DSC-204A: Scalable Data Systems, Fall 2025

Basics: Computer Organization, Operating systems, Storage - 2

Lecturer: Hao Zhang *Scribe: Dongting Cai, Zhihan Li, Kaiming Tao, Zihan Zhou*

1 Floating Point Questions

1.1 Practice Q2: What do the exponent and the fraction control?

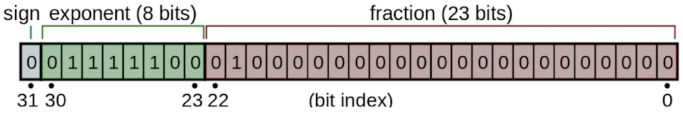


Figure 1: Practice Q2 - A 32-bit floats

1.1.1 Short Answer

Part	Bits	Role
Sign	1	Determines if number is positive or negative
Exponent	8	Controls the scale or range of the number
Fraction	23	Controls the precision or detail of the number

Table 1: Summary of 32-bit Floating-Point Components

The value of a 32-bit floating-point number can be calculated as:

$$\text{Value} = (-1)^{\text{sign}} \times 2^{(\text{exponent}-\text{bias})} \times (1 + \text{fraction})$$

Where fraction $= \sum_{i=1}^{23} b_{23-i} 2^{-i}$.

1.1.2 Further Explanation - What the Sign Does

Sign tells us whether the number is positive or negative.

Sign Bit	Meaning	Result
0	Positive	$(-1)^0 = +1$
1	Negative	$(-1)^1 = -1$

Table 2: Interpretation of Sign

1.1.3 Further Explanation - What the Exponent Does

A floating-point number (e.g. 3.14) is stored in computers kind of like scientific notation, but in **binary**.

In decimals, we can write big or small numbers using powers of 10:

$$3.14 \times 10^0 = 3.14$$

$$3.14 \times 10^3 = 3140$$

$$3.14 \times 10^{-3} = 0.00314$$

The **exponent** (the number above the 10) tells us **how many times to move the decimal point**:

- 10^3 moves it 3 places right → makes the number larger
- 10^-3 moves it 3 places left → makes the number larger

When computers represent decimals as floating-point numbers, their choice of exponent is **powers of 2** instead of powers of 10. Basically, exponent answers the question of "how many times should I move the binary point (the 'decimal' point in binary) left or right".

Think of it like this:

Exponent (after subtracting bias)	**Effect**	**Value**
-3	Move point 3 places left (divide by $2^3 = 8$)	0.125
0	No shift (multiply by $2^0 = 1$)	1.0
+3	Move point 3 places right (multiply by $2^3 = 8$)	8.0

Table 3: Floating-Point Exponent Examples

So, **A larger exponent** → "move point right" → makes the number bigger,
A small exponent → "move point left" → makes the number smaller.

That's why it's called **floating point** — the *point* "floats" left or right depending on the exponent.

1.1.4 Further Explanation - What is Bias and Why We Need It

Computers store numbers in binary, and the exponent bits can only store non-negative integers. However, we need to represent both positive and negative exponents.

To fulfill this need, **bias** is introduced as a fixed number that shifts the stored exponents so we can use them to represent both sides of zero.

Example (single precision, 8-bit exponent):

- The bias = 127, The stored exponent E_{stored} goes from 0 → 255
- But after subtracting bias:

$$E_{\text{actual}} = E_{\text{stored}} - 127 \rightarrow \text{ gives range -127 to +128}$$

Why Bias = 127?
For an exponent field of 8 bits:

$$\text{Bias} = 2^{(\text{exponent}-1)} - 1 = 128 - 1 = 127$$

This centers the zero point in the middle of the possible range, so we can evenly represent small and large numbers.

Thus, **If exponent > bias** → tiny numbers, **If exponent < bias** → big numbers

1.1.5 Further Explanation - What the Fraction Does

The **fraction** is the part that provides the digits or detail of the number. It's what fills in between powers of two. It's called fraction because it represents the part after the binary point in the form of fraction bits. Each bit represents a fraction of a power of 2:

Bit position	Weight (in decimal)
1st bit	$\frac{1}{2} = 0.5$
2nd bit	$\frac{1}{4} = 0.25$
3rd bit	$\frac{1}{8} = 0.125$
4th bit	$\frac{1}{16} = 0.0625$
...	...

Table 4: Fraction Bits as Binary Decimals

So if your fraction bits are 010, that means:

$$1 + 0 \times \frac{1}{2} + 1 \times \frac{1}{4} + 0 \times \frac{1}{8} = 1.25$$

Example
Say the exponent part gives $2^3 = 8$. The fraction then determines the exact number within that range: So:

Fraction bits	Fraction value	Final number (1 + fraction $\times 2^3$)
000000...	0.0	$1.0 \times 8 = 8.0$
100000...	0.5	$1.5 \times 8 = 12.0$
111111...	1.0	$2.0 \times 8 = 16.0$

Table 5: Fraction Examples

- The **exponent** sets the "zoom level" (the range)
- The **fraction** fine-tunes where exactly the number lies inside that range,

1.2 Practice Q3: What is the difference between BF16 and FP16?

(a) FP16 (b) BF16

Figure 2: FP16 vs. BF16

1.2.1 Answer

1.2.2 Why is BF16 Favored in ML?

- **Overflow** occurs when the result of a calculation is **larger** than the largest representable number.
- **Underflow** occurs when the result of a calculation is **smaller** than the smallest representable number.

Feature	**FP16**	**BF16**
Exponent bits	5	8
Fraction bits	10	7
Range	Smaller	Larger
Precision	Higher	Lower
Overflow/Underflow	Easier	Harder
Common use	Graphics, inference	Machine learning training

Table 6: FP16 vs. BF 1

More exponent bits → larger range → less chance of overflow
More exponent bits → larger range → less chance of overflow

For ML training, numbers don't need to be super precise but must stay in a stable range to avoid overflow/underflow during training. Thus, BF16 is favored.

1.3 Other Floating-Point Number Knowledge

1.3.1 Floating-Point Number: Normal vs. Subnormal

Normal number is a floating-point number whose exponent field $\neq 0$ and $\neq$ all 1s (not special values like ∞ or NaN). It uses a hidden leading 1 in the significand, giving the formula:

$$(-1)^{\text{sign}} \times 2^{(\text{exponent - bias})} \times (1 + \text{fraction})$$

Subnormal number is a floating-point number whose exponent field $= 0$ but fraction $\neq 0$. It does **not** use the hidden 1, so the formula becomes:

$$(-1)^{\text{sign}} \times 2^{(1 - \text{bias})} \times \text{fraction}$$

Subnormal numbers extend the range below the smallest normal number, allowing representation of values closer to 0, thereby presenting sudden underflow and a smoother transition to 0.

Figure 3: Normal & Subnormal Values

1.3.2 Minimum Positive Value (Normal vs. Subnormal)

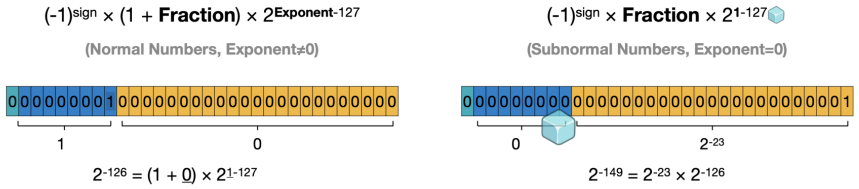


Figure 4: Minimum Positive Value

Category	Exponent Field	Fraction Field	Calculation
Normal Number	Exponent = 1	Fraction = 0	Value = $(1+0) \times 2^{1-127} = 2^{-126}$
Subnormal Number	Exponent = 0	Fraction = 2^{-23}	Value = $2^{-23} \times 2^{-126} = 2^{-149}$

Table 7: Minimum Positive Value

1.3.3 Other Special Cases

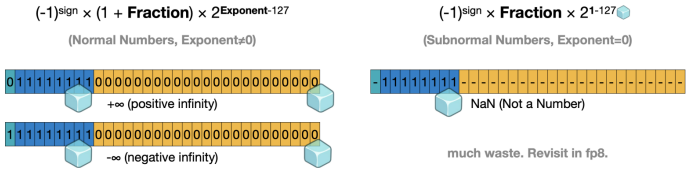


Figure 5: Other Special Values

1.4 Binary Representation - Floating Point Number Conversion

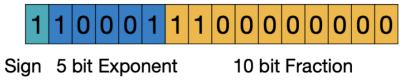


Figure 6: Binary Representation

1. **Determine Sign**: $1 \to (-)$, $0 \to (+)$
2. **Determine Bias**: Bias $= 2^4 - 1 = 15$
3. **Determine exponent bits value**: $(1)2^4 + (0)2^3 + (0)2^2 + (0)2^1 + (1)2^0 = 16 + 1 = 17$
4. **Determine actual exponent value**: $17 - 15 = 2$
5. **Determine fraction**: $(1)2^{-1} + (1)2^{-2} + (0)2^{-3} + \ldots = 2^{-1} + 2^{-2} = 0.5 + 0.25 = 0.75$
6. **Determine subnormal/normal**: all zero exponent $\to$ subnormal; otherwise $\to$ normal. This case is normal.
7. **Calculate decimal value**: normal $\to (-1)^1 \times (1 + 0.75) \times 2^2 = -7.0$

1.5 Floating Point Number - Binary Representation Conversion

What is Decimal 2.5 in BF16?

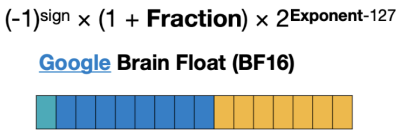


Figure 7: Float to Binary

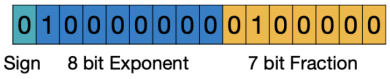


Figure 8: Result: 2.5 in BF16

1. **Write 2.5 in scientific binary form**: $2.5 = 2 + 0.5 = 10_2 + 0.1_2 = 10.1_2 = 1.01_2 \times 2^1 = 1.25 \times 2^1$
2. **Determine sign bit**: 2.5 is positive, so sign bit $= 0$
3. **Determine exponent bit**: Since $2.5 = 1.25 \times 2^1$, the true exponent $E = 1$. The biased exponent $= 127 + 1 = 128 = 10000000_2$
4. **Determine fraction bit**: Significand $= 1.25 = 1 + 0.25$, so fractional part $= 0.25 = 2^{-2} = 0.01_2$. With 7 bits in BF16, fraction bit $= 0100000_2$

2 Advancements in Floating Point Representation

2.1 Evolution from FP32 → FP16 → FP8 (E4M3) → FP8 (E5M2) → FP4

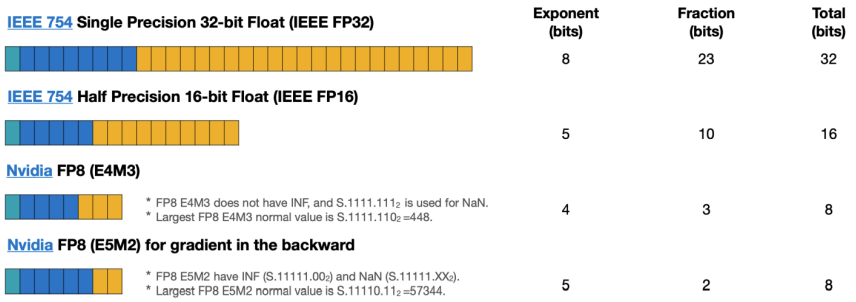


Figure 9: Advancements in Floating Point Representation

- Exponent width → range; Fraction width → precision.
  - Larger exponent → wider dynamic range (less overflow).
  - Larger fraction → higher precision (but more bits and cost).
- **Goal:** Reducing number of bits to increase computing power in the era of deep learning.
- FP8 were used to train DeepWeek and gpt-oss.
- FP8 (E4M3):
  - E4M3 refers to 4 exponent bits and 3 fraction bits.
  - It is one of the most commonly used FP8 format today. (Also representation for the final exam)

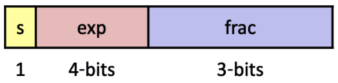


Figure 10: Representation of FP8 (E4M3)

2.2 Why BF16 is better in ML/AI? (BF16 vs FP16)

Property	BF16	FP16
Exponent bits	8	5
Fraction bits	7	10
Range	Wider	Narrower
Precision	Lower	Higher
Overflow risk	Lower	Higher

BF16 is favored in deep-learning hardware because it keeps FP32-level range but with half the bit width.

1. Precision is enough. ML/AI is error-tolerant.
   (a) Because small rounding or quantization errors do not significantly affect model convergence or accuracy.

2. Deep learning is easy to overflow, which we always want to avoid.
3. Closer range to FP32.

3 Writing & Reading Memory Instructions

3.1 Memory Operations Overview

Computer systems perform two fundamental operations when interacting with memory: writing data to memory and reading data from memory. These operations form the basis of all data movement between the CPU and memory hierarchy.

3.1.1 Write Operations (Store)

Write operations transfer data from CPU registers to memory locations. In assembly language, this is typically represented as: In assembly language, this is typically represented as:

```
movq %rax, %rsp
```

This instruction stores the value from register `%rax` into the memory location pointed to by `%rsp`. This is commonly referred to as a **store** operation.

3.1.2 Read Operations (Load)

Read operations transfer data from memory to CPU registers. In assembly language:

```
movq %rsp, %rax
```

This instruction loads the value from the memory location pointed to by `%rsp` into register `%rax`. This is commonly referred to as a **load** operation.

3.2 Computer Architecture Overview

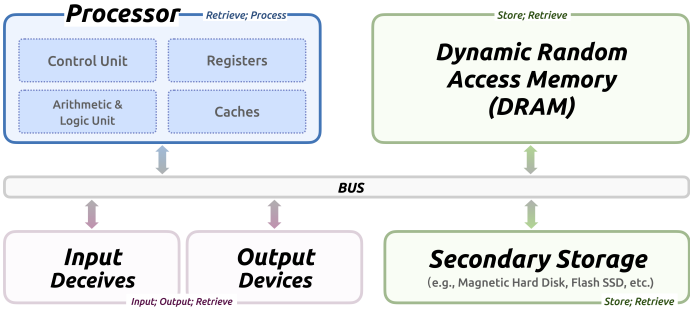


Figure 11: Abstract Computer Parts and Data Flow

The key components include:

- **Processor**: Contains the Control Unit (CU), Arithmetic & Logic Unit (ALU), registers, and caches. Responsible for retrieving and processing data.

- **Dynamic Random Access Memory (DRAM)**: Main memory that stores and retrieves data for active programs.
- **Secondary Storage**: Includes HDDs and SSDs for persistent data storage.
- **Bus**: Communication pathway connecting all components.
- **Input/Output Devices**: Interface for user interaction and data display.

3.3 Bus Structure and Memory Communication

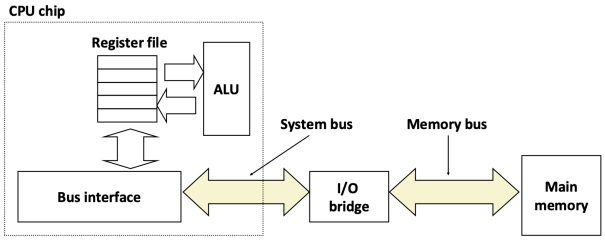


Figure 12: Bus Structure Connecting CPU and Memory

The bus is a critical component consisting of parallel wires that carry address, data, and control signals between the CPU and memory. As shown in the architecture:

- The **system bus** connects the CPU to the I/O bridge
- The **memory bus** connects the I/O bridge to main memory
- The **bus interface** within the CPU manages communication with the external bus

Key characteristics of the bus system:

- Buses are typically shared by multiple devices
- They carry three types of information: 1) **Address Signals** (specifying memory locations), 2) **Data Signals** (actual information being transferred), and 3) **Control Signals** (coordinating the transfer)

3.4 Memory Read Transaction

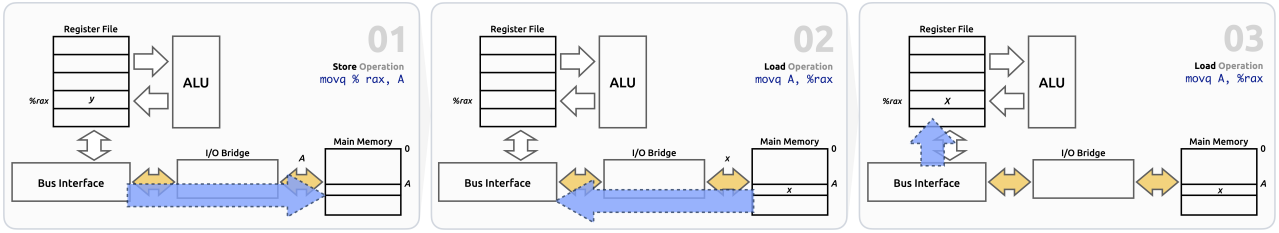


Figure 13: Memory Read Transaction: Three Phases

A memory read transaction (load operation) occurs in three distinct phases:

Phase 1: Address Placement. The CPU places the target address A on the memory bus. The bus interface sends this address through the system bus to the I/O bridge, which then forwards it via the memory bus to main memory. At this point: the address A is broadcast on the bus; main memory prepares to retrieve the data at address A; and the target register (e.g., `%rax`) awaits the incoming data.

Phase 2: Data Retrieval. Main memory reads address A from the memory bus, retrieves the word x stored at that location, and places it on the bus. The memory controller accesses the specified address, data word x is retrieved from the memory cells, and the data is placed on the data lines of the bus.

Phase 3: Data Reception. The CPU reads word x from the bus and copies it into the specified register. The bus interface receives the data from the bus, data is routed to the appropriate register file, and the value x is stored in register `%rax`.

`Example:` For the instruction `movq A, %rax`, the three phases execute sequentially to transfer data from memory location A to register `%rax`.

3.5 Memory Write Transaction

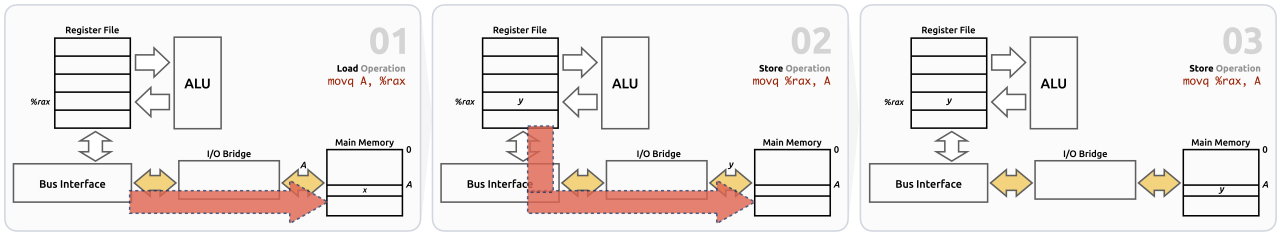


Figure 14: Memory Write Transaction: Three Phases

A memory write transaction (store operation) also occurs in three phases:

Phase 1: Address and Preparation. The CPU places address A on the bus. Main memory reads this address and prepares to receive the corresponding data. Address A is broadcast on the address lines, the memory controller identifies the target location, and main memory enters a wait state for incoming data.

Phase 2: Data Transmission. The CPU places the data word y (from the source register) on the bus. The value y from register `%rax` is sent to the bus interface, data is placed on the data lines of the bus, and the data travels through the bus system to main memory.

Phase 3: Data Storage. Main memory reads data word y from the bus and stores it at address A. The memory controller receives the data from the bus, the value y is written to the memory cells at address A, and the write operation is confirmed complete.

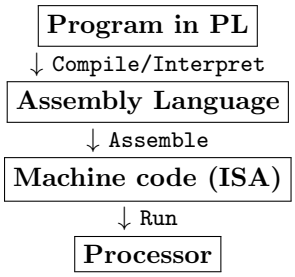
`Example:` For the instruction `movq %rax, A`, the value y stored in register `%rax` is written to memory location A through these three phases.

4 Processors

4.1 Basics of Processors - Concepts

- **Processor**: Hardware to orchestrate and execute *instructions* to *manipulate data* as specified by a program. `Example:` CPU, GPU, FPGA, TPU, embedded, etc.
- **ISA (Instruction Set Architecture)**: The vocabulary of commands of a processor.

4.1.1 How does a program execute on a chip?



4.1.2 How does a processor execute machine code?

The most common approach is load-store architecture. **Registers** are tiny local memory ("scratch space") on the processor where instructions and data are copied. The ISA specifies the bit length/format of machine code commands and provides several commands to manipulate register contents.

4.2 Instruction

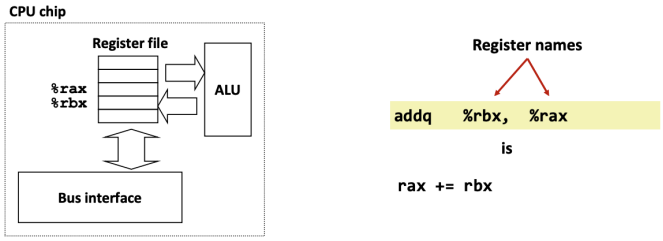


Figure 15: A CPU Chip Processor and Registers

- ALU: silicons, perform compute
- Bus interface: communication bus that share across memory, across processor, and all kinds of hardware

In the figure above on the left we have a CPU chip, notably it has an arithmetic logic unit (ALU) which is where the instructions are executed. In the figure above on the right, we have an example with two registers named rax and rbx. The instruction is to add the value of rbx to rax, which is an example of how processors work at a very low level.

4.3 Speed of Processor (CPU and GPU)

The measure of a processor is typically done in **instructions per second**, which is the number instructions the processor can complete within a single second. In data science, we care more about the computation on **floating point numbers**, known as **FLOPs**.

- **FLOPs** are the number of floating point operations a processor can do, and are an important metric.
- Develop a sense: Intel CPU → 100-200 GFLOPS (billion floating-point operations per second); FP8 Tensor Core → 4k teraFLOPS

4.4 The Problem?

There is a mismatch between processor compute speed and memory read speed. Consider a 100 GFLOPS/s CPU with an 80-160 MB/s HDD. If we use 0.5s to perform 50 GFLOPS, we need to read 50×2=100GB in the remaining 0.5s to keep the CPU busy, requiring 200 GB/s read speed—impossible for HDD.

Solutions: (1) Build faster storage, or (2) Use memory/storage hierarchy.

4.5 Memory/Storage Hierarchy

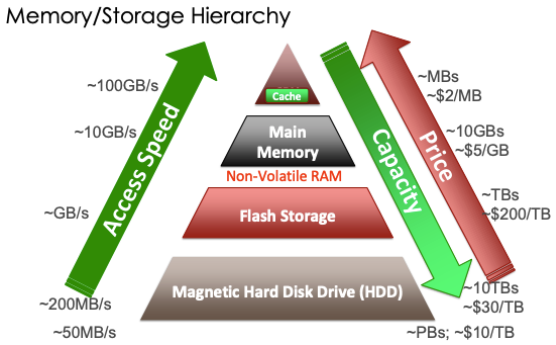


Figure 16: Memory/storage hierarchy: higher = faster and more expensive

- **Cache**: Fastest, smallest. Buffers frequently accessed instructions/data. CPU has three layers (L1, L2, L3); GPU typically has two (SRAM, HBM).
- **RAM/Main Memory**: Moderate speed and cost, larger capacity.
- **Flash Storage (SSD)**: Non-volatile, faster than HDD.
- **Hard Disk Drive (HDD)**: Cheapest, slowest, largest capacity.

4.6 How does a processor execute machine code?

ISA commands manipulate register contents in three ways:

- **Memory access**: Load/store operations between DRAM and registers
- **Arithmetic & logic**: Add, multiply, bitwise ops, comparisons (ALU)
- **Control flow**: Branch, call, jump instructions (Control Unit)

GPU vs CPU: CPUs have more control units for complex branching; GPUs minimize CUs and maximize ALUs for parallel computation.

4.7 GPT Example

Figure 17 shows the simplified GPT workflow:
1. GPT code and weights are fetched from disk to main memory (DRAM) via the bus.
2. Code is interpreted as instructions; weights/data are loaded into DRAM.
3. User prompt arrives via network and is fetched into main memory.
4. Code, weights, and prompt move through caches to registers, where instructions are paired with data and executed on the ALU.
5. Output is sent to memory, then returned to the user's monitor via the bus.

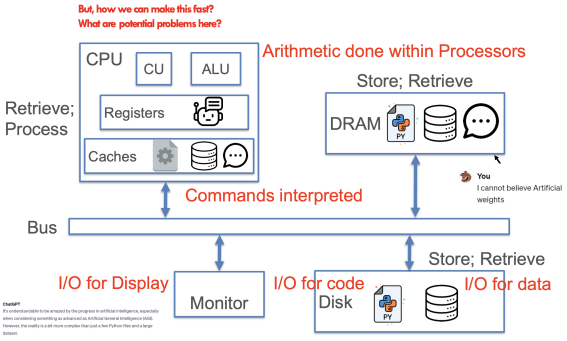


Figure 17: GPT Example

4.8 The CPU-memory gap

Primary problem is that the gap widens between DRAM, disk and CPU speeds with the technology grows (Figure 18). The key to bridge the gap is locality.

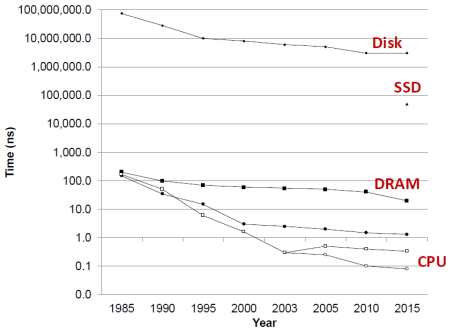


Figure 18: Gap

4.9 Locality

A concrete example of locality: copyij vs copyji

```
void copyij(long int src[2048][2048],
            long int dst[2048][2048])
{
    long int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

```
void copyji(long int src[2048][2048],
            long int dst[2048][2048])
{
    long int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

The first copyij program finishes in 4.3 ms while copyji takes 81.8 ms (20x slower). Memory stores data in row-major order. The first program reads rows sequentially while the second reads by column, which is non-sequential.

Principle of Locality: Many programs tend to use data and instructions with addresses near or equal to those they have used recently.
Temporal locality: Recently referenced items are likely to be referenced again in the near future.
Spatial locality: Items with nearby addresses tend to be referenced close together in time.

4.9.1 Locality Example

```
num_list = [1, 2, 3, 4, 5, 7]
sum = 0;
for (x in num_list)
    sum += x;
return sum;
```

Data references: Array elements accessed in succession (stride-1 reference pattern); variable sum referenced each iteration.
Instruction references: Instructions executed in sequence; loop cycles repeatedly. This demonstrates spatial locality.

4.9.2 Qualitative Estimates of Locality

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Question: Does this function have good locality with respect to array a? **Answer:** Yes, since we sum numbers in sequential order, which matches the storage order in memory.