https://hao-ai-lab.github.io/dsc204a-f25/

DSC 204A: Scalable Data Systems
Fall 2025

Staff

Instructor: Hao Zhang

TAs: Mingjia Huo, Yuxuan Zhang

@haozhangml @haoailab

haozhang@ucsd.edu

Where We Are

Machine Learning Systems

Big Data

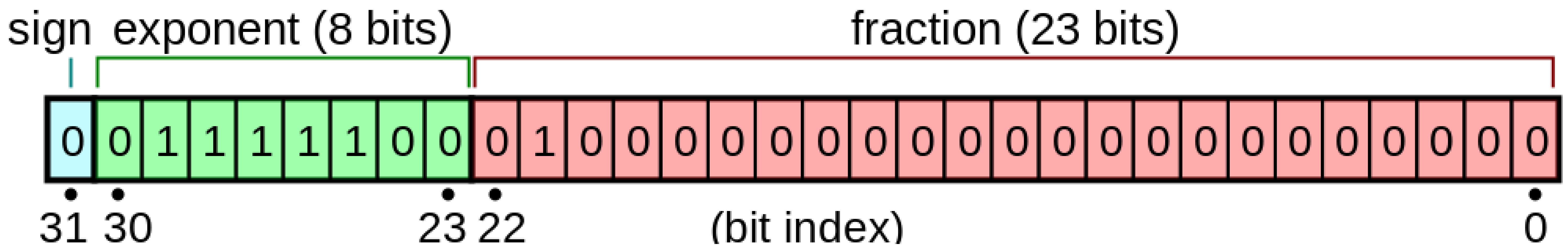
Cloud

Foundations of Data Systems

1980 - 2000

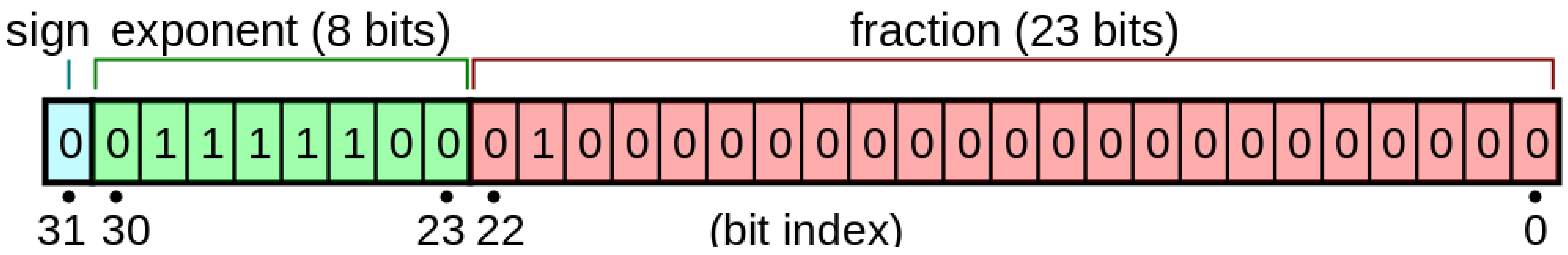
Q2: What does **exponent** and **fraction** control?

- Exponent controls: range, offset
- Fraction controls: actual value, precision



Digital Representation of Data: Bias

- Float:
  - Standard IEEE format for single (aka binary32):

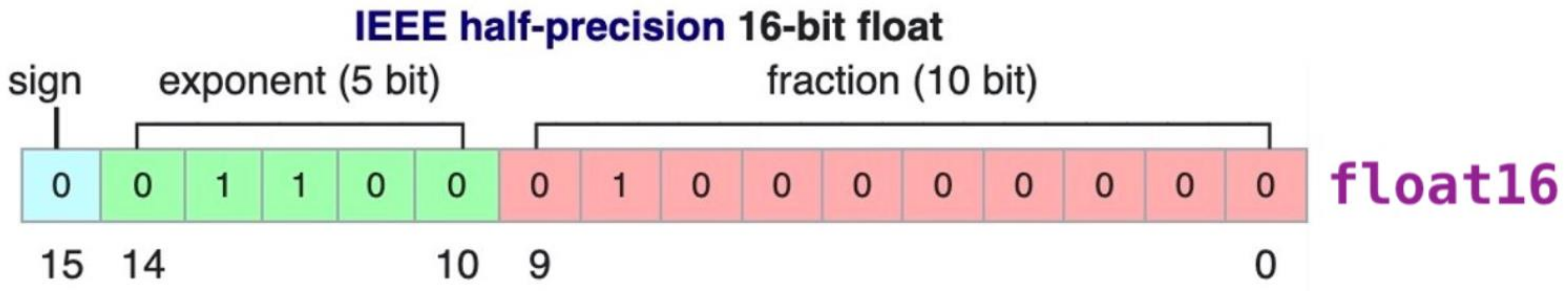


Why?

$$(-1)^{sign} \times 2^{\boxed{exponent-127}} \times (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i})$$

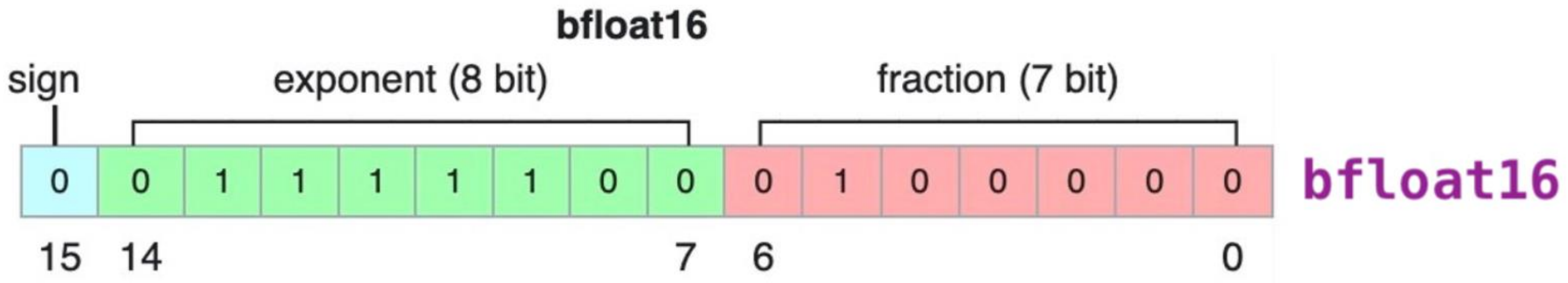
$$(-1)^0 \times 2^{124-127} \times (1 + 1 \cdot 2^{-2}) = (1/8) \times (1 + (1/4)) = 0.15625$$

Q3: What is the difference between BF16 and FP16?



Less exponent -> smaller range -> easier to overflow

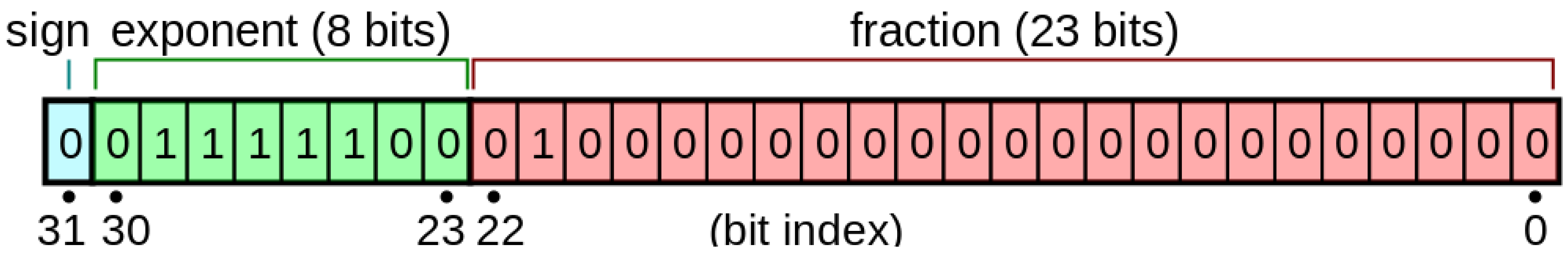
More fraction -> more precise



more exponent -> larger range -> harder to overflow

less fraction -> less precise

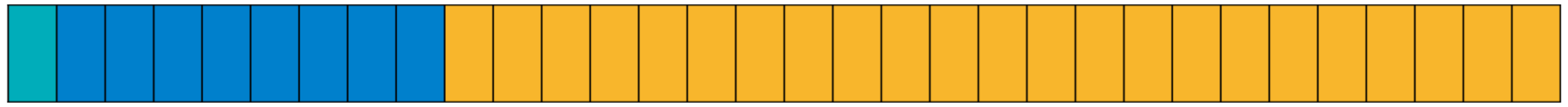
Floating-point Representation



$$(-1)^{sign} \times 2^{exponent-127} \times (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i})$$

Q: How to represent 0?

Floating-point Number: normal vs. subnormal



Sign **8 bit Exponent** **23 bit Fraction**

$(-1)^{\text{sign}} \times (1 + \textbf{Fraction}) \times 2^{\textbf{Exponent}-127}$

(Normal Numbers, Exponent≠0)

Should have been $(-1)^{\text{sign}} \times (1 + \textbf{Fraction}) \times 2^{0-127}$

But we force to be $(-1)^{\text{sign}} \times \textbf{Fraction} \times 2^{1-127}$

(Subnormal Numbers, Exponent=0)

0 | 0 1 1 1 1 1 0 1 | 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

$\underbrace{01111101}_{125}$ $\underbrace{00010000000000000000000}_{0.0625}$

$0.265625 = 1.0625 \times 2^{-2} = (1 + \underline{0.0625}) \times 2^{\underline{125}-127}$

0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

$\underbrace{00000000}_{0}$ $\underbrace{00000000000000000000000}_{0}$

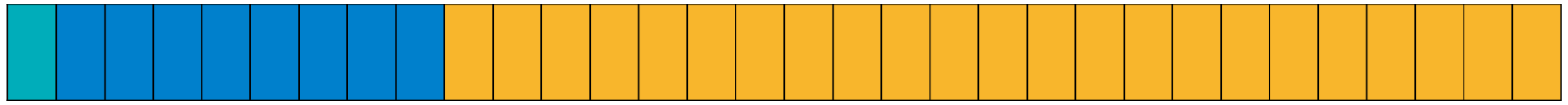
1 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

$\underbrace{00000000}_{0}$ $\underbrace{00000000000000000000000}_{0}$

$0 = 0 \times 2^{-126}$

Q: What is the minimum positive value?

What is the minimum positive value?



Sign **8 bit Exponent** **23 bit Fraction**

$(-1)^{sign} \times (1 + \textbf{Fraction}) \times 2^{\textbf{Exponent}-127}$

(Normal Numbers, Exponent≠0)

0 | 00000001 | 00000000000000000000000

$\underbrace{00000001}_{1}$ $\underbrace{00000000000000000000000}_{0}$

$2^{-126} = (1 + \underline{0}) \times 2^{\underline{1}-127}$

$(-1)^{sign} \times \textbf{Fraction} \times 2^{1-127}$

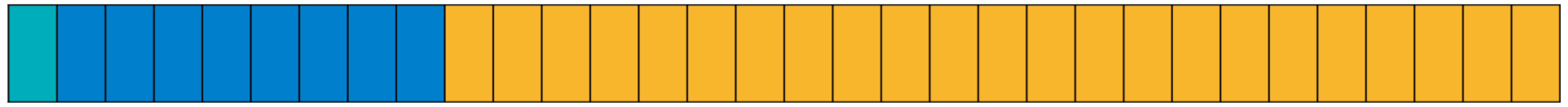
(Subnormal Numbers, Exponent=0)

0 | 00000000 | 00000000000000000000001

$\underbrace{00000000}_{0}$ $\underbrace{00000000000000000000001}_{2^{-23}}$

$2^{-149} = 2^{-23} \times 2^{-126}$

Some Special Values

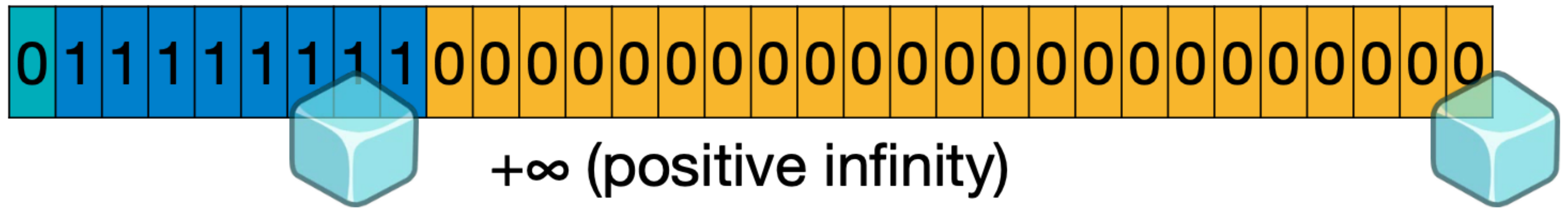


$(-1)^{\text{sign}} \times (1 + \textbf{Fraction}) \times 2^{\textbf{Exponent}-127}$

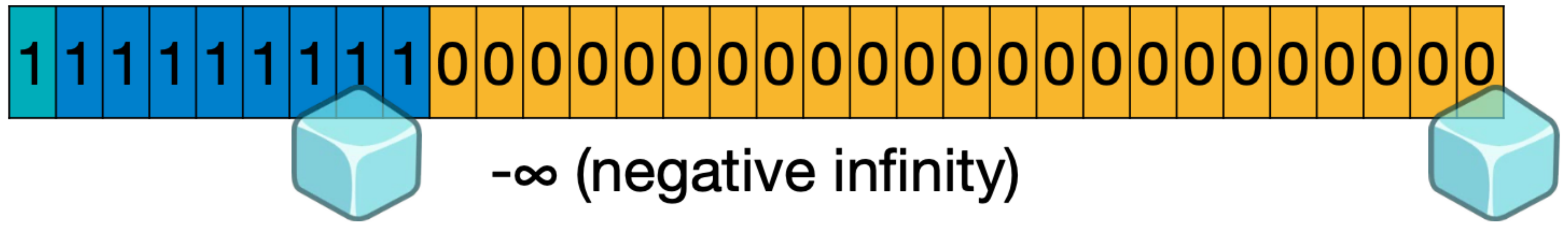
(Normal Numbers, Exponent≠0)

$(-1)^{\text{sign}} \times \textbf{Fraction} \times 2^{1-127}$

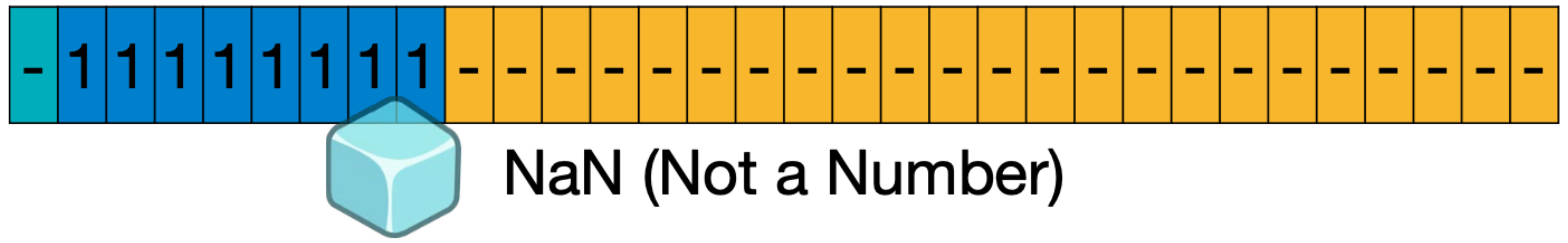
(Subnormal Numbers, Exponent=0)



+∞ (positive infinity)



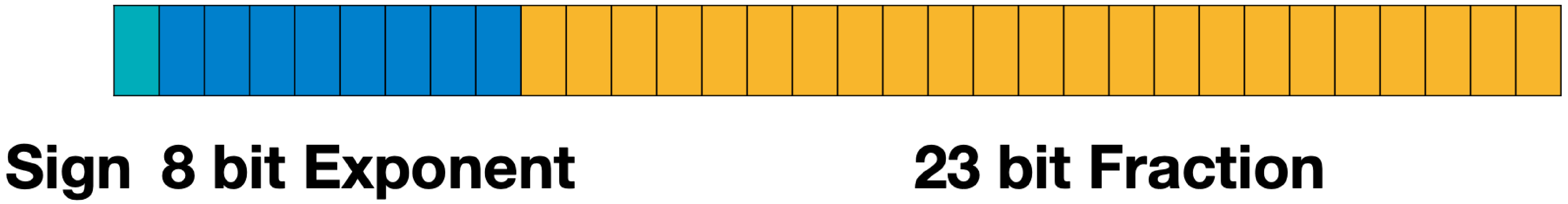
-∞ (negative infinity)



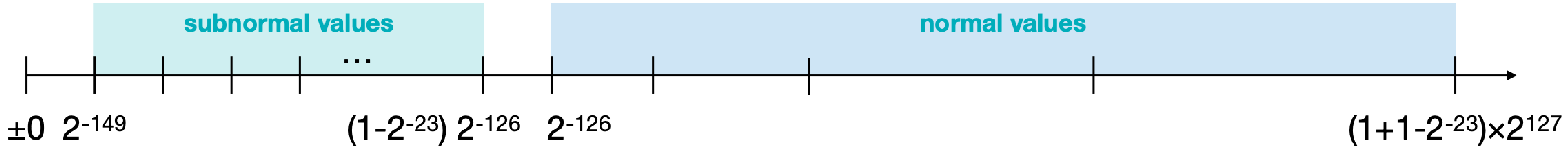
NaN (Not a Number)

much waste. Revisit in fp8.

Summary of fp32



Exponent	Fraction=0	Fraction≠0	Equation
$00_H = 0$	±0	subnormal	$(-1)^{sign} \times \textbf{Fraction} \times 2^{1-127}$
$01_H \ldots FE_H = 1 \ldots 254$	normal		$(-1)^{sign} \times (1 + \textbf{Fraction}) \times 2^{\textbf{Exponent}-127}$
$FF_H = 255$	±INF	NaN	



Exercise

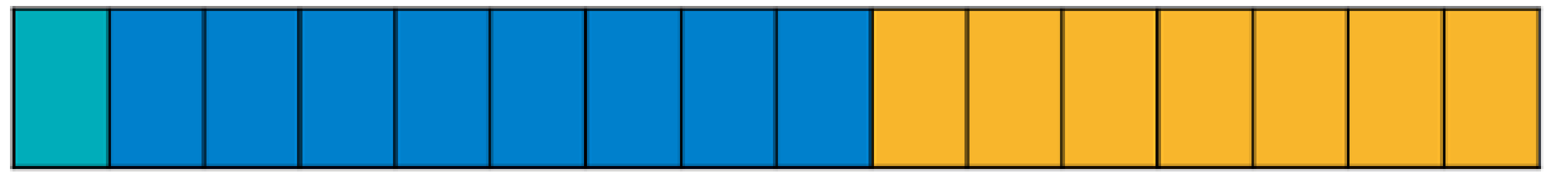
1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0

Sign 5 bit Exponent 10 bit Fraction

- Sign: -
- Exponent
  - Bias:$2^4 - 1 = 15_{10}$
  - $10001_2 - 15_{10} = 17_{10} - 15_{10} = 2_{10}$
- Fraction
  - $1100000000_2 = 0.75_{10}$
- Answer: $-(1 + 0.75) \times 2^2 = -7.10_{10}$

$(-1)^{\text{sign}} \times (1 + \textbf{Fraction}) \times 2^{\textbf{Exponent}\text{-}127}$

Google Brain Float (BF16)



Exercise

What is Decimal 2.5 in BF16?

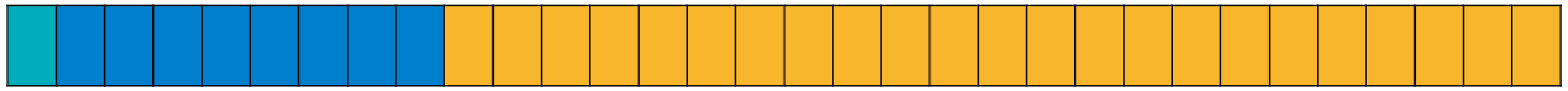
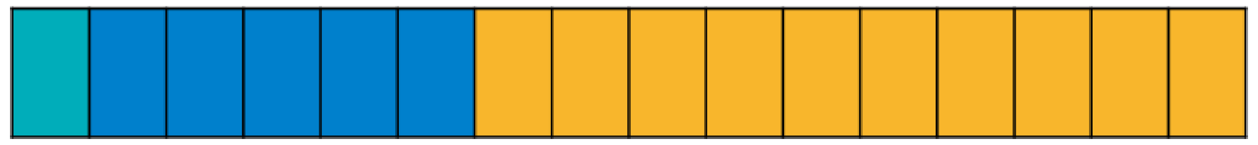
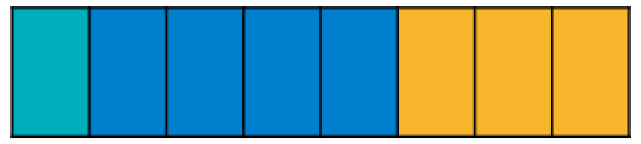
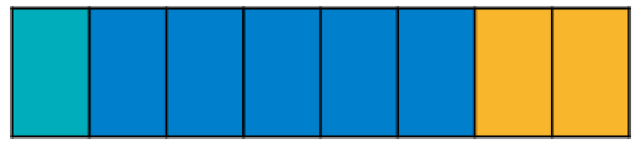
- $2.5 = 1.25 \times 2^1$
- Sign: +
- Exponent: bias is $2^7 - 1 = 127$
  - $x - 127 = 1; x = 128_{10} = 10000000_2$
- Fraction: 7-bit fraction
  - $0.25 = 0100000_2$

0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Sign | 8 bit Exponent | 7 bit Fraction

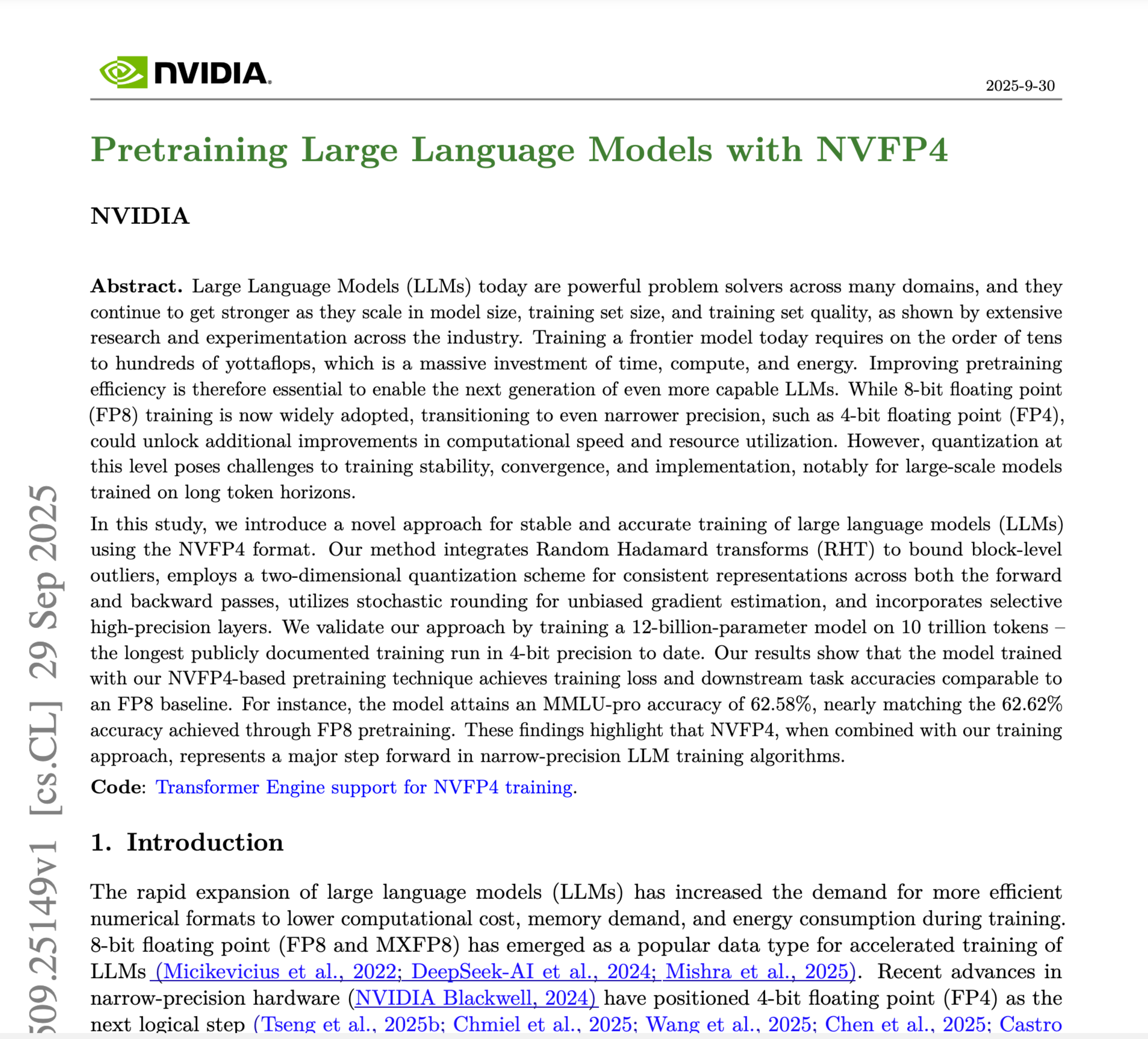
Latest FP8

- Exponent width -> Range; Fraction width -> precision

	Exponent (bits)	Fraction (bits)	Total (bits)
IEEE 754 **Single Precision 32-bit Float (IEEE FP32)** 	8	23	32
IEEE 754 **Half Precision 16-bit Float (IEEE FP16)** 	5	10	16
Nvidia **FP8 (E4M3)**  * FP8 E4M3 does not have INF, and $S.1111.111_2$ is used for NaN. * Largest FP8 E4M3 normal value is $S.1111.110_2 = 448$.	4	3	8
Nvidia **FP8 (E5M2) for gradient in the backward**  * FP8 E5M2 have INF ($S.11111.00_2$) and NaN ($S.11111.XX_2$). * Largest FP8 E5M2 normal value is $S.11110.11_2 = 57344$.	5	2	8

FP4

Introducing NVFP4 for Efficient and Accurate Low-Precision Inference



NVIDIA 2025-9-30

Pretraining Large Language Models with NVFP4

NVIDIA

Abstract. Large Language Models (LLMs) today are powerful problem solvers across many domains, and they continue to get stronger as they scale in model size, training set size, and training set quality, as shown by extensive research and experimentation across the industry. Training a frontier model today requires on the order of tens to hundreds of yottaflops, which is a massive investment of time, compute, and energy. Improving pretraining efficiency is therefore essential to enable the next generation of even more capable LLMs. While 8-bit floating point (FP8) training is now widely adopted, transitioning to even narrower precision, such as 4-bit floating point (FP4), could unlock additional improvements in computational speed and resource utilization. However, quantization at this level poses challenges to training stability, convergence, and implementation, notably for large-scale models trained on long token horizons.

In this study, we introduce a novel approach for stable and accurate training of large language models (LLMs) using the NVFP4 format. Our method integrates Random Hadamard transforms (RHT) to bound block-level outliers, employs a two-dimensional quantization scheme for consistent representations across both the forward and backward passes, utilizes stochastic rounding for unbiased gradient estimation, and incorporates selective high-precision layers. We validate our approach by training a 12-billion-parameter model on 10 trillion tokens – the longest publicly documented training run in 4-bit precision to date. Our results show that the model trained with our NVFP4-based pretraining technique achieves training loss and downstream task accuracies comparable to an FP8 baseline. For instance, the model attains an MMLU-pro accuracy of 62.58%, nearly matching the 62.62% accuracy achieved through FP8 pretraining. These findings highlight that NVFP4, when combined with our training approach, represents a major step forward in narrow-precision LLM training algorithms.

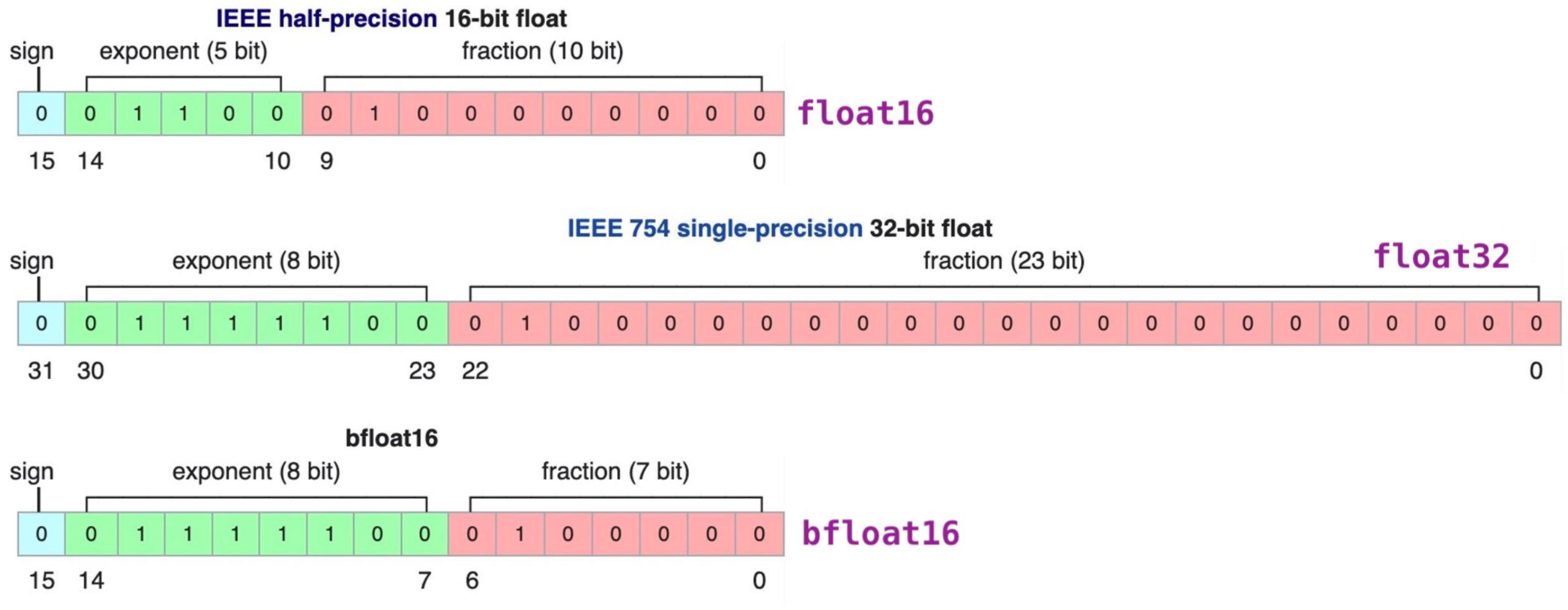
Code: Transformer Engine support for NVFP4 training.

1. Introduction

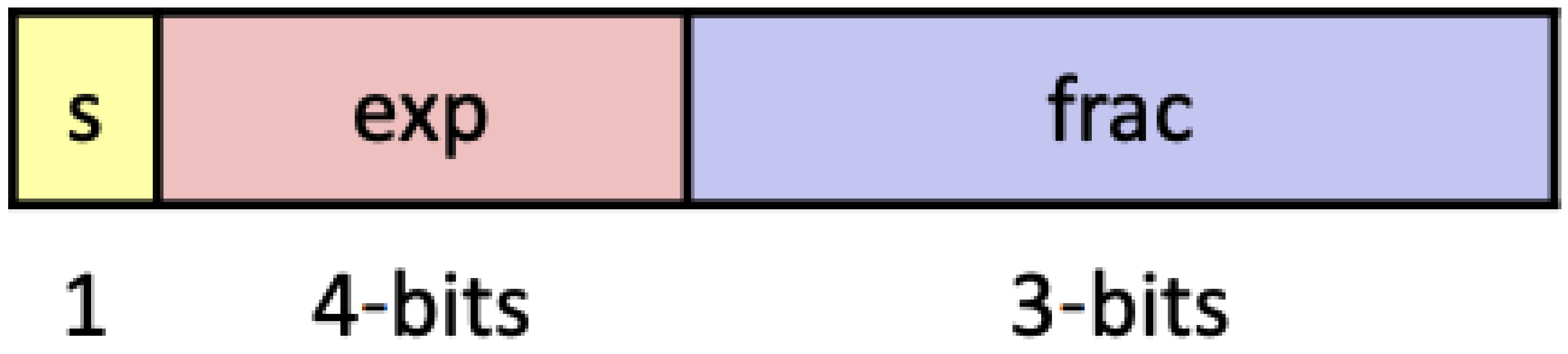
The rapid expansion of large language models (LLMs) has increased the demand for more efficient numerical formats to lower computational cost, memory demand, and energy consumption during training. 8-bit floating point (FP8 and MXFP8) has emerged as a popular data type for accelerated training of LLMs (Micikevicius et al., 2022; DeepSeek-AI et al., 2024; Mishra et al., 2025). Recent advances in narrow-precision hardware (NVIDIA Blackwell, 2024) have positioned 4-bit floating point (FP4) as the next logical step (Tseng et al., 2025b; Chmiel et al., 2025; Wang et al., 2025; Chen et al., 2025; Castro

Why BF16 is better in ML/AI?

1. Precision is enough. ML/AI is error-tolerant (why?)
2. Deep learning is easy to overflow
3. Closer range to fp32



Examples in the final exam: FP8



Demystify ChatGPT

You
I cannot believe Artificial general intelligence is just a few Python files and 350GB of weights

str

[0, 500, 32768, 1008, 922, ...]

List[integers]

GPT =

Disk

PY

A few KBs

+

Parameters:
350 GB

[0, 25116, 1234, 5984, 6, ...]

List[integers]

ChatGPT
It's understandable to be amazed by the progress in artificial intelligence, especially when considering something as advanced as Artificial General Intelligence (AGI). However, the reality is a bit more complex than just a few Python files and a large dataset.

str

Foundation of Data Systems

- Computer Organization
  - Representation of data
  - **processors**, **memory**, **storage**
- OS basics
  - Process, scheduling
  - Memory

Basics of Processors

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

Program in PL

↓ *Compile/Interpret*

Program in Assembly Language

↓ *Assemble*

Machine code tied to ISA

↓

Run on processor

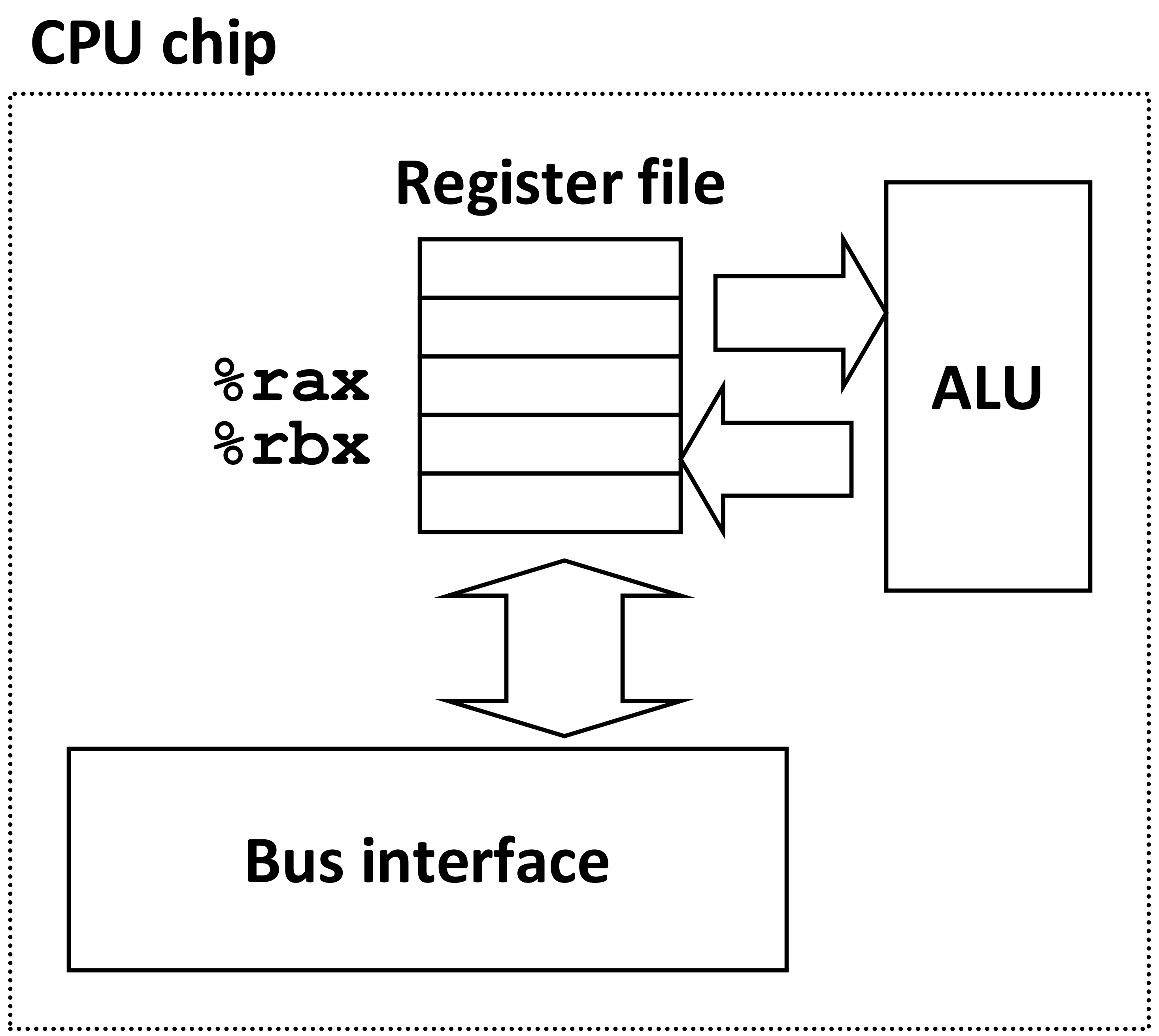
```
80483b4:    55                      push   %ebp
80483b5:    89 e5                   mov    %esp,%ebp
80483b7:    83 e4 f0                and    $0xfffffff0,%esp
80483ba:    83 ec 20                sub    $0x20,%esp
80483bd:    c7 44 24 1c 00 00 00    movl   $0x0,0x1c(%esp)
80483c4:    00
80483c5:    eb 11                   jmp    80483d8 <main+0x24>
80483c7:    c7 04 24 b0 84 04 08    movl   $0x80484b0,(%esp)
80483ce:    e8 1d ff ff ff          call   80482f0 <puts@plt>
80483d3:    83 44 24 1c 01          addl   $0x1,0x1c(%esp)
80483d8:    83 7c 24 1c 09          cmpl   $0x9,0x1c(%esp)
80483dd:    7e e8                   jle    80483c7 <main+0x13>
80483df:    b8 00 00 00 00          mov    $0x0,%eax
80483e4:    c9                      leave
80483e5:    c3                      ret
80483e6:    90                      nop
80483e7:    90                      nop
80483e8:    90                      nop
80483e9:    90                      nop
80483ea:    90                      nop
```

Basics of Processors

Q: *How does a processor execute machine code?*

- Most common approach: load-store architecture
- Registers: Tiny local memory ("scratch space") on proc. into which instructions and data are copied
- ISA specifies bit length/format of machine code commands
- ISA has several commands to manipulate register contents

Instruction



Register names

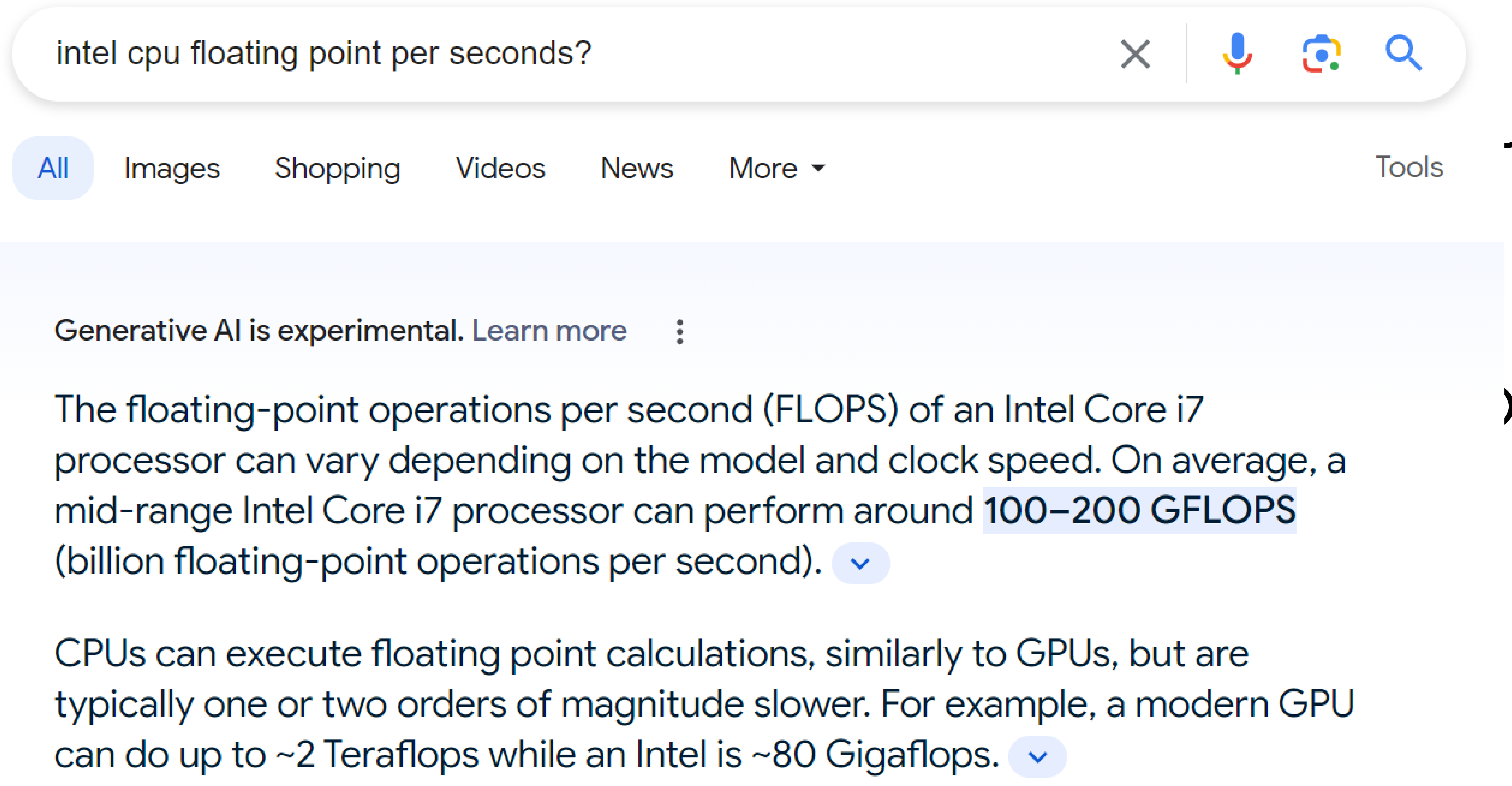
```
addq    %rbx,  %rax
```

is

```
rax += rbx
```

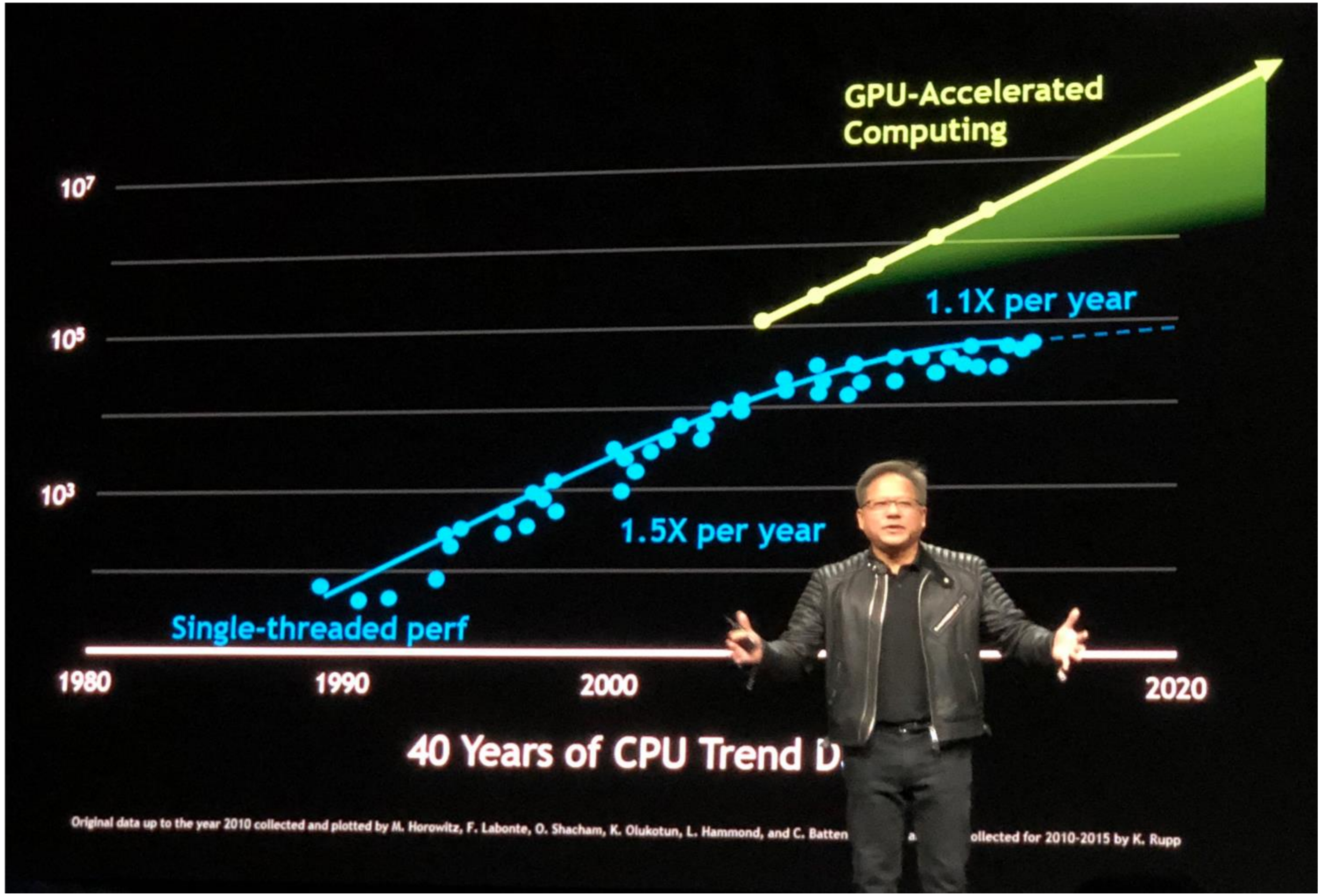
How Fast is Processor (CPU and GPU)

- Instruction / second: number of instru



Form Factor	H100 SXM
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core	989 teraFLOPS[2]
BFLOAT16 Tensor Core	1,979 teraFLOPS[2]
FP16 Tensor Core	1,979 teraFLOPS[2]
FP8 Tensor Core	3,958 teraFLOPS[2]

Moore's Law


GPU-Accelerated
Computing
10^7
10^5
10^3
1.1X per year
1.5X per year
Single-threaded perf
1980
1990
2000
2020
40 Years of CPU Trend D
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
ollected for 2010-2015 by K. Rupp

The Real Problem?

100 GFLOPs/s

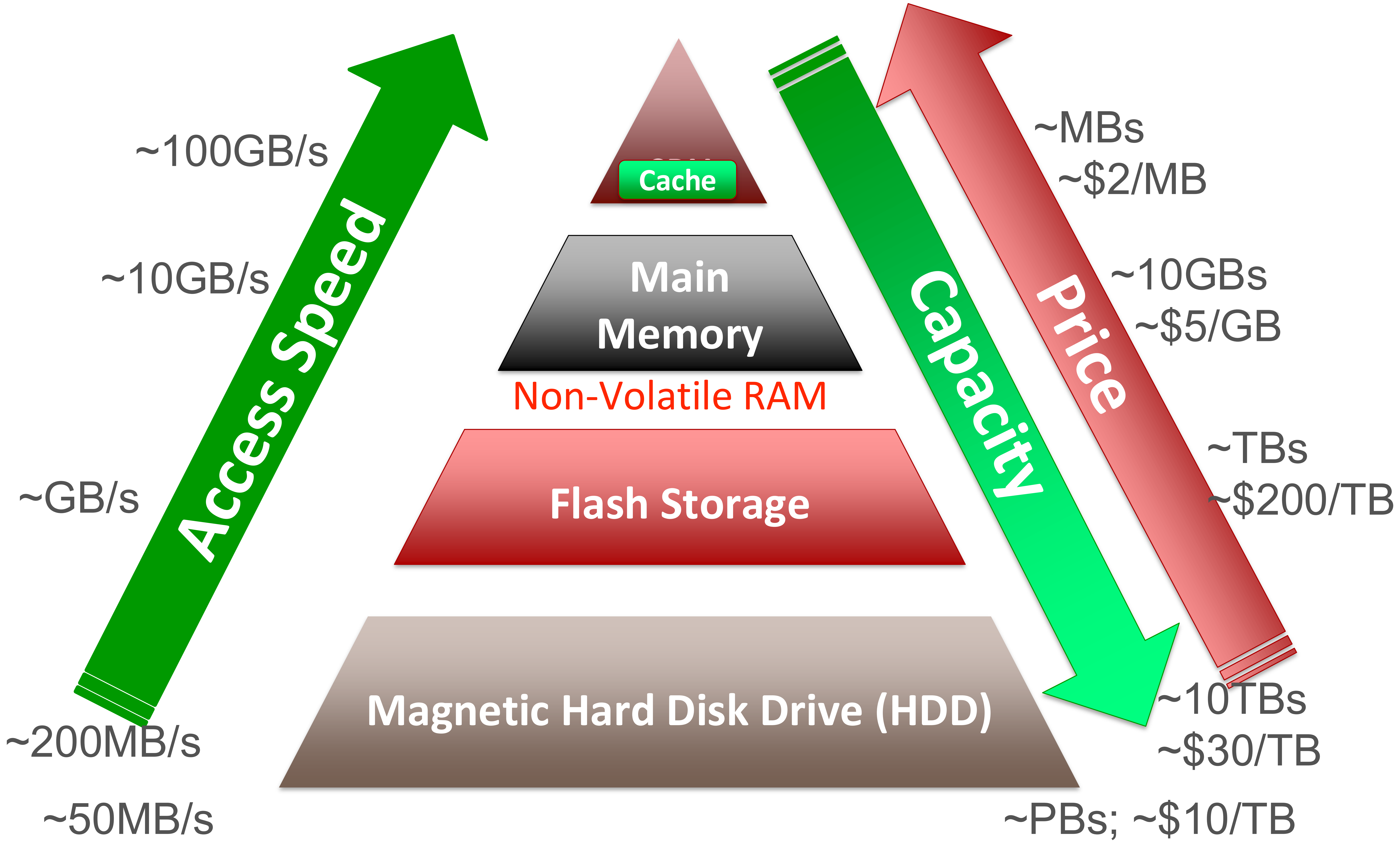
CPU

1. Assume we use 0.5s to perform 50 FLOPs
2. We need to read 50x2=100 GB in the rest of 0.5s to keep the CPU busy
3. We need the CPU to read at a speed of 100GB / 0.5s = 200 GB/s

Magnetic Hard Disk Drive (HDD)

80 – 160 MB/s

Memory/Storage Hierarchy

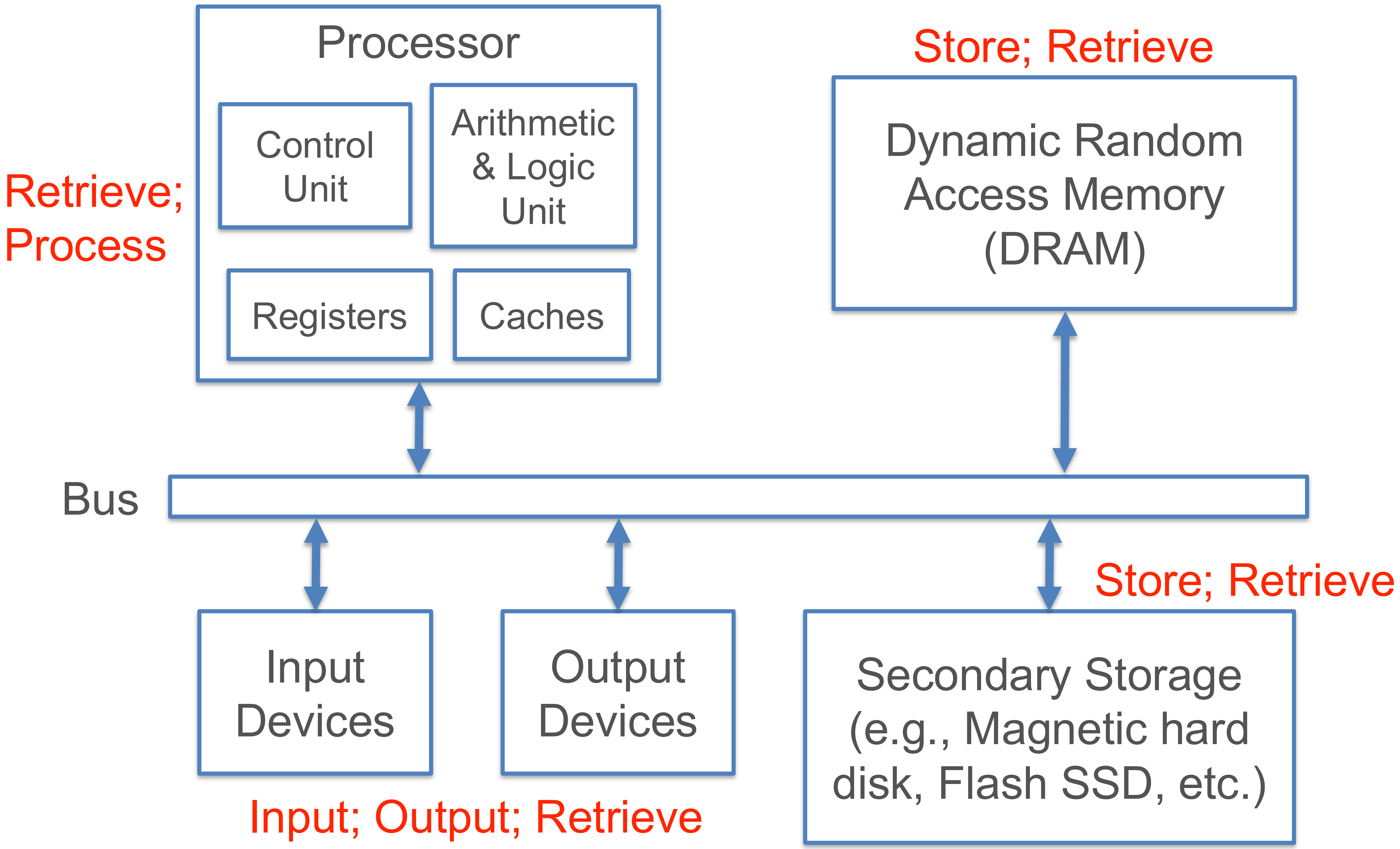


Writing & Reading Memory Instructions

- Write
  - Transfer data from memory to CPU
    movq %rax, %rsp
  - “Store” operation

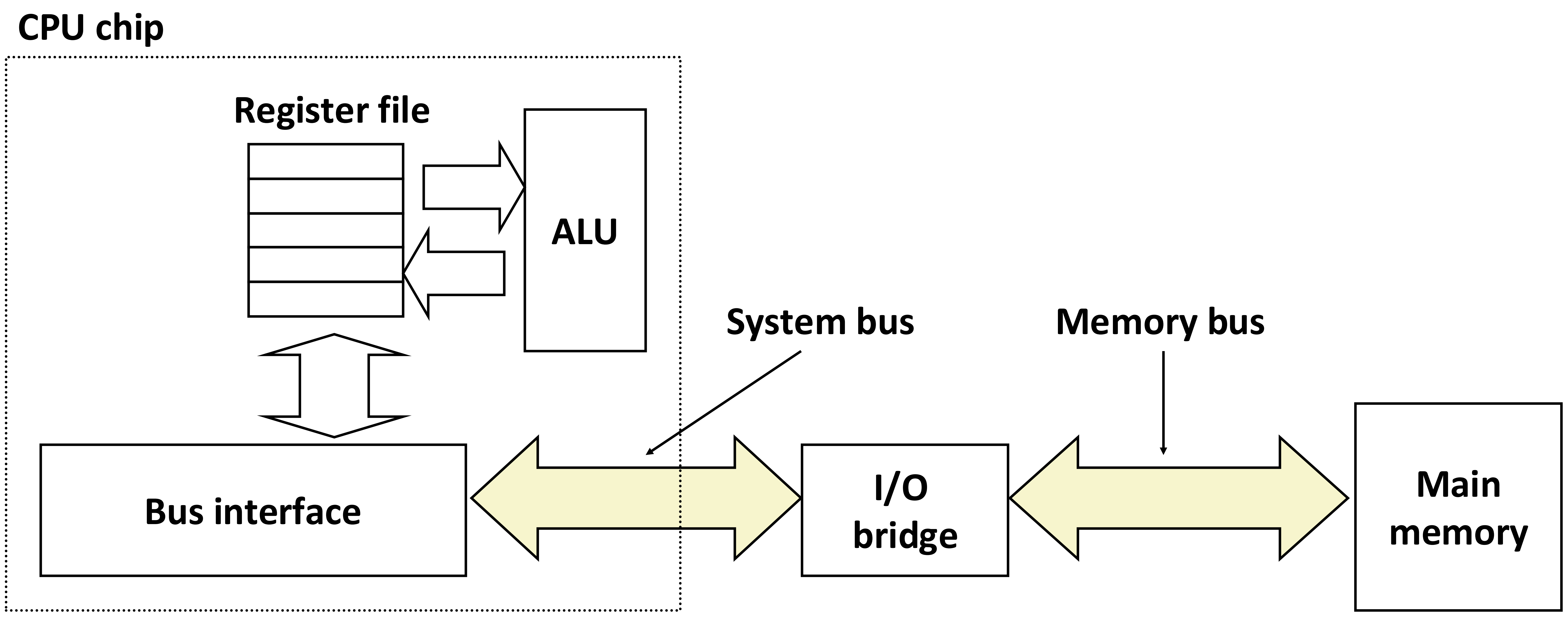
- Read
  - Transfer data from CPU to memory
    movq %rsp, %rax
  - “Load” operation

Abstract Computer Parts and Data

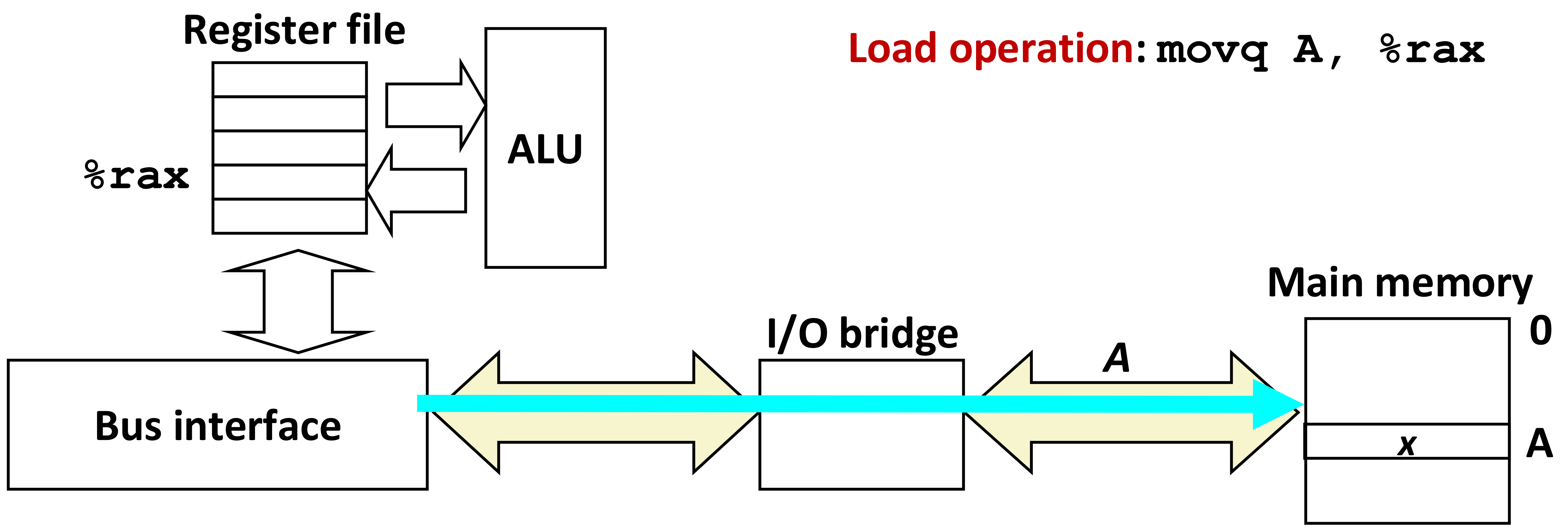


Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.

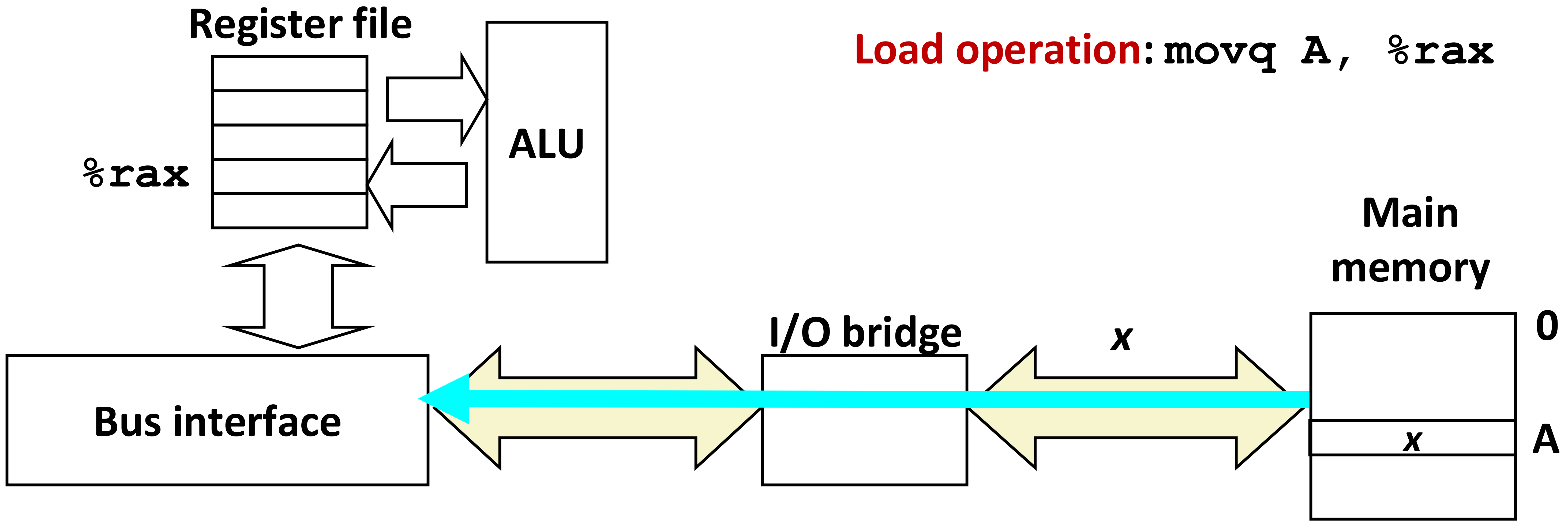


Memory Read Transaction (1)



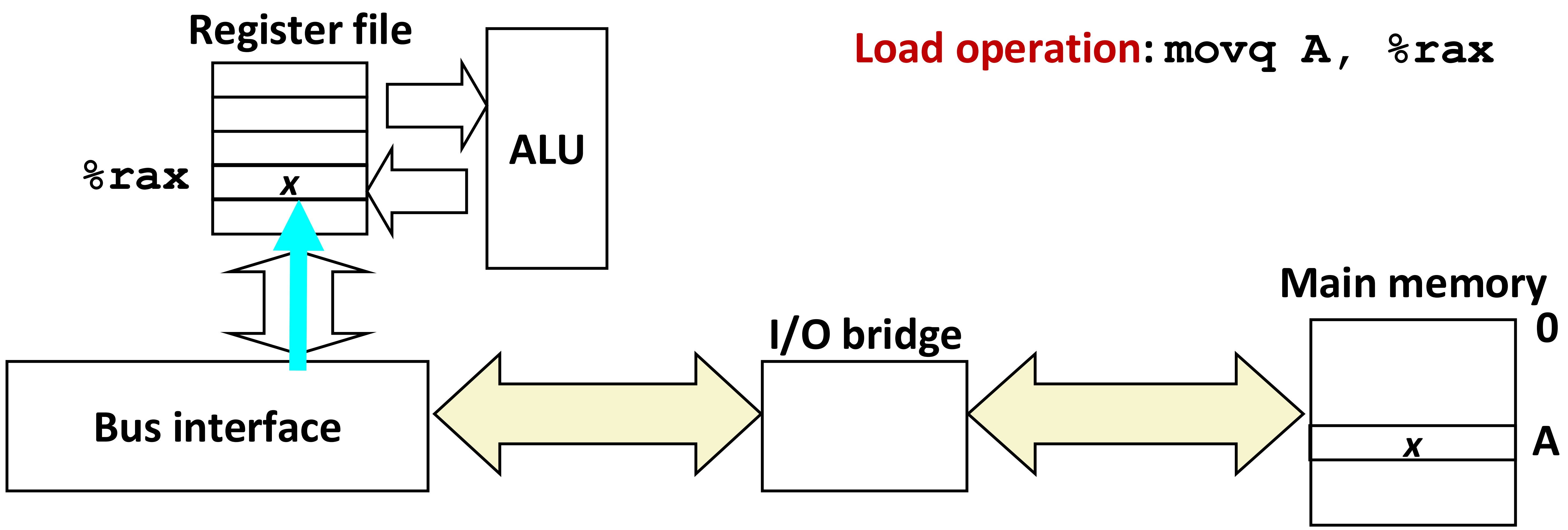
- CPU places address A on the memory bus.

Memory Read Transaction (2)



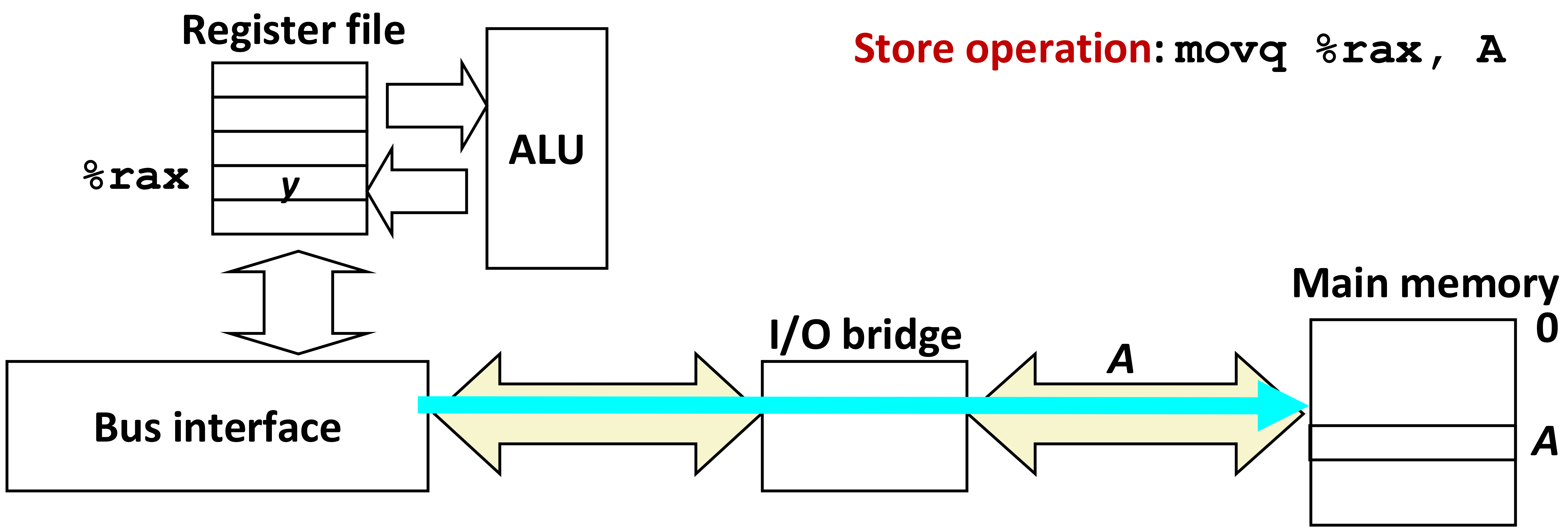
- Main memory reads A from the memory bus, retrieves word x, and places it on the bus.

Memory Read Transaction (3)



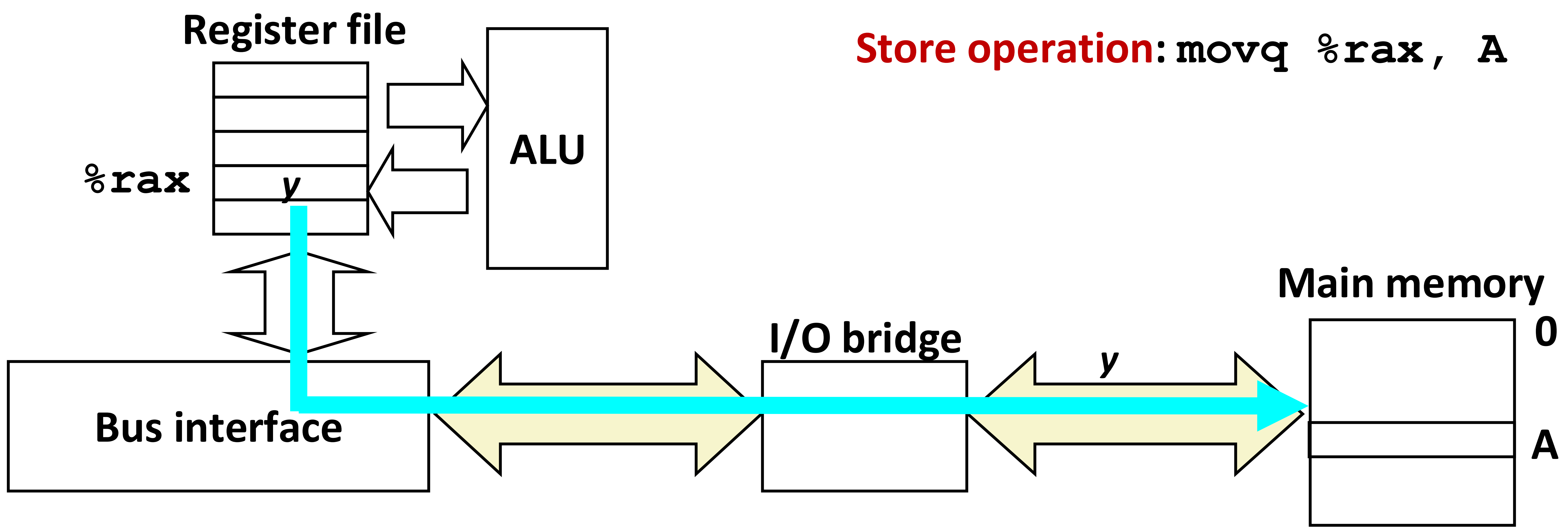
- CPU reads word x from the bus and copies it into register `%rax`.

Memory Write Transaction (1)



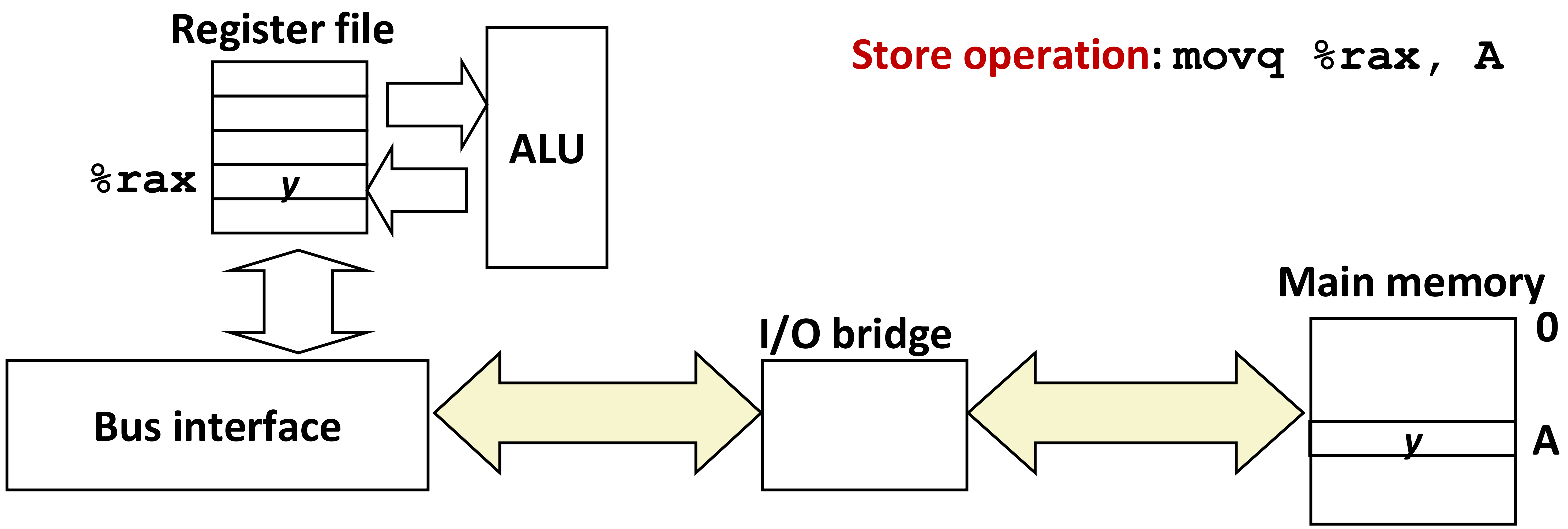
- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.

Memory Write Transaction (2)



- CPU places data word y on the bus.

Memory Write Transaction (3)



- Main memory reads data word y from the bus and stores it at address A.

Basics of Processors

Q: *How does a processor execute machine code?*

- Types of ISA commands to manipulate register contents:
  - Memory access: load (copy bytes from a DRAM address to register); store (reverse); put constant
  - Arithmetic & logic on data items in registers: add/multiply/etc.; bitwise ops; compare, etc.; handled by ALU
  - Control flow (branch, call, etc.); handled by CU
- Caches: Small local memory to buffer instructions/data

If interested in more details: https://www.youtube.com/watch?v=cNN_tTXABUA

You

I cannot believe Artificial general intelligence is just a few Python files and 350GB of weights

What is GPT doing?

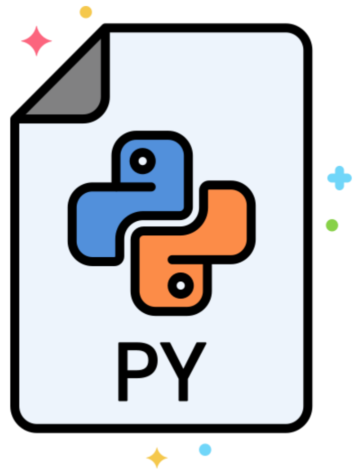
[0, 500, 32768, 1008, 922, ….]

List[integers]

↓

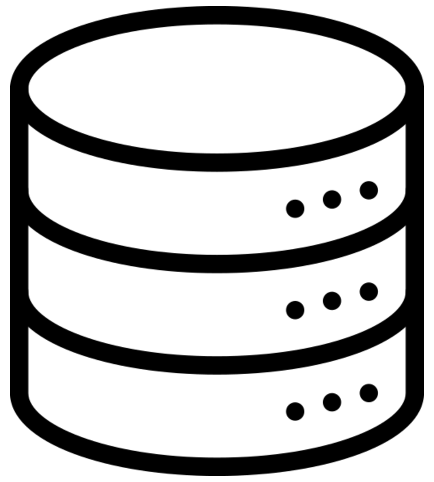
GPT =

Disk

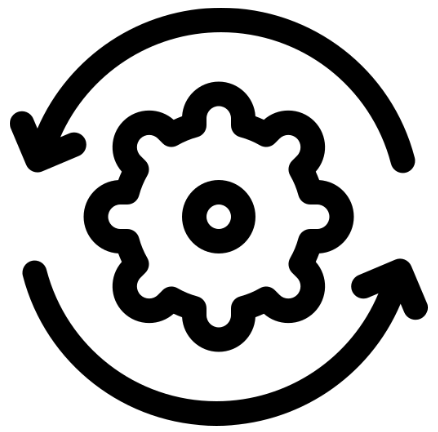


A few KBs

+



Parameters:
350 GB



↓

[0, 25116, 1234, 5984, 6, …]

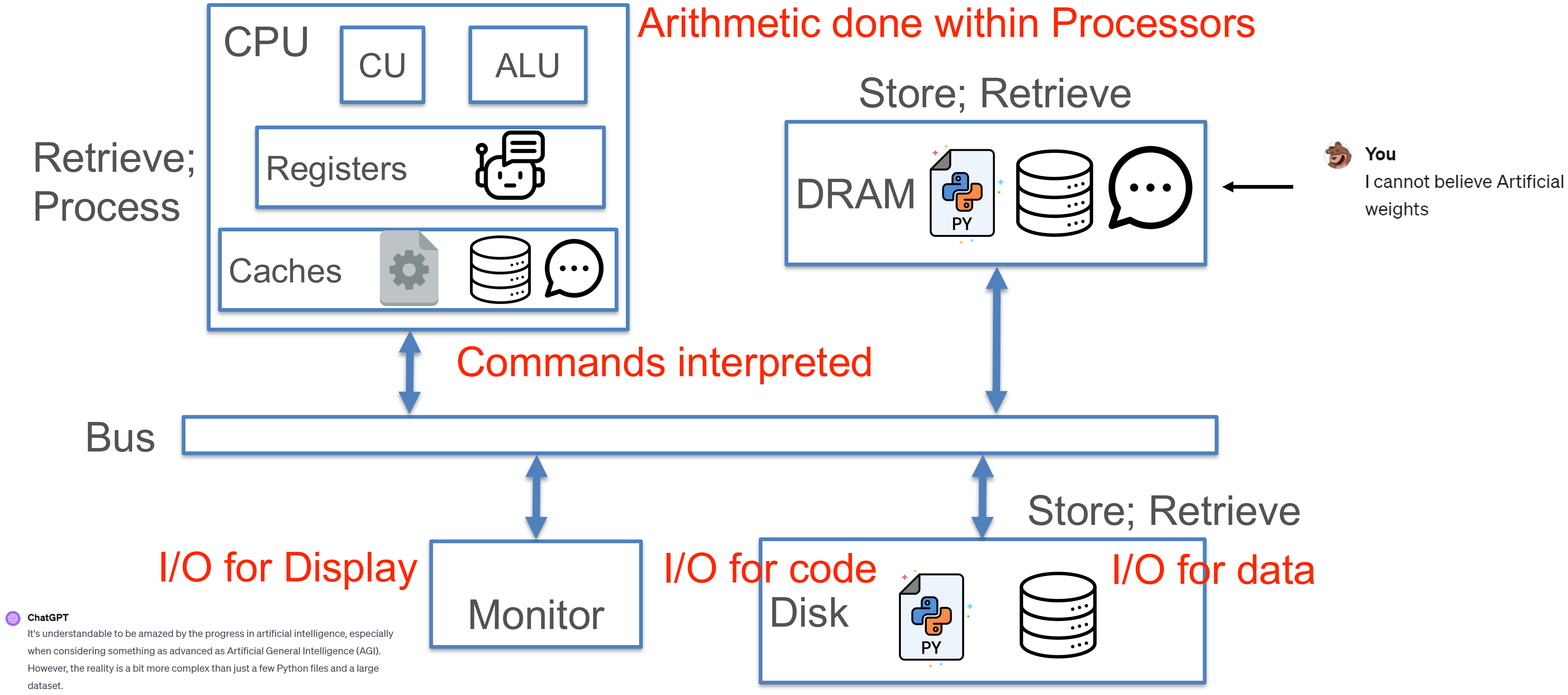
List[integers]

ChatGPT

It's understandable to be amazed by the progress in artificial intelligence, especially when considering something as advanced as Artificial General Intelligence (AGI). However, the reality is a bit more complex than just a few Python files and a large dataset.

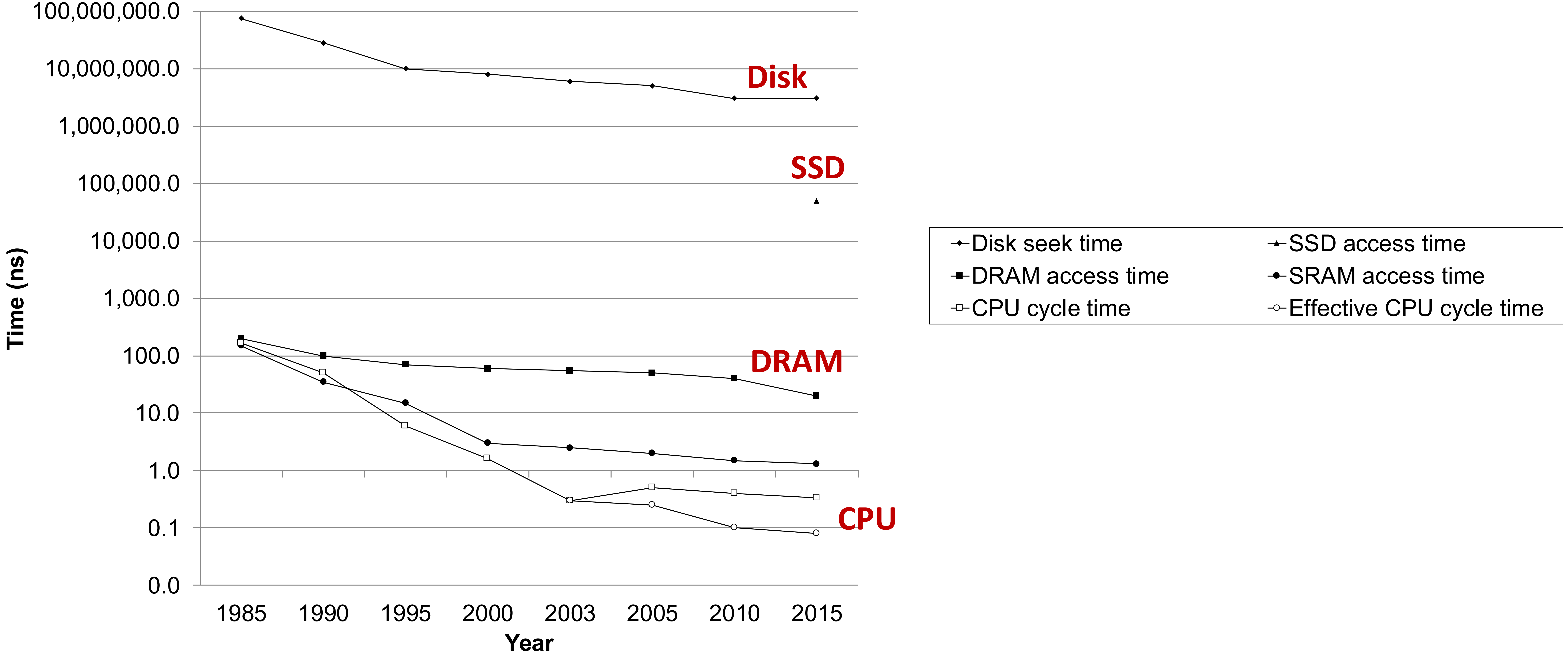
Example

But, how we can make this fast? What are potential problems here?



The CPU-Memory Gap

The gap *widens* between DRAM, disk, and CPU speeds.



Our problem, Simplified

Core Question behind Many System Research

How exactly memory hierarchy solves the gap?

Locality

- The key to bridging this CPU-Memory gap is an important property of computer programs known as locality.

copyij v.s copyji: copy a 2048 X 2048 integer array

```
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];
}
```

4.3 milliseconds

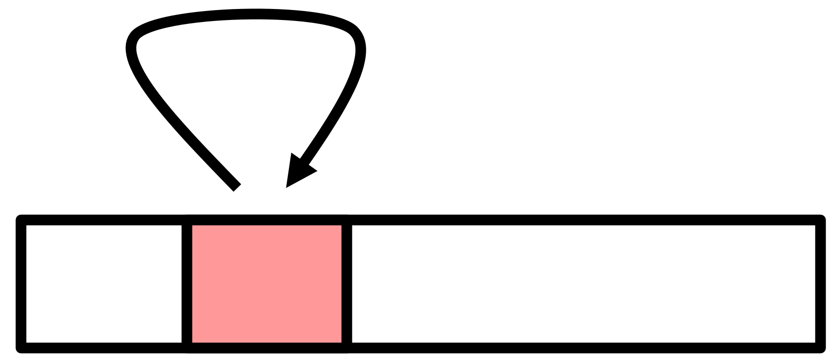
```
void copyji(long int src[2048][2048], long int dst[2048][2048])
{
  long int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

81.8 milliseconds

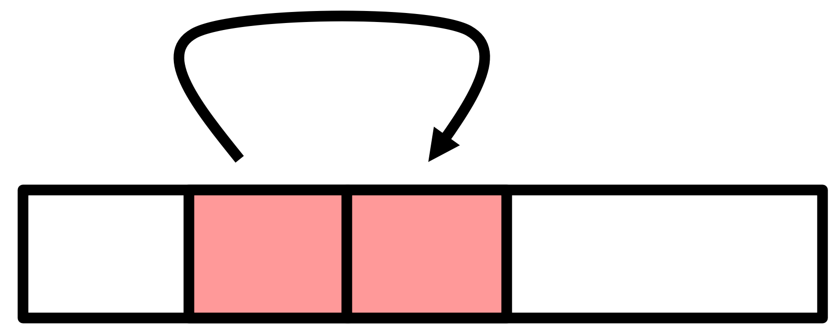
Locality

- 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



Locality Example

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

Spatial or Temporal Locality?

- Data references
  - Reference array elements in succession (stride-1 reference pattern). **spatial**
  - Reference variable **sum** each iteration. **temporal**
- Instruction references
  - Reference instructions in sequence. **spatial**
  - Cycle through loop repeatedly. **temporal**

Qualitative Estimates of Locality

Assuming row-major array

```
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;
}
```

Answer: yes

a [0] [0]	• • •	a [0] [N-1]	a [1] [0]	• • •	a [1] [N-1]	• • •	a [M-1] [0]	• • •	a [M-1] [N-1]

Question: Does this function have good locality with respect to array `a`?

Locality Example

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

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

Answer: no, unless…

M is very small

- Question: Does this function have good locality with respect to array `a`?

a [0] [0]	• • •	a [0] [N-1]	a [1] [0]	• • •	a [1] [N-1]	• • •	a [M-1] [0]	• • •	a [M-1] [N-1]