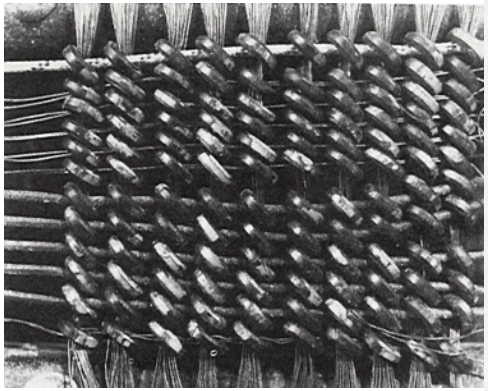
Cache Basics

Dr. Trevor E. Carlson
School of Computing
National University of Singapore
(Slides from Tulika Mitra)

Memory Technology: 1950s

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



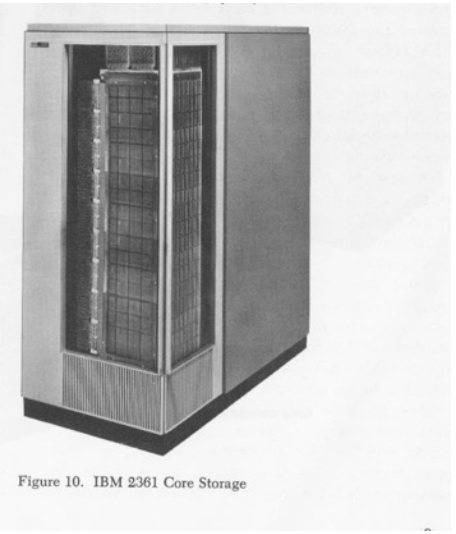


Figure 10. IBM 2361 Core Storage

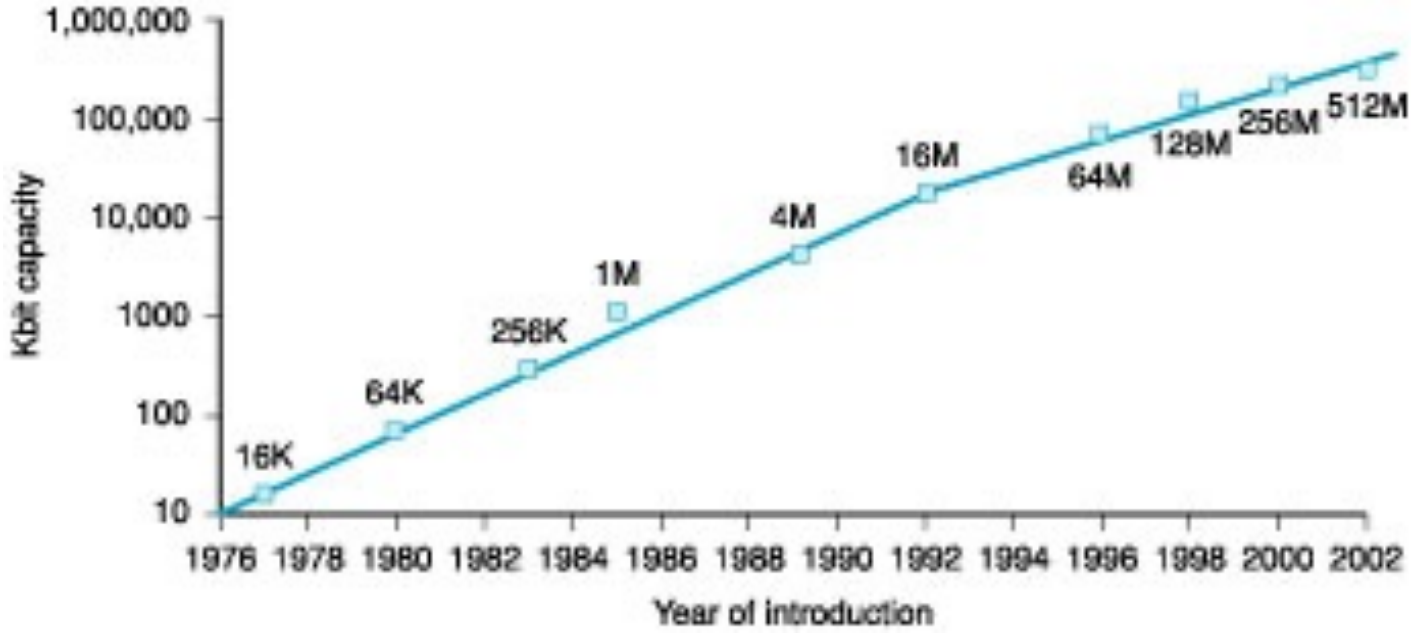
1952: IBM 2361 16KB magnetic core memory

Maurice Wilkes: 2005

Memory Technology today: DRAM

- Infineon Technologies: stores data equivalent to 640 books, 32,000 standard newspaper pages, and 1,600 still pictures or 64 hours of sound

DRAM Capacity Growth

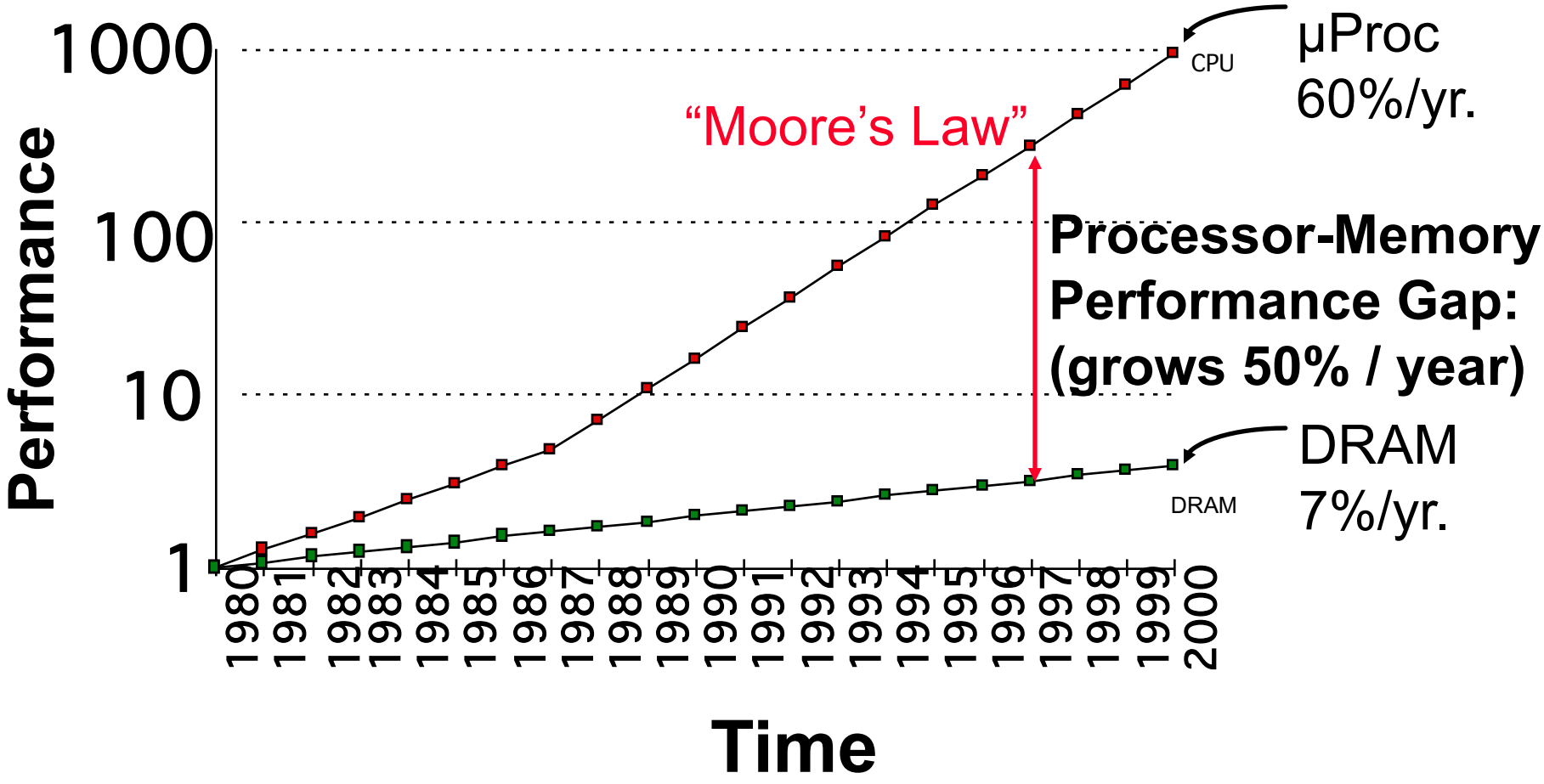


- 4X capacity increase almost every 3 years, i.e., 60% increase per year for 20 years
- Unprecedented growth in density, but we still have a problem

Processor-DRAM Performance Gap

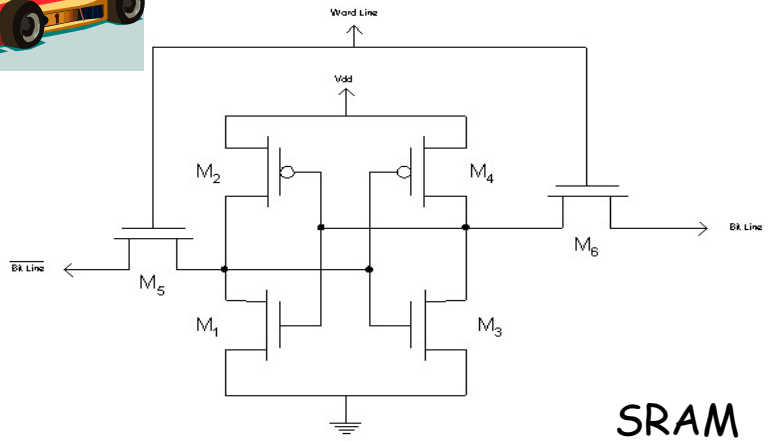
Memory Wall:

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

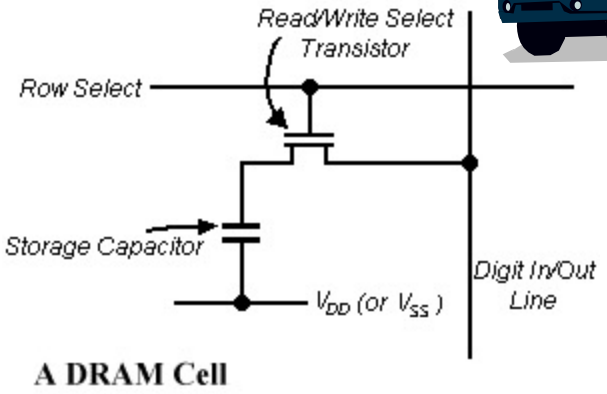


Do we have any faster memory technology?

Fast Memory Technologies: SRAM



SRAM



A DRAM Cell

- SRAM

6 transistor per memory cell → Low density

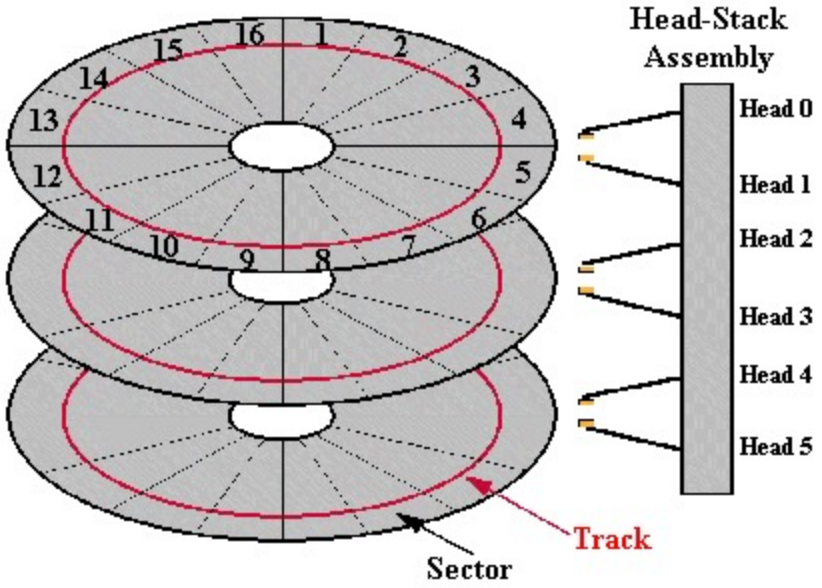
Fast access latency of 0.5 – 5 ns

- DRAM

1 transistor per memory cell → High density

Slow access latency of 50-70ns

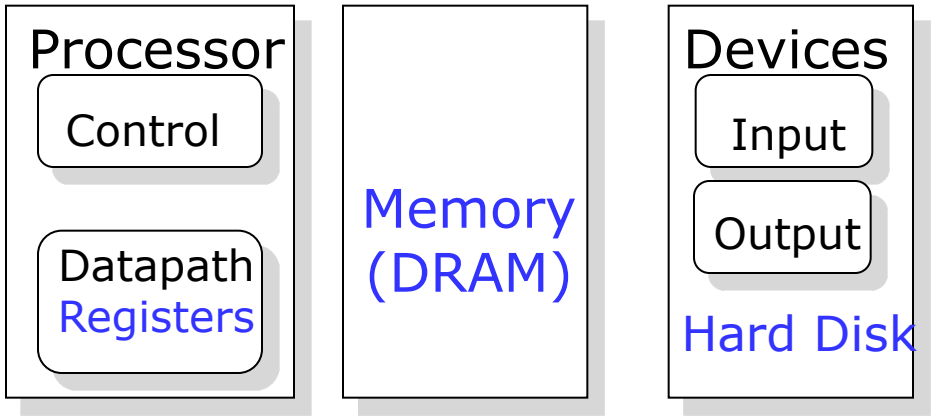
Slow Memory Technologies: Magnetic Disk



Typical high-end hard disk:

Average latency: 5-20 ms
Capacity: 250GB

Quality vs. Quantity

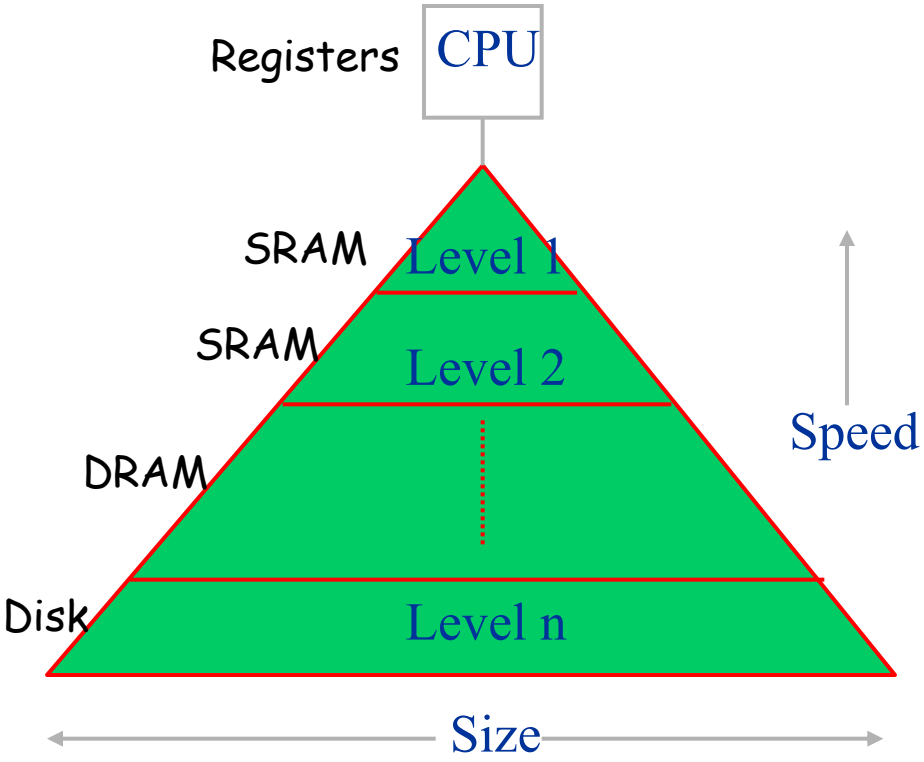


	Capacity	Latency	Cost/GB
Register	100s Bytes	20 ps	$$$$
SRAM	100s KB	0.5-5 ns	$$$
DRAM	100s MB	50-70 ns	$
Hard Disk	100s GB	5-20 ms	Cents
Ideal	1 GB	1 ns	Cheap

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

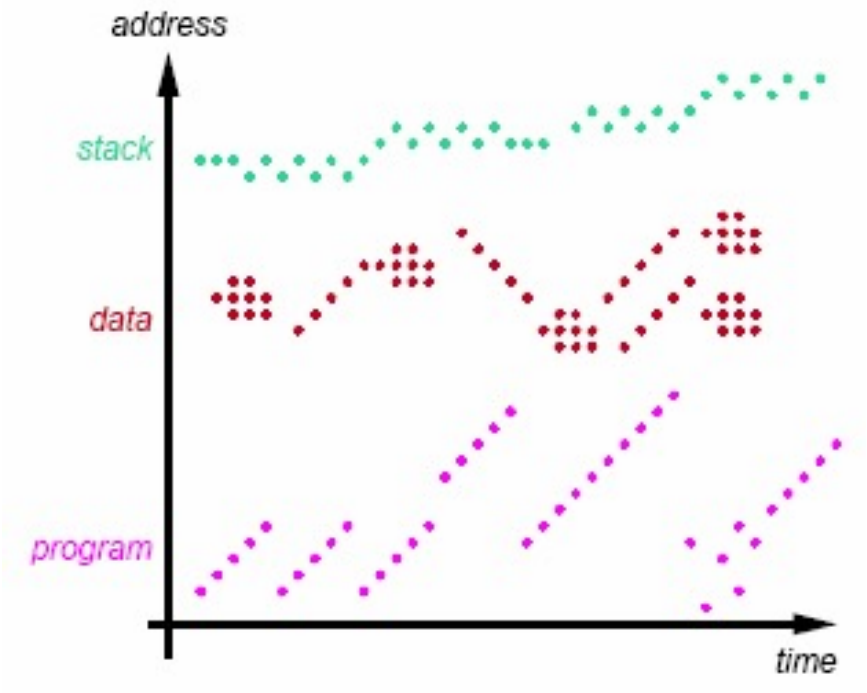


The Basic Idea

- 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

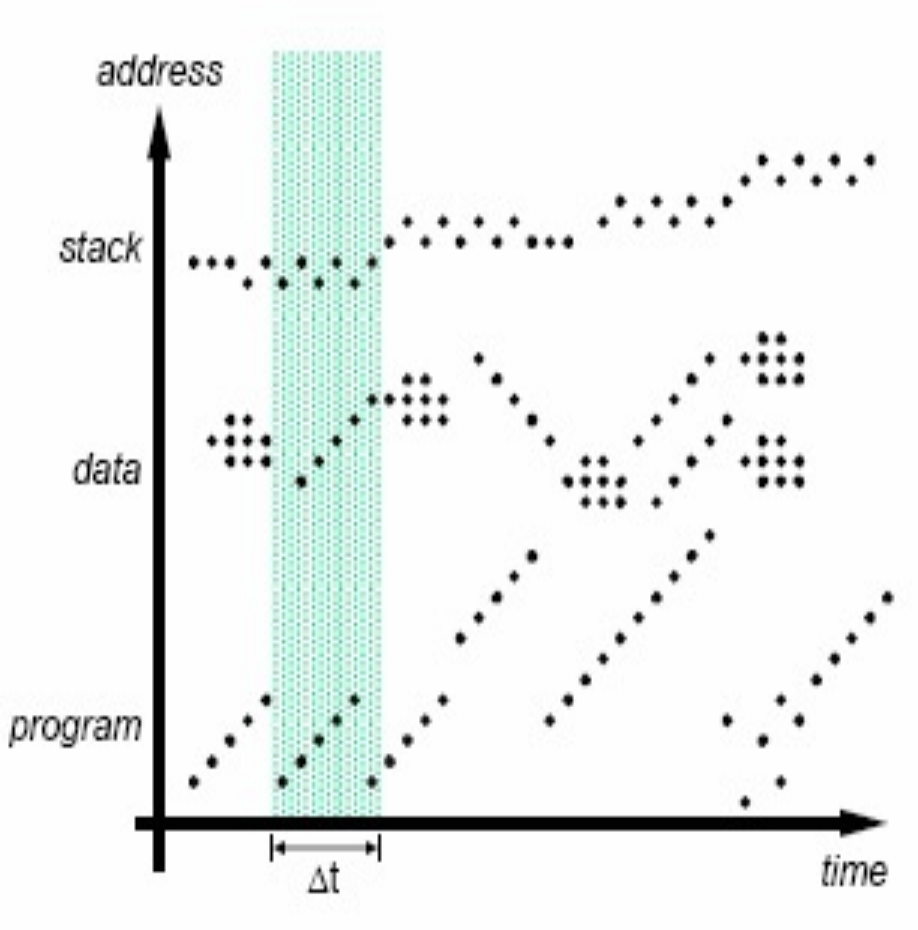
Program accesses only a small portion of the memory address space within a small time interval

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

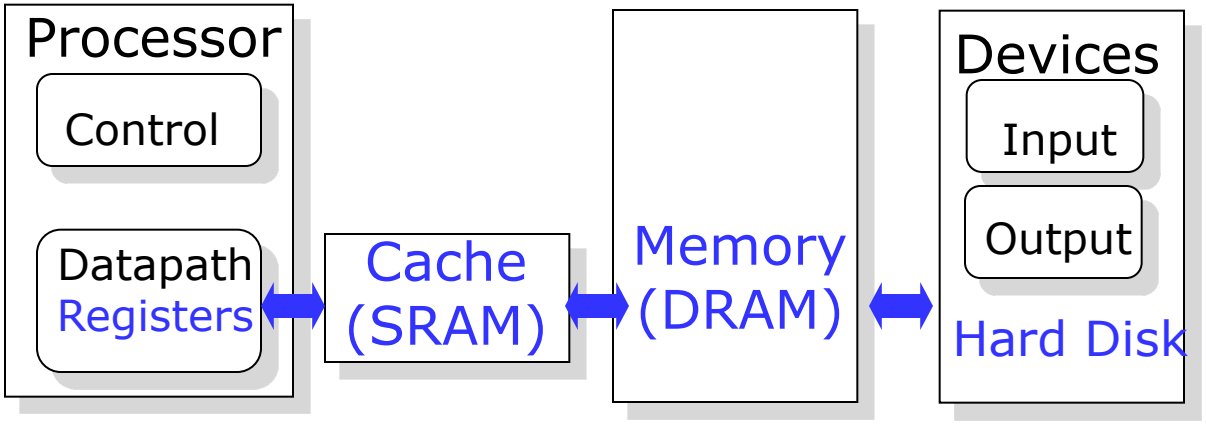


- Set of locations accessed during Δt
- Different phases of execution may use different working sets
- Want to capture the working set and keep it in the memory closest to CPU

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)

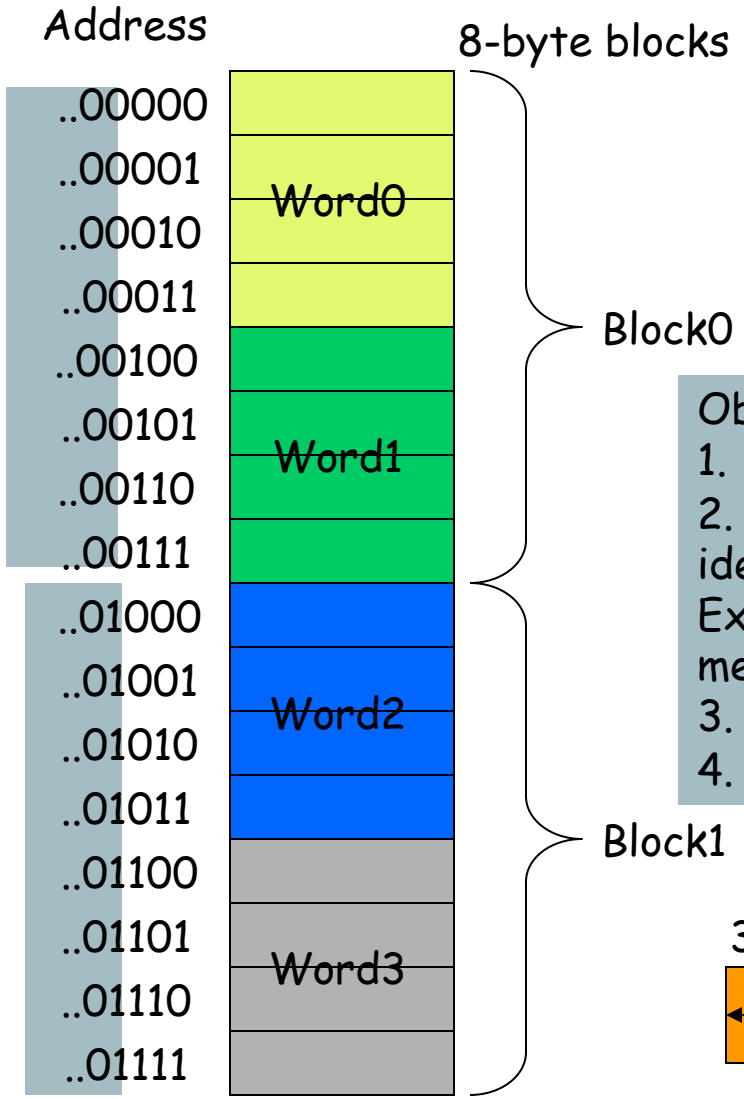


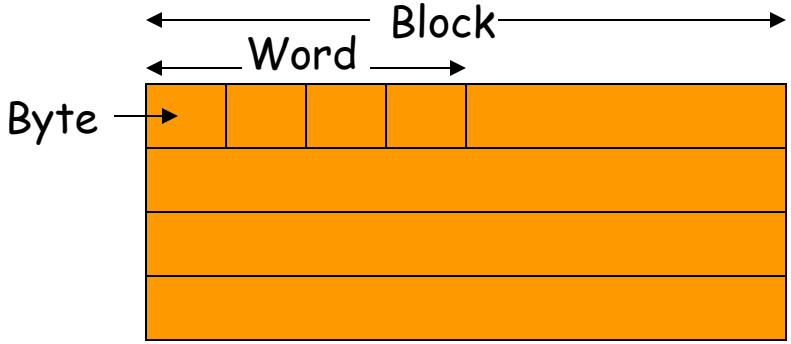
- How to make SLOW main memory appear faster? Cache – a small but fast SRAM near CPU
  - Often in the same chip as CPU
  - Introduced in 1960; almost every processor uses it today
  - Hardware managed: Transparent to programmer
- How to make SMALL main memory appear bigger than it is? Virtual memory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

Cache Block/Line (1/2)

- Unit of transfer between memory and cache
- 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?

Cache Block/Line (2/2)



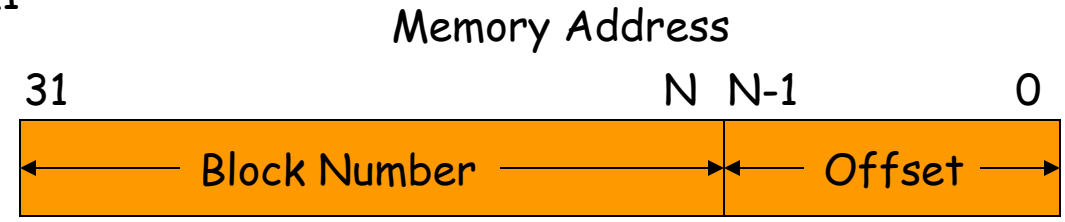


Observations:

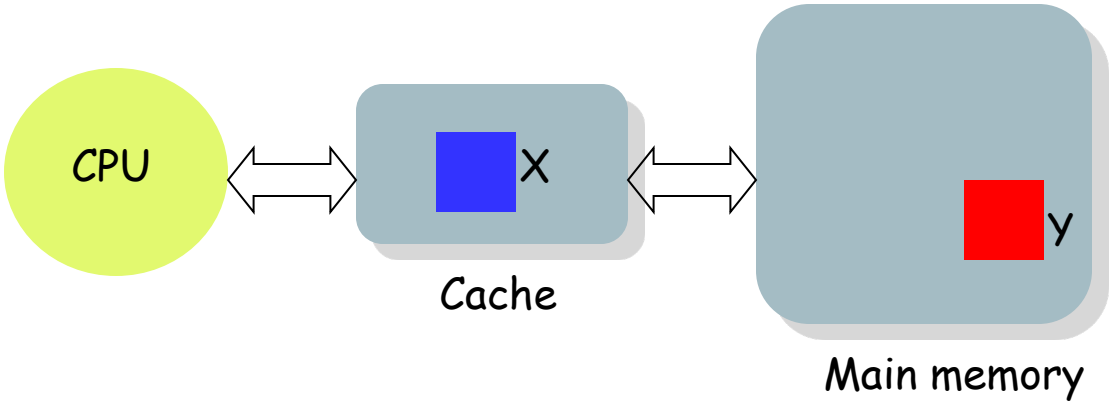
1. 2^N-byte blocks are aligned at 2^N-byte boundaries
2. The addresses of words within a 2^N-byte block have identical (32-N) most significant bits (MSB)

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



Cache Terminology



- Hit: Block appears in cache (e.g., X)
  - Hit rate: Fraction of memory accesses that hit
  - Hit time: Time to access cache
- Miss: Block is not in cache (e.g., Y)
  - Miss rate = 1 – Hit rate
  - Miss penalty: Time to replace cache block + deliver data
- Hit time < Miss penalty

Memory Access Time

Average access time =

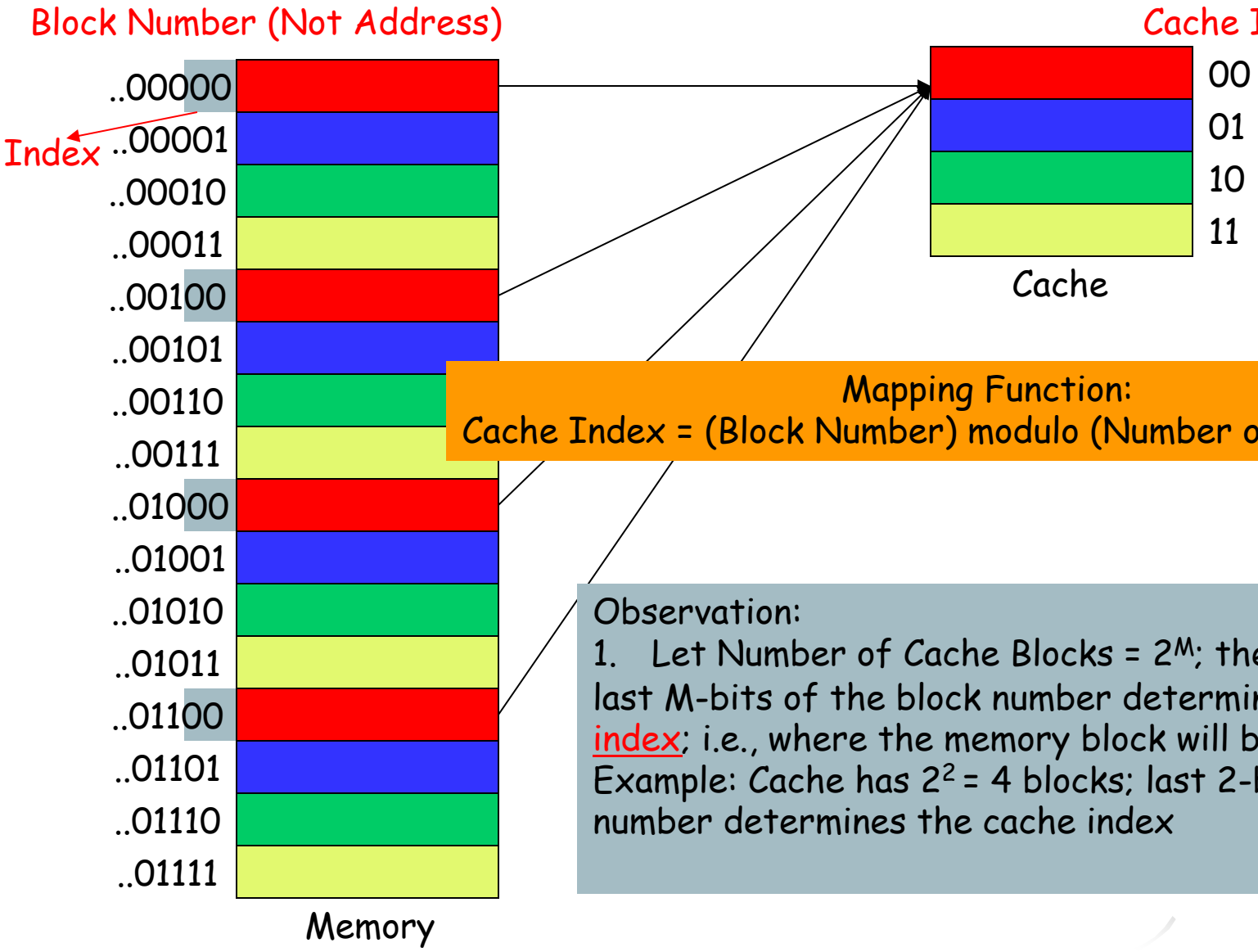
Hit rate x Hit Time + (1-Hit rate) x Miss penalty

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?

$$1 = h \times 0.8 + (1-h) \times 10$$
$$h = 97.8\%$$

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

Direct-Mapped Cache



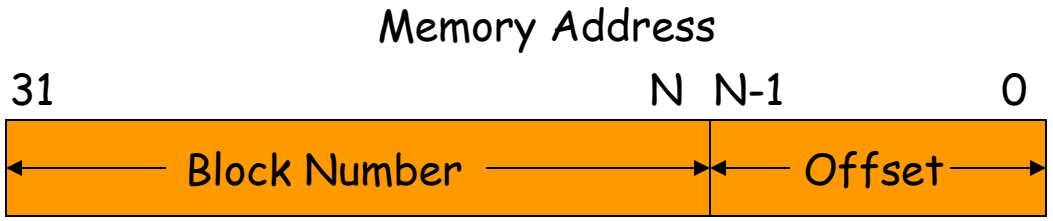
Mapping Function:
Cache Index = (Block Number) modulo (Number of Cache Blocks)

Observation:

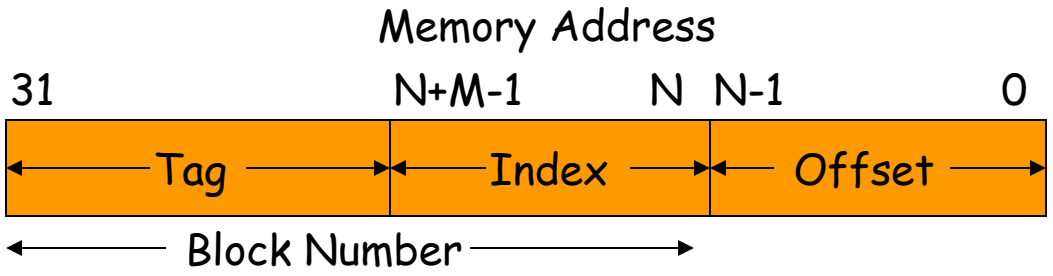
1. Let Number of Cache Blocks = 2^M; then the last M-bits of the block number determines the cache index; i.e., where the memory block will be placed in cache

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

Closer look at mapping

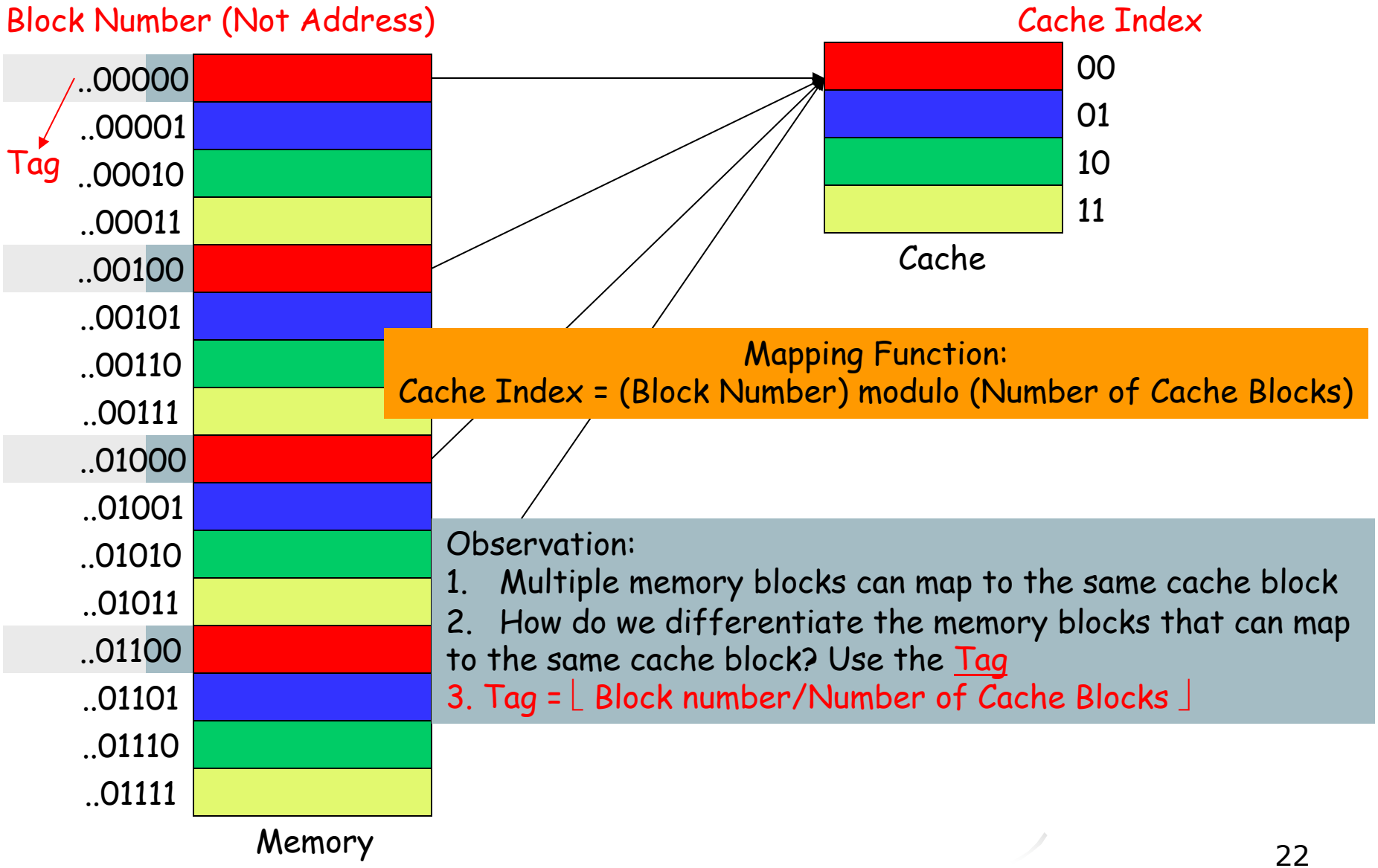


Block size = 2^N-bytes

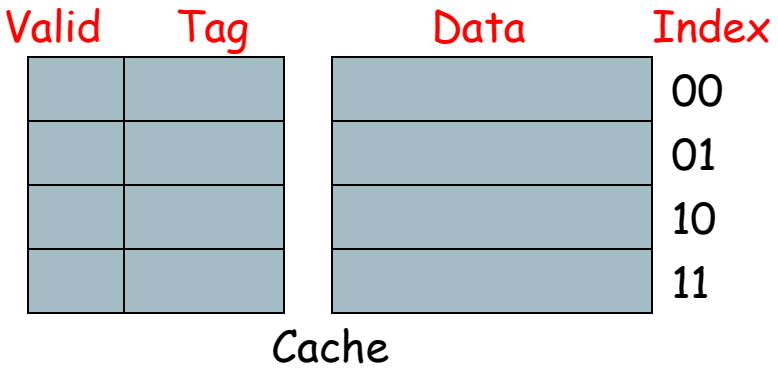


Block size = 2^N-bytes
Number of cache blocks = 2^M
Offset: N bits
Index: M bits
Tag: 32 – (N + M)

Why do we need Tag?



Cache Structure

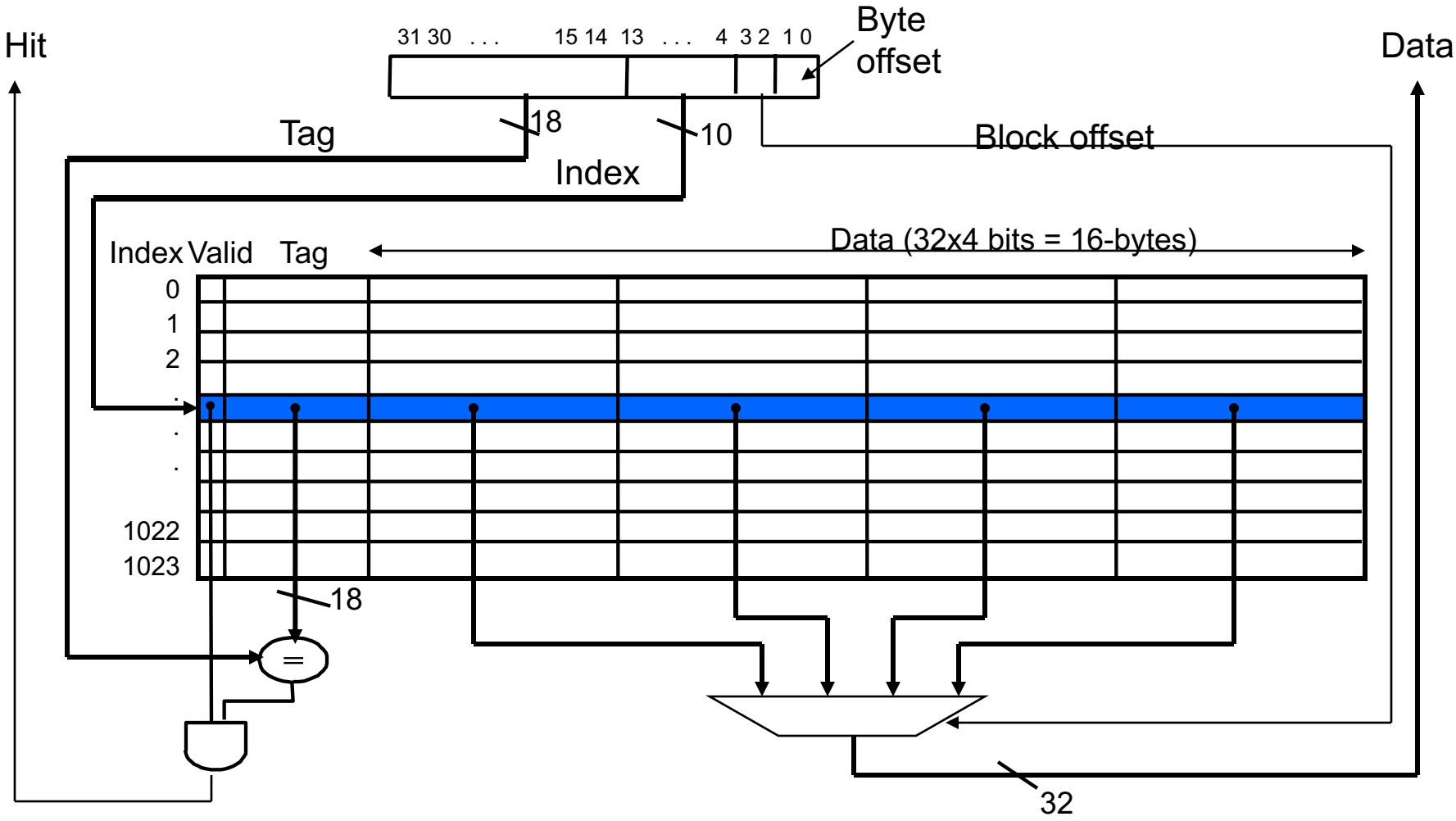


Along with a data block (line), cache contains

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

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

Cache Circuitry



Cache Example (contd.)

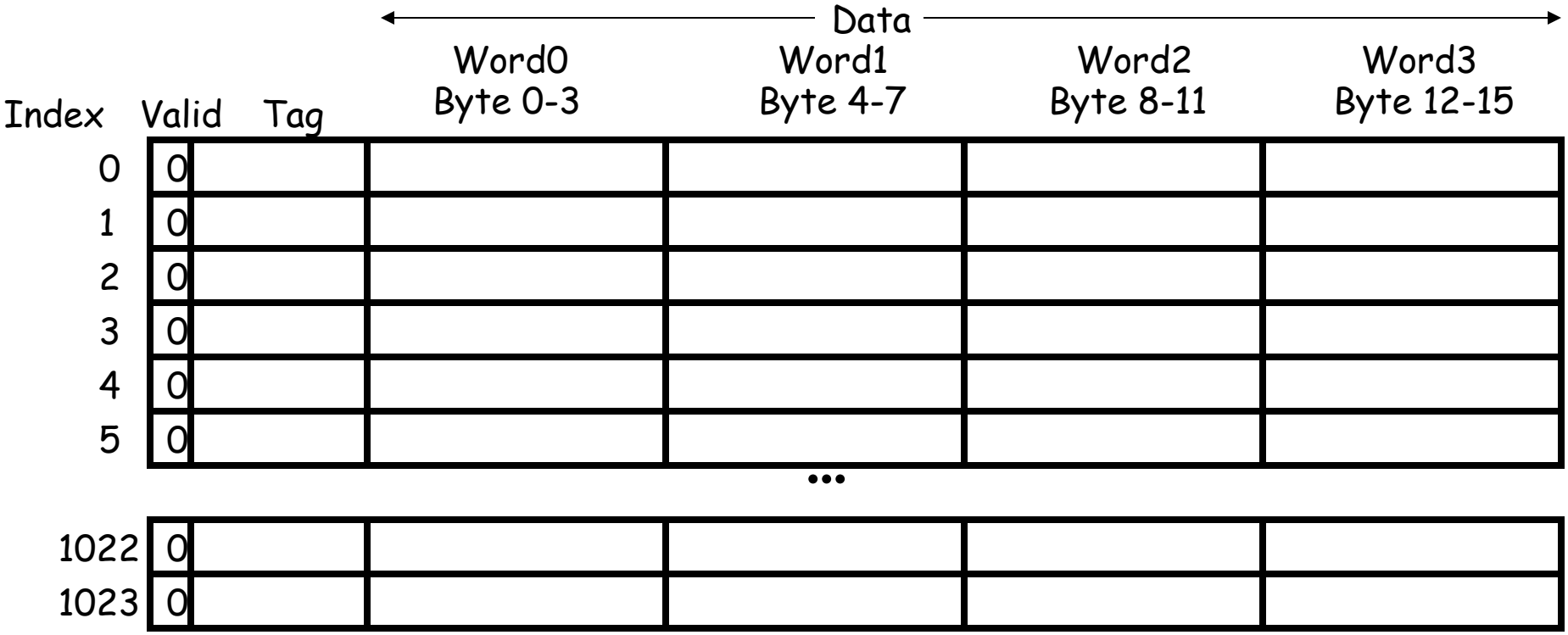
- 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

```
00000000000000000 0000000001 0100
00000000000000000 0000000001 1100
00000000000000000 0000000011 0100
00000000000000010 0000000001 0100
```

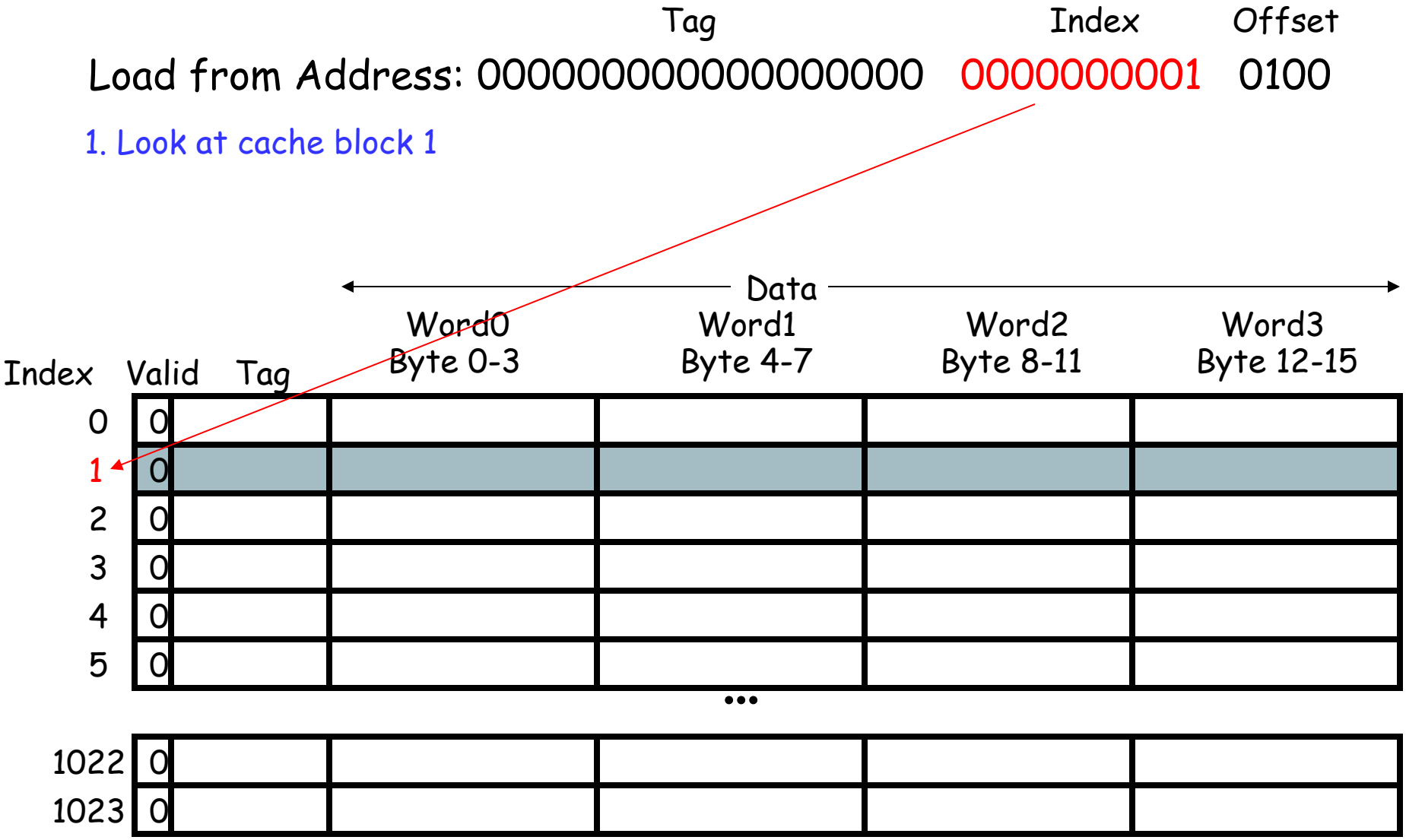
Tag Index Offset

Cache Example (contd.)

Initially cache is empty; all Valid bits are 0



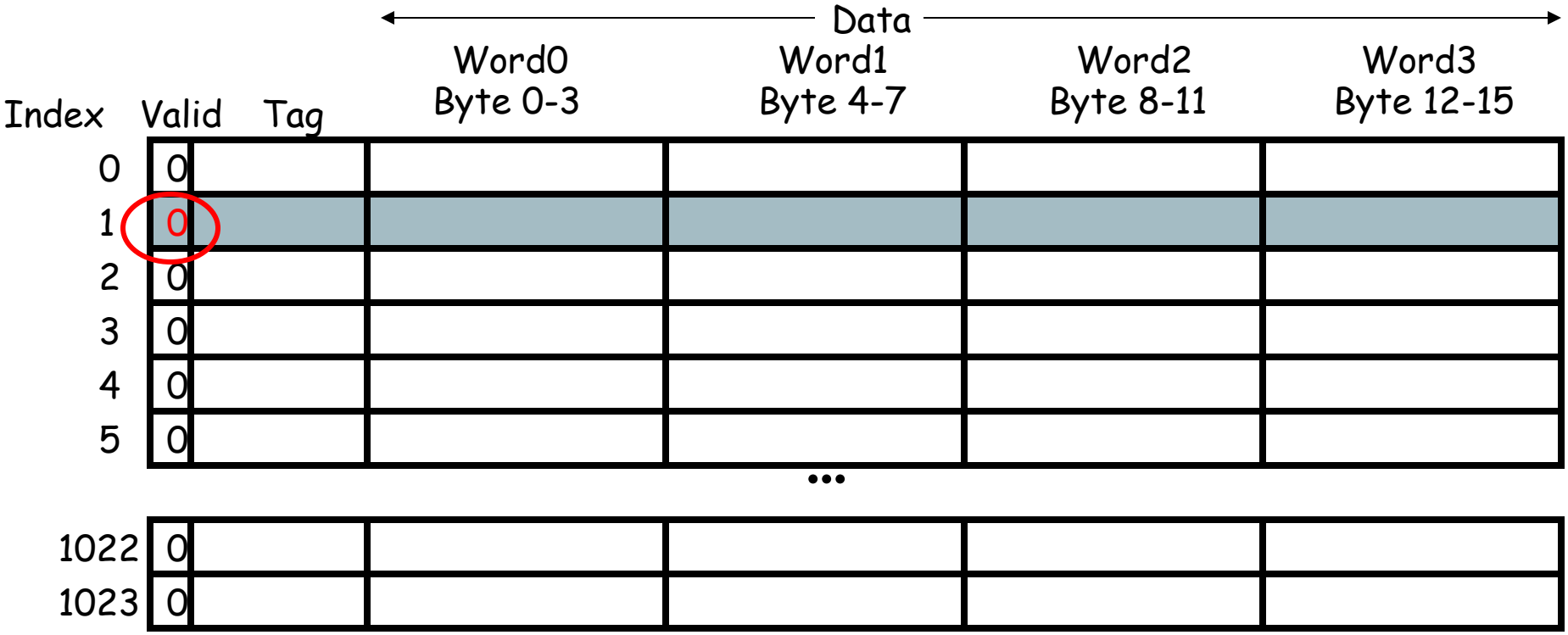
Cache Example (contd.)



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	00000000000000000	0000000001	0100

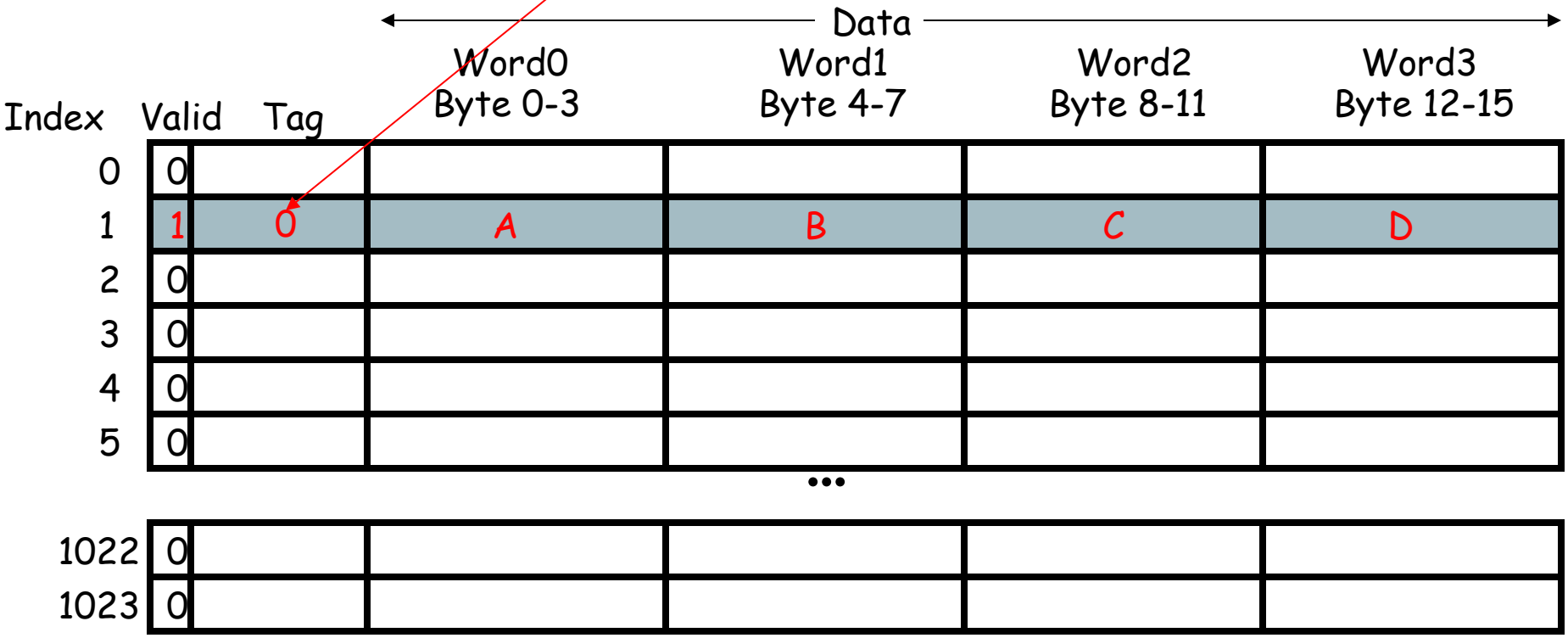
2. Data in block 1 is not valid → Cache Miss [Cold/Compulsory Miss]



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	0000000000000000	0000000001	0100

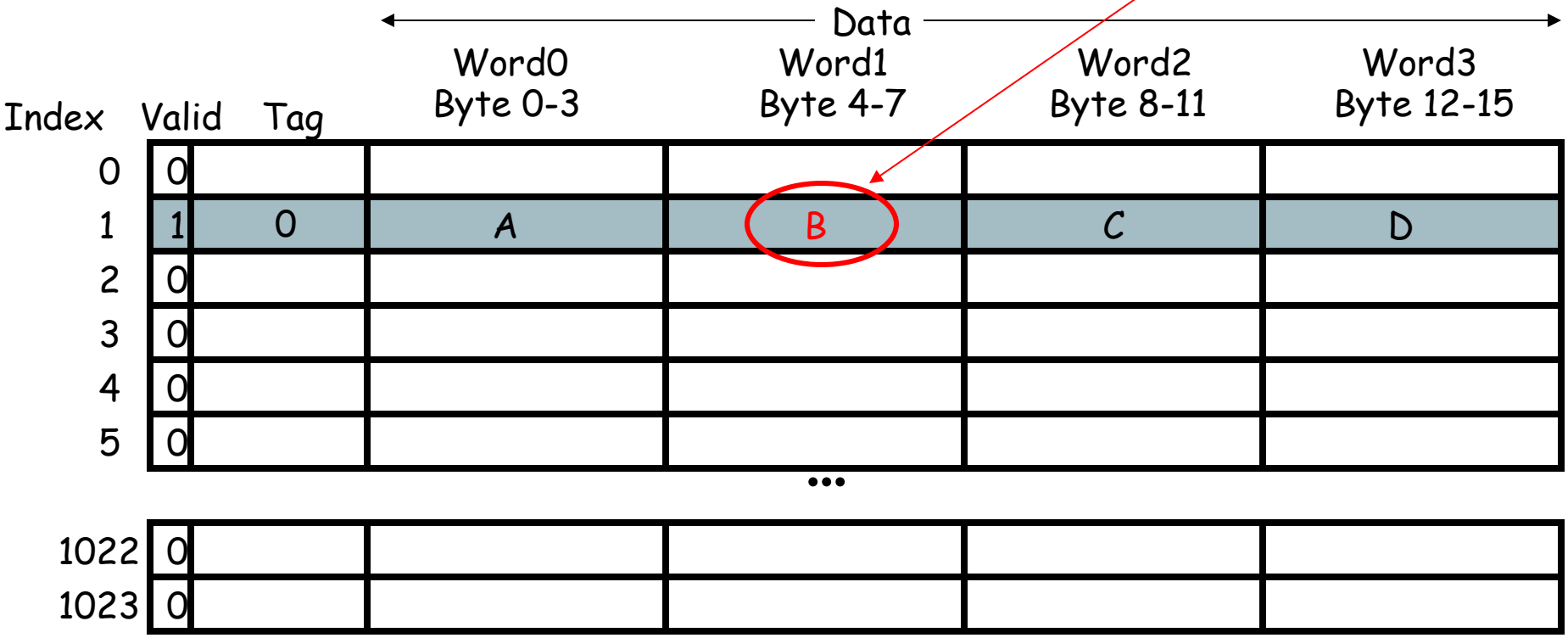
3. Load 16-byte data from memory to cache; set tag and valid bit



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	0000000000000000	0000000001	0100

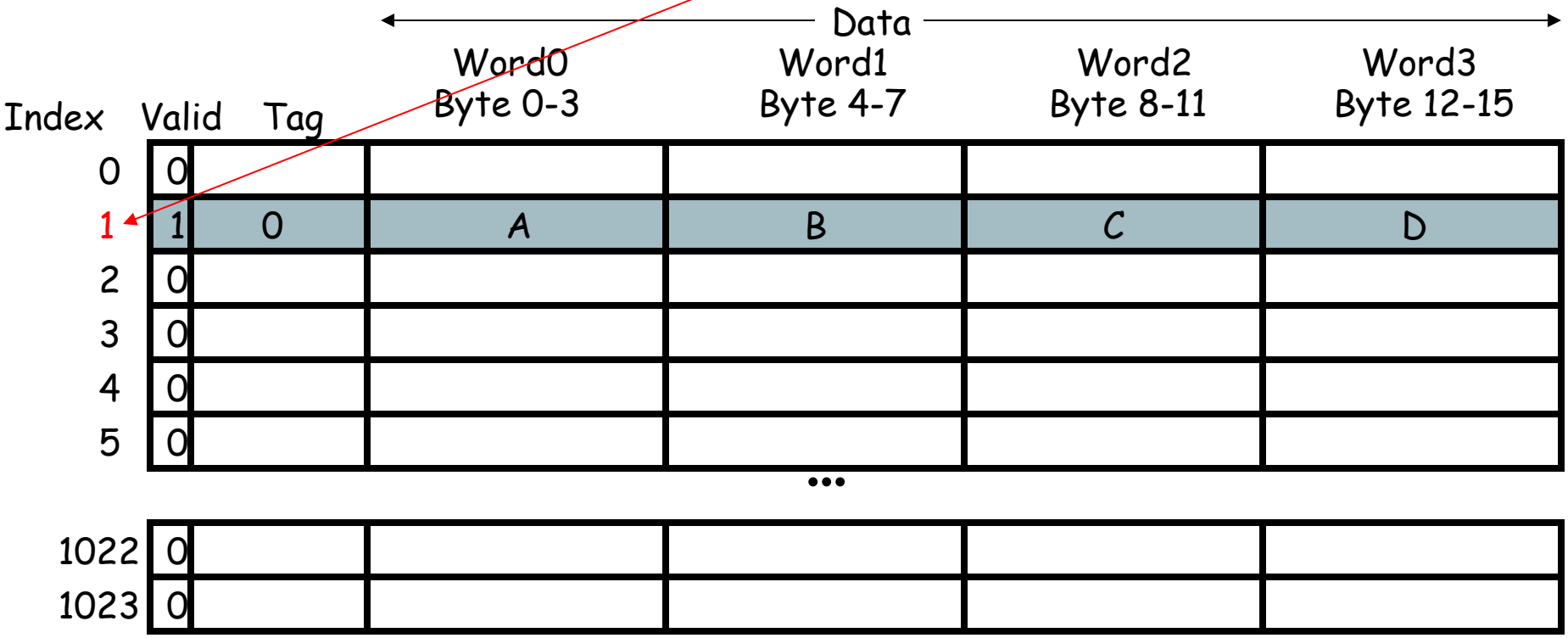
4. Load Word1 (byte offset = 4) from cache to register



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	00000000000000000	0000000001	1100

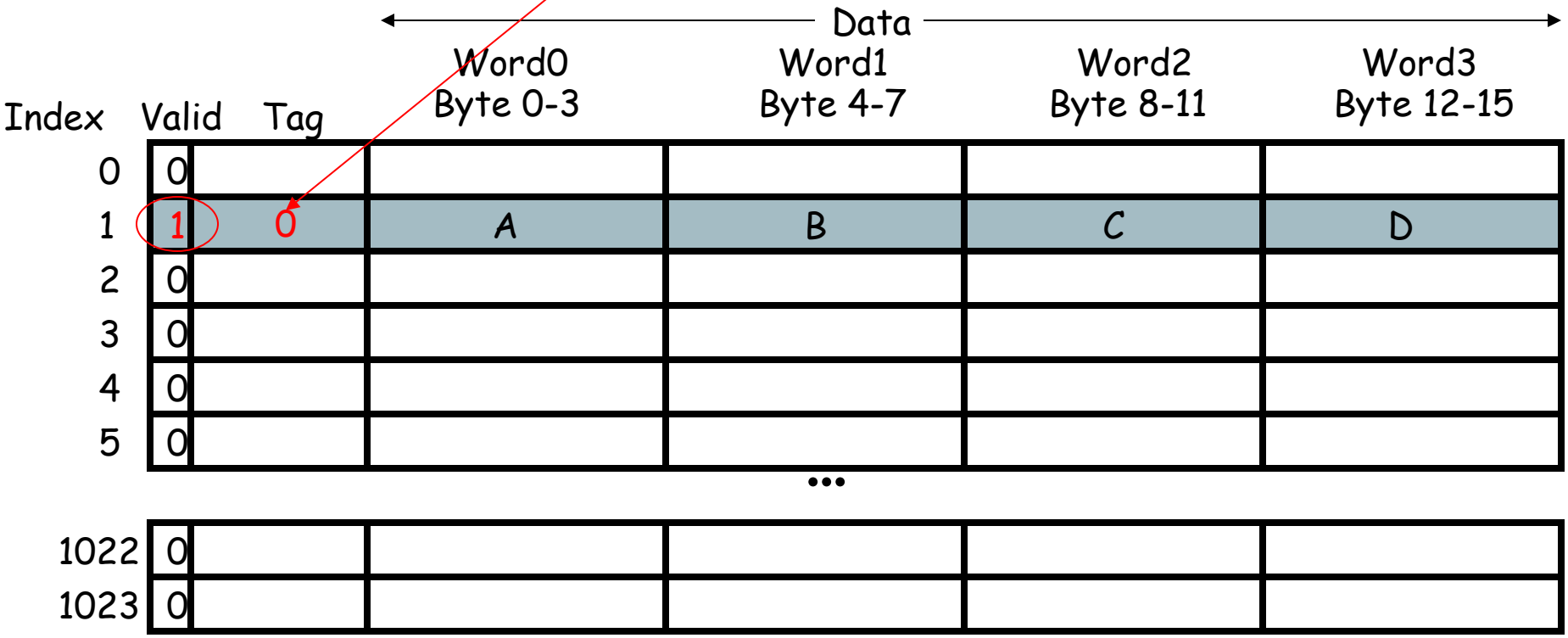
1. Look at cache block 1



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	00000000000000000	0000000001	1100

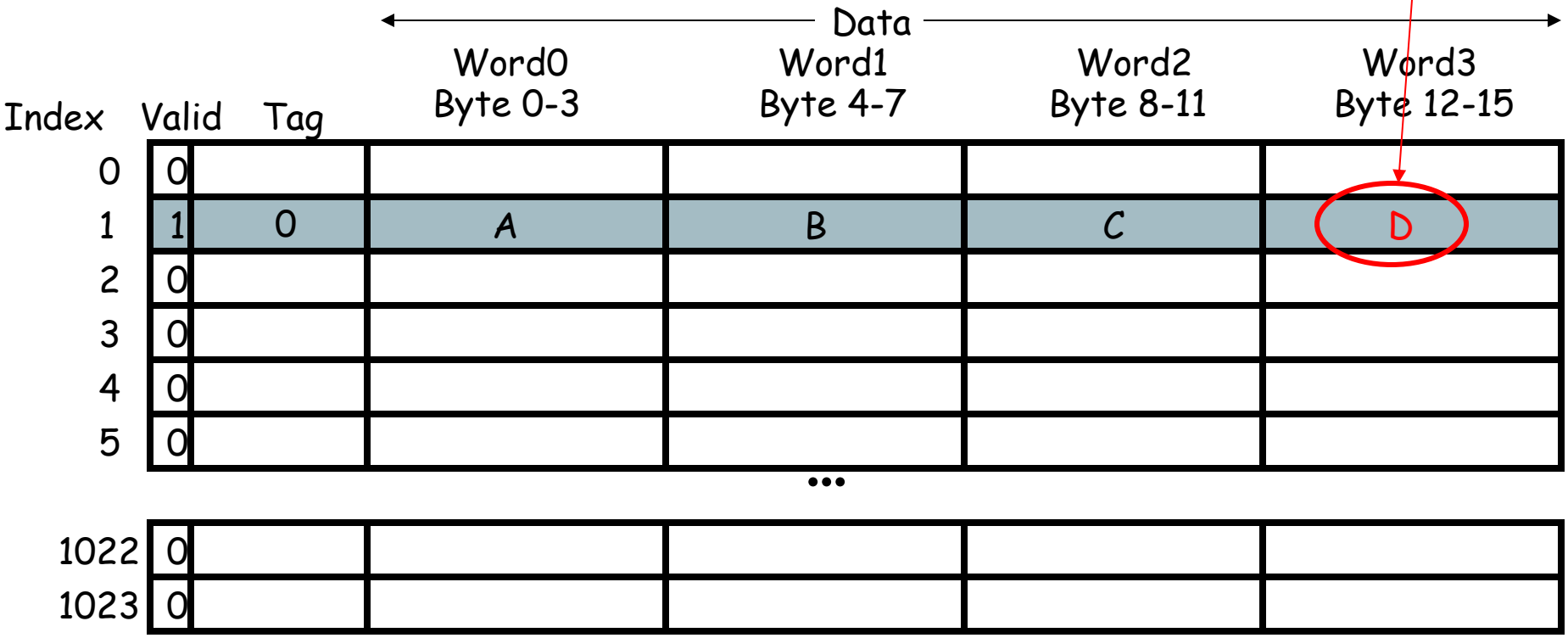
2. Data in block 1 is valid + Tag Match → Cache Hit



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	0000000000000000	0000000001	1100

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

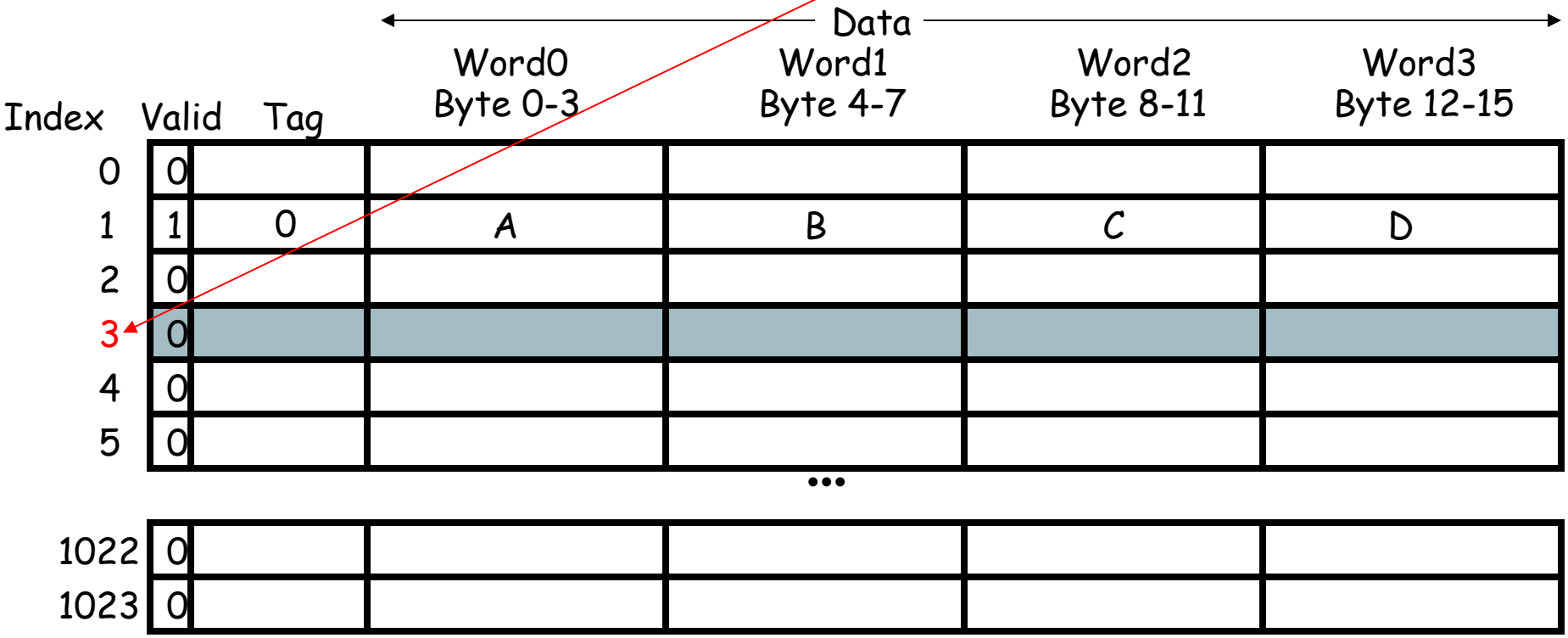


Cache Example (contd.)

Tag Index Offset

Load from Address: 00000000000000000 0000000011 0100

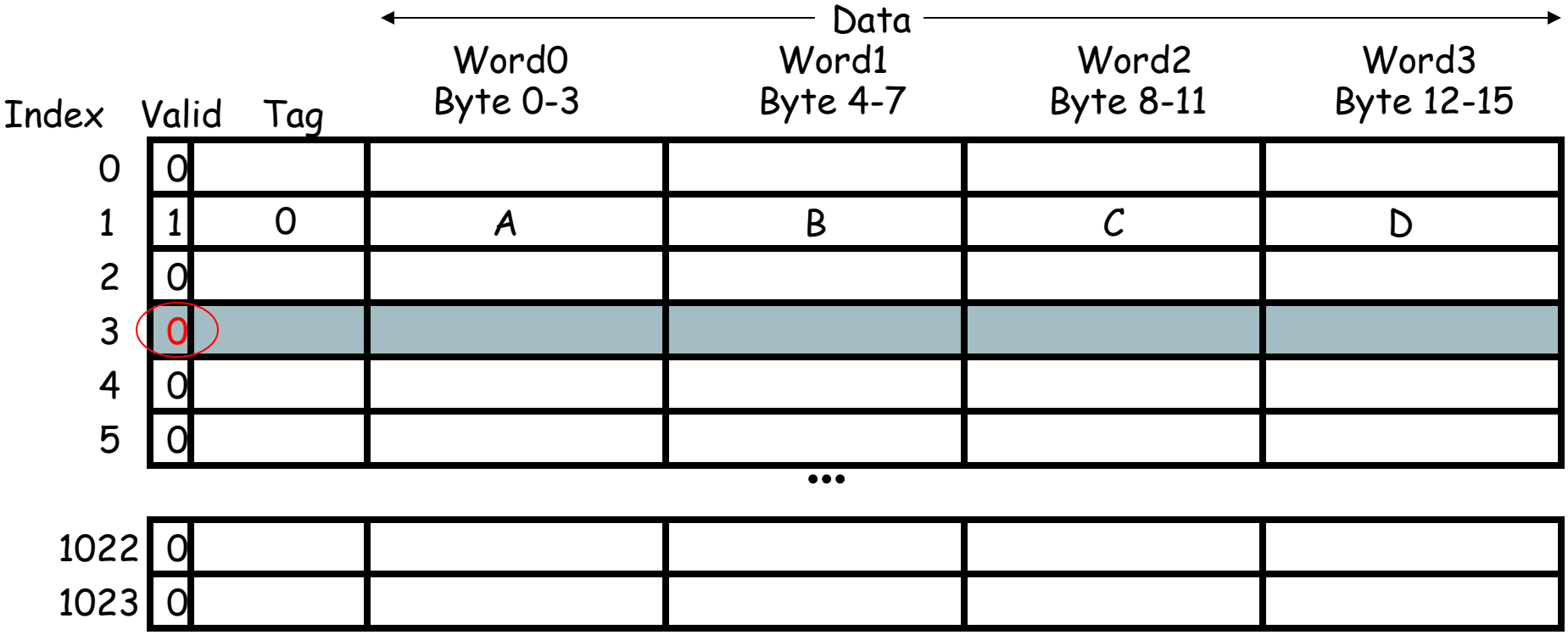
1. Look at cache block 3



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	00000000000000000	0000000011	0100

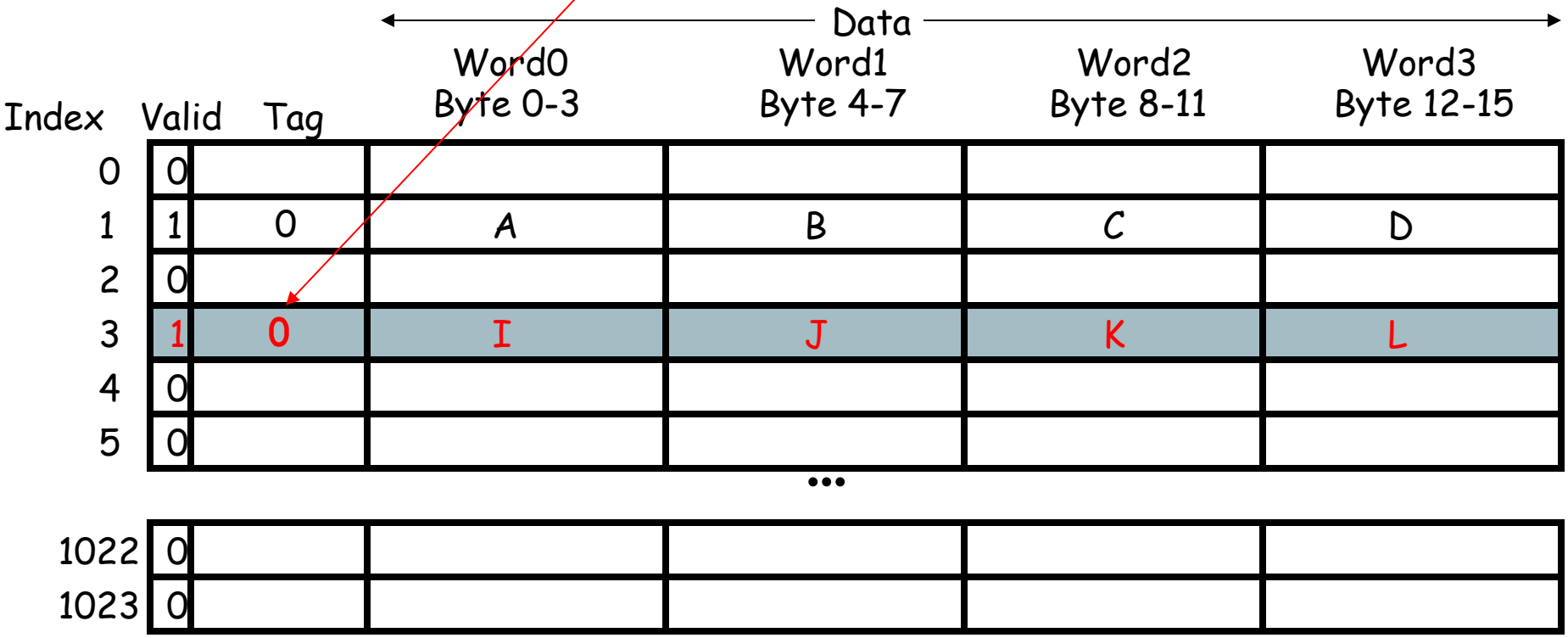
2. Data at cache block 3 in not valid → Cache Miss [Cold/Compulsory Miss]



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	0000000000000000	0000000011	0100

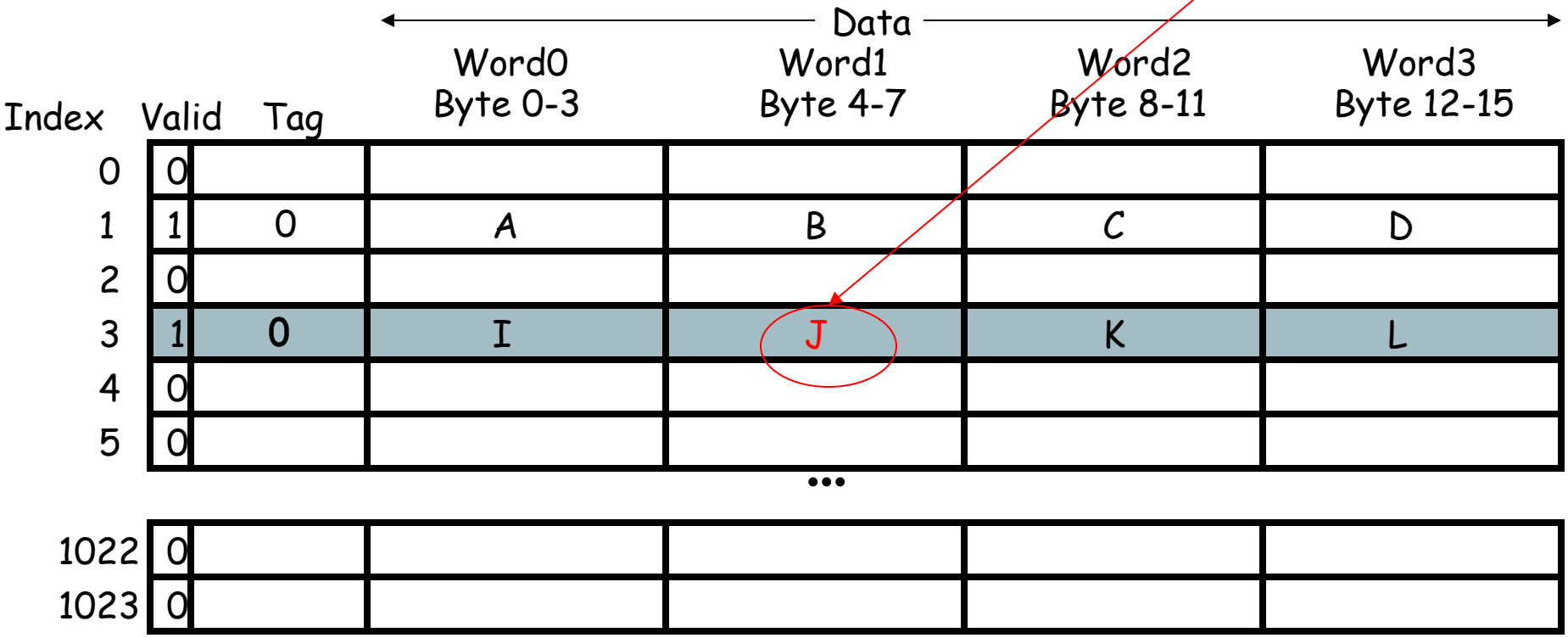
3. Load 16-byte data from main memory to cache; set tag and valid bit



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	0000000000000000	0000000011	0100

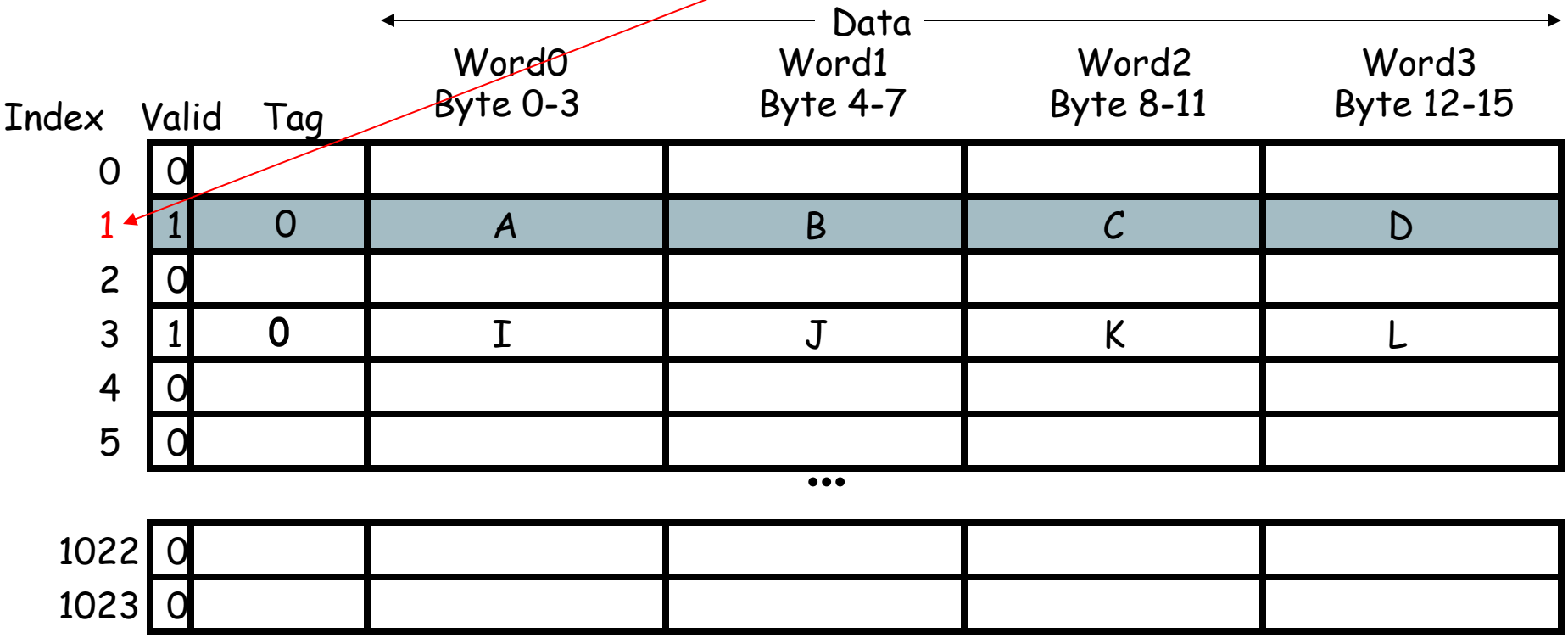
4. Return Word 1 (byte offset = 4) from cache to register



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	0000000000000010	0000000001	0100

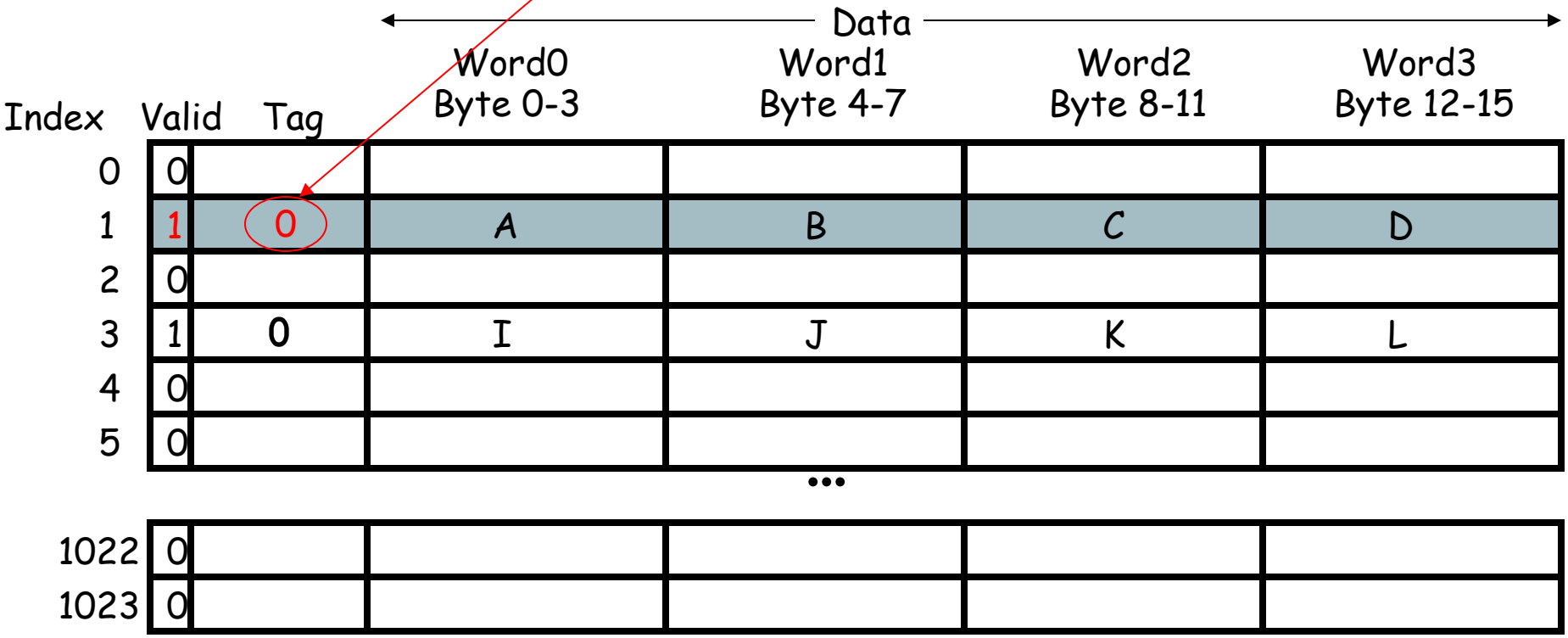
1. Look at cache block 1



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	0000000000000010	0000000001	0100

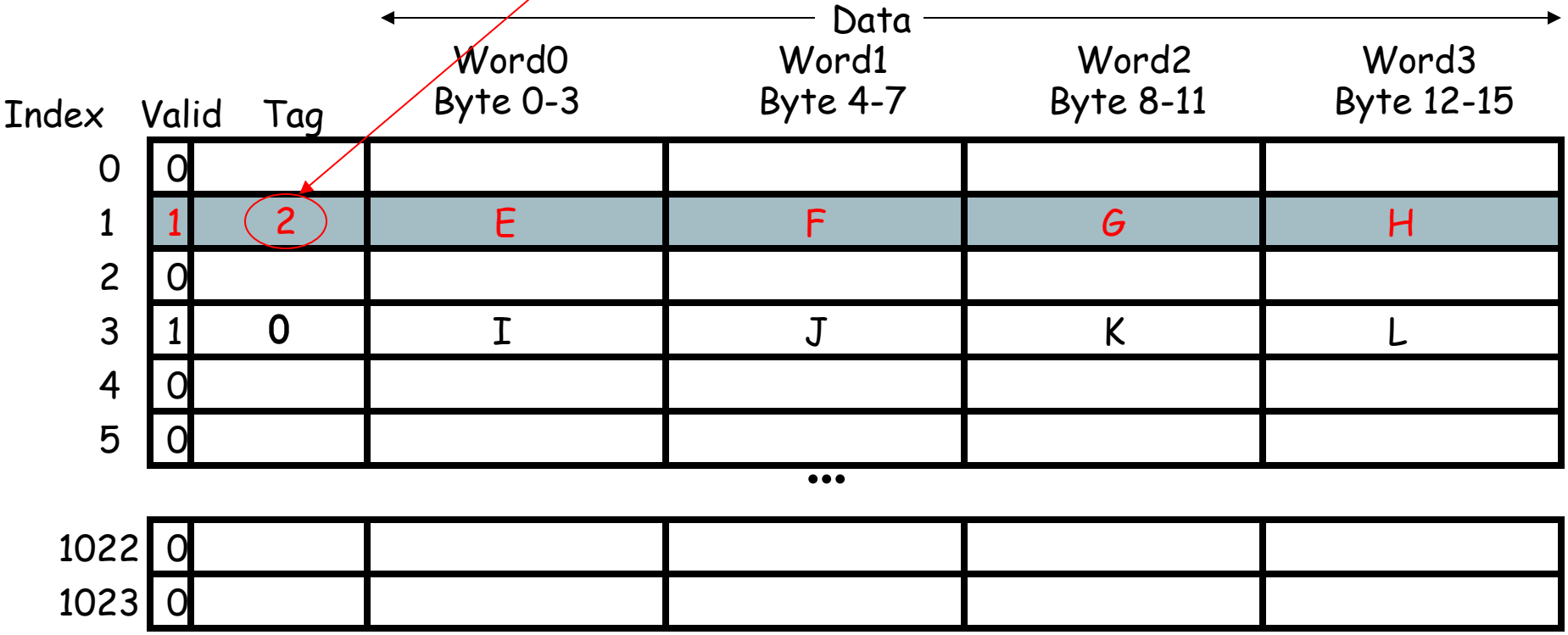
2. Data at cache block 1 is valid but tag mismatch → Cache Miss [Cold Miss]



Cache Example (contd.)

	Tag	Index	Offset
Load from Address:	0000000000000010	0000000001	0100

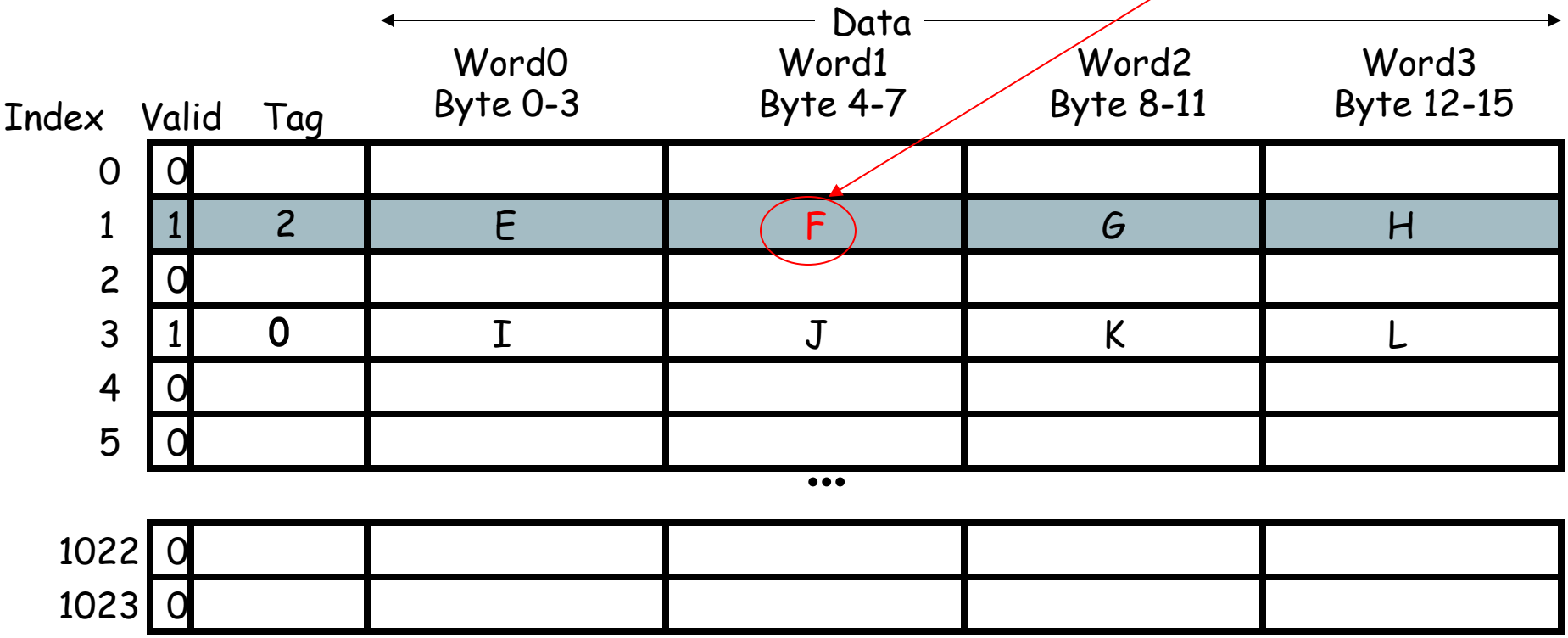
3. Replace cache block 1 with new data and tag



Cache Example (contd.)

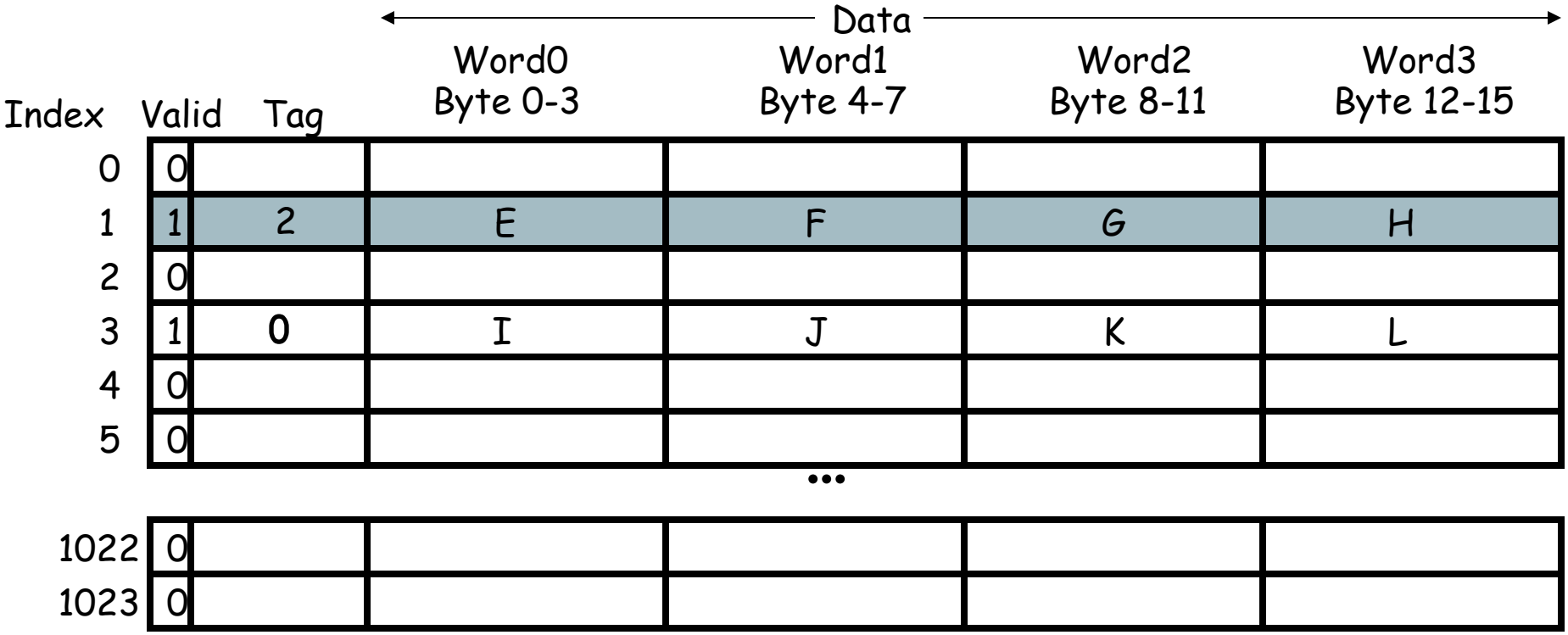
	Tag	Index	Offset
Load from Address:	0000000000000010	0000000001	0100

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

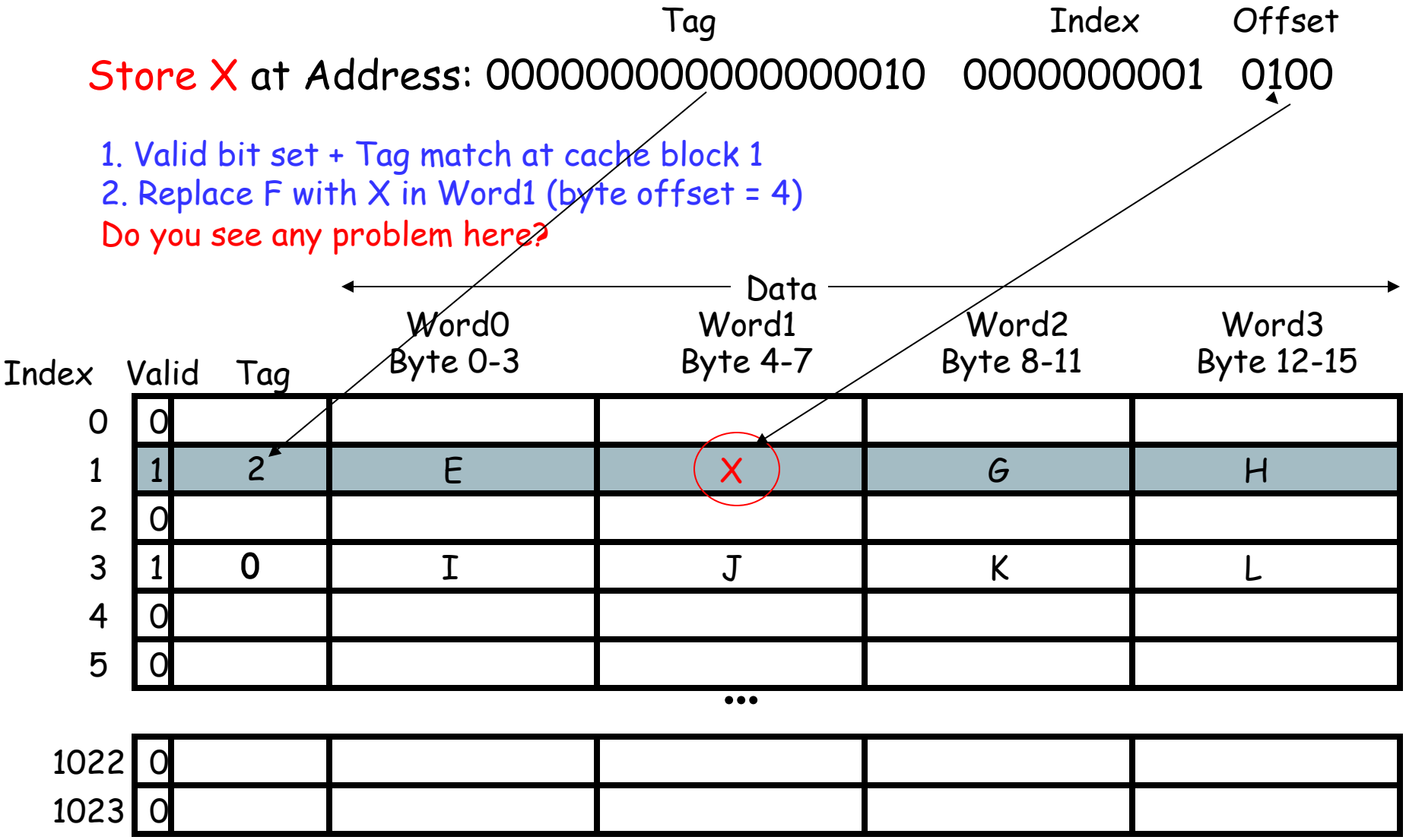


Writes to memory? (1/2)

	Tag	Index	Offset
Store X at Address:	0000000000000010	0000000001	0100



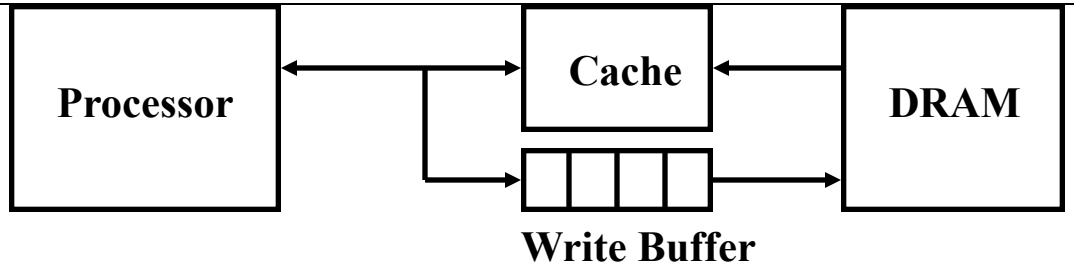
Writes to memory? (2/2)



Write Policy

- Cache and main memory are inconsistent
  - Data is modified in cache; but not in main memory
- 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
- Option 1: Write allocate
  - Load the complete block (16-byte 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?

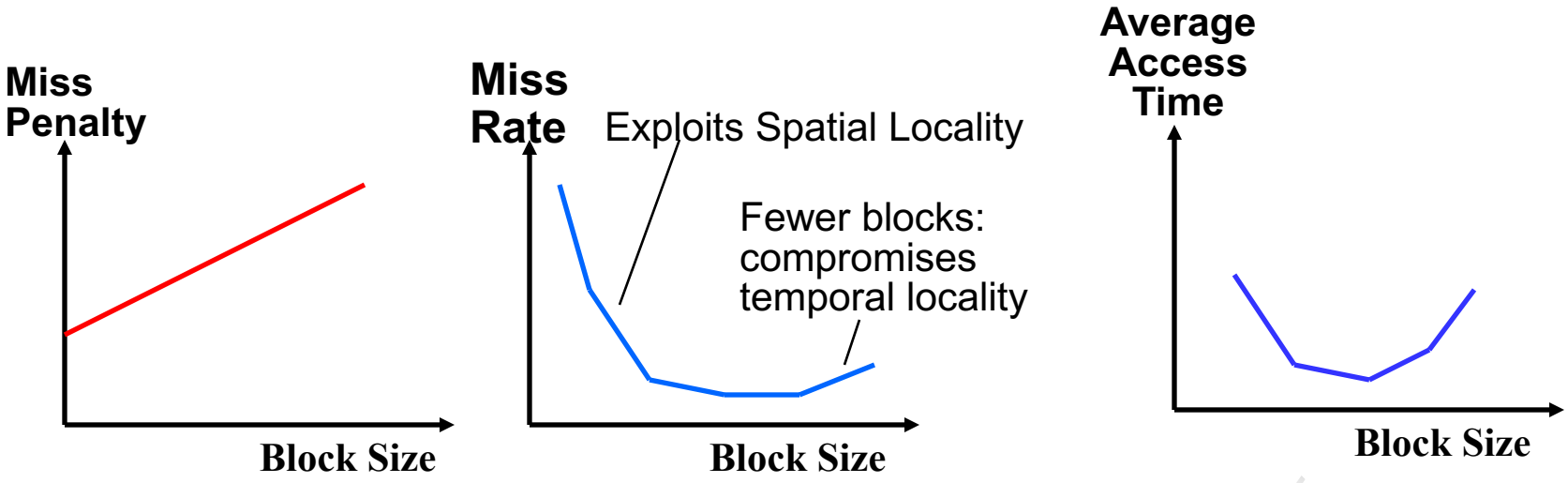
Types of Cache Misses

- Cold/Compulsory Miss: First time a memory address is accessed
  - Cold fact of life; not much we can do about it
  - Solution: Increase cache block size
- Conflict Miss: Two or more distinct memory blocks map to the same cache block
  - Big problem in direct-mapped caches
  - Solution 1: Increase cache size
    - Inherent restriction on cache size due to SRAM technology
  - Solution 2: Set-Associative caches (coming next ..)
- Capacity Miss: Due to limited cache size
  - Capacity Miss goes away if cache size can be increased to fully accommodate working sets
  - Vague definition for now; will become clear when we discuss fully associative caches

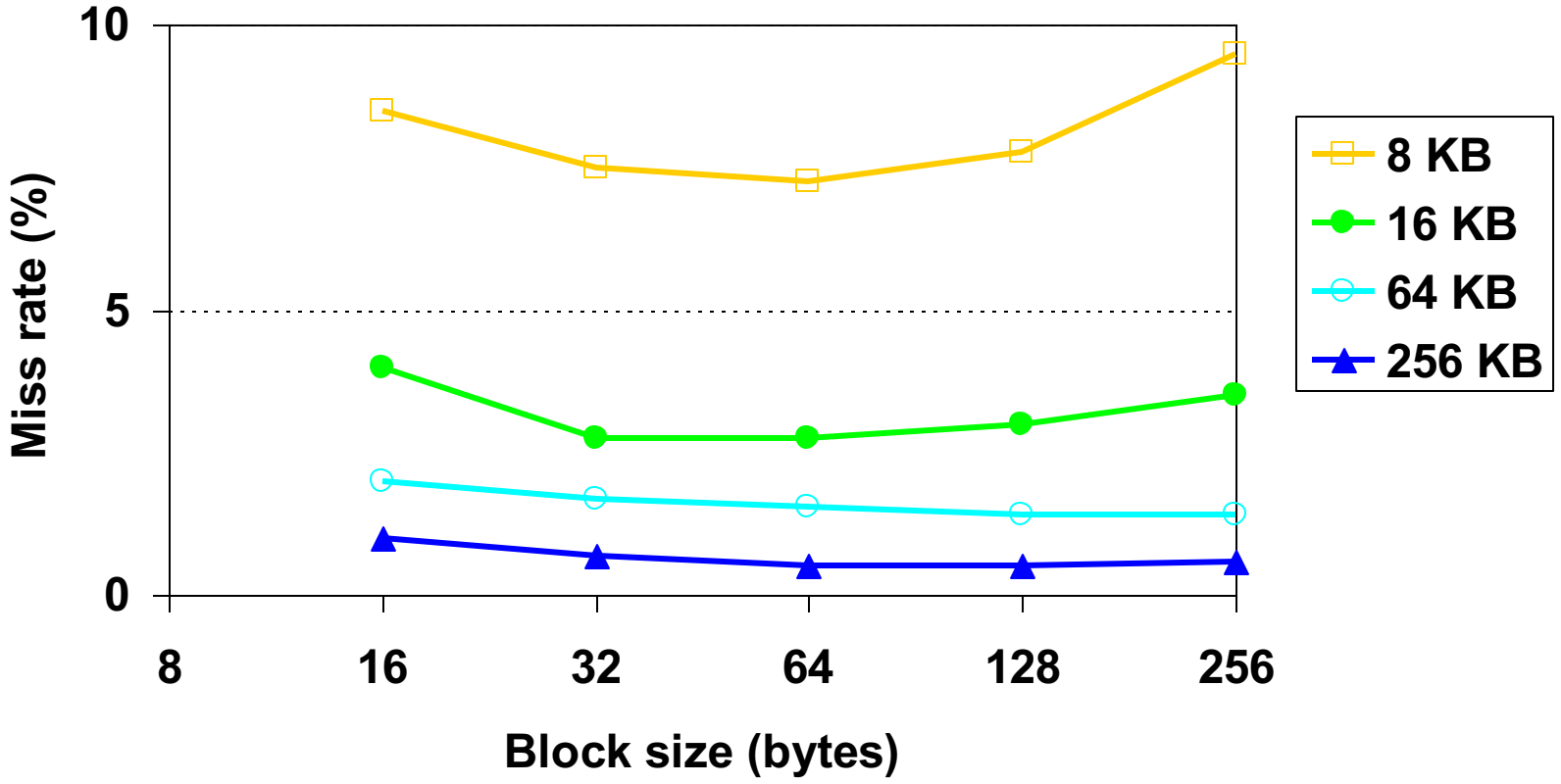
Block Size Tradeoff (1/2)

Average access time = Hit rate x Hit Time + (1-Hit rate) x Miss penalty

- In general, larger block size takes advantage of spatial locality BUT:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size is too big relative to cache size, miss rate will go up
    - Too few cache blocks



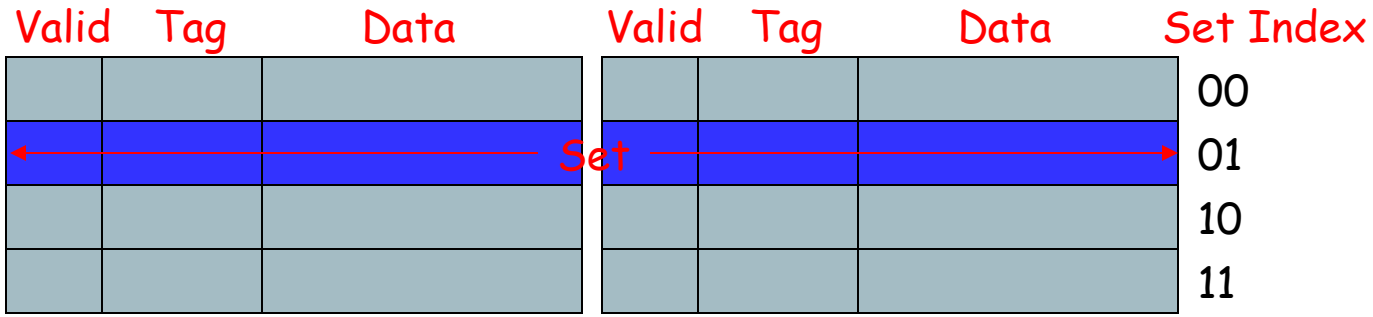
Block Size Tradeoff (2/2)



Set Associative Cache

- A memory block can be placed in a fixed number of locations (n > 1) in the cache
  - n-way set associative cache
- n-way set associative cache consists of a number of sets where each set contains n cache blocks
- Each memory block maps to a unique cache set
- Within the set, a memory block can be placed in any element of the set

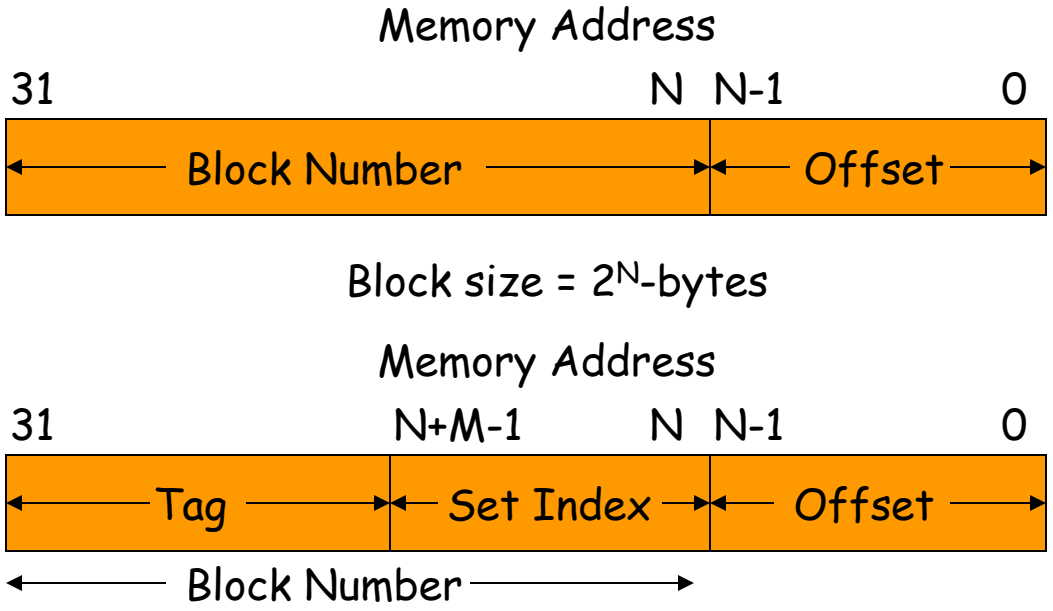
Set Associative Cache Structure



- 2-way Set Associative Cache
  - Each cache set contains two cache blocks
- A memory block can be mapped to a unique cache set
- Within a cache set, the memory block can be placed anywhere
  - Both the cache blocks within a set should be searched in parallel

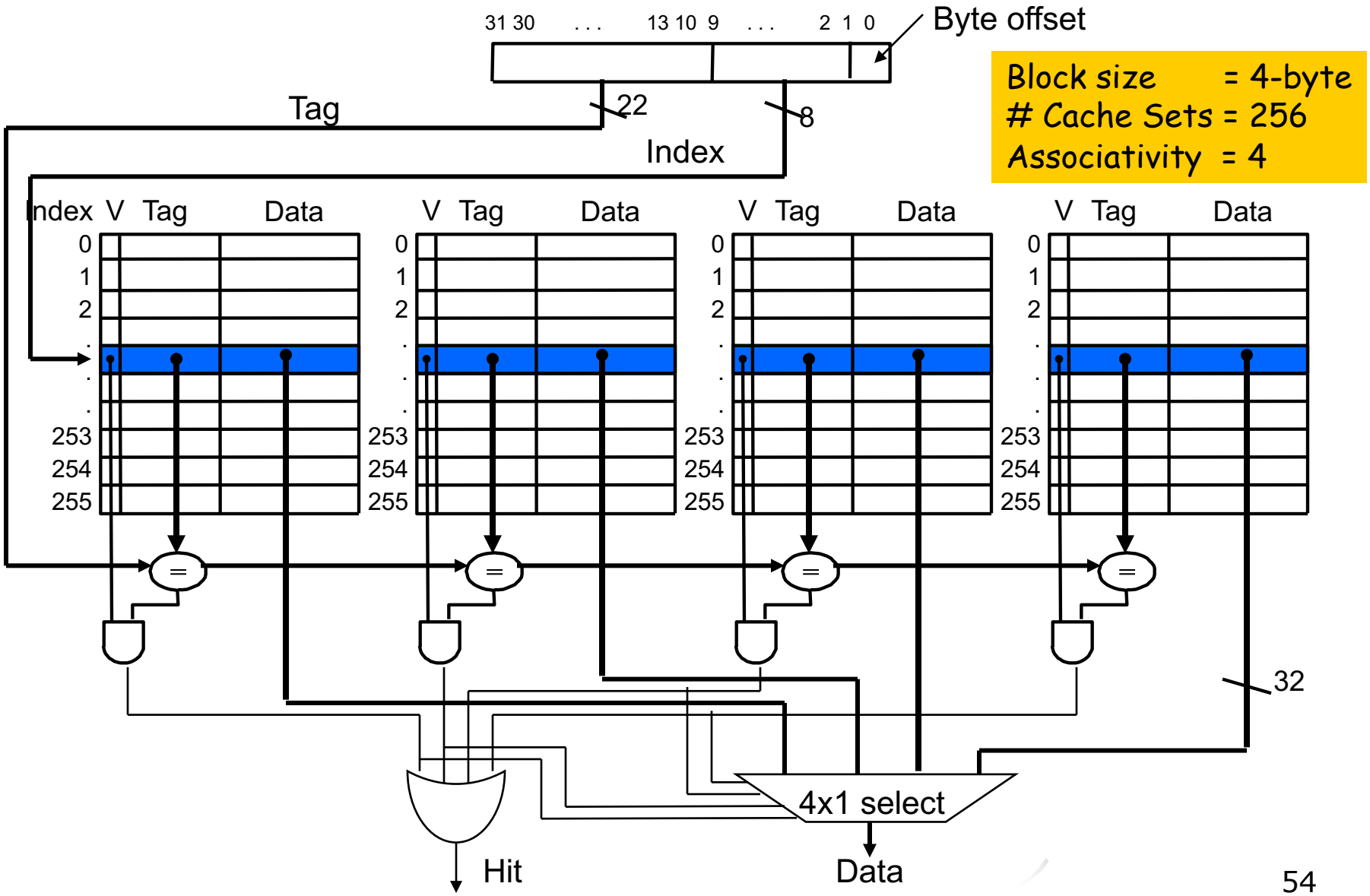
Set Associative Mapping

Cache Set Index = (Block number) modulo (Number of Cache Sets)

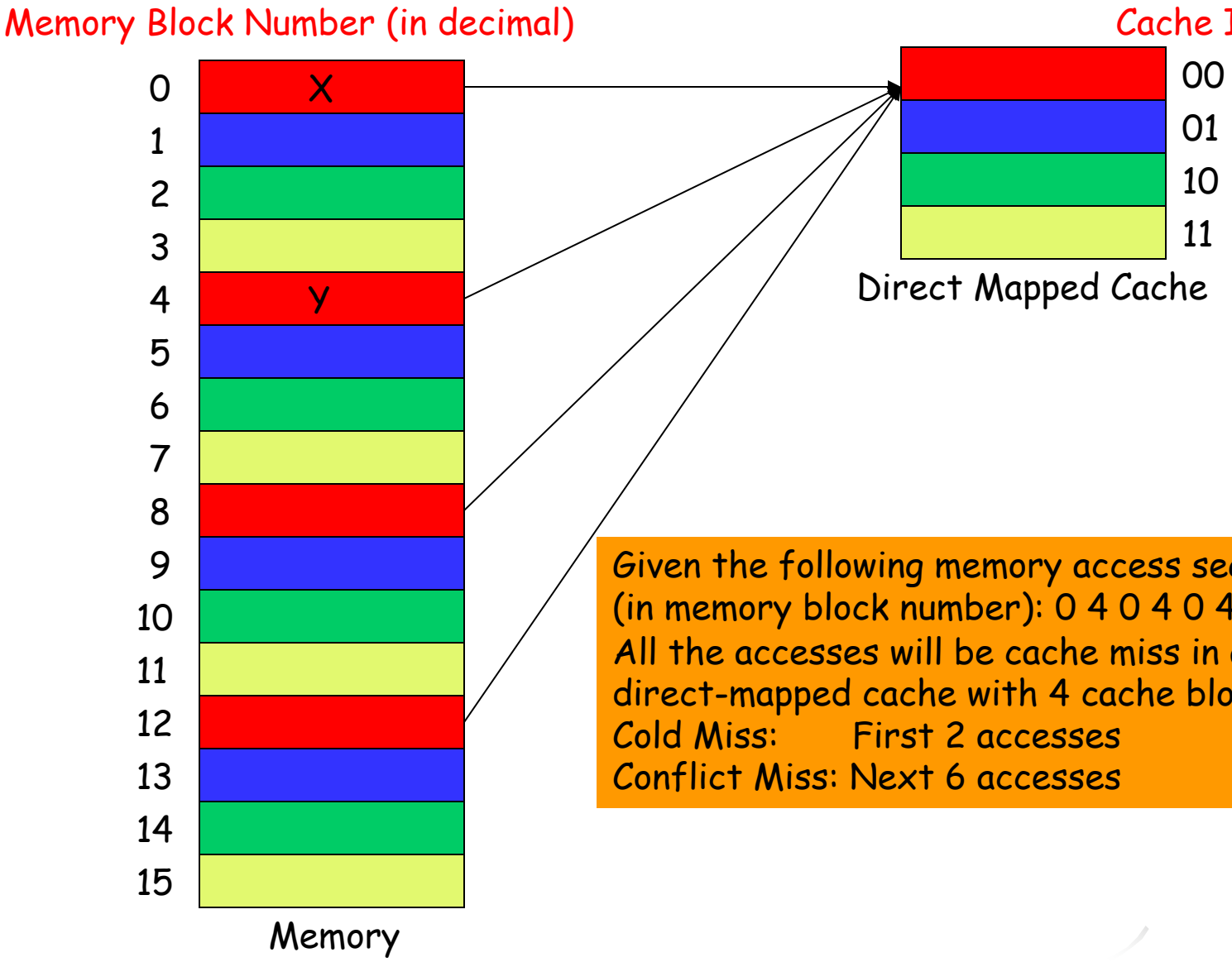


Block size = 2^N-bytes
Number of cache sets = 2^M
Offset: N bits
Set Index: M bits
Tag: 32 – (N + M)

Set Associative Cache Circuitry

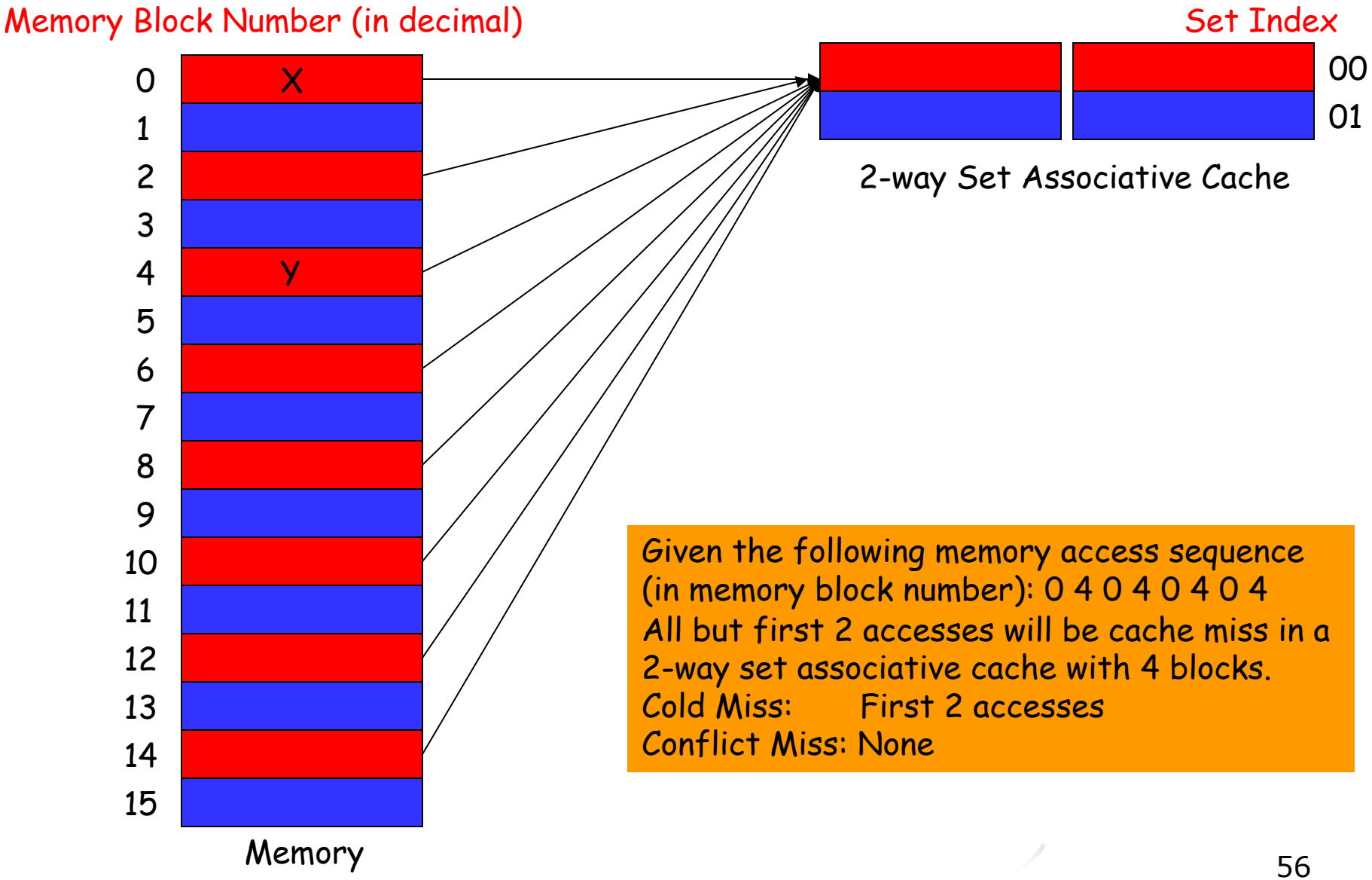


Advantage of Associativity (1/3)



Given the following memory access sequence (in memory block number): 0 4 0 4 0 4 0 4
All the accesses will be cache miss in a direct-mapped cache with 4 cache blocks.
Cold Miss: First 2 accesses
Conflict Miss: Next 6 accesses

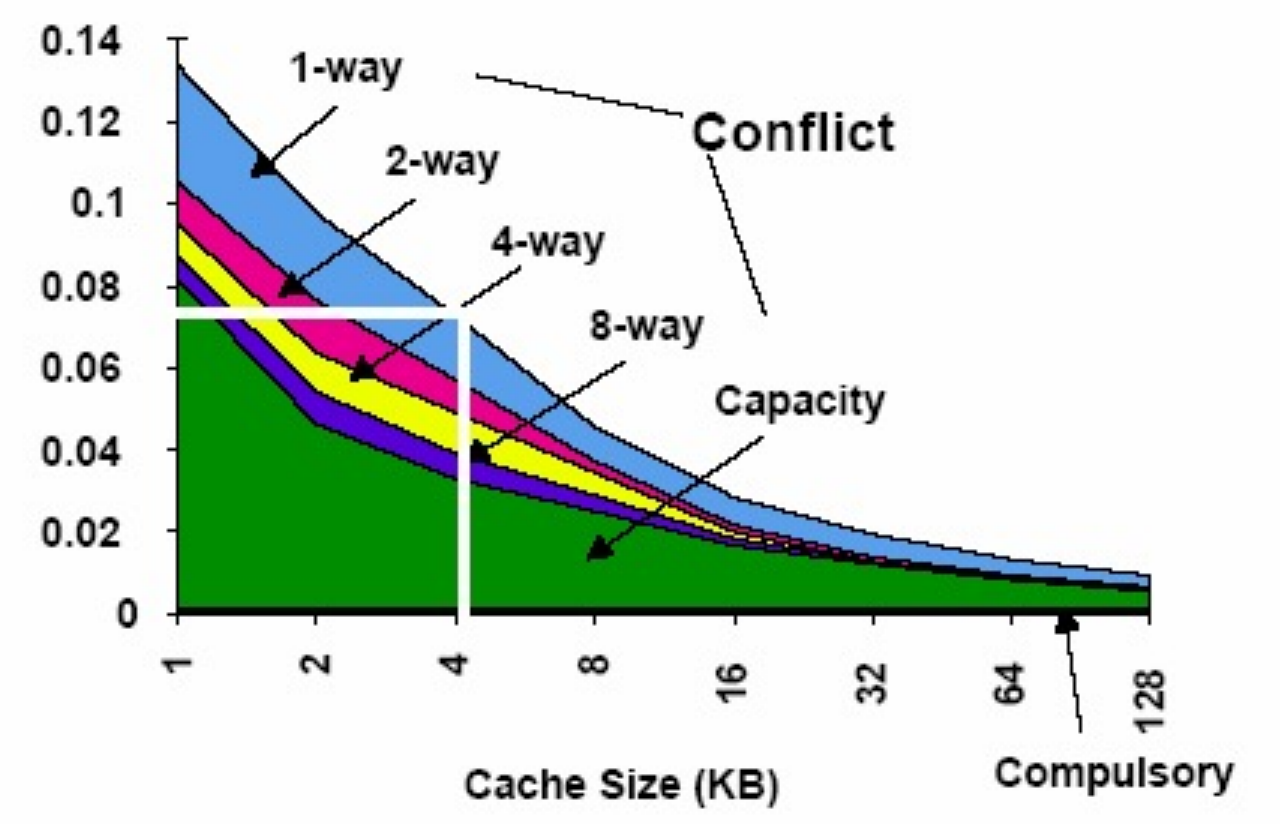
Advantage of Associativity (2/3)



Given the following memory access sequence (in memory block number): 0 4 0 4 0 4 0 4
All but first 2 accesses will be cache miss in a 2-way set associative cache with 4 blocks.
Cold Miss: First 2 accesses
Conflict Miss: None

Advantage of Associativity (3/3)

Rule of Thumb: A direct-mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N/2



Example

- Here is a series of memory address references: 4, 0, 8, 36, 0 for a 2-way set-associative cache with four 8-byte blocks. Indicate hit/miss for each reference.
- Take address = 36
- Memory block number = $\lfloor 36/8 \rfloor = 4$
- Block offset = 36 mod 8 = 4
- Cache set index = 4 mod 2 = 0
- Tag = $\lfloor 4/2 \rfloor = 2$
- Binary: ..0100100

Example (contd.)

Addresses: 4, 0, 8, 36, 0

4 = ..00 0 100

Set Index = 0, Tag = 0, Offset = 4

Both blocks in set 0 are invalid

Cold/Compulsory Miss

Valid	Tag	Word0	Word1	Valid	Tag	Word0	Word1
0				0			
0				0			

Example (contd.)

Addresses: 4, 0, 8, 36, 0

4 = ..00 0 100

Set Index = 0, Tag = 0, Offset = 4

Put memory block in set 0

Valid	Tag	Word0	Word1	Valid	Tag	Word0	Word1
1	0	M[0]	M[4]	0			
0				0			

Example (contd.)

Addresses: 4, 0, 8, 36, 0

0 = ..00 0 000

Set Index = 0, Tag = 0, Offset = 0

Valid +Tag match for first block in set 0

Cache Hit due to spatial locality

Valid	Tag	Word0	Word1	Valid	Tag	Word0	Word1
1	0	M[0]	M[4]	0			
0				0			

Example (contd.)

Addresses: 4, 0, 8, 36, 0

8 = ..00 1 000

Set Index = 1, Tag = 0, Offset = 0

Both blocks in set 1 are invalid

Cold/Compulsory Miss

Valid	Tag	Word0	Word1	Valid	Tag	Word0	Word1
1	0	M[0]	M[4]	0			
0				0			

Example (contd.)

Addresses: 4, 0, 8, 36, 0

8 = ..00 1 000

Set Index = 1, Tag = 0, Offset = 0

Put memory block in set 1

Valid	Tag	Word0	Word1	Valid	Tag	Word0	Word1
1	0	M[0]	M[4]	0			
1	0	M[8]	M[12]	0			

Example (contd.)

Addresses: 4, 0, 8, 36, 0

36 = ..10 0 100

Set Index = 0, Tag = 2, Offset = 4

First block in set 0: Tag mismatch

Second block in set 0: Invalid

Cold/Compulsory Miss

Valid	Tag	Word0	Word1	Valid	Tag	Word0	Word1
1	0	M[0]	M[4]	0			
1	0	M[8]	M[12]	0			

Example (contd.)

Addresses: 4, 0, 8, 36, 0

36 = ..10 0 100

Set Index = 0, Tag = 2, Offset = 4

Put memory block in set 0

Valid	Tag	Word0	Word1	Valid	Tag	Word0	Word1
1	0	M[0]	M[4]	1	2	M[32]	M[36]
1	0	M[8]	M[12]	0			

Example (contd.)

Addresses: 4, 0, 8, 36, 0

0 = ..00 0 000

Set Index = 0, Tag = 0, Offset = 0

First block in set 0: Match

Second block in set 0: Mismatch

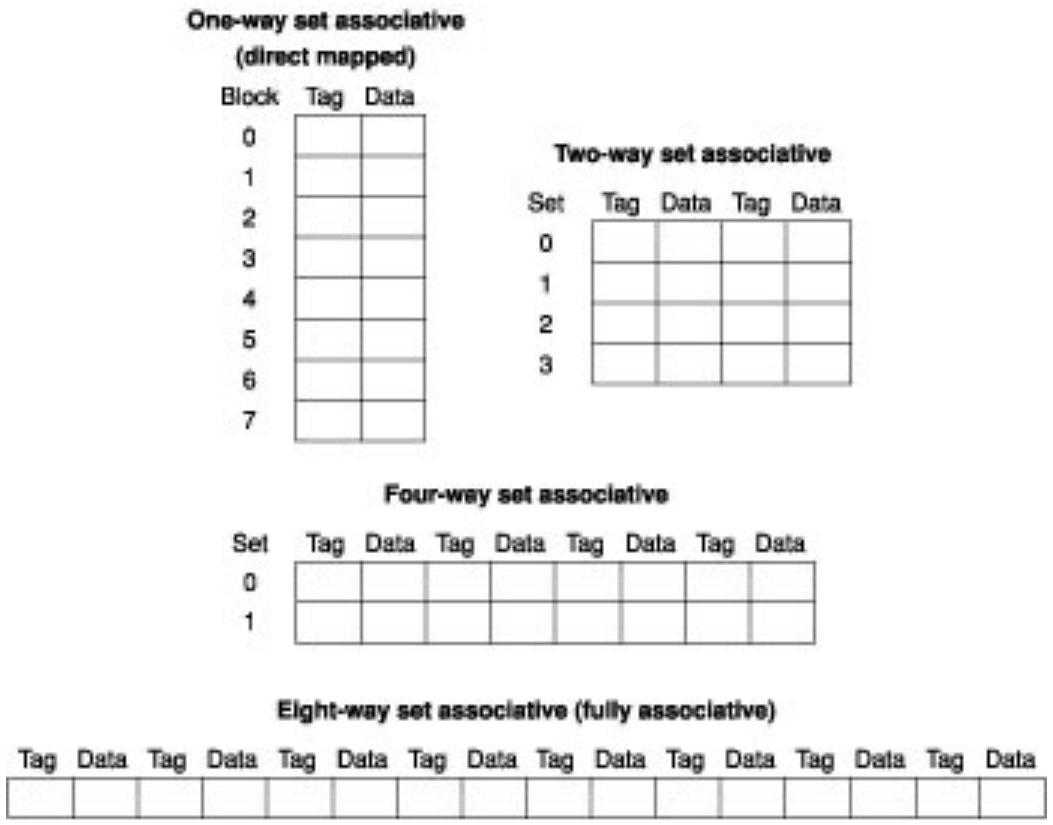
Cache hit due to temporal locality

Valid	Tag	Word0	Word1	Valid	Tag	Word0	Word1
1	0	M[0]	M[4]	1	2	M[32]	M[36]
1	0	M[8]	M[12]	0			

Fully Associative (FA) Cache

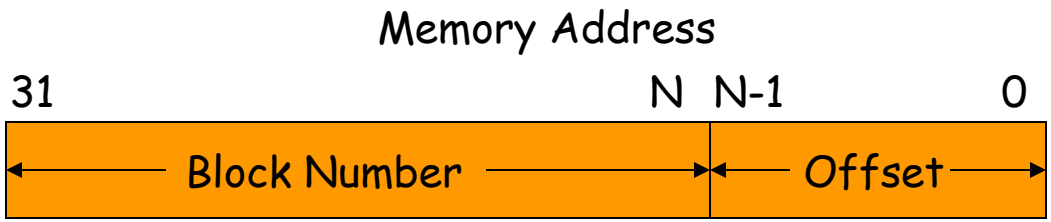
- Extreme case of set-associative cache
- A memory block can be placed in any cache block
- Need to search all the cache blocks in parallel
- Advantage: Zero conflict miss
- Disadvantage: High hardware cost due to comparators
- Not used except for very very small caches

Direct mapped → Fully Associative

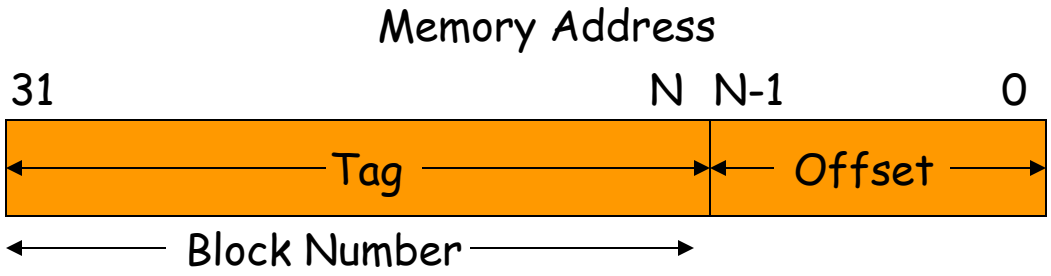


Fully Associative Mapping

No index; only Tag



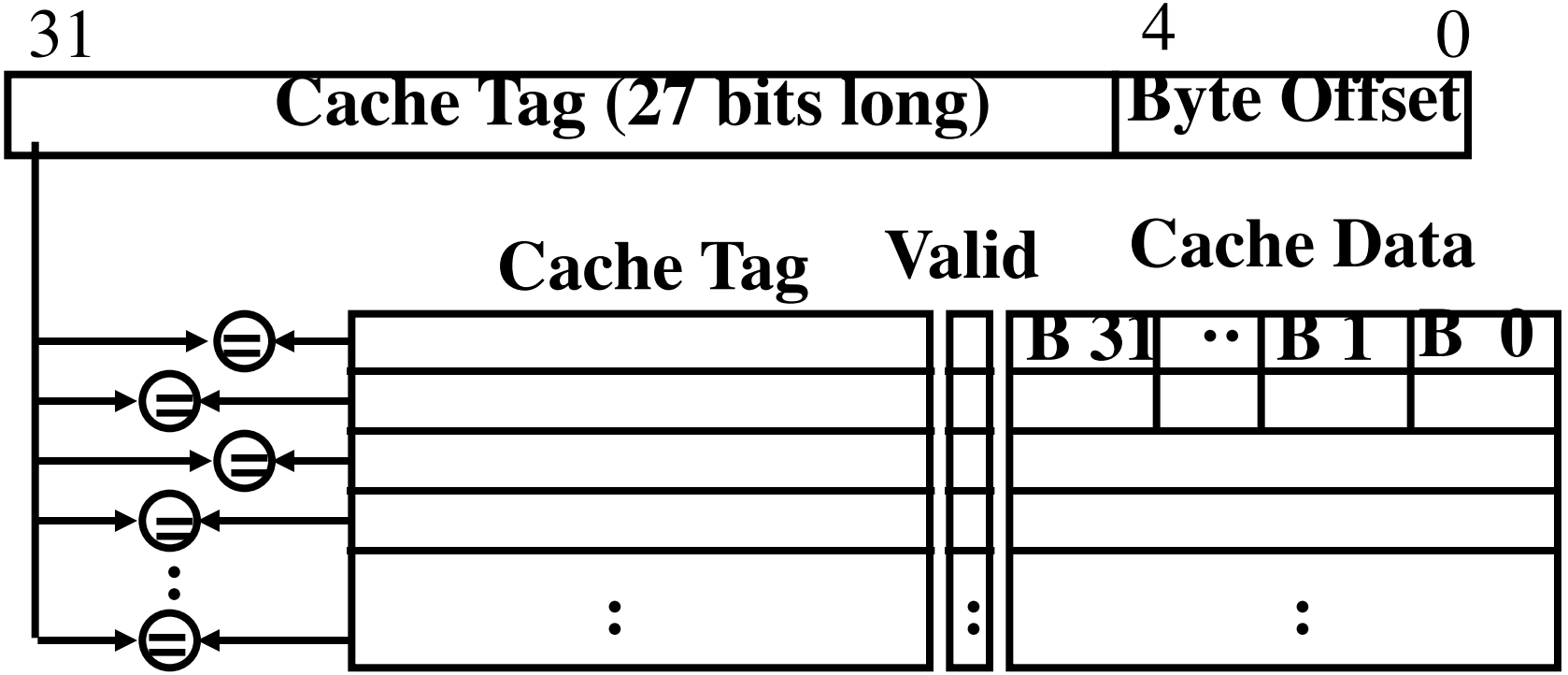
Block size = 2^N-bytes



Block size = 2^N-bytes
Number of cache blocks = 2^M
Offset: N bits
Tag: 32 – N

Fully Associative Cache Circuitry

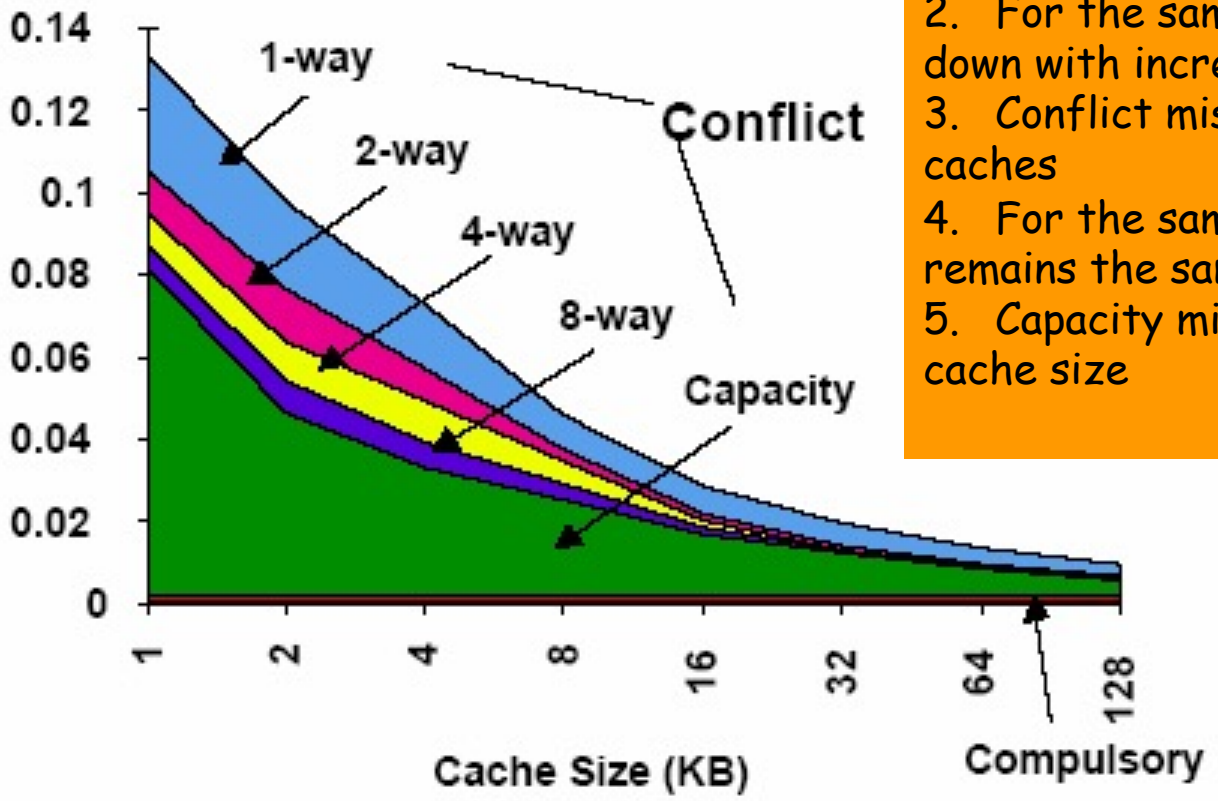
- Fully Associative Cache (32-Byte block size)
  - compare tags in parallel



No Conflict Miss (since data can go anywhere)

Cache Performance

Identical block size



Observations:

1. Cold/compulsory miss remains the same irrespective of cache size/associativity
2. For the same cache size, conflict miss goes down with increasing associativity
3. Conflict miss is zero for fully associative caches
4. For the same cache size, capacity miss remains the same irrespective of associativity
5. Capacity miss decreases with increasing cache size

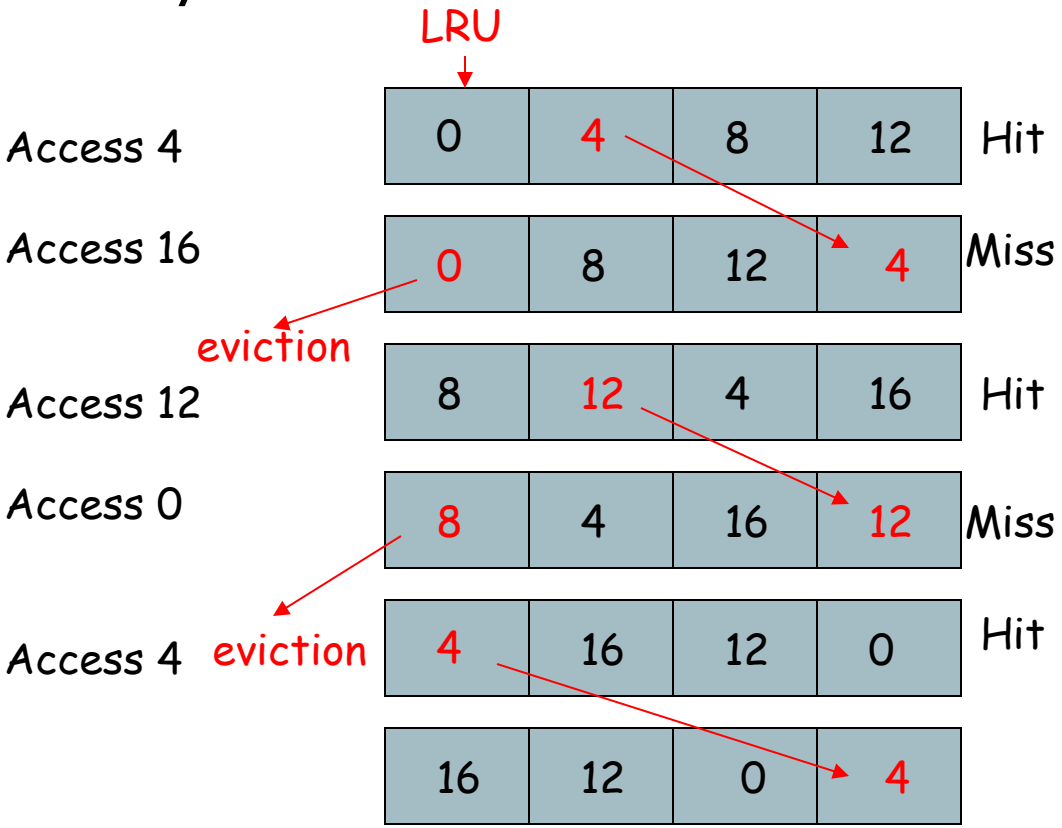
Total Miss = Cold miss + Conflict miss + Capacity miss
Capacity miss (FA) = Total miss (FA) – Cold miss (FA)

Block Replacement Policy (1/3)

- n-way Set Associative or Fully Associative have choices regarding where to place a block (which block to replace)
  - Of course, if there is an invalid block, use it
- Whenever get a cache hit, record the cache block that was touched
- When need to evict a cache block, choose one which hasn't been touched recently: Least Recently Used (LRU)
  - Past is prologue: history suggests it is least likely of the choices to be used soon
  - Why? Because of temporal locality

Block Replacement Policy (2/3)

- LRU policy in action
- Consider the following memory block accesses for a 4-way cache set: 0 4 8 12 4 16 12 0 4



Block Replacement Policy: (3/3)

- Sometimes hard to keep track of LRU if lots of choices
- Second Choice Policy: pick one at random and replace that block
- Advantages
  - Very simple to implement
- Disadvantage
  - May not match with temporal locality

Cache Framework (1/2)

- Block Placement: Where can a block be placed in cache?
  - Direct mapped: one block defined by index
  - n-way set-associative: any one of the n blocks within the set defined by index
  - Fully associative: any cache block
- Block Identification: How is a block found if it is in the cache?
  - Direct mapped: Tag match with only one block
  - n-way set associative: Tag match for all the blocks within the set
  - Fully associative: Tag match for all the blocks within the cache

Cache Framework (2/2)

- Block Replacement: Which block should be replaced on a cache miss?
  - Direct mapped: No choice
  - n-way set associative: LRU or random
  - Fully associative: LRU or random
- Write Strategy: What happens on a write?
  - Write-through vs. write-back
  - Write allocate vs. write no allocate

Improving Miss Penalty (1/2)

Average access time =
Hit rate x Hit Time + (1-Hit rate) x Miss penalty

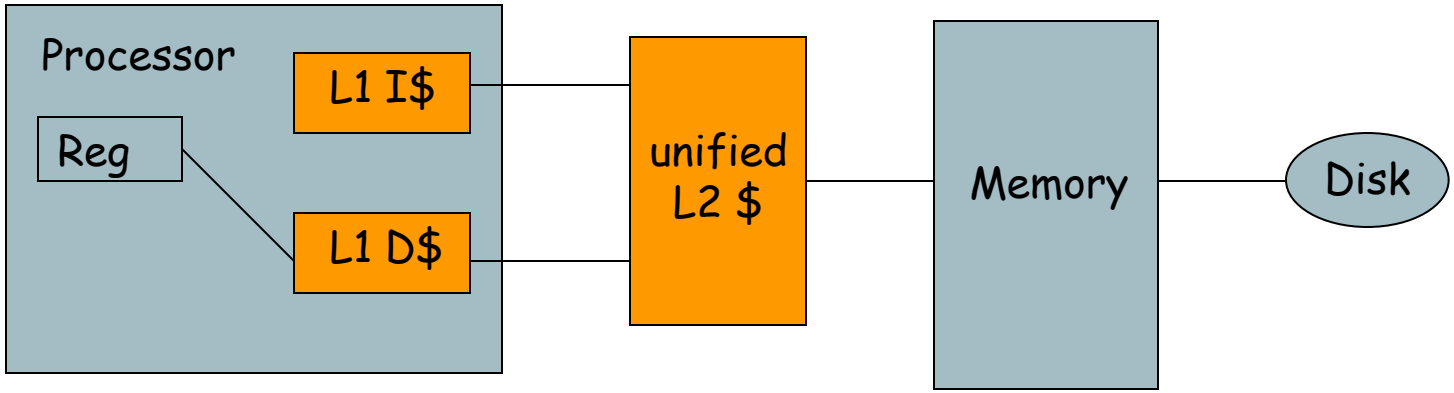
- So far, we tried to improve Miss Rate:
  - Larger block size
  - Larger Cache
  - Higher Associativity
- What about Miss Penalty?

Improving Miss Penalty (2/2)

- When caches started becoming popular, Miss Penalty was about 10 processor clock cycles
- Today 1 GHz Processor
  (1 ns per clock cycle) and 50 ns to go to DRAM
  $\Rightarrow 50$ processor clock cycles!
- Solution: Place another cache between memory and the processor cache:
  Second Level (L2) Cache

Multi-Level Caches

- Options: separate data and instruction caches or a unified cache
- Sample sizes:
  - L1: 32KB, 32-byte block, 4-way set associative
  - L2: 256KB, 128-byte block, 8-way associative
  - L3: 4MB, 256-byte block, Direct mapped



Access Time with Multi-Level Caches

- Access time = L1 hit time * L1 hit rate + L1 miss penalty * L1 miss rate
- We simply calculate the L1 miss penalty as being the access time for the L2 cache
- Access time = L1 hit time * L1 hit rate + (L2 hit time * L2 hit rate + L2 miss penalty * L2 miss rate) * L1 miss rate

Do the numbers for L2 Cache

- Assumptions:
  - L1 hit time = 1 cycle, L1 hit rate = 90%
  - L2 hit time (also L1 miss penalty) = 4 cycles,
    L2 miss penalty= 100 cycles,
    L2 hit rate = 90%
- Access time = L1 hit time * L1 hit rate +
  (L2 hit time * L2 hit rate +
  L2 miss penalty * (1 - L2 hit rate))
  * L1 miss rate

  = 1*0.9 + (4*0.9 + 100*0.1) *(1-0.9)

  = 0.9 + (13.6) * 0.1 = 2.26 clock cycles

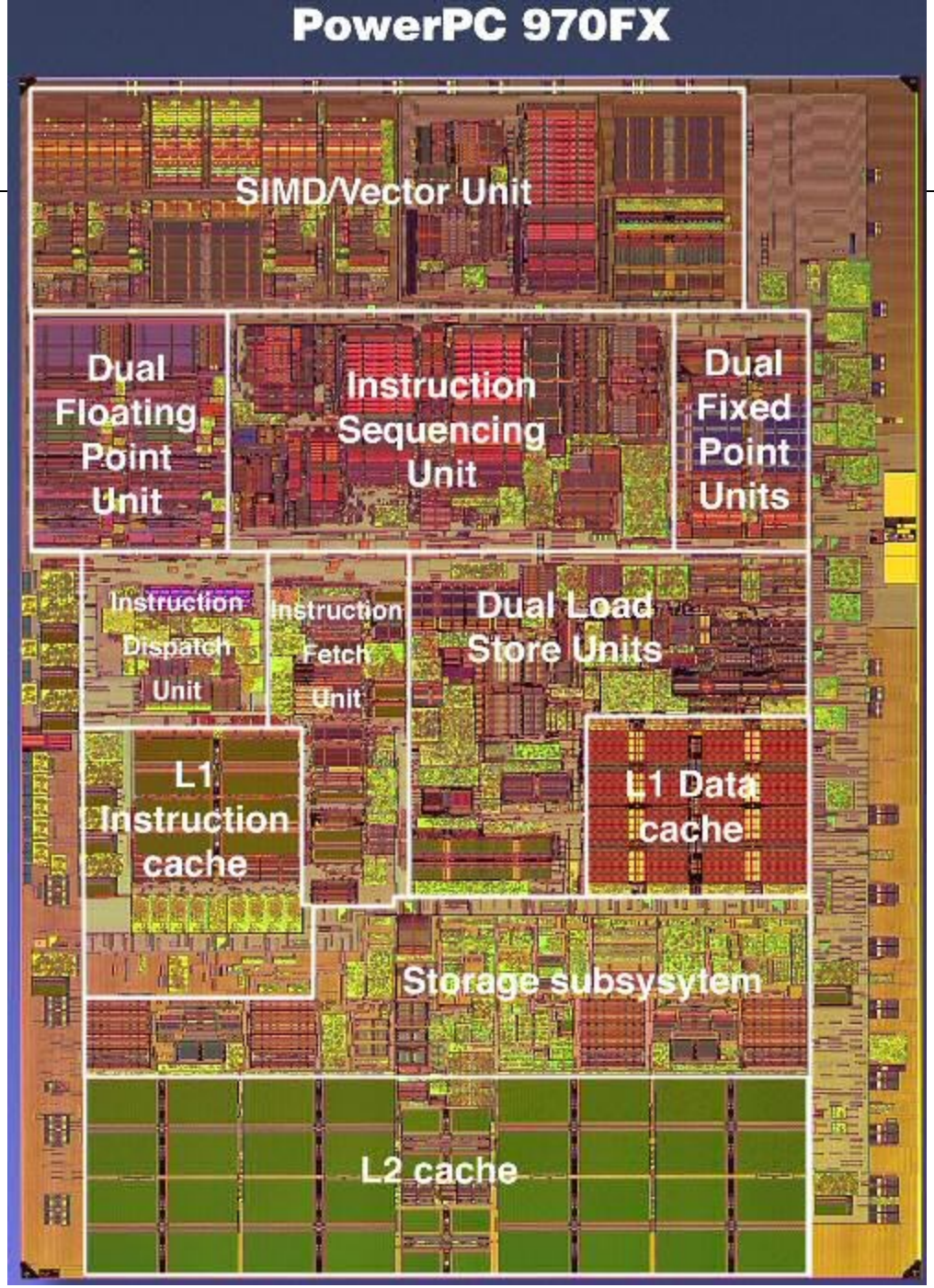
What would it be without the L2 cache?

- Assume that the L1 miss penalty would be 100 clock cycles
- 1 *0.9 + (100)* 0.1
- 10.9 clock cycles vs. 2.26 with L2
- So gain a benefit from having the second, larger cache before main memory

Memory Hierarchy: Apple iMac G5

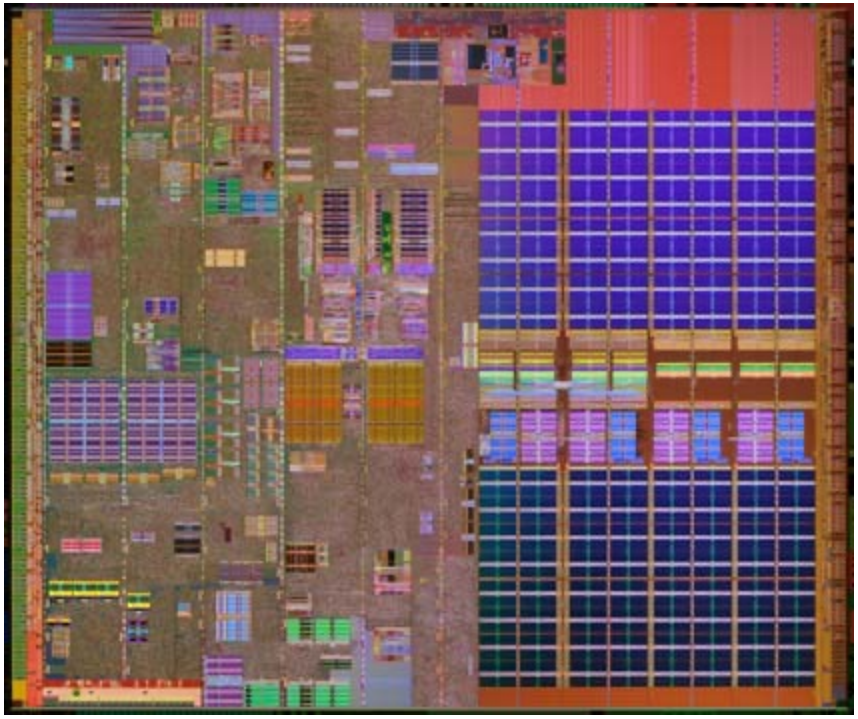
	Reg	L1 I$	L1 D$	L2	DRAM	Disk
Size	1K	64K	32K	512K	256M	80G
Latency (cycles)	1	3	3	11	88	1+E7

iMac G5 1.6 GHz

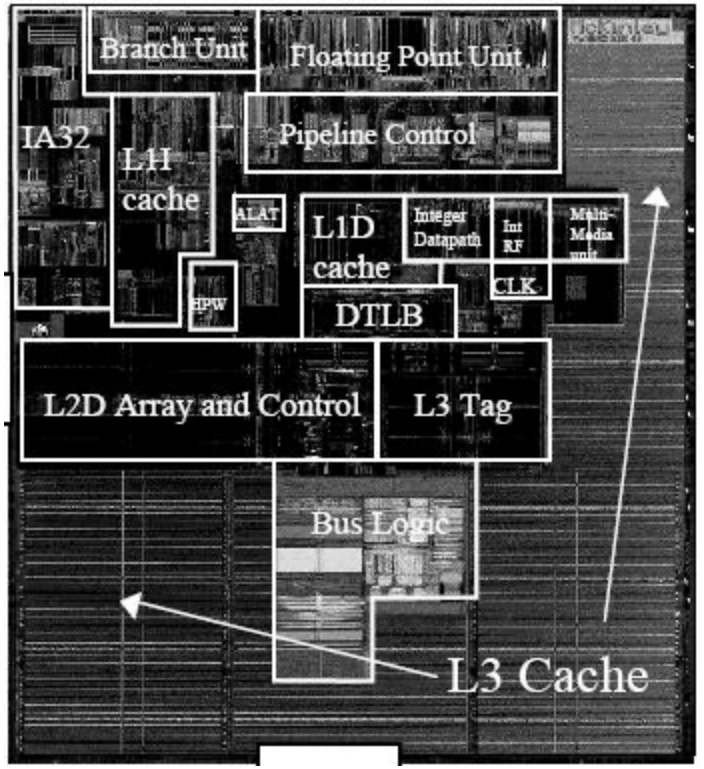


iMac's PowerPC 970FX
All caches on chip

We cannot forget Intel ☺

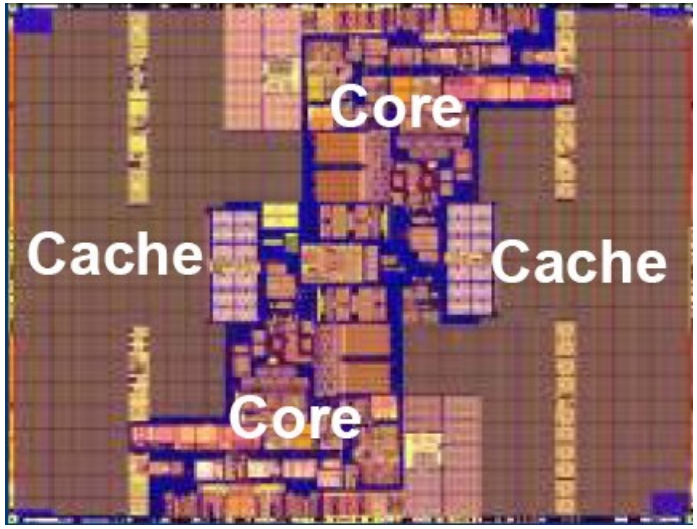


Pentium 4 Extreme Edition
L1: 12KB I$ + 8KB D$
L2: 256KB
L3: 2MB



Itanium 2 McKinley
L1: 16KB I$ + 16KB D$
L2: 256KB
L3: 1.5MB – 9MB

Where are we going?



Intel Itanium Montecito

1.72 billion transistors per die:

core logic — 57M

core caches — 106.5M

24 MB L3 cache — 1550M

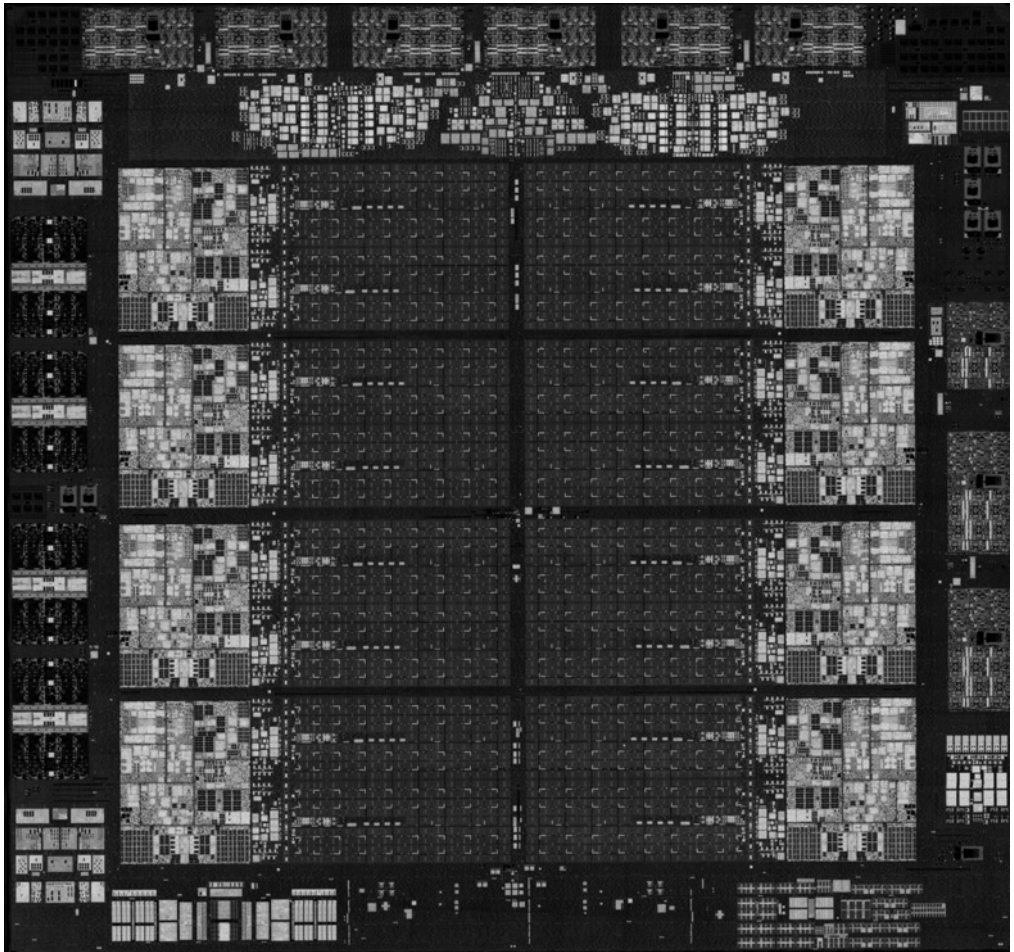
bus logic & I/O — 6.7M

HARDWARE | HOT CHIPS

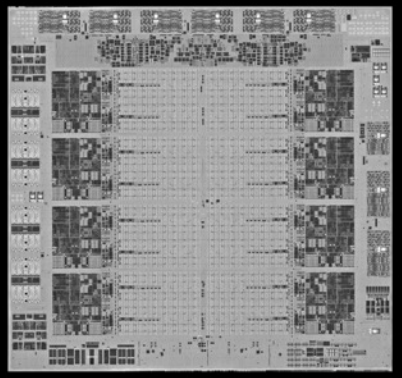
IBM's Next-Gen Z Processor Detailed: Telum Chip Based on 7nm Process, 22.5 Billion Transistors, 8 Cores Running Beyond 5 GHz Clocks

By Hassan Mujtaba

Aug 23, 2021 10:03 EDT

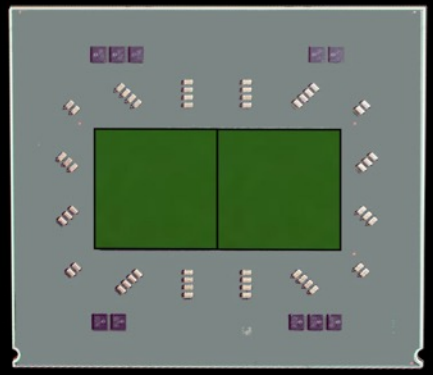


Building large scale systems: connecting up to 32 chips



Single Chip

1 chip
256MB cache

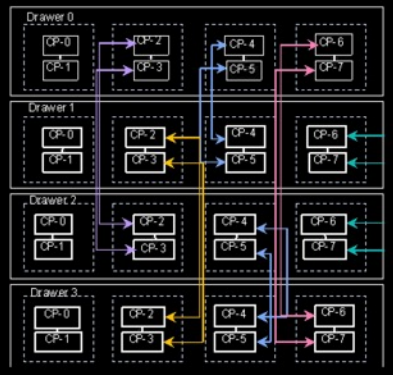


Dual Chip Module

2 chips
512MB cache

4-Socket Drawer

8 chips
2GB cache



4-drawer system

32 chips
8GB cache

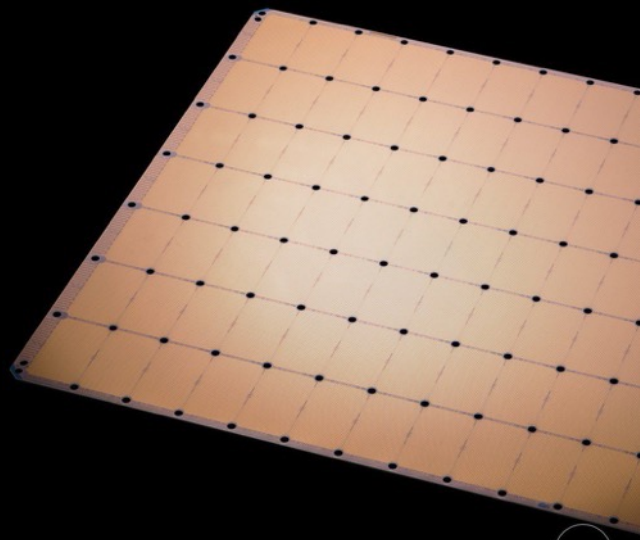
The world's largest chip

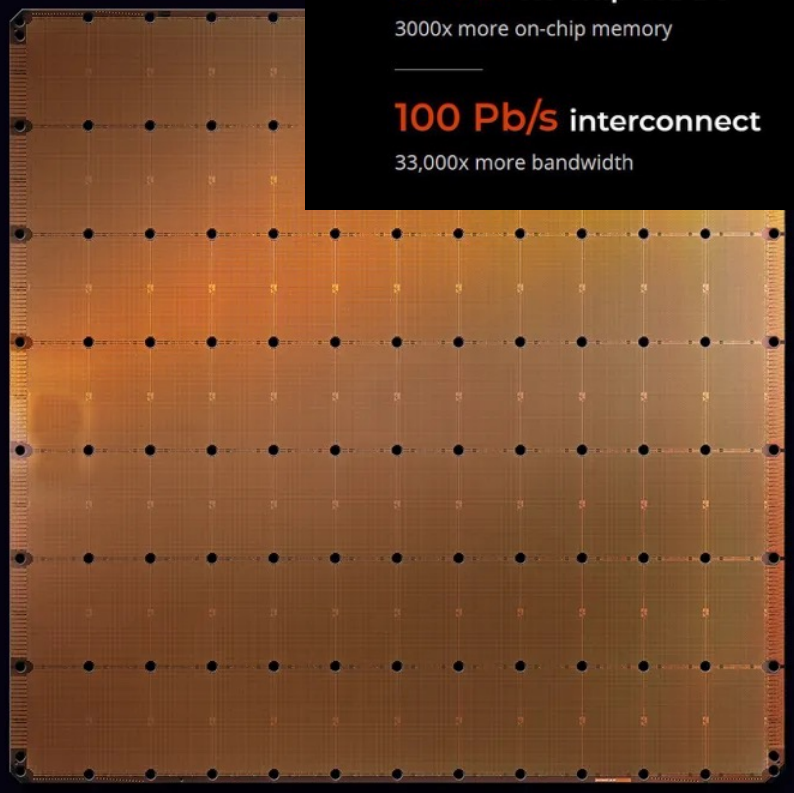
46,225 mm² chip
56x larger than the biggest GPU ever made

400,000 cores
78x more cores

18 GB on-chip SRAM
3000x more on-chip memory

100 Pb/s interconnect
33,000x more bandwidth





Cerebras WSE
1.2 Trillion transistors
46,225 mm² silicon

Largest GPU
21.1 Billion transistors
815 mm² silicon