CMPE110 Lecture 14
Caches II

Heiner Litz

https://canvas.ucsc.edu/courses/12652

Announcements

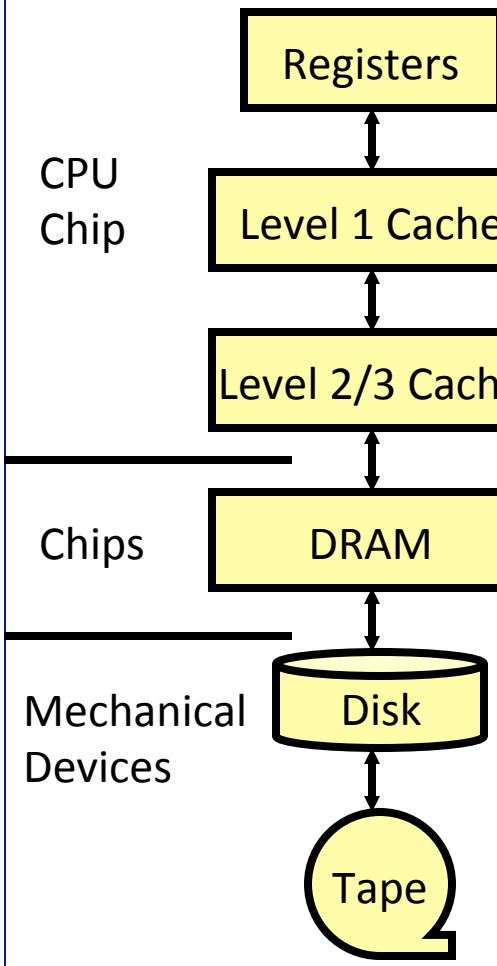
- Midterm has been moved to Monday May 14th
- Review session on Friday May 11th
- Regular lecture on May 4th
- Quiz on next Friday May 4th

Review

Locality

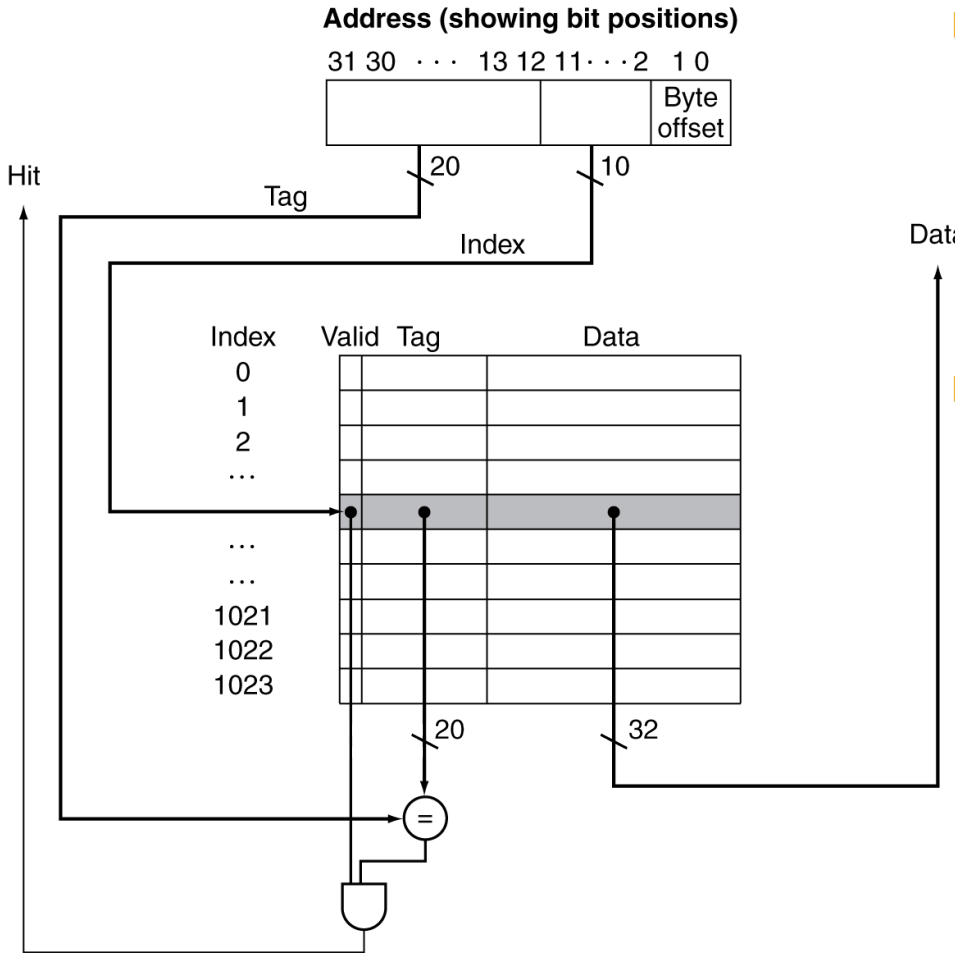
- Principle of locality
  - Programs work on a relatively small portion of data at any time
  - Can predict data accessed in near future by looking at recent accesses
- Temporal locality
  - If an item has been referenced recently, it will probably be accessed again soon
- Spatial locality
  - If an item has been accessed recently, nearby items will tend to be referenced soon
- Examples?

Memory Hierarchy



	Access time	Capacity	Managed by
Registers	1cycle	~500B	software/compiler
Level 1 Cache	1-3cycles	~64KB	hardware
Level 2/3 Cache	10-20cycles	1-100MB	hardware
DRAM	~100cycles	~100GB	software/OS
Disk	10^6-10^7cycles	~2TB	software/OS

Cache Organization & Access



- Assumptions
  - 32-bit address
  - 4 Kbyte cache
  - 1024 blocks, 32 bit/block
- Steps
  1. Use index to read V, tag from cache
  2. Compare read tag with tag from address
  3. If match, return data & hit signal
  4. Otherwise, return miss

Direct Mapped Problems: Conflict misses

- Two blocks are used concurrently and map to same index
  - Only one can fit in the cache, regardless of cache size
  - No flexibility in placing 2nd block elsewhere
- Thrashing
  - If accesses alternate, one block will replace the other before reuse
    - in our previous example 18, 26, 18, 26, … - every reference will miss
  - No benefit from caching
- Conflicts & thrashing can happen quite often

This is a real problem!

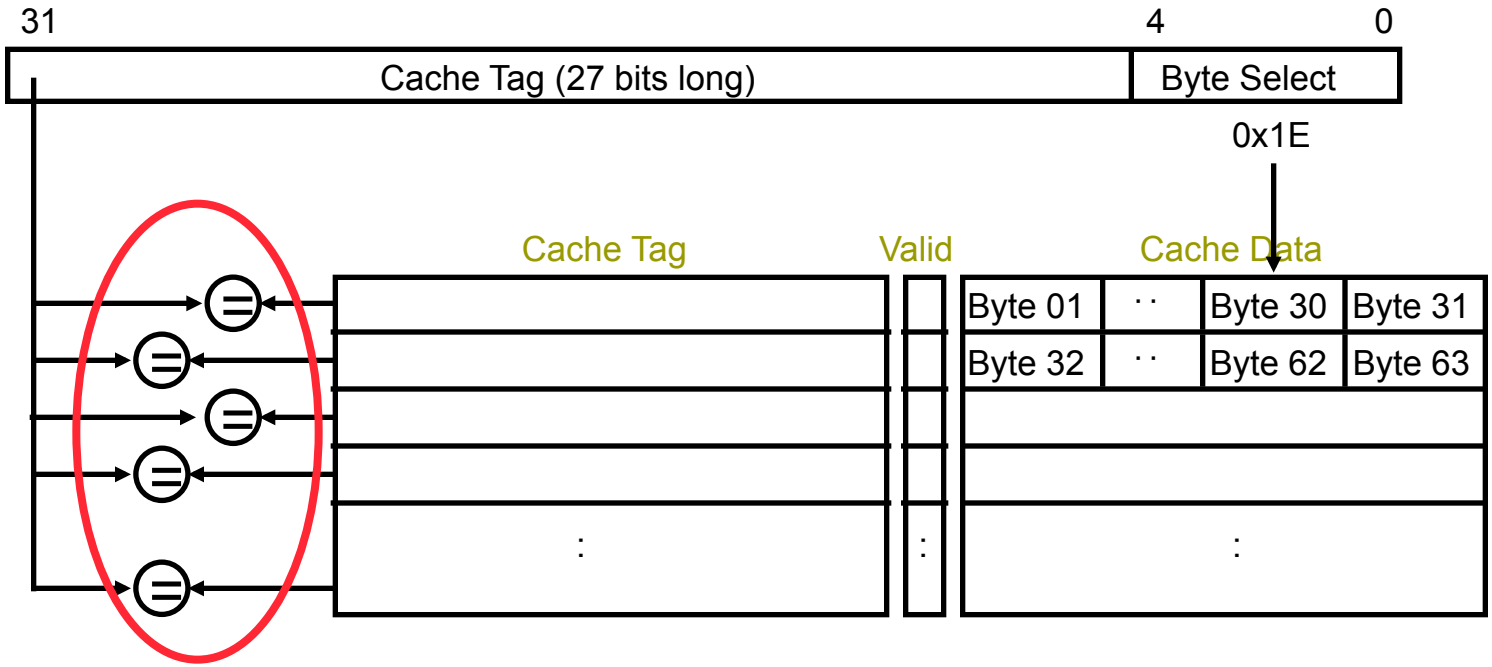
- Consider the following example code:

```
double a[8192], b[8192], c[8192];

void vector_sum()
{
    int i;

    for (i = 0; i < 8192; i++)
        c[i] = a[i] + b[i];
}
```

  - Arrays a, b, and c will tend to conflict in small caches
  - Code will get cache misses with *every* array access (3 per loop)
  - Spatial locality savings from blocks will be eliminated
- How can the severity of the conflicts be reduced?

Fully Associative Cache

- Opposite extreme in that it has no cache index to hash
  - Use any available entry to store memory elements
  - No conflict misses, only capacity misses
  - Must compare cache tags of *all* entries to find the desired one
    - Expensive or slow

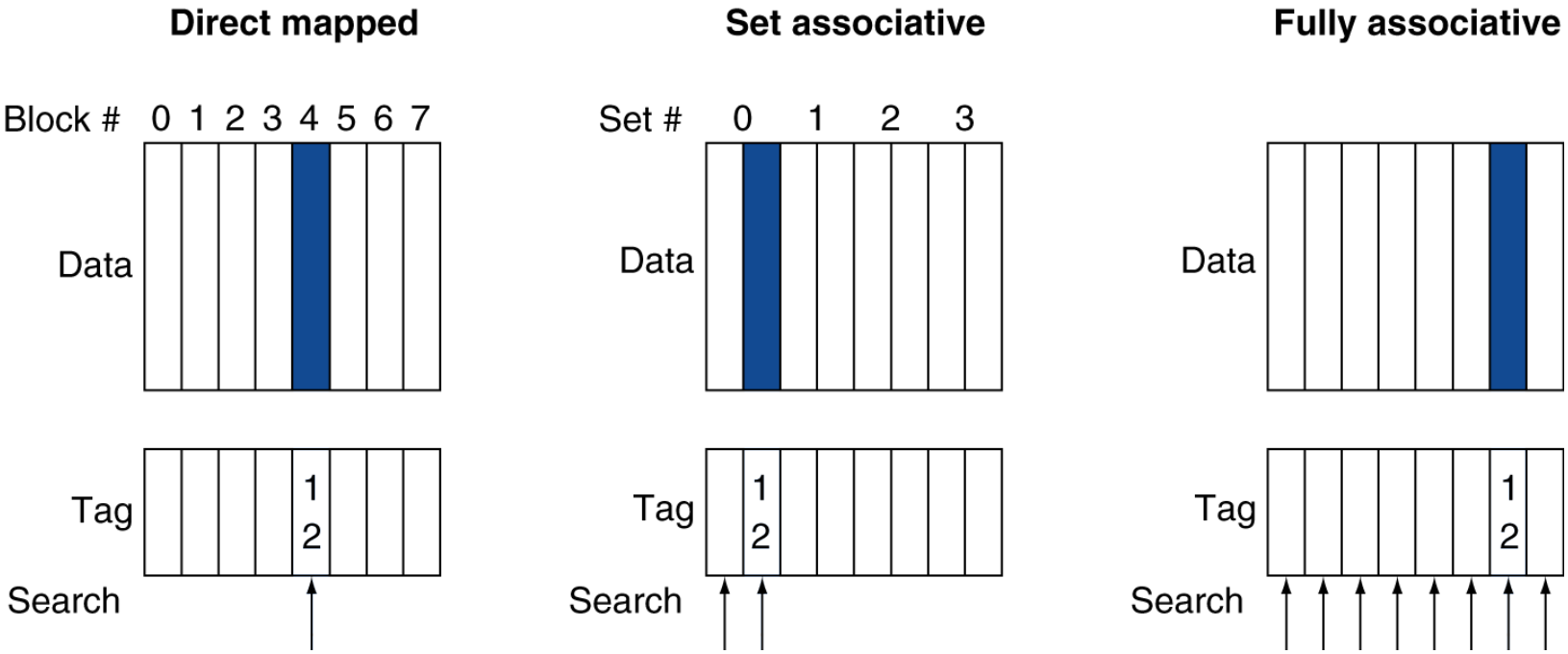


N-way Set Associative Cache

- Compromise between direct-mapped and fully associative
  - Each memory block can go to one of N entries in cache
    - Each "set" can store N blocks; a cache contains some number of sets
  - For fast access, all blocks in a set are search in parallel

- How to think of a N-way associative cache with X sets
  - 1st view: N direct mapped caches each with X entries
    - Caches search in parallel
    - Need to coordinate on data output and signaling hit/miss
  - 2nd view: X fully associative caches each with N entries
    - One cache searched in each case

Associative Cache Example



Associative Cache Example

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

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8

- Direct mapped
  - Index = (address % 4)

time

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity Example

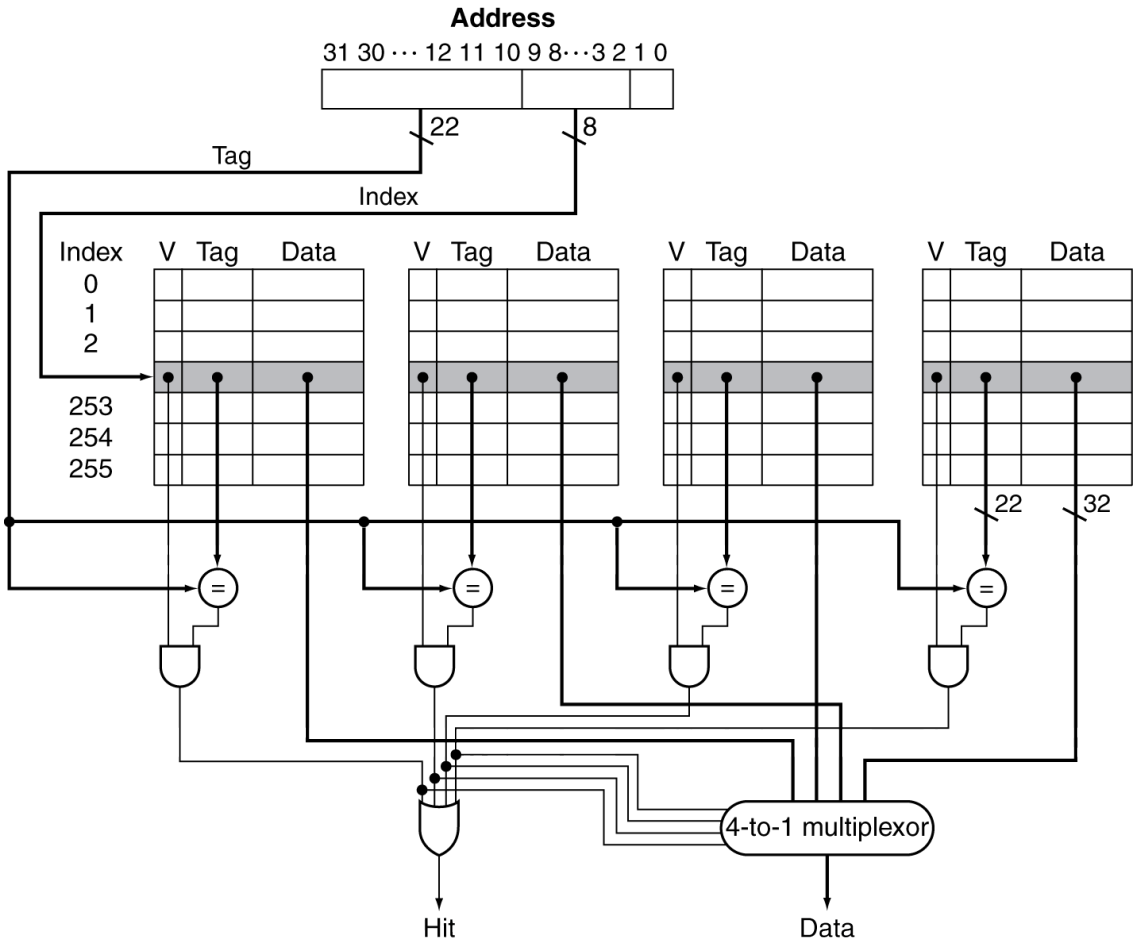
- 2-way set associative
  - Index = (address % 2)

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0 (even)		Set 1 (odd)	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- Fully associative
  - Blocks can go anywhere

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

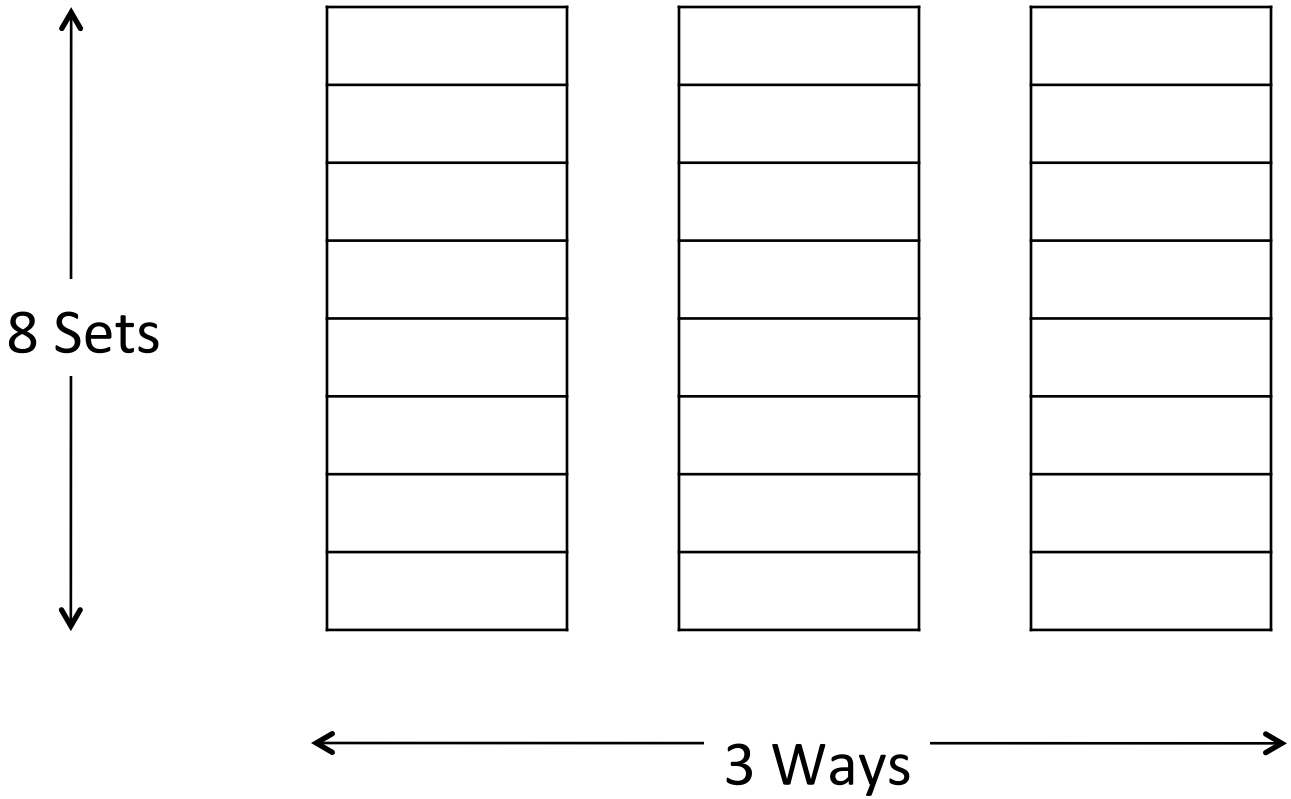
Set Associative Cache Design



Bonus Trick: How to build a 3KB cache?

- It would be difficult to make a DM 3KB cache
  - 3KB is not a power of two
  - Assuming 16-byte blocks, we have 24 blocks to select from
  - (address % 24) is very expensive to calculate
    - Unlike (address % 16) which requires looking at the 4 LS bits
- Solution: start with 4KB 4-way set associative cache
  - Every way can hold 1-KB (8 blocks)
  - Same 3-bit index used to access all 4 cache ways
    - 3 LS bits of address (after eliminating the block offset)
  - Now drop the 4th way of the cache
    - As if that 4th way always reports a miss and never receives data

N ways x S sets



Associative Caches: Pros

- Increased associativity decreases miss rate
  - Eliminates conflicts
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%
- Caveat: cache shared by multiple cores may have need higher associativity

Review: Cache Organization Options 256 bytes, 16 byte block, 16 blocks

Organization	# of sets	# blocks / set	12 bit Address
Direct mapped			tag: 4 | index: 4 | blk off: 4
2-way set associative			tag: 5 | index: 3 | blk off: 4
4-way set associative			tag: 6 | ind: 2 | blk off: 4
8-way set associative			tag: 7 | i: 1 | blk off: 4
16-way (fully) set associative			tag: 8 | blk off: 4

Review: Cache Organization Options
256 bytes, 16 byte block, 16 blocks

Organization	# of sets	# blocks / set	12 bit Address
Direct mapped	16	1	tag: 4 | index: 4 | blk off: 4
2-way set associative			tag: 5 | index: 3 | blk off: 4
4-way set associative			tag: 6 | ind: 2 | blk off: 4
8-way set associative			tag: 7 | i: 1 | blk off: 4
16-way (fully) set associative			tag: 8 | blk off: 4

Review: Cache Organization Options 256 bytes, 16 byte block, 16 blocks

Organization	# of sets	# blocks / set	12 bit Address
Direct mapped	16	1	tag: 4 | index: 4 | blk off: 4
2-way set associative	8	2	tag: 5 | index: 3 | blk off: 4
4-way set associative			tag: 6 | ind: 2 | blk off: 4
8-way set associative			tag: 7 | i: 1 | blk off: 4
16-way (fully) set associative			tag: 8 | blk off: 4

Review: Cache Organization Options 256 bytes, 16 byte block, 16 blocks

Organization	# of sets	# blocks / set	12 bit Address
Direct mapped	16	1	tag: 4 | index: 4 | blk off: 4
2-way set associative	8	2	tag: 5 | index: 3 | blk off: 4
4-way set associative	4	4	tag: 6 | ind: 2 | blk off: 4
8-way set associative			tag: 7 | i: 1 | blk off: 4
16-way (fully) set associative			tag: 8 | blk off: 4

Review: Cache Organization Options 256 bytes, 16 byte block, 16 blocks

Organization	# of sets	# blocks / set	12 bit Address
Direct mapped	16	1	tag: 4 | index: 4 | blk off: 4
2-way set associative	8	2	tag: 5 | index: 3 | blk off: 4
4-way set associative	4	4	tag: 6 | ind: 2 | blk off: 4
8-way set associative	2	8	tag: 7 | i: 1 | blk off: 4
16-way (fully) set associative			tag: 8 | blk off: 4

Review: Cache Organization Options 256 bytes, 16 byte block, 16 blocks

Organization	# of sets	# blocks / set	12 bit Address
Direct mapped	16	1	tag: 4 | index: 4 | blk off: 4
2-way set associative	8	2	tag: 5 | index: 3 | blk off: 4
4-way set associative	4	4	tag: 6 | ind: 2 | blk off: 4
8-way set associative	2	8	tag: 7 | i: 1 | blk off: 4
16-way (fully) set associative	1	16	tag: 8 | blk off: 4

Associative Caches: Cons

- Area overhead
  - More storage needed for tags (compared to same sized DM)
  - N comparators
- Latency
  - Critical path = way access + comparator + logic to combine answers
    - Logic to OR hit signals and multiplex the data ouptuts
  - Cannot forward the data to processor immediately
    - Must first wait for selection and multiplexing
    - Direct mapped assumes a hit and recovers later if a miss
- Complexity: dealing with replacement

Replacement Methods

- Which line do you replace on a miss?
- Direct mapped
  - Easy, you have only one choice
  - Replace the line at the index you need
- N-way set associative
  - Need to choose which way to replace
  - Random (choose one at random)
  - Least Recently Used (LRU) (the one used least recently)
    - Keep encoded permutation – for N-ways, N! orderings.
    - For 4-way cache: How many orderings? How many bits to encode?

What About Writes?

- Where do we put the data we want to write?
  - In the cache?
  - In main memory?
  - In both?
- Caches have different policies for this question
  - Most systems store the data in the cache (why?)
  - Some also store the data in memory as well (why?)
- Interesting observation
  - Processor does not need to “wait” until the store completes

Cache Write Policies: Major Options

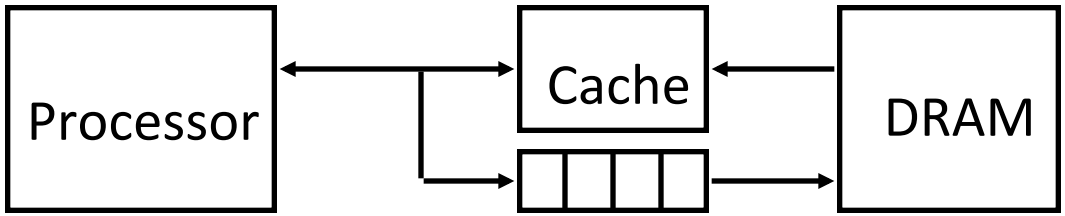
- Write-through (write data go to cache and memory)
  - Main memory is updated on each cache write
  - Replacing a cache entry is simple (just overwrite new block)
  - Memory write causes significant delay if pipeline must stall

- Write-back (write data only goes to the cache)
  - Only the cache entry is updated on each cache write so main memory and the cache data are inconsistent
  - Add “dirty” bit to the cache entry to indicate whether the data in the cache entry must be committed to memory
  - Replacing a cache entry requires writing the data back to memory before replacing the entry if it is “dirty”

Write Policy Trade-offs

- Write-through
  - Misses are simpler and cheaper (no write-back to memory)
  - Easier to implement
    - But requires buffering to be practical (see following slide)
  - Uses a lot of bandwidth to the next level of memory
    - Every write goes to next level
    - Not power efficient!
- Write-back
  - Writes are fast on a hit (no write to memory)
  - Multiple writes within a block require only one “writeback” later

Avoiding the Stalls for Write-Through



- Use Write Buffer between cache and memory
  - Processor writes data into the cache and the write buffer
  - Memory controller slowly “drains” buffer to memory

- Write Buffer: a first-in-first-out buffer (FIFO)
  - Typically holds a small number of writes
  - Can absorb small bursts as long as the long-term rate of writing to the buffer does not exceed the maximum rate of writing to DRAM

Write Buffers Quiz:

- If a write-through cache has a write buffer, what should happen on a read miss?

- Are write-buffers of any use for write-back caches?

Be Careful, Even with Write Hits

- **Reading** from a cache
  - Read tags and data in parallel
  - If it hits, return the data, else go to next level
- **Writing** a cache can take more time
  - First read tag to determine hit/miss (access 1)
  - Then overwrite data on a hit (access 2)
    - Otherwise, you may overwrite dirty data or write the wrong cache way

Cache Write Policy: Write Miss Options

- What happens on a cache write that misses?
  - It's actually two sub-questions
- Do you allocate space in the cache for the address?
  - Write-allocate VS no-write allocate
  - Actions: select a cache entry, evict old contents, update tags, …
- Do you fetch the rest of the block contents from memory?
  - Of interest if you do write allocate
  - Remember a store updates up to 1 word from a wider block
  - Fetch-on-miss Vs no-fetch-on-miss
    - For no-fetch-on-miss must remember which words are valid
    - Use fine-grain valid bits in each cache line

Write Miss Actions

Only works for **DM cache**

	Write through				Write back	
	Write allocate		No write allocate		Write allocate	
Steps	fetch on miss	no fetch on miss	write around	write invalidate	fetch on miss	no fetch on miss
1	pick replacement	pick replacement			pick re-placement	pick re-placement
2				invalidate tag	[write back]	[write back]
3	fetch block				fetch block	
4	write cache	write partial cache			write cache	write partial cache
5	write memory	write memory	write memory	write memory		

Typical Choices

- Write-back caches
  - Write-allocate, fetch-on-miss (why?)
- Write-through caches
  - Write-allocate, fetch-on-miss
  - Write-allocate, no-fetch-on-miss
  - No-write-allocate, write-around
- Which program patterns match each policy?
- Modern HW support multiple policies
  - Selected by OS on at some coarse granularity (e.g. 4KB)

Quiz

- How can we patch our pipeline to have a single unified DRAM for data and instructions but two ports for accessing instructions & data?

Splitting Caches

- Most chips have separate caches for instructions & data
  - Often noted ad $I and $D or I-cache and D-cache
- Advantages
  - Extra access port, bandwidth
  - Low hit time
  - Customize to specific patterns (e.g. line size)
- Disadvantages
  - Capacity utilization
  - Miss rate

Multilevel Caches

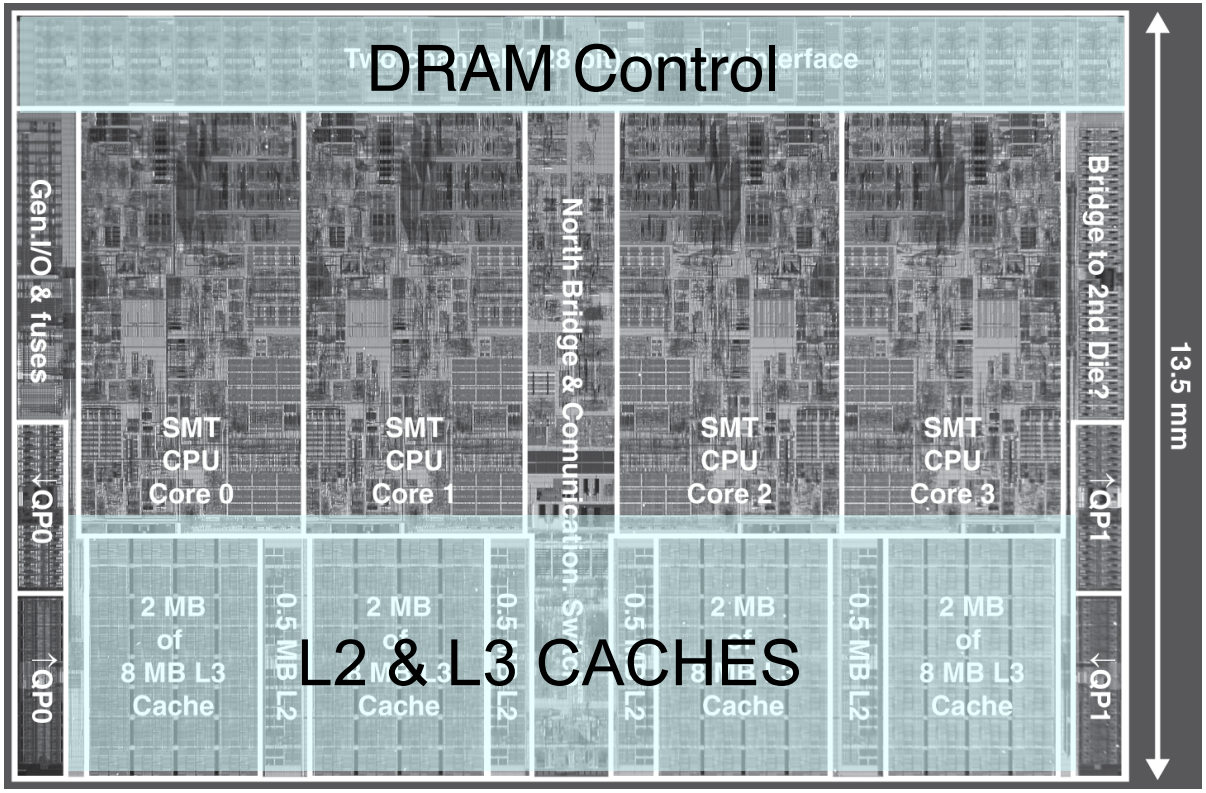
- Primary (L1) caches attached to CPU
  - Small, but fast
  - Focusing on hit time rather than miss rate
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
  - Unified instruction and data (why?)
  - Focusing on low miss rate rather than low hit time (why?)
- Main memory services L2 cache misses
  - Many chips include L3 cache

Multilevel On-Chip Caches

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

Multilevel On-Chip Caches

Intel Nehalem 4-core processor

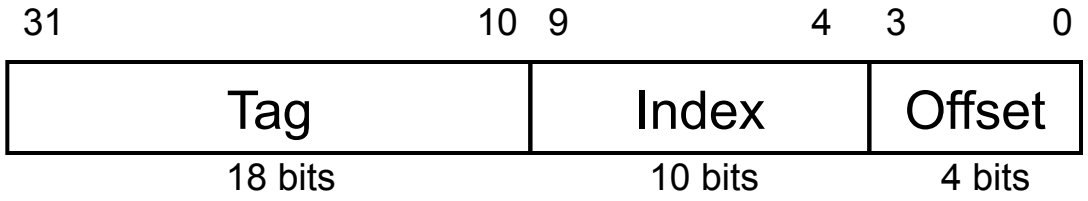


Per core:
-32KB, 4-way L1 $I
-32KB, 8-way L1 $D
-256KB, 8-way L2

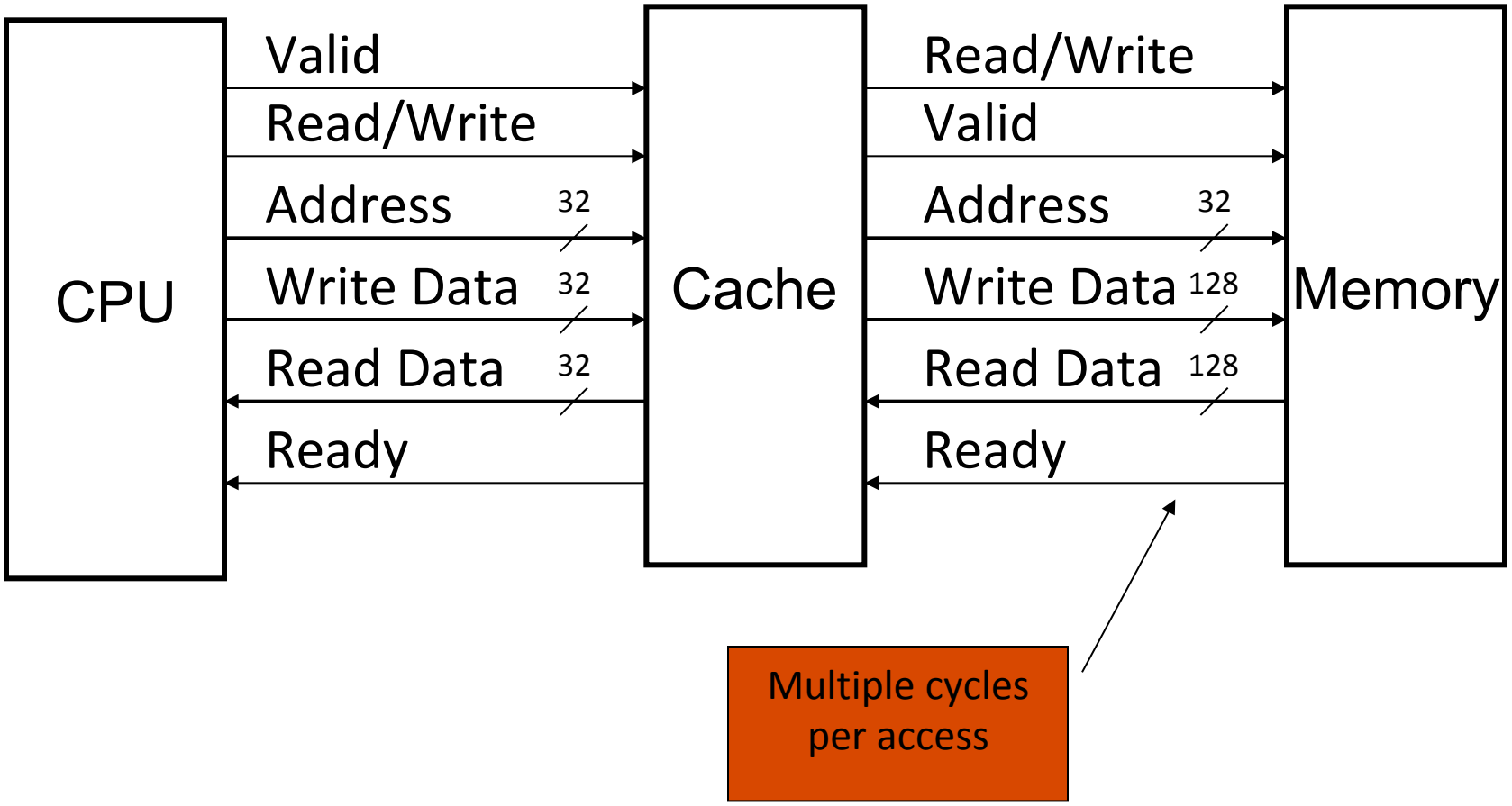
Shared
- 8 MB, 16-way L3

Cache Implementation: Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

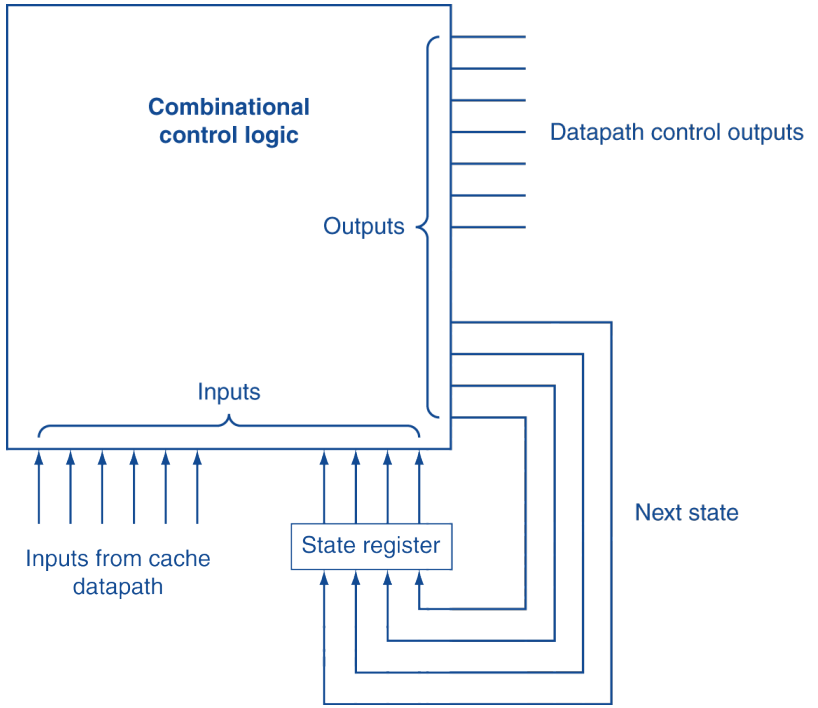


Interface Signals



Reminder: Finite State Machines

- Use an FSM to sequence control steps
- Set of states, transition on each clock edge
  - State values are binary encoded
  - Current state stored in a register
  - Next state
    = fn (current state, current inputs)



- Control output signals = fn (current state)

Cache Controller FSM - WRITE