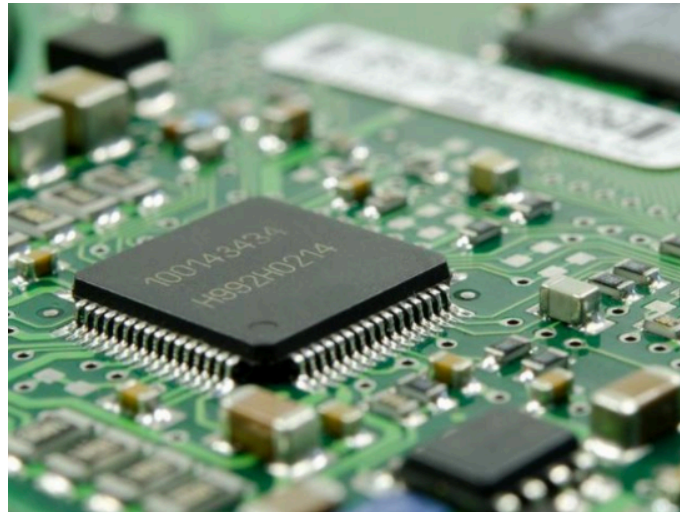




Great Ideas in Computer Architecture

Direct-Mapped and Set Associative Caches

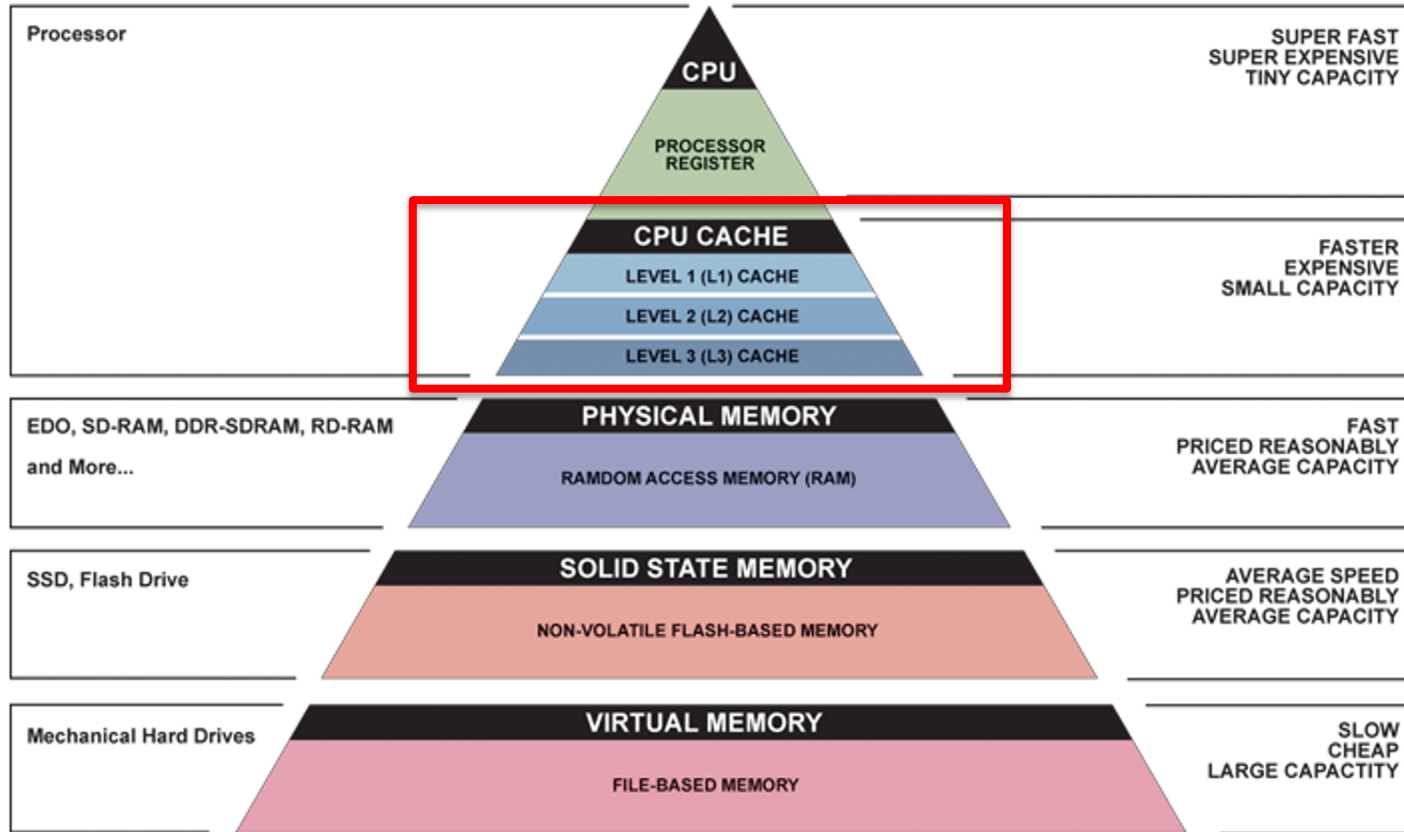
Instructor: Jenny Song



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)

Great Idea #3: Principle of Locality/ Memory Hierarchy



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

Offset

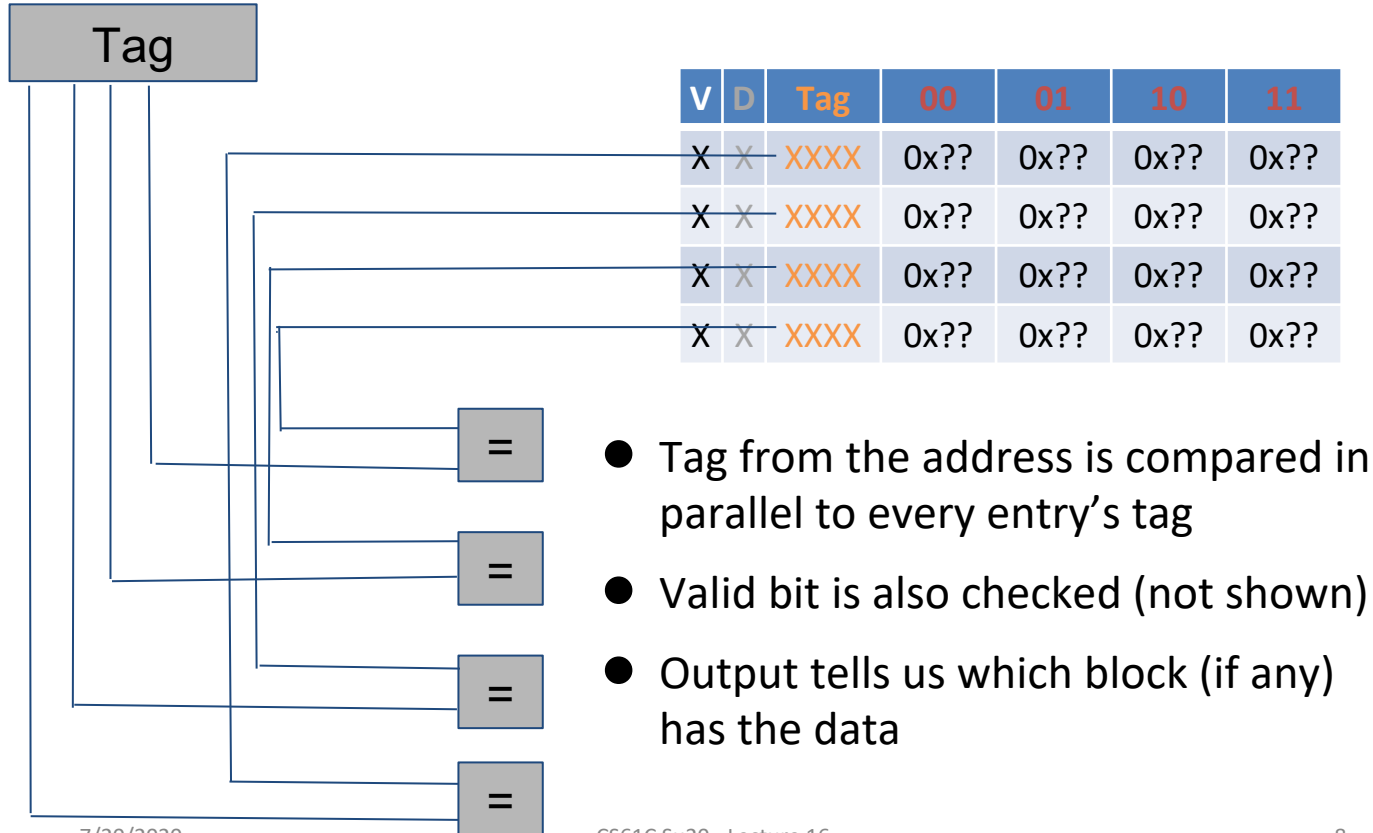
	V	Tag	00	01	10	11	LRU
Slot 0	X	XXXX	0x??	0x??	0x??	0x??	XX
1	X	XXXX	0x??	0x??	0x??	0x??	XX
2	X	XXXX	0x??	0x??	0x??	0x??	XX
3	X	XXXX	0x??	0x??	0x??	0x??	XX

Yesterday's example was write through and looked like this

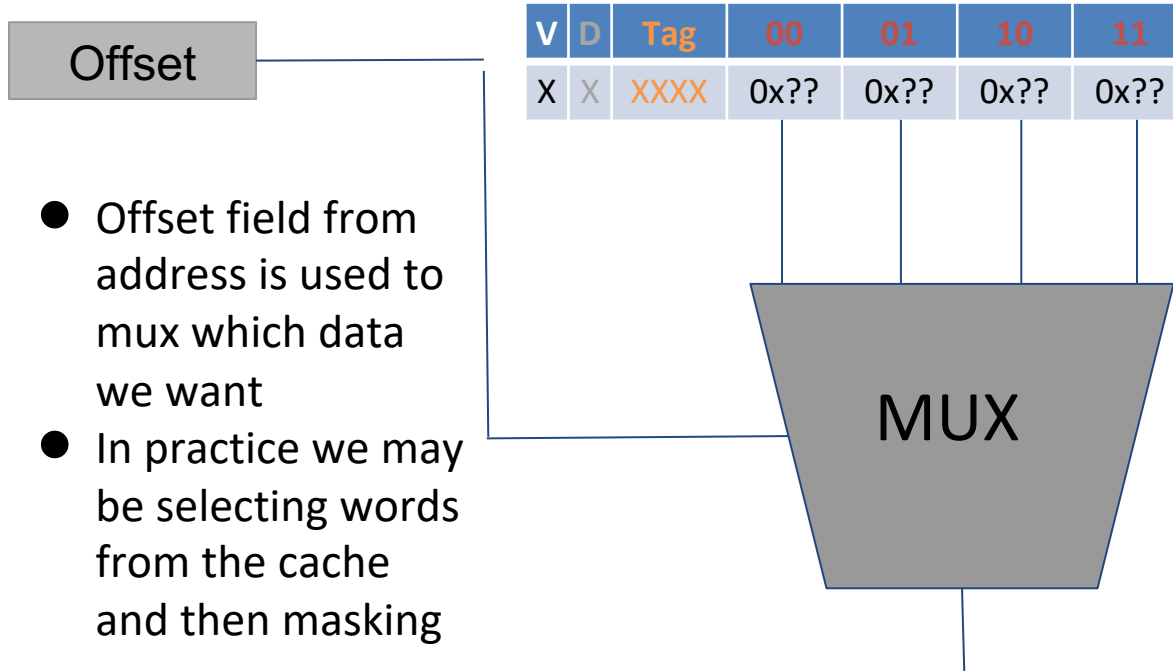
Executing a Load Byte (FA)



Executing a Load Byte (FA)

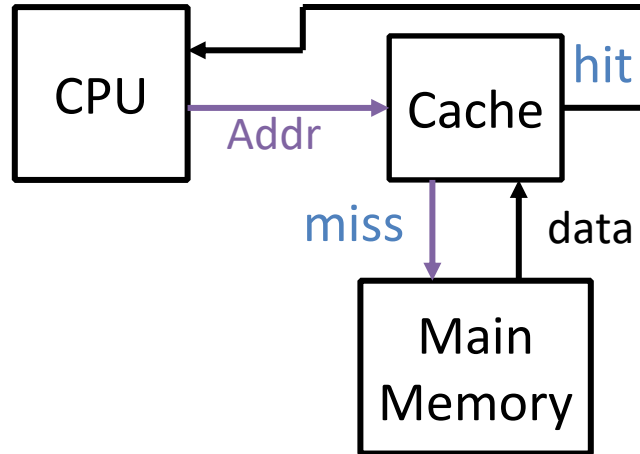


Executing a Load Byte (FA)



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)
 - 2) **Write-Back Policy:** Write data only to cache, then update memory when block is removed
 - 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		
Slot	0	X	X	XXXX	0x??	0x??	0x??	0x??		LRU XX
	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 last time
- Direct-Mapped Caches
- Set Associative Caches
- Cache Performance

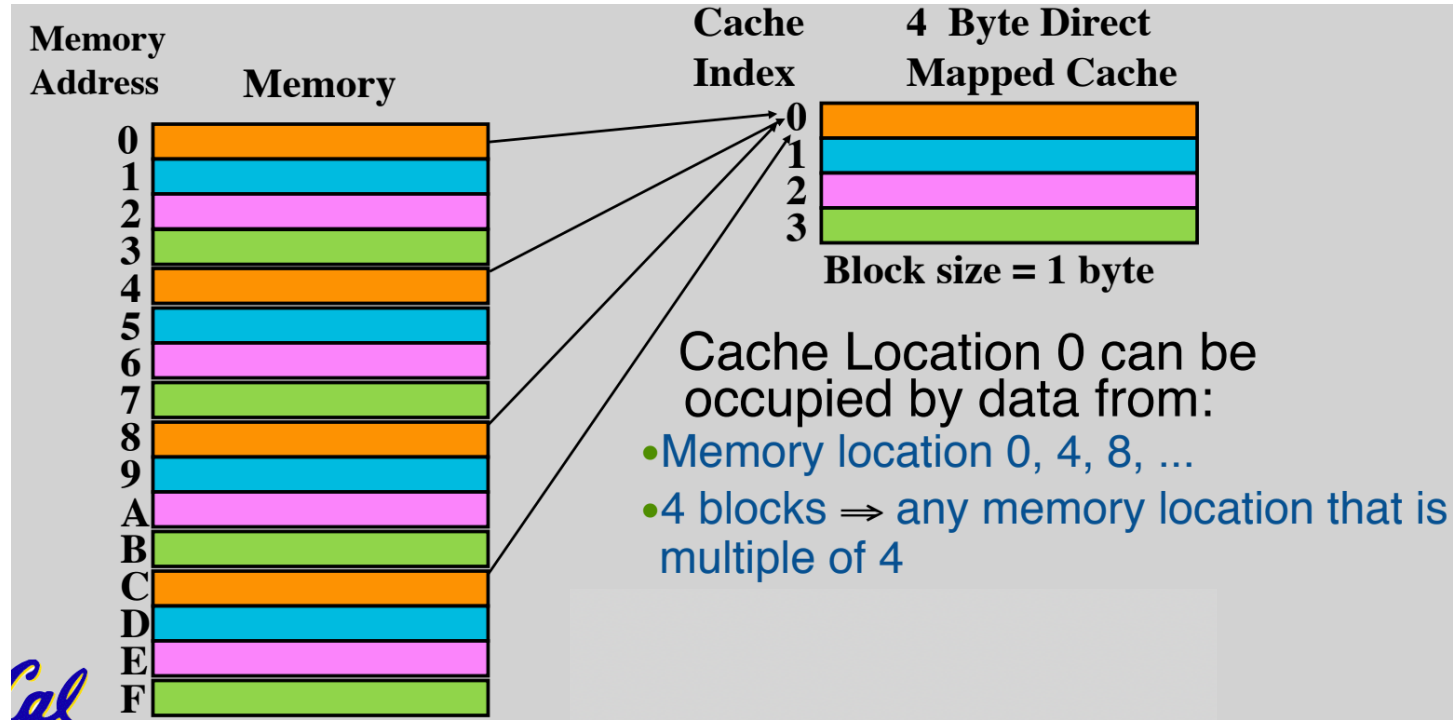
Direct-Mapped Caches (1/3)

- 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)

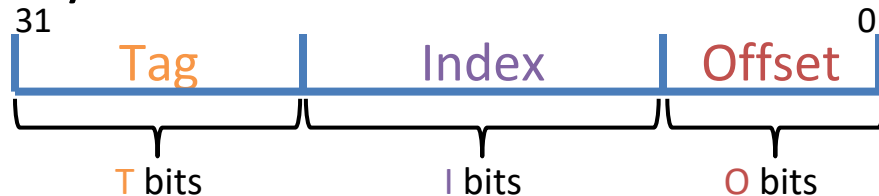
- **Index field:** specifies the cache index (which slot of the cache we should look at)
- **Offset field:** once we've found correct block, specifies which byte within the block we want.
- **Tag field:** the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location.

Direct-Mapped Caches (1/3)



TIO Address Breakdown

- Memory address fields:



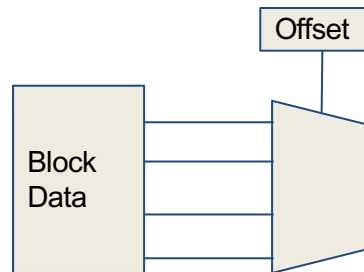
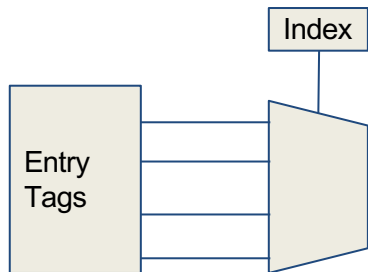
- Meaning of the field sizes:
 - O bits $\leftrightarrow 2^O$ bytes/block = 2^{O-2} words/block
 - I bits $\leftrightarrow 2^I$ slots in cache = cache size / block size
 - T bits = $A - I - O$, where A = # of address bits
($A = 32$ here)

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

Where are the index and offset?

- Index and offset portions of the address are not actually stored in the cache but are stored implicitly via hardware
- Index and offset values are determined from the address, which leads to hardware decisions



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

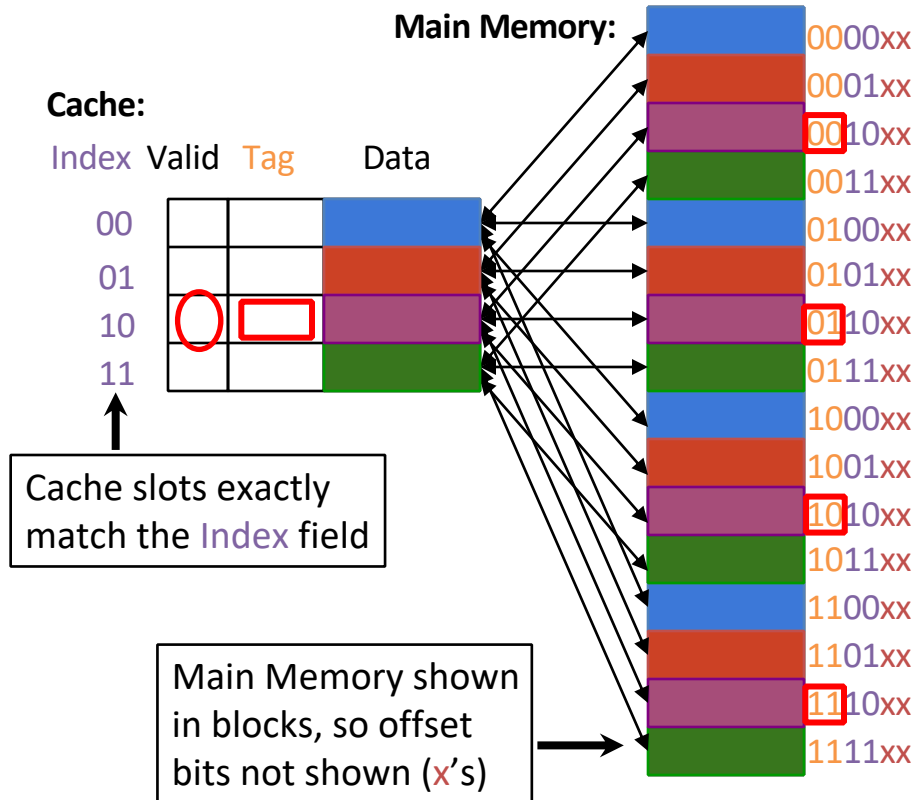
DM Cache Example (2/5)

- 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					
		V	Tag	00	01	10	11
Index	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)



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	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??

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??

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??

000010

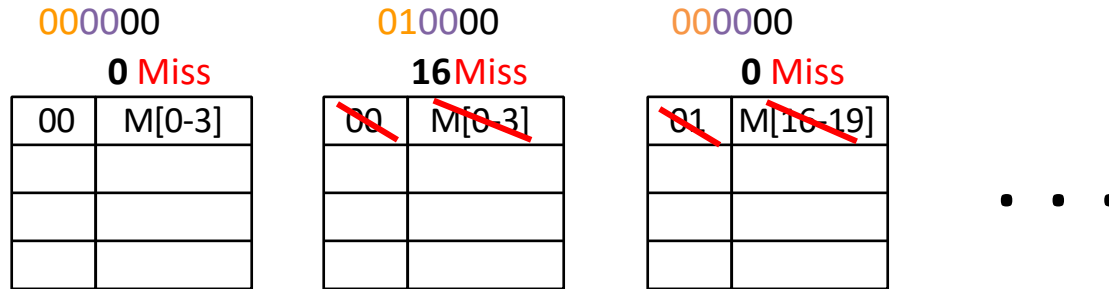
2 hit

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)

Agenda

- Review of last time
- Administtrivia
- Direct-Mapped Caches
- **Set Associative Caches**
- Cache Performance

Set Associative Caches

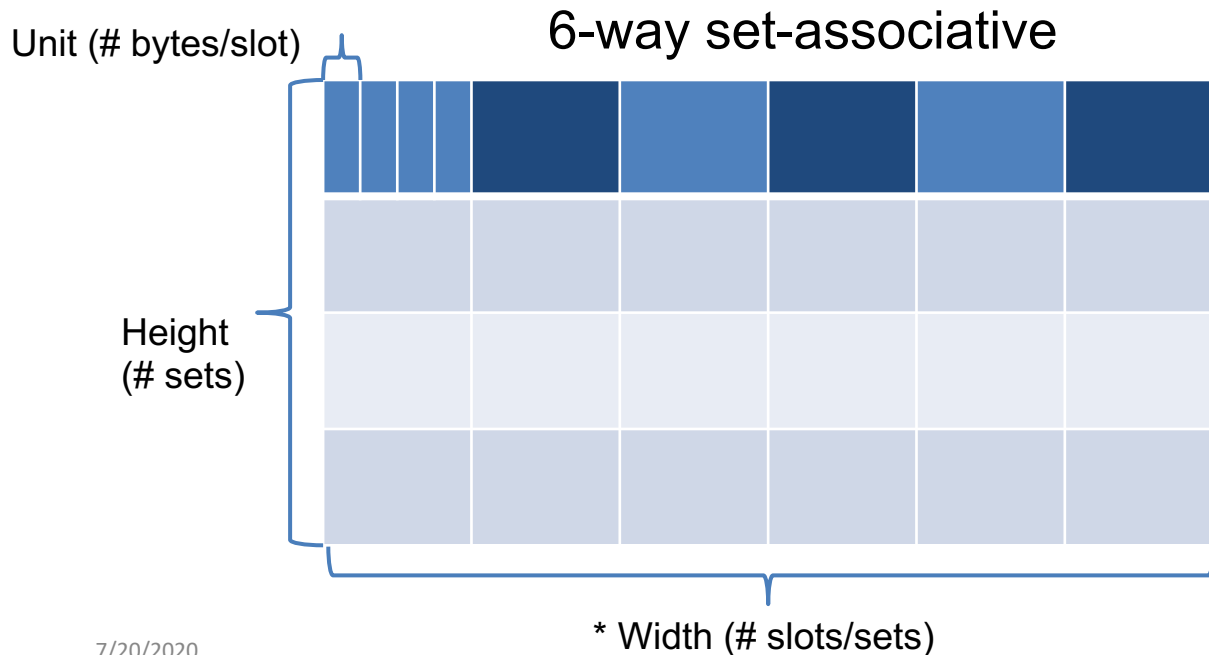
- Compromise!
 - More flexible than DM, more structured than FA
- *N-way set-associative*: Divide $\$$ into sets, each of which consists of N slots
 - Memory block maps to a set determined by **Index** field and is placed in any of the N slots of that set
 - Call N the *associativity*
 - Replacement 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 (1/2)

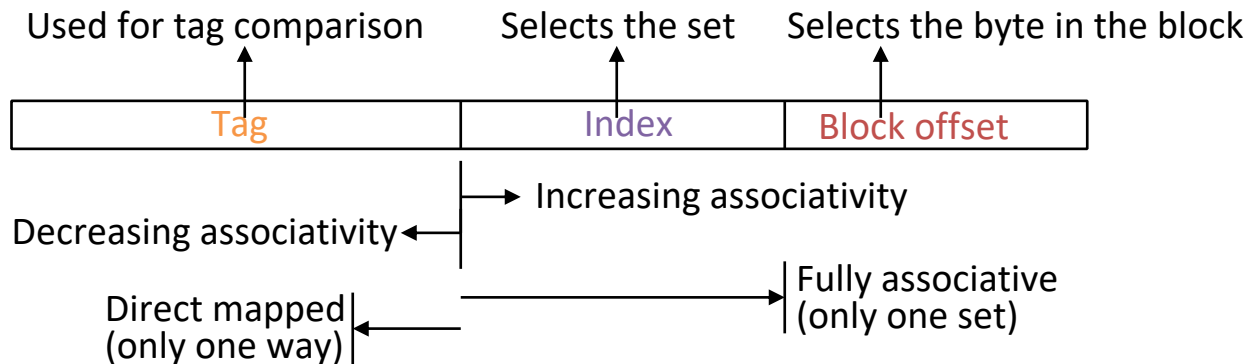
$$\begin{aligned} \text{Cache Area} &= \overset{C}{\text{Height (\# of sets)}} * \overset{N}{\text{Width (\# slots/sets)}} * \overset{K}{\text{Unit Area (\# bytes/slot)}} \\ &= 4 \text{ sets} * 6(\text{way/set}) * 4 \text{ bytes/slot} = 96 \text{ bytes} \end{aligned}$$



6-way
set-associative
4 bytes block
96 bytes cache

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 one slot
- 2-way set associative:** Can go in either slot of a set

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:
 - $O = \log_2(4) = 2$, $I = \log_2(2) = 1$, $T = 6 - 1 - 2 = 3$

Memory Addresses: 
Block address

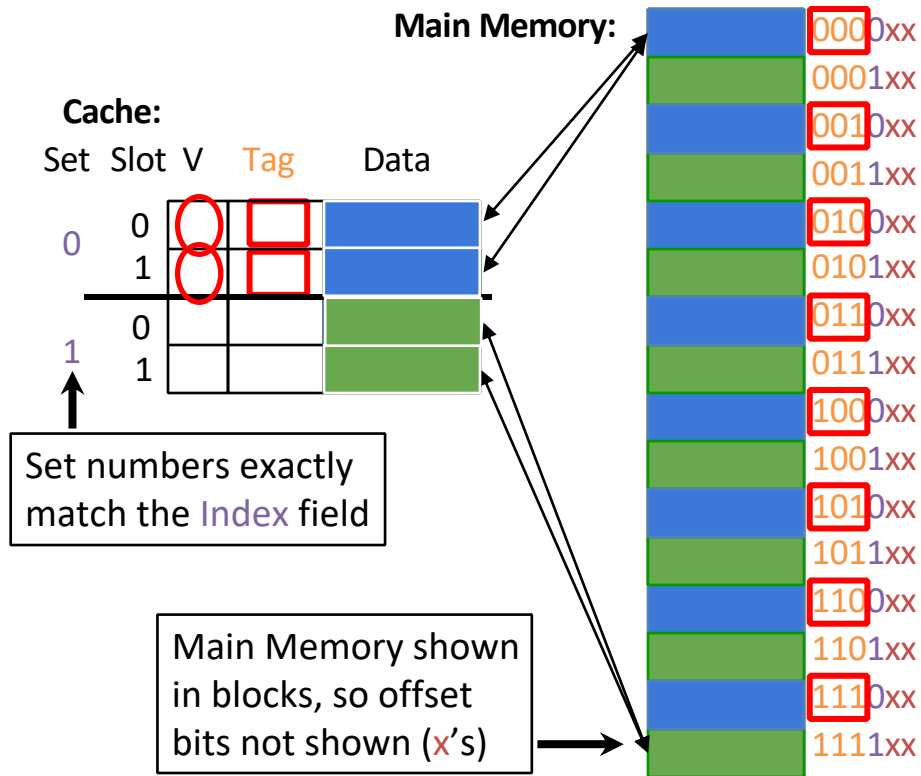
SA Cache Example (2/5)

- 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						Offset						LRU
		V	Tag	00	01	10	11	V	Tag	00	01	10	11	
Index	0	X	XXX	0x?	0x?	0x?	0x?	X	XXX	0x?	0x?	0x?	0x?	X
	1	X	XXX	0x?	0x?	0x?	0x?	X	XXX	0x?	0x?	0x?	0x?	LRU 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_{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??

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:

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]

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]

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]

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

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

	T	I	O
(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