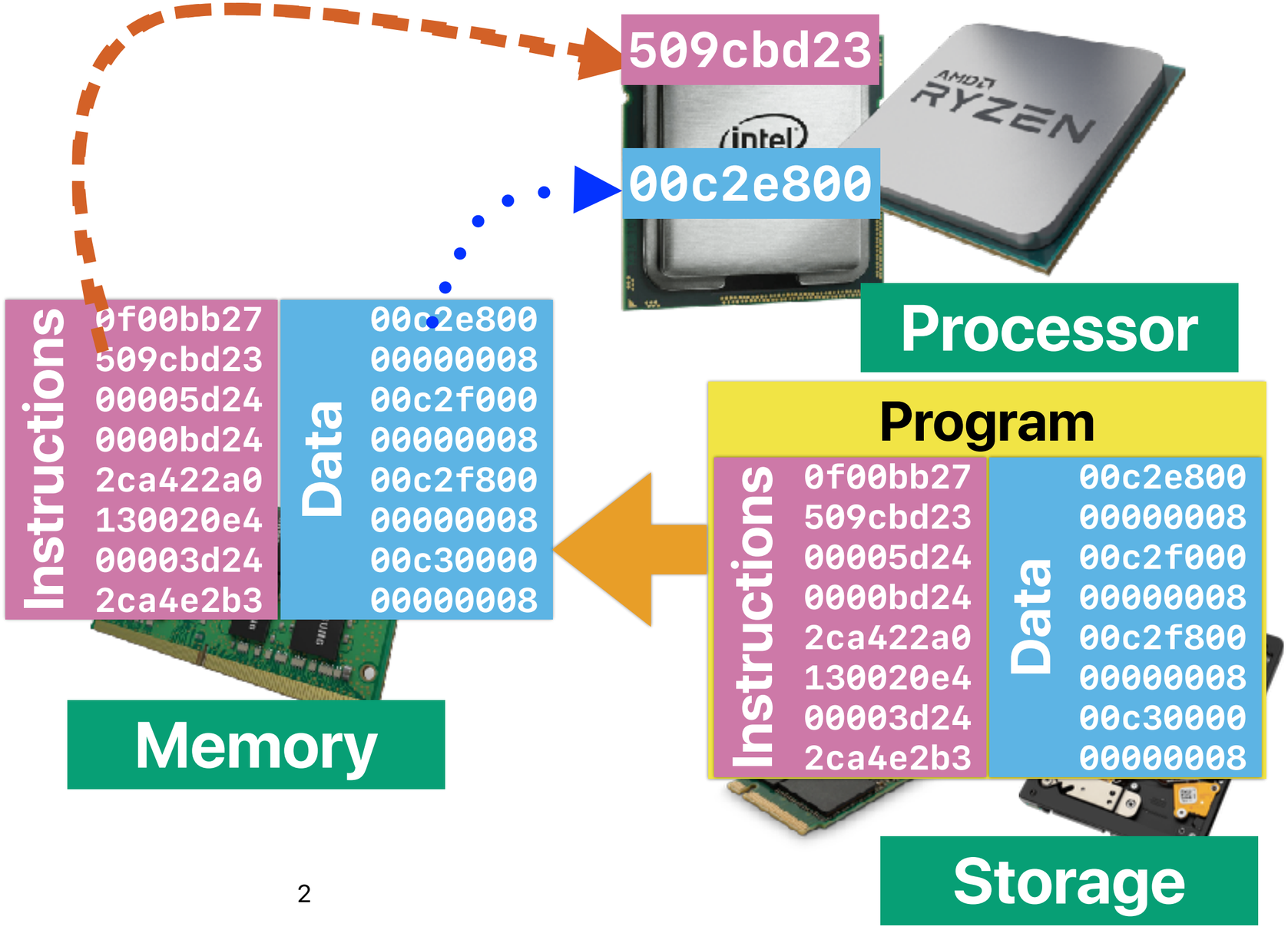
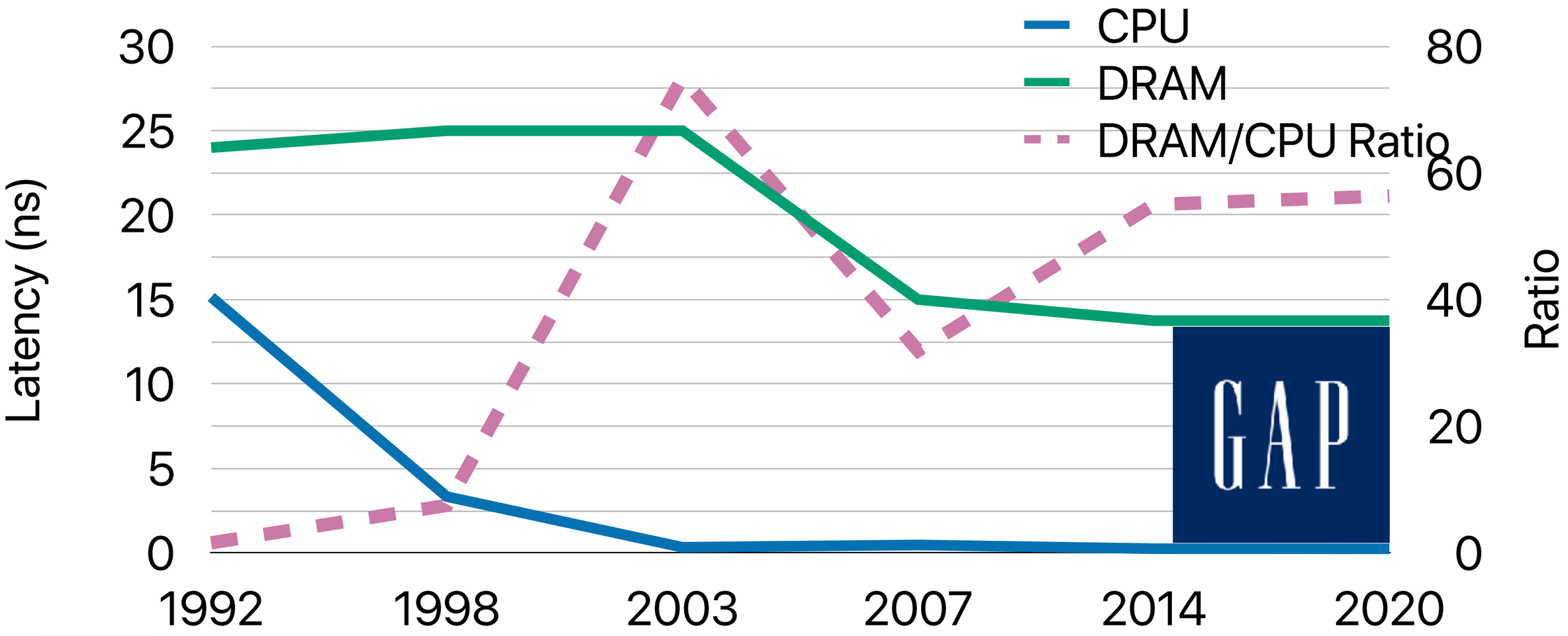
Memory Hierarchy: Basics

Hung-Wei Tseng

Recap: von Neumman Architecture



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

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

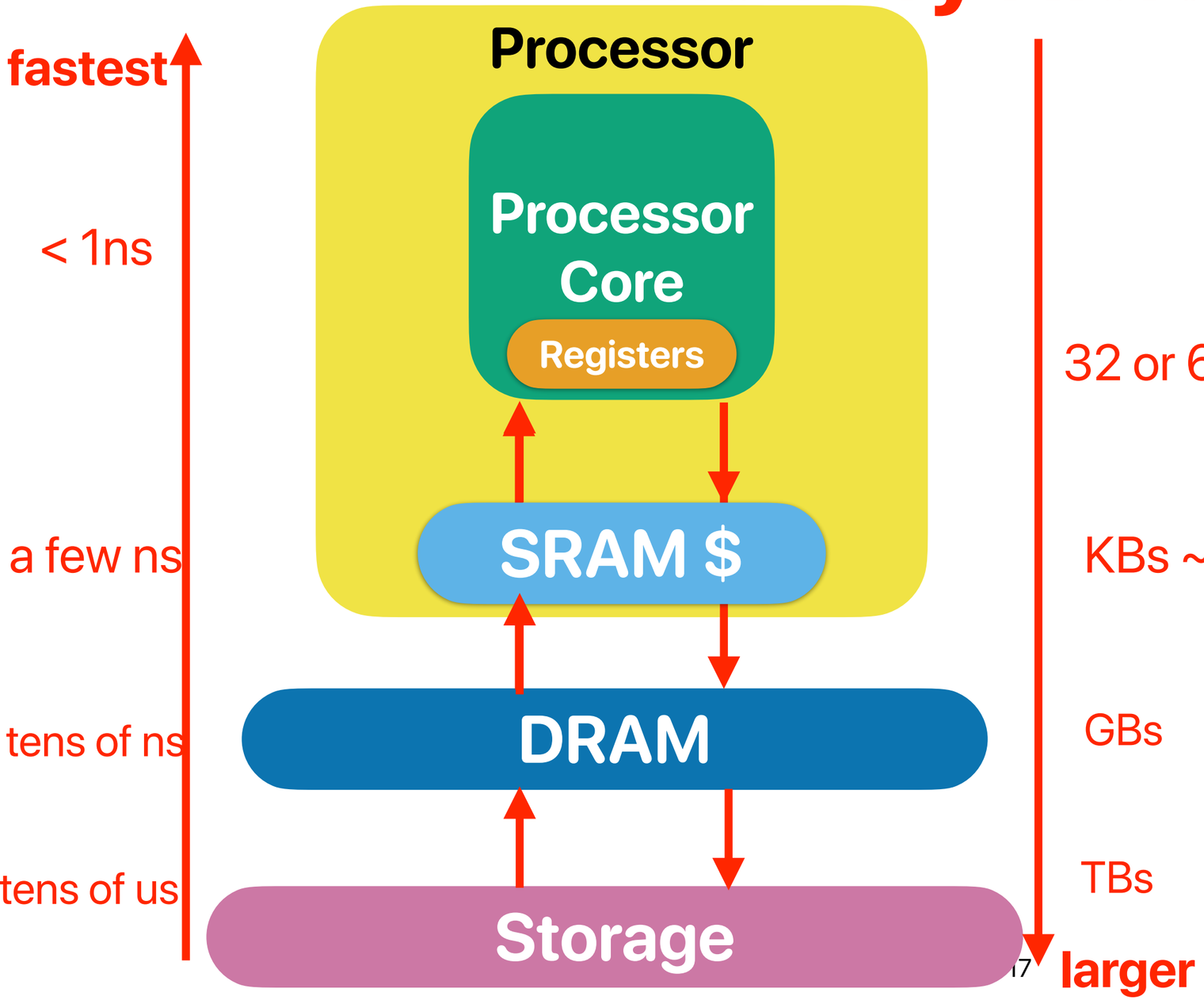
- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!

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 $$$

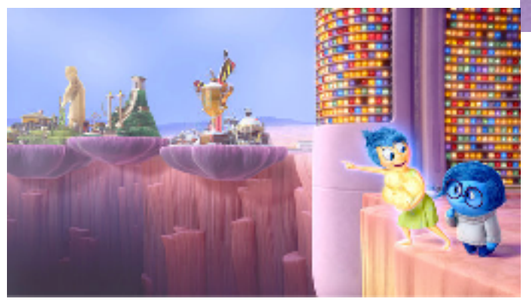
Memory Hierarchy



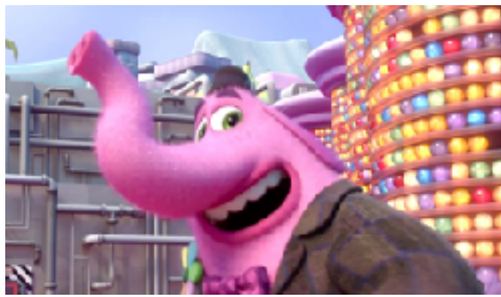
32 or 64 words

KBs ~ MBs

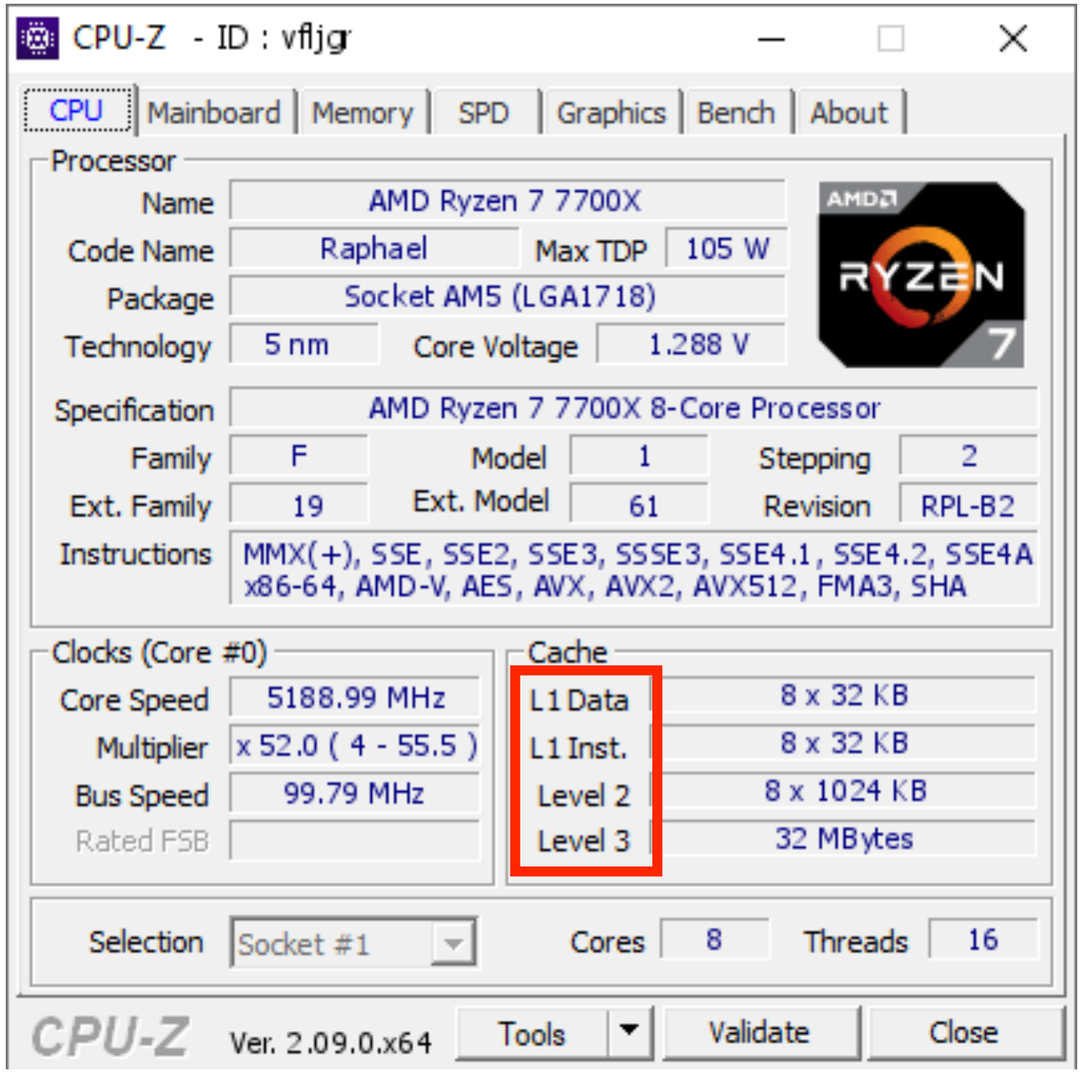
GBs

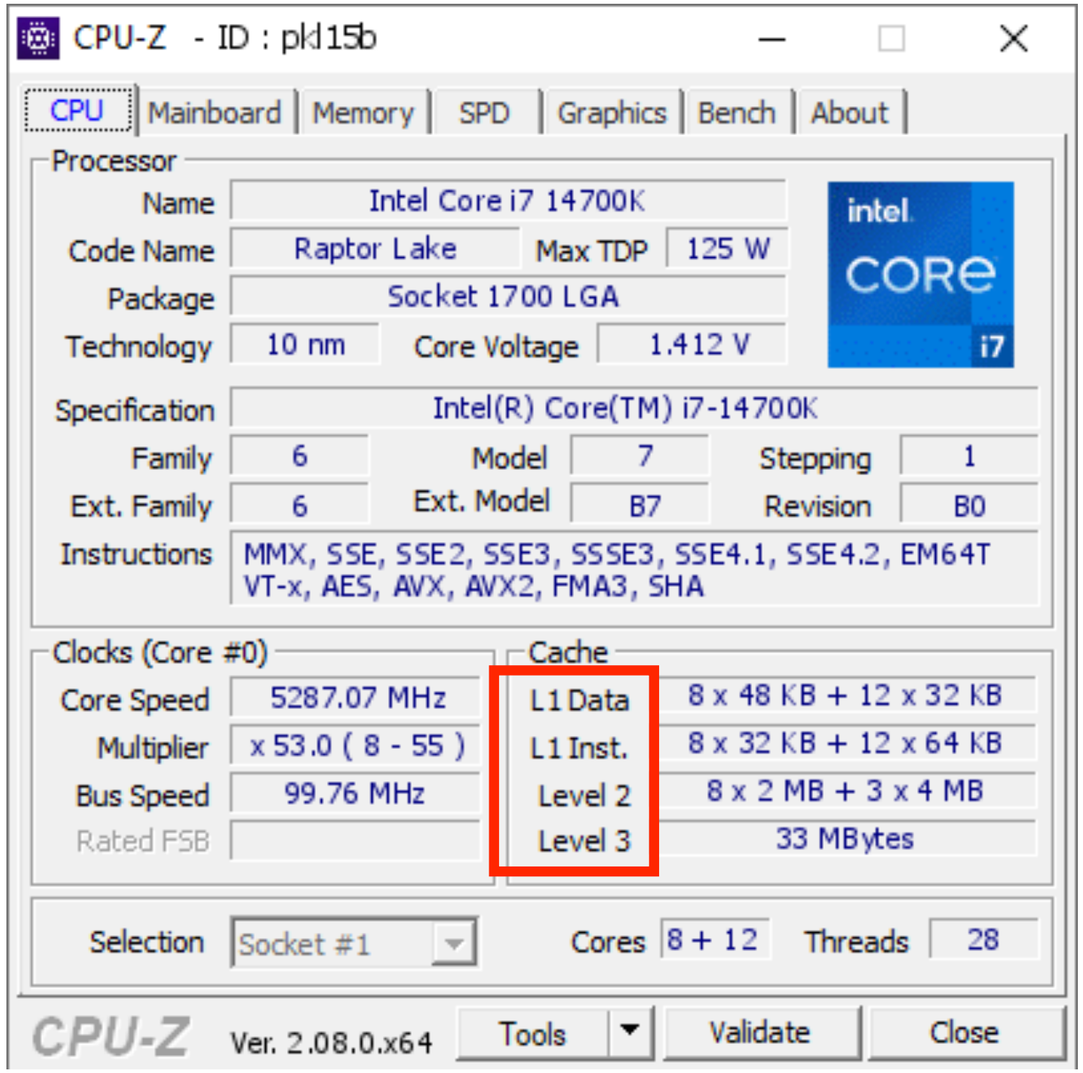


TBs

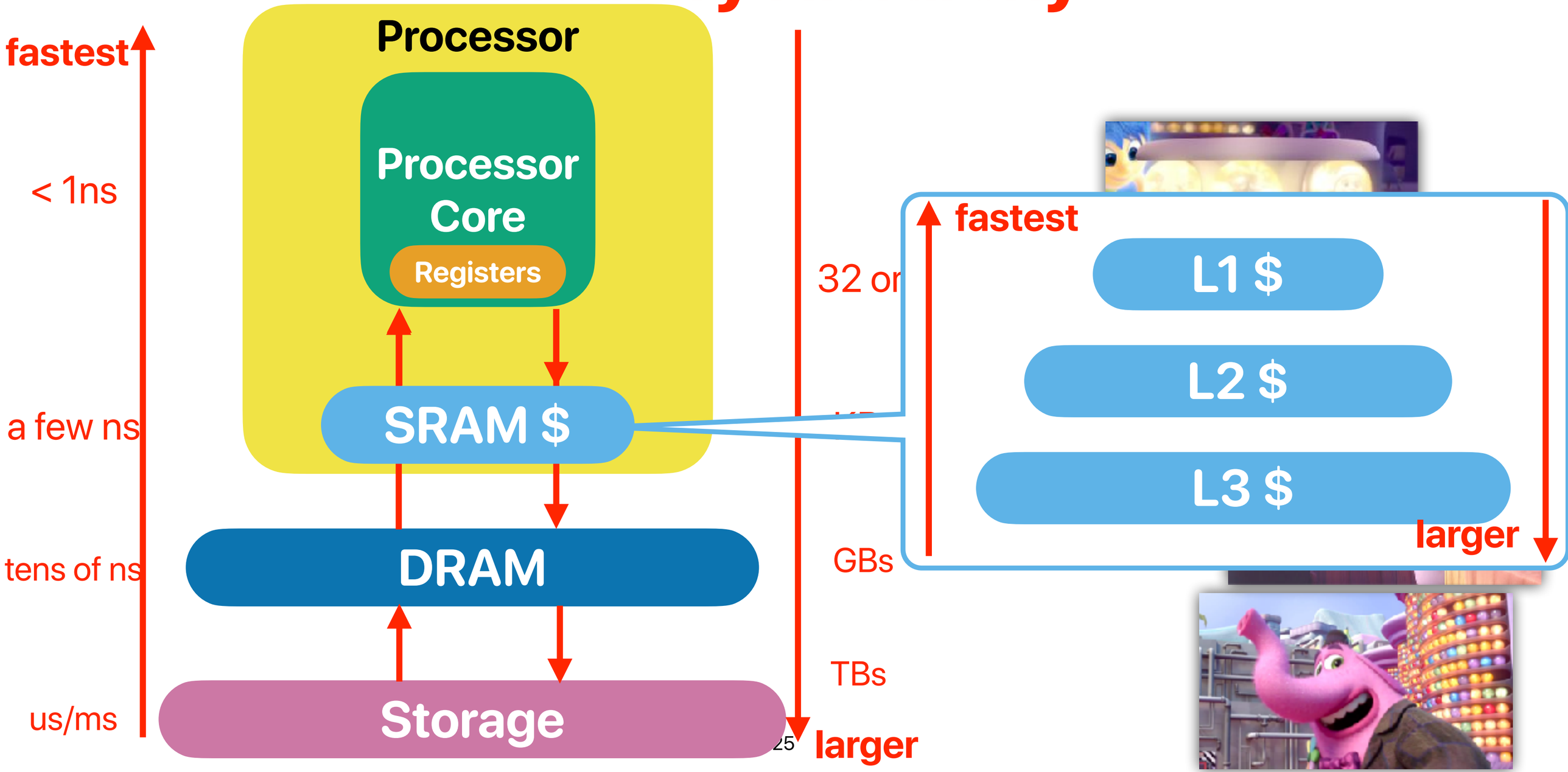


L1? L2? L3?

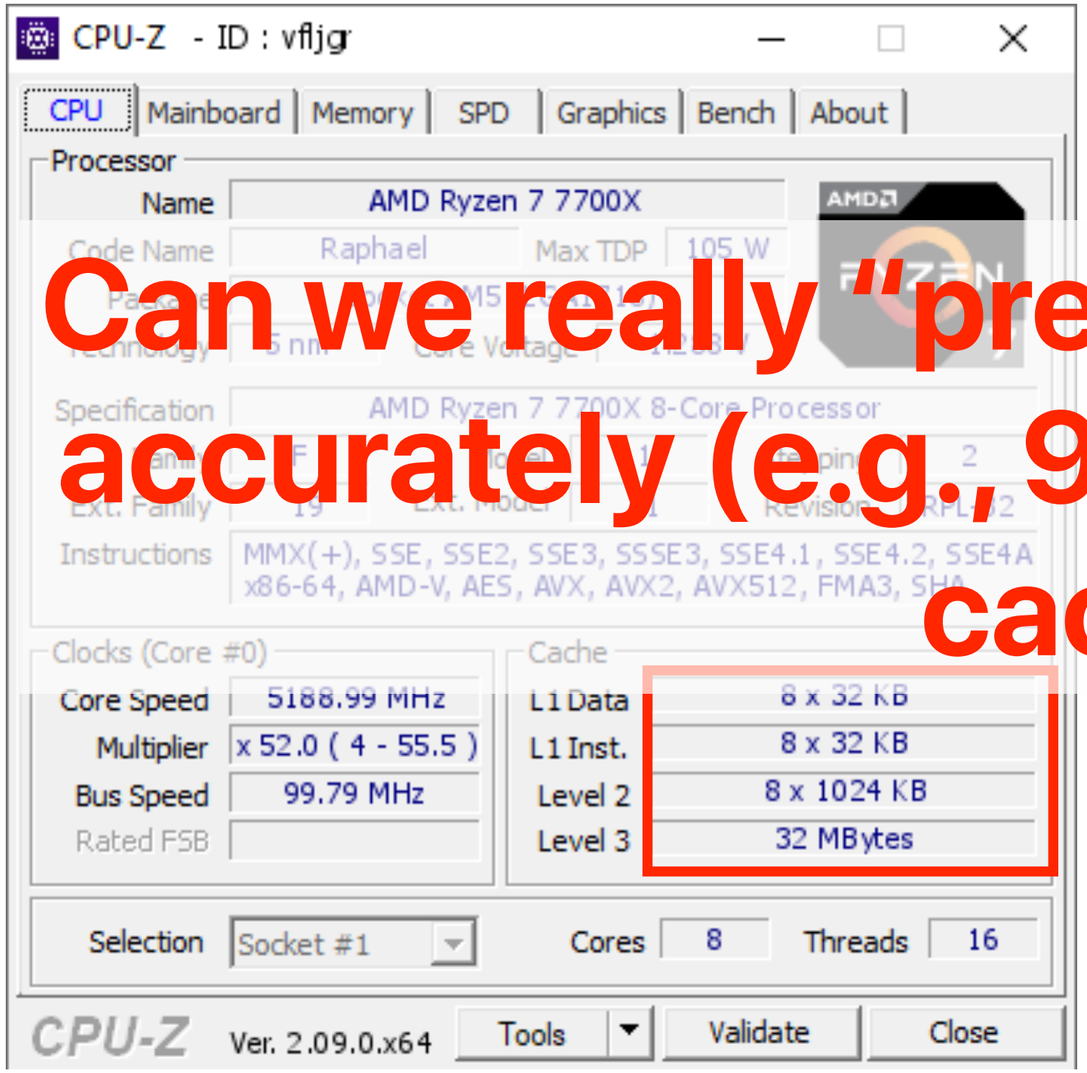

CPU-Z - ID : vfljg
CPU
Mainboard
Memory
SPD
Graphics
Bench
About
Processor
Name
AMD Ryzen 7 7700X
Code Name
Raphael
Max TDP
105 W
Package
Socket AM5 (LGA1718)
Technology
5 nm
Core Voltage
1.288 V
Specification
AMD Ryzen 7 7700X 8-Core Processor
Family
F
Model
1
Stepping
2
Ext. Family
19
Ext. Model
61
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
CPU-Z
Ver. 2.09.0.x64
Tools
Validate
Close

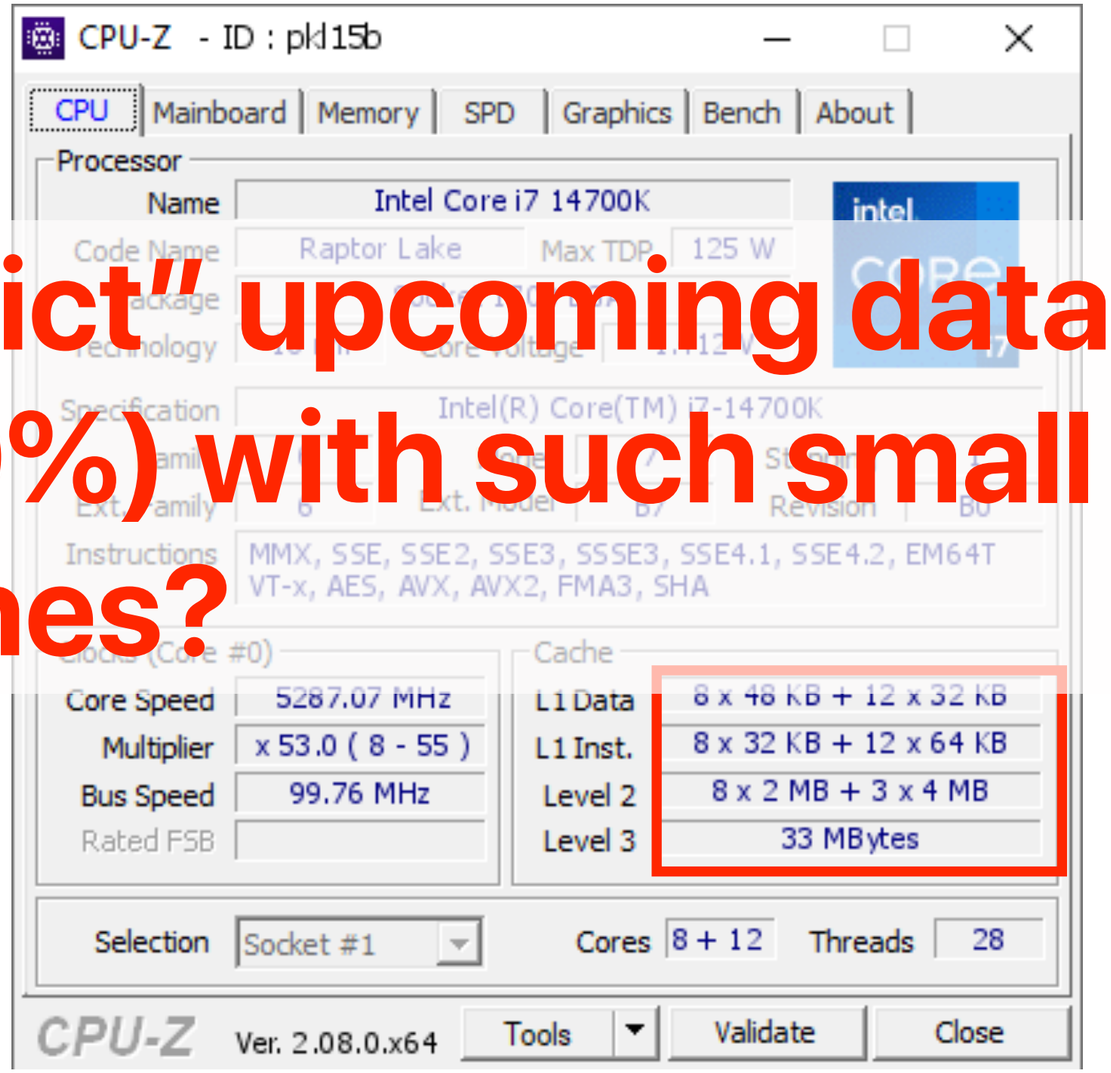

CPU-Z - ID : pkl15b
CPU
Mainboard
Memory
SPD
Graphics
Bench
About
Processor
Name
Intel Core i7 14700K
Code Name
Raptor Lake
Max TDP
125 W
Package
Socket 1700 LGA
Technology
10 nm
Core Voltage
1.412 V
Specification
Intel(R) Core(TM) i7-14700K
Family
6
Model
7
Stepping
1
Ext. Family
6
Ext. Model
B7
Revision
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
CPU-Z
Ver. 2.08.0.x64
Tools
Validate
Close

Memory Hierarchy



L1? L2? L3?





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

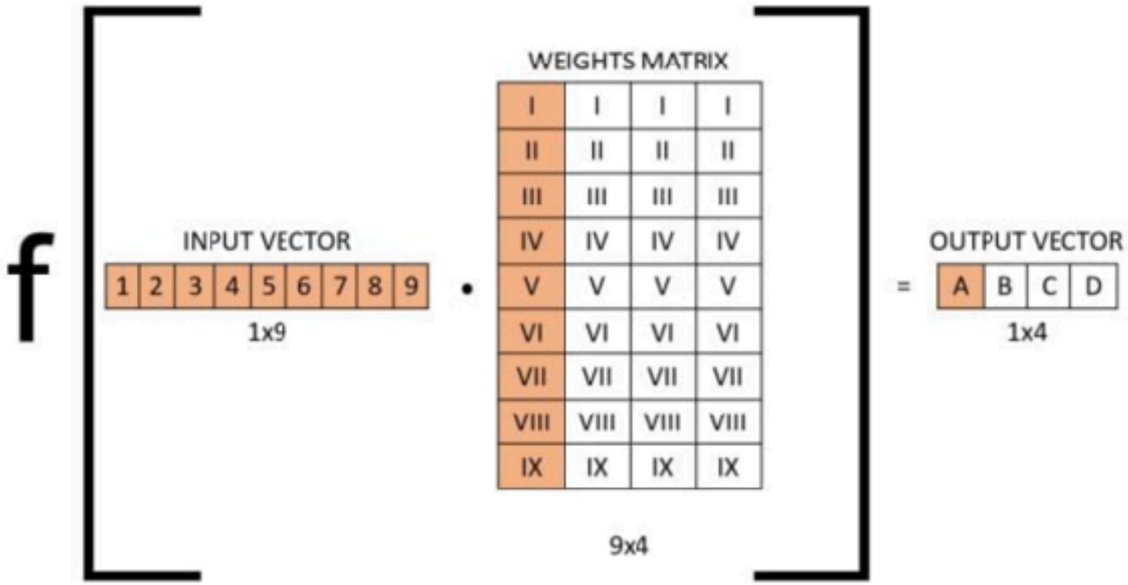
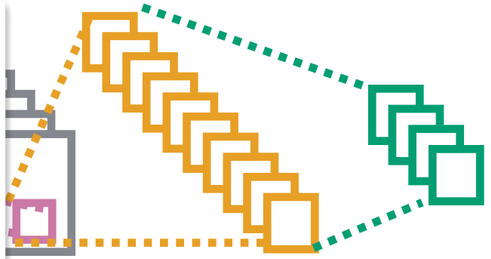


Illustration representing the fully connected layer's input multiplied by the weights matrix to receive the output vector. | Image: Diego Unzueta



Triton (1)
Cat (0)
Human (0)
Tiger (0)

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!

Locality

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

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

  Code — the current instruction, and then the next instruction
  Data — the current data item, and then the next data item

- 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)

Block and the memory space

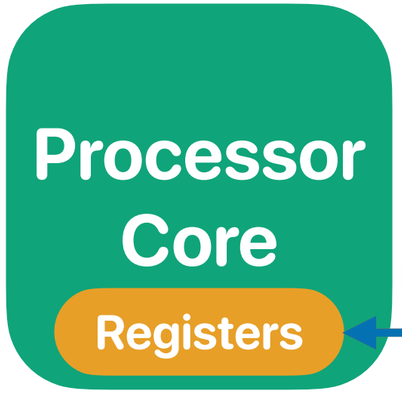
Each byte of memory location has an "address"

Partition the space with fixed-size "blocks" (e.g., 16-byte)

0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH
0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F	0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH
0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F
0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	008A	008B	008C	008D	008E	008F	0090	0091	0092	0093	0094	0095	0096	0097	0098	0099	009A	009B	009C	009D	009E	009F
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
FFE0	FFE1	FFE2	FFE3	FFE4	FFE5	FFE6	FFE7	FFE8	FFE9	FFEA	FFEB	FFEC	FFED	FFEE	FFEF	FFF0	FFF1	FFF2	FFF3	FFF4	FFF5	FFF6	FFF7	FFF8	FFF9	FFFA	FFFB	FFFC	FFFD	FFFE	FFFF

When there is a “movl”

```
movl (0x0024), %eax
movl (0x0020), %eax
```



Every “movl” has to visit the slow memory!

0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH

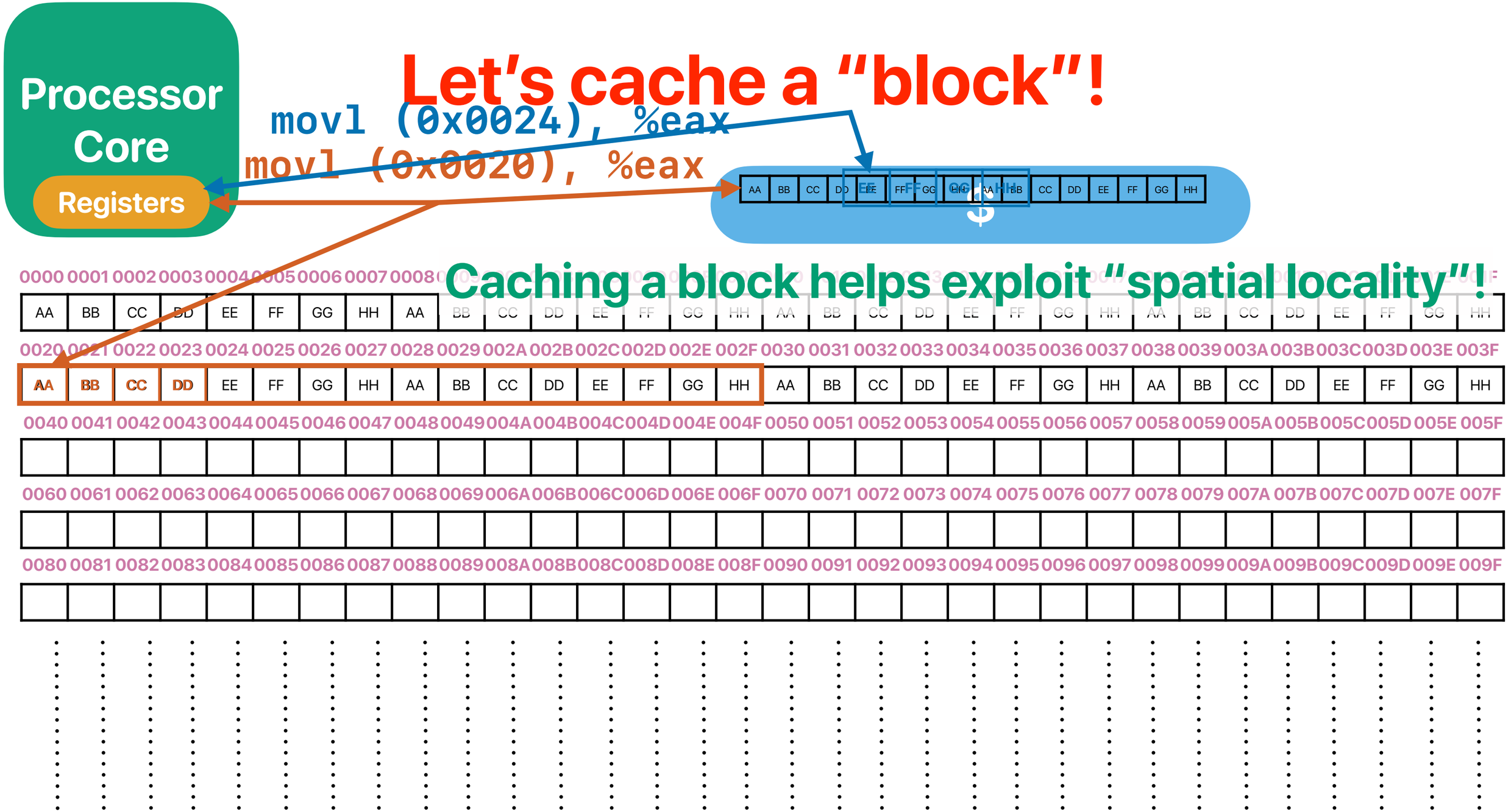
0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F	0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH

0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F

0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F

0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	008A	008B	008C	008D	008E	008F	0090	0091	0092	0093	0094	0095	0096	0097	0098	0099	009A	009B	009C	009D	009E	009F

Let's cache a "block"!



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

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

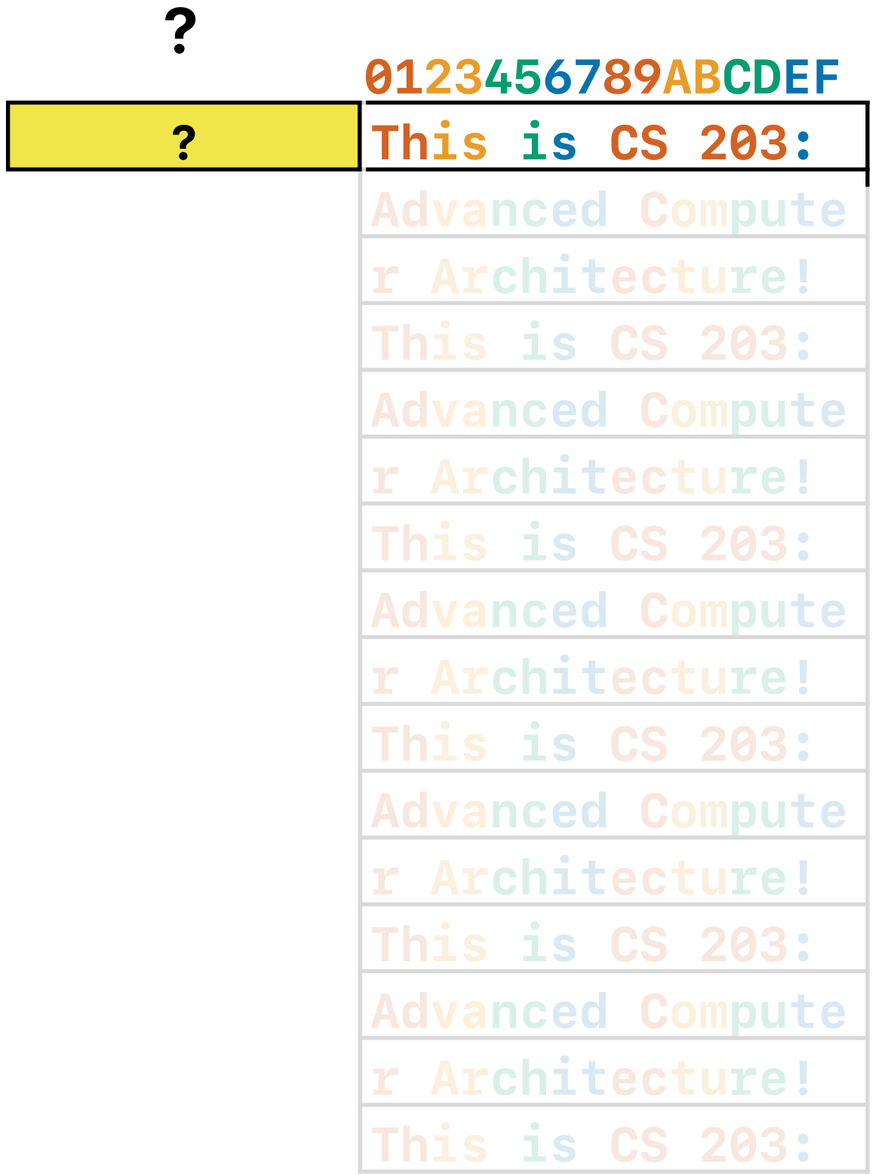
- Temporal locality — application revisit the same thing again and again

  We need to "cache frequently used memory blocks"

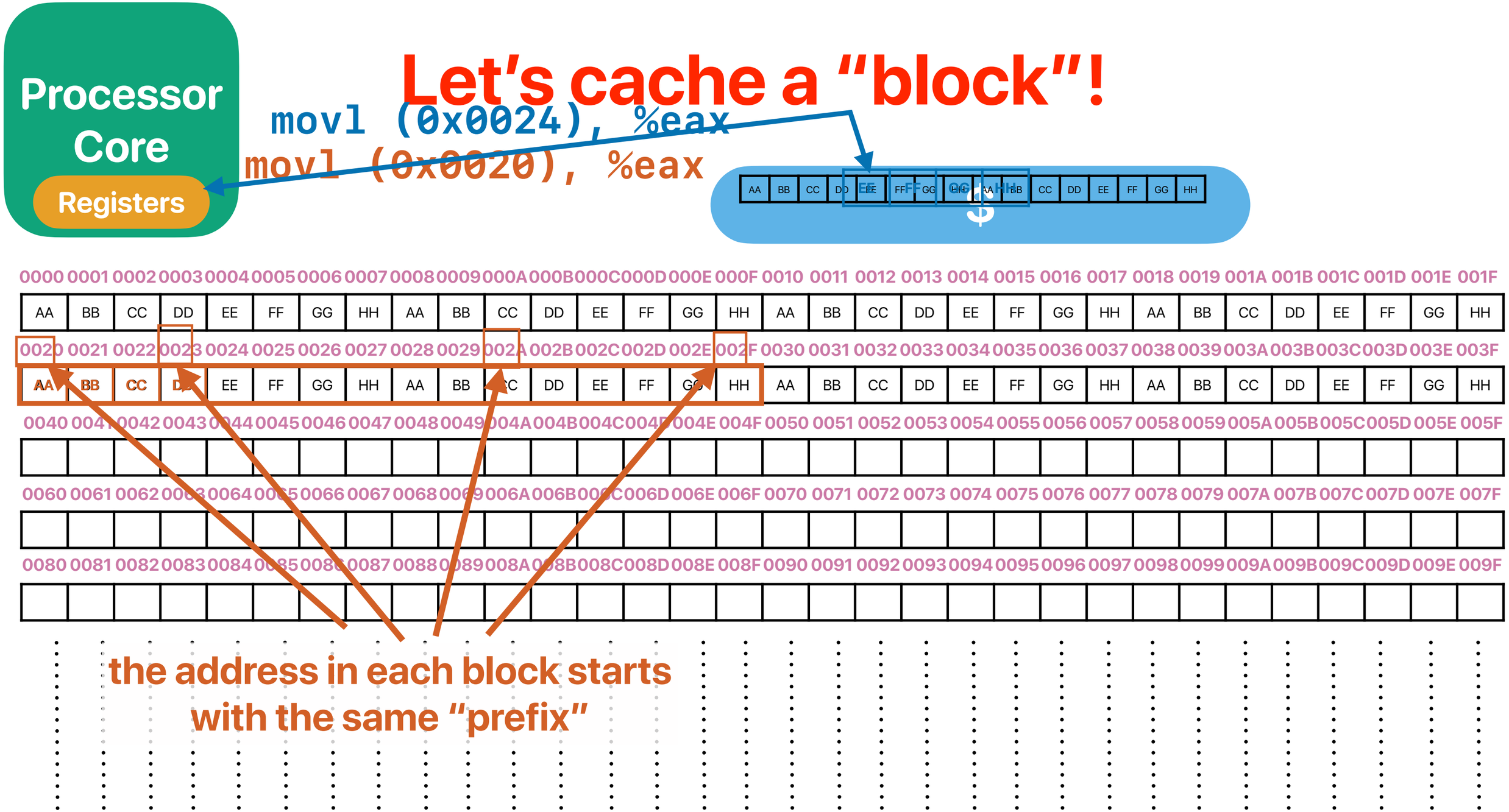
  — the cache should store a few blocks

  — the cache must be able to distinguish blocks

How to tell who is there?



Let's cache a "block"!



Processor Core

Registers

How to tell who is there?

the common address prefix in each block

tag array	0x000 0x001 0x002 0x003 0x004 0x005 0x006 0x007 0x008 0x009 0x00A 0x00B 0x00C 0x00D 0x00E 0x00F 0123456789ABCDEF
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

Processor Core

Registers

block offset

tag

lw 0x0008

lw 0x4048

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

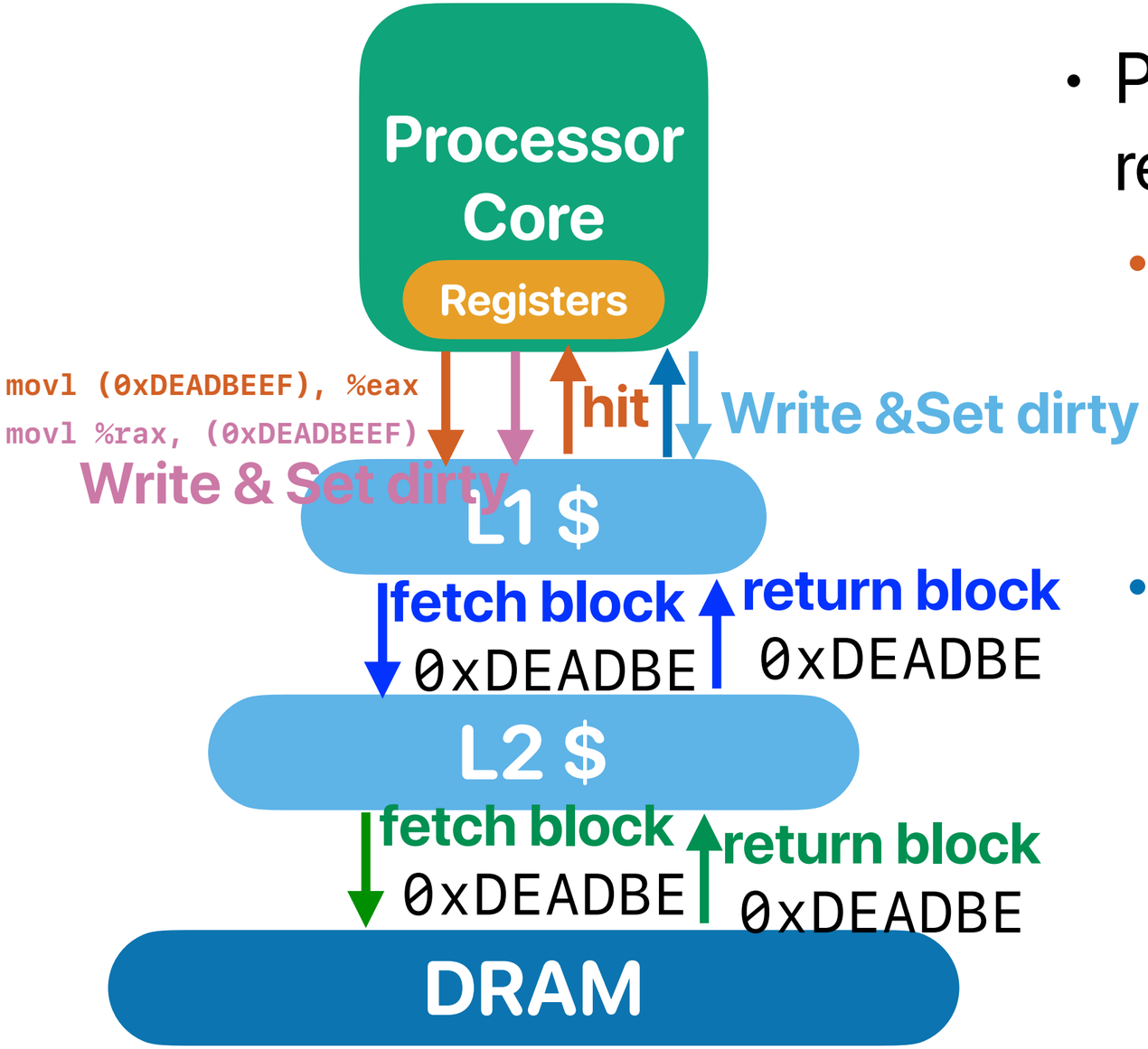
- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!
- Hierarchical caching with small amount of SRAMs will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

Take-aways: designing caches

- 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

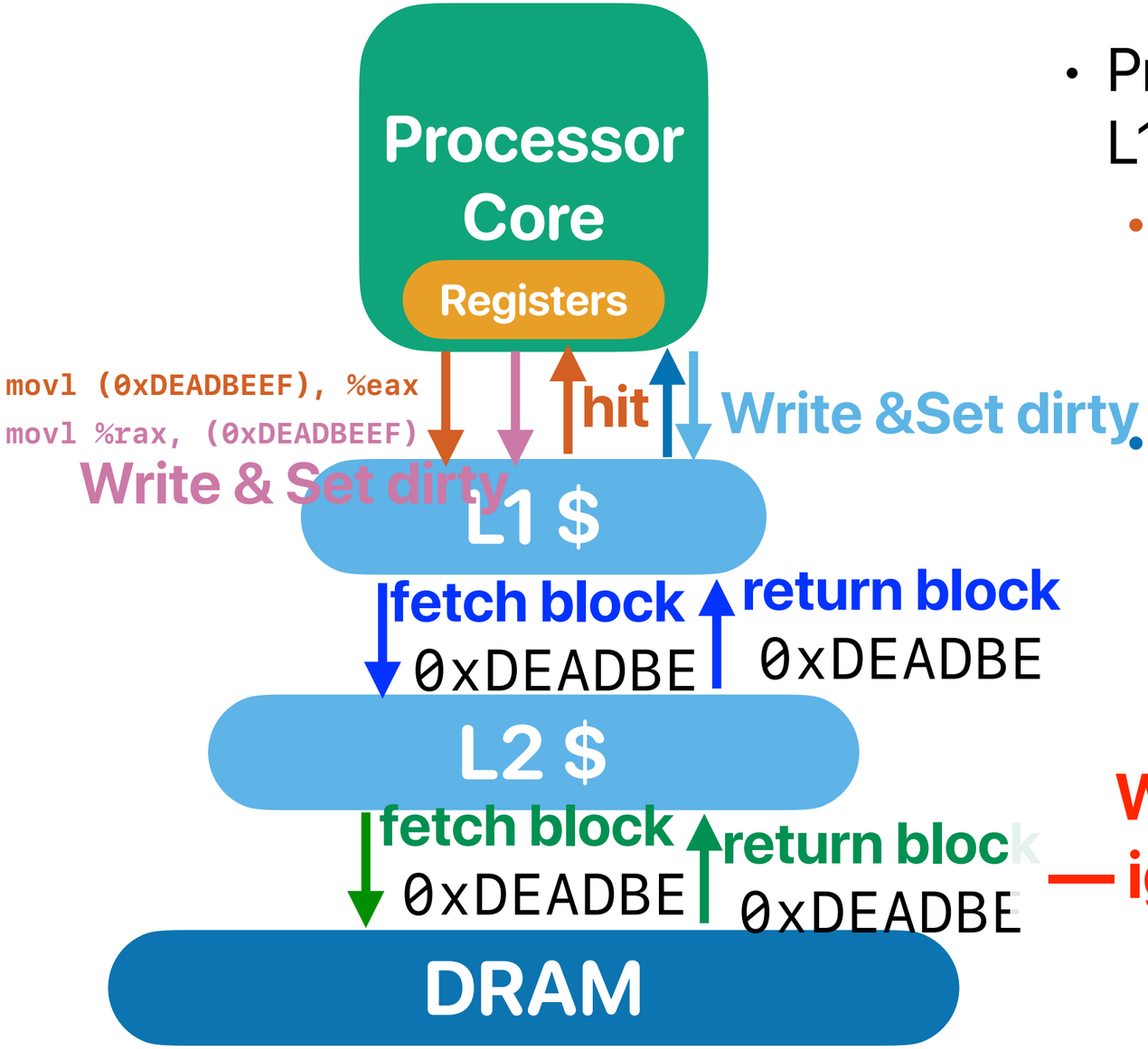
Processor/cache interaction



- Processor 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?

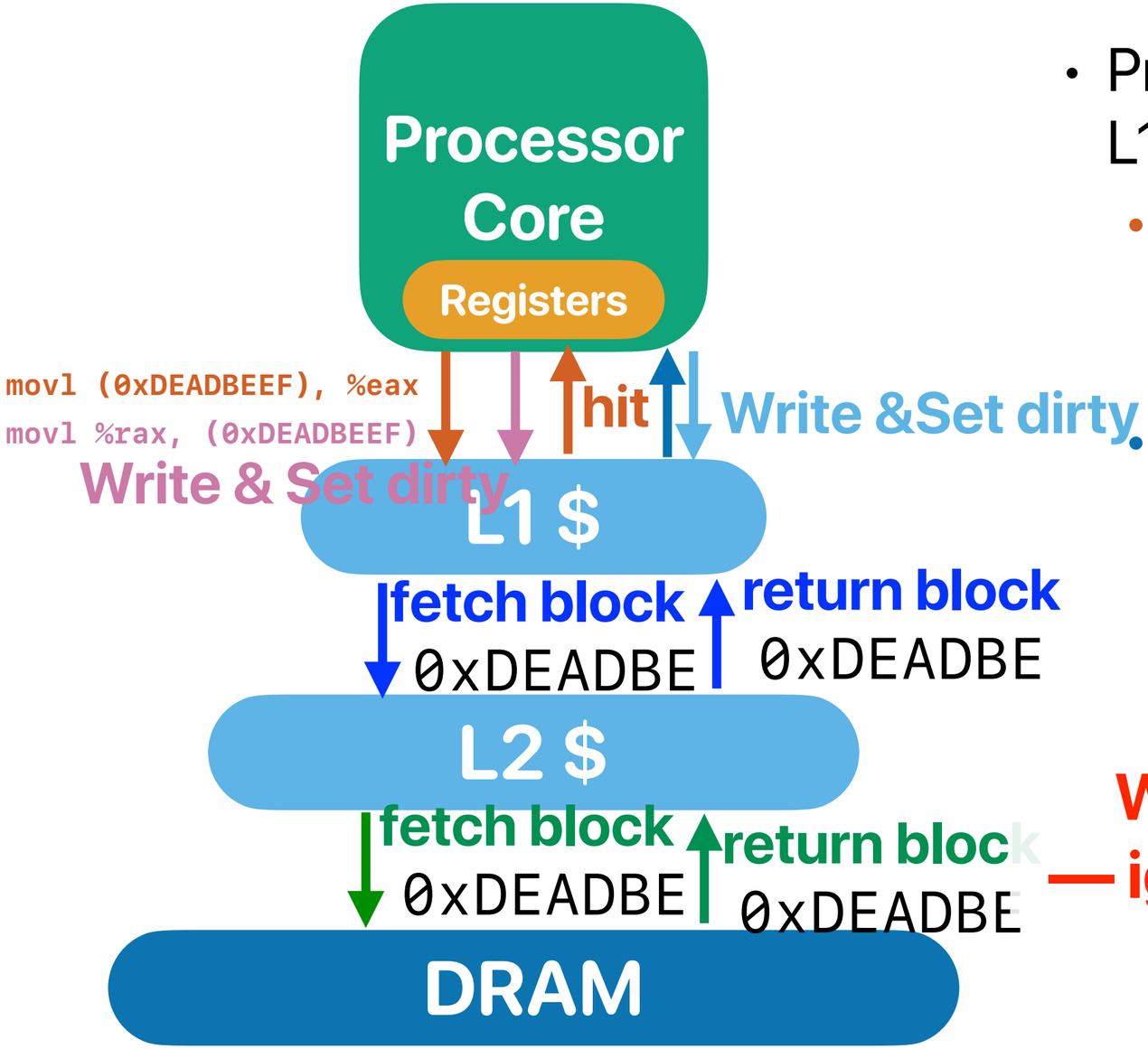
Processor/cache interaction



- 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!

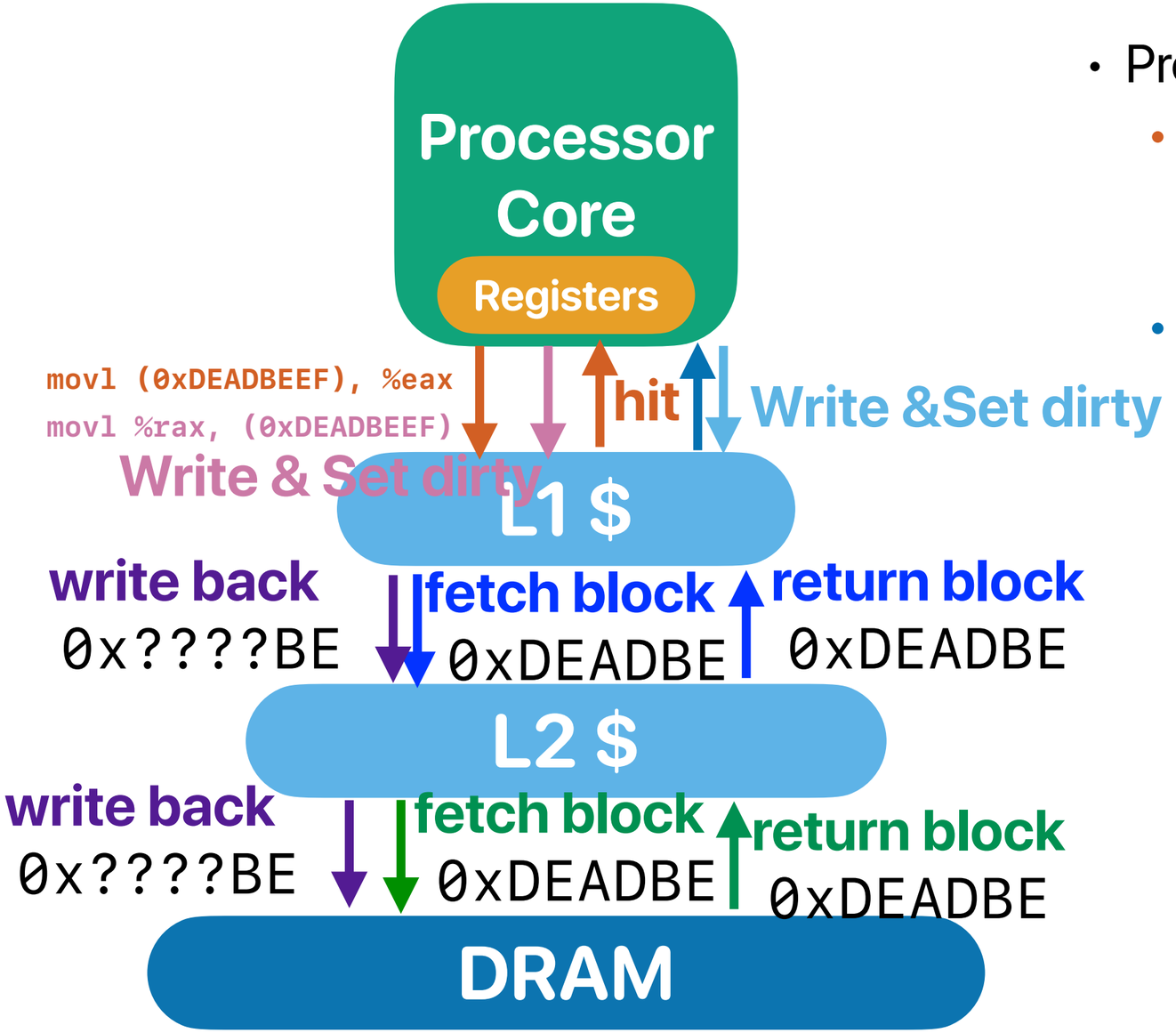
Processor/cache interaction



- 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: designing caches

- 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

How to tell w

Processor Core

Registers

block offset

tag

lw 0x0008

lw 0x4048

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:

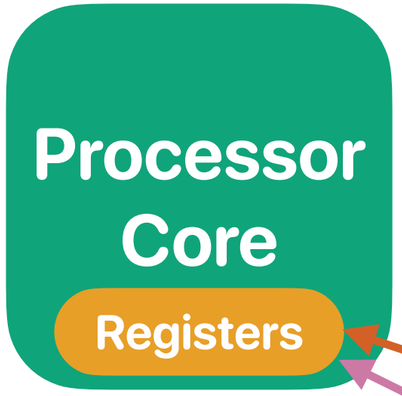
The complexity of search the matching tag—

$O(n)$— will be slow if our cache size grows!

Can we search things faster?

—hash table! $O(1)$

Hash-like structure — direct-mapped cache



block offset

tag

index

load 0x0008

load 0x4048

0x40 not found,
go to lower-level memory

V	D	tag	data (0123456789ABCDEF)
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
0	1	0xCB	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
1	0	0x10	r Architecture!
0	1	0x11	This is CS 203:

Take-aways: designing caches

- 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
- Optimizing cache structures
  - Hash block into "sets" to reduce the search time

Hash-like structure — direct-mapped cache

block offset

tag

index

load 0x0008

load 0x4048

0x40 not found,
go to lower-level memory

V	D	tag	data (0123456789ABCDEF)
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
0	1	0xCB	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
1	0	0x10	r Architecture!
0	1	0x11	This is CS 203:

The biggest issue with hash is —
Collision!

Way-associative cache

memory address: 0x0 8 2 4

set block

tag index offset

memory address: 0b0000100000100100

V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CS 203:

V	D	tag	data
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute

Set

0x1 0

=?

=?

hit?

hit?

What is Associativity?

```
[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: designing caches

- 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
- Optimizing cache structures
  - Hash block into "sets" to reduce the search time
  - Set-associativity to reduce the "collision" problem

Way-associative cache

memory address: 0x0 8 2 4

set block

tag index offset

memory address: 0b00001000001000100

V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CS 203:

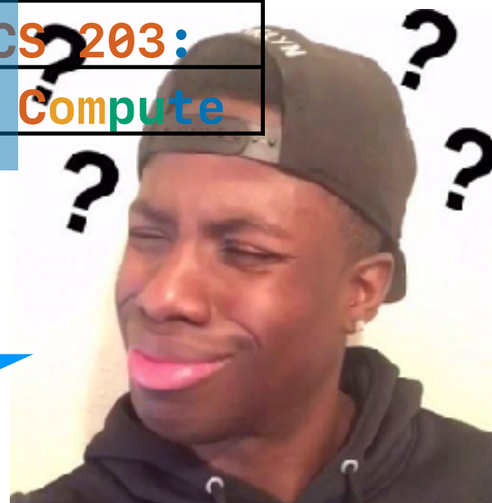
V	D	tag	data
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute

Set

0x1 0

=? =?

hit? hit?



The A, B, Cs of your cache

C = ABS

- **C**: **C**apacity in data arrays
- **A**: Way-**A**ssociativity — how many blocks within a set
  - N-way: N blocks in a set, A = N
  - 1 for direct-mapped cache
- **B**: **B**lock Size (Linesize)
  - How many bytes in a block
- **S**: Number of **S**ets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache

Corollary of C = ABS

	tag	set index	block offset
memory address:	0b00001000	0100	0100

- number of bits in **b**lock offset — lg(**B**)
- number of bits in **s**et index: lg(**S**)
- tag bits: address_length - lg(S) - lg(B)
  - address_length is N bits for N-bit machines (e.g., 64-bit for 64-bit machines)
- (address / block_size) % S = set index

Take-aways: designing caches

- 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
- Optimizing cache structures
  - Hash block into "sets" to reduce the search time
  - Set-associativity to reduce the "collision" problem
- C = A B S
  - C: capacity
  - A: Associativity
  - S: Number of sets
  - lg(S): Number of bits in set index
  - lg(B): Number of bits in block offset

Simulate the cache!

Simulate a direct-mapped cache

- A direct mapped (1-way) cache with 256 bytes total capacity, a block size of 16 bytes
  - # of blocks = $\frac{256}{16} = 16$
  - lg(16) = 4 : 4 bits are used for the index
  - lg(16) = 4 : 4 bits are used for the byte offset
  - The tag is 64 - (4 + 4) = 56 bits
  - For example: `0x 8 0 0 0 0 0 8 0`
    = `0b1000 0000 0000 0000 0000 0000 1000 0000`

tag

index

offset

Matrix vector revisited

```
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;
 }
```

Matrix vector revisited

```
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;
}
```

	Address (Hex) (tag index)	Address (Binary) (tag index)
&a[0][0]	0x558FE0A1D330	0b1010101100011111110000010100001110100110011 0000
&b[0]	0x558FE0A1DC30	0b1010101100011111110000010100001110111000011 0000
&a[0][1]	0x558FE0A1D338	0b1010101100011111110000010100001110100110011 1000
&b[1]	0x558FE0A1DC38	0b1010101100011111110000010100001110111000011 1000
&a[0][2]	0x558FE0A1D340	0b1010101100011111110000010100001110100110100 0000
&b[2]	0x558FE0A1DC40	0b1010101100011111110000010100001110111000100 0000
&a[0][3]	0x558FE0A1D348	0b1010101100011111110000010100001110100110100 1000
&b[3]	0x558FE0A1DC48	0b1010101100011111110000010100001110111000100 1000
&a[0][4]	0x558FE0A1D350	0b1010101100011111110000010100001110100110101 0000
&b[4]	0x558FE0A1DC50	0b1010101100011111110000010100001110111000101 0000
&a[0][5]	0x558FE0A1D358	0b1010101100011111110000010100001110100110101 1000
&b[5]	0x558FE0A1DC58	0b1010101100011111110000010100001110111000101 1000
&a[0][6]	0x558FE0A1D360	0b1010101100011111110000010100001110100110110 0000
&b[6]	0x558FE0A1DC60	0b1010101100011111110000010100001110111000110 0000
&a[0][7]	0x558FE0A1D368	0b1010101100011111110000010100001110100110110 1000
&b[7]	0x558FE0A1DC68	0b1010101100011111110000010100001110111000110 1000
&a[0][8]	0x558FE0A1D370	0b1010101100011111110000010100001110100110111 0000
&b[8]	0x558FE0A1DC70	0b1010101100011111110000010100001110111000111 0000
&a[0][9]	0x558FE0A1D378	0b1010101100011111110000010100001110100110111 1000
&b[9]	0x558FE0A1DC78	0b1010101100011111110000010100001110111000111 1000

Simulate a direct-mapped cache

	V	D	Tag	Data
0	0	0		
1	0	0		
2	0	0		
3	1	0	0x558FE0A1DC	b[0], b[1]
4	1	0	0x558FE0A1DC	b[2], b[3]
5	0	0		
6	0	0		
7	0	0		
8	0	0		
9	0	0		
10	0	0		
11	0	0		
12	0	0		
13	0	0		
14	0	0		
15	0	0		

This cache doesn't work!!! — collisions!

tag index

	Address (Hex)	
&a[0][0]	0x558FE0A1D330	miss
&b[0]	0x558FE0A1DC30	miss
&a[0][1]	0x558FE0A1D338	miss
&b[1]	0x558FE0A1DC38	miss
&a[0][2]	0x558FE0A1D340	miss
&b[2]	0x558FE0A1DC40	miss
&a[0][3]	0x558FE0A1D348	miss
&b[3]	0x558FE0A1DC48	miss
&a[0][4]	0x558FE0A1D350	miss
&b[4]	0x558FE0A1DC50	miss
&a[0][5]	0x558FE0A1D358	miss
&b[5]	0x558FE0A1DC58	miss
&a[0][6]	0x558FE0A1D360	miss
&b[6]	0x558FE0A1DC60	miss
&a[0][7]	0x558FE0A1D368	miss
&b[7]	0x558FE0A1DC68	miss
&a[0][8]	0x558FE0A1D370	miss
&b[8]	0x558FE0A1DC70	miss
&a[0][9]	0x558FE0A1D378	
&b[9]	0x558FE0A1DC78	

Way-associative cache

memory address: 0x0 8 2 4

memory address: 0b0000100000100100

tag | set index | block offset

V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CS 203:

V	D	tag	data
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute

Set

0x1 0

=? =?

hit? hit?

Now, 2-way, same-sized cache

- A 2-way cache with 256 bytes total capacity, a block size of 16 bytes
  - # of blocks = $\frac{256}{16} = 16$
  - # of sets = $\frac{16}{2} = 8$ (2-way: 2 blocks in a set)
  - lg(8) = 3 : 3 bits are used for the index
  - lg(16) = 4 : 4 bits are used for the byte offset
  - The tag is 64 - (4 + 4) = 56 bits
  - For example: `0x 8 0 0 0 0 0 8 0`
    = `0b1000 0000 0000 0000 0000 0000 1000 0000`

tag

index

offset

Matrix vector revisited

```
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;
}
```

tag index tag index

	Address (Hex)	Address (Binary)
&a[0][0]	0x558FE0A1D330	0b1010101100011111110000010100001110100110011 0000
&b[0]	0x558FE0A1DC30	0b1010101100011111110000010100001110111000011 0000
&a[0][1]	0x558FE0A1D338	0b1010101100011111110000010100001110100110011 1000
&b[1]	0x558FE0A1DC38	0b1010101100011111110000010100001110111000011 1000
&a[0][2]	0x558FE0A1D340	0b1010101100011111110000010100001110100110100 0000
&b[2]	0x558FE0A1DC40	0b1010101100011111110000010100001110111000100 0000
&a[0][3]	0x558FE0A1D348	0b1010101100011111110000010100001110100110100 1000
&b[3]	0x558FE0A1DC48	0b1010101100011111110000010100001110111000100 1000
&a[0][4]	0x558FE0A1D350	0b1010101100011111110000010100001110100110101 0000
&b[4]	0x558FE0A1DC50	0b1010101100011111110000010100001110111000101 0000
&a[0][5]	0x558FE0A1D358	0b1010101100011111110000010100001110100110101 1000
&b[5]	0x558FE0A1DC58	0b1010101100011111110000010100001110111000101 1000
&a[0][6]	0x558FE0A1D360	0b1010101100011111110000010100001110100110110 0000
&b[6]	0x558FE0A1DC60	0b1010101100011111110000010100001110111000110 0000
&a[0][7]	0x558FE0A1D368	0b1010101100011111110000010100001110100110110 1000
&b[7]	0x558FE0A1DC68	0b1010101100011111110000010100001110111000110 1000
&a[0][8]	0x558FE0A1D370	0b1010101100011111110000010100001110100110111 0000
&b[8]	0x558FE0A1DC70	0b1010101100011111110000010100001110111000111 0000
&a[0][9]	0x558FE0A1D378	0b1010101100011111110000010100001110100110111 1000
&b[9]	0x558FE0A1DC78	0b1010101100011111110000010100001110111000111 1000

Simulate a 2-way cache

V	D	Tag	Data	V	D	Tag	Data
0	0			0	0		
0	0			0	0		
0	0			0	0		
1	0	0xAB1FC143A6	a[0][0], a[0][1]	1	0	0xAB1FC143B8	b[0], b[1]
1	0	0xAB1FC143A6	a[0][2], a[0][3]	1	0	0xAB1FC143B8	b[2], b[3]
0	0			0	0		
0	0			0	0		
0	0			0	0		

	Address (Hex)	Tag	Index	
&a[0][0]	0x558FE0A1D330	0xAB1FC143A6	0x3	miss
&b[0]	0x558FE0A1DC30	0xAB1FC143B8	0x3	miss
&a[0][1]	0x558FE0A1D338	0xAB1FC143A6	0x3	hit
&b[1]	0x558FE0A1DC38	0xAB1FC143B8	0x3	hit
&a[0][2]	0x558FE0A1D340	0xAB1FC143A6	0x4	miss
&b[2]	0x558FE0A1DC40	0xAB1FC143B8	0x4	miss
&a[0][3]	0x558FE0A1D348	0xAB1FC143A6	0x4	hit
&b[3]	0x558FE0A1DC48	0xAB1FC143B8	0x4	hit
&a[0][4]	0x558FE0A1D350	0xAB1FC143A6	0x5	miss
&b[4]	0x558FE0A1DC50	0xAB1FC143B8	0x5	miss
&a[0][5]	0x558FE0A1D358	0xAB1FC143A6	0x5	hit
&b[5]	0x558FE0A1DC58	0xAB1FC143B8	0x5	hit
&a[0][6]	0x558FE0A1D360	0xAB1FC143A6	0x6	miss
&b[6]	0x558FE0A1DC60	0xAB1FC143B8	0x6	miss
&a[0][7]	0x558FE0A1D368	0xAB1FC143A6	0x6	hit
&b[7]	0x558FE0A1DC68	0xAB1FC143B8	0x6	hit
&a[0][8]	0x558FE0A1D370	0xAB1FC143A6	0x7	miss
&b[8]	0x558FE0A1DC70	0xAB1FC143B8	0x7	miss
&a[0][9]	0x558FE0A1D378	0xAB1FC143A6	0x7	hit
&b[9]	0x558FE0A1DC78	0xAB1FC143B8	0x7	hit

Taxonomy/reasons of cache misses

3Cs of misses

- Compulsory miss
  - Cold start miss. First-time access to a block
- Capacity miss
  - The working set size of an application is bigger than cache size
- Conflict miss
  - Required data replaced by block(s) mapping to the same set
  - Similar collision in hash