

# **Course Review**

# **Chapter 1 (6<sup>th</sup> chapter1)**

Fundamentals of Computer Design

# Great Architecture Ideas



# Great Architecture Ideas

- There are 8 great architectural ideas that have been applied in the design of computers for over half a century now.
- As we cover the material of this course, we should stop to think every now and then which ideas are in play and how they are being applied in the current context.



# Great Architecture Ideas

- Design for **Moore's law**.
  - The number of transistors on a chip doubles every 18-24 months.
  - Architects have to anticipate where technology will be when the design of a system is completed.
- Use **abstraction** to simplify design.
  - Abstraction is used to represent the design at different levels of representation.
  - Lower-level details can be hidden to provide simpler models at higher levels.
- Make the **common case fast**.
  - Identify the common case and try to improve it.
  - Most cost efficient method to obtain improvements.
- Improve performance via **parallelism**.
  - Improve performance by performing operations in parallel.
  - There are many levels of parallelism – instruction-level, process-level, etc.



# Great Architecture Ideas

- Improve performance via **pipelining**.
  - Break tasks into stages so that multiple tasks can be simultaneously performed in different stages.
  - Commonly used to improve instruction throughput.
- Improve performance via **prediction**.
  - Sometime faster to assume a particular result than waiting until the result is known.
  - Known as speculation and is used to guess results of branches.
- Use **a hierarchy of memories**.
  - Make the fastest, smallest, and most expensive per bit memory the first level accessed and the slowest, largest, and cheapest per bit memory the last level accessed.
  - Allows most of the accesses to be caught at the first level and be able to retain most of the information at the last level.
- Improve dependability via **redundancy**.
  - Include redundant components that can both detect and often correct failures.
  - Used at many different levels.



# ★ CPU Performance

In order to determine the effect of a design change on the performance experienced by the user, we can use the following relation:

$$CPU\ Execution\ Time = CPU\ Clock\ Cycles \times Clock\ Period$$

Alternatively,

$$CPU\ Execution\ Time = \frac{CPU\ Clock\ Cycles}{Clock\ Rate}$$

Clearly, we can reduce the execution time of a program by either reducing the number of clock cycles required or the length of each clock cycle.



# Instruction Count and CPI ★

- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction (CPI)
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

$$CPI = \frac{CPU\ Clock\ Cycles}{Instruction\ Count}$$



*CPU Clock Cycles = Instructions for a Program × Average Clock Cycles Per Instruction*

$$CPU\ Time = Instruction\ Count \times CPI \times Clock\ Period$$

$$CPU\ Time = \frac{Instruction\ Count \times CPI}{Clock\ Rate}$$





# Amdahl's Law ★

Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

Amdahl's Law depends on two factors:

- The fraction of the time the enhancement can be exploited.
- The improvement gained by the enhancement while it is exploited.

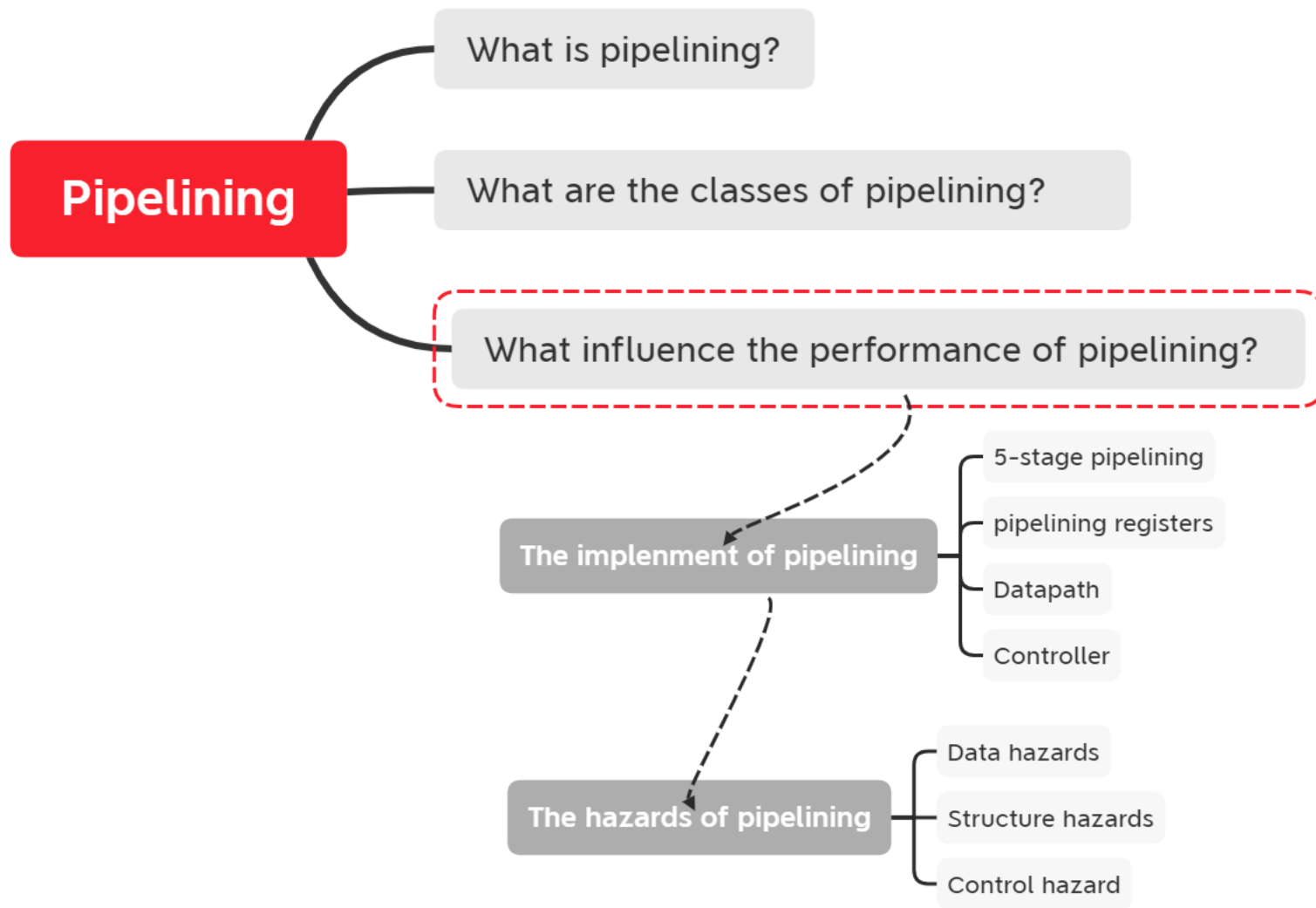
$$\text{Improved Execution Time} = \frac{\text{Affected Execution Time}}{\text{Amount of Improvement}} + \text{Unaffected Execution Time}$$

Make the common case fast!



# Chapter 2 ★ (6<sup>th</sup> Appendix C)

Pinelining



# Instruction Set Architecture

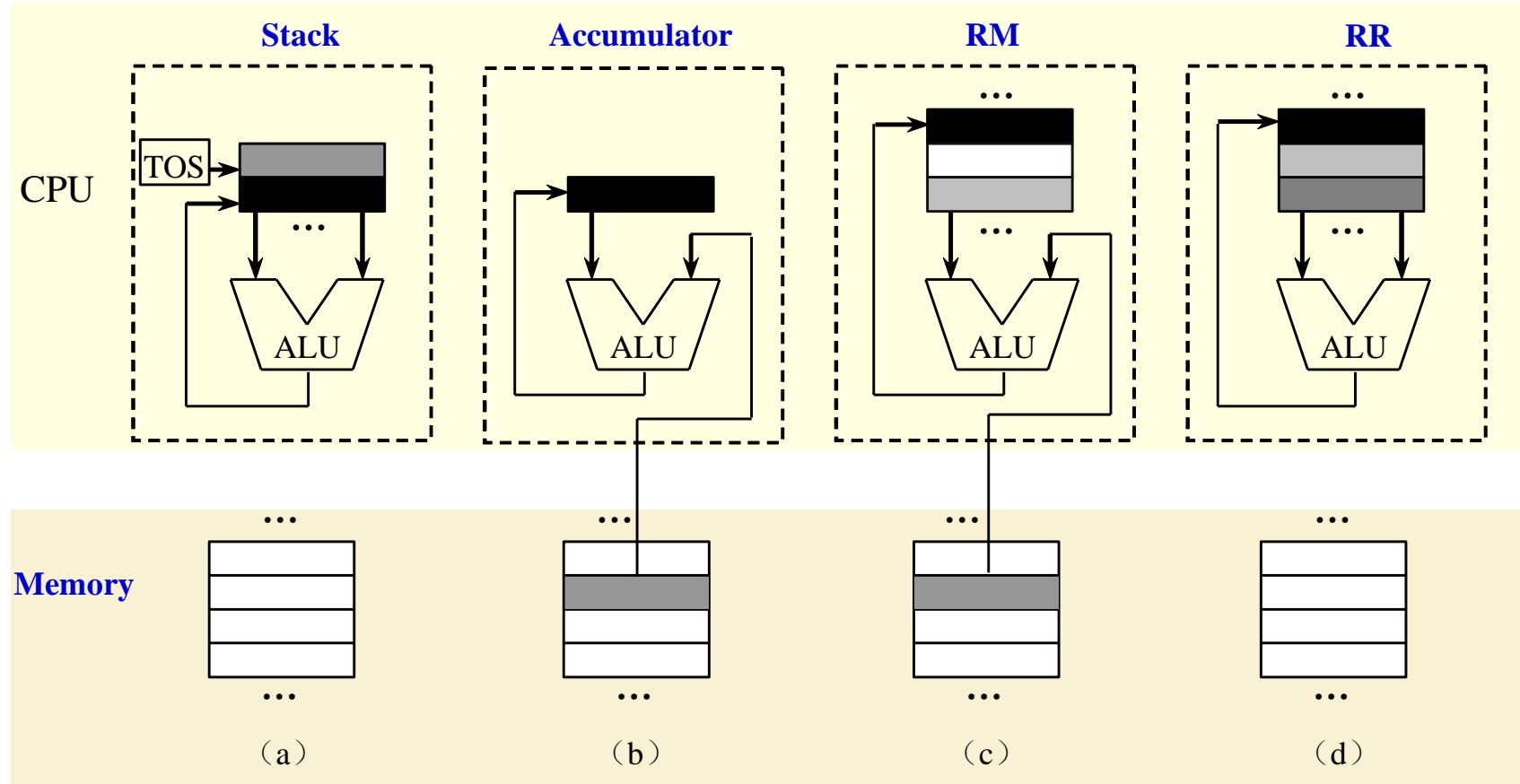
## ISA

- actual programmer-visible instruction set
- the boundary between software and hardware

# ISA: Class

- Most are **general-purpose register architectures** with operands of either registers or memory locations
- Two popular versions
  - register-memory ISA:** e.g., 80x86  
many instructions can access memory
  - load-store ISA:** e.g., ARM, MIPS, RISC-V  
only load or store instructions can access memory

# GPR Classification



# The Four ISA Design Principles

1. Simplicity favors **regularity**
  - Consistent *instruction size, instruction formats, data formats*
  - Eases implementation by simplifying hardware, leading to higher performance
2. Smaller is faster
  - Fewer bits to access and modify
  - Use the register file instead of slower memory
3. Make the **common case fast**
  - e.g. Small constants are common, thus small immediate fields should be used.
4. Good design demands good compromises
  - Compromise with special formats for important exceptions
  - e.g. A long jump (beyond a small constant)



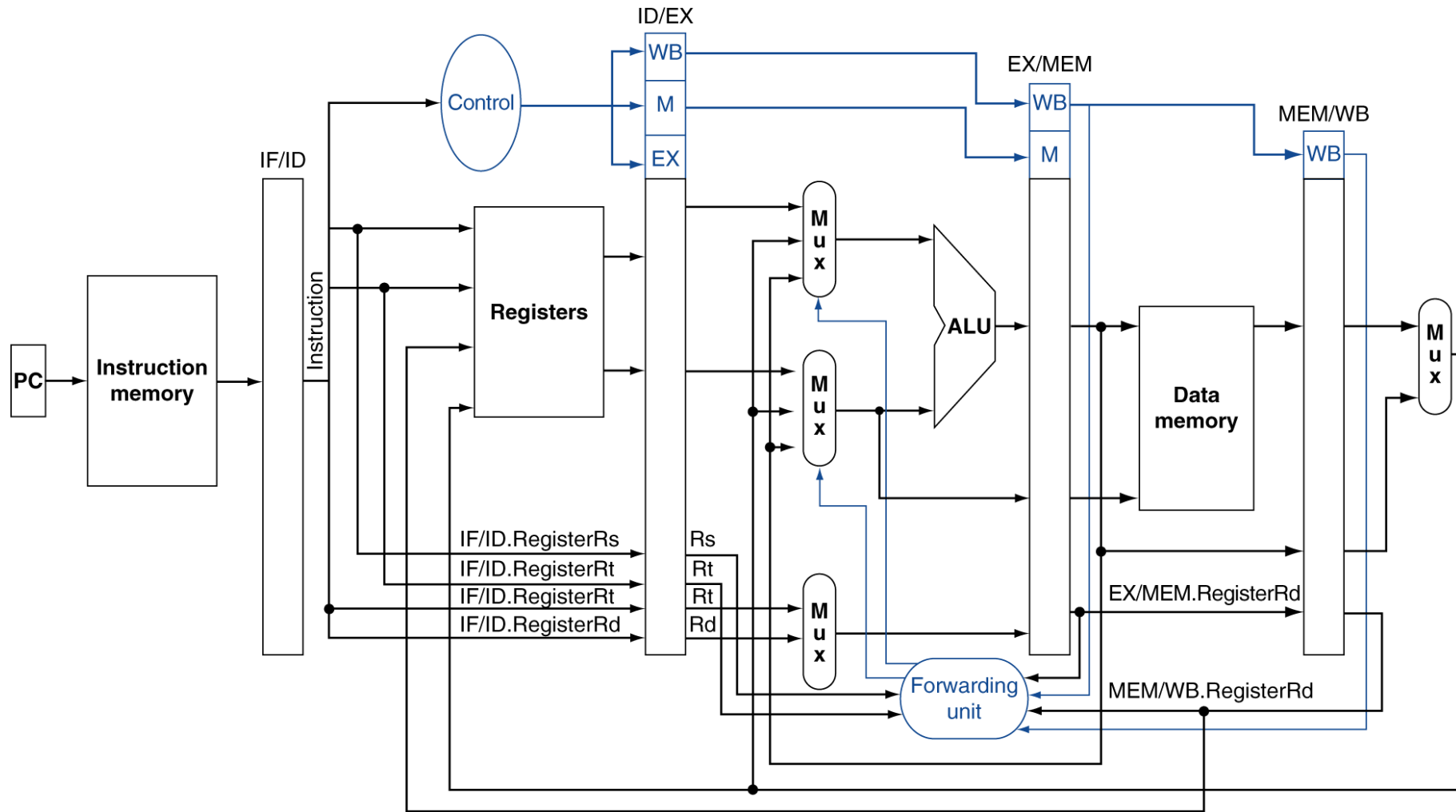
# Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation





# Pipelined Datapath ★



# The Classic Five-Stage Pipeline for a RISC Processor

- A Simple Implementation of RISC-V      Dependences are a property of *programs*.
- Instructions Dependences      • Pipeline Hazards



- Data Dependences

- Name Dependences

- Anti-dependence

- Output-dependence

- Control Dependences

- Data Hazards

- RAW

- WAR

- WAW

- Branch Hazards

- Structural Hazards

**Hazard are properties of the *pipeline organization*.**



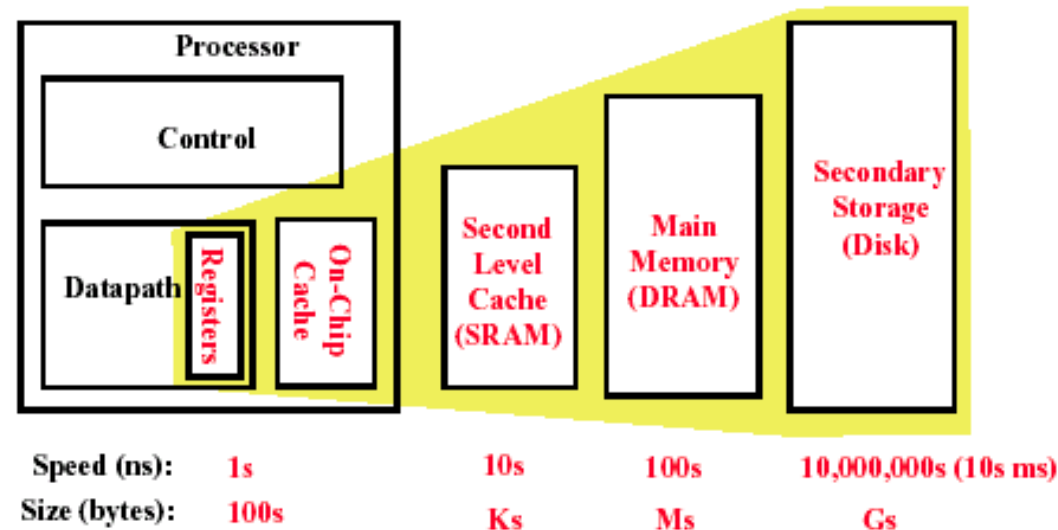
# **Chapter 3 (6<sup>th</sup> chapter2 & Appendix B)**

## Memory Hierarchy

# Memory Hierarchy of a Modern Computer System

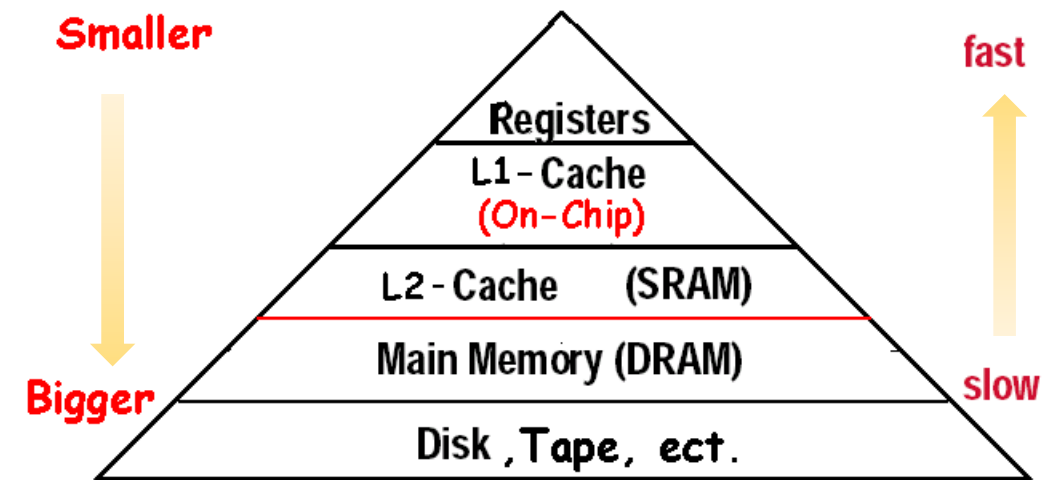
By taking advantage of the principle of locality:

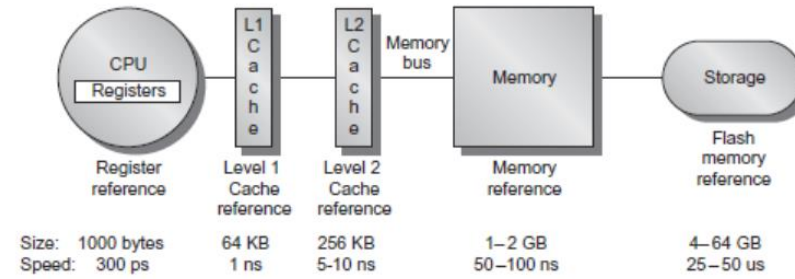
- Present the user with as much memory as is available in the cheapest technology.
- Provide access at the speed offered by the fastest technology.



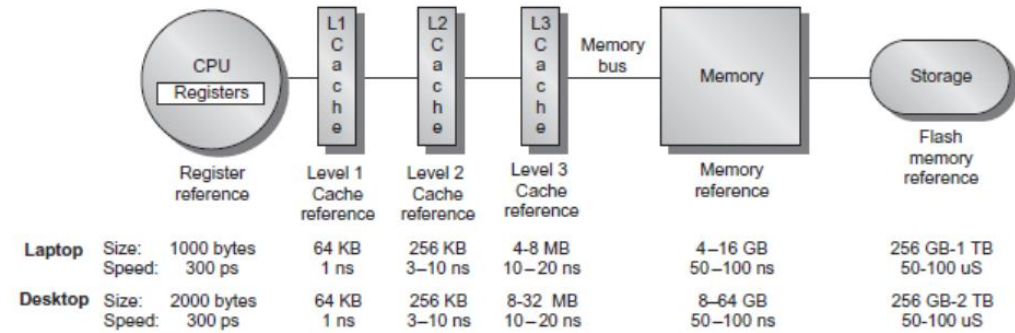
# What is a cache?

- Small, fast storage used to improve average access time to slow memory.
- In computer architecture, almost everything is a cache!
  - Registers “a cache” on variables – software managed
  - First-level cache a cache on second-level cache
  - Second-level cache a cache on memory
  - Memory a cache on disk (virtual memory)
  - TLB a cache on page table
  - Branch-prediction a cache on prediction information?

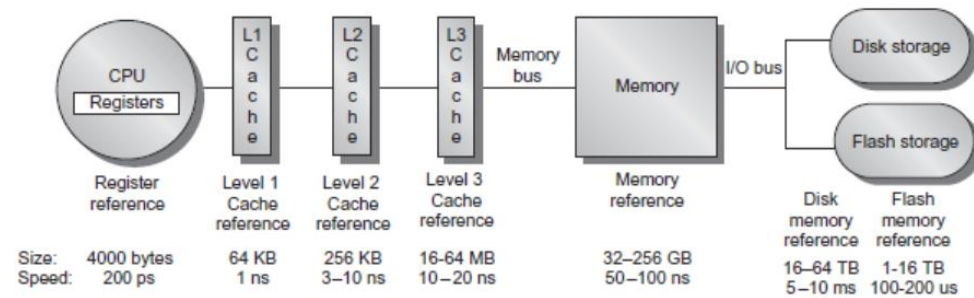




(A) Memory hierarchy for a personal mobile device



(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

# Four Questions for Memory Hierarchy Designers ★

**Caching** is a general concept used in processors, operating systems, file systems, and applications.

There are **Four Questions** for Memory Hierarchy Designers

- Q1: Where can a block be placed in **the upper level/main memory**?

*(Block placement)*



- Fully Associative, Set Associative, Direct Mapped

- Q2: How is a block found if it is in **the upper level/main memory**?

*(Block identification)*



- Tag/Block

- Q3: Which block should be replaced on a **(Virtual Memory)** miss?

*(Block replacement)*



- Random, LRU, FIFO

- Q4: What happens on a write?

*(Write strategy)*



- Write Back or Write Through (with Write Buffer)



## Q4: Write Strategy ★

- When data is written into the cache (on a store), is the data also written to main memory?
  - If the data is written to memory, the cache is called a **write-through cache**
    - Can always discard cached data - most up-to-date data is in memory
    - Cache control bit: only a valid bit
    - memory (or other processors) always have latest data
  - If the data is NOT written to memory, the cache is called a **write-back cache**
    - Can't just discard cached data - may have to write it back to memory
    - Cache control bits: both valid and dirty bits
    - much lower bandwidth, since data often overwritten multiple times
- Write-through adv: Read misses don't result in writes, memory hierarchy is consistent and it is simple to implement.
- Write back adv: Writes occur at speed of cache and main memory bandwidth is smaller when multiple writes occur to the same block.





# Summary

## Read Cache

Read through

Read allocate

## Write Cache

hit

### ***Write-through***

write is done  
synchronously both  
to the cache and to  
the backing store

### ***Write-back***

writing is done  
only to the cache

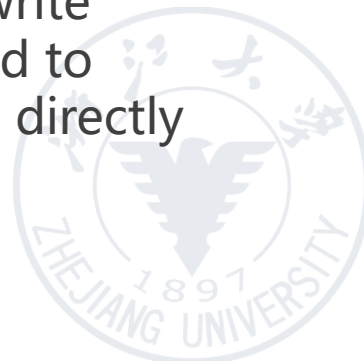
miss

### ***Write allocate***

data at the missed-write  
location is loaded to cache,  
followed by a write-hit  
operation.

### ***No-write allocate***

data at the missed-write  
location is not loaded to  
cache, and is written directly  
to the backing store



# How to Improve

Hence, there are more than 20 cache optimizations into four categories:

★ 
$$AMAT = HitTime + MissRate \times MissPenalty$$

1. Reduce the miss penalty
  - multilevel caches, critical word first, read miss before write miss, merging write buffers, and victim caches
2. Reduce the miss rate
  - larger block size, large cache size, higher associativity, way prediction and pseudo-associativity, and compiler optimizations
3. Reduce the miss penalty and miss rate via parallelism
  - non-blocking caches, hardware prefetching, and compiler prefetching
4. Reduce the time to hit in the cache
  - small and simple caches, avoiding address translation, pipelined cache access, and trace caches



# Average Memory Access Time ★





- Average Memory Access Time

$$\begin{aligned}\text{Average Memory Access Time} &= \frac{\text{Whole accesses time}}{\text{All memory accesses in program}} \\ &= \frac{\text{Accesses time on hitting} + \text{Accesses time on miss}}{\text{All memory accesses in program}} \\ &= \text{Hit time} + (\text{Miss Rate} \times \text{Miss Penalty}) \\ &= (\text{HitTime}_{Inst} + \text{MissRate}_{Inst} \times \text{MissPenalty}_{Inst}) \times \text{Inst}\% \\ &\quad + (\text{HitTime}_{Data} + \text{MissRate}_{Data} \times \text{MissPenalty}_{Data}) \times \text{Data}\%\end{aligned}$$

$$CPUtime = IC \times \left( \frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$



# Summary

- Memory hierarchy
  - From single level to multi level
  - Evaluate the performance parameters of the storage system (average price per bit  $C$ ; hit rate  $H$ ; average memory access time  $T$ )
- Cache basic knowledge 
  - Mapping rules
  - Access method
  - Replacement algorithm
  - Write strategy
  - Cache performance analysis
  - Reduce miss rate
  - Reduce miss penalty 
  - Reduce hit time
- Memory (the influence of memory organization structure on Cache failure rate)



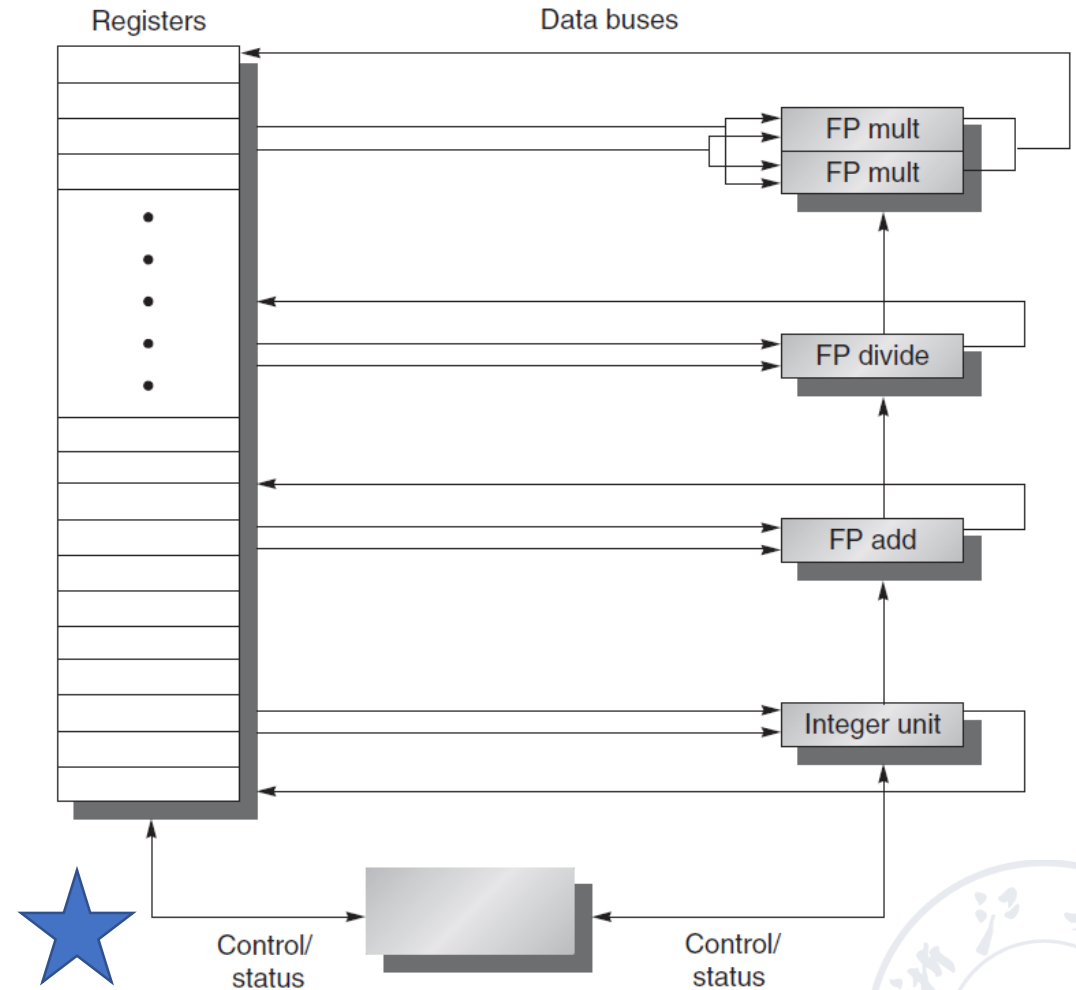
# Chapter 4(chapter3)

ILP

# Dynamic Scheduling

Idea: Dynamic Scheduling

Method: **out-of-order** execution



*Dynamic Scheduling with a Scoreboard*



# Dynamic Scheduling: Scoreboard algorithm

Instruction		Instruction Status			
		IS	RO	EX	WB
FLD	F6, 34(R2)	✓	✓	✓	✓
FLD	F2, 45(R3)	✓	✓	✓	
FMUL.D	F0, F2, F4	✓			
FSUB.D	F8, F6, F2	✓			
FDIV.D	F10, F0, F6	✓			
FADD.D	F6, F8, F2				



Name	Function Component Status								
	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	yes	Load	F2	R3				no	
Mult1	yes	MUL	F0	F2	F4	Integer		no	yes
Mult2	no								
Add	yes	SUB	F8	F6	F2		Integer	yes	no
Divide	yes	DIV	F10	F0	F6	Mult1		no	yes

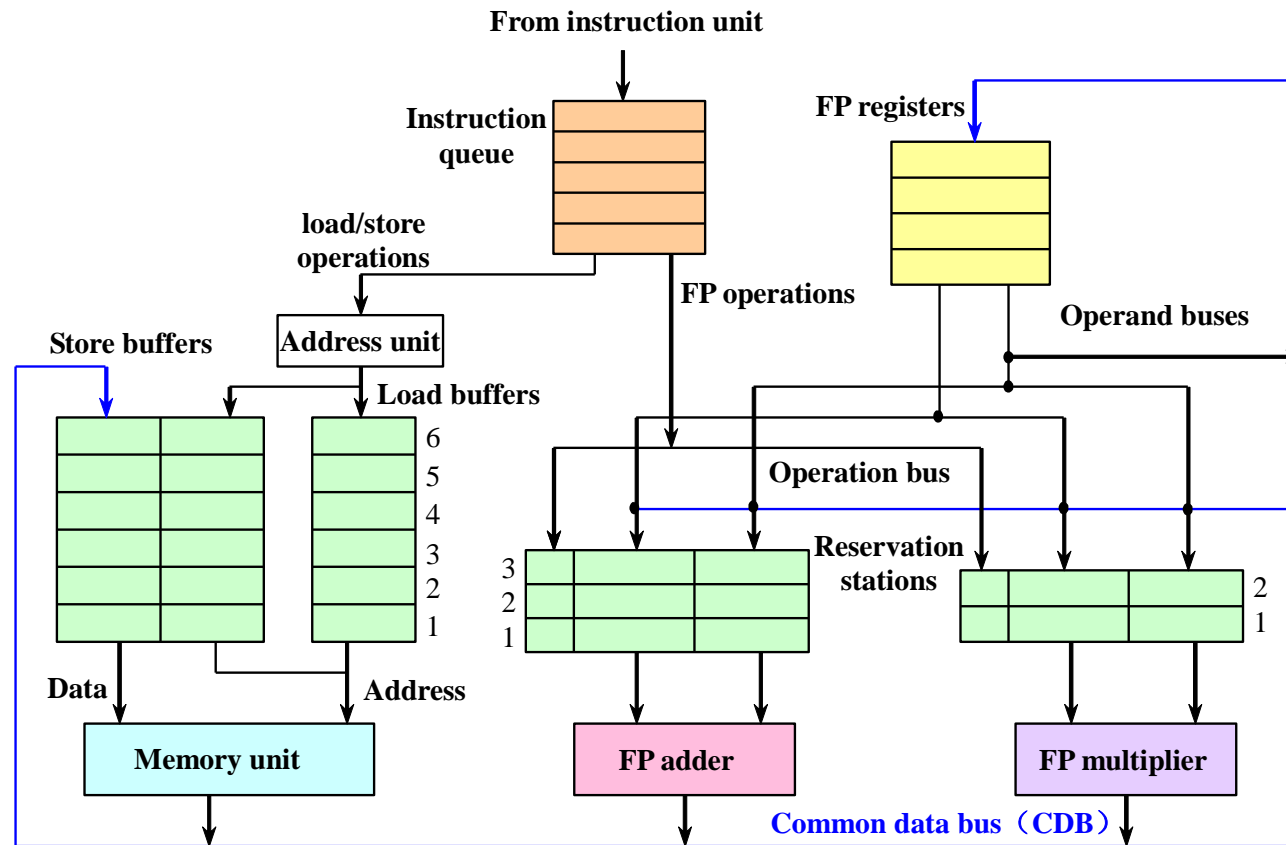
Rj, Rk :      “yes” ——operand is ready but not read;  
                  “no” & “Qj = null” ——operand is read;  
                  “no” & “Qj != null” ——operand is not ready.

	Register Status							
	F0	F2	F4	F6	F8	F10	...	F30
Qi	Mult1	Integer			Add	Divide		





# Tomasulo's Approach ★



*The basic structure of a floating-point unit using Tomasulo's algorithm*



# Tomasulo with Reorder Buffer - Summary

Instruction	Issue	Exec Comp	Writeback	Commit
FLD F6, 34(R2)	1	2	3	4
FLD F2, 45(R3)	2	3	4	5
FMUL.D F0, F2, F4	3	4-13	14	15
FSUB.D F8, F6, F2	4	5-6	7	16
FDIV.D F10, F0, F6	5	14-53	54	55
FADD.D F6, F8, F2	6	7-8	9	56

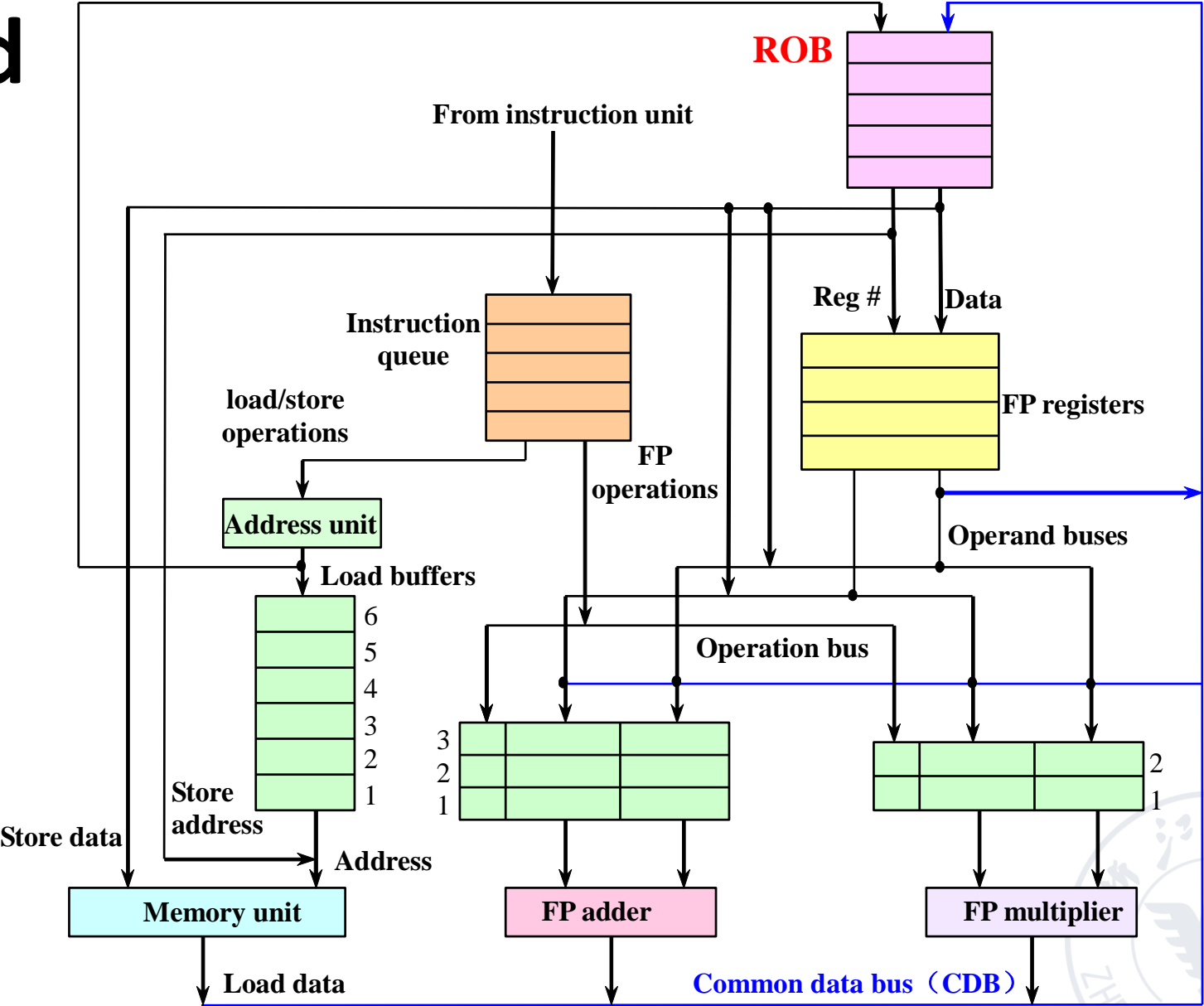
- In-order Issue/Commit, Out-of-Order Execution/Writeback



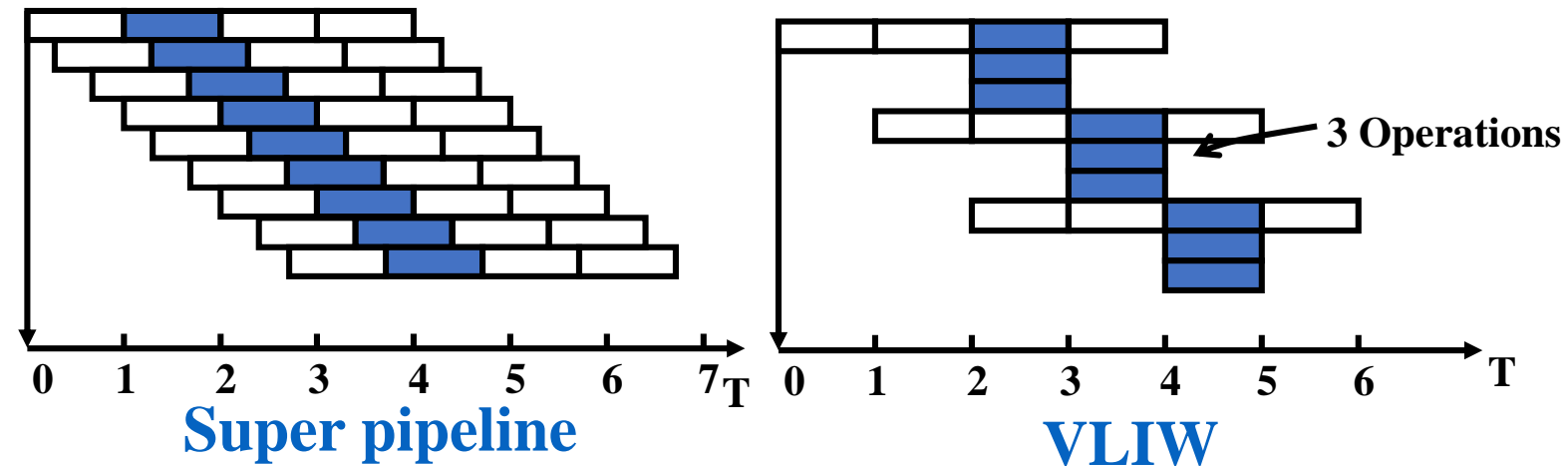
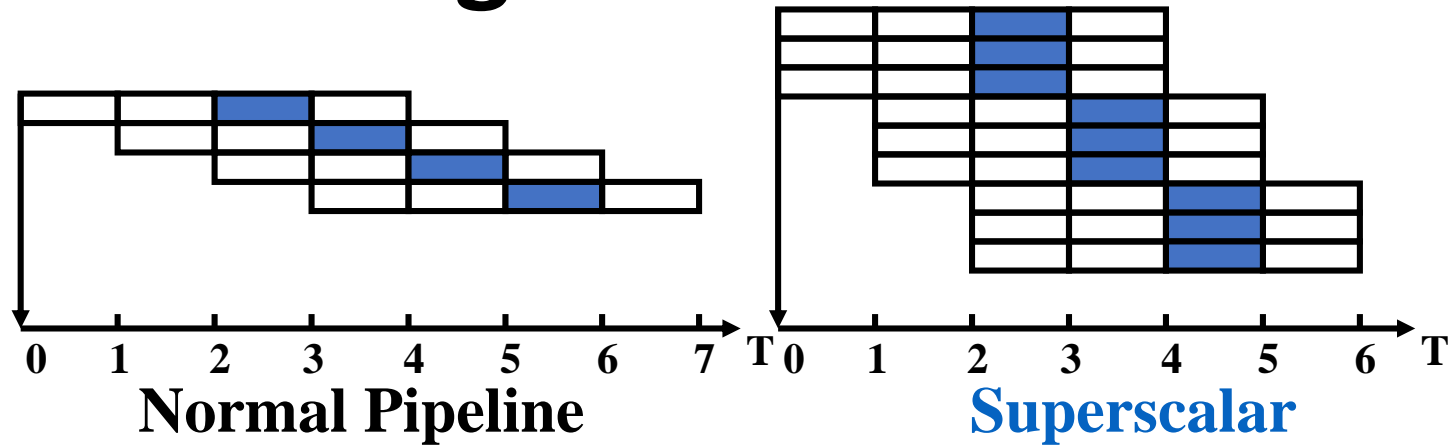
# Hardware-Based Speculation ★

The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation.

Fig.3.16 ★  
Fig.3.17



# Exploiting ILP Using Multiple Issue and Static Scheduling



# Chapter 5(chapter5.1-5.4)

DLP and TLP

# Classes of Parallel Architectures

according to the parallelism  
in the **instruction** and **data** streams  
called for by the instructions:

**SISD**, **SIMD**, **MISD**, **MIMD**

# SISD

- Single instruction stream  
single data stream
- uniprocessor
- Can exploit instruction-level parallelism

# SIMD

- Single instruction stream  
multiple data stream
- The **same instruction** is executed by **multiple processors** using **different data streams**.
- Exploits data-level parallelism
- Data memory for each processor;  
whereas a single instruction memory and control processor.



# MISD

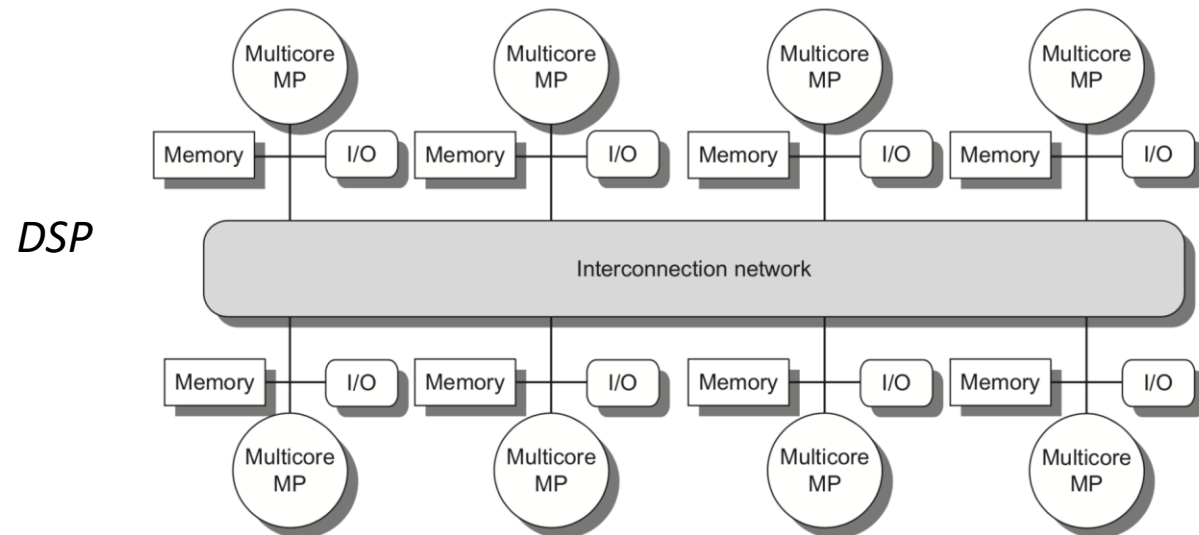
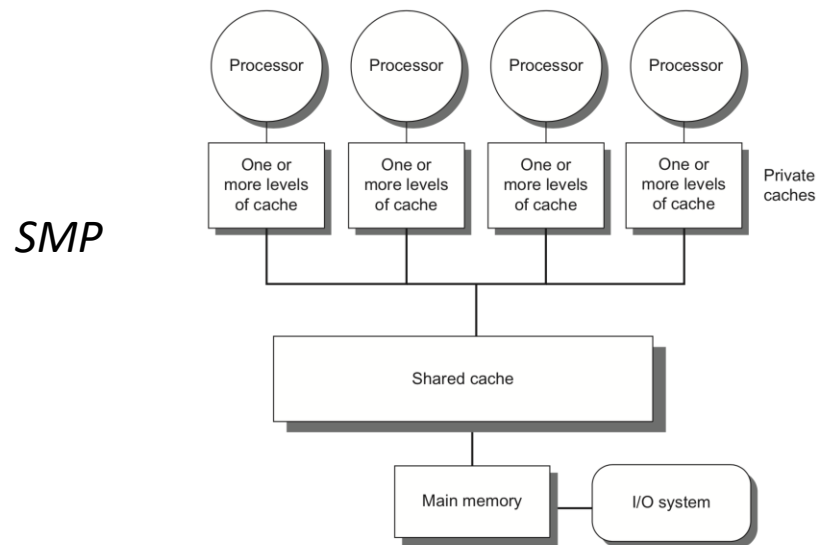
- Multiple instruction streams  
single data stream
- No commercial multiprocessor of this type yet

# MIMD

- Multiple instruction streams  
multiple data streams
- Each processor fetches its own instructions and operates on its own data.
- Exploits task-level parallelism

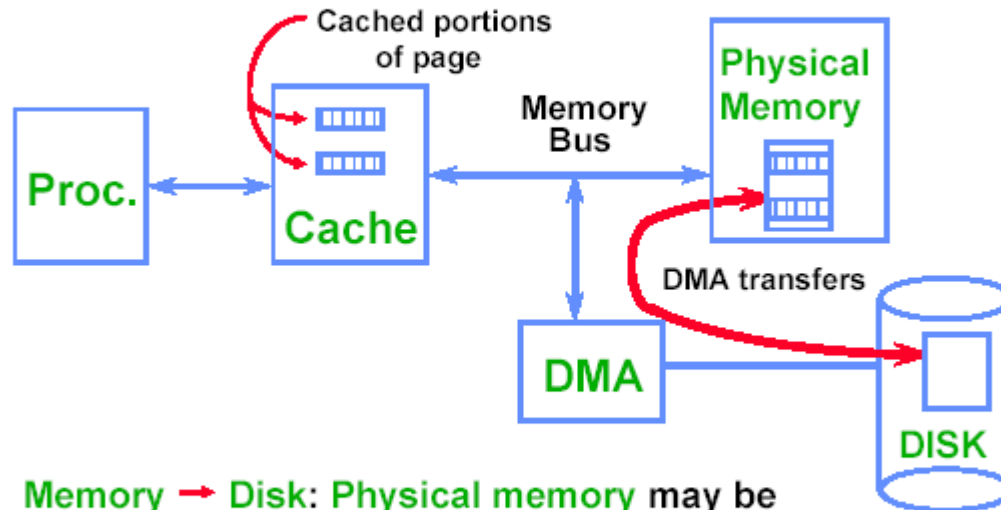
# UMA and NUMA

- UMA is also called *symmetric (shared-memory) multiprocessors (SMP)* or *centralized shared-memory multiprocessors*.
- NUMA is called *distributed shared-memory multiprocessor (DSP)*.



# Cache coherence in uniprocessor

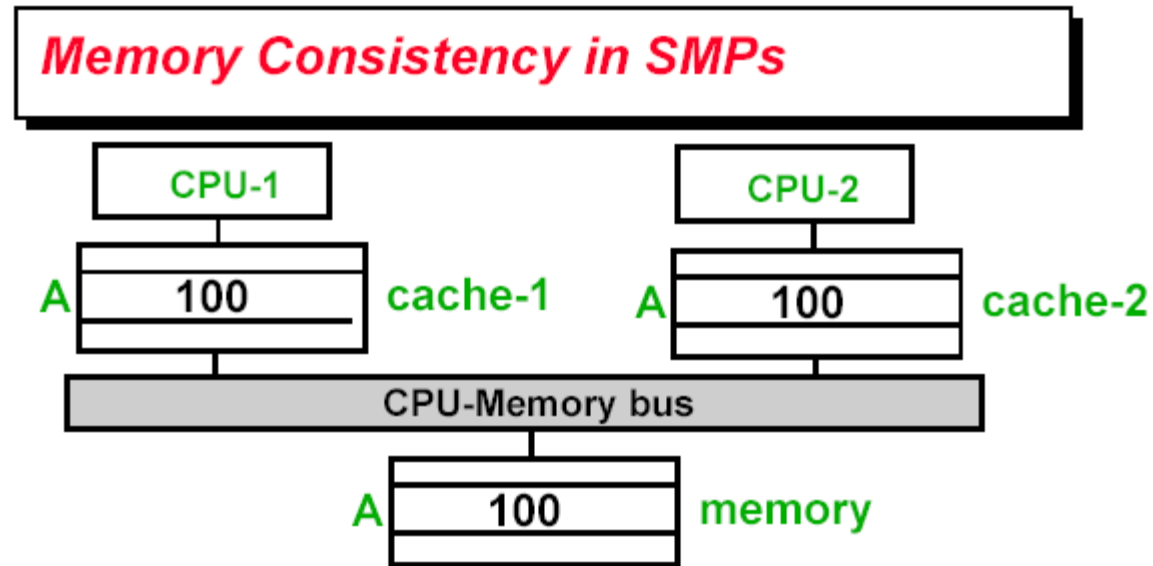
## *Problems with Parallel I/O*



Memory → Disk: Physical memory may be stale if Cache is \_\_\_\_\_.

Disk → Memory: Cache may have data corresponding to \_\_\_\_\_.

# Cache Coherence in Multiprocessor



Suppose CPU-1 updates **A** to 200.

*write-back*: memory and cache-2 have stale values

*write-through*: cache-2 has a stale value

# Cache Coherence Protocols

- **Directory based ( for DSP)**

the sharing status of a particular block of physical memory is kept in one location, called directory

- **Snooping ★ ( for SMP)**

every cache that has a copy of the data from a block of physical memory could track the sharing status of the block

# Snooping Coherence Protocol

- **Write invalidate protocol**

invalidate other copies on a write

exclusive access ensures that no other readable or writable copies of an item exist when the write occurs

# Snooping Coherence Protocol

- **Write invalidate protocol**  
invalidate other copies on a write

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

**write-back cache**

update memory when a block becomes shared simplifies the protocol



# Snooping Coherence Protocol

- **Write update/broadcast protocol**

update all cached copies of a data item when that item is written

consume more bandwidth

**how to implement?**

# Write Invalidate Protocol

**MSI** protocol: three block states ★

- **Invalid**

- **Shared**

indicates that the block in the private cache is potentially shared

- **Modified**

indicates that the block has been updated in the private cache;

implies that the block is **exclusive**

# Write Invalidate Protocol

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or <i>ownership</i> misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared. write-back block
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

# Write Invalidate Protocol

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or <i>ownership</i> misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared. write-back block
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

replacement

write-back cache

# Write Invalidate Protocol

figure 5.4

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or <i>ownership</i> misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared. write-back block
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

update memory to simplify protocol

# Write Invalidate Protocol

figure 5.6

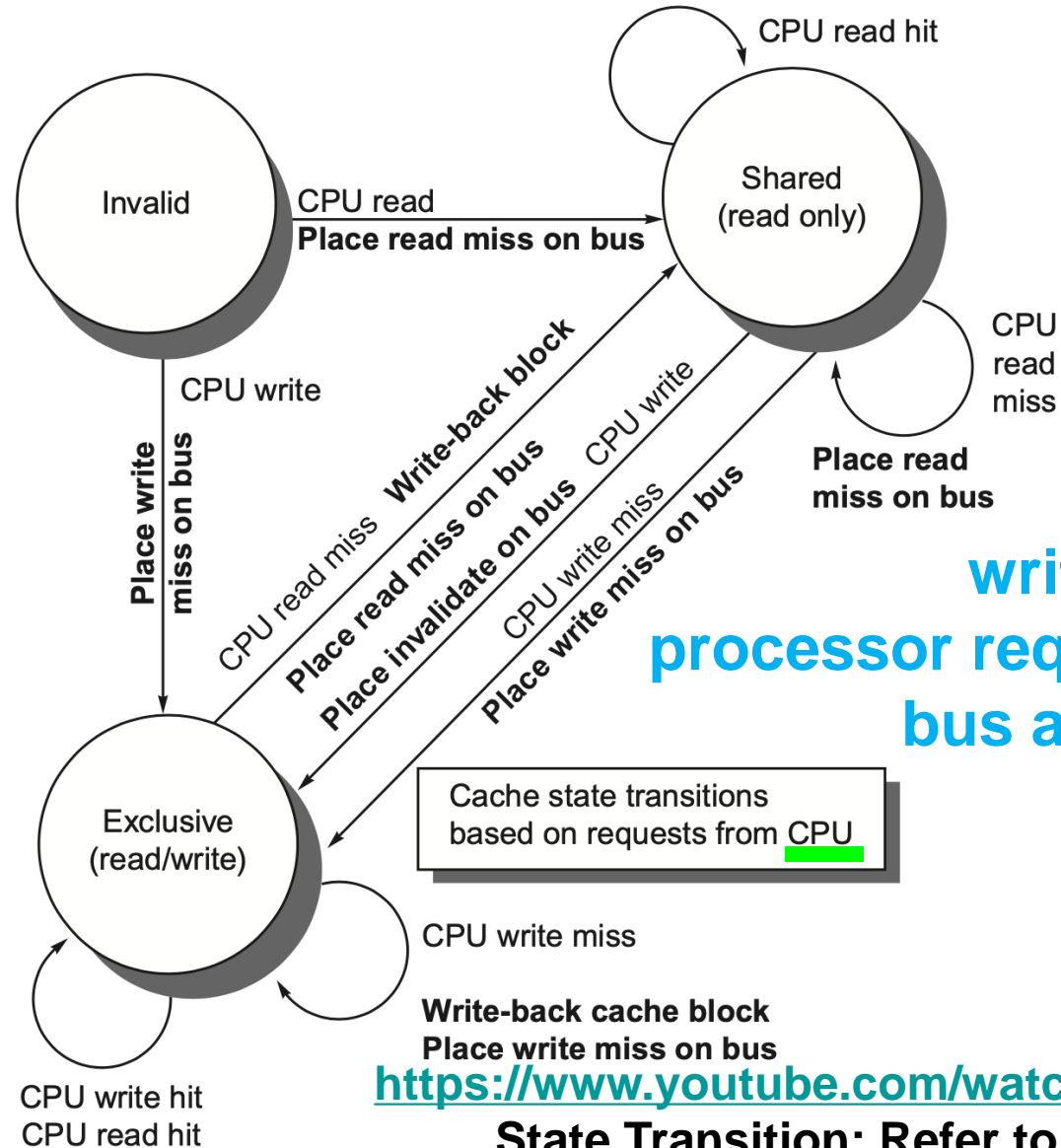
Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or <i>ownership</i> misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared. write-back block
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

need also place cache block on bus to service write miss

**how do states transit?**



# Write Invalidate Protocol



write-back cache  
processor requests on arcs  
bus actions in bold

Fig.5.6

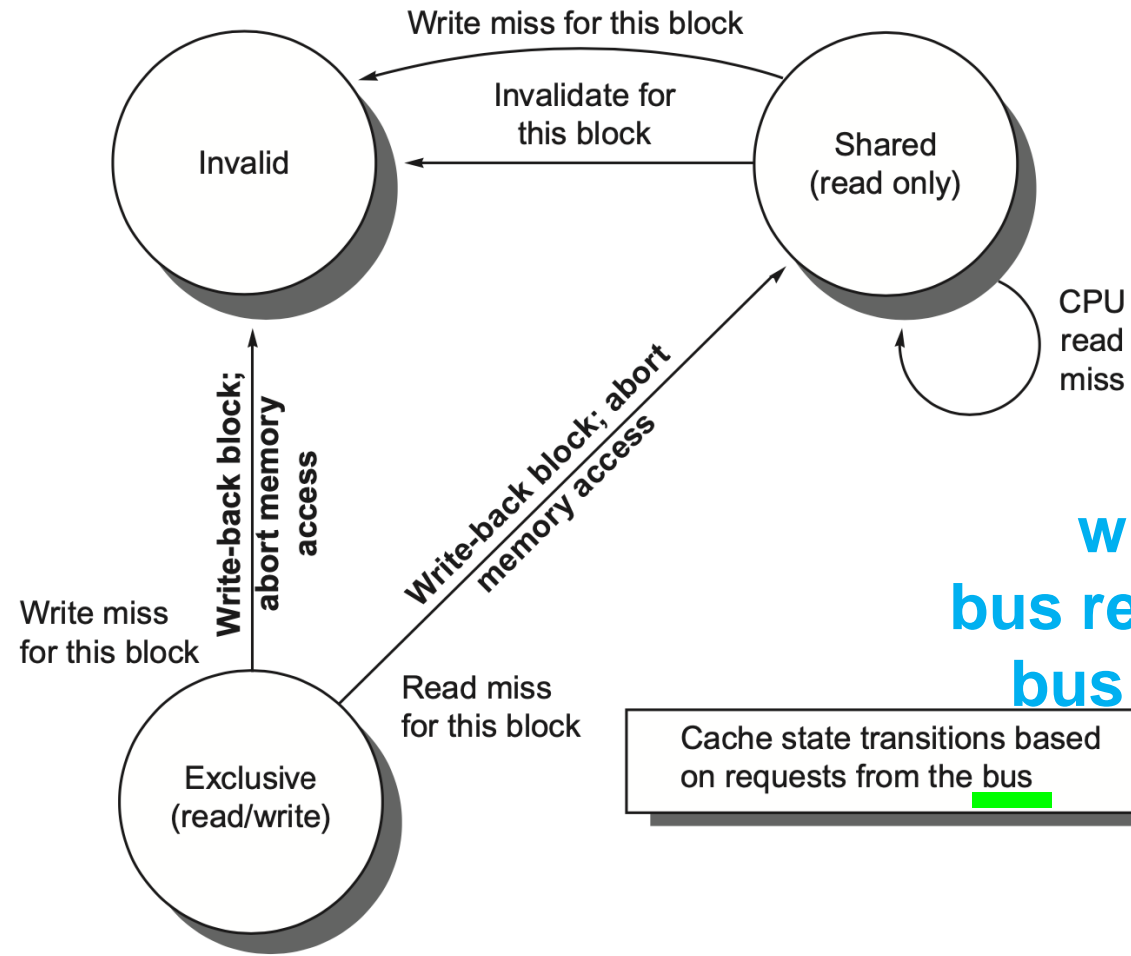


Fig.5.37

<https://www.youtube.com/watch?v=gAUVAel-2Fg>

State Transition: Refer to the previous table

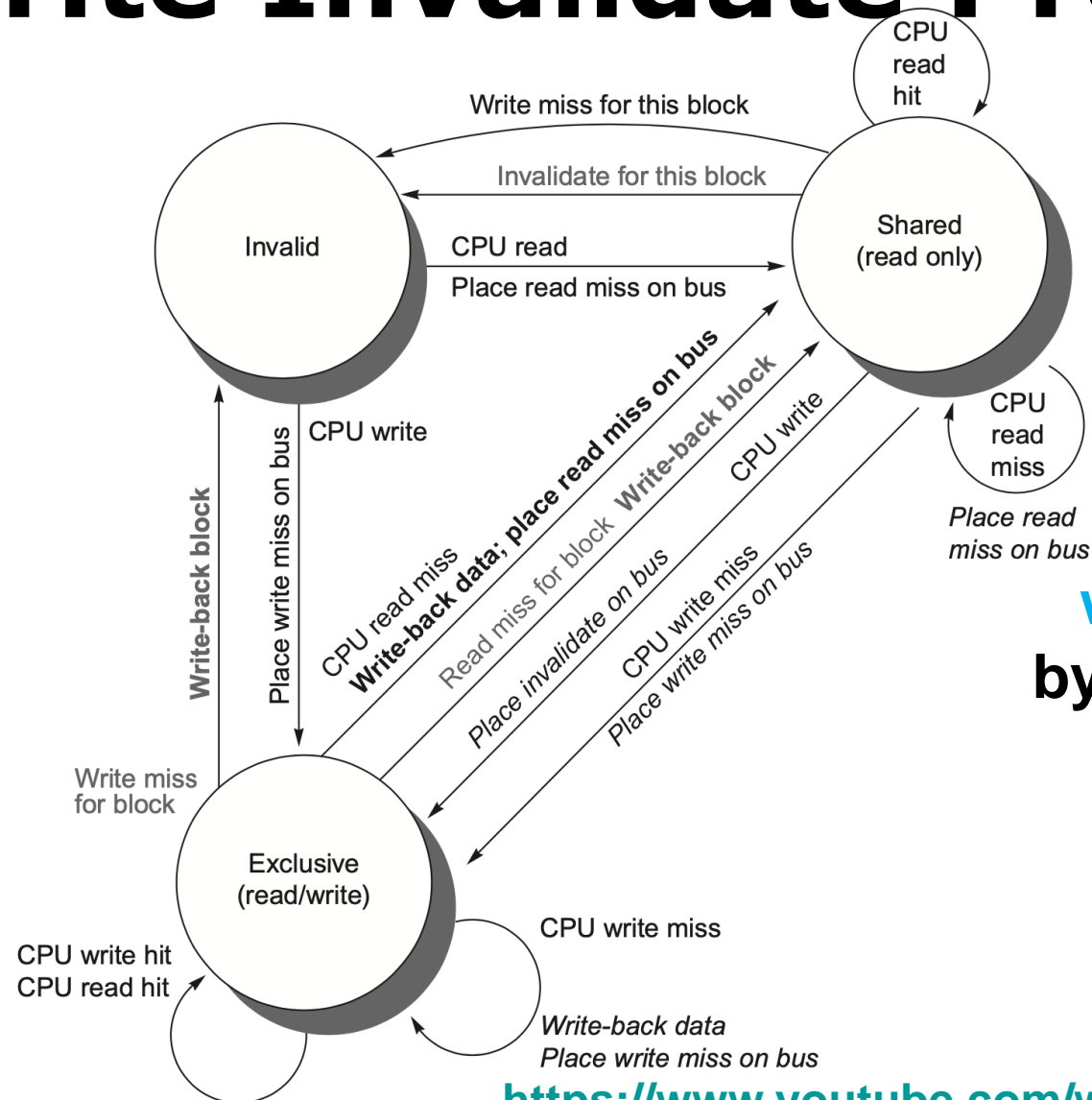
# Write Invalidate Protocol



**write-back cache**  
**bus requests on arcs**  
**bus actions in bold**

## Cache state transitions based on requests from the bus

# Write Invalidate Protocol



**write-back cache**  
by local processor  
by bus activities

<https://www.youtube.com/watch?v=gAUVAel-2Fg>

State Transition: Refer to the previous table

# MSI Extensions: MESI

- **MESI**

**exclusive:** indicates when a cache block is resident only in a single cache but is clean

exclusive->read by others->shared;

exclusive->write->modified

# MSI Extensions: MESI

- **MESI**

**exclusive:** indicates when a cache block is resident only in a single cache but is clean

exclusive->read by others->shared;

exclusive->write->modified (without generating any invalidates)

writing an E-block need not invalidate on bus

# MSI Extensions: MOESI

- **MOESI**

**owned:** indicates that the associated block is owned by that cache and out-of-date in memory

modified -> owned upon a read miss on bus

# MSI Extensions: MOESI

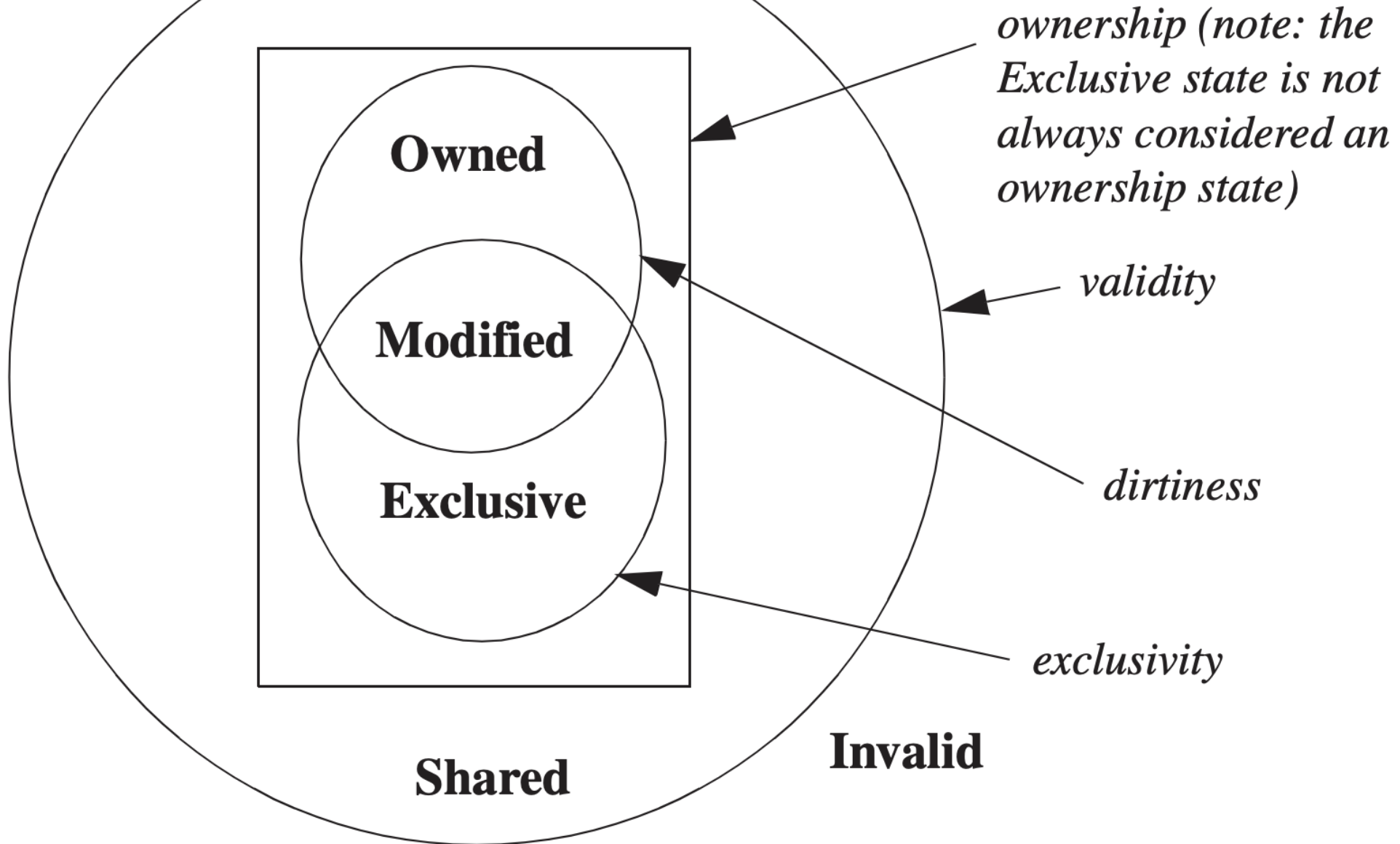
- **MOESI**

**owned:** indicates that the associated block is owned by that cache and out-of-date in memory

modified -> owned upon a read miss on bus

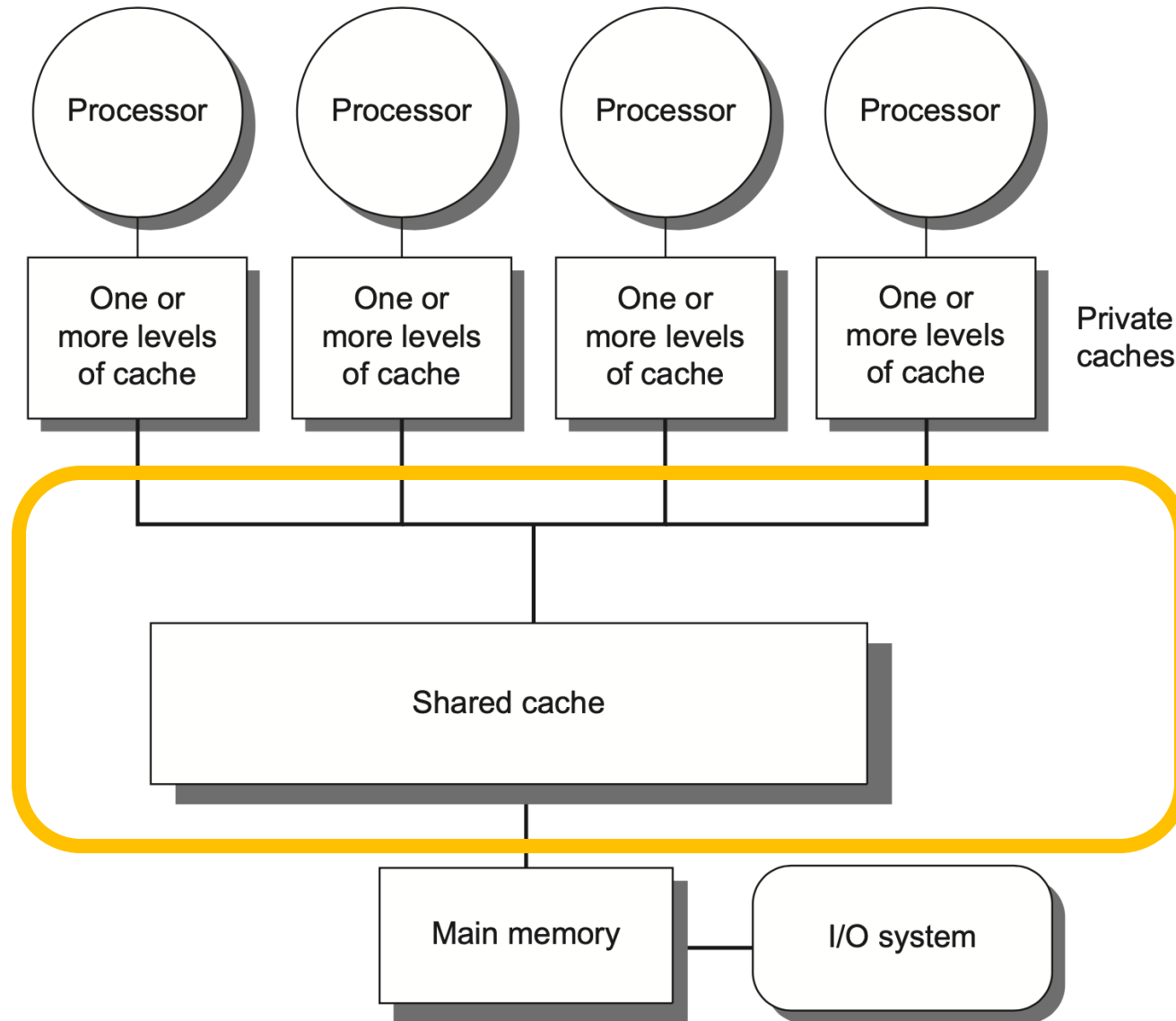
without writing the shared block to memory

# MSI Extensions: MOESI





# Centralized Shared-Memory



# Coherence Miss

- **True sharing miss**

**case 1: first write by a processor to a shared cache block** causes an invalidation to establish ownership of that block;

**case 2:** another processor reads a modified word in that cache block;

- **False sharing miss**

# Coherence Miss

- True sharing miss
- False sharing miss

a single valid bit per cache block;

occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into

# Coherence Miss

- **Example**

assume words x1 and x2 are in the same cache block, which is in shared state in the caches of both P1 and P2.

identify each miss as a true sharing miss, a false sharing miss, or a hit?

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

# Coherence Miss

- **Example**

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

## 1. true sharing miss

since x1 was read by P2 and needs to be invalidated from P2

# Coherence Miss

- **Example**

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

## 2. false sharing miss

since x2 was invalidated by the write of x1 in P1,  
but that value of x1 is not used in P2;

# Coherence Miss

- **Example**

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

### 3. false sharing miss

since the block is in shared state, need to invalidate it to write;

but P2 read x2 rather than x1;

# Coherence Miss

- **Example**

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

## 4. false sharing miss

need to invalidate the block;

P2 wrote x2 rather than x1;



# Coherence Miss

- **Example**

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

## 5. true sharing miss

since the value being read was written by P2 (invalid  
-> shared)

## Others

- memory consistency model
- Spin lock

**The end is the beginning**