COMPUTER SYSTEMS & NUMBER REPRESENTATION

Giant open-book exam cheat sheet (NOTES A + NOTES B) + extra high-value exam add-ons.

Table of Contents

Headings are clickable in most PDF viewers.

NOTES A — DATA REPRESENTATION & NUMBER SYSTEMS

A1) Modern computer as a binary, digital logic system

A modern computer is **binary** and **digital**: it encodes information using two discrete states (0/1). Physically, those states are implemented with reliable voltage ranges (LOW/HIGH). All computation is built from **digital logic** (Boolean algebra) implemented by logic gates and larger blocks (adders, multiplexers, registers, memory).

- Binary is robust to noise: small voltage changes usually don't flip the interpreted 0/1.
- Gates combine into circuits: adders, multiplexers, decoders, registers, memory arrays.
- Programs become binary **machine instructions** stored in memory and executed by the CPU.

A2) Definitions: bit, nibble, byte, word, word size

Term	Definition
Bit	One binary digit (0 or 1). Smallest unit of information.
Nibble	4 bits.
Byte	8 bits. Commonly the smallest addressable unit in memory.
Word	The CPU's natural data unit (what it processes efficiently).
Word size	Number of bits in a word (e.g., 16, 32, 64). Often matches register width and influences addressing.

Add-on) What word size affects (why it matters)

- **Register width**: how big integers/pointers can be in a single register.
- **ALU operation width**: how many bits are added/ANDed/etc. per instruction.
- **Addressing / pointers**: often related to how large an address can be (platform dependent).
- **Performance**: wider words can move/process more data per instruction (but can increase storage/bandwidth needs).

A3) Positional number systems + converting base-b to decimal

In base **b**, each digit position is a power of **b**. For digits $d_n \ldots d_0$, the value is:

Formula: $(d_n \ldots d_0)_b = \Sigma\, d_i \cdot b^i$

Worked Example: Convert 231_4 to decimal.

```
Digits: 2, 3, 1 with powers 4^2, 4^1, 4^0
= 2·16 + 3·4 + 1·1 = 32 + 12 + 1 = 45
```

Quick tip: Allowed digits in base b are 0 to b–1.

A4) Converting to/from binary, octal, hexadecimal

Common bases: binary (2), octal (8), hexadecimal (16).

Binary → Decimal: sum of powers of 2 where bit = 1.

Example: $101101_2 = 1\cdot32 + 0\cdot16 + 1\cdot8 + 1\cdot4 + 0\cdot2 + 1\cdot1 = \mathbf{45}$

Decimal → Binary (division method): repeatedly divide by 2, record remainders, read upward.

Example: Convert 45_{10} to binary
45÷2=22 r1
22÷2=11 r0
11÷2=5 r1
5÷2=2 r1
2÷2=1 r0
1÷2=0 r1
Read remainders bottom→top: $\mathbf{101101}_2$

Add-on) Fractions in base conversions (if they show up)

- **Base-b fraction → decimal:** digits after point use negative powers: $d_{-1}b^{-1} + d_{-2}b^{-2} + \ldots$
- **Decimal fraction → base-b:** repeatedly multiply fractional part by b; the integer parts become the digits.

Example: Convert 0.625_{10} to binary
0.625×2=1.25 → 1
0.25×2=0.5 → 0
0.5×2=1.0 → 1
So $0.625 = 0.101_2$

Binary ↔ Hex: group bits into 4 (nibbles).

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

Worked Example: $1101\ 0010\ 0111_2 = D27_{16}$

Binary ↔ Octal: group bits into 3.

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

Worked Example: $101\ 101\ 001_2 = 551_8$

A5) Examples: what binary data can represent

Binary is just a pattern. The **interpretation** depends on the agreed format/encoding.

- **Unsigned integers** (e.g., 0..255 in 8 bits).
- **Signed integers** (two's complement).
- **Characters** (ASCII, Unicode).
- **Pixels** (RGB), images, video frames.
- **Audio samples** (digital wave values).
- **Instructions** (machine code patterns).
- **Memory addresses** (where something is stored).
- **Flags / booleans** (true/false).

A6) Range of n-bit unsigned integers

With n bits, there are 2^n distinct patterns. For **unsigned** integers the smallest is all zeros and the largest is all ones.

Range: 0 to 2^n – 1

Examples: 4-bit: 0..15, 8-bit: 0..255, 16-bit: 0..65535

A7) Negative numbers + range of n-bit signed integers (two's complement)

Most systems use **two's complement** for signed integers. The leftmost bit is the **sign bit** (0 = non-negative, 1 = negative).

Range: –2^(n–1) to 2^(n–1) – 1

Why asymmetric? 1000…0 represents –2^(n–1), giving one extra negative value.

Compute –x: invert all bits then add 1.

```
Example: Find –10 in 8-bit two's complement
10 = 00001010
invert → 11110101
+1 → 11110110
```

Add-on) Fast way to interpret a two's complement bit pattern

- If MSB = 0 → interpret as normal unsigned value (it's non-negative).
- If MSB = 1 → it's negative: invert bits + 1 to get the magnitude, then apply minus sign.

```
Example: 11101000₂
MSB=1 → negative. Invert: 00010111, +1 → 00011000 = 24 ⇒ value is –24
```

A8) Signed integer operations: addition, negation, subtraction, sign extension

Addition (two's complement): add like unsigned; ignore carry out of MSB. Detect overflow separately.

Add-on) Carry vs overflow (don't mix these)

- **Unsigned overflow**: carry out of the MSB (result didn't fit 0..2^n–1).

- **Signed overflow**: adding two same-sign numbers gives a result with opposite sign.

Operation	Overflow check
Unsigned add	Carry out of MSB = 1 ⇒ overflow
Signed add	Same sign inputs, different sign output ⇒ overflow

Worked Example (no signed overflow): (–10) + (7) in 8-bit

```
–10 = 11110110
7 = 00000111
sum = 11111101 = –3
```

Worked Example (signed overflow): 100 + 50 in 8-bit signed

```
100 = 01100100
50 = 00110010
sum = 10010110 (negative) ⇒ signed overflow (positive + positive became negative)
```

Negation: –x = invert bits + 1

Edge case: the most negative number (1000…0) negates to itself (overflow).

Subtraction: A – B = A + (–B)

```
Example: 12 – 5
12 = 00001100
–5: 00000101 → invert 11111010 → +1 11111011
Add: 00001100 + 11111011 = 00000111 = 7
```

Sign extension: increasing bit-width keeps the value by replicating the sign bit.

```
Example: extend 8-bit –10 to 16-bit
8-bit: 11110110
16-bit: 11111111 11110110
```

Add-on) Sign extension vs zero extension

- **Zero extension** (unsigned): prepend 0s when increasing width.
- **Sign extension** (signed): prepend copies of the sign bit (0 for non-negative, 1 for negative).

```
Example: 8-bit 10110110
As unsigned (182) → zero-extend to 16-bit: 00000000 10110110
As signed (–74) → sign-extend to 16-bit: 11111111 10110110
```

NOTES B — COMPUTER ARCHITECTURE & EXECUTION

B1) Layers of abstraction + hardware vs software

Computers are designed in layers so humans can work at higher levels without thinking about transistors.

Typical layers: hardware → machine code → assembly → high-level languages → applications

Hardware is the physical machinery (circuits, CPU, memory, buses). **Software** is the set of instructions and data that tell hardware what to do.

B2) Machine code vs assembly vs higher-level languages

- **Machine code**: raw binary instructions executed directly by the CPU.
- **Assembly**: human-readable mnemonics that map closely to machine code; assembled into machine code.
- **High-level languages**: more abstract; compiled/interpreted down to machine code (often through intermediate steps).

Add-on) What an instruction looks like (generic idea)

- Most instructions contain an **opcode** (what to do) plus **operand fields** (what to use).
- Operands might be registers, an immediate constant, or a memory address.
- Assembly is essentially a readable form of these fields (e.g., ADD R1, R2).

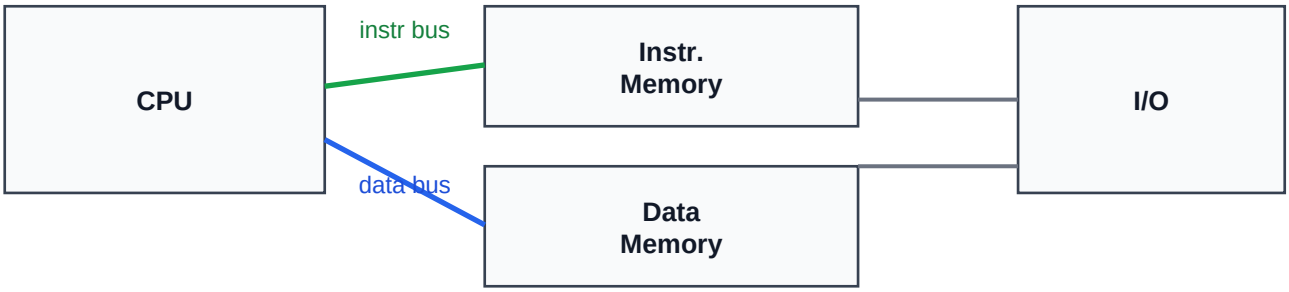
```
Generic format idea: [ opcode | reg fields | immediate / address bits ]
```

B3) Von Neumann vs Harvard; buses and control signals

Both architectures use CPU + memory + I/O, but they differ in how instructions/data are stored and accessed.

Von Neumann (stored-program): one memory for instructions + data

Harvard: separate instruction and data memories (can access both in same cycle)



Feature	Von Neumann	Harvard
Instruction & data memory	Shared	Separate
Buses	Shared instruction/data bus	Separate buses (can fetch instruction and access data simultaneously)
Advantage	Simpler design; flexible memory use	Higher throughput; avoids bottleneck
Disadvantage	Von Neumann bottleneck (bus contention)	More complex; less flexible memory sharing

Buses & signals (key terms):

- **Address bus**: carries the address (which memory location or I/O register to access).
- **Data bus**: carries the actual data being read/written.
- **Control signals**: indicate operations and timing (READ/WRITE, clock, enable, interrupts, etc.).

Add-on) Real-world note: “modified Harvard” is common

Many modern CPUs behave like von Neumann from a programmer viewpoint (one address space), but internally use separate **instruction/data caches** (Harvard-like) to improve throughput.

B4) Memory subsystem basics + capacity formulas

A memory subsystem stores bits in an array of storage cells. The CPU supplies an **address**; an address decoder selects a location; control signals choose read/write; the data bus transfers values.

Capacity relationships:

- Address bus with A bits selects 2^A locations.
- If each location holds W bits, total capacity is 2^A × W bits.
- Convert bits → bytes: divide by 8.

Formula: Capacity = 2^A locations × W bits/location

Worked Example: A = 16, W = 8 bits (byte-addressable).

```
Locations = 2^16 = 65536 addresses
Capacity = 65536 bytes = 64 KiB
```

Worked Example: A = 20, W = 16 bits.

```
Locations = 2^20 = 1,048,576
Capacity = 1,048,576 × 16 bits = 16,777,216 bits
= 2,097,152 bytes ≈ 2 MiB
```

Add-on) What an address usually means (byte addressing)

- Most systems are **byte-addressable**: each address points to 1 byte.
- A multi-byte value uses consecutive addresses.
- Example: a 32-bit (4-byte) word at address 0x1000 uses bytes at 0x1000, 0x1001, 0x1002, 0x1003.

Add-on) Endianness (byte order in memory)

Type	Meaning	Example storing 0x12 34 56 78 at address 0x1000
Big-endian	Most significant byte at lowest address	0x1000:12, 0x1001:34, 0x1002:56, 0x1003:78
Little-endian	Least significant byte at lowest address	0x1000:78, 0x1001:56, 0x1002:34, 0x1003:12

B5) Roles of CPU and memory; code vs data

CPU: executes instructions, performs arithmetic/logic, controls data movement. **Memory**: stores both instructions (code) and values (data).

- **Code** = instruction bytes that tell CPU what to do.
- **Data** = operands/results the program works with.
- Same bits can be interpreted as code or data depending on context.

B6) CPU executing machine code: basic step sequence

The CPU repeats a loop called the **instruction cycle**:

- **Fetch**: PC selects next instruction; read it from memory.
- **Decode**: determine operation + operand sources.
- **Fetch operands**: read registers/memory if needed.
- **Execute**: ALU operation, address calculation, branch decision, etc.
- **Write back**: store result to register/memory.
- **Update PC**: increment or replace via branch/jump.

Add-on) CPU building blocks (super common exam box)

Block	Role
ALU (Arithmetic Logic Unit)	Does arithmetic and bitwise logic (add, sub, AND, OR, shifts, compares).
Control Unit	Coordinates fetch/decode/execute and generates control signals.
Registers	Fast storage for operands, addresses, and control state.
Clock	Synchronizes operations; defines timing of state updates.

B7) Registers vs memory

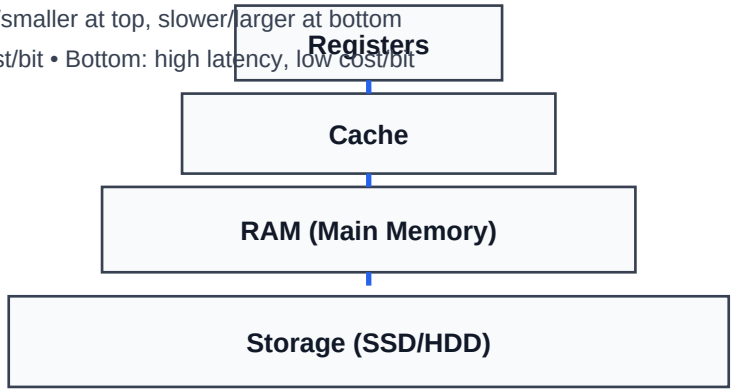
Registers are small storage locations **inside** the CPU; memory is larger storage **outside** the CPU.

	Registers	Memory (RAM)
Location	Inside CPU	Separate chips / memory subsystem
Speed	Fastest	Slower than registers
Capacity	Small (dozens)	Large (GB)
Use	Operands, intermediate results, control state	Program + large data structures

Add-on) Memory hierarchy (why memory “speed” varies)

The CPU can access registers much faster than RAM. Caches sit between CPU and RAM to reduce average access time.

Memory hierarchy: faster/smaller at top, slower/larger at bottom
Top: low latency, high cost/bit • Bottom: high latency, low cost/bit



B8) CPU programmer’s model + key registers + what it hides

The **programmer’s model** is the simplified view of the CPU state and instructions available to the programmer/assembler.

Key registers (definitions):

- **Data register**: holds data values used by ALU operations (operands/results).
- **Address register**: holds memory addresses (points to data or I/O locations).
- **Program Counter (PC)**: holds the address of the next instruction to fetch.
- **Status register** (flags): condition codes like Zero (Z), Negative (N), Carry (C), Overflow (V).

What’s missing from the programmer’s model:

- Pipeline stages (overlapping instruction work).
- Caches and cache misses.
- Branch prediction/speculation.
- Microcode / internal control logic.
- Exact timing (cycles per instruction) and bus activity.

QUICK REFERENCE — FORMULAS & CHECKLISTS

Number systems

Value in base b: Σ d_i · b^i

- Unsigned n-bit range: 0 .. 2^n – 1
- Signed (two’s complement) n-bit range: –2^(n–1) .. 2^(n–1) – 1
- –x in two’s complement: invert bits + 1
- A – B: A + (–B)
- Zero extend (unsigned): add leading 0s
- Sign extend (signed): replicate sign bit

Overflow quick tests

- Unsigned add: carry out of MSB ⇒ overflow
- Signed add: same-sign inputs but opposite-sign output ⇒ overflow

Memory capacity

Capacity = 2^A × W bits (A = address bits, W = bits per location)

Endianness

- Big-endian: MSB at lowest address
- Little-endian: LSB at lowest address