Computer Architecture

Juho Kim

Department of Computer Science and Engineering

Sogang University

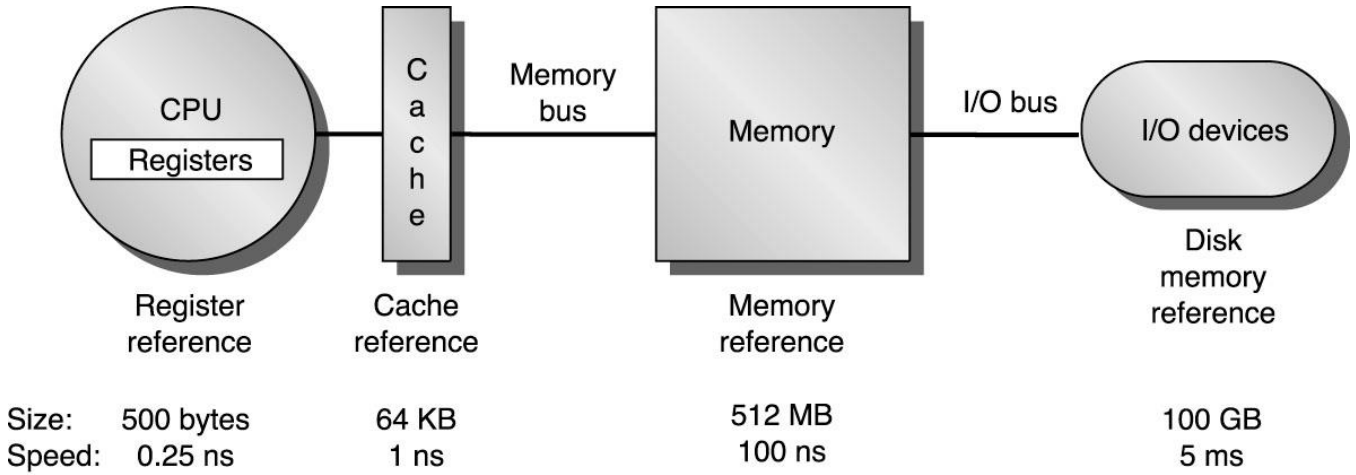
Chapter 5 Memory Hierarchy

Contents of chapter 5

Memory Technology

- **Static RAM (SRAM)**
  - 0.5ns – 2.5ns, $2000 – $5000 per GB
- **Dynamic RAM (DRAM)**
  - 50ns – 70ns, $20 – $75 per GB
- **Magnetic disk**
  - 5ms – 20ms, $0.20 – $2 per GB
- **Ideal memory**
  - Access time of SRAM
  - Capacity and cost/GB of disk

Memory hierarchy design



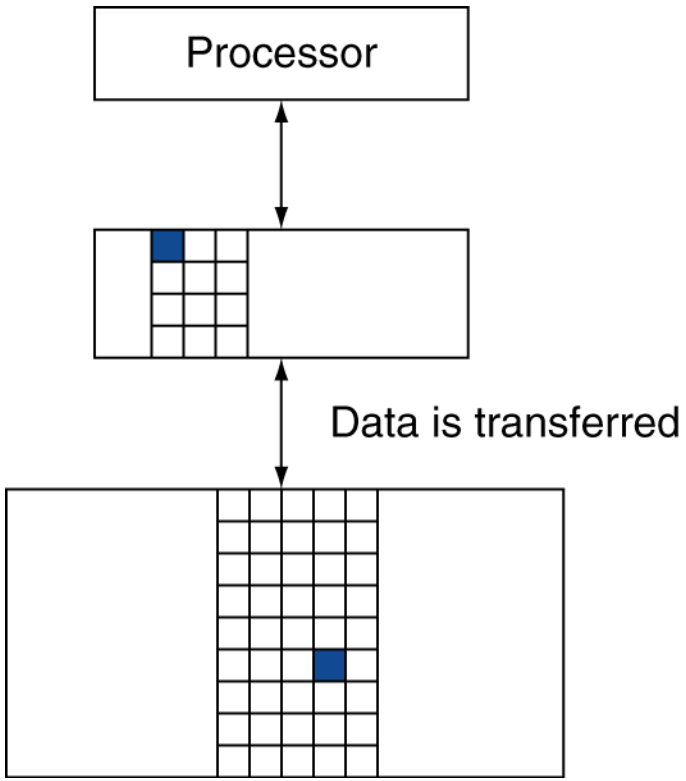
Principle of Locality

- **Programs access a small proportion of their address space at any time**
- **Temporal locality**
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- **Spatial locality**
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

Taking Advantage of Locality

- **Memory hierarchy**
  - Store everything on disk
  - Copy recently accessed (and nearby) items from disk to smaller DRAM memory
    - Main memory
  - Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
    - Cache memory attached to CPU

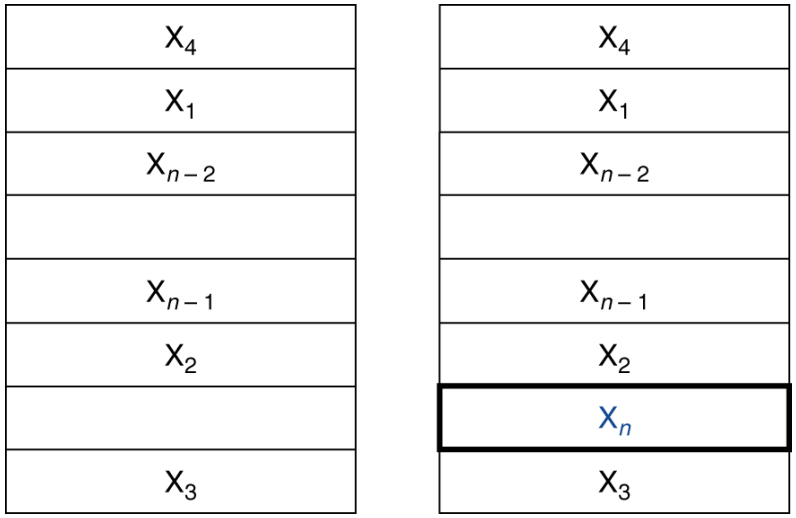
Memory Hierarchy Levels



- **Block (aka line): unit of copying**
  - May be multiple words
- **If accessed data is present in upper level**
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- **If accessed data is absent**
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses
      = 1 – hit ratio
  - Then accessed data supplied from upper level

Cache Memory

- **Cache memory**
  - The level of the memory hierarchy closest to the CPU
- **Given accesses X_1, …, $X_{n–1}$, X_n**

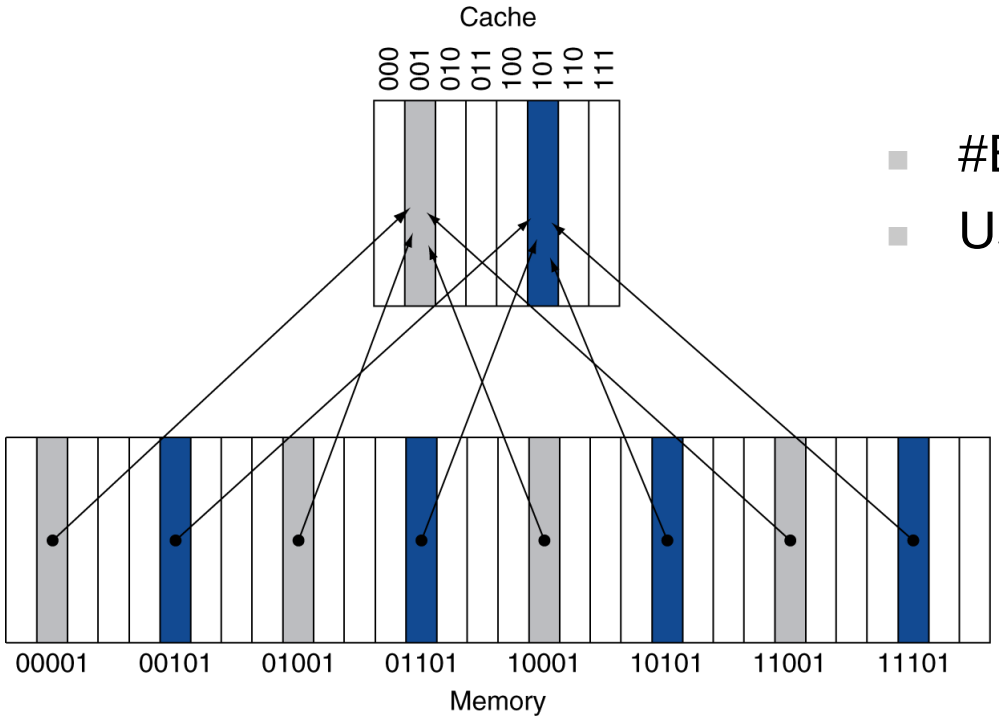


a. Before the reference to X_n b. After the reference to X_n

- How do we know if the data is present?
- Where do we look?

Direct Mapped Cache

- **Location determined by address**
- **Direct mapped: only one choice**
  - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits

Tags and Valid Bits

- **How do we know which particular block is stored in a cache location?**
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag
- **What if there is no data in a location?**
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

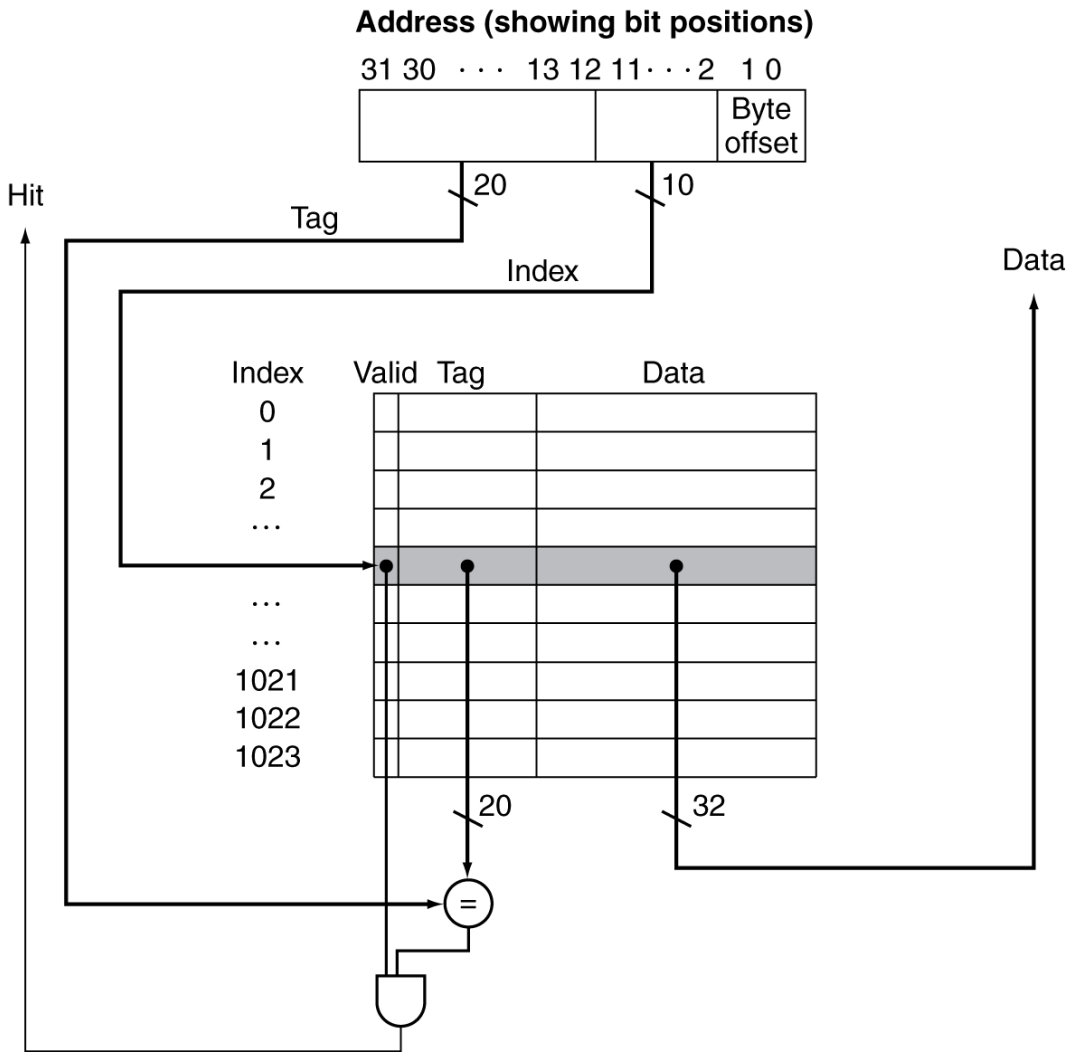
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

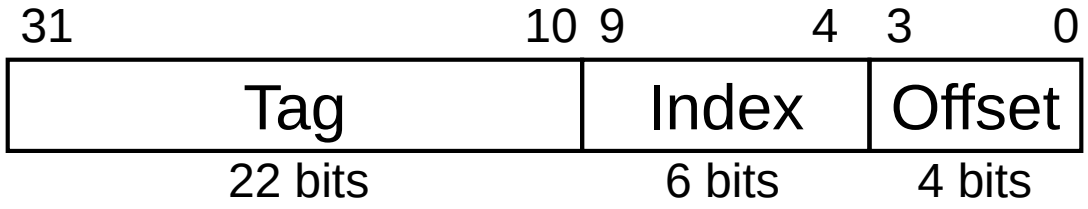
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Address Subdivision



Example: Larger Block Size

- **64 blocks, 16 bytes/block**
  - To what block number does address 1200 map?
- **Block address = ⌊1200/16⌋ = 75**
- **Block number = 75 modulo 64 = 11**



Block Size Considerations

- **Larger blocks should reduce miss rate**
  - Due to spatial locality
- **But in a fixed-sized cache**
  - Larger blocks ⇒ fewer of them
    - More competition ⇒ increased miss rate
  - Larger blocks ⇒ pollution
- **Larger miss penalty**
  - Can override benefit of reduced miss rate
  - Early restart and critical-word-first can help

Cache Misses

- **On cache hit, CPU proceeds normally**
- **On cache miss**
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Write strategy

- What happens on a write (write strategy)?
  - Reads dominate processor cache accesses (instruction accesses)
  - Block read from cache means reading and comparing tag (if read is hit, the requested part of the block is passed on to the CPU immediately)
  - Modifying a block (write)
    - Checking tag for hit (takes longer than read)
    - Size of write is specified from processor => only portion for update

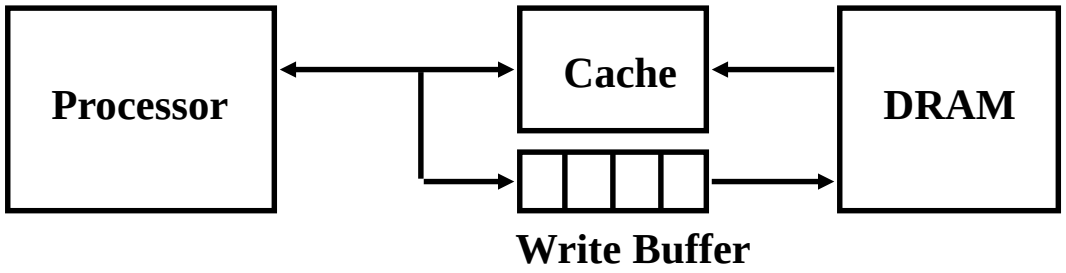
- Write policies
  - Write through: The information is written to both the block in the cache and to the block in the lower-level memory
  - Write back: The information is written only to the block in the cache. The modified cache block is written to main memory only when if is replaced.

The dirty bit is used to reduce replacement (modified block is set to dirty)

Write strategy – write miss

- Write stall: when the CPU must wait for writes to complete during write through => write buffer can reduce write stalls

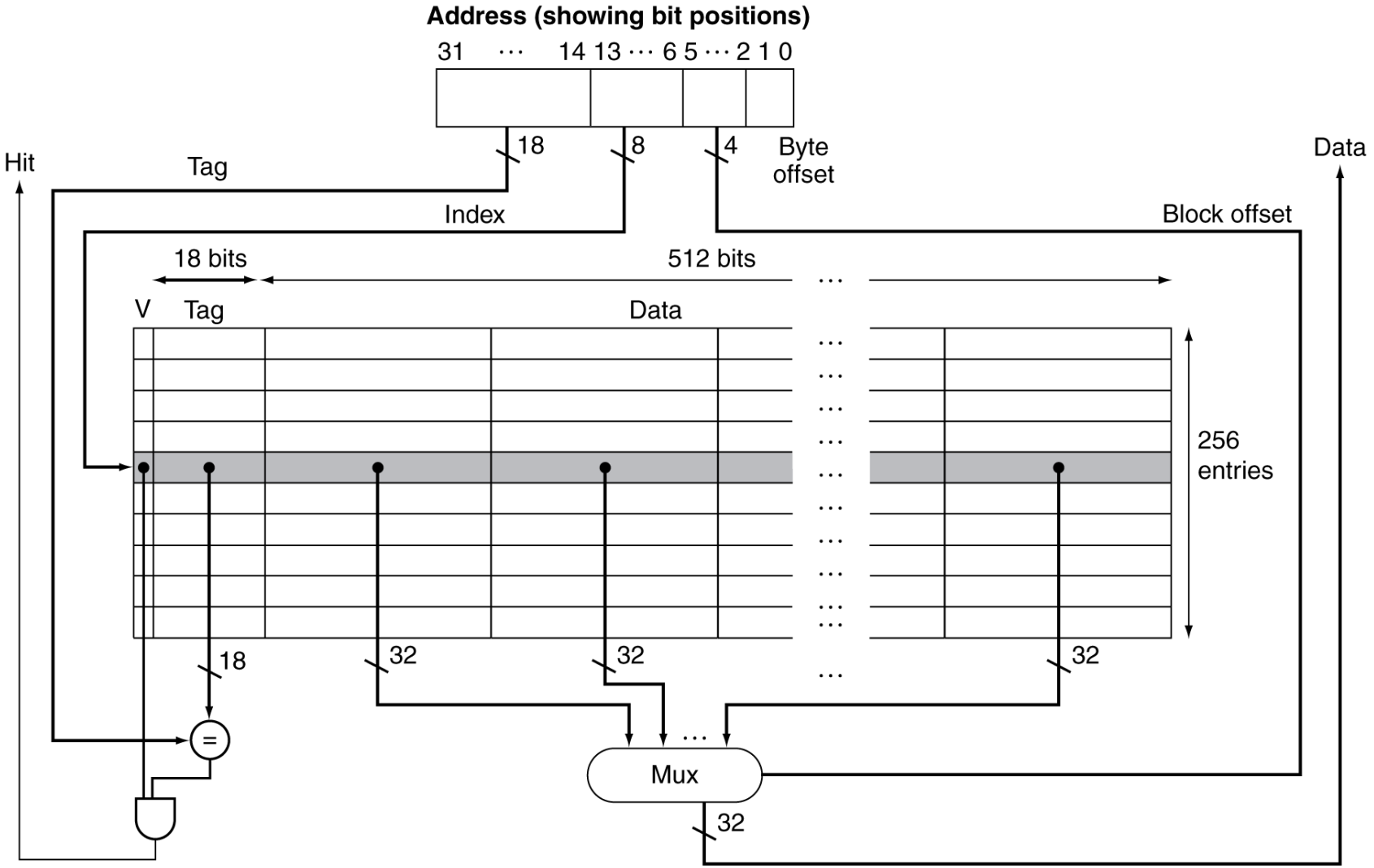
- Two options on a write miss:
  - Write allocate: block is allocated on a write miss, followed by write hit action
  - No write allocate: do not affect cache, instead modify lower-level memory



Example: Intrinsity FastMATH

- **Embedded MIPS processor**
  - 12-stage pipeline
  - Instruction and data access on each cycle
- **Split cache: separate I-cache and D-cache**
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- **SPEC2000 miss rates**
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

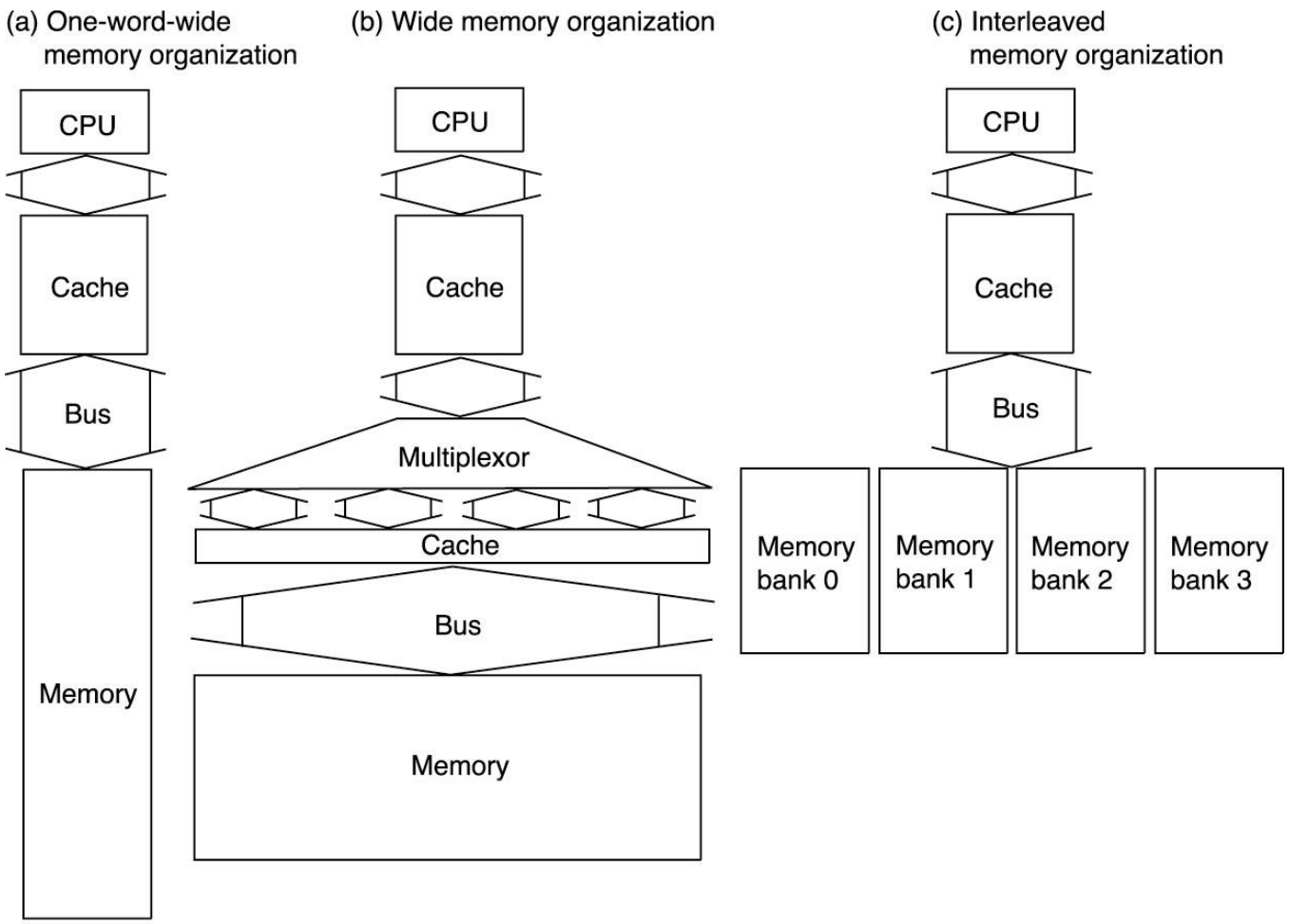
Example: Intrinsity FastMATH



Main Memory Supporting Caches

- **Use DRAMs for main memory**
  - Fixed width (e.g., 1 word)
  - Connected by fixed-width clocked bus
    - Bus clock is typically slower than CPU clock
- **Example cache block read**
  - 1 bus cycle for address transfer
  - 15 bus cycles per DRAM access
  - 1 bus cycle per data transfer
- **For 4-word block, 1-word-wide DRAM**
  - Miss penalty = 1 + 4×15 + 4×1 = 65 bus cycles
  - Bandwidth = 16 bytes / 65 cycles = 0.25 B/cycle

Increasing Memory Bandwidth



Performance comparison at cache miss

- Basic memory has
  - 4 clock cycles to send address
  - 56 clock cycles for the access time per word
  - 4 clock cycles to send a word of data
- Cache block of 4 words (a word is 8 bytes)

$\Rightarrow$ Miss penalty = 4 x (4+56+4) = 256 clock cycles

$\Rightarrow$ Bandwidth = one word/cycles = 32/256 = 1/8 byte/clock

- Wide memory: Doubling or quadrupling the width of the memory will double or quadruple the memory bandwidth. Also, miss penalty drops as bandwidth increases
- Wide memory drawbacks
  1) Wider memory bus cost
  2) Multiplexer becomes critical timing path
  3) Not expandable

Interleaved Memory

- **Simultaneous read and write to each bank**
- **Mapping of addresses to banks affects the behavior**
- **Example) block size = 1 word, memory bandwidth = 1 word, miss rate = 15%, memory access per instruction = 1.2, cache miss penalty = 8 cycles, CPI (without cache miss) = 2**
  - What is the improvement in performance of interleaving two ways and ways vs. doubling the width of memory and bus?

Block size	Miss rate
1 word	15%
2 words	10%
4 words	5%

Basic memory organization: 1 cycle for send address, 6 cycles for access time/word, 1 cycle to send a word of data

CPI for base machine using one word block
= 2 + (1.2 * 0.15 * 8) = 3.44

We can compare CPI

(1) Increasing block size to 2
32 bit bus and memory, no interleaving
= 2 + (1.2 * 0.1 * **2** * 8)
block size
32 bit bus and memory, interleaving
= 2+ (1.2 * 0.1 * (1+6+2))

(2) Increasing block size to 4
32 bit bus and memory, no interleaving
= 2 + (1.2 * 0.05 * 4 * 8)

32 bit bus and memory, interleaving
= 2 + (1.2 * 0.05 * (1+6+4))

DRAM vs. SRAM

- **DRAM**
  - Main memory
  - Refresh to maintain signals
  - 1 transistor for 1 bit
  - More power for refresh
  - Capacity 4 to 8 times of SRAM
  - Slower than SRAM

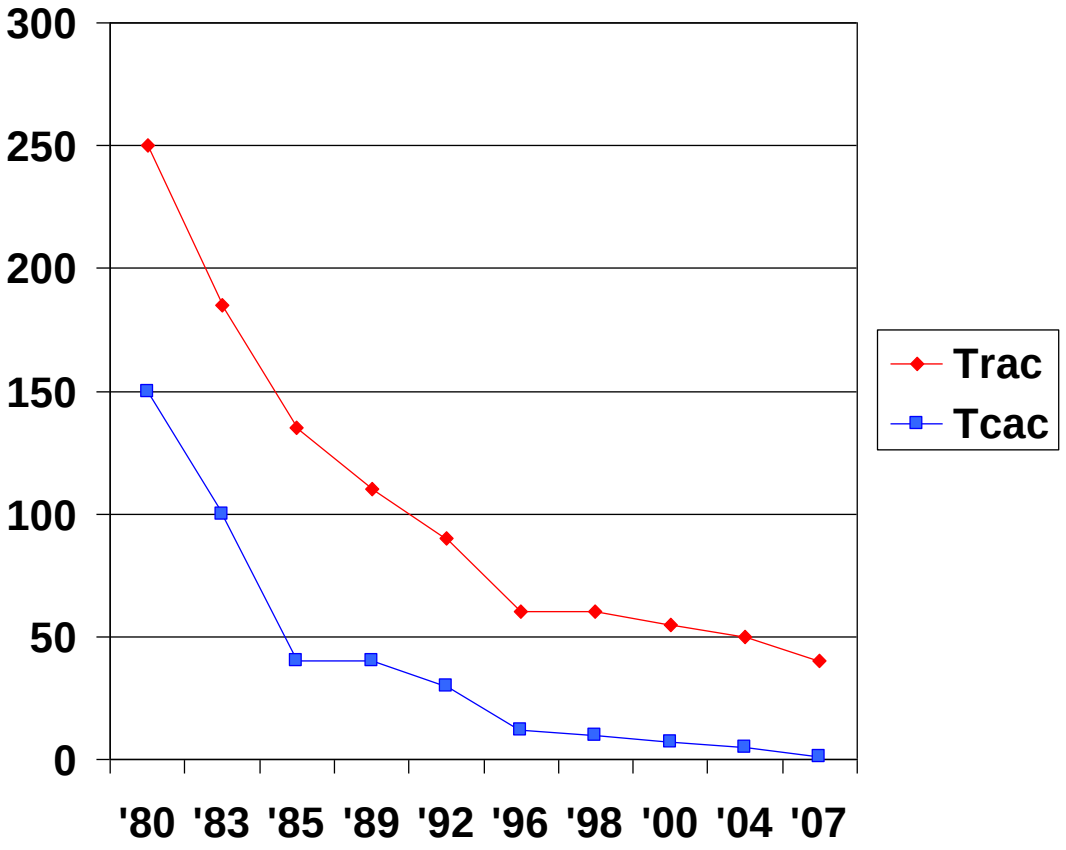
- **SRAM**
  - Cache
  - No need to refresh
  - 6 transistor for 1 bit
  - Less power consumption
  - ¼ to 1/8 times of DRAM
  - 8 to 16 times faster than DRAM

Advanced DRAM Organization

- **Bits in a DRAM are organized as a rectangular array**
  - DRAM accesses an entire row
  - Burst mode: supply successive words from a row with reduced latency
- **Double data rate (DDR) DRAM**
  - Transfer on rising and falling clock edges
- **Quad data rate (QDR) DRAM**
  - Separate DDR inputs and outputs

DRAM Generations

Year	Capacity	$/GB
1980	64Kbit	$1500000
1983	256Kbit	$500000
1985	1Mbit	$200000
1989	4Mbit	$50000
1992	16Mbit	$15000
1996	64Mbit	$10000
1998	128Mbit	$4000
2000	256Mbit	$1000
2004	512Mbit	$250
2007	1Gbit	$50



Measuring Cache Performance

CPU execution time = (CPU clock cycles + Memory stall cycles) x Clock cycle time

(assuming that CPU clock cycles include time to handle a cache hit and miss)

The memory stall cycles depends on both the number of misses and cost per miss, which is called the miss penalty:

$$MemoryStallCycles = NumberMisses \times MissPenalty$$

$$= IC \times \frac{Misses}{Instruction} \times MissPenalty$$

$$= IC \times \frac{MemoryAccess}{Instruction} \times MissRate \times MissPenalty$$

Memory stall cycles = IC x Reads per instruction x Read miss rate x Read miss penalty
+ IC x Writes per instruction x Write miss rate x Write miss penalty

Cache performance example

- **Given**
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- **Miss cycles per instruction**
  - I-cache: 0.02 × 100 = 2
  - D-cache: 0.36 × 0.04 × 100 = 1.44
- **Actual CPI = 2 + 2 + 1.44 = 5.44**
  - Ideal CPU is 5.44/2 =2.72 times faster

Cache Performance

- Average memory access time is better measure of memory hierarchy performance than just miss rate

Average memory access time = Hit time + Miss rate x Miss penalty

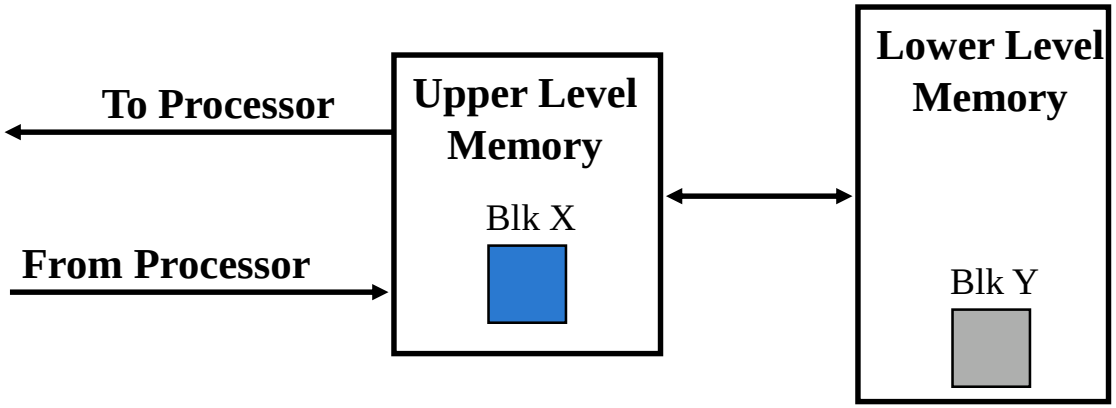
Can average memory access time due to cache miss predict processor performance?

No! Due to 1) other reasons for stalls (I/O devices)

2) out-of order CPU execution

CPU time = (CPU execution clock cycles + Memory stall clock cycles) x Clock cycle time

Where does cache hit time belong in the above equation?



Cache performance example

- In-order execution computer Ultra SPARC III
  - Cache miss penalty = 100 clock cycles
  - Instruction clock cycle = 1.0
  - Average miss rate = 2%
  - Average memory references per instruction = 1.5
  - Average number of cache miss per 1000 instruction = 30

- What is the impact of cache on performance?

  CPU time = IC x (CPI + Memory stall cycles/Instruction) x clock cycle time

  CPU time with cache = IC x (1.0 + (30/1000 x 100)) x clock cycle time

  = IC x 4.00 x clock cycle time

  Again, using miss rate:

  CPU time = IC x (CPI + miss rate x memory access/instruction x miss penalty) x clock cycle time

  = IC x (1.0 + (1.5 x 2% x 100)) x clock cycle time

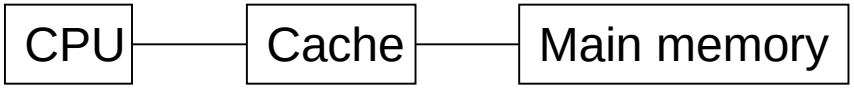
  = IC x 4.00 x clock cycle time

Improving cache performance

- Reducing miss penalty
  - Multilevel cache
  - Critical word first
  - Read miss before write miss
  - Merging write buffers
  - Victim caches
- Reducing miss rate
  - Larger block size, larger cache size, higher associativity, way prediction, psedoassociativity, compiler optimization
- Reducing miss penalty or miss rate via parallelism
  - Non-blocking caches, hardware prefetching, compiler prefetching
- Reducing time to hit in the cache
  - Small and simple caches, avoiding address translation, pipelined cache access, trace caches

Reducing cache miss penalty – multilevel caches

(1) Multilevel caches



- 2 ways to improve cache :
  - faster (to keep up with CPU's speed)
  - larger (to narrow the gap between CPU and Main Memory)

- Multilevel caches (CPU – L1 cache – L2 cache – Main Memory)
  - L1 cache: small and fast as CPU
  - L2 cache: large enough to reduce cache miss

Average memory access time = Hit time L1 + Miss rate L1 x Miss penalty L1

Miss penalty L1 = Hit time L2 + Miss rate L2 x Miss penalty L2

Average memory access time = Hit time L1 + Miss rate L1 x (
Hit time L2 + Miss rate L2 x Miss penalty L2)

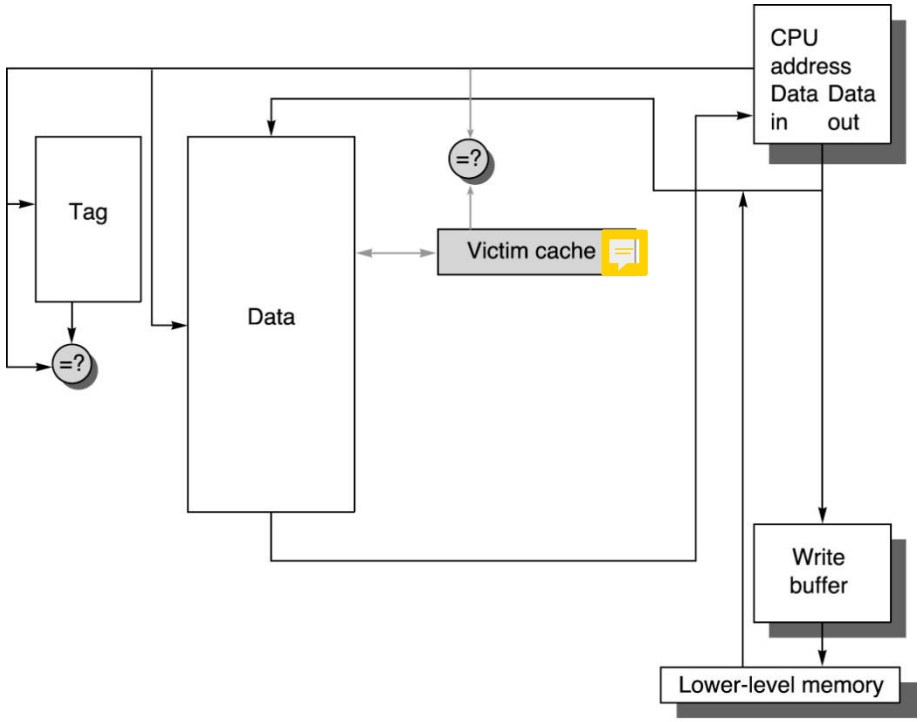
Reducing cache miss penalty – multilevel caches

- Performance analysis is complex for 2nd level cache
  - Local miss rate: Miss rate L1 and Miss rate L2 (simply number of misses divide by total memory access)
  - Global miss rate:
    - First level cache: Miss rate L1
    - Second level cache: Miss rate L1 x Miss rate L2

Average memory stalls per instruction = Misses per instruction L1 x Hit time L2
+ Misses per instruction L2 x Miss penalty L2

Reducing cache miss penalty – Victim cache

(2) Victim caches: To remember what was discarded in case it is needed again



Victim cache contains only blocks that are discarded from a cache because of a miss

If victim cache has desired data, victim block and cache block are swapped

Reducing Miss Rate

- 3 C's for miss categories
  - Compulsory: The very first access to a block cannot be in the cache, so the block must be brought into the cache (cold-start misses, first-reference misses)
  - Capacity: If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur
  - Conflict: If the block placement strategy is set associative or direct mapped, conflict misses will occur because a block may be discarded and later retrieved if too many blocks map to its set
- Miss Rate Reduction
  - Larger block size : reduce miss rate, increase miss penalty
  - Larger caches: reduce miss rate, longer hit time and higher cost
  - Higher associativity: reduce miss rate, increase hit time

Reducing Hit Time

- Small and simple caches
  - Cache hit uses index portion of the address to read the tag memory and then compare it to the address => faster simpler hardware (one chip)
- Avoiding address translation during indexing of the cache

  - Virtual cache address may remove address translation from physical cache => not in use due to protection, cache flushing, OS may use two different virtual address for one physical address

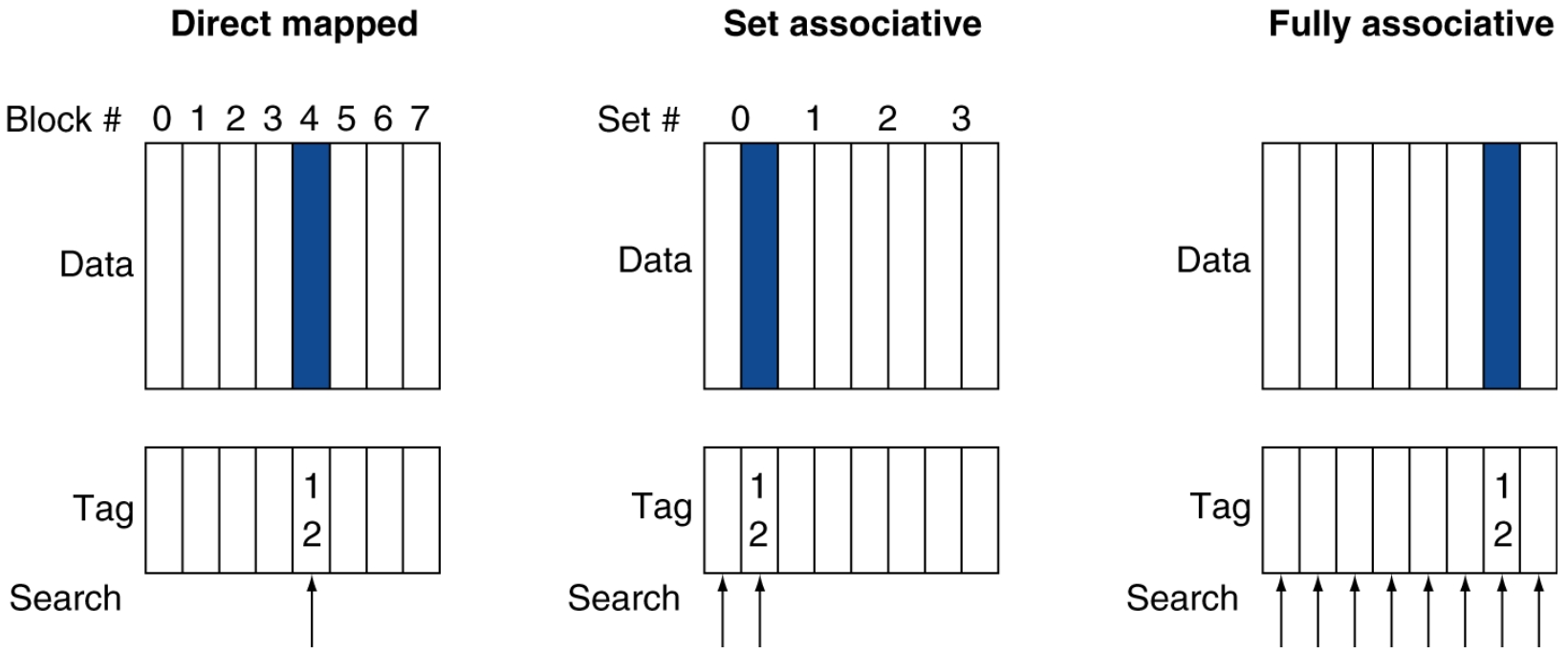
- Pipelined cache access => multiple clocks for hit (fast cycle time)

- Trace caches
  - Supply enough instructions every cycle without dependencies
  - Trace cache finds a dynamic sequence of instructions including taken branches to load into a cache block

Associative Caches

- **Fully associative**
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
- ***n*-way set associative**
  - Each set contains *n* entries
  - Block number determines which set
    - (Block number) modulo (#Sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

Associative Cache Example



Spectrum of Associativity

- **For a cache with 8 entries**

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**

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**

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
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]		

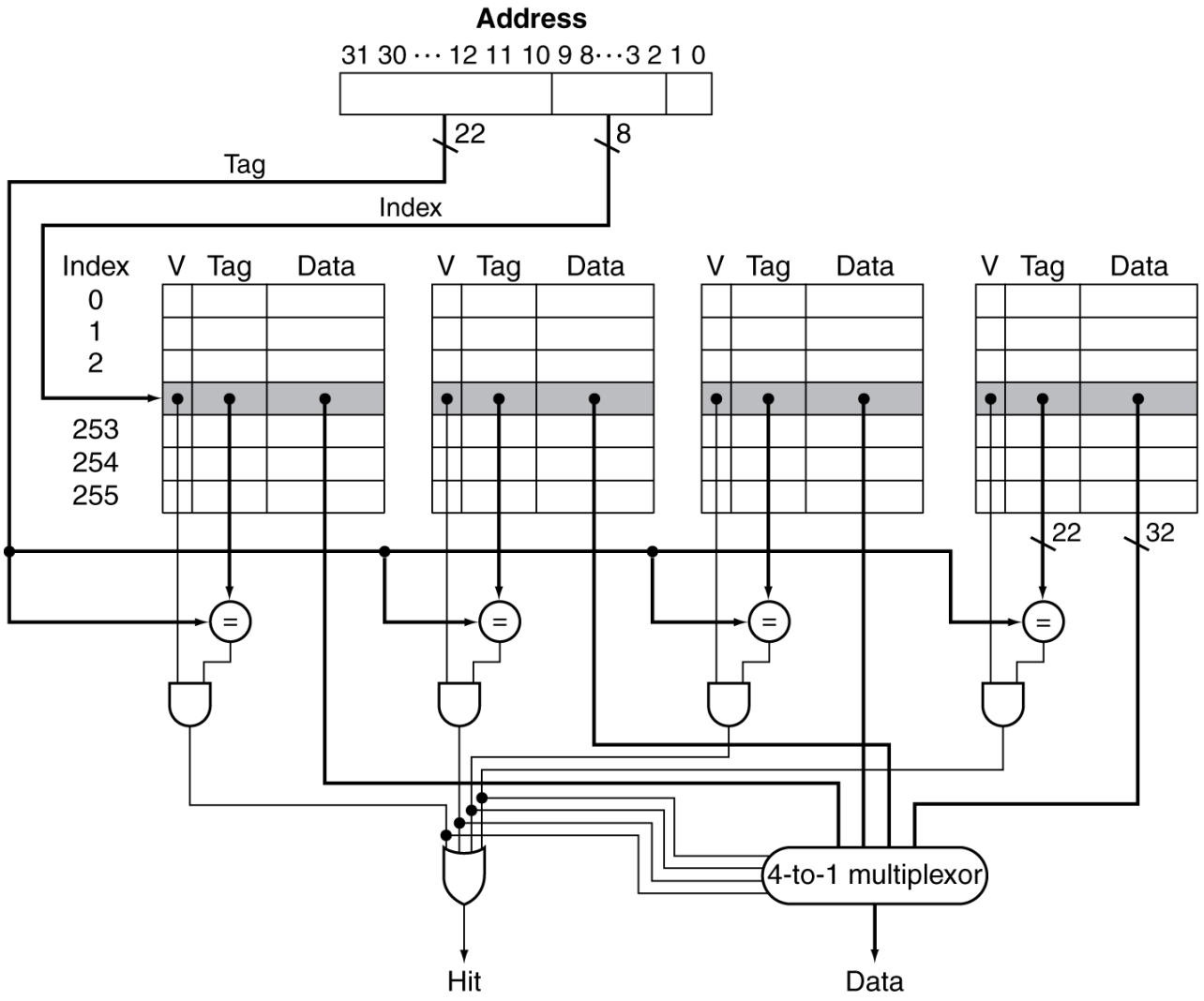
- **Fully associative**

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]	

How Much Associativity

- **Increased associativity decreases miss rate**
  - 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%

Set Associative Cache Organization



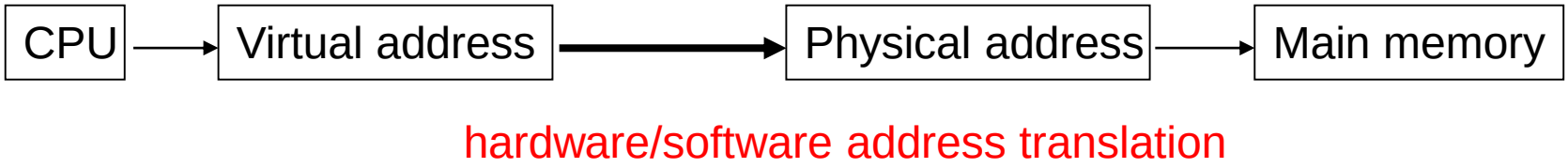
Replacement Policy

- **Direct mapped: no choice**
- **Set associative**
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- **Least-recently used (LRU)**
  - Choose the one unused for the longest time
    - Simple for 2-way, manageable for 4-way, too hard beyond that
- **Random**
  - Gives approximately the same performance as LRU for high associativity

Virtual Memory

- **Use main memory as a “cache” for secondary (disk) storage**
  - Managed jointly by CPU hardware and the operating system (OS)
- **Programs share main memory**
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- **CPU and OS translate virtual addresses to physical addresses**
  - VM “block” is called a page
  - VM translation “miss” is called a page fault

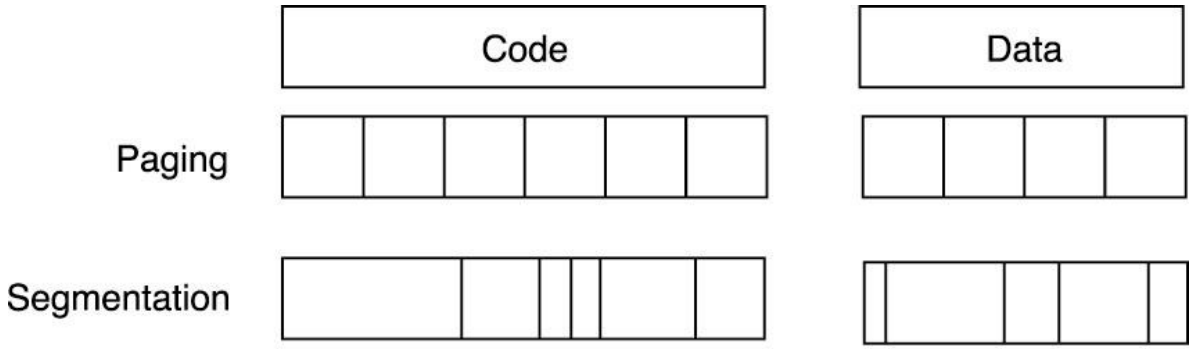
Virtual memory



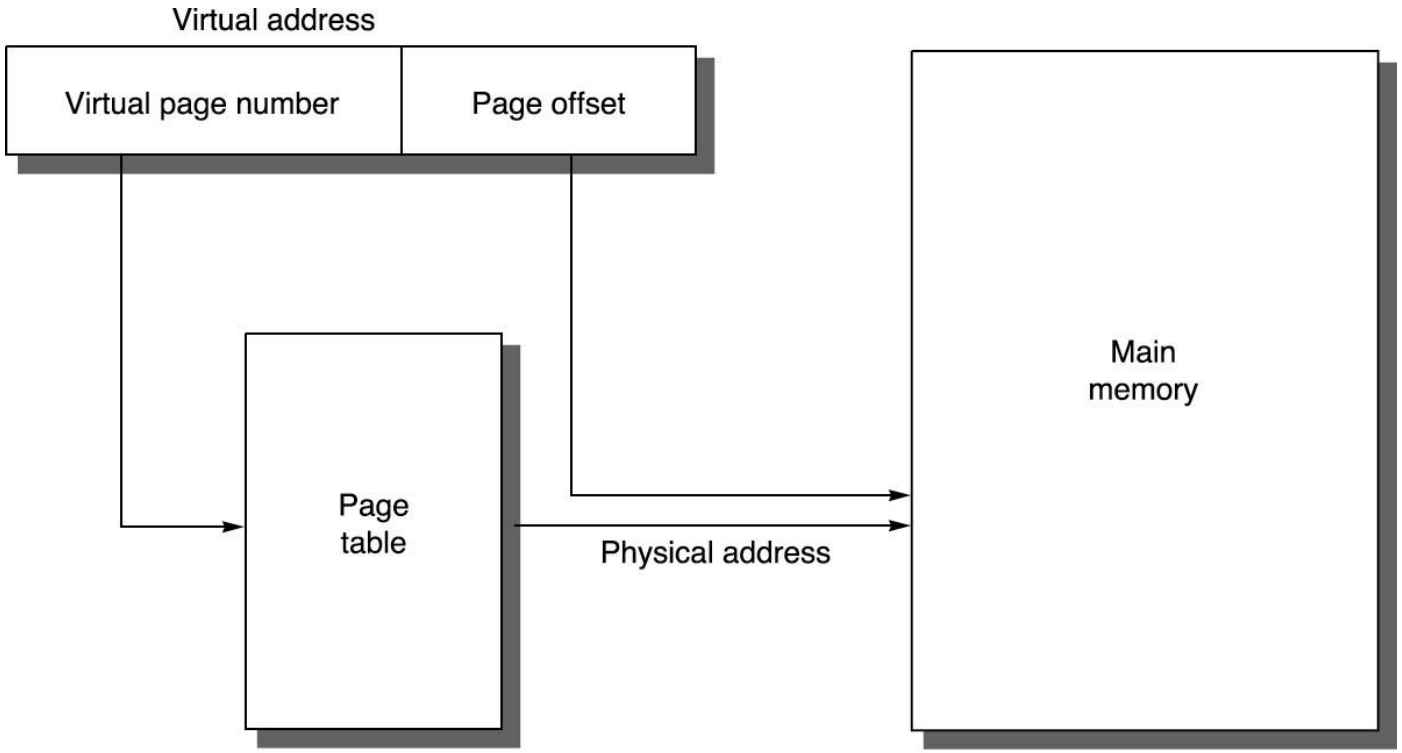
- **Difference between caches and main memory**
  - Cache miss replacement by hardware
  - Virtual memory replacement by operating system -> larger miss penalty
  - Size of processor address => virtual memory size, independent to cache size

2 classes of virtual memory

- **Pages (fixed size blocks): 512 ~ 8192 bytes**
  - Paged address has fixed size address (page# + offset)
- **Segments (variable size) : varies**
  - Segment address has 2 word size address
    - 1 word for segment no.
    - 1 word for offset

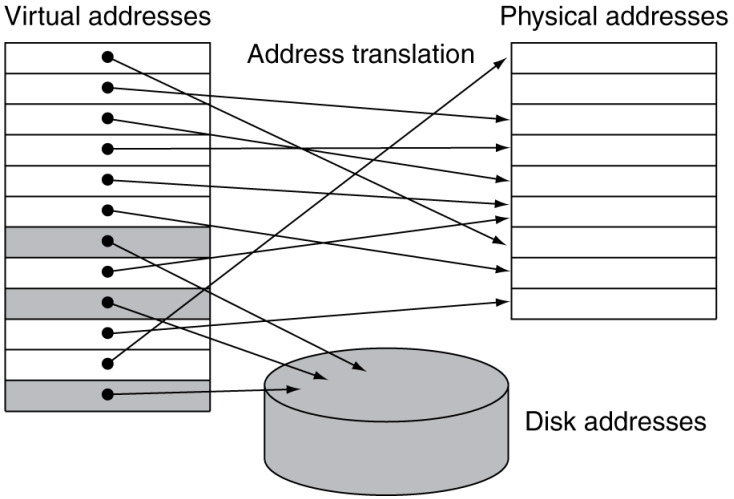


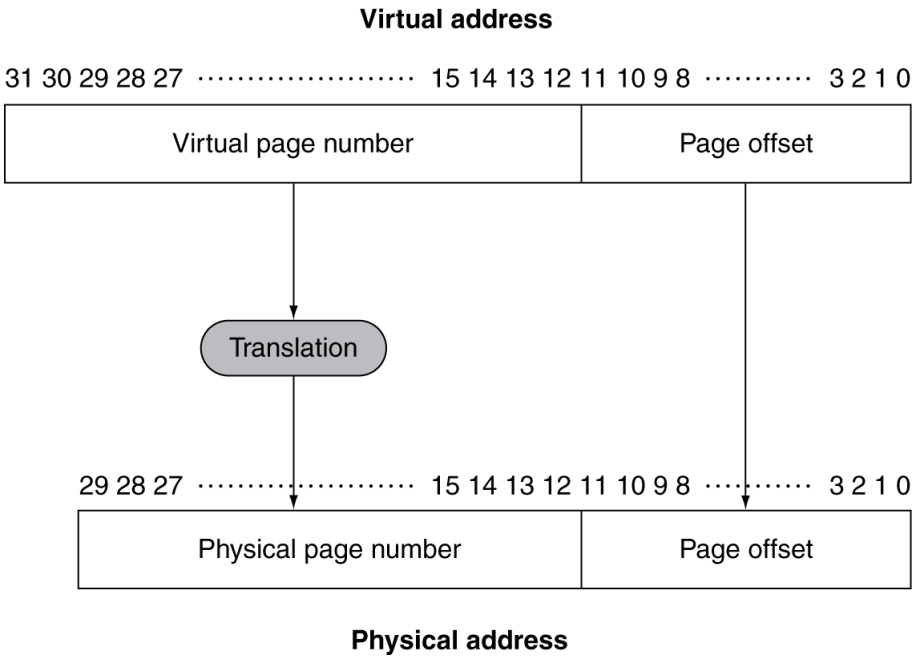
Mapping of a virtual address to a physical address via page table



Address Translation

- **Fixed-size pages (e.g., 4K)**





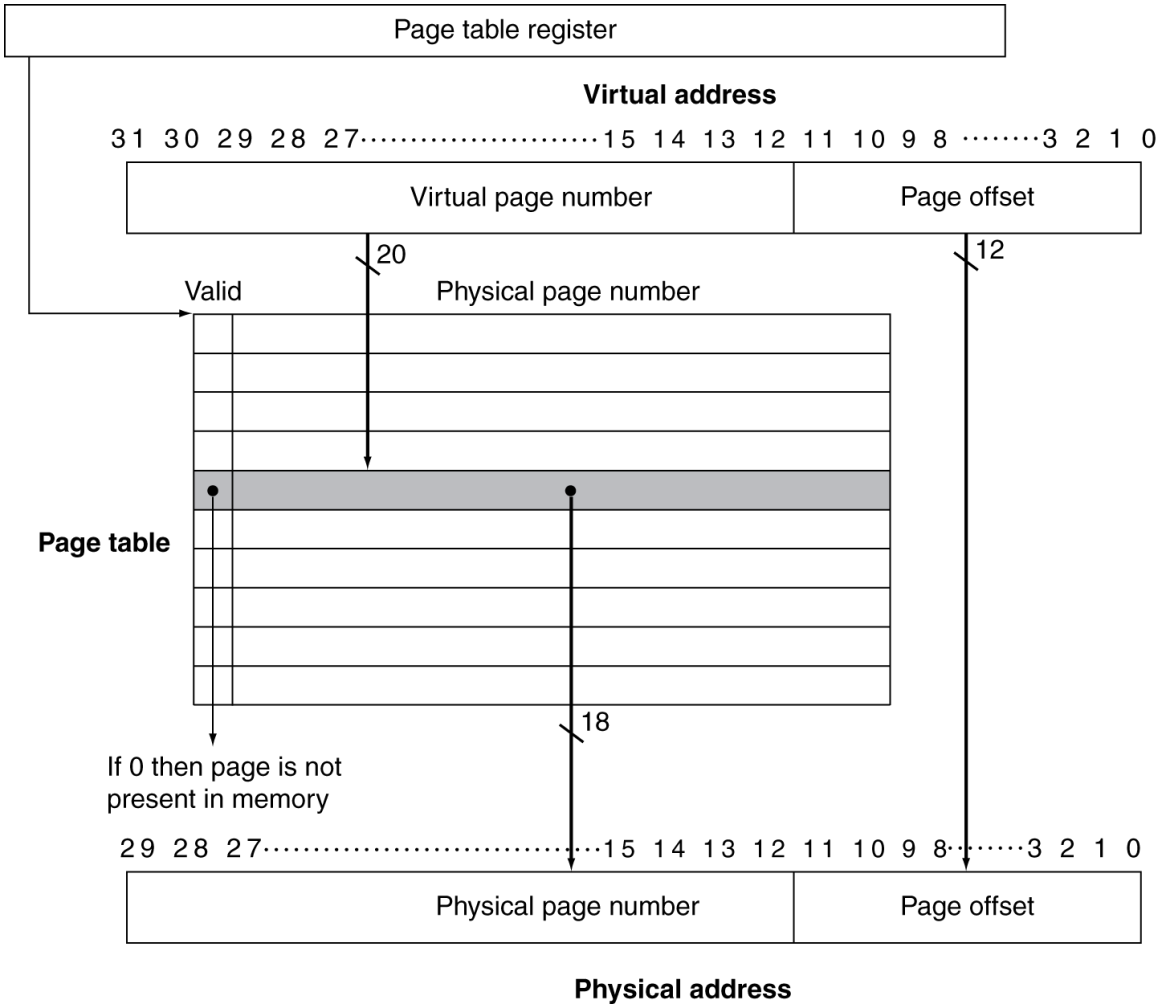
Page Fault Penalty

- **On page fault, the page must be fetched from disk**
  - Takes millions of clock cycles
  - Handled by OS code
- **Try to minimize page fault rate**
  - Fully associative placement
  - Smart replacement algorithms

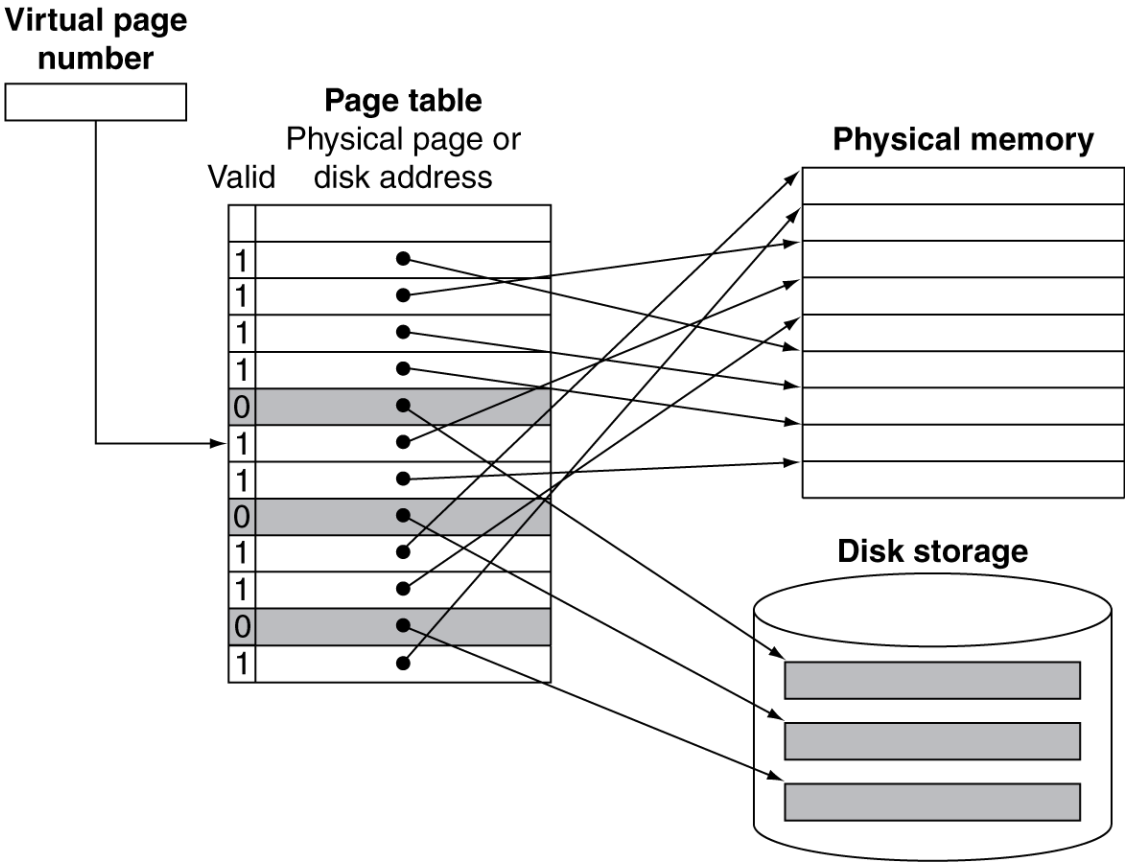
Page Tables

- **Stores placement information**
  - Array of page table entries, indexed by virtual page number
  - Page table register in CPU points to page table in physical memory
- **If page is present in memory**
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, …)
- **If page is not present**
  - PTE can refer to location in swap space on disk

Translation Using a Page Table



Mapping Pages to Storage



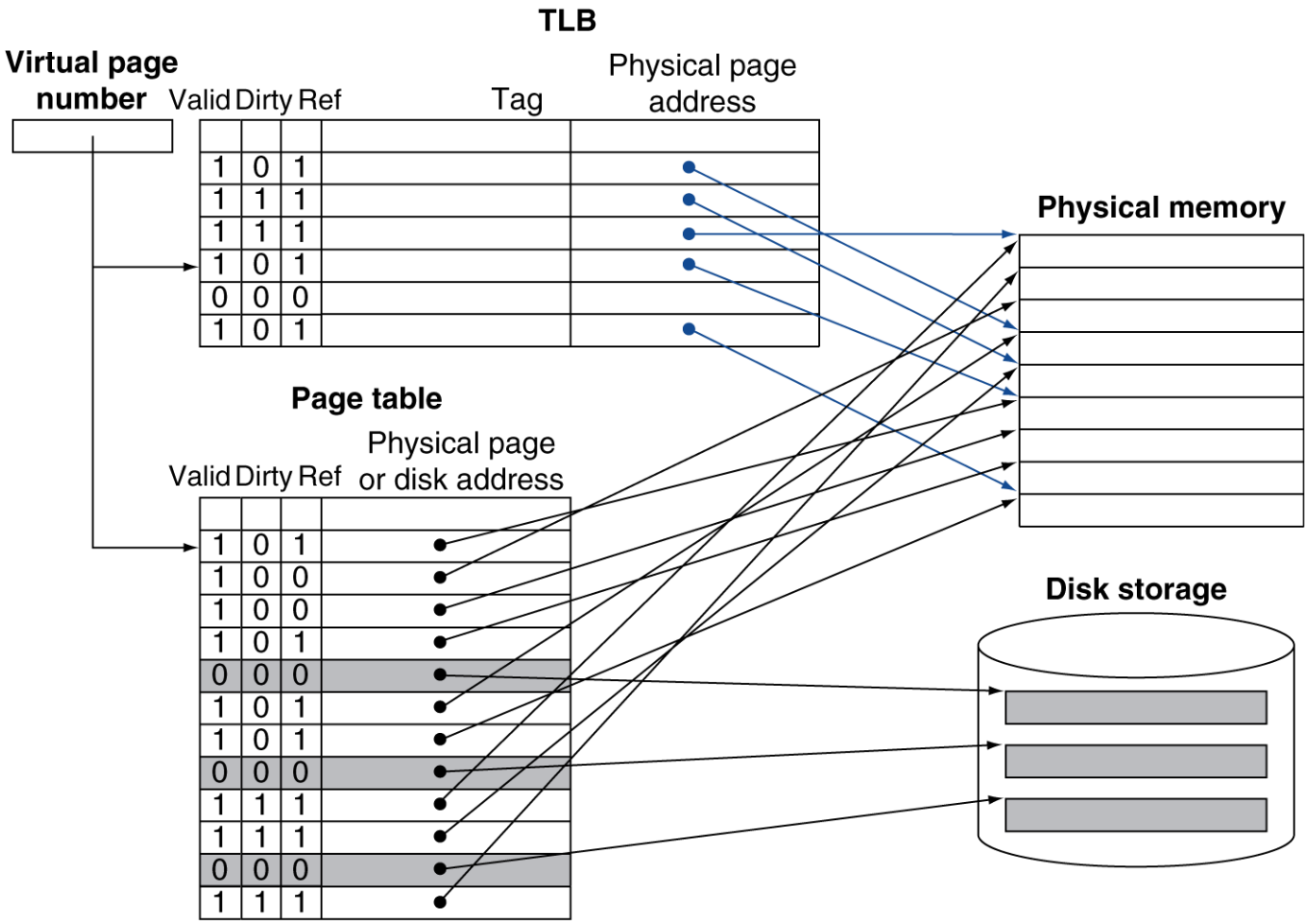
Replacement and Writes

- **To reduce page fault rate, prefer least-recently used (LRU) replacement**
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently
- **Disk writes take millions of cycles**
  - Block at once, not individual locations
  - Write through is impractical
  - Use write-back
  - Dirty bit in PTE set when page is written

Fast Translation Using a TLB

- **Address translation would appear to require extra memory references**
  - One to access the PTE
  - Then the actual memory access
- **But access to page tables has good locality**
  - So use a fast cache of PTEs within the CPU
  - Called a Translation Look-aside Buffer (TLB)
  - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  - Misses could be handled by hardware or software

Fast Translation Using a TLB



TLB Misses

- **If page is in memory**
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
- **If page is not in memory (page fault)**
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

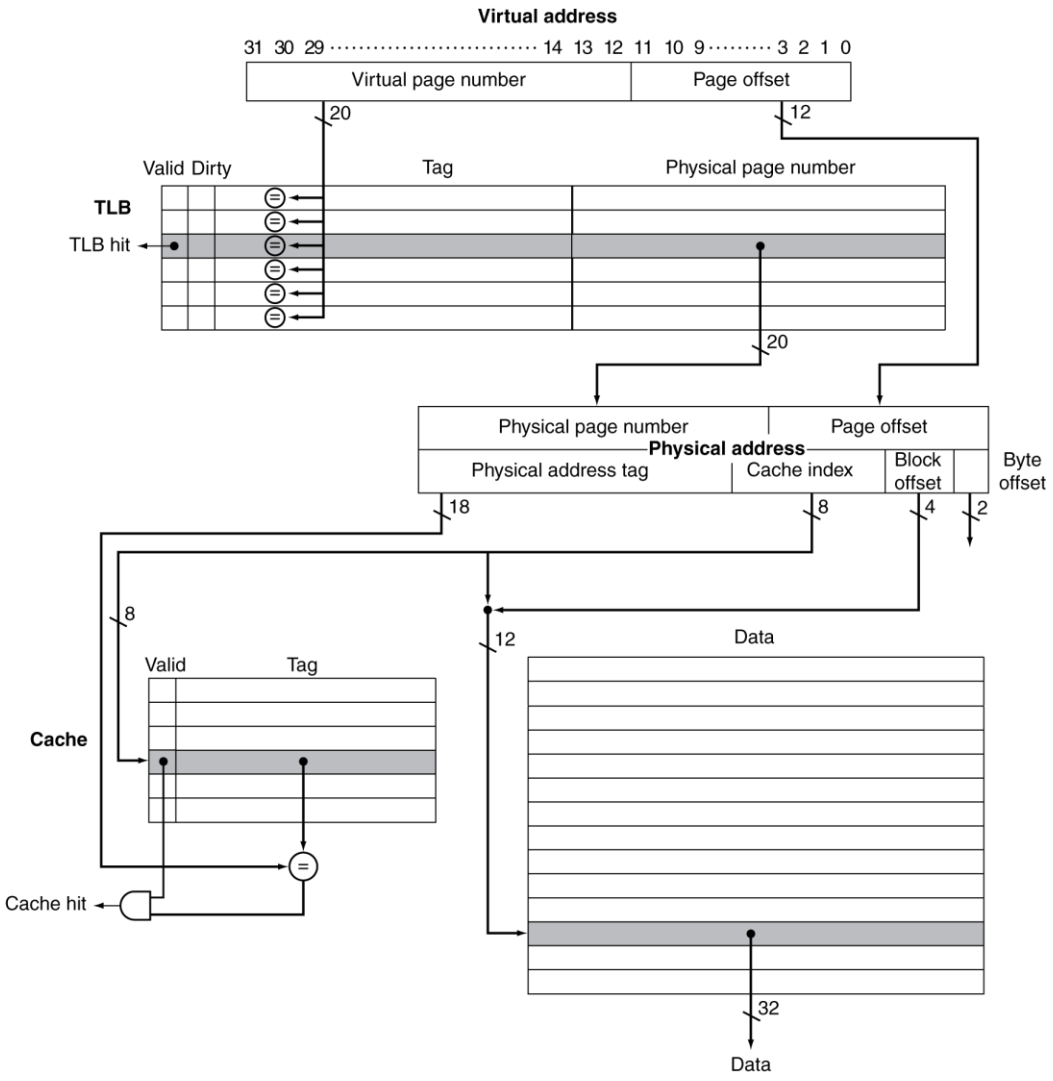
TLB Miss Handler

- **TLB miss indicates**
  - Page present, but PTE not in TLB
  - Page not preset
- **Must recognize TLB miss before destination register overwritten**
  - Raise exception
- **Handler copies PTE from memory to TLB**
  - Then restarts instruction
  - If page not present, page fault will occur

Page Fault Handler

- **Use faulting virtual address to find PTE**
- **Locate page on disk**
- **Choose page to replace**
  - If dirty, write to disk first
- **Read page into memory and update page table**
- **Make process runnable again**
  - Restart from faulting instruction

TLB and Cache Interaction



- **If cache tag uses physical address**
  - Need to translate before cache lookup
- **Alternative: use virtual address tag**
  - Complications due to aliasing
    - Different virtual addresses for shared physical address

Memory Protection

- **Different tasks can share parts of their virtual address spaces**
  - But need to protect against errant access
  - Requires OS assistance
- **Hardware support for OS protection**
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., syscall in MIPS)

A common framework for memory hierarchy (summary)

- **Common principles apply at all levels of the memory hierarchy**
  - Based on notions of caching
- **At each level in the hierarchy**
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy

Block Placement

- **Determined by associativity**
  - Direct mapped (1-way associative)
    - One choice for placement
  - n-way set associative
    - n choices within a set
  - Fully associative
    - Any location
- **Higher associativity reduces miss rate**
  - Increases complexity, cost, and access time

Finding a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

- **Hardware caches**
  - Reduce comparisons to reduce cost
- **Virtual memory**
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate

Replacement

- **Choice of entry to replace on a miss**
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement
- **Virtual memory**
  - LRU approximation with hardware support

Write Policy

- **Write-through**
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer
- **Write-back**
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state
- **Virtual memory**
  - Only write-back is feasible, given disk write latency

Sources of Misses

- **Compulsory misses**
  - First access to a block
- **Capacity misses**
  - Due to finite cache size
  - A replaced block is later accessed again
- **Conflict misses (collision misses)**
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

Cache Design Trade-offs

Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

Virtual Machines

- **Host computer emulates guest operating system and machine resources**
  - Improved isolation of multiple guests
  - Avoids security and reliability problems
  - Aids sharing of resources
- **Virtualization has some performance impact**
  - Feasible with modern high-performance comptuers
- **Examples**
  - IBM VM/370 (1970s technology!)
  - VMWare
  - Microsoft Virtual PC

Virtual Machine Monitor

- **Maps virtual resources to physical resources**
  - Memory, I/O devices, CPUs
- **Guest code runs on native machine in user mode**
  - Traps to VMM on privileged instructions and access to protected resources
- **Guest OS may be different from host OS**
- **VMM handles real I/O devices**
  - Emulates generic virtual I/O devices for guest

Example: Timer Virtualization

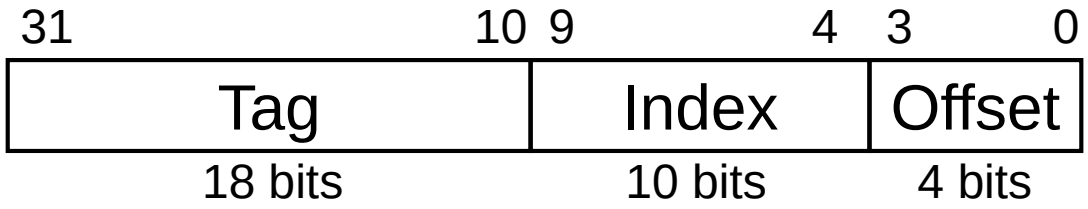
- **In native machine, on timer interrupt**
  - OS suspends current process, handles interrupt, selects and resumes next process
- **With Virtual Machine Monitor**
  - VMM suspends current VM, handles interrupt, selects and resumes next VM
- **If a VM requires timer interrupts**
  - VMM emulates a virtual timer
  - Emulates interrupt for VM when physical timer interrupt occurs

Instruction Set Support

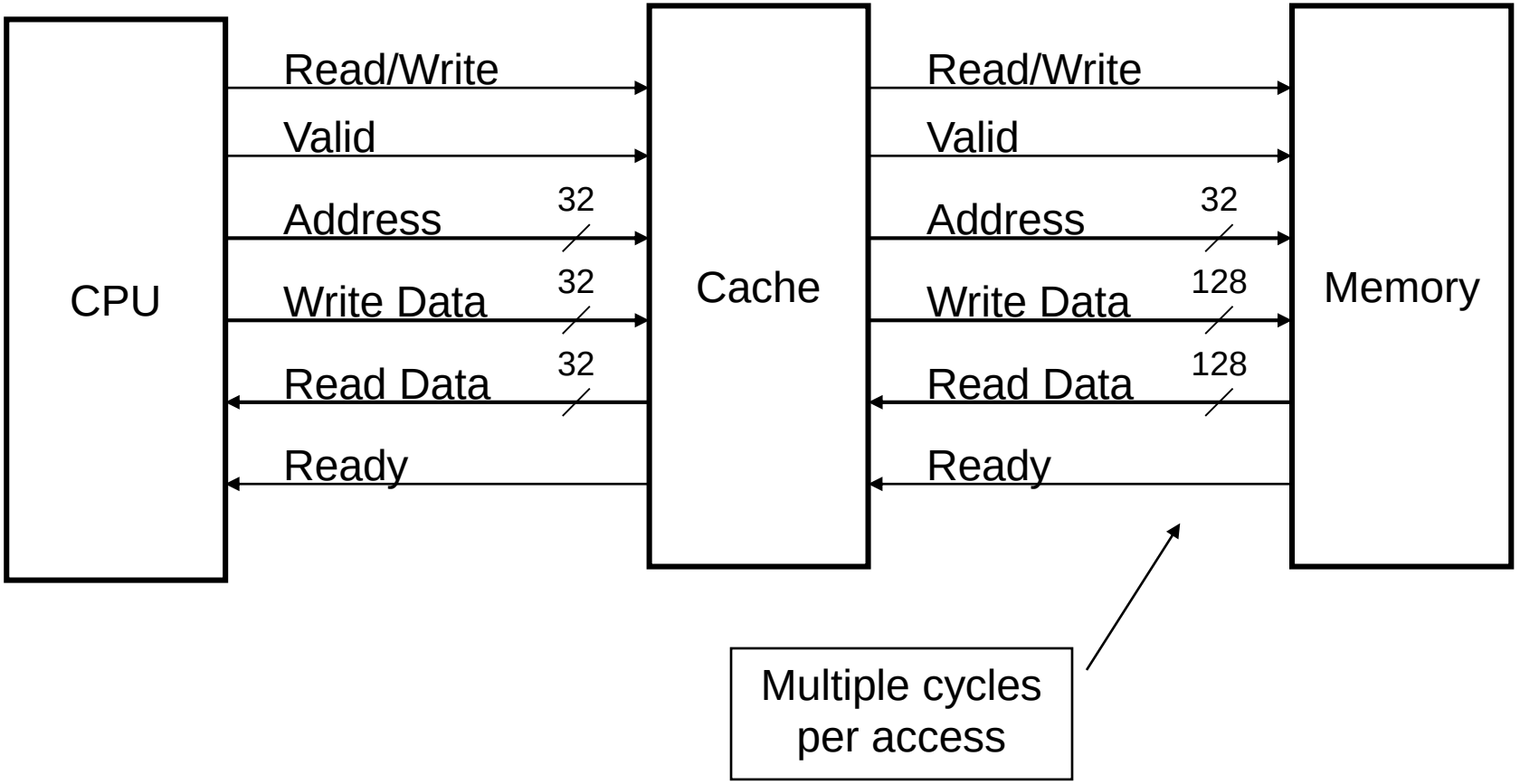
- **User and System modes**
- **Privileged instructions only available in system mode**
  - Trap to system if executed in user mode
- **All physical resources only accessible using privileged instructions**
  - Including page tables, interrupt controls, I/O registers
- **Renaissance of virtualization support**
  - Current ISAs (e.g., x86) adapting

Cache 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

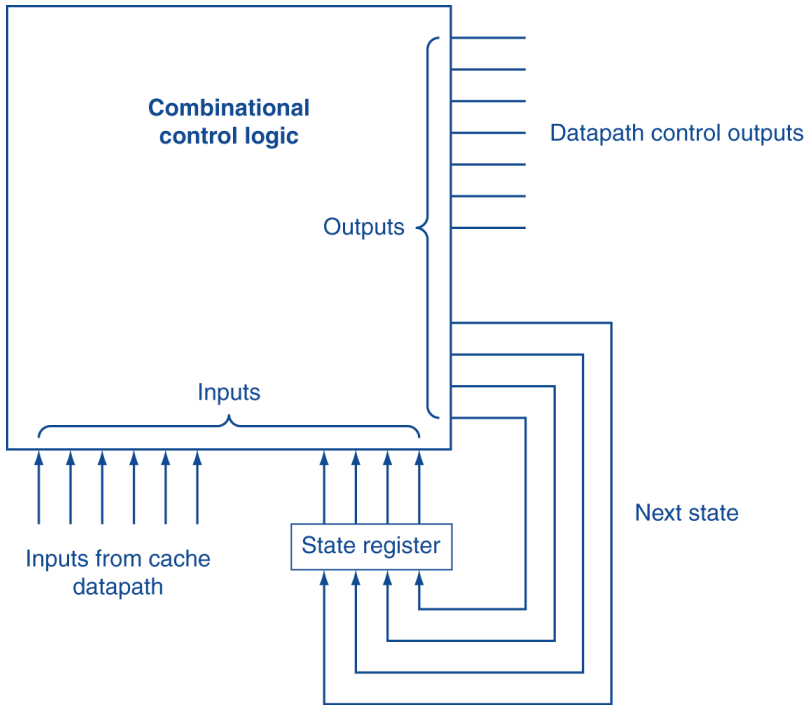


Interface Signals

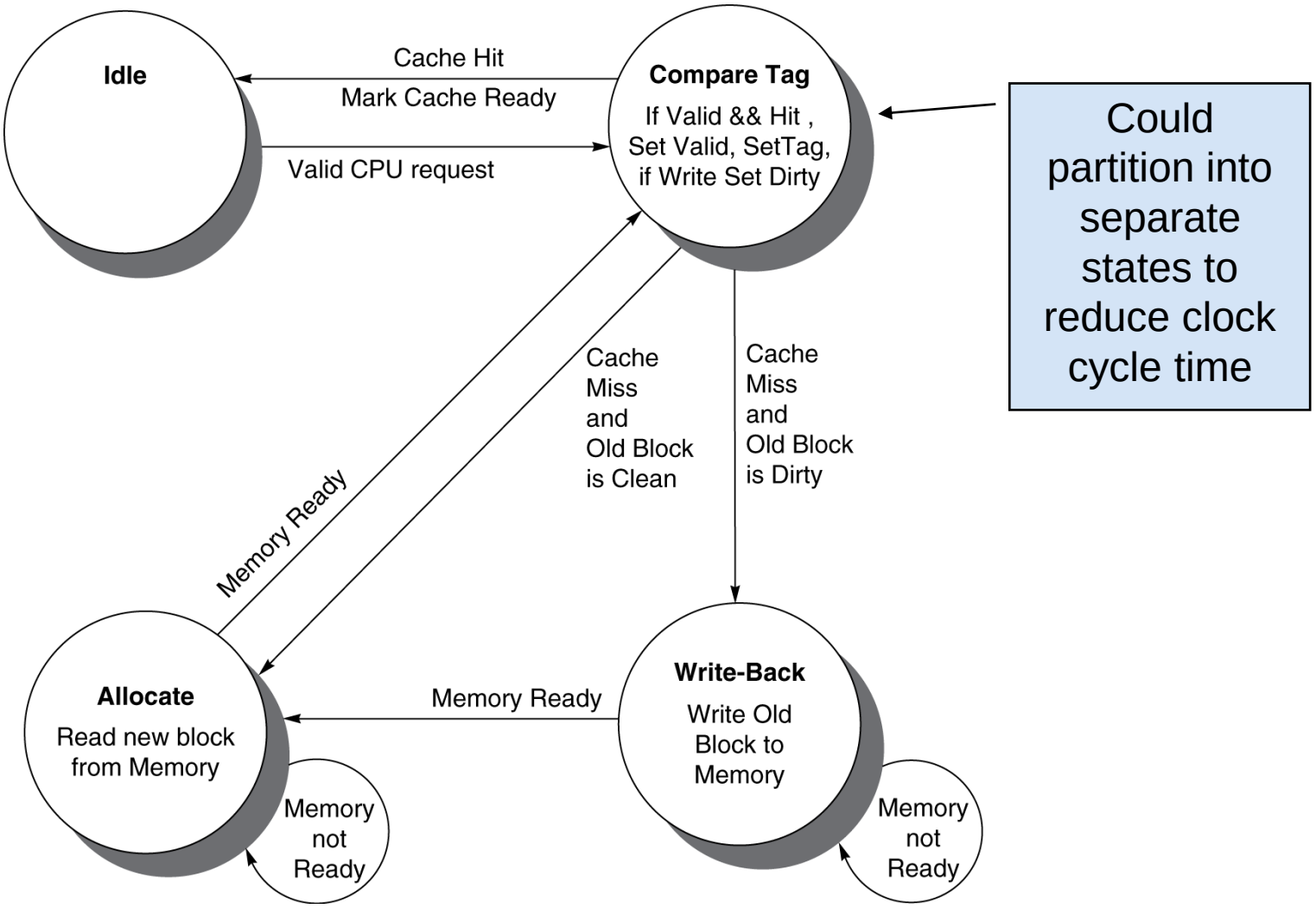


Finite State Machines

- **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
    = f_n (current state,
    current inputs)
- **Control output signals**
  = f_o (current state)



Cache Controller FSM



Cache Coherence Problem

- **Suppose two CPU cores share a physical address space**
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

Coherence Defined

- **Informally: Reads return most recently written value**
- **Formally:**
  - P writes X; P reads X (no intervening writes)
    $\Rightarrow$ read returns written value
  - P_1 writes X; P_2 reads X (sufficiently later)
    $\Rightarrow$ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - P_1 writes X, P_2 writes X
    $\Rightarrow$ all processors see writes in the same order
    - End up with the same final value for X

Cache Coherence Protocols

- **Operations performed by caches in multiprocessors to ensure coherence**
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- **Snooping protocols**
  - Each cache monitors bus reads/writes
- **Directory-based protocols**
  - Caches and memory record sharing status of blocks in a directory

Invalidating Snooping Protocols

- **Cache gets exclusive access to a block when it is to be written**
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

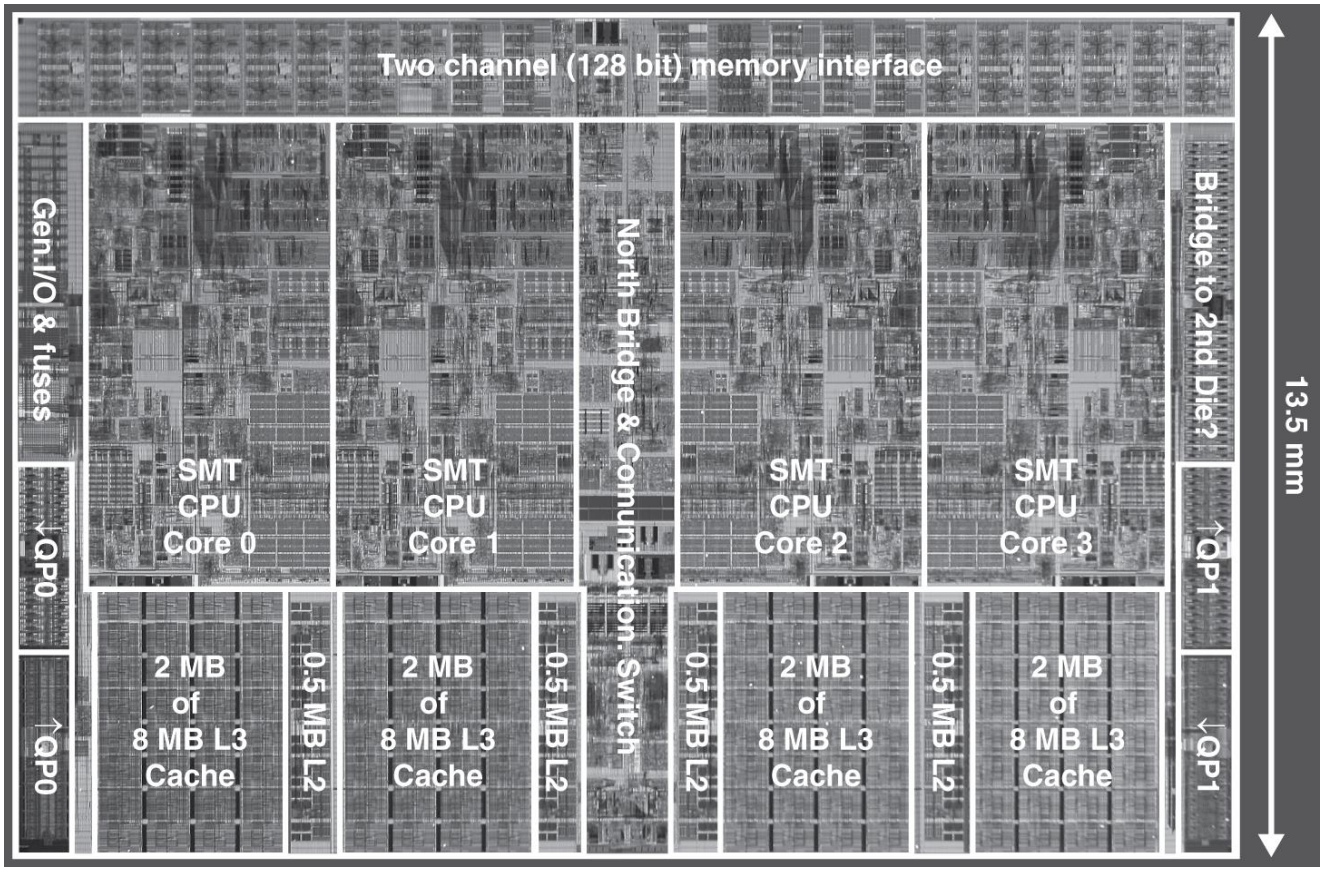
CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

Memory Consistency

- **When are writes seen by other processors**
  - "Seen" means a read returns the written value
  - Can't be instantaneously
- **Assumptions**
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses
- **Consequence**
  - P writes X then writes Y
    $\Rightarrow$ all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

Multilevel On-Chip Caches

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

2-Level TLB Organization

	Intel Nehalem	AMD Opteron X4
Virtual addr	48 bits	48 bits
Physical addr	44 bits	48 bits
Page size	4KB, 2/4MB	4KB, 2/4MB
L1 TLB (per core)	L1 I-TLB: 128 entries for small pages, 7 per thread (2×) for large pages L1 D-TLB: 64 entries for small pages, 32 for large pages Both 4-way, LRU replacement	L1 I-TLB: 48 entries L1 D-TLB: 48 entries Both fully associative, LRU replacement
L2 TLB (per core)	Single L2 TLB: 512 entries 4-way, LRU replacement	L2 I-TLB: 512 entries L2 D-TLB: 512 entries Both 4-way, round-robin LRU
TLB misses	Handled in hardware	Handled in hardware

3-Level Cache Organization

	Intel Nehalem	AMD Opteron X4
L1 caches (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles
L2 unified cache (per core)	256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles

n/a: data not available

Concluding Remarks

- **Fast memories are small, large memories are slow**
  - We really want fast, large memories ☹
  - Caching gives this illusion ☺
- **Principle of locality**
  - Programs use a small part of their memory space frequently
- **Memory hierarchy**
  - L1 cache ↔ L2 cache ↔ … ↔ DRAM memory ↔ disk
- **Memory system design is critical for multiprocessors**