National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 10: Cache I

CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

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
  - CIS510 at Upenn
    - https://www.cis.upenn.edu/~cis5710/spring2019/

Outline

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

Types of Memory

- **Static RAM (SRAM)**
  - 6 or 8 transistors per bit
  - Two inverters (4 transistors) + transistors for reading/writing
  - Optimized for speed (first) and density (second)
  - Fast (sub-nanosecond latencies for small SRAM)
    - Speed roughly proportional to its area (~sqrt(number of bits))
  - Mixes well with standard processor logic

Types of Memory

- **Dynamic RAM (DRAM)**
  - 1 transistor + 1 capacitor per bit
  - Optimized for density (in terms of cost per bit)
  - Slow (> 30 ns internal access, ~50 ns pin-to-pin)
  - Different fabrication steps (does not mix well with logic)
- Nonvolatile storage: Magnetic disk, Flash RAM, Phase-change memory, …

Memory & Storage Technologies

- **Cost** – what can $200 buy (2009)?
  - SRAM: 16MB
  - DRAM: 4,000 MB(4GB) – 250x cheaper than SRAM
  - Flash: 64,000 MB (64GB) = 16x cheaper than DRAM
  - Disk: 2,000,000MB (2TB) – 32x vs Flash (512x vs. DRAM)
- **Latency**
  - SRAM: < 1 to 2ns (on chip)
  - DRAM: ~50ns – 100x or more slower than SRAM
  - Flash: 75,000ns (75 microseconds) – 1500x vs. DRAM
  - Disk: 10,000,000ns (10ms) – 133x vs Flash (200,000x vs DRAM)

Memory & Storage Technologies

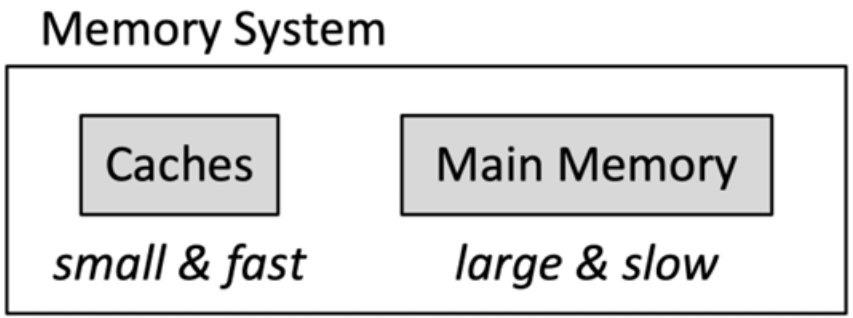
- **Bandwidth**
  - SRAM: 300GB/sec (e.g., 12-port 8-byte register file @ 3GHz)
  - DRAM: ~25GB/s
  - Flash: 0.25GB/s (250MB/s), 100x less than DRAM
  - Disk: 0.1GB/s (100MB/s), 250x vs DRAM, sequential access only

Memory Hierarchy

- **Problems in memories**
  - Bigger is slower
    - Bigger -> takes longer to determine the location
  - Faster is more expensive
    - SRAM vs. DRAM vs. SSD vs. Disk vs. Tape
  - Higher bandwidth is more expensive
    - Need more banks, more ports, more channels, higher frequency or faster technology

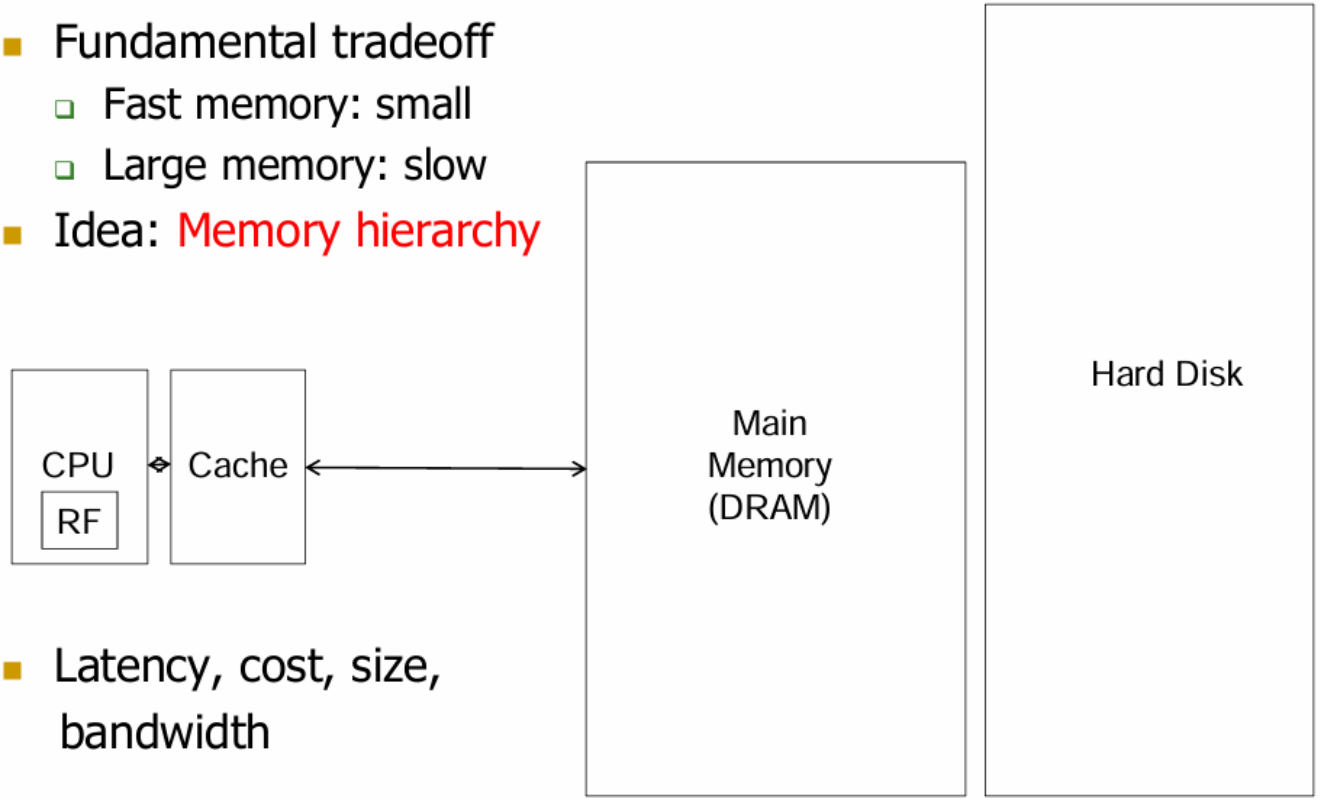
Memory Hierarchy

- **Why memory hierarchy?**
  - We want both fast and large
  - But, we cannot achieve both with a single level of memory
  - Idea: Have multiple levels of storage
    - Bigger and slower as the levels are farther from the processor
    - Ensure most of the data the processor needs is kept in the fast level



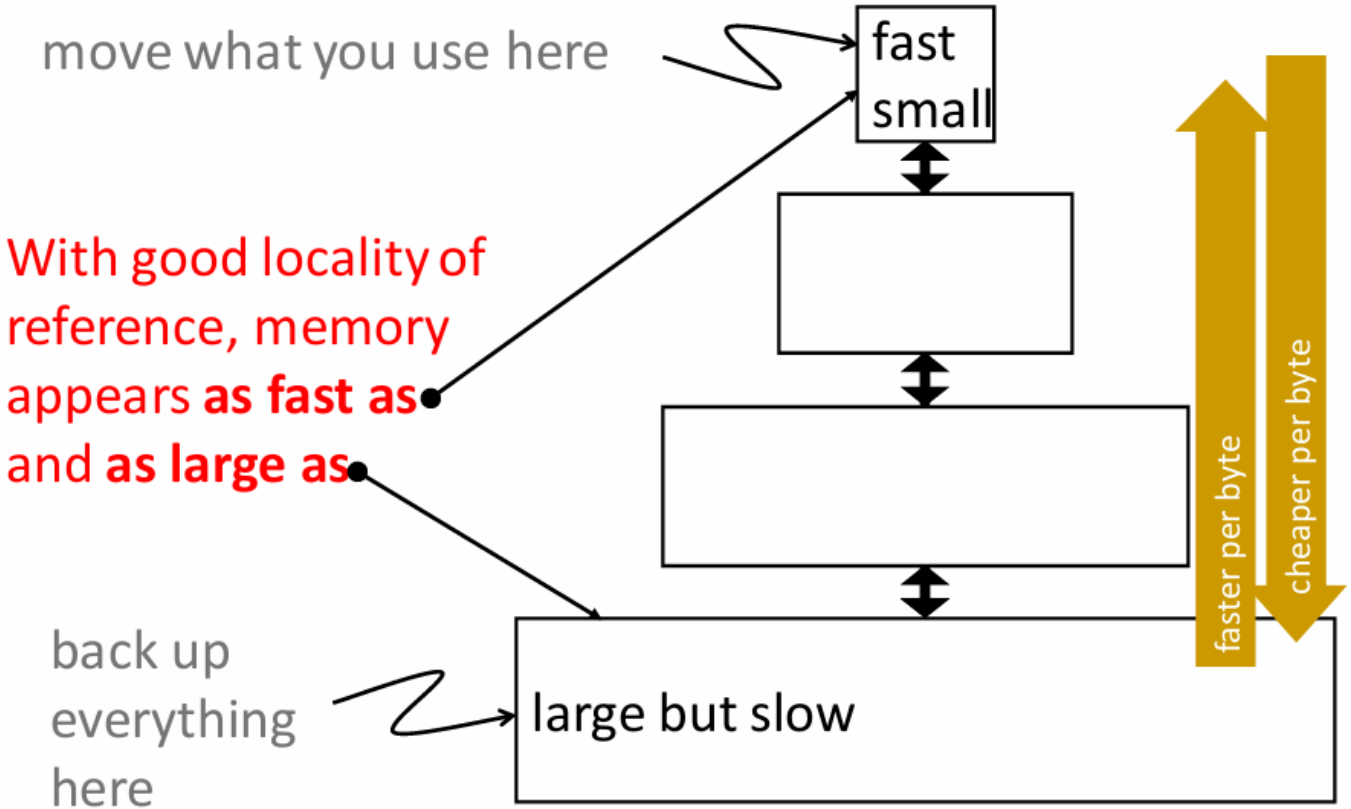
Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Idea: Memory hierarchy

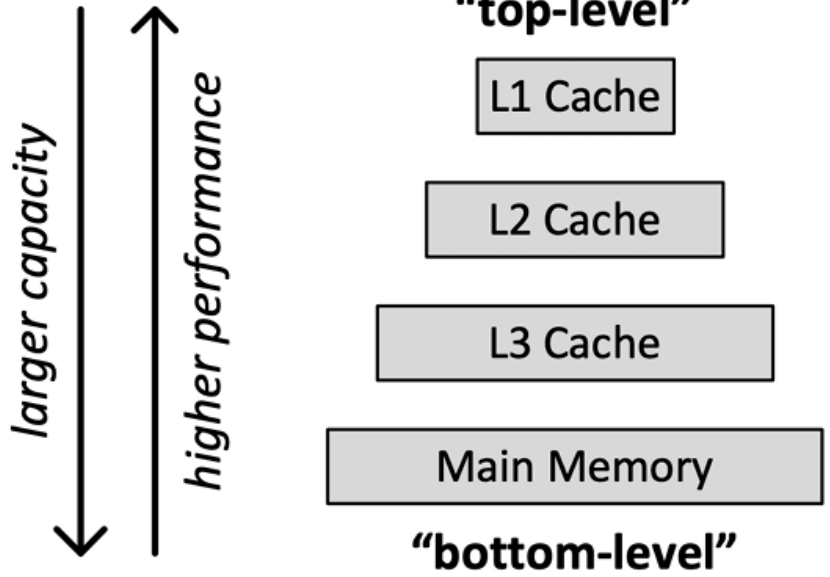


- Latency, cost, size, bandwidth

Memory Hierarchy



Memory Hierarchy

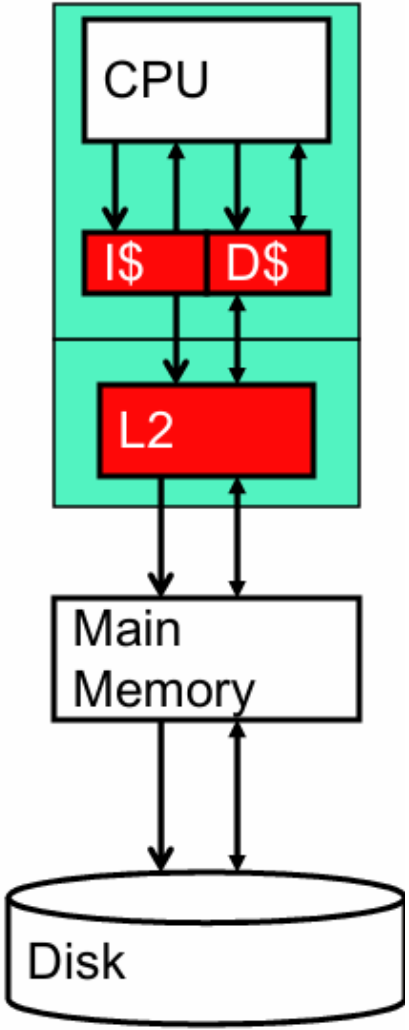


capacity	latency
10's of KB	≈ 1ns
100's of KB	< 5ns
several MB	≈ 10ns
several GB	≈ 100ns

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

The Unit: Cache

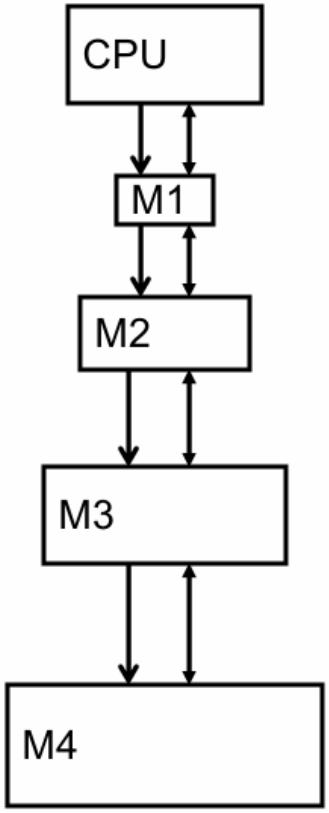
- **Cache: hardware managed**
  - Hardware automatically retrieves missing data
  - Built from fast SRAM, usually on-chip today
  - In contrast to off-chip, DRAM “main memory”
- **Cache organization**
  - Speed vs. Capacity
  - Miss classification



Memory Locality

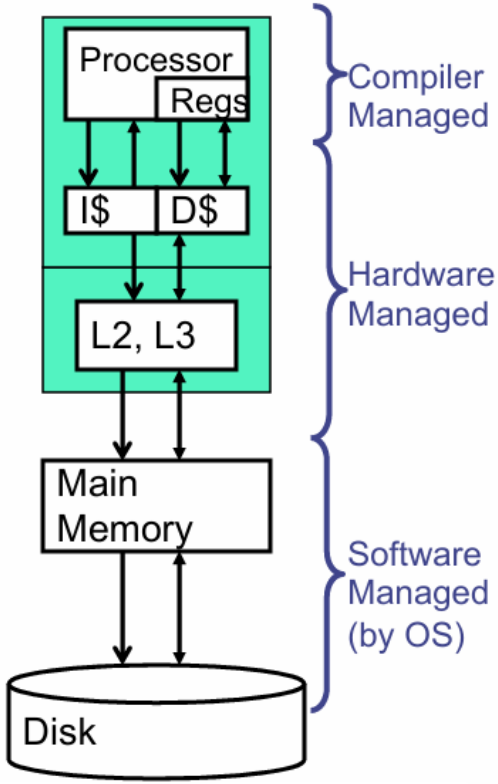
- **Cache** contains copies of data in memory being used
- **Memory** contains copies of data on disk being used
- Caches work on principles of temporal and spatial locality
  - **Temporal locality**: if we use it now, chances are we'll want to use it again soon
    - Data elements accessed **in loops** (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)

Exploiting Locality: Memory Hierarchy



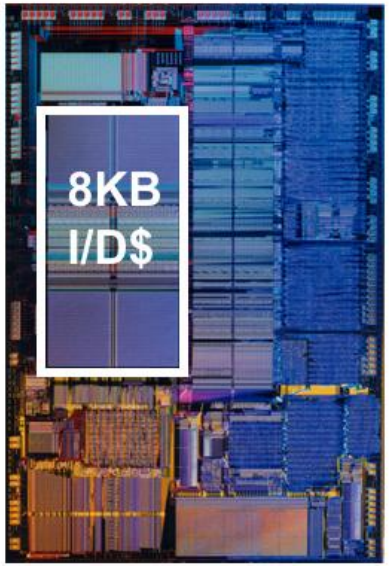
- Hierarchy of memory components
  - Upper components
    - Fast ↔ Small ↔ Expensive
  - Lower components
    - Slow ↔ Big ↔ Cheap
- Connected by "buses"
  - Which also have latency and bandwidth issues
- Most frequently accessed data in M1
  - M1 + next most frequently accessed in M2, etc.
  - Move data up-down hierarchy
- Optimize average access time
  - $latency_{avg} = latency_{hit} + (\%_{miss} * latency_{miss})$
  - Attack each component

Concrete Memory Hierarchy

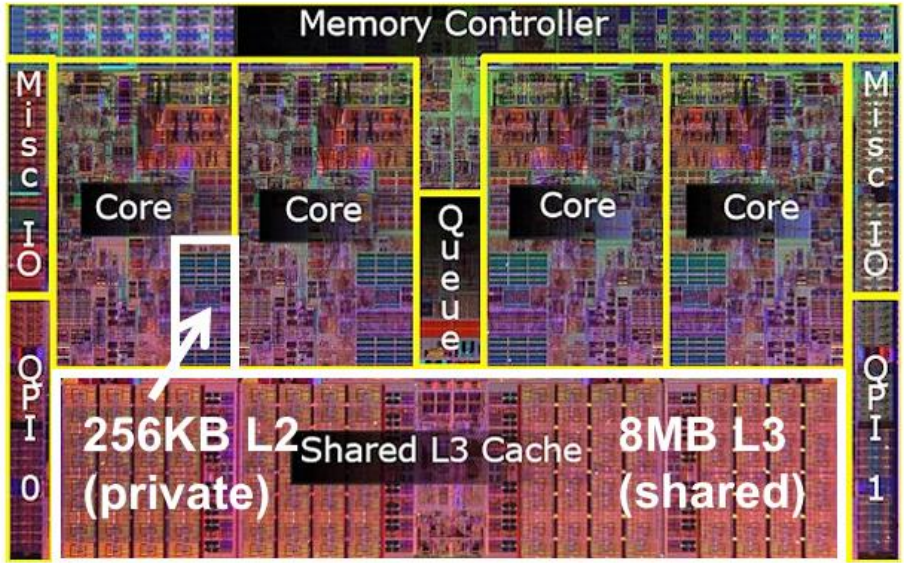


- 0th level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$)
  - Typically 8KB to 64KB each
- 2nd level: **2nd and 3rd cache** (L2, L3)
  - On-chip, typically made of SRAM
  - 2nd level typically ~256KB to 512KB
  - "Last level cache" typically 4MB to 16MB
- 3rd level: **main memory**
  - Made of DRAM ("Dynamic" RAM)
  - Typically 1GB to 4GB for desktops/laptops
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**
  - Uses magnetic disks or flash drives

Evolution of Cache Hierarchies



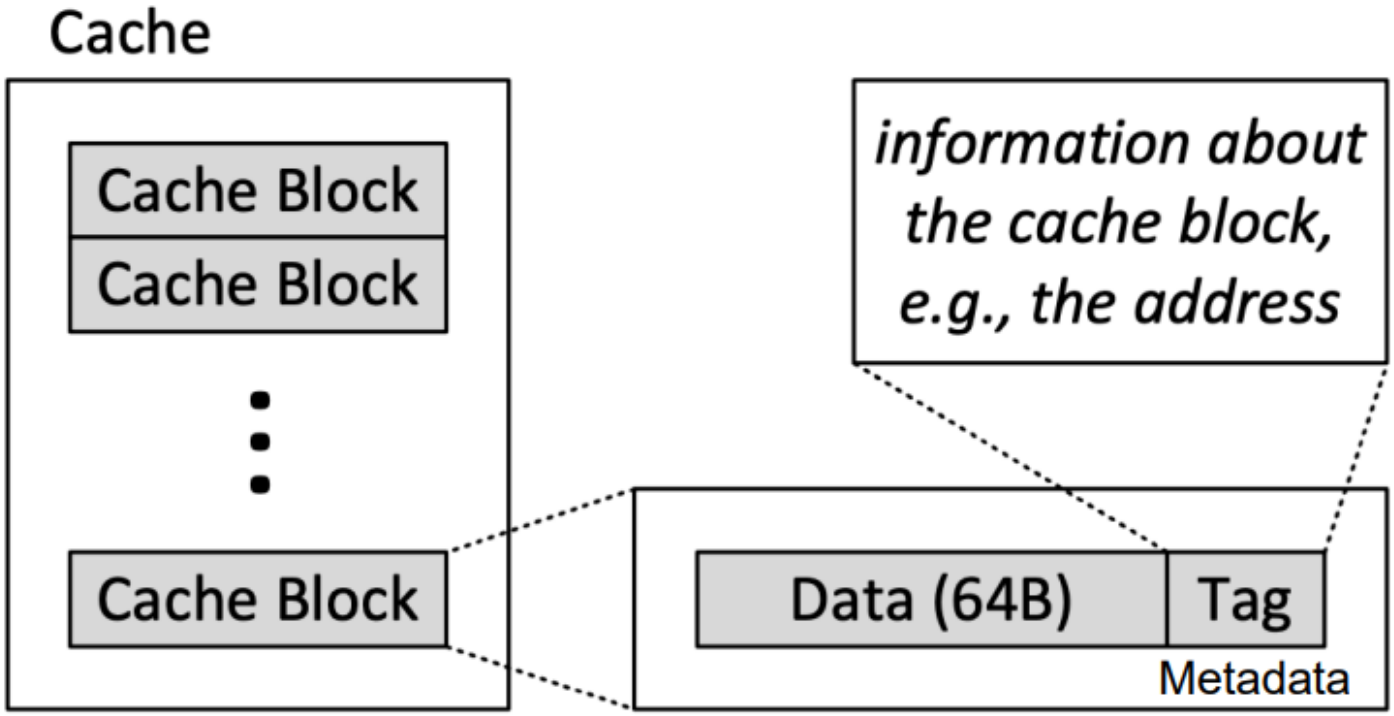
Intel 486



Intel Core i7 (quad core)

- Chips today are 30–70% cache by area

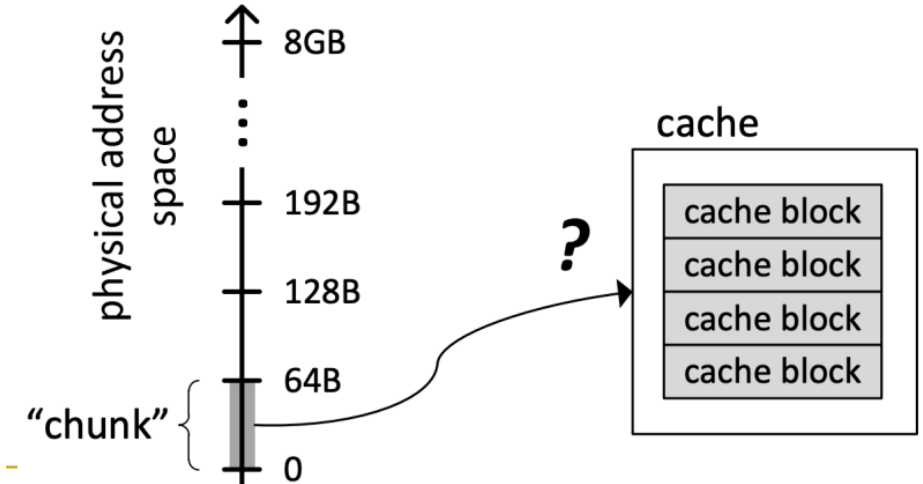
Cache Basics



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

Cache Basics

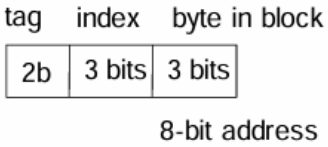
- **A key question**
  - How to map chunks of the main memory address space to blocks in the cache?
  - Which location in cache can a given “main memory chunk” be placed in?



Cache Basics

- Main memory logically divided into fixed-size chunks (**blocks**)
- **Cache** can house only a **limited** number of blocks

- Each **block address** maps to a potential location in the cache, determined by the index bits in the address
  - used to index into the tag and data stores



- Cache access:

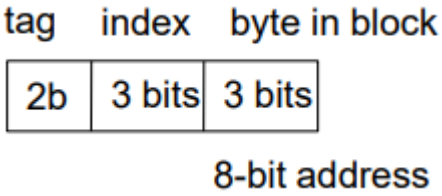
  1) index into the tag and data stores with index bits in address
  2) check valid bit in tag store
  3) compare tag bits in address with the stored tag in tag store

- If the stored tag is valid and matches the tag of the block, then the block is in the cache (cache hit)

Cache Basics

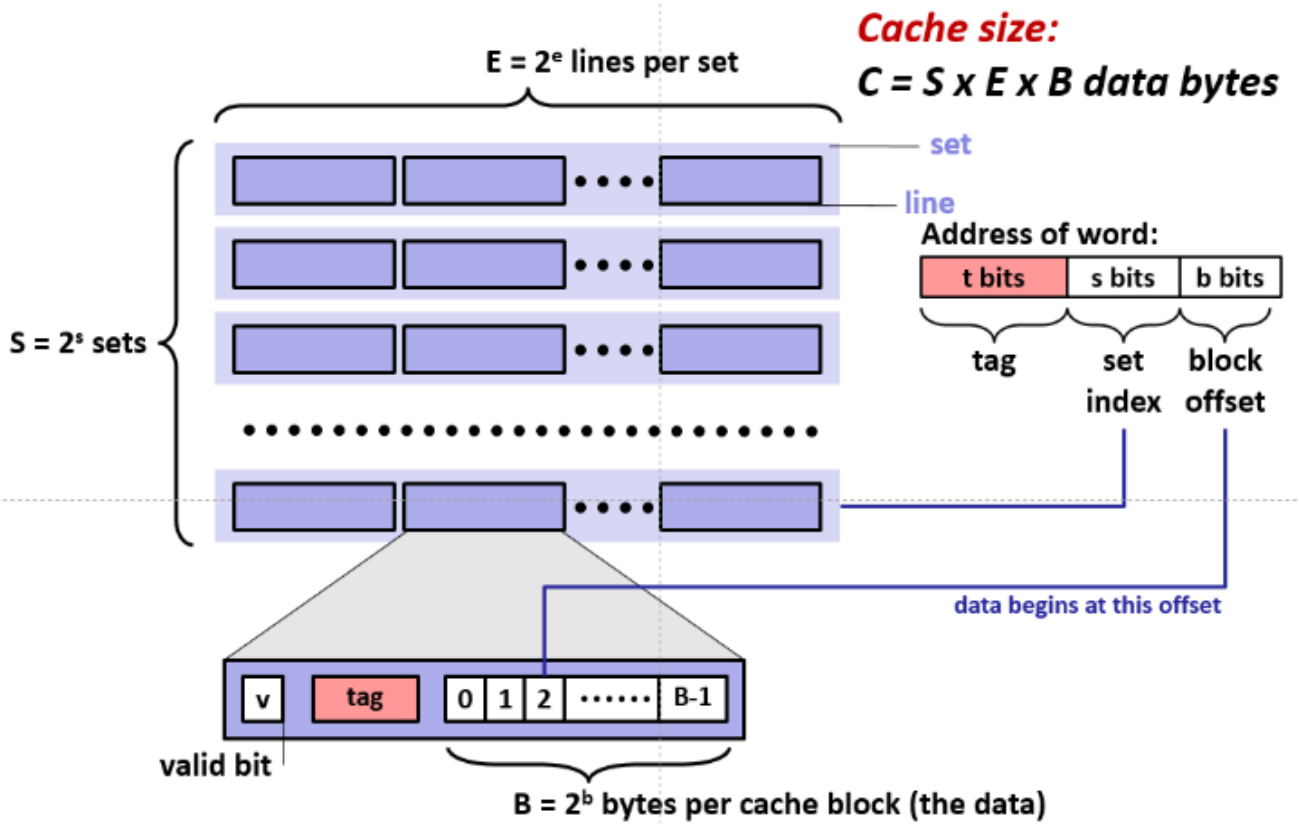
- **Block (line):** Unit of storage in the cache
  - Memory is logically divided into blocks that map to potential locations in the cache
- When reading memory, 3 things can happen
  - **Cache HIT**:
    - Cache block is valid and contains proper address, so read desired word
  - **Cache MISS**:
    - 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



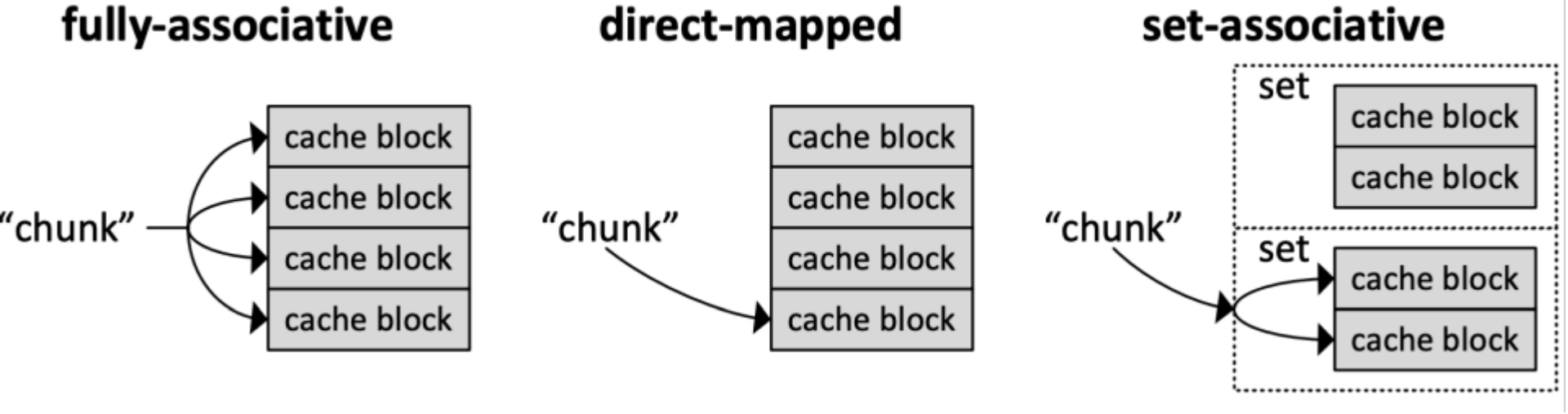
- Each block address maps to a potential location in the cache, determined by the index bits in the address
- **Index**
  - Specifies the cache index (which “row”/block of the cache we should look in)
- **Offset**
  - Once we’ve found correct block, specifies which byte within the block we want
- **Tag**
  - 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



Cache Basics

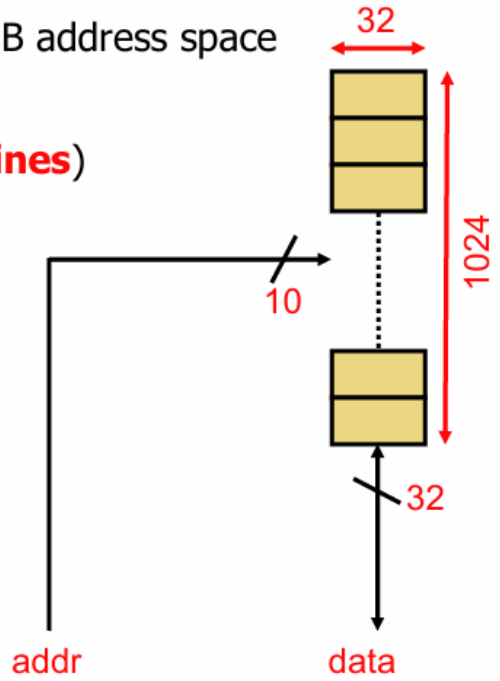
- **Cache associativity**
  - One **set** can contain multiple cache blocks



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

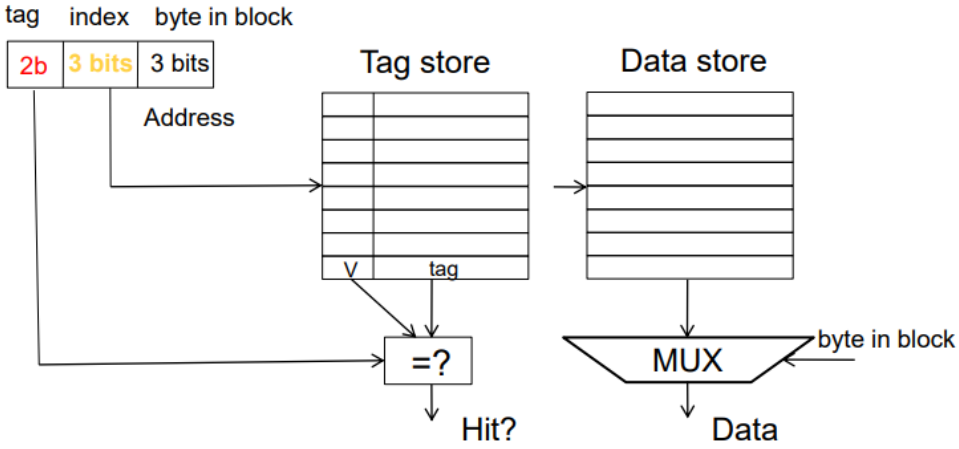
Logical Cache Organization

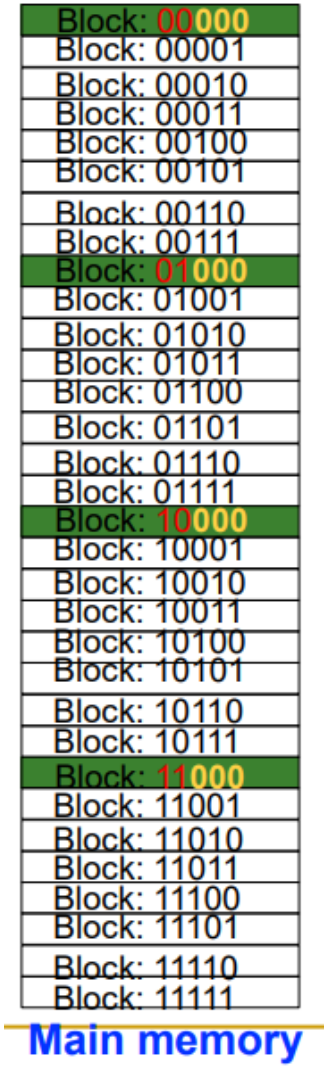
- **Cache is a hardware hashtable**
- The setup
  - 32-bit ISA → 4B words/addresses, 2^{32} B address space
- Logical cache organization
  - 4KB, organized as 1K 4B **blocks** (aka **lines**)
  - Each block can hold a 4-byte word
- Physical cache implementation
  - 1K (1024 bit) by 4B **SRAM**
  - Called **data array**
  - 10-bit address input
  - 32-bit data input/output



Looking Up A Block

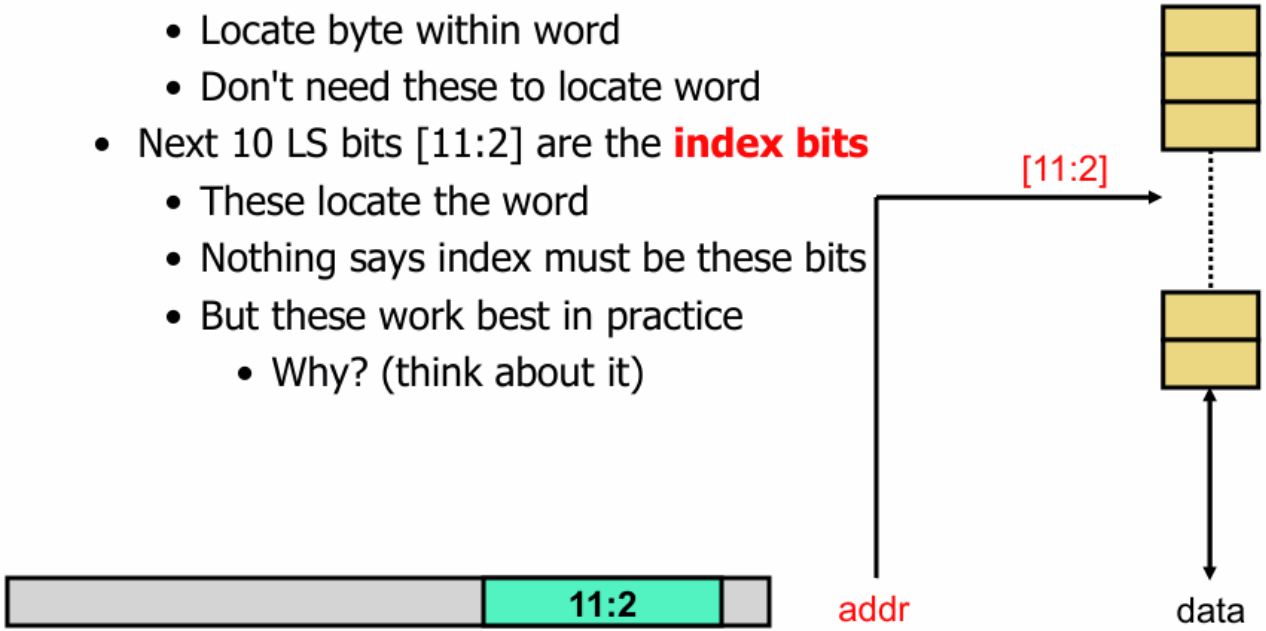
- A byte-addressable main memory
  - 256 bytes, 8-byte blocks -> 32 blocks in memory
  - Assume cache: 64 bytes, 8 blocks





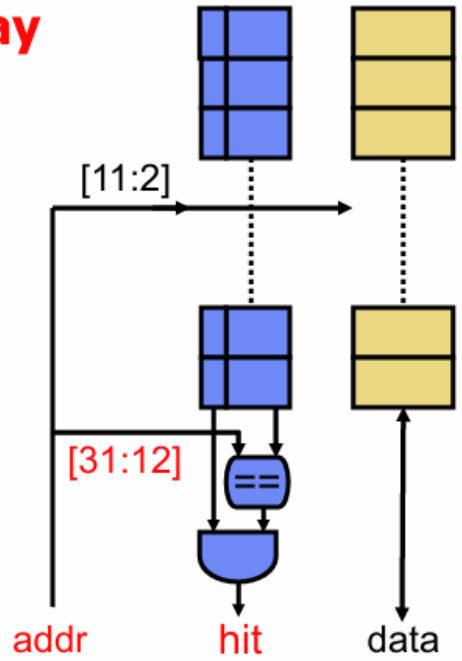
Looking Up A Block

- Q: which 10 of the 32 address bits to use?
- A: bits [11:2]
  - 2 least significant (LS) bits [1:0] are the **offset bits**
    - Locate byte within word
    - Don't need these to locate word
  - Next 10 LS bits [11:2] are the **index bits**
    - These locate the word
    - Nothing says index must be these bits
    - But these work best in practice
      - Why? (think about it)



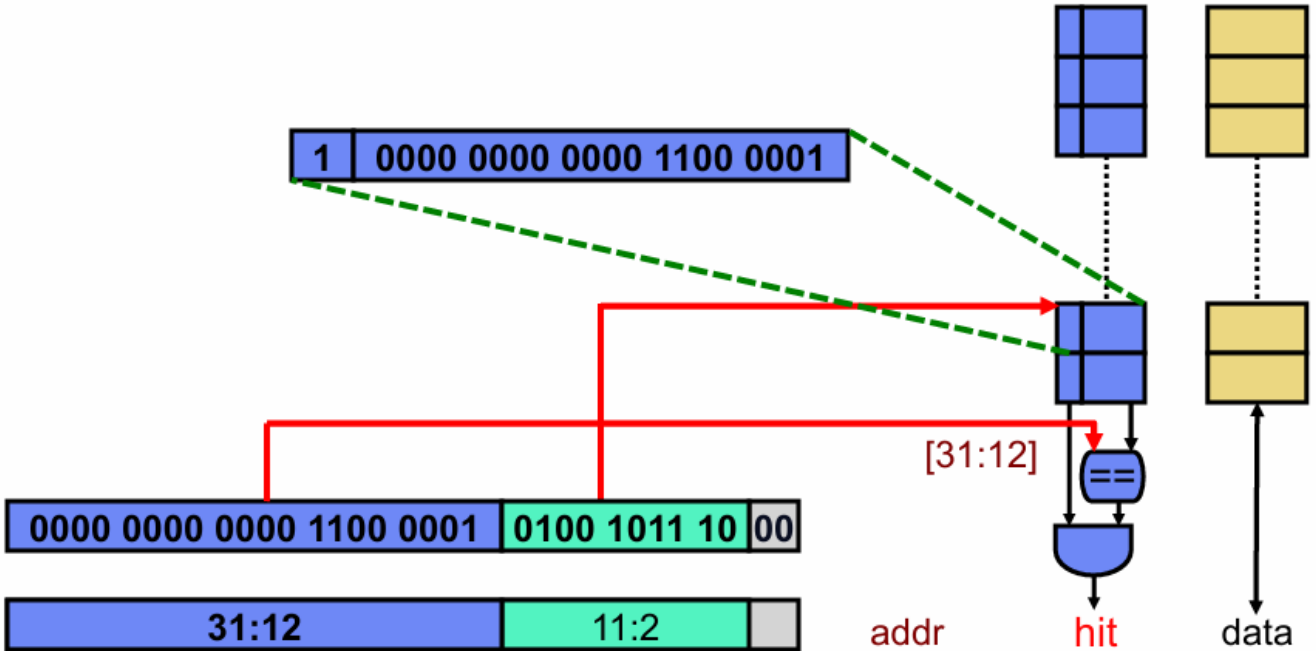
Is this the block you're looking for?

- Each cache row corresponds to 2^{20} blocks
  - How to know which if any is currently there?
  - Tag each cache word with remaining address bits [31:12]
- Build separate and parallel **tag array**
  - 1K by 21-bit SRAM
  - 20-bit (next slide) **tag** + 1 **valid bit**
- Lookup algorithm
  - Read tag indicated by index bits
  - If tag matches & valid bit set:
    then: Hit → data is good
    else: Miss → data is garbage, wait…



Is this the block you're looking for?

- Lookup address `x000C14B8`
  - Index = addr [11:2] = (addr >> 2) & x3FF = `x12E`
  - Tag = addr [31:12] = (addr >> 12) = `x000C1`



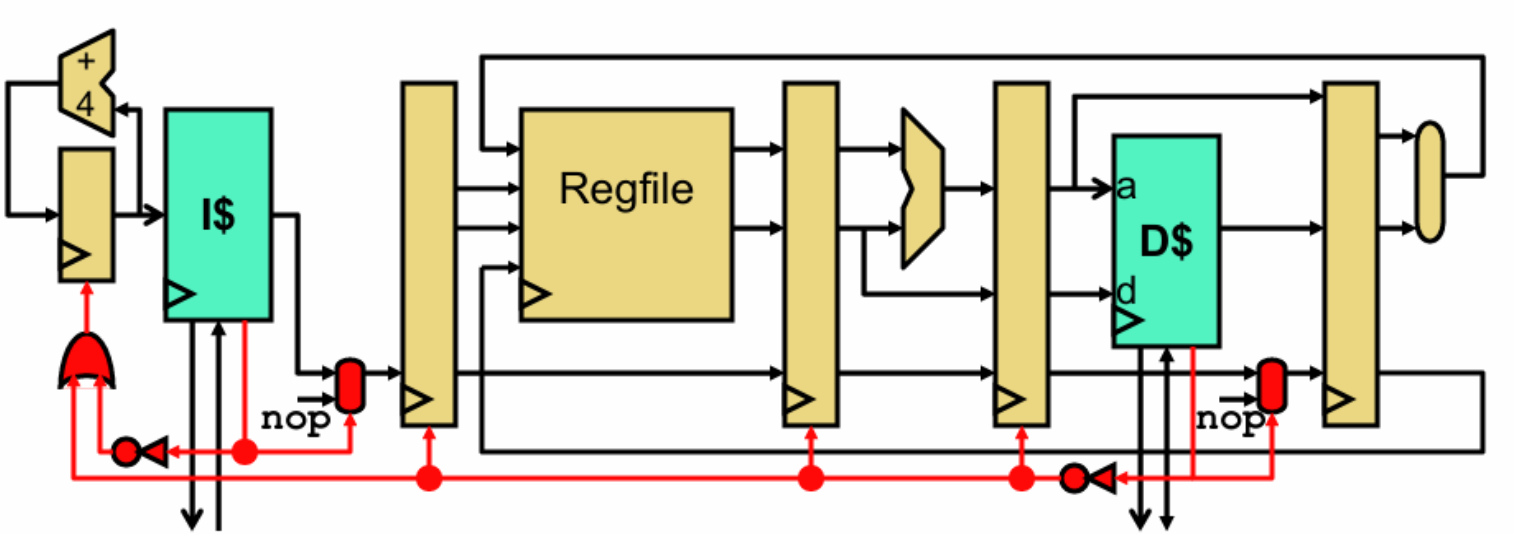
Handling a Cache Miss

- What if requested data isn't in the cache?
  - How does it get in there?

- **Cache controller**: finite state machine
  - Remembers miss address
  - Accesses next level of memory
  - Waits for response
  - Writes data/tag into proper locations

- Bringing a missing block into the cache is a **cache fill**

Cache Misses and Pipeline Stalls

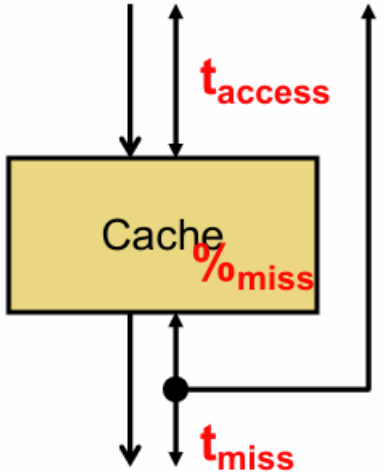


- I$ and D$ misses stall pipeline just like data hazards
  - Stall logic driven by miss signal
    - Cache “logically” re-evaluates hit/miss every cycle
    - Block is filled → miss signal de-asserts → pipeline restarts

Cache Misses

- **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 Performance Equation



- For a cache
  - **Access**: read or write to cache
  - **Hit**: desired data found in cache
  - **Miss**: desired data not found in cache
    - Must get from another component
    - No notion of "miss" in register file
  - **Fill**: action of placing data into cache

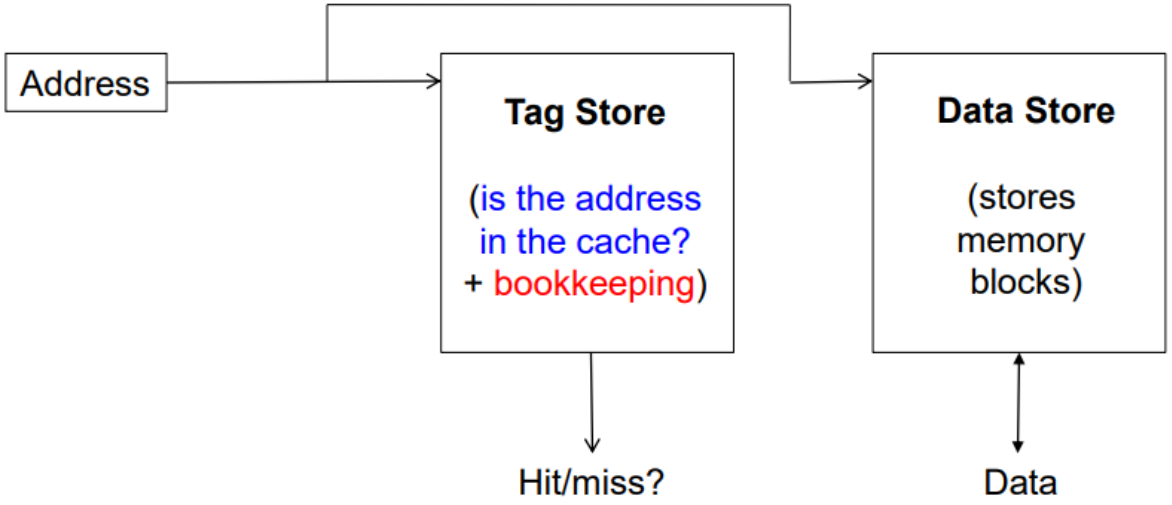
- $\%_{miss}$ (miss-rate): #misses / #accesses
- t_{access}: time to check cache. If hit, we're done.
- t_{miss}: time to read data into cache

- Performance metric: average access time

$$t_{avg} = t_{access} + (\%_{miss} * t_{miss})$$

Cache Performance Equation

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



CPI Calculation with Cache Misses

- Parameters
  - Simple pipeline with base CPI of 1
  - Instruction mix: 30% loads/stores
  - I$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
  - D$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles

- What is new CPI?
  - $CPI_{I\$} = \%_{missI\$} * t_{miss}$ = 0.02*10 cycles = 0.2 cycle
  - $CPI_{D\$} = \%_{load/store} * \%_{missD\$} * t_{missD\$}$ = 0.3 * 0.1*10 cycles = 0.3 cycle
  - $CPI_{new} = CPI + CPI_{I\$} + CPI_{D\$}$ = 1+0.2+0.3 = 1.5

Multi-Word Cache Blocks

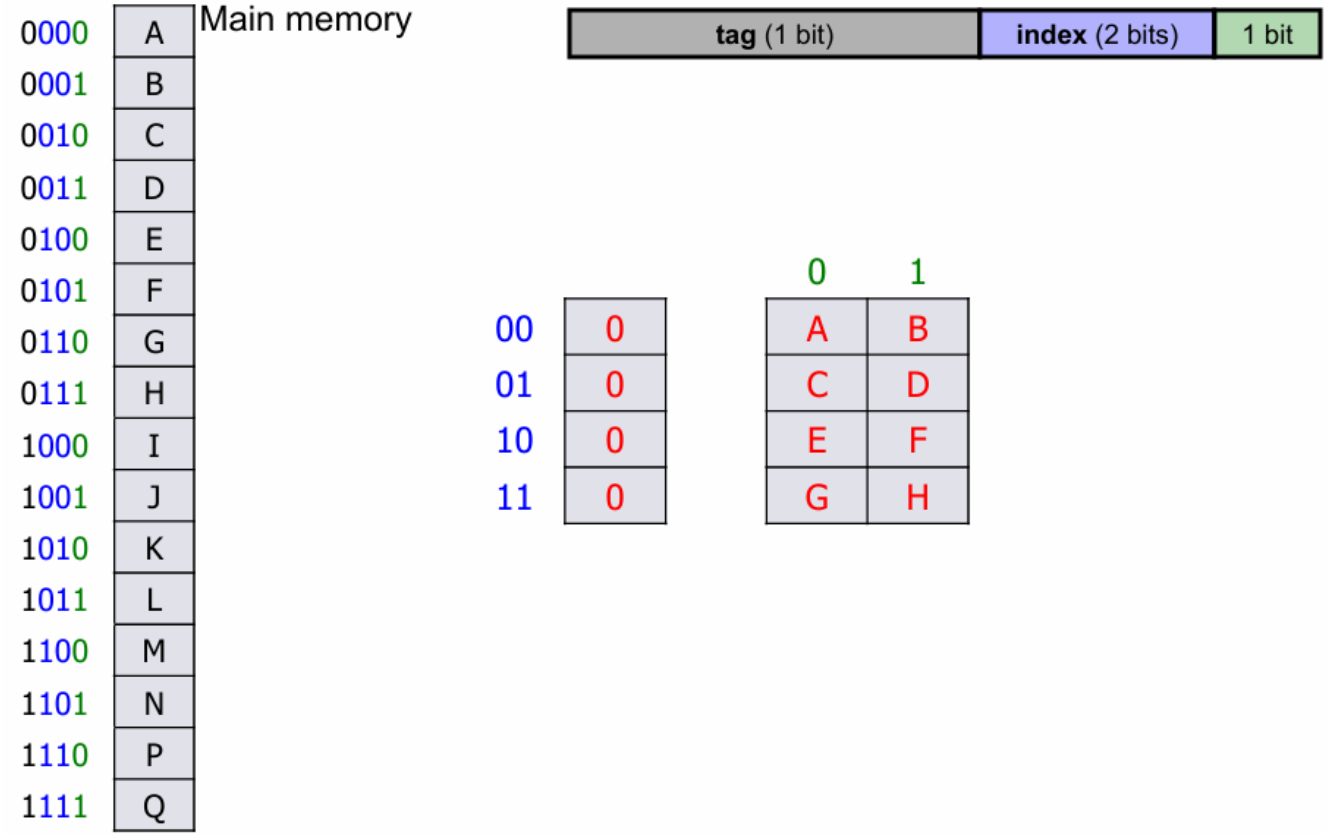
- In most modern implementation we store more than one address (>1 byte) in each cache block.
- The number of bytes or words stored in each cache block is referred to as the **block size**.
- The entries in each block come from a contiguous set of addresses to exploit locality of reference, and to simplify indexing

Cache Examples

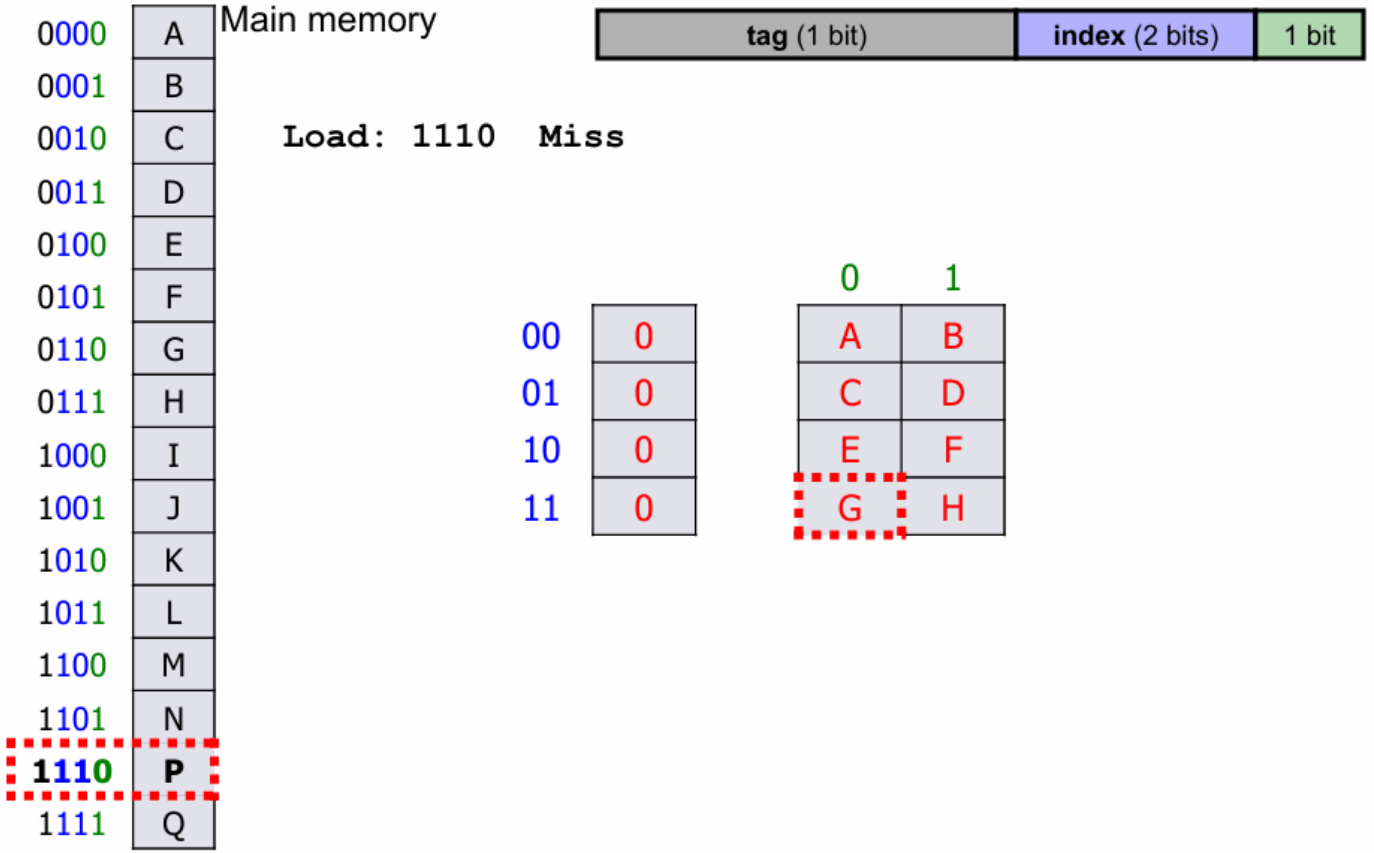
- 4-bit addresses → 16B memory
  - Simpler cache diagrams than 32-bits
- 8B cache, 2B blocks

  - Figure out number of sets: 4 (capacity / block-size)
  - Figure out how address splits into offset/index/tag bits
    - Offset: least-significant $\log_2$(block-size) = $\log_2(2) = 1 \rightarrow 0000$
    - Index: next $\log_2$(number-of-sets) = $\log_2(4) = 2 \rightarrow 0000$
    - Tag: rest = 4 – 1 – 2 = 1 → 0000

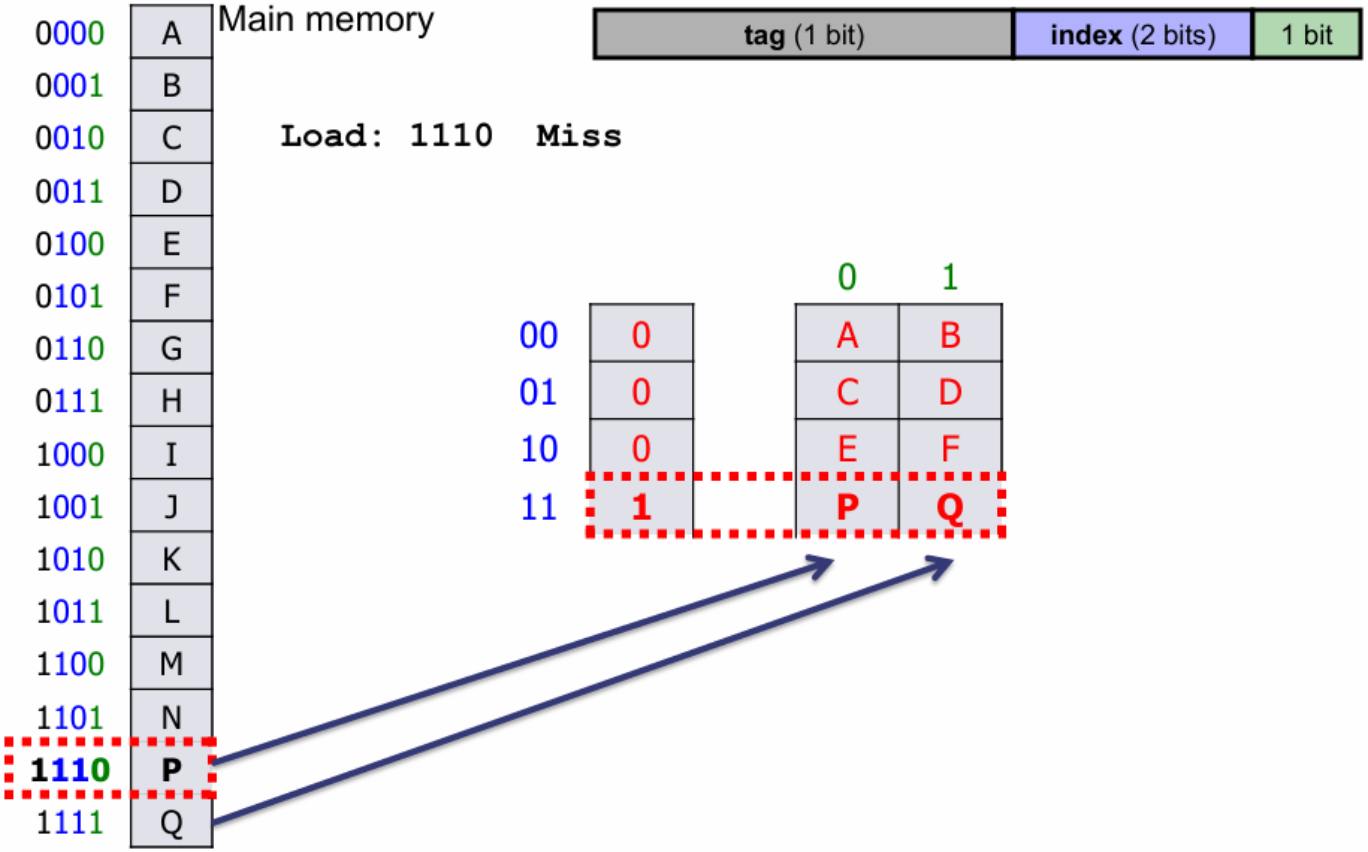
4-bit Address, 8B Cache, 2B Blocks



4-bit Address, 8B Cache, 2B Blocks

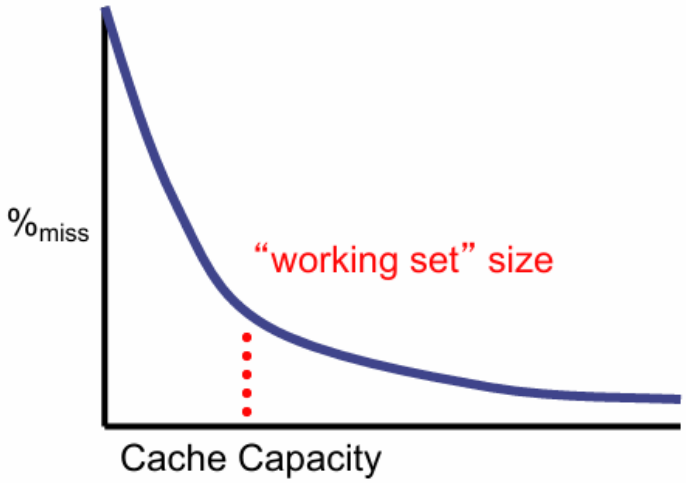


4-bit Address, 8B Cache, 2B Blocks



Capacity and Performance

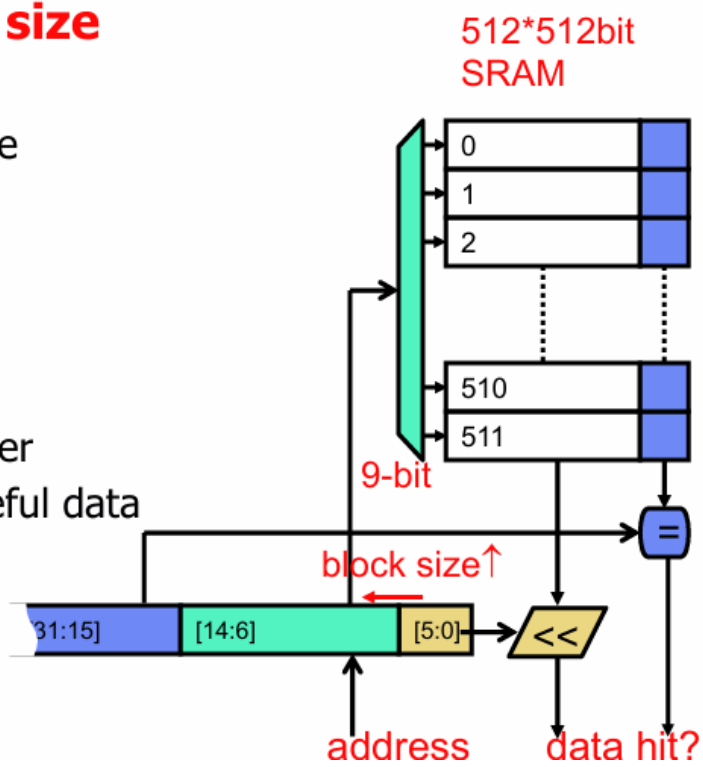
- Simplest way to reduce $\%_{miss}$: increase capacity
  - + Miss rate decreases monotonically
    - **"Working set"**: insns/data program is actively using
    - Diminishing returns
  - – However t_{access} increases
    - Latency proportional to sqrt(capacity)
  - t_{avg} ?



- Given capacity, manipulate $\%_{miss}$ by changing **organization**

Block Size

- Given capacity, manipulate $\%_{miss}$ by changing organization
- One option: increase **block size**
  - Exploit **spatial locality**
  - Notice index/offset bits change
  - Tag remain the same
- Ramifications
  - \+ Reduce $\%_{miss}$ (up to a point)
  - \+ Reduce tag overhead (why?)
  - – Potentially useless data transfer
  - – Premature replacement of useful data

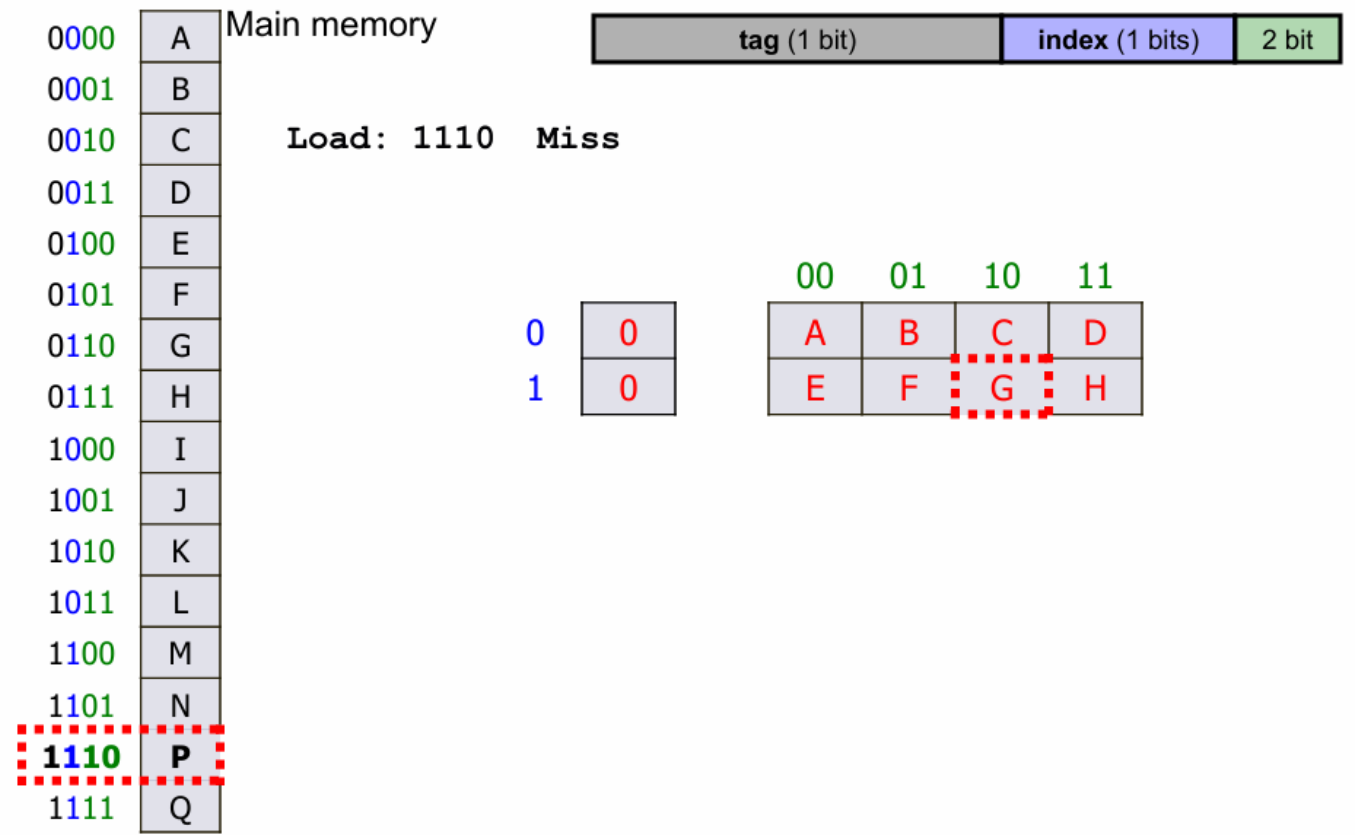


Block Size and Tag Overhead

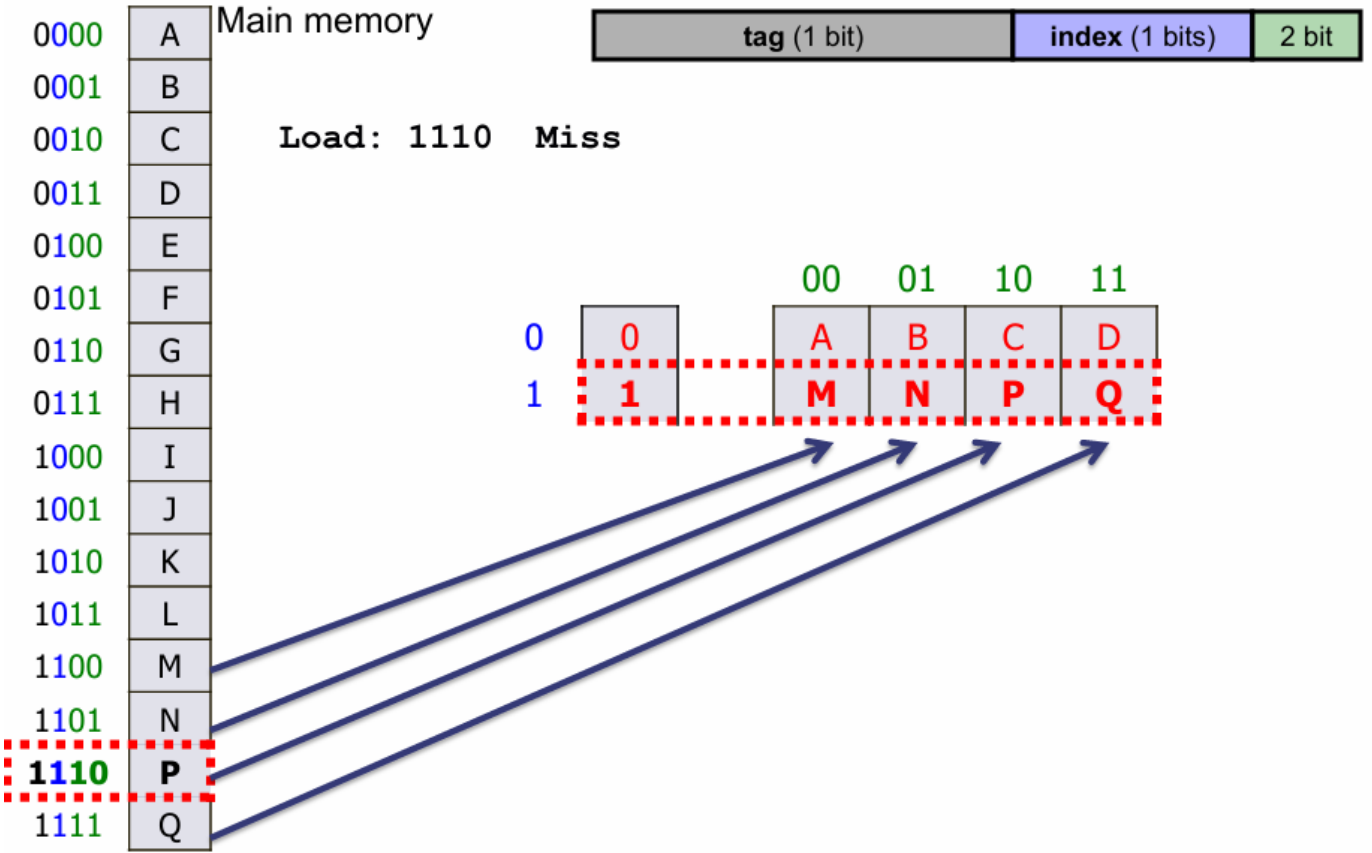
- 4KB cache with 1024 4B blocks?
  - 4B blocks → 2-bit offset, 1024 frames → 10-bit index
  - 32-bit address – 2-bit offset – 10-bit index = 20-bit tag
  - 20-bit tag / 32-bit block = 63% overhead
- 4KB cache with 512 8B blocks
  - 8B blocks → 3-bit offset, 512 frames → 9-bit index
  - 32-bit address – 3-bit offset – 9-bit index = 20-bit tag
  - **20-bit tag / 64-bit block = 32% overhead**
  - Notice: tag size is same, but data size is twice as big
- A realistic example: 64KB cache with 64B blocks
  - 16-bit tag / 512-bit block = **~ 2% overhead**

- **Note: Tags are not optional**

4-bit Address, 8B Cache, 4B Blocks

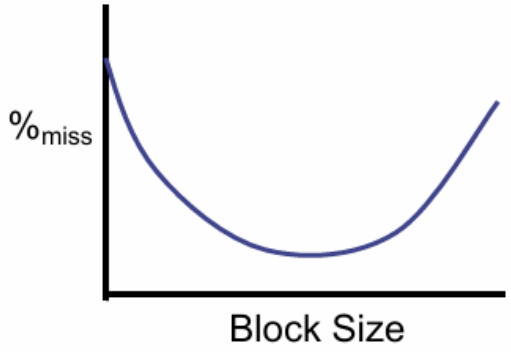


4-bit Address, 8B Cache, 4B Blocks



Effect of Block Size on Miss Rate

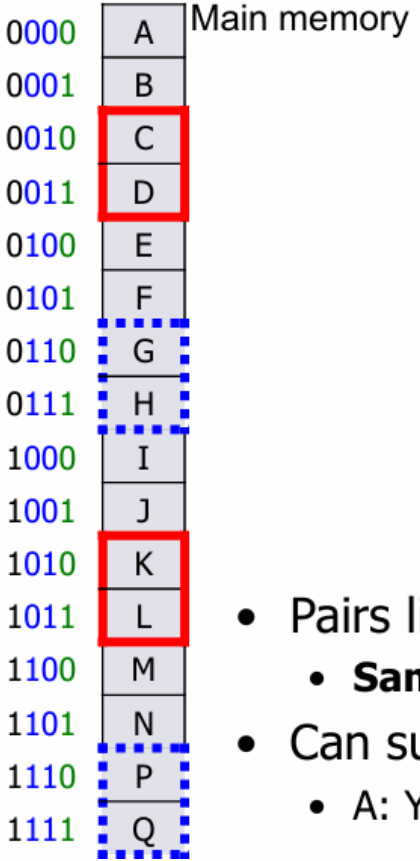
- Two effects on miss rate
  - + **Spatial prefetching (good)**
    - For blocks with adjacent addresses
    - Turns miss/miss into miss/hit pairs
  - – **Interference (bad)**
    - For blocks with non-adjacent addresses (but in adjacent frames)
    - Turns hits into misses by disallowing simultaneous residence
    - Consider entire cache as one big block
- Both effects always present
  - Spatial prefetching dominates initially
    - Depends on size of the cache
  - Good block size is 32–256B
    - Program dependent

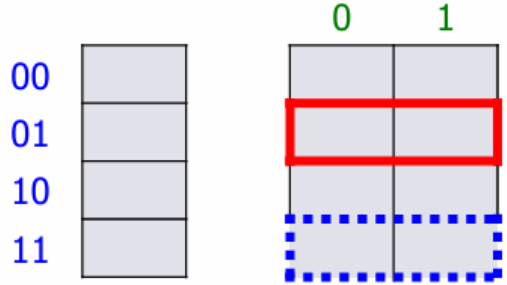


Block Size and Miss Penalty

- Does increasing block size increase t_{miss}?
  - Don't larger blocks take longer to read, transfer, and fill?
  - They do, but…
- t_{miss} of an isolated miss is not affected
  - **Critical Word First / Early Restart (CRF/ER)**
  - Requested word fetched first, pipeline restarts immediately
  - Remaining words in block transferred/filled in the background
- t_{miss}'es of a cluster of misses will suffer
  - Reads/transfers/fills of two misses can't happen at the same time
  - Latencies can start to pile up
  - This is a bandwidth problem

Cache Conflicts

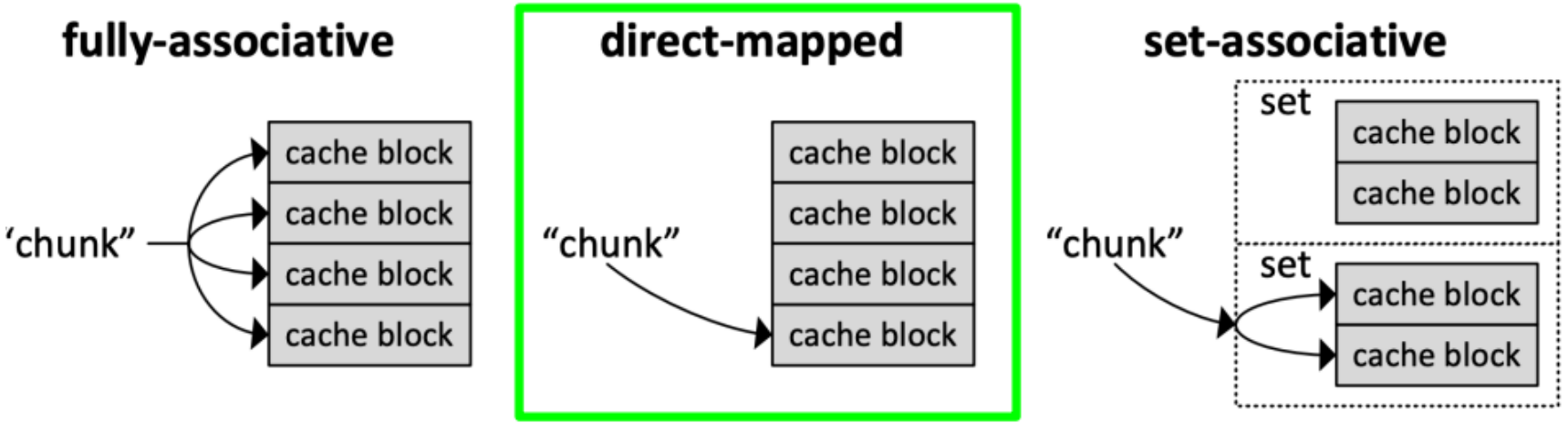




- Pairs like “0010” and “1010” **conflict**
  - **Same index!**
- Can such pairs to simultaneously reside in cache?
  - A: Yes, if we reorganize cache to do so

Direct-Mapped Cache

- **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

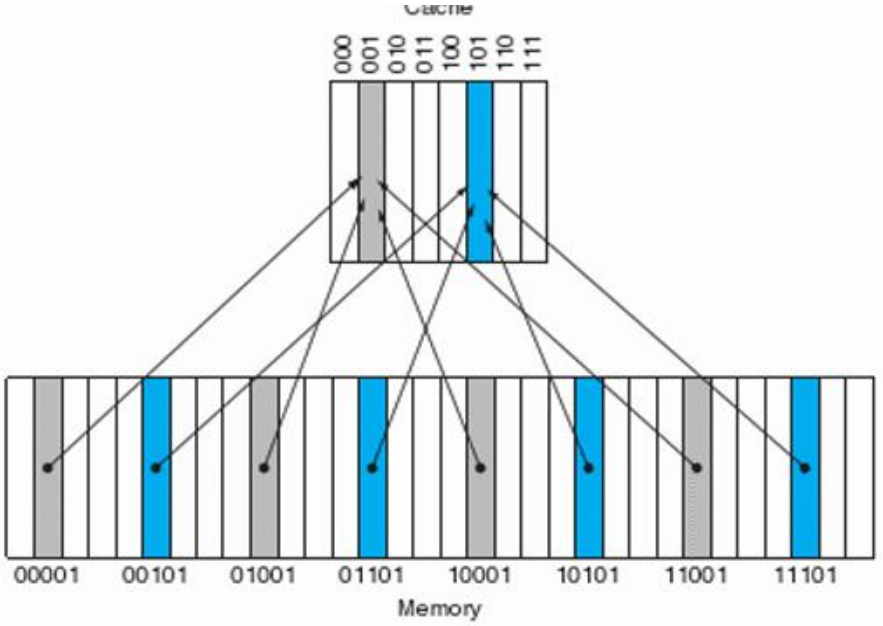
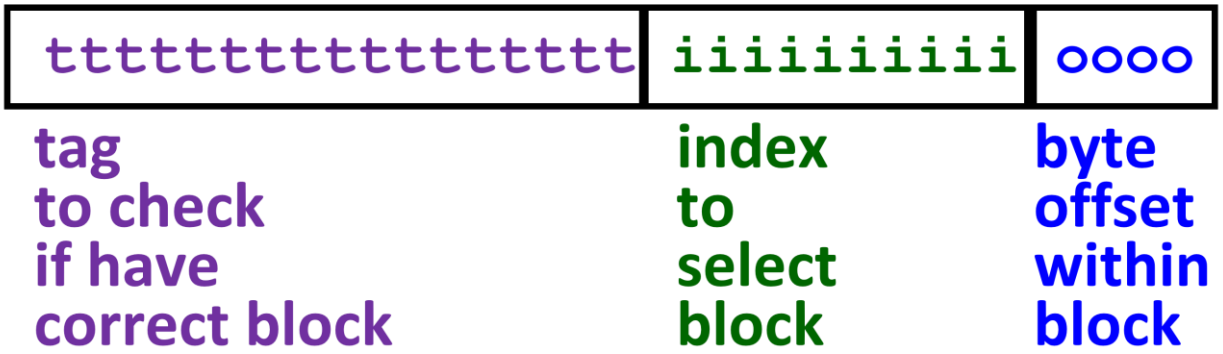


FIGURE 5.5 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the direct-mapped cache word X modulo 8. That is, the low-order $\log_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001_{two}, 01001_{two}, 10001_{two}, and 11001_{two} all map to entry 001_{two} of the cache, while addresses 00101_{two}, 01101_{two}, 10101_{two}, and 11101_{two} all map to entry 101_{two} of the cache.

Direct-Mapped Cache

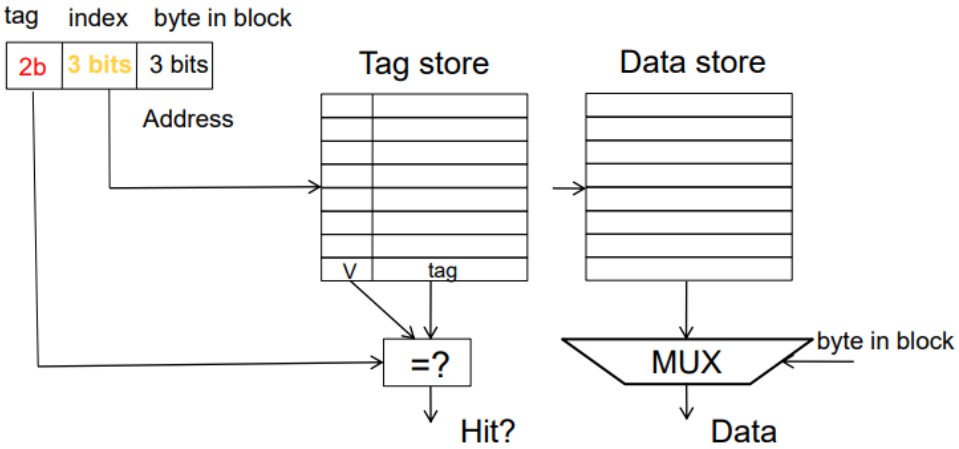
- In a **directed-mapped cache**
  - Multiple memory addresses map to the same cache index, how do we tell which one is in there?
  - What if we have a block size > 1 byte?
  - **Ans: divide memory address into three fields**

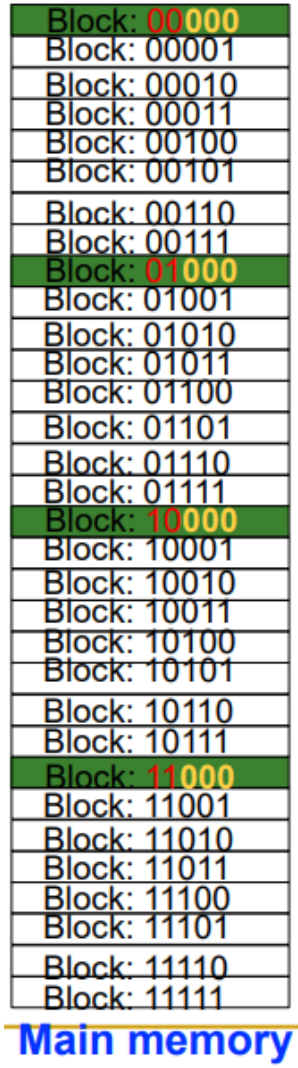


Direct-Mapped Cache

- 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





Direct-Mapped Cache

- **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

- 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

- 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 = $\frac{\text{bytes/cache}}{\text{bytes/block}}$

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

$$= 2^2 \text{ blocks/cache}$$

    - Need **2 bits** to specify this many blocks

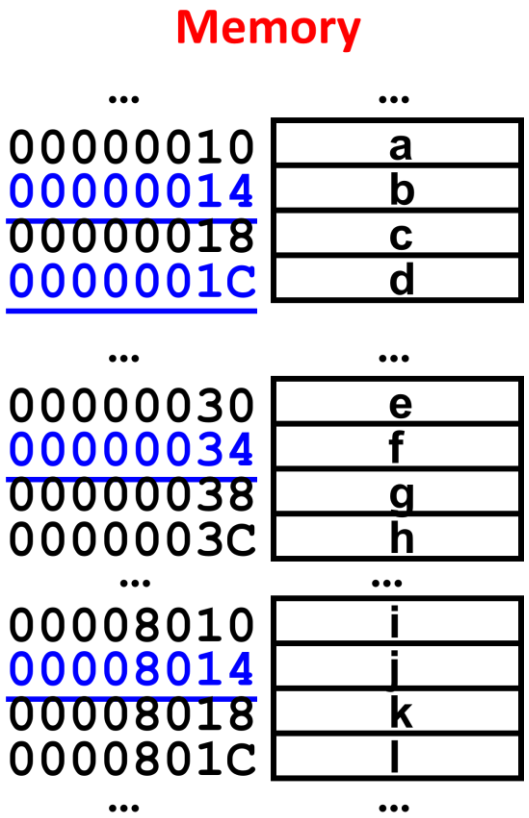
Direct-Mapped Cache Example

- 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

The tag is leftmost 29 bits of memory address

Read Data in Direct-Mapped Cache

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



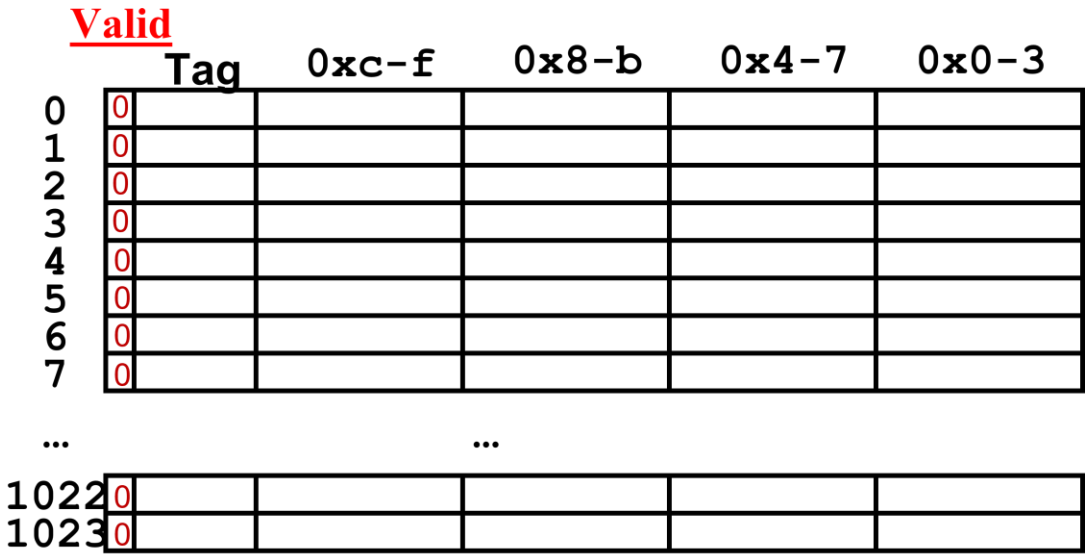
Read Data in Direct-Mapped Cache

- 4 addresses divided into

Tag	Index	Offset	
00000000000000000	000000001	0100	0x00000014
00000000000000000	000000001	1100	0x0000001C
00000000000000000	000000011	0100	0x00000034
00000000000000010	000000001	0100	0x00008014

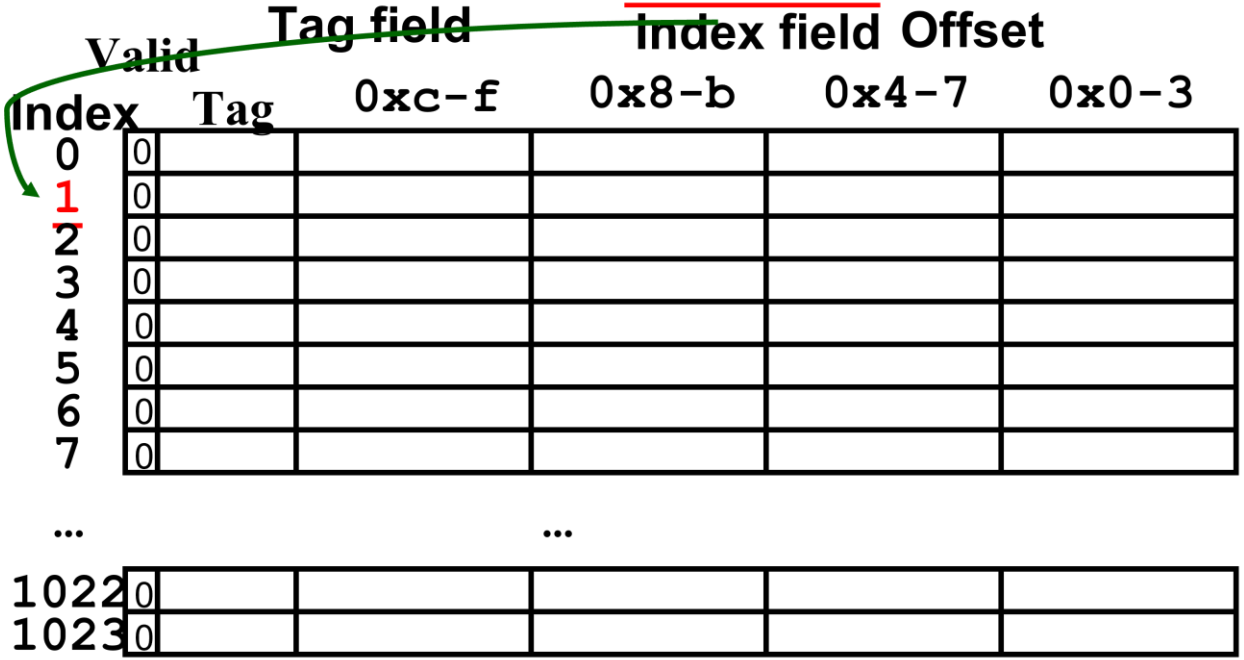
Read Data in Direct-Mapped Cache

- 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)



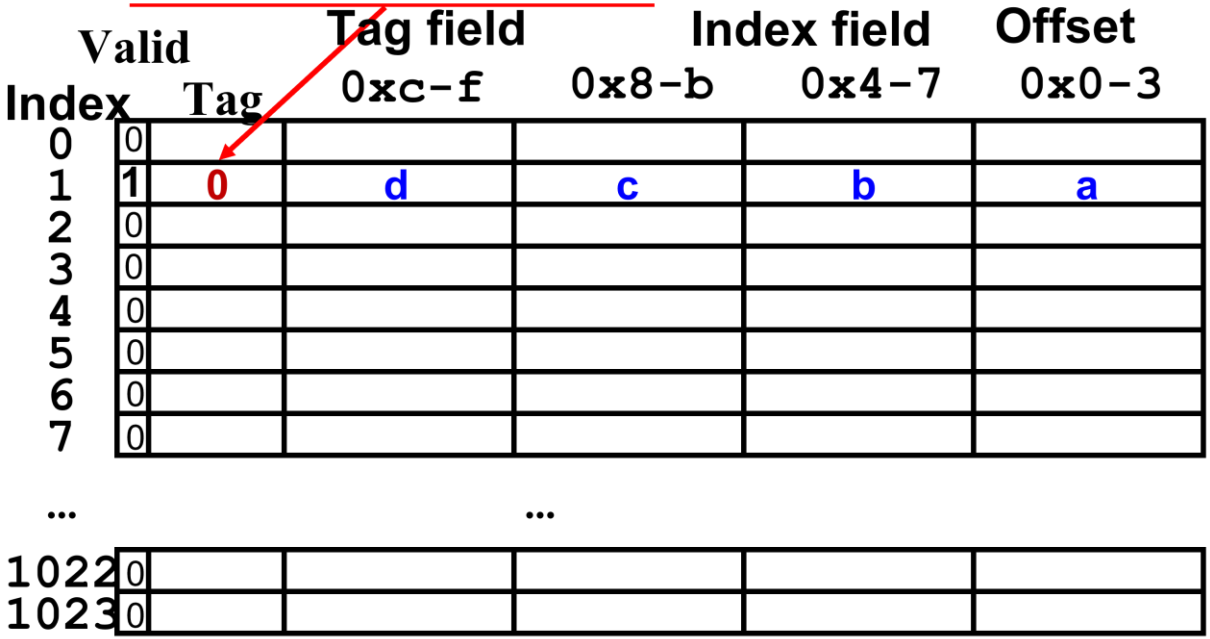
Read Data in Direct-Mapped Cache

- No valid data
  - 00000000000000000 0000000001 0100



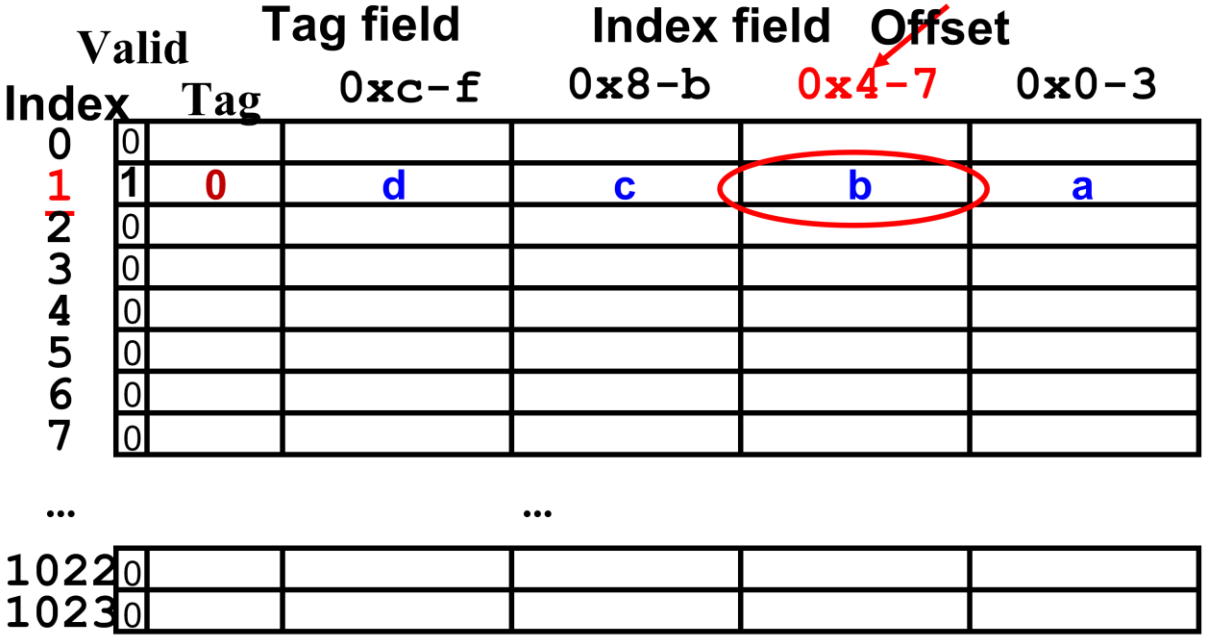
Read Data in Direct-Mapped Cache (5/15)

- Load that data into cache, setting tag, valid
  - 0000000000000000000 0000000001 0100



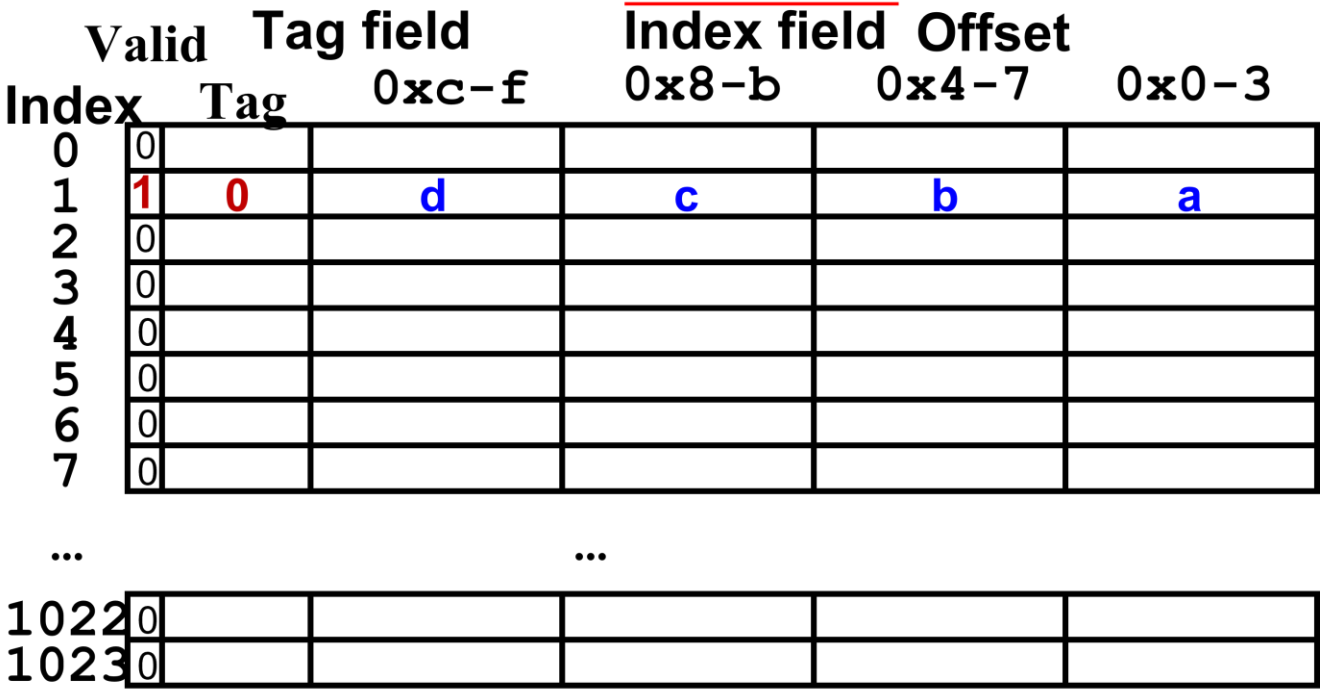
Read Data in Direct-Mapped Cache

- Read from cache at offset, return word b
  - 00000000000000000 0000000001 0100



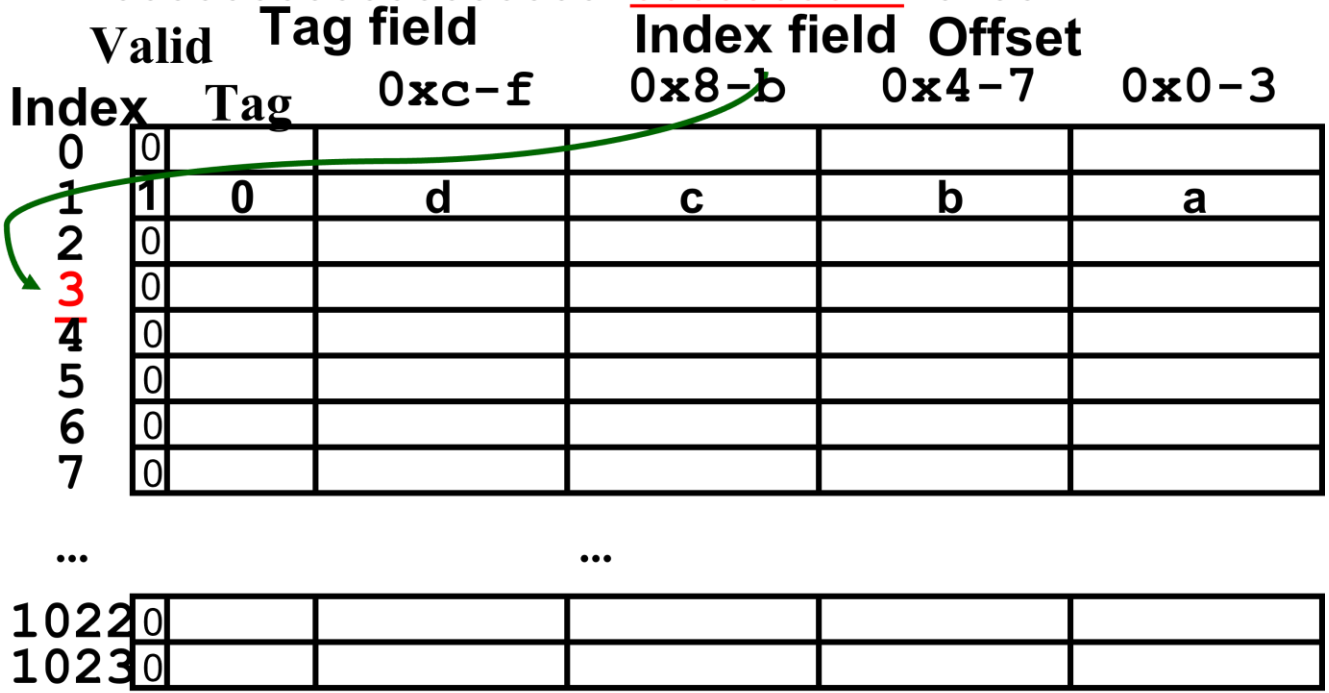
Read Data in Direct-Mapped Cache

- Read 0x00000034
  - 00000000000000000000 0000000011 0100



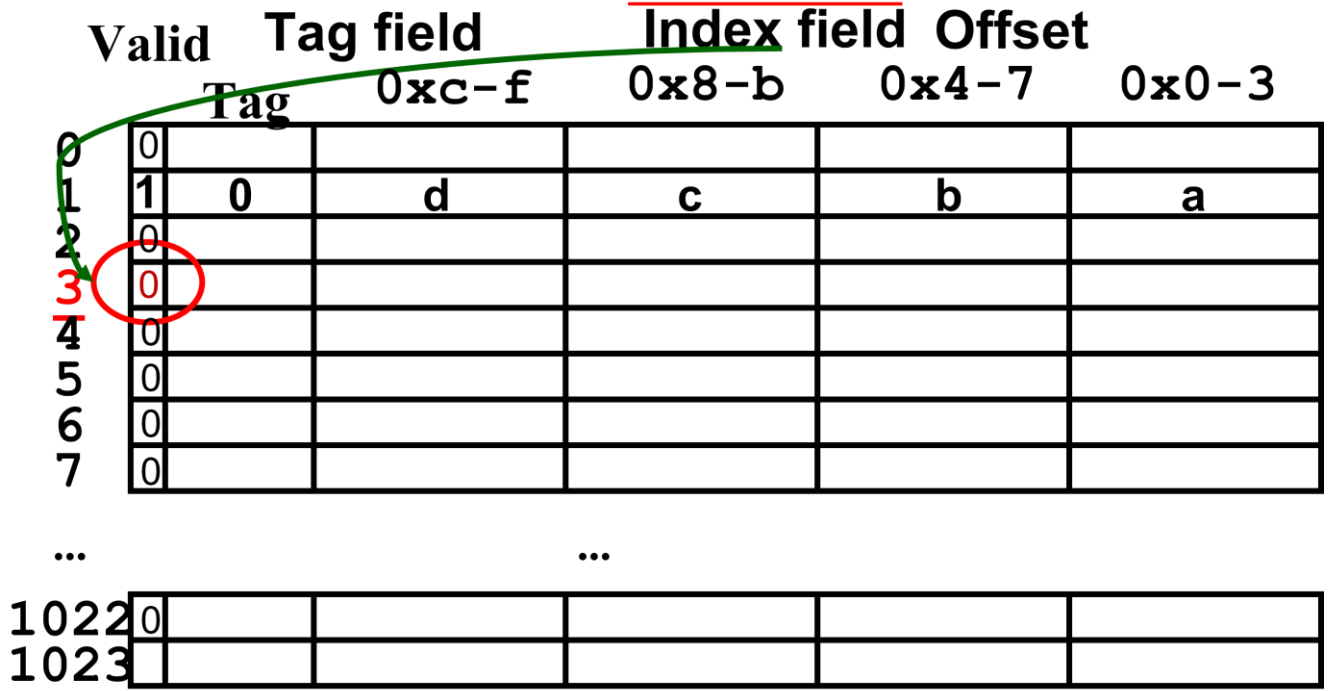
Read Data in Direct-Mapped Cache

- Read block 3
  - 00000000000000000000 0000000011 0100



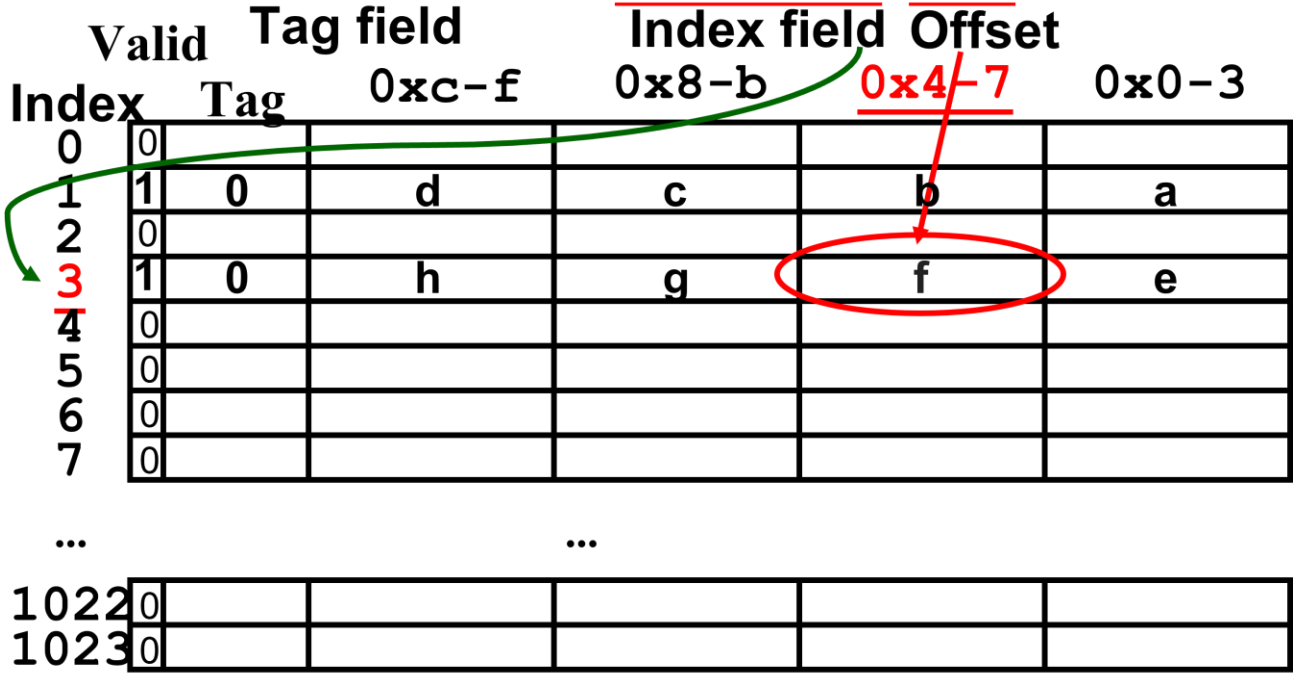
Read Data in Direct-Mapped Cache

- No valid data
  - 00000000000000000000 0000000011 0100



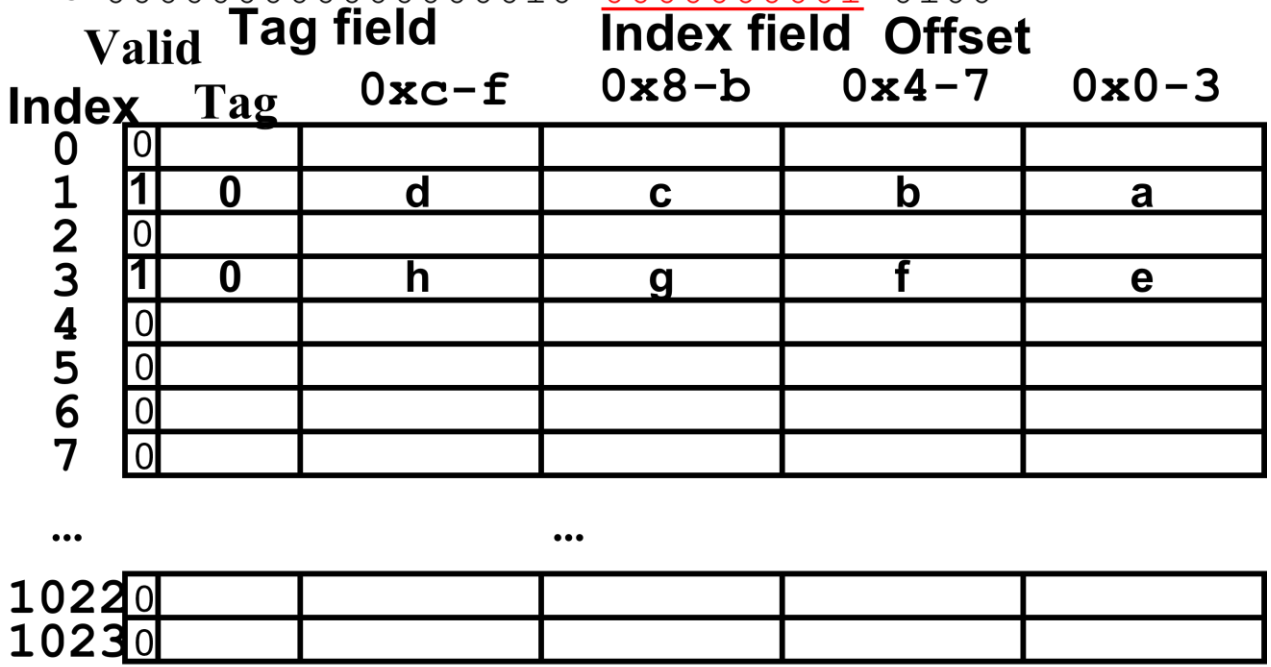
Read Data in Direct-Mapped Cache

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



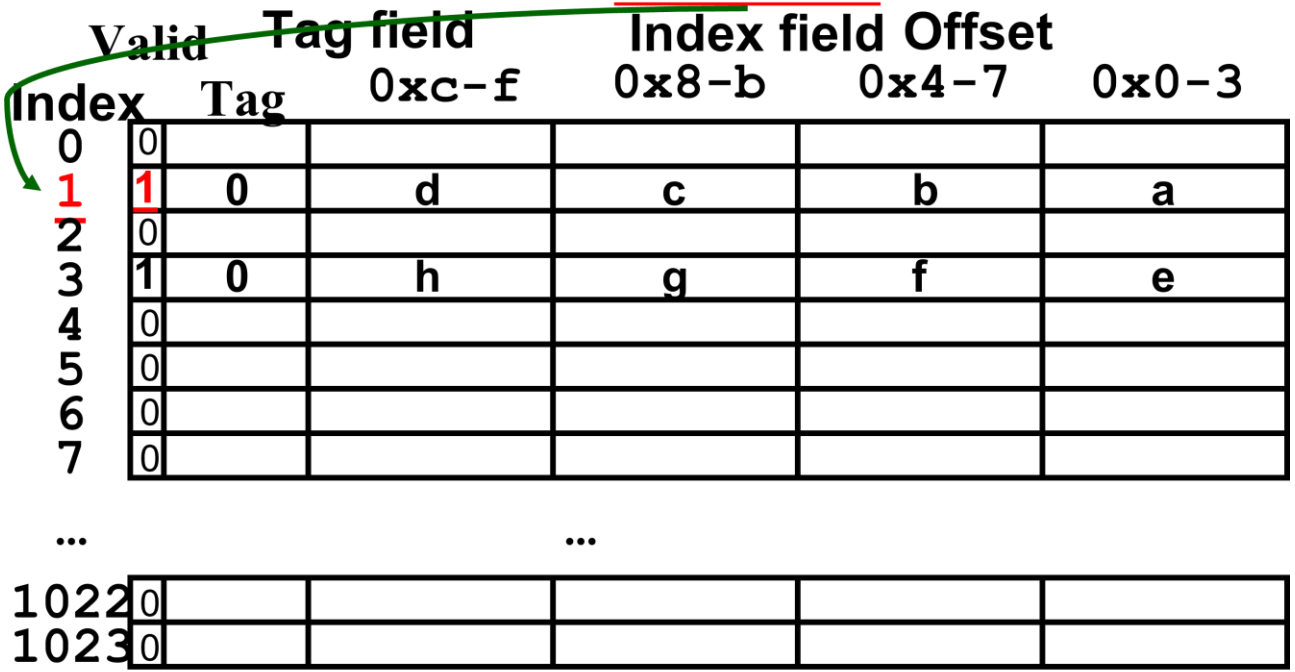
Read Data in Direct-Mapped Cache

- Read 0x00008014
  - 00000000000000010 0000000001 0100



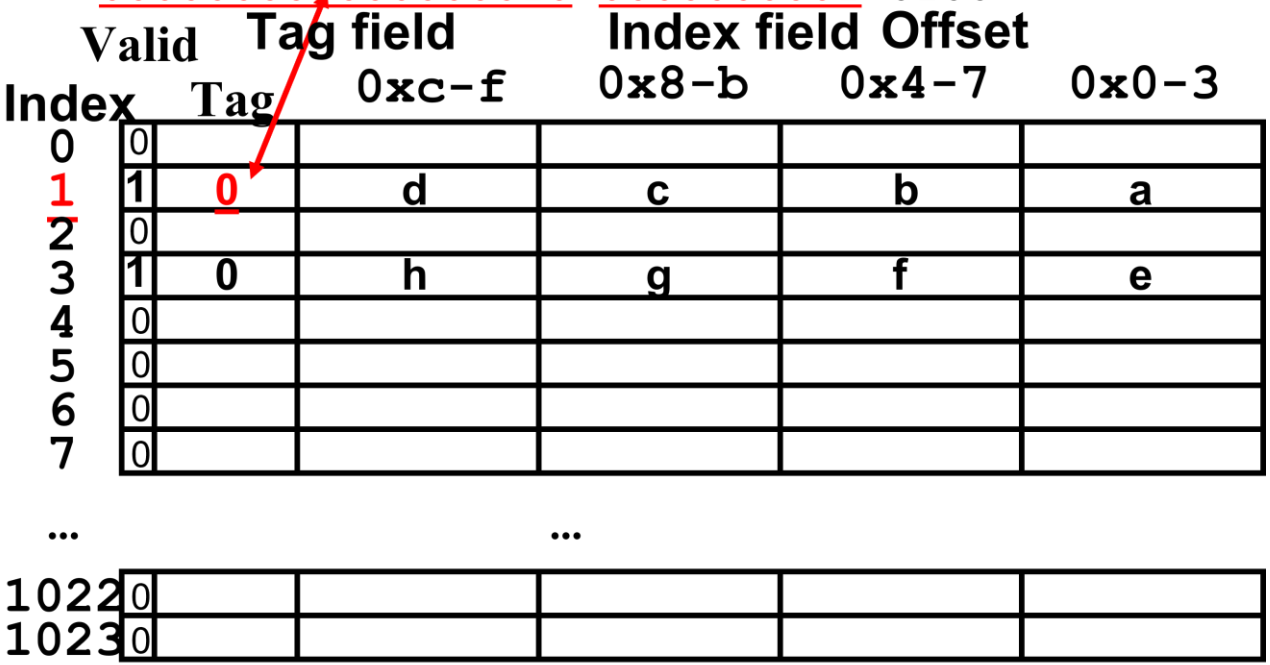
Read Data in Direct-Mapped Cache

- Read cache block 1, data is valid
  - 00000000000000010 0000000001 0100



Read Data in Direct-Mapped Cache

- Cache block 1 tag does not match (0 != 2)
  - 00000000000000010 0000000001 0100



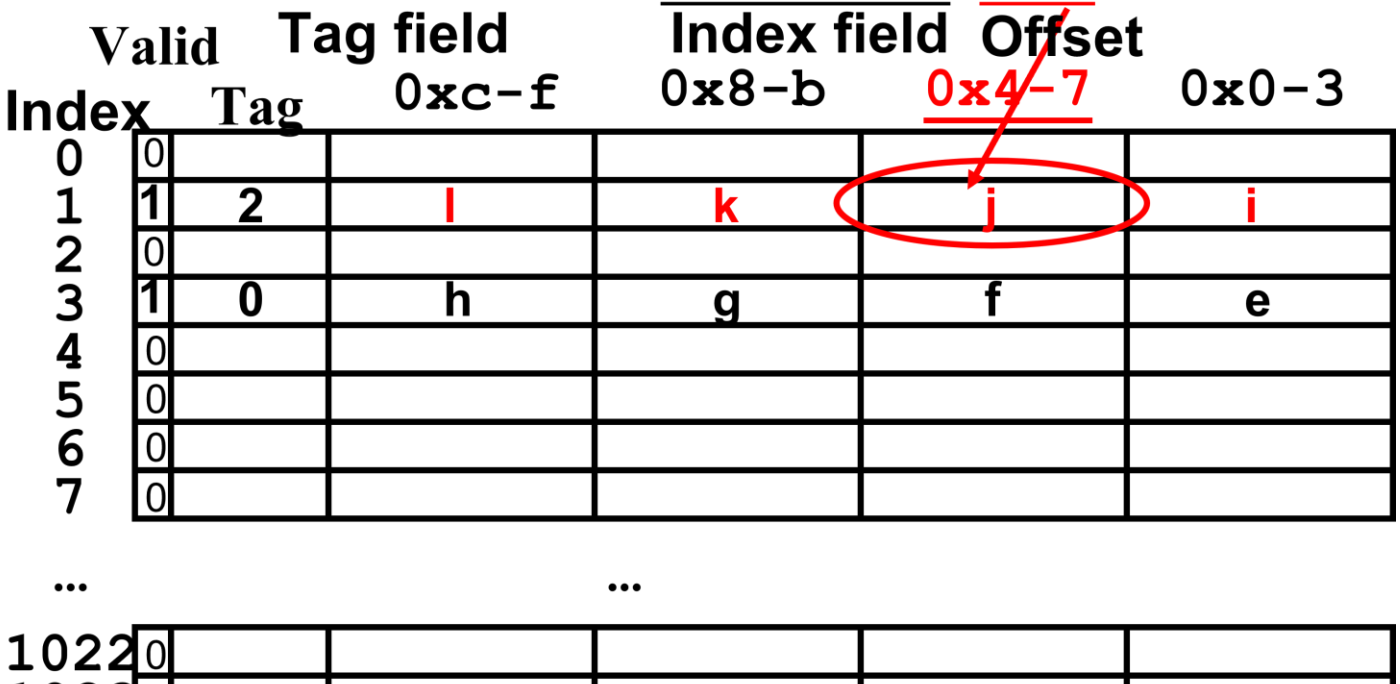
Read Data in Direct-Mapped Cache

- Miss, so replace block 1 with new data & tag
  - 0000000000000010 0000000001 0100

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
		Tag field		Index field		Offset
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

- Return word J
  - 0000000000000010 0000000001 0100



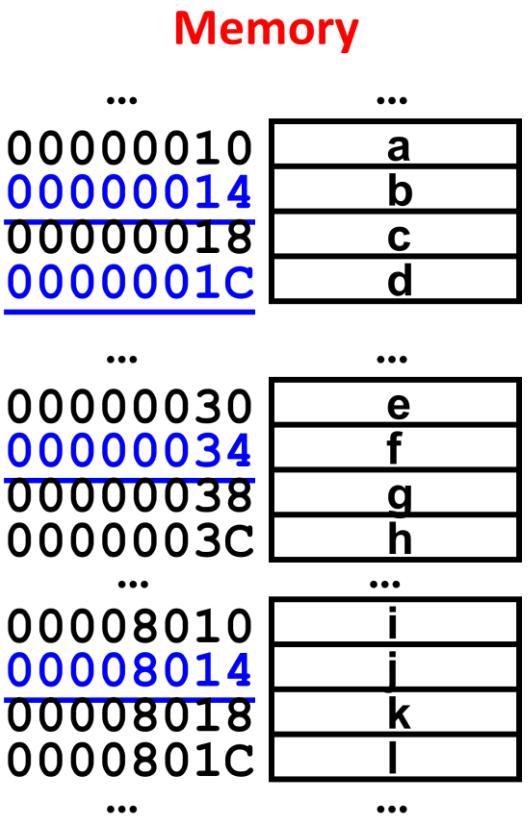
Takeaway Questions

- What is the cache status when reading?
  - Read address 0x00000030?
    - 0000000000000000 0000000011 0000
  - Read address 0x0000001C?
    - 0000000000000000 0000000001 1100

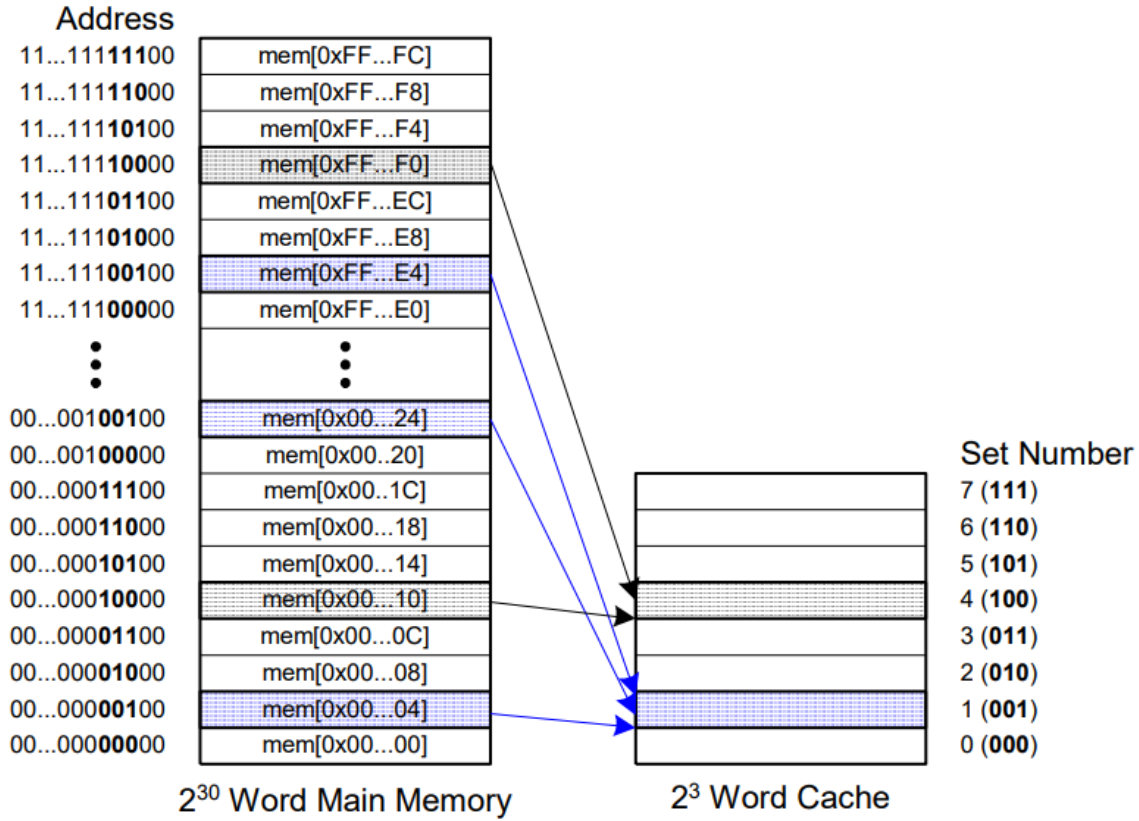
Index		Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

Takeaway Questions

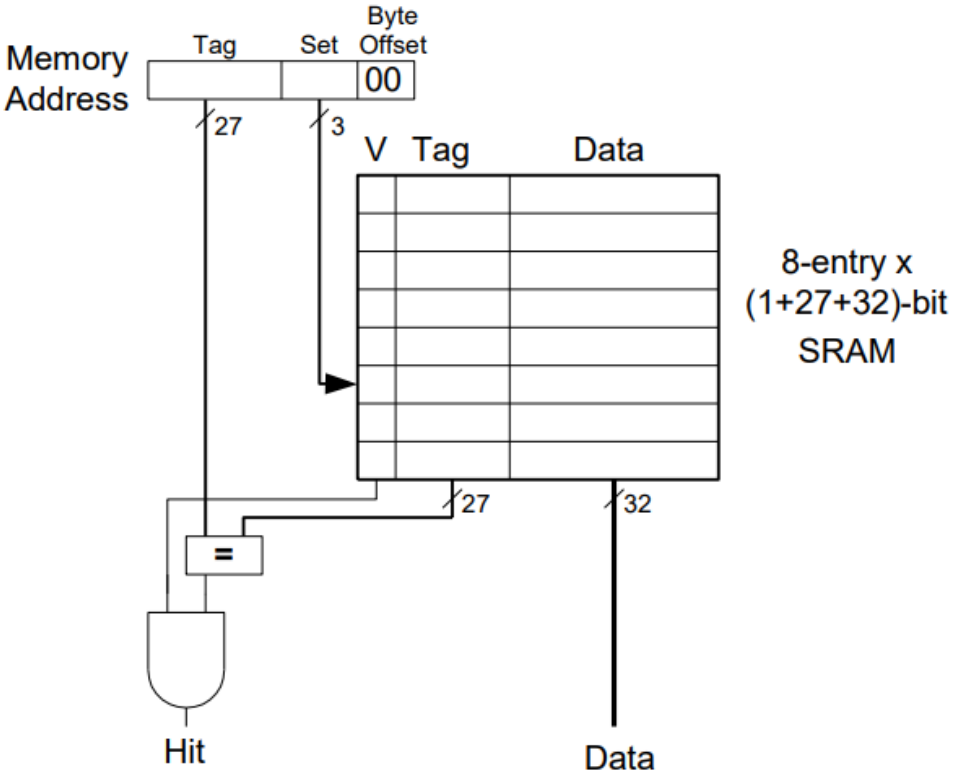
- 0x00000030 a hit
  - Index = 3, Tag matches, offset = 0, value = e
- 0x000001C a miss
  - Index = 1, tag mismatch, so replace from memory, offset = 0xc, value = d
- Read values must = memory values whether or not cached
  - 0x00000030 = e
  - 0x0000001C = d



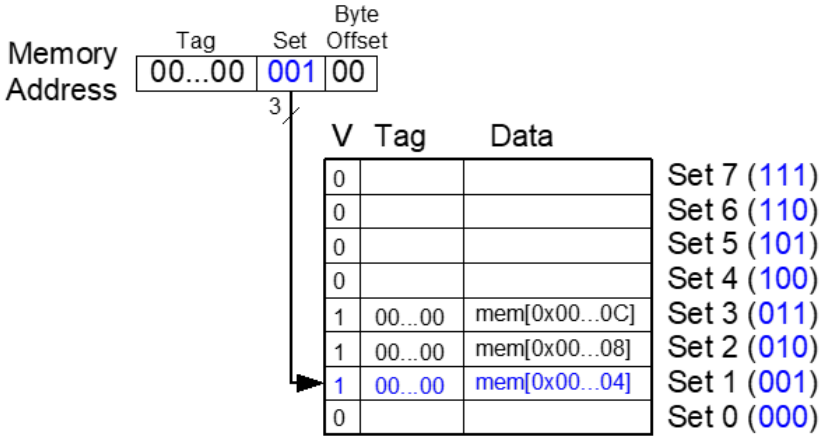
Directed-Mapped Cache Hardware



Directed-Mapped Cache Hardware



Directed-Mapped Cache Hardware

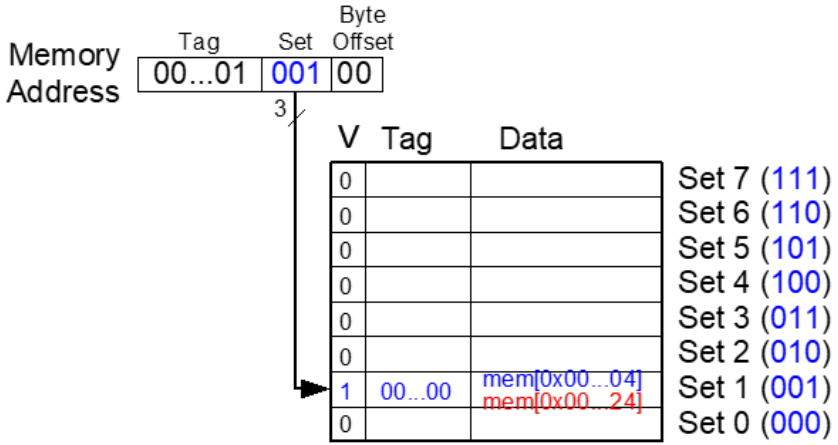



```
# RISC-V assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 3/15 = 20%

Temporal Locality
Compulsory Misses

Directed-Mapped Cache Hardware



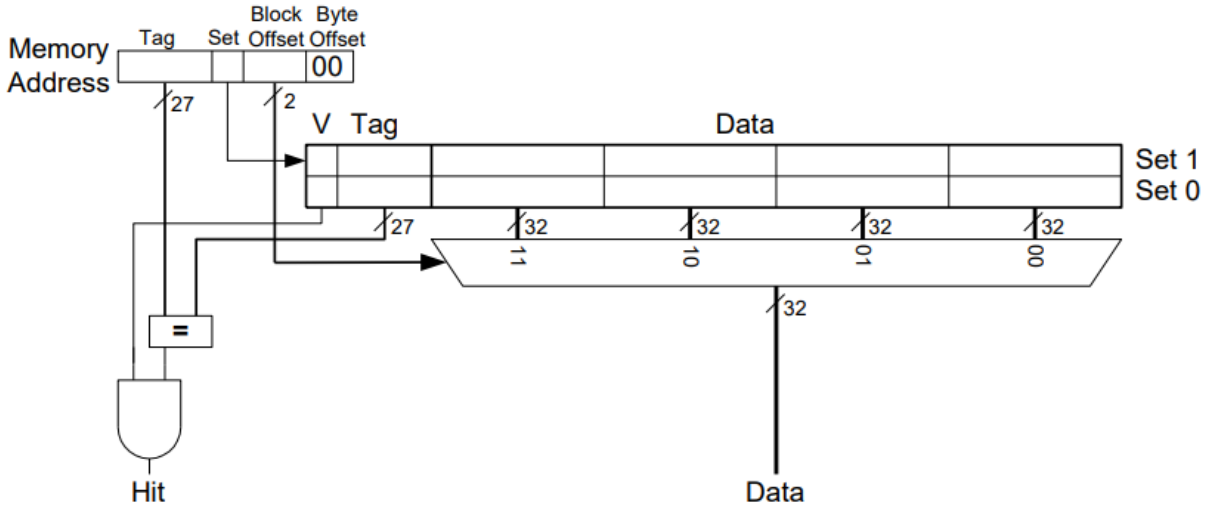

```
# RISC-V assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 10/10
= 100%

Conflict Misses

Directed-Mapped Cache Hardware

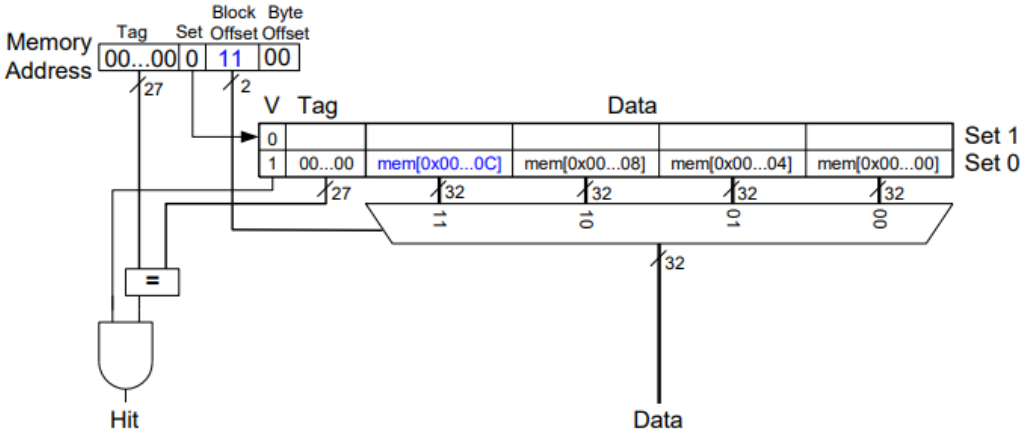
- Increase block size
  - Block size , b = 4 words
  - C = 8 words, direct mapped (1 block per set)
  - Number of blocks, B = C/b = 8/4 = 2



Directed-Mapped Cache Hardware

```
        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 =



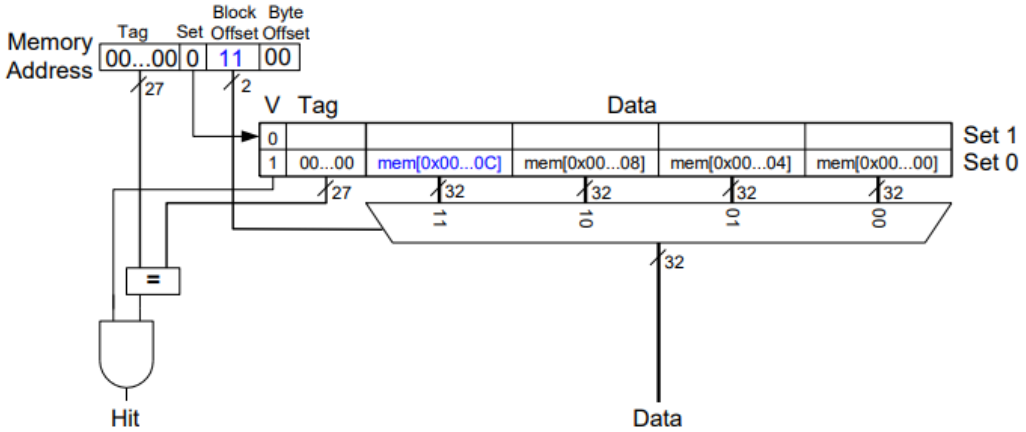
Directed-Mapped Cache Hardware

```
        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