

CS2100

COMPUTER ORGANISATION

<http://www.comp.nus.edu.sg/~cs2100/>

## Lecture #21

---

# Pipelining

## Part II: Hazards



**NUS**  
National University  
of Singapore

School of  
Computing

# Lecture #21: Pipelining II

## 1. Pipeline Hazards

## 2. Structural Hazards

Memory: Split instruction memory and data memory

Register R/W: Half a cycle - W then R

## 3. Instruction Dependencies

## 4. Data Hazards

### 4.1 Forwarding

### 4.2 Stall

### 4.3 Exercises

# Lecture #21: Pipelining II

5. Control Dependency

6. Control Hazards

6.1 Early Branch

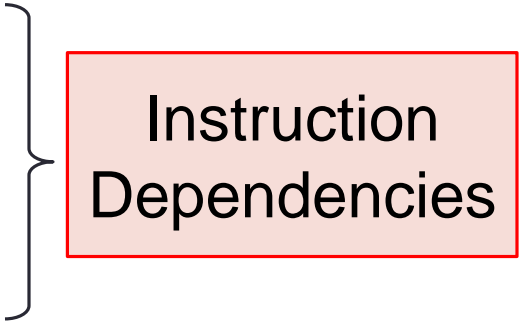
6.2 Branch Prediction

6.3 Delayed Branched

7. Multiple Issue Processors (reading)

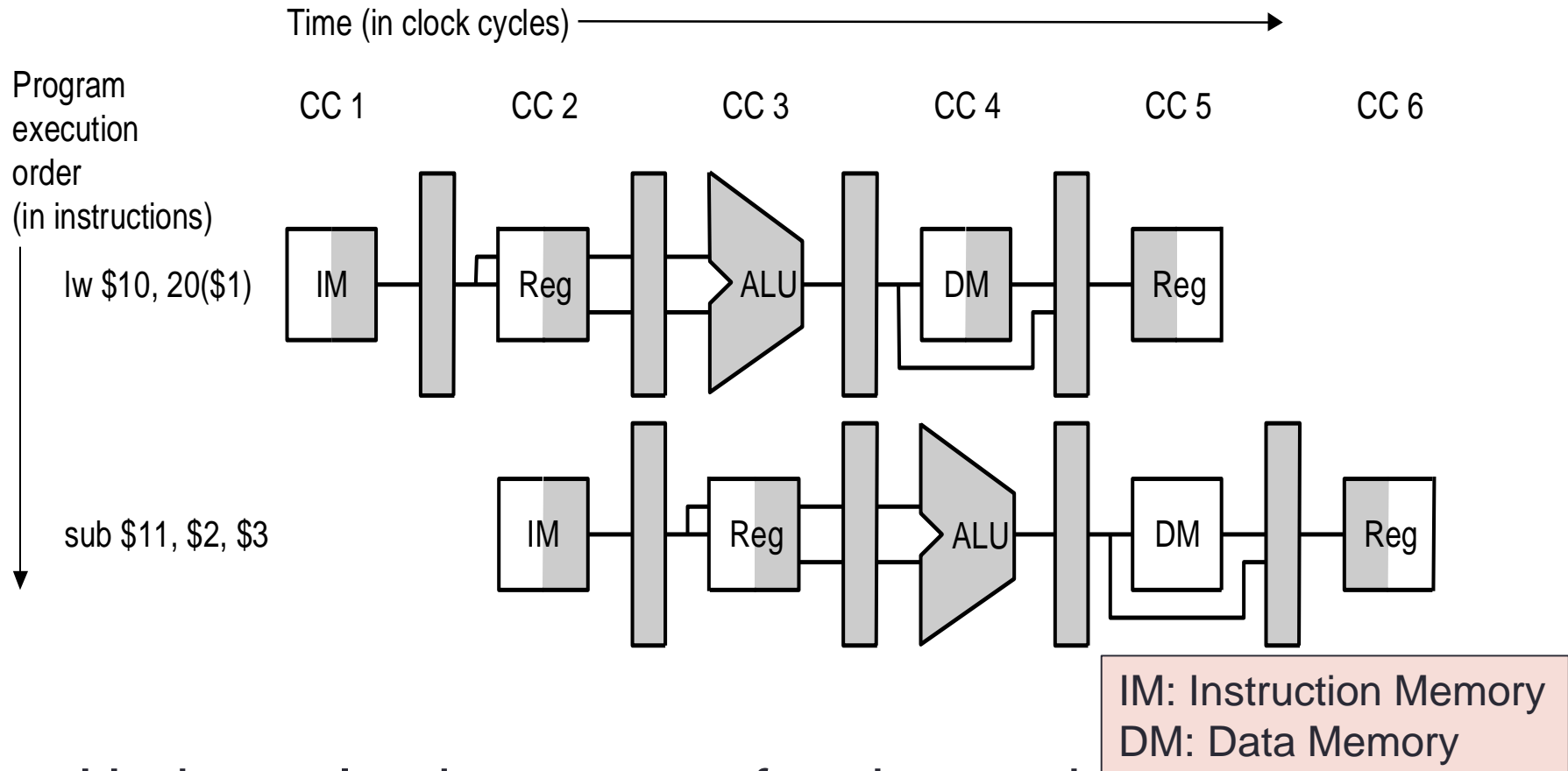
# 1. Pipeline Hazards

- Speedup from pipeline implementation:
  - Based on the assumption that a new instructions can be "pumped" into pipeline every cycle
- However, there are **pipeline hazards**
  - Problems that prevent next instruction from immediately following previous instruction
  - **Structural hazards:**
    - Simultaneous use of a hardware resource
  - **Data hazards:**
    - Data dependencies between instructions
  - **Control hazards:**
    - Change in program flow



Instruction  
Dependencies

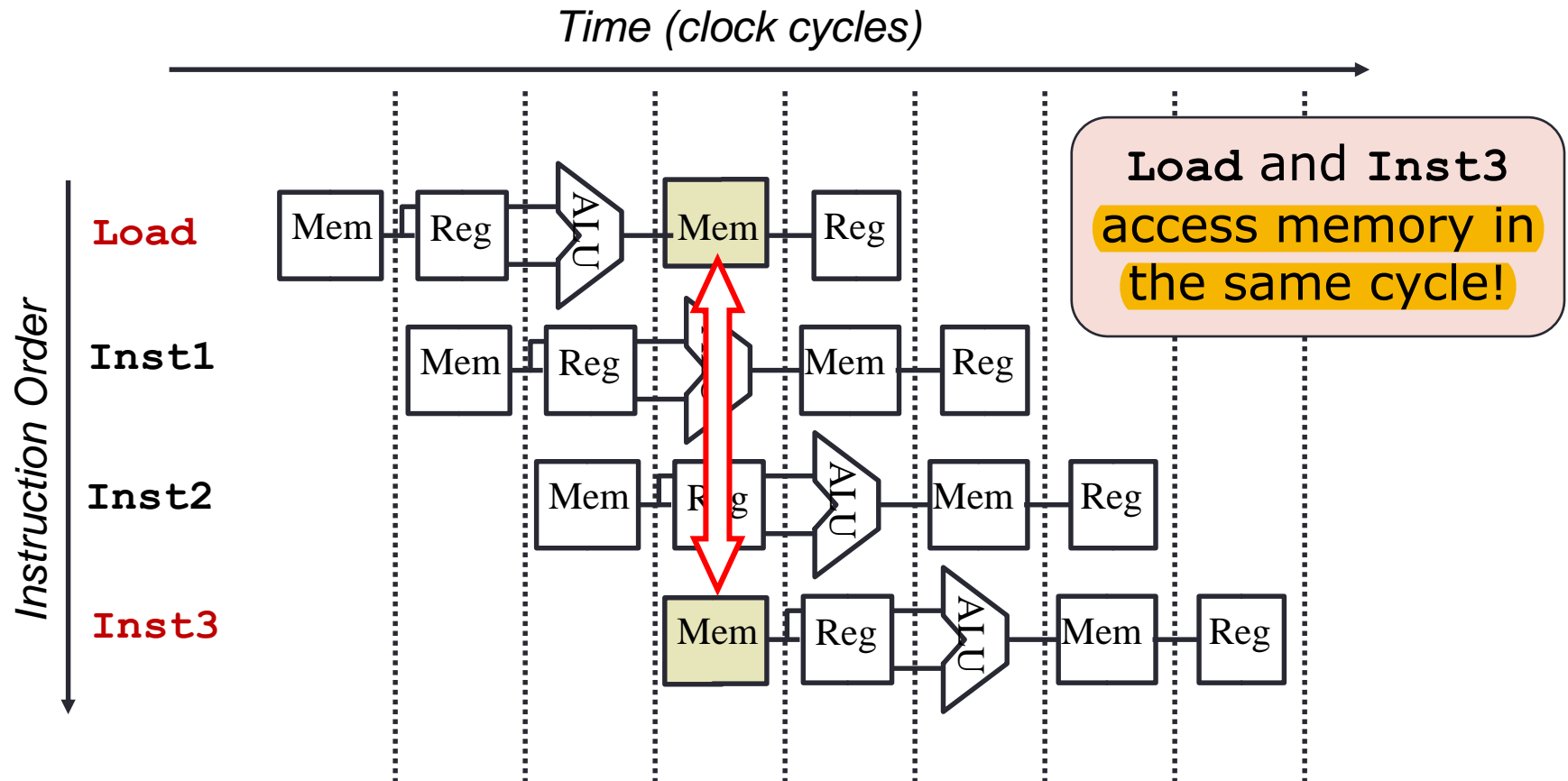
# 1. Graphical Notation for Pipeline



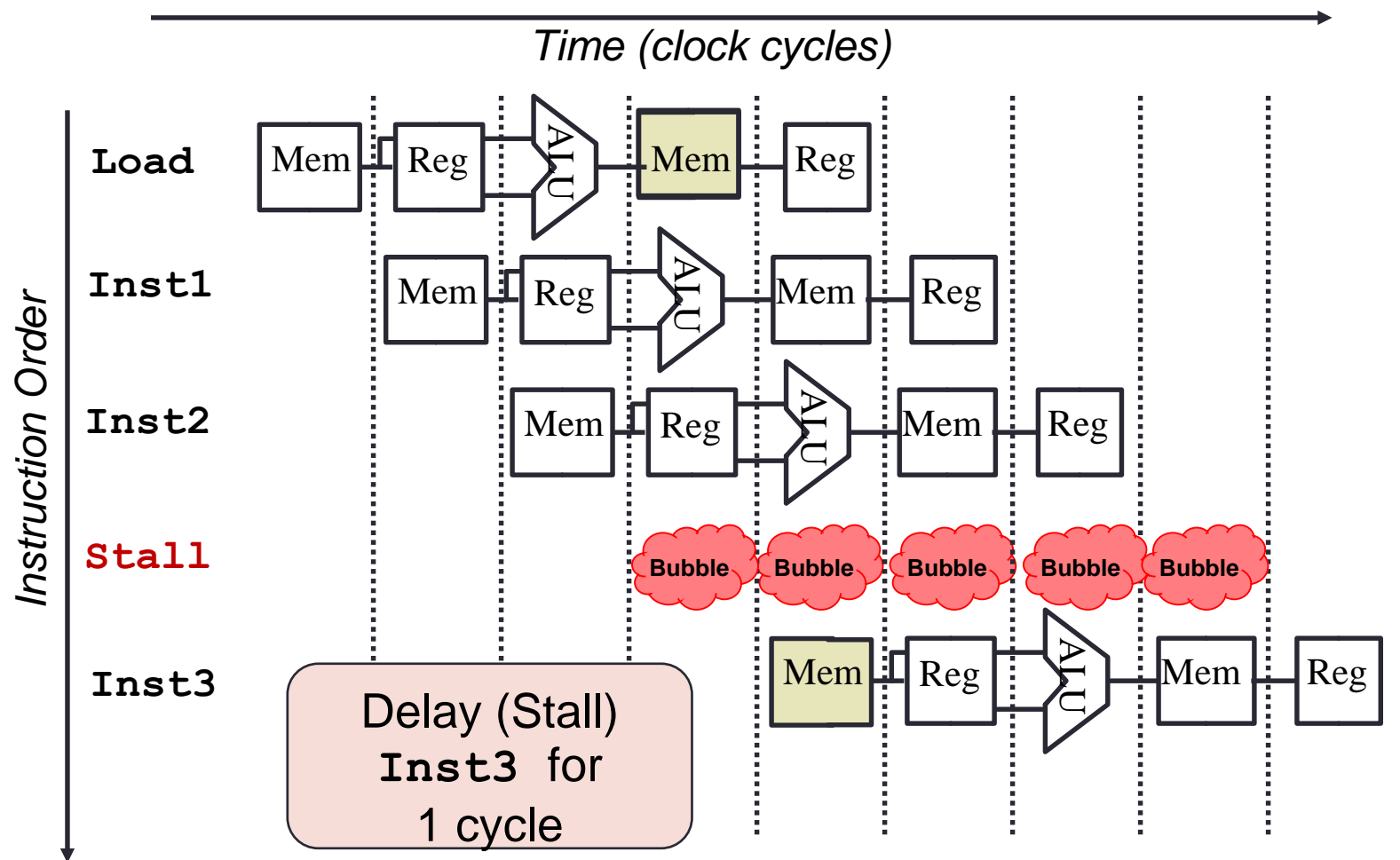
- Horizontal = the stages of an instruction
- Vertical = the instructions in different pipeline stages

## 2. Structural Hazard: Example

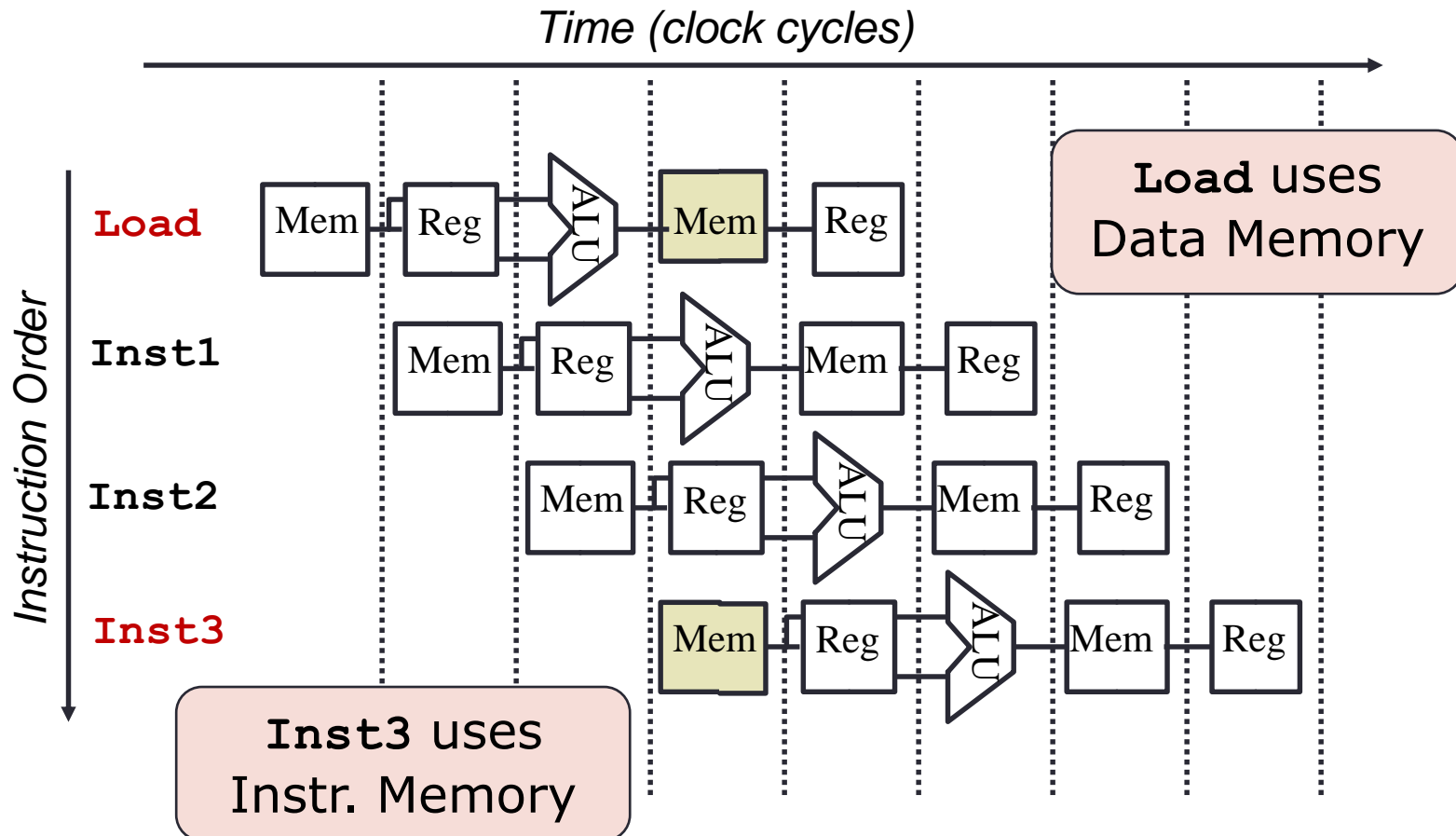
- If there is only a **single memory module**:



## 2. Solution 1: Stall the Pipeline



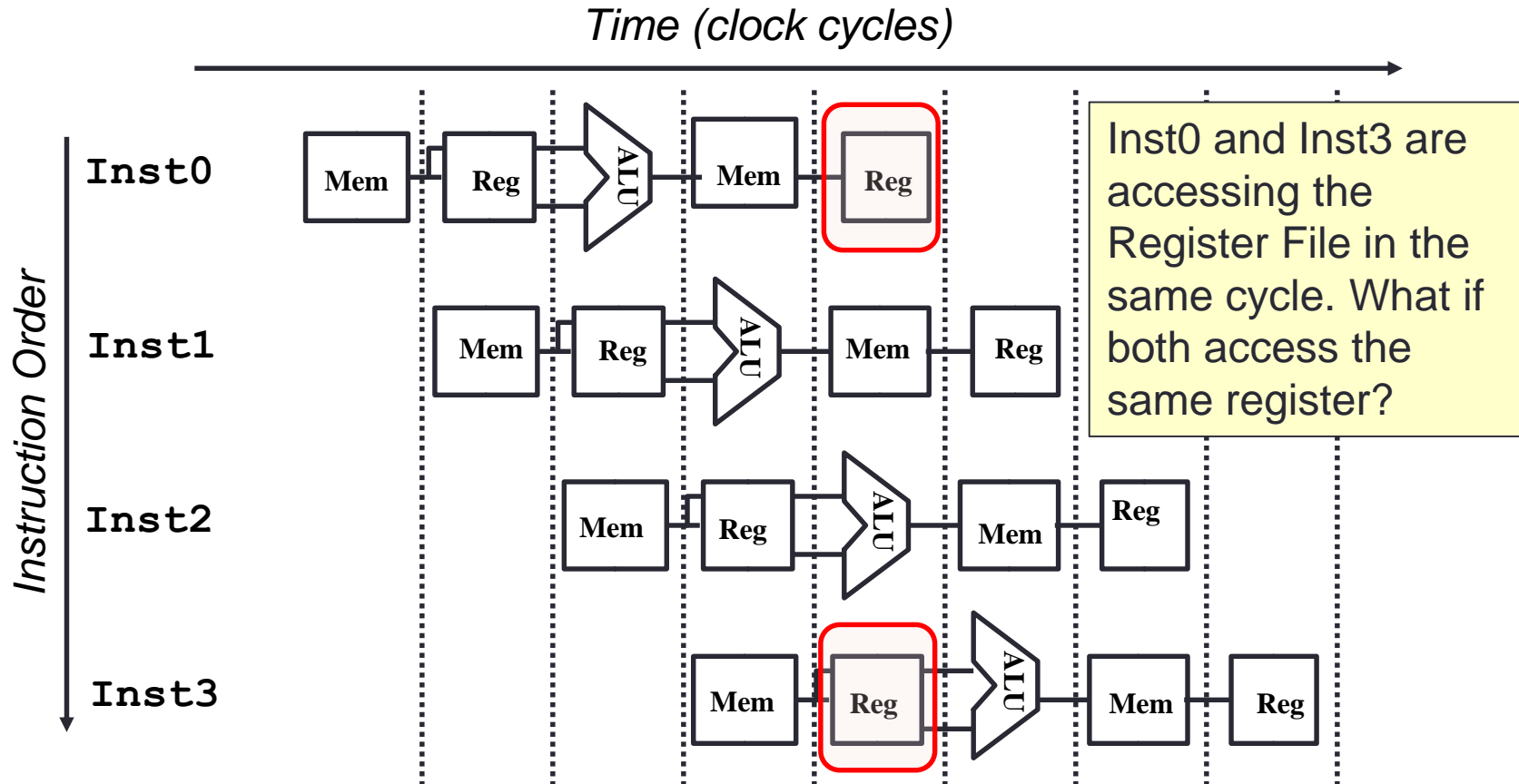
- Split memory into **Data** and **Instruction** memory





## 2. Quiz (1/2)

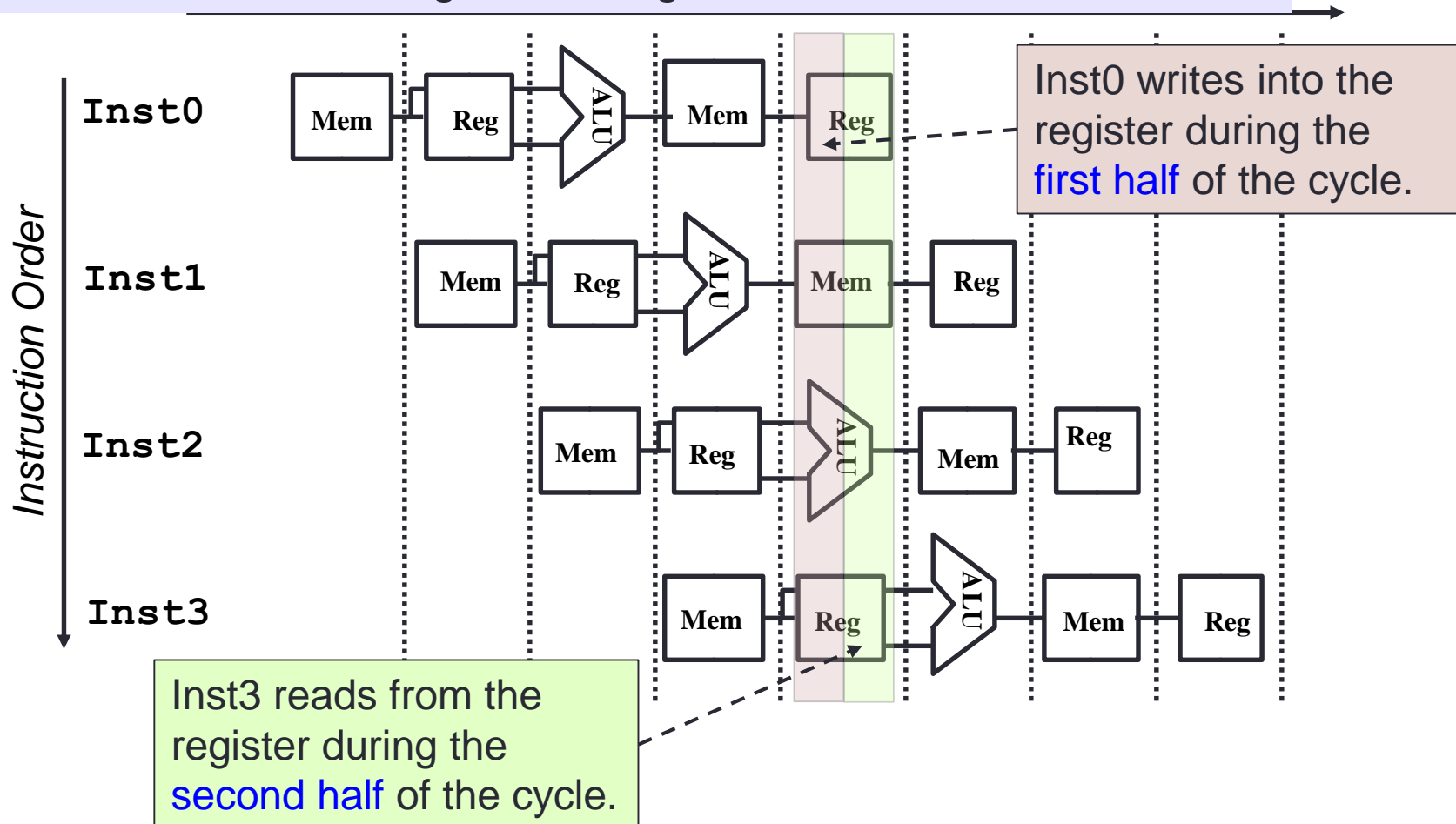
Is there another conflict?



## 2. Quiz (2/2)

Recall that registers are very fast memory.

Solution: **Split cycle into half**; **first half for writing** into a register; second half for reading from a register.



### 3. Instruction Dependencies

- Instructions can have relationship that prevent pipeline execution:
  - Although a partial overlap maybe possible in some cases
- When different instructions accesses (read/write) 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!**

### 3. Data Dependency: RAW

- "Read-After-Write" Definition:
  - Occurs when a later instruction **reads** from the destination register **written** by an earlier instruction
  - Also known as **true data dependency**

```
i1: add $1, $2, $3 #writes to $1
i2: sub $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)**

single cycle will not have this problem

### 3. Other Data Dependencies

- Similarly, we have:
  - **WAR: Write-after-Read** dependency
  - **WAW: Write-after-Write** dependency
- Fortunately, these dependencies **do not cause any pipeline hazards**
- They affect the processor only when instructions are executed out of program order:
  - i.e. in Modern SuperScalar Processor

## 4. RAW Dependency: Hazards?

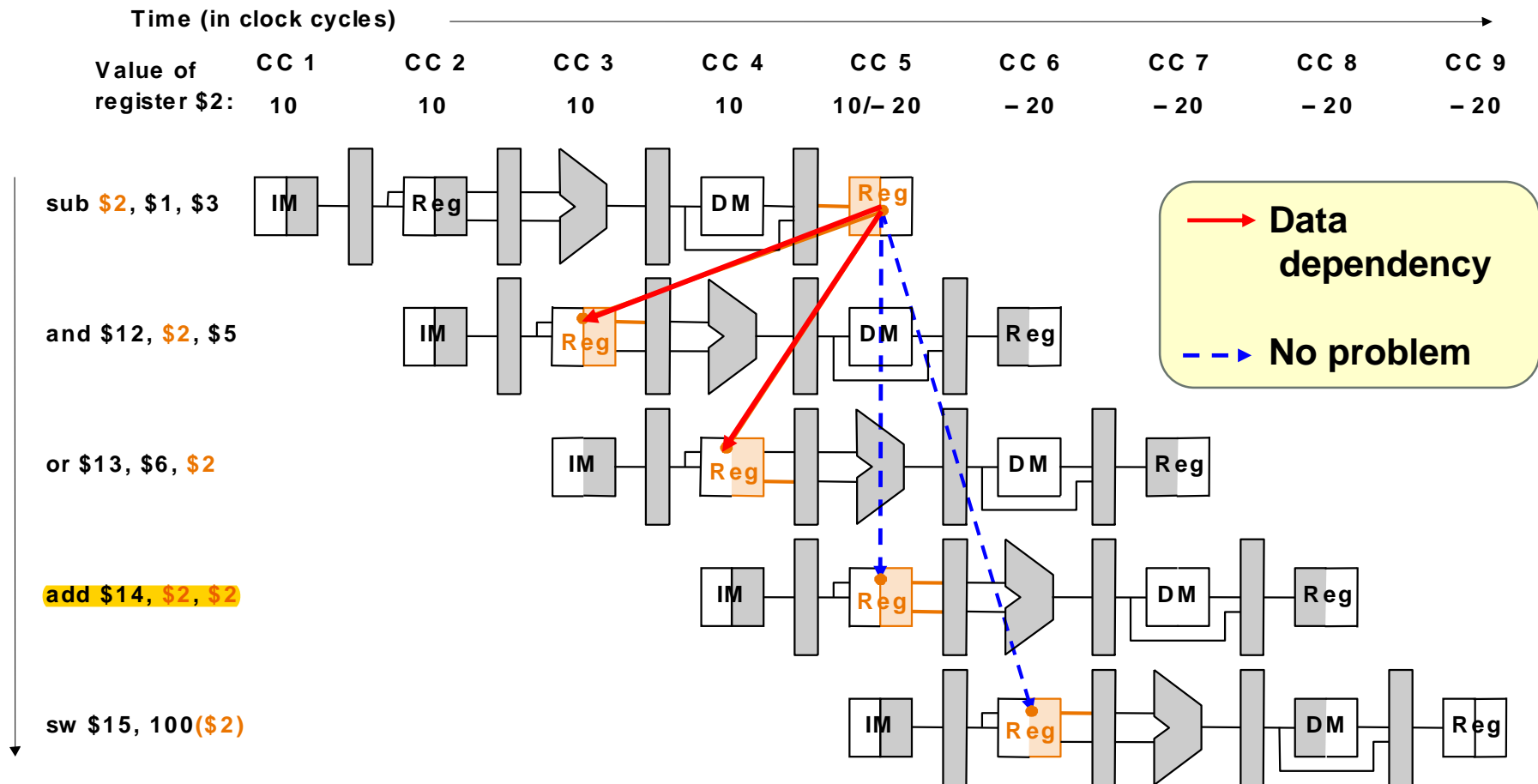
- Suppose we are executing the following code fragment:

```
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?

# 4. RAW Data Hazards

- Value from prior instruction is needed before write back



## 4. RAW Data Hazards: Observations

### ■ Questions:

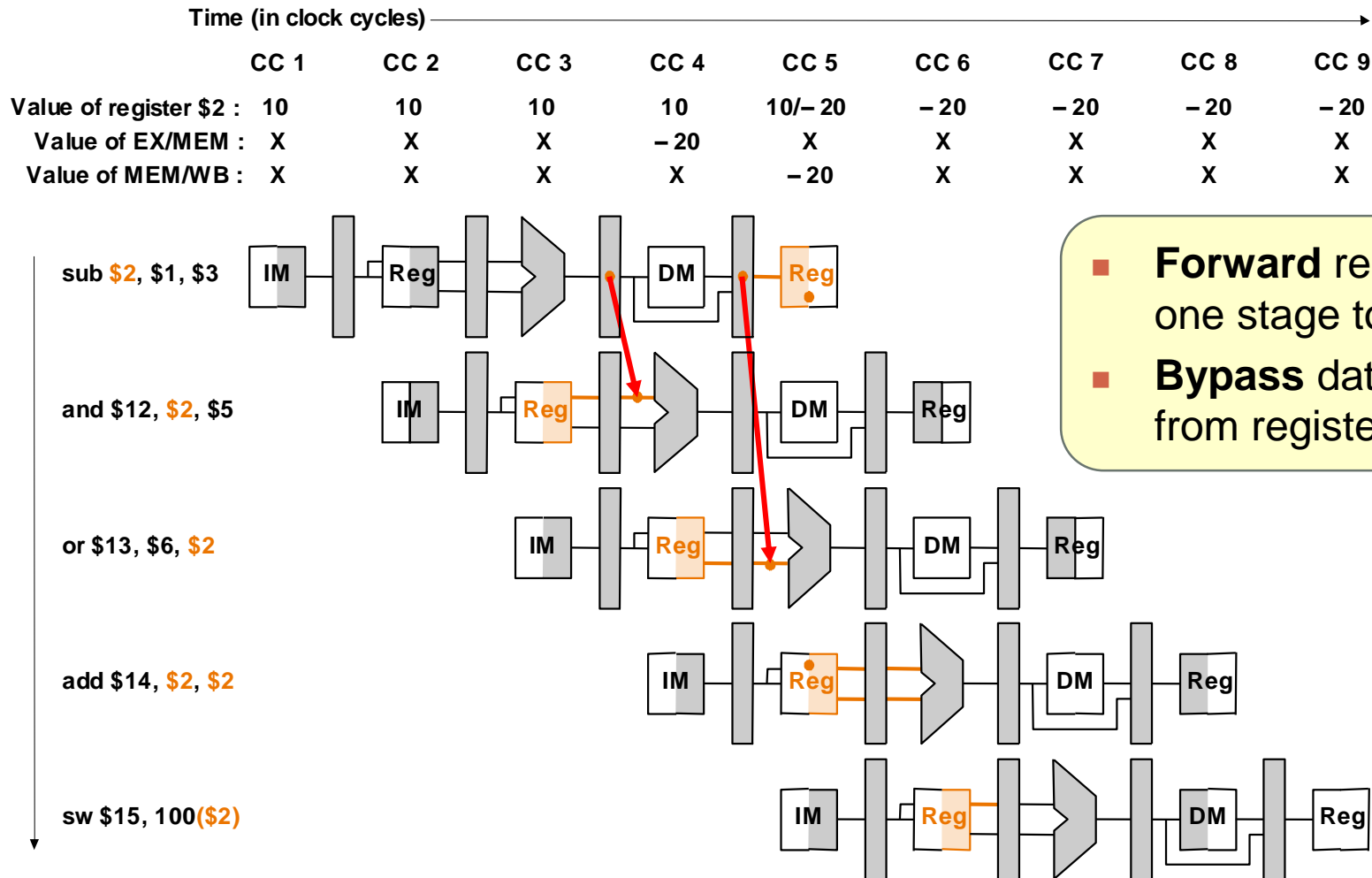
- When is the result from **sub** instruction actually produced?
  - End of EX stage for **sub** or clock cycle 3
- When is the data actually needed by **and**?
  - Beginning of **and**'s EX stage or clock cycle 4
- When is the data actually needed by **or**?
  - Beginning of **or**'s EX stage or clock cycle 5

### ■ Solution:

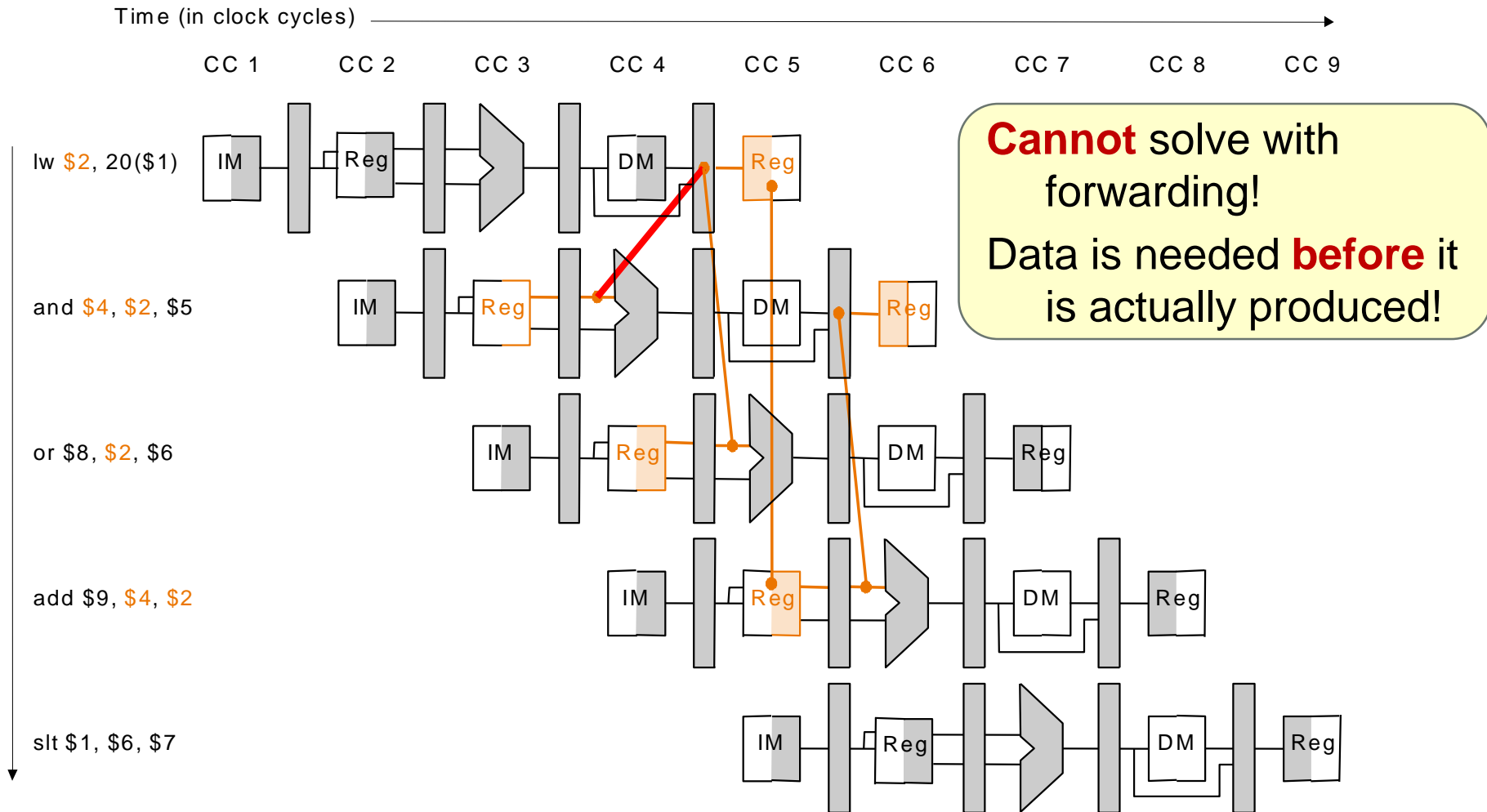
- **Forward** the result to any trailing (later) instructions before it is reflected in register file
- ➔ **Bypass (replace)** the data read from register file



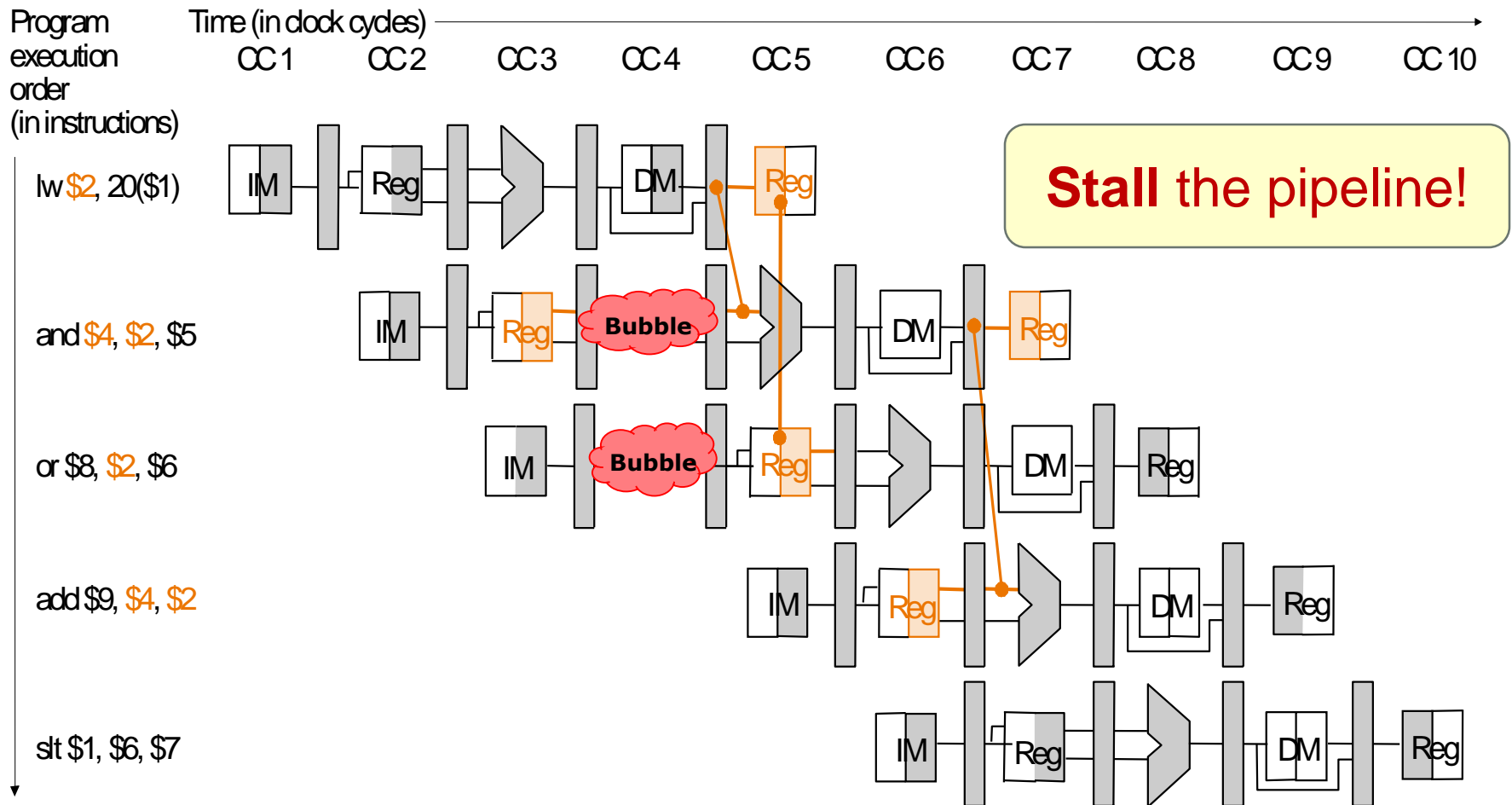
# 4.1 RAW Data Hazards: Forwarding



## 4.2 Data Hazards: **LOAD** Instruction



## 4.2 Data Hazards: **LOAD** Instruction Solution



## 4.3 Exercise #1

- How many cycles will it take to execute the following code on a 5-stage pipeline
  - **without** forwarding?
  - **with** forwarding?

```
sub  $2,  $1,  $3
and  $12,  $2,  $5
or   $13,  $6,  $2
add  $14,  $2,  $2
sw   $15,  100($2)
```

## 4.3 Exercise #1: Without Forwarding

```

sub  $2, $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)

```

	1	2	3	4	5	6	7	8	9	10	11
sub	IF	ID	EX	MEM	WB						
and		IF			ID	EX	MEM	WB			
or					IF	ID	EX	MEM	WB		
add						IF	ID	EX	MEM	WB	
sw							IF	ID	EX	MEM	WB

## 4.3 Exercise #1: With Forwarding

```

sub  $2, $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)

```

	1	2	3	4	5	6	7	8	9	10	11
sub	IF	ID	EX	MEM	WB						
and		IF	ID	EX	MEM	WB					
or			IF	ID	EX	MEM	WB				
add				IF	ID	EX	MEM	WB			
sw					IF	ID	EX	MEM	WB		

## 4.3 Exercise #2

- How many cycles will it take to execute the following code on a 5-stage pipeline
  - **without** forwarding?
  - **with** forwarding?

```
lw    $2, 20($3)
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
sw     $15, 100($2)
```

## 4.3 Exercise #2: Without Forwarding

```

lw    $2, 20($3)
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
sw     $15, 100($2)

```

	1	2	3	4	5	6	7	8	9	10	11
lw	IF	ID	EX	MEM	WB						
and		IF			ID	EX	MEM	WB			
or					IF	ID	EX	MEM	WB		
add						IF	ID	EX	MEM	WB	
sw							IF	ID	EX	MEM	WB



## 4.3 Exercise #2: With Forwarding

```

lw    $2, 20($3)
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
sw     $15, 100($2)

```

	1	2	3	4	5	6	7	8	9	10	11
lw	IF	ID	EX	MEM	WB						
and		IF	ID		EX	MEM	WB				
or			IF		ID	EX	MEM	WB			
add					IF	ID	EX	MEM	WB		
sw						IF	ID	EX	MEM	WB	

## 5. 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:

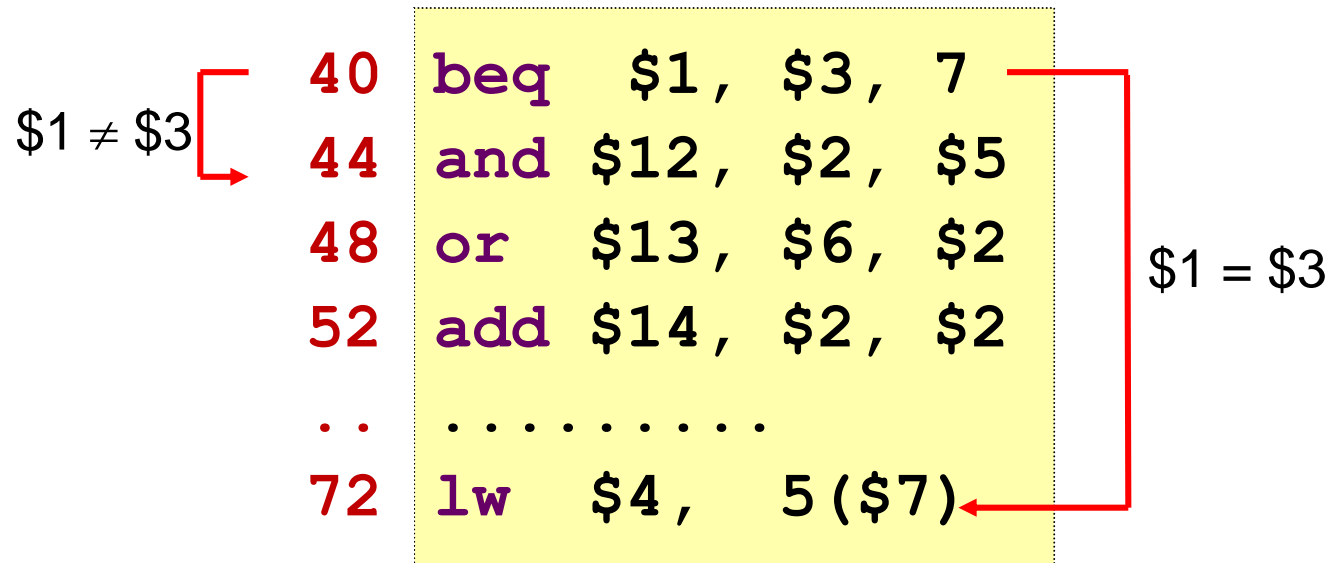
```
i1: beq $3, $5, label    # branch
i2: add $1, $2, $4       # depends on i1
...    ...    ...
```

### ■ Effect of incorrect execution:

- If *i2* is allowed to execute before *i1* is determined, register \$1 maybe incorrectly changed!

## 5. Control Dependency: Example

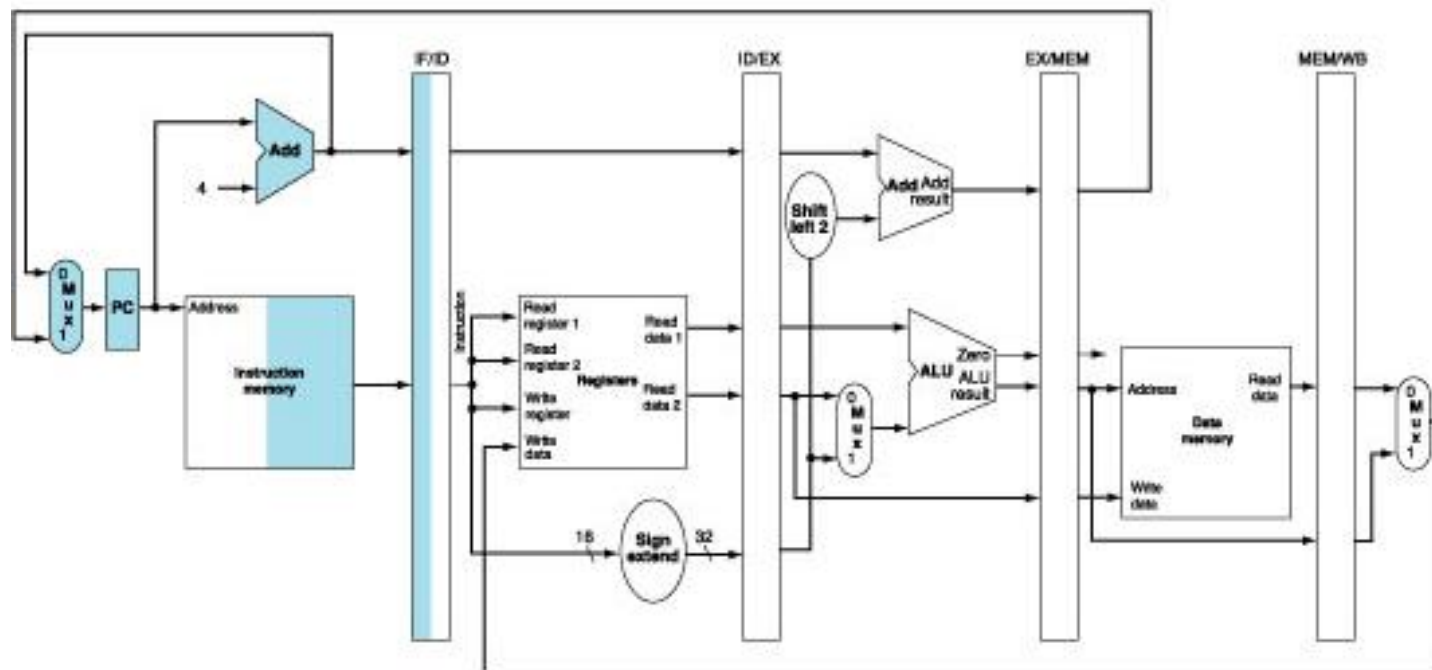
- Let us turn to a code fragment with a conditional branch:



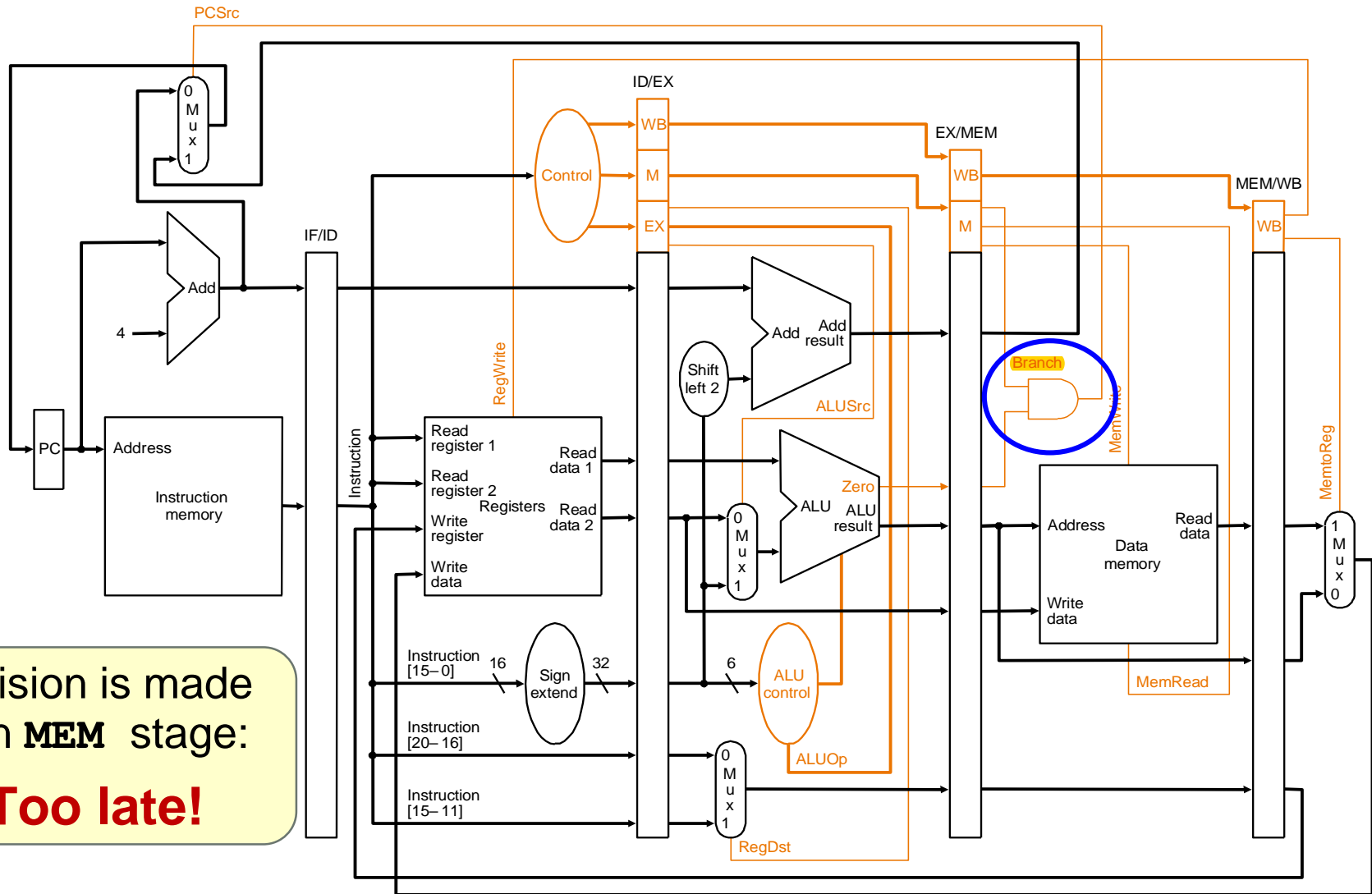
- How does the code affect a pipeline processor?

## 5. Pipeline Execution: **IF** Stage

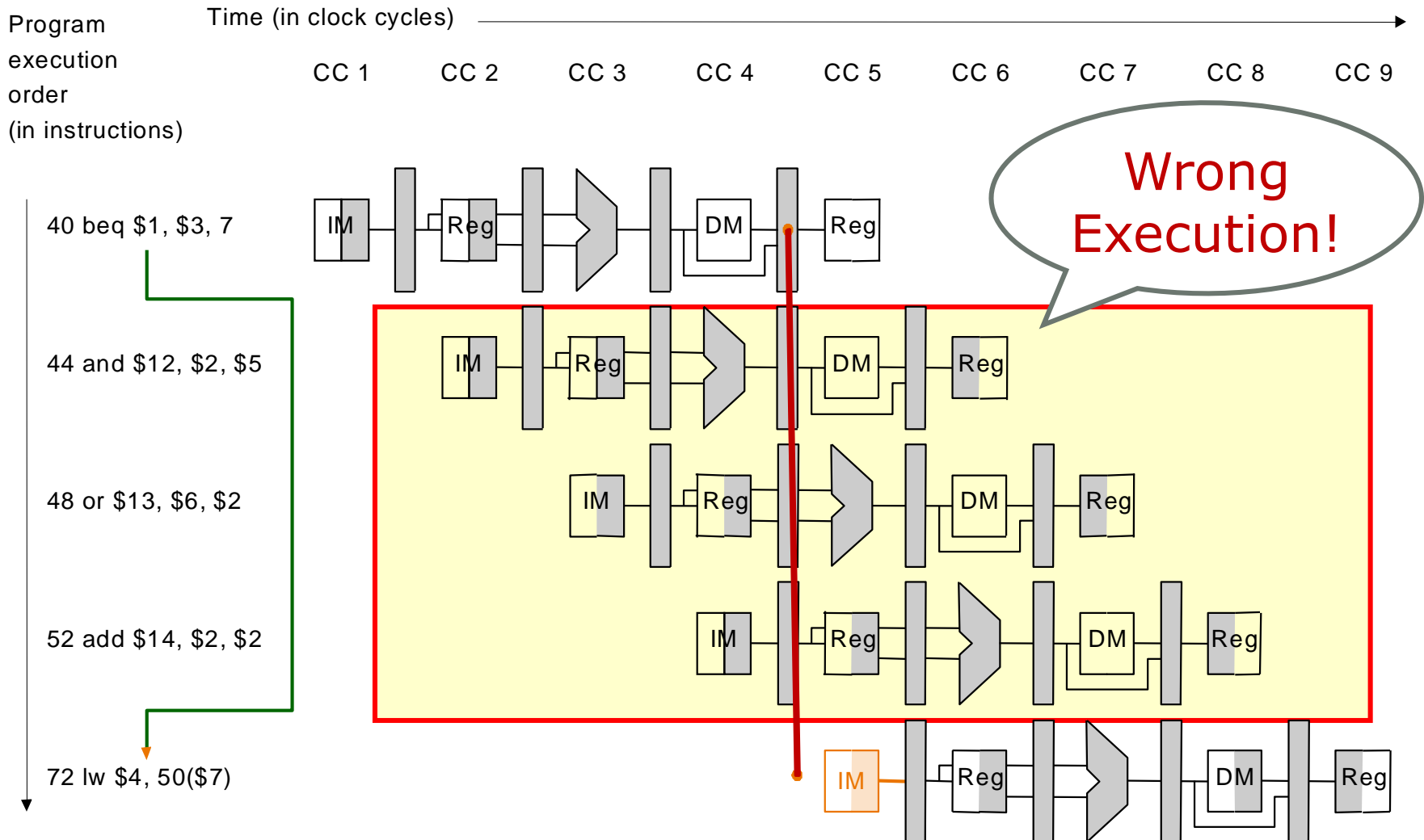
- Read instruction from memory using the address in PC and put it in **IF/ID** register
- PC address is incremented by 4 and then written back to the PC for next instruction



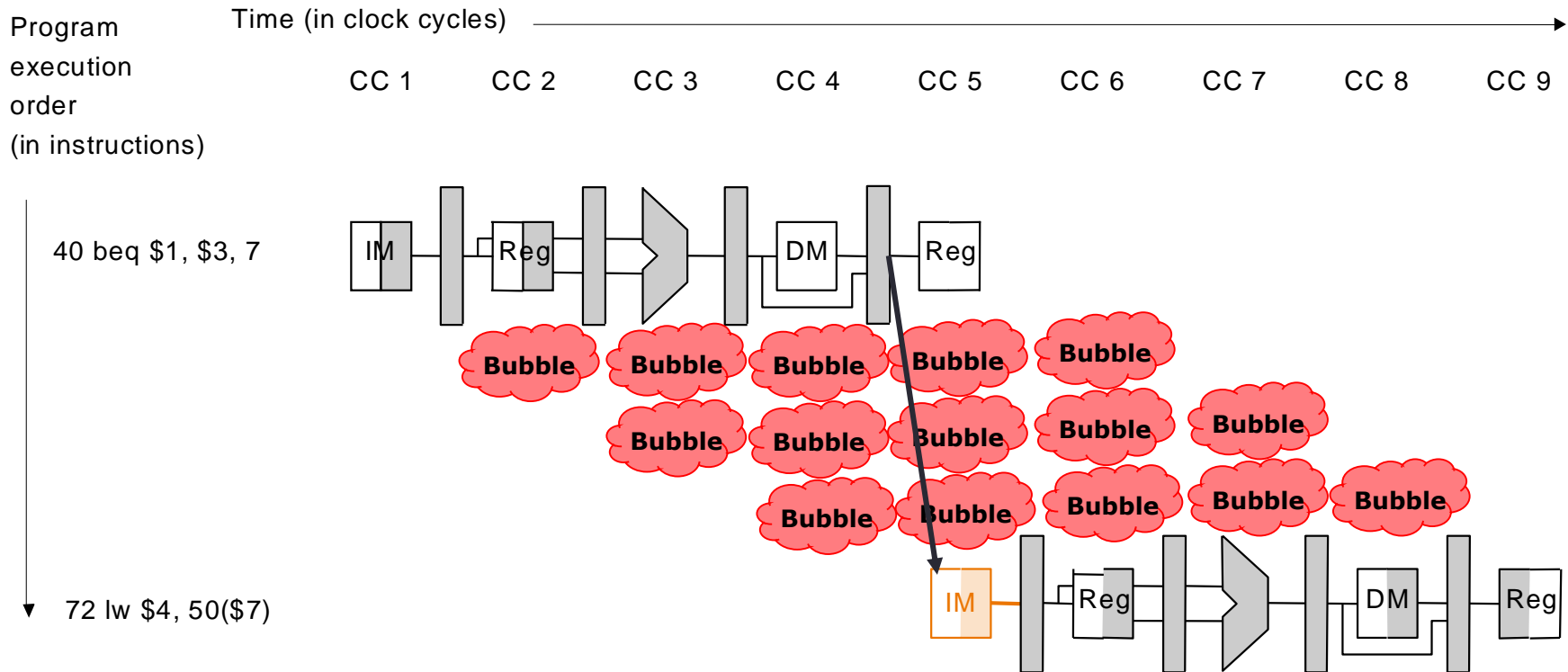
# 5. Control Dependency: Why?



# 5. Control Dependency: Example



## 6. Control Hazards: Stall Pipeline?



- Wait until the branch outcome is known and then fetch the correct instructions
- ➔ Introduces **3 clock cycles delay**

very trivial solution

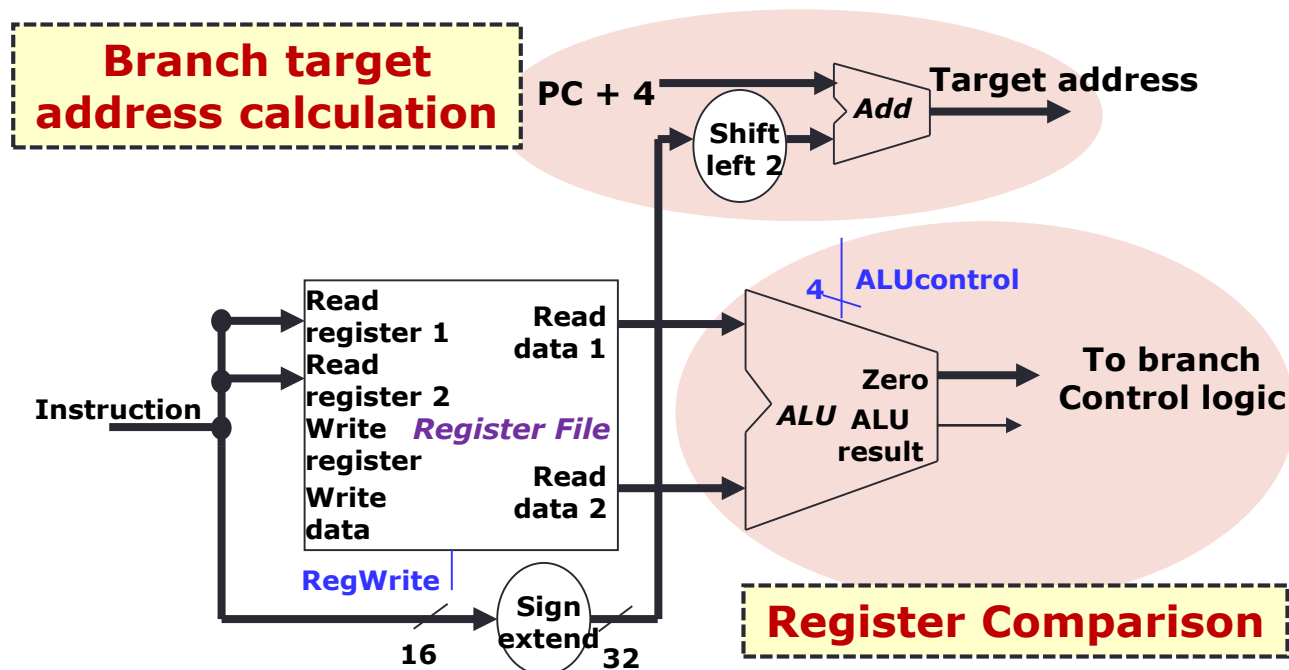
## 6. Control Hazards: Reducing the Penalty

- Branching is very common in code:
  - A 3-cycle stall penalty is too heavy!
- Many techniques invented to reduce the control hazard penalty:
  - Move branch decision calculation to earlier pipeline stage
    - **Early Branch Resolution**
  - Guess the outcome before it is produced
    - **Branch Prediction**
  - Do something useful while waiting for the outcome
    - **Delayed Branching**



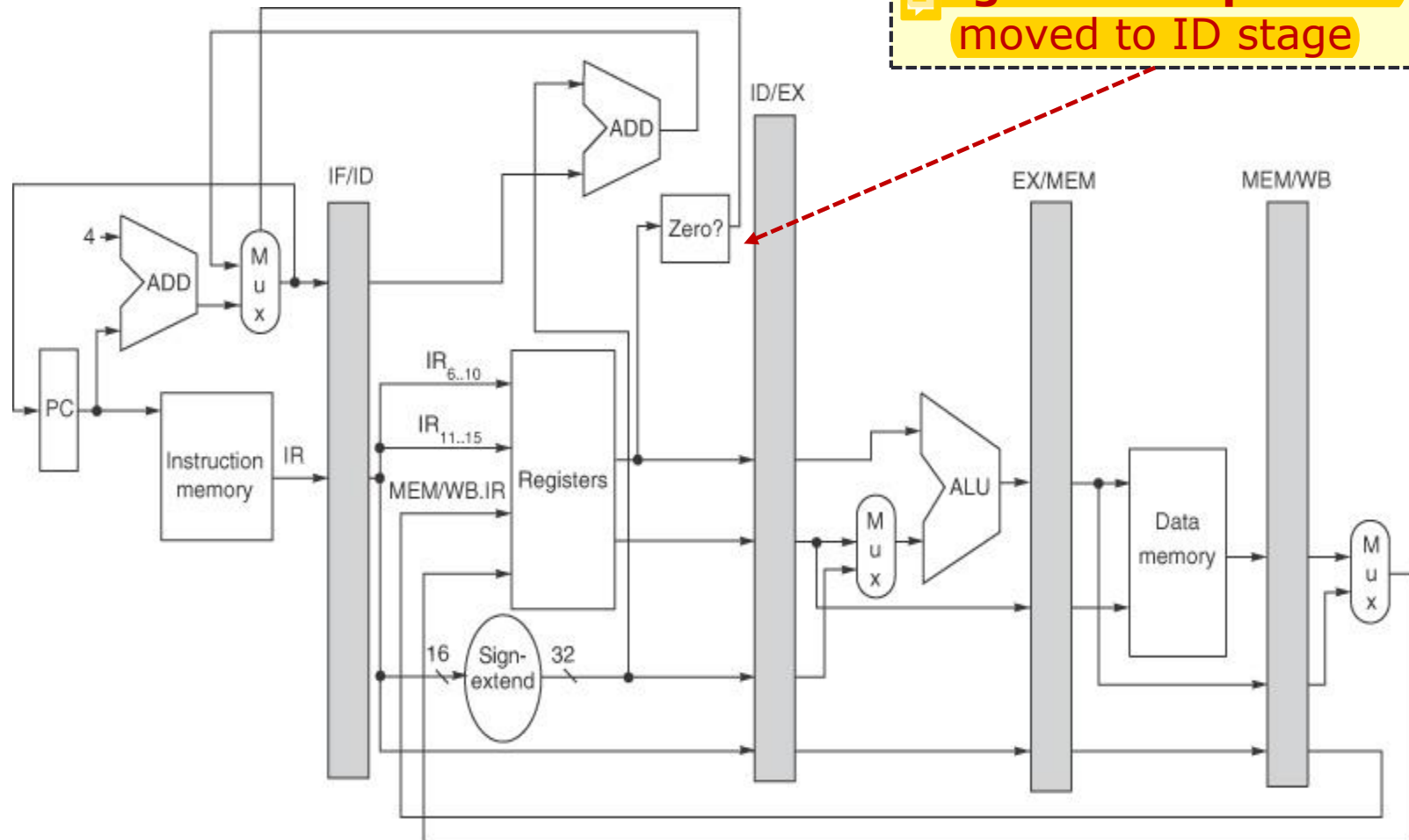
## 6.1 Reduce Stalls: Early Branch (1/3)

- Make decision in **ID** stage instead of **MEM**
  - Move branch target address calculation
  - Move register comparison → cannot use ALU for register comparison any more

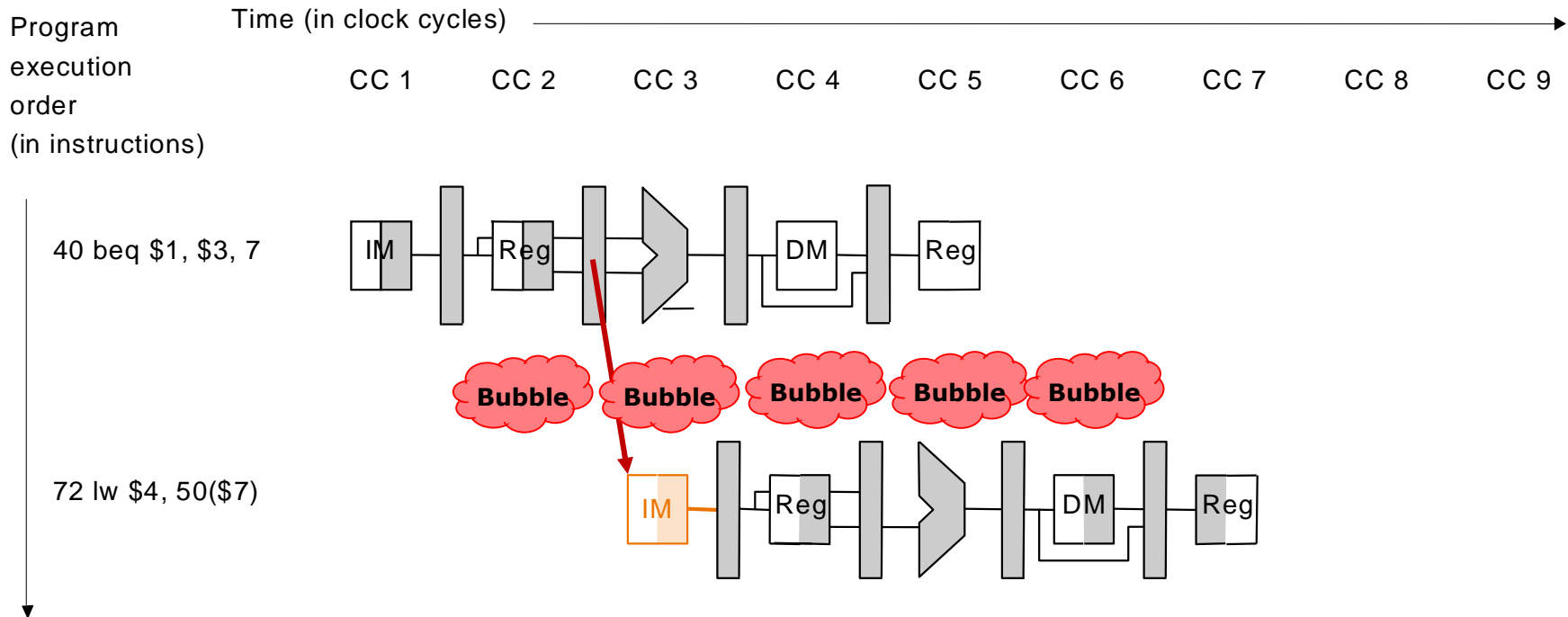


## 6.1 Reduce Stalls: Early Branch (2/3)

**Register comparison moved to ID stage**



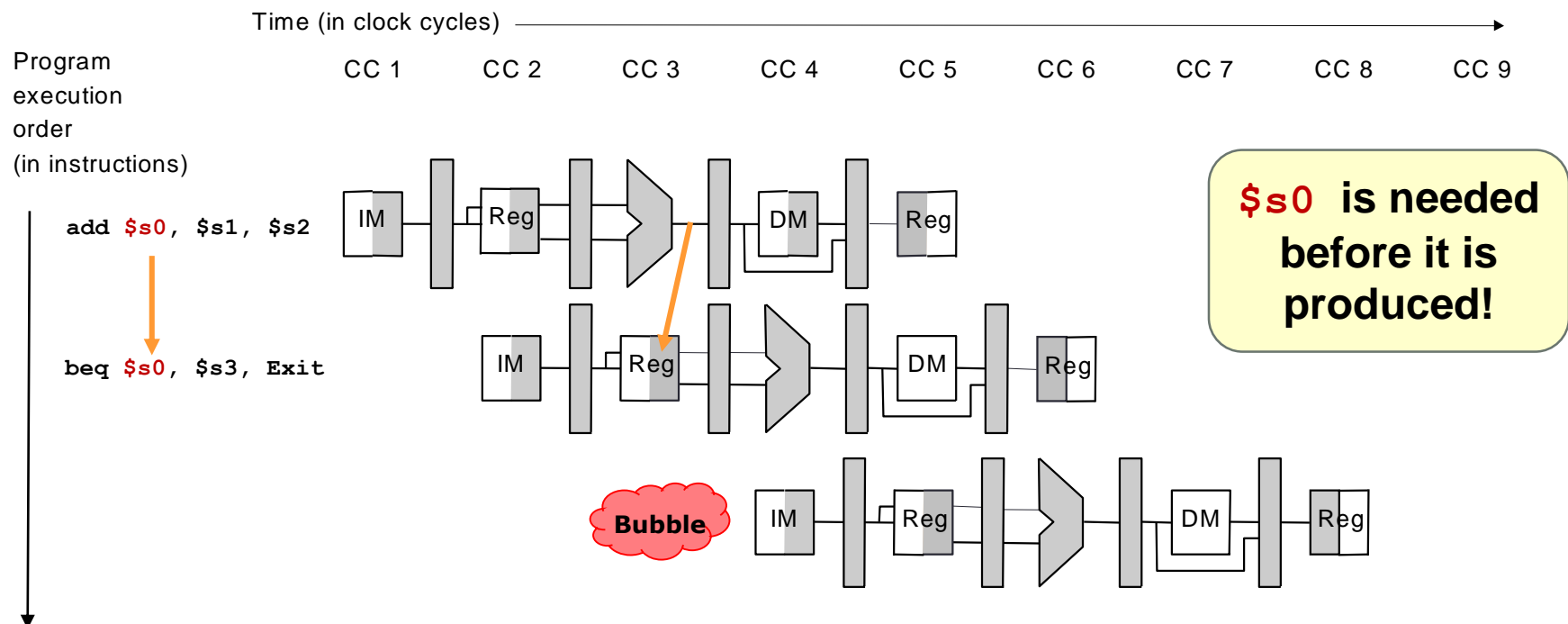
## 6.1 Reduce Stalls: Early Branch (3/3)



- Wait until the branch decision is known:
  - Then fetch the correct instruction
- Reduced from 3 to **1 clock cycle delay**

## 6.1 Early Branch: Problems (1/3)

- However, if the register(s) involved in the comparison is produced by preceding instruction:
  - Further stall is still needed!

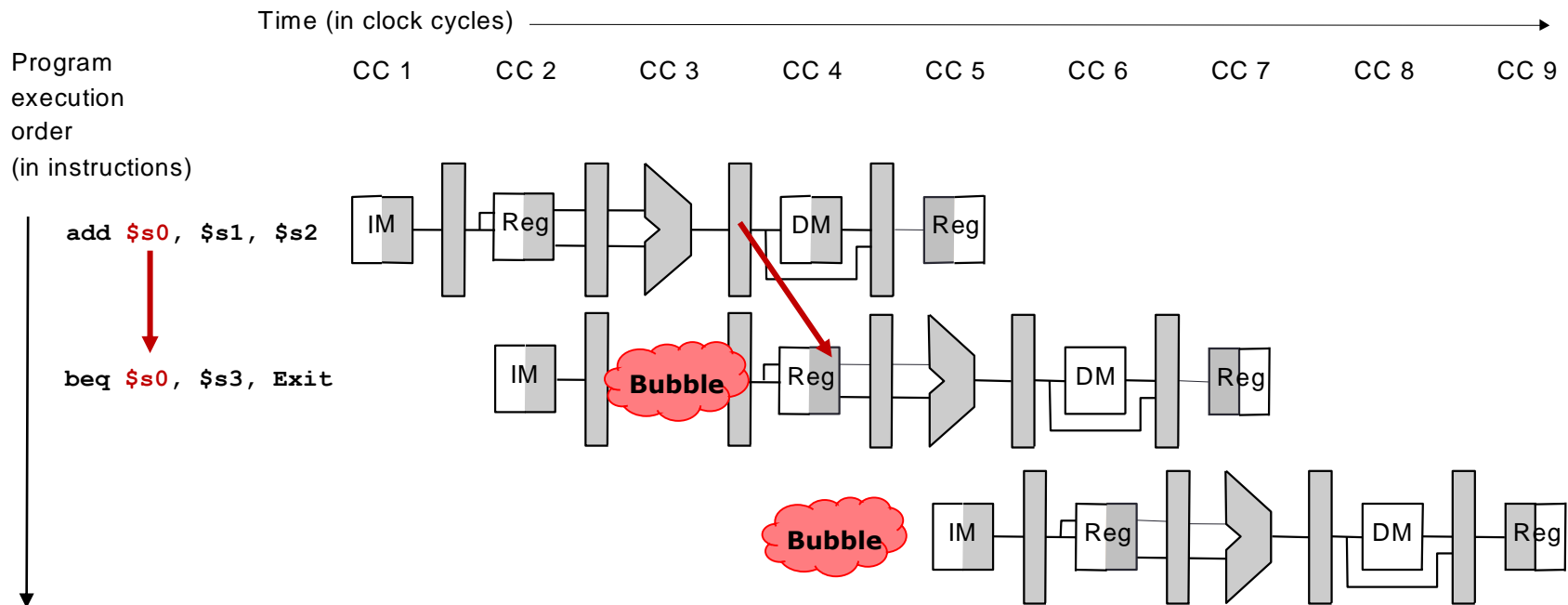


Because normally the subtract function for branching is in stage 3

## 6.1 Early Branch: Problems (2/3)

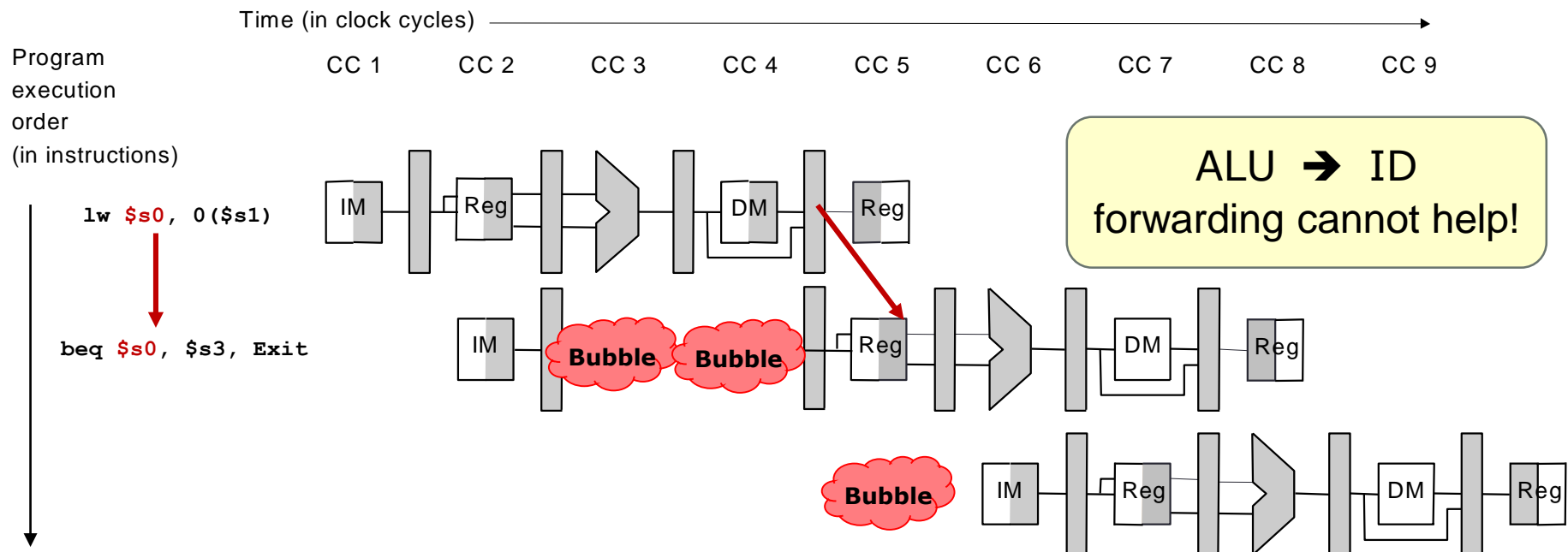
### ■ Solution:

- Add forwarding path from ALU to ID stage
- **One clock cycle delay** is still needed



## 6.1 Early Branch: Problems (3/3)

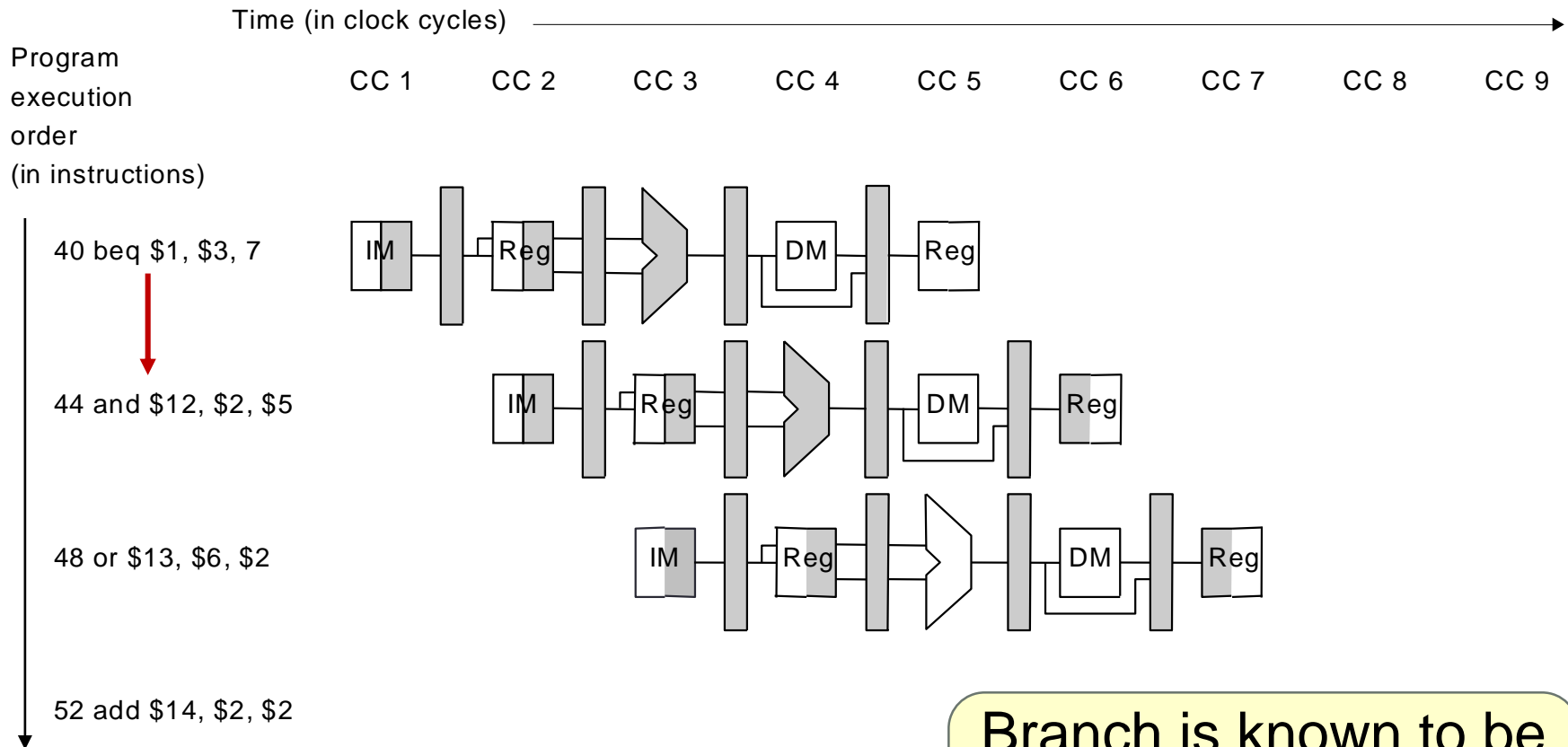
- Problem is worse with **load** followed by **branch**
- **Solution:**
  - MEM to ID forwarding and 2 more stall cycles!
  - In this case, we ended up with 3 total stall cycles  
➔ no improvement!



## 6.2 Reduce Stalls: Branch Prediction

- There are many branch prediction schemes
  - We only cover the simplest in this course 😊
- Simple prediction:
  - All branches are assumed to be **not taken**
  - ➔ Fetch the successor instruction and start pumping it through the pipeline stages
- When the actual branch outcome is known:
  - **Not taken**: Guessed correctly ➔ No pipeline stall
  - **Taken**: Guessed wrongly ➔ Wrong instructions in the pipeline ➔ **Flush** successor instruction from the pipeline

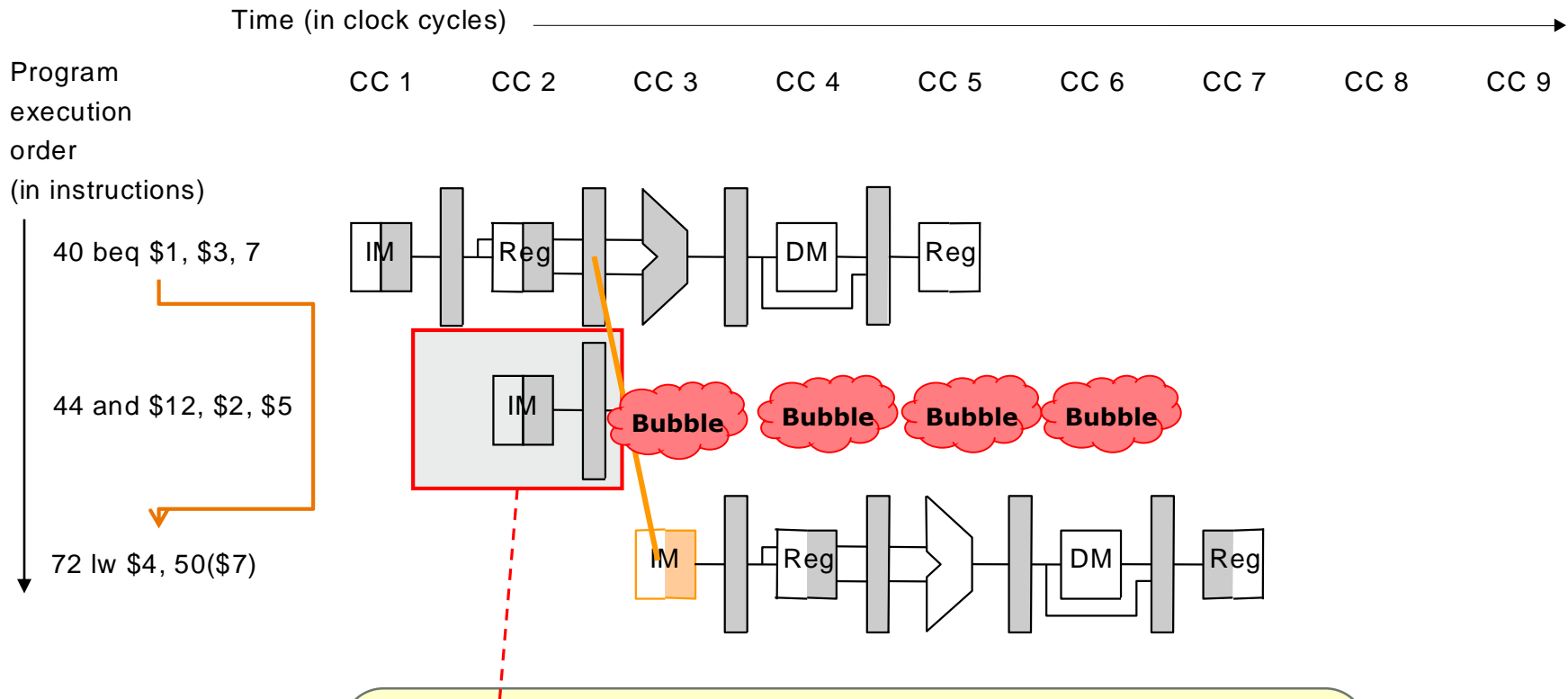
## 6.2 Branch Prediction: Correct Prediction



Branch is known to be  
**not taken** in cycle 3  
→ no stall needed!



## 6.2 Branch Prediction: **Wrong Prediction**



Branch is known to be **taken** in cycle 3  
→ "**and**" instruction should not be executed  
→ Flushed from pipeline

## 6.2 Exercise #3: Branch Prediction

- How many cycles will it take to execute the following code on a 5-stage pipeline **with** forwarding and ...
  - **without** branch prediction?
  - **with** branch prediction (predict not taken)?

```
        addi $s0, $zero, 10
Loop:   addi $s0, $s0, -1
        bne  $s0, $zero, Loop
        sub  $t0, $t1, $t2
```


- Decision making moved to **ID** stage
- Total instructions =  $1 + 10 \times 2 + 1 = 22$
- **Ideal** pipeline =  $4 + 22 = 26$  cycles

## 6.2 Exercise #3: Without Branch Prediction

	1	2	3	4	5	6	7	8	9	10	11
<b>addi<sup>1</sup></b>	IF	ID	EX	MEM	WB						
<b>addi<sup>2</sup></b>		IF	ID	EX	MEM	WB					
<b>bne</b>			IF		ID	EX	Mem	WB			
<b>addi<sup>2</sup></b>						IF	ID	EX	Mem	WB	

- Data dependency between (**addi** \$s0, \$s0, -1) and **bne** incurs 1 cycle of delay. There are 10 iterations, hence 10 cycles of delay.
- Every **bne** incurs a cycle of delay to execute the next instruction. There are 10 iterations, hence 10 cycles of delay.
- Total number of cycles of delay = **20**.
- Total execution cycles = 26 + **20** = **46 cycles**.

## 6.2 Exercise #3: With Branch Prediction

	1	2	3	4	5	6	7	8	9	10	11
addi <sup>1</sup>	IF	ID	EX	MEM	WB						
addi <sup>2</sup>		IF	ID	EX	MEM	WB					
bne			IF	 ID		EX	MEM	WB			
sub				<del>IF</del>	<del>EX</del>	<del>MEM</del>					
addi <sup>2</sup>						IF	ID	EX	MEM	WB	

### Predict not taken.

- The data dependency remains, hence 10 cycles of delay for 10 iterations.
- In the first 9 iterations, the branch prediction is wrong, hence 1 cycle of delay.
- In the last iteration, the branch prediction is correct, hence saving 1 cycle of delay.
- Total number of cycles of delay = 19.
- Total execution cycles = 26 + 19 = **45 cycles**.

## 6.3 Reduce Stalls: Delayed Branch

- **Observation:**

- Branch outcome takes **X** number of cycles to be known  
→ **X** cycles stall

- **Idea:**

- Move **non-control dependent instructions** into the **X** slots following a branch
  - Known as the **branch-delay slot**
- These instructions are executed **regardless of the branch outcome**

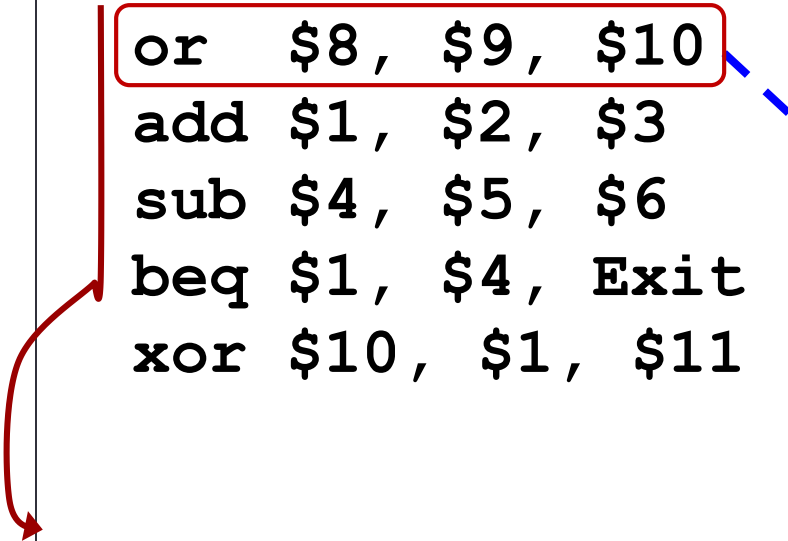
- In our MIPS processor:

- Branch-Delay slot = **1** (with the early branch)

## 6.3 Delayed Branch: Example

### *Non-delayed branch*

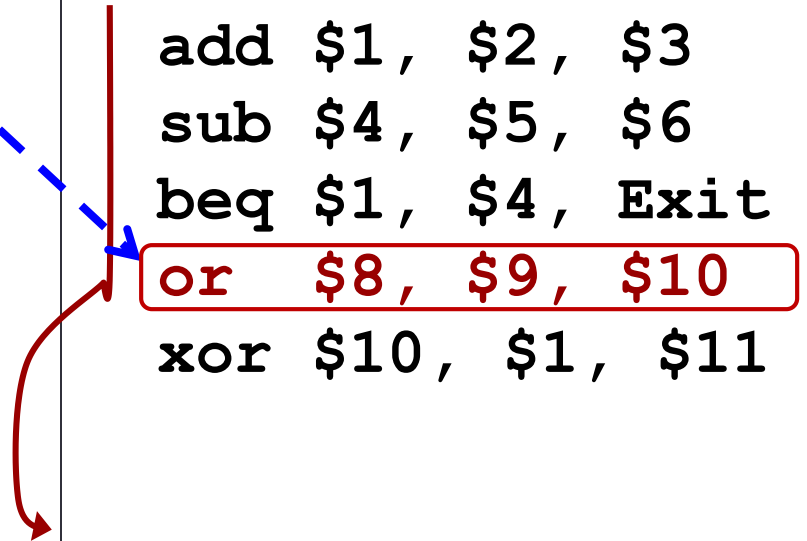
```
or   $8, $9, $10  
add  $1, $2, $3  
sub  $4, $5, $6  
beq  $1, $4, Exit  
xor  $10, $1, $11
```



**Exit:**

### *Delayed branch*

```
add  $1, $2, $3  
sub  $4, $5, $6  
beq  $1, $4, Exit  
or   $8, $9, $10  
xor  $10, $1, $11
```



**Exit:**

- The "**or**" instruction is moved into the delayed slot:
  - Get executed regardless of the branch outcome
- ➔ Same behavior as the original code!

## 6.3 Delayed Branch: Observation

- **Best case scenario**
  - There is an instruction **preceding the branch** which **can be moved** into the delayed slot
    - Program correctness must be preserved!
- **Worst case scenario**
  - Such instruction cannot be found
    - ➔ Add a no-op (**nop**) instruction in the branch-delay slot
- Re-ordering instructions is a common method of program optimization
  - Compiler must be smart enough to do this
  - Usually can find such an instruction at least 50% of the time

## 7. Multiple Issue Processors (1/2)

For reading only

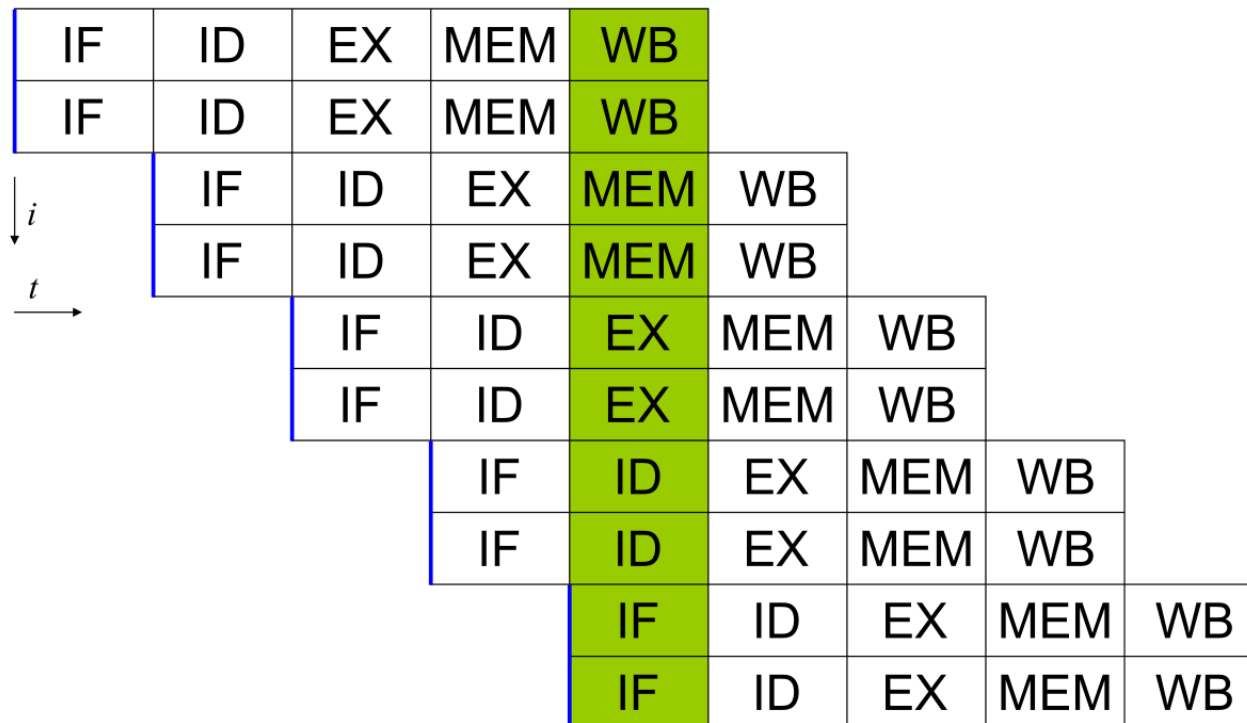
- Multiple Issue processors
  - **Multiple instructions** in every pipeline stage
  - 4 washer, 4 dryer...
- **Static multiple issue:**
  - EPIC (Explicitly Parallel Instruction Computer) or VLIW (Very Long Instruction Word), e.g. IA64
  - Compiler specifies the set of instructions that execute together in a given clock cycle
  - Simple hardware, complex compiler
- **Dynamic multiple issue:**
  - Superscalar processor: Dominant design of modern processors
  - Hardware decides which instructions to execute together
  - Complex hardware, simpler compiler



## 7. Multiple Issue Processors (2/2)

For reading only

- A 2-wide superscalar pipeline:
  - By fetching and dispatching two instructions at a time, a maximum of two instructions per cycle can be completed.



# Summary

- Pipelining is a fundamental concept in computer systems
  - Multiple instructions in flight
  - Limited by length of the longest stage
  - Hazards create trouble by stalling pipeline
- Pentium 4 has 22 pipeline stages!

# Reading

- **3<sup>rd</sup> edition**
  - Sections 6.1 – 6.3
  - Sections 6.4 – 6.6 (data hazards and control hazards in details; read for interest; not in syllabus)
- **4<sup>th</sup> edition**
  - Sections 4.5 – 4.6
  - Sections 4.7 – 4.8 (data hazards and control hazards in details; read for interest; not in syllabus)



End of File