

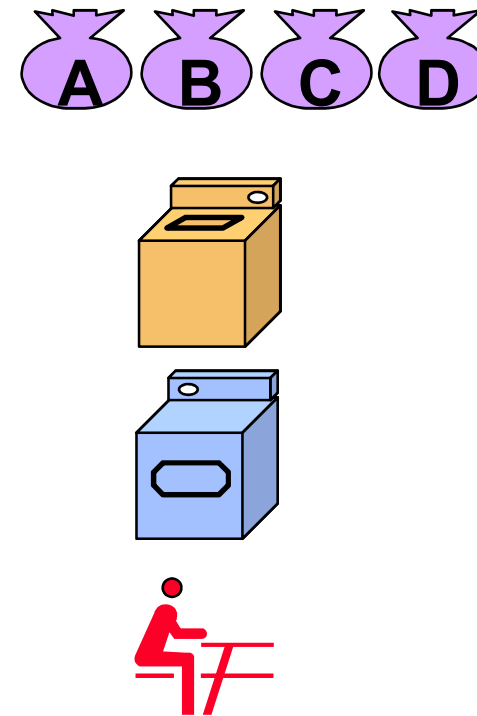
Computer Architecture (2024)

Pipelining

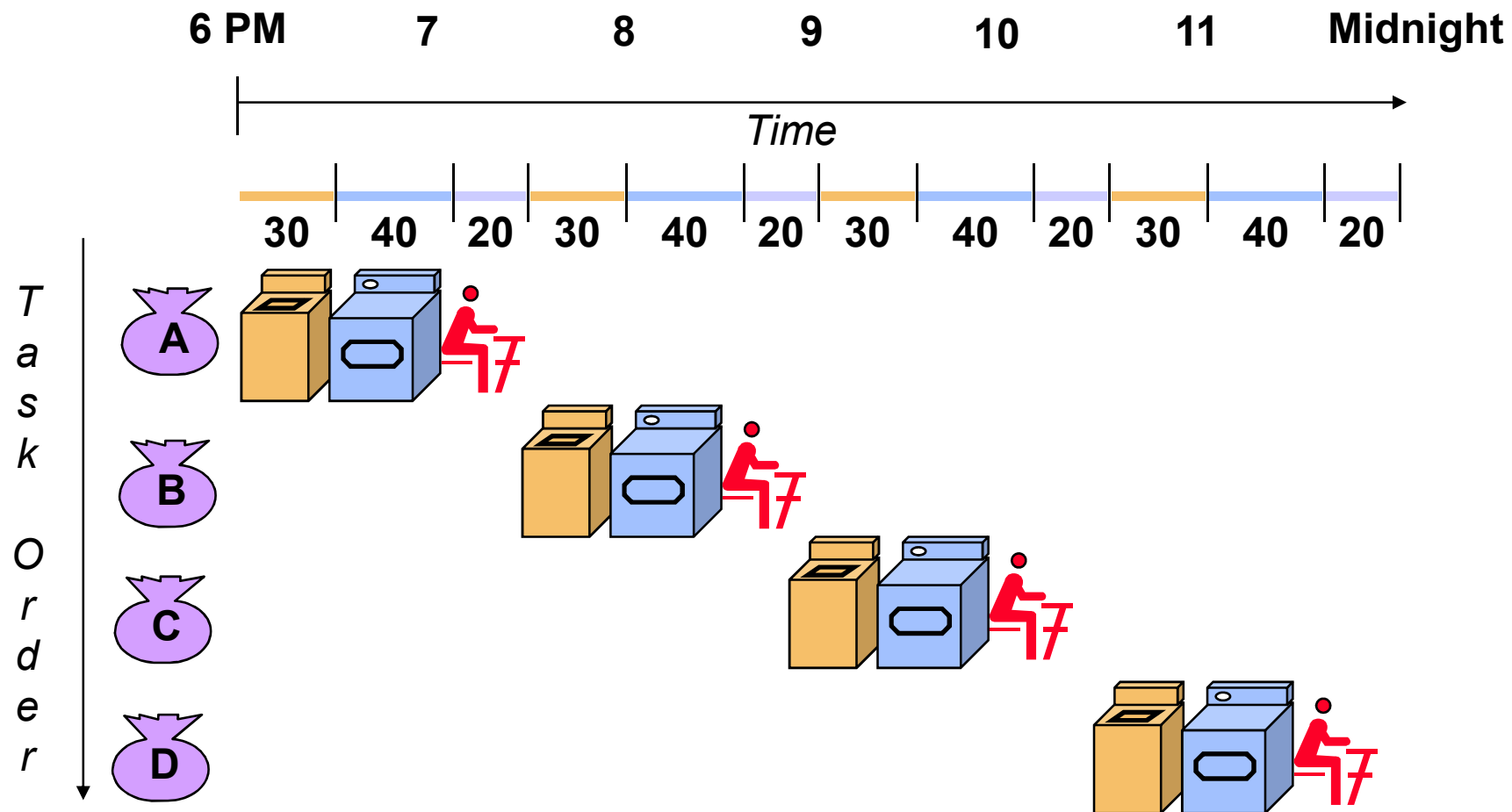
College of Computer Science
Chongqing University

Traditional Pipeline Concept

- Laundry Example
- Ann, Brian, Cathy, Dave each has one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes

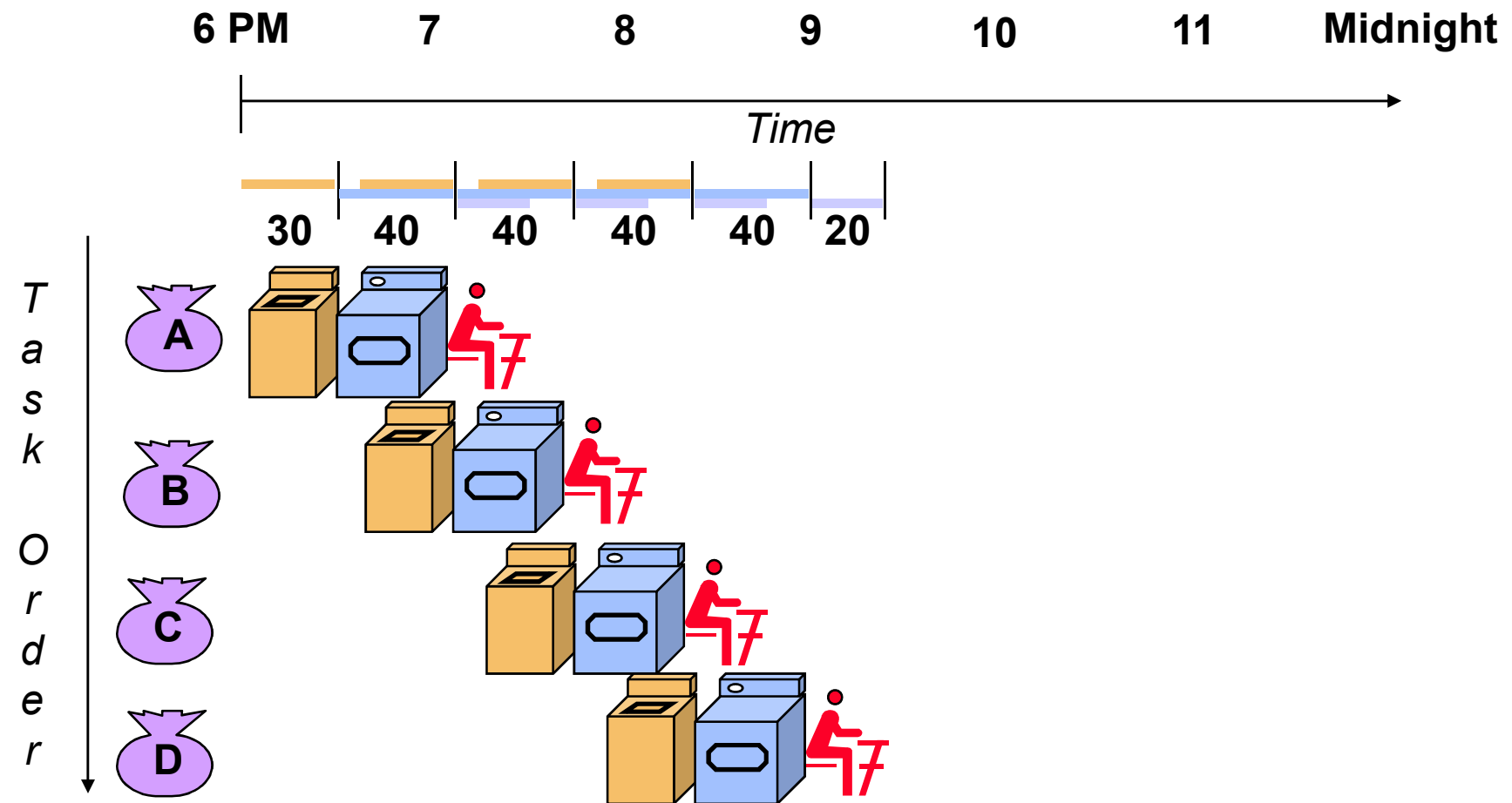


Traditional Pipeline Concept



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Traditional Pipeline Concept



- Pipelined laundry takes 3.5 hours for 4 loads

Overview

- Pipelining improves performance by overlapping multiple instructions
 - Takes advantage of parallelism among the actions needed to execute an instruction.
 - The number of instructions performed per second is increased
 - The time needed for one instruction is not changed, and may be increased.
- The **throughput** of an instruction pipeline is the measure of how often an instruction exits the pipeline.
- Invisible to high level (OS, programs)

Pipeline Stages

We can divide the execution of an instruction into the following 5 “classic” stages:

IF: Instruction Fetch

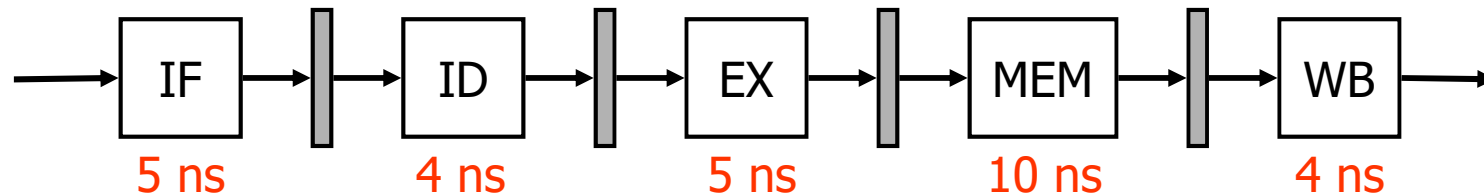
ID: Instruction Decode (w/ register fetch)

EX: Execution

MEM: Memory Access

WB: (Register) Write Back

Pipeline Throughput and Latency

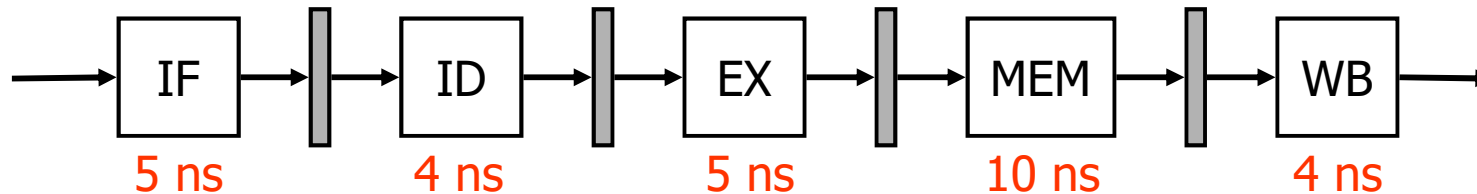


Consider the pipeline above with the indicated delays. We want to know what is the *pipeline throughput* and the *pipeline latency*.

Pipeline throughput: instructions completed per second.

Pipeline latency: how long does it take to execute a single instruction in the pipeline.

Pipeline Throughput and Latency



Pipeline throughput: how often an instruction is completed.

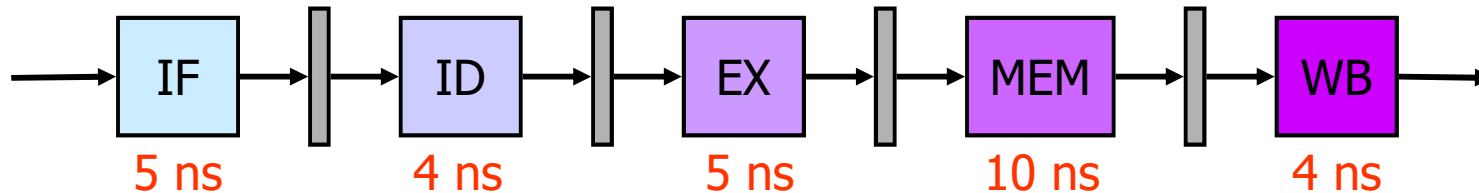
$$\begin{aligned} &= 1instr / \max[lat(IF), lat(ID), lat(EX), lat(MEM), lat(WB)] \\ &= 1instr / \max[5ns, 4ns, 5ns, 10ns, 4ns] \\ &= 1instr / 10ns \quad (\text{ignoring pipeline register overhead}) \end{aligned}$$

Pipeline latency: how long does it take to execute an instruction in the pipeline.

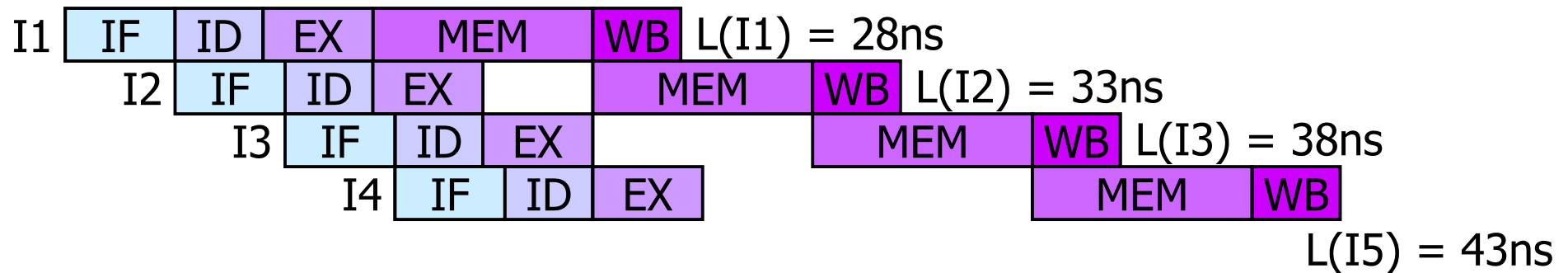
$$\begin{aligned} L &= lat(IF) + lat(ID) + lat(EX) + lat(MEM) + lat(WB) \\ &= 5ns + 4ns + 5ns + 10ns + 4ns = 28ns \end{aligned}$$

Is this right?

Pipeline Throughput and Latency

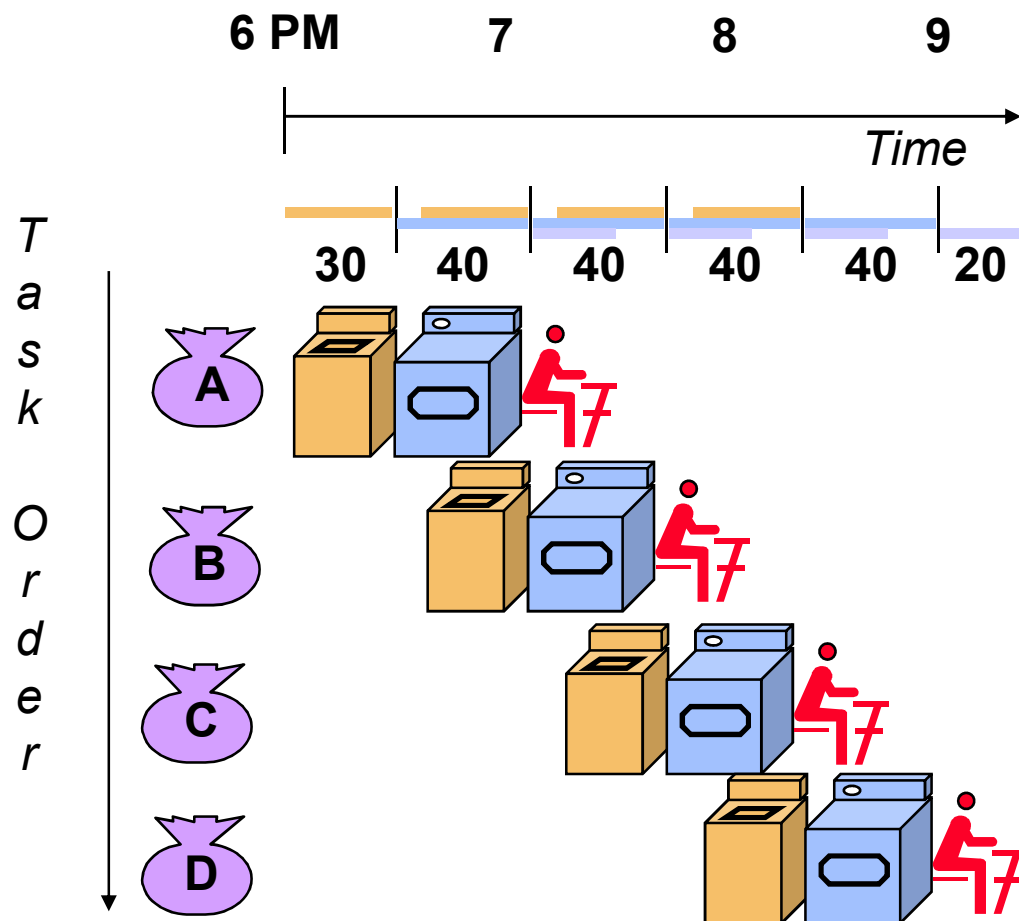


Simply adding the latencies to compute the pipeline latency, only would work for an isolated instruction



We are in **trouble**! The latency is not constant. This happens because this is an unbalanced pipeline. The solution is to make every state the same length as the longest one.

Pipelining Lessons



- Pipelining **doesn't help latency** of a single task, it helps **throughput** of the entire workload
- **Pipeline rate limited** by the **slowest** pipeline stage
- **Multiple** tasks operating simultaneously
- **Potential speedup** = **number of pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to “**fill**” pipeline and time to “**drain**” it reduces speedup

Other Definitions

- Pipe stage (or pipe segment)
 - A decomposable unit of the fetch-decode-execute paradigm
- Pipeline depth
 - Number of stages in a pipeline
- Machine cycle
 - Clock cycle time
- Latch
 - Per phase/stage local information storage unit

Design Issues

- Balance the length of each pipeline stage

$$\text{Throughput} = \frac{\text{Depth of the pipeline}}{\text{Time per instruction on unpipelined machine}}$$

- Problems
 - Usually, stages are not balanced
 - Pipelining overhead
 - Hazards (conflicts)
- Performance (throughput → CPU performance equation)
 - Decrease of the CPI
 - Decrease of the cycle time

We use RISC architecture to illustrate

- All operations on data apply to data in registers, and typically change the entire register (32 or 64 bits per register).
- The only operations that affect memory are load and store.
 - Load and store operations that load or store less than a full register (e.g., a byte, 16 bits, or 32 bits) are often available.
- The instruction formats are few in number, with all instructions typically being one size.

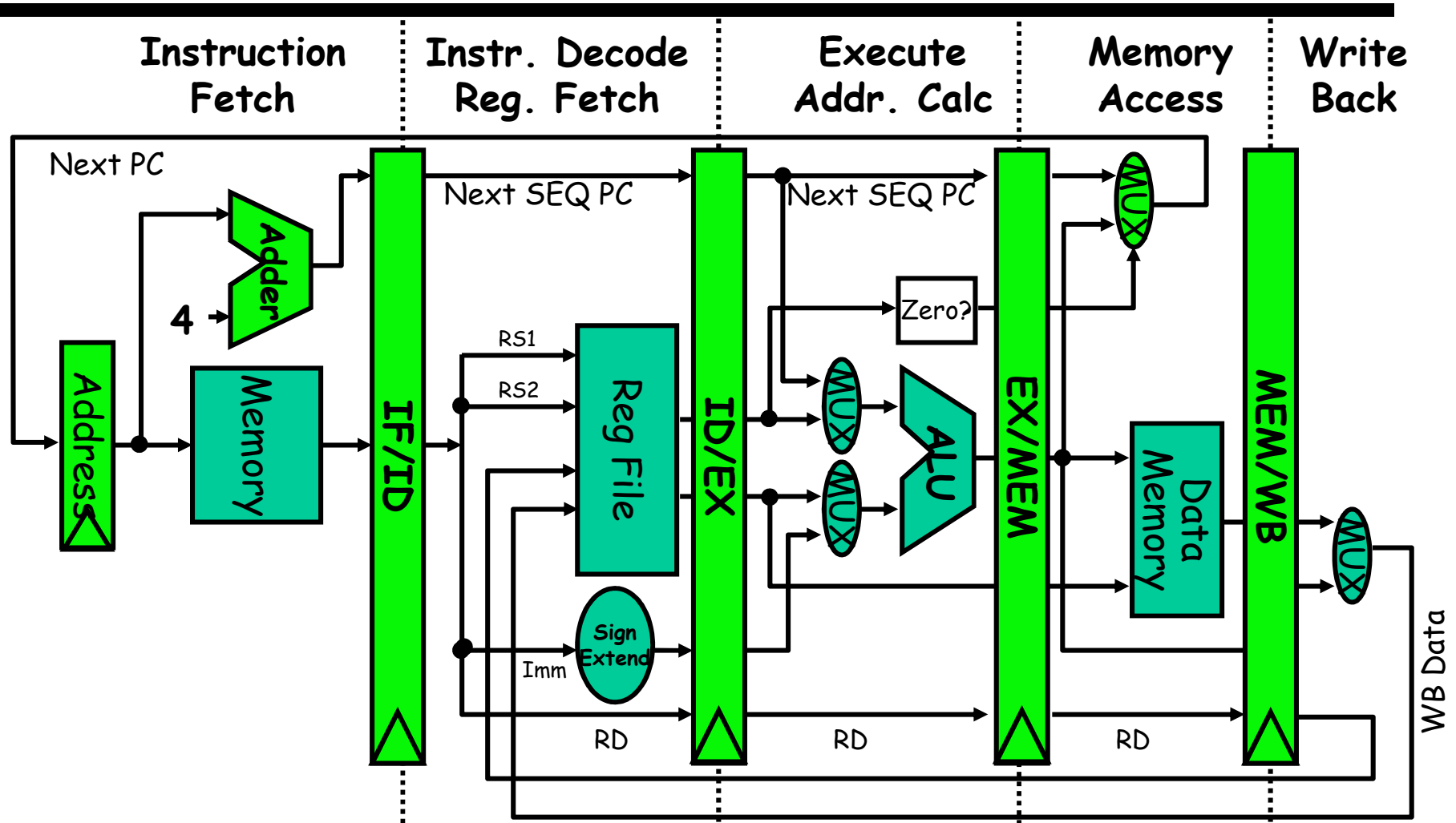
MIPS Instructions, one of RISC ISAs

- ALU instructions
 - Take either two registers or a register and a sign-extended immediate, operate on them, and store the result into a third register.
 - add (DADD), subtract (DSUB), and logical operations (such as AND or OR),
 - Immediate versions of these instructions use the same mnemonics with a suffix of I.
- There are both signed and unsigned forms of the arithmetic instructions;
 - the unsigned forms, which do not generate overflow exceptions— and thus are the same in 32-bit and 64-bit mode—have a U at the end (e.g., DADDU, DSUBU, DADDIU)..

MIPS Instructions, one of RISC ISAs

- Load and store instructions
 - base register + an immediate offset = effective address.
 - In the case of a load instruction, a 2nd register acts as the destination.
 - In the case of a store, the 2nd register operand is the source of the data that is stored into memory.
- Branch and jump instructions:
 - MIPS uses a set of comparisons between a pair of registers or between a register and zero.
 - The branch destination is obtained by adding a sign-extended offset (16 bits in MIPS) to the current PC.

Pipelined MIPS Datapath Schematic



- Data stationary control
 - local decode for each instruction phase / pipeline stage

The 1st stage of the 5-stage MIPS pipeline: Instruction Fetch (IF)

- Fetch the current instruction from memory according to the program counter (PC).
- Update the PC to the address of the next instruction in sequence by adding 4 (since each instruction is 4 bytes) to the PC.

$IR \leftarrow \text{Mem}[PC];$

$NPC \leftarrow PC + 4$

The 2nd stage of the 5-stage MIPS pipeline: Instruction Decoding (ID) w/ Register Fetch

- Decode the instruction and read the registers;
- Do the equality test on the registers as they are read, for a **possible** branch;
- Sign-extend the offset field of the instruction **in case** it is needed;
- Compute the **possible** branch target address by adding the sign-extended offset to the incremented PC;
 $A \leftarrow \text{Regs}[\text{IR}_{6..10}]$
 $B \leftarrow \text{Regs}[\text{IR}_{11..15}]$
 $\text{Imm} \leftarrow ((\text{IR}_{16})^{16} \# \# \text{IR}_{16..31})$
- In an aggressive implementation, branch instruction can be completed \rightarrow by storing the target address into the PC, if the condition test yielded true;

3rd Instruction cycle: Execution (EX)

- Execution or effective address calculation
 - In a load-store architecture, no instruction needs to simultaneously calculate a data address and perform an operation on the data
 - ALU will work on one of the two possibilities
- Do one of the following operations:
 - Memory reference: Base register + Immediate Offset
 - $\text{ALUOutput} \leftarrow A + \text{Imm}$
 - Register - Register ALU instruction: specified by op code
 - $\text{ALUOutput} \leftarrow A \text{ func } B$
 - Register - Immediate ALU instruction: specified by op code
 - $\text{ALUOutput} \leftarrow A \text{ func Imm}$

4th Instruction cycle: Memory Access (MEM)

- Memory access
 - Memory reference
 - $PC \leftarrow NPC$
 - $LMD \leftarrow Mem[ALUOutput]$ (load)
 - $Mem[ALUOutput] \leftarrow B$ (store)

5th Instruction cycle: Write Back (WB)

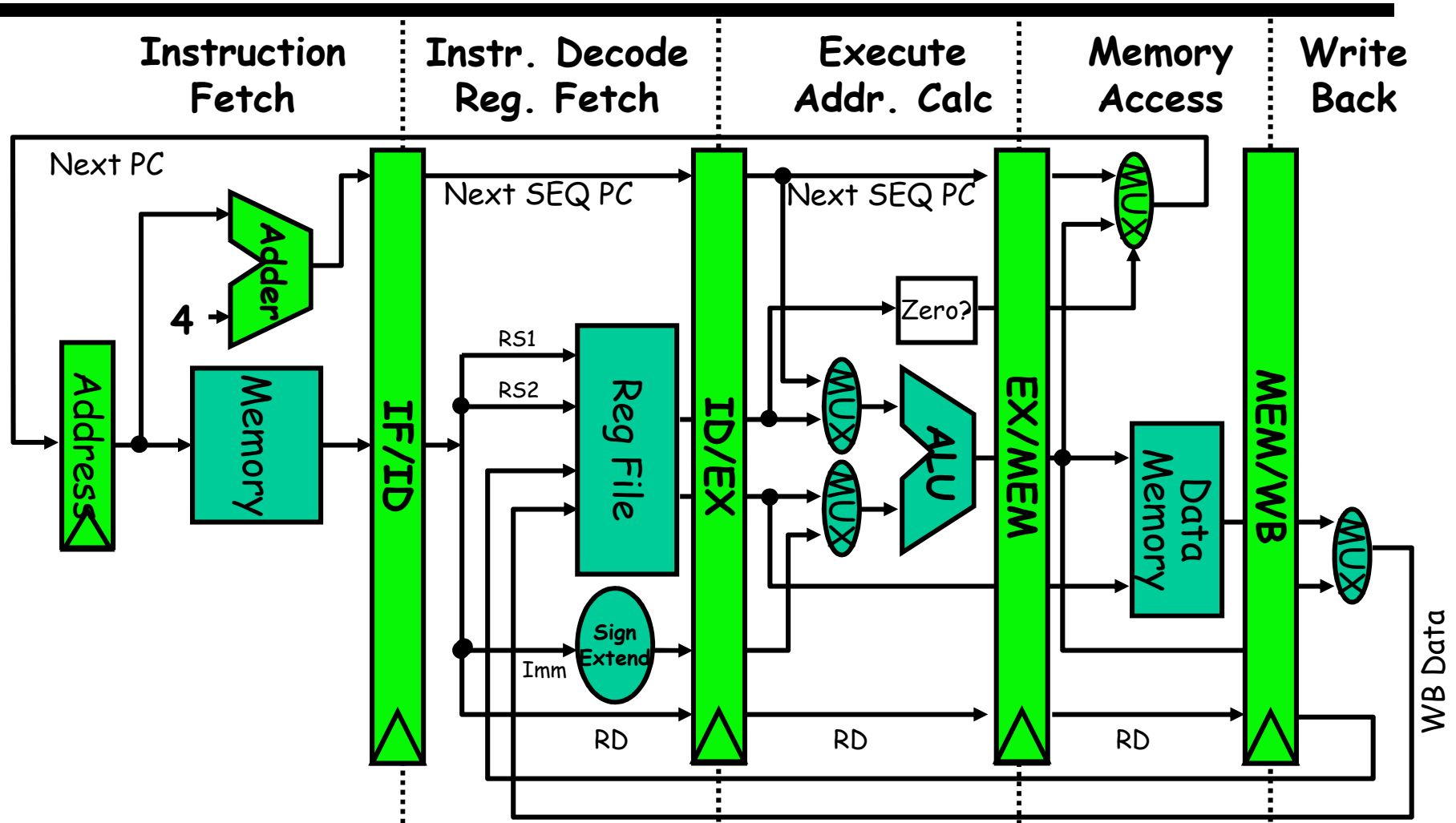
- Write-back (WB)
 - Register - register ALU instruction
 - $\text{Regs}[\text{IR}_{16..20}] \leftarrow \text{ALUOutput}$
 - Register - immediate ALU instruction
 - $\text{Regs}[\text{IR}_{11..15}] \leftarrow \text{ALUOutput}$
 - Load instruction
 - $\text{Regs}[\text{IR}_{11..15}] \leftarrow \text{LMD}$

Summary: The classic 5-stage pipeline

- Branch: 2 cycles, finish at ID;
- Store 4 cycles, finish at MEM;
- All other instructions: 5 cycles, finish at WB.
- Assuming 12% branch instructions, 10% store instructions, leads to an overall CPI of 4.54.

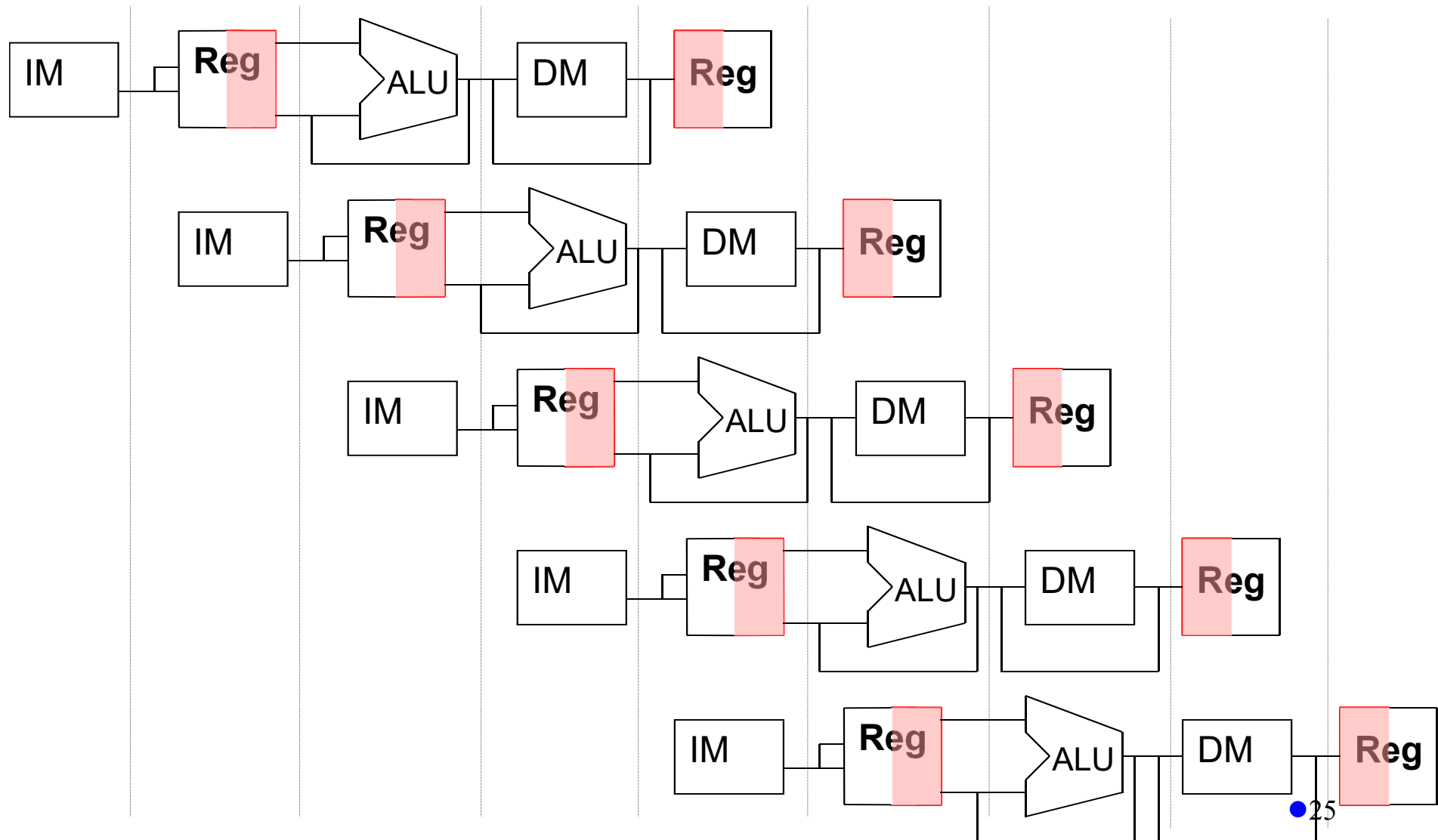
Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Pipelined MIPS Datapath Schematic



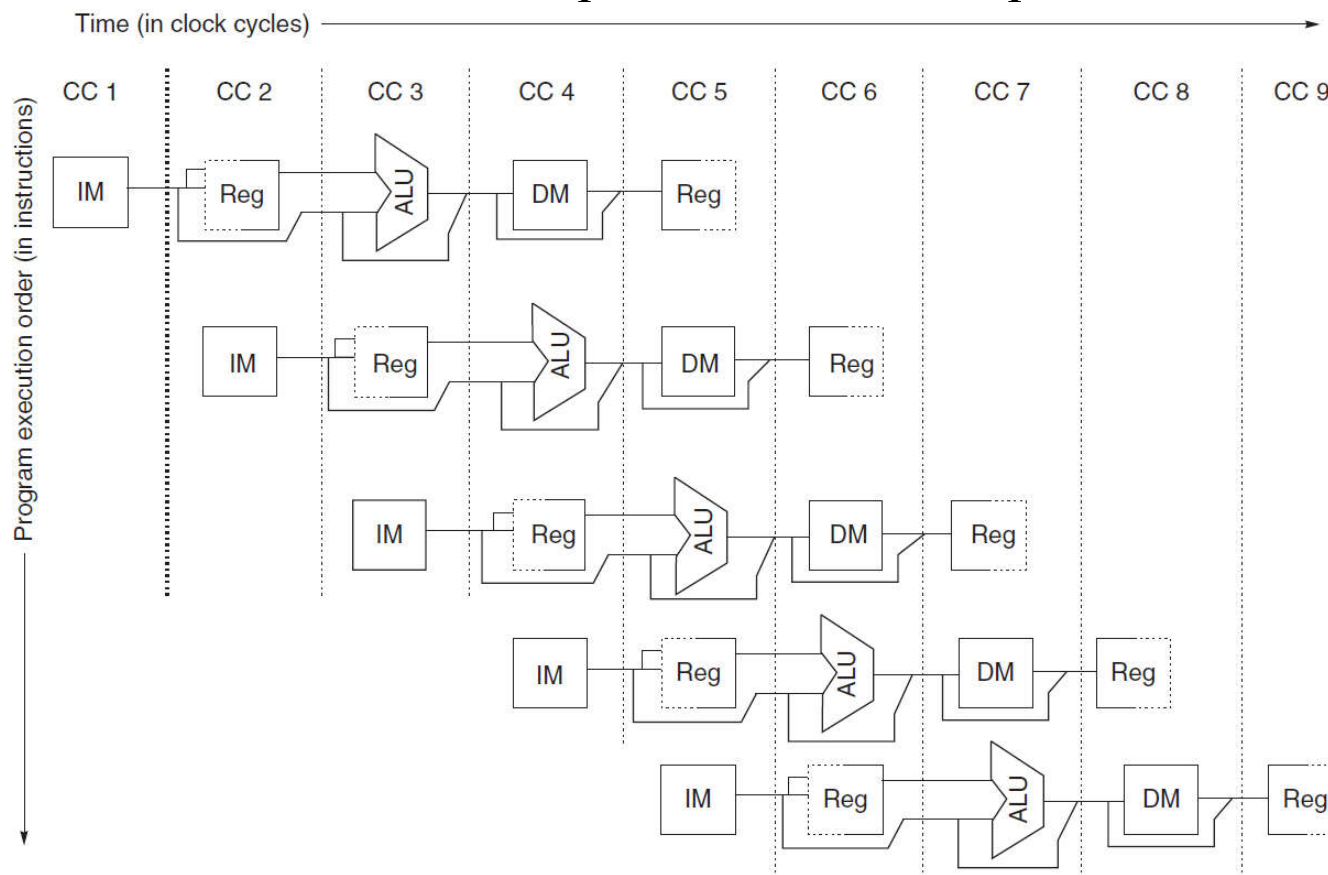
- Data stationary control
 - local decode for each instruction phase / pipeline stage

Pipeline Resources



The challenge on designing a pipelined datapath

- Need to make sure that a hardware unit is not asked to perform two operations in the same cycle.
 - For example, an ALU cannot be asked to compute an effective address and perform a subtract operation at the same time.



IM: instruction memory

DM: data memory

CC: clock cycles

Several Observations

- Use separate instruction and data memories, implemented by separate instruction and data caches;
 - It eliminates a conflict between instruction fetch and data memory access.
- Reg is used in both ID (for reading) and WB (for writing). Hence, two reads (distinct) and one write every clock cycle;
 - Notice that Write is always from earlier instruction and Reads are from later instructions.
 - Perform Write on the 1st half of the cc (rising edge of clock) and Read in the 2nd half (falling edge of clock).
- Operations on PC are not shown.
 - PC must be incremented every clock, and must be done at IF in preparation for the next instruction.
 - A branch does not change the PC until the end of ID stage.
 - This causes a problem, will be discussed shortly.
 - Needs an independent adder only for PC addition/subtraction.

Performance Issues in Pipelining

- Pipelining increases performance by running multiple instructions simultaneously. But it does not reduce the latency of every single instruction.
 - Due to pipeline overhead, the latency of an instruction is in fact increased slightly.
- Pipeline overhead 1: Imbalanced pipeline stages:
 - Slowest stage dominates the total throughput.
- Pipeline overhead 2: Pipeline registers and clock skew
 - Registers introduce setup time into clock frequency.
 - Clock skew: We assume all registers see the clock edges at the exact same time. In real work, due to physical design imperfection, some registers see edges earlier than others – clock skew.
- Hazards: we will discuss it in the next lecture.

MIPS Pipelining: Basic Performance Issues

- Pipelining

- improve CPU Throughput (reciprocal of CPU Time)
- slightly increases execution time
- result: program run faster!

$$[\text{time per instruction}]_{\text{pipelined}} = \frac{[\text{time per instruction}]_{\text{nonpipelined}}}{\text{number of pipeline stages}}$$

- Pipelining can be seen as either

- decreasing the CPI of a multi-cycle un-pipelined implementation
- decreasing the CCT of a single-cycle un-pipelined implementation

$$\text{CPU Time} = \text{IC} \times \text{CPI} \times \text{CCT}$$

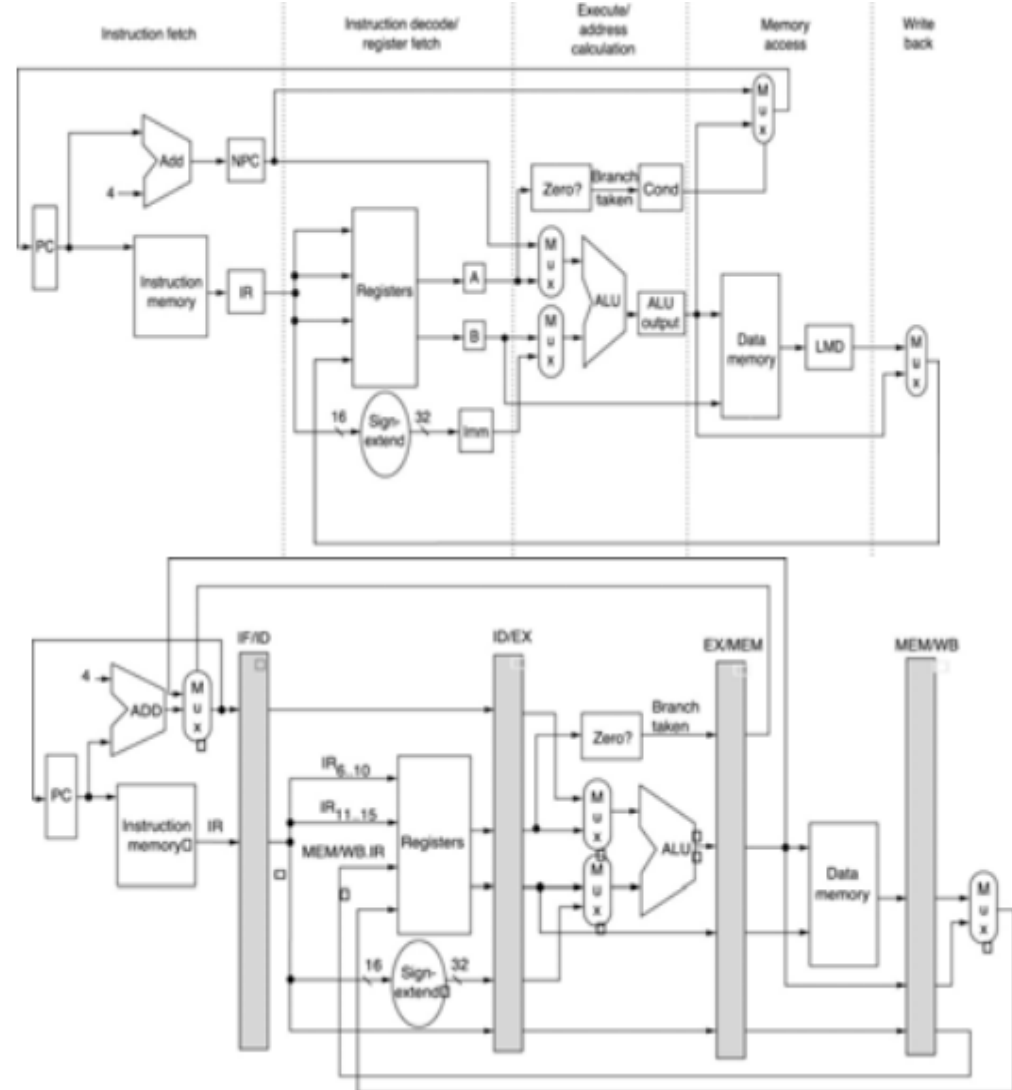
	IC	CPI	CCT
Program	✓		
Compiler	✓		
ISA	✓	✓	
HW organization		✓	✓
HW technology			✓

A question

- Consider the unpipelined processor. Assume that it has a 1ns clock cycle and that it uses 4 cycles for branches and stores and 5 cycles for other operations. Assume that the relative frequencies of branch and store operations are 15% and 10%, respectively.
- Consider a pipelined processor. Assume the slowest stage takes 1ns and clock skew and register setup add 0.2 ns to the clock period.
- How much speedup in the instruction execution rate will we gain from an ideal pipeline?

Example: Pipelining Speedup vs. Multi-cycle Non-Pipelined Implementation

- **Instruction CPI**
 - 4 cycles (branches and stores)
 - 5 cycles (other instructions)
- **Average CPI = 4.75** assuming
 - 15% branches
 - 10% stores
- **Average Instruction ExecTime is 4.75ns**, assuming CCT = 1ns
- **Ideal CPI = 1** (almost always)
- **Average Instruction ExecTime is 1.2ns** with CCT = 1ns and clock overhead = 0.2 ns
- **Speedup is 3.96x** for ideal pipeline
 - as we'll see soon, in reality we need to sum the pipeline stalled clock cycles



MIPS Pipeline: Events per Stage

Stage	ALU Instruction	Load/Store Instruction	Branch Instruction
IF	$IF/ID.IR \leftarrow Mem[PC];$ $IF/ID.NPC \leftarrow \text{if } ((EX/MEM.opc == \text{branch}) \ \& \ EX/MEM.cond) \ EX/MEM.AluOutput \text{ else } PC+4;$		
ID	$ID/EX.A \leftarrow Regs[IF/ID.IR[rs]]; \quad ID/EX.B \leftarrow Regs[IF/ID.IR[rt]];$ $ID/EX.NPC \leftarrow IF/ID.NPC; \quad ID/EX.IR \leftarrow IF/ID.IR;$ $ID/EX.Imm \leftarrow \text{sign-extend}(IF/ID.IR[\text{immediate field}])$		
EX	$EX/MEM.IR \leftarrow ID/EX.IR;$ $EX/MEM.ALUOutput \leftarrow ID/EX.A \ op \ ID/EX.B \ \text{or}$ $EX/MEM.ALUOutput \leftarrow ID/EX.A \ op \ ID/EX.Imm$	$EX/MEM.IR \leftarrow ID/EX.IR;$ $EX/MEM.ALUOutput \leftarrow ID/EX.A \ op \ ID/EX.Imm$	$EX/MEM.ALUOutput \leftarrow ID/EX.NPC + (ID/EX.Imm \ll 2)$ $EX/MEM.Cond \leftarrow ID/EX.A == 0)$
MEM	$MEM/WB.IR \leftarrow EX/MEM.IR$ $MEM/WB.ALUOutput \leftarrow EX/MEM.AluOutput$	$MEM/WB.IR \leftarrow EX/MEM.IR$ $MEM/WB.LMD \leftarrow Mem[EX/MEM.AluOutput]$ <i>or</i> $Mem[EX/MEM.AluOutput] \leftarrow EX/MEM.B$	
WB	$Regs[MEM/WB.IR[rd]] \leftarrow MEM/WB.AluOutput \ \text{or}$ $Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.AluOutput$	<i>for load only:</i> $Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.LMD$	

PIPELINE HAZARDS

Pipeline Hazards and Their Classification

- Pipeline hazard situation
 - The next instruction cannot execute in the following clock cycle
- Three different flavors
 - Structural Hazards
 - arise from resource conflict: the HW cannot support all possible instruction combinations simultaneously in overlapped execution
 - Data Hazards
 - arise when an instruction depends on the result of a previous instruction in a way exposed by the pipeline overlapped execution
 - Control Hazards
 - arise from the pipelining of branches, jumps...
- Hazards may force pipeline stalling
 - Instructions issued after the stalled one must stall also (and fetching is stalled) while all those issued earlier must proceed

Performance of Pipelines with Stalls

Pipelining can be thought of as decreasing the CPI: This is for the case where both unpipelined version and pipelined version take the same amount of stages for instructions. Unpipelined version just does not fetch new instruction until the previous one finishes. The clock cycles of two versions are roughly same.

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\ &= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}} \\ &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}}\end{aligned}$$

Clock cycle unpipelined
= Clock cycle pipelined
If we ignore the pipeline overhead.

With Stalls

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} \\ &= 1 + \text{Pipeline stall clock cycles per instruction}\end{aligned}$$

Speedup, assuming all instructions take the same amount of stages

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

Performance of Pipelines with Stalls

Pipelining can be thought of as reducing the clock cycle: This is for the case where unpipelined version takes only one cycle for all instructions, while pipelined version take multiple stages. Hence the clock cycle of unpipelined version is more than that of the pipelined version.

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} \\ &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

In cases where the stages are perfectly balanced and there is no overhead, the clock cycle of pipelined version is smaller than that of the unpipelined version by a factor equal to the pipeline depth:

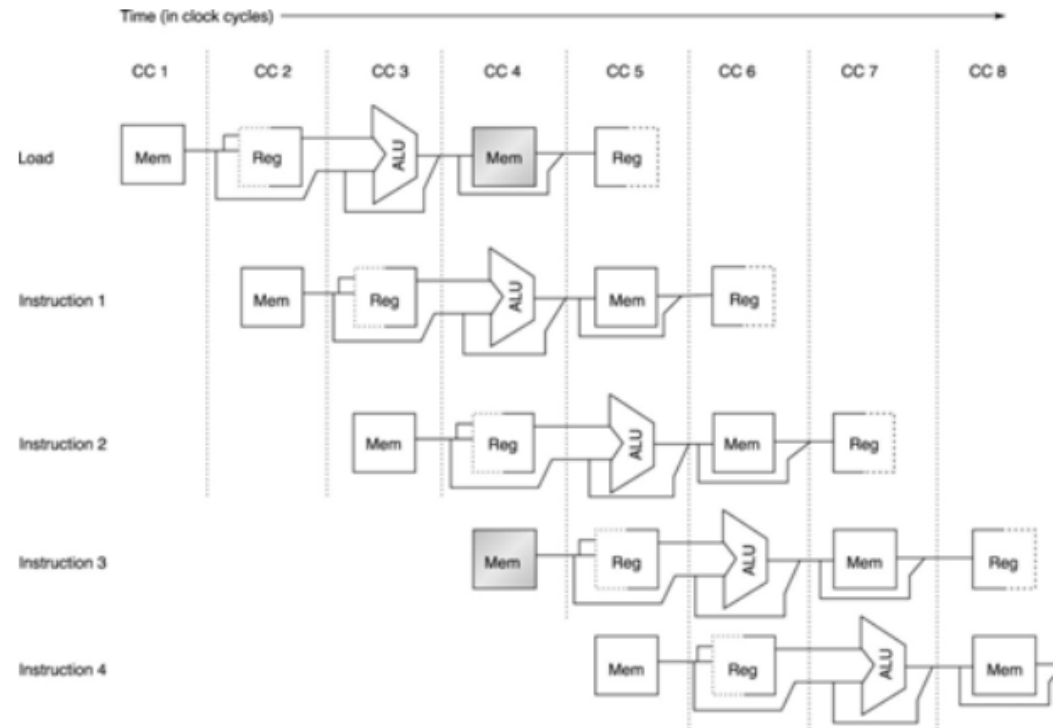
$$\begin{aligned}\text{Clock cycle pipelined} &= \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}} \\ \text{Pipeline depth} &= \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} \\ \text{Speedup from pipelining} &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} \\ &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}\end{aligned}$$

Structural Hazards

- Overlapping instructions require:
 - Pipelining of functional units
 - Duplication of resources
- Structural Hazard happens when the pipeline cannot accommodate **some combination of instructions**
- Common reasons for Structural Hazard
 - A unit is not fully pipelined → needs to finish current instruction before serving the next,
 - e.g. an instruction takes more than one clock cycle to go through.
 - Some resource has not been duplicated enough,
 - e.g. a register file has only one write port, but more than one instructions in pipeline may want to write at one stage.
- Consequences
 - Stall the pipeline until the resource **becomes available**, which creates a **Bubble**
 - Increase CPI from its ideal value

Structural Hazards: Examples

- Some pipelined processors have shared a single memory for data and instructions. As a result, when an instruction contains a data memory reference (at MEM), it will conflict with the instruction reference for a later instruction (at IF)



instruction	1	2	3	4	5	6	7	8	9
LW R10, 20(R1)	IF	ID	EX	MEM	WB				
SUB R11, R2, R3		IF	ID	EX	MEM	WB			
ADD R12, R3, R4			IF	ID	EX	MEM	WB		
STALL									
ADD R14, R5, R6					IF	ID	EX	MEM	WB

Structural Hazards Consideration / Trade-off

- Completely avoiding structural hazards is very costly :
 - Pipelined functional units are more costly (pipelined FP ALU are costly)
 - Duplication of resources (separate instruction and data caches means doubled memory and bandwidth)
- If Structural Hazard is rare, just let it be.
 - For example, if FP operations are rare, then just use a non-pipelined FP ALU.

Impact of Stalls on the Performance of Pipelined Implementation - Example

- Implementation without structural hazards
 - ideal CPI = 1
 - Clock Cycle = 1ns
- Implementation with the “load” structural hazard
 - Clock Cycle = 0.9ns
- Suppose that
 - 40% of the executed instructions are loads or stores
- Which implementation is faster?
 - $(\text{averageInstructionTime})_{\text{noHaz}} = 1 * 1 = 1$
 - $(\text{averageInstructionTime})_{\text{haz}} = (1 + 0.4 * 1) * 0.9 = 1.26$
 - implementation without structural hazard is 1.26 times faster than the other

Impact of Stalls on the Performance of Pipelined Implementation

$$\text{speedupFromPipelining} = \frac{[CPI]_{\text{nonpipelined}} \times [CCT]_{\text{nonpipelined}}}{[CPI]_{\text{pipelined}} \times [CCT]_{\text{pipelined}}}$$

$$\begin{aligned}[CPI]_{\text{pipelined}} &= \text{idealCPI} + \text{pipelineStallCyclesPerInstruction} \\ &= 1 + \text{pipelineStallCyclesPerInstruction}\end{aligned}$$

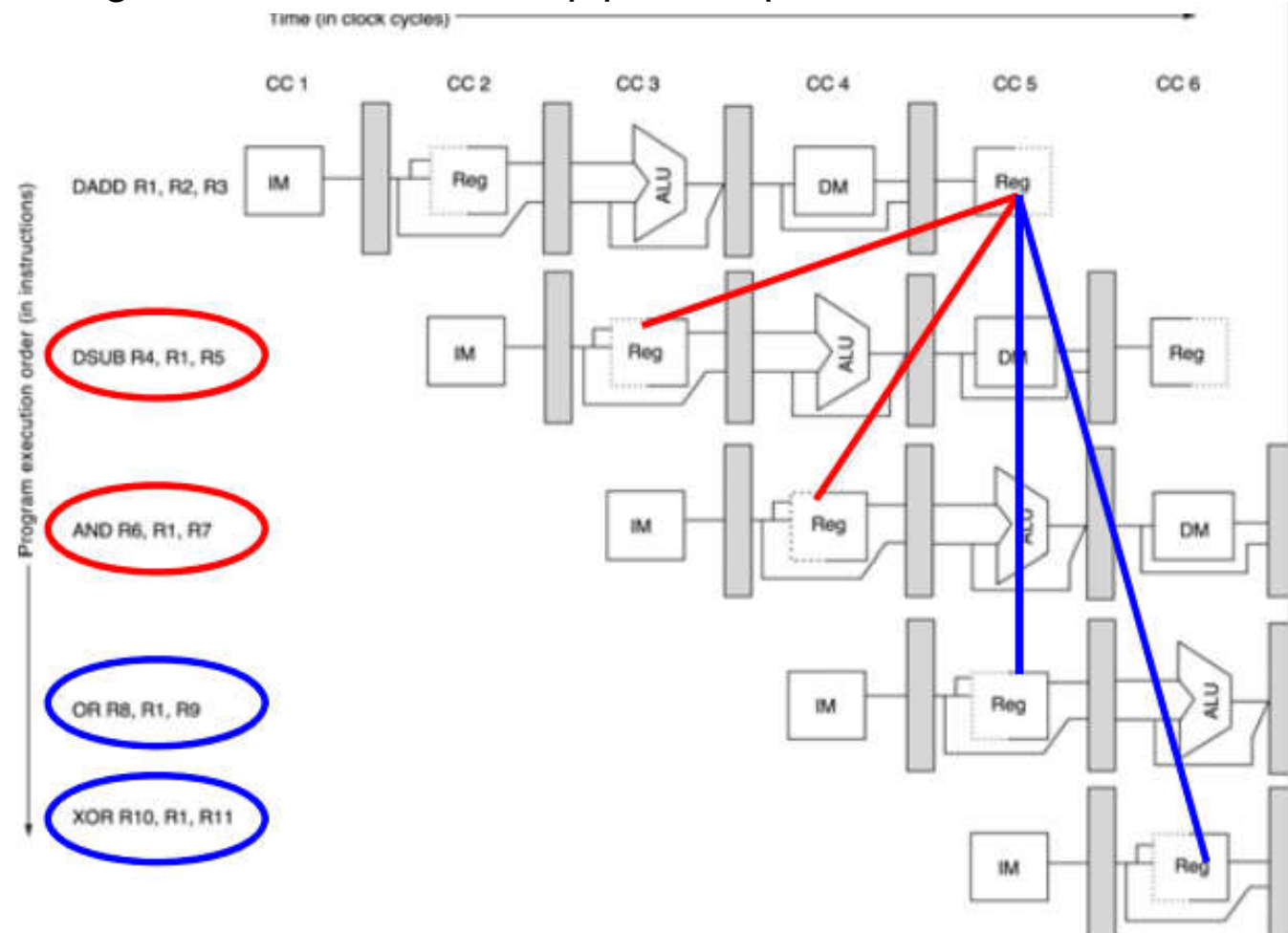
CCT = Clock Cycle Time

- Special case: ignoring the pipeline clock-period overhead and assume that
 - the pipeline stages are perfectly balanced
 - all instructions take the same number of cycles in the non-pipelined implementation (equal to the pipeline depth)

$$\text{speedupFromPipelining} = \frac{\text{pipelineDepth}}{1 + \text{pipelineStallCyclesPerInstruction}}$$

Data Hazards

Data Hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor



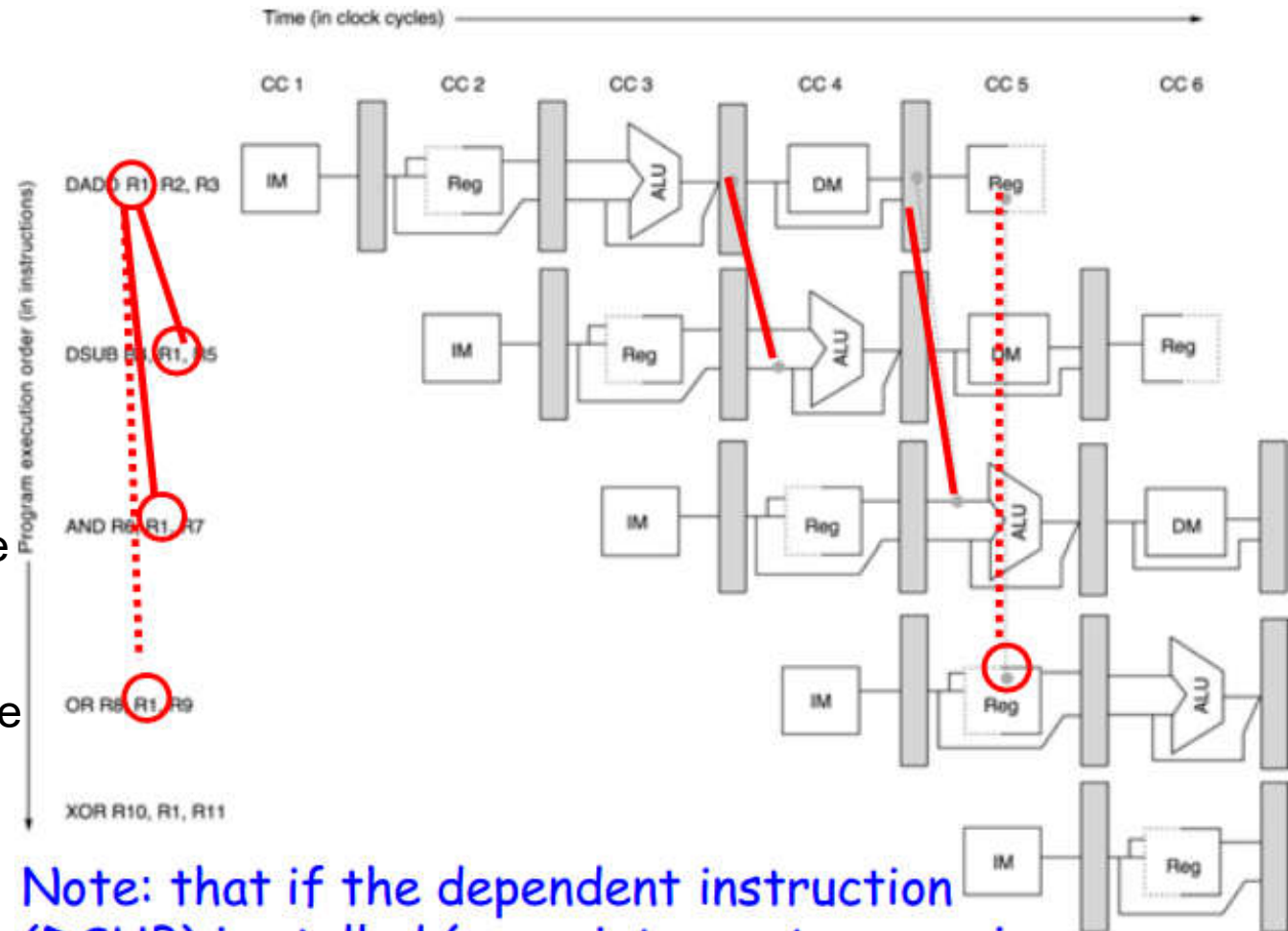
The red lines show data hazards.

This data hazard is called RAW (read after write)

Data Hazards: Forwarding (or By-Passing)

- ALU result is always fed back to the ALU input from both EX/MEM and MEM/WB

If the forwarding hardware detects that the previous ALU operation has written 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.



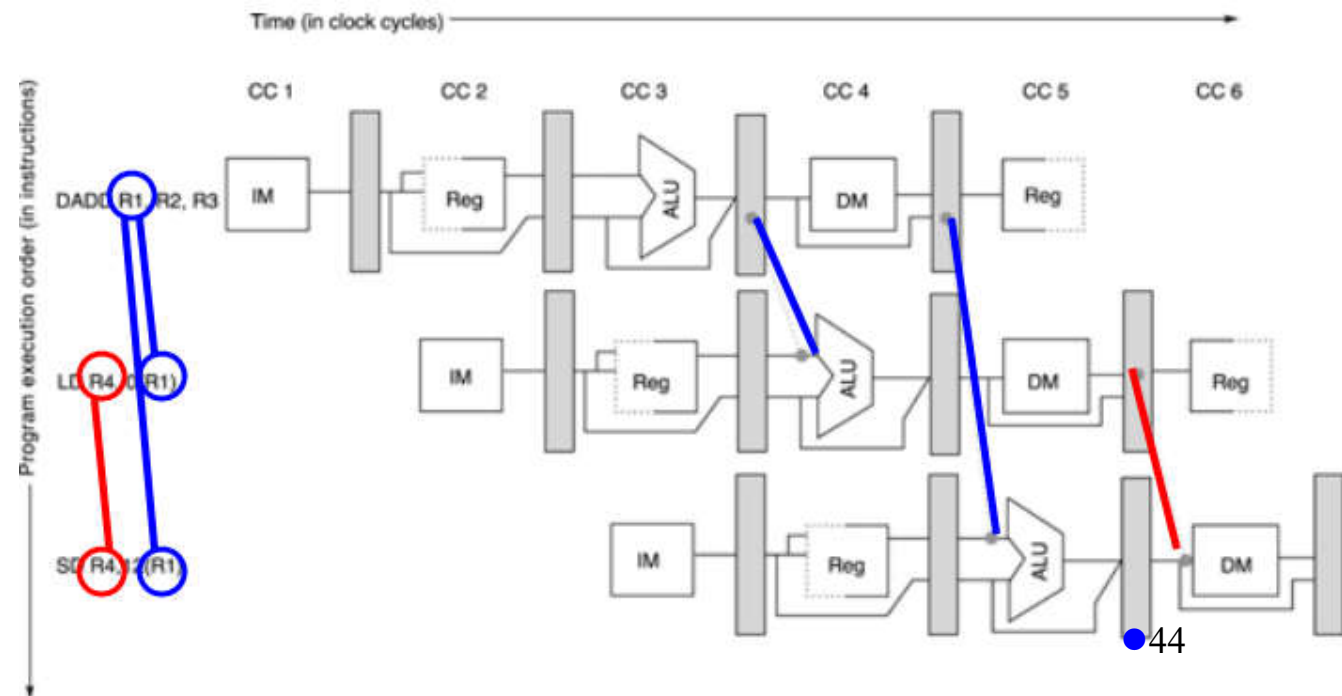
Note: that if the dependent instruction (DSUB) is stalled (or an interrupt occurs in between the two instructions) then the forward path will not be activated

Generalized Forwarding

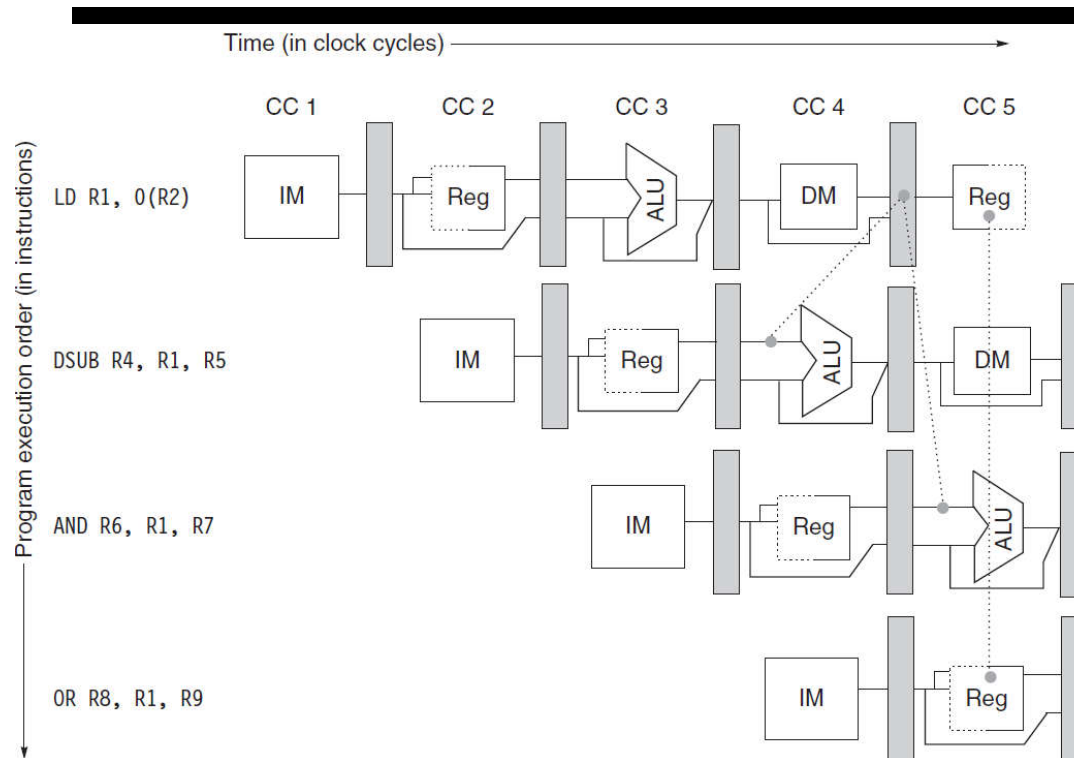
Sometime we need to forward results not only from the immediately previous instruction but also possibly from an instruction started 2 cycles earlier.

Generalized Forwarding: A result can be forwarded from a pipeline register of one unit to the input of another unit.

Note, without forwarding, units only receive operands from its own pipeline register.



Not all Data Hazards can be solved by Forwarding



A pipeline **interlock** (a hardware unit) detects a hazard and stalls the pipeline until the hazard is cleared.

LD	R1,0(R2)	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

Control / Branch Hazards

Greater Performance Loss than Data Hazards

When a branch is executed, there could be two outcomes:

1. If a branch is taken (**taken branch**), PC will be changed to the branch address at the end of ID after the completion of the address calculation and comparison.
2. Otherwise it is called **untaken branch** or fall through, PC is incremented by 4 at the end of IF, PC + 4

Control Hazard: Where should the next IF get the instruction?

Simple solution: Always do 2 back-to-back IFs in the next instruction execution when a Branch instruction is decoded.

The 1st IF fetches from PC+4, like a stall.

Depending on the outcome of ID of Branch, the 2nd IF either re-fetches the same instruction from PC+4 in case fall through, or fetch the instruction from the address calculated by ID of the previous instruction.

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Reducing Pipeline Branch Penalties

One stall cycle for every Branch will yield a performance loss of 10% to 30% depending on the branch frequency!

- Tech. 1: freeze or flush pipeline, like the two back-to-back IF
 - Compiler holds or deletes any instructions after the branch until the branch destination is known
 - Simple in hardware/software, but the penalty is fixed and can't be reduced.
- Tech 2: Guess Branch always Not Taken/Fall Through
 - Always fetch IF from PC+4.
 - In case Guess is right, continue as normal.
 - In case Guess is wrong, since the instruction is at the ID stage, instead of decoding it, change it to a NOOP.
 - Fetch the instruction from the address outcome of Branch instruction.

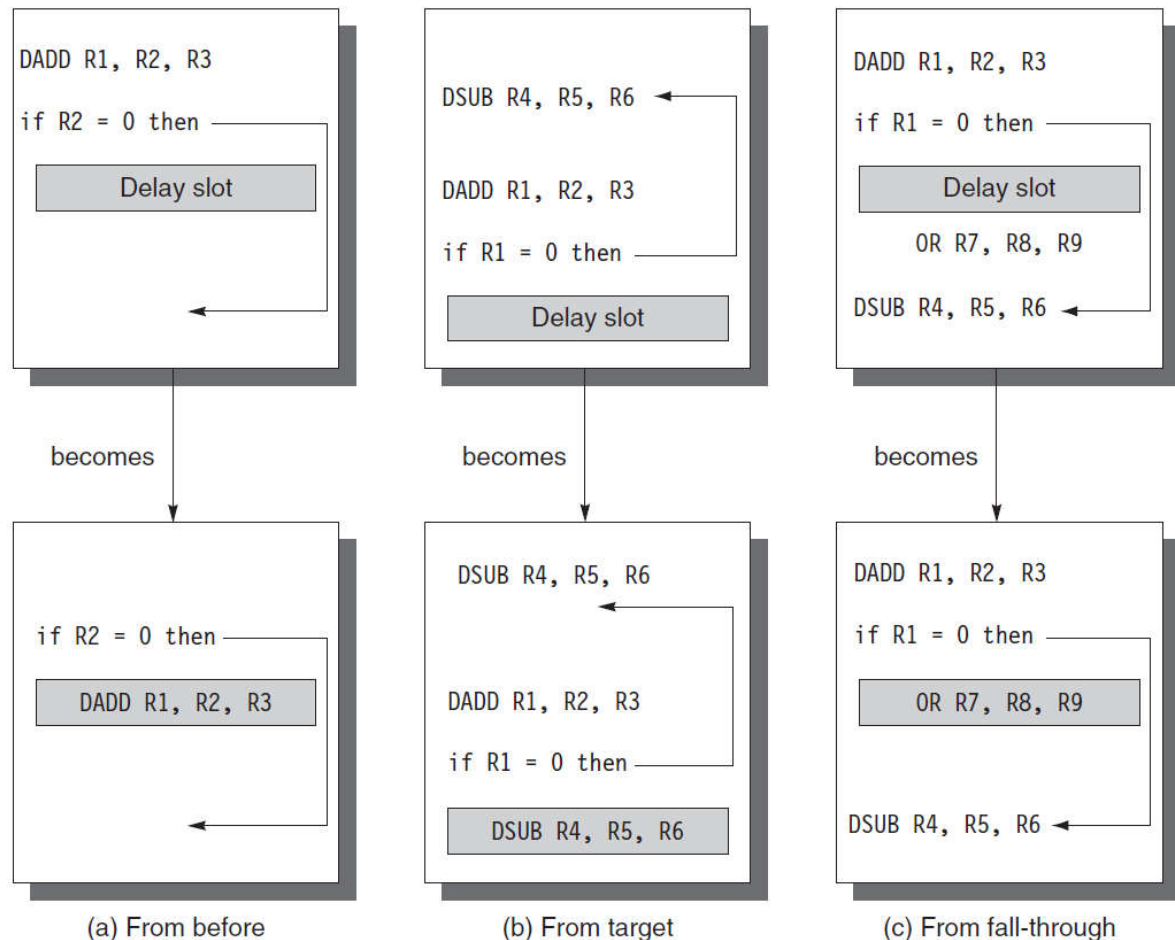
Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Reducing Pipeline Branch Penalties

- Tech. 3: Use a delayed branch
 - **Compiler** schedules one instruction (or more than one depending on the need) right after the branch. The instruction(s) is always executed, after which we will know the ID outcome of the Branch instruction.

(a) is the best.
(b) and (c) are used only when (a) is impossible.
(b) is preferred if Branch has higher probability to be taken.
(c) is preferred if otherwise.
Both (b) and (c) need to undone their changes (registers for example) if their assumptions turn out to be false!



Delayed Branch Limitations and Improvement

- Limitations:
 - the restrictions on the instructions that are scheduled into the delay slots
 - our ability to predict at compile time whether a branch is likely to be taken or not
- Processor introduces a new instruction called Canceling or Nullifying
 - A canceling branch instruction includes the instruction from a predicated direction.
 - If predication is right, the canceling instruction continues.
 - Otherwise, this special instruction changes to NOOP.

Performance of Branch Schemes

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

$$\text{Pipeline stall cycles from branches} = \text{Branch frequency} \times \text{Branch penalty}$$

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Control hazards can cause a greater performance loss for a pipelined implementation than data hazards. In first approximation:

Performance loss can vary **between 10% and 30%** depending on the branch frequency. Generally, the **deeper the pipeline, the worse the branch** penalty (measured in clock cycles)

Example: Branch Penalty (in CPI units) on the MIPS R4000 Processor

Branch scheme	Penalty for jumps	Penalty for untaken branches	Penalty for taken branches
Flush Pipeline	2	3	3
Predict Taken	2	3	2
Predict Untaken	2	0	3

MIPS R4000 takes 3 cycles to compute the target address (i.e. minimum 2 CPI of *additional* penalty) and 4 cycles to know the branch outcome

Strategy	Jumps	Untaken branches	Taken branches	Combined
Frequency (est.)	4%	6%	10%	20%
Flush Pipeline	0.08	0.18	0.30	0.56
Predict Taken	0.08	0.18	0.20	0.46
Predict Untaken	0.08	0.00	0.30	0.38

Regarding Branch Instructions

- Some Branches in some implementations are finished at **MEM**, new PC available at the next cycle.
 - ID decodes it, EXE calculates the target address and tests the condition, and MEM writes the result to PC.
- Some Branches in some implementations are finished at **EXE**, new PC available at the next cycle. (Most Conditional Branches)
 - ID decodes it, EXE calculates the target address and tests the condition and writes the result to PC.
- Some Branches in some implementations are finished at **ID**, new PC available at the next cycle (Most Jumps or System Calls)
 - At the ID stage, while the instruction is being decoded, the target address calculation and condition evaluation are also calculated **by assuming it is a Branch**. At the end of ID, the PC will be updated with either PC+4 (the assumption is wrong), or the new calculated target (the assumption is right).
 - An dedicated adder is needed at ID stage for target calculation.

Terminology for Instruction States

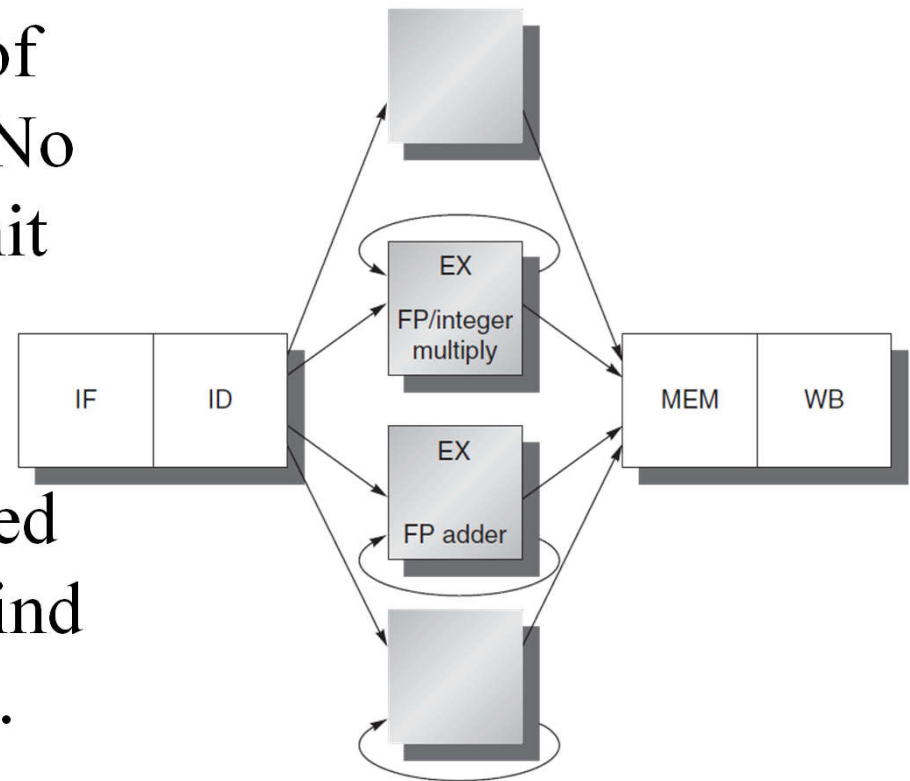
- **Fetch**: instruction has been retrieved from the I-Cache
- **Decode**: instruction has completed the ID stage
- **Issue**: instruction has been decoded and has started its execution
 - In MIPS, it has moved from ID to EX
- **Execute**: instruction has completed its execution in the EX stage
- **Commit**: instruction is guaranteed to complete
 - In MIPS, it has entered the WB (or the end of the MEM stage) and the status vector doesn't contain any exception flag for it

Extend to multi-cycle operations

- The EX of FP cannot be done in 1 or even 2 cycles.
- FP instructions can have the same pipeline as the integer instructions, with two important changes:
 - First, there could be **multiple EX cycles for a FP** operation—the number of cycles can vary for different operations.
 - Second, there may be **multiple FP units**. A stall will occur if the instruction to be issued will cause either a structural hazard for the unit it uses or a data hazard.
- Job description of the 4 units:
 - The main **integer** unit handles loads/stores, integer ALU, Branches
 - FP **adder** handles FP add, subtract, and conversion
 - FP and integer **multiplier**
 - FP and integer **divider**

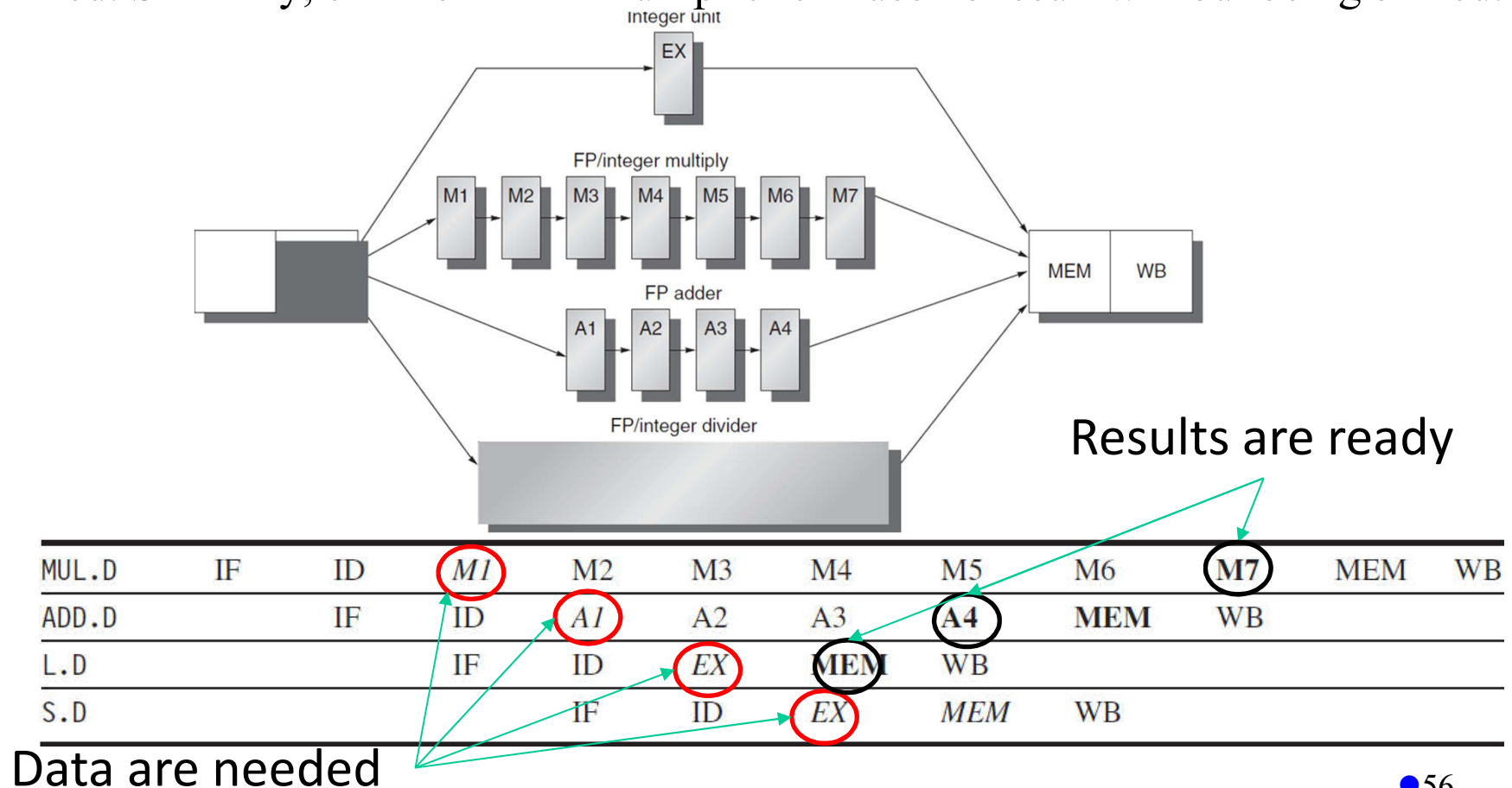
The MIPS pipeline with unpipelined FP units.

- Assume that the EXE stages of these units are not pipelined. No other instruction using that unit may issue until the previous instruction leaves EX.
- If an instruction cannot proceed to EX, the entire pipeline behind that instruction will be stalled.



A pipeline supporting multiple outstanding FP operations.

- The FP multiplier and adder are pipelined and have a depth of 7 and 4 stages.
- The FP divider is not pipelined, but requires 24 cycles to complete.
- For example, the 4th instruction after an FP add can use its result without being stalled. Similarly, 7th after an FP multiplier can use its result without being stalled.



Latency and Initiation Interval

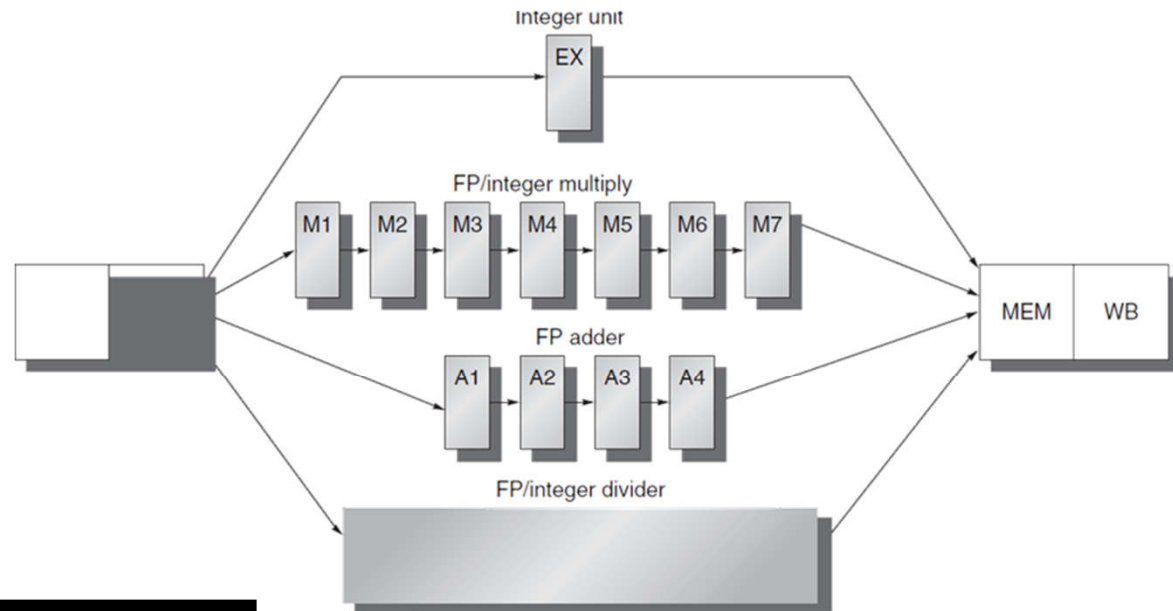
Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

- Two important terms of pipelined operations
 - Latency: the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.
 - Initiation interval: The number of cycles that must elapse between issuing two operations of a given type.
- Since most operations consume their operands at the beginning of EX, the latency is usually the # of stages after EX that an instruction produces a result
 - Integer ALU operations are all 0;
 - Loads are 1, since their results can be used after one intervening cycle.
 - Stores, which consume the value being stored 1 cycle later, are a little different. The latency for the value being stored will be 1 cycle less.
- Pipeline latency is essentially equal to 1 cycle less than the depth of the execution pipeline, which is the number of stages from the EX stage to the stage that produces the result.

Hazards due to Multiple Outstanding Operations

- Structural Hazards
 - sequential (non-pipelined) function units
 - e.g., two “close” DIV operations require stalling
 - number of register writes per cycle may be larger than 1
- More Frequent RAW Data Hazards
 - “instruction **y** (which follows instruction **x** in the program order) tries to read a source register before **x** writes it”
 - longer latency of operations produces more stalls
- Two New Classes of Data Hazards
 - WAW
 - “instruction **y** tries to write a destination register before **x** writes it”
 - WAR – is not possible
 - “instruction **y** tries to write a destination register before **x** reads it”
- Out-of-order completion
 - makes exception handling more complex

Example: WAW Hazard



```
DIV.D F1, F2, F4  
BEQZ R2, target  
LD F9, 0(R10)  
ADD.D F1, F5, F6  
...; other non-branch instructions  
target: SUB.D F8, F9, F1
```

Hazards due to Multiple Outstanding Operations – Example of WAW Hazard

- WAW
 - “instruction **y** tries to write a destination register before **x** writes it”
- One may wonder how can WAW hazards occur since it may seem that a compiler would never generate two writes to the same register without an intervening read
- However such sequences do occur
 - e.g., a program may take a path that the compiler cannot predict, e.g. due to an intervening branch
- Also, in the case of multiple-issue pipelines, an intervening read may be present (and stalled) while the second write could progress

```
DIV.D F1, F2, F4
BEQZ R2, target
LD F9, 0(R10)
ADD.D F1, F5, F6
...; other non-branch instructions
target: SUB.D F8, F9, F1
```

- In this example, if the branch is not taken, a WAW is possible when the DIV completes its execution after the BEQZ, LD, and ADD complete theirs

Hazards due to Multiple Outstanding Operations – Example of Structural Hazard

Instruction	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
ADD R1, R3, R4		IF	ID	EX	MEM	WB					
ADD R2, R4, R5			IF	ID	EX	MEM	WB				
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
ADD R6, R9, R7					IF	ID	EX	MEM	WB		
ADD R8, R3, R9						IF	ID	EX	MEM	WB	
L.D F2, 0(R2)		Fig. C.38					IF	ID	EX	MEM	WB

- With only 1 write port in register file, the processor must serialize the 3 WB instructions
 - Option 1:** Tracking port access during ID and (pre)-stalling corresponding instruction (same logic as for interlock detection)
 - Option 2:** Stall the conflicting instructions with smaller latency (“least-waited”) just before MEM or WB
 - Notice that there is no structural hazard at MEM because, among the 3 instructions, only L.D really needs to access memory. However, this assumes that there is extra HW to make the other two instructions progress into WB as they need

Hazards due to Multiple Outstanding Operations – Example of RAW Hazard



















Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D F4, 0(R2)	IF	ID	EX	MEM	WB												
MUL.D F0,F4,F6		IF	ID		M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2,F0,F8			IF		ID							A1	A2	A3	A4	MEM	WB
S.D F2, 0(R2)					IF							ID	EX				MEM

Fig. C.37

- Deeper pipelines increase the stall frequencies
 - each instruction in the example depends on the previous one
 - each instruction proceeds as soon as the awaited data become available (optimal forwarding)
 - to avoid a RAW hazard the S.D must wait for the ADD.D to provide the new value of F2 (via bypassing) before entering MEM

*Note: Fig. C.37 is not consistent with Fig. C.38 on the handling of structural hazards: it implies that a structural hazard exists on MEM in Cycle 16; this happens when no extra hardware handles the conflict

Hazards due to Multiple Outstanding Operations – Control Implementation

- Assuming that all hazard detections are done in ID, before issuing an instruction **x** we must
 - check for structural hazards
 - the required function unit must be available (e.g., FP Divisor)
 - no future conflicts on the register write port
 - check for RAW data hazards
 - no conflict between **x**'s source register and previous instruction destination registers
 - e.g. if **x** is a FP operation on F2, check that F2 is not a destination in ID/A1, A1/A2, A2/A3
 - check for WAW data hazards
 - no instruction in A1,...,A4, D, M1,...M7 can have the same register destination as **x**
- Logic implementation is similar as for the MIPS integer pipeline

What Makes Pipelining Hard to Implement?

- Exceptional situations where the instruction execution order needs to be changed in unexpected ways.
 - Exceptions and Interrupts.
- Different instruction sets (skipped)

Exceptions: hard to handle

- An instruction in the pipeline can raise exceptions that may force the CPU to abort the following instructions in the pipeline before they complete.
 - CPU needs to make sure that the aborted instructions do not change the CPU state.
- Types of exceptions:
 - I/O device request
 - Invoking an operating system service from a user program
 - Tracing instruction execution
 - Breakpoint (programmer-requested interrupt)
 - Integer arithmetic overflow
 - FP arithmetic anomaly
 - Page fault (not in main memory)
 - Misaligned memory accesses (if alignment is required)
 - Memory protection violation
 - Using an undefined or unimplemented instruction
 - Hardware malfunctions
 - Power failure

Characterizing Exceptions

- Synchronous
 - event occurs at the same place every time a program is executed with the same data
- Coerced
 - Caused by HW event not under control of the user program
 - harder to handle, unpredictable
- User Maskable
 - can be disabled by user program
- Within Instructions
 - typically synchronous
 - recognized after instruction has triggered by instructions completed
- Resume
 - program execution must be completed after exception
- Asynchronous
 - caused by device external to CPU and memory
 - easier to handle, wait for completion of current instruction
- User Requested
 - requested by user program, these are not really exceptions, but still treated in the same way
 - easier to handle, wait for completion of current instruction
- Unmaskable
 - cannot be disabled by user program
- Between Instructions
 - recognized after instruction has completed
- Terminate
 - event stops program execution
 - easier to handle

Maintaining Precise Exceptions with Long-Running Instructions

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

- No dependencies, but likely we have out-of-order completion
- Possible Imprecise Exception Problem if, for instance,
 - **SUB.D** causes an exception at the point where **ADD.D** has already completed but **DIV.D** has not yet
 - further, if also **DIV.D** causes an exception, the state of the system is left inconsistent i.e.
 - as before **DIV.D** was executed but with a new value for **F10**
- Possible solutions
 1. ignore the problem and settle for Imprecise Exception
 2. buffer the results until all previously issued operations complete
 3. let the trap-handling routine rebuild the sequence of exceptions
 4. allow instructions to complete only if all previously issued ones cannot have exception anymore (implemented using stalling) •67

处理中断4种可能的办法（1,2两种）

- 方法1：忽略这种问题，当非精确处理
 - 原来的supercomputer的方法
 - 但现代计算机对IEEE 浮点标准的异常处理，虚拟存储的异常处理要求必须是精确中断。
- 方法2：缓存操作结果，直到早期发射的指令执行完。
 - 当指令运行时间较长时，Buffer区较大
 - Future file (Power PC620 MIPS R10000)
 - 缓存执行结果，按指令序确认
 - history file (CYBER 180/990)
 - 尽快确认
 - 缓存区存放原来的操作数，如果异常发生，回卷到合适的状态

第3 & 4种方法

- 方法3：以非精确方式处理，用软件来修正
 - 为软件修正保存足够的状态
 - 让软件仿真尚未执行完的指令的执行
 - 例如
 - Instruction 1 – A 执行时间较长，引起中断的指令
 - Instruction 2, instruction 3,instruction n-1 未执行完的指令
 - Instruction n 已执行完的指令
 - 由于第n条指令已执行完，希望中断返回后从第n+1条指令开始执行，如果我们保存所有的流水线的PC值，那么软件可以仿真Instruction1 到Instruction n-1 的执行
- 方法4：暂停发射，直到确定先前的指令都可无异常的完成，再发射下面的指令。
 - 在EX段的前期确认（MIPS流水线在前三个周期中）
 - MIPS R2K to R4K 以及Pentium使用这种方法

Homework #2

- Page C-82: Question C.1, C.3, C.7

In-class Quiz #1