



UC Berkeley
Teaching Professor
Lisa Yan

CS61C

Great Ideas in Computer Architecture (a.k.a. Machine Structures)



UC Berkeley
Lecturer
Justin Yokota

Caches II: AMAT, Fully Associative

- [2024 Berkeley Engineering pulse survey](#) (until Friday April 19)
- EECS/CS Undergraduate Town Hall, Thursday April 11 4-5pm, Wozniak Lounge (Soda Hall).

Please participate in both of these if you can. It is important that this department hear your undergraduate voices.

cs61c.org

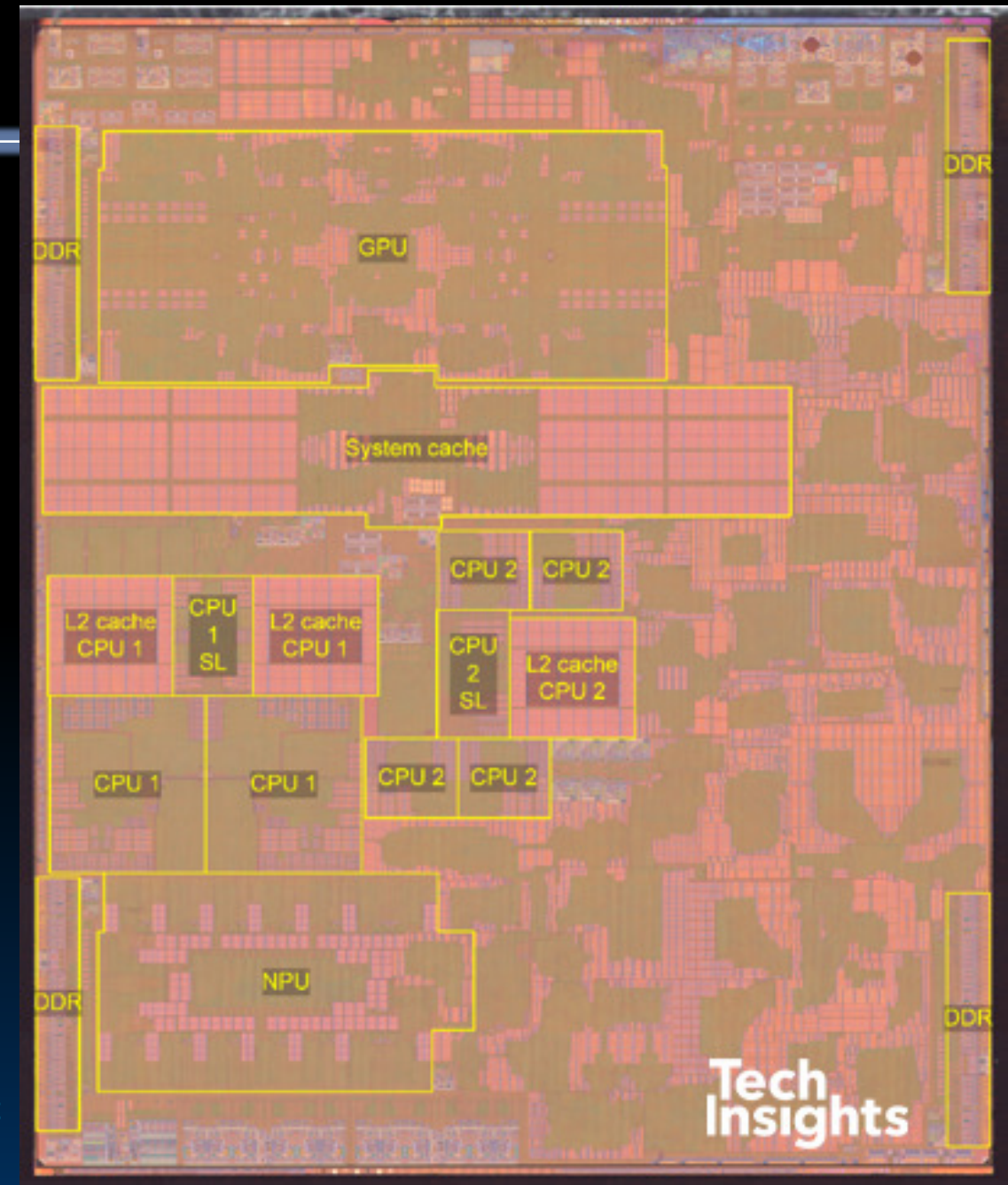
Yan, Yokota



Memory Cache

[review]

- Mismatch between processor and memory speeds leads us to add a new level: The “memory cache”, or **cache** for short.
 - Usually on the same chip as the CPU.
 - **Faster but more expensive** than DRAM memory.
- The cache is a copy of a subset of main memory.
- Most processors have separate caches for **instructions and data**.
(more later)

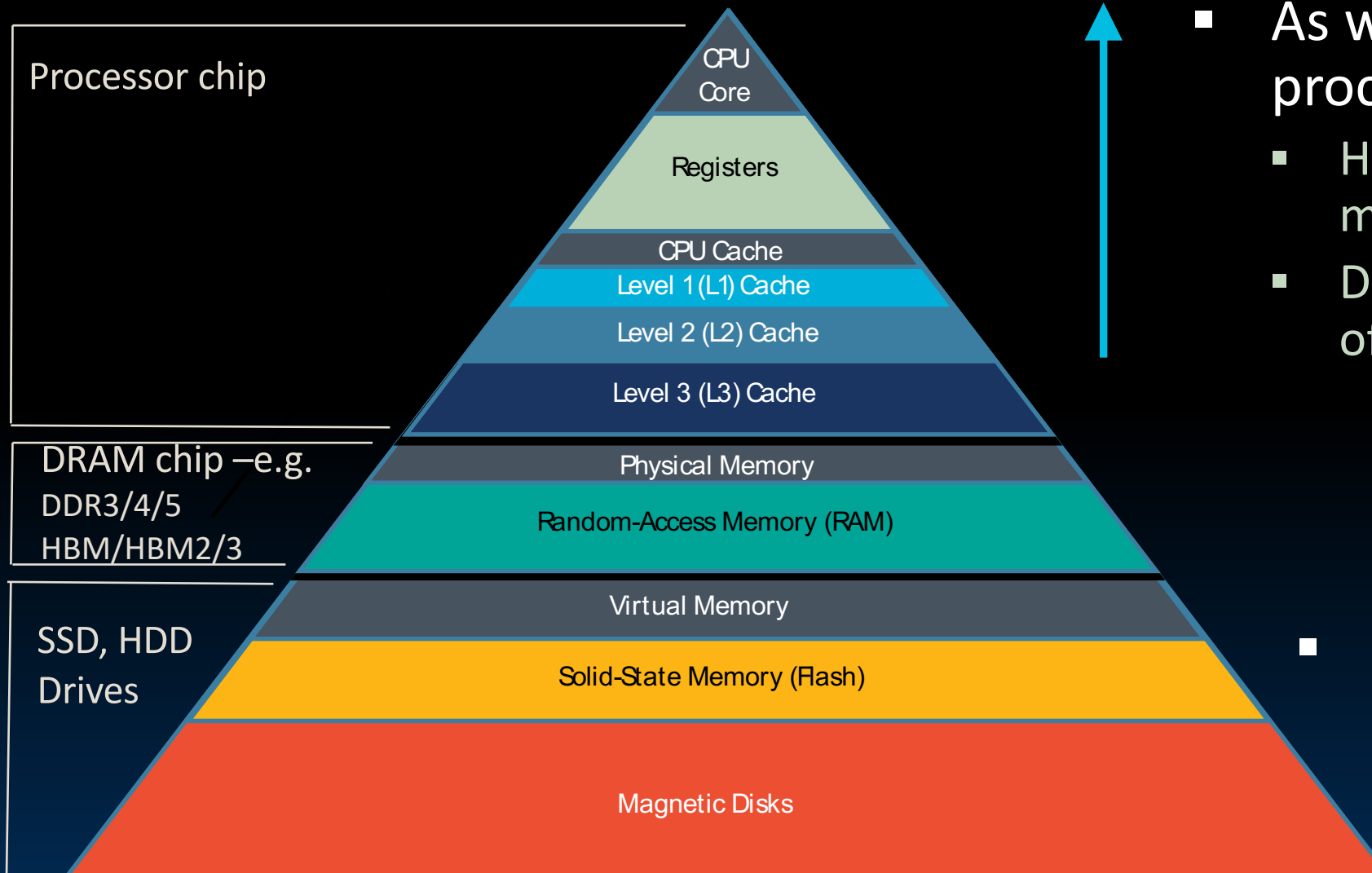


Apple A14 Bionic
TechInsights [[source](#)]

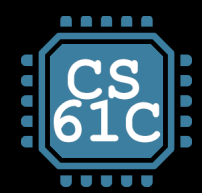
31-Caches-II-AMAT, Fully Associative (2)



Great Idea #3: Principle of Locality / Memory Hierarchy



- As we go closer to the processor:
 - HW is smaller, faster, and more expensive
 - Data is smaller and a subset of lower levels
- The lowest level contains all available data.
 - Nowadays, we don't go to magnetic disk.




AMAT: Average Memory Access Time

- AMAT: Average Memory Access Time
- Cache Design
- Fully Associative Cache
- Warming up the Fully Associative Cache
- Block Replacement Policies
- Write Policies

6 Great Ideas in Computer Architecture

1. Abstraction (Layers of Representation/Interpretation)
2. Moore's Law
3. Principle of Locality/Memory Hierarchy
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via Redundancy



We'll discuss **performance metrics** first, before we look at specifics of cache design.



Cache Hit vs. Cache Miss

- **Cache Hit**
 - The data you were looking for **is** in the cache.
 - Retrieve the data from the cache and bring it to the processor.
- **Cache Miss**
 - The data you were looking for **is not** in the cache.
 - Go to a lower layer in the memory hierarchy to find the data, put the data in the cache.
 - Then, bring the data to the processor.
- **Hit rate**: fraction of access that hit in the cache.
- **Hit time**: time (latency) to access cache memory (including tag comparison [\(more later\)](#))
- **Miss rate**: $1 - \text{Hit rate}$.
- **Miss penalty**: time (latency) to replace a block from lower level in memory hierarchy to cache.

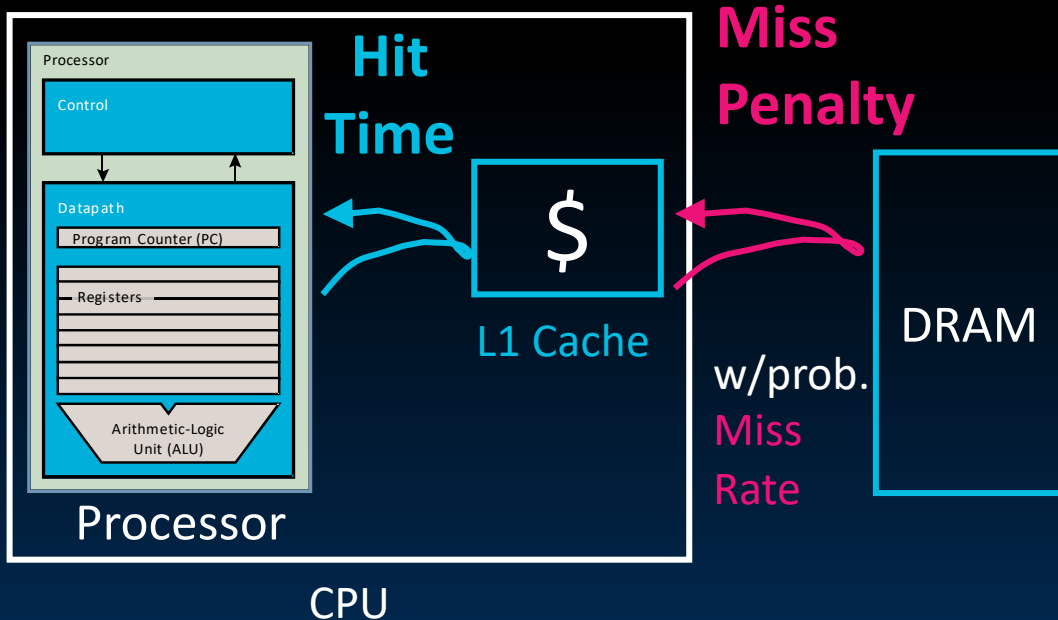
What is the Average Memory Access Time?

- Suppose you have the following:
 - Hit Time = 1 cycle
 - Miss rate = 5%
 - Miss penalty = 20 cycles

What is the Average Memory Access Time, in # cycles?

Hint 1: Think probability and expectation!

Hint 2: On cache miss, total time to retrieve data = Hit time + Miss penalty.



- A. 1
- B. 1.95
- C. 2
- D. 21
- E. Something else

What is the Average Memory Access Time, in cycles?



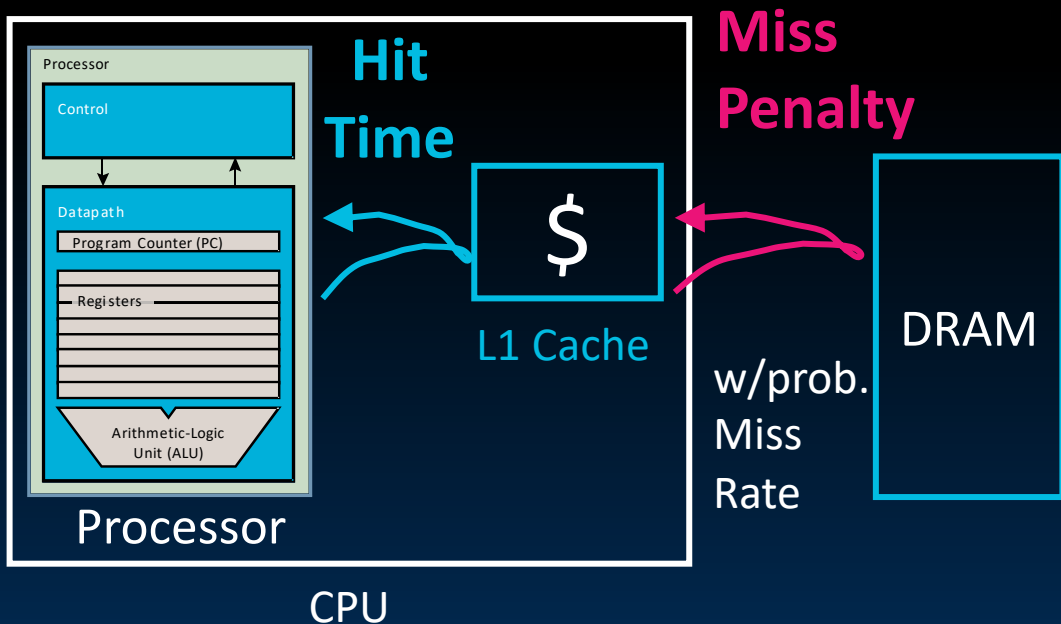
What is the Average Memory Access Time?

- Suppose you have the following:
 - Hit Time = 1 cycle
 - Miss rate = 5%
 - Miss penalty = 20 cycles

What is the **Average Memory Access Time (AMAT)** in # cycles?

Hint 1: Think probability and expectation!

Hint 2: On cache miss, total time to access data = Hit time + Miss penalty.

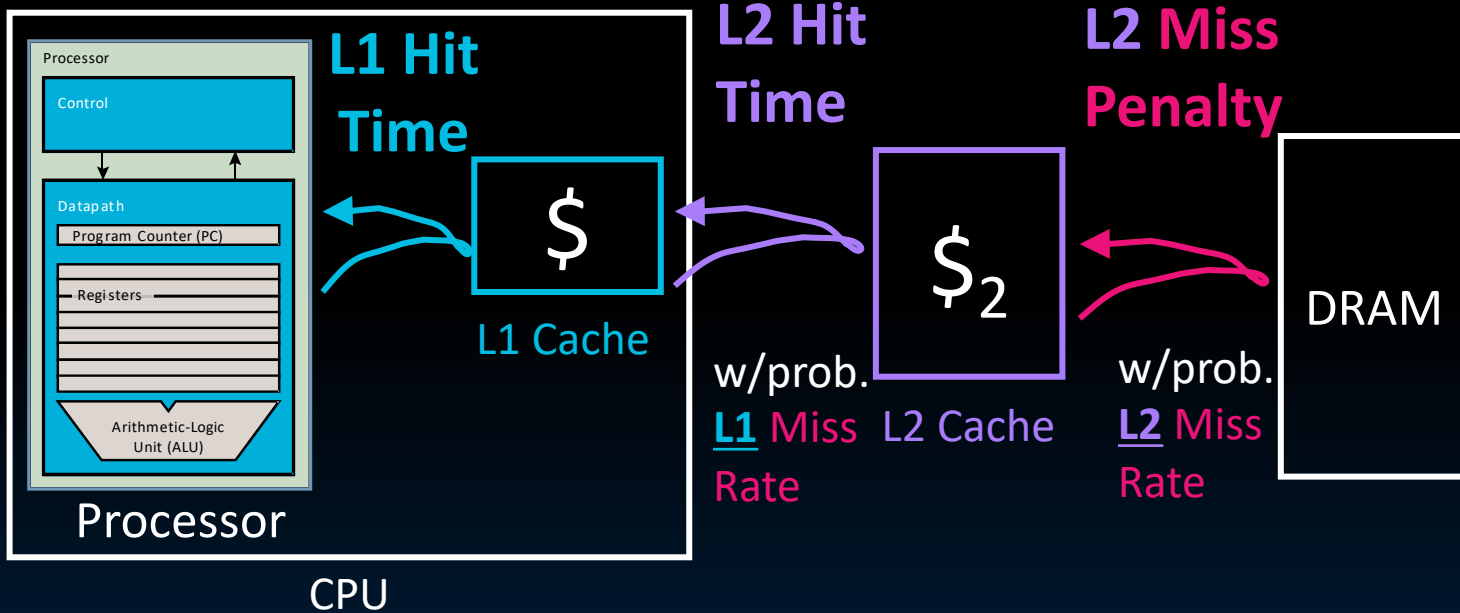


$$\begin{aligned}
 AMAT &= Hit\ Rate \cdot (Hit\ Time) \\
 &\quad + Miss\ Rate \cdot (Hit\ Time + Miss\ Penalty) \\
 &= Hit\ Time + Miss\ Rate \cdot (Miss\ Penalty) \\
 &= (1) + .05(20) = 2
 \end{aligned}$$

AMAT is Recursive

Average Access
Memory Time

$$AMAT = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$



$$AMAT = (\text{L1 Hit Time}) +$$

$$\text{L1 Miss Rate} \cdot (\text{L1 Miss Penalty})$$

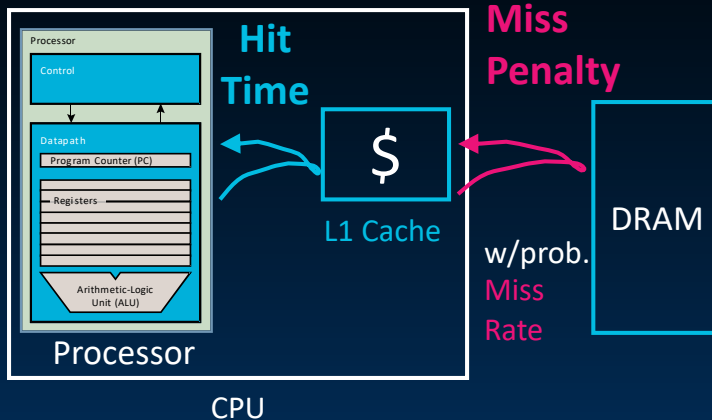
[typos updated during lecture]

$$\text{L2 Hit Time} + \text{L2 Miss Rate} \cdot (\text{L2 Miss Penalty})$$

Example: without L2 cache

$$AMAT = Hit\ Time + Miss\ Rate \times Miss\ Penalty$$

- Without L2 cache, assume:
 - L1 Hit Time = 1 cycle
 - L1 Miss rate = 5%
 - L1 Miss Penalty = 200 cycles
- Avg mem access time
 - $= 1 + 0.05 \times 200$
 - $= \underline{11\text{ cycles}}$



Example: with L2 cache

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

- Without L2 cache, assume:
 - L1 Hit Time = 1 cycle
 - L1 Miss rate = 5%
 - L1 Miss Penalty = 200 cycles
- Avg mem access time

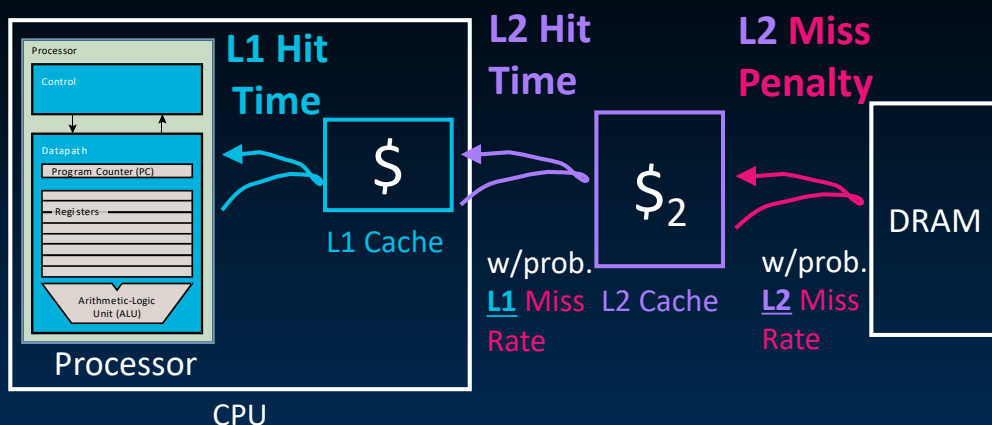
$$= 1 + 0.05 \times 200$$

$$= \underline{11 \text{ cycles}}$$

- With L2 cache, assume:
 - L1 Hit Time = 1 cycle
 - L1 Miss rate = 5%
 - L2 Hit Time = 5 cycles
 - L2 Miss rate = 15% (← % L1 misses that also miss L2)
 - L2 Miss Penalty = 200 cycles
- L1 miss penalty = $5 + 0.15 \times 200 = 35$
- Avg mem access time

$$= 1 + 0.05 \times 35$$

$$= \underline{2.75 \text{ cycles}}$$



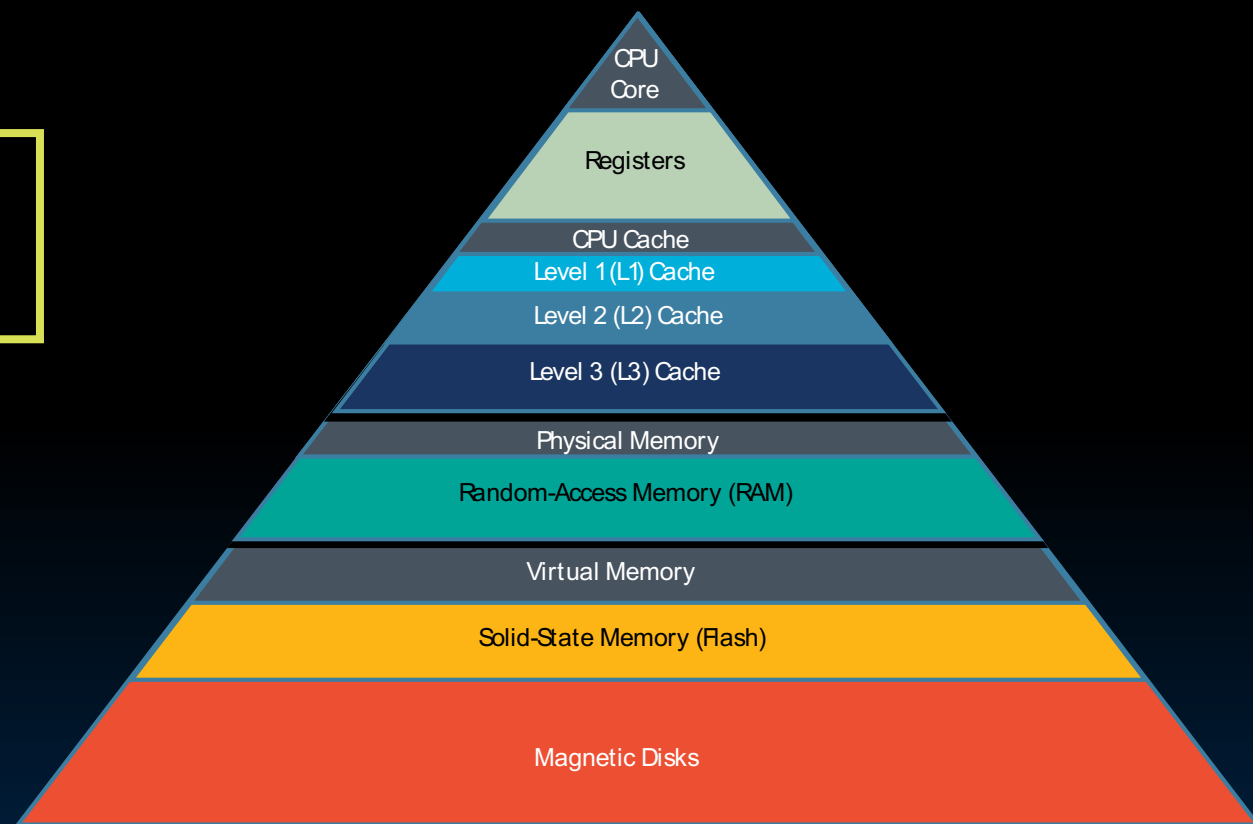
4x faster with L2 cache!
(2.75 vs. 11)

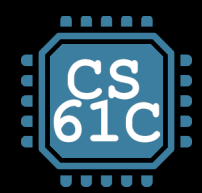
Typical Scale and Reducing Miss Rate

- **L1 cache** (\$)
 - size: tens of KB
 - hit time: complete in one clock cycle
 - miss rates: 1-5%
- **L2 cache** (\$₂)
 - size: hundreds of KB
 - hit time: few clock cycles
 - miss rates: 10-20%
- The **L2** miss rate is the fraction of **L1** misses that also miss in **L2**.
 - Why so high? (more later)
- Ways to reduce miss rate:
- Larger cache
 - limited by cost and technology
 - hit time of first level cache < cycle time (bigger caches are slower)
- More places in the cache to put each block of memory
 - Cache design! Up next!!

How is the Memory Hierarchy Managed?

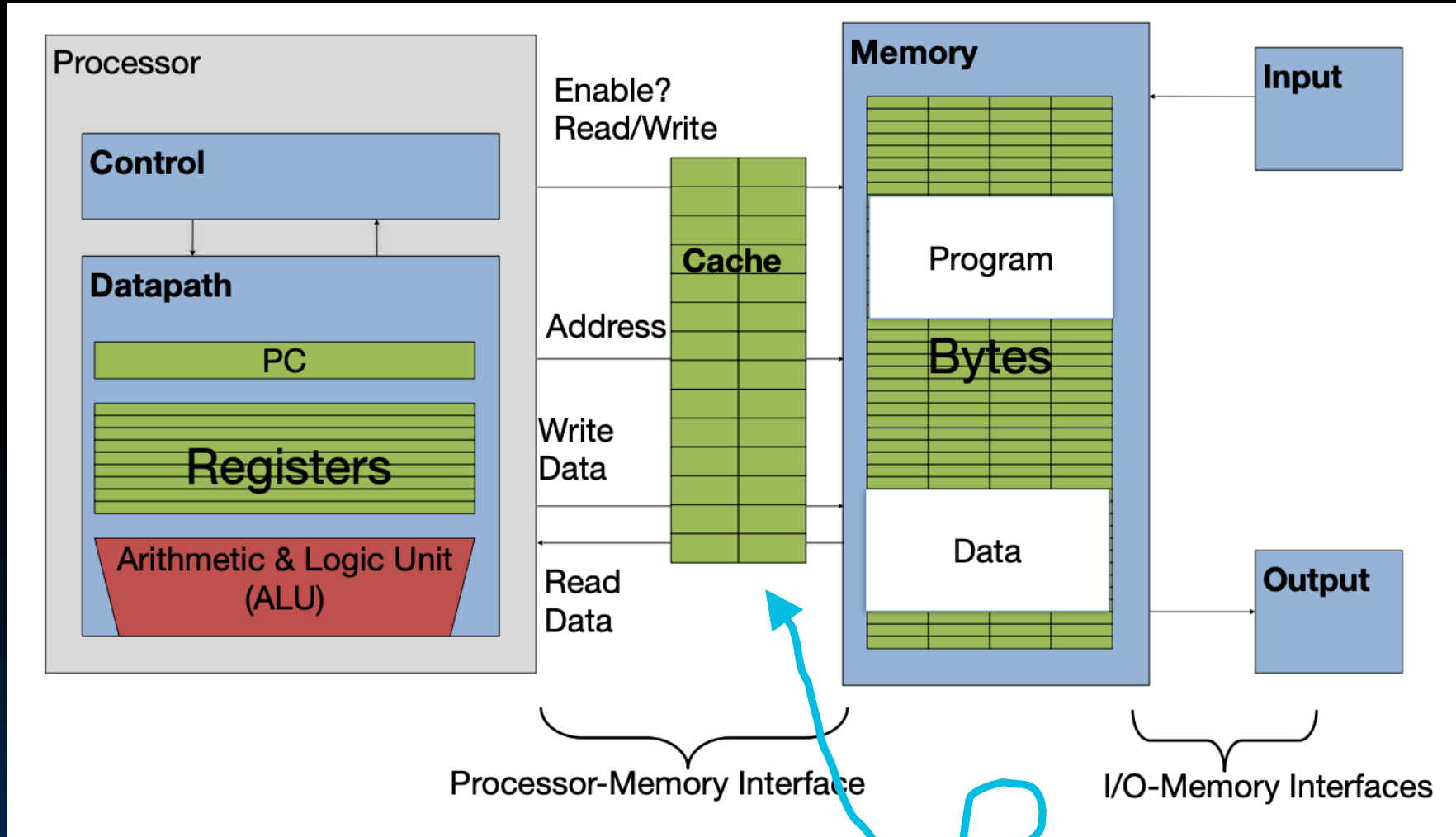
- registers \leftrightarrow memory
 - By compiler (or assembly level programmer)
- cache \leftrightarrow main memory
 - By the cache controller hardware
- main memory \leftrightarrow disks (secondary storage)
 - By the operating system (virtual memory)
 - Virtual to physical address mapping assisted by the hardware ('translation lookaside buffer' or TLB)
 - By the programmer (files) ↗ also a type of cache





Cache Design

- AMAT: Average Memory Access Time
- Cache Design
- Fully Associative Cache
- Warming up the Fully Associative Cache
- Block Replacement Policies
- Write Policies



Blocks (i.e., lines) of data are copied from memory to the cache.

- Load word instruction:
`lw t0 0(t1)`

$t1$ `0x12F0`

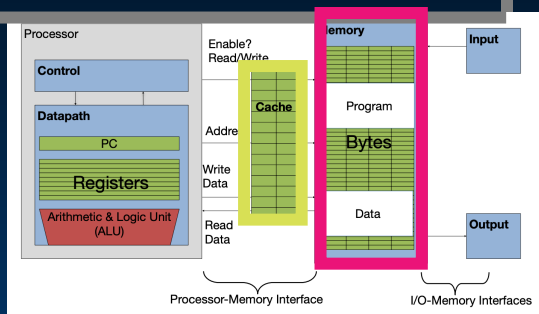
`Memory[0x12F0] = 1234`

Memory access w/o cache:

1. Processor issues address `0x12F0` to **memory**
2. **Memory** reads 1234 @ address `0x12F0`
3. **Memory** sends 1234 to Processor
4. Processor loads 1234 into register `t0`

Memory access **with cache**:

1. Processor issues address `0x12F0` to **cache**
2. **Cache** checks for copy of data, addr. `0x12F0`
3. If hit (finds match): **cache** reads 1234, sends 1234 to processor
If miss (no match): **cache** sends address `0x12F0` to **Memory**
 1. **Memory** reads 1234 @ address `0x12F0`
 2. **Memory** sends block w/1234 to cache
 3. Cache replaces some block to store 1234, addr. `0x12F0`
4. **Cache** sends 1234 to processor
4. Processor loads 1234 into register `t0`



Memory Access with and without Cache

- Load word instruction:
`lw t0 0(t1)`

t1 0x12F0

Memory[0x12F0] = 1234

Memory access w/o cache:

- Processor issues address 0x12F0 to **memory**
- Memory** reads 1234 @ address 0x12F0
- Memory** sends block w/1234 to processor
- Processor loads 1234 into register t0

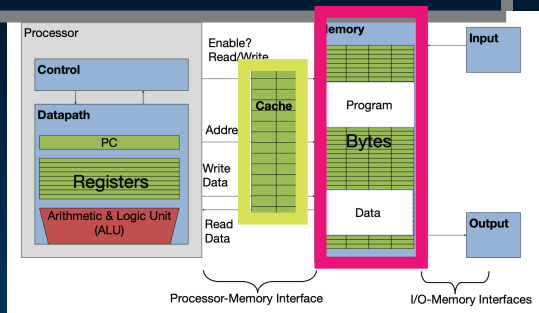
Need (2): value,
Need (3): value

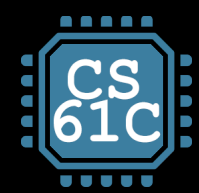
accessible by word
Processor

Memory access **with cache**:

Need (1): address

- Processor issues address 0x12F0 to **cache**
- Cache** checks for copy of data, addr. 0x12F0
- If hit (finds match): **cache** reads 1234, sends 1234 to processor
If miss (no match): **cache** sends address 0x12F0 to **Memory**
 - Memory** reads 1234 @ address 0x12F0
 - Memory** sends block w/1234 to cache
 - Cache replaces some block to store 1234, addr. 0x12F0
 - Cache** sends 1234 to processor
- Processor loads 1234 into register t0





Cache Design: Placement Policies

**Fully
Associative
Cache**

Put a new
block
anywhere

(today)



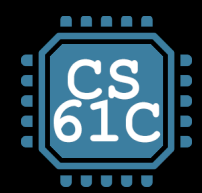
Set-Associative
Cache

(out of scope)

**Direct
Mapped
Cache**

Put a new block in
one specific place

(next time)



Fully Associative Cache

- AMAT: Average Memory Access Time
- Cache Design
- Fully Associative Cache
- Warming up the Fully Associative Cache
- Block Replacement Policies
- Write Policies

Fully Associative Cache

- **Placement policy:** The data can be stored anywhere in the cache.
- For a given memory address:
 - Split into two fields: the **tag** and the **offset**
- To check if a block (i.e., cache line) has our data:
 - Check the tag
 - Then, get the correct byte offset
- Fully Associative:
 - Any block can potentially match our tag, i.e., check all tags for the correct block.

Tag	Data			
	11	10	01	00

Memory address:



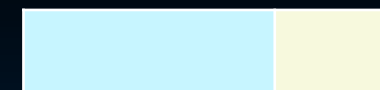
tag **offset**
(i.e., byte offset)

Warming up the Cache: Valid Bit

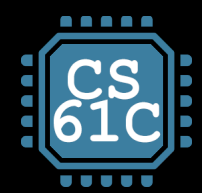
- When starting up a program, the cache necessarily does not have valid information for the program.
- We therefore need an indicator (i.e., **flag**) to tell if each entry is **valid** for this particular program:
 - 1 = valid, 0 = invalid

Valid	Tag	Data			
id		11	10	01	00

Memory address:



tag **offset**
(i.e., byte offset)



Warming up the Fully Associative Cache

- AMAT: Average Memory Access Time
- Cache Design
- Fully Associative Cache
- Warming up the Fully Associative Cache
- Block Replacement Policies
- Write Policies

Caching: The basis of the memory hierarchy [review]

- A **cache** contains copies of data that are being used.
- A cache works on the principles of **temporal and spatial locality**:

	Temporal locality	Spatial locality
Idea	If we use it now, chances are that we'll want to use it again soon.	If we use a piece of memory, chances are we'll use the neighboring pieces soon.
Library Analogy	We keep a book on the desk while we check out another book.	If we check out a book's vol. 1 while we're at it, we'll also check out vol. 2.
Memory	<p>If a memory location is referenced, then it will tend to be referenced again soon.</p> <p>Therefore, keep most recently accessed data items closer to the processor.</p>	<p>If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon.</p> <p>Move blocks consisting of contiguous words closer to the processor .</p>



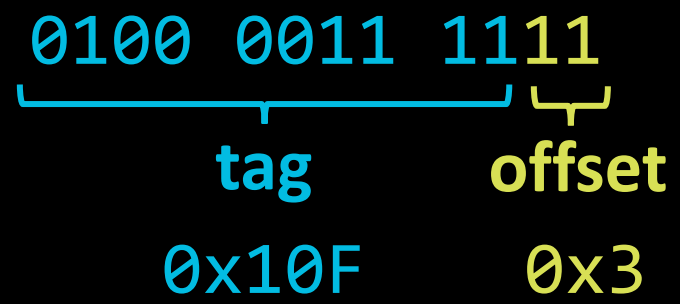
Cache Temperatures

- Cold
 - Cache empty
- Warming
 - Cache filling with values you'll hopefully be accessing again soon
- Warm
 - Cache is doing its job, fair % of hits
- Hot
 - Cache is doing very well, high % of hits

Warming up the Fully Associative Cache

Suppose the cache starts **cold**.

1. Load byte 0x43F



Val id	Tag	Data			
		11	10	01	00
0
0
0
0

a. **Cache miss!** Tag 0x10F is not valid.



Warming up the Fully Associative Cache

Suppose the cache starts **cold**.

1. Load byte 0x43F

0100 0011 1111

tag

offset

0x10F 0x3

0x43C

0x43F

0100 0011 1100 0100 0011 1111

- Cache miss! Tag 0x10F is not valid.
- Load into cache the **4-byte block** from 0x43C to 0x43F. Mark **valid bit**.
- Read byte at 0x3 **offset** and return to processor.

[typos updated during lecture]

Val id	Tag	Data			
		11	10	01	00
1	0x10F
0
0
0

Memory address:

11	2	1 0
(10b)		(2b)

tag

offset

Spatial locality: Even if we only read in one byte, loading from memory will load the **full block** (here, 4B) associated with a tag tag.

Warming up the Fully Associative Cache

Suppose the cache starts **cold**.

1. Load byte 0x43F 0x10F, 0x3

2. Load byte 0x5E2 0x178, 0x2

0x5E0

0x5E3

0101 1110 0000

0101 1110 0011

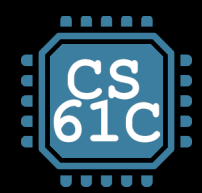
- Cache miss!** Tag 0x178 is not valid.
- Load into cache the **4-byte block** from 0x5E0 to 0x5E3. Mark **valid bit**.
- Read byte at **0x2 offset** and return to processor.

[typos updated during lecture]

Val id	Tag	Data			
		11	10	01	00
1	0x10F
1	0x178
0
0

Memory address:	11	2	1 0
	(10b)		(2b)

tag offset



Terminology

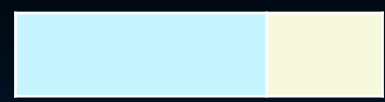
- AMAT: Average Memory Access Time
- Cache Design
- Fully Associative Cache
- Warming up the Fully Associative Cache
- Block Replacement Policies
- Write Policies

Terminology

- Cache block/line: A single entry in the cache
- Block size / line size: # bytes per cache block
- Capacity
 - Total # data bytes that can be stored in a cache
- Tag
 - Identifies data stored at a given cache block
- Valid bit
 - Indicates if data stored at a cache block is valid

Valid	Tag	Data			
id		11	10	01	00

Memory address:



tag **offset**
(i.e., byte offset)

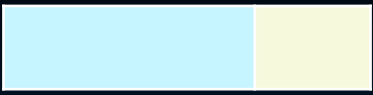


Terminology

- Cache block/line: A single entry in the cache
- Block size / line size: # bytes per cache block
- Capacity
 - Total # data bytes that can be stored in a cache
- Tag
 - Identifies data stored at a given cache block
- Valid bit
 - Indicates if data stored at a cache block is valid

Valid id	Tag	Data			
		11	10	01	00

Memory address:



tag **offset**
(i.e., byte offset)

Suppose we have the following cache, for 12b addresses.

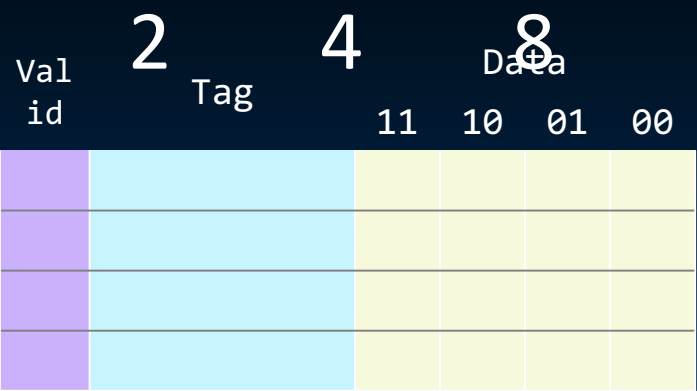
- | | |
|--|-------------------|
| 1. What is the block size / line size, in bytes? | A. 2 |
| 2. What is the capacity, in bytes? | B. 4 |
| 3. For a 12b address, how many bits are the byte offset? | C. 8 |
| 4. For a 12b address, how many bits are the tag? | D. 10 |
| | E. 16 |
| | F. Something else |

Fill in the blank

[activity slide]

Suppose we have the following cache, for 12b addresses.

	A.	B.	C.	D.	E.	F.
1. What is the block size / line size, in bytes?	2	4	8	10	16	Other
2. What is the capacity, in bytes?	2	4	8	10	16	Other
3. For a 12b address, how many bits are the byte offset?	2	4	8	10	16	Other
4. For a 12b address, how many bits are the tag?	2	4	8	10	16	Other



For each of the following questions, place a pin for the correct answer.

0



Fill in the blank

[activity slide]

Suppose we have the following cache, for 12b addresses.

	A.	B.	C.	D.	E.	F.
1. What is the block size / line size, in bytes?	2	4	8	10	16	Other
2. What is the capacity, in bytes?	2	4	8	10	16	Other
3. For a 12b address, how many bits are the byte offset?	2	4	8	10	16	Other
4. For a 12b address, how many bits are the tag?	2	4	8	10	16	Other

Terminology, Part 1 (for 12b addresses!)

Cache block/line: A single entry in the cache

1. Block size / line size: # bytes per cache block
4 bytes

2. Capacity 4 x 4 bytes = **16 bytes**

▪ Total # data bytes that can be stored in a cache

3. Offset $\log_2(\text{block size}) = \mathbf{2 \text{ bits}}$

▪ Identifies byte address of data stored at a given cache block

4. Tag # address bits - # offset bits = **10 bits**

▪ Identifies data stored at a given cache block

Valid	Tag	Data			
id		11	10	01	00

Memory address:	11	2	1 0

tag **offset**
 (i.e., byte offset)

Warming up the Fully Associative Cache

Suppose the cache starts **cold**.

1. Load byte 0x43F 0x10F, 0x3
 2. Load byte 0x5E2 0x178, 0x2
 3. Load word 0x824 0x209, 0x0
-

Val id	Tag	Data			
		11	10	01	00
1	0x10F
1	0x178
1	0x209
0

- Cache miss!** Tag 0x209 is not valid.
- Load into cache the **4-byte block** from 0x20900 to 0x20911. Mark **valid bit**.
- Read word starting at **0x0 offset** and return to processor.

Memory address:	11	2	1 0
	(10b)		(2b)
	tag		offset