CS359 Computer Architecture(上)

Yanjie Ze, June 2021

This file includes Chapter 2,3,4,5,6, noted by Yanjie Ze.

2 MIPS ISA

MIPS-32

R型：6+5+5+5+5+6

I型：6+5+5+16

J型：6+26

MIPS register

byte addressing

一个地址对应一个byte。

Big endian and Little Endian

记住：小字节序：低位对低地址

Load Byte and Save Byte

lb和sb指令对于byte进行操作。

Branch Instructions

```
bne $s0, $s1, Lbl : go to Lbl if s0 != s1
beq $s0, $s1, Lbl : go to Lbl if s0 = s1
```

Branch Destination

Branch指令的目标地址计算公式：

$$PC = PC + 4 + signext(offset << 2)$$

为什么用sign extension而不是zero extension？：

> In the process, why using sign-extend instead of zero-extend? That's something about two's-complement. For an w-bit integer x = [x_{w-1},x_{w-2},....x_1,x_0], the most significant bit of the number is defined as negative weight so x = -$x_{w-1}2^{w-1}$ +$x_{w-2}2^{w-2}$+$x_{w-3}2^{w-3}$+...+$x_0 2^0$, assume x_{w-1} = 1 or x < 0. Then if we apply zero-extend, the most significant bit of the number has the value of 0, causing x > 0. But if we apply sign-extend, making this integer u-bit (u>w), then x' = - 2^{v-1}+2^{v-2}+...+2^{w-1}+....+$x_0 2^0$, It can be proved that x = x'.

☆Problem of Branching Far Away

有可能branch指令的offset不够用，无法到达更远的指令。

因为offset只有16位。

因此可以用jump指令。

具体来说：

当beq的offset太大了的时候，编译器会自动转化成bne+j的两个指令的组合，代码如下图所示。

When the branch destination is so far that it cannot be captured in 16 bits, the assembler will do an extal job to solve this problem. The assembler will change the `beq` instruction

```
beq $s0, $s1, L1
```

to a `bne` instruction and an unconditional jump instruction.

```
    bne $s0, $s1, L2
    j   L1   //jump instruction can achieve 26-bits address
L2: ...
```

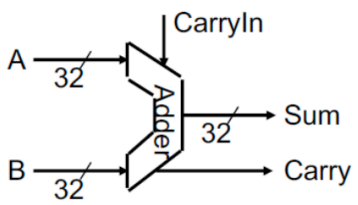
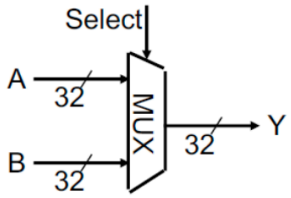
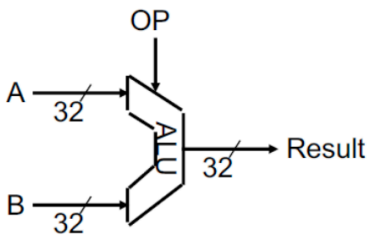
Remember, this job will be done by the assembler, **instead of programmers**. So we don't need to take this into consideration.

3 Single Cycle Processor

这一部分，只讲如下几个指令的实现。

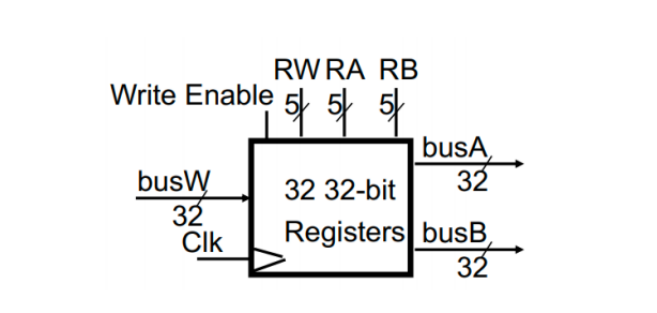
R Type		I Type				J Type
add	sub	ori	lw	sw	beq	j
arithmetic-logical			memory-reference		control flow	

Combinaitional Elements

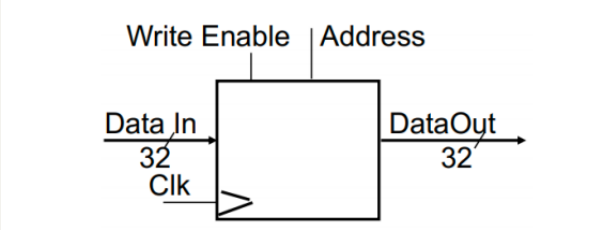
Adder	MUX	ALU
		

Storage Elements (State Elements)

register file:



idealized memory:



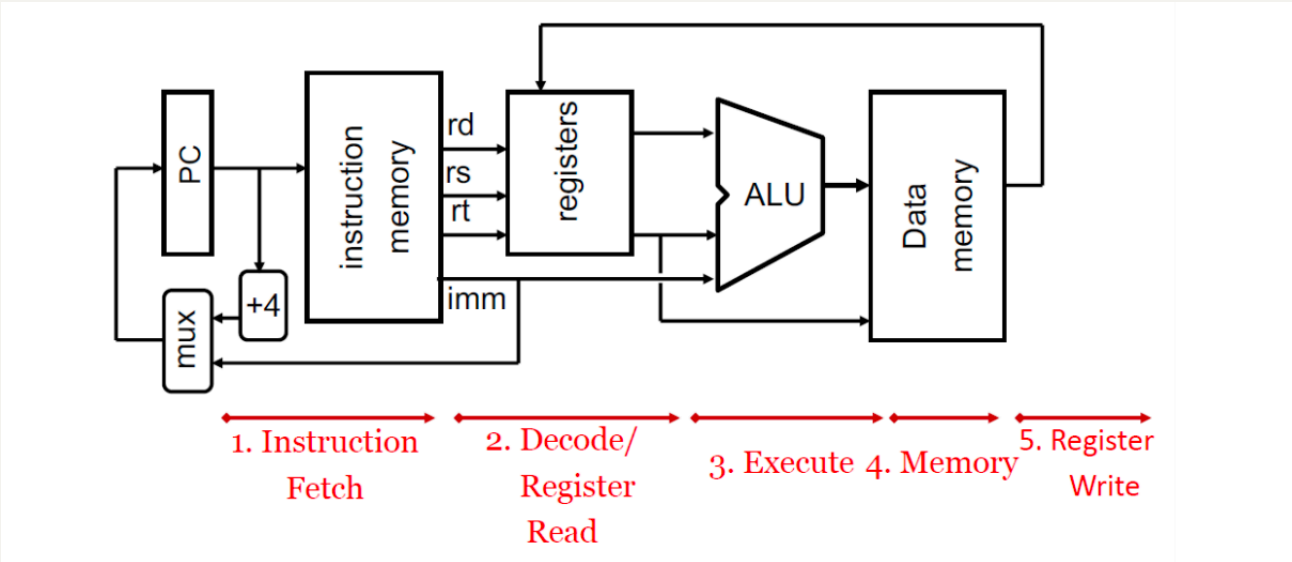
Clocking Methodology

只在时钟上升沿的时候更新。

一个时钟周期：

$$CycleTime = hold + LongestDelayPath + SetUp + ClockSkew$$

Data Path



1. Fetch: read `M[PC]`, then `PC<-PC+4`

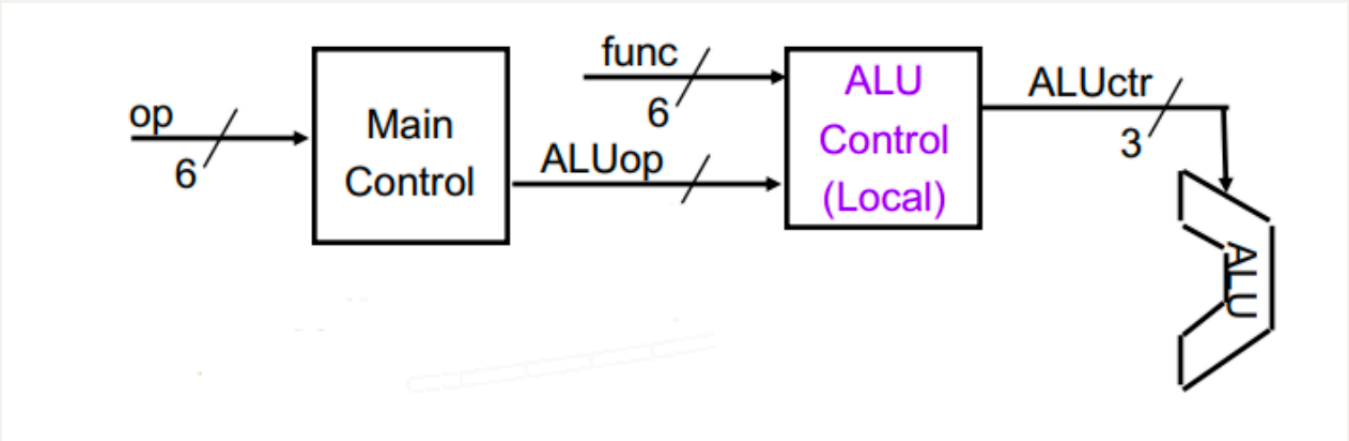
2. Decode
3. Execute:
    - R type: `R[rd]<-R[rs] op R[rt]`
4. Memory
5. Write

Control Signal Logic

func	10 0000	10 0010	xxxxxx	xxxxxx	xxxxxx	xxxxxx	xxxxxx
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
Instruction	*add*	*sub*	*ori*	*lw*	*sw*	*beq*	*jump*
RegDst	1	1	0	0	X	X	X
ALUSrc	0	0	1	1	1	0	X
MemtoReg	0	0	0	1	X	X	X
RegWrite	1	1	1	1	0		0
MemWrite	0	0	0	0	1	0	O
Branch	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	X	X	0	1	1	X	X
ALUctr	*Add*	*Subtr*	*Or*	*Add*	*Add*	*Subtr*	XXX

Two-level Decoding

- 指令中的op经过main control获得ALUop和其他控制信号
- ALUop结合func，经过ALU Control获得ALUctr控制信号



Generating ALUctr

ALUop	*func*	*ALUOperation*	*ALUctr*
0 0 0	X X X X	*Add*	0 0 0
0 X 1	X X X X	*Subtract*	0 0 1
0 1 0	X X X X	*Or*	1 1 0
1 X X	0 0 0 0	*Add*	0 0 0
1 X X	0 0 1 0	*Subtract*	0 0 1
1 X X	0 1 0 0	*And*	0 1 0
1 X X	0 1 0 1	*Or*	1 1 0
1 X X	1 0 1 0	*Subtract*	0 0 1

Features of Single Cycle Processor

1. Mutually Exclusive Datapath Resources
2. Multiplexers
3. Write signals
4. Cycle Time Determined by Longest Path

Disadvantages of Single Cycle Processor

- **Low Time Efficiency.**
  It uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instructions. And that becomes even more problematic for more complex instructions like floating point multiply.

- **Poor Space Utilization.**
  It is wasteful of area since many functional units must be duplicated since they can not be shared during a clock cycle.

4 Multi Cycle Processor

Step	R-type	Mem Ref	Branch	Jump
Instr fetch	IR = Memory[PC]; PC = PC + 4;			
Decode	A = Reg[IR[25-21]]; B = Reg[IR[20-16]]; ALUOut = PC +(sign-extend(IR[15-0])<< 2);			
Execute	ALUOut = A op B;	ALUOut = A + sign-extend (IR[15-0]);	if (A==B) PC = ALUOut;	PC = PC[31-28] ||(IR[25-0] << 2);
Memory access	Reg[IR[15-11]] = ALUOut;	MDR = Memory[ALUOut]; or Memory[ALUOut] = B;		
Write-back		Reg[IR[20-16]] = MDR;		

5 Pipelining

Pipelining Speedup

理想情况是:

$$Time\ to execute\ an\ instruction_{pipelining} = \frac{Time\ to\ execute\ an\ instruction_{sequential}}{num_{stages}}$$

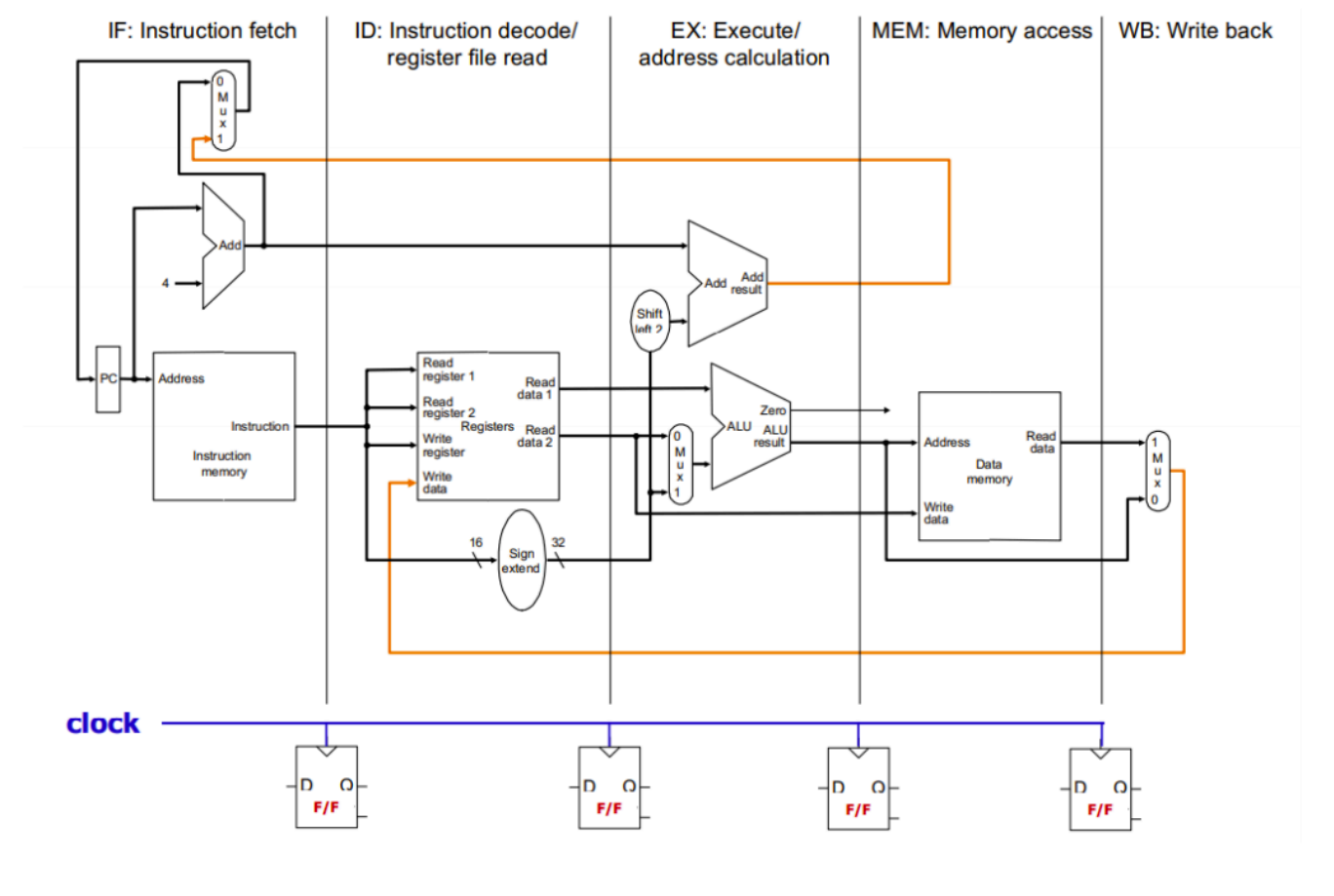
实际是要小一点的。

Difference between Pipelining and Multi-cycle

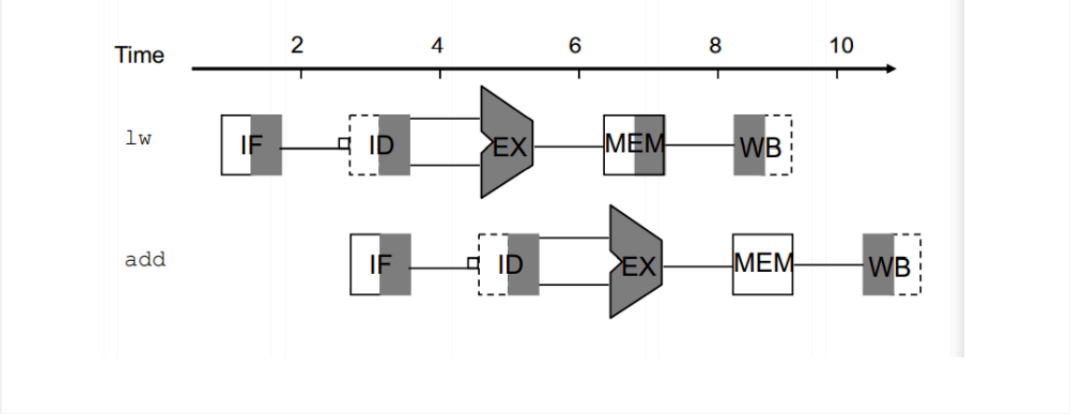
- 一个时钟周期内，多周期处理器只有一条指令在执行，但是pipeline有多条。
- pipeline中的指令都要经过五个stage，多周期处理器的指令只经过它所需要的stage。

Five stages

IF, ID, EX, MEM, WB



右半边黑色表明在读；左半边黑色表明在写。

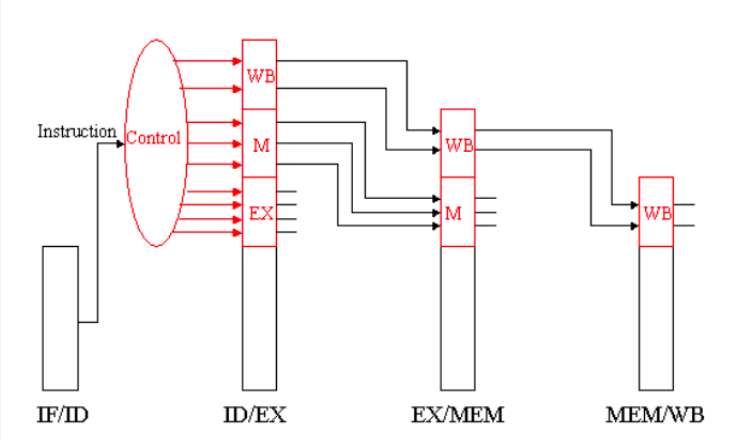


一个例子：lw

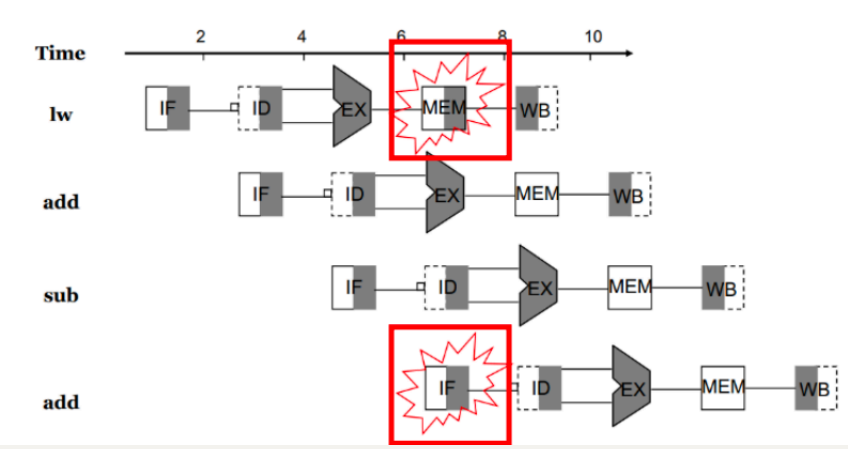
1. IF：instruction存入IF/ID。PC+4也要存入IF/ID。
2. ID：把rs，rt，rd存入ID/EX。还有sign extension后的immediate存入ID/EX。PC+4也要存入ID/EX。
3. EX：把rs寄存器和immediate用ALU加起来，再存入EX/MEM。
4. MEM：读取数据，存入MEM/WB。
5. WB：写入register file。由于这一步需要用到rt，rt要一直保存在流水线寄存器中。

Control Signals

把instruction从IF/ID取出来，在ID阶段获得main control。



Structure Hazarads



两个指令用到同一个硬件。

处理方法：

Solutions	Pros	Cons
Stalling: stages "take turns" to use the resource	+ Low cost, simple	- Bubbling, increased CPI & reduced efficiency
Concurrency & pipelined hardware: modify the existing hardware to support concurrent operations + pipelining	+ Good performance	- Complex implementation (e.g. a fully pipelined floating point multiplier)
Duplicate components / separate caches / multiple buffers	+ Good performance + Best for cheap and divisible resources	- Increased cost - Consistency problem (how to make sure all components have the same data?)
Additional ports to a single component	+ Good performance	- Increased cost - Requires more bandwidth

Data Hazards

三种：

- RAR
- WAR
- WAW

三种解决方法：

- stalling

- forwarding
- code scheduling

Solutions	Pros	Cons
Stalling: the dependent instruction is “pushed back” for one or more clock cycles	+ Low cost, simple	- Bubbling, decrease IPC & efficiency
Forwarding / bypassing: the needed data is passed along to ALU ASAP	+ Good performance	- Can’t always avoid stalls by forwarding → heavily relies on timing and schedule
Code Scheduling: insert non-related instructions between the current instruction and the dependent instruction	+ When successful, it has zero cost + Good performance	- Requires an intelligent, hardware-dependent compiler

有时候stalling是不可避免的，因为lw只有在mem阶段才有结果。

Hazard Detection Unit

Compare Hazard Detection unit with Forwarding Unit

	Hazard Detection Unit	Forwarding Unit
Working Stage	ID	EX
Inputs	Instructions	Control signals, instructions and pipeline registers
Outputs	Control signals including PC and IF/ID write	ALU inputs

Control Hazard

解决方法：

1. stall两次
2. 在ID就进行比较，再stall一次
3. branch指令下一条指令从Delayed Slot里面取，这样就不用stall了。
4. Predict-not-taken(predict失败要添加flush)

Ways to solve exceptions

Ways to solve exceptions: Stopping and Restarting Execution

Two properties of the most difficult exceptions:

1. they occur **within instructions**.
2. they must be **restartable**.

When an exception with both properties occurs, we need to safely shut down and save the state so that the instruction can be restarted in the correct state. We can follow these steps:

(1) Force a *trap instruction* into the pipeline on the next IF.

(2) Until the trap is taken, turn off all writes for *the faulting instruction* and for *all instructions that follow in the pipeline.*

(3) After the exception-handling routine in the OS receives control, it immediately *saves the PC of the faulting instruction.*

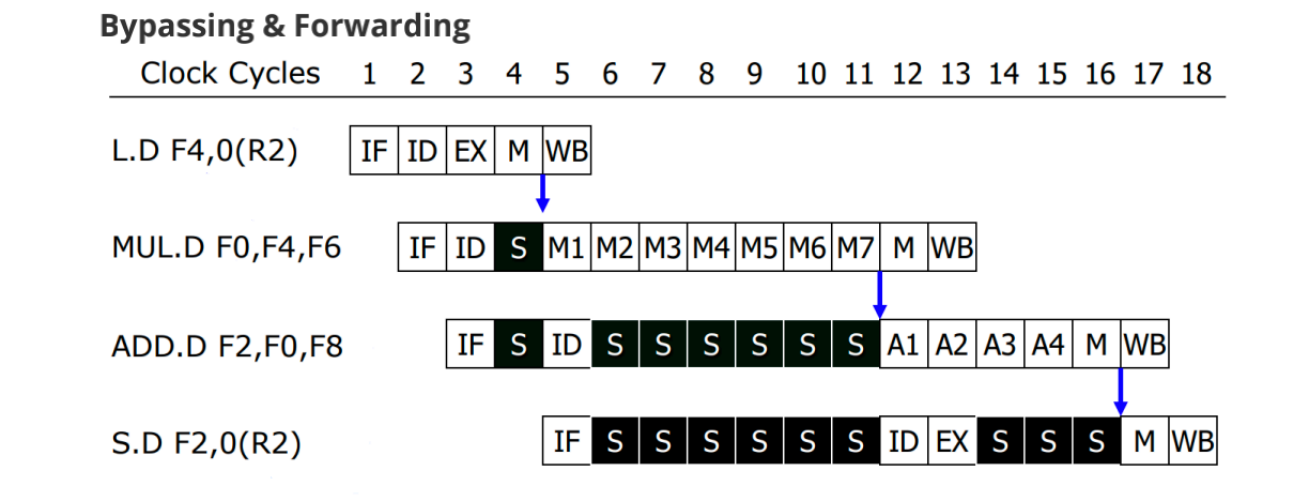
Precise vs Imprecise Exceptions

如果pipeline能完成错误指令前的指令并且可以从错误指令重新开始执行，则是precise的。

Exceptions in MIPS Pipeline

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

Handling Multicycle Operations



Improving Scalar Pipeline

- superscalar pipeline
- superpipeline

6 ILP

Pipeline CPI

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data hazard stalls + Control Stalls

Ideal pipeline CPI is the max performance a pipeline can reach.

Dependency

- Data dependency.
    a. the results of instruction *i* may be used in instruction *j*
    b. instruction *j* has data dependence on instruction *k* and *k* has data dependence on *i*
- Name dependency. Two instructions use the same register or memory location but no flow of information

- Control dependency.
    a. if an instruction depends on a branch then this instruction can not be moved before the branch so that its execution is no longer controlled by the branch.
    b. if an instruction does not depend on a branch then it can not be moved after the branch so that it is controlled by the branch.

Data Hazard

- RAW
- WAW
- WAR

Basic Complier Techniques

- reorder
- loop unrolling

Reducing Branch Costs with Advanced Branch Prediction

- Static branch prediction
- Dynamic branch prediciton

Dynamic branch prediction

- 1-bit prediction

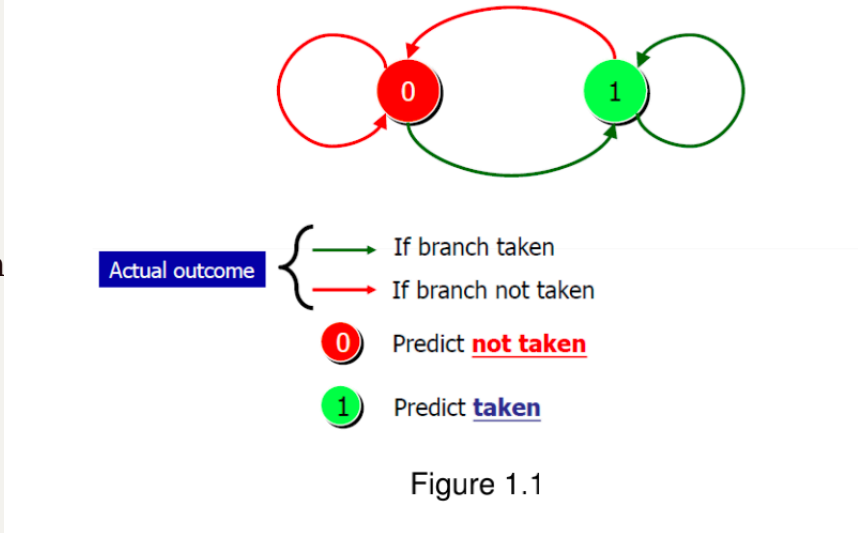
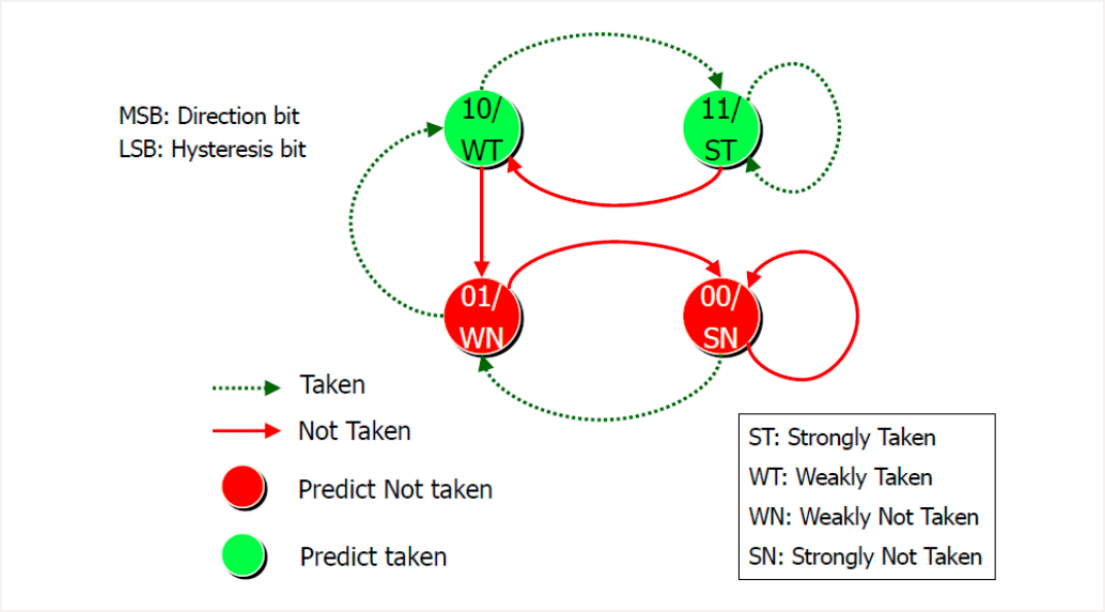


Figure 1.1

- 2-bit prediction



Out-of-order Execution

Dynamic Scheduling

Scoreboard Algorithm