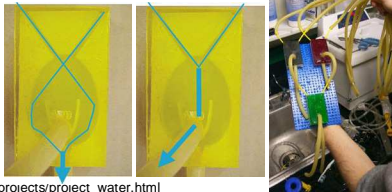


Lecture #23 – Cache II

2008-7-30

Water circuits

"If you think programming a computer is hard, just imagine what it would be if your bits were leaking all over the place."



http://www.blikstein.com/paulo/projects/project_water.html

Albert Chae, Instructor



CS61C L23 Cache II (1)

Chae, Summer 2008 © UCB

Review

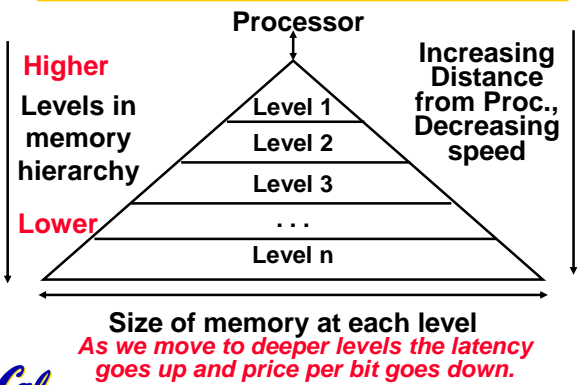
- Pipeline challenge is hazards
 - Forwarding helps w/many data hazards
 - Delayed branch helps with control hazard in 5 stage pipeline
 - Load delay slot / interlock necessary
- More aggressive performance:
 - Superscalar
 - Out-of-order execution
- Use caches to simulate fast large memory



CS61C L23 Cache II (2)

Chae, Summer 2008 © UCB

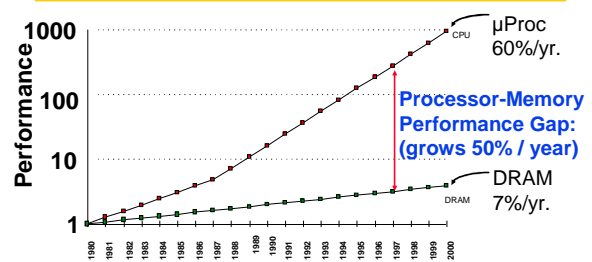
Memory Hierarchy



CS61C L23 Cache II (3)

Chae, Summer 2008 © UCB

Motivation: Why We Use Caches (written \$)



- 1989 first Intel CPU with cache on chip
- 1998 Pentium III has two levels of cache on chip



CS61C L23 Cache II (4)

Chae, Summer 2008 © UCB

Cache Design

- How do we organize cache?
- Where does each memory address map to?
(Remember that cache is subset of memory, so multiple memory addresses map to the same cache location.)
- How do we know which elements are in cache?
- How do we quickly locate them?



CS61C L23 Cache II (5)

Chae, Summer 2008 © UCB

Direct-Mapped Cache (1/4)

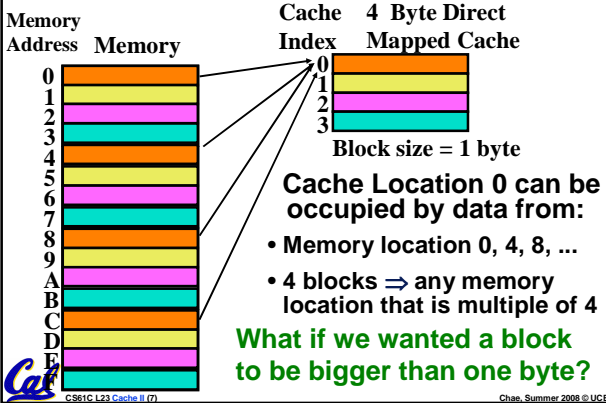
- In a **direct-mapped cache**, each memory address is associated with one possible **block** within the cache
 - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
 - Block is the unit of transfer between cache and memory



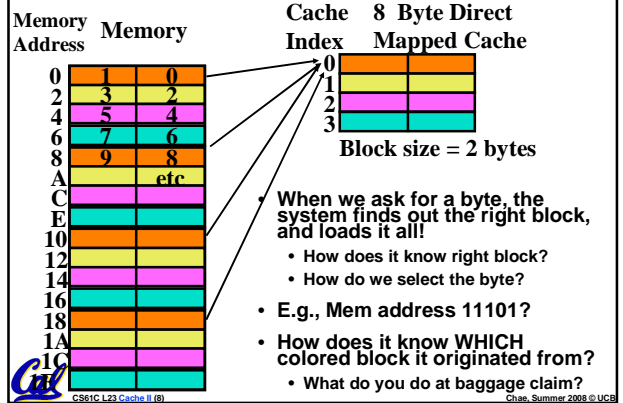
CS61C L23 Cache II (6)

Chae, Summer 2008 © UCB

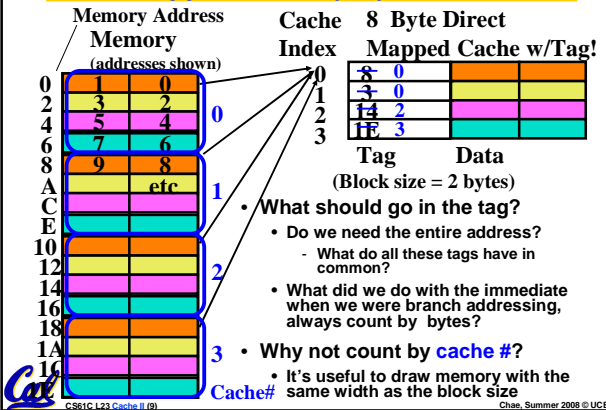
Direct-Mapped Cache (2/4)



Direct-Mapped Cache (3/4)

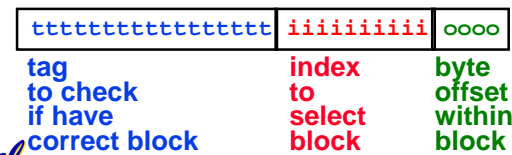


Direct-Mapped Cache (4/4)



Issues with Direct-Mapped

- Since multiple memory addresses map to same cache index, how do we tell which one is in there?
- What if we have a block size > 1 byte?
- Answer: divide memory address into three fields



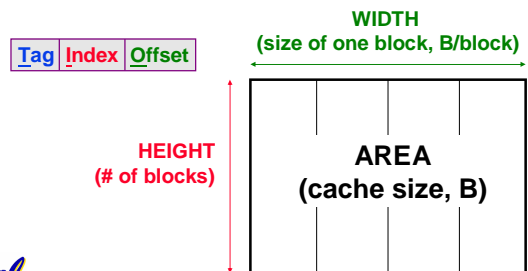
Direct-Mapped Cache Terminology

- All fields are read as unsigned integers.
 - **Index:** specifies the cache index (which "row"/block of the cache we should look in)
 - **Offset:** once we've found correct block, specifies which byte within the block we want
 - **Tag:** 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
- CS61C L23 Cache II (11) Chae, Summer 2008 © UCB

TIO Dan's great cache mnemonic

AREA (cache size, B) = HEIGHT (# of blocks) * WIDTH (size of one block, B/block)

$$2^{(H+W)} = 2^H * 2^W$$



Direct-Mapped Cache Example (1/3)

- Suppose we have a 16KB of data in a direct-mapped cache with 4 word blocks
- Determine the size of the tag, index and offset fields if we're using a 32-bit architecture
- Offset
 - need to specify correct byte within a block
 - block contains 4 words
 - = 16 bytes
 - = 2^4 bytes
- need **4 bits** to specify correct byte



CS61C L23 Cache II (13)

Chae, Summer 2008 © UCB

Direct-Mapped Cache Example (2/3)

- Index: (~index into an "array of blocks")
 - need to specify correct block in cache
 - cache contains 16 KB = 2^{14} bytes
 - block contains 2^4 bytes (4 words)
 - # blocks/cache
 - = $\frac{\text{bytes/cache}}{\text{bytes/block}}$
 - = $\frac{2^{14} \text{ bytes/cache}}{2^4 \text{ bytes/block}}$
 - = 2^{10} blocks/cache
 - need **10 bits** to specify this many blocks



CS61C L23 Cache II (14)

Chae, Summer 2008 © UCB

Direct-Mapped Cache Example (3/3)

- Tag: use remaining bits as tag
 - tag length = addr length - offset - index
 - = 32 - 4 - 10 bits
 - = 18 bits
 - so tag is leftmost **18 bits** of memory address
- Why not full 32 bit address as tag?
 - All bytes within block need same address (4b)
 - Index must be same for every address within a block, so it's redundant in tag check, thus can leave off to save memory (here 10 bits)



CS61C L23 Cache II (15)

Chae, Summer 2008 © UCB

Caching Terminology

- When we try to read memory, 3 things can happen:
 1. **cache hit:** cache block is valid and contains proper address, so read desired word
 2. **cache miss:** nothing in cache in appropriate block, so fetch from memory
 3. **cache miss, block replacement:** wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)

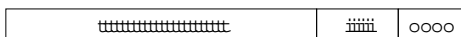


CS61C L23 Cache II (16)

Chae, Summer 2008 © UCB

Peer instruction

- Consider an address split into fields for cache access as follows:



- How big are the cache blocks in words?
 - 4 8 16 32
- How many entries does the cache have?
 - 4 8 16 32
- How big is a cache entry?
 - 24 33 55 56



CS61C L23 Cache II (17)

Chae, Summer 2008 © UCB

Accessing data in a direct mapped cache

- Ex.: 16KB of data, direct-mapped, 4 word blocks

Address (hex)	Value of Word
00000010	a
00000014	b
00000018	c
0000001C	d

- Read 4 addresses

1. 0x00000014	00000030	e
2. 0x0000001C	00000034	f
3. 0x00000034	00000038	g
4. 0x000008014	0000003C	h

- Memory vals here:

00008010	i
00008014	j
00008018	k
0000801C	l



CS61C L23 Cache II (18)

Chae, Summer 2008 © UCB

Accessing data in a direct mapped cache

- 4 Addresses:

- 0x00000014, 0x0000001C, 0x00000034, 0x000008014

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

```

00000000000000000000 0000000001 0100
00000000000000000000 0000000001 1100
00000000000000000000 0000000011 0100
00000000000000000010 0000000001 0100
    
```

Tag
Index
Offset



16 KB Direct Mapped Cache, 16B blocks

- **Valid bit:** determines whether anything is stored in that row (when computer initially turned on, all entries invalid)

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



First Type of Cache Miss

- “Three Cs” Model of Misses

- **1st C: Compulsory 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



1. Read 0x00000014

- 00000000000000000000 0000000001 0100
- Tag field
Index field
Offset

Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
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 (000000001)

- 00000000000000000000 0000000001 0100
- Tag field
Index field
Offset

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



No valid data

- 00000000000000000000 0000000001 0100
- Tag field
Index field
Offset

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



So load that data into cache, setting tag, valid

- 00000000000000000000 0000000001 0100

Valid Tag field Index field Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					

...

1022	0					
1023	0					

Cal CS61C L23 Cache II (25) Chae, Summer 2008 © UC Berkeley

Read from cache at offset, return word b

- 00000000000000000000 0000000001 0100

Valid Tag field Index field Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					

...

1022	0					
1023	0					

Cal CS61C L23 Cache II (26) Chae, Summer 2008 © UC Berkeley

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

- 00000000000000000000 0000000001 1100

Valid Tag field Index field Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					

...

1022	0					
1023	0					

Cal CS61C L23 Cache II (27) Chae, Summer 2008 © UC Berkeley

Index is Valid

- 00000000000000000000 0000000001 1100

Valid Tag field Index field Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					

...

1022	0					
1023	0					

Cal CS61C L23 Cache II (28) Chae, Summer 2008 © UC Berkeley

Index valid, Tag Matches

- 00000000000000000000 0000000001 1100

Valid Tag field Index field Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					

...

1022	0					
1023	0					

Cal CS61C L23 Cache II (29) Chae, Summer 2008 © UC Berkeley

Index Valid, Tag Matches, return d

- 00000000000000000000 0000000001 1100

Valid Tag field Index field Offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					

...

1022	0					
1023	0					

Cal CS61C L23 Cache II (30) Chae, Summer 2008 © UC Berkeley

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

• 000000000000000000000000 0000000011 0100

Valid Tag field Index field Offset

Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3	
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					

...

1022	0				
1023	0				

Cal CS61C L23 Cache II (31) Chae, Summer 2008 © UCB

So read block 3

• 000000000000000000000000 0000000011 0100

Valid Tag field Index field Offset

Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3	
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					

...

1022	0				
1023	0				

Cal CS61C L23 Cache II (32) Chae, Summer 2008 © UCB

No valid data

• 000000000000000000000000 0000000011 0100

Valid Tag field Index field Offset

Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3	
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					

...

1022	0				
1023	0				

Cal CS61C L23 Cache II (33) Chae, Summer 2008 © UCB

Load that cache block, return word f

• 000000000000000000000000 0000000011 0100

Valid Tag field Index field Offset

Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3	
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					

...

1022	0				
1023	0				

Cal CS61C L23 Cache II (34) Chae, Summer 2008 © UCB

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

• 000000000000000000000010 0000000001 0100

Valid Tag field Index field Offset

Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3	
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					

...

1022	0				
1023	0				

Cal CS61C L23 Cache II (35) Chae, Summer 2008 © UCB

So read Cache Block 1, Data is Valid

• 000000000000000000000010 0000000001 0100

Valid Tag field Index field Offset

Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3	
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					

...

1022	0				
1023	0				

Cal CS61C L23 Cache II (36) Chae, Summer 2008 © UCB

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

- 0000000000000000010 0000000001 0100

Valid	Tag field	Index field	Offset			
Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3	
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					

CS61C L23 Cache II (37) Chae, Summer 2008 ©UCB

Miss, so replace block 1 with new data & tag

- 0000000000000000010 0000000001 0100

Valid	Tag field	Index field	Offset			
Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3	
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					

CS61C L23 Cache II (38) Chae, Summer 2008 ©UCB

And return word J

- 0000000000000000010 0000000001 0100

Valid	Tag field	Index field	Offset			
Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3	
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					

CS61C L23 Cache II (39) Chae, Summer 2008 ©UCB

Do an example yourself. What happens?

- Chose from: Cache: Hit, Miss, Miss w. replace
Values returned: a ,b, c, d, e, ..., k, l
- Read address 0x00000030 ?
00000000000000000000 0000000011 0000
- Read address 0x0000001c ?
00000000000000000000 0000000001 1100

Cache

Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f	
Index	Tag					
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...						

CS61C L23 Cache II (40) Chae, Summer 2008 ©UCB

Answers

- 0x00000030 a **hit**
Index = 3, Tag matches, Offset = 0, value = e
- 0x0000001c a **miss**, so replace from memory, Offset = 0xc, value = d
- Since reads, values must = memory values whether or not cached:

Address (hex)	Value of Word
00000010	a
00000014	b
00000018	c
0000001c	d
...	...
00000030	e
00000034	f
00000038	g
0000003c	h
...	...
00008010	i
00008014	j
00008018	k
0000801c	l
...	...

CS61C L23 Cache II (41) Chae, Summer 2008 ©UCB

Administrivia

- Quiz 9 due TODAY 7/30
- Quiz 12 due Friday 8/1
- HW6 due Friday 8/1
- Proj3 out soon, due next Tuesday 8/5
 - Will be hand graded in person, signups will be posted soon
- Drop or grading option deadline
 - August 1
 - summer.berkeley.edu for more details
- Final 8/14 – 9:30-12:30pm in 105 North Gate
- 61A/B conflicts? Talk to me at end of class.

CS61C L23 Cache II (42) Chae, Summer 2008 ©UCB

What to do on a write hit?

- **Write-through**

- update the word in cache block and corresponding word in memory

- **Write-back**

- update word in cache block
- allow memory word to be "stale"

⇒ add 'dirty' bit to each block indicating that memory needs to be updated when block is replaced

⇒ OS flushes cache before I/O...

- Performance trade-offs?



Block Size Tradeoff (1/3)

- Benefits of Larger Block Size

- **Spatial Locality**: if we access a given word, we're likely to access other nearby words soon

- Very applicable with Stored-Program Concept: if we execute a given instruction, it's likely that we'll execute the next few as well

- Works nicely in sequential array accesses too



Block Size Tradeoff (2/3)

- Drawbacks of Larger Block Size

- Larger block size means **larger miss penalty**
 - on a miss, takes longer time to load a new block from next level
- If block size is too big relative to cache size, then there are too few blocks
 - Result: miss rate goes up

- In general, minimize **Average Memory Access Time (AMAT)**

$$= \text{Hit Time} + \text{Miss Penalty} \times \text{Miss Rate}$$



Block Size Tradeoff (3/3)

- **Hit Time** = time to find and retrieve data from current level cache

- **Miss Penalty** = average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)

- **Hit Rate** = % of requests that are found in current level cache

- **Miss Rate** = 1 - Hit Rate



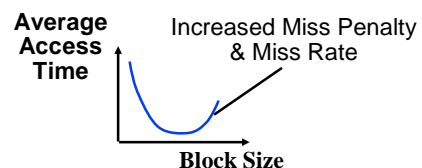
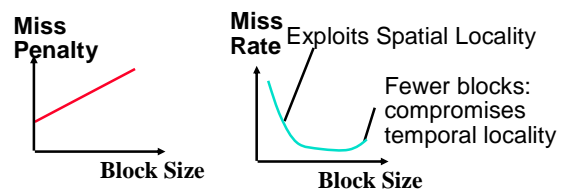
Extreme Example: One Big Block

Valid Bit	Tag	Cache Data
<input type="checkbox"/>		B3 B2 B1 B0

- Cache Size = 4 bytes Block Size = 4 bytes
 - Only **ONE** entry (row) in the cache!
- If item accessed, likely accessed again soon
 - But unlikely will be accessed again immediately!
- The next access will likely to be a miss again
 - Continually loading data into the cache but discard data (force out) before use it again
 - Nightmare for cache designer: **Ping Pong Effect**



Block Size Tradeoff Conclusions



Second Type of Cache Miss

• 2nd C: Conflict Misses

- miss that occurs because two distinct memory addresses map to the same cache location
- two blocks (which happen to map to the same location) can keep overwriting each other
- big problem in direct-mapped caches
- how do we lessen the effect of these?

• 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?



CS61C L23 Cache II (49)

Chae, Summer 2008 © UCB

Fully Associative Cache (1/3)

• Memory address fields:

- Tag: same as before
- Offset: same as before
- Index: non-existent

• What does this mean?

- no “rows”: any block can go anywhere in the cache
- must compare with all tags in entire cache to see if data is there



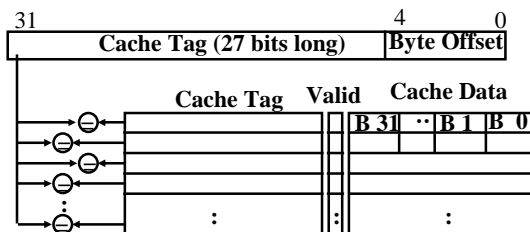
CS61C L23 Cache II (50)

Chae, Summer 2008 © UCB

Fully Associative Cache (2/3)

• Fully Associative Cache (e.g., 32 B block)

- compare tags in parallel



CS61C L23 Cache II (51)

Chae, Summer 2008 © UCB

Fully Associative Cache (3/3)

• Benefit of Fully Assoc Cache

- No Conflict Misses (since data can go anywhere)

• Drawbacks of Fully Assoc 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



CS61C L23 Cache II (52)

Chae, Summer 2008 © UCB

Third Type of Cache Miss

• 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
- sketchy definition, so just get the general idea
- This is the primary type of miss for Fully Associative caches.



CS61C L23 Cache II (53)

Chae, Summer 2008 © UCB

N-Way Set Associative Cache (1/3)

• Memory address fields:

- Tag: same as before
- Offset: same as before
- Index: points us to the correct “row” (called a **set** in this case)

• So what’s the difference?

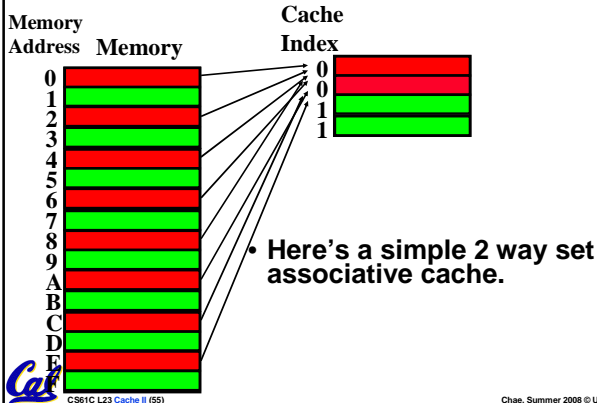
- each set contains multiple blocks
- once we’ve found correct set, must compare with all tags in that set to find our data



CS61C L23 Cache II (54)

Chae, Summer 2008 © UCB

Associative Cache Example



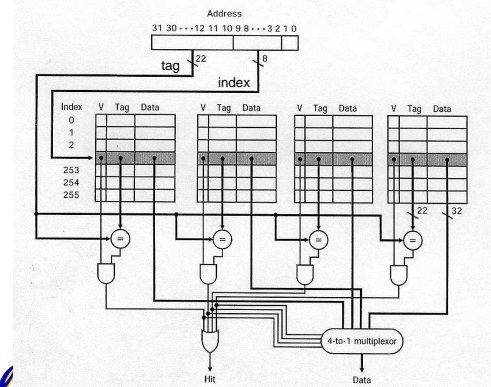
N-Way Set Associative Cache (2/3)

- **Basic Idea**
 - cache is direct-mapped w/respect to sets
 - each set is fully associative
 - basically N direct-mapped caches working in parallel: each has its own valid bit and data
 - **Given memory address:**
 - Find correct set using Index value.
 - Compare Tag with all Tag values in the determined set.
 - If a match occurs, hit!, otherwise a miss.
 - Finally, use the offset field as usual to find the desired data within the block.
- CS61C L23 Cache II (60) Chae, Summer 2008 © UC Berkeley

N-Way Set Associative Cache (3/3)

- **What's so great about this?**
 - even a 2-way set assoc cache avoids a lot of conflict misses
 - hardware cost isn't that bad: only need N comparators
 - **In fact, for a cache with M blocks,**
 - it's Direct-Mapped if it's 1-way set assoc
 - it's Fully Assoc if it's M-way set assoc
 - so these two are just special cases of the more general set associative design
- CS61C L23 Cache II (57) Chae, Summer 2008 © UC Berkeley

4-Way Set Associative Cache Circuit



Peer Instruction

- Mem hierarchies were invented before 1950. (UNIVAC I wasn't delivered 'til 1951)
- If you know your computer's cache size, you can often make your code run faster.
- Memory hierarchies take advantage of spatial locality by keeping the most recent data items closer to the processor.

0:	ABC
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTT
7:	TTT

CS61C L23 Cache II (59) Chae, Summer 2008 © UC Berkeley

Peer Instruction Answer

- "We are...forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less accessible." – von Neumann, 1946
 - Certainly! That's call "tuning"
 - "Most Recent" items ⇒ Temporal locality
- Mem hierarchies were invented before 1950. (UNIVAC I wasn't delivered 'til 1951)
 - If you know your computer's cache size, you can often make your code run faster.
 - Memory hierarchies take advantage of spatial locality by keeping the most recent data items closer to the processor.

0:	ABC
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTT
7:	TTT

CS61C L23 Cache II (60) Chae, Summer 2008 © UC Berkeley

