

# 体系结构部分

---

Week 2 - Lec01

性能量化分析

Great Architecture Ideas

Chapter 2

Instruction-Level Parallelism (ILP) | 指令级并行

Week 2- Lec02

Week3\_Lec01

Scoreboard

Tomasulo's Approach

Week\_3 Lec\_02

ROB

Week4\_Lec01

Week4 - Cache

Cache Locality

Cache Miss

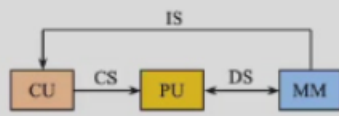
Week5

CPU 漏洞实战分析

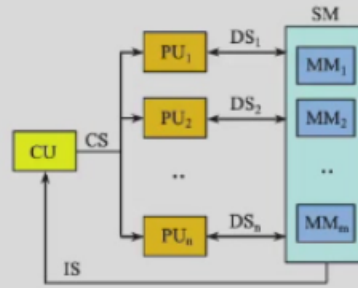
## Week 2 – Lec01

1. 操作系统的分类

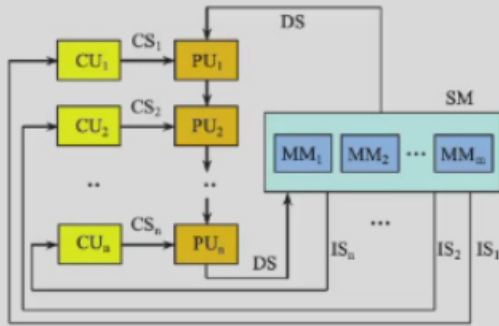
# Classed By Flynn



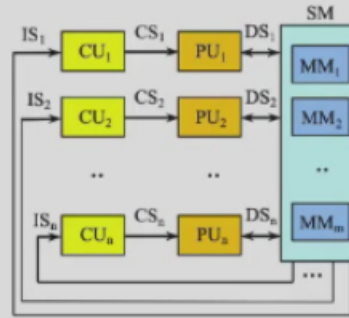
(a) SISD 计算机



(b) SIMD 计算机



(c) MISD 计算机



(d) MIMD 计算机

**IS : Instruction stream**  
**DS : Data stream**  
**CS : Control stream**  
**CU : Control unit**  
**PU : Process unit**  
**MM&SM : Memory**



## 2. Measuring Execution Time

- **Elapsed time**

- Total response time, including all aspects
  - Processing, I/O, OS overhead, idle time
- Determines system performance

- **CPU time**

- Time spent processing a given job
  - Discounts I/O time, other jobs' shares
- Comprises user CPU time and system CPU time
  - User CPU time : CPU time spent in the program itself
  - System CPU time: CPU time spent in the OS, performing tasks on behalf of the program.
- Different programs are affected differently by CPU and system performance

# Measuring Data Size

- bit - Binary digit
- nibble - four bits
- byte - eight bits
- word - four bytes (32 bits) on many embedded/mobile processors and eight bytes (64 bits) on many desktops and servers.
- kibibyte (KiB) [kilobyte (KB)] -  $2^{10}$  (1,024) bytes
- mebibyte (MiB) [megabyte (MB)] -  $2^{20}$  (1,048,576) bytes
- gibibyte (GiB) [gigabyte (GB)] -  $2^{30}$  (1,073,741,824) bytes
- tebibyte (TiB) [terabyte (TB)] -  $2^{40}$  (1,099,511,627,776) bytes
- 3. • pebibyte (PiB) [petabyte (PB)] -  $2^{50}$  (1,125,899,906,842,624) bytes

## 4. 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,

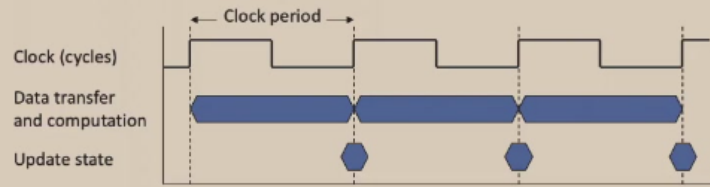
$$CPU\ Execution\ Time = \frac{CPU\ Clock\ Cycles}{Clock\ Rate}$$

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.

## 性能量化分析

## CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns =  $250 \times 10^{-12}$ s
- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz =  $4.0 \times 10^9$ Hz



## CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes  $1.2 \times$  clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$



## 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

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



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

$$CPU \text{ Time} = Instruction \text{ Count} \times CPI \times Clock \text{ Period}$$

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



## Components of CPU performance

The basic components of performance and how each is measured.

Component	Units of Measure
CPU Execution Time for a Program	Seconds for the Program
Instruction Count	Instructions Executed for the Program
Clock Cycles per Instruction	Average Number of Clock Cycles per Instruction
Clock Cycle Time (Clock Period)	Seconds per Clock Cycle

Instruction Count, CPI, and Clock Period combine to form the three important components for determining CPU execution time. *Just analyzing one is not enough!* Performance between two machines can be determined by examining non-identical components.



## CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^n (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^n \left( \text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

Chapter 1 — Fundamentals of computer design — 57



## Performance Summary

### The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI,  $T_c$

Chapter 1 — Fundamentals of computer design — 59



## 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!



Conclusion: Make the common case fast!

## Amdahl's Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

Can't be done!



## Amdahl's Law

- The system performance acceleration rate is limited by the percentage of the execution time of the component to the total execution time in the system.
- Amdahl's law defines the *speedup* that can be gained by using a particular feature.

$$\begin{aligned} \text{Speedup} &= \frac{\text{Performance for entire task}_{\text{Using Enhancement}}}{\text{Performance for entire task}_{\text{Without Enhancement}}} \\ &= \frac{\text{Total Execution Time}_{\text{Without Enhancement}}}{\text{Total Execution Time}_{\text{Using Enhancement}}} \end{aligned}$$

Chapter 1 — Fundamentals of computer design — 63



Amdahl 中存在 2 个 Sp, 局部 + 整体

## Amdahl's Law

- Based on the basic idea that:

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{can not be enhance}} + \text{Execution time}_{\text{enhanced}}$$

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

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

Chapter 1 — Fundamentals of computer design — 64

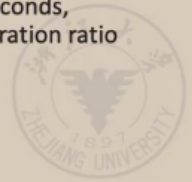


Fraction 指得是比例  $F_{\text{enhanced}}$  表示改进的比例



## Amdahl's Law

- **Improved ratio:** In the system before the improvement, the ratio of the execution time of the improvement part to the total execution time.
  - It is always less than or equal to 1.
  - For example: a program that needs to run for 60 seconds has 20 seconds of calculation that can be accelerated, Then the ratio is 20/60.
- **Component speedup ratio:** The multiple that can be improved after some improvements. It is the ratio of the execution time before the improvement to the execution time after the improvement.
  - Under normal circumstances, the component acceleration ratio is greater than 1.
  - For example: if the system is improved, the execution time of the improved part is 2 seconds, Before the improvement, its execution time was 5 seconds, and the component acceleration ratio was 5/2.



例子:

## Amdahl's Law

- Example 1.1 Increasing the processing speed of a certain function in the computer system to **20 times** the original, but the processing time of this function only accounts for **40%** of the running time of the entire system. After adopting this method to improve performance, how much can the performance of the entire system improve?

Answer:

- Fraction<sub>enhanced</sub> = 40%
- Speedup<sub>enhanced</sub> = 20

$$\text{Speedup} = \frac{1}{0.6 + \frac{0.4}{20}} = 1.613$$



1.



## Amdahl's Law

- Example 1.2 After a computer system adopts floating-point arithmetic components, the floating-point arithmetic speed is increased by *20 times*, and the overall performance of a certain program of the system is increased by *5 times*. Try to calculate the proportion of the floating-point operations in this program.

Answer:

- Speedup<sub>overall</sub> = 5
- Speedup<sub>enhanced</sub> = 20

$$\frac{1}{(1 - \text{Fraction}) + \frac{\text{Fraction}}{20}} = 5$$

Fraction = 84.2%



2.

## 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.



## Chapter 2

### Instruction-Level Parallelism (ILP) | 指令级并行

## Week 2– Lec02

- Branch History Table (BHT)
  - 1–Bit Predictor
  - 2–Bit Predictor
- Branch–Target Buffers (BTB)

## Week3\_Lec01

### Dynamic Scheduling

A major **limitation** of simple pipelining techniques is that:

- they use in-order instruction issue and execution
- For example, consider this code:

```

FDIV.D    F4, F0, F2
FSUB.D    F10, F4, F6
FADD.D    F12, F6, F14
  
```

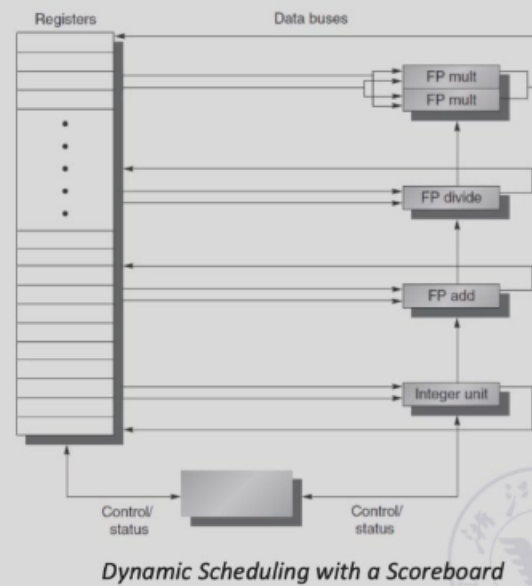
The **FADD.D** instruction cannot execute because the **dependence of FSUB.D on FDIV.D** causes the pipeline to **stall**; yet, **FADD.D** is not data dependent on anything in the pipeline.

Instructions are issued in program order, and if an instruction is stalled in the pipeline no later instructions can proceed.

### Dynamic Scheduling

Idea: Dynamic Scheduling

Method: out-of-order execution



score board 是记录当前程序运行的状态 / 信息 (应该记录哪些信息)

## • Functional unit status

state of the functional unit

- **Busy**: indicates whether the unit is busy or not
- **Op**: operation to perform in the unit
- $F_i$ : destination register
- $F_j, F_k$ : source registers
- $Q_j, Q_k$ : functional units producing  $F_j, F_k$
- $R_j, R_k$ : flags indicating when  $F_j, F_k$  are ready and not yet read. Set to NO after operands are read.

Time	Name	Busy	Op	$F_i$	$F_j$	$F_k$	$Q_j$	$Q_k$	$R_j$	$R_k$
	Integer	Yes	Load	F2	F3				No	
	Mult1	Yes	Mult	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

## • Register result status:

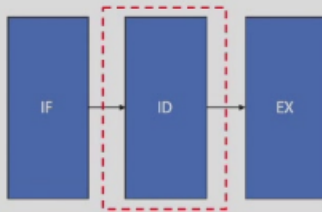
indicates which functional unit (FU) will write each register

	F0	F2	F4	F6	F8	F10	F12	...	F30
Cycle 0	FU		Mult1						

- 功能单元状态 重点看绿色字体和绿色字体后面的解释，这些就是记分牌记下的信息。记分牌是面向功能部件的，在记分牌中每一个功能部件都有一组信息，信息包括部件是否正在忙、部件执行的指令类型、部件现在需要的源寄存器、部件现在的目的寄存器、源寄存器是否准备好（ $R_j$ 、 $R_k$  表示）和如果源寄存器没准备好部件该向哪里要数据（ $Q_j$ 、 $Q_k$  表示，PPT 中有一个表格，表格 Mult1 这一行  $Q_j$  是 Integer，这就表示乘法单元 1 的 F2 源寄存器的数值将由整数部件算出）
- 寄存器结果状态 里面主要记录对于某一个寄存器，是否有部件正准备写入数据。例如上图中 F4 对应 Mult1，这就表明乘法单元1的计算结果将要写入 F4。

# Dynamic Scheduling

- A Simple Implementation of RISC-V



Check **structural** hazards

Check **data** hazards

- When an instruction could execute without hazards, it was issued from ID knowing that all data hazards had been resolved.

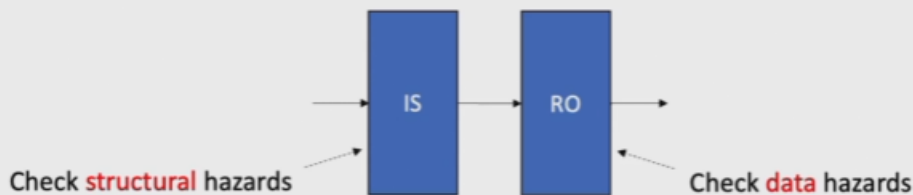
将 ID 拆分

1. IS → Check structural hazards
2. RO → Check data hazards

# Dynamic Scheduling

- To allow out-of-order execution, we essentially split the ID pipe stage of our simple five-stage pipeline into two stages :

- Issue(IS): Decode instructions, check for structural hazards. (in-order issue)
- Read Operands(RO): Wait until no data hazards, then read operands. (out of order execution)



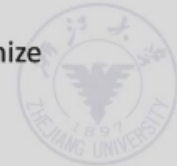
## Dynamic Scheduling

- Out-of-order execution introduces the possibility of **WAR** and **WAW** hazards, which do not exist in the **five-stage** integer pipeline and its logical extension to an in-order floating-point pipeline.

- Consider the following RISC-V floating-point code sequence:

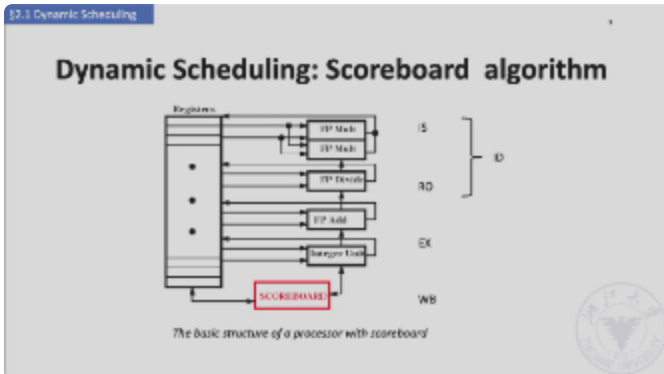
<b>WAW</b>	{	FDIV.D	F10, F0, F2	}	<b>WAR</b>
	FSUB.D	F10, F4, F6			
	FADD.D	F6, F8, F14			

- **Scoreboard algorithm** is an approach to schedule the instructions.
- **Robert Tomasulo** introduces register renaming in hardware to minimize WAW and WAR hazards, named **Tomasulo's Approach**.



- Scoreboard algorithm
- Robert Tomasulo (比scoreboard更自动化)

## Scoreboard



## Dynamic Scheduling: Scoreboard algorithm

- Show what the information tables look like for the following code sequence when only the first load has completed and written its result:

```

FLD      F6, 34 (R2)
FLD      F2, 45 (R3)
FMUL.D   F0, F2, F4
FSUB.D   F8, F2, F6
FDIV.D   F10, F0, F6
FADD.D   F6, F8, F2
  
```



指令状态表:

## 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				



功能部件表:



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.

寄存器状态表:

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

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	✓	✓	✓	

Show what the status tables look like when the FMUL.D is ready to write its result.

(MUL指令较sub指令时间长)

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

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

Show what the status tables look like when the FMUL.D is ready to write its result.

## Tomasulo's Approach

### §2.1 Dynamic Scheduling

## Dynamic Scheduling: The Idea

- Consider the following RISC-V floating-point code sequence:

Anti-dependence WAR hazards (F8)	}	FDIV.D F0, F2, F4	}	Output-dependence WAR hazards (F6)
		FADD.D F6, F0, F8		
		FSD F6, 0(R1)		
		FSUB.D F8, F10, F14		
		FMUL.D F6, F10, F8		



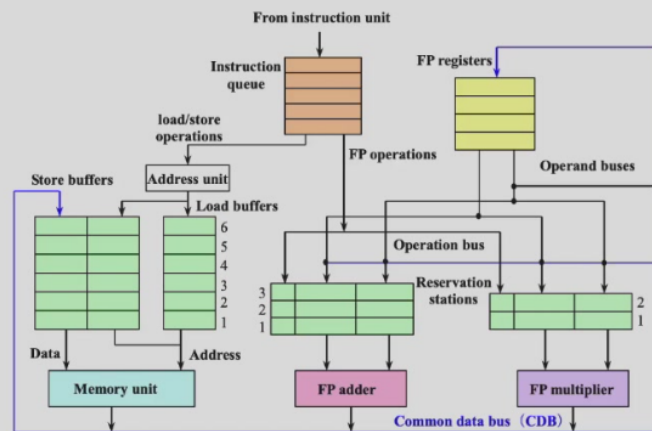
## Tomasulo's Approach

- These name dependences can all be eliminated by register renaming.
  - Assume the existence of two temporary registers, S and T.
  - The sequence can be rewritten without any dependences as :

FDIV.D	F0, F2, F4	
FADD.D	S, F0, F8	} F6 change as S
FSD	S, 0(R1)	
FSUB.D	T, F10, F14	} F8 change as T
FMUL.D	F6, F10, T	



## Tomasulo's Approach



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



- 保留栈有位置，指令就可以进入
- 指令进buffer是按顺序的，但谁的源操作数先ready谁先执行

## Tomasulo's Approach: Main Idea

- It tracks when operands for instructions are available to minimize RAW hazards;
- It introduces register renaming in hardware to minimize WAW and WAR hazards.



(WAW和WAR都可以通过重命名几乎消除)

(Tomasulo 算法可以分为三步)

## Tomasulo's Approach

- Let's look at the **three steps** an instruction goes through :
  - Issue : Get the next instruction from the head of the instruction queue (FIFO)
  - If there is a matching **reservation station** that is **empty**, **issue** the instruction to the station with the operand values, if they are currently in the registers.
  - If there is **not an empty reservation station**, then there is a **structural hazard** and the instruction stalls until a station or buffer is freed.
  - If the operands are not in the registers, **keep track of the functional units** that will produce the operands.
- This step renames registers, eliminating WAR and WAW hazards not in the registers.



- 检查保留栈是否有位置 (有位置就一拍一条指令进入)

## Tomasulo's Approach

### Execute

- When all the operands are available, the operation can be executed at the corresponding functional unit.
- **Load** and **store** require a two-step execution process :
  - It computes the effective address when the base register is available.
  - The effective address is then placed in the **load** or **store** buffer.



## Tomasulo's Approach

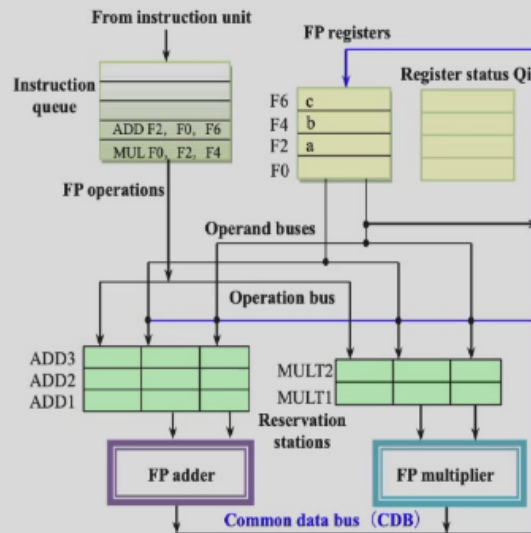
### Write results

- When the result is available, write it on the **CDB** and from there into the registers and into any reservation stations (including store buffers).
- **Stores** are buffered in the store buffer until both the value to be stored and the store address are available, then the result is written as soon as the memory unit is free.



Example

# Tomasulo's Approach



# Tomasulo's Approach

There are three tables for Tomasulo's Approach.

- **Instruction status table:** This table is included only to help you understand the algorithm; it is not actually a part of the hardware.
- **Reservation stations table:** The reservation station keeps the state of each operation that has issued.
- **Register status table (Field  $Q_i$ ):** The number of the reservation station that contains the operation whose result should be stored into this register.



## Tomasulo's Approach

Each reservation station has seven fields:

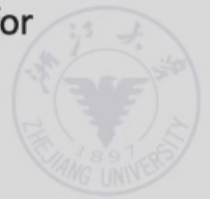
**Op**: The operation to perform on source operands.

**Qj, Qk**: The reservation stations that will produce the corresponding source operand.

**Vj, Vk**: The value of the source operands.

**Busy**: Indicates that this reservation station and its accompanying functional unit are occupied.

**A**: Used to hold information for the memory address calculation for a load or store.



Example

## Dynamic Scheduling: Tomasulo's algorithm

Instruction		Instruction Status		
		Issue	Execute	Write Result
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	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					45+Regs[R3]
Add1	Yes	SUB		Mem[34+Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Reg[F4]	Load2		
Mult2	Yes	DIV		Mem[34+Regs[R2]]	Mult1		

Qi	Register Status							
	F0	F2	F4	F6	F8	F10	...	F30
	Mult1	Load2		Add2	Add1	Mult2		



## Dynamic Scheduling: Tomasulo's algorithm

Instruction	Instruction Status		
	Issue	Execute	Write Result
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	✓	✓	✓

Show what the status tables look like when the FMUL.D is ready to write its result.





## Summary

### 1. Tomasula's Algorithm main contributions

- Dynamic scheduling
- Register renaming---eliminating WAW and WAR hazards
- Load/store disambiguation
- Better than Scoreboard Algorithm



Tom -> issue, Execute, Write Result

## Summary

### 1. Tomasula's Algorithm main contributions

- Dynamic scheduling
- Register renaming---eliminating WAW and WAR hazards
- Load/store disambiguation
- Better than Scoreboard Algorithm



## Summary

### 2. Tomasulo's Algorithm major defects

- Structural complexity.
- Its performance is limited by Common Data Bus.
- A load and a store **can safely be done out of order**, provided they access **different addresses**. If a load and a store access the same address, then either:
  - The load is before the store in program order and interchanging them results in a WAR hazard, or
  - The store is before the load in program order and interchanging them results in a RAW hazard
  - Interchanging two stores to the same address results in a WAW hazard



## Summary

3. The limitations on ILP approaches directly led to the movement to multicore.

## Question

Does **out-of-order execution** mean **out-of-order completion**?

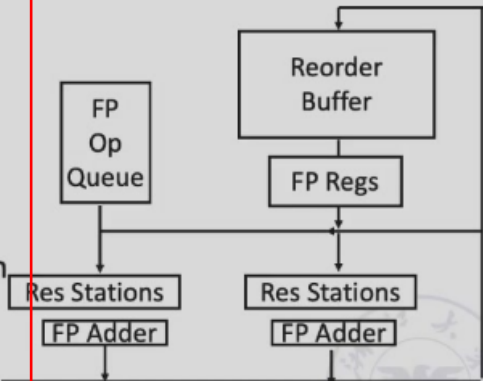


# Hardware-Based Speculation

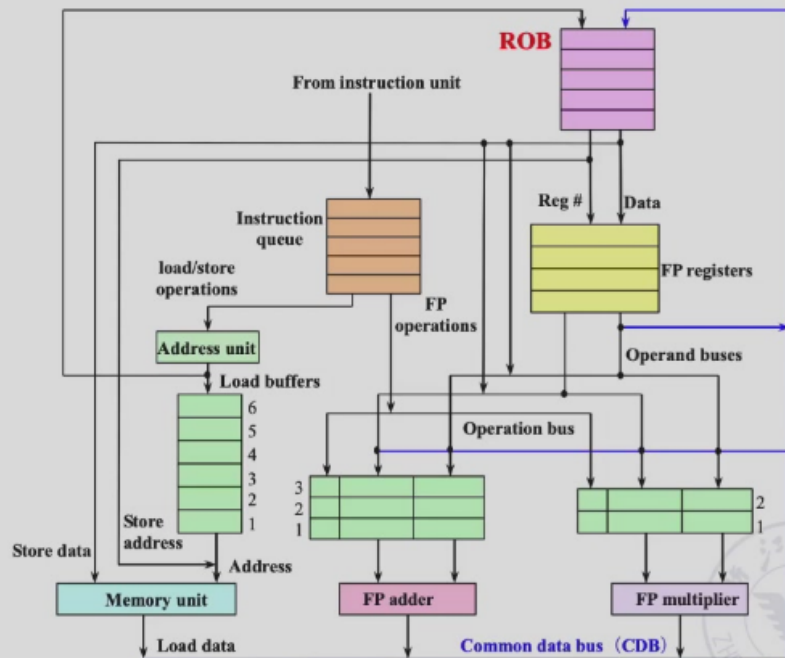
Cache for uncommitted instruction results:

3 fields: instruction type, destination address, value

1. When the program execution phase is completed, replace the value in RS with the number of ROB
2. Increase instruction submission stage
3. ROB provides the number of operations in the completion phase and the commit phase
4. Once the operand is submitted, the result is written to the register
5. In this way, when the prediction fails, it is easy to restore the inferred execution instruction, or when an exception occurs, it is easy to restore the state



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



ROB

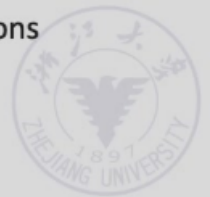
## Hardware-Based Speculation

1. Issue—get instruction from FP Op Queue
2. Execution—operate on operands (EX)
3. Write result—finish execution (WB)
4. Commit—update register with reorder result



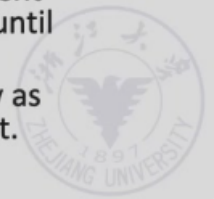
## Hardware-Based Speculation

- Hardware-based speculation combines three key ideas
  - dynamic branch prediction to choose which instructions to execute
  - speculation to allow the execution of instructions before the control dependences are resolved (with the ability to undo the effects of an incorrectly speculated sequence)
  - dynamic scheduling to deal with the scheduling of different combinations of basic blocks



## Hardware-Based Speculation

- WB
  - The ROB holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits
  - The ROB is a source of operands for instructions, just as the reservation stations provide operands in Tomasulo's algorithm.
- instruction commit
  - The key idea behind implementing speculation is to allow instructions to execute out of order but to force them to commit in order and to prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits.
  - The reorder buffer (ROB) provides additional registers in the same way as the reservation stations in Tomasulo's algorithm extend the register set.



ROB 多了 Commit 环节

(拿数据也是从 ROB 拿数据，当轮到 Commit 时将数据输入 ROB 等待 Commit)

(写回后不占用操作单元，即前面和 Tomasulo 是一样的)

## 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



# Hardware-Based Speculation

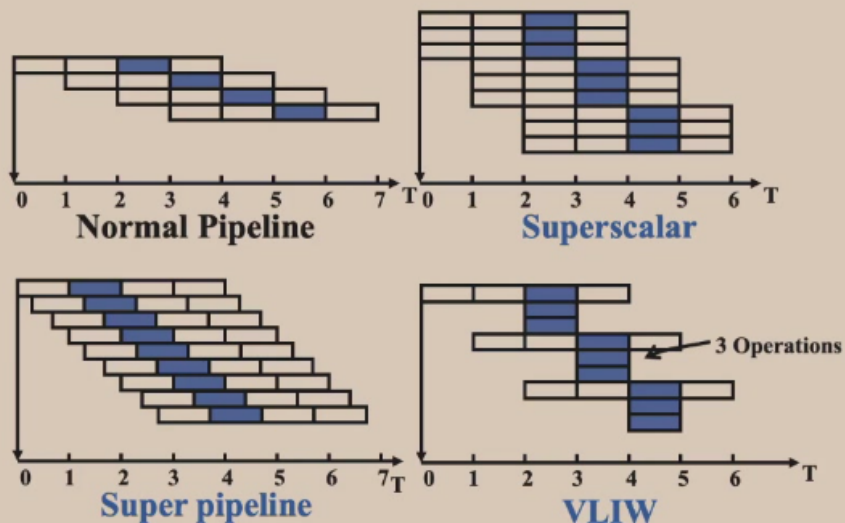
- Instructions are finished in order according to ROB
- It can be precise exception.
- It is easily extended to integer register and integer function unit.
- But the hardware is too complex.



## Week4\_Lec01

北1- 302

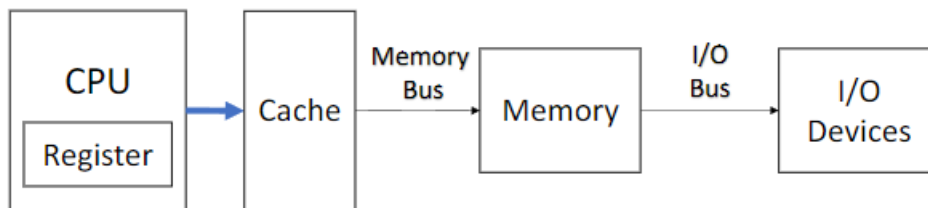
# Superscalar & VLIM



# Week4 – Cache

## Memory

- Register
  - Cache
  - Memory
  - Storage
- Mechanical memory
    - Acoustic wave/torque wave delay line memory
    - Magnetic Drum Memory
    - Magnetic core memory
  - Electronic memory
    - SRAM
    - DRAM
      - SDRAM
    - Flash
    - ROM
      - PROM
      - EPROM
  - Optical memory
- Cache: a safe place for hiding or storing things (1976)

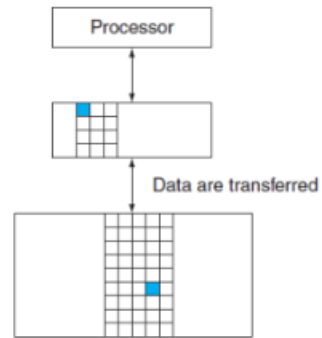


- The highest or first level of the memory hierarchy encountered once the addr leaves the processor
- Employ buffering to reuse commonly occurring items



# Cache Hit/Miss

- When the processor **can/cannot** find a requested data item in the cache



hit rate      miss rate

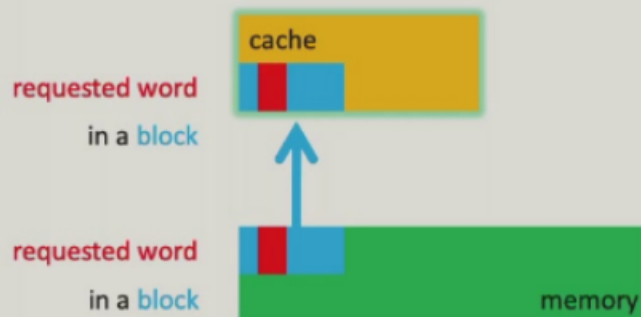
hit time      miss penalty



- 需要寻找的信息在 cache 上叫 hit

## Block/Line Run

- A fixed-size collection of data containing the requested word, retrieved from the main memory and placed into the cache



## Cache Locality

- **Temporal locality**  
need the requested word again soon
- **Spatial locality**  
likely need other data in the block soon

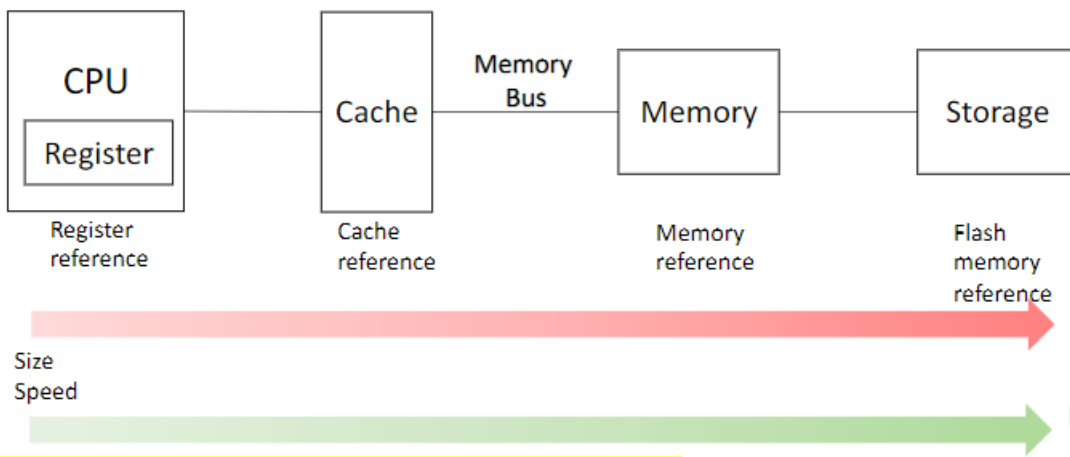


# Cache Miss

- Time required for cache miss depends on:
  - Latency:** the time to retrieve the first word of the block
  - Bandwidth:** the time to retrieve the rest of this block

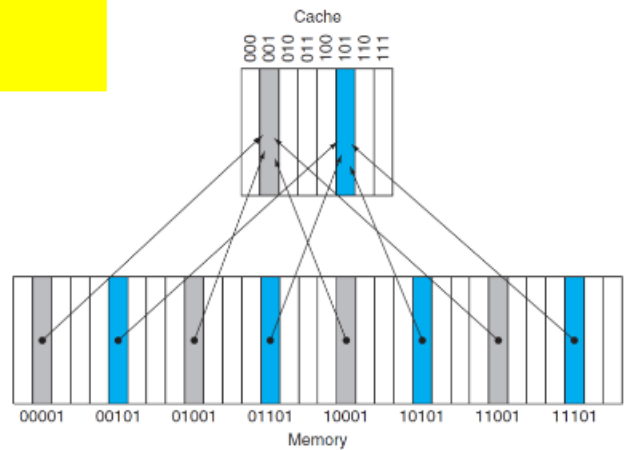
经典模型 ↓

## Processor-Memory Performance Gap

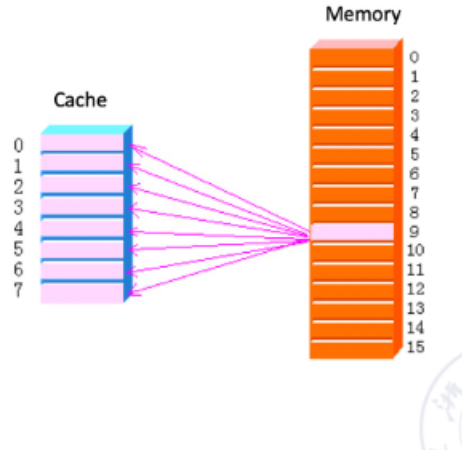
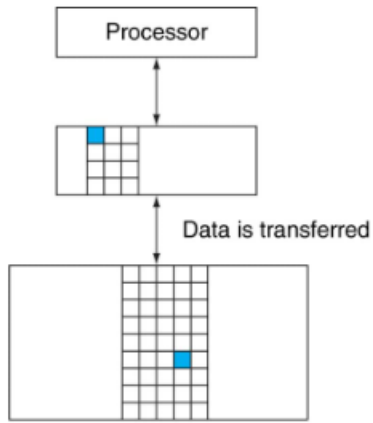


### Q1: Block Placement Direct mapped

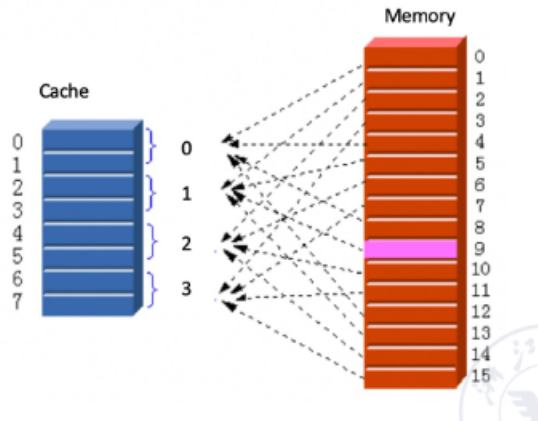
(Block address) modulo (Number of blocks in the cache)



# Fully-associative

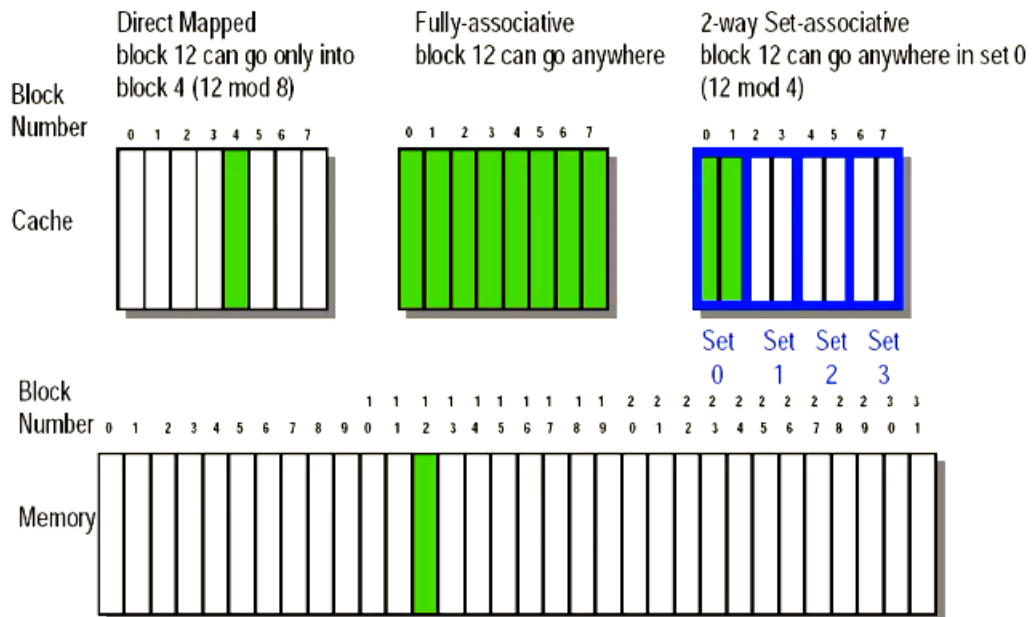


# 2-way Set-associative



组相联取决于 cache 有多少块

# 8-32 Block Placement



## N-way Set-associative

- The higher the degree of association, the higher the utilization of cache space, the lower the probability of block collision and the lower the failure rate.

	$n$	$G$
Full-associative	$M$	$1$
Direct mapped	$1$	$M$
Set-associative	$1 < n < M$	$1 < G < M$

- Most Cache:  $n \leq 4$
- Question: Is the greater the number  $n$ , the better?

## Q2: Block Identification

- Every block has an **address tag** that stores the main memory address of the data stored in the block.

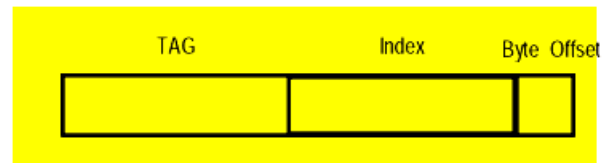
- When checking the cache, the processor will **compare** the requested **memory address to the cache tag** -- if the two are equal, then there is a cache hit and the data is present in the cache

- Often, each cache block also has a **valid bit** that tells if the contents of the cache block are valid

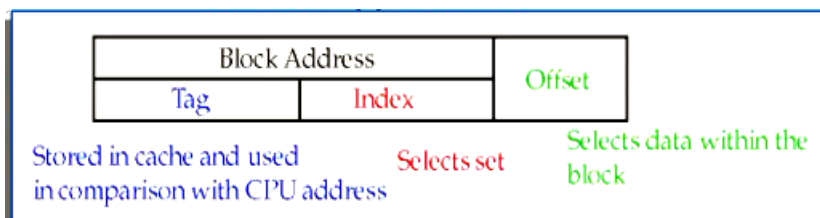
## The Format of the Physical Address

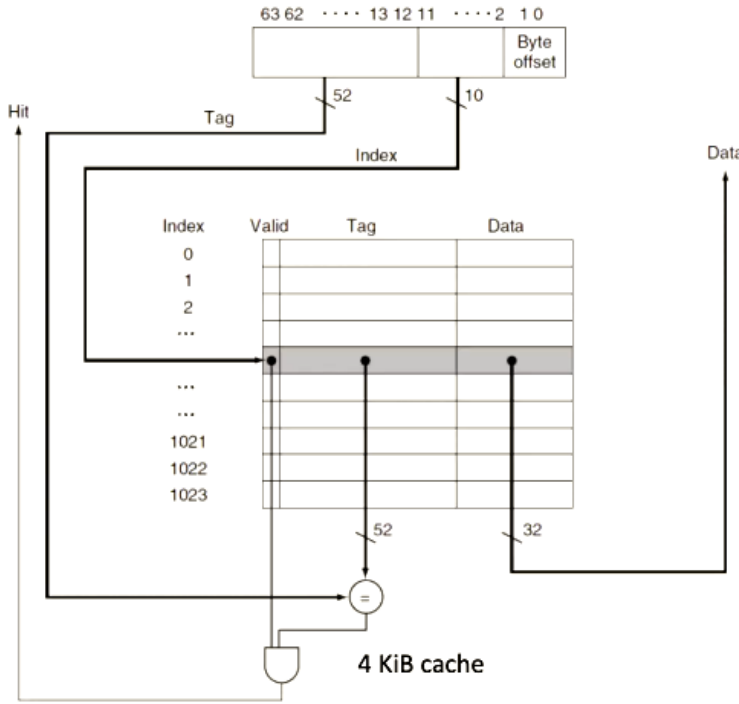
- The **Index** field selects
  - The **set**, in case of a **set-associative cache**
  - The **block**, in case of a **direct-mapped cache**
  - Has as many bits as  **$\log_2(\#sets)$**  for **set-associative caches**, or  **$\log_2(\#blocks)$**  for **direct-mapped caches**

- The **Byte Offset** field selects
  - The byte within the block
  - Has as many bits as  **$\log_2(\text{size of block})$**



- The **Tag** is used to find the matching block within a set or in the cache
  - Has as many bits as  **$Address\_size - Index\_size - Byte\_Offset\_Size$**





- 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

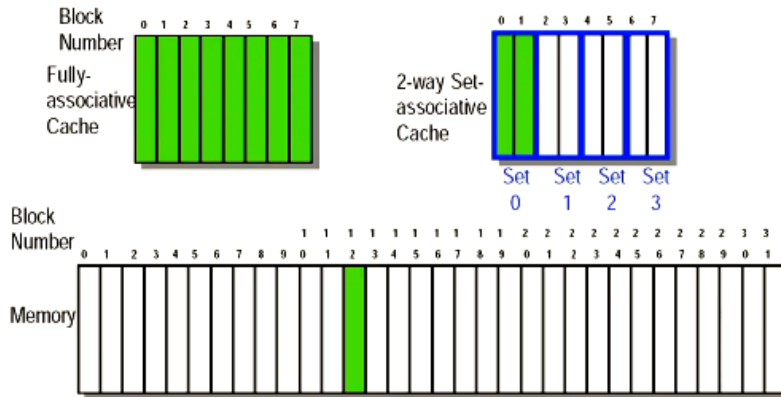
$$\text{tag} = 64 - (n + m + 2)$$

$$\begin{aligned} \text{cache} &= 2^n \times (\text{block size} + \text{tag size} + \text{Valid size}). \\ &= 2^n \times (2^m \times 32 + (64 - n - m - 2) + 1) \\ &= 2^n \times (2^m \times 32 + 63 - n - m). \end{aligned}$$

actual size/complete cache size

## Q3: Block Replacement

- In a direct-mapped cache, there is only one block that can be replaced
- In set-associative and fully-associative caches, there are  $N$  blocks (where  $N$  is the degree of associativity)



直接映射 是不需要替换的

# Strategy of Block Replacement

- Several different replacement policies can be used

- **Random replacement** - *randomly pick any block*

- Easy to implement in hardware, just requires a random number generator
- Spreads allocation uniformly across cache
- May evict a block that is about to be accessed

- **Least-Recently Used (LRU)** - *pick the block in the set which was least recently accessed*

- Assumed more recently accessed blocks more likely to be referenced again
- This requires extra bits in the cache to keep track of accesses.

- **First In, First Out(FIFO)**-*Choose a block from the set which was first came into the cache*

Pre: 10mins -> 程序复现 or 搜集 or 推导

分析 CPU 性能

## Week5

### § 3.3 Four Questions for Cache Designers

## Strategy of Block Replacement

- Several different replacement policies can be used

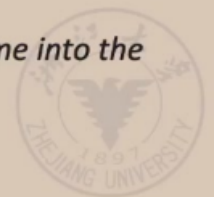
- **Random replacement** - *randomly pick any block*

- Easy to implement in hardware, just requires a random number generator
- Spreads allocation uniformly across cache
- May evict a block that is about to be accessed

- **Least-Recently Used (LRU)** - *pick the block in the set which was least recently accessed*

- Assumed more recently accessed blocks more likely to be referenced again
- This requires extra bits in the cache to keep track of accesses.

- **First In, First Out(FIFO)**-*Choose a block from the set which was first came into the cache*



**cache hit** If the cache reports a hit, the computer continues using the data as if nothing happened.

**cache miss** For a cache miss, we can stall the entire processor, essentially freezing the contents of the temporary and programmer-visible registers, while we wait for memory.



We can now define the steps to be taken on an instruction cache miss:

1. Send the original PC value to the memory. (PC-4)
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.
4. Restart the instruction execution at the first step, which will refetch the instruction, this time finding it in the cache.



Hit rate is related to cache block size

## Stack replacement algorithm

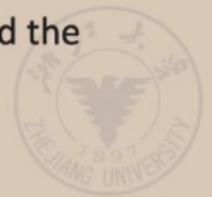
- $B_t(n)$  represents the set of access sequences contained in a cache block of size  $n$  at time  $t$ .
- $B_t(n)$  is the subset of  $B_t(n + 1)$ .



保证命中率随 cache 变大只增不减

## LRU

- How can I implement the LRU replacement algorithm with only ordinary gates and triggers?
- Comparison Pair Method
- Basic idea: Let each cache block be combined in pairs, use a **comparison pair flip-flop** to record the order in which the two cache blocks have been accessed in the comparison pair, and then use a gate circuit to combine the state of each comparison pair flip-flop, you can find the block to be replaced according to the LRU algorithm.





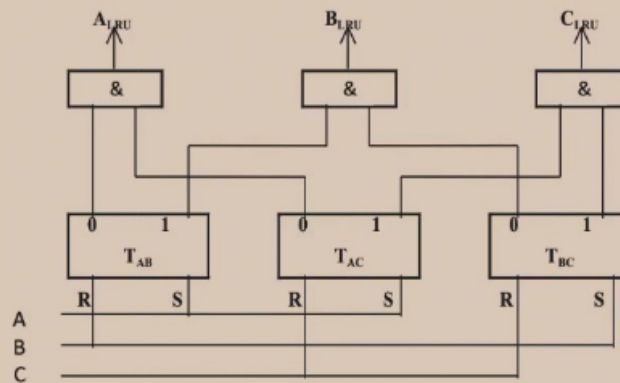
## Example

- There are three cache blocks (A, B, and C), which can be combined into 6 pairs (AB, BA, AC, CA, BC, and CB). Among them, AB and BA, AC and CA, BC and CB are repeated, so only take AB, AC, BC.
- The access sequence of each pair is represented by “comparison pair flip-flops”  $T_{AB}$ ,  $T_{AC}$ , and  $T_{BC}$  respectively.  $T_{AB}$  is "1", which means that A has been accessed more recently than B;  $T_{AB}$  is "0", which means that B has been accessed more recently than A.  $T_{AC}$  and  $T_{BC}$  are similarly defined.



- If the most recently accessed block is A and C is the block that has not been accessed for the longest time, the three flip-flops' states must be respectively:  $T_{AB}=1$ ,  $T_{AC}=1$ , and  $T_{BC}=1$ .
- If the most recently accessed block is B and C is the block that has not been accessed for the longest time, the three flip-flops' states must be respectively:  $T_{AB}=0$ ,  $T_{AC}=1$ , and  $T_{BC}=1$ .
- Therefore, the block C that has not been accessed for the longest will be replaced. In that, the Boolean algebra expression must be:

$$C_{LRU} = T_{AB} \cdot T_{AC} \cdot T_{BC} + \overline{T_{AB}} \cdot T_{AC} \cdot T_{BC} = T_{AC} \cdot T_{BC}$$



- Change the state of the flip-flop after each access.
  - After accessing block A:  $T_{AB}=1$ ,  $T_{AC}=1$
  - After accessing block B:  $T_{AB}=0$ ,  $T_{BC}=1$
  - After accessing block C:  $T_{AC}=0$ ,  $T_{BC}=0$



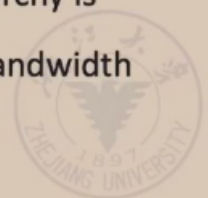
## Hardware usage analysis (if $p$ is the number of cache blocks)

- Since each block may be replaced, its signal needs to be generated with an AND gate, so the number of AND gates will be equal to  $p$ .
- Each AND gate receives inputs from its related flip-flops, for example,  $A_{LRU}$  AND gates must have inputs from  $T_{AB}$  and  $T_{AC}$ ,  $B_{LRU}$  must have inputs from  $T_{AB}$  and  $T_{BC}$ , and the number of comparison pair flip-flops is the block number minus 1, so the input number of the AND gate is  $p - 1$ .
- If  $p$  is the block number, for pairwise combination, the number of comparison pair flip-flops should be  $C_p^2$ , which is  $p \cdot (p-1) / 2$ .



## 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.

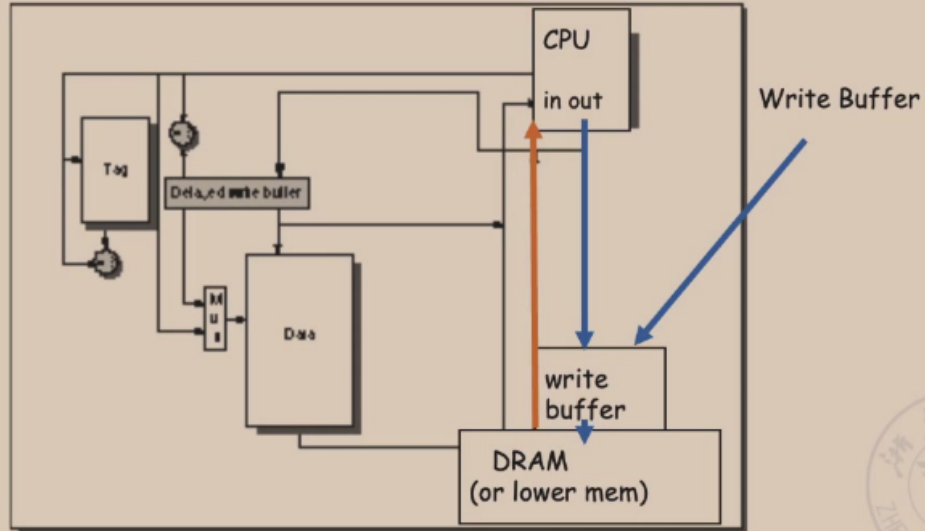


## Write stall

- **Write stall** ---When the CPU must wait for writes to complete during write through
- Write buffers
  - A small cache that can hold a few values waiting to go to main memory.
  - To avoid stalling on writes, many CPUs use a write buffer.
  - This buffer helps when writes are clustered.
  - It does not entirely eliminate stalls since it is possible for the buffer to fill if the burst is larger than the buffer.



## Write buffers



## Write misses

- Write misses
  - If a miss occurs on a write (the block is not present), there are two options.
  - Write allocate
    - The block is loaded into the cache on a miss before anything else occurs.
  - Write around (no write allocate)
    - The block is only written to main memory
    - It is not stored in the cache.
- In general, write-back caches use write-allocate , and write-through caches use write-around .



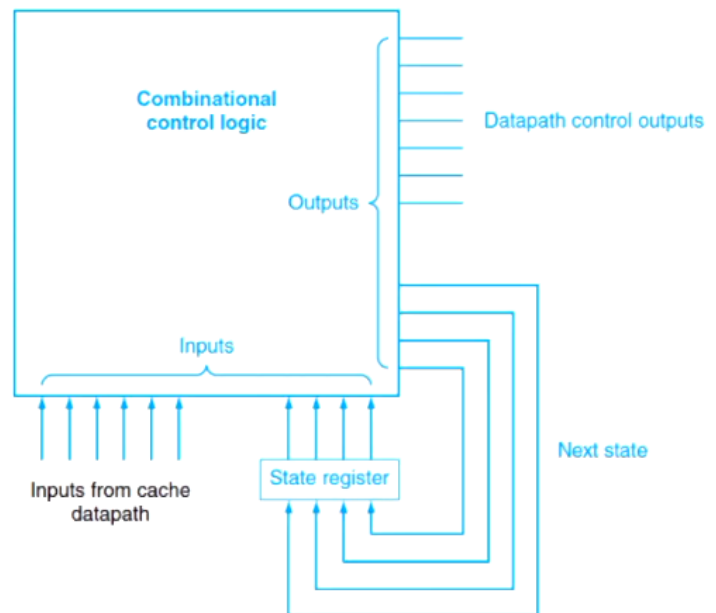
## Using a Finite-State Machine to Control a Simple Cache

### finite-state machine

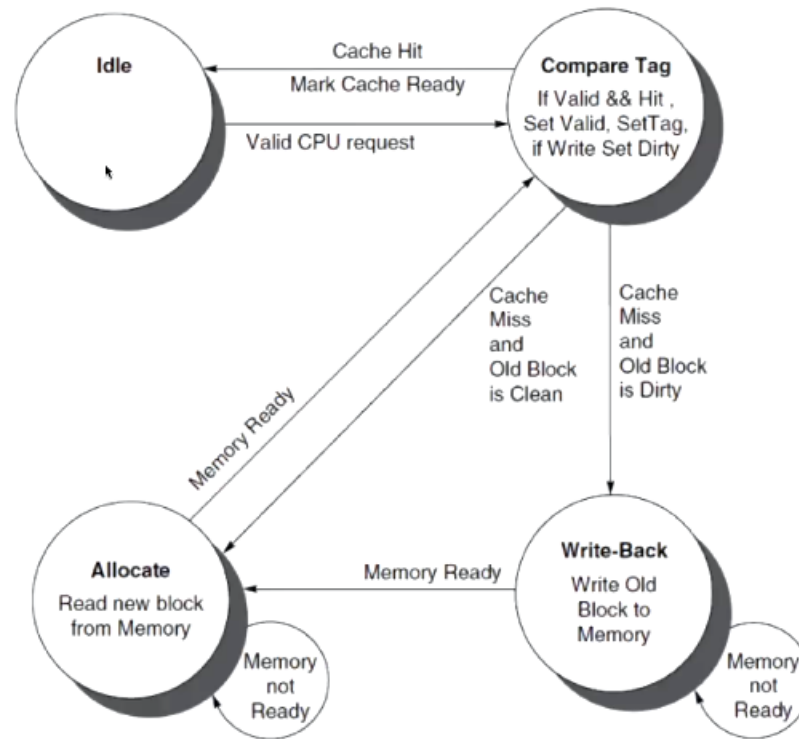
A sequential logic function consisting of a set of inputs and outputs, next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

### next-state function

A combinational function that, given the inputs and the current state, determines the next state of a finite-state machine.



## Using a Finite-State Machine to Control a Simple Cache



## Memory System Performance

CPU Execution time

- CPU Execution time = (CPU clock cycles + Memory stall cycles) × Clock cycle time

Memory stall cycles =  $IC \times \text{MemAccess refs per instructions} \times \text{Miss rate} \times \text{Miss penalty}$

$$CPUtime = IC \times \left( CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty \right) \times CycleTime$$

$$CPUtime = IC \times \left( CPI_{Execution} + \frac{MemMisses}{Inst} \times MissPenalty \right) \times CycleTime$$

- CPI Execution includes ALU and Memory instructions

# Average Memory Access Time

- Average Memory Access Time

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

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

## Example1: Impact on Performance

- Suppose a processor executes at
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- Suppose that 10% of memory operations get 50 cycle miss penalty
- Suppose that 1% of instructions get same miss penalty
- What is the CPUtime and the AMAT ?

• Answer: CPI = ideal CPI + average stalls per instruction = 1.1(cycles/ins) + [ 0.30 (DataMops/ins) x 0.10 (miss/DataMop) x 50 (cycle/miss)] + [ 1 (InstMop/ins) x 0.01 (miss/InstMop) x 50 (cycle/miss)] = (1.1 + 1.5 + .5) cycle/ins = 3.1

• AMAT=(1/1.3)x[1+0.01x50]+(0.3/1.3)x[1+0.1x50]=2.54

# Example2: Impact on Performance

**Assume :** Ideal CPI=1 (no misses)

- L/S's structure . 50% of instructions are data accesses
- Miss penalty is 25 clock cycles
- Miss rate is 2%
- How faster would the computer be if all instructions were cache hits?

**Answer:** first compute the performance for always hits:

$$\begin{aligned} \text{CPU}_{\text{execution time}} &= (\text{CPU clock cycles} + \text{memory stall cycles}) \times \text{clock cycle} \\ &= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle} \\ &= \text{IC} \times 1.0 \times \text{clock cycle} \end{aligned}$$

- Now for the computer with the real cache, first compute memory stall cycles:

$$\begin{aligned} \text{Memory stall cycles} &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Missrate} \times \text{Miss penalty} \\ &= \text{IC} \times (1 + 0.5) \times 0.02 \times 25 = \text{IC} \times 0.75 \end{aligned}$$

The total performance is thus:

$$\begin{aligned} \text{CPU execution time cache} &= (\text{IC} \times 1.0 + \text{IC} \times 0.75) \times \text{Clock cycle} \\ &= 1.75 \times \text{IC} \times \text{Clock cycle} \end{aligned}$$

The performance ratio is the inverse of the execution times

$$\begin{aligned} \frac{\text{CPU execution time}_{\text{cache}}}{\text{CPU execution time}} &= \frac{1.75 \times \text{IC} \times \text{Clock cycle}}{1.0 \times \text{IC} \times \text{clock cycle}} \\ &= 1.75 \end{aligned}$$

The computer with no cache misses is 1.75 time faster.



## Example3: Impact on Performance

Assume: CPI=2(perfect cache)      clock cycle time = 1.0 ns

- MPI(memory reference per instruction) = 1.5
- Size of both caches is 64K and size of both block is 64 bytes
- One cache is direct mapped and other is two-way set associative. the former has miss rate of 1.4%, the latter has miss rate 1.0%
- The selection multiplexor forces CPU clock cycle time to be stretched 1.25 times
- Miss penalty is 75ns, and hit time is 1 clock cycle
- What is the impact of two cache organizations on performance of CPU (first, calculate the average memory access time and then CPU performance)?

Answer :

Average memory access time is

- Average memory access time = Hit time + Miss rate × miss penalty

Thus, the time for each organization is

- Average memory access time<sub>1-way</sub> = 1.0 + (0.014 × 75) = 2.05 ns
- Average memory access time<sub>2-way</sub> = 1.0 × 1.25 + (0.01 × 75) = 2.00 ns

## Example3: Impact on Performance

The average memory access time is better for the 2-way set-associative cache.

CPU performance is

$$\begin{aligned}
 CPU\ time &= IC \times \left( CPI_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time} \\
 &= IC \times \left[ \left( CPI_{\text{execution}} \times \text{Clock cycle time} \right) \right. \\
 &\quad \left. + \left( \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \times \text{Clock cycle time} \right) \right]
 \end{aligned}$$

Substituting 75 ns for (miss penalty × Clock cycle time), the performance of each cache organization is

$$CPU\ time_{1\text{-way}} = IC \times (2 \times 1.0 + (1.5 \times 0.014 \times 75)) = 3.58 \times IC$$

$$CPU\ time_{2\text{-way}} = IC \times (2 \times 1.0 \times 1.25 + (1.5 \times 0.010 \times 75)) = 3.63 \times IC$$

Relative performance is

$$\frac{CPU\ time_{2-way}}{CPU\ time_{-way}} = \frac{3.63 \times Instruction\ count}{3.58 \times Instruction\ count} = \frac{3.63}{3.58} = 1.01$$

In contrast to the results of average memory access time, the direct-mapped leads to slightly better average performance. Since CPU time is our bottom-line evaluation.

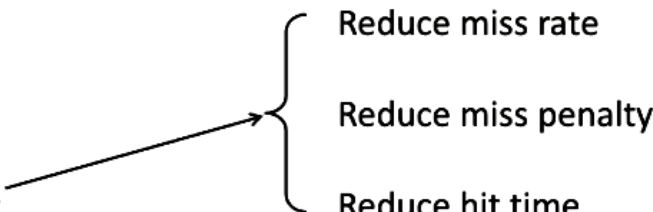
## How to Improve

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

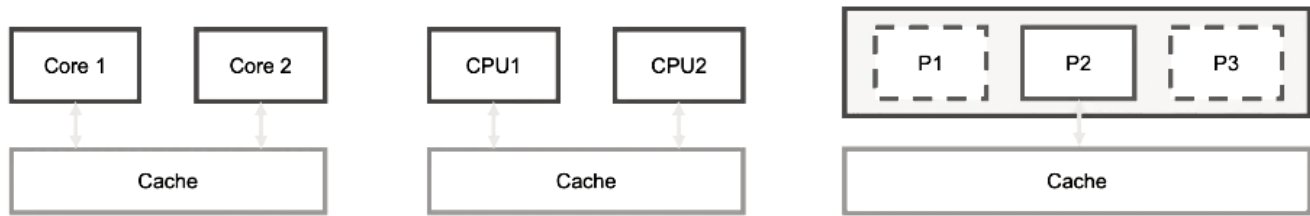
$$AMAT = HitTime + MissRate \times MissPenalty$$

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 time to hit in the cache
  - small and simple caches, avoiding address translation, pipelined cache access, and trace caches
4. Reduce the miss penalty and miss rate via parallelism
  - non-blocking caches, hardware prefetching, and compiler prefetching

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

# Cache Coherence



Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

# Cache Coherence

**Coherence** defines what values can be returned by a read.

A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.

A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.

Writes to the same location are *serialized*. (Writes to the same location by any two processors are seen in the same order by all processors.)

**Consistency** determines when a written value will be returned by a read.

If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A.

- **Migration:** A data item can be moved to a local cache and used there in a transparent fashion. Migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.

- **Replication:** When shared data are being simultaneously read, the caches make a copy of the data item in the local cache. Replication reduces both latency of access and contention for a read shared data item.

- Causes of Cache coherence problems

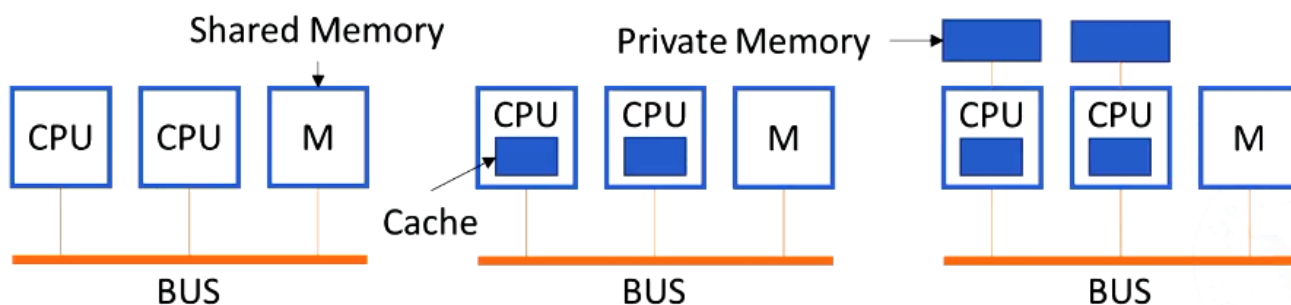
- In modern parallel computers, processors often have Cache. Memory data may have multiple copies in the entire system. This leads to the [cache coherence problem](#).

- Cache coherence protocol

- A set of rules implemented by Cache, CPU, and memory to prevent different versions of the same data from appearing in multiple Caches forms a [cache coherence protocol](#).
  - [Bus snooping protocol](#)
  - [Directory based protocol](#)

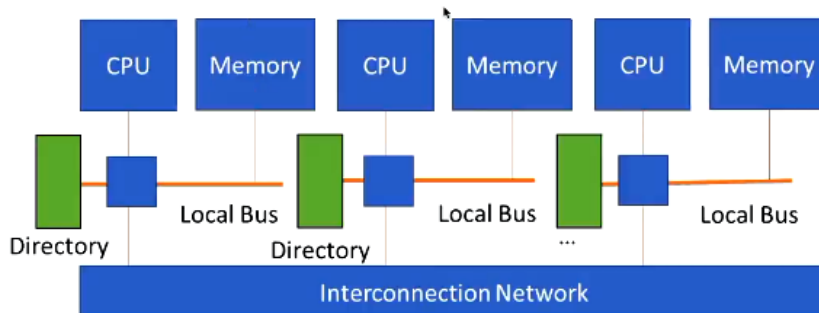
- For UMA: [Snoopy coherence protocols](#)

- In the snoopy coherence protocols, all processors snoop the bus. When a processor modifies the data in the private cache, it broadcasts invalid information or updated data on the bus to invalidate or update other copies.



- For NUMA: [Directory protocol](#)

- The directory protocol uses a directory to record which processors in the system have copies of certain storage blocks in the cache. When a processor wants to write a shared block, it sends an invalid signal to those processors that have copies of the block through the directory in a "point-to-point" way, so that all other copies are invalidated.



- *Snoopy coherence protocols*

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

An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches.

## Snoopy Coherence Protocols

- **Write-through cache coherency protocol**
  - While writing the data in the cache line, the content in the corresponding memory is also modified, and the data in the memory is kept up to date at any time.
- **Write-back cache consistency protocol**
  - The write operation does not directly write to the memory. On the contrary, when the cache line is modified, a certain bit in the cache is set to indicate that the data in the cache line is correct but the data in the memory is out of date. Of course, the line will eventually be written back to memory, but it may be after multiple write operations.

# Write-through Cache Coherency Protocol

- Four situations when the monitoring cache performs read and write operations according to this protocol

	Local Request	Remote Request
Read Miss	Access data from memory	
Read Hit	Use local cache data	
Write Miss	Modify data in memory	
Write Hit	Modify cache and memory	Invalidate the cache item

- There are many changes in the basic protocol of write direct Cache consistency
  - Whether to use **Update Strategy** or **Invalidate Strategy** for remote write hits
  - Whether to transfer the corresponding word into the cache when the cache write is missing, this is whether to use the **Write-allocate Policy**.

## Write Invalidation Protocol

three block states (**MSI protocol**)

- **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;

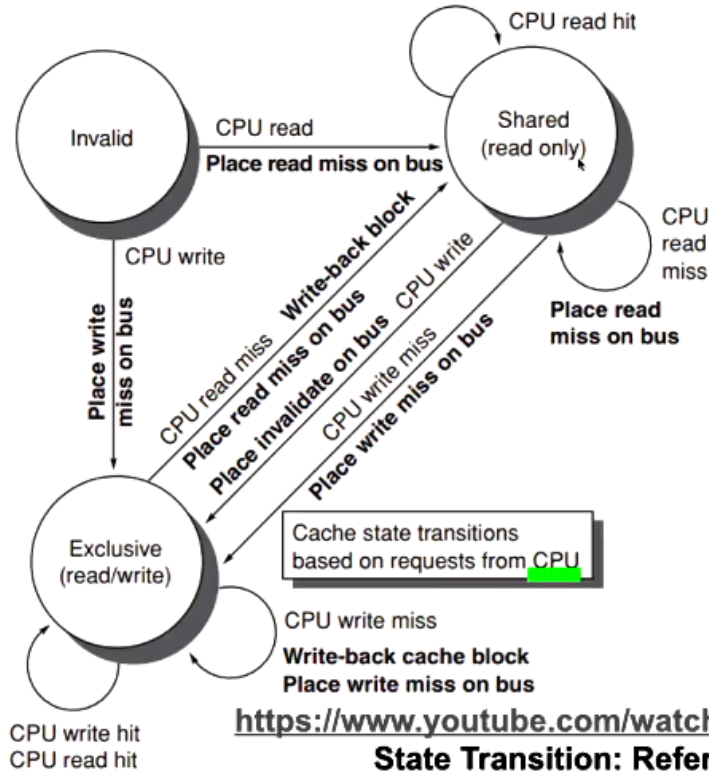
implies that the block is **exclusive**

protocol: 协议

# Write Invalidation 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 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 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 memory to service read miss.
Read miss	bus	modified	coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	bus	shared	coherence	Attempt to write shared block; invalidate the block.
Write miss	bus	shared	coherence	Attempt to write block that is shared; 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.

# Write Invalidation Protocol



cache 中很多问题可以转换成状态机的问题

## MSI Extensions

- **MESI**

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

exclusive->read by others->shared

exclusive->write->modified

add figures?

refer to <https://www.youtube.com/watch?v=OLGEtXV4U3I>

**MESI writes exclusive to modified silently, without broadcast on bus**

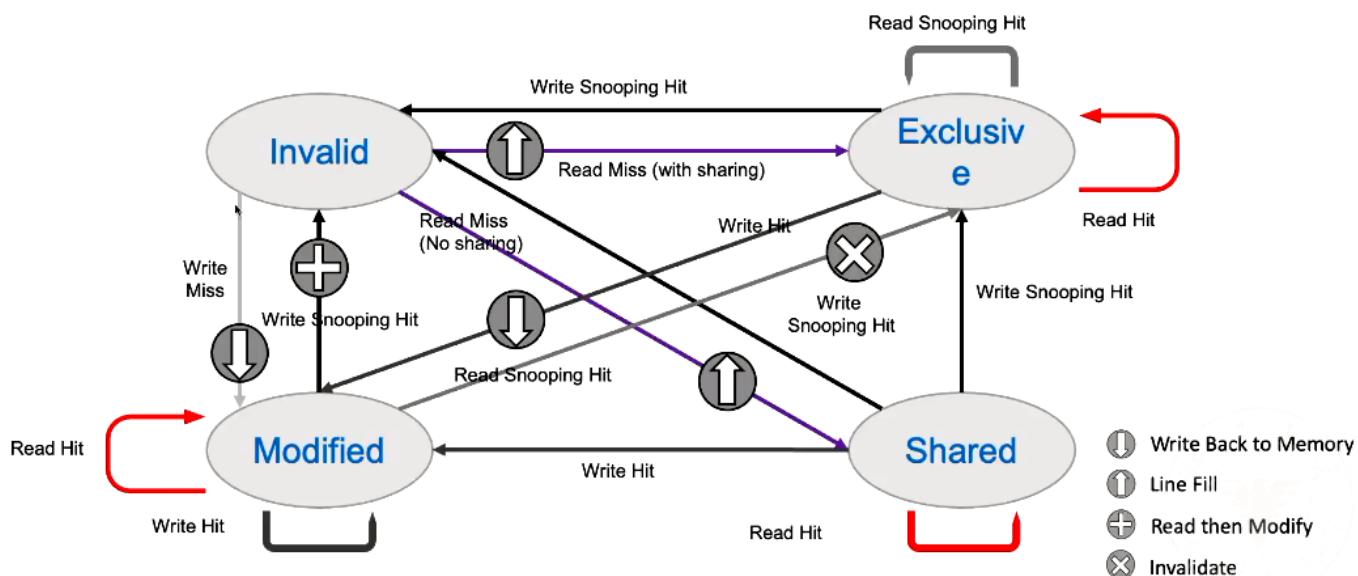


# Write-back Cache Coherency Protocol

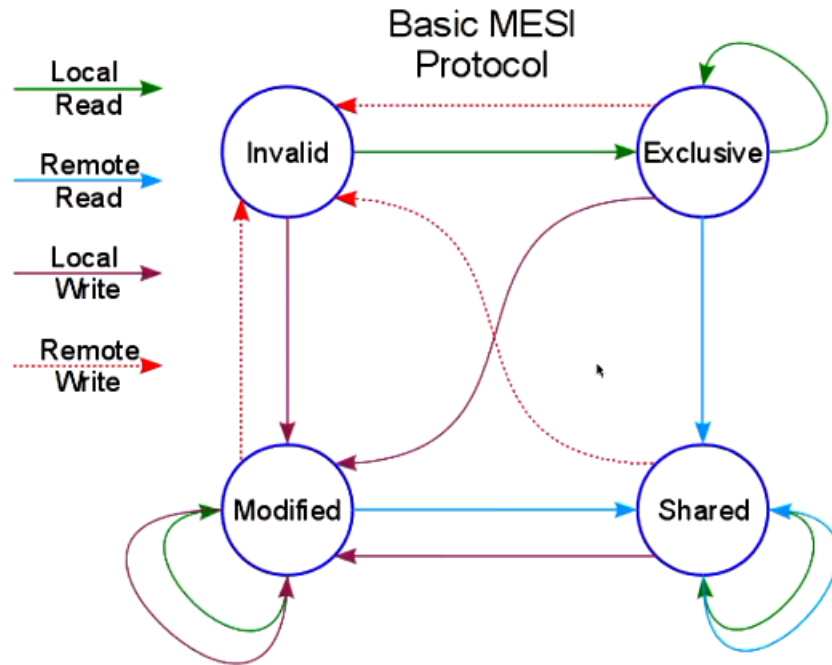
- MESI protocol: It is named after the initial letter of the four states used in the protocol. Each item in this protocol is in one of the following four states:
  - Invalid: The data contained in the cache item is invalid.
  - Shared: This row of data exists in multiple cache items, and the data in 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.

e 代表和 memory 一致

## MESI protocol state transition rules



# MESI protocol state transition rules



多 cache，知道有这些协议，是拓展

会单机中的 Cache 四个策略即可

## CPU 漏洞实战分析

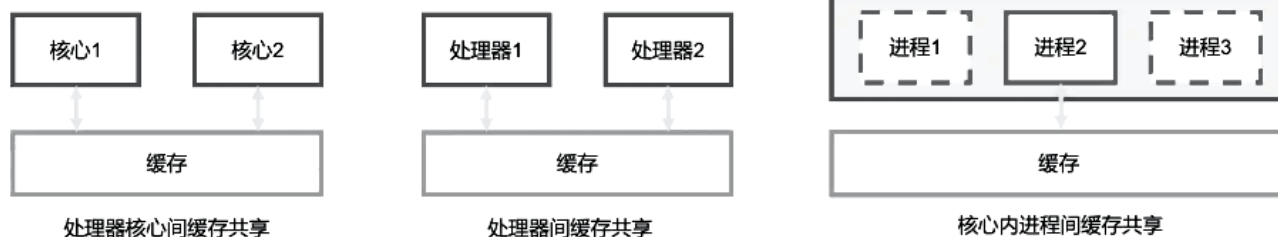
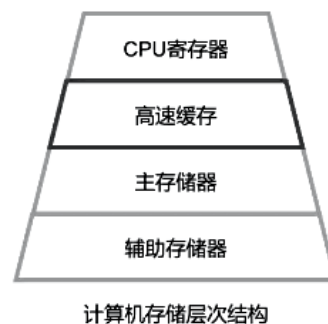
Meltdown & Spectre

## 相关背景

**缓存 (Cache) :** CPU 和主存之间数据交互的桥梁

- 存储敏感数据、密钥等信息而**不易被察觉**
  - 一旦受攻击，敏感信息易被泄露
- 不同应用的数据**共享**同一块缓存且可以互相**替换**
  - 突破安全隔离边界

**缓存侧信道攻击**指攻击者利用访存时延差异推测受害者的访存行为或泄露机密信息，其具有**细粒度、高隐蔽性**等特点。针对处理器缓存的攻击很容易突破安全隔离边界实现**敏感数据泄露**，同时可以**跨平台和CPU**，影响范围广。



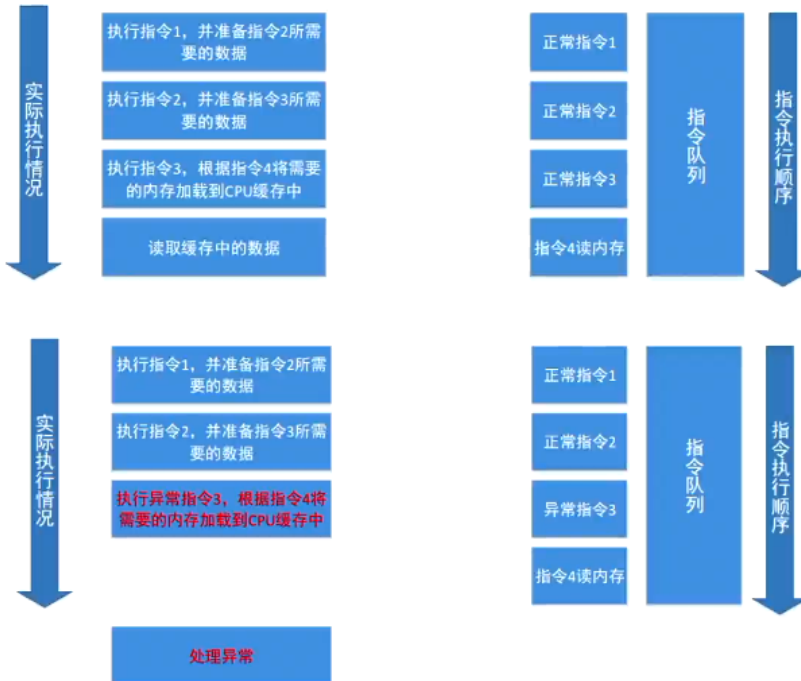
## 漏洞介绍

Meltdown 能够在linux、macos、windows等主流OS上，利用intel的CPU漏洞，来突破内存独立性的限制，能够通过用户程序去访问整个内核空间和其他用户其他程序的空间。其问题的根源在于利用了Intel CPU的乱序执行技术，通过对内存的响应时间差来建立一个侧信道攻击，破坏了位于用户和操作系统之间的基本隔离，“融化”了由硬件来实现的安全边界，因此低权限用户级别的应用程序可以实现“越界”访问系统级的内存，造成数据的泄露。



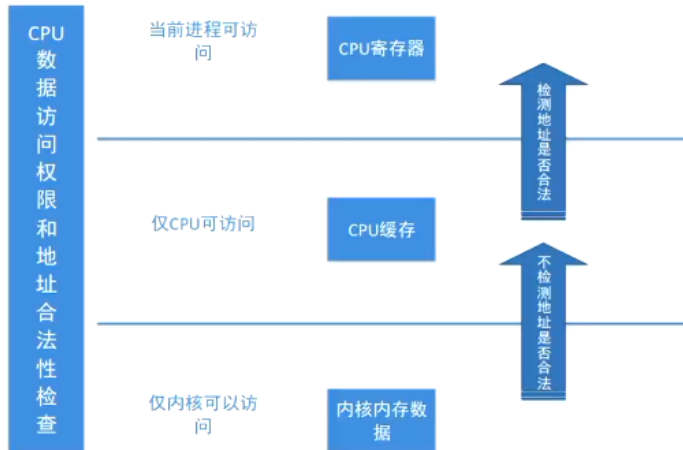
Spectre则是破坏了不同应用程序之间的隔离。问题的根源在于预测执行 (speculative execution)，这是一种优化技术，处理器会推测在未来有用的数据并执行计算。这种技术的目的在于提前准备好计算结果，当这些数据被需要时可立即使用。在此过程中，Intel没有很好地将低权限的应用程序与访问内核内存分开，这意味着攻击者可以使用恶意应用程序来获取应该被隔离的私有数据。

## CPU缓存缺陷分析



对于具有预测执行能力的处理器,在实际CPU执行过程中指令4所需的内存加载环节不依赖于指令3是否能够正常执行,而且从内存到缓存加载这个环节不会验证访问的内存是否合法有效。即使指令3出现异常,指令4无法执行,但指令4所需的内存数据已加载到CPU缓存中,这一结果导致指令4即使加载的是无权限访问的内存数据,该内存数据也会加载到CPU缓存中,因为CPU是在缓存到寄存器这个环节才去检测地址是否合法,而CPU分支预测仅仅是完成内存到CPU缓存的加载,实际指令4并没有被真正的执行,因此非法访问并未触发异常的。

## CPU缓存缺陷分析

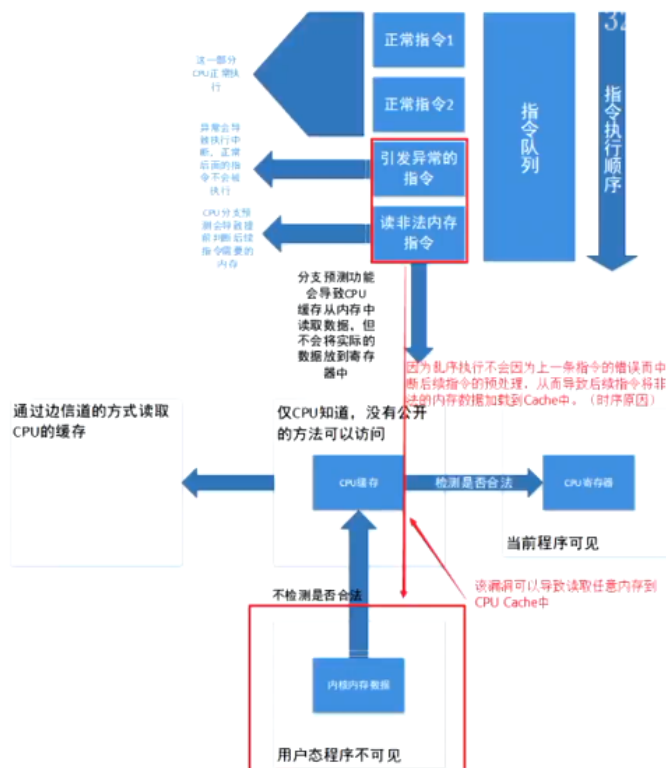


CPU缓存过程对于用户是不可访问的,只有将具体的数据放到CPU寄存器中用户才拥有访问权限,同时用户态程序也没有权限访问内核内存中的数据,因此CPU采用这种逻辑是没有问题的,但是如果方法可以得到CPU缓存中的数据,那么这种逻辑就存在缺陷。

## 侧信道攻击缓存

多核之间实现cache数据共享，从而有效改善了CPU与内存访问速度之间的矛盾。而cache命中和失效对应响应时间有差别，缓存侧信道攻击正是利用CPU缓存与系统内存的读取的时间差异，从而变相猜测出CPU缓存中的数据，结合前边的缓存缺陷部分内容，产生如右图的结果。

对于CPU缓存中的数据，在用户态和内核态都是无法正常访问的，除非当CPU缓存中的数据保存到寄存器中时，会被正常的读取；除此之外，是可以通过侧信道的方式读取CPU的缓存的。

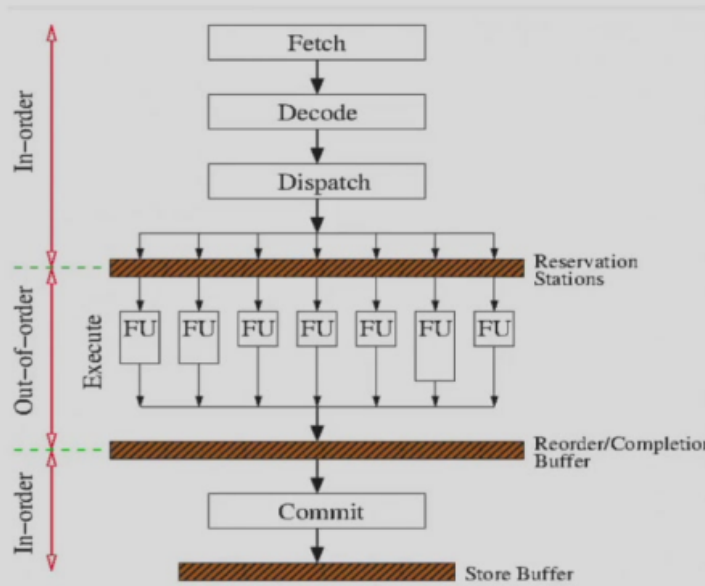


## Meltdown 漏洞原理

Meltdown攻击利用现代CPU中乱序执行 ( out-of-order execution ) 的特性，彻底攻破原本由硬件保证的内存隔离，使得一个仅仅具有普通进程权限的攻击者可以用简单的方法来读取内核内存。

乱序执行是指当CPU中的某些指令需要等待某些资源，比如内存读取时，CPU不会在当前指令停止，而是利用空闲的计算能力继续执行后续的指令。这大大地增加了计算能力的利用率，从而提升了CPU性能。在支持乱序执行的CPU上，指令的执行并不是顺序进行的。比如后面的指令可能在前面指令执行结束之前就开始执行。然而，为了保证程序的正确性，指令retirement必须是顺序进行的，而CPU的安全检查是在指令retirement时才会进行。这样的结果是，在CPU对某一条指令进行安全检查之前，一部分在该指令后面的指令会由于CPU的乱序执行而被提前执行。

## Meltdown 攻击过程



每个阶段执行的操作如下：

- 1) 获取指令，解码后存放到执行缓冲区Reservations Stations；
- 2) 乱序执行指令，结果保存在一个结果序列中；
- 3) 退休期Retired Circle，重新排列结果序列及安全检测（如地址访问的权限检查），提交结果到寄存器。

## Meltdown 攻击示例

1. ;rcx = kernel address
2. ;rbx = probe array
3. mov al, byte [rcx]
4. shl rax, 0xc //一个内存页大小为 4KB, 将 rax 值乘 4096, 便于下一条指令实现对探测数组 probe\_array(rbx[al\*4096])
5. mov rbx, qword [rbx + rax] //的访问, 对于不同的 al 值, 将导致不同的内存页被访问并存放到 CPU 缓存中

Meltdown漏洞的利用过程有4个步骤：

- 1) 指令获取解码；
- 2) 乱序执行3条指令，指令4和指令5要等指令3中的读取内存地址的内容完成后才开始执行，指令5会将要访问的rbx数组元素所在的页加载到CPU Cache中；
- 3) 对2)的结果进行重新排列，对3-5条指令进行安全检测，发现访问违例，会丢弃当前执行的所有结果，恢复CPU状态到乱序执行之前的状态，但是并不会恢复CPU Cache的状态；
- 4) 通过缓存侧信道攻击，不断遍历加载rbx[al\*4096]，由于该数据此时已经在缓存中，攻击者总会遍历出一个加载时间远小于其它的数据，推测哪个内存页被访问过了，从而推断出被访问的内核内存数据。

## Spectre 漏洞原理

Spectre攻击利用了CPU的预测执行对系统进行攻击。预测执行是另外一种CPU优化特性。在分支指令执行时，由于分支指令执行可能需要内存读取（上百个CPU周期），在分支指令执行结束之前，CPU会预测哪一个分支会被运行，提取相应的指令代码并执行，以提高CPU指令流水线的性能。

CPU的预测执行是通过分支预测单元(BPU)进行的。BPU储存了某个分支指令最近执行过的分支跳转的结果。CPU的预测执行遇到分支指令时，会根据BPU的预测结果进行跳转。当预测执行发现预测错误时，预测执行的结果将会被丢弃，CPU的状态会被重置。然而，与乱序执行类似，预测执行对CPU缓存的影响会被保留。Spectre和Meltdown攻击在这一点上比较类似。

### Spectre 攻击过程

Spectre攻击主要分为三个阶段：

- 训练CPU的分支预测单元（BPU），使其在运行漏洞利用代码时会进行特定的预测执行；
- 预测执行使得CPU将要访问的地址的内容读取到CPU Cache中；
- 通过缓存测信道攻击，可以知道哪一个数组元素被访问过，也即对应的内存页存放在CPU Cache中，从而推测出地址的内容。

### Spectre 攻击示例

```
1.  if(x<array1_size){  
2.     y=array2[array1[x]*256];  
3.     //do something using Y that is  
4.     //observable when speculatively executed  
5. }
```

## 缓解措施

1) 序列化指令：限制预测执行。

问题：无法在所有CPU或系统配置中工作。

2) 插入推测执行阻止指令：在每个条件分支及其目标之后的指令处插入此指令。

问题：严重降低性能。

3) 微代码修复现有处理器

问题：修补程序可能会阻止推测执行或阻止推测内存读取，但这会极大地破坏性能。

## Meltdown & Spectre

1) Meltdown：依赖乱序执行

Spectre：使用分支预测来实现推测执行

2) Meltdown：利用了Intel处理器特有的提权漏洞，该漏洞导致推测性执行的指令可以绕过内存保护。从用户空间访问内核内存，这种访问会导致异常，但是在异常被发布之前，访问之后的代码会通过缓存信道泄漏被访问内存的内容。

Spectre：Spectre的利用难度要大很多，对分支预测的利用，无论是分析成本，还是攻击成本，都大了不少，包括AMD、ARM、Intel等在内的大多数处理器都存在该漏洞。此外，被广泛用于缓解Meltdown攻击的KAISER补丁并不能防止Spectre攻击。



# 实验环境

工具清单	
虚拟机	VMware Workstation11.0.0
系统镜像1	ubuntu-16.04.3-desktop-amd64.iso
内核版本1	Linux ubuntu 4.13.0-41-generic
系统镜像2	CentOS-7-x86_64-DVD-1708.iso
内核版本2	Linux localhost.localdomain 3.10.0-693.el7.x86_64
物理处理器型号	Intel (R) Core (TM) i5-7300HQ @2.5GHZ

- **CVE-2017-5753越界检查绕过 (Spectre变种1)**  
 影响: 内核和所有软件  
 缓解: 使用修改后的编译器重新编译软件和内核, 该编译器将LFENCE操作码引入到生成的代码中的适当位置  
 缓解的性能影响: 可以忽略不计
- **CVE-2017-5715命令注入 (Spectre变种2)**  
 影响: 内核  
 缓解1: 通过微码更新的新操作码, 应由最新的编译器使用以保护BTB (通过刷新间接分支预测器)  
 缓解2: 将“retpoline”引入编译器, 并使用它重新编译软件/操作系统  
 缓解的性能影响: 缓解1高, 缓解介质2, 具体取决于CPU

# 漏洞检测

- **CVE-2017-5754流氓数据缓存加载 (Meltdown)**  
 影响: 内核  
 缓解: 更新内核 (使用PTI / KPTI补丁), 更新内核就足够了; 缓解的性能影响: 低到中等
- **CVE-2018-3640流氓系统寄存器读取 (变体3a)**  
 影响: TBC  
 缓解: 仅限微码更新; 缓解的性能影响: 可以忽略不计
- **CVE-2018-3639随机存储绕过 (变体4)**  
 影响: 使用JIT的软件 (没有针对内核的已知利用)  
 缓解: 微码更新+内核更新使受影响的软件能够自我保护; 缓解的性能影响: 低到中等

<http://mdsattacks.com/> diagram

