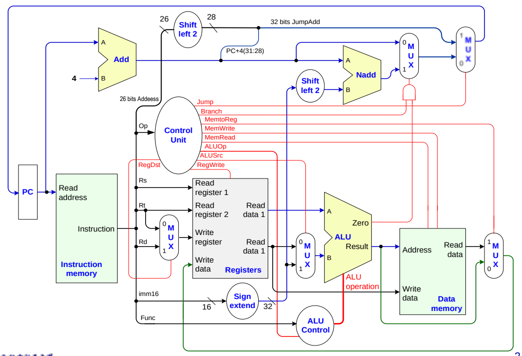
第十五讲

单周期处理器设计



单周期处理器性能分析

MIPS不同类型指令的指令周期

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

数据通路各部分以及各类指令的执行时间

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

单周期处理器性能分析

❖指令执行时间计算

1. 方式一：采用单周期，即所有指令周期固定为单一时钟周期
   - ■ 时钟周期有最长的指令决定（LW指令），为 600ps
   - ■ 指令平均周期 = 600ps
2. 方式二：不同类型指令采用不同指令周期（可变时钟周期）
   - ■ 假设指令在程序中出现的频率
     - – lw指令 ： 25%
     - – sw指令 ： 10%
     - – R型指令 ： 45%
     - – beq指令 ： 15%
     - – j指令 ： 5%
   - ■ 平均指令执行时间

     600*25% + 550*10% + 400*45% + 350*15% + 200*5% = 447.5ps

—若采用可变时钟周期，时间性能比单周期更高；
—但控制比单周期要复杂、困难，得不偿失。
—改进方法：改变每种指令类型所用的时钟数，即采用多周期实现

多周期处理器设计

❖为什么不使用单周期实现方式？

➢ 单周期设计中，时钟周期对所有指令等长

➢ 而时钟周期由计算机中可能的最长路径决定，如：取数指令

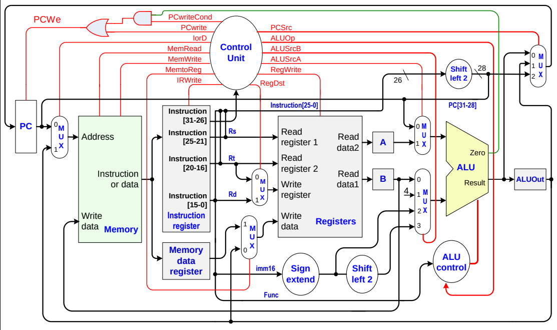
➢ 但某些指令类型本来可以在更短时间内完成，如：跳转指令

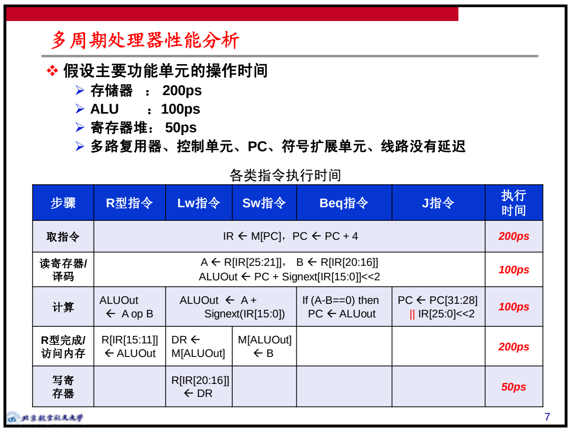
❖多周期方案

➢将指令执行分解为多个步骤，每一步骤一个时钟周期，则指令执行周期为多个时钟周期，不同指令的指令周期包含时钟周期数不一样

➢数据通路中需要增加一个或多个寄存器，以保存指令各执行步骤形成的结果（输出值），以便在指令的后续时钟周期内继续使用

多周期处理器设计





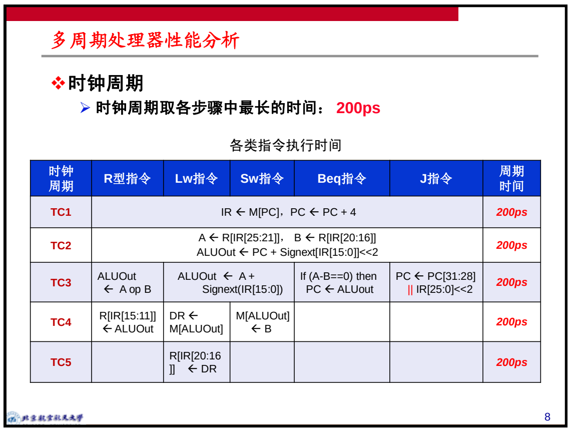
多周期处理器性能分析

❖ 假设主要功能单元的操作时间
- 存储器 ： 200ps
- ALU ： 100ps
- 寄存器堆： 50ps
- 多路复用器、控制单元、PC、符号扩展单元、线路没有延迟

各类指令执行时间

步骤	R型指令	Lw指令	Sw指令	Beq指令	J指令	执行时间
取指令	IR ← M[PC]，PC ← PC + 4					200ps
读寄存器/译码	A ← R[IR[25:21]]， B ← R[IR[20:16]] ALUOut ← PC + Signext[IR[15:0]]<<2					100ps
计算	ALUOut ← A op B	ALUOut ← A + Signext(IR[15:0])		If (A-B==0) then PC ← ALUout	PC ← PC[31:28] || IR[25:0]<<2	100ps
R型完成/访问内存	R[IR[15:11]] ← ALUOut	DR ← M[ALUOut]	M[ALUOut] ← B			200ps
写寄存器		R[IR[20:16]] ← DR				50ps

7



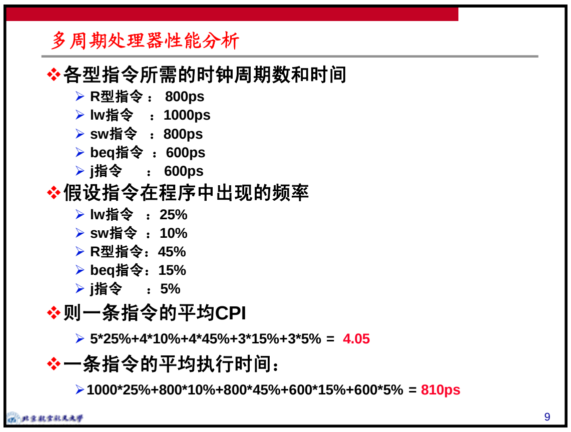
多周期处理器性能分析

❖ 时钟周期
- 时钟周期取各步骤中最长的时间： 200ps

各类指令执行时间

时钟周期	R型指令	Lw指令	Sw指令	Beq指令	J指令	周期时间
TC1	IR ← M[PC]，PC ← PC + 4					200ps
TC2	A ← R[IR[25:21]]， B ← R[IR[20:16]] ALUOut ← PC + Signext[IR[15:0]]<<2					200ps
TC3	ALUOut ← A op B	ALUOut ← A + Signext(IR[15:0])		If (A-B==0) then PC ← ALUout	PC ← PC[31:28] || IR[25:0]<<2	200ps
TC4	R[IR[15:11]] ← ALUOut	DR ← M[ALUOut]	M[ALUOut] ← B			200ps
TC5		R[IR[20:16]] ← DR				200ps

8



多周期处理器性能分析

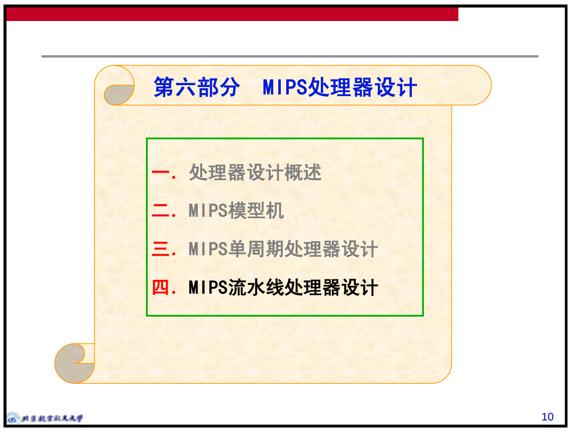
❖各型指令所需的时钟周期数和时间
- R型指令： 800ps
- lw指令 ： 1000ps
- sw指令 ： 800ps
- beq指令 ： 600ps
- j指令 ： 600ps

❖假设指令在程序中出现的频率
- lw指令 ： 25%
- sw指令 ： 10%
- R型指令： 45%
- beq指令： 15%
- j指令 ： 5%

❖则一条指令的平均CPI
- 5*25%+4*10%+4*45%+3*15%+3*5% = 4.05

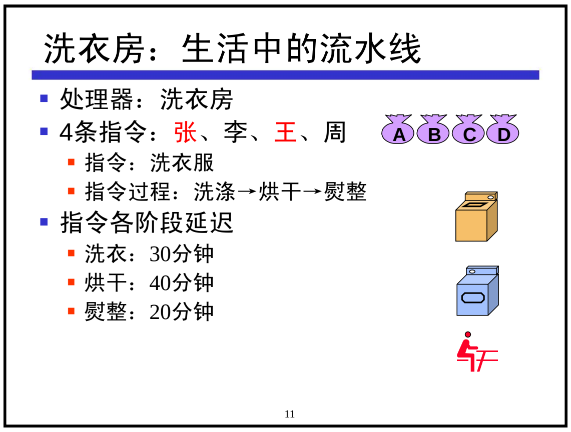
❖一条指令的平均执行时间：
- 1000*25%+800*10%+800*45%+600*15%+600*5% = 810ps

9



第六部分 MIPS处理器设计

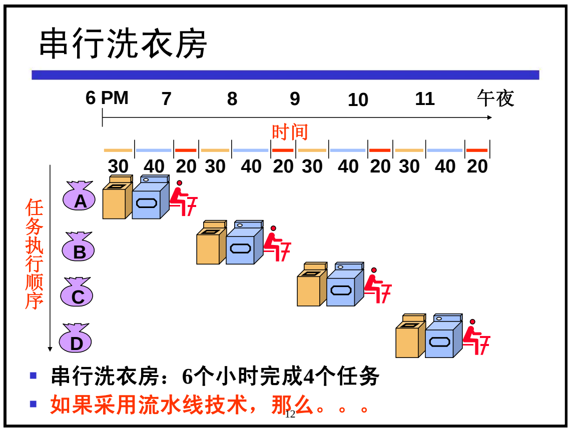
10



洗衣房：生活中的流水线

- 处理器：洗衣房
- 4条指令：张、李、王、周
  - 指令：洗衣服
  - 指令过程：洗涤→烘干→熨整
- 指令各阶段延迟
  - 洗衣：30分钟
  - 烘干：40分钟
  - 熨整：20分钟

11

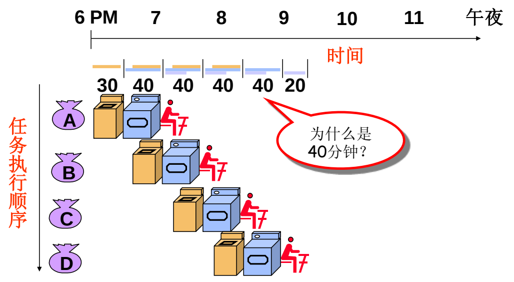


串行洗衣房

- 串行洗衣房：6个小时完成4个任务
- 如果采用流水线技术，那么。。。

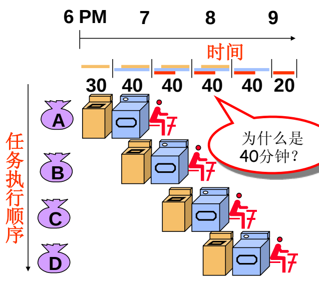
12

流水化洗衣房：尽可能早地开始工作



- 流水化洗衣房：3.5小时完成4个任务

流水线性质



- 流水线不改善单个任务处理延迟(latency)，但改善了整体工作负载的吞吐率
- 流水线速率受限于最慢的流水段
- 多个任务同时工作，但占用不同的资源
- 潜在加速比＝流水线级数
- 流水段执行时间不平衡，则加速比下降
- 填充流水线和排放流水线，加速比下降

Recall: 5 Stages of MIPS Datapath

1) IF: Instruction Fetch, Increment PC

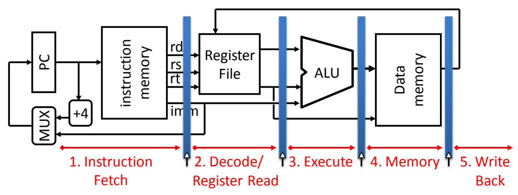
2) ID: Instruction Decode, Read Registers

3) EX: Execution (ALU)
   Load/Store: Calculate Address
   Others: Perform Operation

4) MEM:
   Load: Read Data from Memory
   Store: Write Data to Memory

5) WB: Write Data Back to Register

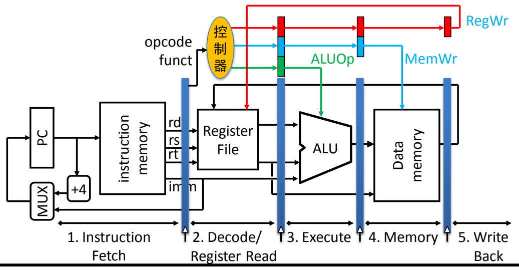
Pipelined Datapath



- Add registers between stages
  - Hold information produced in previous cycle
- 5 stage pipeline
  - Clock rate *potentially* 5x faster

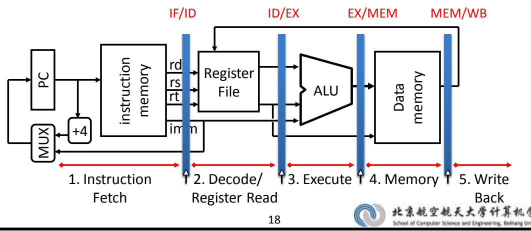
流水的控制信号

- 控制器：译码产生控制信号，与单周期完全相同
- 控制信号流水寄存器：控制信号在寄存器中传递，直至不再需要

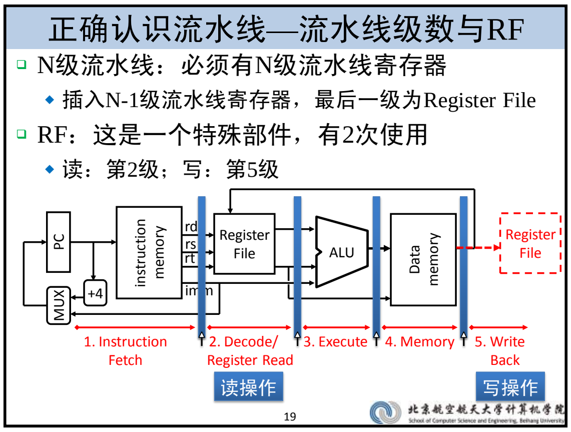


正确认识流水线—流水线寄存器

- 命名法则：前级/后级
  - 示例：IF/ID，前级为读取指令，后级为指令译码(及读操作数)
- 功能：时钟上升沿到来时，保存前级结果；之后输出至下级组合逻辑
  - 也可能直接连接到下级流水线寄存器
    - 例如ID/EX保存的从RF读出的第2个寄存器值，就直接传送到EX/MEM



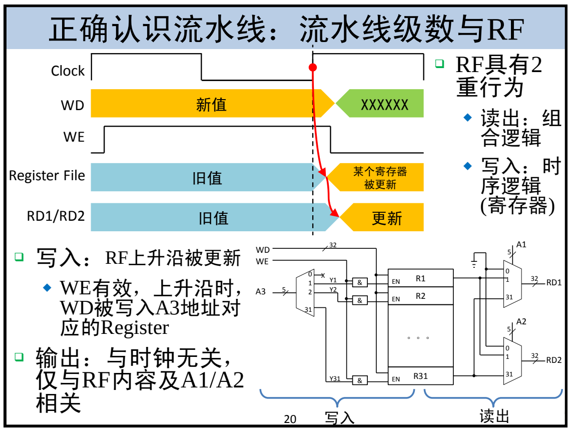
北京航空航天大学计算机学院 School of Computer Science and Engineering, Beihang University



正确认识流水线—流水线级数与RF

- N级流水线：必须有N级流水线寄存器
  - 插入N-1级流水线寄存器，最后一级为Register File
- RF：这是一个特殊部件，有2次使用
  - 读：第2级；写：第5级

PC
MUX
+4
instruction memory
rd
rs
rt
imm
Register File
ALU
Data memory
Register File
1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Write Back
读操作
写操作

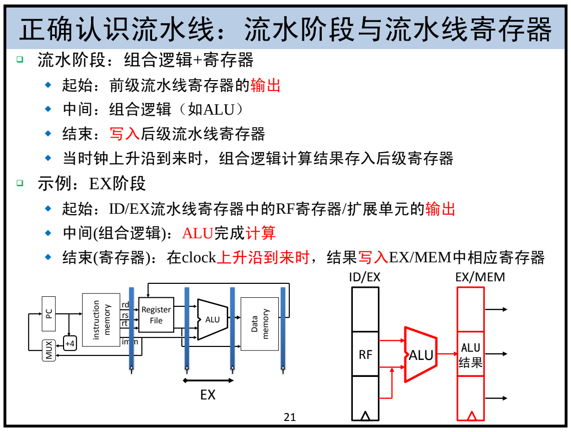


正确认识流水线：流水线级数与RF

Clock
WD
新值
XXXXXX
WE
Register File
旧值
某个寄存器被更新
RD1/RD2
旧值
更新

- RF具有2重行为
  - 读出：组合逻辑
  - 写入：时序逻辑(寄存器)
- 写入：RF上升沿被更新
  - WE有效，上升沿时，WD被写入A3地址对应的Register
- 输出：与时钟无关，仅与RF内容及A1/A2相关

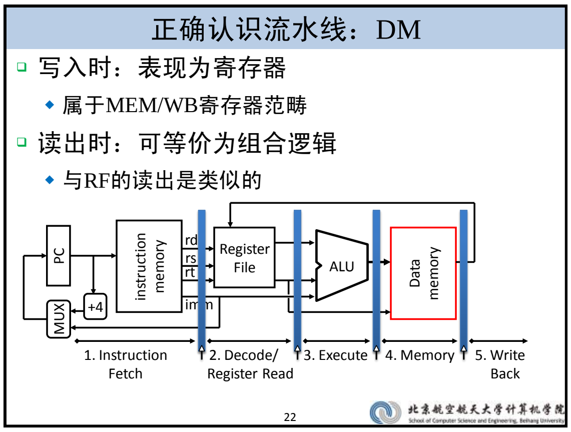
WD
WE
A3
R1
R2
R31
EN
A1
A2
RD1
RD2
写入
读出



正确认识流水线：流水阶段与流水线寄存器

- 流水阶段：组合逻辑+寄存器
  - 起始：前级流水线寄存器的输出
  - 中间：组合逻辑（如ALU）
  - 结束：写入后级流水线寄存器
  - 当时钟上升沿到来时，组合逻辑计算结果存入后级寄存器
- 示例：EX阶段
  - 起始：ID/EX流水线寄存器中的RF寄存器/扩展单元的输出
  - 中间(组合逻辑)：ALU完成计算
  - 结束(寄存器)：在clock上升沿到来时，结果写入EX/MEM中相应寄存器

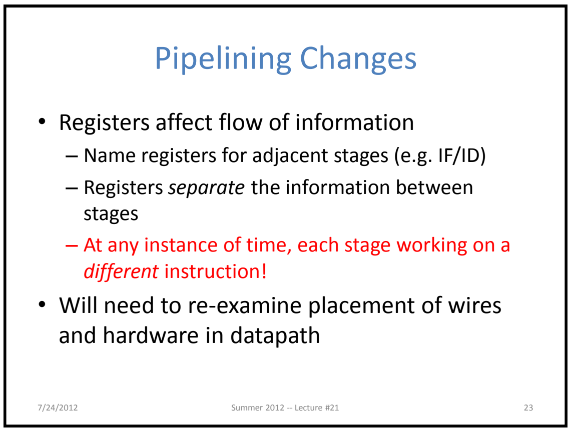
EX
ID/EX
EX/MEM
RF
ALU
ALU 结果



正确认识流水线：DM

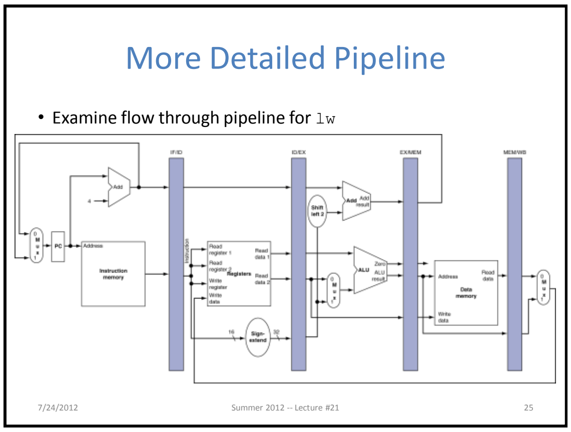
- 写入时：表现为寄存器
  - 属于MEM/WB寄存器范畴
- 读出时：可等价为组合逻辑
  - 与RF的读出是类似的

PC
MUX
+4
instruction memory
rd
rs
rt
imm
Register File
ALU
Data memory
1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Write Back



Pipelining Changes

- Registers affect flow of information
  - Name registers for adjacent stages (e.g. IF/ID)
  - Registers *separate* the information between stages
  - At any instance of time, each stage working on a *different* instruction!
- Will need to re-examine placement of wires and hardware in datapath



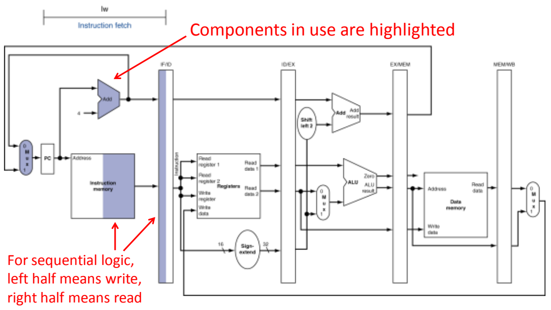
More Detailed Pipeline

- Examine flow through pipeline for `lw`

IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Instruction memory
Registers
Sign-extend
Shift left 2
ALU
Data memory

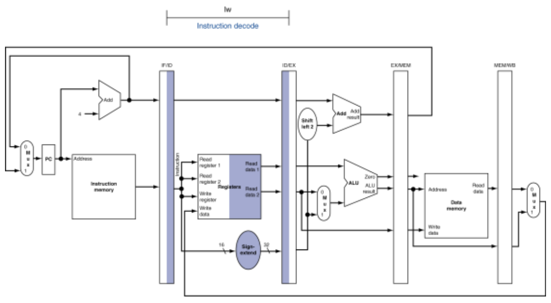
Instruction Fetch (IF) for Load

Components in use are highlighted

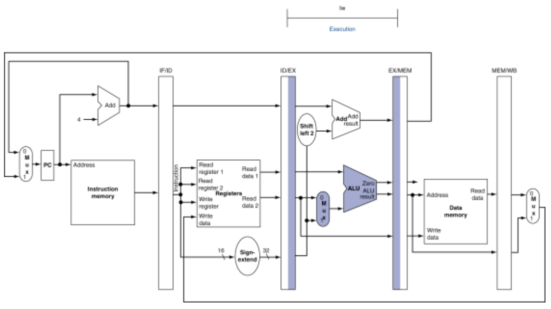


For sequential logic, left half means write, right half means read

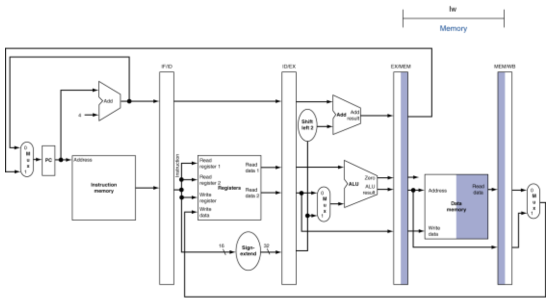
Instruction Decode (ID) for Load



Execute (EX) for Load

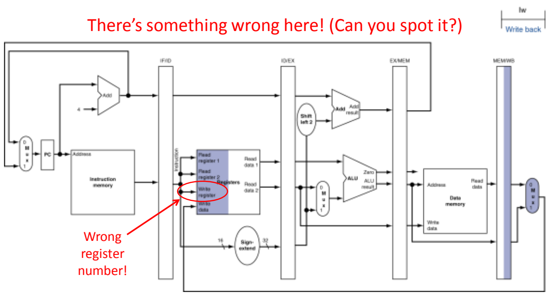


Memory (MEM) for Load



Write Back (WB) for Load

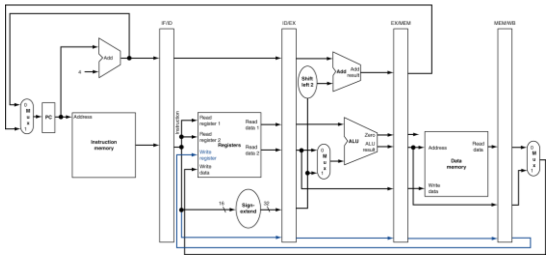
There's something wrong here! (Can you spot it?)



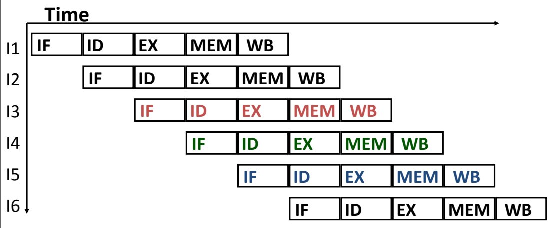
Wrong register number!

Corrected Datapath

- Now any instruction that writes to a register will work properly



Pipelined Execution Representation



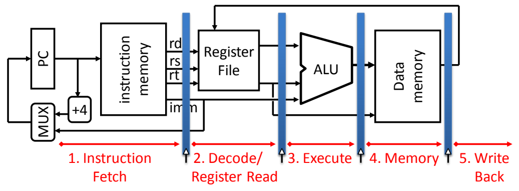
- Every instruction must take same number of steps, so some will idle
  - e.g. MEM stage for any arithmetic instruction

时钟驱动的流水线时空图

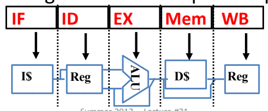
- 本图用途：需精确分析指令/时间/流水线3者关系时
  - 行：某个时钟，指令流分别处于哪些阶段
  - 列：某个部件，在时间方向上的执行了哪些指令
- 注意区分流水阶段与流水线寄存器的关系
- 可以看出，在CLK5后，流水线全部充满
  - 所有部件都在执行指令
    - 只是不同的指令

				IF级	ID/RF级	EX级	MEM级	WB级	
相对PC的地址偏移	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	Instr 1	↑ 1	0→4	Instr 1	Instr 1				
4	Instr 2	↑ 2	4→8	Instr 2	Instr 2	Instr 1			
8	Instr 3	↑ 3	8→12	Instr 3	Instr 3	Instr 2	Instr 1		
12	Instr 4	↑ 4	12→16	Instr 4	Instr 4	Instr 3	Instr 2	Instr1	
16	Instr 5	↑ 5	16→20	Instr 5	Instr 5	Instr 4	Instr 3	Instr 2	Instr1
20	Instr 6	↑ 6	20→24	Instr6	Instr6	Instr5	Instr4	Instr3	Instr2

Graphical Pipeline Diagrams

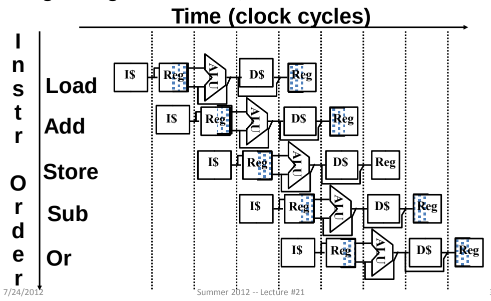


- Use datapath figure below to represent pipeline:



Graphical Pipeline Representation

- RegFile: right half is read, left half is write



第十六讲

Instruction Level Parallelism (ILP)

- Pipelining allows us to execute parts of multiple instructions at the same time using the same hardware!
  - This is known as *instruction level parallelism*

Pipeline Performance (1/2)

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

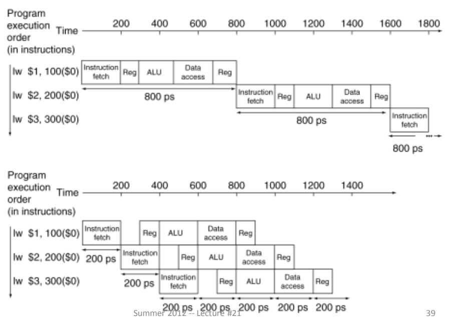
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

Pipeline Performance (2/2)

Single-cycle
T_c = 800 ps

Pipelined
T_c = 200 ps



Pipeline Speedup

- Use T_c ("time between completion of instructions") to measure speedup
  - $T_{c,\text{pipelined}} \geq \dfrac{T_{c,\text{single-cycle}}}{\text{Number of stages}}$
  - Equality only achieved if stages are *balanced* (i.e. take the same amount of time)
- If not balanced, speedup is reduced
- Speedup due to increased throughput
  - Latency for each instruction does not decrease

Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining!
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step
- Memory operands only in Loads and Stores
  - Can calculate address 3rd stage, access memory 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle

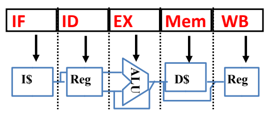
Question: Which of the following signals (buses or control signals) for MIPS-lite does NOT need to be passed into the EX pipeline stage?

☐ PC + 4

☐ MemWr

☐ RegWr

[✓ imm16]



Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) Structural hazard
- A required resource is busy (e.g. needed in multiple stages)

2) Data hazard
- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

3) Control hazard
- Flow of execution depends on previous instruction

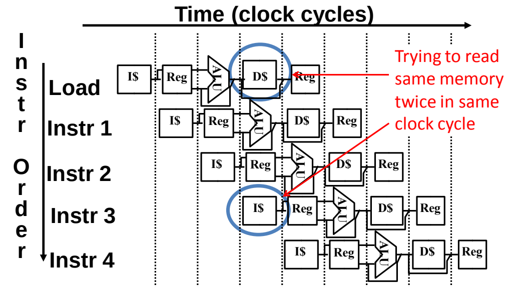
Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - ~~Branch Prediction~~

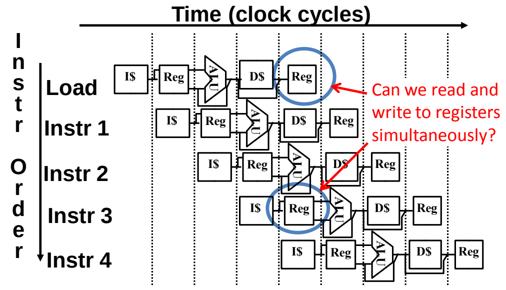
1. Structural Hazards

- Conflict for use of a resource
- MIPS pipeline with a single memory?
  - Load/Store requires memory access for data
  - Instruction fetch would have to *stall* for that cycle
    - Causes a pipeline "*bubble*"
- Hence, pipelined datapaths require separate instruction/data memories
  - Separate L1 I$ and L1 D$ take care of this

Structural Hazard #1: Single Memory



Structural Hazard #2: Registers (1/2)



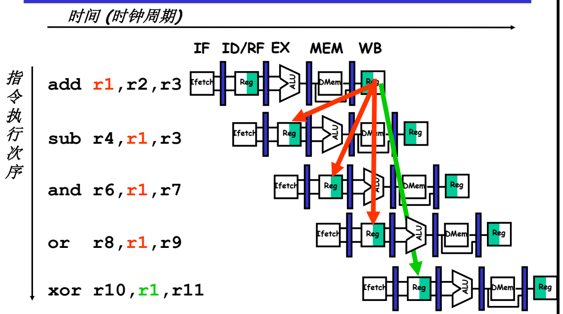
Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) Split RegFile access in two: Write during 1st half and Read during 2nd half of each clock cycle
     - Possible because RegFile access is *VERY* fast (takes less than half the time of ALU stage)
  2) Build RegFile with independent read and write ports
- **Conclusion:** Read and Write to registers during same clock cycle is okay

Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - Branch Prediction

数据冒险



add r1,r2,r3

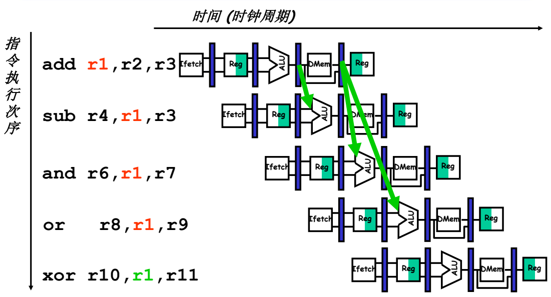
sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9

xor r10,r1,r11

数据冒险解决策略－旁路



add r1,r2,r3

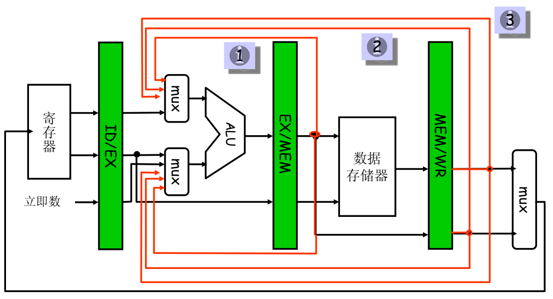
sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9

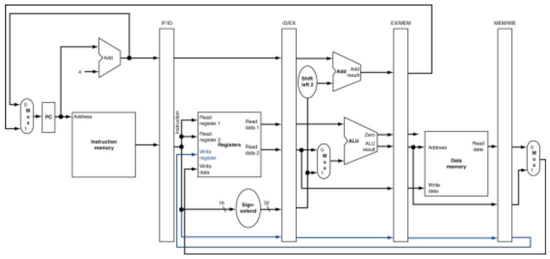
xor r10,r1,r11

调整硬件结构支持旁路



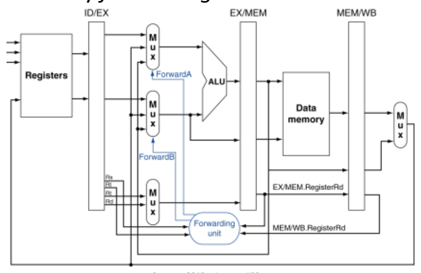
Datapath for Forwarding (1/2)

- What changes need to be made here?



Datapath for Forwarding (2/2)

- Handled by *forwarding unit*



Agenda

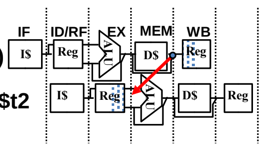
- Structural Hazards
- Data Hazards
  - Forwarding
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - Branch Prediction

Data Hazard: Loads (1/4)

- **Recall:** Dataflow backwards in time are hazards

lw $t0,0($t1)

sub $t3,$t0,$t2



- Can’t solve all cases with forwarding
  - Must *stall* instruction dependent on load, then forward (more hardware)

Data Hazard: Loads (2/4)

- *Hardware* stalls pipeline
  - Called “hardware interlock”

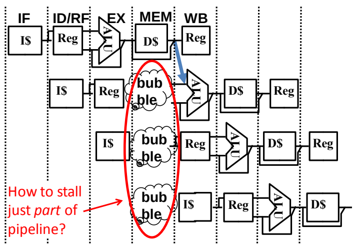
Schematically, this is what we want, but in reality stalls done “horizontally”

lw $t0, 0($t1)

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



How to stall just *part* of pipeline?

Data Hazard: Loads (3/4)

- Stall is equivalent to `nop`

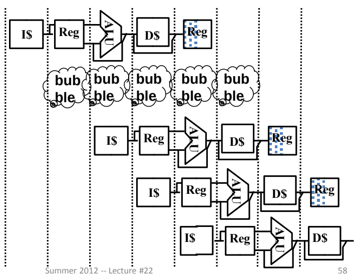
lw $t0, 0($t1)

nop

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



load导致的数据冒险：Clk1上升沿后

- 指令流
  - lw进入IF/ID
  - PC：指向sub指令的地址
    - PC ← PC + 4
  - IM：输出sub指令

后续不再分析PC和IM

					IF级	ID级	EX级	MEM级	WB级
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	`lw $t0, 0($t1)`	↥ 1	0→4	lw→sub	lw				
4	`sub $t3, $t0, $t2`								
8	`and $t5, $t0, $t4`								
12	`or $t7, $t0, $t6`								
16	`add $t1, $t2, $t3`								

load导致的数据冒险：Clk2上升沿后

- 指令流
  - sub进入IF/ID寄存器；lw进入ID/EX寄存器
- 冲突分析：冲突出现
- 执行动作：设置控制信号，在clk3插入nop指令
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4

					IF级	ID级	EX级	MEM级	WB级
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	`lw $t0, 0($t1)`	↥ 1	0→4	lw→sub	**lw**				
4	`sub $t3, $t0, $t2`	↥ 2	4→8	sub→and	**sub**	**lw**			
8	`and $t5, $t0, $t4`								
12	`or $t7, $t0, $t6`								
16	`add $t1, $t2, $t3`								

load导致的数据冒险：Clk3上升沿后

- 指令流
  - sub进入ID/EX；lw进入EX/MEM
  - ID/EX向ALU提供数据
- 冲突分析：冲突解除
  - 转发机制将在clk4时可以发挥作用

					IF级	ID级	EX级	MEM级	WB级
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	`lw $t0, 0($t1)`	↥ 1	0→4	lw→sub	**lw**				
4	`sub $t3, $t0, $t2`	↥ 2	4→8	sub→and	**sub**	**lw**			
8	`and $t5, $t0, $t4`	↥ 3	8→8	and	**sub**	**nop**	**lw**		
12	`or $t7, $t0, $t6`								
16	`add $t1, $t2, $t3`								

load导致的数据冒险：Clk4上升沿后

- 指令流
  - lw：结果存入MEM/WB。
  - sub：进入ID/EX。故ALU的操作数可以从MEM/WB转发
- 执行动作
  - 控制MUX，使得MEM/WB输入到ALU

					IF级	ID级	EX级	MEM级	WB级
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	`lw  $t0, 0($t1)`	↥ 1	0→4	lw→sub	**lw**				
4	`sub $t3, $t0, $t2`	↥ 2	4→8	sub→and	**sub**	**lw**			
8	`and $t5, $t0, $t4`	↥ 3	8→8	and	sub	nop	lw		
12	`or  $t7, $t0, $t6`	↥ 4	8→12	and→or	**and**	sub	nop	lw结果	
16	`add $t1, $t2, $t3`								

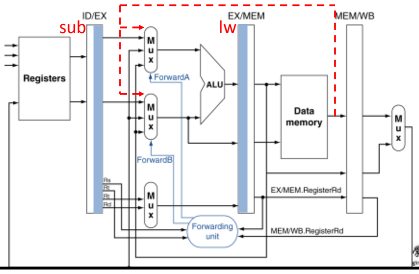
load导致的数据冒险：Clk5上升沿后

- 指令流
  - lw：结果回写至RF
  - sub：结果保存在EX/MEM

					IF级	ID级	EX级	MEM级	WB级
地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	`lw  $t0, 0($t1)`	↥ 1	0→4	lw→sub	**lw**				
4	`sub $t3, $t0, $t2`	↥ 2	4→8	sub→and	**sub**	**lw**			
8	`and $t5, $t0, $t4`	↥ 3	8→8	and	sub	nop	lw		
12	`or  $t7, $t0, $t6`	↥ 4	8→12	and→or	**and**	sub	nop	lw结果	
16	`add $t1, $t2, $t3`	↥ 5	12→16	or→add	**or**	**and**	sub结果	nop	lw结果

load导致的数据冒险

- Q：如果设置从DM到ALU输入的转发，这个设计优劣如何？
  - 设计初衷：将DM读出数据提前1个clock转发至ALU，从而消除lw指令导致的数据相关，无需插入NOP

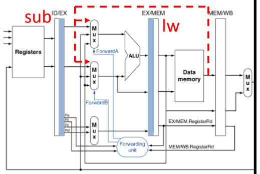


load导致的数据冒险

- A：功能虽然正确，但CPU时钟频率大幅度降低
  - 原设计：$f = 5$GHz
    - 各阶段最大延迟为200ps
  - 新设计：$f = 2.5$GHz
    - EX阶段$_{修改后}$ = ALU延迟 + DM延迟 = 400ps
    - EX阶段延迟成为最大延迟

警惕：木桶原理！

流水线各阶段延迟不均衡，将导致流水线性能严重下降



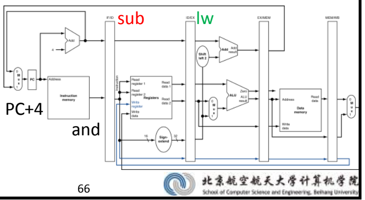
前面PPT的数据

Instr fetch	Register read	ALU op	Memory access	Register write
200ps	100 ps	200ps	200ps	100 ps

如何插入NOP指令？

- 检测条件：IF/ID的前序是lw指令，并且lw的rt寄存器与IF/ID的rs或rt相同
- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4

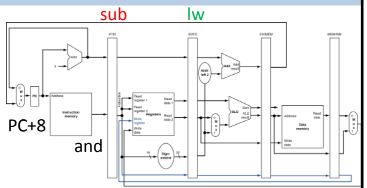
地址	指令
0	`lw  $t0, 0($t1)`
4	`sub $t3, $t0, $t2`
8	`and $t5, $t0, $t4`
12	`or  $t7, $t0, $t6`
16	`add $t1, $t2, $t3`



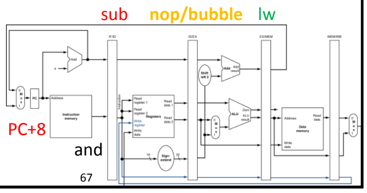
如何插入NOP指令？

Cycle N

地址	指令
0	`lw  $t0, 0($t1)`
4	`sub $t3, $t0, $t2`
8	`and $t5, $t0, $t4`
12	`or  $t7, $t0, $t6`
16	`add $t1, $t2, $t3`



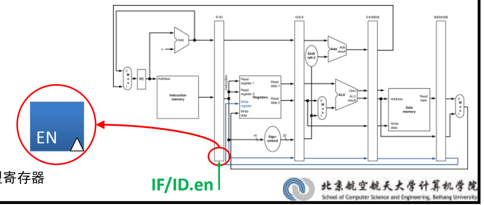
Cycle N+1



如何插入NOP指令？

使能型寄存器

- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4
- 数据通路：将IF/ID修改为使能型寄存器
- 控制系统：增加IF/ID.en控制信号
  - 当IF/ID.en为0时，IF/ID在下个clock上升沿到来时保持不变

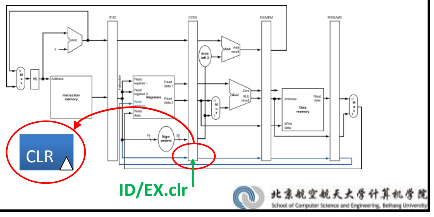


使能型寄存器

如何插入NOP指令？

复位型寄存器

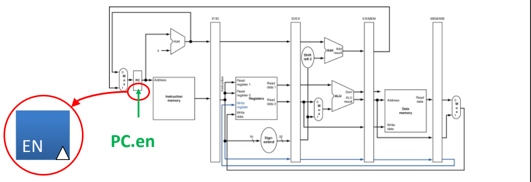
- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4
- 数据通路：将ID/EX修改为复位型寄存器
- 控制系统：增加ID/EX.clr控制信号
  - 当ID/EX.clr为0时，ID/EX在下个clock上升沿到来时被清除为0



如何插入NOP指令？

使能型寄存器

- 执行动作：
  - ①冻结IF/ID：sub继续被保存
  - ②清除ID/EX：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4
- 数据通路：将PC修改为使能型寄存器
- 控制系统：增加PC.en控制信号
  - 当PC.en为0时，PC在下个clock上升沿到来时保持不变



如何插入NOP指令？

- lw冒险处理示例伪代码
- 注意：时序关系
  - 各信号在clk2上升沿后有效
  - NOP是在clk3上升沿后发生，即寄存器值在clk3上升沿到来时发生变化(或保持不变)

```
if (ID/EX.MemRead) &
   ((ID/EX.rt == IF/ID.rs) |
    (ID/EX.rt == IF/ID.rt)
    IF/ID.en ← 禁止
    ID/EX.clr ← 清除
    PC.en ← 禁止
```

地址	指令			IF级	ID级	EX级	MEM级	WB级	
		CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	lw $t0, 0($t1)	↥ 1	0→4	lw→sub	**lw**				
4	sub $t3, $t0, $t2	↥ 2	4→8	sub→and	**sub**	**lw**			
8	and $t5, $t0, $t4	↥ 3	8→8	and	**sub**	**nop**	**lw**		
12	or $t7, $t0, $t6								
16	add $t1, $t2, $t3								

如果没有转发电路呢？

- 由于有转发电路，因此lw指令只插入1个NOP指令
- Q：如果没有转发，需要怎么处理呢？
- A：EX/MEM，MEM/WB也均需要做冲突分析及NOP处理
  - EX/MEM，MEM/WB也需要修改，并增加相应控制信号

地址	指令			IF级	ID级	EX级	MEM级	WB级	
		CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
0	lw $t0, 0($t1)	↥ 1	0→4	lw→sub	**lw**				
4	sub $t3, $t0, $t2	↥ 2	4→8	sub→and	**sub**	**lw**			
8	and $t5, $t0, $t4	↥ 3	8	and	**sub**	**nop**	**lw**		
12	or $t7, $t0, $t6	↥ 4	8	and	**sub**	**nop**	**nop**	**lw结果**	
16	add $t1, $t2, $t3	↥ 5	8	and	**sub**	**nop**	**nop**	**nop**	**lw结果**
		↥ 6	8→12	and→or	**and**	**sub**	**nop**	**nop**	**nop**

Data Hazard: Loads (4/4)

- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle
  - Letting the hardware stall the instruction in the delay slot is equivalent to putting a `nop` in the slot (except the latter uses more code space)
- **Idea:** Let the compiler put an unrelated instruction in that slot → no stall!

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction!
- MIPS code for `A=B+E; C=B+F;`

Stall!

Stall!

```
# Method 1:
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

13 cycles

```
# Method 2:
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

11 cycles

Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - ~~Branch Prediction~~

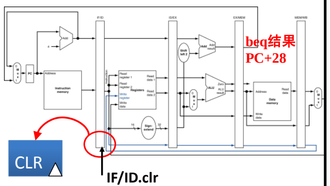
3. Control Hazards

- Branch (`beq`, `bne`) determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- **Simple Solution:** Stall on *every* branch until we have the new PC value
  - How long must we stall?

B指令冒险造成的停顿代价

地址	指令	CLK	PC	IM	IF/ID	ID/EX	EX/MEM	MEM/WB	RF
					IF级	ID级	EX级	MEM级	WB级
0	**beq $1, $3, 24**	↥ 1	0→4	beq→and	beq				
4	and $12, $2, $5	↥ 2	4	and	nop	beq			
8	or $13, $6, $2	↥ 3	4	and	nop	nop	beq结果		
12	add $14, $2, $2	↥ 4	4→28	and→lw	nop	nop	nop		
		↥ 5	28→32	lw→XX	lw	nop	nop	nop	nop
28	**lw $4, 50($7)**								

- 如不对B指令做任何处理，则必须插入3个NOP
  - b指令结果及新PC值保存在EX/MEM，因此PC在clk4才能加载正确值
  - IF/ID在clk5才能存入转移后指令(即lw指令)

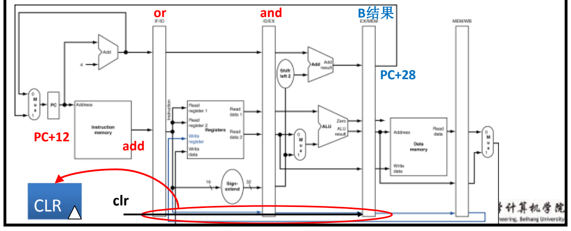


Q：IF/ID.clr表达式？

方案1：假定分支不发生

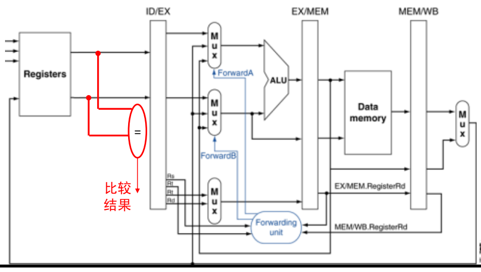
- 即使在ID级发现是B指令也不停顿
- 根据B指令结果，决定是否清除3条后继指令
  - 使得and/or/add不能前进

PC相对偏移	指令
0	beq $1, $3, 24
4	and $12, $2, $5
8	or $13, $6, $2
12	add $14, $2, $2
28	lw $4, 50($7)



方案2：缩短分支延迟

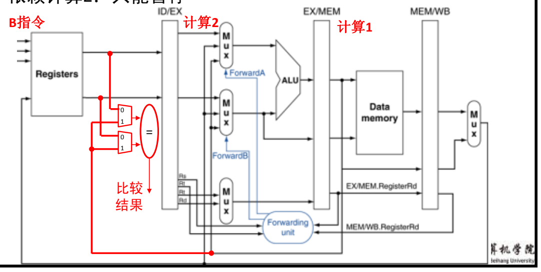
- 在ID阶段放置比较器，尽快得到B指令结果
  - B指令结果可以提前2个clock得到
  - B指令后继可能被废弃的指令减少为1条
    - 当需要转移时，清除IF/ID即可



方案2：缩短分支延迟

- 比较器前置后，会产生数据相关
  - B指令可能依赖于前条指令的结果
- 依赖计算1：从ALU转发数据
- 依赖计算2：只能暂停

Q：如果依赖MEM/WB的结果，是否需要设置转发？
提示：MEM/WB已经有回写通道了，但RF设计满足吗？



3. Control Hazard: Branching

- **Option #3:** *Branch delay slot*
  - Whether or not we take the branch, *always* execute the instruction immediately following the branch
  - Worst-Case: Put a `nop` in the branch-delay slot
  - Better Case: Move an instruction from before the branch into the branch-delay slot
    - Must not affect the logic of program

3. Control Hazard: Branching

- MIPS uses this *delayed branch* concept
  - Re-ordering instructions is a common way to speed up programs
  - Compiler finds an instruction to put in the branch delay slot ≈ 50% of the time
- Jumps also have a delay slot
  - Why is one needed?

Delayed Branch Example

Nondelayed Branch	**Delayed Branch**
`or  $8, $9, $10`	`add $1, $2,$3`
`add $1, $2, $3`	`sub $4, $5, $6`
`sub $4, $5, $6`	`beq $1, $4, Exit`
`beq $1, $4, Exit`	`or  $8, $9, $10`
`xor $10, $1, $11`	`xor $10, $1, $11`
`Exit:`	`Exit:`

Why not any of the other instructions?

Delayed Jump in MIPS

- MIPS Green Sheet for `jal`:
  `R[31]=PC+8; PC=JumpAddr`
  - `PC+8` because of *jump delay slot*!
  - Instruction at `PC+4` always gets executed before `jal` jumps to label, so return to `PC+8`

Question: For each code sequences below, choose one of the statements below:

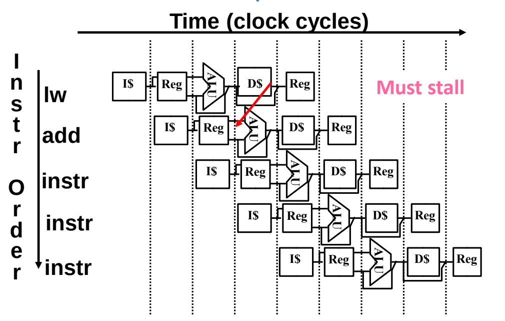
1:
```
lw  $t0,0($t0)
add $t1,$t0,$t0
```

2:
```
add  $t1,$t0,$t0
addi $t2,$t0,5
addi $t4,$t1,5
```

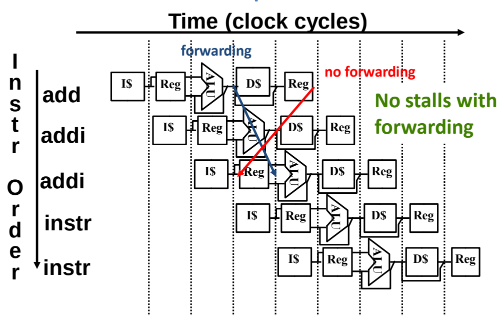
3:
```
addi $t1,$t0,1
addi $t2,$t0,2
addi $t3,$t0,2
addi $t3,$t0,4
addi $t5,$t1,5
```

- ☐ No stalls as is
- ☐ No stalls with forwarding
- ☐ Must stall

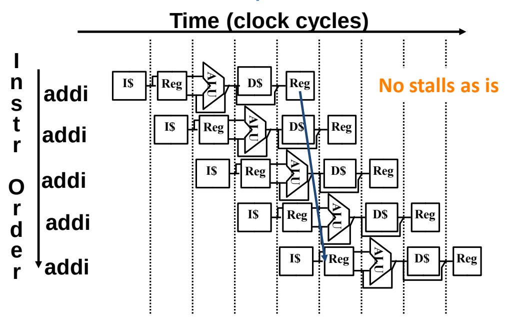
Code Sequence 1



Code Sequence 2



Code Sequence 3



Summary

- Hazards reduce effectiveness of pipelining
  - Cause stalls/bubbles
- Structural Hazards
  - Conflict in use of datapath component
- Data Hazards
  - Need to wait for result of a previous instruction
- Control Hazards
  - Address of next instruction uncertain/unknown
  - Branch and jump delay slots