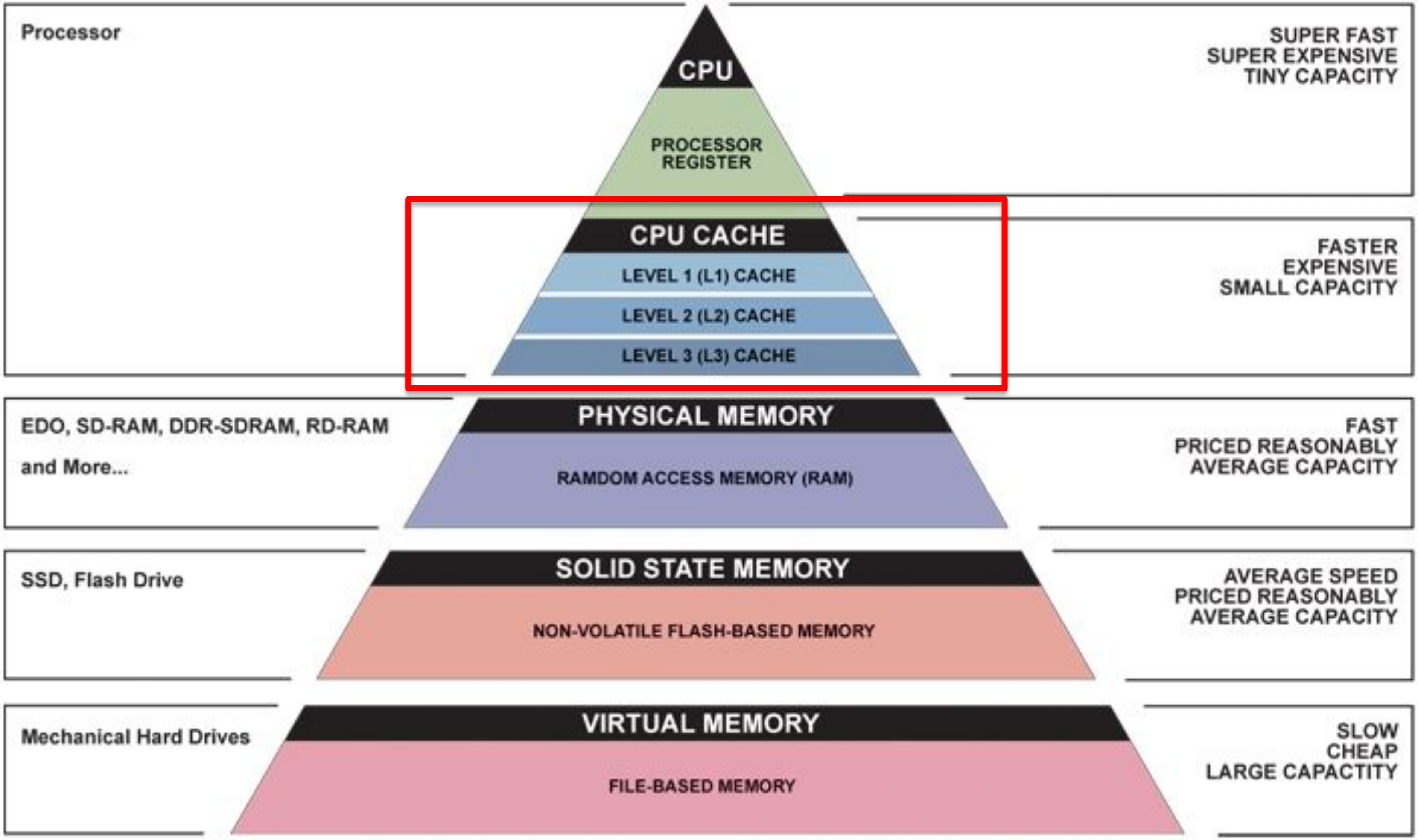
CS 61C

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				
Slot	V	Tag	00	01	10	11	LRU
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??

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??

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]

000100
4 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]

- 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	~~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]

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	~~001~~	~~M[8]~~	~~M[9]~~	~~M[10]~~	~~M[11]~~	~~M[12]~~	~~M[13]~~	~~M[14]~~	~~M[15]~~

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]
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]

LRU
10

(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]
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]

LRU
10

~~(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]
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]

LRU
10

(A) 0 2 12 4 16 8 0 6

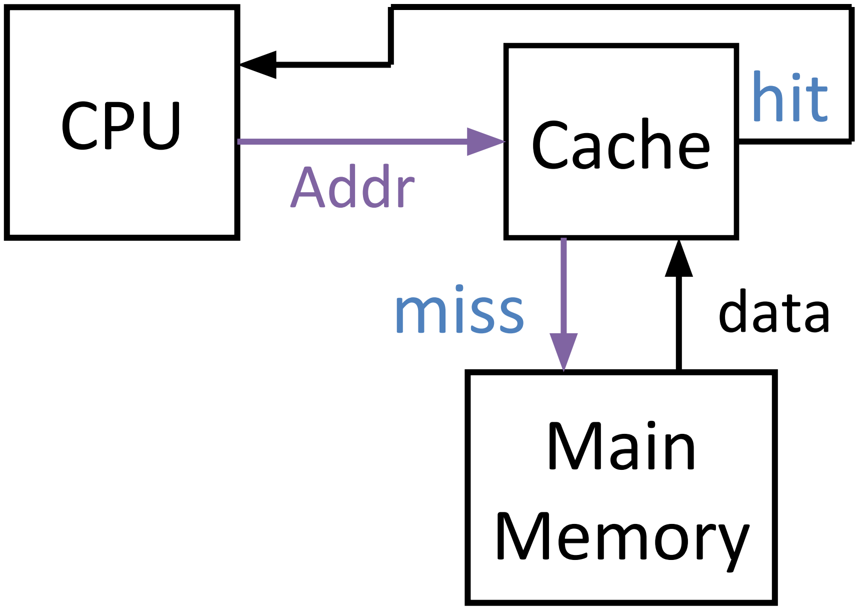
(B) 0 8 4 16 0 12 6 2

(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??
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??

LRU
XX

- 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)

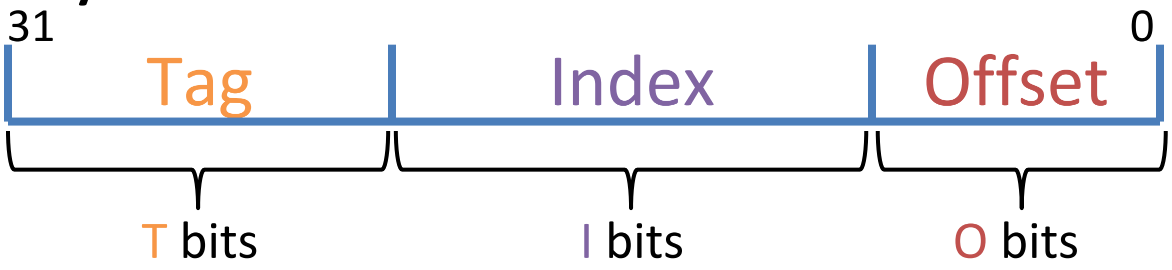
- Each memory block is mapped to *exactly one slot* in the cache (*direct-mapped*)
  - Every block has only one “home”
  - Use hash function to determine which slot
- Comparison with fully associative
  - Check just one slot for a block (faster!)
  - No replacement policy necessary
  - Access pattern 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
    - (*block* address) modulo (# of *blocks* in the cache)
- Tag field maintains same function (identifier), but is now shorter

TIO Address Breakdown

- Memory address fields:



31

0

Tag

Index

Offset

T bits

I bits

O bits

- Meaning of the field sizes:
  - O bits ↔ 2^{O} bytes/block = 2^{O-2} words/block
  - I bits ↔ 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)

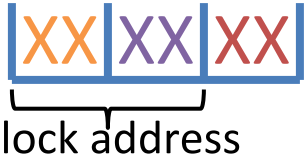
- What's actually in the cache?
  - Block of data ($8 \times K = 8 \times 2^{O}$ bits)
  - Tag field of address as identifier (T bits)
  - Valid bit (1 bit)
  - Dirty bit (1 bit if write-back)
  - No replacement management bits!
- Total bits in cache = # slots × ($8 \times K + T + 1 + 1$)

  $= 2^{I} \times (8 \times 2^{O} + T + 1 + 1)$ 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:
  - $O = \log_2(4) = 2$
  - Cache size / block size = 16/4 = 4, so $I = \log_2(4) = 2$
  - $A = \log_2(64) = 6$ bits, so $T = 6 - 2 - 2 = 2$

Memory Addresses:


XX XX XX

Block address

- Bits in cache = $2^2 \times (8\times2^2 + 2 + 1) = 140$ bits

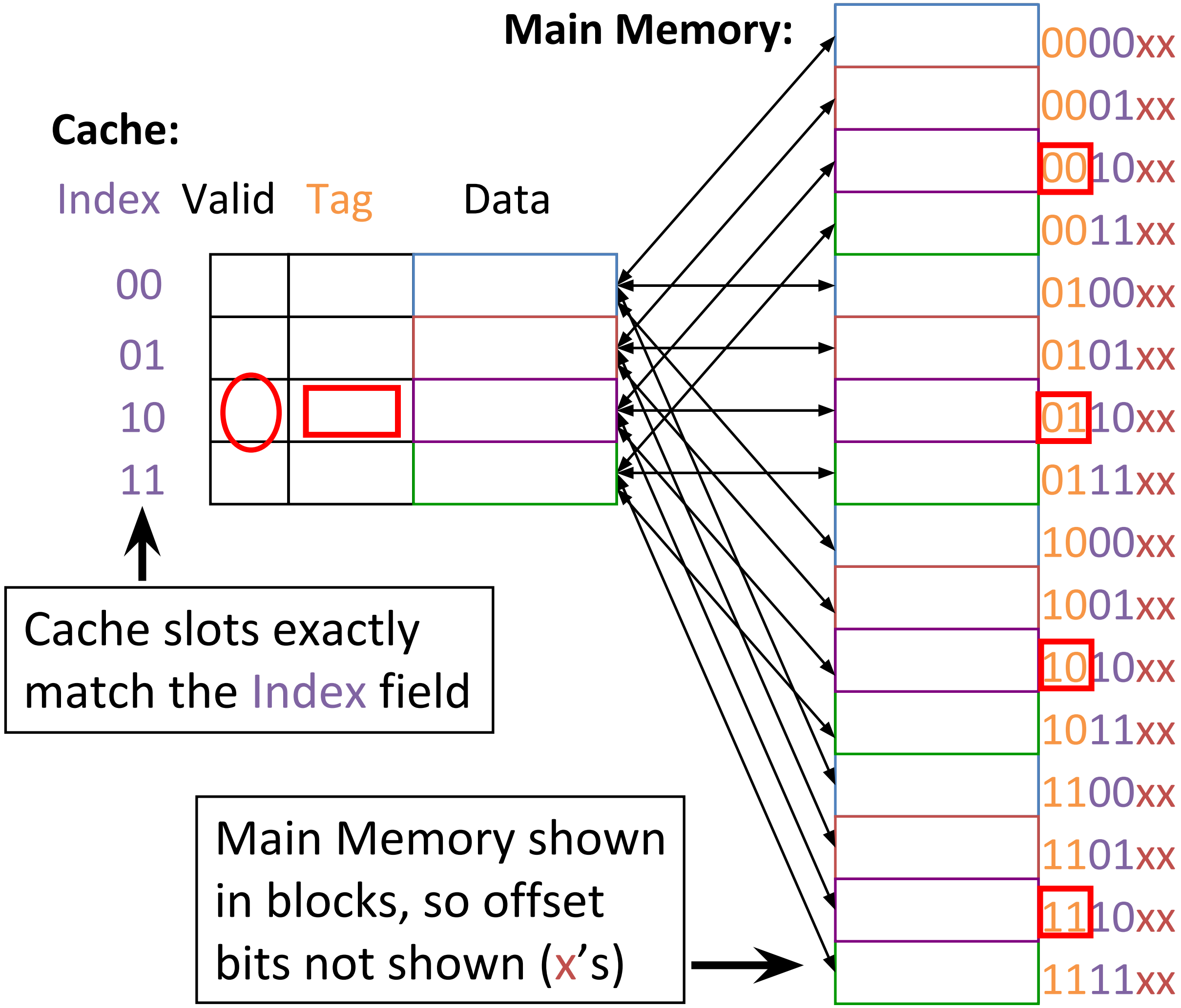
DM Cache Example (2/5)

- 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			
Index	V	Tag	00	01	10	11
00	X	XX	0x??	0x??	0x??	0x??
01	X	XX	0x??	0x??	0x??	0x??
10	X	XX	0x??	0x??	0x??	0x??
11	X	XX	0x??	0x??	0x??	0x??

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

DM Cache Example (3/5)



Cache slots exactly match the Index field

Main Memory shown in blocks, so offset bits not shown (x's)

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??

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??

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??

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	~~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??

010000

16 miss

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??

000000

0 miss

00	1	~~01~~	~~M[16]~~	~~M[17]~~	~~M[18]~~	~~M[19]~~
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,...

000000

0 Miss

00	M[0-3]

010000

16 Miss

~~00~~	~~M[0-3]~~

000000

0 Miss

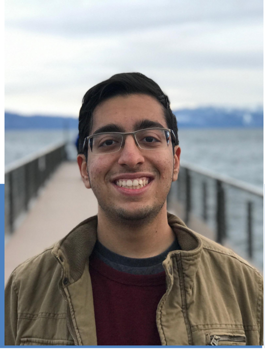
~~01~~	~~M[16-19]~~

. . .

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

Agenda

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

Set Associative Caches

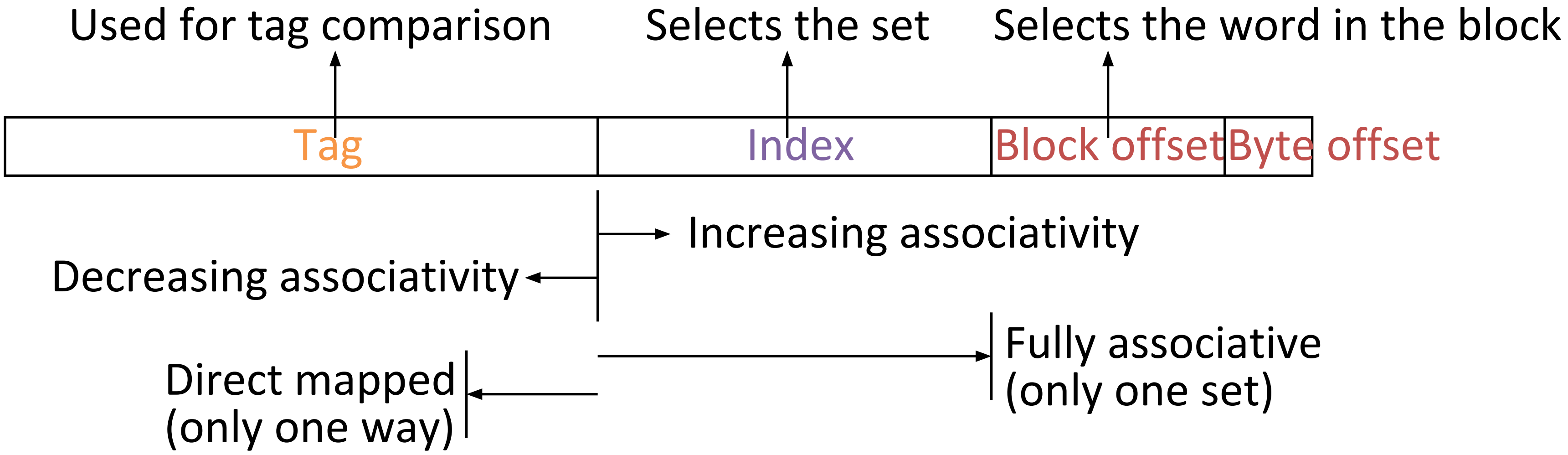
- Compromise!
  - More flexible than DM, more structured than FA
- *N-way set-associative:* Divide $ into sets, each of which consists of N slots
  - Memory block maps to a set determined by Index field and is placed in any of the N slots of that set
  - Call N the *associativity*
  - New hash function: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

Effect of Associativity on TIO (1/2)

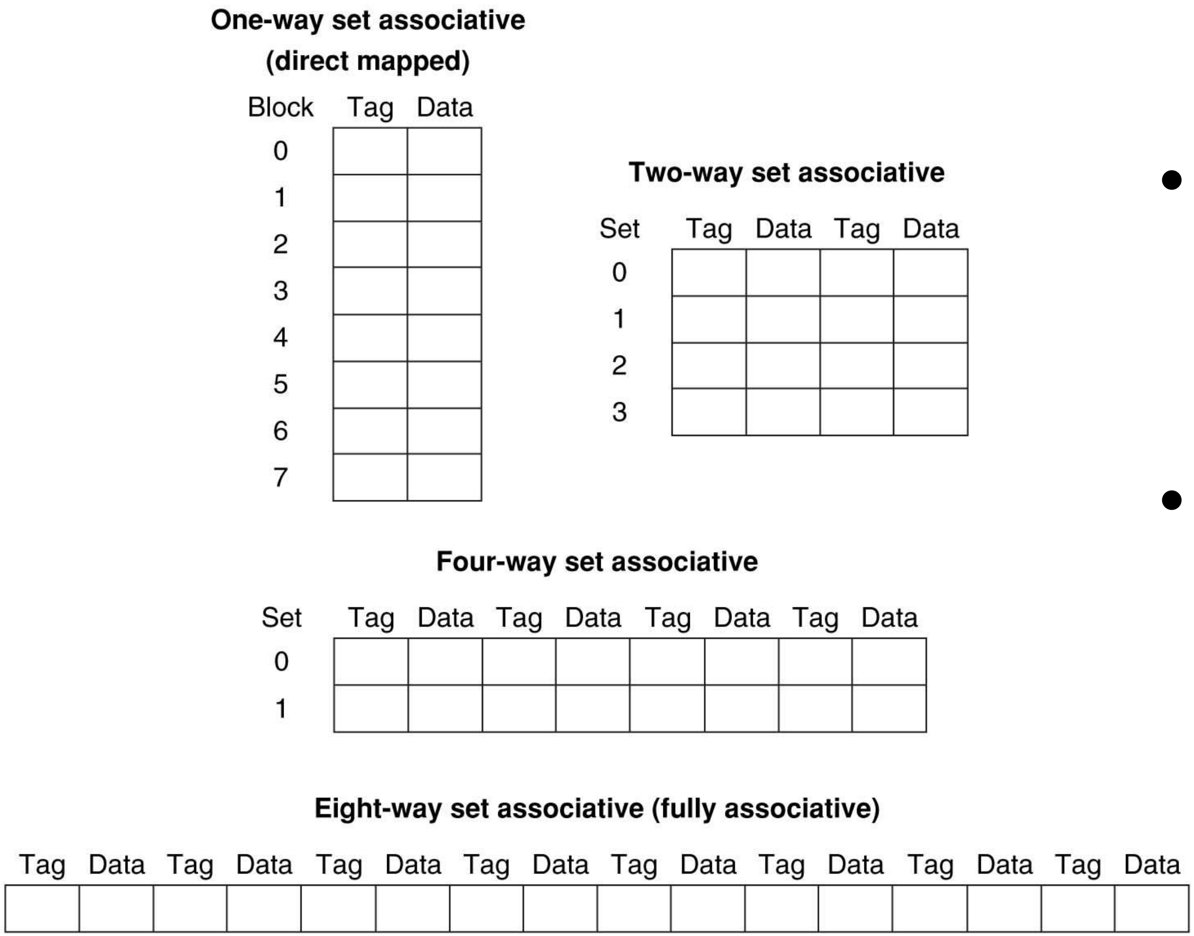
- Here we assume a cache of fixed size (C)
- **Offset:** # of bytes in a block (same as before)
- **Index:** Instead of pointing to a *slot*, now points to a *set*, so $I = \log_2(C \div K \div N)$
  - – Fully associative (1 set): 0 Index bits!
  - – Direct-mapped (N = 1): max Index bits
  - – Set associative: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

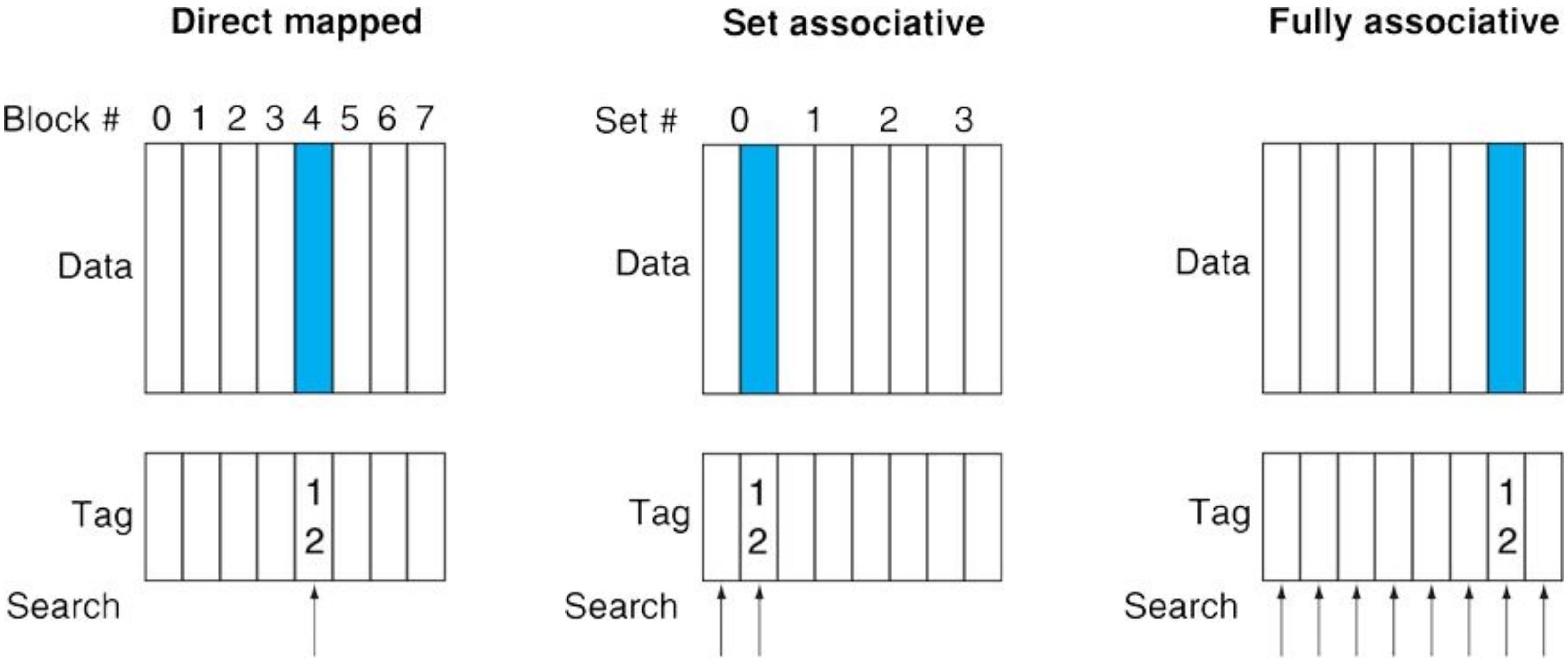
Eight-way set associative (fully associative)

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

- Total size of \$ = *# sets* × *associativity*
- For fixed \$ size, associativity ↑ means # sets ↓ and slots per set ↑
- 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

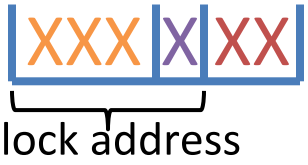


- **Fully associative:** Can go in *any* of the slots (all 1 set)
- **Direct-mapped:** Can only go in slot (12 mod 8) = 4
- **2-way set associative:** Can go in either slot of set (12 mod 4) = 0

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÷K÷N = 4÷1÷2 = 2 sets
- TIO Breakdown:
  - O = $\log_2(4)$ = 2, I = $\log_2(2)$ = 1, T = 6 – 1 – 2 = 3

Memory Addresses:



XXX X XX

Block address

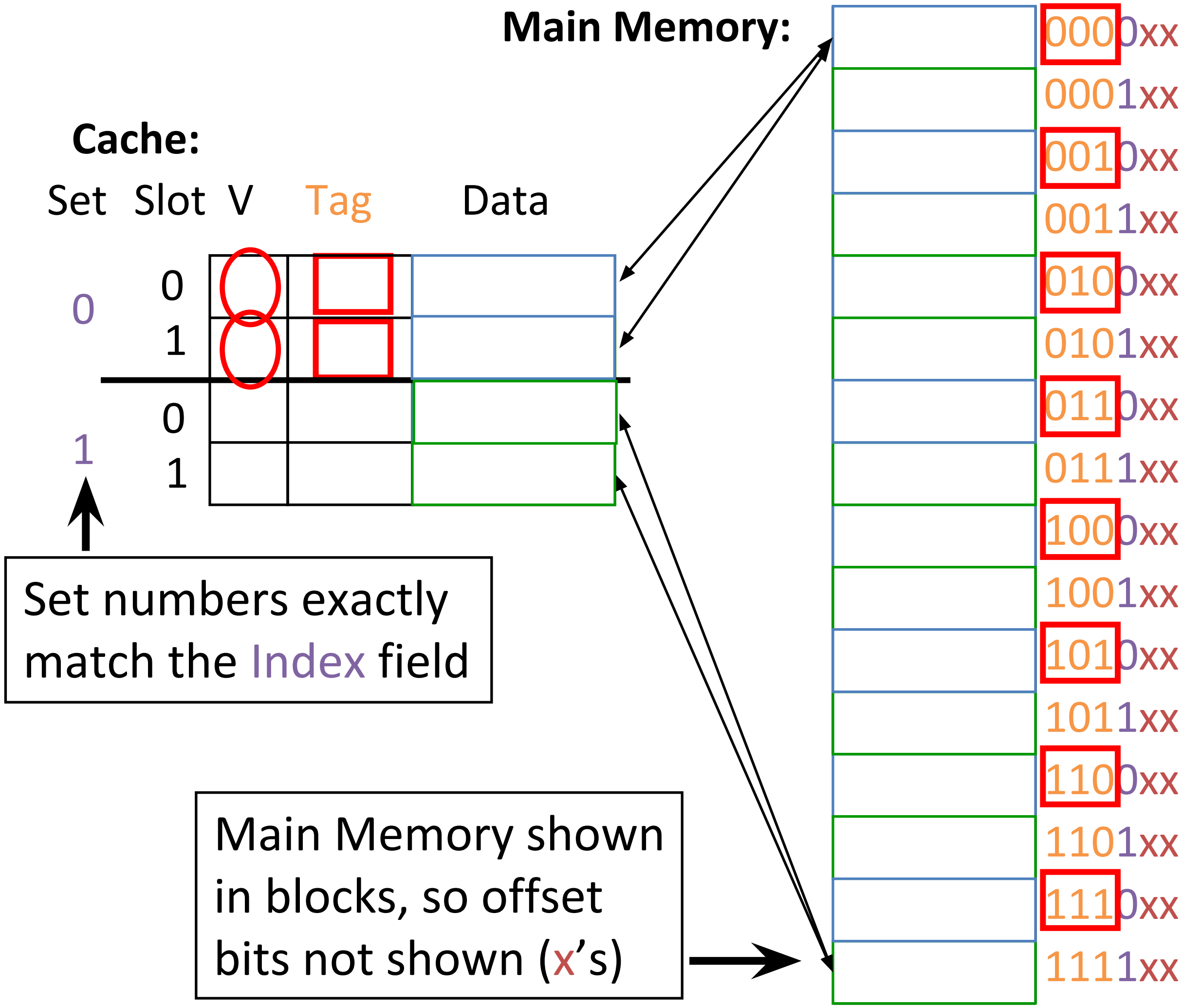
SA Cache Example (2/5)

- 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				
Index		V	Tag	00	01	10	11	
0	0	X	XXX	0x??	0x??	0x??	0x??	LRU
	1	X	XXX	0x??	0x??	0x??	0x??	X
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)



Main Memory:

0000xx
0001xx
0010xx
0011xx
0100xx
0101xx
0110xx
0111xx
1000xx
1001xx
1010xx
1011xx
1100xx
1101xx
1110xx
1111xx

Cache:

Set	Slot	V	Tag	Data
0	0			
	1			
1	0			
	1			

Set numbers exactly match the Index field

Main Memory shown in blocks, so offset bits not shown (x's)

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

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

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??

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??

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??

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: 0 2 4 8 20 16 0 2

M H M M

010100

20 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]

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	~~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

000 001 010	M[0-3] M[8-11] M[16-19]
000 001	M[0-3] M[8-11]

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 KiB = 2^{15} B, N = 4, K = 8 words = 32 B

	T	I	O
(A)	21	8	3
[✓ (B)]	19	8	5
(C)	19	10	3
(D)	17	10	5

O = $\log_2(K)$ = 5 bits

C/K = 2^{10} slots

C/K/N = 2^8 sets

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

T = A – I – O = 19 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