



° **Pipelining: Basic and Intermediate Concepts**

Appendix C

- ☐ **Basics of Pipelining**
- ☐ **Implementation of Pipeline**
- ☐ **Hazards and their Solutions**
- ☐ **Exception Handling**
- ☐ **Pipeline with Floating-Point Instructions**
- ☐ **Dynamic Scheduling of Pipelines**

Principles of Pipelining

❖ What is Pipelining?

An implementation technique where multiple instructions are overlapped in execution

- Each step completes a part of instruction
- All stages must be ready to proceed at the same time

➤ Throughput of a pipeline

How often an instruction exits a pipeline?

Number of instructions completed per unit time

➤ Pipeline Processor Cycle

Time required to move an instruction one step down the pipeline

Determined by the slowest stage

Pipe stage or pipe segment

Introduction to Pipelining

- For perfectly balanced set of stages
Time/Instruction in a pipelined processor =
$$\frac{\text{Time/instruction on an unpipelined machine}}{\text{Number of Pipe Stages}}$$

Or Speedup from pipelining = Number of Pipe Stages

- Limitations
 - Pipeline stages are not perfectly balanced*
 - Pipelining involves some overhead*
- Pipelining yields reduction in average execution time/instruction
By decreasing CPI
- *A speedup technique that is not visible to programmers*

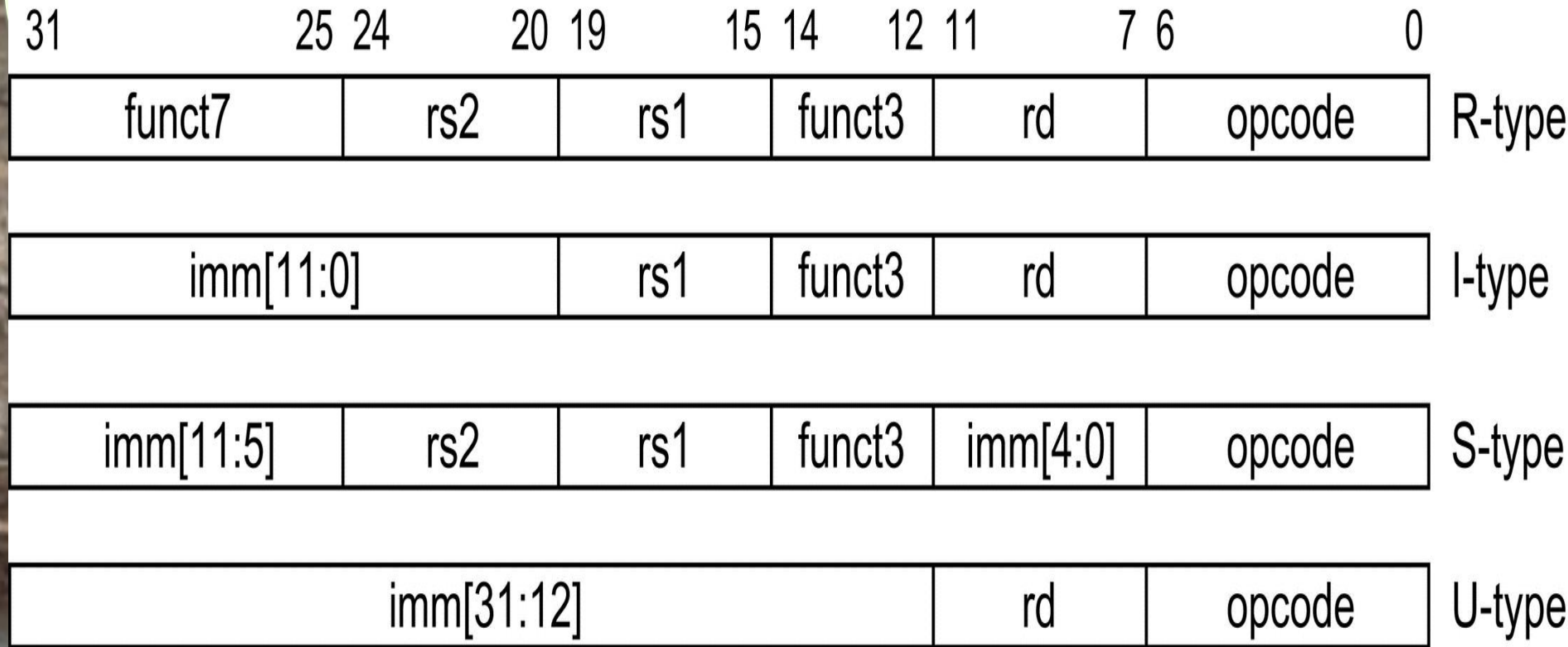
Introduction to Pipelining

- ❖ **The Basics of the RISC-V Instruction Set**
 - All operations are performed on registers and the entire register is involved
 - Only load/store instructions access memory
 - Processor comprises of only a few instruction formats and has single length instructions
- **Focus would be on three classes of integer subset of RISC architecture instructions**
 - Integer ALU Instructions
 - Load/Store word Instructions
 - Branch Instructions
- ❖ **A Simple Implementation of a RISC Instruction Set**
 - Implementation on a non-pipelined machine*
 - An instruction takes at most 5-clock cycles**

Introduction to Pipelining

- **IF (Instruction Fetch Cycle)**
 - Send contents of PC to memory**
 - Fetch current instruction from memory**
 - Update PC by adding 4**
- **ID (Instruction Decode/ Register Fetch Cycle)**
 - Decode operation code**
 - Fetch registers given by the source specifiers**
 - Do equality test on registers for a possible branch**
 - Sign-extend the immediate field**
 - Compute (possible) branch target address**
- ***Fixed field decoding allows all the above operations to be performed in parallel***
 - Some of the operations may not be required**

Instruction layout for RISC-V



- R-type: register-register
- I-type: short immediates and loads
- S-type: stores
- U-type: long immediates

Four major types of instructions

Introduction to Pipelining

- **EX (Execution/Effective Address Cycle)**
 - **ALU operates on operands**
 - **Any one of the following operations is performed**
 - Memory reference instruction**
 - Effective address is calculated*
 - Register-Register ALU instruction**
 - Performs ALU operation on operands*
 - Register-immediate ALU instruction**
 - Performs operations on operand and sign-extended immediate data*
 - Conditional branch**
 - Determine whether the condition is true*
- **MEM (Memory Access Cycle)**
 - **Active only for load/store instructions**

Introduction to Pipelining

Load: Read operand from memory

Store: Store a register content into memory

➤ **WB (Write Back Cycle)**

- **Register-Register ALU instructions or Load instructions are the only active instructions**
Writes the result from ALU or from memory

Overall execution cycles

Branch Instructions: 3 cycles (12%)

Store Instructions: 4 cycles (10%)

All other Instructions: 5 cycles

- ***Average CPI is 4.66 clock cycles (approx.)***
Not an optimal implementation

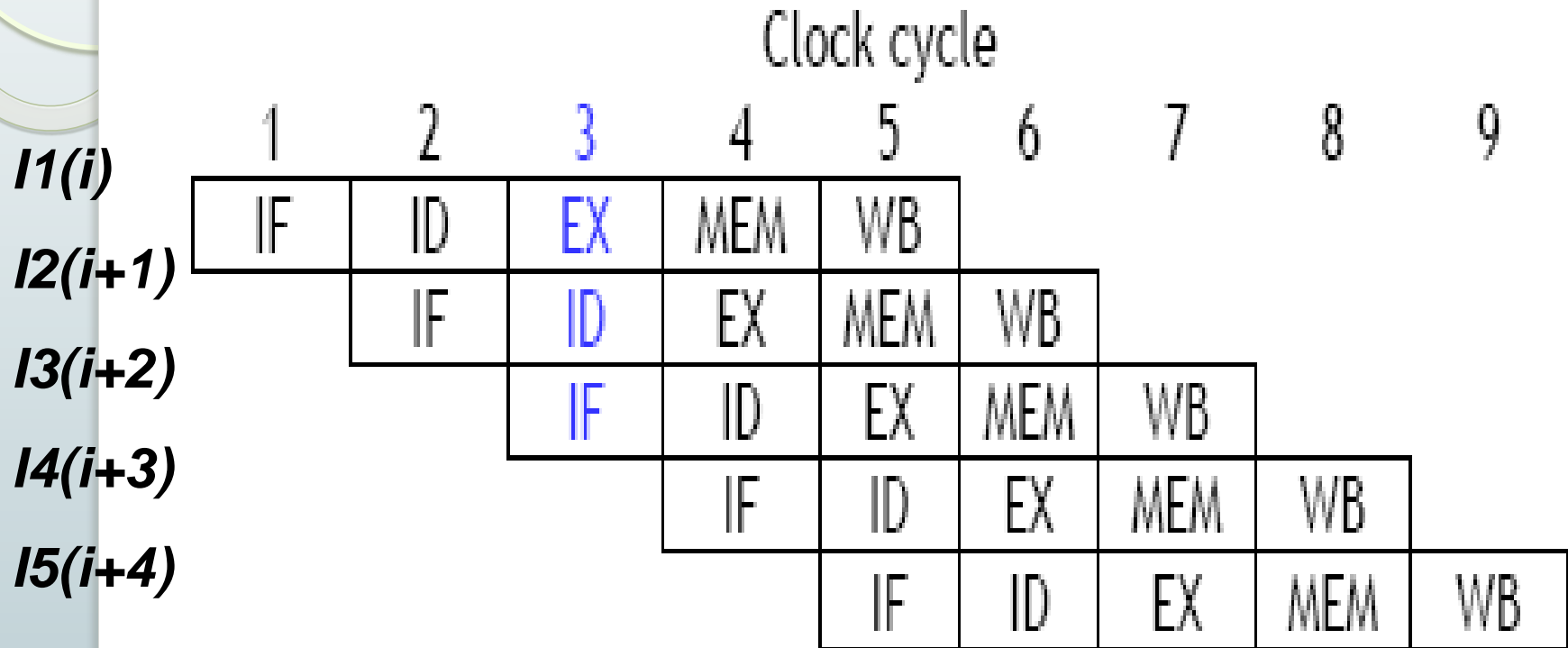
Introduction to Pipelining

- ❖ **The Classic 5-stage Pipeline for a RISC Processor**
 - ➔ **Start a new instruction every clock cycle**
 - ➔ **Instructions overlap should not conflict with use of resources**
- **Data path pipeline shows no conflicts due to three features**
 - ① ***Separate instruction and data caches***

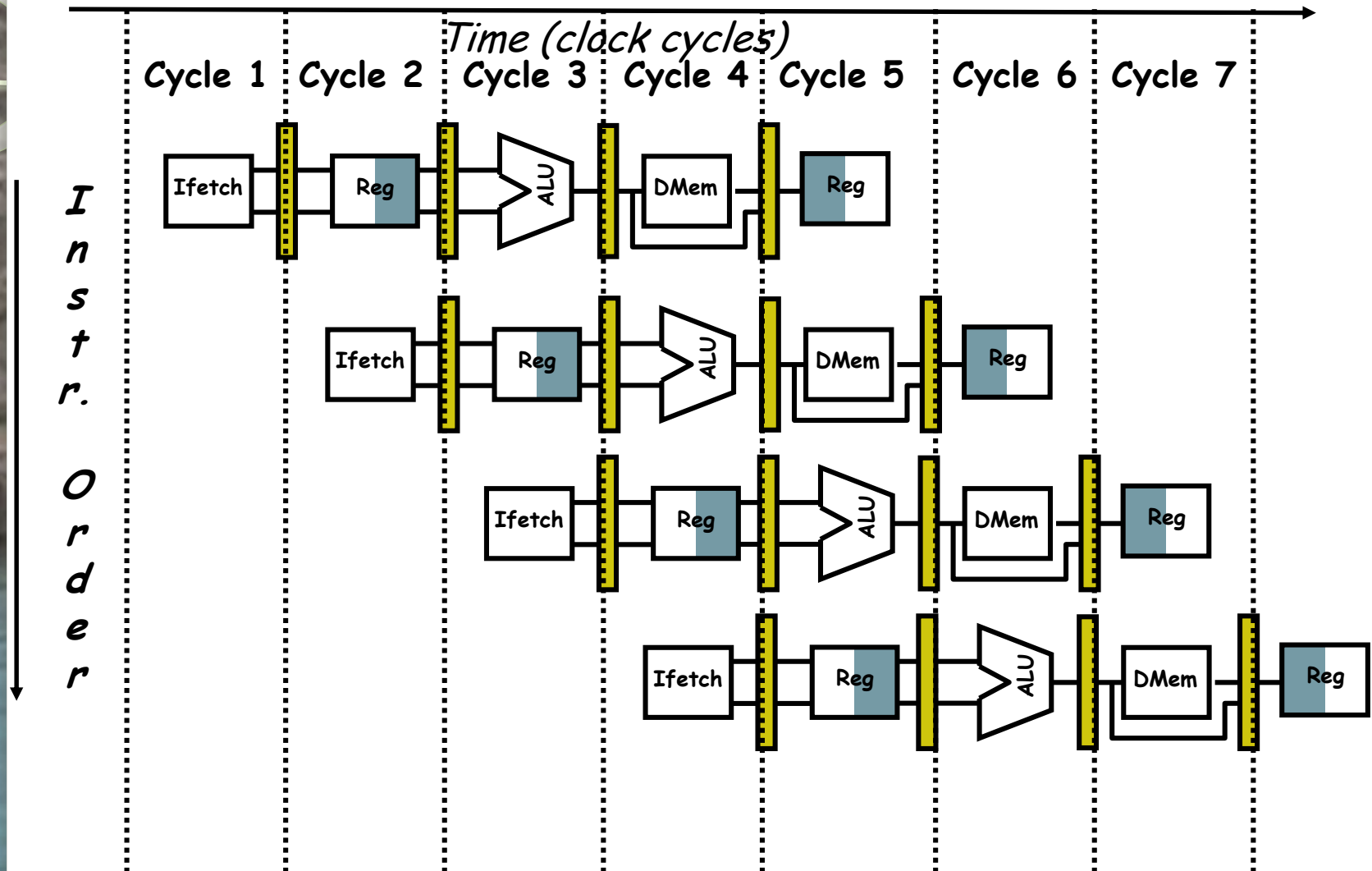
Memory bandwidth is 5-times the bandwidth required in a non-pipelined machine
 - ② ***Register File is accessed in two stages***

Two reads and one write in every cycle
Write is performed in first half and read is performed in the second half of the cycle
 - ③ ***PC is incremented in every cycle to start a new instruction***

Simple RISC Pipeline



Visualizing Pipelining – Series of data paths shifted in time



Introduction to Pipelining


- **Separate ALU is required to calculate branch target address in ID**
- **Instructions should not interfere with each other**
Introduce pipeline registers between successive stages
Pipeline registers also carry intermediate results from one stage to another
- ❖ **Basic Performance Issues in Pipelining**
 - ⇒ **Throughput increases but every instruction takes longer to execute as compared to a non-pipelined machine**
Limits the depth of the pipeline
 - ⇒ **Imbalance among pipe stages contribute to overheads**
Clock cycle time is determined from the delay of the slowest stage

Introduction to Pipelining

- ***Pipeline register delay and clock skew are other overheads***
- **Pipeline registers add setup time**
Time that a register input must be stable before a clock signal that triggers a write occurs
- **Clock skew is the maximum delay between the time when the clock arrives at any two registers**
Imposes a lower limit on the clock cycle time

Example

- Consider the unpipelined 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?



Speedup from pipelining over a non-pipelined architecture, clock skew and setup time = 0.1 ns

Clock cycle = 0.5 ns; ALU instr. (40%) = 4 cycles

Branch instr. (20%) = 4 cycles;

Memory instr. (40%) = 5 cycles

Answer The average instruction execution time on the unpipelined processor is

$$\begin{aligned}\text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI} \\ &= 0.5 \text{ ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5] \\ &= 0.5 \text{ ns} \times 4.4 \\ &= 2.2 \text{ ns}\end{aligned}$$

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 0.1 ns overhead essentially establishes a limit on the effectiveness of pipelining. If the overhead is not affected by changes in the clock cycle, Amdahl's Law tells us that the overhead limits the speedup.

Principles of Pipelining

❑ The Major Hurdles of Pipelining – Pipeline Hazards

Hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle

Reduce performance from the ideal speedup gained by pipelining

▪ There are three classes of pipeline hazards

① Structural Hazards

Occurs due to resource conflicts

Hardware cannot support all possible combinations of instructions simultaneously in overlapped execution

② Data Hazards

When an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline

Pipeline Hazards

③ Control Hazards

Arise from the pipelining of branches and other instructions that change the PC

- Hazards can make it necessary to *stall* the pipeline
- Stall in a pipelined machine requires that some instructions are allowed to proceed while others are delayed
- All instructions issued *later* in the pipeline than the stalled instruction are also stalled
- Instructions issued *earlier* must continue
- No new instructions are fetched during the stall

Pipeline Hazards

❖ Performance of Pipelines with Stalls

- Pipeline speedup =

$$\frac{\text{Average Instruction time w/o pipeline}}{\text{Average Instruction time with pipeline}}$$

$$= \frac{\text{CPI w/o pipelining} * \text{clock cycle w/o pipelining}}{\text{CPI with pipelining} * \text{clock cycle with pipelining}}$$

$$\text{Ideal CPI} = \frac{\text{CPI without pipelining}}{\text{Pipeline depth}} \approx 1$$

- Pipeline Speedup =

$$\frac{\text{Clock cycle w/o pipelining} * \text{Ideal CPI} * \text{Pipeline depth}}{\text{Clock cycle with pipelining} * (\text{Ideal CPI} + \text{pipeline stall cycles})}$$

Pipeline Hazards

- **Pipeline Speedup =**
$$\frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{Ideal CPI} + \text{pipeline stall cycles}}$$
- **Pipeline Speedup =**
$$\frac{\text{Pipeline depth}}{I + \text{pipeline stall cycles}}$$
- **If there are no stalls, the speedup is given by**
The number of pipe stages

Pipeline Hazards

❖ Structural Hazards

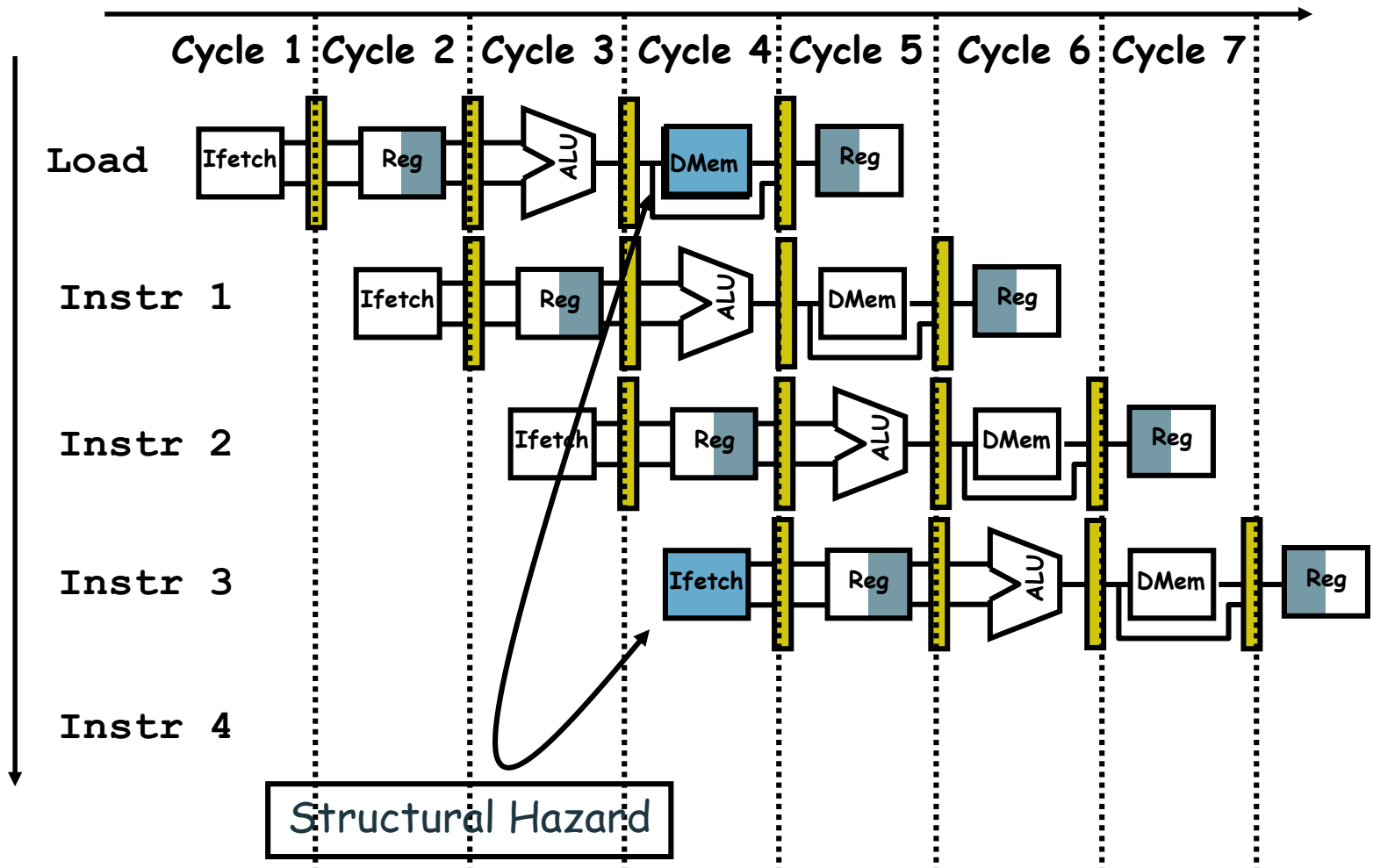
If some combinations of instruction cannot be accommodated due to resource conflicts, there is a structural hazard in the processor

There are two types of structural hazards

- **Arises when some functional unit is not fully pipelined**
A sequence of instructions that all use that functional unit cannot be sequentially initiated
- **Some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute**
The pipeline will stall one of the instructions until the required unit becomes available

One Memory Port/Structural Hazard

Time (clock cycles)



*I
n
s
t
r.

O
r
d
e
r*

Pipeline Hazards

❖ Data Hazards

Occur when the order of access to operands is changed by the pipeline versus the normal order encountered by sequentially executing instructions

- Occur because instructions are overlapped in execution
- Assume *i* occurs before *j* and both use register *x*
- Three different types of data hazards occur

1. Read after Write (RAW)

*When the read of register *x* by *j* occurs before write by *i**

- *j* would use the wrong value of *x*

2. Write after Read (WAR)

*Read of *x* by *i* takes place after a write by *j* to *x*
i uses the wrong value of *x**

Pipeline Hazards

- This hazard is not possible in a simple, five stage integer pipeline
- It may occur
 - When instructions are reordered
 - In dynamically scheduled pipelines

3. Write after Write (WAW)

Write by i into x takes place after write to x by j
All later instructions get the wrong value of x

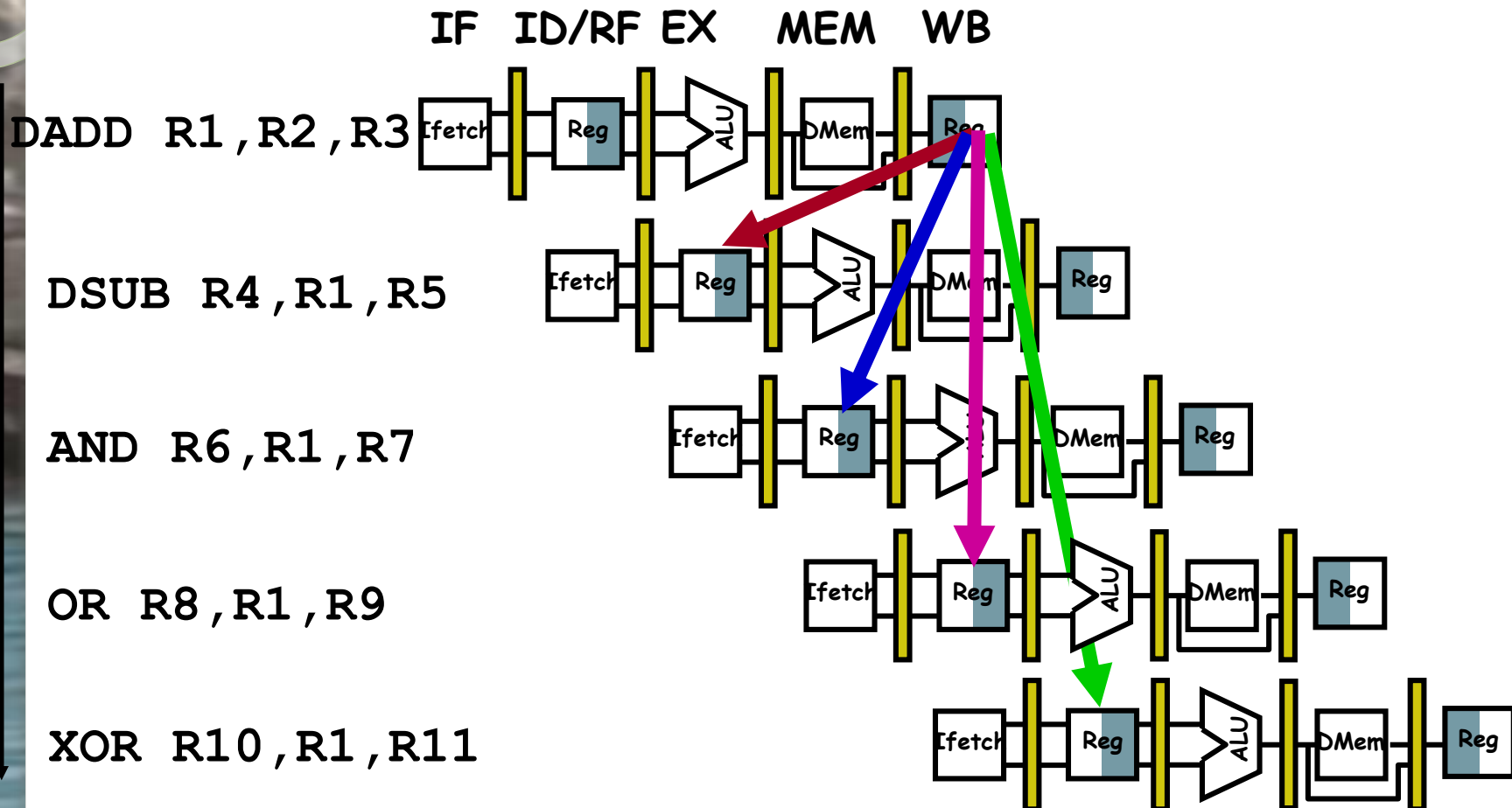
- This hazard is not possible in a simple, five stage, integer pipeline
- Focus only on RAW hazards
- Example code and the hazards that may occur

Data Hazards

Time (clock cycles)

*I
n
s
t
r.*

*O
r
d
e
r*



Pipeline Hazards

➤ Solution

Simple hardware technique called *forwarding*, *bypassing* or *short-circuiting*

○ Key insight

Result is needed by successive instructions only after it has been produced

Minimizing data hazards by forwarding

➤ Data hazards requiring stalls

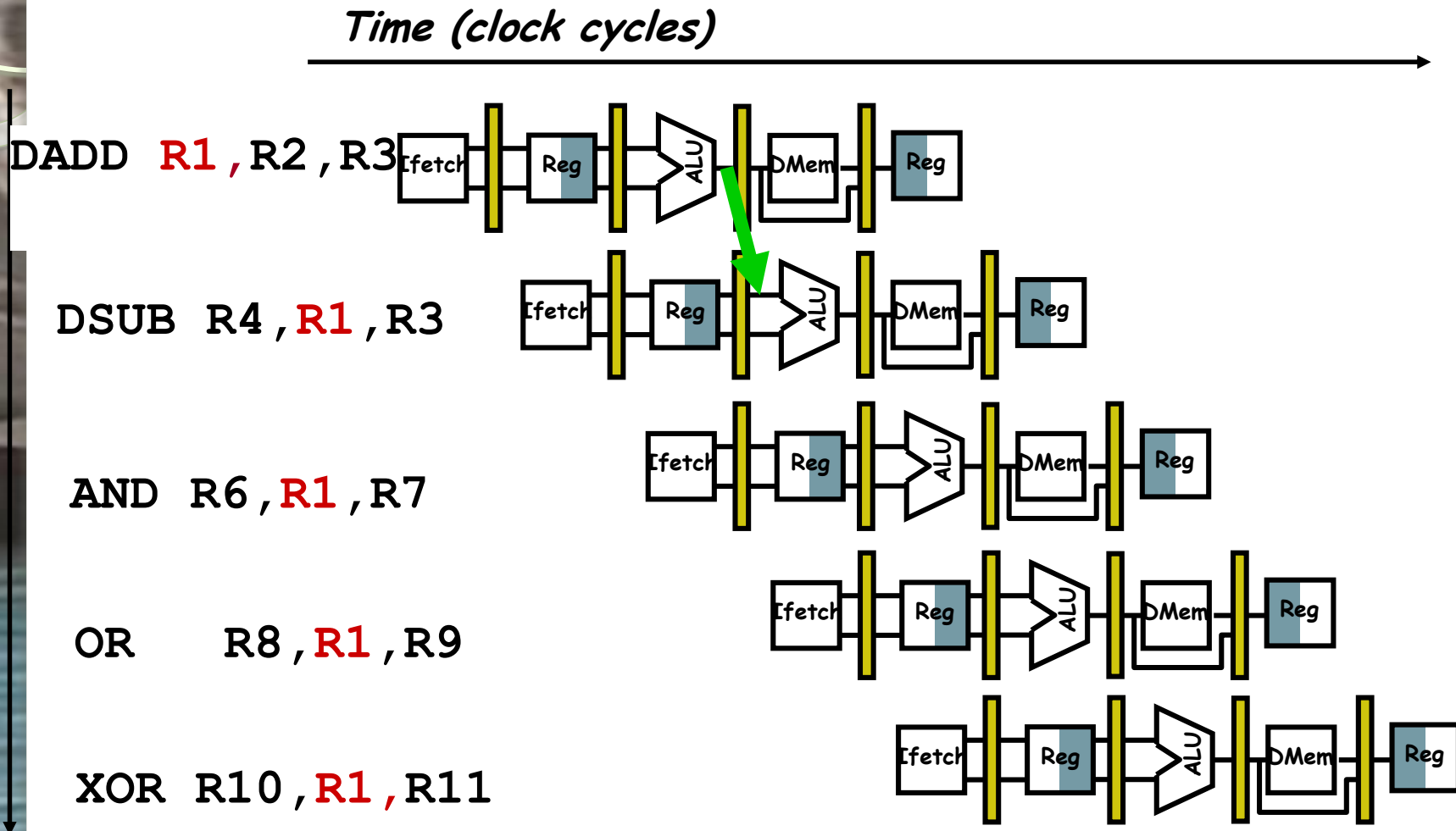
Load instruction followed by an instruction that uses the load output

▪ Add *pipeline interlock* to preserve the correct execution pattern

Pipeline interlock detects hazard and stalls the pipeline until the hazard is cleared – CPI increases

Forwarding to Avoid Data Hazard

Instruction Order

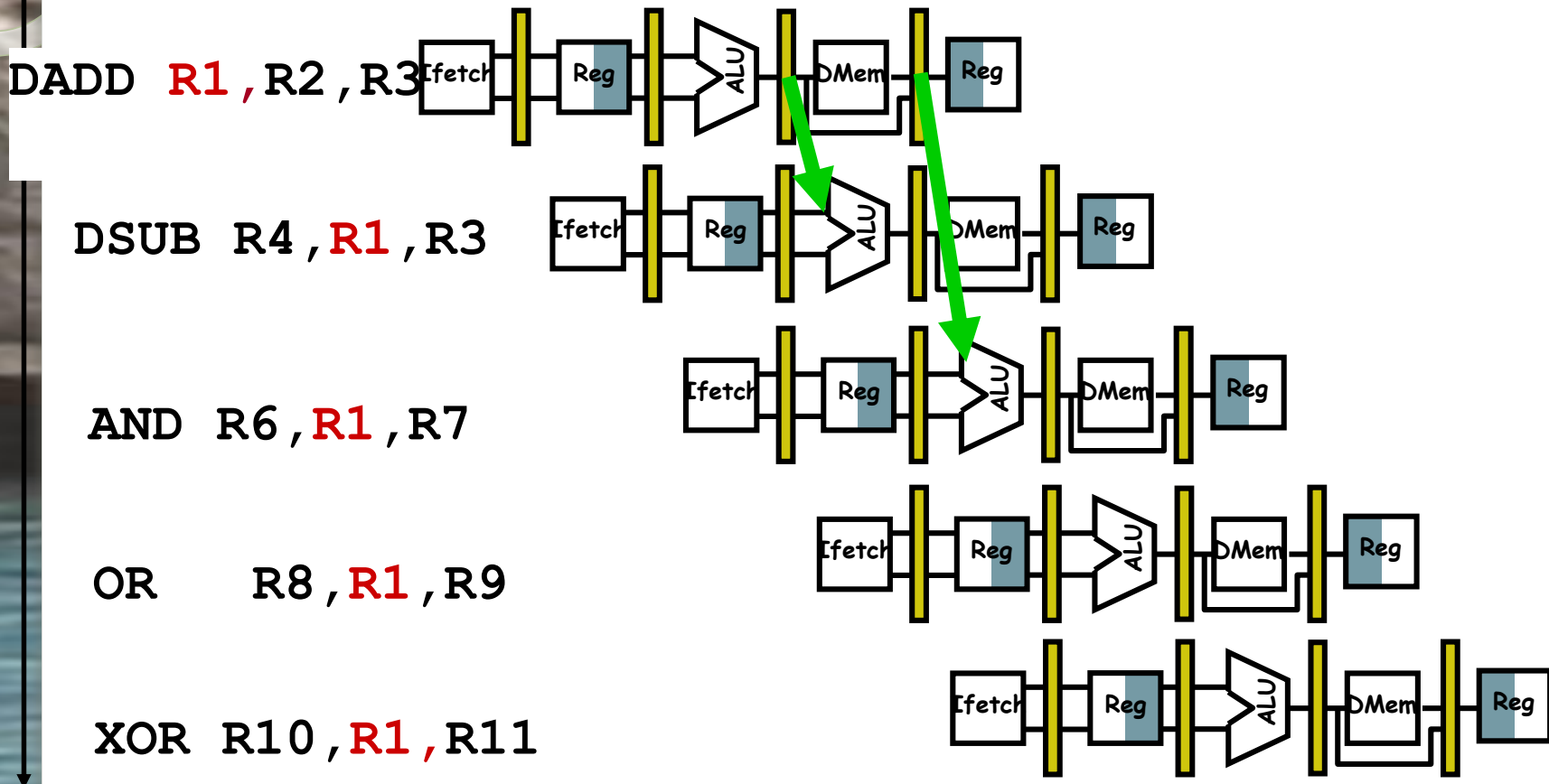


Forwarding to Avoid Data Hazard

I
n
s
t
r.

O
r
d
e
r

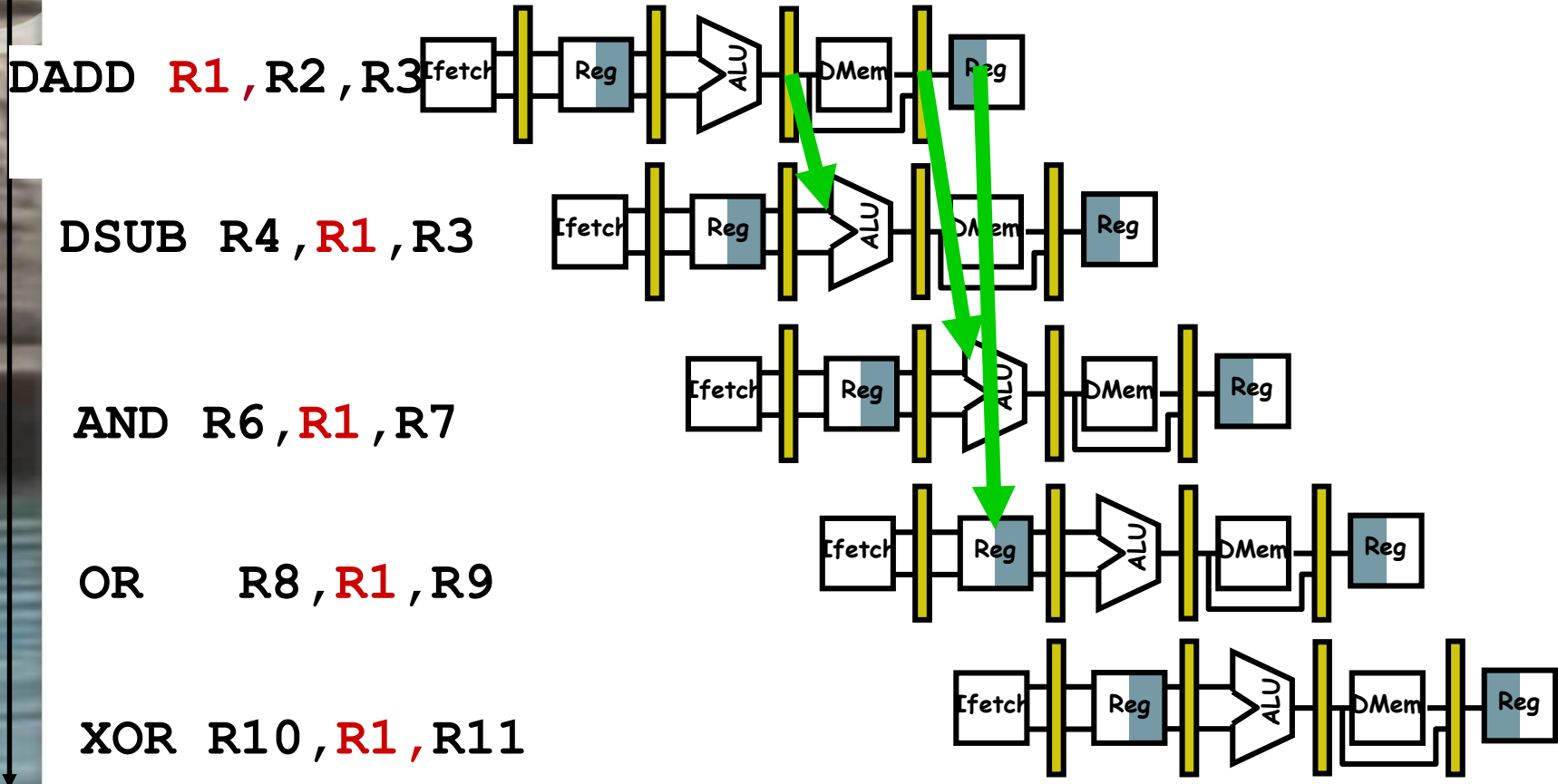
Time (clock cycles)



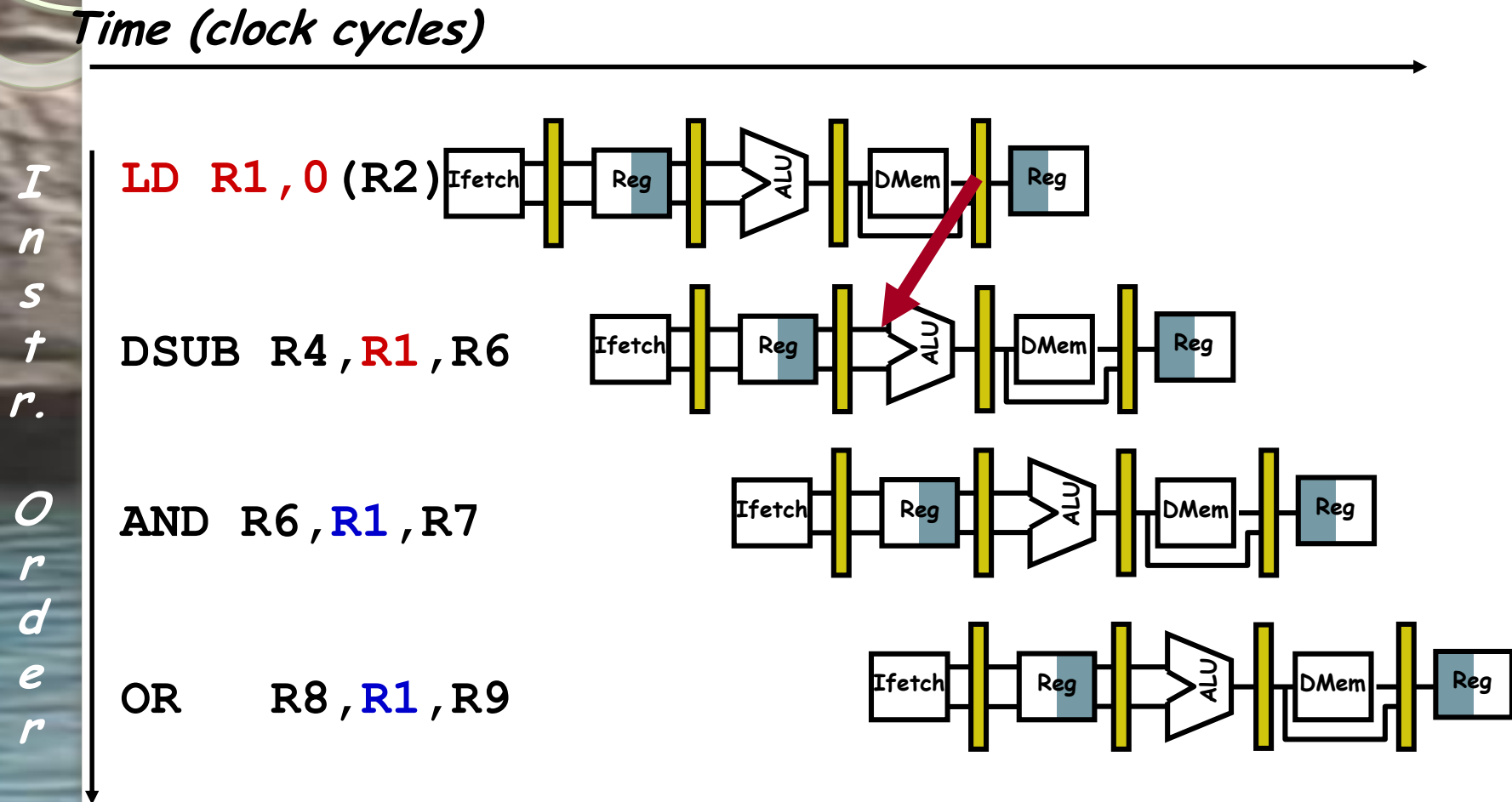
Forwarding to Avoid Data Hazard

Instruction Order

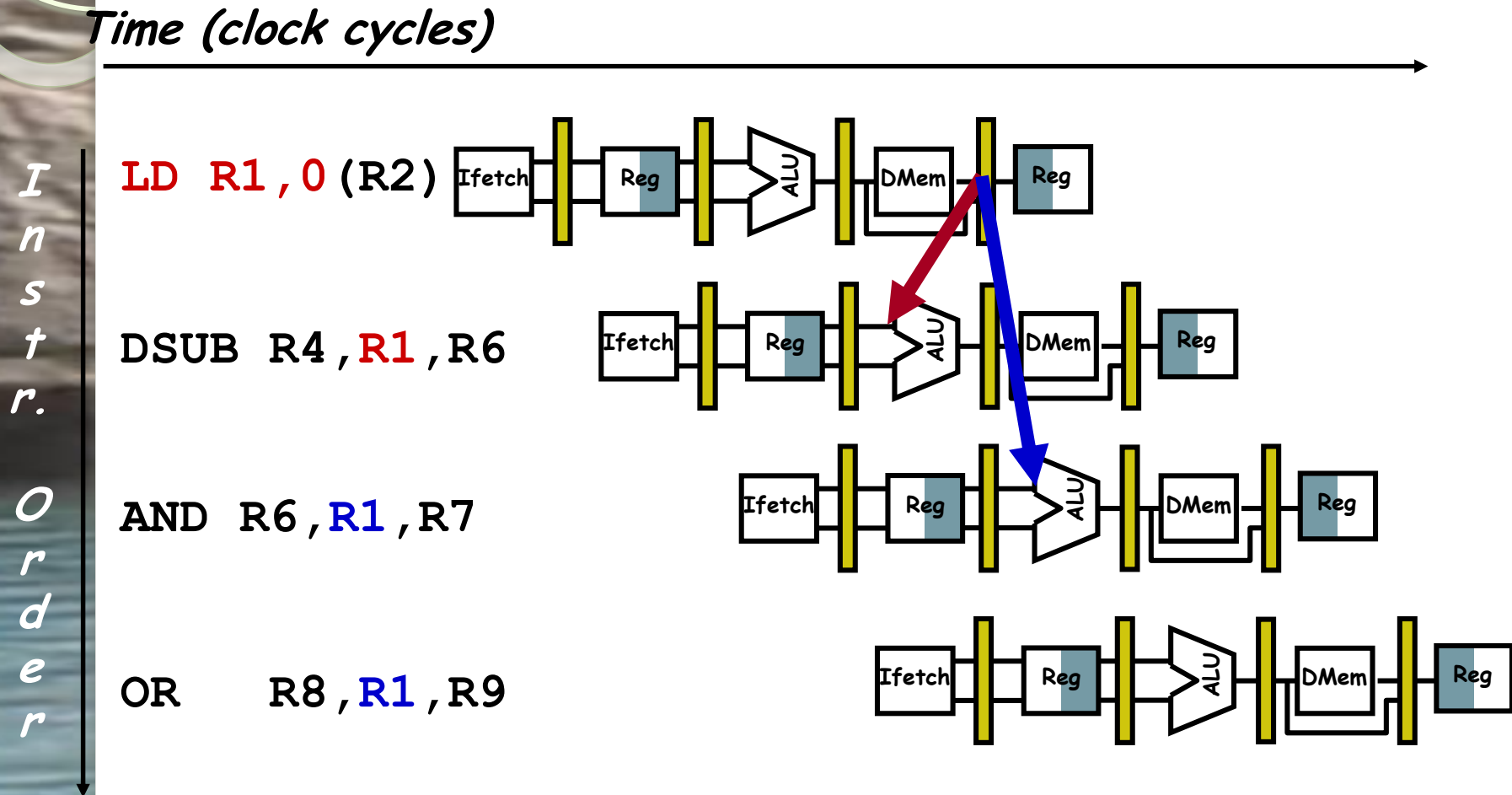
Time (clock cycles)



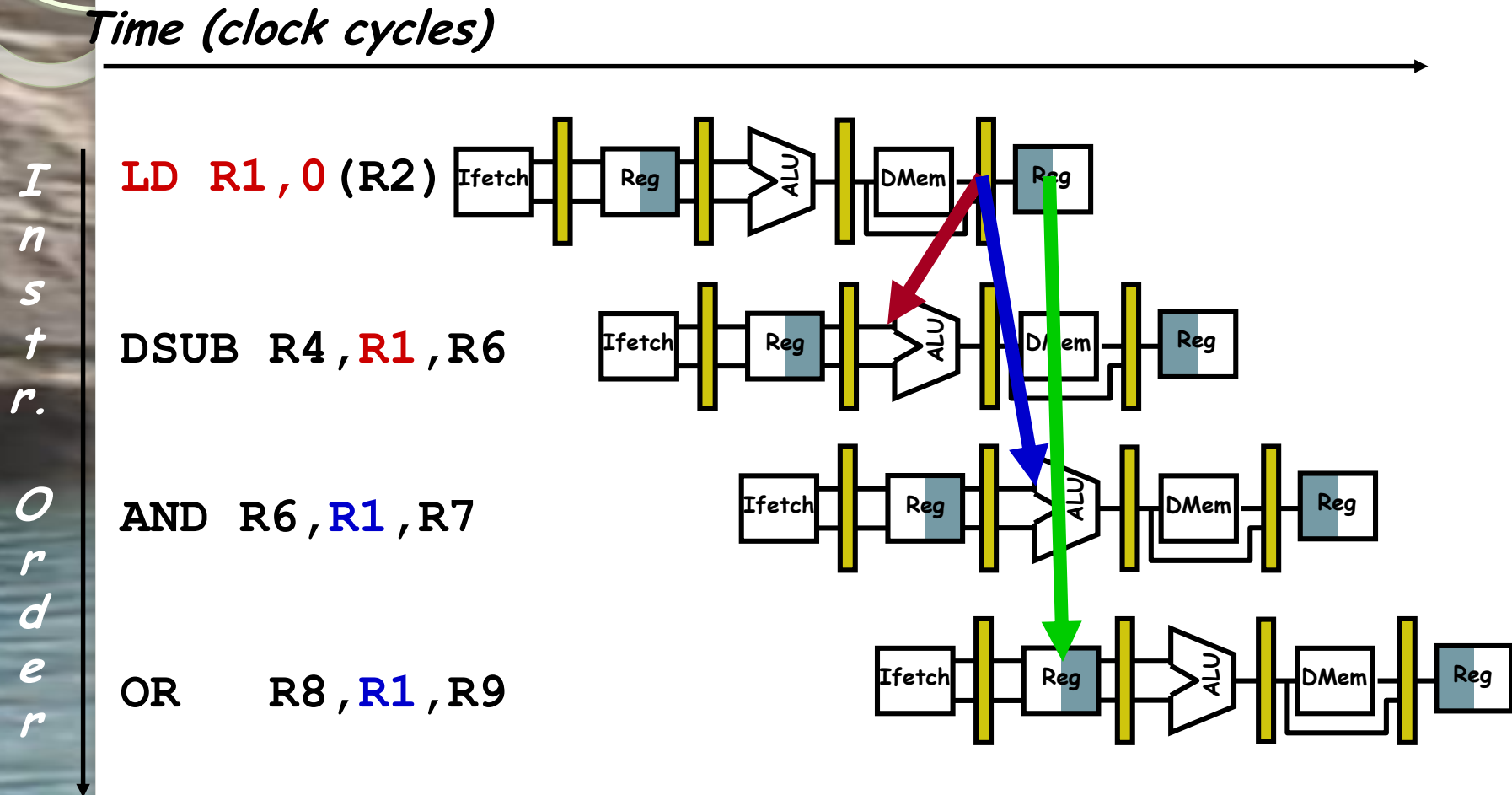
Data Hazard Even with Forwarding



Data Hazard Even with Forwarding

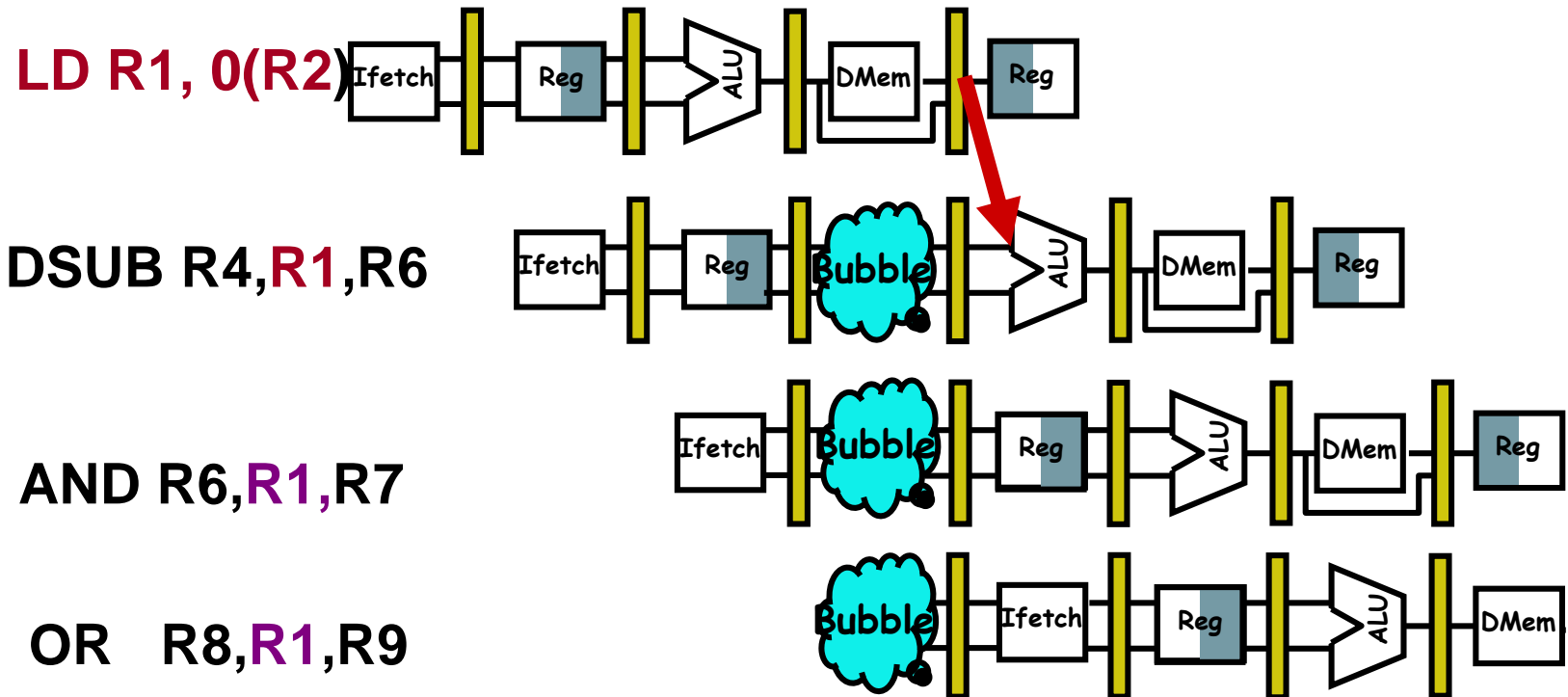


Data Hazard Even with Forwarding



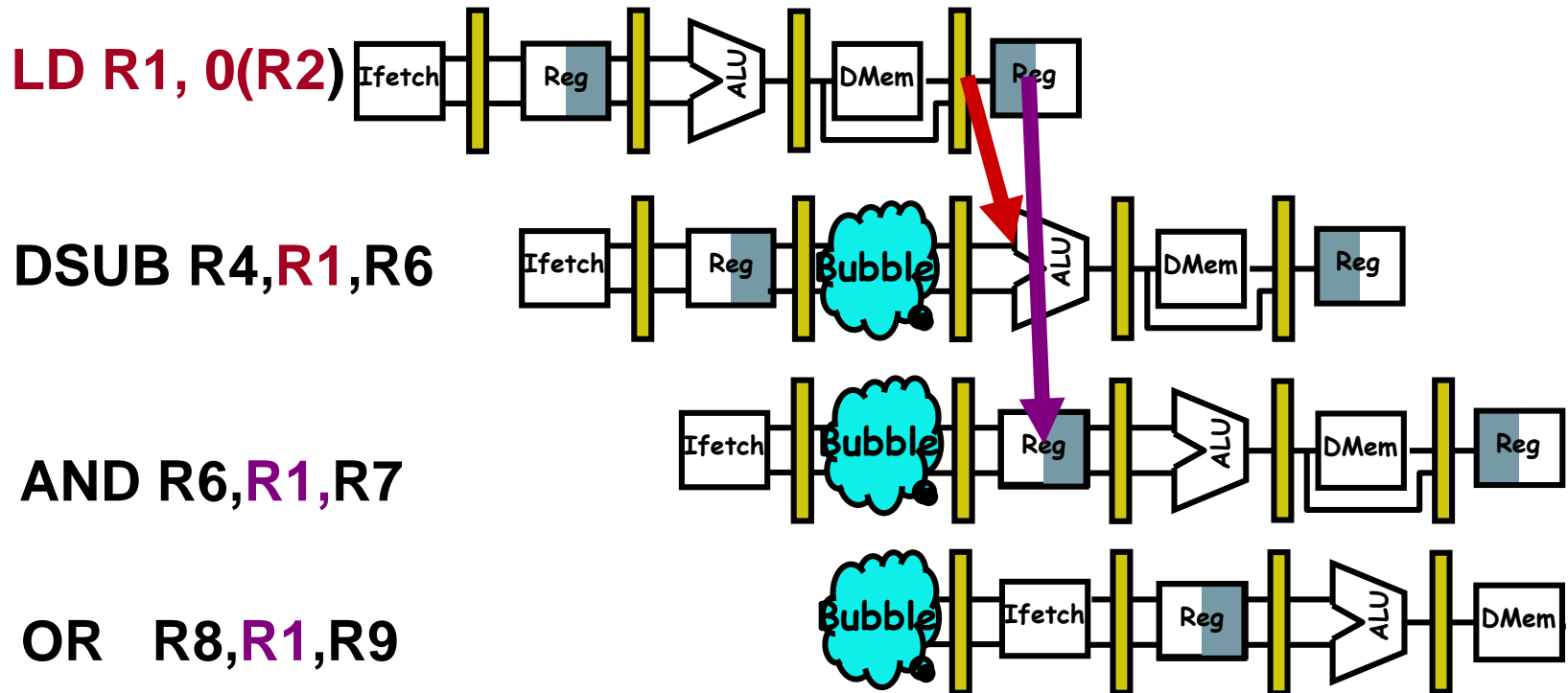
Resolving the Load Data Hazard

Time (clock cycles)



Resolving the Load Data Hazard

Time (clock cycles)



Pipeline Hazards

❖ Branch Hazards

Causes more performance loss than data hazards

Taken or untaken branches

Fall through and content of PC

➤ Reducing Pipeline Branch Penalties

Four simple compile time schemes, where predictions are static

Compile time guesses

Static

Fixed for each branch during the entire execution

① Freeze the pipeline

Simplest

- **Hold all instructions after the branch until the branch direction is known**

Pipeline Hazards

- ② ***Predict the branch as not taken and allow the pipeline to continue***

Do not change the machine state until the branch outcome is definitely known

- ③ ***Predict the branch as taken***

- **Begin fetching and executing at the target as soon as the target address is computed**

- **The simple pipeline of RISC-V has the same penalty for all three situations**

These schemes make more sense in machines with deeper pipeline

- ④ ***Use delayed branch technique***

Used in many early RISC processors

Works well in simpler pipelines

The predict-not-taken scheme

Branch not taken

Branch taken

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

Pipeline Hazards

- The execution cycle with a branch delay of length 1 is
 - Branch instruction*
 - Sequential successor*
 - Branch target if taken*
- Sequential successor is in the branch delay slot
 - This instruction is always executed
- Job of the compiler is to make the successor instruction valid and useful
- Delayed branch technique is useful for simple pipelines where hardware prediction was too expensive
 - Technique complicates implementation when there is dynamic branch prediction

Behavior of a delayed branch scheme

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

Pipeline Hazards

- **Performance of Branch Schemes**

Pipeline Speedup =

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

Pipeline Speedup =

$$\frac{\text{Pipeline depth}}{1 + \text{Branch frequency} * \text{Branch penalty}}$$

- **Both conditional and unconditional branch contribute to the stalls**

Reduce Cost of Branches through Prediction

- **With the increase in the pipeline depth
Branch penalty increases**
 - Delayed branch schemes are not sufficient
- **More aggressive means of predicting branches**
 - **Two classes**
 - **Low-cost static schemes**
 - Relies on information available at compile time
 - **Predict branches dynamically**
- ❖ **Static Branch Prediction**
 - Use profile information from earlier runs
 - **Misprediction rate is higher for integer programs as compared to floating-point programs**