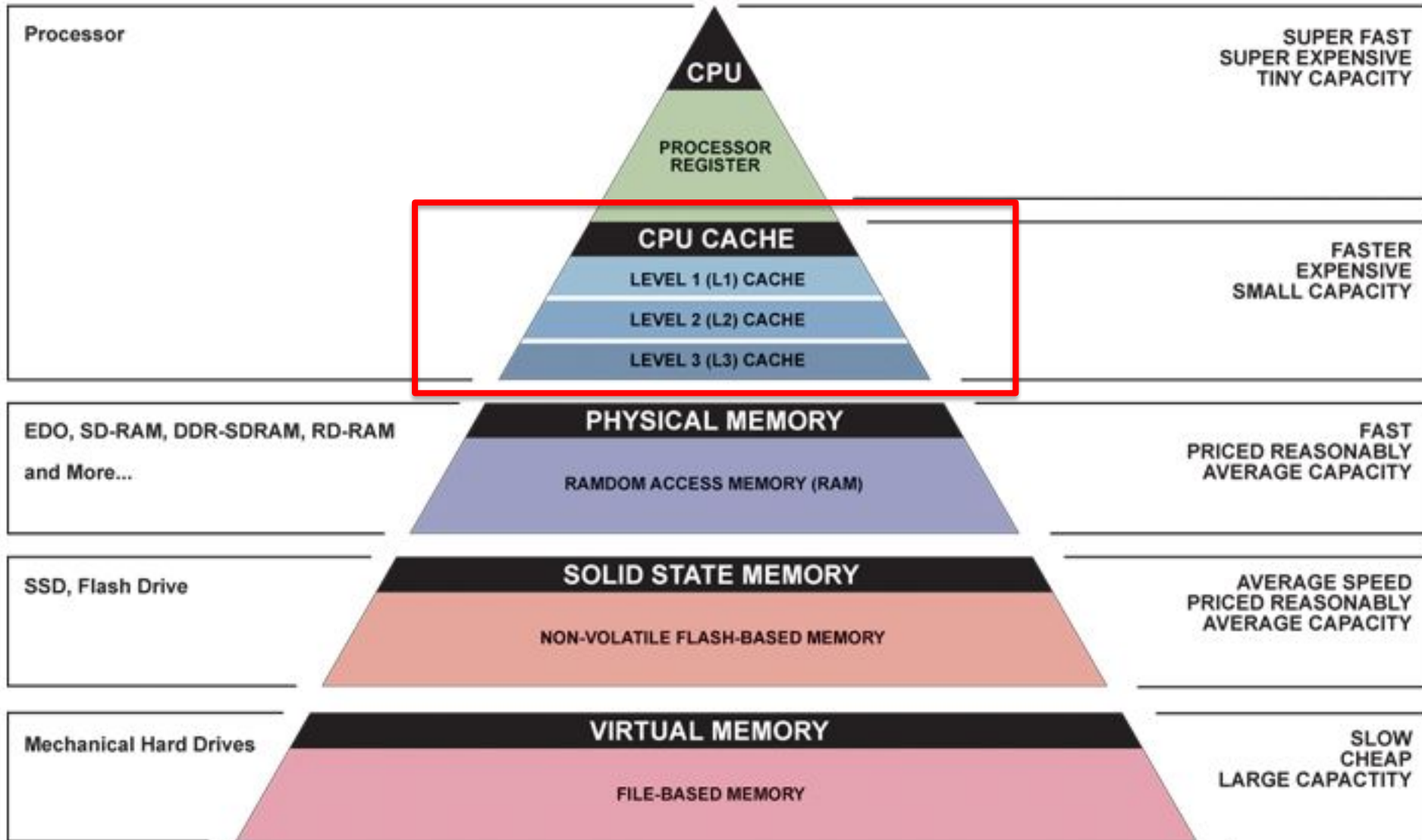


# Great Ideas in Computer Architecture

## *Direct-Mapped and Set Associative Caches*

Instructor: Steven Ho

# Great Idea #3: Principle of Locality/ Memory Hierarchy



# Extended Review of Last Lecture

- Why have caches?
  - Intermediate level between CPU and memory
  - In-between in *size*, *cost*, and *speed*
- Memory (hierarchy, organization, structures) set up to exploit *temporal* and *spatial locality*
  - *Temporal*: If accessed, will access again soon
  - *Spatial*: If accessed, will access others around it
- Caches hold a subset of memory (in *blocks*)
  - We are studying how they are designed for fast and efficient operation (lookup, access, storage)

# Extended Review of Last Lecture

- Fully Associative Caches:
  - Every block can go in any slot
    - Use random or LRU replacement policy when cache full
  - Memory address breakdown (on request)
    - **Tag** field is unique identifier (which block is currently in slot)
    - **Offset** field indexes into block (by bytes)
  - *Each* cache slot holds block data, tag, valid bit, and dirty bit (dirty bit is only for *write-back*)
    - The whole cache maintains LRU bits

# Extended Review of Last Lecture

- On memory access (read or write):
  - 1) Look at ALL cache slots in parallel
  - 2) If Valid bit is 0, then ignore (garbage)
  - 3) If Valid bit is 1 and **Tag** matches, then use that data
- On write, set Dirty bit if write-back

# Extended Review of Last Lecture

$2^6 = 64$  B address space

cache size (C)

block size (K)

- Fully associative cache layout in our example
  - 6-bit address space, 16-byte cache with 4-byte blocks
  - How many blocks do we have?  $C/K = 4$  blocks
  - LRU replacement (2 bits)
  - Offset – 2 bits, Tag – 4 bits

LRU bits

Offset

	V	Tag	00	01	10	11	LRU
Slot 0	X	XXXX	0x??	0x??	0x??	0x??	XX
1	X	XXXX	0x??	0x??	0x??	0x??	XX
2	X	XXXX	0x??	0x??	0x??	0x??	XX
3	X	XXXX	0x??	0x??	0x??	0x??	XX

Yesterday's example was write through and looked like this

# FA Cache Examples (3/4)

1) Consider the following *addresses* being requested:

Starting with a cold cache:      0   2   2   0   16   20   8   4

**000000**  
**0 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

**000010**  
**2 hit**

1	0000	M[0]	M[1]	M[2]	M[3]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

**000010**  
**2 hit**

1	0000	M[0]	M[1]	M[2]	M[3]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

**000000**  
**0 hit**

1	0000	M[0]	M[1]	M[2]	M[3]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

# FA Cache Examples (3/4)

1) Consider the following *addresses* being requested:

Starting with a cold cache:

0    2    2    0    16    20    8    4  
**M**   **H**   **H**   **H**

**010000**  
**16 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
1	0100	M[16]	M[17]	M[18]	M[19]
0	0000	0x??	0x??	0x??	0x??
0	0000	0x??	0x??	0x??	0x??

**001000**  
**8 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
1	0100	M[16]	M[17]	M[18]	M[19]
1	0101	M[20]	M[21]	M[22]	M[23]
1	0010	M[8]	M[9]	M[10]	M[11]

**010100**  
**20 miss**

1	0000	M[0]	M[1]	M[2]	M[3]
1	0100	M[16]	M[17]	M[18]	M[19]
1	0101	M[20]	M[21]	M[22]	M[23]
0	0000	0x??	0x??	0x??	0x??

**000100**  
**4 miss**

<del>1</del>	<del>0000</del>	<del>M[0]</del>	<del>M[1]</del>	<del>M[2]</del>	<del>M[3]</del>
1	0100	M[16]	M[17]	M[18]	M[19]
1	0101	M[20]	M[21]	M[22]	M[23]
1	0010	M[8]	M[9]	M[10]	M[11]

- 8 requests, 5 misses – ordering matters!



# FA Cache Examples (4/4)

## 3) Original sequence, but double block size to 8B

Starting with a cold cache:      0    2    4    8    20   16   0    2

000000

0

miss

1	000	M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]	M[7]
0	000	0x??	0x??	0x??	0x??	0x??	0x??	0x??	0x??

000010

2

hit

1	000	M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]	M[7]
0	000	0x??	0x??	0x??	0x??	0x??	0x??	0x??	0x??

000100

4

hit

1	000	M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]	M[7]
0	000	0x??	0x??	0x??	0x??	0x??	0x??	0x??	0x??

001000

8

miss

1	000	M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]	M[7]
1	001	M[8]	M[9]	M[10]	M[11]	M[12]	M[13]	M[14]	M[15]

# FA Cache Examples (4/4)

## 3) Original sequence, but double block size

Starting with a cold cache:

0 2 4 8 20 16 0 2  
M H H M

010100  
20  
miss

1	<del>000</del>	<del>M[0]</del>	<del>M[1]</del>	<del>M[2]</del>	<del>M[3]</del>	<del>M[4]</del>	<del>M[5]</del>	<del>M[6]</del>	<del>M[7]</del>
1	001	M[8]	M[9]	M[10]	M[11]	M[12]	M[13]	M[14]	M[15]

010000  
16  
hit

1	010	M[16]	M[17]	M[18]	M[19]	M[20]	M[21]	M[22]	M[23]
1	001	M[8]	M[9]	M[10]	M[11]	M[12]	M[13]	M[14]	M[15]

000000  
0  
miss

1	010	M[16]	M[17]	M[18]	M[19]	M[20]	M[21]	M[22]	M[23]
1	<del>001</del>	<del>M[8]</del>	<del>M[9]</del>	<del>M[10]</del>	<del>M[11]</del>	<del>M[12]</del>	<del>M[13]</del>	<del>M[14]</del>	<del>M[15]</del>

000010  
2  
hit

1	010	M[16]	M[17]	M[18]	M[19]	M[20]	M[21]	M[22]	M[23]
1	000	M[0]	M[1]	M[2]	M[3]	M[4]	M[5]	M[6]	M[7]

- 8 requests, 4 misses – cache parameters matter!

## Question:

Starting with the same cold cache as the first 3 examples, which of the sequences below will result in the final state of the cache shown here:

0	1	0000	M[0]	M[1]	M[2]	M[3]	<table border="1"> <tr> <td><b>LRU</b></td> </tr> <tr> <td>10</td> </tr> </table>	<b>LRU</b>	10
<b>LRU</b>									
10									
1	1	0011	M[12]	M[13]	M[14]	M[15]			
2	1	0001	M[4]	M[5]	M[6]	M[7]			
3	1	0100	M[16]	M[17]	M[18]	M[19]			

(A) 0 2 12 4 16 8 0 6

(B) 0 8 4 16 0 12 6 2

(C) 6 12 4 8 2 16 0 0

(D) 2 8 0 4 6 16 12 0

## Question:

Starting with the same cold cache as the first 3 examples, which of the sequences below will result in the final state of the cache shown here:

0	1	0000	M[0]	M[1]	M[2]	M[3]	<table border="1"> <tr> <td>LRU</td> </tr> <tr> <td>10</td> </tr> </table>	LRU	10
LRU									
10									
1	1	0011	M[12]	M[13]	M[14]	M[15]			
2	1	0001	M[4]	M[5]	M[6]	M[7]			
3	1	0100	M[16]	M[17]	M[18]	M[19]			

- (A) ~~0 2 12 4 16 8 0 6~~ ←
- (B) ~~0 8 4 16 0 12 6 2~~ ←
- (C) ~~6 12 4 8 2 16 0 0~~
- (D) 2 8 0 4 6 16 12 0 ✓

## Question:

Starting with the same cold cache as the first 3 examples, which of the sequences below will result in the final state of the cache shown here:

0	1	0000	M[0]	M[1]	M[2]	M[3]	<table border="1"> <tr> <td><b>LRU</b></td> </tr> <tr> <td>10</td> </tr> </table>	<b>LRU</b>	10
<b>LRU</b>									
10									
1	1	0011	M[12]	M[13]	M[14]	M[15]			
2	1	0001	M[4]	M[5]	M[6]	M[7]			
3	1	0100	M[16]	M[17]	M[18]	M[19]			

(A) 0 2 12 4 16 8 0 6

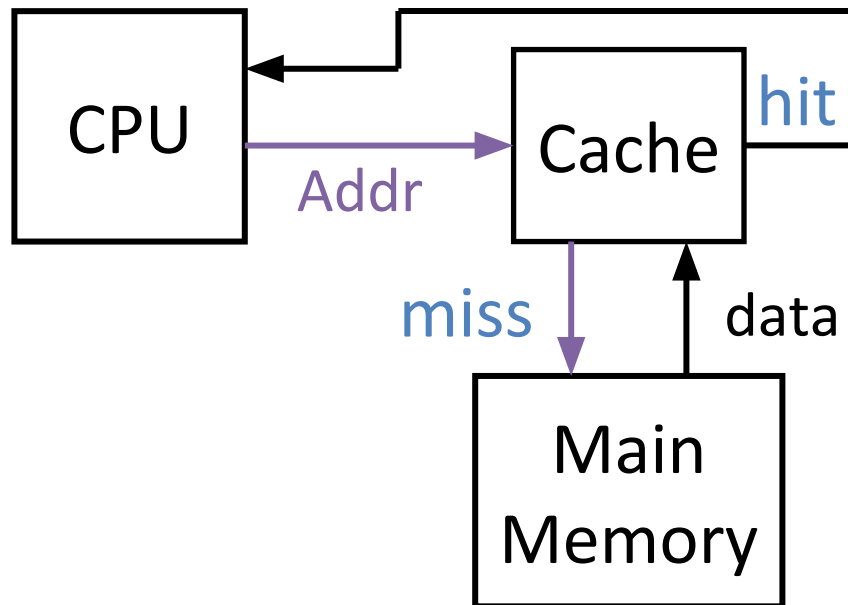
(B) 0 8 4 16 0 12 6 2

(C) 6 12 4 8 2 16 0 0


(D) 2 8 0 4 6 16 12 0

# Memory Accesses

- The picture so far:



# Handling Write Hits

- Write hits (D\$)
    - 1) **Write-Through Policy:** Always write data to cache and to memory (*through* cache)
      - Forces cache and memory to always be consistent
      - Slow! (every memory access is long)
      - Include a *Write Buffer* that updates memory in parallel with processor
- Assume present in all schemes when writing to memory
- 

# Handling Write Hits

- Write hits (D\$)
  - 2) **Write-Back Policy:** Write data only to cache, then update memory when block is removed
    - Allows cache and memory to be inconsistent
    - Multiple writes collected in cache; single write to memory per block
    - **Dirty bit:** Extra bit per cache row that is set if block was written to (is “dirty”) and needs to be written back



# Handling Cache Misses

- Miss penalty grows as block size does
- Read misses (I\$ and D\$)
  - Stall execution, fetch block from memory, put in cache, send requested data to processor, resume
- Write misses (D\$)
  - Always have to update block from memory
  - We have to make a choice:
    - Carry the updated block into cache or not?

# Write Allocate

- *Write Allocate* policy: when we bring the block into the cache after a write miss
- *No Write Allocate* policy: only change main memory after a write miss
  - *Write allocate* almost always paired with *write-back*
    - Eg: Accessing same address many times -> cache it
  - *No write allocate* typically paired with *write-through*
    - Eg: Infrequent/random writes -> don't bother caching it

# Updated Cache Picture

- Fully associative, write through
  - Same as our simplified examples from before
- Fully associative, write back

	V	D	Tag	00	01	10	11	
0	X	X	XXXX	0x??	0x??	0x??	0x??	LRU XX
Slot 1	X	X	XXXX	0x??	0x??	0x??	0x??	
2	X	X	XXXX	0x??	0x??	0x??	0x??	
3	X	X	XXXX	0x??	0x??	0x??	0x??	

- Write miss procedure (write allocate or not) only affects **behavior**, not design

# How do we use this thing?

- Nothing changes from the programmer's perspective
  - Still just issuing `lw` and `sw` instructions
- The rest is handled in hardware:
  - Checking the cache
  - Extracting the data using the offset
- Why should a programmer care?
  - Understanding cache parameters = faster programs

# Agenda

- Review of yesterday
- **Administrativia**
- Direct-Mapped Caches
- Set Associative Caches
- Cache Performance

# Administrivia

- HW3/4 Due today
- HW5 Released, due next Monday (7/23)
- Project 3 Due Friday (7/20)
  - Parties tonight @Soda 405/411 and Friday @Woz (4-6pm for both)
  - If you ask for help please diagnose problem spots
- Guerilla Session on Wed. 4-6pm **@Soda 405**
- Midterm 2 is coming up! Next Wed. in lecture
  - Covering up to Performance
  - Review Session Sunday 2-4pm @GPB 100

# Direct-Mapped Caches (1/3)

- Each memory block is mapped to *exactly one slot* in the cache (*direct-mapped*)
  - Every block has only one “home”
  - Use hash function to determine which slot
- Comparison with fully associative
  - Check just one slot for a block (faster!)
  - No replacement policy necessary
  - Access pattern may leave empty slots in cache

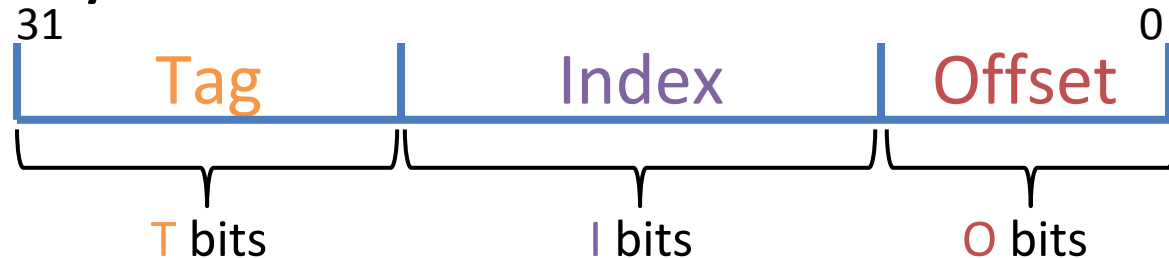
# Direct-Mapped Caches (2/3)

- **Offset field** remains the same as before
- **Recall:** blocks consist of adjacent bytes
  - Do we want adjacent blocks to map to same slot?
  - **Index field:** Apply hash function to block address to determine *which slot* the block goes in
    - $(\text{block address}) \bmod (\# \text{ of blocks in the cache})$
- **Tag field** maintains same function (identifier), but is now shorter



# TIO Address Breakdown

- Memory address fields:



- Meaning of the field sizes:
  - $O$  bits  $\leftrightarrow 2^O$  bytes/block =  $2^{O-2}$  words/block
  - $I$  bits  $\leftrightarrow 2^I$  slots in cache = cache size / block size
  - $T$  bits =  $A - I - O$ , where  $A$  = # of address bits  
( $A = 32$  here)

# Direct-Mapped Caches (3/3)

- What's actually in the cache?
  - Block of data ( $8 \times K = 8 \times 2^0$  bits)
  - Tag field of address as identifier ( $T$  bits)
  - Valid bit (1 bit)
  - Dirty bit (1 bit if write-back)
  - No replacement management bits!
- Total bits in cache = # slots  $\times$  ( $8 \times K + T + 1 + 1$ )  
 $= 2^I \times (8 \times 2^0 + T + 1 + 1)$  bits

# DM Cache Example (1/5)

- Cache parameters:

- Direct-mapped, address space of 64B, block size of 4B, cache size of 16B, write-through

- TIO Breakdown:

Memory Addresses:   
Block address

- $O = \log_2(4) = 2$

- Cache size / block size =  $16/4 = 4$ , so  $I = \log_2(4) = 2$

- $A = \log_2(64) = 6$  bits, so  $T = 6 - 2 - 2 = 2$

- Bits in cache =  $2^2 \times (8 \times 2^2 + 2 + 1) = 140$  bits

# DM Cache Example (2/5)

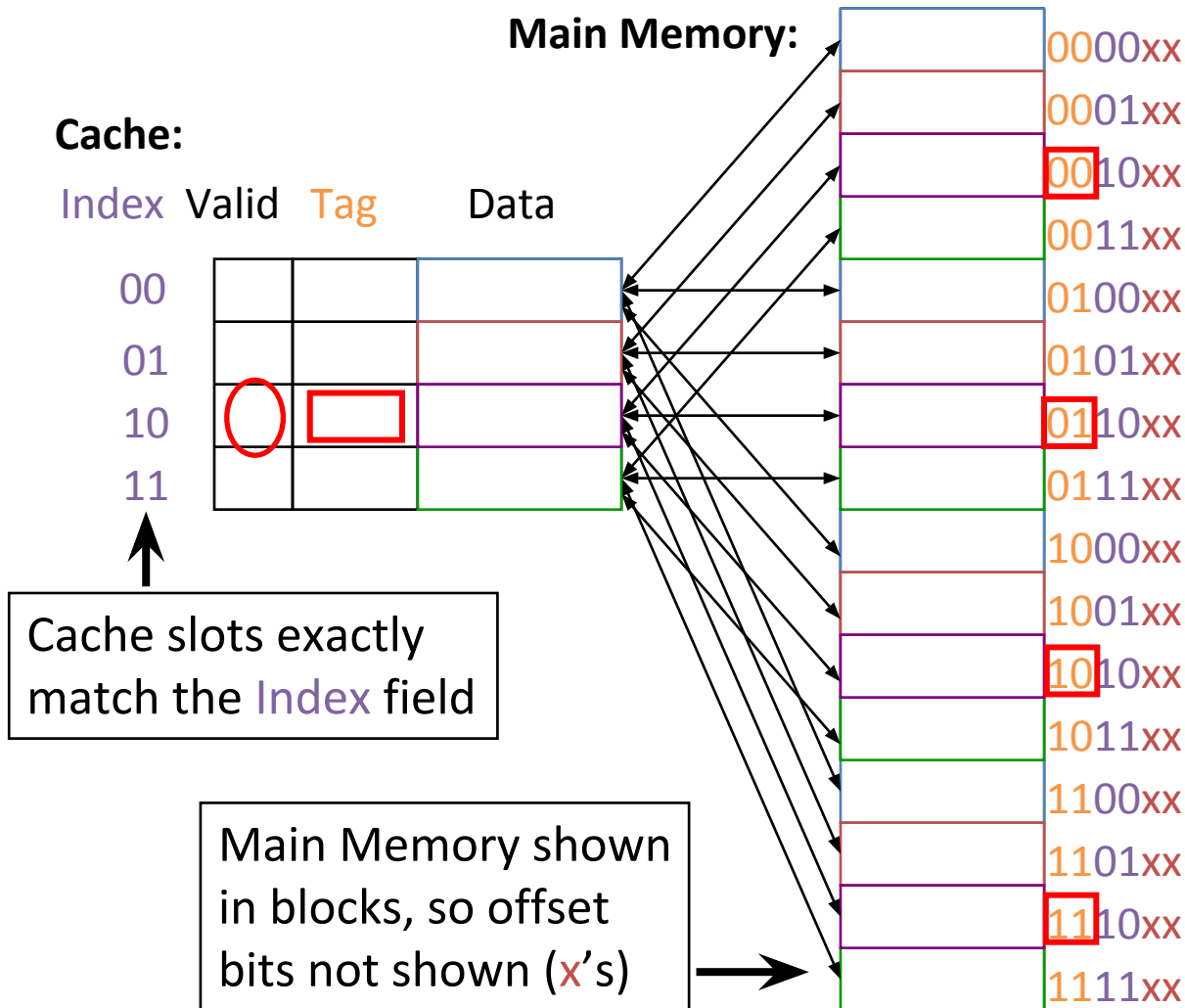
- Cache parameters:
  - Direct-mapped, address space of 64B, block size of 4B, cache size of 16B, write-through
  - **Offset** – 2 bits, **Index** – 2 bits, **Tag** – 2 bits

**Offset**

	<b>V</b>	<b>Tag</b>	<b>00</b>	<b>01</b>	<b>10</b>	<b>11</b>
<b>00</b>	X	XX	0x??	0x??	0x??	0x??
<b>01</b>	X	XX	0x??	0x??	0x??	0x??
<b>10</b>	X	XX	0x??	0x??	0x??	0x??
<b>11</b>	X	XX	0x??	0x??	0x??	0x??

- 35 bits per index/slot, 140 bits to implement

# DM Cache Example (3/5)



Which blocks map to each row of the cache? (see colors)

On a memory request: (let's say  $001011_{two}$ )

- 1) Take Index field (10)
- 2) Check if Valid bit is true in that row of cache
- 3) If valid, then check if Tag matches

# DM Cache Example (4/5)

- Consider the sequence of memory address accesses

Starting with a cold cache:      0   2   4   8   20   16   0   2

**000000**

**0 miss**

00	1	00	M[0]	M[1]	M[2]	M[3]
01	0	00	0x??	0x??	0x??	0x??
10	0	00	0x??	0x??	0x??	0x??
11	0	00	0x??	0x??	0x??	0x??

**000100**

**4 miss**

00	1	00	M[0]	M[1]	M[2]	M[3]
01	1	00	M[4]	M[5]	M[6]	M[7]
10	0	00	0x??	0x??	0x??	0x??
11	0	00	0x??	0x??	0x??	0x??

**000010**

**2 hit**

00	1	00	M[0]	M[1]	M[2]	M[3]
01	0	00	0x??	0x??	0x??	0x??
10	0	00	0x??	0x??	0x??	0x??
11	0	00	0x??	0x??	0x??	0x??

**001000**

**8 miss**

00	1	00	M[0]	M[1]	M[2]	M[3]
01	1	00	M[4]	M[5]	M[6]	M[7]
10	1	00	M[8]	M[9]	M[10]	M[11]
11	0	00	0x??	0x??	0x??	0x??

# DM Cache Example (5/5)

- Consider the sequence of memory address accesses

Starting with a cold cache:      0   2   4   8   20   16   0   2

**010100**

**20 miss**

00	1	00	M[0]	M[1]	M[2]	M[3]
01	1	<del>00</del>	<del>M[4]</del>	<del>M[5]</del>	<del>M[6]</del>	<del>M[7]</del>
10	1	00	M[8]	M[9]	M[10]	M[11]
11	0	00	0x??	0x??	0x??	0x??

**000000**

**0 miss**

00	1	<del>01</del>	<del>M[16]</del>	<del>M[17]</del>	<del>M[18]</del>	<del>M[19]</del>
01	1	01	M[20]	M[21]	M[22]	M[23]
10	1	00	M[8]	M[9]	M[10]	M[11]
11	0	00	0x??	0x??	0x??	0x??

**010000**

**16 miss**

00	1	<del>00</del>	<del>M[0]</del>	<del>M[1]</del>	<del>M[2]</del>	<del>M[3]</del>
01	1	01	M[20]	M[21]	M[22]	M[23]
10	1	00	M[8]	M[9]	M[10]	M[11]
11	0	00	0x??	0x??	0x??	0x??

**000010**

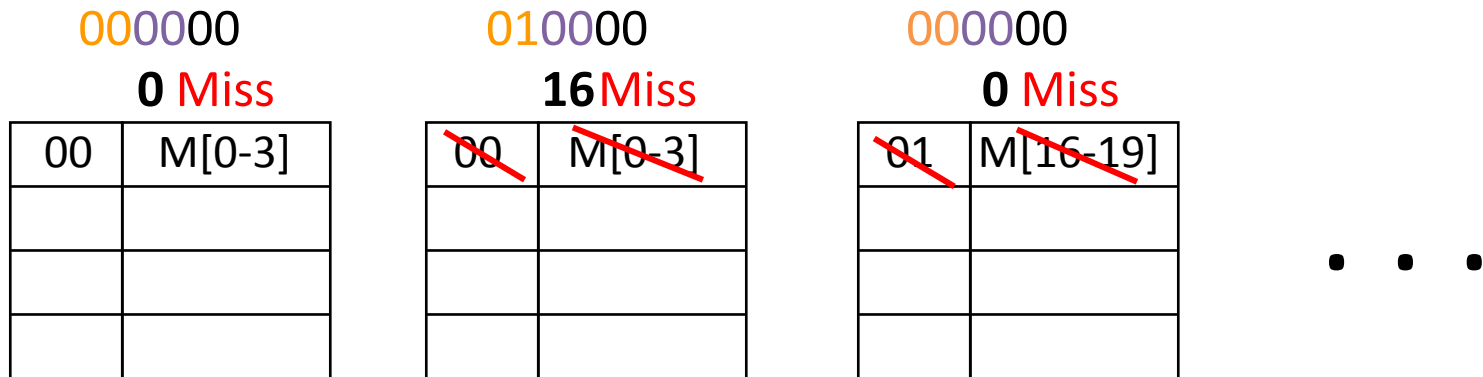
**2 hit**

00	1	00	M[0]	M[1]	M[2]	M[3]
01	1	01	M[20]	M[21]	M[22]	M[23]
10	1	00	M[8]	M[9]	M[10]	M[11]
11	0	00	0x??	0x??	0x??	0x??

- 8 requests, 6 misses – last slot was never used!

# Worst-Case for Direct-Mapped

- Cold DM \$ that holds four 1-word blocks
- Consider the memory accesses: 0, 16, 0, 16,...





# Comparison So Far

- Fully associative
  - Block can go into *any* slot
  - Must check ALL cache slots on request (“slow”)
  - **TO** breakdown (i.e. **I** = 0 bits)
  - “Worst case” still fills cache (more efficient)
- Direct-mapped
  - Block goes into *one specific* slot (set by Index field)
  - Only check ONE cache slot on request (“fast”)
  - **TIO** breakdown
  - “Worst case” may only use 1 slot (less efficient)

# Meet the Staff



	Sukrit	Suvansh
<b>Favorite Villain</b>	The Lannisters	Logisim [De]Evolution
<b>What would you protest</b>	Prerequisite enforcement	CS Design requirement
<b>What are you passionate about?</b>	Music	American football
<b>What you'd want to be famous for?</b>	Arora's Algorithm	Facial Hair

# Agenda

- Review of yesterday
- Administrivia
- Direct-Mapped Caches
- **Set Associative Caches**
- Cache Performance

# Set Associative Caches

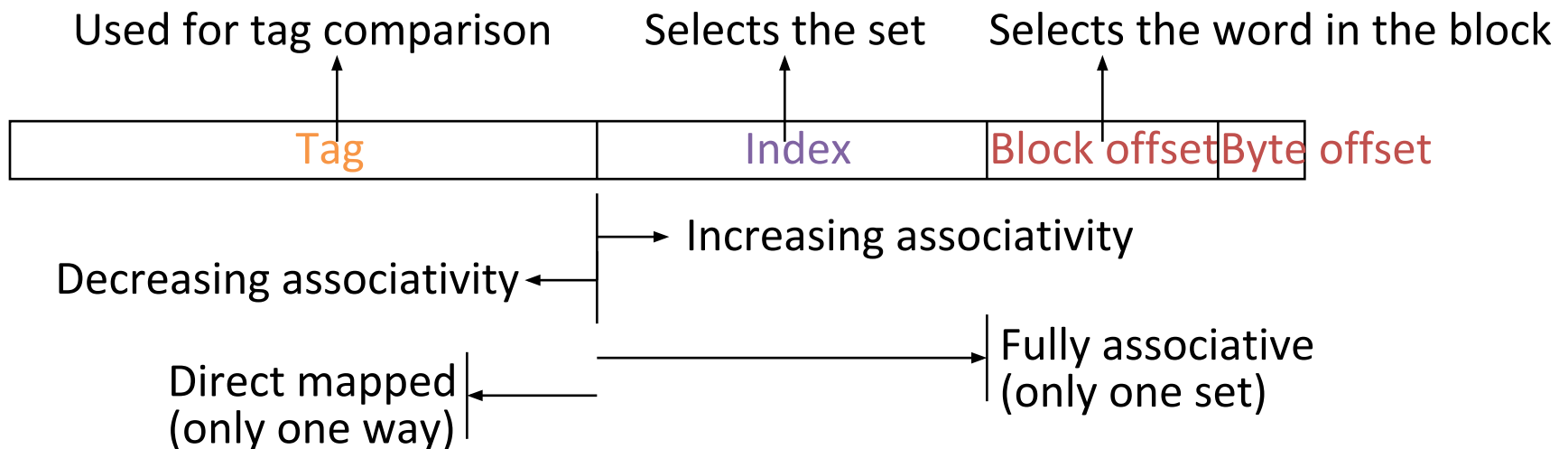
- Compromise!
  - More flexible than DM, more structured than FA
- *N-way set-associative*: Divide  $S$  into sets, each of which consists of  $N$  slots
  - Memory block maps to a set determined by **Index** field and is placed in any of the  $N$  slots of that set
  - Call  $N$  the *associativity*
  - New hash function:  
(block address) modulo (# sets in the cache)
  - Replacement policy applies to every *set*

# Effect of Associativity on TIO (1/2)

- Here we assume a cache of fixed size (C)
- **Offset:** # of bytes in a block (same as before)
- **Index:** Instead of pointing to a *slot*, now points to a *set*, so  $I = \log_2(C \div K \div N)$ 
  - Fully associative (1 set): 0 **Index** bits!
  - Direct-mapped (N = 1): max **Index** bits
  - Set associative: somewhere in-between
- **Tag:** Remaining identifier bits ( $T = A - I - O$ )

# Effect of Associativity on TIO (2/2)

- For a fixed-size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e. the number of slots) and halves the number of sets – decreasing the size of the **Index** by 1 bit and increasing the size of the **Tag** by 1 bit



# Example: Eight-Block Cache Configs

**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								

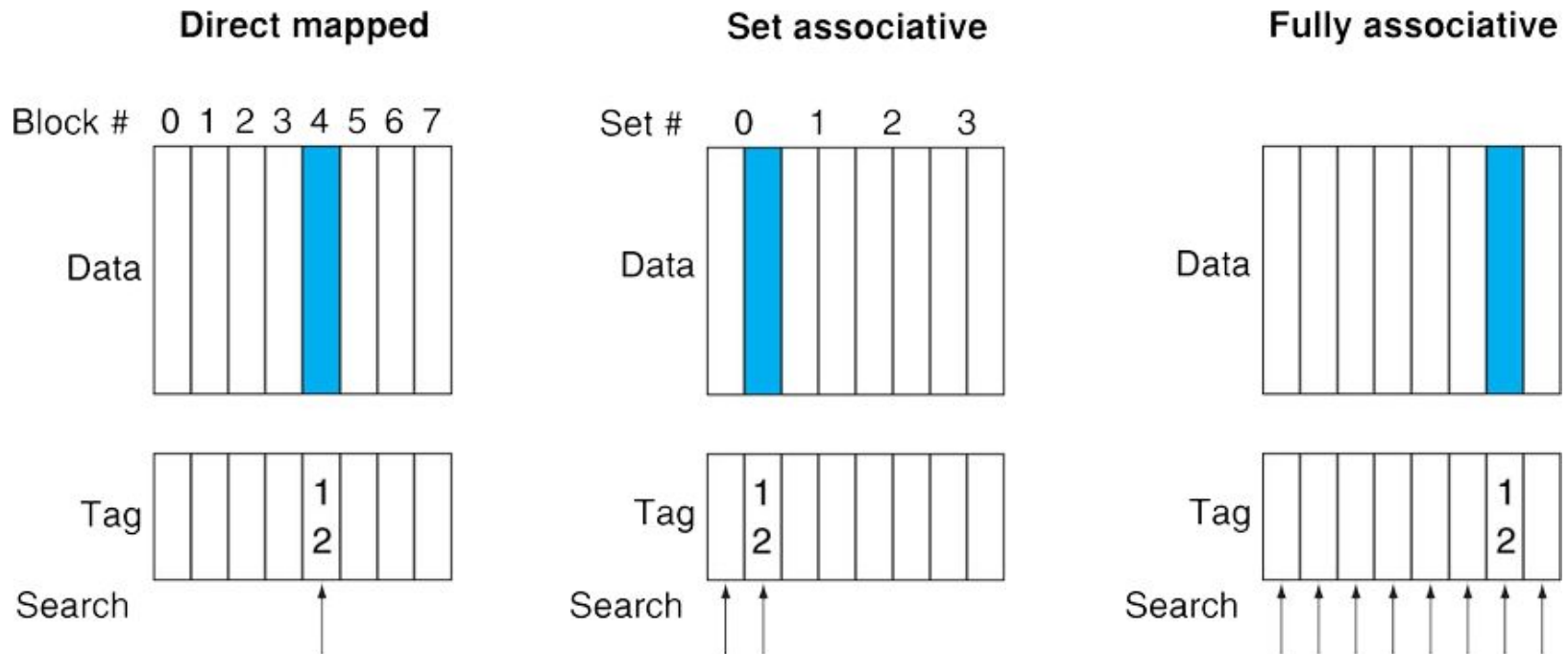
**Eight-way set associative (fully associative)**

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

- Total size of \$ =  $\# \text{ sets} \times \text{associativity}$
- For fixed \$ size, associativity  $\uparrow$  means  $\# \text{ sets} \downarrow$  and slots per set  $\uparrow$
- With 8 blocks, an 8-way set associative \$ is same as a fully associative \$

# Block Placement Schemes

- Place memory block 12 in a cache that holds 8 blocks



- Fully associative:** Can go in *any* of the slots (all 1 set)
- Direct-mapped:** Can only go in slot  $(12 \bmod 8) = 4$
- 2-way set associative:** Can go in either slot of set  $(12 \bmod 4) = 0$



# SA Cache Example (1/5)

- Cache parameters:
  - 2-way set associative, 6-bit addresses, 1-word blocks, 4-word cache, write-through
- How many sets?
  - $C \div K \div N = 4 \div 1 \div 2 = 2$  sets
- TIO Breakdown:
  - $O = \log_2(4) = 2$ ,  $I = \log_2(2) = 1$ ,  $T = 6 - 1 - 2 = 3$



# SA Cache Example (2/5)

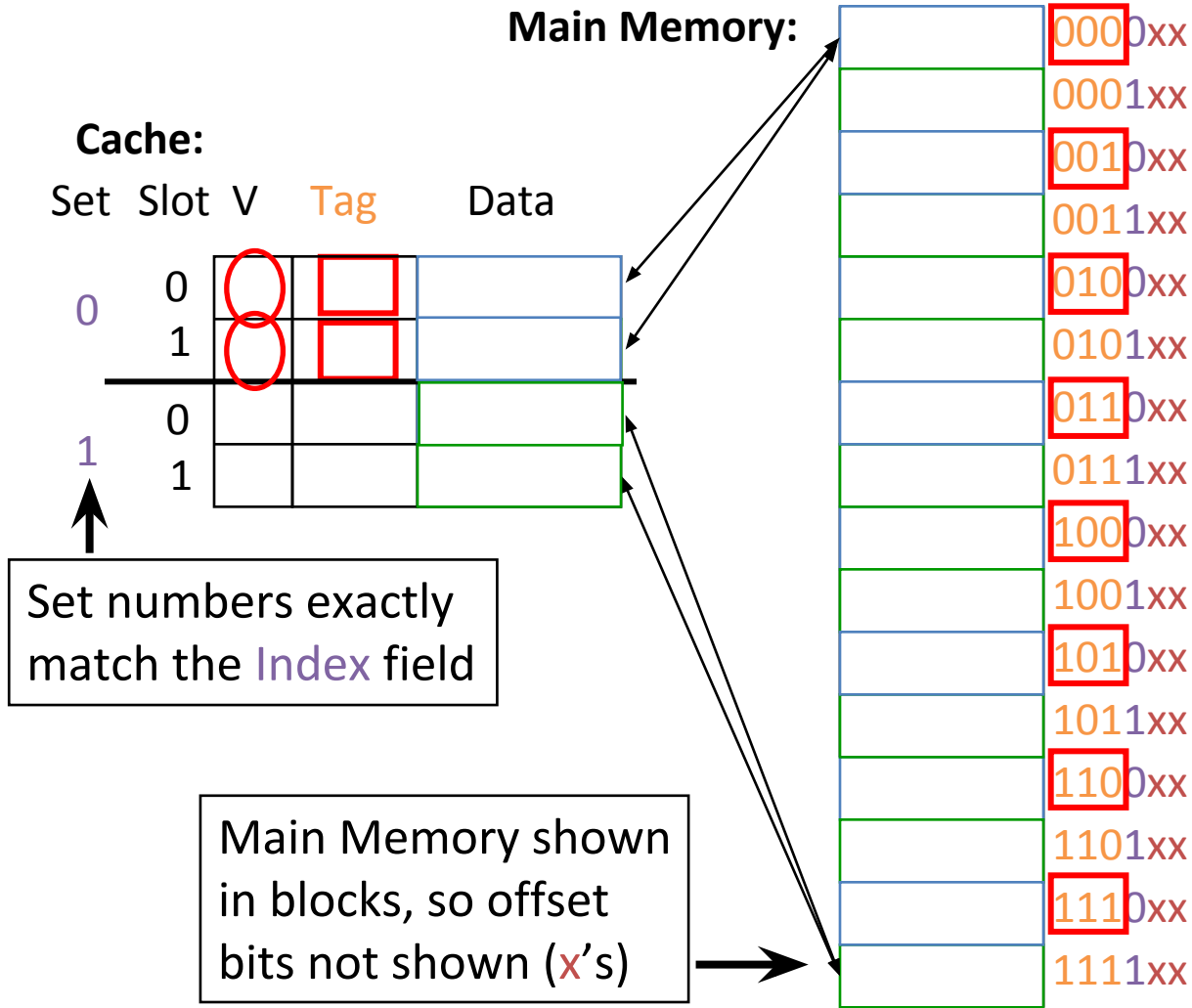
- Cache parameters:
  - 2-way set associative, 6-bit addresses, 1-word blocks, 4-word cache, write-through
  - Offset – 2 bits, Index – 1 bit, Tag – 3 bits

Offset

		V	Tag	00	01	10	11	
0	0	X	XXX	0x??	0x??	0x??	0x??	LRU
	1	X	XXX	0x??	0x??	0x??	0x??	X
Index		<hr/>						
1	0	X	XXX	0x??	0x??	0x??	0x??	LRU
	1	X	XXX	0x??	0x??	0x??	0x??	X

- 37 bits per slot,  $37 * 2 = 74$  bits per set,  $2 * 74 = 148$  bits to implement

# SA Cache Example (3/5)



Each block maps into one set (either slot) (see colors)

On a memory request: (let's say 001011<sub>two</sub>)

- 1) Take Index field (0)
- 2) For EACH slot in set, check valid bit, then compare Tag

# SA Cache Example (4/5)

- Consider the sequence of memory address accesses

Starting with a cold cache:      0   2   4   8   20   16   0   2

**000000**

**0 miss**

0	0	1	000	M[0]	M[1]	M[2]	M[3]
	1	0	000	0x??	0x??	0x??	0x??
1	0	0	000	0x??	0x??	0x??	0x??
	1	0	000	0x??	0x??	0x??	0x??

**000100**

**4 miss**

0	0	1	000	M[0]	M[1]	M[2]	M[3]
	1	0	000	0x??	0x??	0x??	0x??
1	0	1	000	M[4]	M[5]	M[6]	M[7]
	1	0	000	0x??	0x??	0x??	0x??

**000010**

**2 hit**

0	0	1	000	M[0]	M[1]	M[2]	M[3]
	1	0	000	0x??	0x??	0x??	0x??
1	0	0	000	0x??	0x??	0x??	0x??
	1	0	000	0x??	0x??	0x??	0x??

**001000**

**8 miss**

0	0	1	000	M[0]	M[1]	M[2]	M[3]
	1	1	001	M[8]	M[9]	M[10]	M[11]
1	0	1	000	M[4]	M[5]	M[6]	M[7]
	1	0	000	0x??	0x??	0x??	0x??

# SA Cache Example (5/5)

- Consider the sequence of memory address accesses

Starting with a cold cache:

010100

20 miss

0 2 4 8 20 16 0 2  
M H M M

010000

16 miss

0	0	1	000	M[0]	M[1]	M[2]	M[3]
	1	1	001	M[8]	M[9]	M[10]	M[11]
1	0	1	000	M[4]	M[5]	M[6]	M[7]
	1	1	010	M[20]	M[21]	M[22]	M[23]

000000

0 miss

0	0	1	010	M[16]	M[17]	M[18]	M[19]
	1	1	<del>001</del>	<del>M[8]</del>	<del>M[9]</del>	<del>M[10]</del>	<del>M[11]</del>
1	0	1	000	M[4]	M[5]	M[6]	M[7]
	1	1	010	M[20]	M[21]	M[22]	M[23]

0	0	1	<del>000</del>	<del>M[0]</del>	<del>M[1]</del>	<del>M[2]</del>	<del>M[3]</del>
	1	1	001	M[8]	M[9]	M[10]	M[11]
1	0	1	000	M[4]	M[5]	M[6]	M[7]
	1	1	010	M[20]	M[21]	M[22]	M[23]

000010

2 hit

0	0	1	010	M[16]	M[17]	M[18]	M[19]
	1	1	000	M[0]	M[1]	M[2]	M[3]
1	0	1	000	M[4]	M[5]	M[6]	M[7]
	1	1	010	M[20]	M[21]	M[22]	M[23]

- 8 requests, 6 misses

# Worst Case for Set Associative

- Worst case for DM was repeating pattern of 2 into same cache slot (HR = 0/n)
  - Set associative for  $N > 1$ :  $HR = (n-2)/n$
- Worst case for N-way SA with LRU?
  - Repeating pattern of at least  $N+1$  that maps into same set

– Back to HR = 0:      0, 8, 16, 0, 8,  
                                   M M M M M  
                                   ...

000	M[0-19]
000	M[8-31]

**Question:** What is the TIO breakdown for the following cache?

- 32-bit address space
- 32 KiB 4-way set associative cache
- 8 word blocks

$A = 32, C = 32 \text{ KiB} = 2^{15} \text{ B}, N = 4, K = 8 \text{ words} = 32 \text{ B}$

	<b>T</b>	<b>I</b>	<b>O</b>
(A)	21	8	3
(B)	19	8	5
(C)	19	10	3
(D)	17	10	5

$O = \log_2(K) = 5 \text{ bits}$

$C/K = 2^{10} \text{ slots}$

$C/K/N = 2^8 \text{ sets}$

$I = \log_2(C/K/N) = 8 \text{ bits}$

$T = A - I - O = 19 \text{ bits}$

# Summary

- Set associativity determines flexibility of block placement
  - Fully associative: blocks can go anywhere
  - Direct-mapped: blocks go in one specific location
  - N-way: cache split into sets, each of which have  $n$  slots to place memory blocks