Lecture 2

Processor Pipeline

Overview

- Instruction Set Architecture
  - Overview of MIPS
- Quantifying Processor Performance
  - Review of commonly used formula and terms
- Simple Processor Architectures:
  - Single-Cycle Architecture
  - Multi-Cycle Architecture
  - Pipeline Architecture
    - Hazards detection and resolution

INSTRUCTION SET ARCHITECTURE

Design: **Storage Architecture**

- Major design consideration for an ISA:
  - Decide where to store intermediate computation result internally in the processor
  - Influence the format of the instructions
- Dominant storage architecture:
  - **G**eneral **P**urpose **R**egister (**GPR**)
  - Internal storage = a number of addressible registers
- Alternatives, less common storage architectures:
  - Stack, Accumulator, Memory-Only

Design Philosophy

- Two major approaches in Instruction Set Architecture (ISA) design:
  - **C**omplex **I**nstruction **S**et **C**omputer (**CISC**)
    - Provide as many instructions as possible to the programmer
    - Possible for a single instruction to accomplish complex operation, e.g. matrix multiplication
  - **R**educed **I**nstruction **S**et **C**omputer (**RISC**)
    - Provide the minimum number of instructions to the programmer
    - Complicated operations are coded from the simpler counterparts

CISC Instruction Set Architecture

- Instructions carry different amount of information
  - ➔ Varying length of instruction

- Example CISC Microprocessor:
  - The Intel IA32 Architecture:
    - Basic instructions from earlier x86 series are still maintained
    - Each generation adds on more instructions
    - Single Instruction Multiple Data (SIMD) instructions from the Multimedia eXtensions
    - Even more SIMD instructions from the 3DNow! extensions

RISC Instruction Set Architecture

- Observation:
  - Over 70% of CISC instructions not used often.
    - Shrink instruction set to the most common 30%
- Characteristics:
  - Instructions are of fixed length
  - Very simple processor designed to be very fast
  - Almost all instructions limited to registers only
    - Memory operations limited to load and store only
  - Rely on compiler to optimize code
- Example of RISC machine:
  - MIPS CPUs

The Current State of CISC ISA

- RISC has essentially won the "RISC vs CISC" debate:
  - Complex instruction set processor is much harder to design / validate / optimize
- Intel IA32/x86-64 processors use the following approach:
  - Each CISC instruction is broken into a series of simpler RISC-like operation (μops) internally
  - The processing core executes the μops instead of the CISC instruction directly

Example ISA: **MIPS Basics** (1/2)

- We will use MIPS as an example ISA to illustrate pipeline processor designs

- Typical GPR ISA:
  - 32 registers; register 0 always has the value 0
  - We'll denote them as $0, $1, …, $31 (or $r0… $r30)

Category	Type	Format
Arithmetic	Register-to-Register	**ADD rd, rs, rt** rd ← rs + rt
	Immediate Value	**ADDI rt, rs, IMV** rt ← rs + IMV
	Unsigned	**ADDU rd, rs, rt** rd ← rs + rt

MIPS Basics (2/2)

Category	Type	Format
Memory	Load from memory	**`LW rt, offset(rs)`** `rt ← mem[rs + offset]`
	Store to memory	**`SW rt, offset(rs)`** `mem[rs + offset] ← rt`
Branching	Conditional	**`BEQ rs, rt, label`** Jump to label if rs == rt, otherwise continue to next instruction
	Unconditional (Jump)	**`J label`** Jump to label

MIPS Assembly Code: An Example

```c
//Iterative Fibonacci in C
//Assume n is non-negative

int cur = 1, p1 = 1, p2 = 1, j;
//p1 = n-1 fib term
//p2 = n-2 fib term

if( n > 2 )
   for ( j=3; j<=n; j++ ) {
       cur = p1 + p2;
       p2 = p1;
       p1 = cur;
   }

//cur is the nth fib term
```

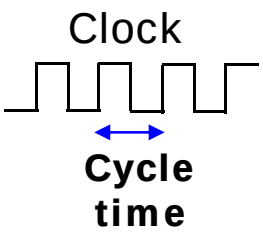
```
#$r1 = cur, $r2 = p1, $r3 = p2
#$r4 = j, $r13 = n

        li $r1, 1
        li $r2, 1
        li $r3, 1

        subi $r10, $r13, 2
        ble  $r10, $r0, done

        li   $r4, 3
loop:   add  $r1, $r2, $r3
        move $r3, $r2
        move $r2, $r1
        addi $r4, $r4, 1
        ble  $r4, $r13, loop
done:
```

PROCESSOR PERFORMANCE

Execution Time: **Clock Cycles**

- **Clock cycles** is the basic time unit in machine

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$



- **Cycle time** (or cycle period or clock period)
  - Time between two consecutive rising edges, measured in seconds.
- **Clock rate** (or clock frequency)

  = 1 / cycle-time

  = number-of-cycles / second

  - Unit is in Hz, 1 HZ == 1 cycle / second

Execution Time: **Introducing CPI**

- A given program will require

Some number of instructions (machine instructions)

↓ × **Average Cycle per Instruction (CPI)**

Some number of cycles

↓ × **cycle time**

Some number of seconds

- We use the average of CPI as different instruction take different number of cycles to finish

Execution Time: **Version 2.0**

- Average Cycle Per Instruction (CPI)

  CPI = (CPU time × Clock rate) / Instruction count

  = Clock cycles / Instruction count

CPU time	=	$\frac{\text{Seconds}}{\text{Program}}$	=	$\frac{\text{Instructions}}{\text{Program}}$	x	$\frac{\text{Cycles}}{\text{Instruction}}$	x	$\frac{\text{Seconds}}{\text{Cycle}}$

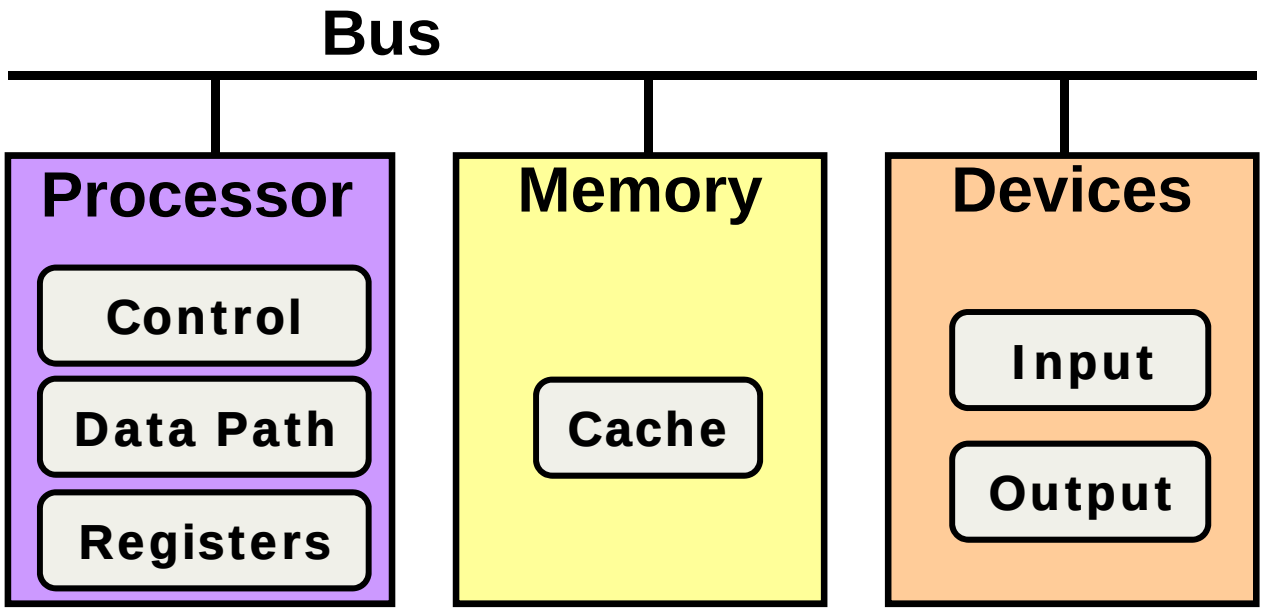
$$\text{CPI} = \sum_{k=1}^{n} \text{CPI}_k \times F_k \quad \textbf{where} \quad F_k = \frac{I_k}{\text{Instruction count}}$$

I_k = Instruction frequency

SIMPLE PROCESSOR ARCHITECTURE

Computer Organization: Recap

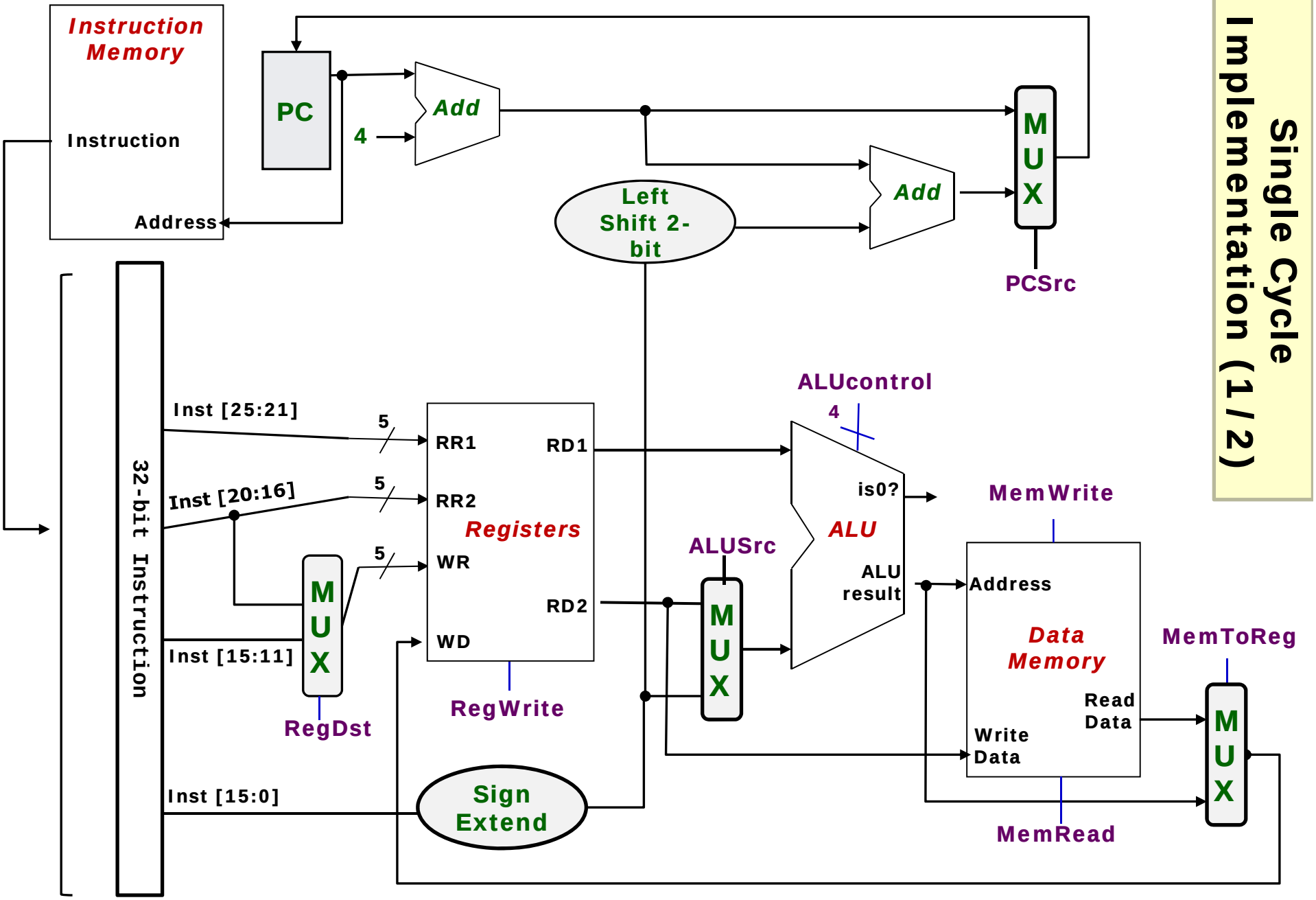
- Major components of a computer system:
  - **Processor**, **Memory** and **Devices**
  - Buses for transporting data between component
- Von Neumann's **stored-memory concept**:
  - Data and program are stored in memory



5-Stage MIPS Instruction Execution

Stage	`add $3, $1, $2`	`lw $3, 20( $1 )`	`beq $1, $2, label`
Fetch	Read inst. at [PC]	Read inst. at [PC]	Read inst. at [PC]
Decode & Operand Fetch	oRead [**$1**] as *opr1* oRead [**$2**] as *opr2*	oRead [**$1**] as *opr1* oUse ***20*** as *opr2*	oRead [**$1**] as *opr1* oRead [**$2**] as *opr2*
ALU	*Result* = *opr1* + *opr2*	*MemAddr* = opr1 + opr2	*Taken* = (*opr1* == *opr2*)? *Target* = **PC** + Label*
Memory Access		Use *MemAddr* to read from memory	
Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	if (*Taken*) **PC** = *Target*

Single Cycle Implementation (1/2)

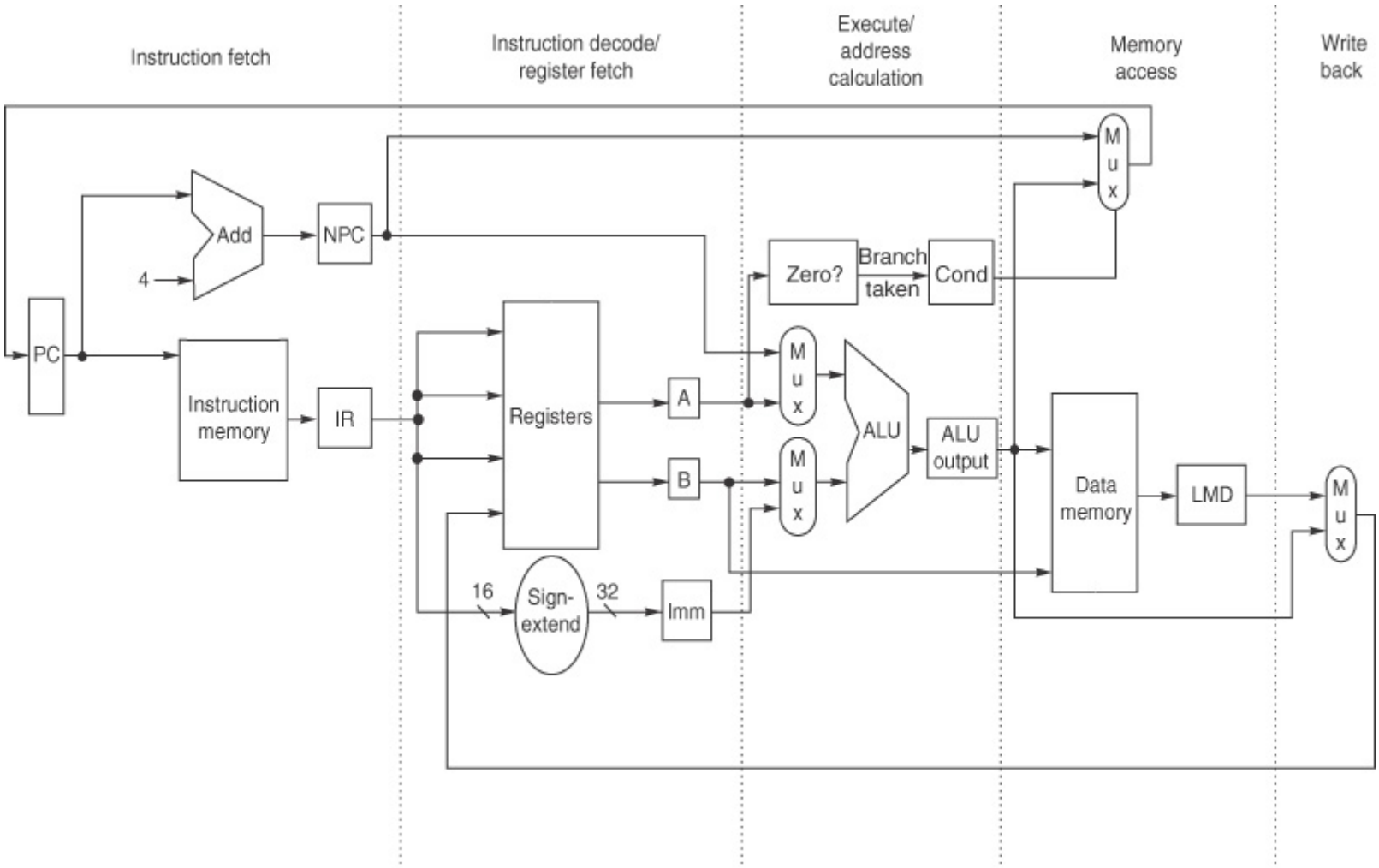


Single-Cycle Implementation(2/2)

- **Characteristics:**
  - Each instruction takes 1 long clock cycle
  - Updating the visible states (PC, data memory, and registers) at the end of the clock cycle
  - No intermediate storage during execution

- **Drawbacks:**
  - Inefficient if instructions require different amount of work

Multi-Cycle Implementation(1/2)



Multi-cycle implementation (2/2)

- **Characteristics:**
  - Each instruction takes different number of cycles to execute

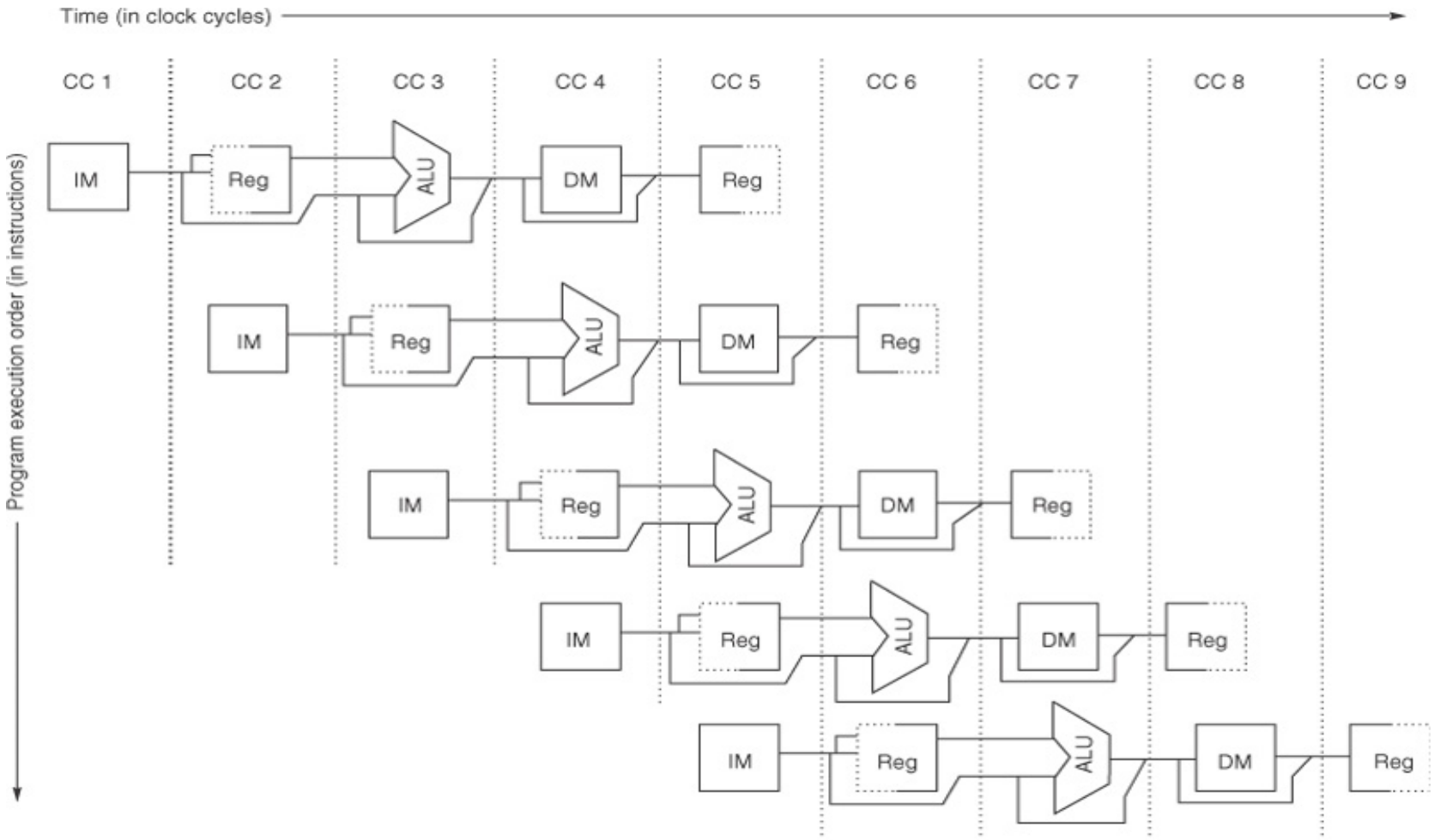
- Average **C**ycle **P**er **I**nstruction (**CPI**):
  - Assumptions:
    - 4 cycles for store and branches
    - 5 cycles for ALU and load
  - Instruction Mix:
    - 40% ALU, 20% Branch, 20% load, 20% store
  - CPI:
    - (20%+20%) x 4 + (40%+20%) x 5 = **4.6**
  - Assume 1ns clock period (1 GHz processor)
    - Average instruction execution time **4.6**ns

Processor Pipelining (1/4)

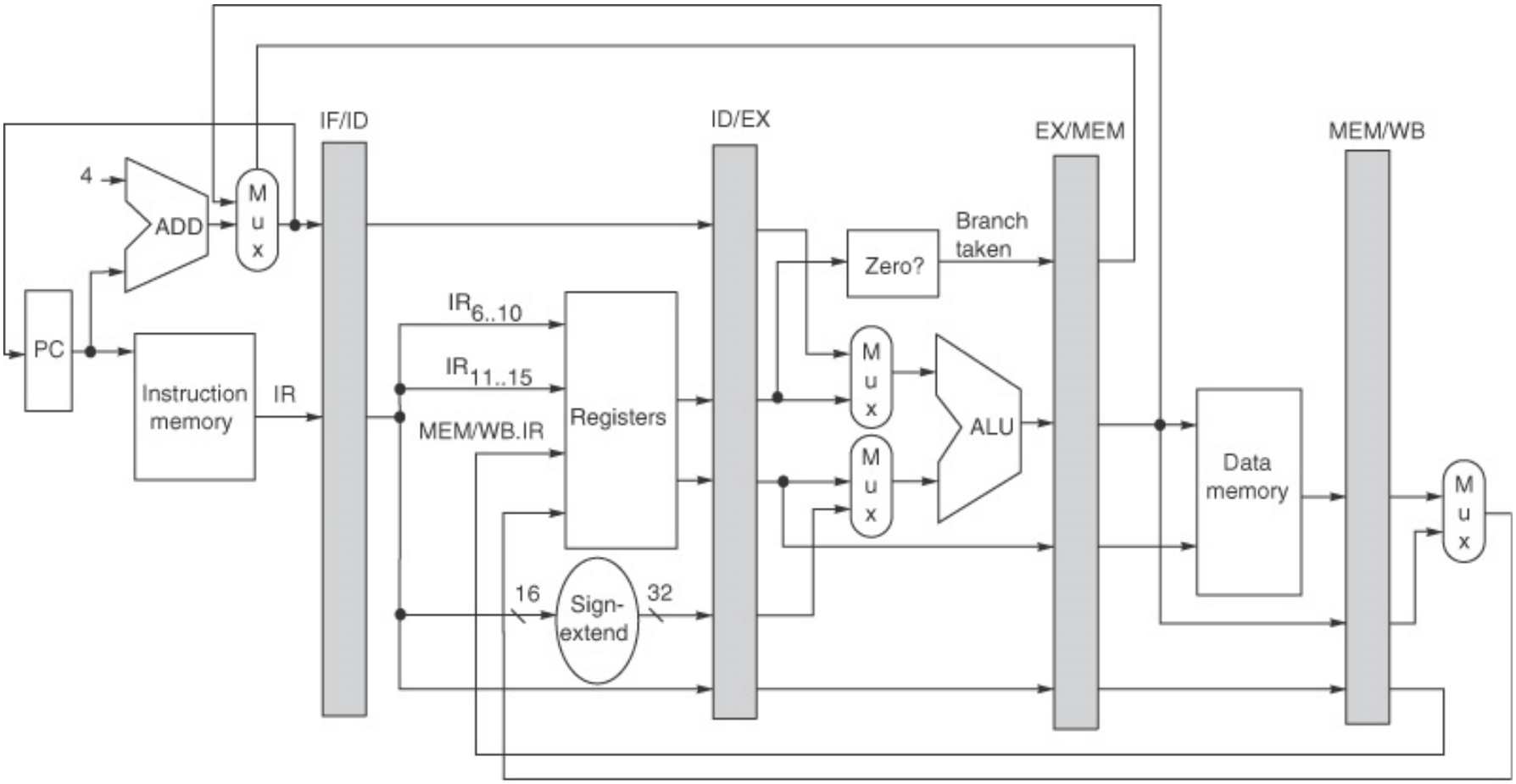
- A technique to improve processor performance by overlapping execution of multiple instructions

- **Basic Idea:**
  - Split the instruction execution into multiple stages
    - Similar to multi-cycle implementation
  - **Start a new instruction in each clock cycle**
  - Different pipeline stages are completing different parts of different instructions in parallel

Processor Pipelining (2/4)



Processor Pipelining (3/4)



Processor Pipelining (4/4)

- Pipeline Registers / Latches
  - Added as buffers between two adjacent pipeline stages
  - Passes data and control signal to the next pipeline stage
  - Content is copied to the next pipeline latch until it is no longer useful
  - Introduces delay/overhead

Processor Pipelining

- **Characteristics:**
  - Each pipeline stage corresponds to a **clock cycle**
  - Each instruction effectively need to travel the whole pipeline depth during execution (why?)

    ➔ i.e. Each instruction takes 5 clock cycles in this example

- **Advantage:**
  - Pipelining helps **throughput** (number of instructions executed in unit time)
  - Pipelining ***does not*** improve **latency** of a single instruction (amount of time taken for execution)

Ideal Pipeline

- **Assumptions:**
  - Perfectly balanced pipeline stages
    - Each stage takes exactly the same amount of time
  - Instruction utilize the same hardware resource in every pipeline stage
    - Good utilization of hardware resource
  - There are no stalls for dependencies
    - Instructions can always be executed in lock-step fashion
- Optimal Behavior:
  - Complete one instruction every cycle
    - **CPI = 1**
  - Speedup is equal to number of pipeline stages

Real Pipeline

- **Reality:**
  - Stages will not be perfectly balanced
  - Pipelining involves overhead (T_d)
- Let
  - T_i = Execution time of the ith pipeline stage
  - N = Total number of pipeline stages
  - Time without pipelining = $\sum_{i=1}^{N} T_i$
  - Time with pipelining = $\max(T_i) + T_d$
  - **Speedup = N** only if

    T_d=0 and T_i = T_j for all i, j

Limitation Pipeline Depth

- Pipeline overheads
  - Pipeline register delay: setup time etc.
  - **Clock skew:** maximum delay between the clock arrival times at any two registers
- Practical limits on the pipeline depth
  - Once clock cycle is as small as sum of clock skew and latch overhead, no further pipelining is useful

ILLUSTRATIVE EXAMPLE

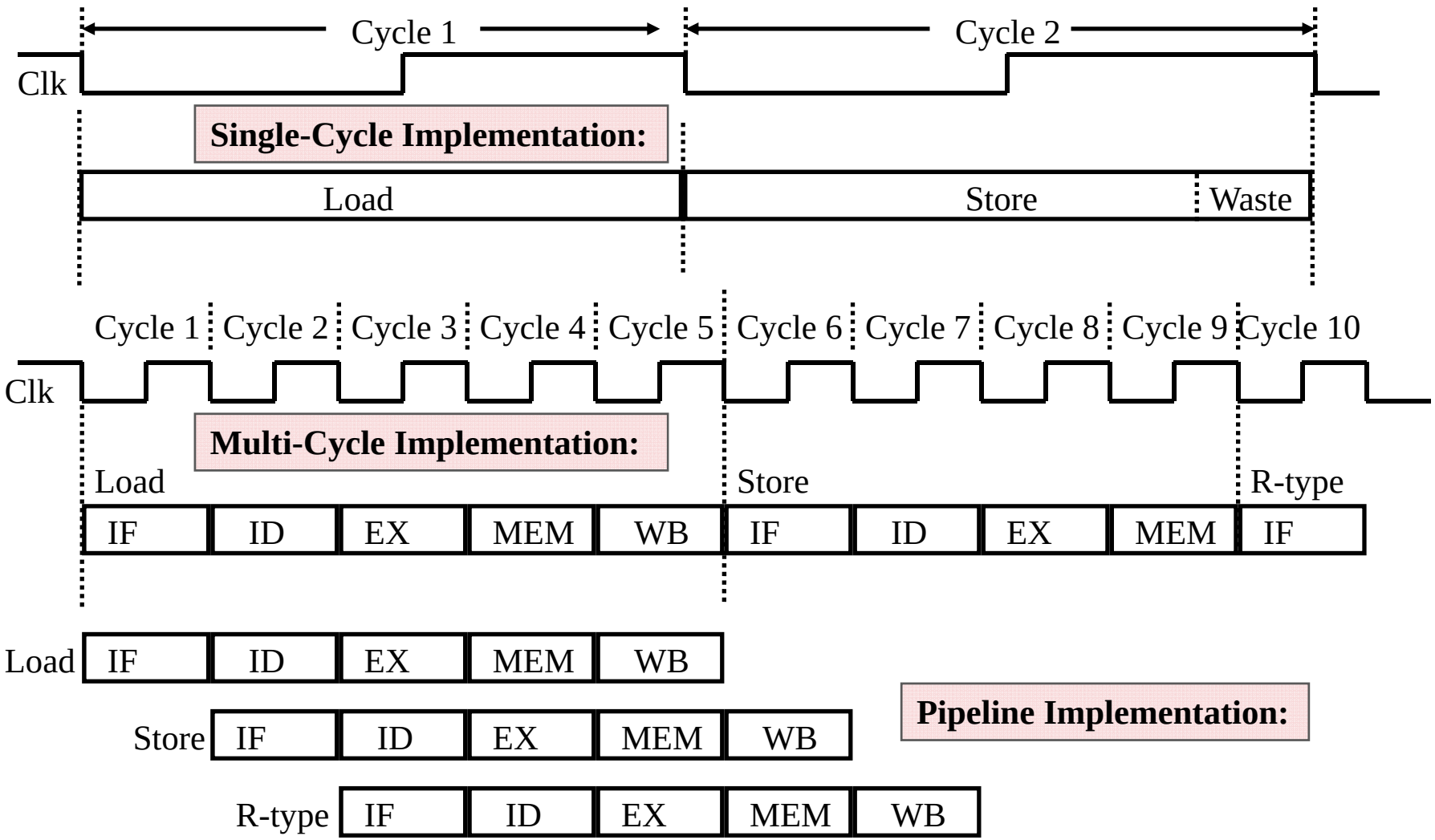
Single-Cycle, Multi-Cycle, Pipeline (1/3)

- Given the following timing information:

Inst	IF	ID	EX	MEM	WB	Total
ALU	2	1	2		1	6
LW	2	1	2	2	1	8
SW	2	1	2	2		7
BEQ	2	1	2			5

- Single-cycle implementation
  - All instructions take as much time as the slowest one
  - Cycle time = 8ns
- Multi-cycle and pipelined implementation
  - Cycle time = 2ns

Single-Cycle, Multi-Cycle, Pipeline (2/3)



Single-Cycle, Multi-Cycle, Pipeline (3/3)

- Suppose we execute 100 instructions
- Single-Cycle Machine
  - 8 ns/cycle x 1 CPI x 100 inst = 800 ns
- Multi-cycle Machine
  - 2 ns/cycle x 4.6 CPI x 100 inst = 920 ns
- Ideal pipelined machine
  - 2 ns/cycle x (1 CPI x 100 inst + 4 cycle pipeline fill) = 208 ns

PIPELINE HAZARDS

Pipeline Performance: **Amdahl's Law**

- Performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used

$$Speedup = \frac{Time_{old}}{Time_{new}}$$

$$Time_{new} = Time_{old} \times \left((1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right)$$

$$Speedup = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

Corollaries

- **Corollary 1:**
  - If the enhancement is only applicable for a fraction of a task, we cannot speed up the task by more than

$$\frac{1}{1 - Fraction_{enhanced}}$$

- **Corollary 2:**
  - Make the common case first (In making a design trade-off, favor the frequent case over the infrequent case)

Pipeline performance (1/2)

$$Speedup = \frac{Avg\ instr\ time\ unpipelined}{Avg\ instr\ time\ pipelined}$$

$$= \frac{CPI\ unpipelined}{CPI\ pipelined} \times \frac{clock\ period\ unpipelined}{clock\ period\ pipelined}$$

- If we assume perfectly balanced pipeline

$$Speedup = \frac{CPI\ unpipelined}{CPI\ pipelined}$$

- If all instructions take equal number of cycles in multi-cycle implementation

$$CPI\ unpipelined = Pipeline\ depth$$

Pipeline performance (2/2)

$$CPI\ \ pipelined = Ideal\ CPI + stall\ cycles\ \ per\ instr$$
$$= 1 + stall\ cycles\ \ per\ instr$$

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

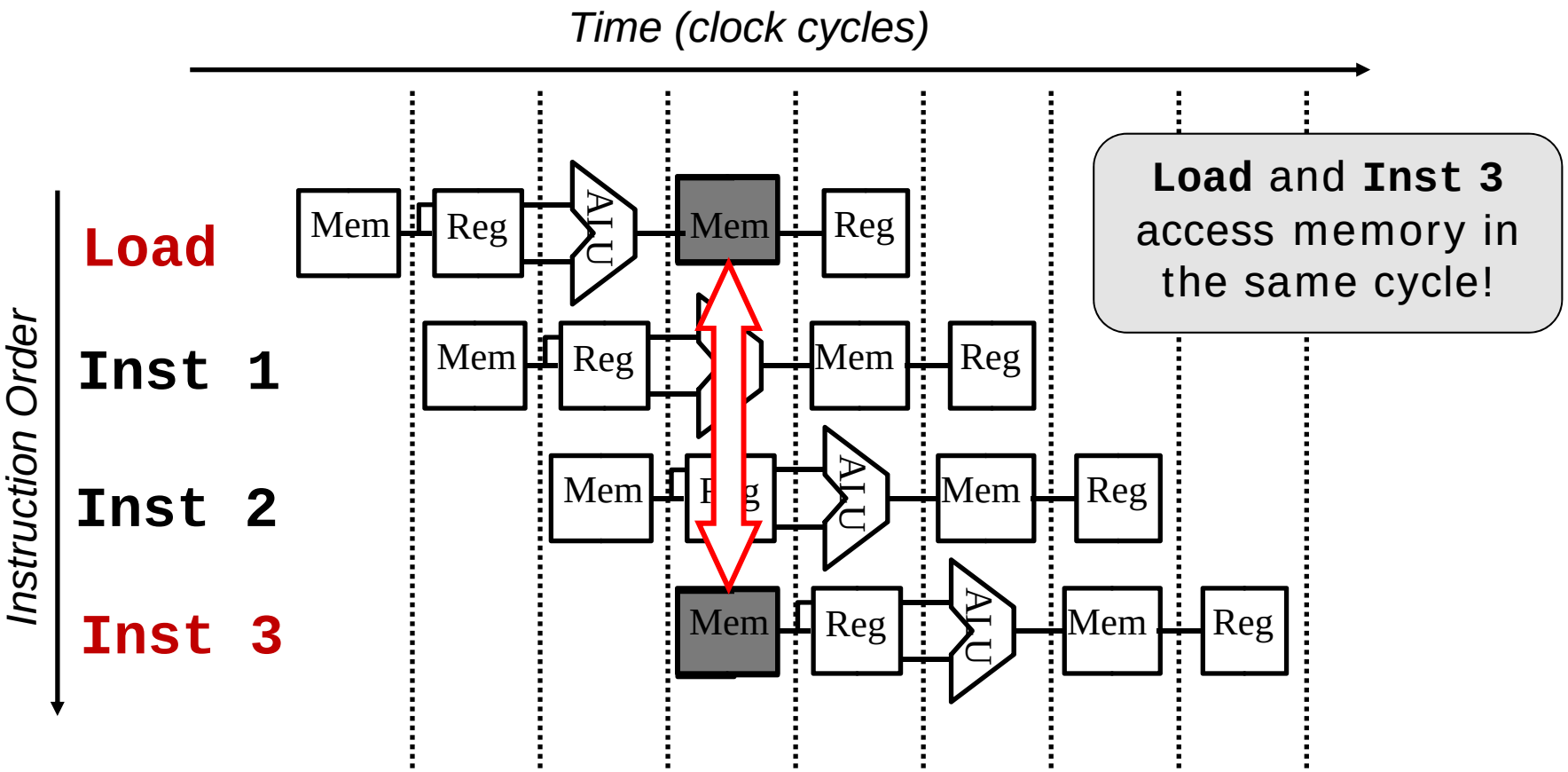
- If there are no pipeline stalls, pipelining can improve performance by the depth of the pipeline

Pipeline Hazards

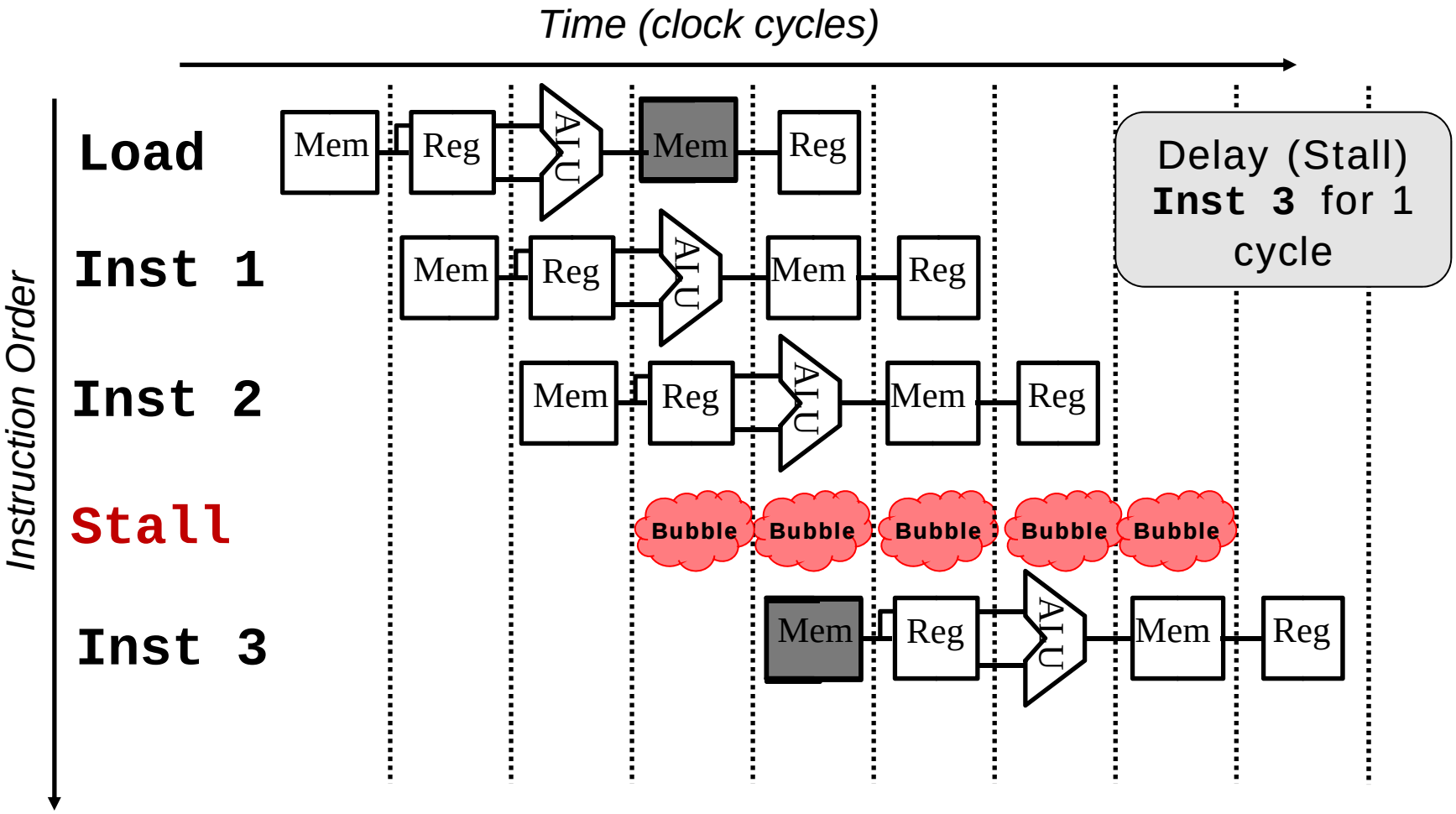
- There are a number of **pipeline hazards** which prevent safe parallel execution of instructions:
  - **Structural hazards**
    - Simultaneous use of a hardware resource
  - **Data hazards**
    - Data dependencies between instructions
  - **Control hazards**
    - Change in program flow

Structural Hazard: Example

- Suppose we have a single memory module:

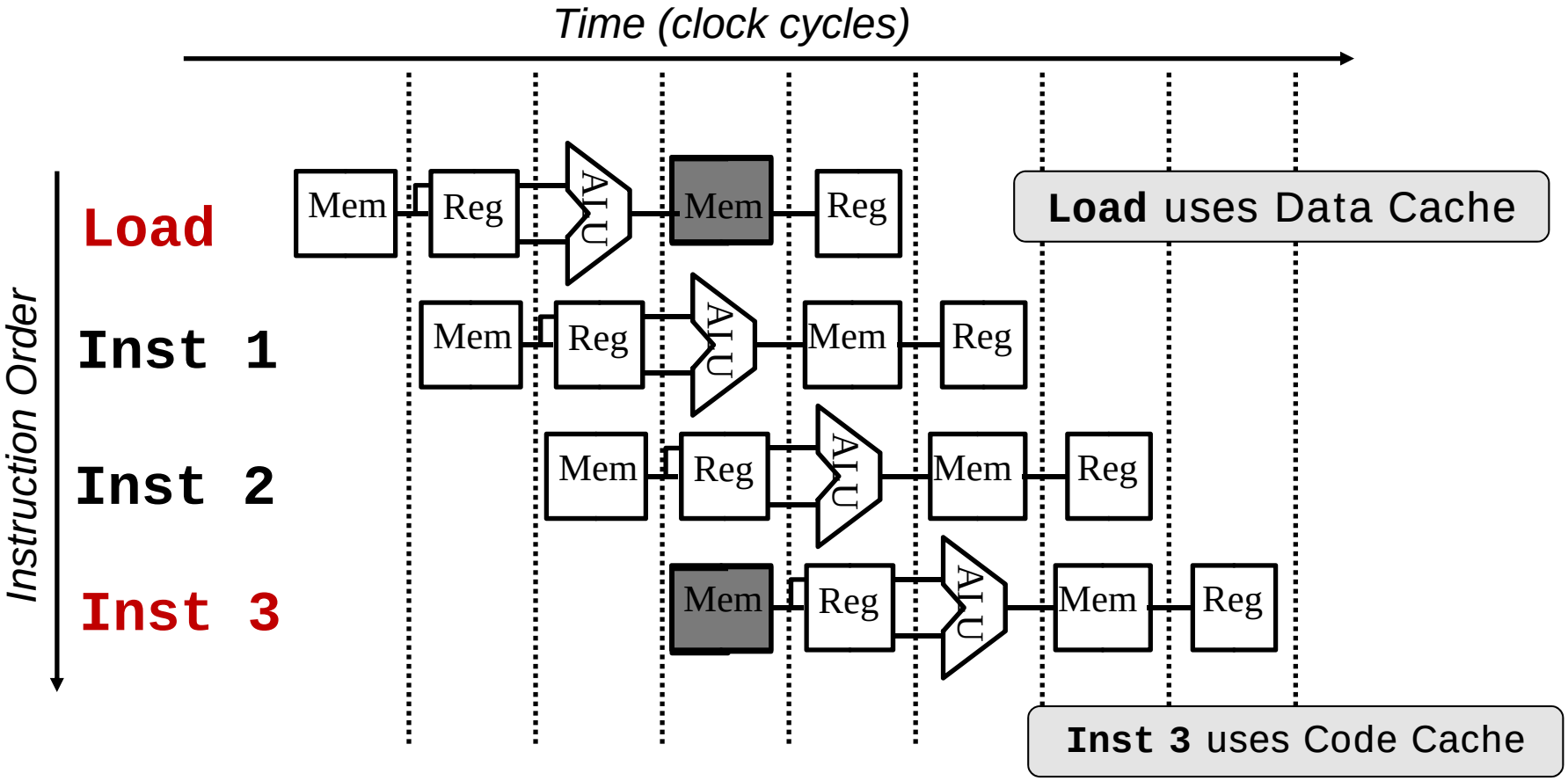


Structural Hazard: Stall as Solution



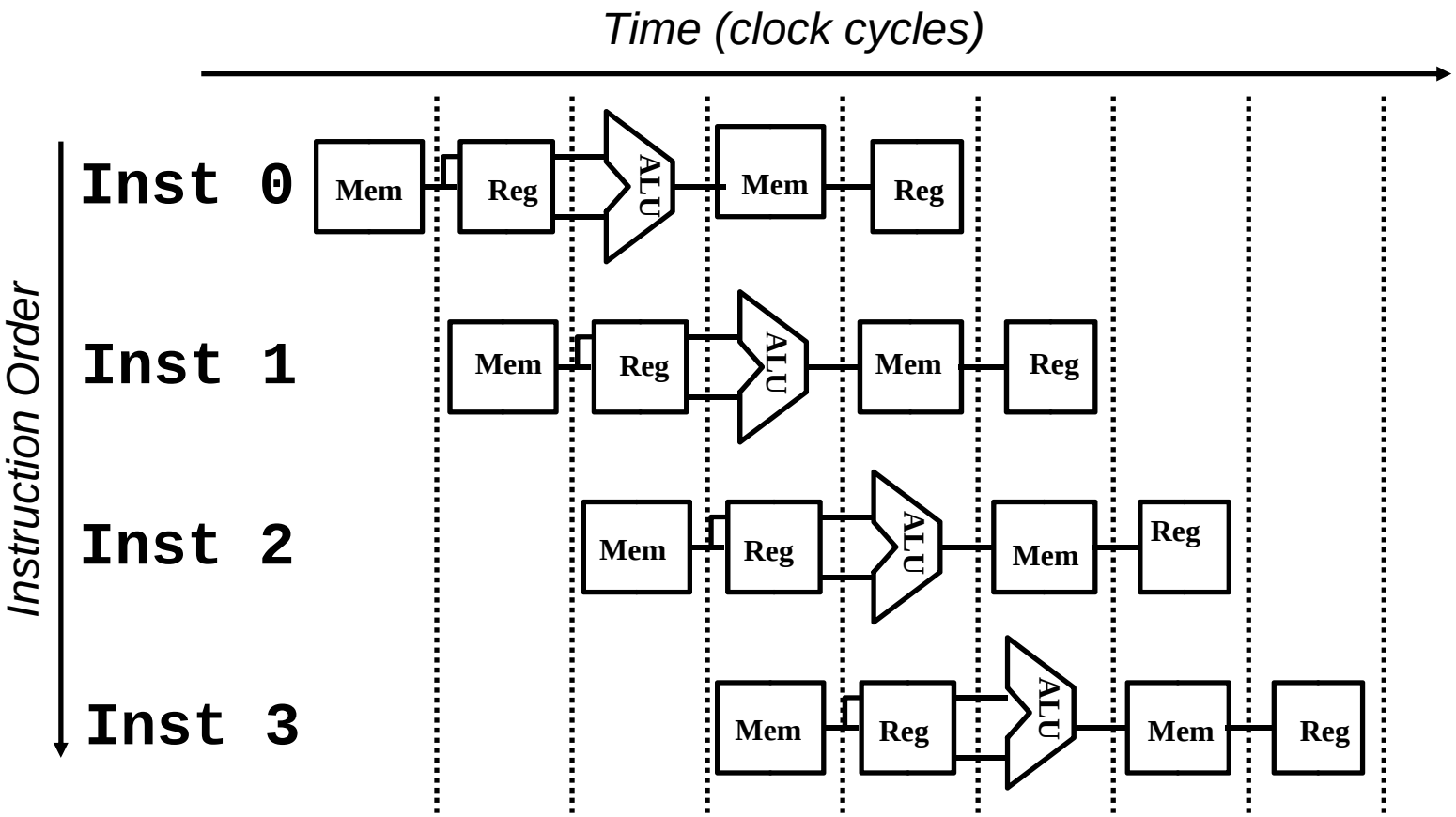
Structural Hazard: Alternative Solution

- Better Solution:
  - Split cache memory into: **Data and Code Cache**



Quiz: What about this?

- Identify another resource conflict and its solution:



Pipeline Hazards

INSTRUCTION DEPENDENCY

Instruction Dependency

- Instructions can have relationship that prevent parallel execution:
  - Although a partial overlap maybe possible in some cases
- When different instructions read or write from the same register
  - Register contention is the cause of dependency
  - Known as **data dependency**
- When the execution of an instruction depends on another instruction
  - Control flow is the cause of dependency
  - Known as **control dependency**
- Failure to handle dependencies can affect **program correctness**!

Program Correctness

- If a program consisting of a sequence $i_0, i_1, i_2, \ldots i_{n-1}$ instructions is executed sequentially
  - Then, any instruction i_t is not executed until i_{t-1} has executed to completion
- If several instructions are executed concurrently (e.g. in a pipeline), the outcome must be exactly the same as the case of sequential execution
  - A different outcome is **WRONG**!
- Let us find out how different types of dependency affect the program correctness

Data Dependency: **R**ead-**A**fter-**W**rite

- **Definition:**
  - Occurs when a later instruction **reads** from the destination register **written** by an earlier instruction
  - aka **true dependency** or RAW dependency

- Example:

```
i1: add $1, $2, $3 ;writes to $1
i2: mul $4, $1, $5 ;reads from $1
```

- Effect of incorrect execution:
  - If **i2** reads register **$1** before **i1** can write back the result, **i2** will get a ***stale result (old result)***

Other Data Dependencies

- Similarly, we have:
  - **WAR**: **W**rite-**a**fter-**R**ead dependency
  - **WAW**: **W**rite-**a**fter-**W**rite dependency

- Fortunately, these dependencies **do not cause any pipeline hazards**

- They affects the processor only when instructions are executed out of program order:
  - i.e. in Modern SuperScalar Processor

Dependencies: **Detect** and **Resolution**

- True data dependency and control dependency are unavoidable:
  - A pipeline design must be equipped to handle these hazards properly

- The main tasks are:
  - **Detection:**
    - Monitor the instructions in the pipeline to detect any dependency
  - **Resolution:**
    - Look for ways to ensure program correctness and reduce overhead

Data Hazards: Example

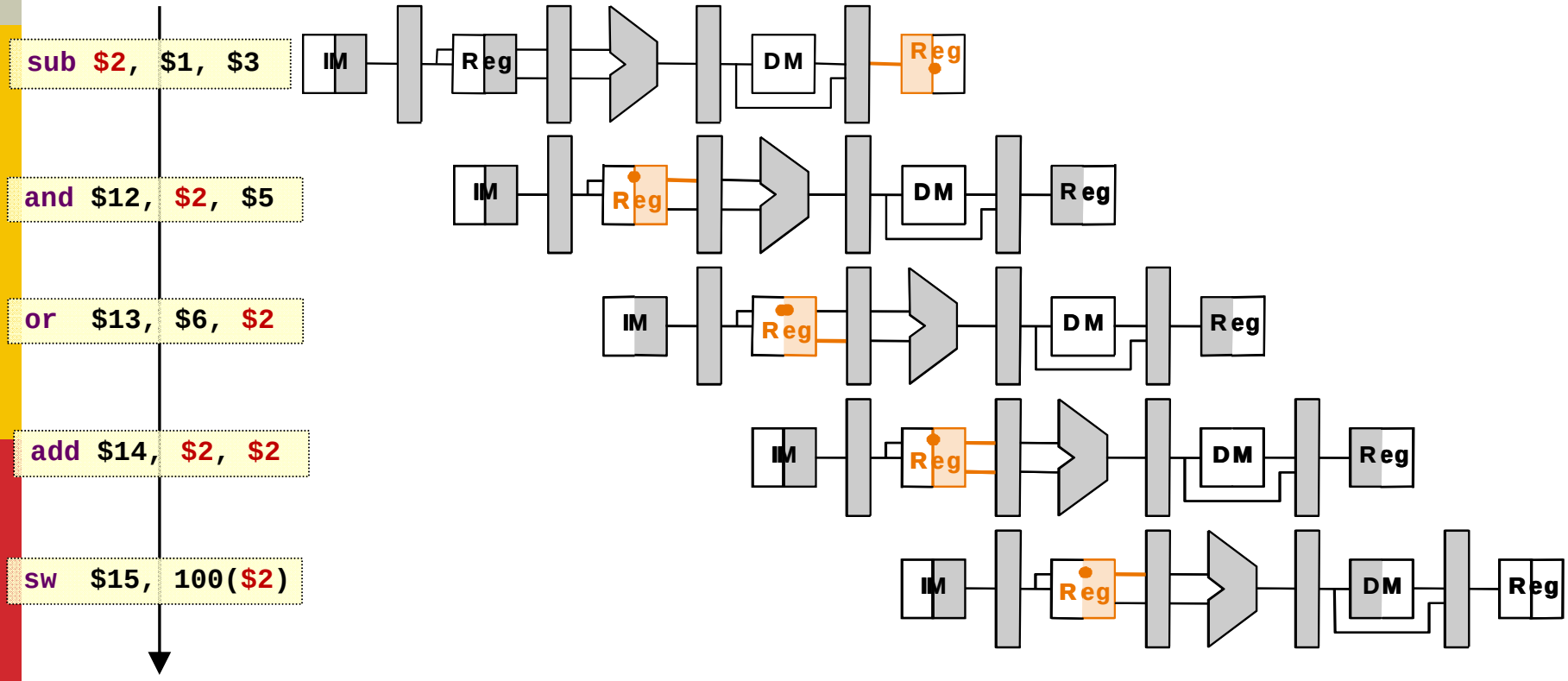
- Observe the following code fragment:

```
sub $2, $1, $3      #i1
and $12, $2, $5     #i2
or  $13, $6, $2     #i3
add $14, $2, $2     #i4
sw  $15, 100($2)    #i5
```

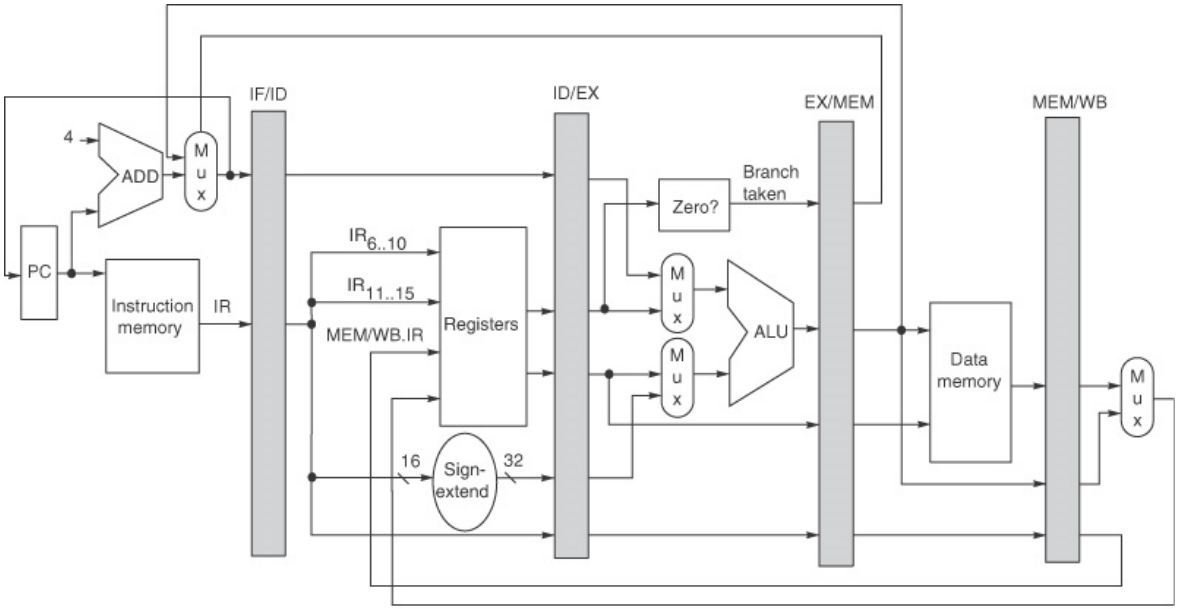
- Note the multiple uses of register **$2**
- Question:
  - Which are the instructions require special handling?

RAW Data Hazards: **Illustration**

CC	1	2	3	4	5	6	7	8	9
[$2]	10	10	10	10	10/-20	-20	-20	-20	-20



Quiz: How to detect RAW hazard?



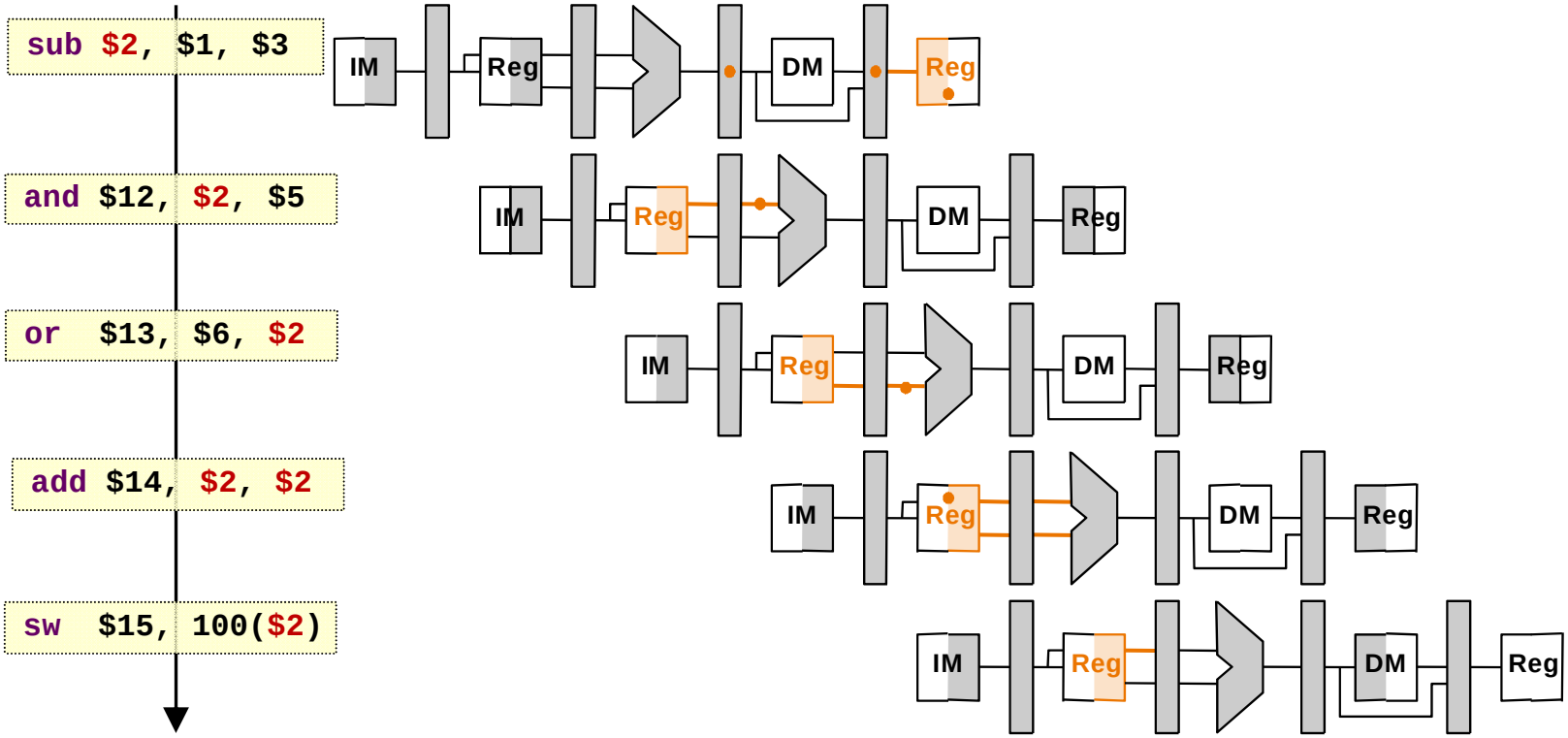
- Recall that:
  - Each latch stores information to be used in execution until it is needed
- Give the "formula/condition" to check for RAW hazard by referring to fields in the latches

Data Hazards: **Resolution**

- We now know how to detect RAW hazard:
  - What is the solution?
- Simplest solution:
  - Just stall the offending instruction until the condition turned false
    - Undesirable due to the penalty
  - Is there more efficient solution?
- Hint: When is the result produced and when is the result needed in RAW hazard?

Data Forwarding: **Illustration**

CC	1	2	3	4	5	6	7	8	9
[$2]	10	10	10	10	10/-20	-20	-20	-20	-20
EX/MEM	X	X	X	-20	X	X	X	X	X
MEM/WB	X	X	X	X	-20	X	X	X	X



Data Hazards: Example 2

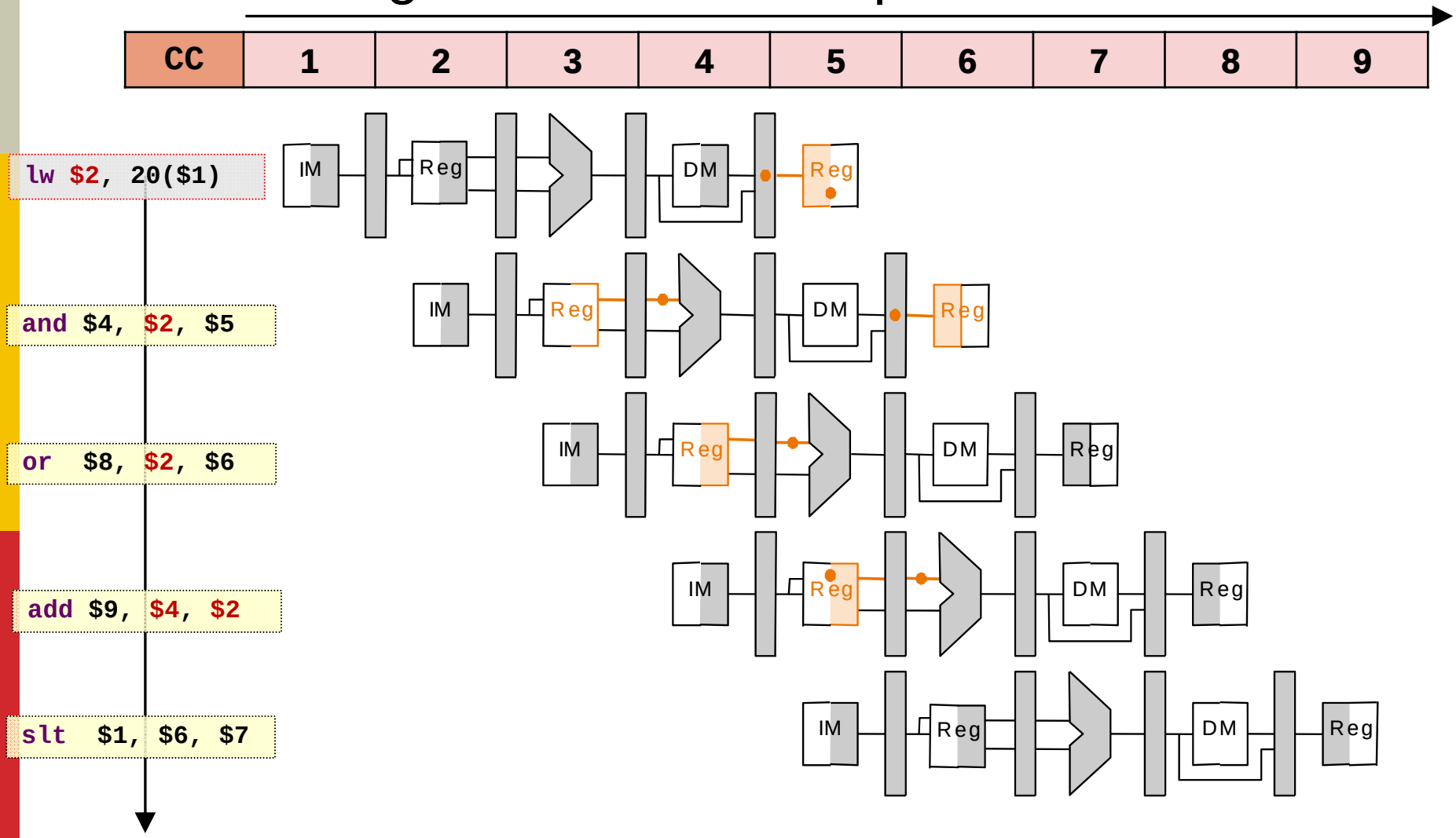
- Let us try another code fragment:

```
lw   $2, 20($1)     #i1
and  $4, $2, $5     #i2
or   $8, $6, $2     #i3
add  $9, $4, $2     #i4
```

- Note the multiple uses of register **$2**
- **Question:**
  - With the forwarding paths discussed, can we execute this code without stalling?

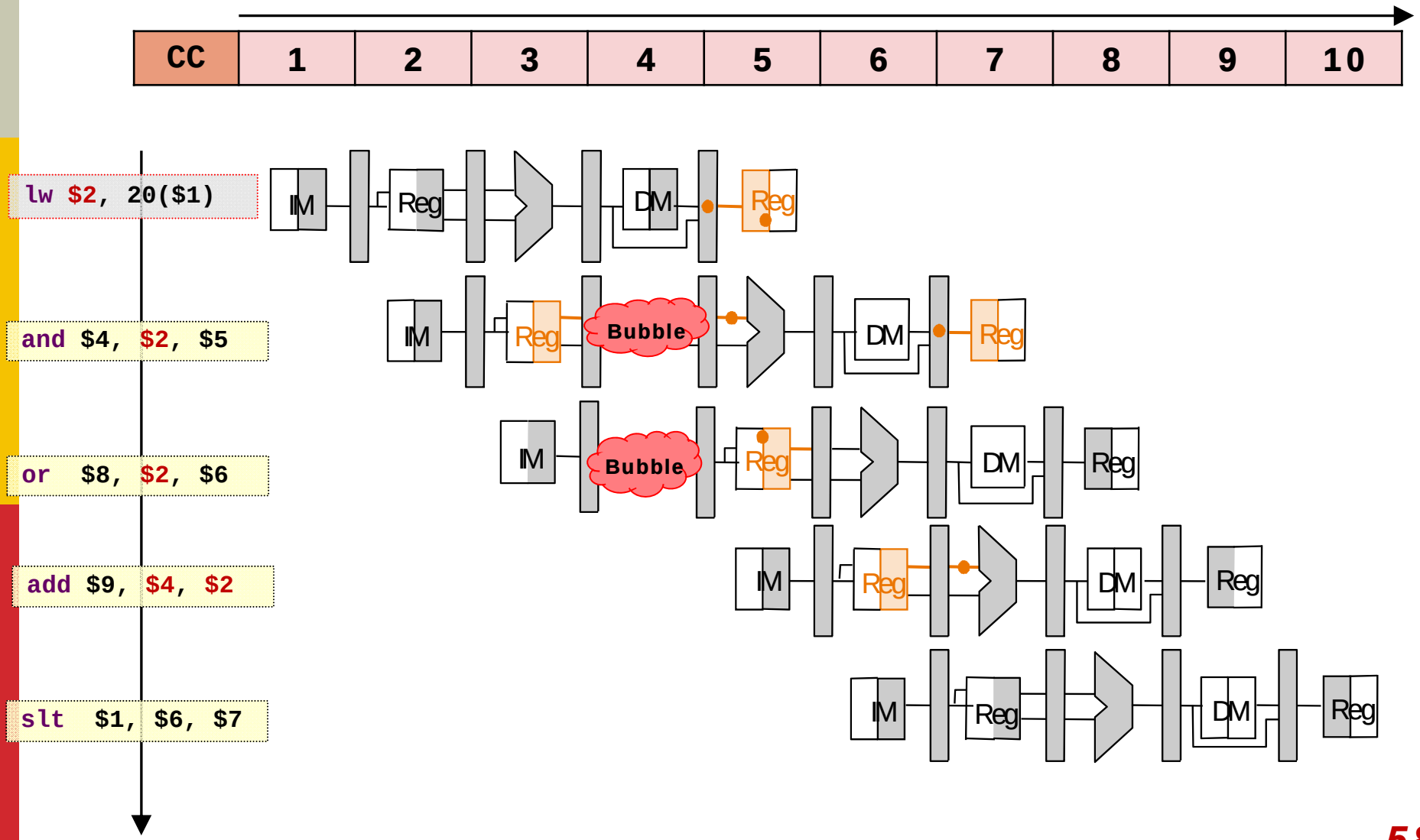
Data Hazards: Problem with **LW**

- Forwarding fails to solve the problem!



Data Hazards: Solution for **LW**

- We have to **stall the pipeline for 1 cycle**



Control Dependency

Control Dependency

- **Definition:**
  - An instruction **`j`** is control dependent on **`i`** if **`i`** controls whether or not **`j`** executes
  - Typically **`i`** would be a branch instruction

- Example:

```
i1: blt there, $0, $1        ; Branch
i2: add $0, $1, $2    ; depends on i1
...    ...    ...
```

- Effect of incorrect execution:
  - If **`i2`** is allowed to execute before **`i1`** is determined, register **`$0`** maybe incorrectly changed!

Control Hazard: Example

r1 ≠ r3

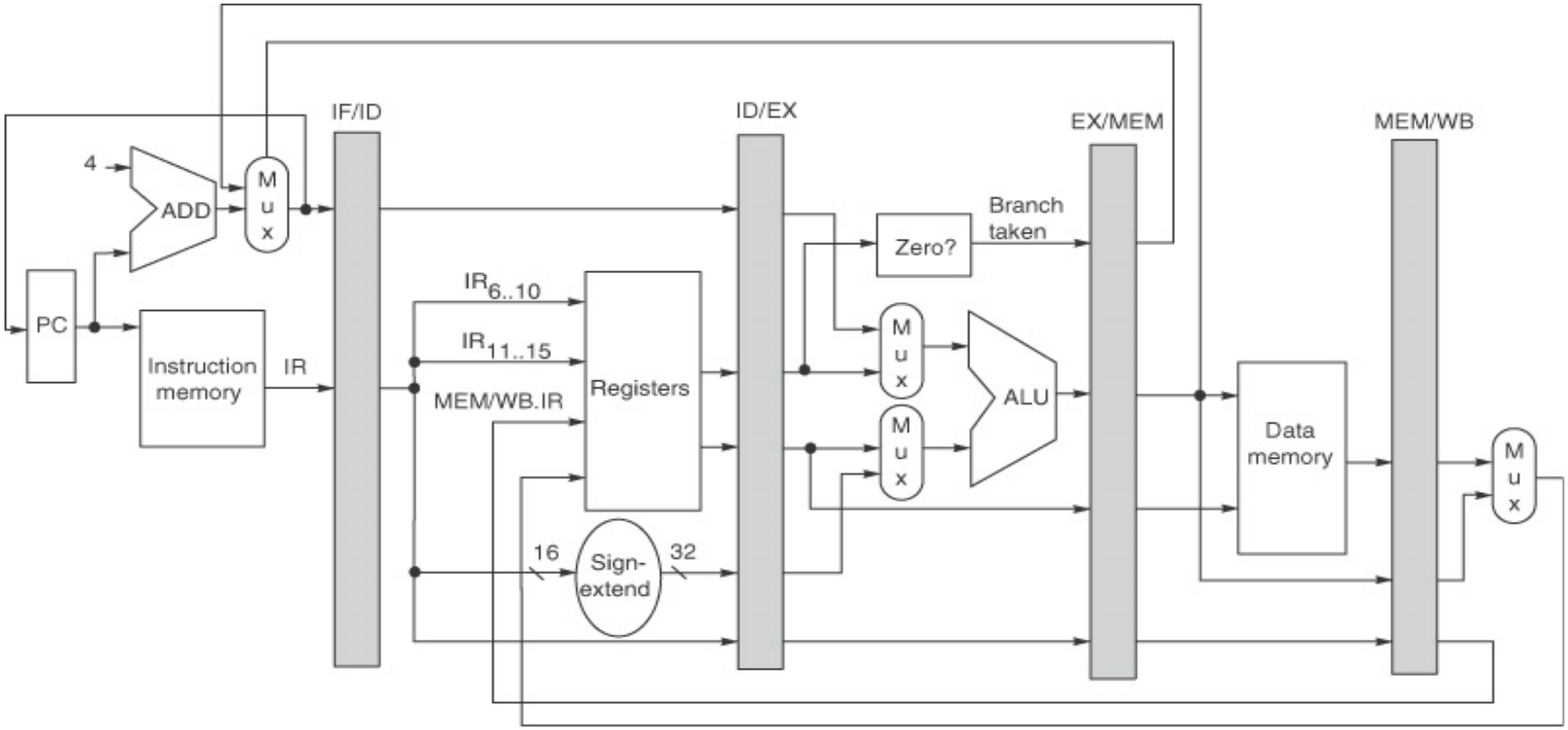
```
40  beq $1,  $3, 7
44  and $12, $2, $5
48  or  $13, $6, $2
52  add $14, $2, $2
..  ... ... ... ...
72  lw  $4,  5($7)
```

r1 = r3

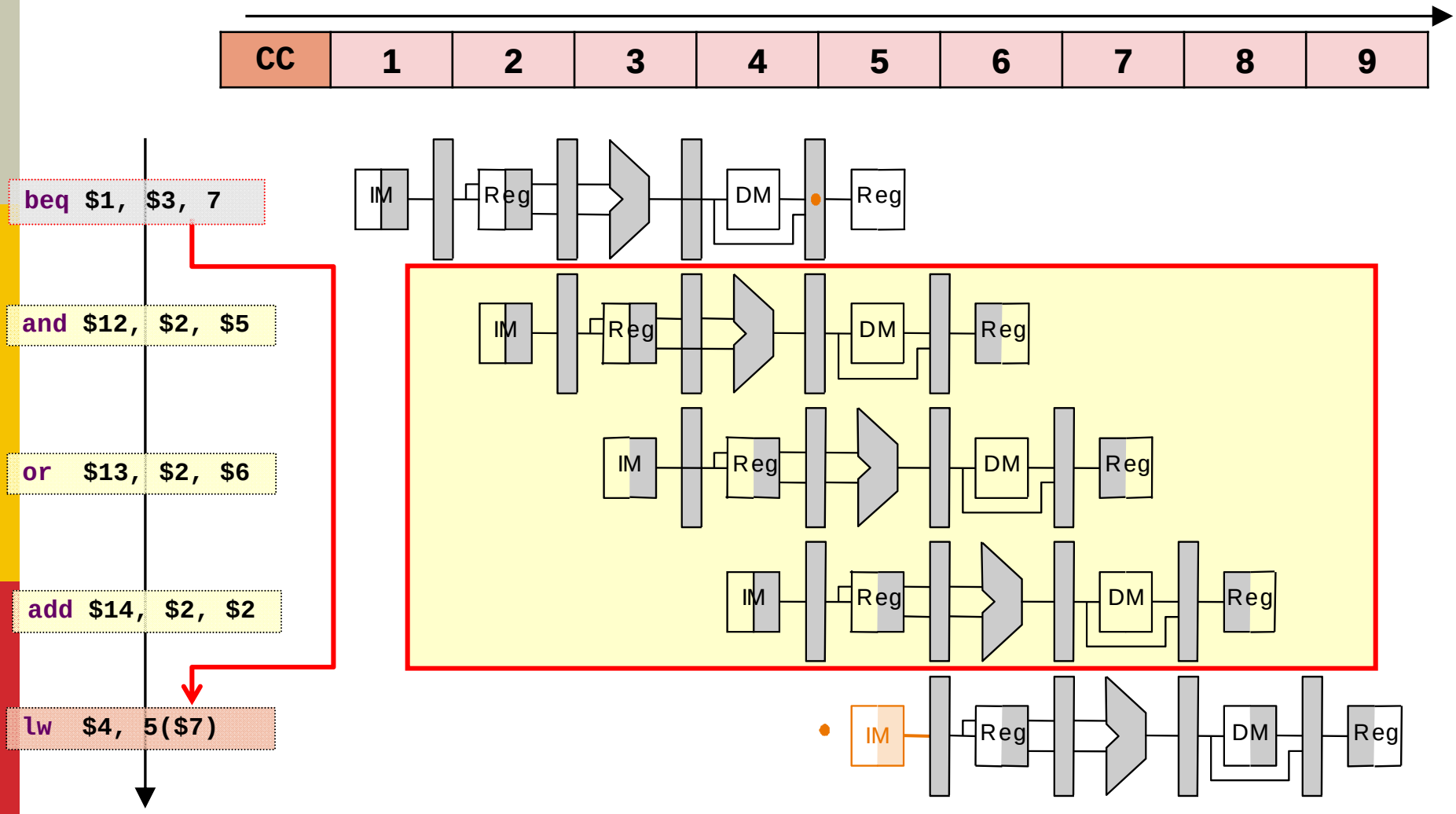
- How does the code affect a pipeline processor?

Control Hazard: Problem

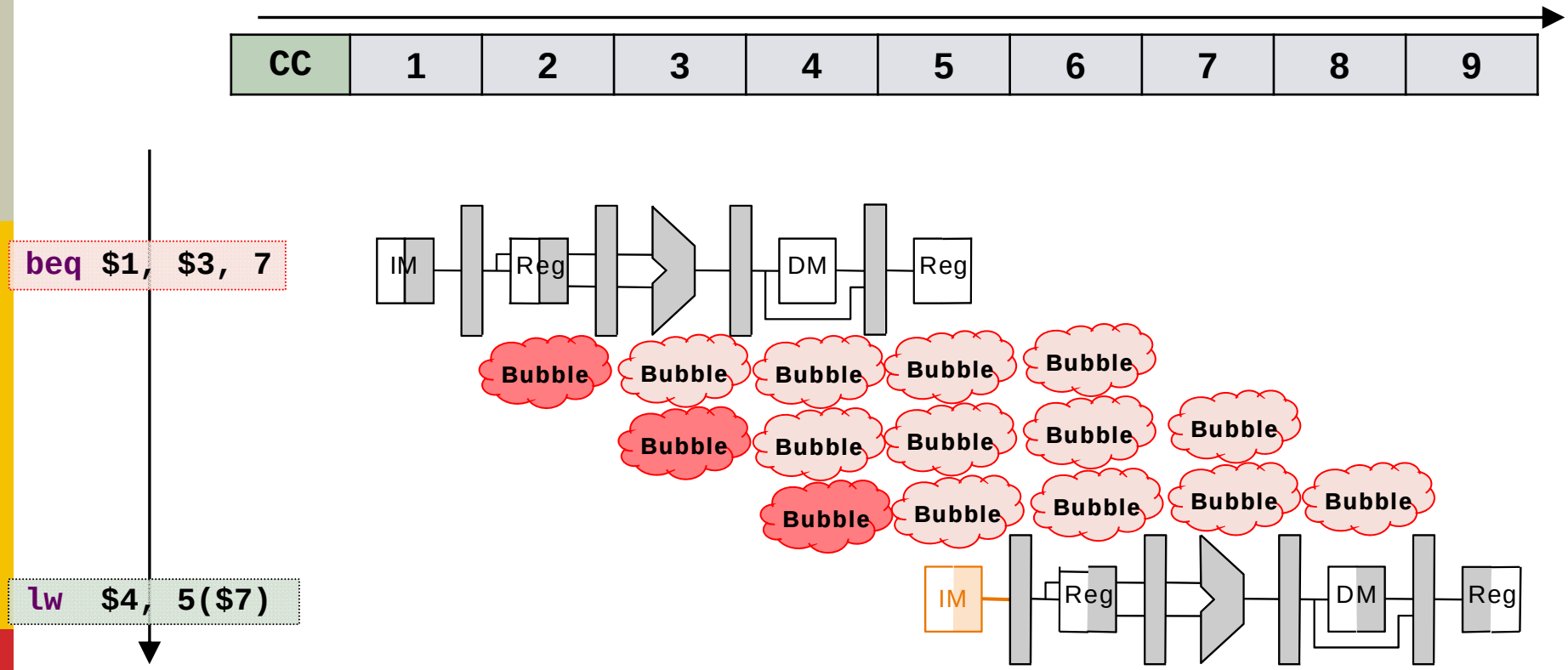
- Branch outcome is known only in MEM stage:
  - **Too late:** subsequent instructions already went into the pipeline!



Control Hazards: Example



Control Hazards: Stall Pipeline



- Wait for the branch outcome and then fetch the correct instructions
  - Introduces **3 clock cycles delay**

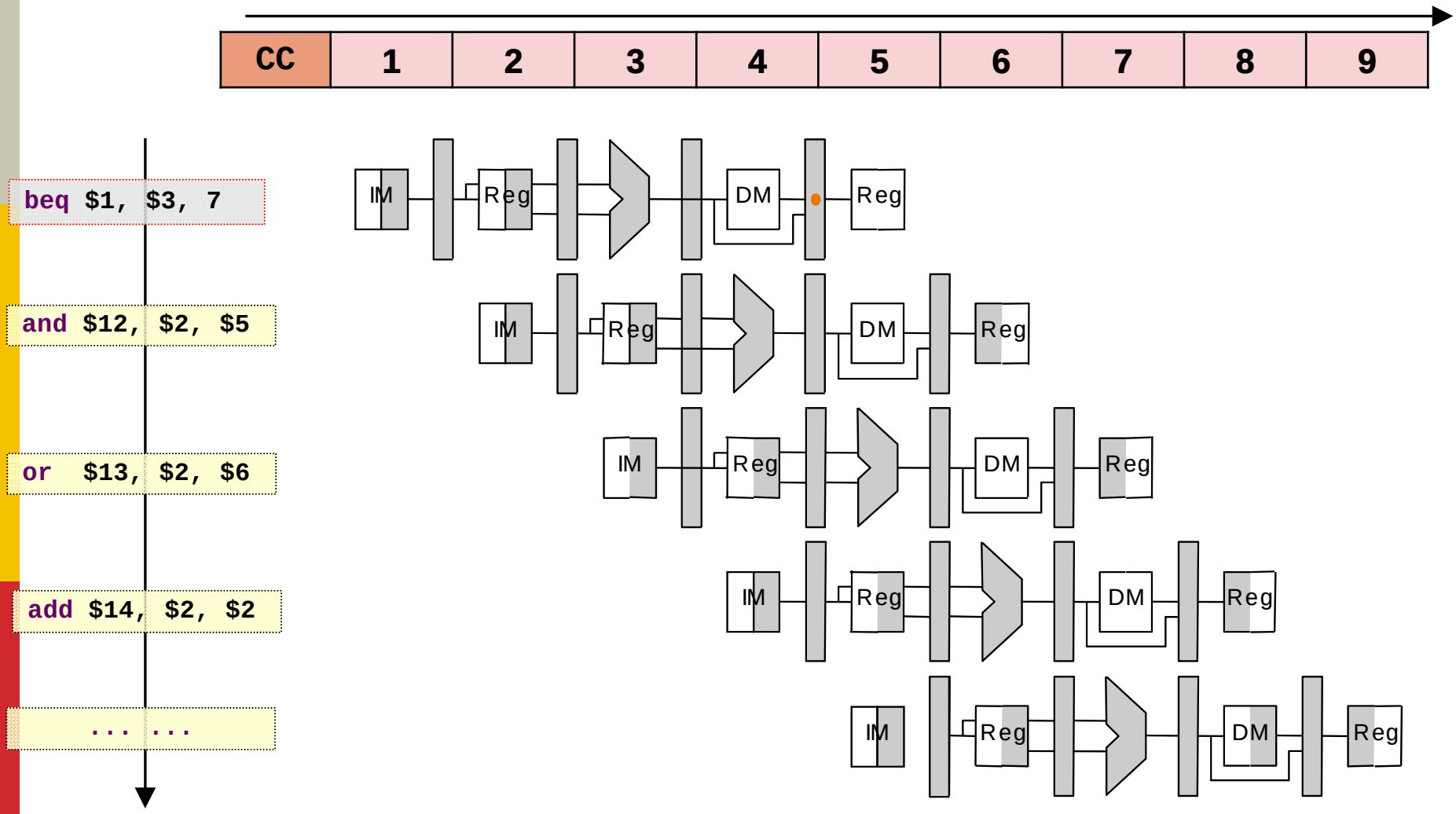
Control Hazard: **Branch Prediction**

- Instead of waiting for the branch outcome
  - Predict (Guess) the outcome right away

- Simple Prediction:
  - Predict all branches are **not taken**
  - ➔ Fetch and execute the fall through path

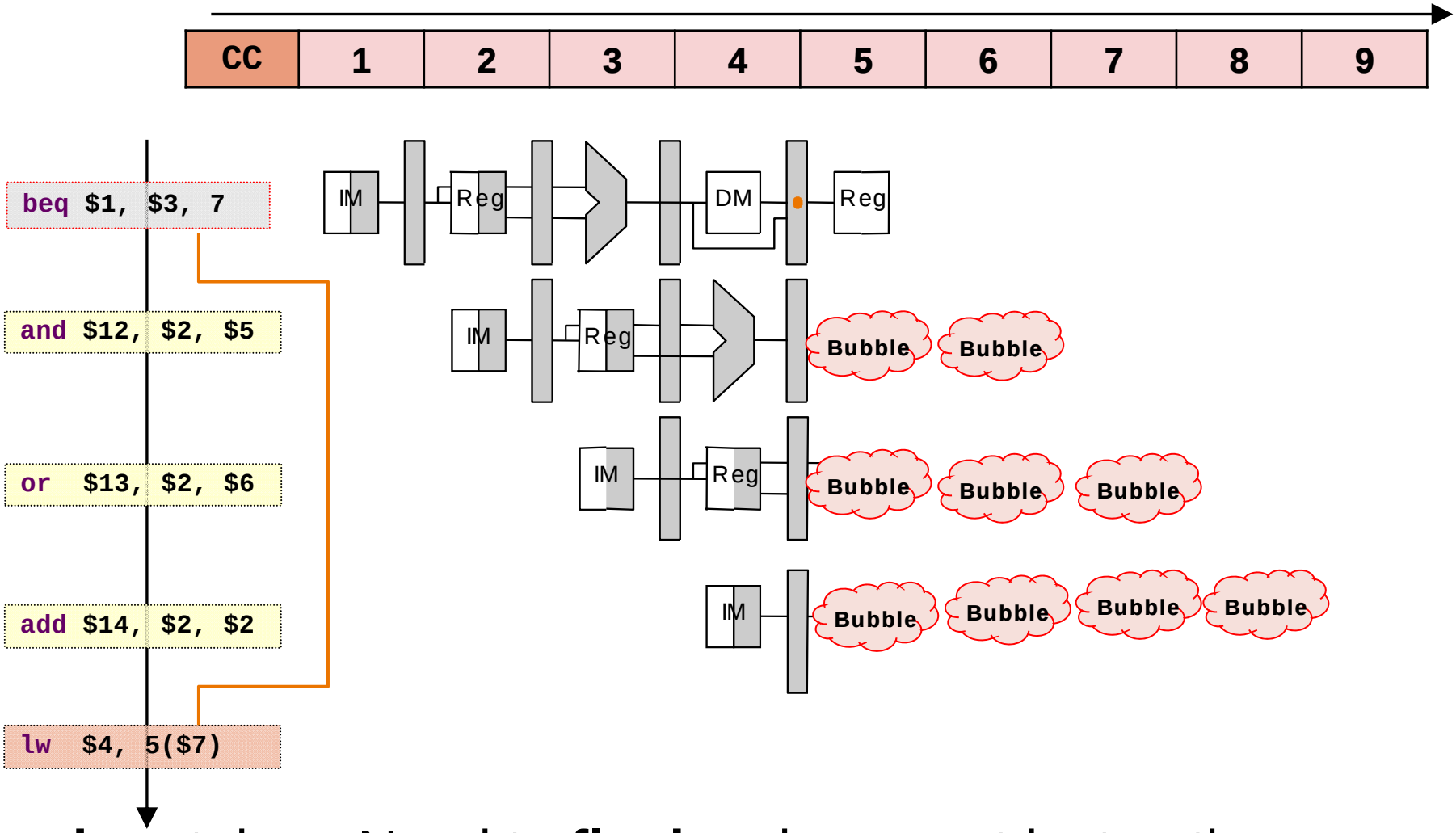
  - When outcome is known:
    - **Correct prediction**: No need to do anything
    - **Wrong prediction**: Flush wrong instructions from the pipeline

Branch Prediction: **Correct Prediction**



- **beq** not taken: Fall through path already in pipeline!

Branch Prediction: **Wrong Prediction**



- **beq** taken: Need to **flush** subsequent instructions and then fetch from the right location

Summary

- Pipelining is the fundamental principle of modern processor architecture design

- Major topics:
  - Ideal pipeline
  - Hazards:
    - Data dependency
    - Control dependency
  - Hazard detection and resolution