National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 10: Cache I

CS10014 Computer Organization

Department of Computer Science
Tsung Tai Yeh
Thursday: 1:20 pm– 3:10 pm
Classroom: EC-022

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS252 at ETHZ
    - https://safari.ethz.ch/digitaltechnik/spring2023
  - CSCE 513 at University of South Carolina
    - https://passlab.github.io/CSCE513/

Outline

- Memory Hierarchy
- Memory Caching
- Cache Basics
- Direct-Mapped Cache
- Read Data in Direct-Mapped Cache
- Directed-Mapped Cache Hardware

Review: Pipelining

- Pipeline challenge is hazards
    - Forwarding helps with many data hazards
    - Delayed branch helps with control hazard in the 5 stage pipeline
    - Data hazards with loads => Load delay slot
        - Interlock => “smart” CPU has HW to detect if conflict with instruction following load, if so it stalls
    - More aggressive performance
        - Superscalar
        - Out-of-order execution

Takeaway Questions

- Assume two processors
  - Unpipelined: 1GHz
  - Pipelined: 4GHz
- Before pipelining
  - Program took 1 second to execute
  - 1% of instructions were mis-predicted branches
  - 5% of instructions triggered load-delay stalls
- What is the performance of the pipelined processor? (seconds)

Takeaway Questions

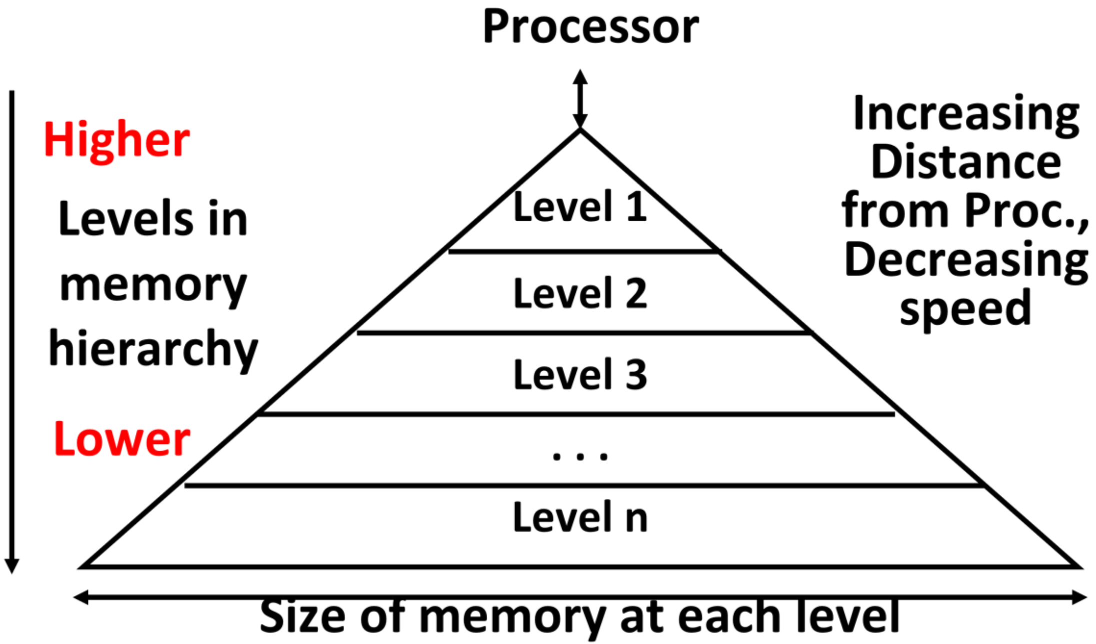
- Assume two processors
  - Unpipelined: 1GHz
  - Pipelined: 4GHz
- Before pipelining
  - Program took 1 second to execute
  - 1% of instructions were mis-predicted branches
  - 5% of instructions triggered load-delay stalls
- What is the performance of the pipelined processor? (seconds)
  - .25 + (.01 * 1 * 2) + (.05 * 1 * 1) = 0.37 seconds

Outline

- Memory Hierarchy
- Memory Caching
- Cache Basics
- Direct-Mapped Cache
- Read Data in Direct-Mapped Cache
- Directed-Mapped Cache Hardware

Memory Hierarchy (1/3)

- As we move to deeper levels, the latency goes up and price per bit goes down


Processor
Higher
Levels in memory hierarchy
Lower
Level 1
Level 2
Level 3
. . .
Level n
Increasing Distance from Proc., Decreasing speed
Size of memory at each level

Memory Hierarchy (2/3)

- **Processor**
  - Holds data in register file (~100 bytes)
  - Registers accessed on nanosecond timescale
- **Memory** (we'll call "main memory")
  - More capacity than registers (~Gbytes)
  - Access time ~50-100 ns
  - Hundreds of clock cycles per memory access
- **Disk**
  - HUGE capacity
  - Very slow: runs ~milliseconds

Memory Hierarchy (3/3)

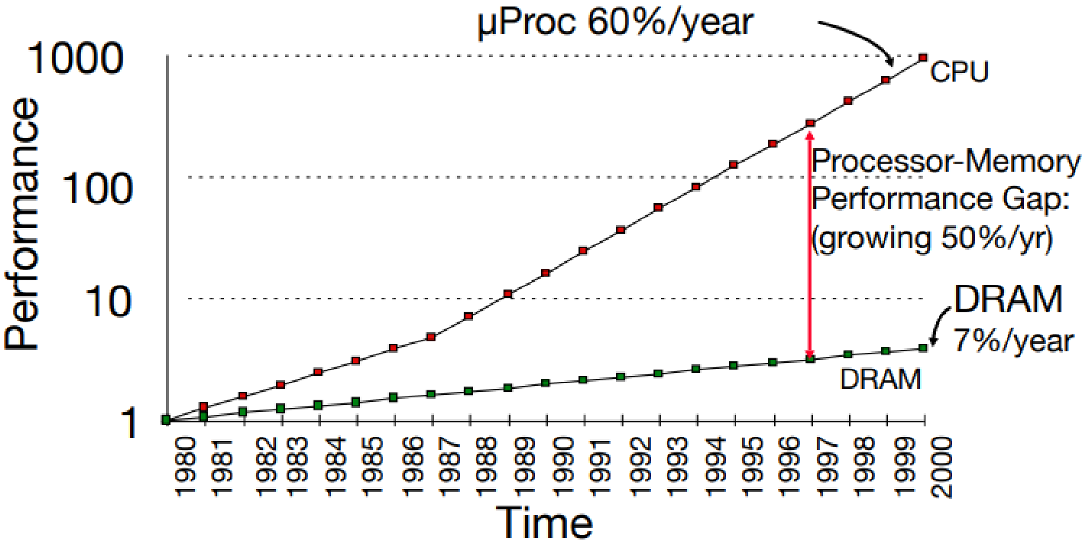
- If level closer to processor, it is:
  - Smaller
  - Faster
  - More expensive
  - Subset of lower levels (contains most recently used data)
- Lowest level (usually disk) contains all available data
- Memory hierarchy presents the processor with the illusion of a very large & fast memory

Outline

- Memory Hierarchy
- Memory Caching
- Cache Basics
- Direct-Mapped Cache
- Read Data in Direct-Mapped Cache
- Directed-Mapped Cache Hardware

Memory Caching (1/6)

- Slow DRAM access has disastrous impact on CPU perf.
  - 1980 processor exec. ~one inst. in the same time as DRAM access
  - 2015 processor exec. ~1000 insts. In the same time as DRAM access



Memory Caching (2/6)

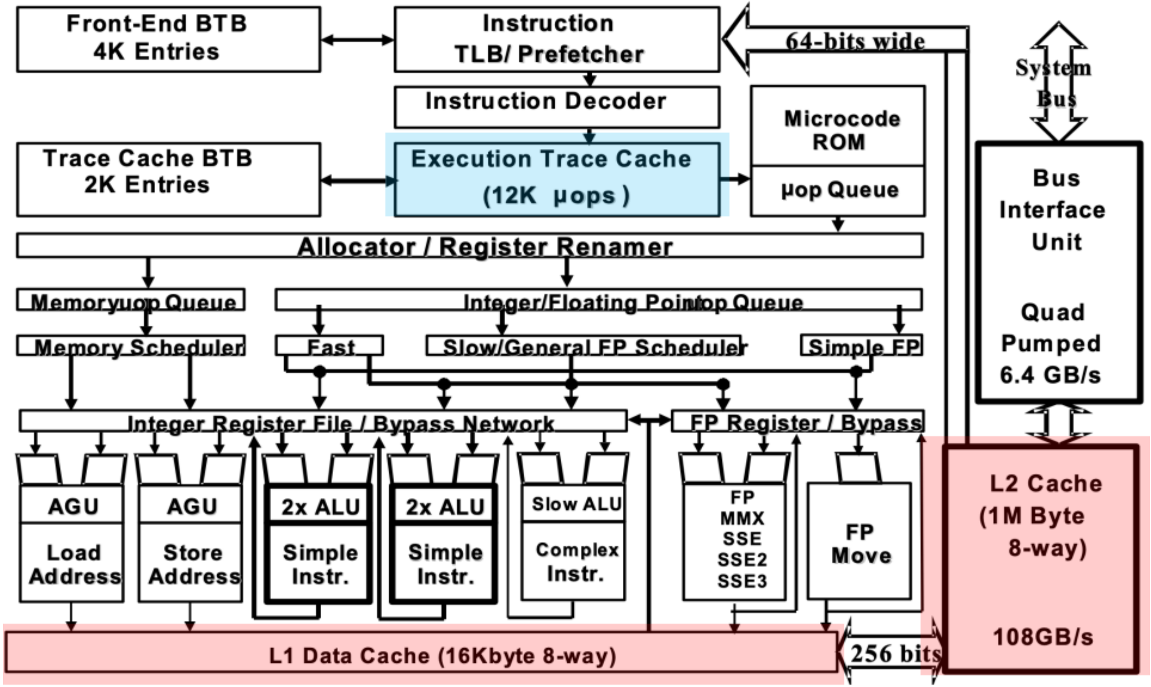
- Mismatch between processor and memory speeds
  - Leads us to add a new level: a memory cache
- **The memory cache**
  - Implemented with same IC processing technology as the CPU (usually integrated on the same chip)
  - Faster but more expensive than DRAM
  - Cache is a copy of a subset of main memory
  - Most processors have separate caches for instructions and data

Memory Caching (3/6)

- **Cache** contains copies of data in memory being used
- **Memory** contains copies of data on disk being used
- Caches work on principles of temporal and spatial locality
    - **Temporal locality**: if we use it now, chances are we’ll want to use it again soon
        - Data elements accessed **in loops** (same data elements are accessed multiple times)
    - **Spatial locality**: if we use a piece of memory, chances are we’ll use the neighboring pieces soon
        - Data elements accessed **in array** (each time different or just next element is being accessing)

Memory Caching (4/6)

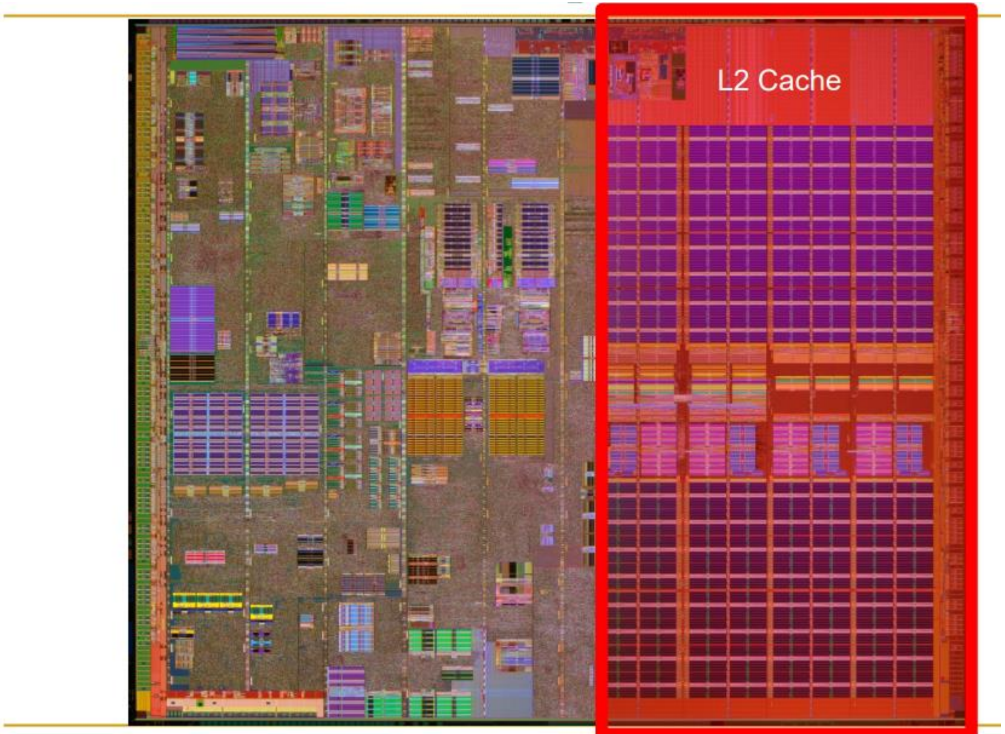
- **Intel Pentium 4 Example**



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2004.

Memory Caching (5/6)

- **Intel Pentium 4 Example**



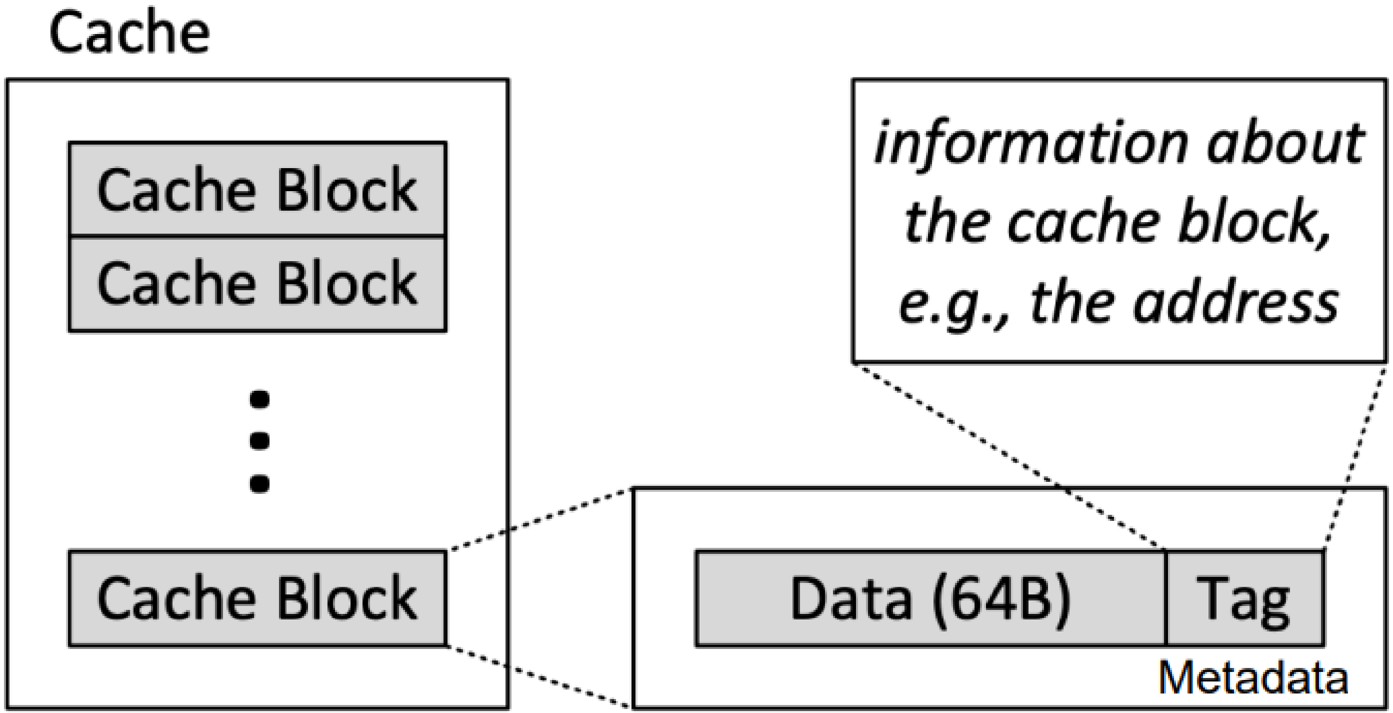
Memory Caching (6/6)

- **Intel Pentium 4 Example**
    - 90 nm, P4, 3.6 GHz
    - **L1 D-cache**
        - C1 = 16 kB
        - T1 = 4 cycle int/ 9 cycle fp
    - **L2 D-cache**
        - C2 = 1024 kB
        - T2 = 18 cycle int / 18 cycle fp
    - **Main memory**
        - T3 = ~50 ns or 180 cycle

Outline

- Memory Hierarchy
- Memory Caching
- Cache Basics
- Direct-Mapped Cache
- Read Data in Direct-Mapped Cache
- Directed-Mapped Cache Hardware

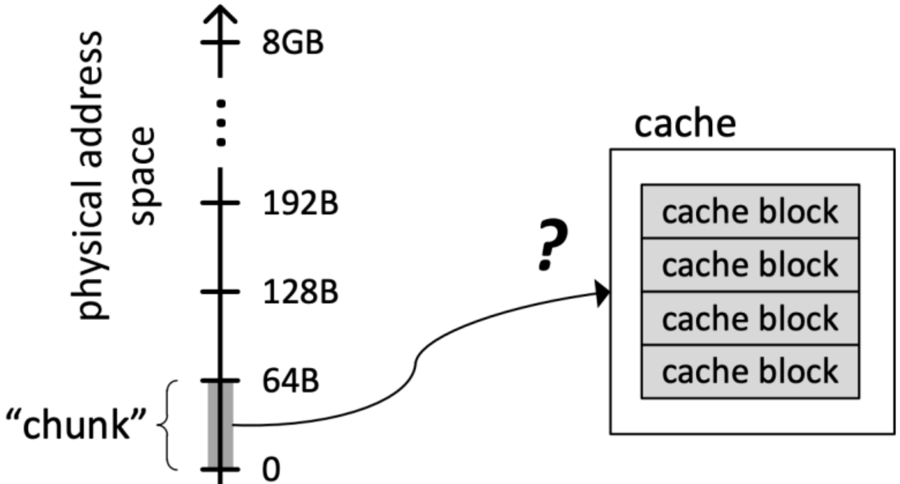
Cache Basics (1/8)



Kim & Mutlu, "Memory Systems," Computing Handbook, 2014
https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

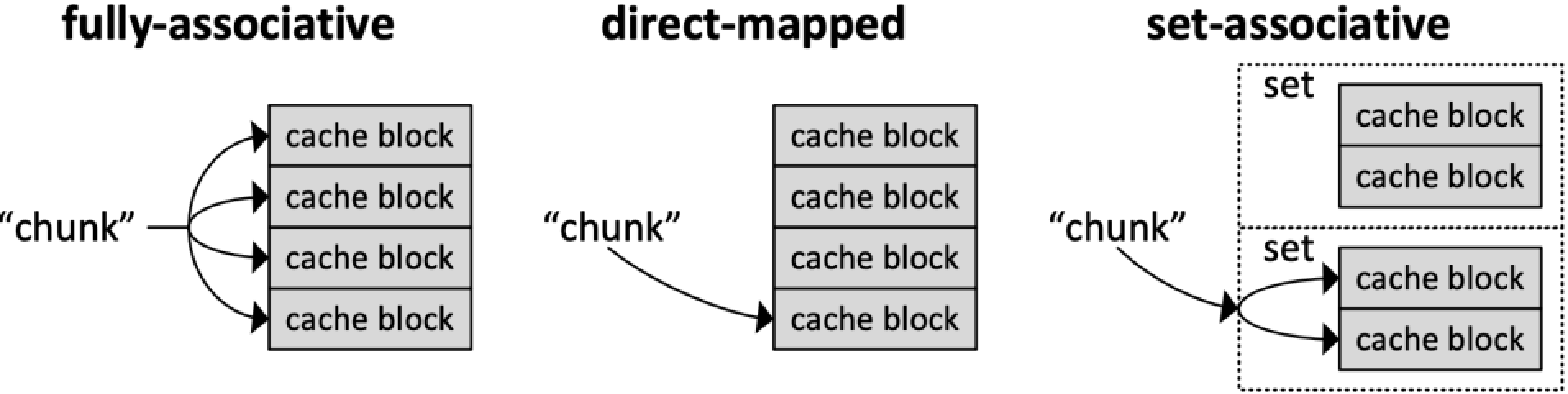
Cache Basics (2/8)

- **A key question**
  - How to map chunks of the main memory address space to blocks in the cache?
  - Which location in cache can a given “main memory chunk” be placed in?



Cache Basics (3/8)

- **Cache associativity**
  - One **set** can contain multiple cache blocks



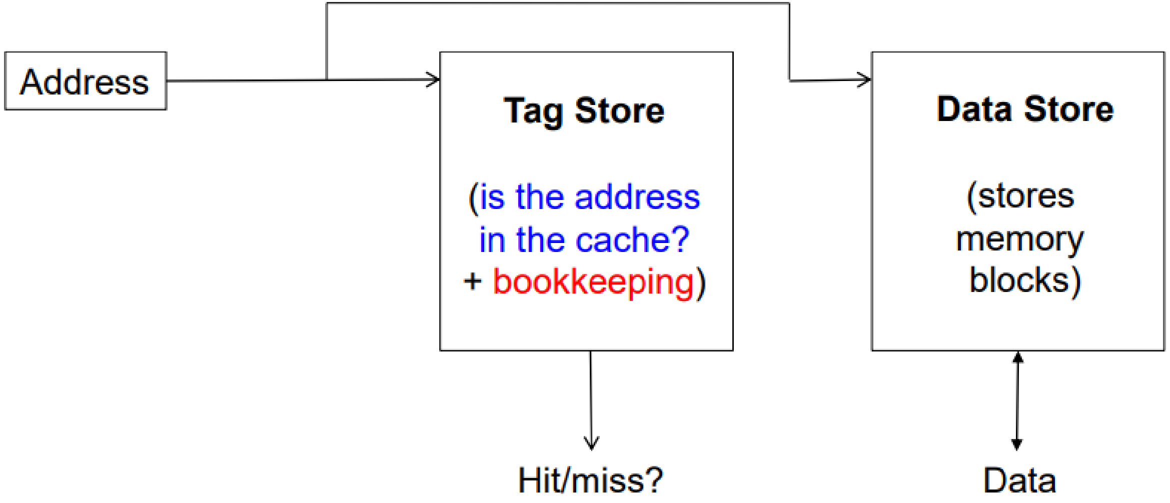
Kim & Mutlu, "Memory Systems," Computing Handbook, 2014

Cache Basics (4/8)

- **Block (line):** Unit of storage in the cache
  - Memory is logically divided into blocks that map to potential locations in the cache
- When reading memory, 3 things can happen
  - **Cache HIT**:
    - Cache block is valid and contains proper address, so read desired word
  - **Cache MISS**:
    - Nothing in cache in appropriate block, so fetch from memory
  - **Cache miss, block replacement**
    - Wrong data is in cache at appropriate block, so discard it and fetch desired data from memory

Cache Basics (5/8)

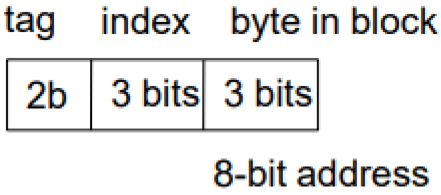
- **Cache hit rate** = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- **Average memory access time (AMAT)**
  - = (hit-rate * hit-latency) + (miss-rate * miss-latency)



Cache Basics (6/8)

- **Types of Misses**
  - **Compulsory**: First time data is accessed
  - **Capacity**: cache too small to hold all data of interest
  - **Conflict**: data of interest maps to same location in cache
  - **Miss penalty**: time it takes to retrieve a block from lower level of hierarchy

Cache Basics (7/8)

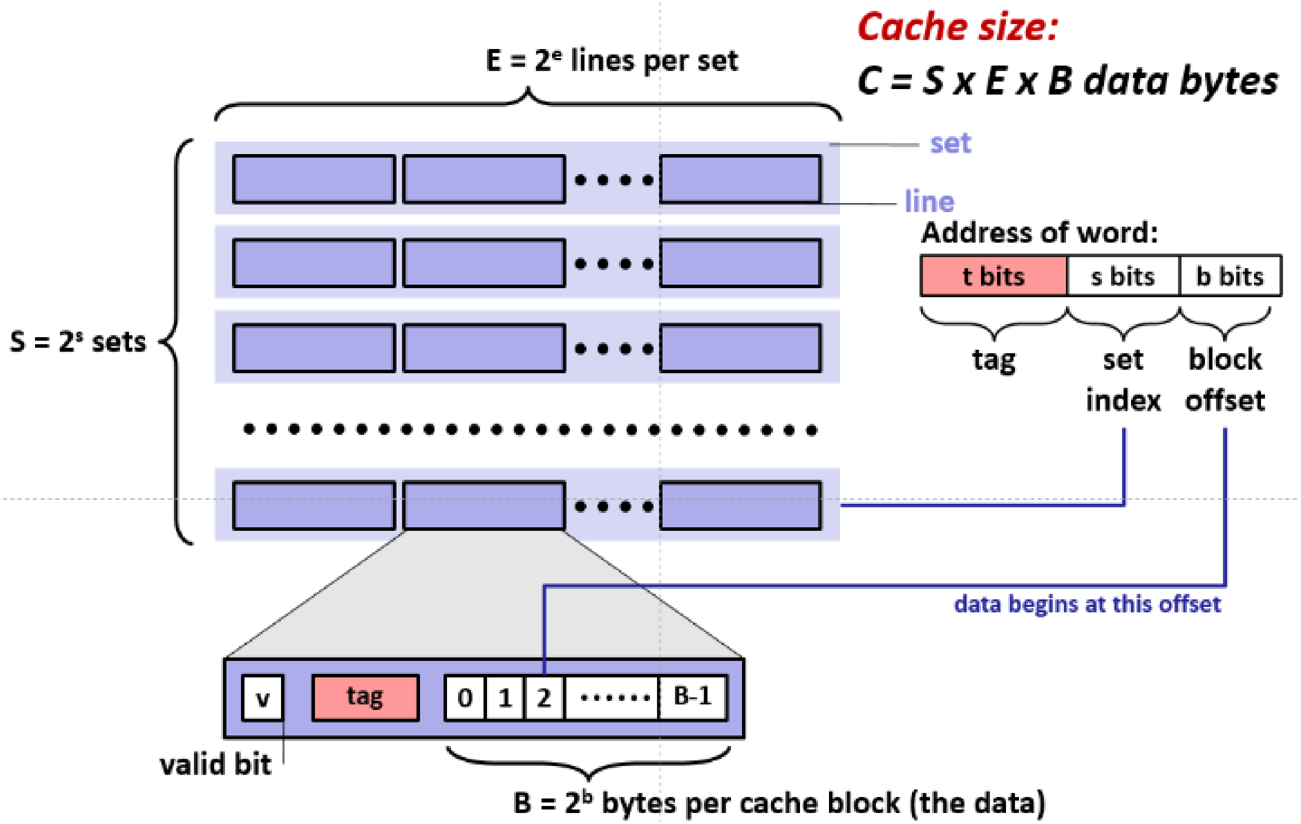


tag	index	byte in block
2b	3 bits	3 bits

8-bit address

- Each block address maps to a potential location in the cache, determined by the index bits in the address
- **Index**
  - Specifies the cache index (which “row”/block of the cache we should look in)
- **Offset**
  - Once we’ve found correct block, specifies which byte within the block we want
- **Tag**
  - The remaining bits after offset and index are determined
  - These are used to distinguish between all the memory address that map to the same location

Cache Basics (8/8)



Cache size:

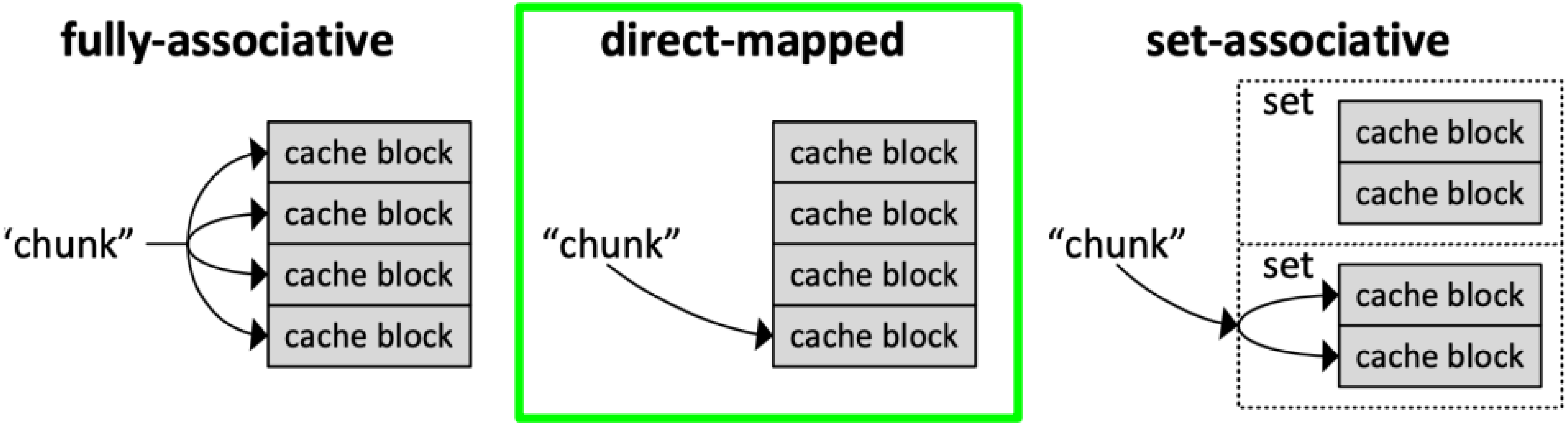
C = S x E x B data bytes

Outline

- Memory Hierarchy
- Memory Caching
- Cache Basics
- Direct-Mapped Cache
- Read Data in Direct-Mapped Cache
- Directed-Mapped Cache Hardware

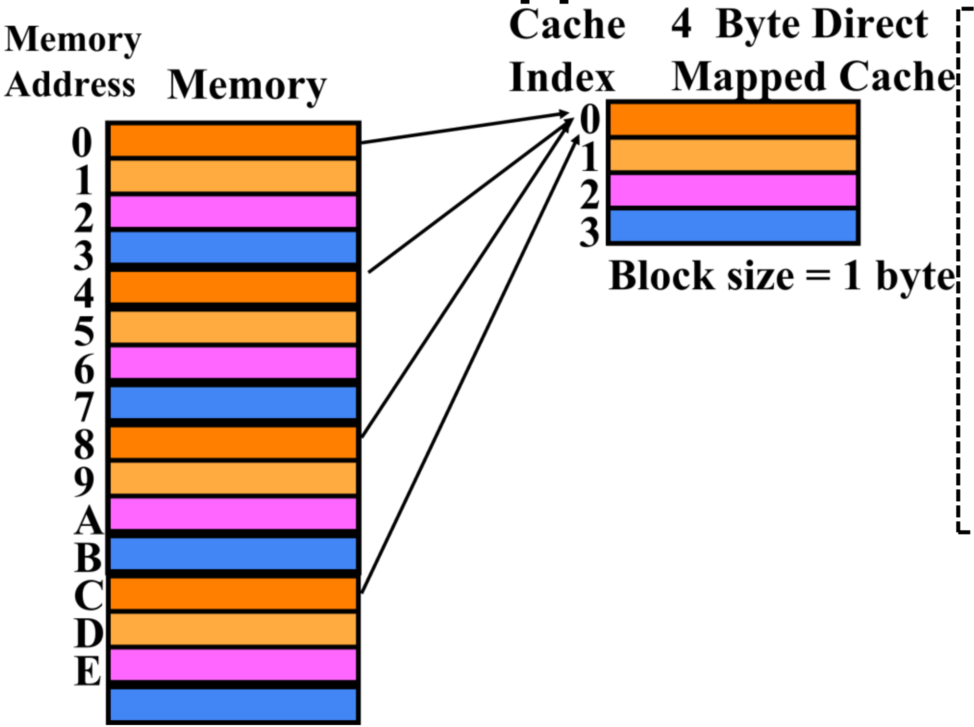
Direct-Mapped Cache (1/7)

- **Directed-mapped cache**
  - A given main memory block can be placed in **only one possible location** in the cache
  - Toy example: 256-byte memory, 64-byte cache, 8-byte blocks



Direct-Mapped Cache (2/7)

- In a **directed-mapped cache**



* Cache location 0 can be occupied by data from:

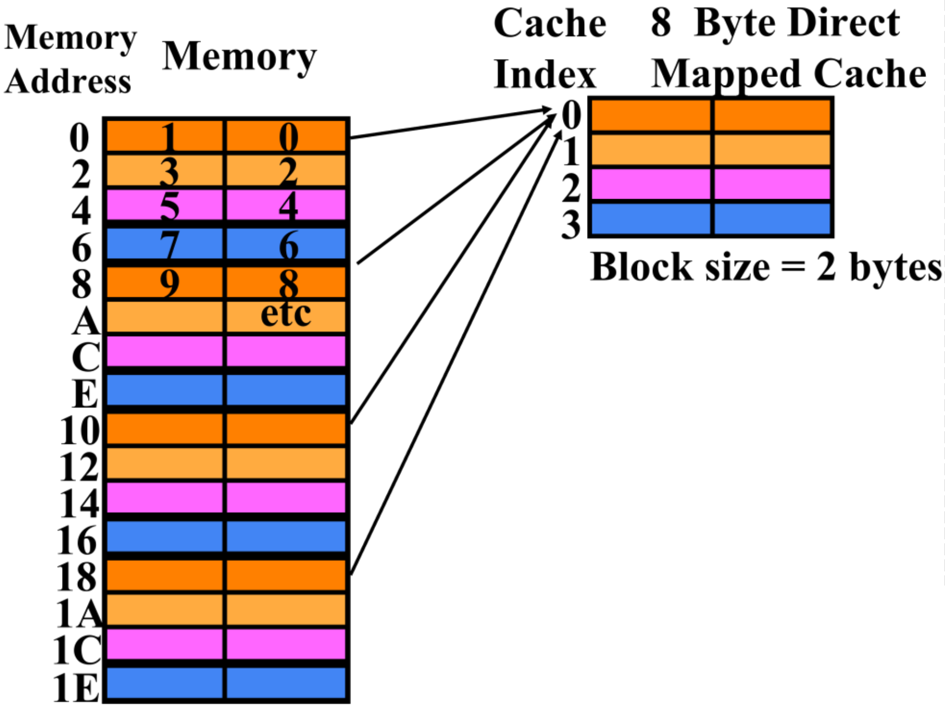
  * Memory location 0, 4, 8

  * 4 blocks => any memory location that is multiple of 4

* What if we want a block to be bigger than one byte?

Direct-Mapped Cache (3/7)

- In a **directed-mapped cache**



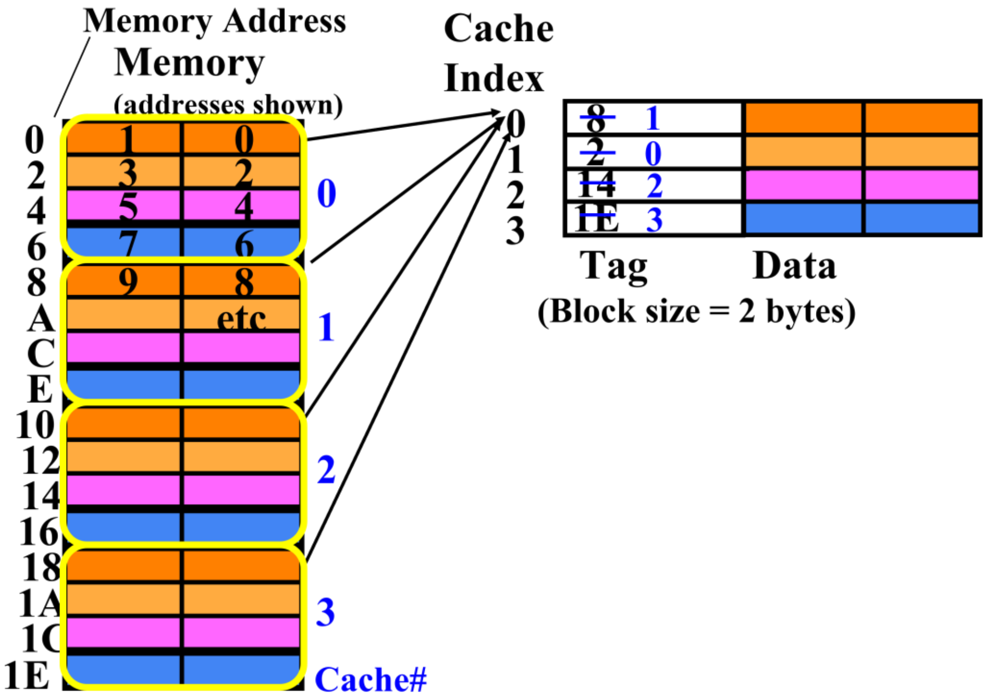
* When we ask for a byte, the system finds out the right block and loads it.

- * How does it know right block?
- * How do we select the byte?
- * E.g. Mem address 11101?

* How does it know WHICH colored block it originated from?

Direct-Mapped Cache (4/7)

- In a **directed-mapped cache**



* What should go in the tag?

* Do we need the entire address?

* What do all these tags have in common?

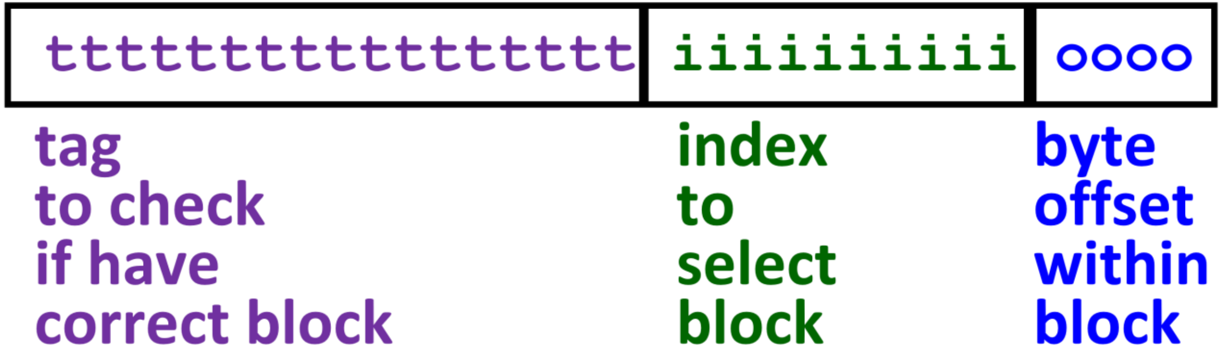
* What did we do with the immediate when we were branch addressing, always count by bytes?

* Why not count by cache #?

* It's useful to draw memory with the same width as the block size

Direct-Mapped Cache (5/7)

- In a **directed-mapped cache**
  - Multiple memory addresses map to the same cache index, how do we tell which one is in there?
  - What if we have a block size > 1 byte?
  - **Ans: divide memory address into three fields**



tttttttttttttttttt iiiiiiiiii oooo

tag
to check
if have
correct block

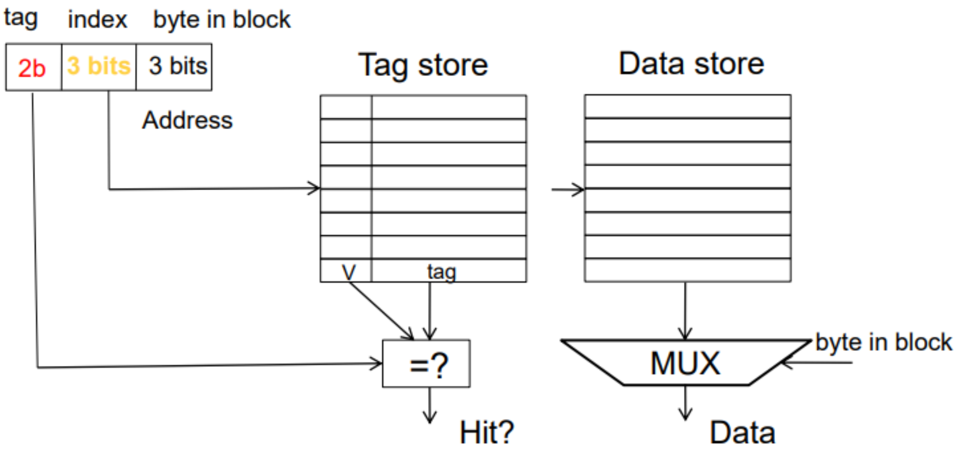
index
to
select
block

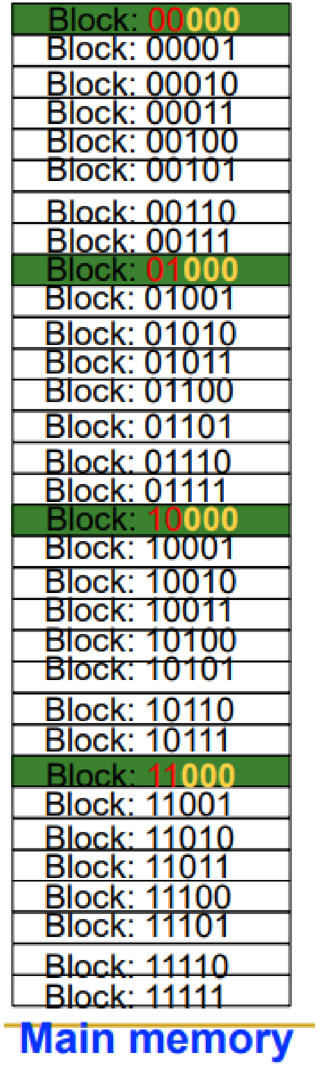
byte
offset
within
block

Direct-Mapped Cache (6/7)

- A byte-addressable main memory
  - 256 bytes, 8-byte blocks -> 32 blocks in memory
  - Assume cache: 64 bytes, 8 blocks
  - Directed-mapped: A block can go to only one location

Blocks with same index contend for the same cache location => cause conflict misses when accessed consecutively




Block: 00000
Block: 00001
Block: 00010
Block: 00011
Block: 00100
Block: 00101
Block: 00110
Block: 00111
Block: 01000
Block: 01001
Block: 01010
Block: 01011
Block: 01100
Block: 01101
Block: 01110
Block: 01111
Block: 10000
Block: 10001
Block: 10010
Block: 10011
Block: 10100
Block: 10101
Block: 10110
Block: 10111
Block: 11000
Block: 11001
Block: 11010
Block: 11011
Block: 11100
Block: 11101
Block: 11110
Block: 11111

Main memory

Direct-Mapped Cache (7/7)

- **Direct-mapped cache**
  - Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
  - One index -> one entry
  - Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index
    - Assume addresses A and B have the same index bits but different tag bits
    - A, B, A, B, A, B, A, B … -> conflict in the cache index
    - All accesses are conflict misses

Direct-Mapped Cache Example (1/3)

- Suppose we have a 8B of data in a direct-mapped cache with 2 byte blocks
- Determine the size of the tag, index, and offset fields if we are using a 32-bit architecture
  - **Offset**
    - Need to specify correct byte within a block
    - Block contains 2 bytes = 2^1 bytes
    - Need **1 bit** to specify correct byte

Direct-Mapped Cache Example (2/3)

- Suppose we have a 8B of data in a direct-mapped cache with 2 byte blocks
  - **Index (index into an “array of blocks”)**
    - Need to specify correct block in cache
    - # blocks/cache $= \dfrac{\text{bytes/cache}}{\text{bytes/block}}$

      $$= \frac{2^3 \text{ bytes/cache}}{2^1 \text{ bytes/block}}$$

      $$= 2^2 \text{ blocks/cache}$$
    - Need **2 bits** to specify this many blocks

Direct-Mapped Cache Example (3/3)

- Suppose we have a 8B of data in a direct-mapped cache with 2 byte blocks
  - Tag: use remaining bits as tag
  - Tag length = address length – offset – index
    = 32 – 1 – 2 bits
    = 29 bits
  - Why not full 32 bit address as tag?
    - All bytes within block need same address (4 bits)
    - Index must be same for every address within a block, so it's redundant in tag check, thus can leave off to save memory

The tag is leftmost 29 bits of memory address

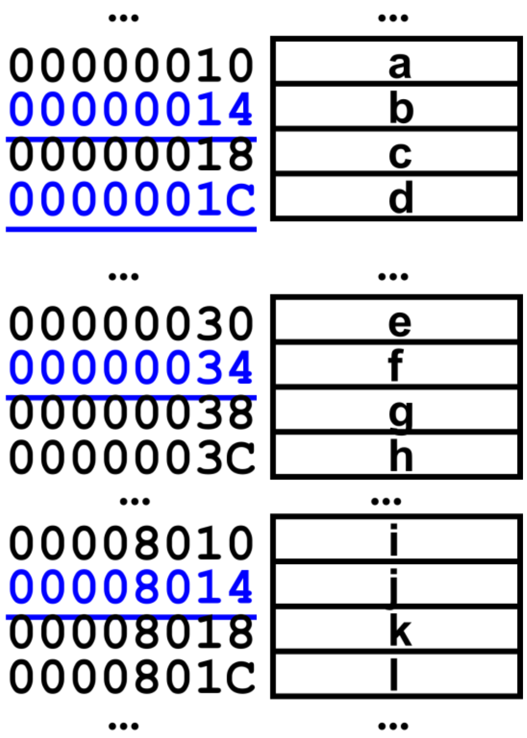
Outline

- Memory Hierarchy
- Memory Caching
- Cache Basics
- Direct-Mapped Cache
- Read Data in Direct-Mapped Cache
- Directed-Mapped Cache Hardware

Read Data in Direct-Mapped Cache (1/15)

- Ex. 16 KB of data, direct-mapped, 4 word block
- Read 4 addresses
  - 0x00000014
  - 0x0000001C
  - 0x00000034
  - 0x00008014

Memory



…	…
00000010	a
00000014	b
00000018	c
0000001C	d
…	…
00000030	e
00000034	f
00000038	g
0000003C	h
…	…
00008010	i
00008014	j
00008018	k
0000801C	l
…	…

Read Data in Direct-Mapped Cache (2/15)

- 4 addresses divided into

000000000000000000	0000000001	0100	0x00000014
000000000000000000	0000000001	1100	0x0000001C
000000000000000000	0000000011	0100	0x00000034
000000000000000010	0000000001	0100	0x00008014
Tag	Index	Offset	

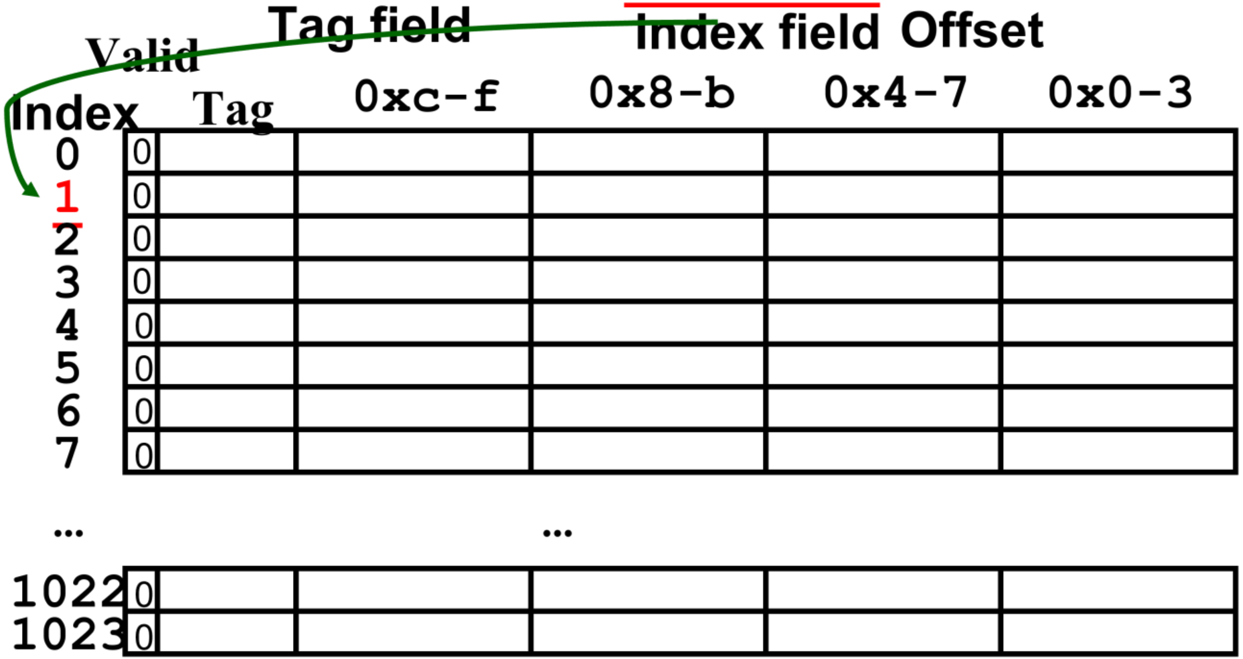
Read Data in Direct-Mapped Cache (3/15)

- 16 KB direct-mapped cache, 16B blocks
  - Valid bit: determines whether anything is stored in that row (when computer initially turned on, all entries invalid)

	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read Data in Direct-Mapped Cache (4/15)

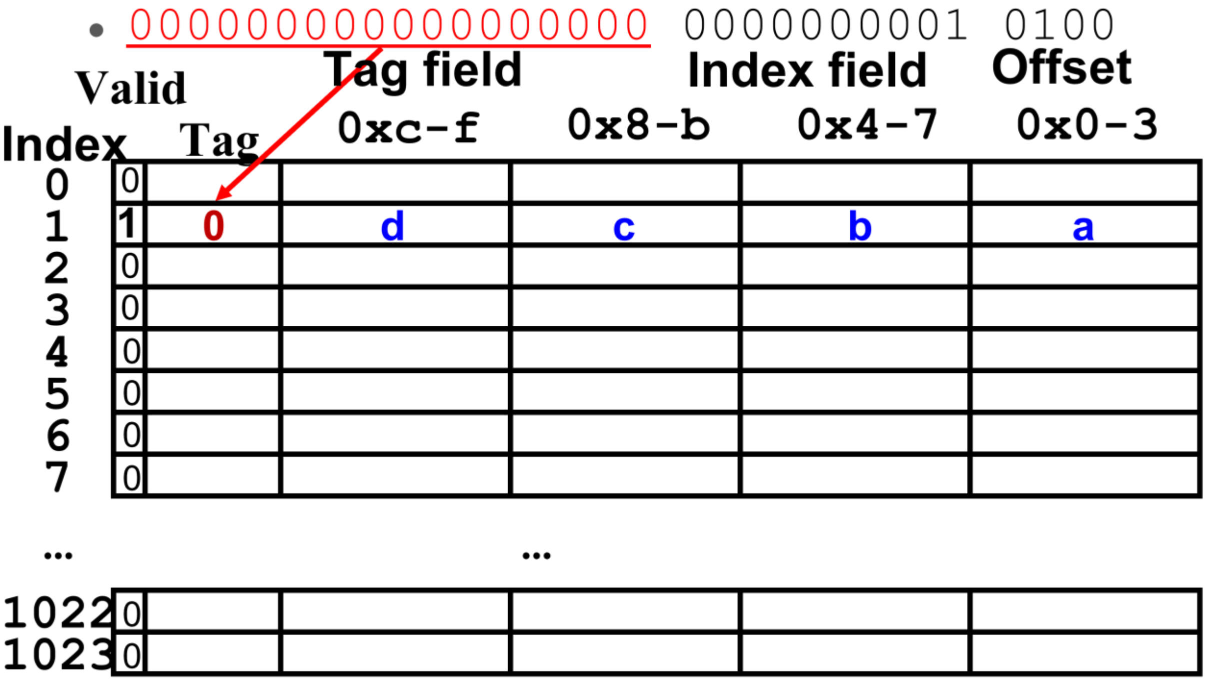
- No valid data
  - 00000000000000000000 0000000001 0100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Read Data in Direct-Mapped Cache (5/15)

- Load that data into cache, setting tag, valid

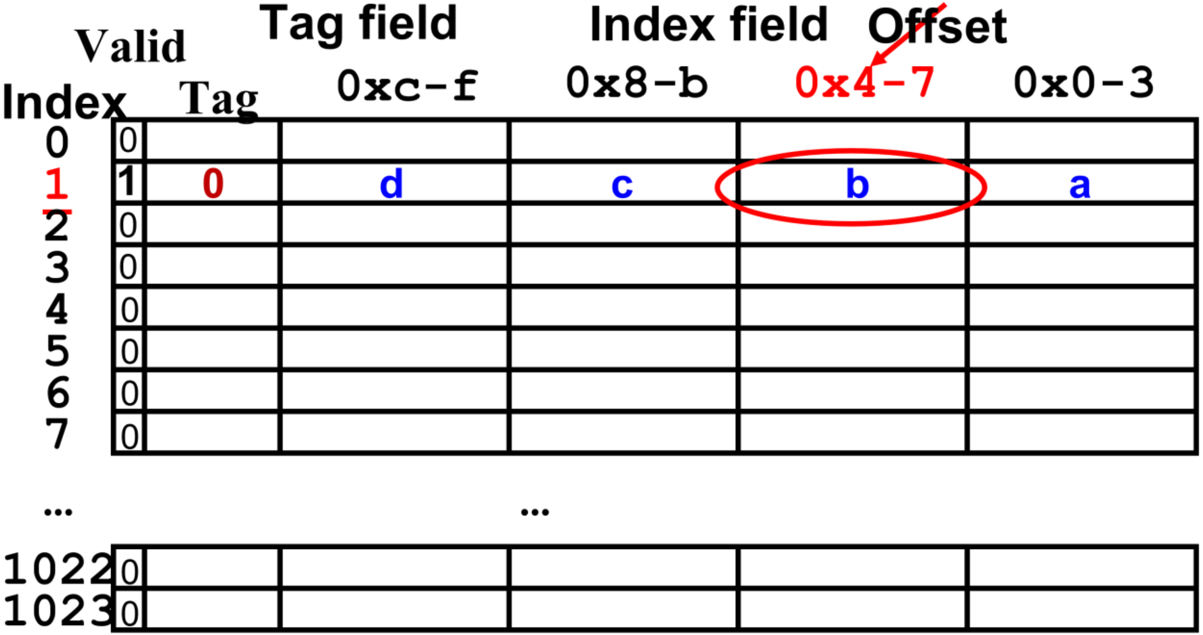


- 000000000000000000 0000000001 0100

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read Data in Direct-Mapped Cache (6/15)

- Read from cache at offset, return word b
  - 000000000000000000 0000000001 0100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read Data in Direct-Mapped Cache (7/15)

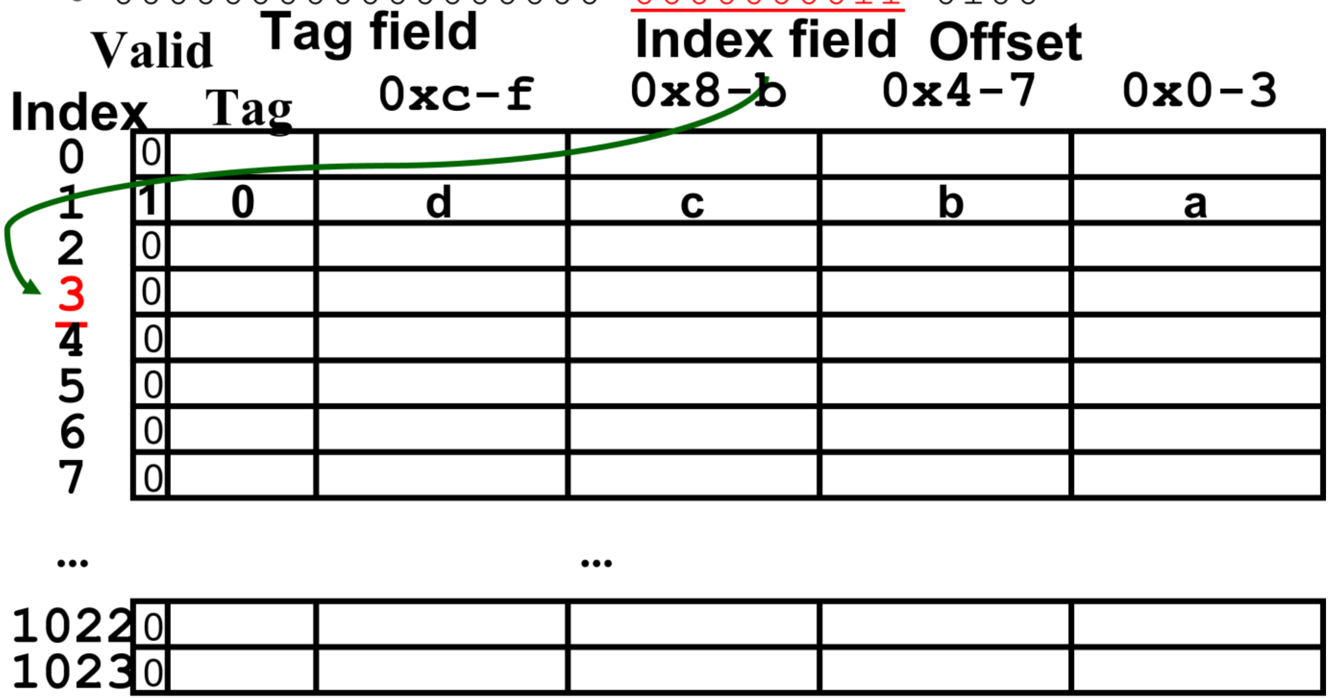
- Read 0x00000034
  - 000000000000000000 0000000011 0100

Tag field Index field Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Read Data in Direct-Mapped Cache (8/15)

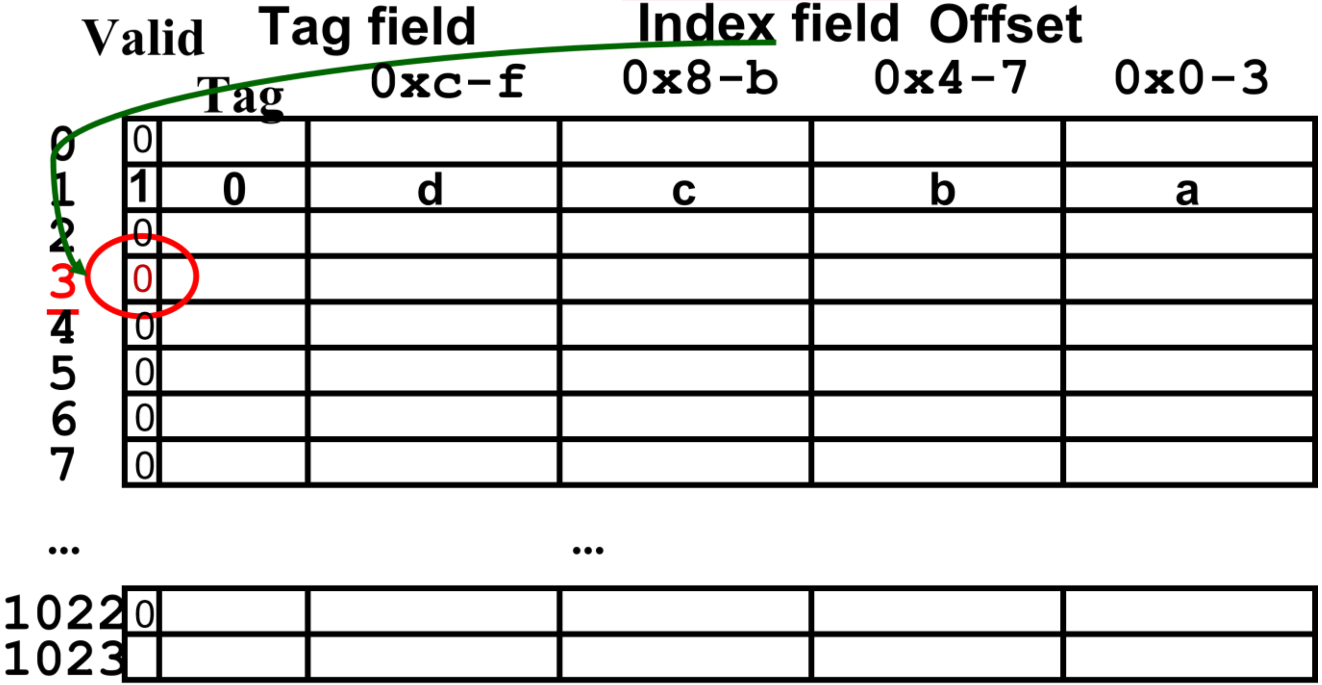
- Read block 3
  - 000000000000000000 0000000011 0100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read Data in Direct-Mapped Cache (9/15)

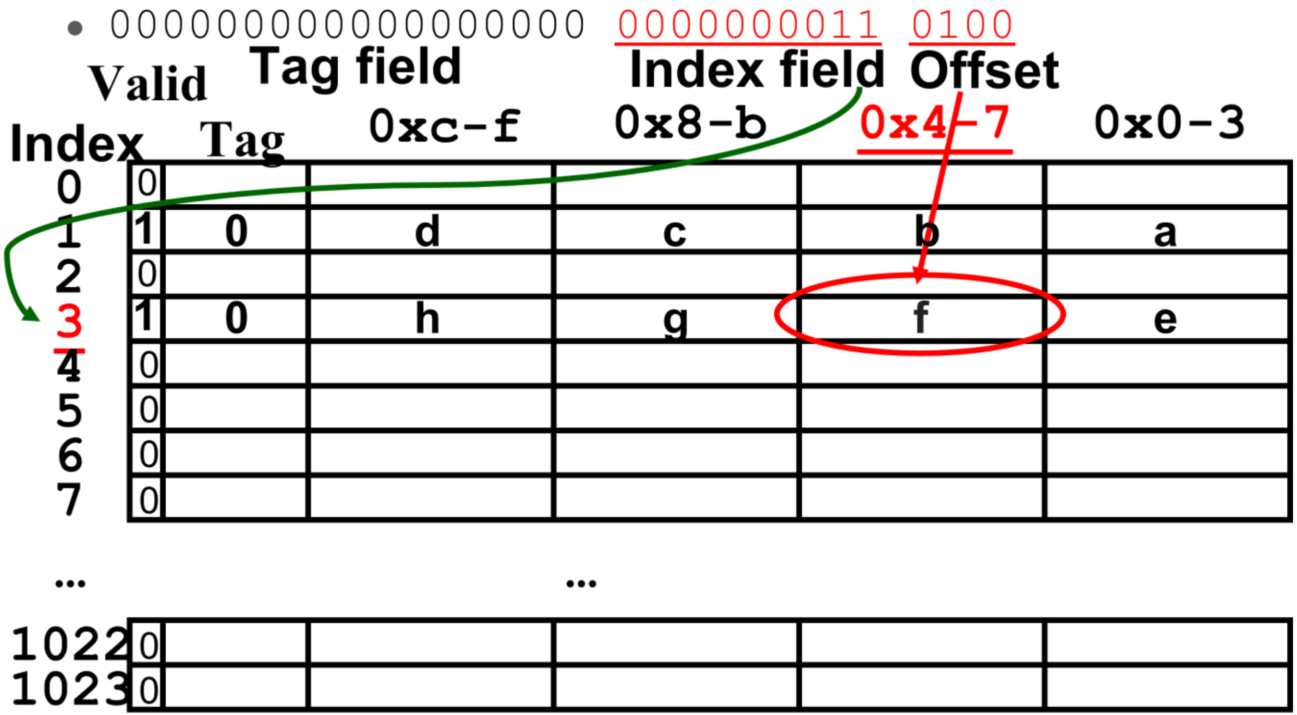
- No valid data
  - 000000000000000000 0000000011 0100



	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023						

Read Data in Direct-Mapped Cache (10/15)

- Load that cache block, return word f
  - 00000000000000000000 0000000011 0100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Read Data in Direct-Mapped Cache (11/15)

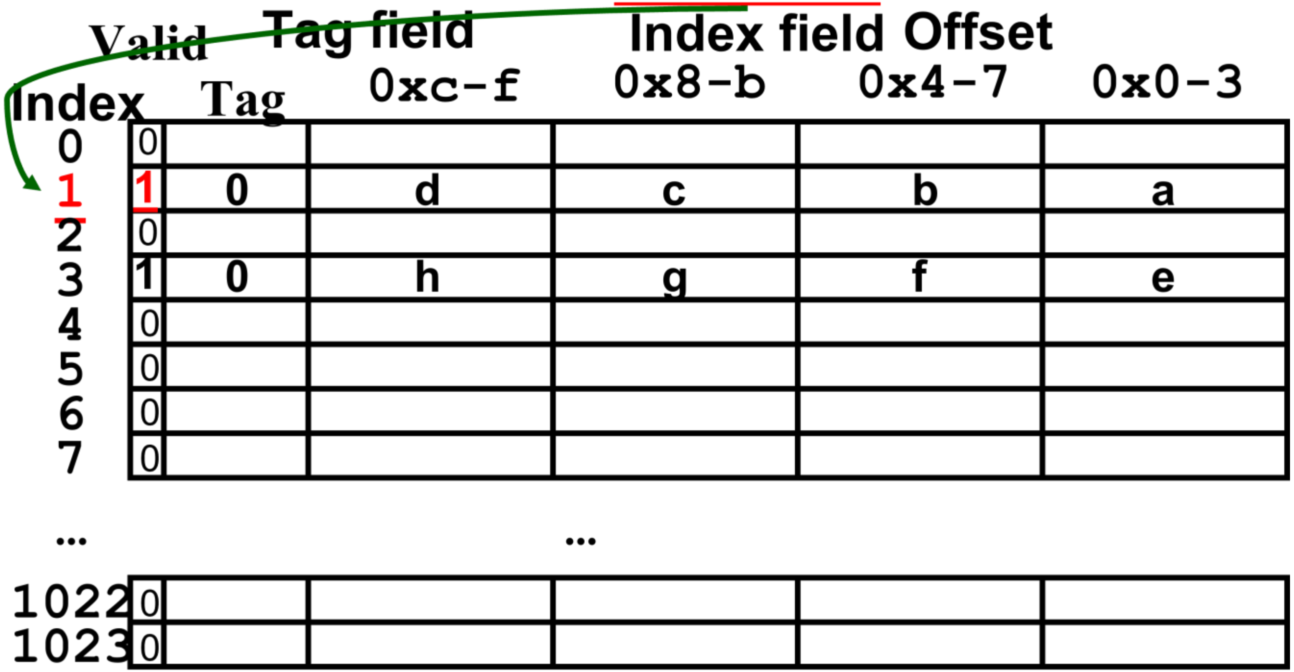
- Read 0x00008014
  - 000000000000000010 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read Data in Direct-Mapped Cache (12/15)

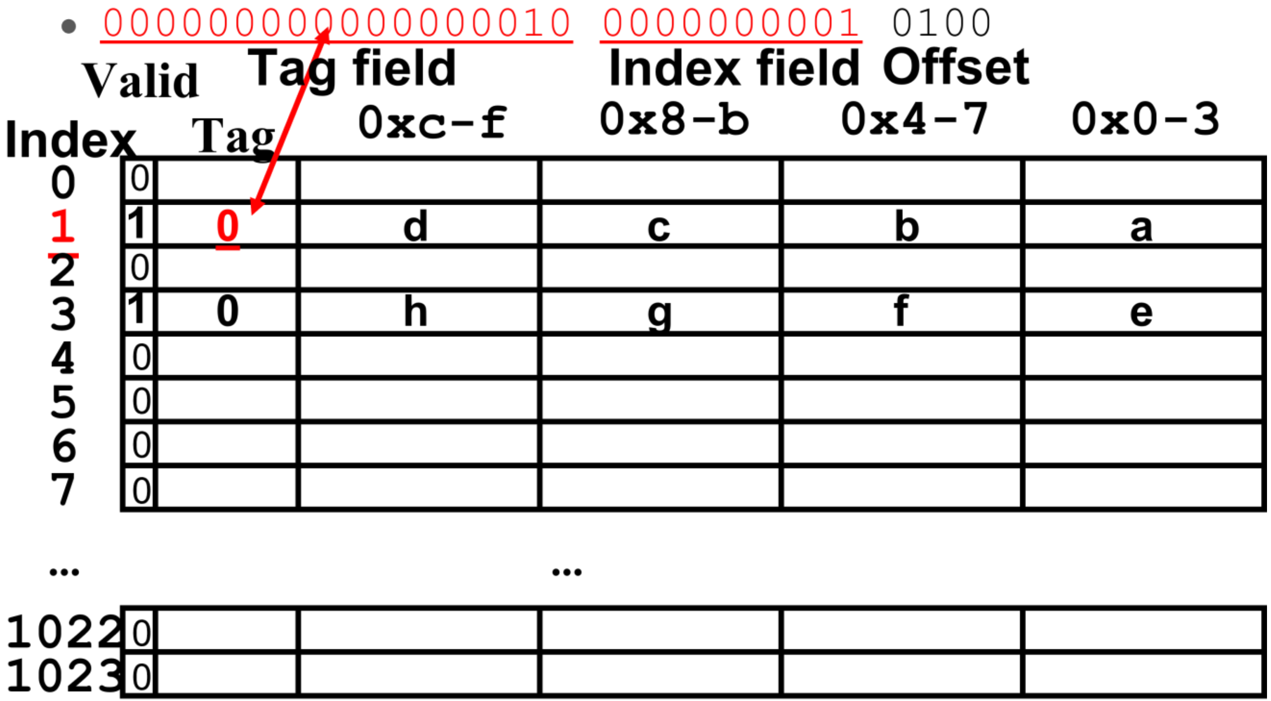
- Read cache block 1, data is valid
  - 000000000000000010 0000000001 0100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read Data in Direct-Mapped Cache (13/15)

- Cache block 1 tag does not match (0 != 2)
  - 000000000000000010 0000000001 0100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Read Data in Direct-Mapped Cache (14/15)

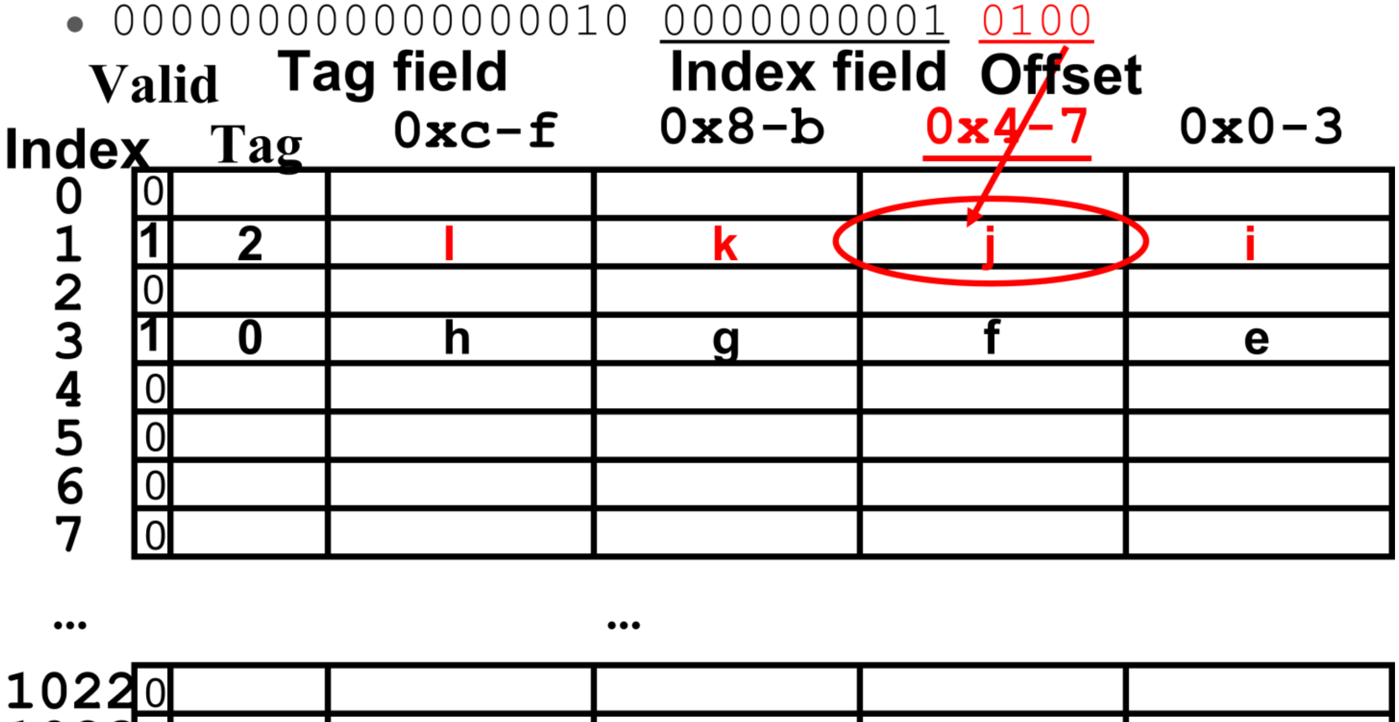
- Miss, so replace block 1 with new data & tag
  - 000000000000000010 0000000001 0100

Valid Tag field Index field Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Read Data in Direct-Mapped Cache (15/15)

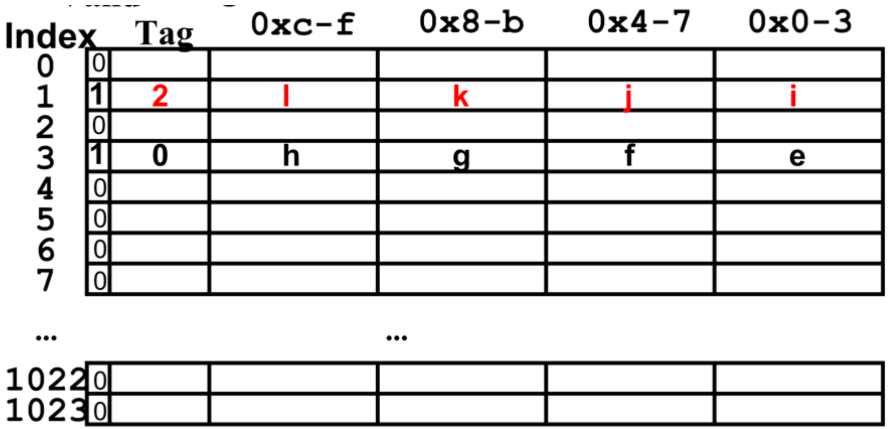
- Return word J
  - 000000000000000010 0000000001 0100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					

Takeaway Questions

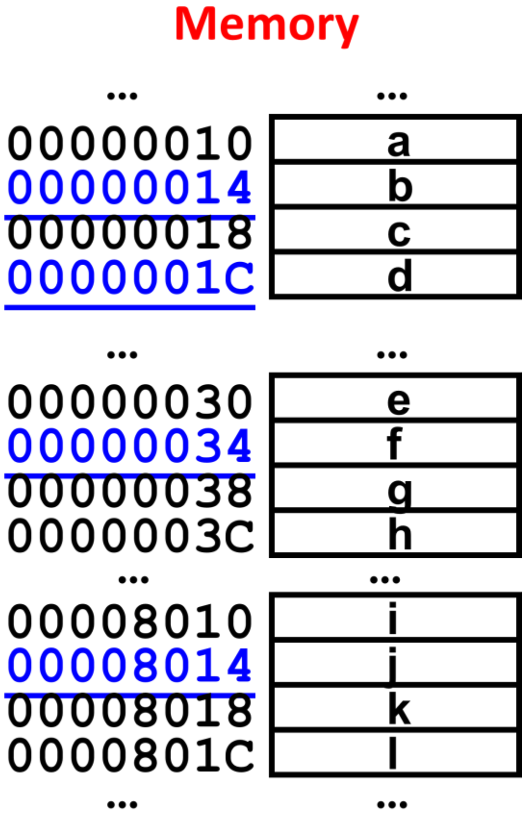
- What is the cache status when reading?
  - Read address 0x00000030?
    - 000000000000000000 0000000011 0000
  - Read address 0x0000001C?
    - 000000000000000000 0000000001 1100



Index		Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Takeaway Questions

- 0x00000030 a hit
  - Index = 3, Tag matches, offset = 0, value = e
- 0x000001C a miss
  - Index = 1, tag mismatch, so replace from memory, offset = 0xc, value = d
- Read values must = memory values whether or not cached
  - 0x00000030 = e
  - 0x0000001C = d



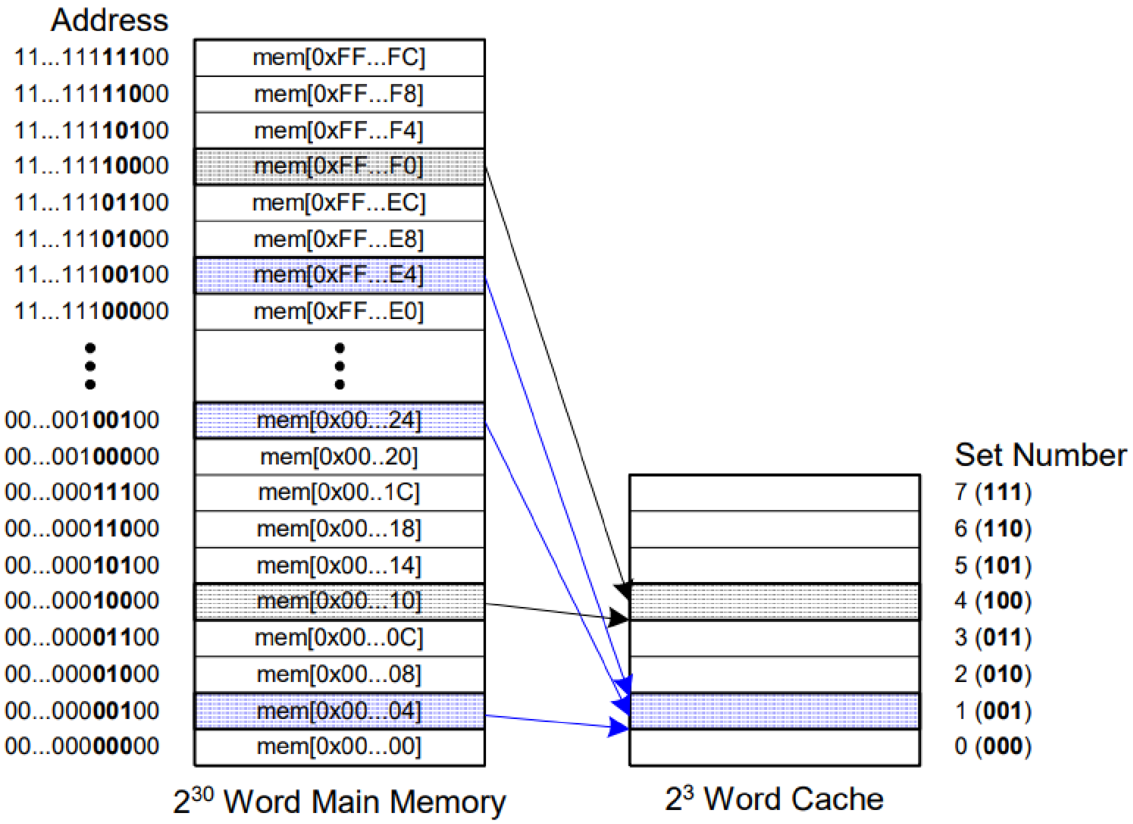
Memory

...	...
00000010	a
00000014	b
00000018	c
0000001C	d
...	...
00000030	e
00000034	f
00000038	g
0000003C	h
...	...
00008010	i
00008014	j
00008018	k
0000801C	l
...	...

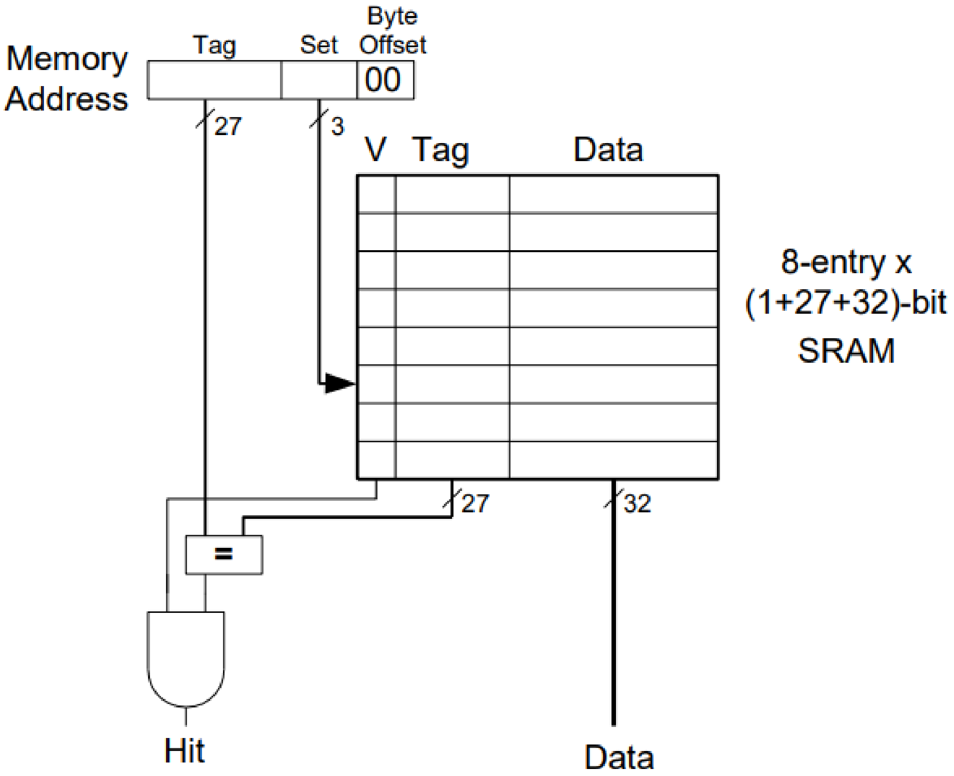
Outline

- Memory Hierarchy
- Memory Caching
- Cache Basics
- Direct-Mapped Cache
- Read Data in Direct-Mapped Cache
- Directed-Mapped Cache Hardware

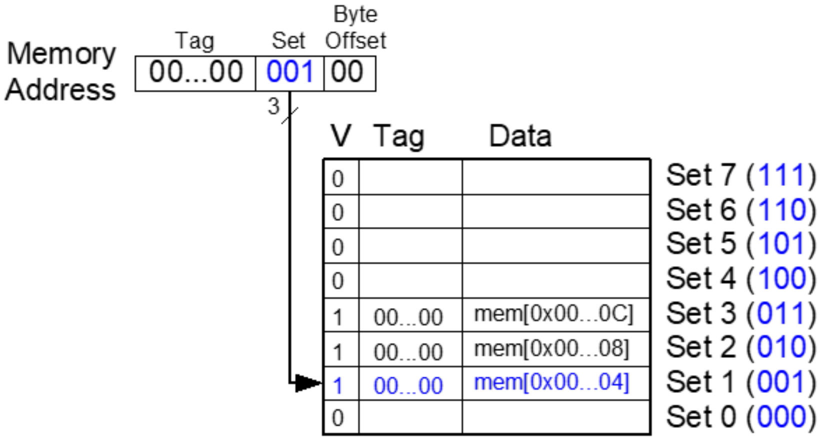
Directed-Mapped Cache Hardware (1/8)



Directed-Mapped Cache Hardware (2/8)



Directed-Mapped Cache Hardware (3/8)



V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
1	00...00	mem[0x00...0C]	Set 3 (011)
1	00...00	mem[0x00...08]	Set 2 (010)
1	00...00	mem[0x00...04]	Set 1 (001)
0			Set 0 (000)

```
# RISC-V assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

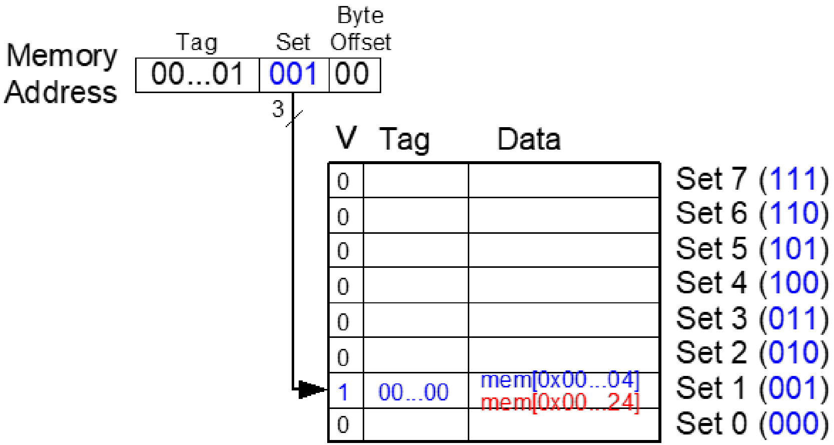
Miss Rate = 3/15 =

20%

Temporal Locality

Compulsory Misses

Directed-Mapped Cache Hardware (4/8)



Memory Address

Tag	Set	Byte Offset
00...01	001	00

V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
0			Set 3 (011)
0			Set 2 (010)
1	00...00	mem[0x00...04] mem[0x00...24]	Set 1 (001)
0			Set 0 (000)

```
# RISC-V assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

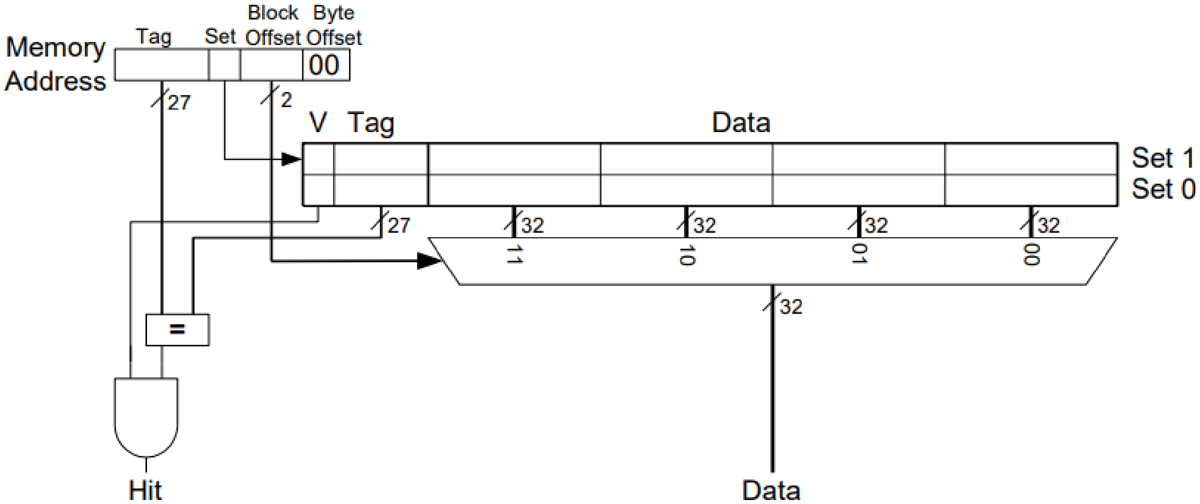
Miss Rate = 10/10

= 100%

Conflict Misses

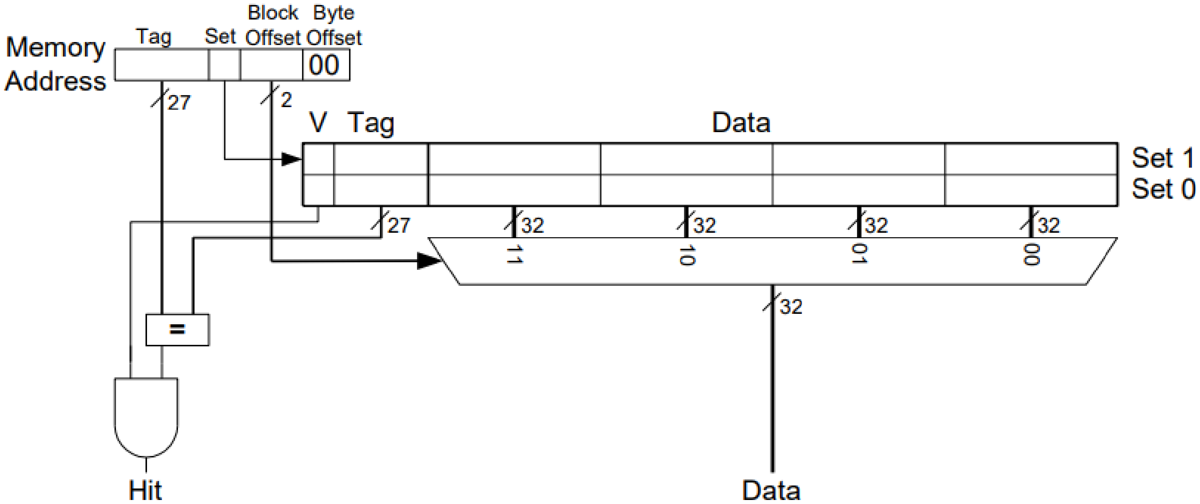
Directed-Mapped Cache Hardware (5/8)

- Increase block size
  - Block size , b = 4 words
  - C = 8 words, direct mapped (1 block per set)
  - Number of blocks, B = C/b = 8/4 = 2



Directed-Mapped Cache Hardware (6/8)

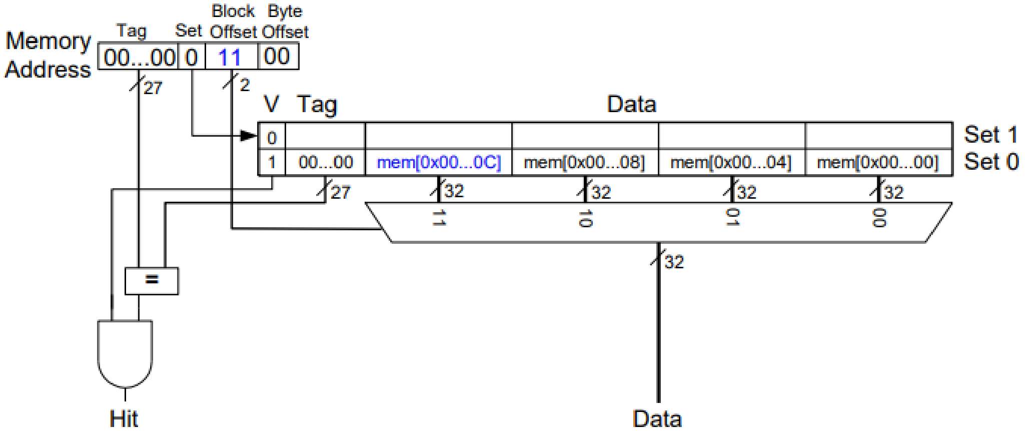
- Increase block size
  - Block size , b = 4 words
  - C = 8 words, direct mapped (1 block per set)
  - Number of blocks, B = C/b = 8/4 = 2



Directed-Mapped Cache Hardware (7/8)

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =



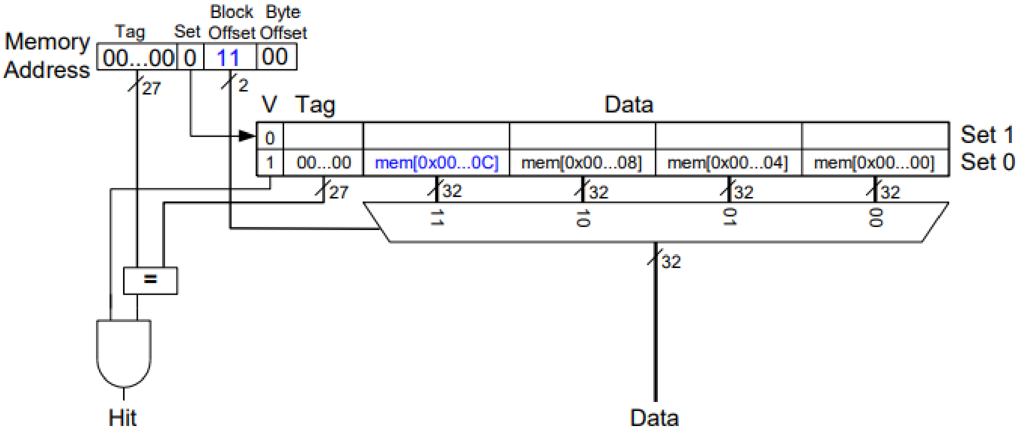
Directed-Mapped Cache Hardware (8/8)

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 1/15

= 6.67%

Larger blocks reduce compulsory misses through spatial locality



Conclusion

- We would like to have the capacity of disk at the speed of the processor: unfortunately this is not feasible
- So we create a memory hierarchy:
  - each successively lower level contains “most used” data from next higher level
  - exploits temporal & spatial locality
  - do the common case fast, worry less about the exceptions
- Locality of reference is a Big Idea