

12.1 - Other types of instruction implementation

- Single cycle implementation
- Multicycle implementation

12.2 - Pipelining implementation

- Datapath
- Control
- Performance

12.3 - Pipelining hazards

- Structural hazards
- Data hazards
- Control hazards

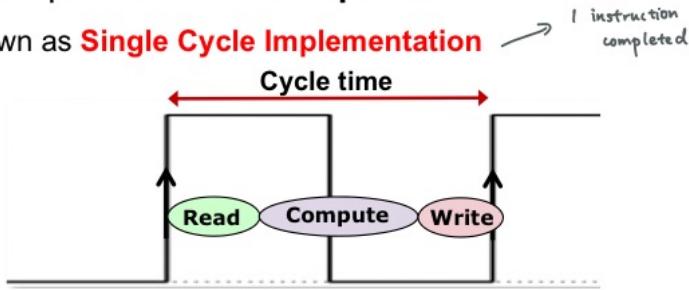
Hazard resolutions:

- Forwarding
- Branch handling

12.1 - Other types of instruction implementation

Single cycle implementation

- The MIPS processor in Lectures #10-#11 performs all these steps **within a clock period**
 - Known as **Single Cycle Implementation**

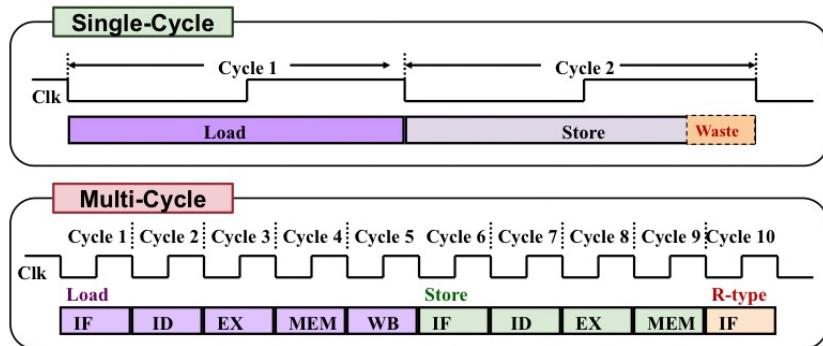


Single Cycle Processor: Performance

- Cycle time:
 - $CT_{seq} = \sum_{k=1}^N T_k$ T_k = Time for operation in stage k
 N = Number of stages
- Each instruction takes **one** cycle to execute:
 - No. of instructions == No. of cycles needed
- Total Execution Time** for **I** instructions:
 - $Time_{seq} = \text{Cycles} \times \text{CycleTime}$
 $= I \times CT_{seq} = I \times \sum_{k=1}^N T_k$

Multicycle implementation

Single-Cycle vs Multi-Cycle



- Observe that **Load** takes **5 cycles** but **Store** takes only **4 cycles** in multi-cycle implementation

Multi-Cycle Processor: Performance

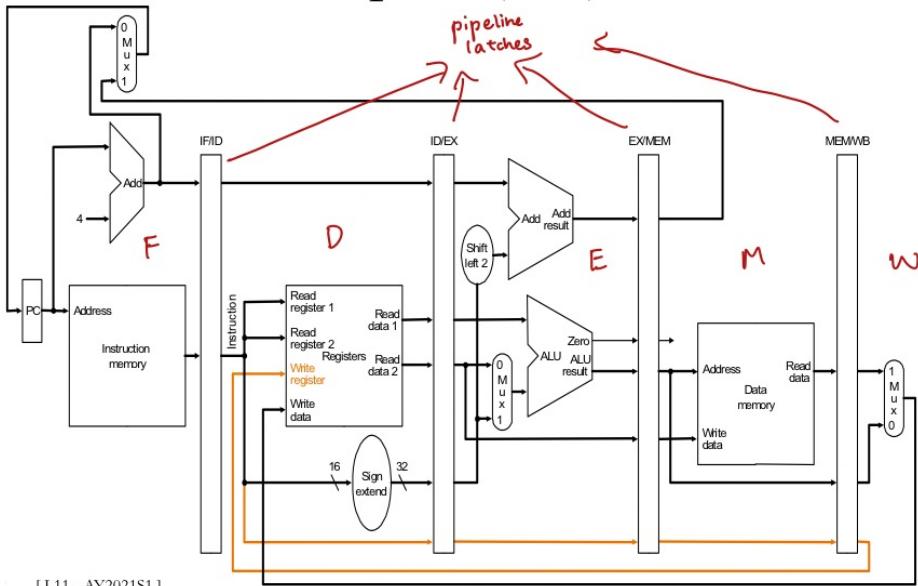
- Cycle time:
 - $CT_{multi} = \max(T_k)$
= longest stage time among the N stages
- Instruction takes variable number of cycles:
 - Use **average cycle per instruction (CPI)**
- Total Execution Time for **I** instructions:
 - $Time_{multi} = \text{Cycles} \times \text{CycleTime}$
 $= I \times \text{Average CPI} \times CT_{multi}$

12.2 - Pipelining implementation

Datapath

- Allow different instructions to be in different execution steps simultaneously

Corrected Datapath (2/2)



[L11 - AY2021S1]

Control

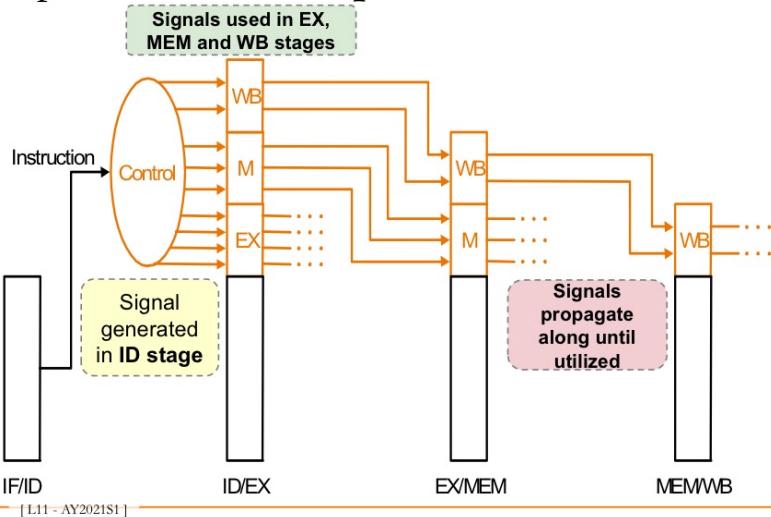
Pipeline Control: Grouping

- Group control signals according to pipeline stage

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0

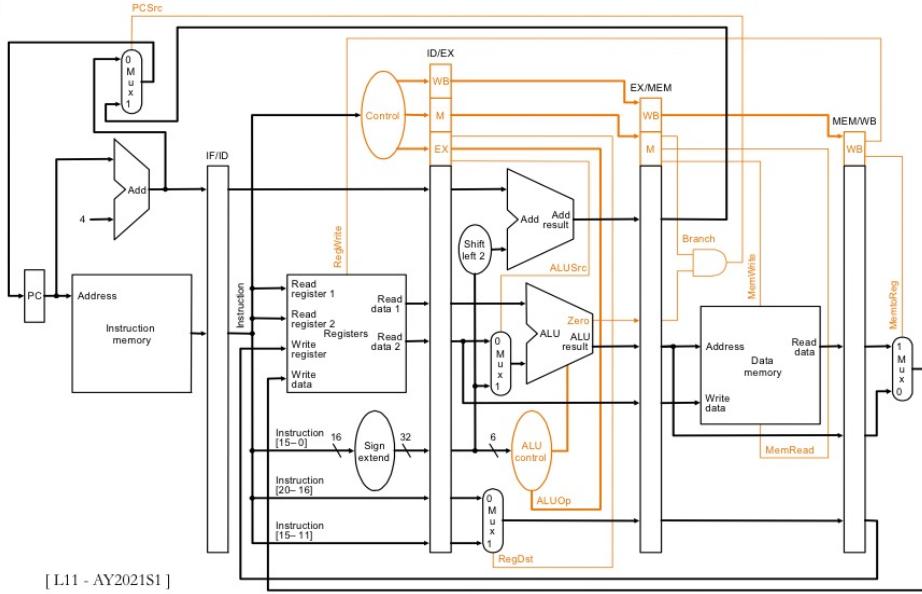
Pipeline Control: Implementation



MIPS Pipeline: Datapath and Control

+ Latches

+ Control info



[L11 - AY2021SI]

Performance

Pipeline Processor: Performance

■ Cycle Time:

- $CT_{pipe} = \max(T_k) + T_d$

$\max(T_k)$
= longest stage time among the N stages

T_d
= Pipeline overhead, e.g. pipeline register

■ Cycles needed for **I** instructions:

- $I + N - 1$

- $(N - 1)$ is the cycles for filling up the pipeline

warm up

■ Total Time needed for **I** instructions :

- $Time_{pipe} = Cycle \times CT_{pipe}$
 $= (I + N - 1) \times (\max(T_k) + T_d)$

Pipeline Processor: Ideal Speedup

- Assumptions for ideal case:
 - a. Every stage takes the same amount of time:
 $\rightarrow \sum_{k=1}^N T_k = N \times T_k$
 - b. No pipeline overhead $\rightarrow T_d = 0$
 - c. Number of instructions I , is much larger than number of stages, N \rightarrow dilute warmup overhead
- The above assumptions can also be used to figure out **how pipeline processor loses performance**

[L11 - AY2021S1]

■ $Speedup_{pipe} = \frac{Time_{seq}}{Time_{pipe}}$

Make use of all 3 assumptions

$$\begin{aligned} &= \frac{I \times \sum_{k=1}^N T_k}{(I+N-1) \times (\max(T_k) + T_d)} \\ &= \frac{I \times N \times T_k}{(I+N-1) \times T_k} \\ &\approx \frac{I \times N \times T_k}{I \times T_k} \\ &\approx N \end{aligned}$$

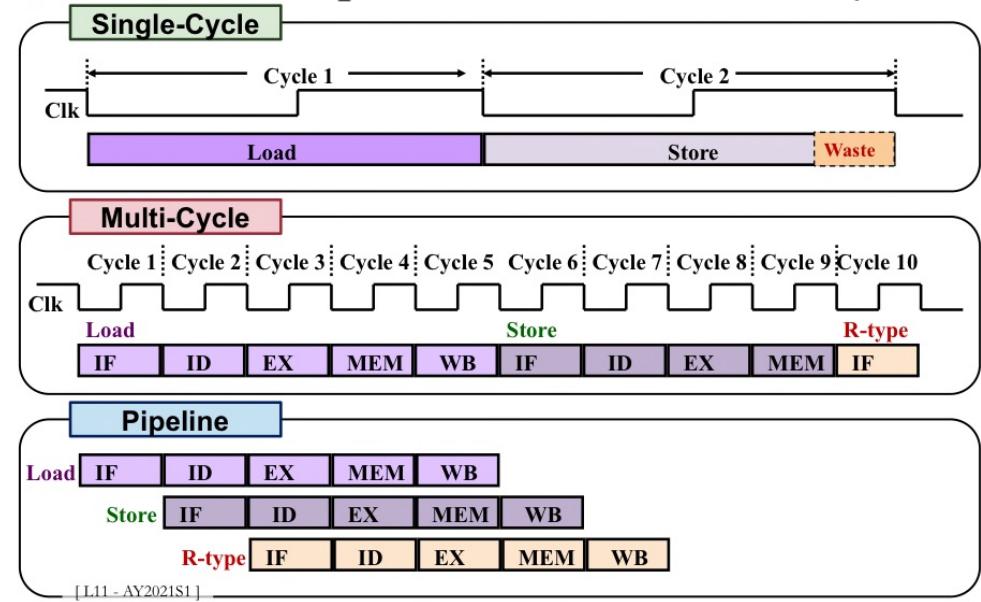
Conclusion:

Pipeline processor can gain **N** times speedup, where **N** is the number of pipeline stages

[L11 - AY2021S1]

Processor implementations : summary

Processor Implementations: Summary

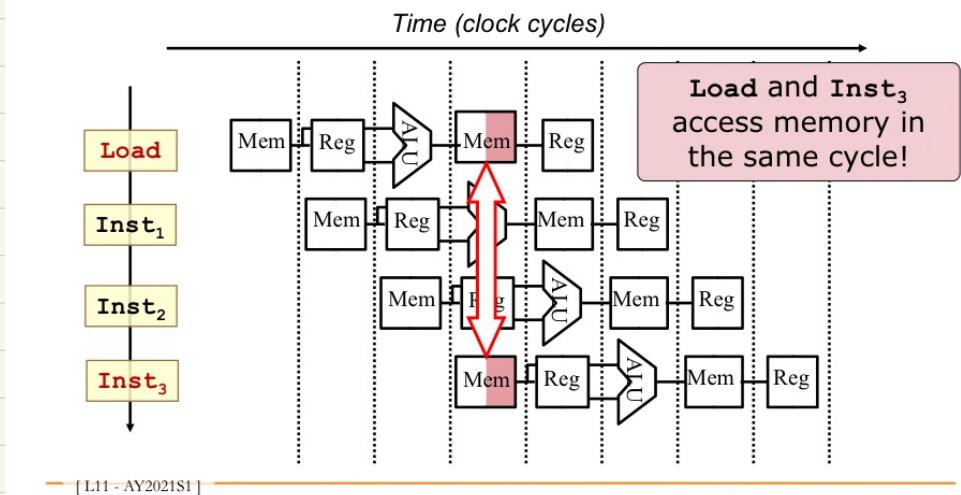


12.3 - Pipelining hazards

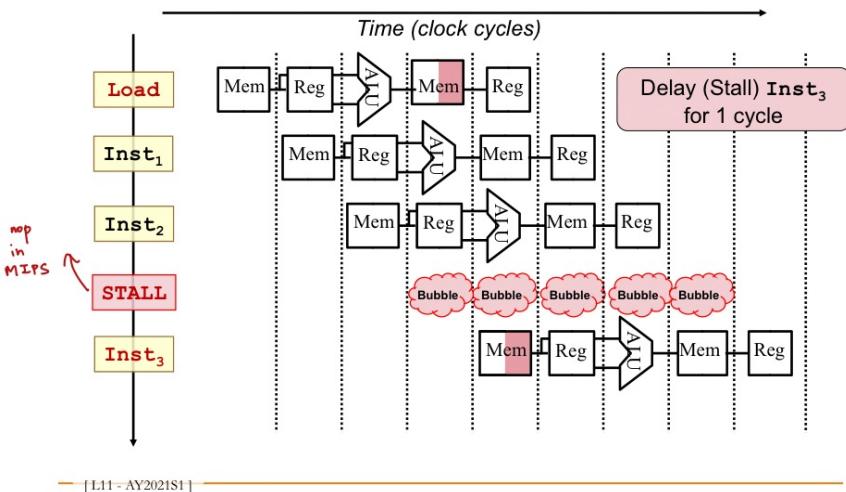
Structural hazards

Structure Hazard: Example

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



Solution 1: Stall the Pipeline



Solution 2: Separate Memory

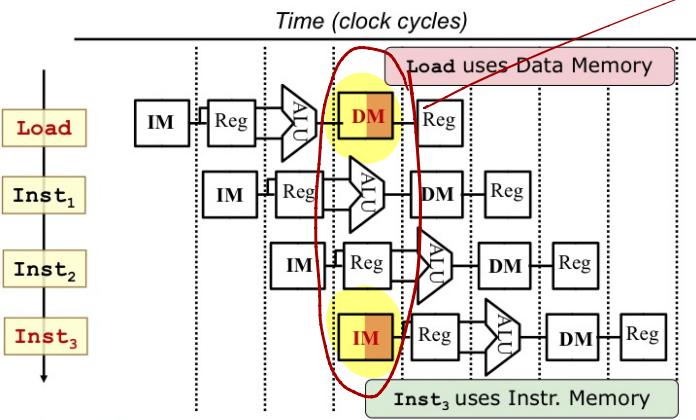
- Split memory into: **Data and Instruction Memory**

Implemented
in MIPS

1 clock cycle:



always preferred



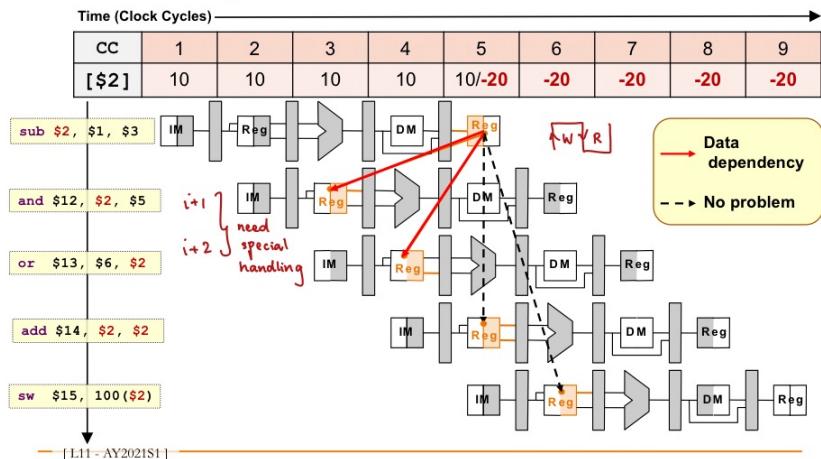
Data hazards (dependencies)

register contention
is the cause

For normal R-type instruction (lw next slide)

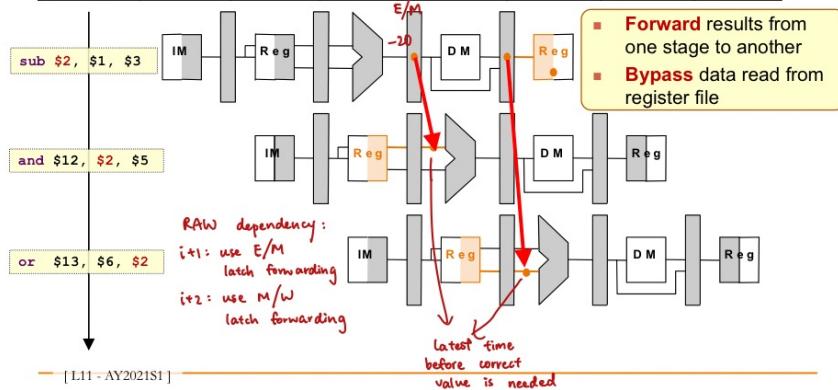
RAW Data Hazards: Illustration

- Value from prior instruction is needed before write back

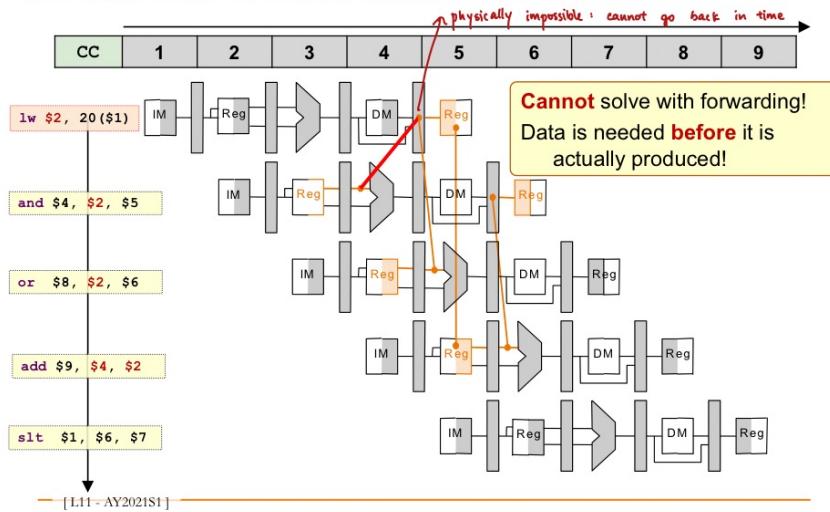


Solution: Forwarding

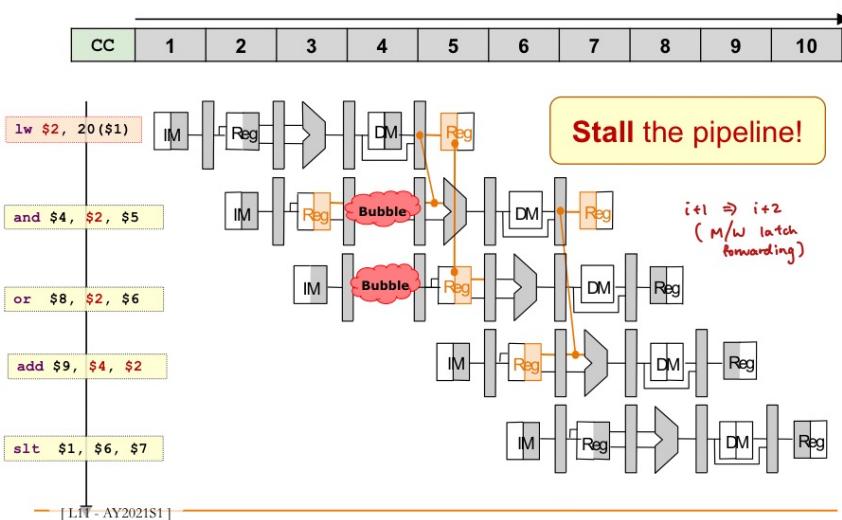
CC	1	2	3	4	5	6	7
[\$2]	10	10	10	10	10/-20	-20	-20
EX/MEM	X	X	X	-20	X	X	X
MEM/WB	X	X	X	X	-20	X	X



Data Hazard: LOAD Instruction



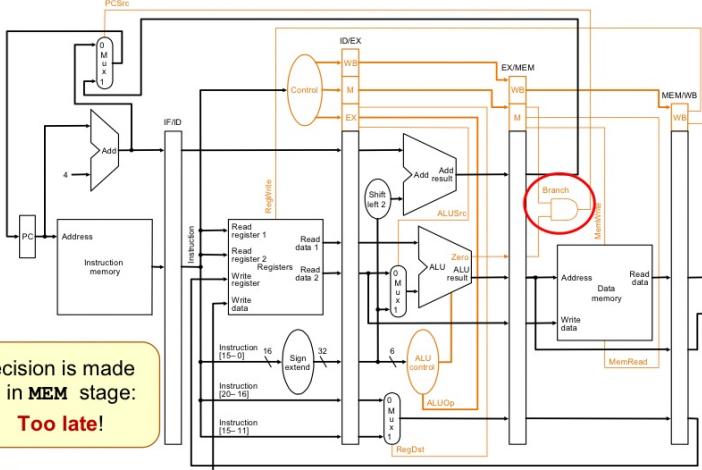
Data Hazard: LOAD Instruction Solution



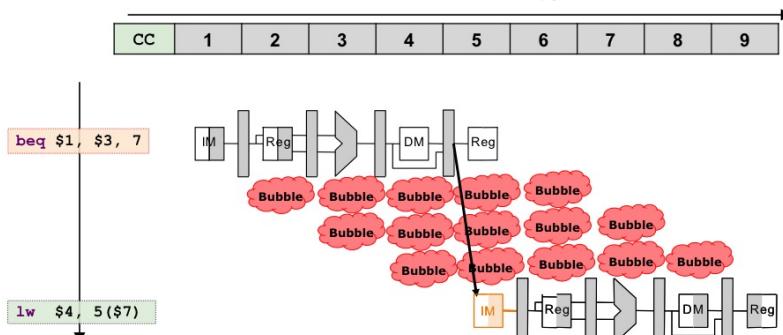
control flow
is the cause

Control hazards (dependencies)

Control Dependency: Why?



Control Hazards: Stall Pipeline?



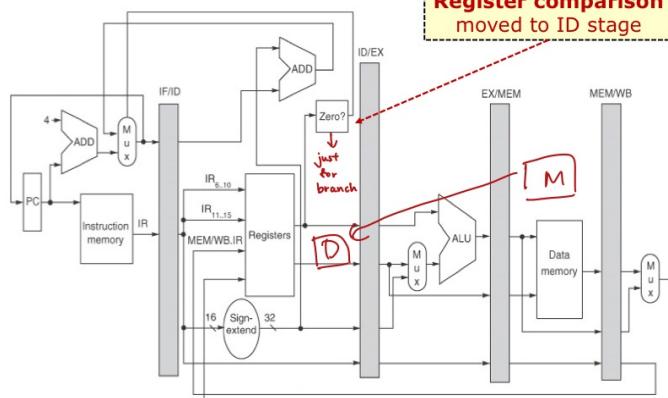
- Wait until the branch outcome is known and then fetch the correct instructions
- ➔ Introduces **3 clock cycles delay**

[L11 - AY2021S1]

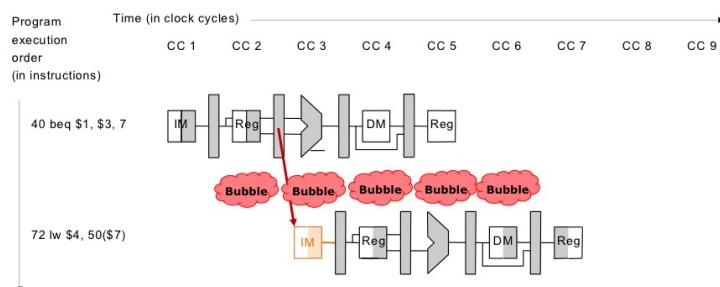
- Solutions :

- ① Early branch resolution
- ② Branch prediction
- ③ Delayed branching

Reduce Stalls: Early Branch(2/3)



Reduce Stalls: Early Branch(3/3)



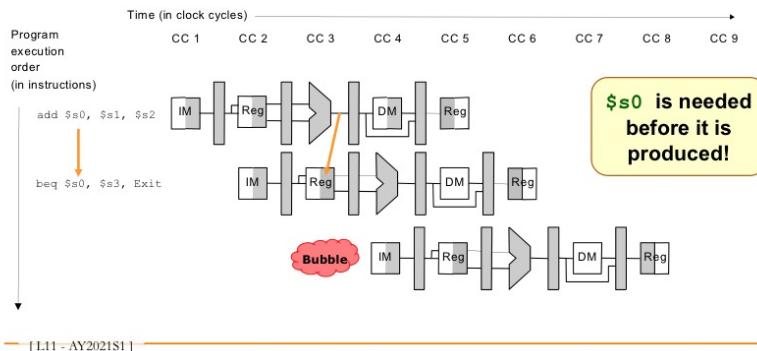
- Wait until the branch decision is known:
 - Then fetch the correct instructions
- Reduced to 1 clock cycle delay

from 3

- Problems with early branch :

Early Branch: Problems(1/3)

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

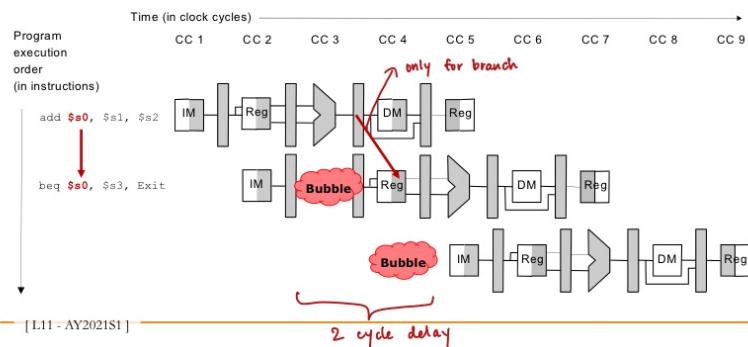


Early Branch: Problems(2/3)

■ Solution:

E/M latch

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



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!

