



**University of Engineering and Technology (UET), Peshawar,
Pakistan**

Lecture 10

CSE-304: Computer Organization and Architecture

BY:

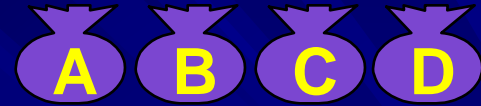
Dr. Muhammad Athar Javed Sethi

What is pipelining?

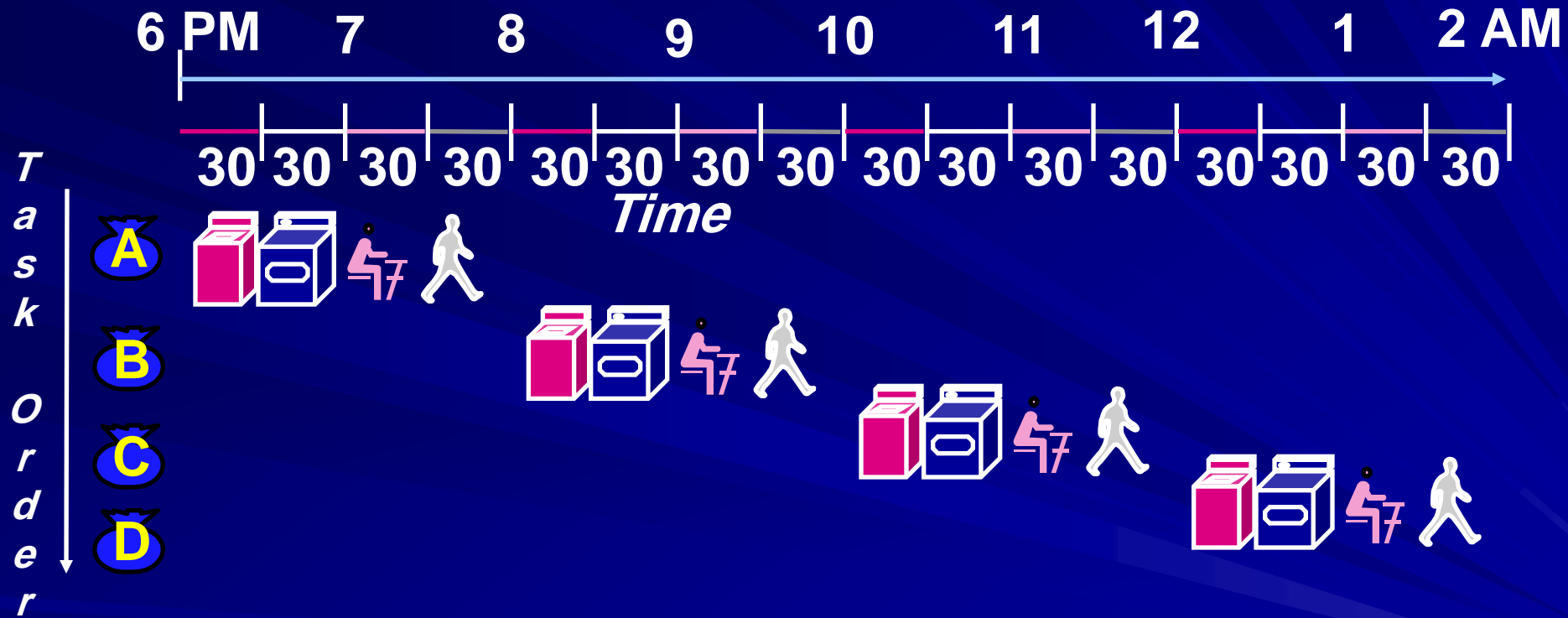
- **Pipelining is an implementation technique whereby multiple instructions are overlapped in execution.**
- **It takes advantage of parallelism that exists among the actions needed to execute an instruction.**
- **Pipelining is the key implementation technique used to make fast CPUs.**
- **Example: Automobile assembly line**

Pipelining is Natural!

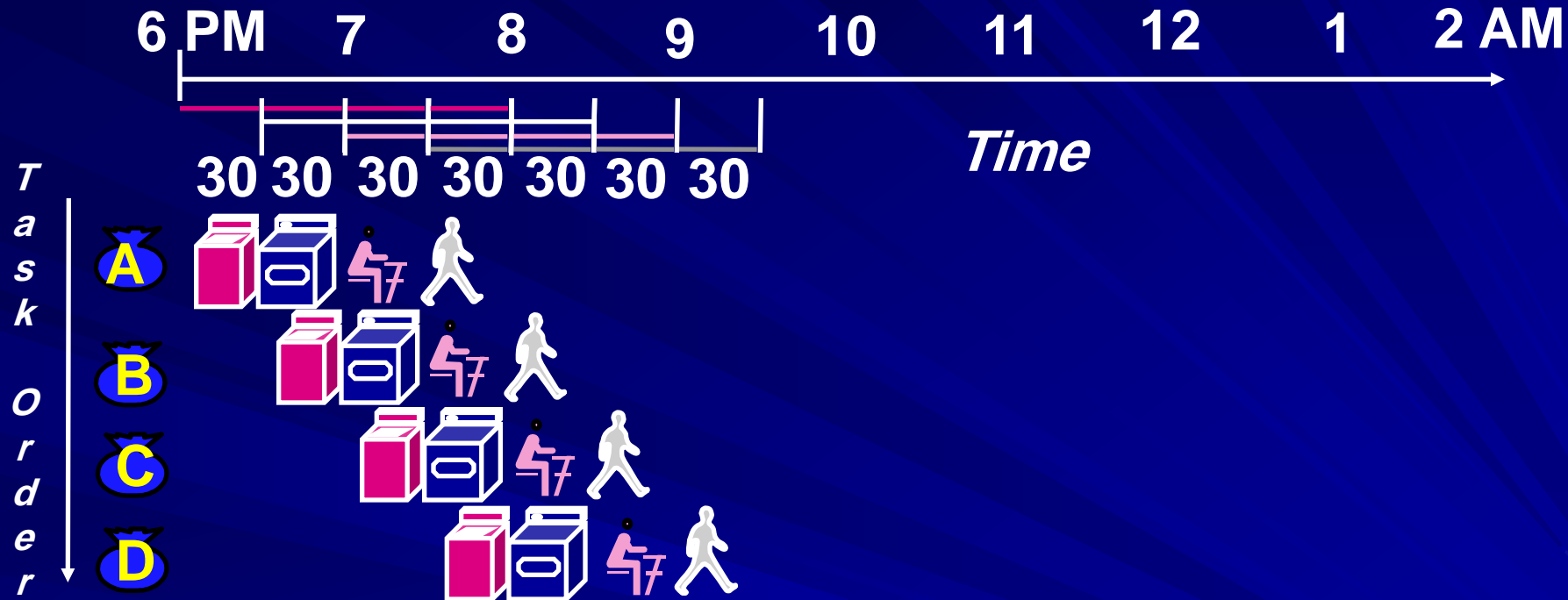
- Laundry Example!
- Four loads: A, B, C, D
- Four laundry operations:
Wash, Dry, fold and place into
drawers
- Washer takes 30 minutes
- Dryer takes 30 minutes
- “Folder” takes 30 minutes
- “Stasher” takes 30 minutes
to put clothes into drawers



Sequential Laundry



Pipelined Laundry: Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads!

Features of Pipelined Processor

- **All the functional units operate independently**
- **Multiple tasks operating simultaneously using different resources**
- **Pipelining doesn't help latency of single task, it helps throughput of entire workload**

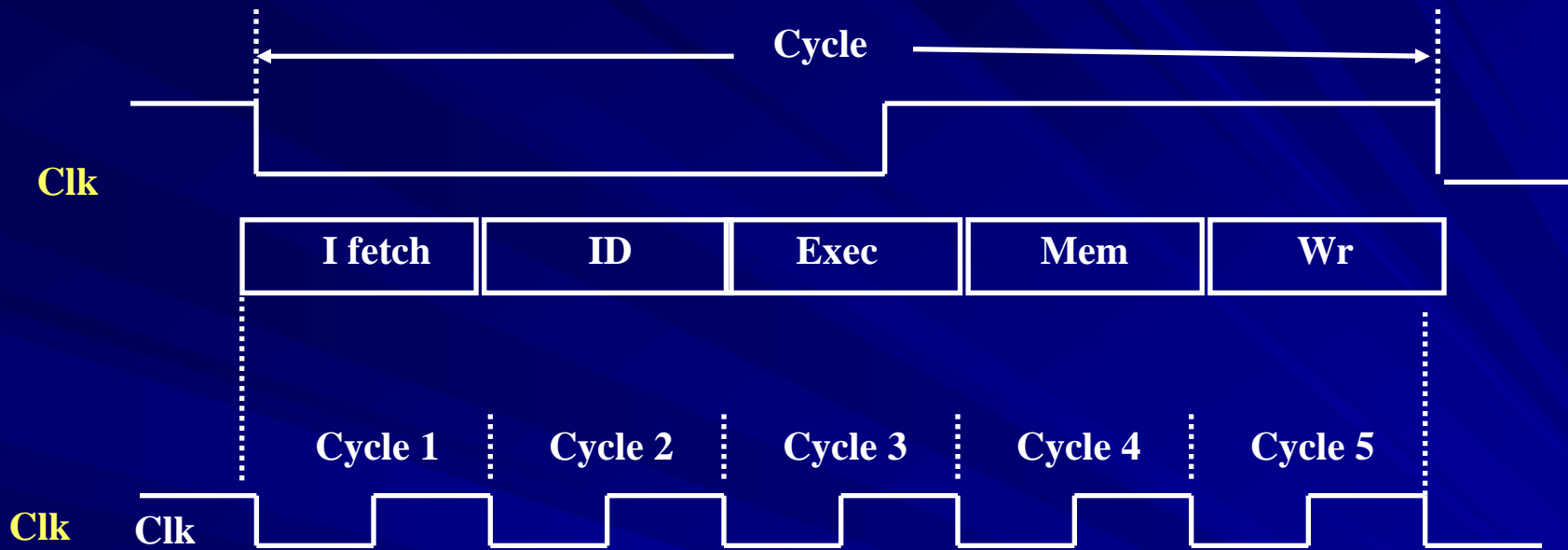
Pipelining Lessons

- Pipeline rate limited by:
 - Slowest pipeline stage
 - Time to **fill** pipeline and time to **drain** it reduces speedup
- Unbalanced lengths of pipe stages reduces speedup
- If washer takes longer time than the dryer then dryer has to wait!
- Stall for Dependences

Five Steps of Data path

- **Instruction fetch (IF)**
- **Instruction Decode (ID)**
- **Execution (Ex)**
- **Memory Access (MEM)**
- **Write back cycle (WB)**

Multiple Cycle Approach



■ The single cycle operations are performed in five steps:

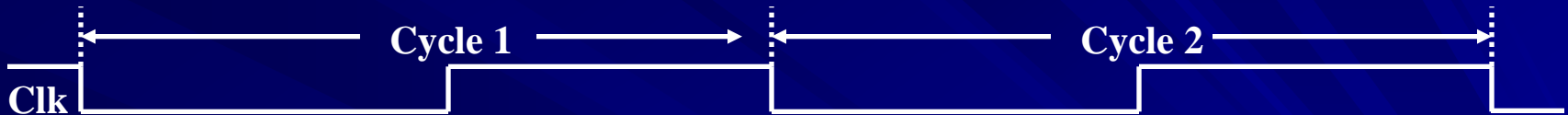
- Instruction Fetch
- Instruction Decode
- Execute
- Memory (Read/write)
- Write (to register file)

Multiple Cycle Approach

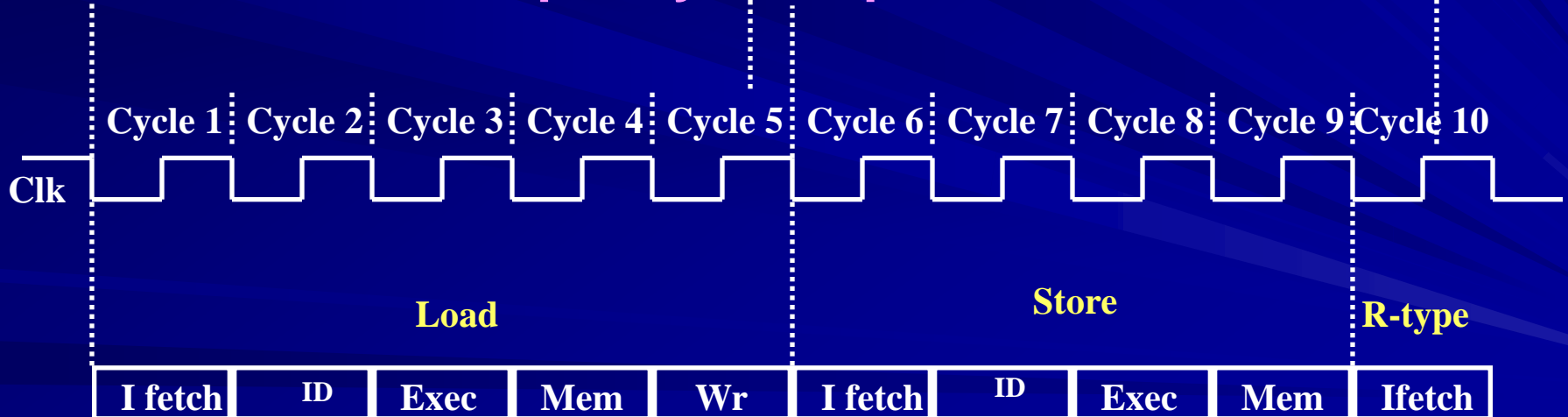
- In the Single Cycle implementation, the cycle time is set to accommodate the longest instruction, the Load instruction.
- In the Multiple Cycles implementation, the cycle time is set to accomplish longest step, the memory read/write
- Consequently, the cycle time for the Single Cycle implementation can be five times longer than the multiple cycle implementation.
- As an example, if $T = 5 \mu \text{ Sec.}$ for single cycle then $T = 1 \mu \text{ Sec.}$ for multi cycle implementation

Single Cycle vs. Multiple Cycle

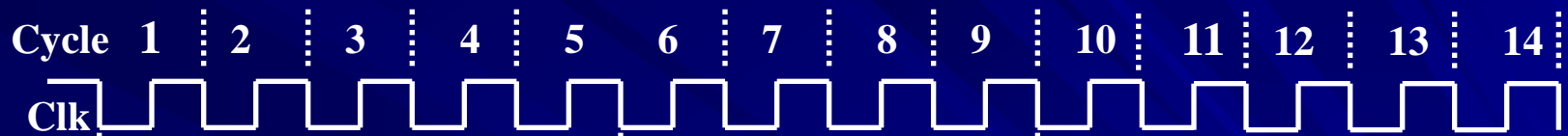
Single Cycle Implementation:



Multiple Cycle Implementation:



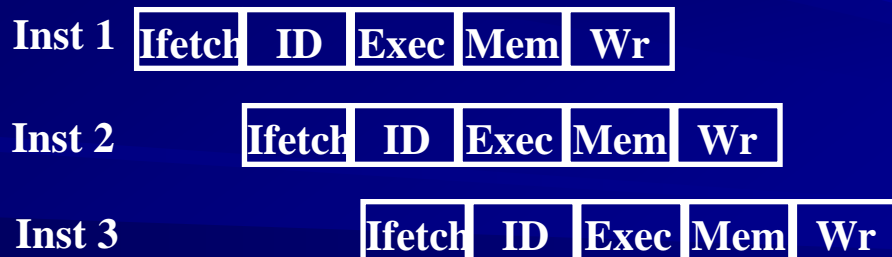
Multiple Cycle verses Pipeline – Pipeline enhances performance



Multiple Cycle Implementation:

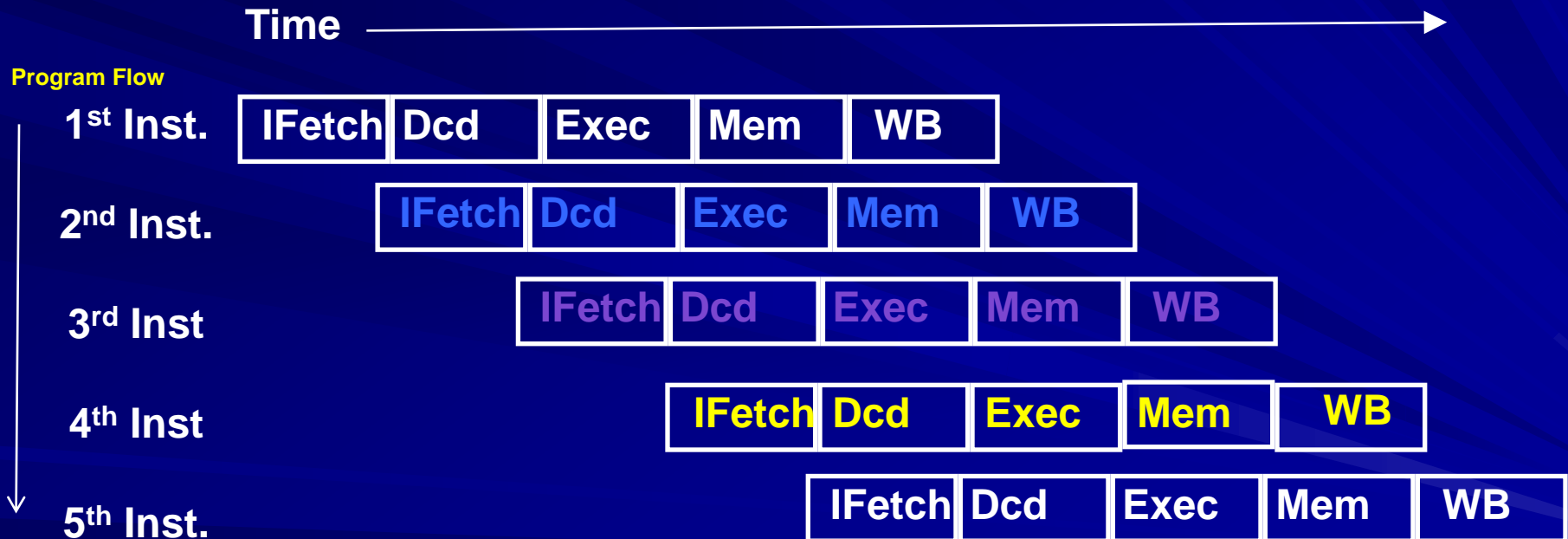


Pipeline Implementation:

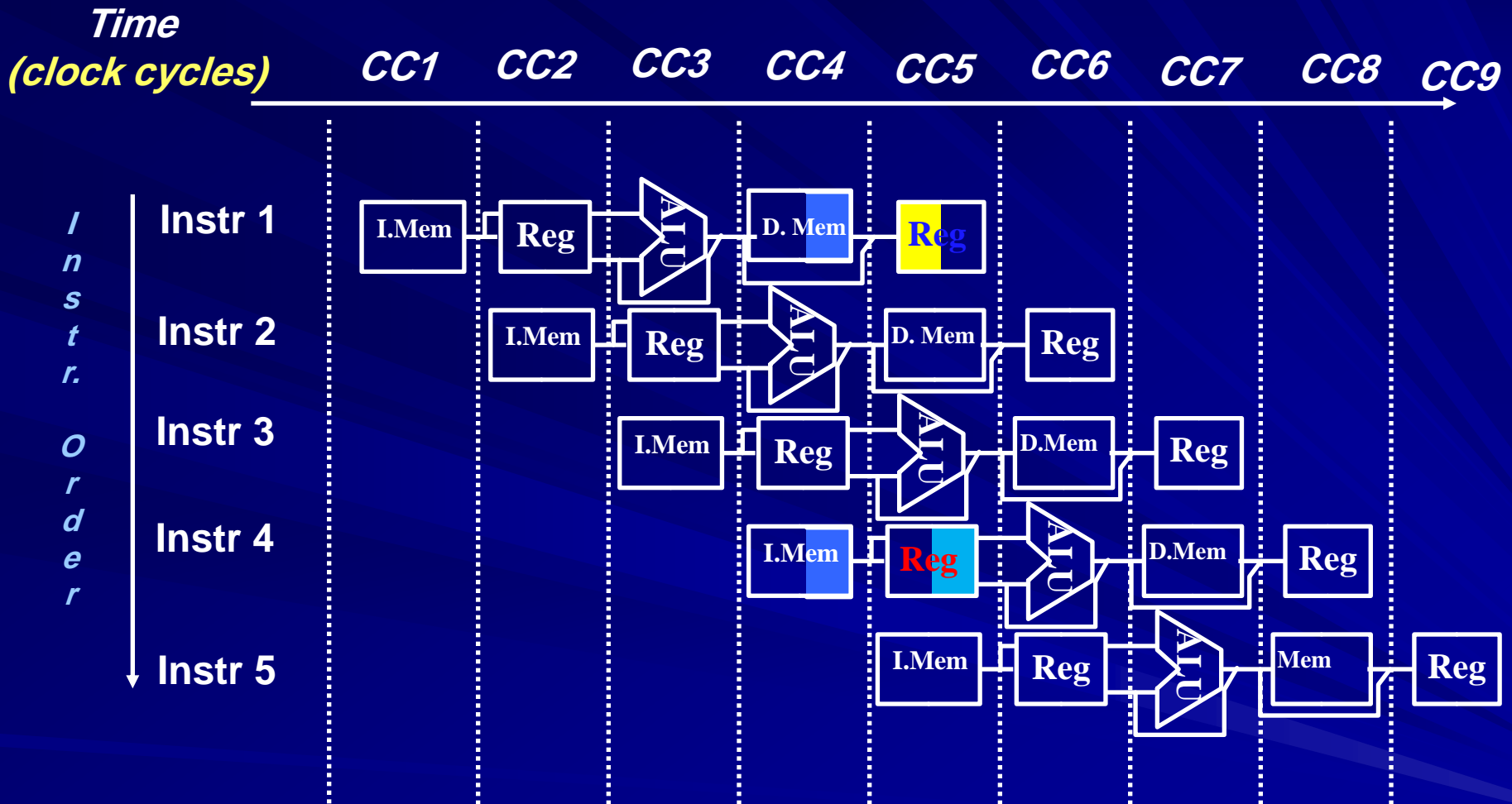


Pipelined Execution Representation

- Conventional Representation

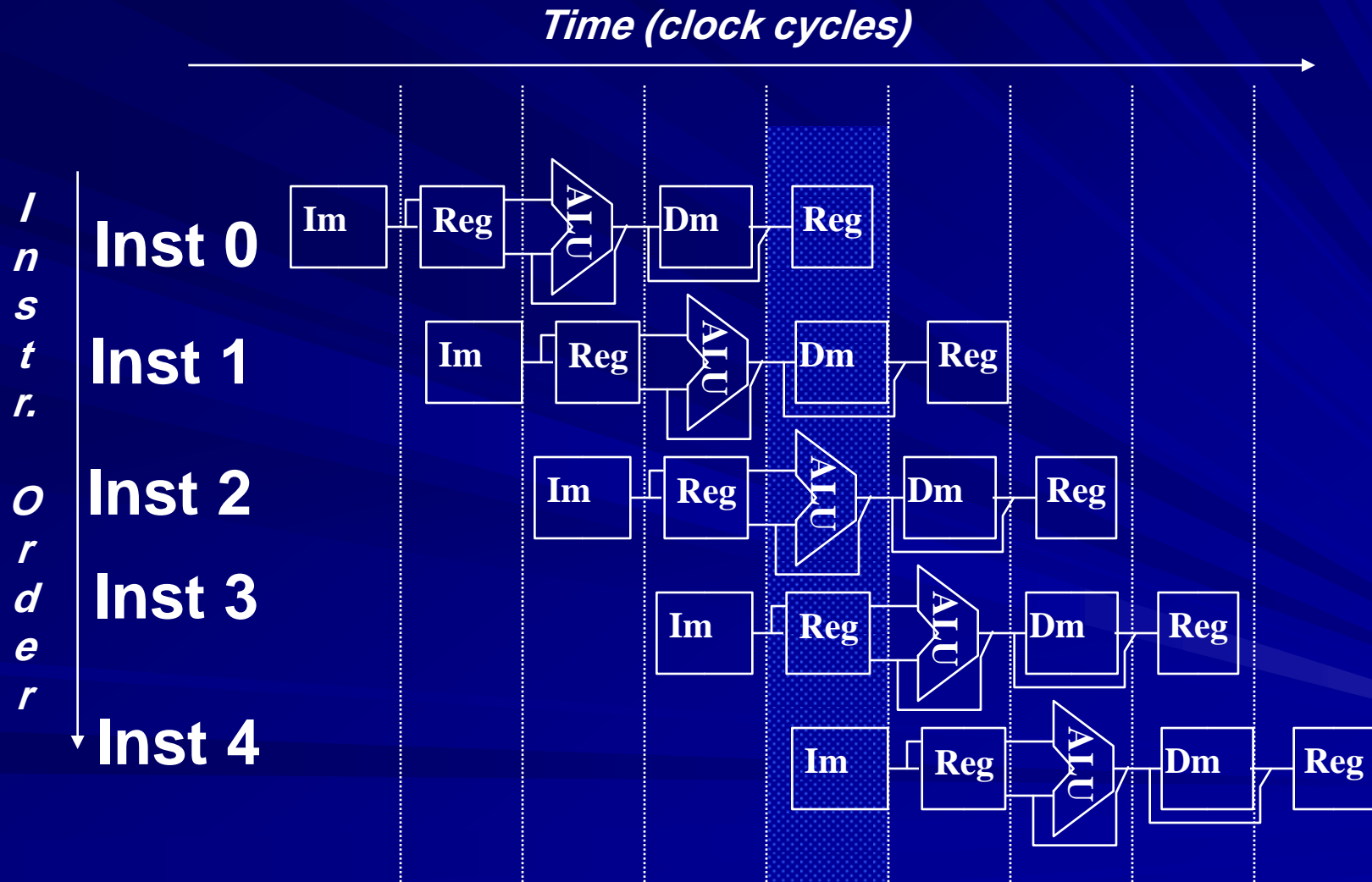


Graphical Representation



Why Pipeline?

Because the resources are there!



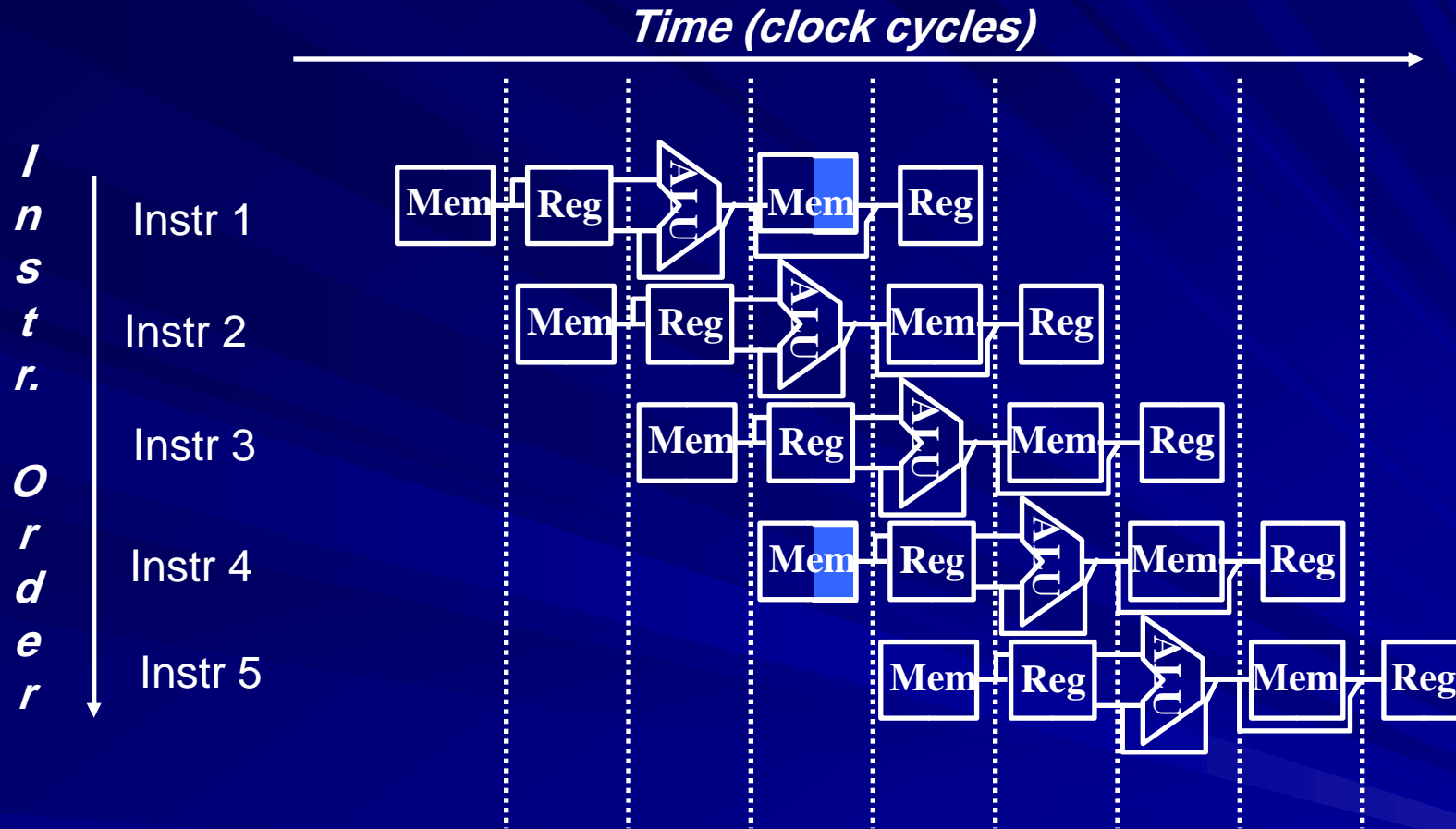
Can pipelining get us into trouble?

- **Structural hazards**
- **Data hazards**
- **Control hazards**

Structural Hazards

- **Attempt to use the same resource two different ways at the same time, e.g.**
 - **Single memory is accessed for instruction fetch and data read in the same clock cycle would be a structural hazard**

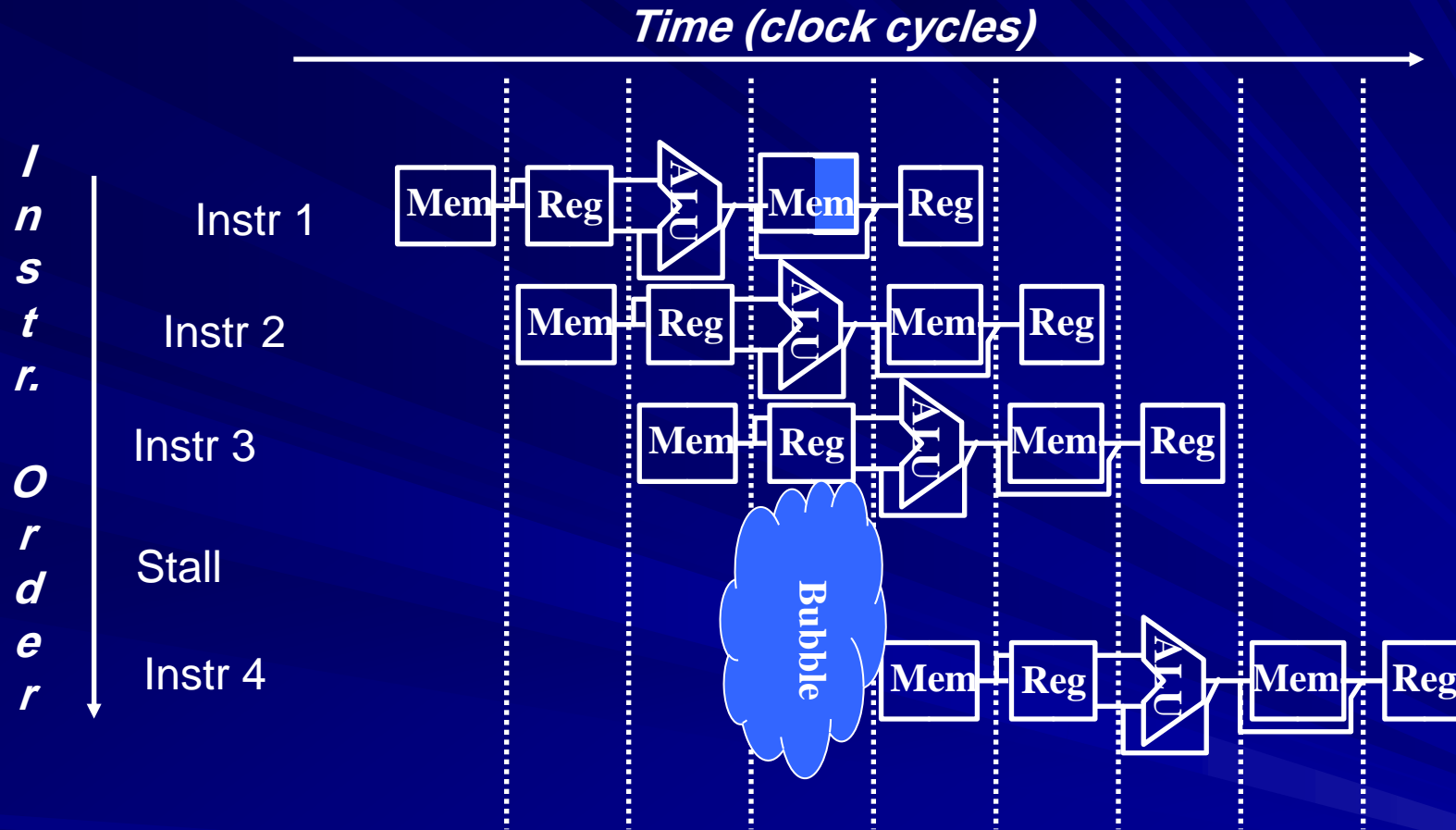
Single Memory is a Structural Hazard



Two memory read operations in the 4th cycle:

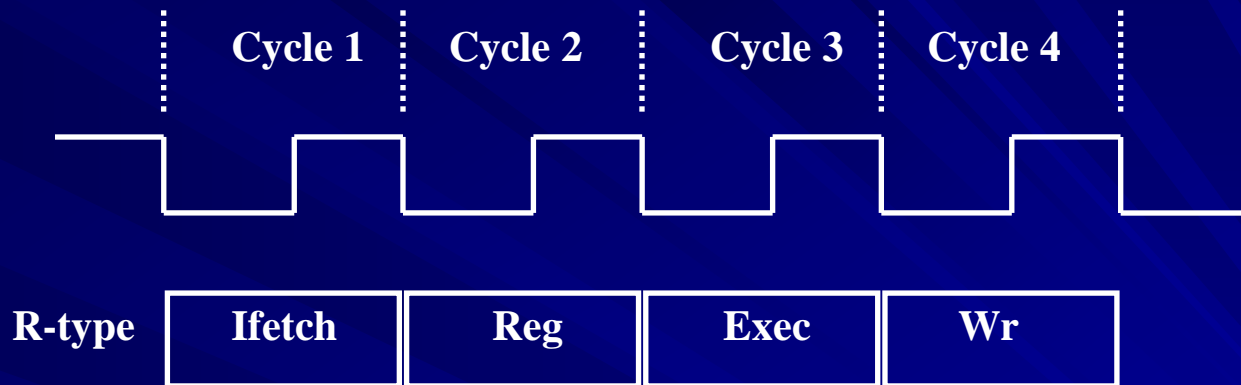
The 1st instruction accesses memory to read data and the 4th instruction fetched from the same memory

Single Memory is a Structural Hazard



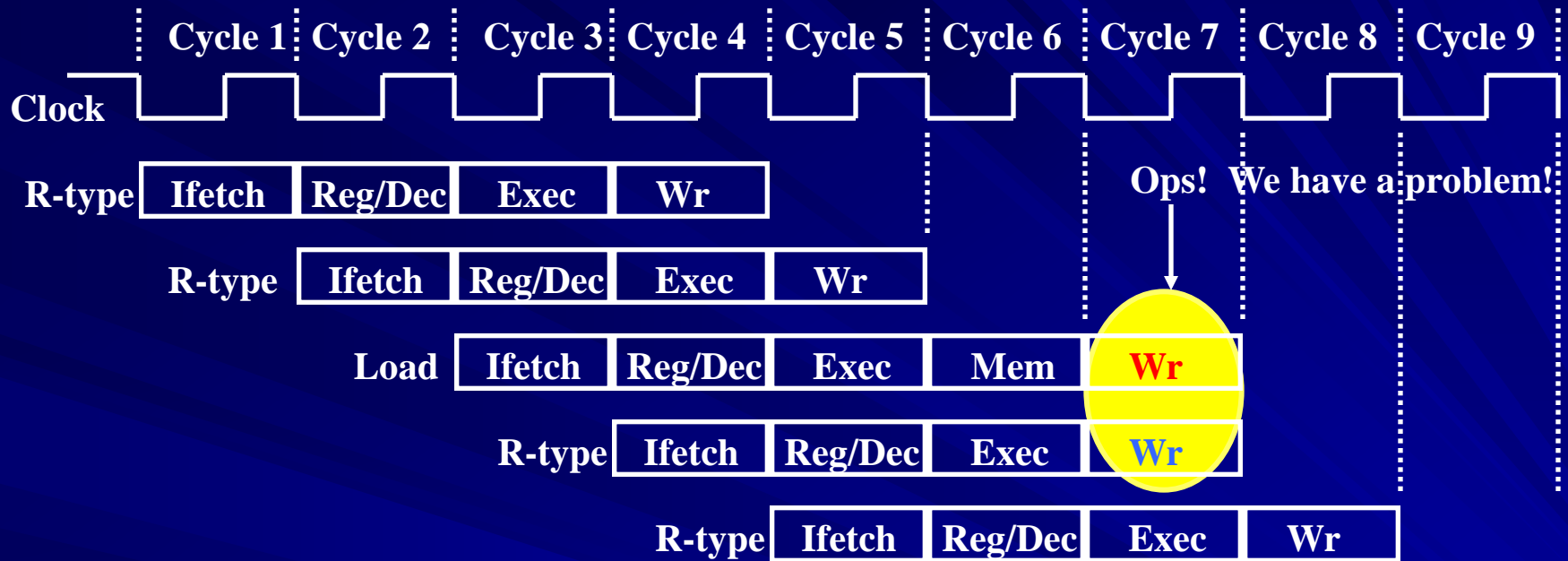
Insert stall (bubble) to avoid memory structural hazard

The Four Stages of R-type



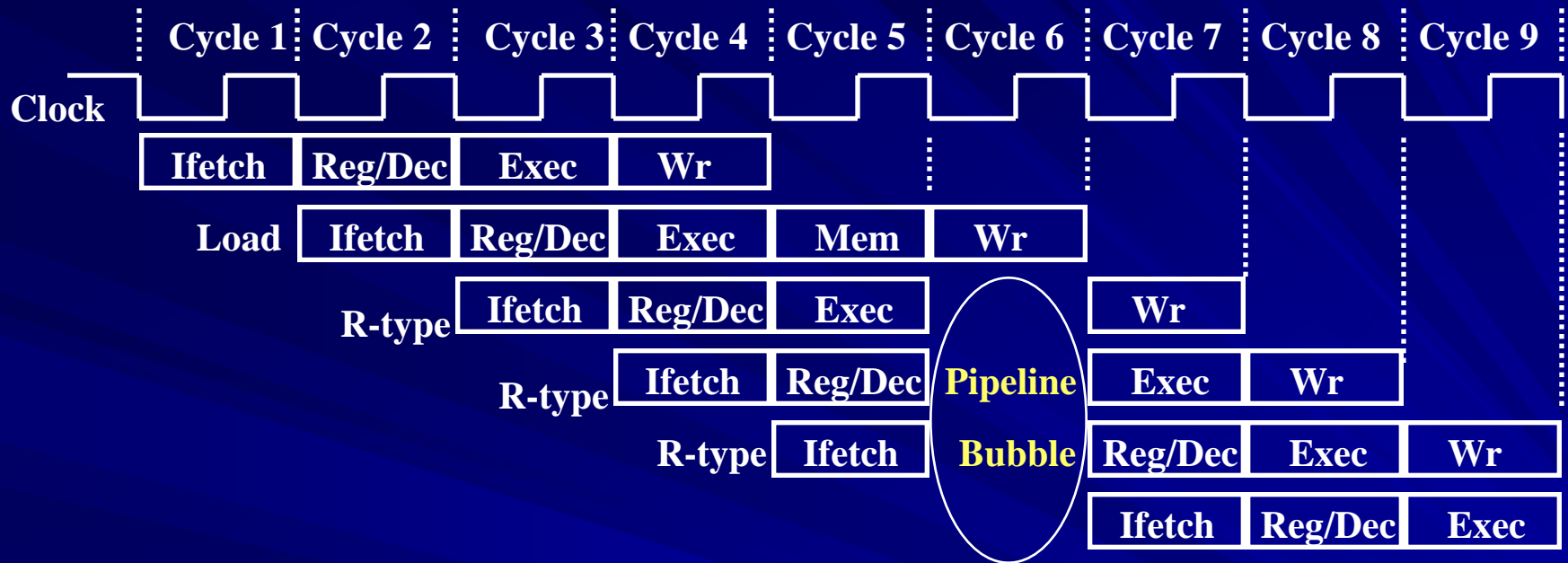
- R-type instruction does not access data memory, so it only takes 4 clocks, or say 4 stages to complete
- Here, the ALU is used to operate on the register operands
- The result is written in to the register during WB stage

Pipelining the R-type and Load Instruction



- We have pipeline conflict or structural hazard:
 - Two instructions try to write to the register file at the same time!
 - Only one write port

Solution 1: Insert “Bubble” into the Pipeline

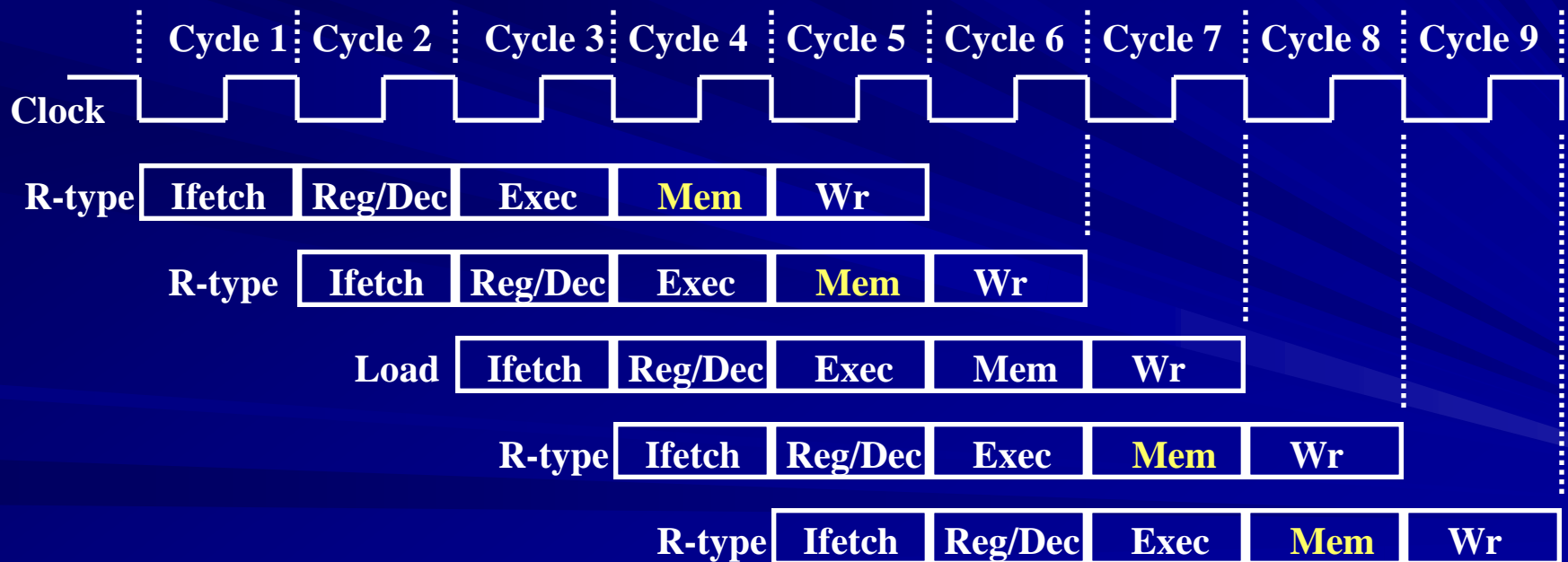


- Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle
 - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

Solution 2: Delay R-type's Write by One Cycle

■ Delay R-type's register write by one cycle:

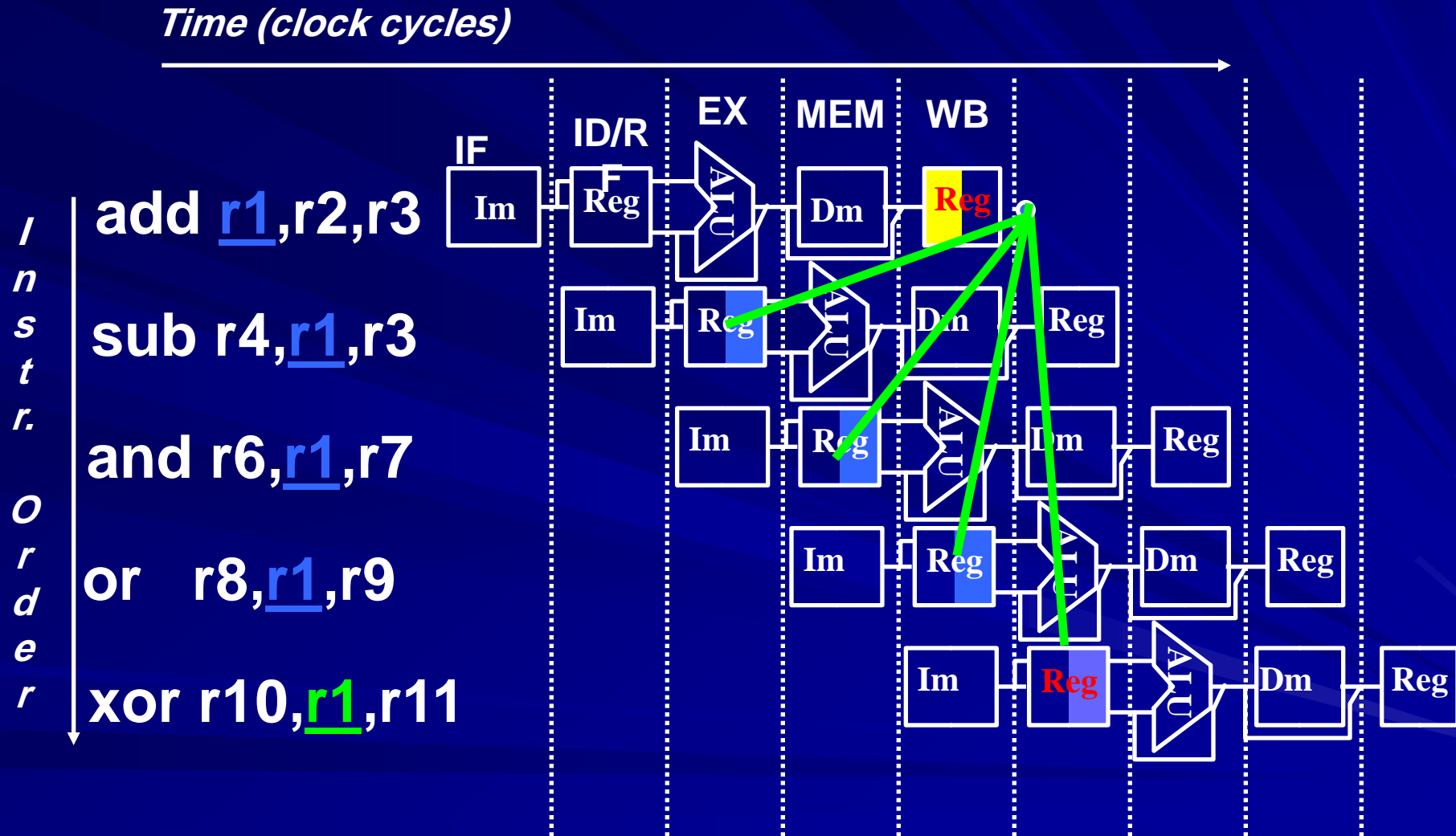
- Now R-type instructions also use Reg at Stage 5
- Mem stage is a **NO-OP** stage: nothing is being done.



DATA Hazards

- **When Data of previous instructions are not available and it is being used by next instructions. This cause data hazard.**

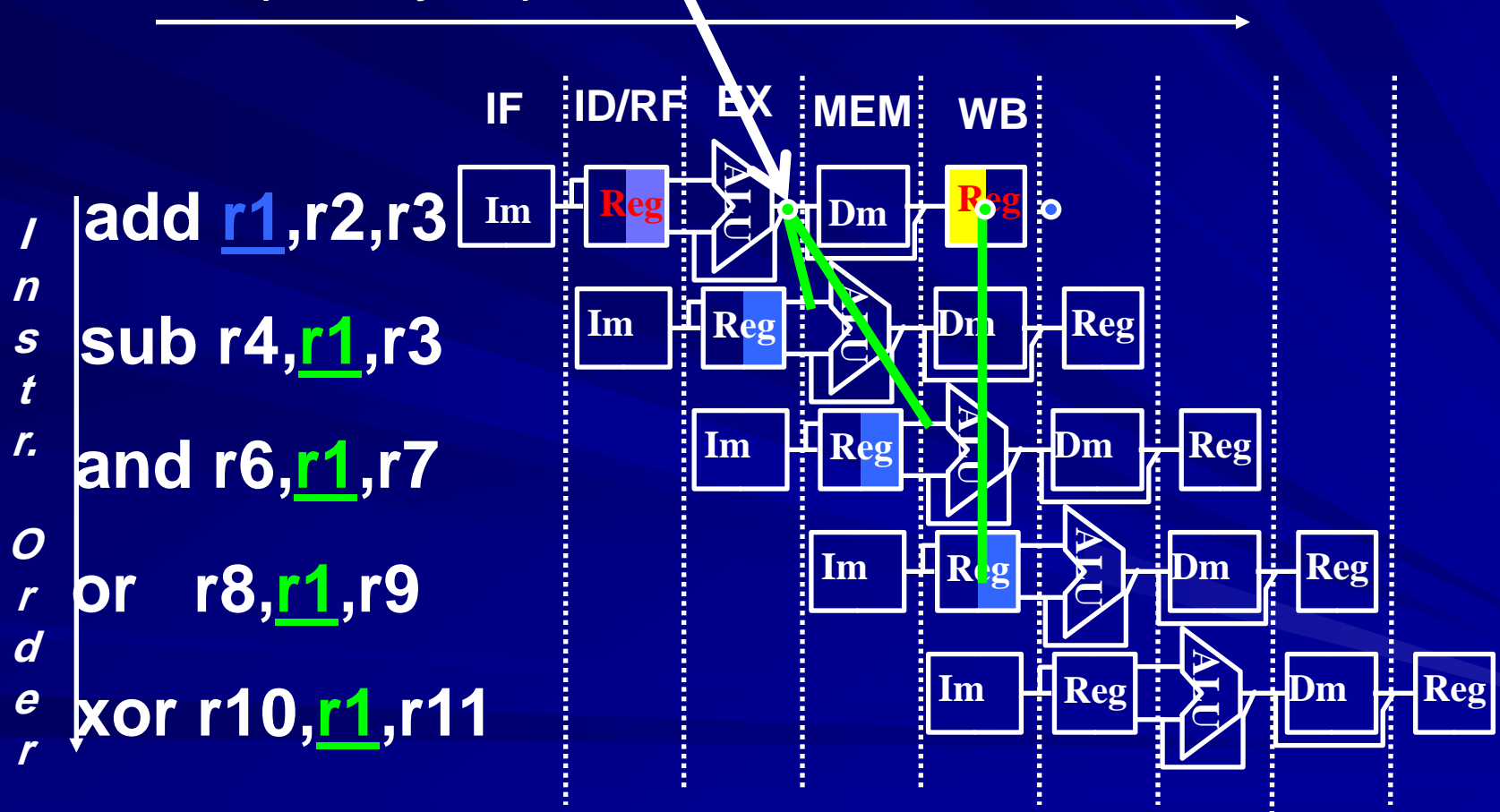
Example: Data Hazard



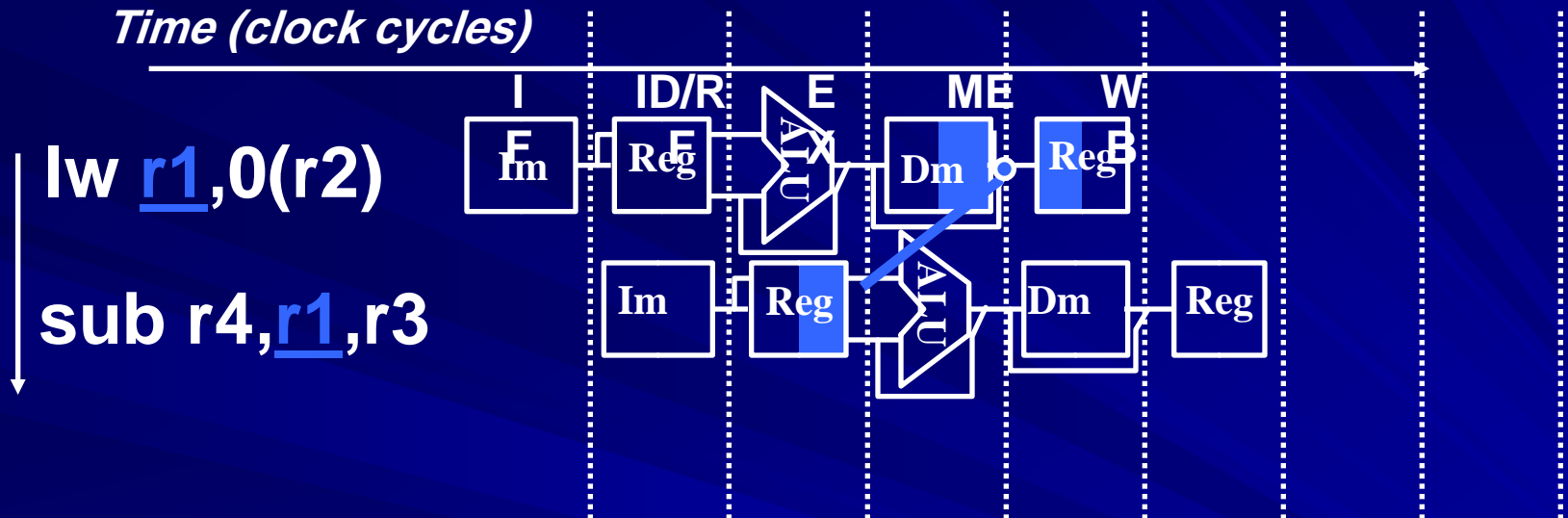
Data Hazard Solution - Forwarding

“Forward” result from one stage to another
From the EX/MEM pipeline register to Sub ALU stage,
MEM/WB pipeline register to AND ALU stage

Time (clock cycles)



Forwarding (or Bypassing): What about Loads?



- Dependencies backwards in time are hazards
- In this case, we can't solve with forwarding:
- Must delay/stall instruction dependent on loads

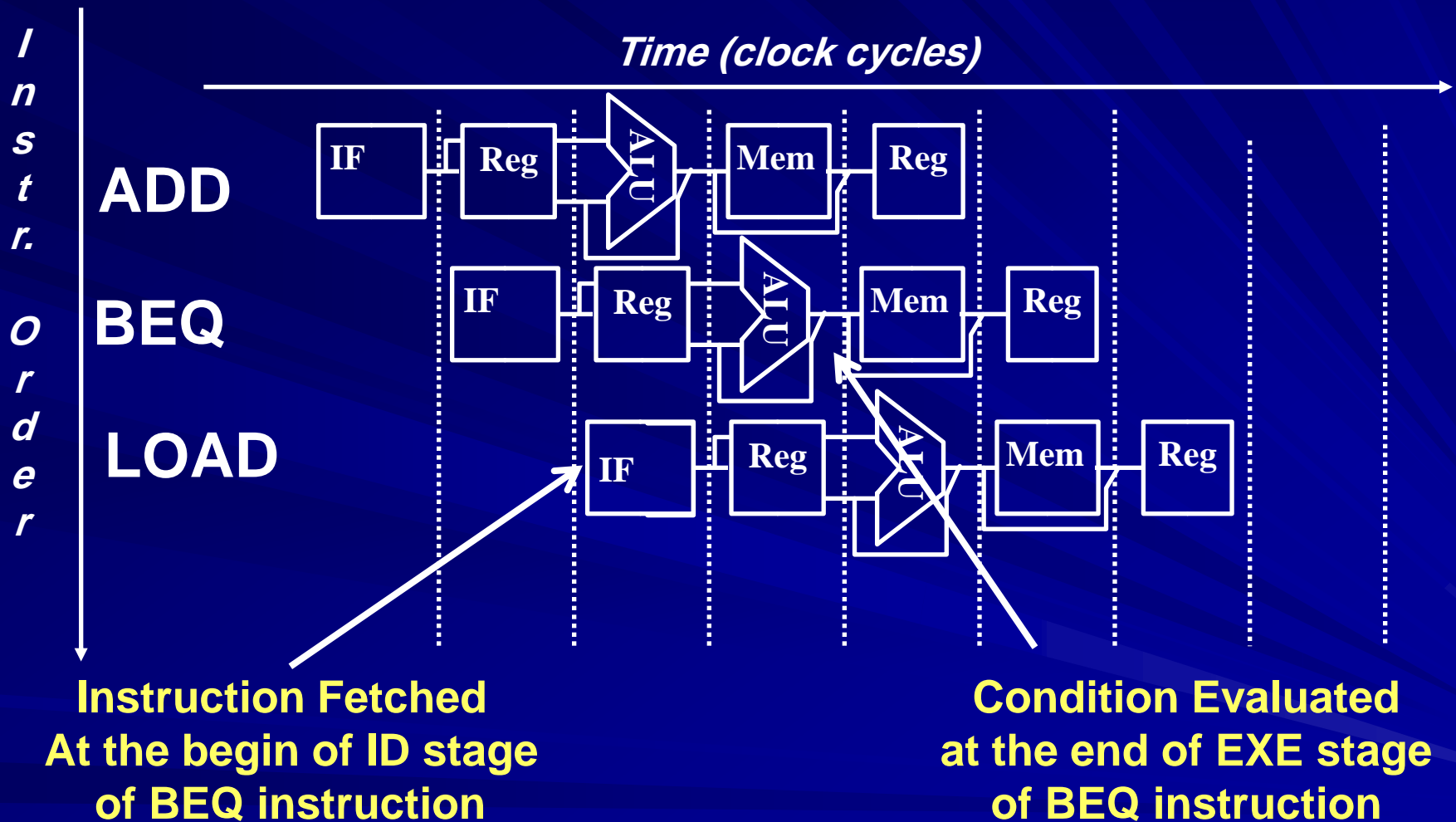
Control Hazards

- **Control hazard occurs when one attempt to take action before condition is evaluated**

Explanation:

- **When Branch instructions is executed it may or may not change the PC to something other than PC+4**
- **Branch Taken: Branch changes the PC+4 to new target**
- **Branch Not Taken: Branch does not change the PC+4**

Control Hazard: Example BEQ Taken



Explanation Branch Hazard

Here, If the BEQ is taken then

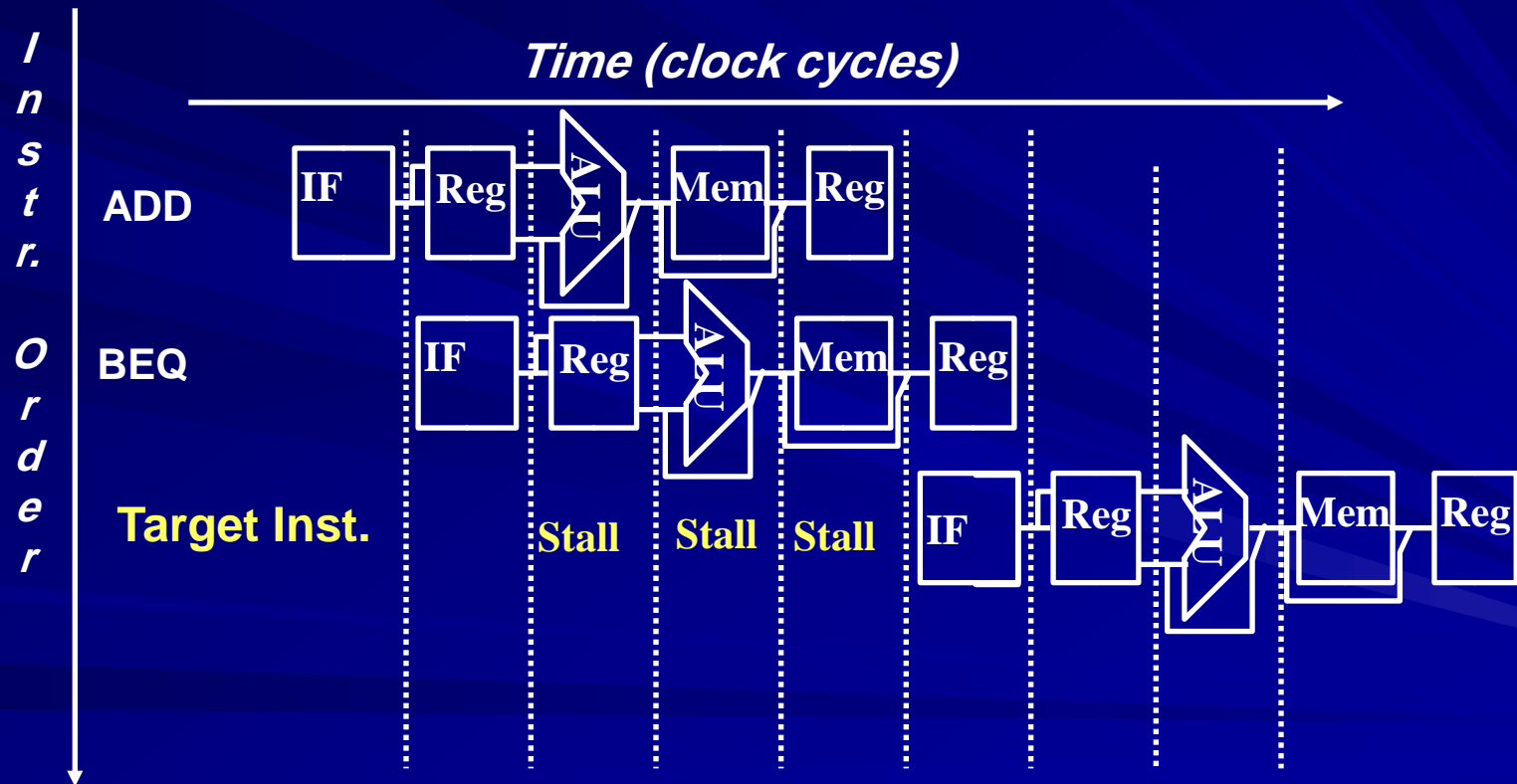
- The next instruction address is determined after evaluating the branch condition in the EX stage;
- But the next instruction (LOAD) is fetched in the ID stage, i.e., before the PC+4 is changed to new target address
- This gives rise to Branch or Control Hazard

Dealing with Branches

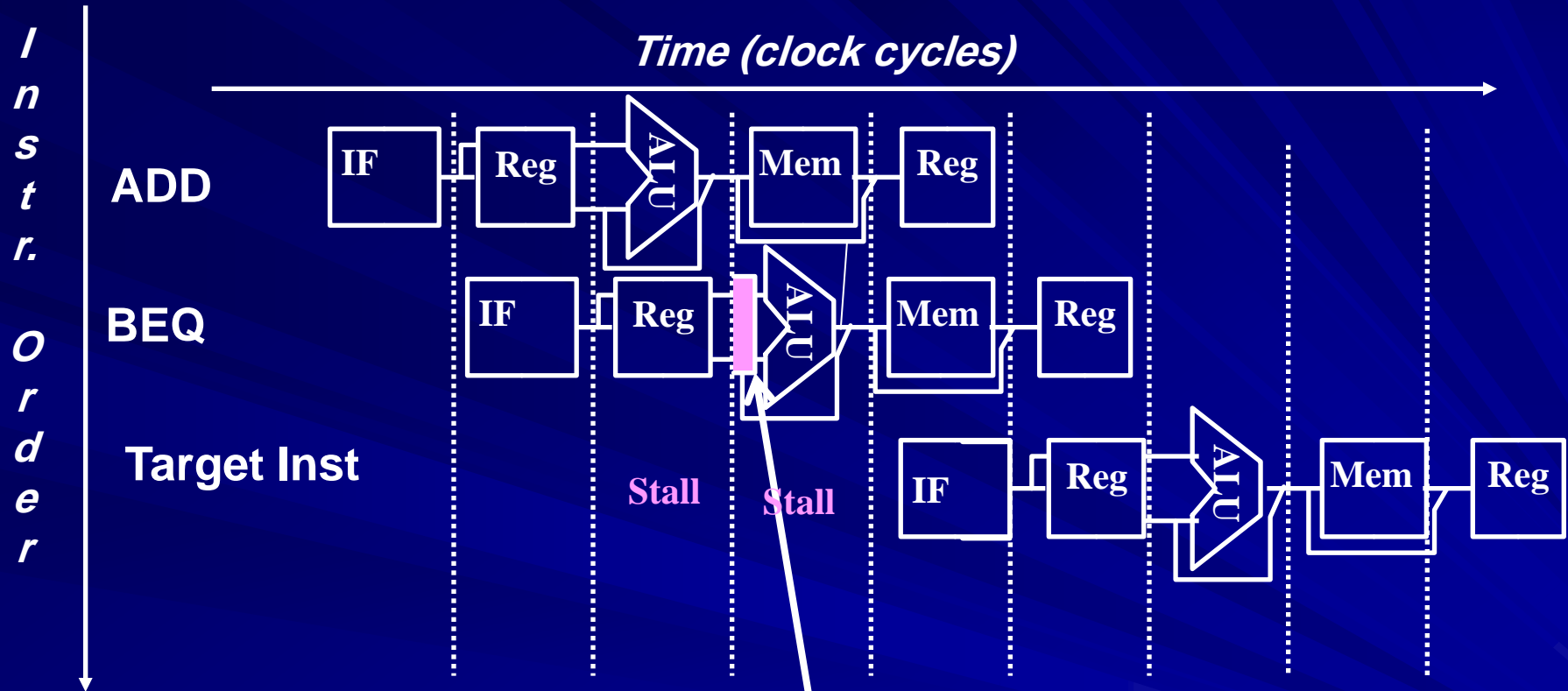
- **Stall**
- **Redo Fetch after branch**
- **Delayed branching**
- **Branch prediction**
- **Multiple Streams**

Solution#1 Stall

- Result of the condition evaluation is available after the EXE stage and the target address is available in the next stage
- Thus 3 stall cycles



Reducing number of Stall



Extra H/W to evaluate condition at the end of ID stage of BEQ instruction

Solution# 2 Redo Fetch after Branch

Branch



Branch
successor



Branch
successor + 1

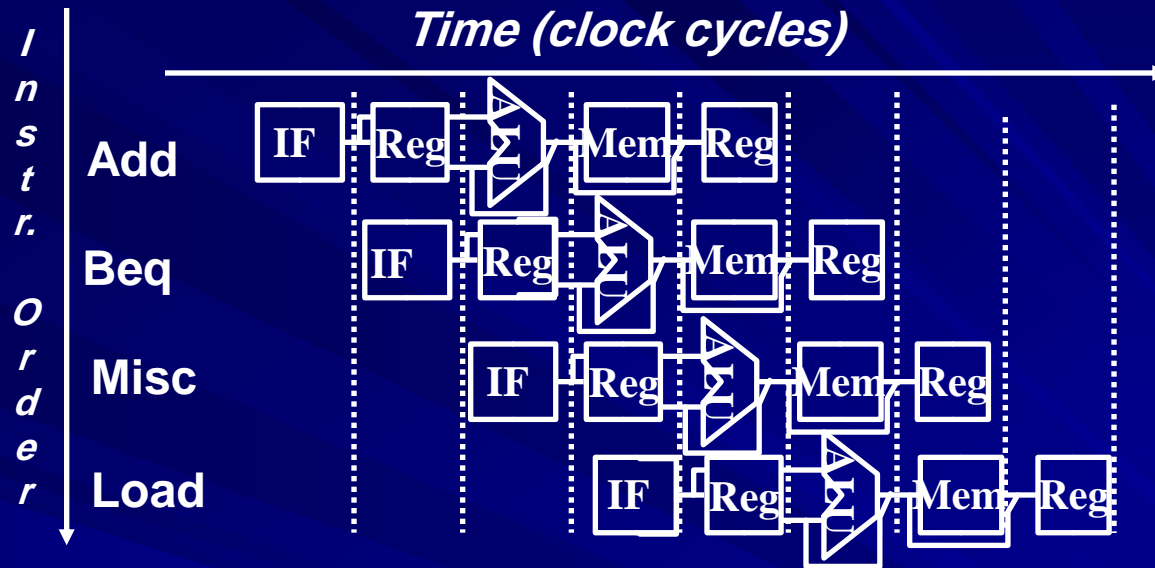


- We know that once a branch has been detected during the Instruction decode stage, the next instruction fetch cycle should essentially be a stall, **if we assume that branch is taken**

Solution# 2 Redo Fetch after Branch

- **However, the instruction fetched in this cycle never performs useful work, and is ignored**
- **Therefore, re-fetch the Branch successor instruction will provide the correct instruction.**

Solution# 3 Delayed Branch – S/W method



- Redefine branch behavior to take place after the next instruction by introducing other instruction (may be No-OP) which is always executed

Solution#4 Prediction

- **This techniques suggest that for a branch instruction we guess one direction of the branch, to begin, then back up if wrong**
- **The two possible predictions are:**
 - **Predict Branch not-taken**
 - **Predict branch taken**

1. Predict – Branch not taken

- This scheme is implemented assuming every branch as branch Not-taken
- So the processor continues to fetch branch as normal instructions

Sequence when branch is not-taken

Branch Inst 'i'	IF	ID	EX	MEM	WB		
Inst. 'i+1'		IF	ID	EX	MEM	WB	
Inst. 'i+2'			IF	ID	EX	MEM	WB
Inst. 'i+2'				IF	ID	EX	MEM WB

Predict Branch not taken

- When the decision has been made, and the branch is taken, then fetch operations are turned into NO-OP and fetch is restarted at the target address

Sequence when branch is taken

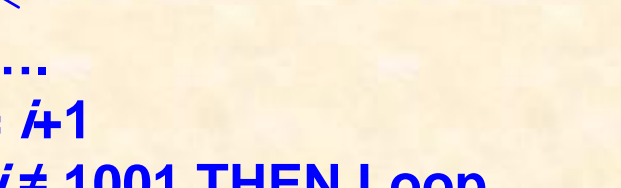
Taken Branch Inst 'i'	IF	ID	EX	MEM	WB		
Inst. 'i+1'		Idle	Idle	Idle	Idle		
Branch target			IF	ID	EX	MEM	WB
Branch target +1'				IF	ID	EX	MEM

2. Predict - Branch taken

- **An alternative way is to treat every branch as Branch taken**
- **As soon as the target address is computed, we assume that the branch is to be taken and start fetching and executing at the target**
- **Let us consider this example of a LOOP to explain the concept:**

2. Predict - Branch taken

```
i = 0
Loop: ←
      .....
      i = i + 1
      IF i ≠ 1001 THEN Loop
      .....
```

A diagram showing a loop structure. A horizontal arrow points from the right to the label 'Loop:'. Below this, there is a block of code: '.....', 'i = i + 1', 'IF i ≠ 1001 THEN Loop', and '.....'. A vertical line descends from the right side of the 'Loop' label, and a horizontal line connects it to the 'Loop' label in the 'IF' statement, forming a loop back to the start of the loop body.

- Here, the branch is taken for 1000 times, so the prediction “Branch Taken” fails 1 in 1000, hence no stall for 1000 times
- Further, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware choice

Solution #5 Multiple Streams

- Have two pipelines
- Pre-fetch each branch into a separate pipeline
- Keep target until branch is executed
- Used by IBM 360/91
- **Results**
 - Leads to bus & register contention
 - Multiple branches lead to further pipelines being needed