CS61CL : Machine Structures

Lecture #11 – Caches

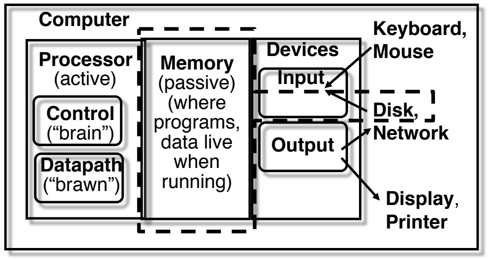
2009-07-29

Jeremy Huddleston

Review : Pipelining

- **Pipeline challenge is hazards**
  - **Forwarding helps w/many data hazards**
  - **Delayed branch helps with control hazard in our 5 stage pipeline**
  - **Data hazards w/Loads ⇒ Load Delay Slot**
    - **Interlock ⇒ "smart" CPU has HW to detect if conflict with inst following load, if so it stalls**
- **More aggressive performance (discussed in section next week)**
  - **Superscalar (parallelism)**
  - **Out-of-order execution**

The Big Picture



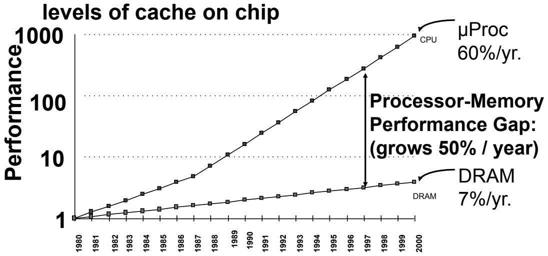
Memory Hierarchy

I.e., storage in computer systems

- **Processor**
  - **holds data in register file (~100 Bytes)**
  - **Registers accessed on nanosecond timescale**
- **Memory (we'll call "main memory")**
  - **More capacity than registers (~Gbytes)**
  - **Access time ~50-100 ns**
  - **Hundreds of clock cycles per memory access?!**
- **Disk**
  - **HUGE capacity (virtually limitless)**
  - **VERY slow: runs ~milliseconds**

Motivation: Why We Use Caches (written $)

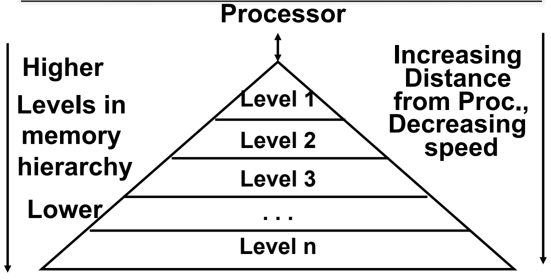
- **1989 first Intel CPU (80486) with cache on chip**
- **1995 first Intel CPU (Pentium Pro) with two levels of cache on chip**



Memory Caching

- **Mismatch between processor and memory speeds leads us to add a new level: a memory cache**
- **Implemented with same IC processing technology as the CPU (usually integrated on same chip): faster but more expensive than DRAM memory.**
- **Cache is a copy of a subset of main memory.**
- **Most processors have separate caches for instructions and data.**

Memory Hierarchy



As we move to deeper levels the latency goes up and price per bit goes down.

Memory Hierarchy

- **If level closer to Processor, it is:**
  - **Smaller**
  - **Faster**
  - **More expensive**
  - **subset of lower levels (contains most recently used data)**
- **Lowest Level (usually disk) contains all available data (does it go beyond the disk?)**
- **Memory Hierarchy presents the processor with the illusion of a very large & fast memory**

Memory Hierarchy Analogy: Library (1/2)

- **You're writing a term paper (Processor) at a table in Doe**
- **Doe Library is equivalent to disk**
  - **essentially limitless capacity**
  - **very slow to retrieve a book**
- **Table is main memory**
  - **smaller capacity: means you must return book when table fills up**
  - **easier and faster to find a book there once you've already retrieved it**

Memory Hierarchy Analogy: Library (2/2)

- Open books on table are cache
  - smaller capacity: can have very few open books fit on table; again, when table fills up, you must close a book
  - much, much faster to retrieve data
- Illusion created: whole library open on the tabletop
  - Keep as many recently used books open on table as possible since likely to use again
  - Also keep as many books on table as possible, since faster than going to library

Memory Hierarchy Basis

- Cache contains copies of data in memory that are being used.
- Memory contains copies of data on disk that are being used.
- Caches work on the principles of temporal and spatial locality.
  - Temporal Locality: if we use it now, chances are we'll want to use it again soon.
  - Spatial Locality: if we use a piece of memory, chances are we'll use the neighboring pieces soon.

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?

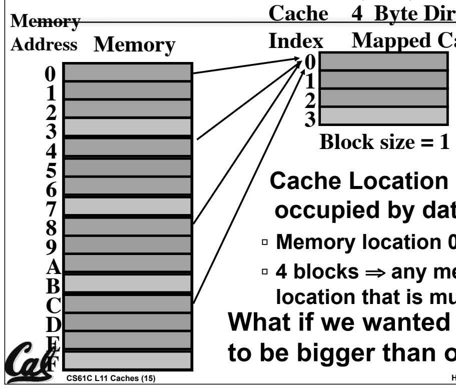
Administrivia

- Project 4 (on Caches) will be in optional groups of two.
- Jeremy's OH today canceled
  - I will have OH on Friday, time will be posted on the newsgroup
- HW7 due tomorrow
  - You MUST have a discussion with your TA in lab tomorrow for credit

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

Direct-Mapped Cache (2/4)

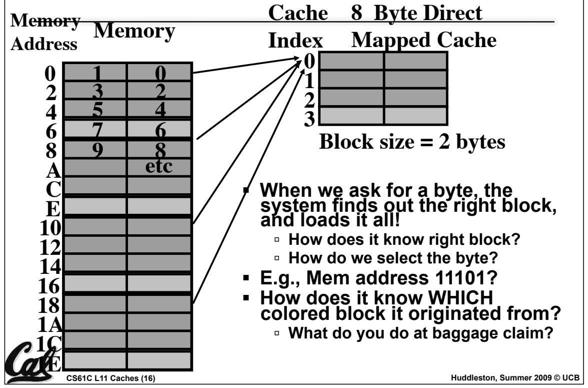


Block size = 1 byte

Cache Location 0 can be occupied by data from:
- Memory location 0, 4, 8, ...
- 4 blocks ⇒ any memory location that is multiple of 4

What if we wanted a block to be bigger than one byte?

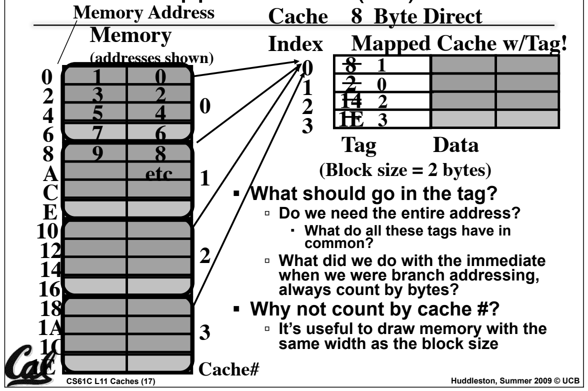
Direct-Mapped Cache (3/4)



Block size = 2 bytes

- When we ask for a byte, the system finds out the right block, and loads it all!
  - How does it know right block?
  - How do we select the byte?
- E.g., Mem address 11101?
- How does it know WHICH colored block it originated from?
  - What do you do at baggage claim?

Direct-Mapped Cache (4/4)



(Block size = 2 bytes)

- What should go in the tag?
  - Do we need the entire address?
    - What do all these tags have in common?
  - What did we do with the immediate when we were branch addressing, always count by bytes?
- Why not count by cache #?
  - It's useful to draw memory with the same width as the block size

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

tttttttttttttttttt	iiiiiiiiii	oooo
tag to check if have correct block	index to select block	byte offset within block

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

---

Direct-Mapped Cache Example (1/3)

- Suppose we have a 8B of data in a direct-mapped cache with 2 byte blocks
  - Sound familiar?
- 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 2 bytes
    $= 2^1$ bytes
  - need 1 bit to specify correct byte

Direct-Mapped Cache Example (2/3)

- Index: (~index into an “array of blocks”)
  - need to specify correct block in cache
  - cache contains 8 B = 2^3 bytes
  - block contains 2 B = 2^1 bytes
  - # blocks/cache

$$= \frac{\text{bytes/cache}}{\text{bytes/block}}$$

$$= \frac{2^3 \text{ bytes/cache}}{2^1 \text{ bytes/block}}$$

$$= 2^2 \text{ blocks/cache}$$

  - need 2 bits to specify this many blocks

Direct-Mapped Cache Example (3/3)

- Tag: use remaining bits as tag
  - tag length = addr length - offset - index
    = 32 - 1 - 2 bits
    = 29 bits
  - so tag is leftmost 29 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)

Caching Terminology

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

---

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)

Index	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					

---

1. Read 0x00000014

- `000000000000000000 0000000001 0100`
  Tag field Index field Offset

Index	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 (0000000001)

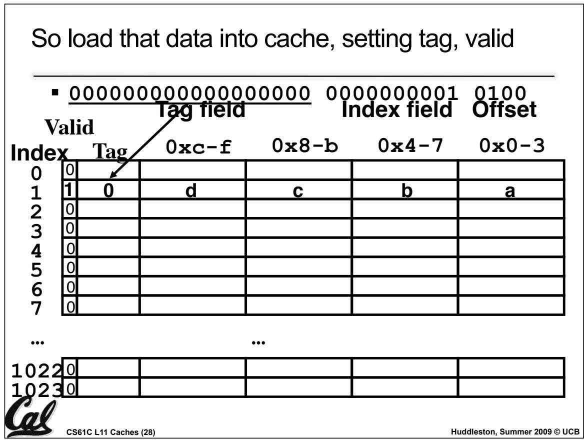
- `000000000000000000 0000000001 0100`
  Tag field Index field Offset

Index	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

- `000000000000000000 0000000001 0100`
  Tag field Index field Offset

Index	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					

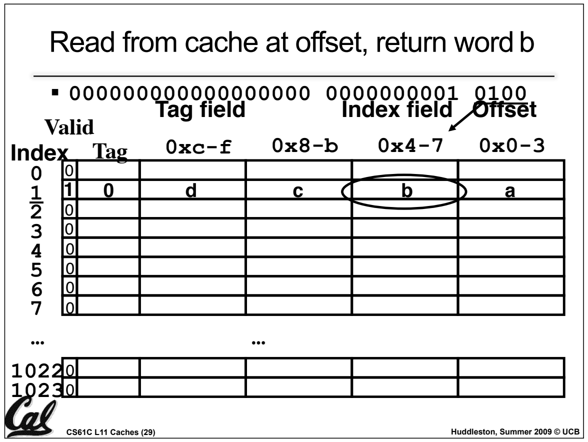


So load that data into cache, setting tag, valid

- 000000000000000000 0000000001 0100

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					

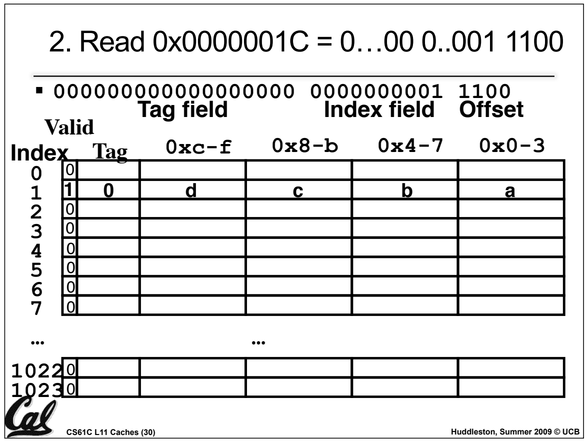


Read from cache at offset, return word b

- 000000000000000000 0000000001 0100

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					

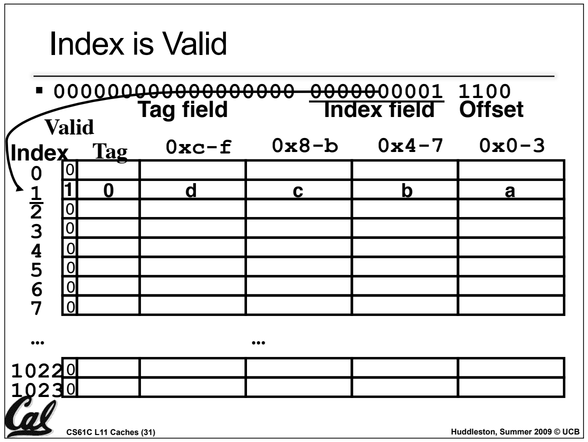


2. Read 0x0000001C = 0…00 0..001 1100

- 000000000000000000 0000000001 1100

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					

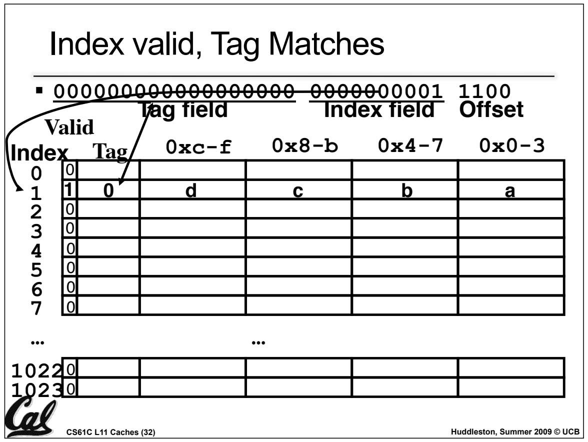


Index is Valid

- 000000000000000000 0000000001 1100

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					

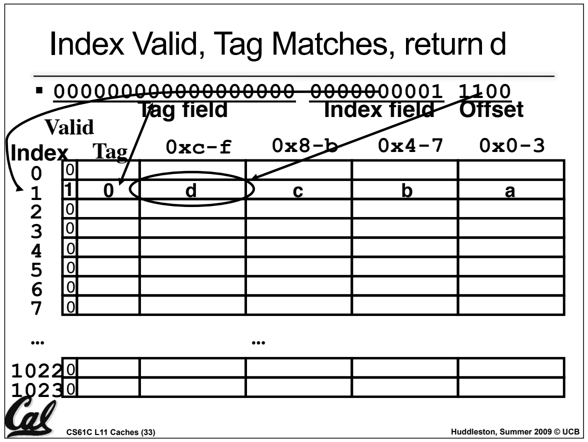


Index valid, Tag Matches

- 000000000000000000 0000000001 1100

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					

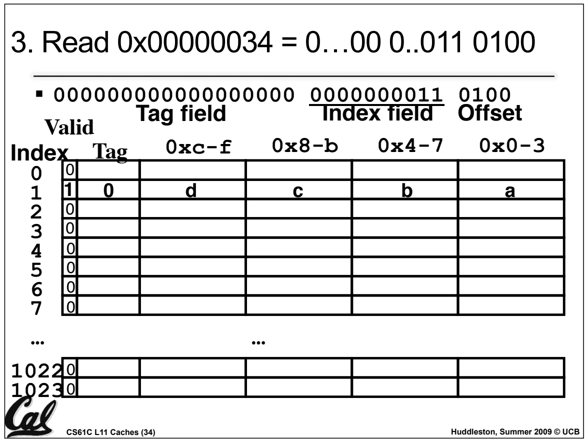


Index Valid, Tag Matches, return d

- 000000000000000000 0000000001 1100

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					

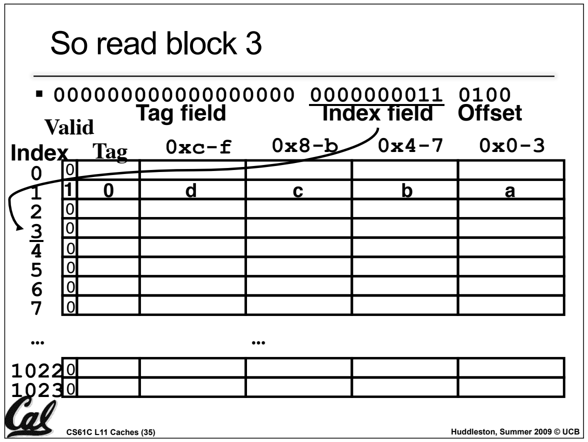


3. Read 0x00000034 = 0…00 0..011 0100

- 000000000000000000 0000000011 0100

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					

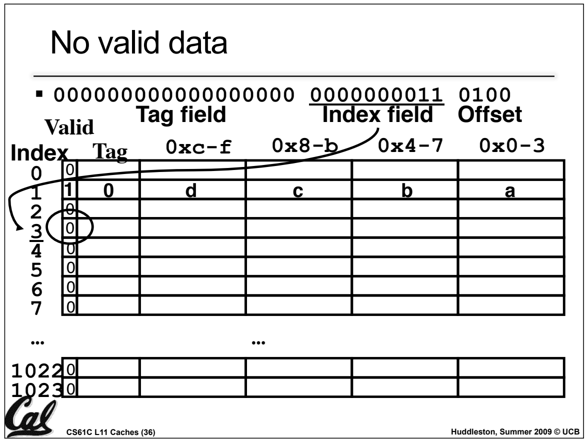


So read block 3

- 000000000000000000 0000000011 0100

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					



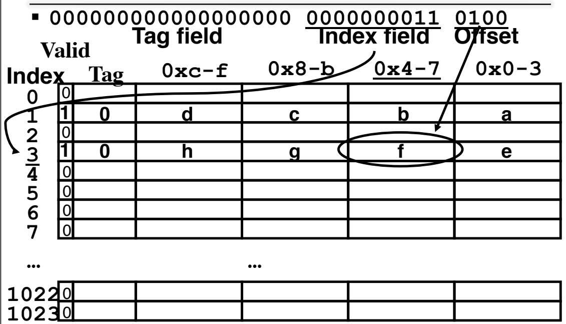
No valid data

- 000000000000000000 0000000011 0100

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					

Load that cache block, return word f



- 000000000000000000 0000000011 0100

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	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

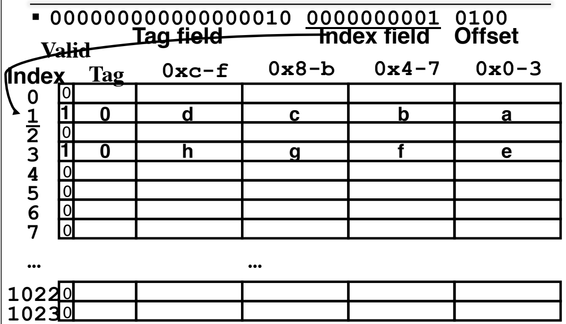
4. Read 0x00008014 = 0…10 0..001 0100

- 000000000000000010 0000000001 0100

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	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

So read Cache Block 1, Data is Valid

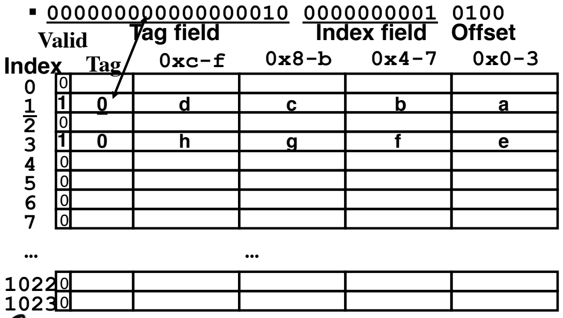


- 000000000000000010 0000000001 0100

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	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

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



- 000000000000000010 0000000001 0100

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	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

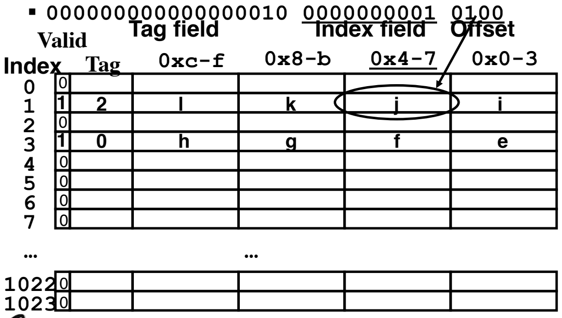
Miss, so replace block 1 with new data & tag

- 000000000000000010 0000000001 0100

Tag field **Index field** **Offset**

Index	Valid	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					

And return word J



- 000000000000000010 0000000001 0100

Tag field **Index field** **Offset**

Index	Valid	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					

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?**

Types of Cache Misses (1/2)

- **“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

Types of Cache Misses (2/2)

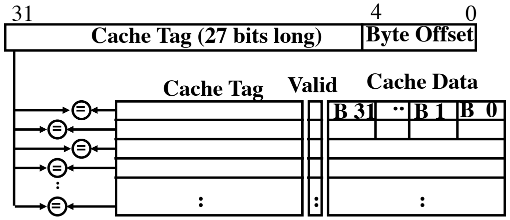
- **2nd C: Conflict Misses**
  - miss that occurs because two distinct memory addresses map to the same cache location
  - two blocks (which happen to map to the same location) can keep overwriting each other
  - big problem in direct-mapped caches
  - how do we lessen the effect of these?
- **Dealing 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?

Fully Associative Cache (1/3)

- **Memory address fields:**
  - Tag: same as before
  - Offset: same as before
  - Index: non-existant
- **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

Fully Associative Cache (2/3)

- **Fully Associative Cache (e.g., 32 B block)**
  - compare tags in parallel



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

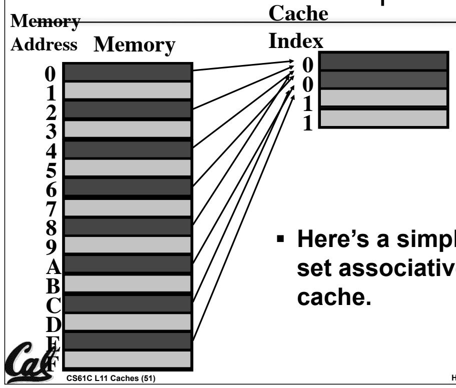
Final Type of Cache Miss

- **3rd C: 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.**

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

Associative Cache Example



- **Here’s a simple 2-way set associative cache.**

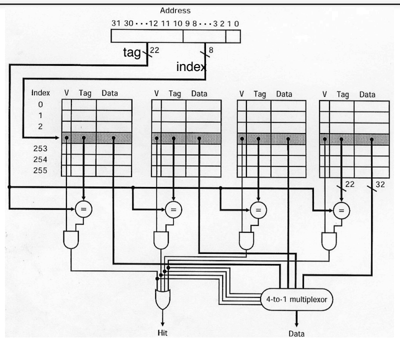
N-Way Set Associative Cache (2/3)

- **Basic Idea**
  - cache is direct-mapped w/respect to sets
  - each set is fully associative with N blocks in it
- **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.

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

4-Way Set Associative Cache Circuit



Block Replacement Policy

- **Direct-Mapped Cache**
  - index completely specifies position which position a block can go in on a miss
- **N-Way Set Assoc**
  - index specifies a set, but block can occupy any position within the set on a miss
- **Fully Associative**
  - block can be written into any position
- **Question: if we have the choice, where should we write an incoming block?**
  - If there are any locations with valid bit off (empty), then usually write the new block into the first one.
  - If all possible locations already have a valid block, we must pick a replacement policy: rule by which we determine which block gets “cached out” on a miss.

Block Replacement Policy: LRU

- **LRU (Least Recently Used)**
  - Idea: cache out block which has been accessed (read or write) least recently
  - Pro: temporal locality ⇒ recent past use implies likely future use: in fact, this is a very effective policy
  - Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires complicated hardware and much time to keep track of this

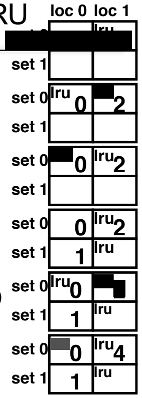
---

Block Replacement 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



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

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

0: <u>hit</u>

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

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

0: <u>hit</u>

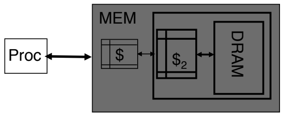
Addresses 0, 2, 0, 1, 4, 0, ...

Big Idea

- **How to choose between associativity, block size, replacement & write policy?**
- **Design against a performance model**
  - Minimize: Average Memory Access Time
    - = Hit Time
      - \+ Miss Penalty x Miss Rate
  - influenced by technology & program behavior
- **Create the illusion of a memory that is large, cheap, and fast - on average**
- **How can we improve miss penalty?**

Improving Miss Penalty

- **When caches first became popular, Miss Penalty ~ 10 processor clock cycles**
- **Today 2400 MHz Processor (0.4 ns per clock cycle) and 80 ns to go to DRAM ⇒ 200 processor clock cycles!**



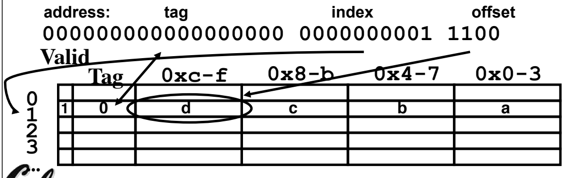
Solution: another cache between memory and the processor cache: <u>Second Level (L2) Cache</u>

And in Conclusion…

- **We would like to have the capacity of disk at the speed of the processor: unfortunately this is not feasible.**
- **So we create a memory hierarchy:**
  - each successively lower level contains “most used” data from next higher level
  - exploits temporal & spatial locality
  - do the common case fast, worry less about the exceptions (design principle of MIPS)
- **Locality of reference is a Big Idea**

And in Conclusion…

- **Mechanism for transparent movement of data among levels of a storage hierarchy**
  - set of address/value bindings
  - address ⇒ index to set of candidates
  - compare desired address with tag
  - service hit or miss
    - load new block and binding on miss



And in Conclusion…

- We’ve discussed memory caching in detail. Caching in general shows up over and over in computer systems
  - Filesystem cache, Web page cache, Game databases / tablebases, Software memoization, Others?
- Big idea: if something is expensive but we want to do it repeatedly, do it once and cache the result.
- Cache design choices:
  - Size of cache: speed v. capacity
  - Block size (i.e., cache aspect ratio)
  - Write Policy (Write through v. write back
  - Associativity choice of N (direct-mapped v. set v. fully associative)
  - Block replacement policy
  - 2nd level cache?
  - 3rd level cache?
- Use performance model to pick between choices, depending on programs, technology, budget, ...

Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

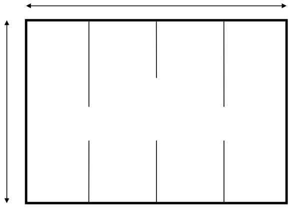
Bonus

TIO The great cache mnemonic

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

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

Tag Index Offset


WIDTH (size of one block, B/block)

HEIGHT (# of blocks)

Accessing data in a direct mapped cache

- Ex.: 16KB of data, direct-mapped, 4 word blocks
  - Can you work out height, width, area?
- Read 4 addresses
  1. `0x00000014`
  2. `0x0000001C`
  3. `0x00000034`
  4. `0x00008014`
- Memory vals here:

Memory

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
...	...

Accessing data in a direct mapped cache

- 4 Addresses:
  - `0x00000014, 0x0000001C, 0x00000034, 0x00008014`
- 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

```
000000000000000000 0000000001 0100
000000000000000000 0000000001 1100
000000000000000000 0000000011 0100
000000000000000010 0000000001 0100
        Tag             Index   Offset
```

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 ?**
  `000000000000000000 0000000011 0000`
- Read address **0x0000001c ?**
  `000000000000000000 0000000001 1100`

Cache Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
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					
...				...		

Answers

- `0x00000030` a <u>hit</u>
  Index = 3, Tag matches, Offset = 0, value = e
- `0x0000001c` a <u>miss</u>
  Index = 1, Tag mismatch, so replace from memory, Offset = `0xc`, value = d
- Since reads, values must = memory values whether or not cached:
  - `0x00000030` = e
  - `0x0000001c` = d

Memory

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
...	...

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

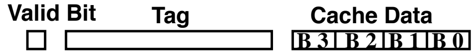
Block Size Tradeoff (2/3)

- 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)
  = Hit Time
  + Miss Penalty x 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

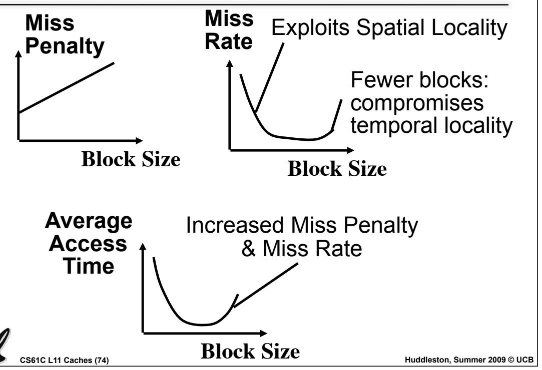
Extreme Example: One Big Block



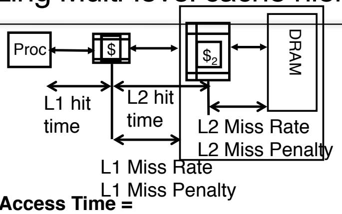
- Cache Size = 4 bytes Block Size = 4 bytes
  - Only ONE entry (row) in the cache!
- If item accessed, likely accessed again soon
  - But unlikely will be accessed again immediately!
- The next access will likely to be a miss again
  - Continually loading data into the cache but discard data (force out)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

---

Block Size Tradeoff Conclusions



Analyzing Multi-level cache hierarchy



Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate * L1 Miss Penalty

L1 Miss Penalty =
L2 Hit Time + L2 Miss Rate * L2 Miss Penalty

Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate *
(L2 Hit Time + L2 Miss Rate * L2 Miss Penalty)

Example

- Assume
  - Hit Time = 1 cycle
  - Miss rate = 5%
  - Miss penalty = 20 cycles
  - Calculate AMAT…
- Avg mem access time
  = 1 + 0.05 x 20
  = 1 + 1 cycles
  = 2 cycles

Ways to reduce miss rate

- Larger cache
  - limited by cost and technology
  - hit time of first level cache < cycle time (bigger caches are slower)
- More places in the cache to put each block of memory – associativity
  - fully-associative
    - any block any line
  - N-way set associated
    - N places for each block
    - direct map: N=1

Typical Scale

- L1
  - size: tens of KB
  - hit time: complete in one clock cycle
  - miss rates: 1-5%
- L2:
  - size: hundreds of KB
  - hit time: few clock cycles
  - miss rates: 10-20%
- L2 miss rate is fraction of L1 misses that also miss in L2
  - why so high?

Example: with L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L2 Hit Time = 5 cycles
  - L2 Miss rate = 15% (% L1 misses that miss)
  - L2 Miss Penalty = 200 cycles
- L1 miss penalty = 5 + 0.15 * 200 = 35
- Avg mem access time = 1 + 0.05 x 35
  = 2.75 cycles

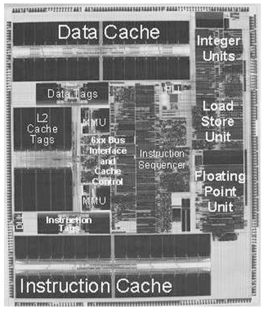
Example: without L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L1 Miss Penalty = 200 cycles
- Avg mem access time = 1 + 0.05 x 200
  = 11 cycles

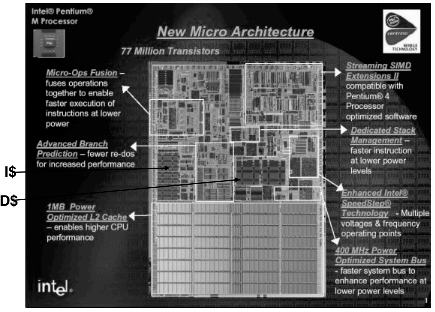
- 4x faster with L2 cache! (2.75 vs. 11)

An actual CPU – Early PowerPC

- Cache
  - 32 KB Instructions and 32 KB Data L1 caches
  - External L2 Cache interface with integrated controller and cache tags, supports up to 1 MByte external L2 cache
  - Dual Memory Management Units (MMU) with Translation Lookaside Buffers (TLB)
- Pipelining
  - Superscalar (3 inst/cycle)
  - 6 execution units (2 integer and 1 double precision IEEE floating point)



An Actual CPU – Pentium M



Intel® Pentium® M Processor

New Micro Architecture

77 Million Transistors

Micro-Ops Fusion – fuses operations together to enable faster execution of instructions at lower power

Advanced Branch Prediction – fewer re-dos for increased performance

1MB Power Optimized L2 Cache – enables higher CPU performance

Streaming SIMD Extensions II compatible with Pentium® 4 Processor optimized software

Dedicated Stack Management – faster instruction at lower power levels

Enhanced Intel® SpeedStep® Technology - Multiple voltages & frequency operating points

400 MHz Power Optimized System Bus - faster system bus to enhance performance at lower power levels

32KB I$

32KB D$