



# *Computer Organization and Architecture*

---

Instructor: Dr. Rushdi Abu  
Zneit

Slide Sources: Patterson &  
Hennessy

# COD Ch. 7

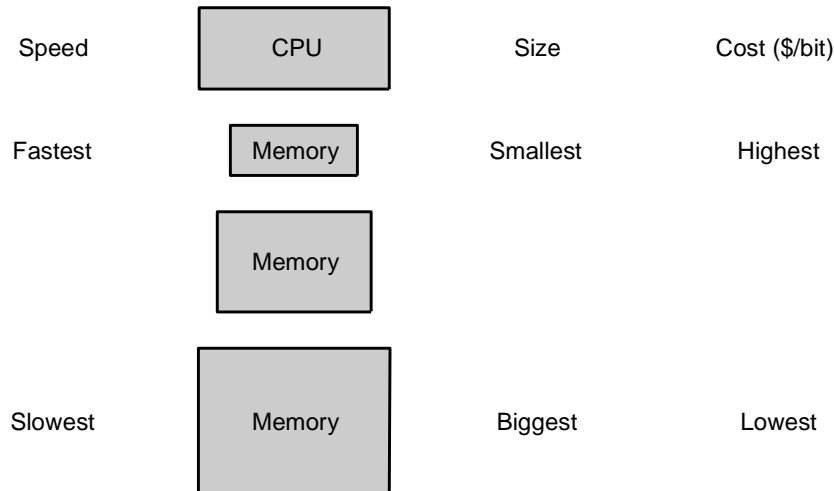


## Large and Fast: Exploiting Memory Hierarchy

---

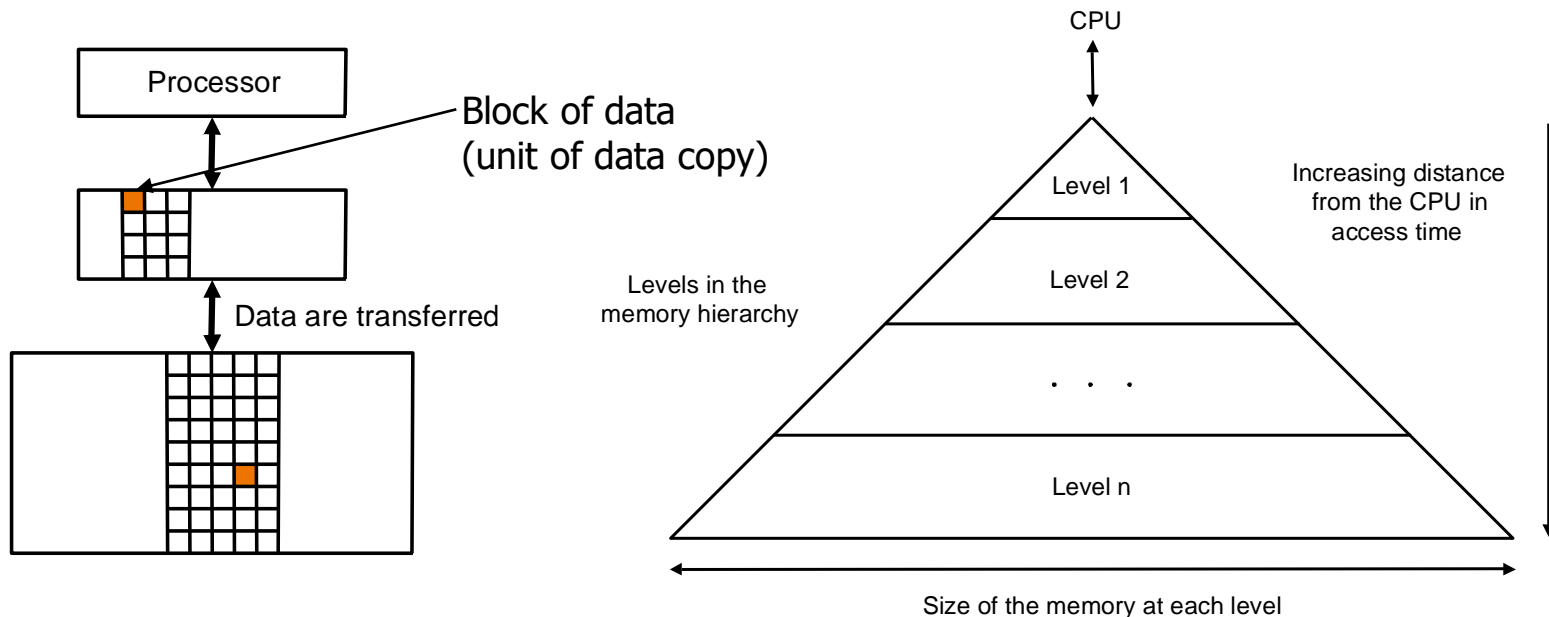
# Memories: Review

- *DRAM* (Dynamic Random Access Memory):
  - value is stored as a charge on capacitor that must be *periodically refreshed*, which is why it is called *dynamic*
  - very small – 1 transistor per bit – but factor of 5 to 10 slower than SRAM
  - used for *main memory*
- *SRAM* (Static Random Access Memory):
  - value is stored on a pair of inverting gates that will *exist indefinitely* as long as there is power, which is why it is called *static*
  - very fast but takes up more space than DRAM – 4 to 6 transistors per bit
  - used for *cache*



# Memory Hierarchy

- Users want large and fast memories...
  - expensive and they don't like to pay...
- Make it seem like they have what they want...
  - *memory hierarchy*
  - hierarchy is *inclusive*, every level is *subset* of lower level
  - performance depends on *hit rates*





# Locality

---

- *Locality* is a principle that makes having a memory hierarchy a good idea
- If an item is referenced then because of
  - *temporal locality*: it will tend to be *again* referenced soon
  - *spatial locality*: *nearby items* will tend to be referenced soon
  - *why does code have locality – consider instruction and data?*



# Hit and Miss

---

- Focus on *any two adjacent* levels – called, **upper** (closer to CPU) and **lower** (farther from CPU) – in the memory hierarchy, because each block copy is always between two adjacent levels
- Terminology:
  - **block**: minimum unit of data to move between levels
  - **hit**: data requested is in upper level
  - **miss**: data requested is not in upper level
  - **hit rate**: fraction of memory accesses that are hits (i.e., found at upper level)
  - **miss rate**: fraction of memory accesses that are not hits
    - **miss rate = 1 – hit rate**
  - **hit time**: time to determine if the access is indeed a hit + time to access and deliver the data from the upper level to the CPU
  - **miss penalty**: time to determine if the access is a miss + time to replace block at upper level with corresponding block at lower level + time to deliver the block to the CPU

# Caches

By simple example

- assume block size = one word of data

X4
X1
X <sub>n-2</sub>
X <sub>n-1</sub>
X2
X3

a. Before the reference to X<sub>n</sub>

X4
X1
X <sub>n-2</sub>
X <sub>n-1</sub>
X2
X <sub>n</sub>
X3

b. After the reference to X<sub>n</sub>

Reference to X<sub>n</sub>  
causes miss so  
it is fetched from  
memory

- Issues:

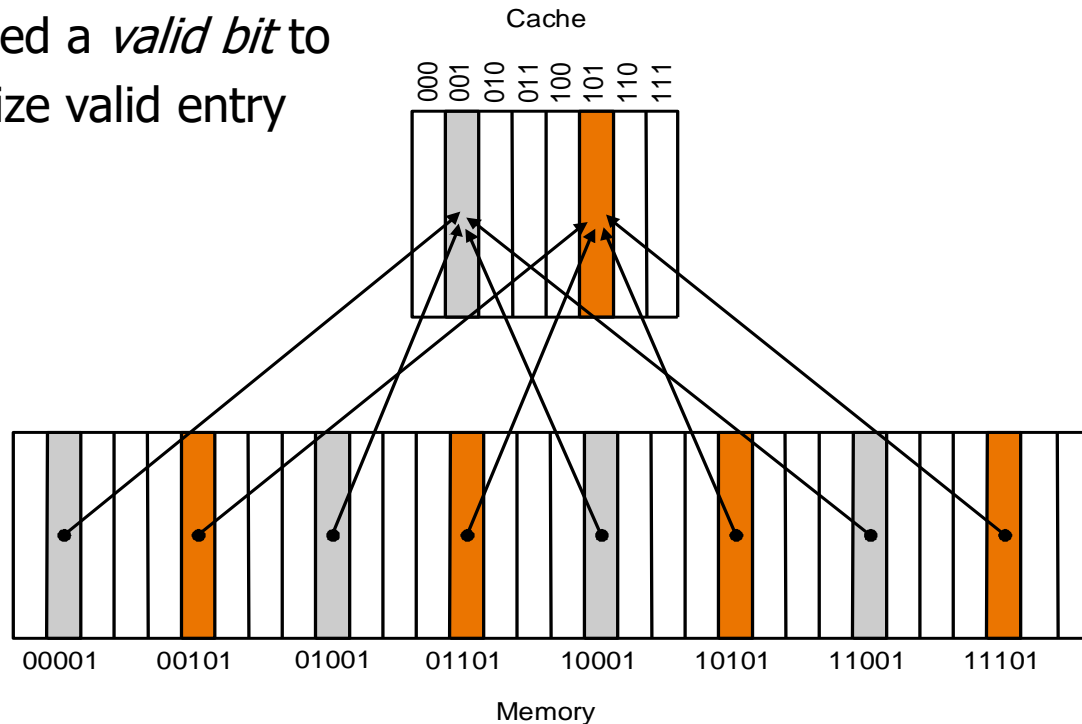
- *how do we know if a data item is in the cache?*
- *if it is, how do we find it?*
- *if not, what do we do?*

- Solution depends on **cache addressing scheme...**

# Direct Mapped Cache

Addressing scheme in *direct mapped* cache:

- cache block address = memory block address *mod* cache size (*unique*)
- if cache size =  $2^m$ , cache address = lower  $m$  bits of  $n$ -bit memory address
- remaining upper  $n-m$  bits kept as *tag bits* at each cache block
- also need a *valid bit* to recognize valid entry







# Accessing Cache

---

- Example:


(0) Initial state:

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

(1) Address referred 10110 (*miss*):

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

(2) Address referred 11010 (*miss*):

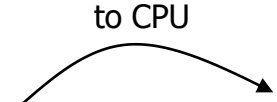


Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem(11010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

(3) Address referred 10110 (*hit*):

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		

to CPU



(4) Address referred 10010 (*miss*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	10	Mem(10010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

- MIPS style:



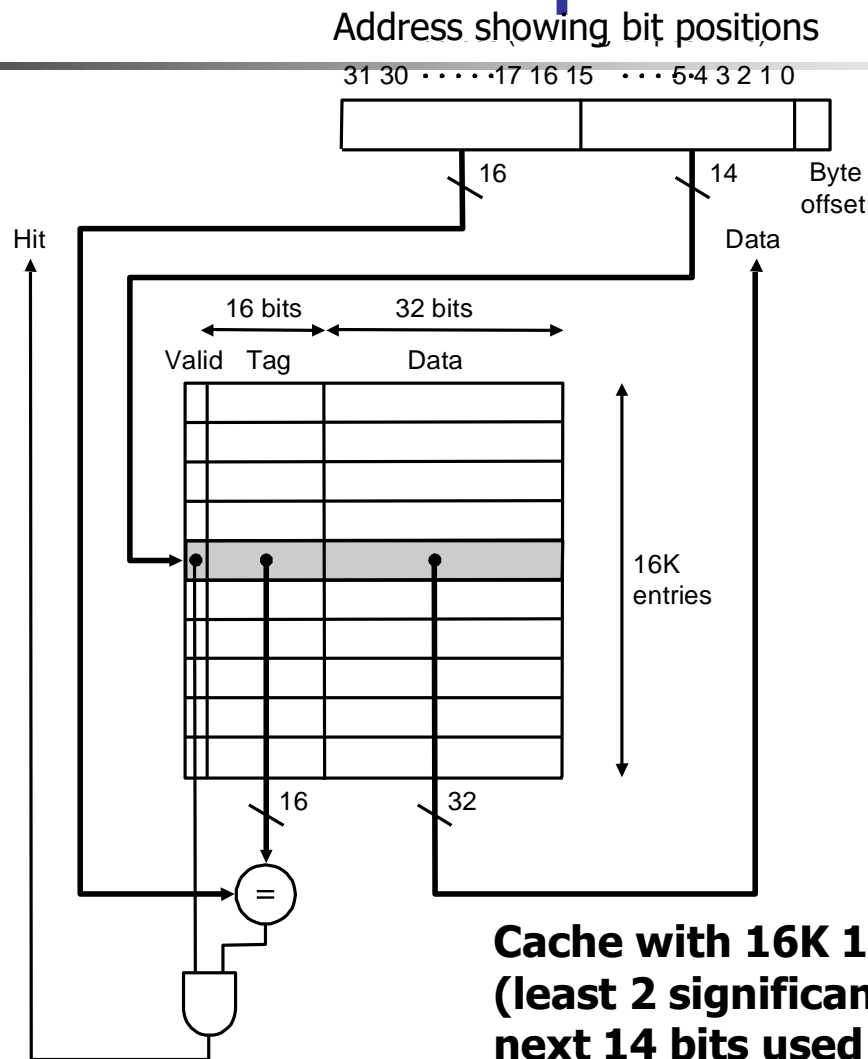


# Cache Read Hit/Miss

---

- *Cache read hit:* no action needed
- *Instruction cache read miss:*
  1. *Send original PC value (current PC – 4, as PC has already been incremented in first step of instruction cycle) to memory*
  2. *Instruct main memory to perform read and wait for memory to complete access – stall on read*
  3. *After read completes write cache entry*
  4. *Restart instruction execution at first step to refetch instruction*
- *Data cache read miss:*
  - Similar to instruction cache miss
  - To reduce data miss penalty allow processor to execute instructions while waiting for the read to complete *until* the word is required – *stall on use* (why won't this work for instruction misses?)

# DECStation 3100 Cache (MIPS R2000 processor)



**Cache with 16K 1-word blocks: *byte offset* (least 2 significant bits) is ignored and next 14 bits used to index into cache**



# Cache Write Hit/Miss

---

## ■ *Write-through* scheme

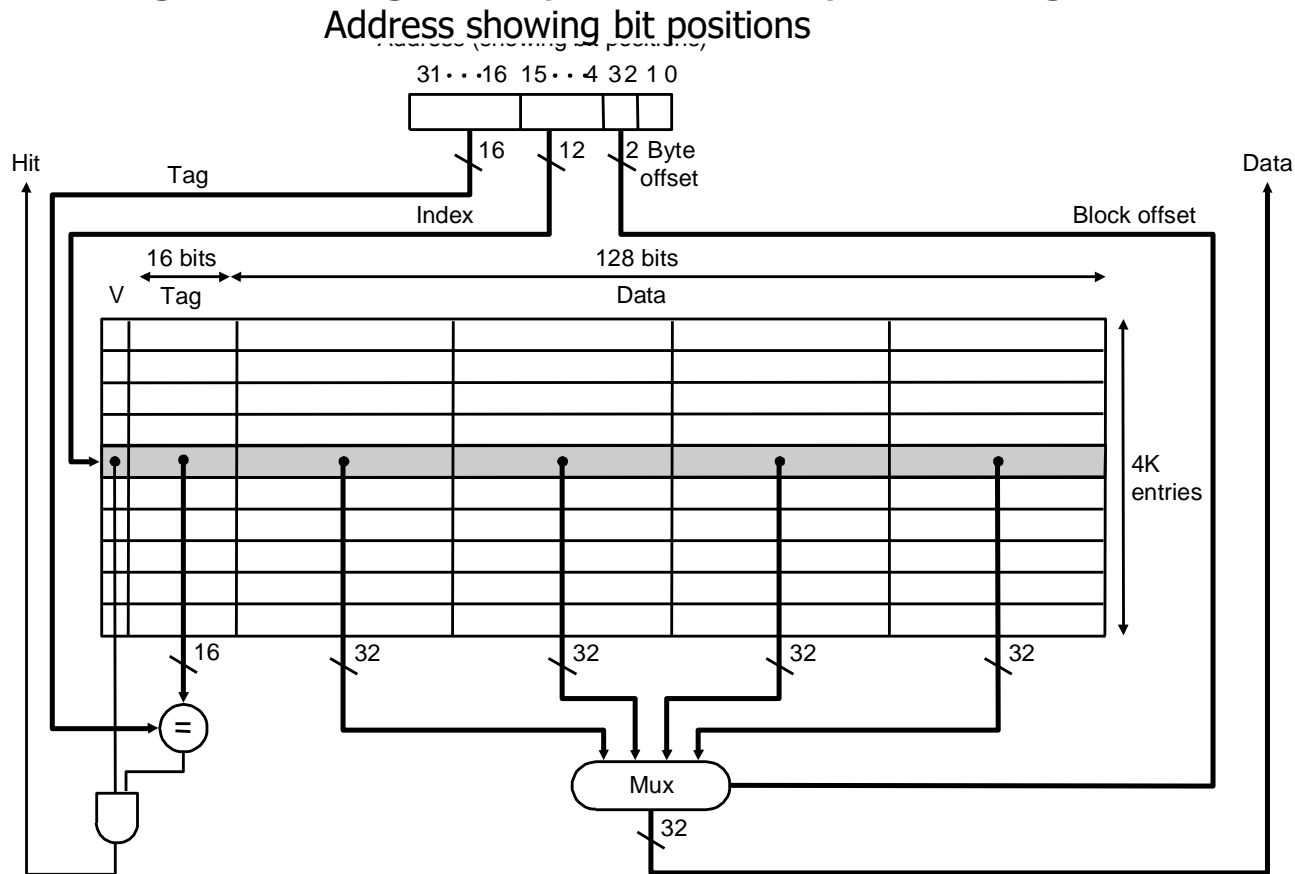
- on *write hit*: replace data in cache *and* memory with *every* write hit to avoid *inconsistency*
- on *write miss*: write the word into cache *and* memory – obviously no need to read missed word from memory!
- Write-through is slow because of always required memory write
  - performance is improved with a *write buffer* where words are stored while waiting to be written to memory – processor can continue execution until write buffer is full
  - when a word in the write buffer completes writing into main that buffer slot is freed and becomes available for future writes
  - DEC 3100 write buffer has 4 words

## ■ *Write-back* scheme

- write the data block *only* into the cache and *write-back* the block to main *only when* it is replaced in cache
- more efficient than write-through, more complex to implement

# Direct Mapped Cache: Taking Advantage of Spatial Locality

- Taking advantage of spatial locality with *larger* blocks:



**Cache with 4K 4-word blocks: *byte offset* (least 2 significant bits) is ignored, next 2 bits are *block offset*, and the next 12 bits are used to index into cache**



# Direct Mapped Cache: Taking Advantage of Spatial Locality

---

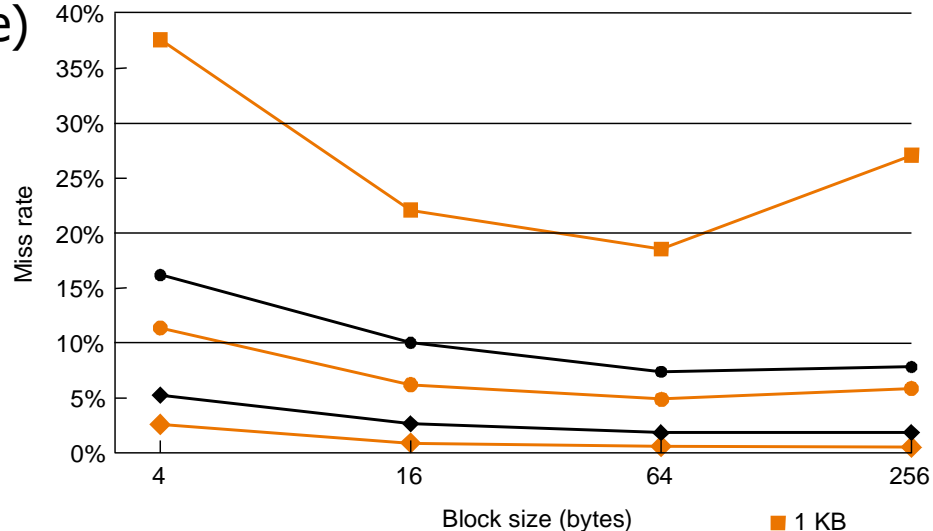
- *Cache replacement* in large (multiword) blocks:
  - word *read miss*: read entire block from main memory
  - word *write miss*: *cannot* simply write word and tag! *Why?!*
  - writing in a *write-through* cache:
    - if *write hit*, i.e., tag of requested address and cache entry are equal, continue as for 1-word blocks by replacing word and writing block to both cache and memory
    - if *write miss*, i.e., tags are unequal, fetch block from memory, replace word that caused miss, and write block to both cache and memory
    - therefore, unlike case of 1-word blocks, a write miss with a multiword block causes a memory read



# Direct Mapped Cache: Taking Advantage of Spatial Locality

Miss rate falls at first with increasing block size as expected, but, as block size becomes a large fraction of total cache size, miss rate may go up because

- there are few blocks
- competition for blocks increases
- blocks get ejected before most of their words are accessed (*thrashing* in cache)



**Miss rate vs. block size for various cache sizes**

■ 1 KB  
● 8 KB  
● 16 KB  
◆ 64 KB  
◆ 256 KB



# Example Problem

---

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 1-word block size, assuming a 32-bit address?*
- Cache data = 128 KB =  $2^{17}$  bytes =  $2^{15}$  words =  $2^{15}$  blocks
- Cache entry size = block data bits + tag bits + valid bit  
$$= 32 + (32 - 15 - 2) + 1 = 48 \text{ bits}$$
- Therefore, cache size =  $2^{15} \times 48 \text{ bits} =$   
 $2^{15} \times (1.5 \times 32) \text{ bits} = 1.5 \times 2^{20} \text{ bits} = 1.5 \text{ Mbits}$ 
  - data bits in cache =  $128 \text{ KB} \times 8 = 1 \text{ Mbits}$
  - total cache size/actual cache data = 1.5



# Example Problem

---

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 4-word block size, assuming a 32-bit address?*
- Cache size = 128 KB =  $2^{17}$  bytes =  $2^{15}$  words =  $2^{13}$  blocks
- Cache entry size = block data bits + tag bits + valid bit  
$$= 128 + (32 - 13 - 2 - 2) + 1 = 144 \text{ bits}$$
- Therefore, cache size =  $2^{13} \times 144 \text{ bits} =$   
 $2^{13} \times (1.25 \times 128) \text{ bits} = 1.25 \times 2^{20} \text{ bits} = 1.25 \text{ Mbits}$ 
  - data bits in cache =  $128 \text{ KB} \times 8 = 1 \text{ Mbits}$
  - total cache size/actual cache data = 1.25



# Example Problem

---

- *Consider a cache with 64 blocks and a **block size of 16 bytes**. What block number does byte address 1200 map to?*
- As block size = 16 bytes:  
byte address 1200  $\Rightarrow$  block address  $\lfloor 1200/16 \rfloor = 75$
- As cache size = 64 blocks:  
block address 75  $\Rightarrow$  cache block  $(75 \bmod 64) = 11$



# Improving Cache Performance

- Use split caches for instruction and data because there is more spatial locality in instruction references:

Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

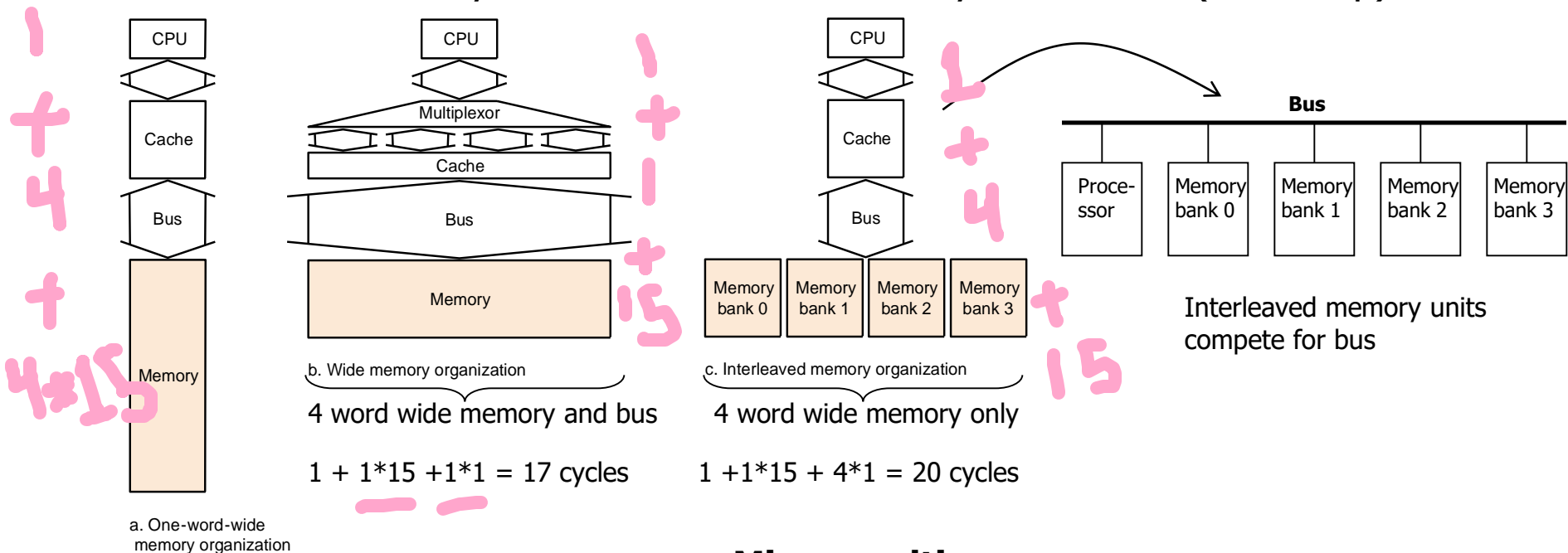
**Miss rates for gcc and spice in a MIPS R2000  
with one and four word block sizes**

- Make reading multiple words (higher bandwidth) possible by increasing physical or logical width of the system...

# Improving Cache Performance by Increasing Bandwidth

Assume:

- cache block of 4 words
- 1 clock cycle to send address to memory address buffer (1 bus trip)
- 15 clock cycles for each memory data access
- 1 clock cycle to send data to memory data buffer (1 bus trip)



**Miss penalties**



# Performance

---

- Simplified model assuming equal read and write miss penalties:
  - $\text{CPU time} = (\text{execution cycles} + \text{memory stall cycles}) \times \text{cycle time}$
  - $\text{memory stall cycles} = \text{memory accesses} \times \text{miss rate} \times \text{miss penalty}$
- Therefore, two ways to improve performance in cache:
  - decrease miss rate
  - decrease miss penalty
  - *what happens if we increase block size?*



# Example Problems

---

- Assume for a given machine and program:
  - instruction cache miss rate 2%
  - data cache miss rate 4%
  - miss penalty always 40 cycles
  - CPI of 2 without memory stalls
  - frequency of load/stores 36% of instructions
- 1. *How much faster is a machine with a perfect cache that never misses?*
- 2. *What happens if we speed up the machine by reducing its CPI to 1 without changing the clock rate?*
- 3. *What happens if we speed up the machine by doubling its clock rate, but if the absolute time for a miss penalty remains same?*





# Solution

---

1.

- Assume instruction count =  $I$
- Instruction miss cycles =  $I \times 2\% \times 40 = 0.8 \times I$
- Data miss cycles =  $I \times 36\% \times 4\% \times 40 = 0.576 \times I$
- So, total memory-stall cycles =  $0.8 \times I + 0.576 \times I = 1.376 \times I$ 
  - in other words, 1.376 stall cycles per instruction
- Therefore, CPI with memory stalls =  $2 + 1.376 = 3.376$
- Assuming instruction count and clock rate remain same for a perfect cache and a cache that misses:  
CPU time with stalls / CPU time with perfect cache  
 $= 3.376 / 2 = 1.688$
- Performance with a perfect cache is better by a factor of 1.688



## Solution (cont.)

---

2.

- CPI without stall = 1
- CPI with stall =  $1 + 1.376 = 2.376$  (clock has not changed so stall cycles per instruction remains same)
- CPU time with stalls / CPU time with perfect cache  
= CPI with stall / CPI without stall  
= 2.376
- Performance with a perfect cache is better by a factor of 2.376
- Conclusion: with higher CPI cache misses “hurt more” than with lower CPI



## Solution (cont.)

---

3.

- With doubled clock rate, miss penalty =  $2 \times 40 = 80$  clock cycles
- Stall cycles per instruction =  $(I \times 2\% \times 80) + (I \times 36\% \times 4\% \times 80)$   
 $= 2.752 \times I$
- So, faster machine with cache miss has  $\text{CPI} = 2 + 2.752 = 4.752$
- CPU time with stalls / CPU time with perfect cache  
 $= \text{CPI with stall} / \text{CPI without stall}$   
 $= 4.752 / 2 = 2.376$
- Performance with a perfect cache is better by a factor of 2.376
- Conclusion: with higher clock rate cache misses “hurt more” than with lower clock rate

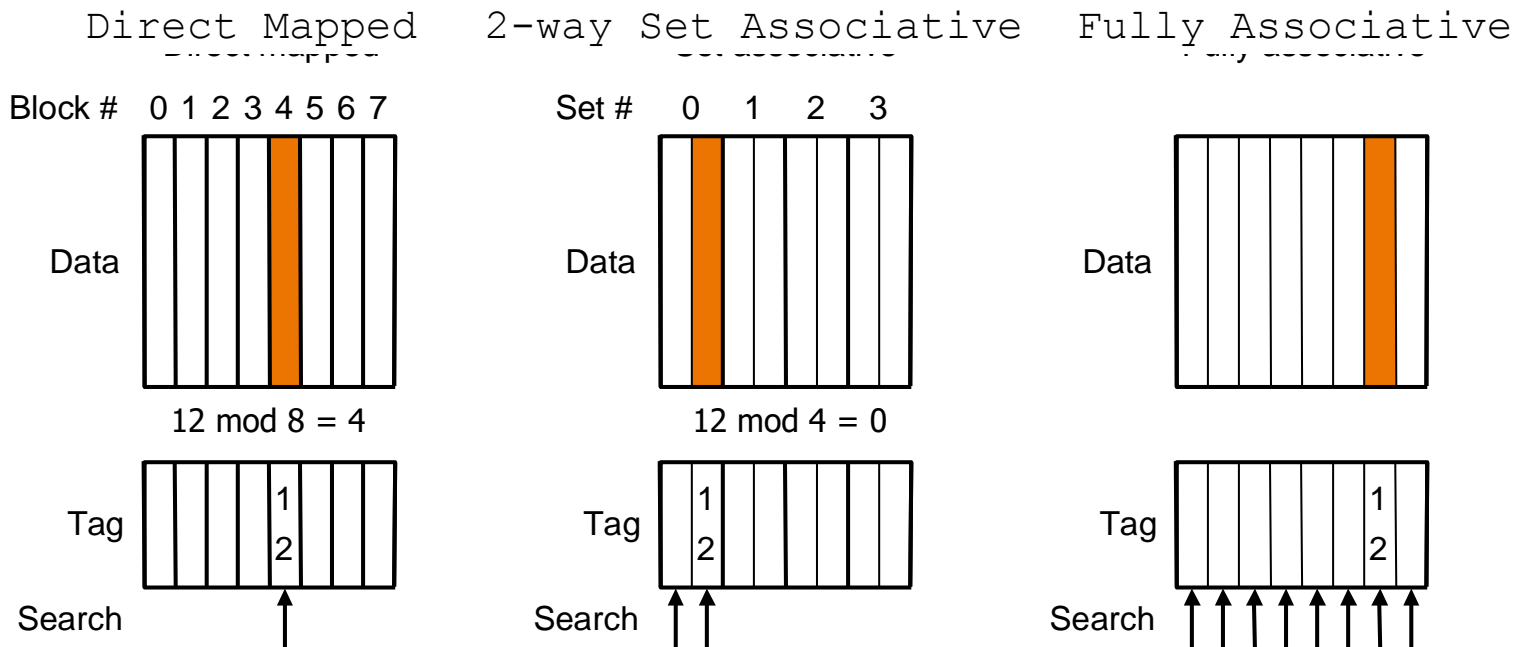


# Decreasing Miss Rates with Associative Block Placement

---

- *Direct mapped*: one *unique* cache location for each memory block
  - cache block address = memory block address *mod* cache size
- *Fully associative*: each memory block can locate *anywhere* in cache
  - *all* cache entries are searched (in parallel) to locate block
- *Set associative*: each memory block can place in a *unique set* of cache locations – if the set is of size *n* it is *n*-way set-associative
  - cache set address = memory block address *mod* number of sets in cache
  - all cache entries in the corresponding set are searched (*in parallel*) to locate block
- Increasing degree of associativity
  - *reduces miss rate*
  - *increases hit time* because of the parallel search and then fetch

# Decreasing Miss Rates with Associative Block Placement



**Location of a memory block with address 12 in a cache with 8 blocks with different degrees of associativity**

# Decreasing Miss Rates with Associative Block Placement

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

**Configurations of an 8-block cache with different degrees of associativity**



# Example Problems

---

- *Find the number of misses for a cache with four 1-word blocks given the following sequence of memory block accesses:*  
$$0, 8, 0, 6, 8,$$
*for each of the following cache configurations*
  1. *direct mapped*
  2. *2-way set associative (use LRU replacement policy)*
  3. *fully associative*
- **Note about LRU replacement**
  - in a 2-way set associative cache LRU replacement can be implemented with one bit at each set whose value indicates the mostly recently referenced block



# Solution

- 1 (direct-mapped)

Block address	Cache block
0	0 ( $= 0 \bmod 4$ )
6	2 ( $= 6 \bmod 4$ )
8	0 ( $= 8 \bmod 4$ )

## Block address translation in direct-mapped cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Cache contents after each reference – red indicates new entry added

- 5 misses





# Solution (cont.)

- 2 (two-way set-associative)

Block address	Cache set
0	0 ( $= 0 \bmod 2$ )
6	0 ( $= 6 \bmod 2$ )
8	0 ( $= 8 \bmod 2$ )

## Block address translation in a two-way set-associative cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Cache contents after each reference – red indicates new entry added

- Four misses



# Solution (cont.)

---

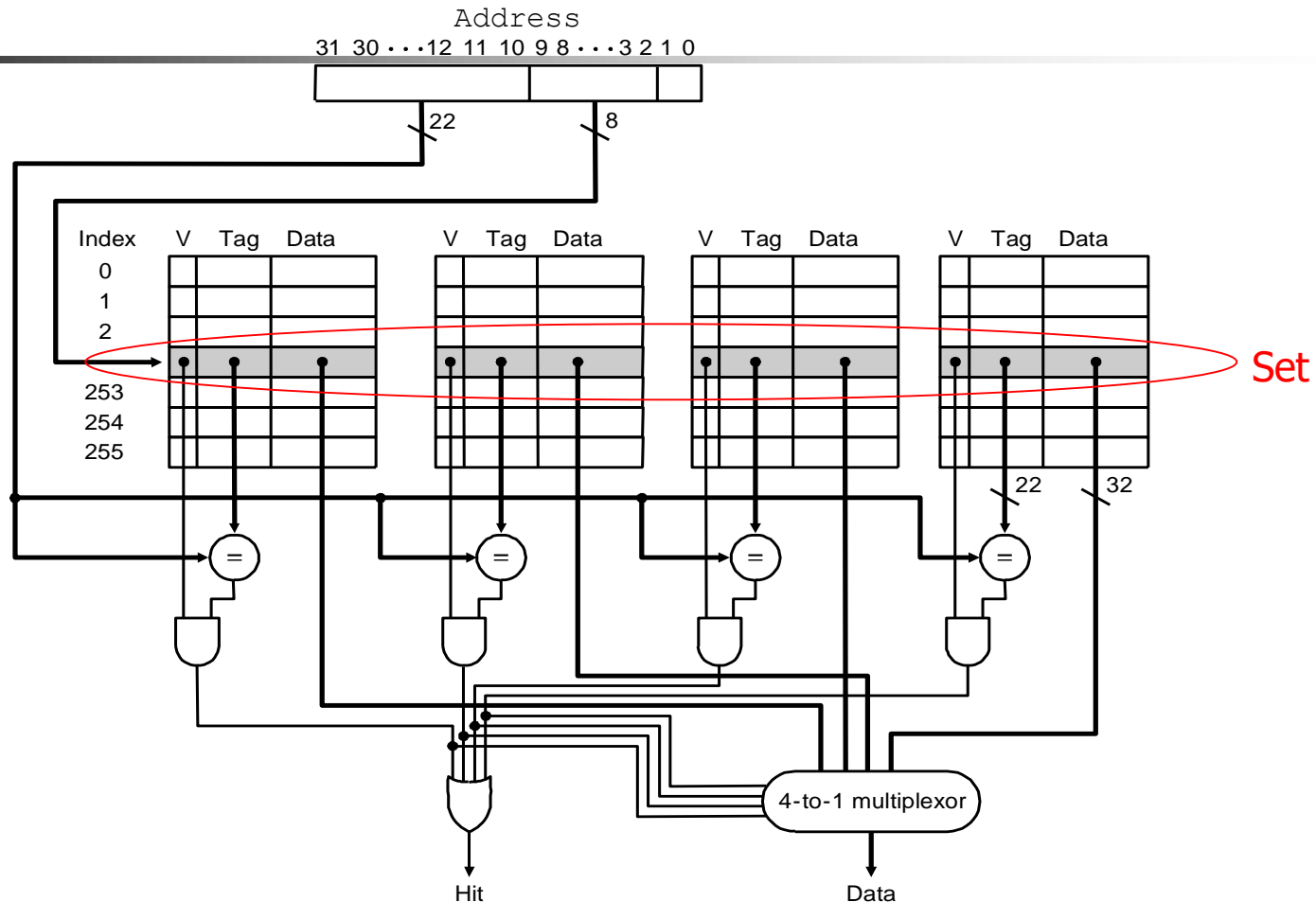
- 3 (fully associative)

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

**Cache contents after each reference – red indicates new entry added**

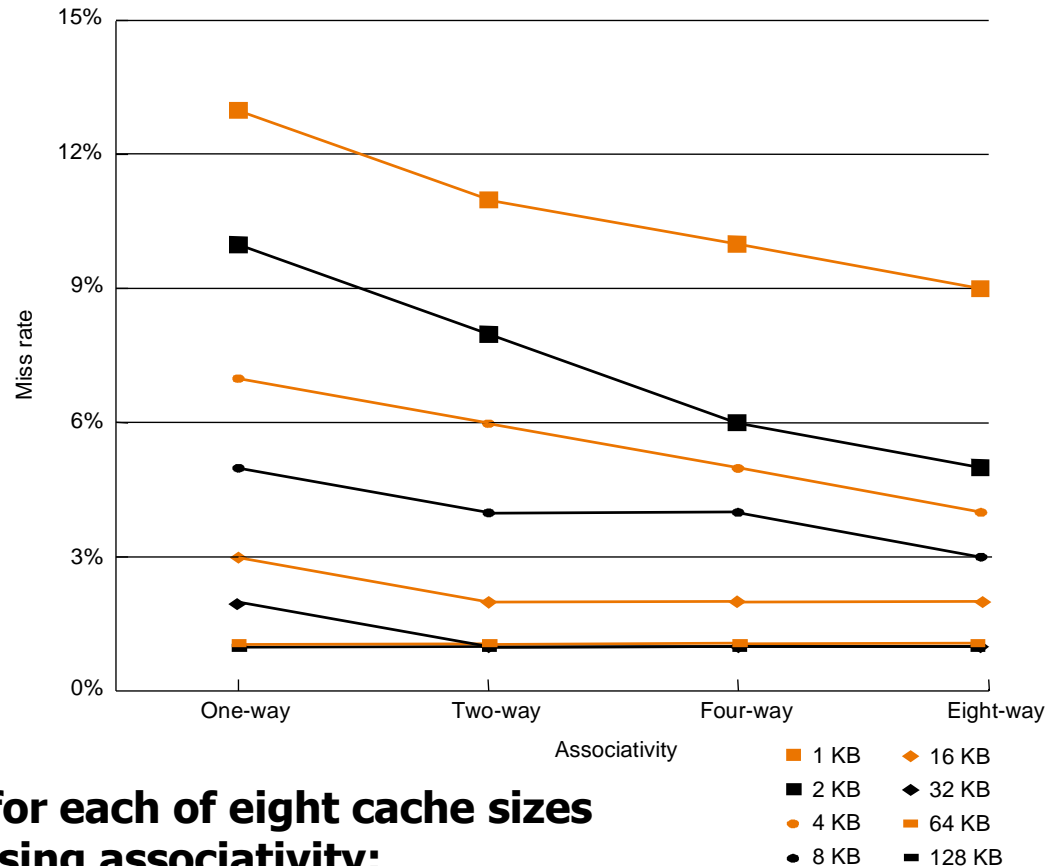
- 3 misses

# Implementation of a Set-Associative Cache



**4-way set-associative cache with 4 comparators and one 4-to-1 multiplexor:  
size of cache is 1K blocks = 256 sets \* 4-block set size**

# Performance with Set-Associative Caches



**Miss rates for each of eight cache sizes  
with increasing associativity:  
data generated from SPEC92 benchmarks  
with 32 byte block size for all caches**



# Decreasing Miss Penalty with Multilevel Caches

- Add a *second-level* cache
  - primary cache is on the same chip as the processor
  - use SRAMs to add a second-level cache, sometimes off-chip, *between main memory and the first-level cache*
  - if miss occurs in primary cache second-level cache is accessed
  - if data is found in second-level cache miss penalty is access time of second-level cache which is much less than main memory access time
  - if miss occurs again at second-level then main memory access is required and large miss penalty is incurred
- Design considerations using two levels of caches:
  - try and optimize the *hit time on the 1<sup>st</sup> level cache* to reduce clock cycle
  - try and optimize the *miss rate on the 2<sup>nd</sup> level cache* to reduce memory access penalties
  - In other words, 2<sup>nd</sup> level allows 1<sup>st</sup> level to go for speed without “worrying” about failure...



# Example Problem

---

- Assume a 500 MHz machine with
  - base CPI 1.0
  - main memory access time 200 ns.
  - miss rate 5%
- *How much faster will the machine be if we add a second-level cache with 20ns access time that decreases the miss rate to 2%?*



# Solution

---

- Miss penalty to main =  $200 \text{ ns} / (2 \text{ ns} / \text{clock cycle}) = 100 \text{ clock cycles}$
- Effective CPI with one level of cache  
= Base CPI + Memory-stall cycles per instruction  
=  $1.0 + 5\% \times 100 = 6.0$
- With two levels of cache, miss penalty to second-level cache  
=  $20 \text{ ns} / (2 \text{ ns} / \text{clock cycle}) = 10 \text{ clock cycles}$
- Effective CPI with two levels of cache  
= Base CPI + Primary stalls per instruction  
+ Secondary stall per instruction  
=  $1 + 5\% \times 10 + 2\% \times 100 = 3.5$
- Therefore, machine with secondary cache is faster by a factor of  
 $6.0 / 3.5 = 1.71$

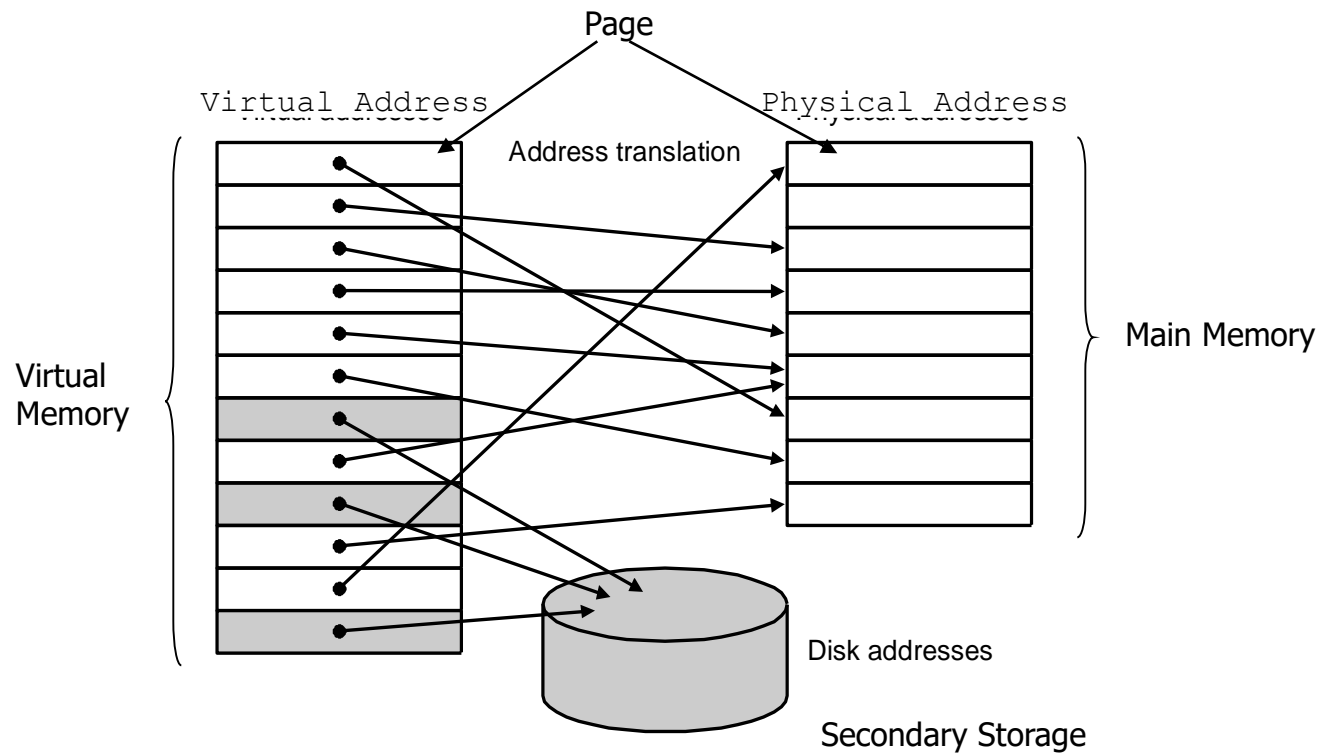


# Virtual Memory

- Motivation: main memory acts as *cache for secondary storage*, e.g., magnetic disk
- *Virtual address space*, i.e., space addressable by a program is determined by ISA
  - e.g., 64-bit MIPS address space size is  $2^{64}$  – recall  $j_r$  instruction
  - typically: main memory size  $\leq$  disk size  $\leq$  virtual address space size
- Program can “pretend” it has main memory of the size of the disk – which is *smaller than* the *virtual memory* (= whole virtual address space), but *bigger than* the actual *physical memory* (=DRAM main memory)
  - *Page table* (as we shall see) transparently converts a virtual memory address to a physical memory address, *if the data is already in main; if not*, it issues call to OS to fetch the data from disk into main
- Virtual memory is organized in fixed-size (power of 2, typically at least 4 KB) blocks, called *pages*. Physical memory is also considered a collection of pages of the same size.
  - the unit of data transfer between disk and physical memory is a page

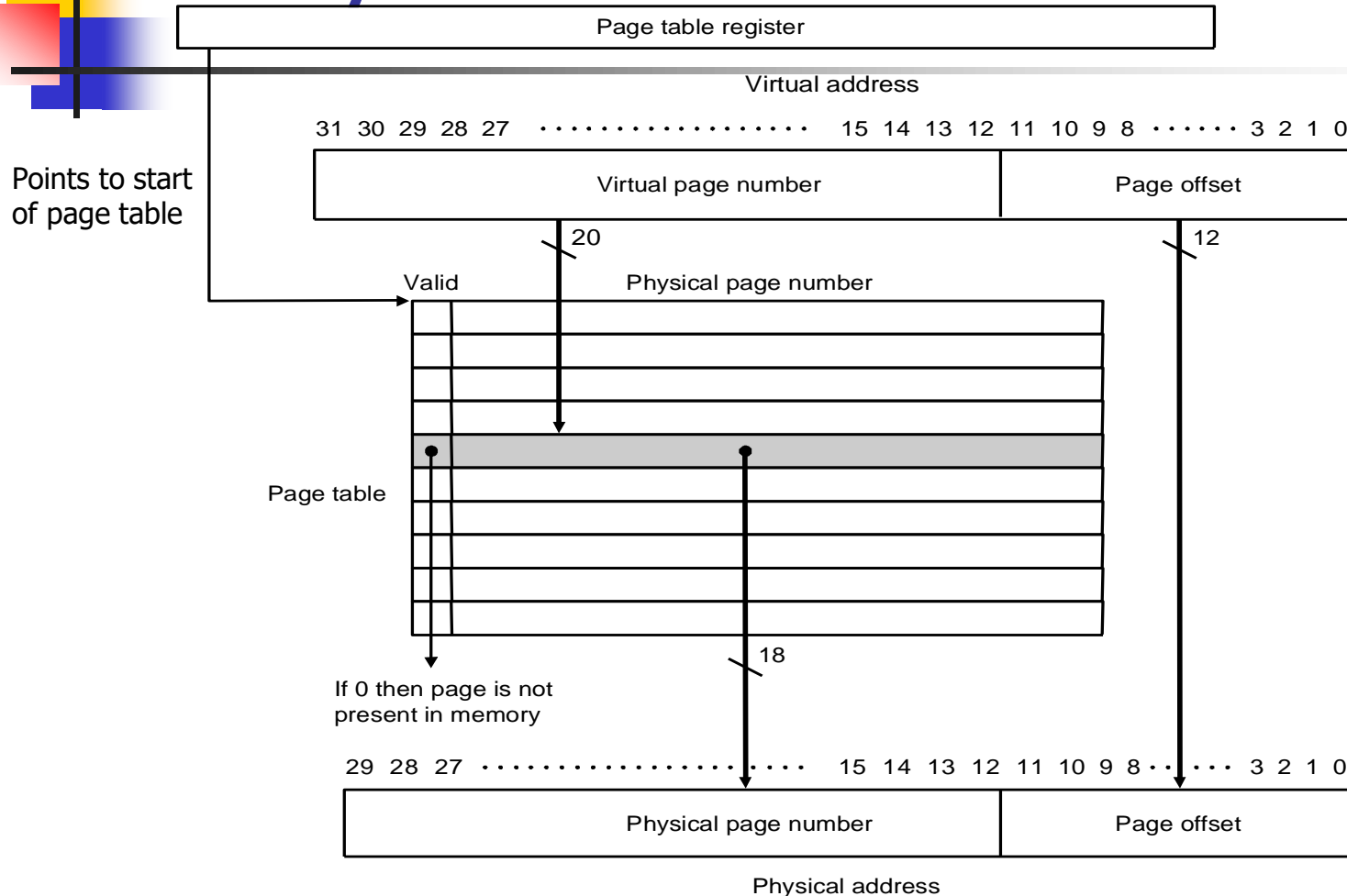


# Virtual Memory



**Mapping of pages from a virtual address to a physical address or disk address**

# Page Table Implements Virtual to Physical Address Translation





# Example Problem

---

- Assume:
  - 32-bit virtual address
  - 4 KB page size
  - 4 bytes per page table entry
- *What is the total page table size if we want to be able to access all of the virtual memory?*



# Solution

---

- No. of page table entries = address space size / page size  
$$= 2^{32} / 2^{12} = 2^{20}$$
- Size of page table = No. of entries  $\times$  entry size  
$$= 2^{20} \times 4 \text{ bytes} = 4 \text{ MB (huge!)}$$
- Note, to avoid large page table size:
  - each program has its own page table
    - *page table register* points to start of program's page table
  - to reduce storage required per program page table
    - page table for a program covers the span of virtual memory containing its own code and data
    - other techniques, e.g., multiple-level page tables, hashing virtual address, etc.



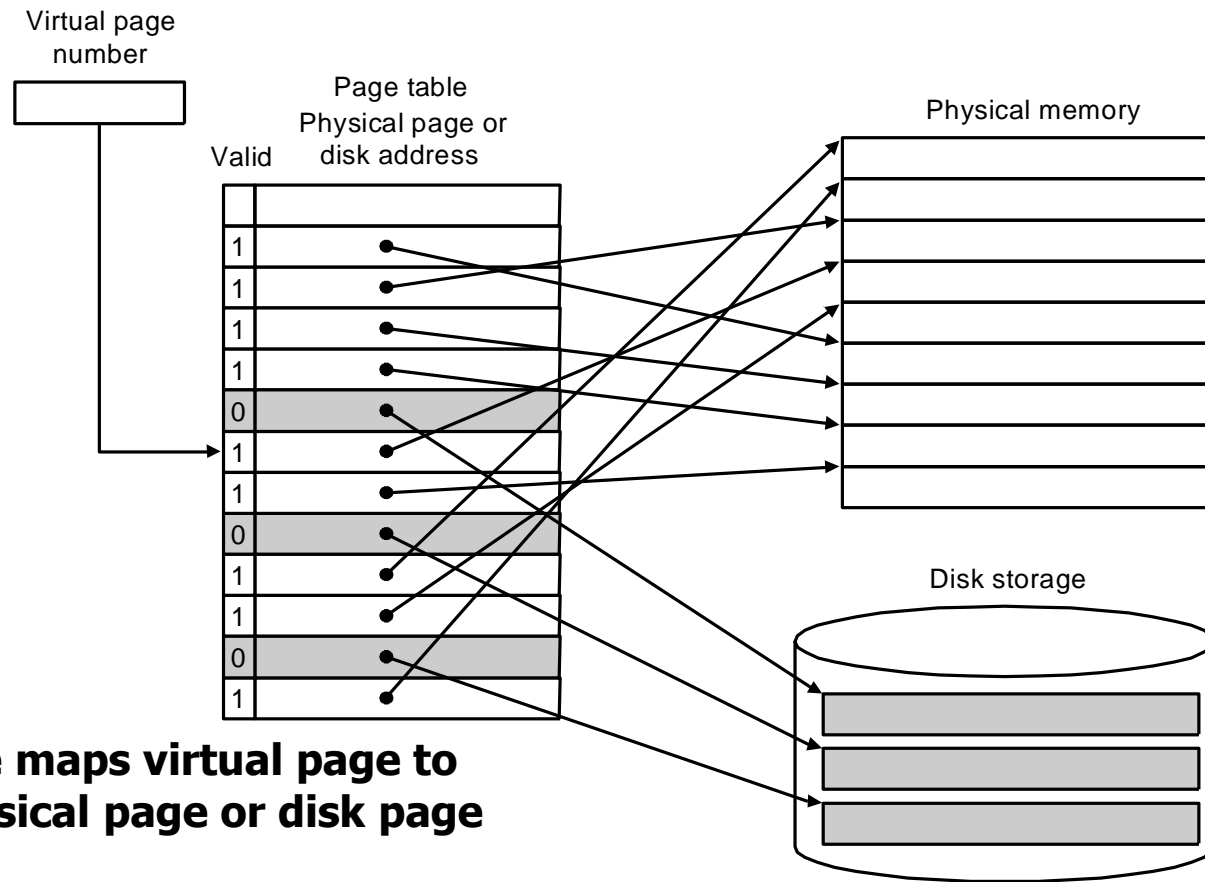
# Page Faults

---

- *Page fault*: page is not in memory, must retrieve it from disk
  - *enormous miss penalty* = millions of cycles
  - therefore, page size should be *large* (e.g., 32 or 64 KB)
    - to make one trip to disk worth a lot
  - reducing page faults is *critical*
    - *LRU replacement* policy – implemented approximately by setting a *use bit* each time a page is accessed, and then periodically clearing all these bits so that pages accessed in a fixed time period are known
    - *fully associative* page placement – consequence of page table
  - handle faults in software instead of hardware
    - as software overhead is still small compared to disk access time
  - using write-through is too expensive, so always use write-back

# Resolving Page Faults using the Page Table to Access Disk

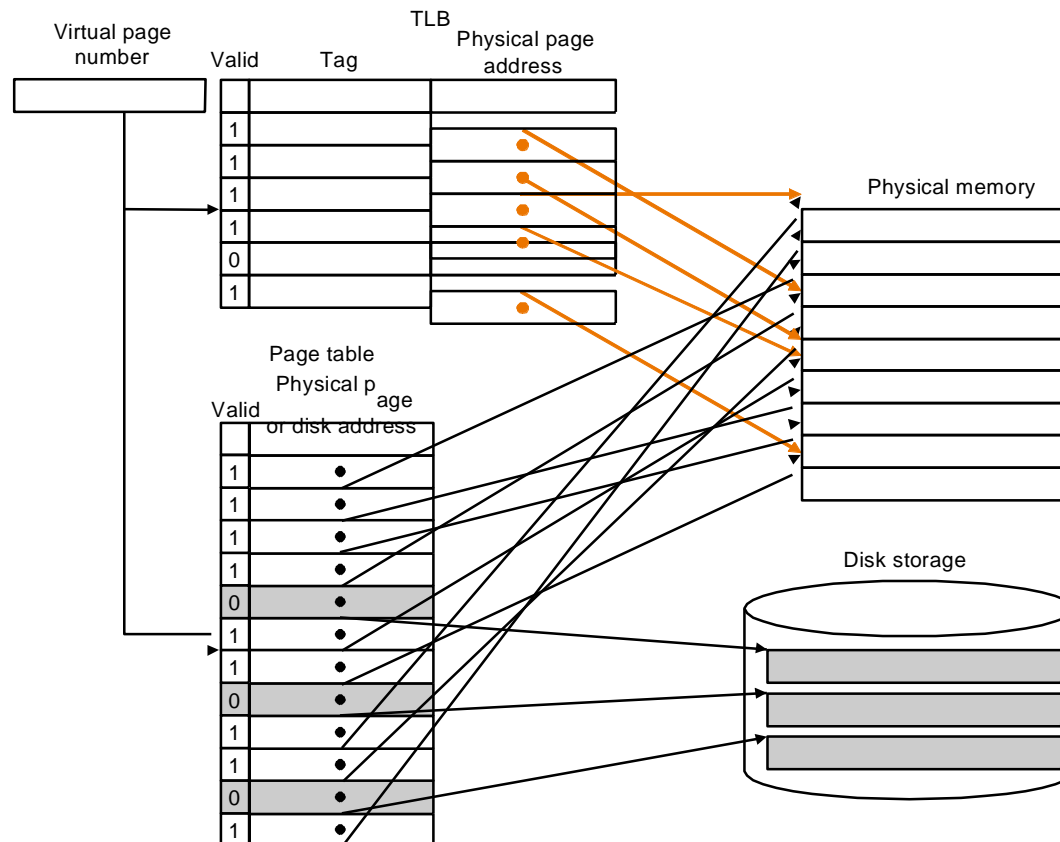
- There is a data structure, either part of or auxiliary to the page table, which records *where each virtual page is stored on disk* (cylinder, sector, block, etc.)



**Page table maps virtual page to either physical page or disk page**

# Making Address Translation Fast with the Translation-lookaside Buffer

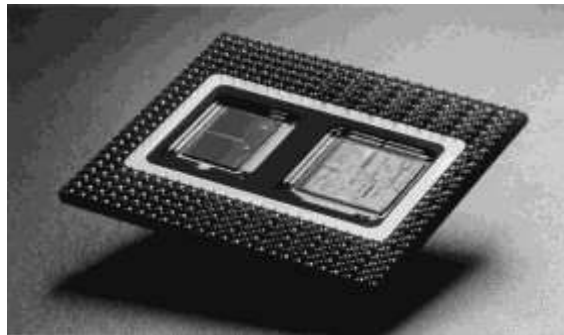
- A *cache* for address translations – *translation-lookaside buffer (TLB)*:



**On a page reference, first look up the virtual page number in the TLB; if there is a TLB miss look up the page table; if miss again then true page fault**

# Modern Systems

Characteristic	Intel Pentium Pro	PowerPC 604
Virtual address	32 bits	52 bits
Physical address	32 bits	32 bits
Page size	4 KB, 4 MB	4 KB, selectable, and 256 MB
TLB organization	A TLB for instructions and a TLB for data Both four-way set associative Pseudo-LRU replacement Instruction TLB: 32 entries Data TLB: 64 entries TLB misses handled in hardware	A TLB for instructions and a TLB for data Both two-way set associative LRU replacement Instruction TLB: 128 entries Data TLB: 128 entries TLB misses handled in hardware



Characteristic	Intel Pentium Pro	PowerPC 604
Cache organization	Split instruction and data caches	Split instruction and data caches
Cache size	8 KB each for instructions/data	16 KB each for instructions/data
Cache associativity	Four-way set associative	Four-way set associative
Replacement	Approximated LRU replacement	LRU replacement
Block size	32 bytes	32 bytes
Write policy	Write-back	Write-back or write-through





# Some Issues

---

- Processor speeds continue to increase very fast
  - much faster than either DRAM or disk access times
- Design challenge: dealing with this growing disparity
- Trends:
  - synchronous SRAMs (provide a burst of data)
  - redesign DRAM chips to provide higher bandwidth or processing
  - restructure code to increase locality
  - use pre-fetching (make cache visible to ISA)