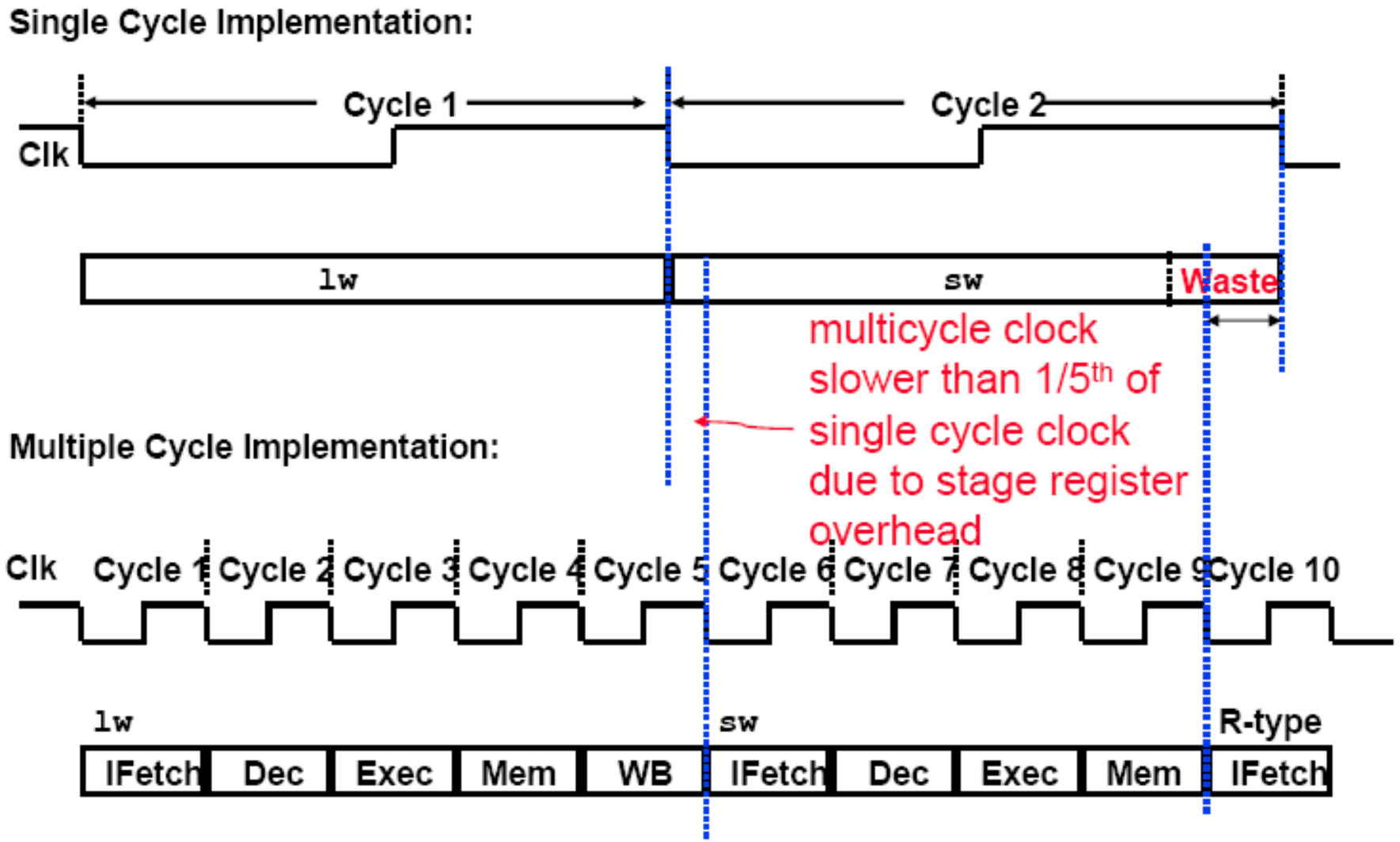
Pipelined Processor Design

Presentation Outline

- **Introduction**
- Pipelined Datapath and Control
- Pipeline Hazards
- Data Hazards and Forwarding
- Load Delay, Hazard Detection, and Stall
- Control Hazards
- Delayed Branch and Dynamic Branch Prediction

Review: Single Cycle vs. Multiple Cycle Timing



How Can We Make It Even Faster?

- Split the multiple instruction cycle into smaller and smaller steps
- There is a point of diminishing returns where as much time is spent loading the state registers as doing the work
- Start fetching and executing the next instruction before the current one has completed
- Pipelining–(all?) modern processors are pipelined for performance
- Fetch (and execute) more than one instruction at a time (out-of-order superscalar and VLIW (epic))
- Fetch (and execute) instructions from more than one instruction stream (multithreading (hyperthreading))

Pipelining

- Pipelining is a general-purpose efficiency technique
  - It is not specific to processors
- Pipelining is used in:
  - Assembly lines
  - Bucket brigades
  - Fast food restaurants
- Pipelining is used in other CS disciplines:
  - Networking
  - Server software architecture
- Useful to increase throughput in the presence of long latency
  - More on that later…

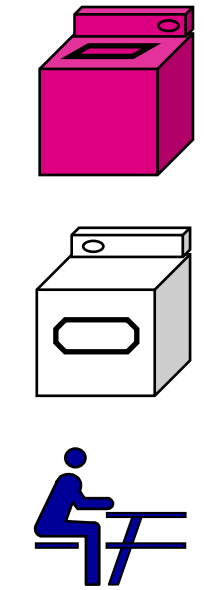
Pipelining Example

- Laundry Example: Three Stages

1. Wash dirty load of clothes

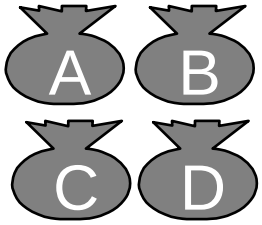
2. Dry wet clothes

3. Fold and put clothes into drawers

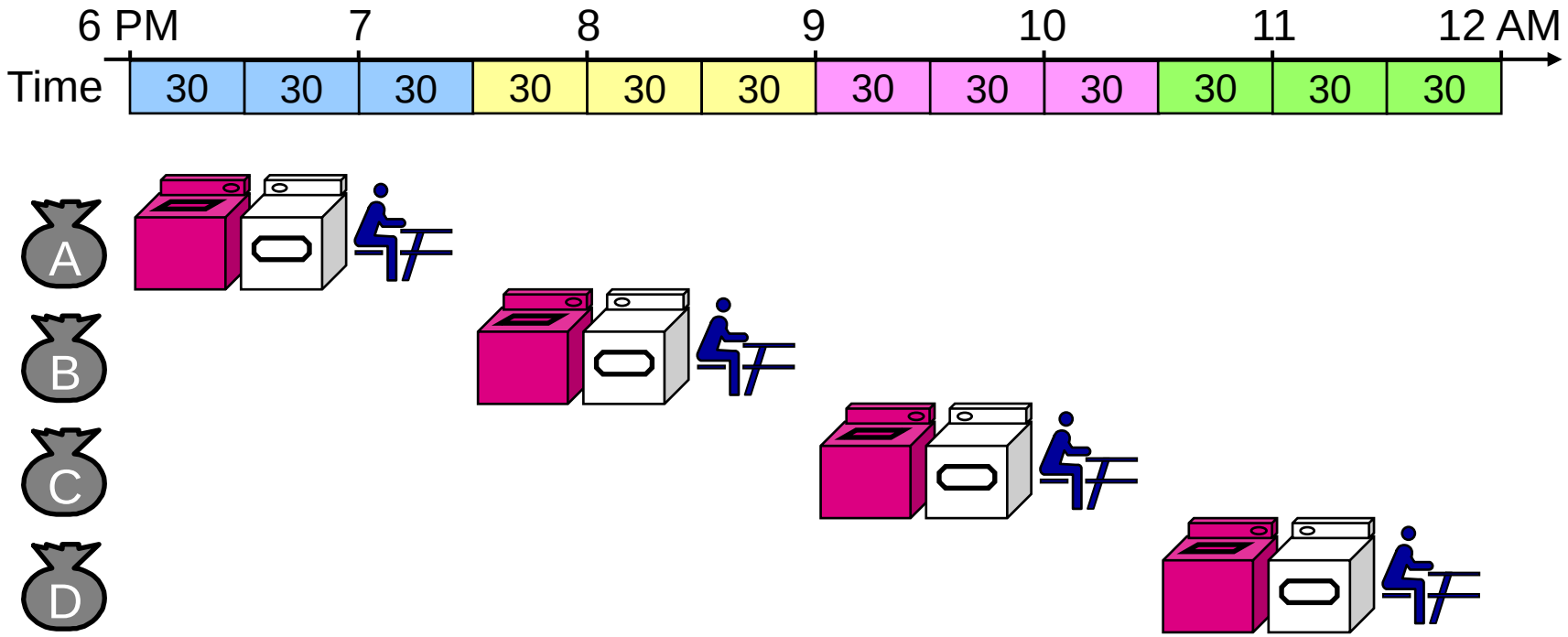


- Each stage takes 30 minutes to complete

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

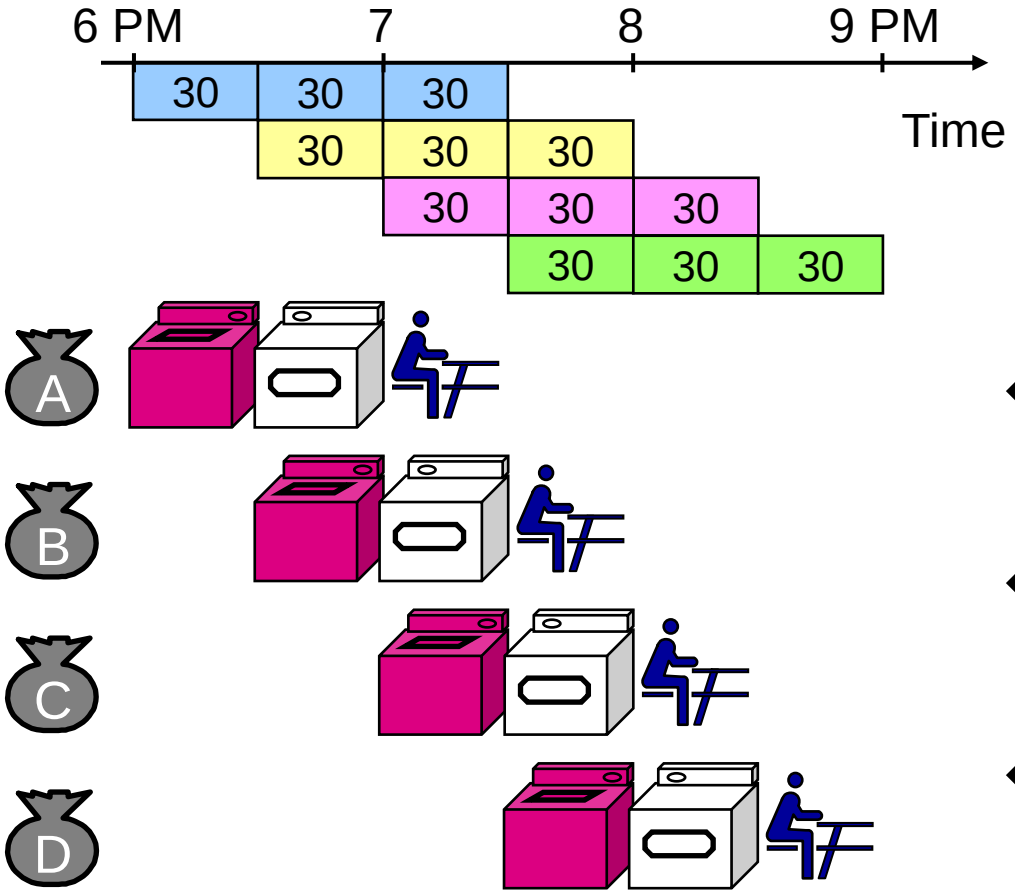


Sequential Laundry



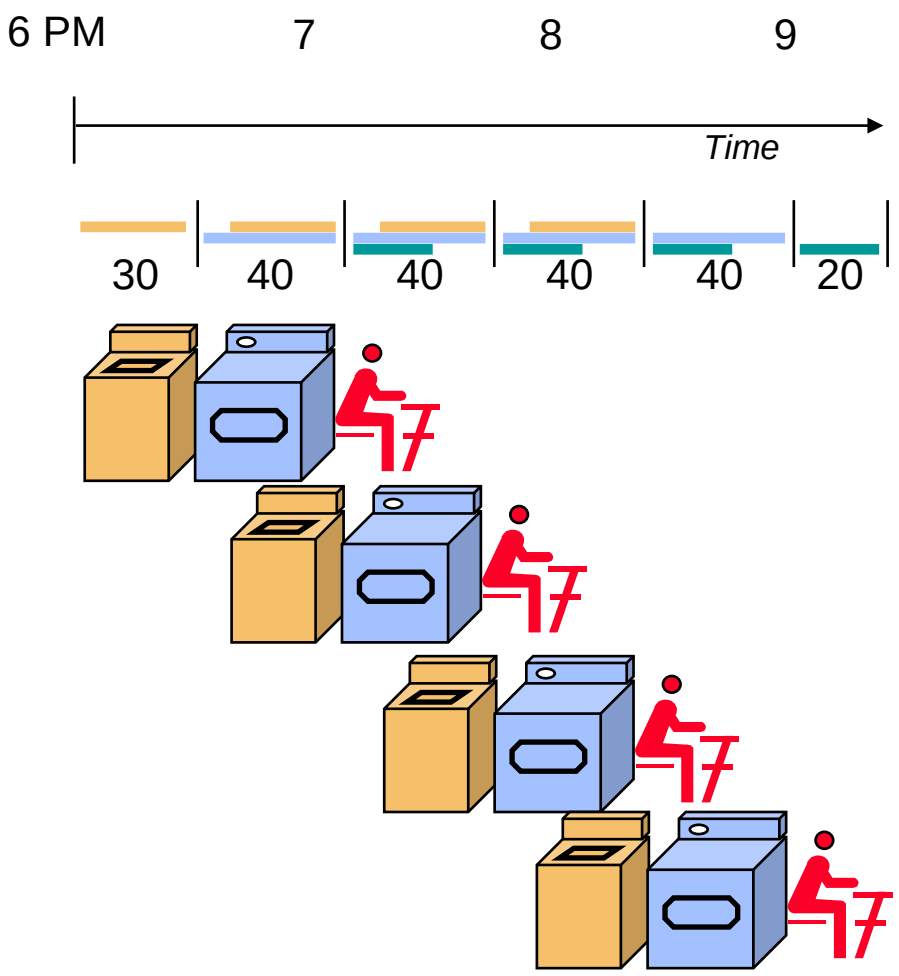
- Sequential laundry takes 6 hours for 4 loads
- Intuitively, we can use pipelining to speed up laundry

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)

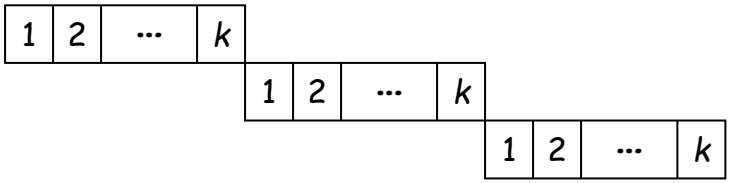
Pipelining Lessons



- Pipelining doesn't help latency of single load, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

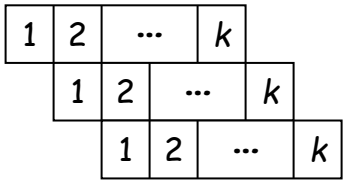
Serial Execution versus Pipelining

- 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 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



Without Pipelining

One completion every *k* time units



With Pipelining

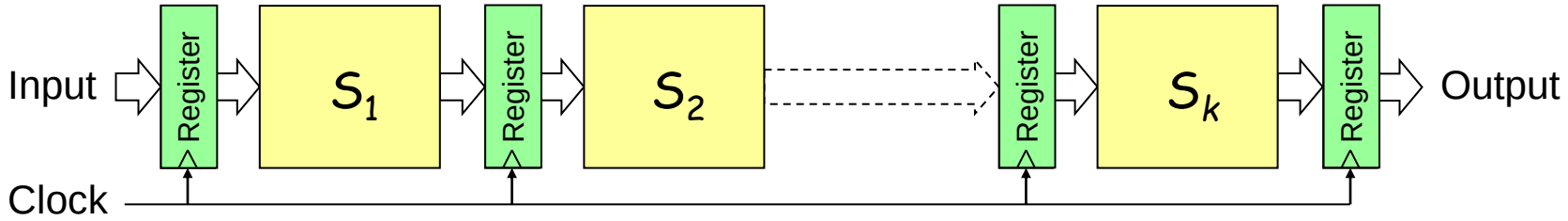
One completion every 1 time unit

Instruction Pipelining

- Instruction pipelining is a CPU implementation technique where multiple operations on a number of instructions are overlapped.
  - For Example: The next instruction is fetched in the next cycle without waiting for the current instruction to complete.
- An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction. Each step is called *a pipeline stage* or *a pipeline segment.*
- The stages or steps are connected in a linear fashion: one stage to the next to form the pipeline (or pipelined CPU datapath) -- instructions enter at one end and progress through the stages and exit at the other end.
- The time to move an instruction one step down the pipeline is equal to *the machine (CPU) cycle* and is determined by the stage with the longest processing delay.

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



Pipelining: Design Goals

- The length of the machine clock cycle is determined by the time required for the slowest pipeline stage.
- An important pipeline design consideration is to balance the length of each pipeline stage.
- If all stages are perfectly balanced, then the time per instruction on a pipelined machine (assuming ideal conditions with no stalls):

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipeline stages}}$$

- Under these ideal conditions:
  - Speedup from pipelining = the number of pipeline stages = n
  - Goal: One instruction is completed every cycle: CPI = 1 .

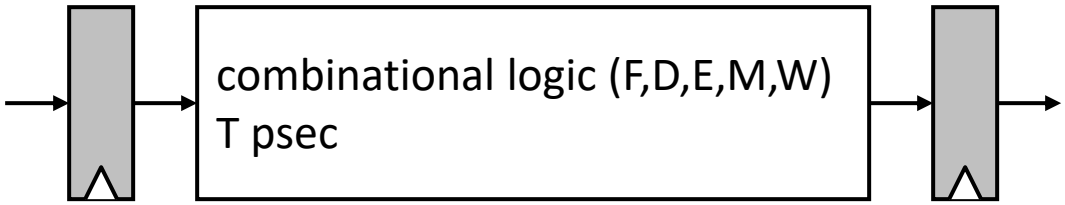
An Ideal Pipeline

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)

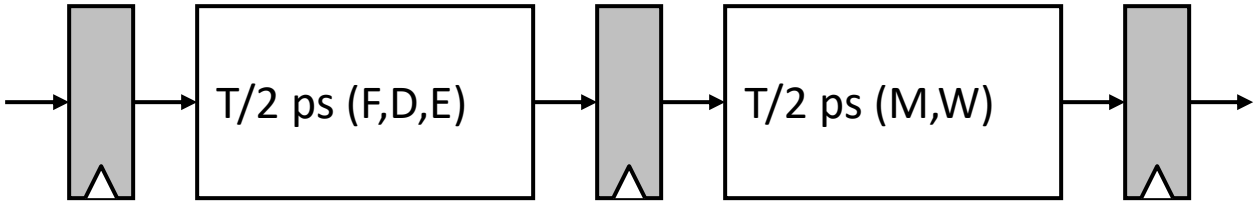
- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- Repetition of independent operations
  - No dependences between repeated operations
- Uniformly partitionable suboperations
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing “cycle”?

Ideal Pipelining

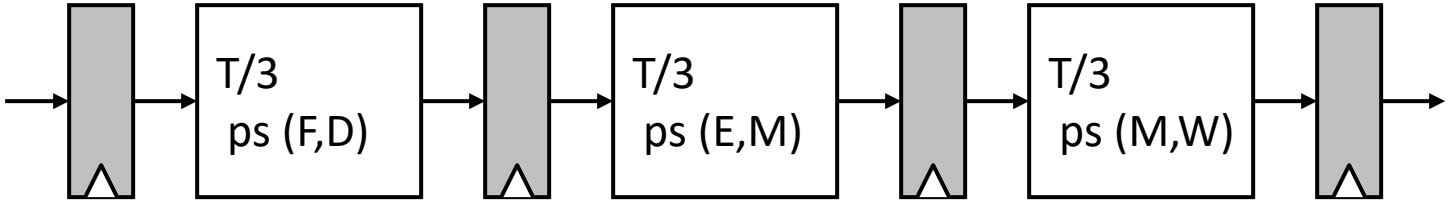
Tput = Throughput



Tput=~(1/T)



Tput=~(2/T)



Tput=~(3/T)

Instruction Pipeline: Not An Ideal Pipeline

- Identical operations ... NOT!
  - ⇒ different instructions → not all need the same stages
    - Forcing different instructions to go through the same pipe stages
    - → external fragmentation (some pipe stages idle for some instructions)
- Uniform suboperations ... NOT!
  - ⇒ different pipeline stages → not the same latency
    - Need to force each stage to be controlled by the same clock
    - → internal fragmentation (some pipe stages are fast but still have to take the same clock cycle time)
- Independent operations ... NOT!
  - ⇒ instructions are not independent of each other
    - Need to detect and resolve inter-instruction dependences to ensure the pipeline provides correct results
    - → pipeline stalls (pipeline is not always moving)

Pipeline Speedup

- 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

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

More Realistic Pipeline: Throughput

- Nonpipelined version with delay T

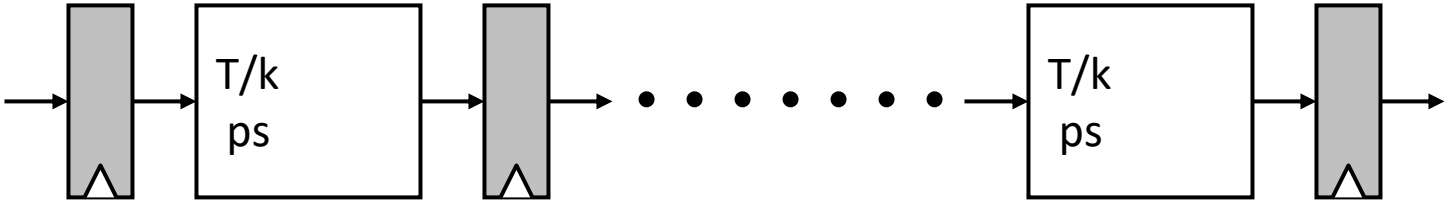
  Tput = 1 / (T+S) where S = register (sequential logic) delay

- k-stage pipelined version

Register delay reduces throughput (sequencing overhead b/w stages)

$Tput_{k\text{-}stage} = 1 / (T/k + S)$

$Tput_{max} = 1 / (1 \text{ gate delay} + S)$



This picture assumes "perfect division of work between stages (T/k)"

More Realistic Pipeline: Cost

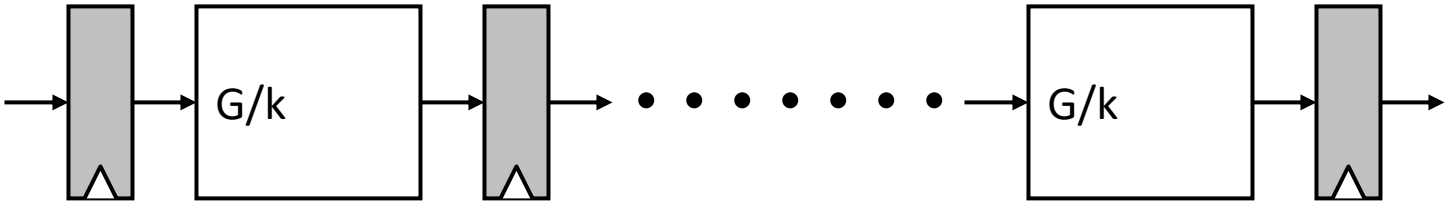
- Nonpipelined version with combinational cost G

  Cost = G+R where R = register cost

- k-stage pipelined version

Registers increase hardware cost

$Cost_{k\text{-}stage} = G + Rk$



This picture ignores resource and register replication that is likely needed (G/k and Rk)

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

Issues in Pipeline Design

- Balancing work in pipeline stages
  - How many stages and what is done in each stage
- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations
- Handling exceptions, interrupts
- Advanced: Improving pipeline throughput
  - Minimizing *stalls*

Example of Pipeline Performance

- Consider a 5-stage instruction execution pipeline …
  - Instruction fetch = ALU = Data memory access = 350 ps
  - Register read = Register write = 250 ps
- Compare single-cycle, multi-cycle, versus pipelined
  - Assume: 20% load, 10% store, 40% ALU, and 30% branch
- **Solution:**

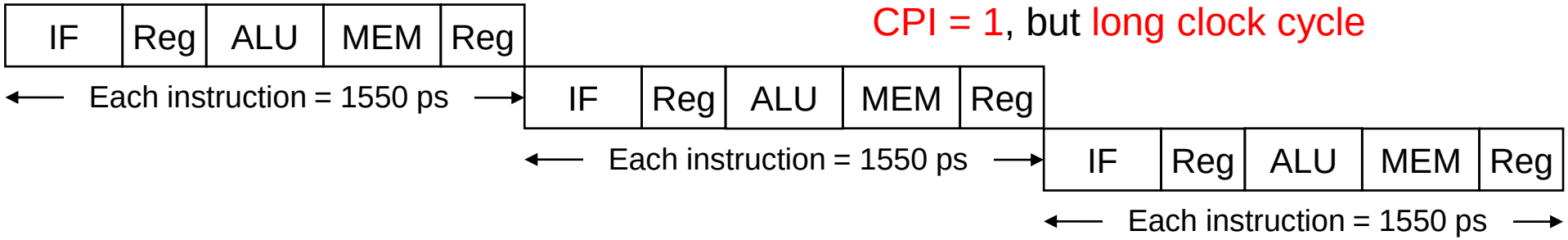
Instruction	Fetch	Reg Read	ALU	Memory	Reg Wr	Time
Load	350 ps	250 ps	350 ps	350 ps	250 ps	1550 ps
Store	350 ps	250 ps	350 ps	350 ps		1300 ps
ALU	350 ps	250 ps	350 ps		250 ps	1200 ps
Branch	350 ps	250 ps	350 ps			950 ps

Single-Cycle, Multi-Cycle, Pipelined

Single-Cycle Execution:

T_{clock} = 350+250+350+350+250 = 1550 ps

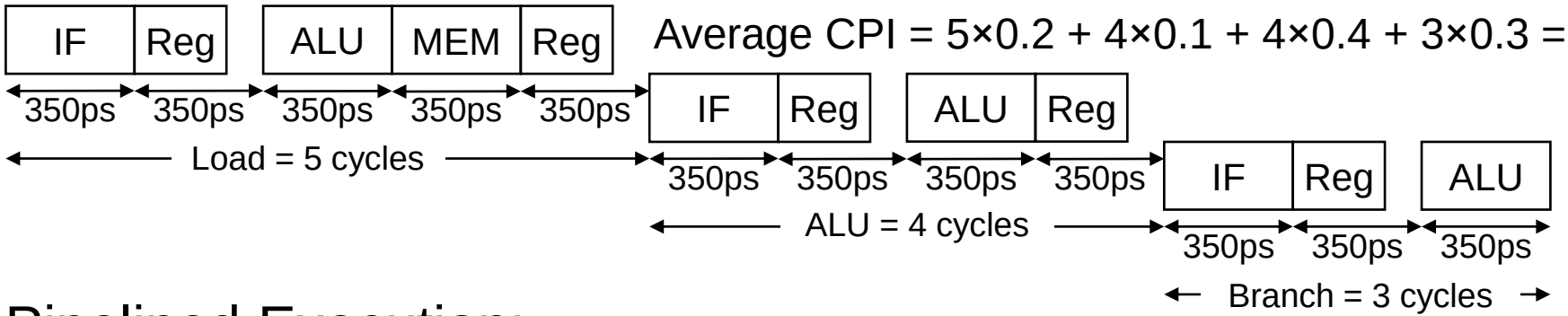
CPI = 1, but long clock cycle



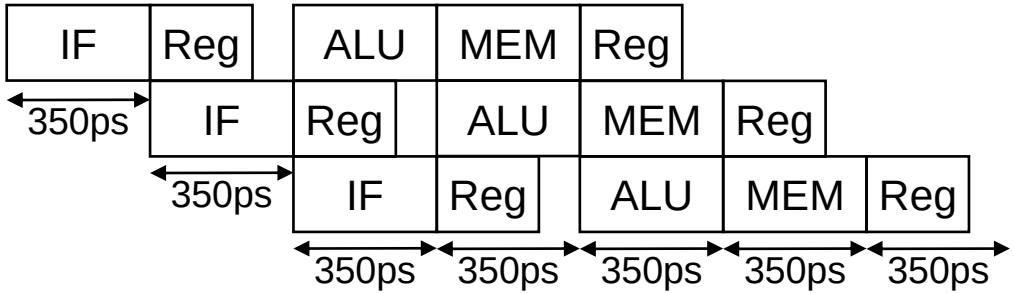
Multi-Cycle Execution:

T_{clock} = 350 ps

Average CPI = 5×0.2 + 4×0.1 + 4×0.4 + 3×0.3 = 3.9



Pipelined Execution:



T_{clock} = 350 ps = max(350, 250)

One instruction completes each cycle

Average CPI = 1

Ignore time to fill pipeline

Single-Cycle, Multi-Cycle, Pipelined

- Single-Cycle CPI = 1, but long clock cycle = 1550 ps
  - Time of each instruction = 1550 ps
- Multi-Cycle Clock = 350 ps (faster clock than single-cycle)
  - But average CPI = 3.9 (worse than single-cycle)
  - Average time per instruction = 350 ps × 3.9 = 1365 ps
  - Multi-cycle is faster than single-cycle by: 1550/1365 = 1.14x
- Pipeline Clock = 350 ps (same as multi-cycle)
  - But average CPI = 1 (one instruction completes per cycle)
  - Average time per instruction = 350 ps × 1 = 350 ps
  - Pipeline is faster than single-cycle by: 1550/350 = 4.43x
  - Pipeline is also faster than multi-cycle by: 1365/350 = 3.9x

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

Presentation Outline

- Introduction
- **Pipelined Datapath and Control**
- Pipeline Hazards
- Data Hazards and Forwarding
- Load Delay, Hazard Detection, and Stall
- Control Hazards
- Delayed Branch and Dynamic Branch Prediction

Basic Pipelined CPU Design Steps

1. Analyze instruction set operations using independent RTN => datapath requirements.
2. Select required datapath components and connections.
3. Assemble an initial datapath meeting the ISA requirements.
4. Identify pipeline stages based on operation, balancing stage delays, and ensuring no hardware conflicts exist when common hardware is used by two or more stages simultaneously in the same cycle.
5. Divide the datapath into the stages identified above by adding buffers between the stages of sufficient width to hold:
   - Instruction fields.
   - Remaining control lines needed for remaining pipeline stages.
   - All results produced by a stage and any unused results of previous stages.
6. Analyze implementation of each instruction to determine setting of control points that effects the register transfer taking pipeline hazard conditions into account . (More on this a bit later)
7. Assemble the control logic.

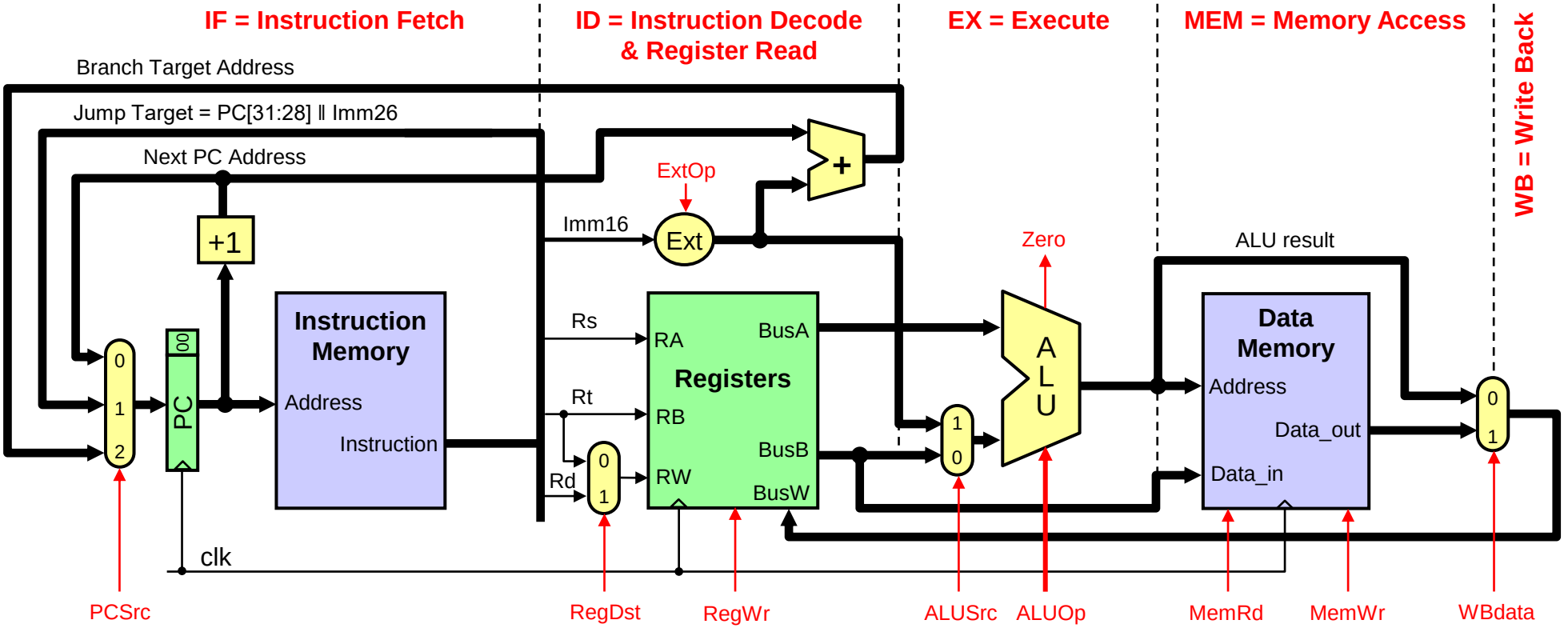
Pipelining the MIPS ISA

- What makes it easy
  - all instructions are the same length (32 bits)
    - can fetch in the 1ststage and decode in the 2ndstage
  - few instruction formats (three) with symmetryacross formats
    - can begin reading register file in 2ndstage
  - memory operations can occur only in loads and stores
    - can use the execute stage to calculate memory addresses
  - each MIPS instruction writes at most one result (i.e., changes the machine state) and does so near the end of the pipeline (MEM and WB)
- What makes it hard
  - structural hazards: what if we had only one memory?
  - control hazards: what about branches?
  - data hazards: what if an instruction's input operands depend on the output of a previous instruction?

Single-Cycle Datapath

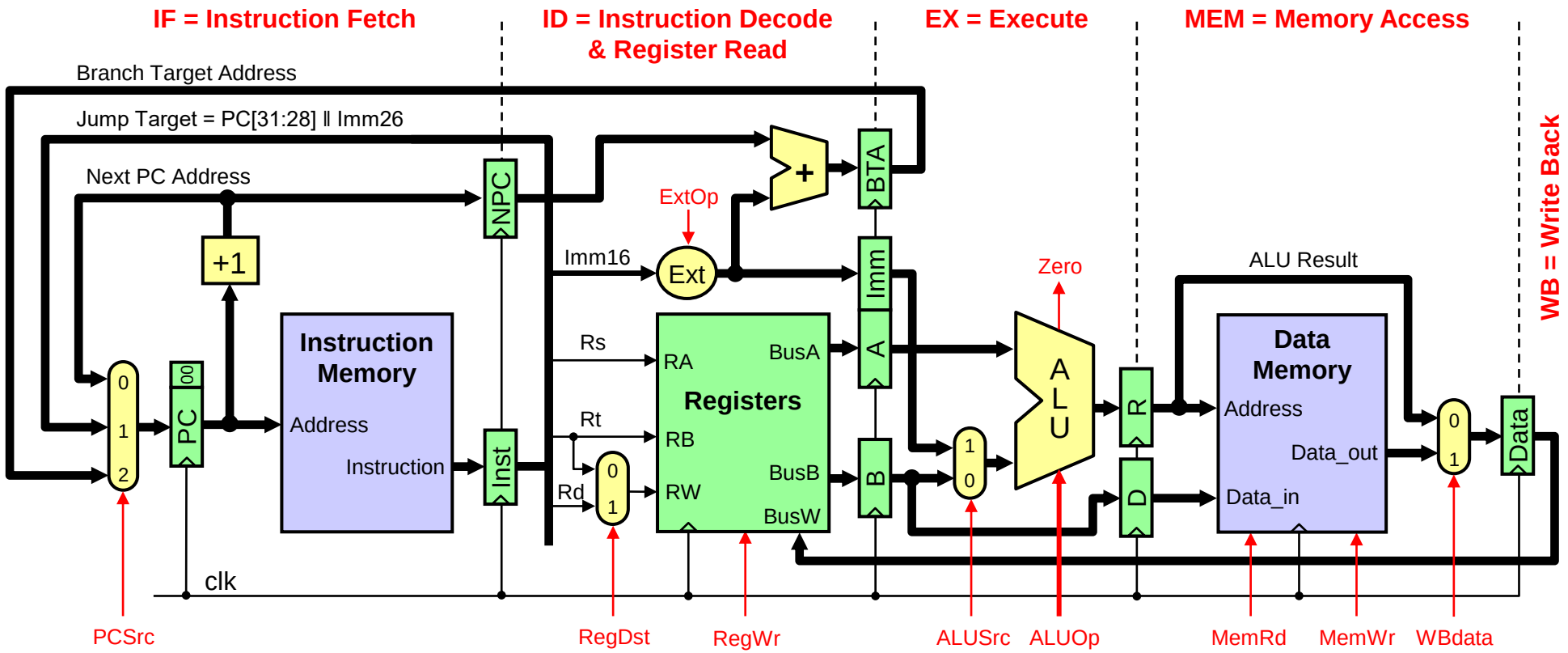
- Shown below is the single-cycle datapath
- How to pipeline this single-cycle datapath?

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



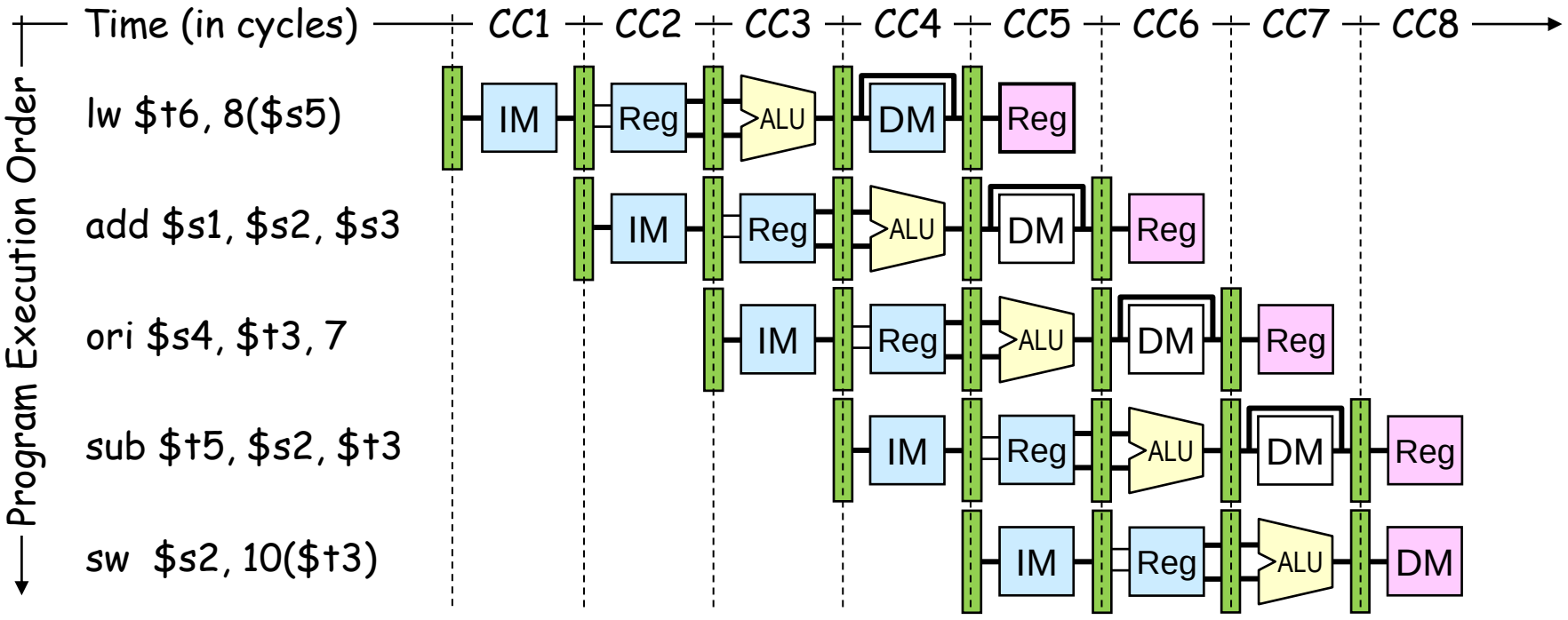
Pipelined Datapath

- 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)



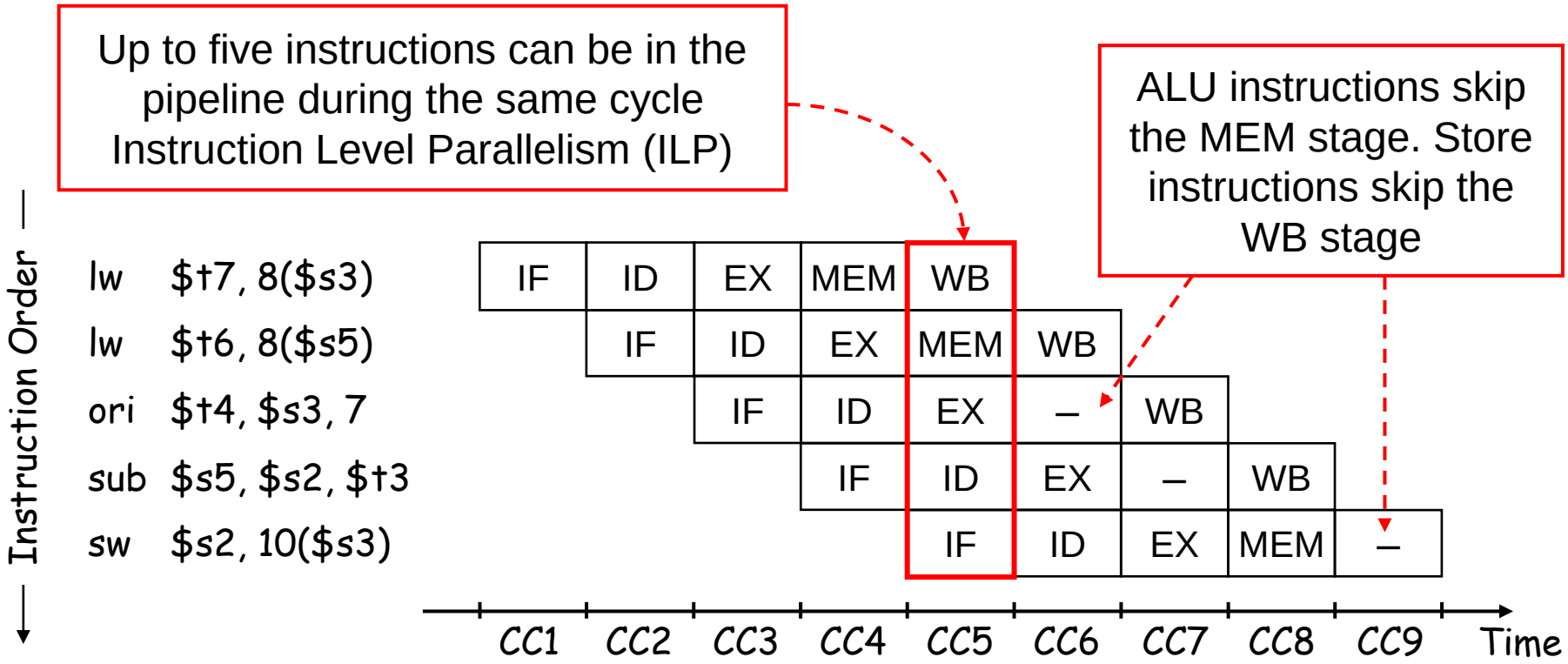
Graphically Representing Pipelines

- 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



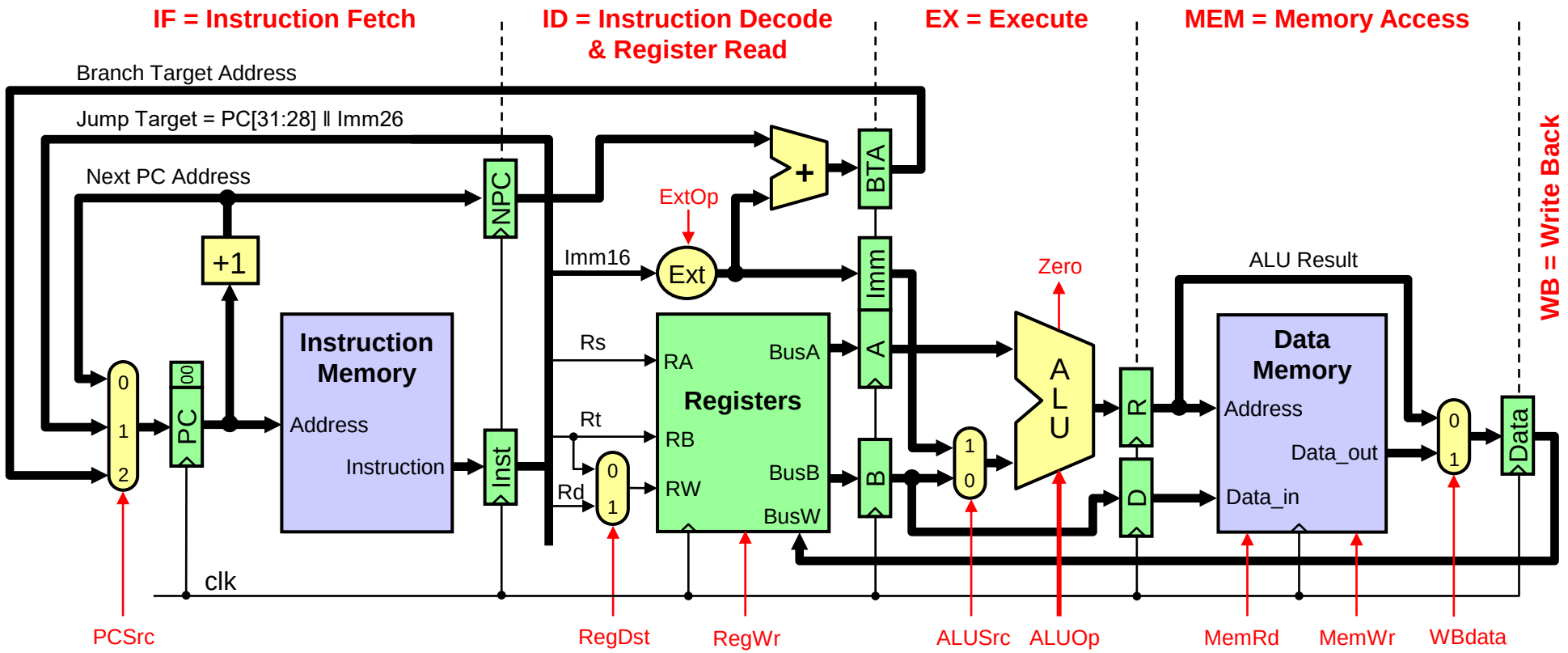
Instruction-Time Diagram

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



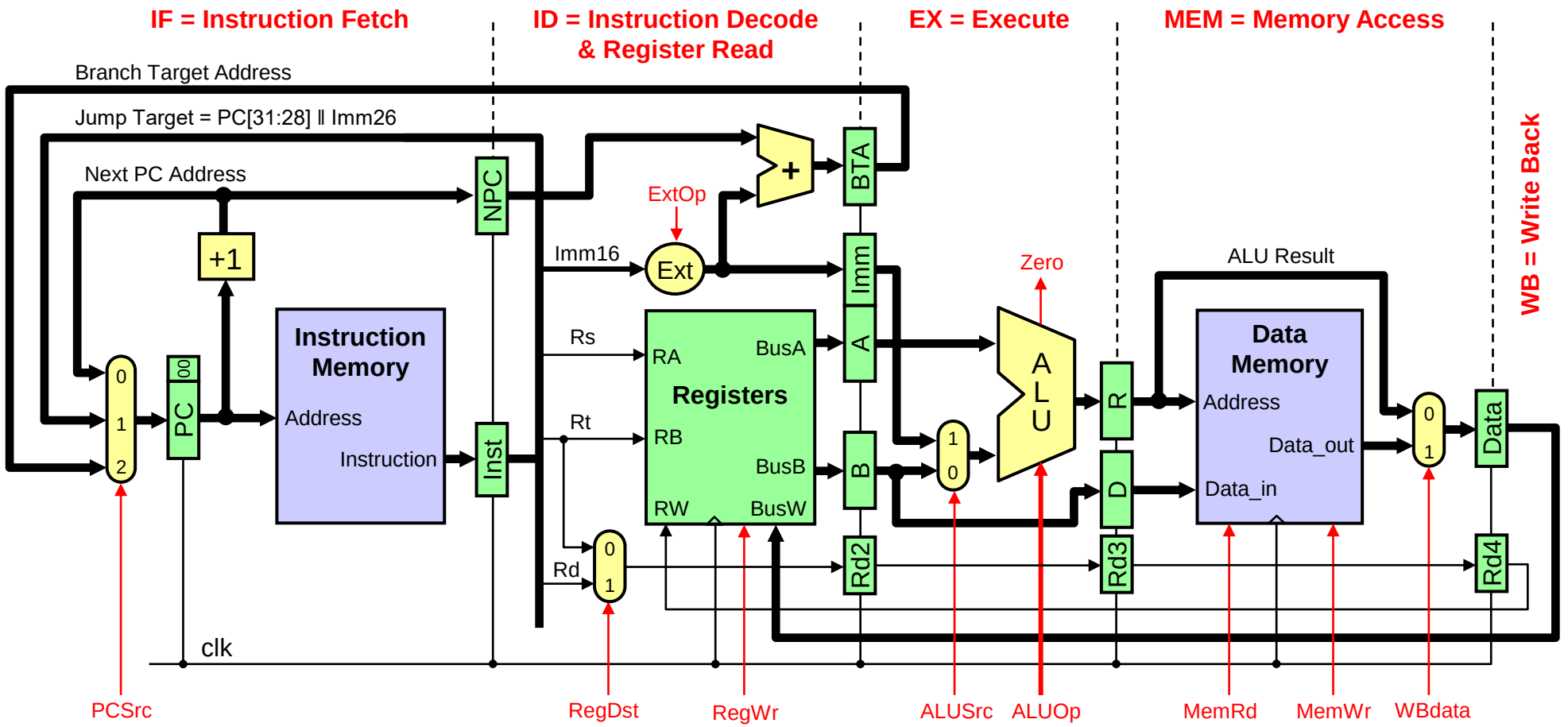
Problem with Register Destination

- 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

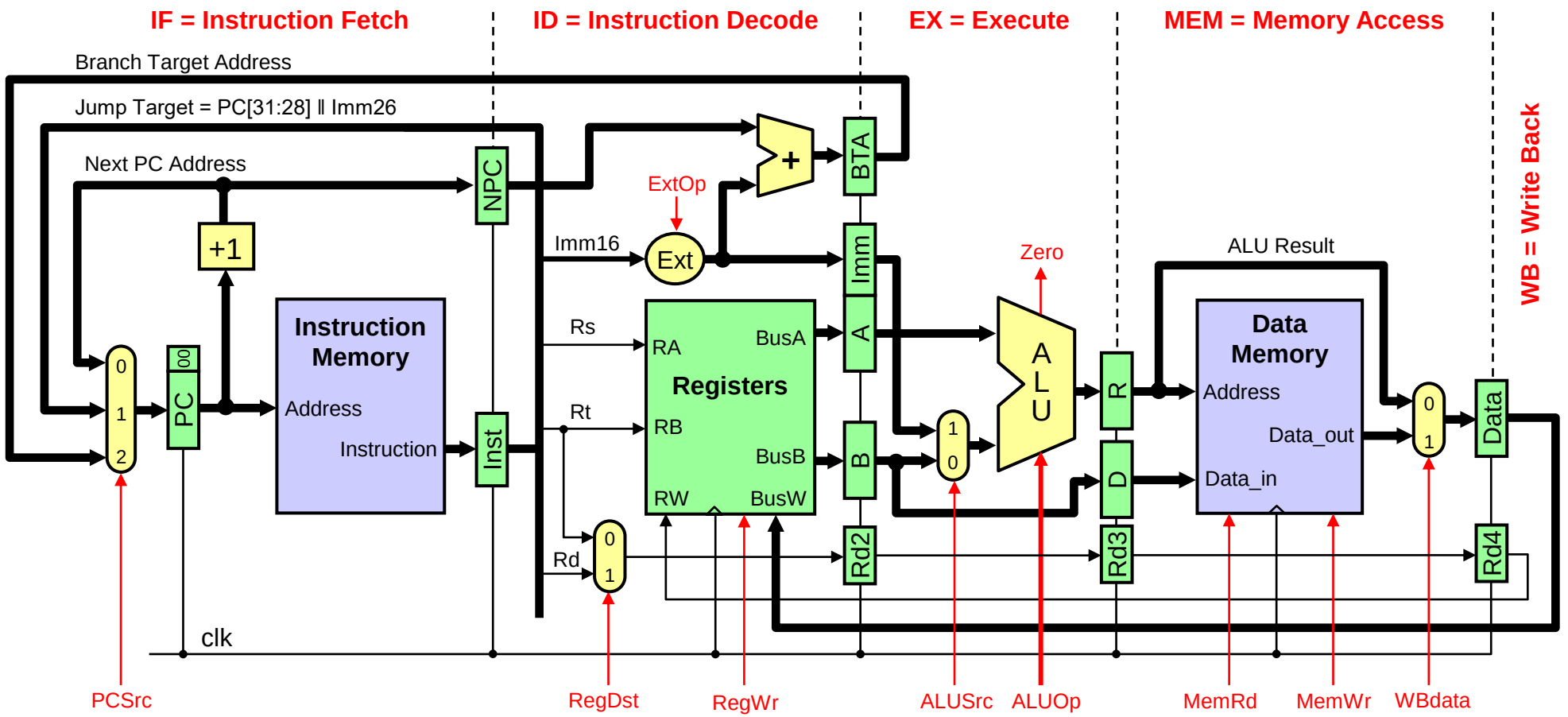
- Destination Register should be **pipelined** from ID to WB
  - The WB stage writes back data knowing the destination register



Pipelined control

- Data is travelling along the pipeline stages
- All data belonging to one instruction must be kept within the stage
- Information transfer only through the pipeline registers
- Control information must travel with the instruction
- Instruction fetch
  - Identical for all instructions
- Instruction decode / register file read
  - Identical for all instructions
- Execution / Address calculation
  - RegDest, ALUOp, ALUSrc
- Memory access
  - Branch, MemRead, MemWrite
- Write back
  - MemToReg, RegWrite

Control Signals



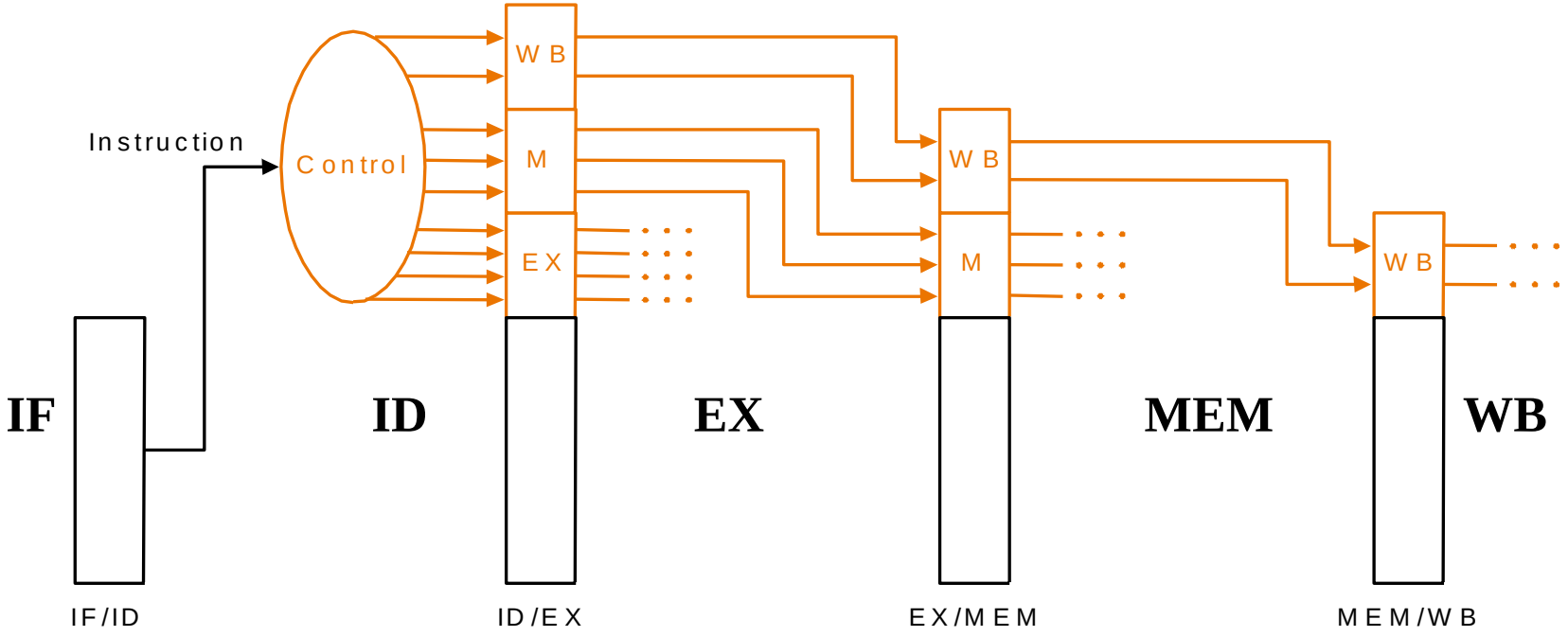
Same control signals used in the single-cycle datapath

Pipelined Control – Cont'd

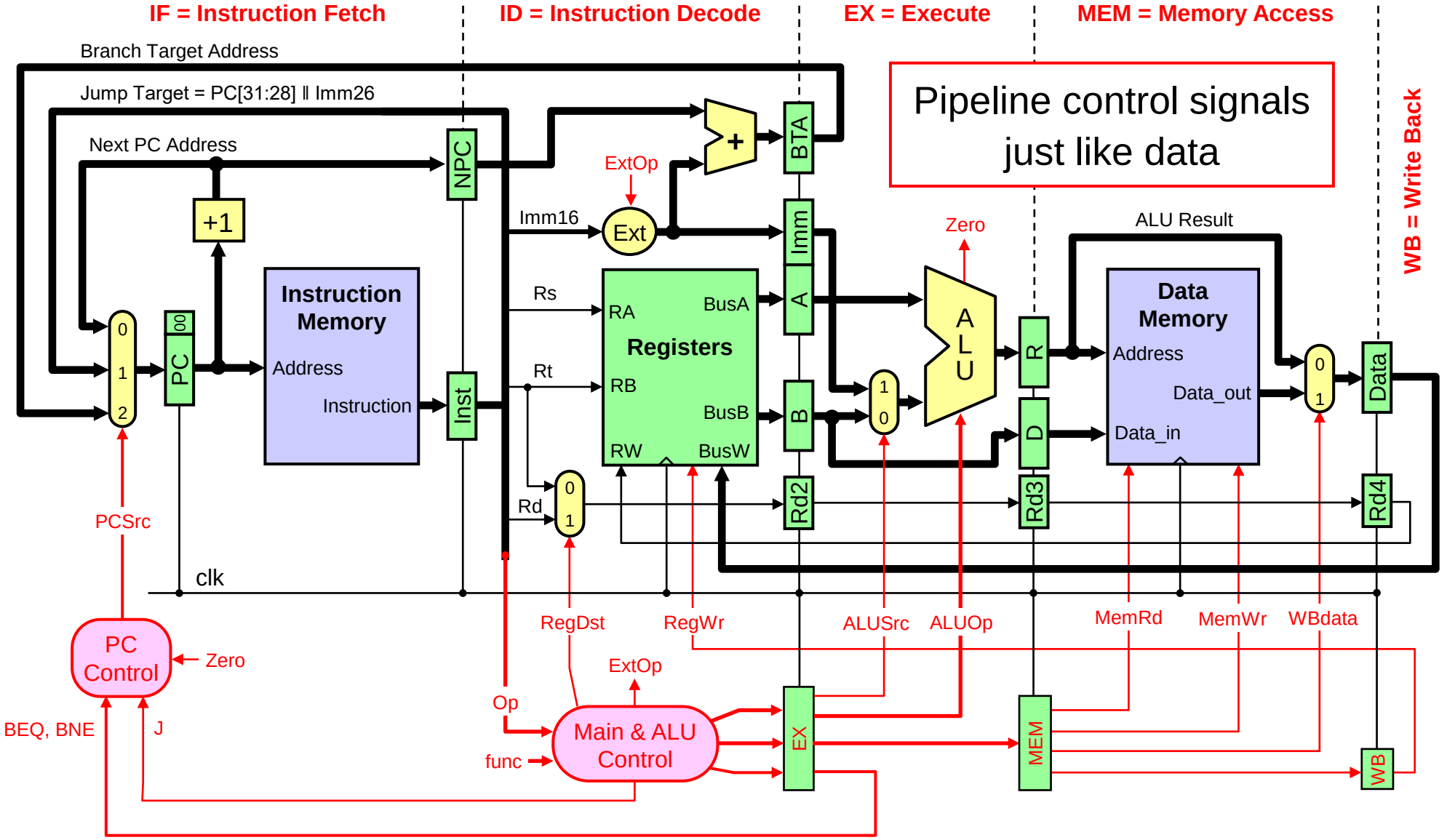
- ID stage generates all the control signals
- Pipeline the control signals as the instruction moves
  - Extend the pipeline registers to include the control signals
- Each stage uses some of the control signals
  - Instruction Decode and Register Read
    - Control signals are generated
    - RegDst 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

Pipeline Control

- Pass needed control signals along from one stage to the next as the instruction travels through the pipeline just like the needed data



Pipelined Data Path with Control



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 . . .

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

How do we deal with hazards?

- Often, pipeline must be *stalled*
- Stalling pipeline usually lets some instruction(s) in pipeline proceed, another/others wait for data, resource, etc.
- A note on terminology:
  - If we say an instruction was "issued *later* than instruction *x*", we mean that it was issued after instruction x and is *not as far along in the pipeline*
  - If we say an instruction was "issued *earlier* than instruction *x"*, we mean that it was issued before instruction *x* and is *further along in the pipeline*

Stalls and performance

- Stalls impede progress of a pipeline and result in deviation from 1 instruction executing/clock cycle
- Pipelining can be viewed to:
  - Decrease CPI or clock cycle time for instruction
  - Let's see what affect stalls have on CPI…

- CPI pipelined = Ideal CPI + Pipeline stall cycles per instruction

  = 1 + Pipeline stall cycles per instruction

- Ignoring overhead and assuming stages are balanced:

$$Speedup = \frac{CPI\ unpipelined}{1 + pipeline\ stall\ cycles\ per\ instruction}$$

Even more pipeline performance issues!

- This results in:

$$Clock\ cycle\ pipelined = \frac{Clock\ cycle\ unpipelined}{Pipeline\ depth}$$

$$Pipeline\ depth = \frac{Clock\ cycle\ unpipelined}{Clock\ cycle\ pipelined}$$

- Which leads to:

$$Speedup\ from\ pipelining = \frac{1}{1 + Pipeline\ stall\ cycles\ per\ instruction} \times \frac{Clock\ cycle\ unpipelined}{Clock\ cycle\ pipelined}$$

$$= \frac{1}{1 + Pipeline\ stall\ cycles\ per\ instruction} \times Pipeline\ depth$$

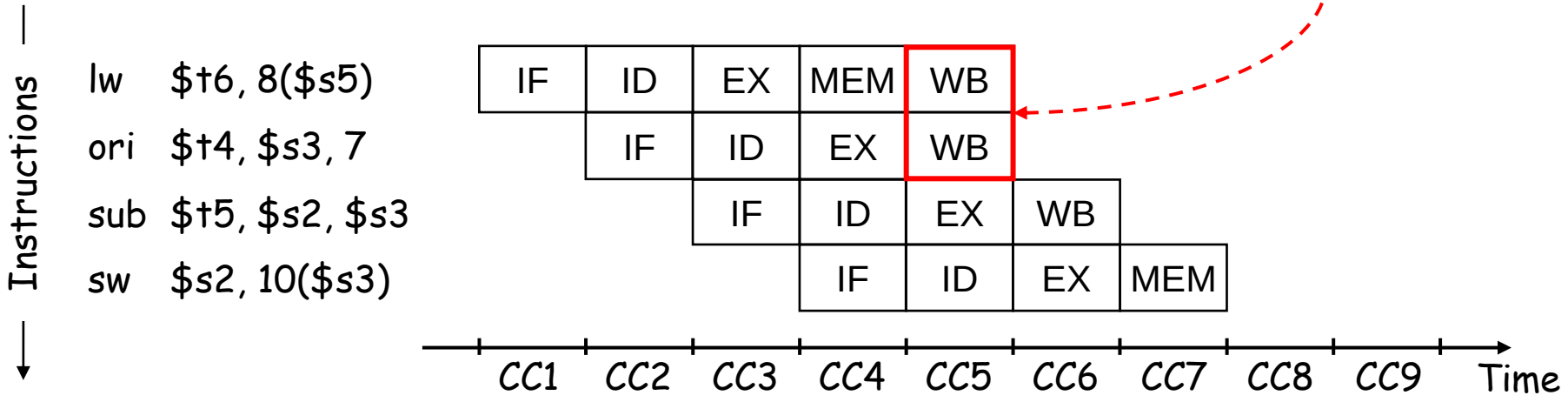
- If no stalls, speedup equal to # of pipeline stages in ideal case

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

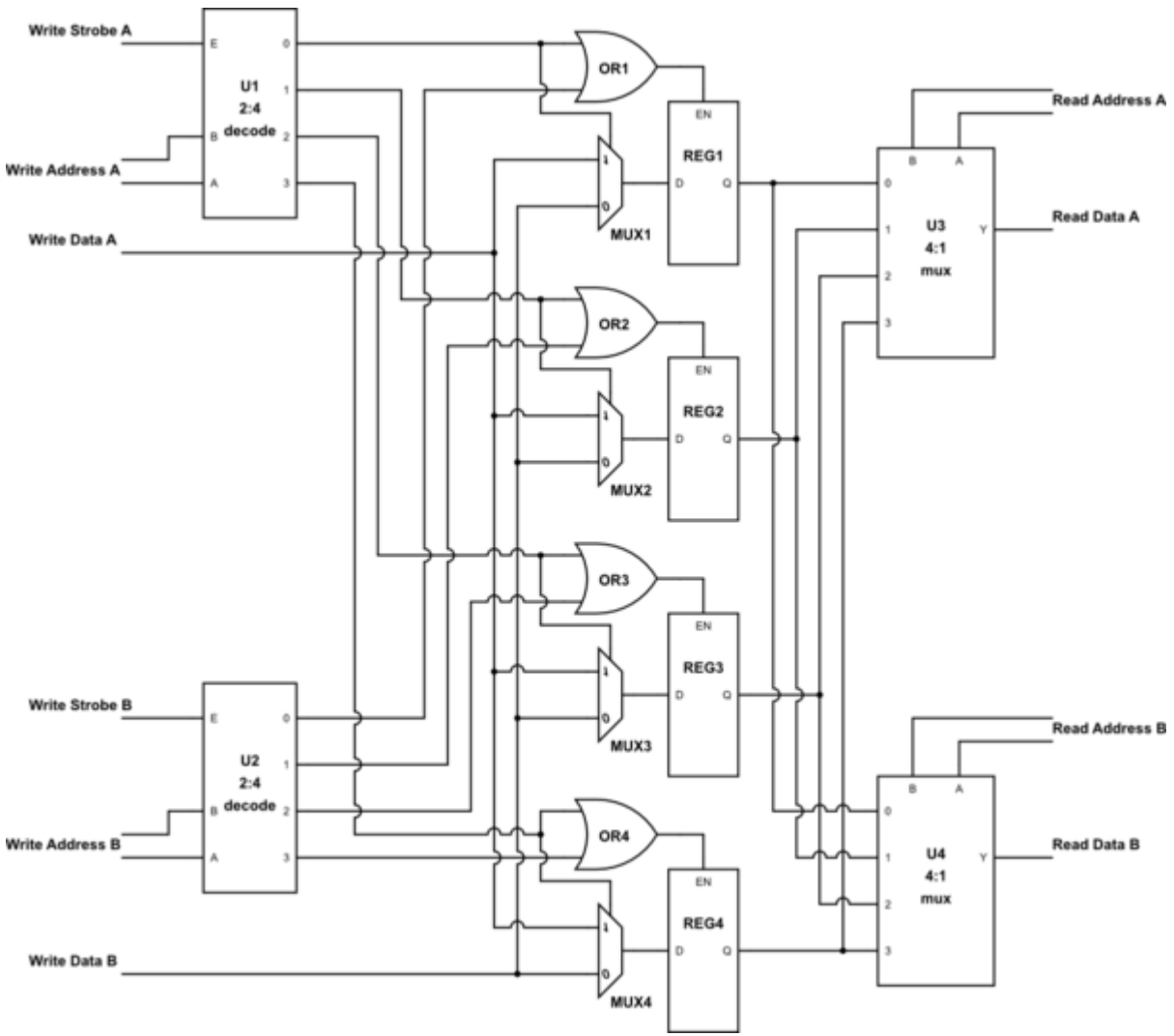
Structural Hazard

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



Handling Write Access To Register Bank

- 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



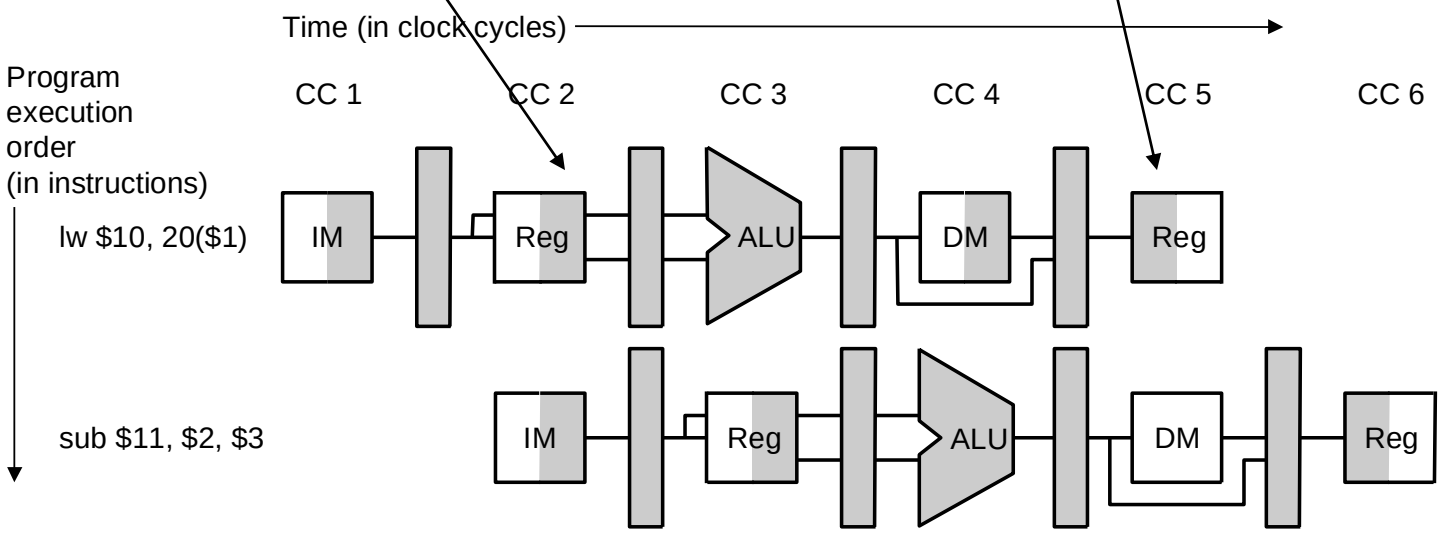
Two Port Write and Two Port Read Register File with 4 Registers

Handling Read/Write Access To Register Bank

- Two instructions need to access the register bank in the same cycle:
  - One instruction to read operands in its instruction decode (ID) cycle.
  - The other instruction to write to a destination register in its Write Back (WB) cycle.
- This represents a potential hardware conflict over access to the register bank.
- Solution: Coordinate register reads and write in the same cycle as follows:

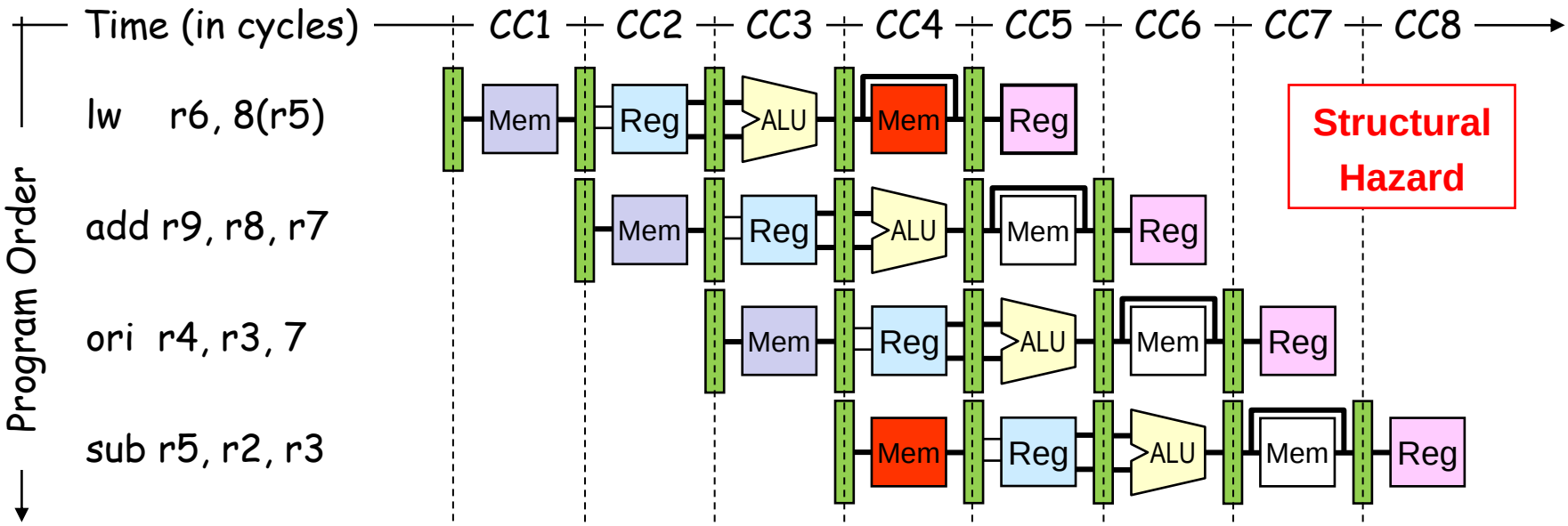
• Operand register reads in Instruction Decode ID cycle occur in the second half of the cycle (indicated here by the dark shading of the second half of the cycle)

• Register write in Write Back WB cycle occur in the first half of the cycle. (indicated here by the dark shading of the first half of the WB cycle)



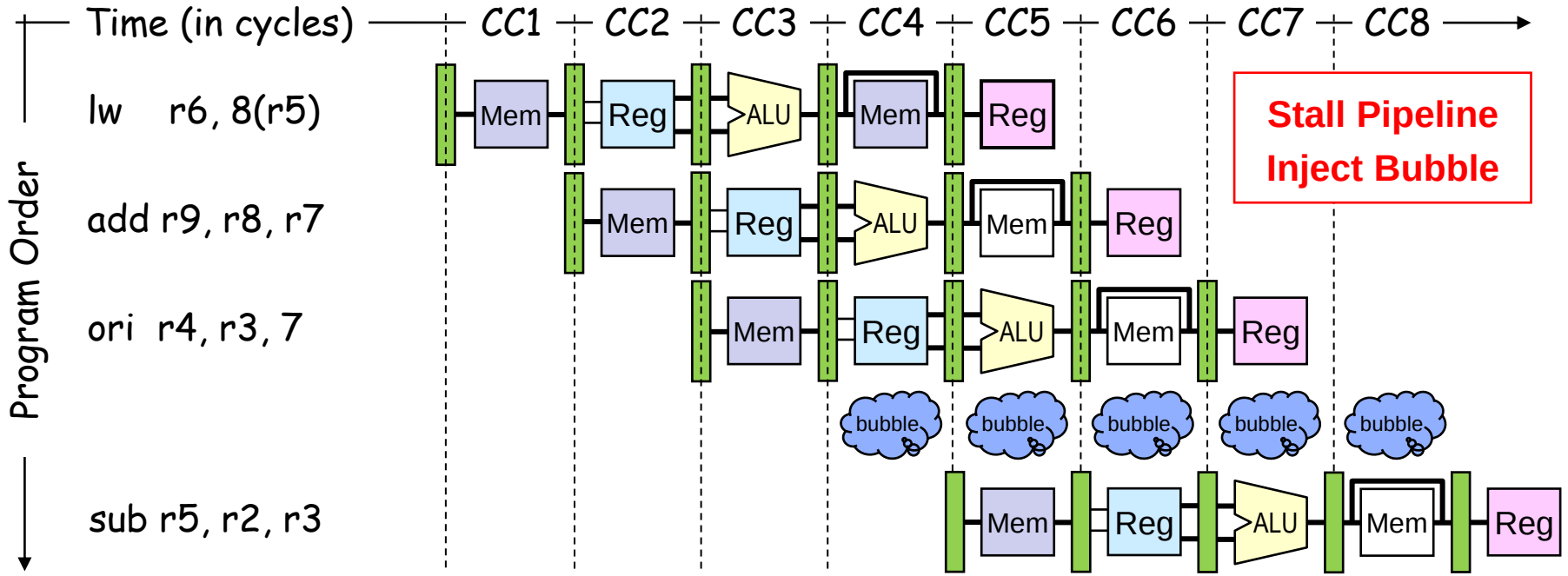
Example 3 of Structural Hazard

- One Cache Memory for both Instructions & Data
  - Instruction fetch requires cache access each clock cycle
  - Load/store also requires cache access to read/write data
  - Cannot fetch instruction and load data if one address port

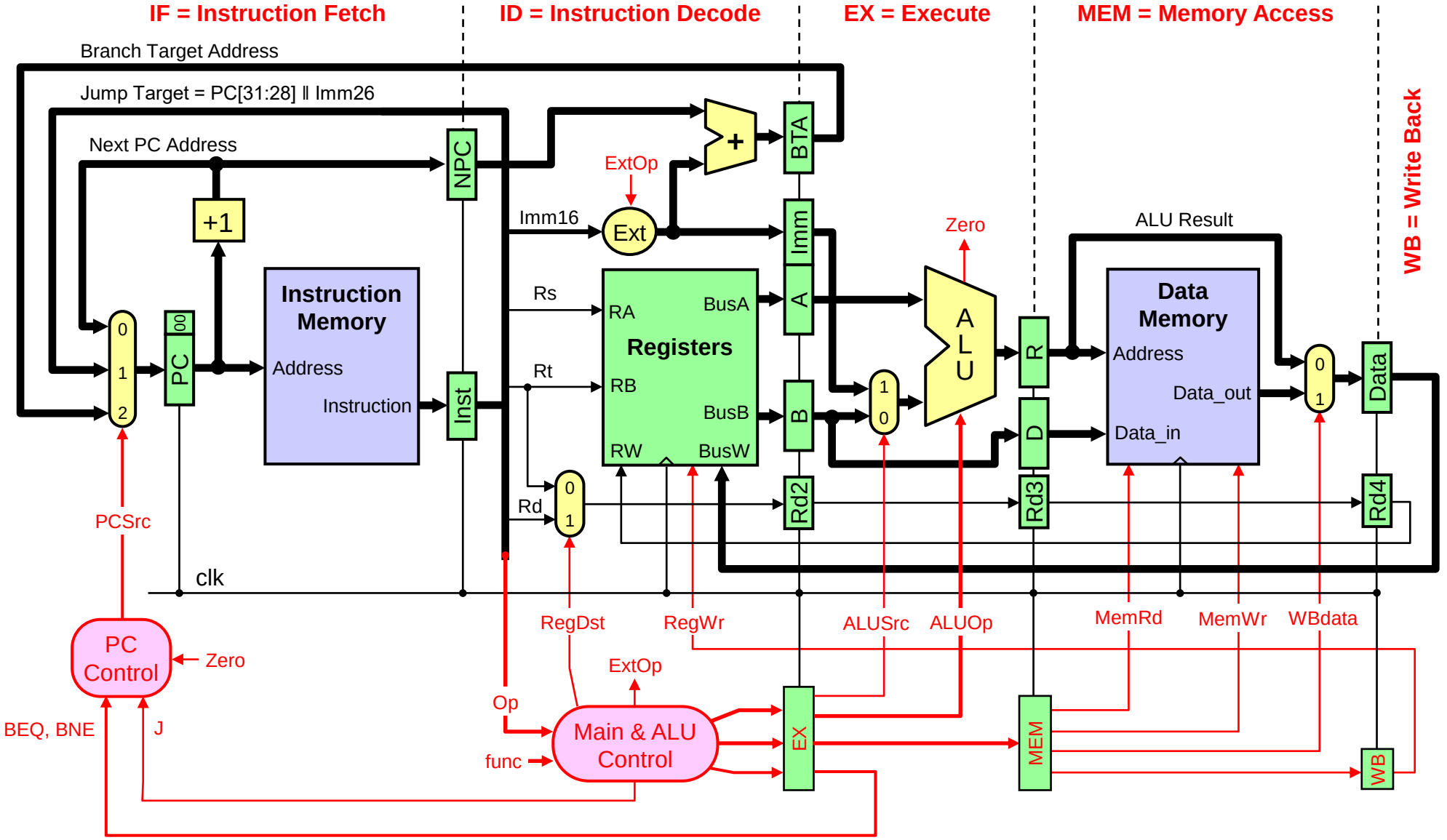


Stalling the Pipeline

- Delay Instruction Fetch ➔ **Stall** pipeline (inject **bubble**)
  - Reduces performance: Stall pipeline for each load and store!
- Better Solution: Use Separate Instruction & Data Caches
  - Addressed independently: No structural hazard and No stalls



Handling Other Cases

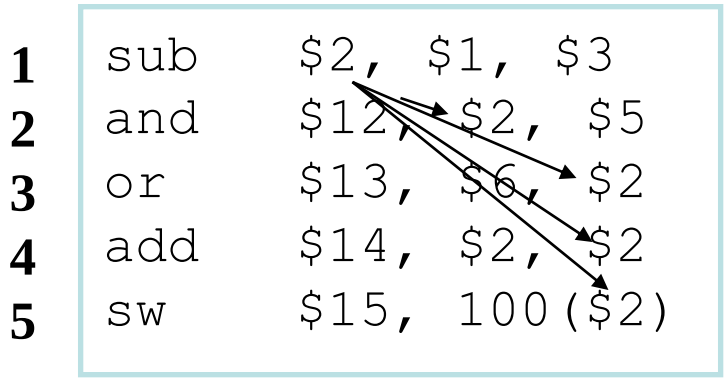


Resolving Structural Hazards Summary

- 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
  - Adding two memories caches
  - Adding adders

Data Hazards

- Data hazards occur when the pipeline changes the order of read/write accesses to instruction operands in such a way that the resulting access order differs from the original sequential instruction operand access order of the unpipelined CPU resulting in incorrect execution.
- Data hazards may require one or more instructions to be stalled in the pipeline to ensure correct execution.
- Example:

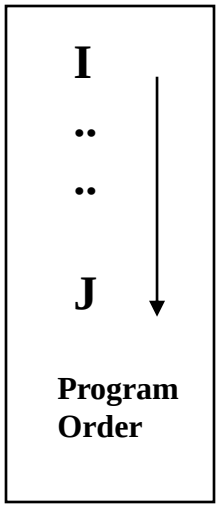


- ✓ **Arrows represent data dependencies between instructions**
- ✓ **Instructions that have no dependencies among them are said to be parallel or independent**
- ✓ **A high degree of Instruction-Level Parallelism (ILP) is present in a given code sequence if it has a large number of parallel instructions**

- ✧ All the instructions after the sub instruction use its result data in register $2
- ✧ As part of pipelining, these instruction are started before sub is completed:
  - Due to this data hazard instructions need to be stalled for correct execution.

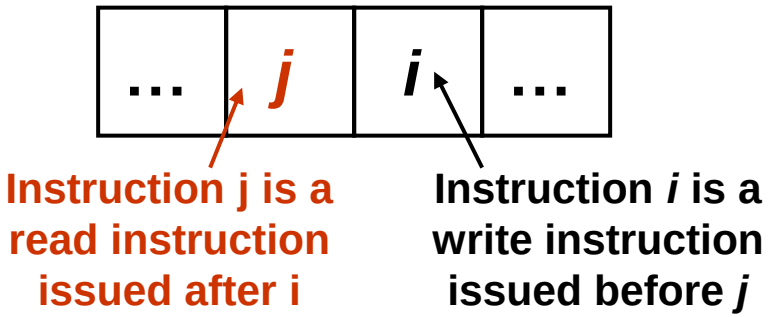
Data Hazard Classification

- Given two instructions *I*, *J*, with *I* occurring before *J* in an instruction stream:
  - RAW (read after write): *A true data dependence*
    J tried to read a source before *I* writes to it, so *J* incorrectly gets the old value.
  - WAW (write after write): *A name dependence*
    J tries to write an operand before it is written by *I*
    The writes end up being performed in the wrong order.
  - WAR (write after read): *A name dependence*
    J tries to write to a destination before it is read by *I*, so *I* incorrectly gets the new value.
  - RAR (read after read): Not a hazard.



Read after write (RAW) hazards

- With RAW hazard, instruction *j* tries to read a source operand before instruction *i* writes it.
- Thus, *j* would incorrectly receive an old or incorrect value
- Graphically/Example:



- Can use stalling or forwarding to resolve this hazard

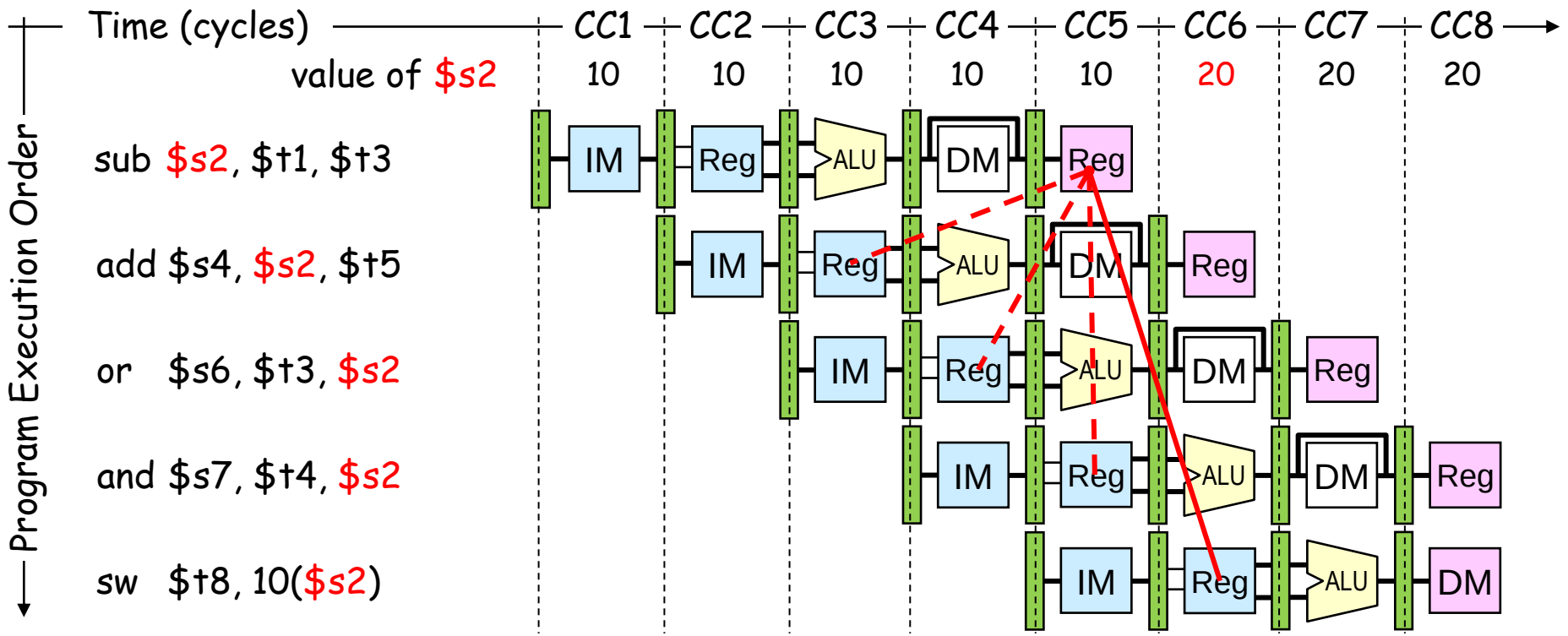
i: ADD R1, R2, R3
j: SUB R4, R1, R6

i: Lw R1, Off(R3)
j: SUB R4, R1, R6

i: ADD R1, R2, R3
j: Sw R1, Off(R6)

i: Lw R1, Off(R3)
j: Sw R1, Off(R6)

Example of a RAW Data Hazard



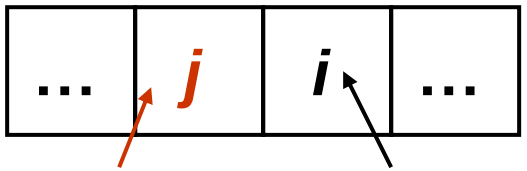
- Result of sub is needed by add, or, and, & sw instructions
- Instructions add & or will read old value of $s2 from reg file
- During CC5, $s2 is written at end of cycle, old value is read

Write after write (WAW) hazards

- With WAW hazard, instruction *j* tries to write an operand before instruction *i* writes it.

- The writes are performed in wrong order leaving the value written by earlier instruction

- Graphically/Example:



Instruction j is a write instruction issued after i

Instruction *i* is a write instruction issued before *j*

i: DIV F1, F2, F3
j: SUB F1, F4, F6

i: Sw F1, off(F2)
j: Sw F1, off(F2)

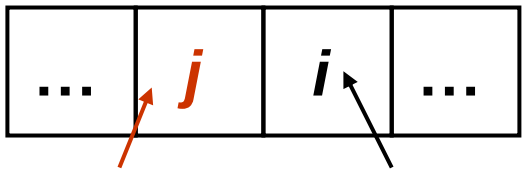
i: Lw F1, off(F2)
j: SUB F1, F4, F6

Write after read (WAR) hazards

- With WAR hazard, instruction *j* tries to write an operand before instruction *i* reads it.

- Instruction *i* would incorrectly receive newer value of its operand;
  - Instead of getting old value, it could receive some newer, undesired value:

- Graphically/Example:



Instruction j is a write instruction issued after i

Instruction *i* is a read instruction issued before *j*

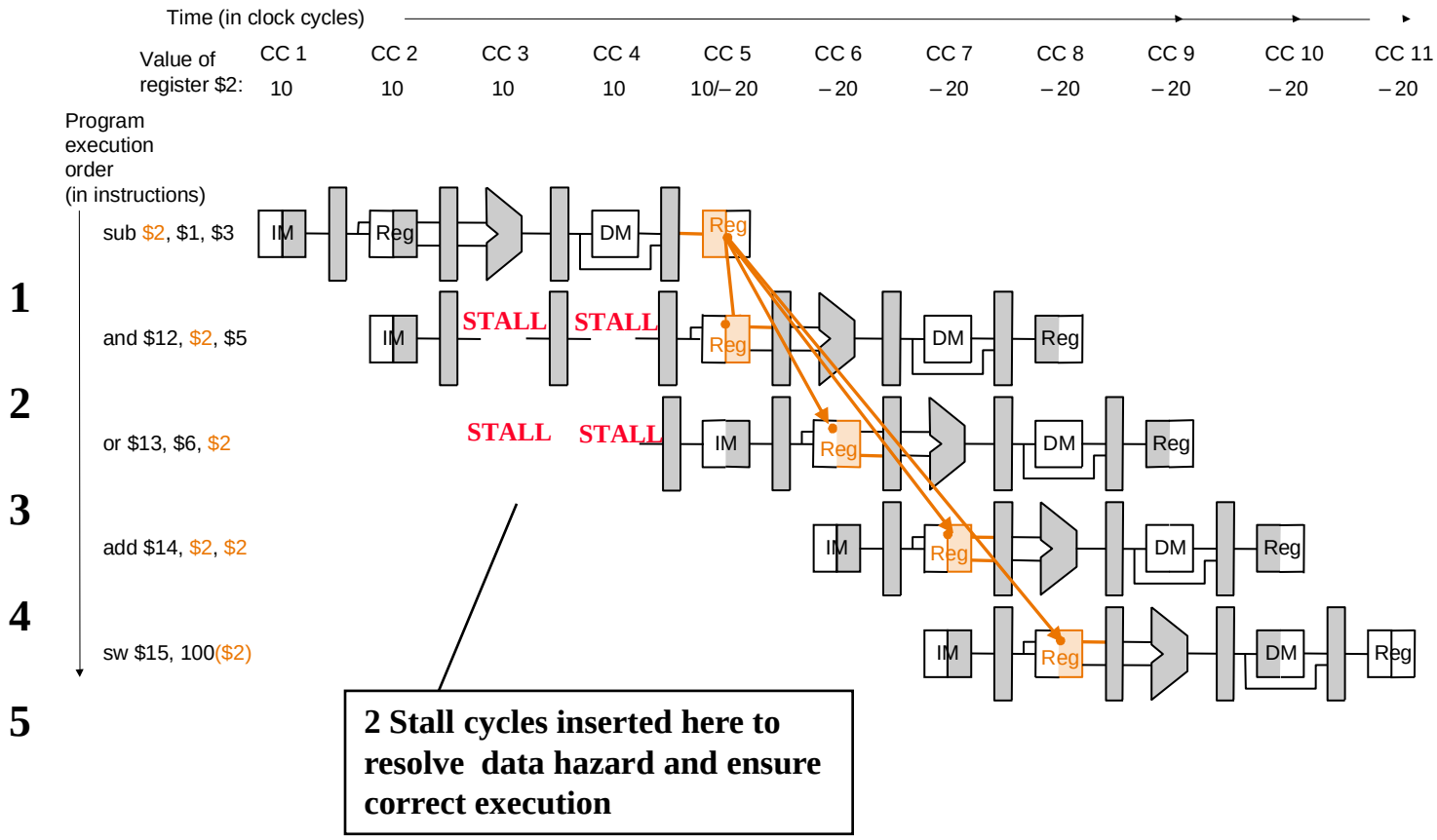
i: DIV F7, F1, F3
j: SUB F1, F4, F6

i: Sw F1, off(F3)
j: SUB F1, F4, F6

i: Lw F1, off(F3)
j: Sw F2, off(F3)

Data Hazard Resolution: Stall Cycles

- **Stall the pipeline by a number of cycles.**
- **The control unit must detect the need to insert stall cycles.**
- **<u>In this case two stall cycles are needed.</u>**



Data Hazard Resolution/Stall Reduction: Data Forwarding

- Observation: why not use temporary results produced by memory/ALU and not wait for them to be written back in the register bank.
- Data Forwarding is a hardware-based technique (also called register bypassing or register short-circuiting) used to eliminate or minimize data hazard stalls that makes use of this observation.
- Using forwarding hardware, the result of an instruction is copied directly (i.e. forwarded) from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)

Forwarding In MIPS Pipeline

- The ALU result from the EX/MEM register may be forwarded or fed back to the ALU input latches as needed instead of the register operand value read in the ID stage.
- Similarly, the Data Memory Unit result from the MEM/WB register may be fed back to the ALU input latches as needed .
- If the forwarding hardware detects that a previous ALU operation is to write the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.
- Hazard types

  1a EX/MEM.RegisterRd = ID/EX.RegisterRs

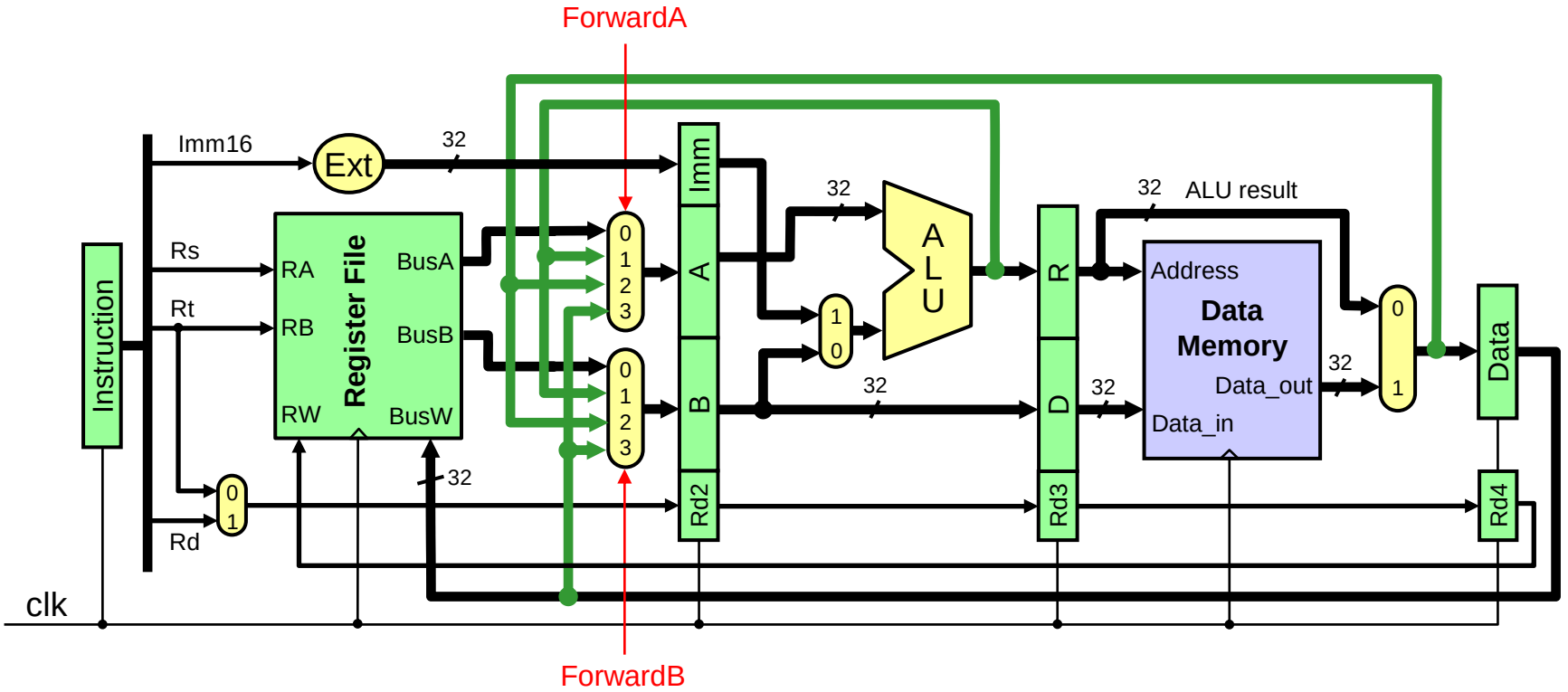
  1b EX/MEM.RegisterRd = ID/EX.RegisterRt

  2a MEM/WB.RegisterRd = ID/EX.RegisterRs

  2b MEM/WB.RegisterRd = ID/EX.RegisterRt

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 2nd previous instruction to A (from MEM stage)
ForwardA = 3	Forward result of 3rd 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 2nd previous instruction to B (from MEM stage)
ForwardB = 3	Forward result of 3rd previous instruction to B (from WB stage)

Forwarding Example

Instruction sequence:

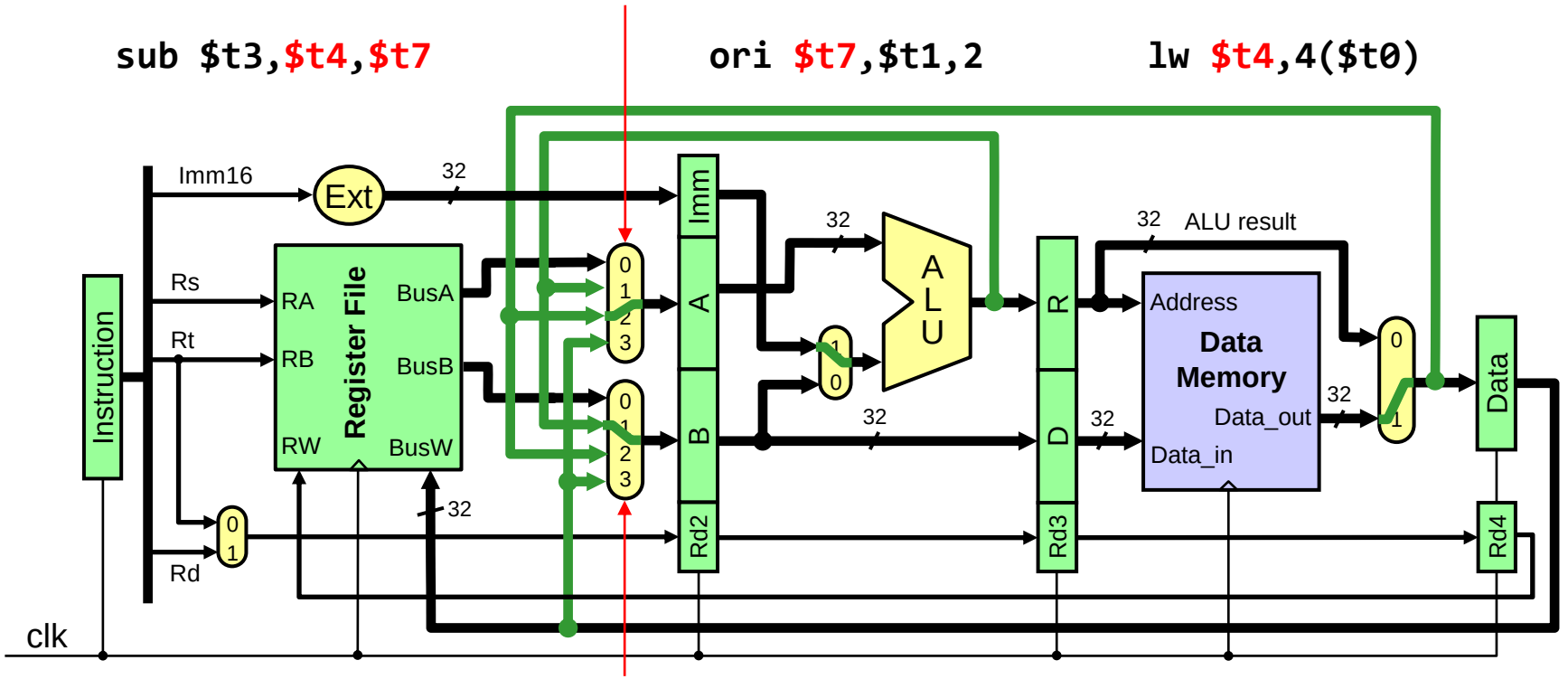
```
lw   $t4, 4($t0)
ori  $t7, $t1, 2
sub  $t3, $t4, $t7
```

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)



ForwardB = 1 (from ALU stage)

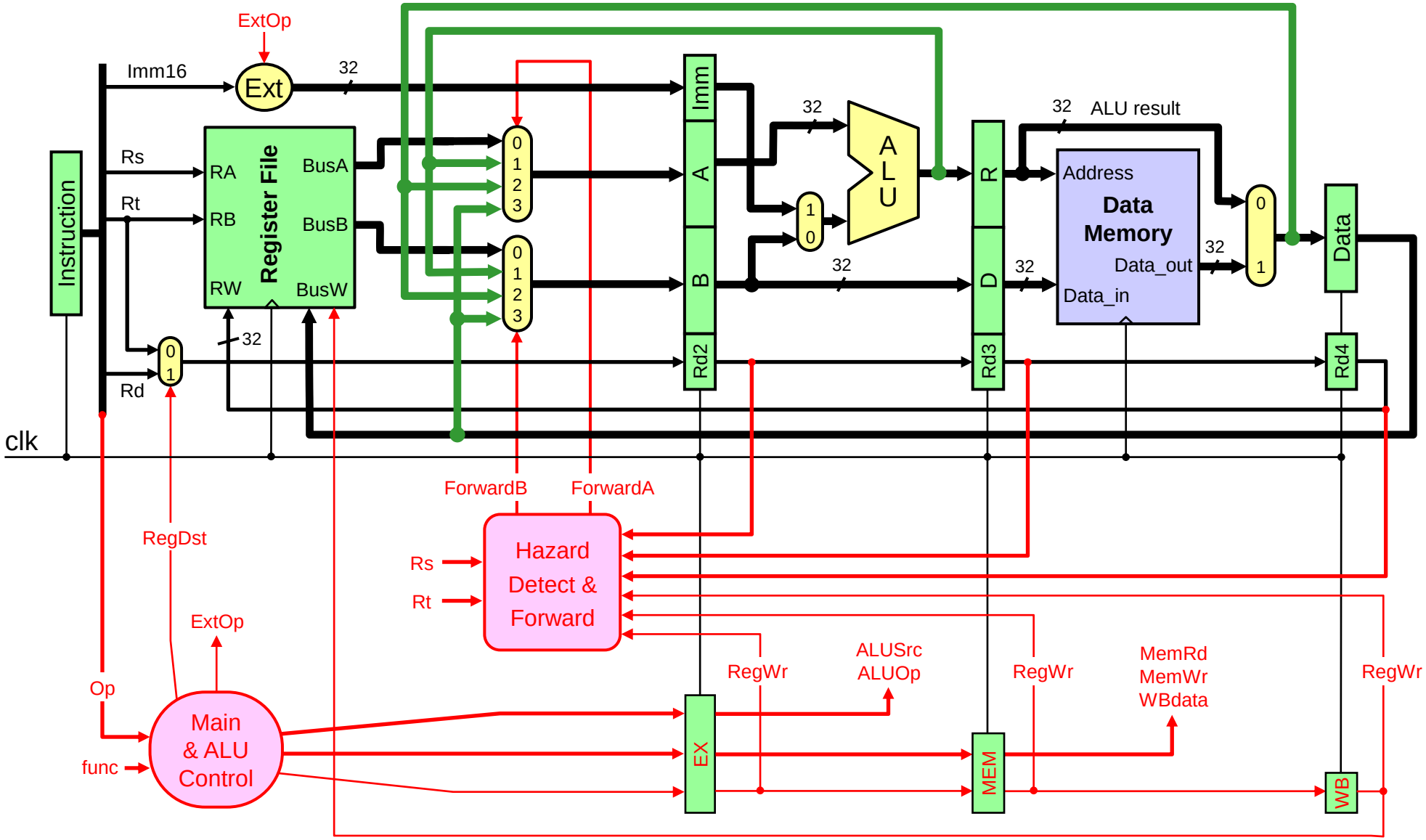
RAW Hazard Detection

- 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

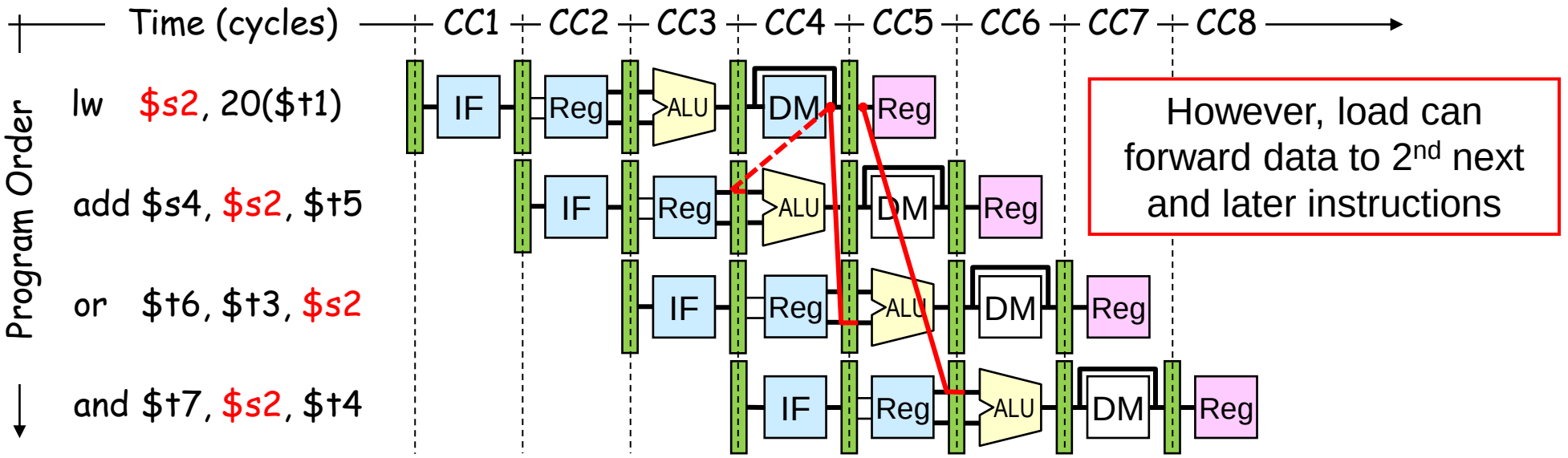


Next . . .

- Pipelined Datapath and Control
- Pipeline Hazards
- Data Hazards and Forwarding
- **Load Delay, Hazard Detection, and Stall**
- 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 read data until end of CC4
  - Cannot forward data to ADD at end of CC3 - NOT possible



Detecting RAW Hazard after Load

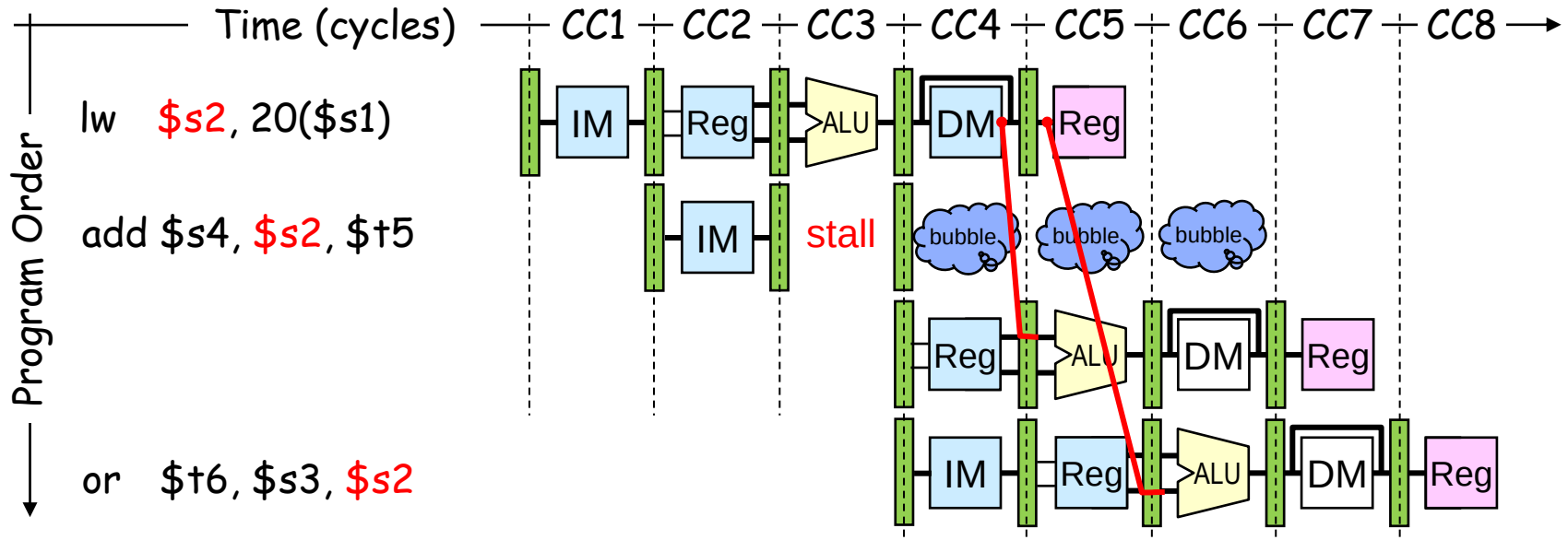
- Detecting a RAW hazard after a Load instruction:
  - The load instruction will be in the EX stage
  - Instruction that depends on the load data is in the decode stage
- Condition for stalling the pipeline

```
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

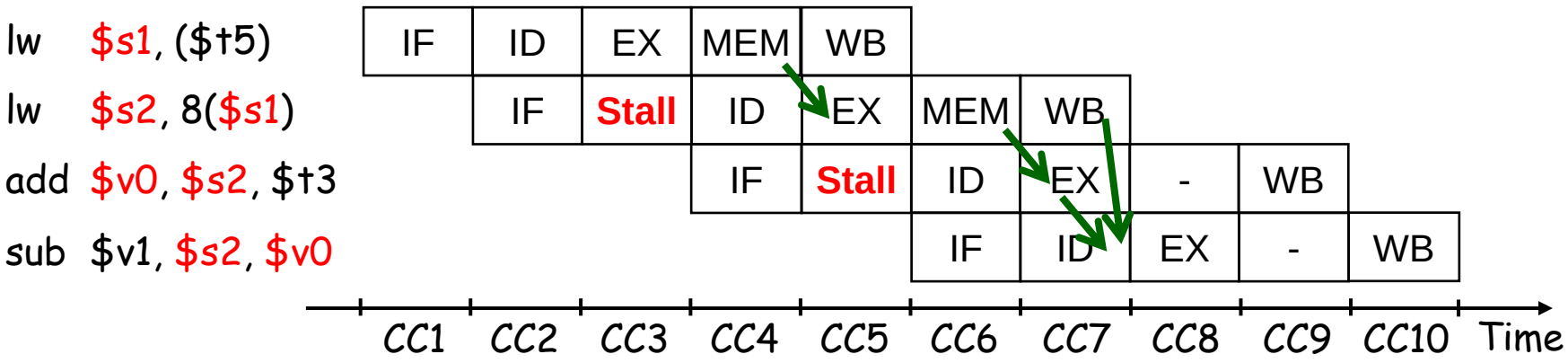
- ADD instruction depends on LW ➔ stall at CC3
  - Allow Load instruction in ALU stage to proceed
  - Freeze PC and Instruction registers (NO instruction is fetched)
  - Introduce a bubble into the ALU stage (bubble is a NO-OP)
- Load can forward data to next instruction after delaying it



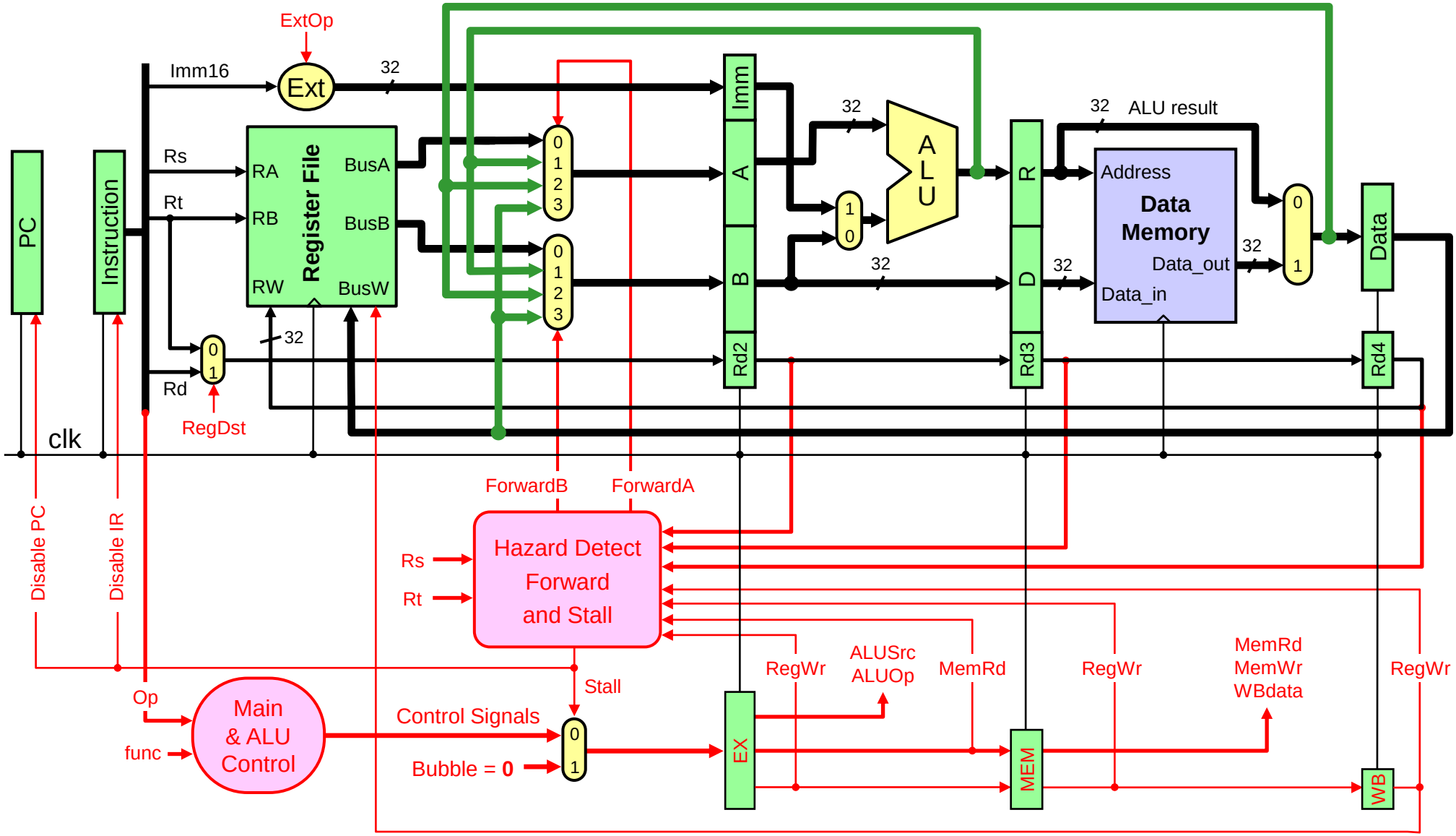
Showing Stall Cycles

- 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



Static Compiler Instruction Scheduling (Re-Ordering) for Data Hazard Stall Reduction

- Many types of stalls resulting from data hazards are very frequent. For example:

$$A = B + C$$

produces a stall when loading the second data value (B).

- Rather than allow the pipeline to stall, the compiler could sometimes schedule the pipeline to avoid stalls.

- Compiler pipeline or instruction scheduling involves rearranging the code sequence (instruction reordering) to eliminate or reduce the number of stall cycles.

Static = At compilation time by the compiler
Dynamic = At run time by hardware in the CPU

Compiler Scheduling Example

- Reorder the instructions to avoid as many pipeline stalls as possible:

Stall →

```
lw    $15, 0($2)
lw    $16, 4($2)
add   $14, $5, $16
sw    $16, 4($2)
```

- The **data hazard occurs on register $16 between the second lw and the add instruction resulting in a stall cycle even with forwarding**
- **With forwarding we (or the compiler) need to find only one independent instruction to place between them, swapping the lw instructions works:**

```
lw    $16, 4($2)
lw    $15, 0($2)
add   $14, $5, $16
sw    $16, 4($2)
```

- **Without forwarding we need two independent instructions to place between them, so in addition a nop is added (or the hardware will insert a stall).**

Or stall cycle →

```
lw    $16, 4($2)
lw    $15, 0($2)
nop
add   $14, $5, $16
sw    $16, 4($2)
```

Compiler Scheduling Example

- 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($0)     # &D = 12($0)
```

- 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)
```

Next . . .

- Pipelined Datapath and Control
- Pipeline Hazards
- Data Hazards and Forwarding
- Load Delay, Hazard Detection, and Stall
- **Control Hazards**
- Delayed Branch and Dynamic Branch Prediction

Control Hazards

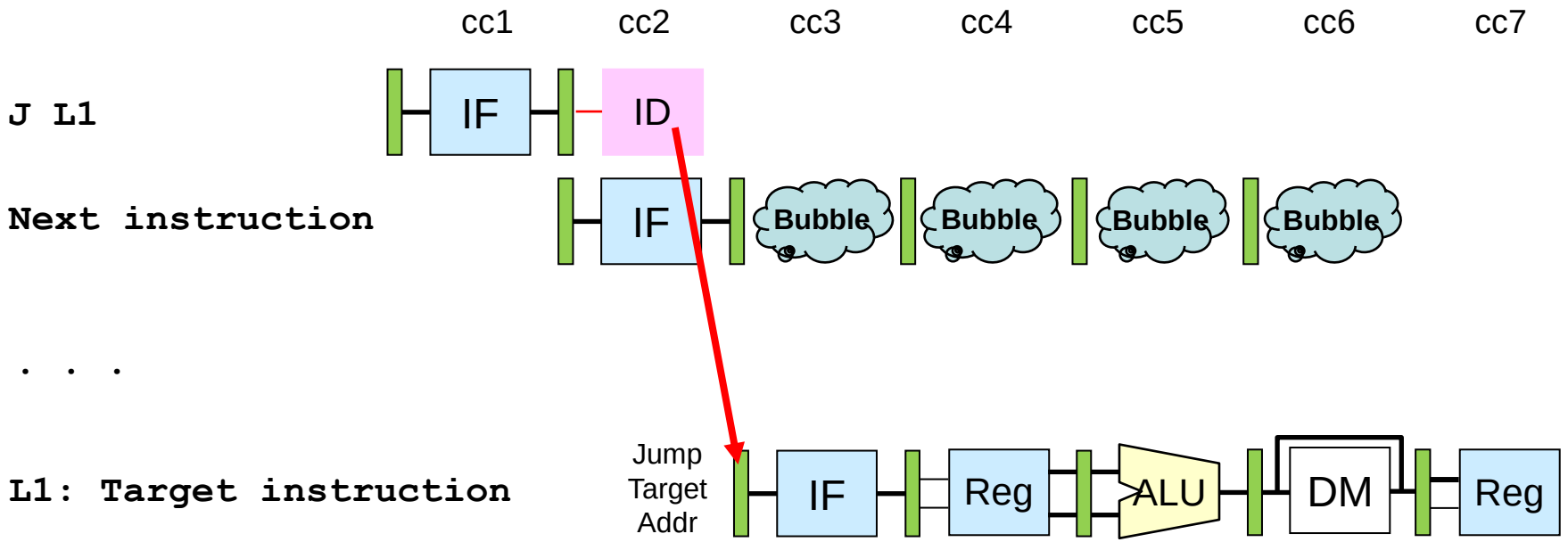
- Jump and Branch can cause great performance loss
- Jump instruction needs only the jump target address
- Branch instruction needs two things:
  - Branch Result — Taken or Not Taken
  - Branch Target Address
    - PC + 4 — If Branch is NOT taken
    - PC + 4 + 4 × immediate — If Branch is Taken
- Jump 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)

What is needed to Calculate next PC?

- For Unconditional Jumps
  - Opcode (J or JAL), PC and 26-bit address (immediate)
- For Jump Register
  - Opcode + function (JR or JALR) and Register[Rs] value
- For Conditional Branches
  - Opcode, branch outcome (taken or not), PC and 16-bit offset
- For Other Instructions
  - Opcode and PC value
- Opcode is decoded in ID stage ➔ Jump delay = 1 cycle
- Branch outcome is computed in EX stage
  - Branch delay = 2 clock cycles

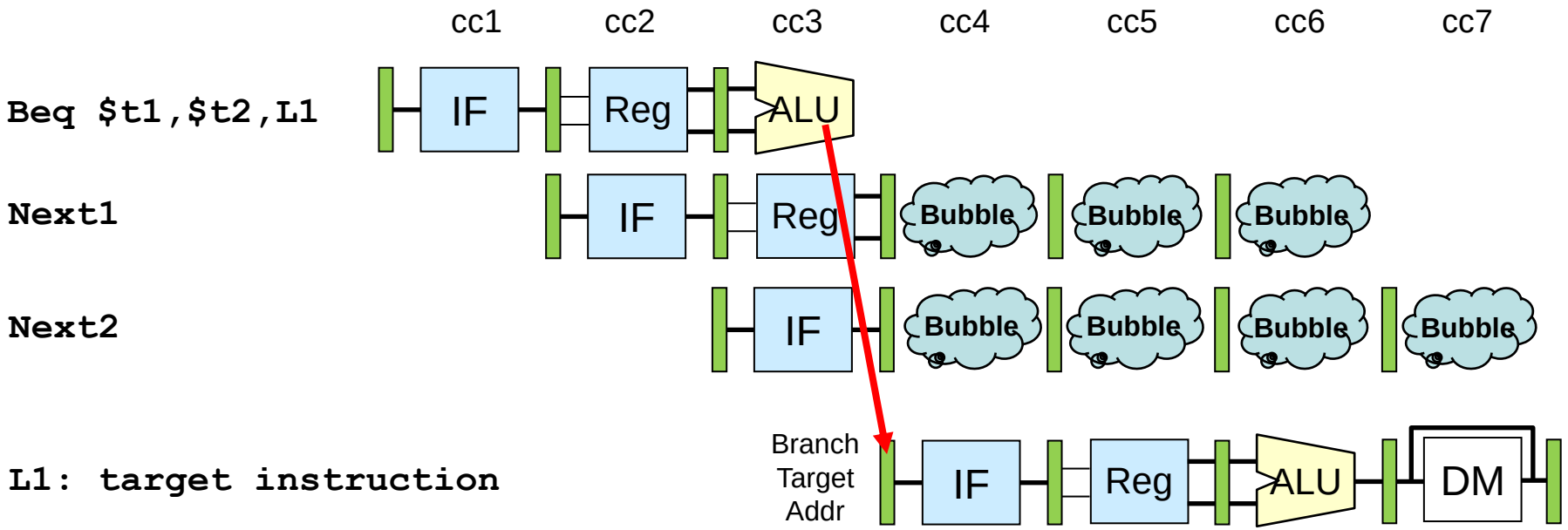
1-Cycle Jump Delay

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



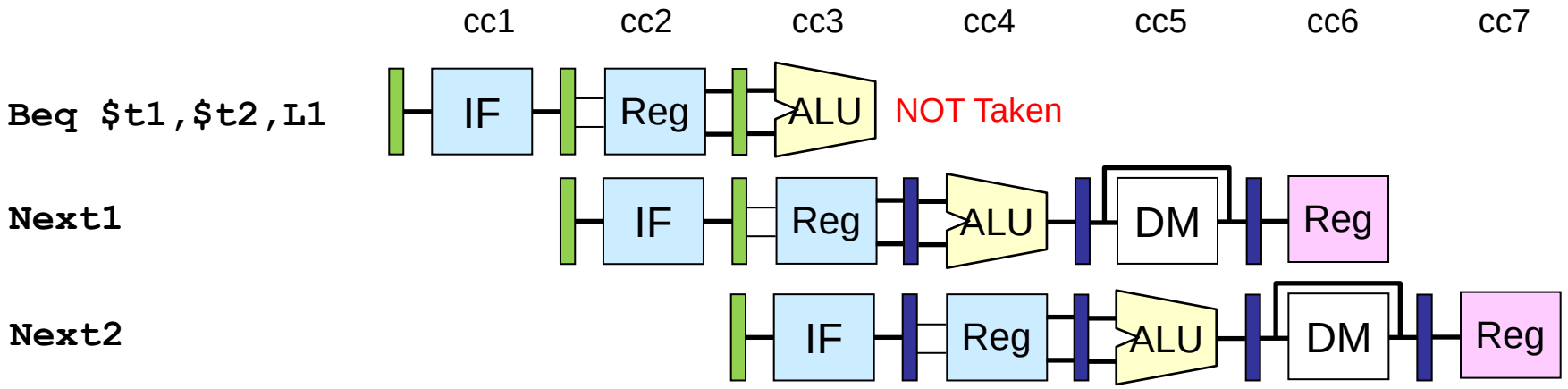
2-Cycle Branch Delay

- 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

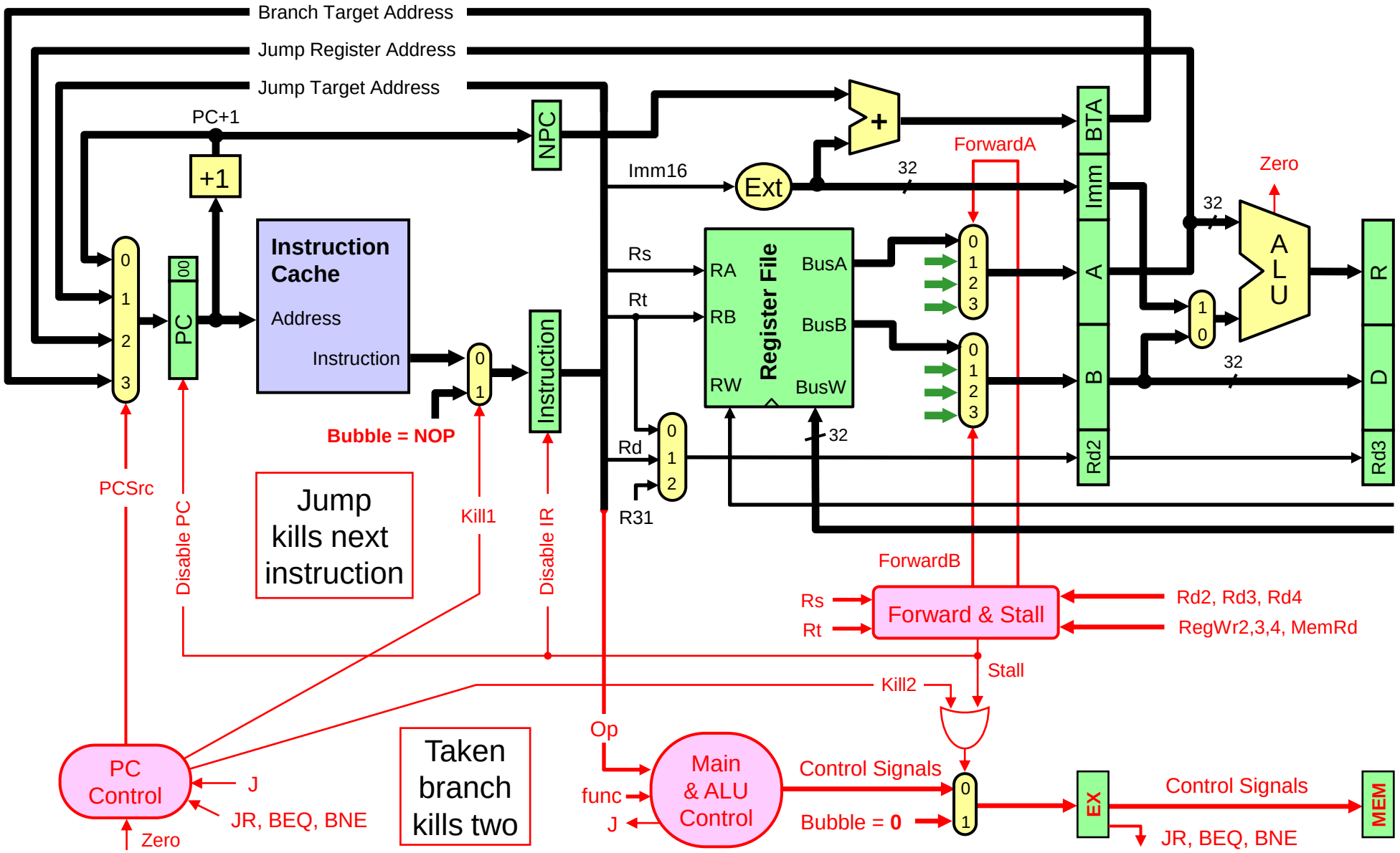


Predict Branch NOT Taken

- 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; }
```

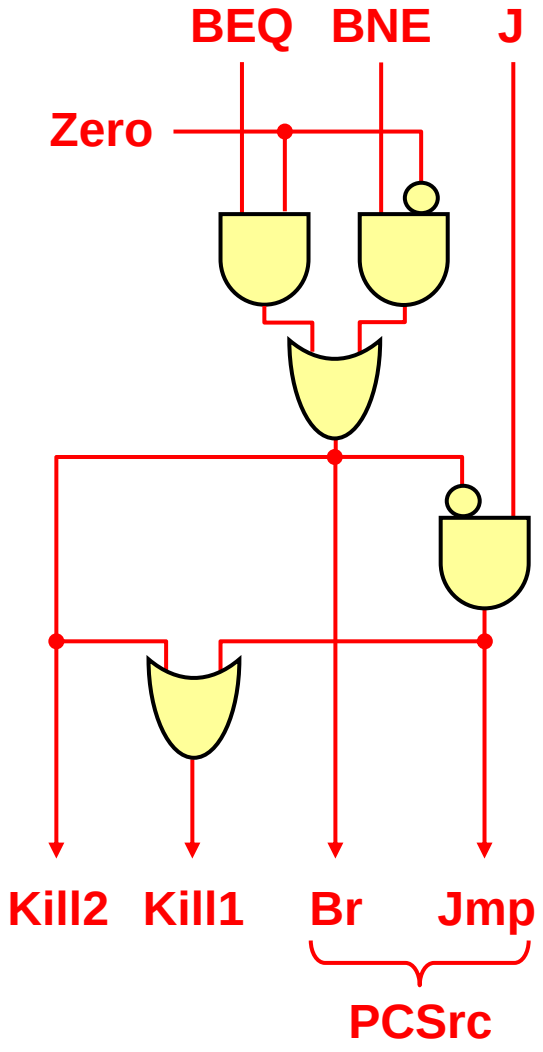
$Br = ((BEQ \cdot Zero) + (BNE \cdot \overline{Zero}))$

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

$Kill1 = J + Br$

$Kill2 = Br$

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



Hardware Reduction of Branch Stall Cycles

- Pipeline hardware measures to reduce branch stall cycles:

  1- Find out whether a branch is taken earlier in the pipeline.

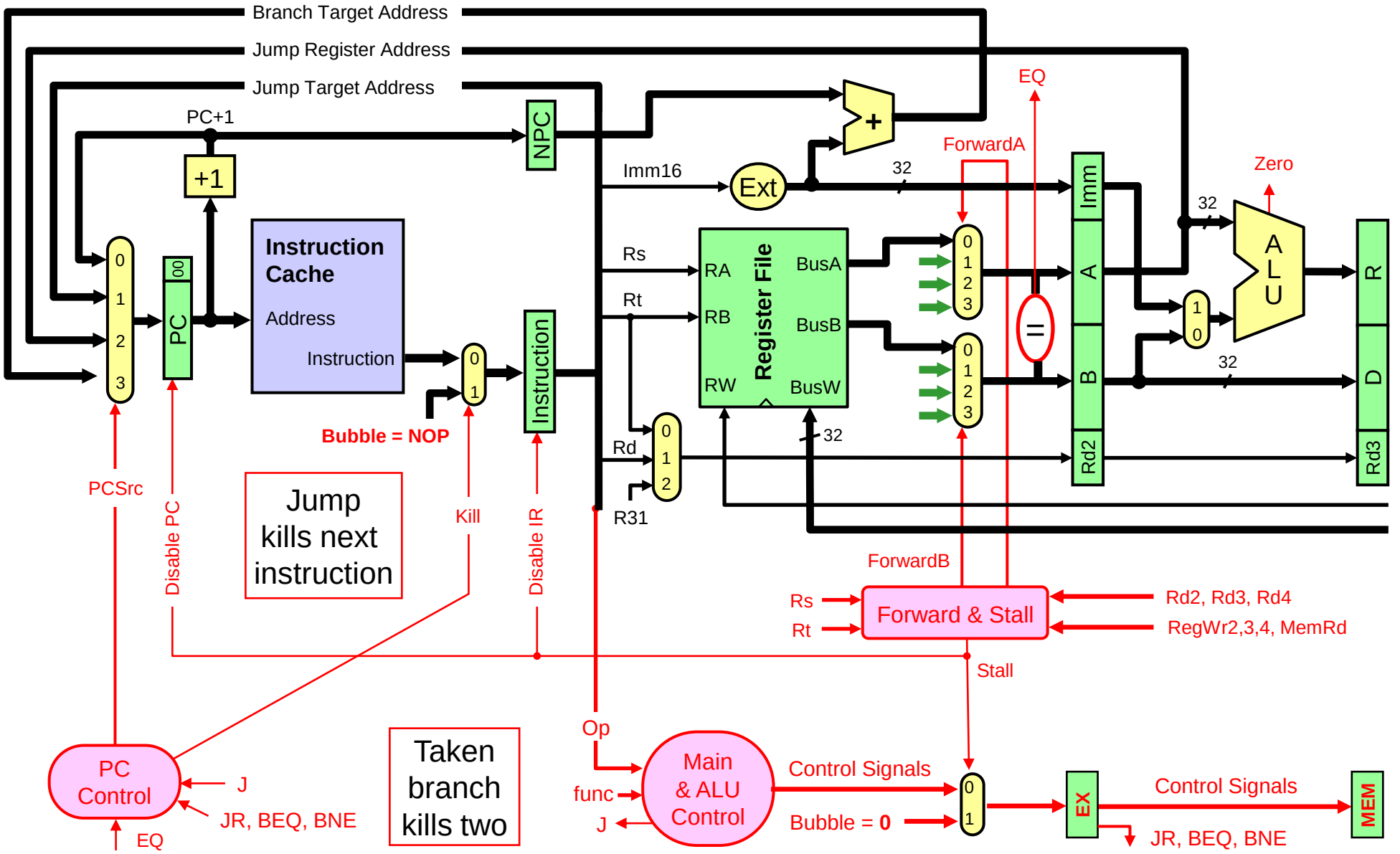
  2- Compute the taken PC earlier in the pipeline.

- In MIPS: **i.e Resolve the branch in an early stage in the pipeline**
  - In MIPS branch instructions BEQ, BNE, test a register for equality to zero.
  - This can be completed in the ID cycle by moving the zero test into that cycle.
  - Both PCs (taken and not taken) must be computed early.
  - Requires an additional adder because the current ALU is not useable until EX cycle.
  - This results in just a single cycle stall on branches.
    - Branch Penalty when taken = stage resolved - 1 = 2 - 1 = 1

Reducing Delay (Penalty) of Taken Branches

- So far: Next PC of a branch known or resolved in EX stage:
- Costs two lost cycles if the branch is taken.
- If next PC of a branch is known or resolved in ID stage, one cycle is saved.
- Branch address calculation can be moved to ID stage (stage 2) using a register comparator, costing only one cycle if branch is taken as shown below.
- Branch Penalty = stage 2 -1 = 1 cycle

Reducing Delay (Penalty) of Taken Branches



Jump and Branch Impact on CPI

- Base CPI = 1 without counting jump and branch stalls
- Unconditional Jump = 5%, Conditional branch = 20%

  and 90% of conditional branches are taken
- 1 stall cycle per jump, 2 stall cycles per taken branch
- What is the effect of jump and branch stalls on the CPI?

Solution:

- Jump adds 1 stall cycle for 5% of instructions = 1 × 0.05
- Branch adds 2 stall cycles for 20% × 90% of instructions

  = 2 × 0.2 × 0.9 = 0.36
- New CPI = 1 + 0.05 + 0.36 = 1.41

Next . . .

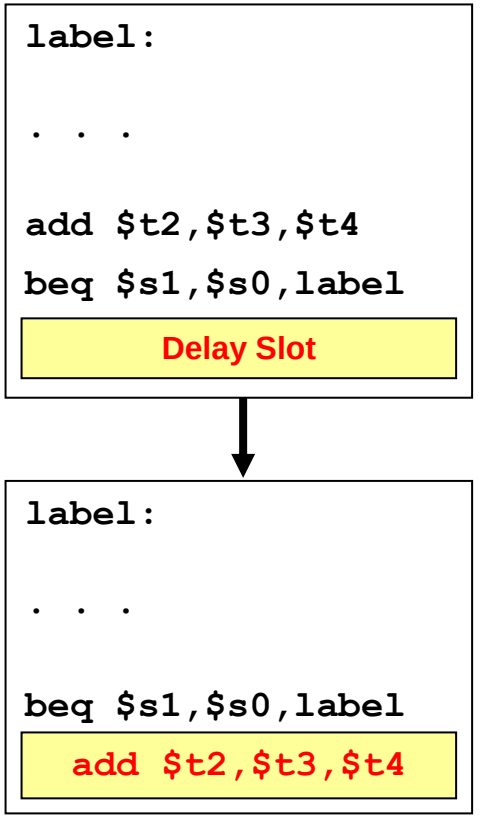
- Pipelined Datapath and Control
- Pipeline Hazards
- Data Hazards and Forwarding
- Load Delay, Hazard Detection, and Stall
- Control Hazards
- **Delayed Branch and Dynamic Branch Prediction**

Branch Hazard Alternatives

- Predict Branch Not Taken (previously discussed)
  - Successor instruction is already fetched
  - Do NOT kill instructions if the branch is NOT taken
  - Kill only instructions appearing after Jump or taken branch
- 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
  - Loop branches are taken most of time
  - Must reduce the branch delay to 0, but how?
  - How to predict branch behavior at runtime?

Delayed Branch

- Define branch to take place after the next instruction
- MIPS defines one delay slot
  - Reduces branch penalty
- Compiler fills the branch delay slot
  - By selecting an independent instruction from before the branch
  - Must be okay to execute instruction in the delay slot whether branch is taken or not
- If no instruction is found
  - Compiler fills delay slot with a NO-OP



Compiler Instruction Scheduling Example With Branch Delay Slot

- Schedule the following MIPS code for the pipelined MIPS CPU with forwarding and reduced branch delay using a single branch delay slot to minimize stall cycles:

```
loop: lw $1,0($2)          #  $1 array element
St……..
      add $1, $1, $3       # add constant in $3
      sw $1,0($2)          # store result array element
      addi $2, $2, -4      # decrement address by 4
St…….
      bne $2, $4, loop  # branch if $2 != $4
```

May st

- Assuming the initial value of $2 = $4 + 40

  (i.e it loops 10 times)

  - What is the CPI and total number of cycles needed to run the code with and without scheduling?

Compiler Instruction Scheduling Example With Branch Delay Slot)

- **Without compiler scheduling**

```
loop: lw $1,0($2)
      Stall
      add $1, $1, $3
      sw $1,0($2)
      addi $2, $2, -4
      bne $2, $4, loop
      Stall (or NOP)
```

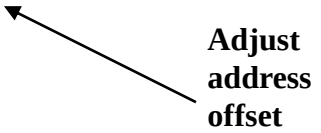
Ignoring the initial 4 cycles to fill the

pipeline:

Each iteration takes = 7 cycles
CPI = 7/5 = 1.4
Total cycles = 7 x 10 = 70 cycles

- **With compiler scheduling**

```
loop: lw $1,0($2)
      addi $2, $2, -4
      add $1, $1, $3
      bne $2, $4, loop
      sw $1, 4($2)
```



Ignoring the initial 4 cycles to fill the

pipeline:

Each iteration takes = 5 cycles
CPI = 5/5 = 1
Total cycles = 5 x 10 = 50 cycles
Speedup = 70/ 50 = 1.4

Drawback of Delayed Branching

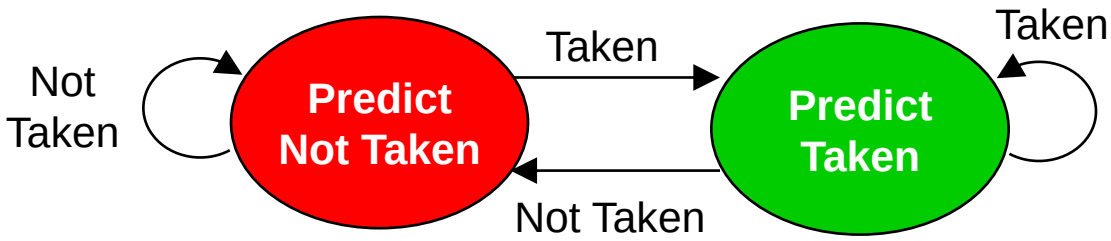
- New meaning for branch instruction
  - Branching takes place after next instruction (Not immediately!)
- Impacts software and compiler
  - Compiler is responsible to fill the branch delay slot
- However, modern processors and deeply pipelined
  - Branch penalty is multiple cycles in deep pipelines
  - Multiple delay slots are difficult to fill with useful instructions
- MIPS used delayed branching in earlier pipelines
  - However, delayed branching lost popularity in recent processors
  - Dynamic branch prediction has replaced delayed branching

Dynamic Branch Prediction

- Prediction of branches at runtime using prediction bits
- Prediction bits are associated with each entry in the BTB
  - Prediction bits reflect the recent history of a branch instruction
- Typically few prediction bits (1 or 2) are used per entry
- We don't know if the prediction is correct or not
- If correct prediction …
  - Continue normal execution – no wasted cycles
- If incorrect prediction (misprediction) …
  - Kill the instructions that were incorrectly fetched ➔ wasted cycles
  - Update prediction bits and target address for future use

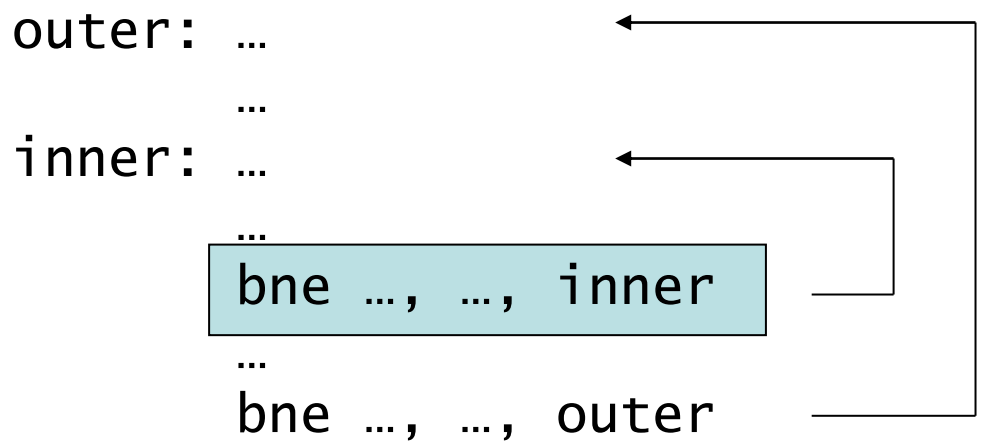
1-bit Prediction Scheme

- Prediction is just a hint that is assumed to be correct
- If incorrect then fetched instructions are killed
- 1-bit prediction scheme is simplest to implement
  - 1 bit per branch instruction (associated with BTB entry)
  - Record last outcome of a branch instruction (Taken/Not taken)
  - Use last outcome to predict future behavior of a branch



1-Bit Predictor: Shortcoming

- Inner loop branch mispredicted twice!
  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around



Pipeline Performance Example

- Assume the following MIPS instruction mix:

Type	Frequency	
Arith/Logic	40%	
Load	30%	of which 25% are followed immediately by an instruction using the loaded value
Store	10%	
branch	20%	of which 45% are taken

- What is the resulting CPI for the pipelined MIPS with forwarding and branch address calculation in ID stage when using the branch not-taken scheme?

- CPI = Ideal CPI + Pipeline stall clock cycles per instruction

  = 1 + stalls by loads + stalls by branches

  = 1 + .3 x .25 x 1 + .2 x .45 x 1

  = 1 + .075 + .09

  = 1.165

Exceptions

- Some exception types
  - Overflow
  - Illegal instruction
  - I/O device request
- Instruction causing an overflow

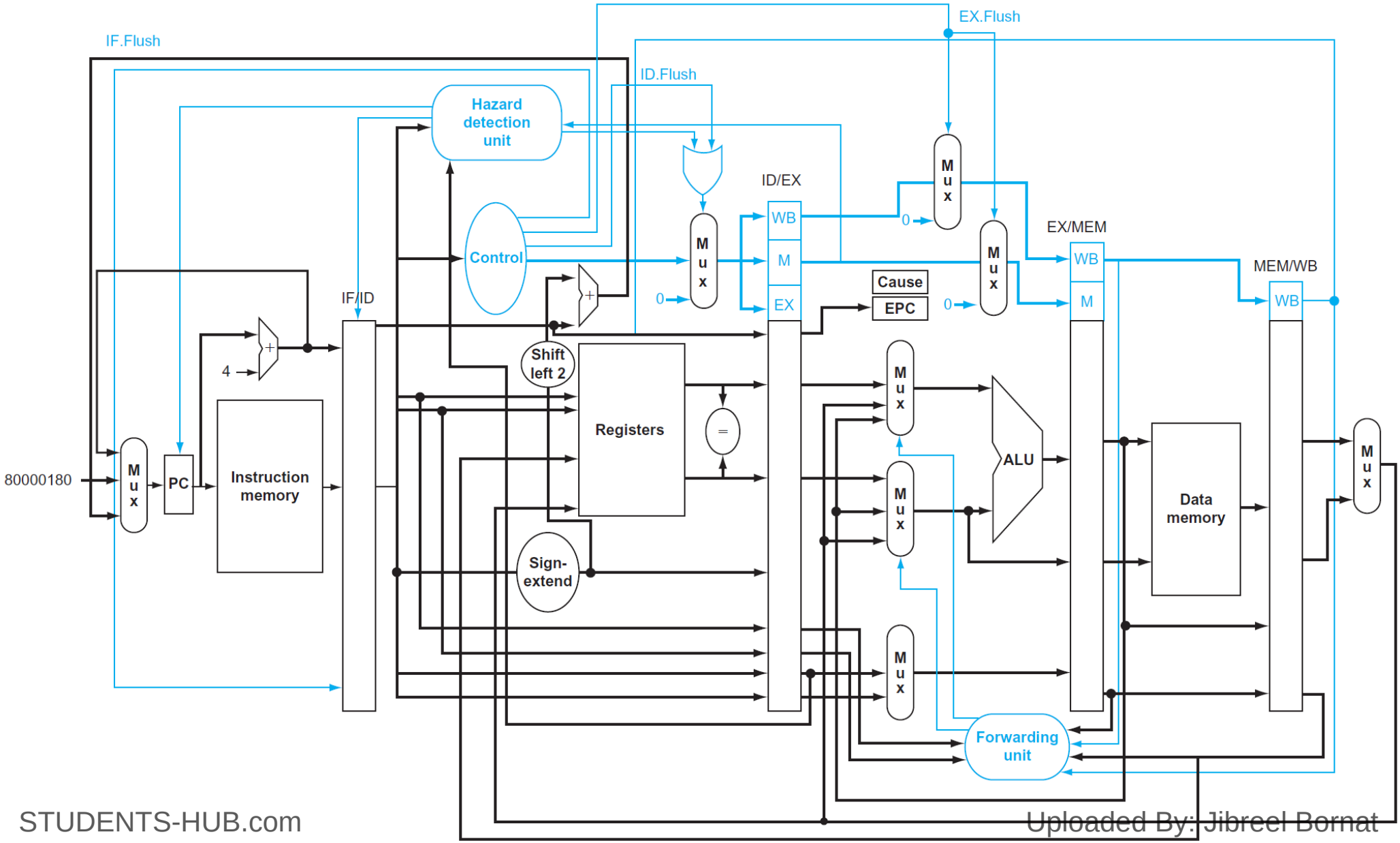
  add $1, $2, $1
- Action
  - Load PC with exception handling address
  - Flush the remaining instructions from pipeline immediately
  - Leave registers untouched

    Example old/new value of $1

Exceptions Handling

- A pipelined implementation treats exceptions as another form of control hazard.
- We will use the same mechanism we used for taken branches, but this time the exception causes the deasserting of control lines.
- To flush instructions in the ID stage, we use the multiplexor already in the ID stage that zeros control signals for stalls. A new control signal, called ID.Flush, is ORed with the stall signal from the hazard detection unit to fl ush during ID. To flush the instruction in the EX phase, we use a new signal called EX.Flush to cause new multiplexors to zero the control lines.
- To start fetching instructions from location 8000 0180hex, which is the MIPS exception address, we simply add an additional input to the PC multiplexor that sends 8000 0180hex to the PC.
- Save the address of the off ending instruction in the *exception program counter* (EPC).

Modified CPU Data-path



The datapath with controls to handle exceptions

- The key additions include:
  - A new input with the value 8000 0180hex in the multiplexor that supplies the new PC value;
  - A Cause register to record the cause of the exception;
  - And an Exception PC register to save the address of the instruction that caused the exception.
- The 8000 0180hex input to the multiplexor is the initial address to begin fetching instructions in the event of an exception.
- Although not shown, the ALU overflow signal is an input to the control unit.

Example

- Given this instruction sequence,

  40_{hex} sub $11, $2, $4

  44_{hex} and $12, $2, $5

  48_{hex} or $13, $2, $6

  $4C_{hex}$ add $1, $2, $1

  50_{hex} slt $15, $6, $7

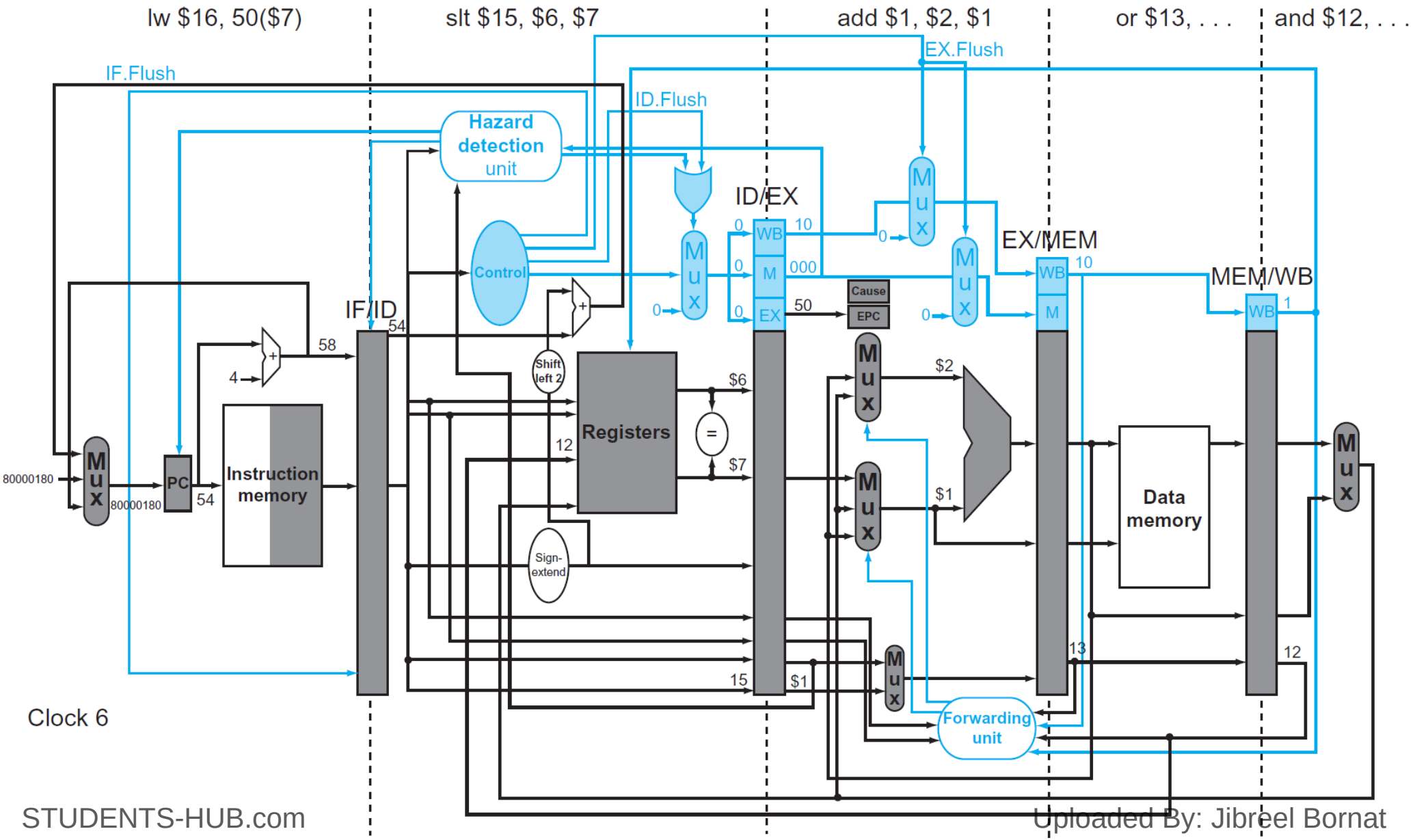
  54_{hex} lw $16, 50($1)

- Assume the instructions to be invoked on an exception begin like this:

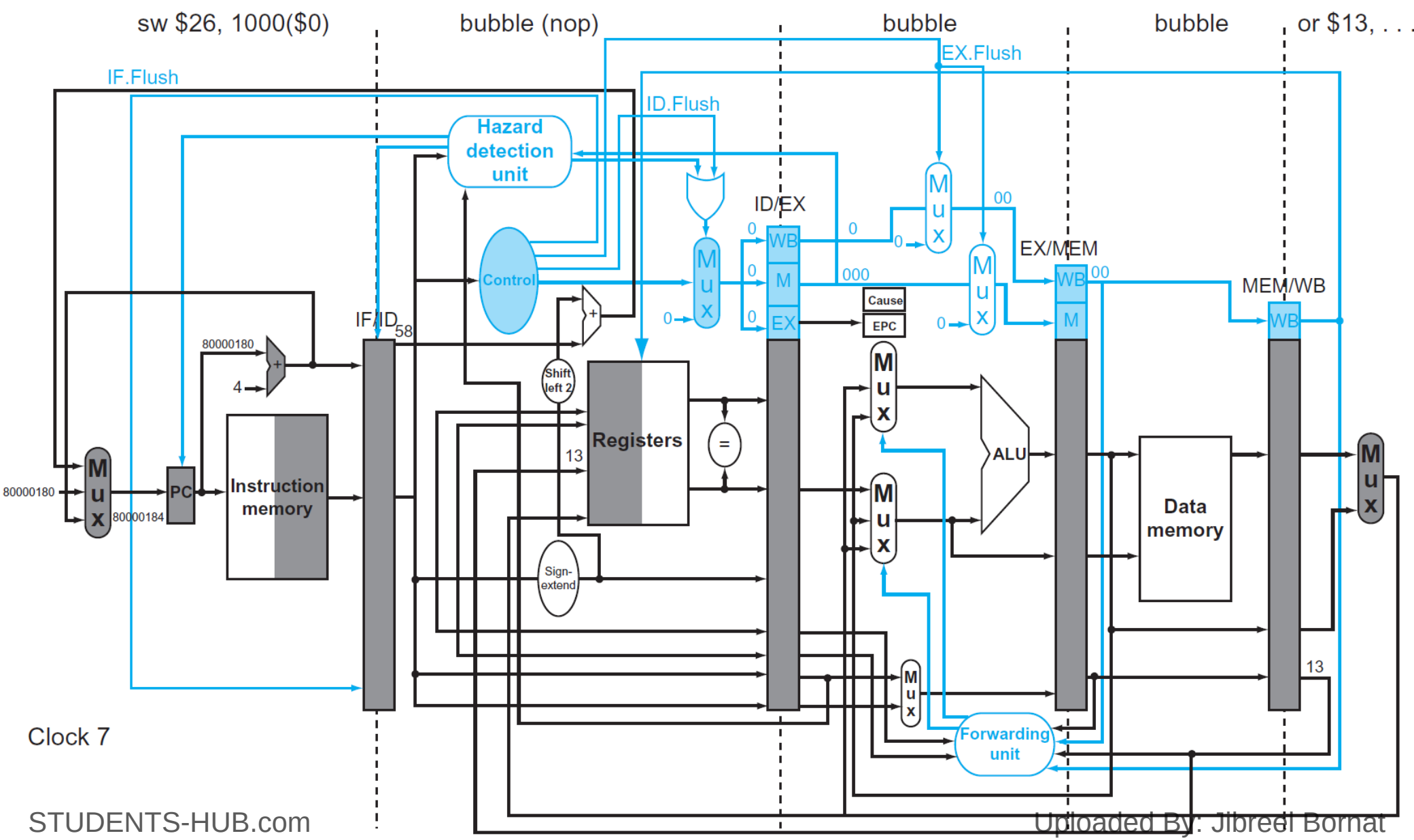
  4000 0040 sw $25, 1000($0)

  4000 0044 sw $26, 1004($0)

Overflow handling



Overflow handling



The result of an exception due to arithmetic overflow ow in the add instruction.

- The overflow is detected during the EX stage of clock 6
  - Saving the address following the add in the EPC register (4C + 4 = 50hex).
  - Overflow causes all the Flush signals to be set near the end of this clock cycle,
  - Deasserting control values (setting them to 0) for the add.
- Clock cycle 7 shows the instructions converted to bubbles in the pipeline plus the fetching of the first instruction of the exception routine—sw $25,1000($0)—from instruction location 8000 0180hex.
- Note that the AND and OR instructions, which are prior to the add, still complete. Although not shown, the ALU overflow signal is an input to the control unit.

Final data path

- Basic pipelined architecture
- Forwarding
- Hazard detection unit
- Branch handling
- Exception handling

In Summary

- Three types of pipeline hazards
  - Structural hazards: conflicts using a resource during same cycle
  - Data hazards: caused by data dependencies between instructions
  - Control hazards: caused by 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 reduces branch delay but needs compiler support
  - BTB with branch prediction can reduce branch delay to zero
  - Branch misprediction should kill the wrongly fetched instructions