CMPE110 Lecture 18
Caches III

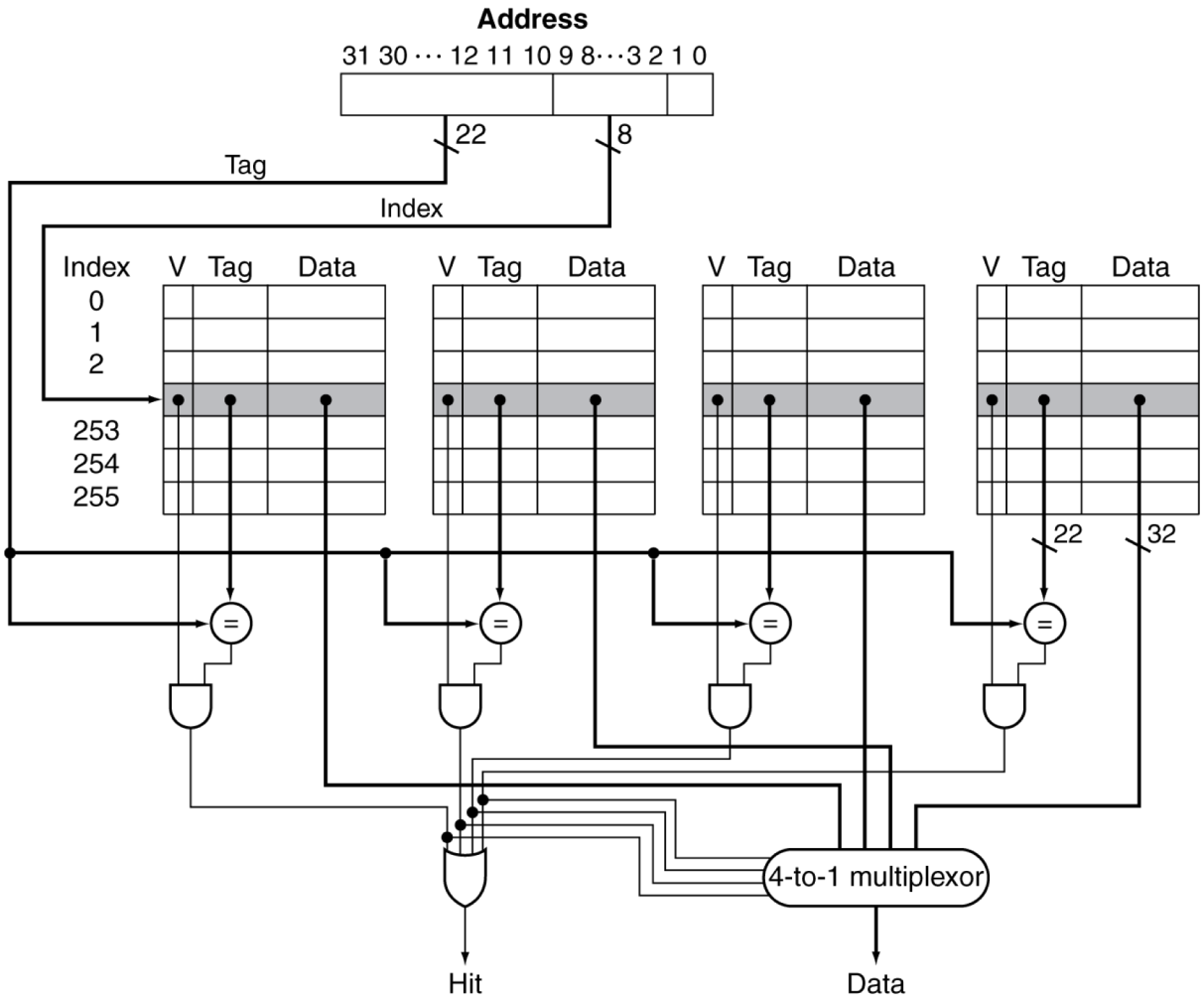
Heiner Litz

https://canvas.ucsc.edu/courses/19290

Announcements

Review

Set Associative Cache Design



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

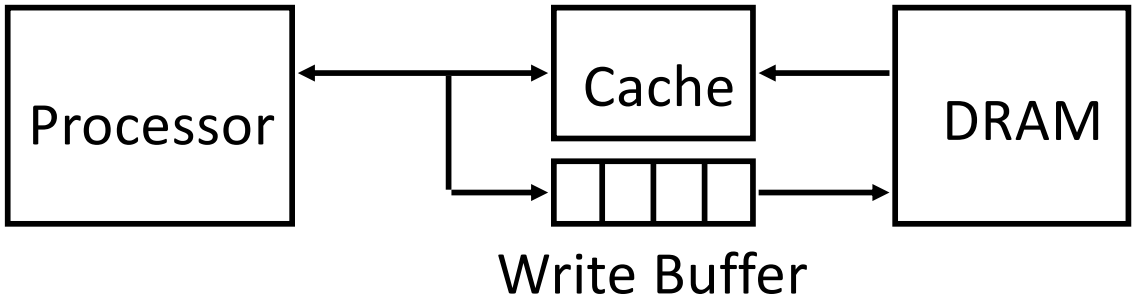
- Write-through (write data go to cache and memory)
  - Main memory is updated on each cache write
  - Replacing a cache entry is simple (just overwrite new block)
  - Memory write causes significant delay if pipeline must stall

- 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

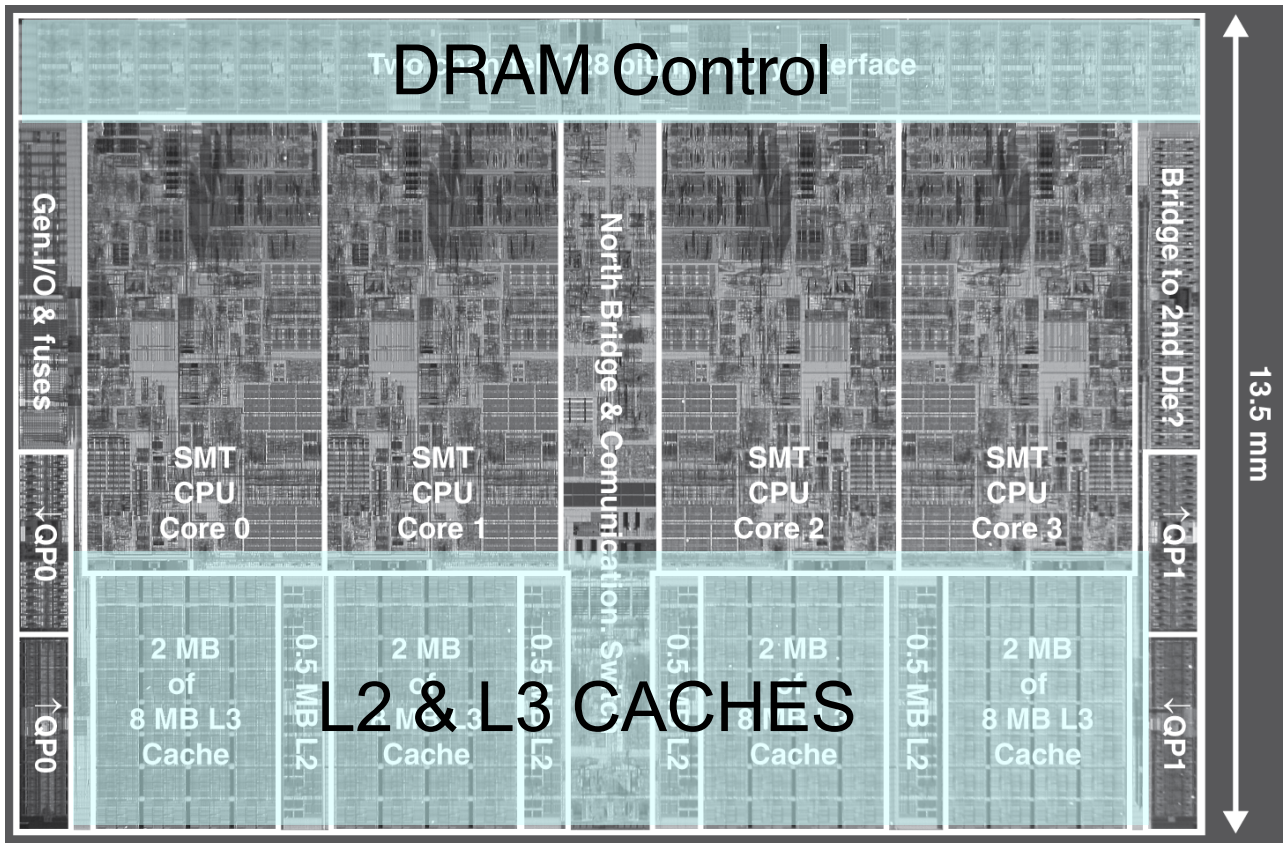
- Primary (L1) caches attached to CPU
  - Small, but fast
  - Focusing on hit time rather than miss rate
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
  - Unified instruction and data (why?)
  - Focusing on low miss rate rather than low hit time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

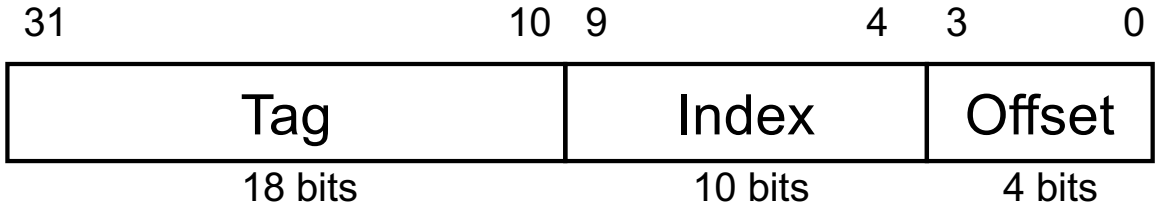


Per core:
-32KB, 4-way L1 $I
-32KB, 8-way L1 $D
-256KB, 8-way L2

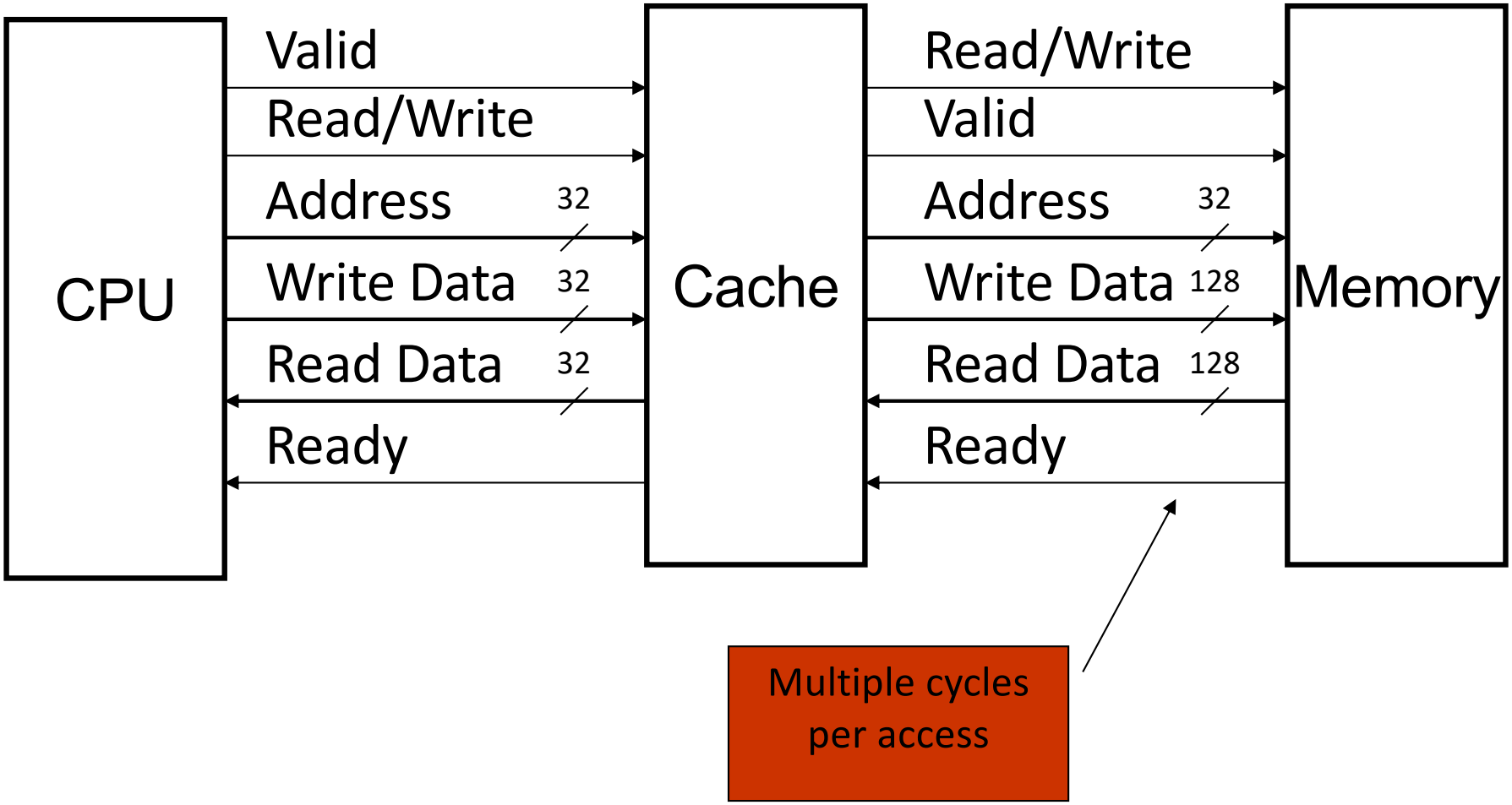
Shared
- 8 MB, 16-way L3

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

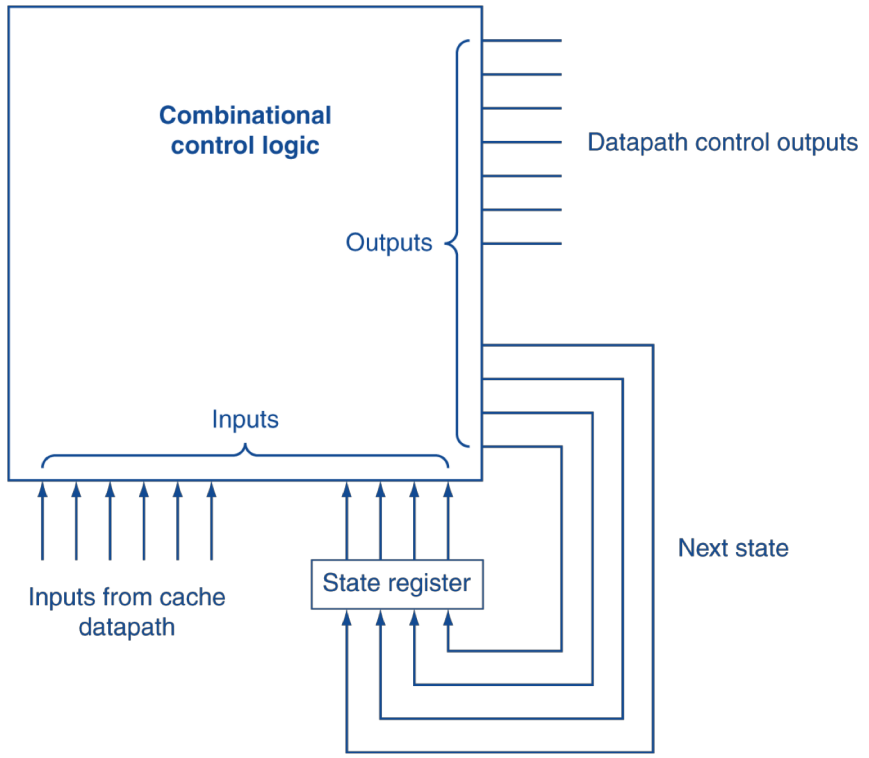


Interface Signals



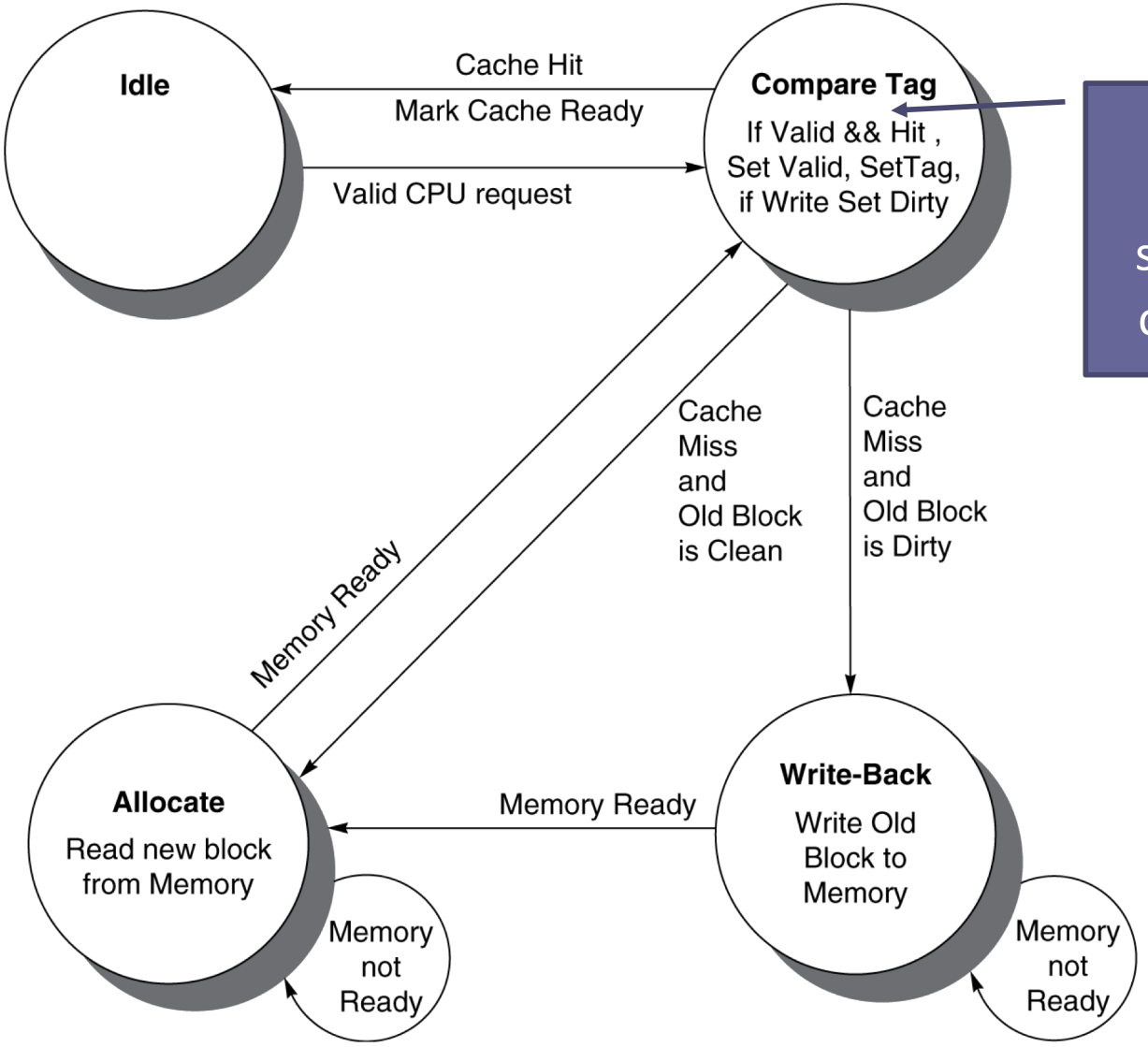
Reminder: Finite State Machines

- 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



Could partition into separate states to reduce clock cycle time