
EECE 321: Computer Organization

Mohammad M. Mansour

Dept. of Electrical and Compute Engineering

American University of Beirut

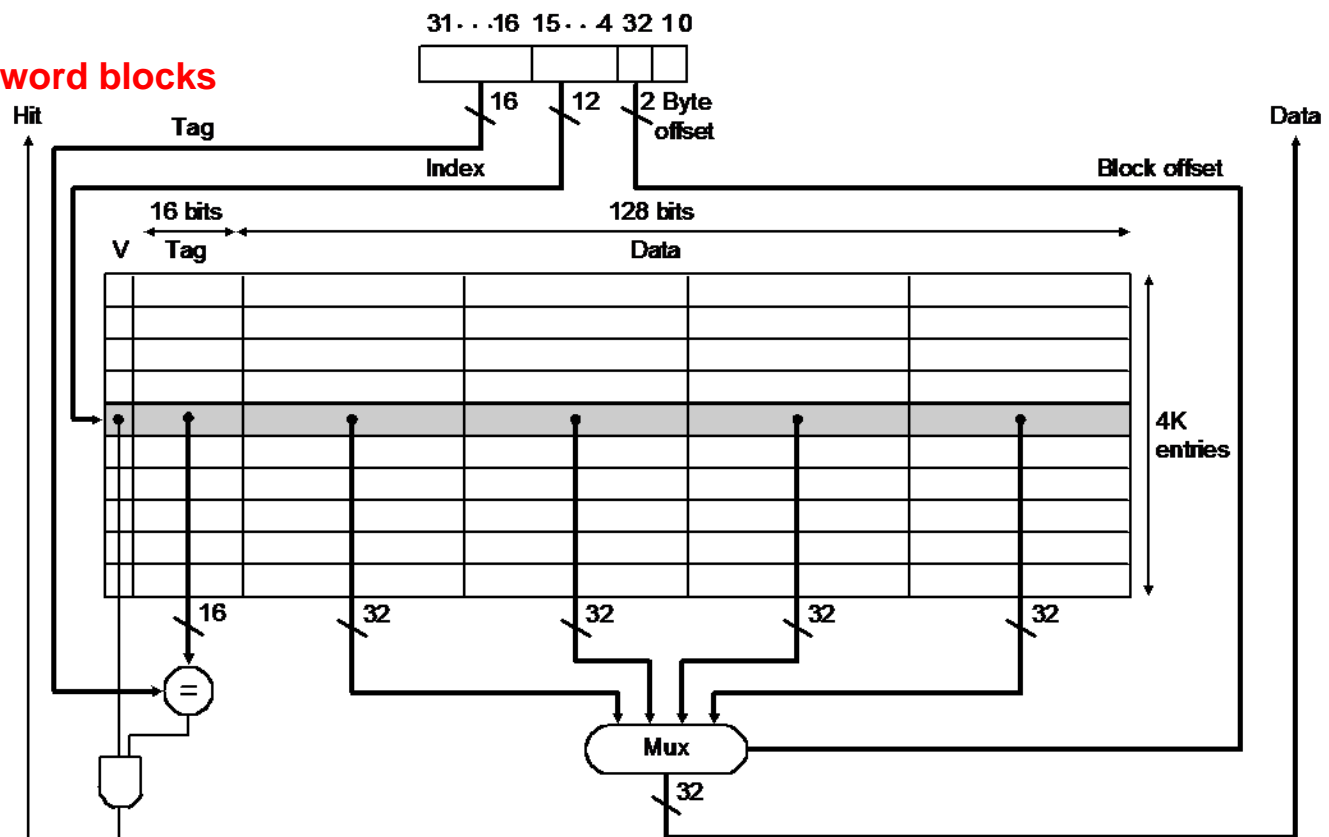
Lecture 31: Caches

Multi-word Block Caches

Taking Advantage of Spatial Locality: Multiword Cache Blocks

- The cache on the DEC machine has a block size of 1 word. It does not take advantage of spatial locality in requests.
 - Example: After fetching the instruction from I-cache at PC now, most probably we are going to fetch instruction at PC+4.
 - So why not make the block size more than one word, enabling fetching of multiple words.

EX: 64KB cache, 4-word blocks



Accessing Data in a Direct Mapped Cache

- Ex: Cache with 16KB of data, direct-mapped, 4-word blocks.
- Need to access four addresses from memory into cache:
 - 0x00000014
 - 0x0000001C
 - 0x00000034
 - 0x00008014

- Divide addresses into fields:

00000000000000000000	0000000001	0100
00000000000000000000	0000000001	1100
00000000000000000000	0000000011	0100
00000000000000000010	0000000001	0100
Tag	Index	Offset

Memory	
Address (hex)	Values of Word
...	...
00000010	A
<u>00000014</u>	B
00000018	C
<u>0000001C</u>	D
...	...
00000030	E
<u>00000034</u>	F
00000038	G
0000003C	H
...	...
00008010	I
<u>00008014</u>	J
00008018	K
0000801C	L
...	...

Accessing data in a direct mapped cache

- Cache organization: 4-word blocks, each word 4 bytes

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

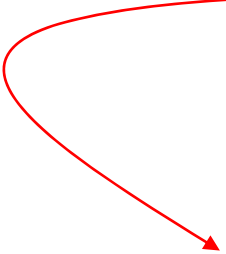
1. Read 0x00000014

000000000000000000000000 00000000001 0100

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

So we read block 1 (0000000001)

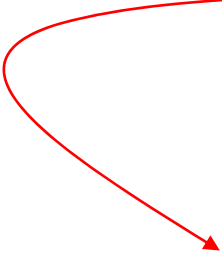
000000000000000000000000 0000000001 0100



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

No Valid Data


000000000000000000000000 0000000001 0100



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

So Load Corresponding Block into Cache, Setting Tag, Valid

00000000000000000000 0000000001 0100



Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
<u>1</u>	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					

Read from cache at offset, Return Word B

000000000000000000000000 00000000001 0100

Index	Valid	Tag	0x0-3	<u>0x4-7</u>	0x8-b	0xc-f
0	0					
<u>1</u>	1	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					


2. Read 0x0000001C = 0...00 0..001 1100

000000000000000000000000 0000000001 1100

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

Index is Valid

00000000000000000000 0000000001 1100



Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
<u>1</u>	<u>1</u>	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					

Index valid, Tag Matches

00000000000000000000 00000000001 1100

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
<u>1</u>	<u>1</u>	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					

Index Valid, Tag Matches, Return D

00000000000000000000 00000000001 1100

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
<u>1</u>	<u>1</u>	0	A	B	C	D
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					


3. Read 0x00000034 = 0...00 0..011 0100

000000000000000000000000 0000000011 0100

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

So Read Block 3

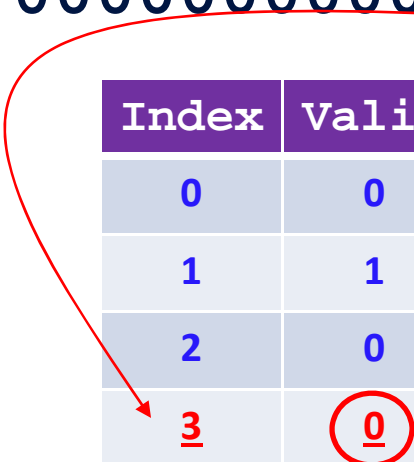
00000000000000000000 0000000011 0100



Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	A	B	C	D
2	0					
<u>3</u>	0					
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					

No Valid Data

000000000000000000000000 0000000011 0100



Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	A	B	C	D
2	0					
<u>3</u>	<u>0</u>					
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					

Load that Cache Block, Return Word F

00000000000000000000 0000000011 0100

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	A	B	C	D
2	0					
<u>3</u>	<u>1</u>	0	E	F	G	h
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					


4. Read 0x00008014 = 0...10 0..001 0100

0000000000000000000010 0000000001 0100

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	A	B	C	D
2	0					
3	1	0	E	F	G	H
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					

So read Cache Block 1, Data is Valid

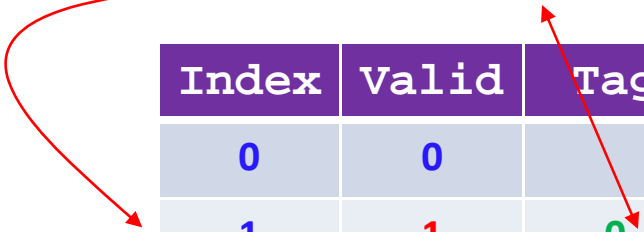
0000000000000000000010 0000000001 0100



Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	<u>1</u>	0	A	B	C	D
2	0					
3	1	0	E	F	G	H
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					

Cache Block 1 Tag does not match (0 != 2)

00000000000000000010 00000000001 0100



Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	<u>1</u>	0	A	B	C	D
2	0					
3	1	0	E	F	G	H
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					

Miss, so replace block 1 with new data & tag

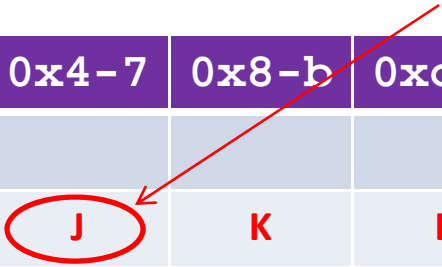
00000000000000000010 00000000001 0100

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	<u>1</u>	2	I	J	K	L
2	0					
3	1	0	E	F	G	H
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					

And return word J

00000000000000000010 00000000001 0100

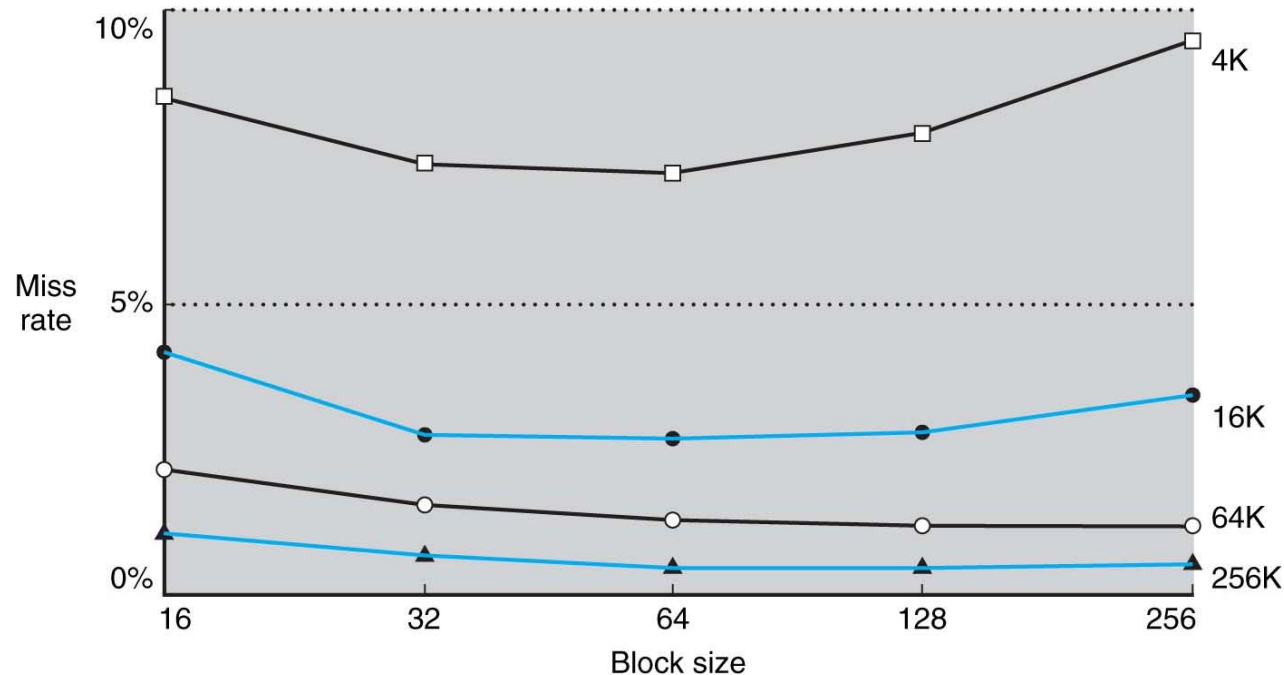
Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	<u>1</u>	2	I	J	K	L
2	0					
3	1	0	E	F	G	H
4	0					
5	0					
6	0					
7	0					
...	.					
1022	0					
1023	0					



Cache Misses in Multiword Block Caches

- Read misses in caches with multiword blocks are handled the same way for 1-word blocks.
 - A miss always brings an entire block
- Write hits and write misses are handled differently.
- Because a block contains more than one word, can't just write tag and data.
- For a write-through cache:
 - Write the data into cache while performing tag comparison.
 - If the tag of the address matches the tag in the cache entry, there is a write-hit.
 - Otherwise, we have a write-miss and must fetch a block from memory. After the block is fetched and placed into the cache, we can rewrite the word that caused the miss into the cache block.
 - Hence, unlike the case with a 1-word block, write misses with a multiword block require reading from memory.
- How far can we keep increasing the block size?
 - Impact on performance: In general, **the miss rate falls when block size increases**
 - Large blocks suit better I-cache due to higher spatial locality
 - But larger block sizes, means less blocks per cache, and there will be more competition for those blocks. Hence, miss rate will go up when block size becomes larger and larger.

Miss Rate Versus Block Size



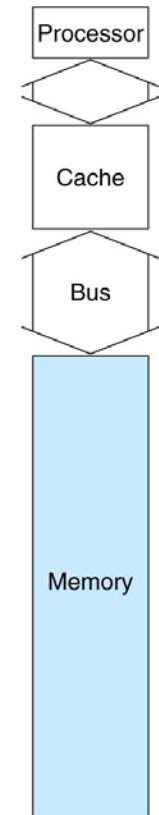
- A more serious problem with increasing block size is that the cost of a miss increases, i.e., **miss penalty increases**.
- The larger the block size, the more time it takes to transfer data from memory to cache, and the more clock cycle the processor has to wait to get its data.
- Result is that the increase in miss penalty offsets the advantages of decreasing miss rate:
- Remember: $\text{Avg. Access} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$

Designing the Memory System to Support Caches

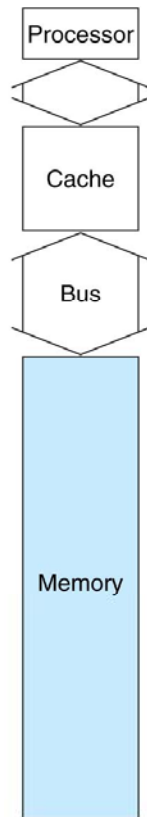
- As cache block size increases, **miss rate** first decreases, then starts to increase
 - But more importantly, the **miss penalty** increases
- How can the memory system be organized to reduce miss penalty for large blocks?
- Example: We have a memory system that has the following memory access times:
 - 1 clock cycle to send address
 - 15 clock cycles for each DRAM access initiated
 - 1 clock cycle to send a word of data
- If we have a cache block of 4 words, one-word-wide interface to processor, and a one-word-wide bank of DRAMs, what is the miss penalty? Assume there is a single bus between cache and memory to send data and addresses on.

Miss Penalty = DRAM Access Time

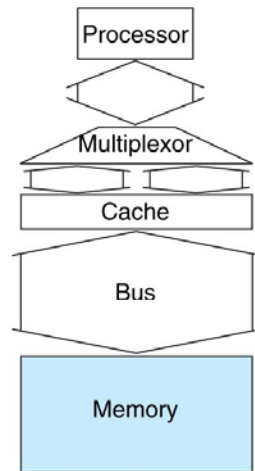
$$\begin{aligned} &= \text{Time to send address} + \\ &\quad + (\text{time to retrieve } \text{word1} + \text{time to send } \text{word1}) \\ &\quad + (\text{time to retrieve } \text{word2} + \text{time to send } \text{word2}) \\ &\quad + (\text{time to retrieve } \text{word3} + \text{time to send } \text{word3}) \\ &\quad + (\text{time to retrieve } \text{word4} + \text{time to send } \text{word4}) \\ &= 1 + (15 + 1) + (15 + 1) + (15 + 1) + (15 + 1) \\ &= 65 \text{ clock cycles} \end{aligned}$$



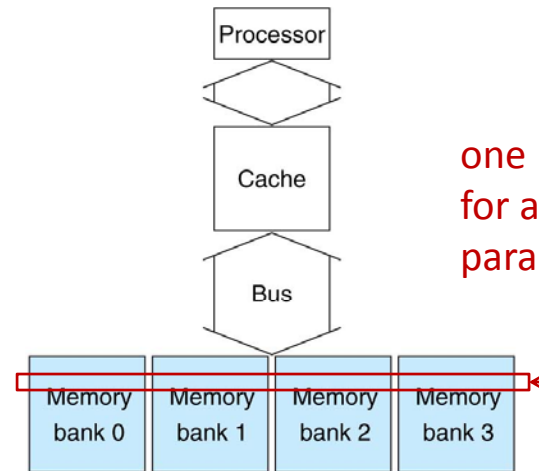
Wide-Memory vs. Interleaved Memory



a. One-word-wide memory organization



b. Wider memory organization



c. Interleaved memory organization

one DRAM access
for all 4 banks in
parallel

- Miss penalty 1 = $1 + (15 + 1) + (15 + 1) + (15 + 1) + (15 + 1) = 65$ clock cycles
- Miss penalty 2 = $1 + (15 + 1) = 17$ clock cycles
- Miss penalty 3 = $1 + (15) + 1 + 1 + 1 + 1 = 20$ clock cycles

Cache Performance

- Cache performance is proportional to *miss rate x miss penalty*.
- Focus on techniques that reduce both factors:
 1. New block placement policies that reduce miss rate
 2. Multi-level caches to reduce miss penalty
- Memory stall clock cycles = Memory Inst. Per program x Miss Rate x Miss Penalty.
- Example: Impact of cache performance on machine performance
 - Assume I-cache miss rate for GCC is 2%
 - D-cache miss rate is 4%
 - Assume CPI on a perfect cache is 2
 - 36% of instruction in GCC are memory instructions
 - Assume miss penalty is 40 clock cycles
 - Compute actual CPI including cache misses
- Answer: $CPI_{\text{misses}} = 2 + (2\% \times 40) + (36\% \times 4\% \times 40) = 2 + 0.8 + 0.56 = 3.36$
 - Percentage cycles on misses is $1.36/3.36 = 41\%$

Cache Performance

- What happens to CPI if the processor is made faster, by doubling its clock rate?
 - Main memory speed is unlikely to change, so miss penalty becomes 80 new “CPU” clock cycles
 - $\text{CPI}_{\text{new}} = 2 + (2\% \times 80) + (36\% \times 4\% \times 80) = 4.75$
 - % cycles on misses is $2.75/4.75 \approx 58\%$
- How would cache misses impact a processor’s performance if it is made faster by lowering its CPI from 2 to 1 (without stalls)?
 - $\text{CPI}_{\text{misses}} = 1 + (2\% \times 40) + (36\% \times 4\% \times 40) = 2.36$
 - % cycles on misses is $1.36/2.36 \approx 58\%$
- Conclusion: If a machine improves both clock rate and CPI, the more pronounced the impact of stall cycles on machine performance becomes.

Types of Cache Misses: The “Three Cs” Model

- “Three Cs” Model of Misses:
 - Compulsory Misses, Conflict Misses, Capacity Misses
- 1st C: Compulsory Misses (aka ‘Cold Start Misses’)
 - Occur when a program is first started
 - Cache does not contain any of that program’s data yet, so misses are bound to occur
 - Can’t be avoided easily, so won’t focus on these in this course
- 2nd C: Conflict Misses
 - Miss that occurs because 2 distinct memory addresses map to the same cache location
 - 2 blocks (which happen to map to the same location) can keep overwriting each other
 - Big problem in direct-mapped caches
- Capacity Misses
 - Miss that occurs because the cache has a limited size
 - Miss that would not occur if we increase the size of the cache
- Dealing with Conflict Misses
 - Solution 1: Make the cache size bigger
 - Fails at some point
 - Solution 2: Multiple distinct blocks can fit in the same cache Index?
 - **Associative Caches**

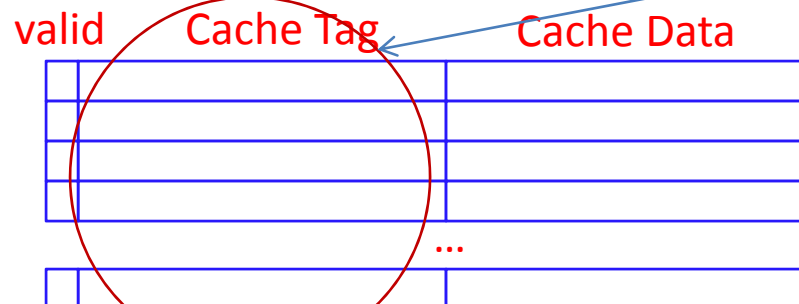
Fully Associative Cache

- Idea: Any block can go anywhere in the cache
- What about the index field in the address? Answer: It does not exist.
- Memory address fields:
 - Tag: same as before
 - Offset: same as before
 - Index: non-existent
- Benefit of Fully Associative Cache
 - No Conflict Misses (since data can go anywhere)
- Drawbacks of Fully Associative Cache
 - Need hardware comparator for every single entry: if we have a 64KB of data in cache with 4B entries, we need 16K comparators: infeasible

Memory address:



Associative
Cache
Organization



All tags must be
considered in
comparison
NO INDEX

Reducing Miss Rate by a More Flexible Block Placement Policy

- N-way Set-Associative Cache:
 - Divide cache into sets
 - A set contains N-blocks instead of one
 - A block from memory maps to a unique set in cache given by the index field, but can be placed in **any** block location in that set.
- Mapping: (Block Number) modulo (Number of sets in cache)
- A direct-map

**One-way set associative
(direct mapped)**

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

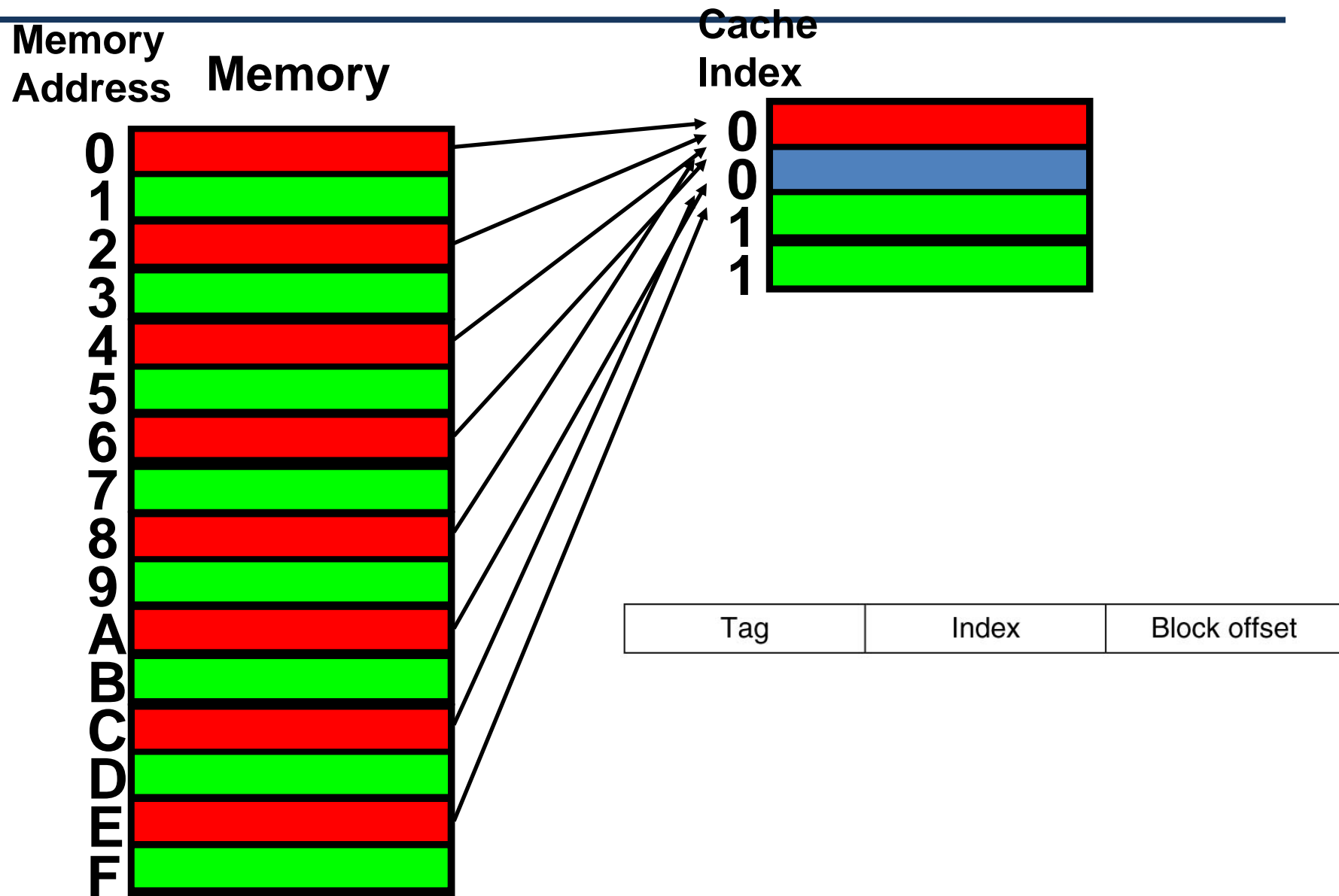
Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

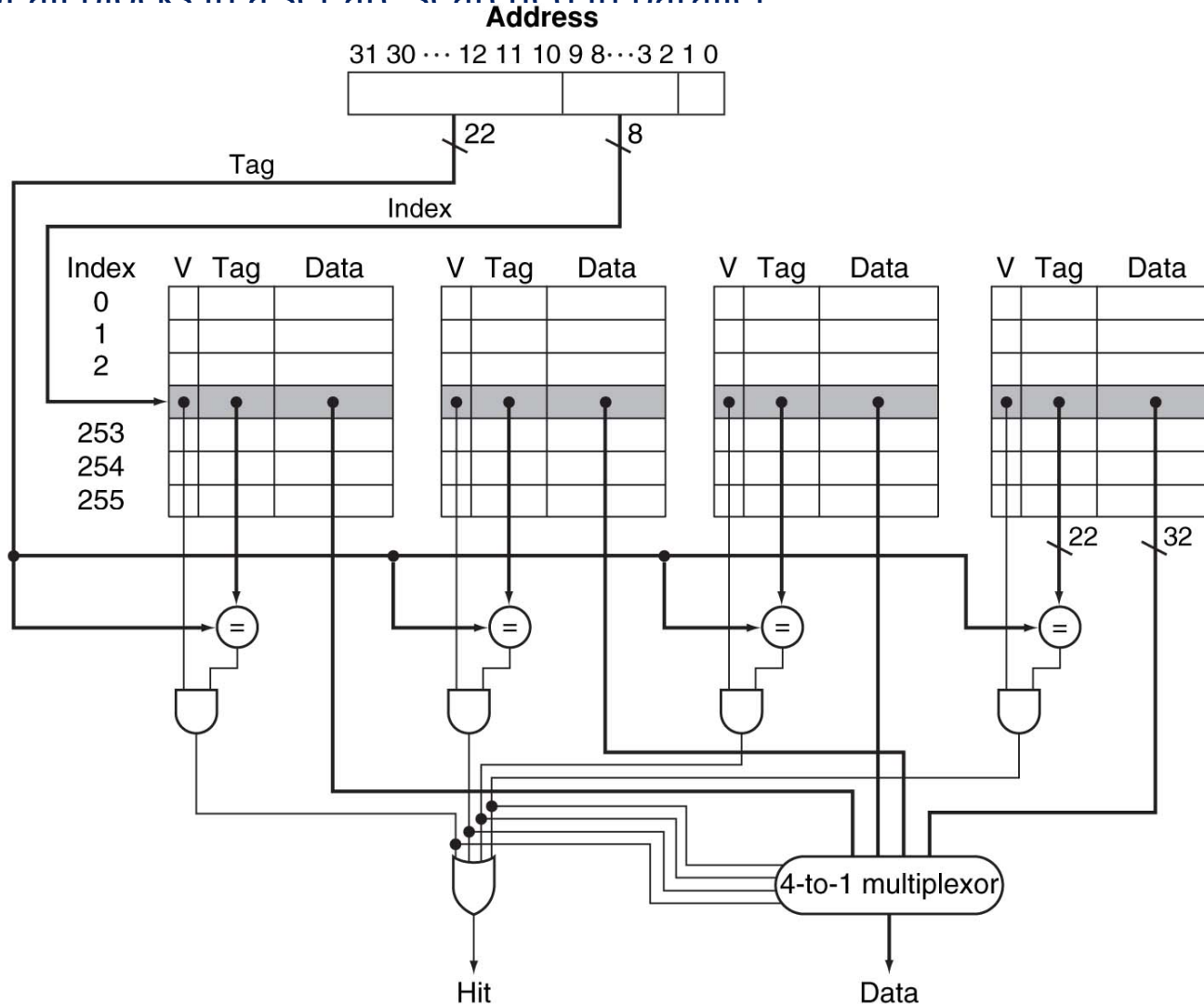
Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

A 2-Way Set Associative Cache Example



Locating a Block

- Tags of all blocks in a set are searched in parallel



Block Replacement Policy

- If a cache block can map to any of the N-blocks in an N-way set associative cache, which one to choose to replace?
- In a direct-mapped cache, there isn't much of a choice; only one possibility.
- The most commonly used scheme is the **least recently used (LRU)** block replacement policy.
- In an LRU policy, the block that has been used for the longest time is replaced.
 - Tracking the use of the blocks is implemented by adding extra bits to the blocks to record history of block access.
 - As associativity increases, LRU policy becomes harder to implement.
- Typically, for large associativity **random block** replacement is used.
- Example: We have a 2-way set associative cache with a four word total capacity and one word blocks. We perform the following word accesses (ignore bytes for this problem):

0, 2, 0, 1, 4, 0, 2, 3, 5, 4
- How many hits and how many misses will there be for the LRU block replacement policy?

Block Replacement Example: LRU

- Addresses 0, 2, 0, 1, 4, 0, ...

0: miss, bring into set 0 (loc 0)

2: miss, bring into set 0 (loc 1)

0: hit

1: miss, bring into set 1 (loc 0)

4: miss, bring into set 0 (loc 1, replace 2)

0: hit

loc 0 loc 1

set 0

iru

0

set 1

set 0

iru

0

2

set 1

set 0

iru

0

2

set 1

set 0

iru

0

2

set 1

iru

1

set 0

iru

0

4

set 1

iru

1

set 0

iru

0

4

set 1

iru

1