- **Instruction Pipelining Review:**
  - **MIPS In-Order Single-Issue Integer Pipeline** (ISA)
  - **Performance of Pipelines with Stalls**
  - **Pipeline Hazards**
    - ***Structural hazards***
    - ***Data (operands) hazards***
      - Minimizing Data hazard Stalls by Forwarding
      - Data Hazard Classification
      - Data Hazards Present in Current MIPS Pipeline
    - ***Control hazards***
      - Reducing Branch Stall Cycles
      - Static Compiler Branch Prediction
      - Delayed Branch Slot
        - Canceling Delayed Branch Slot
- **Pipelining and Handling of Exceptions**
  - ***Precise exception Handling***
- **Extending The MIPS Pipeline to Handle Floating-Point Operations**
  - Pipeline Characteristics With FP Support + Performance
  - Maintaining Precise Exceptions in FP/Multicycle Pipelining

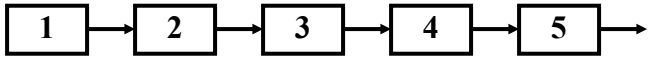
Pipelined MIPS CPU Design from 350 (Comp. Org.)

In-Order Pipelined CPU:
All Instructions Go Through All Pipeline Stages Strictly in Program Order

(In Appendix A)

Instruction Pipelining Review

- Instruction pipelining is CPU implementation technique where multiple operations on a number of instructions are overlapped.
  - Instruction pipelining exploits Instruction-Level Parallelism (ILP)
- 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 -- instructions enter at one end and progress through the stages and exit at the other end.



- The time to move an instruction one step down the pipeline is is equal to *the machine 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.
  - Under ideal conditions (no stall cycles), instruction throughput is one instruction per machine cycle, or ideal CPI = 1 Or 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

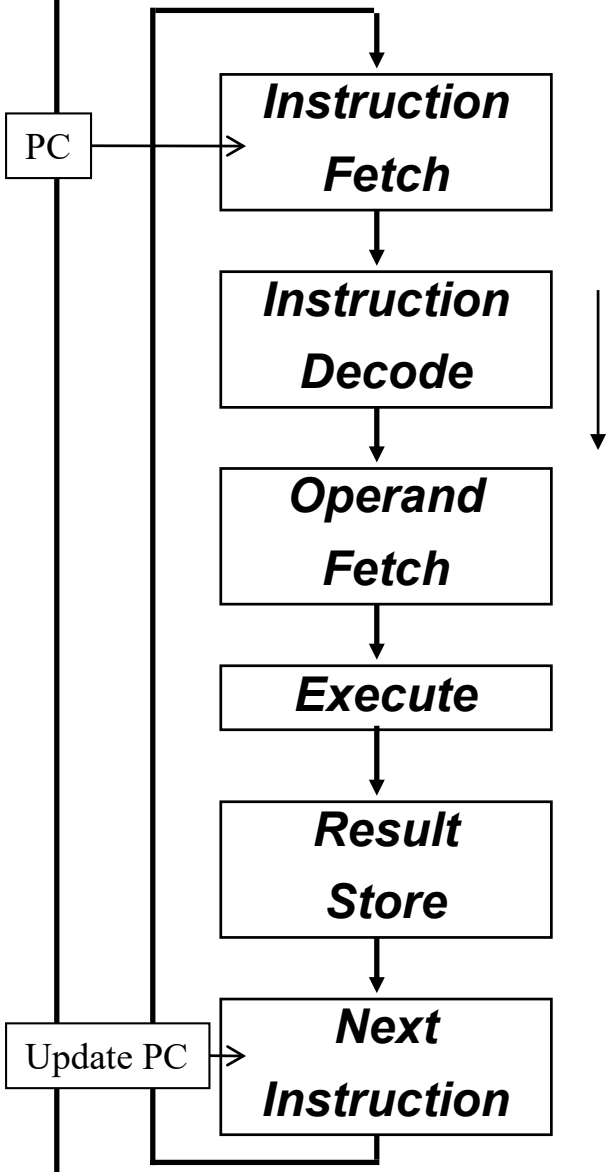
(In Appendix A and 350)

The pipeline described here is called an in-order pipeline because instructions are processed or executed in the original program order

Order = Program order

Generic CPU Machine Instruction Processing Steps

(Implied by The Von Neumann Computer Model)



Obtain instruction from program storage

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

Determine required actions and instruction size

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)

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

(Classic 5-Stage)

MIPS In-Order Single-Issue Integer Pipeline Ideal Operation

i.e. execution in program order

Fill Cycles = number of stages -1

(No stall cycles)

Program Order ↓

	Instruction Number	Clock Number 1	2	3	4	5	6	7	8	9
		Time in clock cycles →								
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

4 cycles = n -1 = 5 -1

Time to fill the pipeline

MIPS Pipeline Stages:

IF = Instruction Fetch

ID = Instruction Decode

EX = Execution

MEM = Memory Access

WB = Write Back

First instruction, I Completed

Last instruction, I+4 completed

n= 5 pipeline stages **Ideal CPI =1**

(or IPC =1)

In-order = instructions executed in original program order

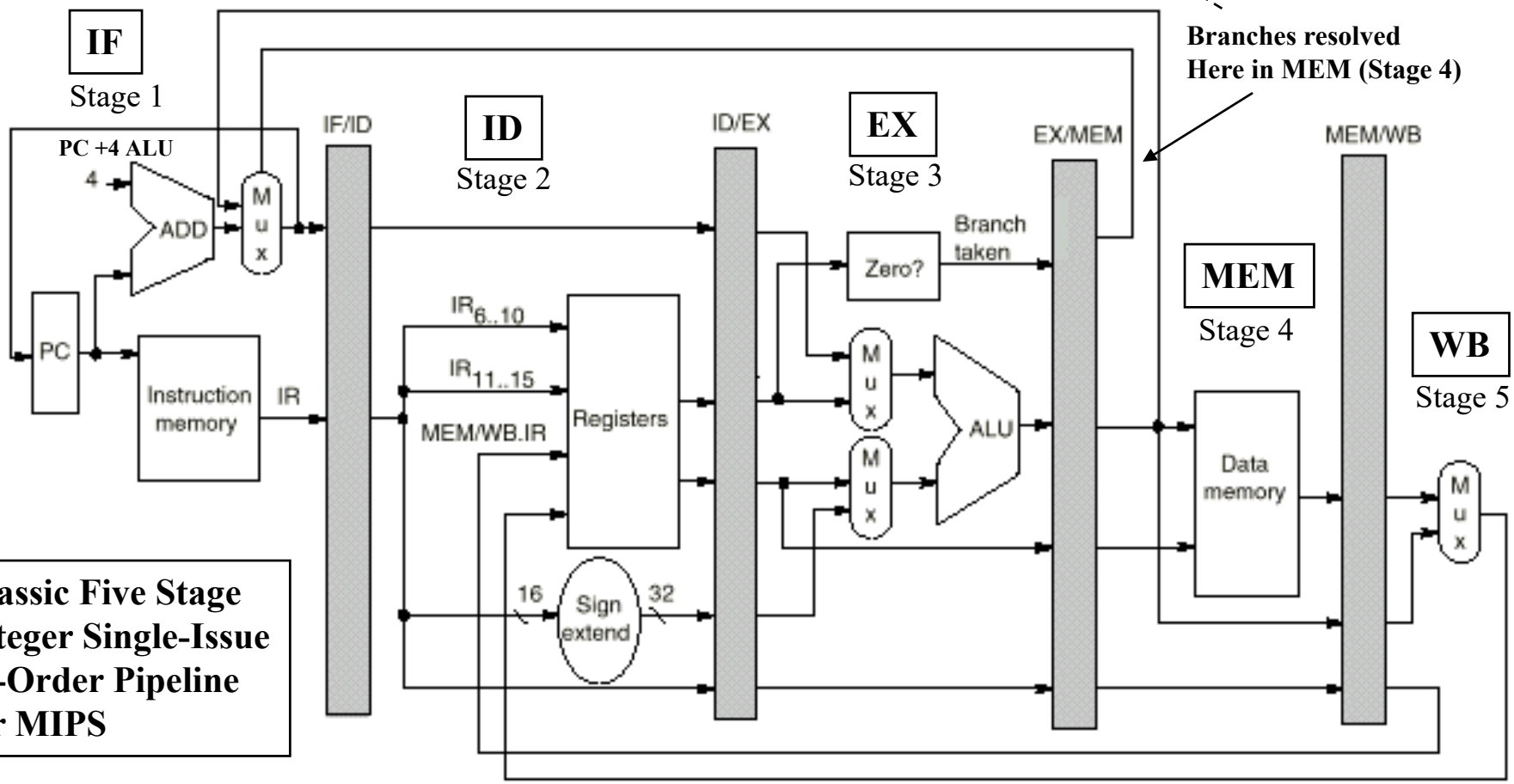
(In Appendix A) **Ideal pipeline operation without any stall cycles**

A Pipelined MIPS Integer Datapath

Data

Pipeline Version 1 (in 350): No Forwarding, Branch resolved in MEM stage

- **Assume register writes occur in first half of cycle and register reads occur in second half.**

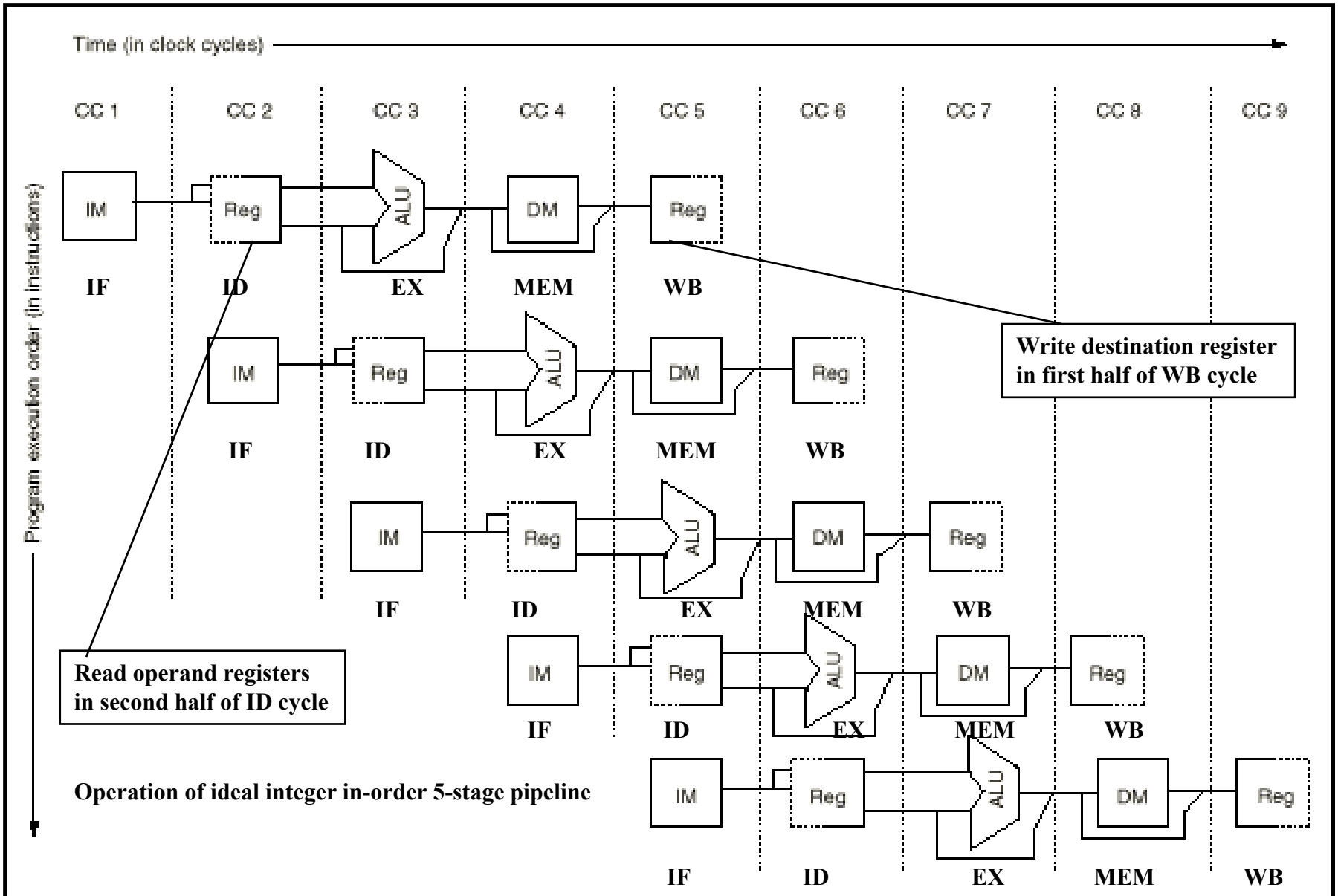


Classic Five Stage Integer Single-Issue In-Order Pipeline for MIPS

The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.

Taken Branch Penalty = 4 -1 = 3 cycles

(In Appendix A and 350)



(In Appendix A and 350)

Pipelining Performance Example

- **Example: For an unpipelined CPU:**
  - **Clock cycle = 1ns, 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 0.2 ns 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

= 1 ns x ((40% + 20%) x 4 + 40%x 5) = 1 ns x 4.4 = 4.4 ns

CPI = 4.4

CPI = 1

In the pipelined implementation five stages are used with an average instruction execution time of: 1 ns + 0.2 ns = 1.2 ns

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

= 4.4 ns / 1.2 ns = 3.7 times faster

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

Pipeline Hazards

- Hazards are situations in pipelining which may 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:

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

  Resource Not available: Hardware Component

  → Hardware structure (component) conflict

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

  - Operands

  Correct Operand (data) value

  → Operand not ready yet when needed in EX or ID

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

  Correct PC

  → Correct PC not available when needed in IF

(In Appendix A and 350)

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

For Single Issue Pipeline

- 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.

May need stall cycles to prevent hardware structures conflicts

- If a hardware 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.

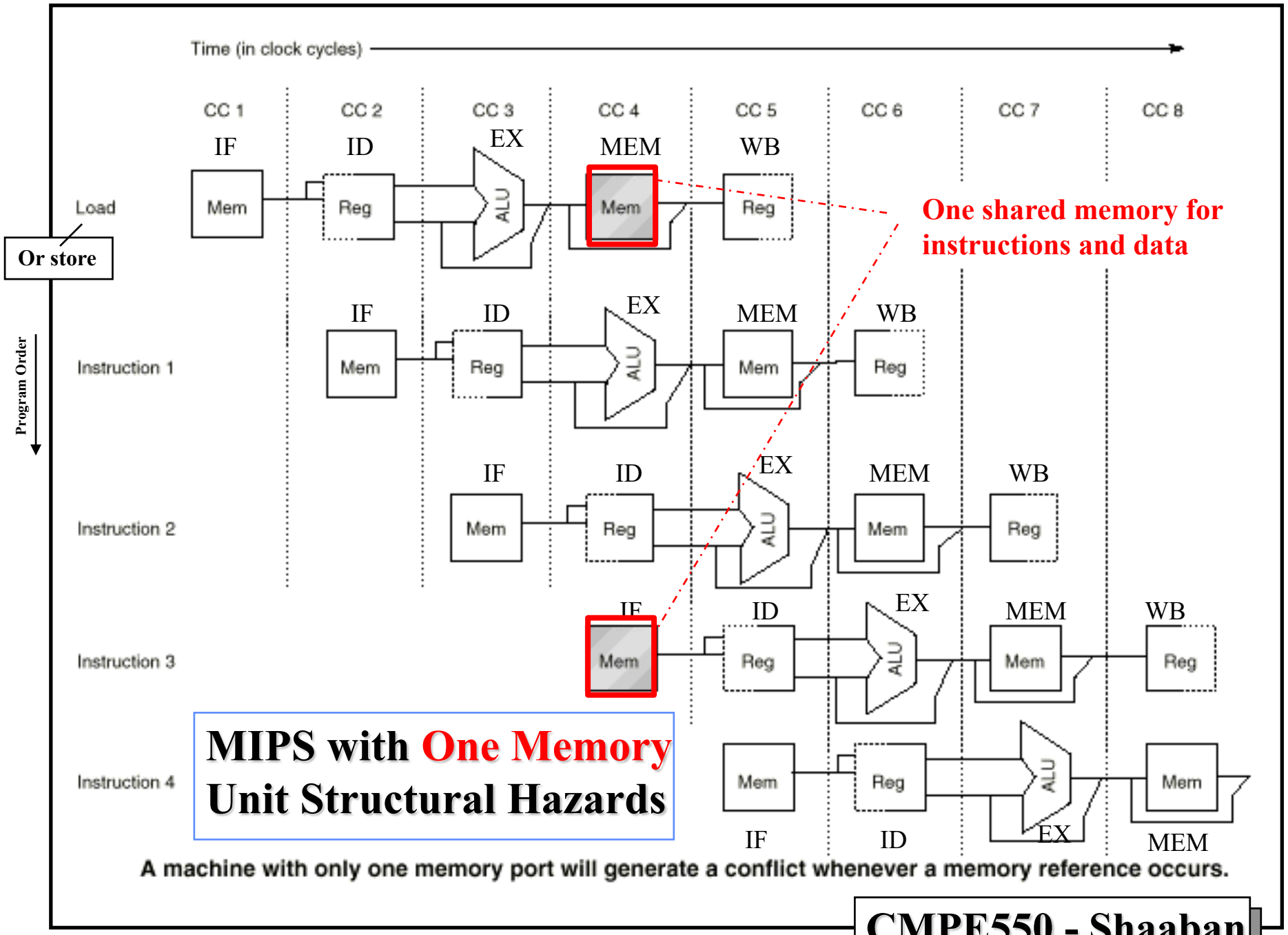
– when a pipelined machine (CPU) has a shared single-memory pipeline stage for data and instructions fetch.

→ stall the pipeline for one cycle for memory data access

i.e. Load/Store Instructions

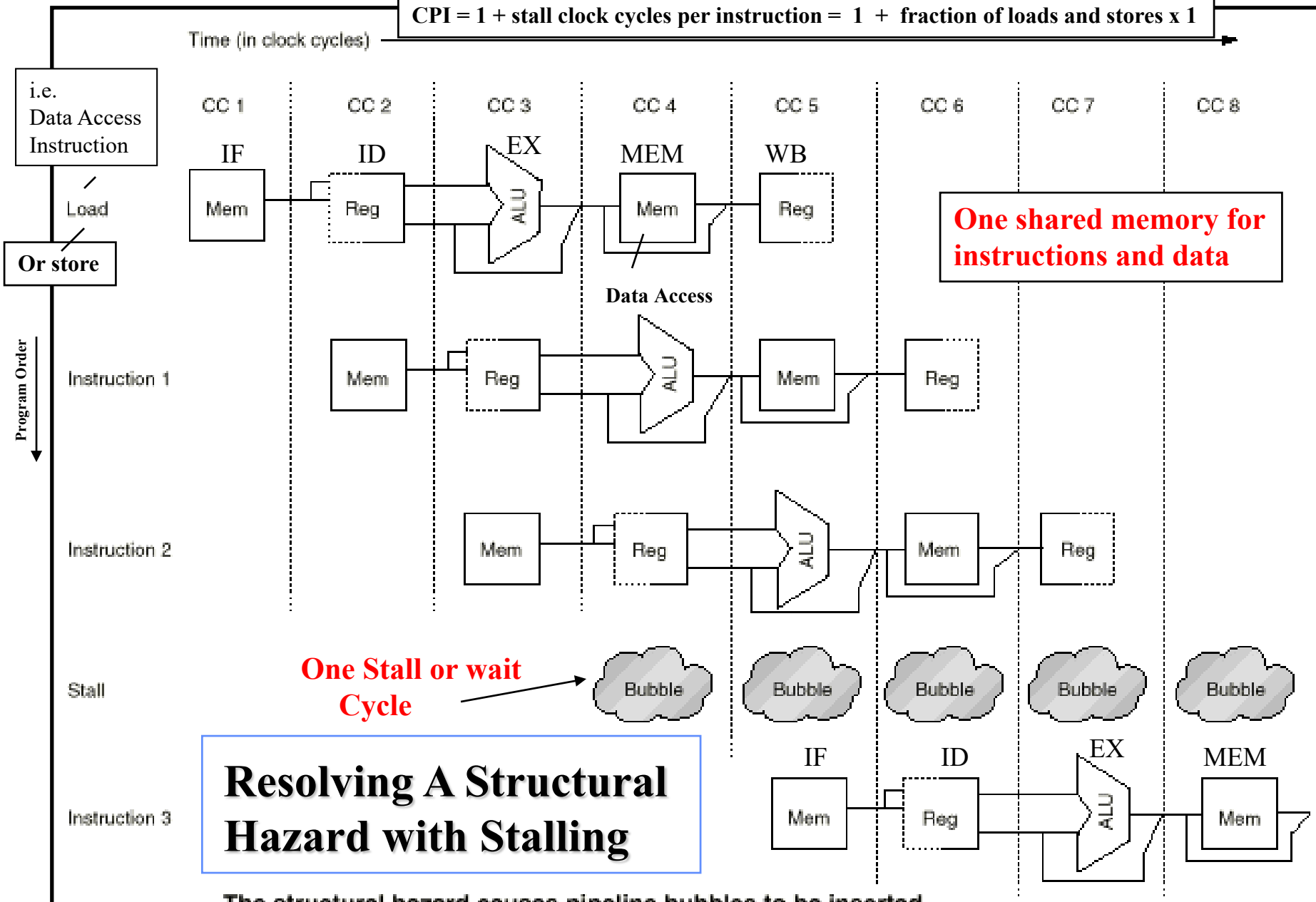
i.e A hardware component the instruction requires for correct execution is not available in the cycle needed

(In Appendix A and 350)



(In Appendix A and 350)

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



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

(In Appendix A and 350)

A Structural Hazard Example

- Given that data references are (i.e loads/stores) 40% 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 = $(1 + \underbrace{0.4 \times 1}_{\text{CPI} = 1.4}) \times \frac{\text{Clock cycle}_{\text{ideal}}}{1.05}$

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

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

(In Appendix A and 350)

Data Hazards

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

Destination

Producer of Result (data) In R1

Consumers of Result (data) From R1

```
1  DADD   R1, R2, R3
2  DSUB   R4, R1, R5
3  AND    R6, R1, R7
4  OR     R8,R1,R9
5  XOR    R10, R1, R11
```

CPI = 1 + stall clock cycles per instruction

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 DADD use the result of the DADD instruction
- DSUB, AND instructions need to be stalled for correct execution.

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

(In Appendix A and 350)

Start with an Empty Graph

∅

Program Order

Destination

Producer of Result (data) In R1

Consumers of Result (data) From R1

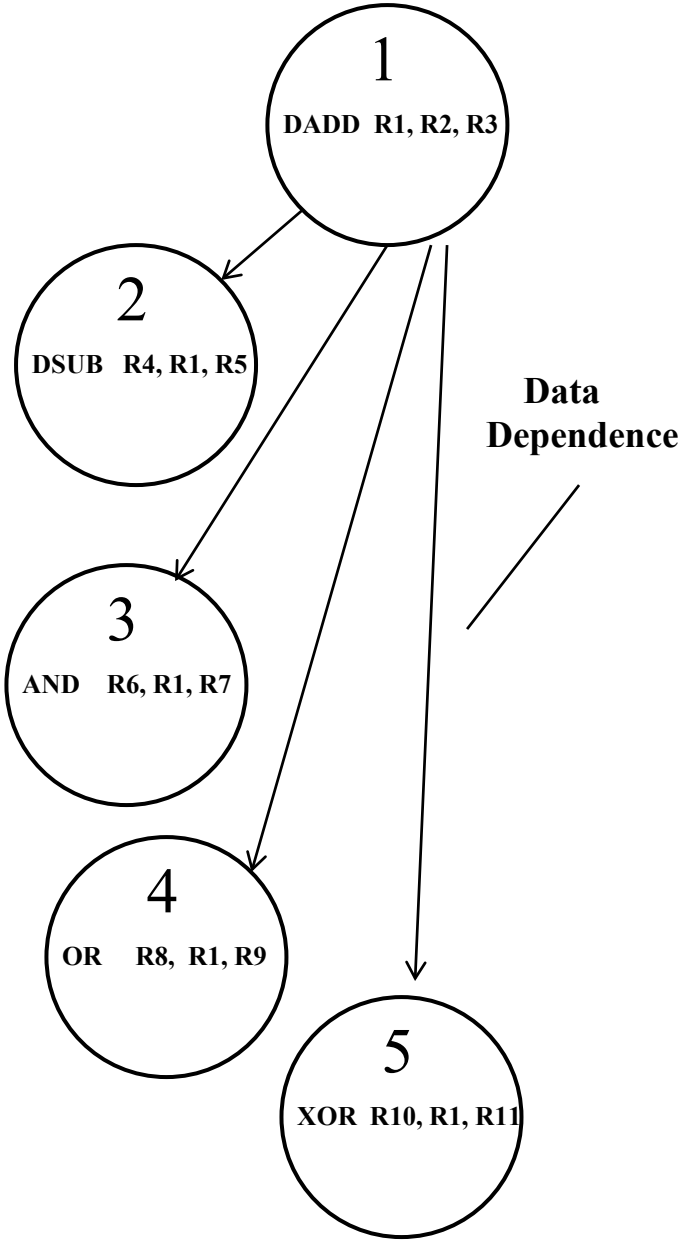
1 DADD R1, R2, R3

2 DSUB R4, R1, R5

3 AND R6, R1, R7

4 OR R8, R1, R9

5 XOR R10, R1, R11



Data Dependence

Directed Data Dependency Graph

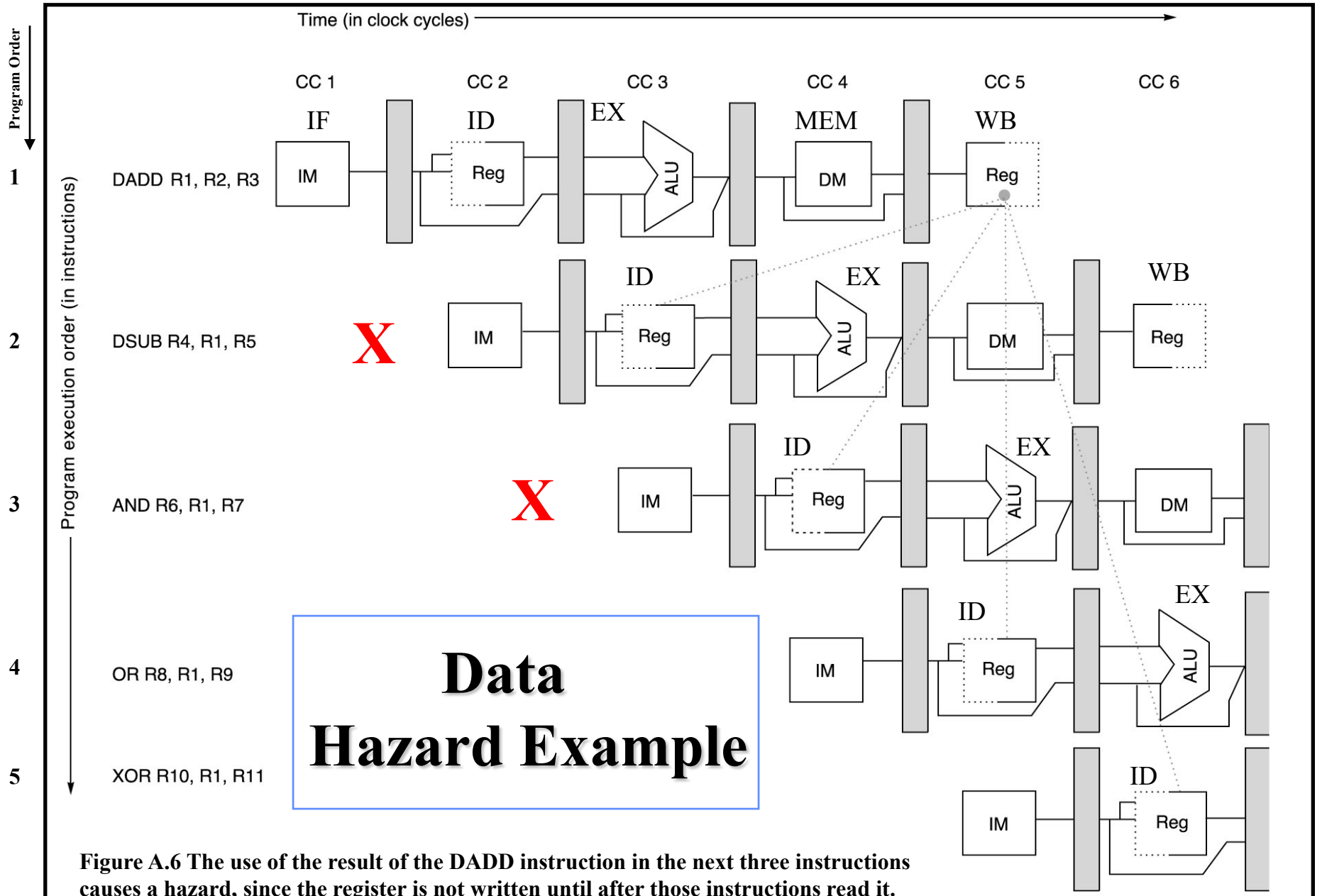


Figure A.6 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

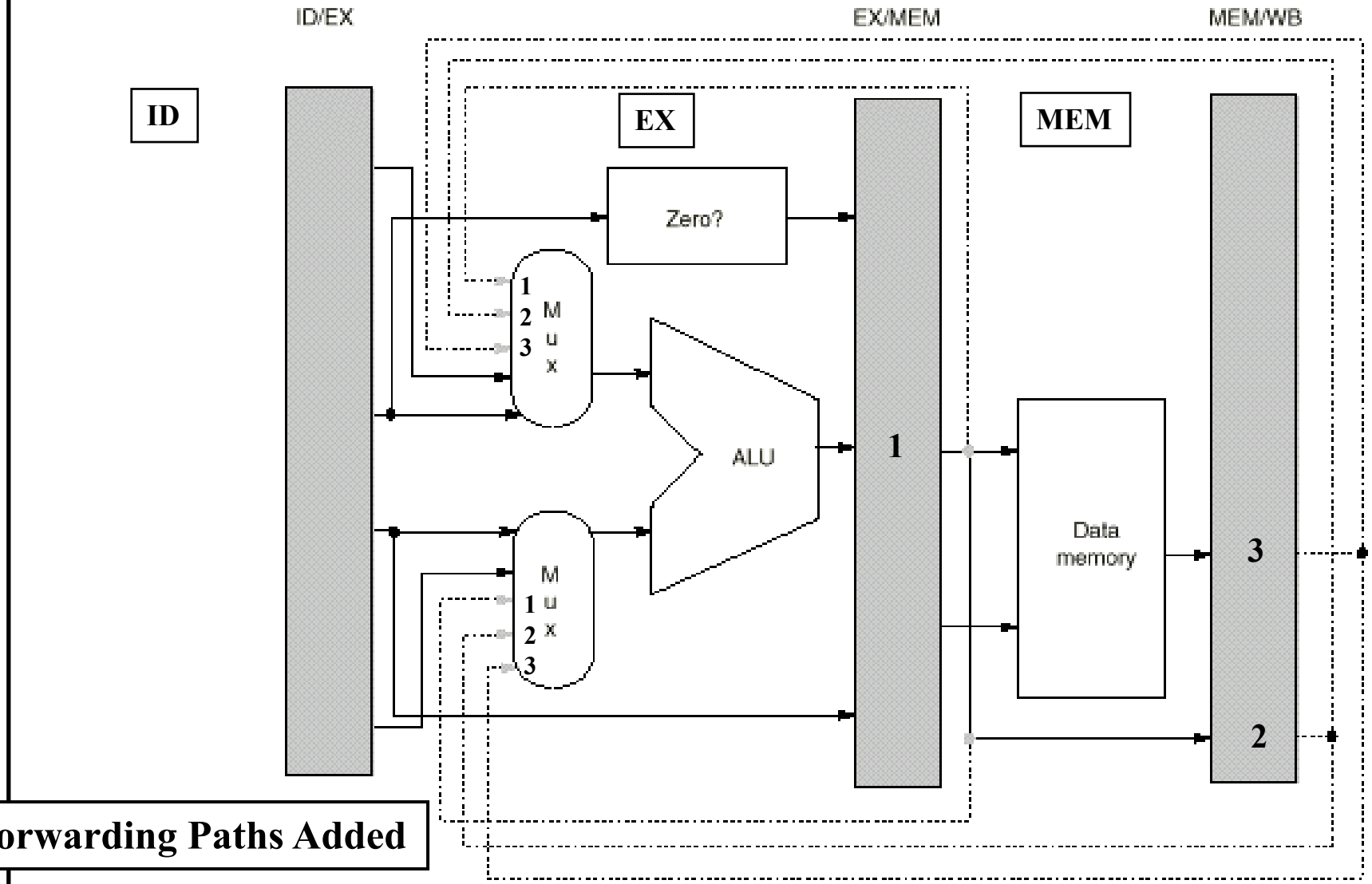
Two stall cycles are needed here (to prevent data hazard)

(In Appendix A)

Minimizing Data Hazard Stalls by Forwarding

- Data forwarding is a hardware-based technique (also called register bypassing or short-circuiting) used to eliminate or minimize data hazard stalls.
- Using forwarding hardware, the result data of an instruction is copied directly from where it is produced (ALU output register, memory read port etc.), to where subsequent instructions need it (ALU input register, data memory write port etc.)
- For example, in the MIPS integer pipeline with forwarding:
  - 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.

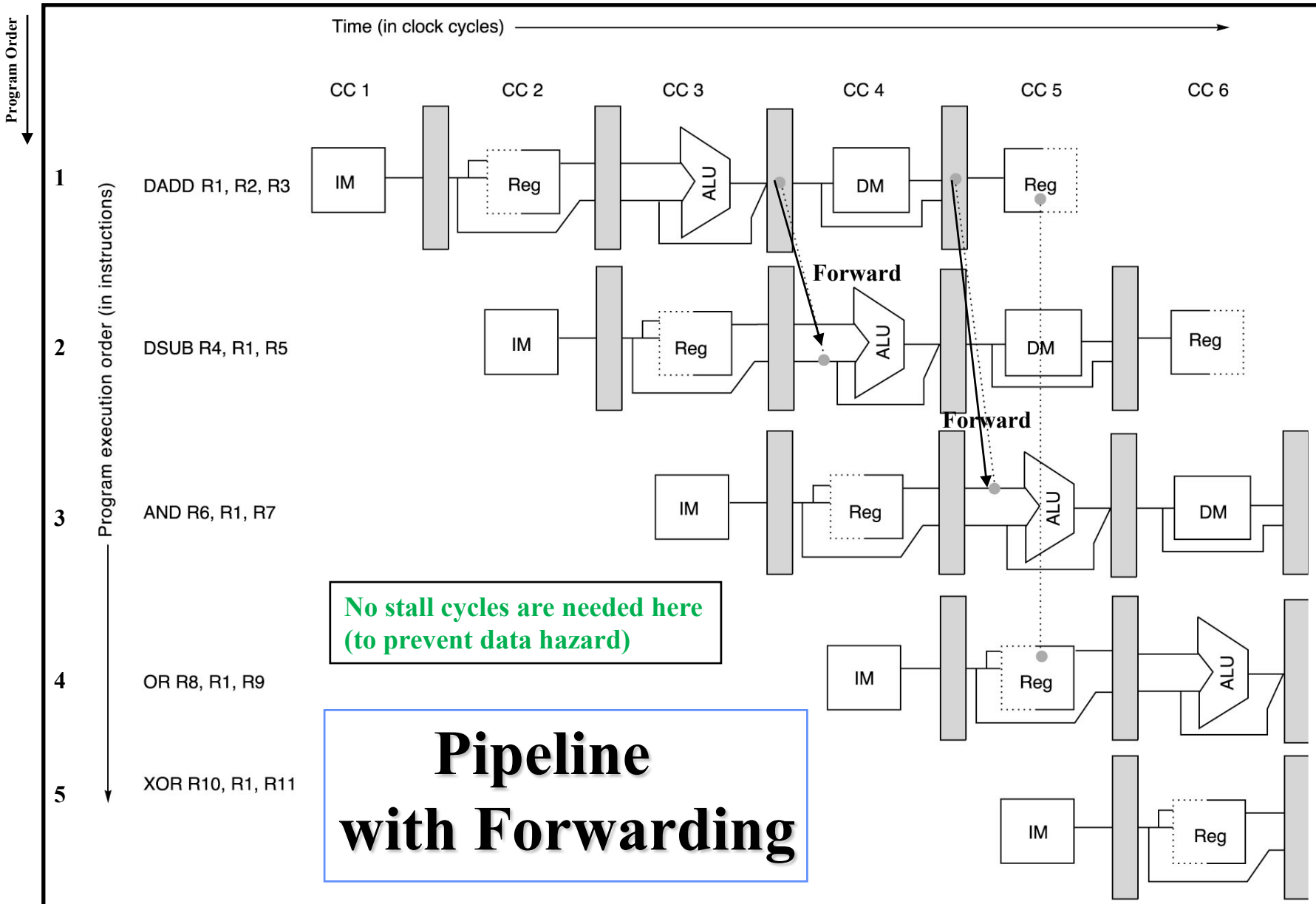
(In Appendix A and 350)



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.

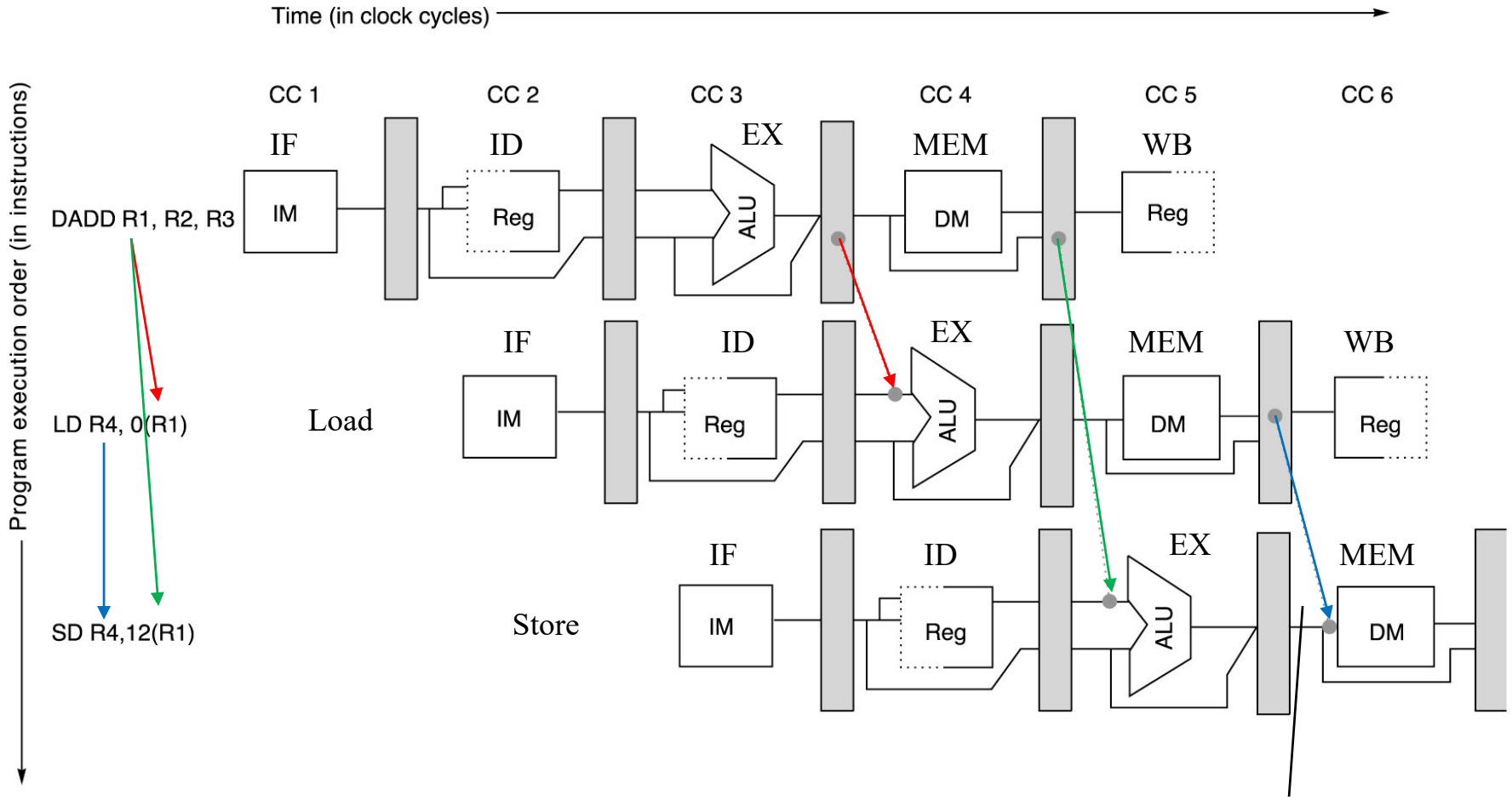
Pipeline Version 2 (in 350): With Forwarding, Branch resolved in MEM stage



A set of instructions that depend on the DADD result uses forwarding paths to avoid the data hazard

(In Appendix A and 350)

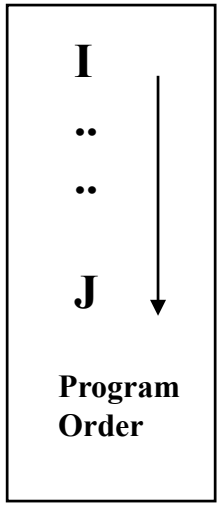
Load/Store Forwarding Example



Forwarding of operand required by store during MEM

Data Hazard Classification

Given two instructions *I*, *J*, with *I* occurring before *J* in an instruction stream (program execution order):



- **RAW (read after write):** *A true data dependence violation*

  J tried to read a source operand before *I* writes to it, so *J* incorrectly gets the old value.

- **WAW (write after write):** *A name dependence violation*

  J tries to write an operand before it is written by *I*

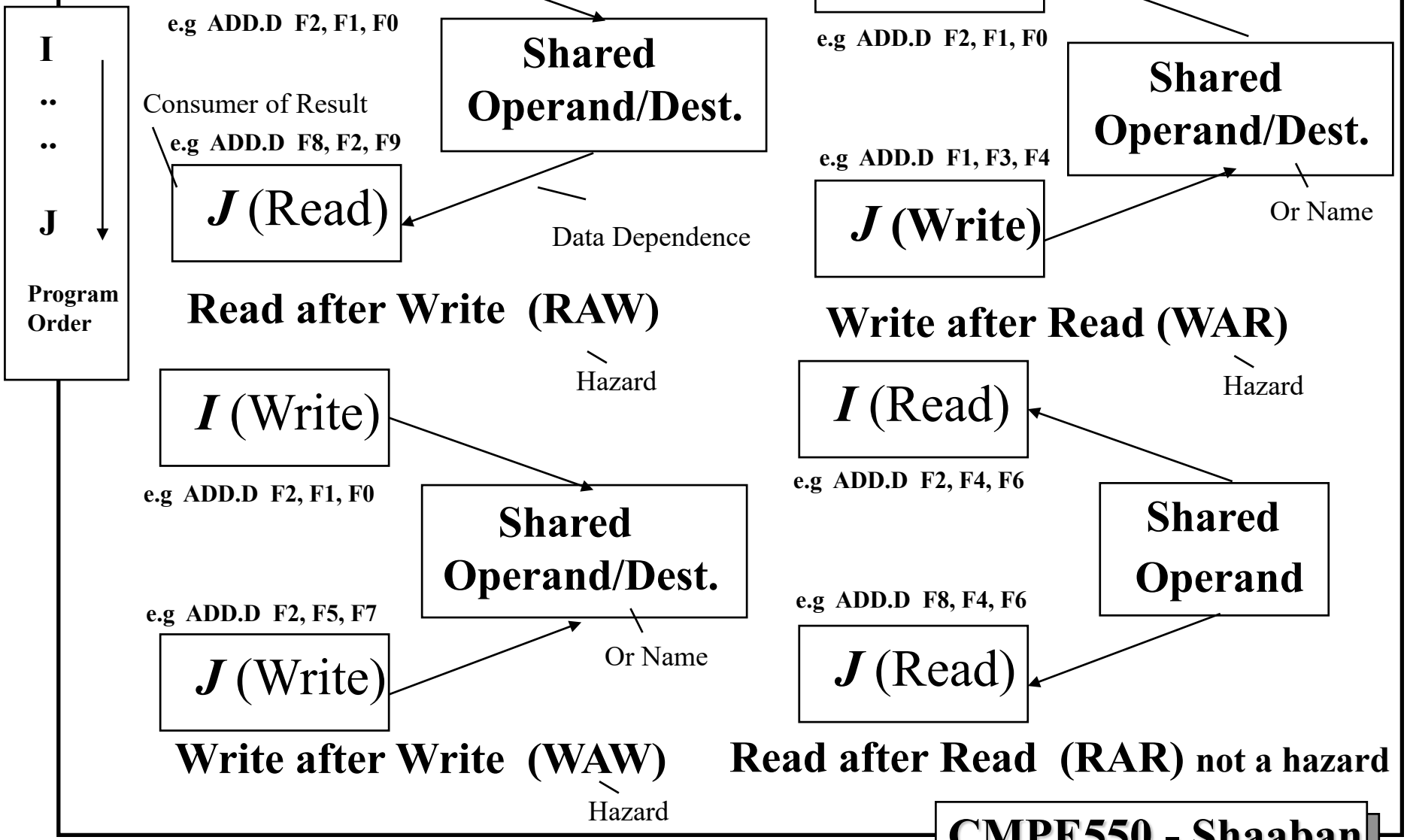
  The writes end up being performed in the wrong order.

- **WAR (write after read):** *A name dependence violation*

  J tries to write to a destination before it is read by *I*, so *I* incorrectly gets the new value.

- RAR (read after read): Not a hazard.

Data Hazard Classification



Data Hazards Present in Current MIPS Pipeline

i.e. In-Order Integer Pipeline

- **Read after Write (RAW) Hazards: Possible?**
  - Results from true data dependencies between instructions.
  - Yes possible, when an instruction requires an operand generated by a preceding instruction with distance less than four.
  - Resolved by:
    - Forwarding and/or Stalling.
- **Write after Read (WAR) Hazard:**
  - Results when an instruction overwrites the result of an instruction before all preceding instructions have read it.
- **Write after Write (WAW) Hazard:**
  - Results when an instruction writes into a register or memory location before a preceding instruction have written its result.
- **Possible? Both WAR and WAW are impossible in the current pipeline. Why?**

MIPS in-order integer pipeline

  - Pipeline processes instructions in the same sequential order as in the program.
  - All instruction operand reads are completed (in ID) before a following instruction overwrites the operand (in WB).
    - → Thus WAR is impossible in current MIPS pipeline.
  - All instruction result writes are done in the same program order.
    - → Thus WAW is impossible in current MIPS pipeline.

i.e WAW impossible because instructions reach WB stage in program order

In-Order Pipelined CPU:
All Instructions Go Through All Pipeline Stages Strictly in Program Order

RAW

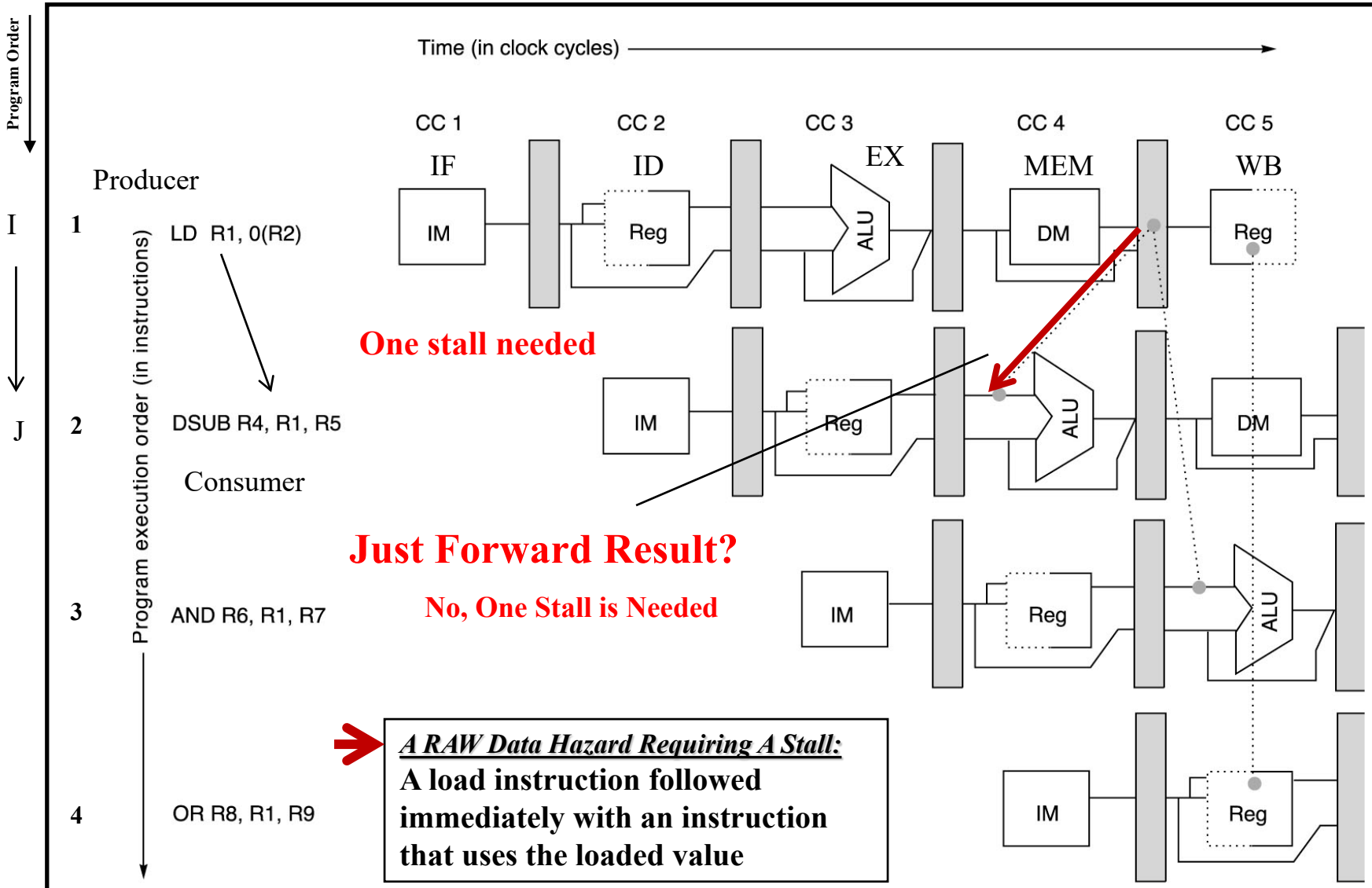
Data Hazards Requiring Stall Cycles

Even with forwarding

- In some code sequence cases, potential data hazards cannot be handled by bypassing. For example:

- The LD (load double word) instruction has the data in clock cycle 4 (MEM cycle).
- The DSUB instruction needs the data of R1 in the beginning of that cycle.
- Hazard prevented by hardware pipeline interlock causing a stall cycle.

(In Appendix A)



(In Appendix A and 350)

Hardware Pipeline Interlocks

- A hardware pipeline interlock detects a data hazard and stalls the pipeline until the hazard is cleared.
- The CPI for the stalled instruction increases by the length of the stall.
- For the Previous example, (no stall cycle):

```
LD R1, 0(R1)       IF   ID    EX     MEM     WB
DSUB R4,R1,R5           IF    ID     EX      MEM   WB
AND R6,R1,R7                  IF     ID      EX    MEM   WB
OR R8, R1, R9                        IF      ID    EX    MEM   WB
```

Incorrect Execution

With Stall Cycle:

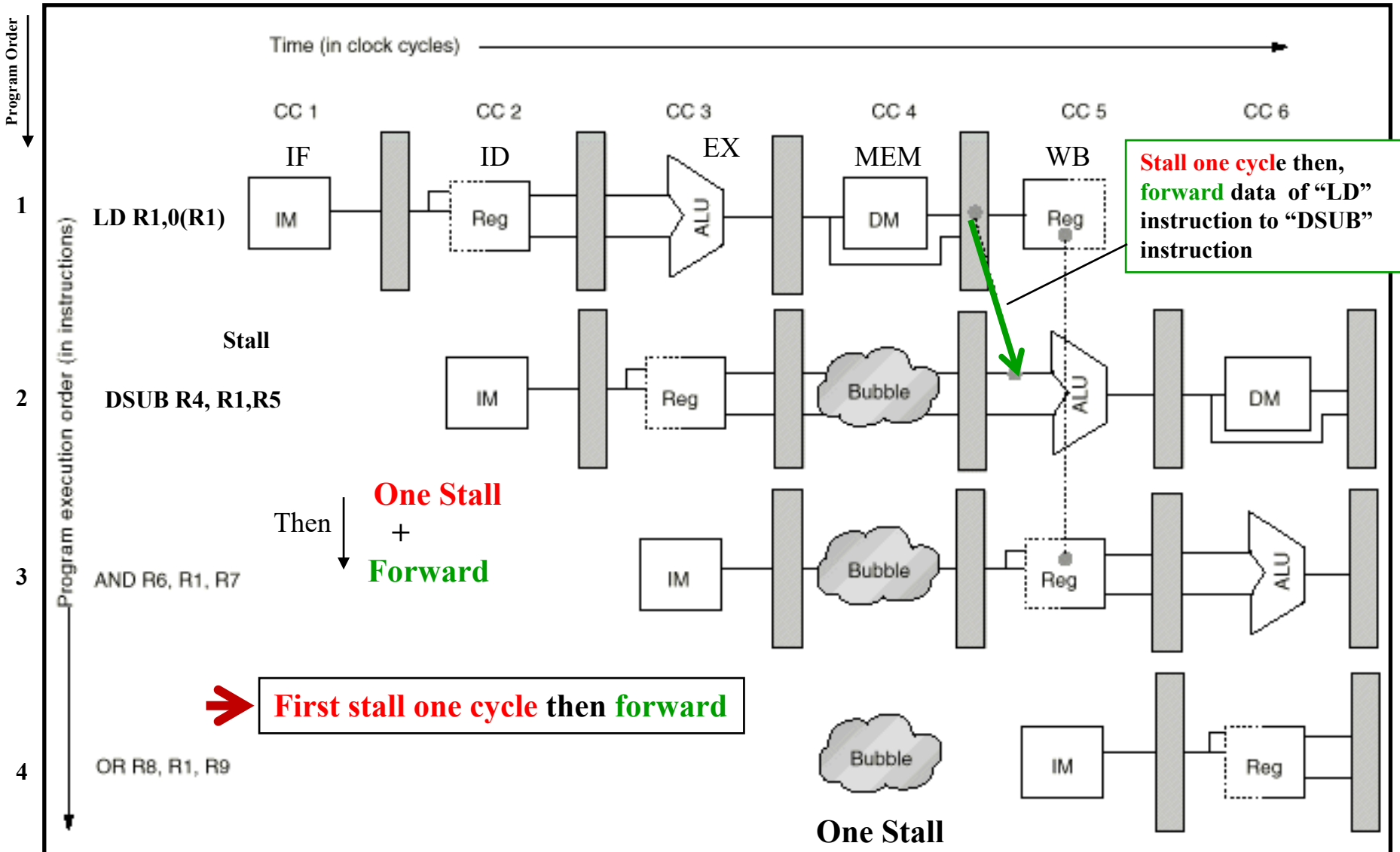
Stall + Forward

```
LD R1, 0(R1)       IF   ID    EX     MEM     WB
DSUB R4,R1,R5           IF    ID     STALL   EX    MEM   WB
AND R6,R1,R7                  IF     STALL   ID    EX    MEM   WB
OR R8, R1, R9                        STALL   IF    ID    EX    MEM   WB
```

Correct Execution

One stall

(In Appendix A)



The load interlock causes a stall to be inserted at clock cycle 4, delaying the SUB instruction and those that follow by one cycle.

A Data Hazard Requiring A Stall:

A load followed immediately by an instruction that uses the loaded value in EX stage results in a single stall cycle even with forwarding as shown.

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
Stall + forward
	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

Static Compiler Instruction Scheduling (Re-Ordering) for Data Hazard Stall Reduction

- Many types of stalls resulting from data hazards are very frequent. For example:

$$A = B + C$$

produces a stall when loading the second data value (B).

- Rather than allow the pipeline to stall, the compiler could sometimes schedule the pipeline to avoid stalls.

i.e. re-order instructions

or reduce

- Compiler pipeline or instruction scheduling involves rearranging the code sequence (instruction reordering) to eliminate or reduce the number of stall cycles.

(In Appendix A)

Static = At compilation time by the compiler
Dynamic = At run time by hardware in the CPU

Static Compiler Instruction Scheduling Example

(i.e. Re-ordering)

- For the code sequence:

1 a = b + c

2 d = e - f

a, b, c, d ,e, and f are in data memory

- Assuming loads have a latency of one clock cycle, the following code or pipeline compiler schedule eliminates stalls:

Re-order Instructions to eliminate stalls

Original code with stalls:

```
1   LD     Rb,b      e.g. 0(R1)
    LD     Rc,c
    Stall
    DADD   Ra,Rb,Rc
    SD     Ra,a
2   LD     Re,e
    LD     Rf,f
    Stall
    DSUB   Rd,Re,Rf
    SD     Rd,d
```

2 stalls for original code

Scheduled code with no stalls:

```
    LD     Rb,b
    LD     Rc,c
    LD     Re,e
    DADD   Ra,Rb,Rc
    LD     Rf,f
    SD     Ra,a
    DSUB   Rd,Re,Rf
    SD     Rd,d
```

No stalls for scheduled code

Pipeline with forwarding assumed here

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).

i.e. version 1 or 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:

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

3 stall cycles

Branch Penalty

Correct PC available here (end of MEM cycle or stage)

- Assuming we always 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

MIPS Pipeline Versions 1 and 2

(In Appendix A and 350)

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

Reducing Branch Stall Cycles

Pipeline hardware measures to reduce 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 IF ? 1-1 = 0?

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

- In MIPS branch instructions BEQZ, BNE, test a register for equality to zero.
- This can be completed in the ID cycle by moving the zero test into that cycle.

MIPS Pipeline Version # 3

- Both PCs (taken and not taken) must be computed early.
- Requires an additional adder because the current ALU is not useable until EX cycle.

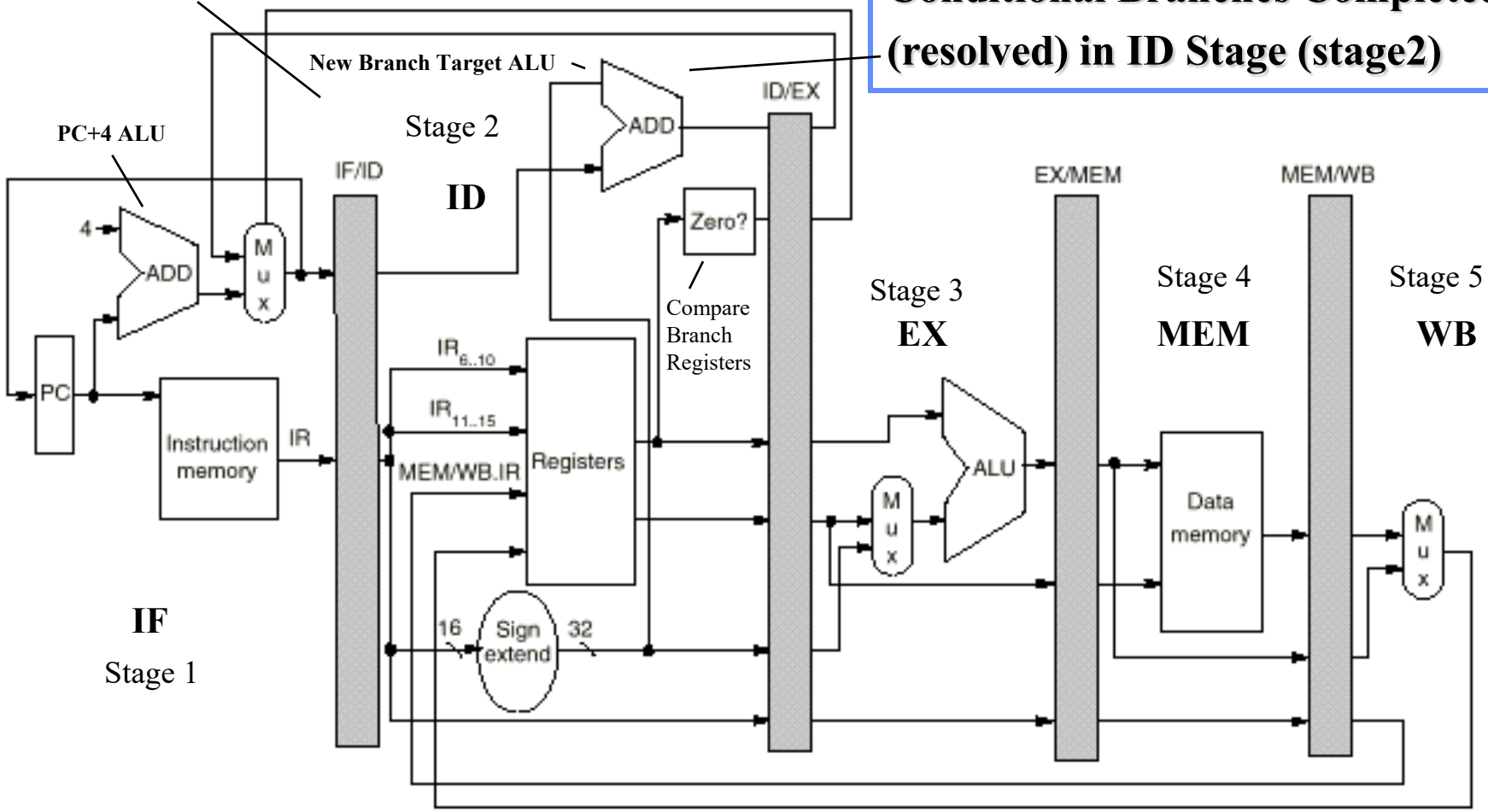
2 -1 = 1

- This results in just a single cycle stall on branches.

As opposed branch penalty = 3 cycles before

Branch resolved in stage 2 (ID)
Branch Penalty = 2 - 1 = 1 cycle

Modified MIPS Pipeline: Conditional Branches Completed (resolved) in ID Stage (stage2)



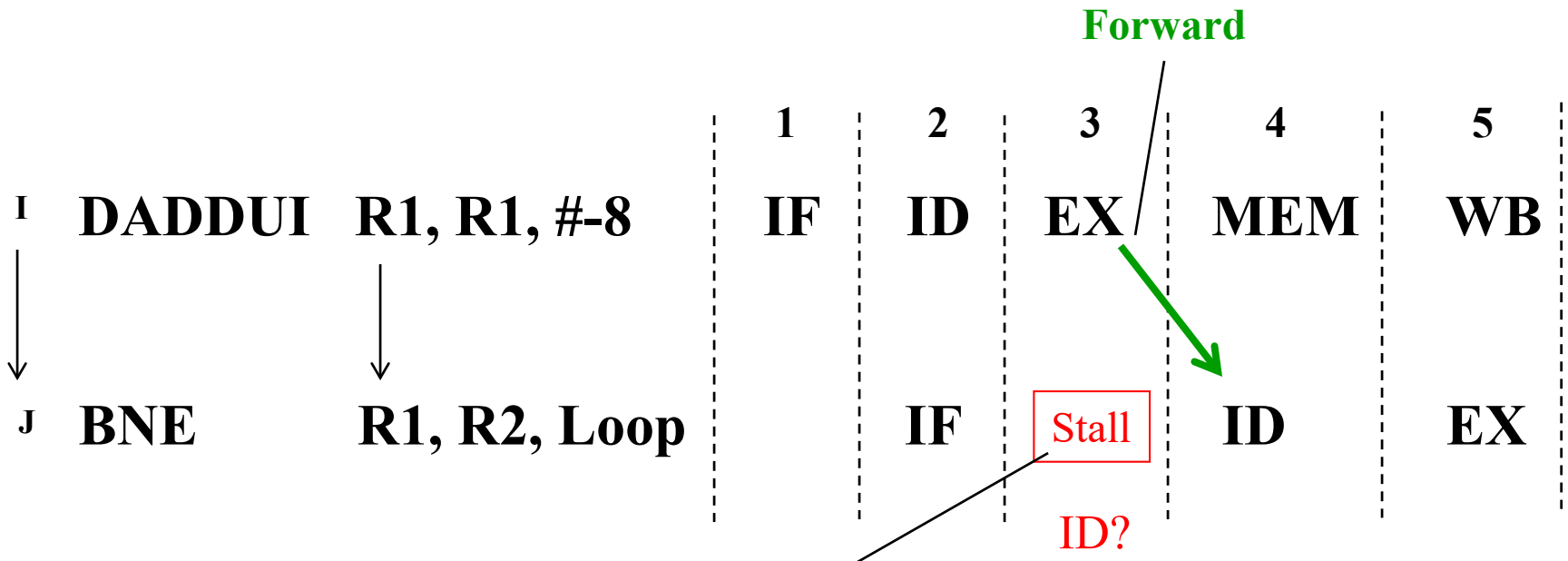
The stall from branch hazards can be reduced by moving the zero test and branch target calculation into the ID phase of the pipeline.

Pipeline Version 3 (in 350): With Forwarding, Branch resolved in ID stage

(In Appendix A and 350)

Early Resolution of Branches And Potential RAW Hazards

Example [For Pipeline Version #3]



One Stall needed
Then Forward

One stall needed even with forwarding due to resolving branch early in ID stage [Needed to prevent RAW Hazard]

Compile-Time Reduction of Branch Penalties

How to handle branches in a pipelined CPU?

1 • One scheme discussed earlier is to *flush or freeze* (i.e. always stall on a branch) the pipeline by 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).

2 • Another method is to *predict* (or assume) *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 (i.e PC+4); *stall occurs here when the branch is taken.*

Most Common

3 • Another method is to *predict that the branch is taken* (or assume) and begin fetching and executing at the target; *stall occurs here if the branch is not taken.* *(harder to implement (target?) more on this later).*

• Delayed Branch: An instruction following the branch in a branch delay slot is executed whether the branch is taken or not (part of the ISA). Supported by all RISC ISAs

Predict Branch Not-Taken Scheme

(or assume)

(most common scheme)

Not Taken Branch (no stall)

Branch Resolved Here in ID

Untaken branch instruction		IF	ID	EX	MEM	WB				
Instruction $i+1$	PC + 4		IF	ID	EX	MEM	WB			
Instruction $i+2$	PC + 8			IF	ID	EX	MEM	WB		
Instruction $i+3$					IF	ID	EX	MEM	WB	
Instruction $i+4$						IF	ID	EX	MEM	WB

Taken Branch (stall)

Branch Resolved Here in ID

Taken branch instruction		IF	ID	EX	MEM	WB				
Instruction $i+1$	PC + 4	Stall	IF	**idle**	**idle**	**idle**	**idle**			
Branch target	Target			IF	ID	EX	MEM	WB		
Branch target + 1					IF	ID	EX	MEM	WB	
Branch target + 2						IF	ID	EX	MEM	WB

Assuming the MIPS pipeline with reduced branch penalty = 1

i.e. Pipeline Version 3

The predict-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom).

Stall when the branch is taken

i.e. Fraction of Dynamic Instruction Count Executed I

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

CPI = 1 + stall clock cycles per instruction

Pipeline Performance Example

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

Type	Frequency	i.e. Fraction of All Instructions Executed I
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 a branch not-taken scheme?**

i.e Pipeline 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

Static Compiler Branch Prediction

- Static Branch prediction encoded in branch instructions using one prediction bit = 0 = Not Taken, = 1 = Taken
  - Must be supported by ISA, Ex: HP PA-RISC, PowerPC, UltraSPARC
- Two basic methods exist to statically predict branches at compile time:

Branch Encoding	X

X = Static Prediction bit
X= 0 Not Taken X = 1 Taken

How?

1 By examination of program behavior and the use of information collected from earlier runs of the program.

  - For example, a program profile may show that most forward branches and backward branches (often forming loops) are taken. The simplest scheme in this case is to just predict the branch as taken.

Program profile-based static branch prediction

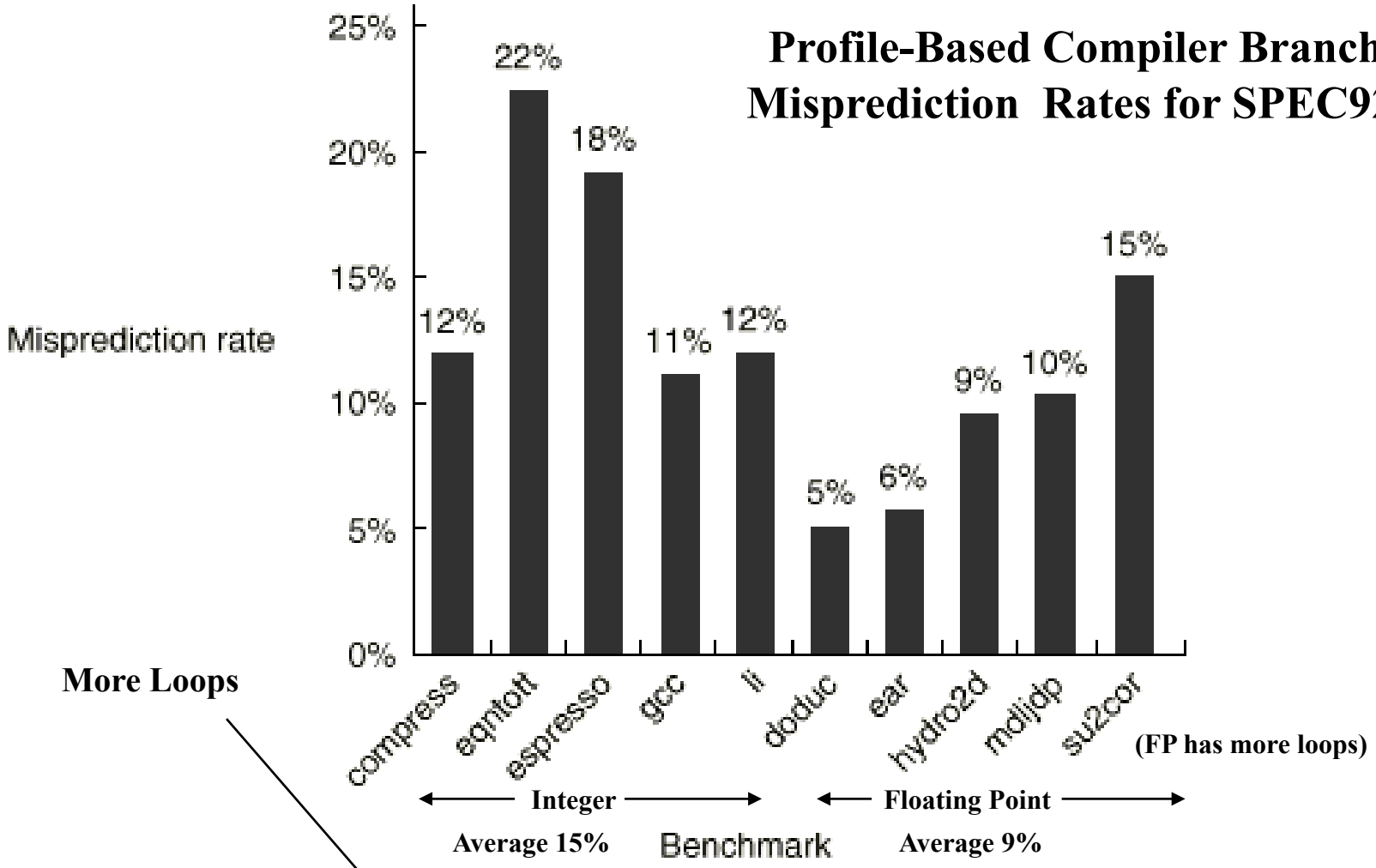
2 To predict branches on the basis of branch direction, choosing backward branches as taken and forward branches as not taken.

Loop?

Static = By the compiler Dynamic = By hardware in the CPU

Static Branch Prediction Performance:

Profile-Based Compiler Branch Misprediction Rates for SPEC92

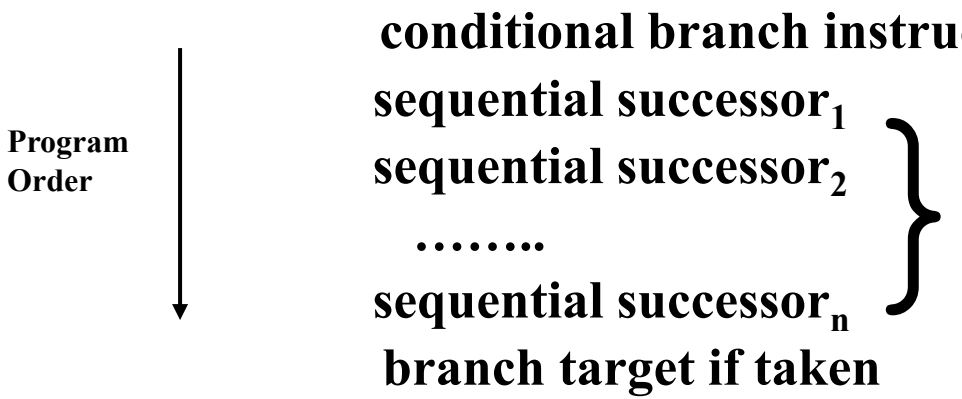


Misprediction rate for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%.

ISA Reduction of Branch Penalties:
Delayed Branch (action)

i.e. ISA Support Needed

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



These instructions in branch delay slots are always executed regardless of branch Direction/outcome

- The sequential successor instruction are said to be in the branch delay slots. <u>These instructions are executed whether or not the branch is taken.</u>
- In Practice, all machines (ISA) that utilize delayed branching have
→ <u>a single instruction delay slot</u>. (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

Not Taken Branch (no stall)

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)	PC + 4	IF	ID	EX	MEM	WB			
Instruction i + 2	PC + 8		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)	PC + 4	IF	ID	EX	MEM	WB			
Branch target	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.

➔ **Single Branch Delay Slot Used** **All RISC ISAs**

Assuming branch penalty = 1 cycle

i.e. Pipeline Version # 3

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:**

Most Common

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

e.g From Body of a loop

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

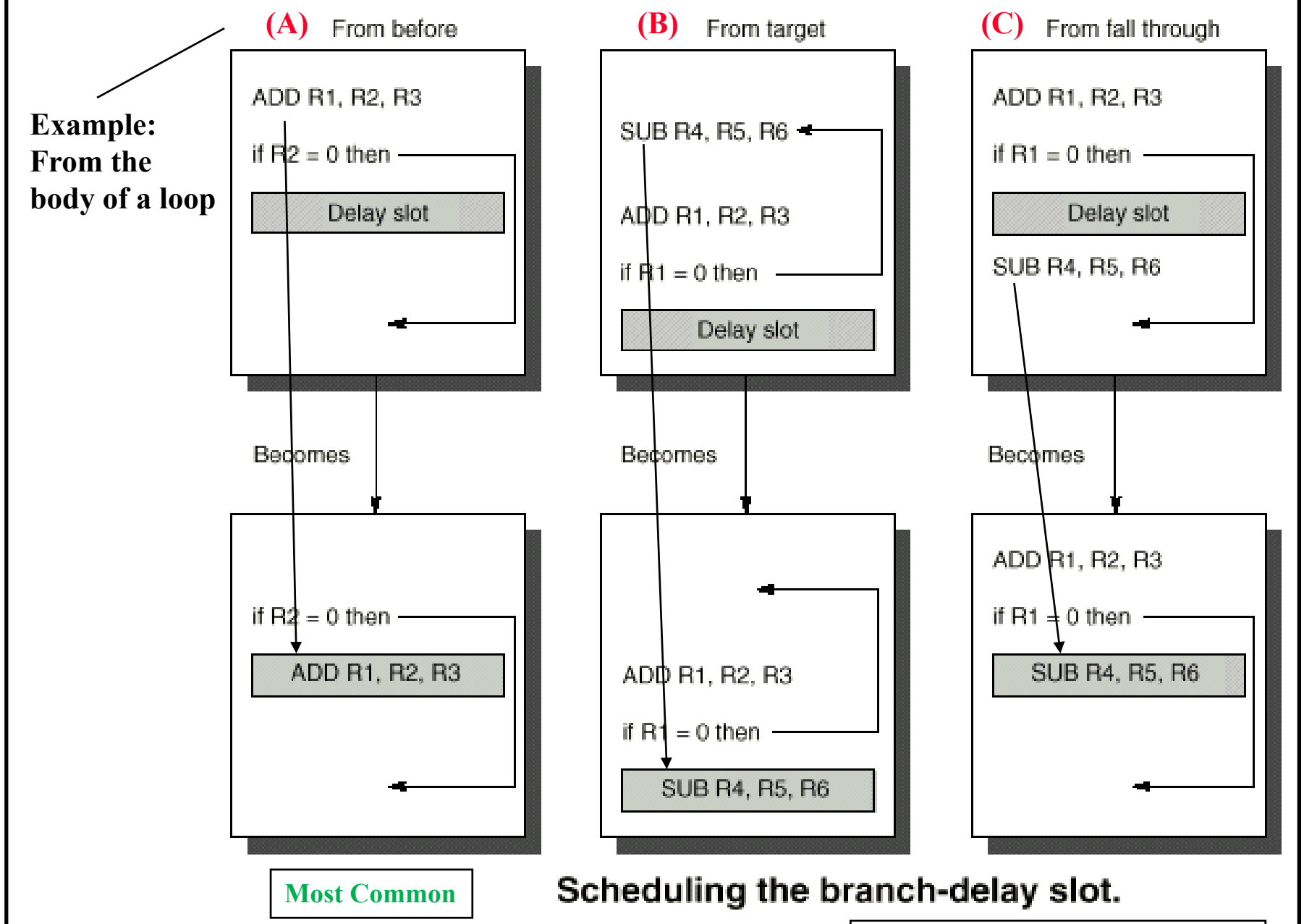
Hard to Find

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.

The performance and usability of cases B, C is improved by using a canceling or nullifying branch.



Example: From the body of a loop

Most Common

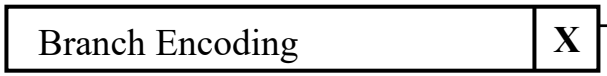
Scheduling the branch-delay slot.

(In Appendix A)

Branch-delay Slot: Canceling Branches

(AKA Canceling Delayed Branch Action Slot)

- In a canceling branch, a static compiler branch direction prediction is included with the branch-delay slot instruction.



X = Static Prediction bit
X= 0 Not Taken X = 1 Taken

- When the branch goes as predicted, the instruction in the branch delay slot is executed normally.
- When the branch does not go as predicted the instruction is turned into a no-op (i.e. cancelled).

Why?

- Canceling branches eliminate the conditions on instruction selection in delay instruction strategies B, C
- The effectiveness of this method depends on whether we predict the branch correctly.

Scheduling strategy	Requirements	Improves performance when?
(a) From before branch	Branch must not depend on the rescheduled instructions.	Always.
(b) From target	Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions.	When branch is taken. May enlarge program if instructions are duplicated.
(c) From fall through	Must be OK to execute instructions if branch is taken.	When branch is not taken.

Pipeline Version # 3 Assumed Here

Delayed-branch scheduling schemes and their requirements.

Branch Goes Not As Predicted

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)	PC + 4	IF	ID	idle	idle	idle			
Instruction i + 2	PC + 8		IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Cancelled
Stall or No-OP

Branch Goes As Predicted

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)	PC + 4	IF	ID	EX	MEM	WB			
Branch target	Target		IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Normal
No Stall

behavior of a predicted-taken cancelling branch depends on whether the branch is taken or not.

Branch Predicted Taken By Compiler

Canceling Branch Example – Predicted Taken

Performance Using Canceling Delay Branches

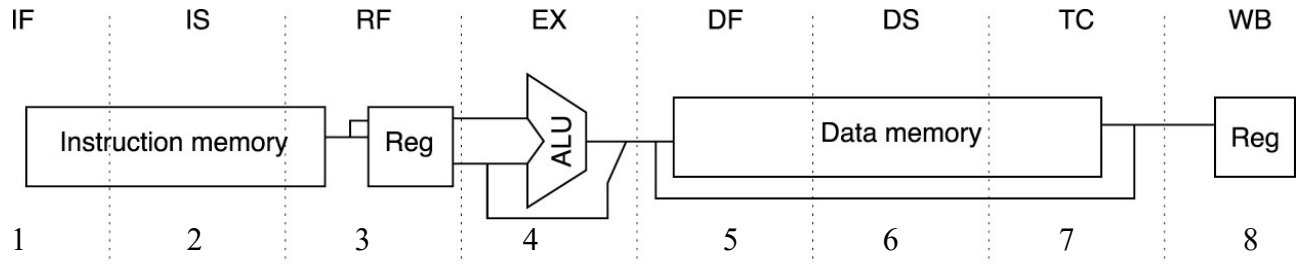
Benchmark	% conditional branches	% conditional branches with empty slots	% conditional branches that are cancelling	% cancelling branches that are cancelled	% branches with cancelled delay slots	Total % branches with empty or cancelled delay slot
compress	14%	18%	31%	43%	13%	31%
eqntott	24%	24%	50%	24%	12%	36%
espresso	15%	29%	19%	21%	4%	33%
gcc	15%	16%	33%	34%	11%	27%
li	15%	20%	55%	48%	26%	46%
Integer average	17%	21%	38%	34%	13%	35%
doduc	8%	33%	12%	62%	8%	41%
ear	10%	37%	36%	14%	5%	42%
hydro2d	12%	0%	69%	24%	16%	17%
mdljdp2	9%	0%	86%	10%	8%	8%
su2cor	3%	7%	17%	57%	10%	17%
FP average	8%	16%	44%	34%	9%	25%
Overall average	12%	18%	41%	34%	11%	30%

Delayed and cancelling delay branches for MIPS allow branch hazards to be hidden 70% of the time on average for these 10 SPEC benchmarks.

➔ **70% Static Prediction Accuracy**

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.**

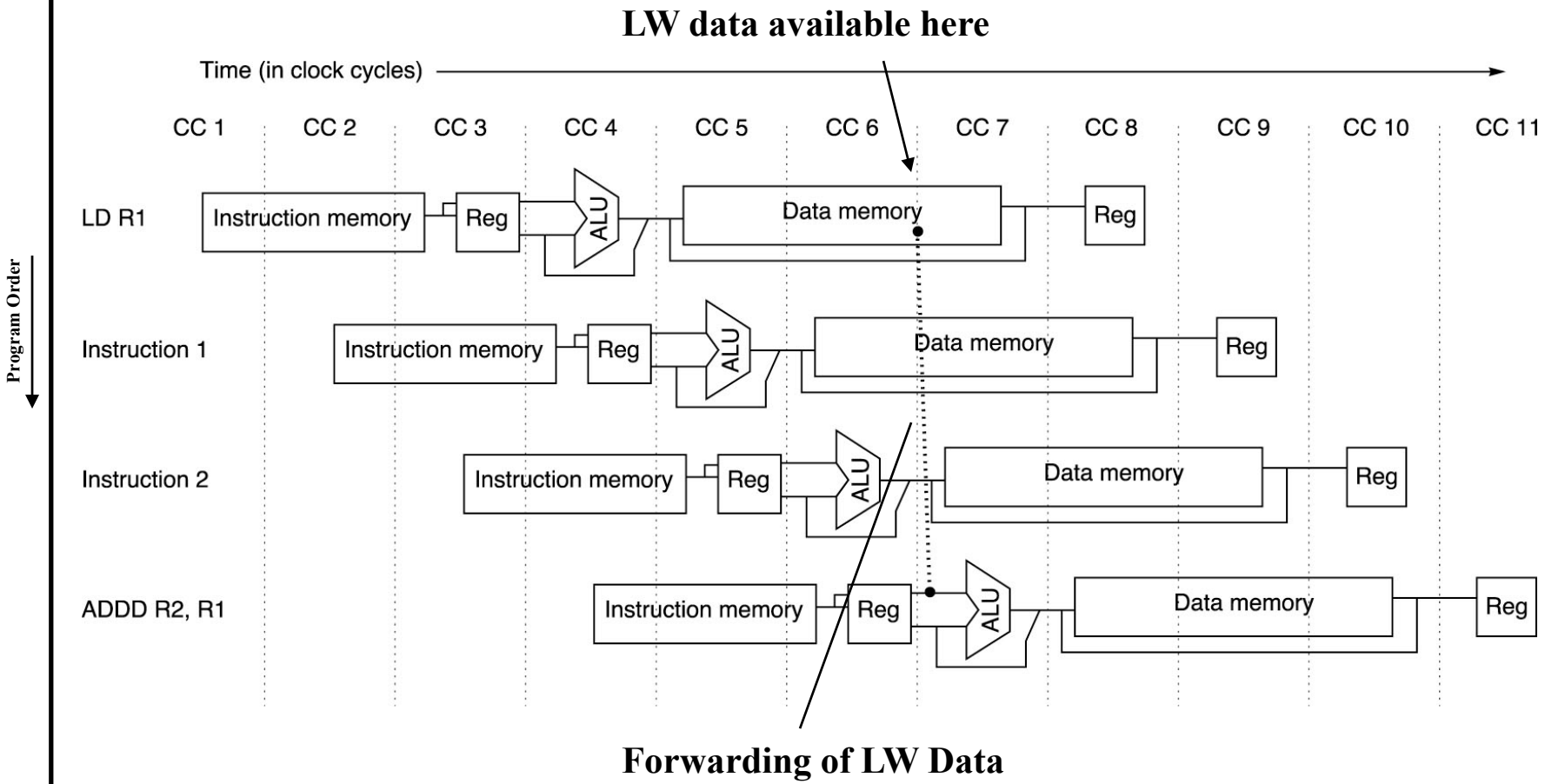
- **Branch resolved in stage 4. Branch Penalty = 3 cycles if taken (2 with branch delay slot)**

(In Appendix A)

Deeper Pipelines = More Stall Cycles and Higher CPI

T = I x CPI x C

MIPS R4000 Example



- 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

(In Appendix A)

Pipelining and Handling of Exceptions

- Exceptions are events that usually occur in normal program execution where the normal execution order of the instructions is changed (often called: interrupts, faults).
- Types of exceptions include:
  - I/O device request
  - Invoking an operating system service
  - Tracing instruction execution
  - Breakpoint (programmer-requested interrupt).
  - Integer overflow or underflow
  - FP anomaly
  - Page fault (not in main memory)
  - Misaligned memory access
  - Memory protection violation
  - Undefined instruction (Bad Opcode)
  - Hardware malfunctions

Characteristics of Exceptions

- **Synchronous vs. asynchronous:**

  Synchronous: occurs at the same place with the same data and memory allocation

  Asynchronous: Caused by devices external to the processor and memory.

- **User requested vs. coerced:**

  User requested: The user task requests the event.

  Coerced: Caused by some hardware event (i.e. I/O device request).

- **User maskable vs. user nonmaskable:**

  User maskable: Can be disabled by the user task using a mask.

- **Within vs. between instructions:**

  Whether it prevents instruction completion by happening in the middle of execution.

- **Resuming vs. terminating:**

  Terminating: The program execution always stops after the event.

  Resuming: the program continues after the event. The state of the pipeline must be saved to handle this type of exception. The pipeline is restartable in this case.

Handling of Resuming Exceptions

- A resuming exception (e.g. a virtual memory page fault) usually requires the intervention of the operating system.

  To handle the exception

- The pipeline must be safely shut down and its state saved for the execution to resume after the exception is handled as follows:

  OS/Kernel Privileged Instruction

  1. Force a trap instruction into the pipeline on the next IF.
  2. Turn of all writes off for the faulting instruction and all following instructions in the pipeline. Place zeroes into pipeline latches (to turn off writes) starting with the instruction that caused the fault to prevent state changes.
  3. Invoke the OS exception handling routine of the operating system saves the PC of the faulting instruction and other state data to be used to return from the exception.

i.e. save program state

Precise Exception Handling

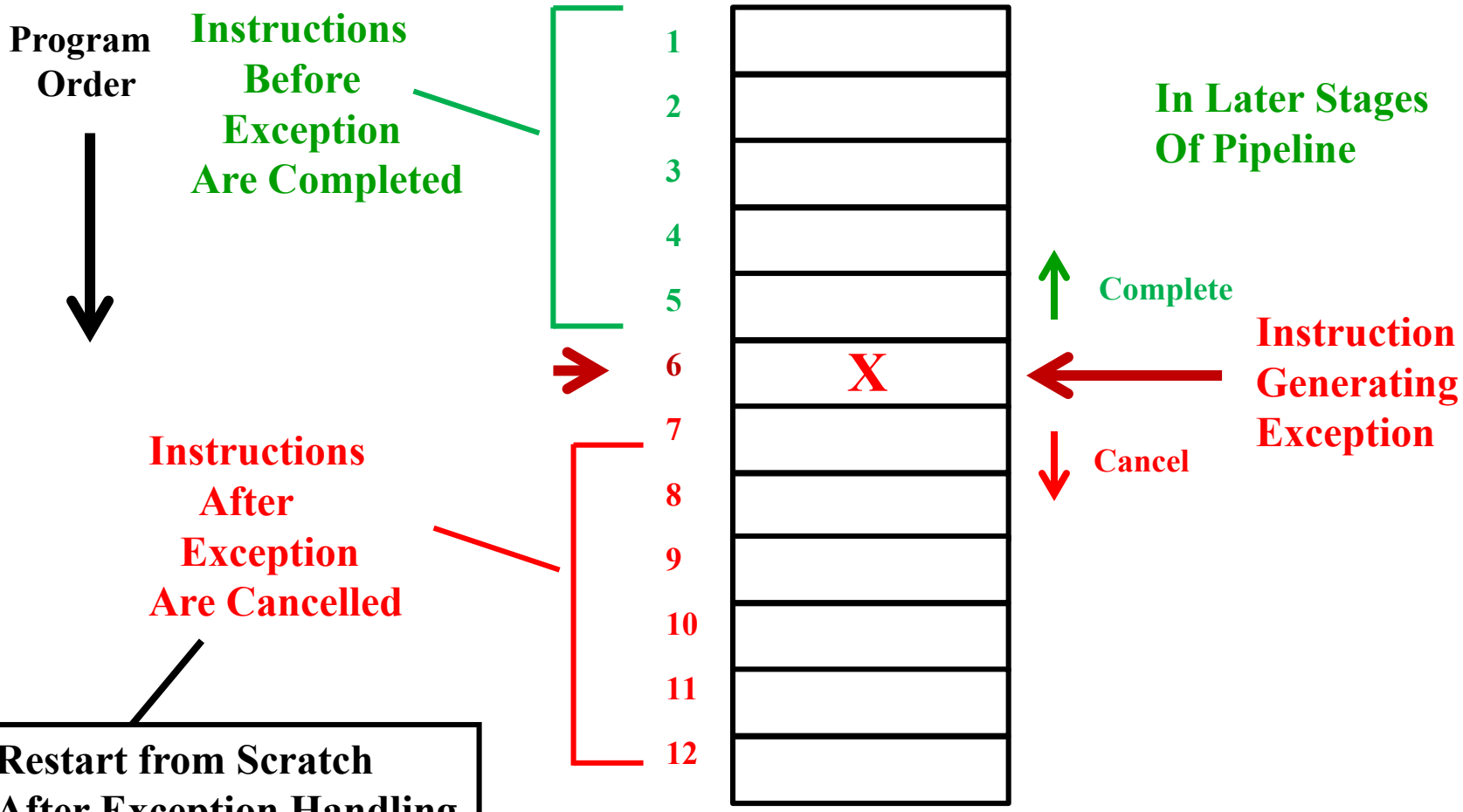
- When using delayed branches, as many PCs as the the length of the branch delay plus one need to be saved and restored to restore the state of the machine.
- After the exception has been handled special instructions are needed to return the machine to the state before the exception occurred (RFE, Return to User code in MIPS).
- ***Precise exceptions*** and handling imply that a pipeline is partially stopped so the instructions just before the faulting instruction are **completed** and those after it are **cancelled** and restarted from scratch.

After handling the exception
(i.e. As if processor is not pipelined)

- Machines with arithmetic trap handlers and demand paging must support precise exceptions.

i.e. Precise exception handling imply handling exceptions as if the processor is not pipelined

Precise Exception Handling



Exceptions in MIPS Integer Pipeline

- The following represent problem exceptions for the MIPS 5 pipeline stages:

IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation.
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data load fetch; misaligned memory access; memory-protection violation
WB	None

Program Order

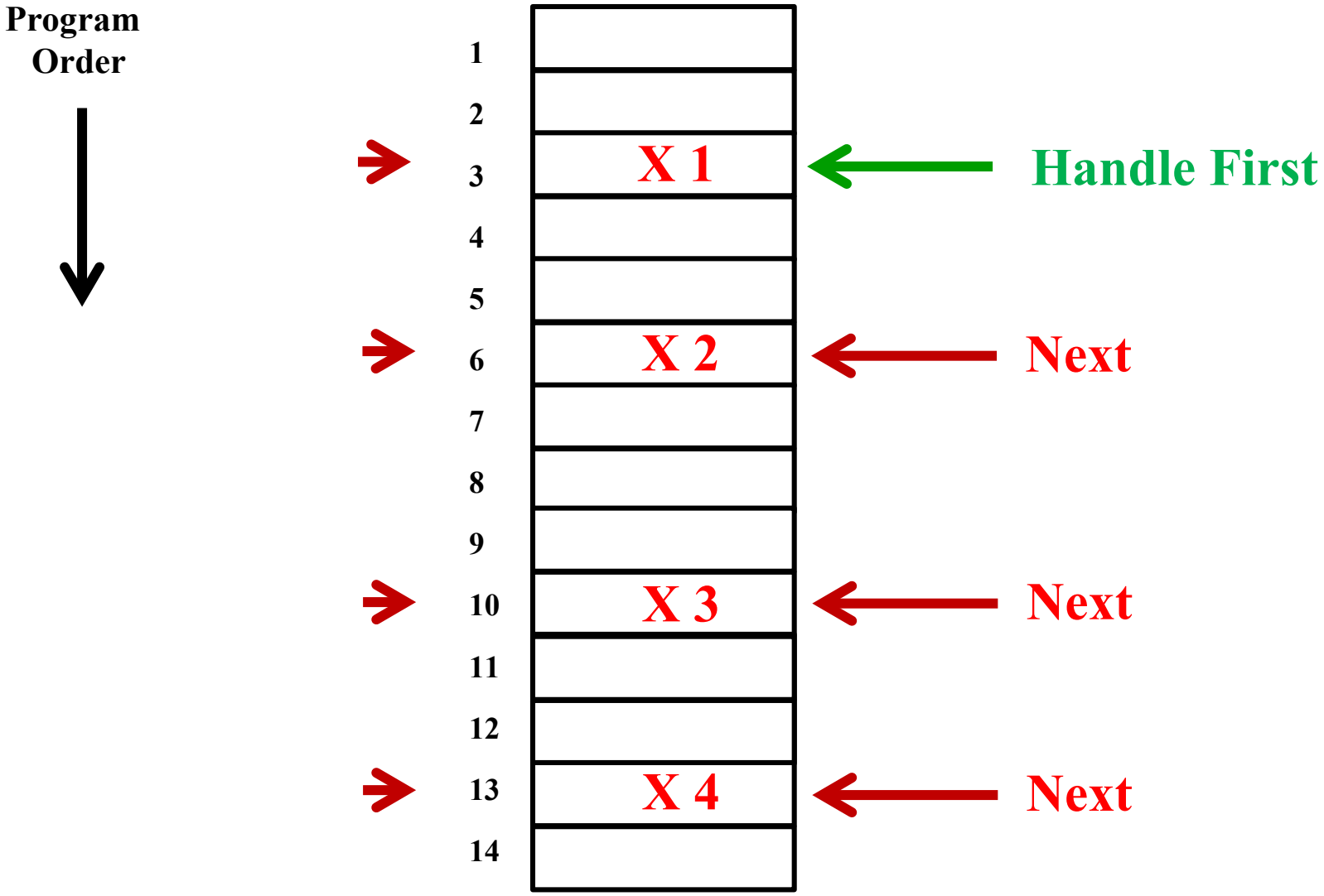
- Example: LD IF ID EX MEM WB
  DADD IF ID EX MEM WB

  can cause a data page fault and an arithmetic exception at the same time (LD in MEM and DADD in EX)

Handled by dealing with data page fault first and then restarting execution, then the second exception will occur but not the first.

i.e. handle exceptions in program order one at a time (as if processor is not pipelined)

Exception Handling Order



i.e. handle exceptions in program order one at a time
(as if processor is not pipelined)

Precise Exception Handling in MIPS
(i.e MIPS Integer Single-Issue In-Order Pipeline)

How?

1

2

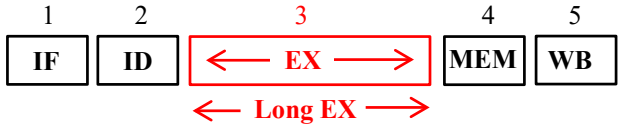
3

- The instruction pipeline is required to handle exceptions of instruction *i* before those of instruction *i+1* (i.e in program order)
- The hardware posts all exceptions caused by an instruction in an exception status vector associated with the instruction which is carried along with the instruction as it goes through the pipeline.
- Once an exception indication is set in the vector, any control signals that cause a data value write is turned off. (For the following instructions)
- When an instruction enters WB the vector is checked, if any exceptions are posted, they are handled in the order they would be handled in an unpipelined machine. (i.e in program order)
- Any action taken in earlier pipeline stages is invalid but cannot change the state of the machine since writes where disabled.

i.e. by later instructions in program order

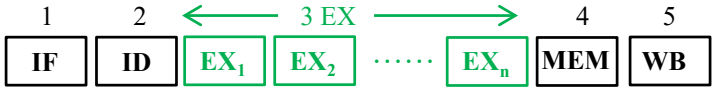
Floating Point/Multicycle Pipelining in MIPS

- Completion of MIPS EX stage floating point arithmetic operations in one or two cycles is impractical since it requires:
  - A much longer CPU clock cycle, and/or
  - An enormous amount of logic.



Solution:

- Instead, the floating-point pipeline will allow for a longer latency (more EX cycles than 1).



- Floating-point operations have the same pipeline stages as the integer instructions with the following differences:
  - The EX cycle may be repeated as many times as needed (more than 1 cycle).
  - There may be multiple floating-point functional units.
  - A stall will occur if the instruction to be issued either causes a structural hazard for the functional unit or cause a data hazard.
- *The latency* of functional units is defined as the number of intervening or stall cycles between an instruction producing the result and the instruction that uses the result (usually equals stall cycles with forwarding used). ←
- *The initiation* or repeat interval is the number of cycles that must elapse between issuing an instruction of a given type.

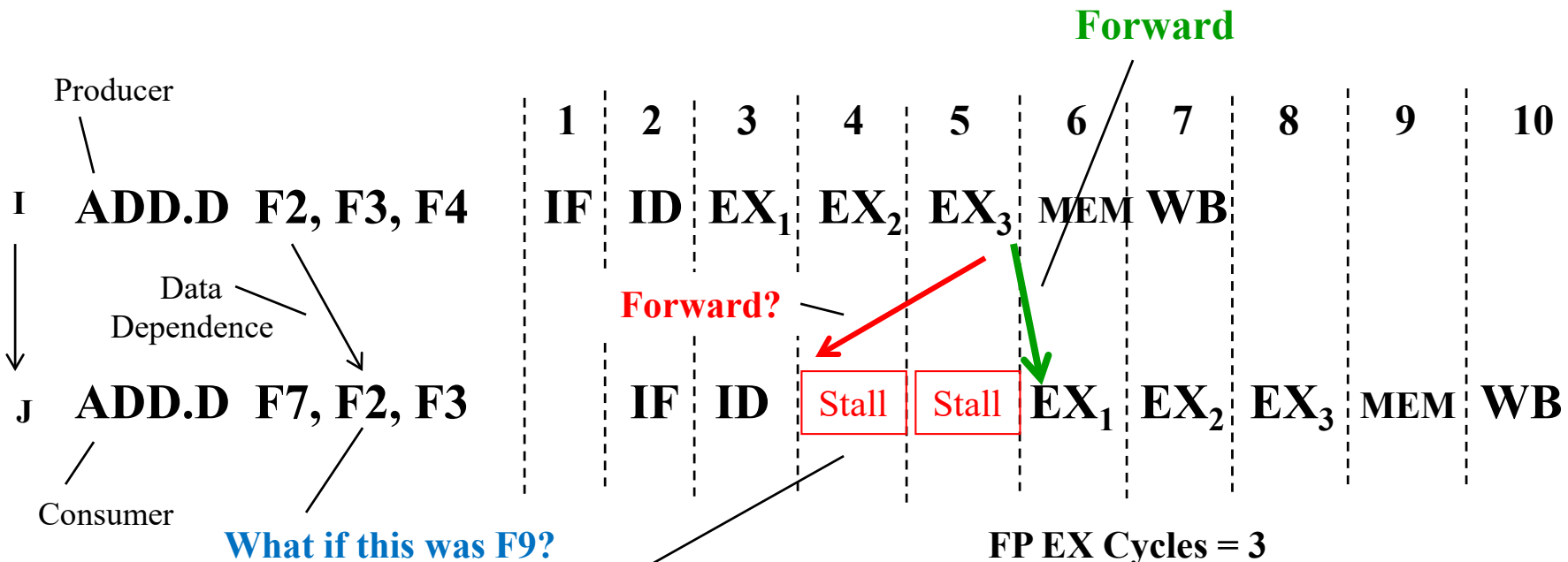
(In Appendix A)

to the same functional unit ←

Multicycle Functional Unit Latency

The latency **of functional units is defined as the number of intervening or stall cycles between an instruction producing the result and the instruction that uses the result (usually equals stall cycles with forwarding used).**

Assuming Floating Point (FP) Execution (EX) Takes = 3 Cycles

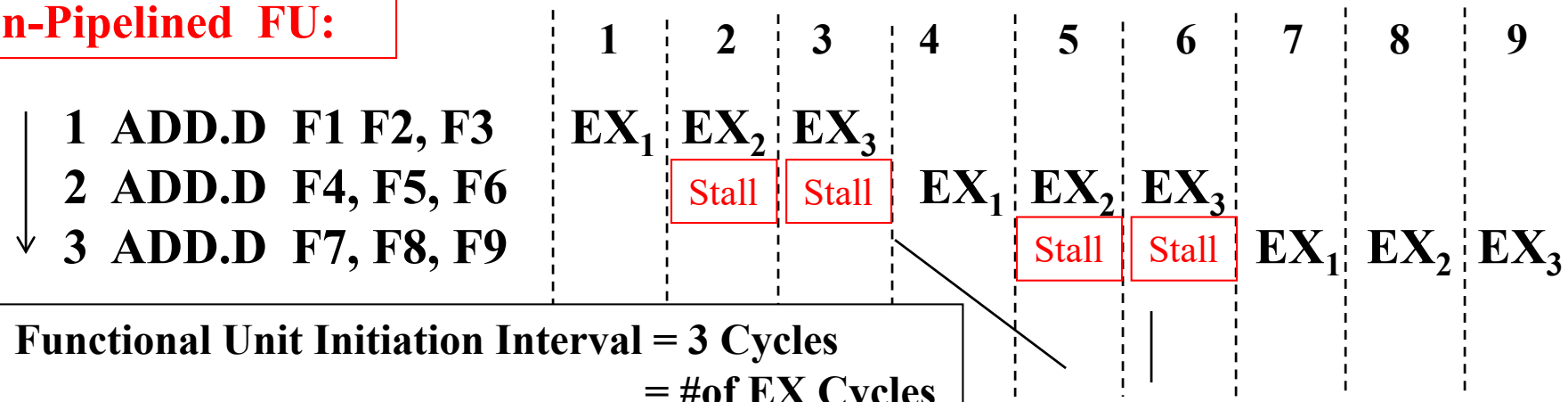


Functional Unit Initiation Interval:

Non-Pipelined Vs. Pipelined Floating Point (FP) Functional Units (FUs)

→ Assume that processor has only one Floating Point Addition Unit
Floating Point (FP) Execution (EX) Takes = 3 Cycles

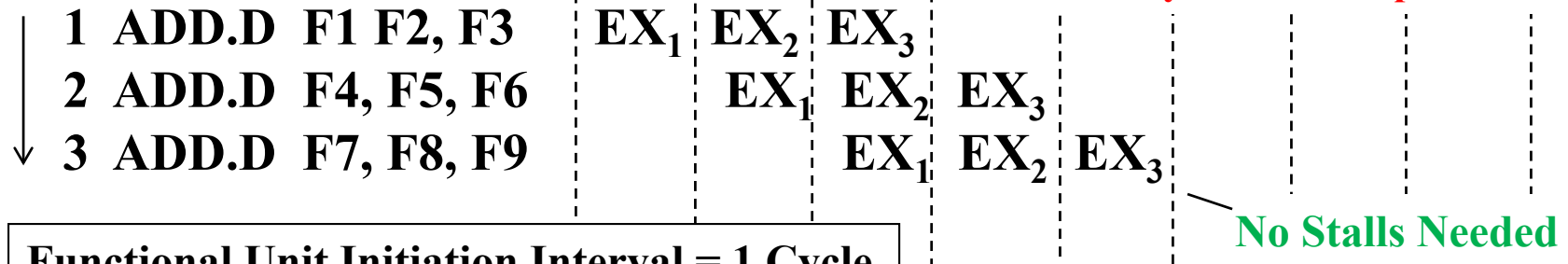
Non-Pipelined FU:



Functional Unit Initiation Interval = 3 Cycles
= #of EX Cycles

Two Stalls Needed
(To Prevent Structural Hazard)
- Only One Non-Pipelined FU -

Pipelined FU:

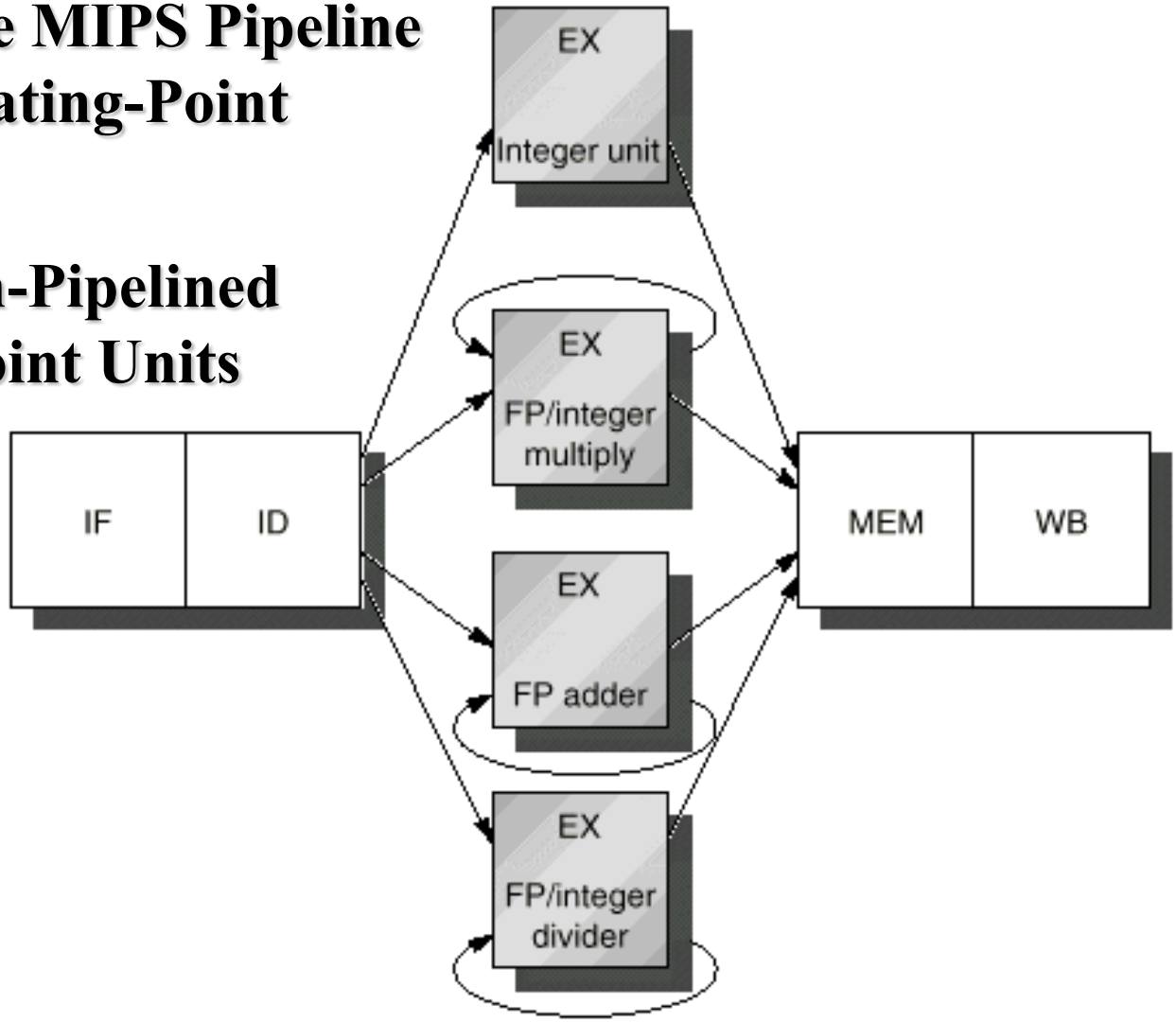


Functional Unit Initiation Interval = 1 Cycle

No Stalls Needed

Extending The MIPS Pipeline to Handle Floating-Point Operations:

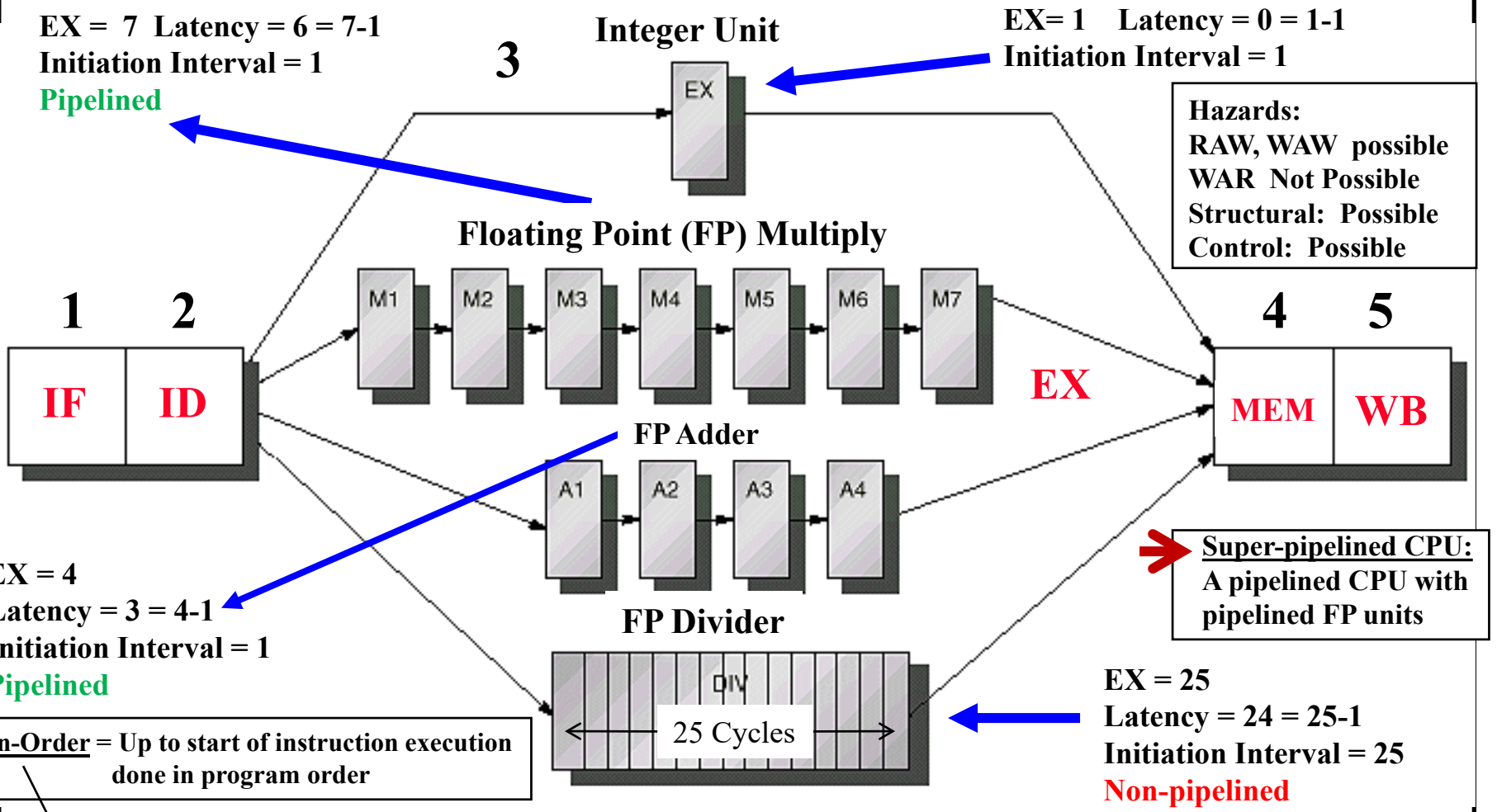
Adding Non-Pipelined Floating Point Units



The MIPS pipeline with three additional unpipelined, floating-point functional units (FP FUs)

(In Appendix A)

Extending The MIPS Pipeline: Multiple Outstanding Floating Point Operations



A pipeline that supports multiple outstanding FP operations.

In-Order Single-Issue MIPS Pipeline with FP Support

(In Appendix A) Pipelined CPU with pipelined FP units = Super-pipelined CPU

Latencies and Initiation Intervals For Functional Units (FUs)

Shown in last slide

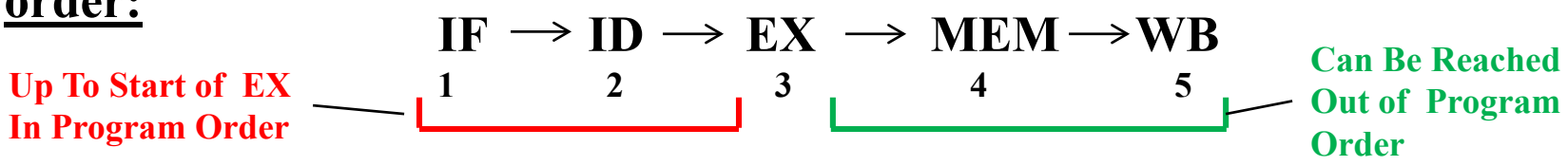
EX Cycles	Functional Unit	Latency		Initiation Interval
1	Integer ALU	0	= 1 - 1	1
2 ?	Data Memory (Integer and FP Loads)	1	= 2 – 1 ?	1
4	FP add	3	= 4 - 1	1
7	FP multiply	6	= 7 - 1	1 Pipelined
25	FP divide	24	= 25 - 1	25 Not Pipelined

Latency usually equals stall cycles when full forwarding is used

(In Appendix A)

Pipeline Characteristics With FP Support

- Instructions are still processed **in-order** in: **IF, ID and up to start of EX** at the rate of one instruction per cycle. **Thus pipeline is still called in-order:**



- **Longer RAW hazard stalls** likely due to long FP latencies. More Stalls ?
- **Structural hazards possible** due to varying instruction times and FP latencies:
  - FP unit may not be available (not pipelined?) ; divide in this case.
  - MEM, WB reached by several instructions simultaneously (New).
- **WAW hazards can occur (New)** since it is now possible for instructions to reach WB out-of-order.
- **WAR hazards impossible**, since register reads occur in-order in ID. i.e. Before a following instruction overwrites value in WB

Provided no WAW hazard results

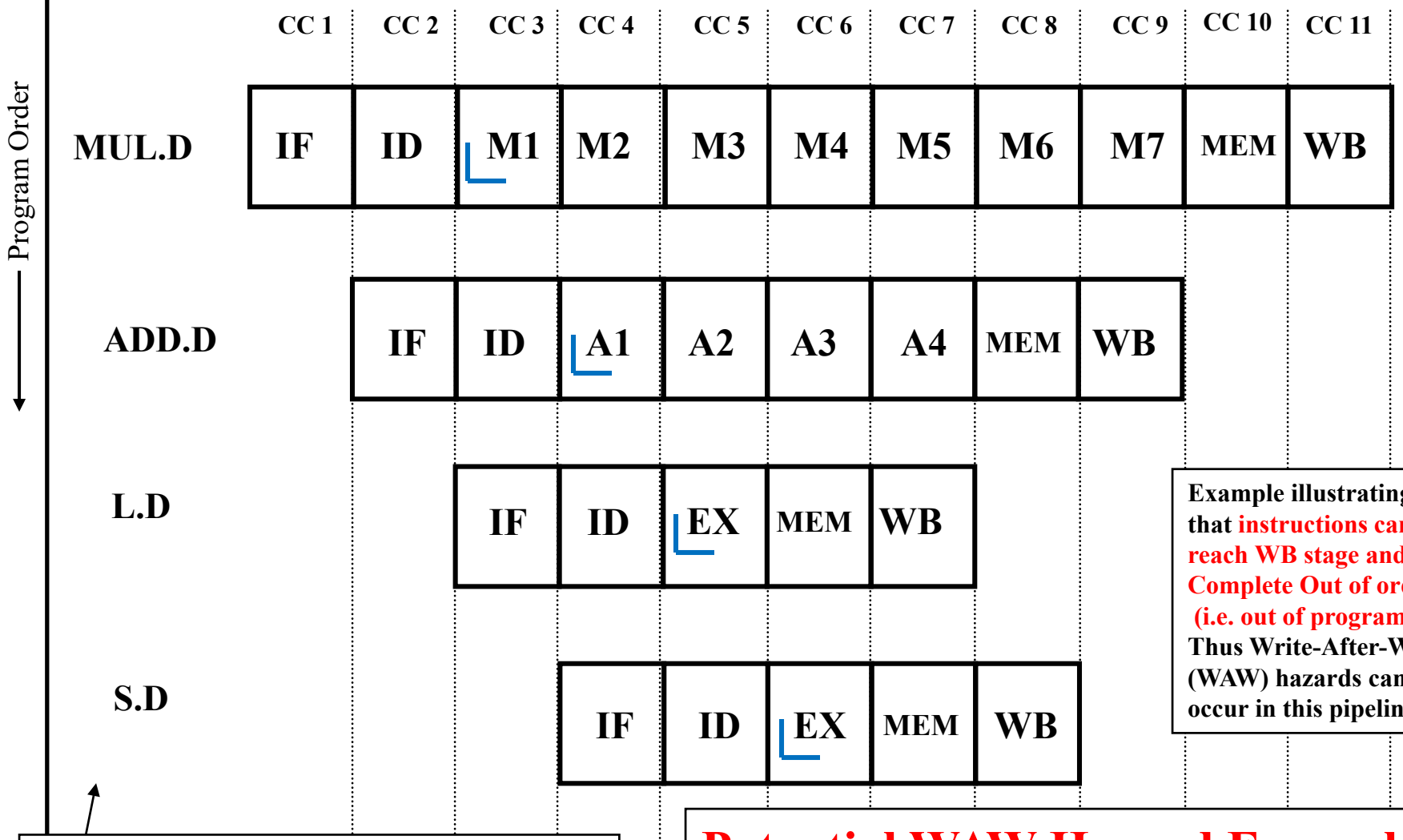
- Instructions can be allowed to **complete out-of-order** requiring **special measures to enforce precise exceptions**.

(In Appendix A)

Order = Program order

FP Operations Pipeline Timing Example

FP Multiply = 7 EX cycles FP ADD = 4 EX Cycles Integer = 1 EX Cycle



Example illustrating that instructions can reach WB stage and Complete Out of order (i.e. out of program order). Thus Write-After-Write (WAW) hazards can occur in this pipeline

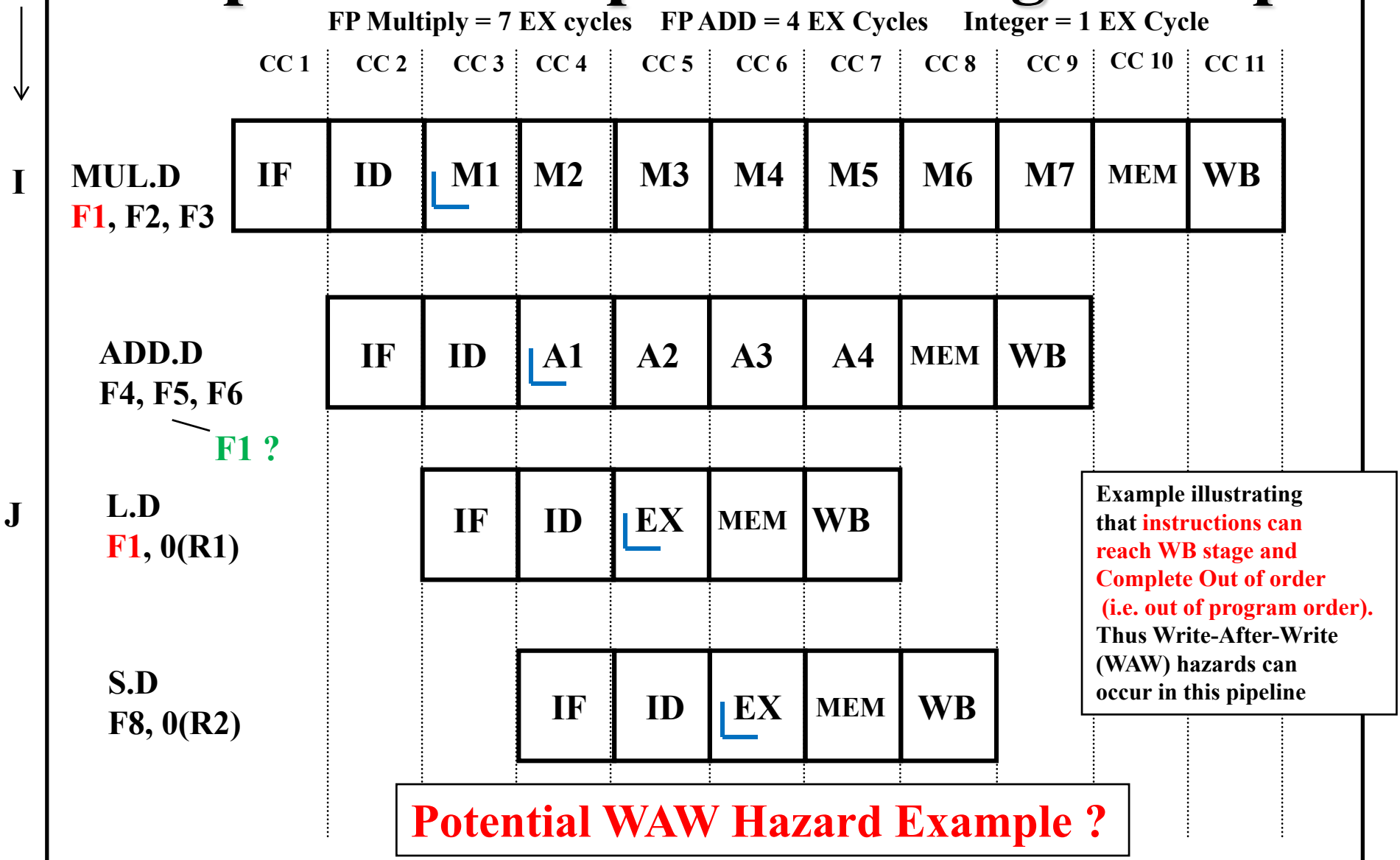
All above instructions are assumed independent

Potential WAW Hazard Example ?

When run on In-Order Single-Issue MIPS Pipeline with FP Support With FU latencies/initiation intervals given in slides 61-62

(In Appendix A)

FP Operations Pipeline Timing Example



When run on In-Order Single-Issue MIPS Pipeline with FP Support With FU latencies/initiation intervals given in slides 61-62

(In Appendix A)

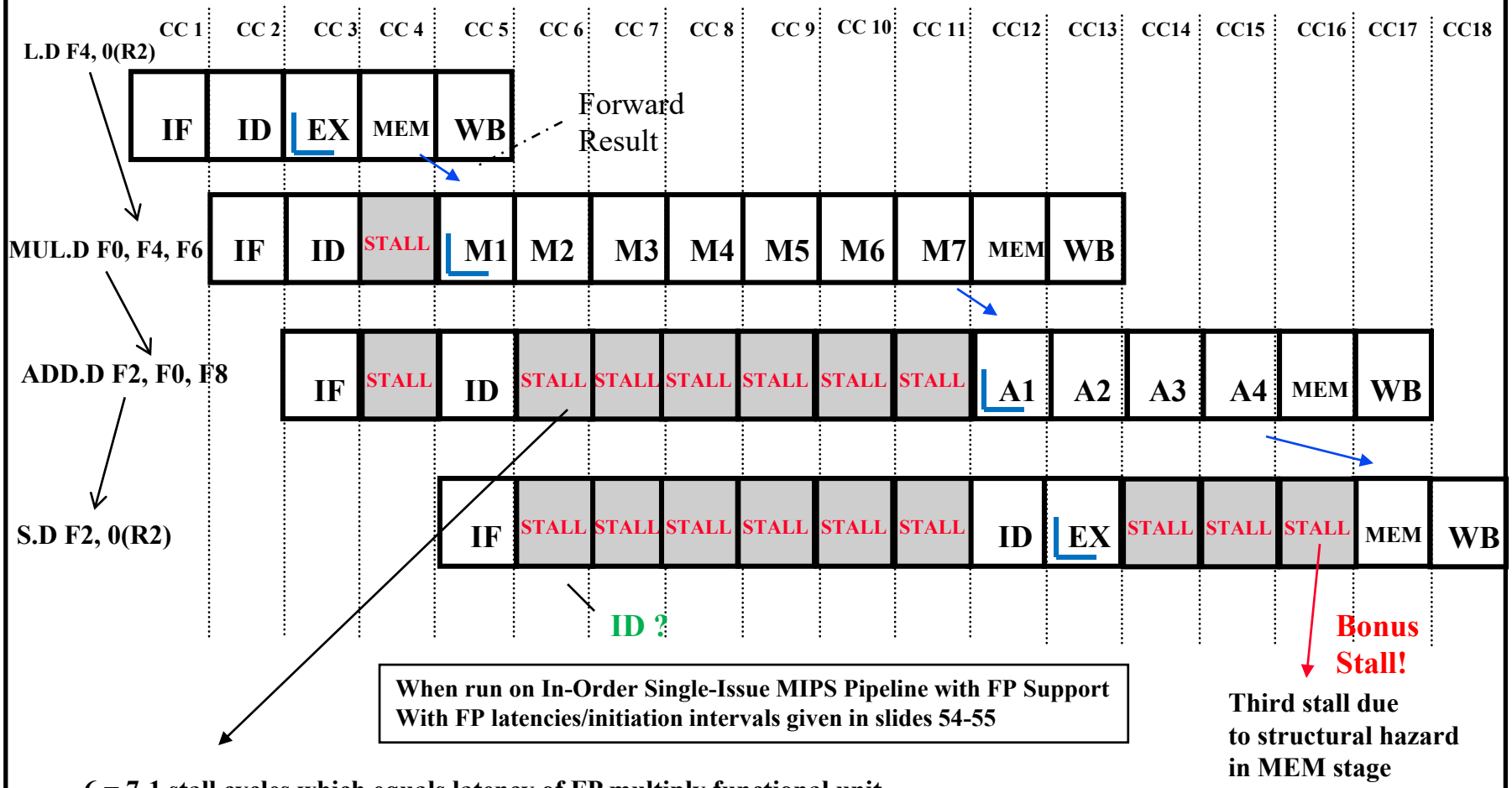
FP Code RAW Hazard Stalls Example

(with full data forwarding in place)

As indicated in slides 61-62: FP Multiply Functional Unit has 7 EX cycles (and 6 cycle latency 6 = 7-1)
FP Add Functional Unit has 4 EX cycles (and 3 cycle latency 3 = 4-1)

Integer = 1 EX Cycle

Program Order

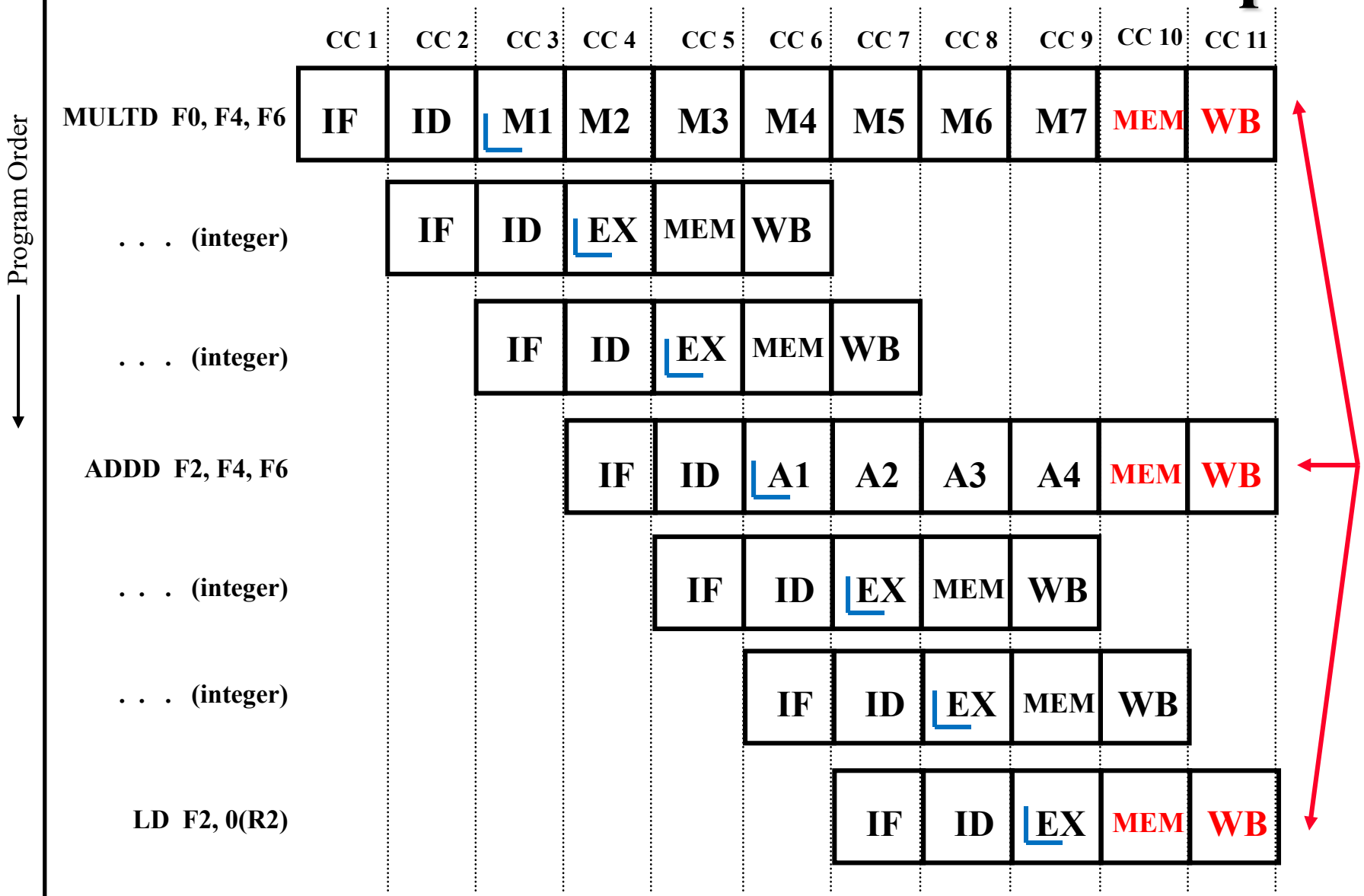


When run on In-Order Single-Issue MIPS Pipeline with FP Support
With FP latencies/initiation intervals given in slides 54-55

6 = 7-1 stall cycles which equals latency of FP multiply functional unit

(In Appendix A)

FP Code Structural Hazards Example



When run on In-Order Single-Issue MIPS Pipeline with FP Support
With FP latencies/initiation intervals given in slides 54-55

Maintaining Precise Exceptions in Multicycle Pipelining

i.e. with FP support

- In the MIPS code segment:

```
DIV.D   F0, F2, F4
ADD.D   F10, F10, F8
SUB.D   F12, F12, F14
```

Exception Generated

Already Done

- The ADD.D, SUB.D instructions can complete before DIV.D is completed causing out-of-order execution completion.
- If DIV.D causes a floating-point arithmetic exception, precise exception handling is harder since both ADD.D, SUB.D have already completed.
- Four approaches have been proposed to remedy this type of situation:

1. Ignore the problem and settle for imprecise exception. Faster?
2. Buffer the results of the operation until all the operations issues earlier are done. (large buffers, multiplexers, comparators) i.e. Stall WB
3. A history file keeps track of the original values of registers (CYBER180/190, VAX) Used to restore original register values if needed
4. A Future file keeps the newer value of a register; when all earlier instructions have completed the main register file is updated from the future file (Stall WB?). On an exception the main register file has the precise values for the interrupted state.

i.e Force in-order completion

Approaches 2 and 4 above are similar : Force in-order completion (in-order WB)

(In Appendix A)

Quiz # 2

On Lecture Notes Set # 2

- **Wednesday, February 8**