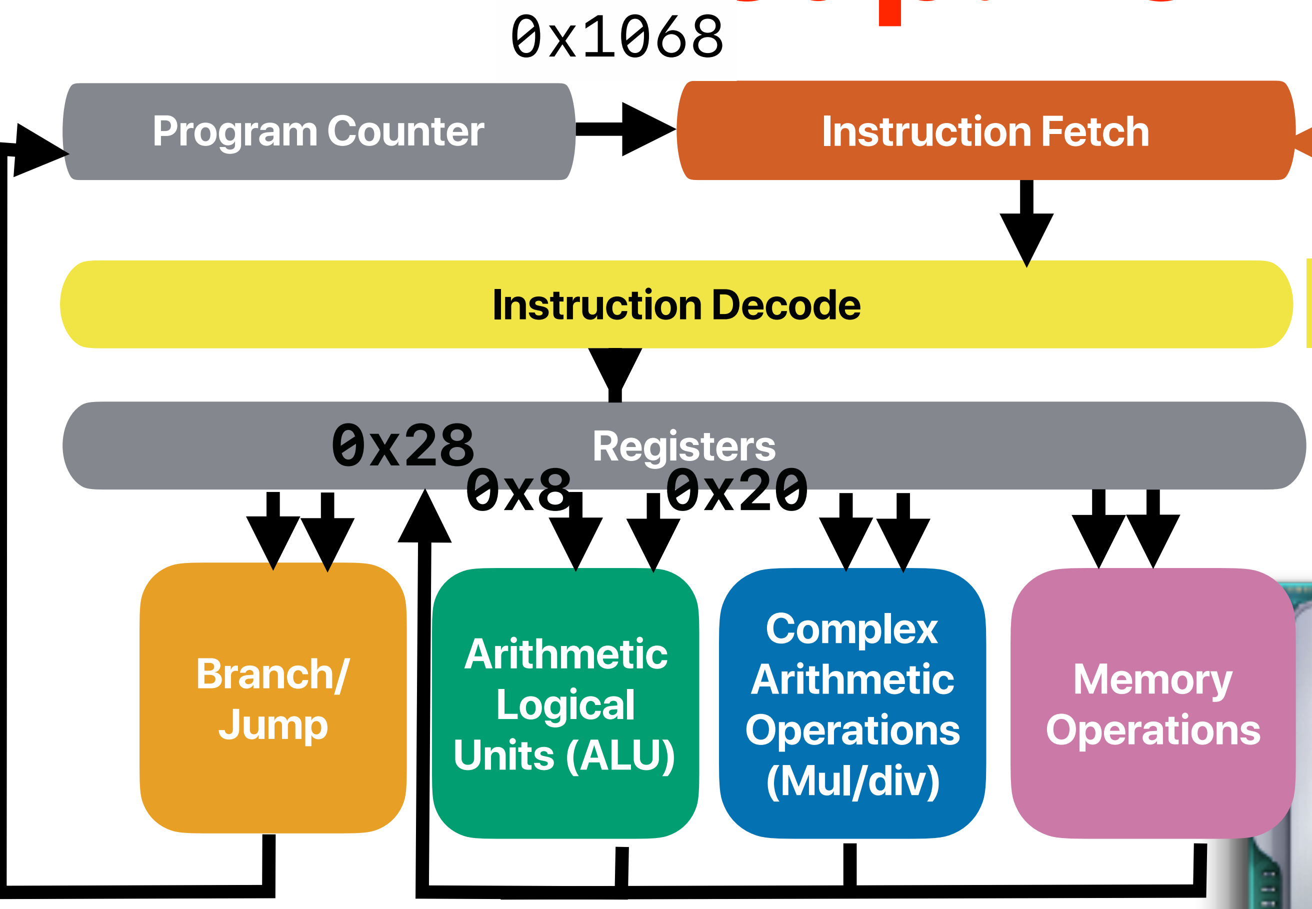
Memory Hierarchy: Basics

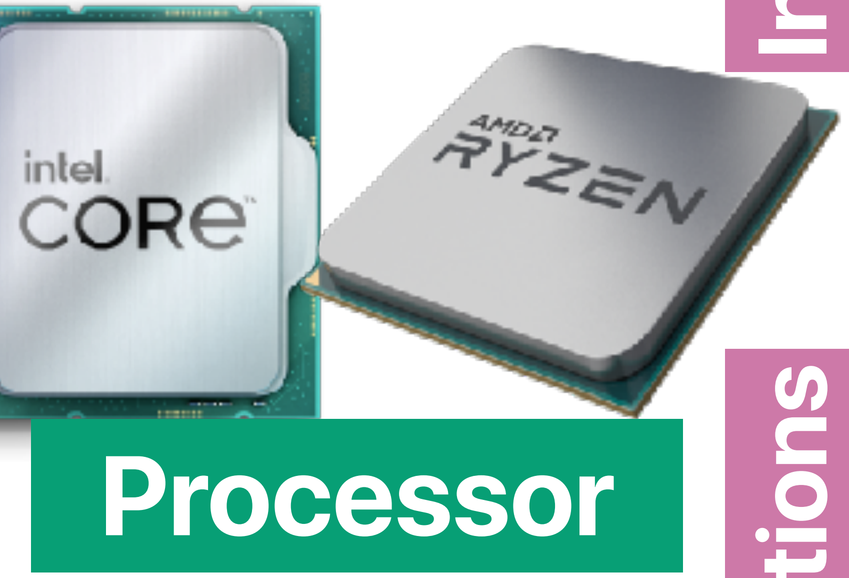
Hung-Wei Tseng

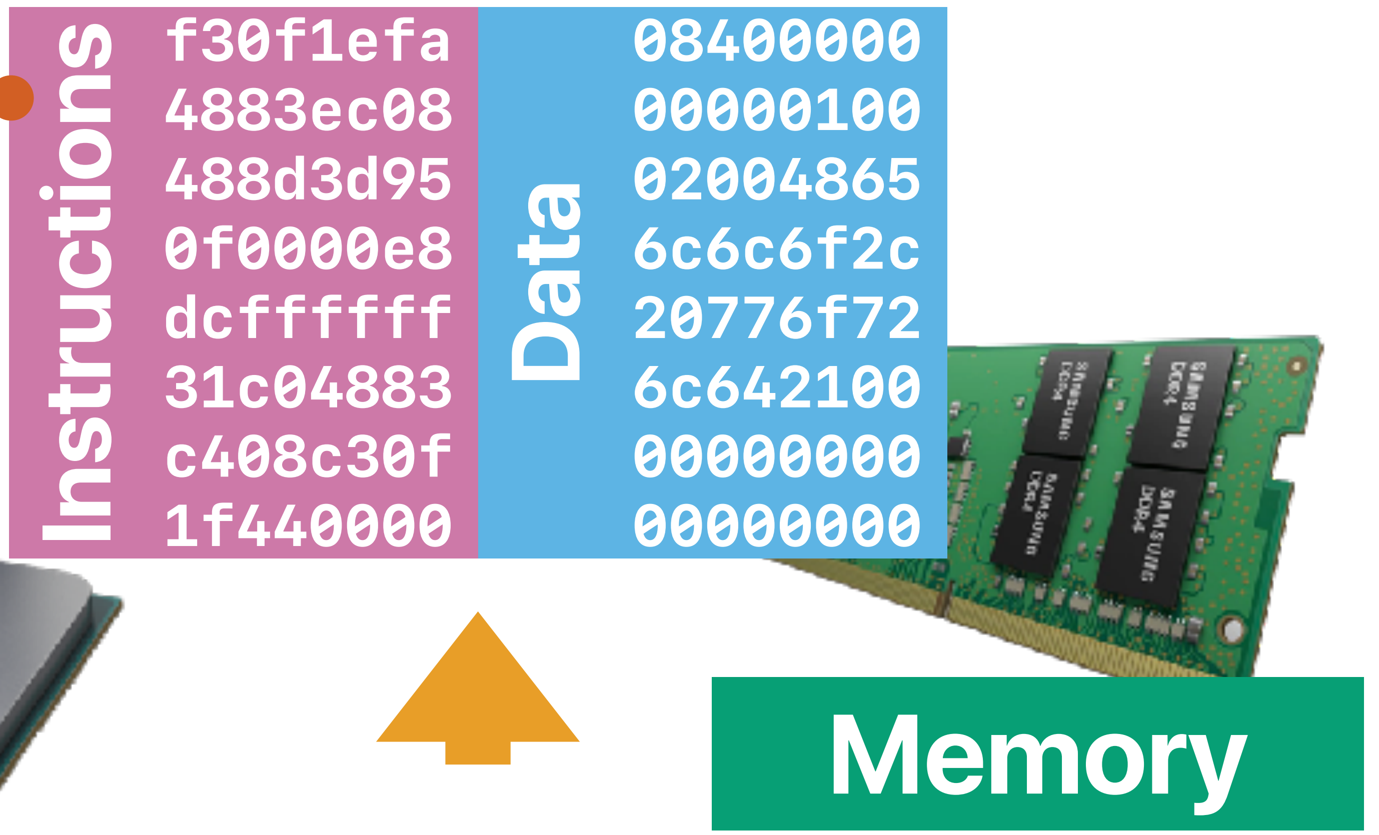
Recap: von Neumann architecture



4883ec08

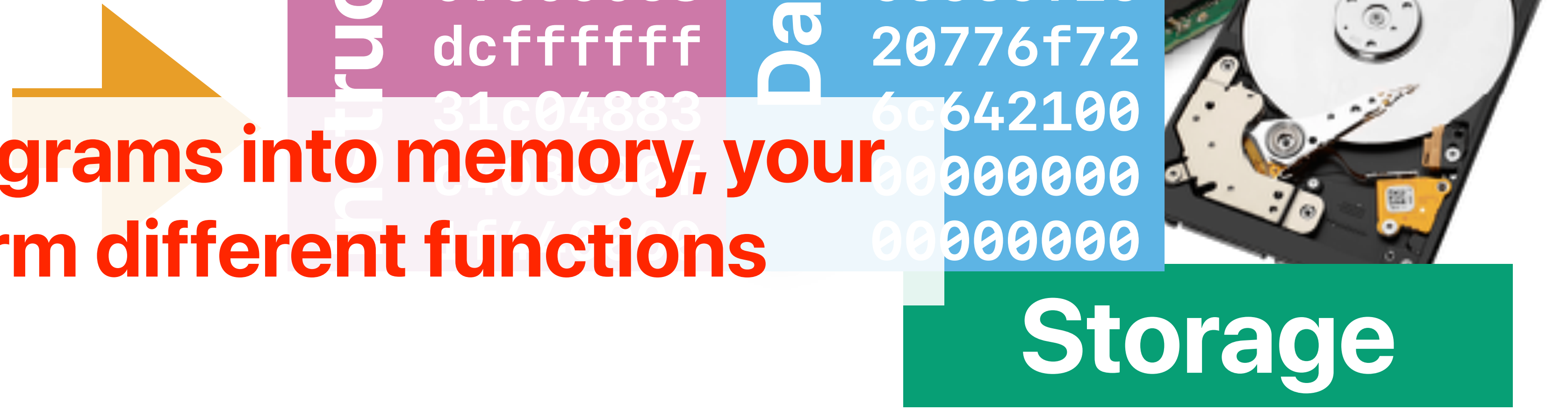
sub $0x8,%rsp





Instructions	Data
f30f1efa	08400000
4883ec08	00000100
488d3d95	02004865
0f0000e8	6c6c6f2c
dcffffff	20776f72
31c04883	6c642100
c408c30f	00000000
1f440000	00000000

```
int main(){
     printf(“Hello, world!\n”);
}
```

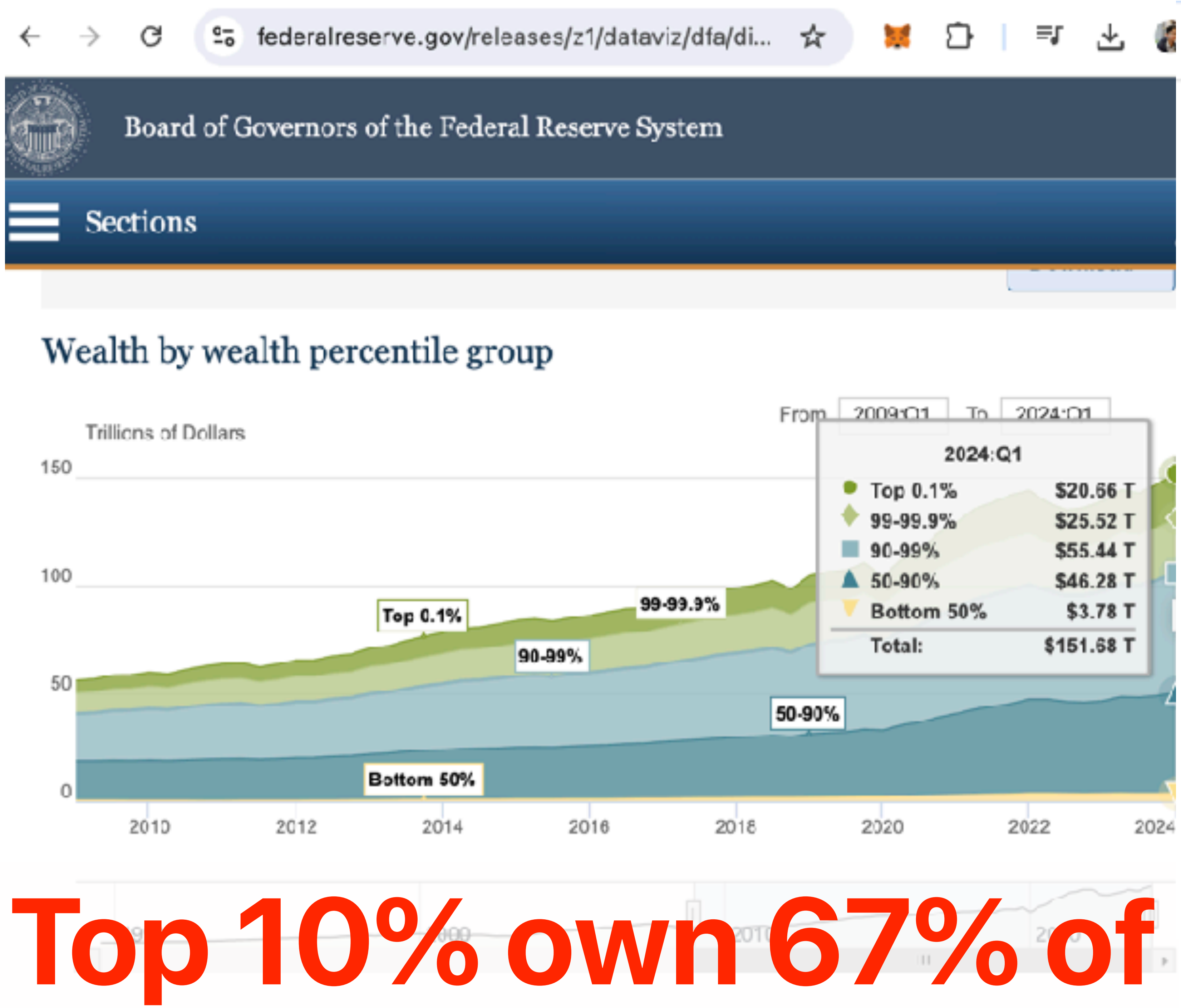


Instructions	Data
f30f1efa	08400000
4883ec08	00000100
488d3d95	02004865
0f0000e8	6c6c6f2c
dcffffff	20776f72
31c04883	6c642100
c408c30f	00000000
1f440000	00000000

By loading different programs into memory, your computer can perform different functions

3:00

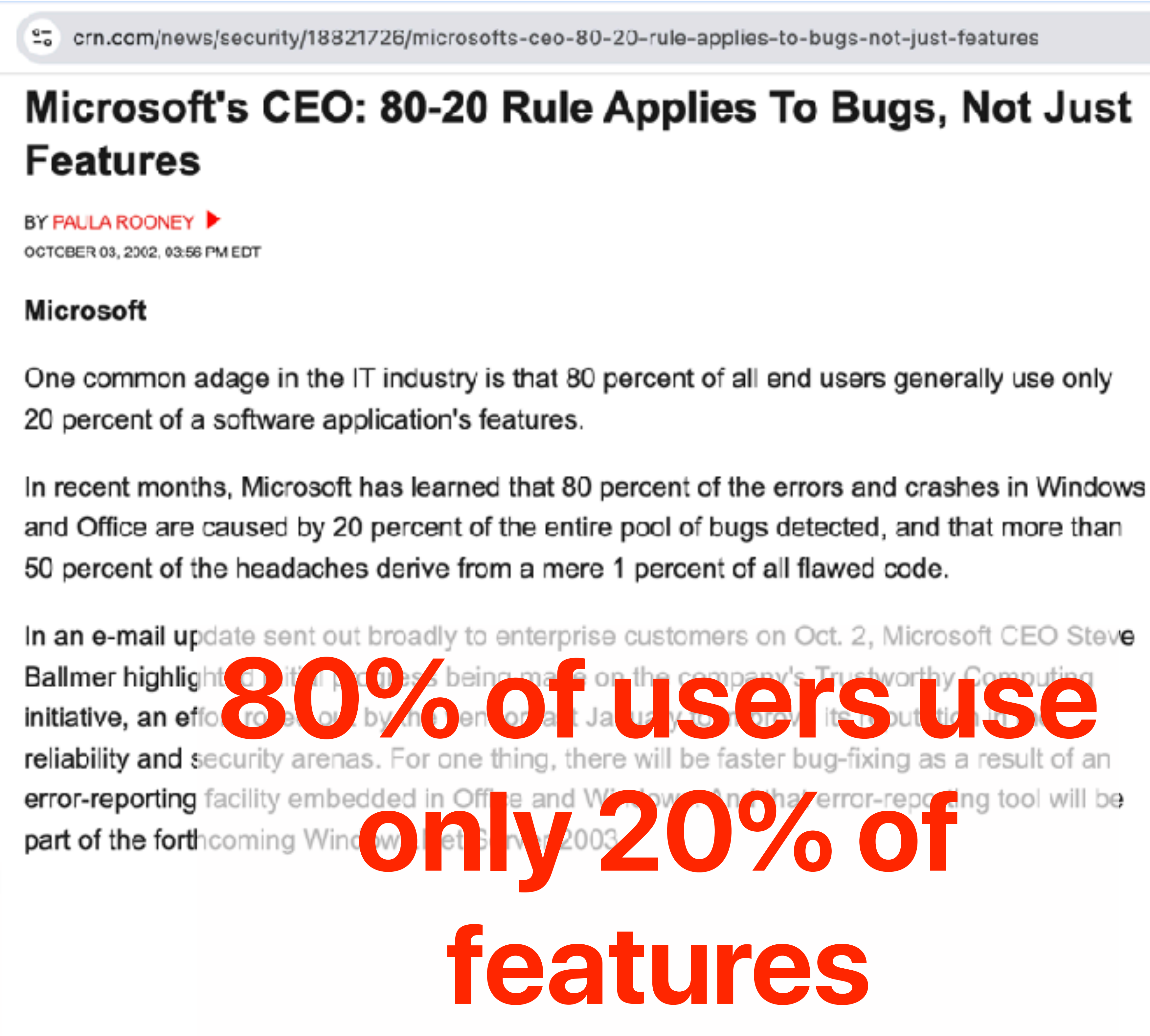
What percentage of “English words” from the dictionary do you think someone has to know for their everyday life?



Wealth by wealth percentile group

2024:Q1	
● Top 0.1%	$20.66 T
◆ 99-99.9%	$25.52 T
■ 90-99%	$55.44 T
▲ 50-90%	$46.28 T
▼ Bottom 50%	$3.78 T
Total:	$151.68 T

Top 10% own 67% of the wealth in the U.S.


crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features

Microsoft's CEO: 80-20 Rule Applies To Bugs, Not Just Features

BY PAULA ROONEY

OCTOBER 03, 2002, 03:56 PM EDT

Microsoft

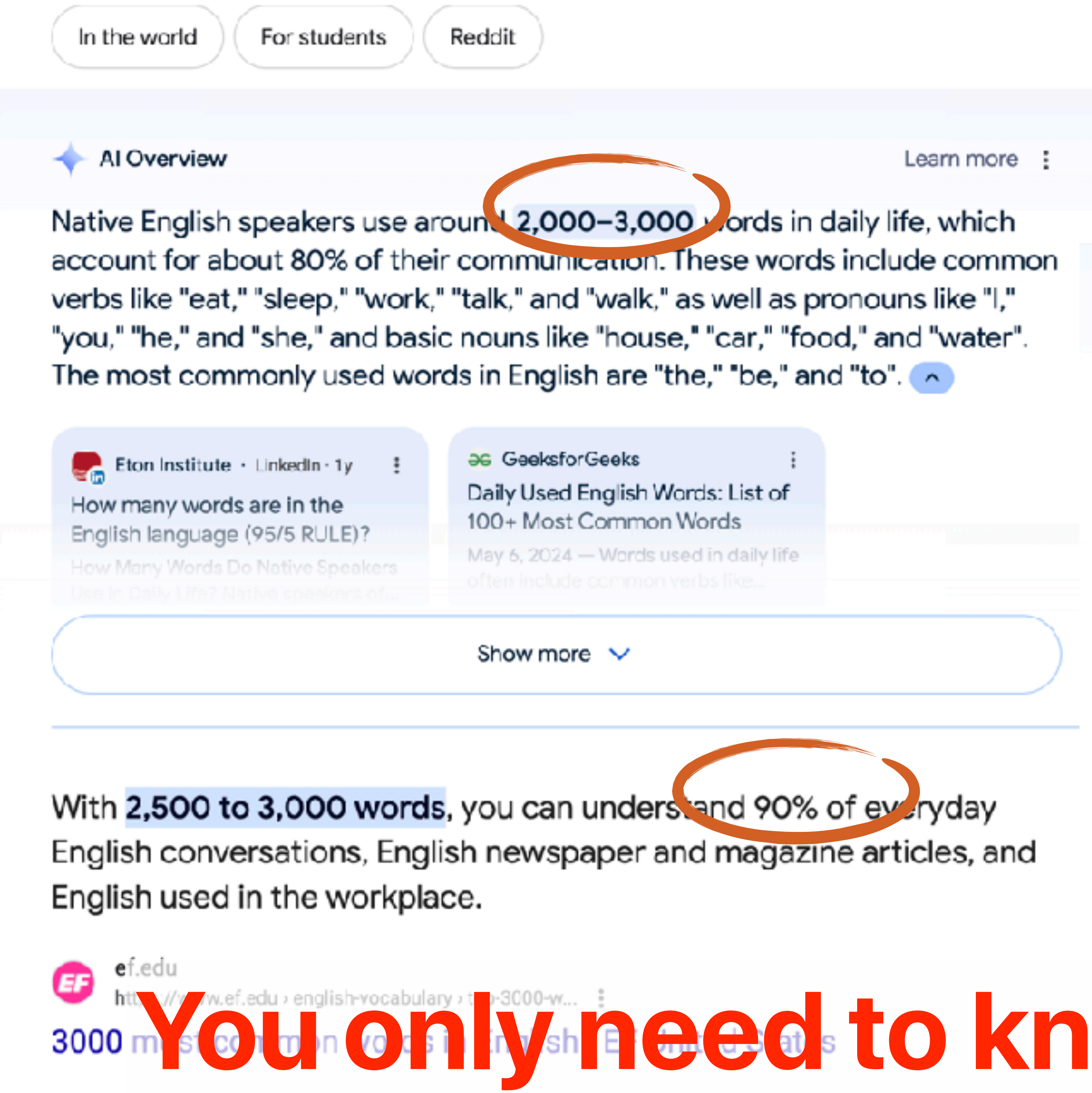
One common adage in the IT industry is that 80 percent of all end users generally use only 20 percent of a software application's features.

In recent months, Microsoft has learned that 80 percent of the errors and crashes in Windows and Office are caused by 20 percent of the entire pool of bugs detected, and that more than 50 percent of the headaches derive from a mere 1 percent of all flawed code.

In an e-mail update sent out broadly to enterprise customers on Oct. 2, Microsoft CEO Steve Ballmer highlighted additional progress being made on the company's Trustworthy Computing initiative, an effort rolled out by the vendor last January to improve its reputation in the reliability and security arenas. For one thing, there will be faster bug-fixing as a result of an error-reporting facility embedded in Office and Windows. And that error-reporting tool will be part of the forthcoming Windows .Net Server 2003

80% of users use only 20% of features

https://en.wikipedia.org/wiki/Pareto_principle



AI Overview

Native English speakers use around 2,000–3,000 words in daily life, which account for about 80% of their communication. These words include common verbs like "eat," "sleep," "work," "talk," and "walk," as well as pronouns like "I," "you," "he," and "she," and basic nouns like "house," "car," "food," and "water". The most commonly used words in English are "the," "be," and "to".

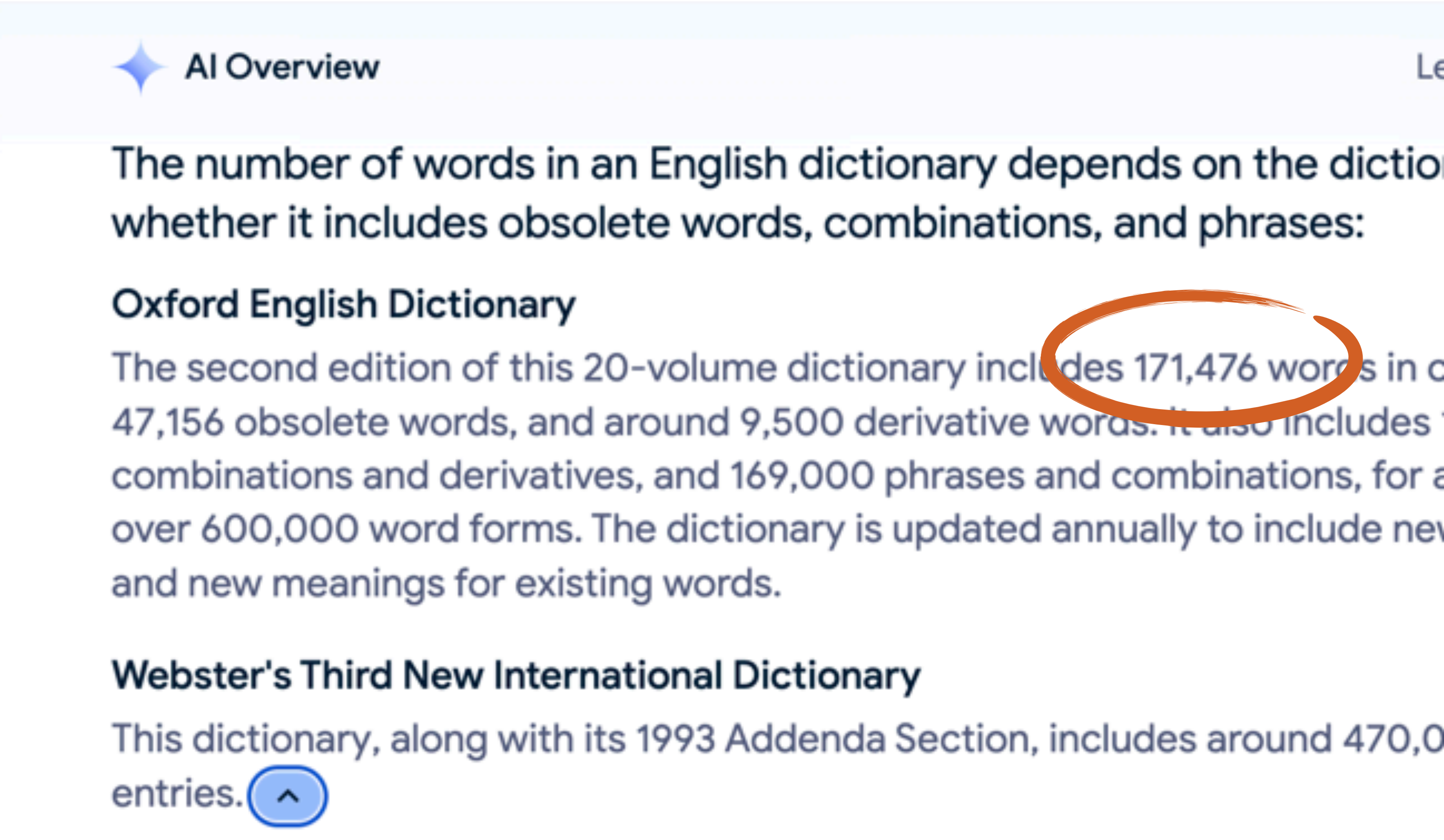
Eton Institute · LinkedIn · 1y
How many words are in the English language (95/5 RULE)?
How Many Words Do Native Speakers Use in Daily Life? Native speakers of...

GeeksforGeeks
Daily Used English Words: List of 100+ Most Common Words
May 6, 2024 — Words used in daily life often include common verbs like...

With 2,500 to 3,000 words, you can understand 90% of everyday English conversations, English newspaper and magazine articles, and English used in the workplace.

ef.edu
https://www.ef.edu › english-vocabulary › top-3000-w...

3000 most common words in English | EF English Stats



AI Overview

The number of words in an English dictionary depends on the dictio
whether it includes obsolete words, combinations, and phrases:

Oxford English Dictionary

The second edition of this 20-volume dictionary includes 171,476 words in c
47,156 obsolete words, and around 9,500 derivative words. It also includes
combinations and derivatives, and 169,000 phrases and combinations, for a
over 600,000 word forms. The dictionary is updated annually to include new
and new meanings for existing words.

Webster's Third New International Dictionary

This dictionary, along with its 1993 Addenda Section, includes around 470,0
entries.

You only need to know 2% English words to understand 90% of conversations

Outline

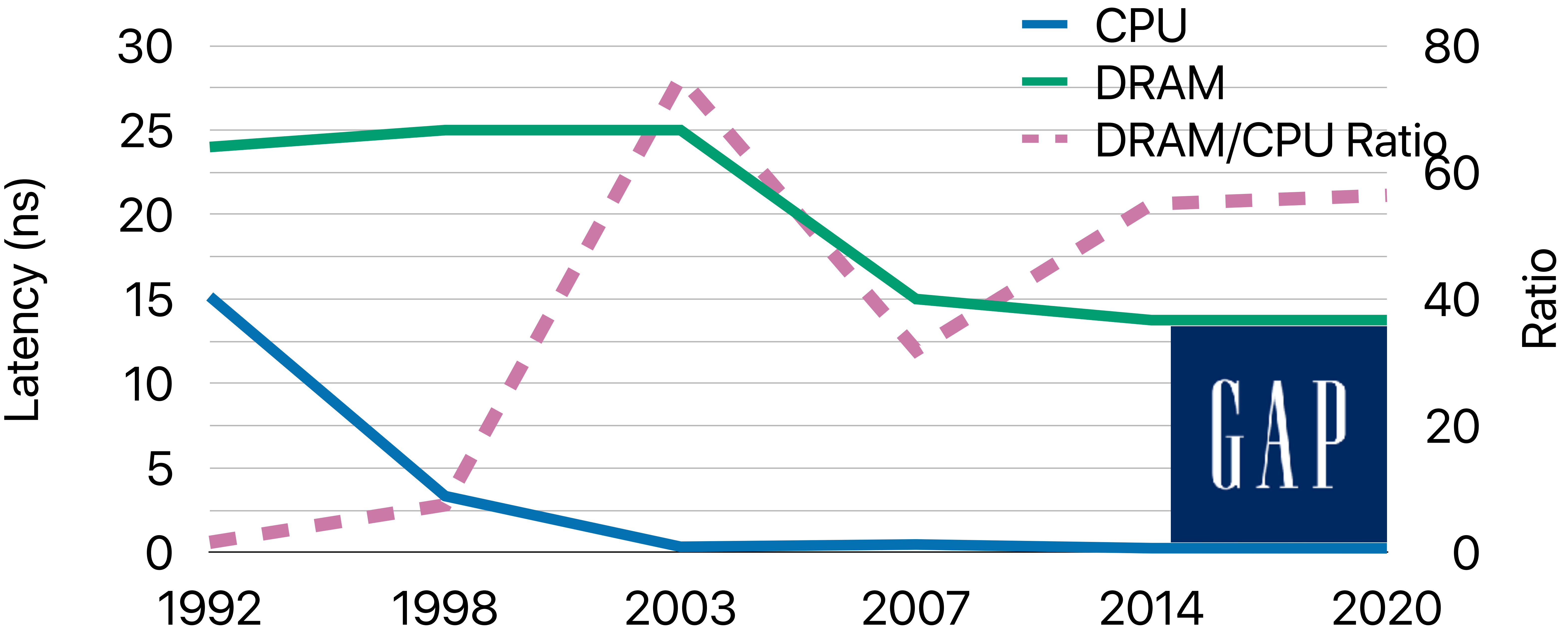
- The memory-wall problem
- The “predictable” code behavior
- Designing a cache that captures the predictability

Modern DRAM performance

SDRAM					DDR				
Data Rate MT/s	Bandwidth GB/s	CAS (clk)	Latency (ns)	Year	Data Rate MT/s	Bandwidth GB/s	CAS (clk)	Latency (ns)	Year
100	0.80	3	**24.00**	**1992**	400	3.20	5	**25.00**	**1998**
133	1.07	3	22.50		667	5.33	5	15.00	
					800	6.40	6	15.00	
		DDR 2					DDR 3		
400	3.20	5	**25.00**	**2003**	800	6.40	6	**15.00**	**2007**
667	5.33	5	15.00		1066	8.53	8	15.00	
800	6.40	6	15.00		1333	10.67	9	13.50	
					1600	12.80	11	13.75	
					1866	14.93	13	13.93	
					2133	17.07	14	13.13	
DDR 4					DDR 5				
1600	12.80	11	**13.75**	**2014**	3200	25.60	22	**13.75**	**2020**
1866	14.93	13	13.92		3600	28.80	26	14.44	
2133	17.07	15	14.06		4000	32.00	28	14.00	
2400	19.20	17	14.17		4400	35.20	32	14.55	
2666	21.33	19	14.25		4800	38.40	34	14.17	
2933	23.46	21	14.32		5200	41.60	38	14.62	
3200	25.20	22	13.75		5600	44.80	40	14.29	
					6000	48.00	42	14.00	
					6400	51.20	46	14.38	

https://www.anandtech.com/show/16143/insights-into-ddr5-subtimings-and-latencies

The “latency” gap between CPU and DRAM



CPU	i486	Pentium II	Pentium 4	Core 2	Core i7-4790K	Core i5-10600K
DRAM	SDRAM	DDR	DDR2	DDR3	DDR4	DDR5

https://www.pollev.com/hungweitseng close in 1:30

The impact of “slow” memory

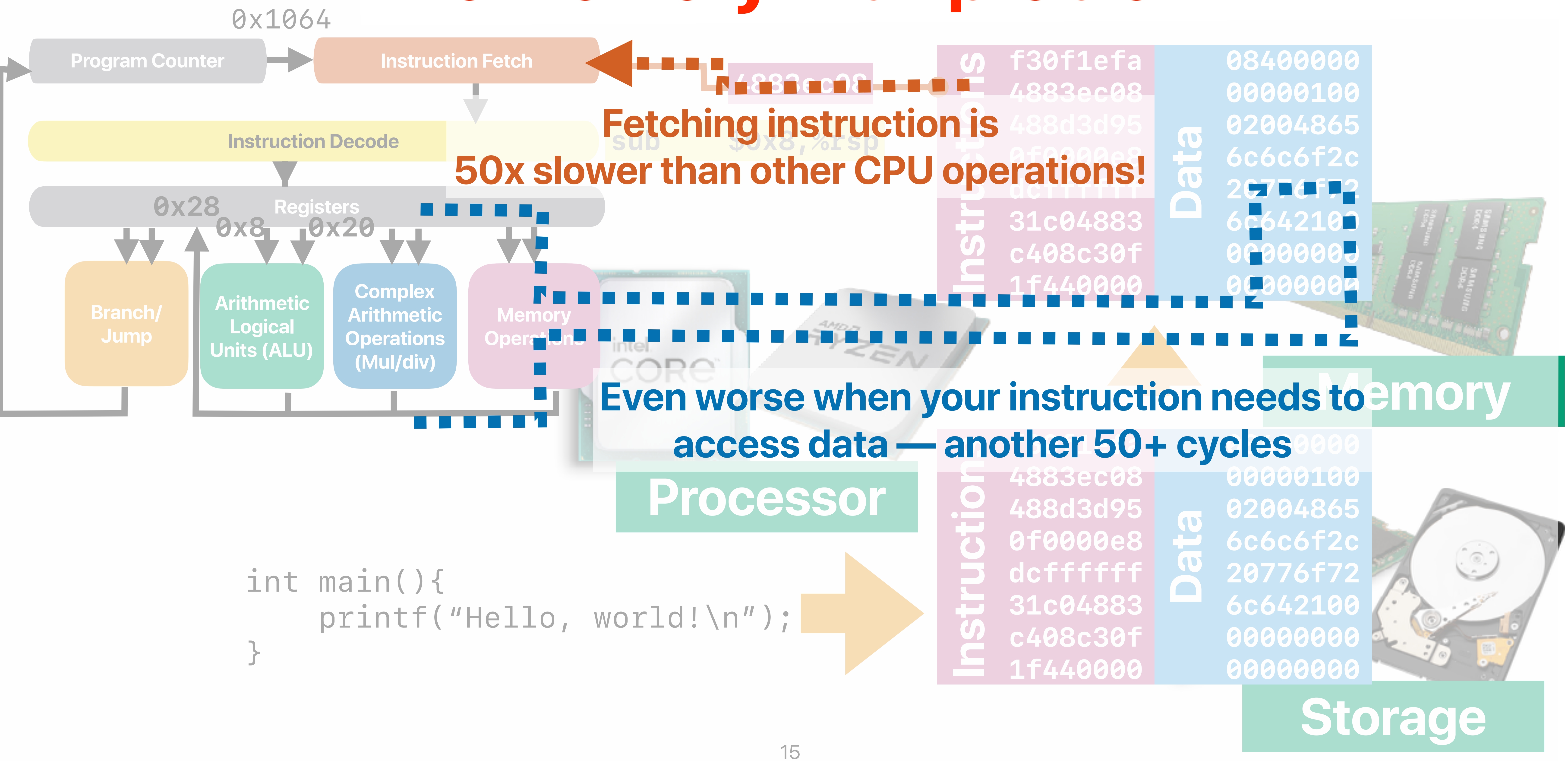
- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access and the processor already fetches the instruction, the CPI is just 1. Now, consider we have DDR5. The program is well-optimized so precharge is never necessary — the memory access latency is 13.75 ns. What’s the average CPI (pick the closest one)?

  A. 9
  B. 12
  C. 15
  D. 56
  E. 67

The impact of “slow” memory

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access and the processor already fetches the instruction, the CPI is just 1. Now, consider we have DDR5. The program is well-optimized so precharge is never necessary — the memory access latency is 13.75 ns. What’s the average CPI (pick the closest one)?
  A. 9
  B. 12
  C. 15
  D. 56
  E. 67

The memory-wall problem



Fetching instruction is 50x slower than other CPU operations!

Even worse when your instruction needs to access data — another 50+ cycles

```
int main(){
    printf("Hello, world!\n");
}
```

The impact of "slow" memory

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access and the processor already fetches the instruction, the CPI is just 1. Now, consider we have DDR5. The program is well-optimized so precharge is never necessary — the memory access latency is 13.75 ns. What's the average CPI (pick the closest one)?
   A. 9
   B. 12
   C. 15
   D. 56
   E. 67

$$CPU\ cycle\ time = \frac{1}{4 \times 10^9} = 0.25ns$$

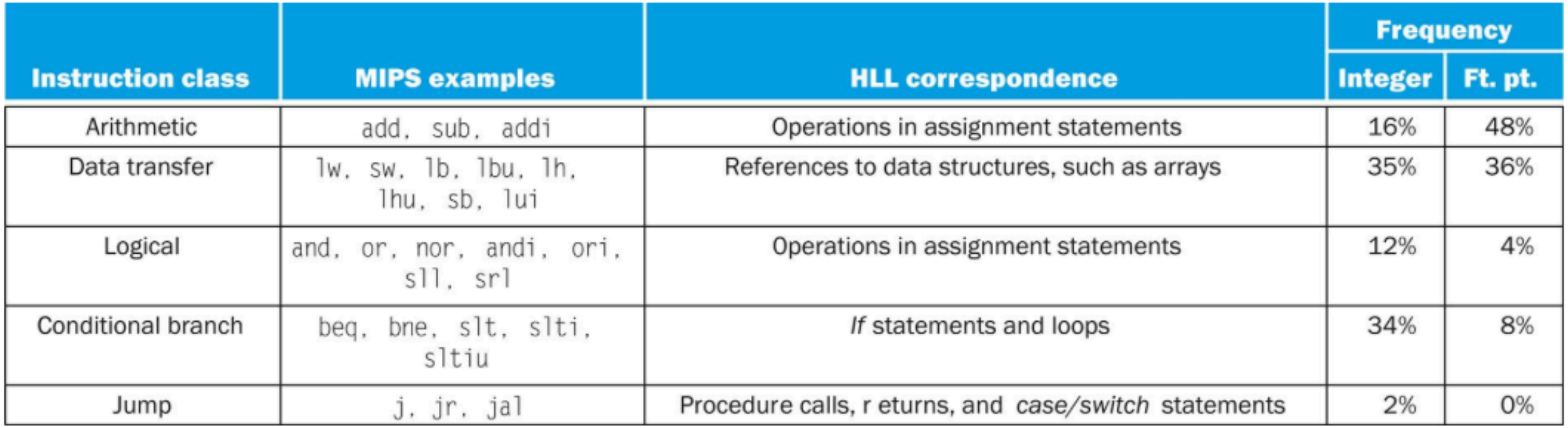
$$Each\ DRAM\ access = \frac{13.75}{0.25} = 55\ cycles$$

$$CPI_{average} = 1 + 100\% \times 55 + 20\% \times 55 = 67\ cycles$$

Don't forget, instructions are also from "memory"

$$\frac{66}{67} = 98.5\ \%\ \textbf{of time, we're dealing with memory accesses!}$$

20% is under-estimating ...



Instruction class	MIPS examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	add, sub, addi	Operations in assignment statements	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	References to data structures, such as arrays	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	Operations in assignment statements	12%	4%
Conditional branch	beq, bne, slt, slti, sltiu	*If* statements and loops	34%	8%
Jump	j, jr, jal	Procedure calls, r eturns, and *case/switch* statements	2%	0%

FIGURE 2.48 MIPS instruction classes, examples, correspondence to high-level program language constructs, and percentage of MIPS instructions executed by category for the average integer and floating point SPEC CPU2006 benchmarks. Figure 3.24 in Chapter 3 shows average percentage of the individual MIPS instructions executed.

Recap: Speedup and Amdahl's Law?

- Definition of "Speedup of Y over X" or say Y is n times faster than X:

  $$speedup_{Y\_over\_X} = n = \frac{Execution\ Time_X}{Execution\ Time_Y}$$

- Amdahl's Law — $Speedup_{enhanced}(f, s) = \frac{1}{(1-f)+\frac{f}{s}}$
  - Corollary 1 — each optimization has an upper bound

    $$Speedup_{max}(f, \infty) = \frac{1}{(1-f)}$$

  - **Corollary 2 — make the common case (the most time consuming case) fast!**

    $$Speedup_{max}(f_1, \infty) = \frac{1}{(1-f_1)}$$
    $$Speedup_{max}(f_2, \infty) = \frac{1}{(1-f_2)}$$
    $$Speedup_{max}(f_3, \infty) = \frac{1}{(1-f_3)}$$
    $$Speedup_{max}(f_4, \infty) = \frac{1}{(1-f_4)}$$

  - Corollary 3: Optimization has a moving target
  - Corollary 4: Exploiting more parallelism from a program is the key to performance gain in modern architectures

    $$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$$

  - Corollary 5: Single-core performance still matters

    $$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$$

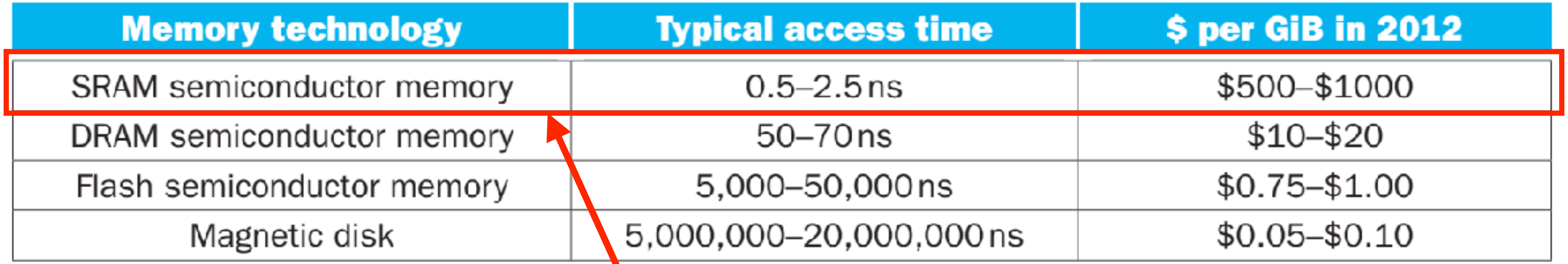
  - Corollary 6: Don't hurt the non-common case too much

    $$Speedup_{enhanced}(f, s, r) = \frac{1}{(1-f)+perf(r)+\frac{f}{s}}$$

Take-aways: inside out our memory hierarchy

- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!

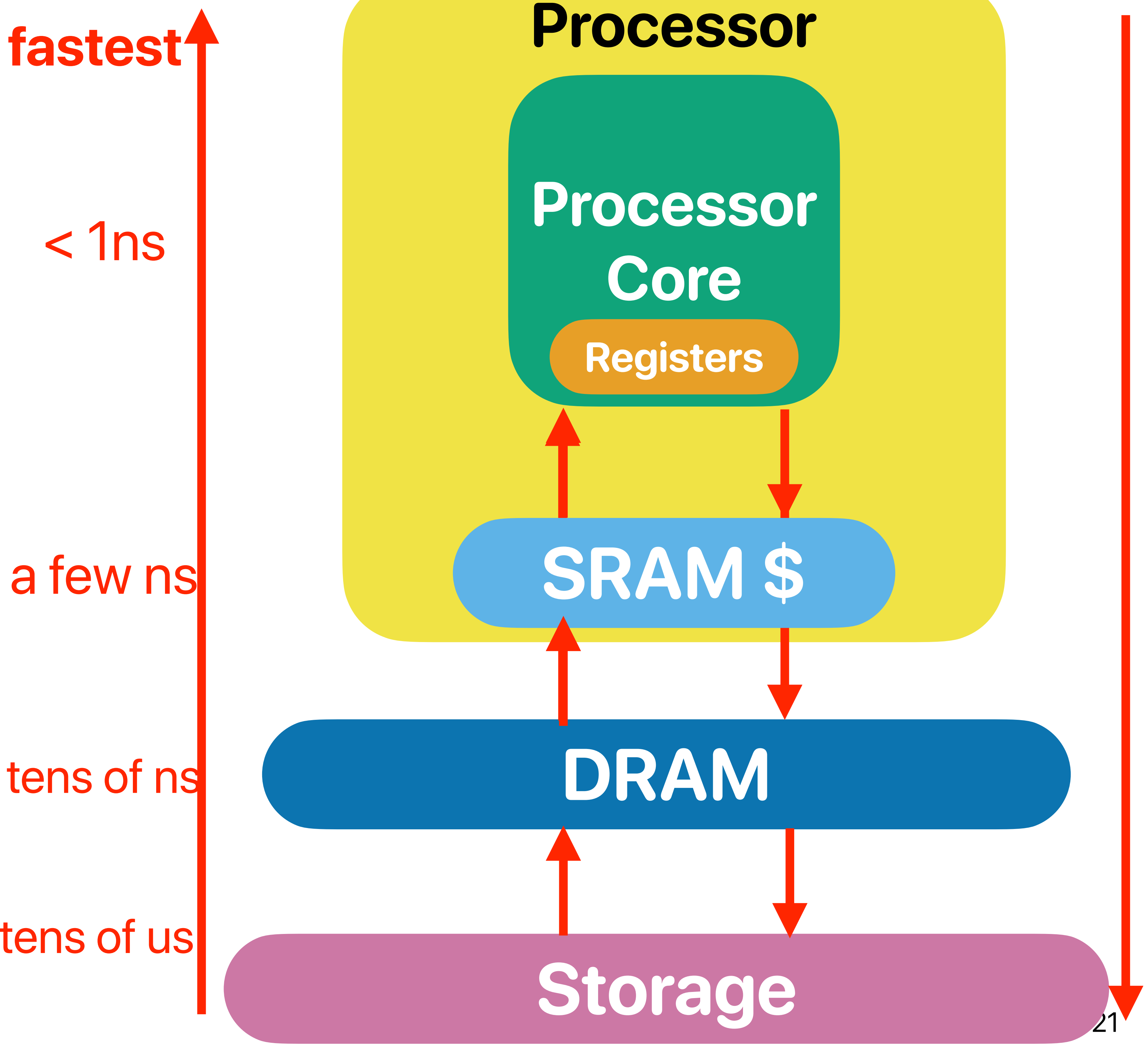
Alternatives?



Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

Fast, but expensive $$$

Memory Hierarchy



32 or 64 words

KBs ~ MBs

GBs

TBs

larger

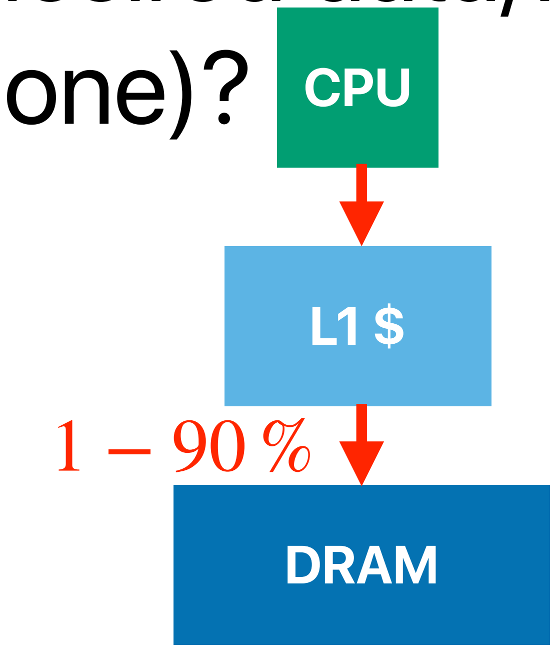
https://www.pollev.com/hungweitseng close in 1:30

How can “memory hierarchy” help in performance?

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to we DDR5, whose latency 13.75 ns, we also got an SRAM cache with latency of just at 0.5 ns and can capture **90%** of the desired data/instructions. what’s the average CPI (pick the closest one)?
   A. 6
   B. 8
   C. 10
   D. 12
   E. 67

How can “memory hierarchy” help in performance?

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to we DDR5, whose latency 13.75 ns, we also got an SRAM cache with latency of just at 0.5 ns and can capture **90%** of the desired data/instructions. what’s the average CPI (pick the closest one)?
  A. 6
  B. 8
  [✓ C. 10]
  D. 12
  E. 67



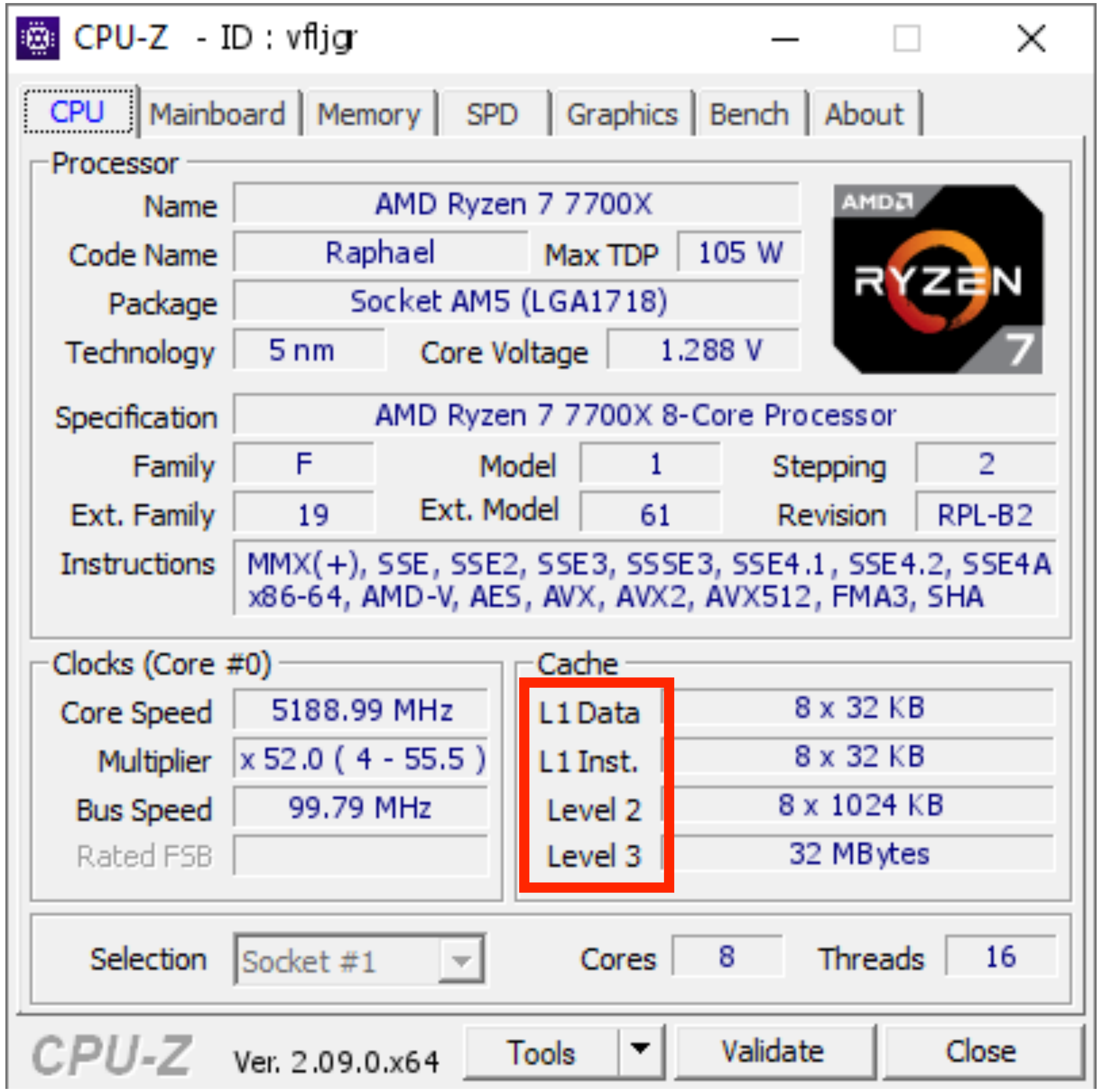
$$CPU\ cycle\ time = \frac{1}{4 \times 10^9} = 0.25ns$$

$$Each\ \$\ access = \frac{0.5}{0.25} = 2\ cycles$$

$$Each\ DRAM\ access = \frac{13.75}{0.25} = 55\ cycles$$

$$CPI_{average} = 1 + 100\% \times [2 + (1 - 90\%) \times 55] + 20\% \times [2 + (1 - 90\%) \times 55] = 10\ cycles$$

L1? L2? L3?



Processor

Name: AMD Ryzen 7 7700X
Code Name: Raphael
Max TDP: 105 W
Package: Socket AM5 (LGA1718)
Technology: 5 nm
Core Voltage: 1.288 V
Specification: AMD Ryzen 7 7700X 8-Core Processor
Family: F
Model: 1
Stepping: 2
Ext. Family: 19
Ext. Model: 61
Revision: RPL-B2
Instructions: MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A x86-64, AMD-V, AES, AVX, AVX2, AVX512, FMA3, SHA

Clocks (Core #0)

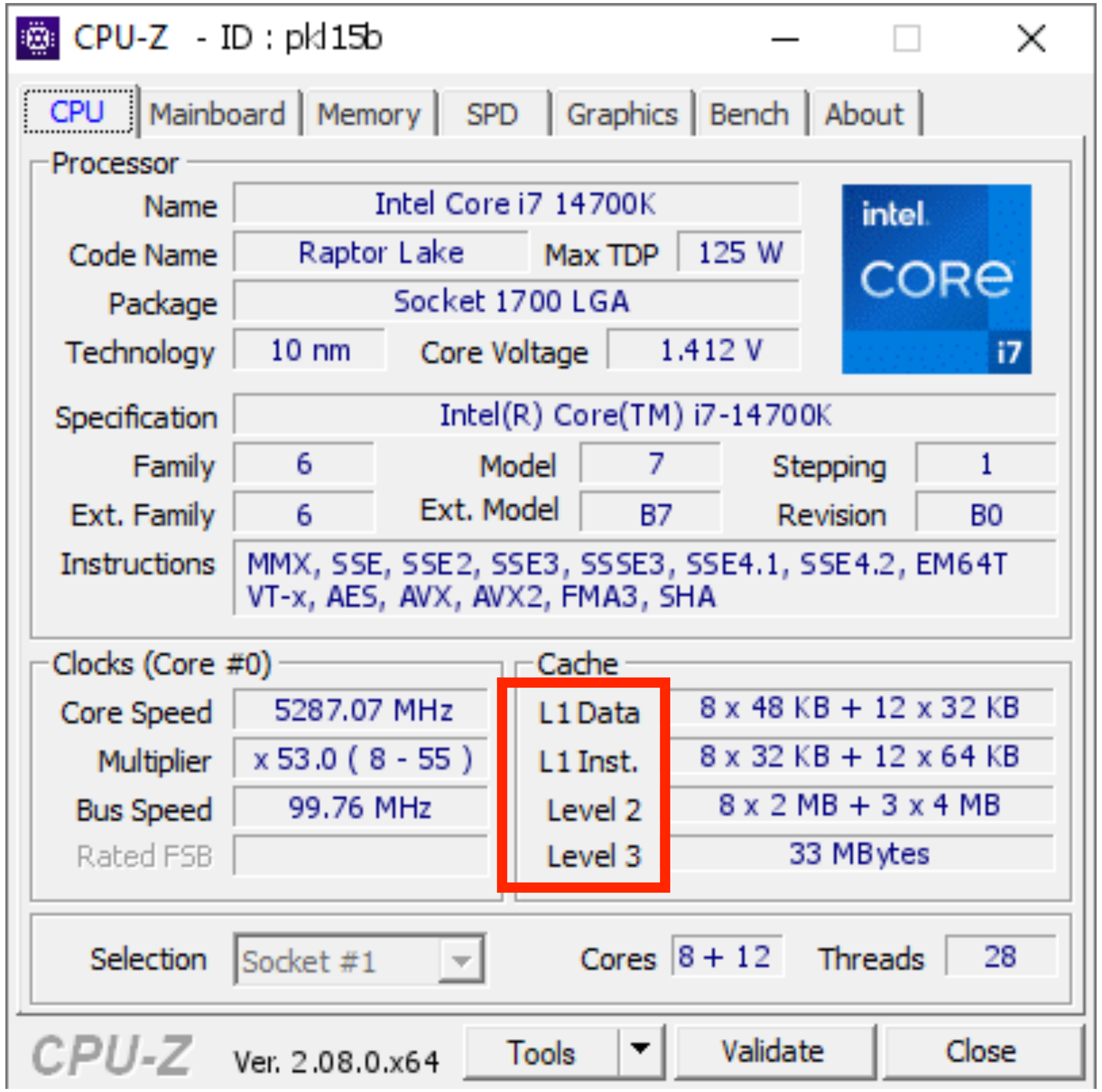
Core Speed: 5188.99 MHz
Multiplier: x 52.0 (4 - 55.5)
Bus Speed: 99.79 MHz
Rated FSB:

Cache

L1 Data: 8 x 32 KB
L1 Inst.: 8 x 32 KB
Level 2: 8 x 1024 KB
Level 3: 32 MBytes

Selection: Socket #1
Cores: 8
Threads: 16

Ver. 2.09.0.x64



Processor

Name: Intel Core i7 14700K
Code Name: Raptor Lake
Max TDP: 125 W
Package: Socket 1700 LGA
Technology: 10 nm
Core Voltage: 1.412 V
Specification: Intel(R) Core(TM) i7-14700K
Family: 6
Model: 7
Stepping: 1
Ext. Family: 6
Ext. Model: B7
Revision: B0
Instructions: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T VT-x, AES, AVX, AVX2, FMA3, SHA

Clocks (Core #0)

Core Speed: 5287.07 MHz
Multiplier: x 53.0 (8 - 55)
Bus Speed: 99.76 MHz
Rated FSB:

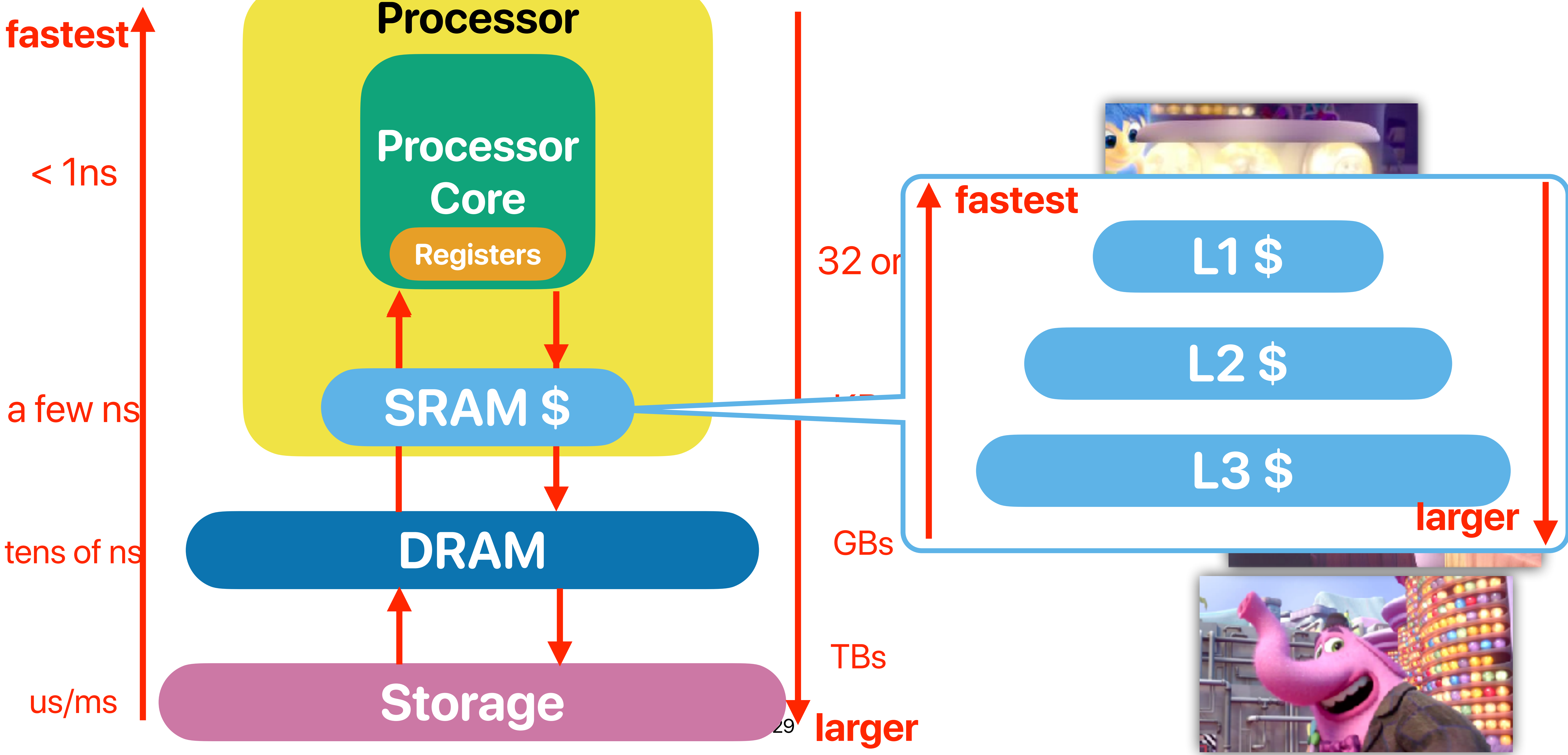
Cache

L1 Data: 8 x 48 KB + 12 x 32 KB
L1 Inst.: 8 x 32 KB + 12 x 64 KB
Level 2: 8 x 2 MB + 3 x 4 MB
Level 3: 33 MBytes

Selection: Socket #1
Cores: 8 + 12
Threads: 28

Ver. 2.08.0.x64

Memory Hierarchy



https://www.pollev.com/hungweitseng close in 1:30

How can a deeper memory hierarchy help in performance?

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to the DDR5, whose latency is 13.75 ns, we also got 2 levels SRAM caches where
  - the 1st-level one at latency of 0.5ns and can capture 90% of the desired data/instructions.
  - the 2nd-level at latency of 5 ns and can capture 60% of the desired data/instructions

  What's the average CPI (pick the closest one)?

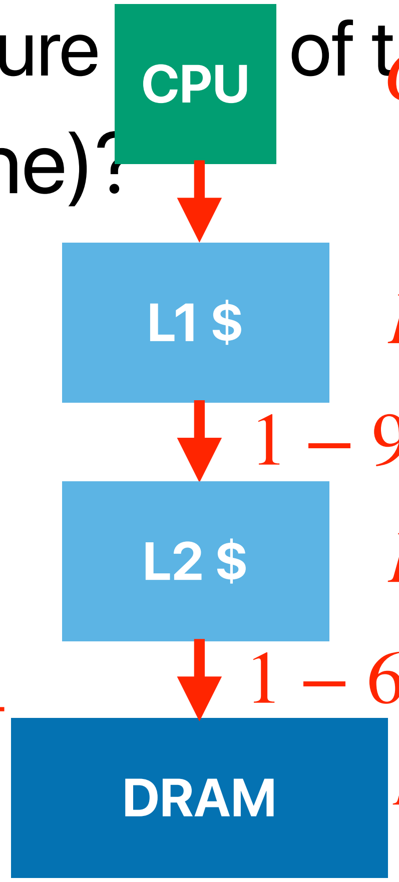
  A. 6
  B. 8
  C. 10
  D. 12
  E. 67

How can a deeper memory hierarchy help in performance?

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to the DDR5, whose latency is 13.75 ns, we also got 2 levels SRAM caches where
  - the 1st-level one at latency of 0.5ns and can capture 90% of the desired data/instructions.
  - the 2nd-level at latency of 5 ns and can capture of the desired data/instructions

What's the average CPI (pick the closest one)?

A. 6
B. 8
C. 10
D. 12
E. 67



$1 - 90\ \%$

$1 - 60\ \%$

$$CPU\ cycle\ time = \frac{1}{4 \times 10^9} = 0.25ns$$

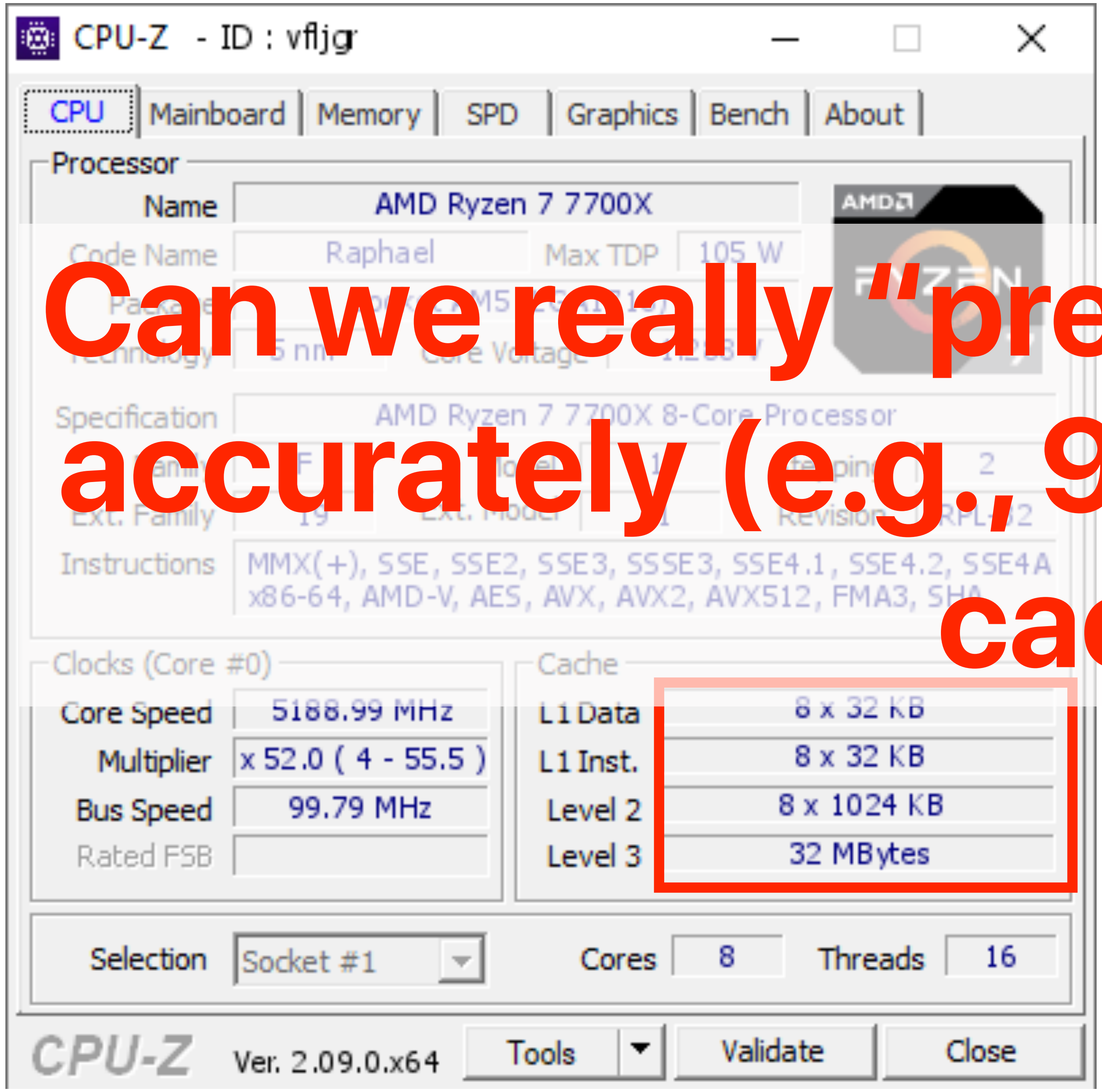
$$Each\ L1_\$\ access = \frac{0.5}{0.25} = 2\ cycles$$

$$Each\ L2_\$\ access = \frac{5}{0.25} = 20\ cycles$$

$$Each\ DRAM\ access = \frac{13.75}{0.25} = 55\ cycles$$

$$CPI_{average} = 1 + 100\% \times [2 + (1 - 90\%) \times (20 + (1 - 60\%) \times 55] +$$
$$20\% \times [2 + (1 - 90\%) \times (20 + (1 - 60\%) \times 55)]$$
$$= 8.44\ cycles$$

L1? L2? L3?



CPU-Z - ID : vfljgr

CPU | Mainboard | Memory | SPD | Graphics | Bench | About

Processor

Name: AMD Ryzen 7 7700X
Code Name: Raphael
Max TDP: 105 W
Technology: 5 nm
Specification: AMD Ryzen 7 7700X 8-Core Processor
Stepping: 2
Ext. Family: 19
Revision: RPL-B2
Instructions: MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A x86-64, AMD-V, AES, AVX, AVX2, AVX512, FMA3, SHA

Clocks (Core #0)

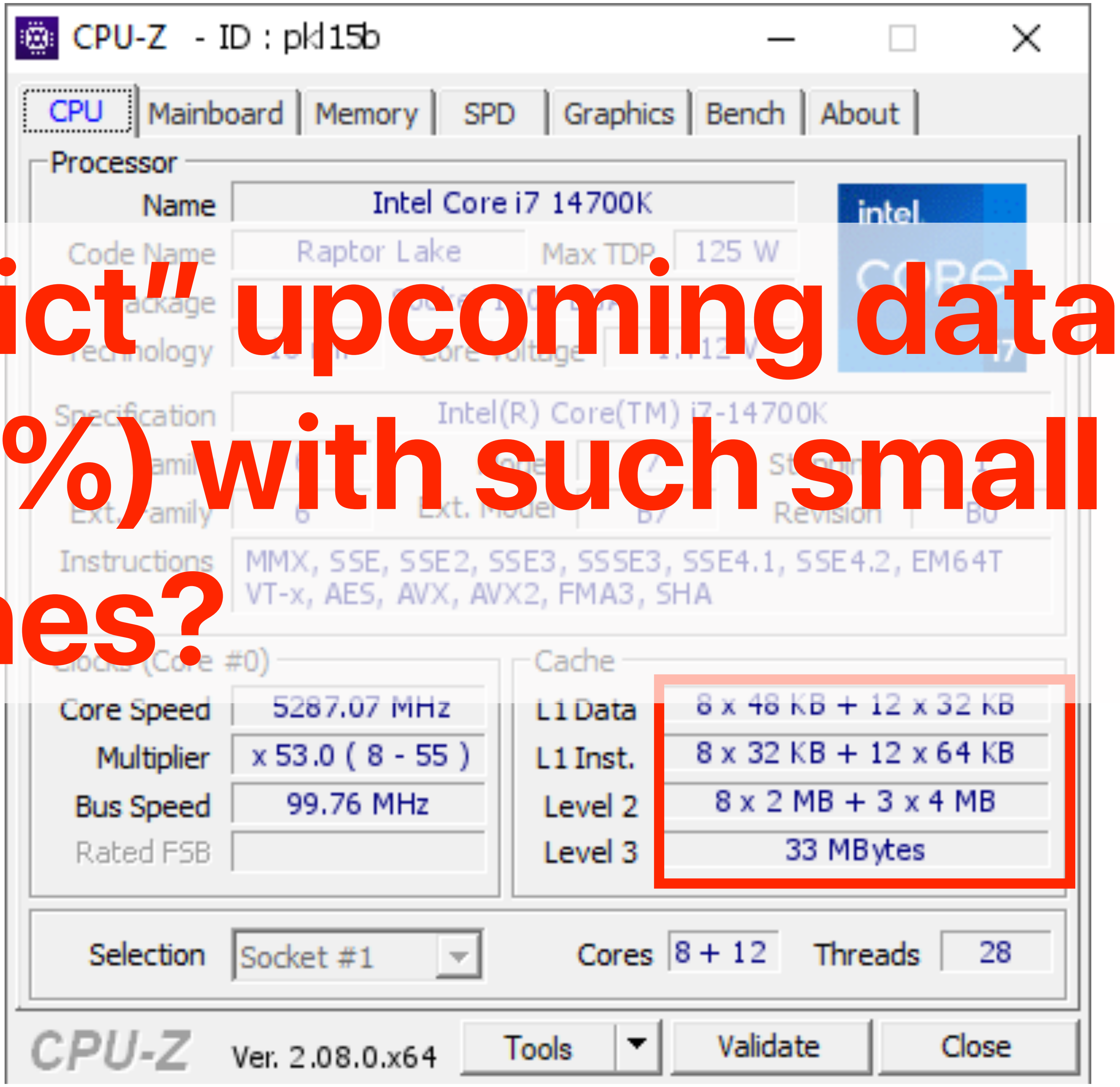
Core Speed: 5188.99 MHz
Multiplier: x 52.0 (4 - 55.5)
Bus Speed: 99.79 MHz
Rated FSB:

Cache

L1 Data: 8 x 32 KB
L1 Inst.: 8 x 32 KB
Level 2: 8 x 1024 KB
Level 3: 32 MBytes

Selection: Socket #1
Cores: 8
Threads: 16

CPU-Z Ver. 2.09.0.x64



CPU-Z - ID : pkl15b

CPU | Mainboard | Memory | SPD | Graphics | Bench | About

Processor

Name: Intel Core i7 14700K
Code Name: Raptor Lake
Max TDP: 125 W
Specification: Intel(R) Core(TM) i7-14700K
Ext. Family: 6
Ext. Model: B7
Revision: B0
Instructions: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T VT-x, AES, AVX, AVX2, FMA3, SHA

Clocks (Core #0)

Core Speed: 5287.07 MHz
Multiplier: x 53.0 (8 - 55)
Bus Speed: 99.76 MHz
Rated FSB:

Cache

L1 Data: 8 x 48 KB + 12 x 32 KB
L1 Inst.: 8 x 32 KB + 12 x 64 KB
Level 2: 8 x 2 MB + 3 x 4 MB
Level 3: 33 MBytes

Selection: Socket #1
Cores: 8 + 12
Threads: 28

CPU-Z Ver. 2.08.0.x64

Can we really "predict" upcoming data accurately (e.g., 90%) with such small caches?

Take-aways: inside out our memory hierarchy

- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!
- Hierarchical caching with small amount of SRAMs will work if we can efficiently capture data and instructions

The predictability of your code

The Machine Learning Inference Pipeline

Feature Extraction

Classification

Convolution + ReLU

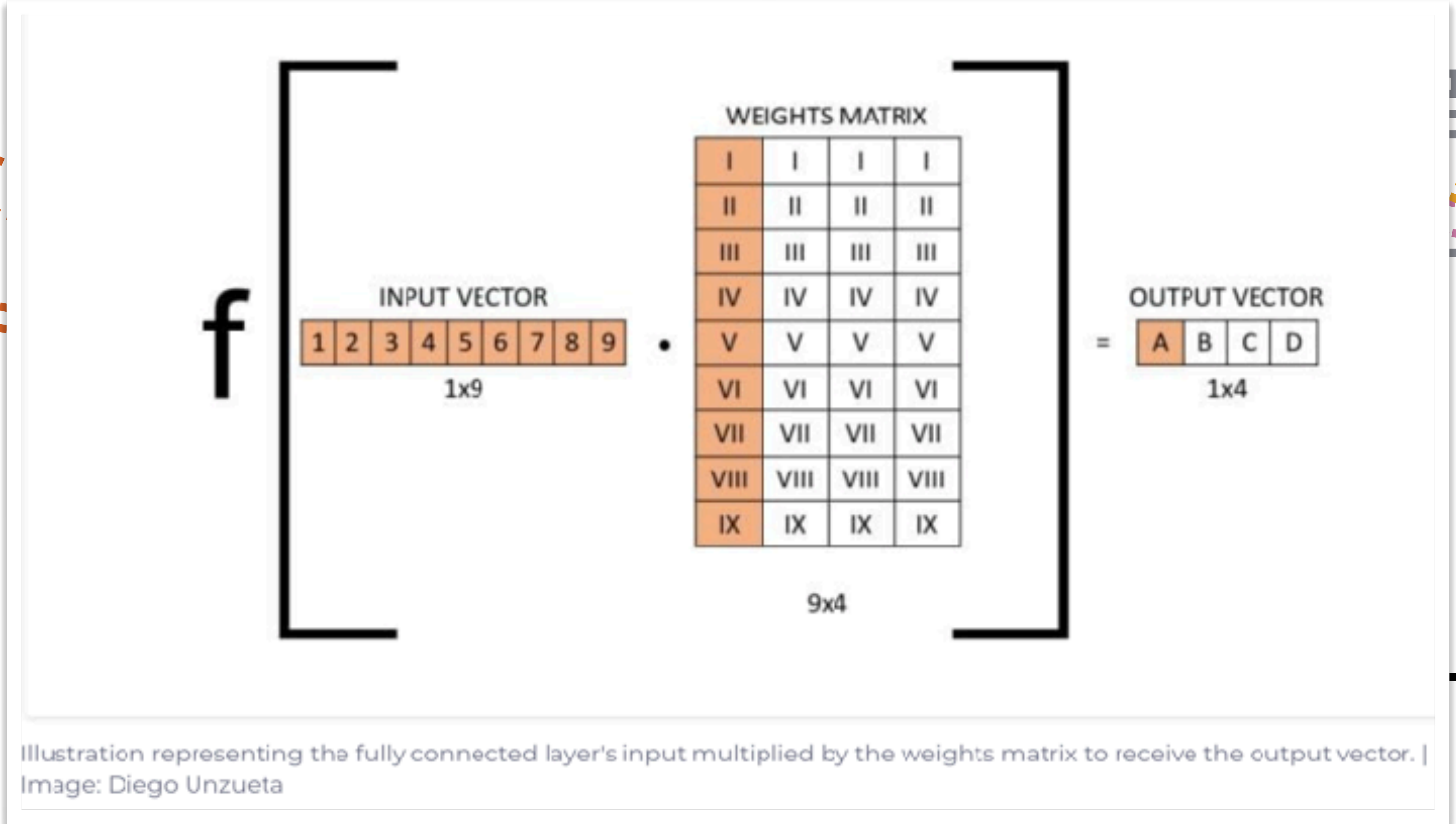
Pooling

Convolution + ReLU

Pooling

Fully Connected

Output Prediction



f [INPUT VECTOR

1	2	3	4	5	6	7	8	9

1x9

·

WEIGHTS MATRIX

I	I	I	I
II	II	II	II
III	III	III	III
IV	IV	IV	IV
V	V	V	V
VI	VI	VI	VI
VII	VII	VII	VII
VIII	VIII	VIII	VIII
IX	IX	IX	IX

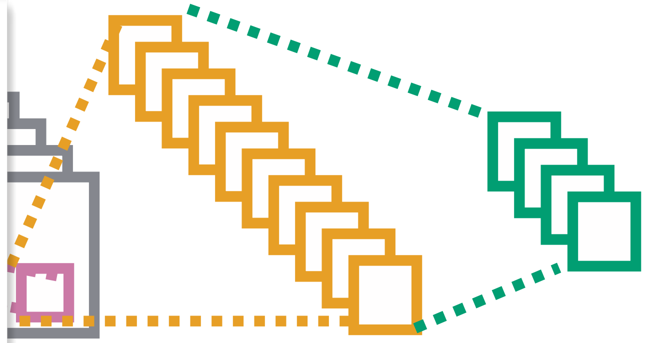
9x4]

= OUTPUT VECTOR

A	B	C	D

1x4

Illustration representing the fully connected layer's input multiplied by the weights matrix to receive the output vector. | Image: Diego Unzueta



Triton (1

Cat (0)

Human (0)

Tiger (0)

https://www.pollev.com/hungweitseng close in 1:30

Locality of data

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
        result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

A. Access of `matrix` has temporal locality, `vector` has spatial locality

B. Both `matrix` and `vector` have temporal locality, and `vector` also has spatial locality

C. Access of `matrix` has spatial locality, `vector` has temporal locality

D. Both `matrix` and `vector` have spatial locality and temporal locality

E. Both `matrix` and `vector` have spatial locality, and `vector` also has temporal locality

Data locality

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
        result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

A. Access of `matrix` has temporal locality, `vector` has spatial locality

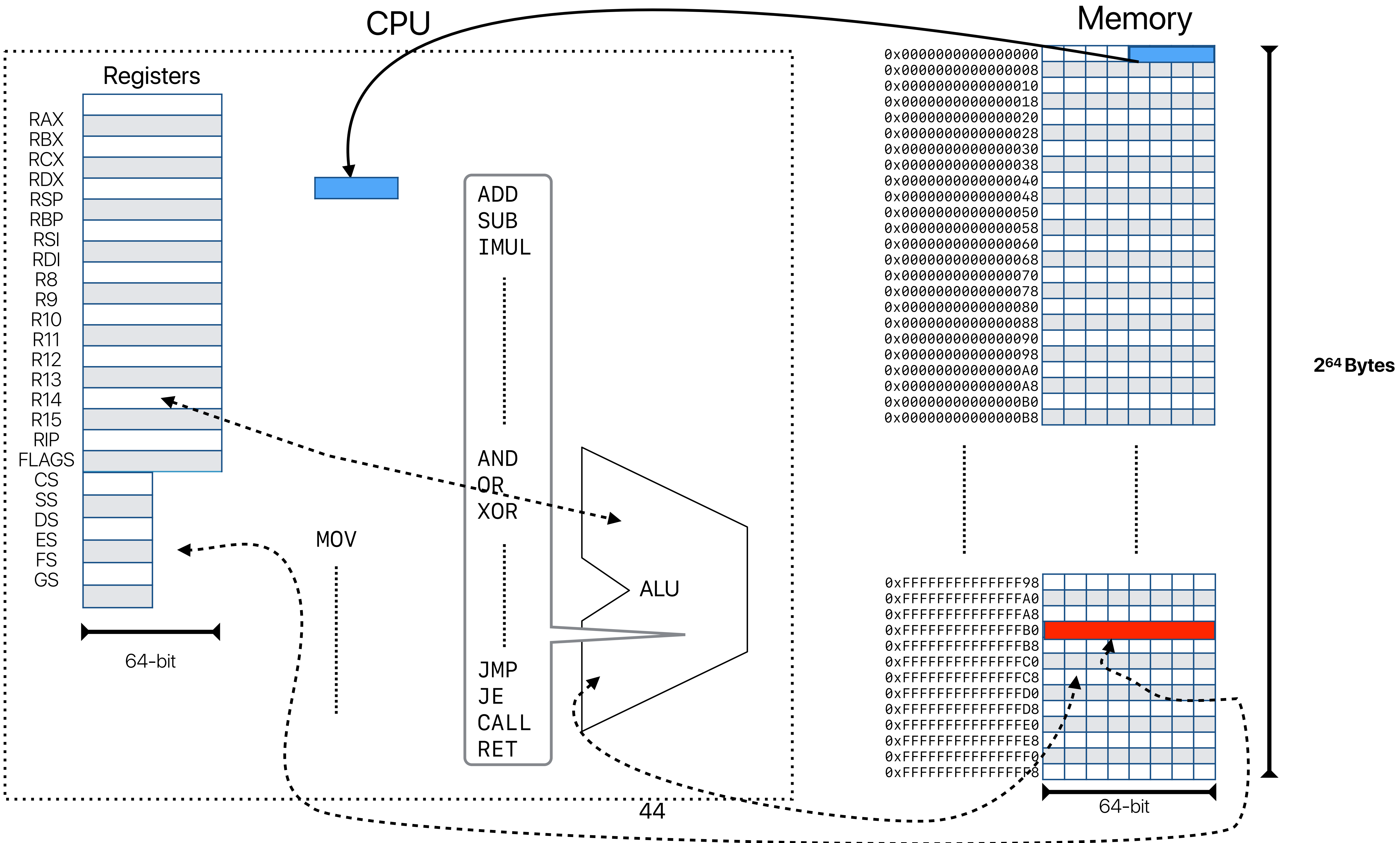
B. Both `matrix` and `vector` have temporal locality, and `vector` also has spatial locality

C. Access of `matrix` has spatial locality, `vector` has temporal locality

D. Both `matrix` and `vector` have spatial locality and temporal locality

E. Both `matrix` and `vector` have spatial locality, and `vector` also has temporal locality

x86 ISA: the abstracted machine



CPU

Registers

RAX
RBX
RCX
RDX
RSP
RBP
RSI
RDI
R8
R9
R10
R11
R12
R13
R14
R15
RIP
FLAGS
CS
SS
DS
ES
FS
GS

64-bit

MOV

ADD
SUB
IMUL

AND
OR
XOR

JMP
JE
CALL
RET

ALU

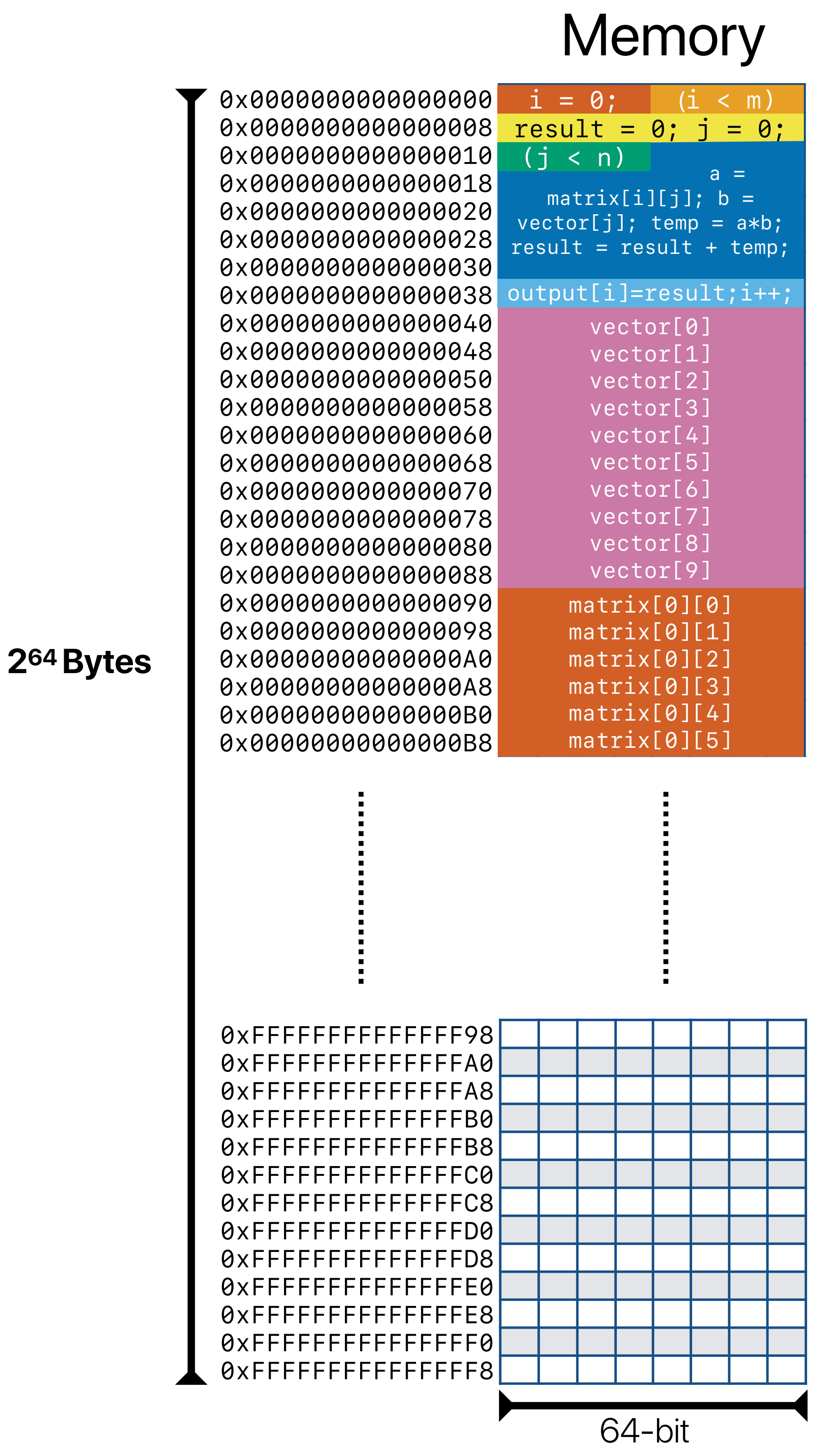
Memory

```
0x0000000000000000
0x0000000000000008
0x0000000000000010
0x0000000000000018
0x0000000000000020
0x0000000000000028
0x0000000000000030
0x0000000000000038
0x0000000000000040
0x0000000000000048
0x0000000000000050
0x0000000000000058
0x0000000000000060
0x0000000000000068
0x0000000000000070
0x0000000000000078
0x0000000000000080
0x0000000000000088
0x0000000000000090
0x0000000000000098
0x00000000000000A0
0x00000000000000A8
0x00000000000000B0
0x00000000000000B8

0xFFFFFFFFFFFFFF98
0xFFFFFFFFFFFFFFA0
0xFFFFFFFFFFFFFFA8
0xFFFFFFFFFFFFFFB0
0xFFFFFFFFFFFFFFB8
0xFFFFFFFFFFFFFFC0
0xFFFFFFFFFFFFFFC8
0xFFFFFFFFFFFFFFD0
0xFFFFFFFFFFFFFFD8
0xFFFFFFFFFFFFFFE0
0xFFFFFFFFFFFFFFE8
0xFFFFFFFFFFFFFFF0
0xFFFFFFFFFFFFFFF8
```

64-bit

2^{64} Bytes

Data/instructions in the memory

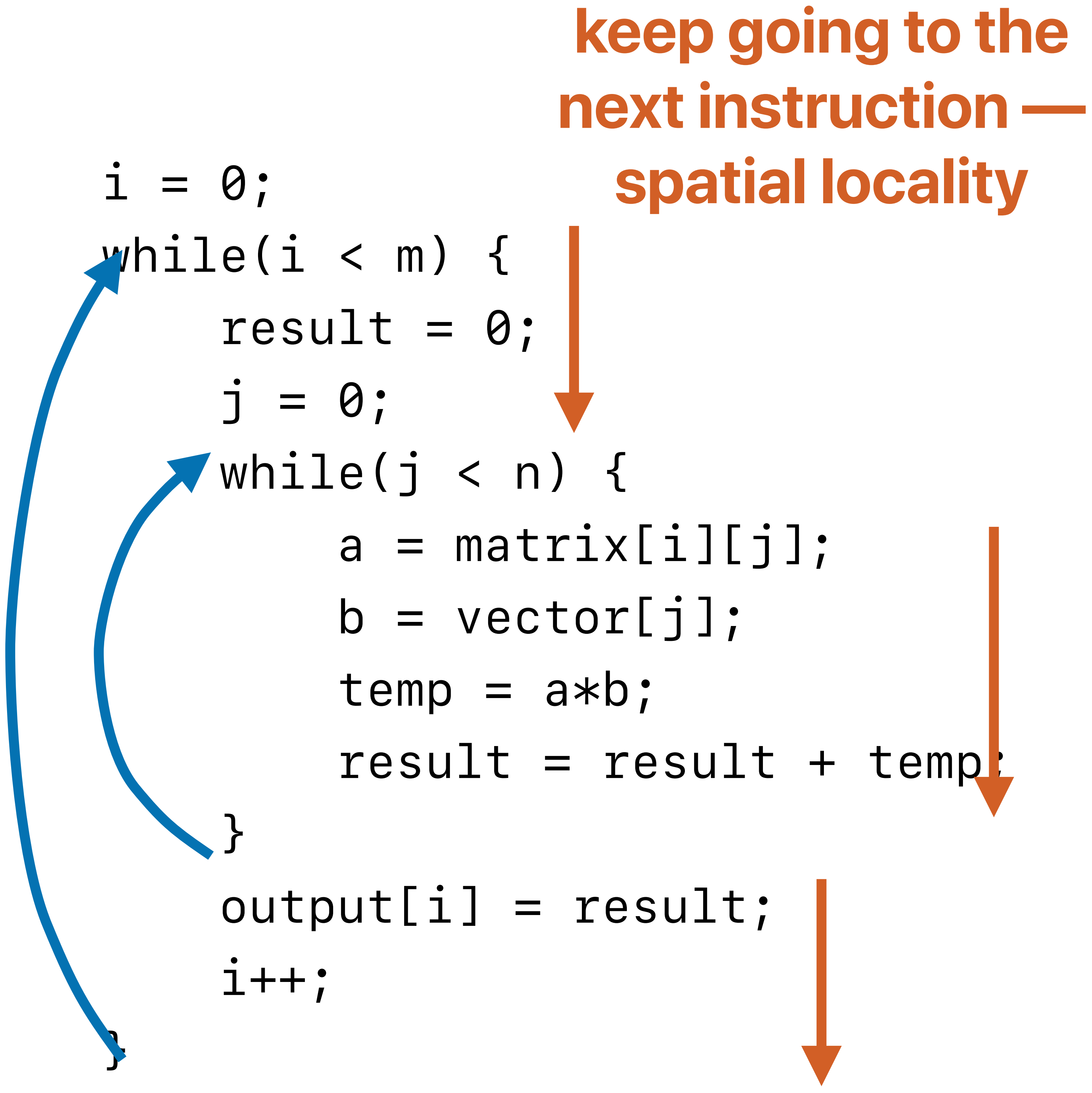


Memory

0x0000000000000000	i = 0; (i < m)
0x0000000000000008	result = 0; j = 0;
0x0000000000000010	(j < n) a =
0x0000000000000018	matrix[i][j]; b =
0x0000000000000020	vector[j]; temp = a*b;
0x0000000000000028	result = result + temp;
0x0000000000000030	
0x0000000000000038	output[i]=result;i++;
0x0000000000000040	vector[0]
0x0000000000000048	vector[1]
0x0000000000000050	vector[2]
0x0000000000000058	vector[3]
0x0000000000000060	vector[4]
0x0000000000000068	vector[5]
0x0000000000000070	vector[6]
0x0000000000000078	vector[7]
0x0000000000000080	vector[8]
0x0000000000000088	vector[9]
0x0000000000000090	matrix[0][0]
0x0000000000000098	matrix[0][1]
0x00000000000000A0	matrix[0][2]
0x00000000000000A8	matrix[0][3]
0x00000000000000B0	matrix[0][4]
0x00000000000000B8	matrix[0][5]
0xFFFFFFFFFFFFFF98	
0xFFFFFFFFFFFFFFA0	
0xFFFFFFFFFFFFFFA8	
0xFFFFFFFFFFFFFFB0	
0xFFFFFFFFFFFFFFB8	
0xFFFFFFFFFFFFFFC0	
0xFFFFFFFFFFFFFFC8	
0xFFFFFFFFFFFFFFD0	
0xFFFFFFFFFFFFFFD8	
0xFFFFFFFFFFFFFFE0	
0xFFFFFFFFFFFFFFE8	
0xFFFFFFFFFFFFFFF0	
0xFFFFFFFFFFFFFFF8	

Code also has locality

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
        result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```



```
i = 0;
while(i < m) {
    result = 0;
    j = 0;
    while(j < n) {
        a = matrix[i][j];
        b = vector[j];
        temp = a*b;
        result = result + temp;
    }
    output[i] = result;
    i++;
}
```

keep going to the next instruction — spatial locality

repeat many times — temporal locality!

Data locality

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
        result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

spatial locality:
matrix[0][0], matrix[0][1], matrix[0][2], ...
vector[0], vector[1], ... , vector[n]
temporal locality:
reuse of vector[0], vector[1], ... ,

A. Access of `matrix` has temporal locality, `vector` has spatial locality
B. Both `matrix` and `vector` have temporal locality, and `vector` also has spatial locality
C. Access of `matrix` has spatial locality, `vector` has temporal locality
D. Both `matrix` and `vector` have spatial locality and temporal locality
[✓ E. Both `matrix` and `vector` have spatial locality, and `vector` also has temporal locality]

Locality

- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then PC + 4
  - Data — the current element in an array, then the next
- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
  - Data — the same data can be read/write many times

Locality

- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then the next instruction
  - Data — the current element in an array, then the next
- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
    - Typically tens of static instructions — at most several KBs
  - Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)

Most of time, your program is just visiting a very small amount of data/instructions within a given window

Take-aways: inside out our memory hierarchy

- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!
- Hierarchical caching with small amount of SRAMs will work if we can efficiently capture data and instructions
- Caching is possible! Most of time, we only work on a small amount of data!
  - Spatial locality
  - Temporal locality

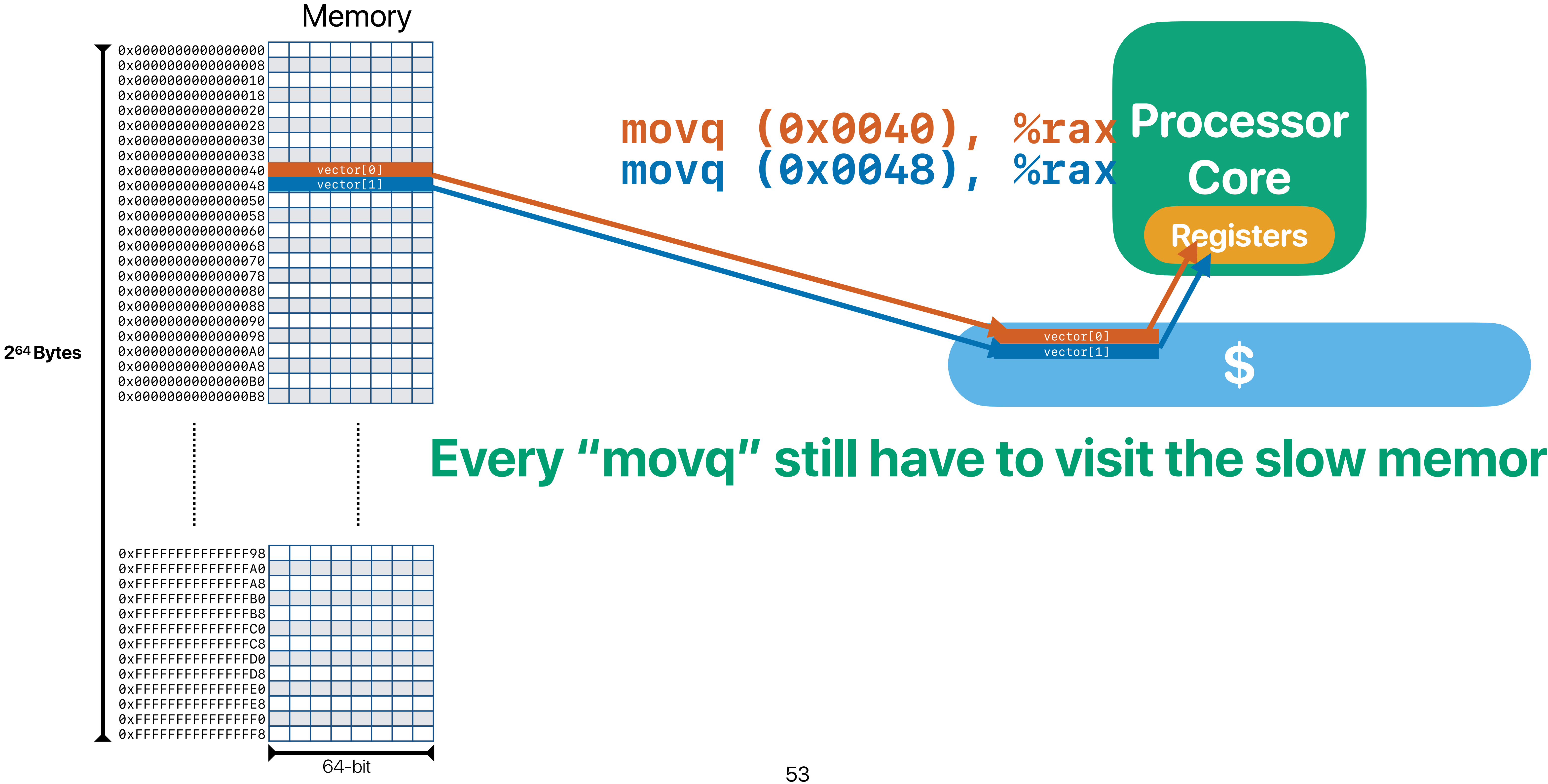
Designing a hardware to exploit locality

- Spatial locality — application tends to visit nearby stuffs in the memory

We need to "cache consecutive memory locations" every time — the cache should store a "block" of code/data

- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
    - Typically tens of static instructions — at most several KBs
  - Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)

If we just cache what has requested

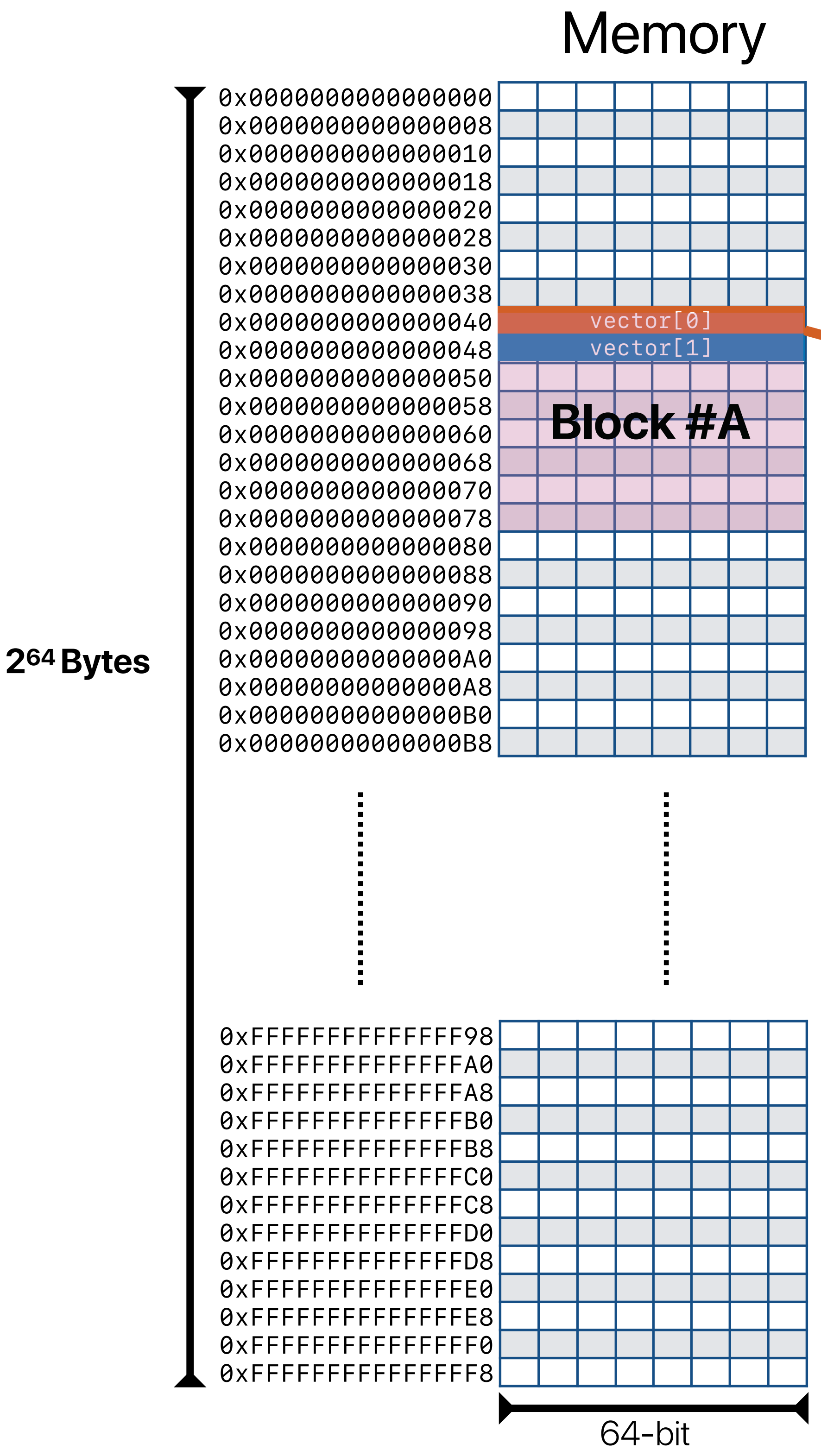


```
0x0000000000000000
0x0000000000000008
0x0000000000000010
0x0000000000000018
0x0000000000000020
0x0000000000000028
0x0000000000000030
0x0000000000000038
0x0000000000000040
0x0000000000000048
0x0000000000000050
0x0000000000000058
0x0000000000000060
0x0000000000000068
0x0000000000000070
0x0000000000000078
0x0000000000000080
0x0000000000000088
0x0000000000000090
0x0000000000000098
0x00000000000000A0
0x00000000000000A8
0x00000000000000B0
0x00000000000000B8

0xFFFFFFFFFFFFFF98
0xFFFFFFFFFFFFFFA0
0xFFFFFFFFFFFFFFA8
0xFFFFFFFFFFFFFFB0
0xFFFFFFFFFFFFFFB8
0xFFFFFFFFFFFFFFC0
0xFFFFFFFFFFFFFFC8
0xFFFFFFFFFFFFFFD0
0xFFFFFFFFFFFFFFD8
0xFFFFFFFFFFFFFFE0
0xFFFFFFFFFFFFFFE8
0xFFFFFFFFFFFFFFF0
0xFFFFFFFFFFFFFFF8
```

```
movq (0x0040), %rax
movq (0x0048), %rax
```

Every “movq” still have to visit the slow memory!

If we can cache a block of data



Memory

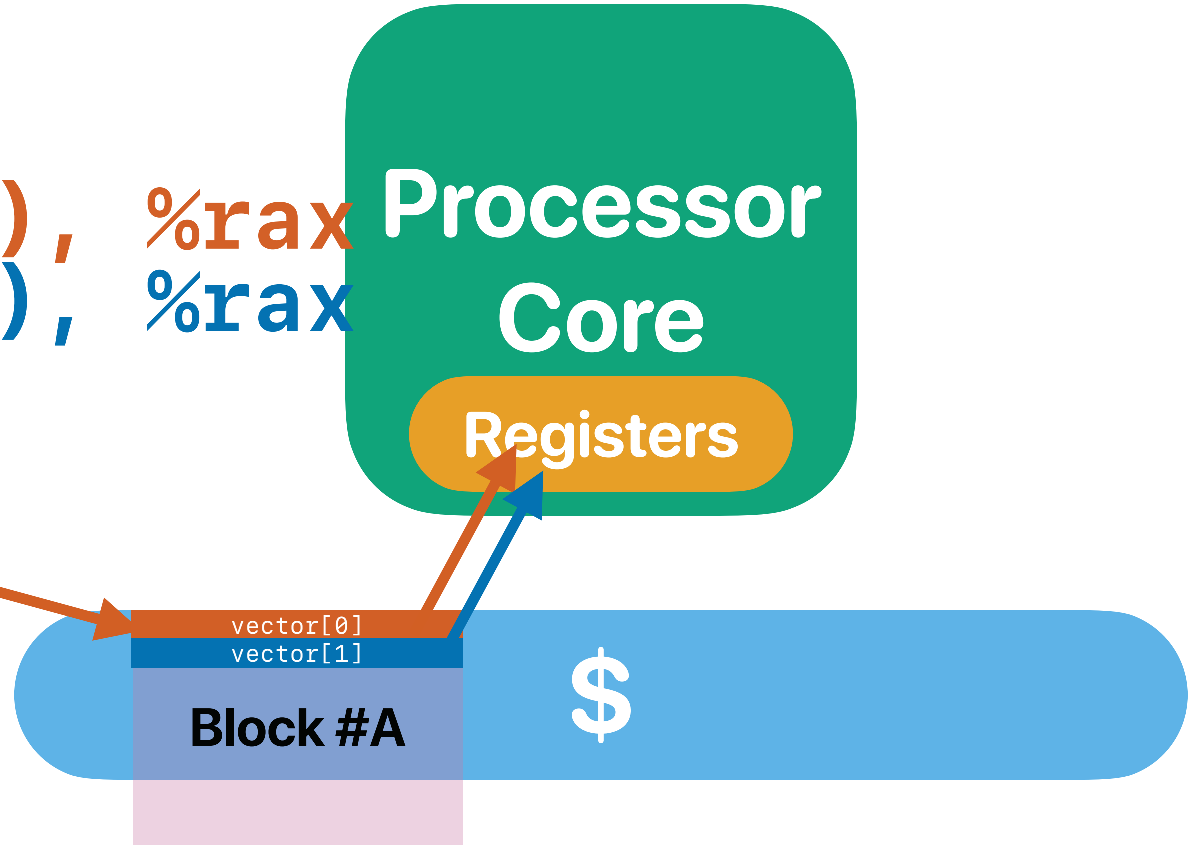
0x0000000000000000
0x0000000000000008
0x0000000000000010
0x0000000000000018
0x0000000000000020
0x0000000000000028
0x0000000000000030
0x0000000000000038
0x0000000000000040 vector[0]
0x0000000000000048 vector[1]
0x0000000000000050
0x0000000000000058
0x0000000000000060 Block #A
0x0000000000000068
0x0000000000000070
0x0000000000000078
0x0000000000000080
0x0000000000000088
0x0000000000000090
0x0000000000000098
0x00000000000000A0
0x00000000000000A8
0x00000000000000B0
0x00000000000000B8

0xFFFFFFFFFFFFFF98
0xFFFFFFFFFFFFFFA0
0xFFFFFFFFFFFFFFA8
0xFFFFFFFFFFFFFFB0
0xFFFFFFFFFFFFFFB8
0xFFFFFFFFFFFFFFC0
0xFFFFFFFFFFFFFFC8
0xFFFFFFFFFFFFFFD0
0xFFFFFFFFFFFFFFD8
0xFFFFFFFFFFFFFFE0
0xFFFFFFFFFFFFFFE8
0xFFFFFFFFFFFFFFF0
0xFFFFFFFFFFFFFFF8

2^{64} Bytes

64-bit

```
movq (0x0040), %rax
movq (0x0048), %rax
```



Processor Core

Registers

vector[0]
vector[1]
Block #A

$

We can now serve vector[1] from the cache!

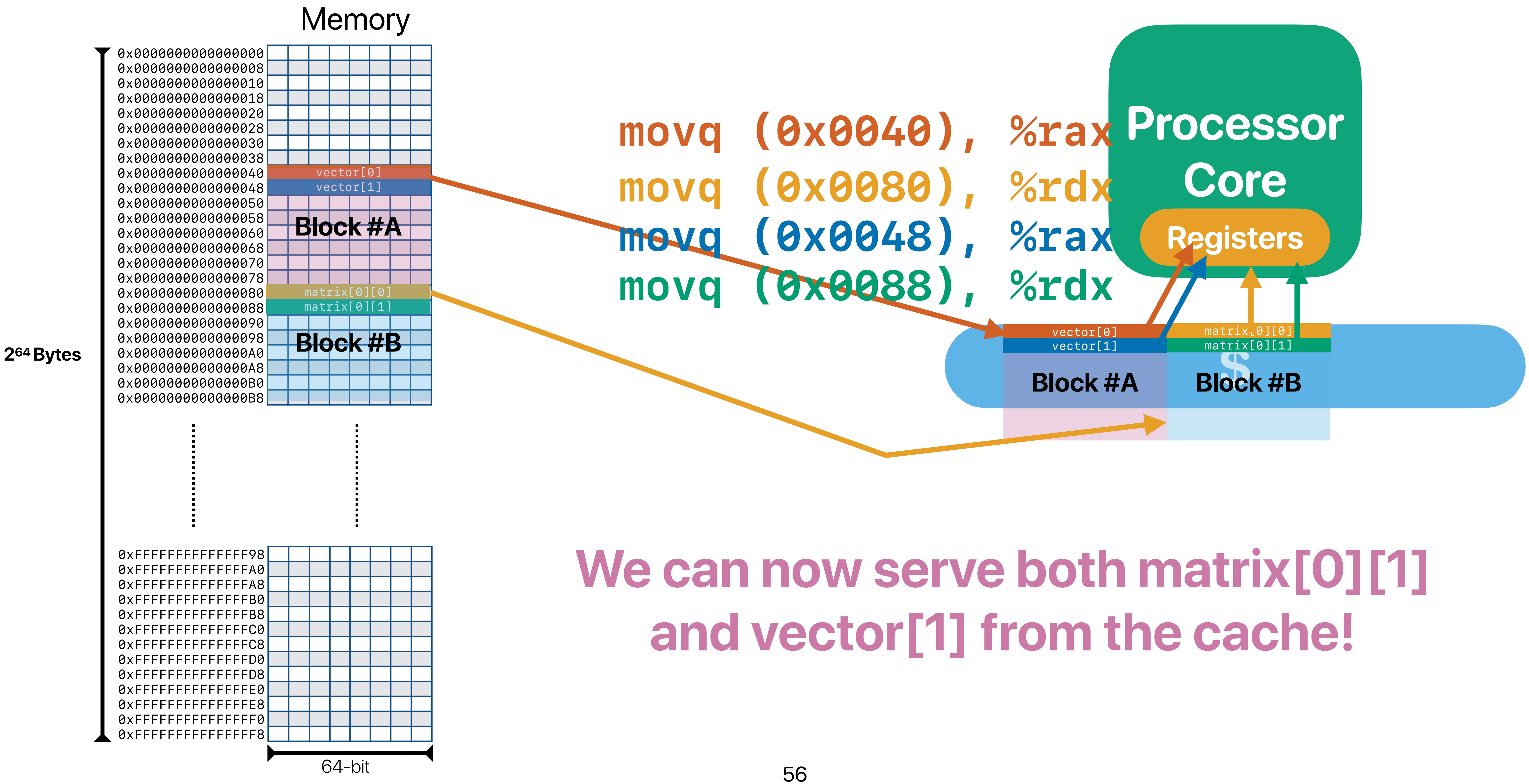
Recap: Locality

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint64_t i = 0; i < m; i++) {
  result = 0;
  for(uint64_t j = 0; j < n; j++) {
    result += matrix[i][j]*vector[j];
  }
  output[i] = result;
}
```

Simply caching one block isn't enough

We need to cache multiple blocks of data!



```
movq (0x0040), %rax
movq (0x0080), %rdx
movq (0x0048), %rax
movq (0x0088), %rdx
```

We can now serve both matrix[0][1] and vector[1] from the cache!

Designing a hardware to exploit locality

- Spatial locality — application tends to visit nearby stuffs in the memory
    - Code — the current instruction, and then the next instruction

We need to "cache consecutive memory locations" every time

— the cache should store a "block" of code/data

- Temporal locality — application revisit the same thing again and again
    - Code — loops, frequently invoked functions
        - Typically tens of static instructions — at most several KBs
    - Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)

We need to "cache frequently used memory blocks"

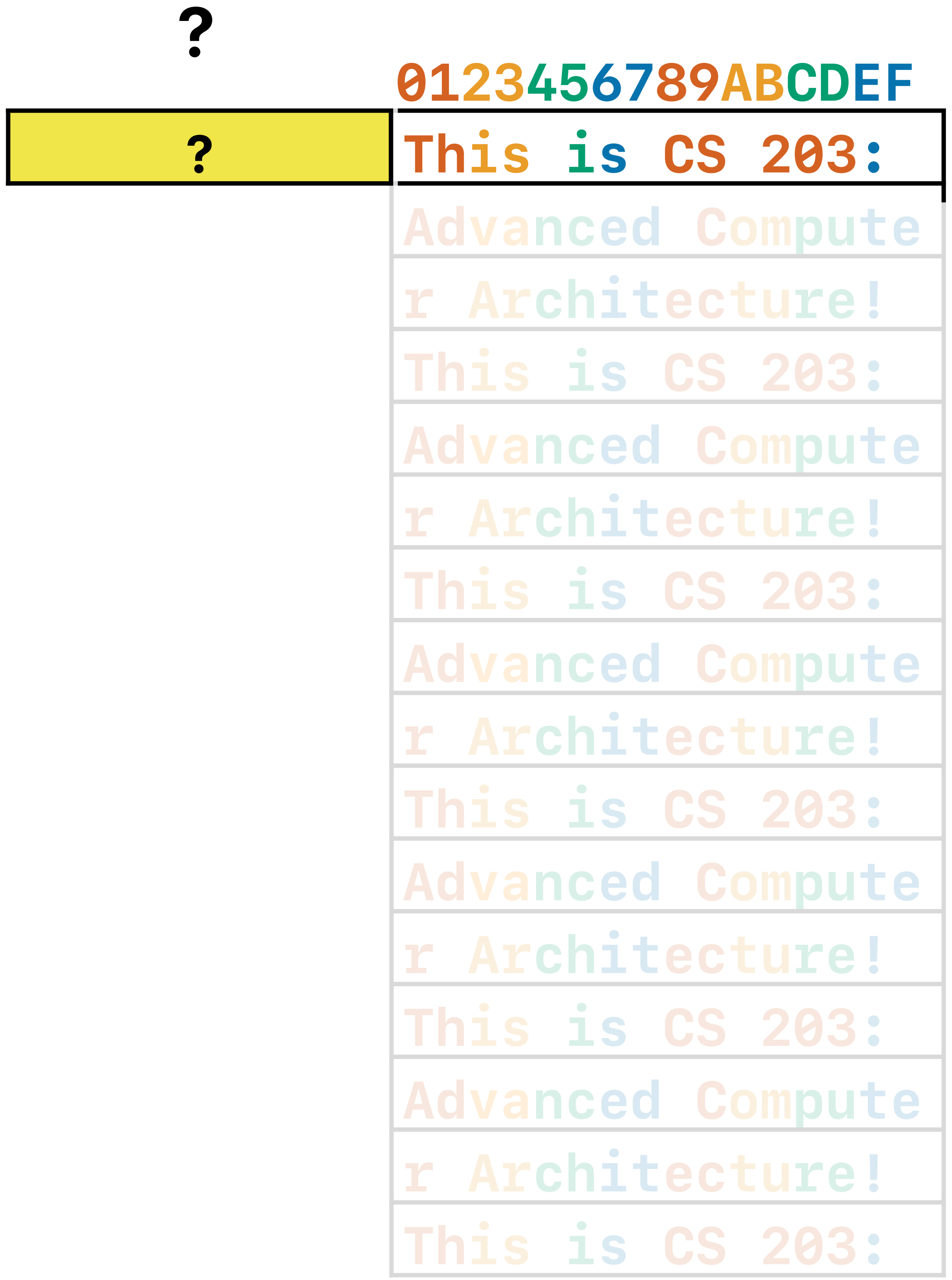
— the cache should store a few blocks

— the cache must be able to distinguish blocks

Processor Core

Registers

How to tell who is there?



?

0123456789ABCDEF

?

This is CS 203:

Advanced Compute

r Architecture!

This is CS 203:

Advanced Compute

r Architecture!

This is CS 203:

Advanced Compute

r Architecture!

This is CS 203:

Advanced Compute

r Architecture!

This is CS 203:

Advanced Compute

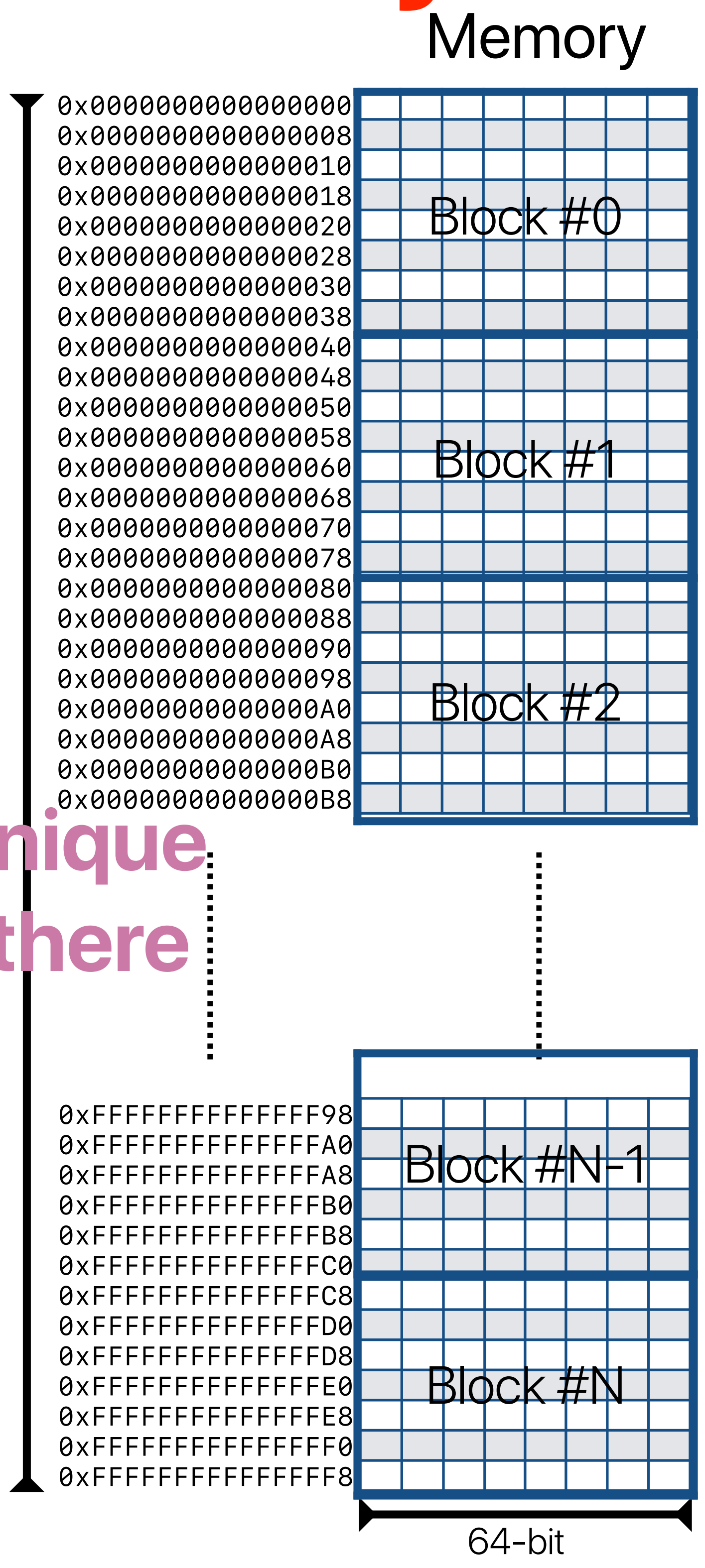
r Architecture!

This is CS 203:

Partition memory addresses into fix-sized chunks

```
0b00000000000000000
0b00000000000001000
0b00000000000010000
0b00000000000011000
0b00000000000100000
0b00000000000101000
0b00000000001000000
0b00000000001001000
0b00000000010000000
0b00000000010001000
0b00000000010010000
0b00000000010011000
0b00000000000100000
0b00000000000101000
0b00000000001000000
0b00000000001001000
```

each block has a unique common prefix in there addresses!



```
0x0000000000000000
0x0000000000000008
0x0000000000000010
0x0000000000000018
0x0000000000000020
0x0000000000000028
0x0000000000000030
0x0000000000000038
0x0000000000000040
0x0000000000000048
0x0000000000000050
0x0000000000000058
0x0000000000000060
0x0000000000000068
0x0000000000000070
0x0000000000000078
0x0000000000000080
0x0000000000000088
0x0000000000000090
0x0000000000000098
0x00000000000000A0
0x00000000000000A8
0x00000000000000B0
0x00000000000000B8

0xFFFFFFFFFFFFFF98
0xFFFFFFFFFFFFFFA0
0xFFFFFFFFFFFFFFA8
0xFFFFFFFFFFFFFFB0
0xFFFFFFFFFFFFFFB8
0xFFFFFFFFFFFFFFC0
0xFFFFFFFFFFFFFFC8
0xFFFFFFFFFFFFFFD0
0xFFFFFFFFFFFFFFD8
0xFFFFFFFFFFFFFFE0
0xFFFFFFFFFFFFFFE8
0xFFFFFFFFFFFFFFF0
0xFFFFFFFFFFFFFFF8
```

Processor Core
Registers
$

Processor Core

Registers

How to tell who is there?

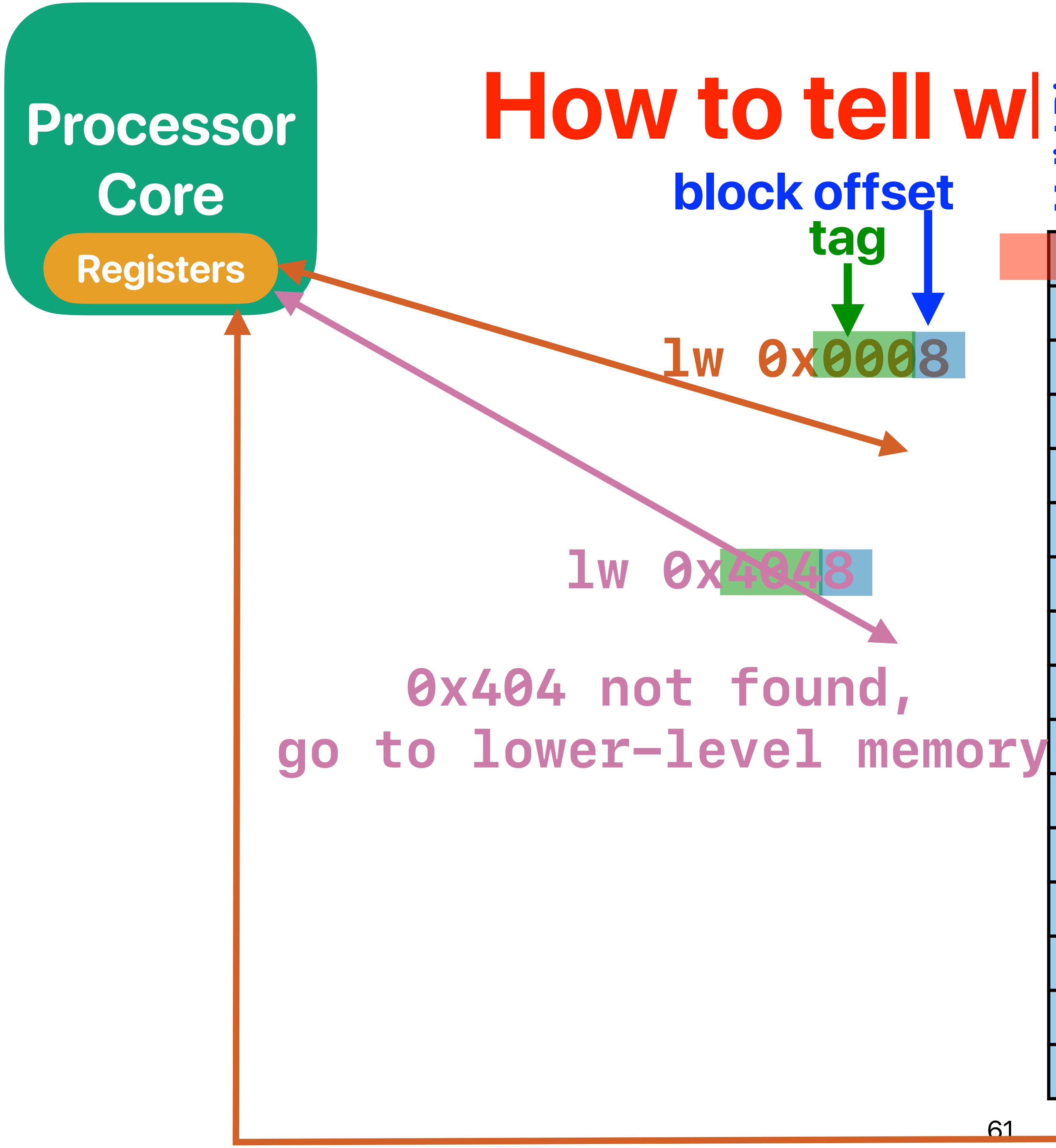
the common address prefix in each block →

0x0000 0x0001 0x0002 0x0003 0x0004 0x0005 0x0006 0x0007 0x0008 0x0009 0x000A 0x000B 0x000C 0x000D 0x000E 0x000F

0123456789ABCDEF

tag array	
0x000	This is CS 203:
0x001	Advanced Compute
0xF07	r Architecture!
0x100	This is CS 203:
0x310	Advanced Compute
0x450	r Architecture!
0x006	This is CS 203:
0x537	Advanced Compute
0x266	r Architecture!
0x307	This is CS 203:
0x265	Advanced Compute
0x80A	r Architecture!
0x620	This is CS 203:
0x630	Advanced Compute
0x705	r Architecture!
0x216	This is CS 203:

How to tell w



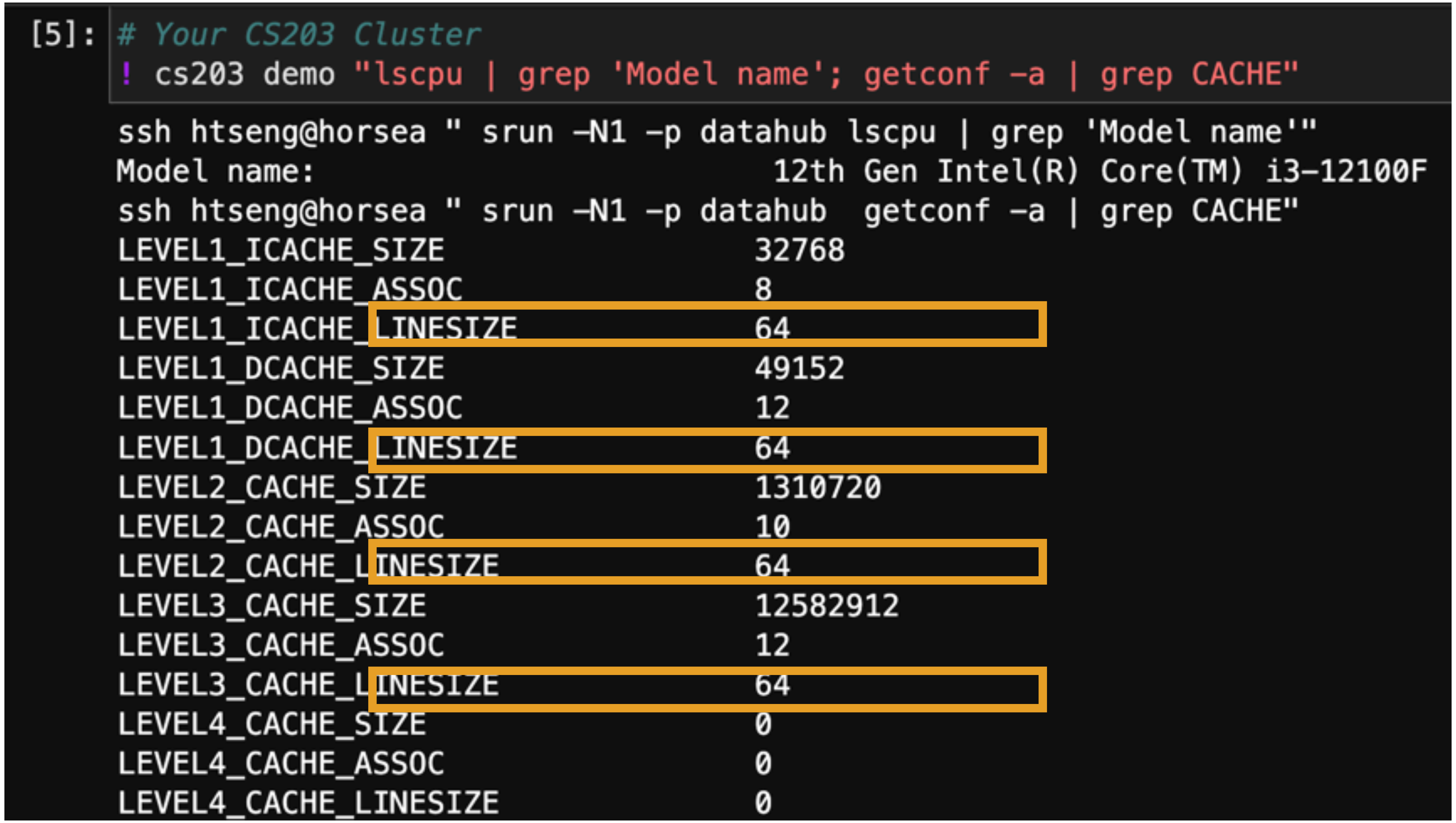
0x404 not found,
go to lower-level memory

Tell if the block here can be used

Tell if the block here is modified

Valid Bit	Dirty Bit	tag	data 0123456789ABCDEF
1	1	0x000	This is CSE1 3:
1	1	0x001	Advanced Compute
1	0	0xF07	r Architecture!
0	1	0x100	This is CS 203:
1	1	0x310	Advanced Compute
1	1	0x450	r Architecture!
0	1	0x006	This is CS 203:
0	1	0x537	Advanced Compute
1	1	0x266	r Architecture!
1	1	0x307	This is CS 203:
0	1	0x265	Advanced Compute
0	1	0x80A	r Architecture!
1	1	0x620	This is CS 203:
1	1	0x630	Advanced Compute
1	0	0x705	r Architecture!
0	1	0x216	This is CS 203:

Blocksize == Linesize



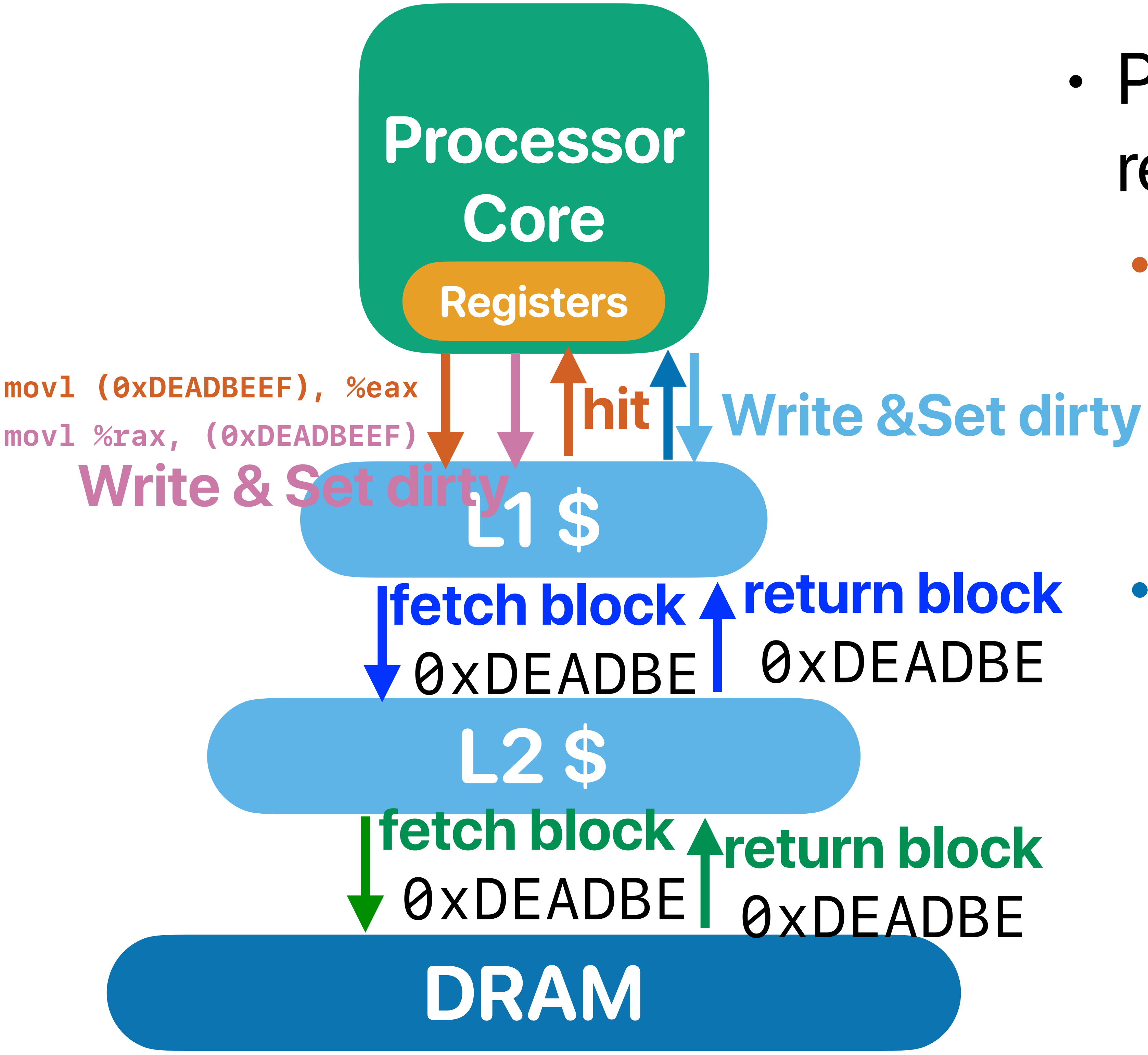
```
[5]: # Your CS203 Cluster
     ! cs203 demo "lscpu | grep 'Model name'; getconf -a | grep CACHE"

     ssh htseng@horsea " srun -N1 -p datahub lscpu | grep 'Model name'"
     Model name:                        12th Gen Intel(R) Core(TM) i3-12100F
     ssh htseng@horsea " srun -N1 -p datahub  getconf -a | grep CACHE"
     LEVEL1_ICACHE_SIZE                32768
     LEVEL1_ICACHE_ASSOC               8
     LEVEL1_ICACHE_LINESIZE            64
     LEVEL1_DCACHE_SIZE                49152
     LEVEL1_DCACHE_ASSOC               12
     LEVEL1_DCACHE_LINESIZE            64
     LEVEL2_CACHE_SIZE                 1310720
     LEVEL2_CACHE_ASSOC                10
     LEVEL2_CACHE_LINESIZE             64
     LEVEL3_CACHE_SIZE                 12582912
     LEVEL3_CACHE_ASSOC                12
     LEVEL3_CACHE_LINESIZE             64
     LEVEL4_CACHE_SIZE                 0
     LEVEL4_CACHE_ASSOC                0
     LEVEL4_CACHE_LINESIZE             0
```

Take-aways: inside out our memory hierarchy

- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!
- Hierarchical caching with small amount of SRAMs will work if we can efficiently capture data and instructions
- Caching is possible! Most of time, we only work on a small amount of data!
  - Spatial locality
  - Temporal locality
- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks

Put everything all together:
How cache interacts with CPU

Processor/cache interaction



Processor Core

Registers

```
movl (0xDEADBEEF), %eax
movl %rax, (0xDEADBEEF)
```

Write & Set dirty

hit

Write &Set dirty

L1 $

fetch block 0xDEADBE

return block 0xDEADBE

L2 $

fetch block 0xDEADBE

return block 0xDEADBE

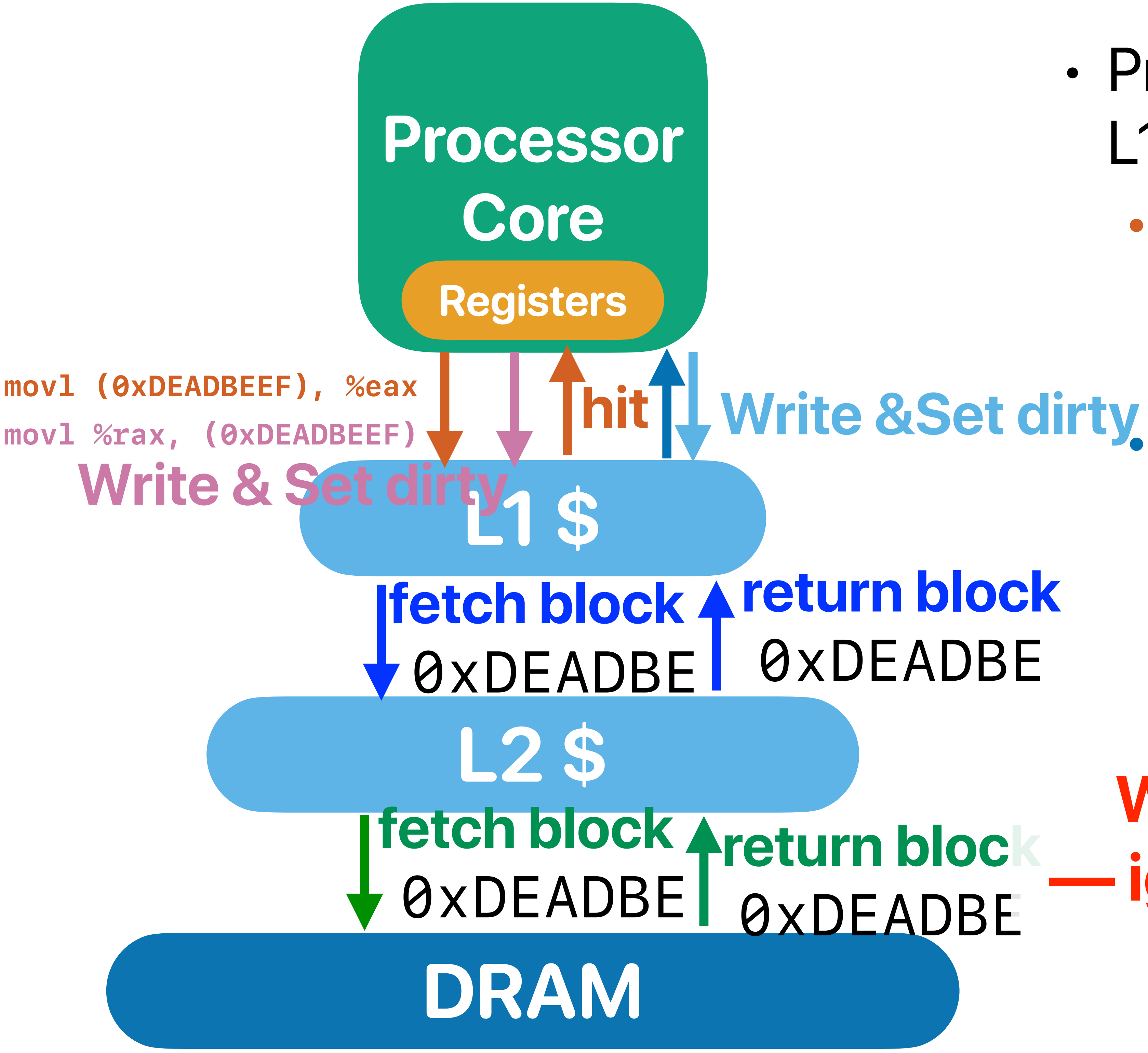
DRAM

- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY** **Why don't we write to L2?**

      — Too slow
  - **if miss**
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache
    - **Present the write "ONLY" in L1 and set DIRTY**

What if we run out of $ blocks?

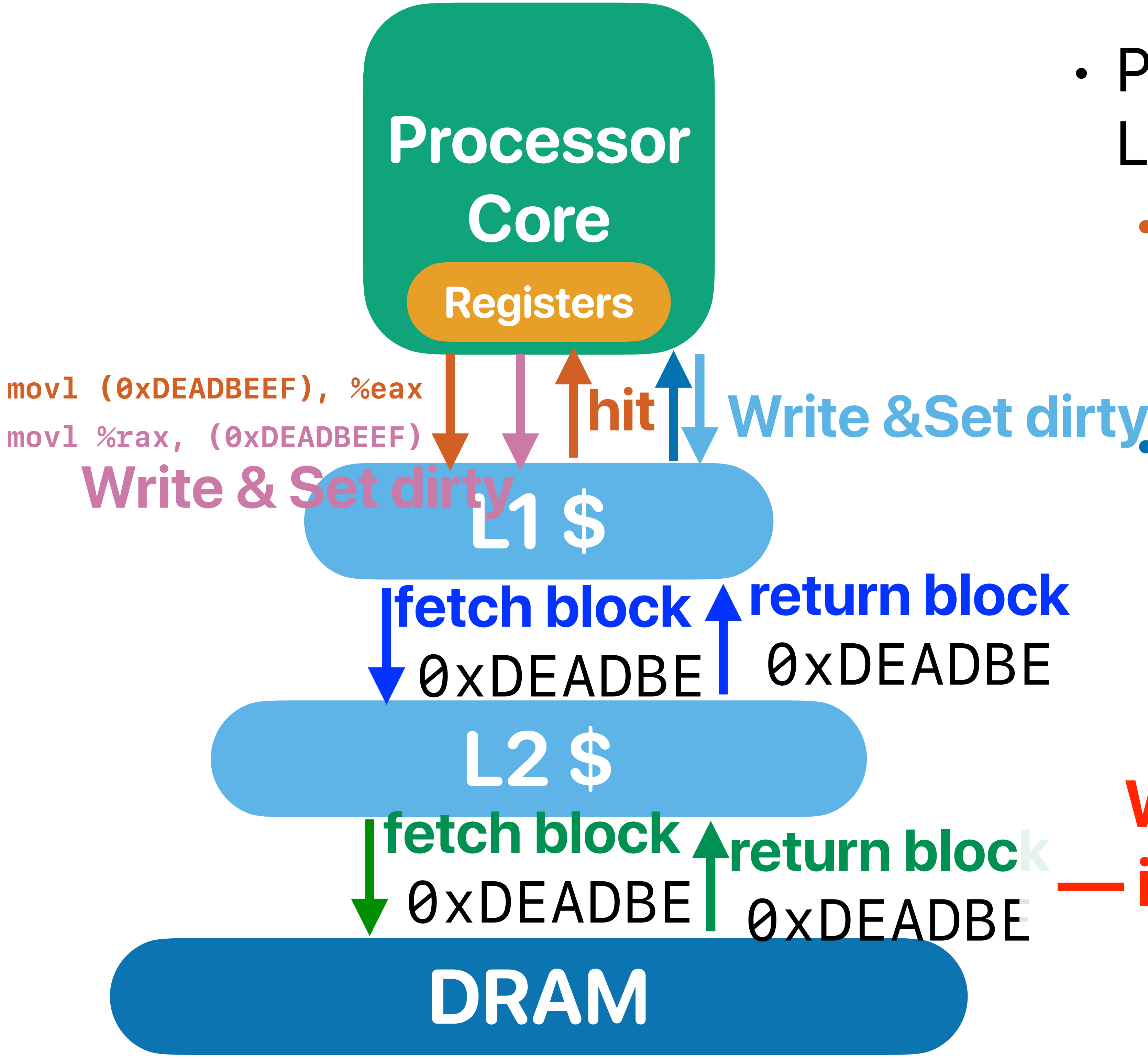
Processor/cache interaction



- Processor sends memory access request to L1-$
  - if hit & it's a read
    - Read: return data
    - Write: Update "ONLY" in L1 and set DIRTY
  - if miss
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy
    - Fetch the requesting block from lower level memory hierarchy and place in the cache
    - Present the write "ONLY" in L1 and set DIRTY

What if the victim block is modified?
— ignoring the update is not acceptable!

Processor/cache interaction



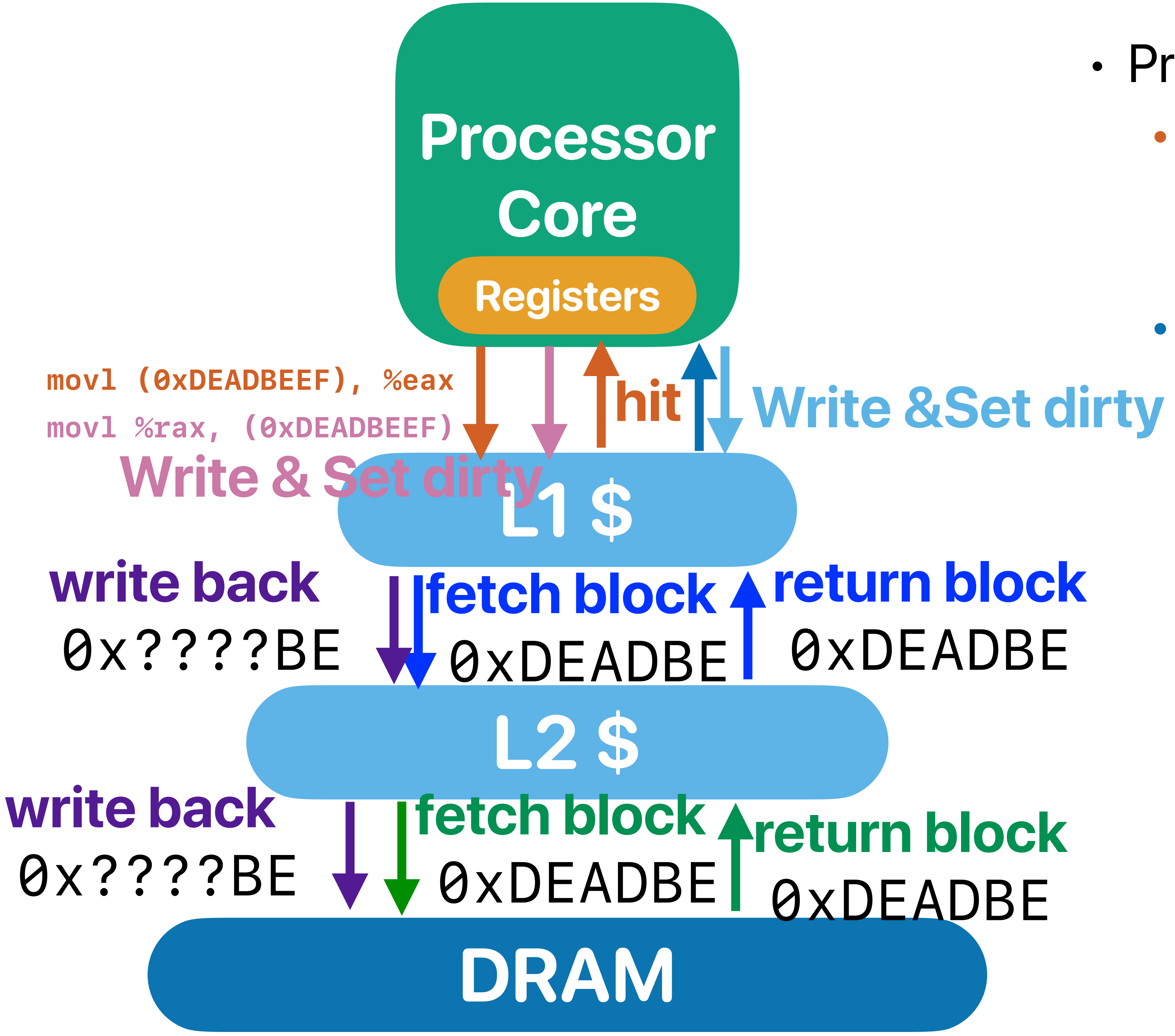
```
movl (0xDEADBEEF), %eax
movl %rax, (0xDEADBEEF)
```

- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY**
  - **if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy
    - Fetch the requesting block from lower level memory hierarchy and place in the cache
    - **Present the write "ONLY" in L1 and set DIRTY**

What if the victim block is modified?

— ignoring the update is not acceptable!

Processor/cache interaction



- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY**
  - **if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
      - If write-back or fetching causes any miss, repeat the same process
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache
    - **Present the write "ONLY" in L1 and set DIRTY**

Take-aways: inside out our memory hierarchy

- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!
- Hierarchical caching with small amount of SRAMs will work if we can efficiently capture data and instructions
- Caching is possible! Most of time, we only work on a small amount of data!
  - Spatial locality
  - Temporal locality
- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use

Announcement

- Reading quiz #3 due next **Tuesday** before the lecture
- Assignment #1 due **tonight**
- Upcoming deadlines
  - Assignment #2 due **in two weeks — 10/24/2024**
    - Already online — please find through the course website
    - Don’t submit the wrong one
  - Programming Assignment #2 due **in 4 weeks — 11/7/2024**
    - Already online — please find through the course website
    - Don’t submit the wrong one
  - Assignment #3 due **in three weeks — 10/31/2024**
    - Will be online next Thursday— please find through the course website
    - Don’t submit the wrong one

Computer
Science &
Engineering

203

つづく