Instruction Pipelining: Basic and Intermediate Concepts

COE 501

Computer Architecture

Prof. Muhamed Mudawar

Computer Engineering Department

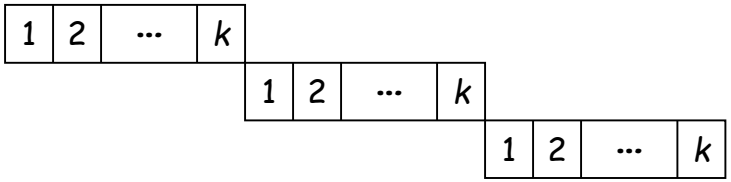
King Fahd University of Petroleum and Minerals

Presentation Outline

- **Pipelining Basics**
- MIPS 5-Stage Pipeline Microarchitecture
- Structural Hazards
- Data Hazards and Forwarding
- Load Delay and Pipeline Stall
- Control Hazards and Branch Prediction

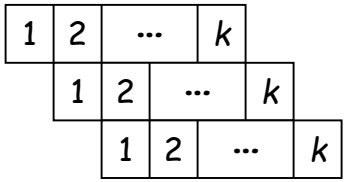
What is Pipelining?

- 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



Serial Execution

One completion every *k* time units

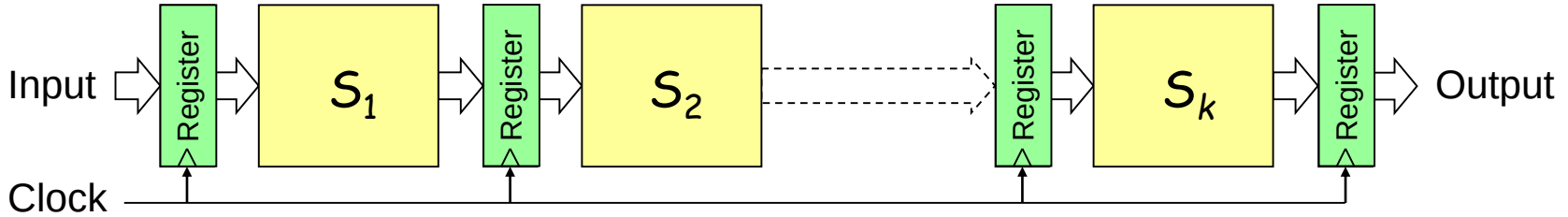


Pipelined Execution

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

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

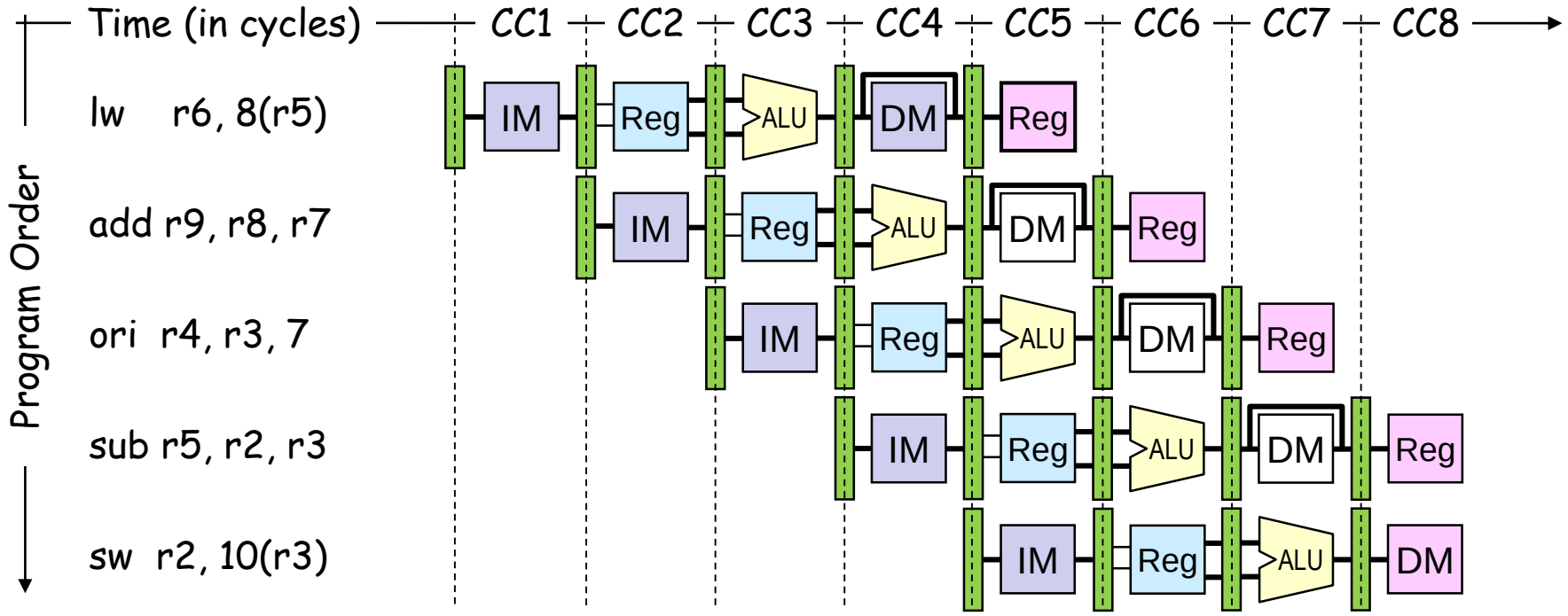
Simple 5-Stage Processor Pipeline

- Five stages, one cycle per stage

1. **IF:** Instruction Fetch from instruction memory
   - Select address: next instruction, jump target, branch target
2. **ID:** Instruction Decode
   - Determine control signals & read registers from the register file
3. **EX:** Execute operation
   - Load and Store: Calculate effective memory address
   - Branch: Calculate address and outcome (Taken or Not Taken)
4. **MEM:** Memory access for load and store only
5. **WB:** Write Back result to register

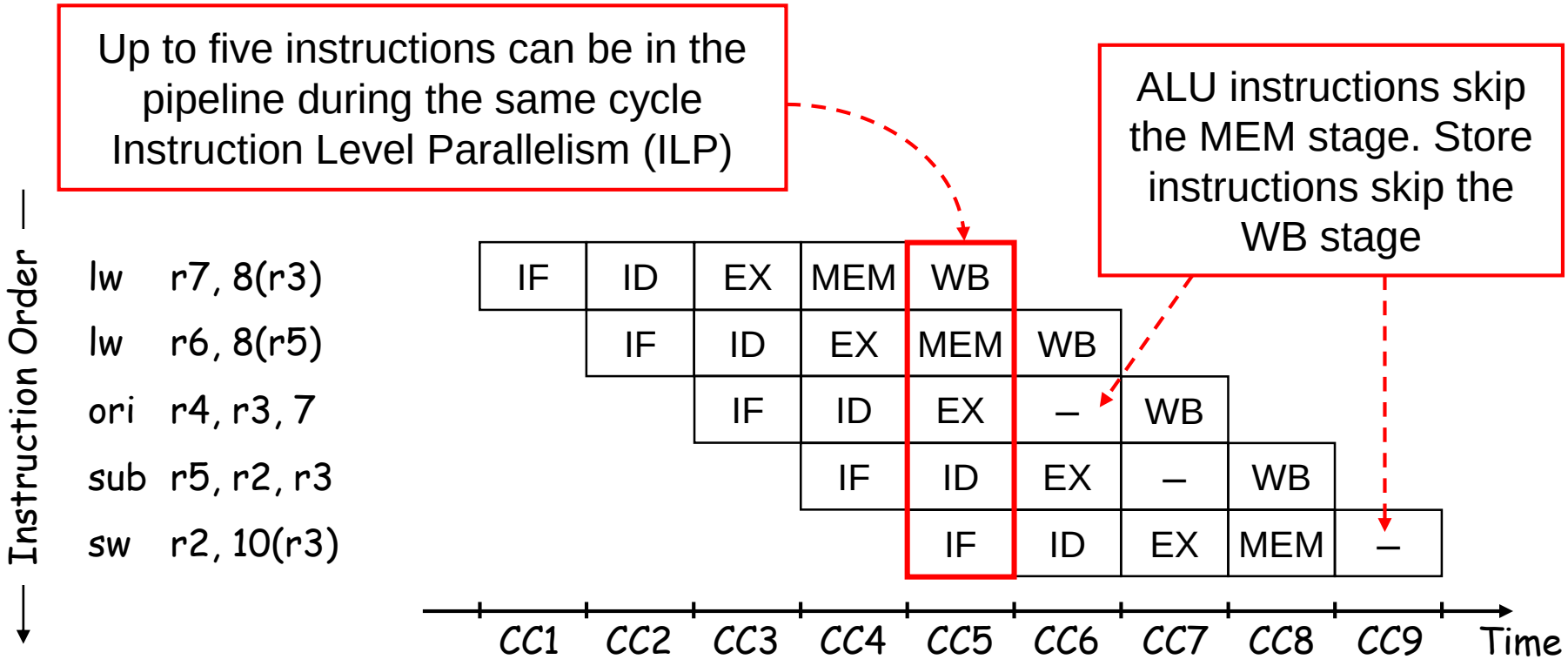
Visualizing the Pipeline

- Multiple instruction execution over multiple clock cycles
  - Instructions are listed in program order from top to bottom
  - Figure shows the use of resources at each stage and each cycle
  - No interference between different instructions in adjacent stages



Timing the Instruction Flow

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



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:**

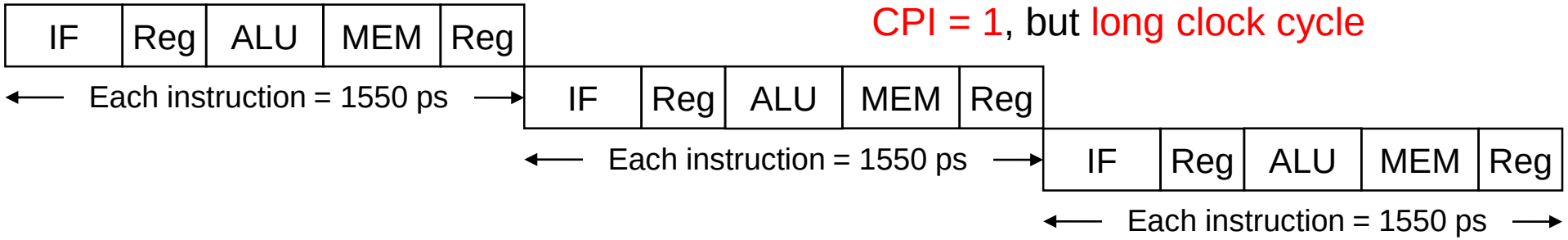
Instruction	Fetch	Reg Read	ALU	Memory	Reg Wr	Time
Load	350 ps	250 ps	350 ps	350 ps	250 ps	1550 ps
Store	350 ps	250 ps	350 ps	350 ps		1300 ps
ALU	350 ps	250 ps	350 ps		250 ps	1200 ps
Branch	350 ps	250 ps	350 ps			950 ps

Single-Cycle, Multi-Cycle, Pipelined

Single-Cycle Execution:

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

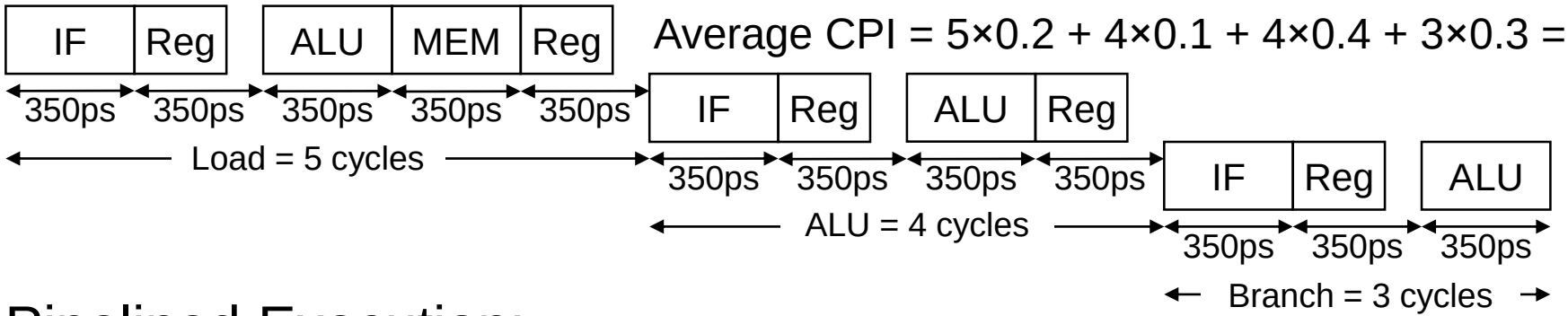
CPI = 1, but long clock cycle



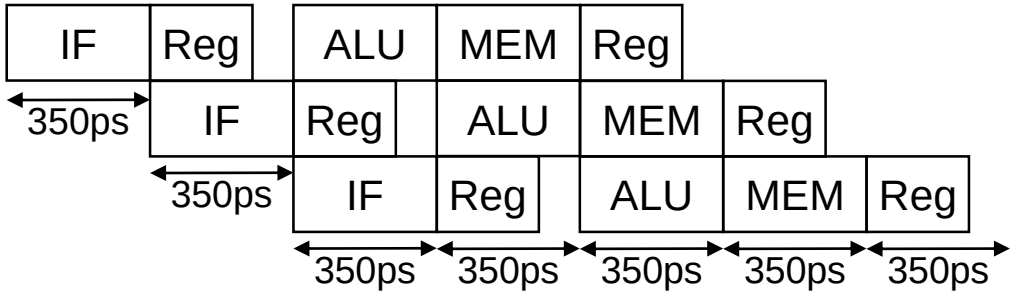
Multi-Cycle Execution:

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

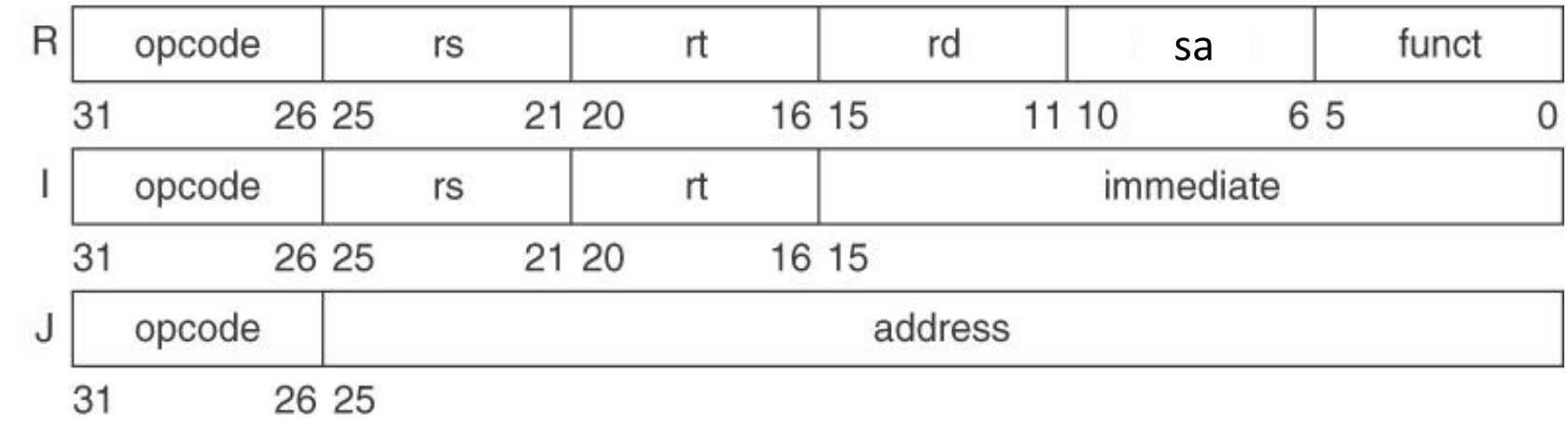
Next . . .

- Pipelining Basics
- **5-Stage Pipeline Microarchitecture**
- Structural Hazards
- Data Hazards and Forwarding
- Load Delay and Pipeline Stall
- Control Hazards and Branch Prediction

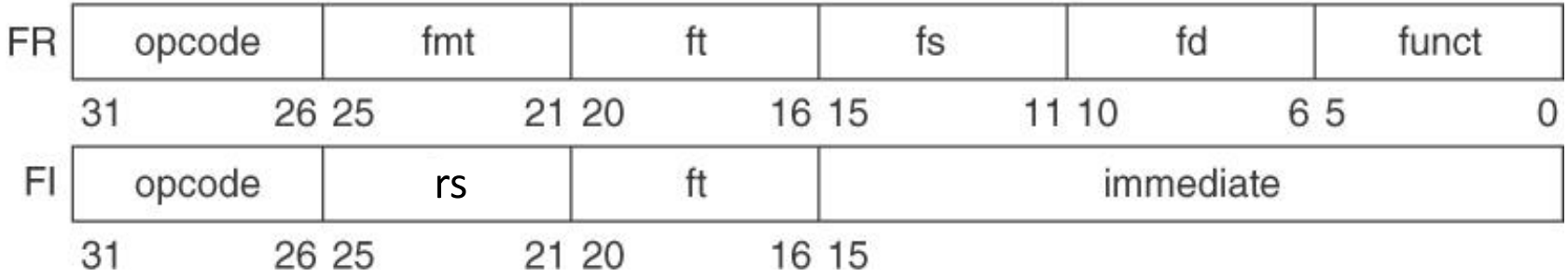
MIPS Instruction Formats

- All instructions are 32 bits with a 6-bit primary opcode
- These are the main instruction formats, not the only ones

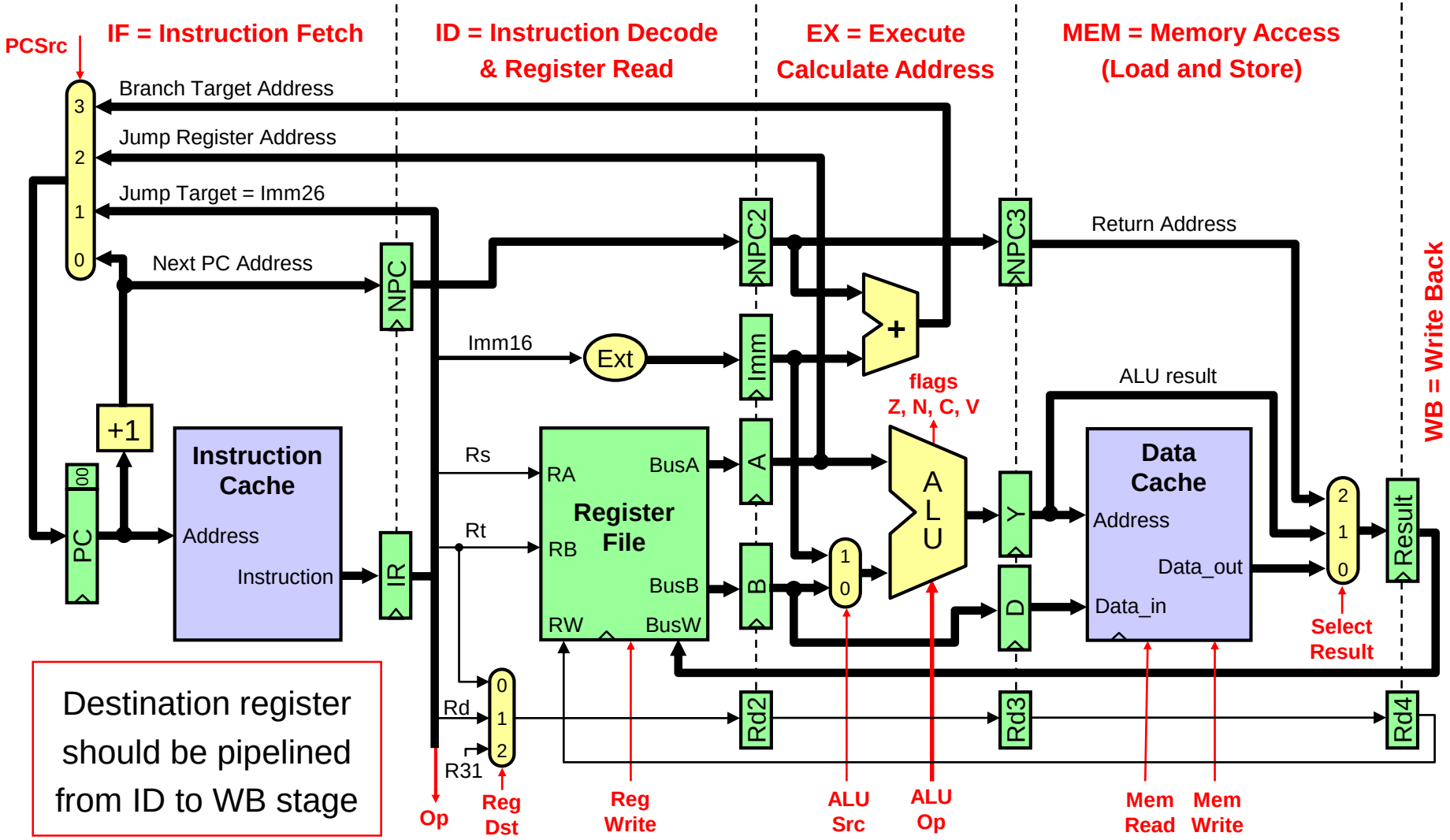
Basic instruction formats



Floating-point instruction formats



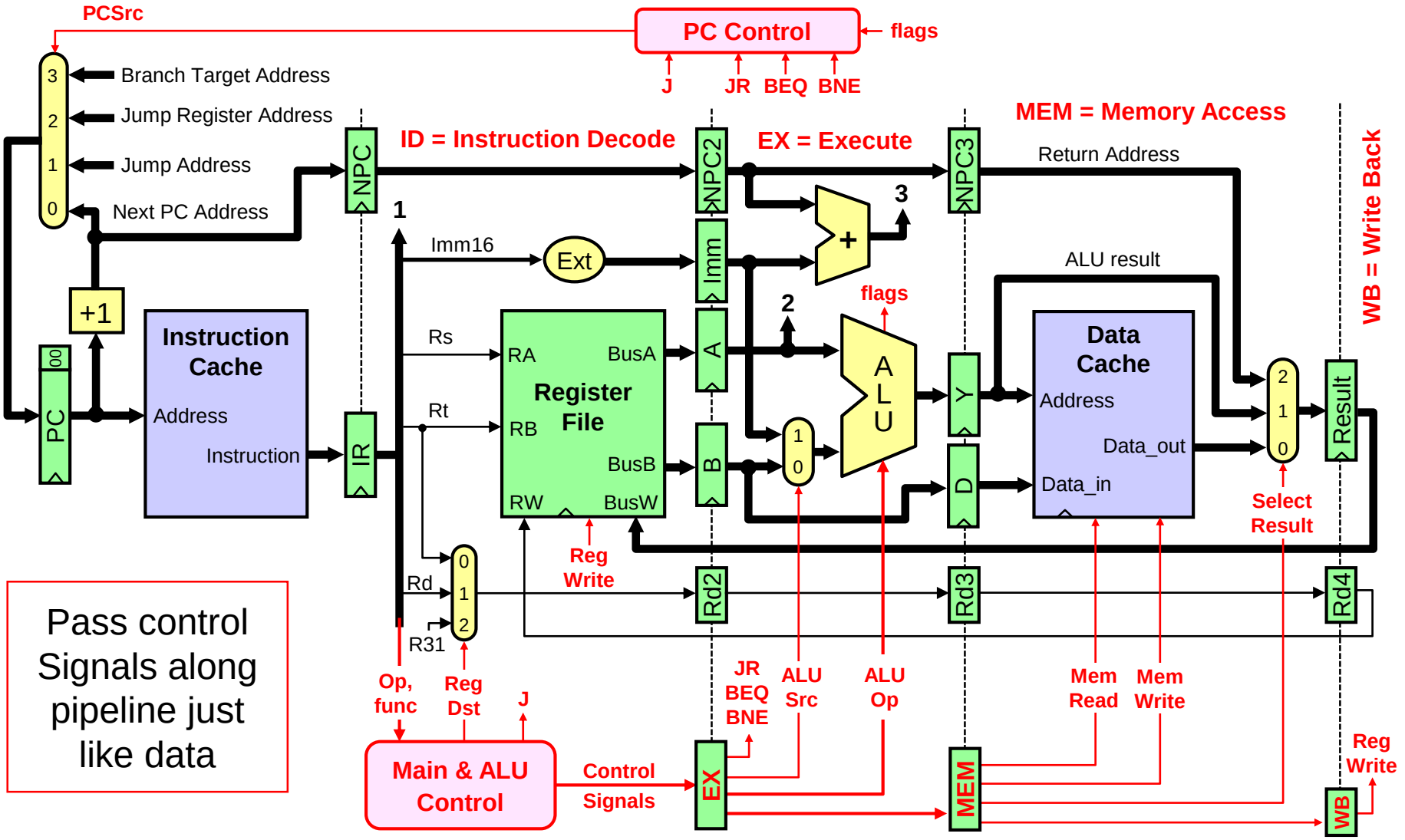
5-Stage Pipeline



Pipelined Control

- 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 (ID) Stage
    - Generate all control signals
    - Select destination register: RegDst control signal
    - PC control uses: J (Jump) control signal for PCSrc
  - Execution Stage ➔ ALUSrc, and ALUOp
    - PC control uses: JR, Beq, Bne, and ALU flags for PCSrc
  - Memory Stage ➔ MemRead, MemWrite, and SelectResult
  - Write Back Stage ➔ RegWrite is used in this stage

Pipelined Datapath + Control



Next . . .

- Pipelining Basics
- 5-Stage Pipeline Microarchitecture
- **Structural Hazards**
- Data Hazards and Forwarding
- Load Delay and Pipeline Stall
- Control Hazards and Branch Prediction

Pipeline Hazards

- **Hazard:** Situation that would cause incorrect execution
  - If next instruction were launched during its designated clock cycle

1. Structural hazard

  - Caused by hardware resource contention
  - Using same resource by two instructions during same clock cycle

2. Data hazard

  - An instruction may depend on the result of a prior instruction still in the pipeline, that did not write back its result into the register file

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

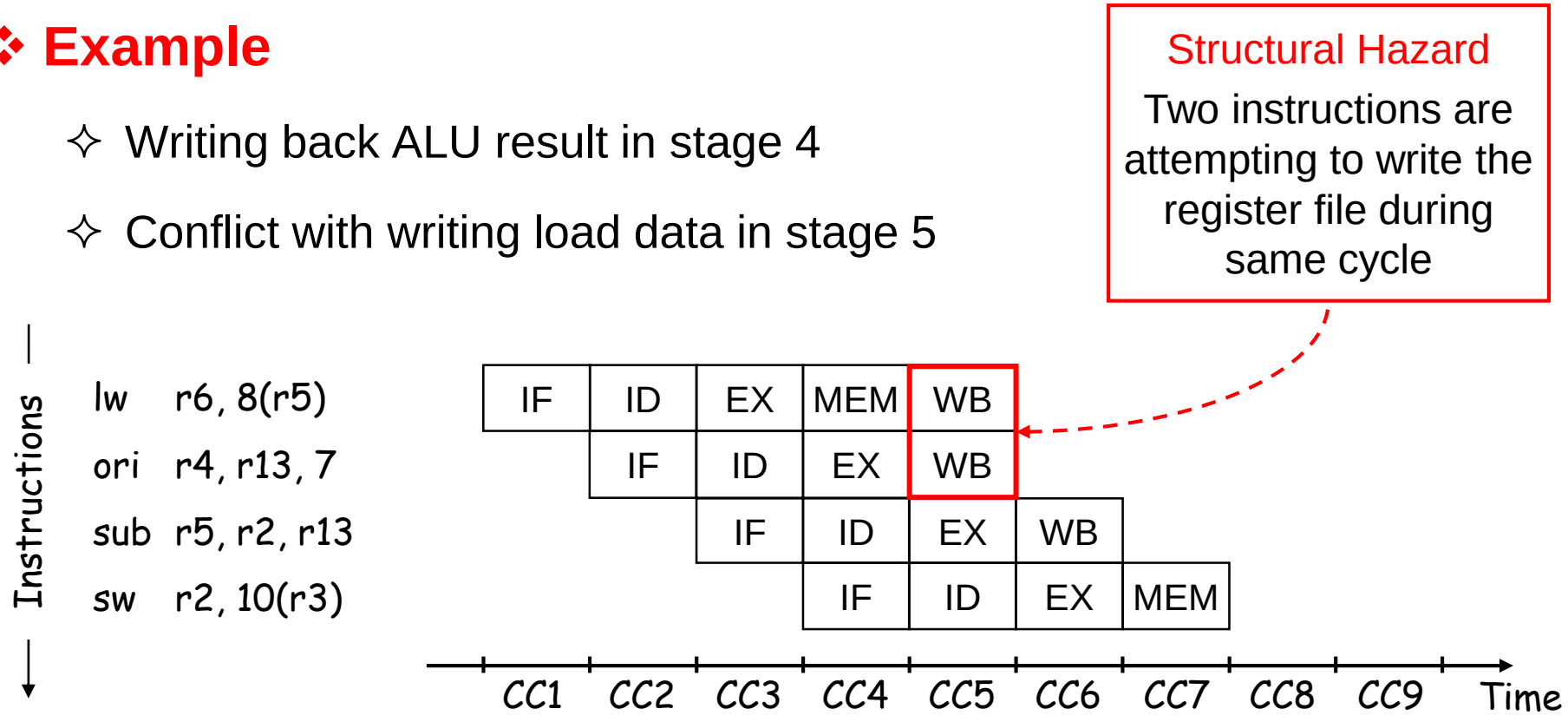
Structural Hazard

❖ Definition

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

❖ Example

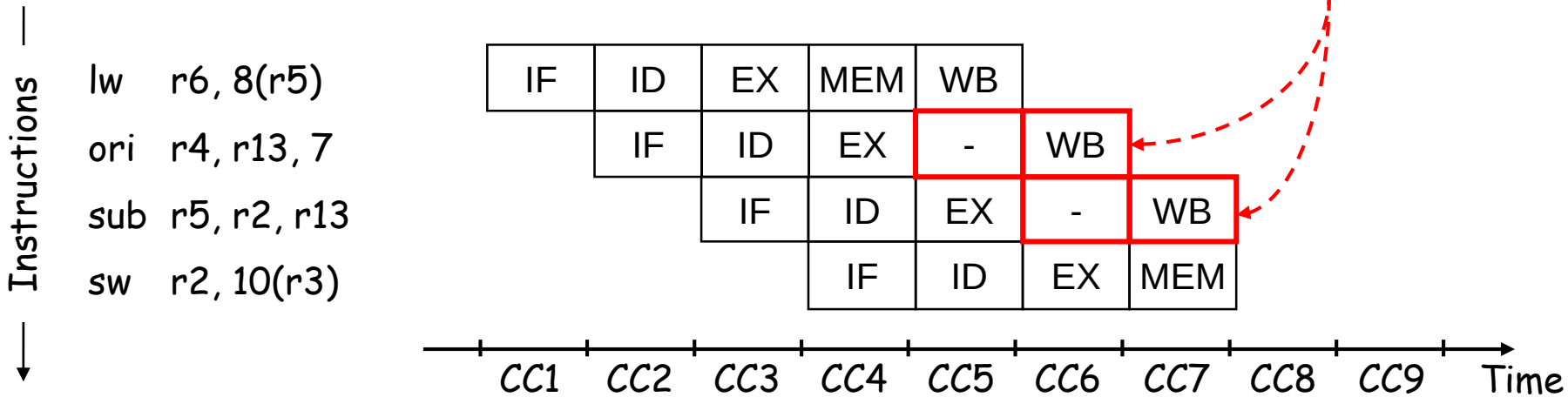
- ✧ Writing back ALU result in stage 4
- ✧ Conflict with writing load data in stage 5



Resolving Structural Hazard

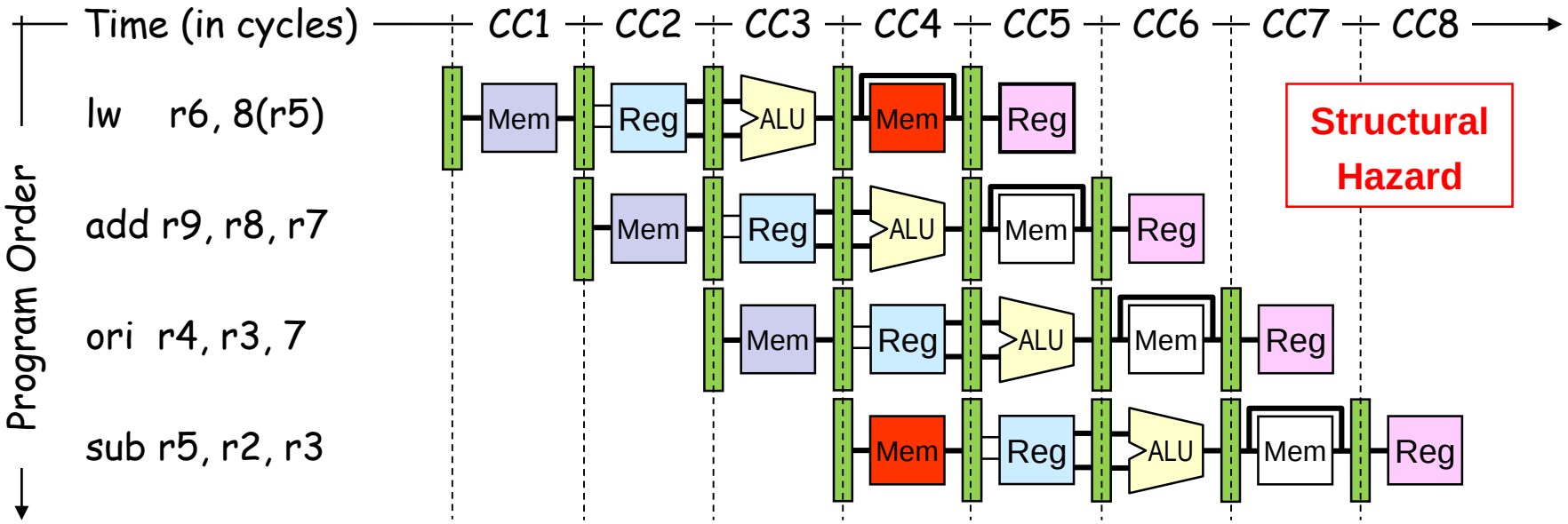
- Is it serious? Yes! cannot be ignored
- Solution 1: Delay access to resource
  - Delay Write Back to Stage 5
- Solution 2: Add more hardware
  - Two write ports for register file (costly)
    - Does not improve performance
    - One fetch ➔ one completion per cycle

Resolving Structural Hazard
Delay access to register file
Write Back occurs only in Stage 5



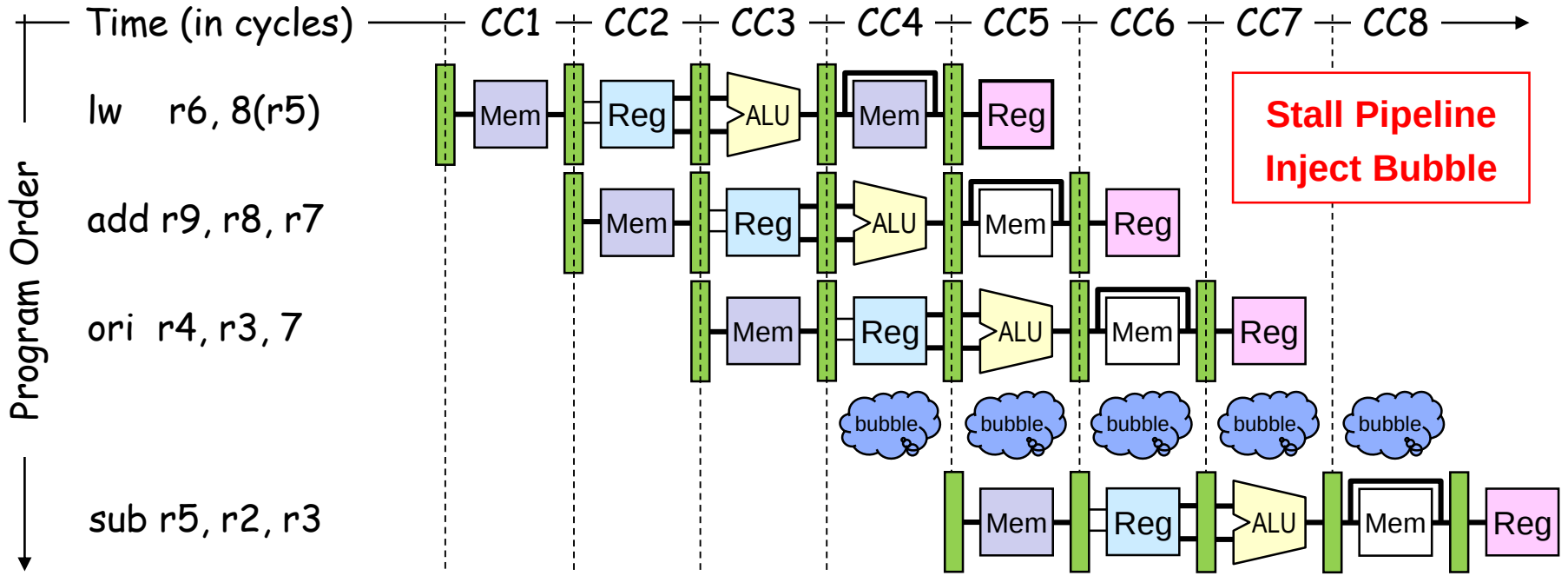
Example 2 of Structural Hazard

- 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



Resolving Structural Hazards

- **Serious Hazard**: structural hazard cannot be ignored
  - Can be eliminated with careful design of the pipeline
- **Solution 1: Delay Access to Hardware Resource**
  - Such as having all write backs to register file in the last stage, or
  - Stall the pipeline until resource is available
- **Solution 2: Add more Hardware Resources**
  - Add more hardware to eliminate the structural hazard
  - Such as having two cache memories for instructions & data
  - I-Cache and D-Cache can be addressed in parallel (same cycle)
    - Known as Harvard Architecture
  - Better than having two address ports for same cache memory

Performance Example

- Processor A: I-Cache + D-Cache (Harvard Architecture)
- Processor B: single-ported cache for both instructions & data
- B has a clock rate 1.05X faster than clock rate of A
- Loads + Stores = 40% of instructions executed
- Ideal pipelined CPI = 1 (if no stall cycles)
- Which processor is faster and by what factor?
- **Solution:**

$CPI_A = 1$, $CPI_B = 1+0.4$ (due to structural hazards)

$$\text{Speedup}_{A/B} = \frac{CPI_B}{CPI_A} \times \frac{\text{Clock rate}_A}{\text{Clock rate}_B} = \frac{1+0.4}{1} \times \frac{1}{1.05} = 1.33$$

Next . . .

- Pipelining Basics
- 5-Stage Pipeline Microarchitecture
- Structural Hazards
- **Data Hazards and Forwarding**
- Load Delay and Pipeline Stall
- Control Hazards and Branch Prediction

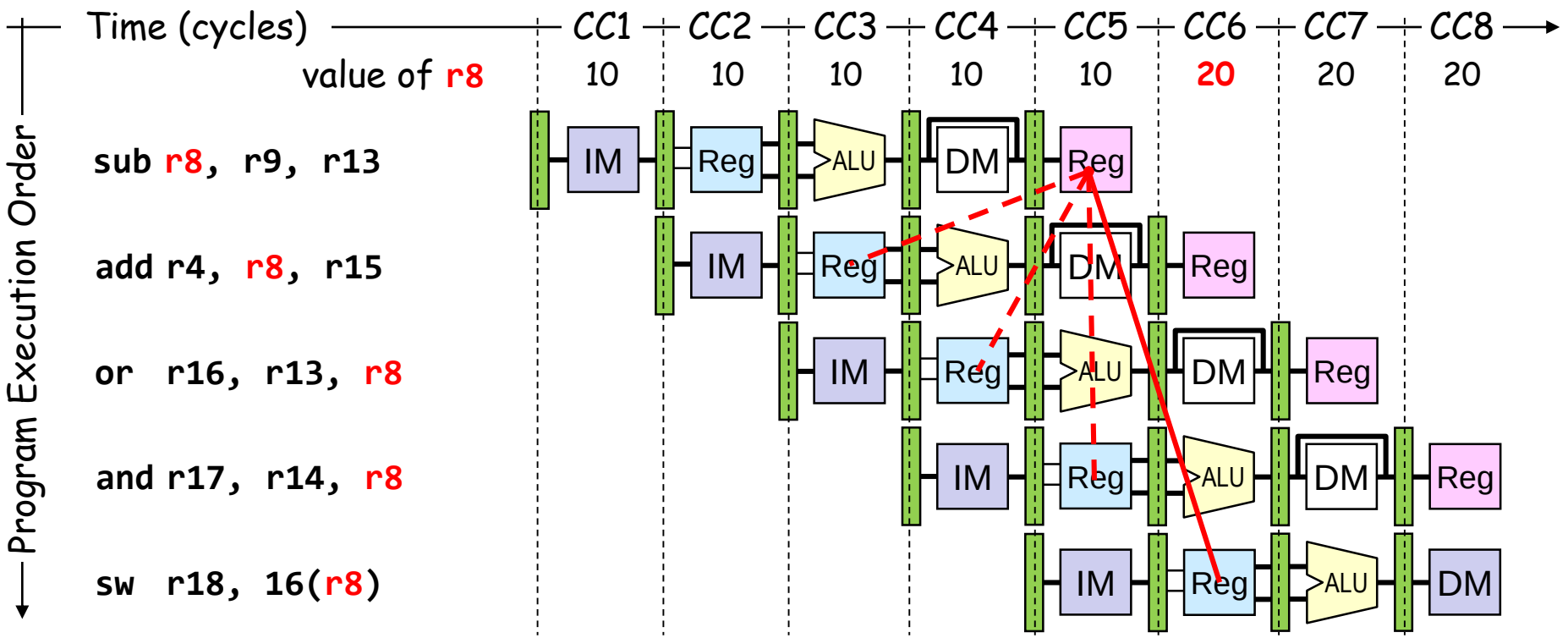
Data Hazard

- Occurs when one instruction depends on the result of a previous instruction still in the pipeline
  - Previous instruction did not write back its result to register file
  - Next instruction reads data before it is written
- **Data Dependence between instructions**
  - Given two instructions **I** and **J**, where **I** comes before **J**
  - Instruction **J** reads an operand written by **I**

```
I: add r8, r6, r10   ; I writes r8
J: sub r7, r8, r14   ; J reads r8
```

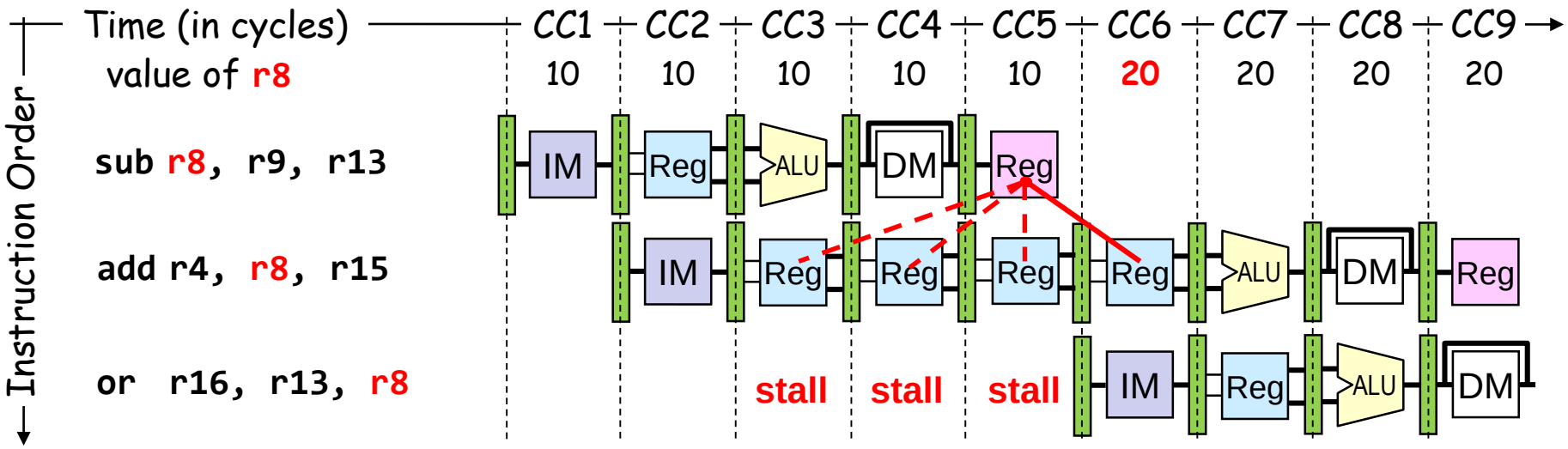
- **Read After Write: RAW Hazard**
  - Hazard occurs when **J** reads register **r8** before **I** writes it

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 **r8** from reg file
- During CC5, **r8** is written at end of cycle, **old value** is read

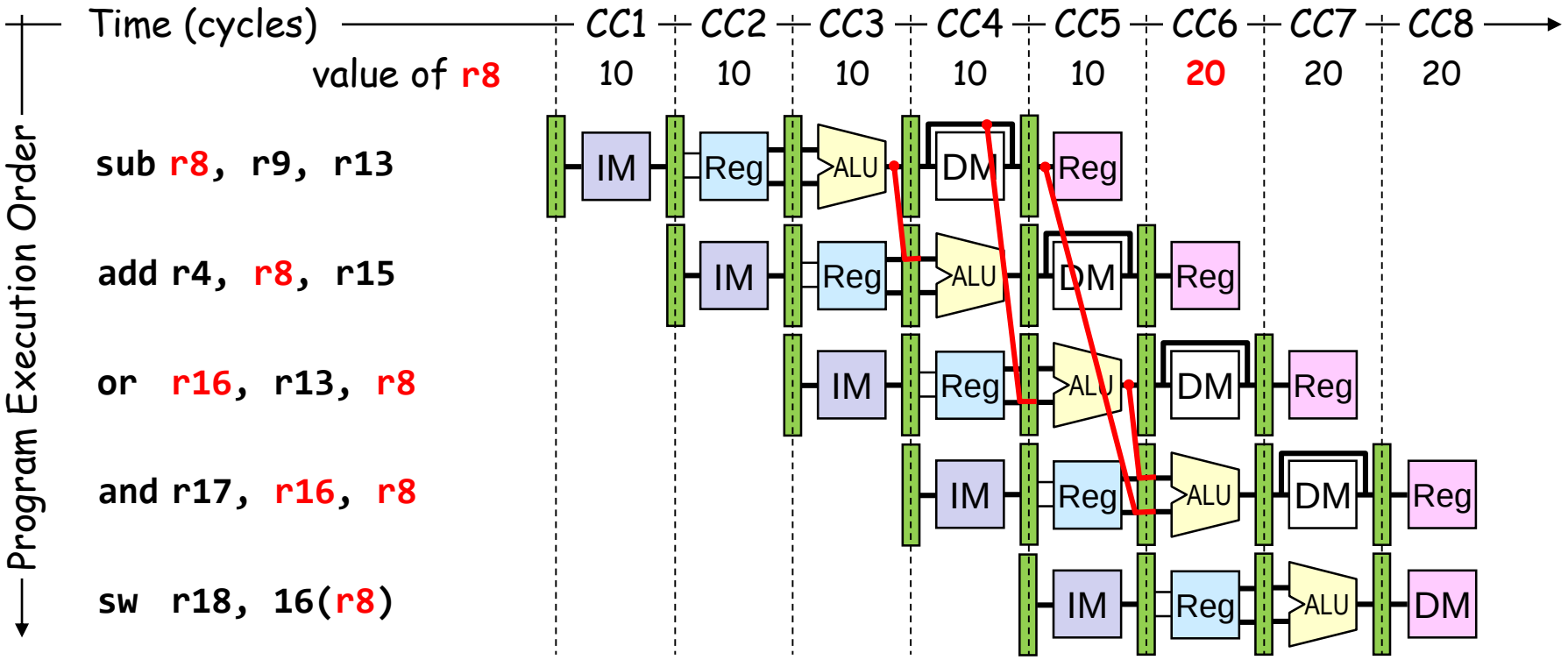
Solution 1: Stalling the Pipeline



- Three stall cycles during CC3 thru CC5 (wasting 3 cycles)
  - Stall cycles delay execution of **add** & fetching of **or** instruction
- The **add** instruction cannot read **r8** until beginning of CC6
  - The **add** instruction remains in the Instruction register until CC6
  - The PC register is not modified until beginning of CC6

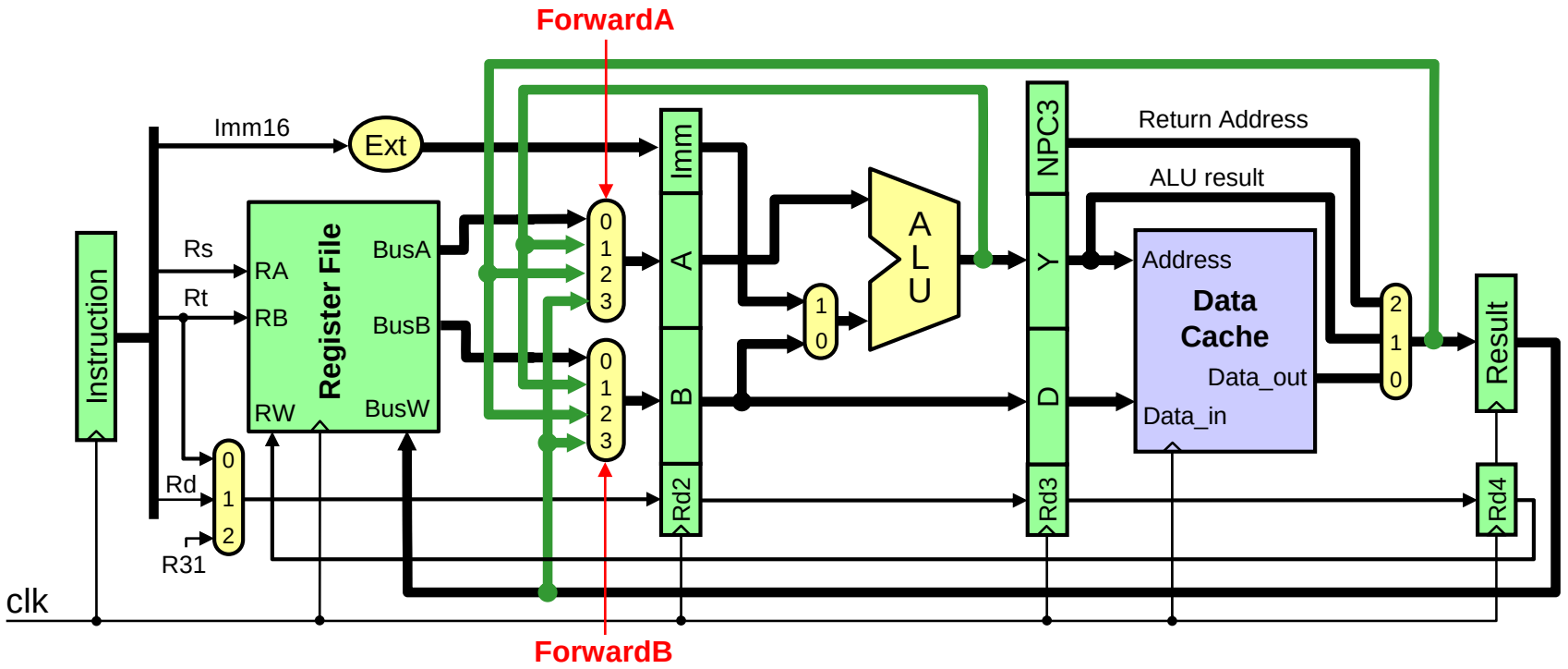
Solution 2: Forwarding ALU Result

- 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 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   r4, 4(r8)
ori  r7, r9, 2
sub  r3, r4, r7
```

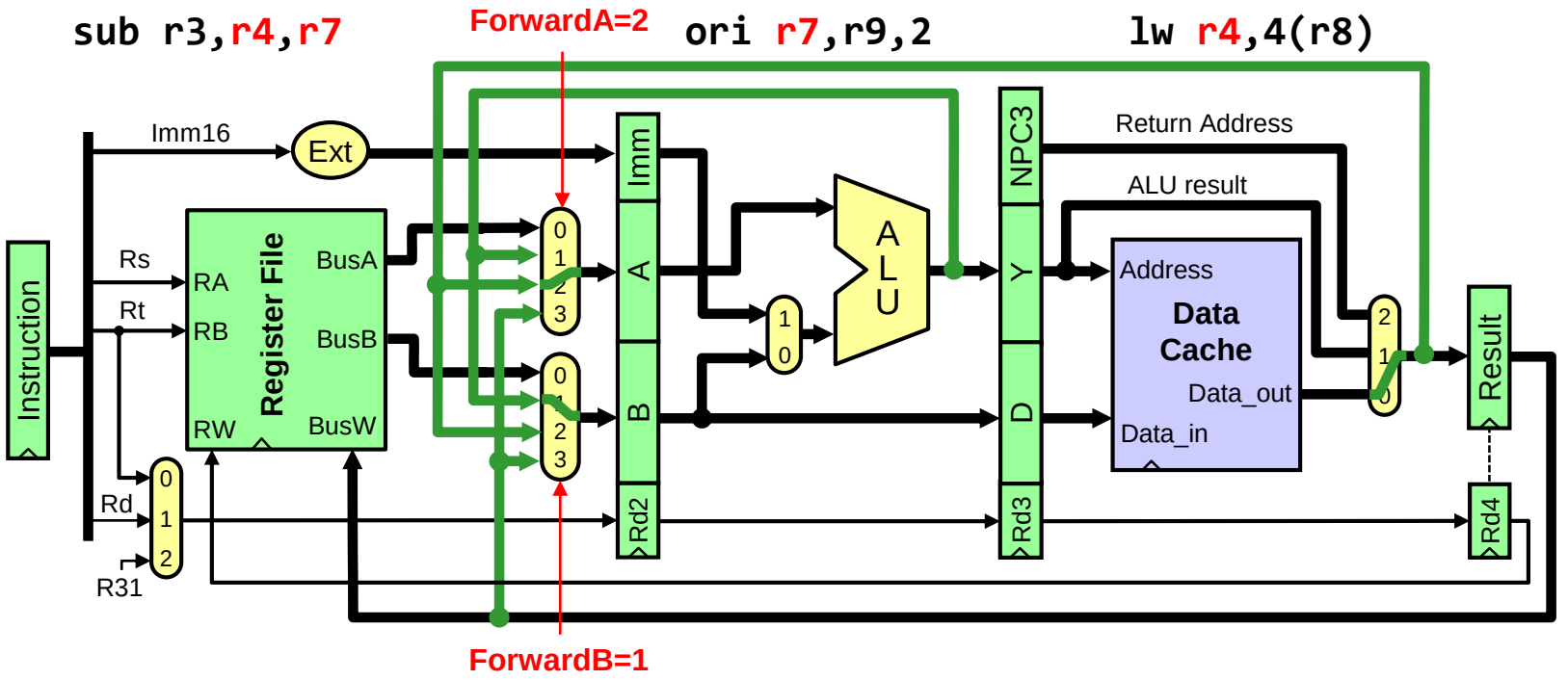
ForwardA = 2 from MEM stage

When **sub** instruction is fetched

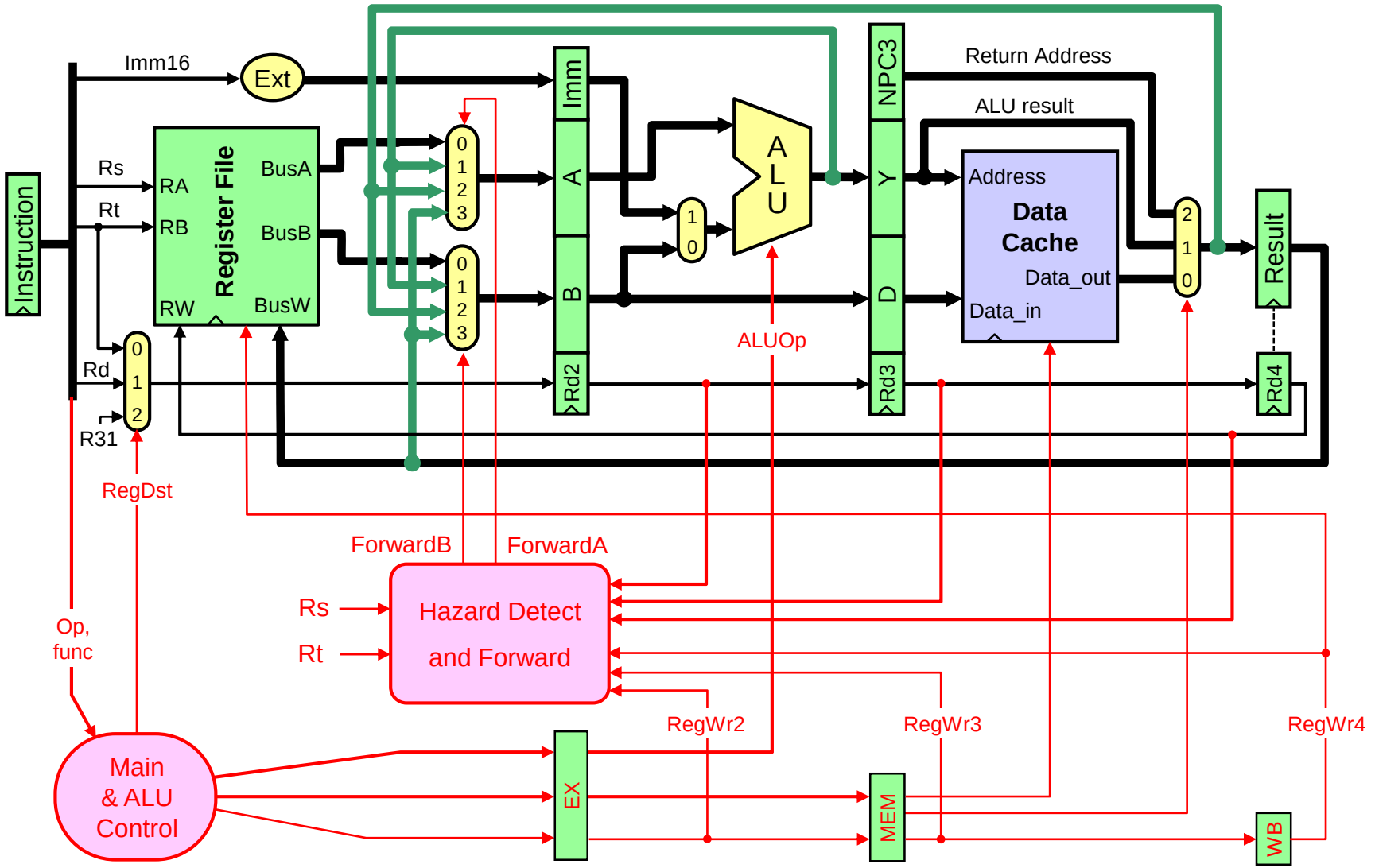
ori will be in the ALU stage

lw will be in the MEM stage

ForwardB = 1 from ALU stage



Hazard Detect and Forward Logic



Forwarding Equations

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

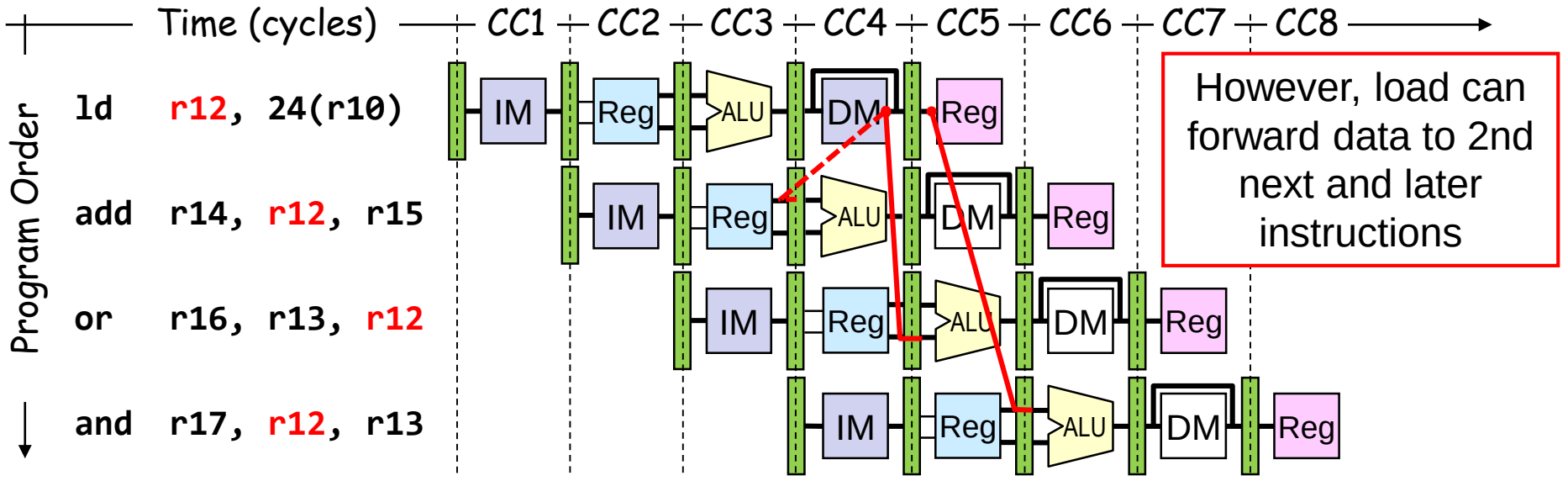
```
if      ((Rs != 0) && (Rs == Rd2) && (RegWr2)) ForwardA = 1
else if ((Rs != 0) && (Rs == Rd3) && (RegWr3)) ForwardA = 2
else if ((Rs != 0) && (Rs == Rd4) && (RegWr4)) ForwardA = 3
else    ForwardA = 0

if      ((Rt != 0) && (Rt == Rd2) && (RegWr2)) ForwardB = 1
else if ((Rt != 0) && (Rt == Rd3) && (RegWr3)) ForwardB = 2
else if ((Rt != 0) && (Rt == Rd4) && (RegWr4)) ForwardB = 3
else    ForwardB = 0
```

Next . . .

- Pipelining Basics
- 5-Stage Pipeline Microarchitecture
- Structural Hazards
- Data Hazards and Forwarding
- **Load Delay and Pipeline Stall**
- Control Hazards and 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 **LD** 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

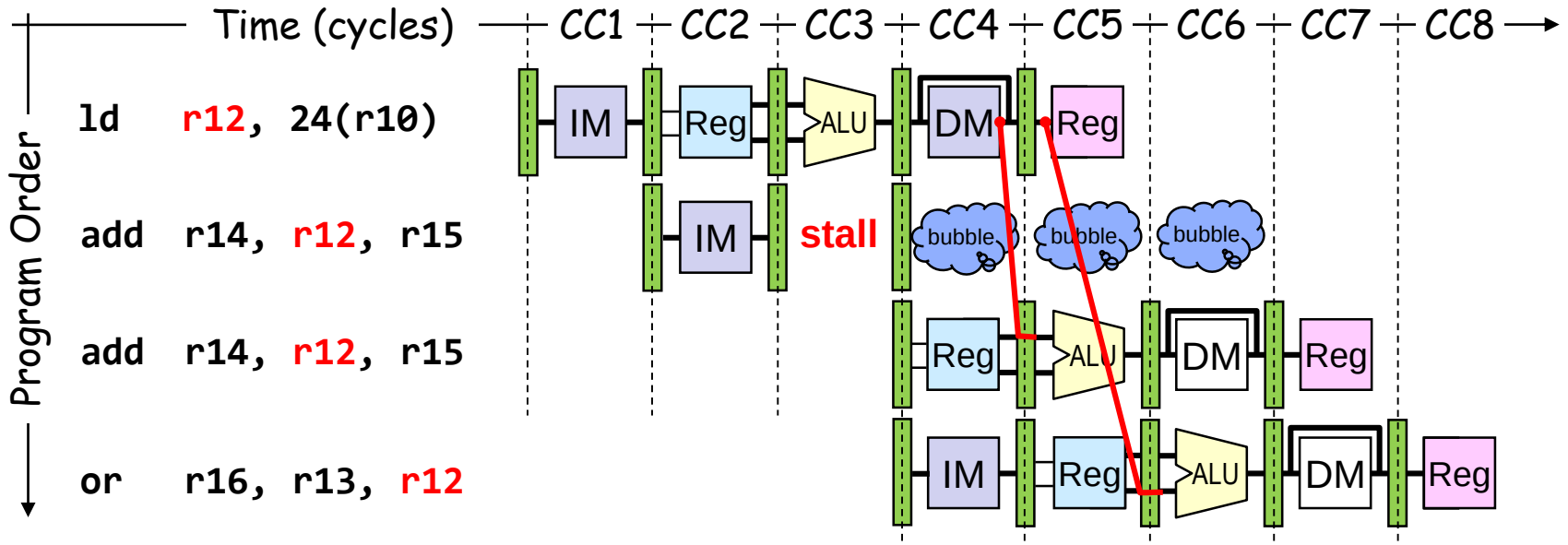
- Detecting a RAW hazard after a Load instruction:
  - The load instruction will be in the EX stage
  - Instruction that depends on the load data is in the decode stage
- Condition for stalling the pipeline

```
if ((EX.MemRead == 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 once cycle
    - Because of RAW hazard

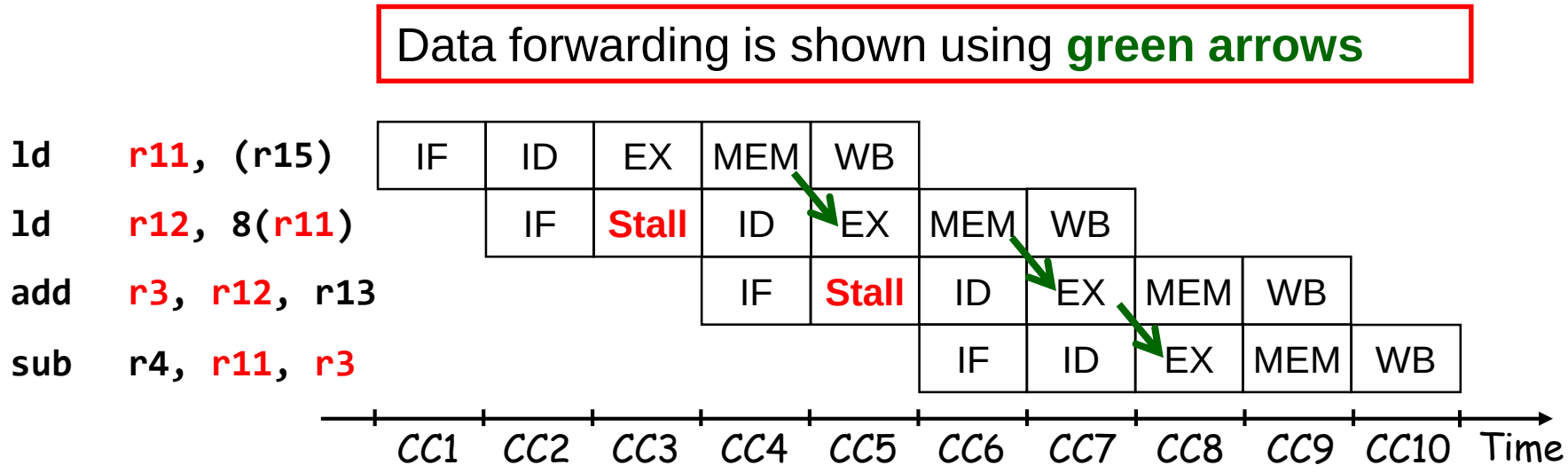
Stall the Pipeline for one Cycle

- **DADD** instruction depends on **LD** ➔ 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

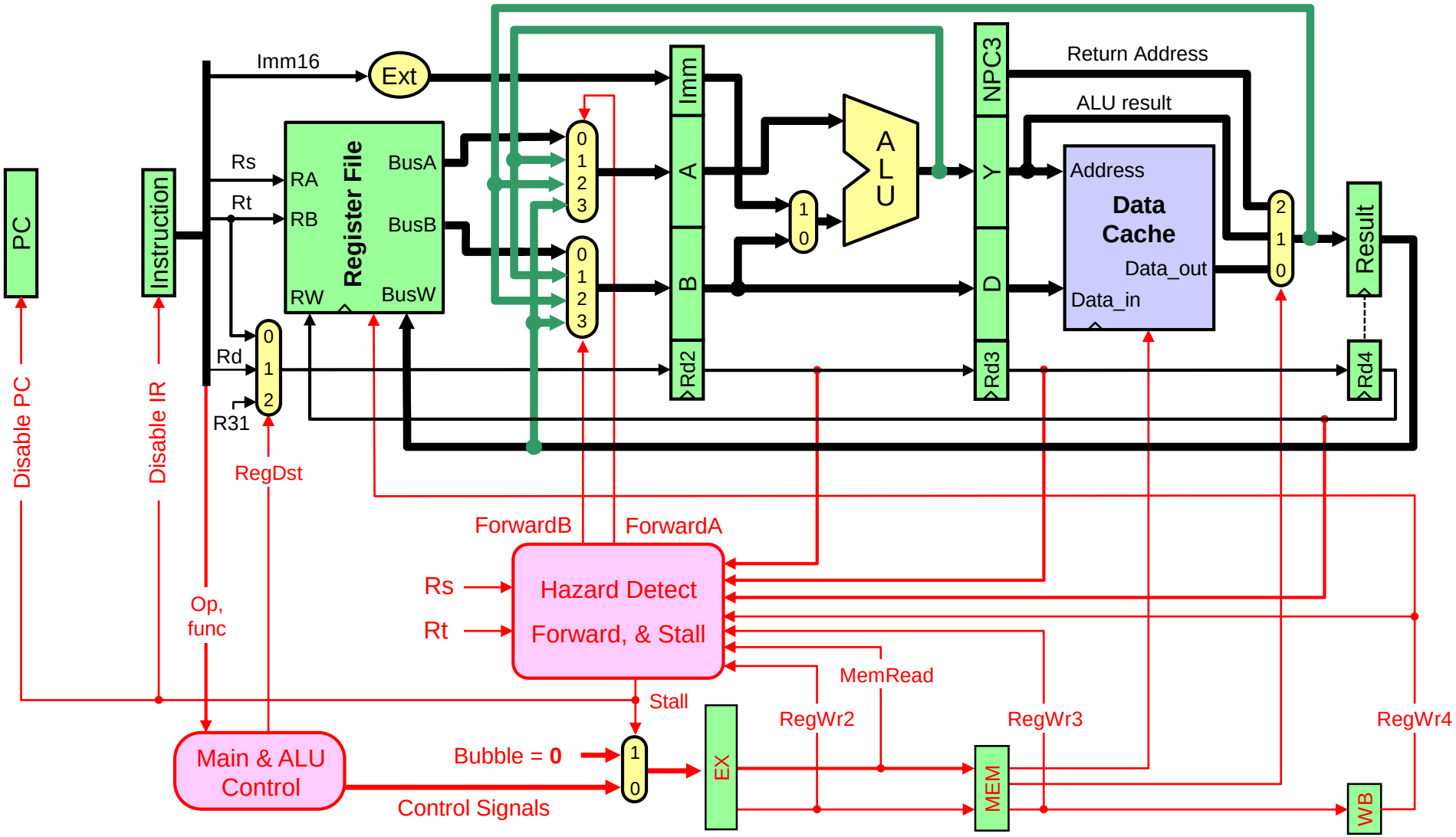


Stall Cycles

- Stall cycles are shown on a timing diagram
- Hazard is detected in the Decode stage
- Stall indicates that instruction is delayed (bubble inserted)
- Instruction fetching is also delayed after a stall
- Example:



Hazard Detect, Forward, and Stall



Compiler 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
```

- Original code: two stall cycles

```
ld   r10, 8(r16)   ; &B = 8+(r16)
ld   r11, 16(r16)  ; &C = 16+(r16)
add  r12, r10, r11 ; stall cycle
sd   r12, 0(r16)   ; &A = (r16)
ld   r13, 32(r16)  ; &E = 32+(r16)
ld   r14, 40(r16)  ; &F = 40+(r16)
sub  r15, r13, r14 ; stall cycle
sd   r15, 24(r16)  ; &D = 24+(r16)
```

- Faster code: No Stalls

```
ld   r10, 8(r16)
ld   r11, 16(r16)
ld   r13, 32(r16)
ld   r14, 40(r16)
add r12, r10, r11
sd   r12, 0(r16)
sub r15, r13, r14
sd   r15, 24(r16)
```

Name Dependence: Write After Read

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

```
I: sub r14, r11, r13  ; r11 is read

J: add r11, r12, r13  ; r11 is written
```

- Called Anti-Dependence: Re-use of register **r11**
- 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, **r15**)

Name Dependence: Write After Write

- Same destination register is written by two instructions

```
I: sub r11, r14, r13  ; r11 is written

J: add r11, r12, r13  ; r11 is written again
```

- Called Output Dependence: Re-use of register **r11**
- 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 Instruction **J** completes and writes **r11** before instruction **I**
- Output dependence can be eliminated by **renaming r11**
- Read After Read is NOT a name dependence

Next . . .

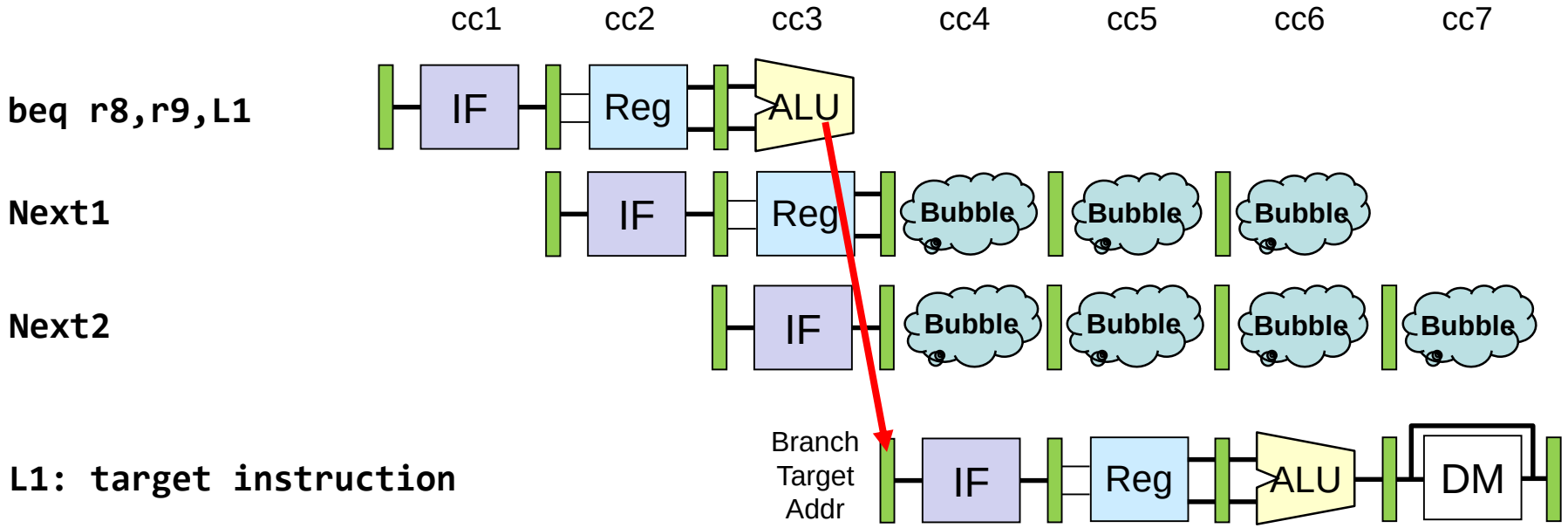
- Pipelining Basics
- 5-Stage Pipeline Microarchitecture
- Structural Hazards
- Data Hazards and Forwarding
- Load Delay and Pipeline Stall
- **Control Hazards and Branch Prediction**

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

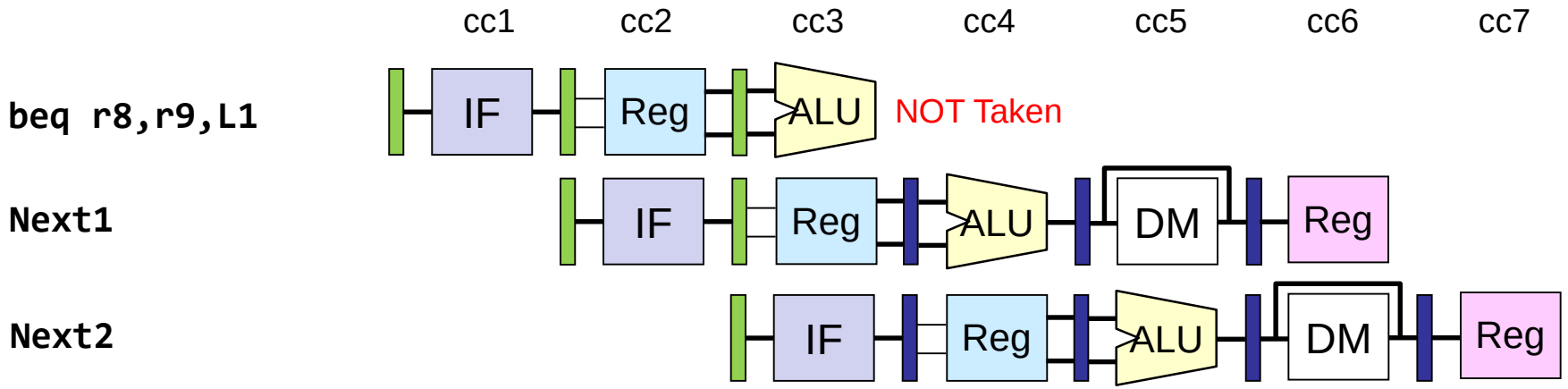
2-Cycle Branch Delay

- 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

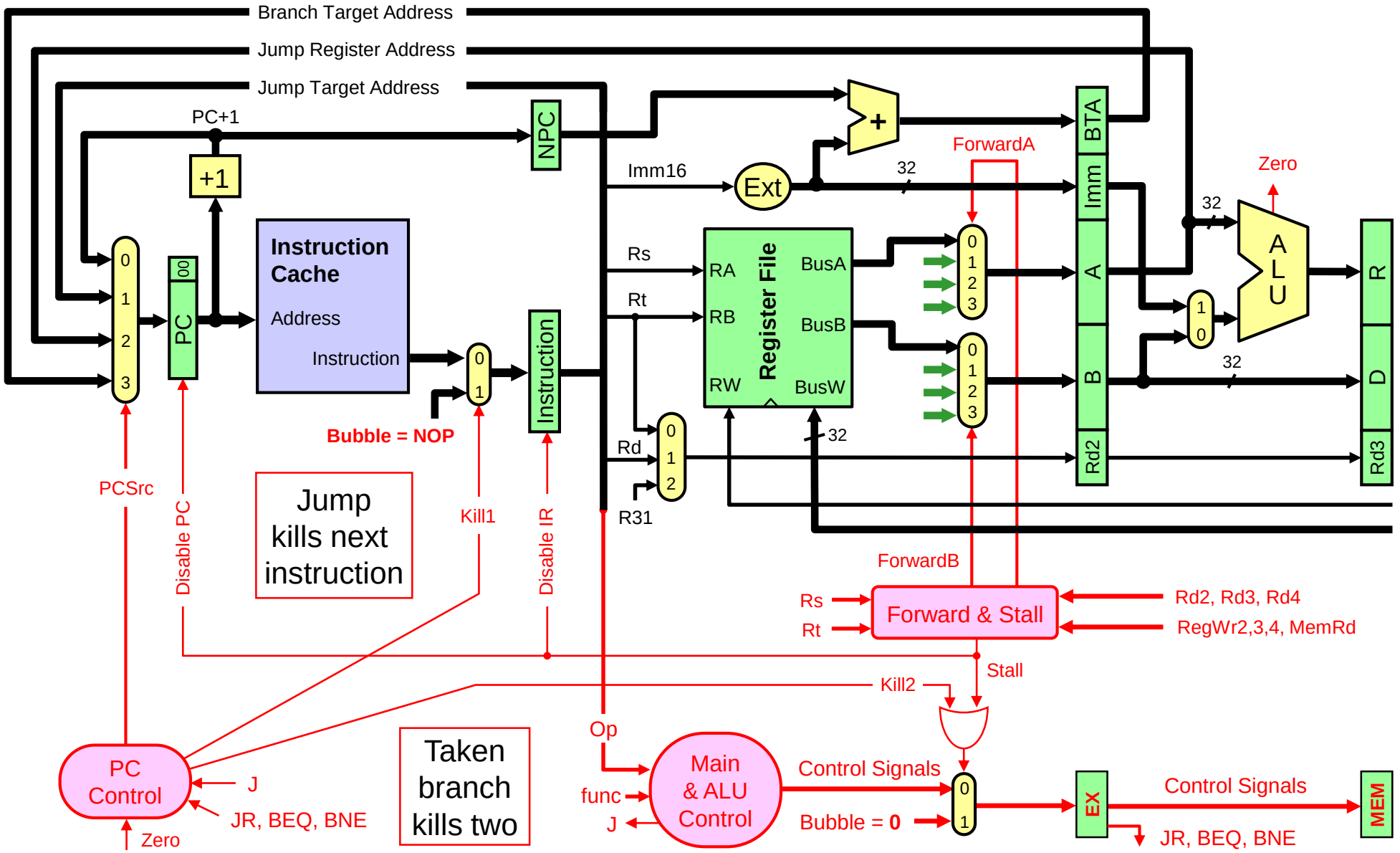


Predict Branch NOT Taken

- 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 clock cycles**



Pipelined Jump and Branch



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

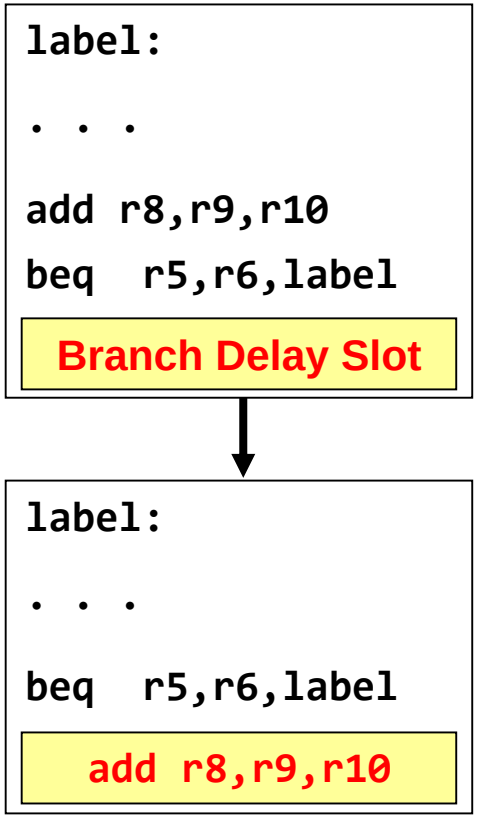
  = 2 × 0.2 × 0.9 = 0.36
- New CPI = 1 + 0.05 + 0.36 = 1.41

Branch Hazard Alternatives

- Predict Branch Not Taken (previously discussed)
  - Successor instruction is already fetched
  - Do NOT kill instruction after branch if 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 (only one slot)
- Dynamic Branch Prediction
  - Loop branches are taken most of time
  - How to predict the branch behavior at runtime?
  - Must reduce the branch delay to zero, but how?

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

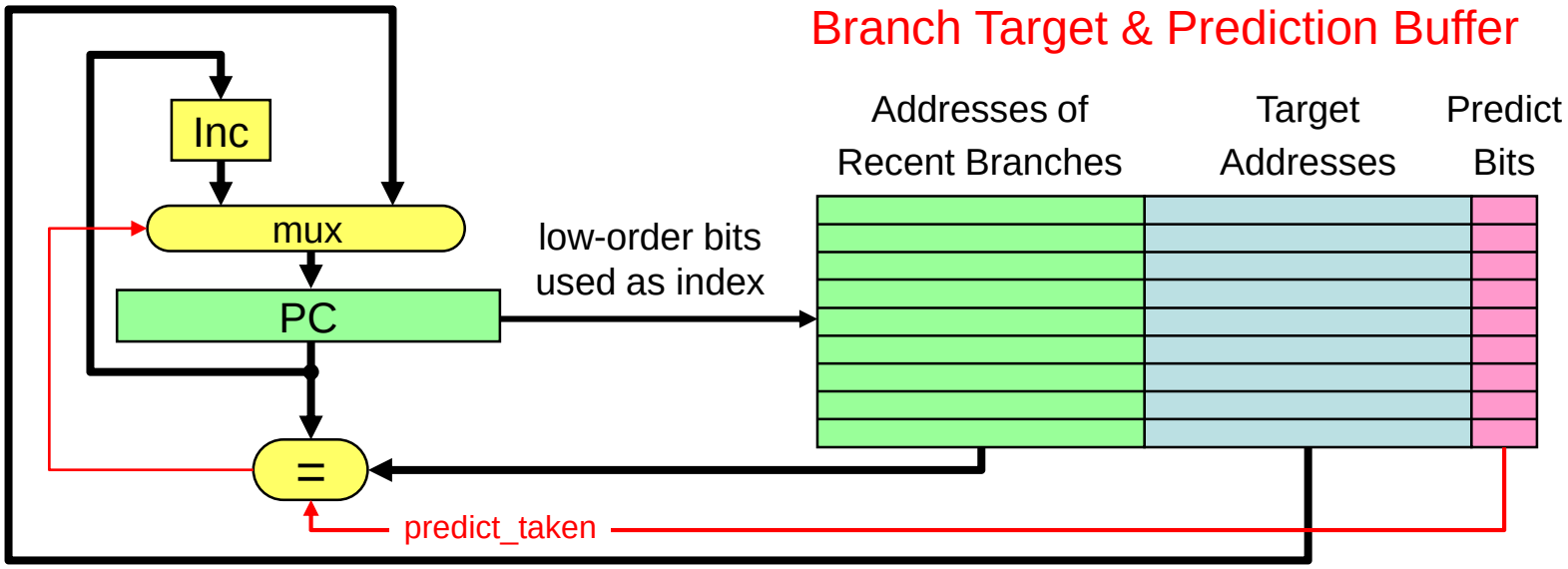


Drawback of Delayed Branching

- 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 deeper 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

Branch Target Buffer (IF Stage)

- The branch target buffer is implemented as a small cache
  - Stores the target addresses of recent branches and jumps
- We must also have prediction bits
  - To predict whether branches are taken or not taken
  - The prediction bits are determined by the hardware at runtime



Branch Target Buffer – cont'd

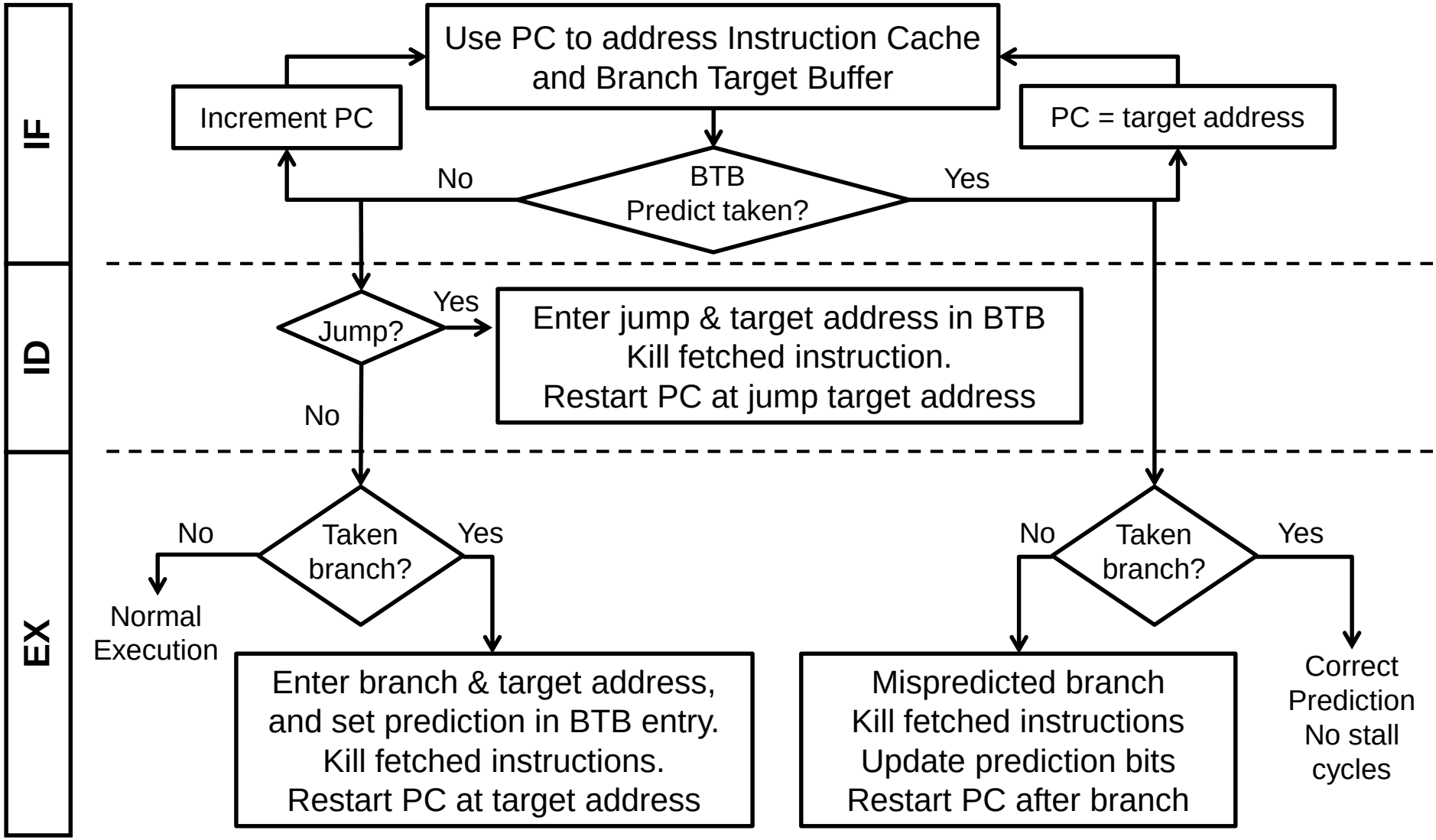
- Each Branch Target Buffer (BTB) entry stores:
  - Address of a recent jump or branch instruction
  - Target address of jump or branch
  - Prediction bits for a conditional branch (Taken or Not Taken)

    To predict jump/branch target address and branch outcome before instruction is decoded and branch outcome is computed
- Use the lower bits of the PC to index the BTB
  - Check if the PC matches an entry in the BTB (jump or branch)
  - If there is a match and the branch is predicted to be Taken then Update the PC using the target address stored in the BTB
- The BTB entries are updated by the hardware at runtime

Dynamic Branch Prediction

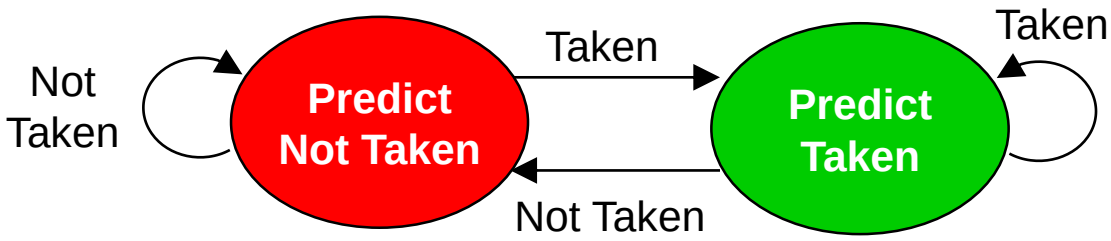
- Prediction of branches at runtime using prediction bits
- Prediction bits are associated with each entry in the BTB
  - Prediction bits reflect the recent history of a branch instruction
- Typically few prediction bits (1 or 2) are used per entry
- We don't know if the prediction is correct or not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

Dynamic Branch Prediction – Cont'd



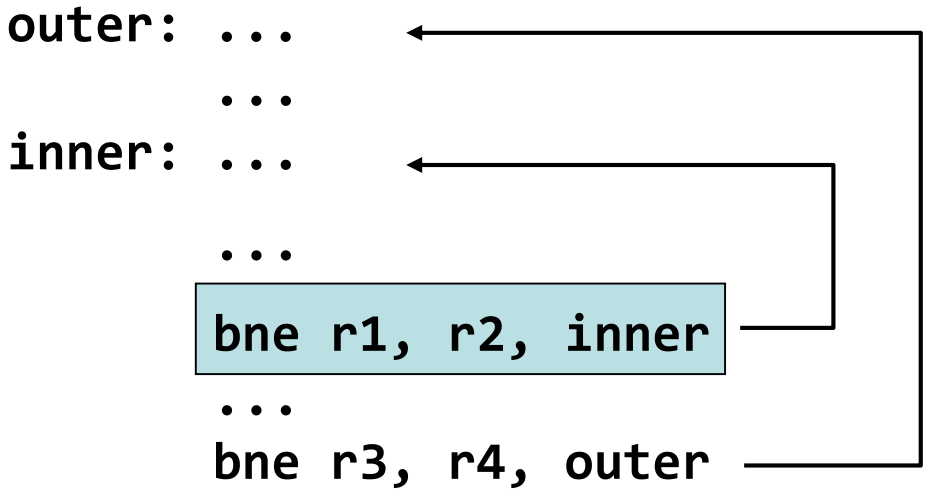
1-bit Prediction Scheme

- 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



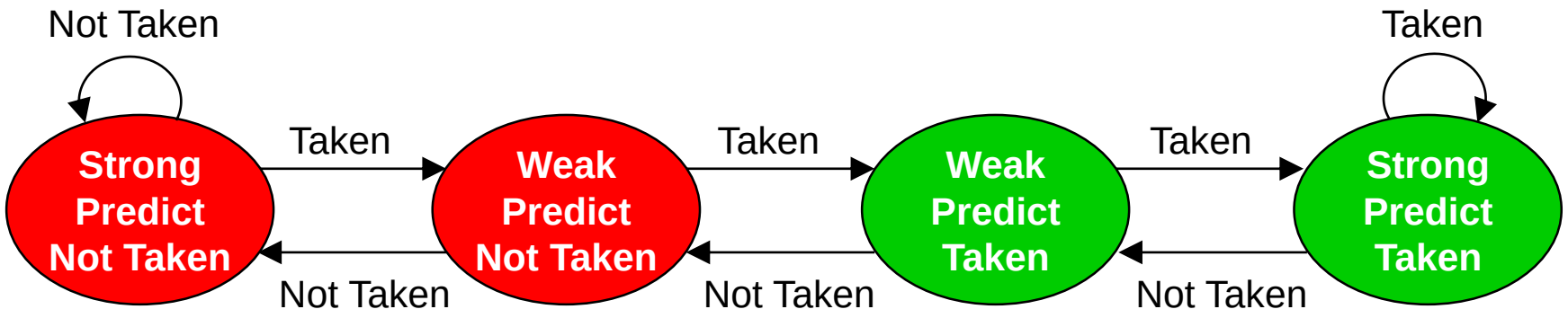
1-Bit Predictor: Shortcoming

- 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



2-bit Prediction Scheme

- 1-bit prediction scheme has a performance shortcoming
- 2-bit prediction scheme works better and is often used
  - 4 states: strong and weak predict taken / predict not taken
- Implemented as a saturating counter
  - Counter is incremented to max=3 when branch outcome is taken
  - Counter is decremented to min=0 when branch is not taken



Evaluating Branch Alternatives

Branch Scheme	Jump	Branch Not Taken	Branch Taken
Predict not taken	Penalty = 2 cycles	Penalty = 0 cycles	Penalty = 3 cycles
Delayed branch	Penalty = 1 cycle	Penalty = 0 cycles	Penalty = 2 cycles
BTB Prediction	Penalty = 2 cycles	Penalty = 3 cycles	Penalty = 3 cycles

- Assume: Jump = 3%, Branch-Not-Taken = 5%, Branch-Taken = 15%
- Assume a branch target buffer with hit rate = 90% for jump & branch
- Prediction accuracy for jump = 100%, for conditional branch = 80%
- What is the impact on the CPI? (Ideal CPI = 1 if no control hazards)

Branch Scheme	Jump = 3%	Branch NT = 5%	Branch Taken = 15%	CPI
Predict not taken	0.03 × 2	0	0.15 × 3 = 0.45	1+0.51
Delayed branch	0.03 × 1	0	0.15 × 2 = 0.30	1+0.33
BTB Prediction	0.03×0.1×2	0.05×0.9×0.2×3	0.15×(0.1+0.9×0.2)×3	1+0.16

In Summary

- Three types of pipeline hazards
  - Structural hazards: conflict 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 can be eliminated by forwarding
  - However, load delay cannot be eliminated and stalls the pipeline
  - Delayed branching reduces branch delay ➔ compiler support
  - BTB with branch prediction can reduce branch delay to zero
  - Branch mis-prediction should kill the wrongly fetched instructions