CS2006: 計算機組織

Memory Hierarchy

Outline

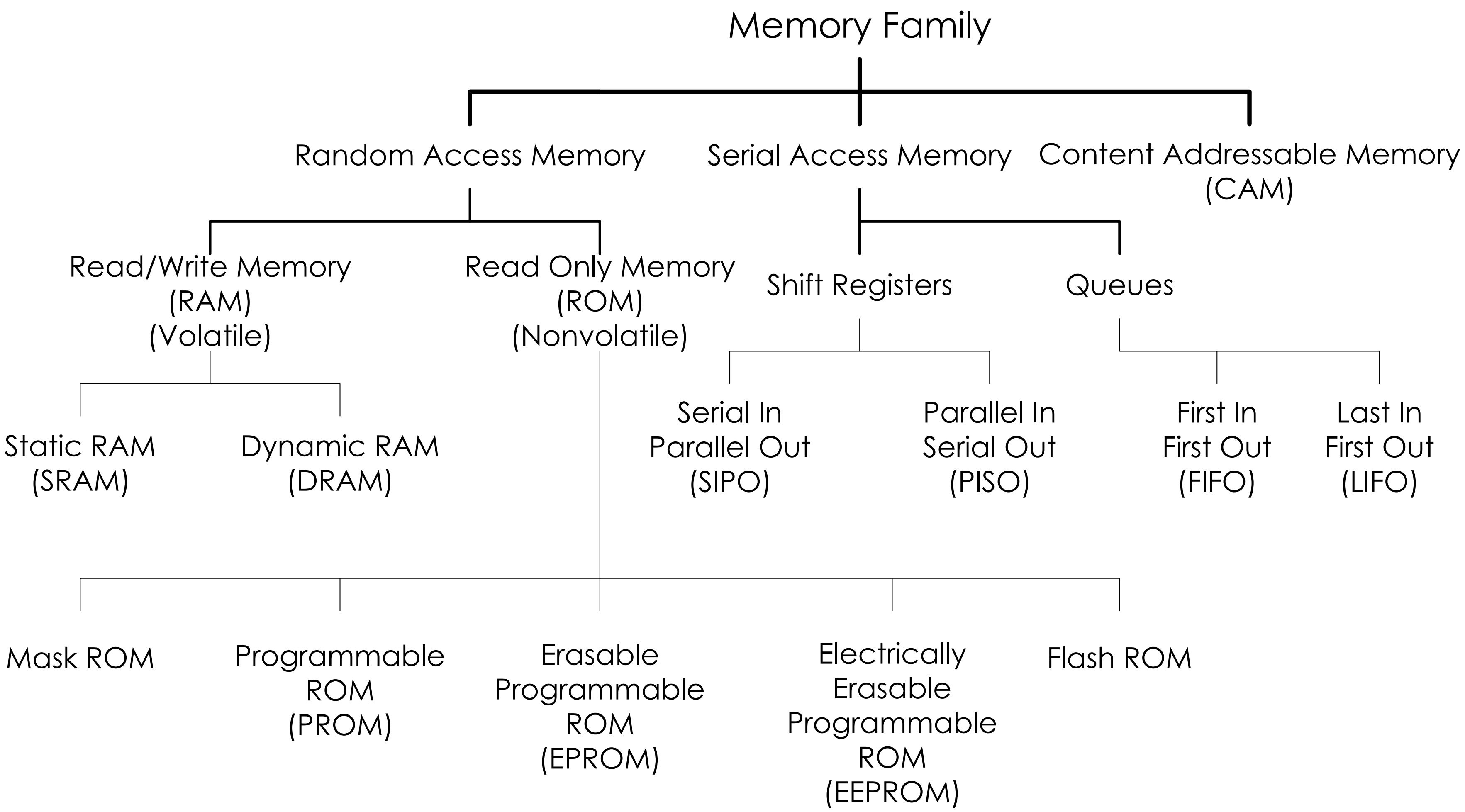
- Memory hierarchy
- The basics of caches
- Measuring and improving cache performance
- Virtual memory
- A common framework for memory hierarchy
- Using a finite state machine to control a simple cache
- Parallelism and memory hierarchies: cache coherence

Memory Technology

- Random access:
  - Access time same for all locations
  - SRAM: *Static Random Access Memory*
    - Low density, high power, expensive, fast
    - Static: content will last (forever until lose power)
    - Address not divided
    - Use for caches
  - DRAM: *Dynamic Random Access Memory*
    - High density, low power, cheap, slow
    - Dynamic: need to be refreshed regularly
    - Addresses in 2 halves (memory as a 2D matrix):

      RAS/CAS (Row/Column Access Strobe)
    - Use for main memory
- Magnetic disk

Memory Family



Memory Family

Random Access Memory

Serial Access Memory

Content Addressable Memory (CAM)

Read/Write Memory (RAM) (Volatile)

Read Only Memory (ROM) (Nonvolatile)

Shift Registers

Queues

Static RAM (SRAM)

Dynamic RAM (DRAM)

Serial In Parallel Out (SIPO)

Parallel In Serial Out (PISO)

First In First Out (FIFO)

Last In First Out (LIFO)

Mask ROM

Programmable ROM (PROM)

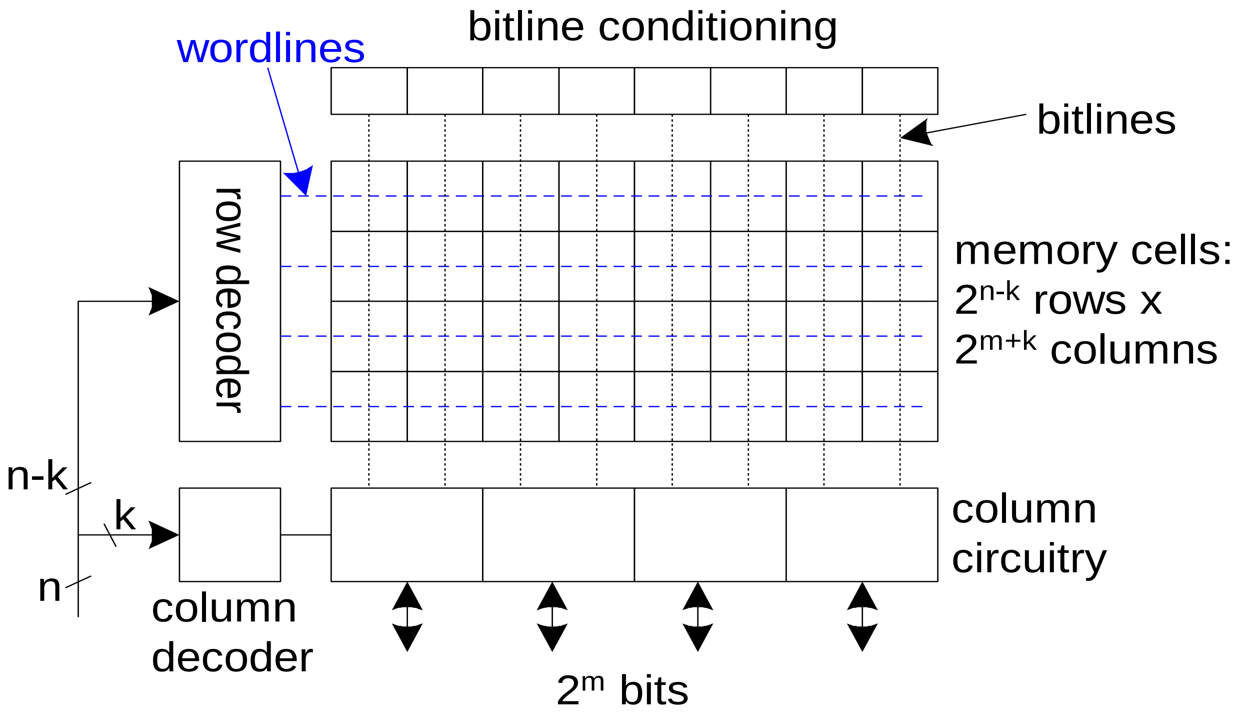
Erasable Programmable ROM (EPROM)

Electrically Erasable Programmable ROM (EEPROM)

Flash ROM

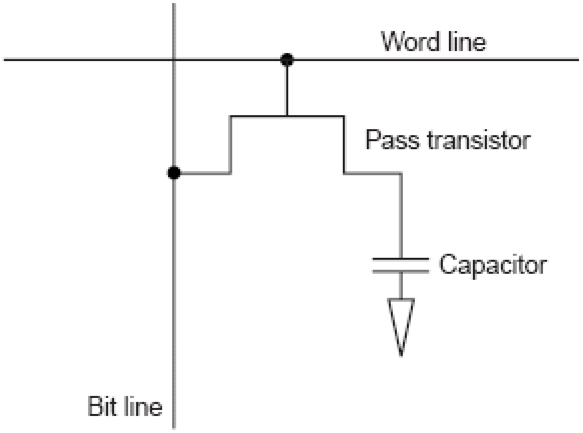
Array Architecture

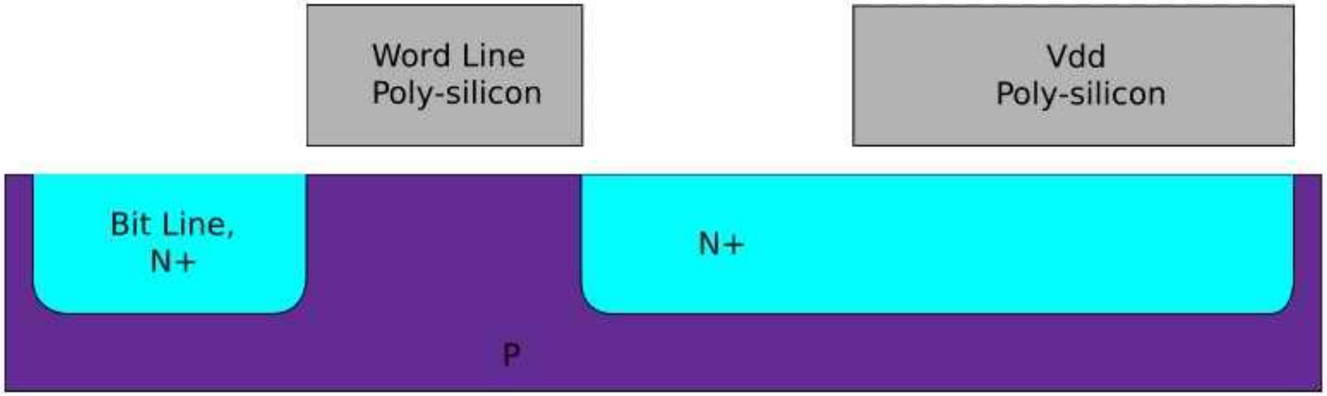
- 2^n *words* of 2^m *bits* each
- If n >> m, fold by 2^k into fewer *rows* of more *columns*

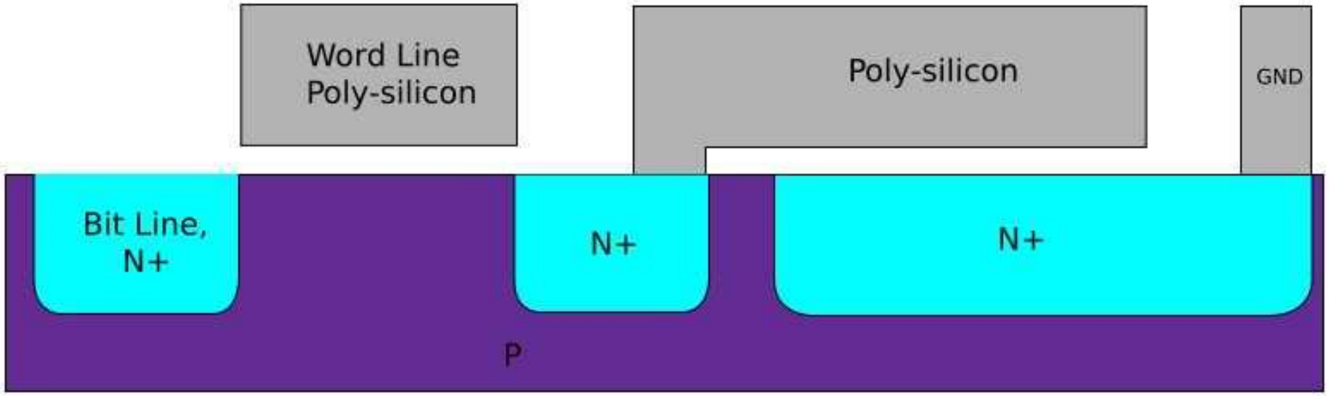


- Good regularity – easy to design
- Very high density if good cells are used

DRAM Cell

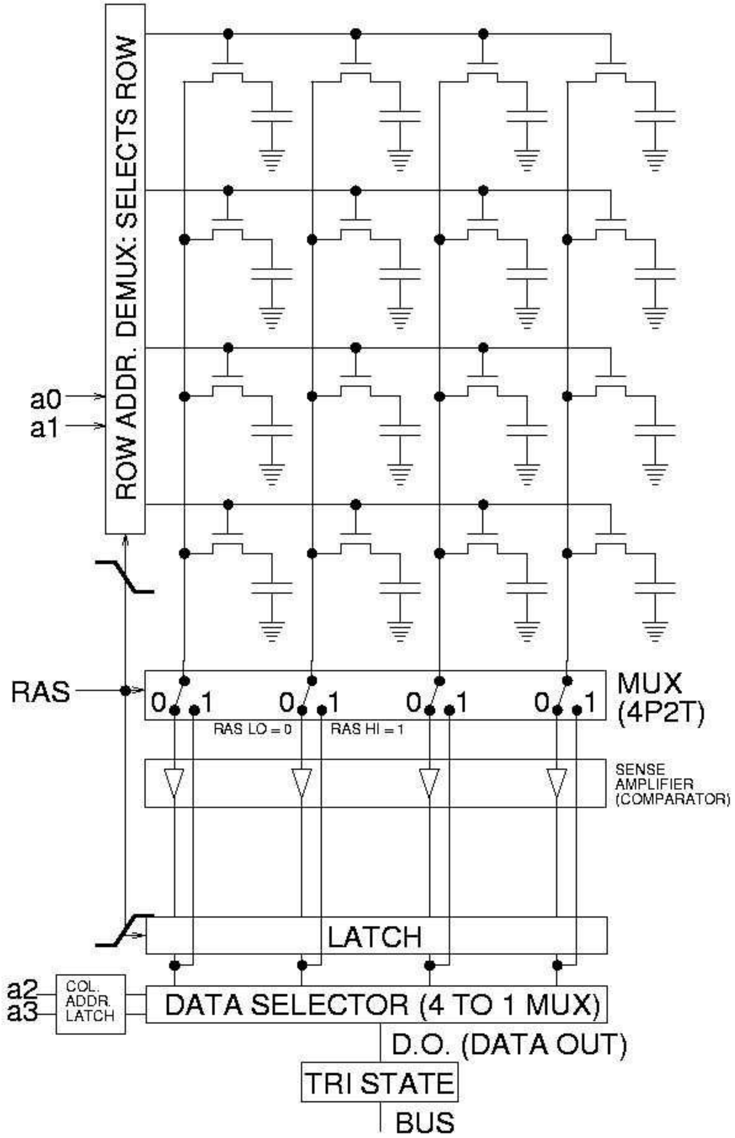






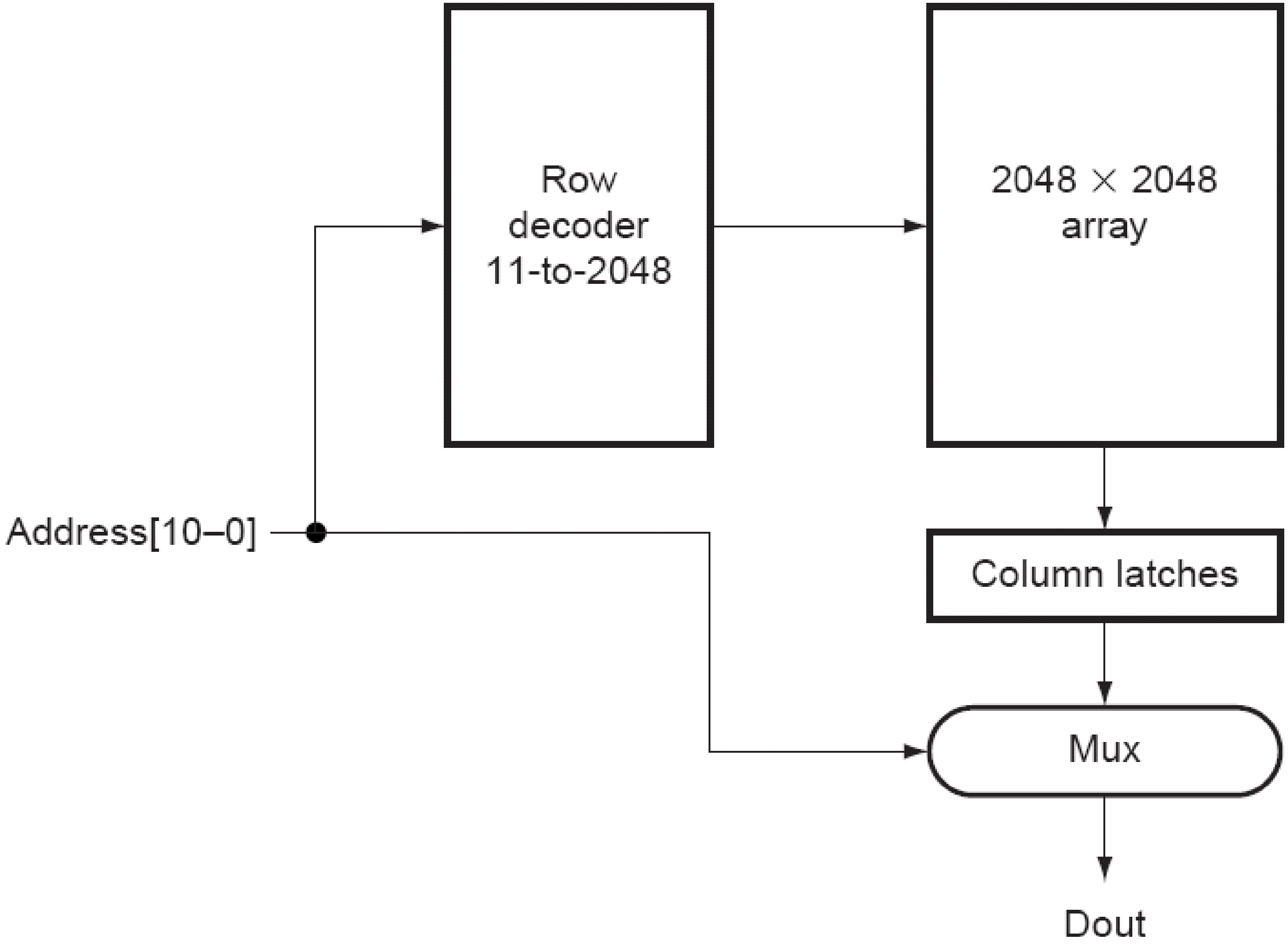
Source: Wikipedia

Access of DRAM (16bit×1)



Source: Wikipedia

Access of DRAM (4M×1)



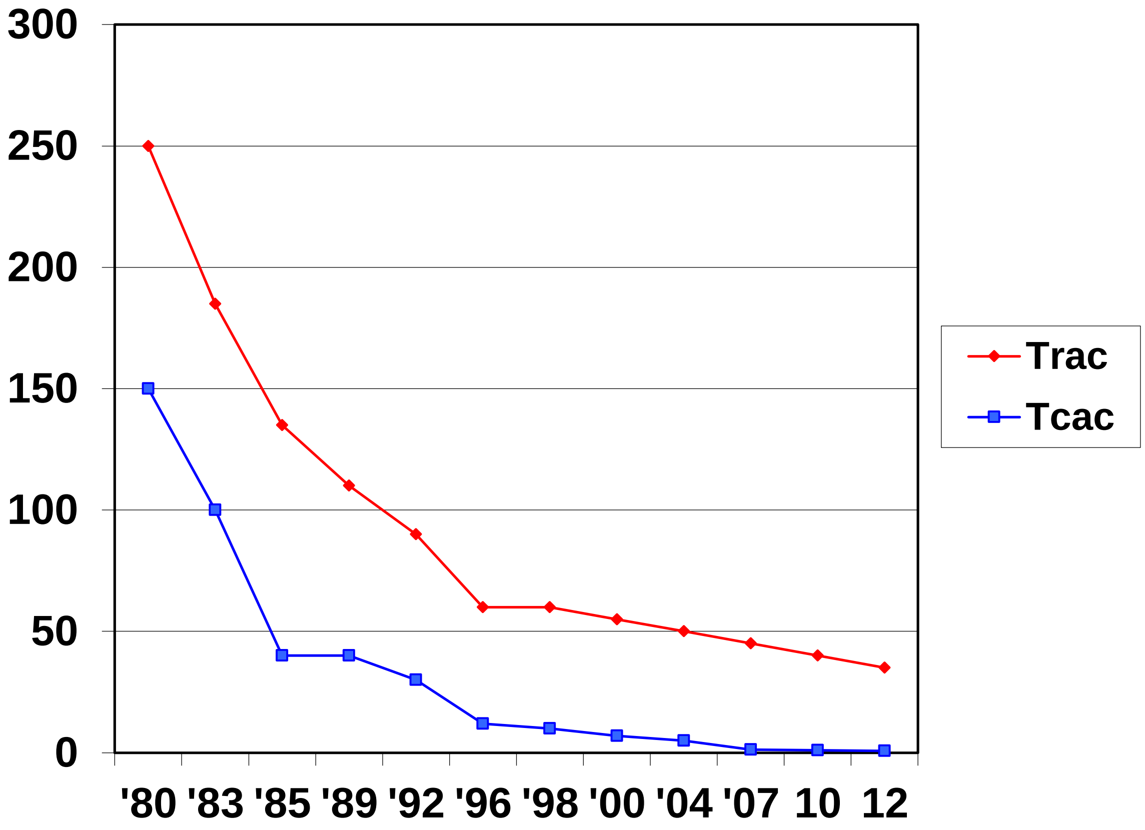
DDR SDRAM

Double Data Rate Synchronous DRAMs

- Burst access from a sequential locations
- Starting address, burst length
- Data transferred under control of clock (300 MHz, 2004)
- Clock is used to eliminate the need of synchronization and the need of supplying successive address
- Data transfer on both raising and falling edges of clock

DRAM Generations

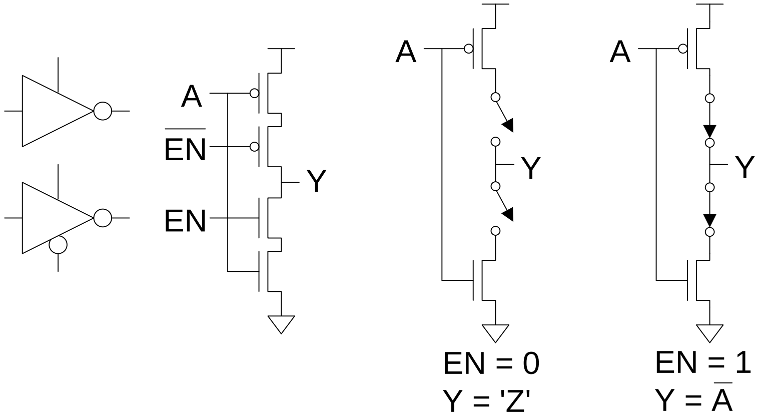
Year	Capacity	$/GB
1980	64Kbit	$1,500,000
1983	256Kbit	$500,000
1985	1Mbit	$200,000
1989	4Mbit	$50,000
1992	16Mbit	$15,000
1996	64Mbit	$10,000
1998	128Mbit	$4,000
2000	256Mbit	$1,000
2004	512Mbit	$250
2007	1Gbit	$50
2010	2Gbit	$30
2012	4Gbit	$1

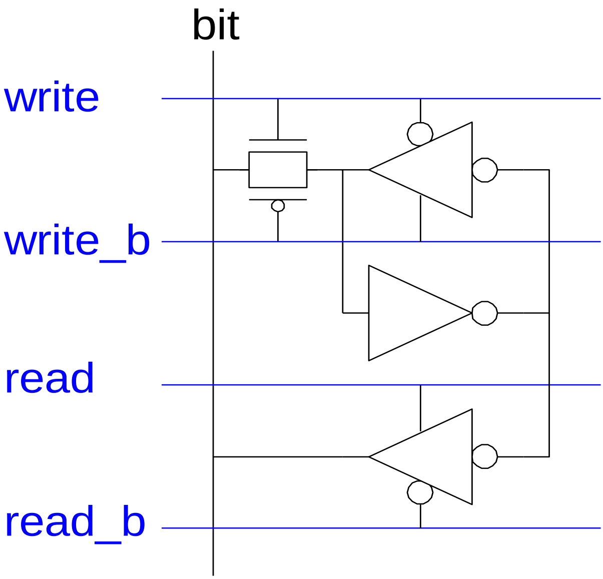


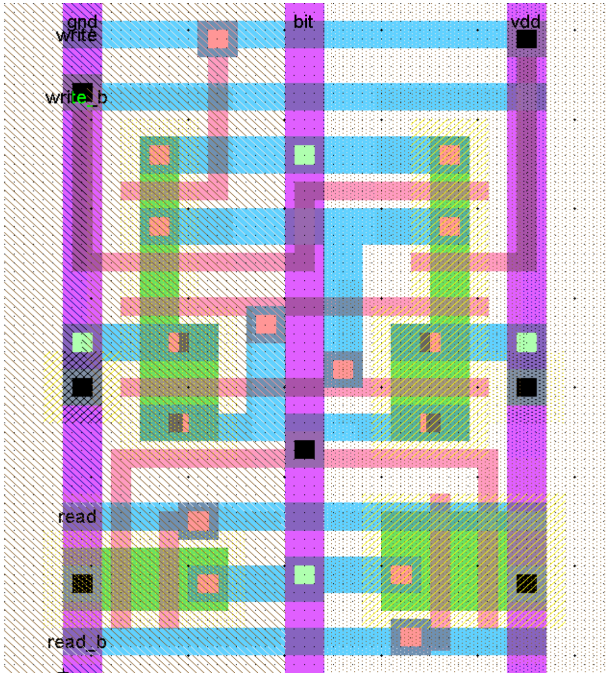
Trac: access time to a new row
Tcac: column access time to existing row

12T SRAM Cell

- Basic building block: SRAM Cell
  - Holds one bit of information, like a latch
  - Must be read and written
- 12-transistor (12T) SRAM cell
  - Use a simple latch connected to bitline

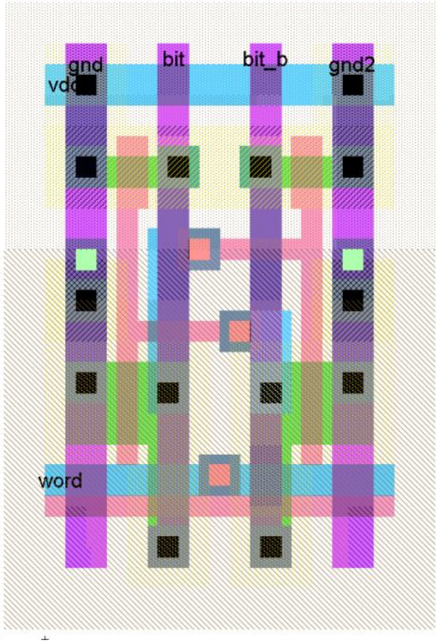


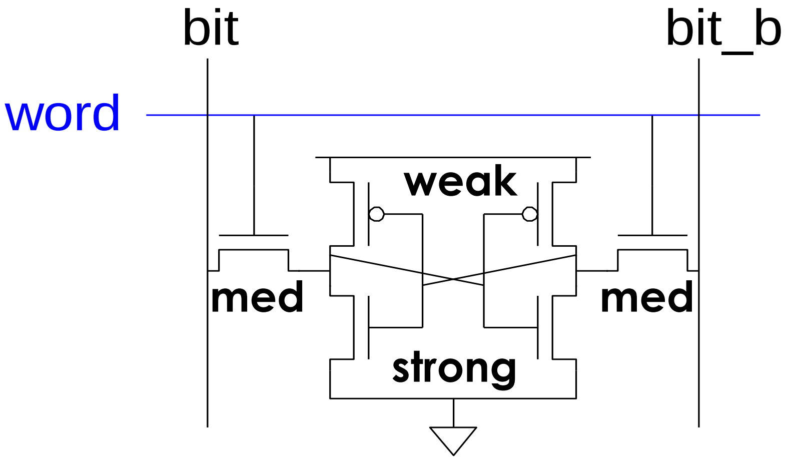




6T SRAM Cell

- Cell size accounts for most of array size
  - Reduce cell size at expense of complexity
- 6T SRAM Cell
  - Used in most commercial chips
  - Data stored in cross-coupled inverters
- Read:
  - Precharge bit, bit_b
  - Raise wordline
- Write:
  - Drive data onto bit, bit_b
  - Raise wordline





Comparisons of Various Technologies

Memory technology	Typical access time	$ per GB in 2004
SRAM	0.5 – 5 ns	$2000 – $5,000
DRAM	50 – 70 ns	$20 – $75
Magnetic disk	5,000,000 – 20,000,000 ns	$0.20 – $2

Technology Trends

	Capacity	Speed (latency)
Logic:	4x in 1.5 years	4x in 3 years
DRAM:	4x in 3 years	2x in 10 years
Disk:	4x in 3 years	2x in 10 years

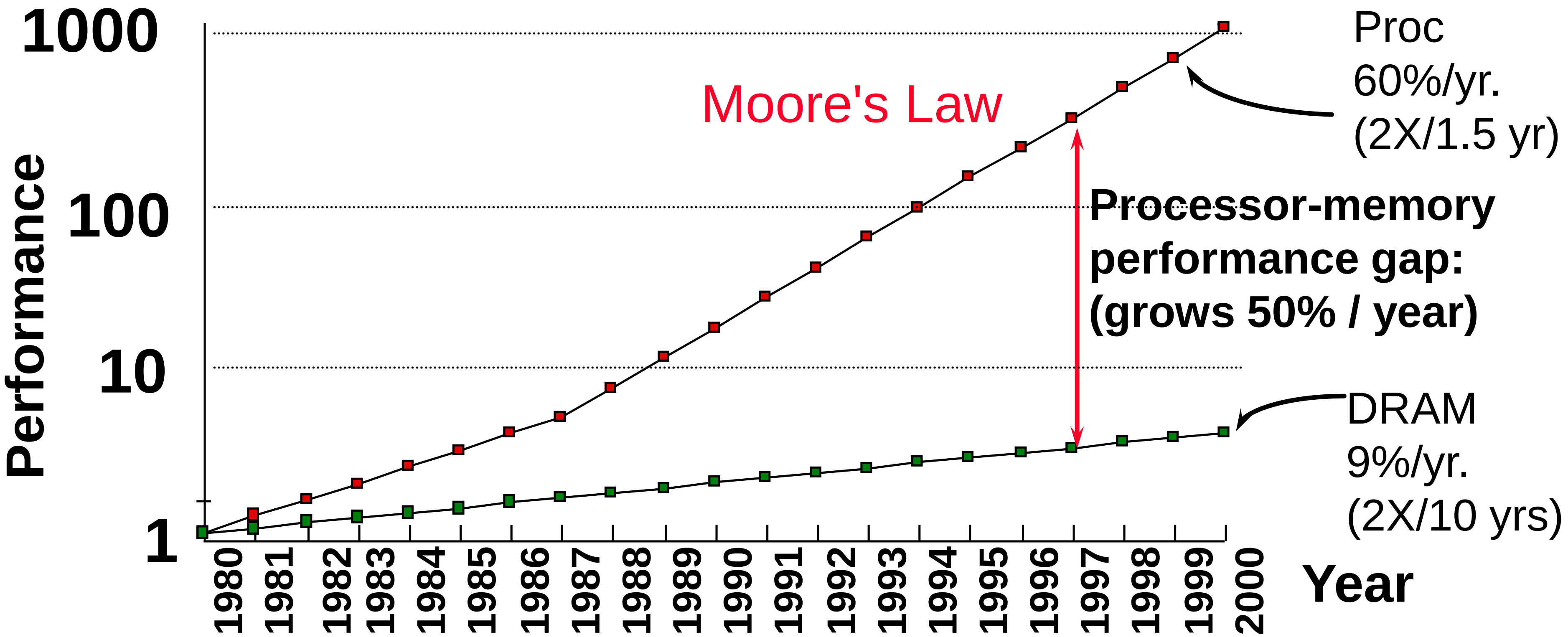
DRAM

Year	Size	Cycle Time
1980	64 Kb	250 ns
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	125 ns
1996	64 Mb	110 ns
1998	128Mb	100 ns
2000	256Mb	90 ns
2002	512Mb	80 ns
2004	1Gb	70 ns
2006	2Gb	60 ns

1000:1!

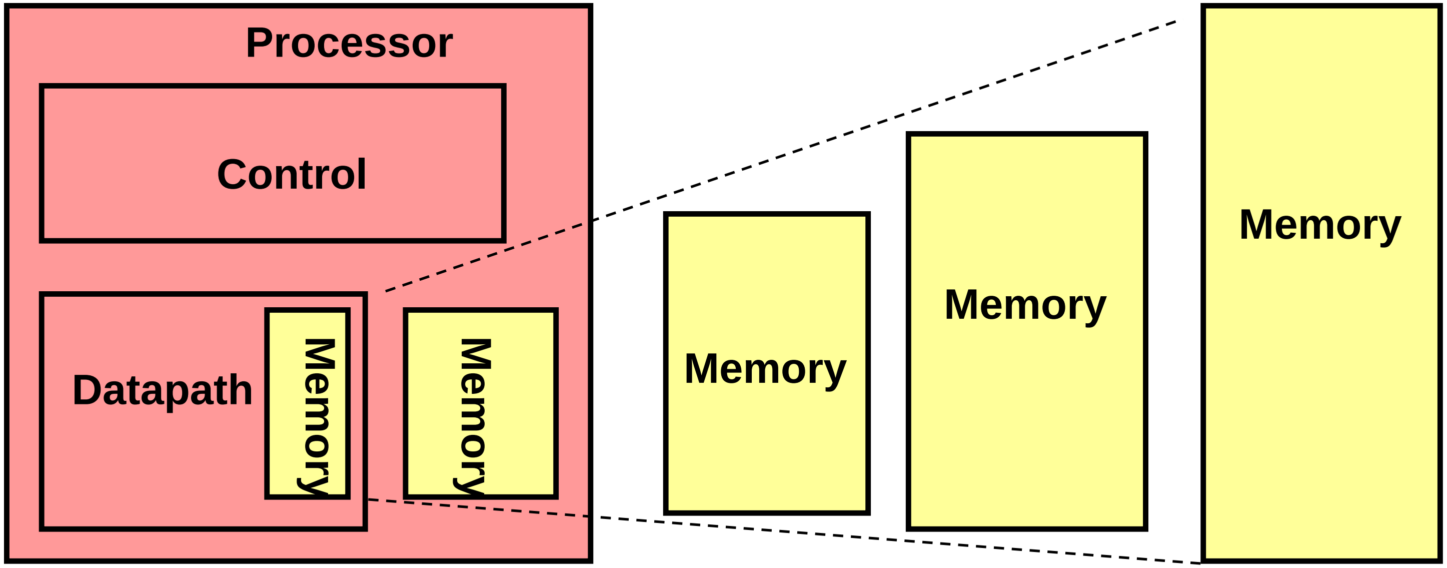
2:1!

Processor Memory Latency Gap



Solution: Memory Hierarchy

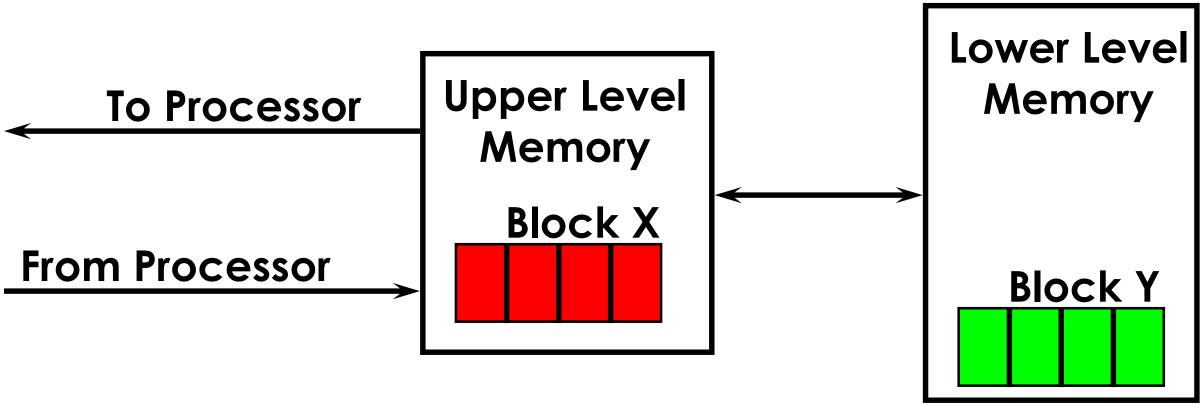
- An Illusion of a large, fast, cheap memory
  - Fact: Large memories slow, fast memories small
  - How to achieve: hierarchy, parallelism
- An expanded view of memory system:



Speed: Fastest ... Slowest
Size: Smallest ... Biggest
Cost: Highest ... Lowest

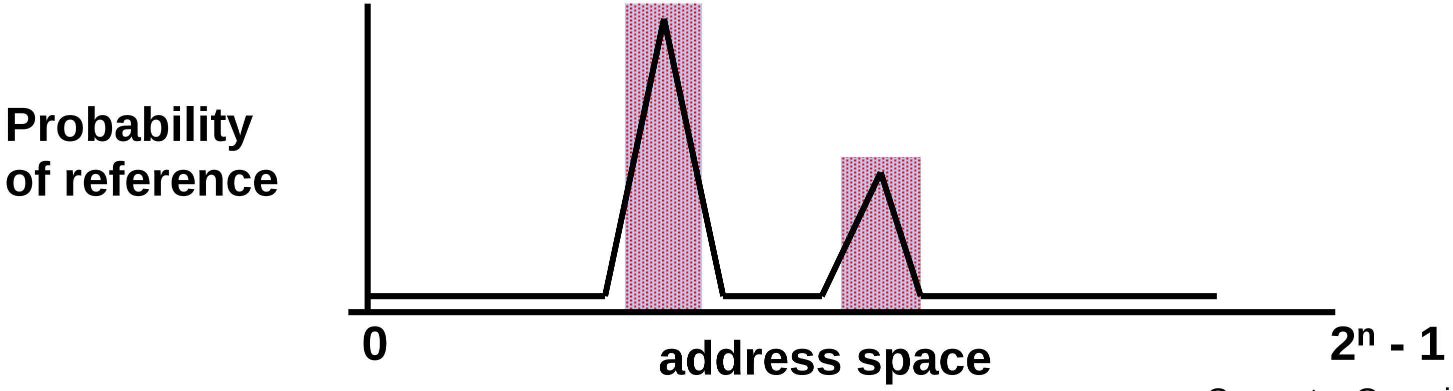
Memory Hierarchy: Principle

- At any given time, data is copied between only two adjacent levels:
  - Upper level: the one closer to the processor
    - Smaller, faster, uses more expensive technology
  - Lower level: the one away from the processor
    - Bigger, slower, uses less expensive technology
- *Block*: basic unit of information transfer
  - Minimum unit of information that can either be present or not present in a level of the hierarchy



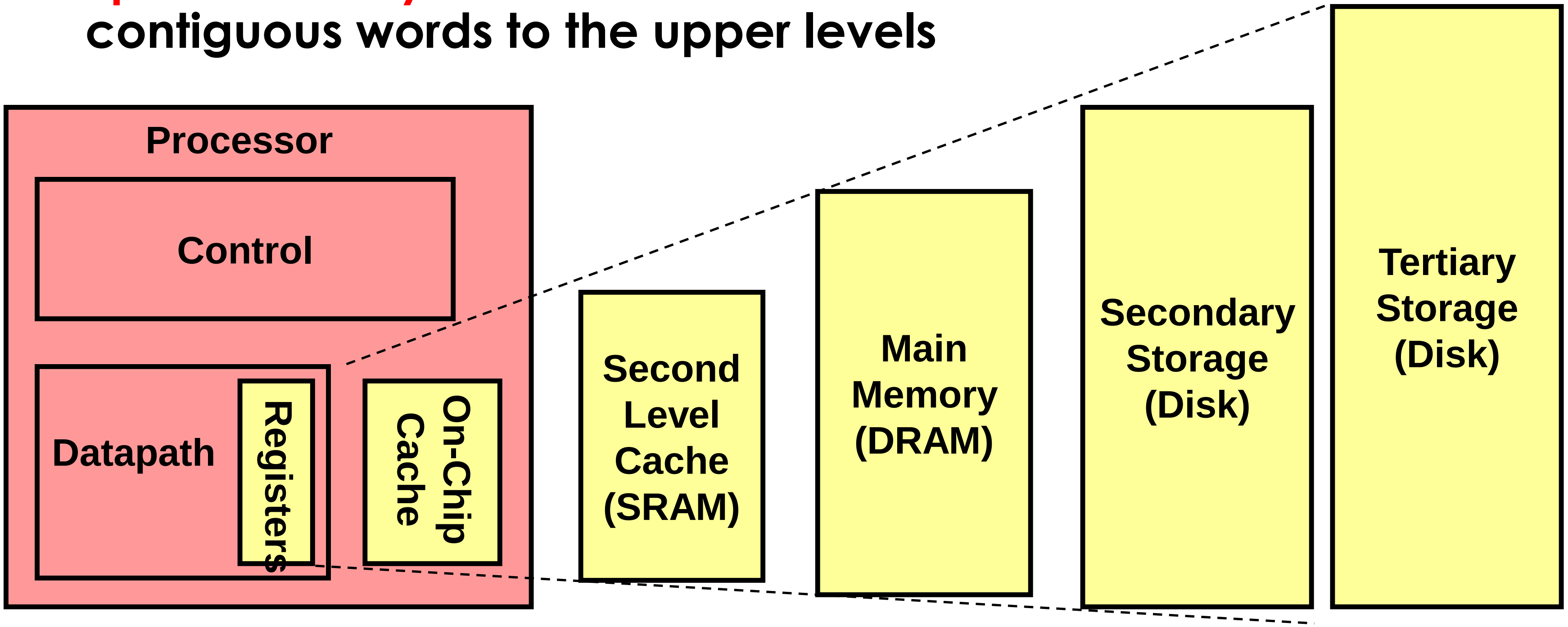
Why Hierarchy Works?

- *Principle of Locality*:
  - Program access a relatively small portion of the address space at any instant of time
  - 80/20 rule: 20% of code executed 80% of time
- Two types of locality:
  - Temporal locality: if an item is referenced, it will tend to be referenced again soon
  - Spatial locality: if an item is referenced, items whose addresses are close by tend to be referenced soon



How Does It Work?

- Temporal locality: keep most recently accessed data items closer to the processor
- Spatial locality: move blocks consists of contiguous words to the upper levels



Speed (ns):	1'ns	10'ns	100'ns	10,000,000'ns (10's ms)	10,000,000,000'ns (10's sec)
Size (bytes):	100's	K's	M's	G's	T's

Levels of Memory Hierarchy

Capacity
Access Time
Cost

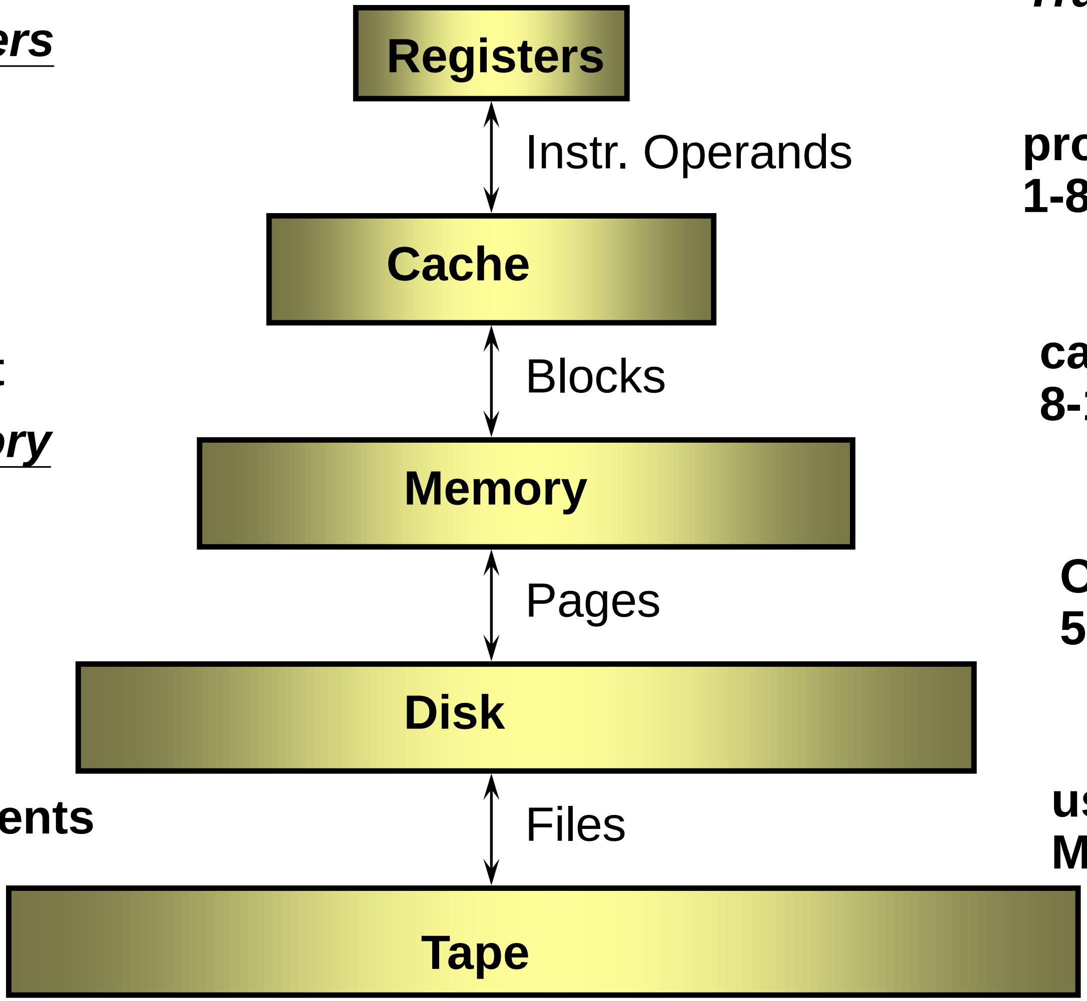
CPU Registers
100s Bytes
<10s ns

Cache
K Bytes
10-100 ns
$.01-.001/bit

Main Memory
M Bytes
100ns-1us
$.01-.001

Disk
G Bytes
ms
10^{-3} - 10^{-4} cents

Tape
infinite
sec-min
10^{-6}



Registers

Instr. Operands

Cache

Blocks

Memory

Pages

Disk

Files

Tape

Upper Level

Staging
Transfer Unit

faster

prog./compiler
1-8 bytes

cache controller
8-128 bytes

OS
512-4K bytes

user/operator
Mbytes

Larger

Lower Level

How Is the Hierarchy Managed?

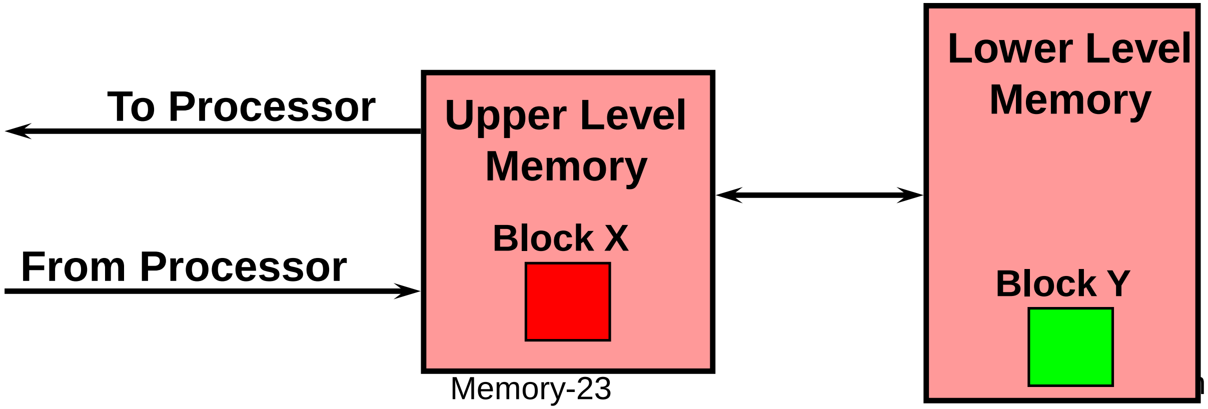
- Registers ⇔ Memory
  - by compiler (programmer?)
- cache ⇔ memory
  - by hardware
- memory ⇔ disks
  - by hardware and software (operating system): virtual memory
  - by programmer: files

Memory Hierarchy Technology

- Performance of main memory:
  - Latency: related directly to *Cache Miss Penalty*
    - Access Time: time between request and word arrives
    - Cycle Time: time between requests
  - Bandwidth: Large Block Miss Penalty (interleaved memory, L2)
- Non-so-random access technology:
  - Access time varies from location to location and from time to time, ex: disk, CDROM
- Sequential access technology: access time linear in location (ex: tape)

Memory Hierarchy: Terminology

- Hit: data appears in upper level (Block X)
  - Hit rate: fraction of memory access found in the upper level
  - Hit time: time to access the upper level
    - RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieved from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: time to replace a block in the upper level + time to deliver the block to the processor (latency + transmit time)
- Hit Time << Miss Penalty



4 Questions for Hierarchy Design

Q1: Where can a block be placed in the upper level?
→ *block placement*

Q2: How is a block found if it is in the upper level?
→ *block identification*

Q3: Which block should be replaced on a miss?
→ *block replacement*

Q4: What happens on a write?
→ *write strategy*

Summary of Memory Hierarchy

- Two different types of locality:
  - Temporal Locality (Locality in Time)
  - Spatial Locality (Locality in Space)
- Using the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology
  - Provide access at the speed offered by the fastest technology
- DRAM is slow but cheap and dense:
  - Good for presenting users with a BIG memory system
- SRAM is fast but expensive, not very dense:
  - Good choice for providing users FAST accesses

Outline

- Memory hierarchy
- The basics of caches
- Measuring and improving cache performance
- Virtual memory
- A common framework for memory hierarchy
- Using a finite state machine to control a simple cache
- Parallelism and memory hierarchies: cache coherence

Levels of Memory Hierarchy

Capacity
Access Time
Cost

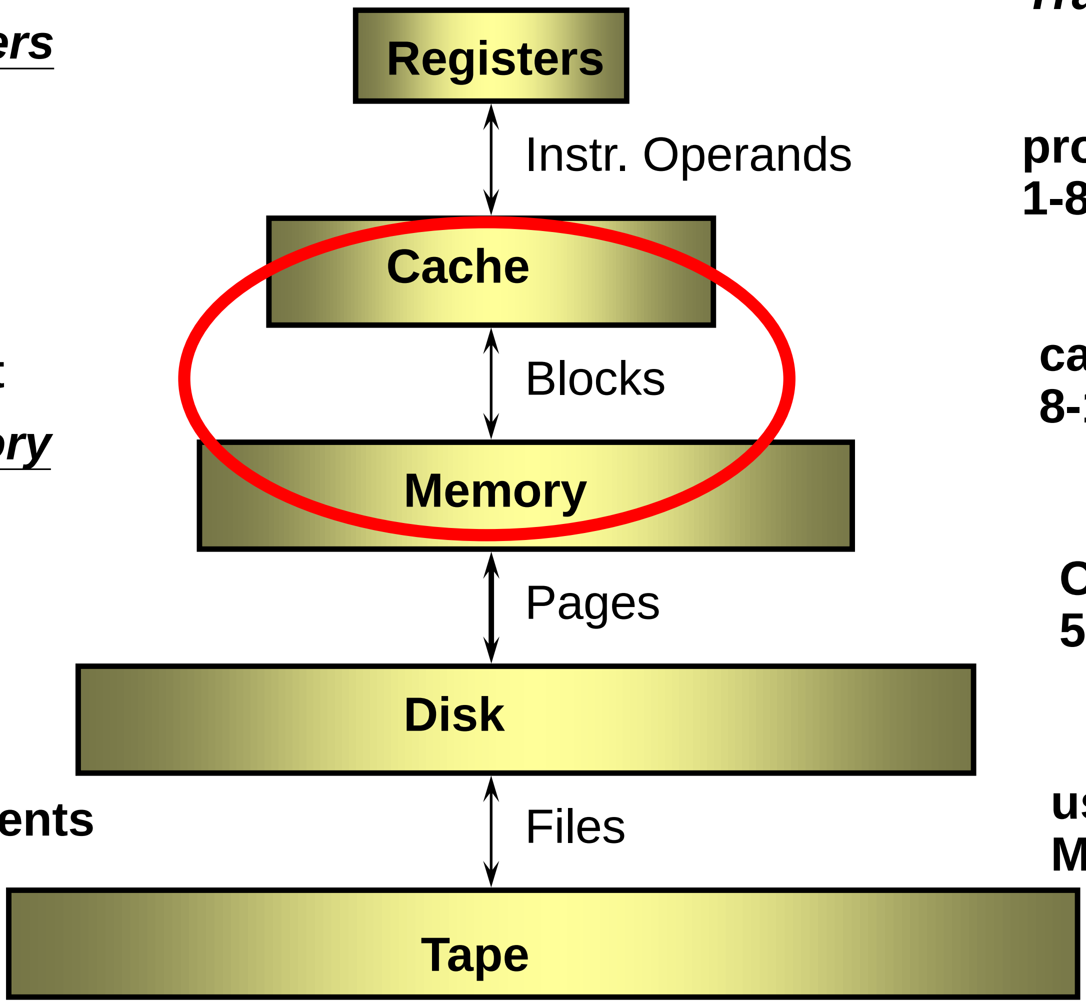
CPU Registers
100s Bytes
<10s ns

Cache
K Bytes
10-100 ns
$.01-.001/bit

Main Memory
M Bytes
100ns-1us
$.01-.001

Disk
G Bytes
ms
10^{-3} - 10^{-4} cents

Tape
infinite
sec-min
10^{-6}



Registers

Instr. Operands

Cache

Blocks

Memory

Pages

Disk

Files

Tape

Upper Level

Staging
Transfer Unit

prog./compiler
1-8 bytes

cache controller
8-128 bytes

OS
512-4K bytes

user/operator
Mbytes

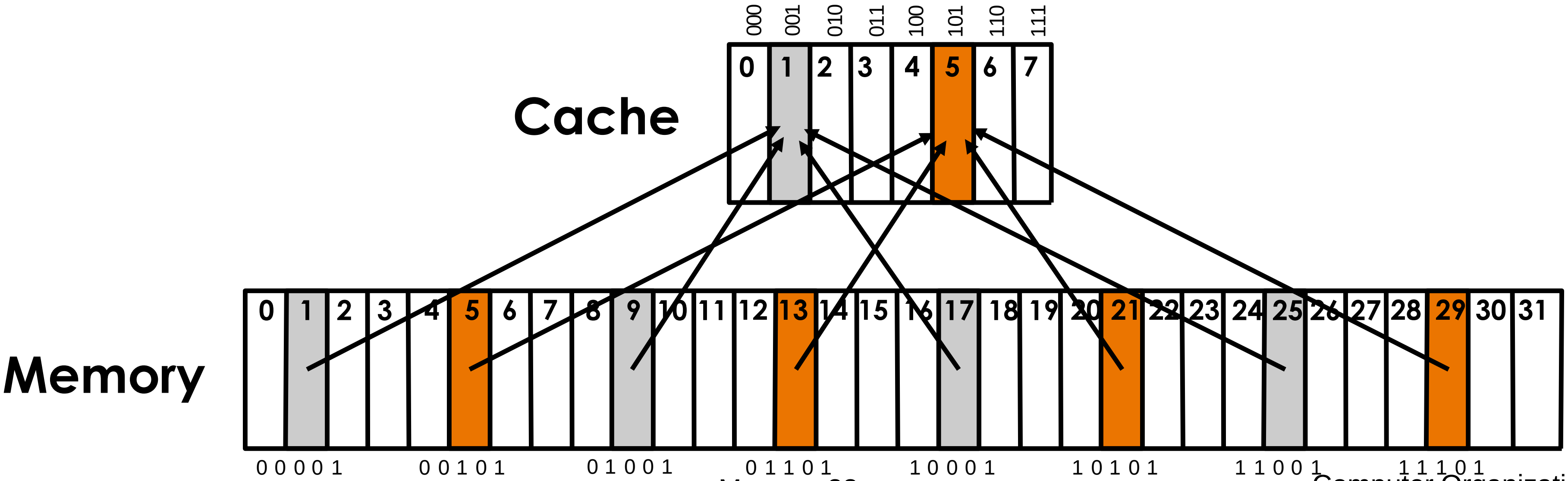
faster

Larger

Lower Level

Basics of Cache

- Our first example: *direct-mapped cache*
- Block Placement:
  - For each item of data at the lower level, there is exactly one location in cache where it might be
  - Address mapping: modulo number of blocks
- Block identification:
  - How to know if an item is in cache?
  - If it is, how do we find it?



Block Identification: Tag & 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
- Word access sequence: 22, 26, 22, 26, 16, 3, 16, 18

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

Cache Example

22, 26, 22, 26, 16, 3, 16, 18

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	N Y	10	Mem[10110]
111	N		

Cache Example

22, 26, 22, 26, 16, 3, 16, 18

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

22, 26, 22, 26, 16, 3, 16, 18

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

22, 26, 22, 26, 16, 3, 16, 18

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

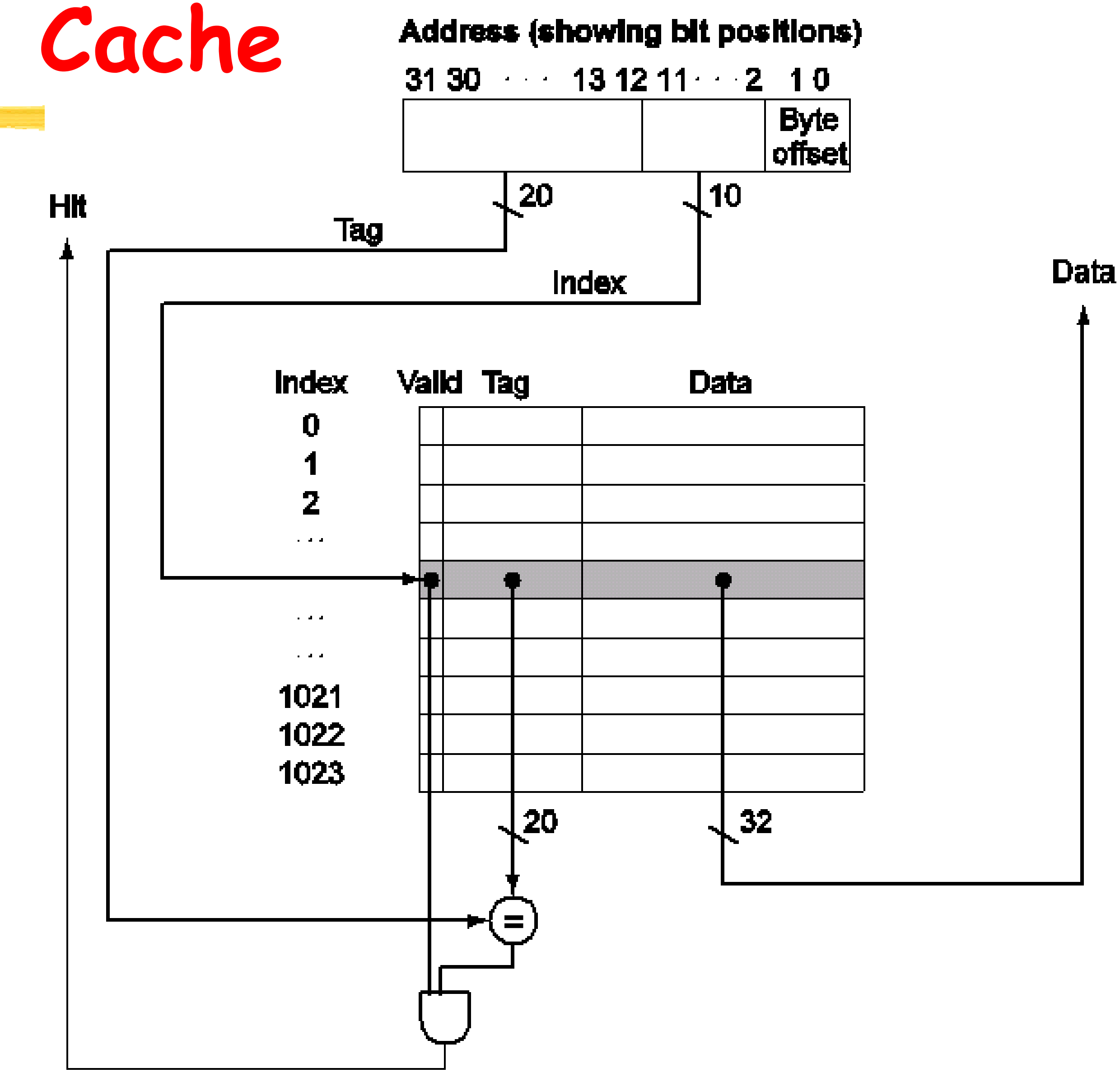
22, 26, 22, 26, 16, 3, 16, 18

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		

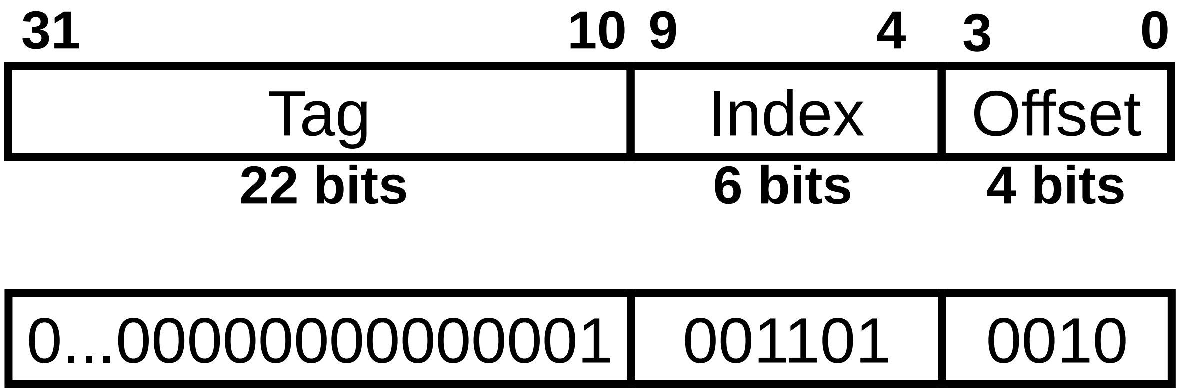
Accessing a Cache

- 1K words, 1-word block:
  - Cache index: lower 10 bits
  - Cache tag: upper 20 bits
  - Valid bit (when start up, valid is 0)



Example: Larger Block Size

- 64 blocks, 16 bytes/block
  - To what block number does byte address 1234 map?
- Block address = $\lfloor 1234/16 \rfloor$ = 77
- Block number = 77 modulo 64 = 13
- Byte offset = 1234 module 16 = 2



31 10	9 4	3 0
Tag	Index	Offset
22 bits	6 bits	4 bits
0...0000000000000001	001101	0010

1234_{10}=$(10011010010)_2$

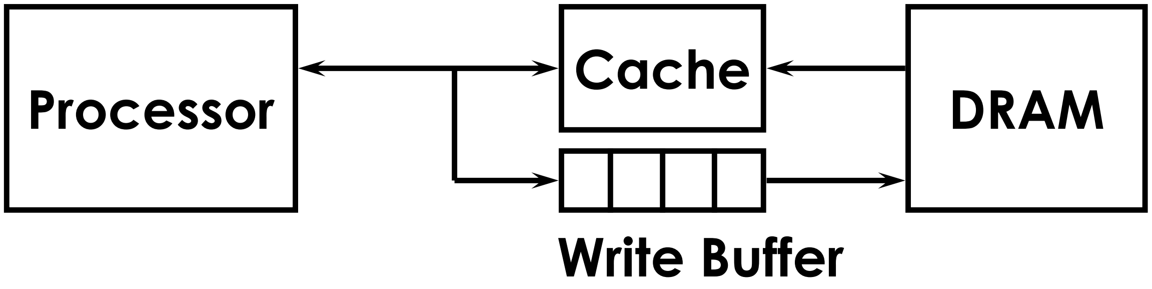
Cache Misses

- 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-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - Ex: 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

Avoid Waiting for Memory in Write Through



- Use a *write buffer* (WB):
  - Processor: writes data into cache and WB
  - Memory controller: write WB data to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
- Memory system designer's nightmare:
  - Store frequency > 1 / DRAM write cycle
  - Write buffer saturation → CPU stalled

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
  - Use a write buffer to allow replacing block to be read first

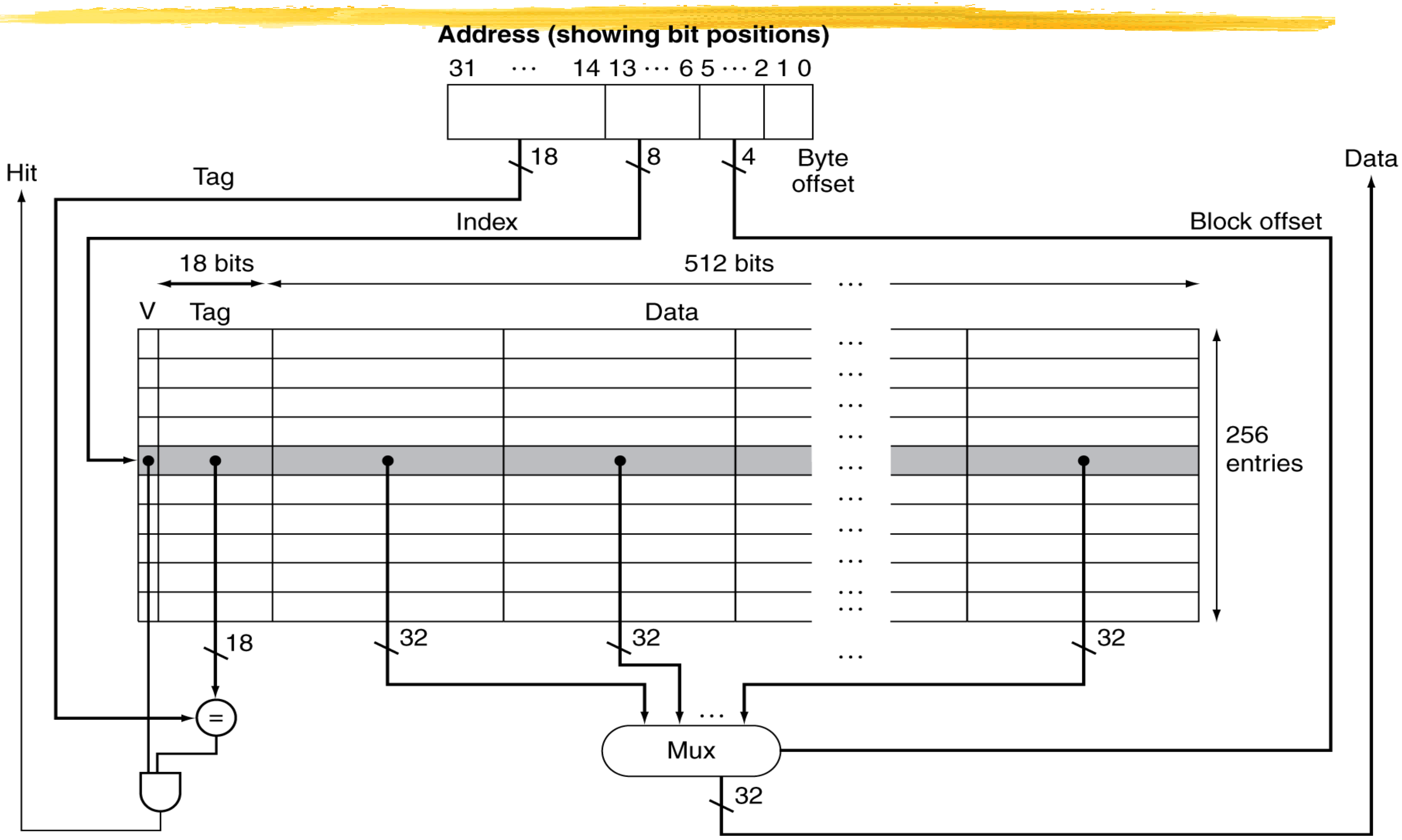
Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (ex: initialization)
- For write-back
  - Usually fetch the block

Example: Intrinsity FastMATH

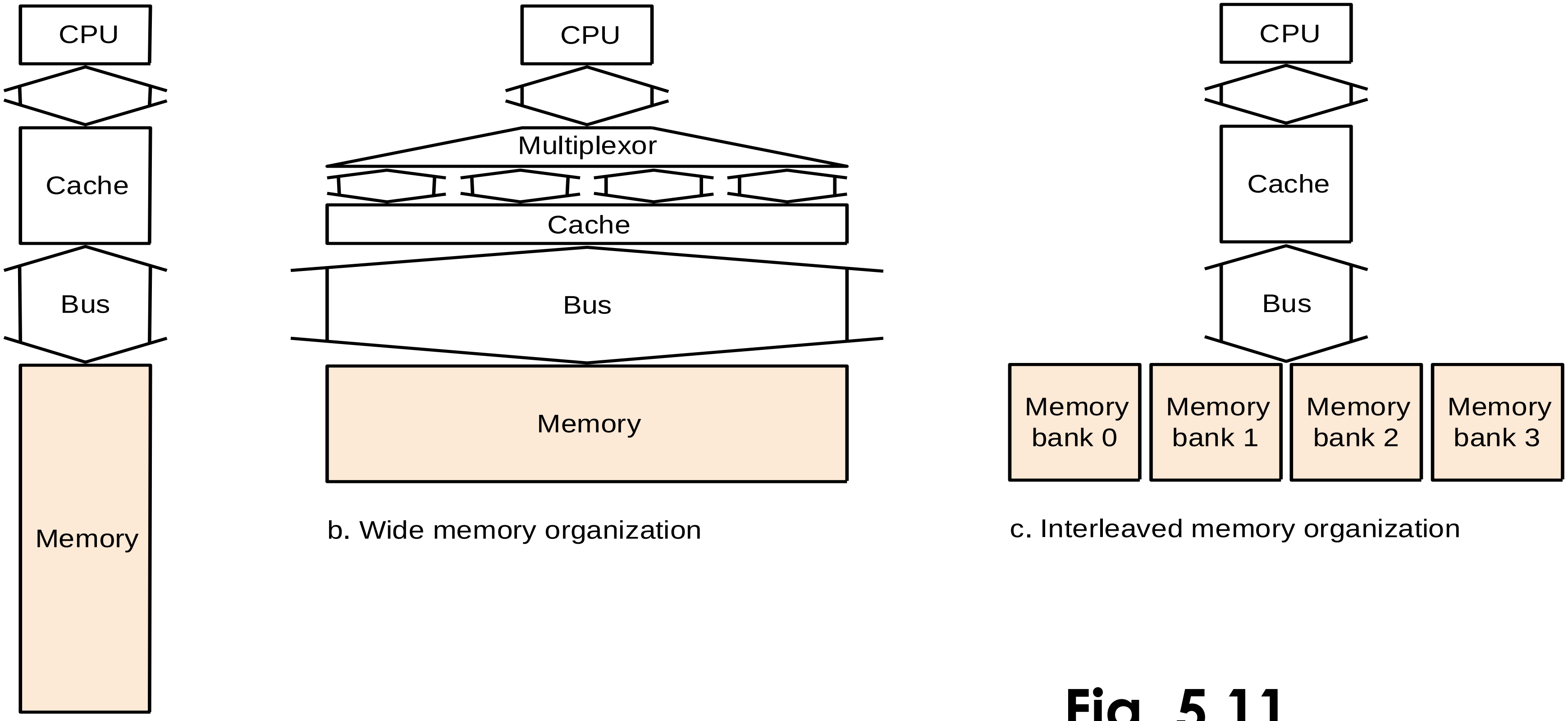
- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: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

Example: Intrinsity FastMATH



Memory Design to Support Cache

- How to increase memory bandwidth to reduce miss penalty?



a. One-word-wide memory organization

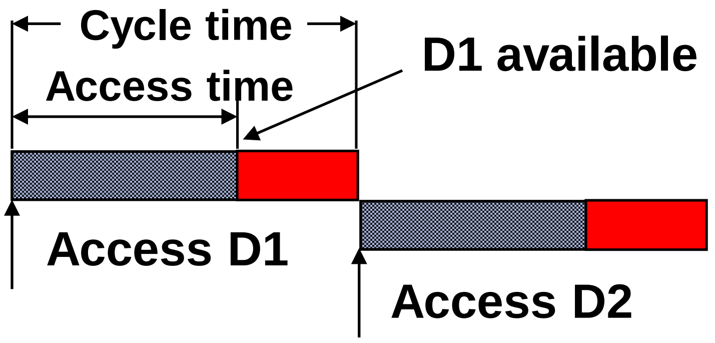
b. Wide memory organization

c. Interleaved memory organization

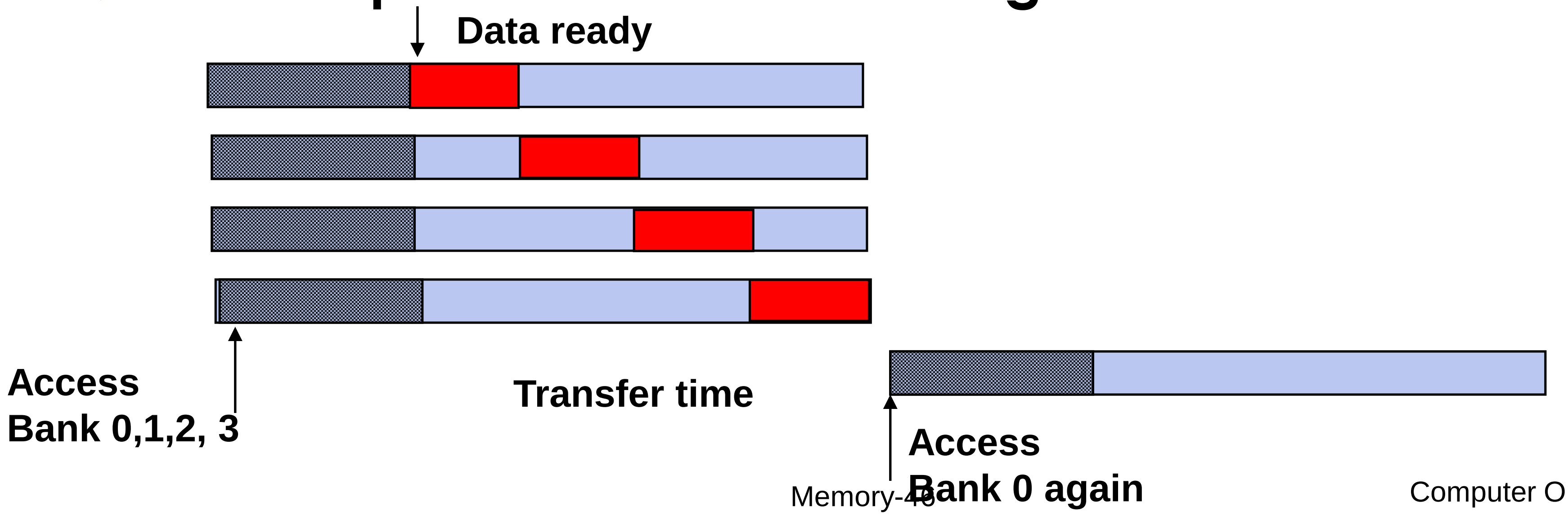
Fig. 5.11

Interleaving for Bandwidth

- Access pattern without interleaving:



- Access pattern with interleaving



Miss Penalty for Different Memory Organizations

Assume

- 1 memory bus clock to send the address
- 15 memory bus clocks for each DRAM access initiated
- 1 memory bus clock to send a word of data
- A cache block = 4 words
- Four memory organizations :
  - A one-word-wide bank of DRAMs
  - Miss penalty = 1 + 4 x 15 + 4 x 1 = 65

  - A two-word-wide bank of DRAMs
  - Miss penalty = 1 + 2 x 15 + 2 x 1 = 33

  - A four-word-wide bank of DRAMs
  - Miss penalty = 1 + 1 x 15 + 1 = 17

  - A four-bank, one-word-wide bus of DRAMs
  - Miss penalty = 1 + 1 x 15 + 4 x 1 = 20

Outline

- Memory hierarchy
- The basics of caches
- Measuring and improving cache performance
- Virtual memory
- A common framework for memory hierarchy
- Using a finite state machine to control a simple cache
- Parallelism and memory hierarchies: cache coherence

Measuring Cache Performance

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

$$\text{Memory stall cycles}$$

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

$$= \frac{\text{Instructio ns}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instructio n}} \times \text{Miss penalty}$$

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

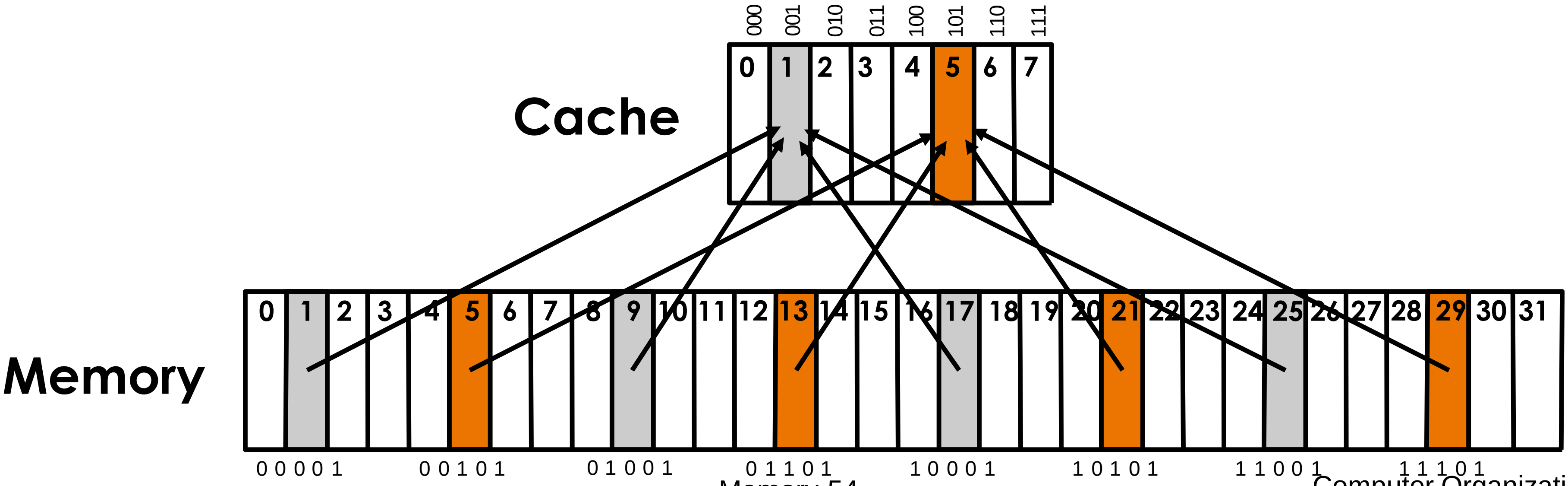
Improving Cache Performance

- Reduce the miss ratio
- Reduce the time to hit in the cache
- Reduce the miss penalty

Basics of Cache

- Our first example: *direct-mapped cache*
- Block Placement:
  - For each item of data at the lower level, there is exactly one location in cache where it might be
  - Address mapping: modulo number of blocks
- Block identification:
  - How to know if an item is in cache?
  - If it is, how do we find it?

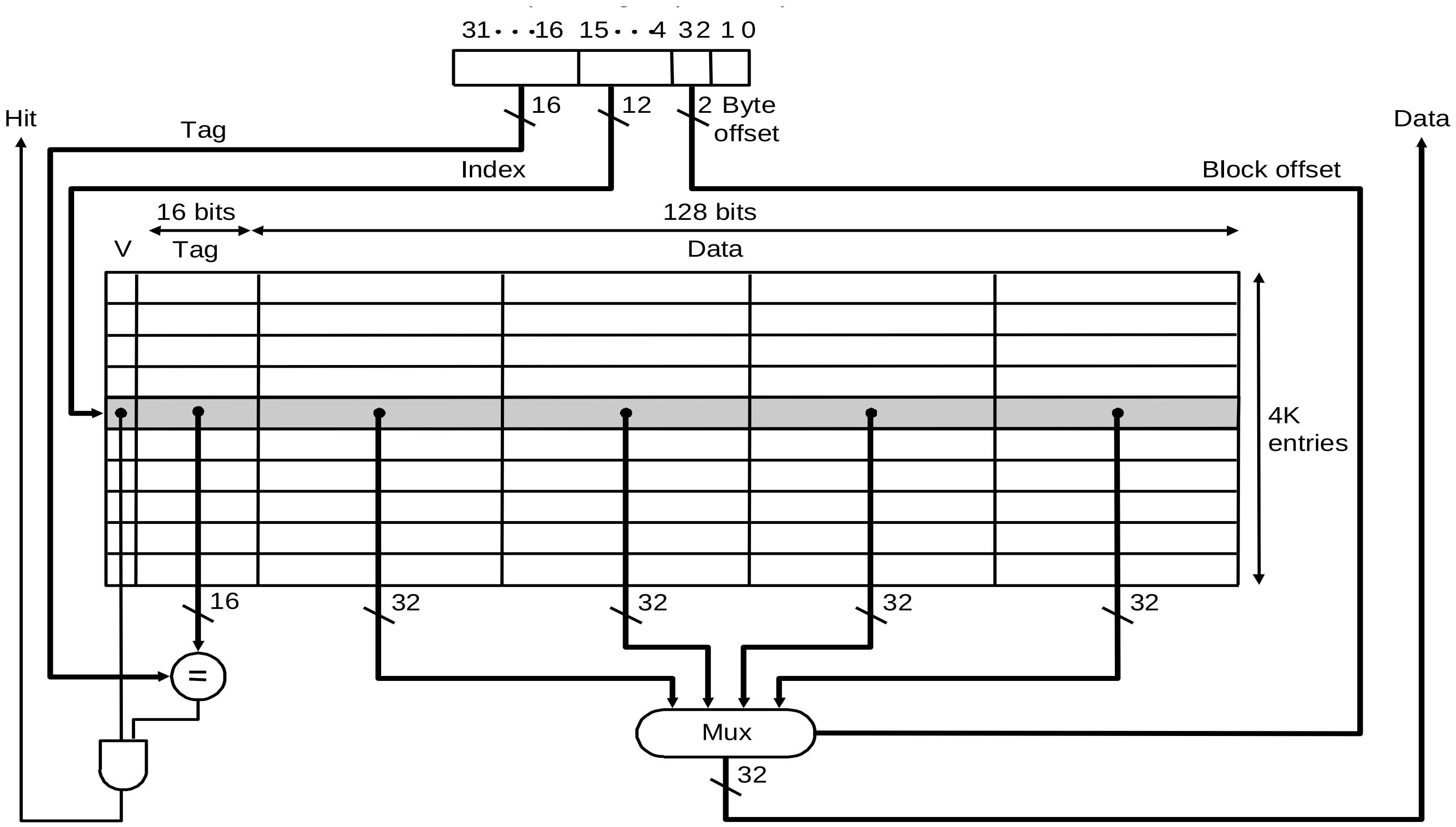
Tag and valid bit



Exploiting Spatial Locality

- Increase block size for spatial locality

Total no. of tags and valid bits reduced

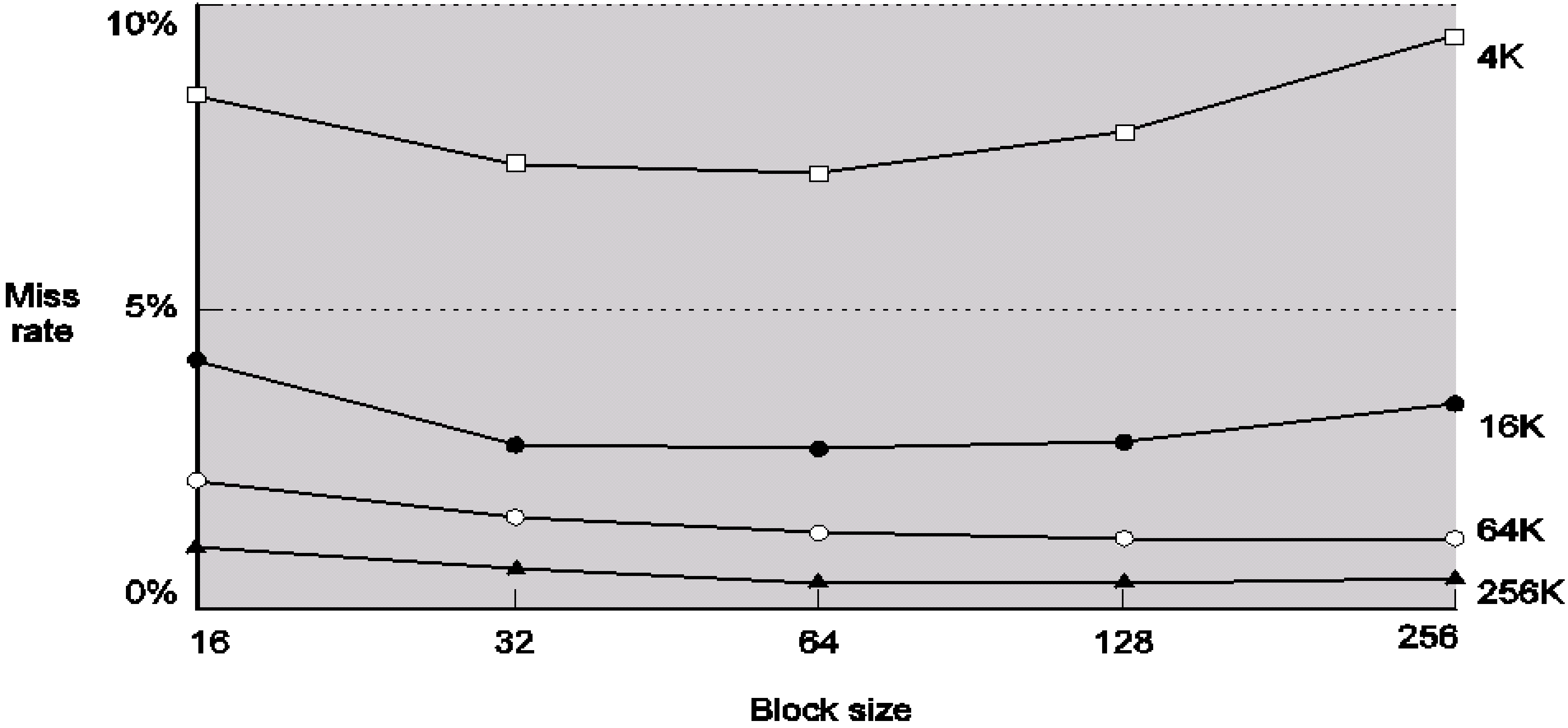


Block Size Considerations

- Larger blocks "should" reduce miss rate
  - Thanks to spatial locality
- But in a fixed-sized cache
  - Larger blocks → fewer of blocks
    - 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

Block Size on Performance

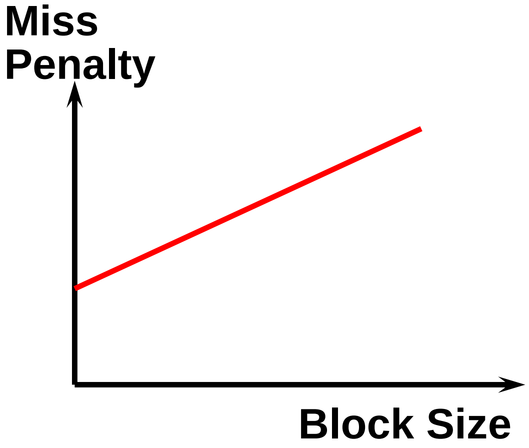
- Increase block size tends to decrease miss rate

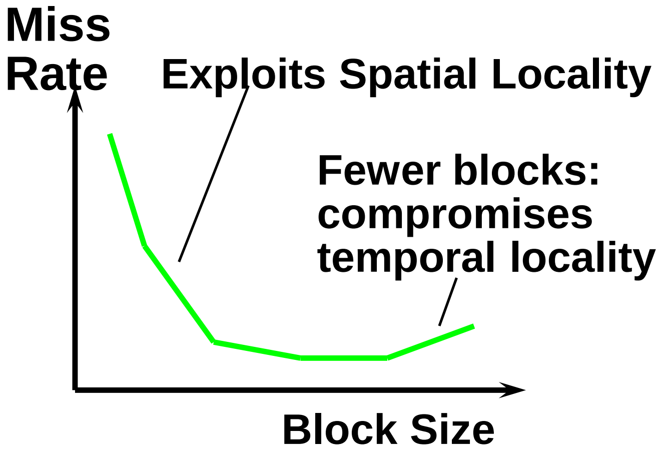


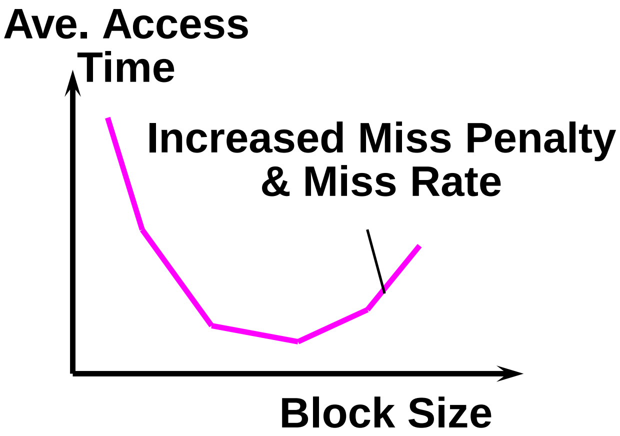
Block Size Tradeoff

- Larger block size takes advantage of spatial locality and improve miss ratio, BUT:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size too big, miss rate goes up
    - Too few blocks in cache → high competition
- Average access time:

  = hit time + miss penalty × miss rate





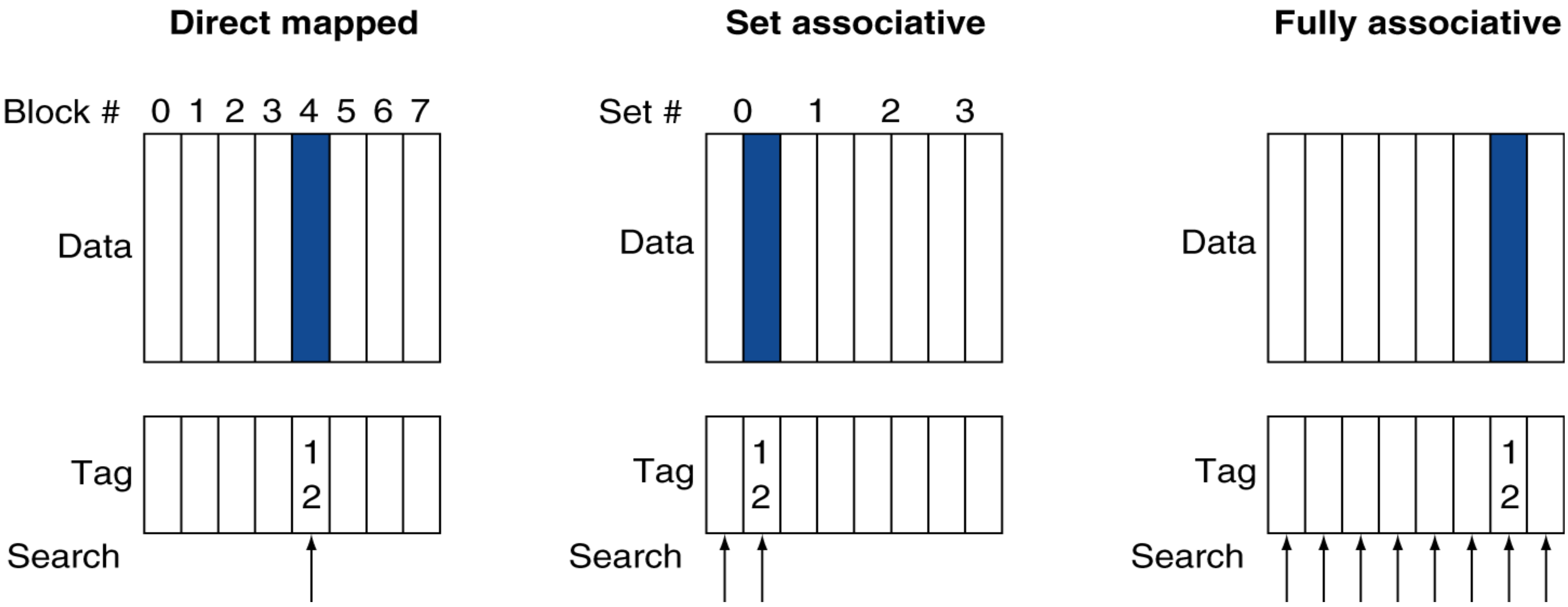


Associative Caches

- 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)

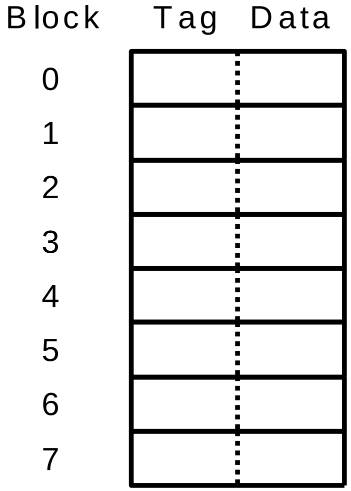
Associative Cache Example

- Placement of a block whose address is 12:

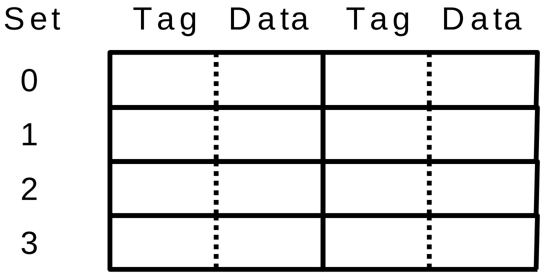


Possible Associativity Structures

One-way set associative
(direct mapped)

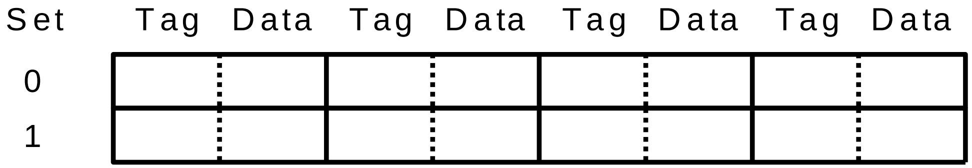


Two-way set associative

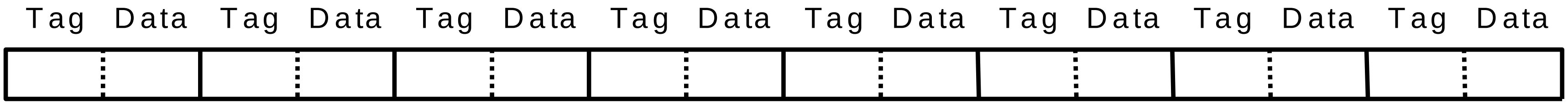


An 8-block cache

Four-way set associative



Eight-way set associative (fully associative)



Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped 0 hit and 5 misses

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 1 hit and 4 misses

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 2 hits and 3 misses

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]	

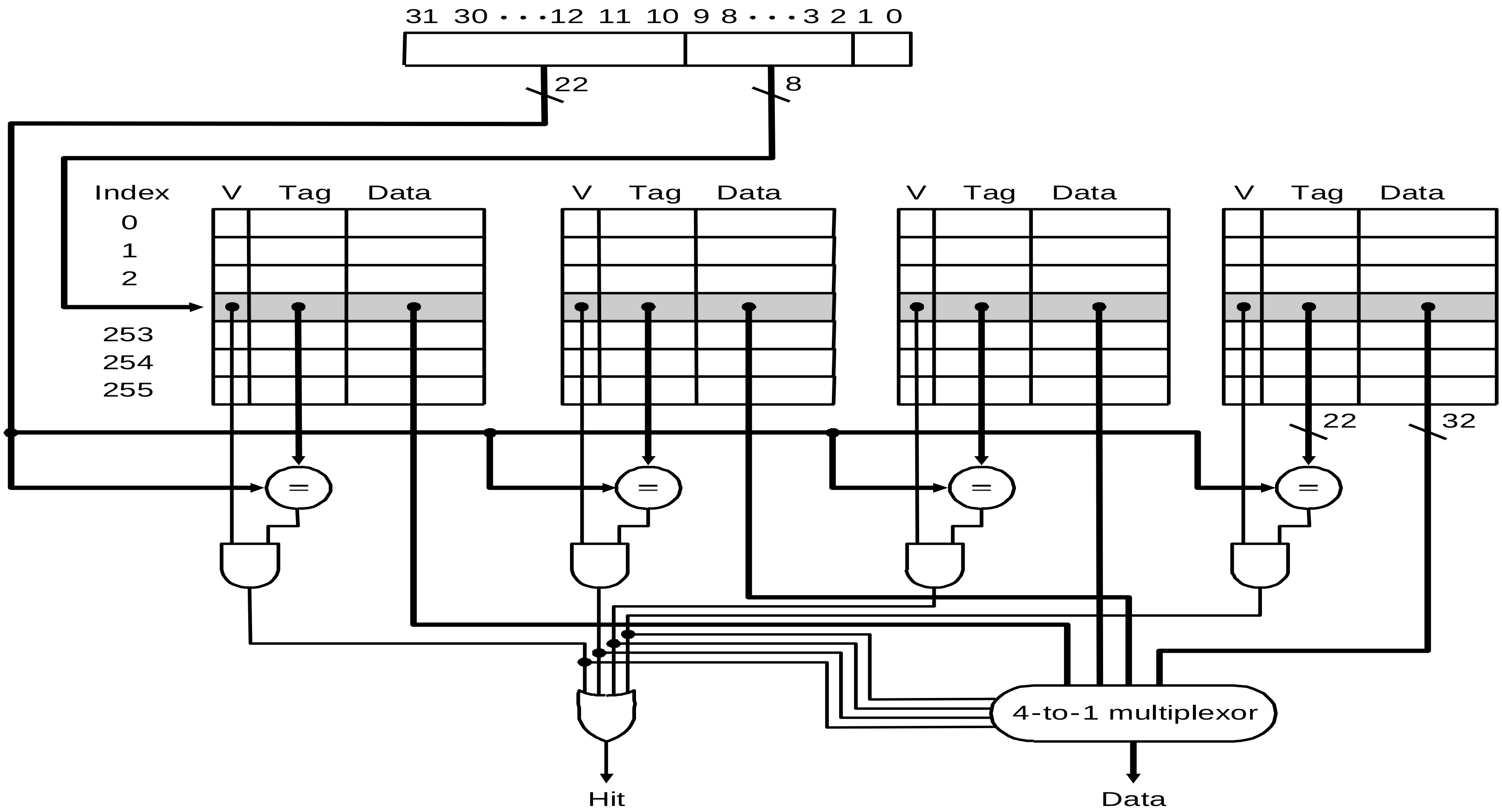
How Much Associativity

- 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%

Tag Comparing Structure

- *N*-way set-associative cache
  - *N* comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss decision and set selection
- Direct mapped cache
  - Cache block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue, recover later if miss (similar to static branch prediction)

A 4-Way Set-Associative Cache



- Increasing associativity shrinks index, expands tag

Data Placement Policy

- Direct mapped cache:
  - Each memory block mapped to one location
  - No need to make any decision
  - Current item replaces previous one in location
- *N*-way set associative cache:
  - Each memory block has choice of *N* locations
- Fully associative cache:
  - Each memory block can be placed in ANY cache location
- Misses in *N*-way set-associative or fully associative cache:
  - Bring in new block from memory
  - Throw out a block to make room for new block
  - Need to decide on which block to be thrown out

Cache Block Replacement

- Easy for direct mapped
- Set associative or fully associative:
  - Random
  - LRU (Least Recently Used):
    - Hardware keeps track of the access history and replace the block that has not been used for the longest time
  - An example of a pseudo LRU (for a two-way set associative) :
    - Use a pointer pointing at each block in turn
    - Whenever an access to the block the pointer is pointing at, move the pointer to the other block
    - When need to replace, replace the block currently pointed at

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
- Many high-end systems include L-3 cache

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

Multilevel Cache Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - 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

Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size
    - Exploit the spatial locality
    - Design issue: strategy for multicore coherency

      Exclusive invalidation

Interactions with Advanced CPUs

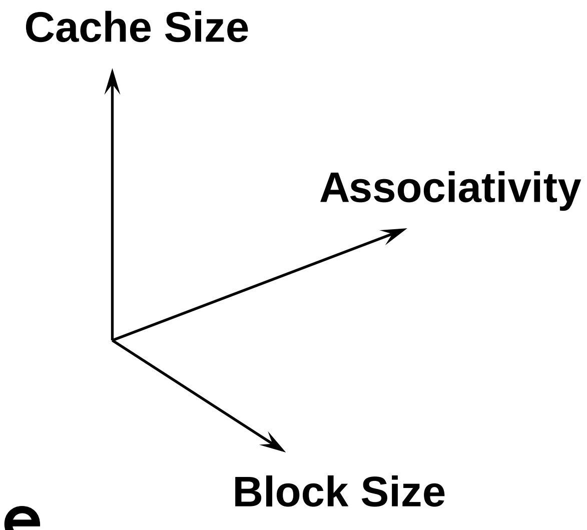
- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyze
  - Use system simulation

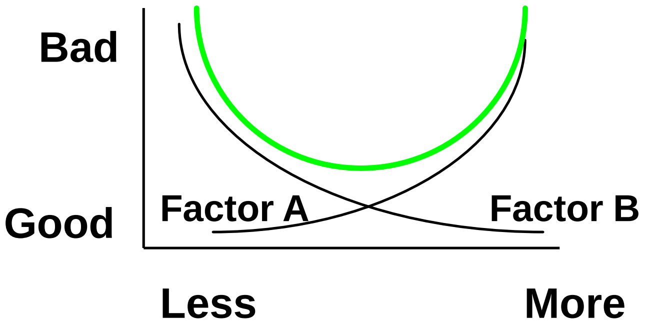
Sources of Cache Misses

- Compulsory (cold start, process migration):
  - First access to a block, not much we can do
  - Note: If you are going to run billions of instruction, compulsory misses are insignificant
- Conflict (collision):
  - >1 memory blocks mapped to same location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Capacity:
  - Cache cannot contain all blocks by program
  - Solution: increase cache size
- Invalidation:
  - Block invalidated by other process (ex: I/O, cache coherence) that updates the memory

Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs. write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins





Cache Summary

- Principle of Locality:
  - Program likely to access a relatively small portion of address space at any instant of time
    - Temporal locality: locality in time
    - Spatial locality: locality in space
- Three major categories of cache misses:
  - Compulsory: ex: cold start misses
  - Conflict: increase cache size or associativity
  - Capacity: increase cache size
- Cache design space
  - total size, block size, associativity
  - replacement policy
  - write-hit policy (write-through, write-back)
  - write-miss policy

Outline

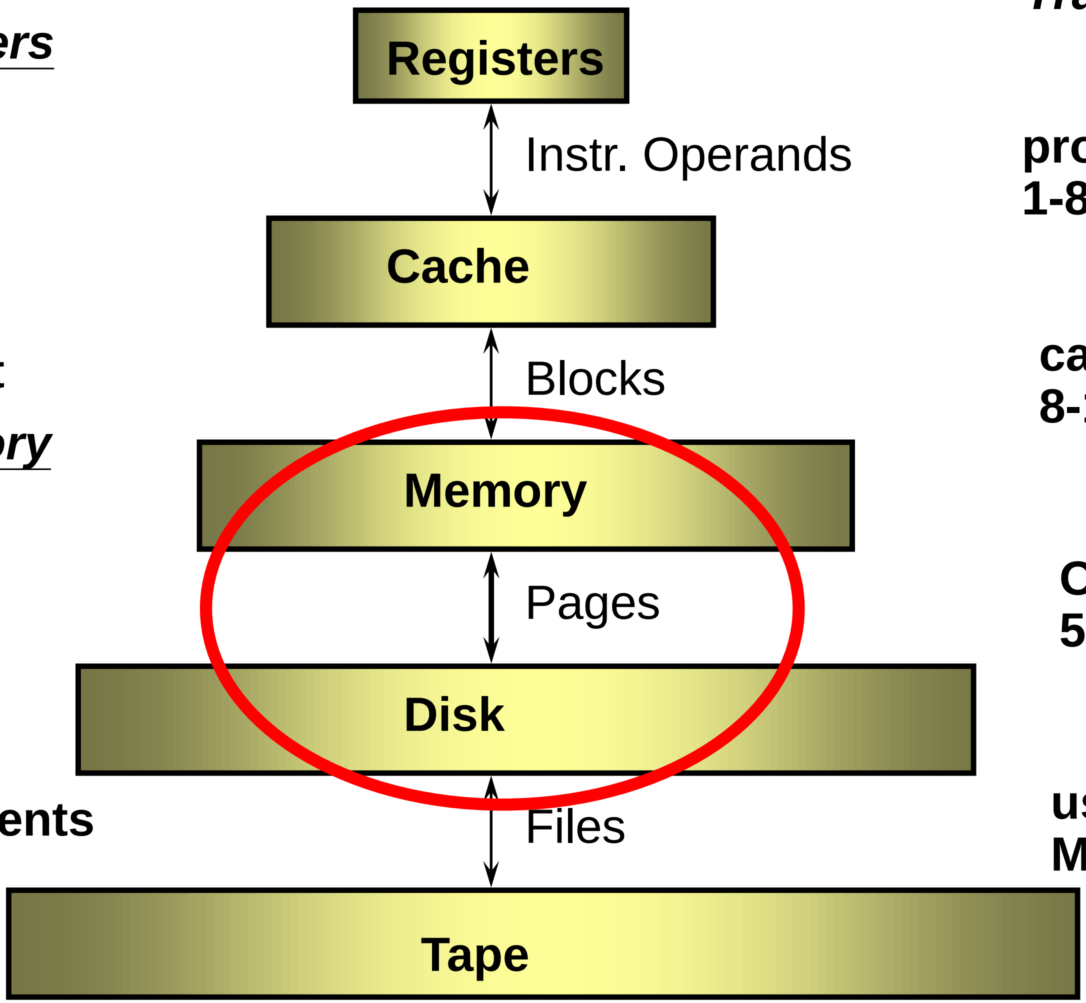
- Memory hierarchy
- The basics of caches
- Measuring and improving cache performance
- Virtual memory
- A common framework for memory hierarchy
- Using a finite state machine to control a simple cache
- Parallelism and memory hierarchies: cache coherence

Levels of Memory Hierarchy

Capacity
Access Time
Cost

Upper Level

Staging
Transfer Unit



CPU Registers
100s Bytes
<10s ns

Cache
K Bytes
10-100 ns
$.01-.001/bit

Main Memory
M Bytes
100ns-1us
$.01-.001

Disk
G Bytes
ms
10^{-3} - 10^{-4} cents

Tape
infinite
sec-min
10^{-6}

prog./compiler
1-8 bytes

cache controller
8-128 bytes

OS
512-4K bytes

user/operator
Mbytes

↑ **faster**

↓ **Larger**

Lower Level

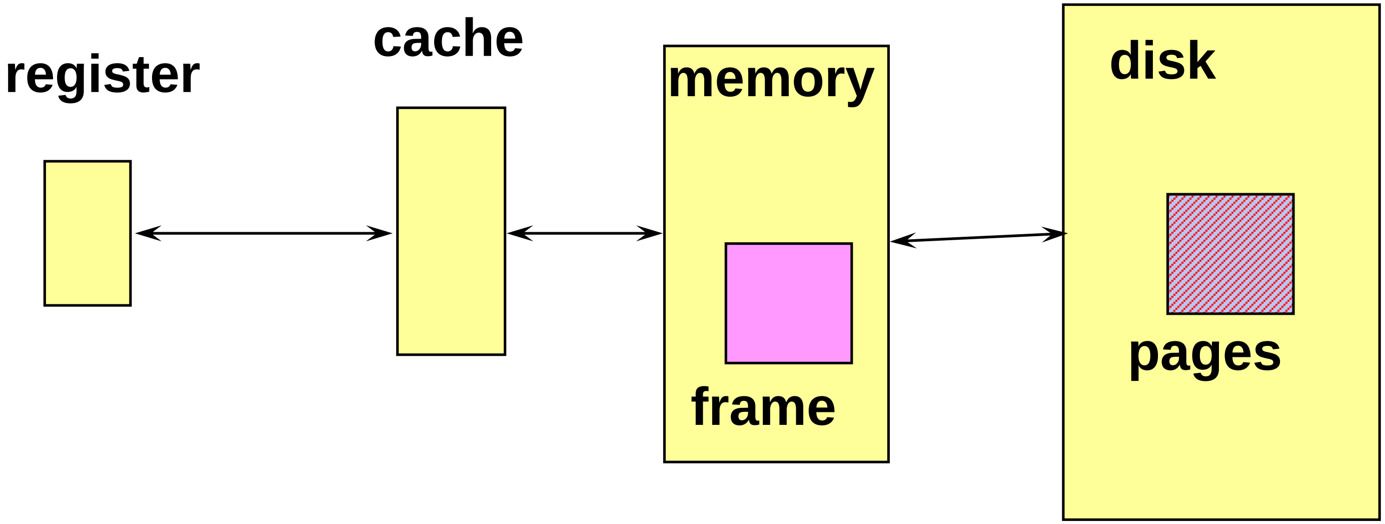
Virtual Memory

- Provide illusion of a large single-level storage
  - Every program has its own address space, starting at address 0, only accessible to itself
    - yet, any program can run anywhere in physical memory
    - executed in a name space (virtual address space) different from memory space (physical address space)
    - virtual memory implements the translation from virtual space to physical space
  - Every program has lots of memory (> physical memory)
- Use main memory as a "cache" for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Many programs run at once with protection and sharing

Virtual Memory - Continued

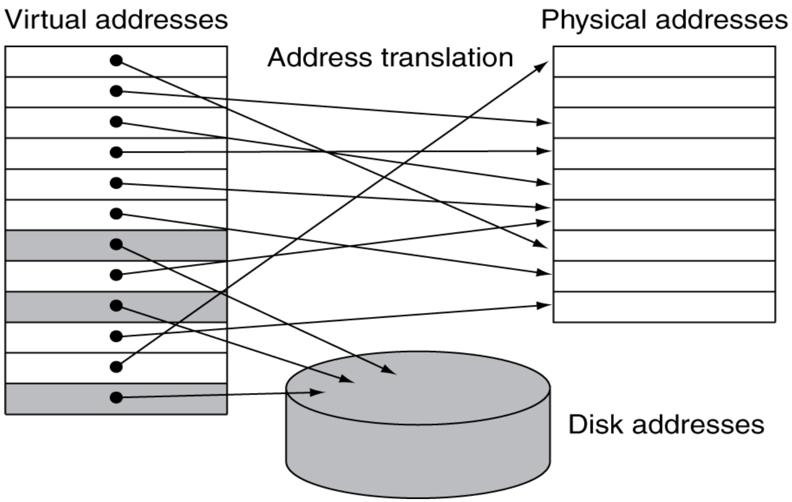
- CPU and OS translate virtual addresses to physical addresses
  - VM "block" is called a page
  - VM translation "miss" is called a page fault

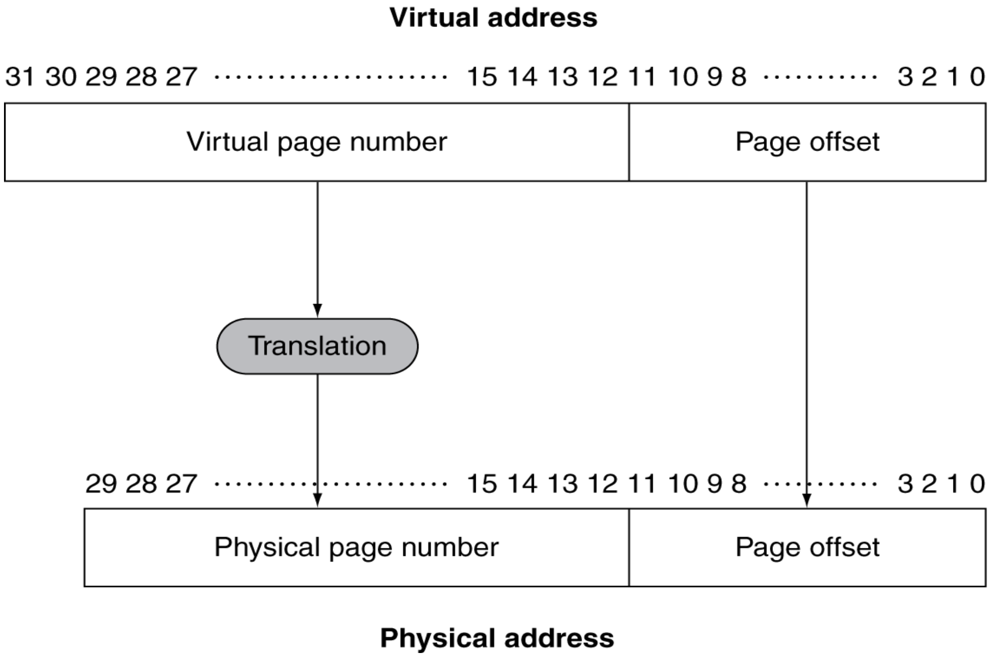
Virtual Memory



Address Translation

- Fixed-size pages (ex: 4K)





Paging

- Virtual and physical address space

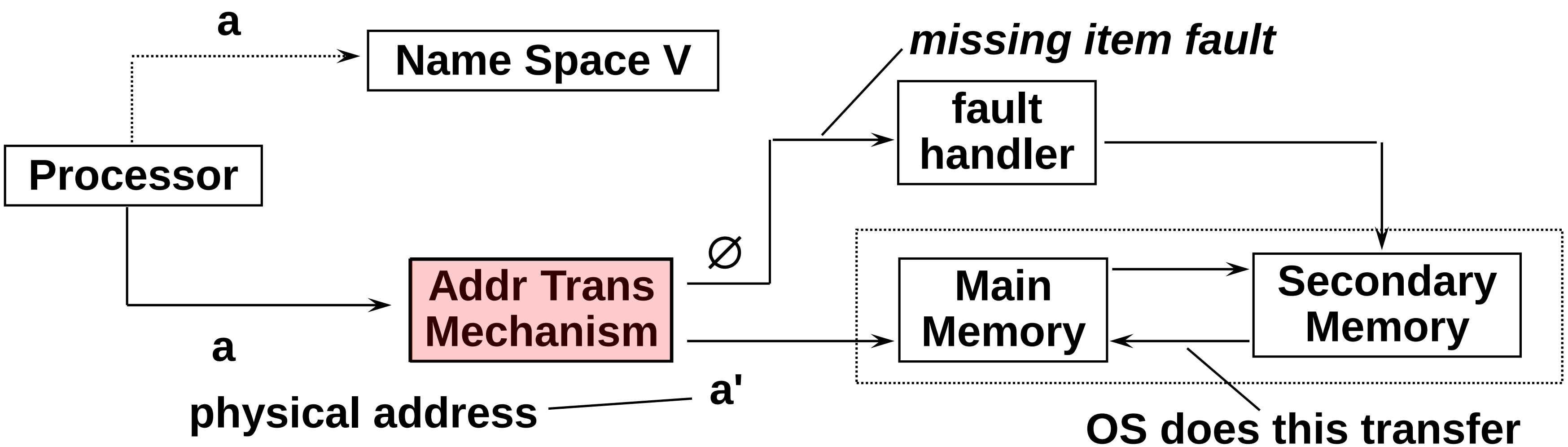
  ↖ *pages* ↖ *page frames (frames)*

  partitioned into blocks of equal size
- Key operation: address mapping

  MAP: V → M ∪ {∅} address mapping function

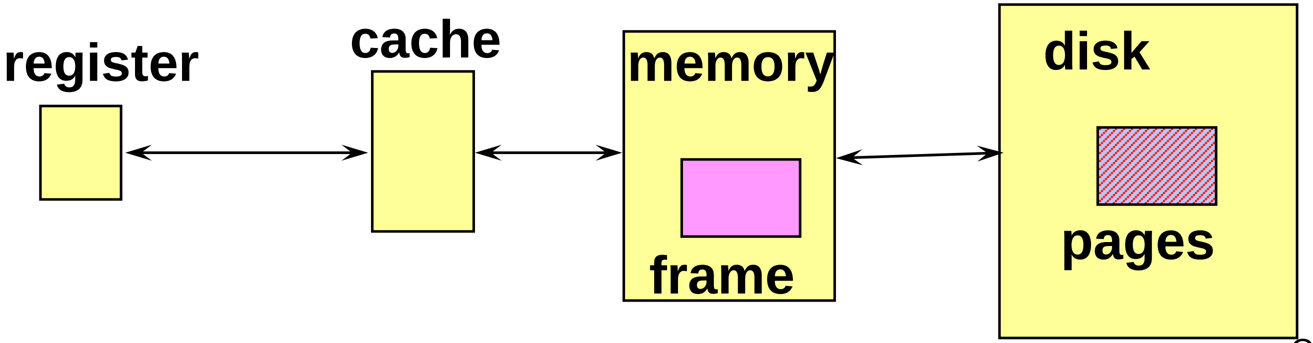
  MAP(a) = a' if data at virtual address a is present in physical address a' and a' in M

  = ∅ if data at virtual address a is not present in M



Basic Issues in Virtual Memory

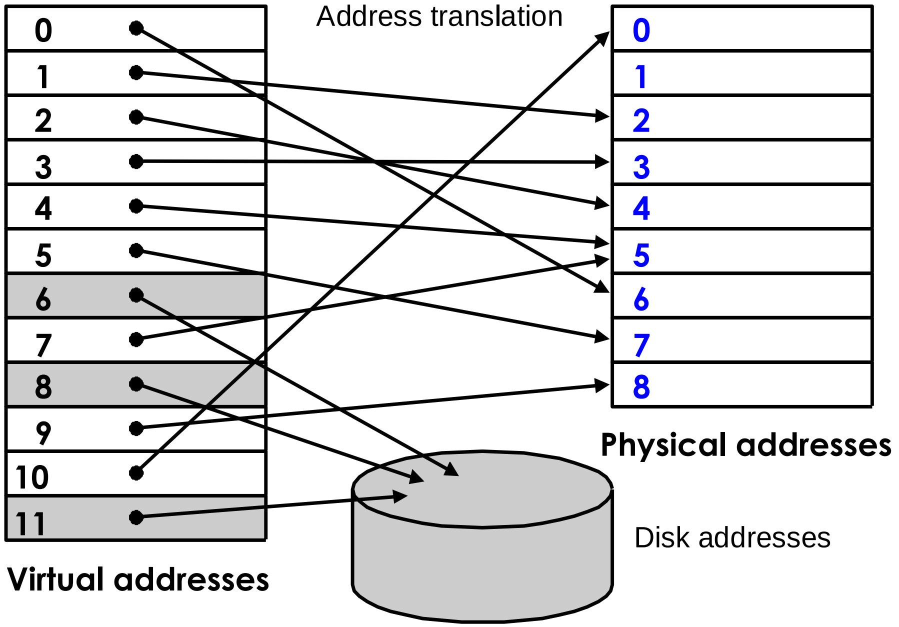
- Size of data blocks that are transferred from disk to main memory
- Which region of memory to hold new block → placement policy
- When memory is full, then some region of memory must be released to make room for the new block → replacement policy
- When to fetch missing items from disk?
  - Fetch only on a fault → demand load policy



Block Size and Placement Policy

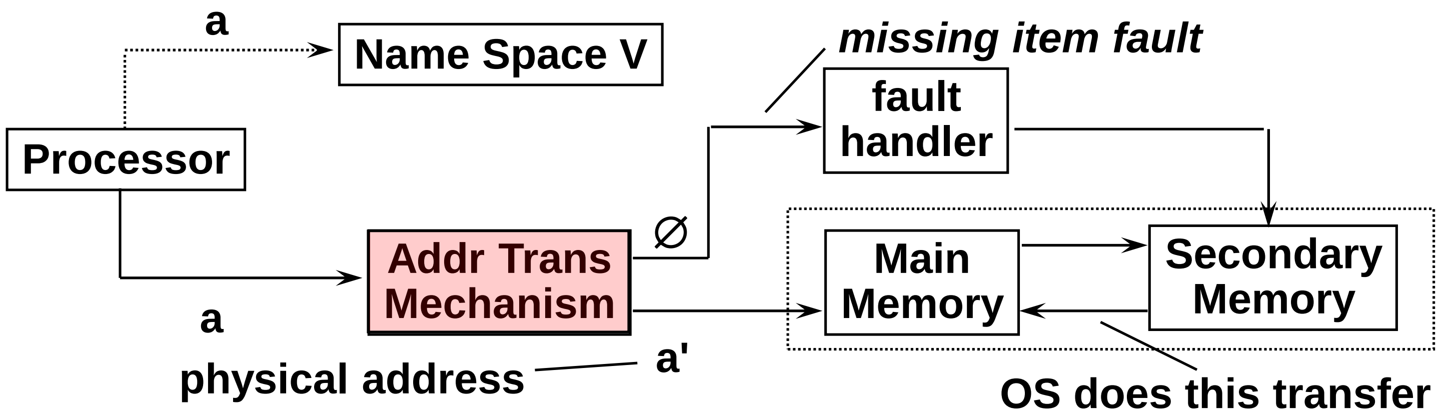
- Huge miss penalty: a page fault may take millions of cycles to process
  - Pages should be fairly large (ex: 4KB) to amortize the high access time
  - Reducing page faults is important
    - fully associative placement
→ use page table (in memory) to locate pages

Address Translation Mechanism



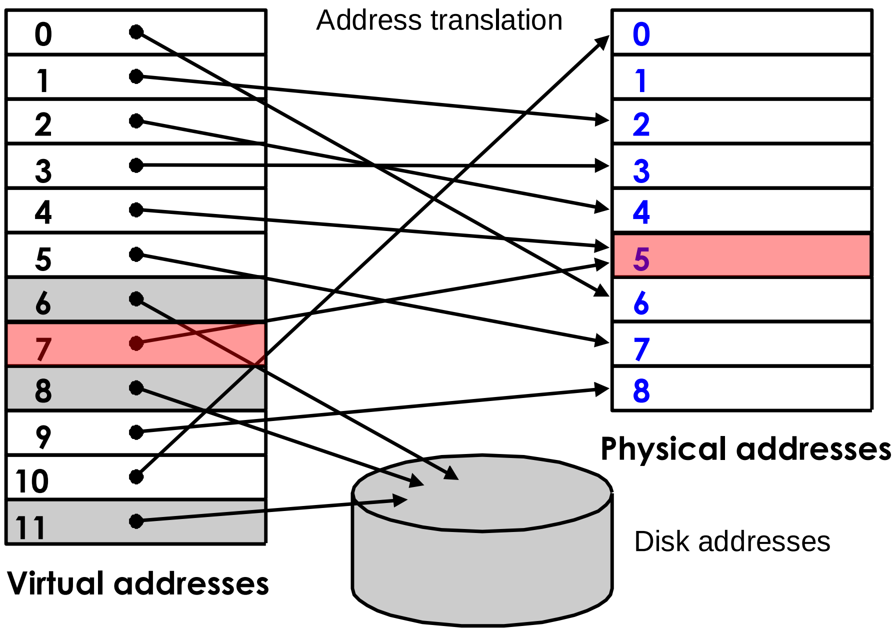
Use lookup table for translation

Virtual	Physical
0	6
1	2
2	4
3	3
4	5
5	7
6	-
7	5
8	-
9	8
10	0
11	-



Address Translation Example

- Virtual memory size: 2^{20} bytes
- Page size: 2^{10} bytes
- Virtual address: 7414
  - Page table index (page number): 7414/1024 = 7
  - Page offset: 7414 % 1024 = 246
  - Physical address: 5×1024 + 246 = 5366



Virtual	Physical
0	6
1	2
2	4
3	3
4	5
5	7
6	-
7	5
8	-
9	8
10	0
11	-

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

Page Fault: What Happens When You Miss?

- Page fault means that page is not resident in memory
- Huge miss penalty: a page fault may take millions of cycles to process
- Hardware must detect situation but it cannot remedy the situation
- Can handle the faults in software instead of hardware, because handling time is small compared to disk access
  - the software can be very smart or complex
    - reducing the page fault rate
  - the faulting process can be context-switched
    - keep processor working

Handling Page Faults

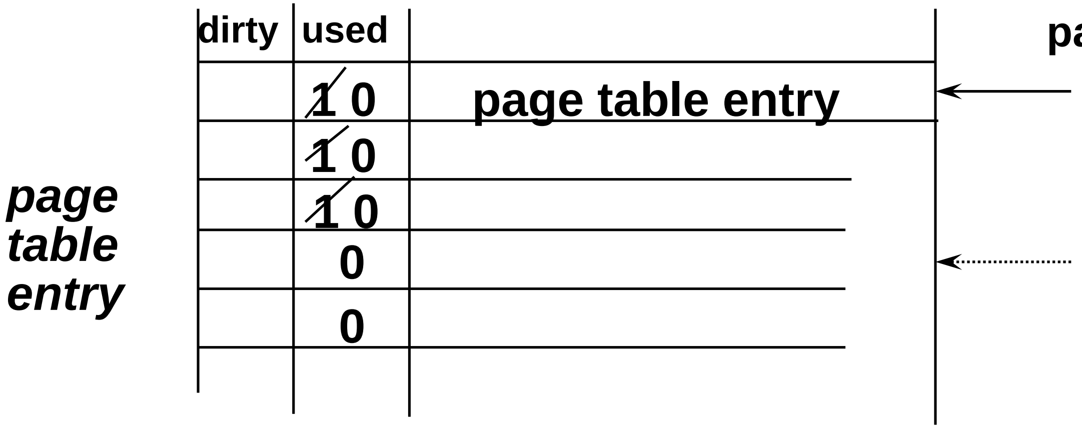
- Hardware must trap to the operating system so that it can remedy the situation
  - Pick a page to discard (may write it to disk)
  - Load the page in from disk
  - Update the page table
  - Resume to program so HW will retry and succeed!
- OS must know where to find the page
  - Create space on disk for all pages of process (swap space)
  - Use a data structure to record where each valid page is on disk (may be part of page table)
  - Use another data structure to track which process and virtual addresses use each physical page → for replacement purpose

Page 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

Page Replacement: 1-bit LRU

- Associated with each page is a *reference flag*:
  ref flag = 1 if page has been referenced in recent past
  = 0 otherwise
- If replacement is necessary, choose any page frame such that its reference bit is 0. This is a page that has not been referenced in the recent past



page table entry

dirty	used	
	~~1~~ 0	page table entry
	~~1~~ 0	
	~~1~~ 0	
	0	
	0	

page fault handler:

last replaced pointer (lrp)

If replacement is to take place, advance lrp to next entry (mod table size) until one with a 0 bit is found; this is the target for replacement; As a side effect, all examined PTE's have their reference bits set to zero.

Or search for a page that is both not recently referenced AND not dirty

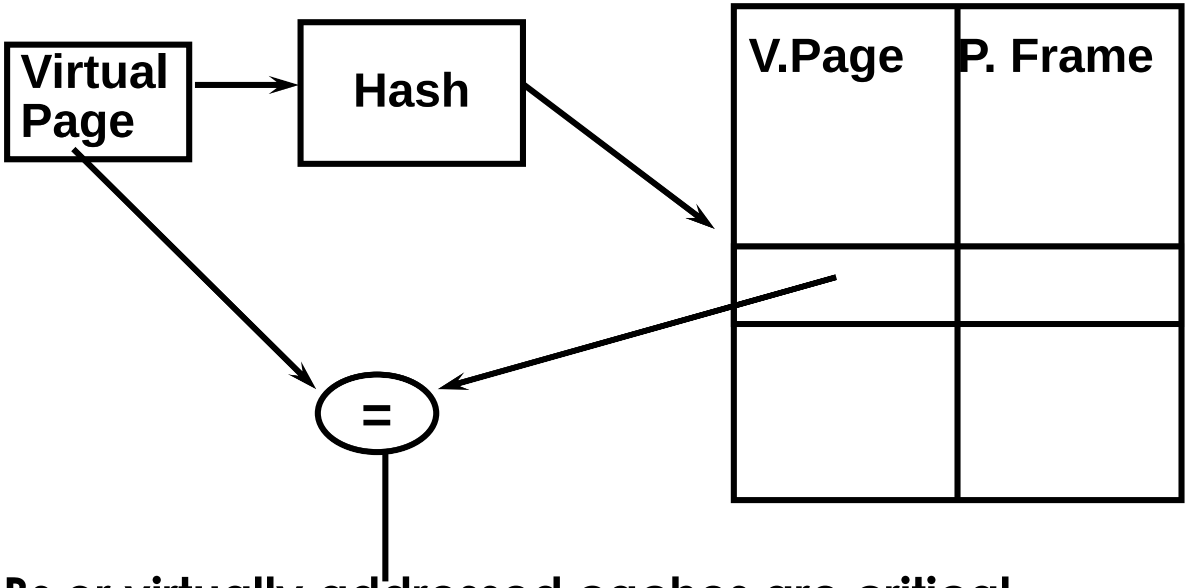
Architecture part: support dirty and used bits in the page table (how?)
→ may need to update PTE on any instruction fetch, load, store

Impact of Paging - Huge Page Table

- Page table occupies storage
  32-bit VA, 4KB page size, 4bytes/entry
  → 2^{20} PTE, 4MB table
- Possible solutions:
  - Use bounds register to limit table size; add more if exceed
  - Let pages to grow in both directions
    → 2 tables, 2 limit registers (one for heap, one for stack)
  - Use hashing → page table same size as physical pages
  - Multiple levels of page tables
  - Paged page table (page table resides in virtual space)

Hashing: Inverted Page Tables

- 28-bit virtual address
- 4 KB per page, and 4 bytes per page-table entry
  - Page table size : 2^{16} (pages) x 4 = 256 KB (per process)
  - Inverted page table :
    - Let the # of physical frames (64 MB) = 2^{14} (frames)
    - 2^{14} x 4 = 64 KB (whole system)

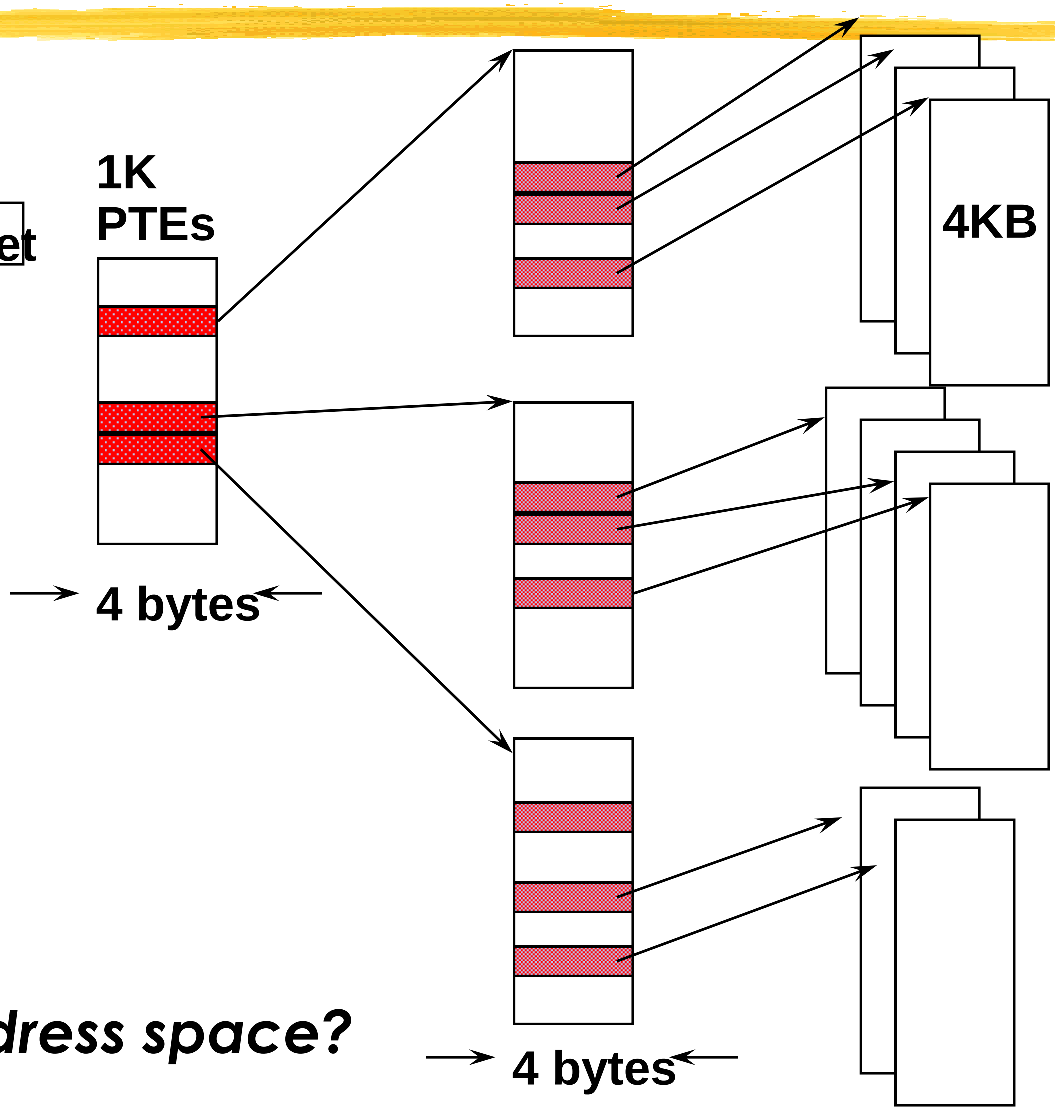


→ TLBs or virtually addressed caches are critical

Two-level Page Tables

32-bit address:

10	10	12
P1 index	P2 index	page offset



4 GB virtual address space

4 KB of PTE1

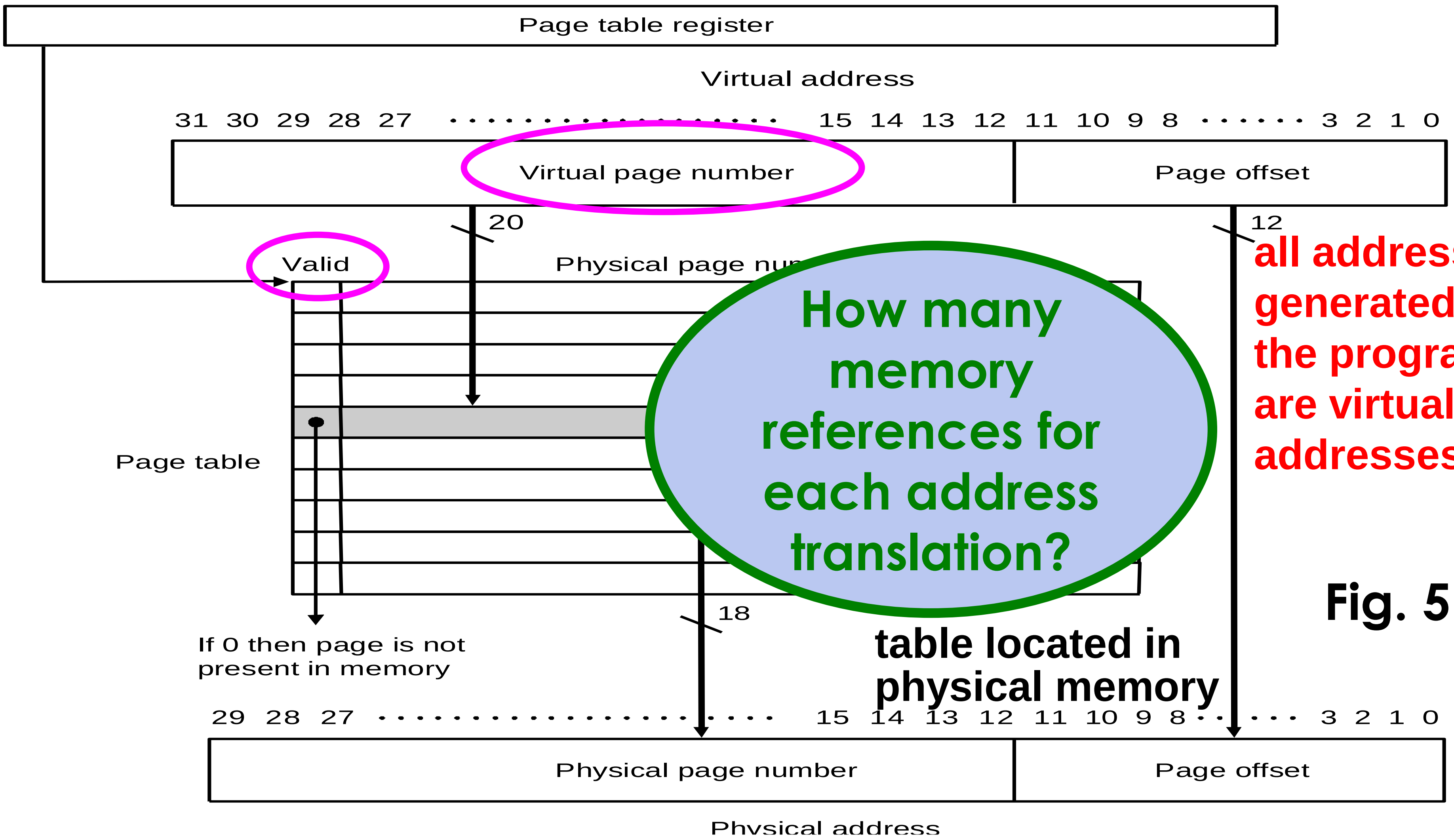
(Each entry indicate if any page in the segment is allocated)

4 MB of PTE2

paged, holes

What about a 48-64 bit address space?

Page Tables



How many memory references for each address translation?

all addresses generated by the program are virtual addresses

table located in physical memory

Fig. 5.21

Impact of Paging – More Memory Access!

- Each memory operation (instruction fetch, load, store) requires a page-table access!
  - Basically double number of memory operations
  - One to access the PTE
  - Then the actual memory access
- access to page tables has good locality

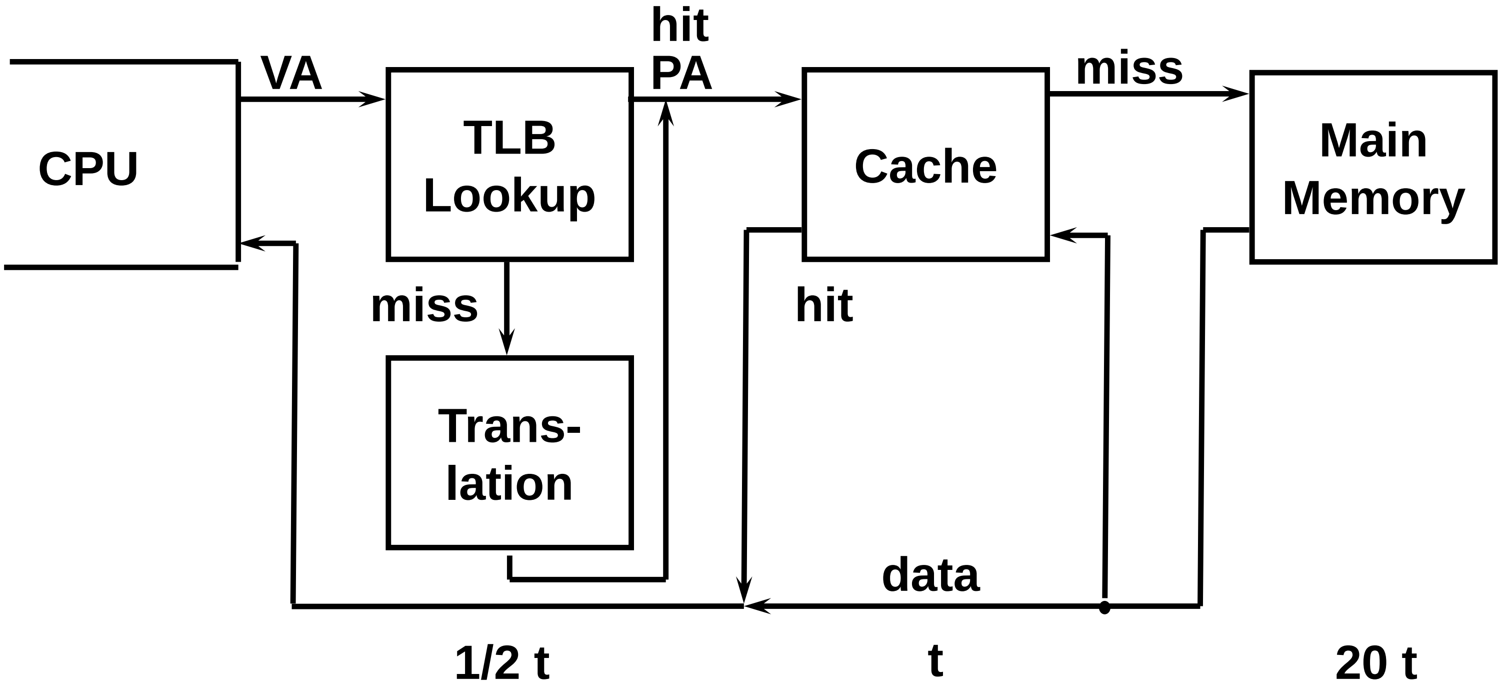
Can we make it faster?

Use Cache!

Making Address Translation Practical

- In VM, memory acts like a cache for disk
  - Page table maps virtual page numbers to physical frames
  - Use a page table cache for recent translation → *Translation Lookaside Buffer* (TLB)

Translation with a TLB



Fast Translation Using a TLB

- Access to page tables has good locality
  - Fast cache of PTEs within the CPU
  - Called a Translation Look-aside Buffer (TLB)

Translation Lookaside Buffer

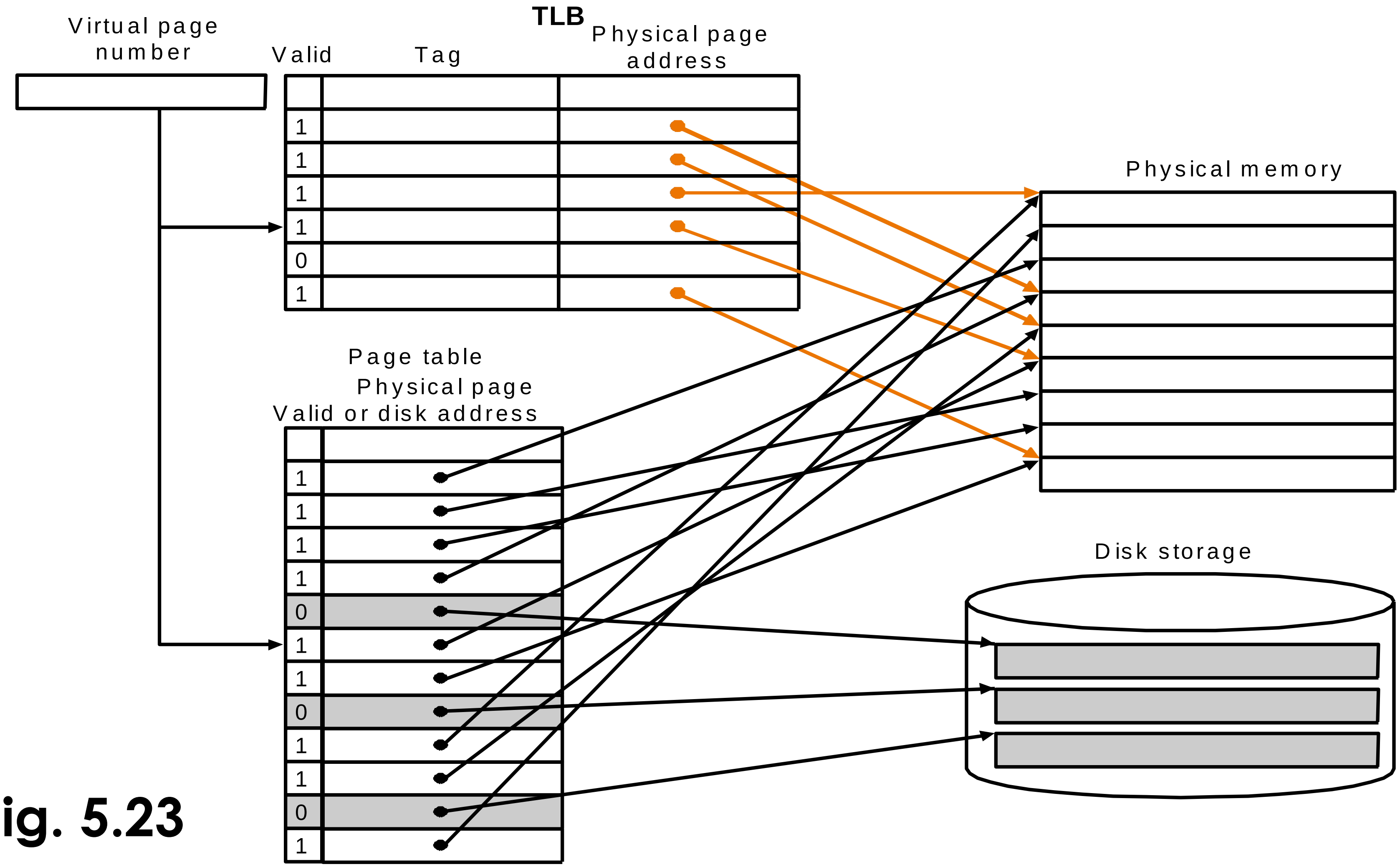


Fig. 5.23

Translation Lookaside Buffer

- Typical RISC processors have *memory management unit* (MMU) which includes TLB and does page table lookup
  - TLB can be organized as fully associative, set associative, or direct mapped
  - TLBs are small, 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

TLB Hit or Miss

- TLB hit on read
- TLB hit on write:
  - Toggle dirty bit (write back to page table on replacement)
- TLB miss:
  - If only TLB miss => hardware or software
  - If page fault also => software

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 present
- 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

Integrating TLB and Cache

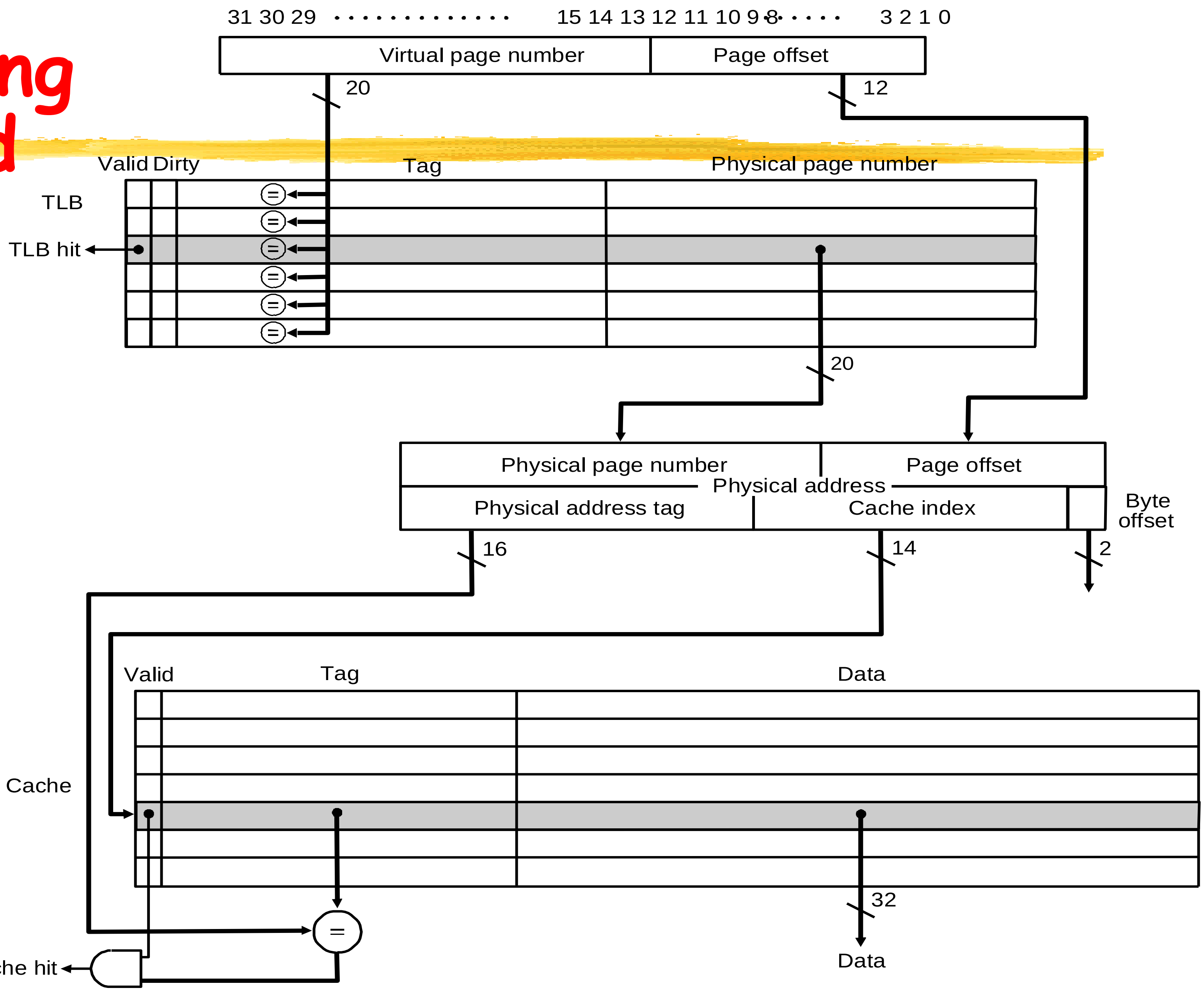
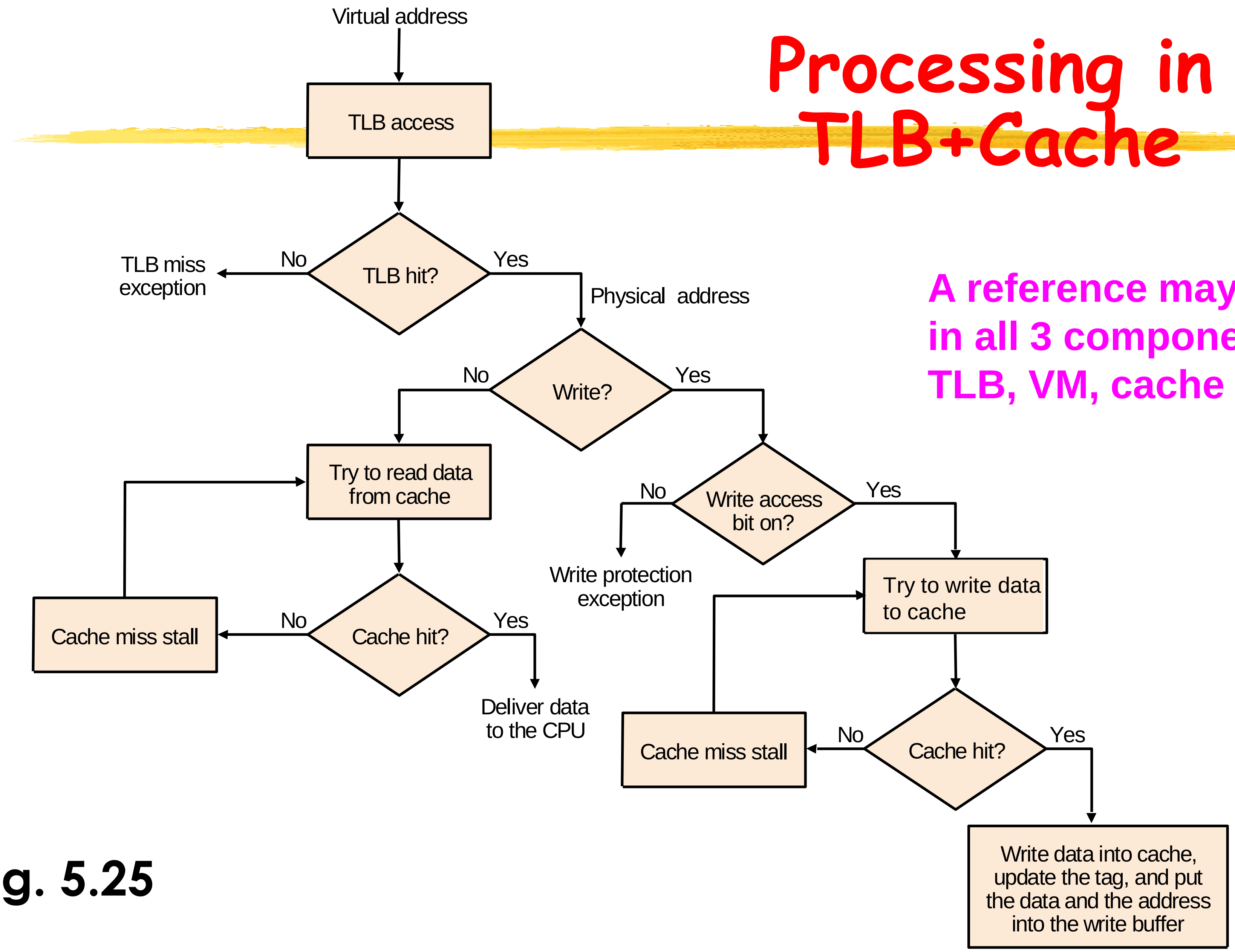


Fig. 5.24

Processing in TLB+Cache



A reference may miss in all 3 components: TLB, VM, cache

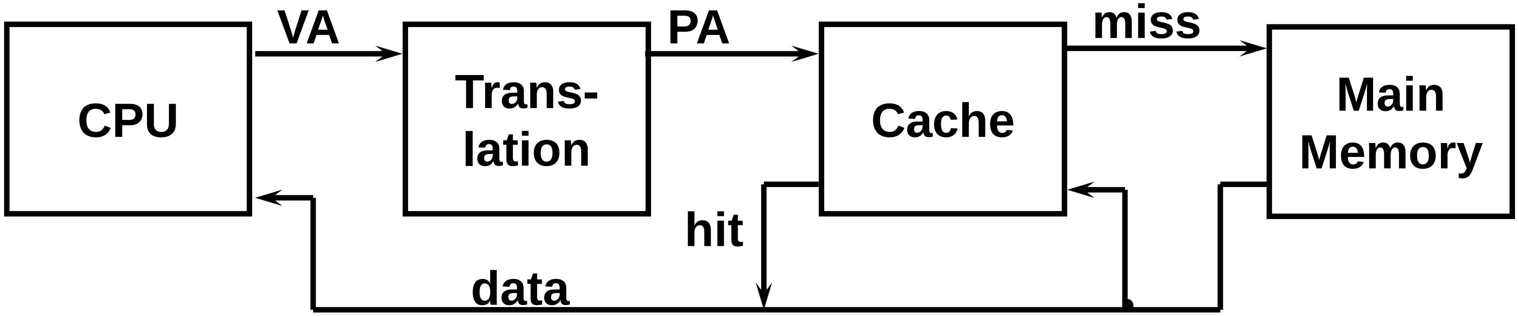
Fig. 5.25

Possible Combinations of Events

TLB	Page table	Cache	Possible? Conditions?
Hit	Hit	Miss	Yes; but page table never checked if TLB hits
Miss	Hit	Hit	TLB miss, but entry found in page table;after retry, data in cache
Miss	Hit	Miss	TLB miss, but entry found in page table; after retry, data miss in cache
Miss	Miss	Miss	TLB miss and is followed by a page fault; after retry, data miss in cache
Hit	Miss	Miss	impossible; not in TLB if page not in memory
Hit	Miss	Hit	impossible; not in TLB if page not in memory
Miss	Miss	Hit	impossible; not in cache if page not in memory

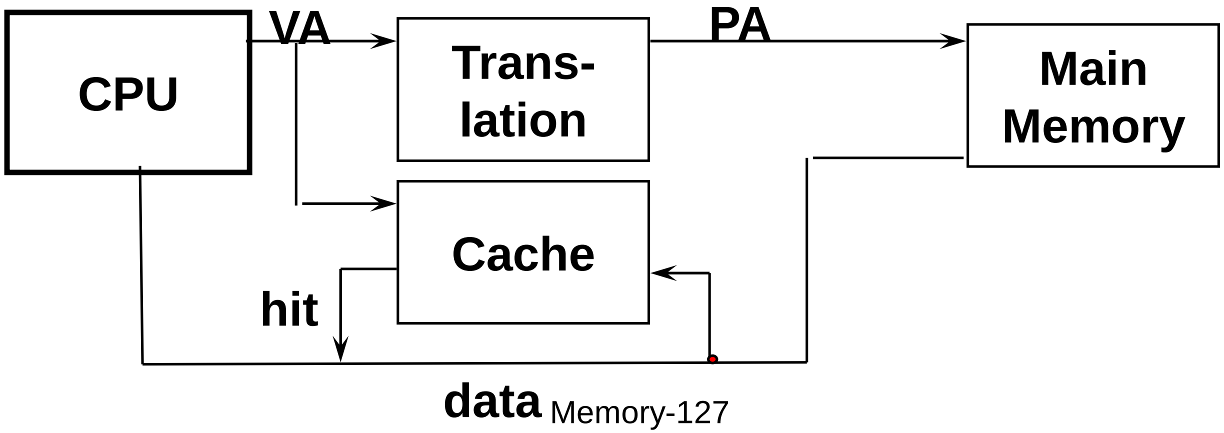
Virtual Address and Cache

- TLB access is serial with cache access
  - Cache is physically indexed and tagged



Can we make it faster?

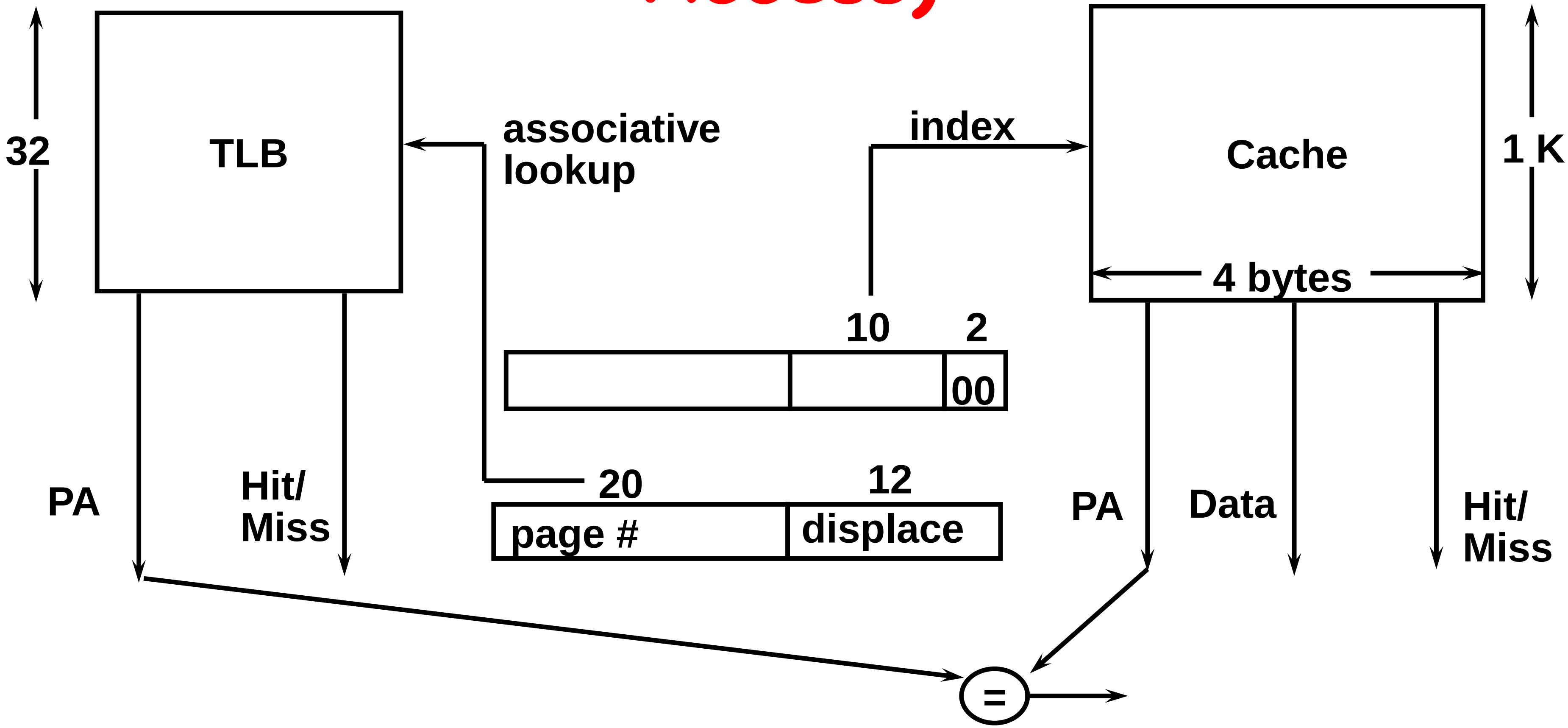
- Alternative: *virtually addressed cache*
  - Cache is virtually indexed and virtually tagged



Virtually Addressed Cache

- Advantage: Require address translation only on miss!
- Problem:
  - Same virtual addresses (different processes) map to different physical addresses: tag + process id
  - *Synonym/alias problem*: two different virtual addresses map to same physical address
    - Two different cache entries holding data for the same physical address!
  - For update: must update all cache entries with same physical address, otherwise memory becomes inconsistent
  - Determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits;
  - Or software enforced alias boundary: same least-significant bits of VA &PA > cache size

An Alternative:Virtually Indexed but Physically Tagged (Overlapped Access)



IF cache hit AND (cache tag == PA) then deliver data to CPU
ELSE IF [cache miss OR (cache tag ! = PA)] and TLB hit THEN
access memory with the PA from the TLB
ELSE do standard VA translation

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
  - 2 modes: kernel, user
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (ex: syscall in MIPS) : CPU from user to kernel

Outline

- Memory hierarchy
- The basics of caches
- Measuring and improving cache performance
- Virtual memory
- A common framework for memory hierarchy
- Using a finite state machine to control a simple cache
- Parallelism and memory hierarchies: cache coherence

The Memory Hierarchy

The BIG Picture

- 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 (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka 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

Challenge in Memory Hierarchy

- Every change that potentially improves miss rate can negatively affect overall performance

Design change	Effects on miss rate	Possible effects
size ↑	capacity miss ↓	access time ↑
associativity ↑	conflict miss ↓	access time ↑
block size ↑	spatial locality ↑	miss penalty ↑

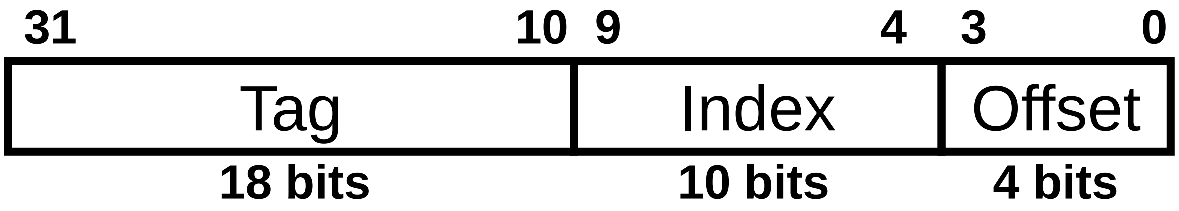
- Trends:
  - Synchronous SRAMs (provide a burst of data)
  - Redesign DRAM chips to provide higher bandwidth or processing
  - Restructure code to increase locality
  - Use prefetching (make cache visible to ISA)

Outline

- Memory hierarchy
- The basics of caches
- Measuring and improving cache performance
- Virtual memory
- A common framework for memory hierarchy
- Using a finite state machine to control a simple cache
- Parallelism and memory hierarchies: cache coherence

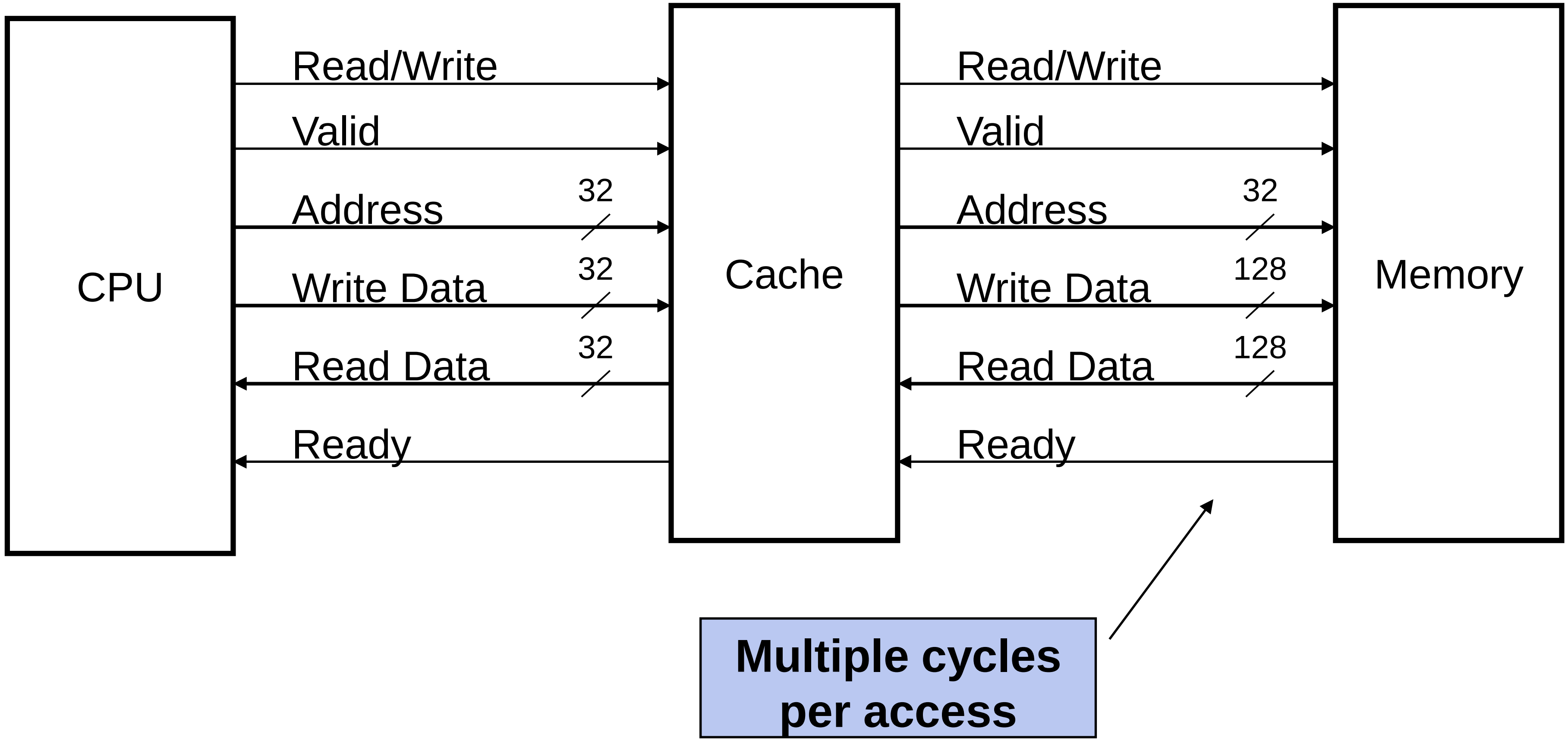
Cache Control

- 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



31 10	9 4	3 0
Tag	Index	Offset
18 bits	10 bits	4 bits

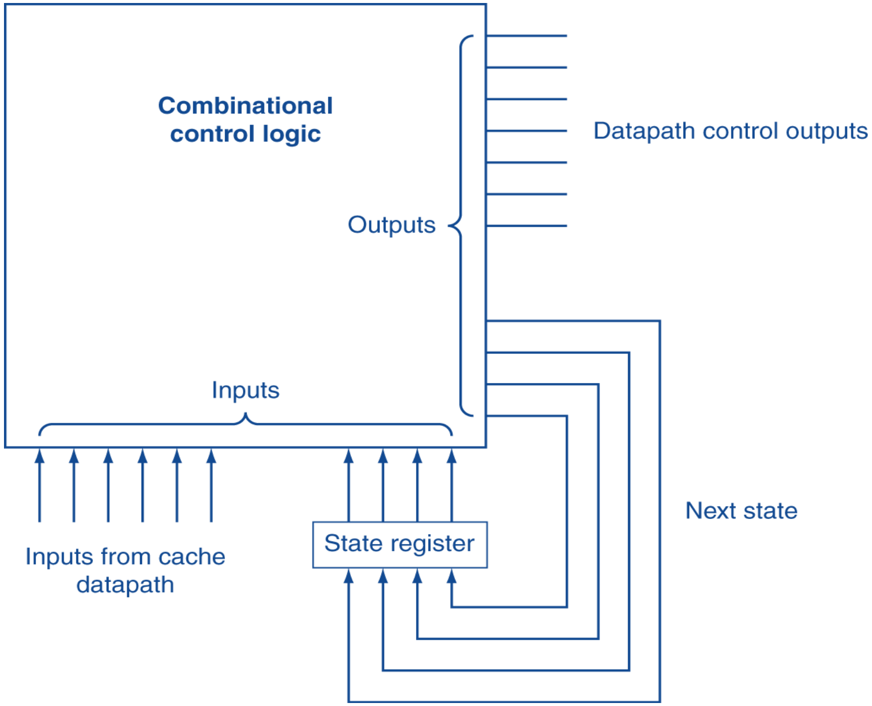
Interface Signals



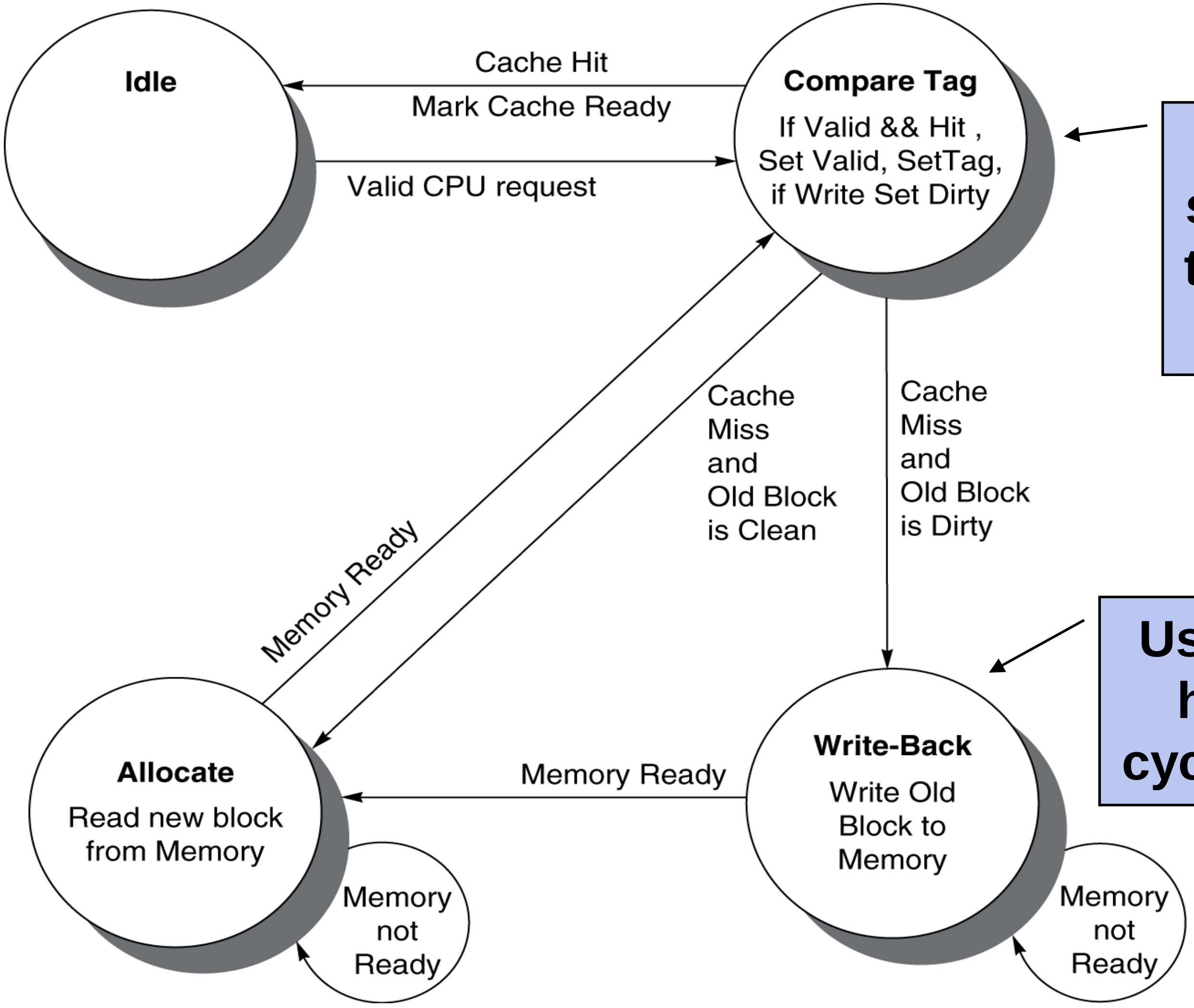
Multiple cycles per access

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



Partition into separate states to reduce clock cycle time

Use write buffer to hide write-back cycle for dirty block

Outline

- Memory hierarchy
- The basics of caches
- Measuring and improving cache performance
- Virtual memory
- A common framework for memory hierarchy
- Using a finite state machine to control a simple cache
- Parallelism and memory hierarchies: cache coherence

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)
    ⇒ read returns written value
  - P_1 writes X; P_2 reads X (sufficiently later)
    ⇒ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - P_1 writes X, P_2 writes X
    ⇒ 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

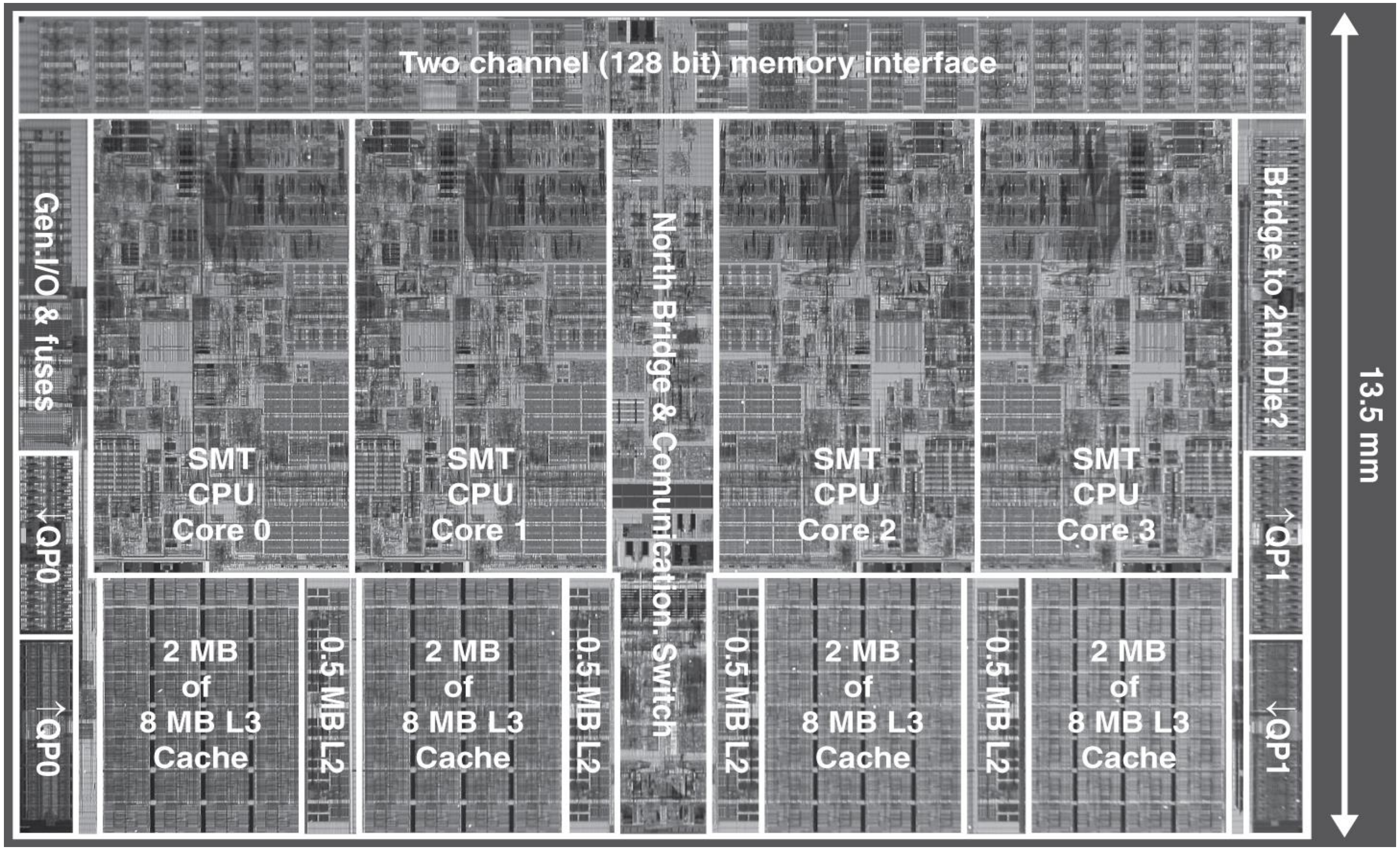
CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

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
    ⇒ 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 (731million transistors)



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache, 2-level TLB

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

Pitfalls

- Byte vs. word addressing
  - Example: 32-byte direct-mapped cache, 4-byte blocks
    - Byte 36 maps to block 1
    - Word 36 maps to block 4
- Ignoring memory system effects when writing or generating code
  - Example: iterating over rows vs. columns of arrays
  - Large strides result in poor locality

Pitfalls

- In multiprocessor with shared L2 or L3 cache
  - Less associativity than cores results in conflict misses
  - More cores ⇒ need to increase associativity
- Using AMAT (average memory access time) to evaluate performance of out-of-order processors
  - Ignores effect of non-blocked accesses
  - Instead, evaluate performance by simulation

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