
Chapter 6:

Enhancing Performance with Pipelining

Overview of Pipelining

- Basic Philosophy
 - Assembly Line Operations (factory)
- Goal
 - Enhance Performance by increasing throughput
- Our Goal:
 - Improve performance by increasing the instruction throughput per clock cycle

Single-Cycle Performance

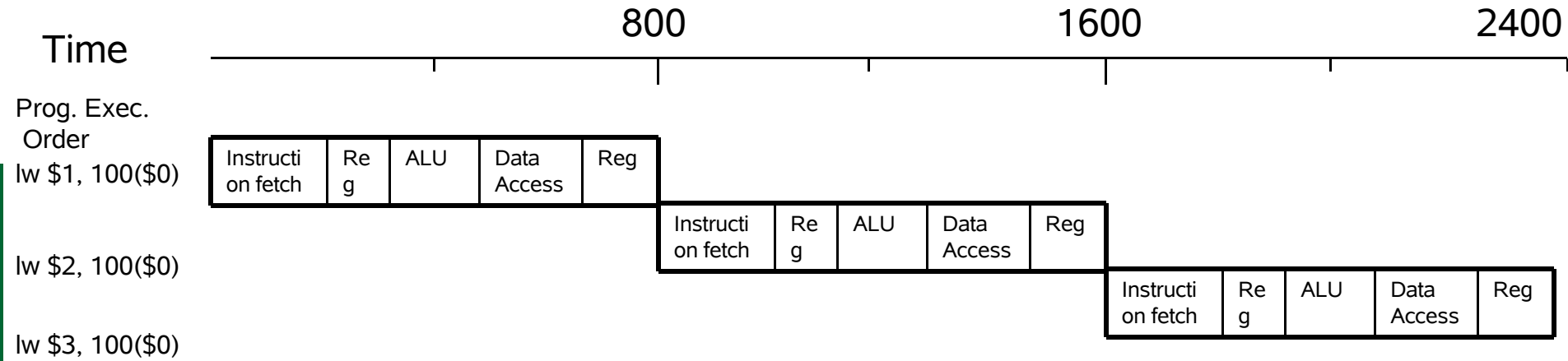
Recall (for a single- cycle clock implementation):

Instruction Class	Major Functional Units used by Instruction Operation Times					Period
	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write	
R-type	200	100	200	0	100	600 ps
Load word	200	100	200	200	100	800 ps
Store word	200	100	200	200		700 ps
Branch	200	100	200	0		500 ps

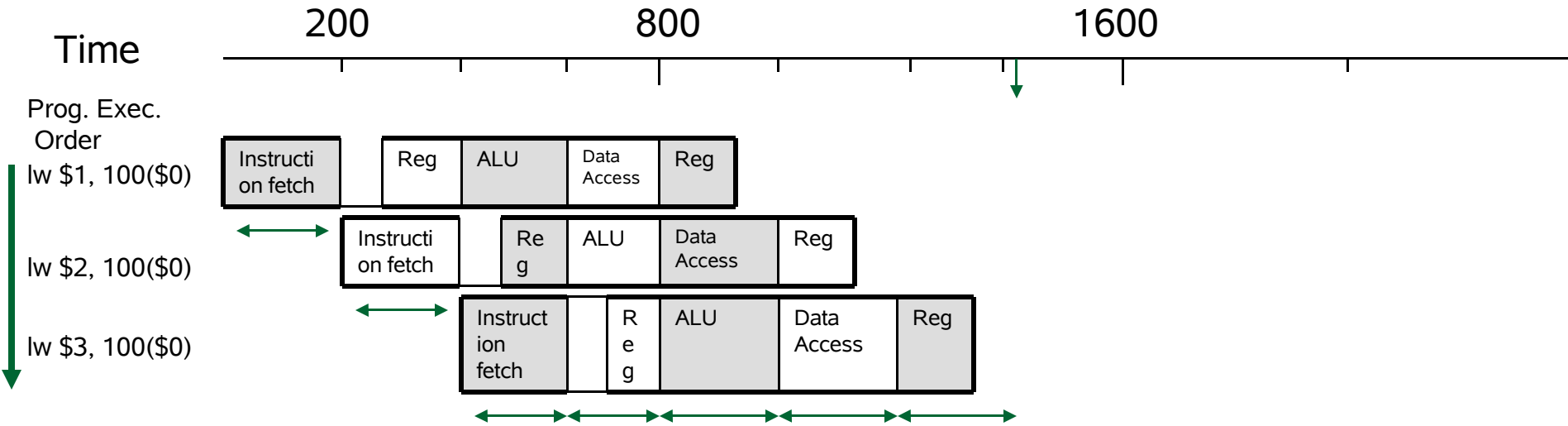
Assume Critical Machine Operation Times: Memory Unit ~ 200 ps, ALU and adders ~ 200ps, Register file (read/write) ~ 100 ps. Estimate **clock cycle** for the machine

Single Clock cycle determined by longest instruction period = **800 ps**

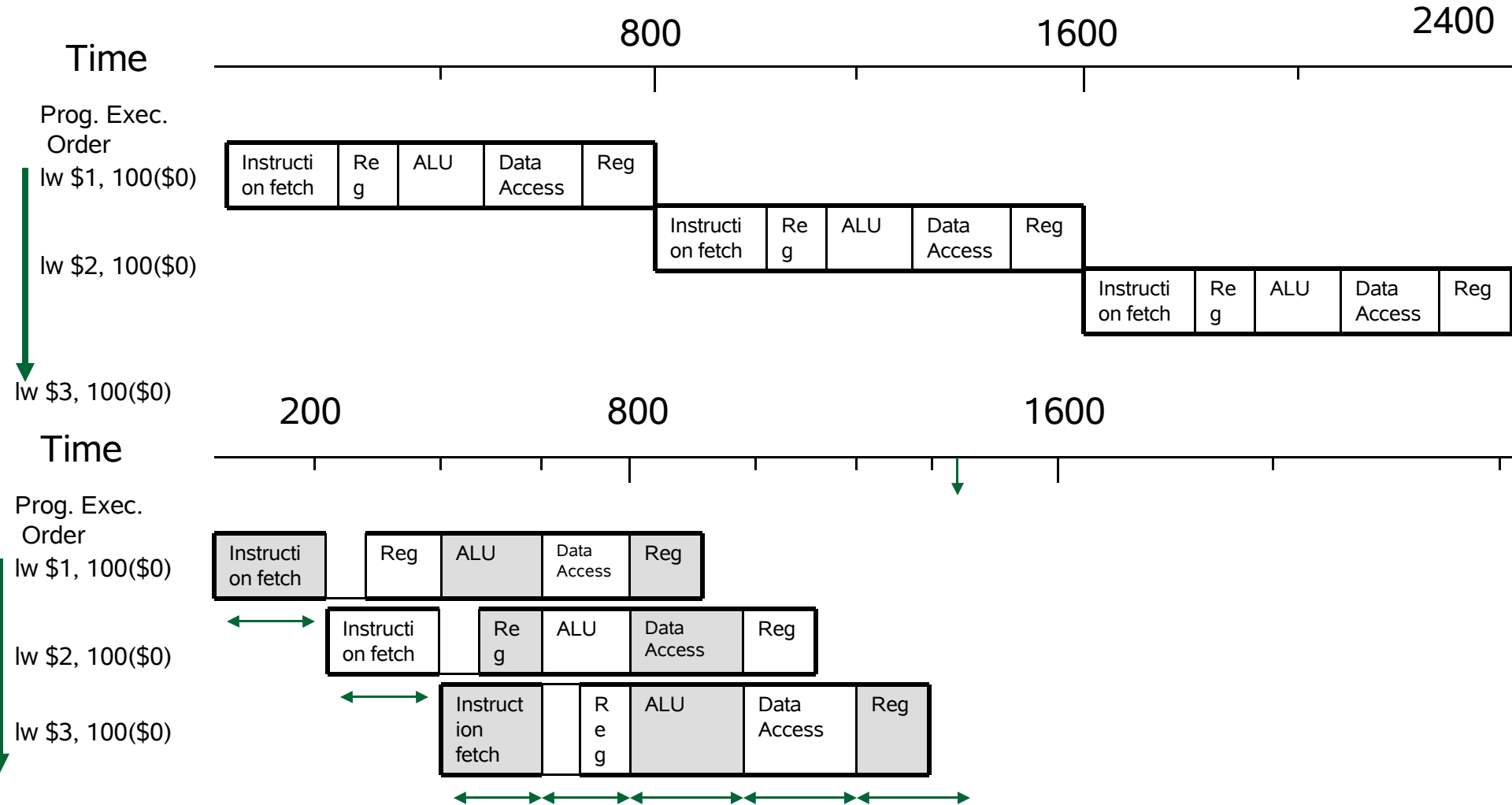
Single Cycle Execution



Pipelined Execution



Single-cycle vs. pipelined execution



Pipelining

- What makes it easy?
 - all MIPS instructions are the same length (→ fetch)
 - just a few instruction formats (→ rs, rt fields are invariant)
 - memory operands appear only in loads and stores
- What makes it hard?
 - structural hazards: suppose we had only one memory
 - control hazards: need to worry about branch instructions
 - data hazards: an instruction depends on a previous instruction
- We'll talk about data hazard and forwarding, stalls and branch hazards
- We'll talk about exception handling

Pipelining

- Increases the number of simultaneous executing instruction
- Increases the rate at which instructions are executed
- Improves instruction throughput

A Pipelined Datapath

Five-stage Pipeline

- Five-stage Pipeline
 - Up to five instructions can be executed in a clock cycle
- Single-cycle datapath can be divided into five stages (refer to Fig 6.9):
 1. IF: Instruction Fetch
 2. ID: Instruction Decode and register file read
 3. EX: Execution and Address Calculation
 4. MEM: Data memory Access
 5. WB: Write Back

How does information flow in typical auto assembly plant?

A Pipelined Datapath

Five-stage Pipeline

■ Information Flow:

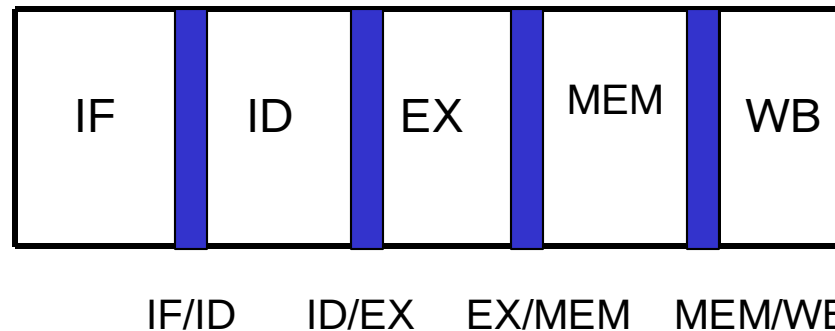
- In general from Left to Right (1->2->3->4->5)
- The WB stage of step 5 writes data into register file of the ID stage in step 2
 - 5 -> 2
- The MEM stage of step 4 controls the multiplexor in the IF stage of step1
 - 4 -> 1

Refer to Figure 6.9 for schematic illustrations

A Pipelined Datapath

Five-stage Pipeline

- Five stages are interconnected by 4 Pipeline Registers (latches)
 - Registers must be wide enough store information



A Pipelined Datapath

Five-stage Pipeline

■ Example:

- ❑ lw \$s1, 100 (\$s0)
- ❑ lw \$s2, 200 (\$s0)
- ❑ lw \$s3, 300 (\$s0)

Inst.	Time (in clock cycles) →-----						
	CC1	CC2	CC3	CC4	CC5	CC6	CC7
lw \$s1, 100 (\$s0)	IF	ID	EX	MEM	WB		
lw \$s2, 200 (\$s0)		IF	ID	EX	MEM	WB	
lw \$s3, 300 (\$s0)			IF	ID	EX	MEM	WB

A Pipelined Datapath

Five-stage Pipeline

1. **IF** stage: Fetches the first, second and third *lw* instructions in cycles CC1, CC2 and CC3 resp.
2. **ID** stage: Reads the *rs* register (\$s0) for the first, second and third instructions in cycles CC2, CC3 and CC4 respectively
4. **EX** stage: Calculates the memory address for the first, second and third instructions during clock cycles CC3, CC4, and CC5 respectively
6. **MEM** stage: Fetches memory words at addresses 100, 200, and 300 during clock cycles CC4, CC5, and CC6, respectively
8. **WB** stage: Copies the memory words into registers \$s1, \$s2, and \$s3 during clock cycles CC5, CC6, and CC7, respectively

Inst.	Time (in clock cycles) →-----						
	CC1	CC2	CC3	CC4	CC5	CC6	CC7
<i>lw</i> \$s1, 100 (\$s0)	IF	ID	EX	MEM	WB		
<i>lw</i> \$s2, 200 (\$s0)		IF	ID	EX	MEM	WB	
<i>lw</i> \$s3, 300 (\$s0)			IF	ID	EX	MEM	WB

Each instruction execution takes 5 clock cycles in the pipeline
The 3 *lw* instructions take 7 clock cycles to execute

A Pipelined Datapath

Five-stage Pipeline Registers

IF/ID Latch: Holds fetched instruction and incremented PC

Allows the ID stage to decode instruction 1, while IF stage fetches instruction 2

ID/EX Latch: Stores the sign-extended immediate value and values fetched from register rs and rt

Allows EX stage to utilize stored values, while the ID stage decodes Inst. 2 and the IF stage fetches register for instruction 3

EX/MEM Latch: Stores the branch target address, the ALU result, ALU output bit and value in the rt register

Allows MEM stage to use stored values, while EX and ID stage execute the following instructions

MEM/WB Latch: Stores ALU result and Data read from memory

Allows WB stage to use stored data, while data memory fetches data for the following Instruction

Inst.	Time (in clock cycles) →-----						
	CC1	CC2	CC3	CC4	CC5	CC6	CC7
lw \$s1, 100 (\$s0)	IF	ID	EX	MEM	WB		
lw \$s2, 200 (\$s0)		IF	ID	EX	MEM	WB	
lw \$s3, 300 (\$s0)			IF	ID	EX	MEM	WB

Pipelined Control

- What needs to be controlled in the auto assembly plant environment?
- Use a Distributed control Strategy:
 - ✓ Instruction Fetch and PC Increment (control signal always asserted)
 - ✓ Instruction Decode / Register Fetch
 - Execution Stage:
 - Set *RegDst*, *ALUOp*, and *ALUSrc* (Refer to Fig. 6.23 and 6.24)
 - Memory Stage:
 - Set *MemRead* and *MemWrite* to control data memory (Fig. 6.24)
 - Write Back
 - *MemtoReg* controls the multiplexor and *RegWrite* stores the multiplexor output in the register file
- Pass control signals along just like data (Refer to Fig 6.26)

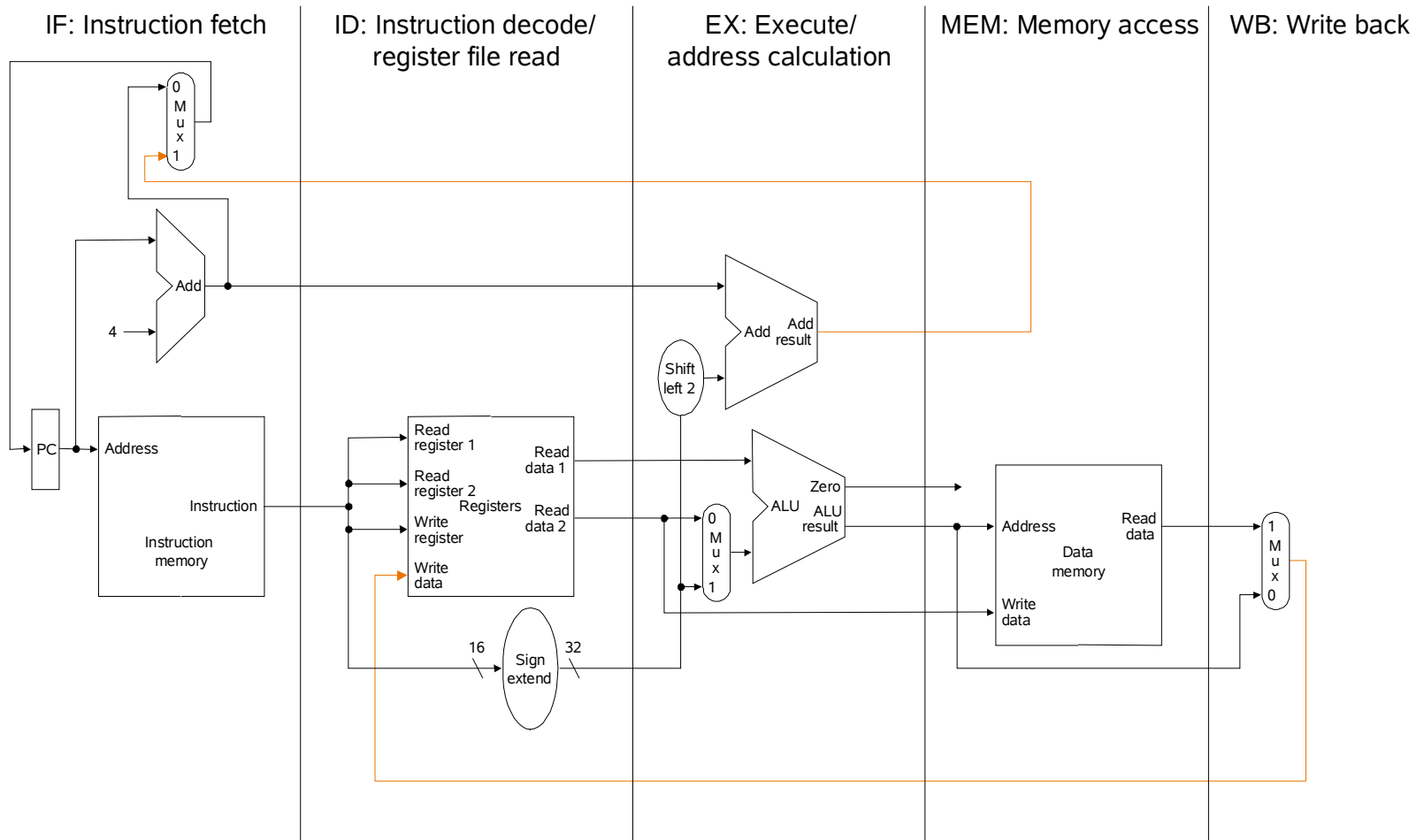
Announcement

- Your bonus questions will be due on Friday (Dec. 16th) afternoon (5:00pm).
- Homework 5 will be due this Friday (Dec. 9th) afternoon (5:00pm). You can expect the homework solutions by next Monday night.
- Extra class: Dec 8th/Thursday from 5:15pm at room 108!
- Final: Dec. 14th/Wednesday at 5:45pm at room 115!

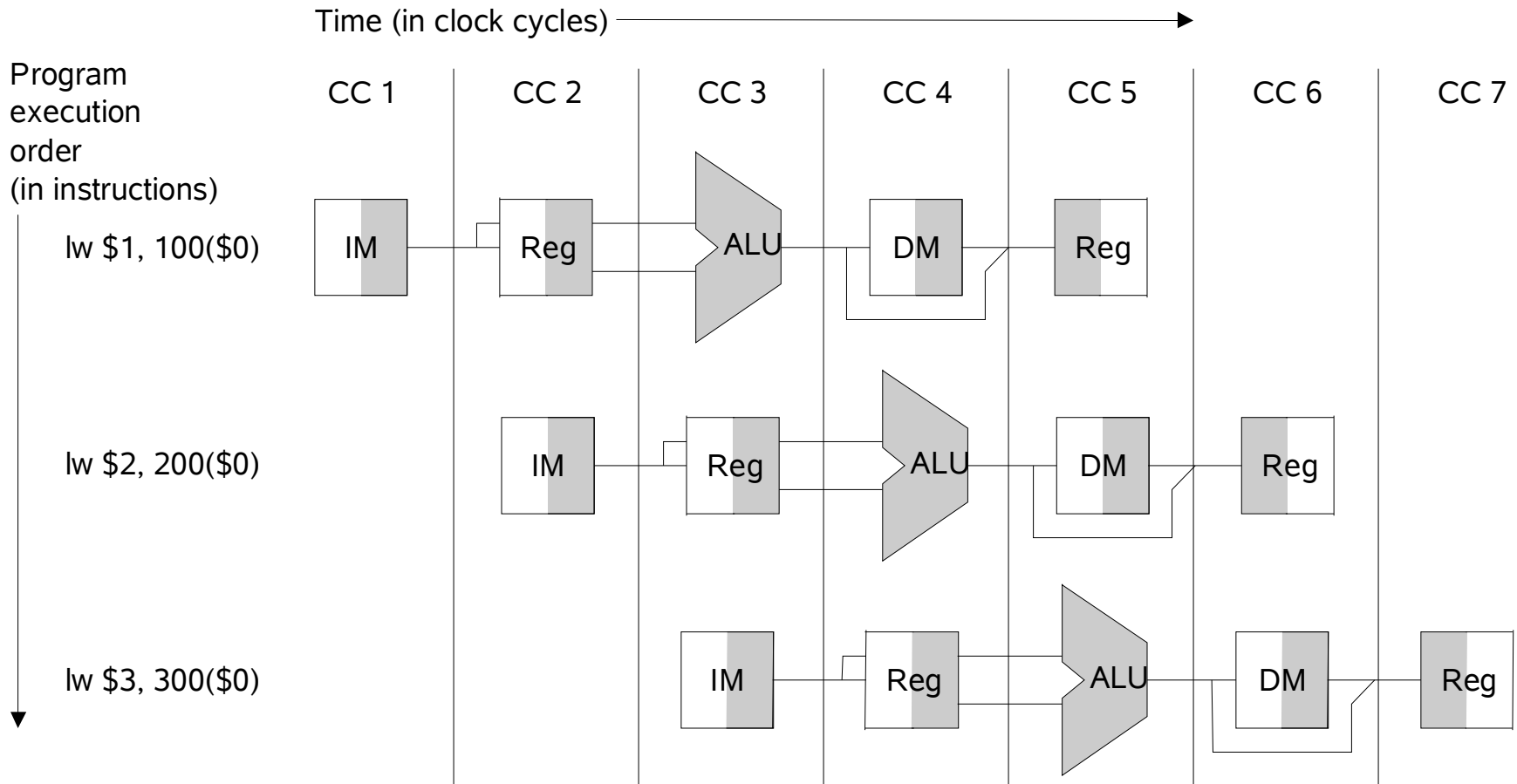
Review (1)

- Instruction execution can be broken down into five stages
 - Instruction fetch (IF)
 - Instruction decode and register fetch (ID)
 - Execute (EX)
 - Memory access (MEM)
 - Write back (WB)
- Every instruction goes through all five stages
- Results are only written to the register file in WB

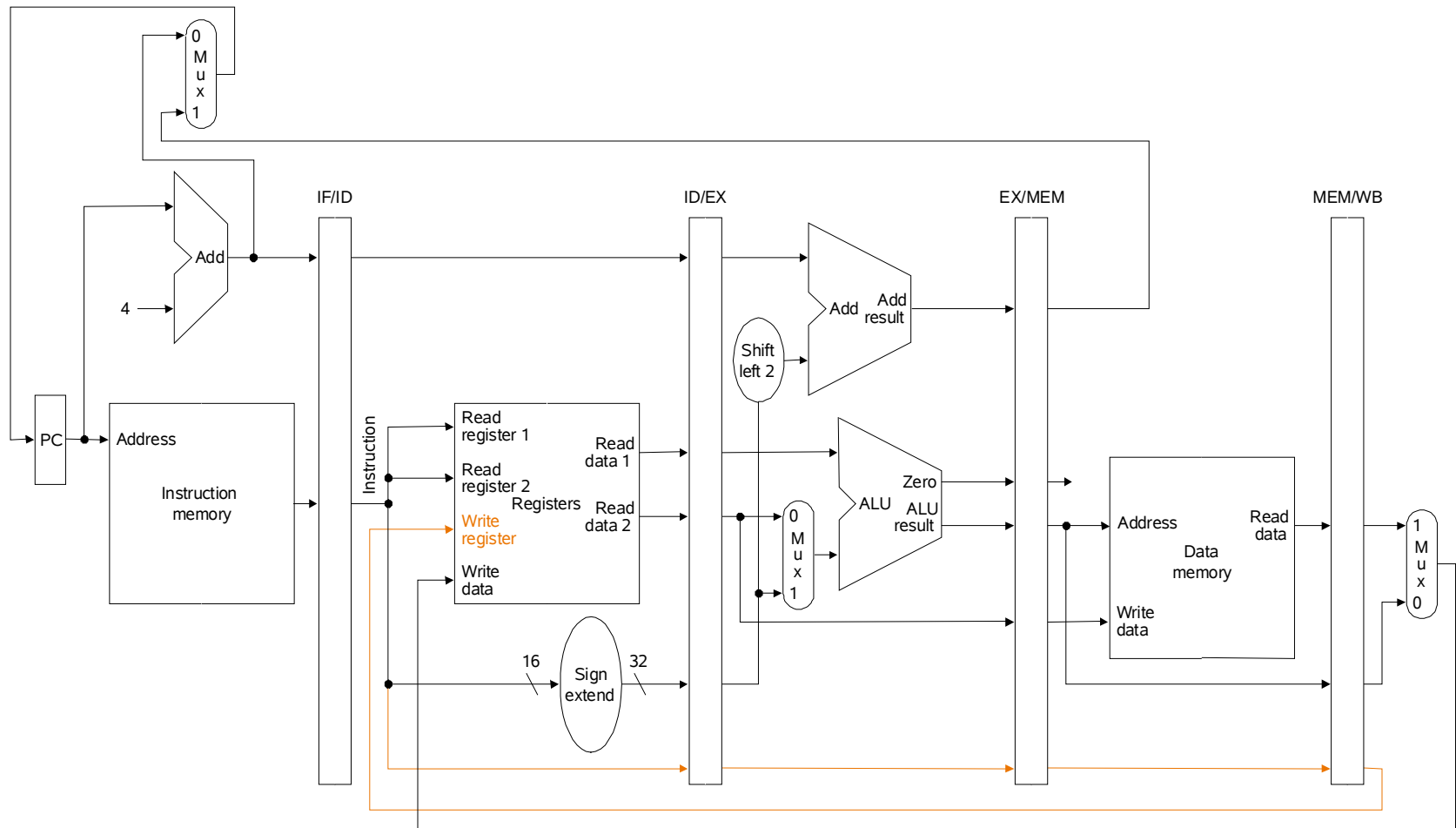
Five-Stage Pipeline



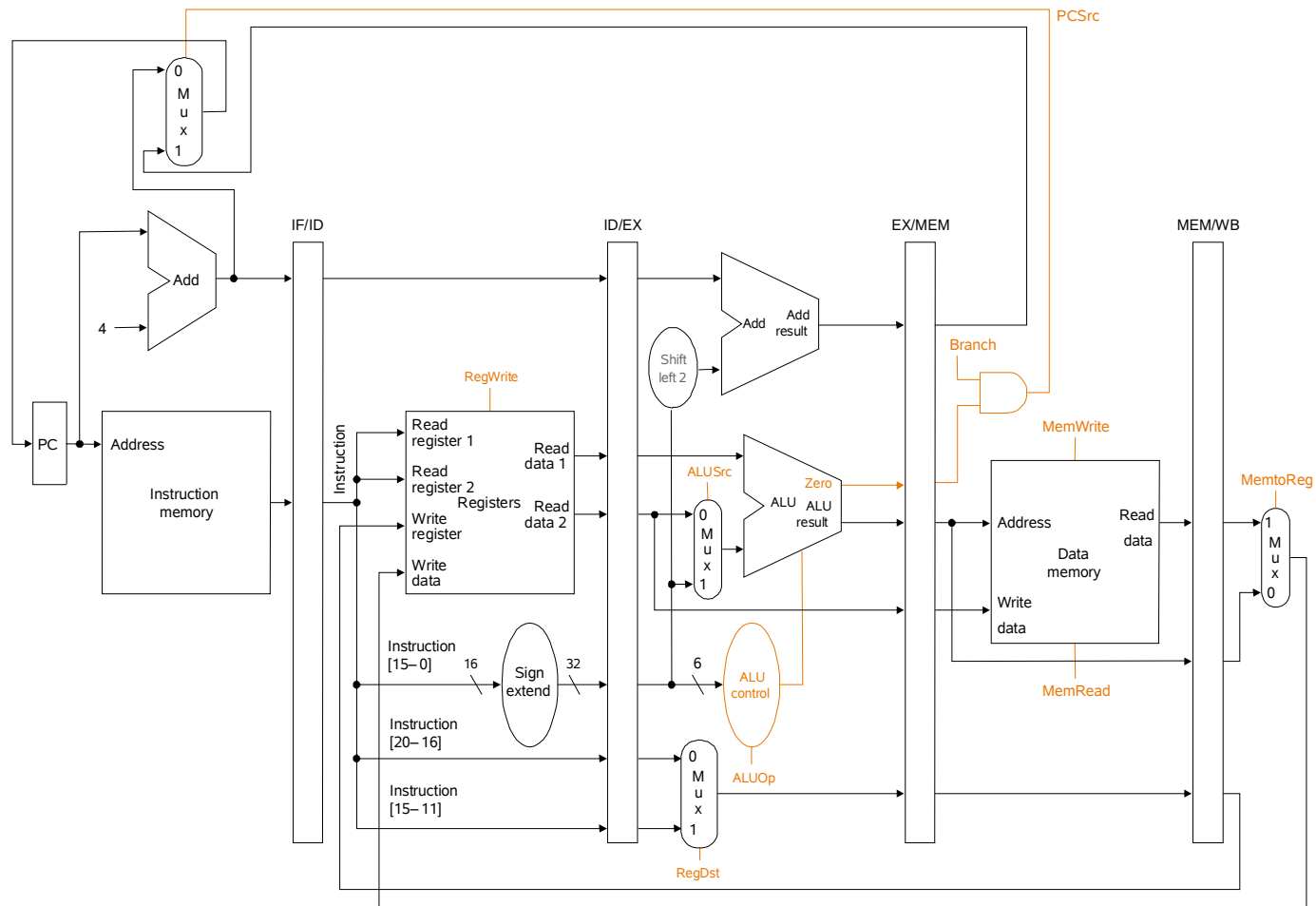
Hardware Usage



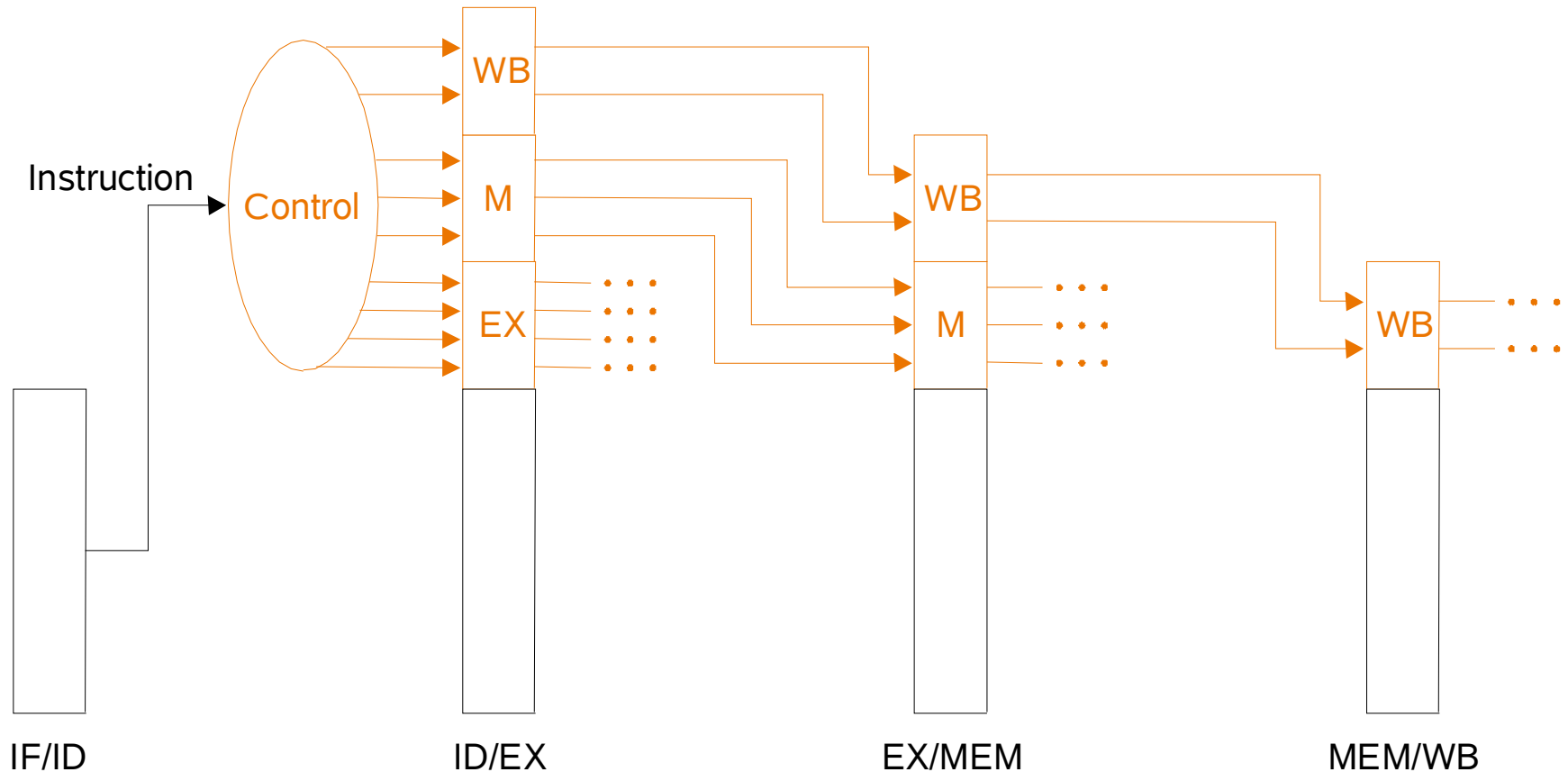
Pipelined Datapath



Datapath and Control Signals



Control Signal Propagation



Data Hazard and Forwarding

Consider the following sequence of Instructions:

- sub \$2,\$1,\$3 # **writes** a new value into \$2
- and \$12,\$2,\$5 # uses new value in \$2
- or \$13,\$6,\$2 # uses new value in \$2
- add \$14,\$2,\$2 # uses new value in \$2
- sw \$15,100(\$2) # uses new value in \$2

Inst.	Time (in clock cycles) →-----								
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
sub \$2,\$1,\$3	IF	ID	EX	MEM	WB				
and \$12,\$2,\$5		IF	ID	EX	MEM	WB			
or \$13,\$6,\$2			IF	ID	EX	MEM	WB		
add \$13,\$6,\$2				IF	ID	EX	MEM	WB	
sw \$15,100(\$2)					IF	ID	EX	MEM	WB

Data Hazard and Forwarding

- When does \$2 receive the new value computed by the *sub* instruction?
- When the WB stage writes \$2 during CC5.
- When does the *and* instruction read \$2?
- When the ID stage fetches \$2 during CC3.
- The *and* instruction will actually read the incorrect *old* value stored in \$2 instead of the *new* value computed by the *sub* instruction.
- The *or* instruction will also read the incorrect *old* value in \$2.

Inst.	Time (in clock cycles) →-----								
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
sub \$2,\$1,\$3	IF	ID	EX	MEM	WB				
and \$12,\$2,\$5		IF	ID	EX	MEM	WB			
or \$13,\$6,\$2			IF	ID	EX	MEM	WB		
add \$13,\$6,\$2				IF	ID	EX	MEM	WB	
sw \$15,100(\$2)					IF	ID	EX	MEM	WB

Data Hazard and Forwarding

- The *add* instruction reads register \$2 during CC5 - does it read the *old* value or the *new* value? It depends on the design of the register file.
- We assume the new value is written into \$2 by the WB stage before the ID stage reads \$2 so the *add* instruction reads the correct value.
- The *sw* instruction will also read the correct *new* value in \$2 during CC6.

Inst.	Time (in clock cycles) →-----								
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
sub \$2,\$1,\$3	IF	ID	EX	MEM	WB				
and \$12,\$2,\$5		IF	ID	EX	MEM	WB			
or \$13,\$6,\$2			IF	ID	EX	MEM	WB		
add \$14,\$2,\$2				IF	ID	EX	MEM	WB	
sw \$15,100(\$2)					IF	ID	EX	MEM	WB

Data Hazard and Forwarding

- In this example, the *and* and *or* instructions have encountered *data hazards*
 - They read a register (or two) too early, i.e., before previous instructions have loaded the register(s) with the correct value(s).

Inst.	Time (in clock cycles) →-----								
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
sub \$2,\$1,\$3	IF	ID	EX	MEM	WB				
and \$12,\$2,\$5		IF	ID	EX	MEM	WB			
or \$13,\$6,\$2			IF	ID	EX	MEM	WB		
add \$14,\$2,\$2				IF	ID	EX	MEM	WB	
sw \$15,100(\$2)					IF	ID	EX	MEM	WB

Data Hazard and Forwarding

- **Solution #1:**
- insert two *nop* instructions between the *sub* and *and* instruction:













Inst.	Time (in clock cycles) →-----										
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
sub \$2,\$1,\$3	IF	ID	EX	MEM	WB						
nop		IF	ID	EX	MEM	WB					
nop			IF	ID	EX	MEM	WB				
and \$12,\$2,\$5				IF	ID	EX	MEM	WB			
or \$13,\$6,\$2					IF	ID	EX	MEM	WB		
add \$14,\$2,\$2						IF	ID	EX	MEM	WB	
sw \$15,100(\$2)							IF	ID	EX	MEM	WB

Inserting the *nop* instructions delays the *and* and *or* instructions two clock cycles
Eliminates the data hazard for the *and* and *or* instructions.
Performance cost of two extra clock cycles.

Data Hazards and Forwarding

Forwarding

- The *sub* instruction result is stored in the EX/MEM pipeline register at the end of CC3
- The *add* instruction could read the \$2 value from the EX/MEM pipeline register and use it during CC4
- The *or* instruction could read the \$2 value from the MEM/WB pipeline register and use it during CC5

Inst.	Time (in clock cycles) →-----								
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
sub \$2,\$1,\$3	IF	ID	EX 	MEM 	WB 				
add \$12,\$2,\$5		IF	ID 	EX 	MEM 	WB			
or \$13,\$6,\$2			IF 	ID 	EX 	MEM	WB		
add \$14,\$2,\$2				IF 	ID 	EX	MEM	WB	
sw \$15,100(\$2)					IF	ID	EX	MEM	WB













Use Temporary Results. Don't wait for results to be written

Data Hazards and Forwarding

Can't always Forward

Consider the following sequence of Instructions:

- lw \$2, 20(\$1) # **load** a new value into \$2
- and \$4,\$2,\$5 # uses new value in \$2
- or \$8,\$2, \$6 # uses new value in \$2
- add \$9,\$4,\$2 # uses new value in \$2
- slt \$1, \$6,\$7

Inst.	Time (in clock cycles) →-----								
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
lw \$2, 20(\$1)	IF	ID	EX 	MEM 	WB				
add \$4,\$2,\$5		IF	ID 	EX 	MEM 	WB			
or \$8,\$2, \$6			IF 	ID 	EX 	MEM	WB		
add \$9,\$4,\$2				IF 	ID 	EX	MEM	WB	
slt \$1, \$6,\$7					IF 	ID	EX	MEM	WB

Data Hazards and Stalls

Hardware solution

- Load Word can cause a hazard
 - Instruction reads a register following a load instruction that writes to the same register
- Need a hazard detection unit to “stall” the load instruction

Inst.	Time (in clock cycles) →-----									
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	
lw \$2, 20(\$1)	IF	ID	EX	MEM	WB					
nop		IF	ID	EX	MEM	WB				
add \$4,\$2,\$5			IF	ID	EX	MEM	WB			
or \$8,\$2, \$6				IF	ID	EX	MEM	WB		
add \$9,\$4,\$2					IF	ID	EX	MEM	WB	
slt \$1, \$6,\$7						IF	ID	EX	MEM	WB

Data Hazards and Stalls

Load Delay

■ Load Delay

- ❑ *load word* is immediately followed by Arithmetic Instruction (e.g., add, subtract) that uses loaded operand
- ❑ *load word* is immediately followed by instruction using the loaded operand to compute memory address

Compiler re-arranges code to eliminate as many *load delays* as possible

Data Hazards and Stalls

Load Delay Solution

- Software Approach
 - Compiler inserts a *nop instruction after the load*
- *Hardware*
 - Pipeline control creates a *stall*

How does hardware detect load delay?

Data Hazards and Stalls

Hardware Load Delay Solution

■ Load Delay Detection

- The MemRead control signal in ID/EX register is set to 1
AND
- RegisterRT index in the ID/EX register == RegisterRS index
AND/OR
- RegisterRT index in the ID/EX register == RegisterRt index in the IF/ID register

■ To stall an instruction (in a pipeline stage)

- Inhibit clock pulse to pipeline register before stage
 - The register keeps old instruction instead of next
 - Instruction in front of pipeline must also be stalled
 - Instructions in rear of pipeline continue to flow to create a gap (bubble)

Data Hazards and Stalls

Installing stall

Inst.	Time (in clock cycles) →-----									
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	
lw \$1, 0(\$4)	IF	ID	EX	MEM	WB					
add \$2,\$1,\$1		IF	ID	EX	MEM	WB				
sub \$3,\$0,\$			IF	ID	EX	MEM	WB			

Clock Cycle	IF	ID	EX	MEM	WB
CC1					
CC2					
CC3					
CC4					
CC5					
CC6					
CC7					

Branch Hazards

- When we decide to branch, other instructions are in the pipeline!

- Consider:

beq \$s1, \$s3, Load

and \$t2, \$2, \$5

or \$t3, \$6, \$2

add \$t4, \$2, \$2

....

Load lw \$4, 50(\$7)

If \$1 != \$3 → Machine executes *fall-thru*

**If \$1 == \$3 → Machine sends control to the branch target address
(the lw instruction)**

Suppose branch is taken (\$1 == \$3)

Branch Hazards

Example: **Branch taken**

beq \$s1, \$s3, Load
and \$12, \$2, \$5
or \$13, \$6, \$2
add \$14, \$2, \$2
Load: lw \$4, 50(7)

Inst.	Time (in clock cycles) →-----									
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	
beq	IF	ID	EX	MEM	WB					
and		IF	ID	EX	MEM	WB				
or			IF	ID	EX	MEM	WB			
add				IF	ID	EX	MEM	WB		
lw					IF	ID	EX	MEM	WB	

CC3: EX Stage \$s1 ? \$s3

CC4: MEM Stage sets PC to branch Target address

.....

CC5: IF Stage fetches the lw instruction

Pipeline incorrectly executes leading instructions before executing branch
Branch Hazard

Branch Hazards

Solution

- **Assume Branch will not be Taken**
 - Allow the fall-thru instructions to execute sequentially
 - Flush the instructions if MEM stage discovers that branch should be taken (change instruction to a *nop* in a pipeline)
- **Reducing Delay of Branches**
 - Add hardware to ID stage to test branch condition and compute target address
 - Penalty is only one cycle instead of three cycles
- **Dynamic Branch Prediction**
 - Add branch-target-buffer (BTB) to hardware
 - Records target address of every taken branch + address of branch instr.
 - the PC equals an address in BTB entry → The IF stage sets PC to branch target address in BTB for the following cycle
 - Instructions from BTB flow thru' pipeline immediately; and flushed if branch is not taken
 - Fall-thru' Instructions, follow the pipeline, and are flushed if branch is taken

Exceptions

- Exception
 - Any unexpected change in program control flow
 - **Interrupt**: Exception caused by an external event (I/O device)
 - Used to detect overflow

Examples of events that trigger exception

Event Type	Scenario
I/O Device Request	Interrupt
Invoke OS from User program	Exception
Arithmetic Overflow	Exception
Using an undefined Instruction	Exception
Hardware Malfunctions	Exception

How Exceptions are Handled

MIPS

■ Triggers:

- Using undefined Instructions
- Arithmetic Overflow

■ Actions

- Processor saves address of offending instruction in EPC
- Processor transfers control to OS
- OS Performs appropriate action depending on event
 - Predefine action in response to overflow or
 - Stops execution of program and report error
- OS either terminates program or returns control to Processor
- Processor uses EPC to restart program execution (fetch the next Instruction)

Exceptions in a Pipelined Computer

- Consider an Arithmetic Overflow:
 - add \$1, \$2, \$1

Pipeline Stages	Action for R-type Instructions
Instruction fetch (IF)	$IR \leq \text{Memory}[PC]$ $PC \leq PC + 4$
Instruction decode/register fetch (ID)	$A \leq \text{Reg}[IR[25:21]]$ $B \leq \text{Reg}[IR[20:16]]$ $ALUOut \leq PC + (\text{sign-extend}(IR[15:0]) \ll 2)$
Execution, address computation, branch/jump completion (EX)	$ALUOut \leq A \text{ op } B$
Memory access or R-type completion (MEM)	$\text{Reg}[IR[15:11]] \leq ALUOut$
Memory read completion (WB)	




Overflow detected in the EX Stage

Exceptions in a Pipelined Computer

CC3: EX/MEM Pipeline Register is contaminated

CC4: Trailing instruction (sub \$3, \$2, \$1) references corrupt data in EX stage

CC4: Trailing instruction (add \$4, \$1, \$5) references corrupt data in ID stage

Inst.	Time (in clock cycles) →-----									
	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	
add \$1, \$2, \$1	IF	ID	EX 	MEM	WB					
sub \$3, \$2, \$1		IF	ID 	EX	MEM	WB				
add \$4,\$1,\$5			ID 	EX	MEM	WB				
or \$8,\$2, \$6				IF	ID	EX	MEM	WB		
add \$9,\$4,\$2					IF	ID	EX	MEM	WB	
slt \$1, \$6,\$7						IF	ID	EX	MEM	WB

Exceptions in a Pipelined Computer

Actions

- Processor transfers control to exception routine
- Flush Instruction in ID stage (Control: *ID.Flush Or'ed* “HDU Stall Signal”)
- Flush Instruction in EX stage (Control: *EX.Flush*)
- Flush instruction in IF Stage (*nop*)
- Save offending address (+ 4 bytes) in EPC

Advanced Pipelining

Extracting More Performance

- **Increase depth of pipeline to overlap more instructions (Instruction Level Parallelism)**
 - Move from 5-stage to an n-stage ($n > 5$)
 - Increases throughput per clock cycle
- **Launch more than one instruction (IF) per clock cycle (multiple issue)**
 - Replicate internal components in datapath (*hardware cost*)
- **Loop Unrolling**
 - Better scheduling of sequential and iterative access to an array
 - Make several copies of the loop body and re-schedule in a single step (*copies must be independent*)

Advanced Pipelining

Extracting More Performance

- **Dynamic Multiple-Issue Processor (Superscalar)**
 - Compiled instructions are scheduled in order
 - Processor determines *dynamically* whether zero, one or more instructions can be executed in a given clock cycle (dependency)
 - All modern processors are superscalar and issue multiple instructions usually with some limitations (e.g., “n”-stage pipeline with “m” instruction issue)