# Course Review

# Chapter 1 (6th 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

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

$$CPU\ Clock\ Cycles = Instructions\ for\ a\ Program \times Average\ Clock\ Cycles\ Per\ Instruction$$

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

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

# 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:

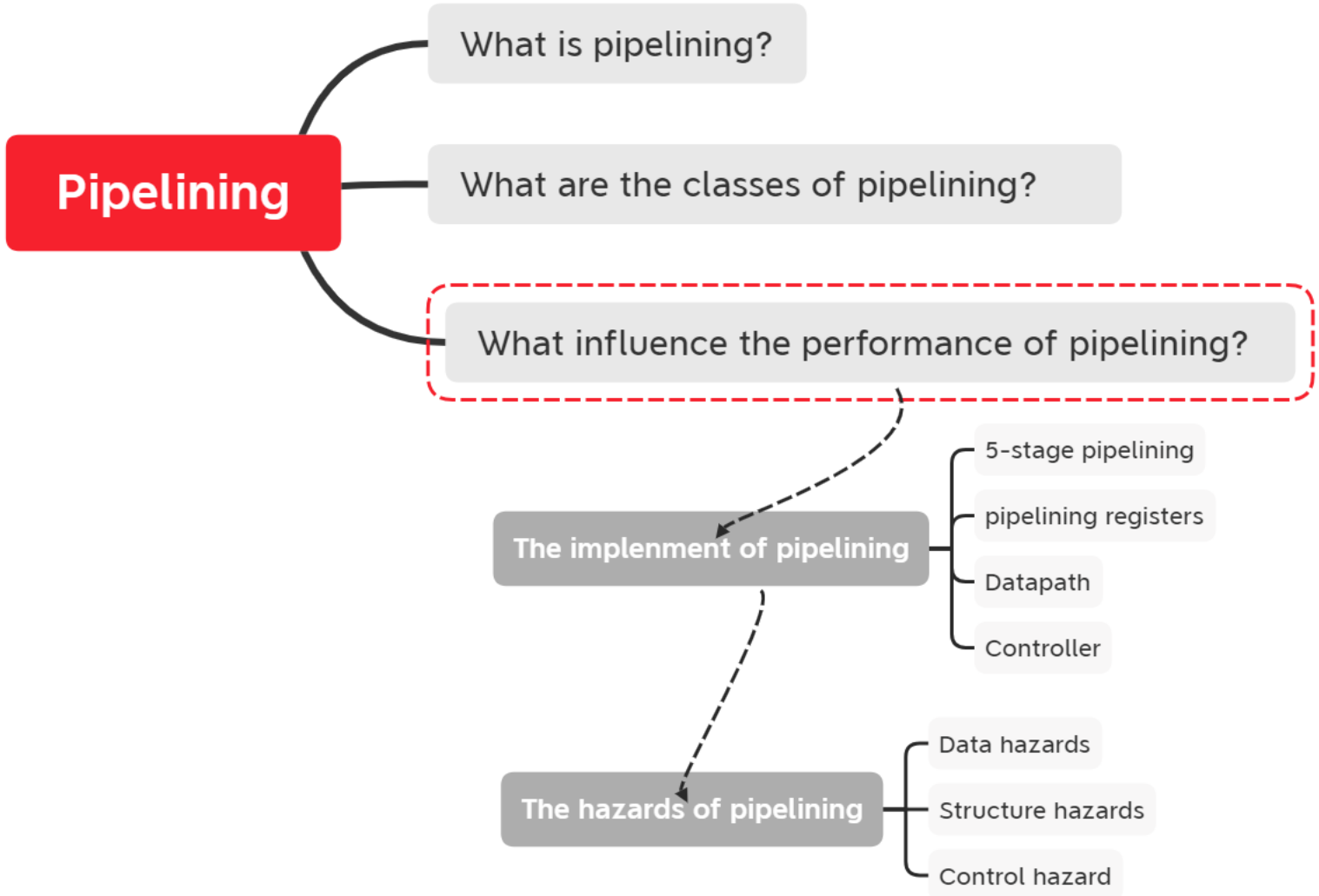
- The fraction of the time the enhancement can be exploited.
- The improvement gained by the enhancement while it is exploited.

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

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

# Chapter 2 (6th Appendix C)

Pinelining


Pipelining
What is pipelining?
What are the classes of pipelining?
What influence the performance of pipelining?
The implenment of pipelining
5-stage pipelining
pipelining registers
Datapath
Controller
The hazards of pipelining
Data hazards
Structure hazards
Control hazard

# Instruction Set Architecture

**ISA**

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

# 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

# The Classic Five-Stage Pipeline for a RISC Processor

- A Simple Implementation of RISC-V
- Instructions Dependences

**Dependences are a property of *programs.***

- 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.***

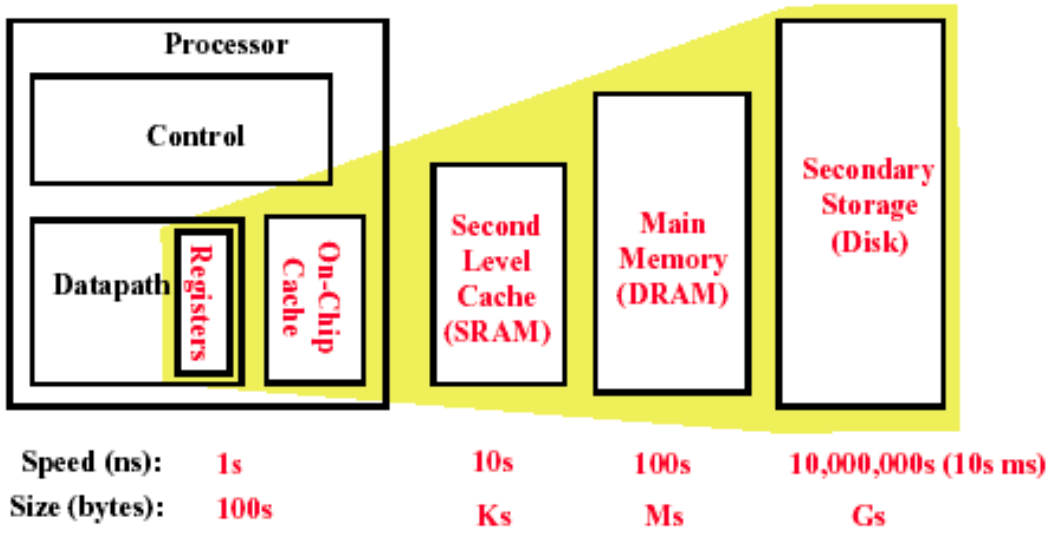
# Chapter 3 (6th chapter2 & Appendix B)

Memory Hierarchy

# Memory Hierarchy of a Modern Computer System

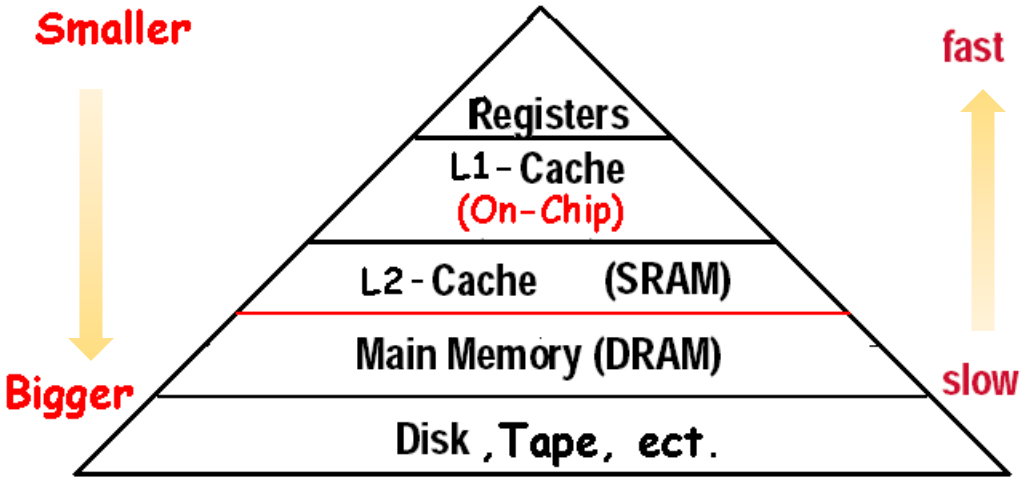
By taking advantage of the principle of locality:

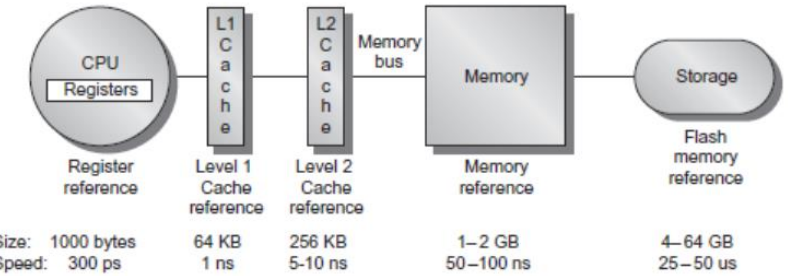
- Present the user with as much memory as is available in the cheapest technology.
- Provide access at the speed offered by the fastest technology.



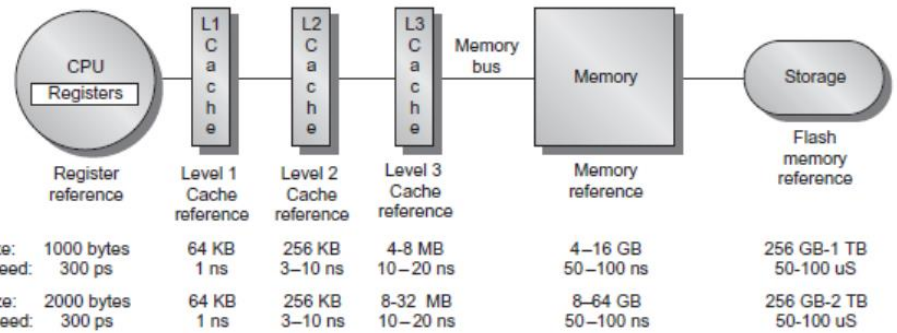
# What is a cache?

- 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?

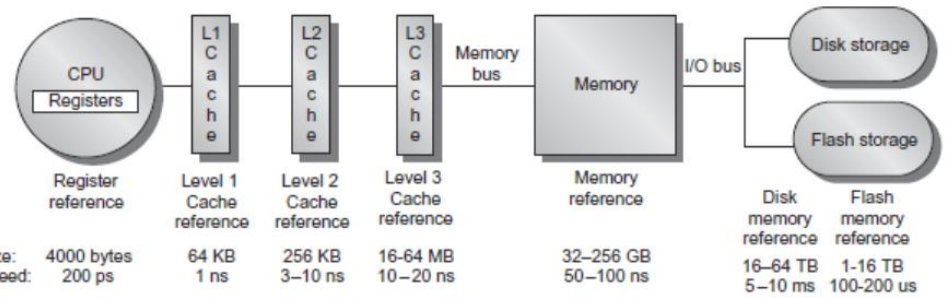





(A) Memory hierarchy for a personal mobile device




(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 ★

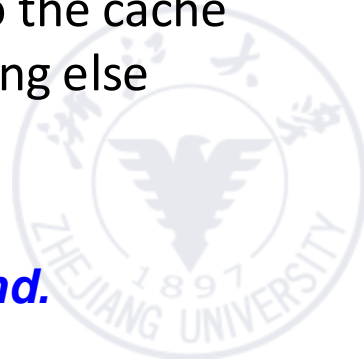
- When data is written into the cache (on a store), is the data also written to main memory?
  - If the data is written to memory, the cache is called a write-through cache
    - Can always discard cached data - most up-to-date data is in memory
    - Cache control bit: only a valid bit
    - memory (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 | Write Cache ★ | |
|---|---|---|
| | hit | miss |
| Read through | ***Write-through***<br>• write is done synchronously both to the cache and to the backing store<br>• the data is written to memory | ***No-write allocate (write around)***<br>• The block is only written to main memory<br>• It is not stored in the cache. |
| Read allocate | ***Write-back***<br>• writing is done only to the cache<br>• the data is NOT written to memory | ***Write allocate***<br>• The block is loaded into the cache on a miss before anything else occurs. |

***In general, write-back caches use write-allocate , and write-through caches use write-around.***



# 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

$$\text{Average Memory Access Time} = \frac{\text{Whole accesses time}}{\text{All memory accesses in program}}$$

$$= \frac{\text{Accesses time on hitting+ Accesses time on miss}}{\text{All memory accesses in program}}$$

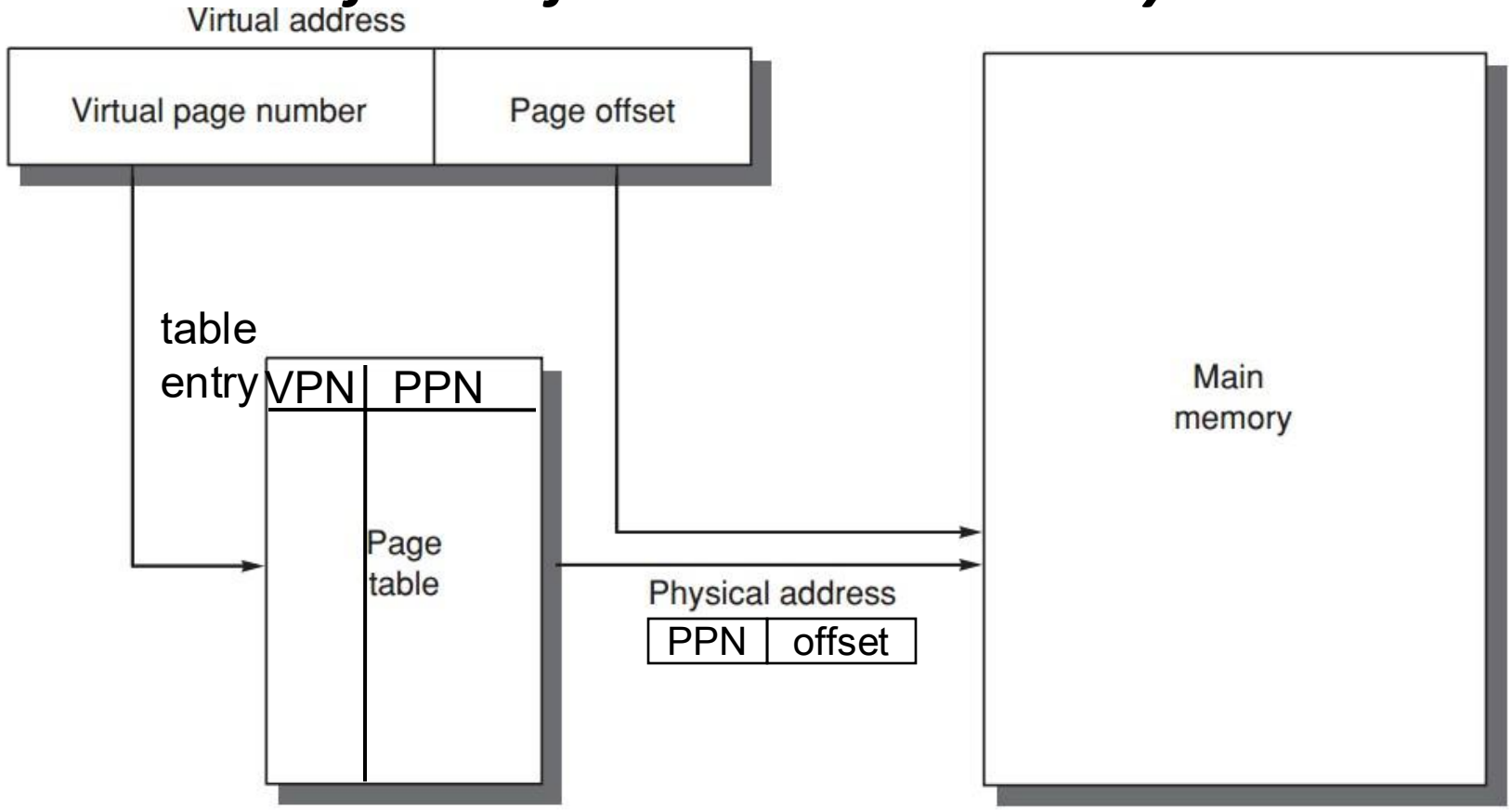
$$= \text{Hit time} + (\text{Miss Rate} \times \text{Miss Penalty})$$

$$= \left(HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}\right) \times Inst\%$$
$$\left(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data}\right) \times Data\%$$

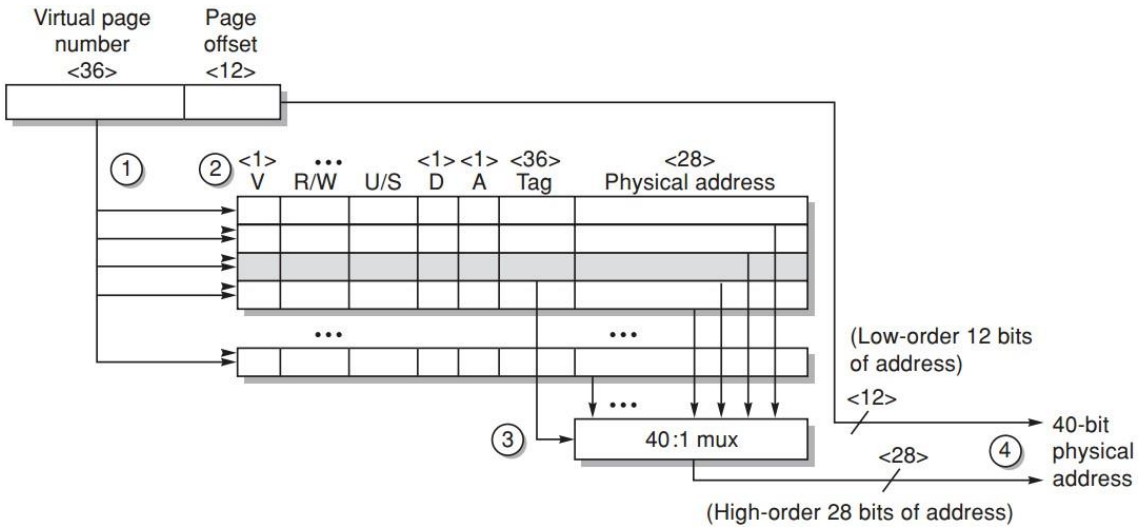
$$CPUtime = IC \times \left( \frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

# Four Mem Hierarchy Q's

- **Q1.** ***Where can a block be placed in main memory?***
- **Q2.** ***How is a block found if it is in main memory?***



# What is TLB ★




- **Translation lookaside buffer (TLB)**

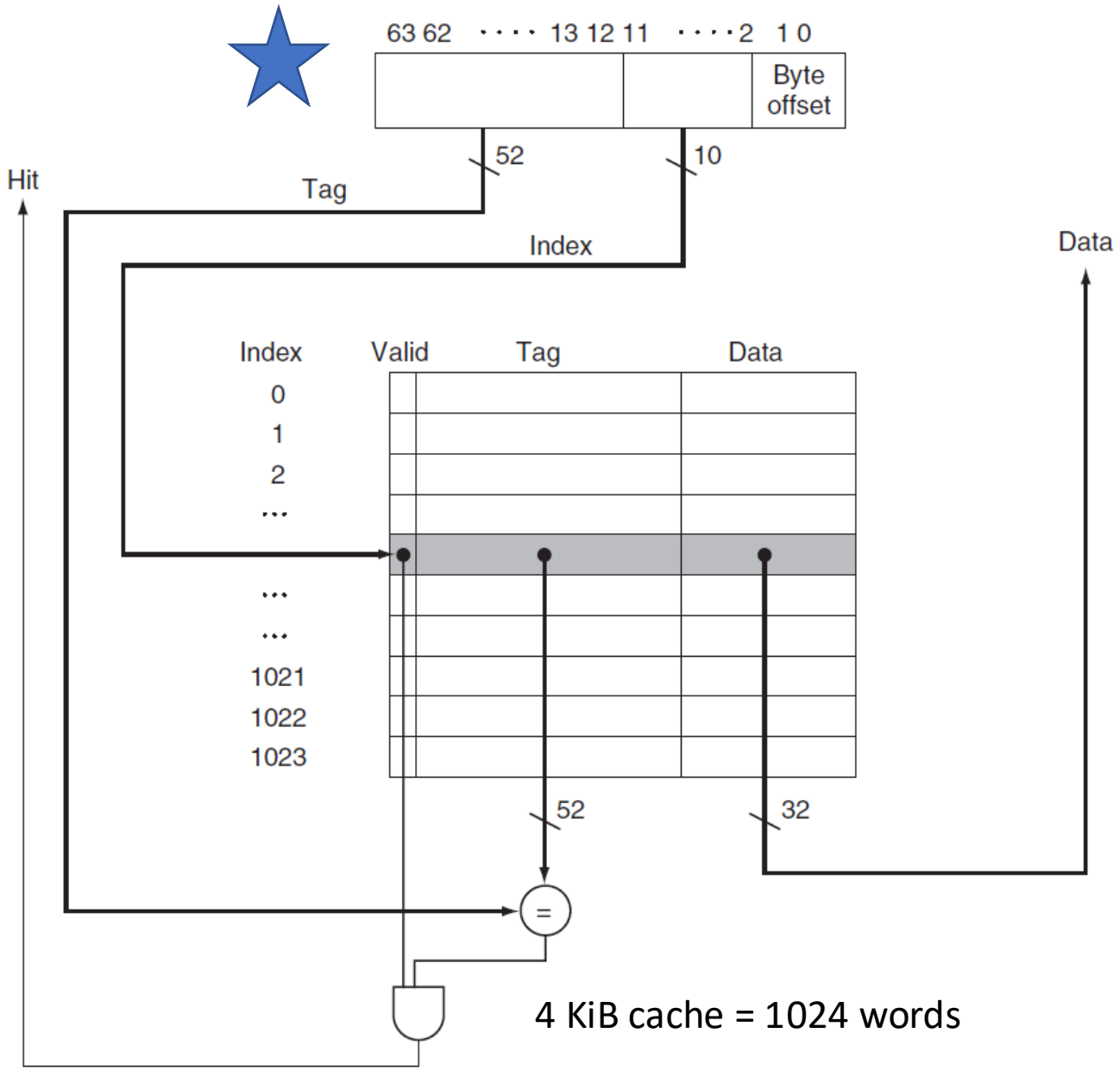
  a special **cache!**

  that keeps (prev) address translations

- TLB entry （full associated cache）

  --tag: virtual page number of the virtual address;

  --data: a physical page frame number, protection field, valid bit, use bit, dirty bit;




4 KiB cache = 1024 words

The size of the block was one word (4 bytes).

- 64-bit addresses
- A direct-mapped cache
- The cache size is $2^n$ blocks, so $n$ bits are used for the index
- The block size is
  - ✓ $2^m$ words ($2^{m+2}$ bytes = $2^{m+5}$ bits)
  - ✓ $m$ bits are used for the word within the block, and two bits are used for the byte part of the address

tag $64 - (n + m + 2)$

cache $2^n \times$ (block size+ tag size + Valid size).

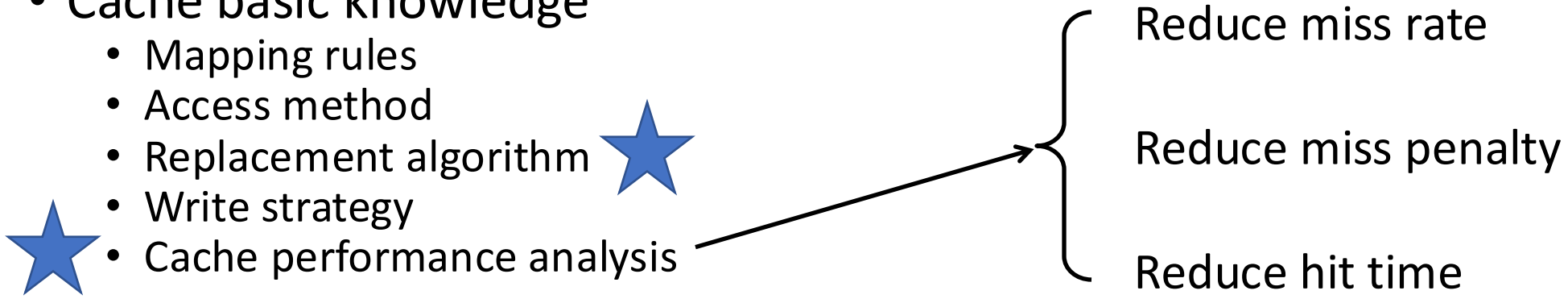
$= 2^n \times (2^m \times 32 + (64 - n - m - 2) + 1)$

$= 2^n \times (2^m \times 32 + 63 - n - m)$.

actual size/complete cache size

# 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




- Memory (the influence of memory organization structure on Cache failure rate)

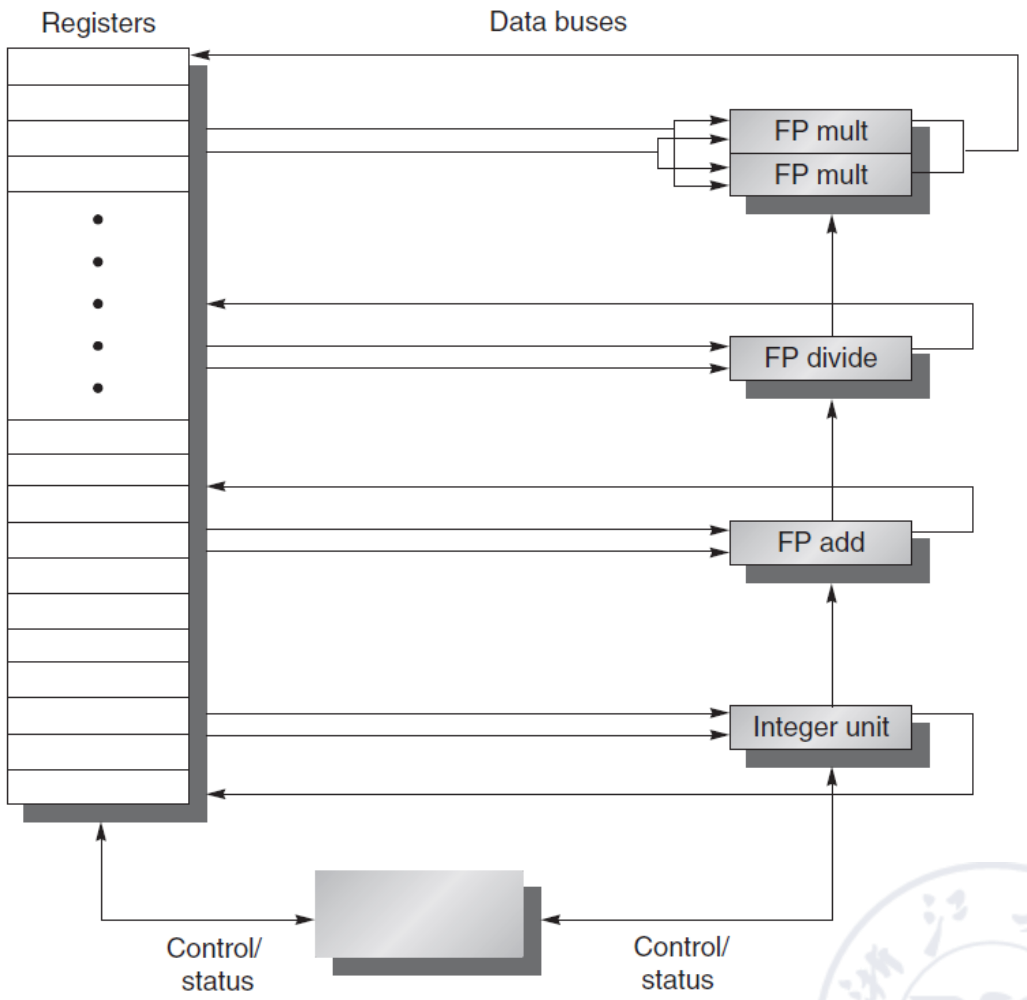
# Chapter 4(chapter3)

ILP

# Dynamic Scheduling

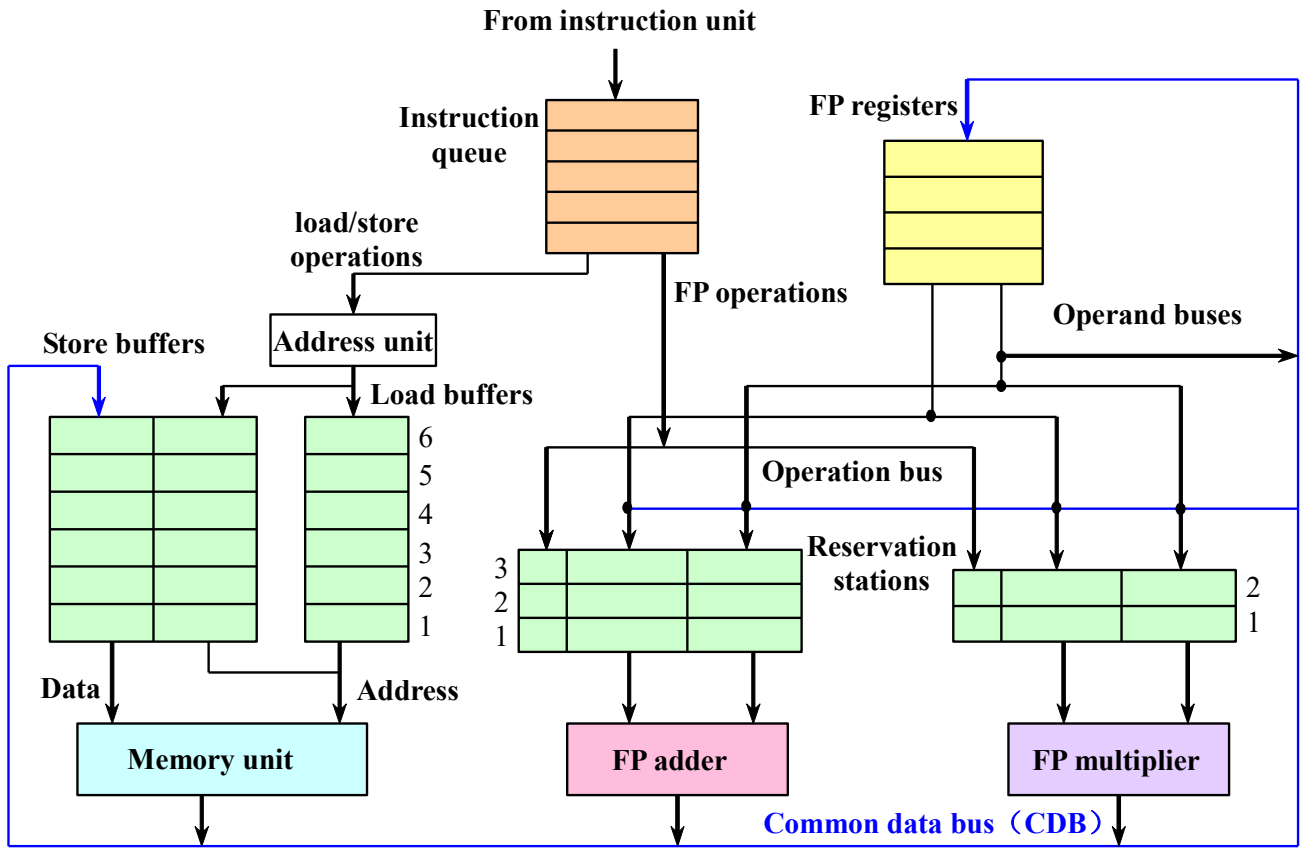
Idea: Dynamic Scheduling

Method: out-of-order execution




*Dynamic Scheduling with a Scoreboard*

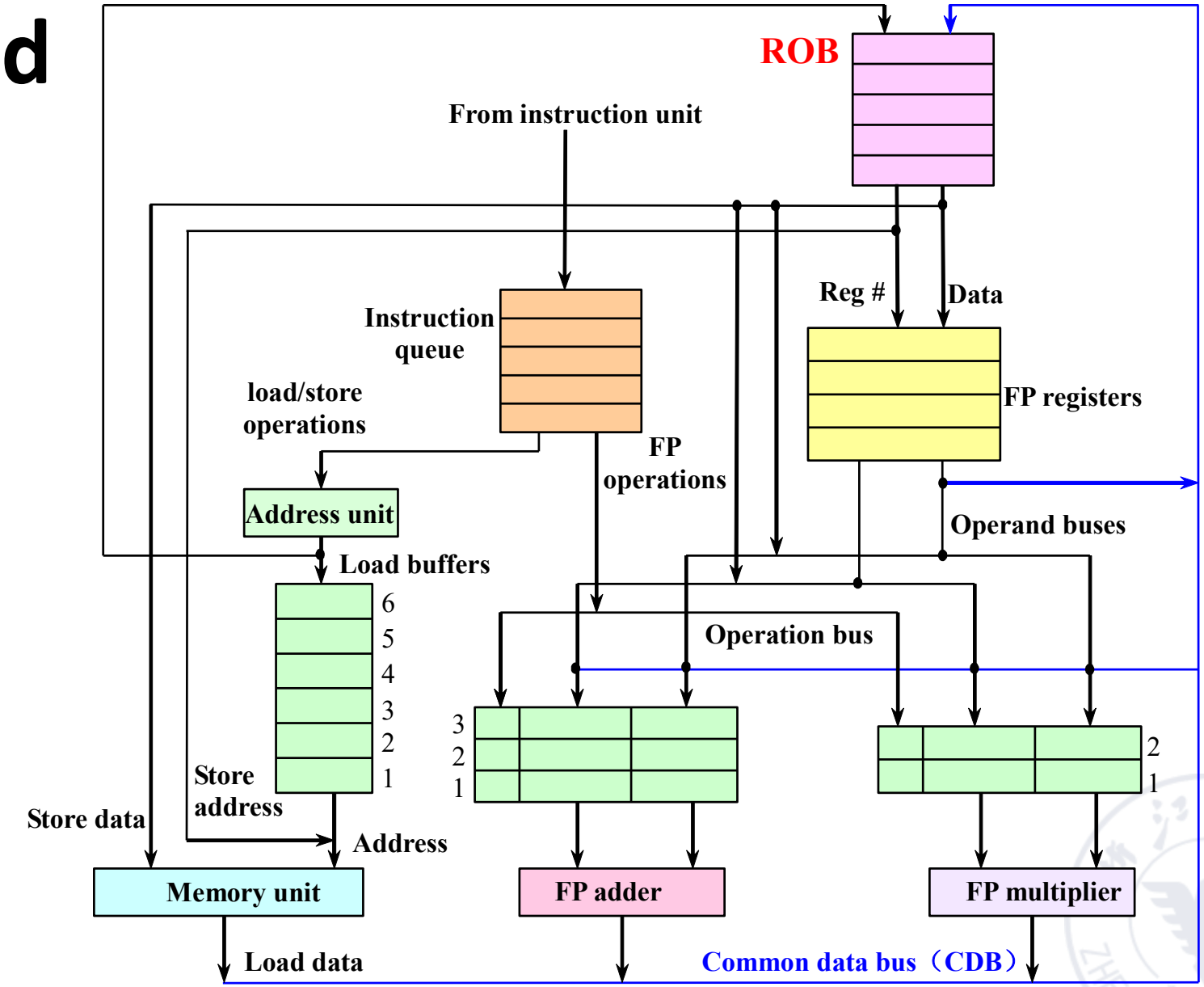
# Tomasulo's Approach




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

# Hardware-Based Speculation ★

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

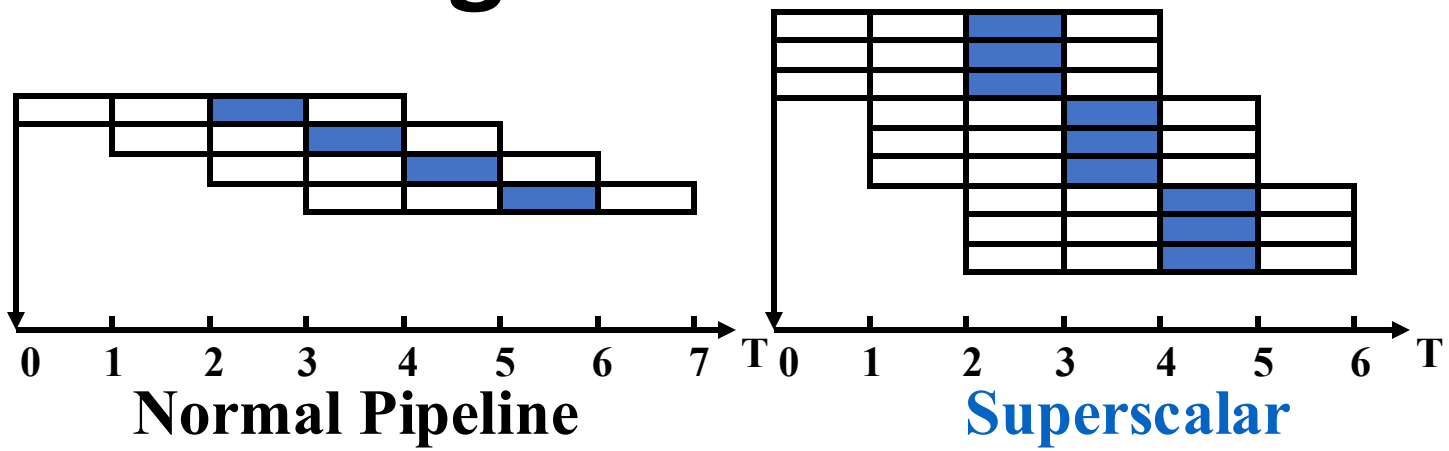


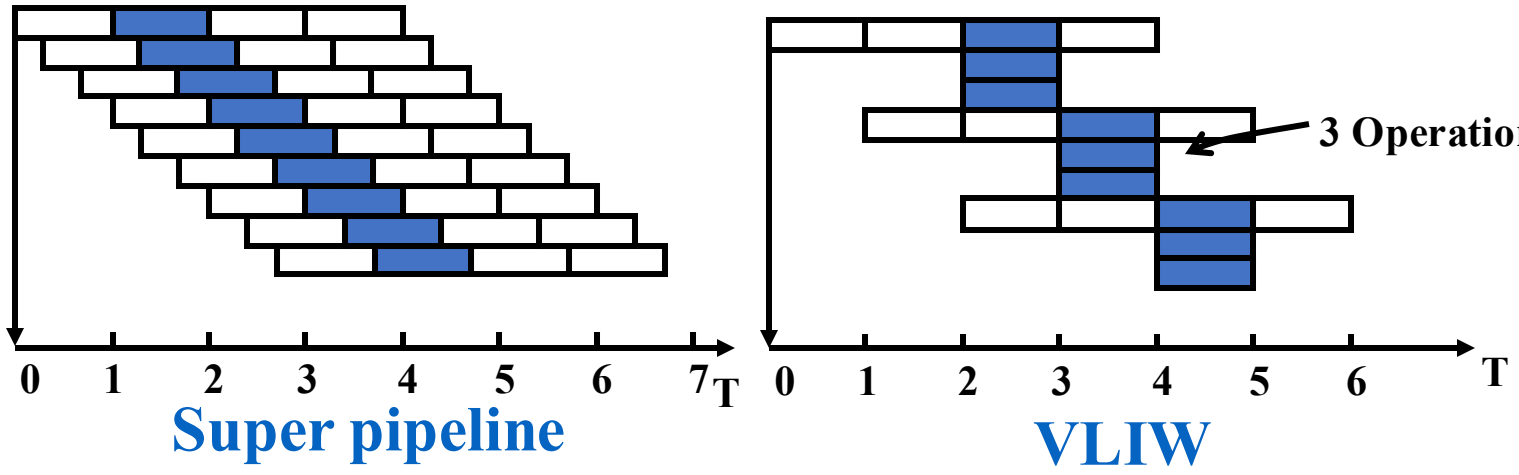
# Tomasulo with Reorder Buffer - Summary

| Instruction | Issue | Exec Comp | Writeback | Commit |
|---|---|---|---|---|
| FLD F6, 34(R2) | 1 | 3 | 4 | 5 |
| FLD F2, 45(R3) | 2 | 4 | 5 | 6 |
| FMUL.D F0, F2, F4 | 3 | 6-15 | 16 | 17 |
| FSUB.D F8, F6, F2 | 4 | 6-7 | 8 | 18 |
| FDIV.D F10, F0, F6 | 5 | 17-56 | 57 | 58 |
| FADD.D F6, F8, F2 | 6 | 9-10 | 11 | 59 |

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

# 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.

# Flynn ★

**Computer Architecture**

- **SISD** — von Neumann Architecture
- **SIMD**
  - Vector Processor
  - Array Processor
- **MIMD**
  - Shared Memory
    - NUMA
      - CC-NUMA
      - NC-NUMA
    - COMA
    - UMA
  - Message Passing
    - Massive Parallel Processor
    - Cluster Of Workstation
      - Cluster
      - Grid
- **MISD** — Unknown

# Vector Processor ★

Vertical and horizontal (grouping) processing method

- The calculation process is:
  - Calculate the first group: $d_{1\sim n} \leftarrow a_{1\sim n} \times (b_{1\sim n} + c_{1\sim n})$
  - Calculate the second group: $d_{(n+1)\sim 2n} \leftarrow a_{(n+1)\sim 2n} \times (b_{(n+1)\sim 2n} + c_{(n+1)+2n})$
  - ...
  - Final calculation the last group: $d_{(s*n+1)\sim N} \leftarrow a_{(s*n+1)\sim N} \times (b_{(s*n+1)\sim N} + c_{(s*n+1)\sim N})$

- The length of the operand vectors.
- Structural hazards among the operations.
- The data dependences.

# Improve the Performance of Vector Processor

Use **vector chaining technology** to speed up the execution of a string of vector instructions.

Vector architects call forwarding of element dependent operations chaining, in that the dependent operations are "chained".

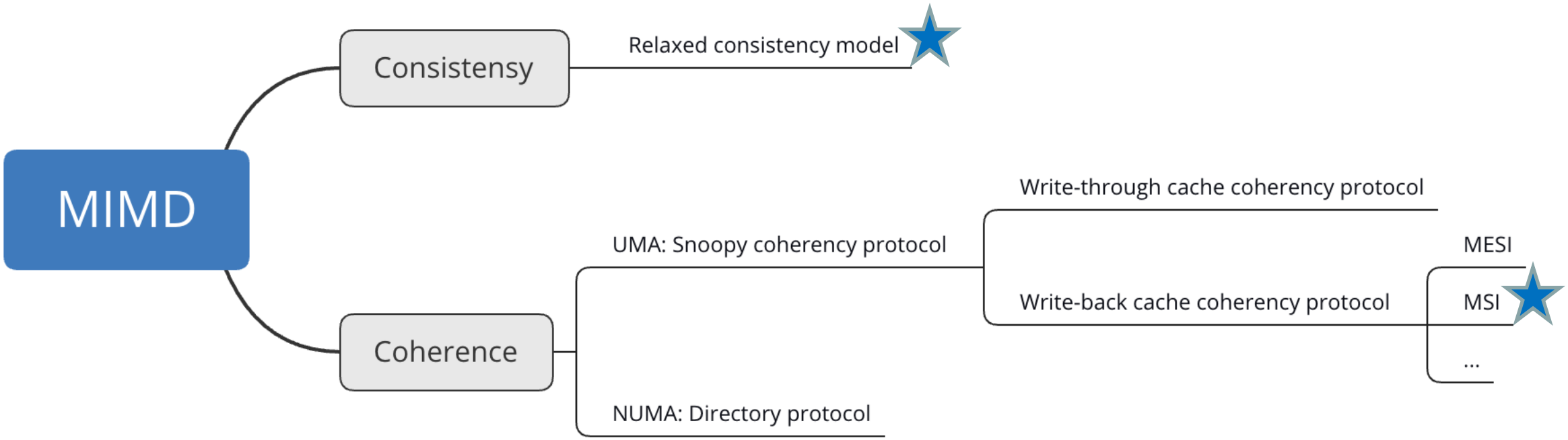
It can reduce pipeline stalls, and improve performance.

# MISD

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

# MIMD ★

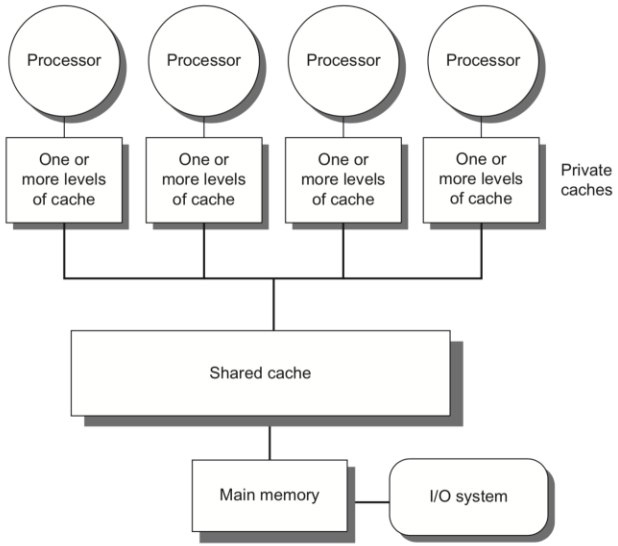
- Multiple instruction streams multiple data streams
- Each processor fetches its own instructions and operates on its own data.
- Exploits task-level parallelism


MIMD
Consistensy
Relaxed consistency model
Coherence
UMA: Snoopy coherency protocol
Write-through cache coherency protocol
Write-back cache coherency protocol
MESI
MSI
...
NUMA: Directory protocol

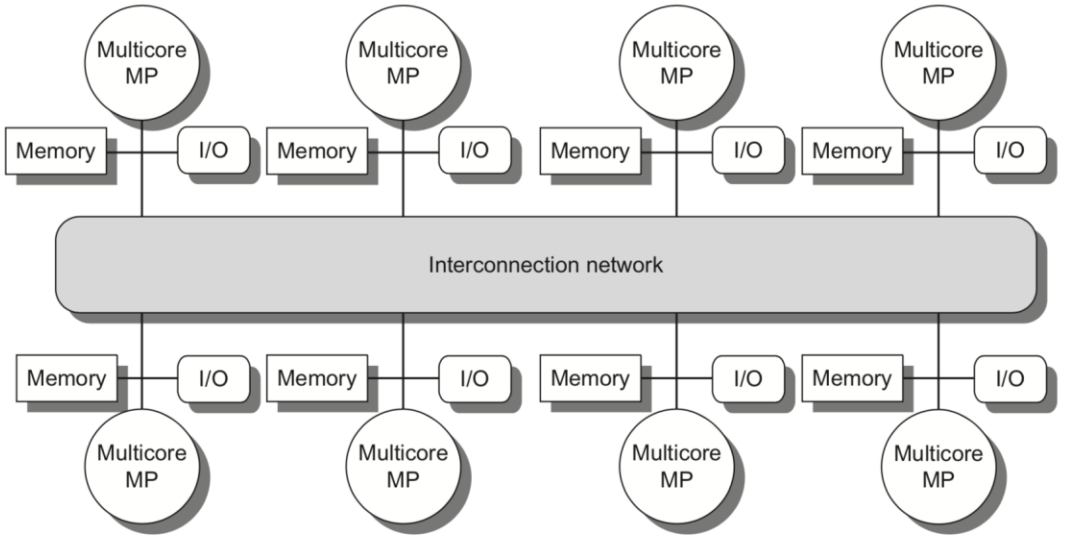
# UMA and NUMA ★

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

*SMP*




*DSP*



# Memory consistency and Cache coherence 

- Cache coherence defines *the behavior of reads and writes to the same memory location.*

- Memory consistency defines *the behavior of reads and writes with respect to accesses to other memory locations.*

# Cache Coherence Protocols ★

- **Directory based （ for DSP）**

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

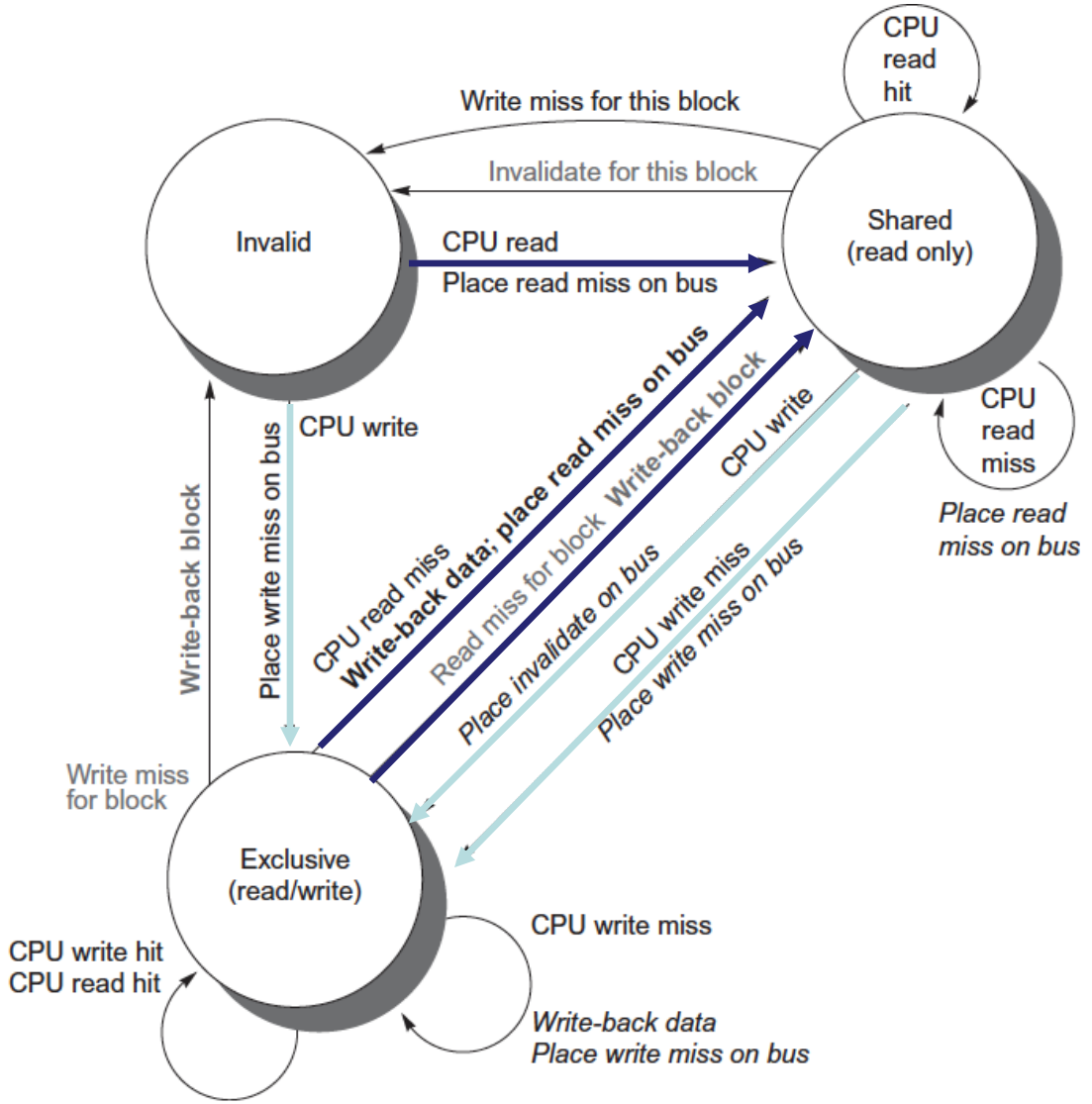
- **Snooping （ for SMP)**

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

# Snooping Cache Coherence Protocol

**MSI protocol**
- ✓ **Invalid**
- ✓ **Shared**
- ✓ **Modified**

implies that the block is **exclusive**



# MSI protocol

The initial cache and memory state for a shared memory multi-processor system are shown in the following figure. Each core processor has a **4-line direct-mapped write back** cache. Basic **write invalidation snooping protocol** is used. Please show the action results (ie. updated parts in the caches and memory) for the three action as that in the example, assuming that all the actions are applied to the initial cache and memory states in the figure. In cache state, I means Invalid, S means Shared, M means Modified.

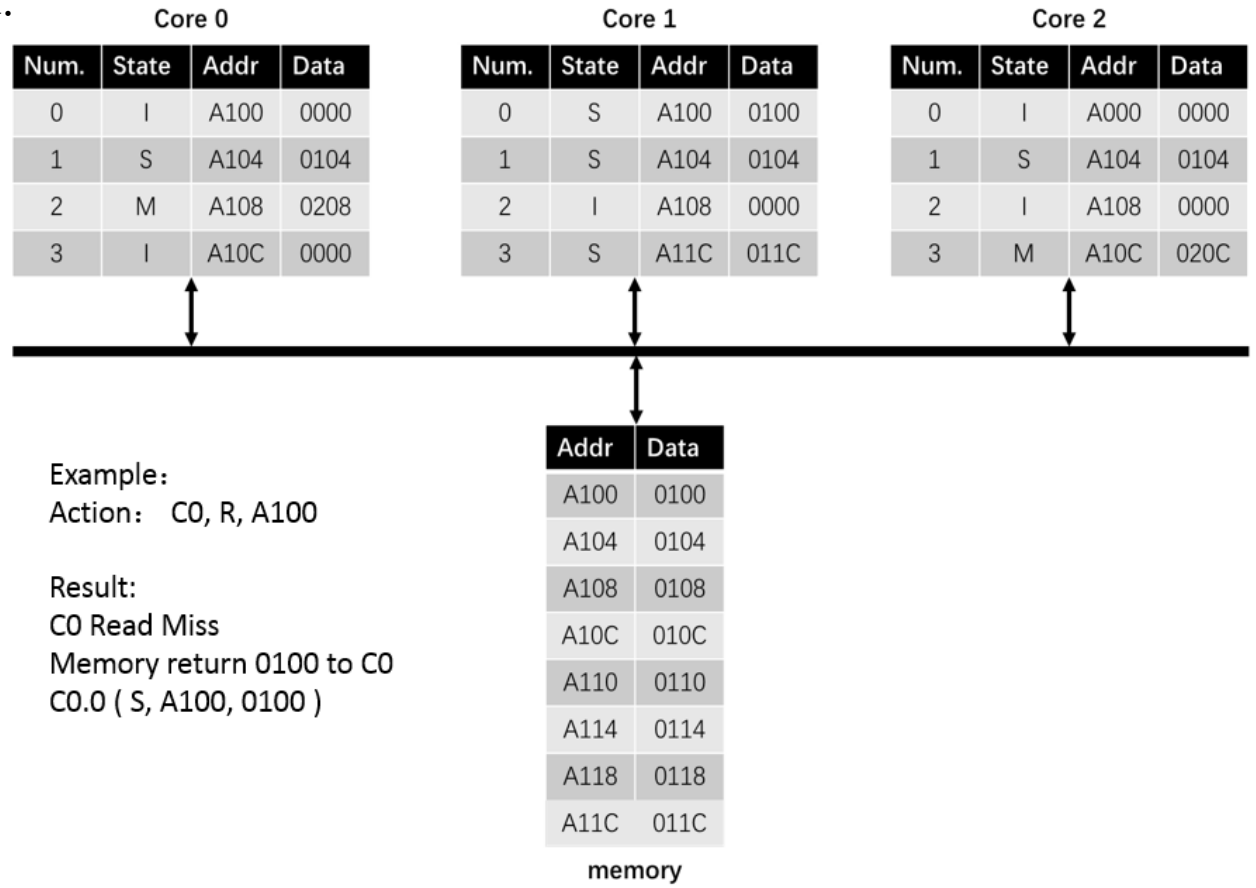
**Action specification format:**

C#, R, <address Axxx> represents
Core # reads address Axxx.

C#, W, <address Axxx>, <value V> represents
Core # writes value V into address Axxx.

Cx.y represents cache line y in core x.

Memory A100, 0000 → 0100 represents
the value in memory location A100 is updated from 0000 to 0100.



# MSI Protocol Extensions (write back)

- MESI protocol: It is named after the initial letter of the four states used in the protocol. Each item in this protocol is in one of the following four states:
  - Invalid: The data contained in the cache item is invalid.
  - Shared: This row of data exists in multiple cache items, and the data in the memory is the latest.
  - Exclusive: No other cache items include this row of data, and the data in memory is the latest.
  - Modified: The data of the item is valid, but the data in the memory is invalid, and there is no copy of the data in other cache items.



# Relaxed Memory Consistency Models

- Rules:
  - X → Y
    - Operation X must complete before operation Y is done
    - Sequential consistency requires:
      - R → W, R → R, W → R, W → W
  - Relax W → R
    - "Total store ordering" or processor consistency
  - Relax W → W and W → R
    - "Partial store order"
  - Relax R → W, R → R, W → W and W → R
    - "Weak ordering" and "release consistency"

**Wish all of your efforts pay back in the final exam!**

# The end is the beginning !