- **CPU performance equation:**

$$T = I \times CPI \times C$$

Both effective CPI and clock cycle C are heavily influenced by CPU design.

- **For single-cycle CPU:**

CPI = 1 – good
Long cycle time – bad

- **On the other hand, for multi-cycle CPU:**

CPI increased (3-5) – bad
Shorter cycle – good

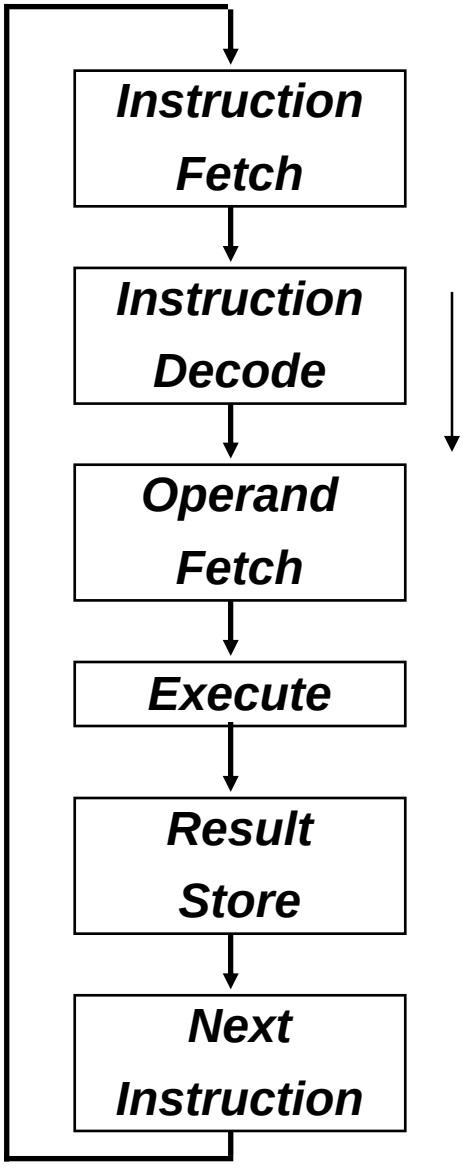
- **How to lower effective CPI without increasing C:**

→ **Solution: CPU Pipelining**

i.e. Instruction processing overlap

Generic CPU Instruction Processing Steps

(Implied by The Von Neumann Computer Model)



Obtain instruction from program storage (memory)

The Program Counter (PC) points to next instruction to be processed

Determine required actions and instruction size

$$T = I \times CPI \times C$$

Locate and obtain operand data

Compute result value or status

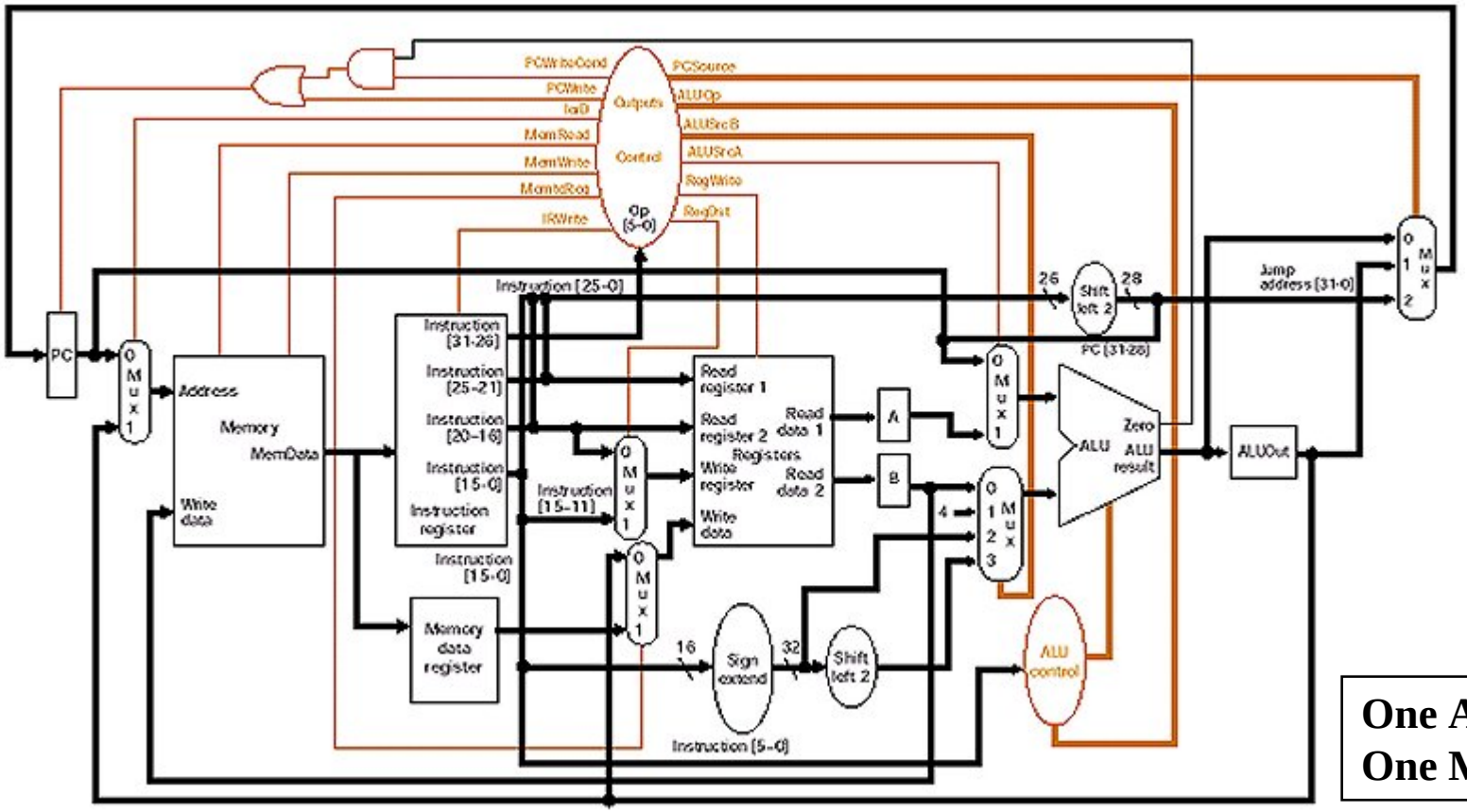
Deposit results in storage for later use

Determine successor or next instruction

(i.e Update PC to fetch next instruction to be processed)

Major CPU Performance Limitation: The Von Neumann computing model implies sequential execution one instruction at a time

MIPS CPU Design: What do we have so far?
Multi-Cycle Datapath (Textbook Version)



One ALU
One Memory

CPI: R-Type = 4, Load = 5, Store 4, Jump/Branch = 3
Only one instruction being processed in datapath

How to lower CPI further without increasing CPU clock cycle time, C?

T = I x CPI x C

Processing an instruction starts when the previous instruction is completed

Operations (Dependant RTN) for Each Cycle

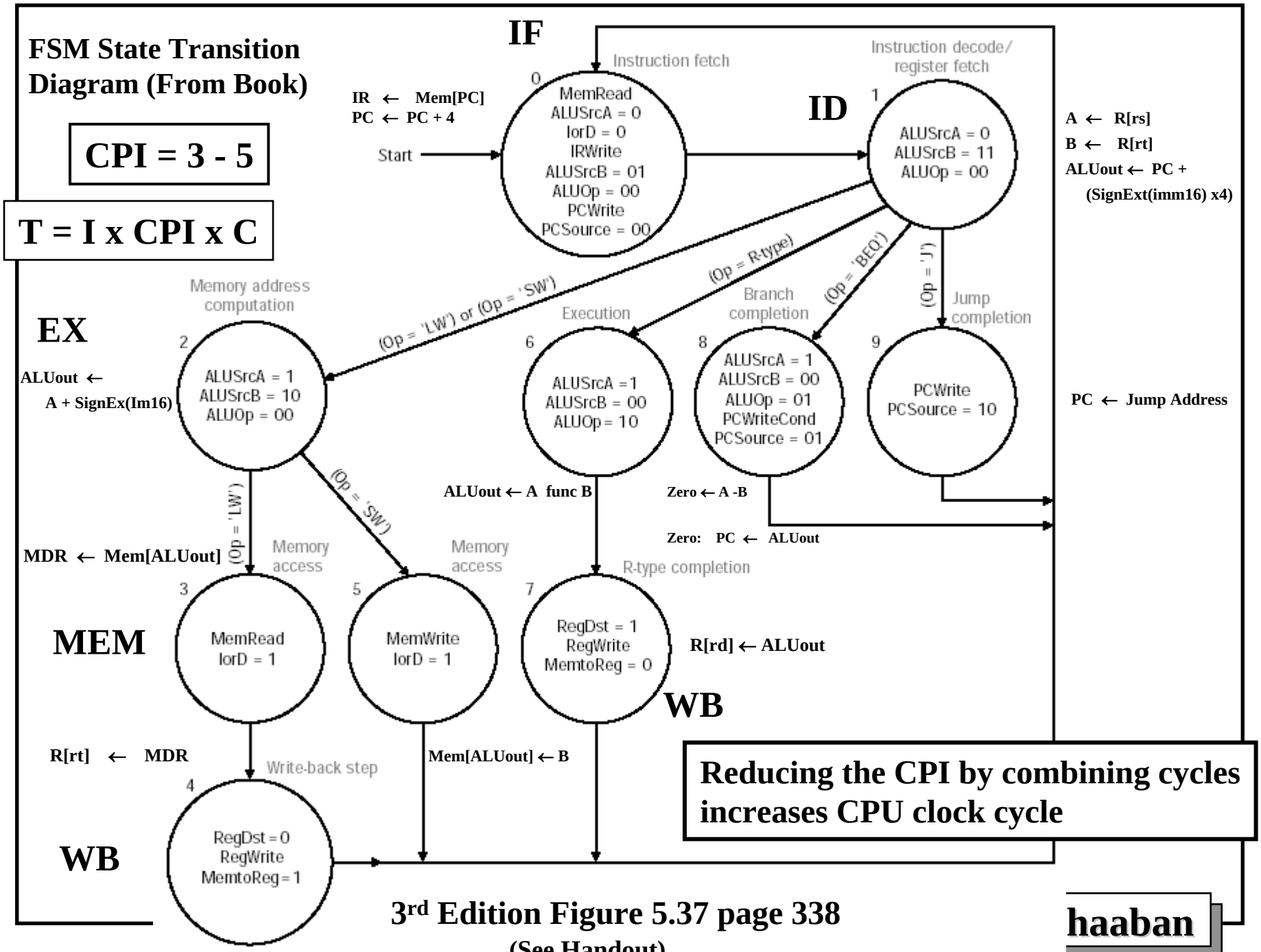
		R-Type	Load	Store	Branch	Jump
IF	Instruction Fetch	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4
ID	Instruction Decode	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)	A ← R[rs] B ← R[rt] ALUout ← PC + (SignExt(imm16) x4)
EX	Execution	ALUout ← A funct B	ALUout ← A + SignEx(Imm16)	ALUout ← A + SignEx(Imm16)	Zero ← A - B Zero: PC ← ALUout	PC ← Jump Address
MEM	Memory		MDR ← Mem[ALUout]	Mem[ALUout] ← B		
WB	Write Back	R[rd] ← ALUout	R[rt] ← MDR			

CPI = 3 – 5 C = 2ns

T = I x CPI x C

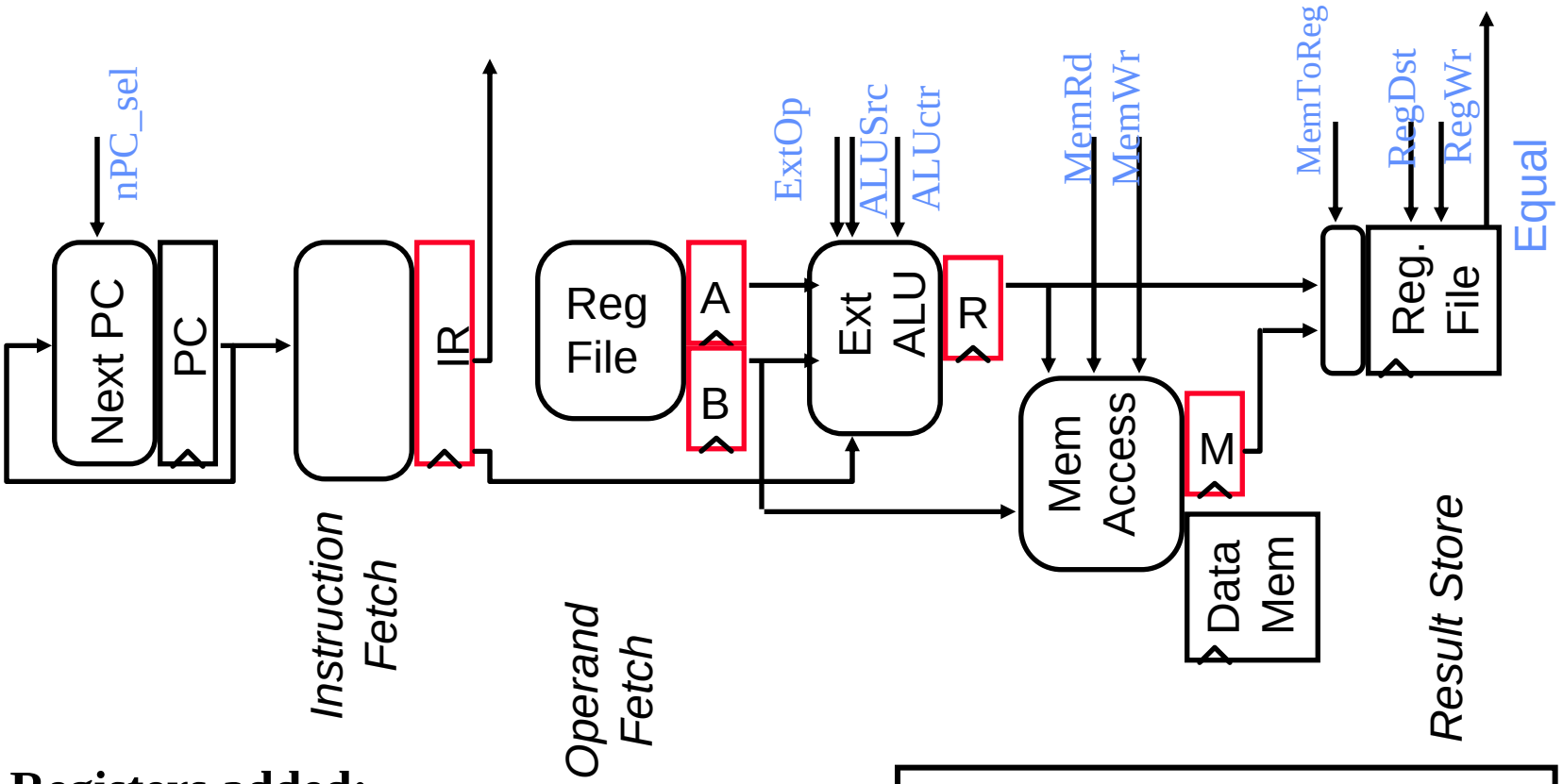
Reducing the CPI by combining cycles increases CPU clock cycle

Instruction Fetch (IF) & Instruction Decode (ID) cycles are common for all instructions



3rd Edition Figure 5.37 page 338
(See Handout)

Multi-cycle Datapath (Our Version)



Registers added:

Three ALUs, Two Memories

IR: **Instruction register**

A, B: **Two registers to hold operands read from register file.**

R: **or ALUOut, holds the output of the ALU**

M: **or Memory data register (MDR) to hold data read from data memory**

Operations (Dependant RTN) for Each Cycle

		R-Type	Logic Immediate	Load	Store	Branch
IF	Instruction Fetch	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4	IR ← Mem[PC] PC ← PC + 4
ID	Instruction Decode	A ← R[rs] B ← R[rt]	A ← R[rs] B ← R[rt	A ← R[rs] B ← R[rt	A ← R[rs] B ← R[rt]	A ← R[rs] B ← R[rt]
EX	Execution	R ← A funct B	R ← A OR ZeroExt[imm16]	R ← A + SignEx(Im16)	R ← A + SignEx(Im16)	Zero ← A - B If Zero = 1: PC ← PC + (SignExt(imm16) x4)
MEM	Memory			M ← Mem[R]	Mem[R] ← B	
WB	Write Back	R[rd] ← R	R[rt] ← R	R[rt] ← M		

CPI = 3 – 5 C = 2ns

Instruction Fetch (IF) & Instruction Decode cycles are common for all instructions

Multi-cycle Datapath Instruction CPI

- **R-Type/Immediate: Require four cycles, CPI =4**
  - **IF, ID, EX, WB**
- **Loads: Require five cycles, CPI = 5**
  - **IF, ID, EX, MEM, WB**
- **Stores: Require four cycles, CPI = 4**
  - **IF, ID, EX, MEM**
- **Branches: Require three cycles, CPI = 3**
  - **IF, ID, EX**
- **Average program $3 \leq CPI \leq 5$ depending on program profile (instruction mix).**

Non-overlapping Instruction Processing:
Processing an instruction starts when the previous instruction is completed.

MIPS Multi-cycle Datapath Performance Evaluation

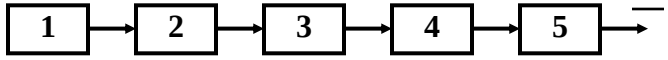
- **What is the average CPI?**
  - **State diagram gives CPI for each instruction type.**
  - **Workload (program) below gives frequency of each type.**

Type	CPI_i for type	Frequency	CPI_i x $freqI_i$
Arith/Logic	4	40%	1.6
Load	5	30%	1.5
Store	4	10%	0.4
branch	3	20%	0.6
		Average CPI:	4.1

Better than CPI = 5 if all instructions took the same number of clock cycles (5).

Instruction CPU Pipelining

- **Instruction pipelining is a CPU implementation technique where multiple operations on a number of instructions are overlapped.**
  - **For Example: The next instruction is fetched in the next cycle without waiting for the current instruction to complete.**
- **An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction. Each step is called *a pipeline stage* or *a pipeline segment.***
- **The stages or steps are connected in a linear fashion: one stage to the next to form the pipeline (or pipelined CPU datapath) -- instructions enter at one end and progress through the stages and exit at the other end.**



5 stage pipeline

- **The time to move an instruction one step down the pipeline is is equal to *the machine (CPU) cycle* and is determined by the stage with the longest processing delay.**
- **Pipelining increases the CPU instruction throughput: The number of instructions completed per cycle.**
  - **Instruction Pipeline Throughput : The instruction completion rate of the pipeline and is determined by how often an instruction exists the pipeline.**
  - **Under ideal conditions (no stall cycles), instruction throughput is one instruction per machine cycle, or ideal effective CPI = 1** Or ideal IPC = 1

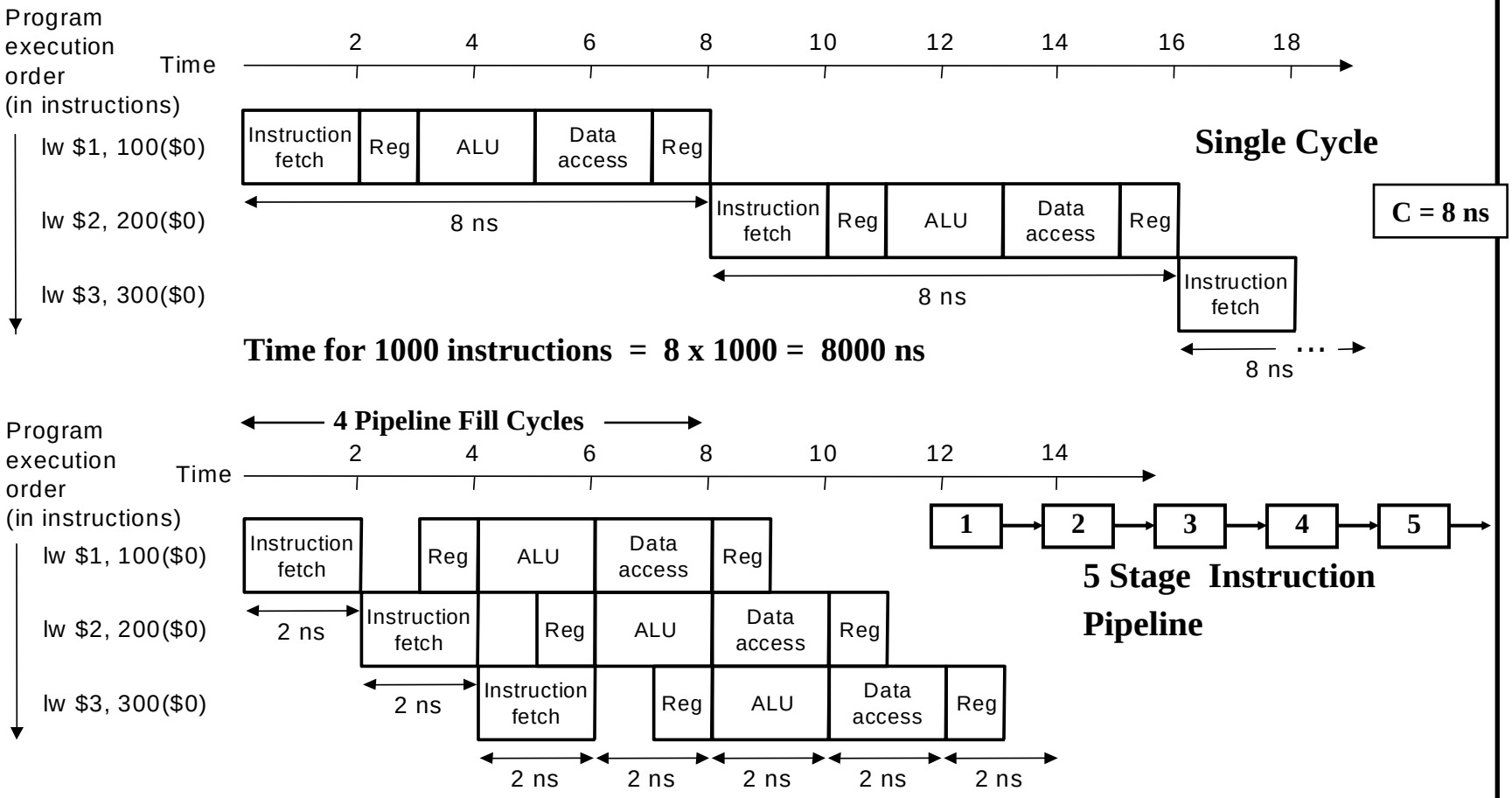
T = I x CPI x C

- **Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency).**

Pipelining may actually increase individual instruction latency

  - **Minimum instruction latency = n cycles, where n is the number of pipeline stages**

Single Cycle Vs. Pipelining



Time for 1000 instructions = time to fill pipeline + cycle time x 1000 = 8 + 2 x 1000 = 2008 ns

Pipelining Speedup = 8000/2008 = 3.98

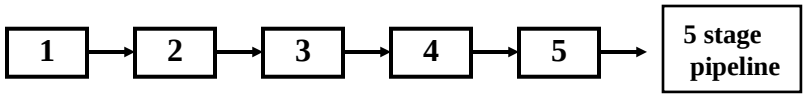
Assuming the following datapath/control hardware components delays:
Memory Units: 2 ns ALU and adders: 2 ns
Register File: 1 ns Control Unit < 1 ns

CPU Pipelining: Design Goals

- The length of the machine clock cycle is determined by the time required for the slowest pipeline stage.

Similar to non-pipelined multi-cycle CPU

- An important pipeline design consideration is to balance the length of each pipeline stage.



- If all stages are perfectly balanced, then the effective time per instruction on a pipelined machine (assuming ideal conditions with no stalls):

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipeline stages}}$$

- **Under these ideal conditions:**
  - Speedup from pipelining = the number of pipeline stages = n
  - **Goal:** One instruction is completed every cycle: **CPI = 1 .**

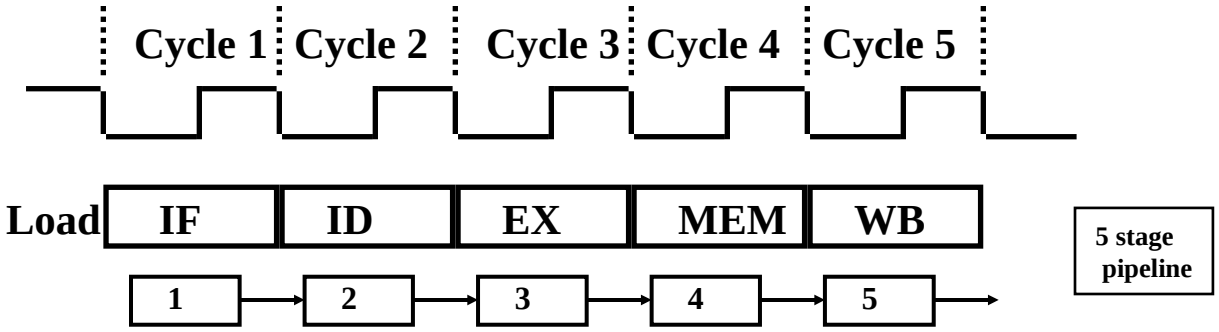
T = I x CPI x C

While keeping clock cycle C short

From MIPS Multi-Cycle Datapath:

Five Stages of Load

5 steps
or 5 cycles
or Stages
n = 5



1- Instruction Fetch (IF) Instruction Fetch

And PC update PC ← PC + 4

- Fetch the instruction from the Instruction Memory.

2- Instruction Decode (ID): Registers Fetch and Instruction Decode.

3- Execute (EX): Calculate the memory address.

4- Memory (MEM): Read the data from the Data Memory.

5- Write Back (WB): Write the data back to the register file.

n = number of pipeline stages (5 in this case)

The number of pipeline stages is determined by the instruction that needs the largest number of cycles

Ideal Pipelined Instruction Processing Timing Representation

(i.e no stall cycles)

CPI = 1 (ideal)

n = 5 stage pipeline

Fill Cycles = number of stages -1 = n -1

Clock cycle Number → Time in clock cycles →

	Instruction Number	1	2	3	4	5	6	7	8	9
1	Instruction I	IF	ID	EX	MEM	WB				
2	Instruction I+1		IF	ID	EX	MEM	WB			
3	Instruction I+2			IF	ID	EX	MEM	WB		
4	Instruction I+3				IF	ID	EX	MEM	WB	
5	Instruction I +4					IF	ID	EX	MEM	WB

Program Order

Ideal CPI = 1

(or IPC =1)

4 cycles = n -1 = 5 -1

Time to fill the pipeline

First instruction, I Completed

Instruction, I+4 completed

n= 5 Pipeline Stages:

1. IF = Instruction Fetch
2. ID = Instruction Decode
3. EX = Execution
4. MEM = Memory Access
5. WB = Write Back

Any individual instruction goes through all five pipeline stages taking 5 cycles to complete
Thus instruction latency= 5 cycles

Pipeline Fill Cycles: No instructions completed yet
Number of fill cycles = Number of pipeline stages - 1
Here 5 - 1 = 4 fill cycles

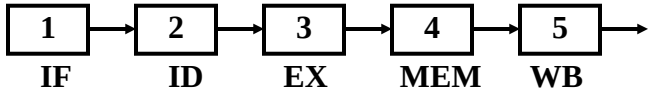
Ideal pipeline operation: After fill cycles, one instruction is completed per cycle giving the ideal pipeline CPI = 1 (ignoring fill cycles) or Instructions per Cycle = IPC = 1/CPI = 1

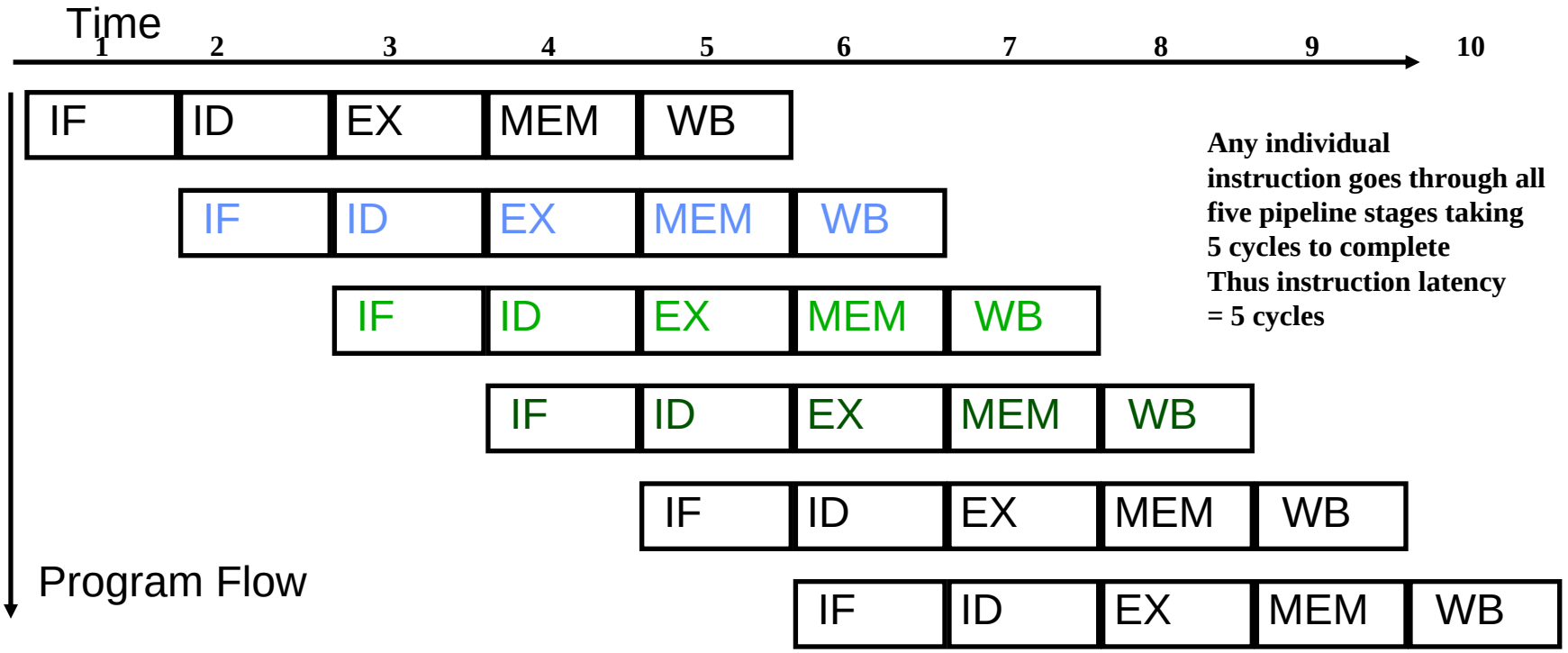
Ideal pipeline operation without any stall cycles

Ideal Pipelined Instruction Processing Representation

(i.e no stall cycles)

5 Stage Pipeline





Any individual instruction goes through all five pipeline stages taking 5 cycles to complete
Thus instruction latency = 5 cycles

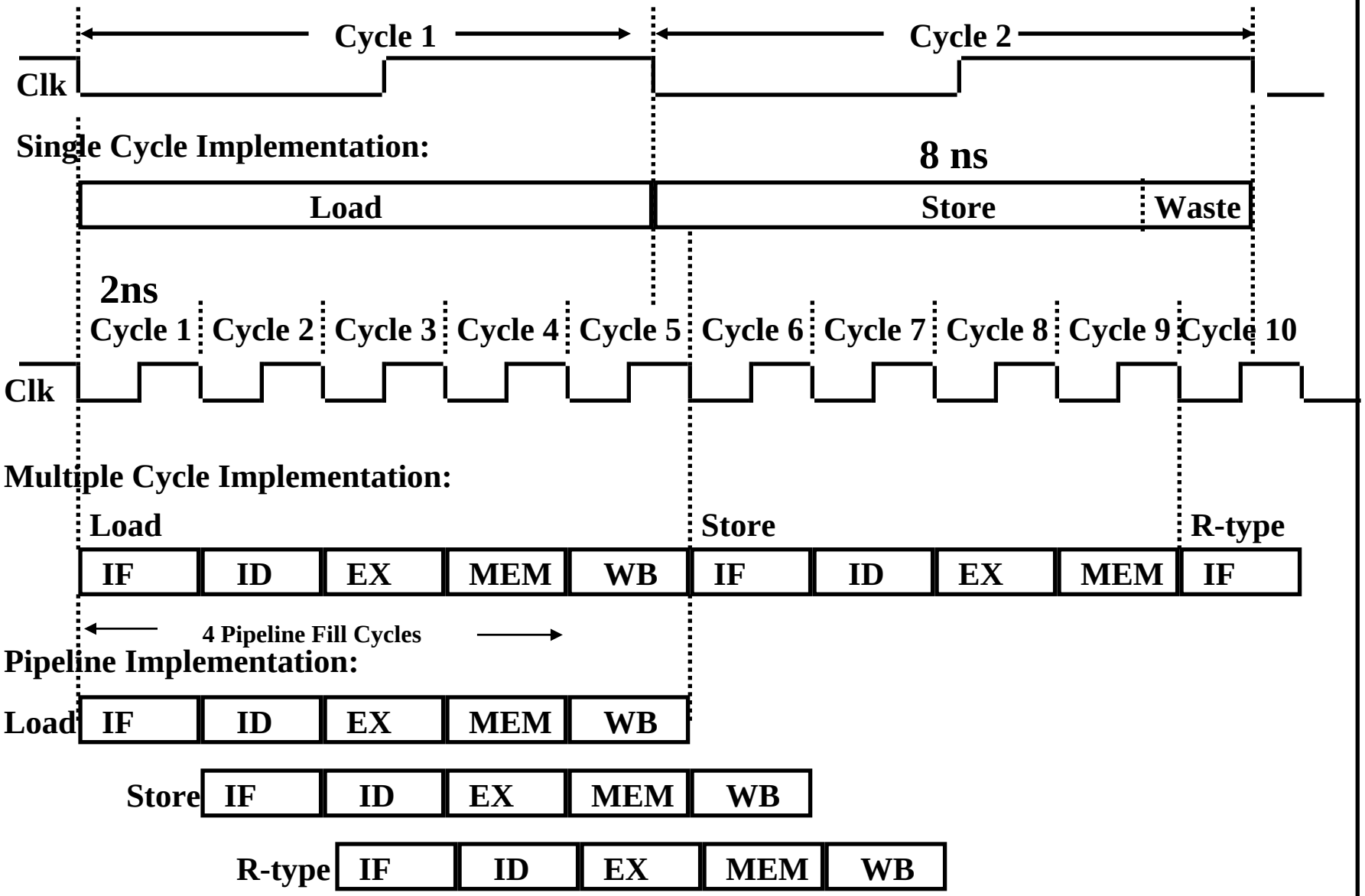
Here n = 5 pipeline stages or steps

Number of pipeline fill cycles = Number of stages - 1 Here 5 -1 = 4

After fill cycles: One instruction is completed every cycle (Effective CPI = 1) (ideally)

Ideal pipeline operation without any stall cycles

Single Cycle, Multi-Cycle, Vs. Pipelined CPU



Assuming the following datapath/control hardware components delays:
Memory Units: 2 ns ALU and adders: 2 ns
Register File: 1 ns Control Unit < 1 ns

Single Cycle, Multi-Cycle, Pipeline: Performance Comparison Example

For 1000 instructions, execution time:

T = I x CPI x C

- **Single Cycle Machine:** **CPI = 1 C = 8 ns**
  - **8 ns/cycle x 1 CPI x 1000 inst = 8000 ns**

- **Multi-cycle Machine:** **3 ≤ CPI ≤ 5 C = 2 ns**
  - **2 ns/cycle x 4.6 CPI (due to inst mix) x 1000 inst = 9200 ns**

Depends on program instruction mix

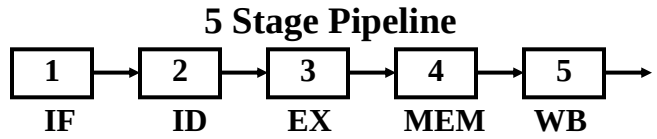
- **Ideal pipelined machine, 5-stages:** **Effective CPI = 1 C = 2 ns**
  - **2 ns/cycle x (1 CPI x 1000 inst + 4 cycle fill) = 2008 ns**
    - **Speedup = 8000/2008 = 3.98 times faster than single cycle CPU**
    - **Speedup = 9200/2008 = 4.58 times faster than multi cycle CPU**

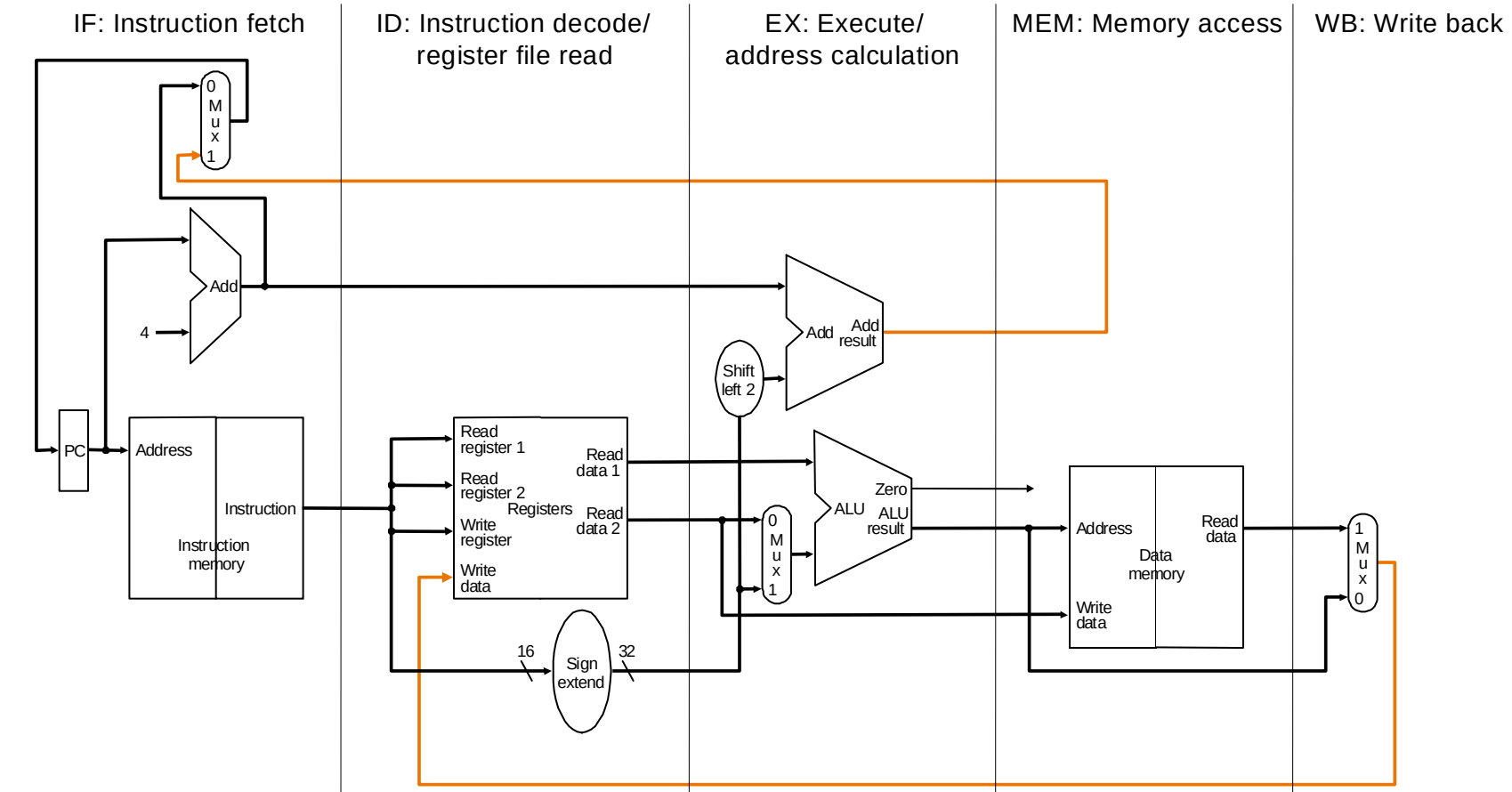
Basic Pipelined CPU Design Steps

1. Analyze instruction set operations using independent RTN => datapath requirements.
2. Select required datapath components and connections.
3. Assemble an initial datapath meeting the ISA requirements.
4. Identify pipeline stages based on operation, balancing stage delays, and ensuring no hardware conflicts exist when common hardware is used by two or more stages simultaneously in the same cycle.
5. Divide the datapath into the stages identified above by adding buffers between the stages of sufficient width to hold:
   - 1 Instruction fields.
   - 2 Remaining control lines needed for remaining pipeline stages.
   - 3 All results produced by a stage and any unused results of previous stages.

   i.e registers
6. Analyze implementation of each instruction to determine setting of control points that effects the register transfer taking pipeline hazard conditions into account . (More on this a bit later)
7. Assemble the control logic.

MIPS Pipeline Stage Identification





1 IF Stage 1

2 ID Stage 2

3 EX Stage 3

4 MEM Stage 4

5 WB Stage 5

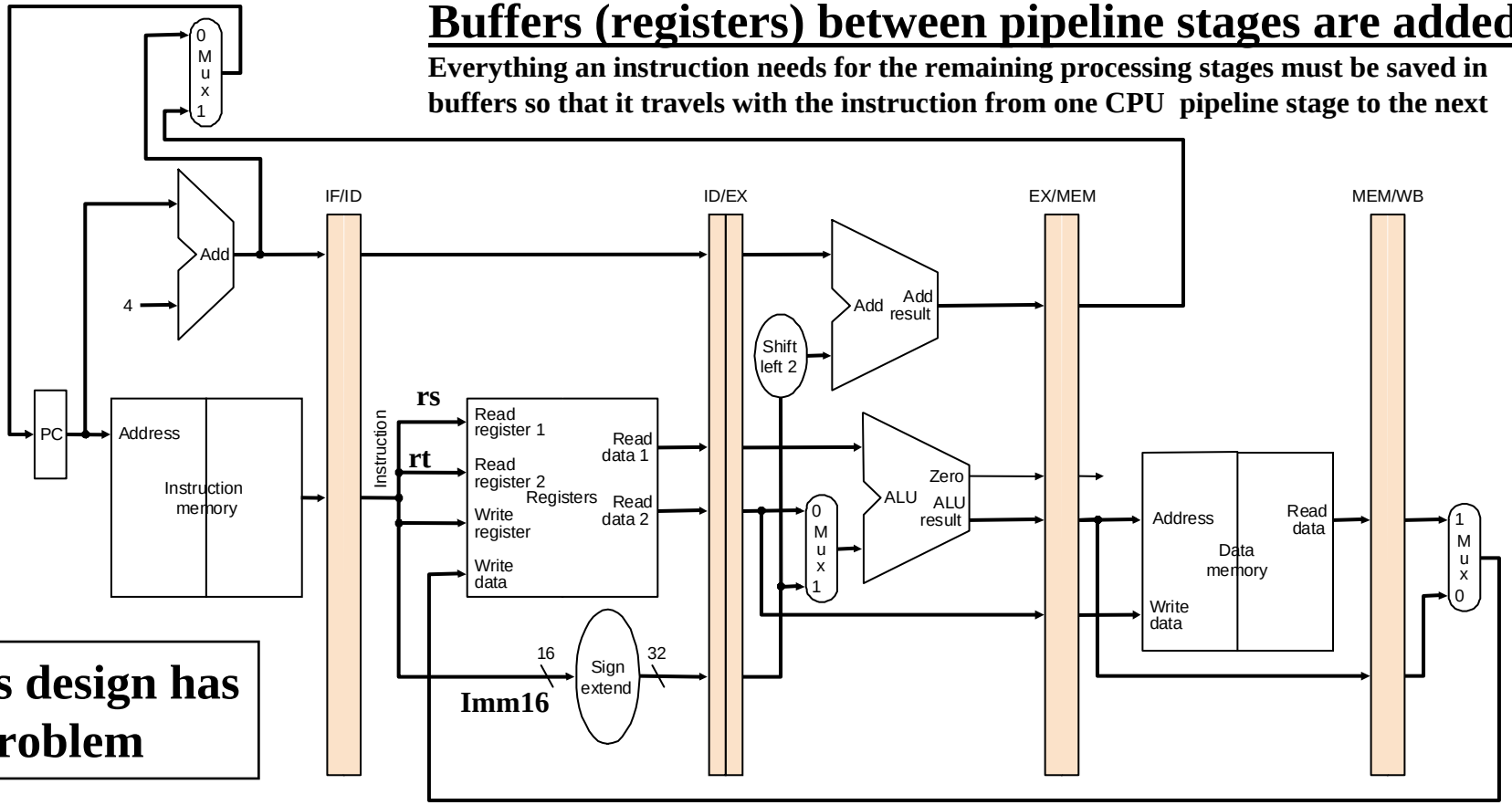
What is needed to divide datapath into pipeline stages?

Start with initial datapath with: 3 ALUs, 2 Memories

MIPS: An Initial Pipelined Datapath

Buffers (registers) between pipeline stages are added:

Everything an instruction needs for the remaining processing stages must be saved in buffers so that it travels with the instruction from one CPU pipeline stage to the next



This design has A problem

IF	ID	EX	MEM	WB
Instruction Fetch	**Instruction Decode**	**Execution**	**Memory**	**Write Back**
Stage 1	**Stage 2**	**Stage 3**	**Stage 4**	**Stage 5**

Can you find a problem even if there are no dependencies?

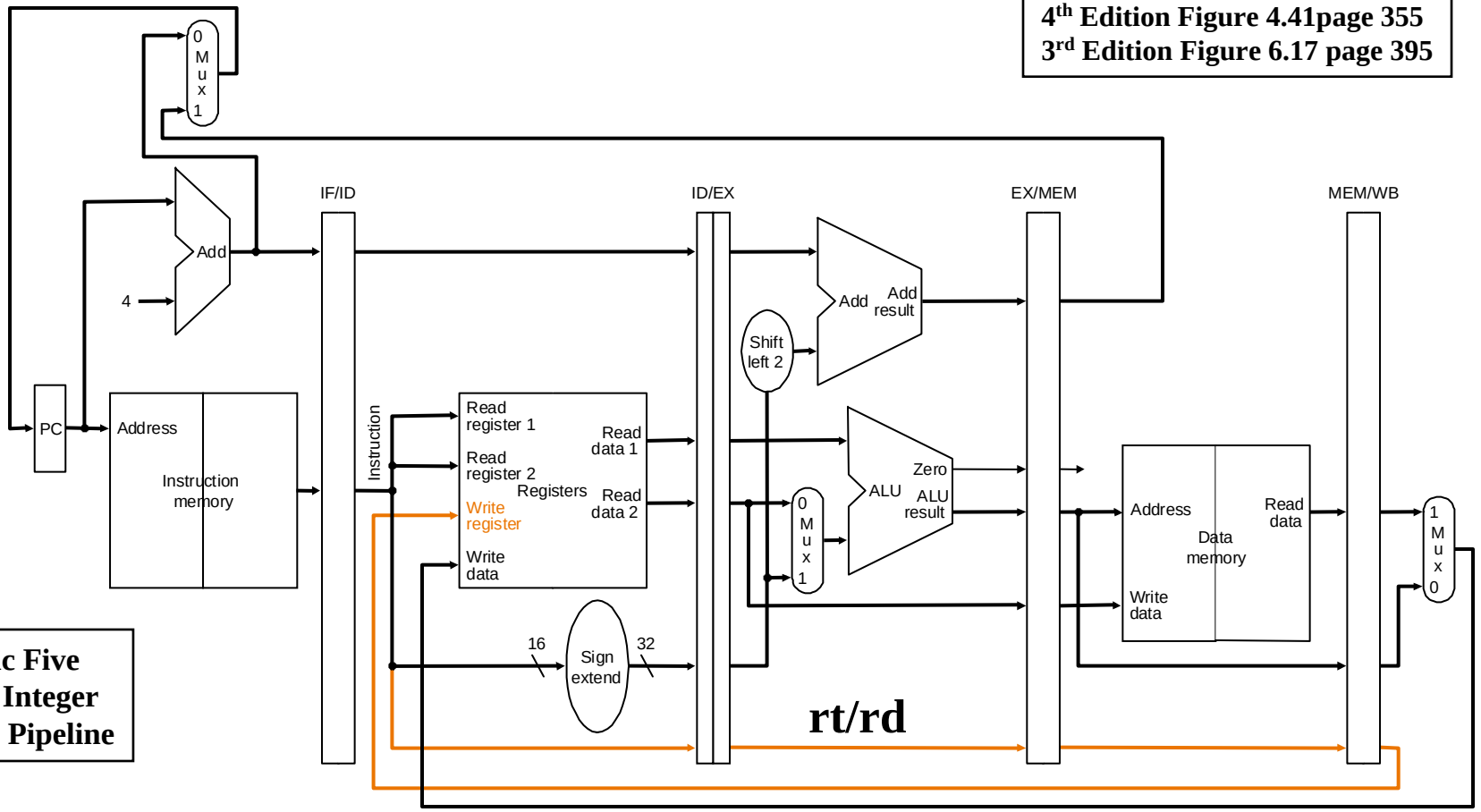
What instructions can we execute to manifest the problem?

n = 5 pipeline stages

Hint: Any values an instruction requires must travel with it as it goes through the pipeline stages including instruction fields still needed in later stages

A Corrected Pipelined Datapath

4th Edition Figure 4.41page 355
3rd Edition Figure 6.17 page 395



Classic Five Stage Integer MIPS Pipeline

IF	ID	EX	MEM	WB
Instruction Fetch	**Instruction Decode**	**Execution**	**Memory**	**Write Back**
Stage 1	Stage 2	Stage 3	Stage 4	Stage 5

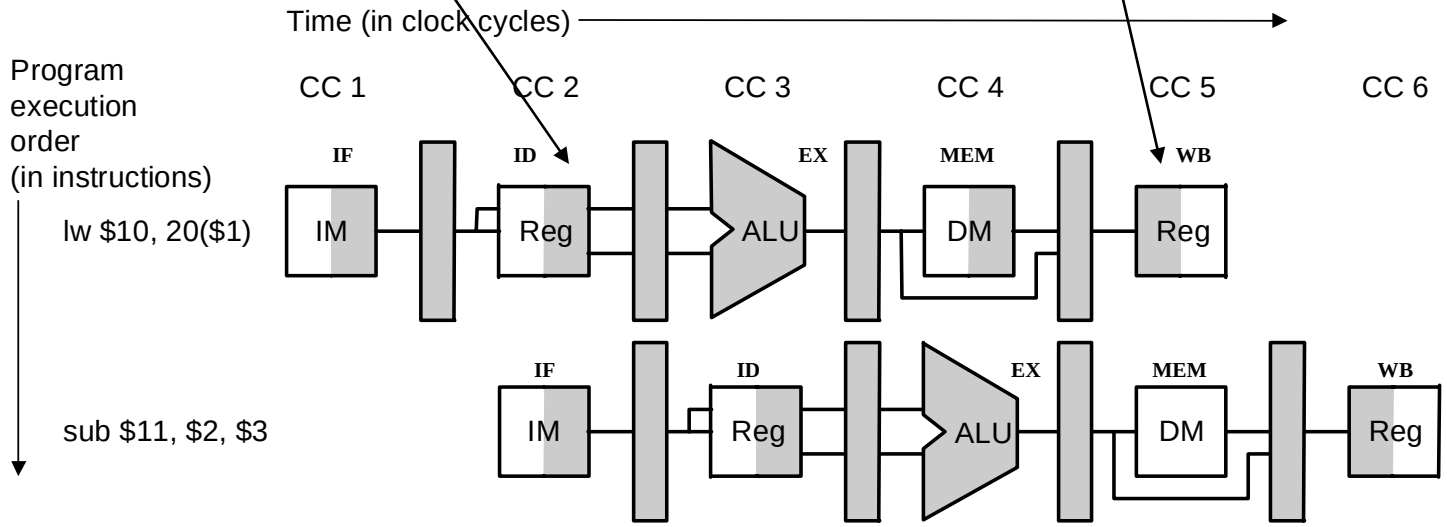
n = 5 pipeline stages

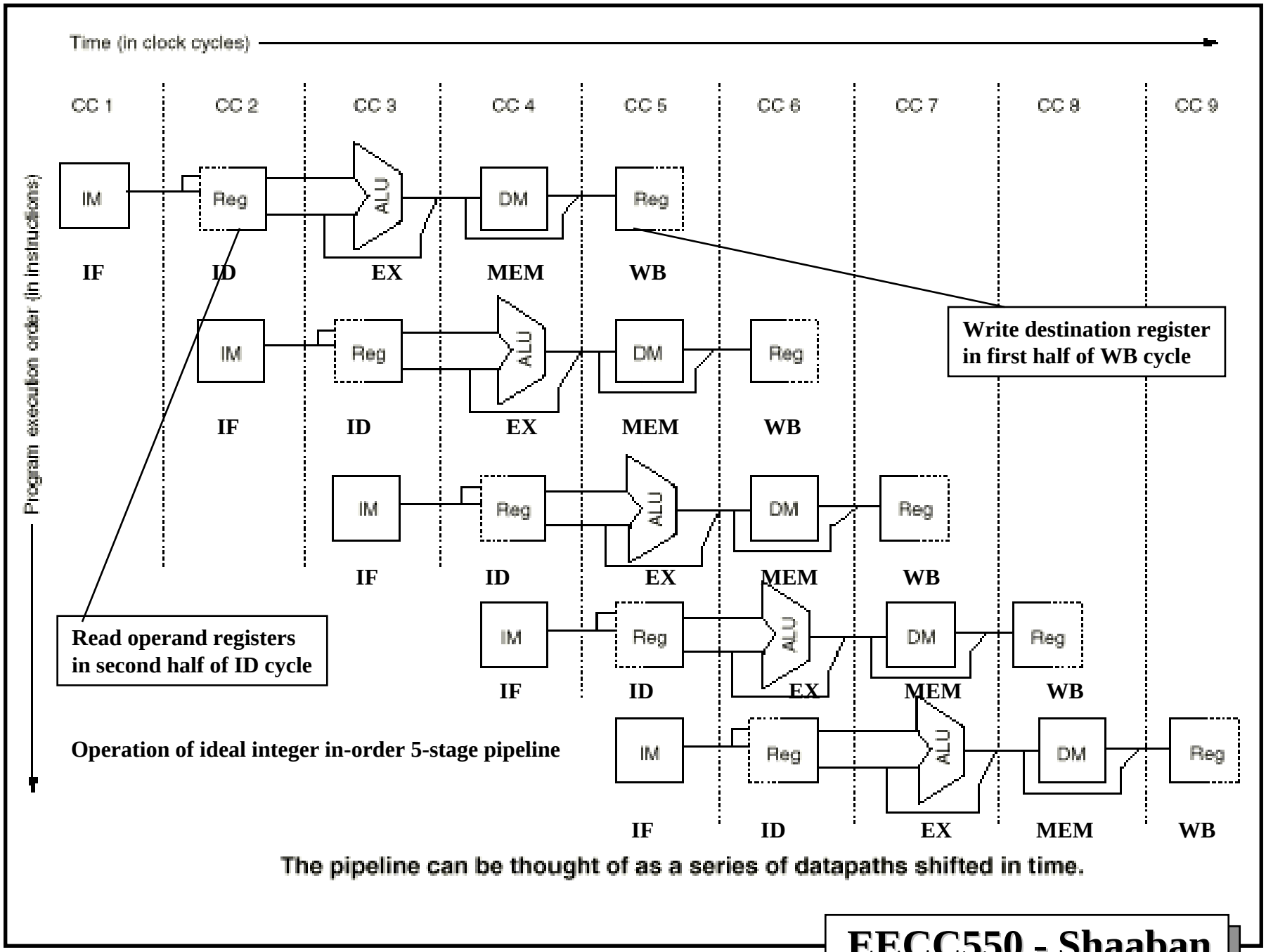
Read/Write Access To Register Bank

- **Two instructions need to access the register bank in the same cycle:**
  - **One instruction to read operands in its instruction decode (ID) cycle.**
  - **The other instruction to write to a destination register in its Write Back (WB) cycle.**
- **This represents a potential hardware conflict over access to the register bank.**
- **Solution: Coordinate register reads and write in the same cycle as follows:**

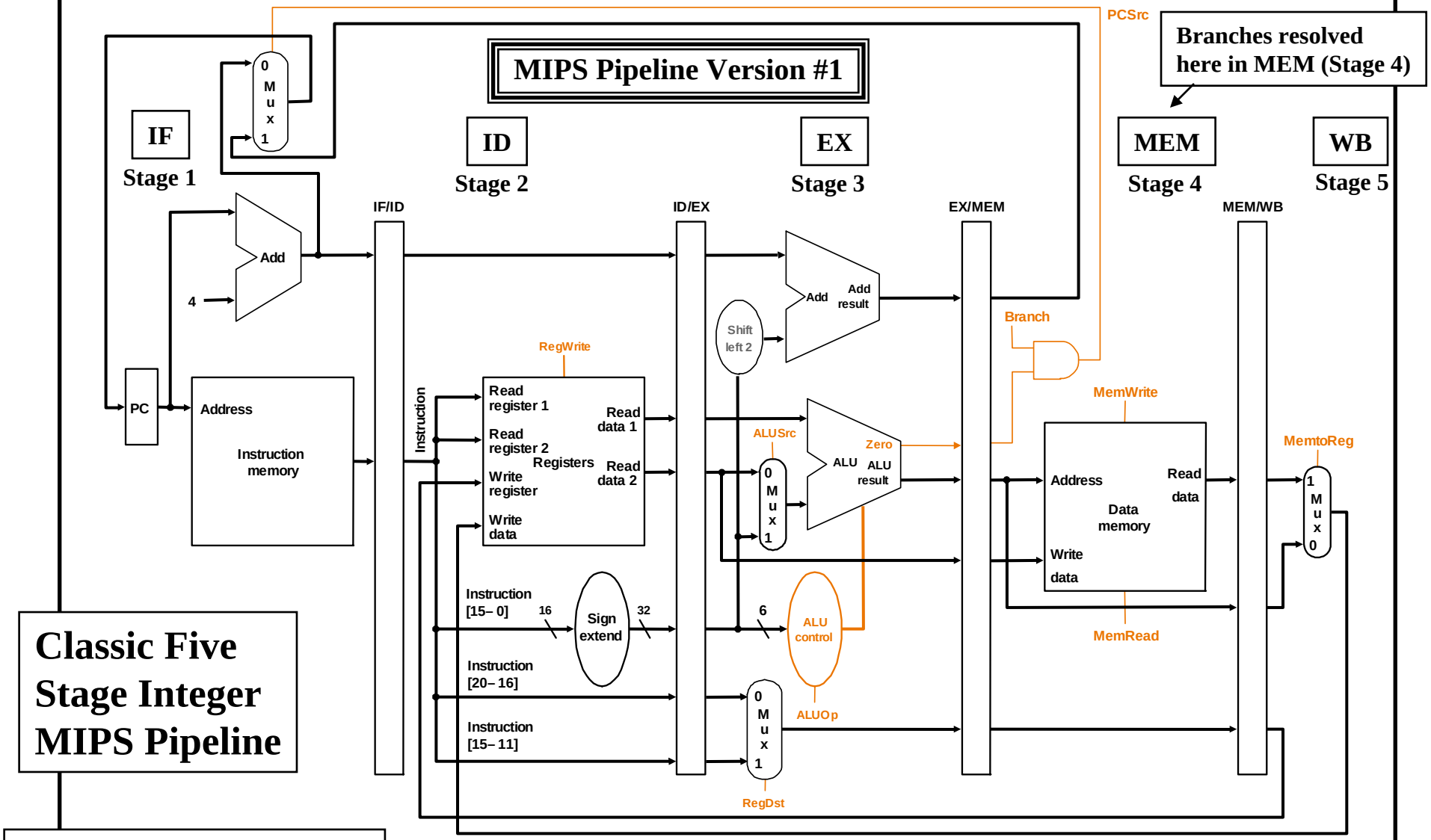
• Operand register reads in Instruction Decode ID cycle occur in the second half of the cycle (indicated here by the dark shading of the second half of the cycle)

• Register write in Write Back WB cycle occur in the first half of the cycle. (indicated here by the dark shading of the first half of the WB cycle)




Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
1
2
3
4
5
IM
Reg
ALU
DM
Reg
IF
ID
EX
MEM
WB
Write destination register in first half of WB cycle
Read operand registers in second half of ID cycle
Operation of ideal integer in-order 5-stage pipeline
The pipeline can be thought of as a series of datapaths shifted in time.

Adding Pipeline Control Points



Classic Five Stage Integer MIPS Pipeline

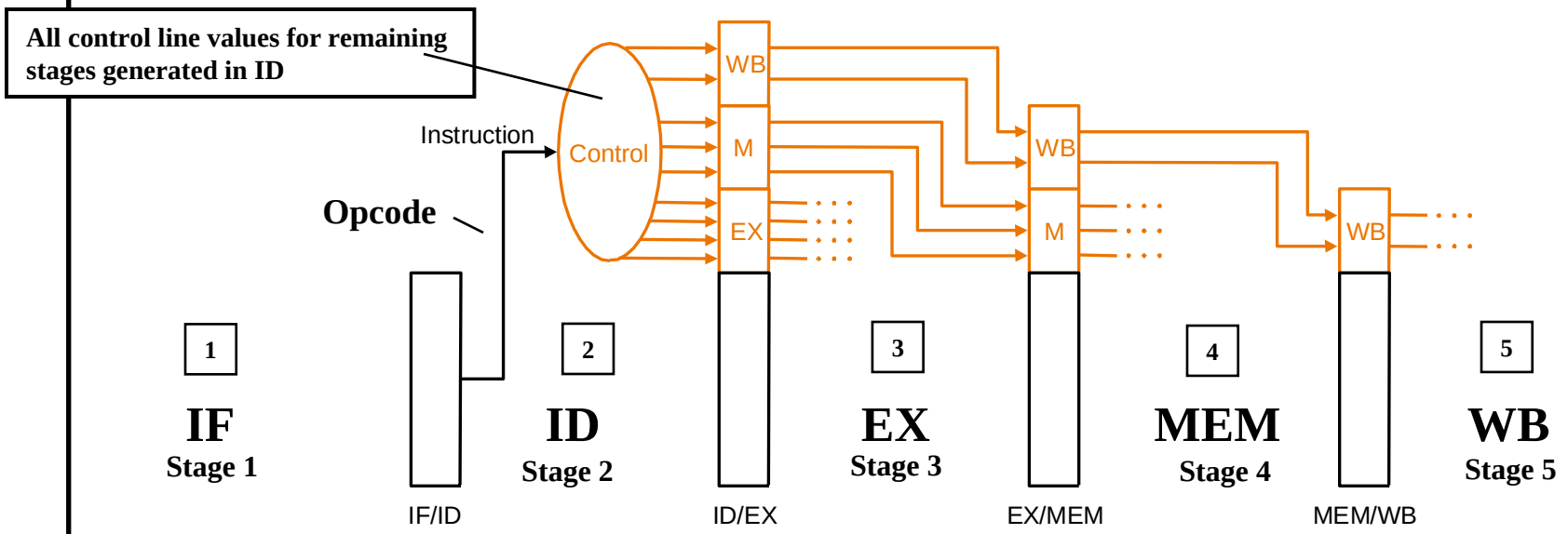
4th Ed. Fig. 4.46 page 359
3rd Ed. Fig. 6.22 page 400

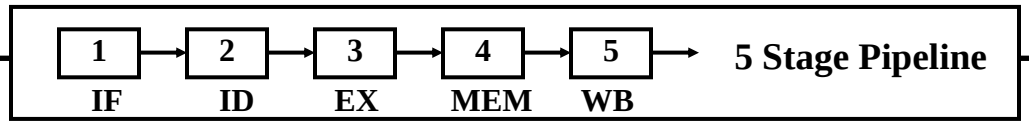
MIPS Pipeline Version 1:
No forwarding, branch resolved in MEM stage

Pipeline Control

- **Pass needed control signals along from one stage to the next as the instruction travels through the pipeline just like the needed data**

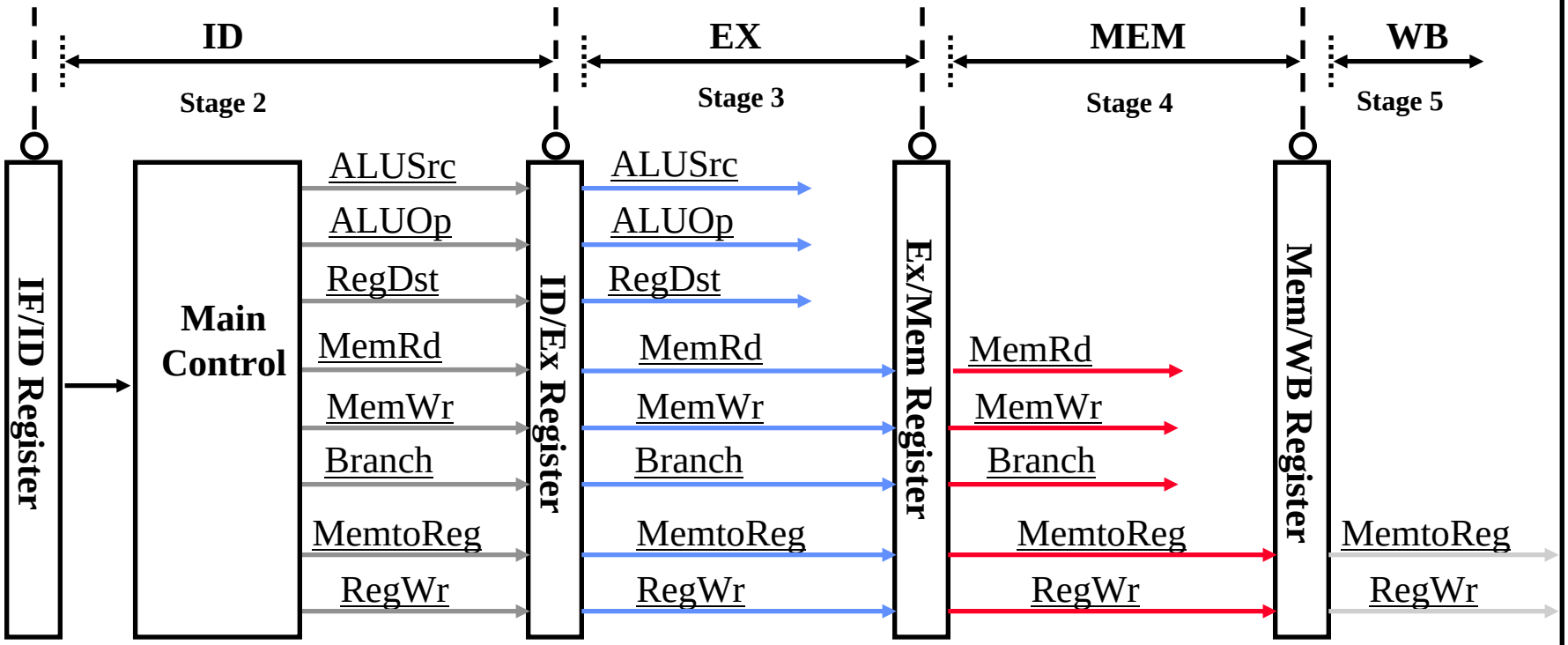
	EX				MEM			WB	
	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
Instruction	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X





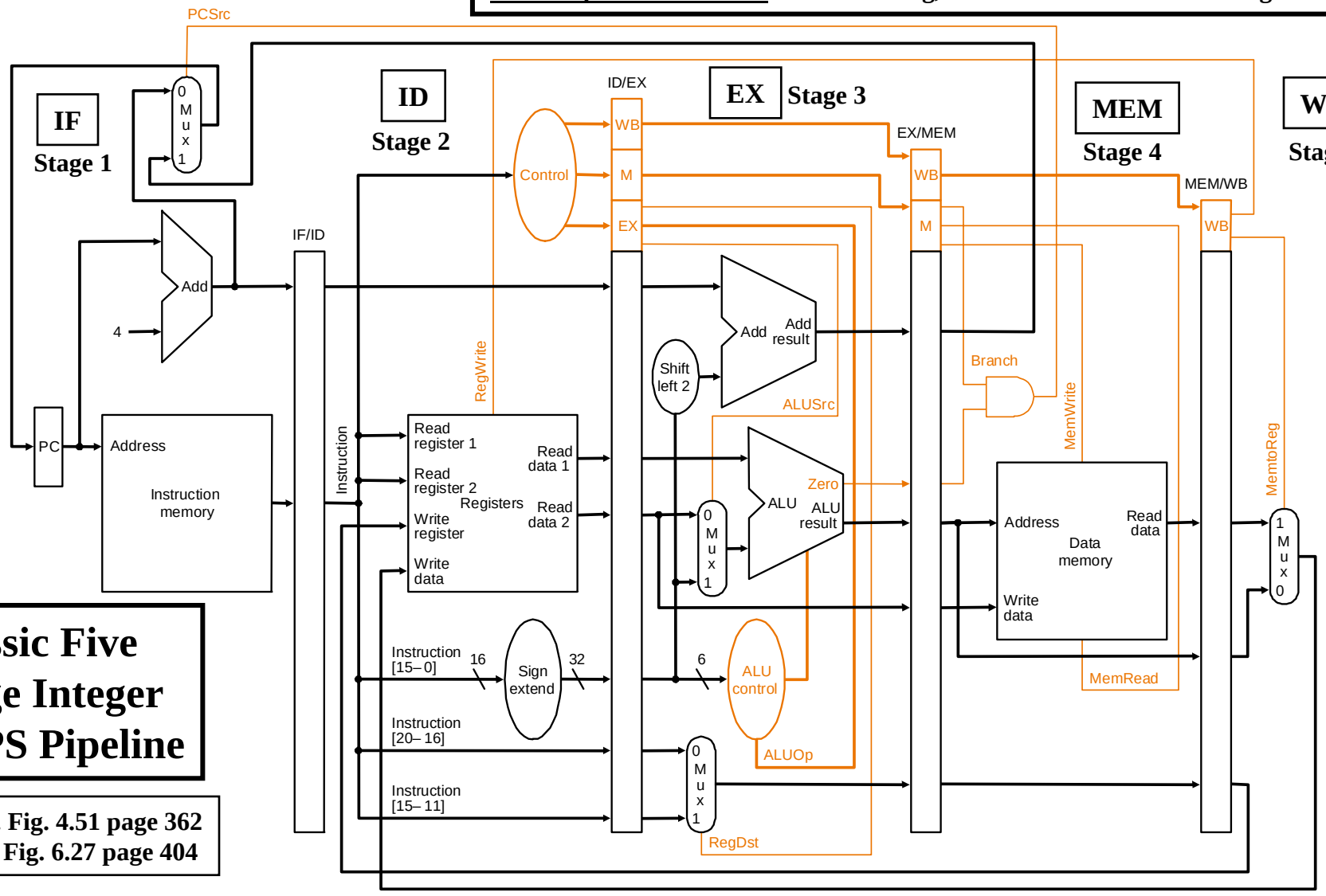
Pipeline Control Signal (Generation/Latching/Propagation)

- **The Main Control generates the control signals during ID**
  - **Control signals for EX (ALUSrc, ALUOp ...) are used 1 cycle later**
  - **Control signals for MEM (MemWr/Rd, Branch) are used 2 cycles later**
  - **Control signals for WB (MemtoReg RegWr) are used 3 cycles later**



Pipelined Datapath with Control Added

MIPS Pipeline Version 1: No forwarding, branch resolved in MEM stage



Classic Five Stage Integer MIPS Pipeline

4th Ed. Fig. 4.51 page 362
3rd Ed. Fig. 6.27 page 404

Target address of branch determined in EX but PC is updated in MEM stage (i.e branch is resolved in MEM, stage 4)

Basic Performance Issues In Pipelining

- **Pipelining increases the CPU instruction throughput:**

  The number of instructions completed per unit time.

  Under ideal conditions (i.e. No stall cycles):

  - **Pipelined CPU instruction throughput is one instruction completed per machine cycle, or CPI = 1** **Ideally**

    (ignoring pipeline fill cycles)

T = I x CPI x C

Or Instruction throughput: Instructions Per Cycle = IPC =1

- **Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency or time).**

  - **It usually slightly increases the execution time of individual instructions over unpipelined CPU implementations due to:**

    - **The increased control overhead of the pipeline and pipeline stage registers delays +**
    - **Every instruction goes though every stage in the pipeline even if the stage is not needed. (i.e MEM pipeline stage in the case of R-Type instructions)**

Here n = 5 stages

Pipelining Performance Example

- **Example: For an unpipelined multicycle CPU:**
  - Clock cycle = 10ns, 4 cycles for ALU operations and branches and 5 cycles for memory operations with instruction frequencies of 40%, 20% and 40%, respectively.
  - If pipelining adds 1ns to the CPU clock cycle then the speedup in instruction execution from pipelining is: ` **i.e. C = 11 ns**

Non-pipelined Average execution time/instruction = Clock cycle x Average CPI

= 10 ns x ((40% + 20%) x 4 + 40%x 5) = 10 ns x 4.4 = 44 ns

CPI = 4.4 **C** **CPI**

In the pipelined CPU implementation, ideal CPI = 1

CPI = 1

Pipelined execution time/instruction = Clock cycle x CPI

= (10 ns + 1 ns) x 1 = 11 ns x 1 = 11 ns

C **CPI**

→ **Speedup from pipelining = Time Per Instruction time unpipelined / Time per Instruction time pipelined**

= 44 ns / 11 ns = 4 times faster

T = I x CPI x C here I did not change

Pipeline Hazards

CPI = 1 + Average Stalls Per Instruction

- Hazards are situations in pipelined CPUs which prevent the next instruction in the instruction stream from executing during the designated clock cycle possibly resulting in one or more stall (or wait) cycles.

i.e A resource the instruction requires for correct execution is not available in the cycle needed

- Hazards reduce the ideal speedup (increase CPI > 1) gained from pipelining and are classified into three classes:

Resource Not available:

– ***Structural hazards***: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.

Hardware Component

Hardware structure (component) conflict

– ***Data hazards***: Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

Correct Operand (data) value

Operand not ready yet when needed in EX

– ***Control hazards***: Arise from the pipelining of conditional branches and other instructions that change the PC.

Correct PC

Correct PC not available when needed in IF

Performance of Pipelines with Stalls

- Hazard conditions in pipelines may make it necessary to stall the pipeline by a number of cycles degrading performance from the ideal pipelined CPU CPI of 1.

Average

CPI pipelined = Ideal CPI + Pipeline stall clock cycles per instruction

= 1 + Pipeline stall clock cycles per instruction

- If pipelining overhead is ignored and we assume that the stages are perfectly balanced then speedup from pipelining is given by:

Speedup = CPI unpipelined / CPI pipelined

= CPI unpipelined / (1 + Pipeline stall cycles per instruction)

- When all instructions in the multicycle CPU take the same number of cycles equal to the number of pipeline stages then:

Speedup = Pipeline depth / (1 + Pipeline stall cycles per instruction)

Structural (or Hardware) Hazards

- **In pipelined machines overlapped instruction execution requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.** *To prevent hardware structures conflicts*

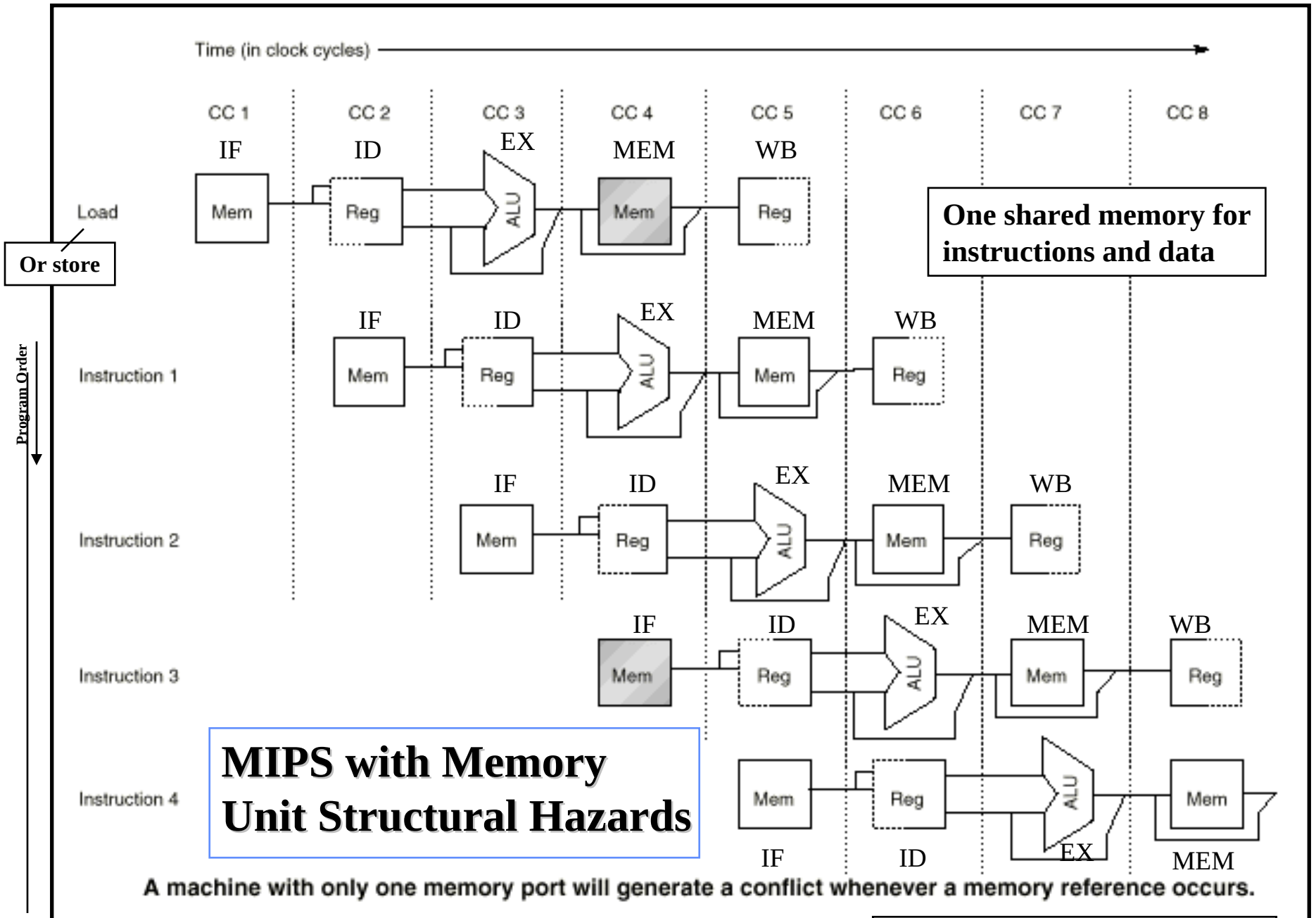
- **If a resource conflict arises due to a hardware resource being required by more than one instruction in a single cycle, and one or more such instructions cannot be accommodated, then a structural hazard has occurred, for example:**

e.g.

  - **When a pipelined machine has a shared single-memory for both data and instructions.**

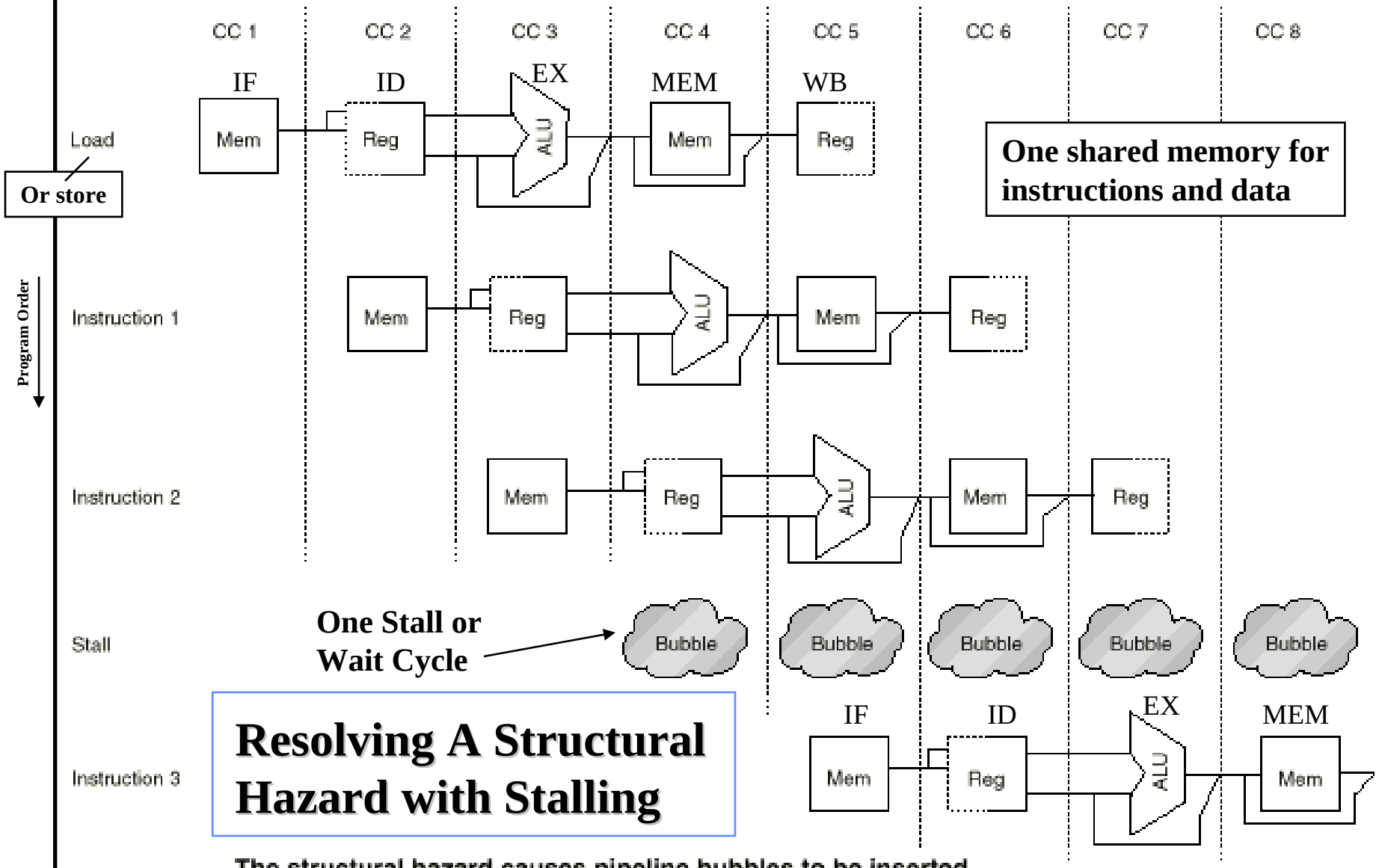
    → **stall the pipeline for one cycle for memory data access**

i.e A hardware component the instruction requires for correct execution is not available in the cycle needed



Instructions 1-4 above are assumed to be instructions other than loads/stores

CPI = 1 + stall clock cycles per instruction = 1 + fraction of loads and stores x 1



Resolving A Structural Hazard with Stalling

The structural hazard causes pipeline bubbles to be inserted.

Instructions 1-3 above are assumed to be instructions other than loads/stores

A Structural Hazard Example

- **Given that data references are 40% (i.e loads/stores) for a specific instruction mix or program, and that the ideal pipelined CPI ignoring hazards is equal to 1.**

- **A machine with a data memory access structural hazards requires a single stall cycle for data references and has a clock rate 1.05 times higher than the ideal machine. Ignoring other performance losses for this machine:**

Average instruction time = CPI X Clock cycle time

Average instruction time = (1 + 0.4 x 1) x $\frac{\text{Clock cycle}_{\text{ideal}}}{1.05}$

CPI = 1.4

= 1.3 X Clock cycle time $_{\text{ideal}}$

i.e. CPU without structural hazard is 1.3 times faster

CPI = 1 + Average Stalls Per Instruction

Data Hazards

i.e Operands

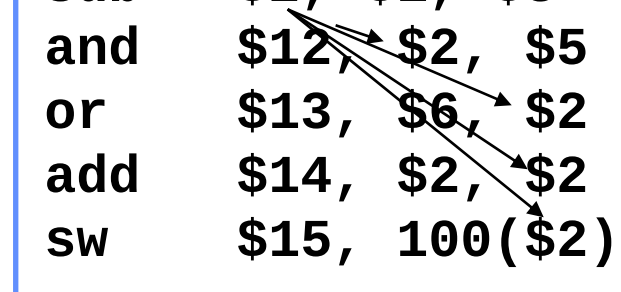
- **Data hazards occur when the pipeline changes the order of read/write accesses to instruction operands in such a way that the resulting access order differs from the original sequential instruction operand access order of the unpipelined CPU resulting in incorrect execution.**
- **Data hazards may require one or more instructions to be stalled in the pipeline to ensure correct execution.**
- **Example:**

CPI = 1 + stall clock cycles per instruction

Producer of Result (data)

Consumers of Result (data)

```
1  sub   $2, $1, $3
2  and   $12, $2, $5
3  or    $13, $6, $2
4  add   $14, $2, $2
5  sw    $15, 100($2)
```



Arrows represent data dependencies between instructions

Instructions that have no dependencies among them are said to be parallel or independent

A high degree of Instruction-Level Parallelism (ILP) is present in a given code sequence if it has a large number of parallel instructions

- All the instructions after the sub instruction use its result data in register $2
- As part of pipelining, these instruction are started before sub is completed:
  - Due to this data hazard instructions need to be stalled for correct execution.

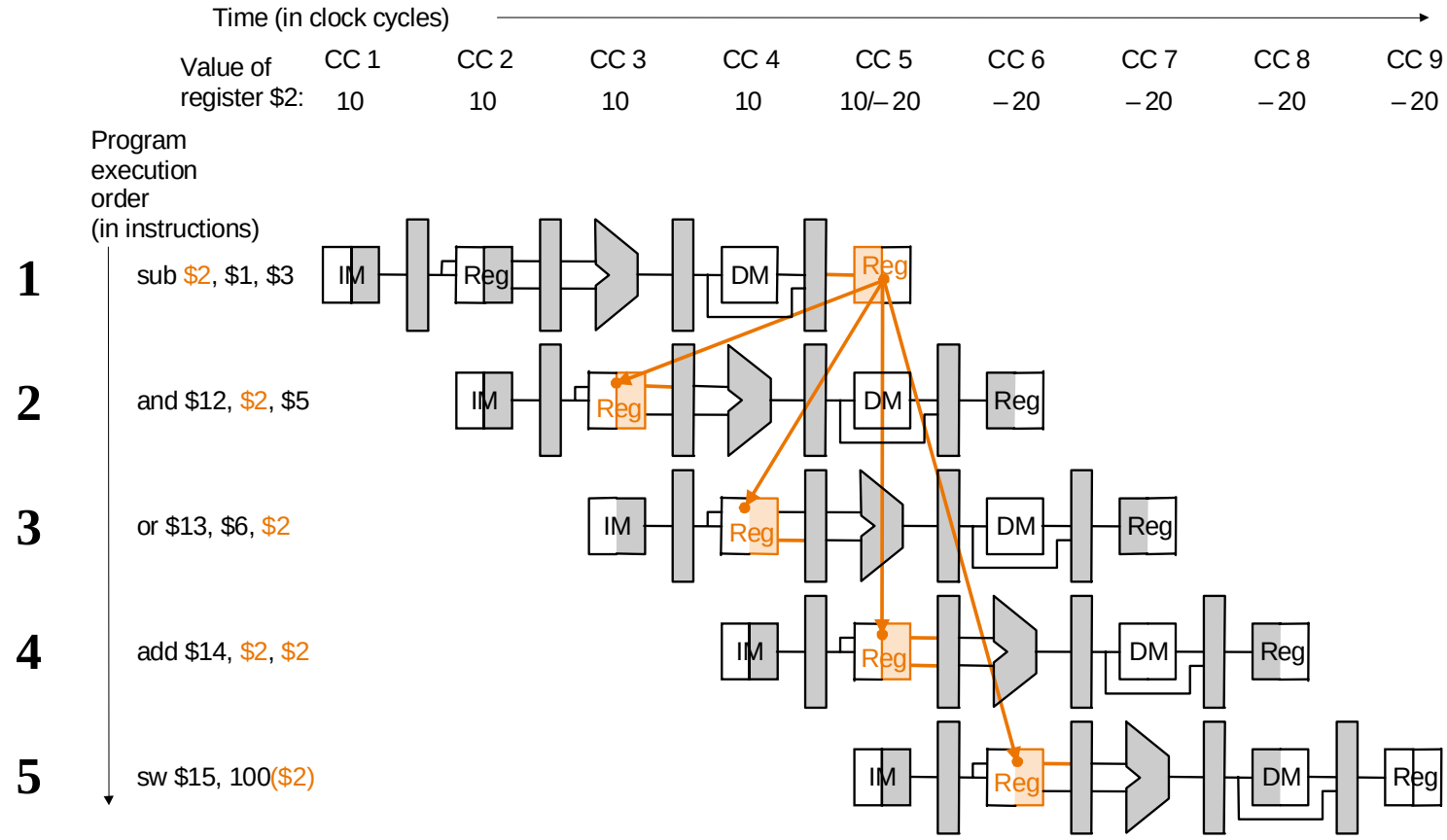
(As shown next)

i.e Correct operand data not ready yet when needed in EX cycle

Data Hazards Example

```
1 sub   $2, $1, $3
2 and   $12, $2, $5
3 or    $13, $6, $2
4 add   $14, $2, $2
5 sw    $15, 100($2)
```

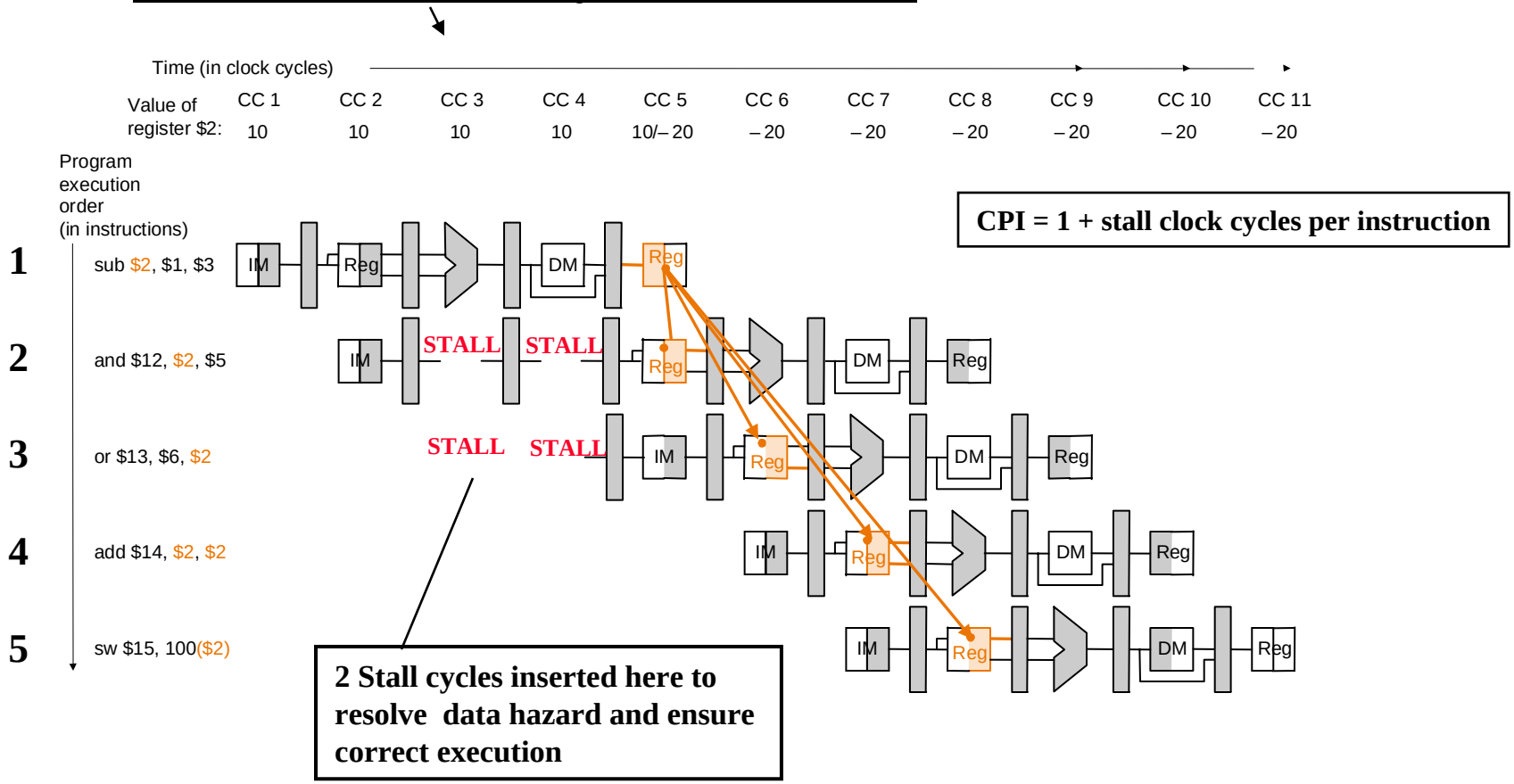
- **Problem with starting next instruction before first is finished**
  - **Data dependencies here that "go backward in time" create data hazards.**



Data Hazard Resolution: Stall Cycles

Stall the pipeline by a number of cycles.
The control unit must detect the need to insert stall cycles.

In this case two stall cycles are needed.



Above timing is for MIPS Pipeline Version #1

Data Hazard Resolution/Stall Reduction: Data Forwarding

- **Observation:**

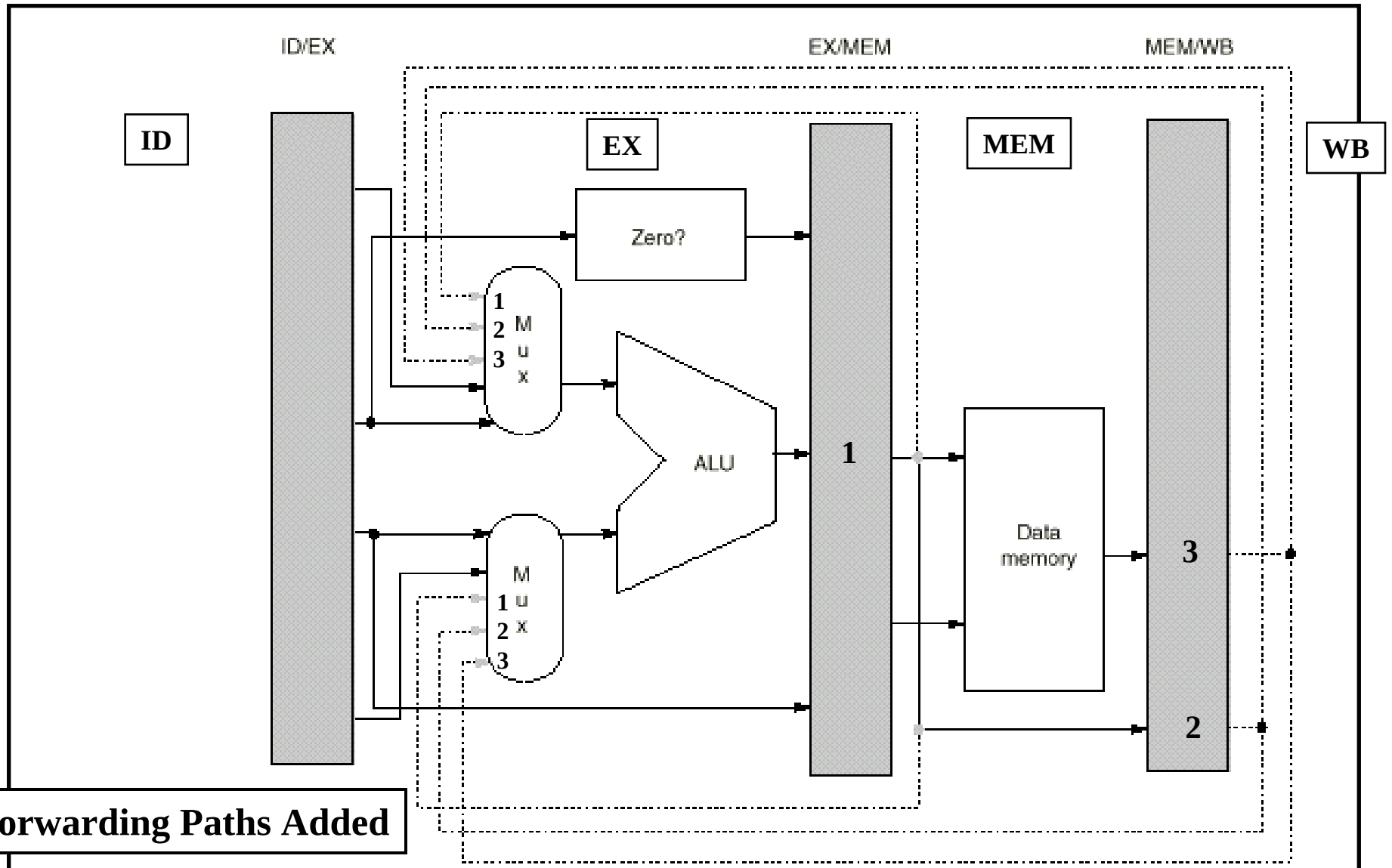
  Why not use temporary results produced by memory/ALU and not wait for them to be written back in the register bank.

- **Data Forwarding is a hardware-based technique (also called register bypassing or register short-circuiting) used to eliminate or minimize data hazard stalls that makes use of this observation.**

- **Using forwarding hardware, the result of an instruction (i.e data) is copied directly (i.e. forwarded) from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)**

Forwarding In MIPS Pipeline

- **The ALU result from the EX/MEM register may be forwarded or fed back to the ALU input latches as needed instead of the register operand value read in the ID stage.**
- **Similarly, the Data Memory Unit result from the MEM/WB register may be fed back to the ALU input latches as needed .**
- **If the forwarding hardware detects that a previous ALU operation is to write the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.**



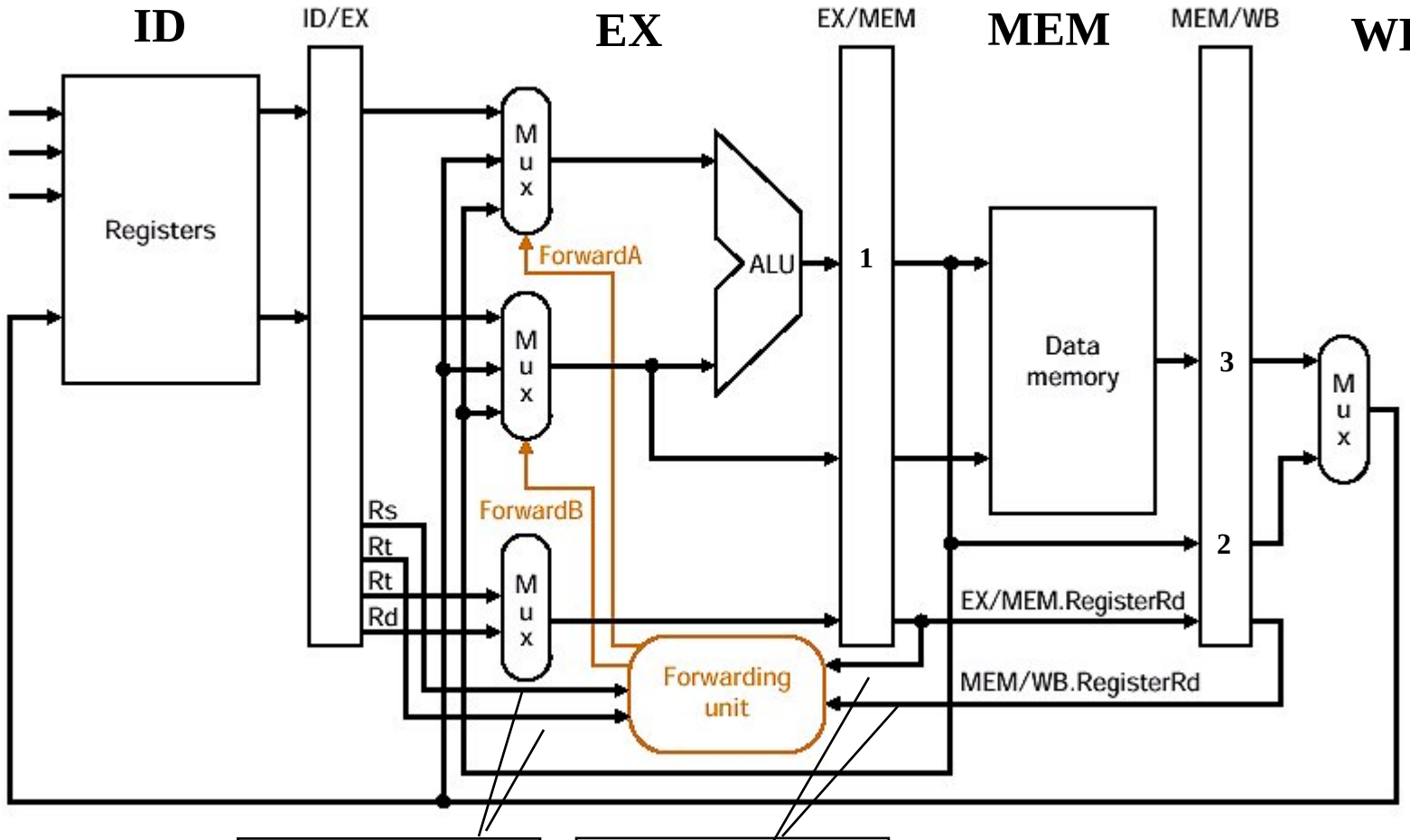
Forwarding Paths Added

Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs.

This diagram shows better forwarding baths than in textbook

Data Hazard Resolution: Forwarding

- **The forwarding unit compares <u>operand registers of the instruction in EX stage</u> with <u>destination registers of the previous two instructions in MEM and WB</u>**
- **If there is a match one or both operands will be obtained from forwarding paths bypassing the registers**

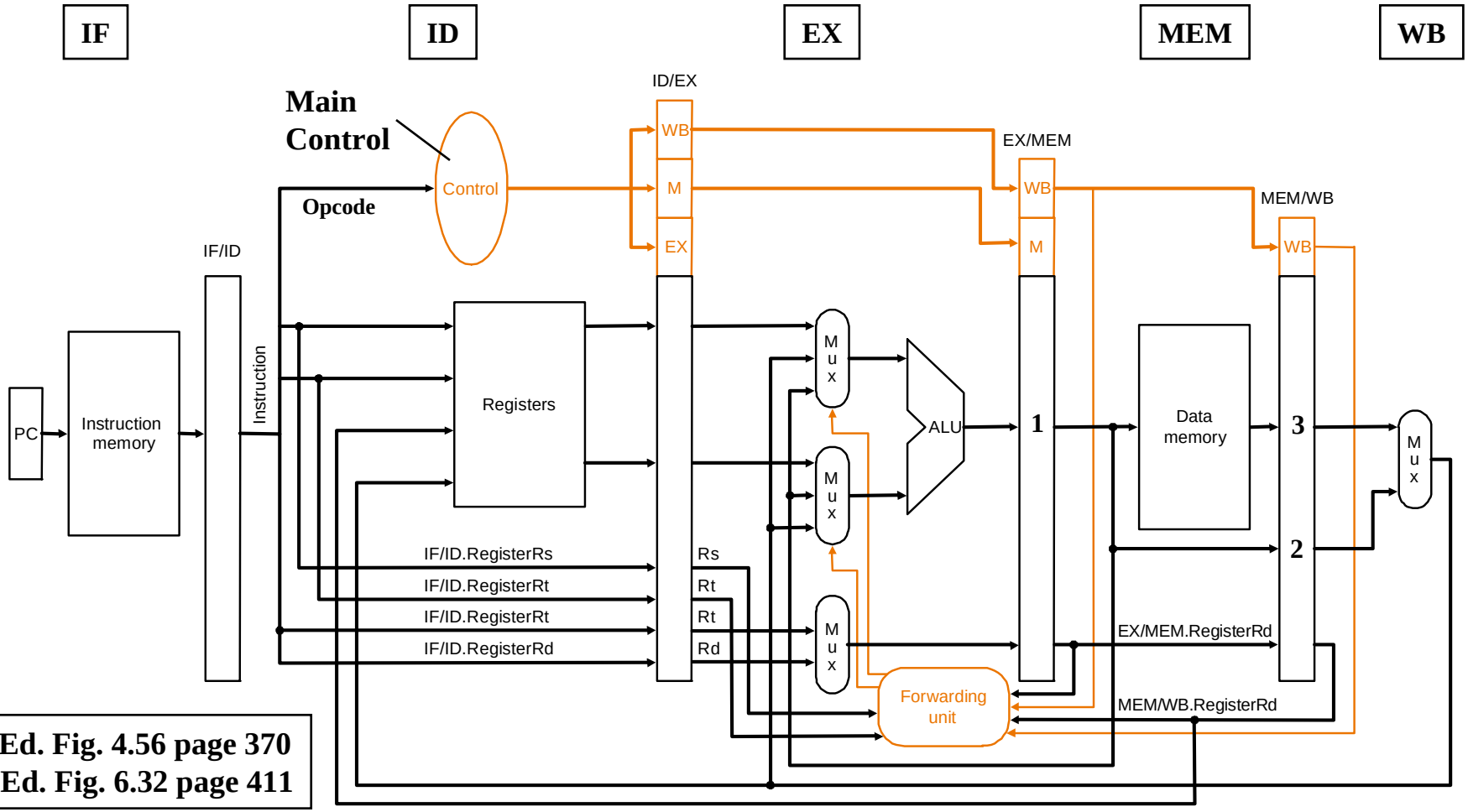


4th Ed. Fig. 4.54 page 368
3rd Ed. Fig. 6.30 page 409

Operand Register numbers of instruction in EX

Destination Register numbers of instructions in MEM and WB

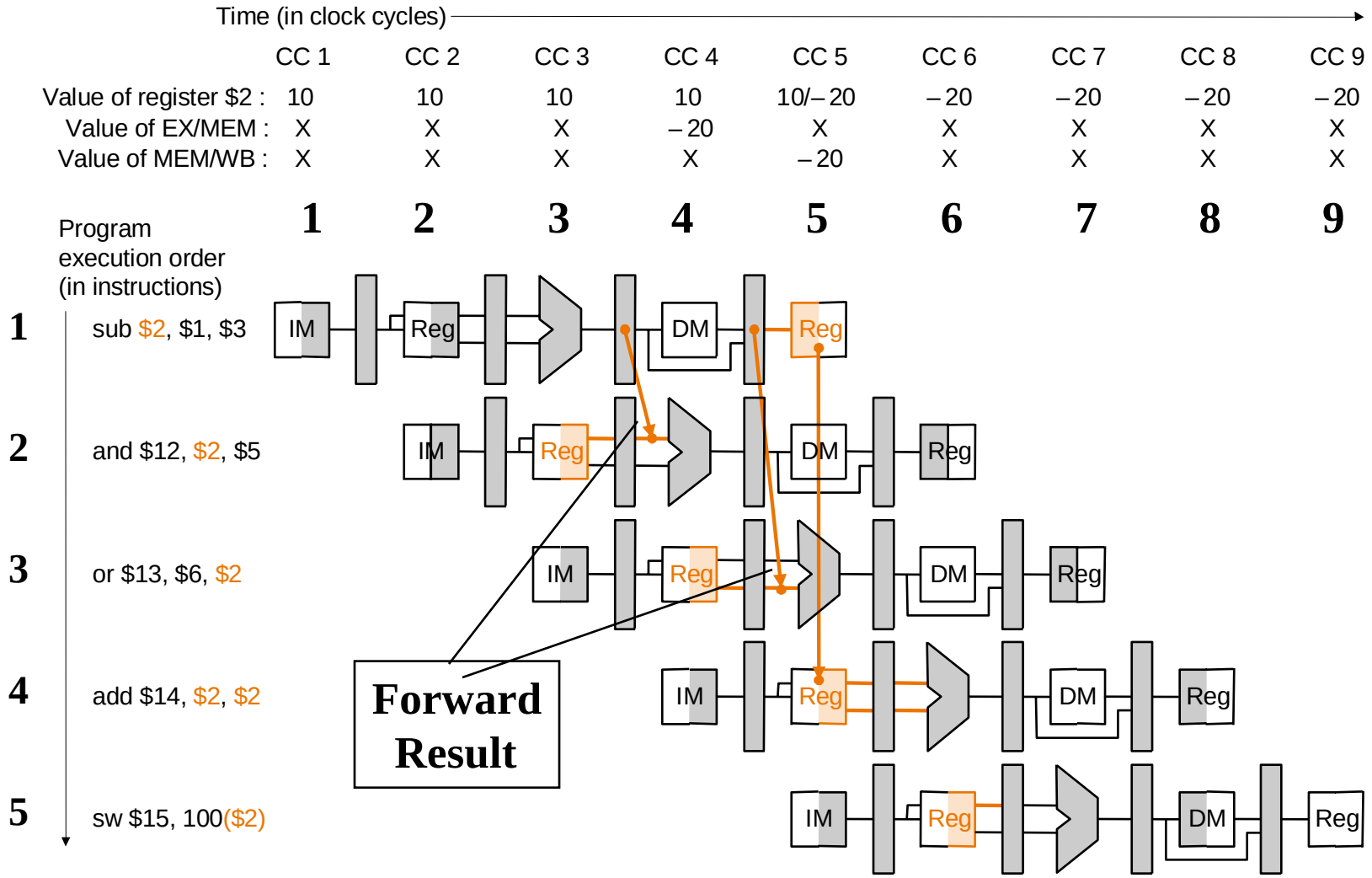
Pipelined Datapath With Forwarding



4th Ed. Fig. 4.56 page 370
3rd Ed. Fig. 6.32 page 411

- The forwarding unit compares operand registers of the instruction in EX stage with destination registers of the previous two instructions in MEM and WB
- If there is a match one or both operands will be obtained from forwarding paths bypassing the registers

Data Hazard Example With Forwarding

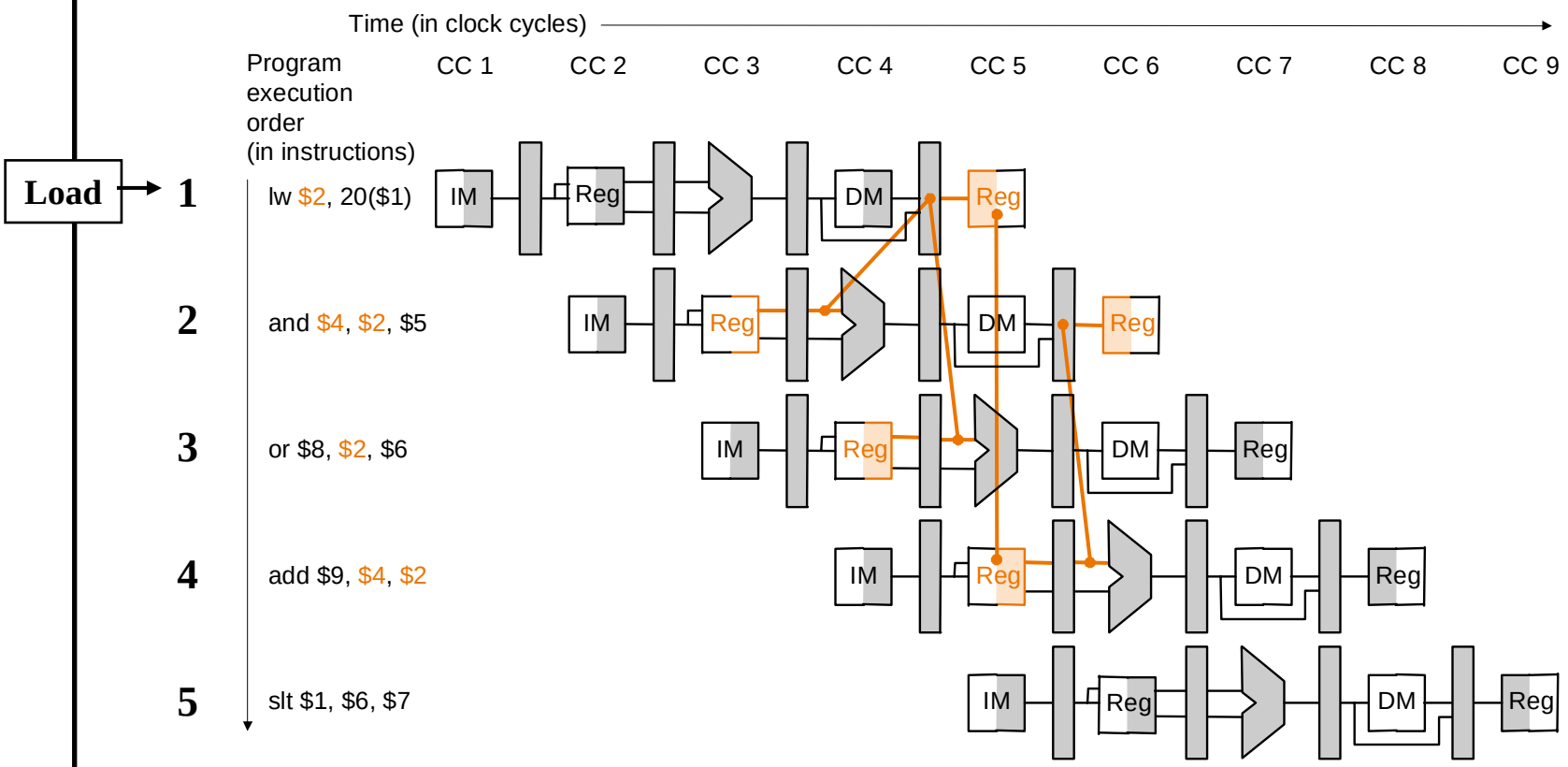


What registers numbers are being compared by the forwarding unit during cycle 5? What about in Cycle 6?

A Data Hazard Requiring A Stall

A load followed immediately by an R-type instruction that uses the loaded value

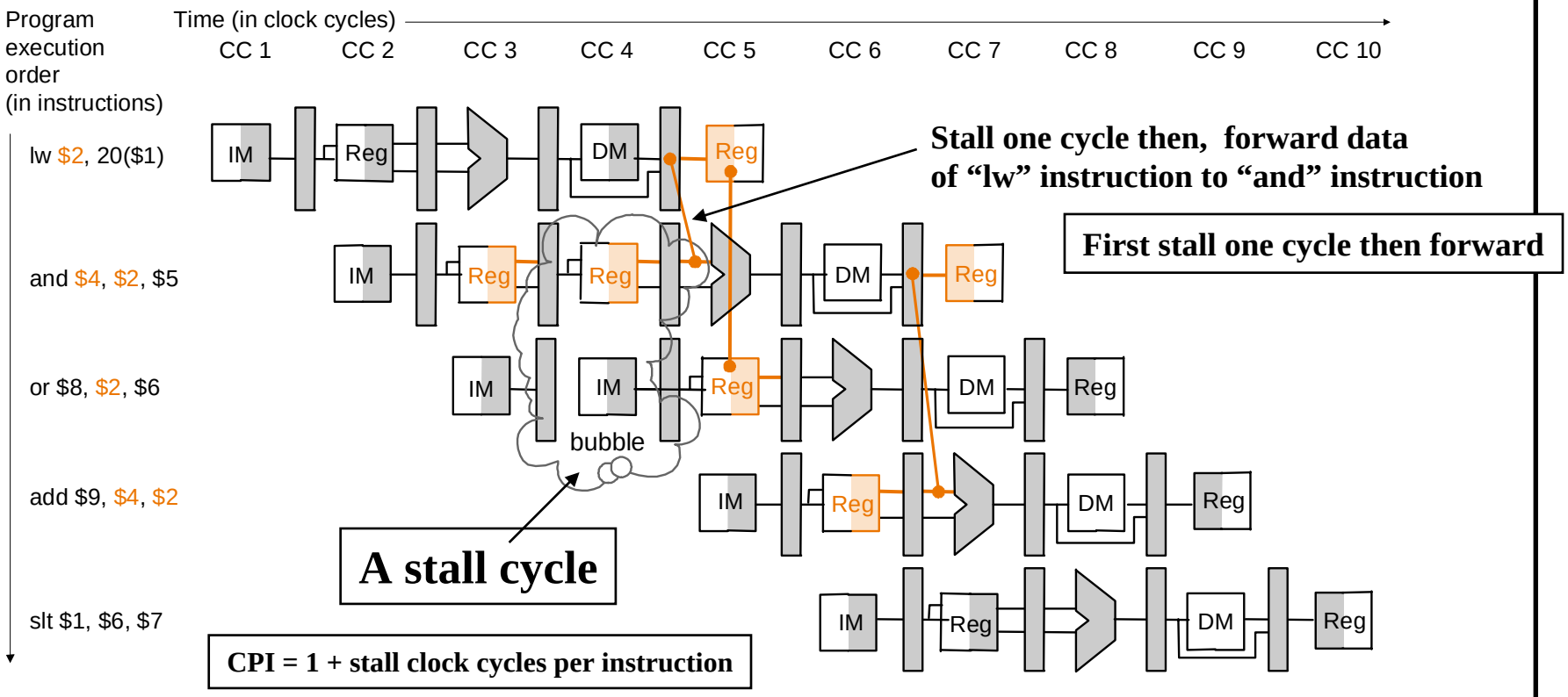
(or any other type of instruction that needs loaded value in EX stage)



Even with forwarding in place a stall cycle is needed (shown next)
This condition must be detected by hardware

A Data Hazard Requiring A Stall

A load followed immediately by an R-type instruction that uses the loaded value results in a single stall cycle even with forwarding as shown:



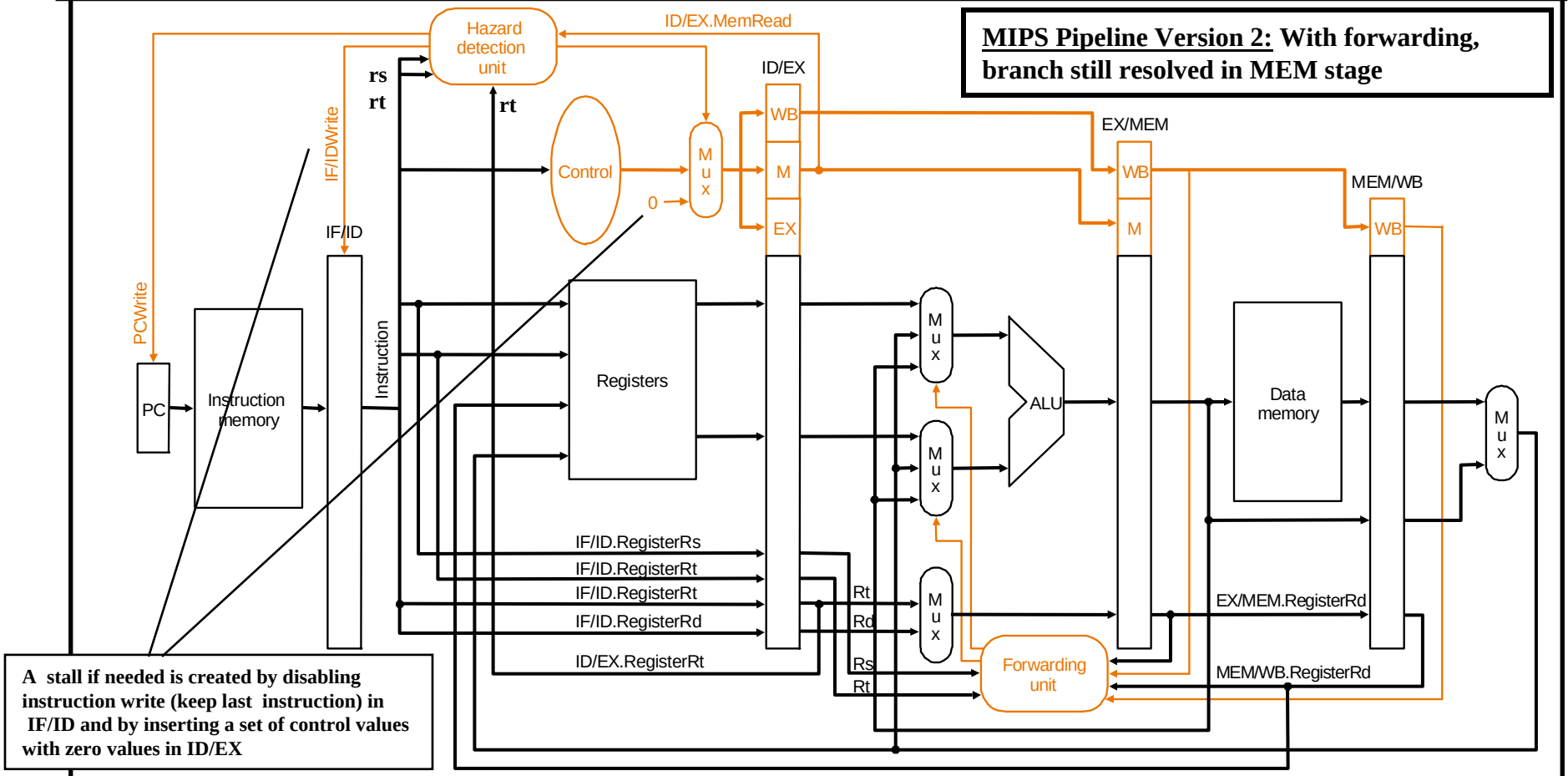
- **We can stall the pipeline by keeping all instructions following the "lw" instruction in the same pipeline stage for one cycle**

What is the hazard detection unit (shown next slide) doing during cycle 3?

Datapath With Hazard Detection Unit

A load followed by an instruction that uses the loaded value is detected by the hazard detection unit and a stall cycle is inserted.
The hazard detection unit checks if the instruction in the EX stage is a load by checking its MemRead control line value
If that instruction is a load it also checks if any of the operand registers of the instruction in the decode stage (ID) match the destination register of the load. In case of a match it inserts a stall cycle (delays decode and fetch by one cycle).

MIPS Pipeline Version 2: With forwarding, branch still resolved in MEM stage



A stall if needed is created by disabling instruction write (keep last instruction) in IF/ID and by inserting a set of control values with zero values in ID/EX

IF | **ID** | **EX** | **MEM** | **WB**

MIPS Pipeline Version #2

4th Edition Figure 4.60 page 375
3rd Edition Figure 6.36 page 416

Situation	Example code sequence	Action

No dependence	`LW` **R1**`,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R6,R7`
	`OR R9,R6,R7`
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

Dependence requiring stall	`LW` **R1**`,45(R2)`
	`ADD R5,`**R1**`,R7`
	`SUB R8,R6,R7`
	`OR R9,R6,R7`
		Comparators detect the use of R1 in the `ADD` and stall the `ADD` (and `SUB` and `OR`) before the `ADD` begins EX. **Stall + Forward**

Dependence overcome by forwarding	`LW` **R1**`,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,`**R1**`,R7`
	`OR R9,R6,R7`
		Comparators detect use of R1 in `SUB` and forward result of load to ALU in time for `SUB` to begin EX. **Forward**

Dependence with accesses in order	`LW` **R1**`,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R6,R7`
	`OR R9,`**R1**`,R7`
		No action required because the read of R1 by `OR` occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions.

Hazard Detection Unit Operation

Compiler Instruction Scheduling (Re-ordering) Example

- **Reorder the instructions to avoid as many pipeline stalls as possible:**

Stall →

```
lw      $15, 0($2)
lw      $16, 4($2)
add     $14, $5, $16
sw      $16, 4($2)
```

Original Code

- **The data hazard occurs on register $16 between the second lw and the add instruction resulting in a stall cycle even with forwarding**
- **With forwarding we (or the compiler) need to find only one independent instruction to place between them, swapping the lw instructions works:**

i.e pipeline version #2

i.e pipeline version #1

No Stalls

```
lw      $16, 4($2)
lw      $15, 0($2)
add     $14, $5, $16
sw      $16, 4($2)
```

Scheduled Code

With Forwarding

- **Without forwarding we need two independent instructions to place between them, so in addition a nop is added (or the hardware will insert a stall).**

Or stall cycle →

```
lw      $16, 4($2)
lw      $15, 0($2)
nop
add     $14, $5, $16
sw      $16, 4($2)
```

Scheduled Code

No Forwarding

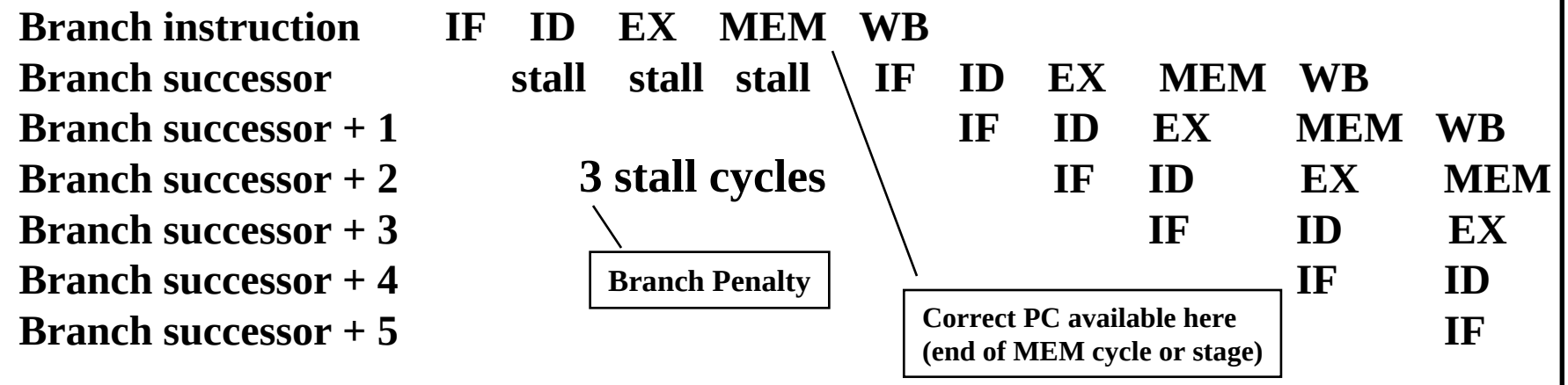
Control Hazards

- **When a conditional branch is executed it may change the PC (when taken) and, without any special measures, leads to stalling the pipeline for a number of cycles until the branch condition is known and PC is updated (branch is resolved).**

Here end of stage 4 (MEM) – For Pipeline Versions 1 or 2

i.e version 2 – **Otherwise the PC may not be correct when needed in IF**

- **In current MIPS pipeline, the conditional branch is resolved in stage 4 (MEM stage) resulting in three stall cycles as shown below:**



Assuming we stall or flush the pipeline on a branch instruction:

Three clock cycles are wasted for every branch for current MIPS pipeline

Branch Penalty = stage number where branch is resolved - 1
here Branch Penalty = 4 - 1 = 3 Cycles

i.e Correct PC is not available when needed in IF

Basic Branch Handling in Pipelines

1 **One scheme discussed earlier is to <u>always stall</u> (*flush or freeze)* the pipeline <u>whenever a conditional branch is decoded</u> by holding or deleting any instructions in the pipeline until the branch destination is known (zero pipeline registers, control lines).**

Pipeline stall cycles from branches = frequency of branches X branch penalty

- **Ex: Branch frequency = 20% branch penalty = 3 cycles**

  CPI = 1 + .2 x 3 = 1.6

CPI = 1 + stall clock cycles per instruction

2 **Another method is to <u>assume or *predict that the branch is not taken*</u> where the state of the machine is not changed until the branch outcome is definitely known. Execution here continues with the next (PC+4) instruction; *stall occurs here when the branch is taken.***

Pipeline stall cycles from branches = frequency of taken branches X branch penalty

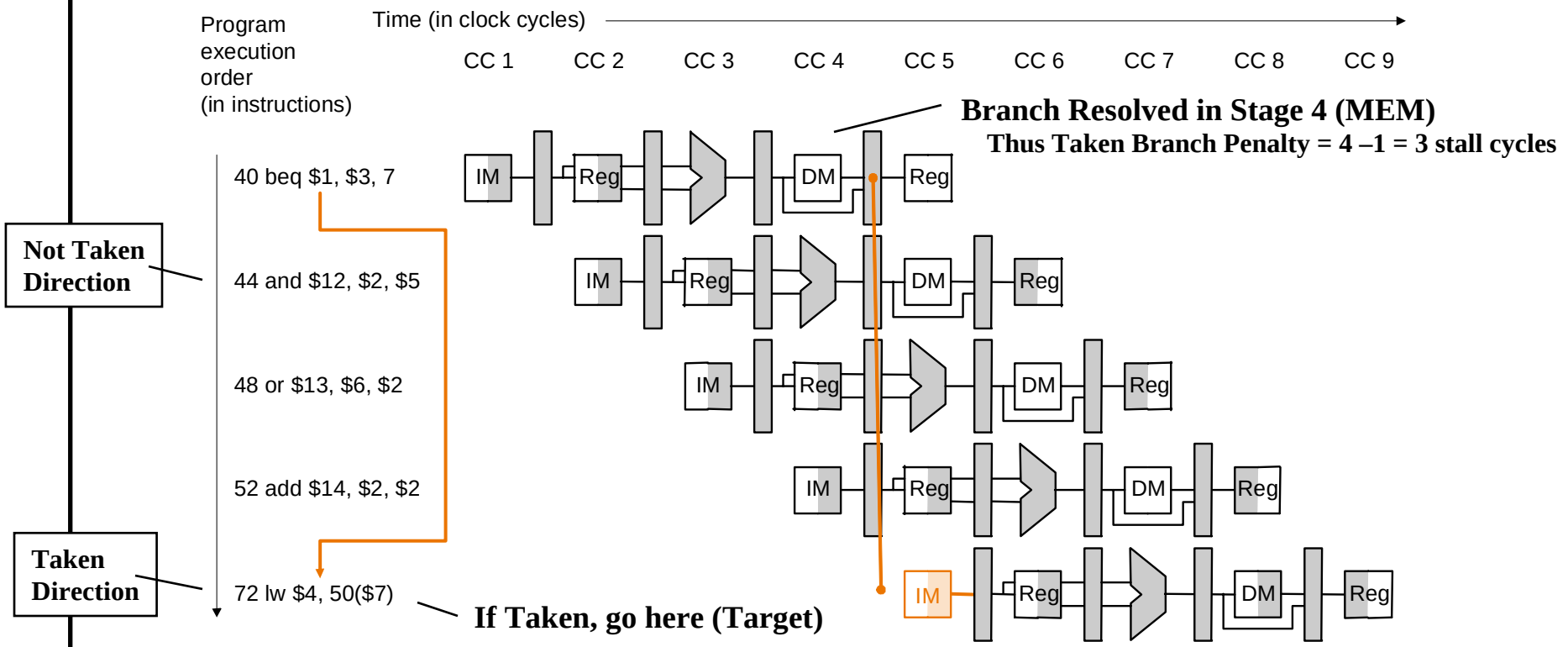
- **Ex: Branch frequency = 20% of which 45% are taken branch penalty = 3 cycles**

  CPI = 1 + .2 x .45 x 3 = 1.27

CPI = 1 + Average Stalls Per Instruction

Control Hazards: Example

- **Three other instructions are in the pipeline before branch instruction target decision is made when BEQ is in MEM stage.**



- **In the above diagram, we are assuming "branch not taken"**
  - **Need to add hardware for flushing the three following instructions if we are wrong losing three cycles when the branch is taken.**

i.e. Taken Branch Penalty

i.e the branch was resolved as taken in MEM stage

Hardware Reduction of Branch Stall Cycles

i.e. pipeline redesign

MIPS Pipeline Version #3

Pipeline hardware measures to reduce taken branch stall cycles:

1- Find out whether a branch is taken earlier in the pipeline.

2- Compute the taken PC earlier in the pipeline.

In MIPS: **i.e Resolve the branch in an early stage in the pipeline**

- In MIPS branch instructions BEQ, BNE, test a register for equality to zero.
- This can be completed in the ID cycle by moving the zero test into that cycle (ID).
- Both PCs (taken and not taken) must be computed early.
- Requires an additional adder in ID because the current ALU is not useable until EX cycle.
- This results in just a single cycle stall on taken branches.
  - **Branch Penalty when taken = stage resolved - 1 = 2 - 1 = 1**

As opposed branch penalty = 3 cycles before (pipelene versions 1 and 2)

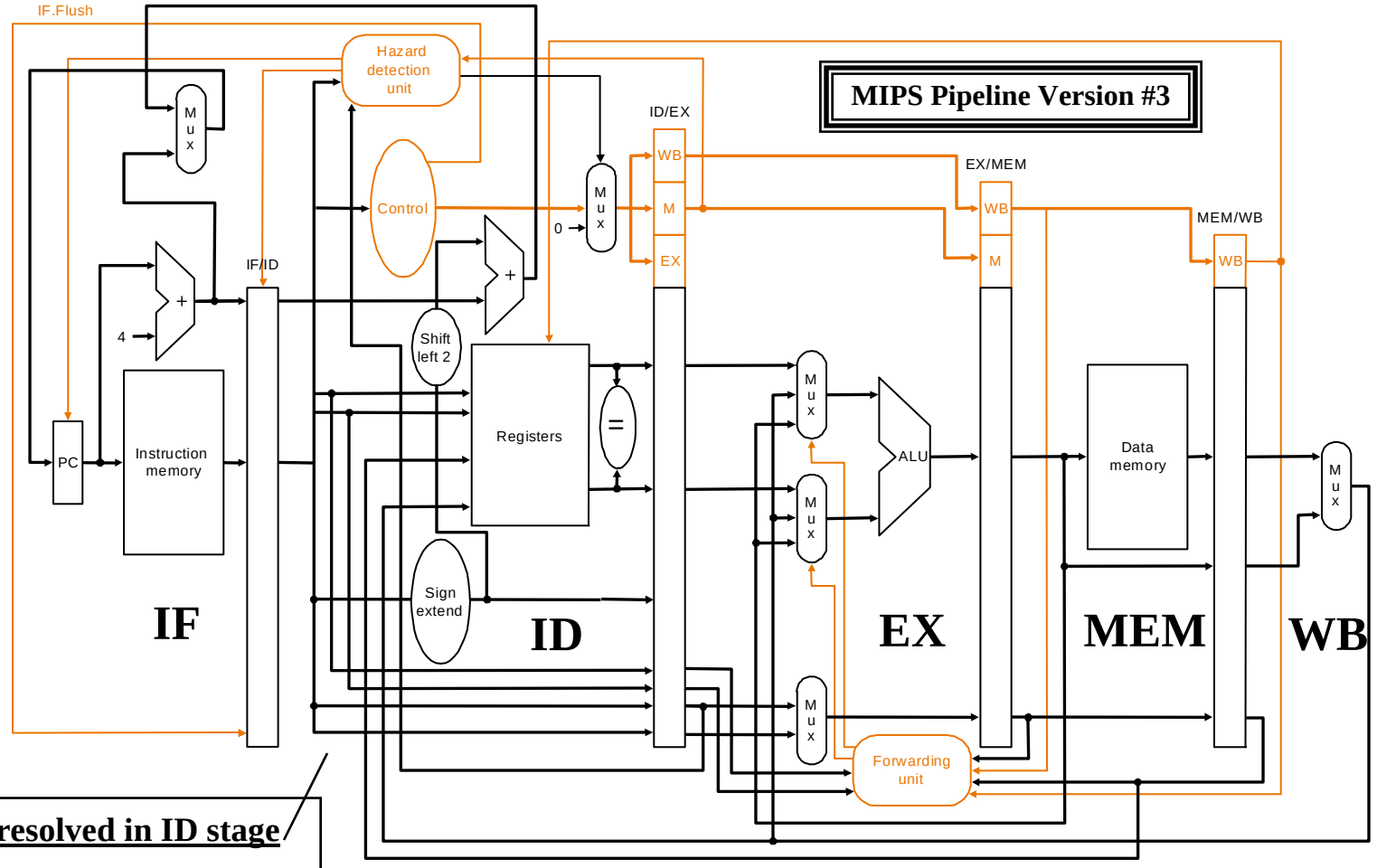
MIPS Pipeline Version 3: With forwarding, branch resolved in ID stage

Reducing Delay (Penalty) of Taken Branches

- So far: Next PC of a branch known or resolved in MEM stage: Costs three lost cycles if the branch is taken.
- If next PC of a branch is known or resolved in EX stage, one cycle is saved.
- Branch address calculation can be moved to ID stage (stage 2) using a register comparator, costing only one cycle if branch is taken as shown below. Branch Penalty = stage 2 -1 = 1 cycle

MIPS Pipeline Version #3

MIPS Pipeline Version 3:
With forwarding, branch resolved in ID stage



MIPS Pipeline Version #3

Here the branch is resolved in ID stage (stage 2)
Thus branch penalty if taken = 2 - 1 = 1 cycle

4th Edition Figure 4.65 page 384
3rd Edition Figure 6.41 page 427

Pipeline Performance Example

- **Assume the following MIPS instruction mix:**

Type	Frequency	
Arith/Logic	40%	
Load	30%	of which 25% are followed immediately by an instruction using the loaded value — 1 stall
Store	10%	
branch	20%	of which 45% are taken — 1 stall

- **What is the resulting CPI for the pipelined MIPS with forwarding and branch address calculation in ID stage when using the branch not-taken scheme?**

i.e Version 3

Branch Penalty = 1 cycle

- **CPI = Ideal CPI + Pipeline stall clock cycles per instruction**

= 1 + stalls by loads + stalls by branches

= 1 + .3 x .25 x 1 + .2 x .45 x 1

= 1 + .075 + .09

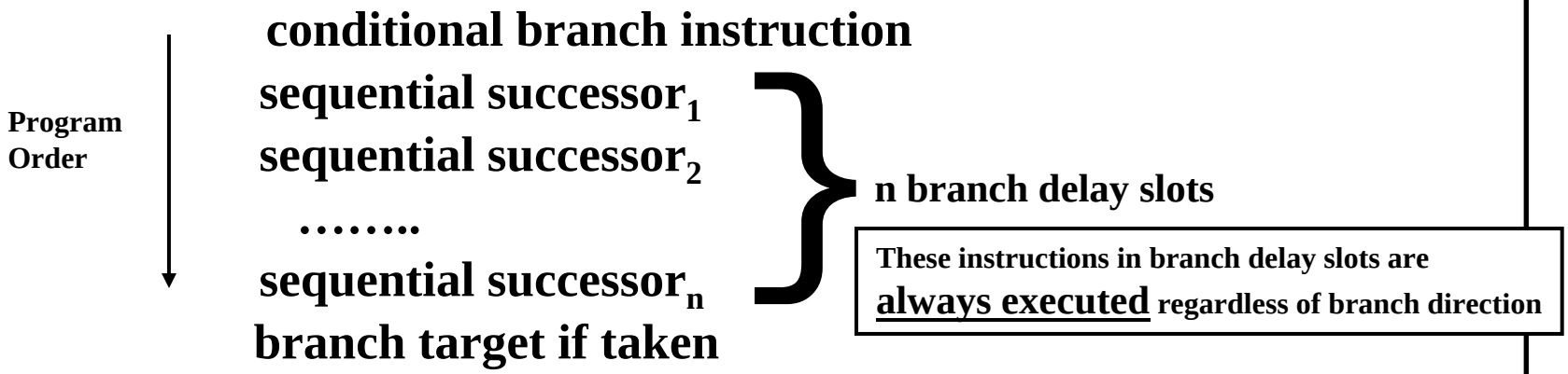
= 1.165

CPI = 1 + Average Stalls Per Instruction

ISA Reduction of Branch Penalties: Delayed Branch (Action)

i.e. ISA Support Needed

- When delayed branch is used in an ISA, the branch action is delayed by *n* cycles (or instructions), following this execution pattern:



- The sequential successor instructions are said to be in the branch delay slots. These instructions are executed whether or not the branch is taken.
- In Practice, all ISAs that utilize delayed branching including MIPS utilize a single instruction branch delay slot. (All RISC ISAs)
  - The job of the compiler is to make the successor instruction in the delay slot a valid and useful instruction.

Delayed Branch Example

(Single Branch Delay slot, instruction or cycle used here)

(All RISC ISAs)

Not Taken Branch (no stall)

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Taken Branch (no stall)

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

The behavior of a delayed branch is the same whether or not the branch is taken.

The instruction in the branch delay slot is executed whether the branch is taken or not

Here, assuming the MIPS pipeline (version 3) with reduced branch penalty = 1

Delayed Branch-delay Slot Scheduling Strategies

The branch-delay slot instruction can be chosen from three cases:

A **An independent instruction from before the branch:**

Always improves performance when used. The branch must not depend on the rescheduled instruction.

Most Common

e.g From Body of a loop

B **An instruction from the target of the branch:**

Improves performance if the branch is taken and may require instruction duplication. This instruction must be safe to execute if the branch is not taken.

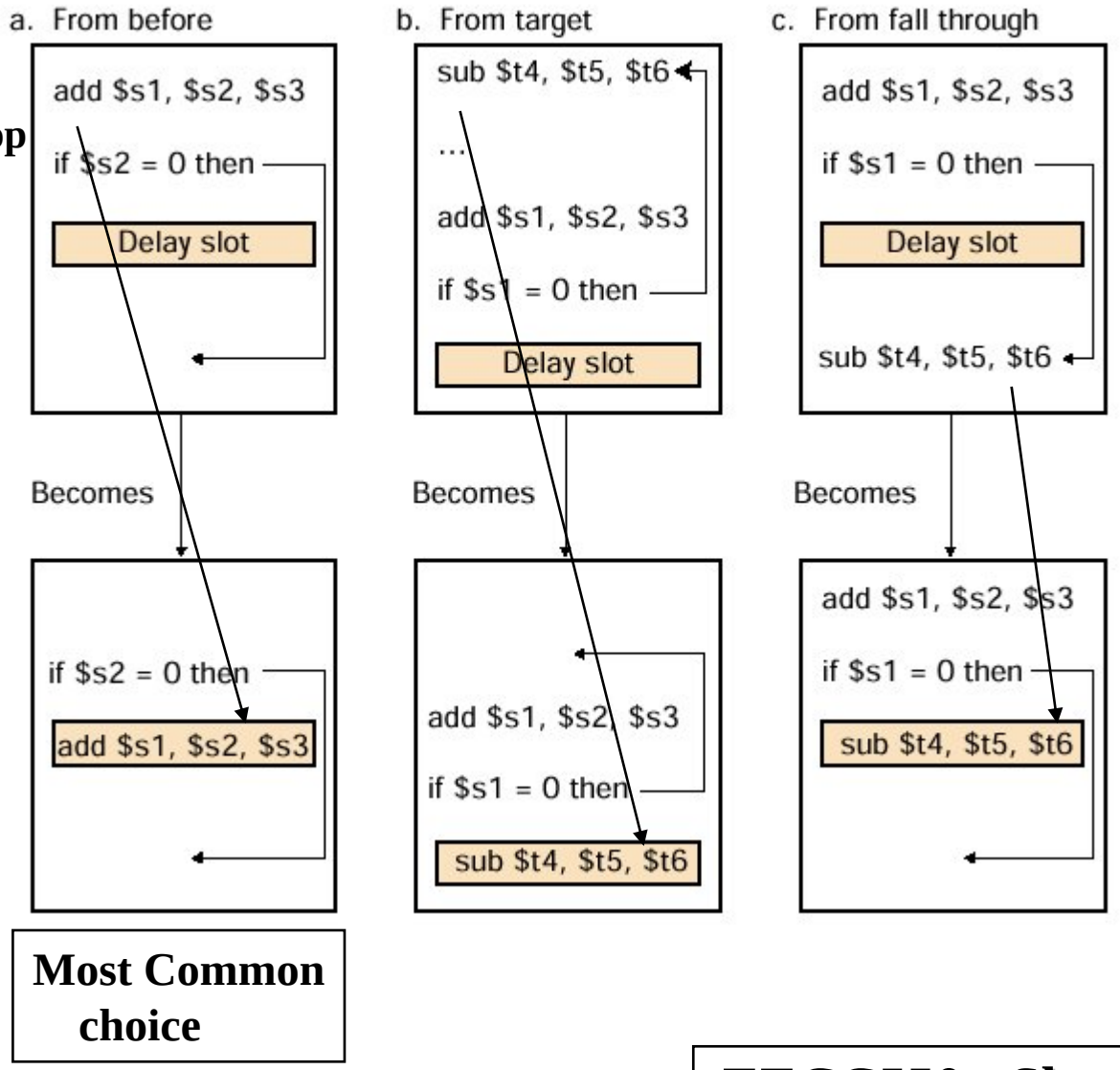
Hard to Find

C **An instruction from the fall through instruction stream:**

Improves performance when the branch is not taken. The instruction must be safe to execute when the branch is taken.

Scheduling The Branch Delay Slot

Example:
From the body of a loop



Most Common choice

Compiler Instruction Scheduling Example With Branch Delay Slot

To reduce or eliminate stalls

- Schedule the following MIPS code for the pipelined MIPS CPU with forwarding and reduced branch delay using a single branch delay slot to minimize stall cycles:

i.e MIPS Pipeline Version 3

```
loop:   lw $1,0($2)          #  $1 array element
        add $1, $1, $3       # add constant in $3
        sw $1,0($2)          # store result array element
        addi $2, $2, -4      # decrement address by 4
        bne $2, $4, loop     # branch if $2 != $4
```

- Assuming the initial value of $2 = $4 + 40

  (i.e it loops 10 times)

  – What is the CPI and total number of cycles needed to run the code with and without scheduling?

For MIPS Pipeline Version 3

Compiler Instruction Scheduling Example (With Branch Delay Slot)

- **Without compiler scheduling**

```
loop:  lw $1,0($2)
       Stall
       add $1, $1, $3
       sw $1,0($2)
       addi $2, $2, -4
       Stall
       bne $2, $4, loop
       Stall (or NOP)
```

Three Stalls Per Iteration

Needed because new value of $2 is not produced yet

Ignoring the initial 4 cycles to fill the pipeline:

Each iteration takes = 8 cycles

CPI = 8/5 = 1.6

Total cycles = 8 x 10 = 80 cycles

- **With compiler scheduling:**

```
loop:  lw $1,0($2)
       addi $2, $2, -4
       add $1, $1, $3
       bne $2, $4, loop
       sw $1, 4($2)
```

No Stalls

Move between lw add

Move to branch delay slot

Adjust address offset

Ignoring the initial 4 cycles to fill the pipeline:

Each iteration takes = 5 cycles

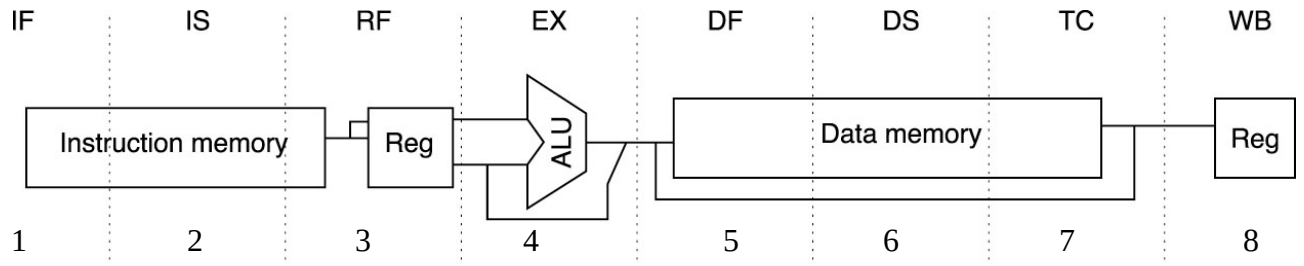
CPI = 5/5 = 1

Total cycles = 5 x 10 = 50 cycles

Speedup = 80/ 50 = 1.6

The MIPS R4000 Integer Pipeline

- **Implements MIPS64 but uses an 8-stage pipeline instead of the classic 5-stage pipeline to achieve a higher clock speed.**



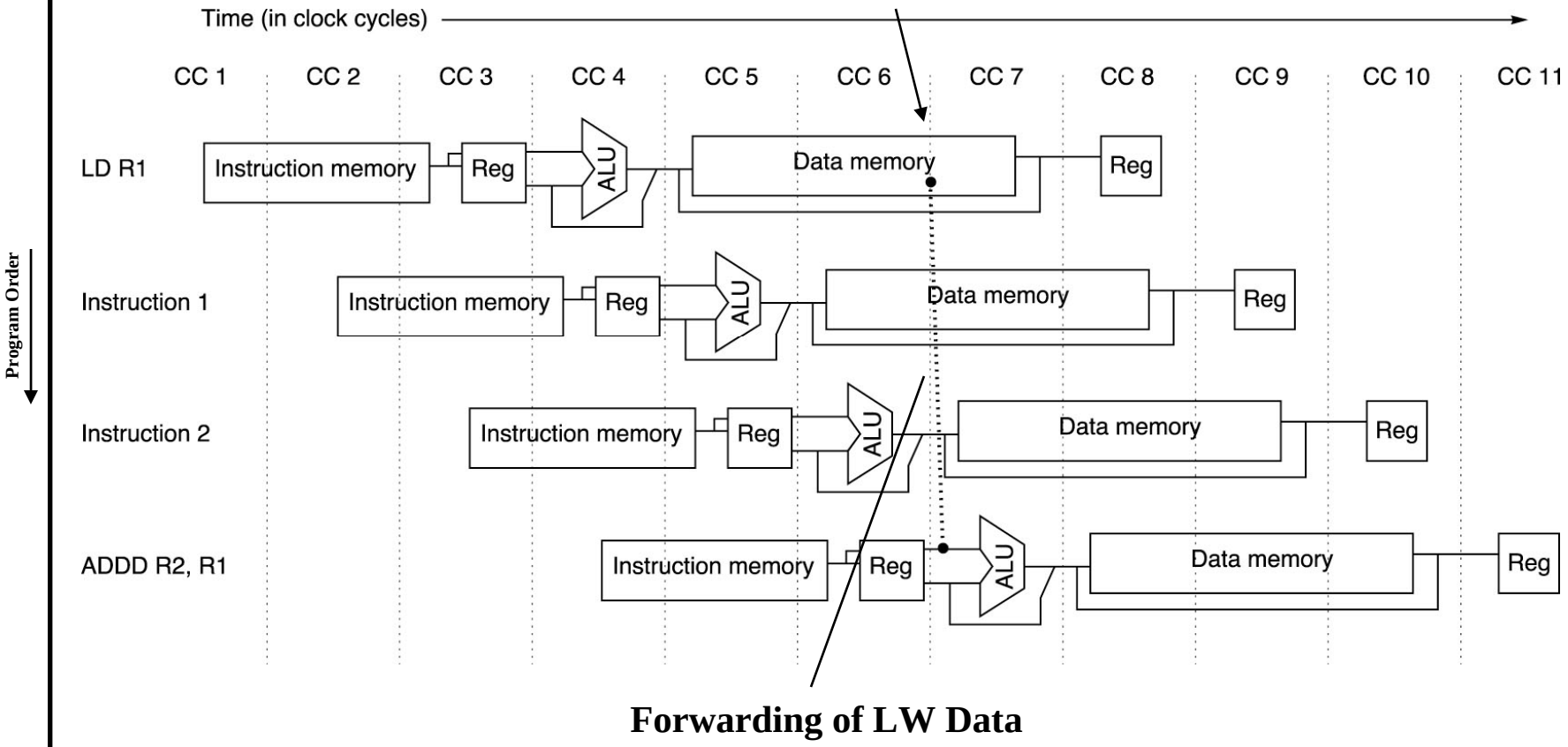
- **Pipeline Stages:**

Branch resolved here in stage 4 Thus branch penalty = 4 -1 = 3 cycles

  - **IF: First half of instruction fetch. Start instruction cache access.**
  - **IS: Second half of instruction fetch. Complete instruction cache access.**
  - **RF: Instruction decode and register fetch, hazard checking.**
  - **EX: Execution including branch-target and condition evaluation.**
  - **DF: Data fetch, first half of data cache access. Data available if a hit.**
  - **DS: Second half of data fetch access. Complete data cache access. Data available if a cache hit**
  - **TC: Tag check, determine data cache access hit.**
  - **WB: Write back for loads and register-register operations.**
  - **Branch resolved in stage 4. Branch Penalty = 3 cycles if taken (2 with branch delay slot)**

MIPS R4000 Example

T = I x CPI x C



- **Even with forwarding the deeper pipeline leads to a 2-cycle load delay (2 stall cycles).**

As opposed to 1-cycle in classic 5-stage pipeline

Thus: Deeper Pipelines = More Stall Cycles

T = I x CPI x C