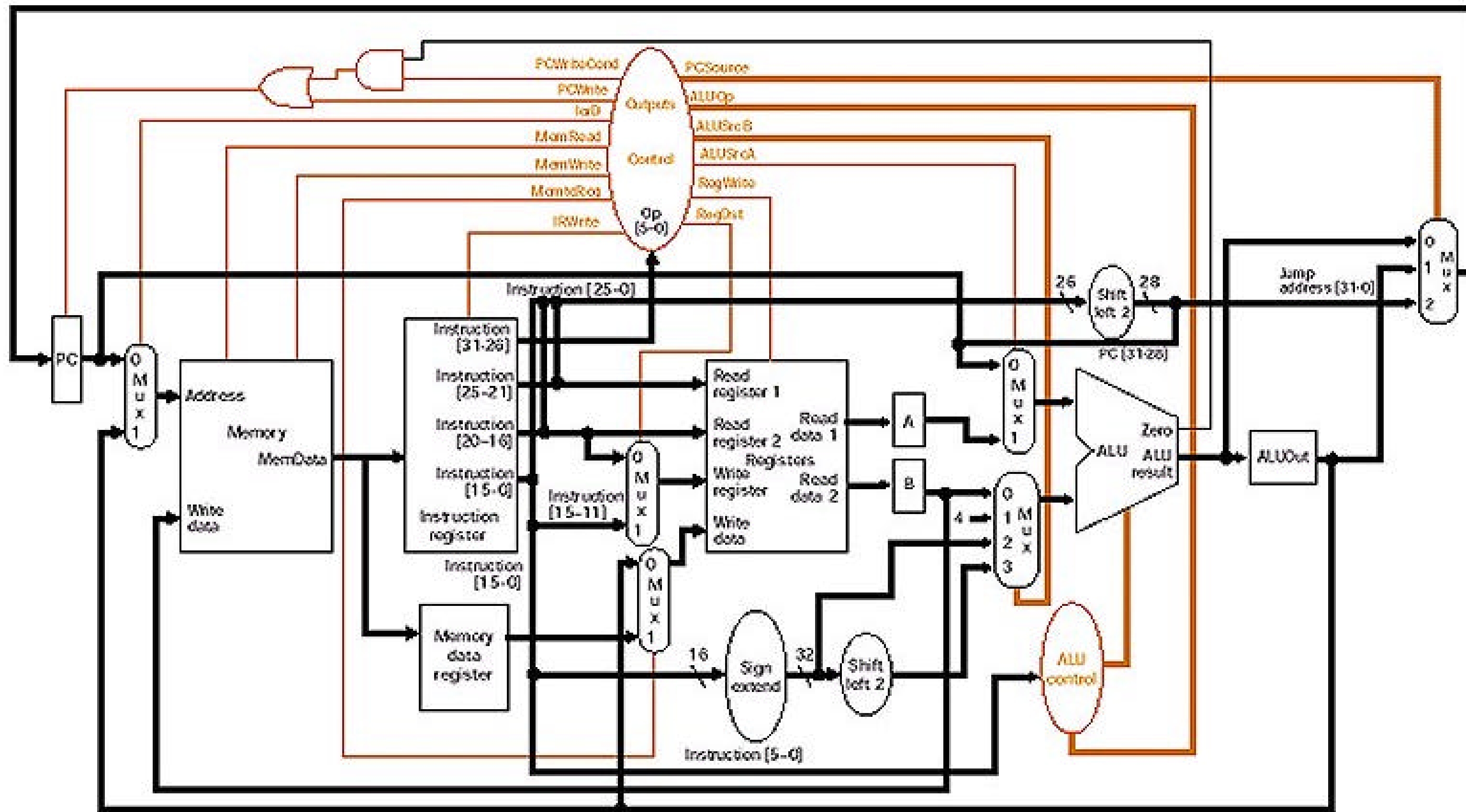
What do we have so far?
Multi-Cycle Datapath (Textbook Version)



CPI: R-Type = 4, Load = 5, Store 4, Branch = 3

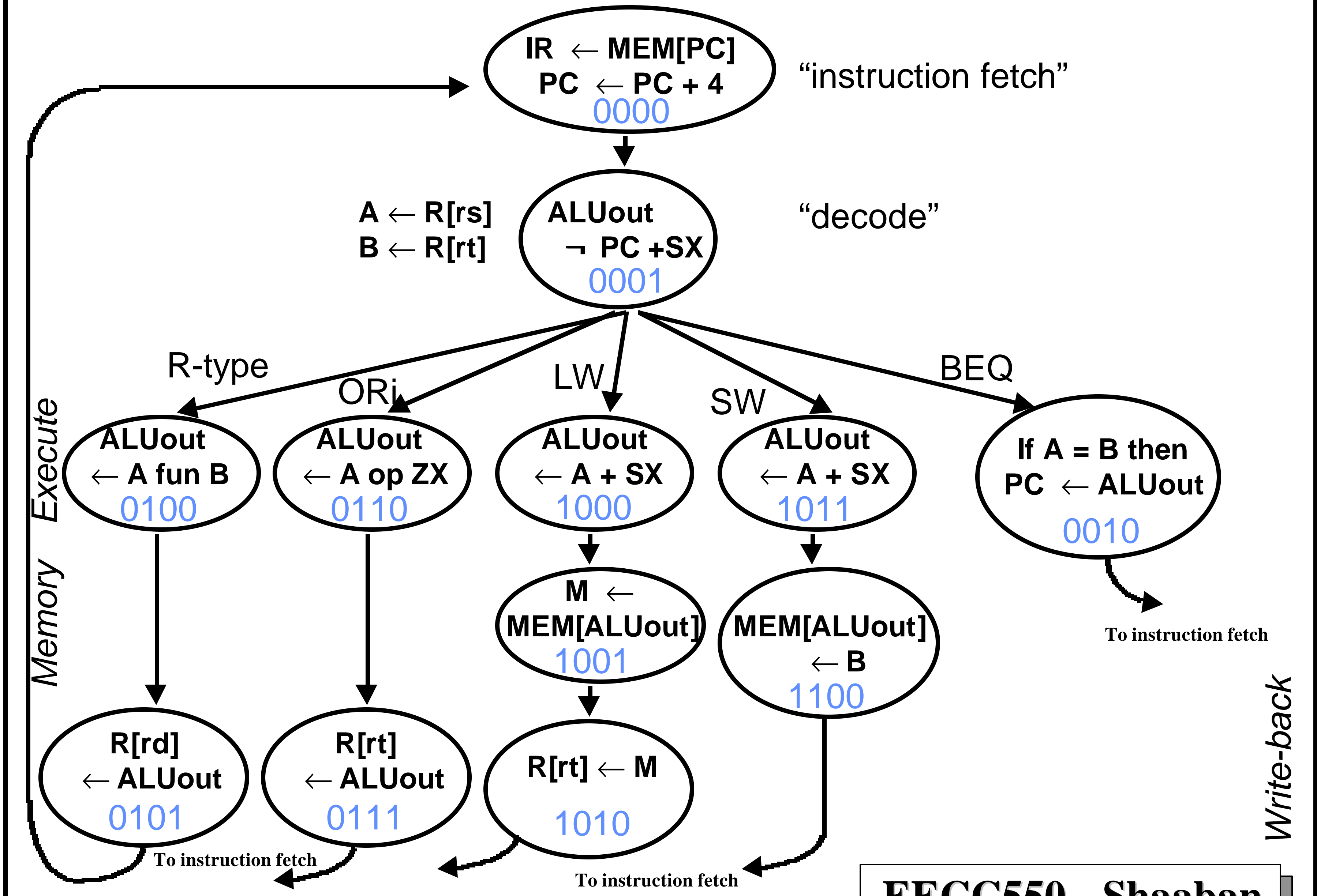
Only one instruction being processed in datapath

How to lower CPI further?

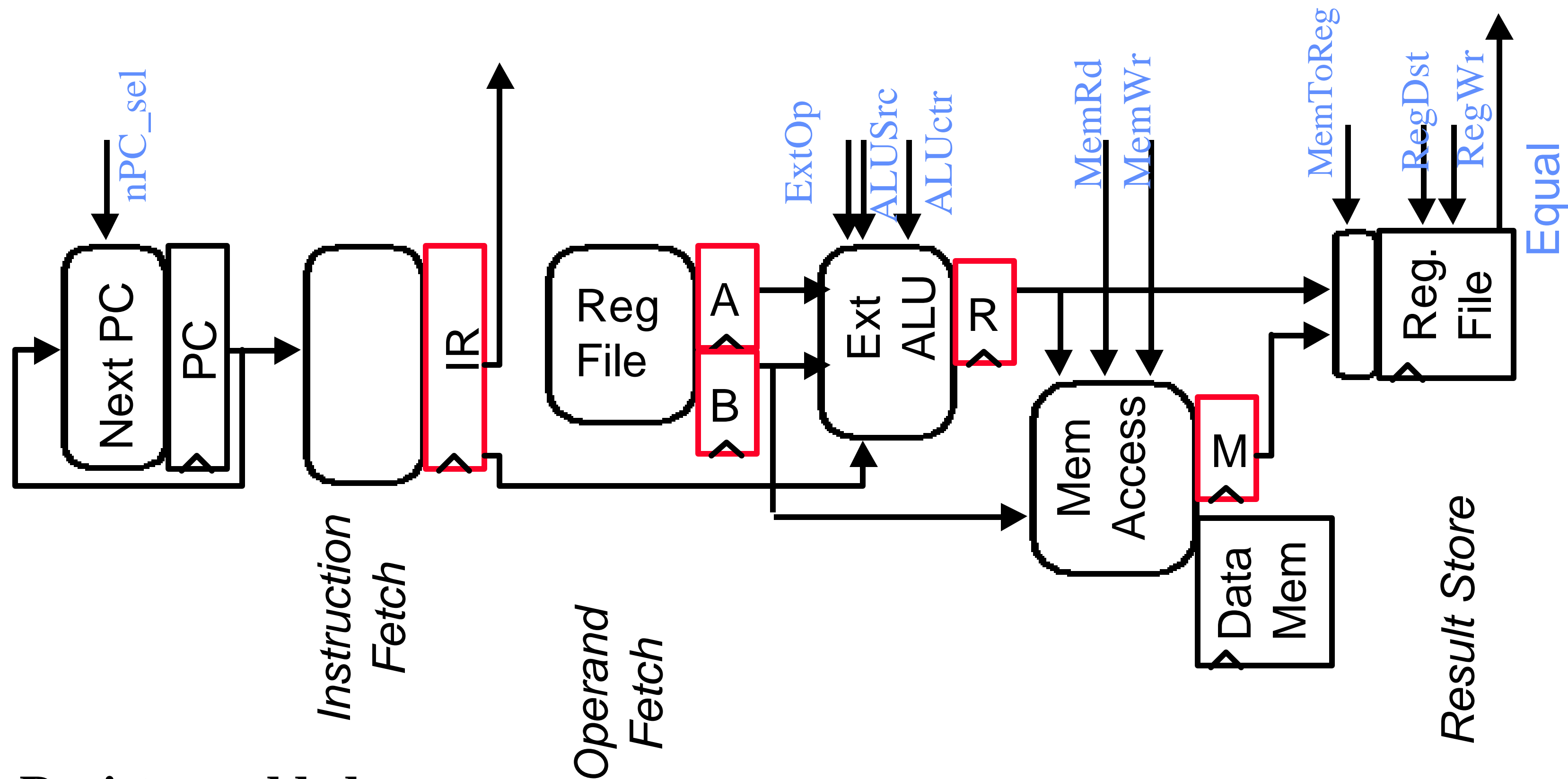
Operations In Each Cycle

		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] 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 + B	ALUout ¬ A OR ZeroExt[imm16]	ALUout ¬ A + SignEx(Im16)	ALUout ¬ A + SignEx(Im16)	If Equal = 1 PC ¬ ALUout
MEM	Memory			M ¬ Mem[ALUout]	Mem[ALUout] ¬ B	
WB	Write Back	R[rd] ¬ ALUout	R[rt] ¬ ALUout	R[rd] ¬ Mem		

Finite State Machine (FSM) Specification



Multi-cycle Datapath (Our Version)



Registers added:

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 In Each Cycle

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

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).**

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

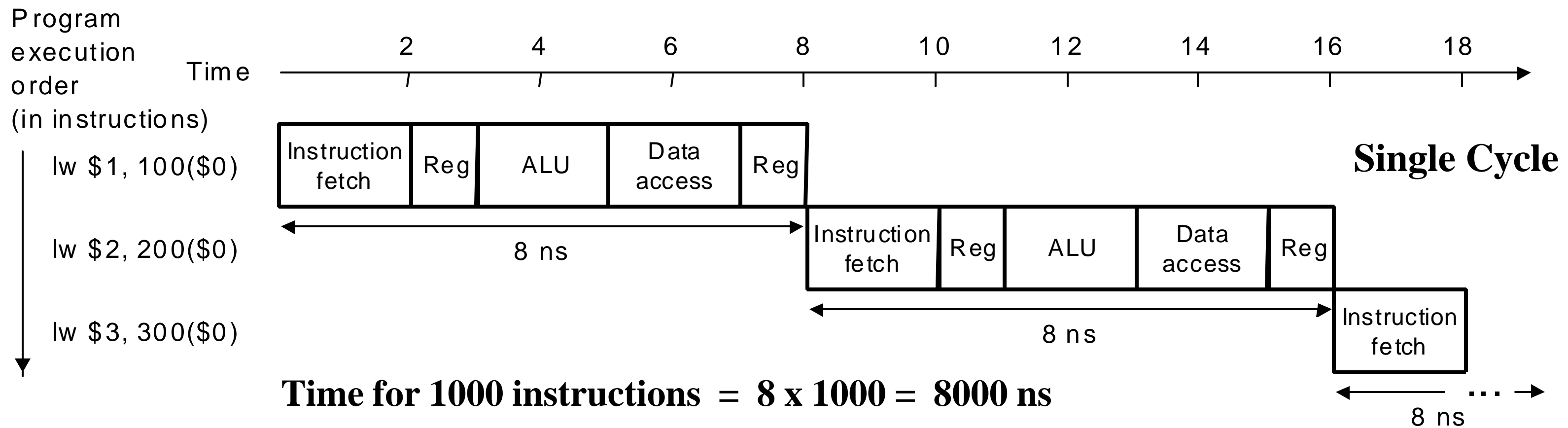
Better than CPI = 5 if all instructions took the same number of clock cycles (5).

Instruction Pipelining

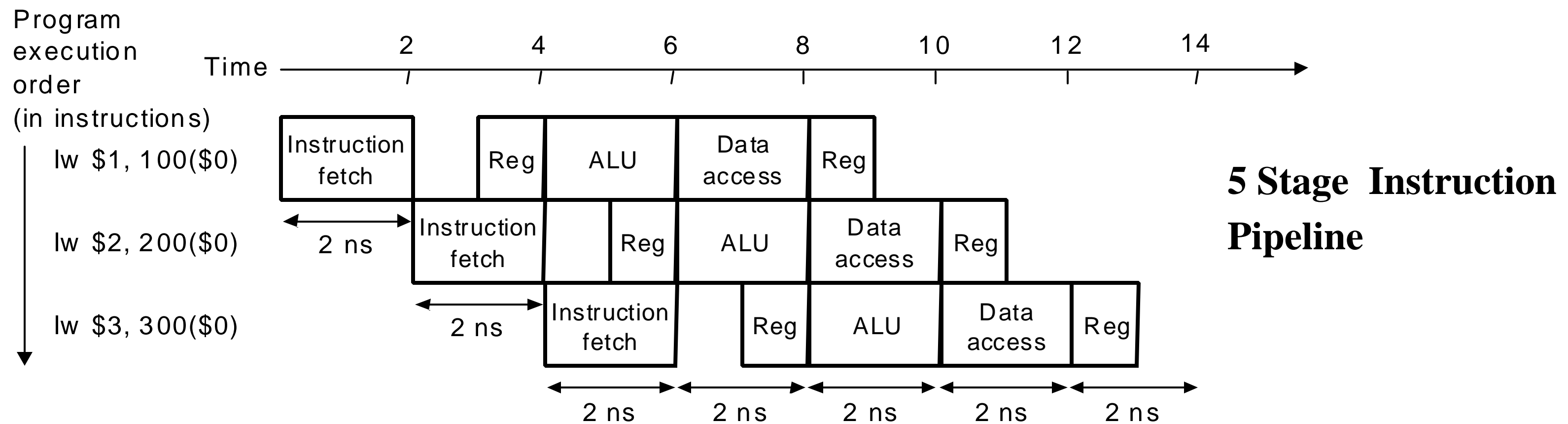
- **Instruction pipelining is a CPU implementation technique where multiple operations on a number of instructions are overlapped.**
  - **The next instruction is fetched in the next cycle without waiting for the current instruction to complete.**
- **An instruction execution pipeline involves a number of steps, where each step completes one part of an instruction.**
- **Each step is called** ***a pipeline stage*** **or** ***a pipeline segment.***
- **The stages or steps are connected one to the next to form a pipeline -- instructions enter at one end and progress through the stages and exit at the other end when completed.**
- **Instruction Pipeline Throughput : The instruction completion rate of the pipeline and is determined by how often an instruction exists the pipeline.**
- **The time to move an instruction one step down the line is is equal to** ***the machine cycle*** **and is determined by the stage with the longest processing delay.**
- **Instruction Pipeline Latency: The time required to complete an instruction:**

  Cycle time x Number of pipeline stages.

Single Cycle Vs. Pipelining



Time for 1000 instructions = 8 x 1000 = 8000 ns



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

Pipelining Speedup = 8000/2008 = 3.98

Pipelining: Design Goals

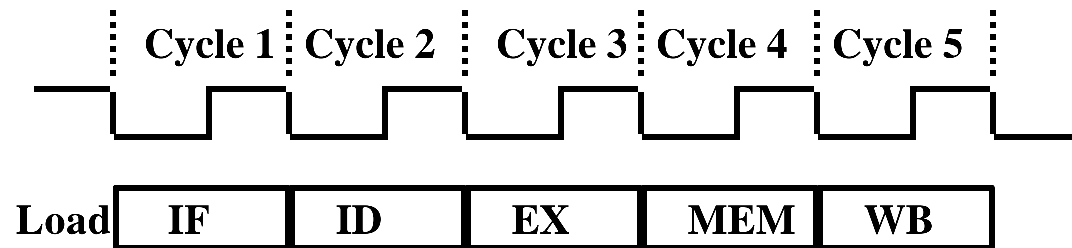
- The length of the machine clock cycle is determined by the time required for the slowest pipeline stage.
- An important pipeline design consideration is to balance the length of each pipeline stage.
- If all stages are perfectly balanced, then the time per instruction on a pipelined machine (assuming ideal conditions with no stalls):

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

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

From MIPS Multi-Cycle Datapath:

Five Stages of Load



1- Instruction Fetch (IF) Instruction Fetch
- Fetch the instruction from the Instruction Memory.

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

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.

Pipelined Instruction Processing Representation

Clock cycle Number — Time in clock cycles →

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

← Time to fill the pipeline →

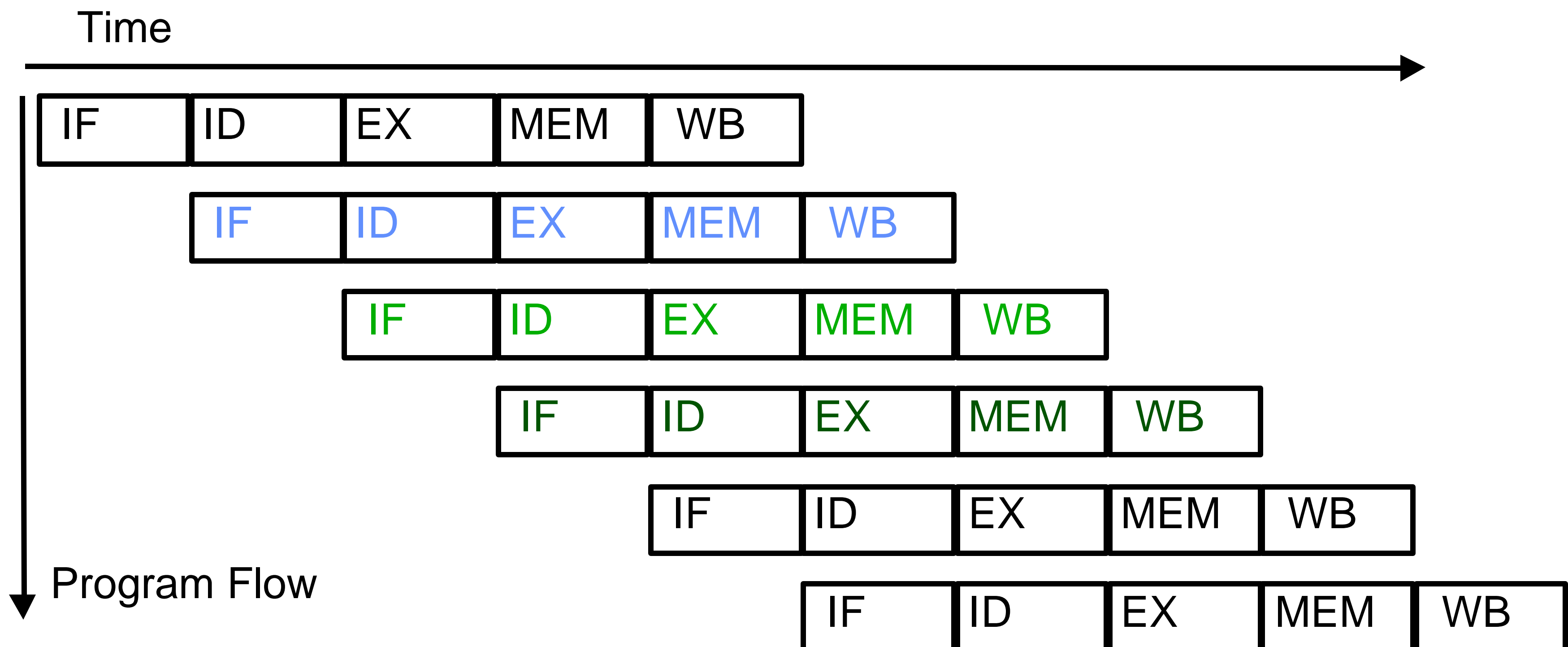
First instruction, I Completed

Last instruction, I+4 completed

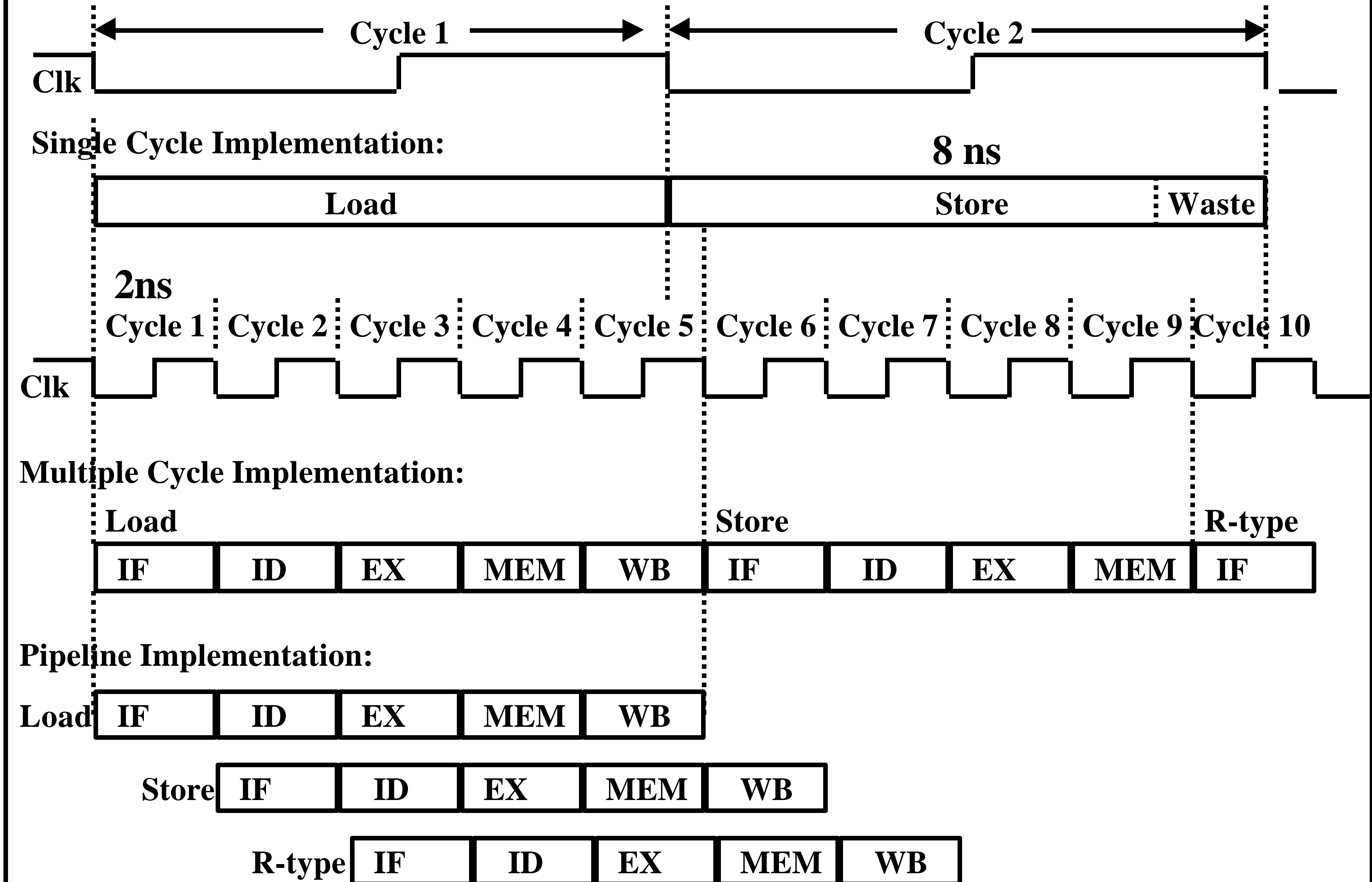
Pipeline Stages:

IF = Instruction Fetch
ID = Instruction Decode
EX = Execution
MEM = Memory Access
WB = Write Back

Pipelined Instruction Processing Representation



Single Cycle, Multi-Cycle, Vs. Pipeline



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

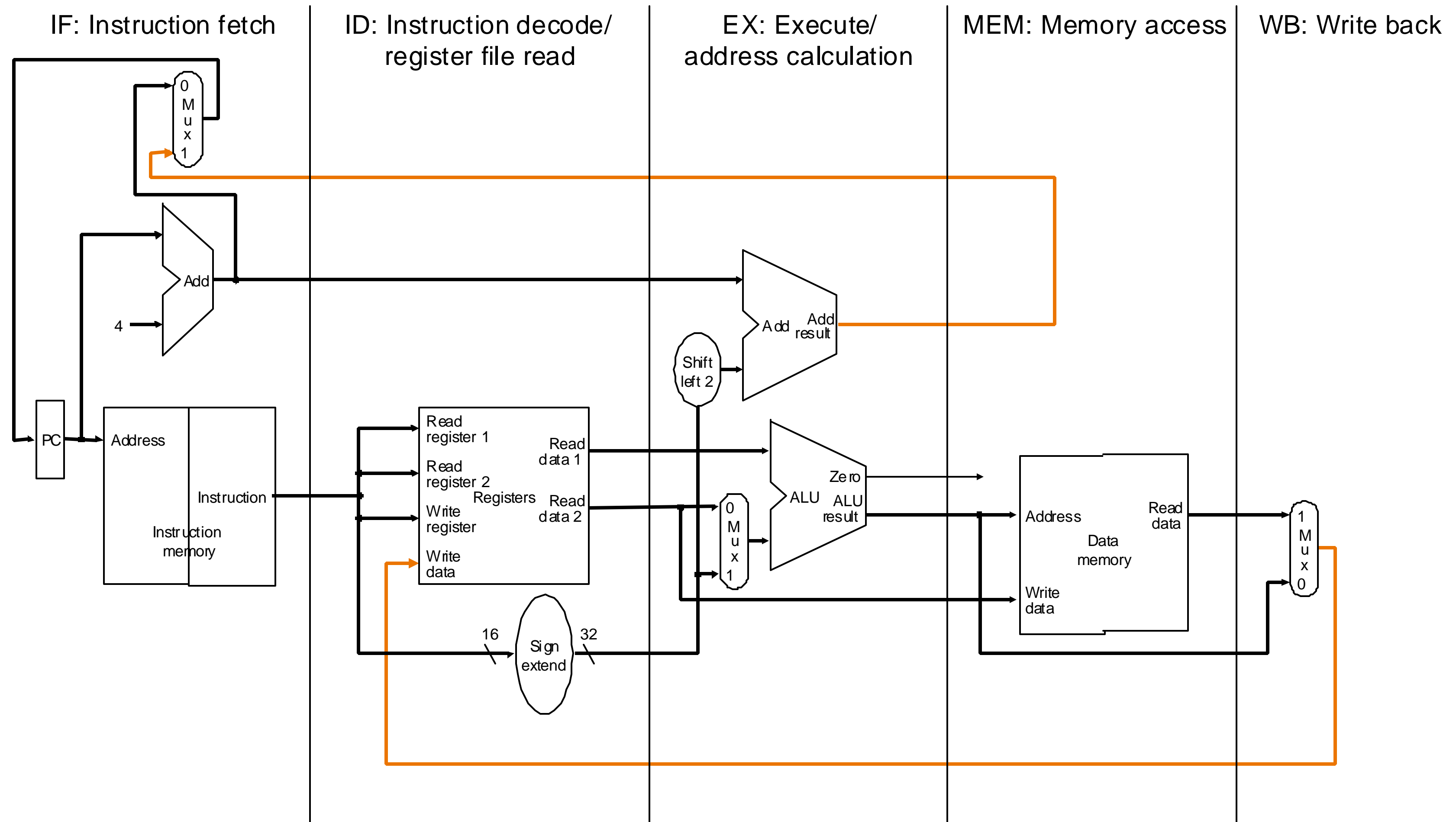
For 1000 instructions, execution time:

- **Single Cycle Machine:**
  - **8 ns/cycle x 1 CPI x 1000 inst = 8000 ns**
- **Multi-cycle Machine:**
  - **2 ns/cycle x 4.6 CPI (due to inst mix) x 1000 inst = 9200 ns**
- **Ideal pipelined machine, 5-stages:**
  - **2 ns/cycle x (1 CPI x 1000 inst + 4 cycle fill) = 2008 ns**

Basic Pipelined CPU Design Steps

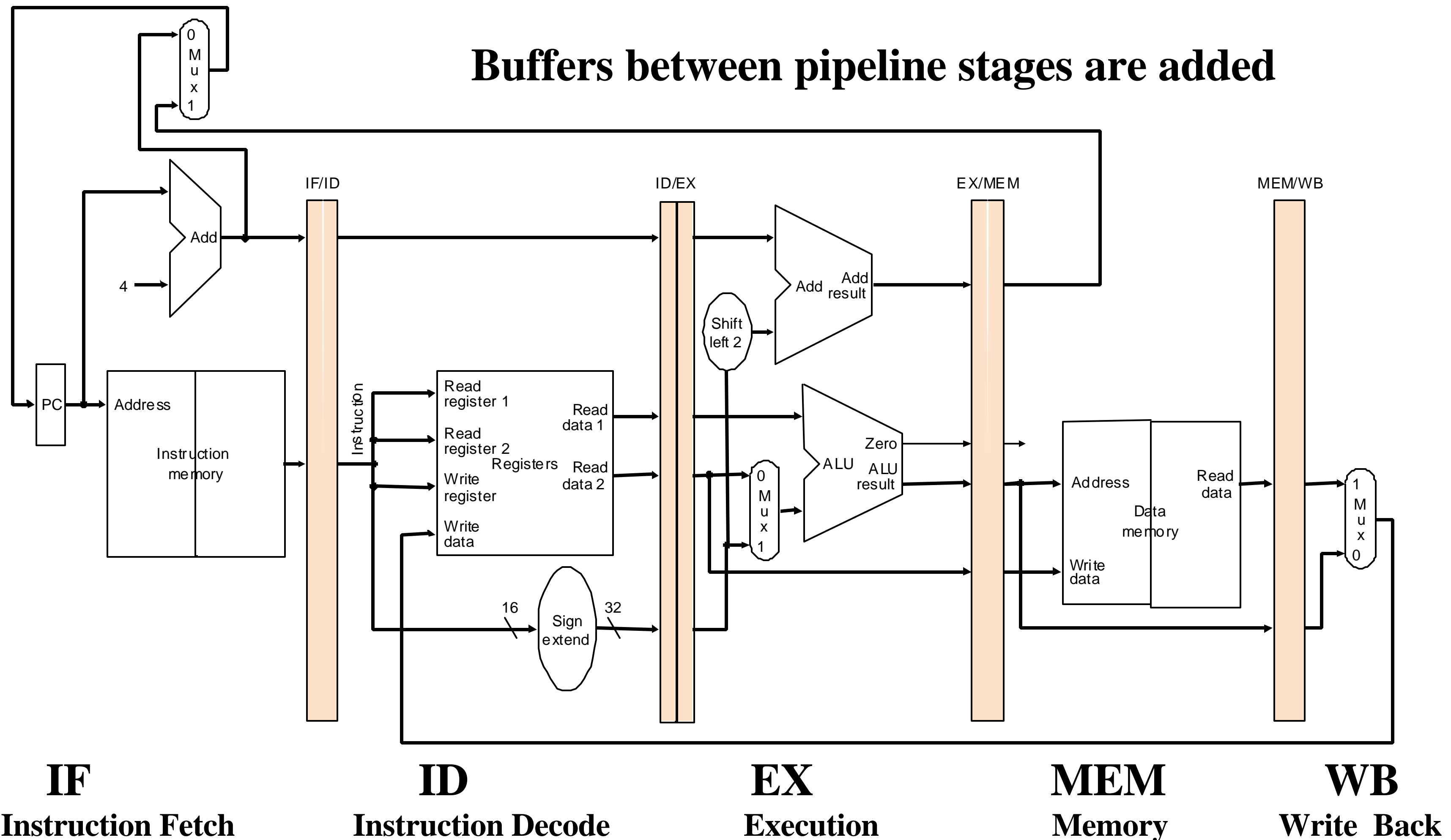
1. Analyze instruction set operations using independent RTN => datapath requirements.
2. Select required datapath components and connections.
3. Assemble an initial datapath meeting the ISA requirements.
4. Identify pipeline stages based on operation, balancing stage delays, and ensuring no hardware 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:
   - Instruction fields.
   - Remaining control lines needed for remaining pipeline stages.
   - All results produced by a stage and any unused results of previous stages.
6. Analyze implementation of each instruction to determine setting of control points that effects the register transfer taking pipeline hazard conditions into account .
7. Assemble the control logic.

MIPS Pipeline Stage Identification



What is needed to divide datapath into pipeline stages?

MIPS: An Initial Pipelined Datapath



IF Instruction Fetch

ID Instruction Decode

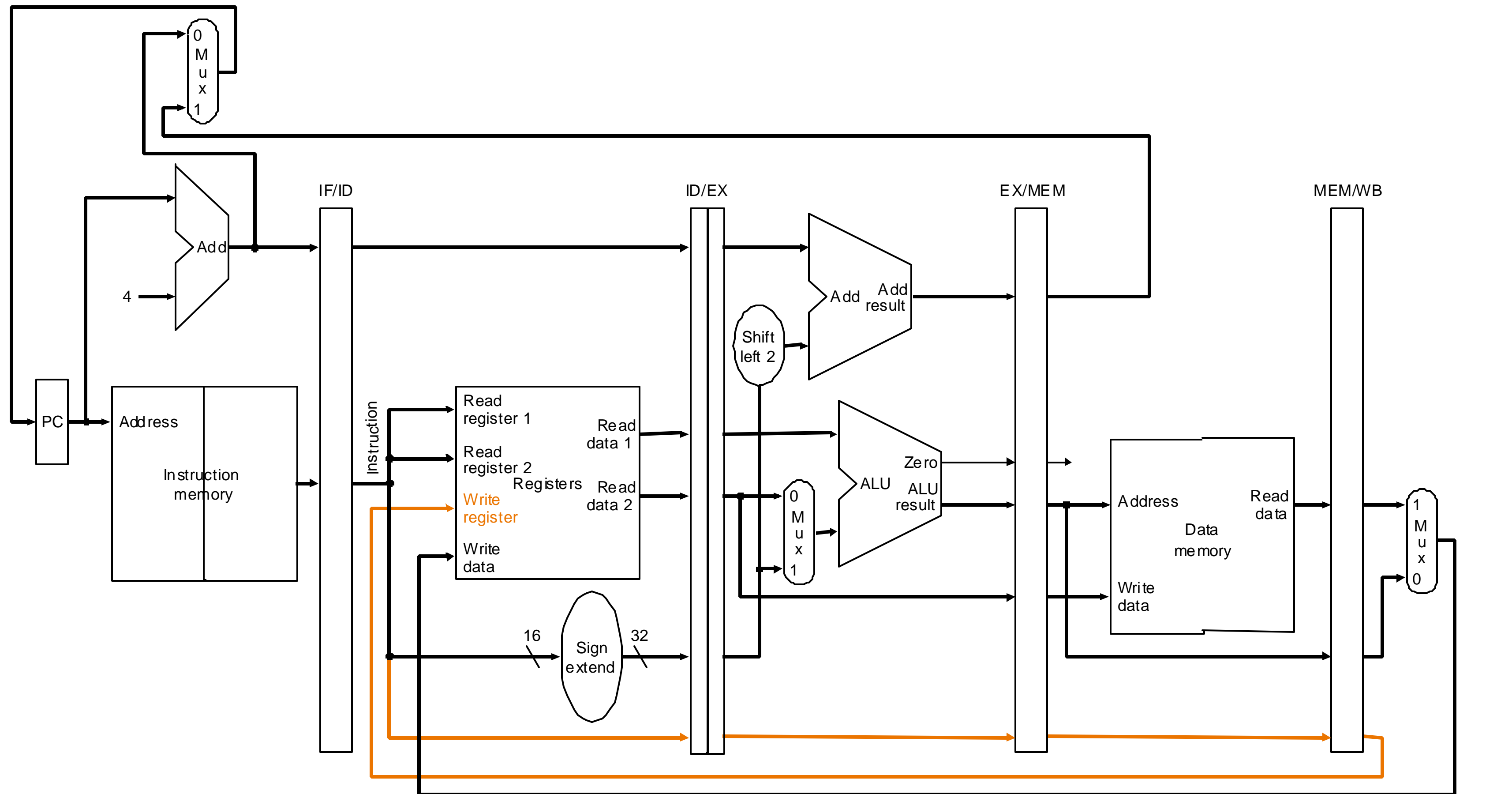
EX Execution

MEM Memory

WB Write Back

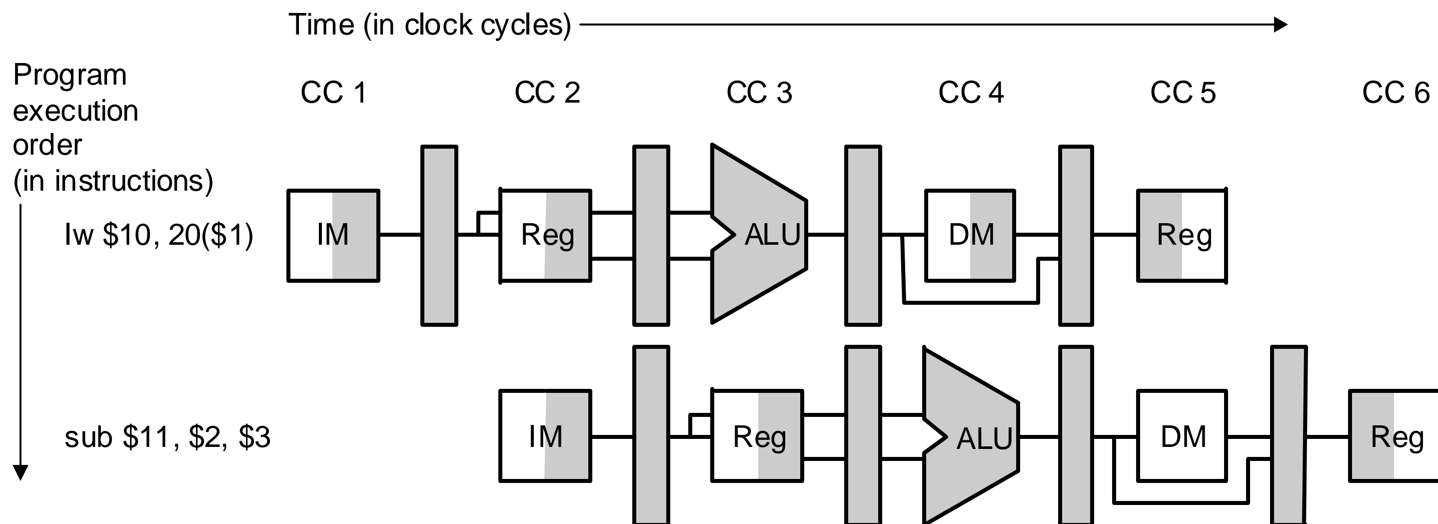
Can you find a problem even if there are no dependencies?
What instructions can we execute to manifest the problem?

A Corrected Pipelined Datapath



IF Instruction Fetch | **ID** Instruction Decode | **EX** Execution | **MEM** Memory | **WB** Write Back

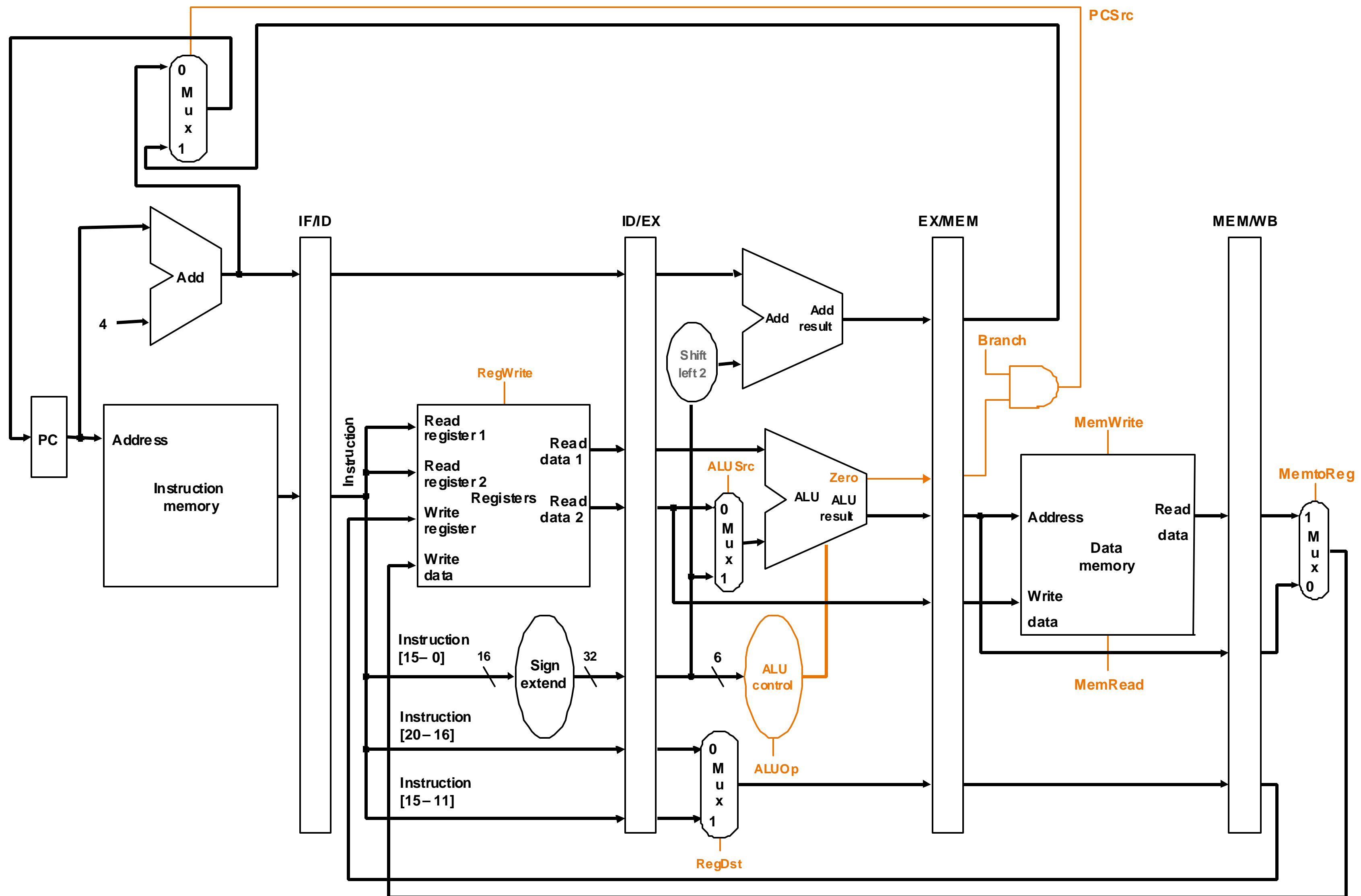
Representing Pipelines Graphically



Can help with answering questions like:

- **How many cycles does it take to execute this code?**
- **What is the ALU doing during cycle 4?**
- **Use this representation to help understand datapaths**

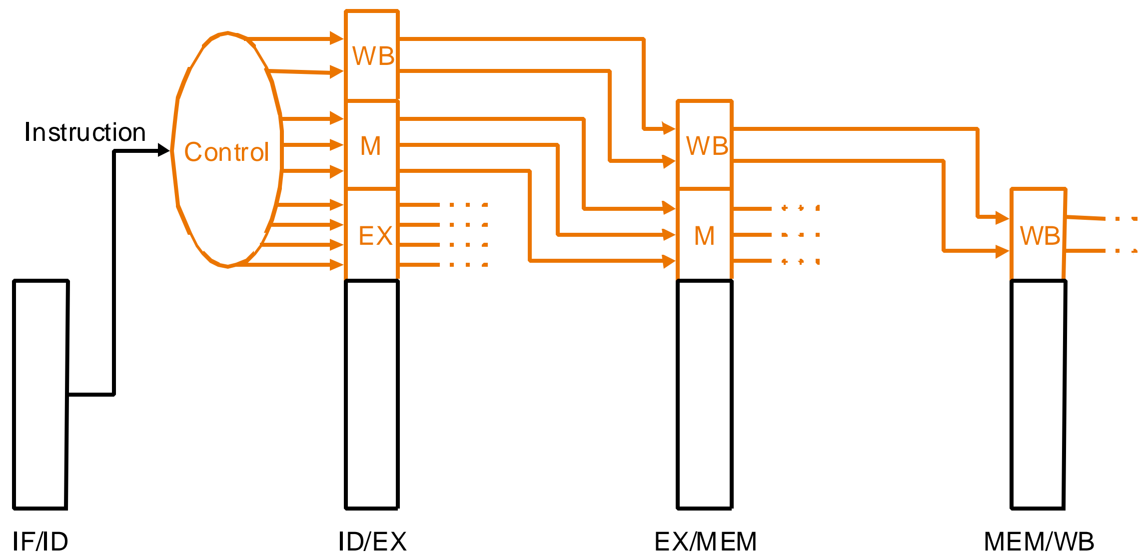
Adding Pipeline Control Points



Pipeline Control

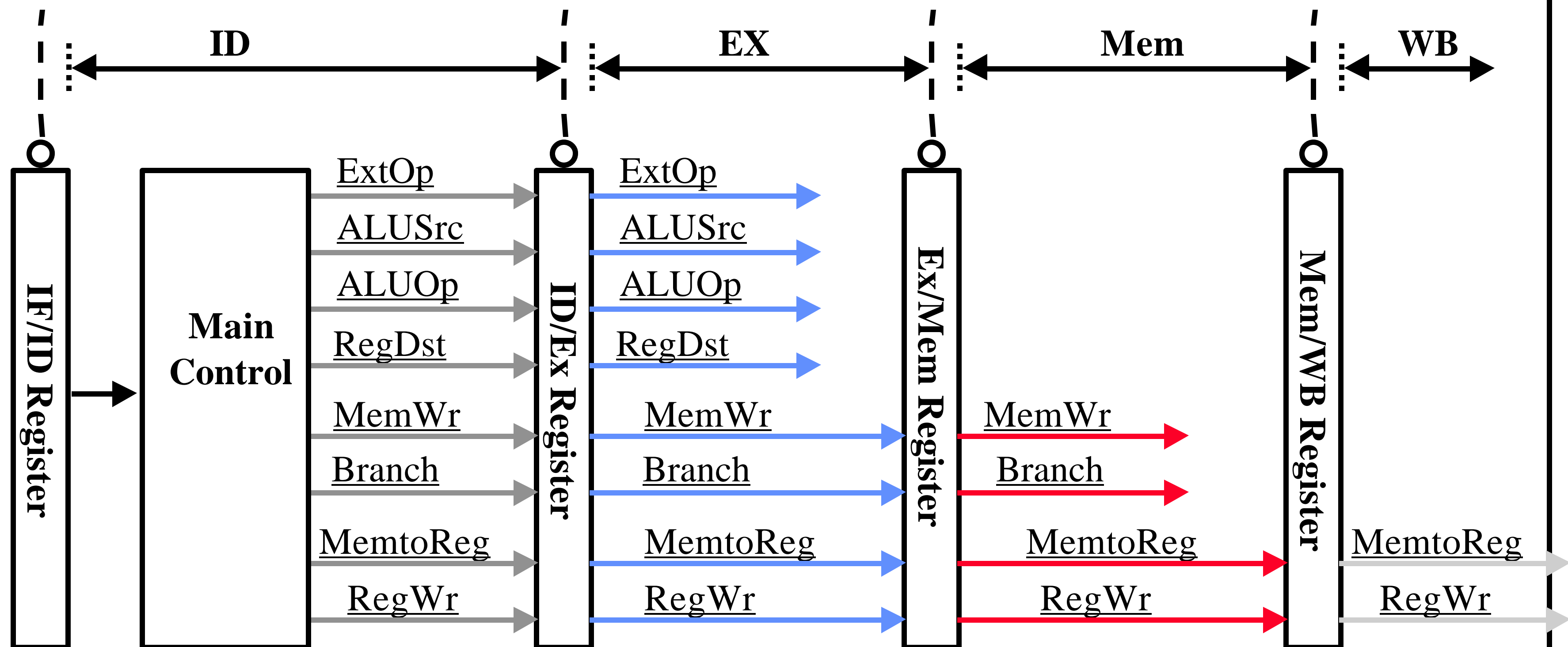
- **Pass needed control signals along from one stage to the next as the instruction travels through the pipeline just like the data**

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	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

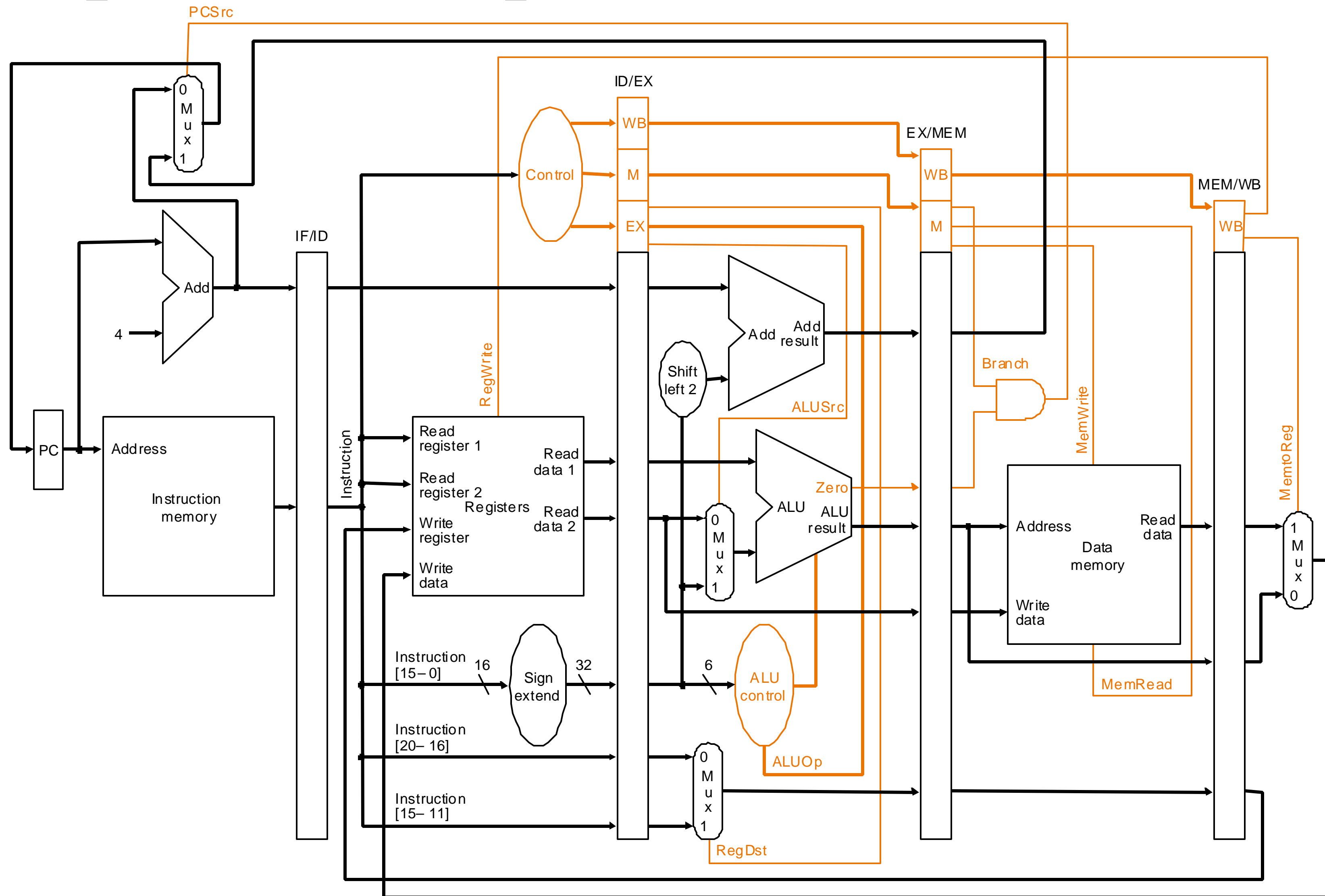


Pipeline Control

- The Main Control generates the control signals during Reg/Dec
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



Pipelined Datapath with Control Added



Target address of branch determined in MEM

Basic Performance Issues In Pipelining

- Pipelining increases the CPU instruction throughput: The number of instructions completed per unit time. Under ideal condition instruction throughput is one instruction per machine cycle, or CPI = 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).
- It usually slightly increases the execution time of each instruction over unpipelined implementations due to the increased control overhead of the pipeline and pipeline stage registers delays.

Pipelining Performance Example

- **Example: For an unpipelined machine:**
  - 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 machine clock cycle then the speedup in instruction execution from pipelining is:

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

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

In the pipelined five implementation five stages are used with an average instruction execution time of: 10 ns + 1 ns = 11 ns

$$\text{Speedup from pipelining} = \frac{\text{Instruction time unpipelined}}{\text{Instruction time pipelined}}$$

= 44 ns / 11 ns = 4 times

Pipeline Hazards

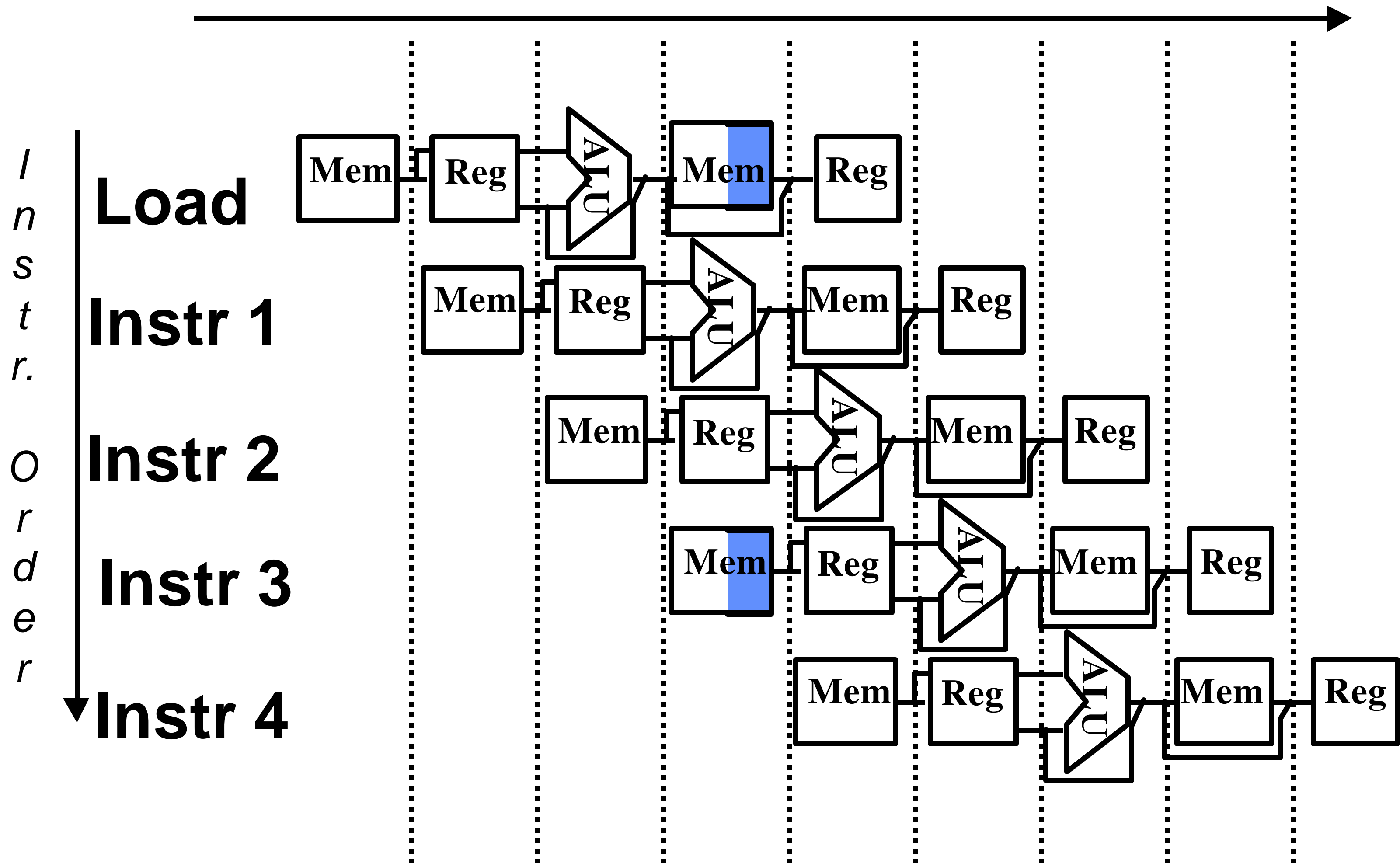
- Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle resulting in one or more stall cycles.
- Hazards reduce the ideal speedup gained from pipelining and are classified into three classes:
  - *Structural hazards*: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.
  - *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.
  - *Control hazards*: Arise from the pipelining of conditional branches and other instructions that change the PC.

Structural 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.**
- **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:**
  - when a machine has only one register file write port
  - or when a pipelined machine has a shared single-memory pipeline for data and instructions.

  → stall the pipeline for one cycle for register writes or memory data access

Structural hazard Example:
Single Memory For Instructions & Data



Detection is easy in this case (right half highlight means read, left half write)

A Structural Hazard Example

- Given that data references (loads/stores) are 40% for a specific instruction mix or program, and that the ideal pipelined CPI ignoring hazards is equal to 1.
- A pipelined CPU 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 CPU. Ignoring other performance losses for this CPU:

$$\text{Average instruction time} = \text{CPI} \times \text{Clock cycle time}$$

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

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

Data Hazards

- **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 to ensure correct execution.**
- **Example:**

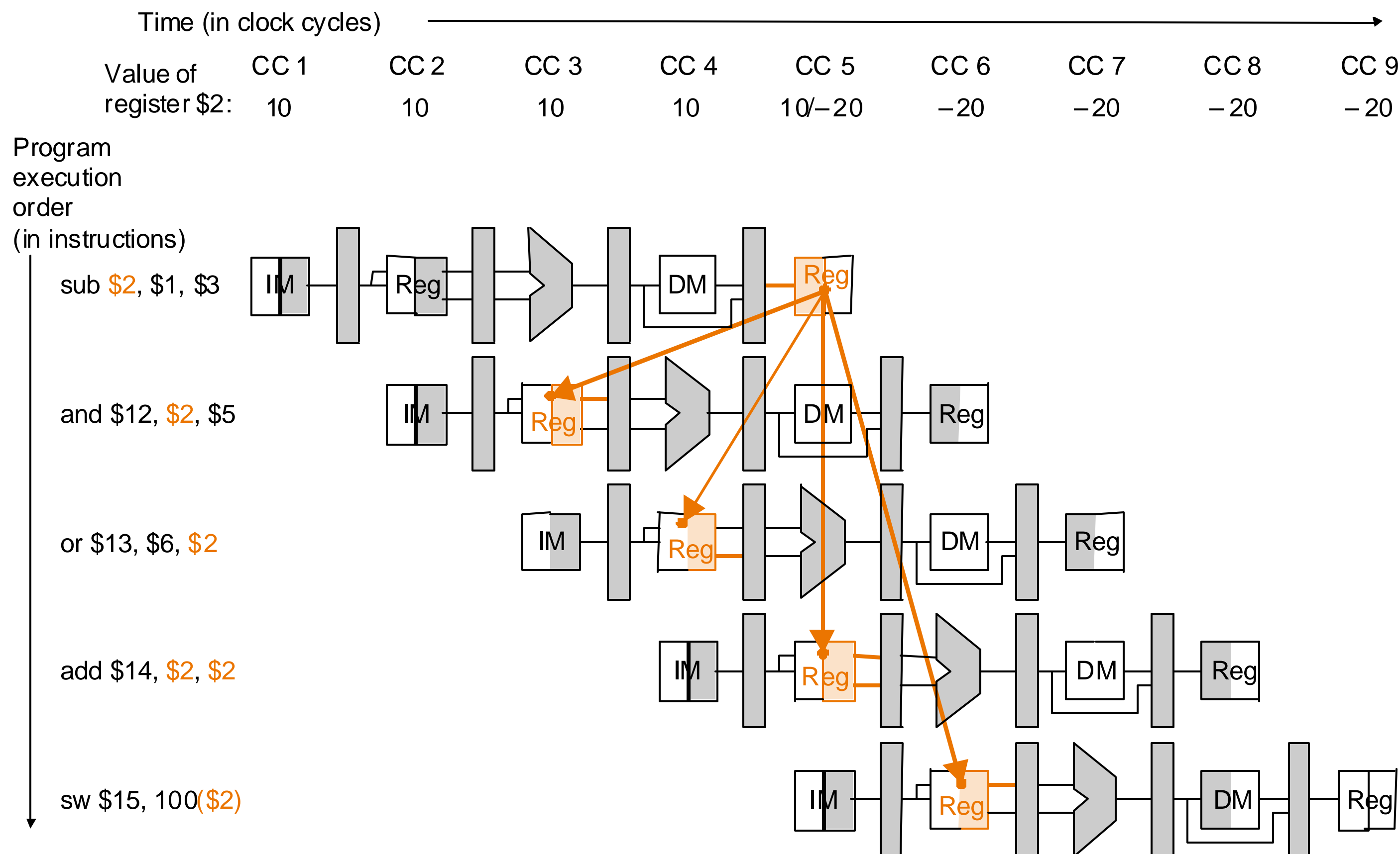
```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

  - **All the instructions after sub use the result of the sub instruction and may need to be stalled for correct execution.**

Data Hazards Example

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

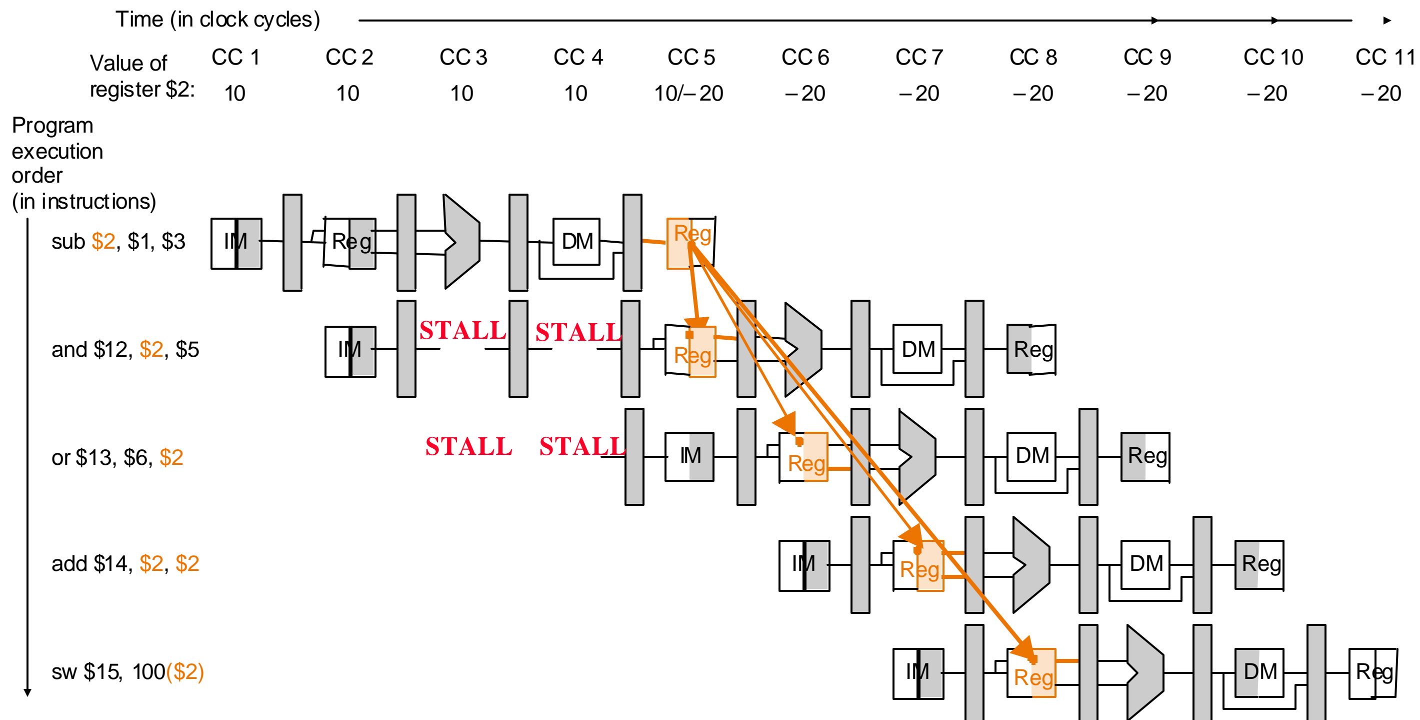
```
sub     $2, $1, $3
and     $12, $2, $5
or      $13, $6, $2
add     $14, $2, $2
sw      $15, 100($2)
```



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.



Performance of Pipelines with Stalls

- **Hazards in pipelines may make it necessary to stall the pipeline by one or more cycles and thus degrading performance from the ideal CPI of 1.**

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

- **If pipelining overhead is ignored and we assume that the stages are perfectly balanced then:**

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

- **When all instructions take the same number of cycles and is equal to the number of pipeline stages then:**

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

Data Hazard Resolution: Compiler Instruction Scheduling

- **The compiler can guarantee that no data hazards exist by re-ordering instructions and/or adding NOP instructions where needed.**
- **For the previous example:**

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

Data Hazard Resolution: Forwarding

- **Observation:**

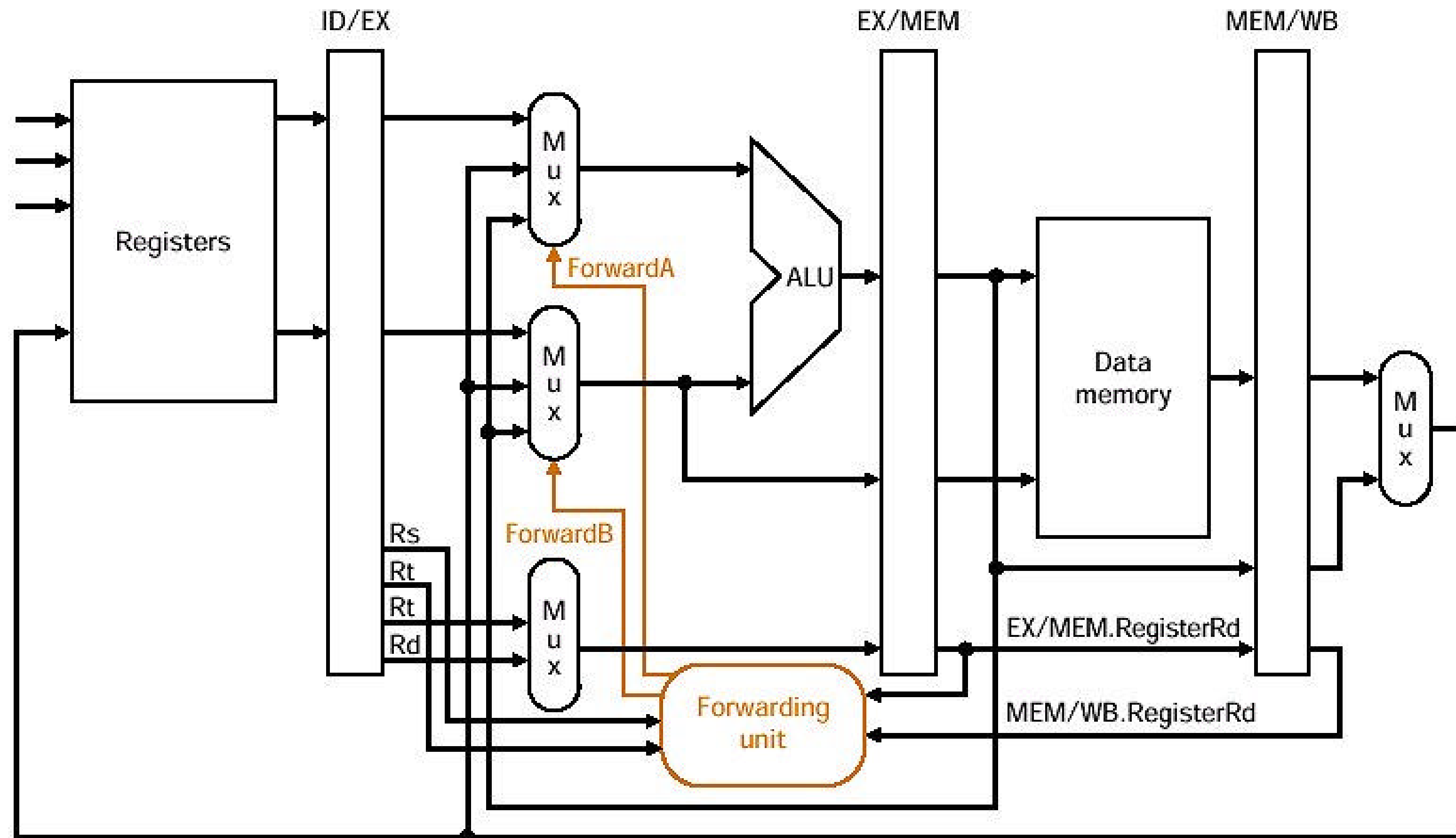
  Why not use temporary results produced by memory/ALU and not wait for them to be written back in the register bank.

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

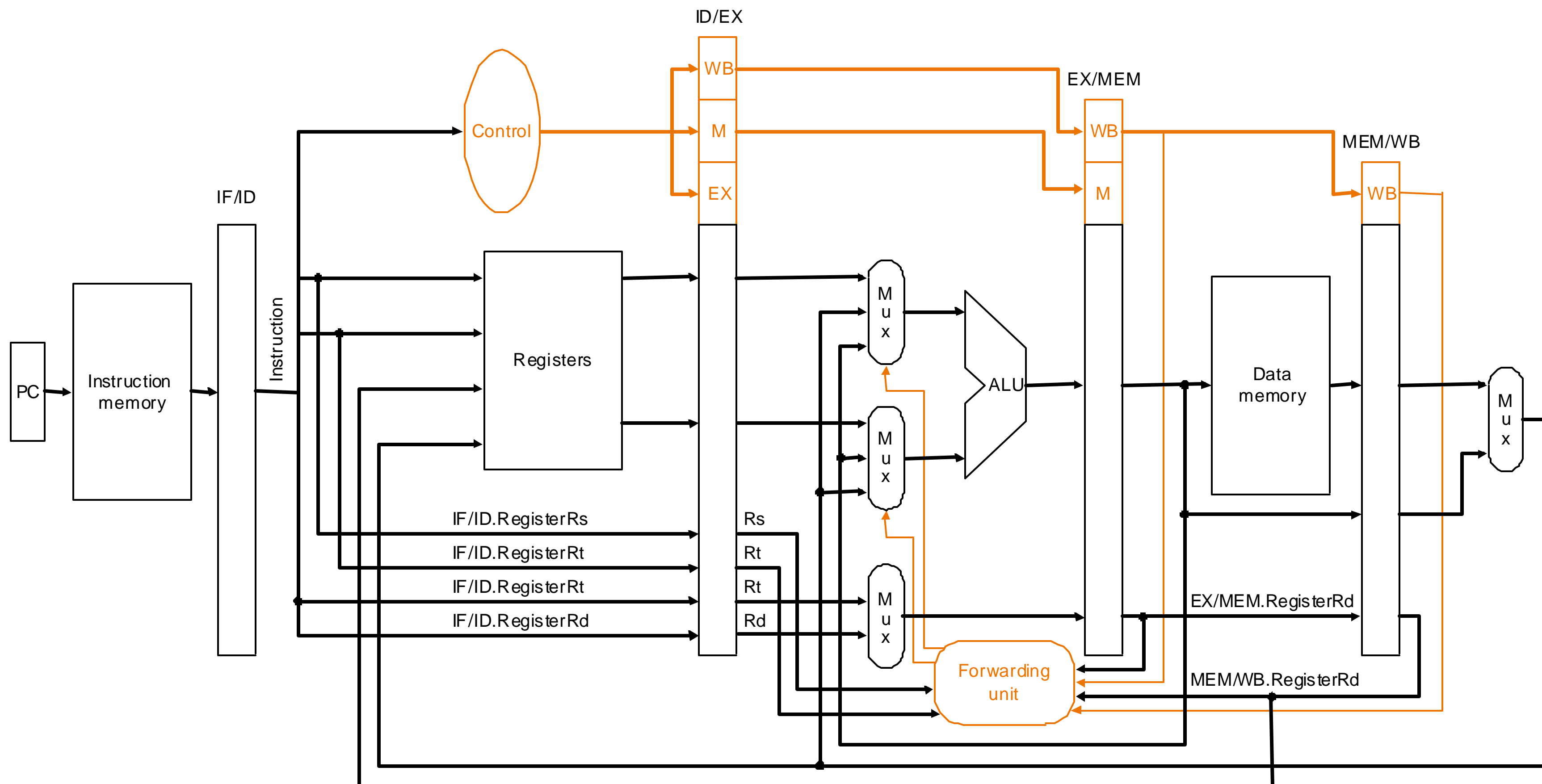
- **Using forwarding hardware, the result of an instruction is copied directly from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)**

Data Hazard Resolution: Forwarding

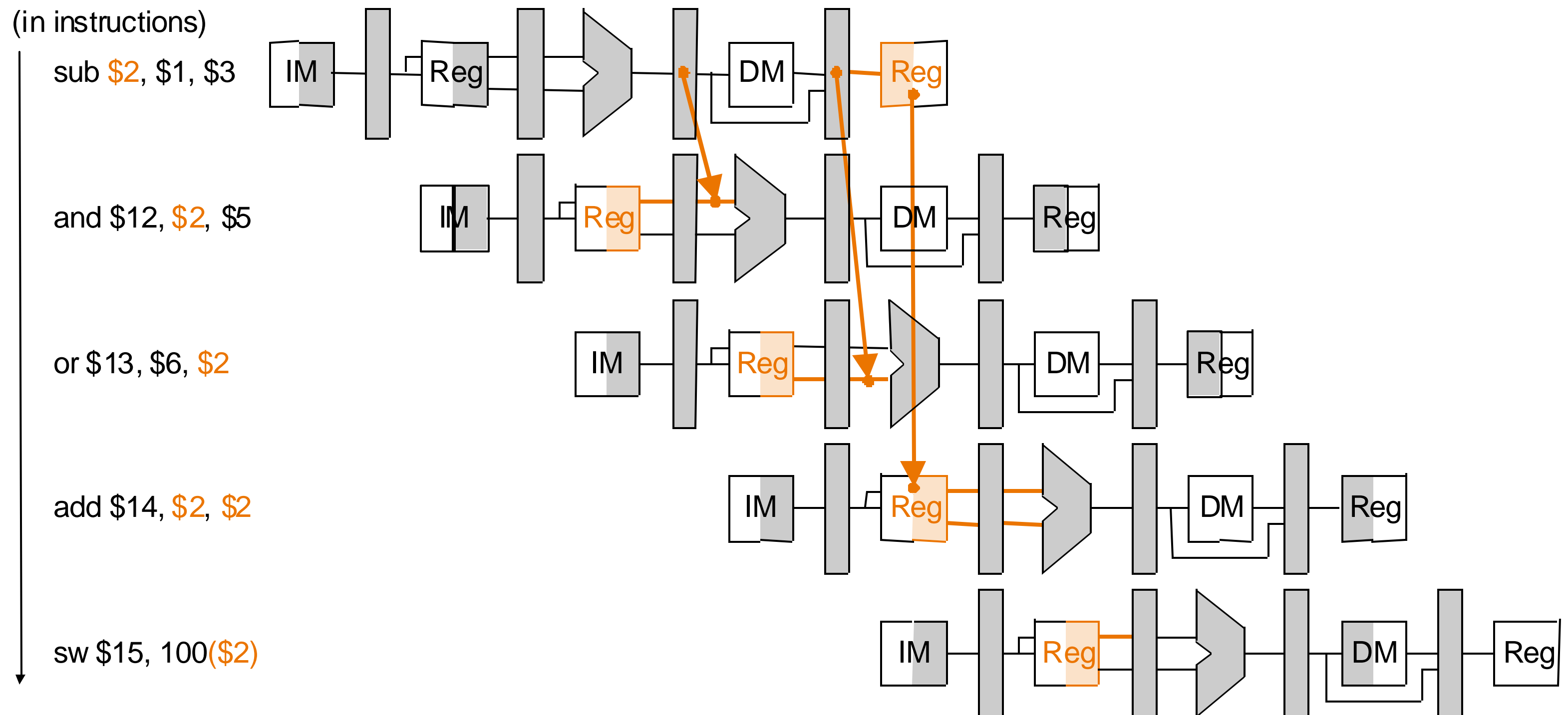
- Register file forwarding to handle read/write to same register
- ALU forwarding



Pipelined Datapath With Forwarding

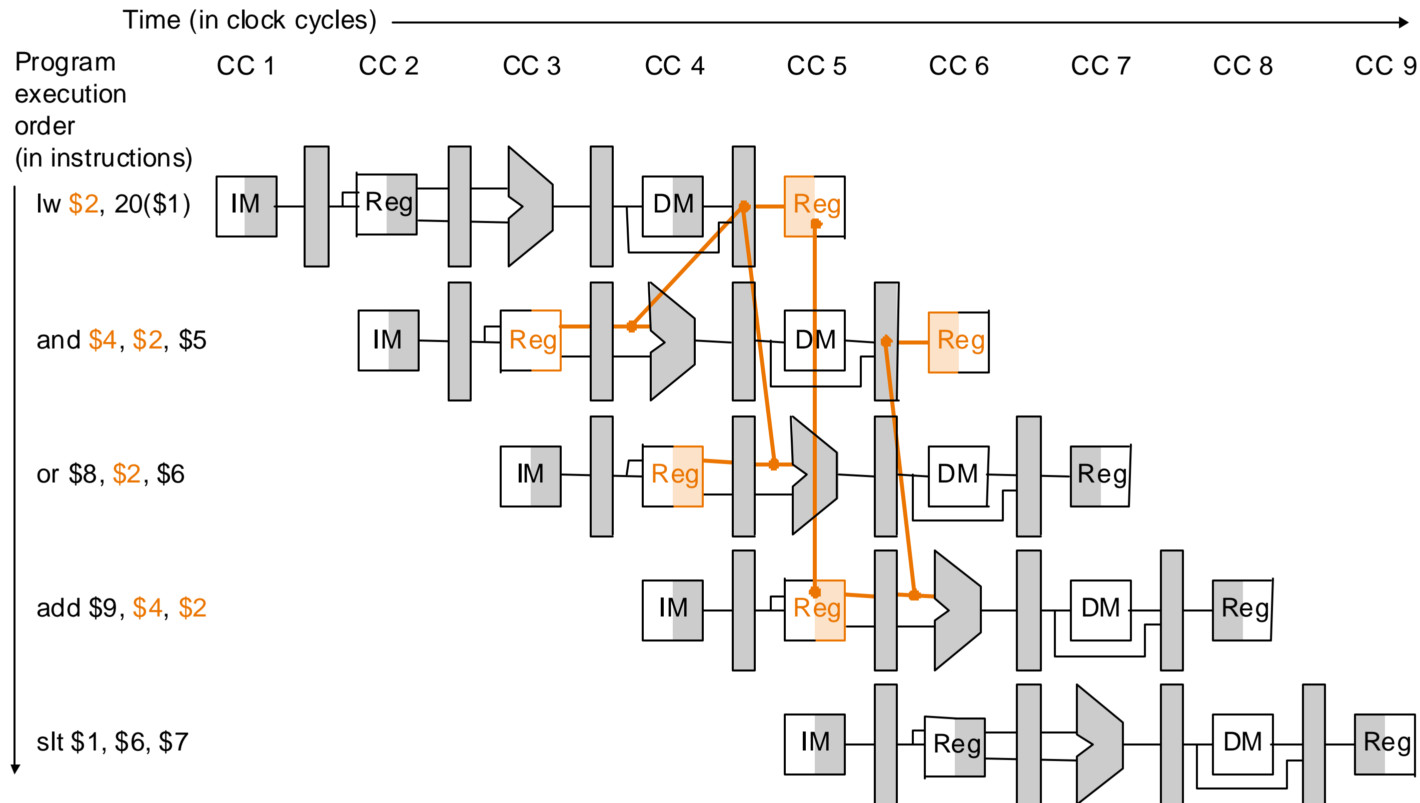


Data Hazard Example With Forwarding



A Data Hazard Requiring A Stall

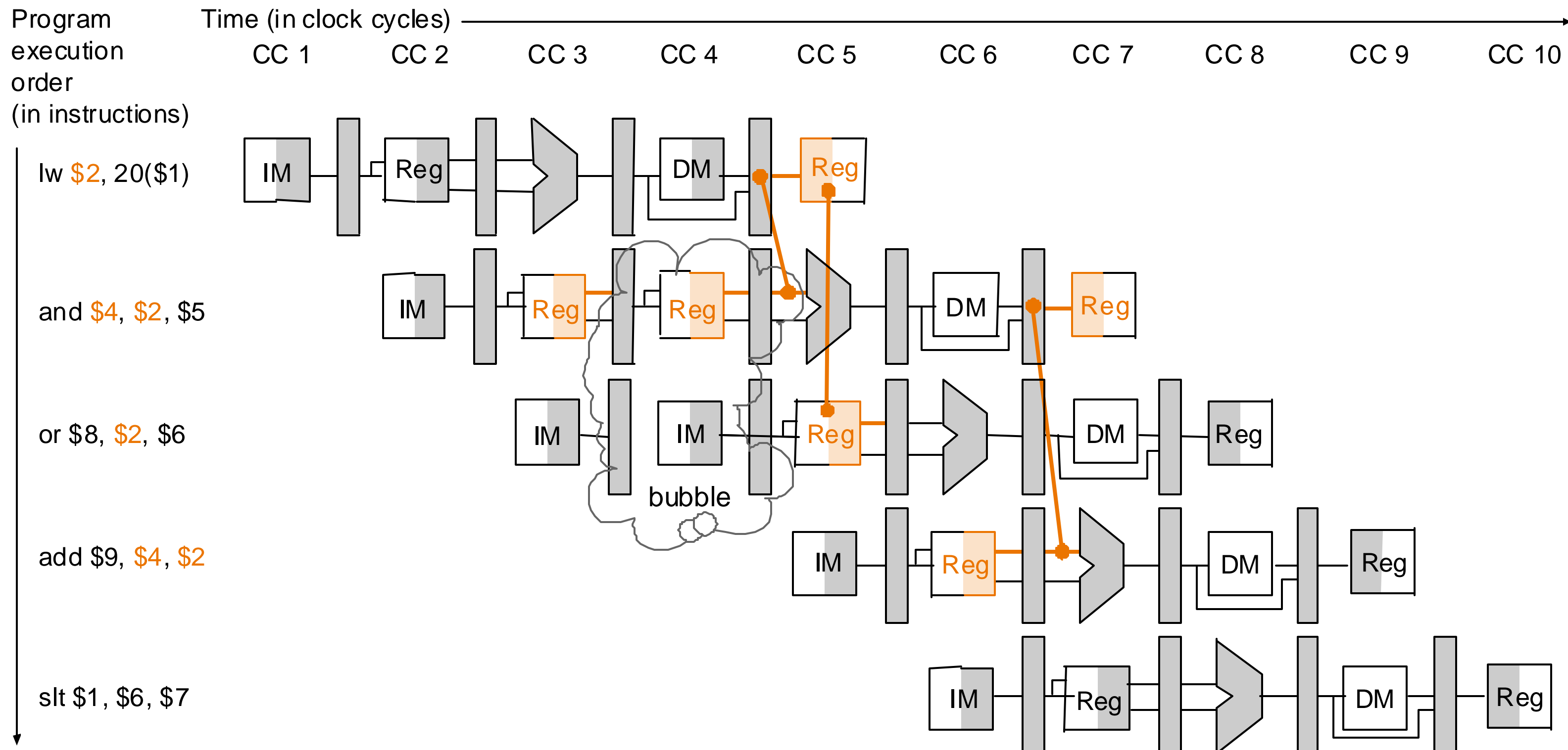
A load followed by an R-type instruction that uses the loaded value



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

A Data Hazard Requiring A Stall

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



- We can stall the pipeline by keeping an instruction in the same stage

Compiler Scheduling Example

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

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

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

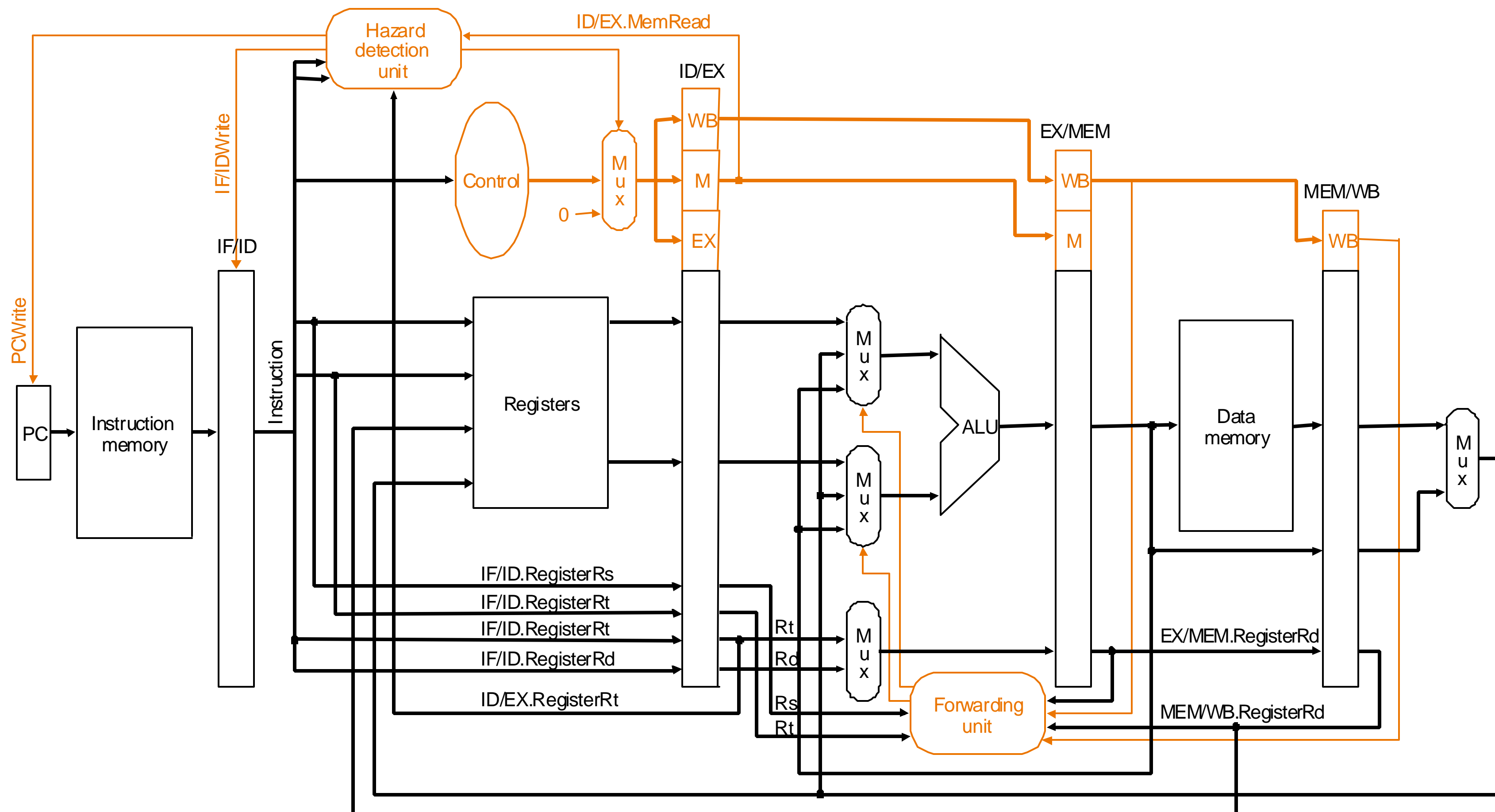
```
lw     $16, 4($2)
lw     $15, 0($2)
add    $14, $5, $16
sw     $16, 4($2)
```

- Without forwarding we need two independent instructions to place between them, so in addition a nop is added.

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

Datapath With Hazard Detection Unit

A load followed by an instruction that uses the loaded value is detected and a stall cycle is inserted.



EECC550 - Shaaban

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.

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.

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.

Control Hazards

- **When a conditional branch is executed it may change the PC and, without any special measures, leads to stalling the pipeline for a number of cycles until the branch condition is known (branch is resolved).**
- **In current MIPS pipeline, the conditional branch is resolved in the MEM stage resulting in three stall cycles as shown below:**

Branch instruction	**IF**	**ID**	**EX**	**MEM**	**WB**					
Branch successor		**stall**	**stall**	**stall**	**IF**	**ID**	**EX**	**MEM**	**WB**	
Branch successor + 1						**IF**	**ID**	**EX**	**MEM**	**WB**
Branch successor + 2							**IF**	**ID**	**EX**	**MEM**
Branch successor + 3								**IF**	**ID**	**EX**
Branch successor + 4									**IF**	**ID**
Branch successor + 5										**IF**

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 = 3 Cycles

Basic Branch Handling in Pipelines

- **One scheme discussed earlier is to *flush or freeze* the pipeline whenever a conditional branch is decoded 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

- **Another method is to *predict that the branch is not taken* where the state of the machine is not changed until the branch outcome is definitely known. Execution here continues with the next instruction; *stall occurs here when the branch is taken*.**

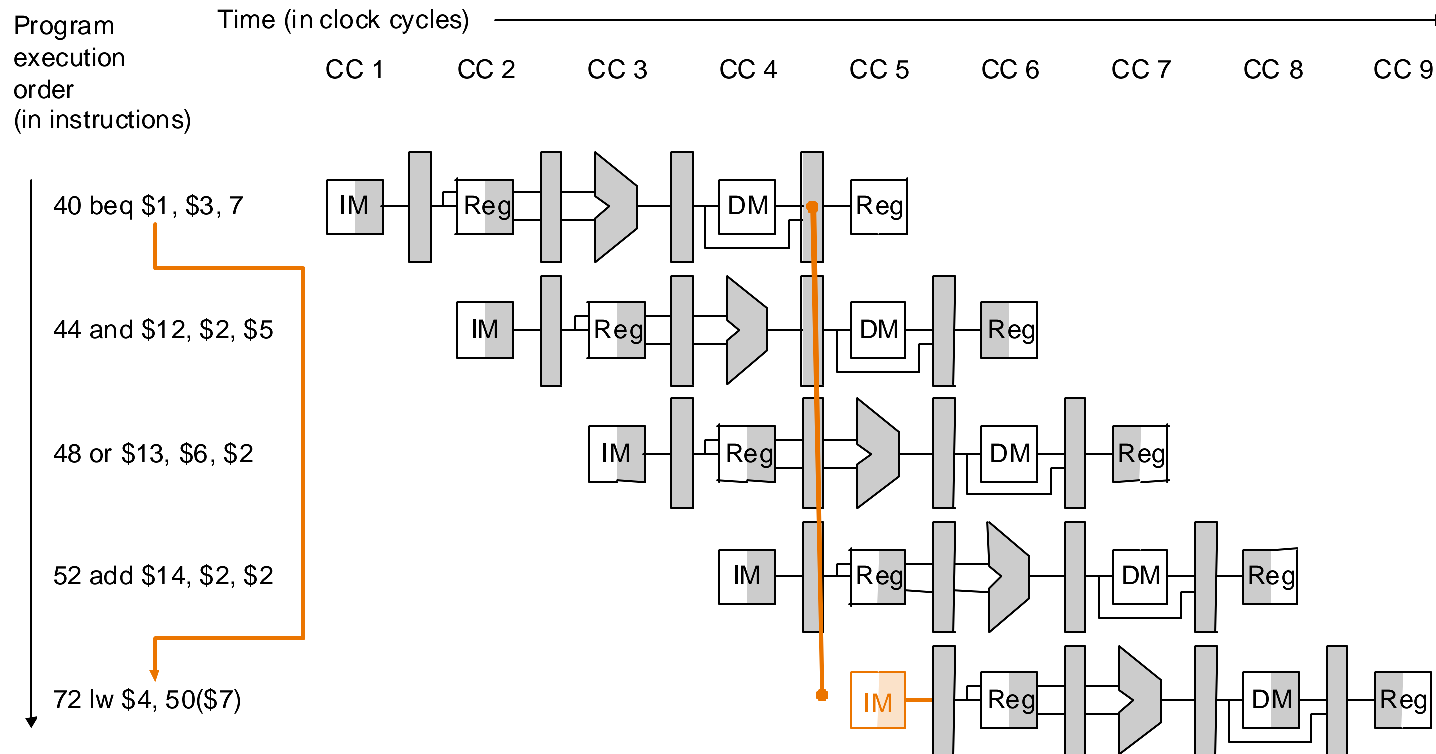
Pipeline stall cycles from branches = frequency of taken branches X branch penalty

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

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

Control Hazards: Example

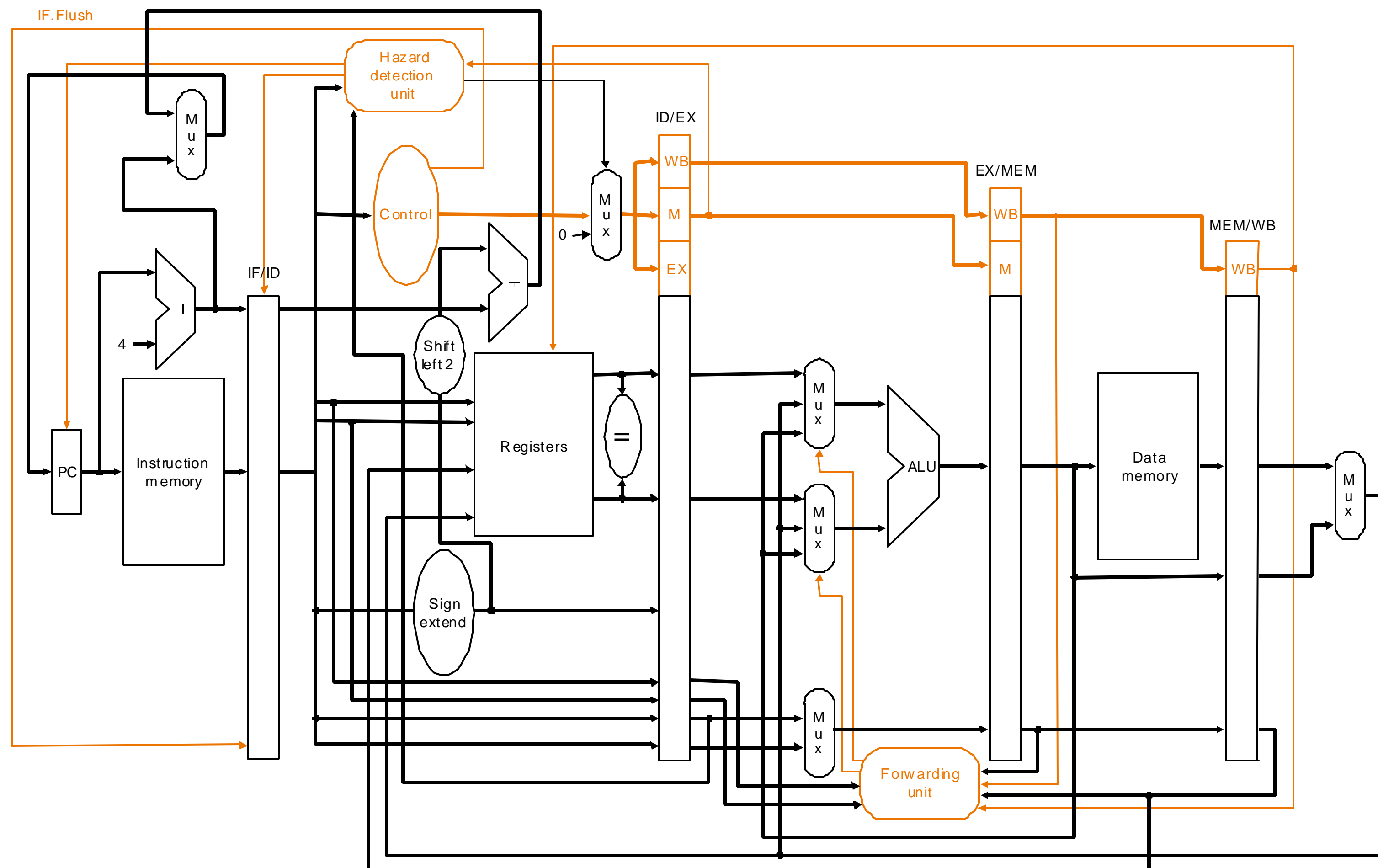
- **Three other instructions are in the pipeline before branch instruction target decision is made when BEQ is in MEM stage.**



- **In the above diagram, we are predicting "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.**

Reducing Delay of Taken Branches

- **Next PC of a branch known in MEM stage: Costs three lost cycles if taken.**
- **If next PC is known in EX stage, one cycle is saved.**
- **Branch address calculation can be moved to ID stage using a register comparator, costing only one cycle if branch is taken.**



Reduction of Branch Penalties: Delayed Branch

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

  conditional branch instruction
  sequential successor$_1$
  sequential successor$_2$
  ……..
  sequential successor$_n$
  branch target if taken

- **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 utiliza a single instruction delay slot.**
  - **The job of the compiler is to make the successor instruction in the delay slot valid and useful instruction.**

Delayed Branch Example

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 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.

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.

B An instruction from the target of the branch:

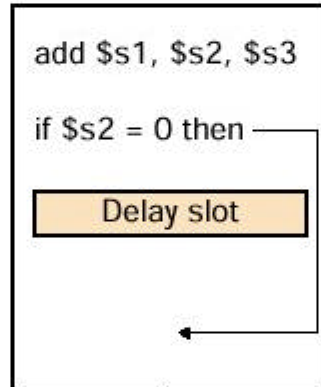
Improves performance if the branch is taken and may require instruction duplication. This instruction must be safe to execute if the branch is not taken.

C An instruction from the fall through instruction stream:

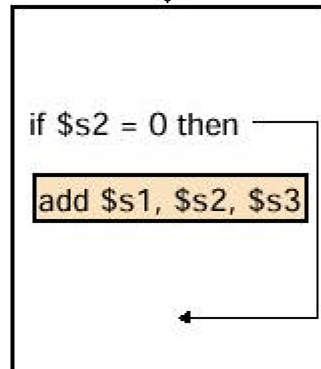
Improves performance when the branch is not taken. The instruction must be safe to execute when the branch is taken.

Scheduling the branch delay slot

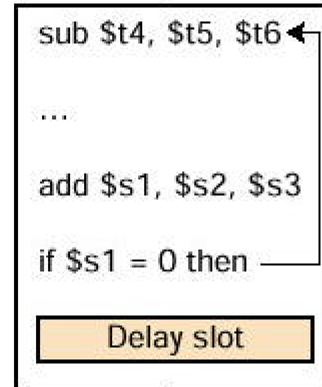
a. From before



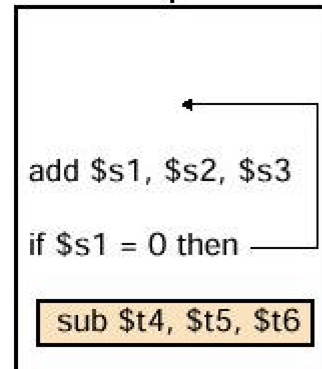
Becomes



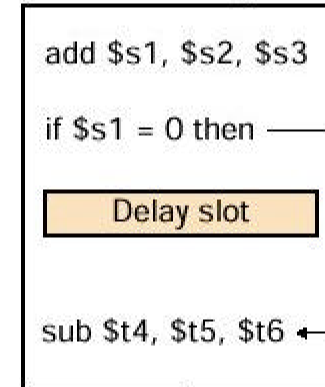
b. From target



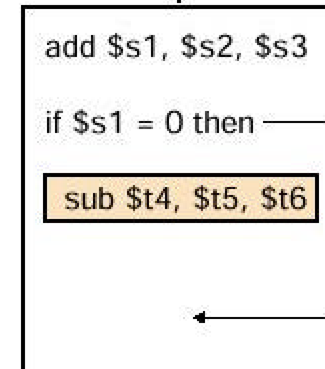
Becomes



c. From fall through



Becomes



Compiler Instruction Scheduling Example With Branch Delay Slot

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

```
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 with and without scheduling?

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
```

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)
```

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 = 1.6

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
Store	10%	
branch	20%	of which 45% are taken (assume not taken)

- **What is the resulting CPI for the pipelined MIPS with forwarding and branch address calculation in ID stage?**

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

$$= 1 + \text{stalls by loads} + \text{stalls by branches}$$

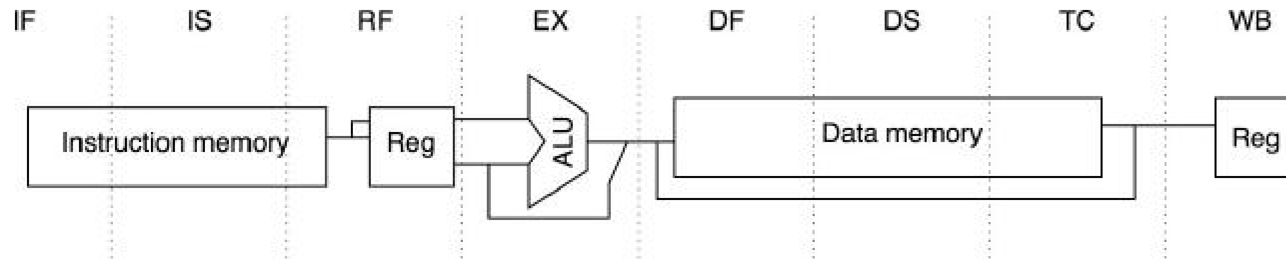
$$= 1 + .3 \times .25 \times 1 + .2 \times .45 \times 1$$

$$= 1 + .075 + .09$$

$$= 1.165$$

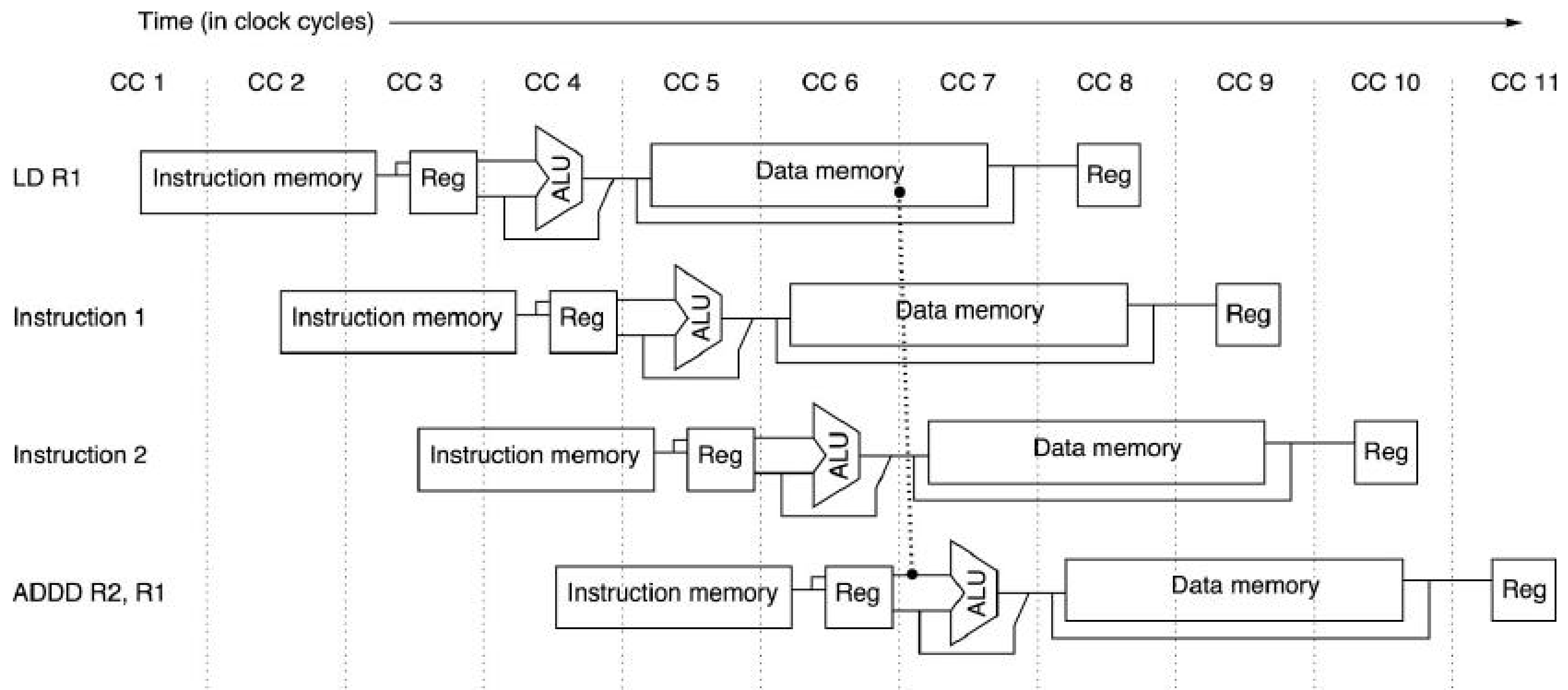
The MIPS R4000 Integer Pipeline

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



- **Pipeline Stages:**
  - **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 Penalty = 3 cycles if taken (2 with branch delay slot)**

MIPS R4000 Example



- Even with forwarding the deeper pipeline leads to a 2-cycle load delay.