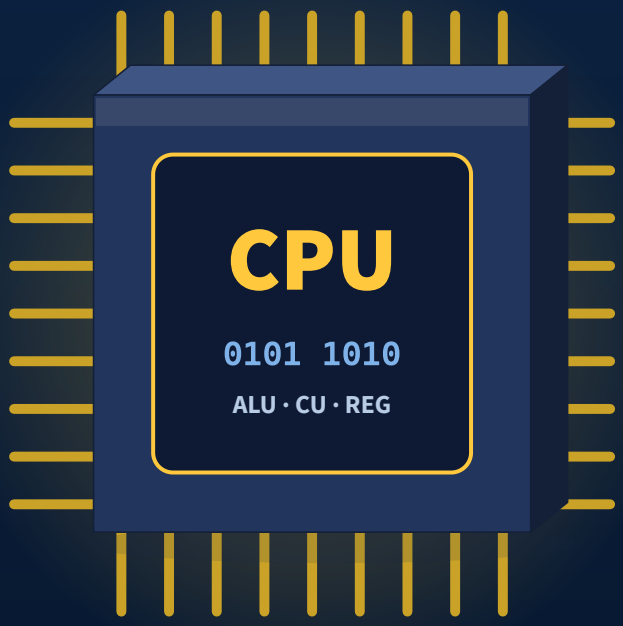
369 TESLA | KALVIYOGI

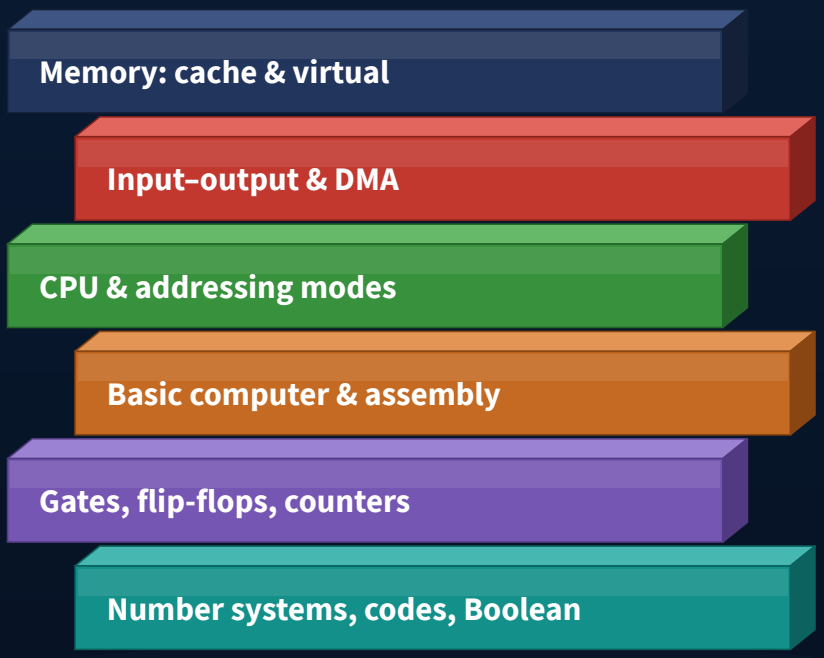
UNIT I

COMPUTER SYSTEM ARCHITECTURE

a brain-friendly guide

Head First style · puzzles & Q&A
Computed truth tables
3D infographics
TRB exam points





$45_{10} = 101101_2 = 2D_{16}$

$A \cdot B + A' \cdot C + B \cdot C = A \cdot B + A' \cdot C$

$S = A \oplus B \oplus Cin$

$Q^+ = J \cdot Q' + K' \cdot Q$

$AR \leftarrow PC$

$IR \leftarrow M[AR],\ PC \leftarrow PC + 1$

$D_7' \ I \ T_3 : AR \leftarrow M[AR]$

$EA = PC + offset$

$BR \rightarrow BG\ (DMA)$

$h \cdot Tc + (1-h) \cdot Tm$

LDA 004 ADD 005 STA 006

$(A + B) \cdot (C + D) \rightarrow AB+CD+*$

VSKM Nagarajan

EDUNIVRA PVT LTD

369 TESLA | KALVIYOGI

UNIT I

COMPUTER SYSTEM ARCHITECTURE

A Brain-Friendly Visual Guide: Number Systems · Codes · Boolean Algebra · Digital Logic · Basic Computer · CPU · I/O · Memory

VSKM Nagarajan

EDUNIVRA PVT LTD

Computer System Architecture — Unit I
VSKM Nagarajan · 369 TESLA | KALVIYOGI

Published by EDUNIVRA PVT LTD.

This edition is a rewritten, redesigned version of the author's Unit I notes. It follows the eight syllabus areas of the source (number systems and codes, Boolean algebra, digital logic circuits and components, basic computer organization and design, programming the basic computer, central processing unit, input–output organization, memory hierarchy) page by page, so nothing from the notes is left out.

Every truth table, conversion table and worked number in this book is **computed** when the book is built, and every "Worked result" box is the checked result of that computation. Technical statements were checked against the standard texts (M. Morris Mano, *Digital Logic and Computer Design* and *Computer System Architecture*). Where the source notes printed a wrong value or a wrong rule, the correct version is taught and a **Watch it!** box tells you what the error was, so that you are not trapped by it in the examination.

Contents

How to Use This Book 12

Number Systems 15

Chapter 1 · Number Systems: The Language of Computers 16

1.1 What is a Number System? 16
1.2 Why Computers Need Number Systems 17
1.3 Positional and Non-Positional Number Systems 18
1.4 Base or Radix of a Number System 19
1.5 Digits, Face Value, Place Value, MSD and LSD 19
1.6 General Positional Representation 20
1.7 The Decimal Number System 21
1.8 The Binary Number System 21
1.9 The Octal Number System 22
1.10 The Hexadecimal Number System 23
1.11 Comparing Decimal, Binary, Octal and Hexadecimal 23

Chapter 2 · Converting Between Bases 27

2.1 The Conversion Map: Principles of Conversion 27
2.2 Decimal Integer to Binary 29
2.3 Decimal Fraction to Binary 30
2.4 Decimal to Octal 31
2.5 Decimal to Hexadecimal 31
2.6 Binary, Octal and Hexadecimal to Decimal 32
2.7 Binary ↔ Octal: Groups of Three 32
2.8 Binary ↔ Hexadecimal: Groups of Four 33
2.9 Octal ↔ Hexadecimal: The Binary Bridge 34
2.10 Fractions Across Bases 34
2.11 Mixed-Base Conversion Problems 35
2.12 Shortcut Tables for Fast Conversion 35
2.13 Worked Conversion Examples 36
2.14 Practice Problems 37

Chapter 3 · Binary Arithmetic 38

3.1 The Rules of Binary Arithmetic 38
3.2 Binary Addition 39
3.3 Binary Subtraction 40
3.4 Binary Multiplication: Shift and Add 41
3.5 Binary Division 41

Chapter 4 · Signed Numbers and Complements 44

4.1 Unsigned and Signed Numbers 44
4.2 Sign-Magnitude 45
4.3 One's Complement 45
4.4 Two's Complement 45
4.5 Ranges of the Three Systems 47
4.6 Subtraction by Adding the Complement 47
4.7 Carry, Overflow and Underflow 48

Codes 52

Chapter 5 · Decimal Codes: BCD and Weighted Codes 53

5.1 What Is a Digital Code? 53
5.2 How Codes Are Classified 54
5.3 Weighted and Non-Weighted Codes 54
5.4 8421 BCD 55
5.5 Valid and Invalid BCD Codes 55
5.6 BCD Addition and the +6 Correction 56
5.7 Other Weighted Codes: 2421, 5211, 7421 and 84-2-1 57
5.8 Self-Complementing Codes 58

Chapter 6 · Excess-3, Gray and Alphanumeric Codes 61
6.1 Non-Weighted Codes 61
6.2 Excess-3 Code 61
6.3 Gray Code: the Reflected Binary Code 62
6.4 Converting Between Binary and Gray 64
6.5 Where Gray Code Is Used 65
6.6 Comparing the Decimal Codes 66
6.7 Alphanumeric Codes and ASCII 66
6.8 EBCDIC 67
6.9 Unicode and UTF-8 68

Chapter 7 · Error Detection and Correction 71
7.1 Errors in Digital Data 71
7.2 Redundancy and Hamming Distance 72
7.3 Parity and the Vertical Redundancy Check 73
7.4 Longitudinal Redundancy Check (Two-Dimensional Parity) 73
7.5 Checksum 74
7.6 Cyclic Redundancy Check (CRC) 74
7.7 The Hamming Code 75
7.8 The Syndrome, SEC and SEC-DED 77

Boolean Algebra 80

Chapter 8 · Boolean Operations and Expressions 81
8.1 What Boolean Algebra Is 81
8.2 Variables, Constants and Binary States 82
8.3 Boolean Algebra Is Not Ordinary Algebra 82
8.4 Where Boolean Algebra Is Used 82
8.5 The Basic Operations: NOT, AND, OR 83
8.6 NAND and NOR 84
8.7 XOR and XNOR 85
8.8 All Seven Operations Side by Side 85
8.9 Writing Expressions: Precedence and Parentheses 86
8.10 Evaluating an Expression Step by Step 87
8.11 From Expression to Truth Table 88
8.12 From Truth Table to Expression 88

Chapter 9 · Laws and Theorems of Boolean Algebra 90
9.1 Laws Are Rewrite Rules 90
9.2 Identity and Null Laws 91
9.3 Idempotent and Complement Laws 91
9.4 Involution (Double Complement) 91
9.5 Commutative and Associative Laws 92
9.6 Distributive Laws 92
9.7 Absorption Laws 93
9.8 Covering (Redundancy) and Consensus 94

9.9 De Morgan's Theorems 95
9.10 Two Ways to Prove an Identity 97
9.11 The Principle of Duality 98
9.12 The Complete Law Chart, and Using It 99

Chapter 10 · Boolean Functions and Canonical Forms 102
10.1 What a Boolean Function Is 102
10.2 Three Ways to Describe a Function 103
10.3 Literals, Terms, Product Terms and Sum Terms 104
10.4 Minterms and Maxterms 104
10.5 Standard and Canonical Forms 105
10.6 Canonical SOP: a Sum of Minterms 106
10.7 Canonical POS: a Product of Maxterms 106
10.8 Converting Between Σ and Π 107
10.9 Why Simplify? 109
10.10 Simplifying with the Laws and De Morgan 109

Chapter 11 · Karnaugh Maps 112
11.1 From Truth Table to Map 112
11.2 The Two-Variable Map 112
11.3 The Three-Variable Map 113
11.4 Grouping Rules and Reading a Loop 114
11.5 The Four-Variable Map 115
11.6 Prime Implicants: Choosing the Loops 116
11.7 Don't-Care Conditions 116
11.8 POS from the Zeros 118
11.9 NAND–NAND and NOR–NOR Realization 118

Digital Logic Circuits 122

Chapter 12 · Digital Systems and Logic Gates 123
12.1 Analog and Digital Systems 123
12.2 Digital Signals and Logic Levels 124
12.3 Positive and Negative Logic 124
12.4 The Digital Computer 125
12.5 Logic Gates as Devices 126
12.6 Universal Gates: NAND and NOR 128

Chapter 13 · From Expressions to Gate Circuits 131
13.1 Postulates: the Rules Everything Else Comes From 131
13.2 Laws as Gate Savers 132
13.3 From Expression to Circuit 133
13.4 From Circuit to Expression: Analysis 134
13.5 Canonical Forms as Two-Level Circuits 135
13.6 Simplify, Then Build 136
13.7 A Design Procedure: the Seat-Belt Buzzer 137

Chapter 14 · Adders and Subtractors 141
14.1 Combinational Circuits 141
14.2 The Half Adder 142
14.3 The Full Adder 142
14.4 A Full Adder from Two Half Adders 143
14.5 The Ripple-Carry Adder 144
14.6 The Carry Look-Ahead Adder 144
14.7 Half and Full Subtractors 145

14.8 The Adder-Subtractor 146
14.9 The BCD Adder 147

Chapter 15 · Comparators, Encoders, Decoders and Multiplexers 150
15.1 Magnitude Comparators 150
15.2 Encoders 151
15.3 The Priority Encoder 151
15.4 Decoders 152
15.5 BCD-to-Decimal and Seven-Segment Decoders 153
15.6 Multiplexers and Demultiplexers 155
15.7 Code Converters, Parity Generators and Checkers 156

Chapter 16 · Latches and Flip-Flops 158
16.1 Sequential Circuits: Memory and Feedback 158
16.2 Clocks, Synchronous Circuits and Triggering 159
16.3 The SR Latch 159
16.4 Gated Latches: Clocked SR and D 160
16.5 The Edge-Triggered D Flip-Flop 161
16.6 JK and T Flip-Flops 162
16.7 Characteristic Tables and Equations 162
16.8 Excitation Tables 163
16.9 Preset and Clear 163
16.10 Race-Around and the Master-Slave Flip-Flop 164
16.11 Converting One Flip-Flop into Another 165
16.12 Choosing a Flip-Flop 165

Chapter 17 · Registers and Counters 167
17.1 Registers 167
17.2 Shift Registers: SISO and SIPO 168
17.3 Parallel-In Serial-Out 168
17.4 PIPO, Bidirectional and Universal Registers 169
17.5 Ring and Johnson Counters 169
17.6 Counters and Modulus 170
17.7 Asynchronous (Ripple) Counters 170
17.8 Synchronous Counters 172
17.9 Decade and MOD-N Counters 172
17.10 Choosing a Counter 173

Chapter 18 · Integrated Circuits, Logic Families and Memory 175
18.1 Integrated Circuits 175
18.2 Scale of Integration: SSI to ULSI 176
18.3 Logic Families and Their Parameters 176
18.4 TTL and CMOS 177
18.5 RTL, DTL, ECL and I^2L 178
18.6 Fan-In, Fan-Out, Noise Margin and Delay 179
18.7 Power and the Speed-Power Product 180
18.8 Memory: Units, Addresses and Data Lines 180
18.9 Memory Organization and Expansion 181
18.10 RAM: SRAM and DRAM 182
18.11 ROM, PROM, EPROM, EEPROM and Flash 183
18.12 Cache, Associative and Virtual Memory 183

Chapter 19 · Circuit Analysis, Timing and Hazards 186
19.1 Analysing a Circuit 186
19.2 From Expression to Circuit 187

19.3 Timing Diagrams and Propagation Delay 187
19.4 How Fast Can the Clock Run? 188
19.5 Hazards and Glitches 189
19.6 Troubleshooting Digital Circuits 190
19.7 Where the Pieces of Part IV Are Used 191

Basic Computer Organization and Design 193

Chapter 20 · The Basic Computer: Registers and the Common Bus 194
20.1 The Stored-Program Computer 194
20.2 Words and the Instruction Format 195
20.3 Direct and Indirect Addressing 195
20.4 The Registers of the Basic Computer 196
20.5 Sequence Counter and Flip-Flops 197
20.6 The Common Bus 197
20.7 Register Transfer and Microoperations 198

Chapter 21 · Instructions, Timing and the Instruction Cycle 201
21.1 Three Kinds of Instruction 201
21.2 The Instruction Set at a Glance 202
21.3 Input–Output Instructions 203
21.4 Timing and Control 203
21.5 Control Functions and Control Signals 205
21.6 The Instruction Cycle: Fetch and Decode 205
21.7 Indirect Step and Execution 207

Chapter 22 · Memory-Reference and Register-Reference Instructions 210
22.1 How a Memory-Reference Instruction Executes 210
22.2 AND, ADD and LDA 211
22.3 STA and BUN 211
22.4 BSA: Branch and Save Return Address 212
22.5 ISZ: Increment and Skip if Zero 212
22.6 Register-Reference Instructions 213
22.7 Circulate, Increment and Skip 213

Chapter 23 · Input–Output and Interrupts 216
23.1 Input–Output Configuration 216
23.2 The Six I/O Instructions 217
23.3 Programmed I/O 217
23.4 Interrupts and the Interrupt Request 218
23.5 The Interrupt Cycle 218
23.6 The Service Routine and the Return 220
23.7 The Complete Computer and Its Control Design 220
23.8 Worked Examples and Self-Test 221

Programming the Basic Computer 224

Chapter 24 · Machine Language and Assembly Language 225
24.1 Programs and Languages 225
24.2 Machine Language 226
24.3 Binary, Octal and Hexadecimal Code 226
24.4 A Machine-Language Program, Traced 227
24.5 Assembly Language: Symbols and Fields 228
24.6 Pseudo-Instructions 229
24.7 Indirect Addresses and Translation 229

Chapter 25 · The Assembler 232
25.1 What an Assembler Does 232
25.2 The Symbolic Program in Memory 232
25.3 Why Two Passes? 233
25.4 The First Pass 233
25.5 The Second Pass 235
25.6 Error Detection 236
25.7 Hand Assembly of a Complete Program 236

Chapter 26 · Program Loops, Arithmetic, Subroutines and I/O Programming 239
26.1 Program Loops 239
26.2 Decisions with Skip Instructions 240
26.3 Arithmetic and Logic from the Instruction Set 242
26.4 Multiplication by Shift and Add 242
26.5 Multiple Precision and Arrays 244
26.6 Subroutines: BSA and BUN I 245
26.7 Parameters, Data Transfer and Nested Calls 246
26.8 Input–Output Programming 246

Central Processing Unit 249

Chapter 27 · General Registers and Stack Organization 250
27.1 The CPU and Its Parts 250
27.2 General Register Organization 251
27.3 The Control Word 251
27.4 Arithmetic, Logic and Shift Microoperations 253
27.5 Register Operations Step by Step 253
27.6 Stacks: Last In, First Out 254
27.7 Register Stack 254
27.8 Memory Stack 255
27.9 Reverse Polish Notation 256

Chapter 28 · Instruction Formats 258
28.1 Fields of an Instruction 258
28.2 Three-Address Instructions 259
28.3 Two-Address Instructions 260
28.4 One-Address Instructions 260
28.5 Zero-Address Instructions 260
28.6 Load–Store (RISC) Instructions 261
28.7 Comparing the Formats 261

Chapter 29 · Addressing Modes 263
29.1 Effective Address and the Mode Field 263
29.2 Implied and Immediate Modes 264
29.3 Register and Register-Indirect Modes 264
29.4 Autoincrement and Autodecrement 264
29.5 Direct and Indirect Modes 265
29.6 Relative Mode 265
29.7 Indexed and Base-Register Modes 266
29.8 Stack Addressing 267
29.9 All Modes on One Instruction 267

Chapter 30 · Instruction Types, Program Control and RISC/CISC 270
30.1 Data Transfer Instructions 270
30.2 Data Manipulation Instructions 271

30.3 Status Bits and Conditional Branches 271
30.4 Subroutine Call, Return and Program Interrupts 272
30.5 Reduced Instruction Set Computers 273
30.6 Complex Instruction Set Computers 274
30.7 RISC versus CISC 275
30.8 Worked CPU Problems 275

Input–Output Organization 278

Chapter 31 · Peripheral Devices and the I/O Interface 279
31.1 Why I/O Needs Organisation 279
31.2 Peripheral Devices 279
31.3 The I/O Interface and Its Functions 280
31.4 Interface Registers: an Example Unit 281
31.5 I/O Ports and Addressing 281
31.6 Isolated versus Memory-Mapped I/O 282

Chapter 32 · Asynchronous Transfer, Interrupts and DMA 284
32.1 Synchronous and Asynchronous Transfer 284
32.2 Strobe Control 285
32.3 Handshaking 285
32.4 Serial and Parallel Transfer 286
32.5 Modes of Transfer: Programmed I/O 286
32.6 Interrupt-Initiated I/O 287
32.7 Priority Interrupts 287
32.8 Direct Memory Access 289
32.9 DMA Transfer Modes and Bus Arbitration 289

Chapter 33 · Serial Communication 291
33.1 Serial Communication Fundamentals 291
33.2 Asynchronous Serial Transmission 292
33.3 Synchronous Serial Transmission 293
33.4 Simplex, Half-Duplex and Full-Duplex 294
33.5 Baud Rate, Bit Rate and Efficiency 295
33.6 Serial Interfaces and Protocols 295
33.7 Error Detection in Serial Communication 296
33.8 Putting the I/O Unit Together: Worked Examples 297

Memory Organization 300

Chapter 34 · Memory Hierarchy, Main and Auxiliary Memory 301
34.1 The Memory Hierarchy 301
34.2 Memory Performance Characteristics 302
34.3 Main Memory Organization 303
34.4 RAM: SRAM and DRAM 304
34.5 ROM and Its Family 304
34.6 Memory Address Map and Chip Selection 305
34.7 Auxiliary Memory 305
34.8 Magnetic Disks and Magnetic Tapes 306
34.9 Optical and Solid-State Storage 307

Chapter 35 · Associative Memory, Cache and Virtual Memory 309
35.1 Associative Memory 309
35.2 Cache Memory and Hit Ratio 310
35.3 Cache Mapping: Direct Mapping 311

35.4 Associative Mapping 312
35.5 Set-Associative Mapping 312
35.6 Replacement and Write Policies 313
35.7 Virtual Memory 314
35.8 Paging, Segmentation and Page Replacement 314
35.9 Memory Management Hardware and the TLB 315

BEFORE YOU BEGIN

How to Use This Book

"I can't believe they put that in an architecture book!"

Who is this book for?

This book is for you if...

- you are preparing for TRB Computer Instructor, PG-TRB, TNPSC or a university paper on Computer Organization;
- you can switch on a computer but have never wondered how a **bit** becomes an **add instruction**;
- you would rather **understand** one example deeply than memorise ten definitions.

This book is probably not for you if...

- you want a dictionary of terms with no pictures;
- you are scared that a book which talks to you, asks you questions and makes you write in the margins cannot be serious. (It is serious. Every answer was checked.)

We know what you're thinking

"How can *this* be a serious architecture book?" "What's with all the pictures?" "Can I really learn K-maps and the instruction cycle this way?" And we know what your **brain** is thinking. Your brain craves novelty. It is always scanning for something unusual. It quietly throws away the boring, repeated, textbook-shaped things, because they do not look important. So the twentieth definition that starts with "It is a method of..." slides straight out of your head.

The way to convince your brain that number systems, flip-flops and DMA are worth keeping is to make them **look important**: pictures with words **inside** them, a conversational voice, questions that make you stop, puzzles, a little surprise. That is what every page of this book tries to do.

Here's what we did

1 Pictures first
Words sit inside or beside the figure they describe, not on another page. Your brain links them better.

2 One story per chapter
Each chapter opens with a real situation (a bank queue, a traffic light, a lift) and the theory explains it.

3 Every number computed
Truth tables, conversions and worked results are computed and checked while the book is built. No typing errors.

4 Mistakes shown
Every error of the source notes is turned into a **Watch it!** box: what was printed, what is right, why.

5 You do the work
Pencil exercises and matching puzzles make you **use** the idea right away. Solutions are at the chapter end.

6 Exam angle
Purple **Exam point** boxes say exactly what the TRB examiner asks and which trap options appear.

The elements you will meet



BRAIN POWER

A question for you to **think** about. There is no printed answer: the point is that your brain does the work. Spend one minute before you read on.

There are no Dumb Questions

Q: **Is it OK to ask very basic questions?**

A: Yes. These boxes collect the questions students are afraid to ask in class. Most of them turn up as MCQ traps.

✎ Sharpen your pencil

An exercise. Write your answers in the book. The worked solution is at the end of the chapter, so do not peek first.



WATCH IT! · COMMON MISTAKE

A wrong statement that students (and sometimes printed notes) make, with the correction and the reason.

Fireside Chats · two concepts, one fireplace

Tonight's guest: SRAM	Tonight's guest: DRAM
Two related ideas talk to each other about their differences. You will never mix them up again.	And the conversation is far easier to remember than a comparison table. (We give the table too.)

EXAM POINT

What the examiner asks, the one-line answer, and the trap option to avoid.

Bend your brain into submission

1. **Slow down.** The more you understand, the less you have to memorise. Stop at every Brain Power and really think.
2. **Do the exercises. Write your own notes.** Using a pencil while you learn is proven to help memory.
3. **Read the Dumb Questions.** They are not optional side-bars; they are part of the core content.
4. **Do not read everything in one place.** Stand up, stretch, move to another seat. Your brain links learning to the surroundings.
5. **Talk about it, out loud.** Explain a K-map or the fetch cycle to a friend or to the wall. If you can explain it, you know it.
6. **Drink water and sleep.** A dehydrated brain thinks slowly, and a lot of the real learning happens after you close the book.
7. **Draw it.** Every circuit, timing diagram and flowchart here is meant to be re-drawn by you, from memory, on a blank page.

PART I · NUMBER SYSTEMS
Bases, conversions, binary arithmetic, complements (Ch 1–4)

PART II · CODES
BCD, weighted codes, Excess-3, Gray, ASCII/Unicode, error codes (Ch 5–7)

PART III · BOOLEAN ALGEBRA
Operations, laws, canonical forms, K-maps (Ch 8–11)

PART IV · DIGITAL LOGIC
Gates, combinational and sequential circuits, ICs, memory (Ch 12–19)

PART V · BASIC COMPUTER
Registers, bus, instruction cycle, instructions, interrupts (Ch 20–23)

PART VI · PROGRAMMING IT
Machine and assembly language, assembler, loops, subroutines (Ch 24–26)

PART VII · THE CPU
General registers, stack, instruction formats, addressing modes, RISC/CISC (Ch 27–30)

PART VIII · INPUT–OUTPUT
Interface, handshaking, priority interrupts, DMA, serial links (Ch 31–33)

PART IX · MEMORY
Hierarchy, main and auxiliary memory, associative, cache and virtual memory (Ch 34–35)

PART I · UNIT I SYLLABUS 1

Number Systems

Chapter 1 Number Systems: The Language of Computers

Chapter 2 Converting Between Bases

Chapter 3 Binary Arithmetic

Chapter 4 Signed Numbers and Complements

Introduction to number systems · conversion between number systems · binary arithmetic · signed-number representation, complements and overflow.

A computer only knows two things: on and off. By the end of this part you will be able to make those two things count, add, subtract and go negative.

CHAPTER · UNIT I · NUMBER SYSTEMS

1 Number Systems: The Language of Computers

Ten fingers gave us ten digits. A switch has only two positions. Guess what the computer uses.

OPENING SITUATION

Your phone says a photo is **3.2 MB**, your Wi-Fi router's address is **192.168.1.1**, a web page colours its heading **#1E90FF**, and a Linux file has permission **755**. Four different ways of writing numbers, all on one screen, and inside the phone every one of them is just a pattern of switches that are either **on** or **off**. This chapter is about how one quantity can wear many different "number costumes" (decimal, binary, octal, hexadecimal), and why the computer insists on the plainest costume of all.

IN THIS CHAPTER

- 1.1 What is a Number System?
- 1.2 Why Computers Need Number Systems
- 1.3 Positional and Non-Positional Number Systems
- 1.4 Base or Radix of a Number System
- 1.5 Digits, Face Value, Place Value, MSD and LSD
- 1.6 General Positional Representation
- 1.7 The Decimal Number System
- 1.8 The Binary Number System
- 1.9 The Octal Number System
- 1.10 The Hexadecimal Number System
- 1.11 Comparing Decimal, Binary, Octal and Hexadecimal

YOU NEED

- School arithmetic: place value, powers ($10^2 = 100$, $2^3 = 8$).
- Nothing else. This is where Unit I begins.

1.1 What is a Number System?

Count the people in a room. You could draw one stick per person, but for a stadium you would run out of paper. So people invented a cleverer trick: a **small set of symbols** plus a **rule** that tells you what each symbol is worth from where it stands. That pair, symbols plus rule, is a number system.

> **DEFINITION 1.1 — NUMBER SYSTEM**
>
> A **number system** is a method of representing quantities using a fixed set of **digits (symbols)** and a set of **rules** for combining them. In a positional system, the number of distinct digits is called the **base** or **radix**, and the base decides the place value of every position.

In decimal, **347** means 3 hundreds + 4 tens + 7 ones = $3 \times 100 + 4 \times 10 + 7 \times 1$. Only ten symbols (0–9), yet with positions they can write any whole number, and with a point they write fractions too. Number systems let us do four jobs:

1 Count and measure
Keep track of objects and events; measure length, weight, time and money.

2 Calculate
Add, subtract, multiply and divide with simple rules that work the same in every base.

3 Store and process
Hold data in memory cells and process it in circuits. This is where computers come in.

Table 1.1 The four bases of this unit, and one quantity (ten) written in each

System	Base	Digits	Ten is written	Where you meet it
Decimal	10	0–9	10_{10}	Daily life: money, time, measurement
Binary	2	0, 1	1010_{2}	Inside every computer and digital circuit
Octal	8	0–7	12_{8}	Older computers, Unix file permissions
Hexadecimal	16	0–9, A–F	A_{16}	Memory addresses, machine code, colour codes

BRAIN POWER

The digits "10" appear in the first column for decimal, and the written form "10" also appears in base 2, 8 and 16. What quantity does "10" stand for in base 2? In base 8? In base 16? What does "10" always mean, in **any** base b?

★ A number system = a set of digits + a rule for what each position is worth.

1.2 Why Computers Need Number Systems

A digital computer is built from billions of tiny electronic switches (transistors). A switch is reliable when it only has to be in one of **two** clearly different states: **OFF (0)** or **ON (1)**. Trying to make one wire hold ten different voltage levels would be slow, expensive and easily confused by electrical noise. So the machine counts in **binary**, and **everything** it handles has to be turned into binary numbers first.

D Data representation
Text, images, sound, video and the instructions themselves are all stored as numbers.

P Processing
The CPU performs arithmetic and logic operations on those numbers to make decisions.

S Storage and addressing
Memory cells hold binary values; every memory location, device and file has a numeric address.

C Coding and communication
Character codes (ASCII, Unicode), error-detecting codes and compression are number systems at work; networks carry streams of bits.

REAL-LIFE VIEW

ATMs, traffic lights, lifts, washing machines and smart TVs all contain small embedded computers. Your phone stores a photo as numbers and shows its size in megabytes; a web page and a WhatsApp message travel over the internet as long strings of 0s and 1s.

Table 1.2 Who uses which base, and why

System	Base	Used in computing for	Why
Decimal	10	What humans type and read: counting, user interfaces	Easy for people to understand
Binary	2	Actual internal operations, storage, processing, communication	Matches the two states of digital circuits
Octal	8	Compact form of binary on older machines; Unix permissions	1 octal digit = 3 bits
Hexadecimal	16	Memory addresses, machine code, debugging, colour codes	1 hex digit = 4 bits; much shorter than binary

There are no Dumb Questions

Q: **If the computer only uses binary, why do we need octal and hexadecimal at all?**

A: For **humans**. The machine never uses them. A 32-bit address in binary has 32 symbols, which people misread and miscopy. In hexadecimal it has 8. Octal and hex are shorthand that we write; the machine still stores the same bits.

Q: **Does a computer convert to decimal to do arithmetic?**

A: No. It adds and subtracts in binary. Decimal appears only at the edges: when you type a number in, and when a program prints it out.

★ **Binary is the computer's language; octal and hex are human shorthand for binary.**

1.3 Positional and Non-Positional Number Systems

Look at the digit 3 in 3, 30 and 300. Same symbol, three different values. That is a **positional** system: *where* a digit stands decides what it is worth. Now look at Roman numerals: X is ten whether it appears in XV, XX or CX. That is a **non-positional** system.

Positional (e.g. Hindu–Arabic decimal)

- Value of a digit depends on its **position**.
- Each position has a place value: a power of the base (1, 10, 100…).
- Has a symbol for **zero**, which holds empty places.
- Arithmetic is easy (column addition, long division).
- Used in everyday life, science, finance and all computers.

Non-positional (e.g. Roman numerals)

- Each symbol has a **fixed** value: I 1, V 5, X 10, L 50, C 100, D 500, M 1000.
- Values are combined by **adding** (VI = 6) and a few **subtracting** pairs (IV = 4, XL = 40).
- No zero; very large numbers become long.
- Arithmetic is clumsy.
- Still used for chapter numbers, clock faces, names of events.

Example. 345_{10} = 3 × 100 + 4 × 10 + 5 × 1. In Roman numerals the same quantity is CCCXLV = C + C + C + XL + V = 100 + 100 + 100 + 40 + 5 = 345.

WATCH IT! · COMMON MISTAKE 1.1 — "ROMAN NUMERALS ONLY ADD"

The Roman value of a symbol never depends on its **place value**, but it is not purely additive either: a smaller symbol placed **before** a larger one is **subtracted** (IV = 4, IX = 9, XL = 40, CM = 900). So the order of symbols matters even in this non-positional system; what it lacks is **place value** and a zero.

★ Positional: the place gives the value. Non-positional: the symbol alone gives the value.

1.4 Base or Radix of a Number System

DEFINITION 1.2 — BASE (RADIX)

The **base** or **radix** b of a positional number system is the **number of distinct digits** it uses. A base-b system has exactly b digits, **0, 1, ..., b − 1**, and its place values are the powers of b: $b^0 = 1$ on the right, then b^1, b^2, b^3 ... to the left.

Notice the digits run only up to **b − 1**. Base 2 has 0 and 1, never 2; base 8 stops at 7; base 16 needs six extra symbols, so it borrows the letters **A, B, C, D, E, F** for 10, 11, 12, 13, 14, 15. When you reach b, you have filled a position, so you write "10" and start again (a "carry").

Table 1.3 Common bases and their digits

Base b	Name	Digits allowed	Place values (right to left)
2	Binary	`0 1`	`1, 2, 4, 8, 16, 32 …`
8	Octal	`0 1 2 3 4 5 6 7`	`1, 8, 64, 512, 4096 …`
10	Decimal	`0 1 2 3 4 5 6 7 8 9`	`1, 10, 100, 1000 …`
16	Hexadecimal	`0–9 A B C D E F`	`1, 16, 256, 4096, 65536 …`

Two quick examples. $(1011)_2 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = \mathbf{11}_{10}$. $(3A)_{16} = 3 \times 16 + 10 \times 1 = 48 + 10 = \mathbf{58}_{10}$. Same rule, different base: change the base, the digits change, the idea stays the same!

EXAM POINT — VALID DIGITS

"Which of these is **not** a valid octal number: 7046, 1777, 2981, 0?" Answer: **2981**, because 8 and 9 are not octal digits. In base b the largest digit is b − 1. The number of digits in base b is b; the base itself is never a digit of that base.

★ Base b ⇒ b digits (0 … b−1) ⇒ place values $b^0, b^1, b^2, \ldots$

1.5 Digits, Face Value, Place Value, MSD and LSD

Every digit in a number has two "values", and examiners love to ask which is which.

F Face value

The digit **itself**, whatever its position. In 5832 the face value of 8 is **8**.

P Place value

Digit × weight of its position. In 5832 the place value of 8 is 8 × 100 = **800**.

M MSD

Most Significant Digit: the **leftmost non-zero** digit, with the highest weight. It decides the size of the number. In 5832 it is **5**.

L LSD

Least Significant Digit: the **rightmost** digit, with the smallest weight. In 5832 it is **2**.

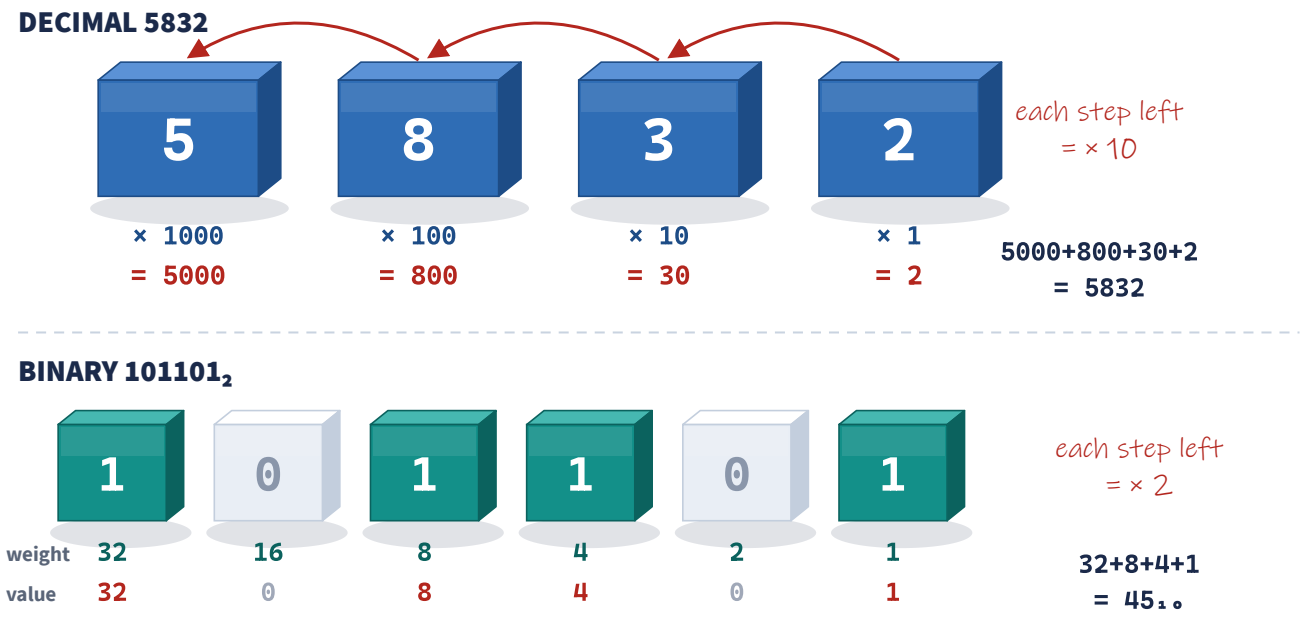


Figure 1.1 Place value = digit × weight. In decimal each step to the left multiplies the weight by 10; in binary by 2. In 101101_2 the MSB (most significant bit) has weight 32 and the LSB has weight 1, so the value is 32 + 8 + 4 + 1 = 45.

Why do MSD and LSD matter? The MSD tells you the **order of magnitude**: 1000 is larger than 999 because its MSD sits in the thousands place. The LSD is where arithmetic starts (you add from the right) and where rounding happens. In binary the same ideas are called the **MSB** and **LSB** (most/least significant **bit**); the LSB also tells you at a glance whether a number is odd (1) or even (0).

There are no Dumb Questions

Q: **In the 8-bit register value 0010 1101, is the MSB the 0 on the left or the first 1?**

A: For a **number**, the most significant digit is the leftmost **non-zero** digit (here the 1 with weight 32). For a **register** or a fixed-width binary word, "MSB" means bit 7, the leftmost bit position, even when it holds 0. Read the question: "MSB of the number" or "MSB of the byte".

★ **Face value = the digit. Place value = digit × weight. MSD = leftmost non-zero, LSD = rightmost.**

1.6 General Positional Representation

All of the above is one formula. Number the integer positions 0, 1, 2 … from the point to the **left**, and the fraction positions −1, −2 … to the **right**:

FORMULA — VALUE OF A BASE-B NUMBER

$$(d_n\ d_{n-1}\ \dots\ d_1\ d_0\ .\ d_{-1}\ \dots\ d_{-m})_b \;=\; \Sigma_{i=-m}^{n}\ d_i \times b^i, \qquad \text{with } b \geq 2 \text{ and } 0 \leq d_i < b$$

Integer positions carry non-negative powers (b^0, b^1, b^2 …); fraction positions carry negative powers ($b^{-1} = 1/b$, $b^{-2} = 1/b^2$ …). n is the highest power (leftmost integer digit) and m the number of fraction digits.

10 425.7 in base 10

$4\times10^2 + 2\times10^1 + 5\times10^0 + 7\times10^{-1}$
$= 400 + 20 + 5 + 0.7 = \mathbf{425.7}$

2 1011.01 in base 2

$1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 + 0\times2^{-1} + 1\times2^{-2}$
$= 8 + 0 + 2 + 1 + 0 + 0.25 = \mathbf{11.25}_{10}$

This one formula is the "**to decimal**" conversion method for every base, so it can be checked mechanically. Applied to the examples of this section:

WORKED RESULT

```
425.7 (base 10) = 425.7      1011.01 (base 2) = 11.25
527 (base 8) = 343      3A (base 16) = 58      101.11 (base 2) = 5.75
independent check: 527 (8) = 343   3A (16) = 58   101101 (2) = 45
```

Result 1.1 The positional formula applied digit by digit; each total agrees with an independent base conversion.

WATCH IT! · COMMON MISTAKE 1.2 — "EVERY NUMBER HAS A FINITE FORM IN EVERY BASE"

The source (p11) says "every number can be uniquely represented in any base b ≥ 2". True for **whole numbers**. For **fractions** it fails: $0.1_{10} = 0.000110011001100\ldots_2$ repeats forever, just as 1/3 = 0.333... in decimal. A computer must cut it off, which is why 0.1 + 0.2 prints **0.30000000000000004** in most languages. A fraction p/q has a finite form in base b only when every prime factor of q divides b (so 1/2, 1/4, 3/8 are finite in binary; 1/10 and 1/5 are not).

★ **Value = Σ digit × base^{position} — positive powers left of the point, negative to the right.**

1.7 The Decimal Number System

You have used it all your life without needing its name. The **decimal** (base-10) system uses ten digits 0–9; each place is a power of 10. Move one place **left** and the value is multiplied by 10; move one place **right** and it is divided by 10.

1 Whole number 4729

```
4×1000 + 7×100 + 2×10 + 9×1
= 4000 + 700 + 20 + 9 = 4729
```

2 Fraction 36.45

```
3×10 + 6×1 + 4×0.1 + 5×0.01
= 30 + 6 + 0.4 + 0.05 = 36.45
```

Key characteristics: base 10, digits 0–9, positional, expandable to very large and very small numbers (integers and decimals). **Applications:** prices and money, measurement, percentages and interest, accounting and banking, statistics, and the input/output of every program.

RELAX · YOU ALREADY KNOW DECIMAL

Decimal is here only as the reference point. Everything you know about it (place value, carrying, the point) carries straight over to the other bases. Only the number 10 changes to 2, 8 or 16.

★ **Decimal: ten digits, place values 10^n, the human-friendly system.**

1.8 The Binary Number System

DEFINITION 1.3 — BINARY

The **binary** number system is the base-2 positional system. It uses only the two digits **0 and 1**, called **bits** (binary digits). Its place values are powers of 2: ... 8, 4, 2, 1 . ½, ¼, ⅛ ...

Table 1.4 Binary place values around the binary point

Position	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}
Value	8	4	2	1	.	1/2	1/4	1/8

left of the point: ×2 each step

right of the point: ÷2 each step

1 1101_2

$1\times8 + 1\times4 + 0\times2 + 1\times1 = 13_{10}$

2 101.11_2

$4 + 0 + 1 + 0.5 + 0.25 = 5.75_{10}$

3 10110_2

$16 + 0 + 4 + 2 + 0 = 22_{10}$

Why binary? Digital devices have two dependable states: ON/OFF, TRUE/FALSE, HIGH/LOW voltage. Two-state circuits are simple, cheap, fast and hard to confuse with noise, so all data (numbers, text, images, sound, video) and all instructions are stored and processed in binary. **Uses:** processors and instructions, memory cells, network transmission, multimedia encoding.

Decimal (base 10)

- Digits 0–9
- Place values 10^n
- High human readability
- Computers convert it to binary on input

Binary (base 2)

- Digits 0, 1
- Place values 2^n
- Long and tiring for humans, ideal for machines
- Used directly by the hardware

MEMORY TRICK — THE POWERS OF 2

Learn 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, **1024** by heart, the way you learnt the 10 times table. $2^{10} = 1024 \approx$ one thousand, so $2^{20} \approx$ a million and $2^{30} \approx$ a billion. That is why a "kilobyte" of memory is 1024 bytes.

★ **Binary: two digits, place values 2^n, the machine's own system.**

1.9 The Octal Number System

DEFINITION 1.4 — OCTAL

The **octal** number system is the base-8 positional system with digits **0–7** and place values $8^0 = 1$, $8^1 = 8$, $8^2 = 64$, $8^3 = 512$, $8^4 = 4096$ … Because $\mathbf{8 = 2^3}$, one octal digit stands for exactly **three bits**.

Worked example. $(527)_8 = 5 \times 64 + 2 \times 8 + 7 \times 1 = 320 + 16 + 7 = \mathbf{343_{10}}$. To get binary, replace each octal digit by its 3-bit group: 5 → 101, 2 → 010, 7 → 111, so $(527)_8 = \texttt{101 010 111}_2$. Going back, group the bits in threes from the right.

Table 1.5 Each octal digit is one 3-bit group

Octal	0	1	2	3	4	5	6	7
Binary	000	001	010	011	100	101	110	111

Where octal is used: early computers whose word length was a multiple of 3 bits (12-, 24-, 36-bit machines) printed memory dumps in octal; Unix/Linux file permissions are octal (**755** = 111 101 101 = rwx r-x r-x); some network tools even accept octal address parts. Today hexadecimal has mostly replaced it.

★ **Octal: digits 0–7, one digit = 3 bits.**

1.10 The Hexadecimal Number System

DEFINITION 1.5 — HEXADECIMAL

The **hexadecimal** (hex) system is the base-16 positional system. Its 16 digits are **0–9 and A–F**, where A = 10, B = 11, C = 12, D = 13, E = 14, F = 15. Place values are 1, 16, 256, 4096, 65536 … Because **$16 = 2^4$**, one hex digit stands for exactly **four bits** (a **nibble**).

Table 1.6 The sixteen hex digits

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

Example. $2F_{16} = 2 \times 16 + 15 \times 1 = \mathbf{47_{10}}$, and in binary 2 → 0010, F → 1111, so $2F_{16} = \texttt{0010 1111}_2$. A memory address such as **0x1A3F** is $\texttt{0001 1010 0011 1111}_2$: four hex digits instead of sixteen bits, so hex cuts the length of a binary string by 75 %. (The prefix **0x** is the programmer's way of saying "this is hex".)

REAL-LIFE VIEW

A web colour **#FF5733** is three hex bytes: red FF = 255, green 57 = 87, blue 33 = 51, an orange-red. **#1E90FF** is "dodger blue". Error codes (0x80070005), MAC addresses (3C:52:82:…) and the addresses in a debugger are all hex.

Fireside Chats · Binary vs Hexadecimal

Binary	Hexadecimal
Let's be honest. I do all the real work. Every gate, every wire, every memory cell speaks me.	Nobody denies it. But when a programmer has to read you, they get a headache. Thirty-two 0s and 1s in a row!
Long, but honest. Each of my digits is one switch. You can see exactly which bit is on.	So can they with me. I am just you, four bits at a time. 1111 is F, 1010 is A. No information lost.
Then why does nobody build a hex computer?	Because a wire with sixteen voltage levels would be a nightmare. I'm a *notation*, not a machine. I live on paper and screens.
Fine. You for the humans, me for the hardware.	Deal. And octal? He used to do my job, three bits at a time, on the old 36-bit machines.

★ **Hex: digits 0–9, A–F; one digit = 4 bits (a nibble); the programmer's shorthand for binary.**

1.11 Comparing Decimal, Binary, Octal and Hexadecimal

All four are positional systems. They differ only in the base and the symbols; the place-value rule **value of a digit = digit × base$^{\text{position}}$** is common to all.

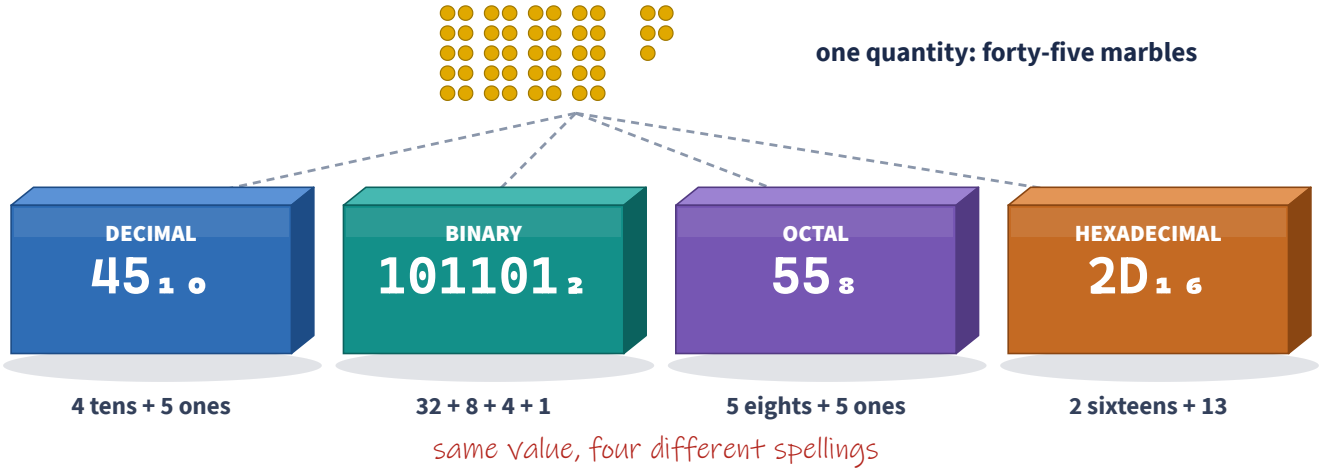


Figure 1.2 One quantity, four representations. Conversion never changes the **value**, only the way it is written: $45_{10} = 101101_2 = 55_8 = 2D_{16}$.

Table 1.7 The four number systems at a glance

Feature	Decimal	Binary	Octal	Hexadecimal
Base	10	2	8	16
Digits	0–9	0, 1	0–7	0–9, A–F
Place values	10^n	2^n	8^n	16^n
One digit equals	one decimal symbol	1 bit	3 bits ($8 = 2^3$)	4 bits ($16 = 2^4$)
Common use	daily life, finance, science	computers, logic circuits	compact binary, permissions	addresses, machine code, colours
45 is written	45	101101	55	2D

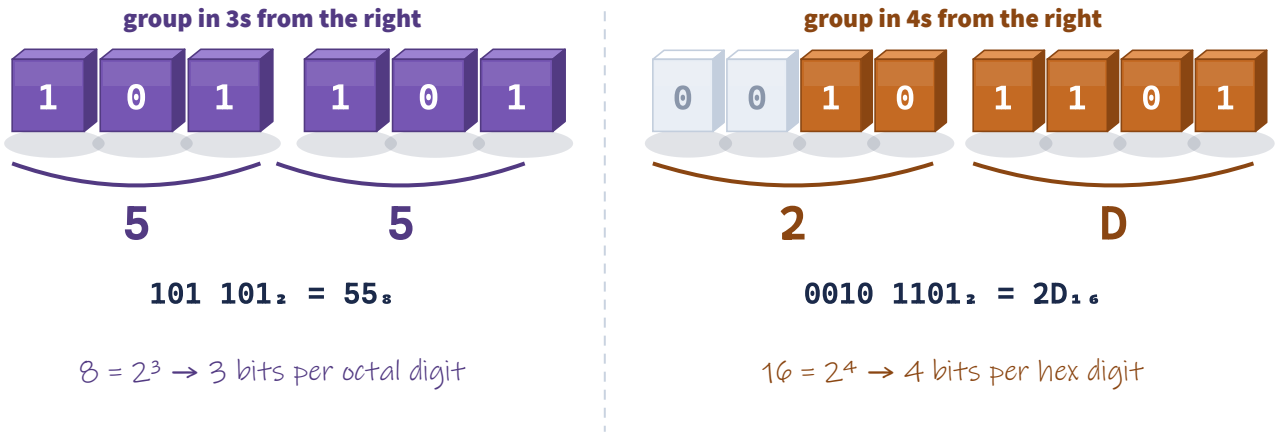


Figure 1.3 Why octal and hex are "shorthand": group the bits from the right, three at a time for octal, four at a time for hex. Pad the leftmost group with zeros (0010 1101). Chapter 2 uses this for fast conversions.

Who Does What?

Binary	•	• Web colour codes and memory addresses
Octal	•	• Prices, marks and measurements
Decimal	•	• Transistor states inside the CPU
Hexadecimal	•	• Unix permission 755
Roman numerals	•	• Chapter IV of a novel

✎ Sharpen your pencil 1

1. In 7425, what are the face value and the place value of 4? _____
2. Convert to decimal: $(11011)_2$ _____ · $(64)_8$ _____ · $(FF)_{16}$ _____ · $(110.101)_2$ _____
3. What is the place value of the 1 in $(1000)_{16}$? _____
4. Is $(189)_8$ a valid octal number? Why? _____
5. How many digits does a base-5 system have, and what is its largest digit? _____
6. Write the permission 640 in binary and say what it allows. _____

EXAM POINT — THE FAVOURITE ONE-LINERS

- Base of hexadecimal = **16**; largest hex digit = **F (15)**; 1 hex digit = **4 bits**; 1 octal digit = **3 bits**.
- "10" in any base b equals **b**. "100" in base b equals $\mathbf{b^2}$.
- Largest n-digit number in base b = $b^n - 1$ (e.g. $FFFF_{16} = 65535$, $7777_8 = 4095$, $1111_2 = 15$).
- Roman numerals are **non-positional**; the Hindu–Arabic decimal system is positional and has a **zero**.

★ **Same value, different base: decimal for people, binary for machines, octal/hex for people reading machines.**

✓ Solution · Sharpen your pencil 1

1. Face value 4; place value 4 × 100 = **400**.
2. $(11011)_2 = 16 + 8 + 2 + 1 =$ **27** · $(64)_8 = 6 \times 8 + 4 =$ **52** · $(FF)_{16} = 15 \times 16 + 15 =$ **255** · $(110.101)_2 = 4 + 2 + 0.5 + 0.125 =$ **6.625**.
3. $16^3 =$ **4096**.
4. **No**: octal digits are 0–7, so 8 and 9 cannot appear.
5. Five digits, 0–4; the largest is **4**.
6. 640 = 110 100 000 → owner rw- (read, write), group r-- (read), others --- (nothing).

Who Does What? Binary → transistor states · Octal → permission 755 · Decimal → prices and marks · Hexadecimal → colour codes and addresses · Roman → Chapter IV.

BULLET POINTS

- A number system is a set of digits plus a rule for combining them.
- In a **positional** system the place of a digit sets its weight; Roman numerals are **non-positional**.
- Base (radix) b = number of digits; the digits are 0 … b − 1.
- Place values are b^0, b^1, b^2 … left of the point and b^{-1}, b^{-2} … right of it.
- Value = $\Sigma\, d_i \times b^i$: the "to decimal" rule for every base.
- Face value = the digit; place value = digit × weight.
- MSD = leftmost non-zero digit; LSD = rightmost digit (MSB/LSB in binary).
- Computers use **binary** because two-state switches are simple and reliable.
- Octal (1 digit = 3 bits) and hex (1 digit = 4 bits) are human shorthand for binary.
- $45_{10} = 101101_2 = 55_8 = 2D_{16}$: conversion changes the form, never the value.
- Some decimal fractions (0.1) never end in binary.

Next: Chapter 2 turns the "same value, different spelling" idea into four mechanical methods: repeated division, repeated multiplication, the weighted sum, and bit grouping.

CHAPTER · UNIT I · NUMBER SYSTEMS

2 Converting Between Bases

Divide your way down, multiply your way right, and group your way across.

OPENING SITUATION

A shopkeeper gives you change for ₹ 45 using only ₹ 32, ₹ 16, ₹ 8, ₹ 4, ₹ 2 and ₹ 1 coins, at most one of each. Which coins? You would grab the biggest coin that fits (32), then the biggest that fits the rest (8), and so on: 32 + 8 + 4 + 1. You have just converted 45 to binary: **101101**, one bit per coin. This chapter gives you four reliable machines for such conversions, so that any number can be moved between decimal, binary, octal and hexadecimal without ever changing its value.

IN THIS CHAPTER

- 2.1 The Conversion Map: Principles of Conversion
- 2.2 Decimal Integer to Binary
- 2.3 Decimal Fraction to Binary
- 2.4 Decimal to Octal
- 2.5 Decimal to Hexadecimal
- 2.6 Binary, Octal and Hexadecimal to Decimal
- 2.7 Binary ↔ Octal: Groups of Three
- 2.8 Binary ↔ Hexadecimal: Groups of Four
- 2.9 Octal ↔ Hexadecimal: The Binary Bridge
- 2.10 Fractions Across Bases
- 2.11 Mixed-Base Conversion Problems
- 2.12 Shortcut Tables for Fast Conversion
- 2.13 Worked Conversion Examples
- 2.14 Practice Problems

YOU NEED

- Chapter 1: base, place value, $\Sigma\, d_i \times b^i$.
- The 3-bit and 4-bit tables (Tables 1.5, 1.6).

2.1 The Conversion Map: Principles of Conversion

CORE PRINCIPLE

Number-system conversion changes the **representation** of a number from one base (radix) to another while preserving its **value**. Value of a number = sum of (digit × power of the base).

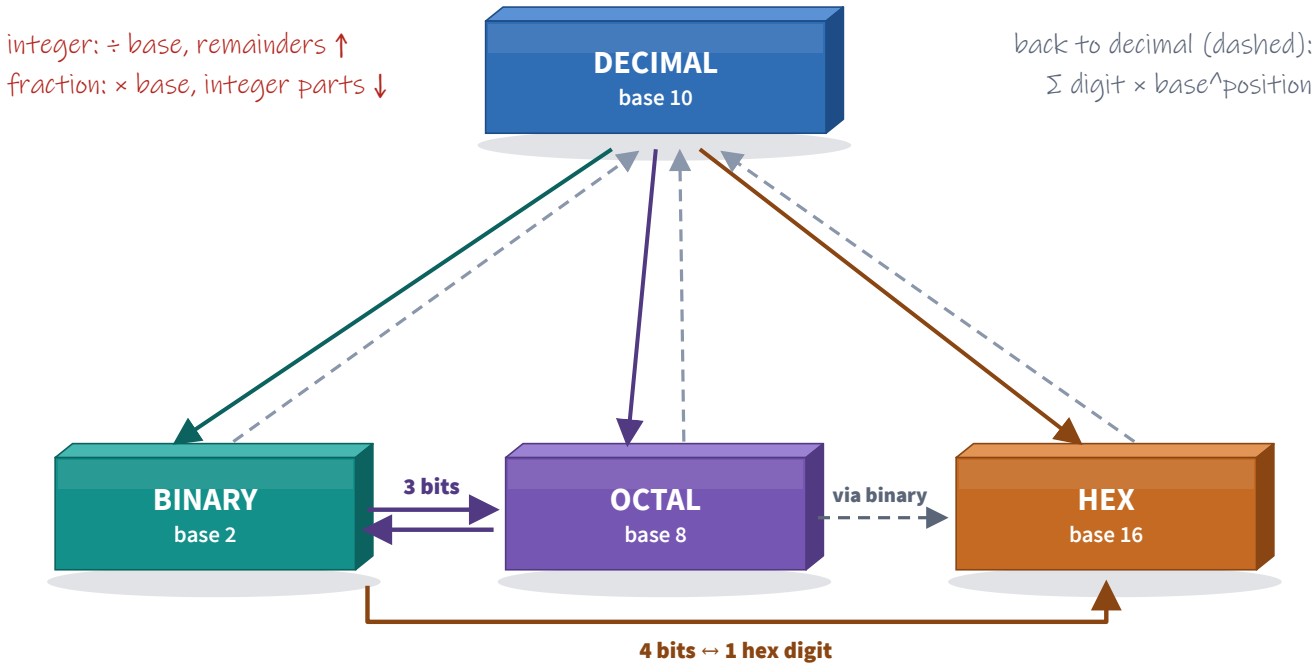


Figure 2.1 The whole chapter on one page. From decimal: divide (integer part) and multiply (fraction part) by the target base. To decimal: the weighted sum. Between binary, octal and hex: group bits; octal ↔ hex goes through binary.

Table 2.1 Integer part versus fractional part

Aspect	Integer part	Fractional part
Operation	Repeated **division** by the target base	Repeated **multiplication** by the target base
Digits collected	The **remainders**	The **integer parts** of the products
Order of digits	Reverse: last remainder is the MSD (read **bottom to top**)	Forward: first integer part is the first digit after the point (read **top to bottom**)
Stop when	The quotient becomes 0	The fraction becomes 0, or you have enough digits
Classic error	Reading remainders top to bottom	Reading integer parts bottom to top

Direct conversion is possible when both bases are powers of 2 (2, 8, 16): just group or ungroup bits. Otherwise use an **indirect** path, usually through decimal. Quick checks: $25_{10} = 16 + 8 + 1 = 11001_2$; $1101_2 = 8 + 4 + 1 = 13_{10}$; $17_8 = 8 + 7 = 15_{10}$; $A_{16} = 10_{10}$.

There are no Dumb Questions

Q: **The notes say octal ↔ hex is a "direct" conversion by grouping. Is there a grouping that maps octal digits straight to hex digits?**

A: No. 3 bits and 4 bits do not line up, so one octal digit never matches one hex digit. "Direct" here means you **avoid decimal**: expand octal to bits, then regroup the bits in fours (§2.9). Binary is the bridge.

★ **Value never changes; integer part ÷, fraction part ×, power-of-2 bases group.**

2.2 Decimal Integer to Binary

Method: repeated division by 2. Divide by 2 and write the remainder (0 or 1). Divide the quotient again, and again, until the quotient is 0. The remainders, read from the **last to the first**, are the binary number. Why bottom to top? The first remainder tells you whether the number is odd, which is the **LSB**; each later remainder belongs to a higher power of 2.

Divide	Quotient	Remainder
45 ÷ 2	22	1
22 ÷ 2	11	0
11 ÷ 2	5	1
5 ÷ 2	2	1
2 ÷ 2	1	0
1 ÷ 2	0	1

↑ read the remainders from bottom to top

$$45_{10} = 101101_2$$

Divide	Quotient	Remainder
156 ÷ 2	78	0
78 ÷ 2	39	0
39 ÷ 2	19	1
19 ÷ 2	9	1
9 ÷ 2	4	1
4 ÷ 2	2	0
2 ÷ 2	1	0
1 ÷ 2	0	1

↑ read the remainders from bottom to top

$$156_{10} = 10011100_2$$

Check by place value: $101101_2 = 32 + 8 + 4 + 1 = 45$. ✓

SHORTCUT — THE HIGHEST-POWER-OF-2 METHOD (THE COIN METHOD)

Find the highest power of 2 that is ≤ the number, write 1 in that place, subtract, and repeat with the rest; every place you skip gets 0. 45: powers 32 16 8 4 2 1 → 45 − 32 = 13 → 13 − 8 = 5 → 5 − 4 = 1 → 1 − 1 = 0. Used: 32, 8, 4, 1 → **101101**. Fast for small numbers in an MCQ.

WATCH IT! · COMMON MISTAKE 2.1 — THREE WAYS TO SPOIL A DIVISION TABLE

(1) Reading the remainders **top to bottom** (you get 101101 backwards = 101101 only by luck; 156 would come out 00111001). (2) Forgetting to write a **0 remainder**. (3) Stopping when the quotient is 1 instead of 0: the final 1 ÷ 2 = 0 remainder **1** is the MSB.

★ **Integer → binary: divide by 2, read remainders upward.**

2.3 Decimal Fraction to Binary

Method: repeated multiplication by 2. Multiply the fraction by 2; the integer part (0 or 1) is the next bit after the point. Keep only the fractional part and multiply again. Read the bits **top to bottom**. Why? Multiplying by 2 shifts the binary point one place right, pushing the first fraction bit (weight ½) across the point into the integer part.

Multiply	Product	Integer part
0.625 × 2	1.25	1
0.25 × 2	0.5	0
0.5 × 2	1	1

↓ read the integer parts from top to bottom

no division table: the integer part is 0

stop when the fraction becomes 0

$$0.625_{10} = 0.101_2$$

Likewise $0.375_{10} = 0.011_2$ (0.75 → 0, 1.5 → 1, 1.0 → 1). **Mixed numbers:** convert the two parts separately and join them at the point: $10.625_{10} = 1010_2 + 0.101_2 = \mathbf{1010.101_2}$.

Non-terminating fractions. Some fractions never reach 0. Watch 0.1:

Multiply	Product	Integer part
0.1 × 2	0.2	0
0.2 × 2	0.4	0
0.4 × 2	0.8	0
0.8 × 2	1.6	1
0.6 × 2	1.2	1
0.2 × 2	0.4	0
0.4 × 2	0.8	0
0.8 × 2	1.6	1
0.6 × 2	1.2	1
0.2 × 2	0.4	0

↓ read the integer parts from top to bottom …

$$0.1_{10} = 0.0001100110\ \ldots_2$$

After the 0.2 → 0.4 → 0.8 → 0.6 → 0.2 cycle starts, the bits **0011** repeat forever: $0.1_{10} = 0.0\underline{00110011}\ldots_2$. You stop at the required number of places (the answer is then an approximation).



BRAIN POWER

Why does 0.625 stop after three steps but 0.1 never stops? Look at the denominators: 0.625 = 5/8 and 0.1 = 1/10. What does a denominator need to be for the fraction to end in binary?

★ **Fraction → binary: multiply by 2, read the integer parts downward.**

2.4 Decimal to Octal

Exactly the same two machines, with 8 instead of 2. **Integer part:** divide by 8, remainders 0–7, read bottom to top. **Fraction part:** multiply by 8, integer parts 0–7, read top to bottom.

Divide	Quotient	Remainder
347 ÷ 8	43	3
43 ÷ 8	5	3
5 ÷ 8	0	5

↑ read the remainders from bottom to top

$$347_{10} = 533_8$$

Verify: $533_8 = 5 \times 64 + 3 \times 8 + 3 = 320 + 24 + 3 = 347$ ✓. Likewise $65_{10} = 101_8$ (65 ÷ 8 = 8 r 1, 8 ÷ 8 = 1 r 0, 1 ÷ 8 = 0 r 1).

Divide	Quotient	Remainder
18 ÷ 8	2	2
2 ÷ 8	0	2

↑ read the remainders from bottom to top

Multiply	Product	Integer part
0.625 × 8	5	5

↓ read the integer parts from top to bottom

integer and fraction parts are converted separately, then joined

0.3125 × 8 = 2.5 → 2, 0.5 × 8 = 4.0 → 4, so 0.3125 = 0.24_8

$$18.625_{10} = 22.5_8$$



WATCH IT! · COMMON MISTAKE 2.2 — OCTAL FRACTIONS

Using division for the fraction, reading the fraction digits bottom to top, or dropping a **0 digit** that a step produced (an integer part 0 is still a valid octal digit: 0.0625 × 8 = 0.5 → 0, then 0.5 × 8 = 4 → 0.04_8).

★ **Decimal → octal: ÷ 8 for the integer, × 8 for the fraction.**

2.5 Decimal to Hexadecimal

Use 16, and remember to **rename** the remainders and integer parts 10–15 as **A–F**.

Divide	Quotient	Remainder
314 ÷ 16	19	A (10)
19 ÷ 16	1	3
1 ÷ 16	0	1

↑ read the remainders from bottom to top

$$314_{10} = 13A_{16}$$

Verify: $13A_{16} = 1 \times 256 + 3 \times 16 + 10 = 256 + 48 + 10 = 314$ ✓. And $255_{10} = \mathbf{FF_{16}}$ (255 ÷ 16 = 15 r 15, 15 ÷ 16 = 0 r 15).

Divide	Quotient	Remainder
26 ÷ 16	1	A (10)
1 ÷ 16	0	1

↑ read the remainders from bottom to top

Multiply	Product	Integer part
0.625 × 16	10	A

↓ read the integer parts from top to bottom

0.625 × 16 = 10 → "A", not "10"!

0.9375 × 16 = 15 → 0.F16

$$26.625_{10} = 1A.A_{16}$$

Hex fraction places are 1/16, 1/256, 1/4096, 1/65536 … Each place is one-sixteenth of the previous one. Not every decimal fraction terminates in hex either; then stop after the required number of places.

★ **Decimal → hex: ÷ 16 and × 16, digits 10–15 written A–F.**

2.6 Binary, Octal and Hexadecimal to Decimal

Going **to** decimal needs no tables at all: write each digit's place value and add (the formula of §1.6). A bit 1 means "take this weight", a bit 0 means "skip it".

2 110.101 ₂

4 + 2 + 0 + ½ + 0 + ⅛
= 6.625_{10}

8 12.4 ₈

1×8 + 2×1 + 4×⅛
= 8 + 2 + 0.5 = 10.5_{10}

16 2D ₁₆

2×16 + 13×1
= 45_{10}

16 A3.4 ₁₆

10×16 + 3×1 + 4×1/16
= 160 + 3 + 0.25 = 163.25_{10}

WATCH IT! · COMMON MISTAKE 2.3 — WEIGHTED-SUM SLIPS

Using powers of 10 instead of the source base; writing 2^3 as 6; forgetting the fraction places ($8^{-1} = 0.125$, $16^{-1} = 0.0625$); adding the weights of the **0** bits too; using "A" as 1 instead of 10.

★ **Any base → decimal: Σ digit × base^{position}, negative positions after the point.**

2.7 Binary ↔ Octal: Groups of Three

Because $8 = 2^3$, each octal digit **is** a 3-bit group. **Binary → octal:** starting at the binary point, group the bits in threes, leftward for the integer part and rightward for the fraction; pad the outer groups with zeros; replace each group by its digit. **Octal → binary:** replace every octal digit by **all three** of its bits, keep the point where it is, and drop leading zeros only at the very end.

→ 11010110 ₂ to octal

011 010 110 → 3 2 6
= 326_8

→ 1011.011 ₂ to octal

001 011 . 011 → 1 3 . 3
= 13.3_8

257 $_8$ to binary

```
010 101 111
= 10101111₂
```

6.34 $_8$ to binary

```
110 . 011 100
= 110.0111₂ (trailing 00 dropped)
```

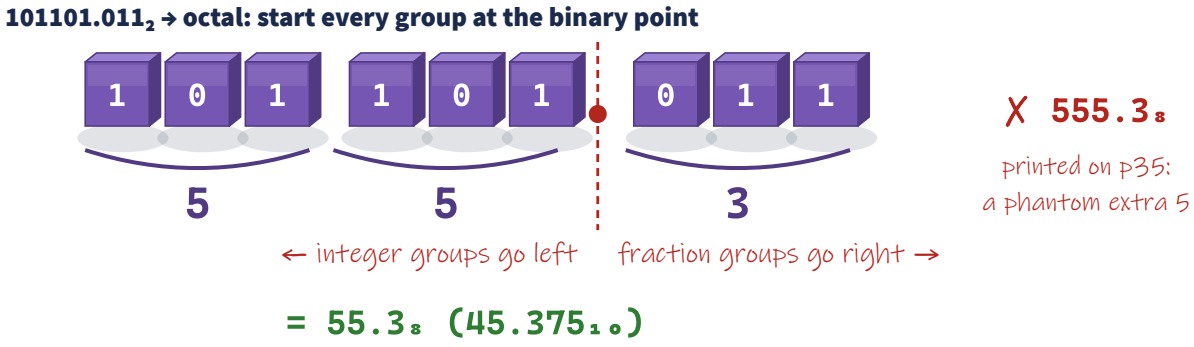


Figure 2.2 Groups always start at the binary point. Grouping the whole string from the left, or inventing an extra group, gives nonsense such as the 555.3_8 printed in the source.

★ Binary ↔ octal: 3 bits per digit, groups start at the point.

2.8 Binary ↔ Hexadecimal: Groups of Four

Same idea with **4-bit groups** ($16 = 2^4$). Pad the leftmost integer group with leading zeros and the rightmost fraction group with trailing zeros.

10111101 $_2$

```
1011 1101 → B D
= BD₁₆
```

1101011.101 $_2$

```
0110 1011 . 1010 → 6 B . A
= 6B.A₁₆
```

2F $_{16}$

```
0010 1111
= 101111₂
```

A.7 $_{16}$

```
1010 . 0111
= 1010.0111₂
```

There are no Dumb Questions

Q: **In 1101011.101 why do I pad the fraction to 1010 and not to 0101?**

A: Zeros added on the **right** of a fraction do not change its value (0.101 = 0.1010), but zeros added on the left do (0.0101 is a different number). For the integer part it is the other way round. Rule: **pad away from the point**.

★ Binary ↔ hex: 4 bits per digit; pad away from the point.

2.9 Octal ↔ Hexadecimal: The Binary Bridge

To go from octal to hex: **(1)** write 3 bits for every octal digit, **(2)** join them, keeping the point, **(3)** regroup in fours from the point outward, padding with zeros, **(4)** write the hex digits. Hex to octal is the mirror image (4-bit expansion, 3-bit regrouping).

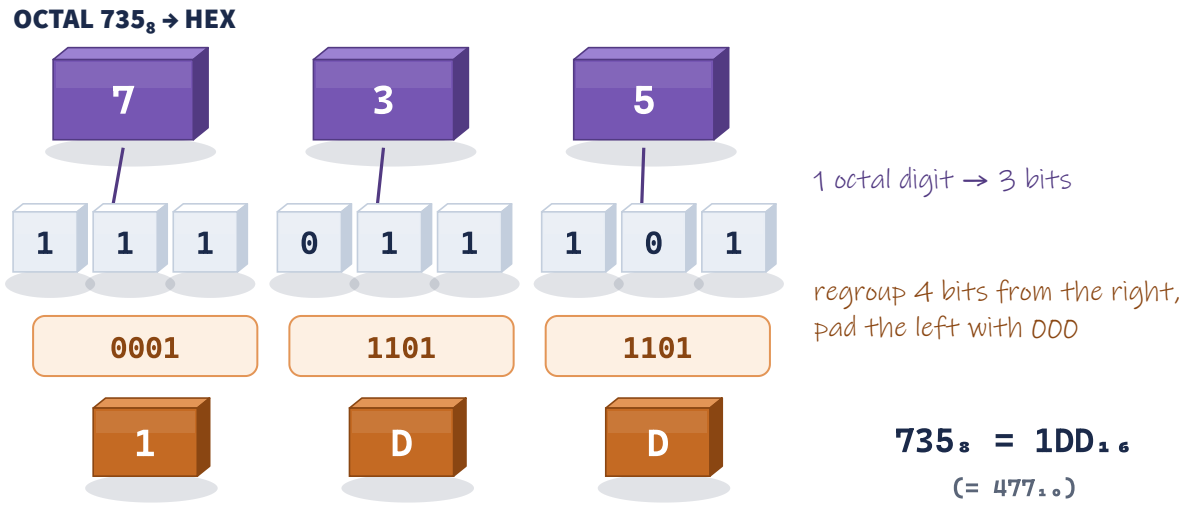


Figure 2.3 The binary bridge: $735_8 \rightarrow 111\ 011\ 101 \rightarrow 0001\ 1101\ 1101 \rightarrow 1DD_{16}$. Check: $7 \times 64 + 3 \times 8 + 5 = 477 = 1 \times 256 + 13 \times 16 + 13$.

8 24.6 8

```
010 100 . 110 → 0001 0100 . 1100
= 14.C16
```

16 9B 16

```
1001 1011 → 010 011 011
= 2338
```

16 2F.A 16

```
0010 1111 . 1010 → 101 111 . 101
= 57.58
```

8 157.3 8

```
001 101 111 . 011 → 0110 1111 . 0110
= 6F.616
```

★ Octal ↔ hex: expand to bits, regroup, never through a digit-to-digit table.

2.10 Fractions Across Bases

Three rules cover every fractional conversion: **decimal → base n:** multiply by n repeatedly, collect integer parts in order; **base n → decimal:** $(0.d_1d_2d_3\ldots)_n = d_1n^{-1} + d_2n^{-2} + d_3n^{-3} + \ldots$; **between 2, 8 and 16:** group from the point (3 bits for octal, 4 for hex).

Table 2.2 10.625_{10} **in all four bases**

Decimal	Binary	Octal	Hexadecimal
0.625	0.101	0.5	0.A
10.625	1010.101	12.5	A.A

Worked exactly, with no rounding, these conversions also show which fractions repeat:

WORKED RESULT

```
decimal  binary            octal           hex
10.625   1010.101          12.5            A.A
0.375    0.011             0.3             0.6
245      11110101          365             F5
478      111011110         736             1DE
0.1      0.000110011001... 0.063146314631... 0.199999999999...
```

Result 2.1 The two machines of this chapter in 15 lines. "..." marks a fraction that has not ended after 12 places: 0.1 never ends in binary, octal or hex, because 10 = 2 × 5 and 5 does not divide any power of 2.

★ **Fraction digits: multiply out of a decimal, add up into a decimal, group between 2, 8 and 16.**

2.11 Mixed-Base Conversion Problems

Strategy: identify the source and target base, choose the shortest path (direct grouping if both are powers of 2, else through decimal), convert carefully, then **verify by converting back** along another path.

1 **101101.011 $_{2}$ → 8, 16**

101 101 . 011 = **55.3$_{8}$**
0010 1101 . 0110 = **2D.6$_{16}$**
(= 45.375$_{10}$)

2 **7A.4 $_{16}$ → 10, 2**

7×16 + 10 + 4/16 = **122.25$_{10}$**
0111 1010 . 0100 = **1111010.01$_{2}$**

3 **365.2 $_{8}$ → 2, 16**

011 110 101 . 010 = **11110101.01$_{2}$**
1111 0101 . 0100 = **F5.4$_{16}$**
(= 245.25$_{10}$)

4 **45.625 $_{10}$ → 2, 8, 16**

101101.101$_{2}$ = 55.5$_{8}$
= 2D.A$_{16}$

WATCH IT! · COMMON MISTAKE 2.4 — TWO WRONG ANSWERS ON SOURCE P35

Example 1 is printed as 101101.011$_2$ = **555.3$_8$**. The bits form only two integer groups (101 101), so the answer is **55.3$_8$** (Figure 2.2); the hex answer 2D.6 is right. **Example 3** is printed as 365.2$_8$ = **1EA.4$_{16}$**. The 4-bit regrouping there added extra bits; 11110101.010 regroups as 1111 0101 . 0100 = **F5.4$_{16}$**. Check both with decimal: 45.375 and 245.25.

★ **Choose the short path, then check with the long one.**

2.12 Shortcut Tables for Fast Conversion

Table 2.3 Which method when

Division method	Multiplication method	Grouping method
Integer part, from decimal to another base: divide by the base, read remainders upward.	Fraction part, from decimal: multiply by the base, read integer parts downward.	Between binary, octal and hex: 3-bit groups for octal, 4-bit for hex, from the point outward.

Table 2.4 Place-value table for the four bases

Base	...	b^3	b^2	b^1	b^0	.	b^{-1}	b^{-2}
Decimal	...	1000	100	10	1	.	0.1	0.01
Binary	...	8	4	2	1	.	0.5	0.25
Octal	...	512	64	8	1	.	0.125	0.015625
Hexadecimal	...	4096	256	16	1	.	0.0625	0.00390625

Grouping shortcuts. ...101110.011_2 → 101 110 . 011 → **56.3_8**; ...11010110.1011_2 → 1101 0110 . 1011 → **$D6.B_{16}$**.

WATCH IT! · COMMON MISTAKE 2.5 — THE OCTAL → HEX SHORTCUT ON SOURCE P36

The shortcut panel converts 52.3_8 → 101 010 . 011 → "**$D2.3_{16}$**". The bits are right but the regrouping is not: 101010.011 regrouped in fours from the point is 0010 1010 . 0110 = **$2A.6_{16}$**. Check: $52.3_8 = 42.375_{10} = 2 \times 16 + 10 + 6/16$. ✓

MEMORY TRICK

"**3 for Octal, 4 for heX**" (the 3 looks like half an 8; X has four arms). Binary digits 0–1, octal 0–7, hex 0–9 A–F; A = 10 ... F = 15. Bases not linked by a power of 2? Go through decimal.

★ **Division, multiplication, grouping: know which tool fits which part.**

2.13 Worked Conversion Examples

Table 2.5 Solved examples (each answer verified by the reverse conversion)

Problem	Method	Working	Answer
93_{10} → binary	÷ 2	`r: 1 0 1 1 1 0 1 (read up)`	**1011101_2**
0.6875_{10} → binary	× 2	`1.375, 0.75, 1.5, 1.0`	**0.1011_2**
110101.101_2 → decimal	weighted sum	`32+16+4+1 + 0.5+0.125`	**53.625**
245_{10} → octal	÷ 8	`245=30·8+5, 30=3·8+6, 3`	**365_8**
478_{10} → hex	÷ 16	`r: 14(E), 13(D), 1`	**$1DE_{16}$**
$3F.8_{16}$ → decimal	weighted sum	`48 + 15 + 8/16`	**63.5**
57.3_8 → binary	3 bits/digit	`101 111 . 011`	**101111.011_2**
11101010.101_2 → hex	4-bit groups	`1110 1010 . 1010`	**$EA.A_{16}$**
$2D.4_{16}$ → octal	binary bridge	`101101.01 → 101 101 . 010`	**55.2_8**
146.7_8 → hex	binary bridge	`1100110.111 → 0110 0110 . 1110`	**$66.E_{16}$**

WATCH IT! · COMMON MISTAKE 2.6 — THE TWO BRIDGE EXAMPLES ON SOURCE P38

The source prints $2D.4_{16}$ = "**265.20_8**" and 146.7_8 = "**$16.E0_{16}$**". In both, the bits were regrouped starting from the **left end** of the string instead of from the point. Regrouped from the point: 0010 1101 . 0100 → 101 101 . 010 = **55.2_8** (= 45.25), and 001 100 110 . 111 → 0110 0110 . 1110 = **$66.E_{16}$** (= 102.875).

★ Every conversion has a check: go back and you must land where you started.

2.14 Practice Problems

✎ Sharpen your pencil 2

A. Decimal to other bases: $(125)_{10}$ → base 2 · $(345)_{10}$ → base 8 · $(1023)_{10}$ → base 16 · $(2024)_{10}$ → base 2 · $(466)_{10}$ → base 16

B. To decimal: $(1111011)_2$ · $(536)_8$ · $(7A3)_{16}$ · $(101100.11)_2$ · $(10111.011)_2$ · $(7.52)_8$ · $(7B)_{16}$

C. Direct: $(101101)_2$ → 8 · $(247)_8$ → 2 · $(A5)_{16}$ → 2 · $(1D7)_{16}$ → 8 · $(745)_8$ → 16 · $(11110010.0111)_2$ → 16 · $(3C.4)_{16}$ → 8

D. Fractions and mixed: $(57.625)_{10}$ → 2 · $(1011.101)_2$ → 10 · $(64.3)_8$ → 10 · $(A.B)_{16}$ → 2 · $(25.5)_{10}$ → 16 · $(0.375)_{10}$ → 2, 8, 16

E. Challenge: $(3C7.9)_{16}$ → 2 · $(110101.011)_2$ → 16 · $(152.625)_{10}$ → 2 · $(124.6)_8$ → 2 and 10

Who Does What?

Repeated division	•	• Hex ↔ octal
Repeated multiplication	•	• Any base → decimal
Weighted sum	•	• Decimal integer → any base
3-bit grouping	•	• Decimal fraction → any base
Binary bridge	•	• Binary ↔ octal

EXAM POINT — THE CONVERSION MCQS

- $(0.6875)_{10}$ in binary? Options 0.1011, 0.1101, 0.1110, 0.10101 → **0.1011**.
- Number of bits needed for a decimal number N: $\lfloor \log_2 N \rfloor + 1$ (e.g. 1023 needs 10 bits, 1024 needs 11).
- Remainders are read **bottom to top**; fraction integer parts **top to bottom**; groups start at the **point**.
- $(FF)_{16} = 255 = (377)_8 = (11111111)_2$: one byte, all ones.

✓ Solution · Sharpen your pencil 2

1. **A.** 1111101 · 531 · 3FF · 11111101000 · 1D2
2. **B.** 123 · 350 · 1955 · 44.75 · 23.375 · 7.65625 · 123
3. **C.** 55 · 10100111 · 10100101 · 727 · 1E5 · F2.7 · 74.2
4. **D.** 111001.101 · 11.625 · 52.375 · 1010.1011 · 19.8 · 0.011_2, 0.3_8, 0.6_{16}
5. **E.** 1111000111.1001 · 35.6 · 10011000.101 · 1010100.11_2 and 84.75_{10}

Who Does What? Repeated division → decimal integer to any base · repeated multiplication → decimal fraction to any base · weighted sum → any base to decimal · 3-bit grouping → binary ↔ octal · binary bridge → hex ↔ octal.

BULLET POINTS

- Conversion changes the form, never the value.
- Decimal integer → base b: divide by b, read remainders bottom to top.
- Decimal fraction → base b: multiply by b, read integer parts top to bottom.
- Mixed numbers: convert the two parts separately.
- Base b → decimal: Σ digit × $b^{position}$.
- Binary ↔ octal: 3-bit groups; binary ↔ hex: 4-bit groups.
- Groups start at the point; pad zeros away from the point.
- Octal ↔ hex: via binary (the bridge).
- Fractions whose denominator has a factor other than 2 never end in binary.
- Always verify by converting back.

Next: Chapter 3 adds, subtracts, multiplies and divides in binary, using the same column rules you know from decimal, only with carries at 2.

CHAPTER · UNIT I · NUMBER SYSTEMS

3 Binary Arithmetic

1 + 1 = 10. Once that stops looking strange, you can do anything a CPU does.

OPENING SITUATION

A car's odometer shows 0999 km. Drive one more kilometre and every wheel rolls over: 1000. Each "9 + 1" writes 0 and pushes a **carry** into the next wheel. Now imagine an odometer whose wheels only have the digits 0 and 1. It would go 0000, 0001, 0010, 0011, 0100… and it would roll over on every "1 + 1". That odometer is exactly what the adder inside a CPU does billions of times a second. In this chapter you add, subtract, multiply and divide with it, using the school methods you already know.

IN THIS CHAPTER

- 3.1 The Rules of Binary Arithmetic
- 3.2 Binary Addition
- 3.3 Binary Subtraction
- 3.4 Binary Multiplication: Shift and Add
- 3.5 Binary Division

YOU NEED

- Chapter 1: binary place values (1, 2, 4, 8 …).
- Chapter 2: converting to decimal, to check your answers.

3.1 The Rules of Binary Arithmetic

Binary arithmetic works with only two digits, 0 and 1, and everything else is the same as decimal: line the numbers up at the right, work column by column, and pass a **carry** (in addition) or take a **borrow** (in subtraction) from the next column. The only difference is *when*: in decimal you carry when a column reaches **10**; in binary, as soon as it reaches **2**.

Table 3.1 All the rules you need

Addition	Subtraction	Multiplication	Division
0 + 0 = 0	0 − 0 = 0	0 × 0 = 0	0 ÷ 1 = 0
0 + 1 = 1	1 − 0 = 1	0 × 1 = 0	1 ÷ 1 = 1
1 + 0 = 1	1 − 1 = 0	1 × 0 = 0	÷ 0 not allowed
1 + 1 = 10 (0, carry 1)	10 − 1 = 1 (0 − 1 with borrow)	1 × 1 = 1	
1 + 1 + 1 = 11 (1, carry 1)			

Decimal (base 10)

- Carry when a column ≥ 10
- 8 + 7 = 15
- 9 − 4 = 5
- Used for human calculation

Binary (base 2)

- Carry when a column ≥ 2
- 1 + 1 = 10
- 10 − 1 = 1
- Used by every computer, ALU and digital device



BRAIN POWER

Binary multiplication of single bits (0×0, 0×1, 1×0, 1×1) gives exactly the same table as the logical **AND** of two bits. What does that tell you about how hard it is to build a 1-bit multiplier in hardware?

★ Same column methods as decimal; the carry comes at 2 instead of 10.

3.2 Binary Addition

Write the numbers with their LSBs aligned. Add each column from right to left, **including the carry** that came in from the right. A column total of 2 writes **0** and carries 1; a total of 3 (1 + 1 + carry) writes **1** and carries 1. Never forget the final carry out of the MSB: it becomes a new leftmost bit.

Example 1 — 13 + 7

carry	1	1	1	1
	1	1	0	1
+	0	1	1	1
=	**1**	**0**	**1**	**0**	**0**

$1101_2 + 0111_2 = \mathbf{10100}_2$ (13 + 7 = 20)

Example 2 — 11 + 13

carry	1	1	1	1
	1	0	1	1
+	1	1	0	1
=	**1**	**1**	**0**	**0**	**0**

$1011_2 + 1101_2 = \mathbf{11000}_2$ (11 + 13 = 24)

Example 3 — a longer sum

carry	1	1	1	1		1	
		1	0	1	1	0	1
+			1	1	1	0	1
=	**1**	**0**	**0**	**1**	**0**	**1**	**0**

$101101_2 + 11101_2 = \mathbf{1001010}_2$ (45 + 29 = 74)

WATCH IT! · COMMON MISTAKE 3.1 — EXAMPLE 3 ON SOURCE P41

The source prints $101101_2 + 11101_2 = \mathbf{10010110_2}$. That is 150, but 45 + 29 = 74. The column work above gives $\mathbf{1001010_2}$ (= 74). Always check a binary sum by converting both sides to decimal.

WATCH IT! · COMMON MISTAKE 3.2 — THE USUAL ADDITION SLIPS

Writing "2" in a column instead of 0-carry-1; forgetting the carry when the column total is 2 or 3; misaligning the numbers (they must be right-aligned); dropping the final carry; adding from left to right.

★ Column total 2 → write 0 carry 1; total 3 → write 1 carry 1.

3.3 Binary Subtraction

Subtract column by column from the right. When the top bit is smaller than the bottom bit (0 − 1), **borrow** 1 from the next higher column: the current column gains 2 (written 10), so 10 − 1 = 1, and the column you borrowed from loses 1. If that column is 0 it must borrow in turn, so a borrow can ripple across several zeros.

No borrow needed: 11 − 5

borrow	−1			
	1	0	1	1
−	0	1	0	1
=	**0**	**1**	**1**	**0**

$1011_2 - 0101_2 = \mathbf{110_2}$ (11 − 5 = 6)

A borrow that ripples: 8 − 7

borrow	−1	−1	−1	
	1	0	0	0
−	0	1	1	1
=	**0**	**0**	**0**	**1**

$1000_2 - 0111_2 = \mathbf{1_2}$ (8 − 7 = 1)

Borrows across several columns: 26 − 7

borrow		−1	−1	−1	
	1	1	0	1	0
−	0	0	1	1	1
=	**1**	**0**	**0**	**1**	**1**

$11010_2 - 00111_2 = \mathbf{10011_2}$ (26 − 7 = 19)

The −1 marks show where a column had to give away 1 to its right-hand neighbour. Direct borrowing is fine on paper, but hardware prefers **adding the complement** of the subtraend (§4.6): then one adder circuit does both jobs.

There are no Dumb Questions

Q: **What happens if the bottom number is bigger, like 0101 − 1000?**

A: Direct subtraction then ends with a borrow that has nowhere to come from. That leftover borrow is the signal that the answer is **negative**. Representing negative answers is exactly what Chapter 4 is about.

★ 0 − 1: borrow 1 from the left, the column becomes 10, and 10 − 1 = 1.

3.4 Binary Multiplication: Shift and Add

Multiply the multiplicand by each bit of the multiplier, starting from the LSB. A **1** bit copies the multiplicand; a **0** bit gives a row of zeros. Each next row is shifted one place left (that is, multiplied by 2). Add all the partial products with binary addition.

1011 × 11

		1	0	1	1
×				1	1
bit 0 = 1		1	0	1	1
bit 1 = 1	1	0	1	1	0
=	1	0	0	0	0	1

$1011_2 \times 11_2 = \mathbf{100001}_2$ (11 × 3 = 33)

1101 × 1011

			1	1	0	1	
×			1	0	1	1	
bit 0 = 1			1	1	0	1	
bit 1 = 1		1	1	0	1	0	
bit 2 = 0	0	0	0	0	0	0	
bit 3 = 1	1	1	0	1	0	0	0
=	1	0	0	0	1	1	1	1

$1101_2 \times 1011_2 = \mathbf{10001111}_2$ (13 × 11 = 143)

THE SHIFT-AND-ADD ALGORITHM (WHAT THE ALU DOES)

1. Look at the multiplier's LSB. If it is 1, add the multiplicand to the running product.
2. Shift the multiplicand left by one place (× 2) and the multiplier right by one place (÷ 2).
3. Repeat until the multiplier becomes 0.

Only additions and shifts, no "times table" at all. Shifting left one place = multiplying by 2; shifting right = integer division by 2.

WATCH IT! · COMMON MISTAKE 3.3 — THE LARGER EXAMPLE ON SOURCE P43

The source prints $1101_2 \times 1011_2 = \mathbf{11100001_2}$ **(225)**. Its own partial products (1101, 11010, 0, 1101000) add to $\mathbf{10001111_2}$ **= 143**, and 13 × 11 = 143. (225 would be 15 × 15.)

★ **Multiply = copy for every 1 bit, shift, and add.**

3.5 Binary Division

Binary long division is easier than decimal long division, because each quotient bit can only be 0 or 1: either the divisor **fits** into the current partial dividend (write 1 and subtract) or it does not (write 0). Bring down the next bit and repeat. Whatever is left at the end is the remainder.

1011 ÷ 10 (11 ÷ 2)

Step	Partial dividend	Compare	Q bit	Subtract
1	bring down 1 → 1	< 10	0	keep 1
2	bring down 0 → 10	≥ 10	1	10 − 10 = 0
3	bring down 1 → 1	< 10	0	keep 1
4	bring down 1 → 11	≥ 10	1	11 − 10 = 1

$1011_2 \div 10_2$: quotient $\mathbf{101}_2$, remainder $\mathbf{1}_2$ (11 ÷ 2 = 5 r 1)

1011 ÷ 11 (11 ÷ 3)

Step	Partial dividend	Compare	Q bit	Subtract
1	bring down 1 → 1	< 11	0	keep 1
2	bring down 0 → 10	< 11	0	keep 10
3	bring down 1 → 101	≥ 11	1	101 − 11 = 10
4	bring down 1 → 101	≥ 11	1	101 − 11 = 10

$1011_2 \div 11_2$: quotient 11_2, remainder 10_2 $(11 \div 3 = 3 \text{ r } 2)$

CHECK EVERY DIVISION

Dividend = Divisor × Quotient + Remainder e.g. 1011 = 10 × 101 + 1 (11 = 2 × 5 + 1) and 1011 = 11 × 11 + 10 (11 = 3 × 3 + 2).

WATCH IT! · COMMON MISTAKE 3.4 — "THE REMAINDER IS ALWAYS 0 OR 1 IN BINARY"

The source (p44) says this. It is true only when you divide by $\mathbf{10_2}$ (2). In general the remainder is any binary number **smaller than the divisor**: $1011_2 \div 11_2$ leaves remainder $\mathbf{10_2}$ (= 2), as computed above.

And to tie the four operations together, the same answers straight from a computer:

WORKED RESULT

```
101101 + 11101 = 1001010
11010 - 00111  = 10011
1101 x 1011    = 10001111
1011 / 11      = quotient 11 remainder 10
1011 shifted left = 10110 (x 2);  shifted right = 101 (/ 2)
```

Result 3.1 The hand results confirmed: 1001010, 10011, 10001111, quotient 11 remainder 10. The last line shows a left shift (× 2) and a right shift (÷ 2), the shifts the ALU uses for multiplying and dividing by 2.

✎ Sharpen your pencil 3

1. Add: $101101_2 + 11011_2$ _____ · $110011_2 + 101101_2$ _____
2. Subtract: $101101_2 - 011011_2$ _____
3. Multiply: $10101_2 \times 111_2$ _____
4. Divide: $11011_2 \div 11_2$ (quotient and remainder) _____
5. What single operation multiplies a binary number by 8? _____

EXAM POINT

- 1 + 1 + 1 = **11** (sum 1, carry 1). 1 + 1 = **10**.
- Binary multiplication is **shift and add**; a left shift by k multiplies by 2^k.
- Remainder < divisor (not "always 0 or 1").
- Computers subtract by **adding the 2's complement** (next chapter).

✓ Solution · Sharpen your pencil 3

1. 1001000_2 (45 + 27 = 72) · 1100000_2 (51 + 45 = 96)
2. 010010_2 (45 − 27 = 18)
3. 10010011_2 (21 × 7 = 147)
4. Quotient 1001_2, remainder 0 (27 ÷ 3 = 9)
5. A left shift by 3 places (append 000).

BULLET POINTS

- Binary arithmetic = decimal methods with carry/borrow at 2.
- 1 + 1 = 10; 1 + 1 + 1 = 11.
- Borrowing makes a column 10 (= 2); the lender loses 1.
- Multiplication: partial products for the 1 bits, shifted, then added.
- Division: each quotient bit is 1 (fits, subtract) or 0.
- Dividend = divisor × quotient + remainder; remainder < divisor.
- Shift left = × 2; shift right = ÷ 2.
- Check every answer in decimal.

Next: Chapter 4 gives binary a minus sign, using sign-magnitude, 1's complement and 2's complement, and shows why the CPU subtracts by adding.

CHAPTER · UNIT I · NUMBER SYSTEMS

4 Signed Numbers and Complements

A wire cannot carry a minus sign. So the computer hides it inside the bits, and then subtracts by adding.

OPENING SITUATION

Your bank shows a balance of −₹250 after an overdraft. A thermometer in Ooty reads −3 °C in January. Negative numbers are everywhere, yet a memory cell can only hold 0 or 1; there is no third state for "−". The engineers' answer was to give up one bit, the leftmost, as a sign, and then to choose a clever *code* for the negative values so that one adder circuit handles plus and minus alike. Three such codes exist. Only one of them won, and by the end of this chapter you will know why.

IN THIS CHAPTER

- 4.1 Unsigned and Signed Numbers
- 4.2 Sign-Magnitude
- 4.3 One's Complement
- 4.4 Two's Complement
- 4.5 Ranges of the Three Systems
- 4.6 Subtraction by Adding the Complement
- 4.7 Carry, Overflow and Underflow

YOU NEED

- Chapter 3: binary addition with carries.
- The idea of a fixed word size (4, 8, 16 bits) from Chapter 1.

4.1 Unsigned and Signed Numbers

A register has a **fixed number of bits**, n. If every bit is a magnitude bit, the pattern is an **unsigned** number from 0 to $2^n - 1$. If the leftmost bit (the MSB) is reserved as a **sign bit**, 0 for + and 1 for −, the pattern is a **signed** number. The same eight bits can therefore mean two different things:

11011011 as unsigned

- All 8 bits are magnitude
- 128 + 64 + 16 + 8 + 2 + 1 = **219**
- Range 0 … 255

11011011 as signed (2's complement)

- MSB 1 ⇒ negative
- 219 − 256 = **−37**
- Range −128 … +127

The bits do not know which meaning you intend. The *instruction* decides: a CPU has, for example, separate "branch if lower" (unsigned) and "branch if less" (signed) tests. There are three ways to code the negative values, and they differ only for negative numbers: positive numbers look the same in all three.

± Sign-magnitude
Sign bit + the plain magnitude. Easy for humans.

1 1's complement
Negate by inverting every bit.

2 2's complement
Invert every bit, then add 1. Used by every modern CPU.

★ **Same bits, two meanings: the instruction chooses unsigned or signed.**

4.2 Sign-Magnitude

Write the magnitude in the lower n − 1 bits and put the sign in the MSB. +25 = `0 0011001`, −25 = `1 0011001`. Reading it back is just as easy: `1 0110010` has sign 1 and magnitude 0110010 = 50, so it is **−50**.

TWO PROBLEMS WITH SIGN-MAGNITUDE

- **Two zeros:** 0000 0000 (+0) and 1000 0000 (−0). Every "is it zero?" test must check both.
- **Addition needs decisions:** to add +5 and −3 the hardware must compare signs, compare magnitudes, subtract the smaller from the larger and pick the sign. That means extra circuitry and extra time.

Range for n bits: $-(2^{n-1}-1)$ … $+(2^{n-1}-1)$; for 8 bits, −127 … +127. Sign-magnitude survives today in one place: the sign bit of floating-point numbers.

★ Sign-magnitude: friendly to read, awkward to add, and it has a −0.

4.3 One's Complement

To negate a number, **invert every bit** (0 ↔ 1). +25 = 00011001, so −25 = **11100110**. Mathematically, the 1's complement of an n-bit N is $(2^n - 1) - N$. The hardware for it is just n NOT gates.

Addition works with one twist: if the addition produces a carry out of the MSB, that carry is **added back into the LSB**. This is the **end-around carry**.

(+5) + (−3), 4-bit 1's complement

carry	1			
	0	1	0	1
+	1	1	0	0
= 1 |	0	0	0	1
+ end-around				1
=	0	0	1	0

(5) + (−3) = **2**

the carry out of the MSB is added back to the LSB (end-around carry)

(+5) + (−6)

carry			1	
	0	1	0	1
+	1	0	0	1
=	1	1	1	0

(5) + (−6) = **−1**

no end-around carry: the result is negative, in 1's complement form

In the second sum there is no carry, so the result 1110 is already the 1's-complement code of −1. One's complement still has **two zeros** (0000 and 1111), and the end-around carry can make the adder take a second pass. Range: −127 … +127 in 8 bits, the same as sign-magnitude.

★ 1's complement: invert all bits; carry out of the MSB goes back round to the LSB.

4.4 Two's Complement

To negate, **invert every bit and add 1**. The 2's complement of an n-bit N is $2^n - N$. Each block below is computed step by step:

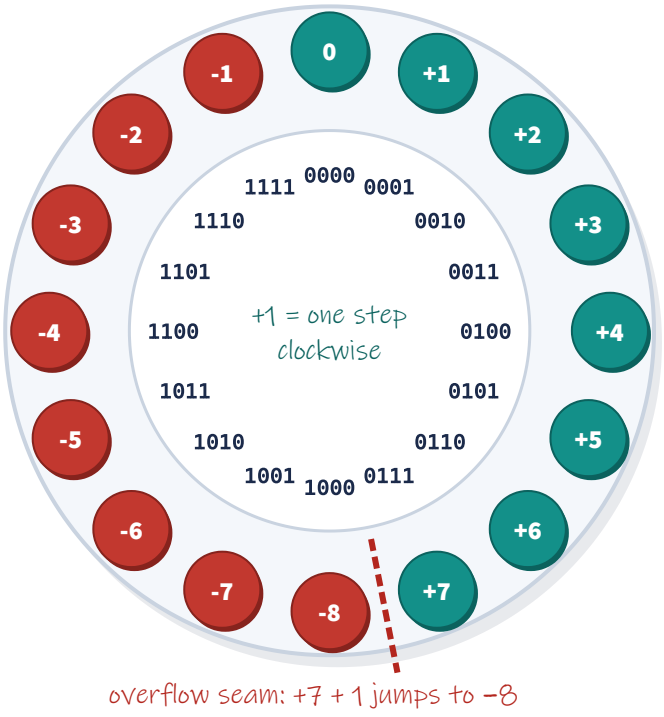
+37 in 8 bits	0010 0101
−37 sign-magnitude	1010 0101
−37 1's complement (invert)	1101 1010
−37 2's complement (invert + 1)	**1101 1011**

+12 in 8 bits	0000 1100
−12 sign-magnitude	1000 1100
−12 1's complement (invert)	1111 0011
−12 2's complement (invert + 1)	**1111 0100**

A SHORTCUT FOR DOING IT BY HAND

Copy the bits from the right up to and including the **first 1**; invert every bit to its left. +12 = 0000 1100 → copy the rightmost "100", invert the rest → **1111 0100** = −12. Same answer as invert-and-add-1, with no addition.

Why does everyone use it? Look at the wheel. Adding 1 always moves one step clockwise, for positive and negative numbers alike, so the *same* plain binary adder works for signed and unsigned numbers. There is only one zero, and the extra pattern 1000 becomes an extra negative number, −8.



4-bit two's complement

range	−8 … +7 (-2^{n-1} … $2^{n-1}-1$)
zero	only one: 0000
MSB	1 ⇒ negative (red half)
−1	1111 (all ones)
−8	1000 has no positive twin

Adding = walking round the wheel. Cross the seam ⇒ signed overflow. Pass 1111 → 0000 ⇒ only a carry out (harmless for signed numbers).

Figure 4.1 The 4-bit two's complement wheel. The bit patterns run 0000 → 1111 round the circle; the left (red) half has MSB 1 and holds −8 … −1. Stepping across the seam between 0111 and 1000 is signed overflow.

WATCH IT! · −128 HAS NO POSITIVE TWIN

In 8 bits, the 2's complement of 1000 0000 (−128) is: invert → 0111 1111, add 1 → **1000 0000** again. +128 does not exist in 8 bits, so negating −128 overflows. The range is **lopsided**: −128 … +127.

There are no Dumb Questions

Q: **To read a negative 2's complement number, do I have to undo the "add 1" first?**

A: No. Taking the 2's complement twice gives back the original, so just take it again: 1101 1011 → invert 0010 0100 → add 1 → 0010 0101 = 37, so the pattern meant −37. Faster still: give the MSB a **negative weight**, −128 + 64 + 16 + 8 + 2 + 1 = −37.

★ 2's complement: invert and add 1. One zero, one adder, range $-2^{n-1} \ldots 2^{n-1} - 1$.

4.5 Ranges of the Three Systems

Table 4.1 What n bits can hold

n	Unsigned	Sign-magnitude	1's complement	2's complement
general	$0 \ldots 2^n - 1$	$-(2^{n-1}-1) \ldots +(2^{n-1}-1)$	$-(2^{n-1}-1) \ldots +(2^{n-1}-1)$	**$-2^{n-1} \ldots +(2^{n-1}-1)$**
4	0 … 15	−7 … +7	−7 … +7	−8 … +7
8	0 … 255	−127 … +127	−127 … +127	−128 … +127
16	0 … 65 535	−32 767 … +32 767	−32 767 … +32 767	−32 768 … +32 767
zeros	one	two (+0, −0)	two (0000, 1111)	**one**

Table 4.2 All sixteen 4-bit patterns, read four ways

Bits	Unsigned	Sign-mag	1's comp	2's comp	Bits	Unsigned	Sign-mag	1's comp	2's comp
0000	0	+0	+0	0	1000	8	−0	−7	−8
0001	1	+1	+1	+1	1001	9	−1	−6	−7
0010	2	+2	+2	+2	1010	10	−2	−5	−6
0011	3	+3	+3	+3	1011	11	−3	−4	−5
0100	4	+4	+4	+4	1100	12	−4	−3	−4
0101	5	+5	+5	+5	1101	13	−5	−2	−3
0110	6	+6	+6	+6	1110	14	−6	−1	−2
0111	7	+7	+7	+7	1111	15	−7	−0	−1

★ Only 2's complement uses all 2^n patterns for different values.

4.6 Subtraction by Adding the Complement

The source (p49–50) shows the classic method for **unsigned** numbers, A − B, which needs no subtractor at all: add A to the complement of B and look at the carry out.

- **Carry out = 1** → the result is positive. In 1's complement add the carry back (end-around); in 2's complement just discard it.
- **No carry** → the result is negative, and its magnitude is the complement of the sum.

9 − 5 using 1's complement

A	1	0	0	1
+ 1's comp of B	1	0	1	0
= 1 |	0	0	1	1
+ end-around				1
=	0	1	0	0

9 − 5 = 4

end-around carry 1 added to the LSB → result positive

5 − 9 using 1's complement

A	0	1	0	1
+ 1's comp of B	0	1	1	0
=	1	0	1	1

5 − 9 = −4

no carry → result negative; magnitude = 1's complement of 1011 = 0100

9 − 5 using 2's complement

A	1	0	0	1
+ 2's comp of B	1	0	1	1
= 1 |	0	1	0	0

9 − 5 = 4

carry out 1 discarded → result positive

5 − 9 using 2's complement

A	0	1	0	1
+ 2's comp of B	0	1	1	1
=	1	1	0	0

5 − 9 = −4

no carry → result negative; magnitude = 2's complement of 1100 = 0100

WATCH IT! · THESE ARE UNSIGNED EXAMPLES

The 4-bit operands 1001 (9) and 0101 (5) on source p49–50 are **unsigned** values. In 4-bit *signed* 2's complement, 1001 means −7 and +9 cannot be written at all. For signed numbers you simply add: A − B = A + (2's complement of B), and the overflow test of §4.7 tells you whether the answer fits.

Fireside Chats · One's Complement vs Two's Complement

One's Complement	Two's Complement
I'm the simple one. Want −N? Flip every bit. A row of NOT gates and you're done.	And then you add a 1. Big deal. It's one carry-in on the adder that's already there.
But you can't deny I'm symmetric. −127 to +127, the same on both sides.	Symmetric, with *two* zeros. Every compare has to ask "is it 0000 or is it 1111?"
The end-around carry handles everything.	The end-around carry makes the adder go round twice. I just throw the carry away.
Old machines like the CDC 6600 and the UNIVAC used me!	And every phone, laptop and microcontroller made today uses me. Nothing personal.

★ **Subtract by adding the complement: carry decides the sign of an unsigned result.**

4.7 Carry, Overflow and Underflow

An n-bit adder always produces n bits. If the true answer needs more, something has to tell you. Two different flags do that job, and they answer two different questions:

Carry (C)

- A carry out of the MSB
- Means: the **unsigned** result did not fit
- For 2's complement signed sums it is normal and is ignored

Overflow (V)

- The **signed** result is outside the range
- Only possible when both operands have the **same sign**
- Detected as $V = C_{in}(MSB) \oplus C_{out}$

(+5) + (+4) in 4 bits

carry	1			
	0	1	0	1
+	0	1	0	0
=	1	0	0	1

OVERFLOW (V = 1): 5 + 4 = 9 is outside the 4-bit range; the bits read −7

$C_{in}(MSB) = 1$, $C_{out} = 0$, $V = C_{in} \oplus C_{out} = 1$

(−2) + (−4)

carry	1			
	1	1	1	0
+	1	1	0	0
= 1 |	1	0	1	0

(−2) + (−4) = −6

carry out of the MSB discarded; $C_{in}(MSB) = 1$, $C_{out} = 1$, $V = C_{in} \oplus C_{out} = 0$

(−6) − 4 = (−6) + (−4)

carry				
	1	0	1	0
+	1	1	0	0
= 1 |	0	1	1	0

OVERFLOW (V = 1): −6 + −4 = −10 is outside the 4-bit range; the bits read 6

carry out of the MSB discarded; $C_{in}(MSB) = 0$, $C_{out} = 1$, $V = C_{in} \oplus C_{out} = 1$

(+7) + (+3)

carry	1	1	1	
	0	1	1	1
+	0	0	1	1
=	1	0	1	0

OVERFLOW (V = 1): 7 + 3 = 10 is outside the 4-bit range; the bits read −6

$C_{in}(MSB) = 1$, $C_{out} = 0$, $V = C_{in} \oplus C_{out} = 1$

Read them against the wheel in Figure 4.1. (−2) + (−4) passes 1111 → 0000 and produces a carry, but it never crosses the seam, so V = 0 and −6 is correct. (+5) + (+4) and (+7) + (+3) cross the seam clockwise, (−6) + (−4) crosses it anticlockwise: all three overflow. A second rule gives the same verdict: **overflow happens exactly when two operands of the same sign give a result of the opposite sign**.

WATCH IT! · COMMON MISTAKE 4.1 — "UNDERFLOW" ON SOURCE P52

The source calls (−6) − 4 in 4 bits an **underflow**. For integers this is still **overflow** (negative overflow): the true result, −10, is below the smallest value, −8, and V = 1. The word **underflow** belongs to floating-point arithmetic, where it means a non-zero result too close to 0 to be represented.

Keeping only the low 4 (or 8) bits shows exactly what the hardware sees:

WORKED RESULT

```
0111 -> 7
1000 -> -8
1111 -> -1
11011011 -> -37
11011011
5 + 4 in 4 bits: 1001 = -7
```

Result 4.1 Reading patterns as signed numbers (MSB has weight -2^{n-1}). Kept to 8 bits, −37 is 11011011. 5 + 4 kept to 4 bits reads as −7: overflow.

Who Does What?

End-around carry	•	• $C_{in}(MSB) \oplus C_{out}$
Two zeros	•	• Floating-point result too close to 0
Overflow flag V	•	• 1's complement addition
Invert and add 1	•	• Sign-magnitude and 1's complement
Underflow	•	• 2's complement

✎ Sharpen your pencil 4

1. Write −25 in 8-bit sign-magnitude, 1's complement and 2's complement. _____
2. What decimal value is the 8-bit 2's complement pattern 1111 0100? _____
3. 8-bit 2's complement: 0100 0000 + 0100 0000. What are the bits, and is there overflow? _____
4. Unsigned 4-bit 0110 − 1010 by adding the 2's complement. _____
5. What is the range of a 6-bit 2's complement number? _____

EXAM POINT

- 2's complement = 1's complement + 1 = $2^n - N$.
- Range of n-bit 2's complement: $\mathbf{-2^{n-1}}$ **to** $\mathbf{2^{n-1} - 1}$ (8-bit: −128 … +127).
- 1's complement and sign-magnitude have two zeros; 2's complement has one.
- Overflow: $\mathbf{V = C_{in} \oplus C_{out}}$ of the sign bit; only when both operands have the same sign.

✓ Solution · Sharpen your pencil 4

1. +25 = 0001 1001 → sign-magnitude **1001 1001**, 1's complement **1110 0110**, 2's complement **1110 0111**.
2. 244 − 256 = **−12** (or invert 0000 1011, add 1 → 0000 1100 = 12).
3. Bits 1000 0000, which read as −128. 64 + 64 = 128 does not fit: **overflow** (C_{in} = 1, C_{out} = 0, V = 1).
4. 2's complement of 1010 is 0110; 0110 + 0110 = 1100 with no carry → negative; 2's complement of 1100 = 0100 → **−4** (6 − 10).
5. $-2^5 \ldots 2^5 - 1$ = **−32 … +31**.

Who does what: end-around carry → 1's complement addition; two zeros → sign-magnitude and 1's complement; V → $C_{in} \oplus C_{out}$; invert and add 1 → 2's complement; underflow → floating-point result too close to 0.

BULLET POINTS

- The MSB is the sign bit in all three signed systems.
- Positive numbers look the same in all three.
- Sign-magnitude: sign + magnitude; two zeros.
- 1's complement: invert; end-around carry; two zeros.
- 2's complement: invert + 1; one zero; one adder for everything.
- 2's complement range is lopsided: one extra negative value.
- Carry = unsigned overflow; V = signed overflow.
- "Underflow" is a floating-point word, not an integer one.

That completes Part I. Part II asks a different question: if bits can stand for numbers, how else can they stand for decimal digits, letters and error-proof messages?

PART II · UNIT I SYLLABUS 2

Codes

Chapter 5 Decimal Codes: BCD and Weighted Codes

Chapter 6 Excess-3, Gray and Alphanumeric Codes

Chapter 7 Error Detection and Correction

Digital codes and their classification · weighted codes (8421 BCD, 2421, 5211, 7421, 84-2-1) · BCD addition · self-complementing codes · Excess-3 · Gray code · ASCII, EBCDIC and Unicode · parity, checksums and the Hamming code.

A pattern of bits means nothing until someone agrees what it stands for. That agreement is a code, and the right code can make a display simpler, a dial safer or a message self-repairing.

CHAPTER · UNIT I · CODES

5 Decimal Codes: BCD and Weighted Codes

Your calculator does not think in binary. It thinks in decimal digits, each one wrapped in four bits.

OPENING SITUATION

Type 305 on a calculator. The display must light the segments for 3, 0 and 5. If the calculator stored 305 in pure binary (100110001), it would have to convert back to decimal digits every time it showed a result. So calculators, digital clocks, petrol-pump displays and electricity meters do something simpler: they keep **each decimal digit in its own group of four bits**. That is a **decimal code**. This chapter shows the most common one, 8421 BCD, how to add with it, and four cousins whose odd-looking weights (2421, 5211, 7421, 84-2-1) buy a useful trick called self-complementing.

IN THIS CHAPTER

- 5.1 What Is a Digital Code?
- 5.2 How Codes Are Classified
- 5.3 Weighted and Non-Weighted Codes
- 5.4 8421 BCD
- 5.5 Valid and Invalid BCD Codes
- 5.6 BCD Addition and the +6 Correction
- 5.7 Other Weighted Codes: 2421, 5211, 7421 and 84-2-1
- 5.8 Self-Complementing Codes

YOU NEED

- Chapter 1: place values and the word "nibble".
- Chapter 3: binary addition with carries.

5.1 What Is a Digital Code?

A **digital code** is an agreed set of bit patterns, each standing for one symbol: a digit, a letter, a punctuation mark, a control signal. Each individual pattern is a **code word**. In 8421 BCD the code word for the digit 5 is `0101`; in ASCII the code word for the letter A is `0100 0001`. Pure binary is about *quantity*; a code is about *meaning*.

123 Numeric codes
Represent digits or numbers: 8421 BCD, 2421, Excess-3, Gray.

A Alphanumeric codes
Letters, digits and symbols: ASCII, EBCDIC, Unicode.

✓ Error-control codes
Detect or correct errors: parity, checksum, CRC, Hamming.

WHAT MAKES A GOOD CODE

- **Uniqueness:** every symbol has exactly one code word, and no two symbols share one.
- **Fixed length:** all code words have the same number of bits, so the receiver knows where each one ends.
- **Easy conversion** to and from the forms people and devices use.
- **Efficiency:** no more bits than the job needs, unless the extra bits buy error detection.

★ **A code is an agreement: this bit pattern means that symbol.**

5.2 How Codes Are Classified

Codes are grouped by **purpose** (numeric, alphanumeric, error-detecting, error-correcting) and by **structure**. Three structural questions come up again and again:

Table 5.1 Three structural questions

Question	Yes	No
Weighted? Does each bit position carry a fixed value?	8421, 2421, 5211, 7421, 84-2-1	Excess-3, Gray, ASCII
Sequential? Is each code word the previous one plus 1 in binary?	8421 BCD, Excess-3	Gray, 2421 (jumps from 0100 to 1011)
Self-complementing? Does inverting all bits of the code for d give the code for 9 − d?	2421, 5211, 84-2-1, Excess-3	8421 BCD, 5421, 7421, Gray

There are no Dumb Questions

Q: **Can a code with more bits do the job better automatically?**

A: No. Extra bits help only if they are used for something, such as a parity bit for error detection. Otherwise they just cost storage and wires. Parity bits, by the way, can *detect* a single error but cannot *correct* it (Chapter 7).

★ **Weighted? Sequential? Self-complementing? Three questions classify most codes.**

5.3 Weighted and Non-Weighted Codes

In a **weighted code** each of the four bit positions has a fixed weight, and the digit is the **sum of the weights of the 1-bits**. The name of the code lists the weights from MSB to LSB: 8421, 2421, 5211, 7421, 84-2-1 (weights 8, 4, −2, −1).

DECODE 0101 IN FOUR WEIGHTED CODES

```
8421: 4 + 1 = 5
2421: 4 + 1 = 5 by weight, but 0101 is
not used (5 is 1011)
5211: 2 + 1 = 3
84−2−1: 4 − 1 = 3
```

NON-WEIGHTED CODES

The patterns are assigned by a rule other than a weight sum. **Excess-3**: code = d + 3 in binary, so 5 → 1000, and 1000 is not "5" under any weights. **Gray code**: neighbours differ in one bit. **ASCII**: code points assigned to characters.

WATCH IT! · COMMON MISTAKE 5.1 — A WEIGHT SUM IS NOT ENOUGH

Some weighted codes allow two patterns with the same sum: in 2421, both 0101 and 1011 add to 5. The code therefore needs a fixed **standard assignment** table (Table 5.2). Only patterns in that table are valid code words.

★ **Weighted: value = sum of the weights of the 1-bits. Non-weighted: value by some other rule.**