Cache 1

CSCM601252 – **Introduction to Computer Organization**

Instructor: Erdefi Rakun

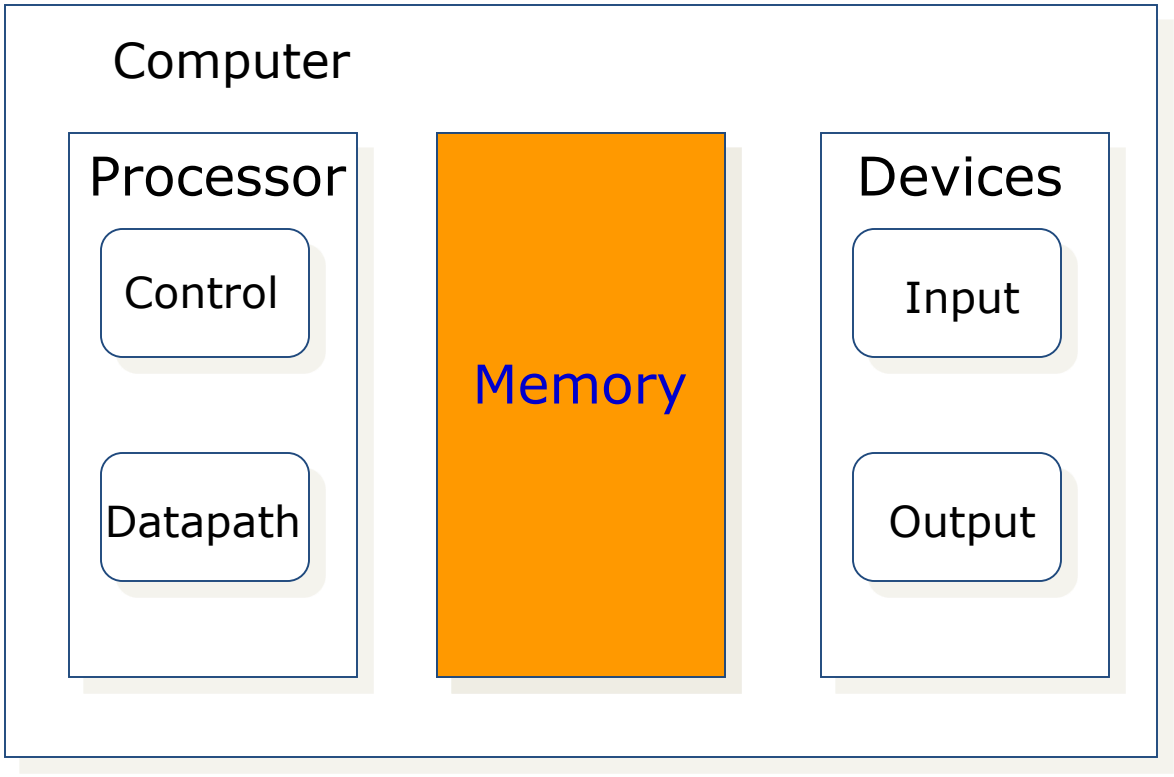
Fasilkom UI

Outline

- Memory Hierarchy
- Locality
- The Cache Principle
- Direct-Mapped Cache
- Cache Structure and Circuitry
- Write Policy

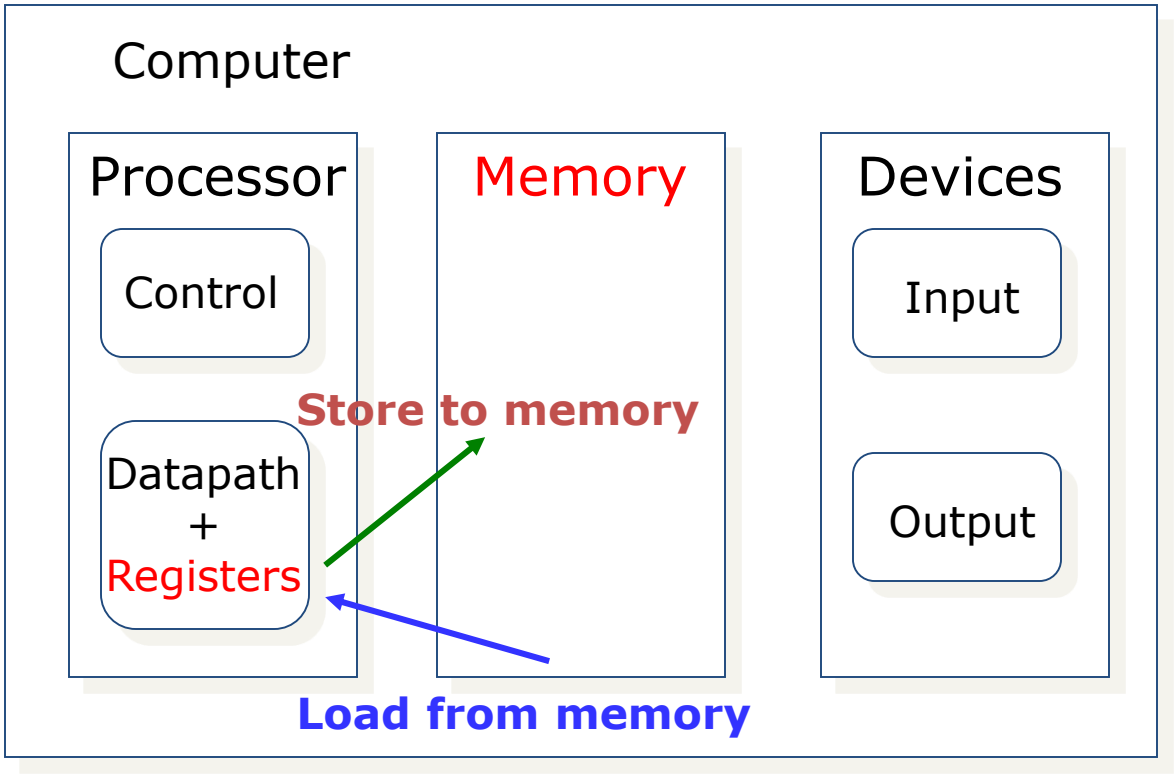
Note: These slides are taken from Aaron Tan's slide

Today's Focus



Ack: Some slides here are taken from Dr Tulika Mitra's CS1104 notes.

Data Transfer: THE BIG PICTURE



Registers are in the datapath of the processor. If operands are in memory we have to load them to processor (registers), operate on them, and store them back to memory

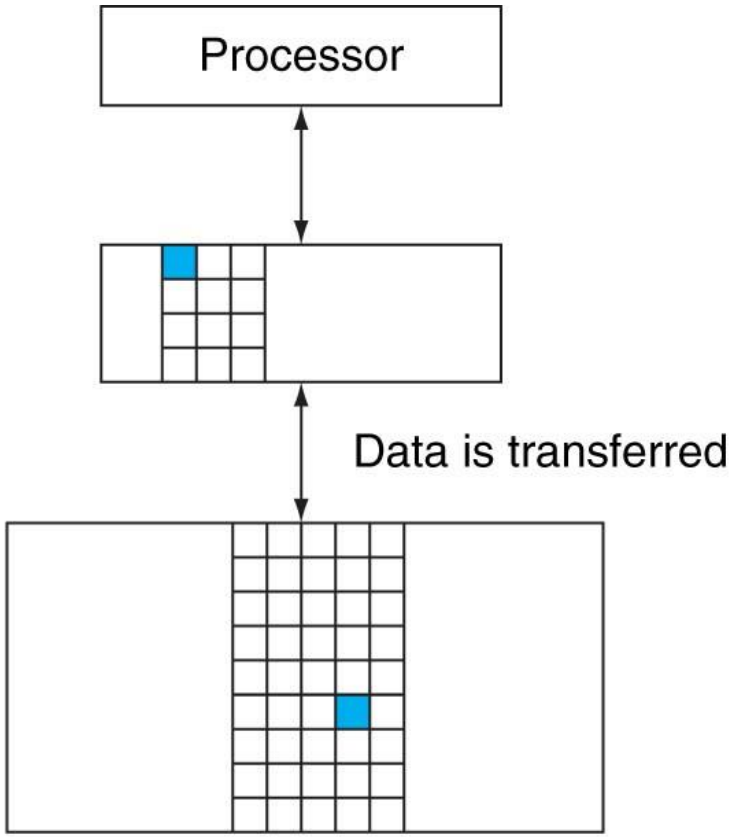
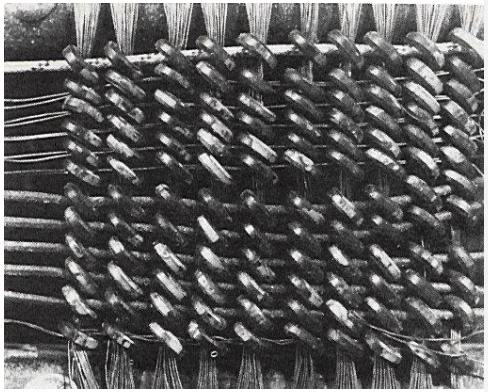


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a *block* or a *line.* Usually we transfer an entire block when we copy something between levels.

Memory Technology: 1950s

1948: Maurice Wilkes examining
EDSAC's delay line memory tubes
16-tubes each storing 32 17-bit words



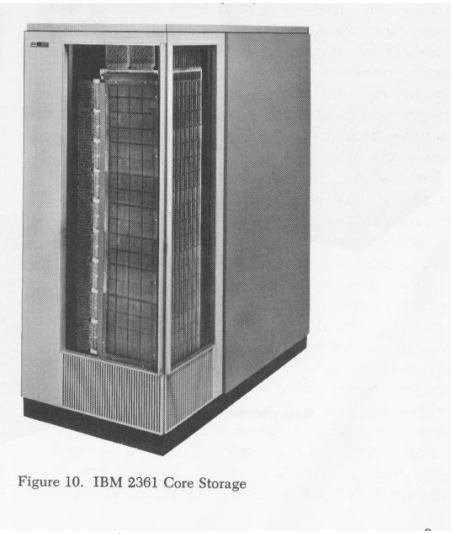


Figure 10. IBM 2361 Core Storage

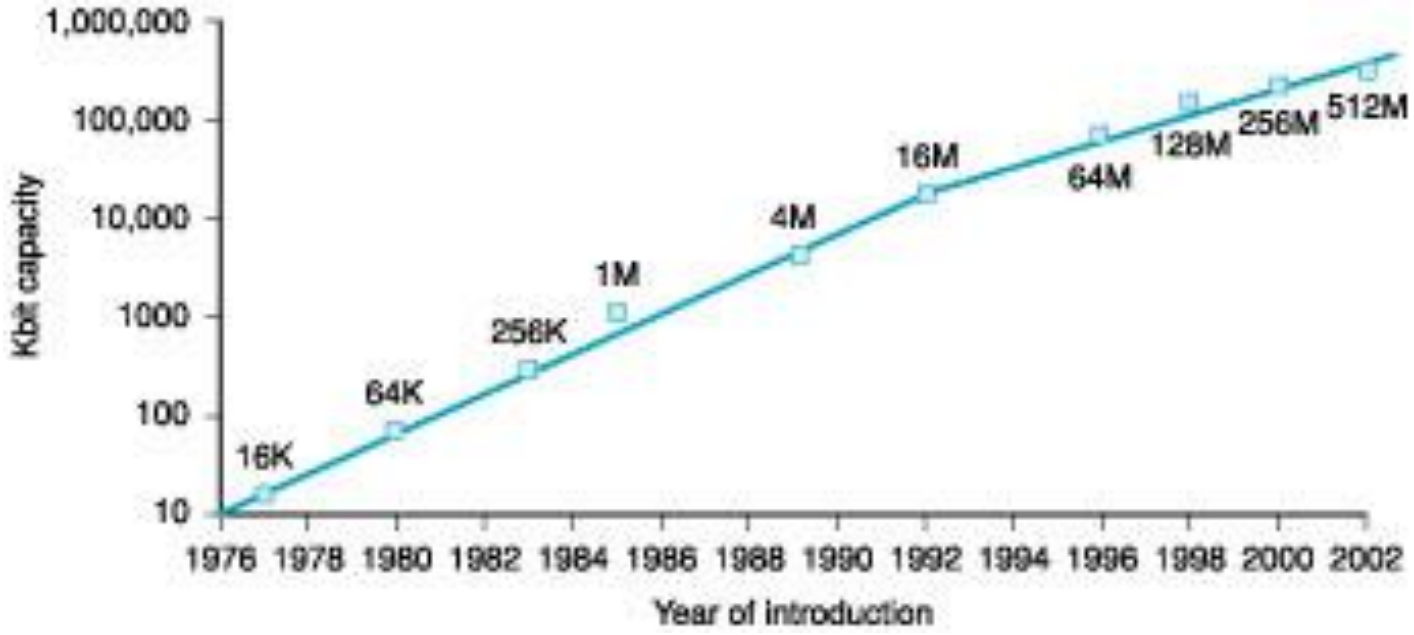
1952: IBM 2361 16KB magnetic core memory

Maurice Wilkes: 2005

Memory Technology 2005: DRAM

- Infineon Technologies: stores data equivalent to 640 books, 32,000 standard newspaper pages, and 1,600 still pictures or 64 hours of sound
- Is this latest? Try to find out from Google ☺

DRAM Capacity Growth

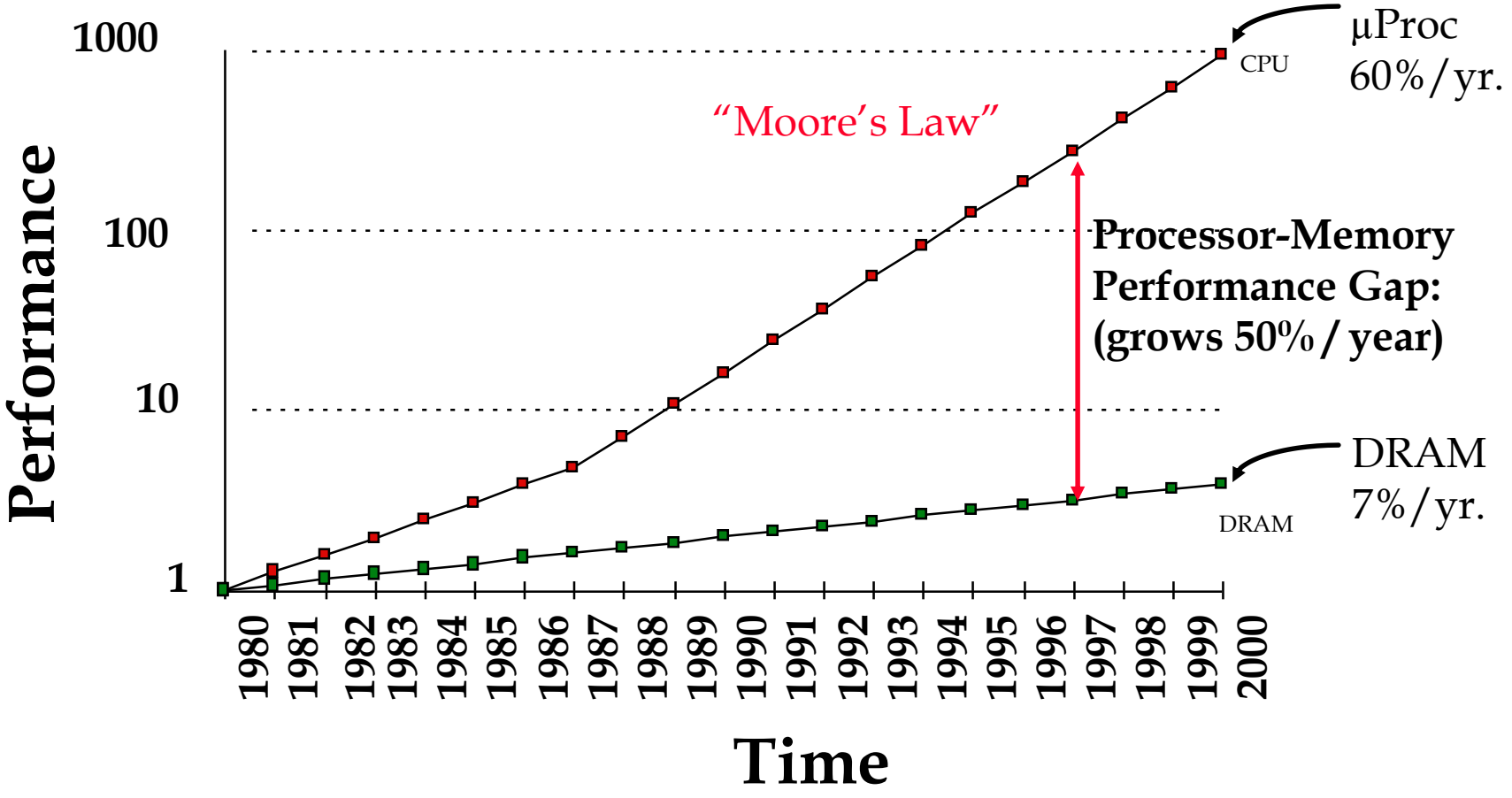


- 4X capacity increase almost every 3 years, i.e., 60% increase per year for 20 years
- Unprecedented growth in density, but we still have a problem

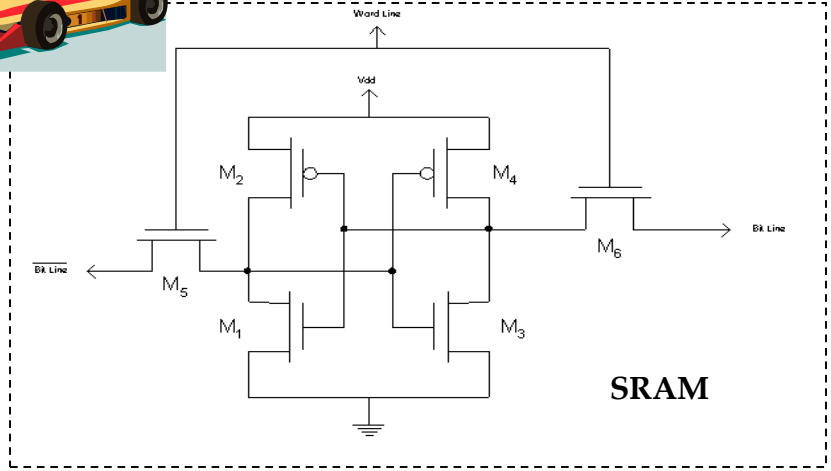
Processor-DRAM Performance Gap

Memory Wall:

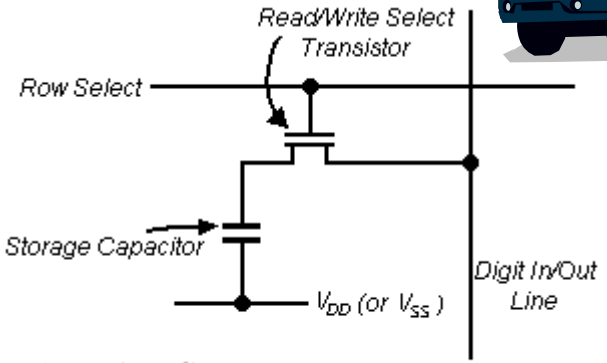
1 GHz Processor ➔ 1 ns per clock cycle but 50 ns to go to DRAM
50 processor clock cycles per memory access!!



Faster Memory Technologies: SRAM



SRAM



A DRAM Cell

- SRAM

6 transistors per memory cell → Low density

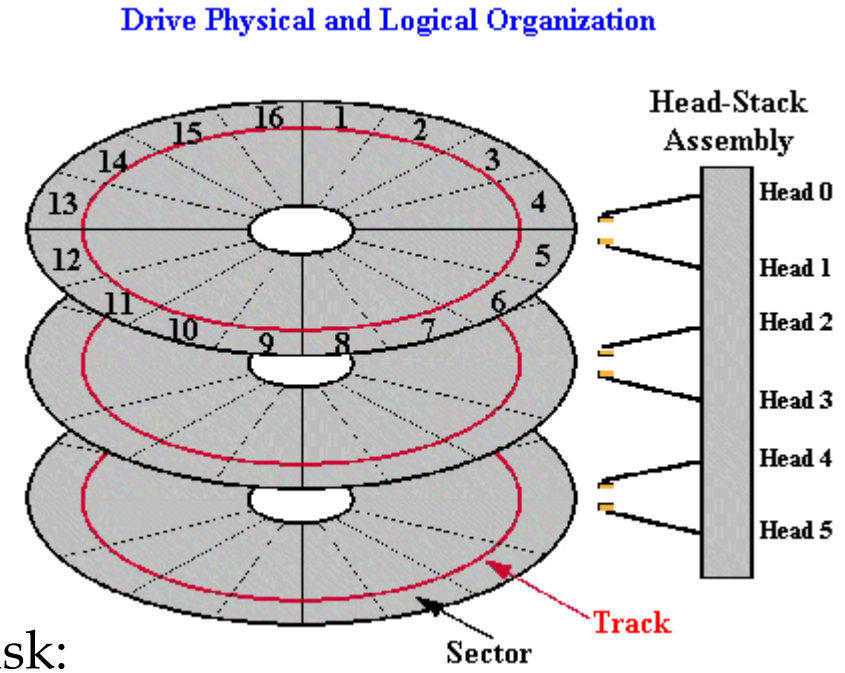
Fast access latency of 0.5 – 5 ns

- DRAM

1 transistor per memory cell → High density

Slow access latency of 50-70ns

Slow Memory Technologies: Magnetic Disk



Typical high-end hard disk:

Average latency: 5-20 ms
Capacity: 250GB

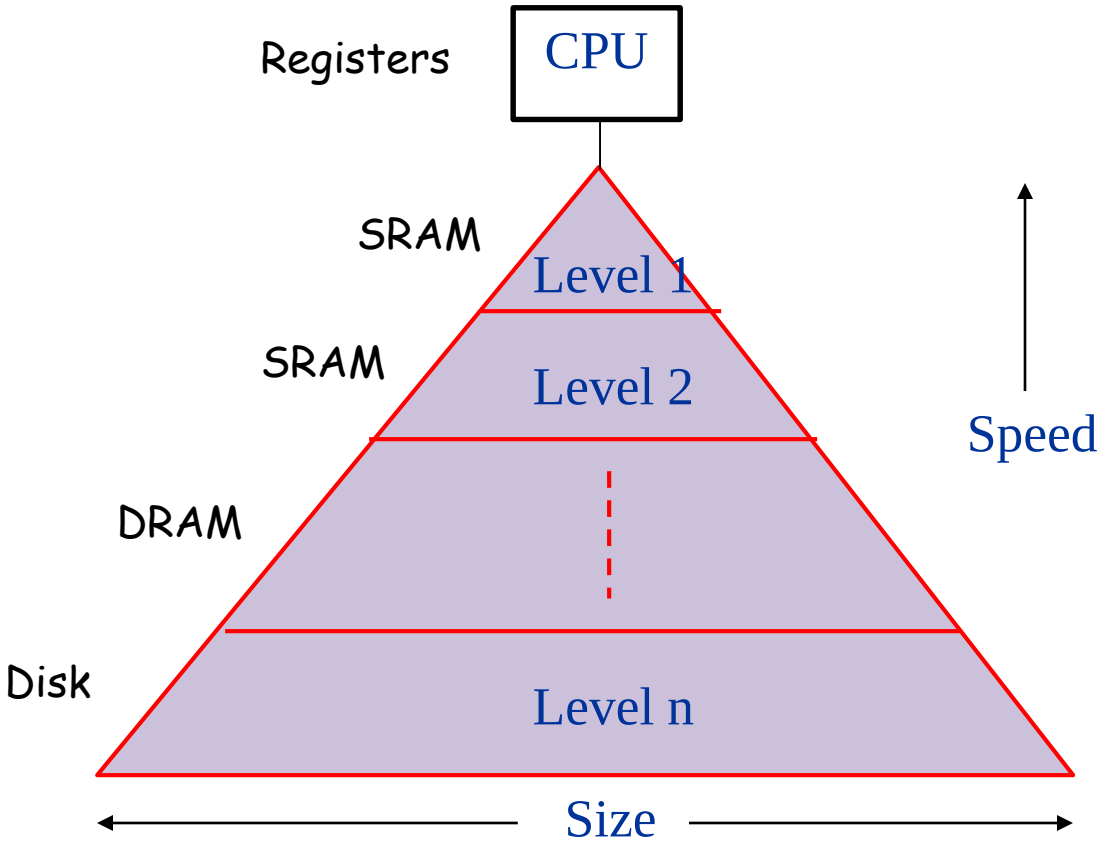
QUALITY vs QUANTITY

	Capacity	Latency	Cost/GB
Register	100s Bytes	20 ps	$$$$
SRAM	100s KB	0.5-5 ns	$$$
DRAM	100s MB	50-70 ns	$
Hard Disk	100s GB	5-20 ms	Cents
Ideal	1 GB	1 ns	Cheap

Best of Both Worlds

- What we want: A BIG and FAST memory
- Memory system should perform like 1GB of SRAM (1ns access time) but cost like 1GB of slow memory
- Key concept: Use a hierarchy of memory technologies
  - Small but fast memory near CPU
  - Large but slow memory farther away from CPU

MEMORY HIERARCHY

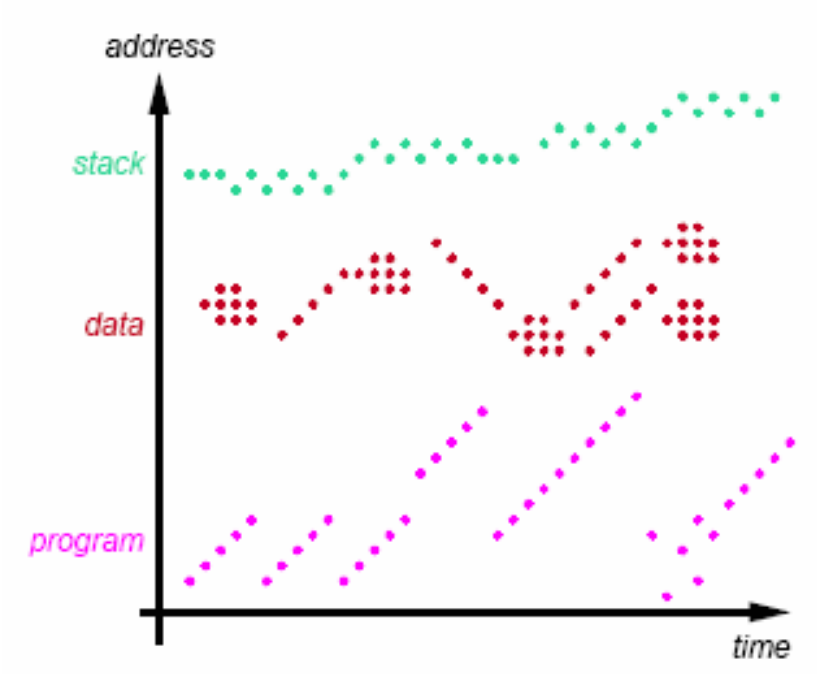


The Basic Idea

- Keep the frequently and recently used data in smaller but faster memory
- Refer to bigger and slower memory only when you cannot find data/instruction in the faster memory
- Why does it work? Principle of Locality

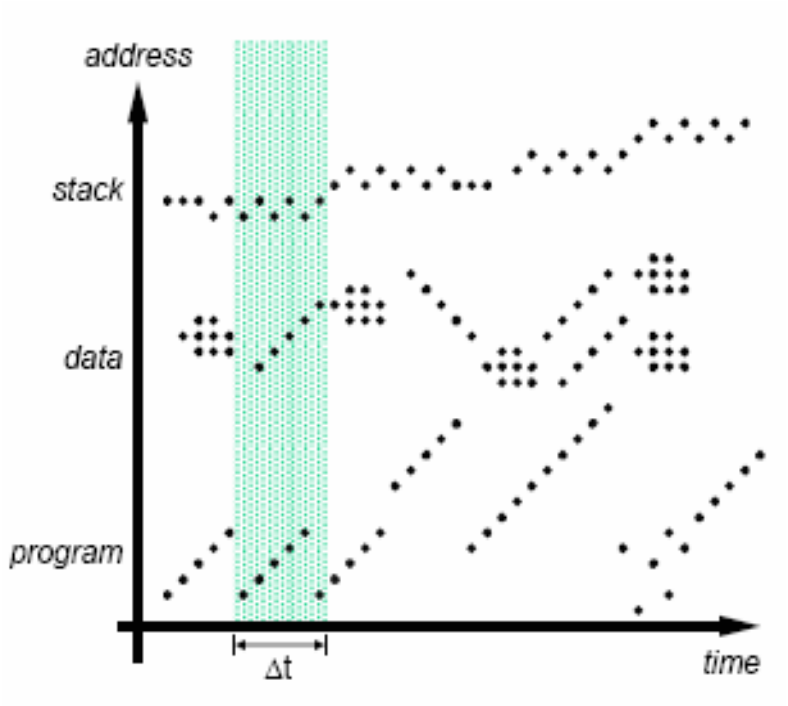
Program accesses only a small portion of the memory address space within a small time interval

LOCALITY



- Temporal locality
  - If an item is referenced, it will tend to be referenced again soon
- Spatial locality
  - If an item is referenced, nearby items will tend to be referenced soon
- Locality for instruction?
- Locality for data?

Working Set

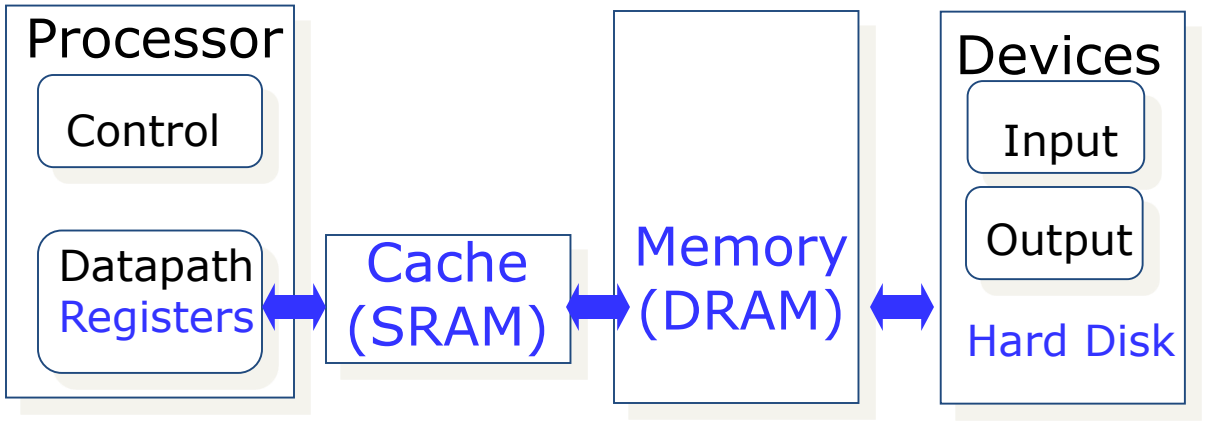


- Set of locations accessed during Δt
- Different phases of execution may use different working sets
- Want to capture the working set and keep it in the memory closest to CPU

Exploiting Memory Hierarchy (1/2)

- Visible to Programmer (Cray etc.)
  - Various storage alternatives, e.g., register, main memory, hard disk
  - Tell programmer to use them cleverly
- Transparent to Programmer (except for performance)
  - Single address space for programmer
  - Processor automatically assigns locations to fast or slow memory depending on usage patterns

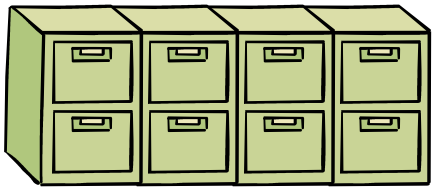
Exploiting Memory Hierarchy (2/2)



- How to make SLOW main memory appear faster?
  Cache – a small but fast SRAM near CPU
  - Often in the same chip as CPU
  - Introduced in 1960; almost every processor uses it today
  - Hardware managed: Transparent to programmer
- How to make SMALL main memory appear bigger than it is?
  Virtual memory [covered in CS2106]
  - OS managed: Again transparent to programmer

The CACHE Principle

10 minute access time



Find LEE Nicholas

10 sec access time

Look for the document in your desk first.
If not on desk, then look in the cabinet.

Cache Block/Line (1/2)

- Unit of transfer between memory and cache
- Recall: MIPS word size is 4 bytes
- Block size is typically one or more words
- Example: 16-byte block ≅ 4-word block
  32-byte block ≅ 8-word block
- Why block size is bigger than word size?
- What is the unit of transfer for virtual memory (hard disk and memory)?
- Should it be bigger than or smaller than cache blocks?

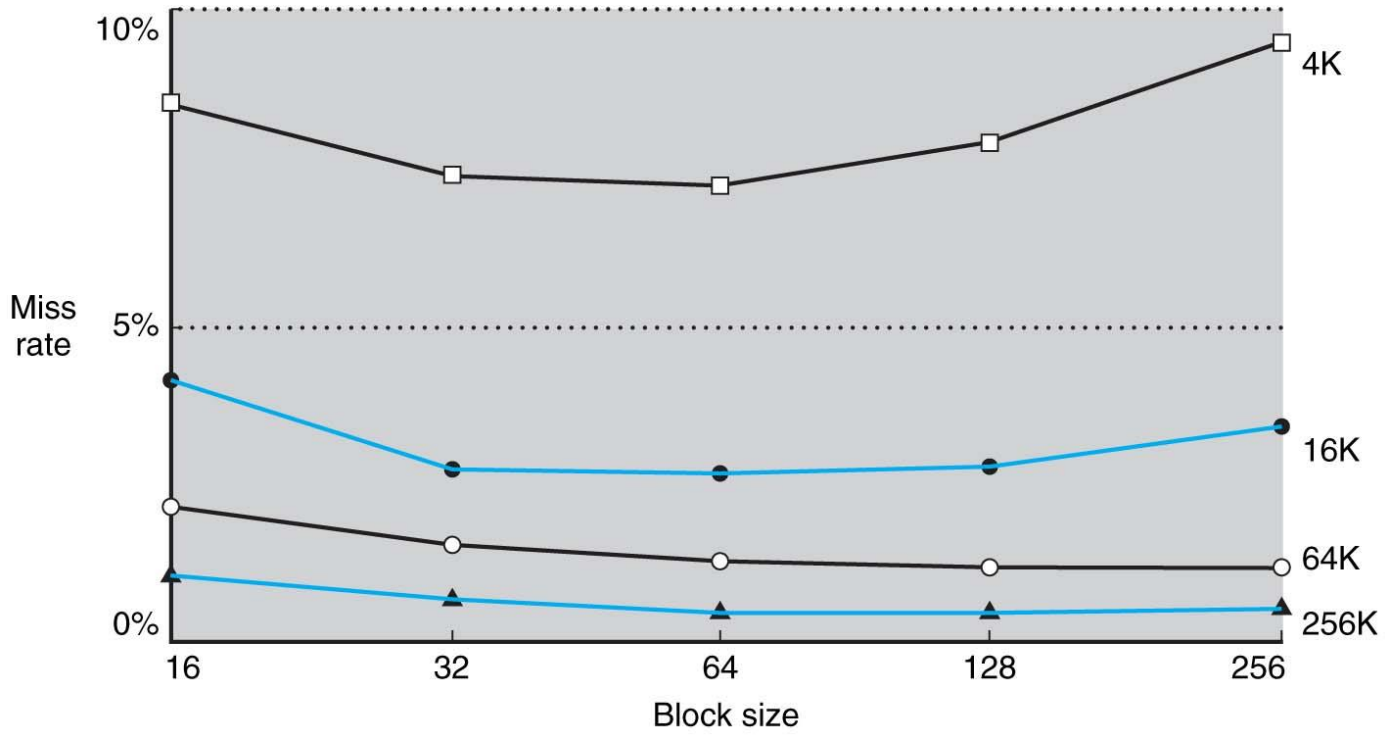
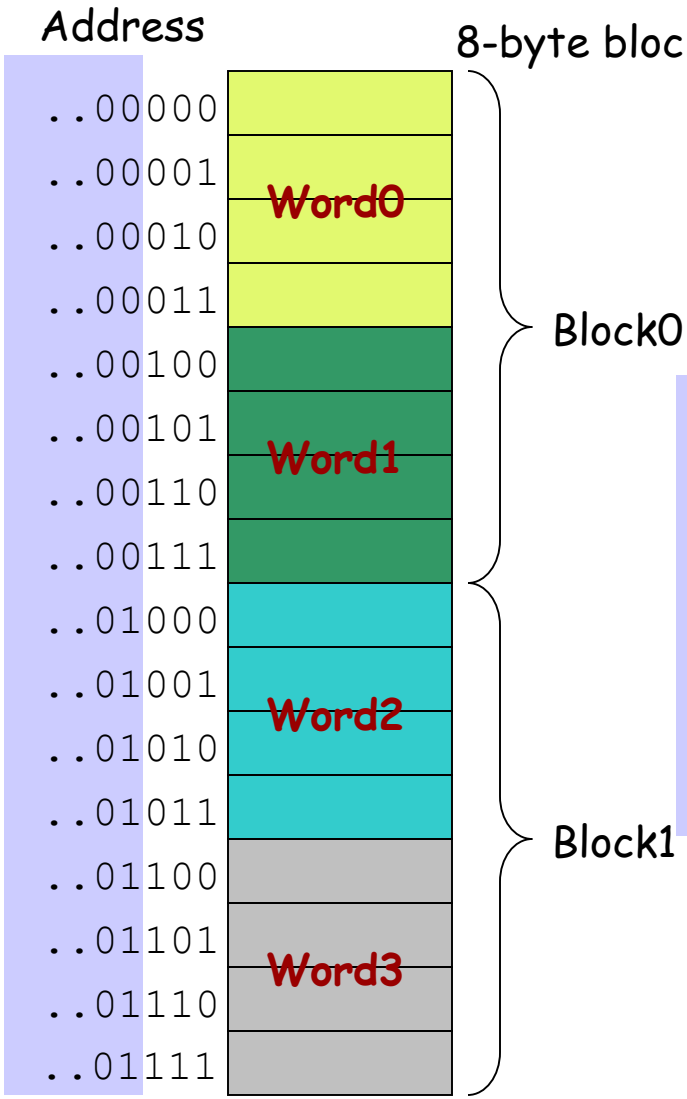
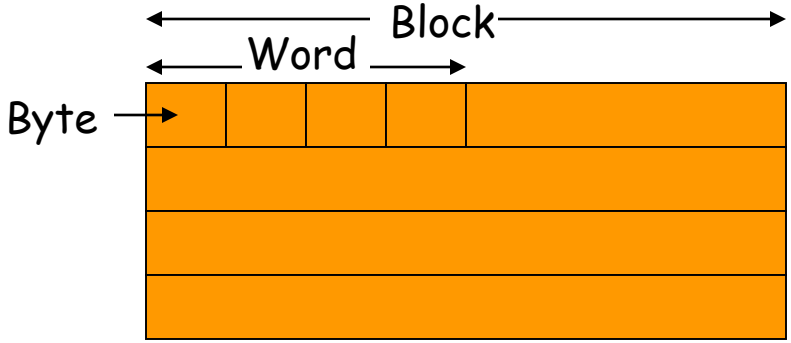


FIGURE 5.8 Miss rate versus block size. Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.) Unfortunately, SPEC2000 traces would take too long if block size were included, so this data is based on SPEC92.

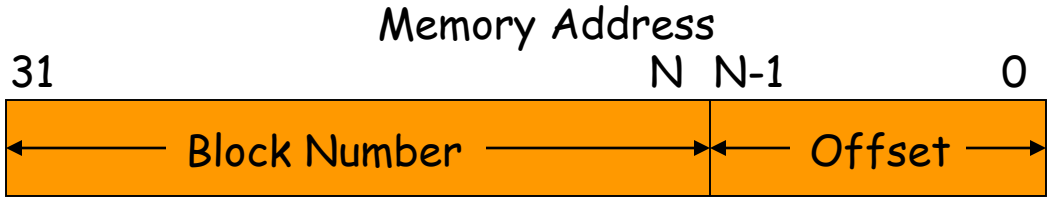
Cache Block/Line (2/2)





Observations:

1. 2^N-byte blocks are aligned at 2^N-byte boundaries
2. The addresses of words within a 2^N-byte block have identical (32-N) most significant bits (MSB). Example: For 2^3 = 8-byte block, (32-3) = 29 MSB of the memory address are identical
3. 29-MSB signifies the block number
4. N-LSB specifies the byte offset within a block



Memory Access Time

Average access time =

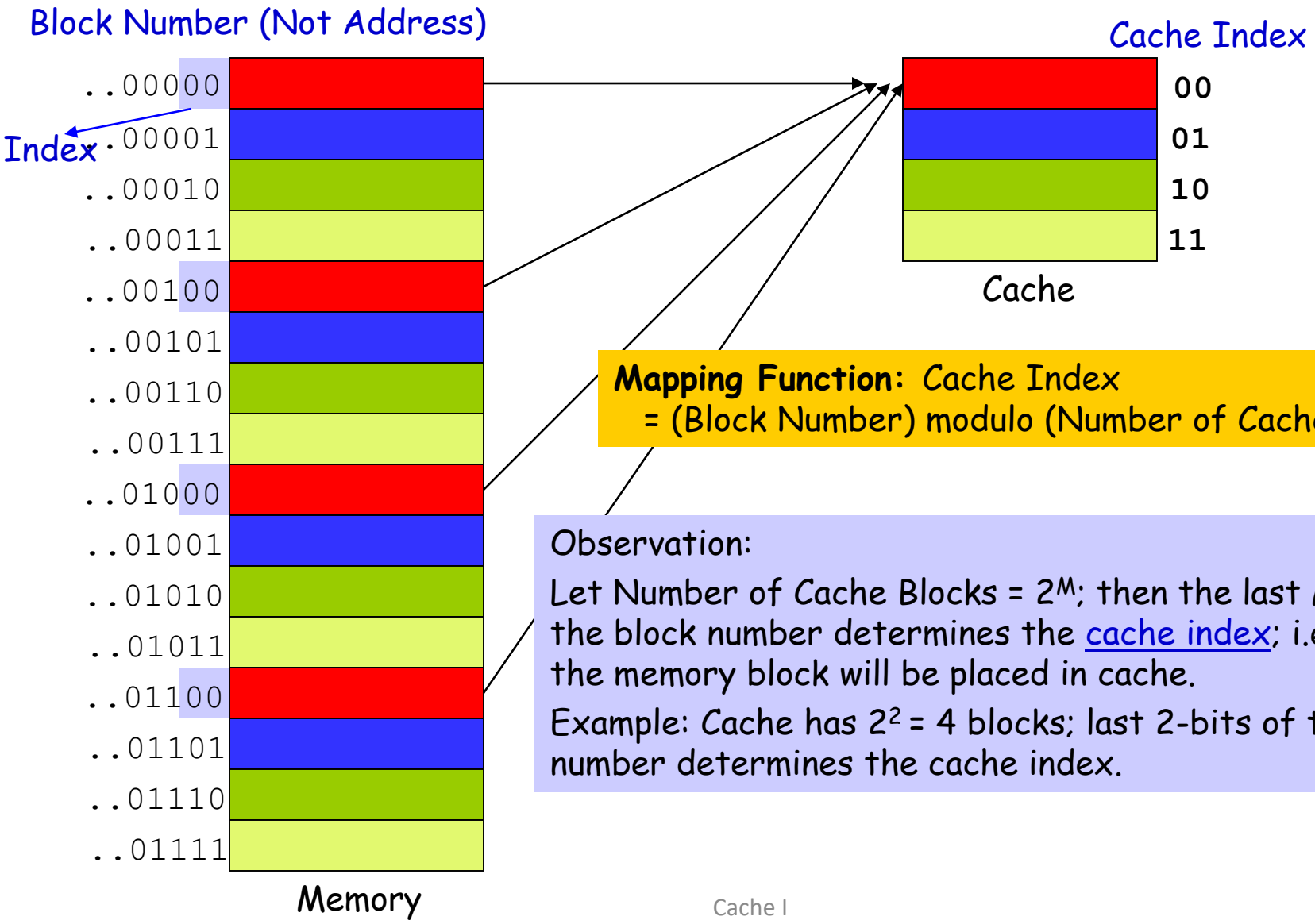
Hit rate × Hit Time + (1 – Hit rate) × Miss penalty

Suppose our on-chip SRAM (cache) has 0.8 ns access time, but the fastest DRAM we can get has an access time of 10ns. How high a hit rate do we need to sustain an average access time of 1ns?

$$1 = h \times 0.8 + (1-h) \times (10+0.8)$$
$$h = 98\%$$

Wow ! Can we really achieve this high hit rate with cache?

Direct-Mapped Cache (2/2)



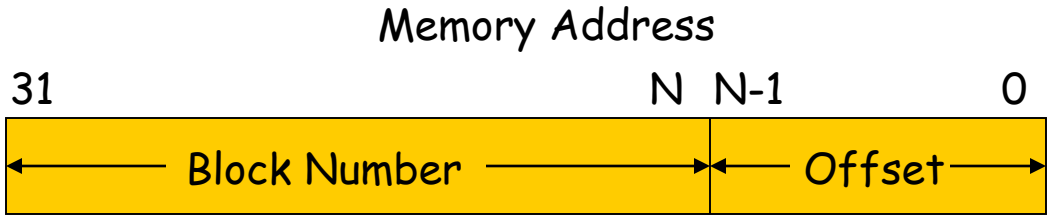
Mapping Function: Cache Index
= (Block Number) modulo (Number of Cache Blocks)

Observation:

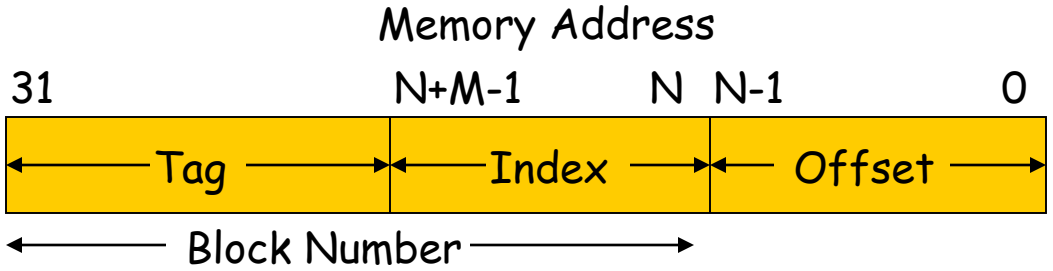
Let Number of Cache Blocks = 2^M; then the last M-bits of the block number determines the cache index; i.e., where the memory block will be placed in cache.

Example: Cache has $2^2 = 4$ blocks; last 2-bits of the block number determines the cache index.

Closer Look At Mapping

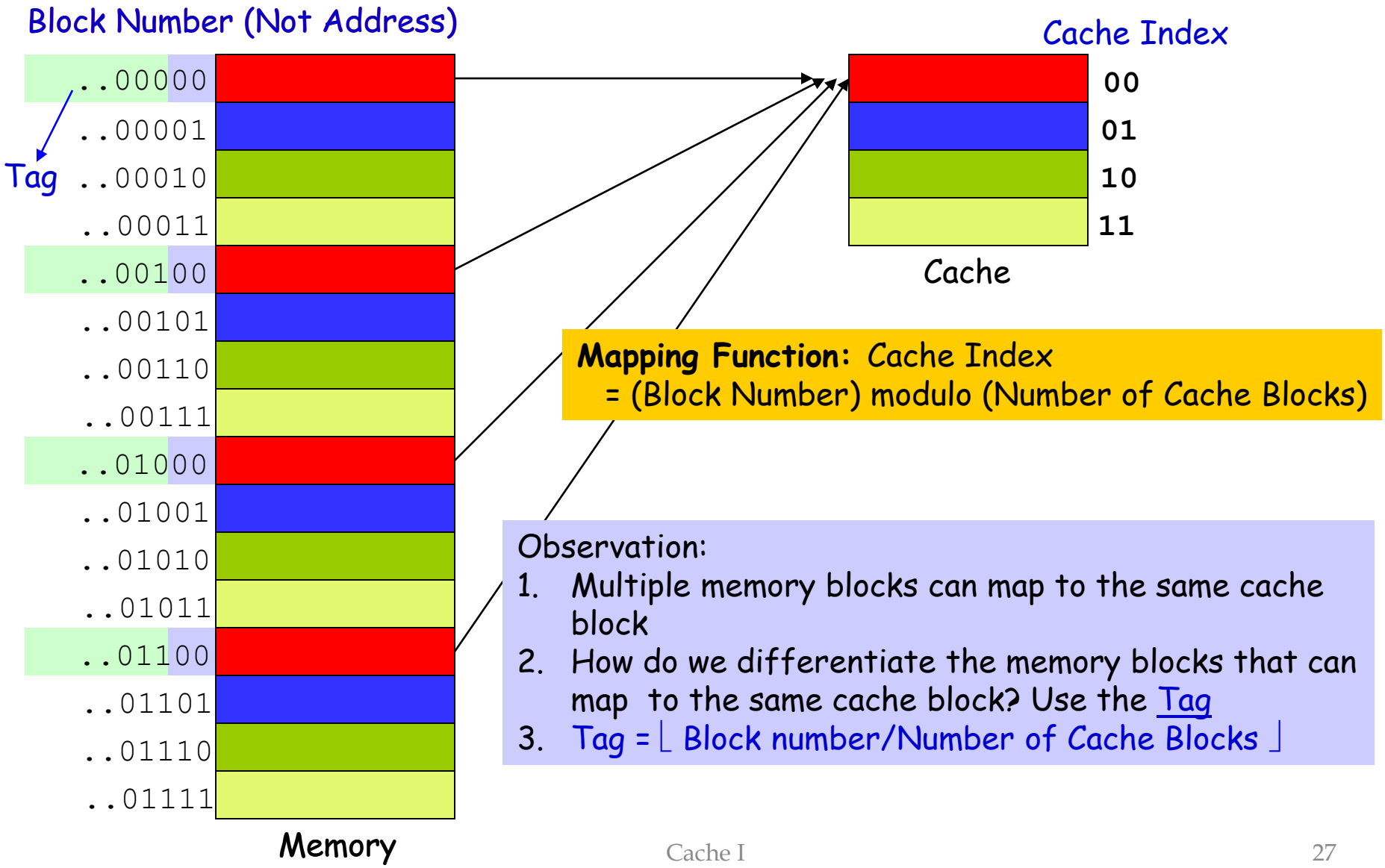


Block size = 2^N bytes

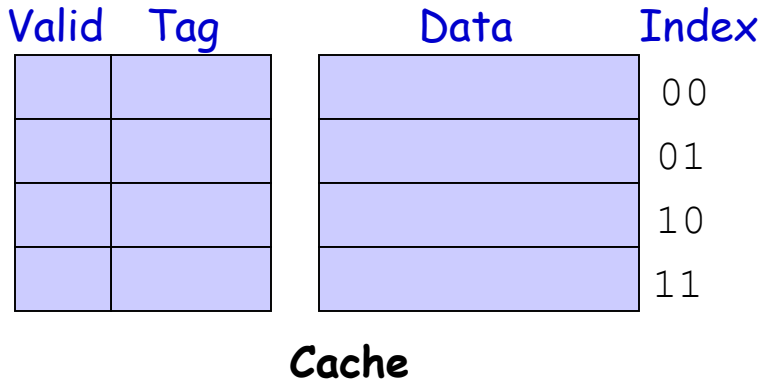


Block size = 2^N bytes
Number of cache blocks = 2^M
Offset: N bits
Index: M bits
Tag: 32 – (N + M)

Why Do We Need Tag?



Cache Structure



Cache

Along with a data block (line), cache contains

1. Tag of the memory block
2. Valid bit indicating whether the cache line contains valid data – why?

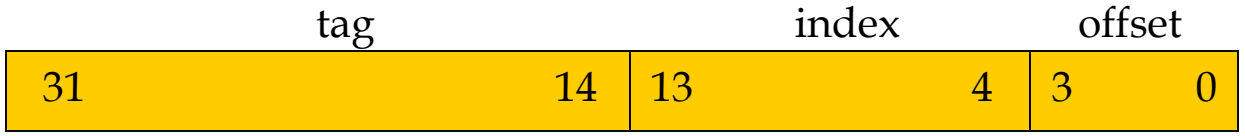
Cache hit: (Tag[index] == Tag[memory address]) AND (Valid[index] == TRUE)

Cache: Example (1/2)

- We have 4GB main memory and block size is 2^4=16 bytes (N = 4). How many memory blocks are there?
  - 4GB/16 bytes = 256M memory blocks
- How many bits are required to uniquely identify the memory blocks?
  - 256M =2^{28} memory blocks will require 28 bits
  - We can also calculate it as (32-N) = 28
- We have 16KB cache. How many cache blocks are there?
  - 16KB/16 bytes = 1024 cache blocks

Cache: Example (2/2)

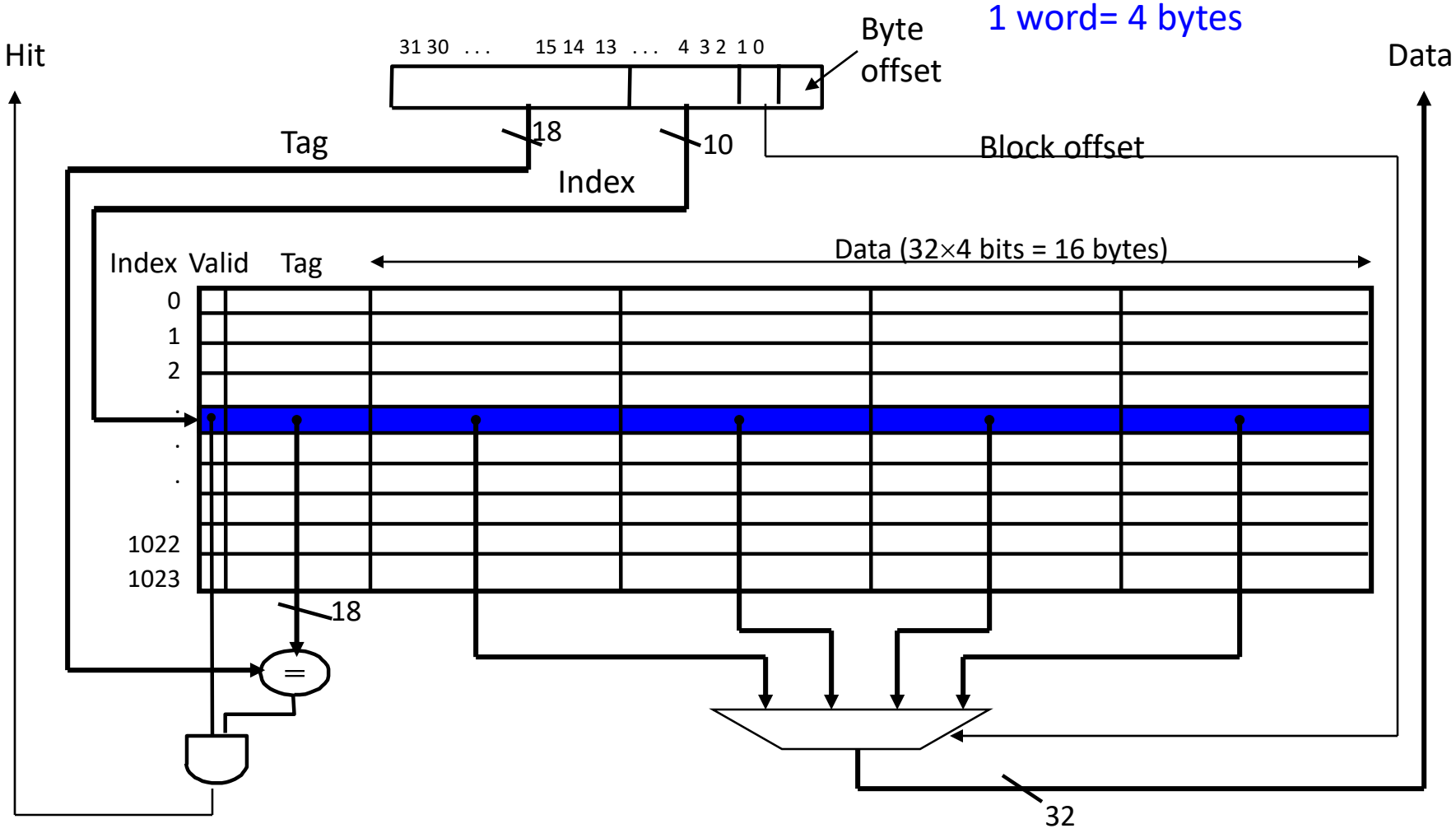
- How many bits do we need for cache index?
  - 1024 cache blocks ➔ 10-bit index (M=10)
- How many memory blocks can map to the same cache block?
  - 256M memory blocks/1K cache blocks = 256K
- How many Tag bits do we need?
  - 256K = 2^{18} memory blocks can be uniquely distinguished using 18 bits
  - We can also calculate it as 32 – (M+N) = 18



Cache Circuitry

A 16-KB cache using 4-word (16-byte) blocks.

1 word= 4 bytes



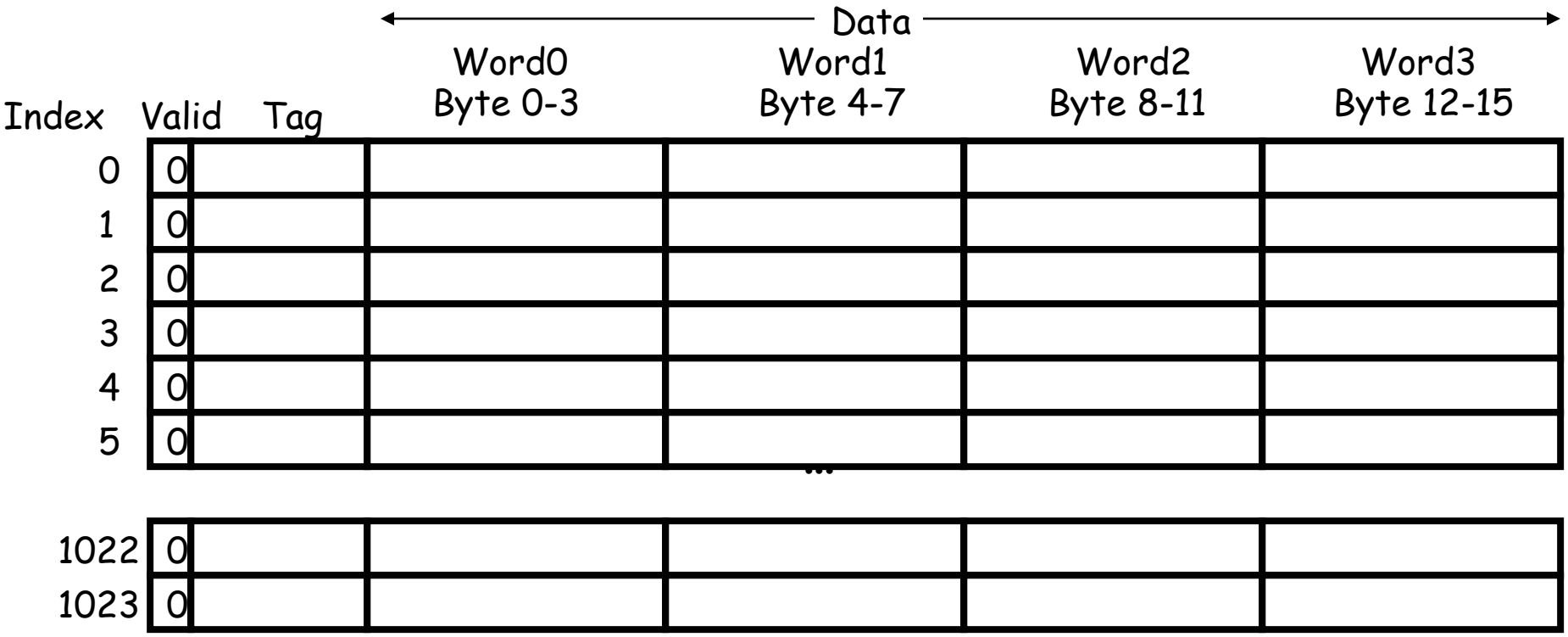
Accessing Data (1/17)

- Let's go through accessing some data in a direct mapped, 16KB cache:
  - 16-byte blocks x 1024 cache blocks.
  - Examples: 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields.

Tag	Index	Offset
00000000000000000	0000000001	0100
00000000000000000	0000000001	1100
00000000000000000	0000000011	0100
00000000000000010	0000000001	1000

Accessing Data (2/17)

- Initially cache is empty; all Valid bits are 0

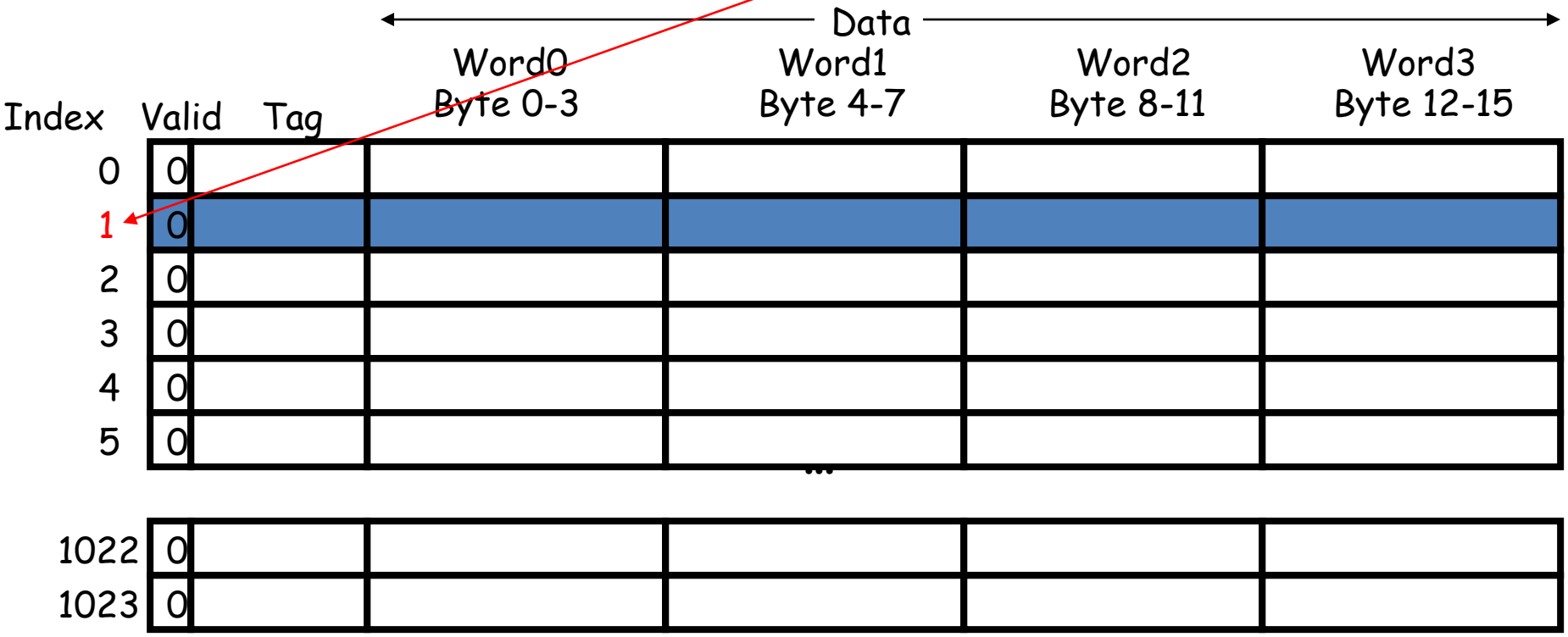


Accessing Data (3/17)

Load from Address: 00000000000000000 0000000001 0100

Tag Index Offset

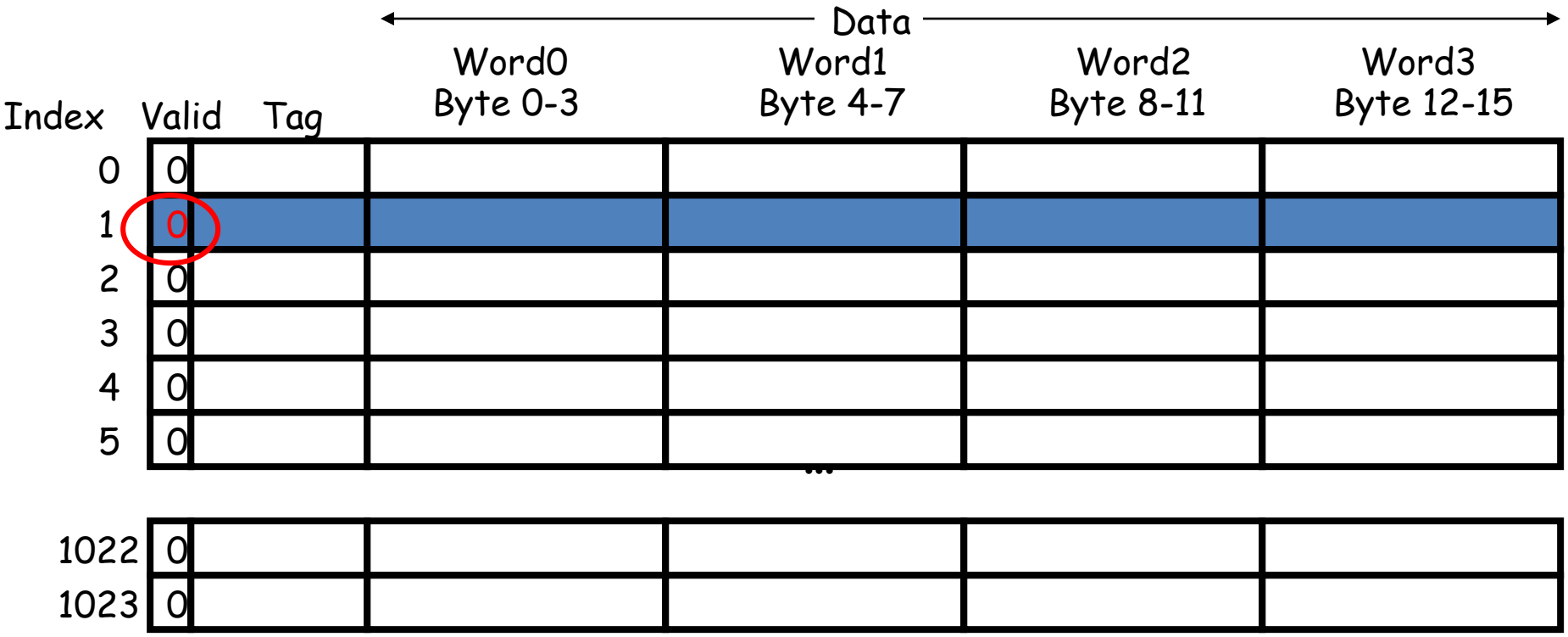
1. Look at cache block 1



Accessing Data (4/17)

	Tag	Index	Offset
Load from Address:	00000000000000000	0000000001	0100

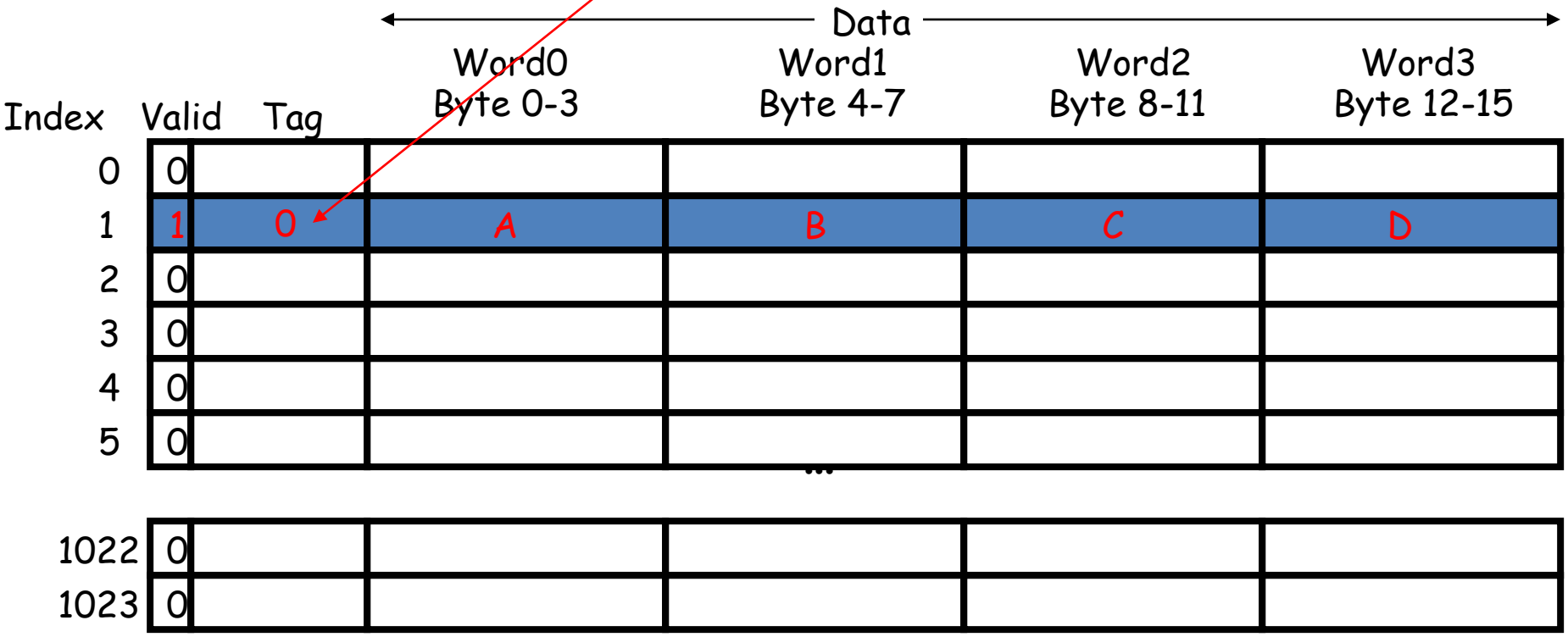
2. Data in block 1 is not valid → Cache Miss [Cold/Compulsory Miss]



Accessing Data (5/17)

	Tag	Index	Offset
Load from Address:	00000000000000000	0000000001	0100

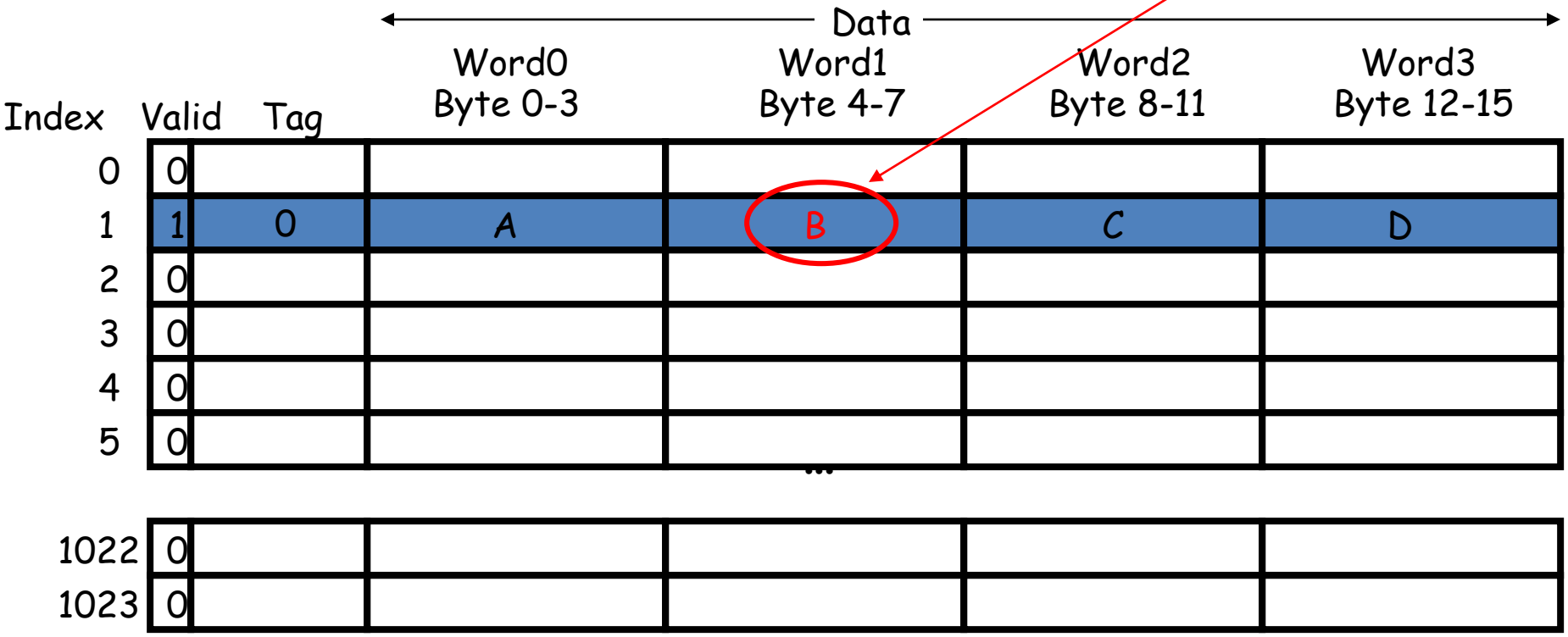
3. Load 16-byte data from memory to cache; set tag and valid bit



Accessing Data (6/17)

	Tag	Index	Offset
Load from Address:	00000000000000000	0000000001	0100

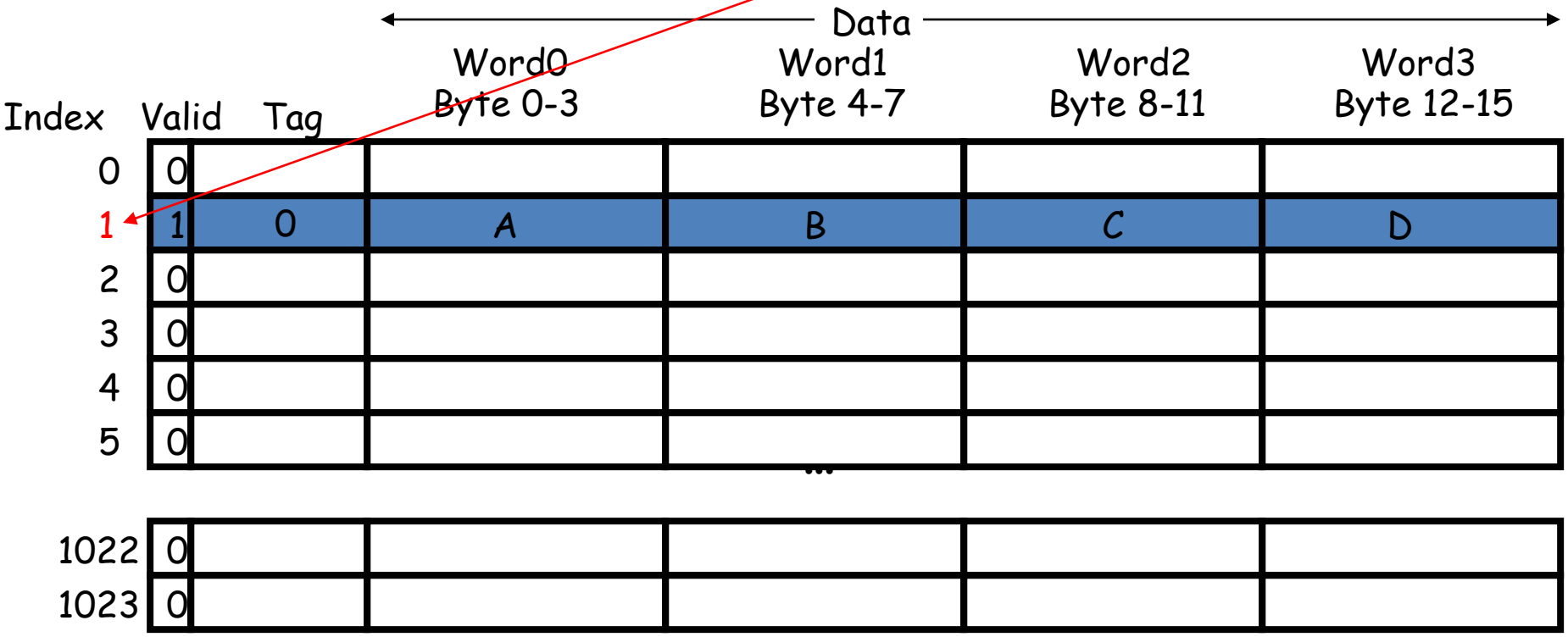
4. Load Word1 (byte offset = 4) from cache to register



Accessing Data (7/17)

	Tag	Index	Offset
Load from Address:	00000000000000000	0000000001	1100

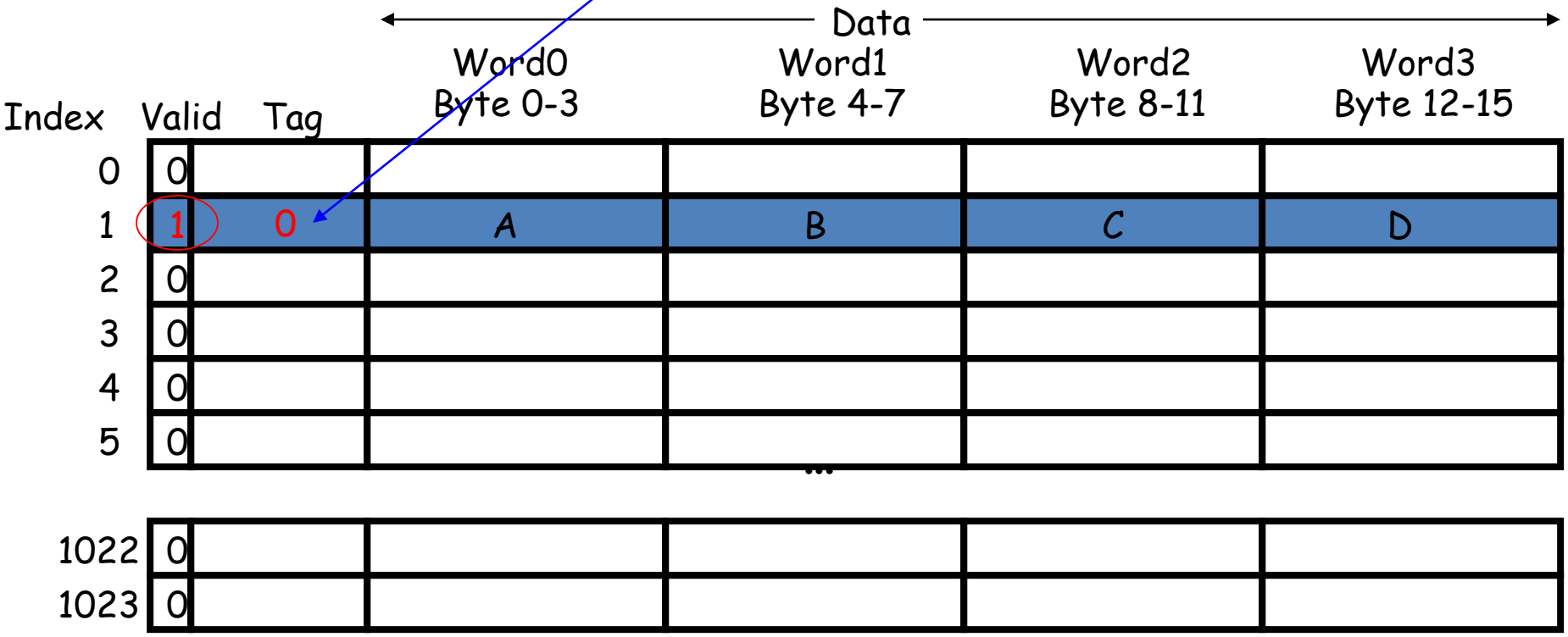
1. Look at cache block 1



Accessing Data (8/17)

	Tag	Index	Offset
Load from Address:	00000000000000000	0000000001	1100

2. Data in block 1 is valid + Tag Match → Cache Hit

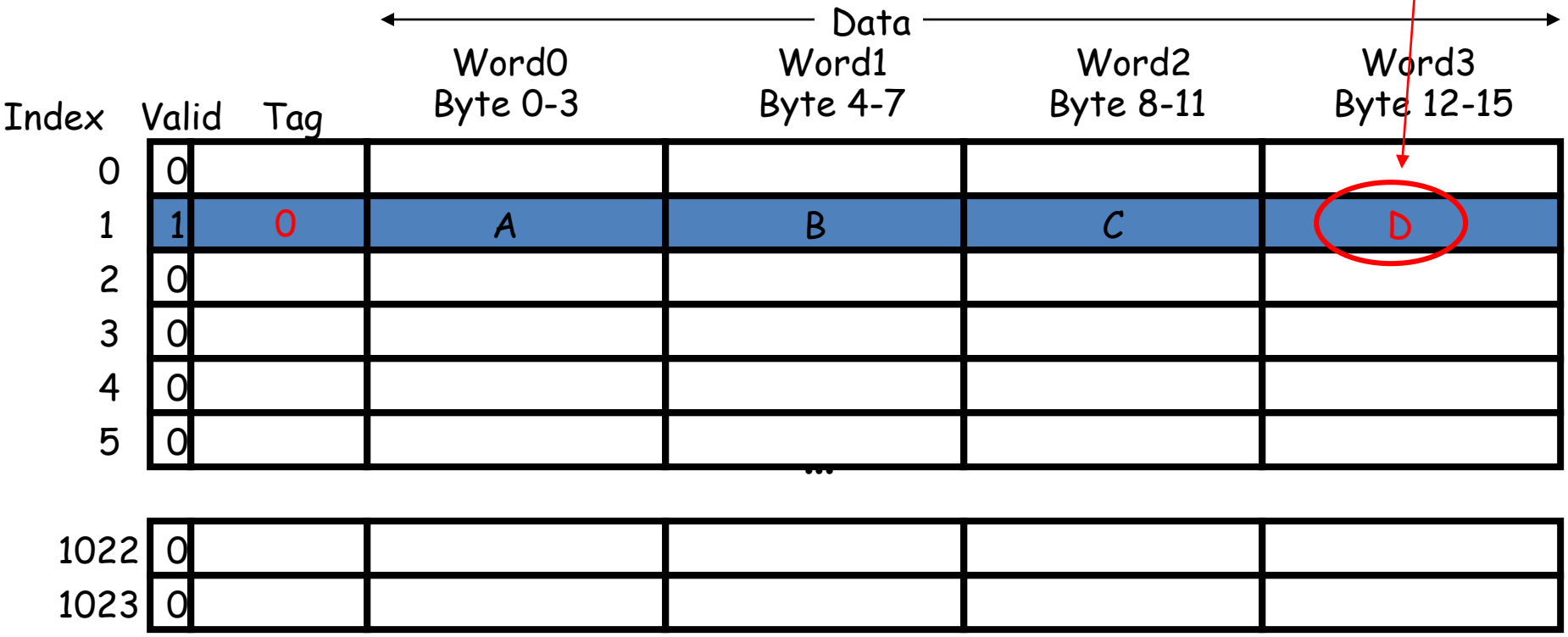


Accessing Data (9/17)

Load from Address: 0000000000000000 0000000001 1100

(Tag: 0000000000000000 | Index: 0000000001 | Offset: 1100)

3. Load Word3 (byte offset = 12) from cache to register [Spatial Locality]

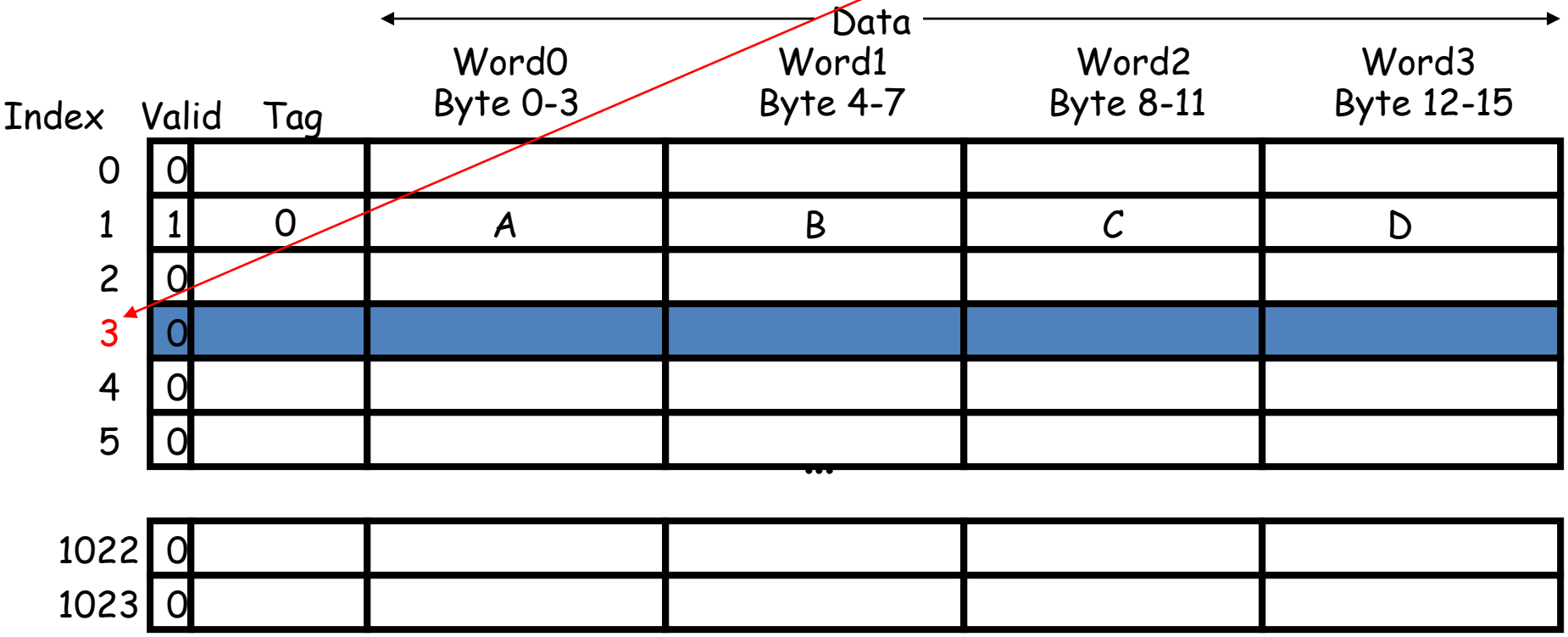


Accessing Data (10/17)

Tag Index Offset

Load from Address: 00000000000000000 0000000011 0100

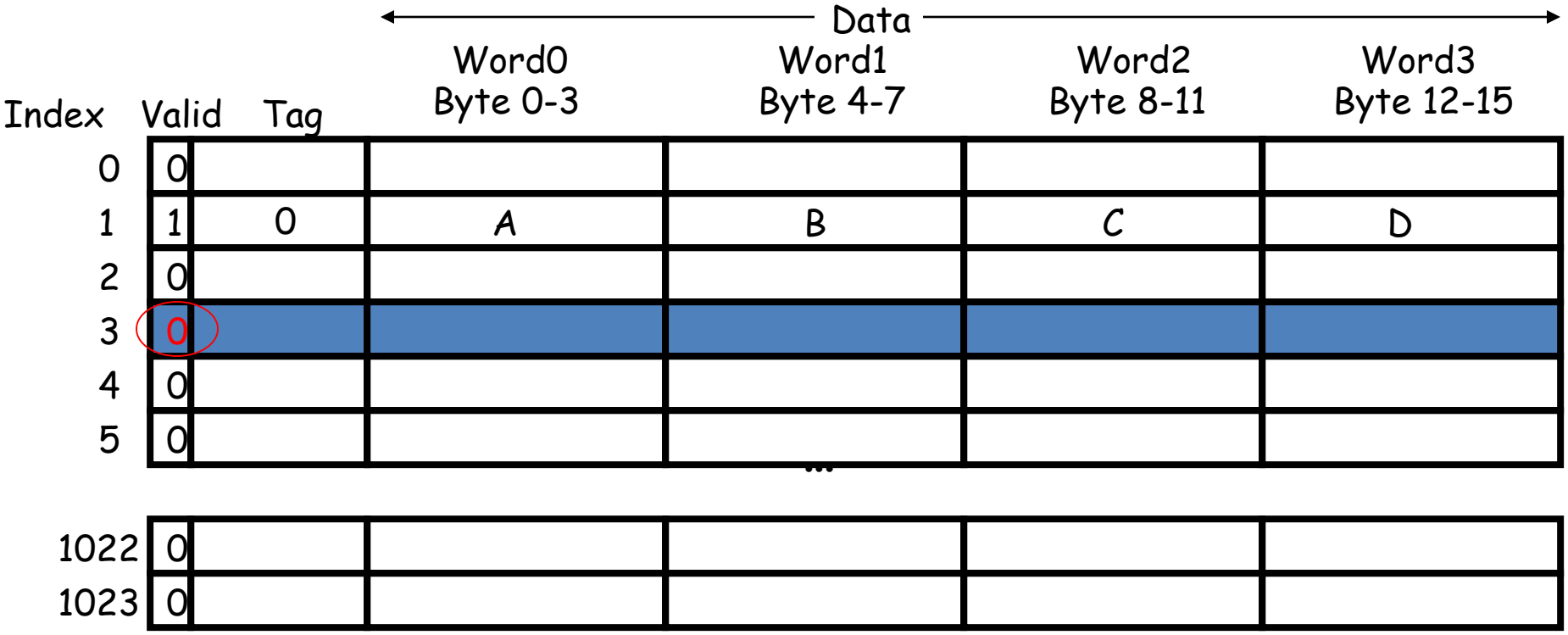
1. Look at cache block 3



Accessing Data (11/17)

	Tag	Index	Offset
Load from Address:	00000000000000000	0000000011	0100

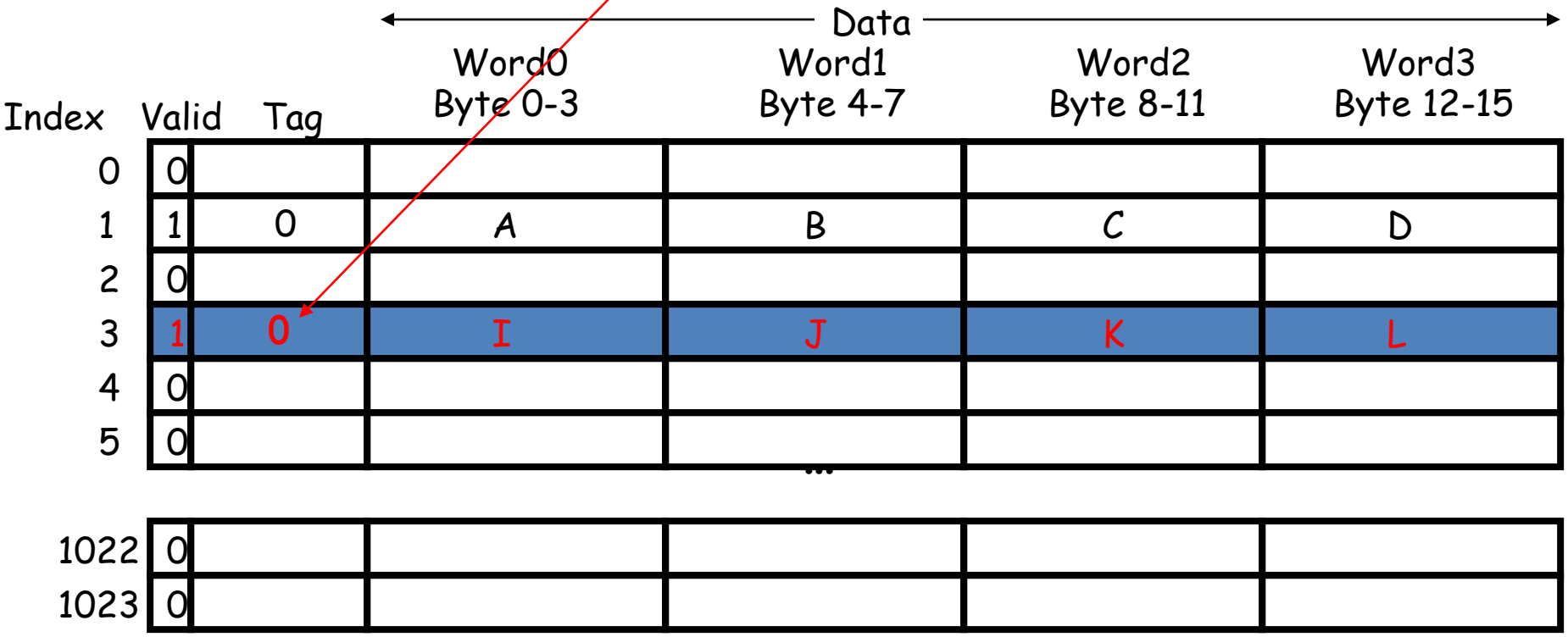
2. Data at cache block 3 in not valid → Cache Miss [Cold/Compulsory Miss]



Accessing Data (12/17)

	Tag	Index	Offset
Load from Address:	00000000000000000000	0000000011	0100

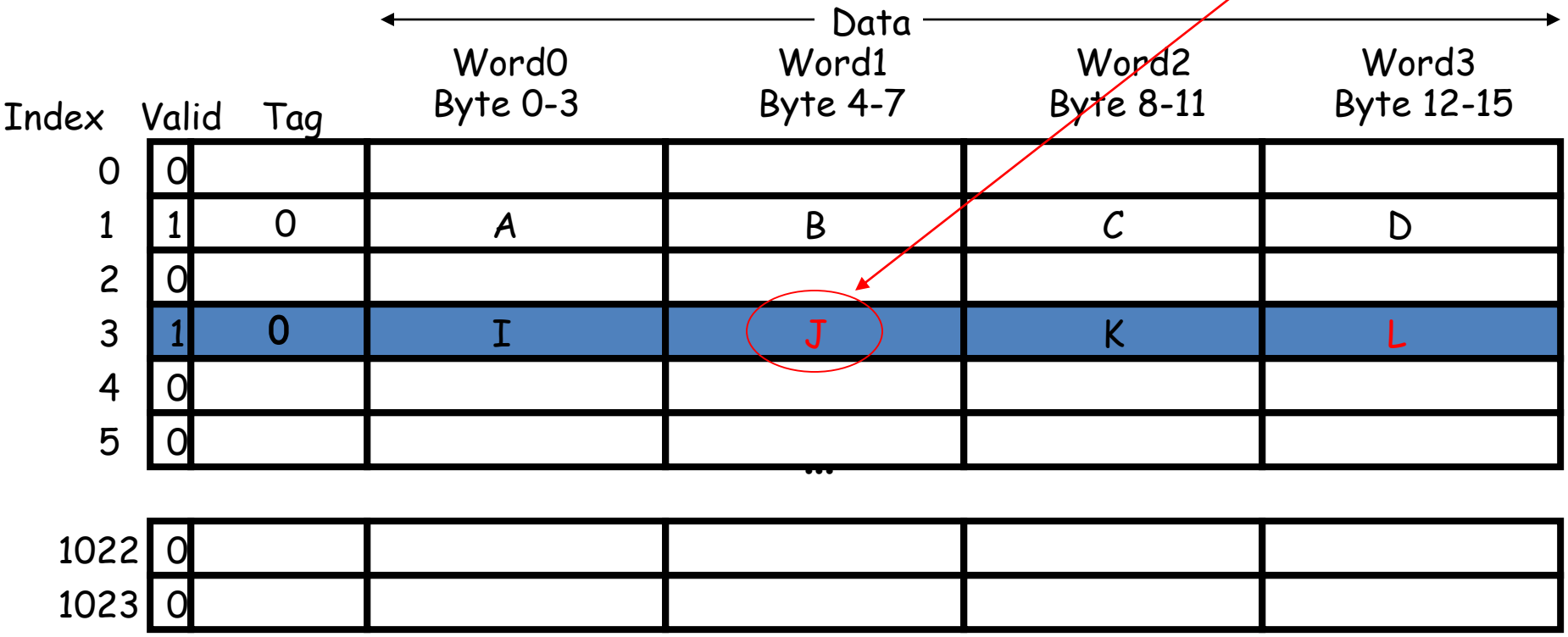
3. Load 16-byte data from main memory to cache; set tag and valid bit



Accessing Data (13/17)

	Tag	Index	Offset
Load from Address:	00000000000000000	0000000011	0100

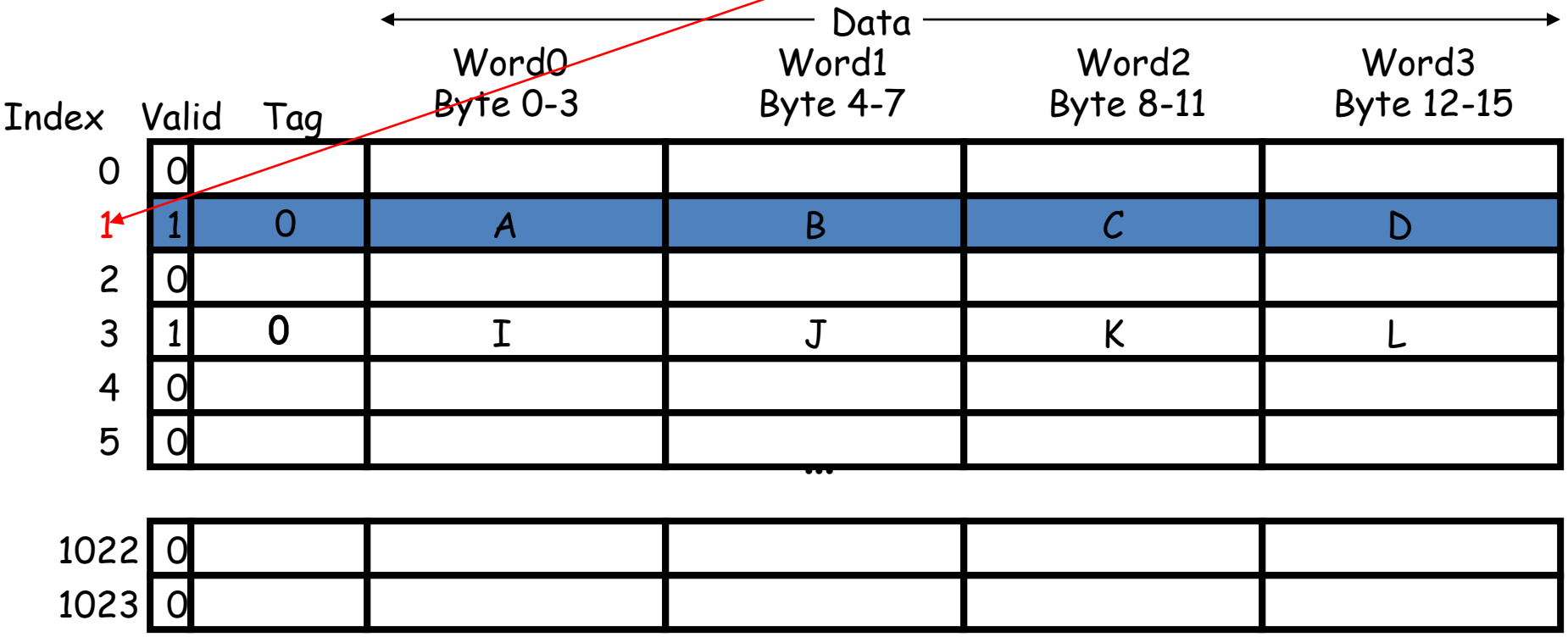
4. Return Word 1 (byte offset = 4) from cache to register



Accessing Data (14/17)

	Tag	Index	Offset
Load from Address:	0000000000000010	0000000001	0100

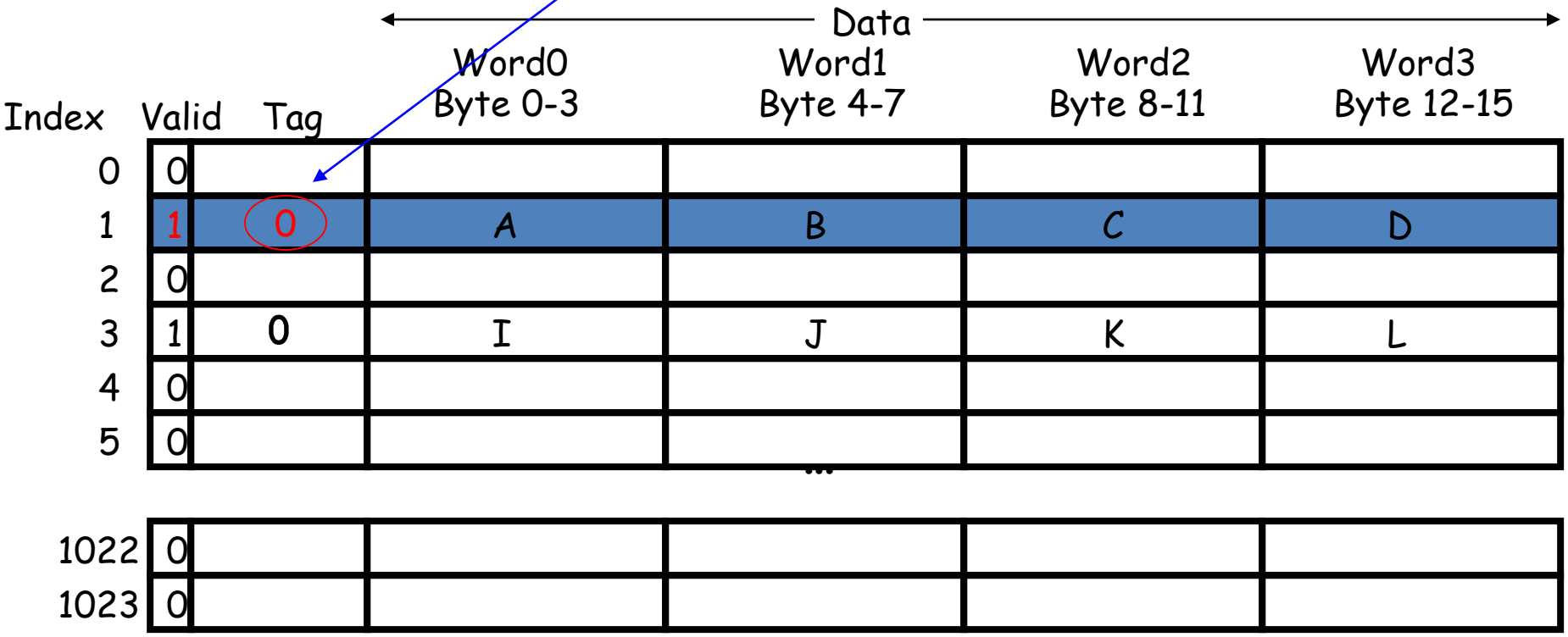
1. Look at cache block 1



Accessing Data (15/17)

	Tag	Index	Offset
Load from Address:	0000000000000010	0000000001	0100

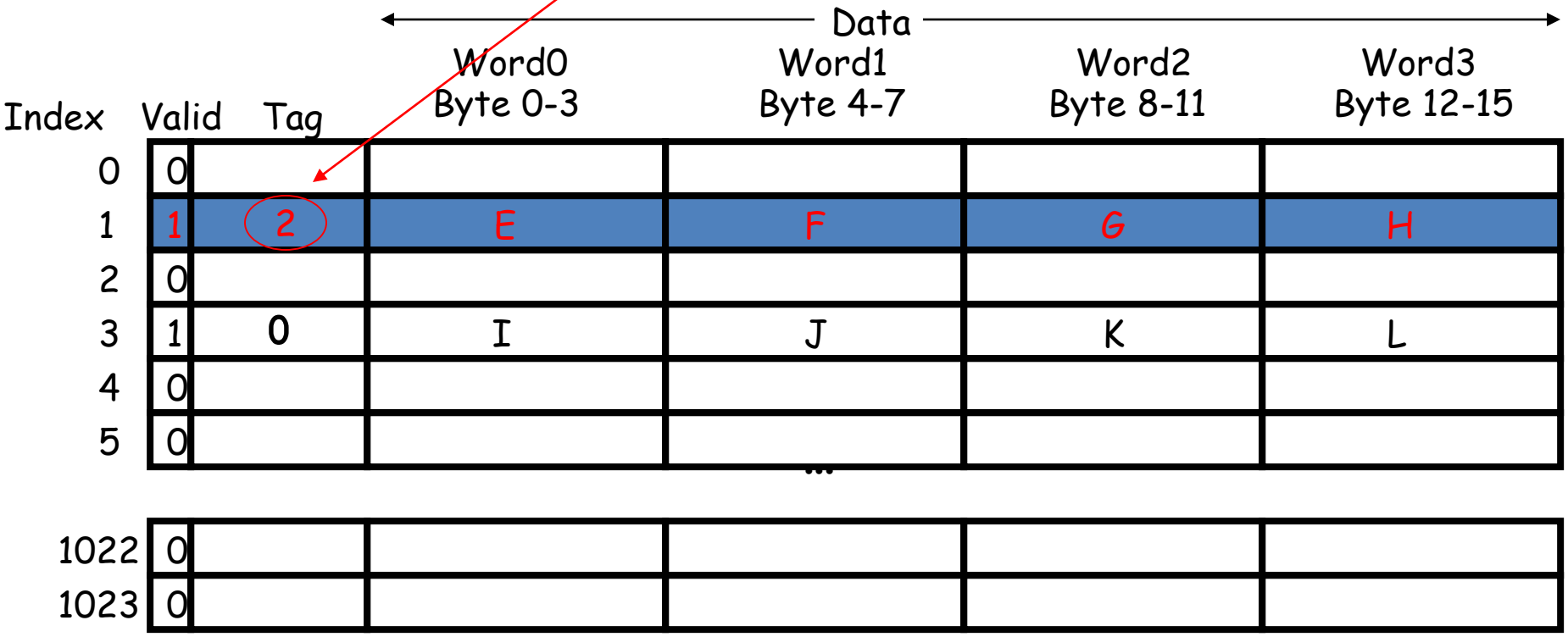
2. Data at cache block 1 is valid but tag mismatch → Cache Miss [Cold Miss]



Accessing Data (16/17)

	Tag	Index	Offset
Load from Address:	0000000000000010	0000000001	0100

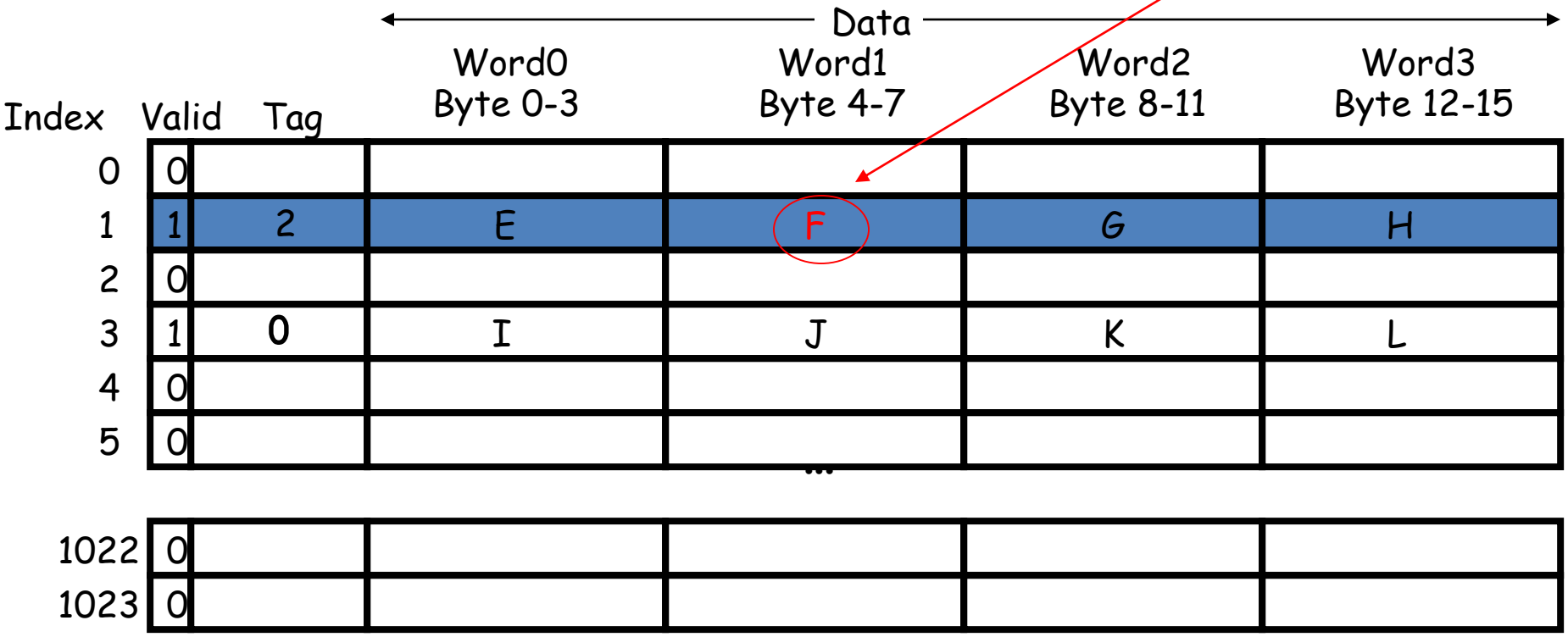
3. Replace cache block 1 with new data and tag



Accessing Data (17/17)

	Tag	Index	Offset
Load from Address:	0000000000000010	0000000001	0100

4. Load Word 1 (byte offset = 4) from cache to register



Try It Yourself #1 (1/2)

- We have 4GB main memory and block size is 2^3=8 bytes (N = 3). How many memory blocks are there?

- How many bits are required to uniquely identify the memory blocks?

- We have 32-byte cache. How many cache blocks (lines) are there?

Try It Yourself #1 (2/2)

- How many bits do we need for cache index?

- How many memory blocks can map to the same cache block (line)?

- How many Tag bits do we need?

Try It Yourself #2 (1/10)

- Here is a series of memory address references: 4, 0, 8, 12, 36, 0, 4 for a direct-mapped cache with four 8-byte blocks. Indicate hit/miss for each reference. Assume that 1 word contains 4 bytes.

Number of blocks in cache = 4 = 2^2
Index = ? bits
Size of block = 8 bytes = 2^3 bytes
Offset = ? bits
Memory size = unknown

Try It Yourself #2 (2/10)

Addresses: 4, 0, 8, 12, 36, 0, 4

Index	Valid	Tag	Word0	Word1
0	0			
1	0			
2	0			
3	0			

Try It Yourself #2 (3/10)

Memory

Address	Data
0	A
4	B
8	C
12	D
…	…
32	I
36	J
…	…

Try It Yourself #2 (4/10)

Addresses: 4, 0, 8, 12, 36, 0, 4

Address 4: …00000000000 00 100

Index	Valid	Tag	Word0	Word1
0				
1				
2				
3				

Try It Yourself #2 (5/10)

Addresses: 4, 0, 8, 12, 36, 0, 4

Address 0: …00000000000 00 000

Index	Valid	Tag	Word0	Word1
0				
1				
2				
3				

Try It Yourself #2 (6/10)

Addresses: 4, 0, 8, 12, 36, 0, 4

Address 8: …00000000000 01 000

Index	Valid	Tag	Word0	Word1
0				
1				
2				
3				

Try It Yourself #2 (7/10)

Addresses: 4, 0, 8, 12, 36, 0, 4

Address 12: …00000000000 01 100

Index	Valid	Tag	Word0	Word1
0				
1				
2				
3				

Try It Yourself #2 (8/10)

Addresses: 4, 0, 8, 12, 36, 0, 4

Address 36: …00000000001 00 100

Index	Valid	Tag	Word0	Word1
0				
1				
2				
3				

Try It Yourself #2 (9/10)

Addresses: 4, 0, 8, 12, 36, 0, 4

Address 0: …00000000000 00 000

Index	Valid	Tag	Word0	Word1
0				
1				
2				
3				

Try It Yourself #2 (10/10)

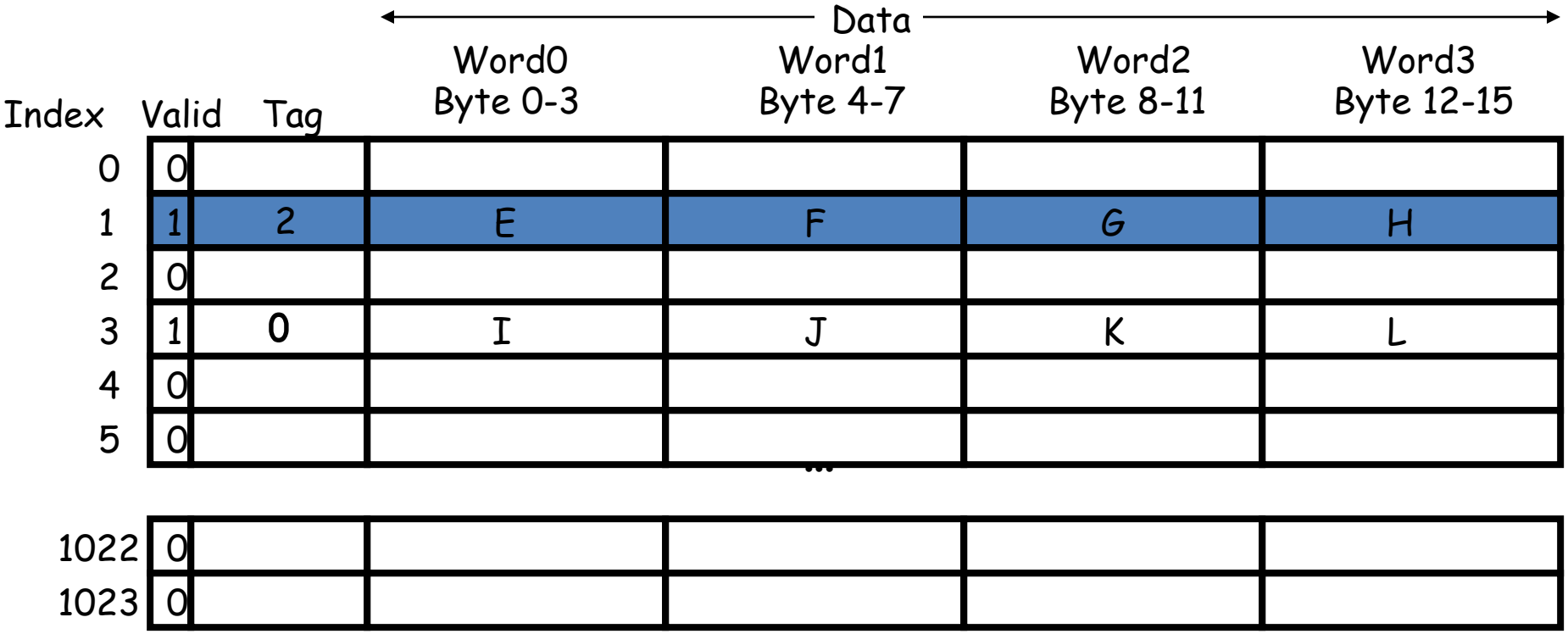
Addresses: 4, 0, 8, 12, 36, 0, 4

Address 4: …00000000000 00 100

Index	Valid	Tag	Word0	Word1
0				
1				
2				
3				

Write To Memory? (1/2)

	Tag	Index	Offset
Store X at Address:	0000000000000010	0000000001	0100



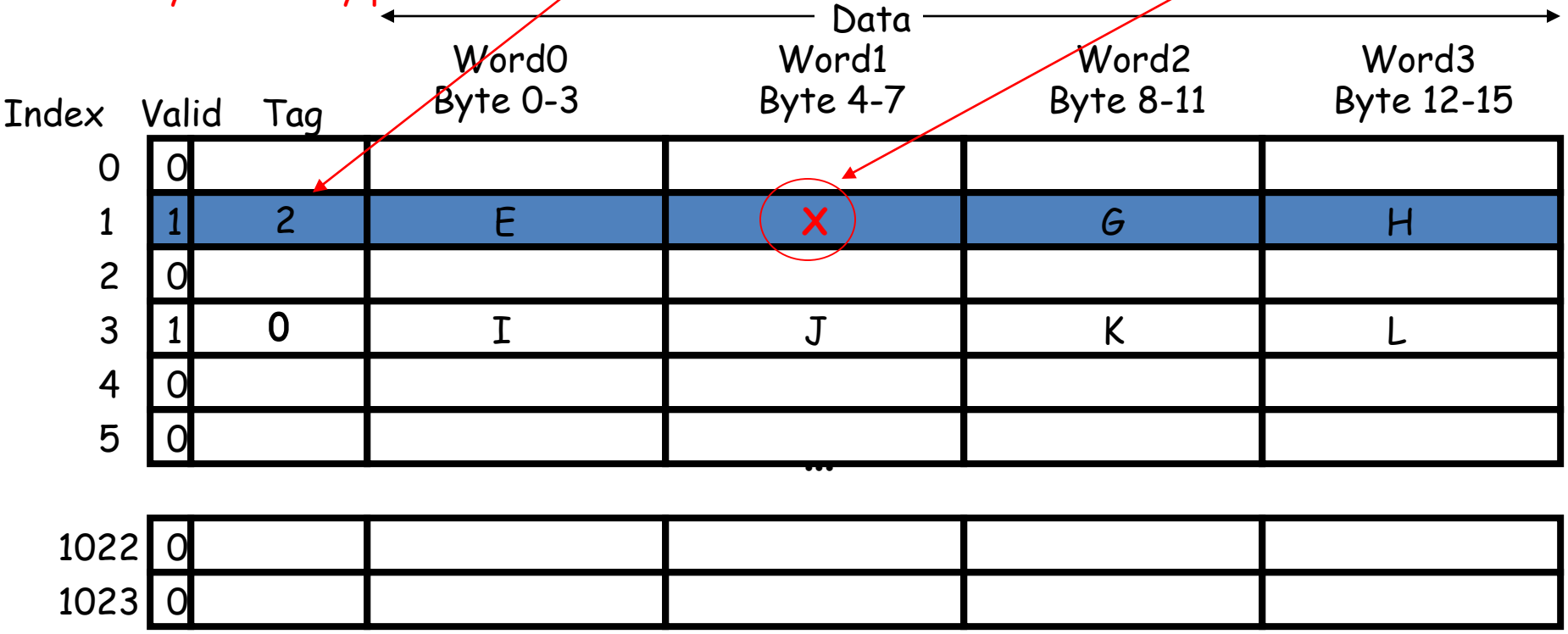
Write To Memory? (2/2)

Store X at Address: 0000000000000010 0000000001 0100

(Tag) (Index) (Offset)

1. Valid bit set + Tag match at cache block 1
2. Replace F with X in Word1 (byte offset = 4)

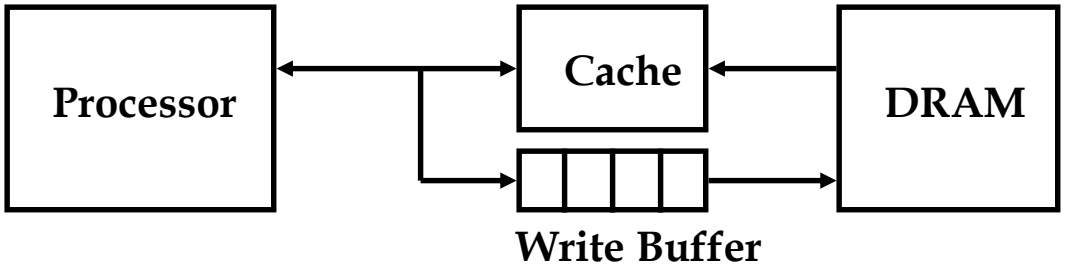
Do you see any problem here?



Write Policy

- Cache and main memory are inconsistent
  - Data is modified in cache; but not in main memory
  - So what to do on a data-write hit?
- Solution 1: Write-through cache
  - Write data both to cache and to main memory
  - Problem: Write will operate at the speed of main memory → very very slow
  - Solution: Put a write buffer between cache and main memory
- Solution 2: Write-back cache
  - Only write to cache; delay the write to main memory as much as possible

Write Buffer for Write Through



- Write Buffer between Cache and Memory
  - Processor: writes data to cache + write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle; otherwise overflow

Write-Back Cache

- Add an additional bit (Dirty bit) to each cache block
- Write data to cache and set Dirty bit of that cache block to 1
  - Do not write to main memory
- When you replace a cache block, check its Dirty bit; If Dirty = 1, then write that cache block to main memory

Cache write only on replacement

Cache Miss on Write

- On a read miss, you need to bring data to cache and then load from there to register – no choice here
- What happens on a write miss?
- Option 1: Write allocate
  - Load the complete block (16 bytes in our example) from main memory to cache
  - Change only the required word in cache
  - Write to main memory depends on write policy
- Option 2: Write no-allocate
  - Do not load the block to cache
  - Write the required word in main memory only
- Which one is best for write-back cache?

SUMMARY

- Memory hierarchy gives the illusion for fast but big memory
- Hardware-managed cache is an integral component of today's processors

READING ASSIGNMENT

- Large and Fast: Exploiting Memory Hierarchy
  - Chapter 7 sections 7.1 – 7.2 (3rd edition)
  - Chapter 5 sections 5.1 – 5.4 (4th , 5th edition)