Computer Architecture & Assembly Language 14:332:331

Lecture 8

Memory Hierarchy

Naghmeh Karimi

Fall 16

Adapted from *Computer Organization and Design*, *5th Edition*, Patterson & Hennessy, © 2013, Elsevier, *and Computer Organization and Design*, *4th Edition*, Patterson & Hennessy, © 2008, Elsevier and Mary Jane Irwin's slides from Penn State University.

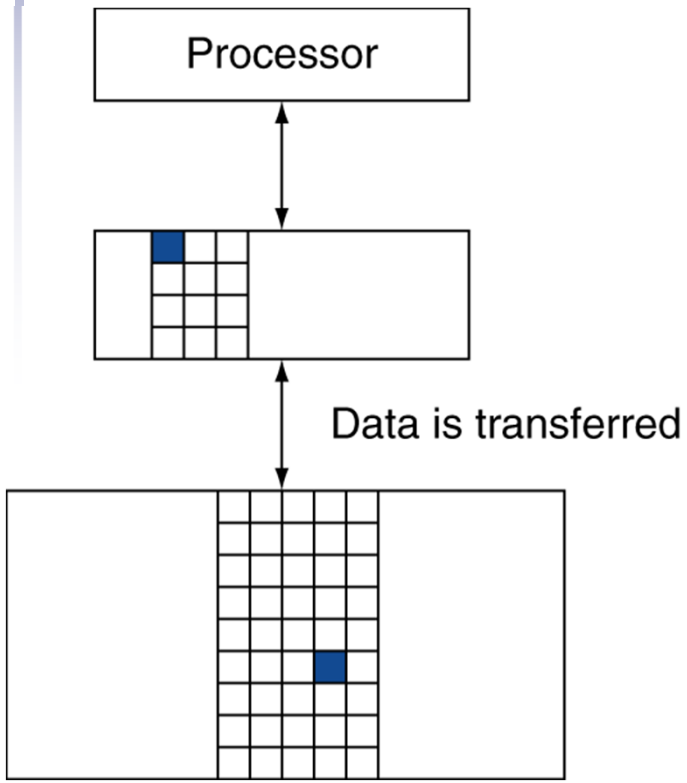
Principle of Locality

- 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

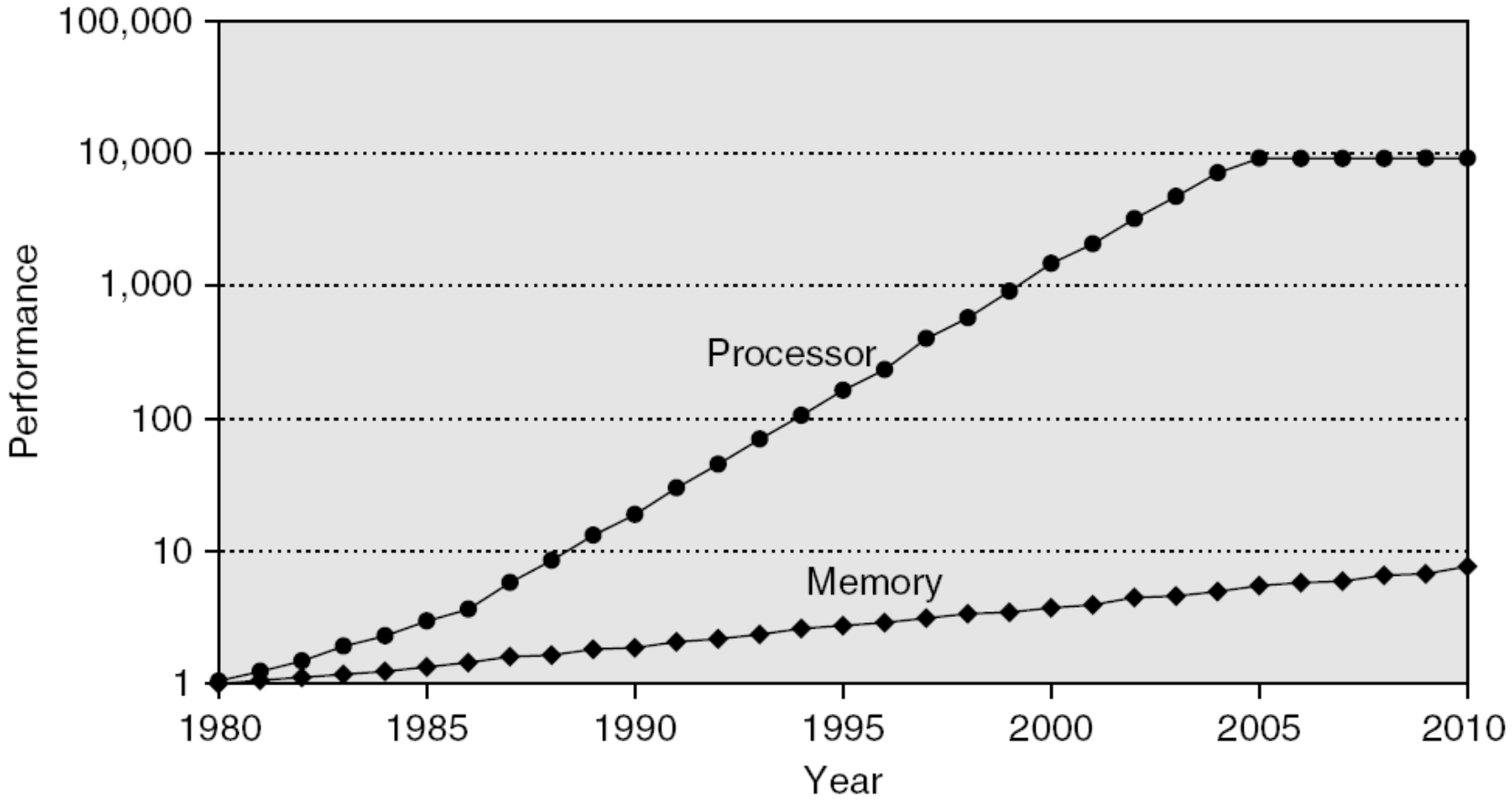


- Block (aka line): unit of copying
  - May be multiple words
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses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

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 Performance Gap

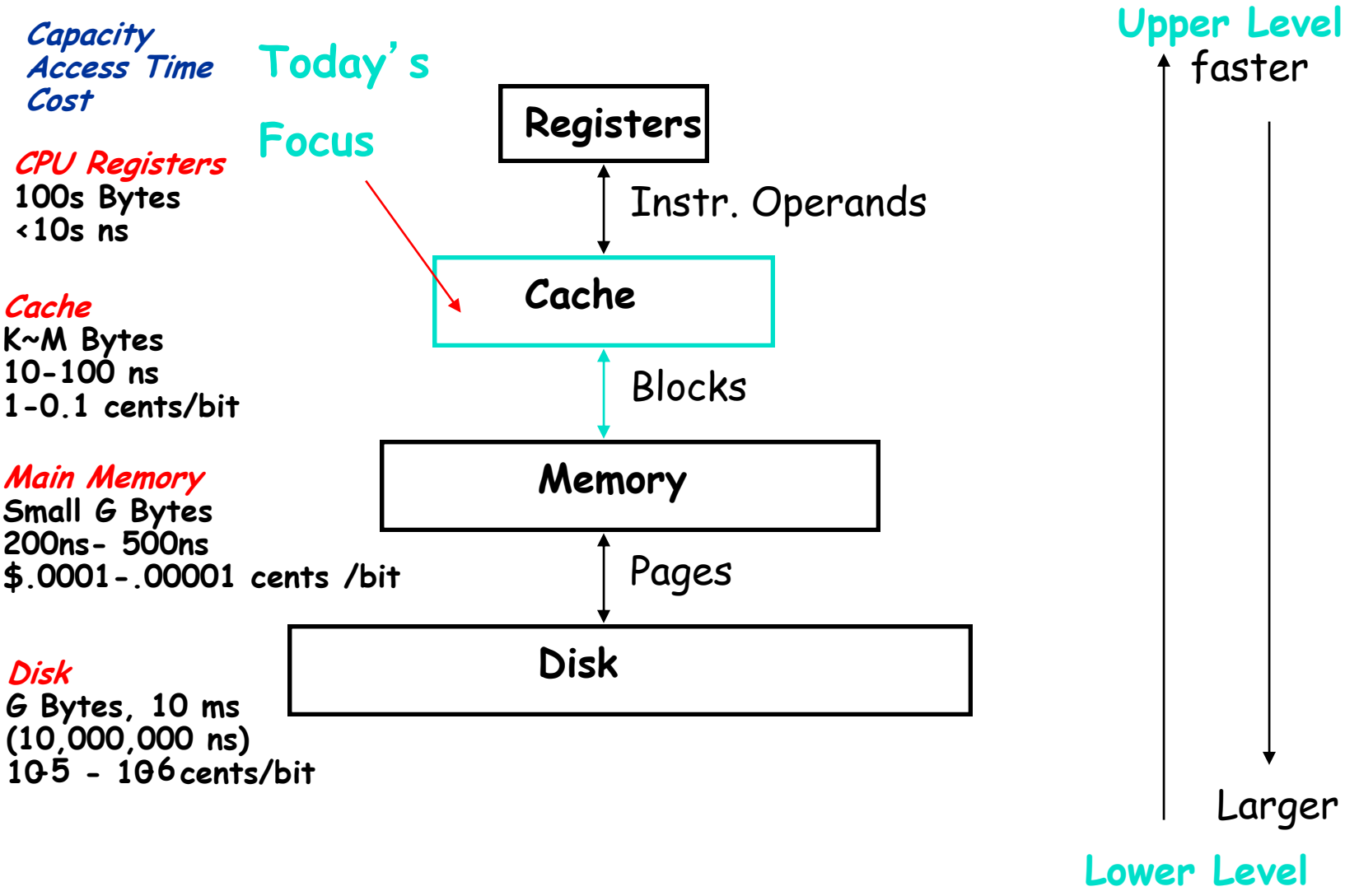


Processor-Memory Performance GAP

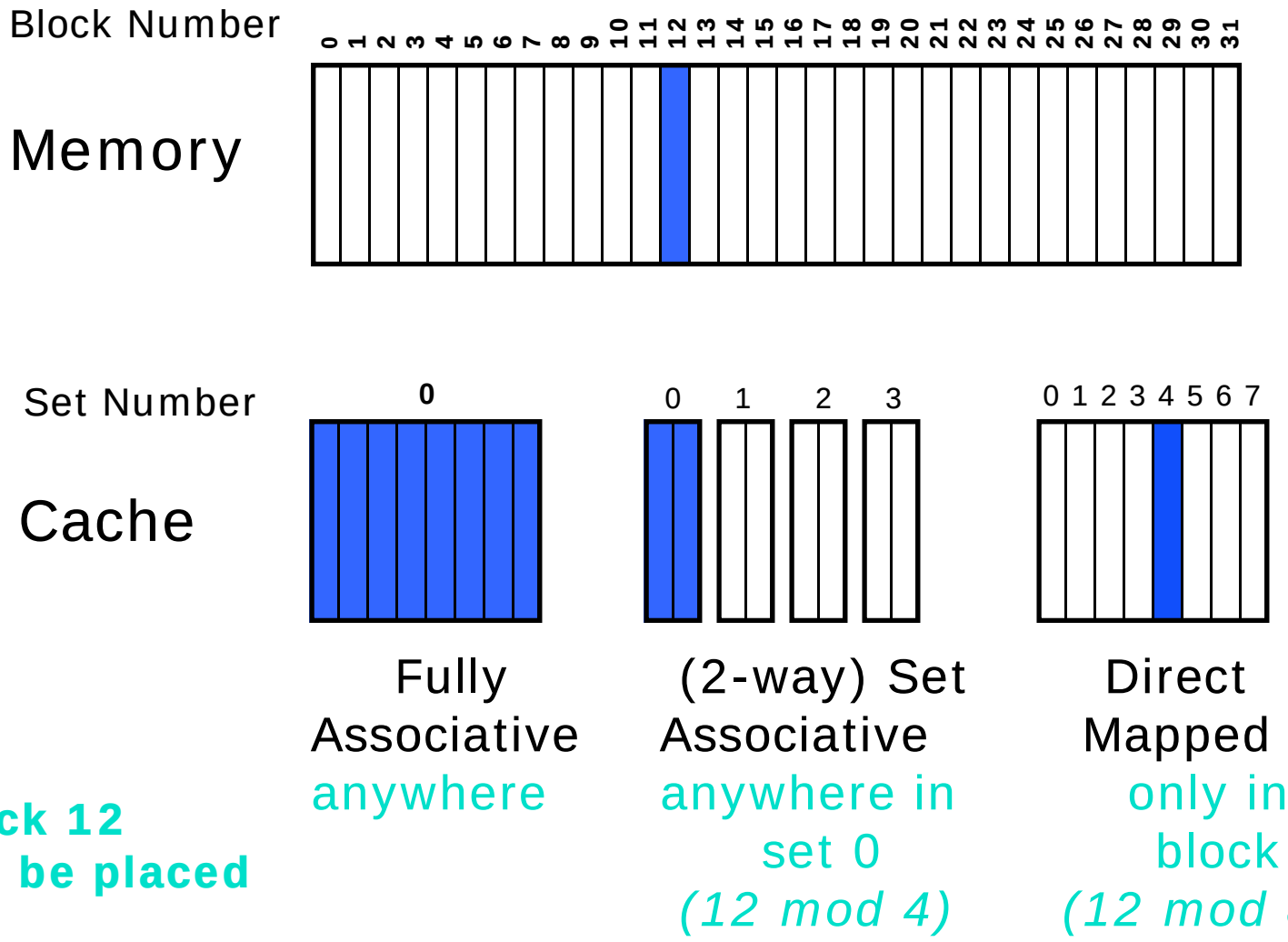
- Goal: Illusion of large, fast, cheap memory. Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access

- Solution: Put smaller, faster “cache” memories between CPU and DRAM. Create a “memory hierarchy”.

Levels of the Memory Hierarchy

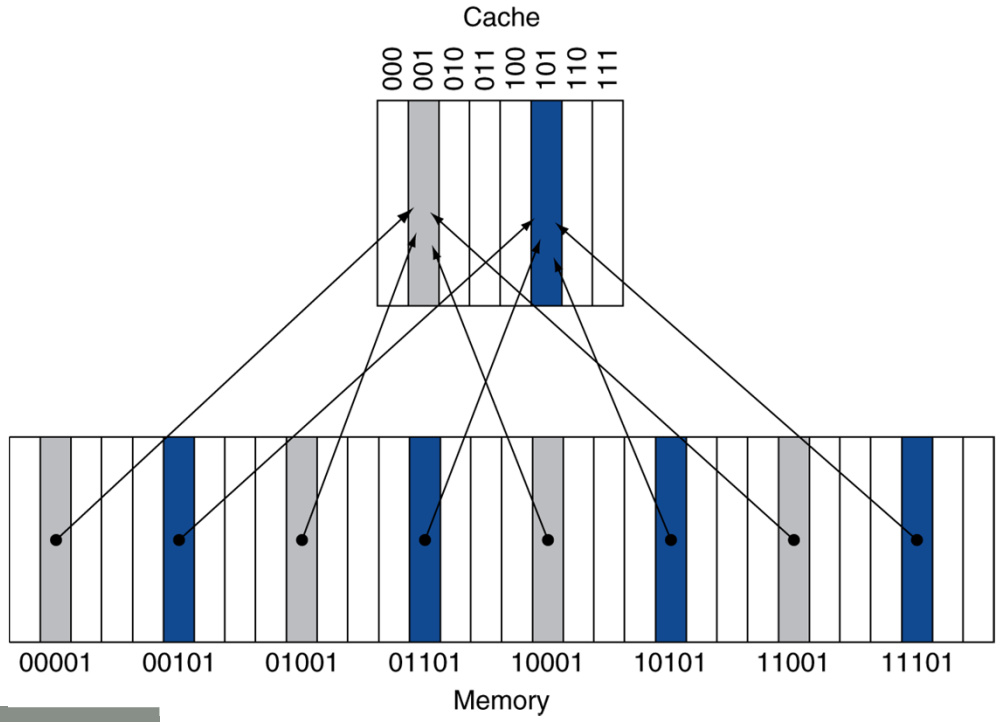


Q1: Where can a block be placed in Cache?



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

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
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

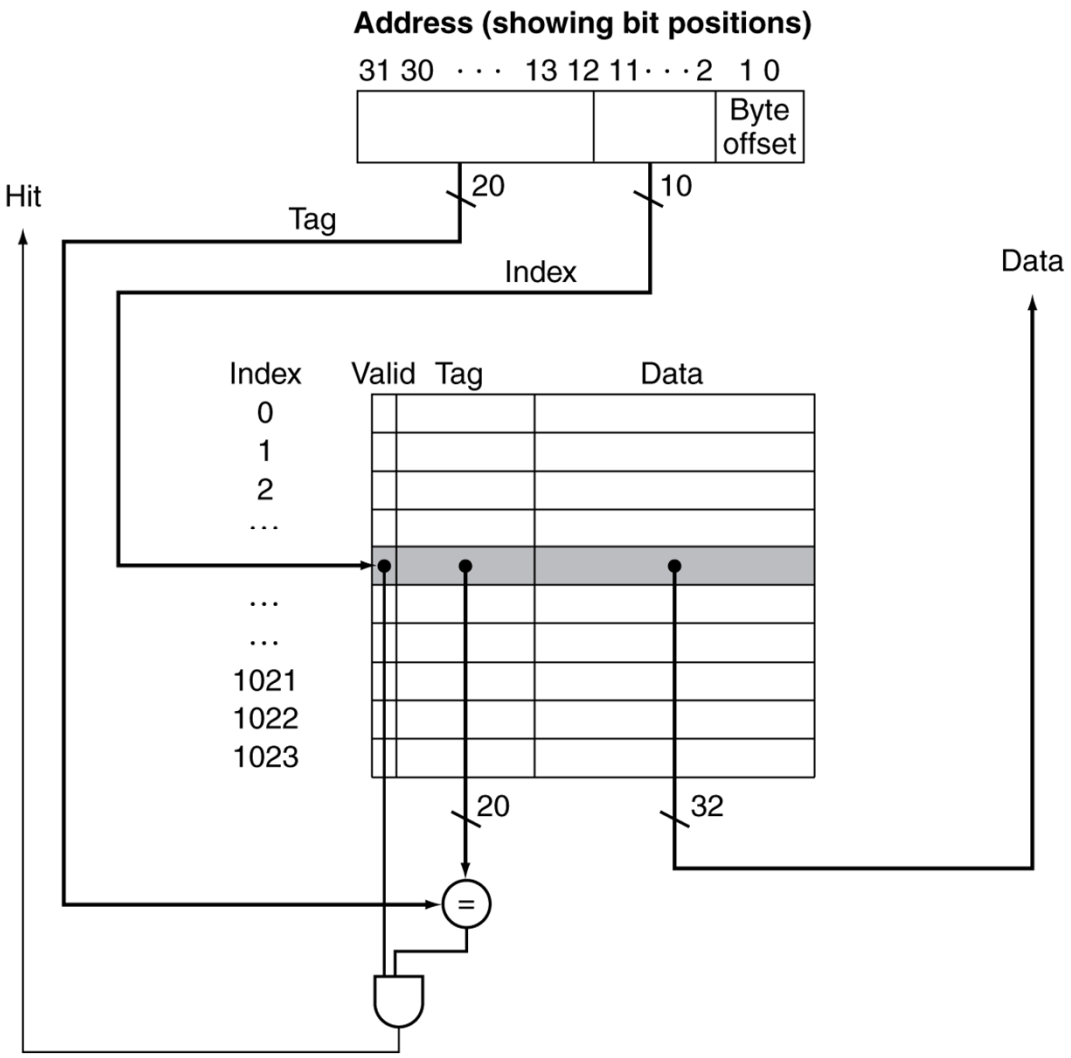
Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

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		

Hit_rate= 3/8= 37.5%

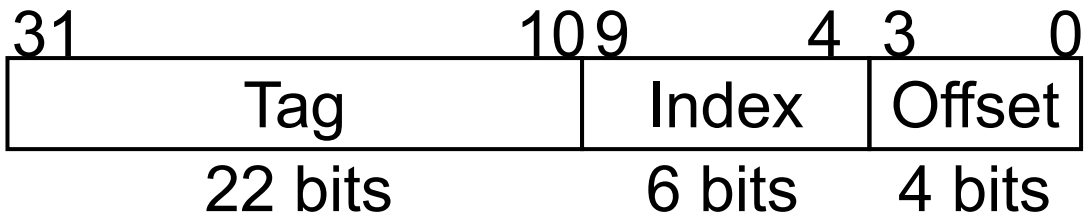
Miss_rate=5/8=62.5%

Address Subdivision



Example: Larger Block Size

- 64 blocks, 16 bytes/block
  - To what block number does address 1200 map?
- Block address = $\lfloor 1200/16 \rfloor = 75$
- Block number = 75 modulo 64 = 11



0000000000000000000001 001011 0000

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 miss penalty
  - Can override benefit of reduced miss rate

Cache size (direct mapped)

- How large needs the cache be?

- Each block in the cache is formed of the valid bit + tag bits + data bits

Example: What is the total size of a cache that can store 64 kB of data. Please assume that, the cache is direct mapped. Each block is one word (=32 bits). Main memory is addressed with 32 bits.

Cache size (direct mapped)

- Each block in the cache is formed of the valid bit + tag bits + data bits

Example: What is the total size of a cache that can store 64 kB of data. Please assume that, the cache is direct mapped. Each block is one word (=32 bits). Main memory is addressed with 32 bits.

Answer:

Number of blocks in cache= 64KB/(32 bit=4Byte)=16K=$2^4*2^{10}=2^{14}$

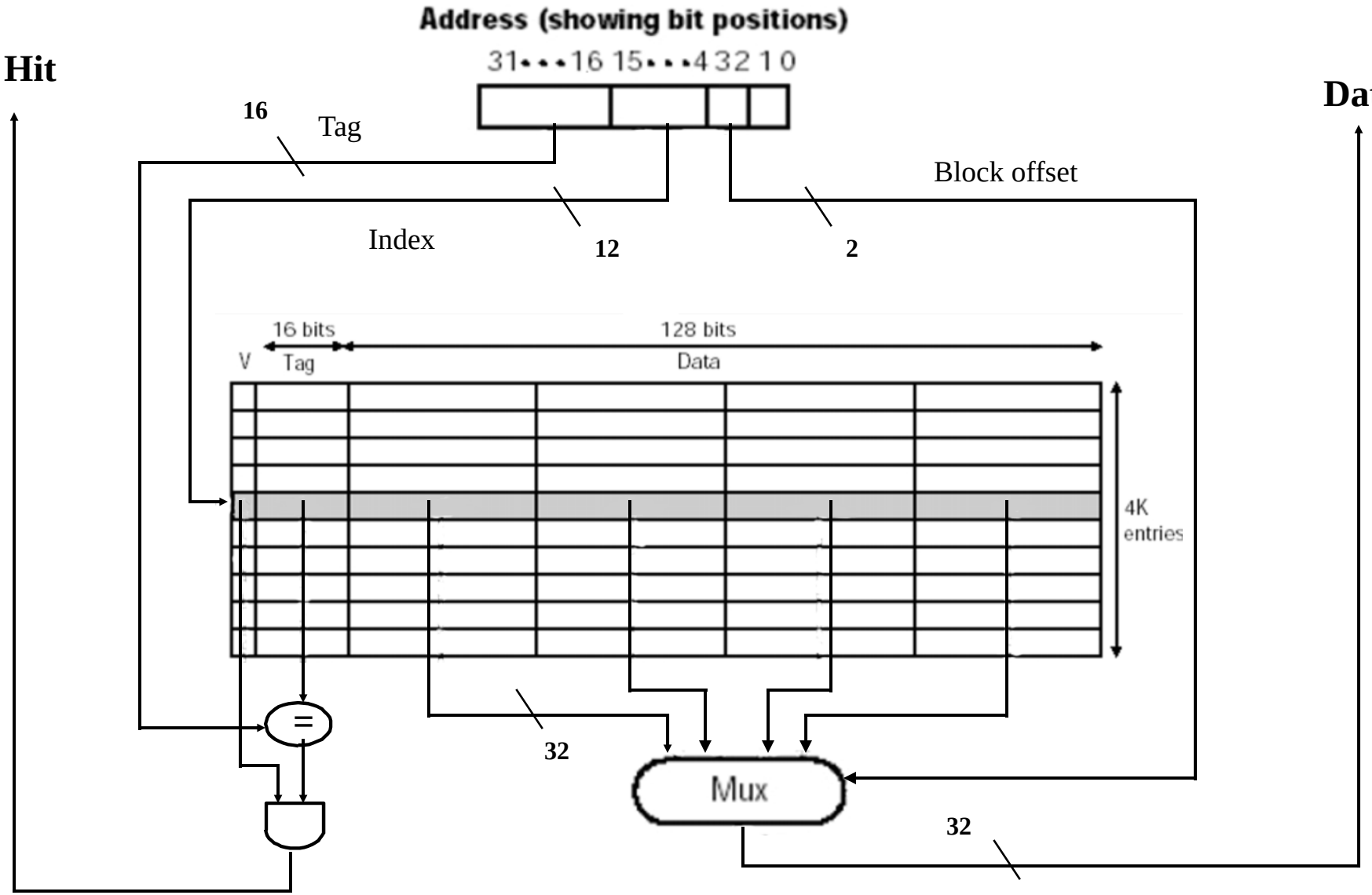
→ Index is $\log_2 2^{14}$= 14 bits

→ Each block is 1 word=32 bits=4 byte → offset=$\log_2 4$=2

- Thus each block needs → 1(for valid bit)+ (32-14-2)+32=49
- Thus total size= $2^{14} \times (49)$= 802,816 bits = 100 kB

- Note1: 2 relates to the number of bits we need to address each byte in a block. Here each word is 4 bytes. So 2 bits needed.
- Note 2: Number of tag bits: 32-14-2=16
- Note 3: Total size of main memory =$2^{32}=2^2*2^{10}*2^{10}*2^{10}$=2GB

32-bit cache with 16-byte blocks



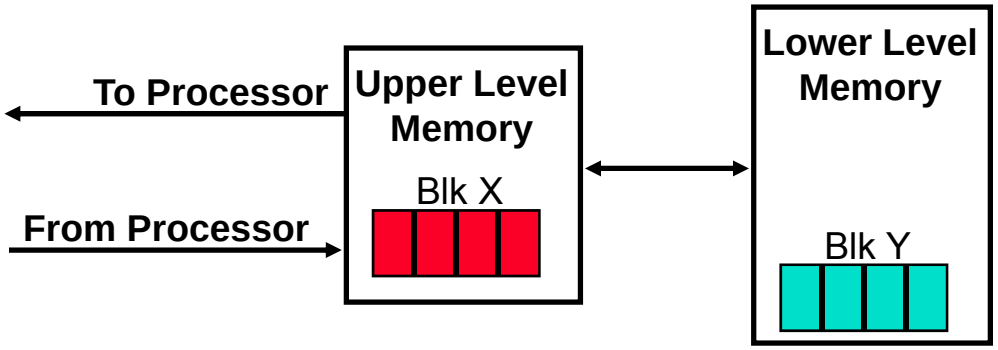
Cache size for a 4-word block cache

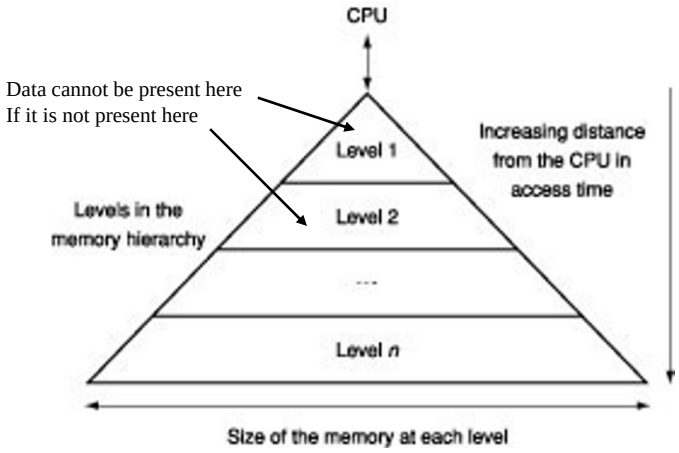
- How many total bits are required for a cache size that holds 16 kB of data?
- 1 block= 4 words and 1 word=4 bytes → 16 KB data is equivalent to 1K(=1024) blocks
- So cache includes 1024 blocks or 2^{10} blocks. Thus the cache needs 10 index bits. n=10
- Block size is 4 words= 4*4 bytes= 16 bytes → block offset:4 bit
- The total number of bits is the total number of rows times the size of each row.
- The tag bits are 32 – 10 – 4
- Block size is 4x32+ (32-10-4)+1
- Cache size is 2^{10}x(128+19)= 1024x147 bits

Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy

How do we apply the principle of locality?





- **Hit**: data needed by CPU appears in *some* block in the upper level (Block X)
- **Hit Rate**: the fraction of accesses found in the upper level
- **Hit Time**: Time to access the upper level = RAM access time + Time to determine if the access is a hit/miss
- **Miss penalty**: time for data to be retrieve from a *lower* level memory (needed data is in Block Y) and deliver it to the processor
- Hit Time << Miss Penalty

Measuring Cache Performance

- Components of CPU time
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- With simplifying assumptions:

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

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

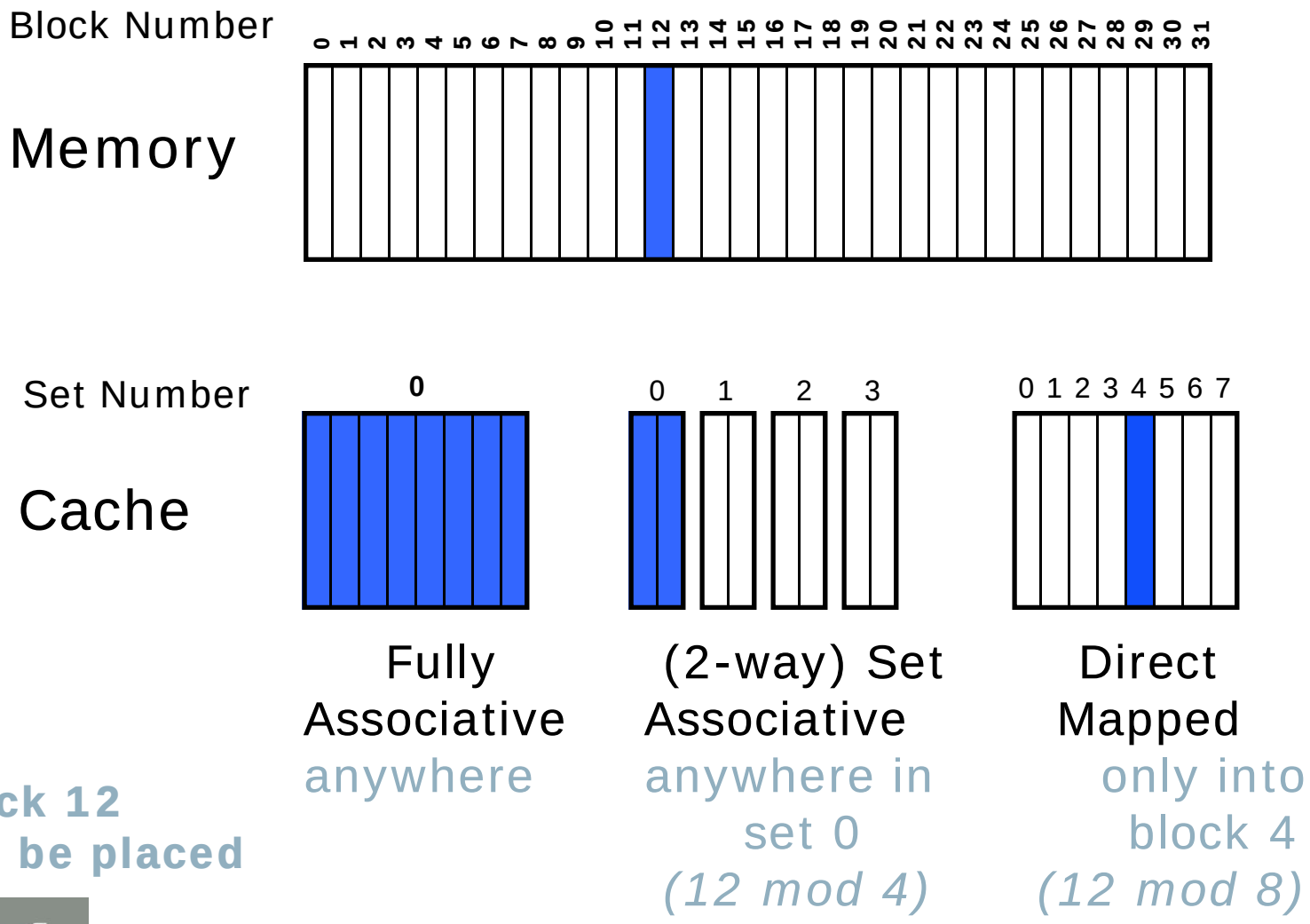
Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Performance Summary

- When CPU performance increased
  - Miss penalty becomes more significant
- Decreasing base CPI
  - Greater proportion of time spent on memory stalls
- Increasing clock rate
  - Memory stalls account for more CPU cycles
- Can’t neglect cache behavior when evaluating system performance

Q1: Where can a block be placed in Cache?



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)

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]		

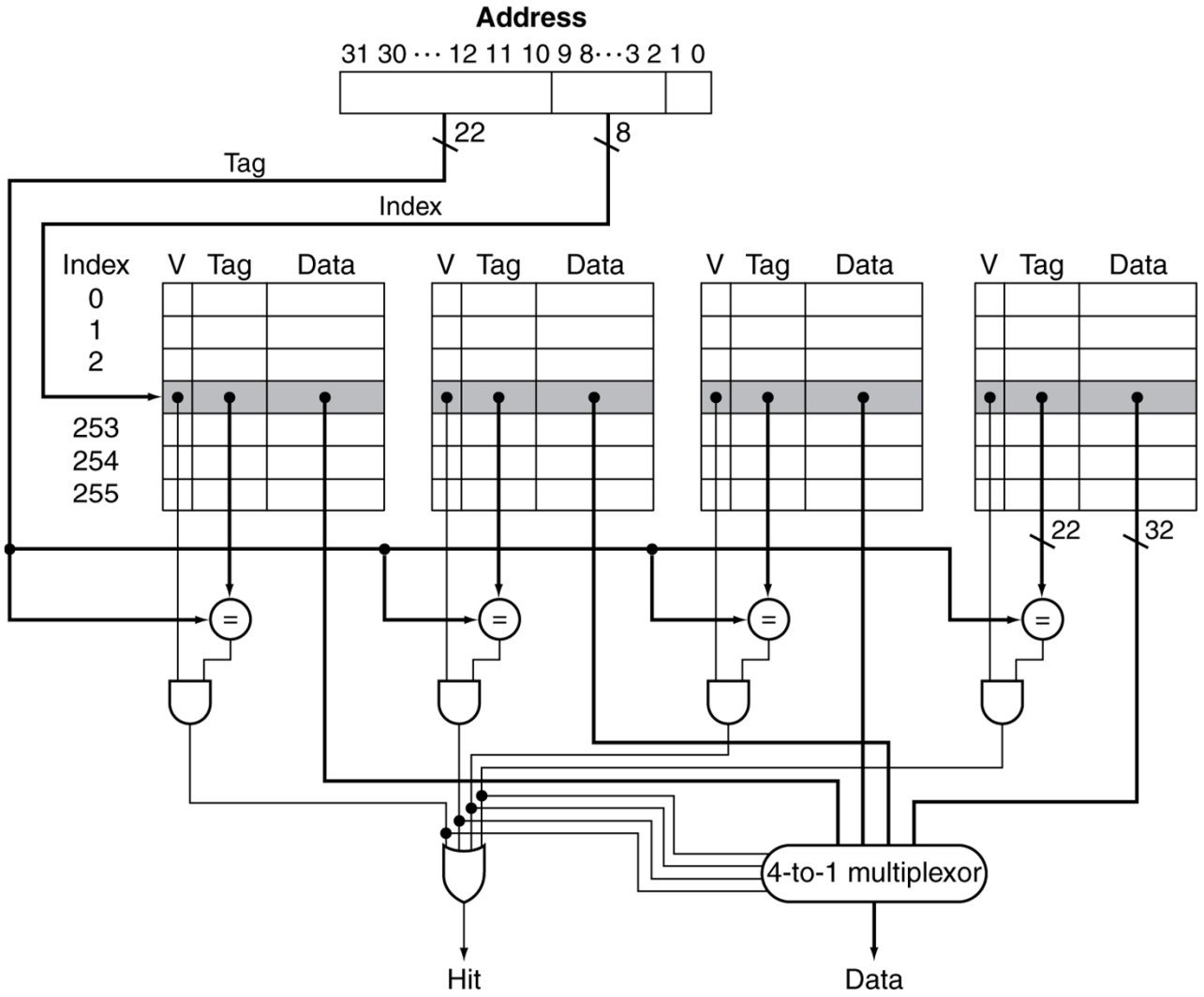
- Fully associative

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

Multilevel Caches

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Sources of (Cache) Misses

- Compulsory: first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant
- Conflict (collision): Multiple memory locations (blocks) mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Capacity: Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size
- Invalidation: other process (e.g., I/O) updates memory

Questions for Memory Hierarchy

- **Q1: Where can a block be placed in the cache?** ***(Block placement)***
- **Q2: How is a block found if it is in the cache?** ***(Block identification)***
- **Q3: Which block should be replaced on a miss?** ***(Block replacement)***
- **Q4: What happens on a write?**

  (Write strategy)

Q4: What happens on a write?

- **Cache hit:**
  - ***write through:*** **write both cache & memory**
    - **generally higher traffic but simplifies cache coherence**
  - ***write back:*** **write cache only (memory is written only when the entry is evicted)**
    - **a dirty bit per block can further reduce the traffic**
- **Cache miss:**
  - ***no write allocate:*** **only write to main memory**
  - ***write allocate*** ***(aka fetch on write):*** **fetch into cache**
- **Common combinations:**
  - **write through and no write allocate**
  - **write back with write allocate**

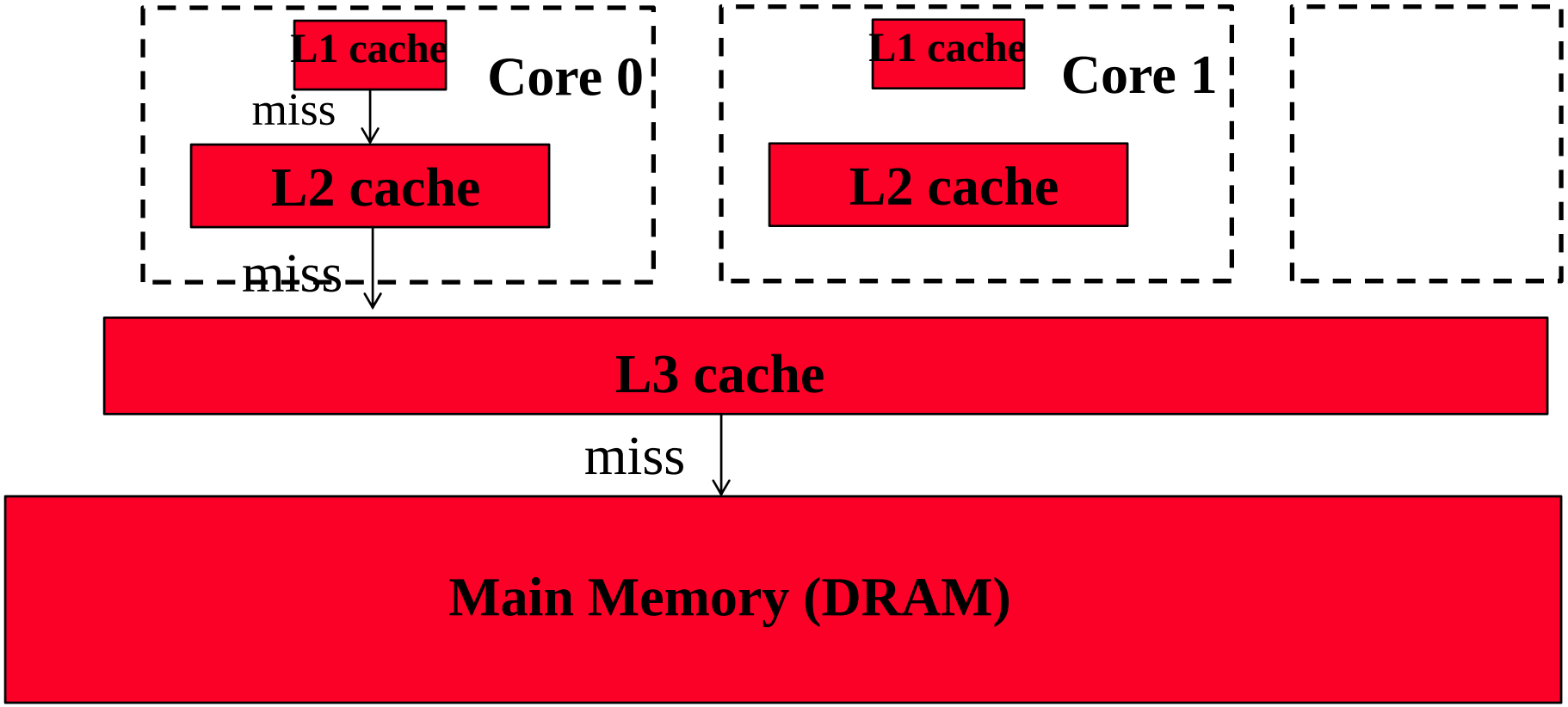
Write-Through

- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1×100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full

Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

Further Improvements – three-level cache



$$AMAT = L1_{hit\ time} + L1_{miss\ rate} \times (L2_{hit\ time} + L2_{miss\ rate} \times (L3_{hit\ time} + L3_{miss\ rate} \times DRAM_{access\ time}))$$

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Example (cont.)

- Now add L-2 cache
  - Access time = 5ns→ 5/0.25=20 clock cycle
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 400 cycles
- CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- Performance ratio = 9/3.4 = 2.6

Virtual Memory

- When multiple applications (processes) run at the same time, the main memory (DRAM) becomes *too small*
- `Virtual memory` extends the memory hierarchy to the hard disk, and treats the RAM as "cache" for the hard disk.
- This way each process is allocated a portion of the RAM, and each program has its own `range` of physical memory addresses
- Virtual memory "translates" (maps) the virtual addresses of each program to physical addresses in main memory
- `Protections` have to be in place in case of data sharing.

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 - continued

- When there is a page fault - *millions* of clock cycles as penalty - it is treated in software through the exception mechanism.

- Software can reduce page faults by cleverly deciding which pages to replace in DRAM (older pages)
- `Pages always exist on the hard disk`, but are loaded into DRAM only when needed by the program.

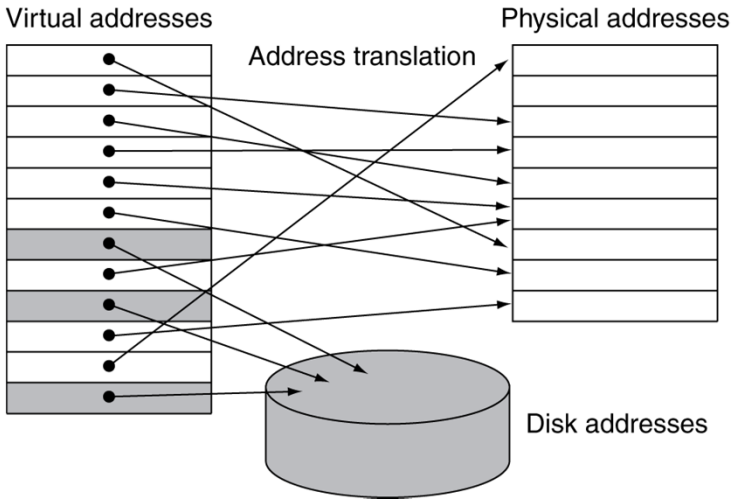
- A `write-back` mechanism insures that pages that were altered (written into in RAM) are saved on disk before being discarded. Write-through is not practical – too long

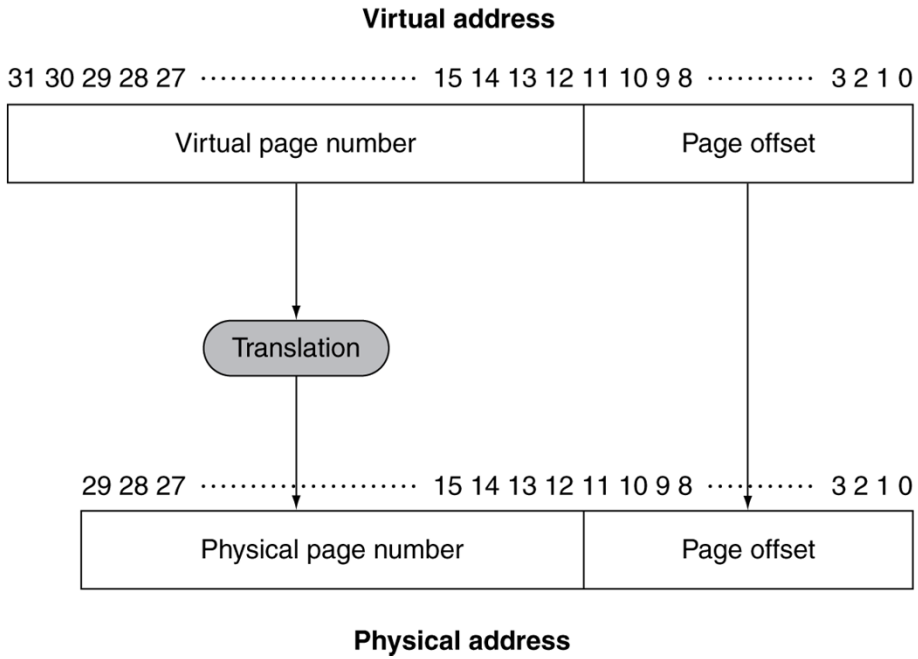
Virtual Memory - continued

- The translation mechanism is provided by a `page table`
- Each program has its *own page table* which contains the physical addresses of the pages and is indexed by the virtual page number.
- Each program when it has possession of the CPU has its pointer to the page table loaded ahead of time by the OS and its page table is read.
- Since each process has its own page table, `programs can have same virtual address space` because the page table will have different mappings for different programs (protection)

Address Translation

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





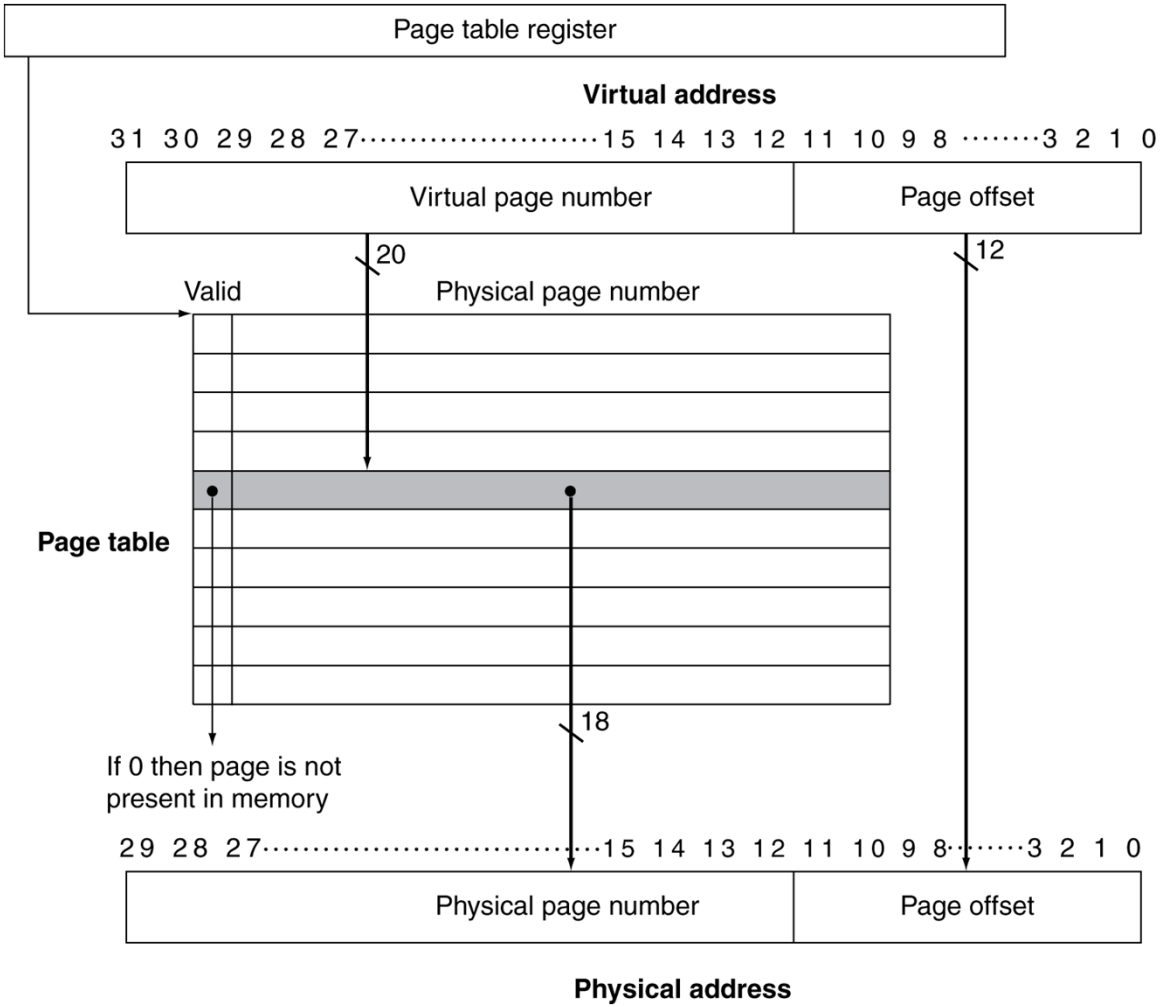
Page Fault Penalty

- 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

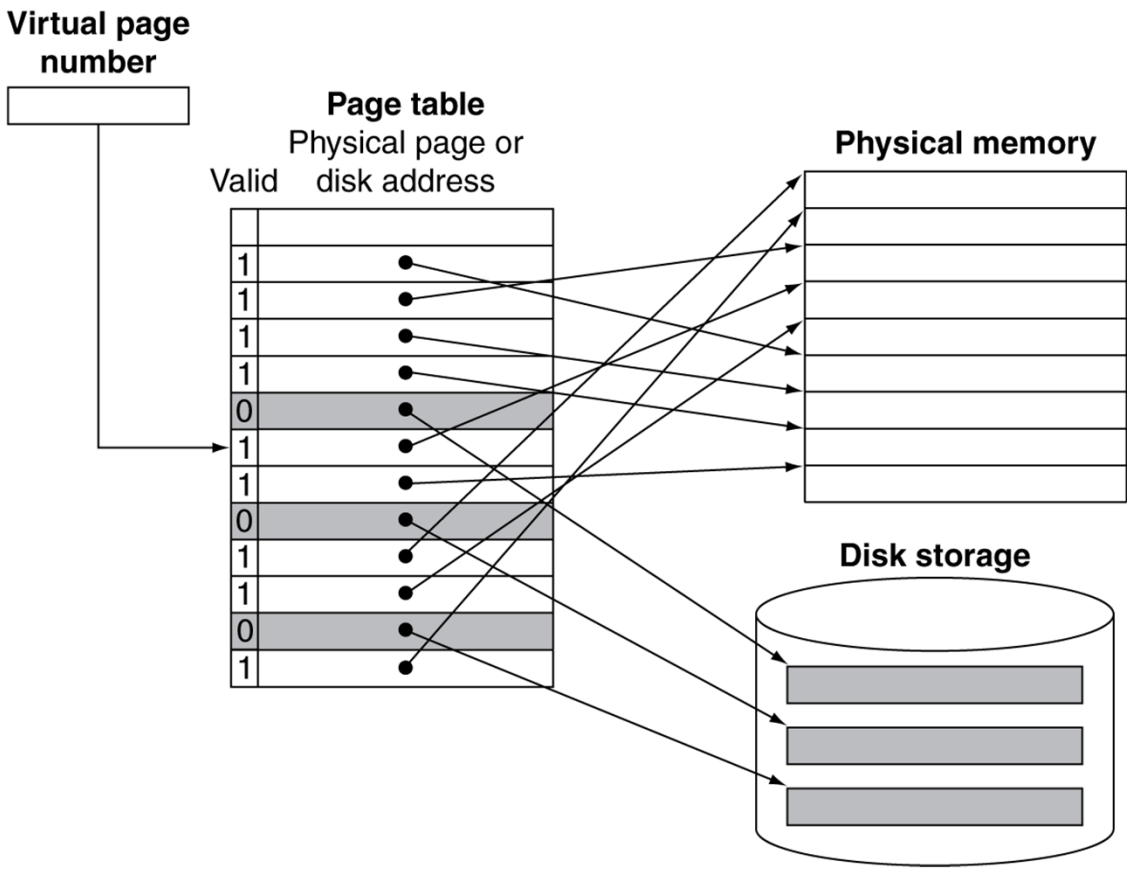
Page Tables

- Stores placement information
  - Array of page table entries (PTE), 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



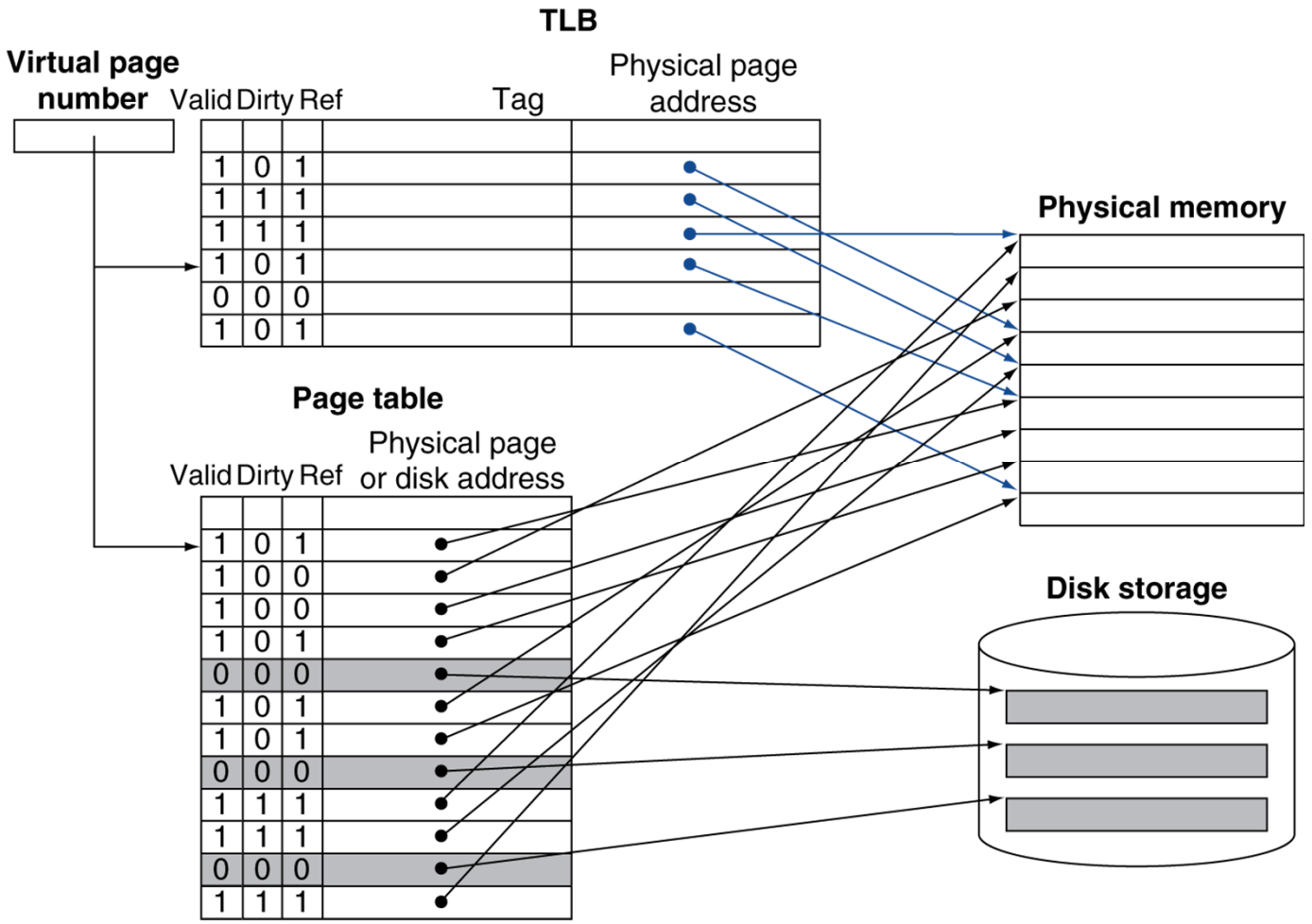
Replacement and Writes

- 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

Example:

Consider a virtual memory system with 40-bit virtual byte address, 16 KB page and 36-bit physical byte address.

- What is the total size of the page table for each process on this machine, assuming that the valid, protection, dirty and use bits take a total of 4 bits and that all the virtual pages are in use?

Assume that disk addresses are not stored on the page table.

- Page table size = #entries $\times$ entry size

The #entries = # pages in virtual address =

$$\frac{2^{40} \text{ bytes}}{16 \times 10^3 \text{ bytes/page}}$$

$$= \frac{2^{40}}{2^4 \times 2^{10}} = 2^{26} \text{ entries}$$

The width of each entry is 4+ 36 = 40 bits

Thus the size of the page table is $\frac{2^{26} \times 40}{2^3} = 5 \times 2^{26}$ bytes= 335 MB