

PIPELINING: BASIC AND INTERMEDIATE CONCEPTS

CHAPTER # 03



WHAT IS PIPELINING?

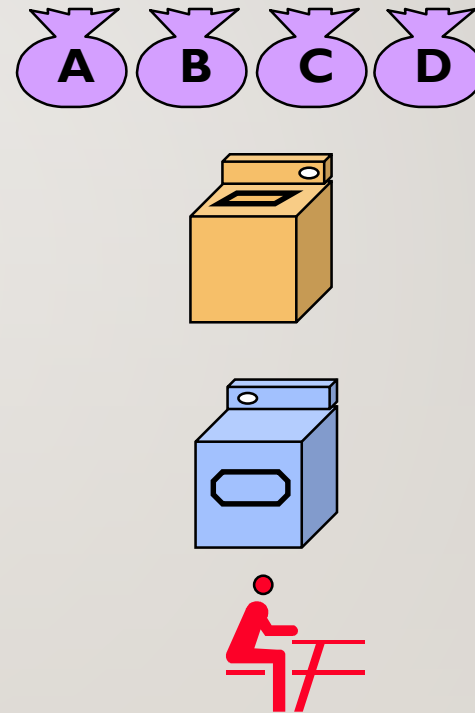
- Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction.
- Today, pipelining is the key implementation technique used to make fast processors, and even processors that cost less than a dollar are pipelined.
- A pipeline is like an assembly line. In an automobile assembly line, there are many steps, each contributing something to the construction of the car.

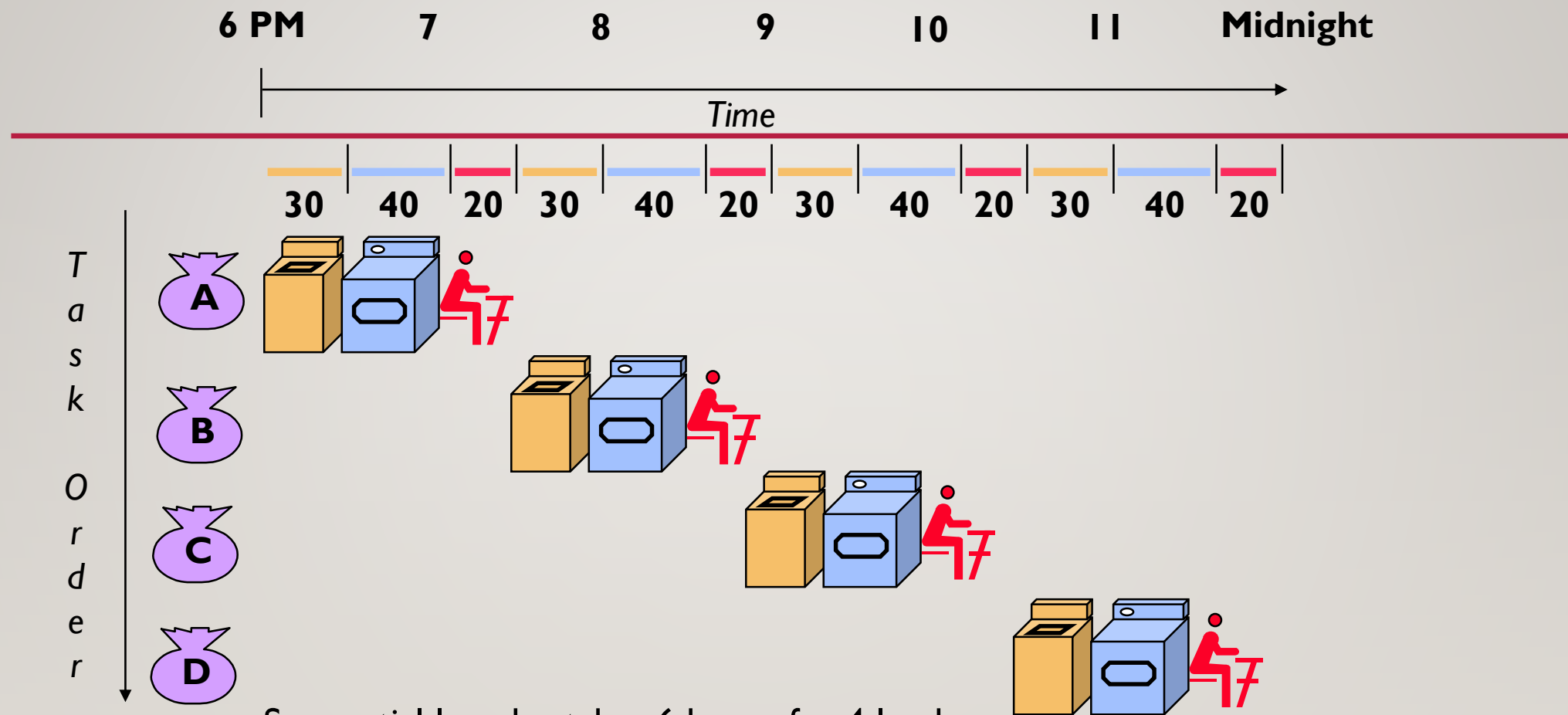
WHAT IS PIPELINING?

- Each step operates in parallel with the other steps, although on a different car.
- In a computer pipeline, each step in the pipeline completes a part of an instruction. Like the assembly line, different steps are completing different parts of different instructions in parallel. Each of these steps is called a pipe stage or a pipe segments.
- The stages are connected one to the next to form a pipe—instructions enter at one end, progress through the stages, and exit at the other end, just as cars would in an assembly line.

WHAT IS PIPELINING

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

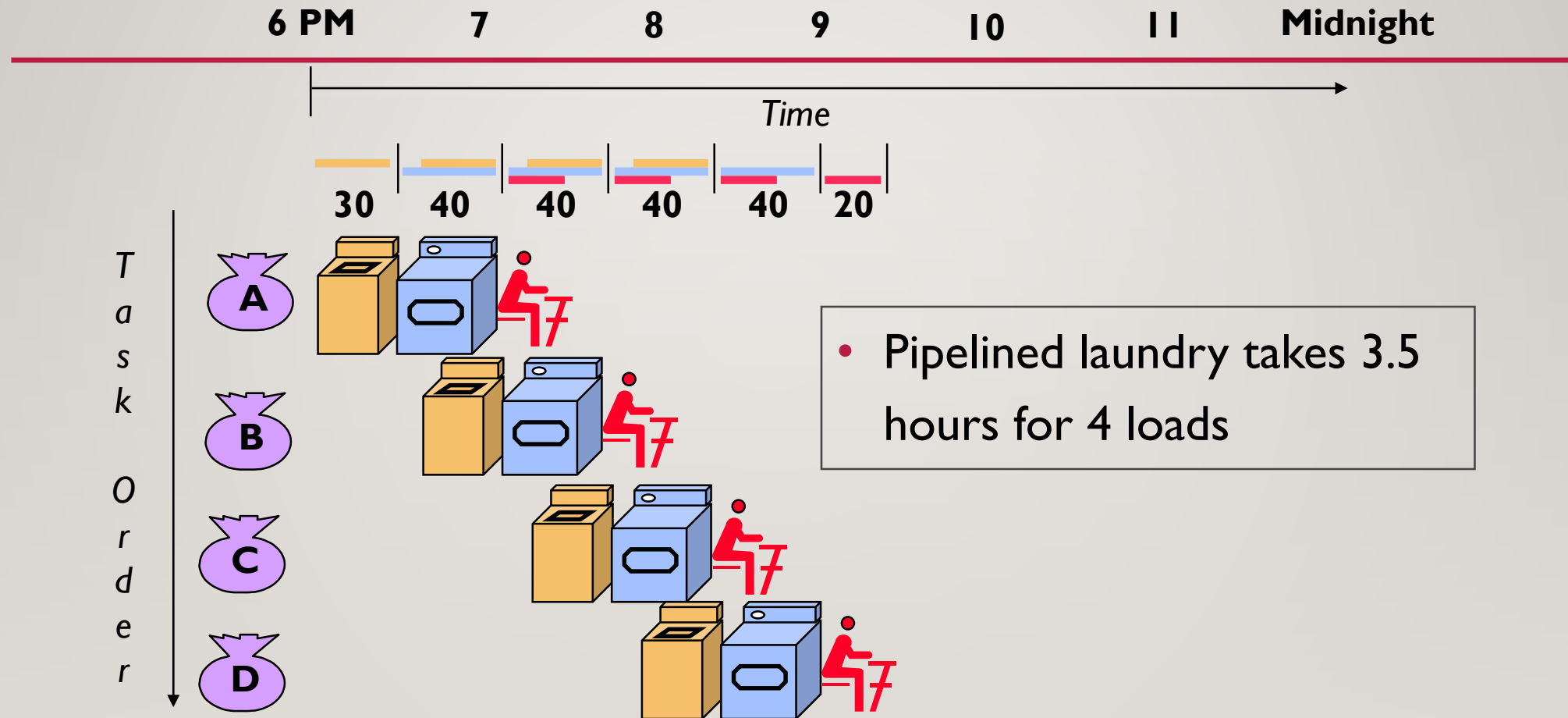




If they learned pipelining, how long would laundry take?

WHAT IS PIPELINING

START WORK ASAP



WHAT IS PIPELINING

- In an automobile assembly line, throughput is defined as the number of cars per hour and is determined by how often a completed car exits the assembly line.
- Likewise, the throughput of an instruction pipeline is determined by how often an instruction exits the pipeline.
- Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time, just as we would require in an assembly line.

WHAT IS PIPELINING

- The time required between moving an instruction one step down the pipeline is a processor cycle.
- Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage, just as in an auto assembly line the longest step would determine the time between advancing cars in the line.
- In a computer, this processor cycle is almost always 1 clock cycle.

WHAT IS PIPELINING

- The pipeline designer's goal is to balance the length of each pipeline stage, just as the designer of the assembly line tries to balance the time for each step in the process. If the stages are perfectly balanced, then the time per instruction on the pipelined processor—assuming ideal conditions—is equal to:

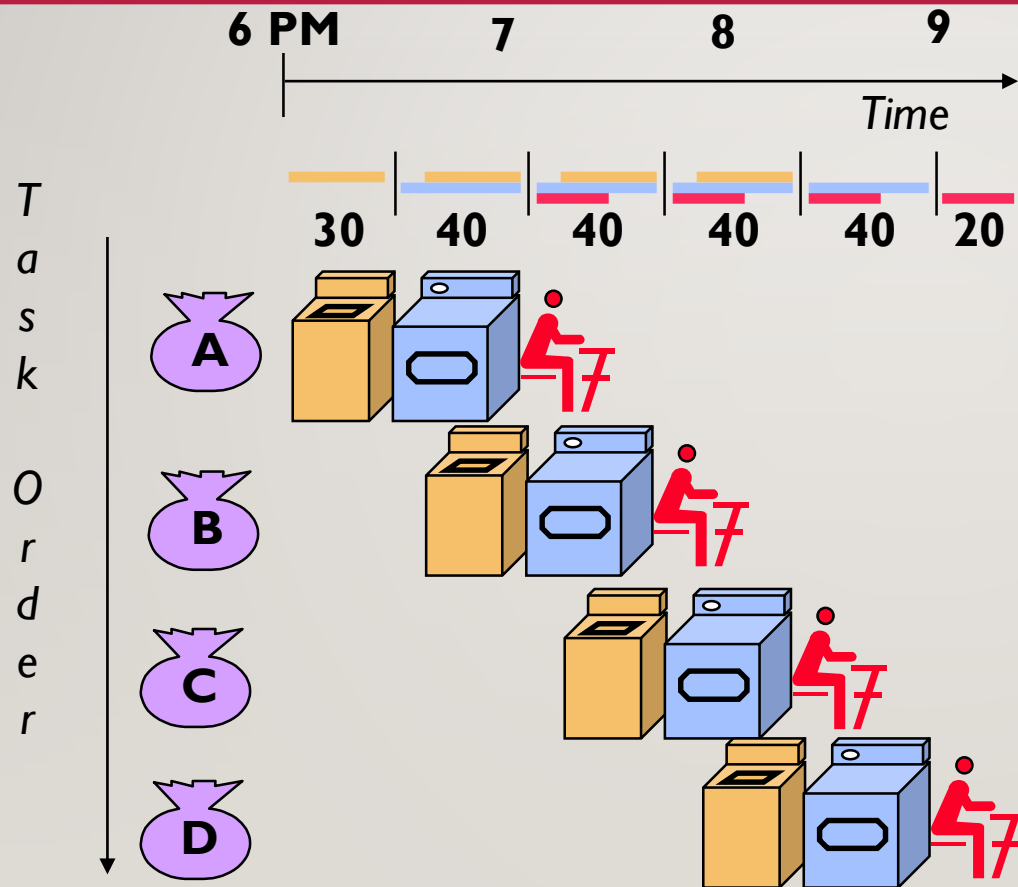
$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

WHAT IS PIPELINING

- Under these conditions, the speedup from pipelining equals the number of pipe stages, just as an assembly line with n stages can ideally produce cars n times as fast. Usually, however, the stages will not be perfectly balanced; furthermore, pipelining does involve some overhead.
- Thus, the time per instruction on the pipelined processor will not have its minimum possible value, yet it can be close.
- Pipelining yields a reduction in the average execution time per instruction. If the starting point is a processor that takes multiple clock cycles per instruction, then pipelining reduces the CPI. This is the primary view we will take.

What Is Pipelining

PIPELINING LESSONS



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

A SIMPLE IMPLEMENTATION OF A RISC INSTRUCTION SET

- Every instruction in this RISC subset can be implemented in, at most, 5 clock cycles. The 5 clock cycles are as follows.
- **Instruction fetch cycle (IF):**
- Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential instruction by adding 4 (because each instruction is 4 bytes) to the PC.

A SIMPLE IMPLEMENTATION OF A RISC INSTRUCTION SET

- **Instruction decode/register fetch cycle (ID):**
- Decode the instruction and read the registers corresponding to register source specifiers from the register file.
- 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 SIMPLE IMPLEMENTATION OF A RISC INSTRUCTION SET

- **Execution/effective address cycle (EX):**
- The ALU operates on the operands prepared in the prior cycle, performing one of three functions, depending on the instruction type.
 - ■ Memory reference—The ALU adds the base register and the offset to form the effective address.
 - ■ Register-Register ALU instruction—The ALU performs the operation specified by the ALU opcode on the values read from the register file.
 - ■ Register-Immediate ALU instruction—The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.
 - ■ Conditional branch—Determine whether the condition is true.

A SIMPLE IMPLEMENTATION OF A RISC INSTRUCTION SET

- **Memory access (MEM):**
- If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.
- **Write-back cycle (WB):**
- Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

A SIMPLE IMPLEMENTATION OF A RISC INSTRUCTION SET

ALU Instructions: `op $x, $y, $z`

Instr. Fetch & PC Increm.	Read of Source Regs. \$y and \$z	ALU OP ($y \text{ op } z$)	Write Back of Destinat. Reg. \$x
------------------------------	-------------------------------------	---------------------------------	-------------------------------------

Load Instructions: `lw $x, offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. \$y	ALU Op. ($y + \text{offset}$)	Read Mem. $M(y + \text{offset})$	Write Back of Destinat. Reg. \$x
------------------------------	--------------------------	------------------------------------	-------------------------------------	-------------------------------------

Store Instructions: `sw $x, offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. \$y & Source \$x	ALU Op. ($y + \text{offset}$)	Write Mem. $M(y + \text{offset})$
------------------------------	---------------------------------------	------------------------------------	--------------------------------------

Conditional Branch: `beq $x, $y, offset`

Instr. Fetch & PC Increm.	Read of Source Regs. \$x and \$y	ALU Op. ($x - y$) & ($PC + 4 + \text{offset}$)
------------------------------	-------------------------------------	---

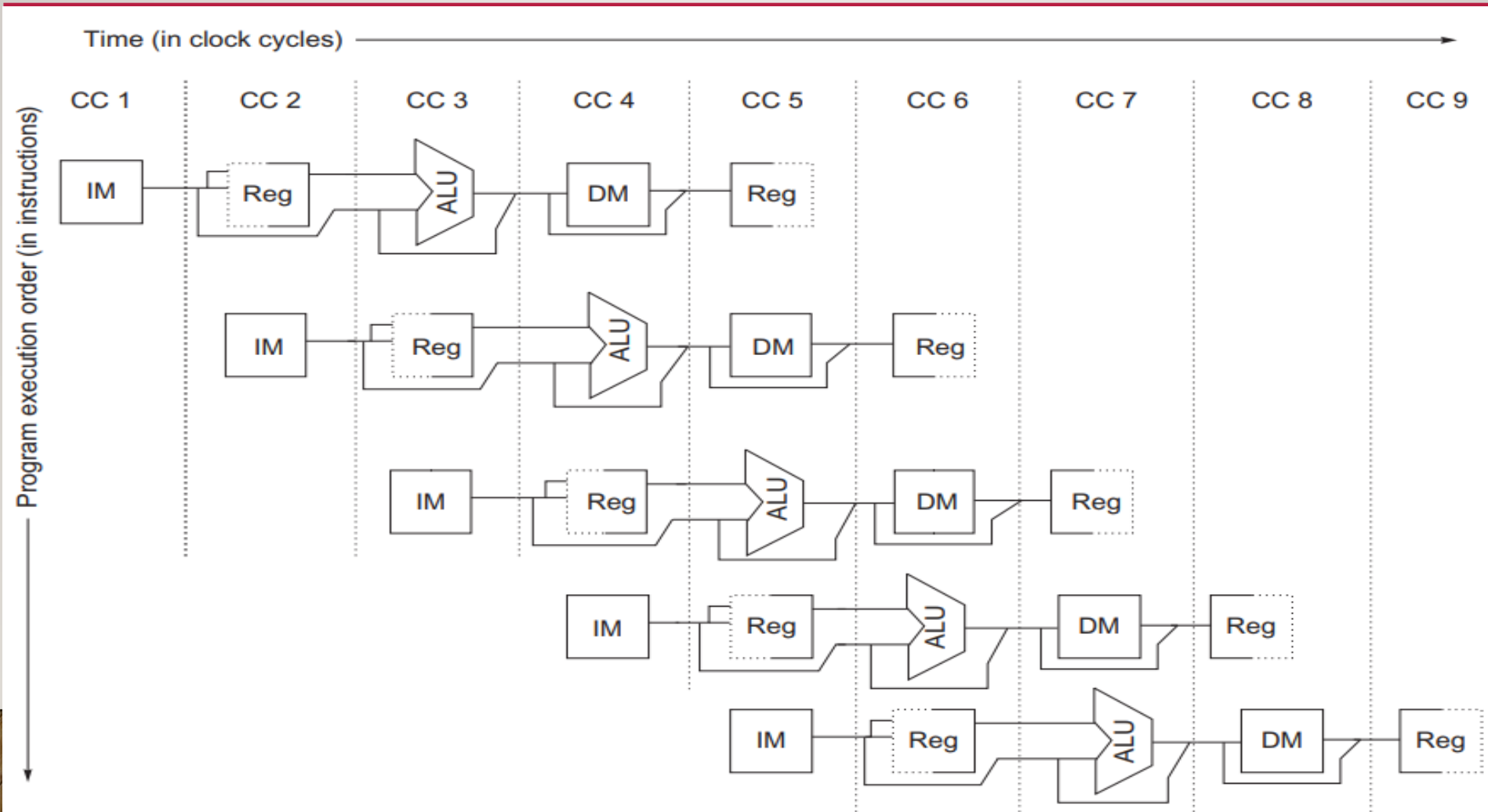
THE CLASSIC FIVE-STAGE PIPELINE FOR A RISC PROCESSOR

- We can pipeline the execution described in the previous section with almost no changes by simply starting a new instruction on each clock cycle.
- Each of the clock cycles from the previous section becomes a pipe stage—a cycle in the pipeline. This results in the execution pattern shown in Figure, which is the typical way a pipeline structure is drawn.
- Although each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions.

THE CLASSIC FIVE-STAGE PIPELINE FOR A RISC PROCESSOR

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

THE CLASSIC FIVE-STAGE PIPELINE FOR A RISC PROCESSOR



BASIC PERFORMANCE ISSUES IN PIPELINING

- Pipelining increases the processor instruction throughput—the number of instructions completed per unit of time—but it does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction due to overhead in the control of the pipeline.
- In addition to limitations arising from pipeline latency, limits arise from imbalance among the pipe stages and from pipelining overhead. Imbalance among the pipe stages reduces performance because the clock can run no faster than the time needed for the slowest pipeline stage. Pipeline overhead arises from the combination of pipeline register delay and clock skew.

EXAMPLE

- Consider the un-pipelined processor in the previous section. Assume that it has a **4 GHz** clock (or a **0.5 ns** clock cycle) and that it uses **four** cycles for ALU operations and branches and **five** cycles for memory operations. Assume that the relative frequencies of these operations are **40%**, **20%**, and **40%**, respectively. Suppose that due to clock skew and setup, pipelining the processor adds **0.1 ns** of overhead to the clock. Ignoring any latency impact, *how much speedup in the instruction execution rate will we gain from a pipeline?*

SOLUTION

- The average instruction execution time on the un-pipelined processor is:

$$\begin{aligned}\text{CPU time} &= \left(\sum_{i=1}^n \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time} \\ &= 0.5 \text{ ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5] \\ &= 0.5 \text{ ns} \times 4.4 \\ &= 2.2 \text{ ns}\end{aligned}$$

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 0.5 + 0.1 or 0.6 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

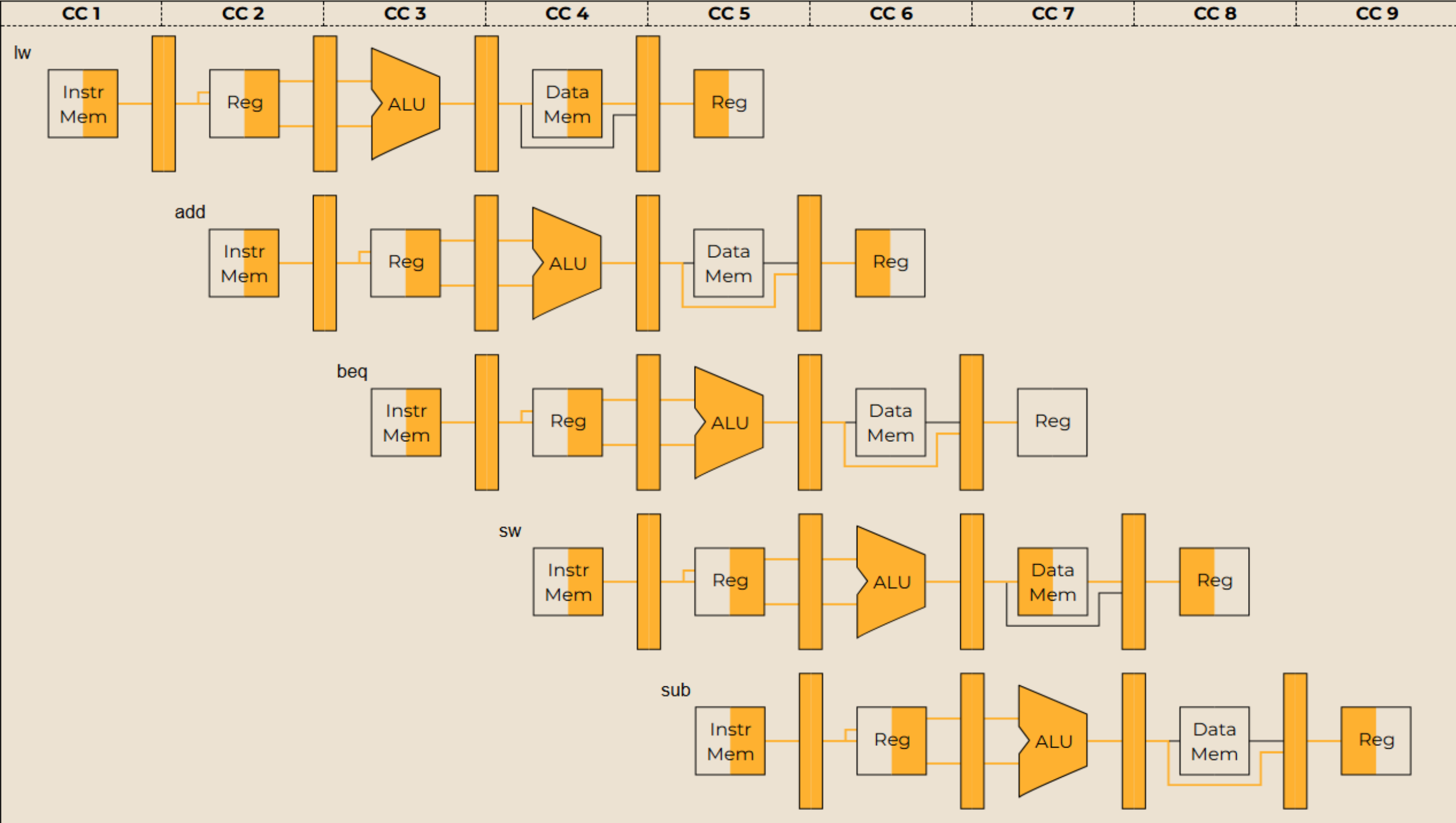
$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\ &= \frac{2.2 \text{ ns}}{0.6 \text{ ns}} = 3.7 \text{ times}\end{aligned}$$

THE MAJOR HURDLE OF PIPELINING—PIPELINE HAZARDS

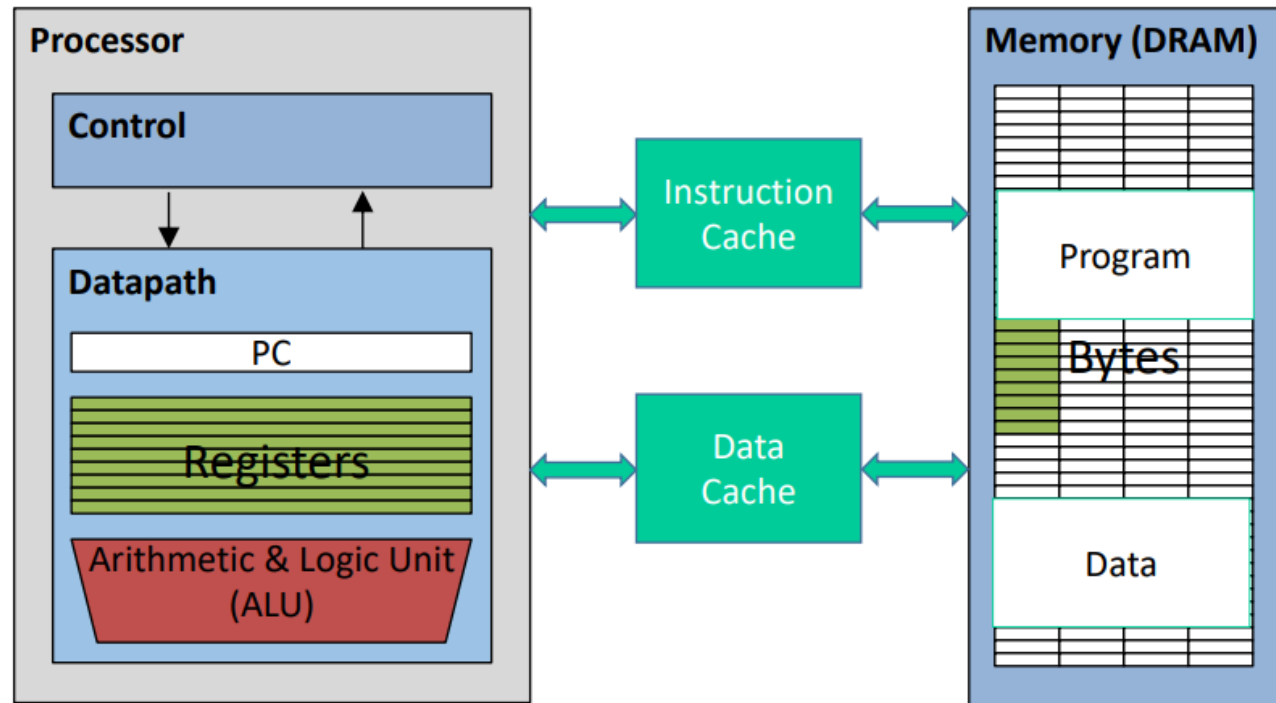
- There are situations, called hazards, that prevent the next instruction in the instruction stream from executing during its designated clock cycle.
- Hazards reduce the performance from the ideal speedup gained by pipelining.
- There are three classes of hazards:
 - 1. Structural hazards
 - 2. Data hazards
 - 3. Control hazards

THE MAJOR HURDLE OF PIPELINING—PIPELINE HAZARDS

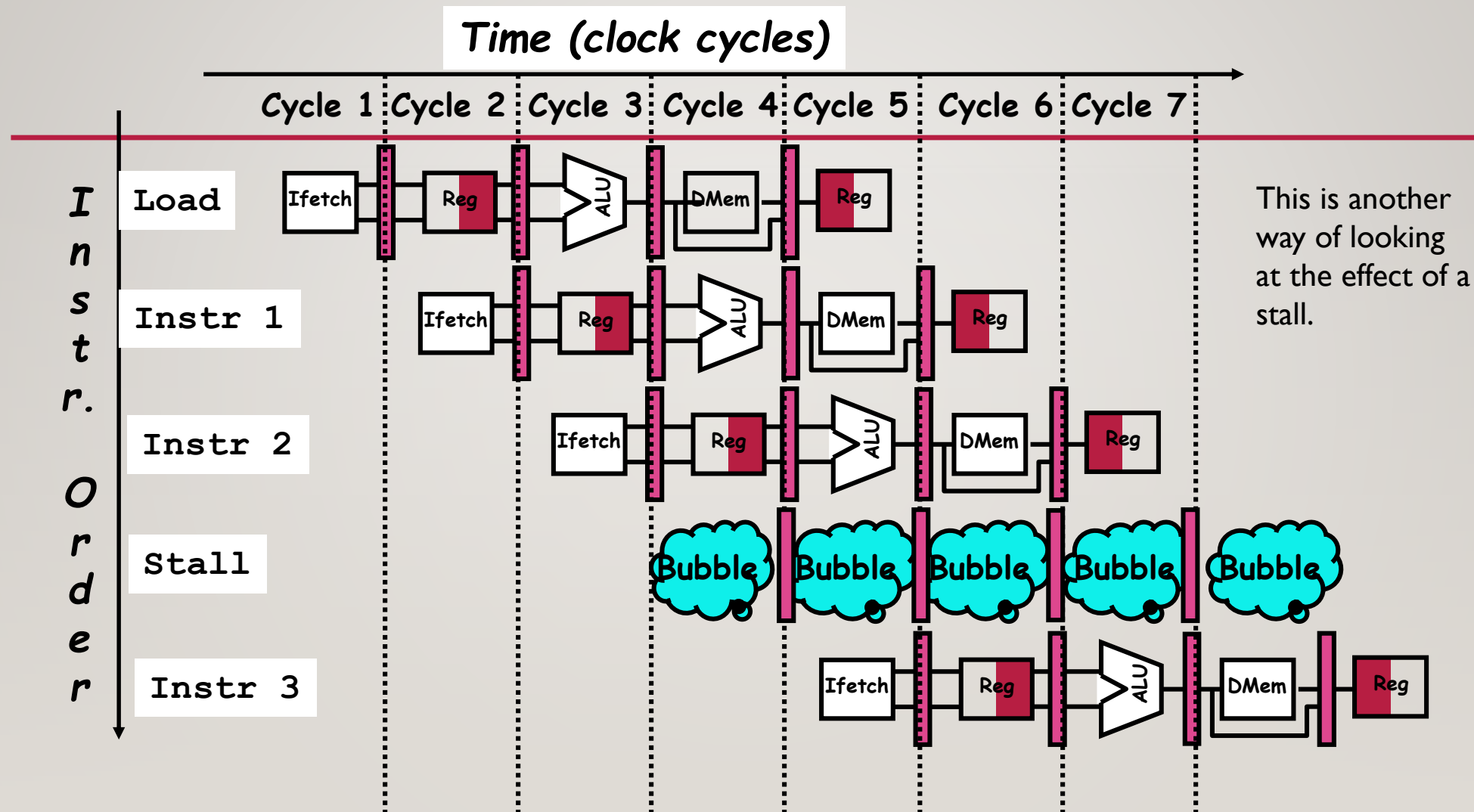
- **Structural hazards:** HW cannot support this combination of instructions (single person to fold and put clothes away).
- **Data hazards:** Instruction depends on result of prior instruction still in the pipeline (missing sock)
- **Control hazards:** Pipelining of branches & other instructions that change the PC
- Common solution is to **stall** the pipeline until the hazard is resolved, inserting one or more “**bubbles**” in the pipeline



STRUCTURAL HAZARDS:



Caches: small and fast “buffer” memories

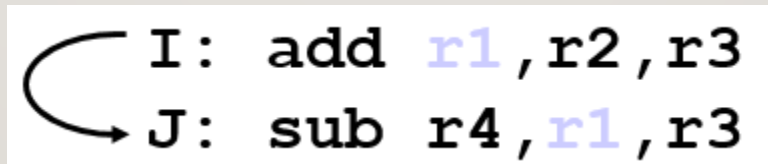


STALLS IN PIPELINE

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

DATA HAZARDS

- A data hazard occurs when the pipeline execution must be stalled because one step must wait for another one to complete. This comes up when a planned instruction can not execute in the planned clock cycle because the data needed is not yet available.
- **Read After Write (RAW)** Instr_j tries to read operand before Instr_i writes it.

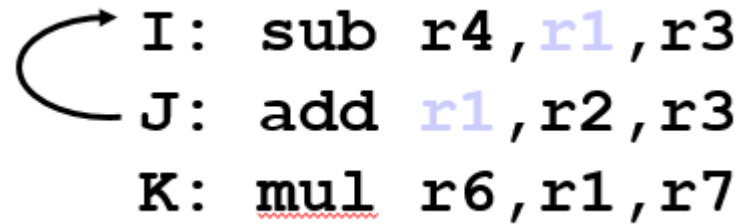


```
I: add r1, r2, r3
J: sub r4, r1, r3
```

The diagram illustrates a Read After Write (RAW) hazard. It shows two instructions, I and J, with a curved arrow pointing from instruction I to instruction J, indicating a data dependency. Instruction I is `I: add r1, r2, r3` and instruction J is `J: sub r4, r1, r3`. The register `r1` is highlighted in blue in both instructions, showing that instruction J is trying to read the value of `r1` before instruction I has finished writing to it.

DATA HAZARDS

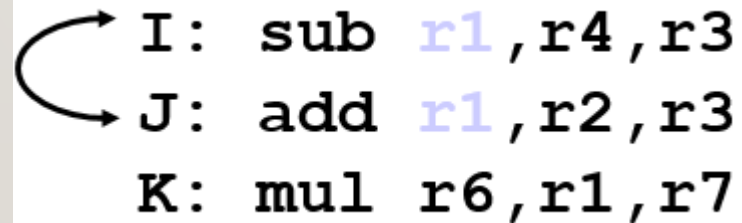
- **Write After Read (WAR)** Instr_j tries to write operand before Instr_i reads it



```
I:  sub  r4, r1, r3
J:  add  r1, r2, r3
K:  mul r6, r1, r7
```

The diagram illustrates a Write After Read (WAR) hazard. Instruction I reads register r1. Instruction K writes to register r1 before instruction J reads it. A curved arrow points from the 'r1' in instruction I to the 'r1' in instruction K, indicating the write operation occurring before the read operation.

- **Write After Write (WAW)** Instr_j tries to write operand before Instr_i writes it

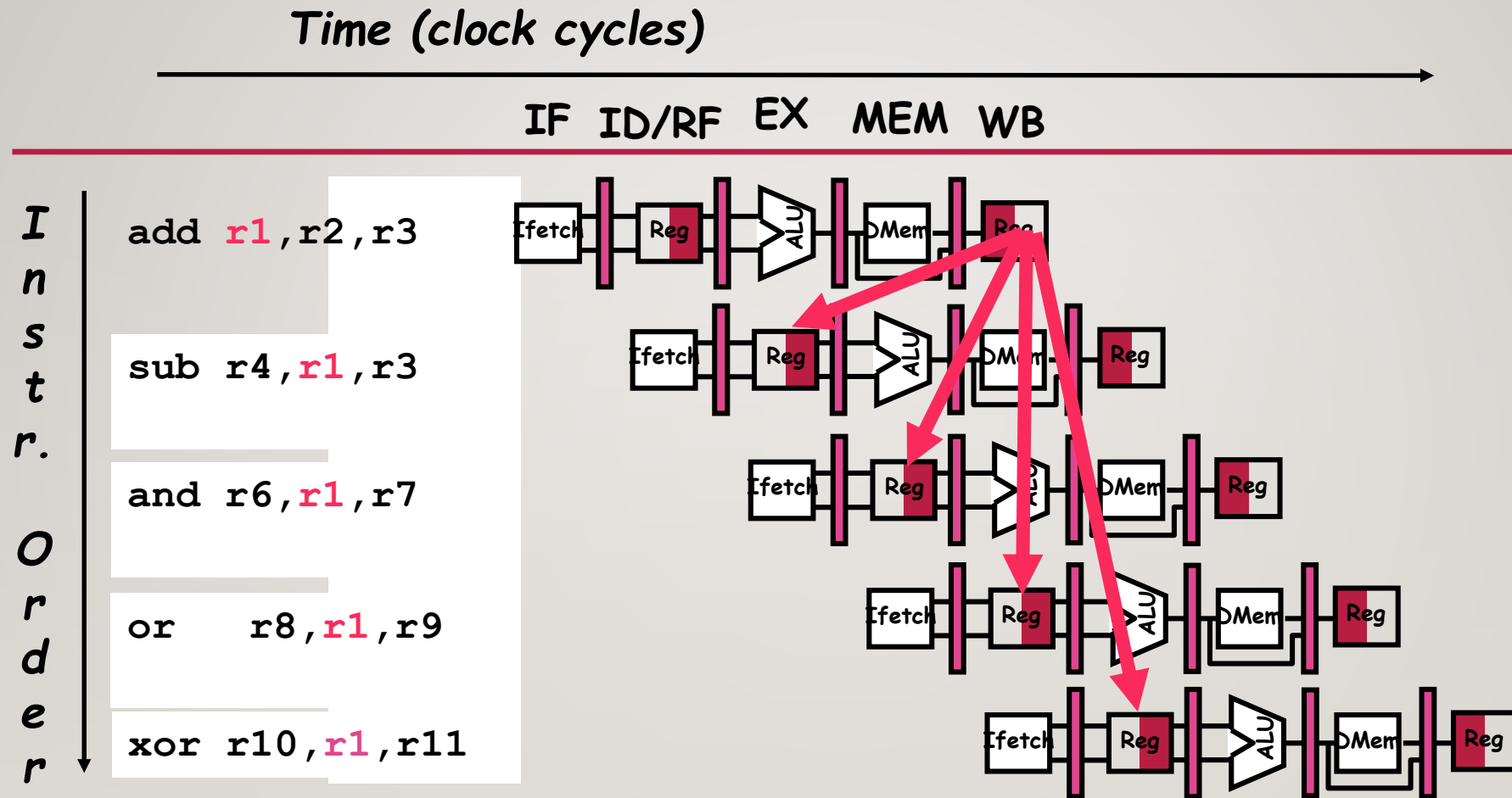


```
I:  sub  r1, r4, r3
J:  add  r1, r2, r3
K:  mul  r6, r1, r7
```

The diagram illustrates a Write After Write (WAW) hazard. Instruction I writes to register r1. Instruction J writes to register r1 before instruction K reads it. A curved arrow points from the 'r1' in instruction I to the 'r1' in instruction J, indicating the write operation occurring before the read operation.

EXAMPLE

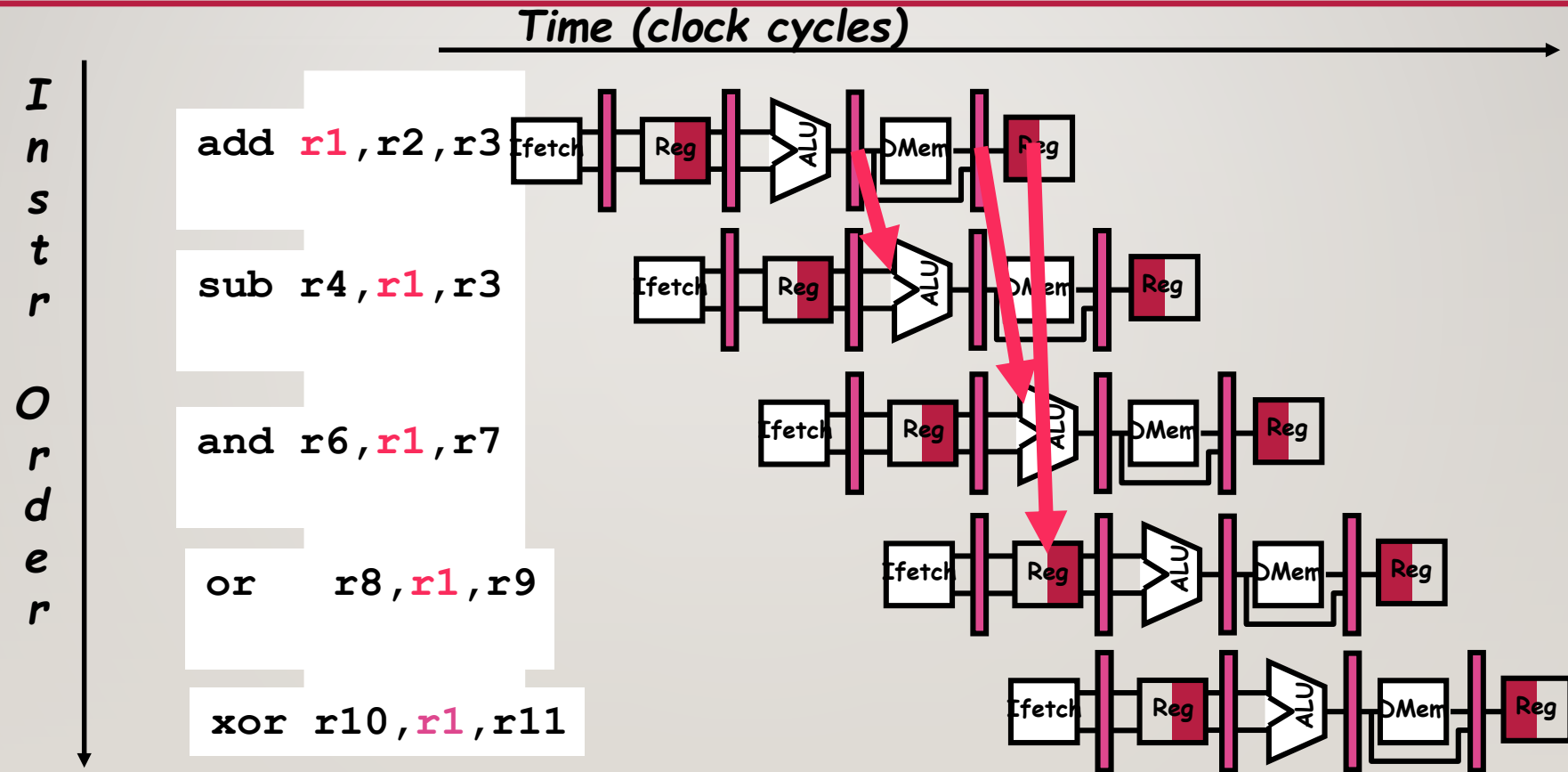
- Consider the pipelined execution of these instructions:
- add x1,x2,x3
- sub x4,x1,x5
- and x6,x1,x7
- or x8,x1,x9
- xor x10,x1,x11



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

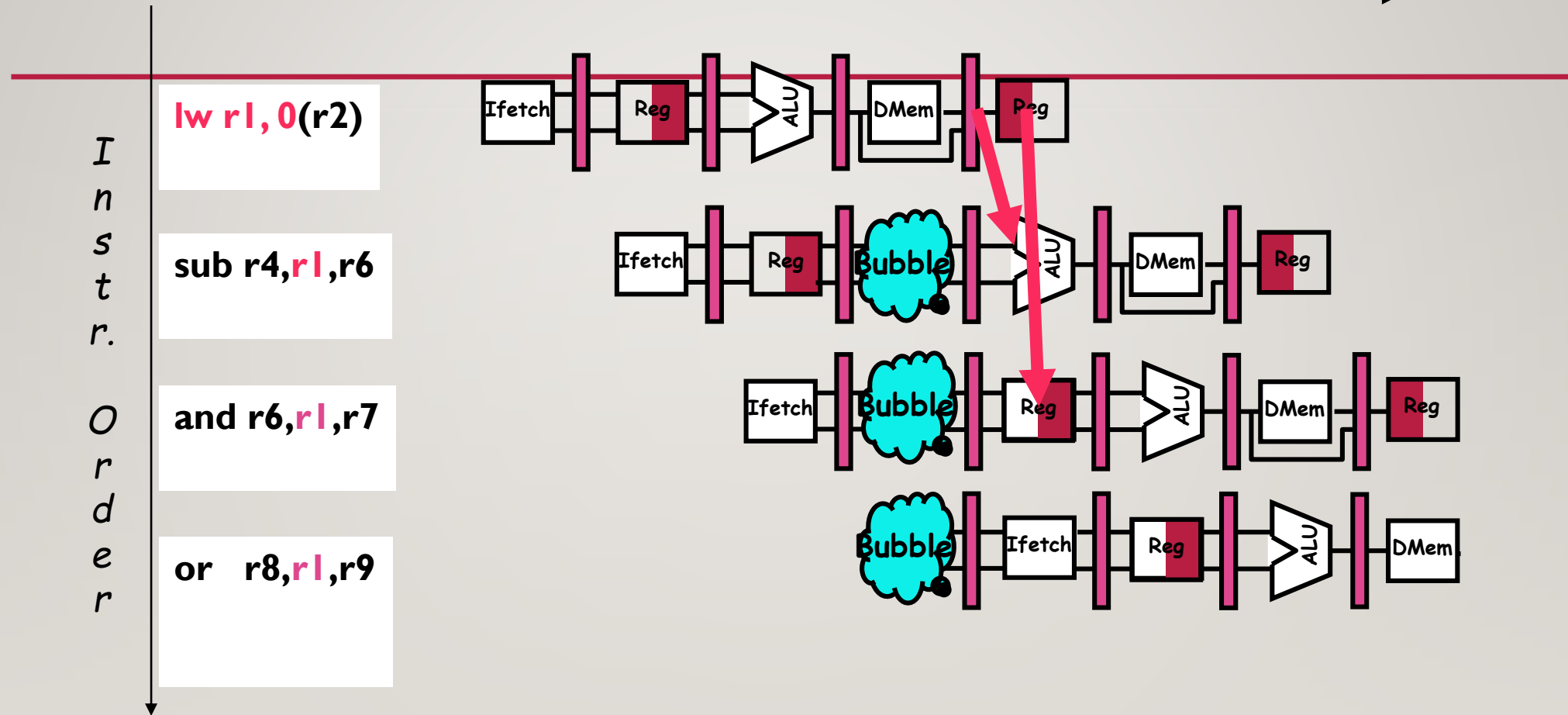
Forwarding To Avoid Data Hazard

Forwarding is the concept of making data available to the input of the ALU for subsequent instructions, even though the generating instruction hasn't gotten to WB in order to write the memory or registers.



Time (clock cycles)

The stall is necessary as shown here.



There are some instances where hazards occur, even with forwarding.

This is another representation of the stall.

LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB 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

LW R1, 0(R2)	IF	ID	EX	MEM	WB				
SUB 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

Pipeline Scheduling

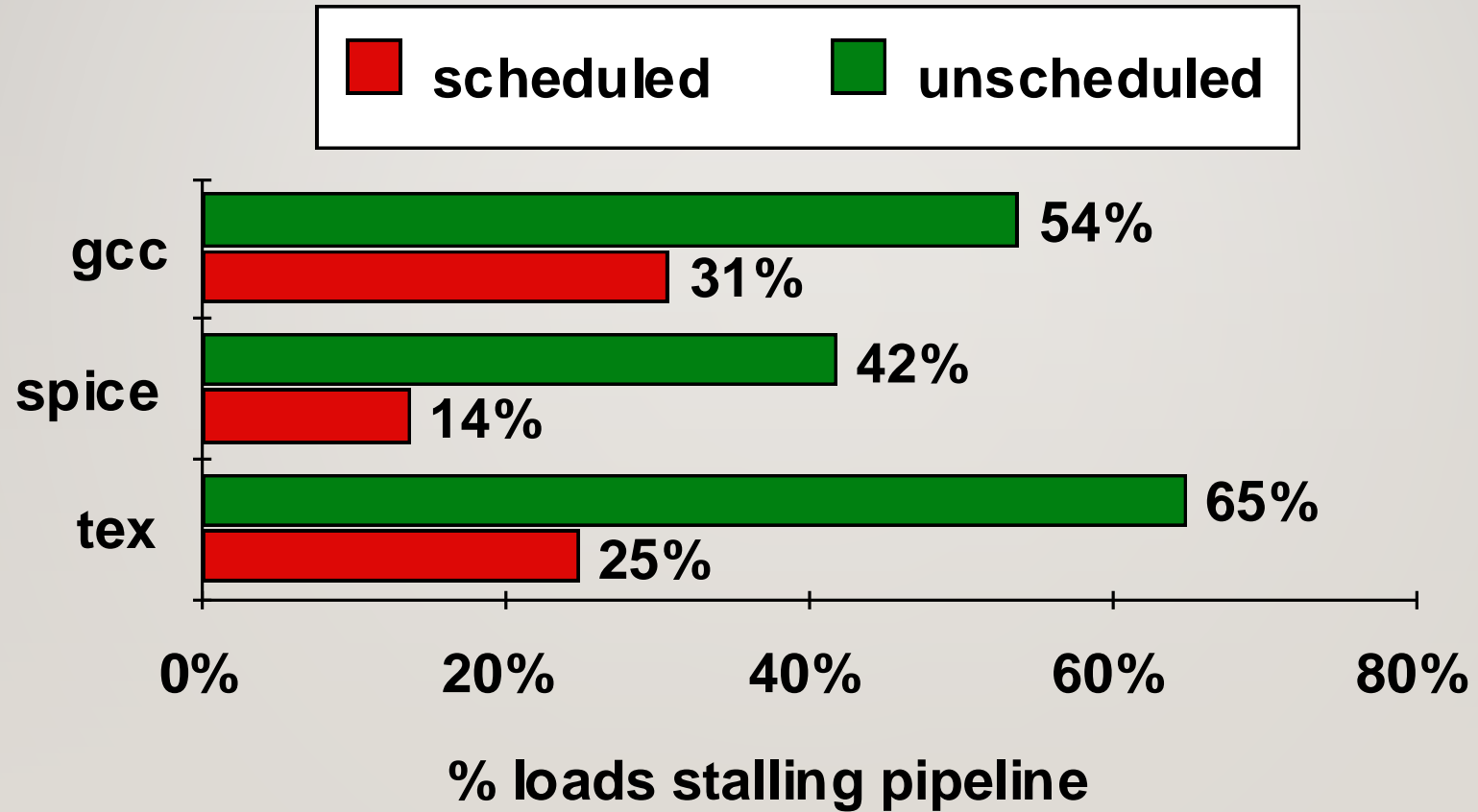
Instruction scheduled by compiler - move instruction in order to reduce stall.

lw Rb, b	code sequence for a = b+c before scheduling
lw Rc, c	
Add Ra, Rb, Rc	stall
sw a, Ra	
lw Re, e	code sequence for d = e+f before scheduling
lw Rf, f	
sub Rd, Re, Rf	stall
sw d, Rd	

Arrangement of code after scheduling.

```
lw Rb, b
lw Rc, c
lw Re, e
Add Ra, Rb, Rc
lw Rf, f
sw a, Ra
sub Rd, Re, Rf
sw d, Rd
```

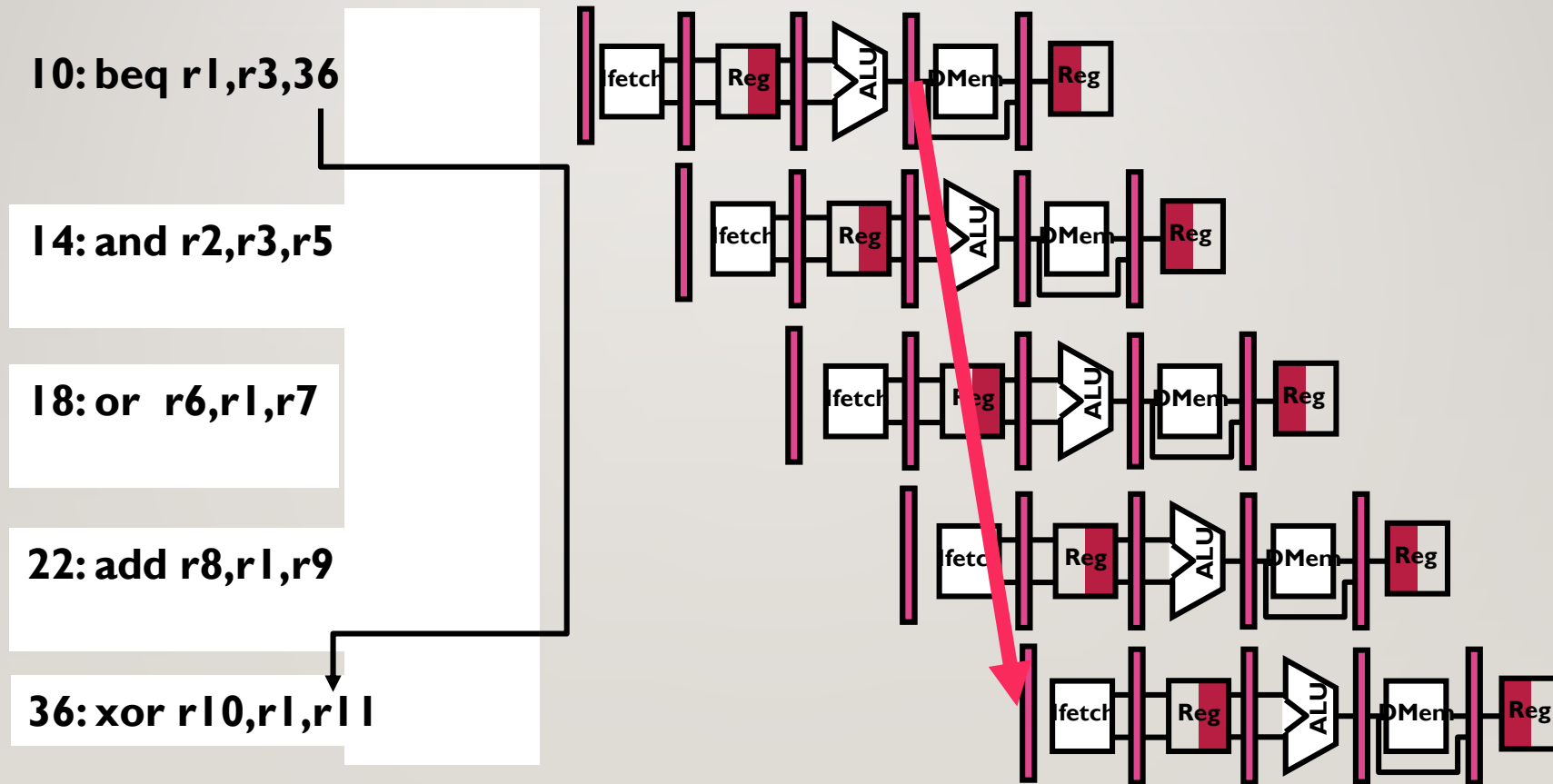
Pipeline Scheduling



CONTROL HAZARDS

- A control hazard is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.
- Control hazards can cause a greater performance loss for our RISC V pipeline than do data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4. Recall that if a branch changes the PC to its target address, it is a taken branch; if it falls through, it is not taken, or untaken. If instruction i is a taken branch, then the PC is usually not changed until the end of ID, after the completion of the address calculation and comparison.

CONTROL HAZARD ON BRANCHES THREE STAGE STALL



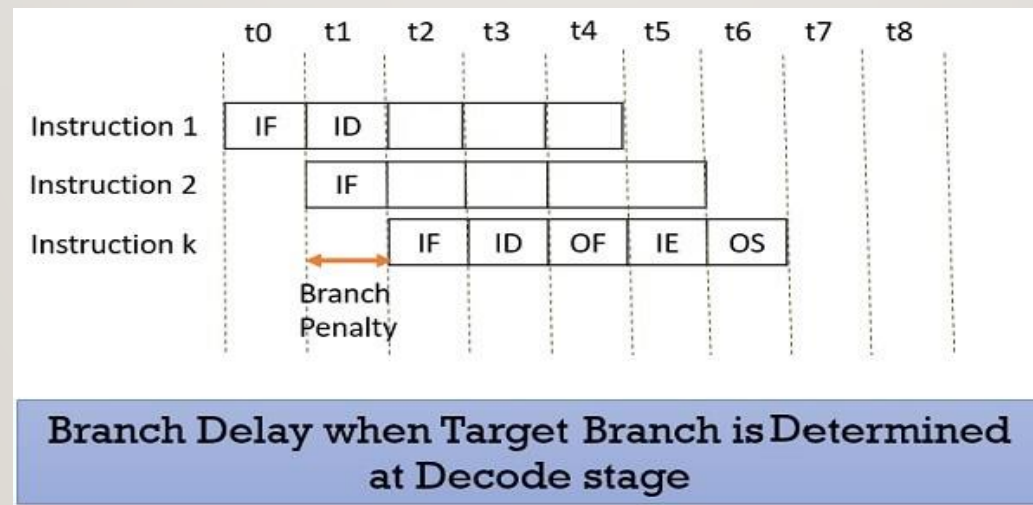
REDUCING PIPELINE BRANCH PENALTIES

- *1. Freeze or flush the pipeline: the simplest scheme*
- Hold or delete any instructions after the branch until the branch destination is known .
- This is the solution shown in Figure
- The branch penalty is fixed and cannot be reduced by software

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

- 2. *Treat every branch as not taken.*
- Continue to fetch instructions as if there were no branch.
- Restart fetch at target address (and turn previously fetched instruction into a NOP) if the branch is taken.



REDUCING PIPELINE BRANCH PENALTIES

- *3. Treat every branch as taken*
- As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target..
- This buys us a one-cycle improvement when the branch is actually taken, because we know the target address at the end of ID, one cycle before we know whether the branch condition is satisfied in the ALU stage. In either a predicted-taken or predicted-not-taken scheme, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware's choice.

REDUCING PIPELINE BRANCH PENALTIES

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

- A fourth scheme, which was heavily used in early RISC processors is called **delayed branch**. In a delayed branch, the execution cycle with a branch delay of one is:

```
branch instruction  
sequential successor1  
branch target if taken
```

- Although it is possible to have a branch delay longer than one, in practice almost all processors with delayed branch have a single instruction delay; other techniques are used if the pipeline has a longer potential branch penalty. The job of the compiler is to make the successor instructions valid and useful.

REDUCING PIPELINE BRANCH PENALTIES

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

