

# Computer Architecture and Organization

---

INSTRUCTOR: YAN-TSUNG PENG

DEPT. OF COMPUTER SCIENCE, NCCU

# Memory Technology

# Memory Technology

Year introduced	Chip size	\$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	\$1,500,000	250 ns	150 ns
1983	256 Kibibit	\$500,000	185 ns	100 ns
1985	1 Mebibit	\$200,000	135 ns	40 ns
1989	4 Mebibit	\$50,000	110 ns	40 ns
1992	16 Mebibit	\$15,000	90 ns	30 ns
1996	64 Mebibit	\$10,000	60 ns	12 ns
1998	128 Mebibit	\$4,000	60 ns	10 ns
2000	256 Mebibit	\$1,000	55 ns	7 ns
2004	512 Mebibit	\$250	50 ns	5 ns
2007	1 Gibibit	\$50	45 ns	1.25 ns
2010	2 Gibibit	\$30	40 ns	1 ns
2012	4 Gibibit	\$1	35 ns	0.8 ns

# Types of Memory

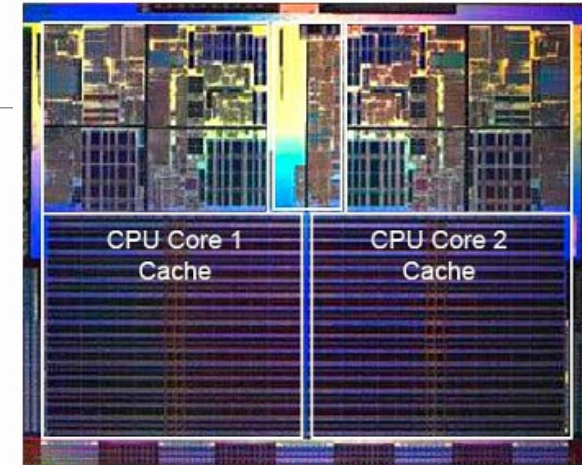
- SRAM (Static random access)
  - Cache
- DRAM (Dynamic random access)
  - Main memory
- Flash
  - Mostly used in a cellphone as secondary memory
- Disk
  - Primarily used in a PC as secondary memory

Memory technology	Typical access time	\$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	\$500–\$1000
DRAM semiconductor memory	50–70 ns	\$10–\$20
Flash semiconductor memory	5,000–50,000 ns	\$0.75–\$1.00
Magnetic disk	5,000,000–20,000,000 ns	\$0.05–\$0.10

# Memory Technology

## ■ **SRAM:** *Static Random Access Memory*

- Low density (small size), expensive, fast
- Typically integrated into the CPU chip
- Static: content will last (forever until lose power)
- It requires four to six transistors per bit.
- Memory array with (usually) a single access port
- Static: data stored in a cell and kept on a pair of inverting gates (D-latch) as long as power is applied
- Usually used in Caches



Random Access means you can access any random location at any time and the access time will be the same. (not the case for disks)

# Memory Technology

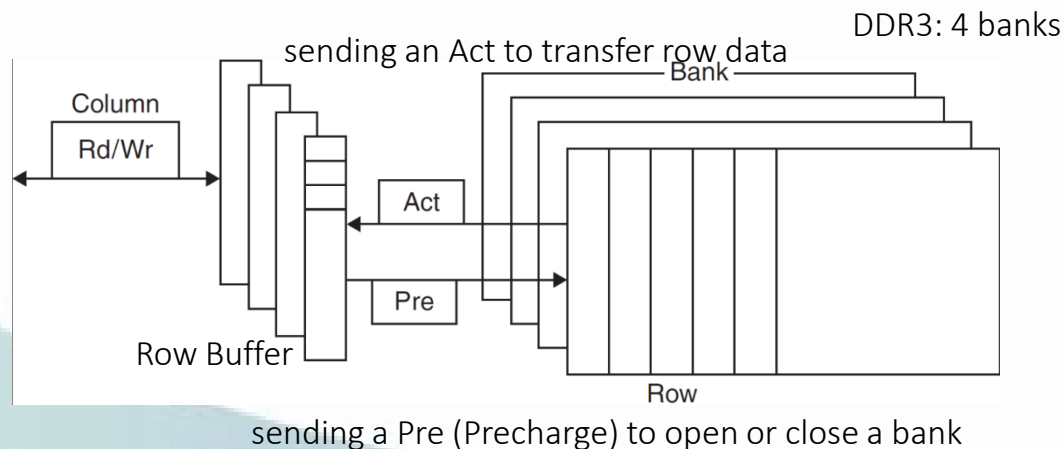
---

## ■ **DRAM:** *Dynamic Random Access Memory*

- High density, cheap, slow
- It uses only a single transistor per bit
- Dynamic: data kept in a cell is stored as a charge in a capacitor, which needs to be refreshed periodically
- Two-level decoding, consisting of a row access followed by a column access (Row/Column Access Strobe)
- Mainly used in main memory

# DRAM Technology

- Data stored as a charge in a capacitor
- Single transistor used to access the charge
  - Must periodically be refreshed
    - Read contents and write them back
    - Address is split into row and column addresses
    - Performed on a “row” first to activate row buffer and then use the column address to access the data



# Flash

---

- A type of EEPROM
  - Electrically erasable programmable read-only memory
- Nonvolatile semiconductor storage
  - 100x – 1000x faster than disk
  - Smaller, lower power, more robust
  - But more \$/GB (between disk and DRAM)

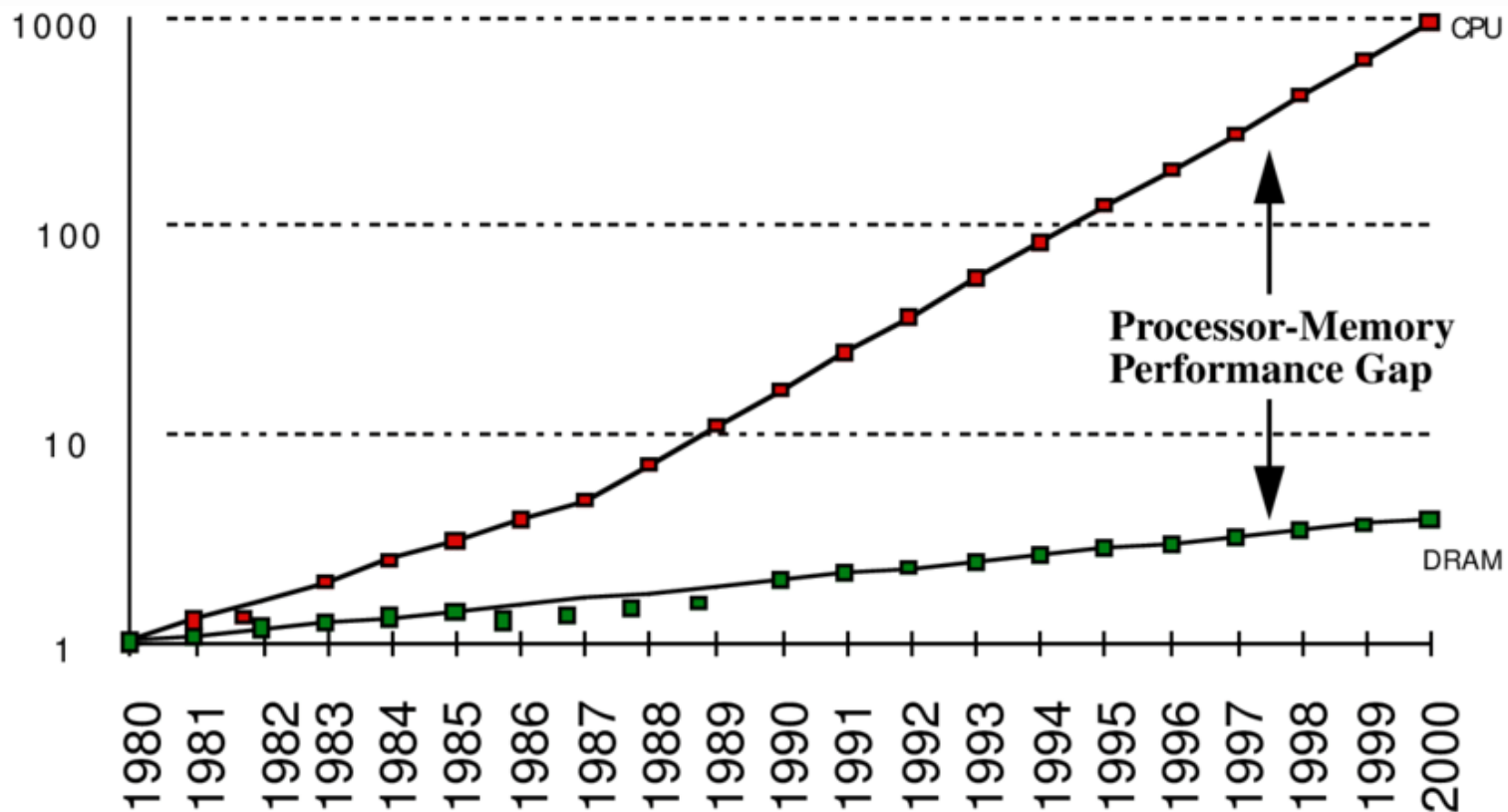


# Flash Types

---

- NOR flash: bit cell like a NOR gate
  - Random read/write access
  - Used for instruction memory in embedded systems
- NAND flash: bit cell like a NAND gate
  - Cheaper per GB
  - Used for USB keys, media storage, ...
- Flash bits wears out after 1000's of accesses
  - Wear levelling: remapping frequently written blocks to less trodden blocks

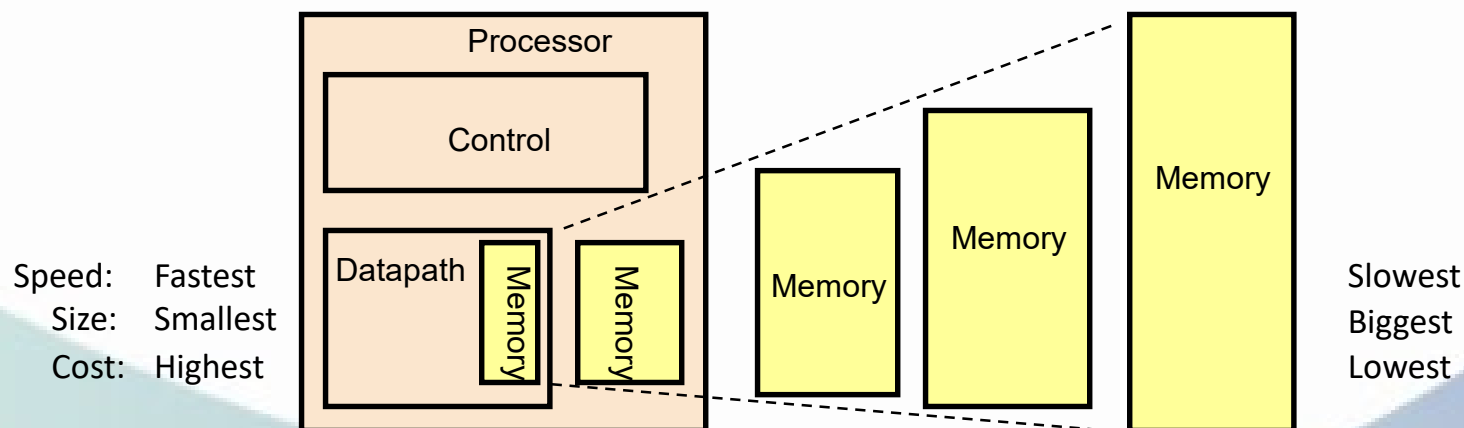
# Processor-Memory Performance Gap

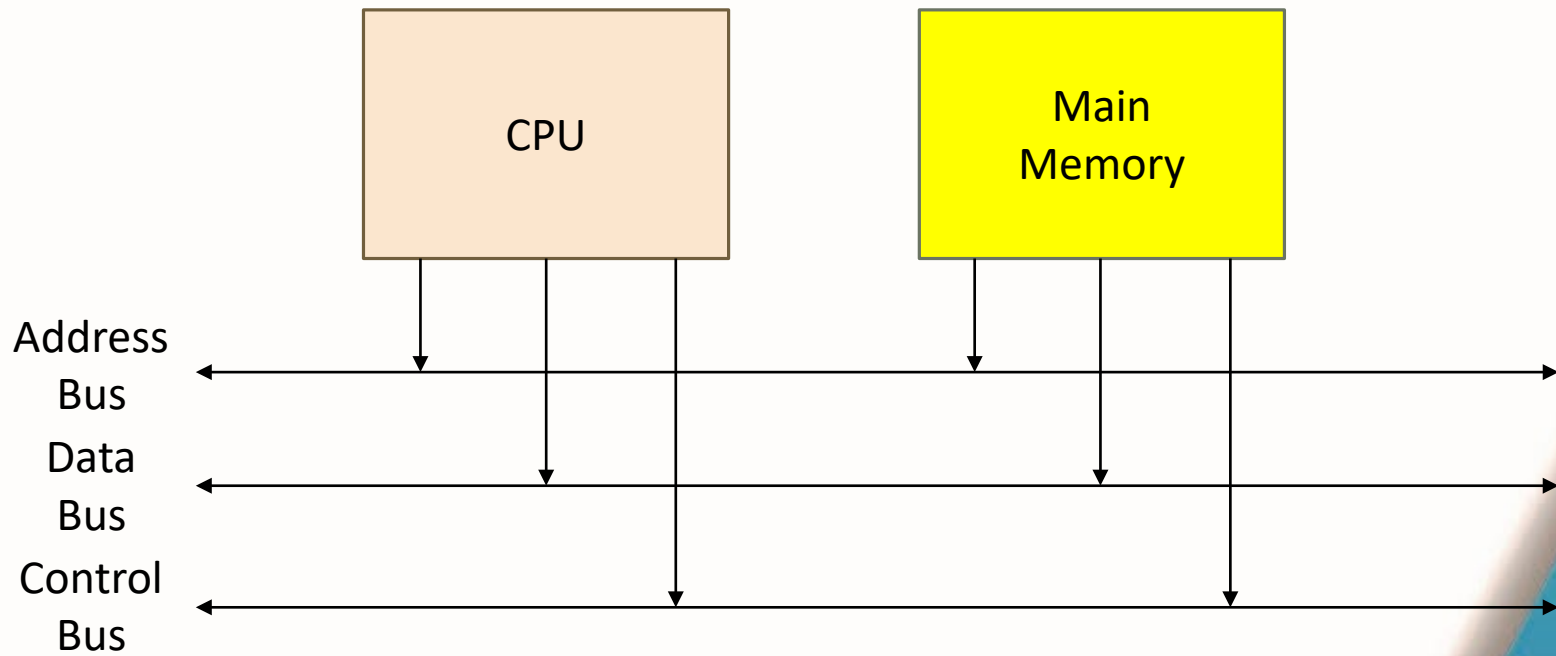


Hennessy, J.L.; Patterson, D.A. Computer Organization and Design, 2nd ed. San Francisco: Morgan Kaufmann Publishers, 1997.

# Memory Hierarchy

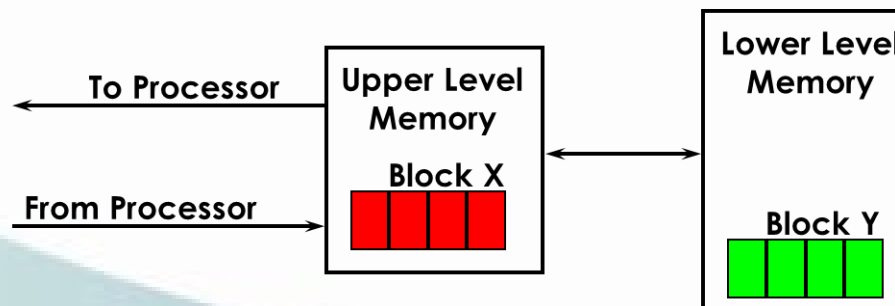
- Large memories slow, fast memories small
- Efficient Access can be achieved: hierarchy, parallelism
- Memory Hierarchy:

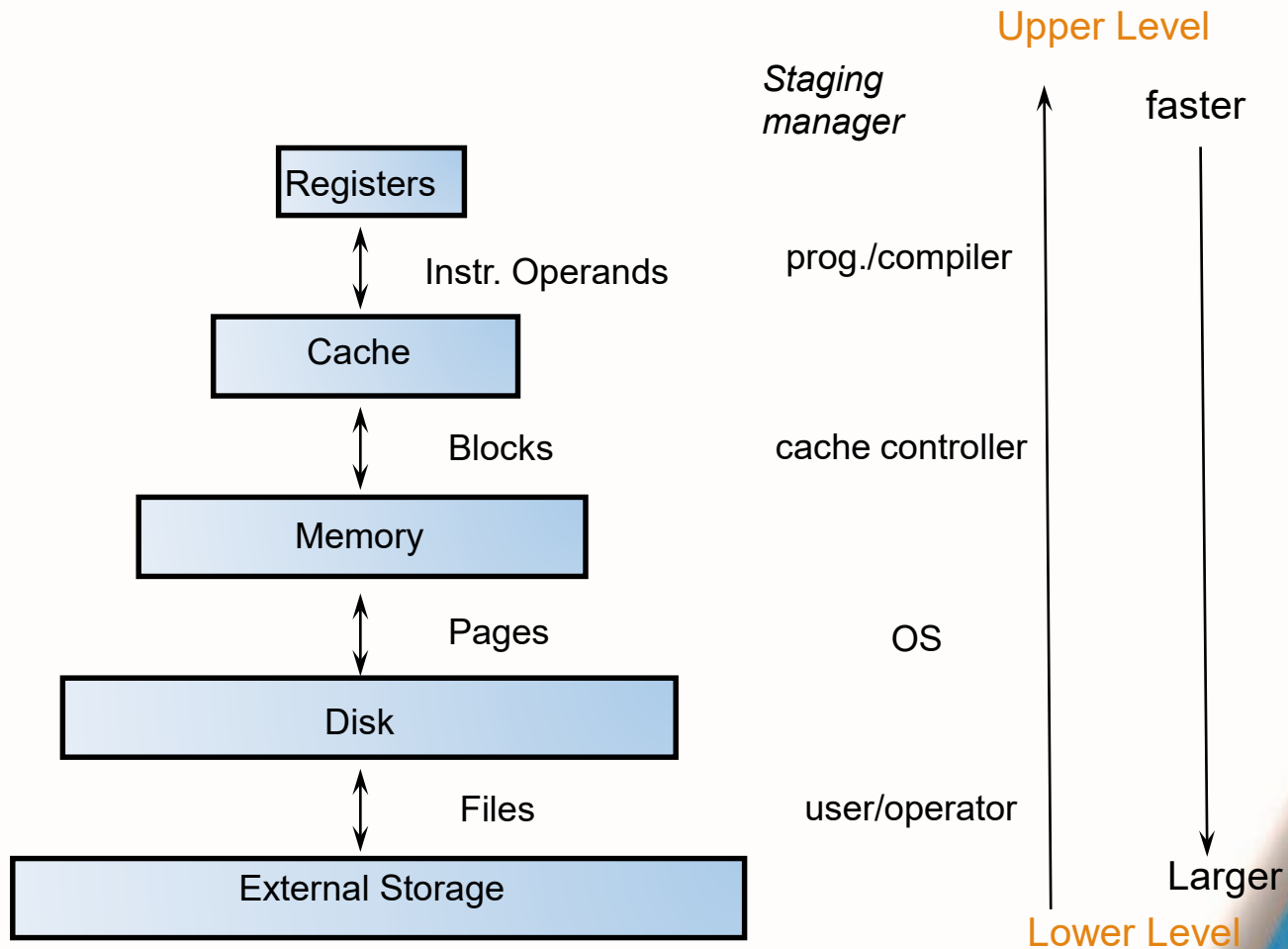




# Data Moving in Memory Hierarchy

- 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





# Principle of Locality

---

- Programs access a small proportion of their address space at any time
- Most programs use 90% of time to execute a small piece of code
- Two types of localities: Temporal and Spatial
- Accessing data in the cache is much more quickly than that not in the cache since it will need to be retrieved from main memory, which is slower.
- Temporal locality
  - If an item is referenced recently, it is likely to be accessed again soon.
  - e.g., instructions in a loop, induction variables (ex: `i++`)
- Spatial locality
  - If data that is physically close to another piece of data is more likely to be accessed soon.
  - e.g., sequential instruction access, array data

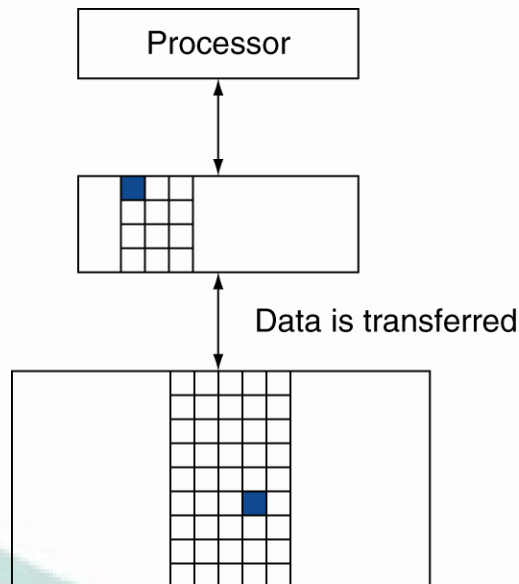
# Taking Advantage of Locality

---

- Memory hierarchy
  - Structure that leverages multiple levels of memories
  - As the distances from the CPU increases, the size of memories and the access time both increase
- Store everything on disk (the lowest level has the largest capacity)
- 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

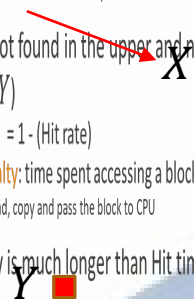


# Memory Hierarchy Levels



- If accessed data are present in upper level
  - Hit: access satisfied by upper level
    - Hit rate: hits/accesses
- If accessed data are absent in upper level, it will be fetched from lower level
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss rate: misses/accesses  
 $= 1 - \text{hit rate}$

# Terminology

- Hit: data found in the upper level ( $X$ )
    - Hit rate: possibility of data found in the upper level
    - Hit time: time spent accessing data in the upper level memory
      - Include the time to determine hit/miss
  - Miss: data not found in the upper and needs to be loaded from the lower level ( $Y$ )
    - Miss Rate =  $1 - (\text{Hit rate})$
    - Miss Penalty: time spent accessing a block in the lower level
      - Includes find, copy and pass the block to CPU
  - Miss Penalty is much longer than Hit time
- Hit: data found in the upper level ( $X$ )
    - Hit rate: possibility of data found in the upper level
    - Hit time: time spent accessing data in the upper level memory
      - Include the time to determine hit/miss
  - Miss: data not found in the upper and needs to be loaded from the lower level ( $Y$ )
    - Miss Rate =  $1 - (\text{Hit rate})$
    - Miss Penalty: time spent accessing a block in the lower level
      - Includes find, copy and pass the block to CPU
  - Miss Penalty is much longer than Hit time

# Cache Basics

# Cache Memory

## ■ Cache memory

- The level of the memory hierarchy closest to the CPU

## ■ Given accesses $X_1, \dots, X_{n-1}, X_n$

a cache miss of  $X_n$

$X_4$
$X_1$
$X_{n-2}$
$X_{n-1}$
$X_2$
$X_3$

a. Before the reference to  $X_n$

$X_n$  is loaded into the cache

$X_4$
$X_1$
$X_{n-2}$
$X_{n-1}$
$X_2$
$X_n$
$X_3$

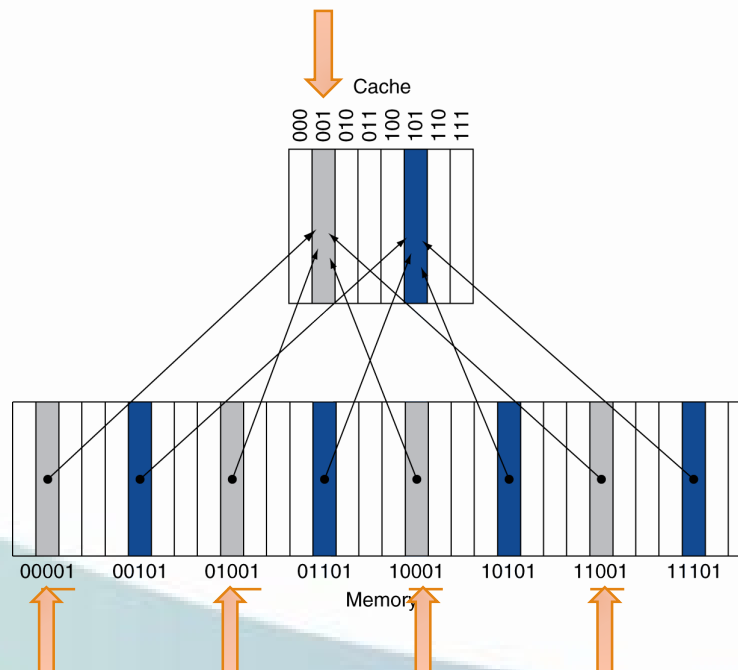
b. After the reference to  $X_n$

## ■ Questions:

- Where to place the loaded data? *block placement*
- How to locate specific data? *block finding*
- How to deal with a miss? *block replacement*
- How to deal with a write? *block write*

# Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - $(\text{Block address}) \bmod (\text{\#Blocks in cache})$



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

# Example: 8-block cache

Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Mem Addr (5-bit)	Decimal
00001	1
01001	9=8+1
10001	17=16+1
11001	25=24+1
Mem Addr (5-bit)	Decimal
00101	5
01101	13=8+5
10101	21=16+5
11101	29=24+5

# Tags and Valid Bits

❑ How do we know which particular block is stored in a cache location?

❑ Only need the high-order bits: Tag

❑ What if there is no data in a location?

❑ Valid bit: 1 = present, 0 = not present

❑ Initially, the valid bit is 0 (no data)

Mem Addr (5-bit)	Tag	Cache Address (3-bit)
01000	01	000
10001	10	001

Index	Valid?	Tag	Data
000	1	01	(32-bit word..)
001	1	10	(32-bit word..)
010	0		
011	..	..	...

Tag+Index=Address

logically



Address	Data
01000	(32-bit word..)
10001	(32-bit word..)

# 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

■ 8-blocks, 1 word/block, direct mapped

■ Initial state

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		
111	N		

# Cache Example

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

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		
<b>110</b>	<b>Y</b>	<b>10</b>	<b>Mem[10110]</b>
111	N		

# Cache Example

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

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

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

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

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

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

# Cache Example

■ 8-blocks, 1 word/block, direct mapped

■ Initial state

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		
<b>010</b>	<b>Y</b>	<b>11</b>	<b>Mem[11010]</b>
011	N		
100	N		
101	N		
<b>110</b>	<b>Y</b>	<b>10</b>	<b>Mem[10110]</b>
111	N		

# Cache Example

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

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

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

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

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

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

# Cache Example

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

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Hit	000

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



# Cache Example

■ 8-blocks, 1 word/block, direct mapped

■ Initial state

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	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

# Cache Example

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

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

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

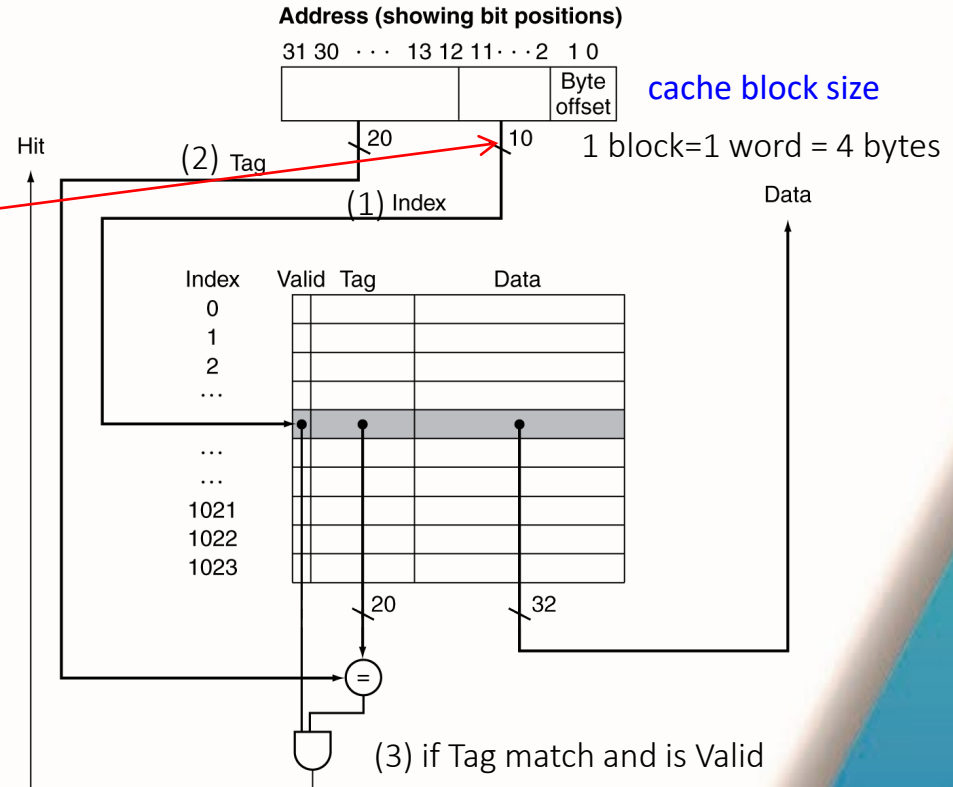
# Address Subdivision

A block can store 1 word => 2 bits

The cache has 1024 blocks in total => 10 bits

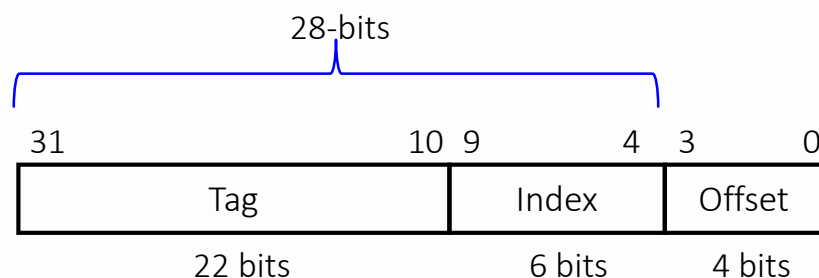
How many bits does the cache need?

$$[1 \text{ (valid bit)} + 20 \text{ (tag)} + 32 \text{ (data)}] * 1024 \text{ (block)}$$



# Larger Block Size

- If the cache has 64 blocks, each block has 16 bytes (4 words)



$$2^6 = 64$$

Offset: 16 bytes =  $2^4 \Rightarrow 4$  bits

Index: use 6 bits to address 64 blocks

Tags:  $32 - 6 - 4 = 22$  bits

$$[1 \text{ (valid bit)} + 22 \text{ (tag)} + 32 * 4 \text{ (data)}] * 64 \text{ (block)}$$

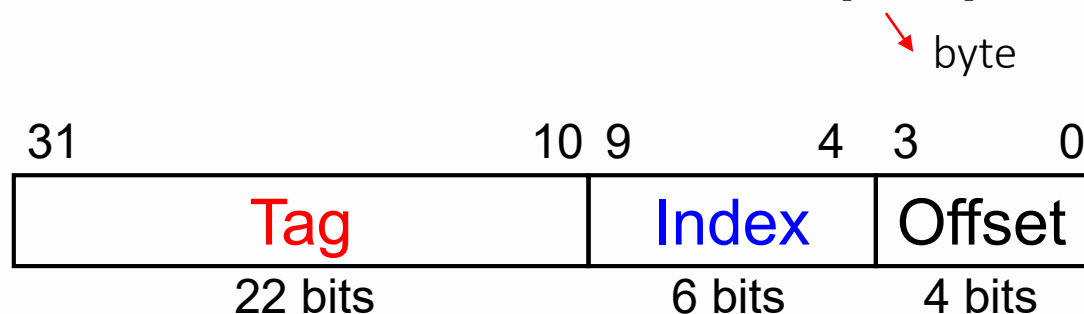
Byte offset: 2 bits

Block offset: 2 bits  
(used to index the block word)

# Example: Map Address to Block

If the cache has 64 blocks, each block has 16 bytes (4 words)

What is the block number for MEM[1200]?



Since each block takes 16 bytes, MEM[1200] would reside in  $\lfloor 1200/16 \rfloor = 75_{th}$  block

But we only have 64 blocks, thus **Index:  $75 \% 64 = 11$**

$$1200_{10} = \mathbf{1001011}0000_2$$

# Larger Block Size

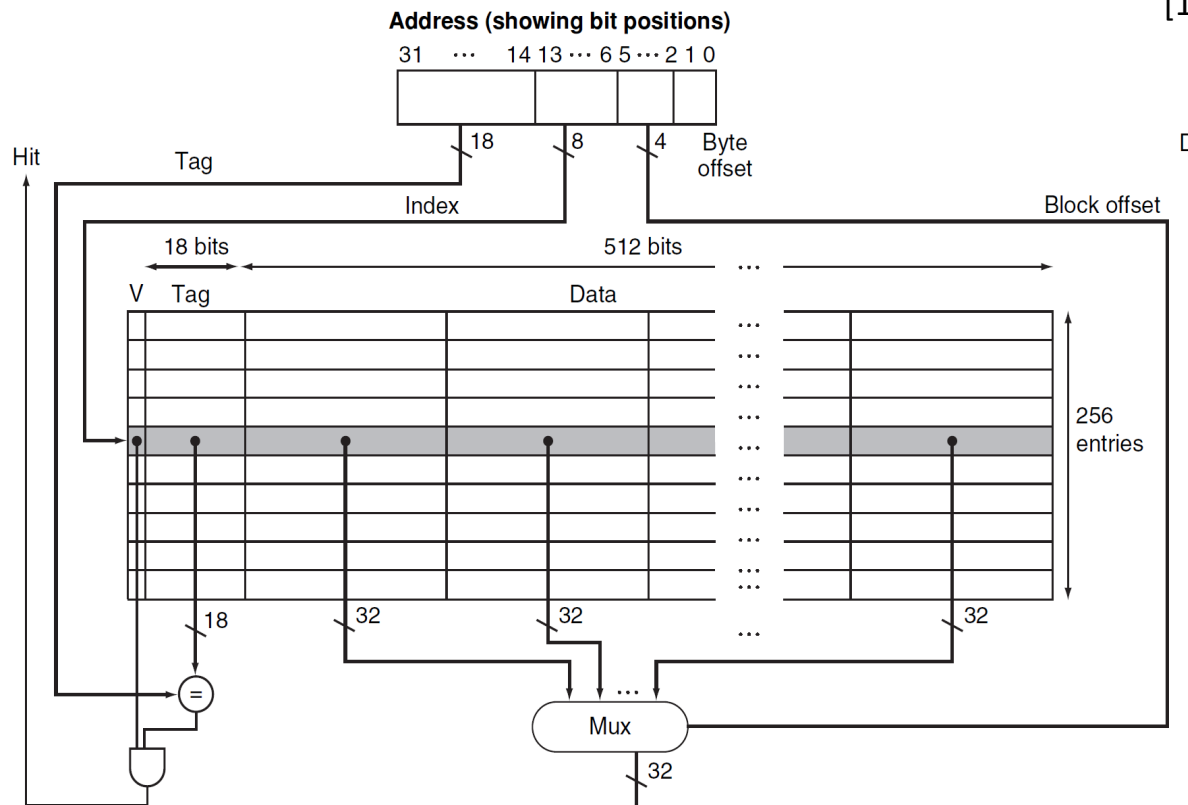
- If the cache has 256 blocks, each block has 16 words

$$[1 \text{ (valid bit)} + 18 \text{ (tag)} + 32 \times 16 \text{ (data)}] \times 256 \text{ (block)}$$

$$\text{Offset: } 16 \times 4 \text{ bytes} = 2^6 \Rightarrow 6 \text{ bits}$$

Index: use 8 bits to address 256 blocks

$$\text{Tags: } 32 - 8 - 6 = 18 \text{ bits}$$



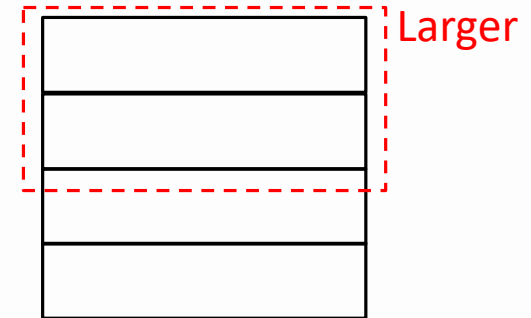
# Block Size Matters

- Block size vs miss rate

- Larger block size could reduce miss rate
  - Due to spatial locality

- Since cache size is fixed

- Larger blocks means the number of blocks decreases
  - More collisions will then increase miss rate

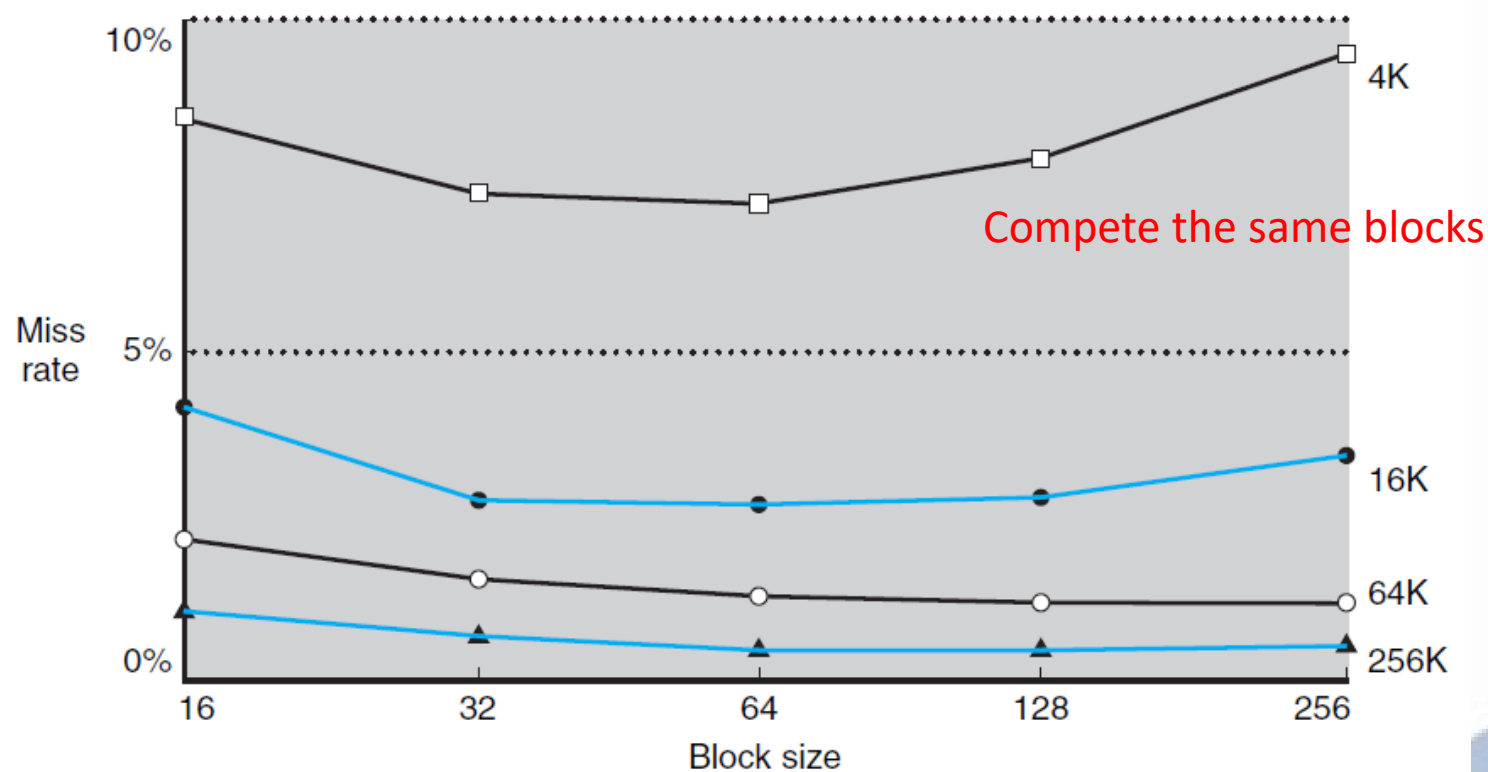


- Larger blocks leads to pollution (more data that may not be needed)

- Larger miss penalty

- Time to fetch the block has two parts: (1) the latency to the first word; (2) the transfer time for the rest of the block.
- Miss penalty will increase as block size increases.
- Can override benefit of reduced miss rate

# Block Size Considerations





# Cache Misses

---

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
    - Cache miss vs page fault -> stall vs os interrupt
  - Fetch block from next level of hierarchy
    - Instruction cache miss: Restart instruction fetch
    - Data cache miss: stall CPU until memory finishes its task

# Read-Through/Non-Read Through

---

- On cache miss,
  - “Read Through” means going to main memory directly to read the missed data and move the data block to cache simultaneously.
  - “Non-Read Through”
    - Move the data from main memory to cache first
    - After the missed data has been loaded to cache, CPU will read the data from cache
    - Slower than “Read Through”

# Write-Through

---

- On data-write, if the block that needs to be updated is in cache
  - But then cache and memory would be inconsistent
- Write through: update both cache and memory
  - Writes need more time
  - e.g., if base CPI = 1, 10% of instructions are stores, “write to memory” takes 100 cycles
    - $\text{Effective CPI} = 1 + 0.1 \times 100 = 11$
  - Good for Multi-processor due to data consistency
- 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, only update the block in cache
  - data only written to main memory when it is evicted from the cache or when it needs to be updated.
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - More complicate to implement

# Write Allocation

---

- What should happen on a write miss?
  - Data not in the cache when write
- Two methods
  - Allocate on miss (fetch on write) – worked with write back
    - fetch the block from memory to cache and only update data in the cache
    - Work with write back when hit (not considering data consistency)
    - Cache gets the latest data
  - Write around: don't fetch the block – worked with write through
    - Only write data into memory
    - Since programs often write a whole block before reading it (e.g., initialization)
    - Work with write through when hit (considering data consistency)
    - Memory gets the latest data

# Fetch on Write/Write Around

---

- On data-write, if the block that needs to be updated is not in cache
- Fetch on Write
  - First, move the data block from memory to cache
  - Then do write back
- Write Around
  - Write the data block in the memory directly without touching cache
- In general, for data write, one would do “write back” on data hit and “fetch on write” on data miss or
- “write through” on data hit and “write around” on data miss

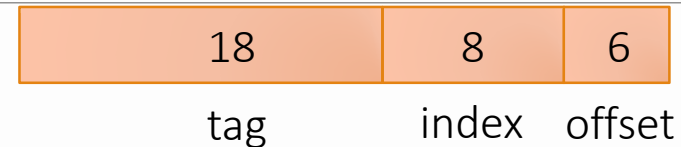
# Case: Intrinsity FastMATH

---

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB:  $256 \text{ blocks} \times 16 \text{ words/block}$
  - D-cache: offers write-through or write-back
  - one-entry write buffer
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

# Case: Intrinsity FastMATH

- Cache has 256 blocks
- Each block has 16 words



$\therefore 2^4 \text{ words} = 2^6 \text{ bytes}$

$\therefore \text{offset} = 6$

$\therefore 2^8 \text{ blocks}$

$\therefore \text{index} = 8$

Steps:

1. determine offset based on block size
2. Determine index based on # of blocks
3. The rest is Tag



# Case : Intrinsity FastMATH

- How many bits does the Cache need?

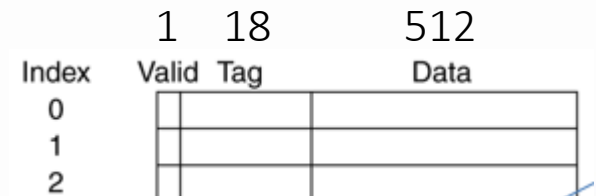


Key: compute # of blocks from index bit length, which is  $2^8$

Each entry has three fields: valid, tag, and data

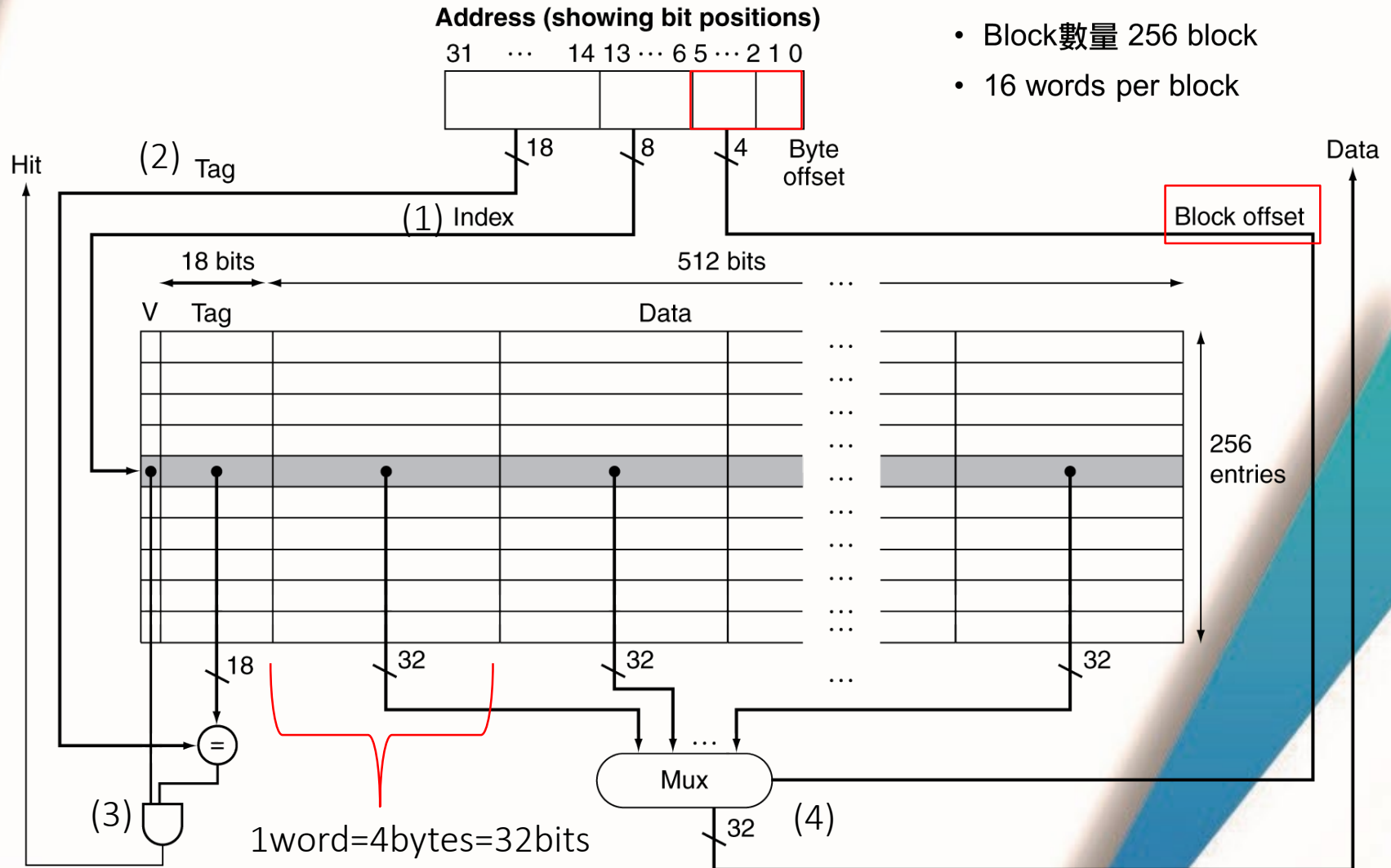
*Data = 16words = 64bytes = 512bits*

It takes  $2^8 \times (512 + 1 + 18)$  bits



# Case: Intrinsity FastMATH

- Block數量 256 block
- 16 words per block



# Introduction

---

- Metric used to evaluate cache performance
- How to increase cache performance
  - Reduce miss rate
    - Associative mapping
  - Reduce miss penalty
    - Multilevel caching

# Measuring Cache Performance

*CPUTime* =

(CPU Execution clock cycles + Memory Stall clock cycles) × Clock cycle time

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

# Measuring Cache Performance

Memory-stall cycles=Read-stall cycles+Write-stall cycles

Read-stall cycles =  $\frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$

Write-stall cycles =  $\left( \frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty} \right) + \text{Write buffer stalls}$

can be ignored in a system with reasonable write buffer depth

In most write-through cache, read and write miss penalties are the same

# Measuring Cache Performance

---

- With simplified assumptions:

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, no stall) = 2
- ❑ Load & stores are 36% of instructions

## Memory stall cycles

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

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

## For Every instruction ❑ Miss cycles per instruction (Miss CPI)

- ❑ I-cache:  $1 \times 0.02 \times 100 = 2$
- ❑ D-cache:  $0.36 \times 0.04 \times 100 = 1.44$

## ❑ Actual CPI = Base+ I-cache+D-cache

$$= 2 + 2 + 1.44 = 5.44$$

- ❑ Ideal CPU is  $5.44/2 = 2.72$  times faster!

# Cache Performance Example

- Actual CPI = Base + I-cache + D-cache  
$$= 2 + 2 + 1.44 = 5.44$$
  - Ideal CPU is  $5.44/2 = 2.72$  times faster
- If we have a CPU with Base CPI=1  
Actual CPI(new)= $1+2+1.44=4.44$
- The negative impact of memory stall on performance increases using a better CPU

$$CPI_{old} : \frac{3.44}{5.44} = 63\%$$

$$CPI_{new} : \frac{3.44}{4.44} = 77\%$$



# Average Access Time

- Hit time is also important to performance
- Average memory access time (AMAT)
  - $AMAT = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$  → Miss time
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, l-cache miss rate = 5%
  - $AMAT = 1 + 0.05 \times 20 \times 1 = 2\text{ns}$ 
    - 2 cycles per instruction
    - 1 clock cycle takes 1 ns

# Performance Summary

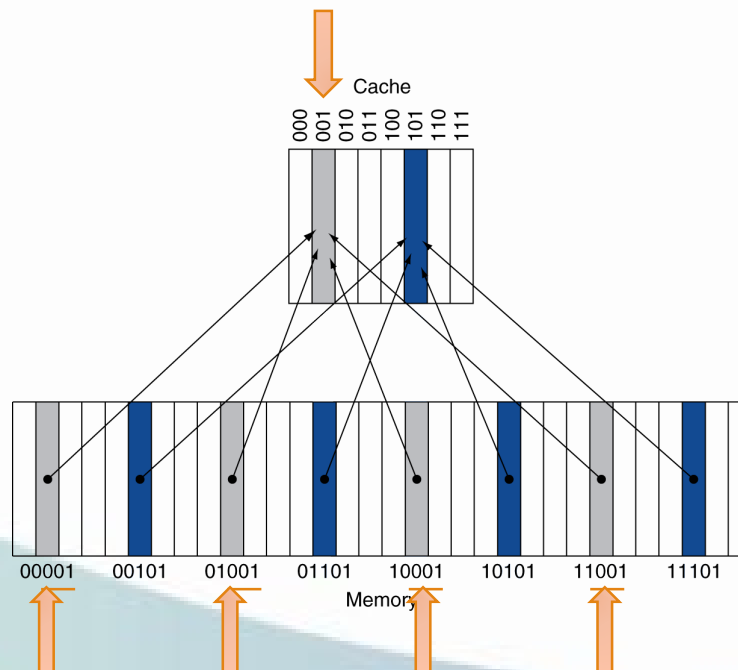
---

- When CPU performance increases
  - Miss penalty becomes more significant
  - Decreasing base CPI
    - Greater proportion of time spent on memory stalls
- Can't neglect cache behavior when evaluating system performance

# Cache Mapping Mechanisms

# Direct Mapped Cache

- Location determined by address
- Direct mapped: only one choice
  - $(\text{Block address}) \bmod (\text{\#Blocks in cache})$



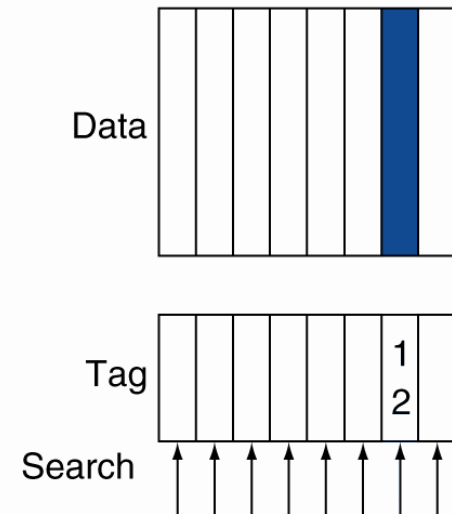
- $\#$ Blocks is a power of 2
- Use low-order address bits

# Associative Caches

---

- Fully associative vs. Direct-mapped
  - Direct-mapped: A given block can only go in a specific cache entry
  - Fully associative: A given block can go in any cache entry
- Fully associative cache
  - All entries need to be searched at once
  - Comparator per entry (expensive)
- $n$ -way set associative
  - Each set contains  $n$  entries
    - Search all entries in a given set at once
    - $n$  comparators (less expensive)
  - Block number determines which set
    - $(\text{Block number}) \% (\text{\#sets in cache})$

### Fully associative



Block can go in  
anywhere in the cache  
(need to search all  
entries when accessed)

# Spectrum of Associativity

- For a cache with 8 entries

**One-way set associative  
(direct mapped)**

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

**Two-way set associative** 2 slots per set

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

**Four-way set associative** 4 slots per set

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

**Eight-way set associative (fully associative)** 8 slots per set

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

# Associativity Example

## ❑ Compare 4-block caches

- Direct mapped, 2-way set associative, fully associative
- Block access sequence: 0, 8, 0, 6, 8

## ❑ Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	$0\%4=0$	miss	Mem[0]			
8	$8\%4=0$	miss	Mem[8]			
0	$0\%4=0$	miss	Mem[0]			
6	$6\%4=2$	miss	Mem[0]		Mem[6]	
8	$8\%4=0$	miss	Mem[8]		Mem[6]	

If accessing the same entry with different address consistently, it will keep causing misses



# Associativity Example

## ❑ 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	$0\%2=0$	miss	Mem[0]			
8	$8\%2=0$	miss	Mem[0]	Mem[8]		
0	$8\%2=0$	hit	Mem[0]	Mem[8]		
6	$6\%2=0$	miss	Mem[0]	Mem[6]		
8	$8\%2=0$	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]	

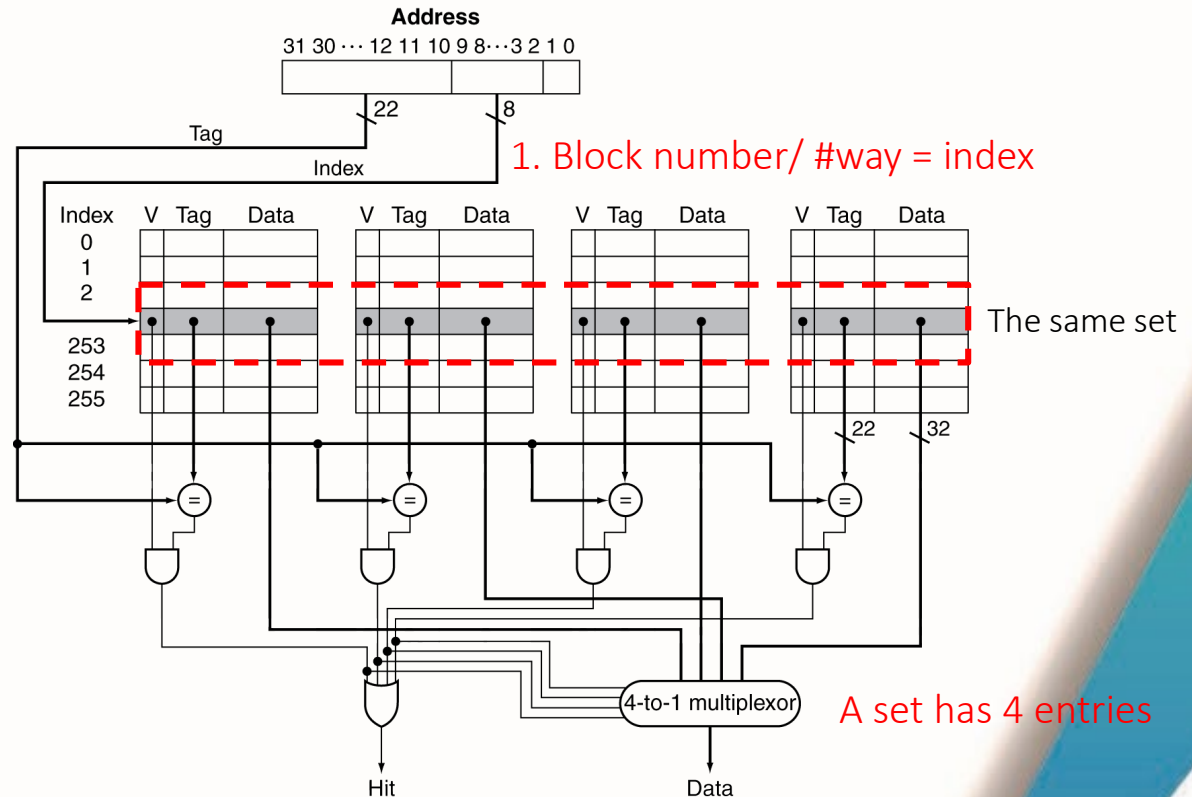
# Trade-off : Associativity vs. Hit Rate

---

- Increased associativity decreases miss rate
  - But with diminishing returns
  - And with increasing hardware costs
- The data miss rate for SPEC CPU2000 with a 64 KB cache of a 16-word blocks. The associativity ranges from direct mapped to eight-way.
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

# 4-Way Set Associative Cache Organization

P.396



# Example: Size of Tags vs. Set Associativity

- Based on the **size of the cache**, the **size of the block** in the cache, and the **number of set** used, we can determine
  - total number of sets
  - Addressing (number of bits used for tag, index, and offset)
- For example
  - 4096 blocks
  - 4-word/block
  - 2-way
  - 32-bit memory address

$$4\text{ words} = 16\text{ bytes} = 2^4\text{ bytes} \Rightarrow \text{offset} = 4\text{ bits}$$
$$2\text{-way} \Rightarrow \text{每個set有2個blocks}$$
$$4096\text{ blocks} = 2^{12}\text{ blocks} = \frac{2^{12}}{2}\text{ sets} = 2^{11}\text{ sets}$$

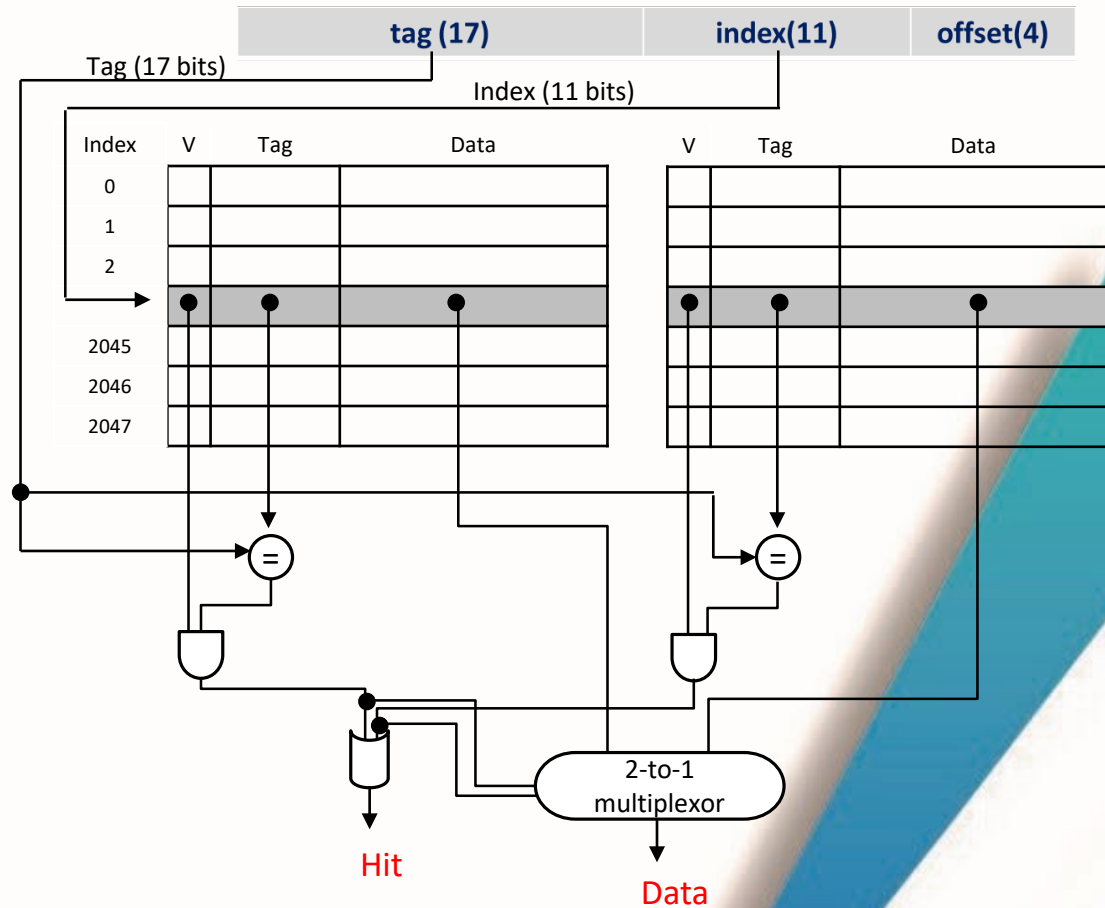
$$\left\{ \begin{array}{l} \text{offset bit} = 4 \\ \text{index bit} = 11 \\ \text{tag bit} = 32 - 11 - 4 = 17 \end{array} \right.$$

## Following the previous setting

– Total number of tag bits

$$32 - 11 (\text{index bits}) - 4 (\text{offset bits}) = 17$$

$$\text{total tag bits} = 17 \times 2^{11} \times 2$$



# 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
- First in First Out (FIFO)
  - Simple to be implemented, but causing more misses than LRU
- Random
  - Gives approximately the same performance as LRU for high associativity

# Multilevel Caches

---

- Primary cache attached to CPU
  - Small, but fast
- Level-2 cache works when the primary cache misses
  - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some 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

## ■ Clock cycles with just primary cache

- Miss penalty =  $100\text{ns}/0.25\text{ns} = 400$  cycles
- Effective CPI =  $1 + 0.02 \times 400 = 9$  clock cycles

$$\therefore 4\text{GHz} = 4 \times 10^9 \text{ Hz}$$

$$\therefore 1 \text{ cycle} = \frac{1}{4 \times 10^9} \text{ s} = 0.25\text{ns}$$



# Example (cont.)

---

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- L1 miss penalty (L2 hit)
  - $5\text{ns}/0.25\text{ns} = 20 \text{ cycles}$
- L2 miss penalty
  - 400 cycles (memory access)
- $\text{CPI} = 1 + 0.02 \times 20 + 0.005 \times 400 = 3.4 \text{ cycles}$ 

hit

Miss L1

Miss L2
- Performance ratio =  $9/3.4 = 2.6$

# Multilevel Cache Considerations

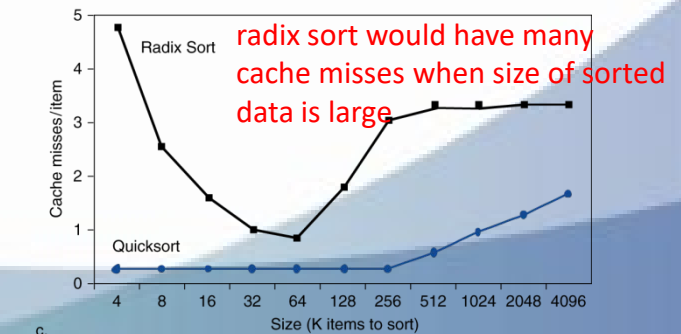
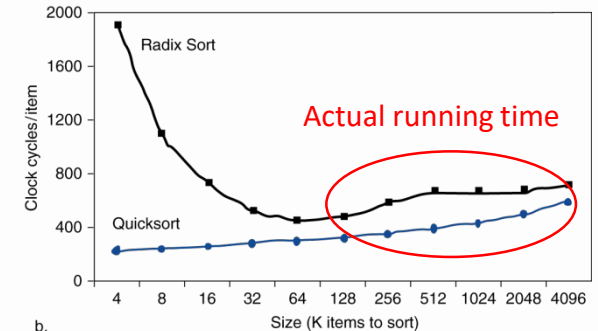
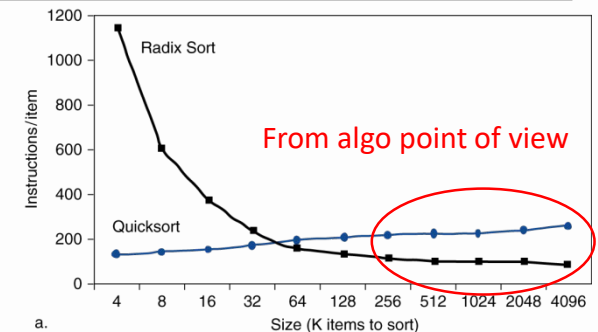
---

- Make L1 faster and L2 miss less
- 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
    - Because L1 is smaller; and with less miss penalty

# Interactions with Software

- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access

The basic idea of cache optimization is to use all the data in a block repeatedly before it is replaced on a miss.



# Virtual Memory

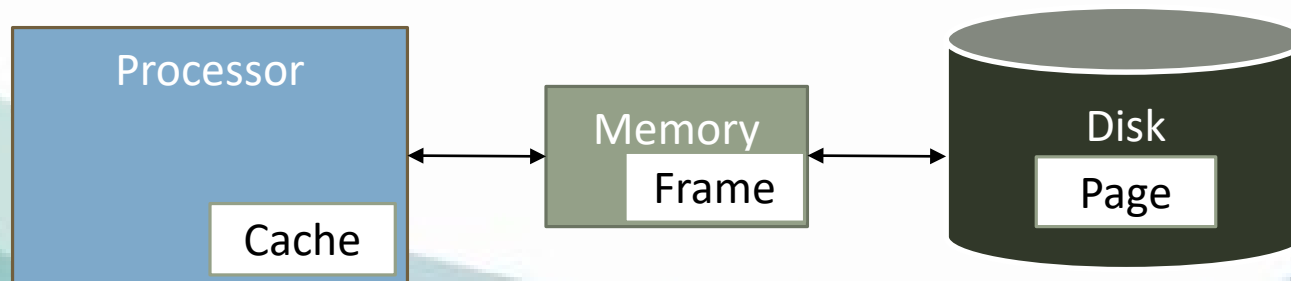
# Introduction

## ■ Main motivations

- Allow efficient and safe sharing of memory
- Remove the programming burden

## ■ Core idea

- Each program is compiled into its own address space (virtual address)
- Translate via **page table** between physical and virtual address spaces



# Introduction (cnt.)

## ■ Virtual Memory (VM)

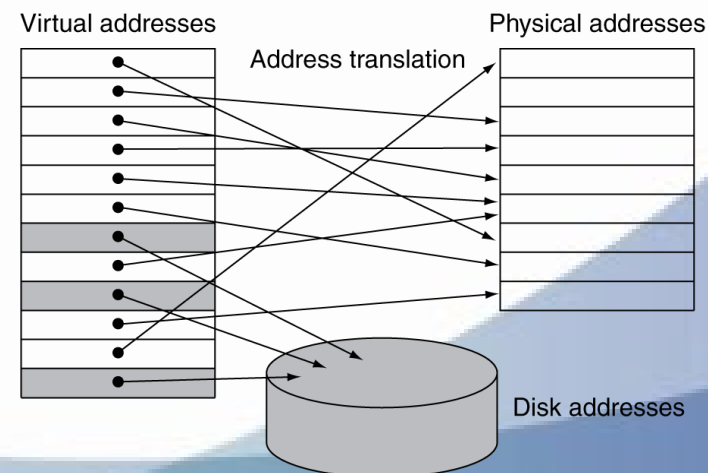
- Use memory as cache form the disk
- Each program has a private virtual address space holding its code and data
- Protected from other programs
- Managed by HW and OS jointly

## ■ Programs vs. VM

- Each has a private VM allocated holding its frequently used code and data

## ■ Translate virtual to physical addresses

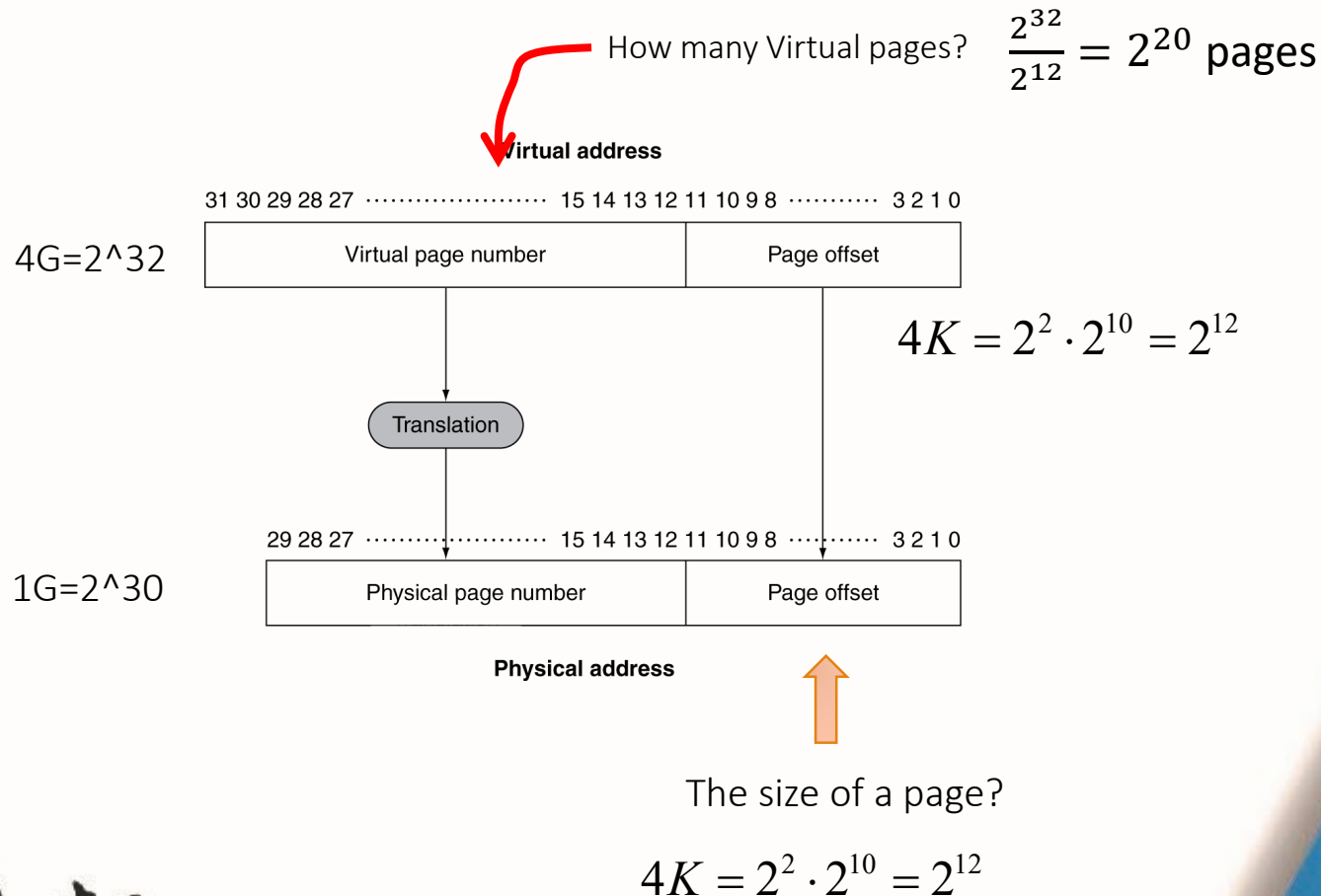
- Done by CPU and OS
- In VM “block” is called a “page”
- In Physical memory, “block” is called a “frame”
- VM translation “miss” is called a “page fault”



# Address Translation

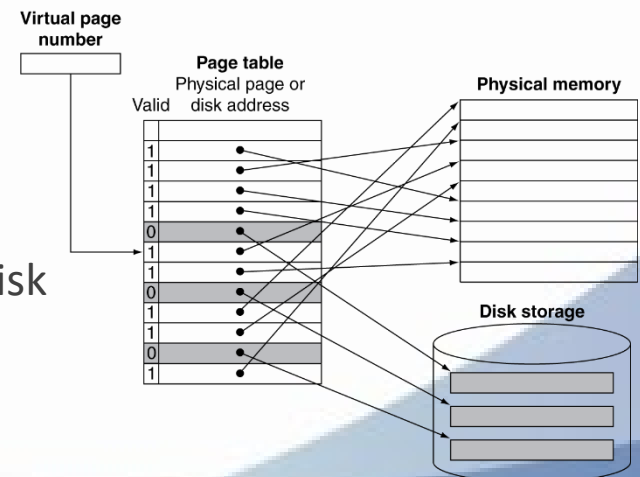
Assume physical memory has 1GB, VM has 4GB

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



# Page Tables (PTE)

- Stores translation information
  - Stored in main memory
  - Containing page table entries in an array form, indexed by virtual page number
  - “Page table register” stored in CPU has the address to the page table in physical memory
  - with other status bits (referenced, dirty, valid, ...)
- Page hit
  - Page present in memory
  - Found physical page number in PTE,
- Page fault
  - PTE can refer to location in “swap space” on disk





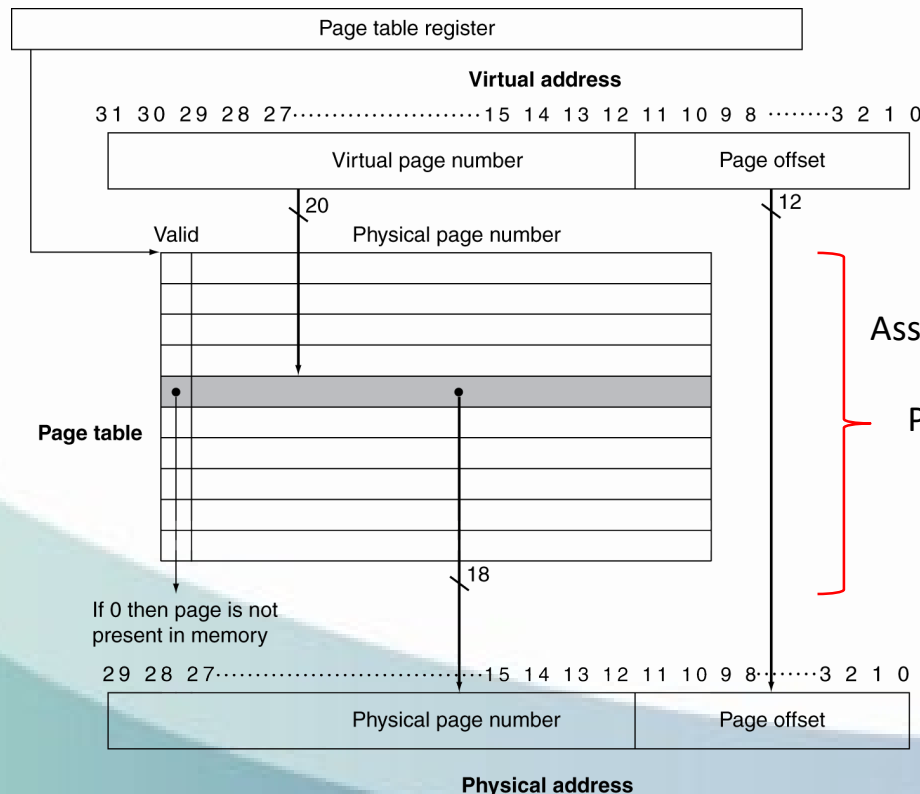
# Page Fault Penalty

---

- When page fault occurs, the page must be fetched from disk
  - Takes millions of clock cycles
  - Handled by OS
  - Write policy: Write back
    - write through not feasible
      - Considering access time, write back is more feasible
- Try to minimize page fault rate
  - Fully associative placement
  - Smart replacement algorithms (suitable for software implementation)

# Page Tables

- Stores virtual-physical address mappings in memory
  - Page table register in CPU points to page table in main memory
  - Array of entries indexed by virtual page number



Not every virtual address corresponds to a physical address

Assume each page table entry takes 4 bytes

Page table size :  $2^{20} \times 4$  bytes

Requiring two accesses to memory

# Page Fault

---

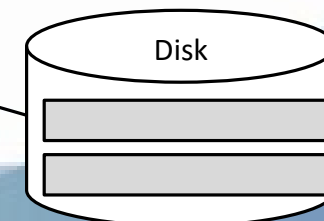
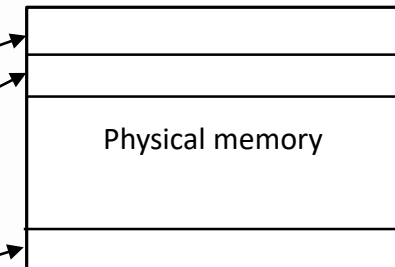
- Page fault means the page is not in physical memory
- It entails millions of cycles to process
- Using software to handle this
  - The time spent on software handling is relatively small compared to disk access
  - Can involve context-switched

# Replacement and Writes

- To reduce page fault rate, prefer least-recently used (LRU) replacement
  - Reference 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
  - Read/Write one page at once
  - Write through is impractical
  - Use write-back
  - Dirty bit in PTE set when page is written

Page Table with 3 status bits (referenced, dirty, valid)

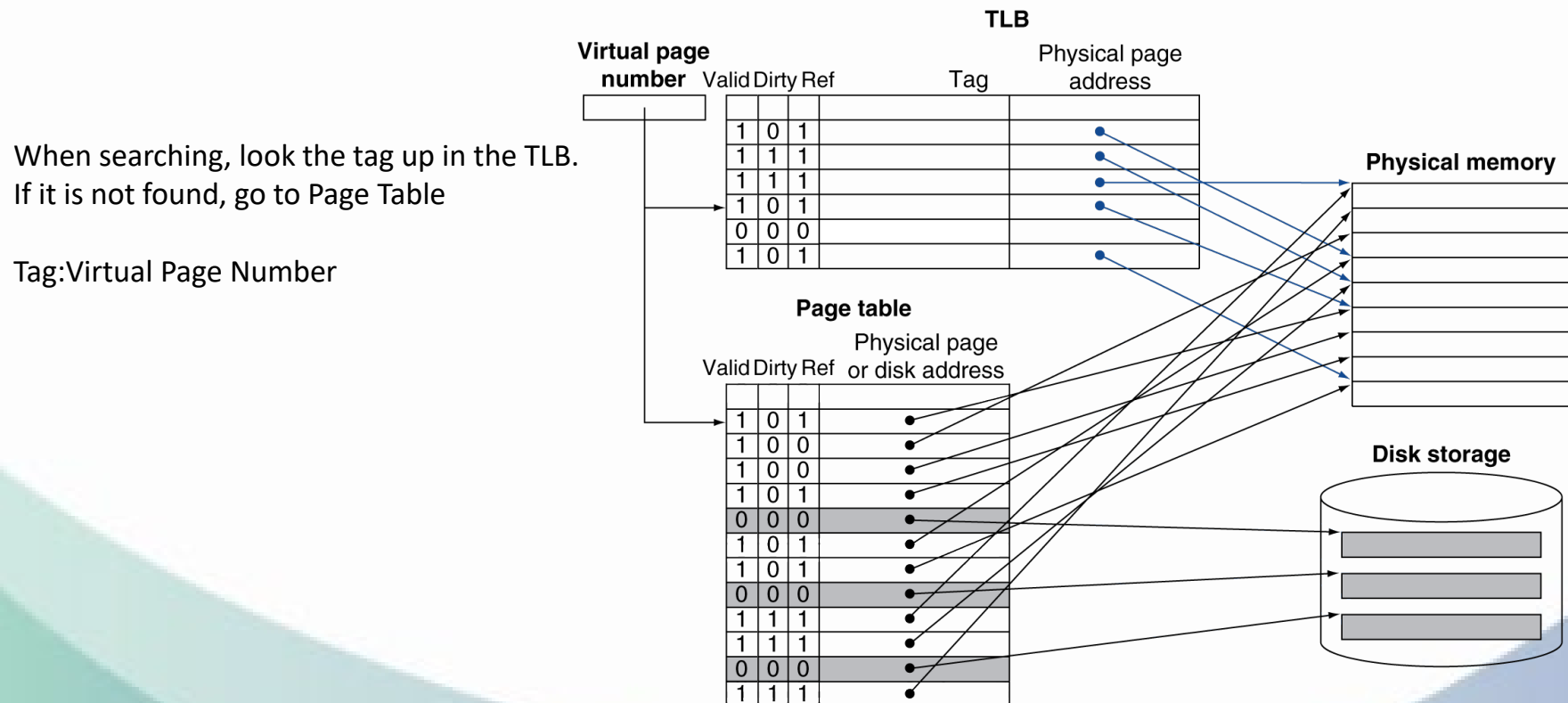
V	D	R	Physical address/disk address
1	0	1	
0	0	0	
1	1	1	
1	0	0	



# Fast Translation Using a TLB

- To resolve the issue of a large Page table and long access time
  - A large page table can be addressed by
    - Multi-level page tables
    - Inverted Page Tables with hashing (ex: with the same number of pages in the physical memory)
- In essence, it is a cache for Page Table
- 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 entries, 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 Hit

---

## ■ TLB Hit

- Read
- Write
  - set dirty bit to 1 in TLB (in cache) for writing back to page table (in memory) on replacement

# TLB Misses

---

- If page is in memory (valid bit==1)
  - 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 (in disk)
- 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

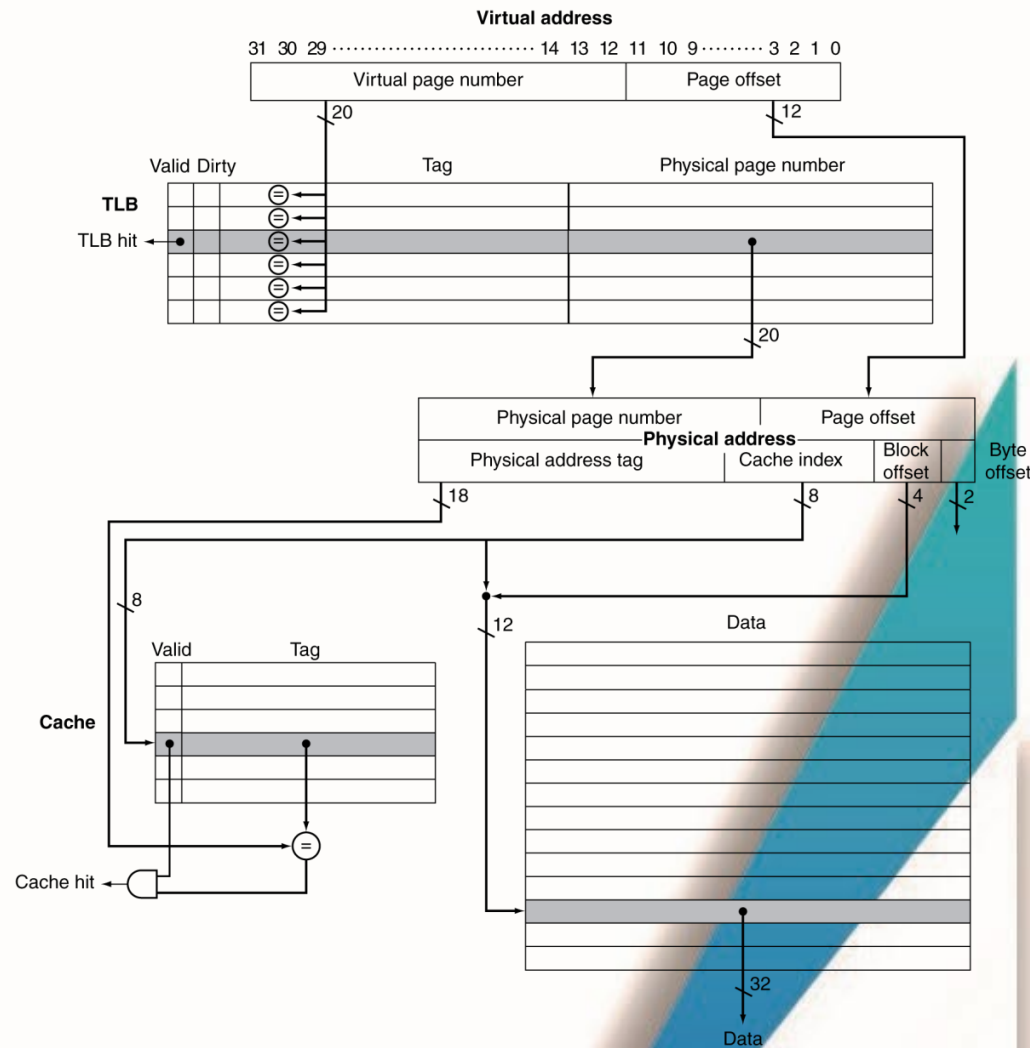
# Page Fault Handler

---

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

# TLB and Cache Interaction

- If cache tag uses physical address
  - Need to translate before cache lookup
- Two alternatives:
  - use virtual address tag
    - Complications due to aliasing
      - Different virtual addresses for shared physical address
  - Virtually Indexed but physically tagged

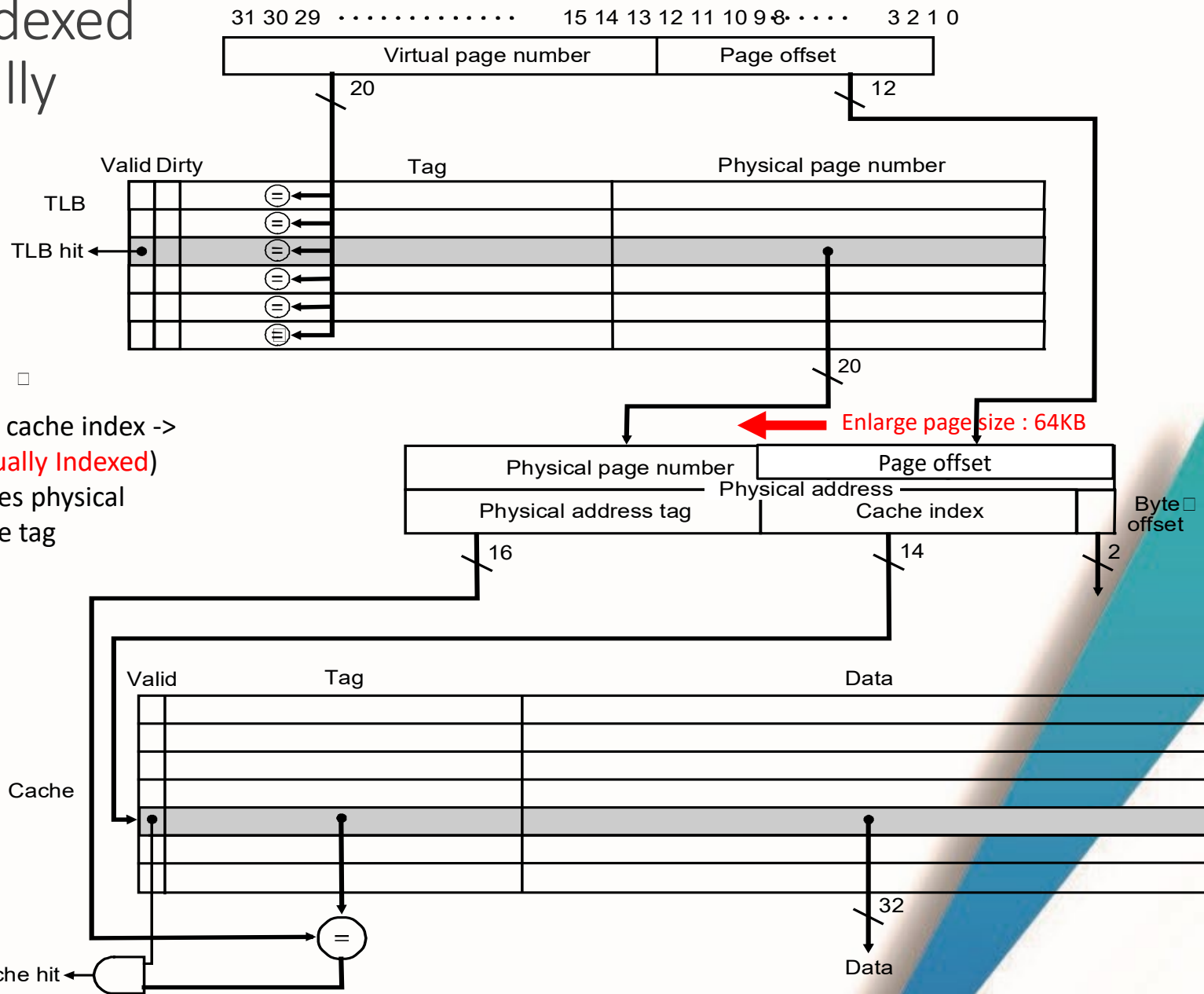


# Virtually Addressed Cache

---

- 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!
    - Consistency issue: For update: must update all cache entries with same physical address or memory becomes inconsistent
    - Determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits

# Virtually Indexed but Physically Tagged

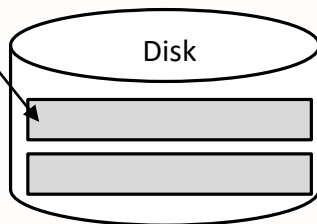
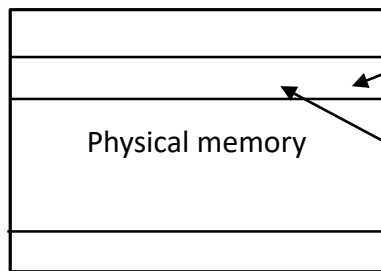


Page Table

V	D	R	Physical address/disk address
1	1	1	
1	0	1	
			⋮
1	0	0	

Page fault

Page hit



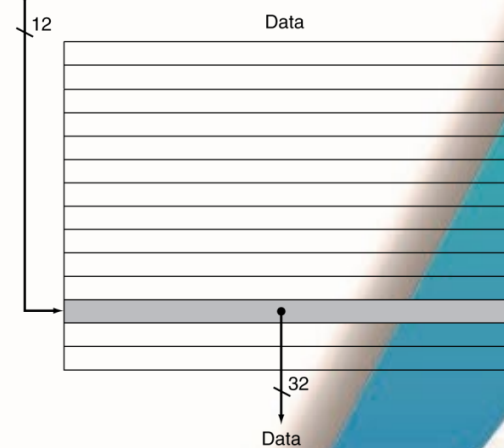
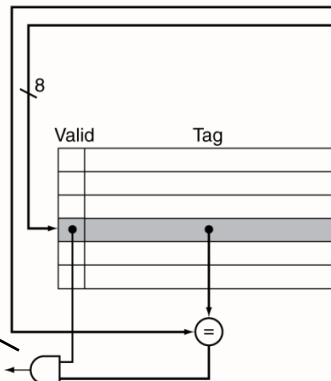
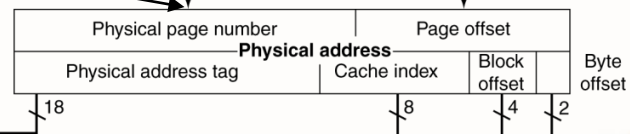
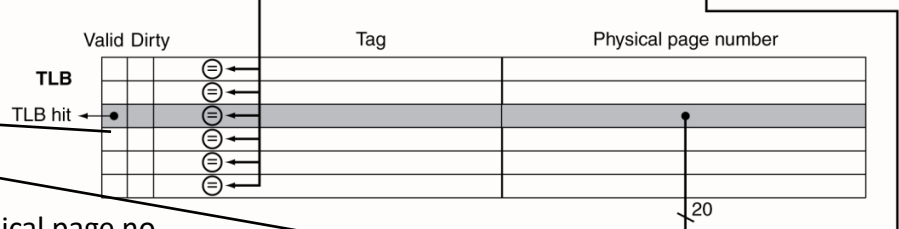
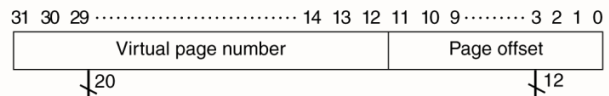
TLB miss

Physical page no

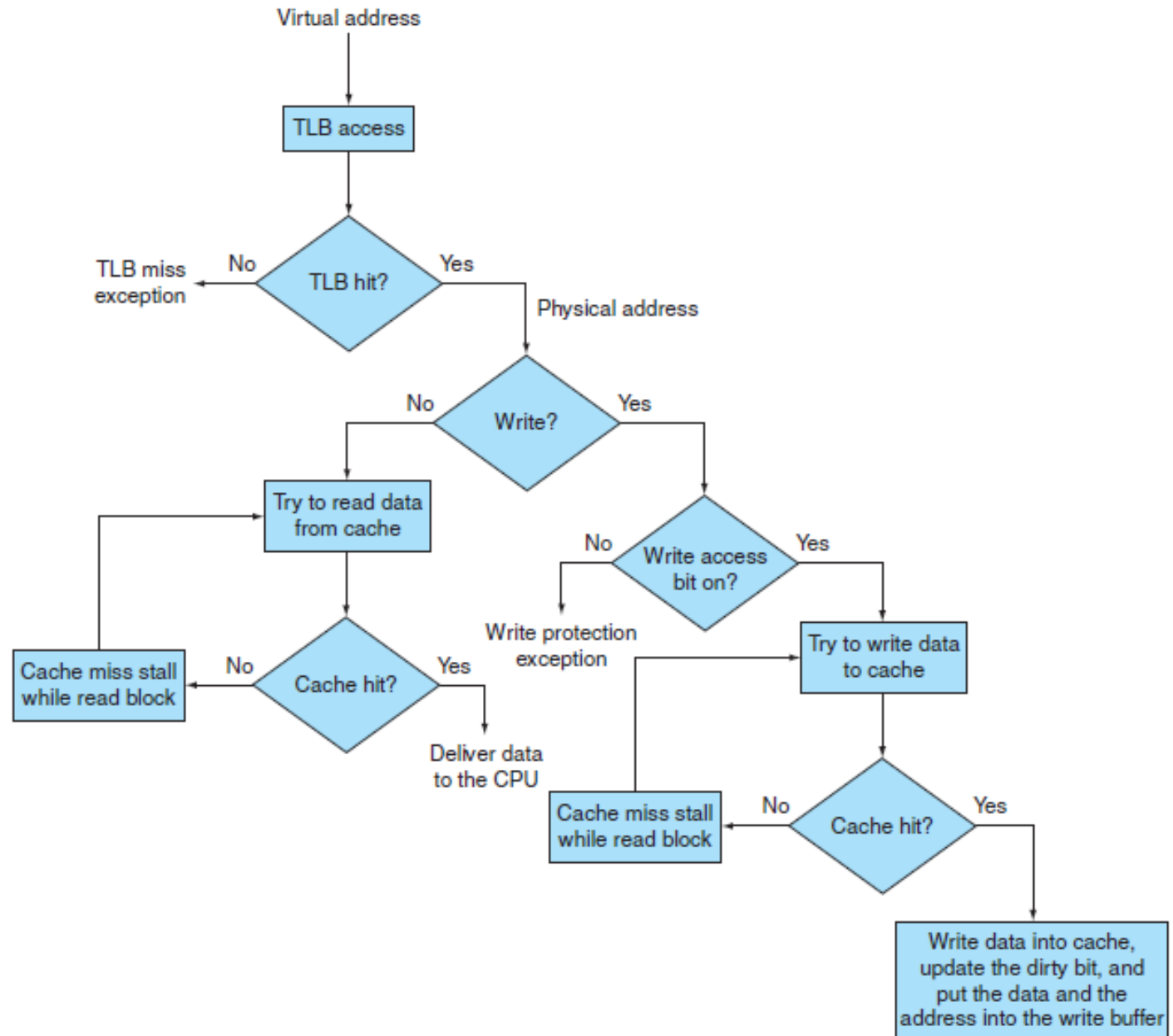
Cache miss

Cache hit

Virtual address



# TLB and Cache Interaction



# Possible Combinations of Events

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



# Memory Protection

---

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

# Memory Hierarchy

---

- 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

## ■ Caches

- 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

## ■ Virtual memory

- Usually full associativity (with full PTE)
  - reduced miss rate

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

# Replacement

---

- Choice of entry to replace on a miss
- 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
    - Close to LRU, easier to implement
  - First in First Out (FIFO)
    - Simple to be implemented, but causing more misses than LRU
- Virtual memory
  - Reference bits used for LRU approximation

# 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

# Fetch on Write/Write Around

---

- On data-write, if the block that needs to be updated is not in cache
- Fetch on Write
  - fetch the block from memory to cache and only update data in the cache
    - Simulating “write back” since loading the block from memory to cache turns out to be cache hit
- Write Around
  - Write the data block in the memory directly without touching cache
- In general, for data write, one would do “write back” on data hit and “fetch on write” on data miss or
- “write through” on data hit and “write around” on data miss

# 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
  - When cache size increases, the number of capacity misses are reduced
- 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
  - When associativity increases, it reduces conflict misses



# Cache Design Trade-offs

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