

Pipelined Processor Design

COE 233

Logic Design and Computer Organization

Dr. Muhamed Mudawar

King Fahd University of Petroleum and Minerals

Presentation Outline

❖ **Serial versus Pipelined Execution**

❖ Pipelined Datapath and Control

❖ Pipeline Hazards

❖ Data Hazards and Forwarding

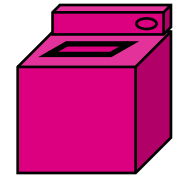
❖ Load Delay, Hazard Detection, and Stall

❖ Control Hazards

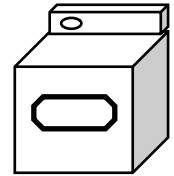
Laundry Example

❖ Laundry Example: Three Stages

1. Wash dirty load of clothes



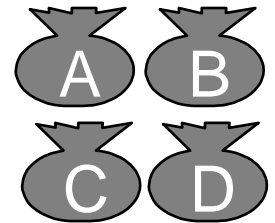
2. Dry wet clothes



3. Fold and put clothes into drawers

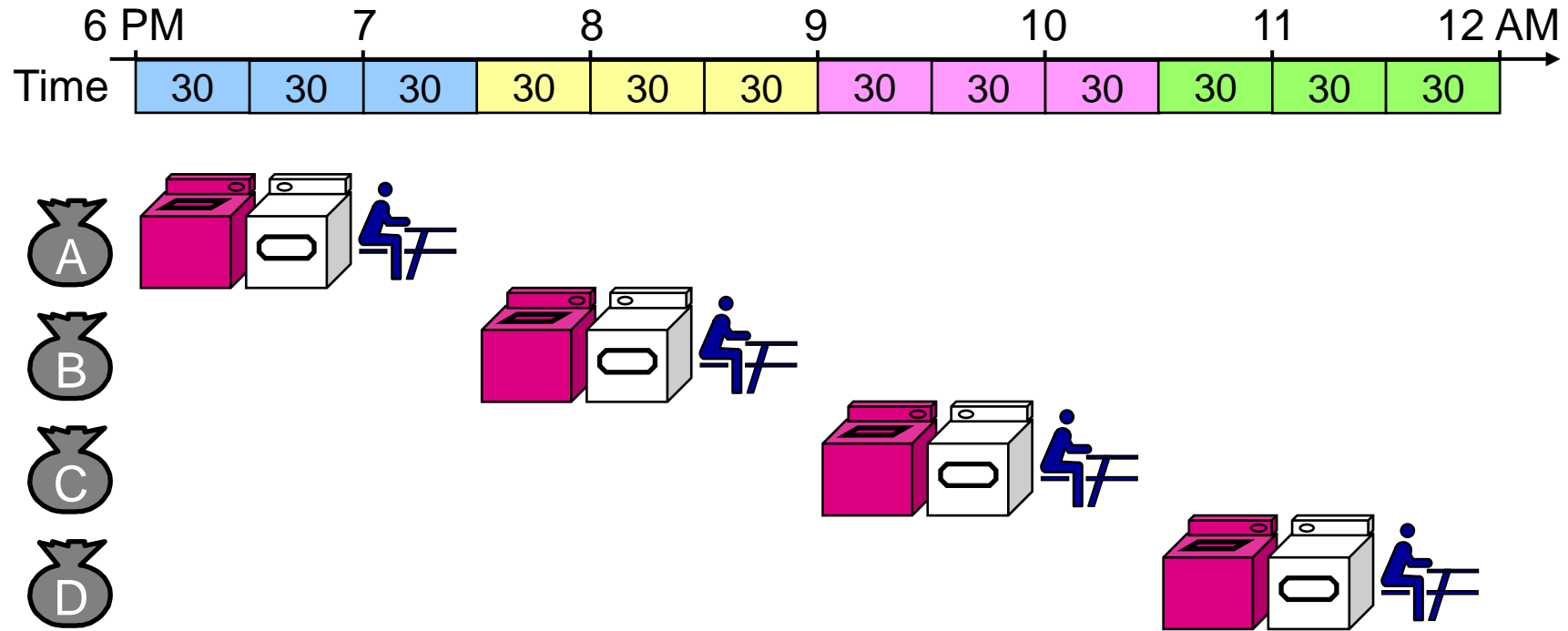


❖ Each stage takes 30 minutes to complete



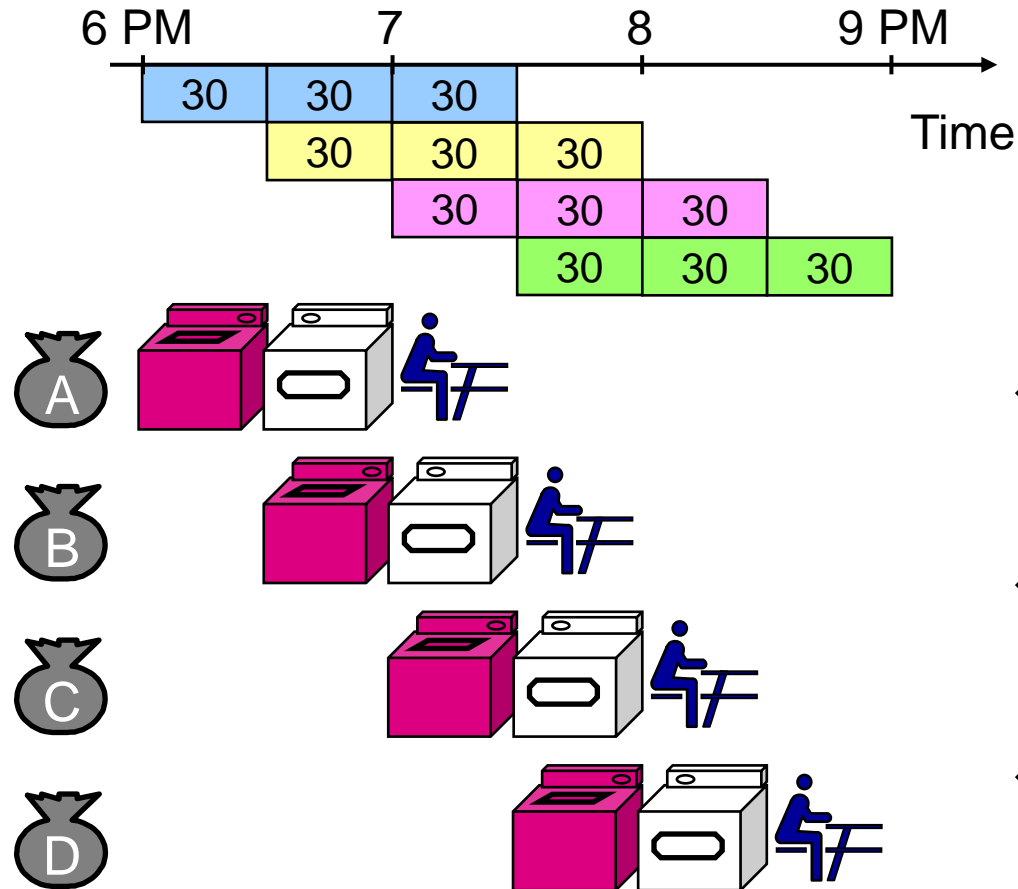
❖ Four loads of clothes to wash, dry, and fold

Sequential Laundry



- ❖ Sequential laundry takes **6 hours** for **4 loads**
- ❖ Intuitively, we can use **pipelining** to speed up laundry

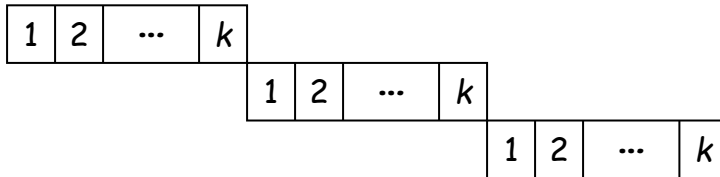
Pipelined Laundry: Start Load ASAP



- ❖ Pipelined laundry takes **3 hours** for **4 loads**
- ❖ Speedup factor is **2** for **4 loads**
- ❖ Time to wash, dry, and fold one load is still the same (90 minutes)

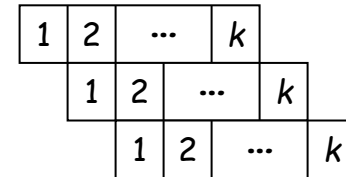
Serial versus Pipelined Execution

- ❖ Consider a task that can be divided into k subtasks
 - ✧ The k subtasks are executed on k different stages
 - ✧ Each subtask requires one time unit
 - ✧ The total execution time of the task is k time units
- ❖ Pipelining is to overlap the execution
 - ✧ The k stages work in parallel on k different tasks
 - ✧ Tasks enter/leave pipeline at the rate of one task per time unit



Serial Execution

One completion every k time units

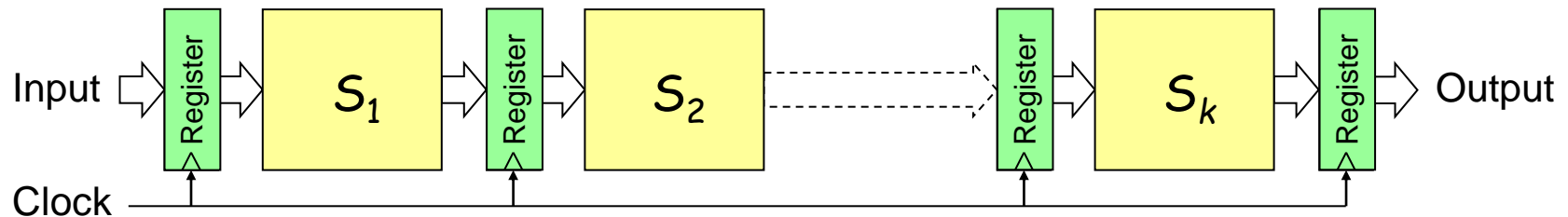


Pipelined Execution

One completion every 1 time unit

Synchronous Pipeline

- ❖ Uses **clocked registers** between stages
- ❖ Upon arrival of a clock edge ...
 - ✧ All registers hold the results of previous stages simultaneously
- ❖ The pipeline stages are **combinational logic** circuits
- ❖ It is desirable to have **balanced** stages
 - ✧ Approximately equal delay in all stages
- ❖ Clock period is determined by the **maximum stage delay**



Pipeline Performance

- ❖ Let τ_i = time delay in stage S_i
- ❖ Clock cycle $\tau = \max(\tau_i)$ is the **maximum stage delay**
- ❖ Clock frequency $f = 1/\tau = 1/\max(\tau_i)$
- ❖ A pipeline can process n tasks in $k + n - 1$ cycles
 - ✧ k cycles are needed to complete the first task
 - ✧ $n - 1$ cycles are needed to complete the remaining $n - 1$ tasks
- ❖ Ideal speedup of a k -stage pipeline over serial execution

$$S_k = \frac{\text{Serial execution in cycles}}{\text{Pipelined execution in cycles}} = \frac{nk}{k + n - 1} \quad S_k \rightarrow k \text{ for large } n$$

MIPS Processor Pipeline

❖ Five stages, one cycle per stage

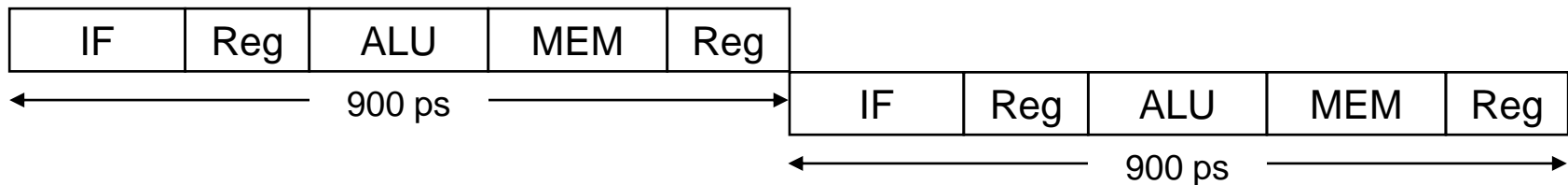
1. IF: **Instruction Fetch** from instruction memory
2. ID: **Instruction Decode**, register read, and J/Br address
3. EX: **Execute** operation or calculate load/store address
4. MEM: **Memory access** for load and store
5. WB: **Write Back** result to register

Single-Cycle vs Pipelined Performance

- ❖ Consider a 5-stage instruction execution in which ...
 - ✧ Instruction fetch = ALU operation = Data memory access = 200 ps
 - ✧ Register read = register write = 150 ps
- ❖ What is the clock cycle of the single-cycle processor?
- ❖ What is the clock cycle of the pipelined processor?
- ❖ What is the speedup factor of pipelined execution?

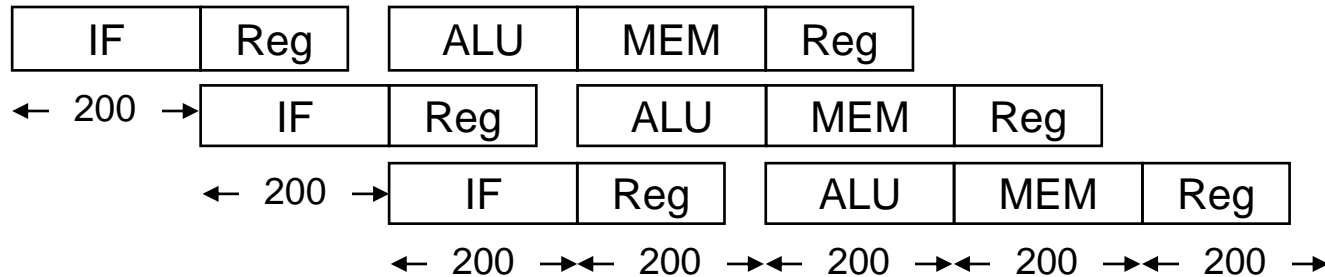
❖ Solution

Single-Cycle Clock = $200+150+200+200+150 = 900 \text{ ps}$



Single-Cycle versus Pipelined - cont'd

❖ Pipelined clock cycle = $\max(200, 150) = 200 \text{ ps}$



❖ CPI for pipelined execution = 1

✧ One instruction completes each cycle (ignoring pipeline fill)

❖ Speedup of pipelined execution = $900 \text{ ps} / 200 \text{ ps} = 4.5$

✧ Instruction count and CPI are equal in both cases

❖ Speedup factor is less than 5 (number of pipeline stage)

✧ Because the pipeline stages are not balanced

Pipeline Performance Summary

- ❖ Pipelining doesn't improve **latency** of a single instruction
- ❖ However, it improves **throughput** of entire workload
 - ✧ Instructions are initiated and completed at a higher rate
- ❖ In a **k-stage** pipeline, **k** instructions operate **in parallel**
 - ✧ Overlapped execution using multiple hardware resources
 - ✧ Potential speedup = **number of pipeline stages k**
- ❖ Pipeline rate is limited by **slowest** pipeline stage
- ❖ Unbalanced lengths of pipeline stages reduces speedup
- ❖ Also, time to **fill** and **drain** pipeline reduces speedup

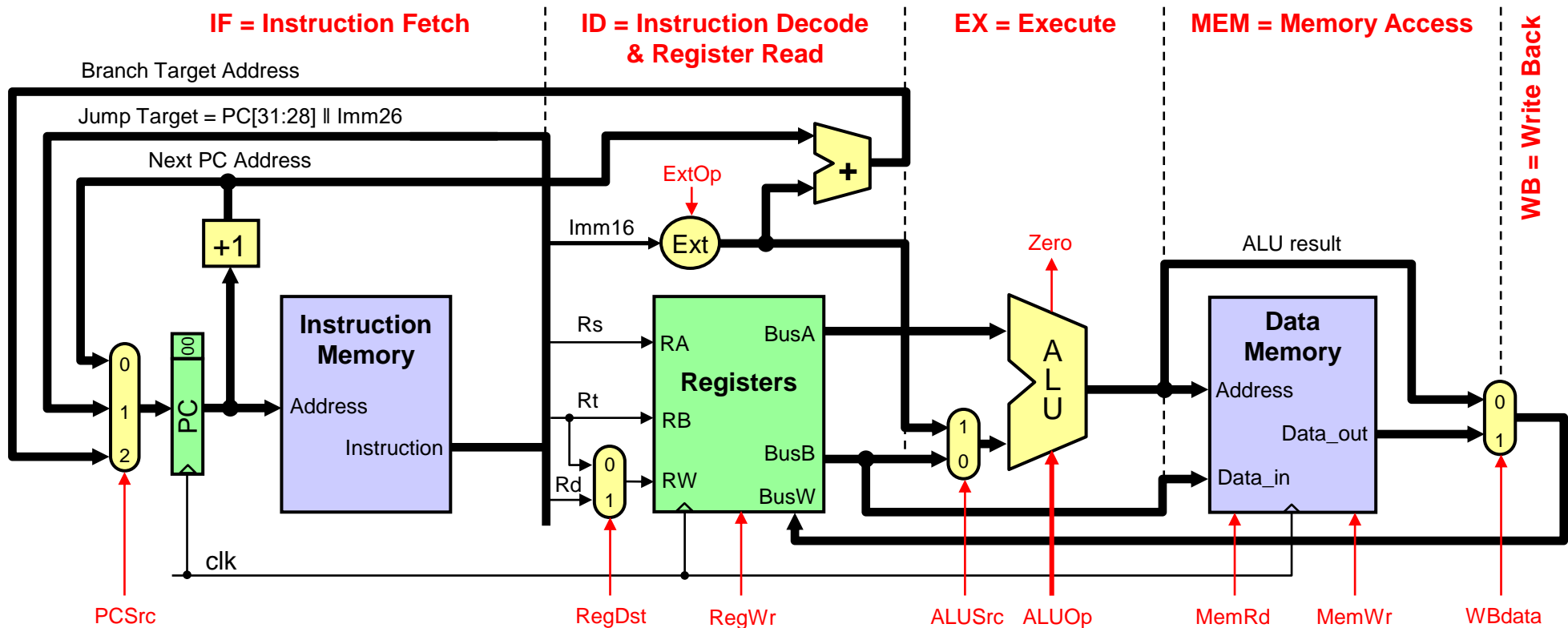
Next . . .

- ❖ Serial versus Pipelined Execution
- ❖ **Pipelined Datapath and Control**
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall
- ❖ Control Hazards

Single-Cycle Datapath

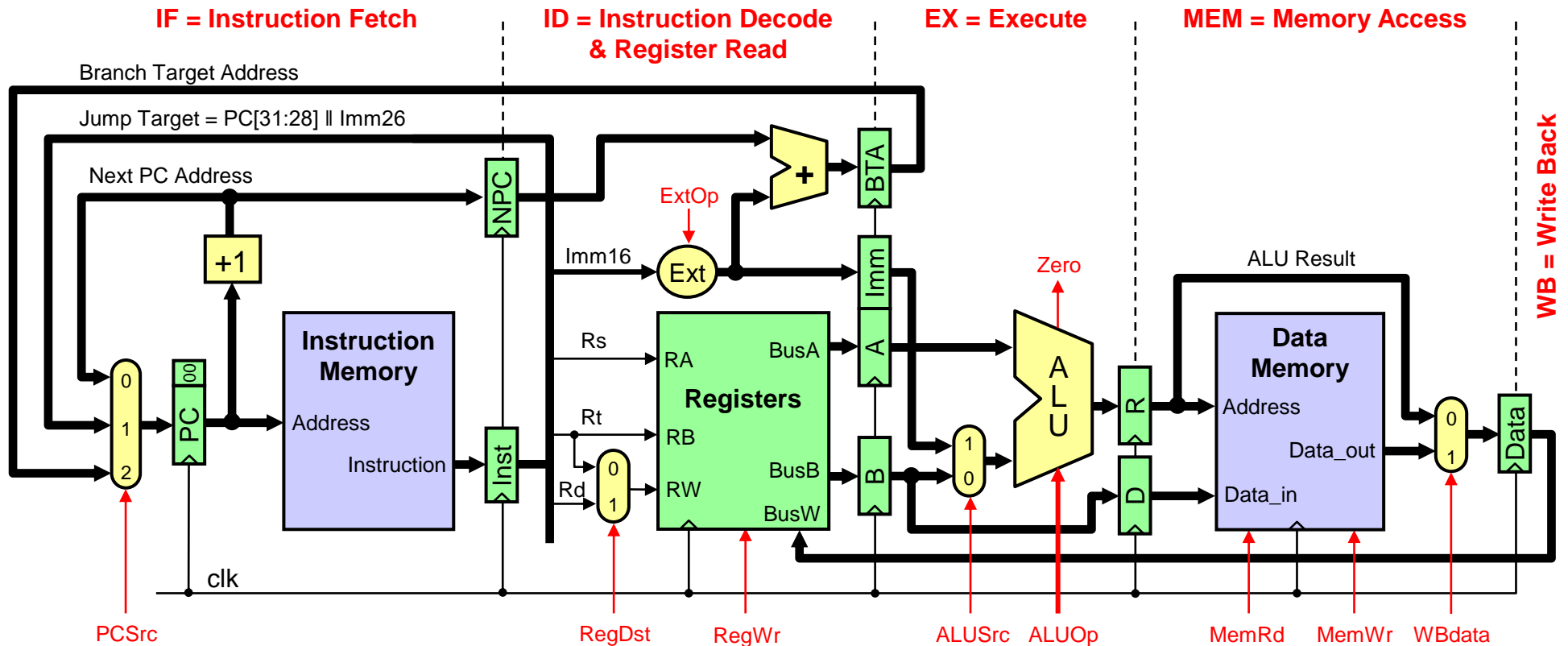
- ❖ Shown below is the single-cycle datapath
- ❖ How to pipeline this single-cycle datapath?

Answer: Introduce **pipeline registers** at end of each stage



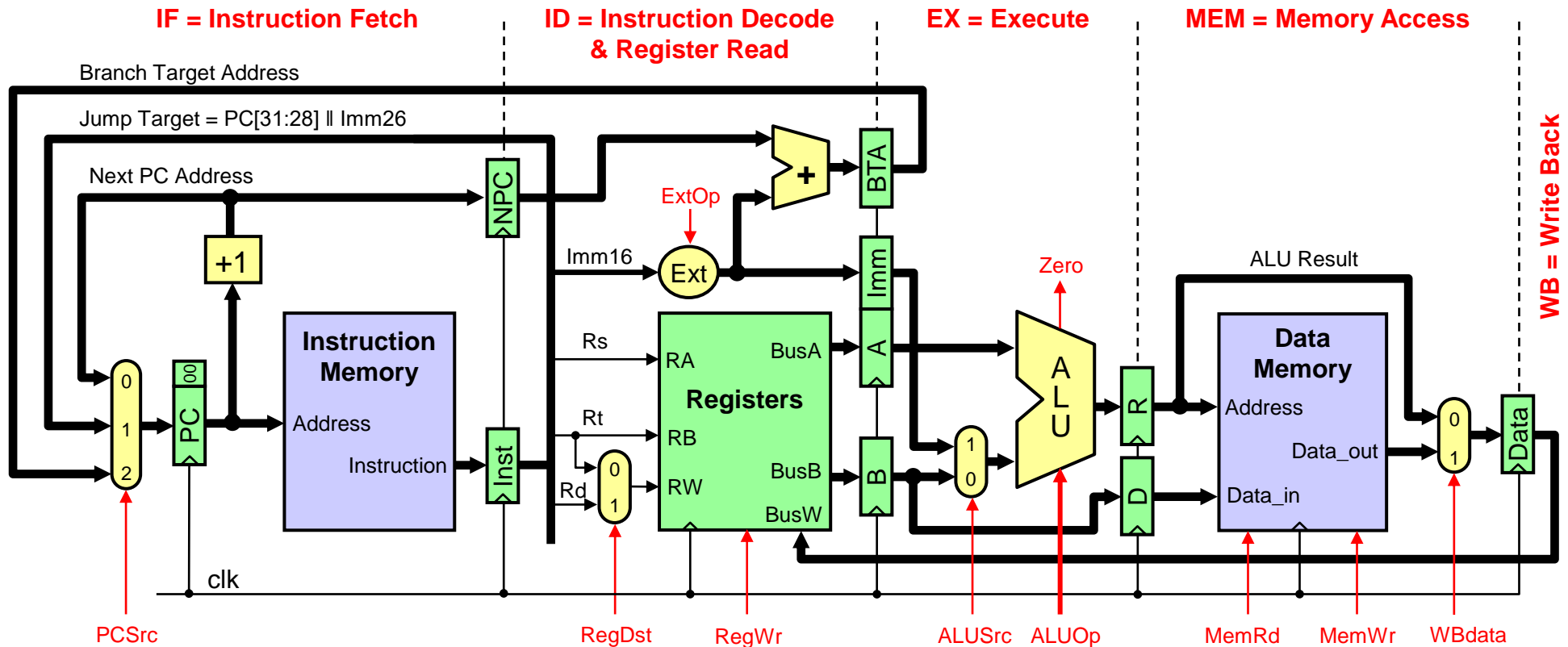
Pipelined Datapath

- ❖ Pipeline registers are shown in **green**, including the **PC**
- ❖ **Same clock edge** updates all pipeline registers and PC
 - ✧ In addition to updating register file and data memory (for store)



Problem with Register Destination

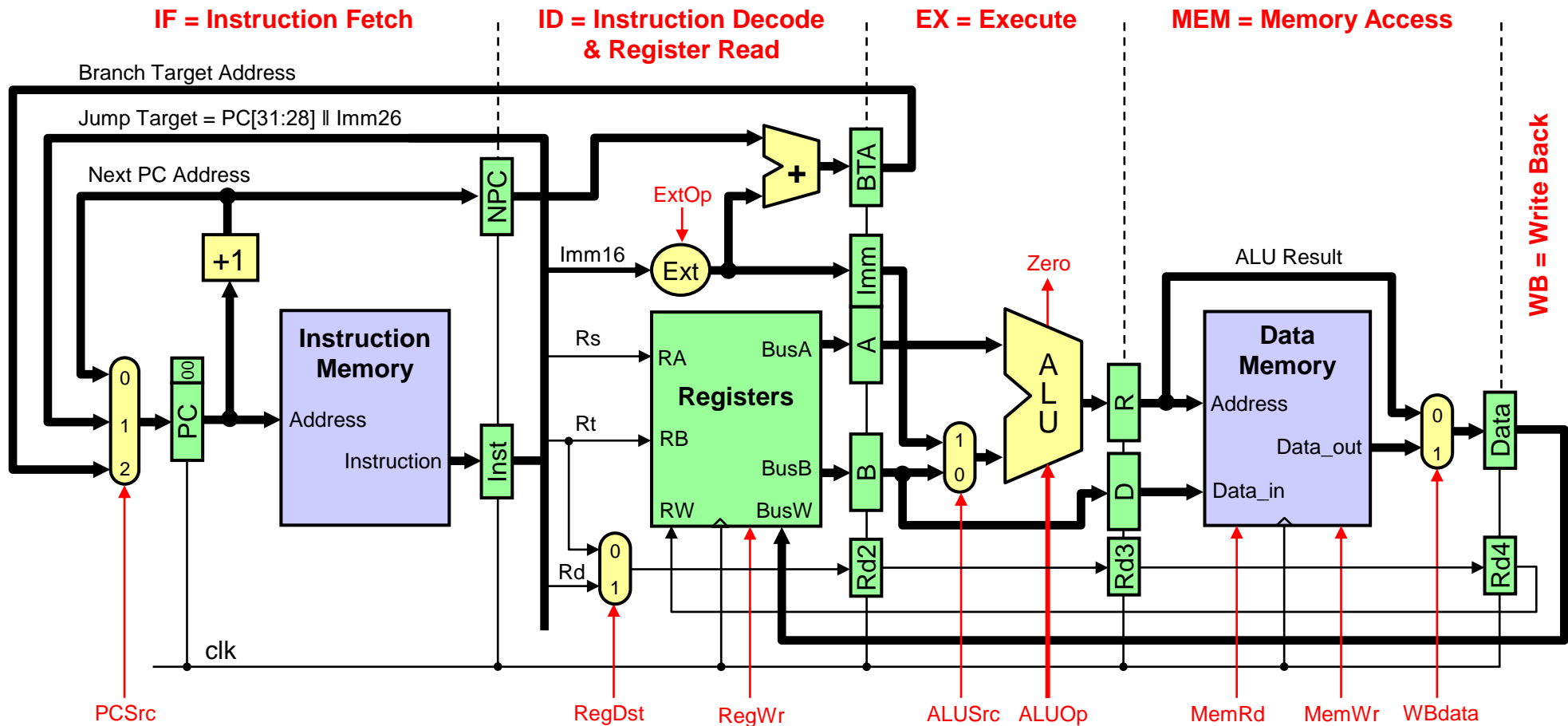
- ❖ Instruction in ID stage is different from the one in WB stage
 - ✧ WB stage is writing to a **different destination register**
 - ✧ Writing the destination register of the instruction in the ID Stage



Pipelining the Destination Register

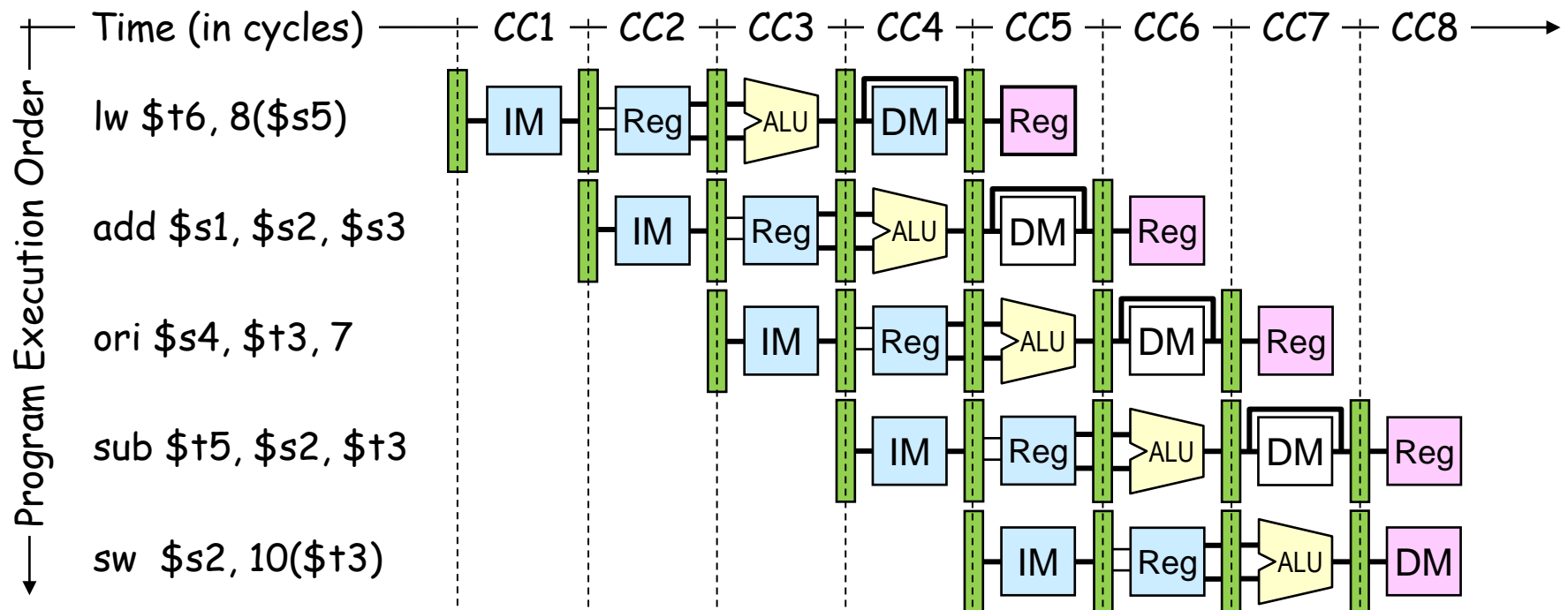
❖ Destination Register should be **pipelined** from ID to WB

✧ The WB stage writes back data knowing the destination register



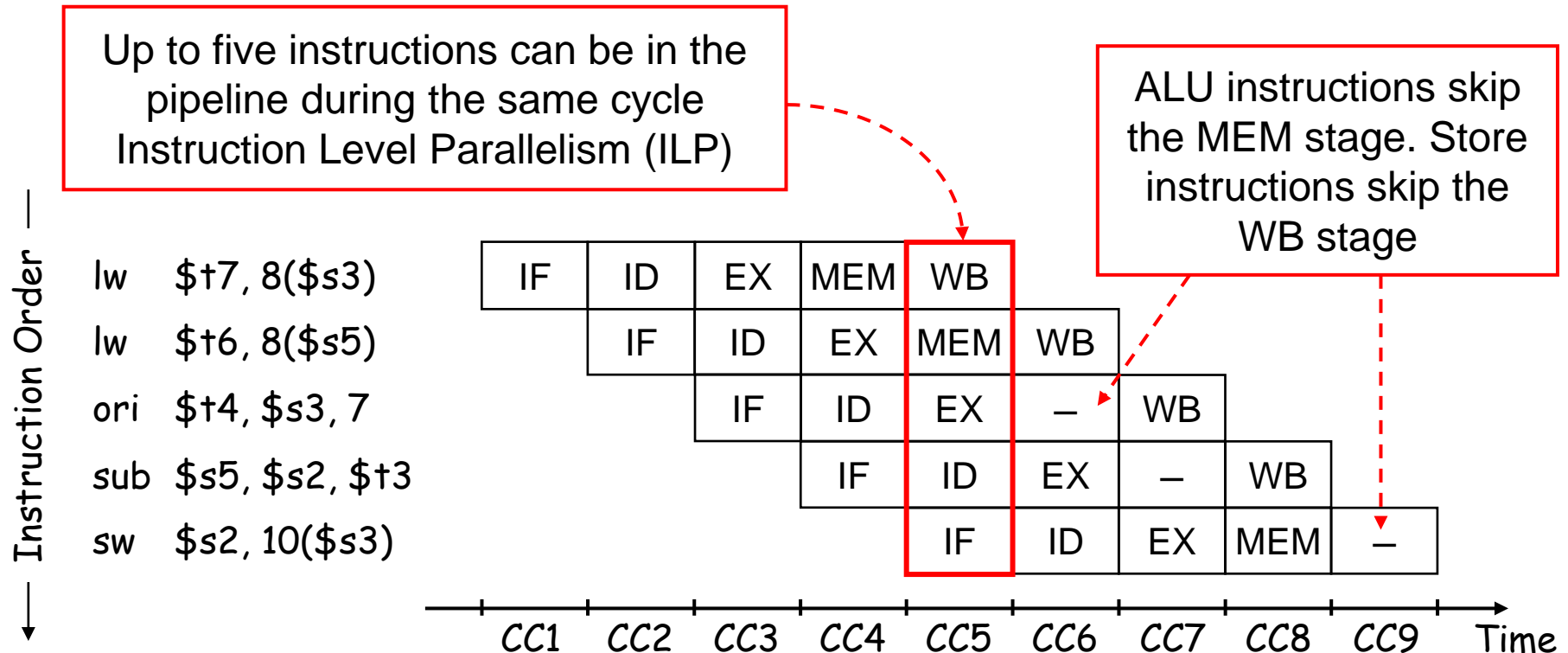
Graphically Representing Pipelines

- ❖ Multiple instruction execution over multiple clock cycles
 - ✧ Instructions are listed in execution order from top to bottom
 - ✧ Clock cycles move from left to right
 - ✧ Figure shows the use of resources at each stage and each cycle

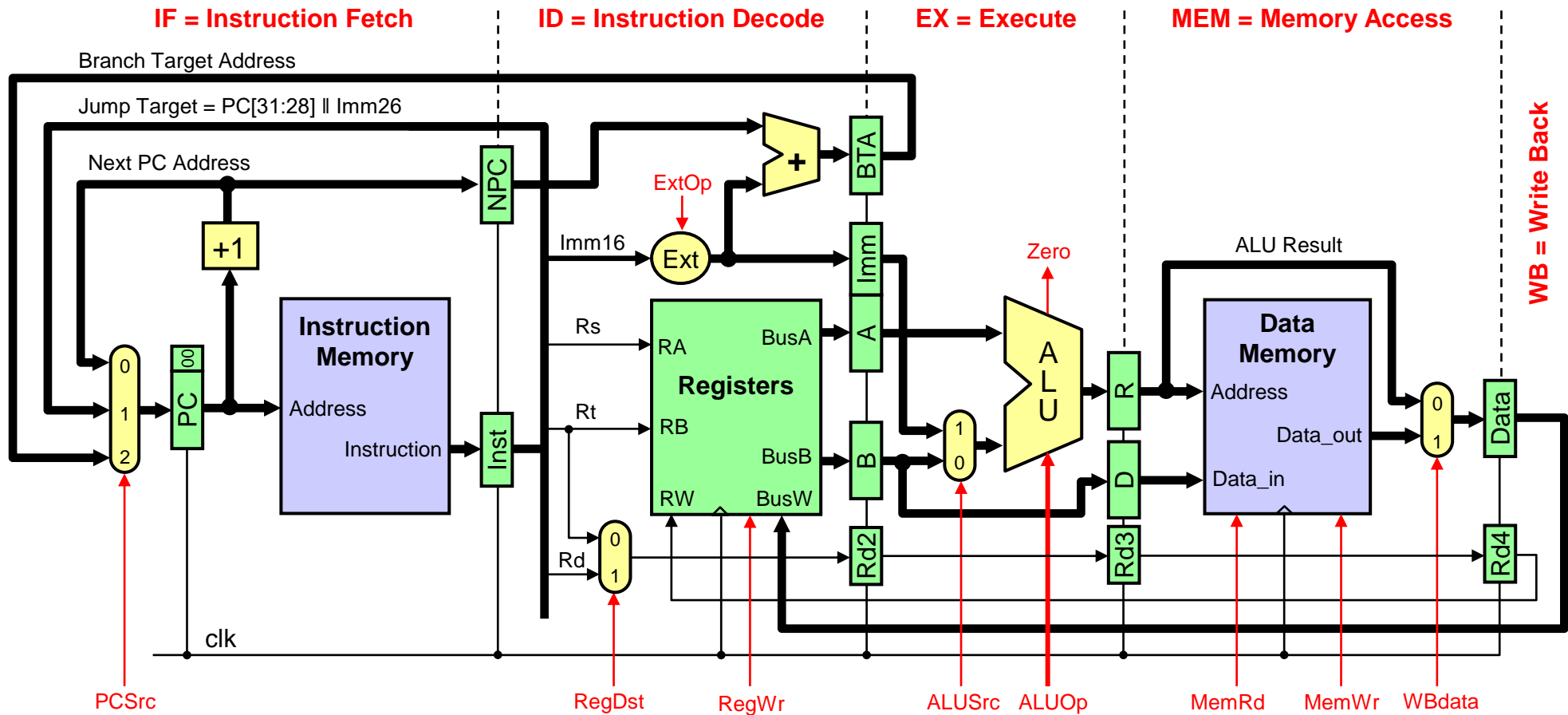


Instruction-Time Diagram

- ❖ Instruction-Time Diagram shows:
 - ✧ Which instruction occupying what stage at each clock cycle
- ❖ Instruction flow is pipelined over the 5 stages

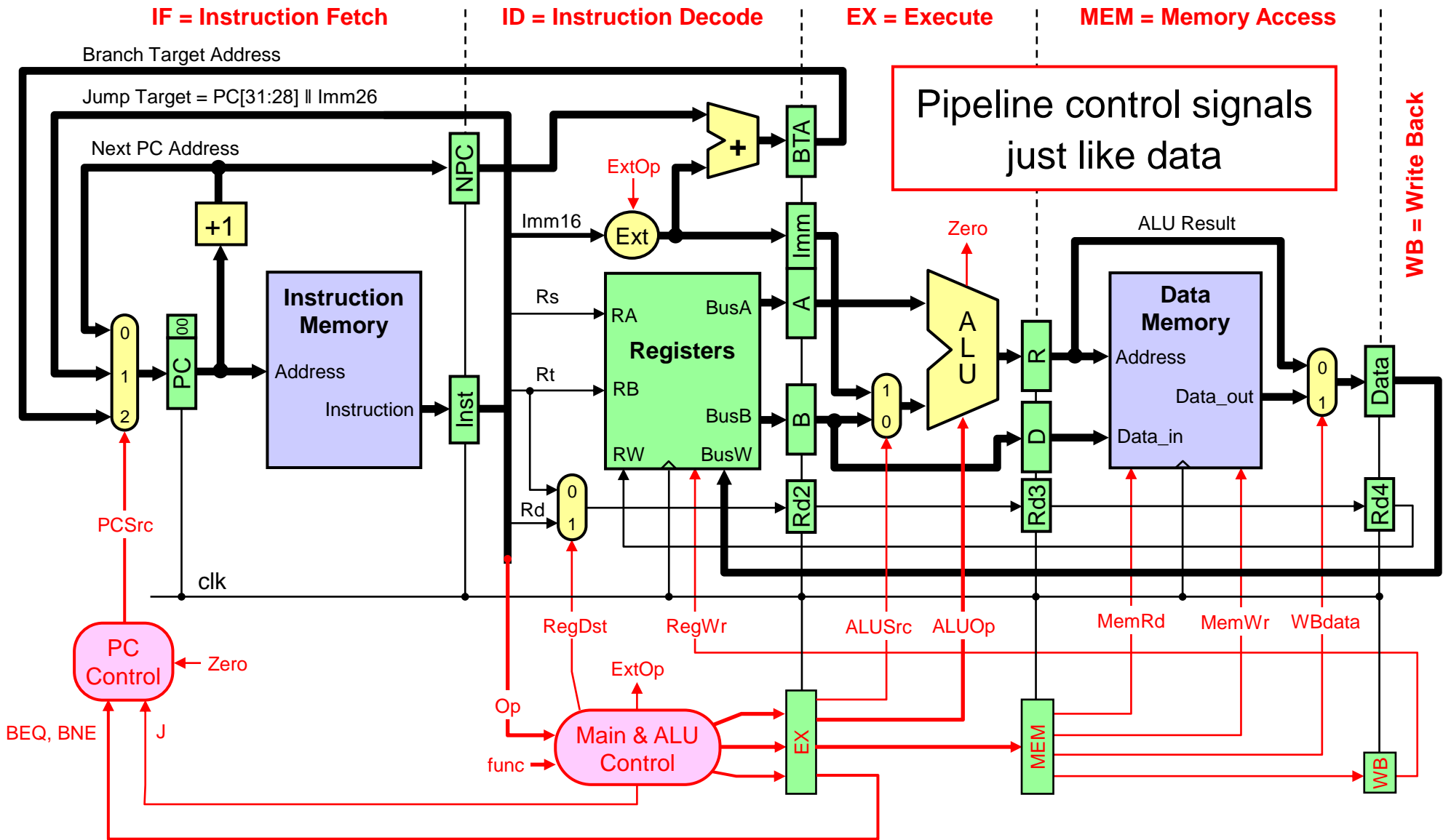


Control Signals



Same control signals used in the single-cycle datapath

Pipelined Control



Pipelined Control - Cont'd

- ❖ ID stage generates all the control signals
- ❖ Pipeline the control signals as the instruction moves
 - ✧ Extend the pipeline registers to include the control signals
- ❖ Each stage uses some of the control signals
 - ✧ Instruction Decode and Register Read
 - Control signals are generated
 - **RegDst** and **ExtOp** are used in this stage, **J** (Jump) is used by PC control
 - ✧ Execution Stage => **ALUSrc, ALUOp, BEQ, BNE**
 - ALU generates **zero** signal for PC control logic (Branch Control)
 - ✧ Memory Stage => **MemRd, MemWr, and WBdata**
 - ✧ Write Back Stage => **RegWr** control signal is used in the last stage

Control Signals Summary

Op	Decode Stage		Execute Stage		Memory Stage			Write Back	PC Control
	RegDst	ExtOp	ALUSrc	ALUOp	MemRd	MemWr	WBdata	RegWr	PCSrc
R-Type	1=Rd	X	0=Reg	func	0	0	0	1	0 = next PC
ADDI	0=Rt	1=sign	1=Imm	ADD	0	0	0	1	0 = next PC
SLTI	0=Rt	1=sign	1=Imm	SLT	0	0	0	1	0 = next PC
ANDI	0=Rt	0=zero	1=Imm	AND	0	0	0	1	0 = next PC
ORI	0=Rt	0=zero	1=Imm	OR	0	0	0	1	0 = next PC
LW	0=Rt	1=sign	1=Imm	ADD	1	0	1	1	0 = next PC
SW	X	1=sign	1=Imm	ADD	0	1	X	0	0 = next PC
BEQ	X	X	0=Reg	SUB	0	0	X	0	0 or 2 = BTA
BNE	X	X	0=Reg	SUB	0	0	X	0	0 or 2 = BTA
J	X	X	X	X	0	0	X	0	1 = jump target

PCSrc = 0 or 2 (BTA) for BEQ and BNE, depending on the zero flag

Next . . .

- ❖ Serial versus Pipelined Execution
- ❖ Pipelined Datapath and Control
- ❖ Pipeline Hazards**
- ❖ Data Hazards and Forwarding**
- ❖ Load Delay, Hazard Detection, and Stall
- ❖ Control Hazards

Pipeline Hazards

❖ **Hazards:** situations that would cause incorrect execution

✧ If next instruction were launched during its designated clock cycle

1. Structural hazards

✧ Caused by resource contention

✧ Using same resource by two instructions during the same cycle

2. Data hazards

✧ An instruction may compute a result needed by next instruction

✧ Data hazards are caused by data dependencies between instructions

3. Control hazards

✧ Caused by instructions that change control flow (branches/jumps)

✧ Delays in changing the flow of control

❖ Hazards complicate pipeline control and limit performance

Structural Hazards

❖ Problem

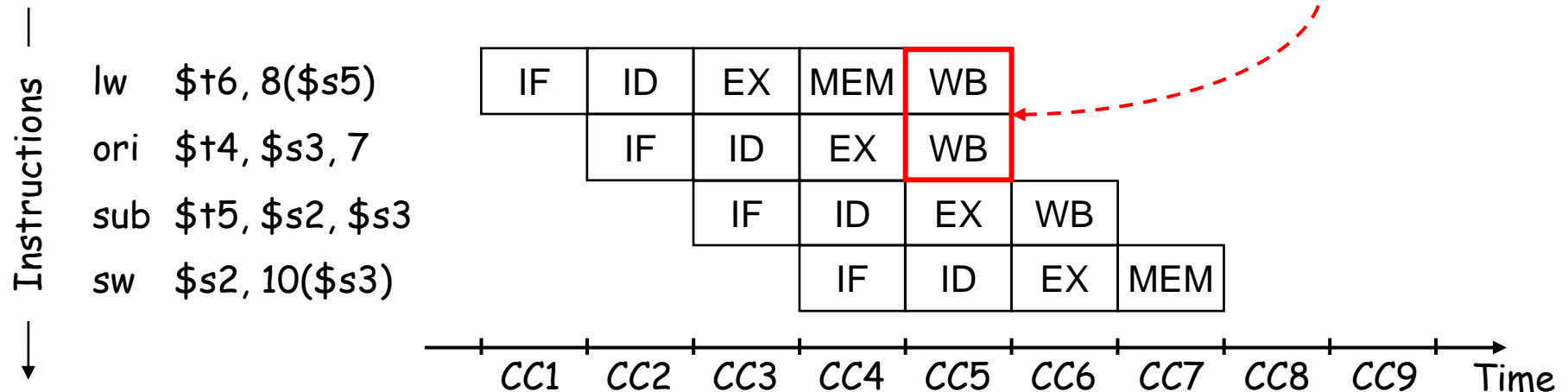
- ✧ Attempt to use the same hardware resource by two different instructions during the same clock cycle

❖ Example

- ✧ Writing back ALU result in stage 4
- ✧ Conflict with writing load data in stage 5

Structural Hazard

Two instructions are attempting to write the register file during same cycle



Resolving Structural Hazards

❖ Serious Hazard:

- ✧ Hazard cannot be ignored

❖ Solution 1: Delay Access to Resource

- ✧ Must have mechanism to delay instruction access to resource
- ✧ Delay all write backs to the register file to stage 5
 - ALU instructions bypass stage 4 (memory) without doing anything

❖ Solution 2: Add more hardware resources (more costly)

- ✧ Add more hardware to eliminate the structural hazard
- ✧ Redesign the register file to have two write ports
 - First write port can be used to write back ALU results in stage 4
 - Second write port can be used to write back load data in stage 5

Data Hazards

- ❖ Dependency between instructions causes a data hazard
- ❖ The dependent instructions are close to each other
 - ✧ Pipelined execution might change the order of operand access

❖ Read After Write – RAW Hazard

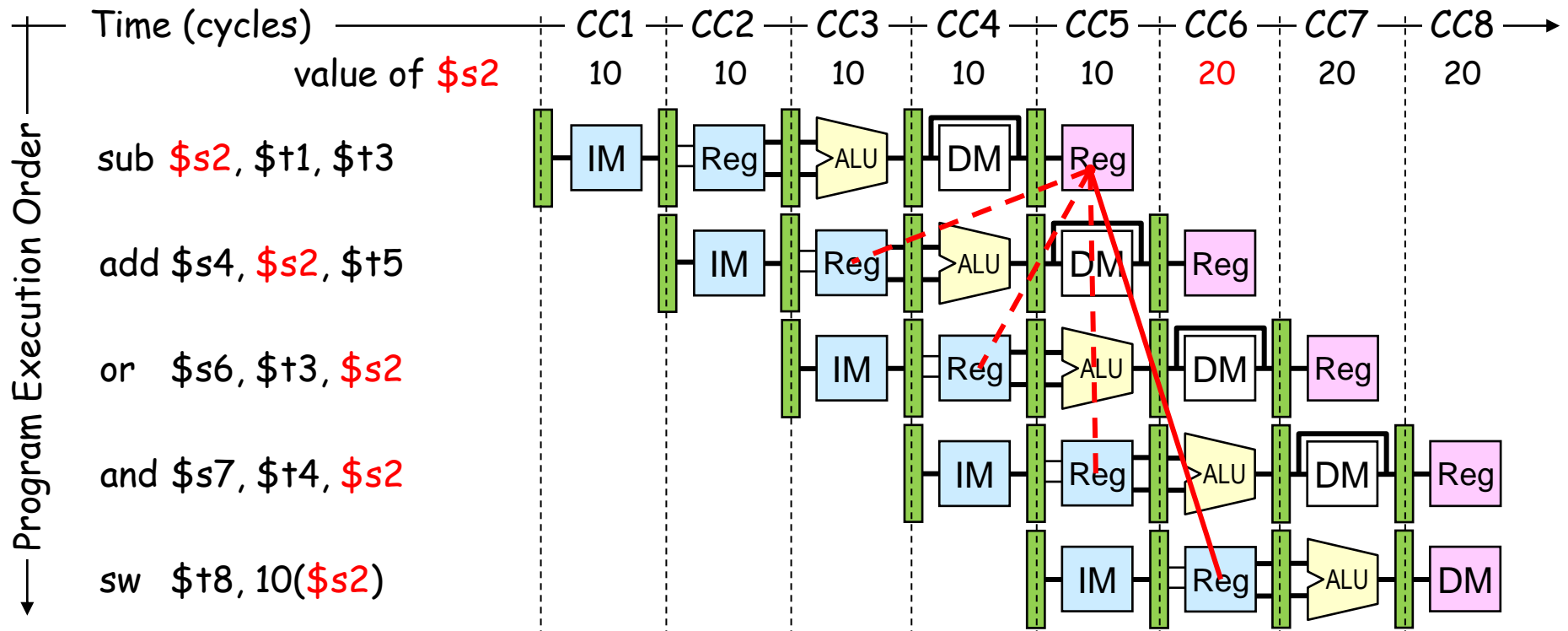
- ✧ Given two instructions *I* and *J*, where *I* comes before *J*
- ✧ Instruction *J* should read an operand after it is written by *I*
- ✧ Called a **data dependence** in compiler terminology

I: add **\$s1**, \$s2, \$s3 # **\$s1 is written**

J: sub \$s4, **\$s1**, \$s3 # **\$s1 is read**

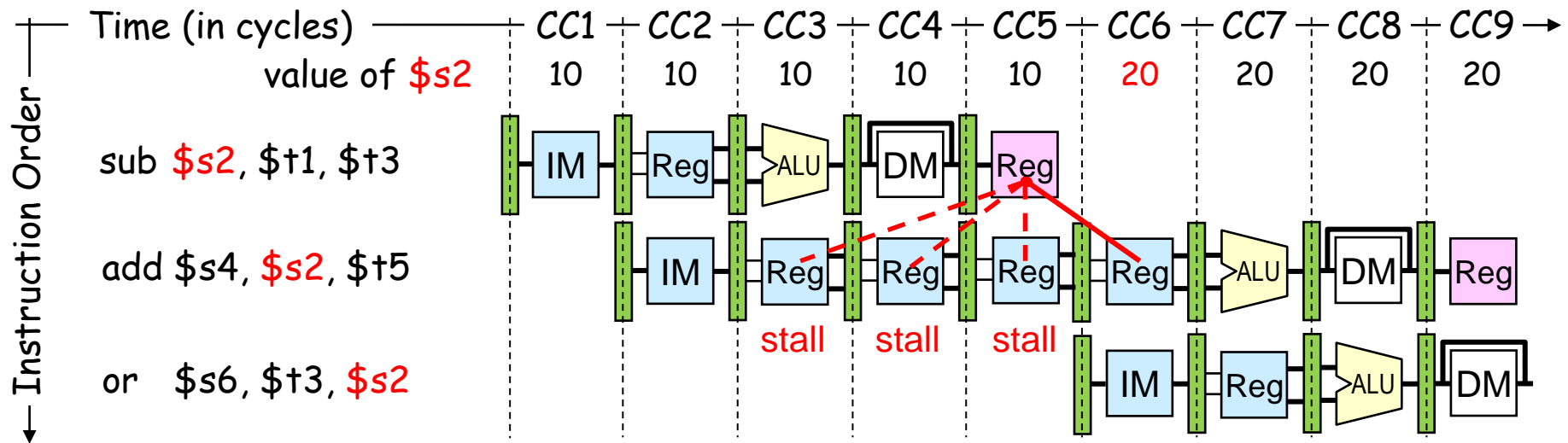
- ✧ Hazard occurs when *J* reads the operand before *I* writes it

Example of a RAW Data Hazard



- ❖ Result of **sub** is needed by **add**, **or**, **and**, & **sw** instructions
- ❖ Instructions **add** & **or** will read **old value** of **\$s2** from reg file
- ❖ During CC5, **\$s2** is written at end of cycle, **old value** is read

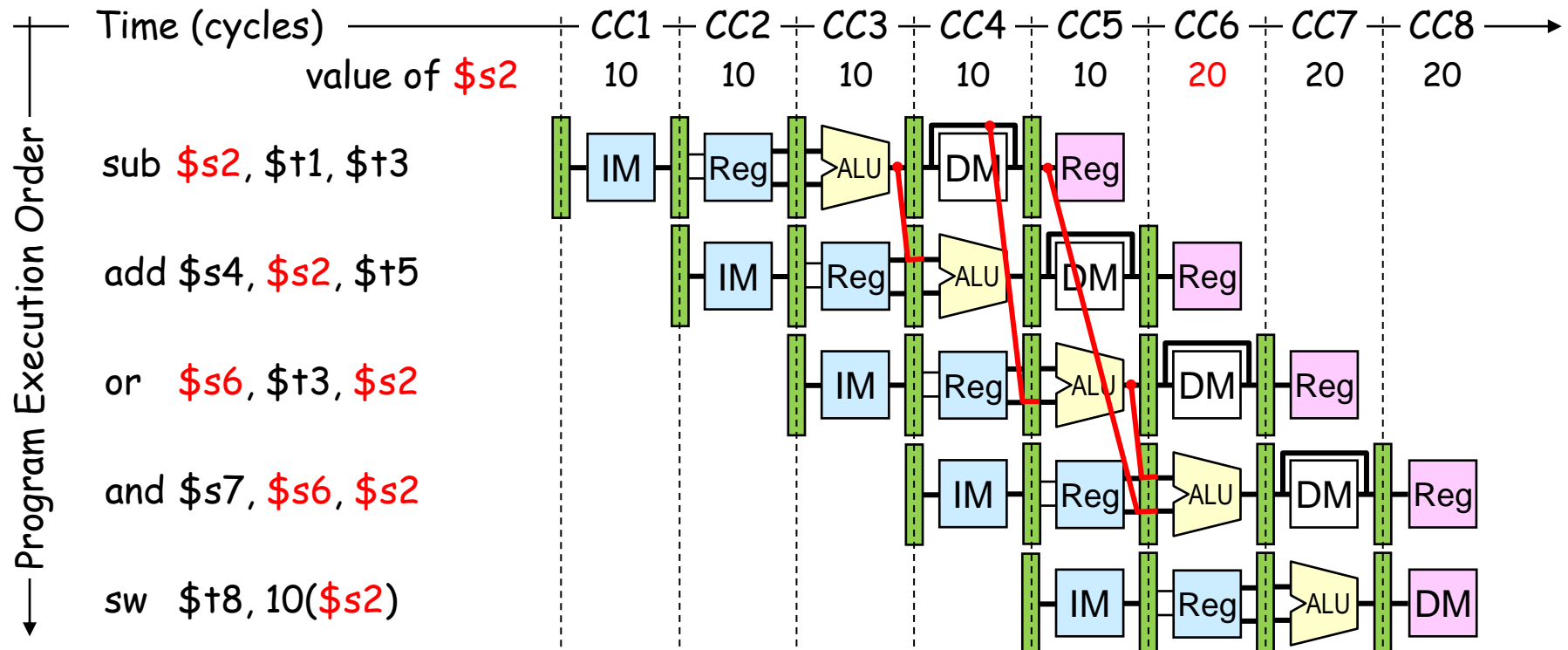
Solution 1: Stalling the Pipeline



- ❖ Three stall cycles during **CC3** thru **CC5** (wasting 3 cycles)
 - ✧ The 3 stall cycles delay the execution of **add** and the fetching of **or**
 - ✧ The 3 stall cycles insert 3 bubbles (No operations) into the ALU
- ❖ The **add** instruction remains in the second stage until **CC6**
- ❖ The **or** instruction is not fetched until **CC6**

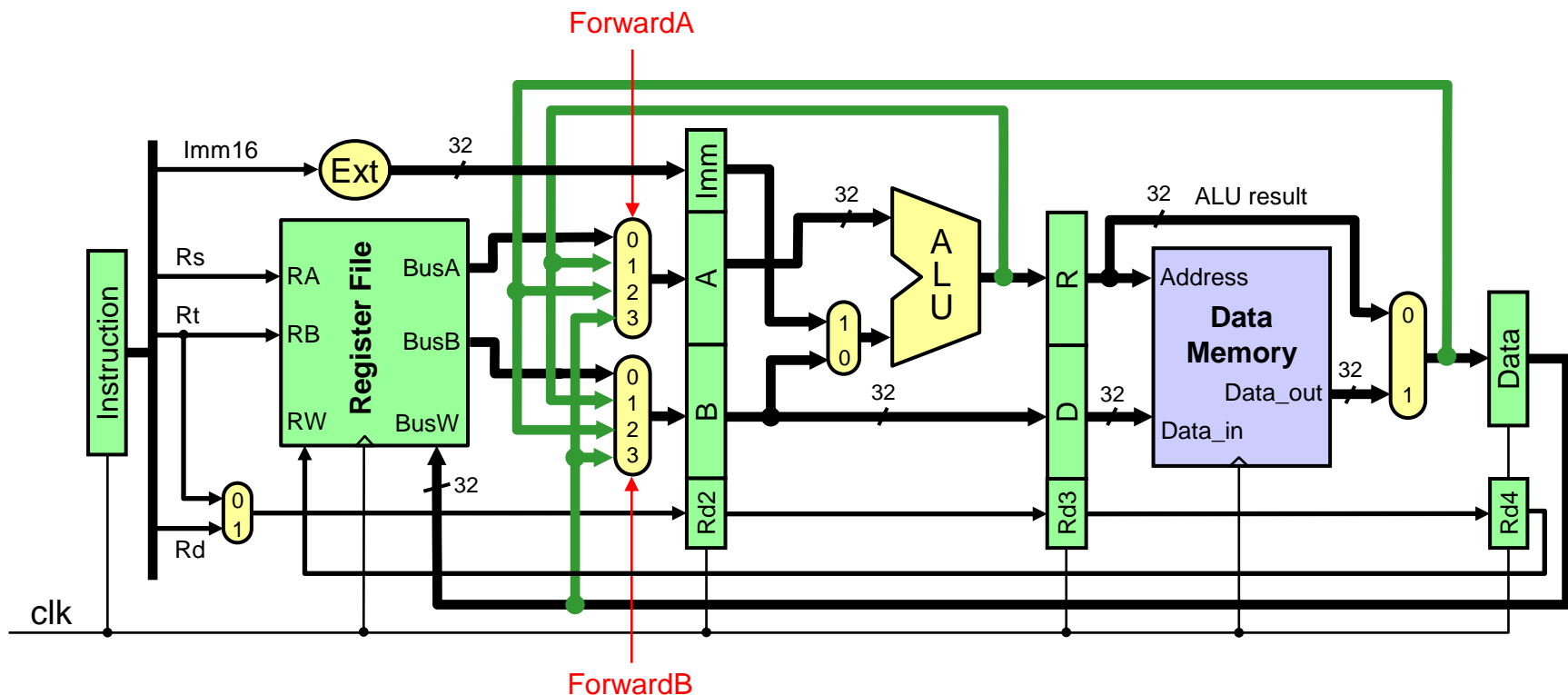
Solution 2: Forwarding ALU Result

- ❖ The **ALU result** is **forwarded** (fed back) to the **ALU input**
 - ✧ No bubbles are inserted into the pipeline and **no cycles are wasted**
- ❖ ALU result is forwarded from **ALU**, **MEM**, and **WB** stages



Implementing Forwarding

- ❖ Two multiplexers added at the inputs of A & B registers
 - ✧ Data from **ALU stage**, **MEM stage**, and **WB stage** is fed back
- ❖ Two signals: **ForwardA** and **ForwardB** to control forwarding



Forwarding Control Signals

Signal	Explanation
ForwardA = 0	First ALU operand comes from register file = Value of (Rs)
ForwardA = 1	Forward result of previous instruction to A (from ALU stage)
ForwardA = 2	Forward result of 2 nd previous instruction to A (from MEM stage)
ForwardA = 3	Forward result of 3 rd previous instruction to A (from WB stage)
ForwardB = 0	Second ALU operand comes from register file = Value of (Rt)
ForwardB = 1	Forward result of previous instruction to B (from ALU stage)
ForwardB = 2	Forward result of 2 nd previous instruction to B (from MEM stage)
ForwardB = 3	Forward result of 3 rd previous instruction to B (from WB stage)

Forwarding Example

Instruction sequence:

lw **\$t4**, 4(\$t0)

ori **\$t7**, \$t1, 2

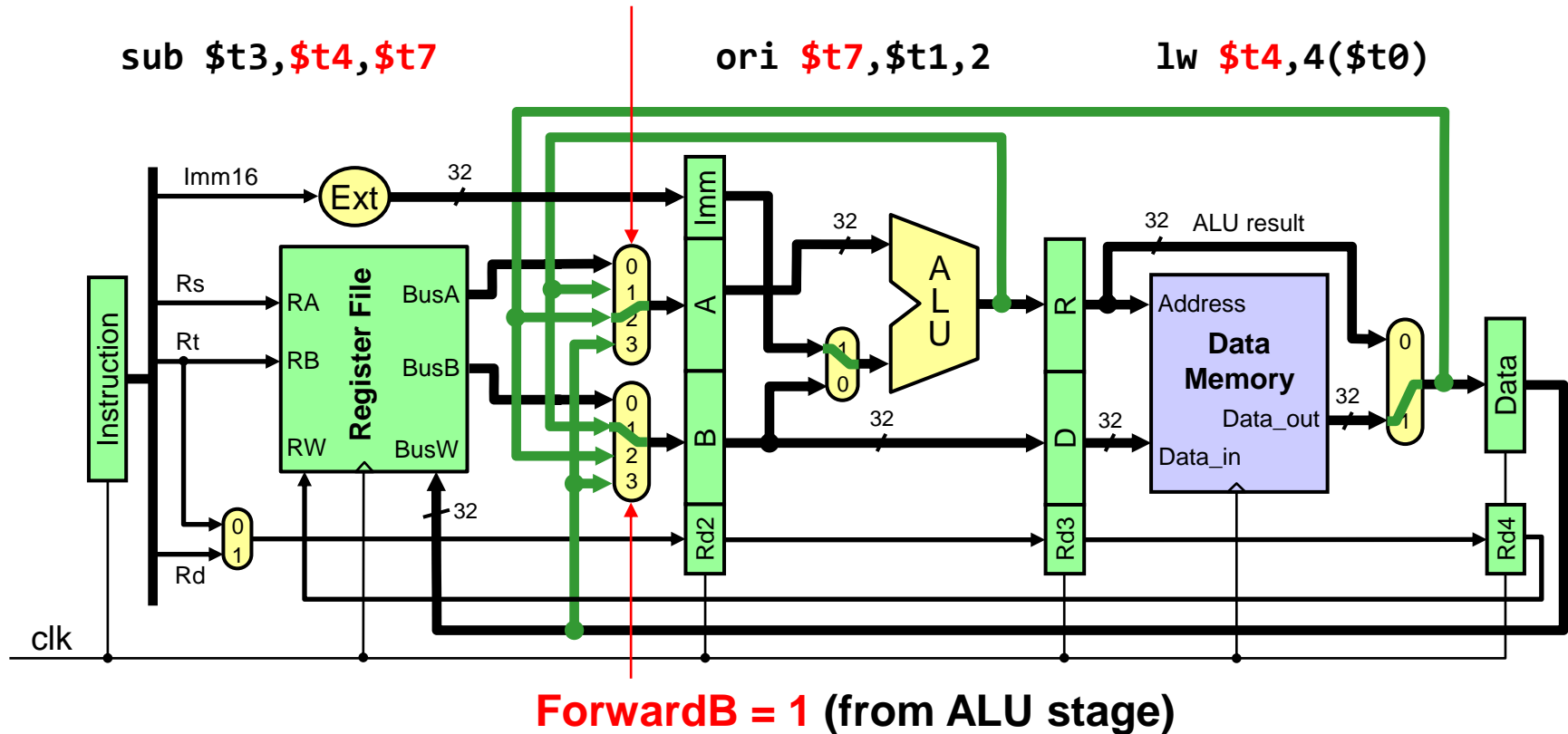
sub **\$t3**, **\$t4**, **\$t7**

When **sub** instruction in ID stage

ori will be in the ALU stage

lw will be in the MEM stage

ForwardA = 2 (from MEM stage)



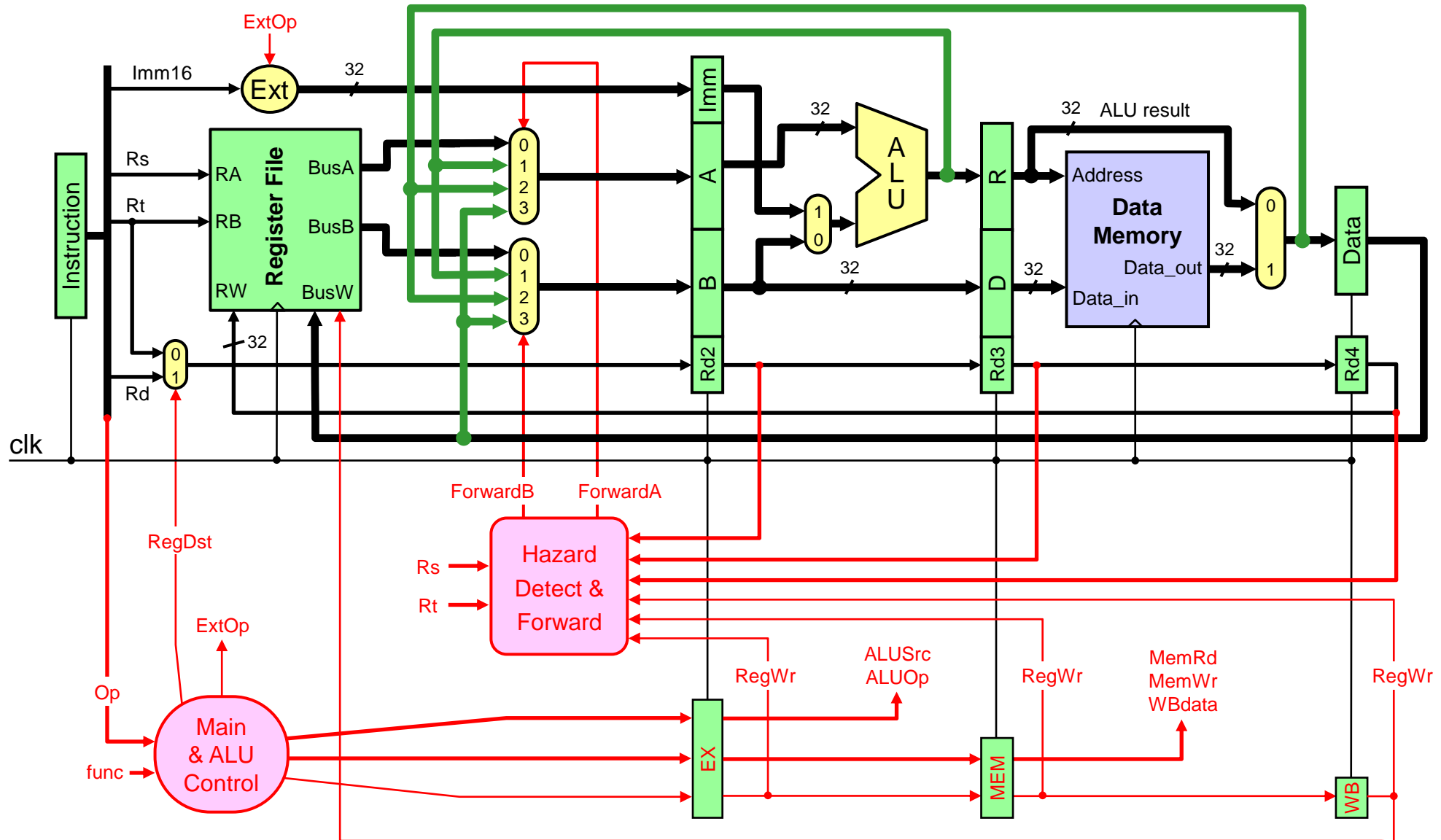
RAW Hazard Detection

- ❖ **Current** instruction is being decoded in the **Decode** stage
- ❖ **Previous** instruction is in the **Execute** stage
- ❖ **Second previous** instruction is in the **Memory** stage
- ❖ **Third previous** instruction is in the **Write Back** stage

```
If      ((Rs != 0) and (Rs == Rd2) and (EX.RegWr)) ForwardA = 1
Else if ((Rs != 0) and (Rs == Rd3) and (MEM.RegWr)) ForwardA = 2
Else if ((Rs != 0) and (Rs == Rd4) and (WB.RegWr)) ForwardA = 3
Else    ForwardA = 0
```

```
If      ((Rt != 0) and (Rt == Rd2) and (EX.RegWr)) ForwardB = 1
Else if ((Rt != 0) and (Rt == Rd3) and (MEM.RegWr)) ForwardB = 2
Else if ((Rt != 0) and (Rt == Rd4) and (WB.RegWr)) ForwardB = 3
Else    ForwardB = 0
```

Hazard Detecting and Forwarding Logic



Next . . .

- ❖ Serial versus Pipelined Execution
- ❖ Pipelined Datapath and Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall**
- ❖ Control Hazards

Load Delay

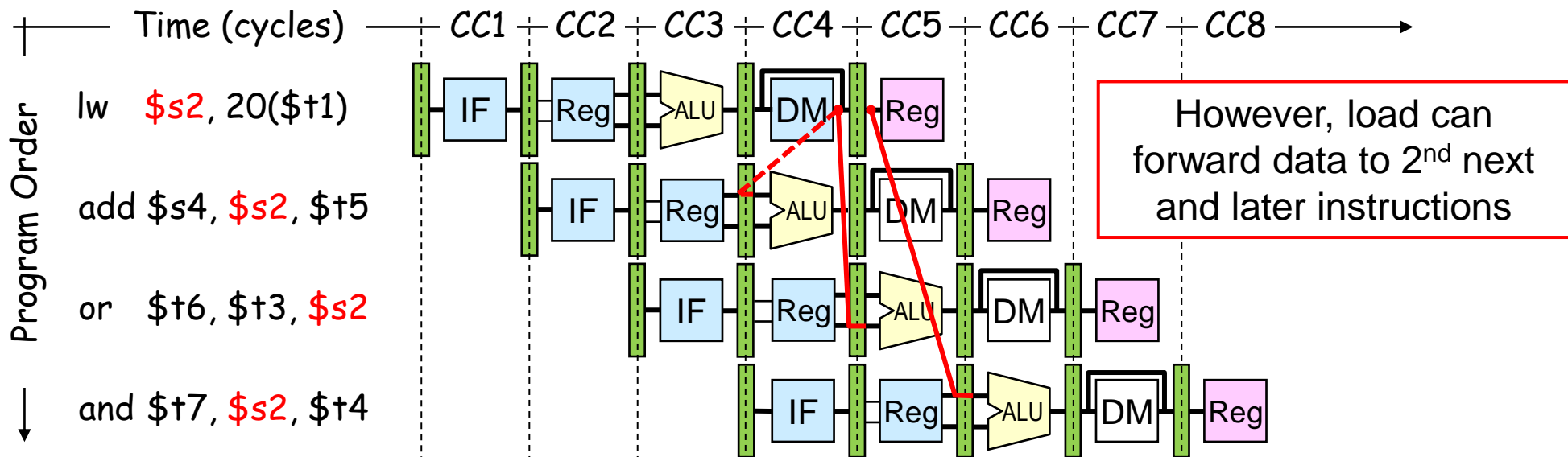
❖ Unfortunately, not all data hazards can be forwarded

✧ **Load** has a delay that cannot be eliminated by forwarding

❖ In the example shown below ...

✧ The **LW** instruction does not read data until end of CC4

✧ Cannot forward data to **ADD** at end of CC3 - **NOT possible**



Detecting RAW Hazard after Load

❖ Detecting a RAW hazard after a Load instruction:

- ✧ The **load** instruction will be in the **EX** stage
- ✧ Instruction that depends on the load data is in the decode stage

❖ Condition for stalling the pipeline

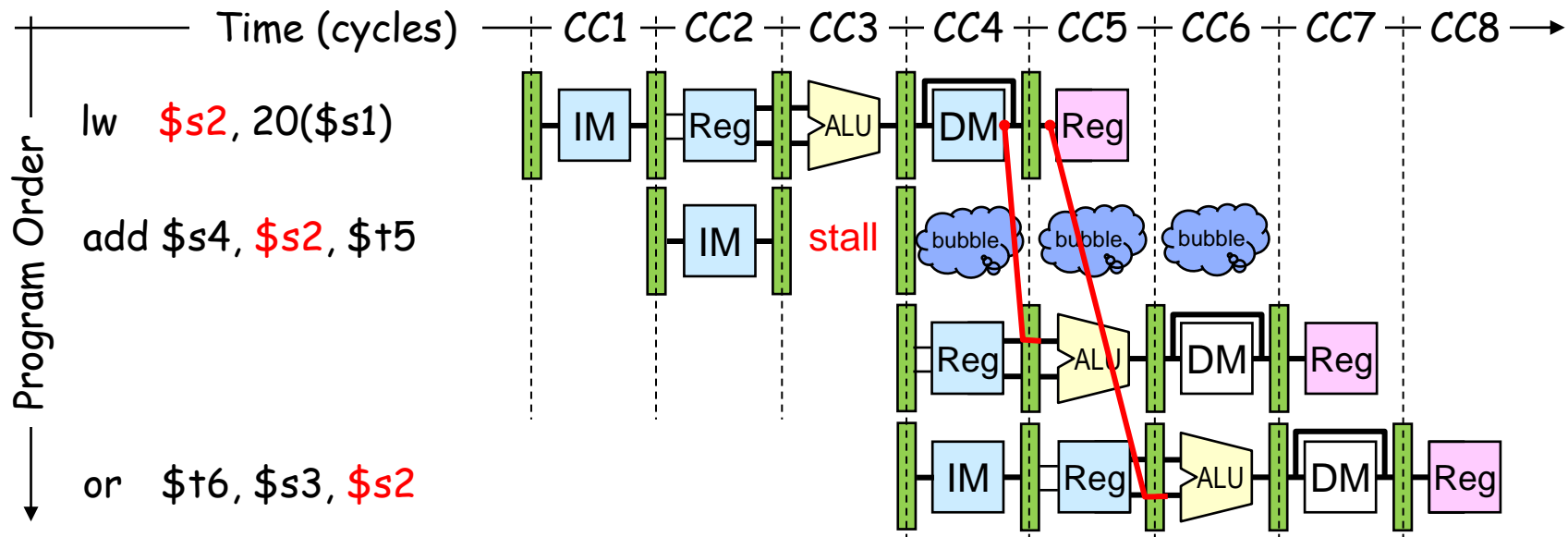
```
if ((EX.MemRd == 1) // Detect Load in EX stage  
    and (ForwardA==1 or ForwardB==1)) Stall // RAW Hazard
```

❖ Insert a **bubble** into the EX stage after a load instruction

- ✧ Bubble is a **no-op** that wastes one clock cycle
- ✧ Delays the dependent instruction after load by one cycle
 - Because of RAW hazard

Stall the Pipeline for one Cycle

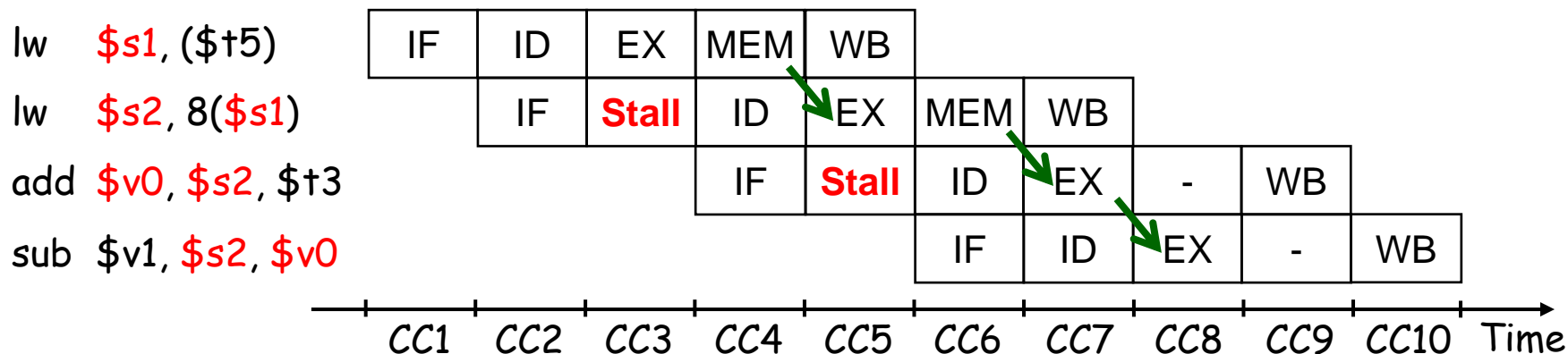
- ❖ **ADD** instruction depends on **LW** → stall at CC3
 - ✧ Allow **Load** instruction in **ALU** stage to proceed
 - ✧ Freeze **PC** and **Instruction** registers (NO instruction is fetched)
 - ✧ Introduce a **bubble** into the **ALU** stage (bubble is a NO-OP)
- ❖ **Load** can forward data to next instruction after delaying it



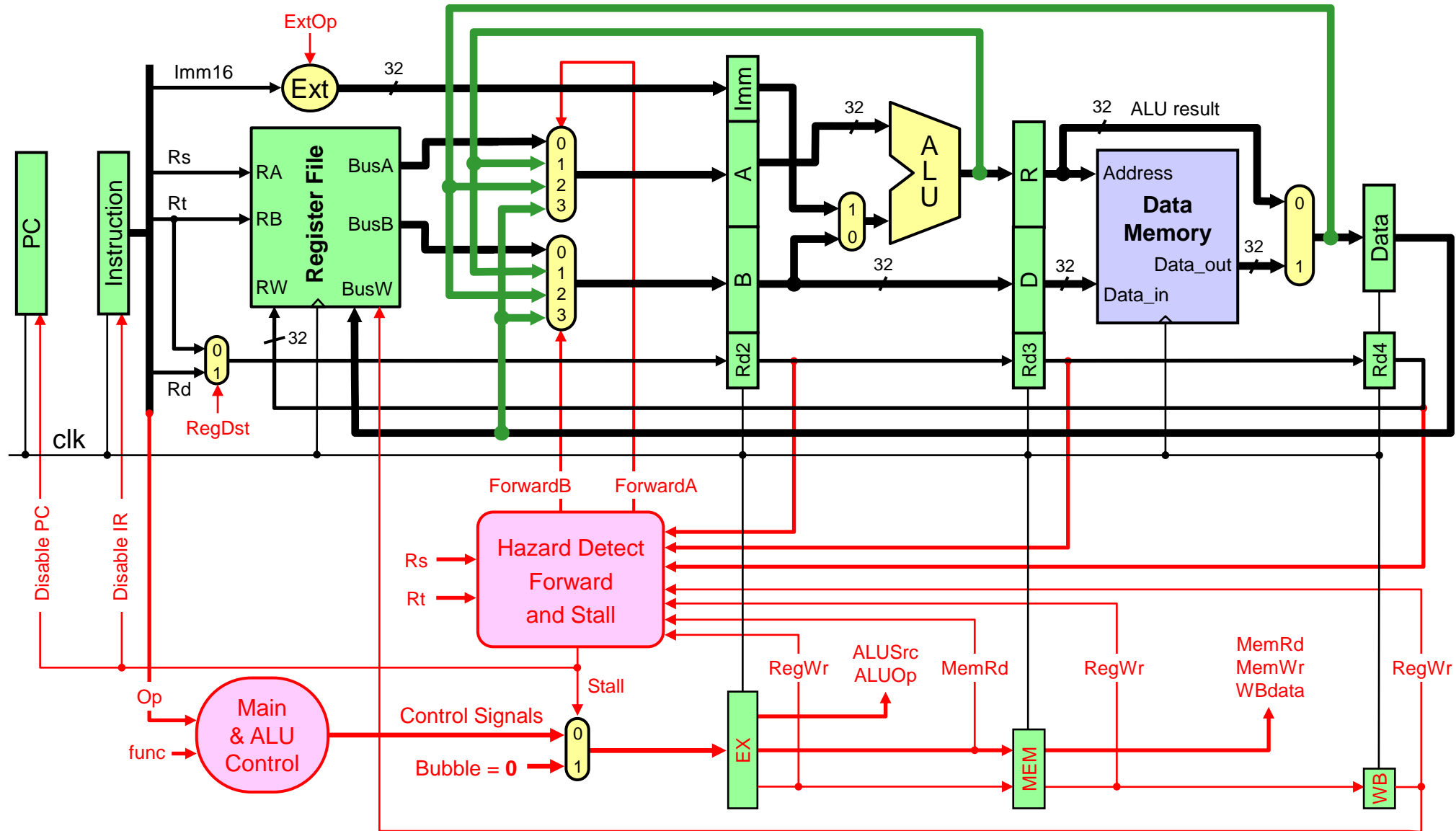
Showing Stall Cycles

- ❖ Stall cycles can be shown on instruction-time diagram
- ❖ Hazard is detected in the Decode stage
- ❖ Stall indicates that instruction is delayed
- ❖ Instruction fetching is also delayed after a stall
- ❖ Example:

Data forwarding is shown using **green arrows**



Hazard Detecting and Forwarding Logic



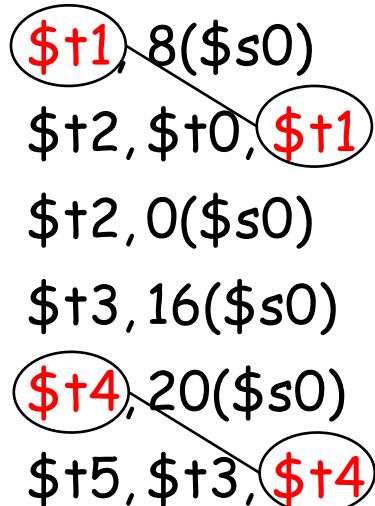
Code Scheduling to Avoid Stalls

- ❖ Compilers reorder code in a way to avoid load stalls
- ❖ Consider the translation of the following statements:

A = B + C; D = E - F; // A thru F are in Memory

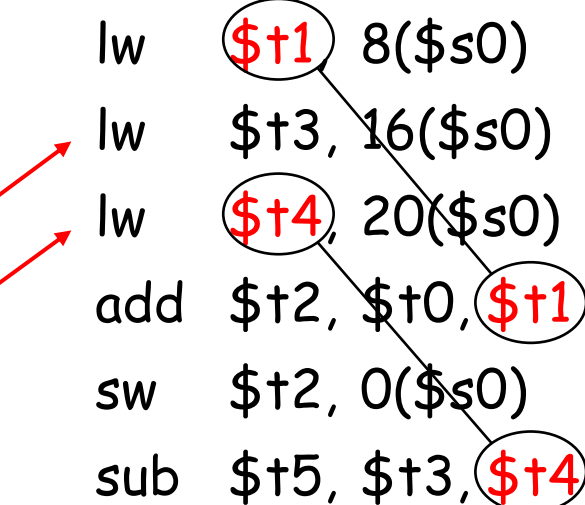
❖ Slow code:

```
lw    $t0, 4($s0)    # &B = 4($s0)
lw    $t1, 8($s0)    # &C = 8($s0)
add   $t2, $t0, $t1   # stall cycle
sw    $t2, 0($s0)     # &A = 0($s0)
lw    $t3, 16($s0)    # &E = 16($s0)
lw    $t4, 20($s0)    # &F = 20($s0)
sub   $t5, $t3, $t4   # stall cycle
sw    $t5, 12($s0)    # &D = 12($s0)
```



❖ Fast code: No Stalls

```
lw    $t0, 4($s0)
lw    $t1, 8($s0)
lw    $t3, 16($s0)
lw    $t4, 20($s0)
add   $t2, $t0, $t1
sw    $t2, 0($s0)
sub   $t5, $t3, $t4
sw    $t5, 12($s0)
```



Name Dependence: Write After Read

- ❖ Instruction J should write its result after it is read by I
- ❖ Called **anti-dependence** by compiler writers

I: sub \$t4, **\$t1**, \$t3 # **\$t1 is read**

J: add **\$t1**, \$t2, \$t3 # **\$t1 is written**

- ❖ Results from reuse of the name **\$t1**
- ❖ NOT a data hazard in the 5-stage pipeline because:
 - ✧ Reads are always in stage 2
 - ✧ Writes are always in stage 5, and
 - ✧ Instructions are processed in order
- ❖ Anti-dependence can be eliminated by **renaming**
 - ✧ Use a different destination register for **add** (eg, **\$t5**)

Name Dependence: Write After Write

- ❖ Same destination register is written by two instructions

- ❖ Called **output-dependence** in compiler terminology

I: sub **\$t1**, \$t4, \$t3 # **\$t1 is written**

J: add **\$t1**, \$t2, \$t3 # **\$t1 is written again**

- ❖ Not a data hazard in the 5-stage pipeline because:

 - ✧ All writes are ordered and always take place in stage 5

- ❖ However, can be a hazard in more complex pipelines

 - ✧ If instructions are allowed to complete out of order, and

 - ✧ Instruction J completes and writes **\$t1** before instruction I

- ❖ Output dependence can be eliminated by **renaming \$t1**

- ❖ **Read After Read is NOT a name dependence**

Next . . .

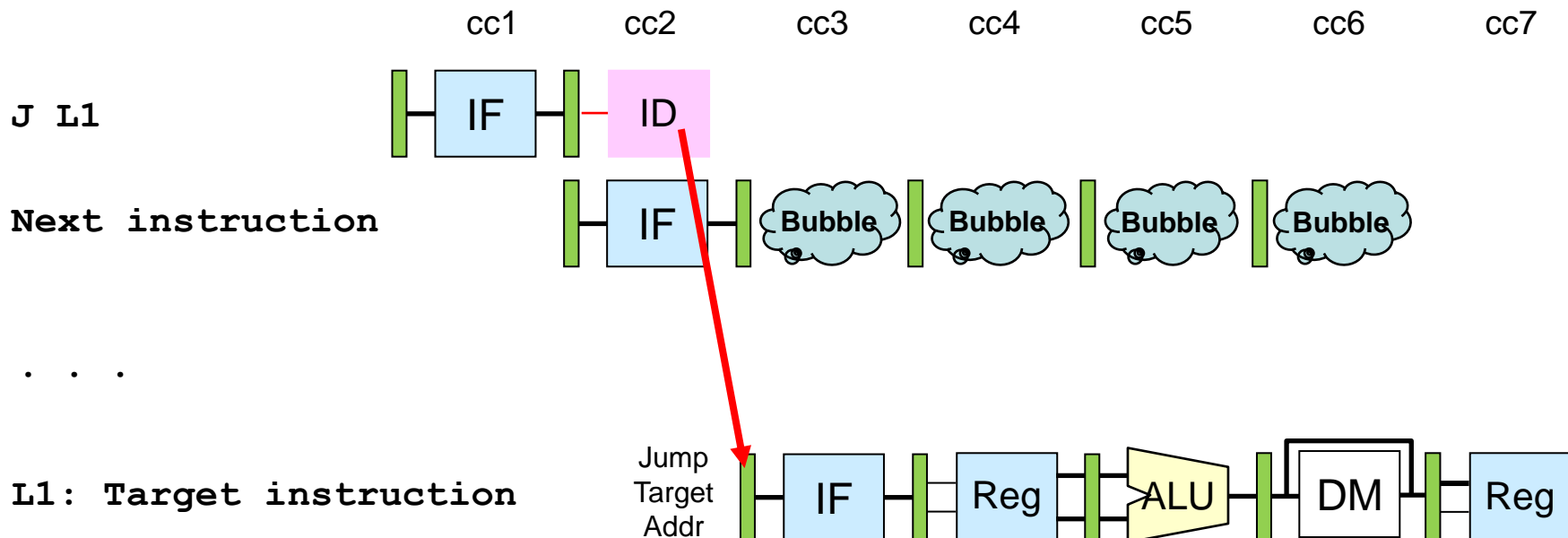
- ❖ Serial versus Pipelined Execution
- ❖ Pipelined Datapath and Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall
- ❖ **Control Hazards**

Control Hazards

- ❖ Jump and Branch can cause great performance loss
- ❖ Jump instruction needs only the **jump target address**
- ❖ Branch instruction needs two things:
 - ✧ **Branch Result** Taken or Not Taken
 - ✧ **Branch Target Address**
 - $PC + 4$ If Branch is NOT taken
 - $PC + 4 + 4 \times \text{immediate}$ If Branch is Taken
- ❖ Jump and Branch targets are computed in the ID stage
 - ✧ At which point a new instruction is already being fetched
 - ✧ Jump Instruction: 1-cycle delay
 - ✧ Branch: 2-cycle delay for branch result (taken or not taken)

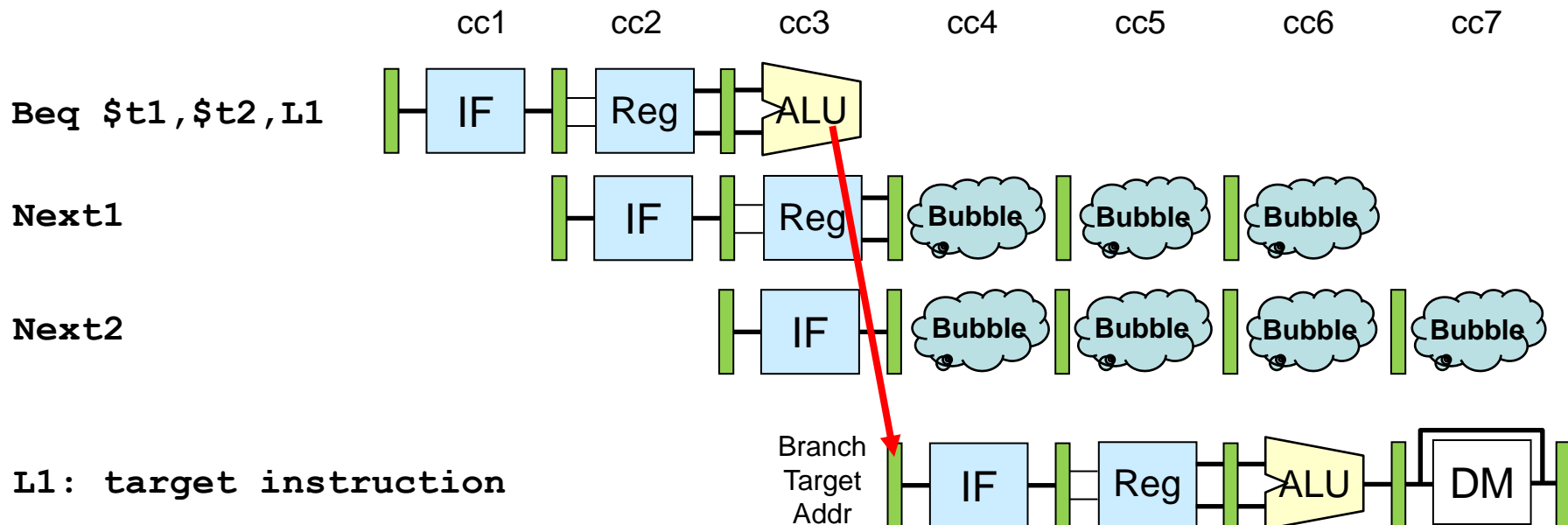
1-Cycle Jump Delay

- ❖ Control logic detects a **Jump** instruction in the 2nd Stage
- ❖ **Next** instruction is fetched anyway
- ❖ Convert **Next** instruction into **bubble** (Jump is always **taken**)



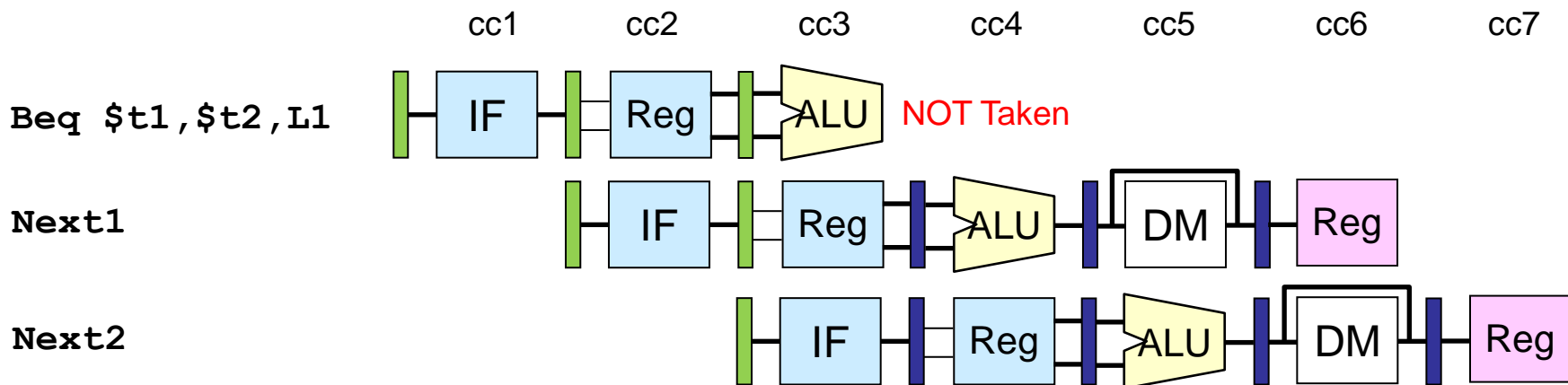
2-Cycle Branch Delay

- ❖ Control logic detects a **Branch** instruction in the 2nd Stage
- ❖ ALU computes the **Branch outcome** in the 3rd Stage
- ❖ **Next1** and **Next2** instructions will be fetched anyway
- ❖ Convert **Next1** and **Next2** into bubbles **if branch is taken**

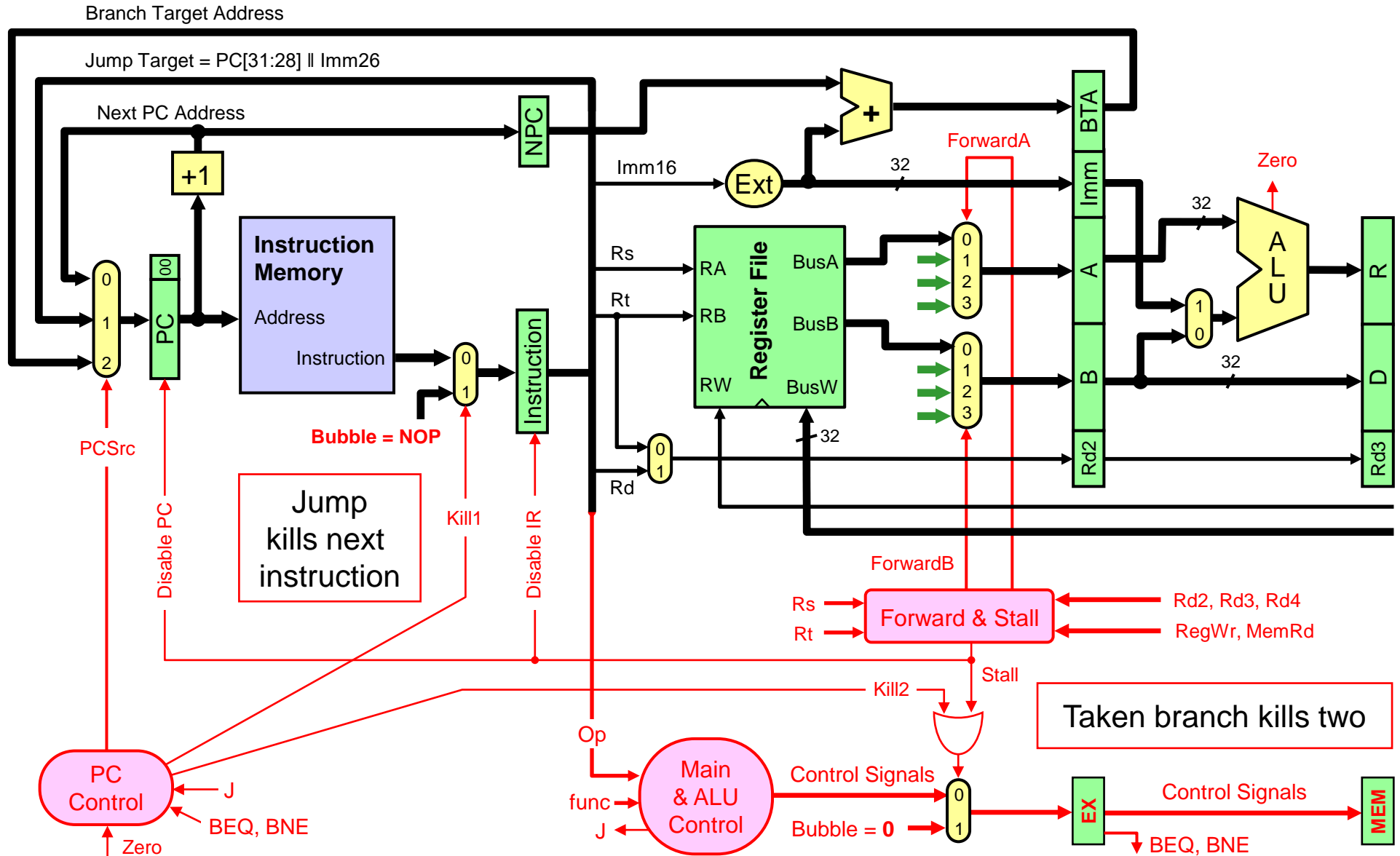


If Branch is NOT Taken ...

- ❖ Branches can be predicted to be NOT taken
- ❖ If **branch outcome** is **NOT taken** then
 - ✧ **Next1** and **Next2** instructions can be executed
 - ✧ Do not convert **Next1** & **Next2** into bubbles
 - ✧ **No wasted cycles**



Pipelined Jump and Branch



PC Control for Pipelined Jump and Branch

```
if ((BEQ && Zero) || (BNE && !Zero))  
    { Jmp=0; Br=1; Kill1=1; Kill2=1; }  
else if (J)  
    { Jmp=1; Br=0; Kill1=1; Kill2=0; }  
else  
    { Jmp=0; Br=0; Kill1=0; Kill2=0; }
```

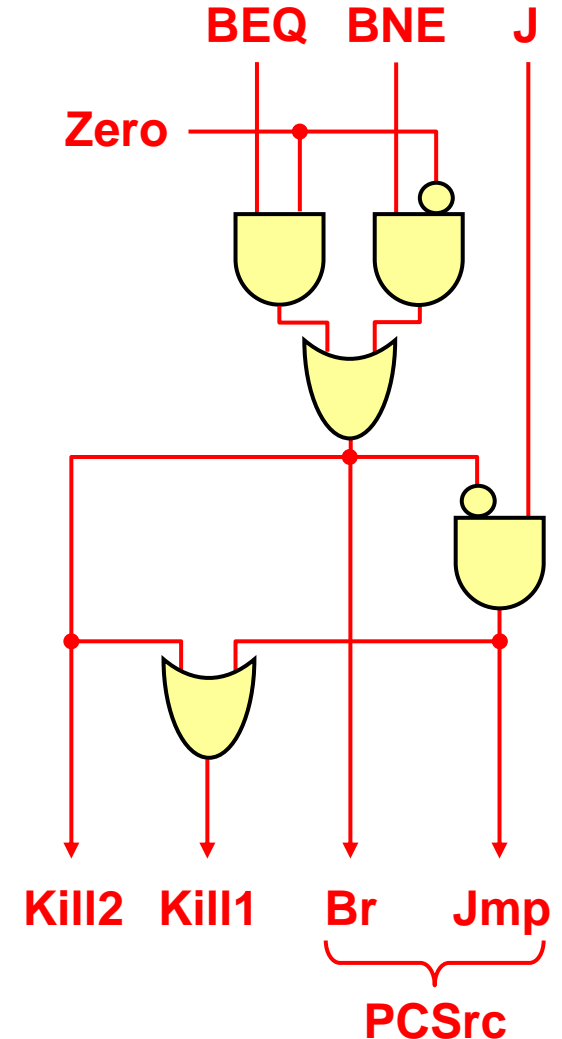
$$Br = ((BEQ \cdot Zero) + (BNE \cdot \overline{Zero}))$$

$$Jmp = J \cdot \overline{Br}$$

$$Kill1 = J + Br$$

$$Kill2 = Br$$

$$PCSrc = \{ Br, Jmp \} \quad // 0, 1, \text{ or } 2$$



Jump and Branch Impact on CPI

- ❖ Base CPI = 1 without counting jump and branch
- ❖ Unconditional Jump = 5%, Conditional branch = 20%
- ❖ 90% of conditional branches are taken
- ❖ Jump kills next instruction, Taken Branch kills next two
- ❖ What is the effect of jump and branch on the CPI?

Solution:

- ❖ Jump adds 1 wasted cycle for 5% of instructions = 1×0.05
- ❖ Branch adds 2 wasted cycles for $20\% \times 90\%$ of instructions
 $= 2 \times 0.2 \times 0.9 = 0.36$
- ❖ New CPI = $1 + 0.05 + 0.36 = 1.41$ (due to wasted cycles)