3$ behaviour in almost every programming language.

5.2 Normalised scientific notation

Normalised scientific notation

A number is written in the form

$$m \times b^e$$

where m is the **significand** (or **mantissa**), b is the **base** and e is the **exponent**. Normalisation requires

$$1 \leq |m| < b$$

i.e. the significand's absolute value is at least 1 and less than the base.

In binary ($b = 2$) normalisation therefore forces the significand into the form $1.xxxx$ — exactly one 1 before the point.

Binary number	Normalised	Shift
1010110101_2	1.010110101×2^9	point moved 9 left
-1001010.1011_2	-1.0010101011×2^6	point moved 6 left
0.00010101011_2	1.0101011×2^{-4}	point moved 4 right

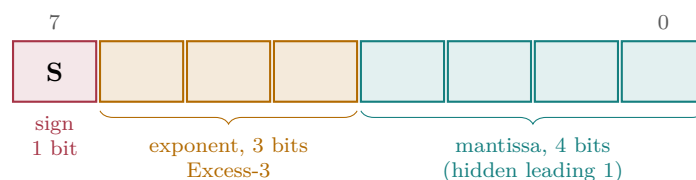
The hidden bit

Since normalisation guarantees the leading digit is always 1, storing it would waste a bit. Most formats — including IEEE 754 — omit it and store only the fractional part after the point. It is called the **hidden** or **implicit** bit and is reinstated automatically when the number is read back.

The course's 8-bit teaching format follows this convention: a stored mantissa of 1001 means the significand 1.1001.

5.3 The 8-bit teaching format

Lecture 3.2.1 uses a small format that makes hand calculation feasible:



Field	Meaning
Sign (1 bit)	0 = positive, 1 = negative
Exponent (3 bits)	the exponent e , encoded in Excess-3
Mantissa (4 bits)	the bits after the point of the normalised significand

5.4 Excess- K (offset binary) encoding

The exponent may be positive or negative, so it needs a signed encoding. Floating point formats do not use two's complement for this; they use **Excess- K** , also called **offset binary** or **biased** representation.

Excess- K

A value v is stored as the unsigned binary number $v + K$. Conversely, a stored pattern r represents the value $r - K$.

$$\text{Representation} = \text{Value} + K$$

$$\text{Value} = \text{Representation} - K$$

Choosing K

There is no universal standard, but for an n -bit field K is almost always

$$K = 2^{n-1} - 1 \quad \text{or} \quad K = 2^{n-1}$$

IEEE 754 single precision uses $n = 8$ and $K = 2^7 - 1 = 127$. The course's 8-bit format uses $n = 3$ and $K = 3 = 2^2 - 1$.

5.4.1 Ranges

The minimum value in Excess- K is always $-K$ (from the all-zeros pattern). The maximum depends on the field width:

$$\text{max} = (2^n - 1) - K$$

Field width	K	Minimum	Maximum
3 bits	3	-3	+4
4 bits	7	-7	+8
4 bits	8	-8	+7
8 bits	127	-127	+128

5.4.2 Excess- K compared with two's complement

4-bit patterns				(continued)			
Binary	Unsigned	Excess-7	Excess-8	Binary	Unsigned	Excess-7	Excess-8
0000	0	-7	-8	1000	8	1	0
0001	1	-6	-7	1001	9	2	1
0010	2	-5	-6	1010	10	3	2
0011	3	-4	-5	1011	11	4	3
0100	4	-3	-4	1100	12	5	4
0101	5	-2	-3	1101	13	6	5
0110	6	-1	-2	1110	14	7	6
0111	7	0	-1	1111	15	8	7

Why biased and not two's complement?

Notice the ordering in the table: as the bit pattern increases as an unsigned number, the represented value increases monotonically, from most negative to most positive. Two's

complement does not have this property — 1000_2 is the *most negative* value but the largest unsigned pattern in the top half.

Consequence: two floating point numbers of the same sign can be compared by comparing their bit patterns as plain unsigned integers. Comparison hardware becomes trivial. That is the entire reason for the bias.

5.5 Encoding: decimal to floating point

Procedure

1. Convert the decimal number to binary (integer part and fraction part separately).
2. Normalise: write as $\pm 1.xxxx \times 2^e$.
3. Sign bit: 0 if positive, 1 if negative.
4. Mantissa: the bits after the point, truncated or padded to the field width. *Do not store the leading 1.*
5. Exponent: compute $e + K$ and write it as unsigned binary.

Worked Example 2 Encode 3.125_{10} in the 8-bit format

Step 1 — to binary. $3 = 11_2$; $0.125 = 0.001_2$, so $3.125_{10} = 11.001_2$.

Step 2 — normalise.

$$11.001_2 = 1.1001_2 \times 2^1$$

Step 3 — sign. Positive $\Rightarrow$ 0

Step 4 — mantissa. Bits after the point: 1001 (exactly 4 bits, no padding needed).

Step 5 — exponent. $e = 1$, $K = 3$, so the stored value is $1 + 3 = 4 = 100_2$.

0	1	0	0	1	0	0	1
---	---	---	---	---	---	---	---

 $= 01001001_2$

Worked Example 3 Encode -7.25_{10} in the 8-bit format

Step 1. $7 = 111_2$ and $0.25 = 0.01_2$, so $-7.25 = -111.01_2$.

Step 2. $-111.01_2 = -1.1101_2 \times 2^2$.

Step 3. Negative $\Rightarrow$ sign = 1.

Step 4. Mantissa = 1101.

Step 5. $e = 2$, so stored exponent = $2 + 3 = 5 = 101_2$.

1	1	0	1	1	1	0	1
---	---	---	---	---	---	---	---

 $= 11011101_2$

5.6 Decoding: floating point to decimal

Procedure — run the encoding backwards

1. Read the sign bit.
2. Read the exponent field as unsigned, then subtract K .
3. Prefix the mantissa with the hidden “1.”
4. Evaluate $\pm 1.\text{mantissa} \times 2^e$.

Worked Example 4 *Decode 10101011₂*

Split the fields: 1 | 010 | 1011

Sign = 1 $\Rightarrow$ the number is negative.

Exponent field 010₂ = 2; actual exponent = 2 - 3 = -1.

Significand = 1.1011₂ (hidden bit restored).

So the value is

$$-1.1011_2 \times 2^{-1} = -0.11011_2$$

Converting the binary fraction:

$$-\left(\frac{1}{2} + \frac{1}{4} + 0 + \frac{1}{16} + \frac{1}{32}\right) = -(0.5 + 0.25 + 0.0625 + 0.03125) = -0.84375_{10}$$

5.7 Round-off error

Round-off error

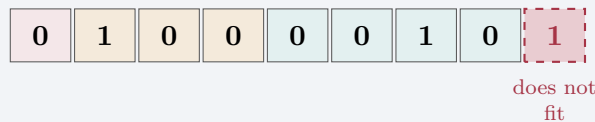
The difference between the exact value of a number and the value actually stored, caused by the mantissa having a fixed, finite number of bits.

Worked Example 5 *Round-off in 2.3125₁₀*

Step 1. 2.3125₁₀ = 10.0101₂ (check: $2 + \frac{1}{4} + \frac{1}{16} = 2.3125$ ✓)

Step 2. Normalise: 1.00101₂ $\times 2^1$.

Step 3. The mantissa needs **five** bits (00101) but the format provides only **four**.



Step 4. The last bit is discarded. What is stored is 01000010₂, i.e. 1.0010₂ $\times 2^1$.

Step 5. Reading that back:

$$1.0010_2 \times 2^1 = 10.010_2 = 2.25_{10}$$

Step 6. The round-off error is therefore

$$2.3125 - 2.25 = 0.0625_{10}$$

The consequence

There are numbers that simply **cannot be represented exactly** in a floating point format. Two practical rules follow:

- Never test floating point values for exact equality. Compare against a tolerance: $\text{abs}(a-b) < \text{epsilon}$.
- Errors accumulate. Summing a million values each carrying a small error can produce a visibly wrong total.

5.8 Range of the 8-bit format

To find the extremes, push each field to its limit and reason about what that does to the value.

Quantity	Sign	Exp	Mantissa	Value
Largest positive	0	111	1111	$+1.1111_2 \times 2^4 = +31$
Smallest positive	0	000	0000	$+1.0000_2 \times 2^{-3} = +0.125$
Largest negative	1	000	0000	$-1.0000_2 \times 2^{-3} = -0.125$
Smallest negative	1	111	1111	$-1.1111_2 \times 2^4 = -31$

“Largest negative” means closest to zero

The wording trips people up. Ordering the four values on the number line:

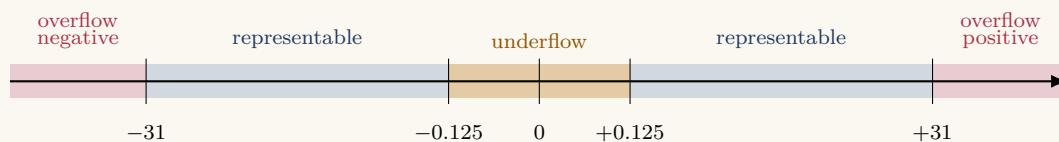
$$\underbrace{-31}_{\text{smallest}} < \underbrace{-0.125}_{\text{largest neg.}} < 0 < \underbrace{+0.125}_{\text{smallest pos.}} < \underbrace{+31}_{\text{largest}}$$

“Largest negative” is -0.125 because $-0.125 > -31$. To get it, make the *magnitude* as small as possible: minimum exponent, minimum mantissa.

5.9 Overflow and underflow

Left as “search and find” in Lecture 3.2.1

The representable values form four bands with gaps at the extremes and around zero:

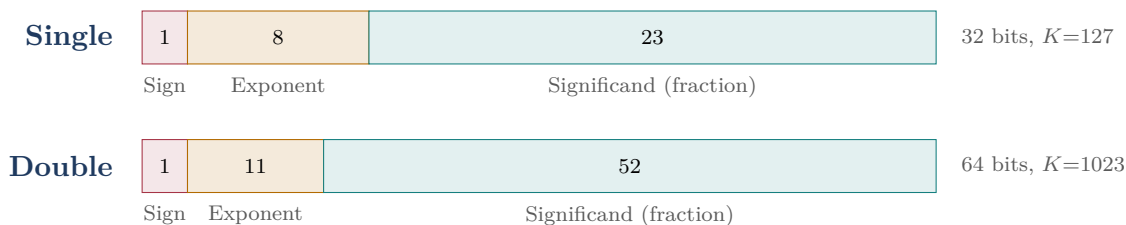


Condition	Meaning
Positive overflow	Result is positive and larger than the largest representable positive value ($> +31$ here).
Negative overflow	Result is negative and smaller than the most negative representable value (< -31).
Positive underflow	Result is positive but too close to zero to represent ($0 < x < +0.125$).
Negative underflow	Result is negative but too close to zero to represent ($-0.125 < x < 0$).

Overflow is the serious failure: the answer is unrepresentably large and the result is typically $\pm\infty$ or a wraparound. **Underflow** is usually benign: the result is flushed to zero, losing precision but staying in the right neighbourhood.

5.10 The IEEE 754 standard

The 8-bit format above is a teaching device. Real hardware follows **IEEE 754**, published in 1985 by the Institute of Electrical and Electronics Engineers, which is why floating point results agree across different machines.



5.10.1 Exponent encoding and reserved patterns

For single precision the 8-bit exponent field could in principle hold 0 to 255, i.e. values -127 to $+128$ after removing the bias of 127. But the two extreme patterns are **reserved**:

Exponent field	Unbiased	Meaning
00000000	-127	reserved: zero and subnormal numbers
00000001–11111110	$-126 \dots +127$	ordinary (normal) numbers
11111111	$+128$	reserved: infinity and NaN

So the usable exponent range is

$$-126 \leq e \leq +127$$

5.10.2 Reserved bit patterns

Value	Sign	Exponent & significand
Positive infinity	0	11111111 000000000000000000000000
Negative infinity	1	11111111 000000000000000000000000
Not a Number (NaN)	0	11111111 1xxxxxxxxxxxxxxxxxxxxxxxxx
Positive zero	0	00000000 000000000000000000000000
Negative zero	1	00000000 000000000000000000000000

What the special values are for

- **Infinity** is produced by overflow and by division of a non-zero number by zero. Arithmetic on it behaves sensibly: $1/\infty = 0$, $\infty + 1 = \infty$.
- **NaN** is produced by genuinely undefined operations: $0/0$, $\infty - \infty$, $\sqrt{-1}$. NaN propagates through every subsequent operation, and famously $\text{NaN} \neq \text{NaN}$.
- **Two zeros** are retained here (unlike in two's complement) so that the sign of a value that underflowed is not lost. $+0$ and -0 compare as equal.
- **Subnormal (denormal) numbers**, signalled by a zero exponent field with a non-zero significand, drop the hidden bit and fill the gap between zero and the smallest normal number, giving gradual rather than abrupt underflow.

5.10.3 IEEE 754 single precision worked example

Worked Example 6 *Encode 175.625_{10} in IEEE 754 single precision*

Step 1 — convert to binary.

$$175 = 10101111_2, \quad 0.625 = 0.101_2$$

$$175.625_{10} = 10101111.101_2$$

Step 2 — normalise. The point moves 7 places left:

$$10101111.101_2 = 1.010111101_2 \times 2^7$$

Step 3a — sign bit. Positive $\Rightarrow$ 0.

Step 3b — biased exponent. With $K = 127$:

$$127 + 7 = 134 = 10000110_2$$

Step 3c — significand. Take the bits after the point, 010111101, and pad on the *right* with zeros to 23 bits.

0	10000110	01011110100000000000000
sign	exponent	significand

Worked Example 7 *Decode an IEEE 754 single precision number*

Given 0 | 01100111 | 001010001100010000000000

Sign = 0 $\Rightarrow$ positive.

Exponent. $01100111_2 = 64 + 32 + 4 + 2 + 1 = 103_{10}$, so

$$e = 103 - 127 = -24$$

Significand. Restore the hidden bit and evaluate the set positions (3, 5, 9, 10, 14):

$$1 + 2^{-3} + 2^{-5} + 2^{-9} + 2^{-10} + 2^{-14} = 1.15924072265625$$

Value.

$$1.15924072265625 \times 2^{-24} = 6.9096131 \times 10^{-8}$$

Small discrepancy with the tutorial

Tutorial 02 gives the answer as $6.909613013 \times 10^{-8}$. That comes from truncating the significand to 1.1592407 before multiplying. Carrying the exact significand 1.15924072265625 gives $6.9096131483 \times 10^{-8}$. The two agree to seven significant figures; the difference is itself a small round-off error, which is rather fitting.

5.10.4 IEEE 754 single precision range

Left as homework in Lecture 3.2.1

Quantity	Exact form	Approx.
Largest positive	$(2 - 2^{-23}) \times 2^{127}$	3.4028235×10^{38}
Smallest positive (normal)	1.0×2^{-126}	$1.1754944 \times 10^{-38}$
Smallest positive (subnormal)	2^{-149}	$1.4012985 \times 10^{-45}$
Largest negative (normal)	-1.0×2^{-126}	$-1.1754944 \times 10^{-38}$
Smallest negative	$-(2 - 2^{-23}) \times 2^{127}$	$-3.4028235 \times 10^{38}$

The largest exponent field usable for ordinary numbers is 11111110 = 254, giving $e = 254 - 127 = 127$; the largest significand is 1.111...1 with 23 ones, which equals $2 - 2^{-23}$. Multiplying gives the first row. The smallest normal uses exponent field 00000001 ($e = -126$) with significand exactly 1.0.

5.11 Summary

Concept	Key point
Binary fraction	Digits right of the point carry weights $2^{-1}, 2^{-2}, \dots$
Decimal $\rightarrow$ binary fraction	Repeatedly multiply by 2; read integer parts downwards.
Normalisation	Force the form $1.xxxx \times 2^e$; the leading 1 is hidden.
Excess- K	Stored = value + K ; makes bit patterns compare in value order.
Round-off error	Caused by a finite mantissa; never test floats for equality.
Overflow	Magnitude too large — gives $\pm\infty$.
Underflow	Magnitude too small — flushed towards zero.
IEEE 754 single	1 + 8 + 23 bits, $K = 127$, usable $e \in [-126, 127]$.
IEEE 754 double	1 + 11 + 52 bits, $K = 1023$.

PART II

Boolean Algebra and Logic Minimisation

Chapter 6

Boolean Algebra and Logic Gates

Where this fits

Boolean algebra is the mathematics of two-state systems, and logic gates are its physical realisation. This chapter establishes the operators, their truth tables and gate symbols, and the mechanical translation between an algebraic expression and a circuit diagram. Sources: Lecture 2.1.1 and Tutorial 03, merged.

6.1 Origins and the two states

Boolean algebra was introduced by **George Boole**, a British mathematician, in 1854 — the branch of algebra in which the variables take only the truth values *true* and *false*. Claude Shannon's 1937 master's thesis showed that it describes switching circuits exactly, which is why an abstract 19th-century algebra sits at the foundation of digital hardware.

The two states

A Boolean variable takes exactly one of two states:

State 0 (false, low) State 1 (true, high)

By convention a variable written plainly, x , is being considered in state 1; a variable written with a negation, $\bar{x}$, denotes state 0.

6.2 The basic operations

Three operations are *primitive* — everything else is built from them.

$A \cdot B$
AND

$A + B$
OR

$\bar{A}$
NOT

Boolean function

A **Boolean function** is an expression built from binary variables and Boolean operations, e.g.

$$f(x, y) = x \cdot \bar{y} + \bar{x} \cdot (x + y)$$

Because each variable has only two possible values, a function of n variables has a **finite** number — exactly 2^n — of input combinations, and can therefore be described exhaustively by a table.

6.3 Truth tables

Truth table

A table listing the output of a Boolean function for every possible combination of its input variables. A function of n variables has 2^n rows.

Worked Example 1 Building a truth table

$F = x + y \cdot \bar{z}$ has three variables, so $2^3 = 8$ rows. It helps to add an intermediate column for each sub-term.

x	y	z	$y \cdot \bar{z}$	F
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	0	0
1	0	0	0	1
1	0	1	0	1
1	1	0	1	1
1	1	1	0	1

Note the input pattern: list the combinations in **binary counting order** (000, 001, 010, ...). Doing this consistently means row i always corresponds to minterm i , which matters in Chapter 7.

Method for hand evaluation

1. Count the variables $\rightarrow$ that fixes the number of rows, 2^n .
2. Fill the input columns in binary counting order.
3. Add one column per sub-expression, working from the innermost outward.
4. Combine the sub-columns to get the final output.

Intermediate columns cost nothing and eliminate most arithmetic slips.

6.4 From algebra to hardware

Logic gate

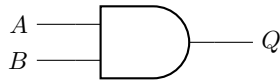
A **logic gate** is a piece of hardware that emulates a Boolean algebraic operation and operates on Boolean inputs. Internally it is built from transistors, resistors and diodes.

Gates are manufactured as **integrated circuits** (ICs). A single IC package usually contains several gates of *one* type — e.g. four 2-input NAND gates in a 14-pin package. This packaging economics is precisely why single-gate-type design (Chapter 10) matters commercially.

6.5 The gate catalogue

6.5.1 AND

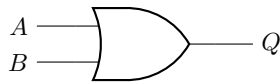
Output is 1 only when *all* inputs are 1. Written $A \cdot B$ or $A \wedge B$.



A	B	$Q = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

6.5.2 OR

Output is 1 when *at least one* input is 1. Written $A + B$ or $A \vee B$.



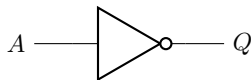
A	B	$Q = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

OR is inclusive

$1 + 1 = 1$, not 2 and not 0. Boolean OR is *inclusive* — “at least one”. The exclusive version is XOR, below. This is the single most common source of confusion for people coming from ordinary arithmetic.

6.5.3 NOT (inverter)

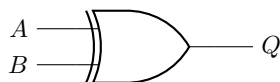
Output is the complement of the input. Written $\bar{A}$, A' , $\neg A$ or $\sim A$.



A	$Q = \bar{A}$
0	1
1	0

6.5.4 XOR (exclusive OR)

Output is 1 when the inputs *differ*. Written $A \oplus B$.



A	B	$Q = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

XOR is the “difference detector”

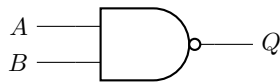
Three readings of XOR, all useful:

- *inequality*: $A \oplus B = 1 \iff A \neq B$;
- *controlled inverter*: $A \oplus 1 = \bar{A}$ and $A \oplus 0 = A$;
- *addition mod 2*: it is the sum bit of a half adder.

The second reading is how a single adder circuit is made to perform subtraction (Section 12.6).

6.5.5 NAND (NOT-AND)

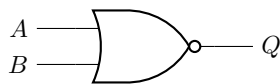
The complement of AND: $\overline{A \cdot B}$.



A	B	$Q = \overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

6.5.6 NOR (NOT-OR)

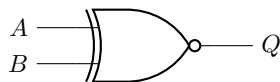
The complement of OR: $\overline{A + B}$.



A	B	$Q = \overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

6.5.7 XNOR (exclusive NOR)

The complement of XOR: 1 when the inputs are *equal*.

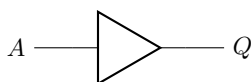


A	B	$Q = \overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

XNOR is the **equality comparator** — the building block of any circuit that compares two multi-bit values.

6.5.8 Buffer

Passes its input through unchanged.



A	$Q = A$
0	0
1	1

Why build a gate that does nothing?

Logically a buffer is the identity, but electrically it is useful. It **restores signal strength** so one output can drive many inputs, and it **introduces a deliberate propagation delay** used to align signals that would otherwise arrive at different times. Chapter 14 shows why timing alignment matters.

6.5.9 Complete reference

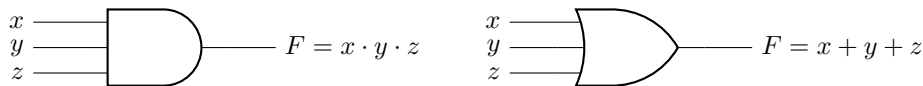
Gate	Expression	00	01	10	11
AND	$A \cdot B$	0	0	0	1
OR	$A + B$	0	1	1	1
XOR	$A \oplus B$	0	1	1	0
NAND	$\overline{A \cdot B}$	1	1	1	0
NOR	$\overline{A + B}$	1	0	0	0
XNOR	$\overline{A \oplus B}$	1	0	0	1

Memorising the table

Learn AND, OR, XOR — then NAND, NOR, XNOR are just those three rows inverted. Note also that NAND and NOR are each other's De Morgan duals: $\overline{A \cdot B} = \overline{A} + \overline{B}$ and $\overline{A + B} = \overline{A} \cdot \overline{B}$.

6.6 Multi-input gates

Gates with more than two inputs are commonly available.



AND, OR, XOR, NAND and NOR all extend to n inputs. Their meanings generalise naturally:

- n -input AND: 1 only if *every* input is 1.
- n -input OR: 1 if *any* input is 1.
- n -input XOR: 1 if an *odd number* of inputs are 1 (a parity generator).

Multi-input XOR is not “exactly one”

A 3-input XOR outputs 1 for inputs 001, 010, 100 and also 111 — odd parity, not “exactly one high”. This is exactly why a 3-input XOR computes the sum bit of a full adder (Section 12.2).

6.7 Operator precedence

Order	Symbol	Operation
1 (highest)	()	Parentheses evaluated first
2	$\neg, \sim, \neg$	NOT
3	$\wedge, \cdot$	AND
4 (lowest)	$\vee, +$	OR evaluated last

The analogy that makes this stick

AND behaves like multiplication and OR like addition. Just as $a + b \times c$ means $a + (b \times c)$, so

$$x + y \cdot z \text{ means } x + (y \cdot z), \text{ not } (x + y) \cdot z$$

This is also why AND terms are called *products* and OR terms *sums*.

Worked Example 2 Precedence in practice

Evaluate $F = \bar{A} + B \cdot C$ for $A = 1, B = 1, C = 1$.

1. NOT first: $\bar{A} = \bar{1} = 0$.
2. AND next: $B \cdot C = 1 \cdot 1 = 1$.
3. OR last: $0 + 1 = 1$.

$F = 1$.

Had the expression been $(\bar{A} + B) \cdot C$ the parentheses would force $(0 + 1) \cdot 1 = 1$ — same answer here, but for $A = 1, B = 0, C = 1$ the two differ: $\bar{A} + B \cdot C = 0 + 0 = 0$ whereas $(\bar{A} + B) \cdot C = 0 \cdot 1 = 0$. Try $A = 0, B = 0, C = 0$: first gives 1, second gives 0.

6.8 Drawing a circuit from an expression

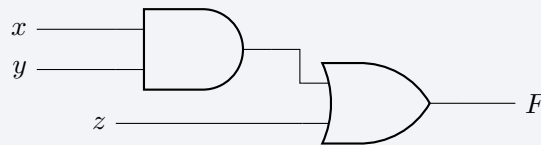
Procedure

1. Identify the **last** operation to be performed under the precedence rules — that gate is the output gate, drawn rightmost.
2. Work backwards: each operand of that gate is either an input variable or the output of another gate.
3. Continue until only input variables remain on the left.

Building right-to-left is far more reliable than trying to build left-to-right.

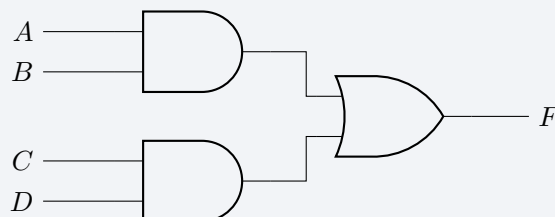
Worked Example 3 Draw $F = x \cdot y + z$

Precedence says AND binds tighter, so the last operation is the OR. The output gate is therefore an OR gate, fed by $(x \text{ AND } y)$ and by z directly.

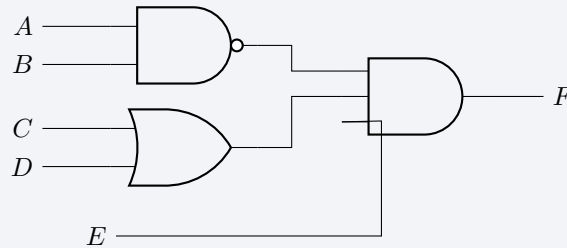


Worked Example 4 Draw $AB + CD$ — Tutorial 03

Two AND gates feeding one OR gate.



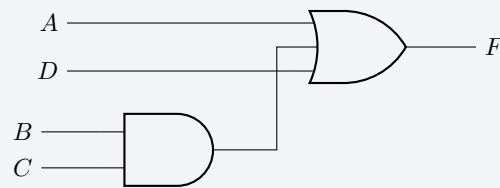
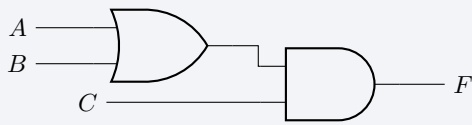
Worked Example 5 Draw $\overline{A \cdot B} \cdot (C + D) \cdot E$ — Tutorial 03



Worked Example 6 Draw the Tutorial 03 exercise set

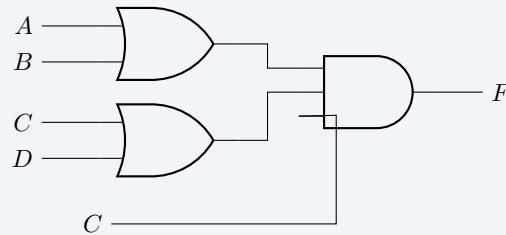
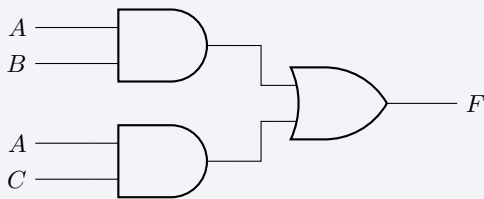
(a) $(A + B)C$

(b) $A + BC + D$



(c) $AB + AC$

(d) $(A + B)(C + D)C$



Many circuits, one function

The same expression can be drawn in several equivalent ways, and different expressions can produce identical truth tables. $F = x \cdot y + z$ can be built with an AND and an OR, or entirely from NAND gates (Chapter 10), or from a different but equivalent algebraic form.

The simplest structure is always preferred, for two reasons:

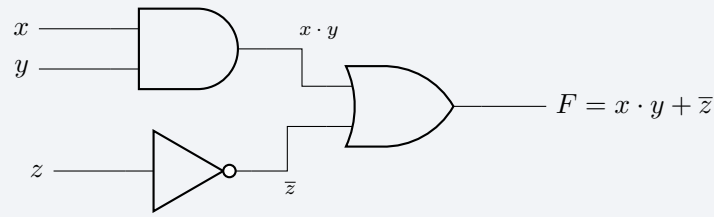
- **cost** — fewer gates means fewer transistors and less silicon;
- **efficiency** — fewer gates in series means shorter propagation delay and a faster circuit.

Chapters 8 and 9 are entirely devoted to finding that simplest form.

6.9 Reading a circuit back into an expression

The reverse translation is equally mechanical: label the output of every gate, working left to right, until you reach the final output.

Worked Example 7 Circuit to expression



Label the AND output $x \cdot y$ and the inverter output $\bar{z}$; the OR gate combines them, giving $F = x \cdot y + \bar{z}$.

6.10 Summary

Concept	Key point
Boolean variable	Two states only, 0 and 1.
Primitive operations	AND, OR, NOT — everything else derives from these.
Truth table	2^n rows for n variables; list inputs in counting order.
Logic gate	Hardware realisation of a Boolean operation.
XOR	Difference detector; also the half-adder sum.
XNOR	Equality detector; basis of comparators.
Precedence	$() > \text{NOT} > \text{AND} > \text{OR}$ — mirrors arithmetic.
Expression $\rightarrow$ circuit	Find the last operation; draw it rightmost; work backwards.
Simplicity	Fewer gates = lower cost and shorter delay.

Chapter 7

Canonical and Standard Forms

Where this fits

A truth table describes behaviour; an expression describes structure. This chapter gives the mechanical translation in both directions, and introduces the two *canonical* forms — sum of minterms and product of maxterms — that guarantee a unique starting point for simplification. Source: Lecture 2.1.2–2.1.5.

7.1 One behaviour, many expressions

A single truth table can be written as many different Boolean expressions, and each expression yields a different circuit.

$$F_1 = \bar{x}\bar{y}z + \bar{x}yz + x\bar{y}\bar{z} + x\bar{y}z$$

$$F_2 = x\bar{y} + \bar{x}z$$

x	y	z	F_1	F_2
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	0	0
1	1	1	0	0

The two columns are identical — the expressions are logically equivalent. But F_1 needs four 3-input AND gates plus a 4-input OR gate (and inverters); F_2 needs two 2-input AND gates and one OR gate.

Why we care

Simple and minimal circuit structures are always preferred, for **efficiency** (shorter propagation delay) and **cost** (fewer gates, less silicon, fewer ICs). Finding the minimal form is the whole business of Chapters 8 and 9. This chapter provides the canonical starting point those techniques operate on.

7.2 Expression to truth table

Procedure

1. Identify the variables. If there are n , the table has 2^n rows.
2. Enumerate all 2^n input combinations in binary counting order.
3. Evaluate each term of the expression per row, then combine.

Worked Example 1 *Expression to truth table*

$$F(x, y, z) = x \cdot y \cdot \bar{z} + \bar{y} \cdot z$$

Three variables $\Rightarrow 2^3 = 8$ rows.

x	y	z	$xy\bar{z}$	$\bar{y}z$	F
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	0	0	0

The term $xy\bar{z}$ is 1 only in row 110; the term $\bar{y}z$ is 1 whenever $y = 0$ and $z = 1$, i.e. rows 001 and 101.

7.3 Minterms and maxterms

7.3.1 Minterm

Minterm (standard product)

For n variables, a **minterm** is an AND of *all* n variables, each appearing exactly once, either plain or complemented. There are 2^n minterms.

For three variables x, y, z the eight minterms are:

x	y	z	Minterm	Symbol
0	0	0	$\bar{x} \cdot \bar{y} \cdot \bar{z}$	m_0
0	0	1	$\bar{x} \cdot \bar{y} \cdot z$	m_1
0	1	0	$\bar{x} \cdot y \cdot \bar{z}$	m_2
0	1	1	$\bar{x} \cdot y \cdot z$	m_3
1	0	0	$x \cdot \bar{y} \cdot \bar{z}$	m_4
1	0	1	$x \cdot \bar{y} \cdot z$	m_5
1	1	0	$x \cdot y \cdot \bar{z}$	m_6
1	1	1	$x \cdot y \cdot z$	m_7

The minterm rule

For a given row, a variable appears **plain** if its value is 1 and **complemented** if its value is 0.

Each minterm is 1 for *exactly one* row of the truth table — the row it was built from — and 0 everywhere else. That one-to-one correspondence is what makes the sum-of-minterms construction work.

Watch out

$x \cdot y$ is **not** a minterm in a three-variable context: z is missing. A minterm must contain every variable. Terms that omit variables belong to the *standard* forms (Section 7.7), not the canonical ones.

7.3.2 Maxterm**Maxterm (standard sum)**

For n variables, a **maxterm** is an OR of *all* n variables, each appearing exactly once, either plain or complemented. There are 2^n maxterms.

x	y	z	Maxterm	Symbol
0	0	0	$x + y + z$	M_0
0	0	1	$x + y + \bar{z}$	M_1
0	1	0	$x + \bar{y} + z$	M_2
0	1	1	$x + \bar{y} + \bar{z}$	M_3
1	0	0	$\bar{x} + y + z$	M_4
1	0	1	$\bar{x} + y + \bar{z}$	M_5
1	1	0	$\bar{x} + \bar{y} + z$	M_6
1	1	1	$\bar{x} + \bar{y} + \bar{z}$	M_7

The maxterm rule — note it is the *opposite* of minterms

For a given row, a variable appears **plain** if its value is 0 and **complemented** if its value is 1.

Each maxterm is 0 for exactly one row and 1 everywhere else. Formally, $M_i = \bar{m}_i$.

The inversion catches everyone

Row 011 gives minterm $\bar{x} \cdot y \cdot z$ but maxterm $x + \bar{y} + \bar{z}$. If you write the maxterm with the same polarities as the minterm, every subsequent step will be wrong. Say it out loud: *minterm — 1 means plain; maxterm — 0 means plain.*

7.4 Truth table to canonical form**7.4.1 Sum of minterms****Reasoning**

Consider one row where $F = 1$, say $x = 0, y = 1, z = 0$. The function is 1 when x is 0 **and** y is 1 **and** z is 0 — a conjunction — so that row is captured by $\bar{x} \cdot y \cdot \bar{z}$.

The function is 1 if the inputs match row 2 **or** row 4 **or** row 5 ... — a disjunction. So OR the minterms of all rows where $F = 1$.

Worked Example 2 *Truth table to sum of minterms*

x	y	z	F	
0	0	0	0	
0	0	1	0	
0	1	0	1	$\bar{x}y\bar{z}$
0	1	1	0	
1	0	0	1	$x\bar{y}\bar{z}$
1	0	1	1	$x\bar{y}z$
1	1	0	1	$xy\bar{z}$
1	1	1	1	xyz

Collect the marked rows and OR them:

$$\begin{aligned}
 F &= \bar{x} \cdot y \cdot \bar{z} + x \cdot \bar{y} \cdot \bar{z} \\
 &\quad + x \cdot \bar{y} \cdot z + x \cdot y \cdot \bar{z} \\
 &\quad + x \cdot y \cdot z
 \end{aligned}$$

In compact notation,

$$F(x, y, z) = \sum m(2, 4, 5, 6, 7)$$

7.4.2 Product of maxterms

There are two routes to the same answer. The first is a shortcut; the second shows why the shortcut is valid.

Shortcut

Take every row where $F = 0$, write its *maxterm*, and AND them together.

Worked Example 3 Truth table to product of maxterms

Using the same table, $F = 0$ in rows 000, 001 and 011.

x	y	z	Maxterm
0	0	0	$x + y + z$
0	0	1	$x + y + \bar{z}$
0	1	1	$x + \bar{y} + \bar{z}$

$$F = (x + y + z) \cdot (x + y + \bar{z}) \cdot (x + \bar{y} + \bar{z}) = \prod M(0, 1, 3)$$

7.4.3 Deriving the product of maxterms properly

The derivation, step by step

Step 1. Write $\overline{F}$ as a sum of minterms — take the rows where $F = 0$:

$$\overline{F} = \overline{x}\overline{y}\overline{z} + \overline{x}\overline{y}z + \overline{x}yz$$

Step 2. Complement both sides:

$$F = \overline{\overline{F}} = \overline{\overline{x}\overline{y}\overline{z} + \overline{x}\overline{y}z + \overline{x}yz}$$

Step 3. Apply De Morgan's theorem to turn the complement of a sum into a product of complements:

$$F = \overline{(\overline{x}\overline{y}\overline{z})} \cdot \overline{(\overline{x}\overline{y}z)} \cdot \overline{(\overline{x}yz)}$$

Step 4. Apply De Morgan again inside each bracket — the complement of a product is the sum of complements:

$$F = (x + y + z) \cdot (x + y + \overline{z}) \cdot (x + \overline{y} + \overline{z})$$

which is exactly the shortcut's answer.

De Morgan's theorem

$$\overline{A \cdot B} = \overline{A} + \overline{B} \qquad \overline{A + B} = \overline{A} \cdot \overline{B}$$

Both hold for any number of variables. In words: *break the bar, change the sign.*

Intuition for maxterms

Take one term of the product, say $(x + \overline{y} + \overline{z})$. An AND of terms is 0 if *any* term is 0. This term is 0 only when $x = 0$, $\overline{y} = 0$ and $\overline{z} = 0$ simultaneously — i.e. $x = 0, y = 1, z = 1$, which is exactly row 011.

So each maxterm “knocks out” precisely one row. Multiply together the maxterms for all the rows that must be 0, and every other row is left at 1.

7.5 Canonical form to truth table

7.5.1 Sum of minterms to truth table

Procedure

1. Each minterm identifies exactly one row — find it and write 1.
2. Fill every remaining row with 0.

Worked Example 4 *SoM to truth table*

$F = \overline{x}\overline{y} + x\overline{y} + xy$ (two variables)

x	y	F	
0	0	1	from $\overline{x}\overline{y}$
0	1	0	no minterm
1	0	1	from $x\overline{y}$
1	1	1	from xy

7.5.2 Product of maxterms to truth table

Procedure

1. **Negate every literal** in a maxterm — this converts it into the corresponding minterm (De Morgan).
2. That minterm identifies the row; write **0** there.
3. Fill every remaining row with 1.

The intuition: if a single maxterm evaluates to zero, the whole product is zero.

Worked Example 5 *PoM to truth table*

$$F = (x + y + z) \cdot (x + y + \bar{z}) \cdot (x + \bar{y} + \bar{z})$$

Negating the literals in each factor:

$$(x + y + z) \rightarrow \bar{x} \bar{y} \bar{z} = \text{row } 000$$

$$(x + y + \bar{z}) \rightarrow \bar{x} \bar{y} z = \text{row } 001$$

$$(x + \bar{y} + \bar{z}) \rightarrow \bar{x} y z = \text{row } 011$$

x	y	z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

7.6 Sigma and Pi notation

Compact canonical notation

$$F(x, y, z) = \sum m(i_1, i_2, \dots) \quad F(x, y, z) = \prod M(j_1, j_2, \dots)$$

$\sum$ lists the row indices where $F = 1$; $\prod$ lists the row indices where $F = 0$. **Row indices start from 0**, and the variable order in the argument list fixes the bit significance.

The two index sets are complementary

For an n -variable function, every row index from 0 to $2^n - 1$ appears in exactly one of the two lists. So

$$F = \sum m(2, 4, 5, 6, 7) \iff F = \prod M(0, 1, 3)$$

Converting between the canonical forms requires no algebra at all — just take the complement of the index set.

7.7 Standard forms

Standard form

In a **canonical** form every term must contain *all* the variables. In a **standard** form a term need not contain all the variables.

There are two standard forms, exactly paralleling the canonical ones.

7.7.1 Sum of Products (SoP)

An OR over several terms, each an AND of several literals.

$$F = \bar{x} \cdot y + \bar{y} \cdot \bar{z} + x \cdot \bar{y} \cdot z + y$$

Note that the terms have different sizes — $\bar{x}y$ has two literals, $x\bar{y}z$ has three, and y has one. All are legal in a standard form.

7.7.2 Product of Sums (PoS)

An AND over several terms, each an OR of several literals.

$$F = y \cdot (x + z) \cdot (\bar{x} + \bar{y} + \bar{z})$$

Form	Structure	All variables in every term?
Sum of Minterms (SoM)	OR of ANDs	yes — canonical
Product of Maxterms (PoM)	AND of ORs	yes — canonical
Sum of Products (SoP)	OR of ANDs	no — standard
Product of Sums (PoS)	AND of ORs	no — standard

The relationship

Canonical forms are *unique*: a given truth table has exactly one SoM and exactly one PoM. Standard forms are not unique, and they are what you get *after* simplification. The workflow is:

$$\text{truth table} \rightarrow \underbrace{\text{SoM or PoM}}_{\text{canonical, unique}} \rightarrow \underbrace{\text{SoP or PoS}}_{\text{standard, minimal}}$$

7.8 Choosing between SoM and PoM

Rule of thumb

Count the output column.

- More 0s than 1s $\Rightarrow$ use **sum of minterms** (fewer terms to write).
- More 1s than 0s $\Rightarrow$ use **product of maxterms**.

The number of terms equals the number of rows you have to account for, so pick whichever value is rarer.

7.9 Worked exercise

Worked Example 6 Both canonical forms from one table — Lecture 2.1.5

Derive the expression in sum-of-minterms and product-of-maxterms form.

Sum of minterms. Six rows give 1:

Row	x	y	z	F
0	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

$$F = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + \bar{x}y\bar{z} \\ + x\bar{y}\bar{z} + xy\bar{z} + xyz$$

$$F(x, y, z) = \sum m(0, 1, 2, 4, 6, 7)$$

Product of maxterms. Only two rows give 0 — rows 3 and 5:

$$\text{row } 011 \rightarrow (x + \bar{y} + \bar{z})$$

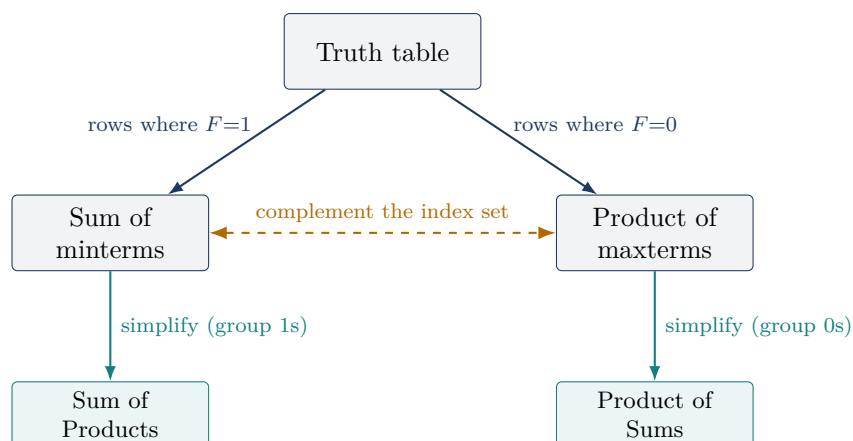
$$\text{row } 101 \rightarrow (\bar{x} + y + \bar{z})$$

$$F = (x + \bar{y} + \bar{z}) \cdot (\bar{x} + y + \bar{z})$$

$$F(x, y, z) = \prod M(3, 5)$$

Observe: the PoM form needs two terms where SoM needs six — exactly the rule of thumb in action, since this table has six 1s and only two 0s. Also note $\{0, 1, 2, 4, 6, 7\}$ and $\{3, 5\}$ together make up $\{0, \dots, 7\}$.

7.10 Summary



Concept	Key point
Minterm	AND of all n variables; 1 in exactly one row; plain if bit is 1.
Maxterm	OR of all n variables; 0 in exactly one row; plain if bit is 0.
$\sum m$	Indices of rows where $F = 1$.
$\prod M$	Indices of rows where $F = 0$.
Canonical	Every term contains every variable; unique per truth table.
Standard	Terms may omit variables; the result of simplification.
De Morgan	$\overline{A \cdot B} = \bar{A} + \bar{B}$; $\overline{A + B} = \bar{A} \cdot \bar{B}$.

Chapter 8

Algebraic Simplification

Where this fits

Boolean algebra is a genuine mathematical system with its own axioms, laws and theorems. Applied correctly they transform an expression into a logically equivalent but cheaper one. This chapter is the toolkit; Chapter 9 gives a graphical method that is more systematic. Sources: Lecture 2.2.1 and Tutorial 04, merged.

8.1 Postulates (axioms)

Postulate

A statement assumed to be true, without proof, as the basis for mathematical reasoning. Boolean algebra's postulates are simply the operator definitions written out for every combination of constants.

AND

$$0 \cdot 0 = 0$$

$$0 \cdot 1 = 0$$

$$1 \cdot 0 = 0$$

$$1 \cdot 1 = 1$$

OR

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

NOT

$$\overline{0} = 1$$

$$\overline{1} = 0$$

8.2 The laws

Every law below can be verified by exhaustive truth table — there are only 2^n cases to check, so “proof” in Boolean algebra is always finite and mechanical.

8.2.1 Annulment law

$$A \cdot 0 = 0$$

$$A + 1 = 1$$

The result is fixed regardless of A .

A	0	$A \cdot 0$
1	0	0
0	0	0

A	1	$A + 1$
1	1	1
0	1	1

8.2.2 Identity law

$$A \cdot 1 = A$$

$$A + 0 = A$$

1 is the identity for AND, 0 is the identity for OR — exactly as 1 and 0 behave for multiplication and addition.

8.2.3 Idempotent law

$$A \cdot A = A$$

$$A + A = A$$

No exponents, no coefficients

$A \cdot A$ is A , not A^2 . $A + A$ is A , not $2A$. Boolean algebra has neither powers nor multipliers — a fact that makes simplification much easier than ordinary algebra, and which trips up people who over-apply arithmetic intuition.

8.2.4 Complement law

$$A \cdot \bar{A} = 0 \qquad A + \bar{A} = 1$$

A variable and its complement cannot both be 1, and at least one of them must be. These two are the workhorses of simplification: spotting them is how terms collapse.

8.2.5 Commutative law

$$A \cdot B = B \cdot A \qquad A + B = B + A$$

8.2.6 Associative law

$$(A \cdot B) \cdot C = A \cdot (B \cdot C) \qquad (A + B) + C = A + (B + C)$$

Together, commutativity and associativity mean terms can be freely reordered and regrouped — which is what licenses “collect the y terms”.

8.2.7 Double negation law

$$\bar{\bar{A}} = A$$

Two inversions cancel. This is used constantly when converting circuits to NAND-only form (Chapter 10).

8.2.8 Distributive law

$$A \cdot (B + C) = A \cdot B + A \cdot C$$

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

The second form has no arithmetic analogue

$A \cdot (B + C) = AB + AC$ matches ordinary algebra. But $A + BC = (A + B)(A + C)$ does *not* — in arithmetic $a + bc \neq (a + b)(a + c)$. In Boolean algebra it holds, and it is exactly what converts an SoP expression into a PoS one. Verify it once by truth table so you believe it:

A	B	C	BC	$A+BC$	$A+B$	$A+C$	$(A+B)(A+C)$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Columns 5 and 8 agree. ✓

8.2.9 Redundancy law (absorption)

$$A + A \cdot B = A \qquad A \cdot (A + B) = A$$

Why absorption works

$A + AB = A(1 + B) = A \cdot 1 = A$. The term AB can only be 1 when A is already 1, so it never adds anything — it is redundant.

A	B	$A \cdot B$	$A + A \cdot B$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

The last column equals A . ✓

8.2.10 Absorption variants

$$A + \bar{A} \cdot B = A + B \qquad A \cdot (\bar{A} + B) = A \cdot B$$

Not the same as plain absorption — here the complement lets the *other* variable survive. Derivation of the first:

$$A + \bar{A}B \stackrel{\text{distrib.}}{=} (A + \bar{A})(A + B) \stackrel{\text{compl.}}{=} 1 \cdot (A + B) \stackrel{\text{identity}}{=} A + B$$

8.2.11 Combining (adjacency) law

$$A \cdot B + \bar{A} \cdot B = B \qquad (A + B) \cdot (\bar{A} + B) = B$$

The most important law for minimisation

$$AB + \bar{A}B = B(A + \bar{A}) = B \cdot 1 = B.$$

If two terms are identical except that one variable appears plain in one and complemented in the other, that variable is **irrelevant** and can be eliminated. This single law is the entire mathematical content of the Karnaugh map — grouping two adjacent cells is exactly one application of it.

8.2.12 De Morgan's theorem

$$\overline{A \cdot B} = \bar{A} + \bar{B} \qquad \overline{A + B} = \bar{A} \cdot \bar{B}$$

Verified exhaustively:

A	B	$A \cdot B$	$\overline{A \cdot B}$	$\bar{A} + \bar{B}$
0	0	0	1	1
0	1	0	1	1
1	0	0	1	1
1	1	1	0	0

A	B	$A + B$	$\overline{A + B}$	$\bar{A} \cdot \bar{B}$
0	0	0	1	1
0	1	1	0	0
1	0	1	0	0
1	1	1	0	0

How to apply it

Break the bar and change the operator. When a bar spans several terms, split it and swap AND $\leftrightarrow$ OR. The theorem generalises to any number of variables:

$$\overline{A \cdot B \cdot C} = \bar{A} + \bar{B} + \bar{C} \qquad \overline{A + B + C} = \bar{A} \cdot \bar{B} \cdot \bar{C}$$

8.3 Complete law reference

Law	AND form	OR form
Annulment	$A \cdot 0 = 0$	$A + 1 = 1$
Identity	$A \cdot 1 = A$	$A + 0 = A$
Idempotent	$A \cdot A = A$	$A + A = A$
Complement	$A \cdot \bar{A} = 0$	$A + \bar{A} = 1$
Commutative	$AB = BA$	$A + B = B + A$
Associative	$(AB)C = A(BC)$	$(A+B)+C = A+(B+C)$
Distributive	$A(B+C) = AB + AC$	$A + BC = (A+B)(A+C)$
Absorption	$A(A+B) = A$	$A + AB = A$
Absorption var.	$A(\bar{A}+B) = AB$	$A + \bar{A}B = A + B$
Combining	$(A+B)(\bar{A}+B) = B$	$AB + \bar{A}B = B$
Double negation	$\bar{\bar{A}} = A$	
De Morgan	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A+B} = \bar{A} \cdot \bar{B}$

Duality

Notice the table's structure: swapping every AND with OR and every 0 with 1 turns each law into its partner. This is the **principle of duality**, and it means you only ever have to memorise half the table.

8.4 Simplification strategy

A practical order of attack

1. **Push complements inward** with De Morgan until bars sit on single variables only.
2. **Cancel double negations.**
3. **Expand** products over sums (distributive) to reach an SoP form.
4. **Delete impossible terms** — anything containing $A \cdot \bar{A}$ is 0.
5. **Collapse duplicates** with idempotence.
6. **Factor out common variables** and look for $A + \bar{A} = 1$.
7. **Absorb** redundant terms.
8. Repeat until nothing changes.

Algebraic simplification has no stopping rule

Unlike a K-map, algebra gives no guarantee that you have reached the minimum, and no signal that you are finished. Two people can simplify the same expression correctly and stop at different places. For exam work: use algebra when the expression is small or a specific law is clearly indicated, and use a K-map (Chapter 9) when you need a guaranteed minimum.

8.5 Worked examples

Worked Example 1 *Expand and collapse — Lecture 2.2.1*

Simplify $F = (x + \bar{y} + \bar{z}) \cdot (x + \bar{y} \cdot z)$.

Expand (distributive, every term of the first bracket against every term of the second):

$$F = x \cdot x + x \cdot \bar{y}z + \bar{y} \cdot x + \bar{y} \cdot \bar{y}z + \bar{z} \cdot x + \bar{z} \cdot \bar{y}z$$

Simplify each product:

$$x \cdot x = x \quad (\text{idempotent})$$

$$\bar{y} \cdot \bar{y} \cdot z = \bar{y}z \quad (\text{idempotent})$$

$$\bar{z} \cdot \bar{y} \cdot z = \bar{y} \cdot (z \cdot \bar{z}) = 0 \quad (\text{complement})$$

So

$$F = x + x\bar{y}z + x\bar{y} + \bar{y}z + x\bar{z}$$

Factor x out of the terms that contain it:

$$F = x(1 + \bar{y}z + \bar{y} + \bar{z}) + \bar{y}z$$

Annulment ($1 + \text{anything} = 1$):

$$F = x \cdot 1 + \bar{y}z$$

$$\boxed{F = x + \bar{y} \cdot z}$$

Worked Example 2 *Simplification exercise — Lecture 2.2.1*

Simplify $F(x, y, z) = x\bar{y} + x\bar{z} + y\bar{z} + xyz + yz$.

Step 1 — collect the y terms. Terms $y\bar{z}$ and yz share y :

$$y\bar{z} + yz = y(\bar{z} + z) = y \cdot 1 = y$$

$$F = x\bar{y} + x\bar{z} + xyz + y$$

Step 2 — absorb xyz . It contains y , and y is already a term:

$$xyz + y = y(xz + 1) = y \cdot 1 = y$$

$$F = x\bar{y} + x\bar{z} + y$$

Step 3 — absorption variant on $x\bar{y} + y$, which has the form $\bar{A}B + A$ with $A = y$, $B = x$:

$$x\bar{y} + y = x + y$$

$$F = x + y + x\bar{z}$$

Step 4 — absorb $x\bar{z}$ into x (plain absorption):

$$x + x\bar{z} = x(1 + \bar{z}) = x$$

$$\boxed{F = x + y}$$

Worked Example 3 *Using De Morgan — Lecture 2.2.1*Simplify $F = \overline{xy} + \overline{yz} + xz + \overline{xz}$.**Step 1 — split the bar over the outer sum** (De Morgan), pairing the terms:

$$F = \overline{(xy + yz)} \cdot \overline{(xz + \overline{xz})}$$

Step 2 — split again inside each bracket:

$$F = \overline{xy} \cdot \overline{yz} \cdot \overline{xz} \cdot \overline{\overline{xz}}$$

Step 3 — De Morgan on each product, cancelling double negations:

$$F = (\overline{x} + \overline{y}) \cdot (y + \overline{z}) \cdot (\overline{x} + \overline{z}) \cdot (x + z)$$

Step 4 — expand the first pair and the second pair:

$$\begin{aligned} (\overline{x} + \overline{y})(y + \overline{z}) &= \overline{x}y + \overline{x}\overline{z} + \underbrace{\overline{y}y}_{=0} + \overline{y}\overline{z} = \overline{x}y + \overline{x}\overline{z} + \overline{y}\overline{z} \\ (\overline{x} + \overline{z})(x + z) &= \underbrace{\overline{x}x}_{=0} + \overline{x}z + \overline{z}x + \underbrace{\overline{z}z}_{=0} = \overline{x}z + x\overline{z} \end{aligned}$$

Step 5 — multiply the two results. Six cross-products, of which four contain a variable and its complement and therefore vanish:

$$\begin{aligned} F &= (\overline{x}y + \overline{x}\overline{z} + \overline{y}\overline{z})(\overline{x}z + x\overline{z}) \\ &= \overline{x}y\overline{x}z + \underbrace{\overline{x}y x\overline{z}}_{x\overline{x}=0} + \underbrace{\overline{x}\overline{z}\overline{x}z}_{z\overline{z}=0} + \underbrace{\overline{x}\overline{z} x\overline{z}}_{x\overline{x}=0} + \underbrace{\overline{y}\overline{z}\overline{x}z}_{z\overline{z}=0} + \overline{y}\overline{z} x\overline{z} \\ &= \overline{x}y\overline{z} + x\overline{y}\overline{z} \end{aligned}$$

$$\boxed{F = \overline{x} \cdot y \cdot \overline{z} + x \cdot \overline{y} \cdot \overline{z}}$$

Verification by truth table. Let $G = xy + \overline{yz} + xz + \overline{xz}$, so $F = \overline{G}$:

x	y	z	xy	$\overline{yz}$	xz	$\overline{xz}$	G	F
0	0	0	0	0	0	1	1	0
0	0	1	0	1	0	0	1	0
0	1	0	0	0	0	1	1	0
0	1	1	0	0	0	0	0	1
1	0	0	0	0	0	0	0	1
1	0	1	0	1	1	0	1	0
1	1	0	1	0	0	0	1	0
1	1	1	1	0	1	0	1	0

 $F = 1$ only at rows 011 and 100, i.e. $\overline{x}y\overline{z} + x\overline{y}\overline{z}$. ✓

8.6 Tutorial 04 examples

Worked Example 4 $C + \overline{BC}$

$$\begin{aligned}
 C + \overline{BC} &= C + \overline{B} + \overline{C} && \text{De Morgan} \\
 &= C + \overline{C} + \overline{B} && \text{Commutative} \\
 &= 1 + \overline{B} && \text{Complement} \\
 &= 1 && \text{Annulment}
 \end{aligned}$$

The function is a constant — no circuit is needed at all. Recognising this kind of collapse is worth the effort.

Worked Example 5 $\overline{AB}(\overline{A} + B)(\overline{B} + B)$

$$\begin{aligned}
 \overline{AB}(\overline{A} + B)(\overline{B} + B) &= \overline{AB}(\overline{A} + B)(1) && \text{Complement} \\
 &= \overline{AB}(\overline{A} + B) && \text{Identity} \\
 &= (\overline{A} + \overline{B})(\overline{A} + B) && \text{De Morgan} \\
 &= \overline{AA} + \overline{AB} + \overline{B}\overline{A} + \overline{B}B && \text{Distributive} \\
 &= \overline{A} + \overline{AB} + \overline{B}\overline{A} + \overline{B}B && \text{Idempotent} \\
 &= \overline{A} + \overline{AB} + \overline{B}\overline{A} + 0 && \text{Complement} \\
 &= \overline{A} + \overline{AB} + \overline{A}\overline{B} && \text{Identity} \\
 &= \overline{A} + \overline{A}(B + \overline{B}) && \text{Distributive} \\
 &= \overline{A} + \overline{A}(1) && \text{Complement} \\
 &= \overline{A} + \overline{A} && \text{Identity} \\
 &= \overline{A} && \text{Idempotent}
 \end{aligned}$$

Faster route: $(\overline{B} + B) = 1$, and by the combining law $(\overline{A} + \overline{B})(\overline{A} + B) = \overline{A}$ directly. Ten lines become two.

Worked Example 6 $(A + C)(AD + \overline{AD}) + AC + C$

$$\begin{aligned}
 (A+C)(AD + \overline{AD}) + AC + C &= (A+C)A(D + \overline{D}) + AC + C && \text{Distributive} \\
 &= (A+C)A(1) + AC + C && \text{Complement} \\
 &= (A+C)A + AC + C && \text{Identity} \\
 &= A + AC + C && \text{Absorption} \\
 &= A + C(A + 1) && \text{Distributive} \\
 &= A + C(1) && \text{Annulment} \\
 &= A + C && \text{Identity}
 \end{aligned}$$

$$\boxed{F = A + C}$$

8.7 Summary

Situation	Law to reach for
A bar over several terms	De Morgan — break the bar, change the operator.
Two terms differ in one variable's polarity	Combining law — that variable drops out.
A term contains another term entirely	Absorption — the longer term is redundant.
A term contains A and $\overline{A}$	Complement — the term is 0, delete it.
A factor $(1 + \text{anything})$ appears	Annulment — the factor is 1.
Repeated variable in one term	Idempotent — collapse to a single copy.
Stuck	Expand fully to SoP, then look again; or switch to a K-map.

Chapter 9

Karnaugh Maps

Where this fits

Algebraic simplification works but gives no guarantee of minimality and no signal that you are done. The Karnaugh map is a **gate-level minimisation technique**: a graphical rearrangement of the truth table in which the combining law becomes visible as geometric adjacency. Applied correctly, its rules **guarantee** the simplest expression. Sources: Lecture 2.2.1 and Tutorials 05, 06, merged.

9.1 What a K-map is

Karnaugh map

A K-map is a two-dimensional arrangement of the truth table into cells, one cell per minterm, ordered so that **any two horizontally or vertically adjacent cells differ in exactly one variable**.

A function of n variables has 2^n rows in its truth table and therefore 2^n cells in its K-map.

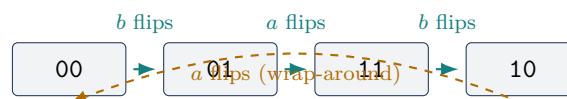
Variables	Cells	Usual shape
2	4	2×2
3	8	2×4
4	16	4×4

9.1.1 The Gray-code ordering

The column and row labels are **not** in binary counting order. They run

00, 01, 11, 10

which is **Gray code** — each label differs from its neighbour in exactly one bit.



Why Gray code, not binary order?

The whole method rests on the combining law

$$A \cdot B + \bar{A} \cdot B = B$$

which applies only when two terms differ in *exactly one* variable. Gray ordering makes that condition equivalent to “the cells are next to each other”, so a purely visual test replaces algebra.

If the columns ran 00, 01, 10, 11, then columns 2 and 3 would differ in *two* bits and grouping them would be invalid.

The edges wrap

The leftmost and rightmost columns are adjacent, and the top and bottom rows are adjacent — check the Gray sequence: $10 \rightarrow 00$ also flips only one bit. Conceptually the map is drawn on a torus (a doughnut), not a flat rectangle. This is why the four corner cells of a 4-variable map form a legal group.

9.2 The three steps

K-map procedure

1. **Mapping** — draw the grid and write the function's output in each cell.
2. **Grouping** — circle the 1s (for SoP) or the 0s (for PoS) according to the rules in Section 9.4.
3. **Deriving** — for each group, write the product (or sum) of the variables that stay *unchanged* across the group; combine the groups.

9.3 A first example, step by step

Worked Example 1 Three-variable K-map — Lecture 2.2.1

Simplify $F = \bar{x}\bar{y}z + \bar{x}yz + x\bar{y}\bar{z} + xy\bar{z}$.

Step 1 — Mapping. Three variables, so eight cells: xy across, z down. Place a 1 in the cell for each minterm.

		xy			
		00	01	11	10
z	0	0	0	1	1
	1	1	1	0	0

Step 2 — Grouping. Two obvious pairs, each of two cells.

		xy			
		00	01	11	10
z	0	0	0	1	1
	1	1	1	0	0

— top group: columns 11 and 10
— bottom group: columns 00 and 01

Step 3 — Deriving.

Red group (row $z = 0$, columns $xy = 11$ and 10):

- $x = 1$ in both columns — **unchanged**.
- y is 1 then 0 — changes, so it drops out.
- $z = 0$ throughout the row — **unchanged**.

Term: $x \cdot \bar{z}$.

Teal group (row $z = 1$, columns $xy = 00$ and 01):

- $x = 0$ in both — unchanged.
- y changes — drops out.

- $z = 1$ — unchanged.

Term: $\bar{x} \cdot z$.

$$F = x \cdot \bar{z} + \bar{x} \cdot z$$

which is $x \oplus z$ — the function does not depend on y at all.

9.4 Grouping rules

Rule 1 — groups contain one value only

Group only 1s (giving Sum of Products) or only 0s (giving Product of Sums). Never mix.

	x	
	0	1
y	0	1
	1	0

Correct

	x	
	0	1
y	0	1
	1	0

Incorrect — mixes 1s and 0s

Rule 2 — groups are rectangular, never diagonal

Groups extend horizontally or vertically only.

	x	
	0	1
y	0	1
	1	0

Correct

	x	
	0	1
y	0	1
	1	0

Incorrect — diagonal

Why not diagonal?

Diagonally adjacent cells differ in *two* variables, so the combining law does not apply. In the map above, the two 1s are $\bar{x}y$ and $x\bar{y}$ — nothing cancels.

Rule 3 — group sizes are powers of two

A group must contain 2^k cells: 1, 2, 4, 8, 16, ... — never 3, 6 or 12.

	ab			
	00	01	11	10
cd	00	01	11	10
	1	1	1	1
	0	0	0	0
	0	0	1	0
	0	0	0	0

Correct — sizes 4 and 1

	ab			
	00	01	11	10
cd	00	01	11	10
	1	1	1	0
	0	0	0	0
	1	1	1	1
	1	0	0	0

Incorrect — a group of 3

Why powers of two?

Each doubling of the group size eliminates exactly one variable:

Group size	Variables eliminated	Literals left ($n=4$)
1	0	4
2	1	3
4	2	2
8	3	1
16	4	0 (the function is constant 1)

A group of 2^k cells covers *all* 2^k combinations of k variables, which is precisely the condition for those k variables to be irrelevant. A group of 3 covers only three of four combinations, so nothing can be factored out cleanly.

Rule 4 — each group must be as large as possible

	<i>ab</i>			
	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	1	1	0	0
10	1	1	0	0

Correct — both groups maximal

	<i>ab</i>			
	00	01	11	10
00	1	1	1	1
01	0	0	0	0
11	1	1	0	0
10	1	1	0	0

Incorrect — two pairs where one quad fits

A smaller group leaves an extra literal in the term, so the circuit needs an extra gate input for no benefit.

Rule 4.1 — overlap groups if that makes them larger

Cells may belong to more than one group.

	<i>ab</i>			
	00	01	11	10
00	1	1	1	1
01	1	1	0	0
11	0	0	0	0
10	0	0	0	0

Correct — the two overlap

	<i>ab</i>			
	00	01	11	10
00	1	1	1	1
01	1	1	0	0
11	0	0	0	0
10	0	0	0	0

Incorrect — second group not maximised

Overlapping costs nothing: a 1 covered twice is still 1, since $A + A = A$.

Rule 4.2 — groups may wrap around the edges

The top row is adjacent to the bottom row, and the leftmost column to the rightmost.

	<i>ab</i>			
	00	01	11	10
00	1	1	0	1
01	0	0	0	0
11	0	0	0	0
10	1	1	0	1

Correct — the four corners form one group of 4

	<i>ab</i>			
	00	01	11	10
00	1	1	0	1
01	0	0	0	0
11	0	0	0	0
10	1	1	0	1

Incorrect — wrap-around not exploited

Rule 4.3 — use as few groups as possible

Having maximised each group, use the minimum number of them needed to cover every 1 (or every 0). Each group becomes one term in the final expression, so fewer groups means fewer gates.

Rules in priority order

1. Only one value per group.
2. Rectangular, never diagonal.
3. Size must be a power of two.
4. Make each group as large as possible (overlapping and wrapping if that helps).
5. Use as few groups as possible while covering everything.

Rules 4 and 5 can conflict; **maximise size first**, then minimise count.

9.5 SoP versus PoS

Desired form	Group	Read each group as
Sum of Products	the 1s	a product; variable plain if it is 1
Product of Sums	the 0s	a sum; variable plain if it is 0

Two ways to do PoS — pick one and stick to it

Method A (direct). Group the 0s and, for each group, write a sum term using the *opposite* polarity convention: variable plain if its constant value is 0, complemented if it is 1.

Method B (via the complement). Group the 0s and read them as if they were 1s — this gives $\overline{F}$ in SoP form. Then apply De Morgan to get F .

Method B involves an extra step but reuses the SoP reading rule, so it produces fewer sign errors. Both give the same answer.

9.6 Worked example: four variables, both forms

Worked Example 2 *Lecture 2.2.1 exercise*

Simplify

$$F = \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}c\overline{d} + \overline{a}b\overline{c}\overline{d} + \overline{a}bcd + a\overline{b}\overline{c}\overline{d} + a\overline{b}c\overline{d}$$

Mapping. Four variables: ab across, cd down.

		ab			
		00	01	11	10
cd	00	1	1	0	1
	01	0	0	0	0
	11	0	0	0	0
	10	1	1	0	1

Grouping for SoP. Two groups of four, both wrapping top-to-bottom.

		ab			
		00	01	11	10
cd	00	1	1	0	1
	01	0	0	0	0
	11	0	0	0	0
	10	1	1	0	1

Group 1

		ab			
		00	01	11	10
cd	00	1	1	0	1
	01	0	0	0	0
	11	0	0	0	0
	10	1	1	0	1

Group 2

Deriving.

Group 1 — columns $ab = 00, 01$ and rows $cd = 00, 10$:

- $a = 0$ in both columns — unchanged.
- b changes (0 then 1) — drops out.
- c changes between rows 00 and 10 — drops out.
- $d = 0$ in both rows — unchanged.

Term: $\bar{a} \cdot \bar{d}$.

Group 2 — columns $ab = 00, 10$ (wrapping) and rows $cd = 00, 10$:

- a changes — drops out.
- $b = 0$ in both columns — unchanged.
- c changes — drops out.
- $d = 0$ — unchanged.

Term: $\bar{b} \cdot \bar{d}$.

$$F = \bar{a} \bar{d} + \bar{b} \bar{d}$$

Six 4-literal terms have become two 2-literal terms.

Now in Product of Sums form. Group the 0s instead.

		<i>ab</i>			
		00	01	11	10
<i>cd</i>	00	1	1	0	1
	01	0	0	0	0
	11	0	0	0	0
	10	1	1	0	1

— rows $cd = 01, 11$ (all columns):
a group of eight with $d = 1$ unchanged.

— column $ab = 11$ (all rows): a
group of four with $a = 1, b = 1$
unchanged.

Reading via the complement (Method B). Treating the 0-groups as if they were 1s gives

$$\overline{F} = d + a \cdot b$$

Applying De Morgan:

$$F = \overline{d + ab} = \overline{d} \cdot \overline{ab} = \overline{d} \cdot (\overline{a} + \overline{b})$$

$$F = (\overline{a} + \overline{b}) \cdot \overline{d}$$

Consistency check. Factor the SoP answer: $\overline{a}\overline{d} + \overline{b}\overline{d} = \overline{d}(\overline{a} + \overline{b})$ — identical. ✓

9.7 Second worked example

Worked Example 3 *Both forms from a given map — Lecture 2.2.1*

Derive the simplified expression in SoP and PoS form.

		<i>ab</i>			
		00	01	11	10
<i>cd</i>	00	1	0	0	1
	01	0	1	1	0
	11	0	0	1	0
	10	1	0	0	1

This is $F = \sum m(0, 2, 5, 8, 10, 13, 15)$.

Sum of Products. Three groups:

		<i>ab</i>			
		00	01	11	10
<i>cd</i>	00	1	0	0	1
	01	0	1	1	0
	11	0	0	1	0
	10	1	0	0	1

— four corners: $b = 0, d = 0 \Rightarrow \overline{b}\overline{d}$

— row $cd=01$, cols 01, 11: $b=1, c=0, d=1 \Rightarrow \overline{b}\overline{c}d$

— col $ab=11$, rows 01, 11: $a=1, b=1, d=1 \Rightarrow abd$

$$F = \overline{b}\overline{d} + \overline{b}\overline{c}d + abd$$

Product of Sums. Group the 0s:

		ab			
		00	01	11	10
cd	00	1	0	0	1
	01	0	1	1	0
	11	0	0	1	0
	10	1	0	0	1

— rows 00, 10 × cols 01, 11:
 $b=1, d=0 \Rightarrow b\bar{d}$

— rows 01, 11 × cols 00, 10:
 $b=0, d=1 \Rightarrow \bar{b}d$

— row $cd=11$, cols 00, 01:
 $a=0, c=1, d=1 \Rightarrow \bar{a}cd$

Reading the 0-groups as $\bar{F}$:

$$\bar{F} = b\bar{d} + \bar{b}d + \bar{a}cd$$

De Morgan:

$$F = (\bar{b} + d) \cdot (b + \bar{d}) \cdot (a + \bar{c} + \bar{d})$$

Both answers were verified against the full 16-row truth table.

9.8 Don't-care conditions

Don't-care condition

An input combination for which the function's output is **not defined** — either because it can never occur, or because nothing downstream depends on it. A function with such combinations is called **incompletely specified** (or incompletely defined). Don't-cares are written $\times$ (or X, or d) in the map.

How to exploit them

A don't-care cell may be treated as **1** or as **0** — *whichever produces larger groups*. You may also ignore it entirely.

The one restriction: **no group may consist only of don't-cares**. There would be no reason to cover a cell whose value nobody cares about.

Worked Example 4 *Don't-cares — Lecture 2.2.1 / Tutorial 06*

$$F(a, b, c, d) = \sum m(1, 3, 5, 7, 9), \quad \text{don't cares} = \sum d(11, 13)$$

Mapping.

		ab			
		00	01	11	10
cd	00	0	0	0	0
	01	1	1	$\times$	1
	11	1	1	0	$\times$
	10	0	0	0	0

Without using the don't-cares (suboptimal).

		<i>ab</i>			
		00	01	11	10
<i>cd</i>	00	0	0	0	0
	01	1	1	×	1
	11	1	1	0	×
	10	0	0	0	0

A quad giving $\bar{a}d$, plus an isolated cell needing all four literals:

$$F = \bar{a}d + ab\bar{c}d$$

Two terms, six literals.

Optimal grouping #1 — treat the $\times$ at $(cd=01, ab=11)$ as 1, making the whole $cd = 01$ row a group of four:

		<i>ab</i>			
		00	01	11	10
<i>cd</i>	00	0	0	0	0
	01	1	1	×	1
	11	1	1	0	×
	10	0	0	0	0

— full row $cd=01$: $c=0, d=1 \Rightarrow \bar{c}d$

— cols 00, 01 $\times$ rows 01, 11: $a=0, d=1 \Rightarrow \bar{a}d$

$$F = \bar{c}d + \bar{a}d$$

Two terms, four literals.

Optimal grouping #2 — use the other $\times$ instead:

		<i>ab</i>			
		00	01	11	10
<i>cd</i>	00	0	0	0	0
	01	1	1	×	1
	11	1	1	0	×
	10	0	0	0	0

— cols 00, 10 (wrapping) $\times$ rows 01, 11: $b=0, d=1 \Rightarrow \bar{b}d$

— as before $\Rightarrow \bar{a}d$

$$F = \bar{b}d + \bar{a}d$$

Also two terms, four literals.

Solution	Terms	Total literals
Ignoring don't-cares	2	6
Optimal #1	2	4
Optimal #2	2	4

Two different but equally minimal answers — a reminder that the *minimum cost* is unique but the *minimal expression* need not be.

9.9 Why the K-map works

Four questions, one answer

Why do we group? Why must groups be 2^k ? Why only horizontal and vertical? Why does only one variable flip between neighbours?

All four have the same answer: a group identifies a set of minterms sharing a common assignment to some variables, over *all* possible assignments of the remaining ones — and those remaining variables are therefore irrelevant.

Consider a group of two cells:

		<i>ab</i>			
		00	01	11	10
<i>c</i>	0	0	1	0	0
	1	0	1	0	0

The grouped terms are $\bar{a}b\bar{c}$ and $\bar{a}bc$. Both have the common state $\bar{a}b$, and that state appears with *both* possible values of c . Therefore

$$\bar{a}b\bar{c} + \bar{a}bc = \bar{a}b(\bar{c} + c) = \bar{a}b$$

The variable c is irrelevant, and the combining law removes it.

Generalising

- A group of **2** cells covers both states of **one** variable $\Rightarrow$ one variable eliminated.
- A group of **4** cells covers all four states of **two** variables $\Rightarrow$ two variables eliminated.
- A group of 2^k cells eliminates k variables.

This is why group sizes must be powers of two — only then does the group cover *every* combination of the eliminated variables.

Now consider an invalid grouping:

		<i>ab</i>			
		00	01	11	10
<i>c</i>	0	0	1	0	1
	1	0	1	1	0

— **valid:** terms $\bar{a}b\bar{c}$ and $\bar{a}bc$ share $\bar{a}b$ across all states of c .

— **also valid:** abc and $\bar{a}bc$ share ac across all states of $b \Rightarrow ac$.

But grouping $\bar{a}b\bar{c}$ with $a\bar{b}\bar{c}$ (a diagonal) would share *nothing* — every variable changes.

Restricting groups to horizontal and vertical rectangles, and ordering the labels so only one variable flips between neighbours, together guarantee that every legal group is a valid application of the combining law.

9.10 Tutorial 06 exercise: the fire alarm

Worked Example 5 Fire alarm system — past paper, 15 marks

A fire alarm system has three smoke detectors S_1, S_2, S_3 . The alarm F activates if **at least two** of the three detectors are activated. (1 = activated.)

(1) Truth table.

S_1	S_2	S_3	F	
0	0	0	0	none active
0	0	1	0	one active
0	1	0	0	one active
0	1	1	1	two active
1	0	0	0	one active
1	0	1	1	two active
1	1	0	1	two active
1	1	1	1	all three

Canonical SoP:

$$F = \overline{S_1}S_2S_3 + S_1\overline{S_2}S_3 + S_1S_2\overline{S_3} + S_1S_2S_3$$

(2a) Simplified SoP via K-map.

		S_1S_2			
		00	01	11	10
S_3	0	0	0	1	0
	1	0	1	1	1

— column 11: $S_1=1, S_2=1 \Rightarrow S_1S_2$
— row $S_3=1$, cols 01, 11: $S_2=1 \Rightarrow S_2S_3$
— row $S_3=1$, cols 11, 10: $S_1=1 \Rightarrow S_1S_3$

$$F = S_1S_2 + S_2S_3 + S_1S_3$$

Read aloud: “the alarm sounds if detectors 1 and 2 fire, or 2 and 3, or 1 and 3” — exactly the specification, which is a good sanity check.

(2b) Simplified PoS via K-map. Group the 0s.

		S_1S_2			
		00	01	11	10
S_3	0	0	0	1	0
	1	0	1	1	1

— row $S_3=0$, cols 00, 01: $S_1=0, S_3=0 \Rightarrow (S_1 + S_3)$
— column 00: $S_1=0, S_2=0 \Rightarrow (S_1 + S_2)$
— row $S_3=0$, cols 00, 10 (wrap): $S_2=0, S_3=0 \Rightarrow (S_2 + S_3)$

$$F = (S_1 + S_2) \cdot (S_1 + S_3) \cdot (S_2 + S_3)$$

Also readable: “at least one of each pair must fire”. Part (3) of this question — implementing it with NAND-only and NOR-only gates — is done in Section 10.8.

9.11 Summary

Step / rule	Detail
Map size	2^n cells for n variables.
Label order	Gray code 00, 01, 11, 10 — one bit changes at a time.
Adjacency	Includes wrap-around at all four edges.
Group value	1s for SoP, 0s for PoS — never mixed.
Group shape	Rectangular; no diagonals.
Group size	Must be 2^k ; each doubling removes one variable.
Maximise then minimise	Largest groups first, then fewest groups.
Overlap	Allowed and often required ($A + A = A$).
Don't-cares	Use as 0 or 1 freely; never form an all- $\times$ group.
Reading a group	Keep the variables that stay constant; drop the rest.

K-map versus algebra

	Algebra	K-map
Guarantees minimum?	no	yes , if rules followed
Tells you when to stop?	no	yes , when all covered
Practical variable limit	any	about 4–6
Error-prone?	yes	less so

Beyond about six variables the map becomes unreadable and algorithmic methods (Quine–McCluskey, or software minimisers) take over.

Chapter 10

Single Gate Type Circuits

Where this fits

Having minimised an expression, the last design question is which physical gates to build it from. Manufacturing pressure pushes towards using one gate type throughout — and remarkably, NAND alone (or NOR alone) suffices for any Boolean function whatsoever. Sources: Lecture 2.2.1 and Tutorial 05, merged.

10.1 Why one gate type?

The commercial argument

- Manufacturing ICs containing several *different* gate types is more expensive than manufacturing a package of identical gates.
- A mixed-type IC leaves **unused gates** whenever a design does not need that particular mix — wasted silicon and wasted board area.
- A single-type design lets you buy one part number in volume and populate every gate.

A standard 7400-series package holds four 2-input NAND gates; if a design uses only NANDs, every gate in every package gets used.

10.2 Functional completeness

Functionally complete set

A set of logical operators is **functionally complete** if it can express *every* possible truth table — equivalently, if any Boolean function can be built using only members of that set.

AND, OR and NOT are the three primary functionalities, so:

The test

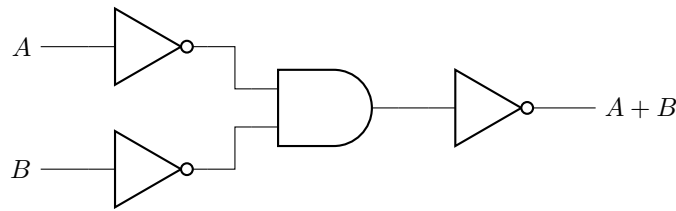
A set of gates is functionally complete if it can reproduce **AND, OR and NOT**. Show those three and you are done, because every Boolean function can be written in sum-of-minterms form, which uses nothing but those three operations.

By definition {AND, OR, NOT} is functionally complete. The interesting question is how much can be removed.

10.3 {AND, NOT} is complete

The missing operation is OR. Using double negation and De Morgan:

$$A + B = \overline{\overline{A + B}} = \overline{\overline{A} \cdot \overline{B}}$$

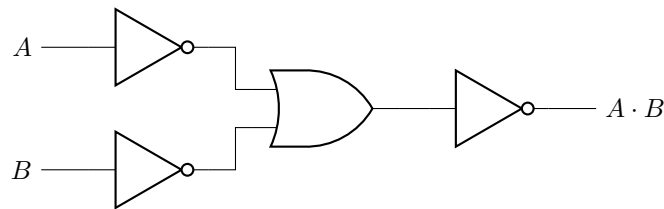


OR is therefore expressible with AND and NOT, so {AND, NOT} is functionally complete.

10.4 {OR, NOT} is complete

Symmetrically, the missing operation is AND:

$$A \cdot B = \overline{\overline{A \cdot B}} = \overline{\overline{A} + \overline{B}}$$



Complete sets so far

{AND, OR, NOT}, {AND, NOT}, {OR, NOT}, {NAND}, {NOR}

The last two are the striking ones — a *single* gate type each.

10.5 NAND as a universal gate

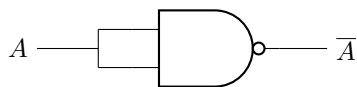
Universal gate

A gate that is functionally complete on its own. NAND and NOR are the two universal gates.

10.5.1 NOT from NAND

Tie both inputs together (or tie one input high):

$$\overline{A \cdot A} = \overline{A}$$

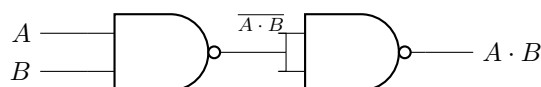


A	$\overline{A \cdot A}$
0	1
1	0

10.5.2 AND from NAND

NAND then invert (the inverter itself being a NAND):

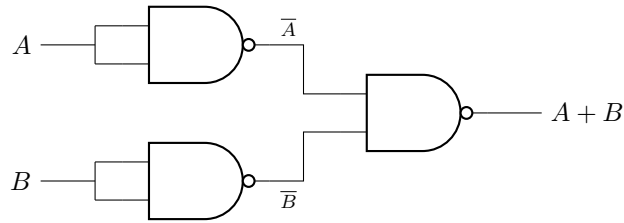
$$\overline{\overline{A \cdot B}} = A \cdot B$$



10.5.3 OR from NAND

Invert both inputs first, then NAND them:

$$A + B = \overline{\overline{A} \cdot \overline{B}}$$



A	B	$\overline{A}$	$\overline{B}$	$\overline{A} \cdot \overline{B}$	$\overline{\overline{A} \cdot \overline{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

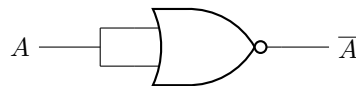
matches OR. ✓

Since AND, OR and NOT are all realisable, **NAND is universal** — and so XOR, NOR and XNOR can also be built from NANDs alone.

10.6 NOR as a universal gate

10.6.1 NOT from NOR

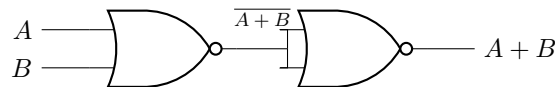
$$\overline{A + A} = \overline{A}$$



10.6.2 OR from NOR

NOR then invert:

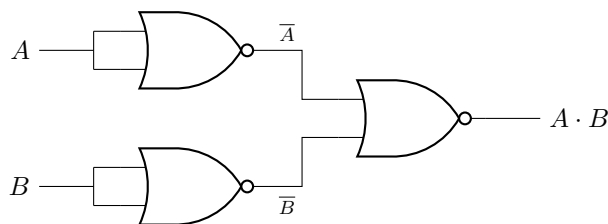
$$\overline{\overline{A + B}} = A + B$$



10.6.3 AND from NOR

Invert both inputs, then NOR:

$$A \cdot B = \overline{\overline{A} + \overline{B}}$$



The duality, one more time

Compare the two sections. Everything swaps:

	NAND	NOR
NOT	tie inputs together	tie inputs together
Native	AND, by inverting the output	OR, by inverting the output
Derived	OR, by inverting the inputs	AND, by inverting the inputs

NAND is naturally suited to **SoP** expressions, NOR to **PoS** expressions. Choosing the right one for the form you have minimises gate count.

10.7 Converting a circuit to NAND-only

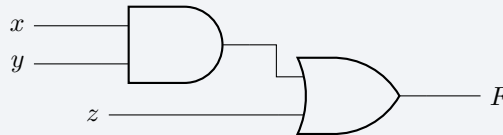
Procedure

1. Start from a **Sum of Products** expression.
2. Replace each AND gate with a NAND followed by an inverter.
3. Replace each OR gate with an OR whose inputs are inverted — which is a NAND.
4. Build every inverter from a NAND with its inputs tied.
5. **Cancel adjacent inverter pairs** — two inversions in series on the same wire do nothing ($\overline{\overline{A}} = A$).

Step 5 is where most of the gate saving happens.

Worked Example 1 Convert $F = x \cdot y + z$ to NAND-only — Lecture 2.2.1

Original circuit:

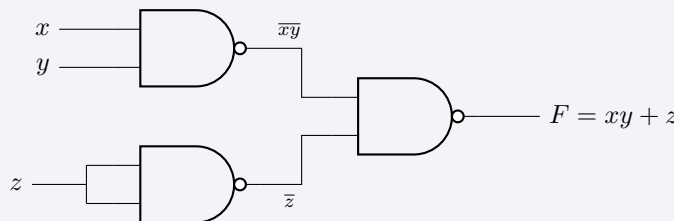


Algebraically: apply double negation, then De Morgan.

$$F = xy + z = \overline{\overline{xy + z}} = \overline{\overline{xy} \cdot \overline{z}}$$

That final form is precisely a NAND of two things: $\overline{xy}$ (a NAND) and $\overline{z}$ (a NAND used as an inverter).

NAND-only circuit — three gates:



Verification:

$$\overline{\overline{xy} \cdot \overline{z}} \stackrel{\text{De Morgan}}{=} \overline{\overline{xy} + \overline{\overline{z}}} \stackrel{\text{double neg.}}{=} xy + z \quad \checkmark$$

x	y	z	$\overline{xy}$	$\overline{z}$	$\overline{xy} \cdot \overline{z}$	F
0	0	0	1	1	1	0
0	0	1	1	0	0	1
0	1	0	1	1	1	0
0	1	1	1	0	0	1
1	0	0	1	1	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	1
1	1	1	0	0	0	1

$F = 1$ exactly when $xy = 1$ or $z = 1$. ✓

Naive substitution wastes gates

Substituting gate-for-gate first gives a circuit with pairs of inverters in series — one from converting AND to NAND+NOT, another from converting OR to NAND with inverted inputs. Those pairs cancel. Always do the cancellation step; the difference here is five gates versus three.

10.8 The fire alarm in NAND-only and NOR-only form

Worked Example 2 *Tutorial 06, Exercise 1 part 3*

From Section 9 the fire alarm function is

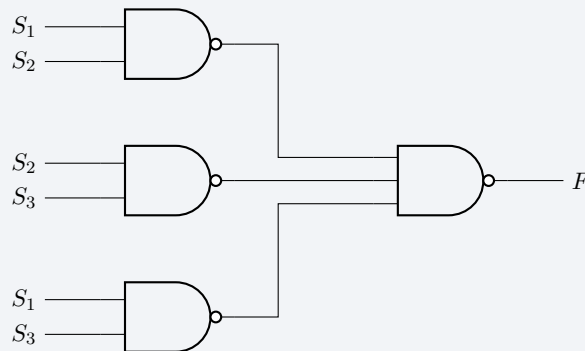
$$F = S_1S_2 + S_2S_3 + S_1S_3 \quad (\text{SoP})$$

$$F = (S_1 + S_2)(S_1 + S_3)(S_2 + S_3) \quad (\text{PoS})$$

NAND-only, from the SoP form. Double-negate and apply De Morgan to the outer OR:

$$F = \overline{\overline{S_1S_2 + S_2S_3 + S_1S_3}} = \overline{\overline{S_1S_2} \cdot \overline{S_2S_3} \cdot \overline{S_1S_3}}$$

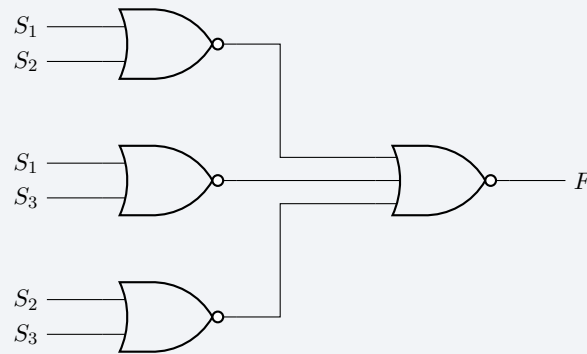
Three 2-input NANDs feeding one 3-input NAND:



NOR-only, from the PoS form. Double-negate and apply De Morgan to the outer AND:

$$F = \overline{\overline{(S_1 + S_2)(S_1 + S_3)(S_2 + S_3)}} = \overline{\overline{S_1 + S_2} + \overline{S_1 + S_3} + \overline{S_2 + S_3}}$$

Three 2-input NORs feeding one 3-input NOR:



Both use exactly four gates. This is the general pattern:

Form	Gate	Structure
Sum of Products	NAND	one NAND per product term + one NAND to combine
Product of Sums	NOR	one NOR per sum term + one NOR to combine

The two-level rule

A minimised two-level SoP expression converts to a two-level NAND-only circuit with a *one-for-one* gate substitution — every AND becomes a NAND, the OR becomes a NAND, and nothing else changes. Likewise a two-level PoS expression converts one-for-one to NOR gates.

This is the single most useful fact in this chapter: minimise to two levels first, then the gate conversion is free.

10.9 Summary

Concept	Key point
Why single gate type	Cheaper ICs, no unused gates.
Functionally complete	Can express AND, OR and NOT — hence everything.
Complete sets	{AND,OR,NOT}, {AND,NOT}, {OR,NOT}, {NAND}, {NOR}.
Universal gate	Functionally complete alone: NAND and NOR.
NOT from NAND/NOR	Tie the inputs together.
NAND suits	Sum of Products (two-level, one-for-one substitution).
NOR suits	Product of Sums (two-level, one-for-one substitution).
Key simplification	Cancel inverter pairs: $\overline{\overline{A}} = A$.

PART III

Combinational Logic Circuits

Chapter 11

Designing Combinational Circuits

Where this fits

Parts I and II gave the raw materials: number representations, gates, and minimisation. This chapter puts them together into a repeatable design procedure, demonstrated on a full worked design — a BCD to Excess-3 code converter. Sources: Lecture 2.3.1 and Tutorial 07, merged.

11.1 Two kinds of digital circuit

Combinational

Output depends *only* on the current inputs.

Purely logic gates.

No feedback paths, no memory.

Examples: adders, decoders, multiplexers.

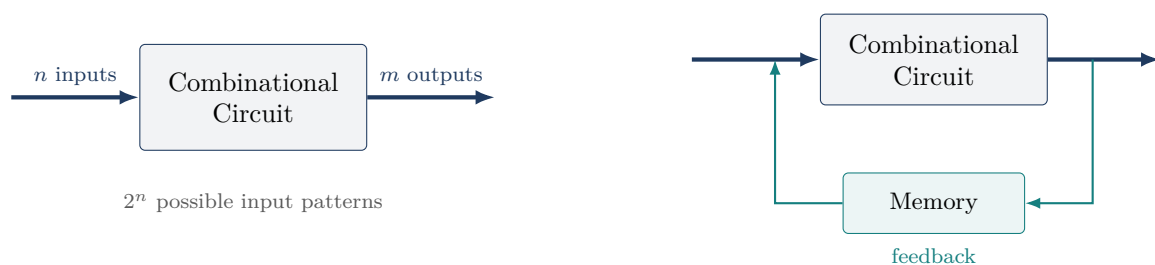
Sequential

Output depends on the current inputs *and* on previous outputs.

Logic gates plus storage elements.

Contains feedback and/or memory.

Examples: flip-flops, registers, counters.



Three equivalent descriptions

Any combinational circuit can be described in three interchangeable ways:

- a **truth table** with 2^n rows;
- m **Boolean functions**, one per output;
- a **logic gate diagram**.

Design is the process of moving from the first to the third, via the second.

11.2 Standard building blocks

Certain combinational circuits recur so often that they are manufactured as standard parts: **adders, subtractors, decoders, encoders, multiplexers, demultiplexers, comparators**. These are available as integrated circuits:

MSI	Medium Scale Integration — tens to hundreds of gates
LSI	Large Scale Integration — thousands of gates
VLSI	Very Large Scale Integration — hundreds of thousands upwards

Chapters 12 and 13 cover these blocks in detail.

11.3 The design procedure

Six steps

1. **Specify** the problem in words; identify inputs and outputs.
2. **Size** the interface — how many input and output lines are needed? (k distinct values need $\lceil \log_2 k \rceil$ lines.)
3. **Tabulate** — build the complete truth table, marking any impossible input patterns as don't-cares.
4. **Minimise** — draw one K-map *per output* and derive a simplified expression for each.
5. **Draw** the gate-level circuit for each output.
6. **Verify** against the original truth table.

One K-map per output, not one per circuit

A circuit with m outputs is m separate Boolean functions sharing the same inputs. Each needs its own map and its own minimisation. Students often try to squeeze all outputs into one map — it cannot be done.

11.4 Worked design: BCD to Excess-3 converter

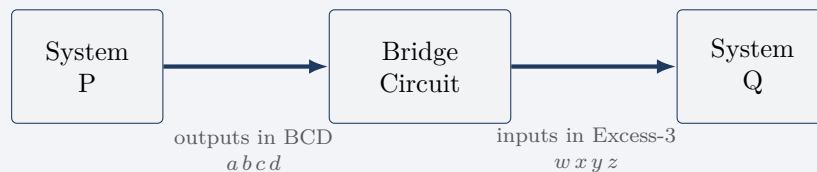
Worked Example 1 *Full design — Lecture 2.3.1*

Step 1 — The problem.

Two digital systems must be connected:

- System **P** produces the decimal digits 0–9 encoded in **Binary Coded Decimal** (BCD): each digit is its plain 4-bit binary value.
- System **Q** expects the same digits in **Excess-3** encoding: each digit is stored as (digit + 3) in 4-bit binary.

A **bridge circuit** (code converter) is needed between them.



Step 2 — Sizing the interface.

P produces 10 different patterns. Since $2^3 = 8 < 10 \leq 16 = 2^4$, we need $\lceil \log_2 10 \rceil = 4$ input lines. Q likewise accepts 10 Excess-3 patterns, so 4 output lines. The bridge is a **4-input, 4-output** combinational circuit.

Step 3 — The conversion table.

Decimal	BCD input				Excess-3 output			
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>w</i>	<i>x</i>	<i>y</i>	<i>z</i>
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

Inputs 1010–1111 never occur $\Rightarrow$ **don't-care conditions**

Sanity check: decimal 9 gives Excess-3 $1100 = 12_{10} = 9 + 3$. ✓

Key observation. The circuit has 4 inputs, so $2^4 = 16$ patterns are possible — but only 10 are defined. The remaining six (1010 through 1111) are **don't-cares**, and exploiting them will simplify the result considerably.

Step 4 — One K-map per output.

Output *w*:

		<i>ab</i>			
		00	01	11	10
<i>cd</i>	00	0	0	×	1
	01	0	1	×	1
	11	0	1	×	×
	10	0	1	×	×

— right two columns, all rows (8 cells):

$a = 1 \Rightarrow \mathbf{a}$

— cols 01, 11 $\times$ rows 11, 10: $b = 1, c = 1 \Rightarrow \mathbf{bc}$

— cols 01, 11 $\times$ rows 01, 11: $b = 1, d = 1 \Rightarrow \mathbf{bd}$

$$w = a + bc + bd$$

Output *x*:

		<i>ab</i>			
		00	01	11	10
<i>cd</i>	00	0	1	×	0
	01	1	0	×	1
	11	1	0	×	×
	10	1	0	×	×

— cols 00, 10 (wrap) $\times$ rows 11, 10: $b = 0, c = 1 \Rightarrow \mathbf{\bar{b}c}$

— cols 00, 10 (wrap) $\times$ rows 01, 11: $b = 0, d = 1 \Rightarrow \mathbf{\bar{b}d}$

— row $cd=00$, cols 01, 11: $b = 1, c = 0, d = 0 \Rightarrow \mathbf{b\bar{c}\bar{d}}$

$$x = \bar{b}c + \bar{b}d + b\bar{c}\bar{d}$$

Output *y*:

		ab			
		00	01	11	10
cd	00	1	1	×	1
	01	0	0	×	0
	11	1	1	×	×
	10	0	0	×	×

— entire row $cd=00$: $c = 0, d = 0 \Rightarrow \bar{c}\bar{d}$

— entire row $cd=11$: $c = 1, d = 1 \Rightarrow cd$

$$y = \bar{c}\bar{d} + cd$$

which is $\bar{c} \oplus \bar{d}$ — an **XNOR** gate.

Output z :

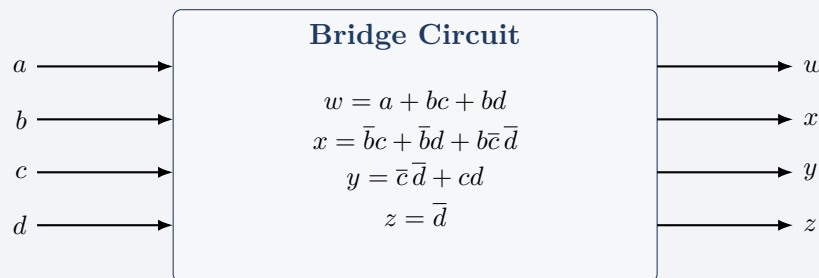
		ab			
		00	01	11	10
cd	00	1	1	×	1
	01	0	0	×	0
	11	0	0	×	×
	10	1	1	×	×

— rows $cd = 00$ and $cd = 10$, wrapping top-to-bottom: a group of **eight** cells with $d = 0$ throughout.

$$z = \bar{d}$$

A single inverter. Which makes sense: adding 3 to any number flips its least significant bit, because 3 is odd.

Step 5 — The completed design.



Step 6 — Verification. Spot-check decimal 6, input $abcd = 0110$:

$$w = a + bc + bd = 0 + (1)(1) + (1)(0) = 1$$

$$x = \bar{b}c + \bar{b}d + b\bar{c}\bar{d} = 0 + 0 + (1)(0)(1) = 0$$

$$y = \bar{c}\bar{d} + cd = (0)(1) + (1)(0) = 0$$

$$z = \bar{d} = 1$$

$$wxyz = 1001 = 9_{10} = 6 + 3. \checkmark$$

What this example demonstrates

- **Don't-cares are valuable.** Six of sixteen input patterns were undefined, and using them turned z from a multi-term expression into a single inverter.
- **Recognise standard functions.** y turned out to be XNOR — one gate instead of three.
- **Outputs are independent.** Four separate K-maps, four separate minimisations.

- **Sanity-check against the specification.** $z = \bar{d}$ has an arithmetic explanation, which is strong evidence the derivation is right.

11.5 A note on BCD

Binary Coded Decimal

An encoding in which each decimal digit is represented separately by its own 4-bit binary value. The number 47_{10} becomes 0100 0111 in BCD, not 101111_2 as in pure binary.

	Advantage	Disadvantage
BCD	Trivial conversion to/from decimal display; no rounding error on decimal fractions	Wastes 6 of 16 patterns per digit ($\approx 20\%$ storage overhead)

BCD is still used in calculators, digital clocks, point-of-sale terminals and financial systems — anywhere exact decimal behaviour matters more than compactness.

Why Excess-3 existed

Excess-3 is a *self-complementing* code: the 9's complement of a digit is obtained by simply inverting all four bits. For example, decimal 4 is Excess-3 0111; inverting gives 1000, which is Excess-3 for 5, and $4 + 5 = 9$. ✓

This made subtraction dramatically simpler in early decimal computers — the same trick as two's complement, one level up. It also gives no all-zeros code word, which was useful for detecting a dead transmission line.

11.6 Summary

Step	Detail
Specify	Identify what the circuit must do; name inputs and outputs.
Size	k distinct values need $\lceil \log_2 k \rceil$ lines.
Tabulate	All 2^n rows; mark impossible patterns as don't-cares.
Minimise	One K-map <i>per output</i> ; exploit the don't-cares.
Draw	Gate diagram per output; look for standard functions (XOR, XNOR).
Verify	Substitute a few rows back into the derived expressions.

Chapter 12

Adders, Subtractors and the ALU

Where this fits

Addition is the single most important operation in a processor — subtraction, multiplication, division, address calculation and comparison are all built on it. This chapter derives the adder from its truth table and scales it up to multi-bit arithmetic. Sources: Lecture 2.3.1 and Tutorial 07, merged.

12.1 Half adder

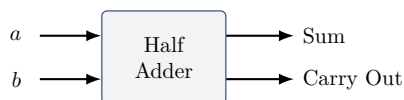
Half adder

A combinational circuit that adds **two** single bits, producing a **Sum** and a **Carry out**. It cannot accept a carry in — hence “half”.

There are four cases:

$$\begin{array}{r} 0 \\ + 0 \\ \hline 0 \end{array} \quad \begin{array}{r} 0 \\ + 1 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ + 0 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ + 1 \\ \hline 10 \end{array}$$

Only the last case produces a carry — and it is precisely that carry which forces the circuit to have two outputs.



Inputs		Outputs	
<i>a</i>	<i>b</i>	Sum	Carry Out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Reading the outputs off the table

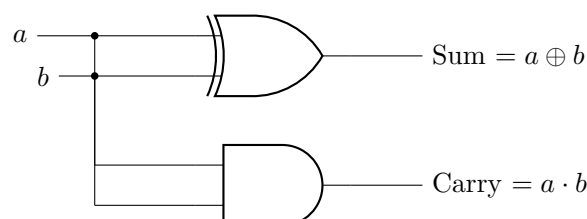
The **Sum** column is 0, 1, 1, 0 — the **XOR** truth table.

The **Carry** column is 0, 0, 0, 1 — the **AND** truth table.

$$\text{Sum} = a \oplus b$$

$$\text{Carry Out} = a \cdot b$$

No K-map needed — just recognition.



12.2 Full adder

A half adder cannot chain, because adding multi-bit numbers requires each column to accept a carry from the column to its right:

Carry in	I_3	I_2	I_1	
Number 1 (x)	x_3	x_2	x_1	x_0
Number 2 (y)	y_3	y_2	y_1	y_0
Sum	S_3	S_2	S_1	S_0
Carry out	C_3	C_2	C_1	C_0

Full adder

A combinational circuit that adds **three** bits — two operand bits and a carry in — producing a Sum and a Carry out.



Inputs			Outputs	
a	b	C_{in}	Sum	C_{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

12.2.1 Deriving the expressions

Canonically, from the rows where each output is 1:

$$\text{Sum} = \bar{a}\bar{b}C_{in} + \bar{a}b\overline{C_{in}} + a\bar{b}\overline{C_{in}} + abC_{in}$$

$$C_{out} = \bar{a}bC_{in} + a\bar{b}C_{in} + ab\overline{C_{in}} + abC_{in}$$

C_{out} simplifies neatly. Using abC_{in} three times (idempotence):

$$\begin{aligned}
 C_{out} &= (\bar{a}bC_{in} + abC_{in}) + (a\bar{b}C_{in} + abC_{in}) + (ab\overline{C_{in}} + abC_{in}) \\
 &= bC_{in}(\bar{a} + a) + aC_{in}(\bar{b} + b) + ab(\overline{C_{in}} + C_{in}) \\
 &= bC_{in} + aC_{in} + ab
 \end{aligned}$$

The standard results

$$\boxed{\text{Sum} = a \oplus b \oplus C_{in}}$$

$$\boxed{C_{out} = ab + bC_{in} + aC_{in}}$$

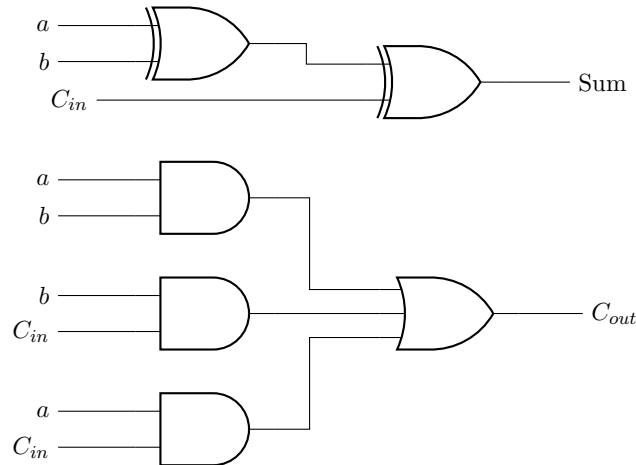
Equivalently, in the form used when building from half adders:

$$\text{Sum} = a \oplus b \oplus C_{in} \quad C_{out} = a \cdot b + C_{in} \cdot (a \oplus b)$$

Reading the two outputs in plain English

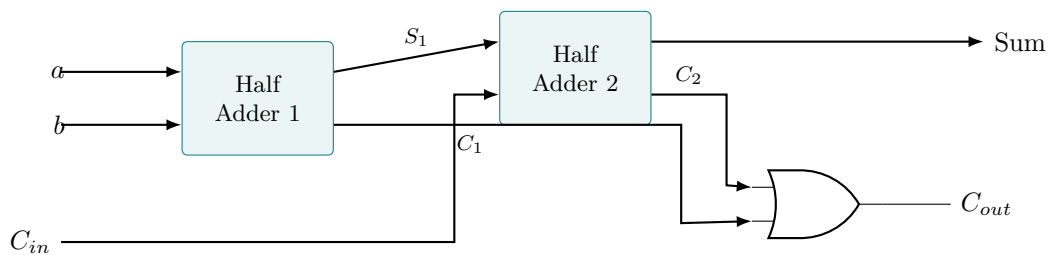
Sum is 1 when an *odd number* of the three inputs are 1 (rows 001, 010, 100, 111) — that is exactly 3-input XOR.

Carry out is 1 when *at least two* of the three inputs are 1 — a majority function. Note this is the same function as the fire alarm in Chapter 9!



12.2.2 Full adder from two half adders

A full adder adds three bits; a half adder adds two. So: add the first two bits, then add the carry in to that result.



Worked Example 1 Verifying the two-half-adder construction

Sum: half adder 1 gives $S_1 = a \oplus b$; half adder 2 gives $S_2 = S_1 \oplus C_{in} = a \oplus b \oplus C_{in}$. ✓

a	b	C_{in}	$S_1 = a \oplus b$	$S_2 = S_1 \oplus C_{in}$	Sum (expected)
0	0	0	0	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	0	0	0	0
0	0	1	0	1	1
0	1	1	1	0	0
1	0	1	1	0	0
1	1	1	0	1	1

Carry out: the two half-adder carries are ORed.

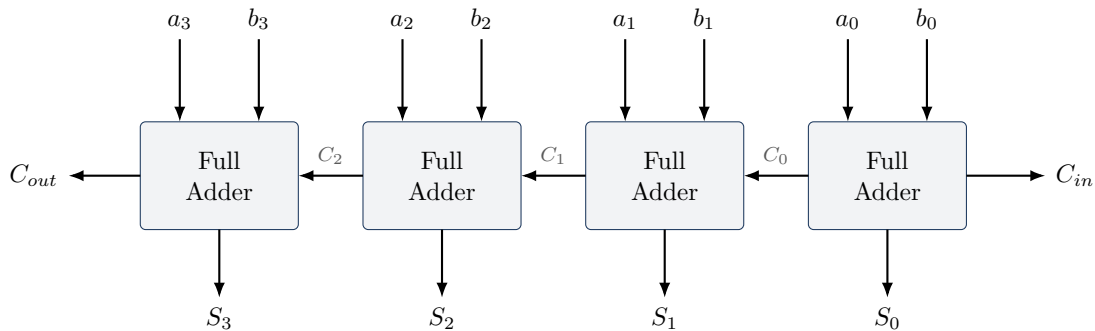
a	b	C_{in}	C_1	C_2	$C_{out} = C_1 + C_2$
0	0	0	0	0	0
0	1	0	0	0	0
1	0	0	0	0	0
1	1	0	1	0	1
0	0	1	0	0	0
0	1	1	0	1	1
1	0	1	0	1	1
1	1	1	1	0	1

Why OR is safe here

Notice C_1 and C_2 are **never both 1** — the combination $C_1 = C_2 = 1$ is a don't-care that can never arise. If a and b are both 1 then $S_1 = 0$, so half adder 2 cannot generate a carry. This is why a plain OR suffices rather than another adder.

12.3 Ripple carry adder

Chaining n full adders, each one's carry out feeding the next one's carry in, gives an n -bit adder.



Ripple carry adder

An n -bit adder built from n full adders in series, in which the carry “ripples” from the least significant stage to the most significant.

The performance problem

Stage i cannot compute its correct output until stage $i - 1$ has settled. The worst-case delay is therefore **proportional to n** : a 64-bit ripple adder is roughly 64 gate-delays deep.

Real processors use **carry-lookahead** adders, which compute all the carries in parallel from “generate” ($g_i = a_i b_i$) and “propagate” ($p_i = a_i \oplus b_i$) signals, reducing the delay to roughly $\log n$ at the cost of more gates. The trade-off — area against speed — is the recurring theme of hardware design.

12.4 Half subtractor

Half subtractor

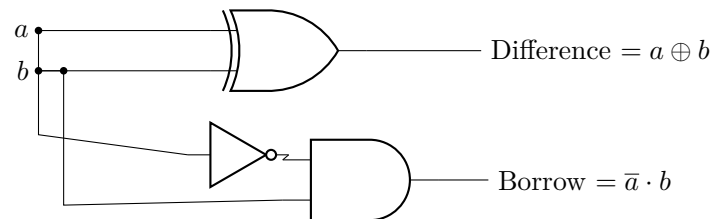
A combinational circuit computing $a - b$ for two single bits, producing a **Difference** and a **Borrow out**. It does not accept a borrow in.

Inputs		Outputs	
a	b	Diff	B. Out
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

$$\text{Difference} = a \oplus b$$

$$\text{Borrow Out} = \bar{a} \cdot b$$

The borrow is set only in the case $0 - 1$, when the subtractor must borrow from the next digit up.



Compare with the half adder

	Half adder	Half subtractor
First output	Sum = $a \oplus b$	Difference = $a \oplus b$
Second output	Carry = $a \cdot b$	Borrow = $\bar{a} \cdot b$

Identical apart from one inverter. So yes — a half adder *can* be repurposed as a half subtractor by inverting a into the AND gate.

Direction matters

The half subtractor computes $a - b$, **not** $b - a$. The circuit is not symmetric: swapping the inputs changes the borrow output (though not the difference, since XOR is symmetric).

12.5 Full subtractor

Full subtractor

Computes $a - b - B_{in}$ for single bits, where a is the **minuend**, b the **subtrahend** and B_{in} the borrow from the previous (lower) position.

Inputs			Outputs	
a	b	B_{in}	Diff	B_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$\text{Diff} = a \oplus b \oplus B_{in}$$

$$B_{out} = \bar{a}b + \bar{a}B_{in} + bB_{in}$$

equivalently

$$B_{out} = \bar{a}b + (\bar{a} + b) B_{in}$$

Full adder versus full subtractor

$$C_{out} = ab + aC_{in} + bC_{in}$$

$$B_{out} = \bar{a}b + \bar{a}B_{in} + bB_{in}$$

Identical apart from complementing a . The Difference expression is identical to the Sum expression. This structural similarity is why the same silicon can do both.

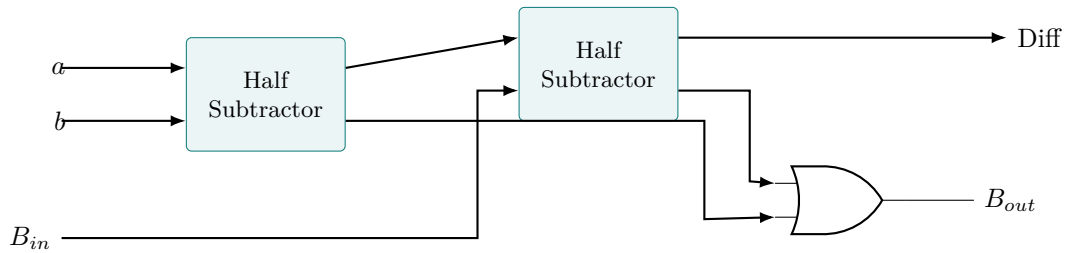
Worked Example 2 Checking a row

Row $abB_{in} = 011$: compute $0 - 1 - 1 = -2$.

In two's-complement-free borrow arithmetic: borrow 2 from the next column, giving $2 - 1 - 1 = 0$. So Difference = 0 and $B_{out} = 1$. ✓

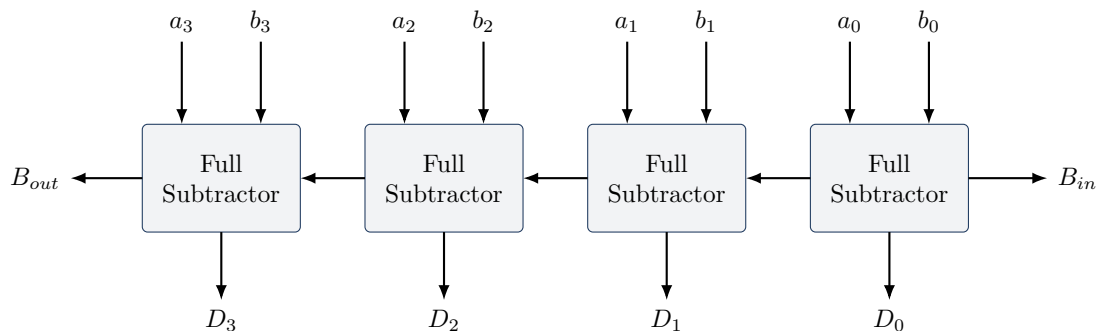
Row $abB_{in} = 111$: $1 - 1 - 1 = -1$. Borrow 2: $2 + 1 - 1 - 1 = 1$. Difference = 1, $B_{out} = 1$. ✓

12.5.1 Full subtractor from half subtractors



Exactly the same topology as the full adder built from half adders.

12.5.2 Multi-bit subtraction



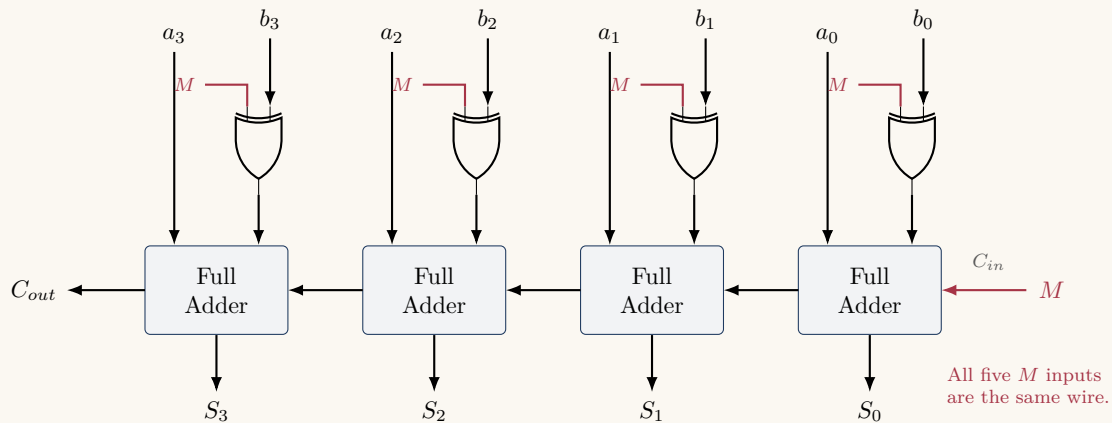
12.6 The adder–subtractor

How real hardware subtracts

Rather than building both an adder and a subtractor, processors build one adder and exploit two's complement (Chapter 4):

$$A - B = A + \overline{B} + 1$$

A single control line M selects the mode:



How it works. Each b_i passes through an XOR gate whose other input is M :

- $M = 0$: $b_i \oplus 0 = b_i$, and $C_{in} = 0$ — plain addition $A + B$.
- $M = 1$: $b_i \oplus 1 = \overline{b_i}$, and $C_{in} = 1$ — so the adder computes $A + \overline{B} + 1$, which is $A - B$.

The XOR gate acting as a *controlled inverter* is the trick that makes this work — exactly the property noted in Chapter 6. One adder, one control bit, both operations.

12.7 Multipliers

Left as an exercise in Lecture 2.3.1

A 2×2 bit multiplier. Let $A = a_1a_0$ and $B = b_1b_0$; the product $P = p_3p_2p_1p_0$ needs four bits, since $3 \times 3 = 9 = 1001_2$.

Long multiplication gives the partial products:

$$\begin{array}{r}
 \\
 \times \\
 \hline
 b_0 b_0 \\
 + a_1 b_1 b_1 \\
 \hline
 p_3 p_2 p_0
 \end{array}$$

Each partial product is a single AND gate. Summing the columns needs two half adders:

$$\begin{aligned} p_0 &= a_0b_0 \\ p_1 &= a_1b_0 \oplus a_0b_1 && \text{(HA 1 sum)} \\ c_1 &= a_1b_0 \cdot a_0b_1 && \text{(HA 1 carry)} \\ p_2 &= a_1b_1 \oplus c_1 && \text{(HA 2 sum)} \\ p_3 &= a_1b_1 \cdot c_1 && \text{(HA 2 carry)} \end{aligned}$$

Total: **4 AND gates and 2 half adders.**

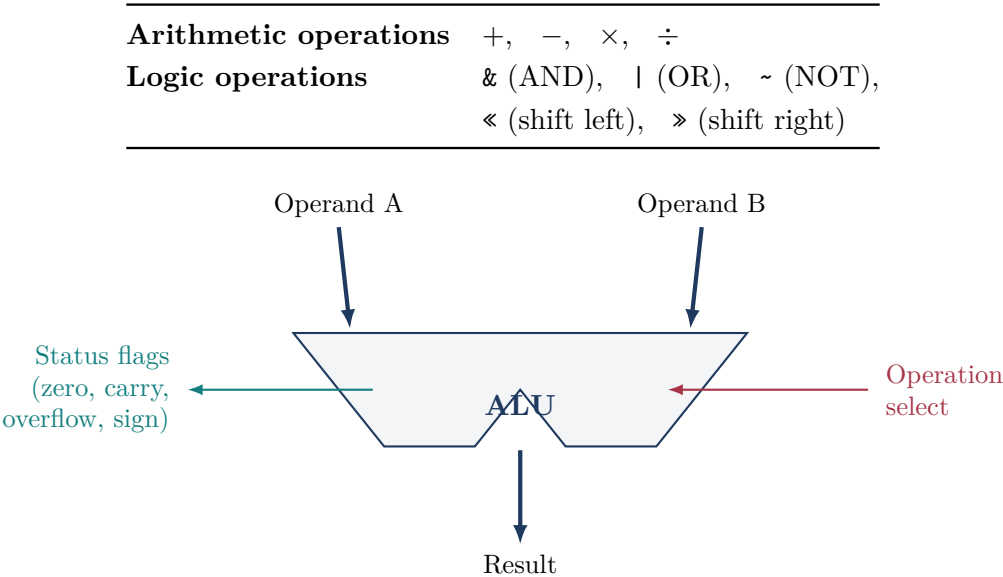
Check 3×3 : $a_1a_0 = 11$, $b_1b_0 = 11$. $p_0 = 1$; $a_1b_0 = 1$ and $a_0b_1 = 1$ so $p_1 = 0$, $c_1 = 1$; $a_1b_1 = 1$ so $p_2 = 1 \oplus 1 = 0$ and $p_3 = 1$. $P = 1001 = 9_{10}$. ✓

Scaling up. An $n \times n$ array multiplier needs n^2 AND gates and roughly $n(n - 1)$ adder cells — growing *quadratically*, which is why multiplication costs more silicon and more cycles than addition (Chapter 18).

12.8 The Arithmetic Logic Unit

ALU

The **Arithmetic Logic Unit** is the part of the CPU that performs computation. It contains many circuit units — adders, subtractors, multipliers, shifters and logic gates — with a multiplexer (Chapter 13) selecting which result to output.



The ALU is combinational

Everything in this chapter is combinational — outputs depend only on the current inputs. The ALU has no memory of its own; the *registers* that feed it and store its results are sequential circuits, covered in Part IV.

12.9 Summary

Circuit	Inputs	Outputs	Expressions
Half adder	a, b	Sum, C_{out}	$a \oplus b$; ab
Full adder	a, b, C_{in}	Sum, C_{out}	$a \oplus b \oplus C_{in}$; $ab + aC_{in} + bC_{in}$
Half subtractor	a, b	Diff, B_{out}	$a \oplus b$; $\bar{a}b$
Full subtractor	a, b, B_{in}	Diff, B_{out}	$a \oplus b \oplus B_{in}$; $\bar{a}b + \bar{a}B_{in} + bB_{in}$

The pattern worth remembering

The first output is always XOR of all inputs. The second output is a majority function — of a, b, C_{in} for the adder, and of $\bar{a}, b, B_{in}$ for the subtractor. Learn that and you can reconstruct all four circuits from memory.

Chapter 13

Decoders, Encoders and Multiplexers

Where this fits

These are the *routing and selection* building blocks. Adders compute; decoders, encoders and multiplexers decide *where data goes* and *which data is used*. They appear inside memory, inside the control unit, and on every bus. Sources: Lecture 2.3.1 and Tutorial 07, merged.

13.1 Decoders

Decoder

A circuit that interprets an n -bit input code by activating exactly **one** of its 2^n output lines. Each output corresponds to one minterm of the input variables.



exactly one output is active at a time

Sizing

An n -bit code space contains 2^n distinct codes, so a decoder needs 2^n output channels to give each code its own line. Common sizes: 2×4 , 3×8 , 4×16 .

A decoder may have *fewer* than 2^n outputs if some codes are unused — a BCD-to-decimal decoder has 4 inputs but only 10 outputs.

13.1.1 The 2×4 decoder

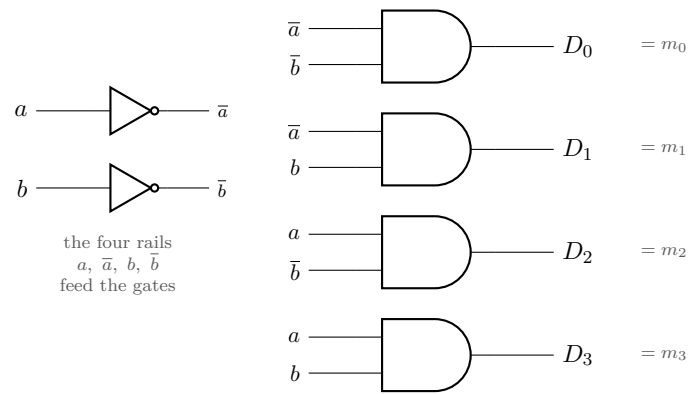
a	b	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

$$D_0 = \bar{a} \cdot \bar{b} \quad (m_0)$$

$$D_1 = \bar{a} \cdot b \quad (m_1)$$

$$D_2 = a \cdot \bar{b} \quad (m_2)$$

$$D_3 = a \cdot b \quad (m_3)$$

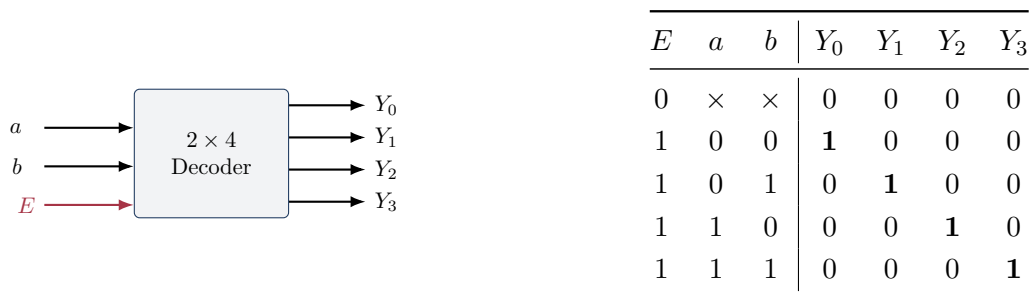


The decoder *is* a minterm generator

Each AND gate produces one minterm. Feed in the code $ab = 01$ and only D_1 goes high. This is the property exploited in Section 13.1.4.

13.1.2 Decoders with an enable input

Most practical decoders add a single **Enable** line E :



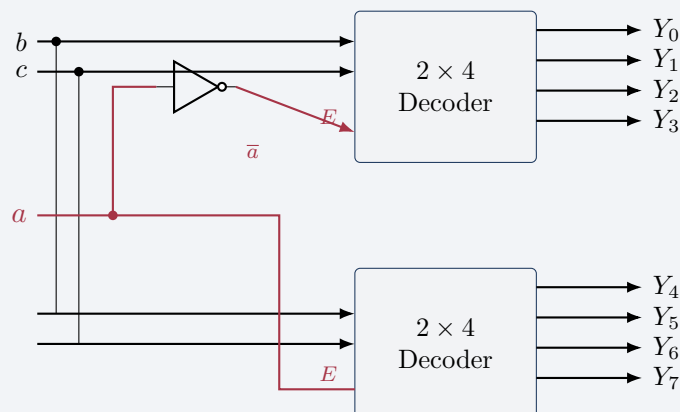
- $E = 1$: the decoder operates normally.
- $E = 0$: *all* outputs are inactive, regardless of the inputs.

Implementation is trivial — add E as an extra input to every AND gate.

13.1.3 Building bigger decoders from smaller ones

Worked Example 1 Two 2×4 decoders make one 3×8 — Lecture 2.3.1 exercise

Feed the two low-order bits b, c to both decoders' data inputs, and use the high-order bit a to select *which* decoder is enabled.



a	b	c	Active output	
0	0	0	Y_0	top decoder enabled ($\bar{a} = 1$)
0	0	1	Y_1	
0	1	0	Y_2	
0	1	1	Y_3	
1	0	0	Y_4	bottom decoder enabled ($a = 1$)
1	0	1	Y_5	
1	1	0	Y_6	
1	1	1	Y_7	

Conclusion: the circuit is equivalent to a single 3×8 decoder. The enable input is what makes this composition possible, and the same trick scales: two 3×8 decoders plus an inverter give a 4×16 .

13.1.4 Decoder as a general function generator

The idea

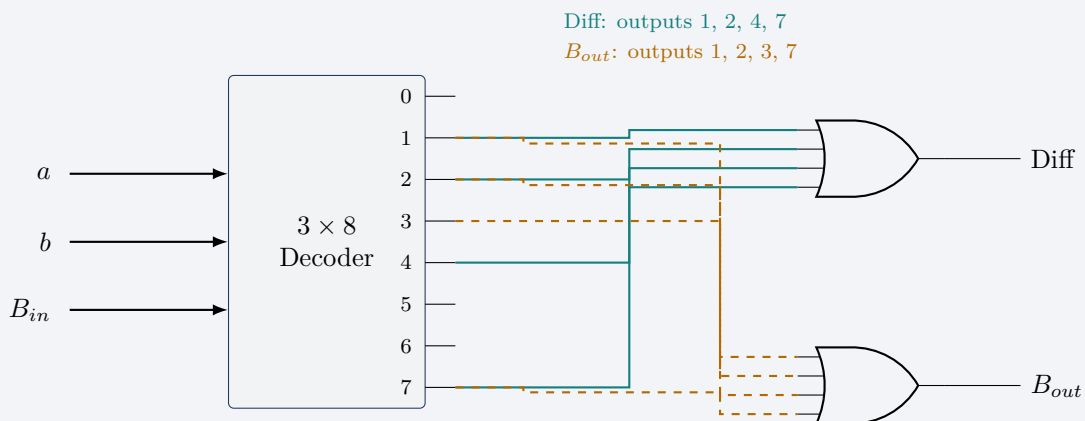
- Any Boolean function can be written as a **sum of minterms**.
- An n -input decoder generates **all** 2^n minterms of n variables — one per output line.
- So: connect the relevant decoder outputs to an OR gate and you have the function. *No minimisation required.*

One decoder can serve several functions at once, each with its own OR gate — which is why this is efficient when a circuit has many outputs over the same inputs.

Worked Example 2 Full subtractor from a 3×8 decoder — Lecture 2.3.1

From Chapter 12, the full subtractor's truth table gives

$$\text{Diff} = \sum m(1, 2, 4, 7) \quad B_{out} = \sum m(1, 2, 3, 7)$$



Two OR gates and one decoder replace the whole gate-level design.

When is this a good idea?

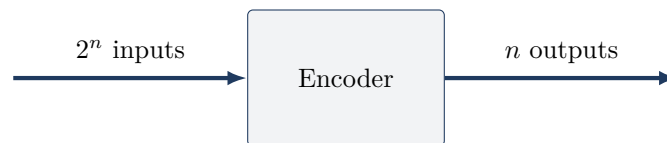
Good when a circuit has many outputs sharing the same inputs (code converters, instruction decoders) — the decoder cost is paid once.

Poor for a single output with few minterms — a minimised gate network is smaller. A 4×16 decoder is 16 AND gates; if your function needs three product terms, use three AND gates instead.

13.2 Encoders

Encoder

The inverse of a decoder: it accepts up to 2^n input lines, of which exactly one is active, and produces the n -bit binary code identifying that line.



13.2.1 4×2 encoder

Y_0	Y_1	Y_2	Y_3	A_1	A_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

$$A_1 = Y_2 + Y_3$$

$$A_0 = Y_1 + Y_3$$

Just two OR gates.

Reading the equations off the table

Output bit A_k is 1 for exactly those input lines whose index has bit k set. A_1 (weight 2) is set for lines 2 and 3; A_0 (weight 1) is set for lines 1 and 3. Hence the OR terms.

13.2.2 8×3 encoder

Left as “search, read, explore” in Lecture 2.3.1

Eight inputs $D_0 \dots D_7$, three outputs $A_2 A_1 A_0$. Applying the same rule — A_k ORs every line whose index has bit k set:

Active input	A_2	A_1	A_0
D_0	0	0	0
D_1	0	0	1
D_2	0	1	0
D_3	0	1	1
D_4	1	0	0
D_5	1	0	1
D_6	1	1	0
D_7	1	1	1

$$A_2 = D_4 + D_5 + D_6 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

$$A_0 = D_1 + D_3 + D_5 + D_7$$

Three 4-input OR gates.

Two problems with the simple encoder:

1. **D_0 is never referenced.** The output 000 appears both when D_0 is active and when *nothing* is active — an ambiguity. Solutions add a “valid” output that is the OR of all inputs.
2. **Multiple active inputs give nonsense.** If D_3 and D_5 are both high, the output is $011 \mid 101 = 111$, i.e. D_7 — which is wrong for both.

Priority encoder. The standard fix: if several inputs are active, encode the *highest-numbered* one. For a 4×2 priority encoder:

$$A_1 = D_3 + D_2$$

$$A_0 = D_3 + D_2' D_1$$

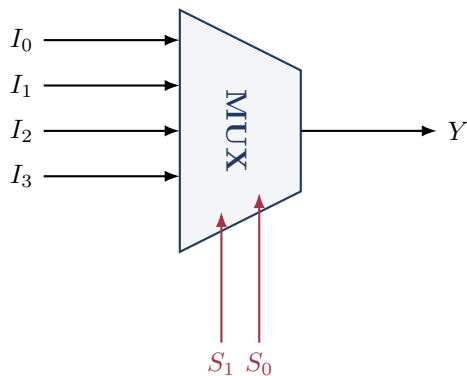
$$V = D_0 + D_1 + D_2 + D_3 \quad (\text{valid})$$

Priority encoders are what turn many simultaneous interrupt request lines into a single interrupt number for the CPU — see the interrupt handling in Chapter 19.

13.3 Multiplexers

Multiplexer (MUX)

A circuit that routes **one of many** input channels to a **single** output channel. Which input is selected is determined by a set of **control (select) lines**. Also called a **data selector**.



S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Sizing

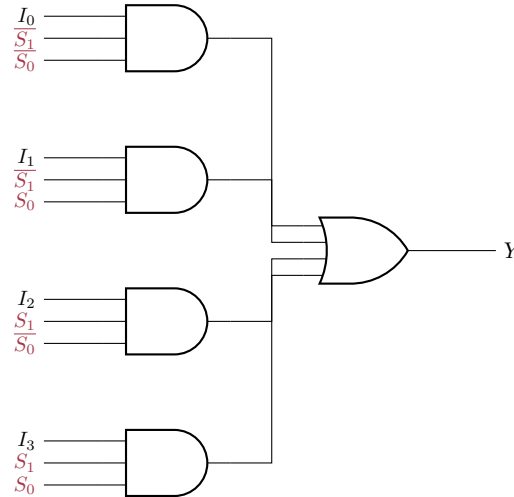
n control lines select among 2^n inputs:

Control lines	Inputs	Name
1	2	2 : 1 MUX
2	4	4 : 1 MUX
3	8	8 : 1 MUX
4	16	16 : 1 MUX

13.3.1 Internal structure

Each input is ANDed with the minterm of the control lines that selects it, and all the results are ORed:

$$Y = I_0 \overline{S_1} \overline{S_0} + I_1 \overline{S_1} S_0 + I_2 S_1 \overline{S_0} + I_3 S_1 S_0$$



A MUX contains a decoder

The AND gates are generating the four minterms of S_1, S_0 — that is exactly a 2×4 decoder. A multiplexer is a decoder whose outputs gate the data inputs, followed by an OR. Recognising this makes both circuits easier to remember.

13.3.2 Multiplexer as a function generator

A standard exam trick

A $2^n : 1$ MUX can implement *any* function of n variables: wire the variables to the select lines and hard-wire each data input to the constant 0 or 1 that the truth table requires. For $F(x, y) = \sum m(0, 2, 3)$ using a 4:1 MUX with $S_1 = x$, $S_0 = y$:

$$I_0 = 1, \quad I_1 = 0, \quad I_2 = 1, \quad I_3 = 1$$

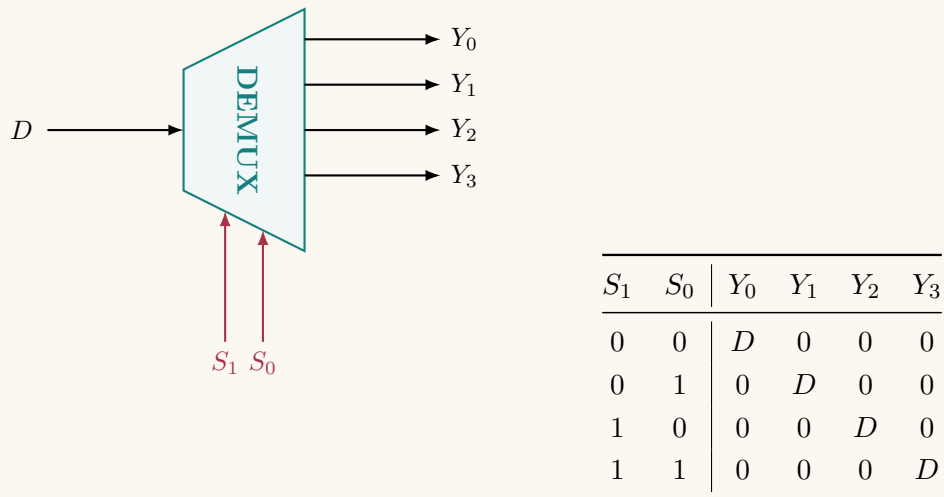
Better still, a $2^{n-1} : 1$ MUX can implement any function of n variables by feeding the last variable (or its complement, or a constant) into the data inputs — halving the multiplexer size.

13.4 Demultiplexers

Left as a reading exercise in Lecture 2.3.1

Demultiplexer (DEMUX)

The inverse of a multiplexer: it routes a **single** input channel to **one of many** output channels, chosen by the control lines.



$$Y_0 = D\overline{S_1}\overline{S_0}, \quad Y_1 = D\overline{S_1}S_0, \quad Y_2 = DS_1\overline{S_0}, \quad Y_3 = DS_1S_0$$

A DEMUX is a decoder with an enable. Compare these equations with the enabled decoder in Section 13.1.2: identical, with D playing the role of E . Manufacturers frequently sell one chip labelled “decoder/demultiplexer”.

MUX and DEMUX together form the classic long-distance link: many channels are multiplexed onto one shared wire at the transmitting end and demultiplexed back into many channels at the receiving end.



13.5 Where these blocks are used

Worked Example 3 *Uses in a computer system — Lecture 2.3.1 exercise*

Decoders

- **Memory address decoding.** The high-order address bits select which memory chip or bank responds; the decoder drives the chip-select lines. This is visible in the RAM diagram in Chapter 16.
- **Instruction decoding.** The control unit decodes the opcode field of an instruction to activate the correct execution circuitry (Chapter 20).
- **Register file write port.** The destination register number is decoded to a one-hot enable across all registers.
- **Display drivers.** BCD-to-seven-segment decoders drive digital displays.

Multiplexers

- **Bus sharing.** Several devices need to drive one bus; a MUX selects which one at any moment.
- **ALU output selection.** The ALU computes several results in parallel; the operation-select lines drive a MUX that picks the one to keep (Section 12.8).
- **Register file read port.** The source register number selects which register’s contents appear on the read port.

- **Operand selection.** Choosing between a register value and an immediate value in the instruction (Chapter 21).

Encoders

- **Interrupt controllers.** Many interrupt request lines are encoded (with priority) into a single interrupt number.
- **Keyboards.** A key press on a matrix is encoded into a scan code.

13.6 Summary

Block	Inputs	Outputs	Function
Decoder	n	2^n	Activates the one output named by the input code; generates all minterms.
Encoder	2^n	n	Outputs the binary index of the active input line.
Multiplexer	2^n data + n select	1	Routes the selected input to the output.
Demultiplexer	1 data + n select	2^n	Routes the input to the selected output.

The relationships in one line

Decoder and encoder are inverses. MUX and DEMUX are inverses. A MUX is a decoder plus AND/OR gating; a DEMUX *is* an enabled decoder. Learn the decoder properly and the other three follow.

PART IV

Sequential Logic Circuits

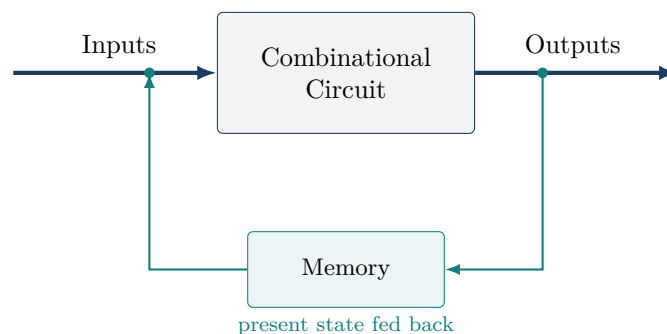
Sequential Logic Fundamentals

Where this fits

Combinational circuits have no memory — given the same inputs they always produce the same outputs. But a computer must *remember*: a counter must know what it counted to, a register must hold a value. Adding memory turns a combinational circuit into a sequential one, and introduces the new dimension of **time**. Sources: Lecture 2.4.1 and Tutorial 08, merged.

14.1 Combinational versus sequential

Combinational	Sequential
Outputs depend only on the immediate inputs.	Outputs depend on the immediate inputs <i>and</i> on previous outputs.
Purely logic gates.	Combinational logic <i>plus</i> memory.
No feedback paths, no memory elements.	Contains feedback and storage elements.
Fully described by a truth table.	Requires a state table or state diagram.
Examples: adders, decoders, multiplexers.	Examples: flip-flops, registers, counters.



14.2 State and sequence

State

The contents of the memory elements at a given moment. A circuit with k memory elements (each holding one bit) has 2^k possible states.

The defining relation

$$\text{next state} = f(\text{inputs}, \text{present state})$$

$$\text{outputs} = g(\text{inputs}, \text{present state})$$

A sequential circuit is therefore best understood as a **time sequence** of inputs, outputs and internal states, rather than as a single input–output mapping.

Notation used throughout Part IV:

$Q(t)$	the present state — what the memory holds now
$Q(t + 1)$	the next state — what it will hold after the next trigger

14.3 Asynchronous and synchronous circuits

Asynchronous (event driven)	Synchronous (clock driven)
State changes <i>immediately</i> after the inputs change.	State changes only at discrete instants in time.
No clock signal.	Uses a clock signal.
Faster — limited only by propagation delay.	Slower — limited by the clock period.
Difficult to design.	Easy to design.
Can become unstable.	Stable by construction.

Why asynchronous design is hard

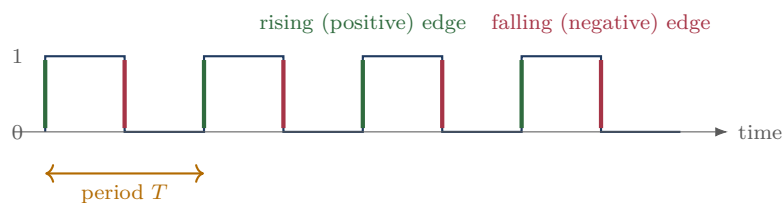
An asynchronous circuit is a combinational circuit with feedback. Because different signal paths have different propagation delays, an input change can race around the feedback loop and arrive at the wrong moment — producing a **race condition** in which the final state depends on which path happened to be faster. The circuit may even oscillate.

Synchronous design eliminates this entire class of problem by simply agreeing that nothing changes until the clock says so. That gives every signal time to settle. The price is speed — the clock period must be long enough for the slowest path.

14.4 The clock signal

Clock signal

A signal that oscillates repeatedly between logic 0 and logic 1 at a fixed rate. It controls when the memory elements of a sequential circuit are permitted to change their outputs.



Clock terminology

- **Positive (rising) edge** — the transition from 0 to 1.
- **Negative (falling) edge** — the transition from 1 to 0.
- **Period T** — the duration of one complete cycle.
- **Frequency $f = 1/T$** — cycles per second, in hertz.

The clock frequency and period reappear in Chapter 18 as clock rate and clock cycle time — the same quantities, viewed from the performance side.

14.5 Clocked synchronous sequential circuits

The design pattern

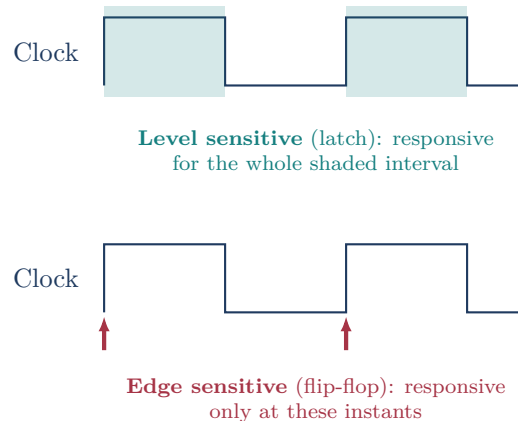
- A separate controller signal — the clock — provides synchronisation.
- The clock is connected to **all** storage elements in the circuit.
- The clock pulses determine **when** state changes occur.
- The other inputs determine **what** the change will be.
- Propagation delays and the clock interval are critical: the clock period must exceed the longest combinational delay between two storage elements.

The timing constraint sets the clock speed

If the logic between two flip-flops takes 12 ns to settle, the clock period cannot be shorter than 12 ns — so the maximum clock frequency is about 83 MHz. Making a processor faster means shortening the *longest* such path, which is called the *critical path*. This is why pipelining (Chapter 22) helps: splitting long logic into shorter stages permits a faster clock.

14.6 Level sensitive versus edge sensitive

This distinction is the single most important idea in this part.



Latch

A storage element that operates on signal **levels** rather than transitions — it is **level sensitive**. It is an asynchronous device, and latches are the building blocks from which flip-flops are made.

Flip-flop

A storage element used in clocked synchronous sequential circuits, whose state changes are triggered by a clock pulse **transition** — it is **edge sensitive**. It stores one bit of information.

The problem level sensitivity causes

Suppose a latch is enabled for the whole high phase of the clock. If its input changes twice during that phase, the output changes twice — so the circuit performs *two* state transitions in a single clock period. Feed such a latch's output back to its own input and it will oscillate for as long as the clock is high.

Flip-flops solve this by sampling only at the instant of an edge. Whatever happens between edges is ignored, so exactly one transition occurs per clock cycle no matter how noisy the input is.

Latch

Built from logic gates.
No clock signal.
Level sensitive.
Faster than a flip-flop.
Asynchronous.

Flip-flop

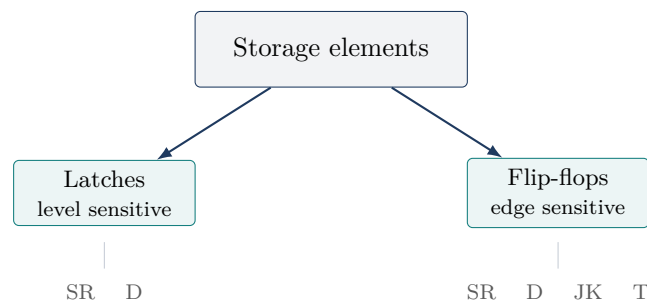
Built from latches plus a clock signal.
Has a clock signal.
Edge sensitive.
Slower than a latch.
Synchronous.

Clock signal versus enable signal

A **clock** makes a flip-flop act at its positive or negative edge (in special cases both). An **enable** makes a device function for as long as the signal is high, and stops it when the signal is low.

Both gate *when* the device is active, but the clock does so at an instant and the enable over an interval.

14.7 Storage element family



The four types — **SR**, **D**, **JK** and **T** — are covered one by one in Chapter 15.

14.8 Summary

Term	Meaning
Sequential circuit	Output depends on inputs <i>and</i> stored past outputs.
State	Contents of the memory elements; k bits give 2^k states.
$Q(t)$, $Q(t+1)$	Present state, next state.
Asynchronous	Changes immediately; fast but hard to design; can be unstable.
Synchronous	Changes only on the clock; slower but reliable.
Clock	Repeating 0/1 signal that paces state changes.
Rising / falling edge	$0 \rightarrow 1$ / $1 \rightarrow 0$ transition.
Latch	Level sensitive, no clock, building block of flip-flops.
Flip-flop	Edge sensitive, clocked, stores one bit.

Chapter 15

Latches and Flip-Flops

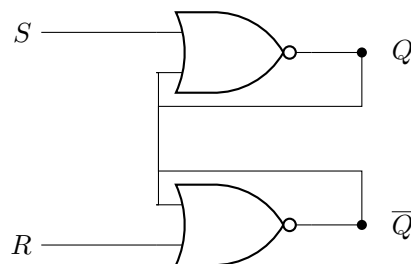
Where this fits

This chapter builds the one-bit memory cell from scratch — starting with two cross-coupled gates and progressively fixing each problem that appears, until we arrive at the edge-triggered flip-flop used in every real register. Sources: Lecture 2.4.1 and Tutorial 08, merged.

15.1 The S-R latch

The core trick: cross-coupling

Take two NOR gates and feed each one's output back into the other's input. The resulting loop has two stable configurations — and it will sit in whichever one it was last put into. That is memory, built from nothing but gates.



S	R	$Q(t+1)$
0	0	$Q(t)$ (hold)
0	1	0 (reset)
1	0	1 (set)
1	1	Forbidden

- $S = \mathbf{Set}$ — drives Q to 1.
- $R = \mathbf{Reset}$ — drives Q to 0.
- Two outputs, one the complement of the other.
- When $Q = 1$ the latch is in the **Set** state, otherwise in the **Reset** state.

Why $S = R = 1$ is forbidden

Setting both inputs high forces *both* NOR outputs to 0, so $Q = \bar{Q} = 0$ — a contradiction, since they are supposed to be complements.

Worse, the state that follows depends on which input falls back to 0 first. If they fall simultaneously the latch enters a **race** and settles unpredictably into 0 or 1. The result is not merely undefined; it is *non-deterministic*. This defect is what motivates the D latch in Section 15.3.

The NAND version

The same latch can be built from two cross-coupled **NAND** gates. The inputs then become **active low** — usually written $\bar{S}$ and $\bar{R}$ — and the forbidden combination becomes $\bar{S} = \bar{R} = 0$.

NOR latch (active high)			NAND latch (active low)		
S	R	$Q(t+1)$	$\bar{S}$	$\bar{R}$	$Q(t+1)$
0	0	hold	1	1	hold
0	1	0	1	0	0
1	0	1	0	1	1
1	1	forbidden	0	0	forbidden

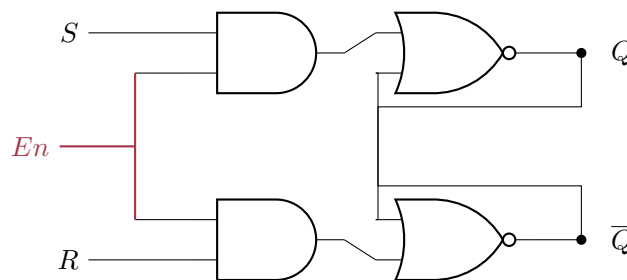
15.1.1 Characteristic equation

$$Q(t+1) = S + \bar{R} \cdot Q(t) \quad \text{subject to } S \cdot R = 0$$

Read it as: the output is 1 if we are setting it, *or* if it was already 1 and we are not resetting it.

15.2 S-R latch with a control signal

The plain latch responds to its inputs continuously. Adding an **Enable** gates that responsiveness.



En	S	R	$Q(t+1)$
0	×	×	$Q(t)$ (hold — inputs ignored)
1	0	0	$Q(t)$
1	0	1	0
1	1	0	1
1	1	1	Forbidden

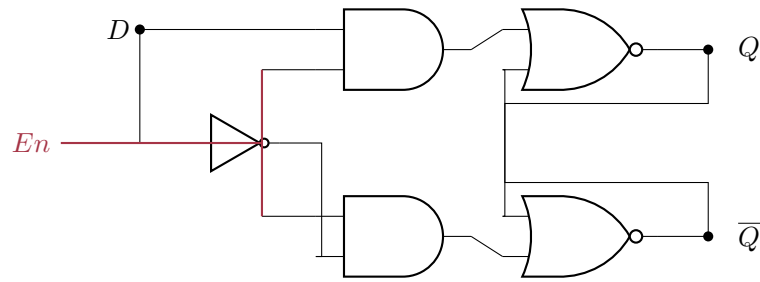
When $En = 0$ the AND gates force S and R to 0 at the latch, which is the hold condition. En must be asserted for the latch to be operational.

15.3 The D latch

Eliminating the forbidden state

The trouble with S-R is that S and R can both be 1. So: derive R from S with an inverter. Now they are *always* complementary and the forbidden combination is structurally impossible.

The single remaining input is called D (for **Data**).



En	D	$Q(t+1)$
0	$\times$	$Q(t)$
1	0	0
1	1	1

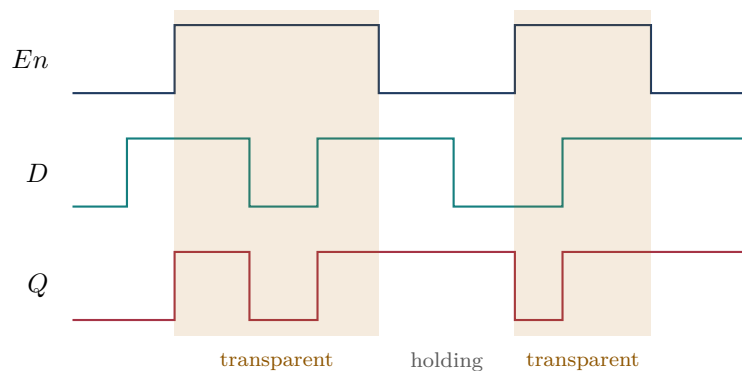
The value on D goes straight through to Q while the circuit is enabled; when disabled, Q keeps its last value.

$$Q(t+1) = D \quad (\text{when enabled})$$

Two nicknames

The D latch is also called a **transparent latch** — while enabled it is “transparent”, with the output simply following the input. And the value it holds after En drops is the value D had at that moment, which is why it is sometimes described as a *data* or *delay* latch.

15.3.1 Timing behaviour

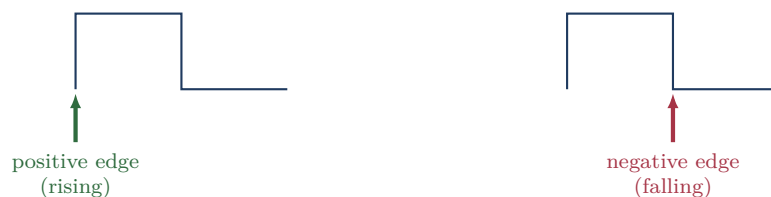


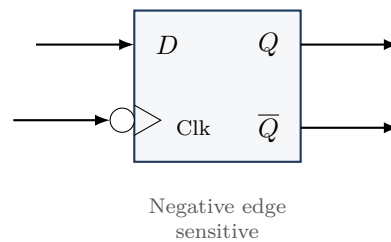
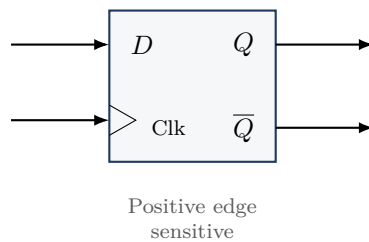
Notice that during a transparent window, Q tracks *every* change of D . Between $t = 1.5$ and $t = 4.5$ the output changes three times within one enable pulse — exactly the multiple-transition problem flagged in Chapter 14.

15.4 Edge triggering and the D flip-flop

Edge-triggered flip-flop

A storage device that samples its input only at the instant of a clock edge and holds that value for the rest of the cycle.





Reading flip-flop symbols

The small **triangle** on the clock input means “edge triggered”. An additional **bubble** (circle) in front of it means “negative edge”; no bubble means “positive edge”. This convention appears on every schematic you will meet.

Full behaviour

D	Clk	$Q(t+1)$
0	no edge	$Q(t)$
0	↑	0
1	no edge	$Q(t)$
1	↑	1

At the active edge

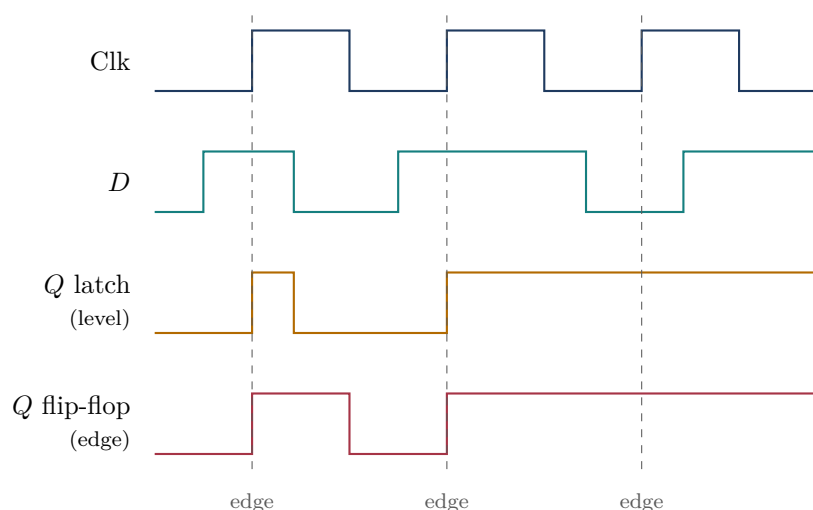
D	$Q(t+1)$
0	0
1	1

$$Q(t+1) = D$$

A slip in the tutorial's table

Tutorial 08 lists the D flip-flop as $\text{CLK}=1, D=0 \rightarrow Q(t+1) = Q(t)$. That is not right — with $D = 0$ at the active edge, Q must become 0. The correct statement is simply $Q(t+1) = D$ at the active edge, and $Q(t+1) = Q(t)$ at all other times.

15.4.1 Latch versus flip-flop timing, side by side

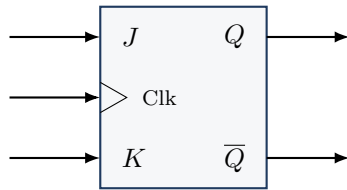


Between $t = 1.4$ and $t = 2.8$ the latch output follows D down again; the flip-flop holds the sampled value until the next edge. That is the whole difference, and it is why flip-flops, not latches, are used to build registers and counters.

15.5 The J-K flip-flop

Reclaiming the forbidden state

The D flip-flop removed the forbidden state by throwing away an input. The J-K flip-flop keeps both inputs and gives the previously forbidden combination a *useful* meaning: **toggle**.



J	K	$Q(t+1)$
0	0	$Q(t)$ hold
0	1	0 reset
1	0	1 set
1	1	$\overline{Q}(t)$ toggle

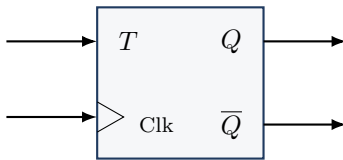
$$Q(t+1) = J \cdot \overline{Q}(t) + \overline{K} \cdot Q(t)$$

Why J-K is the most general flip-flop

Its four input combinations cover every possible one-bit behaviour: hold, force to 0, force to 1, and invert. Anything a D or T flip-flop can do, a J-K can do — which is why counters (Chapter 16) are usually drawn with J-K flip-flops.

15.6 The T flip-flop

Tie J and K together and call the common input T (for **T**oggle):



T	$Q(t+1)$
0	$Q(t)$ hold
1	$\overline{Q}(t)$ toggle

$$Q(t+1) = T \oplus Q(t)$$

The frequency divider

Hold T at 1 and the output flips on every clock edge — so Q completes one full cycle for every *two* clock cycles. A T flip-flop with $T = 1$ is a **divide-by-2** circuit, and chaining them gives divide-by-4, divide-by-8, and so on. That is precisely how the ripple counter in Chapter 16 works.

15.7 Characteristic and excitation tables

15.7.1 Characteristic tables — “given the inputs, what happens?”

Type	Characteristic equation	Behaviour
S-R	$Q(t+1) = S + \overline{R}Q$ (with $SR = 0$)	hold / reset / set
D	$Q(t+1) = D$	follow the input
J-K	$Q(t+1) = J\overline{Q} + \overline{K}Q$	hold / reset / set / toggle
T	$Q(t+1) = T \oplus Q$	hold / toggle

15.7.2 Excitation tables — “I want this transition; what inputs?”

Not in the slides, but essential for designing counters

When designing a sequential circuit you usually know the state transition you want and need to work out the inputs that produce it. That is what an **excitation table** gives — the characteristic table read backwards.

Transition		S-R		J-K		D	T
$Q(t)$	$Q(t+1)$	S	R	J	K	D	T
0	0	0	×	0	×	0	0
0	1	1	0	1	×	1	1
1	0	0	1	×	1	0	1
1	1	×	0	×	0	1	0

The × entries are don’t-cares, and they are what make J-K flip-flops convenient: half the table is don’t-care, so the K-maps that follow simplify dramatically.

Reading an entry. To go from $Q = 1$ to $Q = 0$ with a J-K flip-flop: K must be 1 (either reset or toggle will do it), and J can be anything — if $J = 0$ it resets, if $J = 1$ it toggles, and both give 0.

15.8 Summary

Device	Inputs	Problem it solves	Remaining issue
S-R latch	S, R	Stores a bit at all	Forbidden state $S=R=1$
S-R latch + En	S, R, En	Controls <i>when</i> it responds	Still has a forbidden state
D latch	D, En	Forbidden state impossible	Level sensitive — transparent
D flip-flop	D, Clk	Edge sensitive, one change per cycle	Cannot toggle directly
J-K flip-flop	J, K, Clk	Adds toggle; fully general	—
T flip-flop	T, Clk	Simplest toggling element	Cannot set/reset directly

The line of reasoning worth remembering

Cross-coupled gates give memory but a forbidden state → add an enable to control timing → derive R from S to kill the forbidden state (D latch) → replace level sensitivity with edge triggering (D flip-flop) → restore the second input with a useful meaning (J-K) → simplify to one input for counting (T).

Every step fixes exactly one defect of the previous one.

Registers, Counters and Memory

Where this fits

A flip-flop holds one bit. Group them and you get a register; chain them so each triggers the next and you get a counter; array them with a decoder and you get RAM. These three structures are where sequential logic becomes recognisable computer hardware. Sources: Lecture 2.4.1, extended.

16.1 Registers

Register

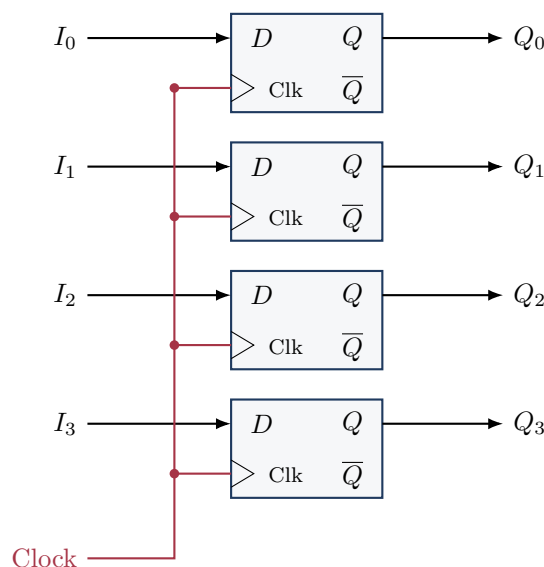
A series of flip-flops sharing a **common clock input**. An n -bit register consists of n flip-flops and stores an n -bit word. A 32-bit register contains 32 flip-flops.

A register is often combined with a combinational circuit to give it extra capability:

- the **flip-flops** hold the information;
- the **combinational circuit** processes or routes it.

16.1.1 Storage register

The simplest register: a group of D flip-flops with their clocks tied together.



Behaviour

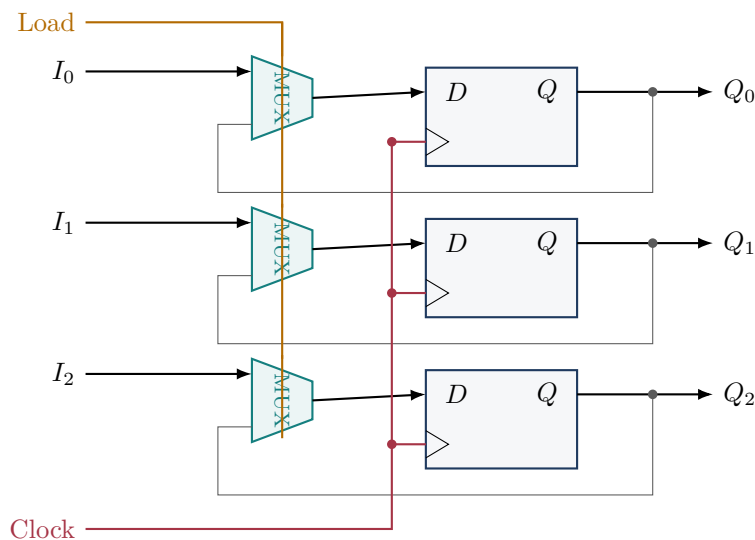
- On the active clock edge, every flip-flop simultaneously captures its I_i input.
- The stored word can be read from the Q outputs at any time — reading is non-destructive and needs no clock.
- A separate **asynchronous clear** input is often provided to force all flip-flops to 0 immediately, without waiting for a clock edge.

The problem with this design

It loads a *new value on every single clock edge*. There is no way to say “hold what you have”. Since the clock runs continuously, this register cannot actually store anything for more than one cycle.

16.1.2 Register with parallel load

The fix is a **Load** control line that selects, per flip-flop, between the new input and the current output.



Load	Effect at the next clock edge
1	each flip-flop loads I_i — new data written
0	each flip-flop reloads its own Q_i — value preserved

Note the combination of ideas

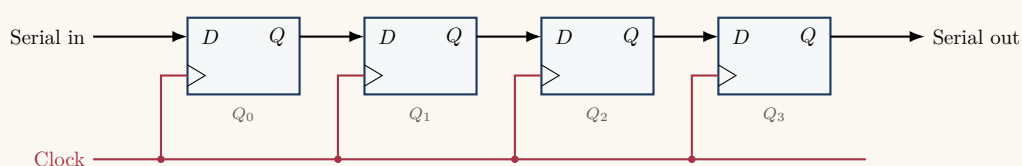
Here a *combinational* circuit (the multiplexer, Chapter 13) and a *sequential* one (the flip-flop) are combined to give behaviour neither has alone. This is the standard pattern for all real datapath hardware.

16.2 Shift registers

Left as “explore” in Lecture 2.4.1

Shift register

A register in which each flip-flop’s output feeds the *next* flip-flop’s input, so that on every clock edge the whole stored pattern moves one position along.



Example. Loading 1011 serially, most significant bit first, into a 4-bit shift register:

Clock	Serial in	Q_0	Q_1	Q_2	Q_3
initial	—	0	0	0	0
1	1	1	0	0	0
2	0	0	1	0	0
3	1	1	0	1	0
4	1	1	1	0	1

After four clock pulses the register holds 1011. It took four cycles — that is the cost of serial loading, and the reason parallel load exists.

Four configurations.

Type	In / Out	Use
SISO	serial / serial	Delay line; each bit emerges n cycles later.
SIPO	serial / parallel	Serial-to-parallel conversion — receiving data one bit at a time over a wire and presenting it as a word.
PISO	parallel / serial	Parallel-to-serial conversion — transmitting a word one bit at a time.
PIPO	parallel / parallel	An ordinary storage register.

Shifting as arithmetic. Shifting a binary number left by one position multiplies it by 2; shifting right divides it by 2 (discarding the remainder).

$$0011_2 (3) \xrightarrow{\text{shift left}} 0110_2 (6) \xrightarrow{\text{shift left}} 1100_2 (12)$$

This is why the « and » operators are in the ALU's repertoire (Section 12.8), and why compilers replace multiplication by a power of two with a shift — it is far cheaper than a multiplier.

Arithmetic versus logical right shift. A *logical* right shift brings in zeros; an *arithmetic* right shift replicates the sign bit, so that dividing a negative two's complement number by 2 stays negative:

$$11110000_2 (-16) \xrightarrow{\text{arithmetic}\gg} 11111000_2 (-8)$$

16.3 Counters

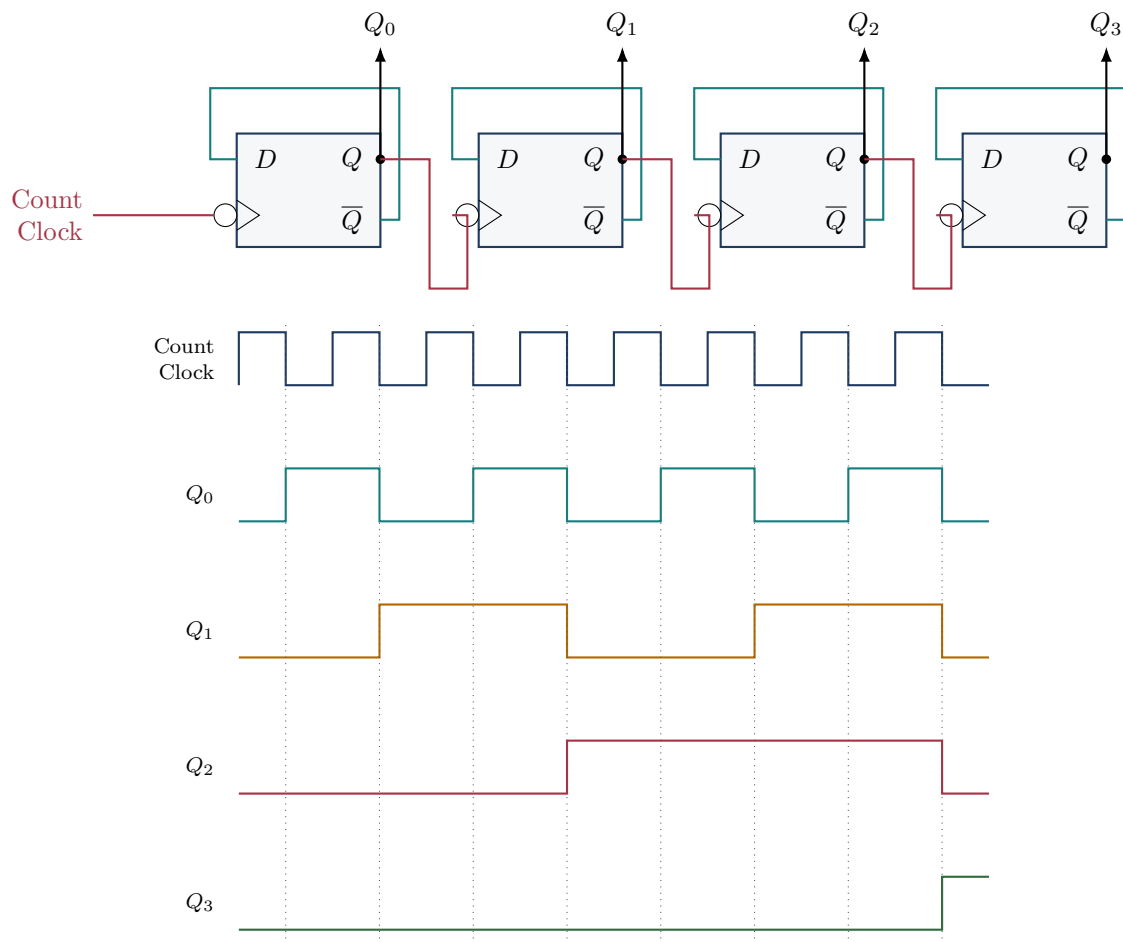
Counter

An ordered set of flip-flops that produces a defined sequence of states in response to a series of triggers.

Ripple (asynchronous) counter	Synchronous counter
Each flip-flop is triggered not by the clock but by the <i>output transition of the previous flip-flop</i> .	All flip-flops share a common clock and change simultaneously.
Simple — almost no extra logic.	Needs combinational logic on the inputs.
Slow — the change ripples through the chain.	Fast — one flip-flop delay total.
Transient false states occur during the ripple.	No transient states.

16.3.1 Binary ripple counter

Built from negative-edge-triggered D flip-flops with each $\bar{Q}$ fed back to its own D input — making each flip-flop toggle on every trigger it receives.



Pulse	Q_3	Q_2	Q_1	Q_0	Decimal
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	2
3	0	0	1	1	3
4	0	1	0	0	4
5	0	1	0	1	5
6	0	1	1	0	6
7	0	1	1	1	7
8	1	0	0	0	8

Why the pattern is exactly binary counting

Q_0 toggles on every clock pulse — period 2. Q_1 toggles on every falling edge of Q_0 — period 4. Q_2 toggles on every falling edge of Q_1 — period 8.

Each stage divides the frequency by two, which reproduces the place values of binary. A ripple counter is literally a chain of divide-by-2 circuits.

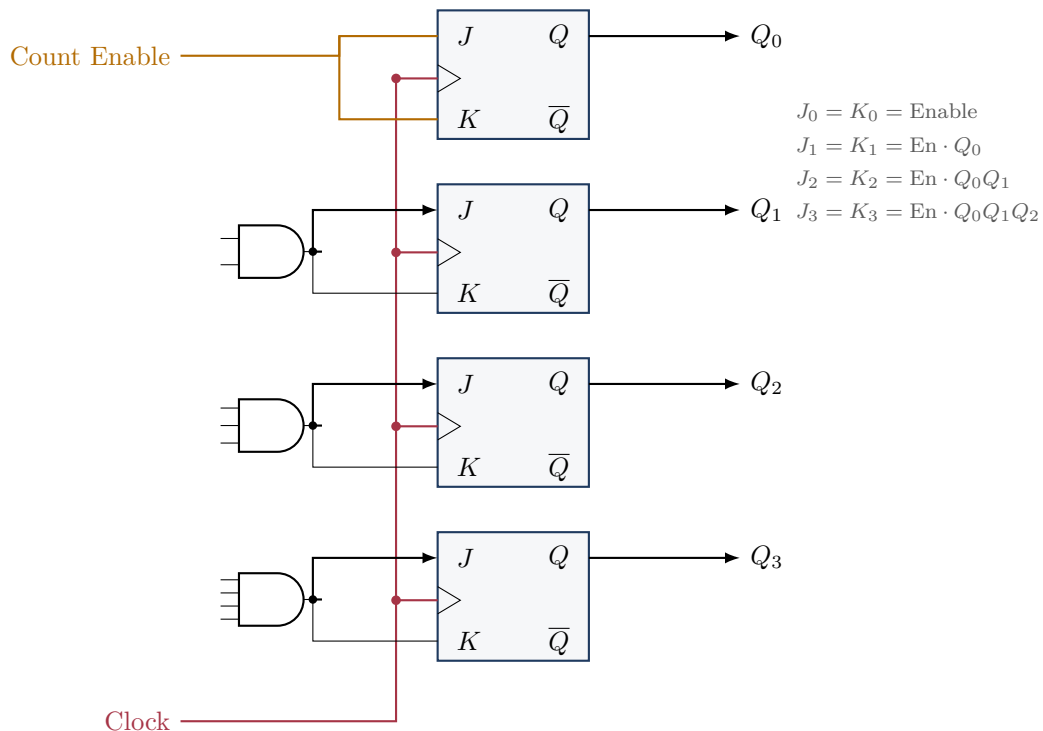
The ripple problem

The stages do *not* change simultaneously. Going from 0111 to 1000, the transitions arrive one after another and the counter briefly displays $0110 \rightarrow 0100 \rightarrow 0000 \rightarrow 1000$. Those intermediate values are real voltages on real wires. Any circuit that samples the counter during the ripple can read a completely wrong value.

The total settling time is $n \times t_{pd}$ for n stages, which caps the maximum counting frequency. Synchronous counters fix both problems.

16.3.2 Synchronous binary counter

All flip-flops receive the *same* clock; combinational logic decides which ones should toggle.



The design rule

Flip-flop i should toggle exactly when *all* lower-order bits are 1 — because that is precisely when adding 1 carries into position i :

$$J_i = K_i = \text{Enable} \cdot Q_0 \cdot Q_1 \cdots Q_{i-1}$$

The AND chain is the carry-propagation logic of an incrementer.

Worked Example 1 *Tracing the first clock pulse*

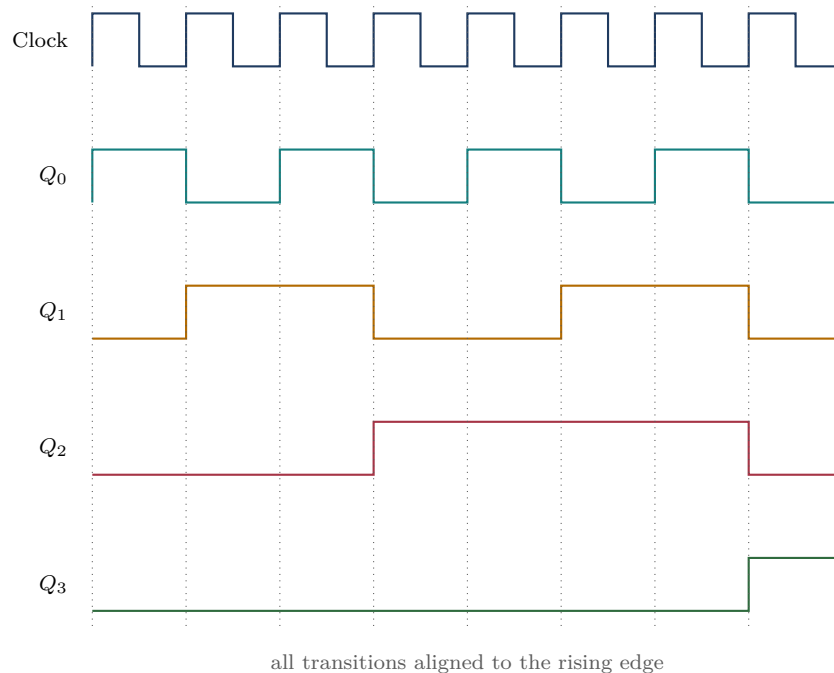
Set-up. $Q_3Q_2Q_1Q_0 = 0000$; positive-edge-triggered J-K flip-flops; Count Enable = 1; a common clock to all four.

At the first positive edge:

- Flip-flop 0: $J = K = 1$ (Enable) $\Rightarrow$ toggles.
- Flip-flop 1: $J = K = \text{En} \cdot Q_0 = 1 \cdot 0 = 0 \Rightarrow$ holds.
- Flip-flop 2: $J = K = \text{En} \cdot Q_0Q_1 = 0 \Rightarrow$ holds.
- Flip-flop 3: $J = K = 0 \Rightarrow$ holds.

Effect. Only Q_0 changes: $0000 \rightarrow 0001$. Nothing changes again until the next positive edge.

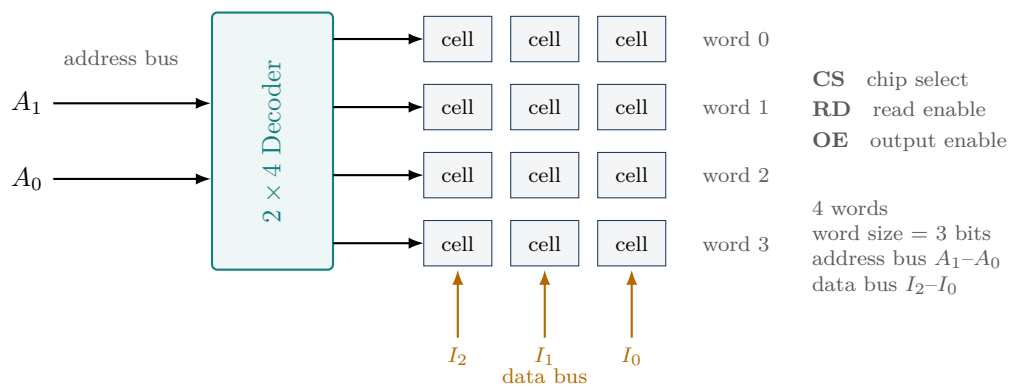
At the second edge: now $Q_0 = 1$, so flip-flop 1 sees $J = K = 1$ and toggles as well, while Q_0 toggles back to 0: $0001 \rightarrow 0010$. ✓



Compare with the ripple counter's timing diagram: here every edge is aligned to the clock, so there are no transient states.

16.4 Random Access Memory

Registers and decoders combine to give addressable memory.



Recognising the parts

A RAM chip is built entirely from components already covered:

- a **decoder** converts the address into a one-hot word-select line (Chapter 13);
- each **cell** is a latch or flip-flop storing one bit (Chapter 15);
- **multiplexers** / tri-state buffers route the selected word onto the data bus.

An n -bit address bus selects among 2^n words — the same formula as Chapter 1. Here $n = 2$ gives 4 words.

Signal	Purpose
Address bus (A_1-A_0)	Selects which word is accessed
Data bus (I_2-I_0)	Carries data in or out
CS (chip select)	Activates this chip; lets several chips share one bus
RD (read enable)	Selects read rather than write
OE (output enable)	Connects the outputs to the bus

Which technology is used where

The cells in a RAM chip are built from one of two technologies — **SRAM** (a latch per bit, fast but bulky) or **DRAM** (one transistor and a capacitor, dense but needing periodic refresh). SRAM is used for cache, DRAM for main memory.

Both are covered properly in Section 23.2.3, alongside ROM, flash and the rest of the memory hierarchy.

16.5 Summary

Structure	Key point
Register	n flip-flops on a common clock; stores an n -bit word.
Parallel load	A MUX per bit chooses between new data and the current value.
Shift register	Each output feeds the next input; SISO/SIPO/PISO/PIPO.
Shift as arithmetic	Left shift $\times 2$; right shift $\div 2$.
Ripple counter	Each stage clocks the next; simple but has transient states.
Synchronous counter	Common clock, AND-chain enables; no transients.
Counter rule	Bit i toggles when all lower bits are 1.
RAM	Decoder + storage cells + multiplexers; n address bits give 2^n words.

PART V

Computer Organisation and Architecture

Chapter 17

CPU Organisation

Where this fits

Parts II–IV built gates, then adders and multiplexers, then registers and counters. Assembled at the next level up, those components become a processor. This chapter is the architectural overview; the following chapters cover performance, the execution cycle, and the instruction set. Source: Lecture 3.1.

17.1 The Central Processing Unit

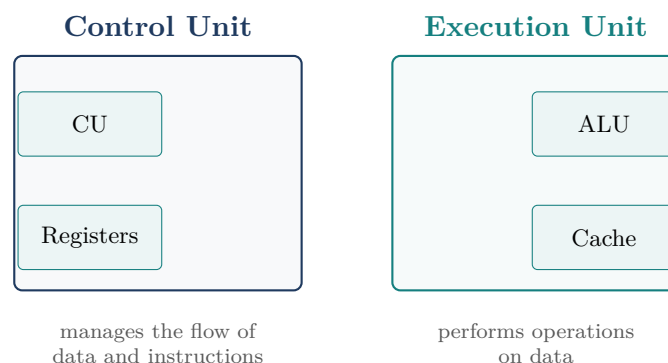
CPU

The **Central Processing Unit** executes instructions and performs arithmetic and logic operations — the “brain” of the computer.

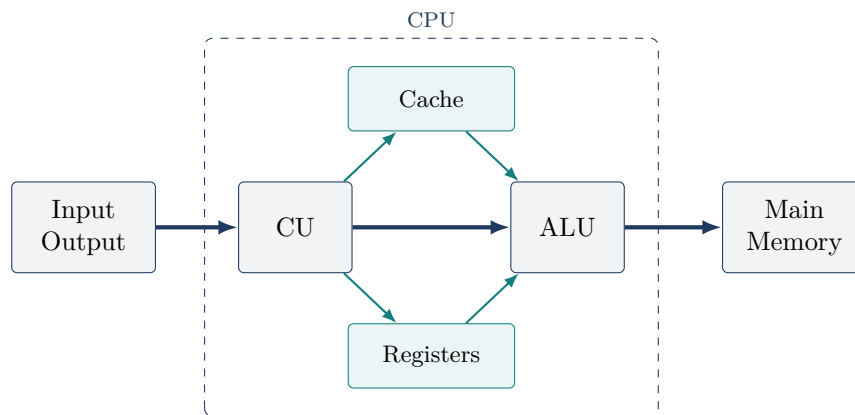
17.1.1 Key components

Component	Role
Arithmetic Logic Unit (ALU)	Performs the actual calculations and logical operations.
Control Unit (CU)	Fetches and decodes instructions; directs everything else.
Registers	Small, very fast storage inside the CPU.
Cache	Fast intermediate memory holding recently used data and instructions.
I/O interfaces	Connect the CPU to the rest of the system.

17.1.2 Two conceptual halves



17.1.3 Hardware block diagram



17.2 The Control Unit

Responsibilities

- Fetch instructions from main memory.
- Decode instructions.
- Determine the appropriate operation for the ALU to perform.
- Control the flow of data within the CPU.

The control unit computes nothing itself. It is a sequential circuit (Part IV) that generates the correct sequence of control signals — register enables, multiplexer selects, ALU operation codes — to make the rest of the hardware carry out each instruction.

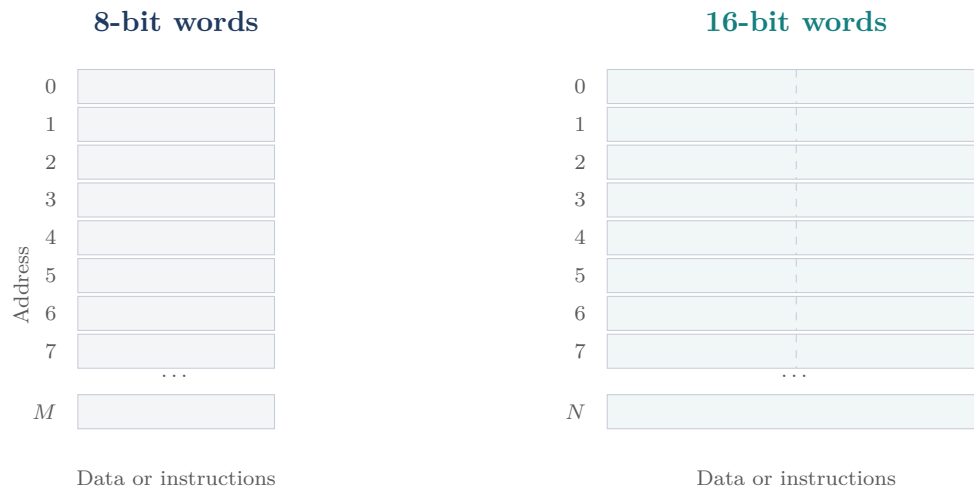
17.3 The Arithmetic and Logic Unit

The ALU performs the mathematical and logical operations specified by instructions, on binary numbers. Its internal construction was covered in Chapter 12.

17.4 Main memory

Structure

- A series of binary storage cells.
- The smallest unit is a **bit**.
- Bits are stored in groups; each group is a **word**.
- A word is a fixed-size group of bits.
- Each word has a **unique address** that identifies it.



Memory attributes to keep straight

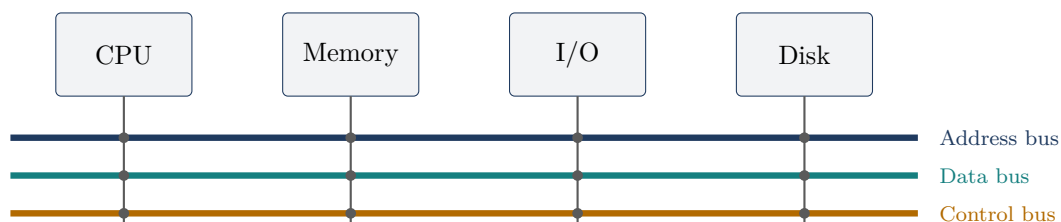
- **Address length** — how many bits an address has; determines the address range.
- **Word size** — how many bits one word holds.
- **Addressability** — word addressable or byte addressable.

The sizing formulas and worked examples are in Section 1.4; they are examined regularly, so review them.

17.5 Buses

Bus

A set of wires connecting multiple subsystems to form a shared data path, allowing parallel movement of data.



Bus	Purpose
Address bus	The lines the CPU uses to send the memory address of the data it wants to read or write. The memory controller uses this address to locate the correct memory location.
Data bus	The lines the CPU uses to send and receive data. On a read, the CPU puts an address on the address bus and the memory controller replies with data on the data bus. On a write, the CPU puts both the address and the data out.
Control bus	Carries control signals — memory read/write commands, interrupt requests, and so on. It also lets the CPU control the flow of data on the other two buses.

Bus performance

- At any moment only **one** device may use the bus — so sharing creates a **bottleneck**.
- Bus speed depends on the **width** of the bus (how many bits move in parallel) and the **number of devices** sharing it.

Address bus width sets the memory limit

The address bus width is exactly the “address length” of Section 1.4. A 32-bit address bus can address 2^{32} locations; if the memory is byte addressable that is 4 GB — which is precisely why 32-bit operating systems could not use more than 4 GB of RAM without special tricks.

17.6 The memory controller

Memory controller

The unit that manages the flow of data to and from main memory. It coordinates communication between the CPU and the memory modules and handles tasks such as error checking and correction. It is typically integrated into the CPU or into the motherboard chipset.

17.7 The clock

What the clock does at system level

- The clock generates pulses (ticks).
- A pulse sets the pace for everything that happens in the computer.
- It therefore determines how quickly instructions are executed.
- A **fixed number of clock pulses** is required to complete each instruction.
- Clock frequency is measured in hertz (Hz).

This is the same clock signal as in Chapter 14, now viewed from the system level. Chapter 18 turns it into performance arithmetic.

17.8 Summary

Component	Role
Control Unit	Fetches and decodes instructions; issues control signals.
ALU	Performs arithmetic and logic on binary data.
Registers	Fast storage inside the CPU for data, addresses and instructions.
Cache	Fast buffer between registers and main memory.
Main memory	Addressable array of words holding data and instructions.
Address bus	Carries the address; its width caps addressable memory.
Data bus	Carries the data being read or written.
Control bus	Carries read/write, interrupt and other control signals.
Memory controller	Coordinates CPU–memory communication.
Clock	Paces all activity; frequency determines execution rate.

Chapter 18

CPU Performance

Where this fits

“Which machine is faster?” has a precise answer, and it is not simply “the one with the higher clock speed”. This chapter develops the CPU time equation and works through every calculation type in Lecture 3.1 — the most computational chapter in Part V, and a reliable source of exam questions.

18.1 Time and frequency units

Duration		Frequency	
1 second		1 cycle/s	= 1 hertz (Hz)
$\frac{1}{1000}$ s	= 1 millisecond (ms)	10^6 cycles/s	= 1 megahertz (MHz)
$\frac{1}{1\,000\,000}$ s	= 1 microsecond (μ s)	10^9 cycles/s	= 1 gigahertz (GHz)
$\frac{1}{1\,000\,000\,000}$ s	= 1 nanosecond (ns)		

Decimal here, binary for storage

Note the inconsistency: frequency prefixes are powers of *ten* (1 MHz = 10^6 Hz) while storage prefixes in Chapter 1 were powers of *two* (1 MB = 2^{20} bytes). This is genuinely how the conventions differ — do not carry 1024 into a frequency calculation.

18.2 Clock speed and clock cycle time

Clock speed / clock rate

The number of clock pulses per second. One million pulses per second is 1 MHz.

Clock cycle time

The period of the clock that synchronises the processor’s circuits — the reciprocal of the clock frequency.

$$\text{Clock Cycle Time} = \frac{1}{\text{Clock Speed}} \qquad \text{Clock Speed} = \frac{1}{\text{Clock Cycle Time}}$$

Worked Example 1 *Cycle time from clock speed*

(a) A CPU with a 1 MHz clock:

$$t = \frac{1}{10^6} \text{ s} = 10^{-6} \text{ s} = 1 \mu\text{s}$$

(b) A 750 MHz CPU:

$$t = \frac{1}{750\,000\,000} \text{ s} = 1.333 \times 10^{-9} \text{ s} \approx 1.33 \text{ ns}$$

(c) A 2.5 GHz clock:

$$t = \frac{1}{2.5 \times 10^9} \text{ s} = 0.4 \times 10^{-9} \text{ s} = 0.4 \text{ ns}$$

Worked Example 2 Clock rate from cycle time

A clock cycle time of 20 ns:

$$f = \frac{1}{20 \times 10^{-9}} \text{ Hz} = \frac{1}{2 \times 10^{-8}} \text{ Hz} = 0.5 \times 10^8 \text{ Hz} = 5 \times 10^7 \text{ Hz} = \mathbf{50 \text{ MHz}}$$

Slide erratum

Lecture 3.1 gives this answer as “ $0.5 \times 10^8 \text{ Hz} = 500 \text{ MHz}$ ”. The arithmetic $0.5 \times 10^8 = 5 \times 10^7$ is right, but $5 \times 10^7 \text{ Hz}$ is 50 MHz, not 500 MHz. Sanity check: 1 ns $\leftrightarrow$ 1 GHz, so 20 ns must be twenty times slower, i.e. 50 MHz.

18.3 CPU execution time

The starting point

$$T = (\text{Number of Clock Cycles}) \times (\text{Clock Cycle Time}) = \frac{\text{Number of Clock Cycles}}{\text{Clock Rate}}$$

18.3.1 Bringing in the instruction count

Does execution time depend on the number of instructions? Yes — but not simply, because **different instructions take different numbers of cycles**. The bridge is:

$$\text{CPU Clock Cycles} = N \times \text{CPI}$$

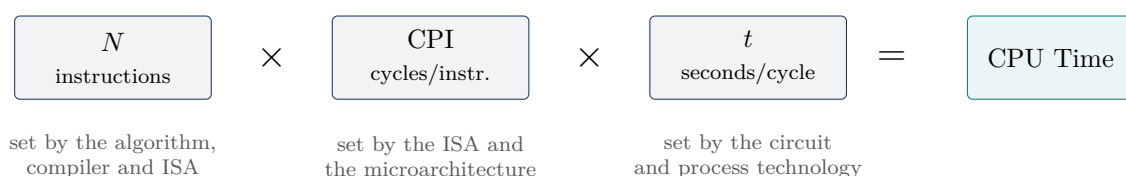
- N — the number of instructions executed;
- CPI — the *average* clock cycles per instruction.

The CPU time equation

$$\text{CPU Time} = N \times \text{CPI} \times t$$

$$\text{CPU Time} = \frac{N \times \text{CPI}}{r}$$

N	number of instructions in the program
CPI	average cycles per instruction
t	clock cycle time
r	clock rate ($r = 1/t$)



Why clock speed alone is meaningless

All three factors matter, and they trade off against each other. A higher clock rate (smaller t) is worthless if the design achieves it by raising CPI. This is why comparing two processors by GHz alone — the “megahertz myth” — is unreliable.

18.4 Cycles per instruction

CPI

The average number of clock cycles per instruction. If instruction type i has frequency F_i (as a fraction of all instructions executed) and takes CPI_i cycles, then

$$\text{Overall CPI} = \sum_i \text{CPI}_i \times F_i$$

Worked Example 3 Computing CPI — Lecture 3.1

A benchmark program has 100 instructions:

- 25 are load/store operations, each taking 2 cycles;
- 50 are add operations, each taking 1 cycle;
- 25 are square root operations, each taking 50 cycles.

Convert the counts to fractions and apply the formula:

$$\text{CPI} = (0.25 \times 2) + (0.50 \times 1) + (0.25 \times 50)$$

$$\text{CPI} = 0.5 + 0.5 + 12.5 = \mathbf{13.5}$$

Cross-check by total cycles:

$$\frac{25(2) + 50(1) + 25(50)}{100} = \frac{50 + 50 + 1250}{100} = \frac{1350}{100} = 13.5 \quad \checkmark$$

What this example really shows

The square roots are only a quarter of the instructions but consume $1250/1350 \approx 93\%$ of the cycles. A weighted average is dominated by rare, expensive operations — which is why profiling before optimising matters, and why an average alone can be misleading.

18.5 Comparing two machines

Worked Example 4 Which machine is faster? — Lecture 3.1

Two machines implement the same ISA, running the same program P :

	Machine A	Machine B
Clock cycle time	10 ns	20 ns
CPI for P	2.0	1.2

Since both run the *same* program on the *same* ISA, N is identical and cancels.

$$\text{Cycles for A} = 2.0 \times N$$

$$\text{Cycles for B} = 1.2 \times N$$

$$\text{CPU Time for A} = 2N \times 10 \text{ ns} = 20N \text{ ns}$$

$$\text{CPU Time for B} = 1.2N \times 20 \text{ ns} = 24N \text{ ns}$$

$$\boxed{\text{Machine A is faster}} \quad \text{by a factor } \frac{24N}{20N} = 1.2\times$$

Note that B has the *better* CPI but the *worse* overall time — its slower clock more than cancels the advantage.

Worked Example 5 *Homework from Lecture 3.1*

	Machine A	Machine B
Clock cycle time	10 ns	30 ns
CPI	2.0	0.5

$$\text{CPU Time for A} = N \times 2.0 \times 10 \text{ ns} = 20N \text{ ns}$$

$$\text{CPU Time for B} = N \times 0.5 \times 30 \text{ ns} = 15N \text{ ns}$$

$$\boxed{\text{Machine B is faster}} \quad \frac{20N}{15N} \approx 1.33\times$$

Here the CPI advantage *does* outweigh the slower clock. There is no shortcut — you must multiply the numbers out.

A CPI below 1 is possible

Machine B's CPI of 0.5 means it completes two instructions per cycle on average. That requires a *superscalar* design with multiple execution units working in parallel. Nothing about the equation forbids $\text{CPI} < 1$.

18.6 Instruction count: static versus dynamic

The distinction

The number of instructions the CPU *executes* is not the number of instructions *written* in the program.

Static count — instructions present in the program text. Determines program size.

Dynamic count — instructions actually executed at run time. This is the N in the CPU time equation.

Loops make the dynamic count enormously larger than the static count.

Worked Example 6 *Counting instructions in nested loops — Lecture 3.1*

```

250 instructions
for i = 1, 100 do
  20 instructions
  for j = 1, 100 do
    40 instructions
    for k = 1, 100 do
      10 instructions
    endfor
  endfor
endfor

```

Assume each **for** statement costs **two** instructions per iteration: one to increment the index and one to test the exit condition.

Work from the inside out.

Innermost loop (k). Each iteration runs the 10 body instructions plus the loop's own 2:

$$(2 + 10) \times 100 = \mathbf{1,200}$$

Middle loop (j). Each iteration runs its own 2, its 40 body instructions, and the entire inner loop:

$$(2 + 40 + 1,200) \times 100 = 1,242 \times 100 = \mathbf{124,200}$$

Outer loop (i). Each iteration runs its own 2, its 20 body instructions, and the whole middle loop:

$$(2 + 20 + 124,200) \times 100 = 124,222 \times 100 = \mathbf{12,422,200}$$

Whole program. Add the 250 straight-line instructions at the start:

$$250 + 12,422,200 = \mathbf{12,422,450}$$

Static count — just add up what is written:

$$250 + 20 + 40 + 10 + \underbrace{(3 \times 2)}_{\text{three for statements}} = \mathbf{326}$$

Static instruction count	326
Dynamic instruction count	12,422,450
Ratio	$\approx 38,000\times$

A program you can print on one page executes over twelve million instructions.

The optimisation lesson

Removing one instruction from the innermost loop saves 10^6 executed instructions. Removing one from the straight-line prologue saves exactly one. Optimisation effort belongs in the innermost loop — which is the practical content of “profile before you optimise”.

18.7 Comparing code sequences

Worked Example 7 *Choosing between two code sequences — Lecture 3.1*

A compiler writer must choose between two code sequences for the same high-level statement.

Instruction count per type

Sequence	A	B	C
1	2	1	2
2	4	1	1

CPI from the hardware spec

Instruction type	CPI
A	1
B	2
C	3

	Sequence 1	Sequence 2
Instruction count	$2 + 1 + 2 = 5$	$4 + 1 + 1 = 6$
Total cycles	$2(1) + 1(2) + 2(3) = 10$	$4(1) + 1(2) + 1(3) = 9$
CPI	$10/5 = 2.0$	$9/6 = 1.5$

Answers to the three questions:

1. *Which executes more instructions?* Sequence 2 (6 versus 5).
2. *Which is faster?* **Sequence 2** — 9 cycles versus 10. (Both run on the same machine, so the clock cycle time is the same and cycles alone decide.)
3. *Which has the lower CPI?* **Sequence 2** — 1.5 versus 2.0.

More instructions can still be faster

Sequence 2 executes *more* instructions yet finishes sooner, because its instructions are cheaper on average. Instruction count alone is not a performance metric — only the product $N \times \text{CPI} \times t$ is.

This is exactly the RISC design argument: use more, simpler instructions with a low CPI and a fast clock, rather than fewer complex ones.

18.8 What limits throughput

The three factors, and what moves them

Factor	Influenced by
Number of instructions	Algorithm, programming language, compiler, ISA
Cycles per instruction	ISA, microarchitecture, memory system
Clock cycle time	Circuit design, critical path length, process technology

Common limitation	Why
Multiplication > addition	An n -bit multiplier is roughly n adders deep (Section 12.7).
Floating point > integer	Requires exponent alignment, normalisation and rounding (Chapter 5).
Main memory > registers	Registers are on-chip; memory is off-chip across a shared bus.

The memory wall

The gap between the last two rows is the most severe in modern systems. A register access costs well under one cycle; a main memory access can cost hundreds. This is what caches exist to hide, and why a cache miss can dominate a program's execution time even though it never appears in the instruction count.

18.9 Summary

Quantity	Formula
Clock cycle time	$t = 1/r$
Clock rate	$r = 1/t$
Total cycles	$N \times \text{CPI}$
CPU time	$N \times \text{CPI} \times t = (N \times \text{CPI})/r$
Overall CPI	$\sum_i \text{CPI}_i \times F_i$
CPI from totals	(total cycles) / (total instructions)

Exam technique

1. Write down which of N , CPI, t you are given and which cancel.
2. If the two machines run the same program on the same ISA, N cancels.
3. Convert everything to consistent units *before* multiplying.
4. Compute both times fully, then compare — never compare a single factor.
5. Sanity-check: does the ratio look plausible?

Von Neumann Architecture and the FDE Cycle

Where this fits

This chapter covers the single most consequential idea in computing — the stored program — and the repeating cycle by which a processor carries out that program. Source: Lecture 3.2.

19.1 Computer architecture

Computer architecture

The design of a computer system: its hardware components and their organisation, and the way those components interact with software. It encompasses the CPU, memory, I/O devices and buses.

19.1.1 Notable early architectures

Machine	Era	Significance
Harvard architecture	1930s	Separate memory spaces for instructions and data.
Atanasoff–Berry (ABC)	late 1930s	Built by John Atanasoff and Clifford Berry in the late 1930s and early 1940s; used a special binary system.
Colossus	WWII	Used by the British to break German codes; vacuum tubes, switches and other components.
ENIAC	mid-1940s	Vacuum tubes; huge space and power requirements; programmed manually by setting switches and plugging cables.
EDVAC	1945	Introduced the stored program concept.

19.2 The stored program concept

The problem it solved

Programming the earlier machines was extremely laborious. ENIAC was reprogrammed by physically rewiring it — setting switches, plugging and unplugging cables. A change of program meant a change of hardware, taking days.

Scientists therefore explored mechanisms to represent the program in a **stored form, alongside the data**. The processing unit could then obtain its instructions by *reading* them from the stored program, so a program could be altered and set easily. John von Neumann proposed this in 1945, in “First Draft of a Report on the EDVAC”.

Stored program concept

A computer program is a series of instructions telling the computer what to do. The program is stored in the **same memory** as the data it operates on. The processor reads the instructions from memory, interprets them, and executes them one at a time, repeating until the program's task is complete.

Memory



Bus →



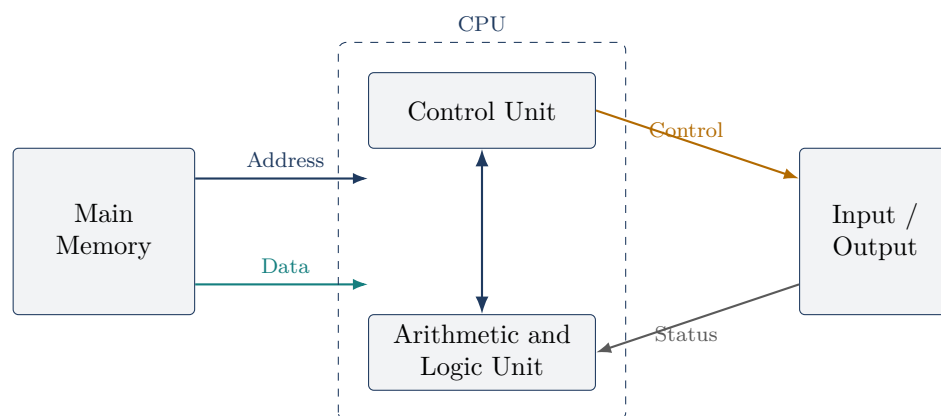
- Program and data share one memory.
- The CPU reads instructions, interprets them, and executes them one at a time.
- Repeated until the program completes.

Why it mattered so much

A computer can be programmed for a wide range of tasks simply by **changing the program stored in memory** — far more flexible and versatile than earlier machines that had to be physically rewired. This is the foundation of modern computer architecture, and it enabled the entire software industry.

It also has a subtler consequence: since programs are just data, a program can *read and write other programs*. Compilers, assemblers, linkers, operating systems and interpreters all depend on this.

19.3 Von Neumann architecture



19.3.1 The six defining principles

1. **The CPU** performs arithmetic and logic operations and controls the flow of data between memory and I/O devices. It consists of the **ALU** and the **Control Unit**.
2. **Memory stores both data and instructions in a single address space**, allowing the computer to execute instructions and access data in a unified way.
3. Execution follows a **fetch–decode–execute cycle**: the CPU fetches an instruction from memory, decodes it, executes it, then stores the result back.

4. A single **Instruction Set Architecture (ISA)** defines what the CPU can understand and execute, and it operates on both data and instructions stored in memory.
5. The **stored program concept** means programs can be written and modified without physically changing the hardware.
6. The **I/O system** lets the computer communicate with external devices — keyboards, displays, disk drives.

19.3.2 Components of a von Neumann machine

CPU	Memory	Input/Output	Bus
Registers	Instruction set		

Von Neumann versus Harvard

Von Neumann	Harvard
One memory for instructions and data.	Separate memories for instructions and data.
One bus — instruction fetch and data access must take turns.	Separate buses — can fetch an instruction and read data simultaneously.
Simpler and cheaper.	Faster, but more hardware.
Programs can modify themselves; loaders and compilers are straightforward.	Instruction memory is typically read-only at run time.

The von Neumann bottleneck. Because instructions and data share one bus, the CPU cannot fetch the next instruction while reading data for the current one. That shared path limits throughput no matter how fast the CPU is.

Modern reality: a hybrid. Most processors today are von Neumann at the main memory level but Harvard inside — separate L1 instruction and data caches allow simultaneous access, recovering most of the Harvard advantage while keeping the single address space that software expects.

19.4 Registers

Register

A piece of hardware, located on the CPU, that can store binary information. It offers far faster access than main memory and can hold instructions, data or addresses.

19.4.1 Key registers

Abbr.	Name	Purpose
MAR	Memory Address Register	Holds the memory location of the data or instruction that needs to be accessed.
MDR	Memory Data Register	Holds data being transferred to or from memory.
AC	Accumulator	Where intermediate arithmetic and logic results are stored.
PC	Program Counter	Contains the address of the <i>next</i> instruction to be executed.
CIR / IR	Current Instruction Register	Contains the current instruction during processing.

How MAR and MDR work together

Every memory access uses both: the MAR says *where*, the MDR carries *what*. The MAR connects to the address bus, the MDR to the data bus — which is why the MAR is as wide as an address and the MDR as wide as a word.

19.4.2 The register sequence during execution

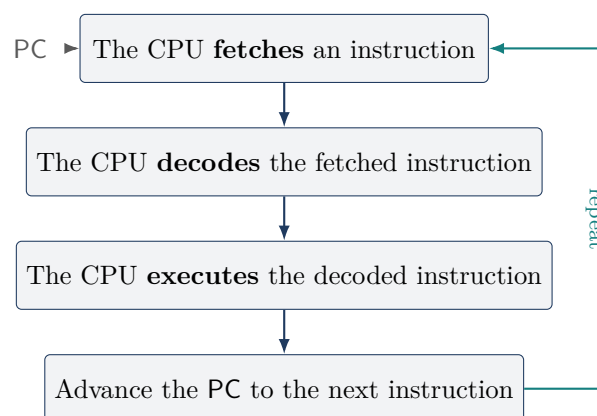
When a program is executed:

1. The CPU starts fetching the next instruction from memory using the PC.
2. The address of the instruction is loaded into the MAR.
3. The IR receives the instruction pointed to by the address in the MAR.
4. The CU decodes and identifies the operation encoded in the instruction in the IR.
5. The relevant circuitry is activated to execute the operation.

19.5 The Fetch–Decode–Execute cycle

The universal machine cycle

All general-purpose computers follow the same basic cycle. The FDE cycle represents the repeated set of steps applied to each instruction in a program.



19.5.1 Instruction processing in detail

1. **Fetch.** The CPU fetches the next instruction from memory. It is read from program memory and loaded into the instruction register.
2. **Decode.** The instruction is interpreted, and the necessary operands are fetched from memory and stored in the CPU's registers.
3. **Execute.** The CPU performs the operation specified — adding two numbers, comparing two values, and so on.
4. **Update the program counter.** The PC is incremented to point to the next instruction in memory.
5. **Repeat** until the program is complete.

19.5.2 The fetch phase in register transfer notation

The three steps

$MAR \leftarrow PC$	copy the contents of the PC into the MAR
$IR \leftarrow M[MAR]$	fetch the instruction at that address into the IR
$PC \leftarrow PC + 1$	advance the program counter

The increment is not always 1

The PC is incremented by 1 only if the memory system is **word addressable**. If the memory is byte addressable:

Addressability	Instruction size	PC increment
Word addressable	one word	+1
Byte addressable	16 bits (2 bytes)	+2
Byte addressable	32 bits (4 bytes)	+4

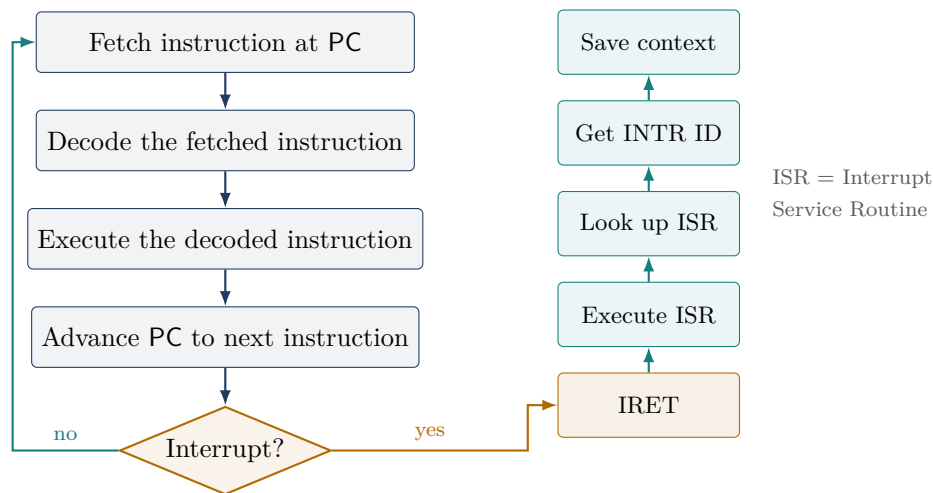
The rule is simply: advance by *however many addressable units one instruction occupies*. Notice the control-flow example in Chapter 20, where instruction addresses go 30, 32, 34, ... — byte addressable with 16-bit instructions.

19.5.3 Decode and execute

The control circuitry interprets the **opcode** portion of the IR and activates the corresponding execution hardware. Structurally this is a decoder (Chapter 13) driving a set of control lines — which is the concrete reason decoders matter.

19.6 Interrupts

Real processors extend the loop with a check for interrupts after each instruction.



Why interrupts exist

Without them the CPU would have to *poll* every device continuously to see whether it needed attention — wasting cycles on devices that are idle. An interrupt lets the device signal the CPU instead, so the processor only does work when there is work to do. The interrupt ID typically arrives from a **priority encoder** (Chapter 13) that turns many request lines into one number. “Save context” means pushing the PC and register contents somewhere safe so the interrupted program can resume as though nothing happened.

19.7 Summary

Concept	Key point
Stored program	Instructions live in the same memory as data and can be changed without rewiring.
Von Neumann architecture	CPU (ALU + CU), single memory, single ISA, I/O, buses.
Harvard architecture	Separate instruction and data memories; faster, more hardware.
Von Neumann bottleneck	One shared bus limits instruction and data throughput.
PC	Address of the next instruction.
MAR / MDR	Where / what for every memory access.
IR (CIR)	Holds the instruction currently being processed.
AC	Holds intermediate arithmetic results.
Fetch phase	$MAR \leftarrow PC$; $IR \leftarrow M[MAR]$; $PC \leftarrow PC + k$.
PC increment k	Number of addressable units one instruction occupies.
Interrupts	Let devices signal the CPU instead of being polled.

Chapter 20

Instruction Set Architecture

Where this fits

The ISA is the contract between hardware and software: it defines exactly what instructions exist and how each is encoded in bits. This chapter works through a complete small machine — its instruction set, its encoding, and programs written for it. Source: Lecture 3.2.

20.1 What an ISA is

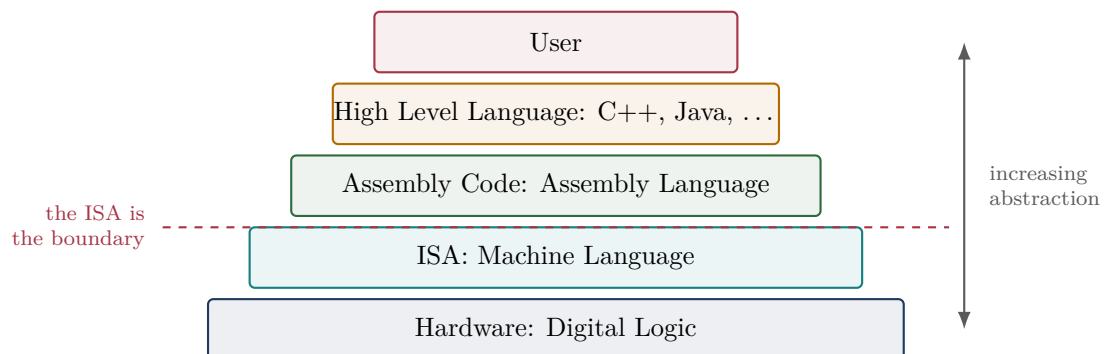
Instruction Set Architecture

The ISA of a machine specifies the **instructions the computer can perform** and the **format of each instruction**.

Why the ISA is the crucial interface

Everything above the ISA is software; everything below is hardware. Two processors implementing the same ISA can have utterly different internal designs yet run the same binaries — which is exactly the situation in the Chapter 18 comparison of Machines A and B. This separation lets hardware evolve without breaking existing software.

20.2 Levels of a modern computing system



20.3 Types of machine instruction

Category	Purpose
Data transfer	Transfer data between registers and memory cells.
Arithmetic / logic	Perform addition, AND, OR, XOR, shifts, and so on.
Control	Control the execution flow of the program.

20.4 The example machine

Lecture 3.2 defines a small but complete architecture. Everything that follows refers to it.

Feature	Specification
Main memory	256 bytes, addresses 00_{16} – FF_{16} , byte addressable
Registers	16 general purpose, 0_{16} – F_{16}
Instruction size	16 bits: 4 bits opcode + 12 bits operand
Integer format	8-bit, two's complement
Floating point format	8-bit: 1 sign bit, 3 exponent bits, 4 mantissa bits
Instruction set size	16 instructions, 0_{16} – F_{16}

Where the numbers come from

- 4 opcode bits $\Rightarrow 2^4 = 16$ possible instructions.
- 4 register-select bits $\Rightarrow 2^4 = 16$ registers.
- 8 address bits $\Rightarrow 2^8 = 256$ addressable bytes.

Every one of these is the n bits $\Rightarrow 2^n$ patterns rule from Chapter 1. Note also that the floating point format is precisely the 8-bit format built in Chapter 5.

20.5 Data transfer instructions

Mnemonic	Operands	Meaning
L	R, A	LOAD register R with the content of memory cell A
LI	R, I	LOAD register R with I (an <i>immediate</i> number)
ST	R, A	STORE the content of register R into memory cell A
LR	R1, R2	LOAD register R1 with the content of register R2

Worked Example 1 *Swapping two memory cells — Lecture 3.2*

Swap the contents of memory cells 30_{16} and 40_{16} .

	Before	After
L 1, 30 /* R1 <- M[30] */	M[30 ₁₆] 0011 0000	0100 0000
L 2, 40 /* R2 <- M[40] */	M[40 ₁₆] 0100 0000	0011 0000
ST 1, 40 /* M[40] <- R1 */	R1 —	0011 0000
ST 2, 30 /* M[30] <- R2 */	R2 —	0100 0000

Why two registers are needed. Storing R1 into 40_{16} destroys the old value of 40_{16} , so it must already be safe in R2. This is the classic “you need a temporary variable to swap” problem, at machine level.

20.6 Arithmetic and logic instructions

Mnemonic	Operands	Meaning
ADD	R0, R1, R2	Add the numbers in R1 and R2, represented in two's complement , and place the result in R0
AFP	R0, R1, R2	Add the numbers in R1 and R2, represented in floating point , result in R0
OR	R0, R1, R2	OR the bit patterns in R1 and R2, result in R0
AND	R0, R1, R2	AND the bit patterns, result in R0
XOR	R0, R1, R2	XOR the bit patterns, result in R0

Two separate ADD instructions

ADD and **AFP** exist as distinct instructions because two's complement integers and floating point numbers are *different encodings* of the same bit patterns (Chapters 4 and 5). The hardware must be told which interpretation to apply — the bits alone cannot say.

Worked Example 2 Integer addition

```
L   R1, 30
L   R2, 40
ADD R0, R1, R2
ST  R0, 80
```

Location	Content
M[30 ₁₆]	0011 0000
M[40 ₁₆]	0100 0000
M[80 ₁₆]	0111 0000
R1	0011 0000
R2	0100 0000
R0	0111 0000

Check: $00110000_2 = 48$, $01000000_2 = 64$, and $48 + 64 = 112 = 01110000_2$. ✓

Worked Example 3 Bit masking — Lecture 3.2

Mask off the first (high) 4 bits of the binary string held at memory address 80₁₆, storing the result at A0₁₆.

```
L   1, 80
LI  2, 0F
AND 0, 1, 2
ST  0, A0
```

	Content
M[80 ₁₆]	0011 0110 source
R1	0011 0110 loaded
R2	0000 1111 the <i>mask</i> , 0F ₁₆
R0	0000 0110 masked result
M[A0 ₁₆]	0000 0110 stored

Why AND masks. $x \cdot 0 = 0$ and $x \cdot 1 = x$ (annulment and identity laws, Chapter 8). So a 0 in the mask forces that bit to zero, and a 1 lets it through unchanged.

```
      0011 0110
AND  0000 1111
-----
      0000 0110
```

The high nibble is cleared and the low nibble survives. Masking is how individual flag bits are extracted from packed status words — a technique used constantly in systems programming.

20.7 Control instructions

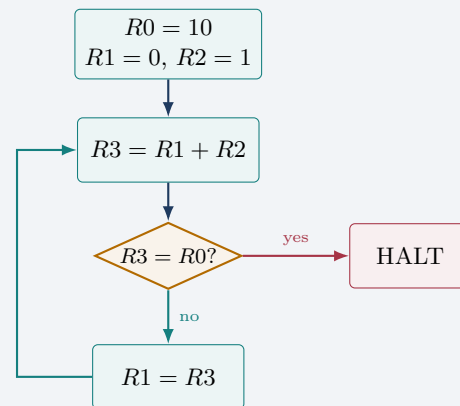
Mnemonic	Operands	Meaning
JMP	R, A	JUMP to the instruction at memory cell A if the bit pattern in R equals the one in R0. (If equal, reset the PC to that address.)
HALT	—	HALT execution

Getting an unconditional jump for free

JMP is conditional — it compares a register against R0. But `JMP 0, addr` compares R0 with R0, which is *always* true. So `JMP 0, addr` is an unconditional jump. Small instruction sets routinely use tricks like this to avoid spending a precious opcode.

Worked Example 4 A counting loop — Lecture 3.2

Addr	Instruction	Effect
30 ₁₆	LI 0, 0A	R0 = 0A
32 ₁₆	LI 1, 00	R1 = 00
34 ₁₆	LI 2, 01	R2 = 01
36 ₁₆	ADD 3, 1, 2	R3 = R1 + R2
38 ₁₆	JMP 3, 3E	if R3 == R0 exit
3A ₁₆	LR 1, 3	R1 = R3
3C ₁₆	JMP 0, 36	loop back
3E ₁₆	HALT	



Trace. R0 holds the limit 10, R2 the constant 1, and R1 the running value.

Pass	R1 before	R3	Action
1	0	1	$1 \neq 10 \Rightarrow R1 = 1$, loop
2	1	2	$2 \neq 10 \Rightarrow R1 = 2$, loop
⋮			
9	8	9	$9 \neq 10 \Rightarrow R1 = 9$, loop
10	9	10	$10 = 10 \Rightarrow$ HALT

Two things to notice

Addresses go up by two — 30₁₆, 32₁₆, 34₁₆, ... — because the memory is byte addressable and each instruction occupies 16 bits = 2 bytes. This is exactly the $PC \leftarrow PC + 2$ case from Chapter 19.

This eight-instruction program executes about 60 instructions — the static versus dynamic count distinction from Chapter 18, in miniature.

20.8 Instruction encoding

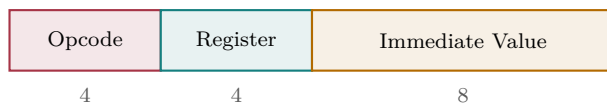
The design problem

Every instruction must fit in 16 bits. But the instructions need different things: some need a register and an 8-bit address, some need three registers, some need nothing at all. *How do we make them all fit?*

The answer is **multiple instruction formats**. The 4-bit opcode always occupies the same position, and the decoder uses it to determine how to interpret the remaining 12 bits.

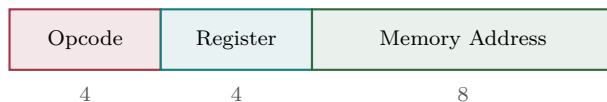


20.8.1 Type 1 — register and immediate value



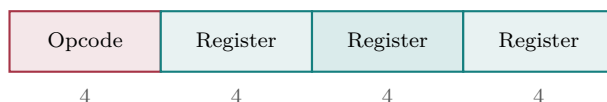
Opcode	Instruction	Meaning
2_{16}	LI R, I	Load Immediate
A_{16}	RL R, I	Rotate Left
B_{16}	RR R, I	Rotate Right
C_{16}	SL R, I	Shift Left
D_{16}	SR R, I	Shift Right

20.8.2 Type 2 — register and memory address



Opcode	Instruction	Meaning
1_{16}	L R1, 0x11	Load from a memory address
3_{16}	ST R2, 0x12	Store at a memory address
E_{16}	JMP R3, 0x11	Conditional jump

20.8.3 Type 3 — three registers



Opcode	Instruction	Meaning
5_{16}	ADD R1,R2,R3	Addition ($R1 = R2 + R3$)
6_{16}	AFP R1,R2,R3	Floating point addition
7_{16}	OR R1,R2,R3	Logical OR
8_{16}	AND R1,R2,R3	Logical AND
9_{16}	XOR R1,R2,R3	Logical XOR

20.8.4 Type 4 — two registers



Opcode	Instruction	Meaning
4_{16}	LR R1, R2	Load Register ($R1 = R2$)

20.9 The complete instruction set

Op	Instruction	Type	Op	Instruction	Type
0_{16}	<i>spare</i>	—	8_{16}	AND R0,R1,R2	3
1_{16}	L R, A	2	9_{16}	XOR R0,R1,R2	3
2_{16}	LI R, I	1	A_{16}	RL R, I	1
3_{16}	ST R, A	2	B_{16}	RR R, I	1
4_{16}	LR R1, R2	4	C_{16}	SL R, I	1
5_{16}	ADD R0,R1,R2	3	D_{16}	SR R, I	1
6_{16}	AFP R0,R1,R2	3	E_{16}	JMP R, A	2
7_{16}	OR R0,R1,R2	3	F_{16}	HALT	—

Opcode 0_{16} is free for any additional instruction — MULT R0,R1,R2, for example.

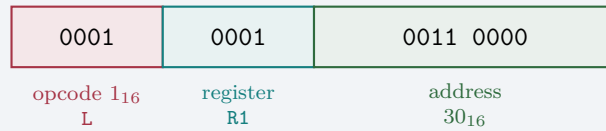
20.10 Assembling a program

Worked Example 5 Assembler to machine code — Lecture 3.2

The swap program, assembled:

Assembler	Machine code	Hex
L 1, 30	0001 0001 0011 0000	1130_{16}
L 2, 40	0001 0010 0100 0000	1240_{16}
ST 1, 40	0011 0001 0100 0000	3140_{16}
ST 2, 30	0011 0010 0011 0000	3230_{16}

Reading off the first instruction:



How it sits in memory. Each 16-bit instruction occupies two bytes:

Address	Byte	
10_{16}	0001 0001	} L 1, 30
11_{16}	0011 0000	
12_{16}	0001 0010	} L 2, 40
13_{16}	0100 0000	
14_{16}	0011 0001	} ST 1, 40
15_{16}	0100 0000	
16_{16}	0011 0010	} ST 2, 30
17_{16}	0011 0000	
30_{16}	1001 1001	data
40_{16}	0110 1101	data

Note the stored program in action. Addresses 10_{16} – 17_{16} hold instructions and 30_{16} , 40_{16} hold data — in the *same* memory, distinguished only by how the CPU chooses to read them.

Assembling the counting loop

The slides give the counting loop only in assembler. Encoding it with the instruction formats above:

Addr	Assembler	Fields	Hex
30_{16}	LI 0, 0A	$2_{16} \mid 0_{16} \mid 0A_{16}$	$200A_{16}$
32_{16}	LI 1, 00	$2_{16} \mid 1_{16} \mid 00_{16}$	2100_{16}
34_{16}	LI 2, 01	$2_{16} \mid 2_{16} \mid 01_{16}$	2201_{16}
36_{16}	ADD 3, 1, 2	$5_{16} \mid 3_{16} \mid 1_{16} \mid 2_{16}$	5312_{16}
38_{16}	JMP 3, 3E	$E_{16} \mid 3_{16} \mid 3E_{16}$	$E33E_{16}$
$3A_{16}$	LR 1, 3	$4_{16} \mid 0_{16} \mid 1_{16} \mid 3_{16}$	4013_{16}
$3C_{16}$	JMP 0, 36	$E_{16} \mid 0_{16} \mid 36_{16}$	$E036_{16}$
$3E_{16}$	HALT	$F_{16} \mid 000_{16}$	$F000_{16}$

The whole program is 16 bytes: 200A 2100 2201 5312 E33E 4013 E036 F000. Being able to move between assembler and hex in both directions is a standard exam skill.

20.11 Case study: sizing an architecture

Worked Example 6 *Lecture 3.2 case study*

Consider an architecture with:

- word addressable memory, word size = 16 bits;
- instructions 16 bits long, of which 4 bits are opcode and 4 bits are a register identifier;
- addresses 8 bits long.

Q. What is the maximum number of registers this machine can have?

The register identifier field is 4 bits:

$$2^4 = \mathbf{16} \text{ registers}$$

Q. What is the largest memory this machine can have?

Addresses are 8 bits and the memory is word addressable with 16-bit words:

$$2^8 = 256 \text{ words} = 256 \times 2 \text{ bytes} = \mathbf{512} \text{ bytes}$$

Q. Do all the instructions fit in 16 bits?

Check the worst cases:

Instruction	Fields	Total
L 2, A1	4 + 4 + 8	16 bits ✓
LI 0, 0A	4 + 4 + 8	16 bits ✓
ADD 3, 1, 2	4 + 4 + 4 + 4	16 bits ✓
AND 1, 1, 3	4 + 4 + 4 + 4	16 bits ✓
JMP 3, 3E	4 + 4 + 8	16 bits ✓
LR 1, 3	4 + 4 + 4 + 4	16 bits (4 unused) ✓
ST 0, 3E	4 + 4 + 8	16 bits ✓
HALT	4	16 bits (12 unused) ✓

Every instruction fits, but some waste bits. That waste is the price of a **fixed-length** instruction format — and the price is worth paying, because the CPU always knows where the next instruction starts without decoding the current one. Variable-length encodings (as in x86) pack more densely but are much harder to decode and to pipeline (Chapter 22).

20.12 Summary

Concept	Key point
ISA	The instructions available and the format of each; the hardware/-software boundary.
Instruction categories	Data transfer, arithmetic/logic, control.
Opcode	Fixed-position field that selects the instruction and the format.
Instruction formats	Different field layouts sharing a fixed total width.
Fixed length	Simple to fetch and decode; wastes some bits.
Immediate	Operand value embedded in the instruction itself.
Masking	AND with a bit pattern to clear selected bits.
Unconditional jump	JMP 0, addr — comparing R0 with itself.
Field sizing	n -bit field $\Rightarrow 2^n$ registers / addresses / opcodes.

Chapter 21

Addressing Modes

Where this fits

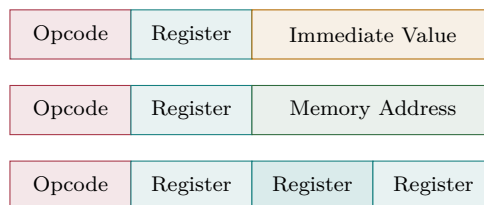
An instruction's operand field is a pattern of bits. What those bits *mean* — a value, a register number, a memory address, or a pointer to an address — is determined by the addressing mode. Source: Lecture 3.3.

21.1 The idea

Addressing mode

A specification of **where the instruction's operands are located**. An addressing mode can specify a constant, a register, or a location in memory.

Recall the instruction layouts from Chapter 20:



Same 12 bits, three different meanings.

How does the CPU know which mode is in use?

In this ISA, the **opcode itself** determines the format — LI always takes an immediate, L always takes a memory address. Larger architectures add an explicit **mode field** to the instruction so the same opcode can be used with several addressing modes.

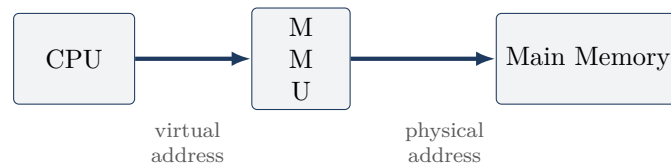
21.2 Effective and physical addresses

Effective address

The location of the **actual operand**. In a system without virtual memory the effective address is either a main memory address or a register. In a system *with* virtual memory it is a virtual address or a register.

Physical address

The real hardware location in memory. Translating an effective (virtual) address into a physical one is the job of the **Memory Management Unit (MMU)**, which manages and controls access to memory. Because of the MMU, finding the physical address is **invisible to the programmer**.



21.3 The common addressing modes

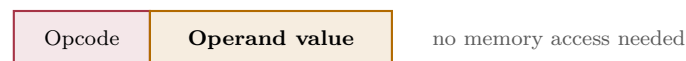
Immediate	Register
Direct	Register indirect
Indirect	Displacement
	Stack

Typical computer architectures provide several of these.

21.3.1 Immediate addressing

Immediate addressing

The instruction **contains the operand value itself**. The operation code is immediately followed by the value to be referenced — the data item to be operated on is part of the instruction.



no memory access needed

LI R1, 0x0A — load register R1 with the immediate number $0A_{16}$

Pros

Very fast — the value is already in the instruction, so no memory reference is needed.

Cons

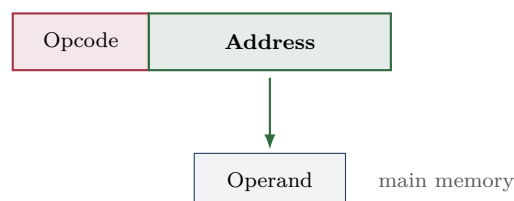
Less flexible: the value is fixed at compile time.

The size of the number is restricted to the size of the address field.

21.3.2 Direct addressing

Direct addressing

The instruction's address field contains the **effective address** of the operand.



LOAD R1, 0x80 — load R1 with the content of memory cell 80_{16}

Pros

Fast — the value is accessed in a single memory reference.

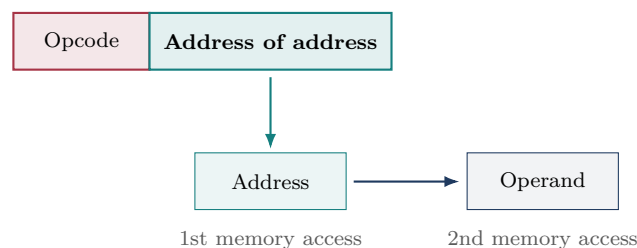
More flexible than immediate, since the value is not baked into the instruction.

Cons

Provides only a limited address space (the address field is narrower than a word).

21.3.3 Indirect addressing**Indirect addressing**

The instruction's address field contains a **pointer** to the effective address of the operand — an address of an address.

**Worked Example 1** *Indexing into an array — Lecture 3.3*

Suppose an array of numbers is stored in memory starting at address 1000_{16} , and we want the value at index 2.

Register R1 holds the base address of the array (1000_{16}) and R2 holds the index (2).

Step 1 — compute the address of the element:

```
ADD R3, R1, R2
```

Now R3 holds the address of the desired element.

Step 2 — load the value at that address, using indirect addressing:

```
LD R4, (R3)
```

The **brackets** indicate a reference to the value *pointed to* by the address enclosed within them.

The same idea in C

Declaring `int *ptr;` creates a variable that stores the memory address of an integer. The dereference operator `*` follows the pointer:

```
int x = 42;
int *ptr = &x;    // ptr holds the address of x
int value = *ptr;  // dereference: value becomes 42
```

`ptr` is direct addressing; `*ptr` is indirect addressing. Pointers in C are a direct, deliberate exposure of the hardware's addressing modes — which is why C is described as a low-level language.

Pros

A much larger memory space is available, since an entire *word* can hold the address rather than just the instruction's address field.

Cons

Instruction execution requires **two** memory references to fetch the operand: one to get its address and a second to get its value.

21.3.4 Register addressing (register direct)

Register addressing

Similar to direct addressing, but a **register** rather than a memory location holds the operand.

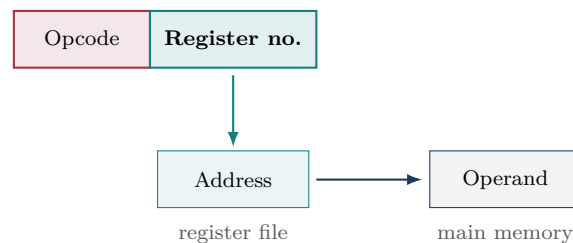
ADD R1, R2, R3

This is the fastest mode of all — registers are inside the CPU, so there is no bus transaction at all. It is why compilers work so hard at *register allocation*: keeping hot variables in registers rather than memory.

21.3.5 Register indirect addressing

Register indirect addressing

A variant of indirect addressing. The instruction's operand specifies a **register that holds a memory location** (the effective address) of the data item to be used.



Only *one* memory access is needed, since the pointer is already in a register. This is the mode LD R4, (R3) actually uses.

21.3.6 Displacement addressing

Displacement addressing

A combination of direct addressing and register indirect addressing. The instruction comes with **two address fields**: the effective address is the sum of a register's contents and a constant offset.

$$\text{effective address} = (\text{register}) + \text{displacement}$$



Why displacement addressing is everywhere

It maps directly onto how programs actually use memory:

- **Array indexing** — register holds the base address, displacement is the element offset (or vice versa).
- **Struct / object fields** — register points at the object, displacement is the field offset.
- **Local variables** — register is the stack frame pointer, displacement is the variable's position in the frame.

MOV EAX, [EBP-8] in x86 or lw \$t0, 12(\$sp) in MIPS are both displacement addressing. It is the workhorse mode of real machines.

21.3.7 Stack addressing

Only listed in the slides

The operand is implicitly at the **top of the stack**, so no address field is needed at all. Instructions such as PUSH, POP and stack-based arithmetic use the stack pointer implicitly.

Advantage: extremely compact instructions — an ADD that pops two values and pushes their sum needs *only* an opcode.

Disadvantage: data must be arranged on the stack in the right order, which costs extra instructions.

Stack machines are the basis of the Java Virtual Machine and of Reverse Polish calculators.

21.4 Worked comparison

Worked Example 2 *The activity from Lecture 3.3*

Memory contains:

Address	Content
100 ₁₆	210 ₁₆
210 ₁₆	350 ₁₆
350 ₁₆	050 ₁₆

The instruction is LOAD R1, 0x100. What value is loaded in each mode?



Mode	Loaded value	Reasoning
Immediate	100 ₁₆	The operand <i>is</i> the value in the instruction.
Direct	210 ₁₆	Fetch the content of address 100 ₁₆ .
Indirect	350 ₁₆	Fetch the content of 100 ₁₆ (= 210 ₁₆), then fetch the content of 210 ₁₆ .

Count the memory accesses: immediate needs **zero**, direct needs **one**, indirect needs **two**. That is exactly the cost–flexibility trade-off running through this chapter.

21.5 Summary

Mode	Operand is...	Mem. acc.	Typical use
Immediate	in the instruction	0	Constants
Register	in a register	0	Arithmetic operands
Direct	at the given address	1	Global variables
Register indirect	at the address in a register	1	Following a pointer
Displacement	at register + offset	1	Arrays, structs, locals
Indirect	at the address stored at the given address	2	Pointer tables, jump tables
Stack	at the top of the stack	implicit	Stack machines, JVM

The one-line rule

Each extra level of indirection buys flexibility and costs a memory access. Immediate is fastest and least flexible; indirect is slowest and most flexible; displacement is the practical compromise, which is why real ISAs lean on it.

Chapter 22

Instruction Level Pipelining

Where this fits

The final topic, and the one that connects everything: pipelining improves throughput by overlapping the fetch–decode–execute cycles of consecutive instructions. It directly attacks the CPI term of the performance equation from Chapter 18. Source: Lecture 3.3.

22.1 The idea

The core insight

Modern CPUs divide the fetch–decode–execute cycle into small steps. Each step is made **independent**, so the smaller steps can be performed in parallel.

*Divide the execution of an instruction into several stages, then execute each stage of **different** instructions in parallel.*

Nothing makes a single instruction finish sooner. What changes is that many instructions are *in flight* at once, so instructions *complete* much more often.

The laundry analogy

Washing, drying and folding one load takes three hours. Doing four loads sequentially takes twelve. But once the first load moves to the dryer, the washer is free — start load two. With three machines running in parallel, four loads take six hours, not twelve. The first load still takes three hours. That is exactly a pipeline: latency unchanged, throughput improved.

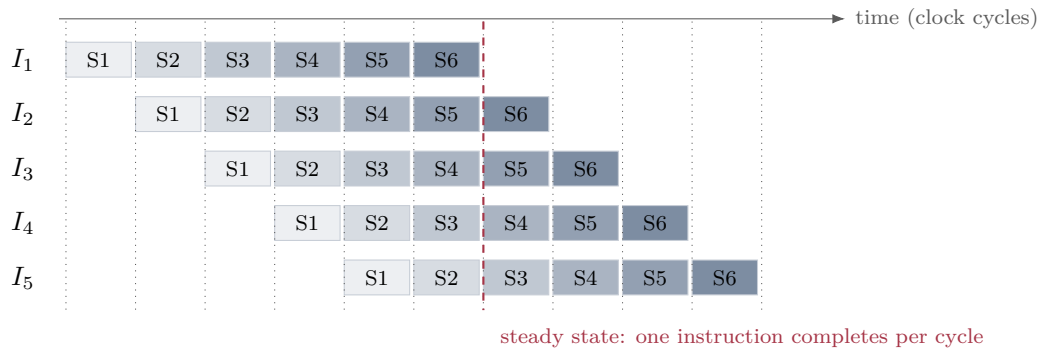
22.2 A typical set of stages

Stage	Action
1	Fetch instruction
2	Decode opcode
3	Calculate effective address of operands
4	Fetch operands
5	Execute instruction
6	Store result

Note that stage 3 is where the addressing modes of Chapter 21 are resolved.

22.3 Overlapping execution

Instruction level pipelining overlaps these smaller steps across a set of consecutive instructions.



The stages must be balanced

Each pipeline stage must take about the same time, because the clock period is set by the *slowest* stage. A six-stage pipeline whose stages take 2, 2, 2, 2, 10, 2 nanoseconds runs at 10 ns per stage — the five fast stages spend most of their time idle, and the speedup is far below what the stage count suggests. Balancing the stages is one of the central problems in processor design.

22.4 Pipeline efficiency

Deriving the timing

Suppose we have a k -stage pipeline, the clock cycle time is t_p (so each stage takes t_p), and there are n instructions to process.

- Instruction I_1 must pass through all k stages, requiring $k \times t_p$ time to complete.
- The remaining $n - 1$ instructions then emerge from the pipeline **one per cycle**, taking a further $(n - 1)t_p$.

Therefore the total time to complete n instructions on a k -stage pipeline is

$$k \times t_p + (n - 1)t_p = \boxed{(k + n - 1)t_p}$$

Speedup

Without a pipeline, each of the n instructions takes all k stages sequentially, i.e. $k \times t_p$ each, for a total of $k t_p n$. So

$$\text{Speedup} = \frac{k \times t_p \times n}{(k + n - 1)t_p} = \frac{kn}{k + n - 1}$$

Worked Example 1 Applying the formulas

A 6-stage pipeline with a cycle time of 2 ns runs 100 instructions.

Total time:

$$(k + n - 1)t_p = (6 + 100 - 1) \times 2 \text{ ns} = 105 \times 2 = 210 \text{ ns}$$

Without a pipeline:

$$k t_p n = 6 \times 2 \times 100 = 1200 \text{ ns}$$

Speedup:

$$\frac{1200}{210} = \frac{6 \times 100}{6 + 100 - 1} = \frac{600}{105} \approx 5.71$$

The limit

As n grows large, $k + n - 1 \rightarrow n$, so

$$\lim_{n \rightarrow \infty} \frac{k n}{k + n - 1} = k$$

The **maximum possible speedup equals the number of stages**, approached but never quite reached. The shortfall is the **pipeline fill time** — the $k - 1$ cycles at the start before the first instruction emerges.

n (instructions)	Speedup ($k = 6$)	% of ideal
1	$6/6 = 1.00$	17%
10	$60/15 = 4.00$	67%
100	$600/105 \approx 5.71$	95%
1000	$6000/1005 \approx 5.97$	99.5%

Pipelining pays off only for long instruction streams — which is exactly what real programs are (Chapter 18: twelve million instructions from a one-page program).

Connecting back to CPI

In the ideal steady state one instruction completes per cycle, so $\text{CPI} \rightarrow 1$. Splitting the work into shorter stages also permits a *shorter* clock cycle t . Pipelining therefore improves two of the three terms in $\text{CPU Time} = N \times \text{CPI} \times t$ simultaneously — which is why every modern processor is pipelined.

22.5 Hazards — why the ideal is not reached

Left as homework in Lecture 3.3: advantages, disadvantages, limitations and issues

Advantages

- Much higher instruction throughput; CPI approaches 1.
- Better hardware utilisation — every stage is busy every cycle.
- Allows a higher clock rate, since each stage does less work.
- No change required to the instruction set — it is invisible to software.

Disadvantages and limitations

- Latency of a single instruction *increases* slightly, due to pipeline register overhead.
- Considerable extra hardware: registers between stages, forwarding paths, hazard detection logic.
- Stages must be balanced or the benefit collapses.
- Deeper pipelines suffer more from every hazard below.

The three classes of hazard

1. Structural hazards — two instructions need the same hardware resource in the same cycle. For example, one instruction fetching data from memory while another fetches an instruction, over a single shared bus — the **von Neumann bottleneck** from Chapter 19.

Solutions: duplicate the resource; use separate instruction and data caches (the Harvard-inside approach); or stall one instruction.

2. Data hazards — an instruction needs a result that an earlier instruction has not yet produced.

ADD R1, R2, R3

SUB R4, R1, R5

SUB needs R1, but ADD has not written it back yet.

Solutions: **forwarding** (route the ALU result directly to the next instruction's input rather than waiting for write-back); insert **stalls** (pipeline bubbles); or let the compiler reorder instructions to fill the gap.

3. Control hazards — a branch or jump means the CPU does not know which instruction to fetch next until the branch is resolved. By then several wrong instructions may already be in the pipeline.

Solutions: **branch prediction** (guess the outcome and speculatively execute, discarding the work if wrong); **delayed branches** (define the instruction after a branch to always execute); or resolve branches earlier in the pipeline.

The cost of a wrong guess. If a branch is mispredicted, every instruction fetched after it must be discarded and the pipeline refilled — costing roughly $k - 1$ cycles. This is why deep pipelines need very accurate branch predictors, and why unpredictable branches are expensive in performance-critical code.

Beyond simple pipelining

- **Superscalar** — multiple pipelines in parallel, issuing several instructions per cycle. This is how CPI drops *below* 1, as in Machine B of Chapter 18.
- **Out-of-order execution** — execute instructions as their operands become ready rather than in program order, then commit results in order.
- **Speculative execution** — proceed past an unresolved branch on a prediction and undo the work if wrong.

22.6 Summary

Concept	Key point
Pipelining	Overlap the stages of consecutive instructions.
Effect	Throughput improves; single-instruction latency does not.
Stage balance	Clock period is set by the slowest stage.
Time for n instructions	$(k + n - 1) t_p$
Speedup	$\frac{kn}{k + n - 1}$, approaching k for large n
Maximum speedup	k , the number of stages
Structural hazard	Two instructions need the same resource.
Data hazard	An operand is not ready yet; fixed by forwarding or stalls.
Control hazard	Branch target unknown; fixed by prediction.

Closing the loop

This last chapter draws on almost everything before it. The pipeline registers between stages are flip-flops (Chapter 15). The stages decode opcodes using decoders (Chapter 13) and compute in the ALU (Chapter 12), which is built from adders, which are built from gates (Chapter 6), which realise Boolean algebra (Chapter 8) operating on binary representations (Chapter 4).

That chain — from a single bit to a pipelined processor — is what this course is about.

PART VI

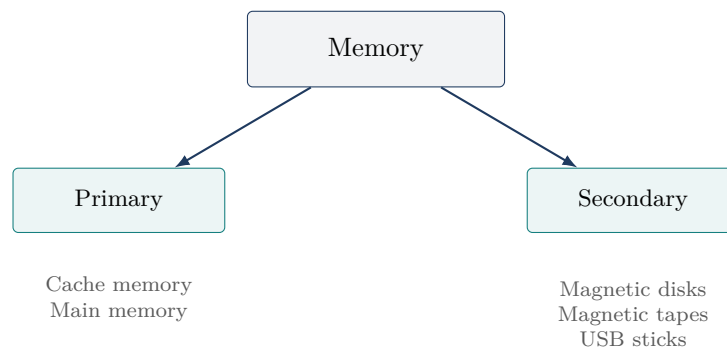
Memory Systems

Memory Technologies and the Hierarchy

Where this fits

Chapter 17 treated main memory as a flat array of addressable words. That abstraction is convenient but false: real systems use half a dozen storage technologies of wildly different speed and cost, arranged in a hierarchy. This chapter explains what those technologies are, why the hierarchy exists, and the property of programs — *locality* — that makes it work. Source: Lecture 4.1.1.

23.1 Primary and secondary memory



The distinction

Primary memory is directly accessible by the CPU — it sits on the address and data buses, and the processor can read a word from it with a single instruction. Cache and main memory are primary.

Secondary memory is not directly addressable by the CPU. Data must first be copied into primary memory by the operating system before it can be used. Disks, tapes and USB sticks are secondary.

23.2 Types of memory

23.2.1 Read Only Memory (ROM)

Property	Non-volatile — retains its contents without power
PROM	Programmable Read Only Memory — can be written once, by the purchaser rather than the manufacturer
EEPROM	Electrically Erasable Programmable Read Only Memory — can be erased and rewritten electrically, in place

ROM holds firmware — the boot code that must survive a power cycle, such as the BIOS or UEFI that runs before an operating system exists.

23.2.2 Random Access Memory (RAM)

Property	Volatile — contents are lost when power is removed
DRAM	Dynamic RAM
SRAM	Static RAM

“Random access” does not mean “unpredictable”

It means *any location can be reached in the same amount of time*, regardless of which location was accessed previously. Contrast this with **sequential access** on a magnetic tape, where reaching a distant record means winding through everything in between. Confusingly by this definition ROM is also random access; the RAM/ROM naming is historical rather than logical.

23.2.3 SRAM versus DRAM

	SRAM	DRAM
Storage cell	6 transistors forming a latch	1 transistor + 1 capacitor
Refresh needed?	no	yes , every few milliseconds
Speed	fast	slower
Density	low	high
Cost per bit	high	low
Power (idle)	lower	higher (refresh)
Typical use	cache , registers	main memory

Why the difference matters

An SRAM cell is a cross-coupled latch — exactly the circuit built in Chapter 15 — and holds its value as long as power is applied. A DRAM cell stores a bit as charge on a tiny capacitor, which *leaks*. The memory controller must therefore read and rewrite every row periodically; that is what “dynamic” means.

One transistor versus six is roughly a sixfold density advantage, which is why main memory is measured in gigabytes and cache in megabytes. The whole memory hierarchy exists because we cannot afford to make all memory out of SRAM.

23.2.4 Flash memory

Built from	Floating-gate metal–oxide–semiconductor field-effect transistors
Volatility	Non-volatile
Operations	Read <i>and</i> write
Common uses	Memory cards, USB flash drives, solid state drives

The floating gate

An ordinary MOSFET switches according to the voltage on its gate. A flash cell adds a second, electrically isolated gate — the *floating* gate — buried in insulator. Charge

forced onto it stays there for years with no power, and it shifts the threshold voltage of the transistor, which is what encodes the stored bit.

Flash therefore fills the gap that ROM and RAM leave open: non-volatile *and* rewritable.

23.2.5 Solid state devices (SSD)

Advantages over HDD	Limitations
More resistant to physical shock (no moving parts)	Not appropriate for long-term storage: without power the cells deplete their charge over time
Run silently	
Faster access time	

An SSD is not an archival medium

The lecture makes this point explicitly and it is worth taking seriously. A flash cell's trapped charge leaks slowly; an unpowered consumer SSD left on a shelf can begin losing data after a year or two, and faster at high temperature. For archival storage, magnetic tape or periodically-refreshed disks remain the correct choice.

3D XPoint — “read more about it” in Lecture 4.1.1

3D XPoint was a non-volatile memory technology jointly developed by **Intel and Micron** and announced in 2015. Intel marketed it under the brand name **Optane**.

Its selling point was filling the gap between DRAM and flash:

	DRAM	3D XPoint	NAND flash
Volatile?	yes	no	no
Access granularity	byte	byte	block
Relative latency	fastest	intermediate	slowest
Relative cost per bit	highest	intermediate	lowest

Being byte-addressable *and* non-volatile made it unusual: it could serve either as very fast storage or as slow, persistent main memory.

Status update. The lecture slide describes 3D XPoint as “newly developed”, which was reasonable when the technology appeared, but it has since been **discontinued**. Optane products were sold from 2017; Intel announced the wind-down of the business in 2022, Micron exited the technology and sold its 3D XPoint fabrication plant, and the final Optane persistent memory modules were scheduled to ship by the end of 2025. No new 3D XPoint products are being developed.

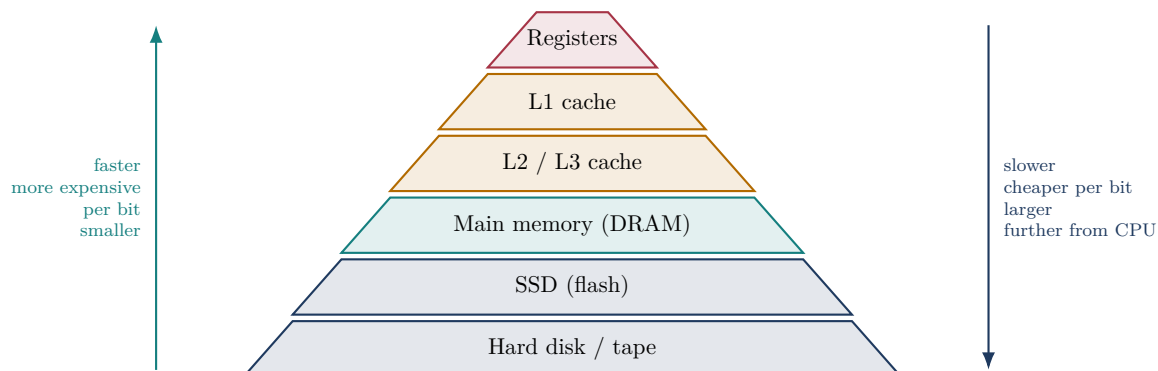
The commercial failure is itself instructive: a technology that is better-than-flash and worse-than-DRAM has to be cheap enough to justify a third tier in the hierarchy, and 3D XPoint never reached that price point while DRAM and flash both kept improving.

23.3 The memory hierarchy

Memory hierarchy

The organisation of a computer's storage components into levels, ranked by several attributes that vary together:

- **Access time** (latency)
- **Cost** per bit
- **Size** (capacity)
- **Distance from the CPU**



The attributes move together

Every level is *faster*, *smaller* and *dearer* than the one below it. That is not a coincidence — it is forced by physics and economics. Fast storage needs more transistors per bit (SRAM's six versus DRAM's one), so it costs more per bit, so you can afford less of it, so it must be small. And small means it can be placed physically closer to the CPU, which itself reduces latency, because signals travel at a finite speed.

Typical orders of magnitude

Exact figures vary enormously between processors, but the *ratios* are stable and worth internalising:

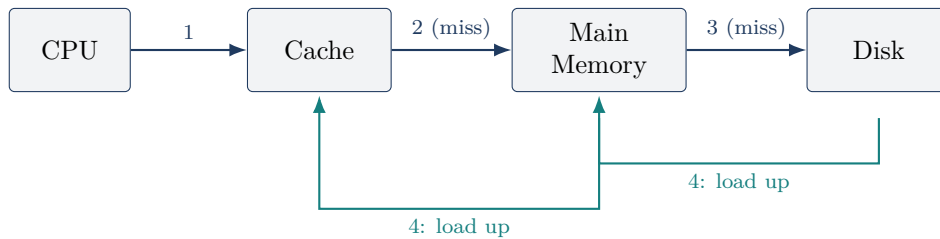
Level	Typical latency	Typical size	Technology
Register	< 1 ns	~1 kB	flip-flops
L1 cache	~1 ns	32–64 kB	SRAM
L2 cache	~4 ns	256 kB–1 MB	SRAM
L3 cache	~10–20 ns	8–32 MB	SRAM
Main memory	~60–100 ns	8–64 GB	DRAM
SSD	~50–100 μ s	0.5–4 TB	flash
Hard disk	~5–10 ms	1–20 TB	magnetic

The span from register to disk is roughly **seven orders of magnitude**. To put that in human terms: if an L1 cache access took one second, a main memory access would take about a minute and a disk access would take about four months.

23.4 The typical data access pattern

What happens on a memory request

1. The CPU requests data from its **nearest** storage — the cache.
2. If not found there, the request goes to **main memory**.
3. If not found there, the request goes to the **hard disk**.
4. Once the data is found it is loaded into main memory, and then into the cache.



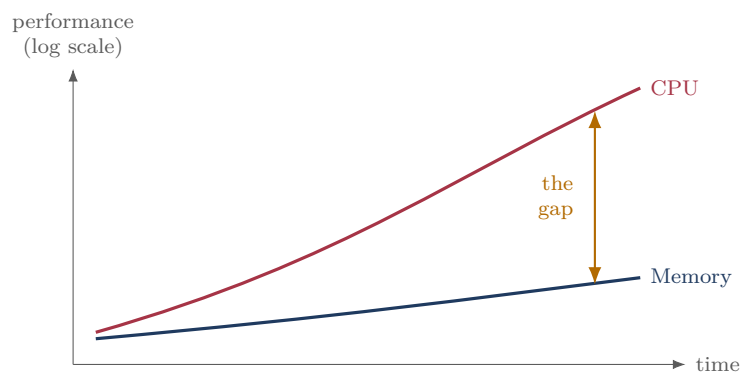
Each level is a cache for the level below

The hierarchy is recursive. The cache holds a subset of main memory; main memory holds a subset of the disk (this is what *virtual memory* does). The same ideas — blocks, hits, misses, replacement policies — apply at every level. Learn them once for the CPU cache and they transfer to page tables, database buffer pools, CDNs and web browser caches.

23.5 The CPU–memory gap

The problem

- CPU cycle time decreases rapidly as technology improves.
- Memory access times **cannot match** the CPU's speed improvement.
- Ordering by speed: HDD > SSD > DRAM > SRAM $\gg$ CPU (larger means slower).
- The CPU–memory gap therefore limits overall system performance.
- The **principle of locality** is used to bridge the gap.



Why a faster CPU alone does not help

Recall the CPU time equation from Chapter 18:

$$\text{CPU Time} = N \times \text{CPI} \times t$$

Shrinking t (a faster clock) does nothing for the cycles spent *waiting* for memory — those appear as an inflated CPI. If a program stalls for 100 ns on every cache miss, doubling the clock rate merely doubles the number of cycles it spends stalled. This is the practical meaning of the “memory wall” mentioned at the end of Chapter 18.

23.6 The principle of locality

Principle of locality

The tendency of a CPU to access the same set of memory locations repeatedly over a short period of time.

Equivalently: *programs tend to refer to data and instructions with addresses near or equal to those they have referred to recently.*

Spatial locality

Items with *nearby addresses* tend to be referenced close together in time.

Temporal locality

Recently referenced items are likely to be referenced again in the near future.

Where locality comes from — look at any loop

```
total = 0
for i = 0 to 999 do
    total = total + a[i]
endfor
```

Temporal locality: `total` and `i` are read and written on every one of the 1000 iterations. So are the loop’s own instructions.

Spatial locality: `a[0]`, `a[1]`, `a[2]`... occupy consecutive addresses and are touched in order.

Both forms arise from the same two features present in essentially every program: **loops** (which re-execute the same instructions on the same variables) and **arrays** traversed in order (which walk through consecutive addresses).



Why locality rescues the hierarchy

If memory accesses were uniformly random across a 16 GB address space, a 32 kB cache would hit roughly twice in a million accesses and would be worthless.

But accesses are *not* random. Because of locality, a small cache holding the recently-used and nearby data captures the overwhelming majority of references — hit ratios of 95% and above are routine. The entire memory hierarchy is a bet on locality, and in practice it is a bet that pays.

How each form of locality is exploited

Locality	Hardware mechanism
Temporal	Keep recently used data in the cache; evict the least recently used (Section 25.6).
Spatial	Fetch a whole block , not a single word — so the neighbours of the requested item arrive for free (Section 24.3).

Every design decision in Chapters 24 and 25 traces back to one of these two rows.

23.7 Summary

Term	Meaning
Primary memory	Directly addressable by the CPU: cache, main memory.
Secondary memory	Not directly addressable: disks, tapes, USB sticks.
ROM	Non-volatile, read-only; variants PROM and EEPROM.
RAM	Volatile; variants DRAM and SRAM.
SRAM	6-transistor latch cell; fast, low density, used for cache.
DRAM	1-transistor + capacitor cell; needs refresh; used for main memory.
Flash	Floating-gate transistors; non-volatile and rewritable.
SSD	Flash-based; shock resistant, silent, fast; poor for archival storage.
Memory hierarchy	Levels ordered by latency, cost, size and distance from CPU.
CPU–memory gap	CPU speed improves faster than memory latency.
Locality	Programs reuse recent addresses (temporal) and nearby ones (spatial).

Chapter 24

Cache Memory

Where this fits

The cache is the hardware answer to the CPU–memory gap. This chapter covers what a cache is, how its performance is measured, and two of the three design decisions every cache must make — its size and its write policy. The third, the mapping scheme, is large enough to need its own chapter. Source: Lecture 4.1.1.

24.1 What a cache is

Cache memory

A **smaller and faster** memory space that keeps data belonging to a **larger and slower** memory space. It resides on the CPU chip.

Typical characteristics:

- **Transient** — highly volatile.
- May keep **frequently referred** data (exploiting temporal locality).
- May keep data that **would be referred in future** (exploiting spatial locality).

The cache is invisible

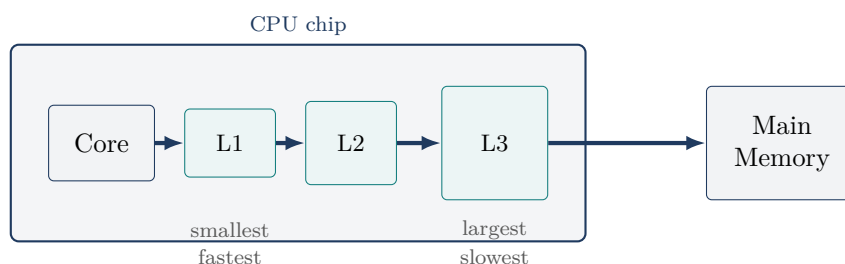
Nothing in the instruction set mentions the cache. A program issues `L R1, 0x80` and either gets its data in 1 ns or in 100 ns depending on whether that address happens to be cached. The cache is a pure *performance* optimisation, managed entirely by hardware and transparent to software.

This is the opposite of registers, which programs name explicitly. It is also why cache behaviour is such a common source of mysterious performance differences between two programs that do the same amount of “work”.

24.2 Cache hierarchy

Why there is more than one cache

- Cache memories are typically **SRAMs**.
- Per-unit cost is high.
- A faster *and* larger cache space would be expensive.
- As a solution, a **hierarchy of caches** is used — different levels, L1, L2 and L3.



Level	Typical size	Notes
L1	32–64 kB	Fastest; usually <i>split</i> into separate instruction and data caches, private to one core.
L2	256 kB–1 MB	Larger, a few times slower; usually private to a core.
L3	8–32 MB	Largest; usually <i>shared</i> between all cores on the chip.

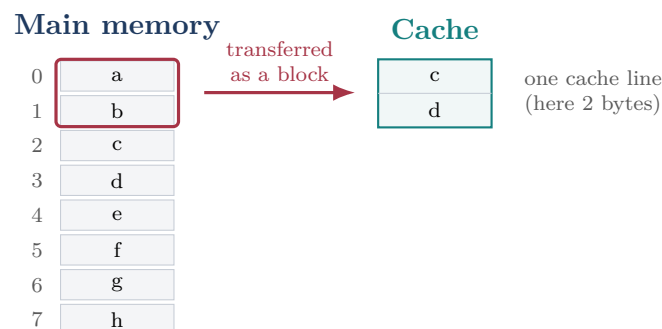
The hierarchy is the same trade-off, applied recursively

Each level answers “how do we get both speed and capacity?” the same way the overall memory hierarchy does — by not trying. A small fast level catches most references; a larger slower level catches most of what the first one misses. Splitting L1 into instruction and data halves is the Harvard-inside arrangement mentioned in Chapter 19.

24.3 Cache lines and cache entries

Data moves in blocks, not words

When the CPU asks for one byte and the cache misses, the cache does *not* fetch one byte. It fetches a whole surrounding block. This is precisely how **spatial locality** is exploited: the neighbours arrive for free, and the next few requests will probably be for them.



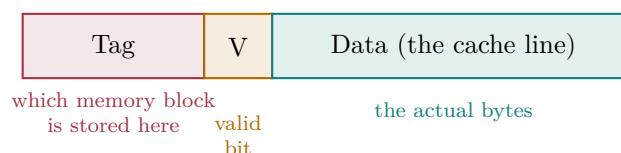
Cache line / cache block

Data is transferred between memory and cache in blocks of **fixed size**, called **cache lines** (or cache blocks). A single cache line may contain multiple bytes or words of data.

Cache entry

When a cache line is copied from memory into the cache, a **cache entry** is created. The entry includes the copied data *and* the requested memory location identifier, called a **tag**. The amount of data in a cache entry and in a cache line are always equivalent in size.

A cache entry



The valid bit

A cache powers up containing arbitrary bits, which happen to form a legal-looking tag and data. The **valid bit** distinguishes “this entry holds real data” from “this entry has never been filled”. A tag match with $V = 0$ is still a miss — see the search procedure in Section 25.2.4.

Choosing the line size is a trade-off

Larger lines exploit spatial locality better and need fewer tag bits per byte of data — but they waste bandwidth if the program does not actually use the neighbours, and there are fewer lines so conflicts become more likely.

Smaller lines waste less on unused data but capture less spatial locality and spend proportionally more space on tags.

Typical real caches use 32-byte or 64-byte lines. The lecture’s examples use 2-byte lines purely so the arithmetic stays small enough to do by hand.

24.4 Cache performance

Hit and miss

Cache hit — the data requested for processing by the CPU *is* found in the cache memory.

Cache miss — the requested data is *not* found.

Hit ratio and miss ratio

The **cache hit ratio** HR is the probability of finding requested data in the cache.

$$\text{Cache Miss Ratio} = 1 - \text{Cache Hit Ratio}$$

24.4.1 Effective access time

Effective Access Time (EAT)

The average time it takes for the CPU to access data.

$$EAT = HR \times T_c + (1 - HR) \times T_m$$

HR hit ratio

T_c cache access time in a cache hit

T_m data access time in a cache miss

Read it as a weighted average

A fraction HR of accesses cost T_c ; the remaining fraction $(1 - HR)$ cost T_m . The formula is just the expected value. Any weighted-average question — CPI in Chapter 18, EAT here — has the same shape.

Worked Example 1 Basic EAT calculation

A cache has an access time of 10 ns, main memory 100 ns, and the hit ratio is 0.90.

$$EAT = 0.90 \times 10 + 0.10 \times 100 = 9 + 10 = \mathbf{19 \text{ ns}}$$

Note what this says: although 90% of accesses are fast, the 10% that miss contribute *more* to the average than all the hits combined.

Worked Example 2 *How much does the hit ratio matter?*

Same $T_c = 10$ ns and $T_m = 100$ ns, varying the hit ratio:

HR	EAT	Speedup vs. no cache
0.00	100 ns	1.00×
0.50	55 ns	1.82×
0.90	19 ns	5.26×
0.95	14.5 ns	6.90×
0.99	10.9 ns	9.17×
1.00	10 ns	10.00×

The lesson. Going from $HR = 0.90$ to $HR = 0.99$ — eliminating nine tenths of the remaining misses — cuts EAT almost in half. Small improvements in hit ratio at the top end are worth a great deal, which is why processor designers spend so much silicon on cache.

Worked Example 3 *Working backwards to a required hit ratio*

$T_c = 5$ ns, $T_m = 80$ ns. What hit ratio gives $EAT \leq 10$ ns?

$$\begin{aligned}
 10 &\geq 5 HR + 80(1 - HR) \\
 10 &\geq 5 HR + 80 - 80 HR \\
 10 - 80 &\geq -75 HR \\
 -70 &\geq -75 HR \\
 HR &\geq \frac{70}{75} = 0.9\bar{3}
 \end{aligned}$$

A hit ratio of at least **93.3%** is required.

Two conventions for the miss cost — know which one is being used

This course's formula treats T_m as the *total* time for a miss:

$$EAT = HR \times T_c + (1 - HR) \times T_m$$

Many textbooks instead assume the cache is always checked *first*, so a miss costs the cache lookup *plus* the memory access:

$$EAT = T_c + (1 - HR) \times T_m \quad \text{or} \quad EAT = HR \times T_c + (1 - HR) \times (T_c + T_m)$$

With $T_c = 10$, $T_m = 100$, $HR = 0.9$ the first gives 19 ns and the second 20 ns. **Use the form given in your course**, and if a question is ambiguous, state which convention you assumed — that is normally worth the mark.

Extending EAT to multiple levels

The formula nests. For an L1/L2/main-memory hierarchy:

$$EAT = HR_1 T_1 + (1 - HR_1) [HR_2 T_2 + (1 - HR_2) T_m]$$

The bracketed term is just “the EAT of everything below L1”, which is why the recursive framing in Chapter 23 is useful.

Example. $T_1 = 1$ ns, $HR_1 = 0.95$; $T_2 = 10$ ns, $HR_2 = 0.80$; $T_m = 100$ ns.

$$EAT = 0.95(1) + 0.05[0.80(10) + 0.20(100)] = 0.95 + 0.05(28) = 0.95 + 1.4 = \mathbf{2.35} \text{ ns}$$

Without the L2, the same L1 would give $0.95 + 0.05(100) = 5.95$ ns — so the second level more than halves the average.

24.5 The three key design decisions

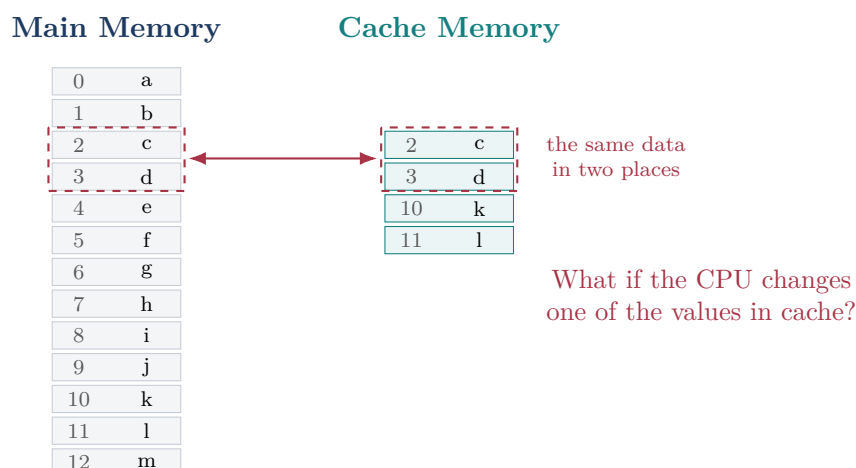
What a cache designer must choose

1. **Cache volume** — its capacity, or size.
2. **Write policy** — write back, or write through.
3. **Mapping scheme** — direct mapped, or set associative (and if associative, a line replacement policy).

Decision 1 is the straightforward trade-off already discussed: more capacity gives a higher hit ratio but costs more and is slower to search. Decisions 2 and 3 are the substance of the rest of this chapter and the next.

24.6 Write policy

24.6.1 The consistency problem



Why a write policy is needed

- Cache is **redundant** storage — the same data appears in two locations.
- Data may be **shared across processes**.

- Maintaining consistency is a challenge when the same data appears in multiple locations.
- **Inconsistency** arises the moment data is altered (written).
- The cache therefore needs a **write policy** to ensure consistency between cache and main memory.

24.6.2 Write through

Write through

Whenever a write operation happens, it is performed on **both** the cache and main memory **synchronously**.



Advantage	Disadvantage
Comparatively simple to implement.	Bus traffic will be increased — every single write goes to memory.
Memory is always up to date, which simplifies sharing between cores and devices.	Writes run at memory speed, not cache speed.

24.6.3 Write back

Write back

Write operations on a data item are performed **only on the cache**. But whenever the data is **evicted** from the cache, it is written to main memory.



Advantage	Disadvantage
Lazy writes — repeated writes to the same location cost only one memory write, at eviction.	Complex to implement.
Much less bus traffic.	Memory is temporarily stale, so other cores or DMA devices can read out-of-date values.

The dirty bit

A write-back cache adds one more bit per entry: the **dirty bit**, set when the entry is written. On eviction:

- dirty bit = 1 $\Rightarrow$ the line differs from memory, so write it back;
- dirty bit = 0 $\Rightarrow$ the line is unchanged, so simply discard it.

Without the dirty bit, every eviction would need a memory write and much of the policy's advantage would be lost.

Worked Example 4 *Comparing the two policies on the same access pattern*

A program writes to the same variable 100 times, then the line is evicted. Assume a cache write costs 1 unit and a memory write costs 10 units.

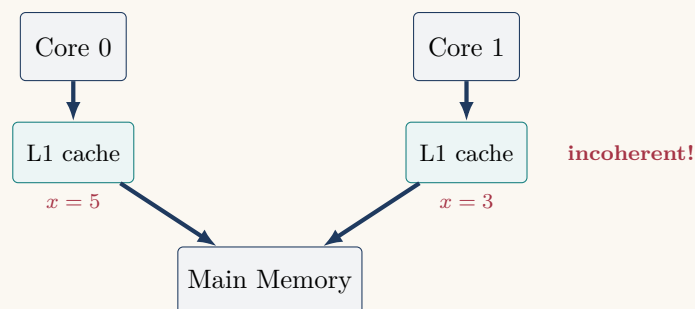
	Write through	Write back
Cache writes	$100 \times 1 = 100$	$100 \times 1 = 100$
Memory writes	$100 \times 10 = 1000$	$1 \times 10 = 10$
Total	1100	110

A tenfold difference, arising entirely from temporal locality in the write stream. Now consider the opposite pattern — 100 writes to 100 *different* lines, each evicted once. Both policies perform 100 memory writes and the advantage disappears entirely.

Conclusion: write back wins whenever writes are clustered, which they usually are. Most modern caches use write back, and accept the complexity.

Cache coherence — the reading exercise in Lecture 4.1.1

The consistency problem gets substantially harder with more than one core, because each core has its own private L1 cache.



The problem. Core 0 and Core 1 both cache the variable x . Core 0 writes 5; Core 1 writes 3. Each core's cache is internally consistent but they disagree with each other, and a subsequent read returns whichever value the reading core happens to hold.

Cache coherence is the set of protocols that prevents this. The classic approach is **snooping**: every cache watches the shared bus, and when it sees another cache write to an address it holds, it invalidates its own copy. The best-known family of protocols is **MESI**, in which every cache line is in one of four states:

M — Modified	This cache has the only copy, and it has been changed. Memory is stale.
E — Exclusive	This cache has the only copy and it matches memory.
S — Shared	Several caches may hold this line; all match memory.
I — Invalid	This entry does not hold usable data.

The invariant enforced is simple: *a line may be in state M in at most one cache at a time.* Everything else follows from maintaining it.

24.7 Summary

Term	Meaning
Cache	Small fast memory on the CPU chip holding data from a larger slower memory.
Cache hierarchy	L1 / L2 / L3, increasing in size and latency.
Cache line	Fixed-size block transferred between memory and cache.
Cache entry	Stored line <i>plus</i> its tag (and valid / dirty bits).
Tag	Identifies which memory block an entry holds.
Valid bit	Marks whether an entry has ever been filled.
Dirty bit	Marks whether a write-back entry differs from memory.
Hit / miss	Requested data found / not found in the cache.
Hit ratio HR	Probability of a hit; miss ratio = $1 - HR$.
EAT	$HR \times T_c + (1 - HR) \times T_m$.
Write through	Write to cache and memory together; simple, more bus traffic.
Write back	Write to cache only; write to memory on eviction; lazy, complex.
Cache coherence	Keeping multiple cores' caches consistent (e.g. MESI).

Cache Mapping Schemes

Where this fits

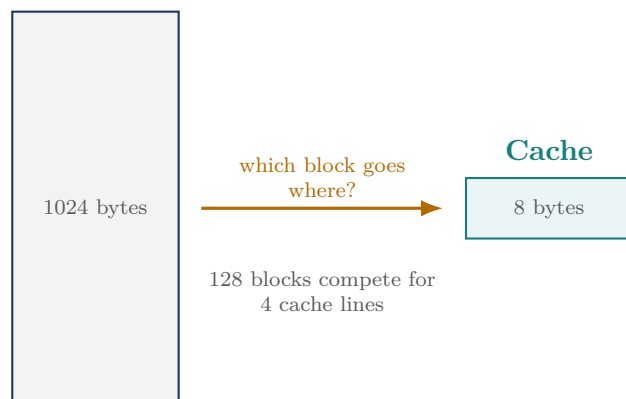
The third and largest cache design decision: given that the cache is far smaller than main memory, *where* in the cache may a given memory block be placed? The answer determines how addresses are split into fields, how a lookup works, and whether a replacement policy is needed at all. Source: Lecture 4.1.1.

25.1 Why a mapping scheme is needed

The problem

- The CPU refers to data or instructions **by their addresses**.
- Data is first searched for in the cache.
- Thus cached data must be **accessible using addresses**.
- But the cache is *much* smaller than main memory.
- Cache memory locations therefore **cannot** be given main memory addresses.
- A **mapping scheme** is required.

Main memory



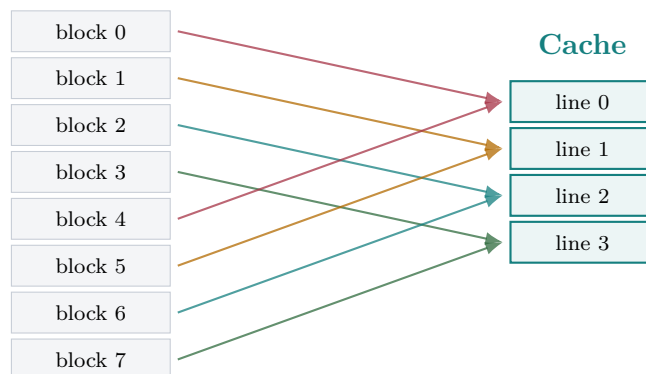
Scheme	Rule
Direct mapped	Each memory block has exactly <i>one</i> permitted cache position.
Fully associative	A block may go <i>anywhere</i> in the cache.
Set associative	A block is restricted to one <i>set</i> , but may go anywhere within that set.

25.2 Direct mapped cache

Direct mapped cache

- Memory is identified as **blocks of fixed size**.
- Each block has a **pre-defined cache entry position**.
- A block in memory will **always** map to the same cache entry position.
- The number of blocks in main memory is much larger than the number of cache entries, so **several memory blocks map to the same cache position**.
- A **tag** is used to identify which of those blocks is currently stored.

Memory blocks



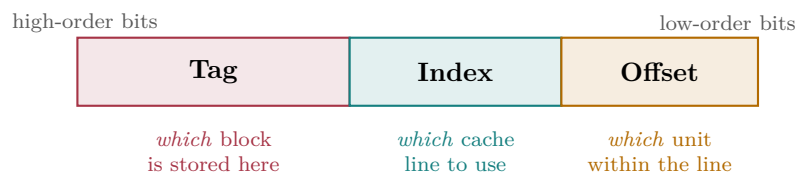
block i always maps to line $(i \bmod 4)$

The mapping rule

$$\text{cache line} = (\text{memory block number}) \bmod (\text{number of cache lines})$$

Because the number of lines is a power of two, that modulo is simply *taking the low-order bits of the block number* — which is why the address splits into fields with no arithmetic at all.

25.2.1 Splitting the address



The three field-width formulas — memorise these

$$\text{offset bits} = \log_2 \left(\frac{\text{cache line size}}{\text{size of one addressable unit}} \right) \quad \text{addressable units per line}$$

$$\text{index bits} = \log_2 \left(\frac{\text{cache volume}}{\text{cache line size}} \right) \quad \text{number of cache lines}$$

$$\text{address bits} = \log_2 \left(\frac{\text{memory size}}{\text{size of one addressable unit}} \right) \quad \text{number of addresses}$$

$$\text{tag bits} = \text{address bits} - \text{index bits} - \text{offset bits}$$

The single most common mistake

Every division must be in the **same units**, and “addressable unit” means *whatever one address refers to* — a byte in a byte-addressable machine, a word in a word-addressable one. If the memory is word addressable with 32-bit words, then one address covers 4 bytes, so a 32 kB cache line holds $32\text{kB}/4\text{B} = 2^{13}$ *addressable units*, not 2^{15} . Getting this wrong shifts the offset field by two bits and every subsequent answer is wrong.

25.2.2 A worked layout

Take the lecture’s running example:

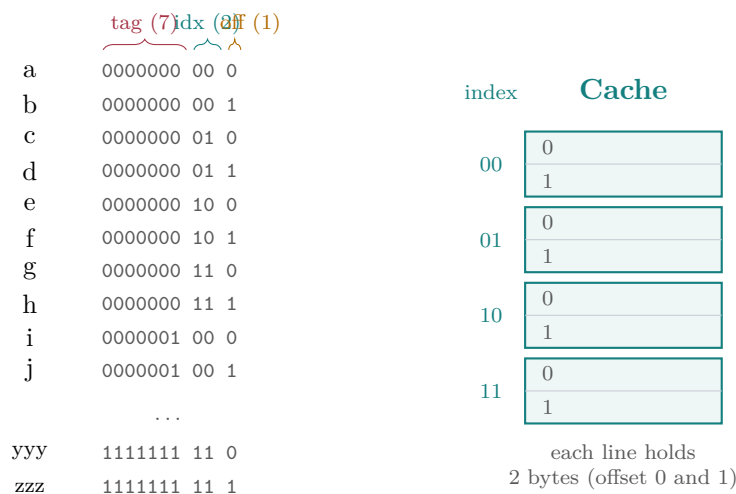
- Byte addressable main memory, total 1024 bytes
- 2 bytes per cache line
- Cache volume = 8 bytes

$$\text{offset} = \log_2(2/1) = \log_2 2 = \mathbf{1} \text{ bit}$$

$$\text{index} = \log_2(8/2) = \log_2 4 = \mathbf{2} \text{ bits}$$

$$\text{address} = \log_2(1024/1) = \log_2 1024 = \mathbf{10} \text{ bits}$$

$$\text{tag} = 10 - 2 - 1 = \mathbf{7} \text{ bits}$$



Halving the cache halves the index

If the cache volume were 4 bytes instead of 8, there would be only $4/2 = 2$ lines, so index = 1 bit and tag = 8 bits. Capacity and tag width trade off directly: a smaller cache needs *more* tag bits per line, because more memory blocks compete for each position.

25.2.3 Worked example: a conflict

The example below is the one to remember, because it shows the exact weakness that motivates every other mapping scheme.

Worked Example 1 *Reading addresses 1, 4 and 8 — Lecture 4.1.1*

Using the layout above (tag 7 | index 2 | offset 1):

Address	Binary	Tag	Index	Offset
1	0000000 00 1	00000000	00	1
4	0000000 10 0	00000000	10	0
8	0000001 00 0	00000001	00	0

Step 1 — CPU reads address 1. Index 00, so cache line 0. The line is empty $\Rightarrow$ **miss**. The whole 2-byte block (addresses 0 and 1, holding **a** and **b**) is loaded into line 00 with tag 00000000.

Step 2 — CPU reads address 4. Index 10, so cache line 2. Empty $\Rightarrow$ **miss**. Block (addresses 4, 5 — **e**, **f**) loaded into line 10 with tag 00000000.

	Tag	Data	
00	00000000	a b	
01			
10	00000000	e f	
11			

After reading addresses 1 and 4.

Step 3 — CPU reads address 8. Index 00 — *the same line as address 1*. Line 0 is occupied, and its stored tag 00000000 does **not** match the required tag 00000001 $\Rightarrow$ **miss**. The existing entry is **evicted** and replaced with the block containing addresses 8 and 9 (**i**, **j**), tag 00000001.

00	00000001	i j	a,b evicted
01			
10	00000000	e f	
11			

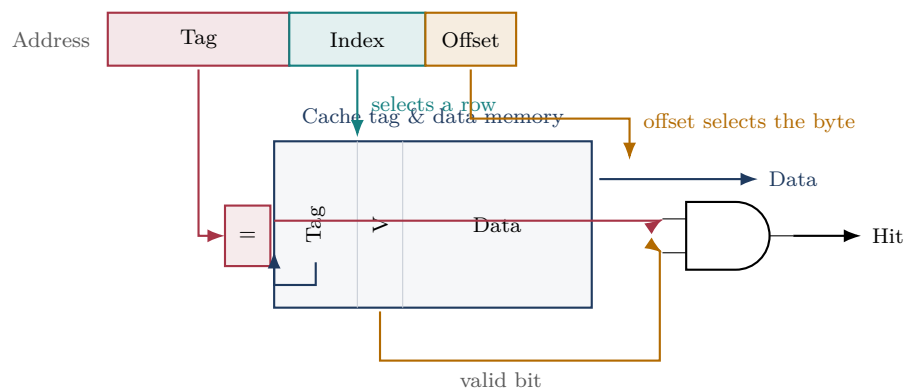
After reading address 8.

The point. Addresses 1 and 8 are far apart in memory but collide in the cache, because they share the same index. Even though three of the four cache lines sit empty, the block from address 1 had to go. This is a **conflict miss**, and it is the defining weakness of direct mapping.

25.2.4 How a lookup works

Search procedure

1. Use the **Low Order Address Bits** (LOAB) of the address — the index — to select an entry in the cache tag memory.
2. Use a **comparator** to check whether the stored tag matches the **High Order Address Bits** of the address.
3. If the tag matches, check the **valid bit**:
 - valid bit = 1 $\Rightarrow$ **cache hit**
 - valid bit = 0 $\Rightarrow$ **cache miss**
4. If the tag does not match $\Rightarrow$ **cache miss**.



Why direct mapping is fast

Only **one** comparator is needed, and the index picks the row before the comparison starts. Contrast this with a fully associative cache, where every entry must be compared simultaneously.

25.3 Field-width exercises

Worked Example 2 Exercise 1 — byte addressable

- 2 MB direct-mapped cache
- Cache line size 32 kB
- Byte addressable 1 GB memory

Find the lengths of the tag, index and offset.

Offset — how many addressable units within a cache line?

$$\frac{32 \text{ kB}}{1 \text{ byte}} = \frac{2^5 \times 2^{10} \text{ bytes}}{2^0 \text{ bytes}} = 2^{15} \Rightarrow \mathbf{15 \text{ bits}}$$

Index — how many cache entries can we create?

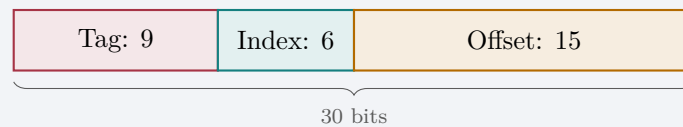
$$\frac{2 \text{ MB}}{32 \text{ kB}} = \frac{2 \times 2^{20} \text{ bytes}}{2^5 \times 2^{10} \text{ bytes}} = \frac{2^{21}}{2^{15}} = 2^6 \Rightarrow \mathbf{6 \text{ bits}}$$

Address length — how many addresses are there?

$$\frac{1 \text{ GB}}{1 \text{ byte}} = 2^{30} \Rightarrow \mathbf{30 \text{ bits}}$$

Tag:

$$30 - 6 - 15 = \mathbf{9 \text{ bits}}$$



Worked Example 3 *Exercise 2 — word addressable*

- 2 MB direct-mapped cache
- Cache line size 32 kB
- **Word** addressable 1 GB memory
- Word size = 32 bits = 4 bytes

The key phrase

“Word = smallest unit which has an address.” Every division must therefore be by 4 bytes, not 1.

Offset — how many addressable units within a cache line?

$$\frac{32 \text{ kB}}{32 \text{ bits}} = \frac{32 \times 2^{10} \text{ bytes}}{4 \text{ bytes}} = \frac{2^{15} \text{ bytes}}{2^2 \text{ bytes}} = 2^{13} \Rightarrow \mathbf{13 \text{ bits}}$$

Index — how many cache entries can we create?

$$\frac{2 \text{ MB}}{32 \text{ kB}} = \frac{2 \times 2^{20} \text{ bytes}}{2^5 \times 2^{10} \text{ bytes}} = \frac{2^{21} \text{ bytes}}{2^{15} \text{ bytes}} = 2^6 \Rightarrow \mathbf{6 \text{ bits}}$$

Minimum address length — how many addresses do we have?

$$\frac{1 \text{ GB}}{32 \text{ bits}} = \frac{2^{30} \text{ bytes}}{4 \text{ bytes}} = 2^{28} \Rightarrow \mathbf{28 \text{ bits}}$$

Tag — this is the value the slide leaves as “?”:

$$28 - 6 - 13 = \mathbf{9 \text{ bits}}$$



Sanity check. The index is unchanged between the two exercises, which makes sense — the number of cache lines depends only on the cache and line sizes, not on how memory is addressed. Both the offset and the address length shrank by exactly 2 bits ($\log_2 4$), so the tag is unchanged too.

25.3.1 Advantages and disadvantages of direct mapping

Left as homework in Lecture 4.1.1

Advantages	Disadvantages
Simplest scheme to implement — the index picks the line directly.	Conflict misses. Two hot blocks with the same index evict each other repeatedly, even when the rest of the cache is empty.
Fastest lookup — only one tag comparison, and it can start as soon as the index is decoded.	Thrashing. Alternating references to two conflicting addresses give a 0% hit ratio, which can be <i>worse</i> than having no cache at all.
Cheapest — one comparator, no replacement logic, no need to record usage order.	Hit ratio is sensitive to address alignment , so performance can change dramatically for reasons invisible in the source code.
Lowest power, since only one entry is read and compared.	No replacement <i>choice</i> exists — the hardware has no way to keep a valuable block.

The thrashing scenario in concrete terms. With the layout from Section 25.2, a loop that alternately reads address 1 and address 8 misses *every single time*: each reference evicts the block the next reference needs. The fix is either a different mapping scheme, or padding the data structures so the hot addresses no longer collide.

25.4 Fully associative cache

Fully associative cache

The opposite of direct mapping. There are **no restrictions** — a cache line can be placed **anywhere available**.

Advantage	Disadvantage
The most flexible mapping scheme — no conflict misses at all; a block is evicted only when the whole cache is full.	Searching is expensive. There is no index to narrow the search, so <i>every</i> entry's tag must be compared, which needs one comparator per entry.

Address split for a fully associative cache

There is **no index field** — any block may occupy any line, so nothing selects a line in advance:

$$\text{Address} = \underbrace{\text{Tag}}_{\text{everything else}} \mid \underbrace{\text{Offset}}_{\text{within the line}}$$

The tag is correspondingly wider, which also makes the tag memory larger.

25.5 Set associative cache

Set associative cache

A **hybrid** of fully associative and direct mapped caches.

- The cache is divided into **equal-sized sets**.
- A cache line can be mapped into **only one set** — directly mapped to a set.
- Within the mapped set, the cache line can be placed **anywhere available** — fully associative within a set.

If a set in the cache can accommodate N cache lines, then it is called an “ N -Way Set Associative Cache”.

25.5.1 The associativity spectrum

Worked Example 4 *Recognising N — Lecture 4.1.1 exercise*

Each small box is a cache line and the heavy divisions show the sets. All four caches hold 8 lines.

1-way associative



8 sets

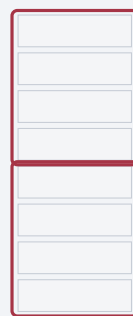
**Direct Mapped
Cache**

2-way associative



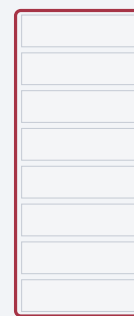
4 sets

4-way associative



2 sets

8-way associative

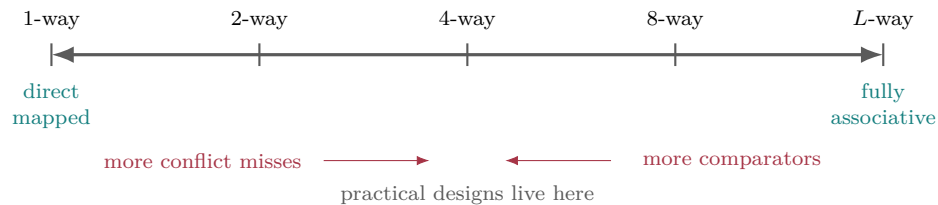


1 set

**Fully Associative
Cache**

Sets	Lines per set	Name	Also known as
8	1	1-way	direct mapped
4	2	2-way	—
2	4	4-way	—
1	8	8-way	fully associative

The insight worth taking away. Direct mapped and fully associative are not separate schemes competing with set associative — they are the two *endpoints* of a single spectrum. A direct mapped cache is 1-way set associative; a fully associative cache with L lines is L -way set associative. Everything in between trades lookup cost against conflict misses.



25.5.2 Address split for set associative



Only one formula changes

The index now selects a *set* rather than a line:

$$\text{index bits} = \log_2(\text{number of sets}) = \log_2\left(\frac{\text{cache volume}}{\text{line size} \times N}\right)$$

Offset and address-length formulas are unchanged, and the tag is still “whatever is left over”. Doubling the associativity halves the number of sets, so it moves exactly one bit from the index to the tag.

25.5.3 Worked example: 2-way set associative

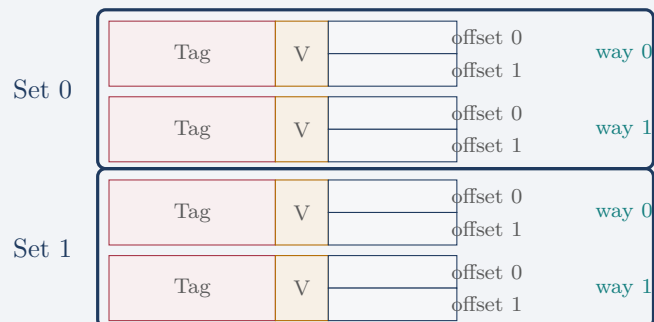
Worked Example 5 The same memory, now 2-way — Lecture 4.1.1

- Byte addressable, 2-way set associative
- Total 1024 bytes in memory
- 2 bytes per cache line
- Cache volume = 8 bytes

Working out the layout.

- 2-way $\Rightarrow$ within a single set there is room for **two** cache lines, so the set size = 4 bytes.
- Cache volume is 8 bytes, so there will be only **two sets**.
- A cache line placed in a set is identified by a **tag**.
- Within a cache entry, each unit of memory is identified by the **offset**.

$$\begin{aligned}\text{offset} &= \log_2(2/1) = \mathbf{1} \text{ bit} \\ \text{set index} &= \log_2(2 \text{ sets}) = \mathbf{1} \text{ bit} \\ \text{address} &= \log_2(1024) = \mathbf{10} \text{ bits} \\ \text{tag} &= 10 - 1 - 1 = \mathbf{8} \text{ bits}\end{aligned}$$

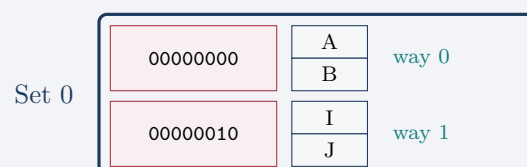


Now replay the direct-mapped conflict. With direct mapping, referring to addresses 1 and then 8 forced a replacement. What happens here?

Address	Binary	Tag	Set	Offset
1	00000000 0 1	00000000	0	1
8	00000010 0 0	00000010	0	0
4	00000001 0 0	00000001	0	0

CPU refers address 1. The mapping set is 0. It can be placed anywhere within set 0 (assuming the cache is empty at the beginning), so the block A,B is loaded with tag 00000000.

CPU refers address 8. The mapping set is again 0 — but it can **still be placed**, because set 0 can accommodate another cache line. The block I,J is loaded into the second way with tag 00000010.



No eviction.

Both blocks coexist — the conflict that broke the direct-mapped cache is gone.

But if the CPU now refers address 4 or 5, there *will* be an eviction. The mapping set is still 0, and both ways are now occupied, so one cache entry must be replaced with the new cache line.



Which cache entry should be evicted?

That question — which the direct-mapped cache never had to ask, because there was only one candidate — is what replacement policies answer.

25.6 Cache replacement policies

Why a policy is needed

When implementing set associative caches we must affirm a **cache replacement policy**: *which cache entry should be evicted when the set is fully occupied?*

This decision should be taken in alignment with the **locality of reference** that we are trying to exploit.

25.6.1 The common policies

Policy	Rule
Least Recently Used (LRU)	The cache entry which is unused for the longest time will be evicted.
First In First Out (FIFO)	The cache entry which has the longest period of stay will be evicted.
Least Frequently Used (LFU)	The cache line with the least access frequency will be evicted.
Random	Evict an arbitrary entry.

LRU and FIFO are not the same thing

FIFO measures time since the block *arrived*. LRU measures time since the block was last *used*. A block that arrived long ago but is read on every iteration is the *first* candidate for FIFO and the *last* candidate for LRU — which is exactly the case where FIFO performs badly.

25.6.2 Worked comparison

Worked Example 6 *The same reference string under three policies*

A set holds **3** lines. The CPU references blocks in this order:

A, B, C, A, B, D, A, B, D, A, C, D, A

FIFO — evict whichever block arrived earliest.

Reference	A	B	C	A	B	D	A	B	D	A	C	D	A
Hit/Miss	M	M	M	H	H	M	M	M	H	H	M	M	M
Evicted	–	–	–	–	–	A	B	C	–	–	D	A	B

$$\text{Hits} = 4/13 \approx 31\%$$

LRU — evict whichever block was used least recently.

Reference	A	B	C	A	B	D	A	B	D	A	C	D	A
Hit/Miss	M	M	M	H	H	M	H	H	H	H	M	H	H
Evicted	–	–	–	–	–	C	–	–	–	–	B	–	–

$$\text{Hits} = 8/13 \approx 62\%$$

LFU — evict whichever block has been accessed fewest times.

Reference	A	B	C	A	B	D	A	B	D	A	C	D	A
Hit/Miss	M	M	M	H	H	M	H	H	H	H	M	M	H
Evicted	–	–	–	–	–	C	–	–	–	–	D	C	–

$$\text{Hits} = 7/13 \approx 54\%$$

Policy	Hits	Hit ratio
FIFO	4	30.8%
LFU	7	53.8%
LRU	8	61.5%

Why FIFO does so badly here. At the sixth reference, all three policies must evict something. FIFO throws out A — the block that arrived first, but which is also the *most heavily used* block in the whole string. A is requested again immediately, causing a cascade of further misses. LRU and LFU both correctly identify C as the least valuable block.

25.6.3 Advantages and disadvantages

“Discuss advantages and disadvantages” — Lecture 4.1.1 homework

Policy	Advantages	Disadvantages
LRU	Directly exploits <i>temporal</i> locality, so it usually gives the best hit ratio of the practical policies. Well understood and predictable.	Expensive in hardware: the usage order of all N ways must be recorded and updated on <i>every</i> access. Cost grows quickly with associativity, so true LRU is rare beyond 4-way.
FIFO	Very cheap — a single counter per set, no update needed on a hit.	Ignores usage entirely, so it can evict the hottest block. Also suffers Bélády’s anomaly : a <i>larger</i> cache can produce <i>more</i> misses.
LFU	Protects genuinely popular blocks well; good where a stable working set is accessed repeatedly.	Needs a counter per line. Suffers cache pollution : a block that was popular long ago keeps a high count and resists eviction even after it stops being used. Poor for phase-changing workloads.
Random	Cheapest of all — no state whatsoever. Immune to pathological access patterns, since an adversary cannot predict it.	Ignores locality, so on average slightly worse than LRU — though in practice the gap is surprisingly small, which is why real high-associativity caches often use random or pseudo-LRU.

Pseudo-LRU is the usual engineering compromise: a small tree of bits approximates the LRU ordering with a fraction of the hardware. For an 8-way set, true LRU needs to order 8 items ($\log_2 8! \approx 16$ bits), while tree-based pseudo-LRU needs only 7.

25.7 Comparing the three mapping schemes

	Direct mapped	Set associative	Fully associative
Placement	one fixed line	anywhere in one set	anywhere
Address fields	tag index offset	tag set offset	tag offset
Comparators	1	N	one per line
Replacement policy	not needed	required	required
Conflict misses	many	few	none
Lookup speed	fastest	moderate	slowest
Hardware cost	lowest	moderate	highest
Typical use	simple caches	almost all real caches	TLBs, small buffers

What real processors do

Set associative, essentially always. A common arrangement is 8-way L1 caches and 16-way L3 caches, with pseudo-LRU replacement. Fully associative organisation survives only where the structure is small enough that comparing every entry is affordable — most notably in the **TLB**, the small cache of address translations used by the MMU (Chapter 21).

25.8 Summary

Concept	Key point
Mapping scheme	Rule deciding where a memory block may sit in the cache.
Direct mapped	One permitted line per block; $\text{line} = \text{block} \bmod (\text{number of lines})$.
Fully associative	Any line; no index field; needs a comparator per entry.
Set associative	Directly mapped to a set, fully associative within it.
N -way	N lines per set; 1-way = direct mapped, L -way = fully associative.
Offset bits	$\log_2(\text{addressable units per cache line})$.
Index bits	$\log_2(\text{number of lines})$, or number of <i>sets</i> if associative.
Address bits	$\log_2(\text{number of addressable units in memory})$.
Tag bits	$\text{address} - \text{index} - \text{offset}$.
Conflict miss	A miss caused by mapping restrictions, not by capacity.
LRU	Evict least recently used; best hit ratio, costly hardware.
FIFO	Evict oldest arrival; cheap, can evict hot blocks.
LFU	Evict least frequently used; risks cache pollution.
Random	No state; surprisingly competitive at high associativity.

Exam procedure for any field-width question

1. Write down the **size of one addressable unit** (byte or word) — everything else depends on it.
2. $\text{Offset} = \log_2(\text{line size} \div \text{unit size})$.
3. Count the lines: $\text{cache volume} \div \text{line size}$. If N -way, divide again by N to get the number of *sets*. $\text{Index} = \log_2(\text{that})$.
4. $\text{Address length} = \log_2(\text{memory size} \div \text{unit size})$.
5. $\text{Tag} = \text{address} - \text{index} - \text{offset}$.
6. Check the three fields add up to the address length.

PART VII

Appendices

Appendix A

Formula and Rule Sheet

How to use this appendix

Everything that has to be recalled rather than derived, in one place. If you can reproduce this appendix from memory you can reconstruct most of the course.

A.1 Counting and sizing

n bits	2^n distinct patterns
k distinct values	need $\lceil \log_2 k \rceil$ bits
Truth table for n variables	2^n rows
K-map for n variables	2^n cells
Decoder with n inputs	2^n outputs
Encoder with 2^n inputs	n outputs
MUX with n select lines	2^n data inputs
n -bit address	2^n addressable locations, range 0 to $2^n - 1$
Memory size	$2^n \times$ (size of one addressable unit)

A.2 Number representation

Scheme	Range for n bits	$n = 8$
Unsigned	$0 \dots 2^n - 1$	$0 \dots 255$
Signed magnitude	$-(2^{n-1}-1) \dots + (2^{n-1}-1)$	$-127 \dots + 127$
One's complement	$-(2^{n-1}-1) \dots + (2^{n-1}-1)$	$-127 \dots + 127$
Two's complement	$-2^{n-1} \dots + (2^{n-1}-1)$	$-128 \dots + 127$

One's complement of x	invert every bit: $\bar{x}$
Two's complement of x	$\bar{x} + 1$
Two's complement shortcut	copy from the right up to and including the first 1, then invert
One's complement addition	add, then add any carry-out back in (end-around carry)
Two's complement addition	add, then discard any carry-out
Overflow (2's comp.)	$V = C_{n-1} \oplus C_n$, or: same-signed operands give an opposite-signed result
Sign extension	replicate the sign bit into the new high-order positions

A.3 Floating point

Normalised form	$\pm 1.xxxx \times 2^e$; the leading 1 is <i>hidden</i>
Excess- K encode	stored = value + K
Excess- K decode	value = stored - K
Usual K for n bits	$K = 2^{n-1} - 1$ (or 2^{n-1})
Excess- K minimum	$-K$
Excess- K maximum	$(2^n - 1) - K$
Round-off error	exact value - stored value

Format	Sign	Exponent	Significand
Course 8-bit	1	3 (Excess-3)	4
IEEE 754 single	1	8 (Excess-127)	23
IEEE 754 double	1	11 (Excess-1023)	52

IEEE 754 single: usable exponent $-126 \leq e \leq +127$; field 00000000 reserved for zero and subnormals, field 11111111 for infinity and NaN.

A.4 Boolean algebra

Law	AND form	OR form
Annulment	$A \cdot 0 = 0$	$A + 1 = 1$
Identity	$A \cdot 1 = A$	$A + 0 = A$
Idempotent	$A \cdot A = A$	$A + A = A$
Complement	$A \cdot \bar{A} = 0$	$A + \bar{A} = 1$
Commutative	$AB = BA$	$A + B = B + A$
Associative	$(AB)C = A(BC)$	$(A+B)+C = A+(B+C)$
Distributive	$A(B+C) = AB + AC$	$A + BC = (A+B)(A+C)$
Absorption	$A(A+B) = A$	$A + AB = A$
Absorption var.	$A(\bar{A}+B) = AB$	$A + \bar{A}B = A + B$
Combining	$(A+B)(\bar{A}+B) = B$	$AB + \bar{A}B = B$
Double negation	$\bar{\bar{A}} = A$	
De Morgan	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A+B} = \bar{A} \cdot \bar{B}$

Operator precedence: () > NOT > AND > OR.

Duality: swapping every AND $\leftrightarrow$ OR and 0 $\leftrightarrow$ 1 turns each law into its partner.

A.5 Gate truth tables

Gate	Expression	00	01	10	11
AND	$A \cdot B$	0	0	0	1
OR	$A + B$	0	1	1	1
XOR	$A \oplus B$	0	1	1	0
NAND	$\overline{A \cdot B}$	1	1	1	0
NOR	$\overline{A + B}$	1	0	0	0
XNOR	$\overline{A \oplus B}$	1	0	0	1

A.6 Canonical forms

Minterm	AND of all variables; plain if the bit is 1 ; 1 in one row only
Maxterm	OR of all variables; plain if the bit is 0 ; 0 in one row only
$\sum m(\dots)$	row indices where $F = 1$
$\prod M(\dots)$	row indices where $F = 0$
Relationship	the two index sets are complementary; $M_i = \overline{m_i}$
Choosing	more 0s than 1s $\Rightarrow$ SoM; more 1s than 0s $\Rightarrow$ PoM

A.7 Karnaugh map rules

1. Groups contain only 1s (for SoP) or only 0s (for PoS) — never mixed.
2. Groups are rectangular; horizontal and vertical only, never diagonal.
3. Group sizes must be 2^k : 1, 2, 4, 8, 16.
4. Each group must be as large as possible; overlap and wrap around the edges if that helps.
5. Use as few groups as possible while covering everything.
6. Don't-cares may be treated as 0 or 1 — but no group may be entirely don't-cares.

Label order is Gray code: 00, 01, 11, 10. A group of 2^k cells eliminates k variables. Read each group as the variables that stay constant.

A.8 Combinational building blocks

Circuit	First output	Second output
Half adder	Sum = $a \oplus b$	$C_{out} = ab$
Full adder	Sum = $a \oplus b \oplus C_{in}$	$C_{out} = ab + aC_{in} + bC_{in}$
Half subtractor	Diff = $a \oplus b$	$B_{out} = \overline{a}b$
Full subtractor	Diff = $a \oplus b \oplus B_{in}$	$B_{out} = \overline{a}b + \overline{a}B_{in} + bB_{in}$

Adder-subtractor: $A - B = A + \overline{B} + 1$; XOR each b_i with mode bit M and set $C_{in} = M$.

Functionally complete sets: {AND,OR,NOT}, {AND,NOT}, {OR,NOT}, {NAND}, {NOR}. NAND suits SoP; NOR suits PoS; both convert two-level expressions one-for-one.

A.9 Flip-flops

Type	Characteristic equation	Behaviour
S-R	$Q(t+1) = S + \overline{R}Q$ ($SR = 0$)	hold / reset / set / forbidden
D	$Q(t+1) = D$	follow the input
J-K	$Q(t+1) = J\overline{Q} + \overline{K}Q$	hold / reset / set / toggle
T	$Q(t+1) = T \oplus Q$	hold / toggle

Transition		S-R		J-K		D	T
$Q(t)$	$Q(t+1)$	S	R	J	K	D	T
0	0	0	$\times$	0	$\times$	0	0
0	1	1	0	1	$\times$	1	1
1	0	0	1	$\times$	1	0	1
1	1	$\times$	0	$\times$	0	1	0

Counter design rule: bit i toggles when all lower-order bits are 1, i.e. $J_i = K_i = \text{En} \cdot Q_0 Q_1 \cdots Q_{i-1}$.

A.10 Performance

Clock cycle time	$t = 1/r$
Clock rate	$r = 1/t$
Total clock cycles	$N \times \text{CPI}$
CPU time	$N \times \text{CPI} \times t = (N \times \text{CPI})/r$
Overall CPI	$\sum_i \text{CPI}_i \times F_i$
CPI from totals	total cycles $\div$ total instructions

Static instruction count = instructions written.

Dynamic instruction count = instructions executed — this is the N above.

A.11 Fetch cycle and pipelining

Fetch phase	$\text{MAR} \leftarrow \text{PC}; \text{IR} \leftarrow M[\text{MAR}]; \text{PC} \leftarrow \text{PC} + k$
k	addressable units per instruction (1 word-addr.; 2 or 4 byte-addr.)
Pipeline time	$(k + n - 1)t_p$ for n instructions, k stages
Pipeline speedup	$\frac{kn}{k + n - 1}$
Maximum speedup	k (approached as $n \rightarrow \infty$)

A.12 Memory and cache

Hit ratio	HR ; miss ratio = $1 - HR$
Effective access time	$EAT = HR \times T_c + (1 - HR) \times T_m$
Two-level EAT	$EAT = HR_1 T_1 + (1 - HR_1) [HR_2 T_2 + (1 - HR_2) T_m]$
Temporal locality	Recently used items are used again soon $\Rightarrow$ keep them.
Spatial locality	Nearby addresses are used together $\Rightarrow$ fetch whole blocks.
Write through	Write to cache and memory together.
Write back	Write to cache only; write to memory on eviction (dirty bit).

Cache address fields

Offset bits	$\log_2(\text{line size} \div \text{addressable unit size})$
Index bits (direct mapped)	$\log_2(\text{cache volume} \div \text{line size})$
Index bits (N -way)	$\log_2(\text{cache volume} \div (\text{line size} \times N))$
Address bits	$\log_2(\text{memory size} \div \text{addressable unit size})$
Tag bits	address – index – offset

Direct mapped: line = block number mod number of lines.

Fully associative: no index field — address is tag | offset.

1-way = direct mapped; L -way (all lines in one set) = fully associative.

LRU	Evict the entry unused for the longest time.
FIFO	Evict the entry resident for the longest time.
LFU	Evict the entry with the lowest access count.
Random	Evict an arbitrary entry.

A.13 Addressing modes

Mode	Operand located	Memory accesses
Immediate	in the instruction	0
Register	in a register	0
Direct	at the given address	1
Register indirect	at the address held in a register	1
Displacement	at (register) + offset	1
Indirect	at the address stored at the given address	2
Stack	top of stack	implicit

Appendix B

Reference Tables

B.1 Powers of two

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1 024

1 byte's worth
1 kB

n	2^n
11	2 048
12	4 096
13	8 192
14	16 384
15	32 768
16	65 536
17	131 072
18	262 144
19	524 288
20	1 048 576
30	—

2 bytes' worth
1 MB
1 GB

Worth memorising outright

$2^{10} = 1024 \approx 10^3$, $2^{20} \approx 10^6$, $2^{30} \approx 10^9$.

So $2^{32} = 4 \times 2^{30} \approx 4$ billion — which is why a 32-bit address bus reaches 4 GB.

B.2 Base conversion table

Decimal	Binary	Octal	Hex
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

B.3 Binary fractions

Position	Weight	Decimal
2^{-1}	$\frac{1}{2}$	0.5
2^{-2}	$\frac{1}{4}$	0.25
2^{-3}	$\frac{1}{8}$	0.125
2^{-4}	$\frac{1}{16}$	0.0625
2^{-5}	$\frac{1}{32}$	0.03125
2^{-6}	$\frac{1}{64}$	0.015625

B.4 Excess- K reference (4-bit fields)

Binary	Unsigned	Excess-7	Excess-8
0000	0	−7	−8
0001	1	−6	−7
0010	2	−5	−6
0011	3	−4	−5
0100	4	−3	−4
0101	5	−2	−3
0110	6	−1	−2
0111	7	0	−1
1000	8	1	0
1001	9	2	1
1010	10	3	2
1011	11	4	3
1100	12	5	4
1101	13	6	5
1110	14	7	6
1111	15	8	7

B.5 BCD and Excess-3

Decimal	BCD	Excess-3
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

B.6 Minterms and maxterms (three variables)

x	y	z	Minterm m_i	Maxterm M_i
0	0	0	$\bar{x}\bar{y}\bar{z}$	$x + y + z$
0	0	1	$\bar{x}\bar{y}z$	$x + y + \bar{z}$
0	1	0	$\bar{x}y\bar{z}$	$x + \bar{y} + z$
0	1	1	$\bar{x}yz$	$x + \bar{y} + \bar{z}$
1	0	0	$x\bar{y}\bar{z}$	$\bar{x} + y + z$
1	0	1	$x\bar{y}z$	$\bar{x} + y + \bar{z}$
1	1	0	$xy\bar{z}$	$\bar{x} + \bar{y} + z$
1	1	1	xyz	$\bar{x} + \bar{y} + \bar{z}$

B.7 K-map cell indices

The minterm index of each cell, for the standard layouts used in these notes.

Four variables (ab across, cd down)

Three variables (xy across, z down)

		xy			
		00	01	11	10
z	0	0	2	6	4
	1	1	3	7	5

		ab			
		00	01	11	10
cd	00	0	4	12	8
	01	1	5	13	9
	11	3	7	15	11
	10	2	6	14	10

How to read these

For the four-variable map with variables (a, b, c, d) and a as the most significant bit, the index of a cell is

$$i = 8a + 4b + 2c + d$$

Do not memorise the grids — compute the index from the row and column labels. Note how neighbouring cells always differ by a single bit, as required.

B.8 Example machine instruction set

Opcode	Instruction	Format	Meaning
0_{16}	<i>spare</i>	—	available
1_{16}	L R, A	Type 2	$R \leftarrow M[A]$
2_{16}	LI R, I	Type 1	$R \leftarrow I$
3_{16}	ST R, A	Type 2	$M[A] \leftarrow R$
4_{16}	LR R1, R2	Type 4	$R1 \leftarrow R2$
5_{16}	ADD R0,R1,R2	Type 3	integer addition
6_{16}	AFP R0,R1,R2	Type 3	floating point addition
7_{16}	OR R0,R1,R2	Type 3	bitwise OR
8_{16}	AND R0,R1,R2	Type 3	bitwise AND
9_{16}	XOR R0,R1,R2	Type 3	bitwise XOR
A_{16}	RL R, I	Type 1	rotate left
B_{16}	RR R, I	Type 1	rotate right
C_{16}	SL R, I	Type 1	shift left
D_{16}	SR R, I	Type 1	shift right
E_{16}	JMP R, A	Type 2	jump to A if $R = R0$
F_{16}	HALT	—	stop execution

Format	Field layout (16 bits total)
Type 1	Opcode (4) Register (4) Immediate (8)
Type 2	Opcode (4) Register (4) Address (8)
Type 3	Opcode (4) Reg (4) Reg (4) Reg (4)
Type 4	Opcode (4) Unused 0000 Reg (4) Reg (4)

Appendix C

Remaining Tutorial Exercises, Solved

What is here

Exercises from the tutorials that were not worked through in the main chapters, with full solutions. Exercises that *were* worked through appear in place — see the index of worked examples below.

Exercise	Source	Solved in
Number system conversions	Tutorial 01	Section 2.8
Drawing logic circuits	Tutorial 03	Section 6.8
Boolean law proofs	Tutorial 04	Chapter 8
Simplification examples 1–3	Tutorial 04	Section 8.6
K-map SoP / PoS	Tutorial 06	Sections 9.6, 9.7
Don't-care grouping	Tutorial 06	Section 9.8
Fire alarm (all three parts)	Tutorial 06	Sections 9.10, 10.7
BCD to Excess-3 converter	Lecture 2.3.1	Section 11.4
3×8 from two 2×4 decoders	Lecture 2.3.1	Section 13.1.3
Full subtractor from a decoder	Lecture 2.3.1	Section 13.1.4
8×3 encoder	Lecture 2.3.1	Section 13.2.2
Demultiplexers	Lecture 2.3.1	Section 13.4
Uses of decoders and MUXes	Lecture 2.3.1	Section 13.5
CPI and machine comparison	Lecture 3.1	Chapter 18
Addressing mode activity	Lecture 3.3	Section 21.4
Pipelining advantages / hazards	Lecture 3.3	Section 22.5
Overflow and underflow	Lecture 3.2.1	Section 5.8
IEEE 754 ranges	Lecture 3.2.1	Section 5.9.4
Shift registers	Lecture 2.4.1	Section 16.2
Two-bit multiplier	Lecture 2.3.1	Section 12.7
3D XPoint	Lecture 4.1.1	Section 23.2.6
Cache coherence	Lecture 4.1.1	Section 24.6.4
Tag / index / offset exercises	Lecture 4.1.1	Section 25.3
Direct mapped pros and cons	Lecture 4.1.1	Section 25.2.5
N -way identification	Lecture 4.1.1	Section 25.5.1
Replacement policy pros and cons	Lecture 4.1.1	Section 25.6.3

C.1 Tutorial 03 — truth tables

Worked Example 1 $A(B + C)$

A	B	C	$B+C$	$A(B+C)$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	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

$F = \sum m(5, 6, 7)$ — one when A is set and at least one of B, C is set.

Worked Example 2 $(A + B) \cdot \overline{(C \cdot A)}$

A	B	C	$A+B$	$C \cdot A$	$\overline{C \cdot A}$	F
0	0	0	0	0	1	0
0	0	1	0	0	1	0
0	1	0	1	0	1	1
0	1	1	1	0	1	1
1	0	0	1	0	1	1
1	0	1	1	1	0	0
1	1	0	1	0	1	1
1	1	1	1	1	0	0

$F = \sum m(2, 3, 4, 6)$.

Worked Example 3 $Z = A \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot C$

A	B	C	$B \cdot C$	$\overline{B \cdot C}$	$A(\overline{B \cdot C})$	$\overline{A}(B \cdot C)$	Z
0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
0	1	1	1	0	0	1	1
1	0	0	0	1	1	0	1
1	0	1	0	0	0	0	0
1	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0

$Z = \sum m(3, 4)$ — exactly the answer obtained algebraically in Worked Example 8.3.

Worked Example 4 $(A \oplus B) \oplus C$

A	B	C	$A \oplus B$	$(A \oplus B) \oplus C$
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	0	1

$F = \sum m(1, 2, 4, 7)$ — **odd parity**: the output is 1 when an odd number of inputs are 1. Compare with the full adder's Sum output (Section 12.2); they are the same function.

C.2 Tutorial 05 — K-map simplification, validated algebraically**Worked Example 5** (a) $\overline{A}B + A\overline{B} + AB$ **K-map:**

		A		
		0	1	
B	0	0	1	— column $A=1 \Rightarrow A$ — row $B=1 \Rightarrow B$
	1	1	1	

$$F = A + B$$

Algebraic check:

$$\begin{aligned}
 \overline{A}B + A\overline{B} + AB &= \overline{A}B + A(\overline{B} + B) && \text{Distributive} \\
 &= \overline{A}B + A(1) && \text{Complement} \\
 &= \overline{A}B + A && \text{Identity} \\
 &= A + B && \text{Absorption variant}
 \end{aligned}$$

✓

Worked Example 6 (b) $\overline{A}B\overline{C} + \overline{A}BC + \overline{A}\overline{B}C + ABC$

The minterms are m_2, m_3, m_1, m_7 , so $F = \sum m(1, 2, 3, 7)$.

		AB				
		00	01	11	10	
C	0	0	1	0	0	— column $AB=01$: $A=0, B=1 \Rightarrow \overline{A}B$ — row $C=1$, cols 00, 01: $A=0 \Rightarrow \overline{A}C$ — row $C=1$, cols 01, 11: $B=1 \Rightarrow BC$
	1	1	1	1	0	

$$F = \overline{A}B + \overline{A}C + BC$$

All three groups are essential: the cell $(C=0, AB=01)$ is covered only by $\overline{A}B$, the cell $(C=1, AB=00)$ only by $\overline{A}C$, and $(C=1, AB=11)$ only by BC .

Algebraic check:

$$\begin{aligned}
 \overline{A}B\overline{C} + \overline{A}BC + \overline{A}\overline{B}C + ABC &= \overline{A}B(\overline{C} + C) + \overline{A}\overline{B}C + ABC \\
 &= \overline{A}B + \overline{A}\overline{B}C + ABC \\
 &= \overline{A}(B + \overline{B}C) + ABC && \text{Distributive} \\
 &= \overline{A}(B + C) + ABC && \text{Absorption variant} \\
 &= \overline{A}B + \overline{A}C + ABC \\
 &= \overline{A}B + \overline{A}BC + ABC + \overline{A}C && \text{re-adding } \overline{A}BC \subseteq \overline{A}B \\
 &= \overline{A}B + BC(\overline{A} + A) + \overline{A}C \\
 &= \overline{A}B + BC + \overline{A}C
 \end{aligned}$$

✓

Worked Example 7 (c) $BC + B\overline{C} + BA$

		AB			
		00	01	11	10
C	0	0	1	1	0
	1	0	1	1	0

One group of four — columns $AB = 01$ and 11 , both rows. The only unchanged variable is $B = 1$.

$$F = B$$

Algebraic check:

$$\begin{aligned}
 BC + B\overline{C} + AB &= B(C + \overline{C}) + AB && \text{Distributive} \\
 &= B + AB && \text{Complement, Identity} \\
 &= B && \text{Absorption}
 \end{aligned}$$

✓

Worked Example 8 (d) $AB + A(B + C) + B(B + C)$

Expand first:

$$AB + AB + AC + BB + BC = AB + AC + B + BC$$

		AB			
		00	01	11	10
C	0	0	1	1	0
	1	0	1	1	1

— cols 01, 11, both rows: $B=1 \Rightarrow B$

— row $C=1$, cols 11, 10: $A=1 \Rightarrow AC$

$$F = B + AC$$

Algebraic check:

$$\begin{aligned}
 AB + AC + B + BC &= (AB + B) + (BC) + AC && \text{regroup} \\
 &= B + BC + AC && \text{Absorption} \\
 &= B + AC && \text{Absorption}
 \end{aligned}$$

✓

Worked Example 9 (e) $A\bar{B} + A\overline{(B+C)} + B\overline{(B+C)}$

Apply De Morgan first: $\overline{(B+C)} = \bar{B}\bar{C}$.

$$A\bar{B} + A\bar{B}\bar{C} + B\bar{B}\bar{C}$$

The last term contains $B\bar{B} = 0$, so it vanishes:

$$A\bar{B} + A\bar{B}\bar{C} = A\bar{B}(1 + \bar{C}) = A\bar{B}$$

		AB			
		00	01	11	10
C	0	0	0	0	1
	1	0	0	0	1

One group of two — column $AB = 10$, both rows: $A=1, B=0$.

$$F = A\bar{B}$$

✓

C.3 Further practice

Suggested self-test

Work these without looking at the notes, then check against the answers.

Data representation

1. Convert 219_{10} to binary, octal and hexadecimal.
2. What is 11010110_2 interpreted as (a) unsigned, (b) signed magnitude, (c) one's complement, (d) two's complement?
3. Compute $45_{10} - 78_{10}$ in 8-bit two's complement. Does it overflow?
4. Represent -5.75_{10} in the 8-bit format (1 sign, 3-bit Excess-3 exponent, 4-bit mantissa). What is the round-off error, if any?
5. A memory system has a 14-bit address and is word addressable with 16-bit words. Give the number of addresses, the address range and the total memory size.

Boolean algebra and minimisation

6. Simplify $F = \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C + ABC$ algebraically and by K-map.
7. Derive both canonical forms for $F(A, B, C) = \sum m(0, 1, 4, 5, 6)$.
8. Minimise $F(a, b, c, d) = \sum m(0, 1, 2, 3, 4, 5, 10, 11, 14, 15)$ in SoP form.
9. Implement $F = \bar{A}B + A\bar{B}$ using NAND gates only.

Circuits

10. Design a 2-bit comparator: two 2-bit inputs, three outputs ($A > B$, $A = B$, $A < B$).
11. Show how a 4:1 multiplexer can implement $F(A, B) = \sum m(0, 1, 3)$.
12. Draw the state sequence of a 3-bit synchronous *down* counter and give the J-K input equations.

Memory and caching

13. A 512 kB direct-mapped cache has 64-byte lines and serves a byte addressable 4 GB memory. Give the tag, index and offset widths.
14. The same cache is made **4-way set associative**. What changes?
15. A cache has $T_c = 2$ ns, $T_m = 60$ ns and a hit ratio of 0.94. Find the EAT. What hit ratio would be needed to halve it?
16. A direct-mapped cache has 4 lines. A program repeatedly accesses blocks 0, 4, 0, 4, ... What is the hit ratio, and why? What single change would fix it?
17. Classify each as exploiting spatial or temporal locality: (a) fetching a 64-byte line for a 1-byte request; (b) LRU replacement; (c) keeping a loop counter in cache.

Organisation and performance

18. A program has 2×10^9 instructions with CPI 2.5, running on a 3 GHz processor. What is the execution time?
19. Machine X: 1.8 GHz, CPI 1.4. Machine Y: 2.4 GHz, CPI 2.1. Which is faster for the same program, and by how much?
20. Assemble **AND 5, 2, 7** and **ST 3, C4** for the example machine of Chapter 20.
21. A 5-stage pipeline with $t_p = 1.5$ ns runs 250 instructions. Give the total time and the speedup over a non-pipelined design.

C.4 Answers to the further practice

1. $219_{10} = 11011011_2 = 333_8 = DB_{16}$.
2. (a) 214_{10} (b) -86_{10} (c) -41_{10} (d) -42_{10} .
3. $45 = 00101101_2$; -78 : $78 = 01001110_2$, two's complement 10110010_2 . Sum = $11011111_2 = -33_{10}$. ✓No overflow — the operands had opposite signs.
4. $5.75 = 101.11_2 = 1.0111 \times 2^2$. Sign = 1; exponent $2 + 3 = 5 = 101_2$; mantissa 0111_2 . Result 11010111_2 ; **no round-off error** — the mantissa fits exactly.
5. $2^{14} = 16\,384$ addresses; range 0 to 16 383; size $16\,384 \times 2$ bytes = 32 768 bytes = 32 kB.
6. $C(\overline{A}\overline{B} + \overline{A}B + A\overline{B} + AB) = C \cdot 1 = C$. The K-map shows the entire $C = 1$ row grouped as one block of four.
7. SoM: $\overline{A}\overline{B}C + \overline{A}BC + A\overline{B}C + ABC$. PoM: $\prod M(2, 3, 7) = (A + \overline{B} + C)(A + \overline{B} + \overline{C})(\overline{A} + \overline{B} + \overline{C})$.
8. $F = \overline{a}\overline{b} + \overline{a}c + ac$ — three groups of four: the whole $ab=00$ column, the top-left quad ($a=0, c=0$), and the bottom-right quad ($a=1, c=1$). Verified against all sixteen rows.
9. Four NAND gates. With $g_1 = \overline{AB}$:

$$g_2 = \overline{A \cdot g_1}, \quad g_3 = \overline{B \cdot g_1}, \quad F = \overline{g_2 \cdot g_3} = A \oplus B$$

Verified over all four input combinations.

10. Use XNOR gates for bitwise equality: $E_1 = \overline{A_1 \oplus B_1}$, $E_0 = \overline{A_0 \oplus B_0}$. Then $(A=B) = E_1 E_0$; $(A>B) = A_1 \overline{B_1} + E_1 A_0 \overline{B_0}$; $(A<B) = \overline{A_1} B_1 + E_1 \overline{A_0} B_0$.
11. Set $S_1 = A$, $S_0 = B$, then hard-wire $I_0 = 1$, $I_1 = 1$, $I_2 = 0$, $I_3 = 1$.
12. Sequence 7, 6, 5, 4, 3, 2, 1, 0, 7, ... Bit i toggles when all lower bits are **0**: $J_0 = K_0 = 1$, $J_1 = K_1 = \overline{Q_0}$, $J_2 = K_2 = \overline{Q_0 Q_1}$.
13. Offset = $\log_2(64/1) = 6$; lines = $512\text{ kB}/64\text{ B} = 2^{13}$ so index = 13; address = $\log_2(2^{32}) = 32$; tag = $32 - 13 - 6 = \mathbf{13}$ bits.
14. Sets = $2^{13}/4 = 2^{11}$, so the index drops to **11** bits and the tag grows to **15**. The offset is unchanged at 6. Doubling associativity always moves one bit from index to tag.

15. $EAT = 0.94(2) + 0.06(60) = 1.88 + 3.6 = \mathbf{5.48}$ ns. For $EAT = 2.74$: $2.74 = 2HR + 60(1 - HR) \Rightarrow 58HR = 57.26 \Rightarrow HR = \mathbf{0.987}$.
16. Blocks 0 and 4 both map to line 0 mod 4 = 0, so every access evicts the block the next access needs: hit ratio = **0** — **thrashing**. Making the cache 2-way set associative lets both blocks live in the same set simultaneously, giving a hit ratio approaching 100%.
17. (a) **spatial** — the neighbours are fetched in anticipation; (b) **temporal** — recency of use predicts future use; (c) **temporal** — the same address is reused every iteration.
18. $T = N \times \text{CPI}/r = (2 \times 10^9 \times 2.5)/(3 \times 10^9) = 1.67$ s.
19. X: $\text{CPI}/r = 1.4/1.8 \times 10^9 = 0.778$ ns per instruction. Y: $2.1/2.4 \times 10^9 = 0.875$ ns. **X is faster** by a factor of $0.875/0.778 = 1.125$.
20. AND 5, 2, 7: opcode 8_{16} , Type 3 $\Rightarrow 8527_{16}$. ST 3, C4: opcode 3_{16} , Type 2 $\Rightarrow 33C4_{16}$.
21. Time = $(k + n - 1)t_p = (5 + 250 - 1) \times 1.5 = 254 \times 1.5 = 381$ ns. Non-pipelined = $5 \times 1.5 \times 250 = 1875$ ns. Speedup = $1875/381 = 4.92$ (out of a maximum of 5).