

Pipelined Processor Design

ICS 233

Computer Architecture and Assembly Language

Dr. Aiman El-Maleh

College of Computer Sciences and Engineering

King Fahd University of Petroleum and Minerals

[Adapted from slides of Dr. M. Mudawar, ICS 233, KFUPM]

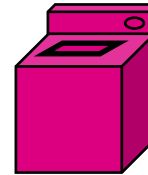
Outline

- ❖ **Pipelining versus Serial Execution**
- ❖ Pipelined Datapath
- ❖ Pipelined Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall Unit
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

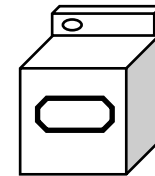
Pipelining Example

❖ Laundry Example: Three Stages

1. Wash dirty load of clothes



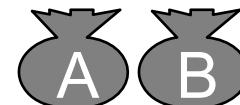
2. Dry wet clothes



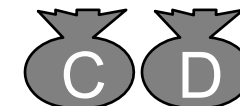
3. Fold and put clothes into drawers



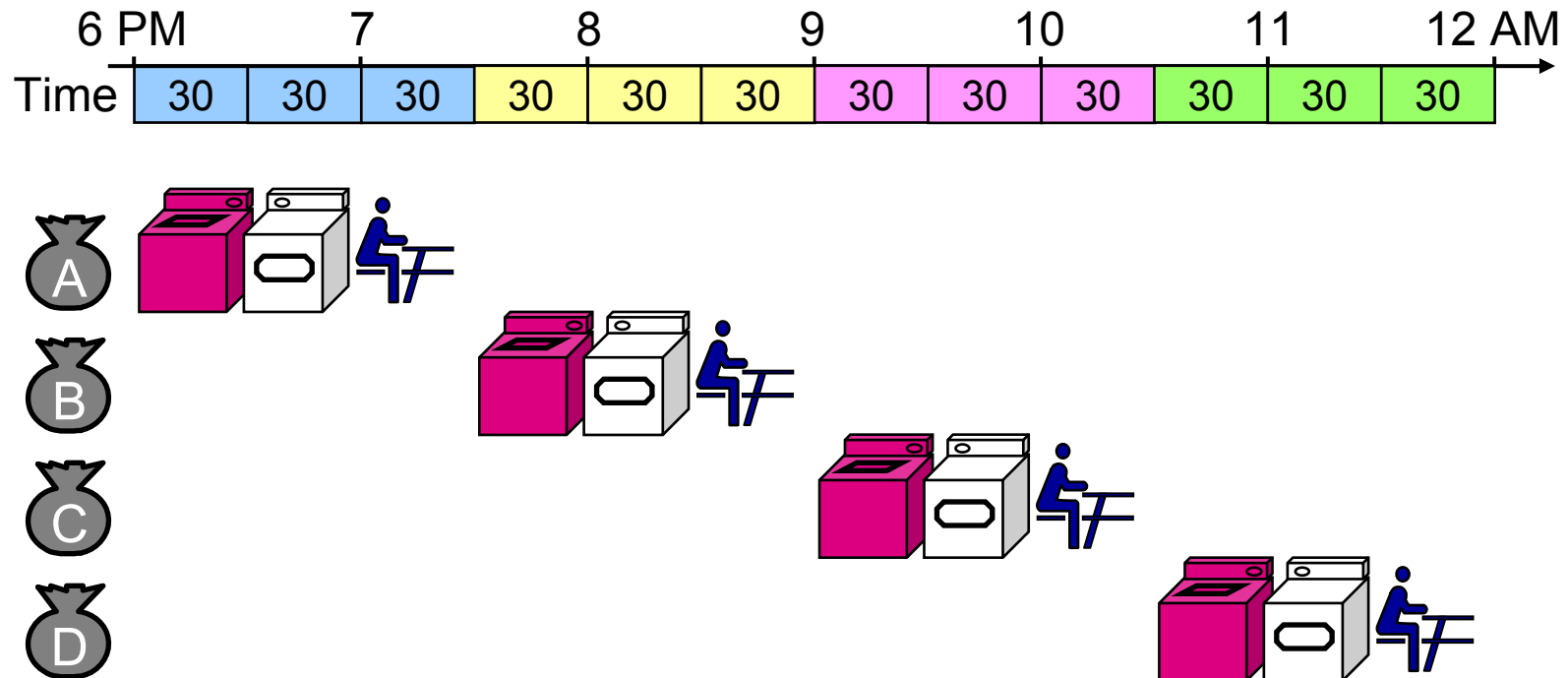
❖ Each stage takes 30 minutes to complete



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

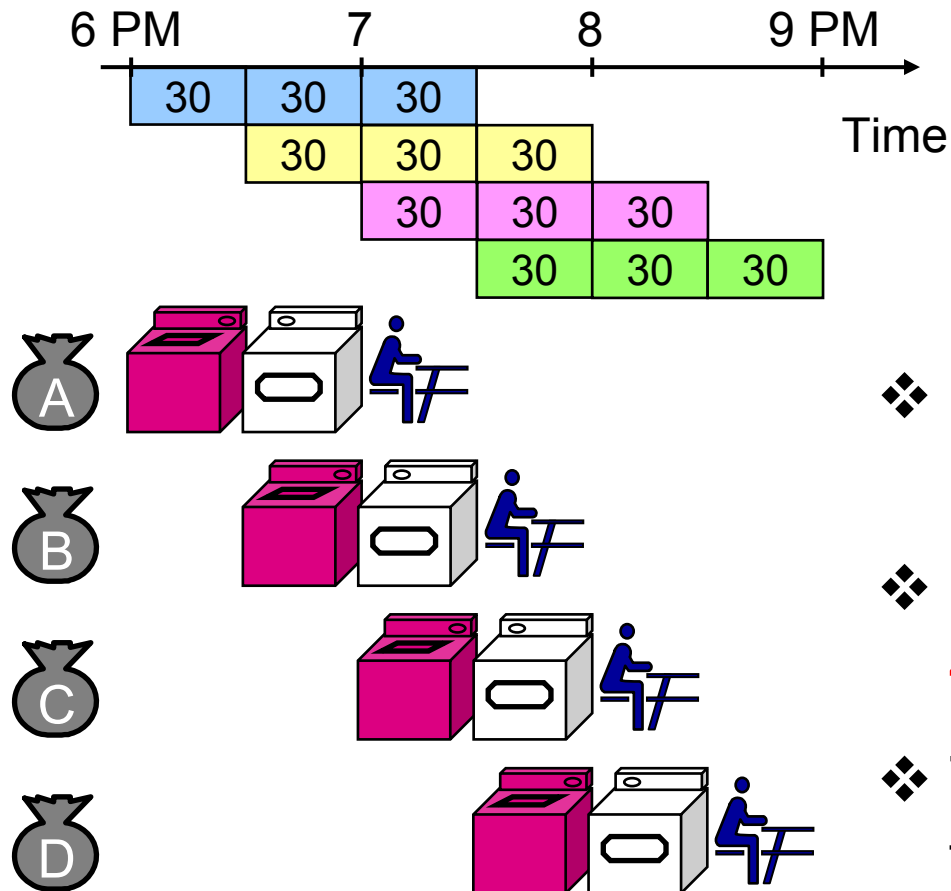


Sequential Laundry



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

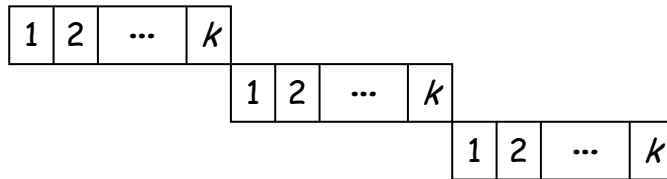
Pipelined Laundry: Start Load ASAP



- ❖ 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)

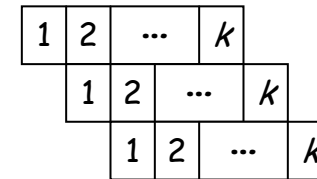
Serial Execution versus Pipelining

- ❖ 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 start a new task before finishing previous
 - ✧ The k stages work in parallel on k different tasks
 - ✧ Tasks enter/leave pipeline at the rate of one task per time unit



Without Pipelining

One completion every k time units

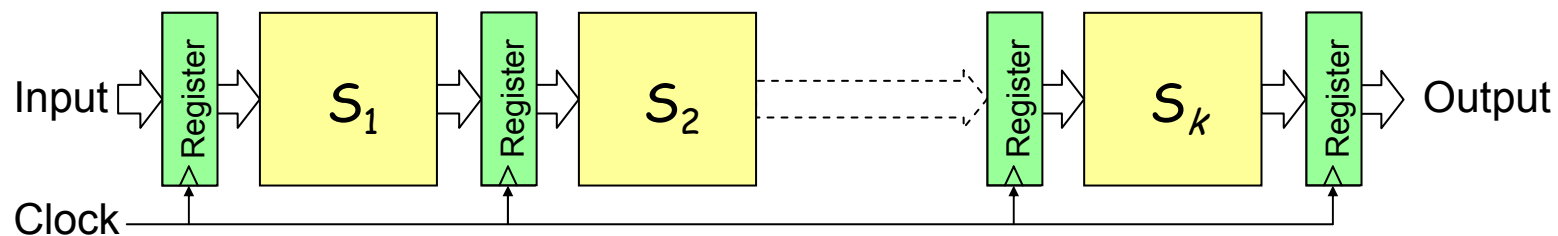


With Pipelining

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

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

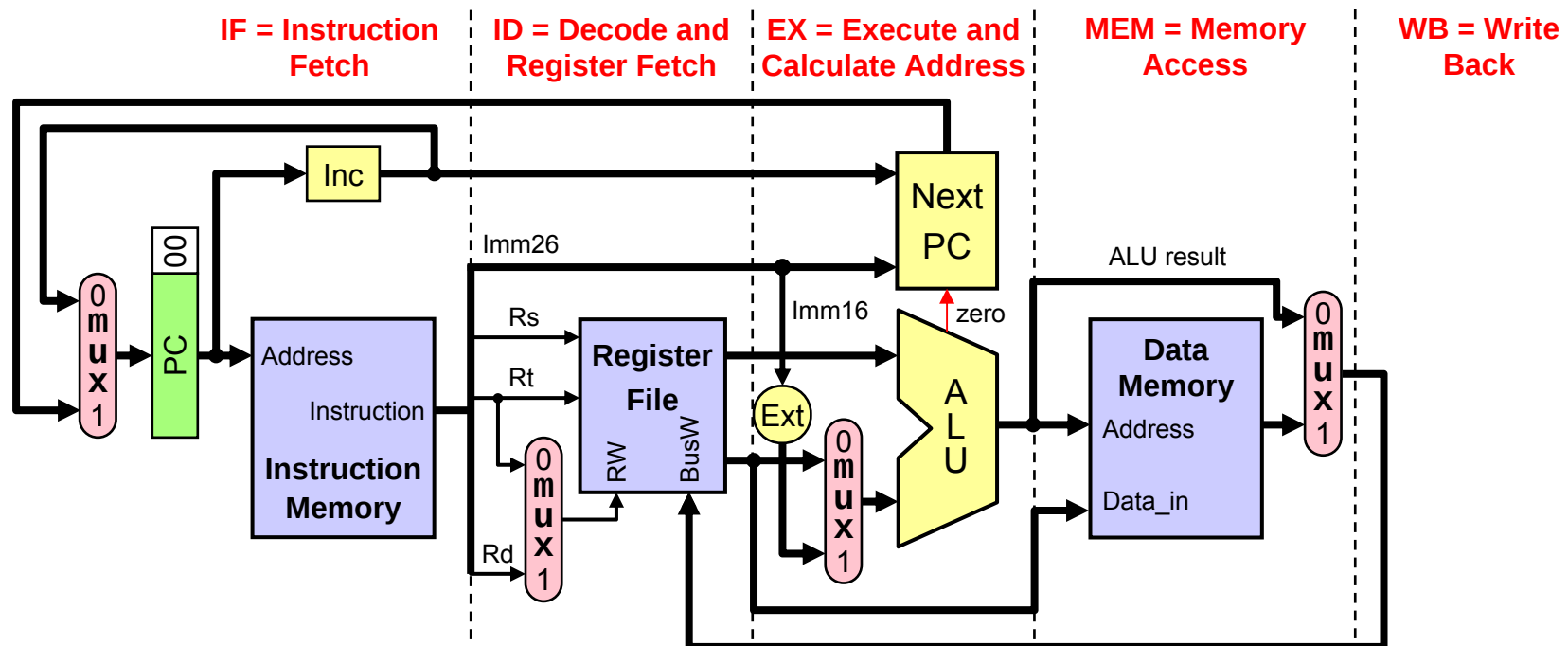
Next ...

- ❖ Pipelining versus Serial Execution
- ❖ **Pipelined Datapath**
- ❖ Pipelined Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall Unit
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

Single-Cycle Datapath

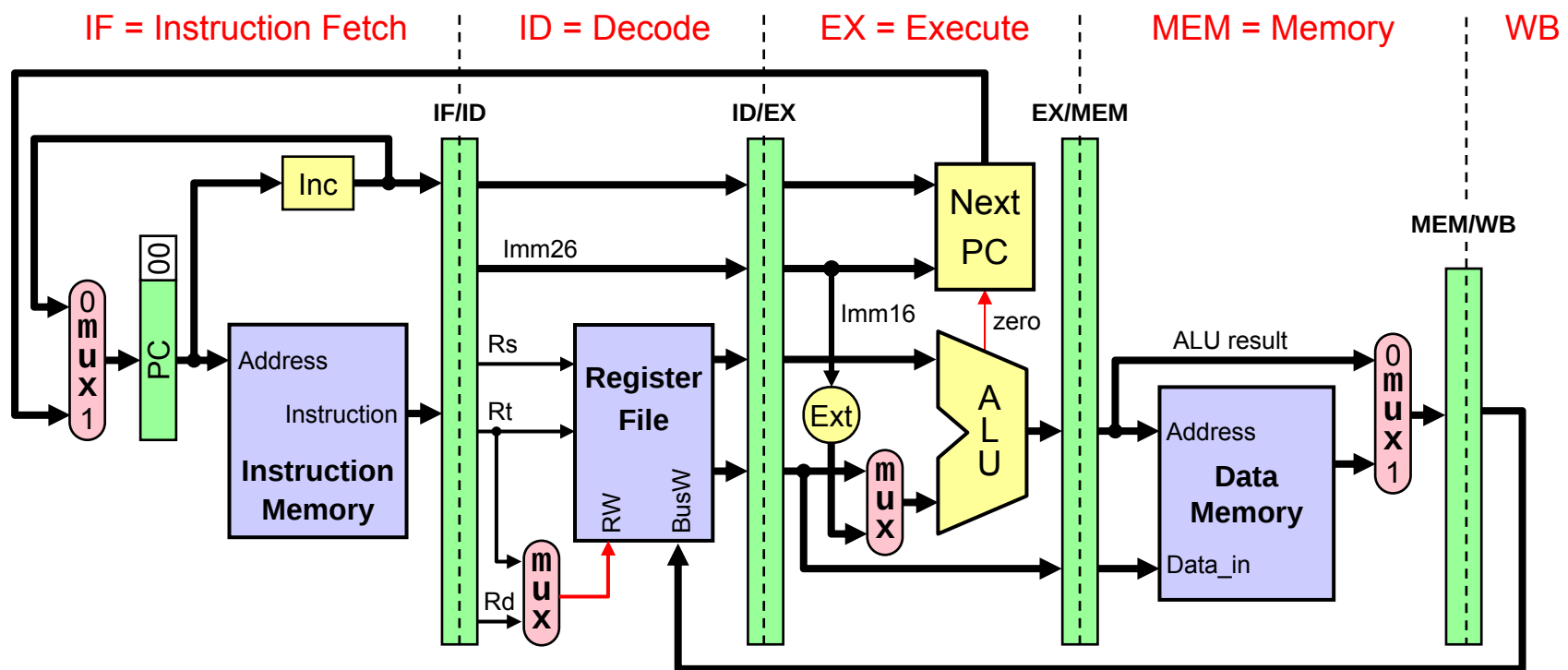
- ❖ Shown below is the single-cycle datapath
- ❖ How to pipeline this single-cycle datapath?

Answer: Introduce registers at the end of each stage



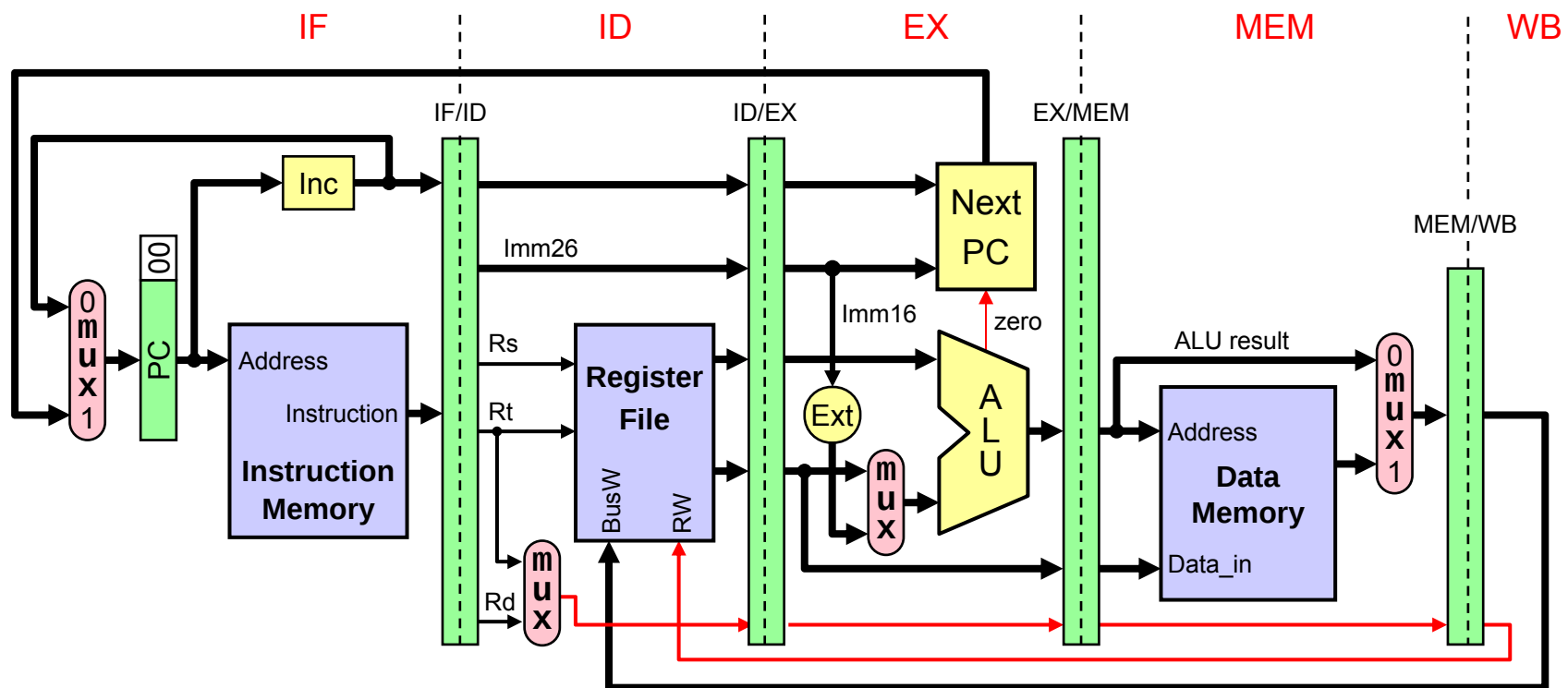
Pipelined Datapath

- ❖ Pipeline registers, in green, separate each pipeline stage
- ❖ Pipeline registers are labeled by the stages they separate
- ❖ Is there a problem with the register destination address?



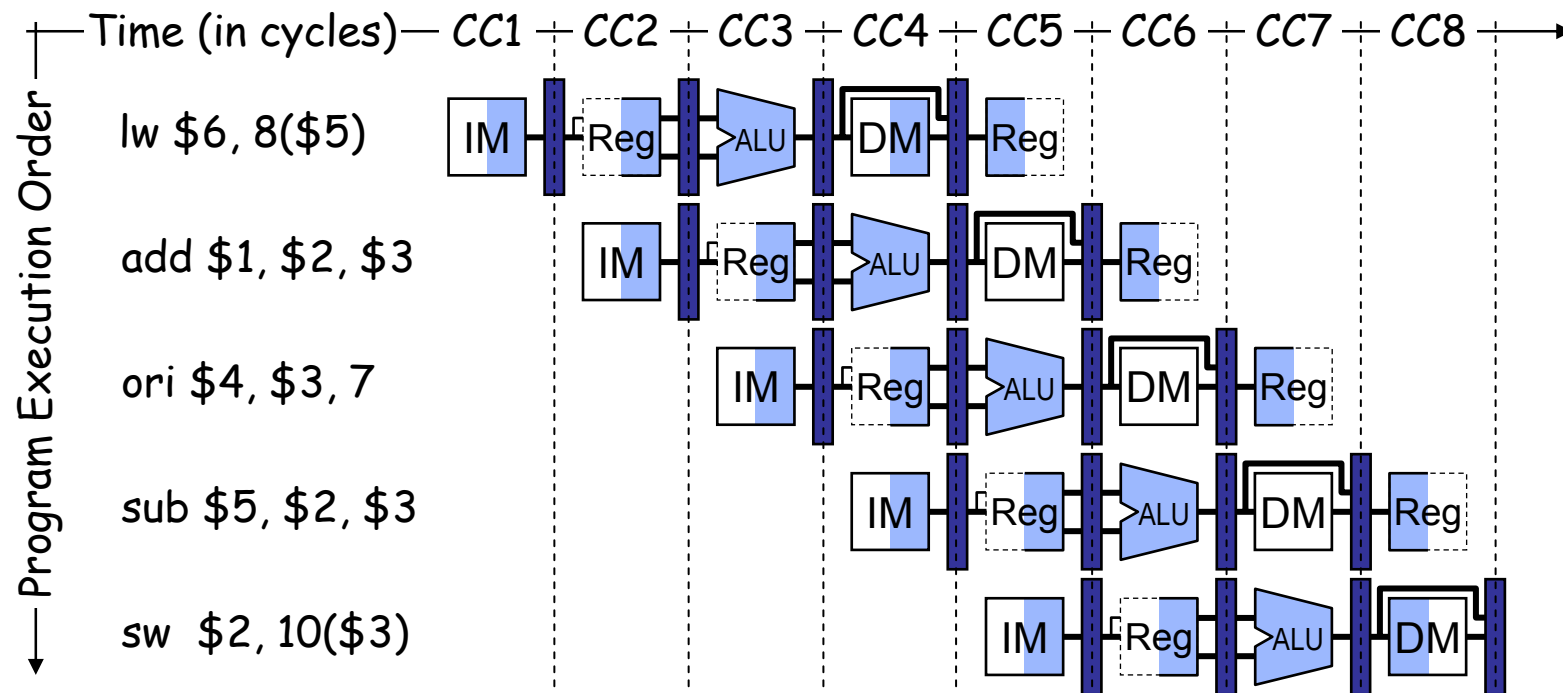
Corrected Pipelined Datapath

- ❖ Destination register number should come from MEM/WB
 - ✧ Along with the data during the written back stage
- ❖ Destination register number is passed from ID to WB stage



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

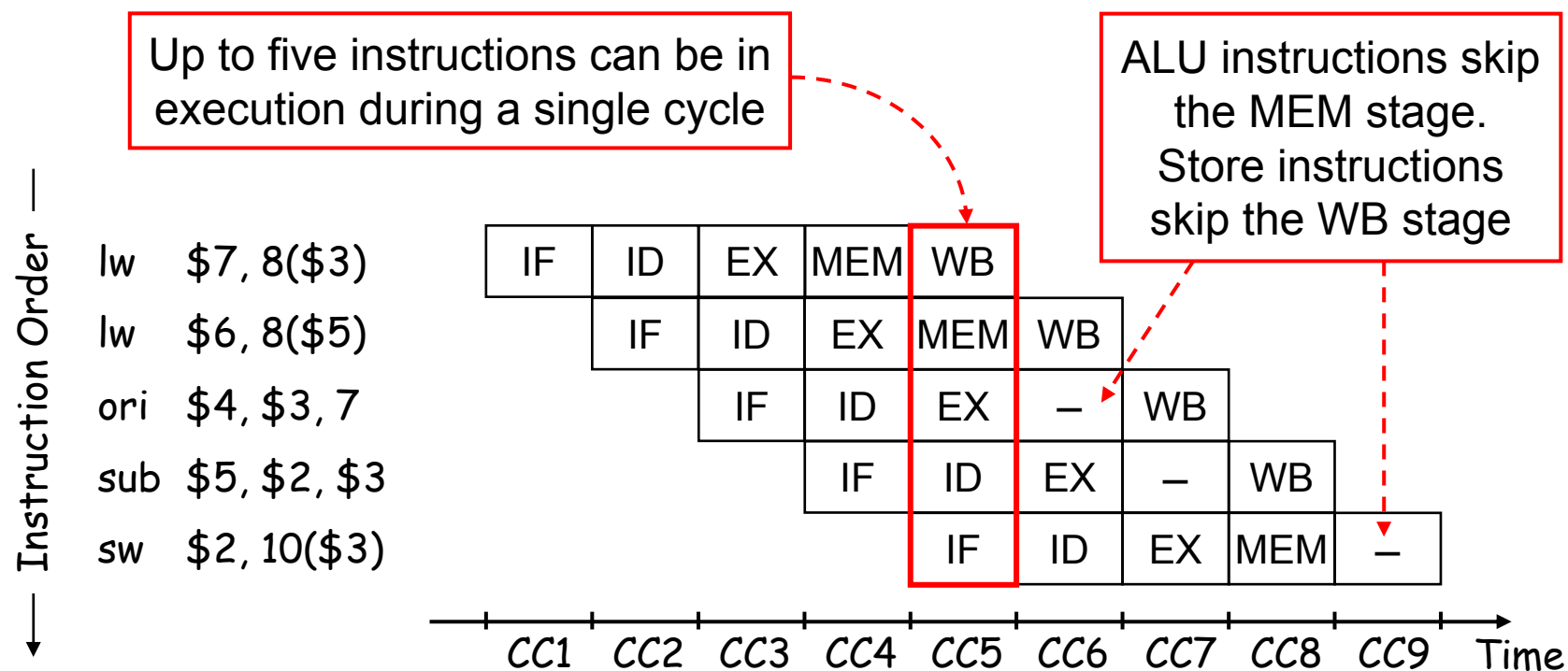


Instruction-Time Diagram

❖ Diagram shows:

✧ Which instruction occupies what stage at each clock cycle

❖ Instruction execution is pipelined over the 5 stages



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

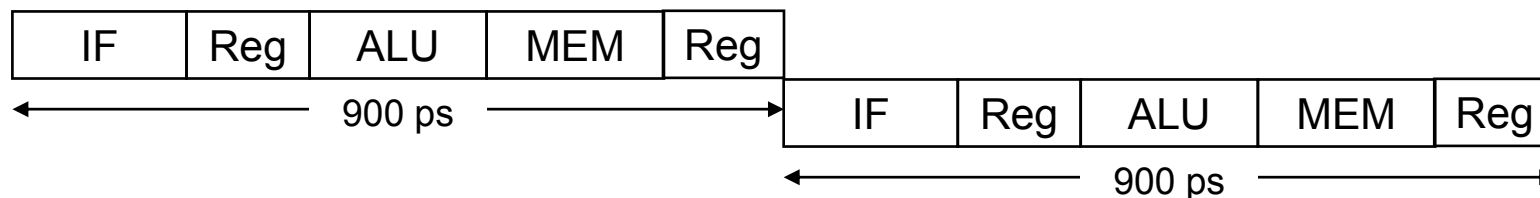
❖ What is the single-cycle non-pipelined time?

❖ What is the pipelined cycle time?

❖ What is the speedup factor for pipelined execution?

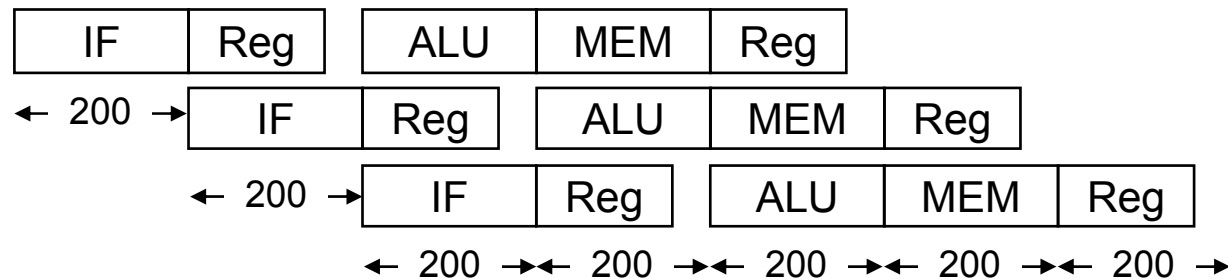
❖ **Solution**

Non-pipelined cycle = $200 + 150 + 200 + 200 + 150 = 900 \text{ ps}$



Single-Cycle versus Pipelined - cont'd

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



❖ 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

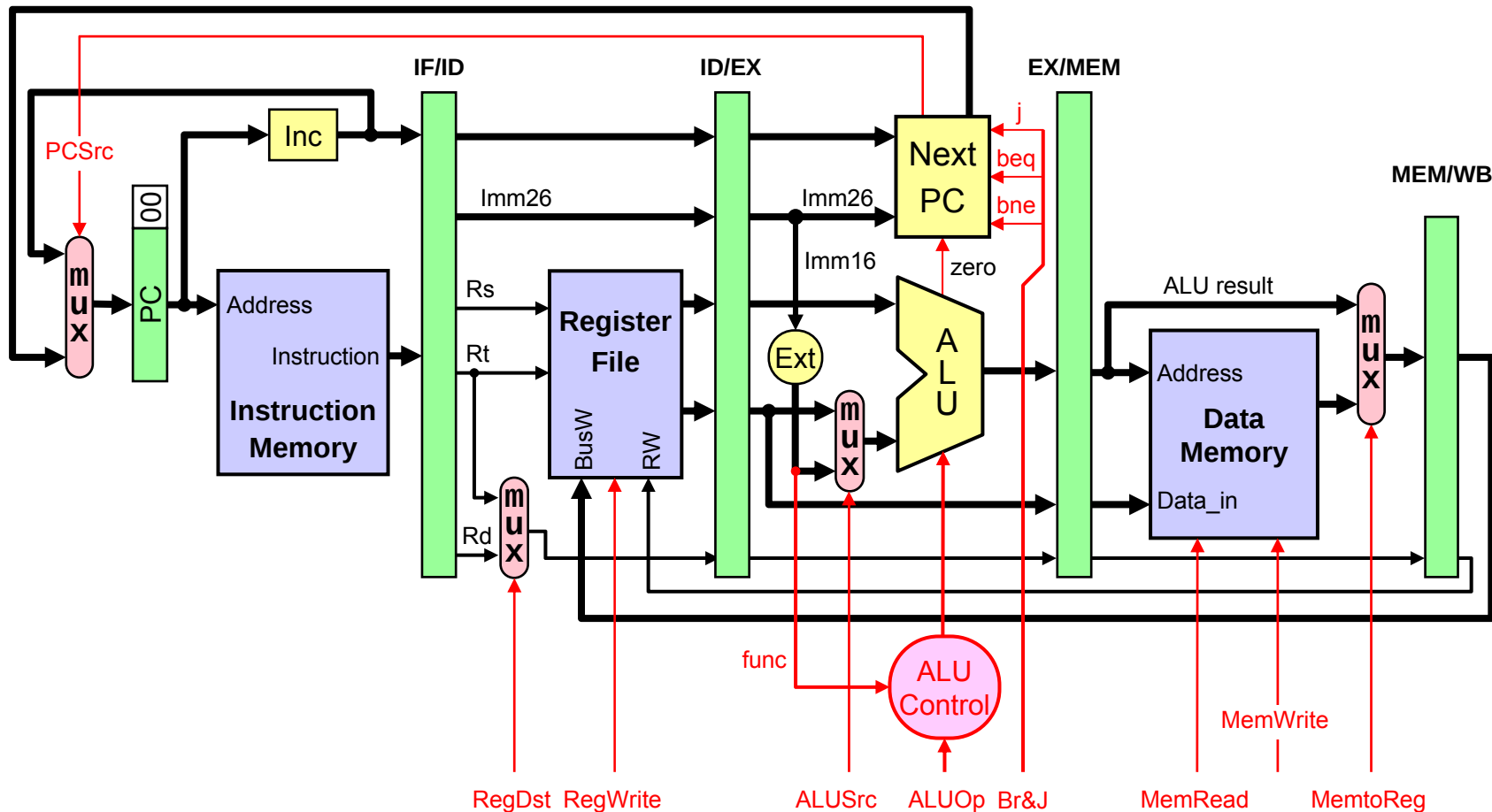
❖ Speedup factor is less than 5 (number of pipeline stage)

✧ Because the pipeline stages are not balanced

Next ...

- ❖ Pipelining versus Serial Execution
- ❖ Pipelined Datapath
- ❖ **Pipelined Control**
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall Unit
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

Control Signals

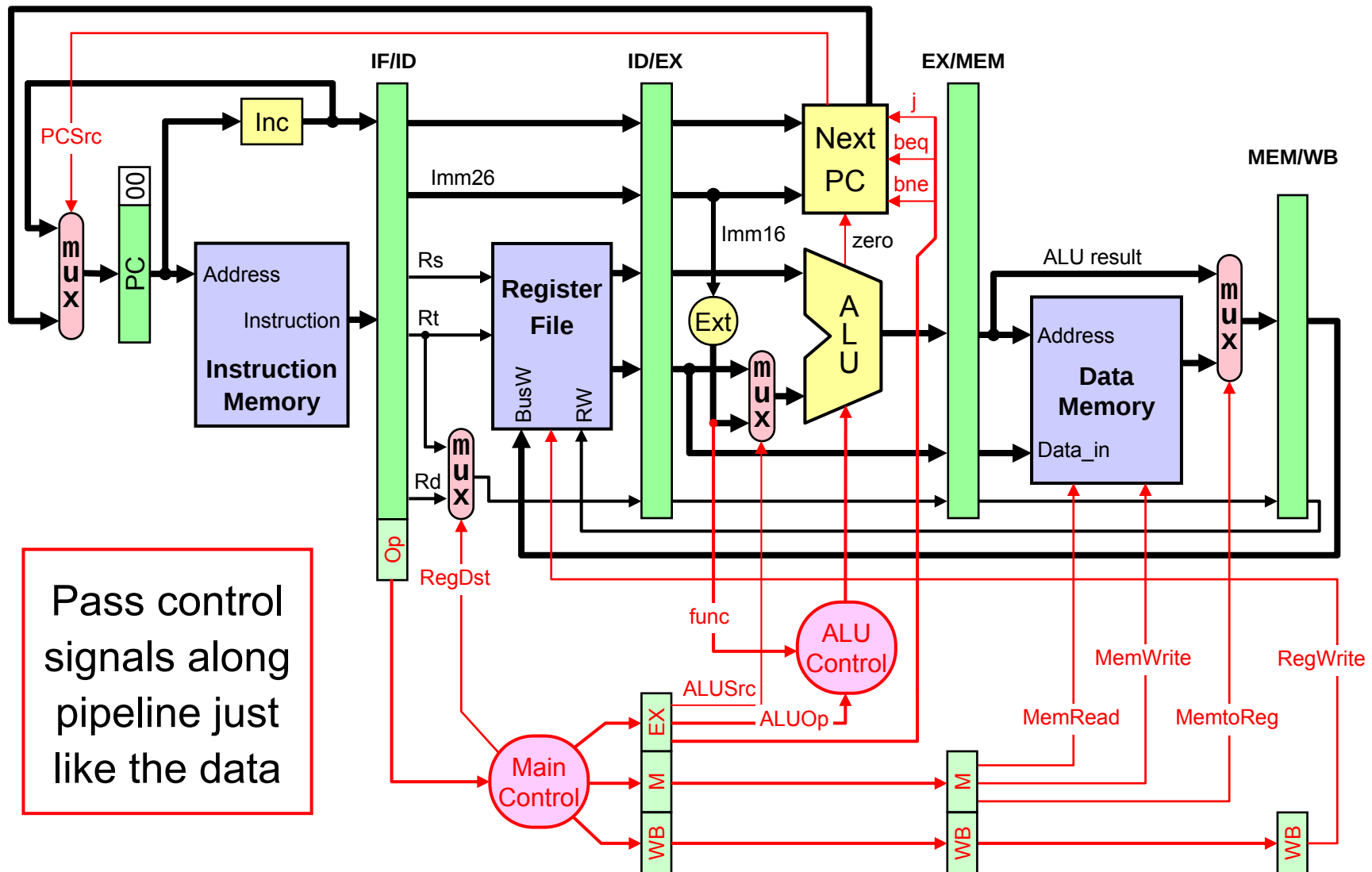


Similar to control signals used in the single-cycle datapath

Control Signals - cont'd

Op	Decode Signal	Execute Stage Control Signals					Memory Stage Control Signals			Writeback Signal
	RegDst	ALUSrc	ALUOp	Beq	Bne	j	MemRead	MemWrite	MemtoReg	RegWrite
R-Type	1=Rd	0=Reg	R-Type	0	0	0	0	0	0	1
addi	0=Rt	1=Imm	ADD	0	0	0	0	0	0	1
slti	0=Rt	1=Imm	SLT	0	0	0	0	0	0	1
andi	0=Rt	1=Imm	AND	0	0	0	0	0	0	1
ori	0=Rt	1=Imm	OR	0	0	0	0	0	0	1
lw	0=Rt	1=Imm	ADD	0	0	0	1	0	1	1
sw	x	1=Imm	ADD	0	0	0	0	1	x	0
beq	x	0=Reg	SUB	1	0	0	0	0	x	0
bne	x	0=Reg	SUB	0	1	0	0	0	x	0
j	x	x	x	0	0	1	0	0	x	0

Pipelined Control



Pass control signals along pipeline just like the data

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 Fetch
 - Control signals are generated
 - **RegDst** is used in this stage
 - ✧ Execution Stage => **ALUSrc** and **ALUOp**
 - Next PC uses **Beq**, **Bne**, **J** and **zero** signals for branch control
 - ✧ Memory Stage => **MemRead**, **MemWrite**, and **MemtoReg**
 - ✧ Write Back Stage => **RegWrite** is used in this stage

Pipelining 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**
 - ✧ Unbalanced lengths of pipeline stages reduces speedup
- ❖ Pipeline rate is limited by **slowest** pipeline stage
- ❖ Also, time to **fill** and **drain** pipeline reduces speedup

Next ...

- ❖ Pipelining versus Serial Execution
- ❖ Pipelined Datapath
- ❖ Pipelined Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall Unit
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

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

✧ Hardware can detect dependencies between instructions

3. Control hazards

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

✧ Due to delays in changing the flow of control

❖ Hazards complicate pipeline control and limit performance

Structural Hazards

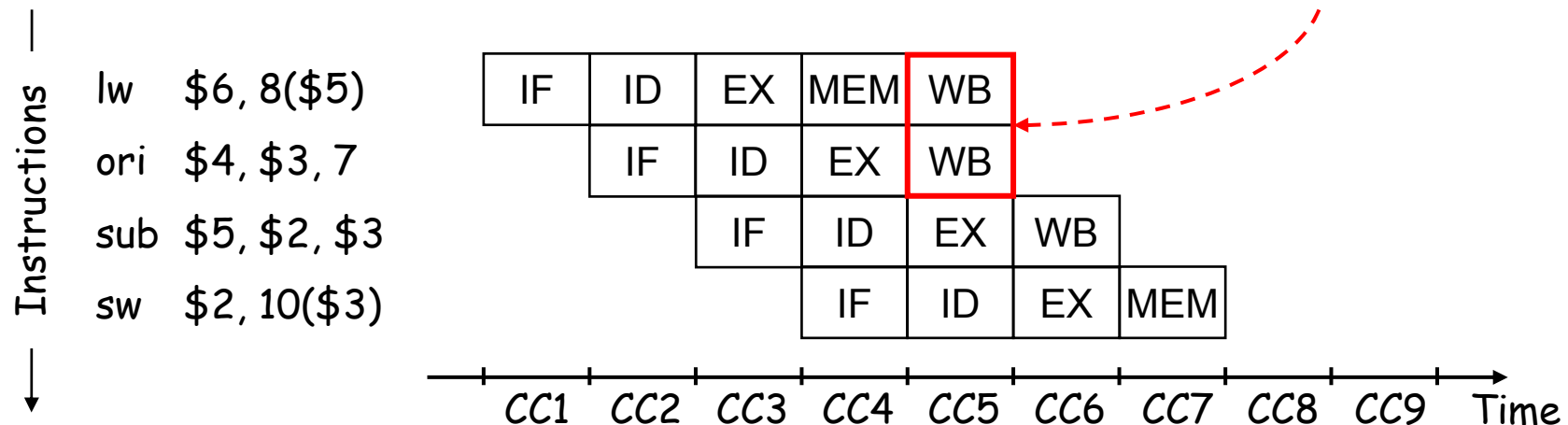
❖ Problem

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

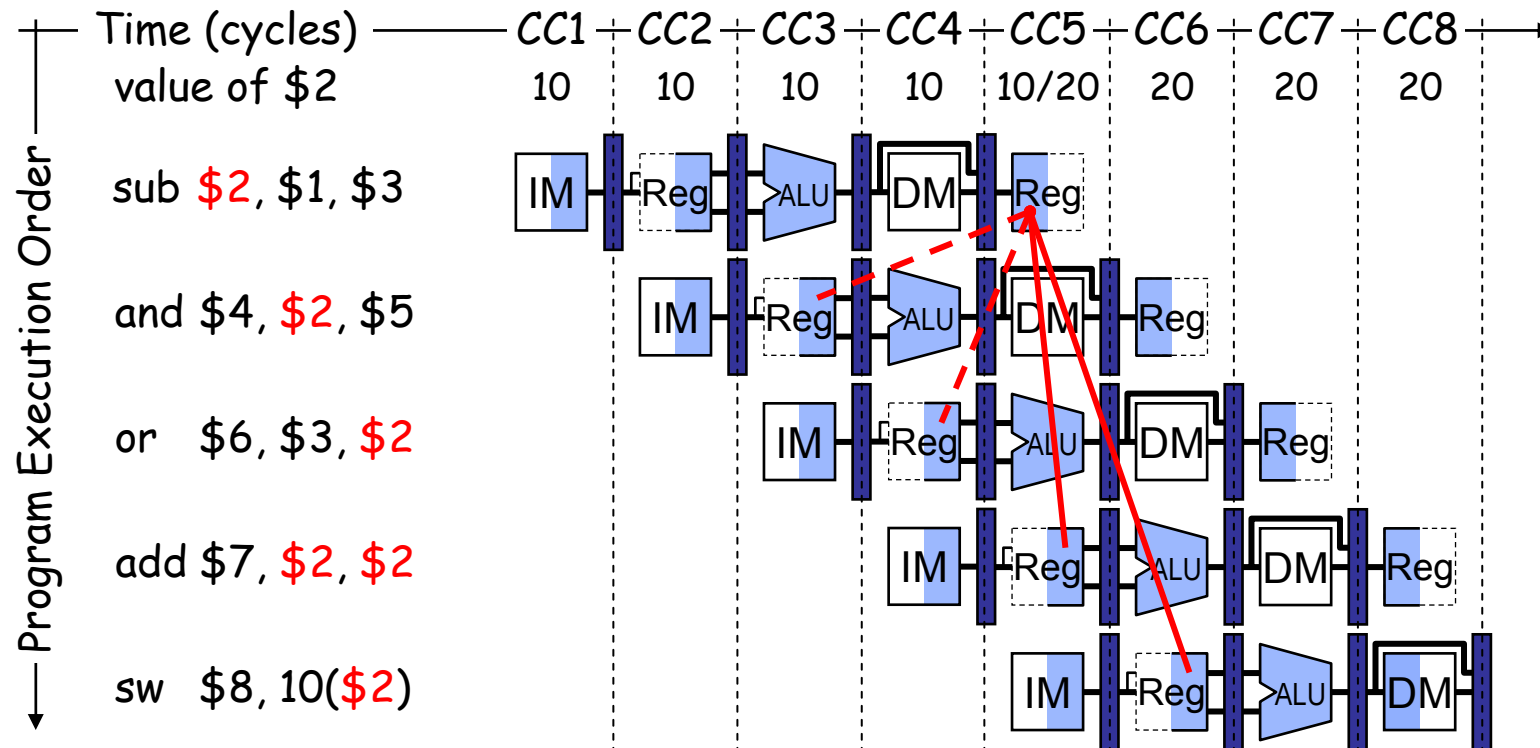
Next ...

- ❖ Pipelining versus Serial Execution
- ❖ Pipelined Datapath
- ❖ Pipelined Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall Unit
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

Data Hazards

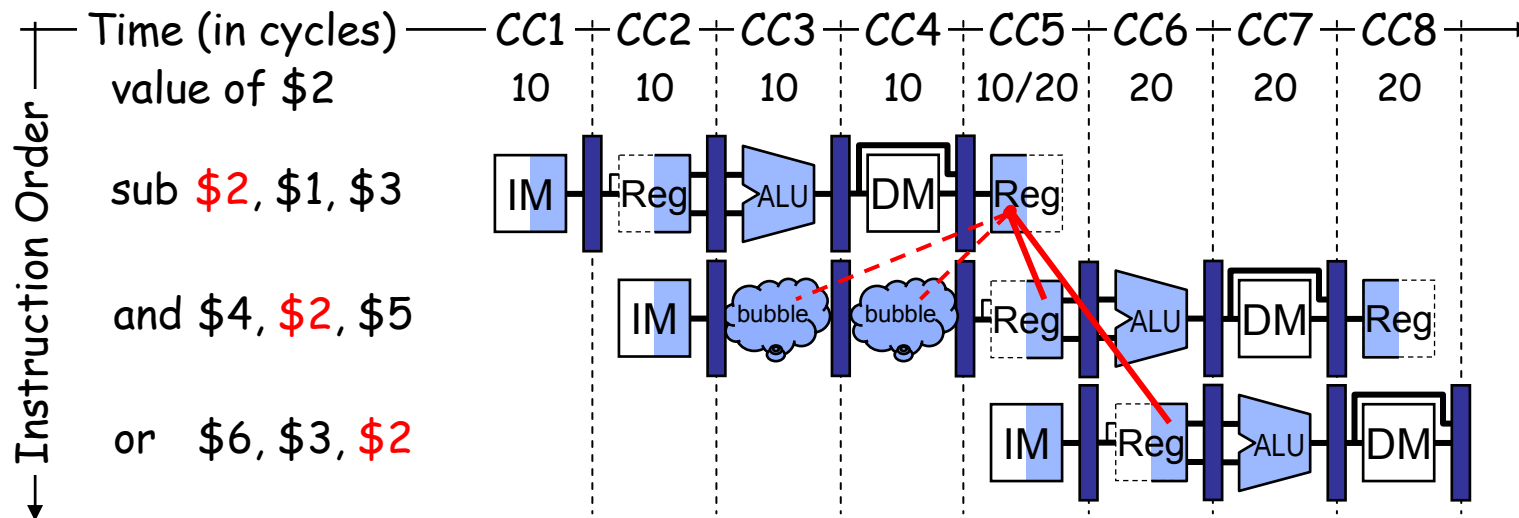
- ❖ Dependency between instructions causes a data hazard
- ❖ The dependent instructions are close to each other
 - ✧ Pipelined execution might change the order of operand access
- ❖ Read After Write – RAW Hazard
 - ✧ Given two instructions *I* and *J*, where *I* comes before *J* ...
 - ✧ Instruction *J* should read an operand after it is written by *I*
 - ✧ Called a **data dependence** in compiler terminology
 - I: add \$1, \$2, \$3 # r1 is written**
 - J: sub \$4, \$1, \$3 # r1 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 **and**, **or**, **add**, & **sw** instructions
- ❖ Instructions **and** & **or** will read **old value** of **\$2** from reg file
- ❖ During CC5, **\$2** is written and read – **new value** is read

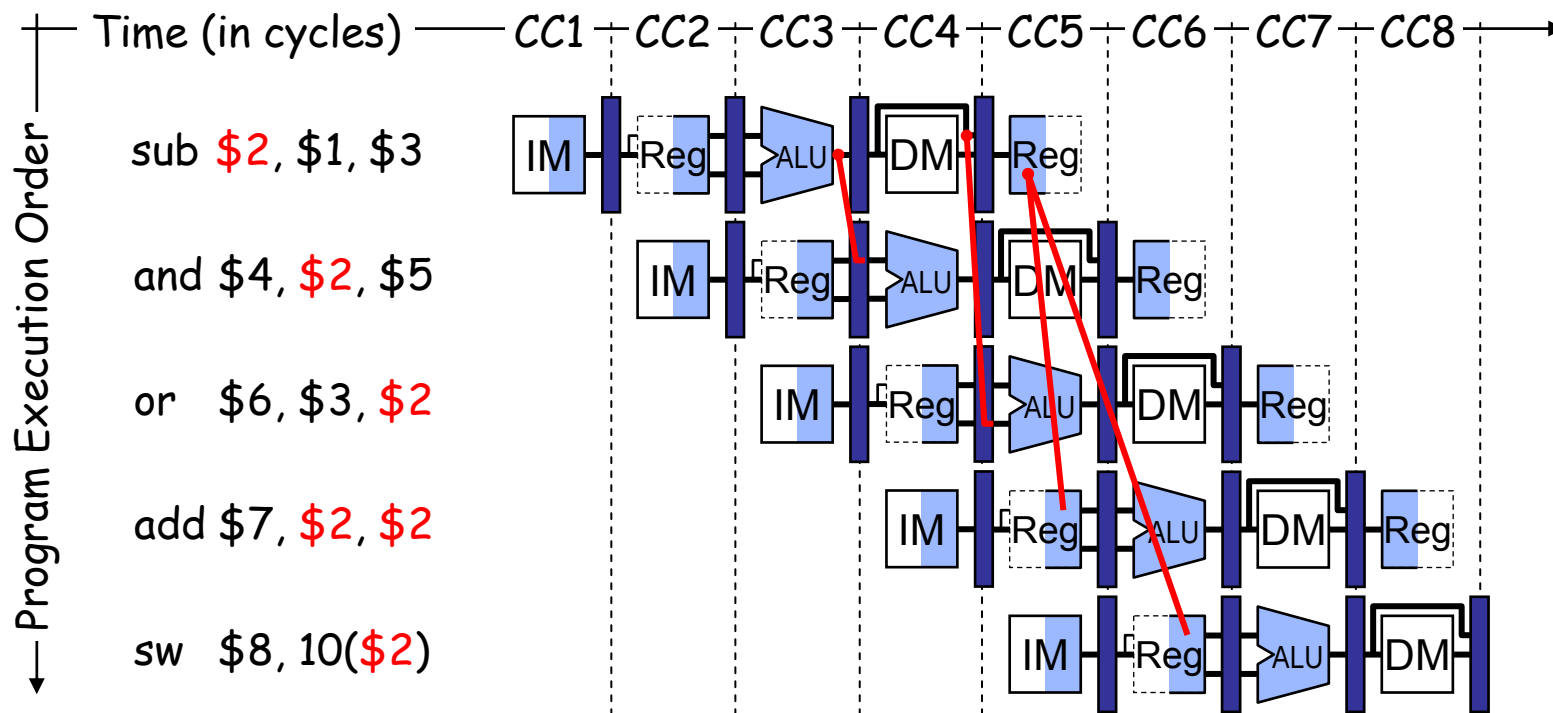
Solution 1: Stalling the Pipeline



- ❖ The **and** instruction cannot fetch **\$2** until **CC5**
 - ✧ The **and** instruction remains in the **IF/ID** register until **CC5**
- ❖ Two **bubbles** are inserted into **ID/EX** at end of **CC3** & **CC4**
 - ✧ Bubbles are **NOP** instructions: do not modify registers or memory
 - ✧ Bubbles delay instruction execution and waste clock cycles

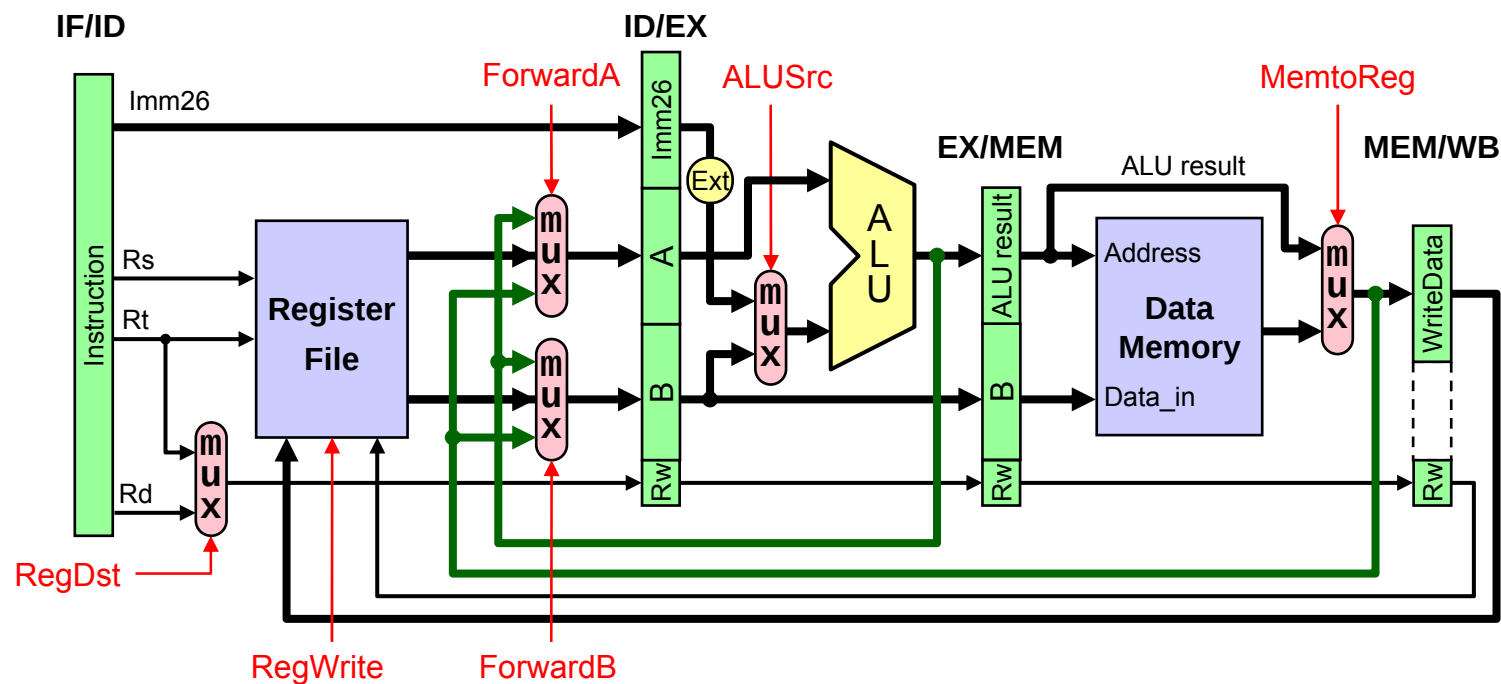
Solution 2: Forwarding ALU Result

- ❖ 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 exists in either **EX/MEM** or **MEM/WB** register



Implementing Forwarding

- ❖ Two multiplexers added at the inputs of A & B registers
 - ✧ ALU output in the EX stage is forwarded (fed back)
 - ✧ ALU result or Load data in the MEM stage is also forwarded
- ❖ Two signals: ForwardA and ForwardB control forwarding



RAW Hazard Detection

- ❖ RAW hazards can be detected by the pipeline
- ❖ **Current** instruction being decoded is in **IF/ID** register
- ❖ **Previous** instruction is in the **ID/EX** register
- ❖ **Second previous** instruction is in the **EX/MEM** register
- ❖ RAW Hazard Conditions:

$$\text{IF/ID.Rs} = \text{ID/EX.Rw}$$

$$\text{IF/ID.Rt} = \text{ID/EX.Rw}$$

} Raw Hazard detected with
Previous Instruction

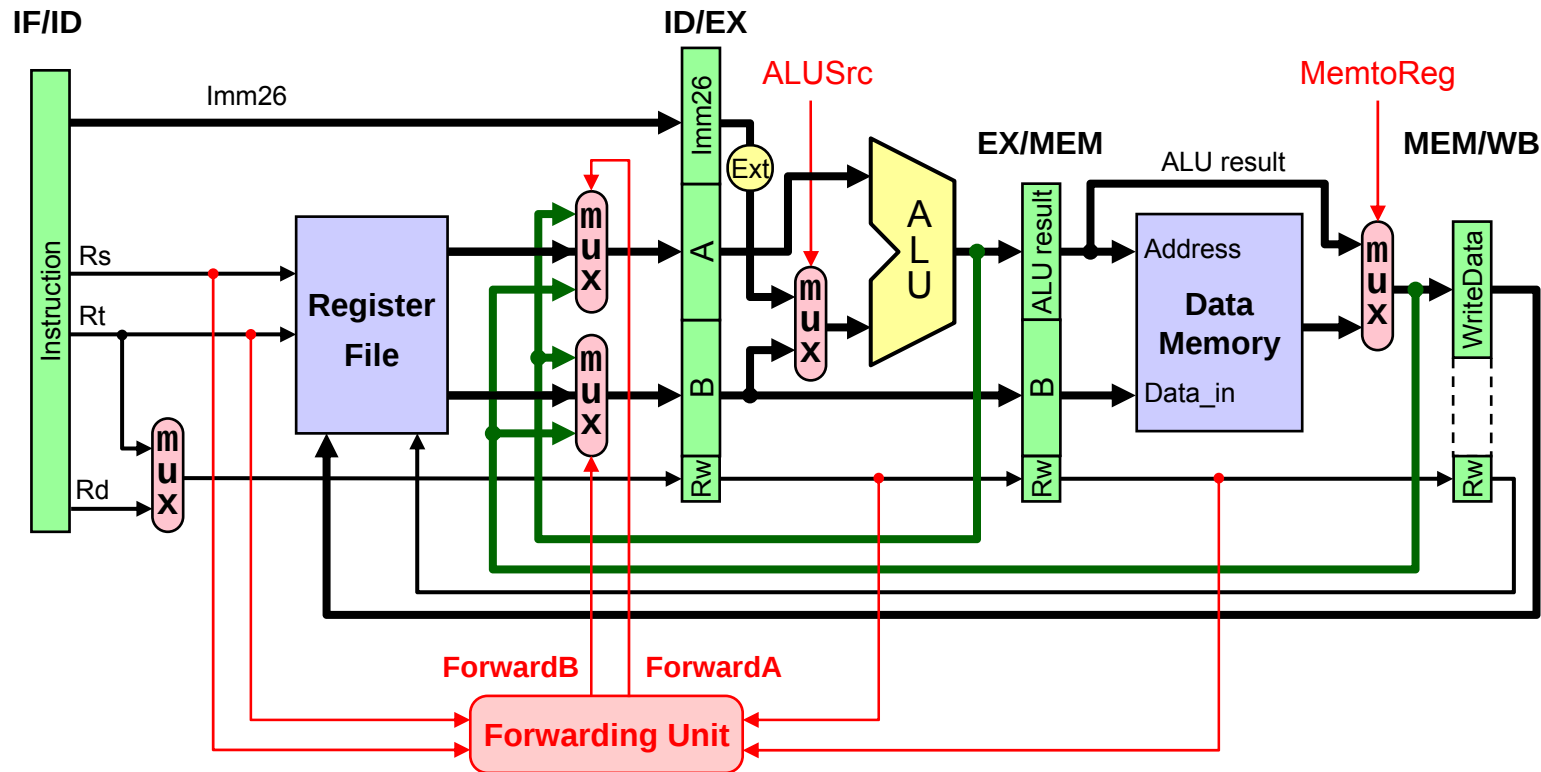
$$\text{IF/ID.Rs} = \text{EX/MEM.Rw}$$

$$\text{IF/ID.Rt} = \text{EX/MEM.Rw}$$

} Raw Hazard detected with
Second Previous Instruction

Forwarding Unit

- ❖ Forwarding unit generates **ForwardA** and **ForwardB**
 - ✧ That are used to control the two forwarding multiplexers
- ❖ Uses **Rs** and **Rt** in **IF/ID** and **Rw** in **ID/EX** & **EX/MEM**



Forwarding Control Signals

Control Signal	Explanation
ForwardA = 00	First ALU operand comes from the register file
ForwardA = 01	Forwarded from the previous ALU result
ForwardA = 10	Forwarded from data memory or 2 nd previous ALU result
ForwardB = 00	Second ALU operand comes from the register file
ForwardB = 01	Forwarded from the previous ALU result
ForwardB = 10	Forwarded from data memory or 2 nd previous ALU result

```
if (IF/ID.Rs == ID/EX.Rw and ID/EX.Rw ≠ 0 and ID/EX.RegWrite) ForwardA = 01
elseif (IF/ID.Rs == EX/MEM.Rw and EX/MEM.Rw ≠ 0 and EX/MEM.RegWrite)
    ForwardA = 10
```

```
else ForwardA = 00
```

```
if (IF/ID.Rt == ID/EX.Rw and ID/EX.Rw ≠ 0 and ID/EX.RegWrite) ForwardB = 01
elseif (IF/ID.Rt == EX/MEM.Rw and EX/MEM.Rw ≠ 0 and EX/MEM.RegWrite)
    ForwardB = 10
```

```
else ForwardB = 00
```

Forwarding Example

Instruction sequence:

lw \$4, 100(\$9)

add \$7, \$5, \$6

sub \$8, \$4, \$7

When **lw** reaches the MEM stage

add will be in the ALU stage

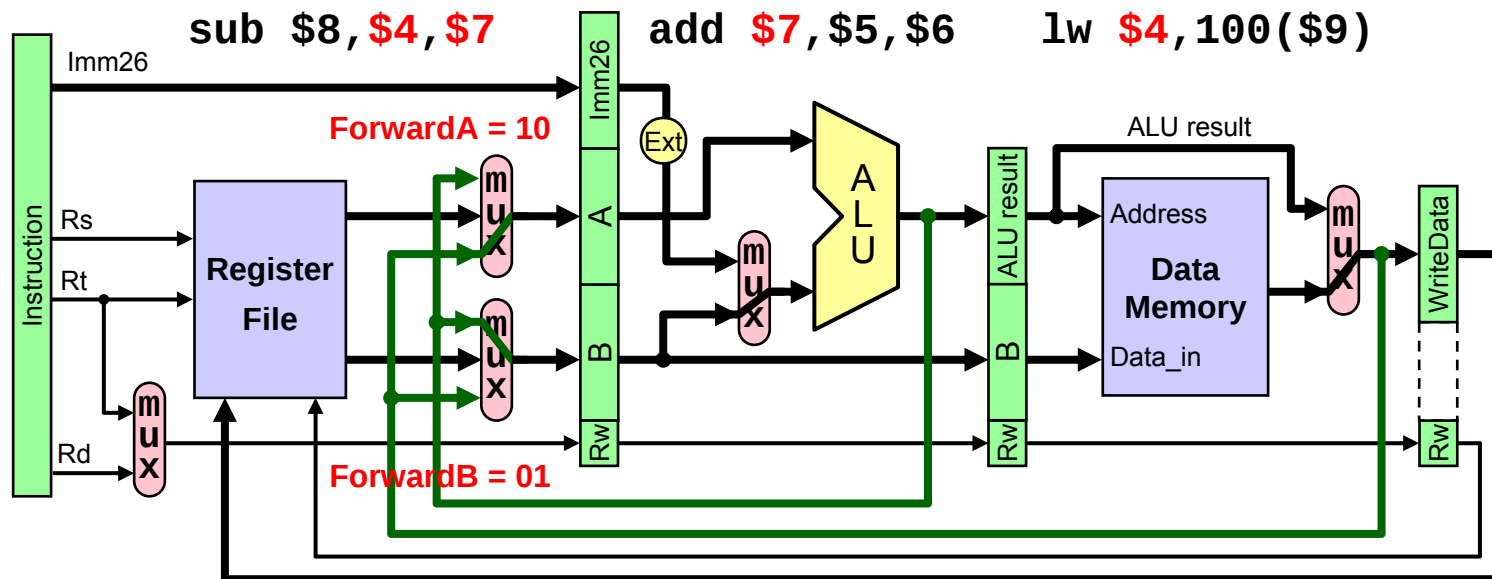
sub will be in the Decode stage

ForwardA = 10

ForwardB = 01

Forward data from MEM stage

Forward ALU result from ALU stage

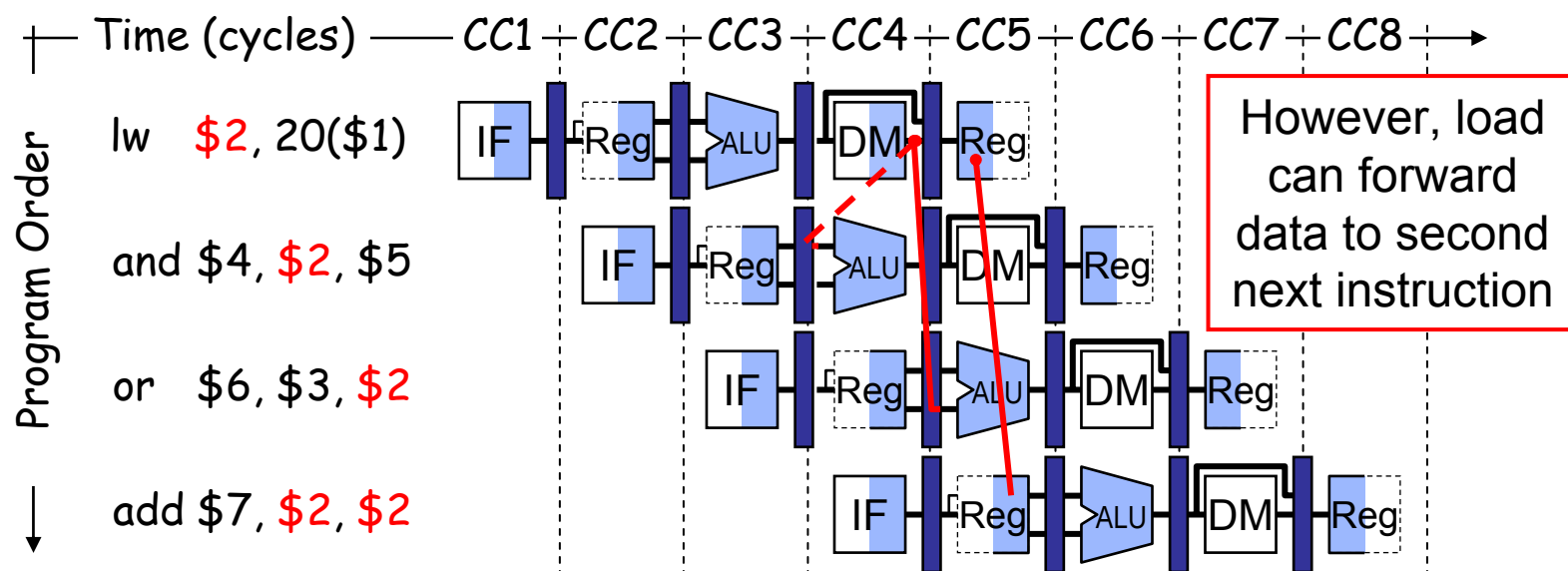


Next ...

- ❖ Pipelining versus Serial Execution
- ❖ Pipelined Datapath
- ❖ Pipelined Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall Unit
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

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 have data until end of CC4
 - ✧ **AND** instruction wants data at beginning of CC4 - **NOT possible**

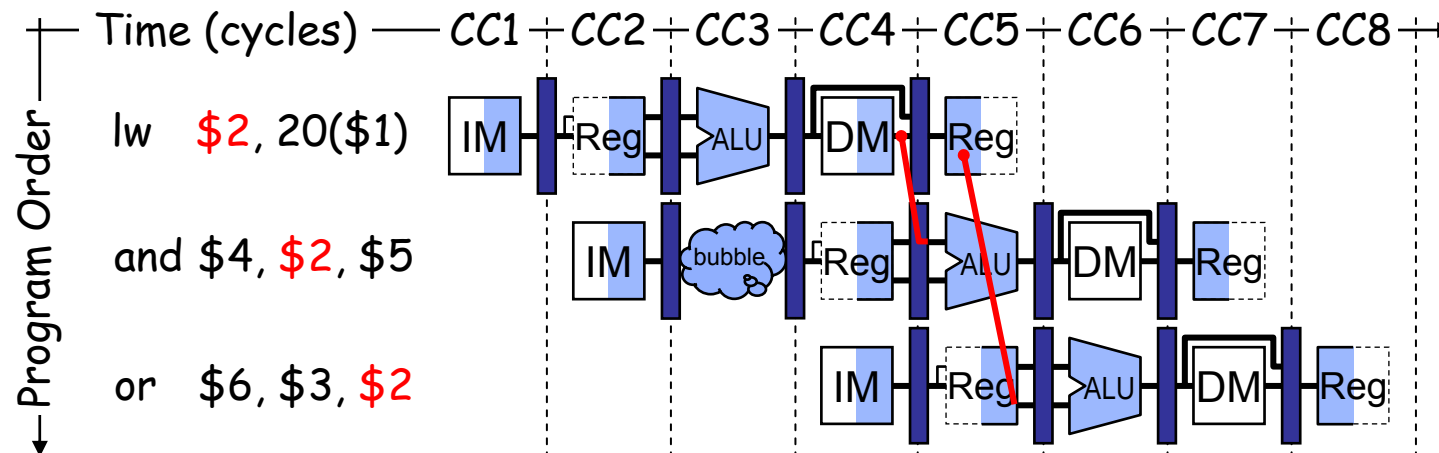


Detecting RAW Hazard after Load

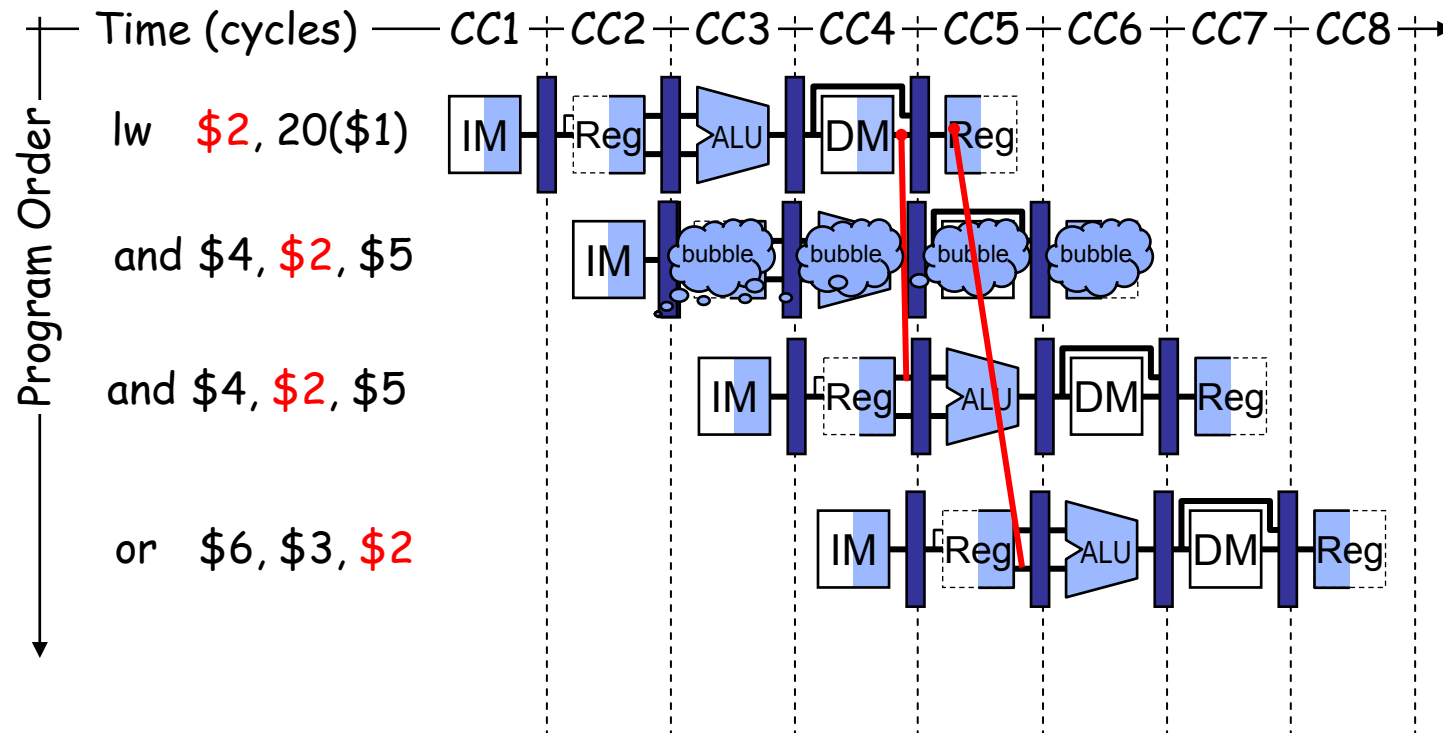
- ❖ Detecting a RAW hazard after a Load instruction:
 - ✧ The **load** instruction will be in the **ID/EX** register
 - ✧ Instruction that needs the load data will be in the **IF/ID** register
- ❖ Condition for stalling the pipeline
 - if $((ID/EX.MemRead == 1) \text{ and } (ID/EX.Rw \neq 0) \text{ and } ((ID/EX.Rw == IF/ID.Rs) \text{ or } (ID/EX.Rw == IF/ID.Rt)))$ Stall
- ❖ Insert a **bubble** after the load instruction
 - ✧ Bubble is a **no-op** that wastes one clock cycle
 - ✧ Delays the instruction after load by one cycle
 - Because of RAW hazard

Stall the Pipeline for one Cycle

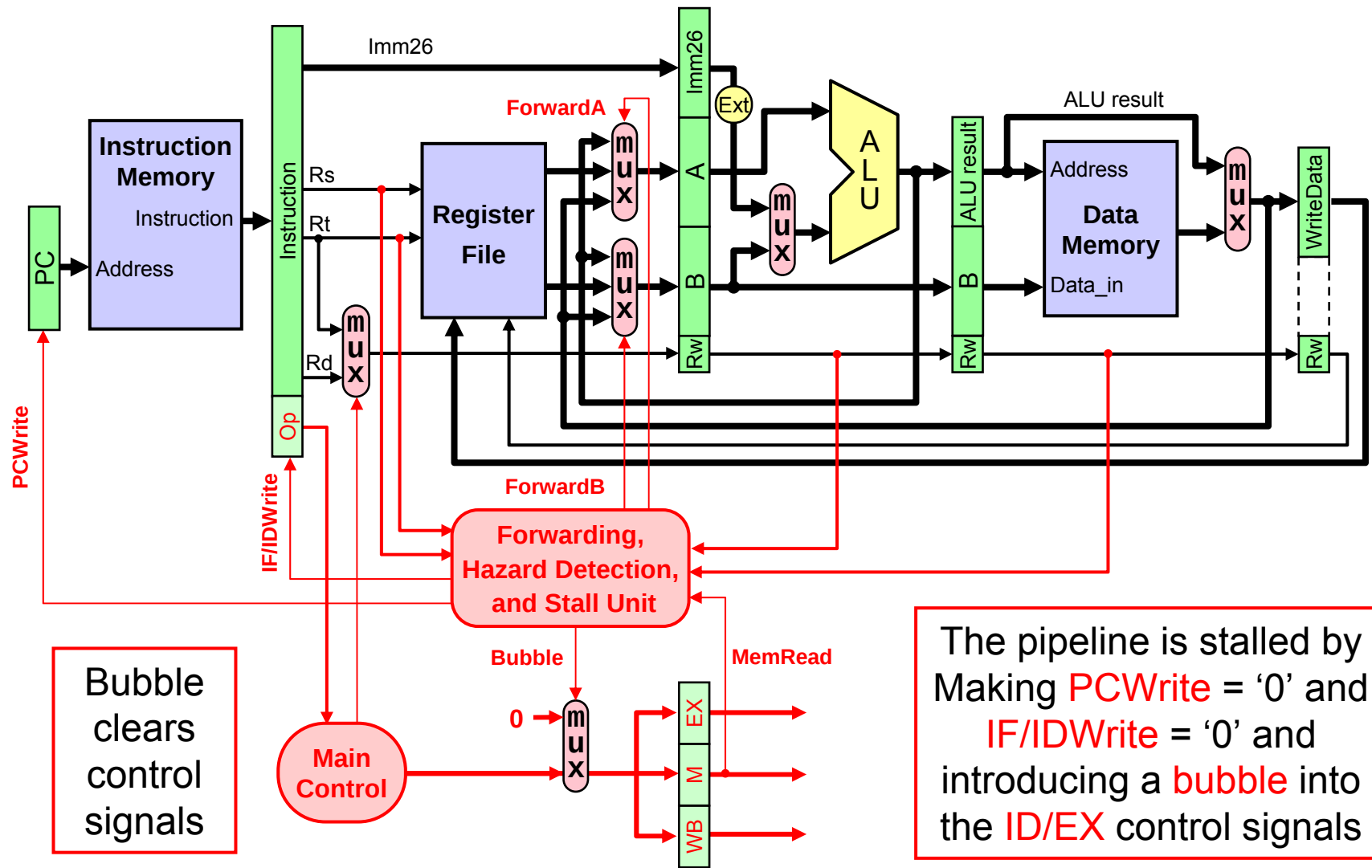
- ❖ Freeze the **PC** and the **IF/ID** registers
 - ✧ No new instruction is fetched and instruction after load is stalled
- ❖ Allow the **Load** instruction in **ID/EX** register to proceed
- ❖ Introduce a **bubble** into the **ID/EX** register
- ❖ **Load** can forward data to next instruction after delaying it



Stall the Pipeline for one Cycle



Hazard Detection and Stall Unit



Compiler Scheduling

- ❖ Compilers can schedule code in a way to avoid load stalls
- ❖ Consider the following statements:

$a = b + c; \quad d = e - f;$

❖ Slow code:

```
lw    $10, ($1)      # $1 = addr b
lw    $11, ($2)       # $2 = addr c
add   $12, $10, $11   # stall
sw    $12, ($3)       # $3 = addr a
lw    $13, ($4)       # $4 = addr e
lw    $14, ($5)       # $5 = addr f
sub   $15, $13, $14   # stall
sw    $15, ($6)       # $6 = addr d
```

❖ Fast code: No Stalls

```
lw    $10, 0($1)
lw    $11, 0($2)
lw    $13, 0($4)
lw    $14, 0($5)
add   $12, $10, $11
sw    $12, 0($3)
sub   $15, $13, $14
sw    $14, 0($6)
```



Write After Read - WAR Hazard

❖ Instruction **J** should write its result after it is read by **I**

❖ Called an **anti-dependence** by compiler writers

I: sub \$4, **\$1**, \$3 # **\$1 is read**

J: add **\$1**, \$2, \$3 # **\$1 is written**

❖ Results from reuse of the name **\$1**

❖ Hazard occurs when **J** writes **\$1** before **I** reads it

❖ Cannot occur in our basic 5-stage pipeline because:

✧ Reads are always in stage 2, and

✧ Writes are always in stage 5

✧ Instructions are processed in order

Write After Write - WAW Hazard

- ❖ Instruction **J** should write its result after instruction **I**
- ❖ Called an **output-dependence** in compiler terminology
 - I: sub \$1, \$4, \$3 # \$1 is written**
 - J: add \$1, \$2, \$3 # \$1 is written again**
- ❖ This hazard also results from the reuse of name **\$1**
- ❖ Hazard occurs when writes occur in the wrong order
- ❖ Can't happen in our basic 5-stage pipeline because:
 - ✧ All writes are ordered and always take place in stage 5
- ❖ WAR and WAW hazards can occur in complex pipelines
- ❖ **Notice that Read After Read – RAR is NOT a hazard**

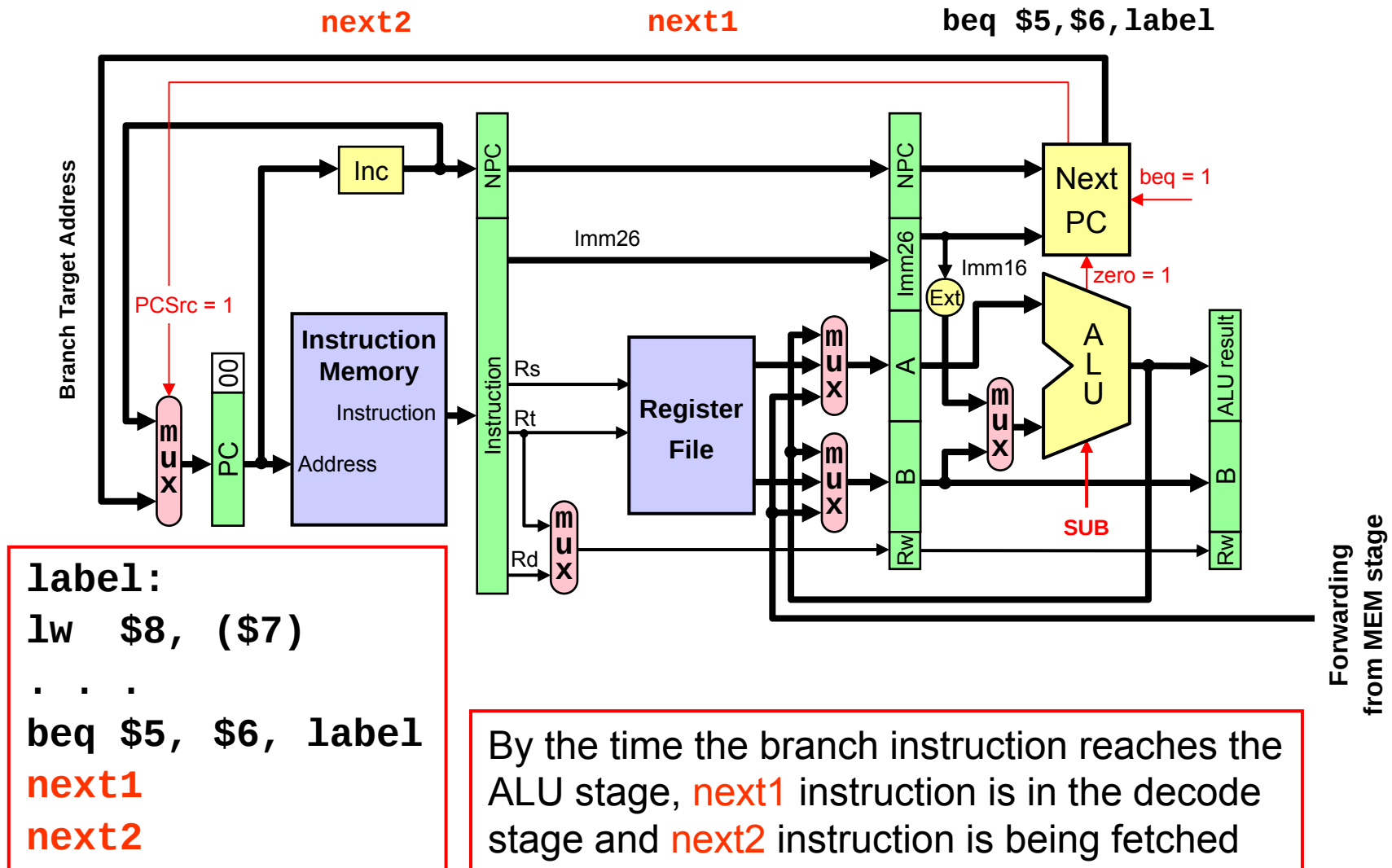
Next ...

- ❖ Pipelining versus Serial Execution
- ❖ Pipelined Datapath
- ❖ Pipelined Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall Unit
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

Control Hazards

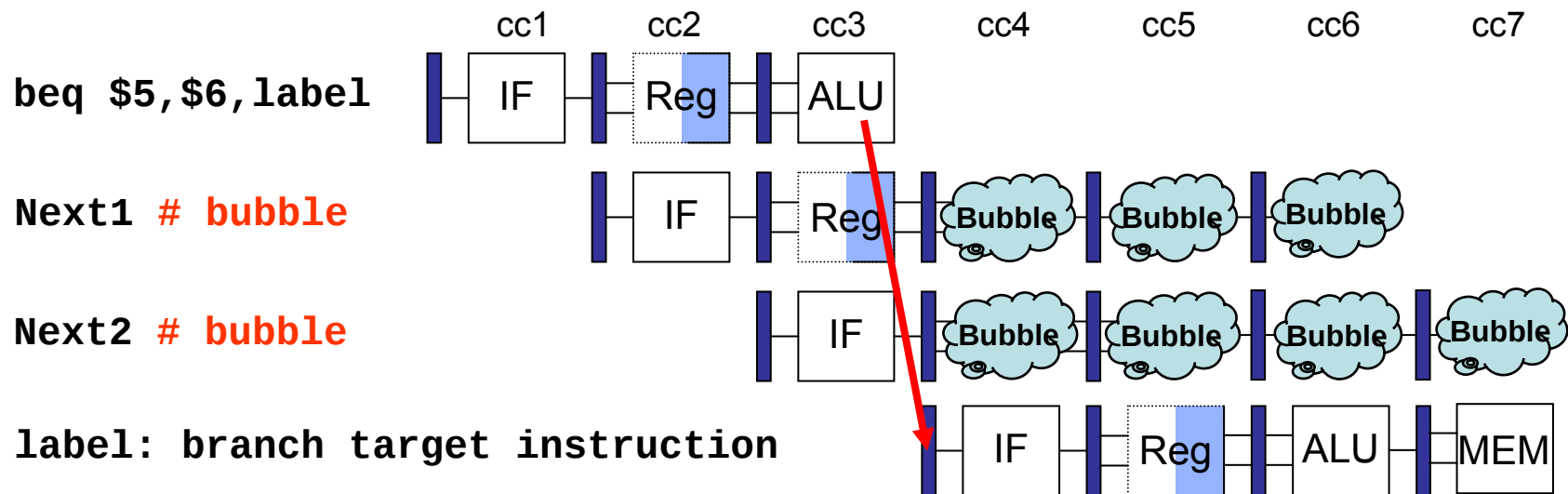
- ❖ Branch instructions can cause great performance loss
- ❖ Branch instructions need two things:
 - ✧ Branch Result Taken or Not Taken
 - ✧ Branch target
 - $PC + 4$ If Branch is NOT taken
 - $PC + 4 + 4 \times \text{immediate}$ If Branch is Taken
- ❖ Branch instruction is decoded in the ID stage
 - ✧ At which point a new instruction is already being fetched
- ❖ For our pipeline: 2-cycle branch delay
 - ✧ Effective address is calculated in the ALU stage
 - ✧ Branch condition is determined by the ALU (zero flag)

Branch Delay = 2 Clock Cycles



2-Cycle Branch Delay

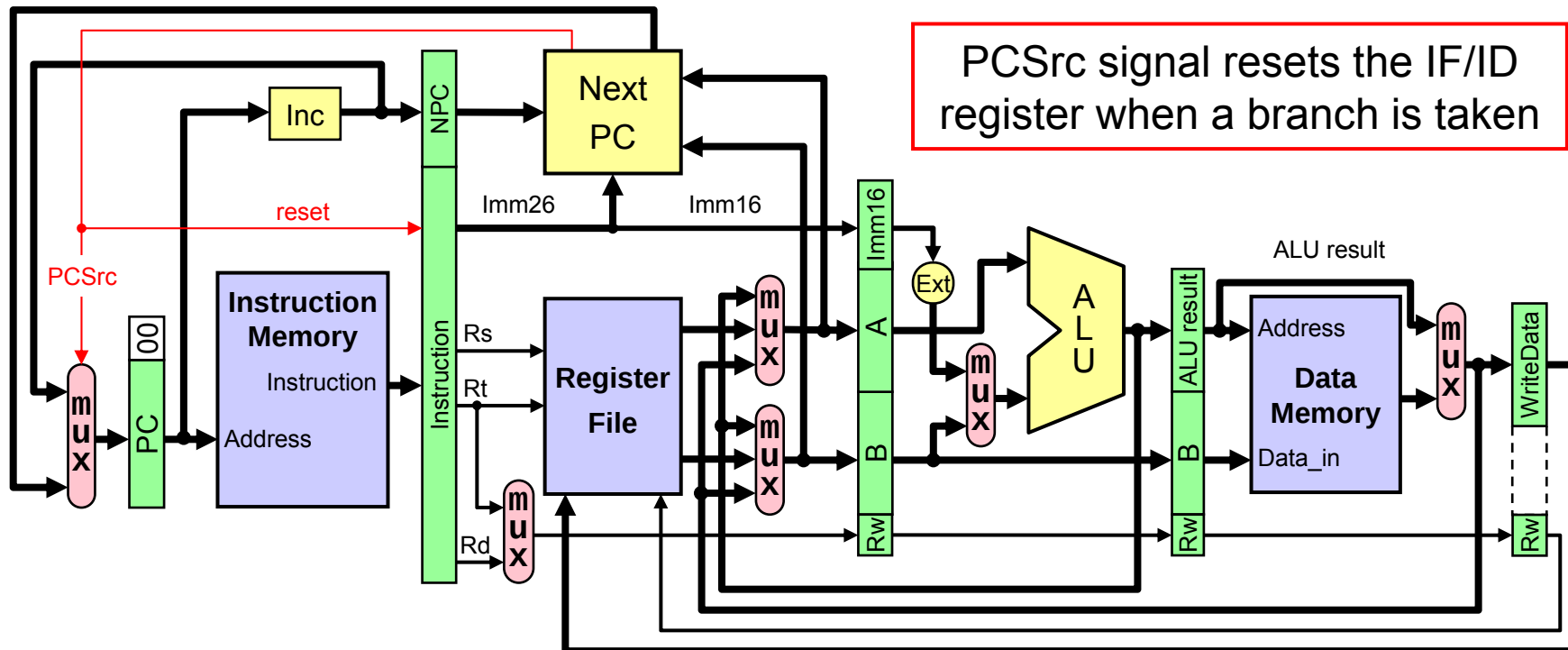
- ❖ **Next1** thru **Next2** instructions will be fetched anyway
- ❖ Pipeline should flush **Next1** and **Next2** if branch is **taken**
- ❖ Otherwise, they can be executed if branch is not taken



Reducing the Delay of Branches

- ❖ Branch delay can be reduced from 2 cycles to **just 1 cycle**
- ❖ Branches can be determined earlier in the Decode stage
 - ✧ **Next PC** logic block is moved to the **ID stage**
 - ✧ A comparator is added to the Next PC logic
 - To determine branch decision, whether the branch is taken or not
- ❖ Only **one instruction** that follows the branch will be fetched
- ❖ If the branch is taken then only one instruction is flushed
- ❖ We need a control signal to **reset the IF/ID register**
 - ✧ This will convert the fetched instruction into a NOP

Modified Datapath

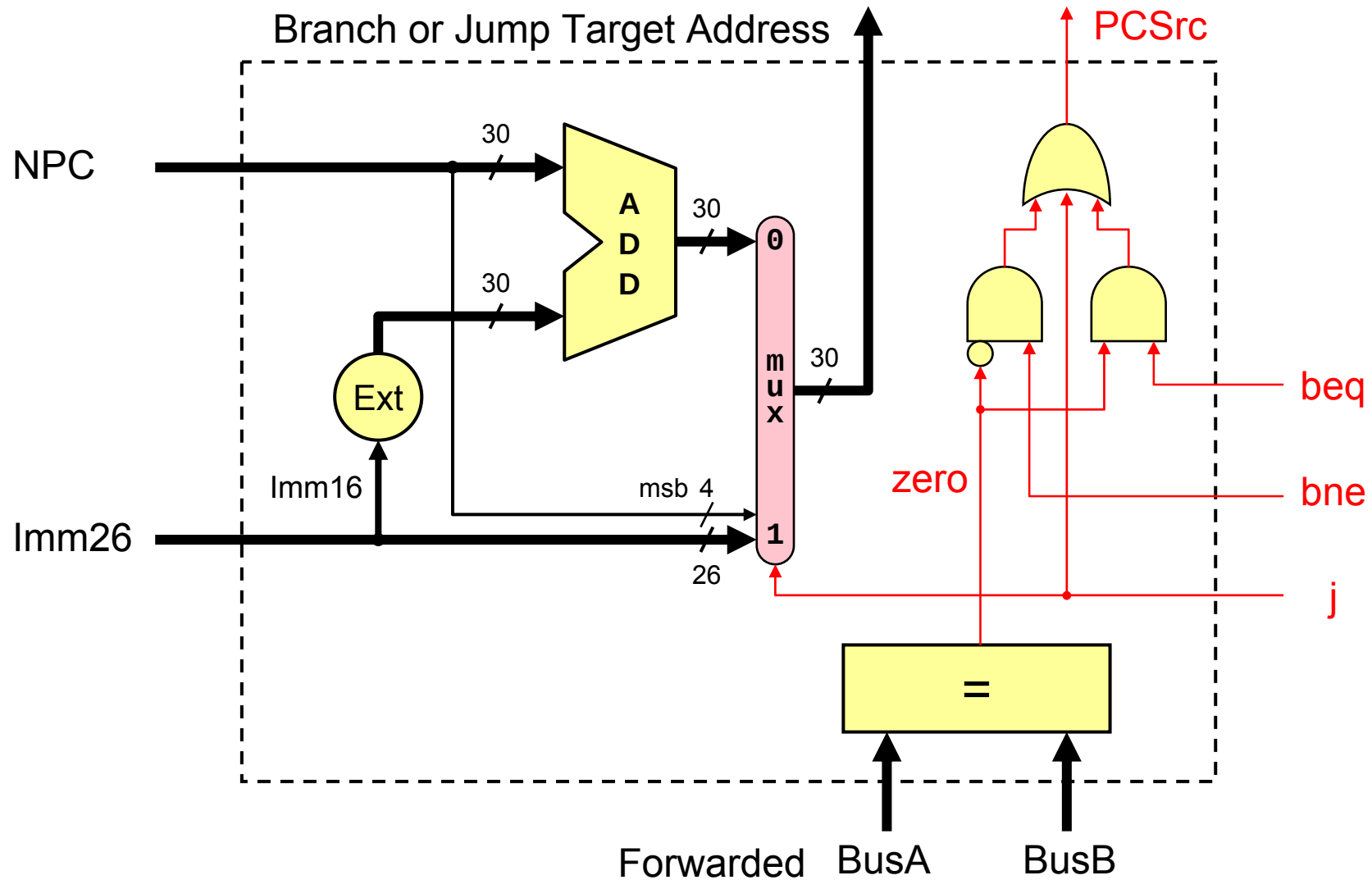


Next PC block is moved to the Instruction Decode stage

Advantage: Branch and jump delay is reduced to one cycle

Drawback: Added delay in decode stage => longer cycle

Details of Next PC



Next ...

- ❖ Pipelining versus Serial Execution
- ❖ Pipelined Datapath
- ❖ Pipelined Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall Unit
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

Branch Hazard Alternatives

- ❖ **Predict Branch Not Taken** (modified datapath)
 - ✧ Successor instruction is already fetched
 - ✧ About half of MIPS branches are not taken on average
 - ✧ Flush instructions in pipeline only if branch is actually taken
- ❖ **Delayed Branch**
 - ✧ Define branch to take place **AFTER** the next instruction
 - ✧ Compiler/assembler **fills the branch delay slot (for 1 delay cycle)**
- ❖ **Dynamic Branch Prediction**
 - ✧ Can predict backward branches in loops $\Rightarrow$ taken most of time
 - ✧ However, **branch target address is determined in ID stage**
 - ✧ Must reduce branch delay from 1 cycle to 0, but how?

Delayed Branch

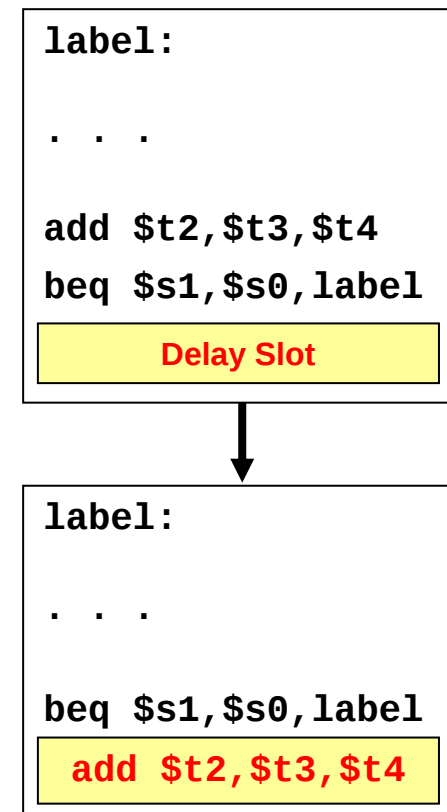
- ❖ Define branch to take place **after** the next instruction
- ❖ For a 1-cycle branch delay, we have one **delay slot**

branch instruction

branch delay slot (next instruction)

branch target (if branch taken)

- ❖ Compiler **fills the branch delay slot**
 - ✧ By selecting an **independent instruction**
 - ✧ From before the branch
- ❖ If no independent instruction is found
 - ✧ Compiler fills delay slot with a NO-OP



Zero-Delayed Branch

❖ Disadvantages of delayed branch

- ✧ Branch delay can increase to multiple cycles in deeper pipelines
- ✧ Branch delay slots must be filled with useful instructions or no-op

❖ How can we achieve **zero-delay for a taken branch?**

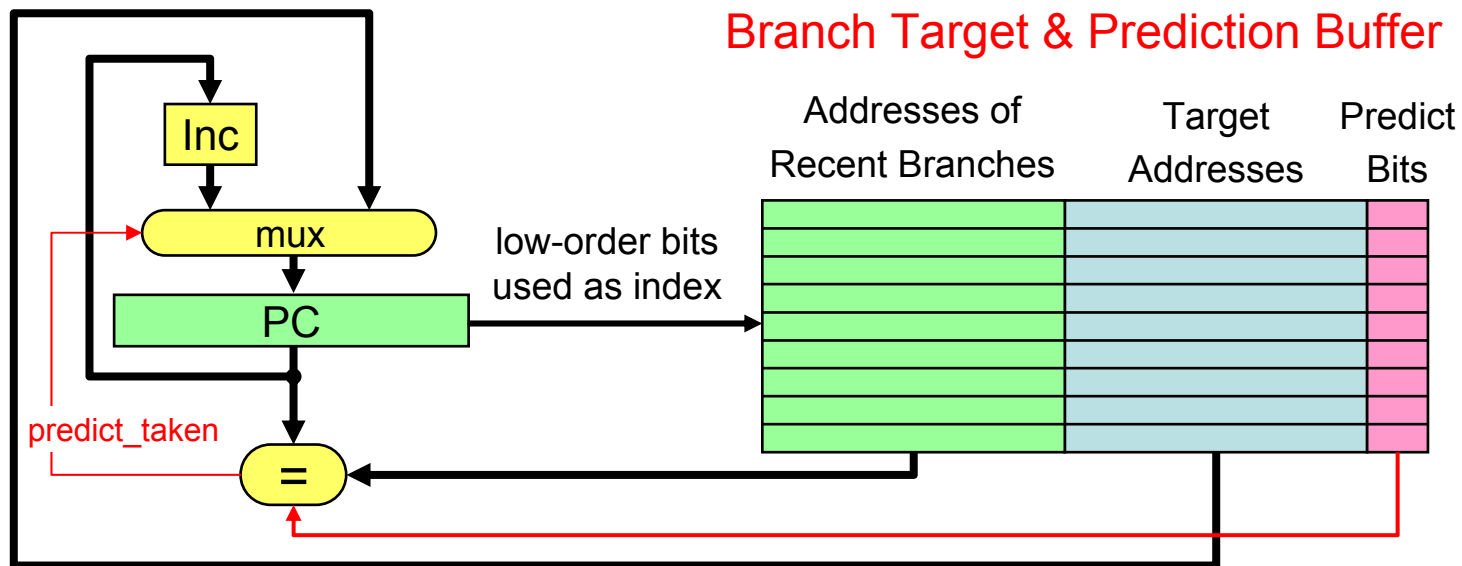
- ✧ Branch target address is computed in the ID stage

❖ **Solution**

- ✧ Check the PC to see if the instruction being fetched is a branch
- ✧ Store the **branch target address** in a **branch buffer** in the **IF stage**
- ✧ If branch is predicted taken then
 - **Next PC = branch target fetched from branch target buffer**
- ✧ Otherwise, if branch is predicted not taken then **Next PC = PC+4**

Branch Target and Prediction Buffer

- ❖ The **branch target buffer** is implemented as a small cache
 - ✧ Stores the branch target address of recent branches
- ❖ We must also have **prediction bits**
 - ✧ To **predict** whether branches are taken or not taken
 - ✧ The prediction bits are dynamically determined by the hardware

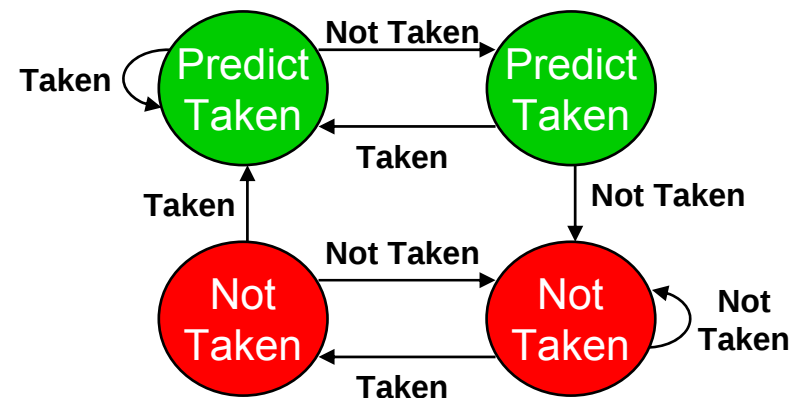


Dynamic Branch Prediction

- ❖ Prediction of branches at runtime using **prediction bits**
 - ✧ One or few prediction bits are associated with a branch instruction
- ❖ Branch prediction buffer is a small memory
 - ✧ Indexed by the lower portion of the address of branch instruction
- ❖ The simplest scheme is to have 1 prediction bit per branch
- ❖ We don't know if the prediction bit is correct or not
- ❖ If correct prediction ...
 - ✧ Continue normal execution – no wasted cycles
- ❖ If incorrect prediction (misprediction) ...
 - ✧ Flush the instructions that were incorrectly fetched – wasted cycles
 - ✧ Update prediction bit and target address for future use

2-bit Prediction Scheme

- ❖ Prediction is just a hint that is assumed to be correct
- ❖ If incorrect then fetched instructions are flushed
- ❖ 1-bit prediction scheme has a performance shortcoming
 - ✧ A loop branch is almost always taken, except for last iteration
 - ✧ 1-bit scheme will mispredict twice, on first and last loop iterations
- ❖ 2-bit prediction schemes work better and are often used
 - ✧ A prediction must be wrong twice before it is changed
 - ✧ A loop branch is mispredicted only once on the last iteration



Pipeline Hazards Summary

❖ Three types of pipeline hazards

- ✧ Structural hazards: conflicts using a resource during same cycle
- ✧ Data hazards: due to data dependencies between instructions
- ✧ Control hazards: due to branch and jump instructions

❖ Hazards limit the performance and complicate the design

- ✧ Structural hazards: eliminated by careful design or more hardware
- ✧ Data hazards are eliminated by forwarding
- ✧ However, load delay cannot be eliminated and stalls the pipeline
- ✧ Delayed branching can be a solution when branch delay = 1 cycle
- ✧ Branch prediction can reduce branch delay to zero
- ✧ Branch misprediction should flush the wrongly fetched instructions