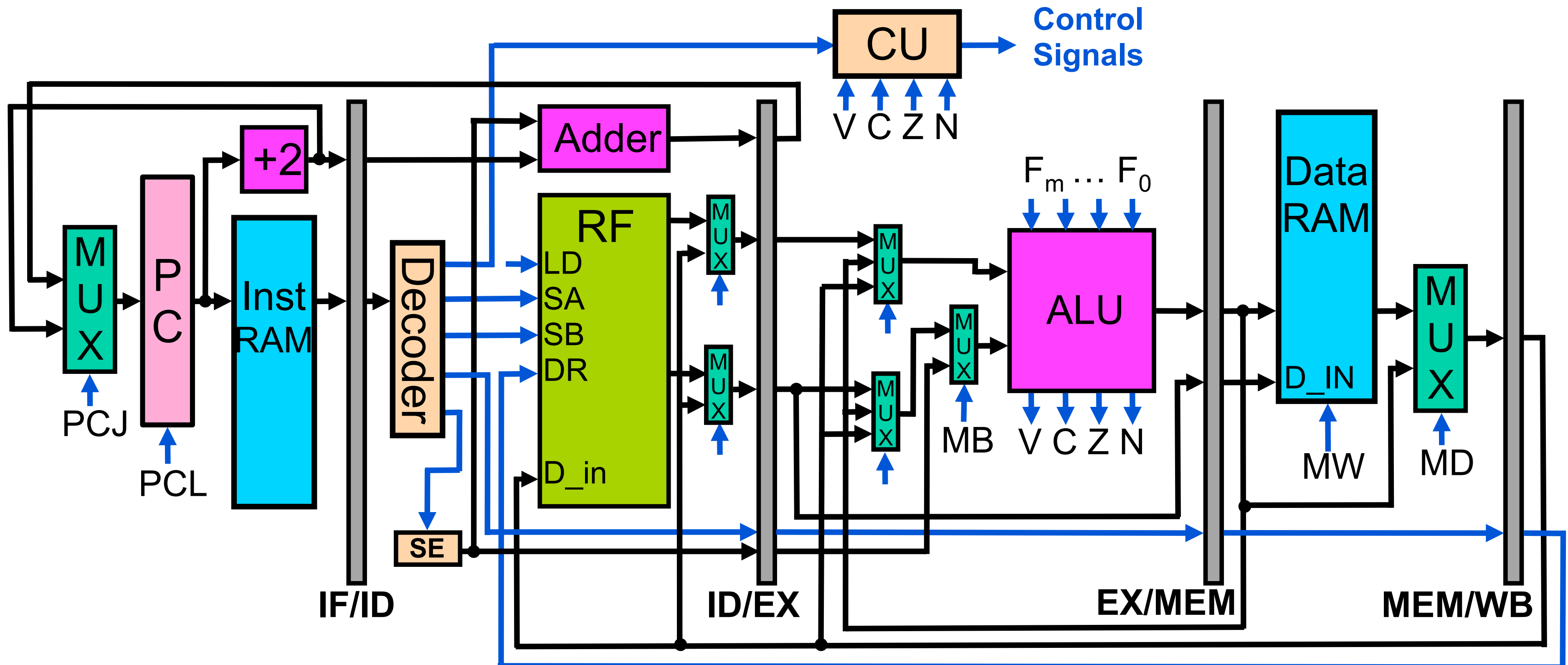
ECE 2300
Digital Logic & Computer Organization

Fall 2016

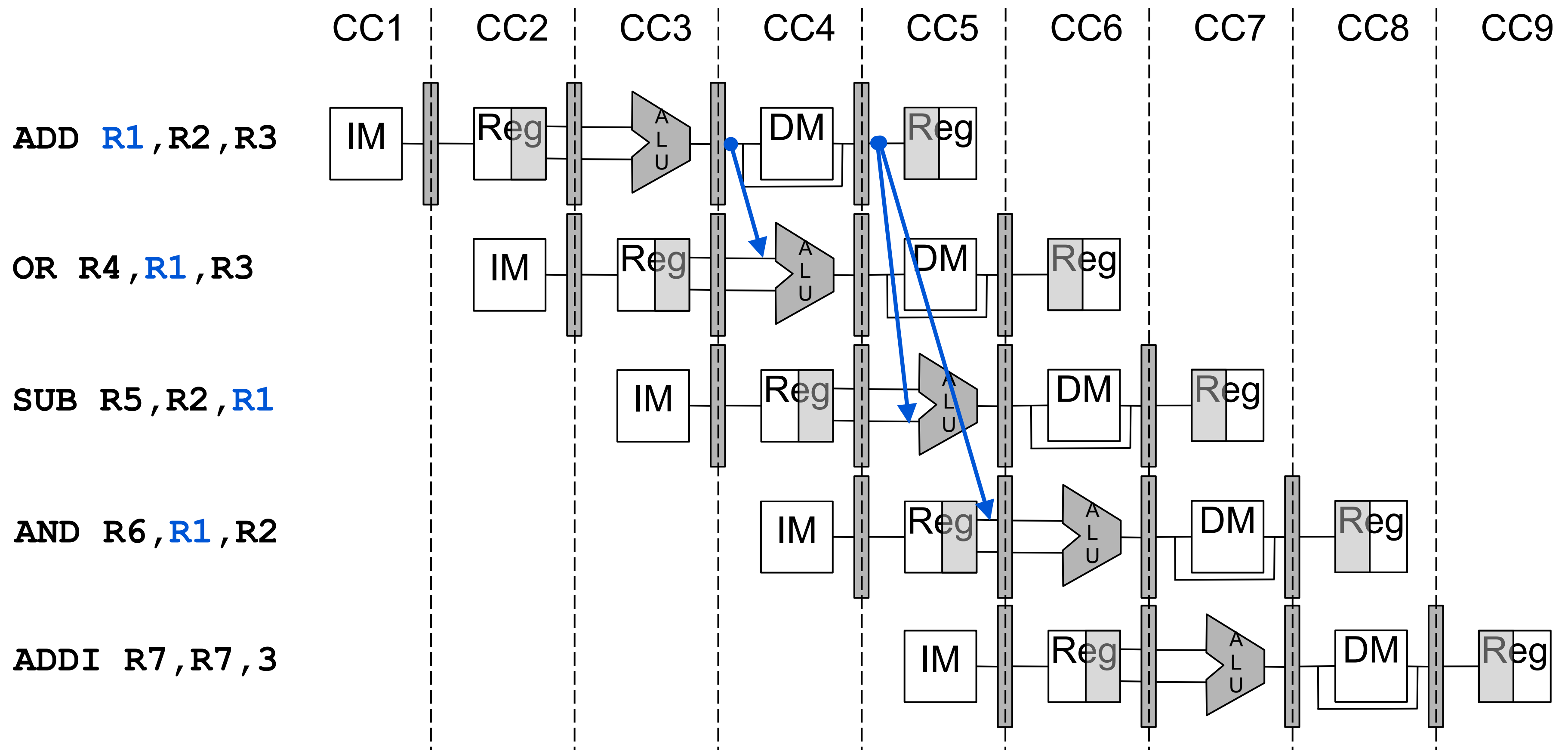
More Pipelined Microprocessor

Cornell University

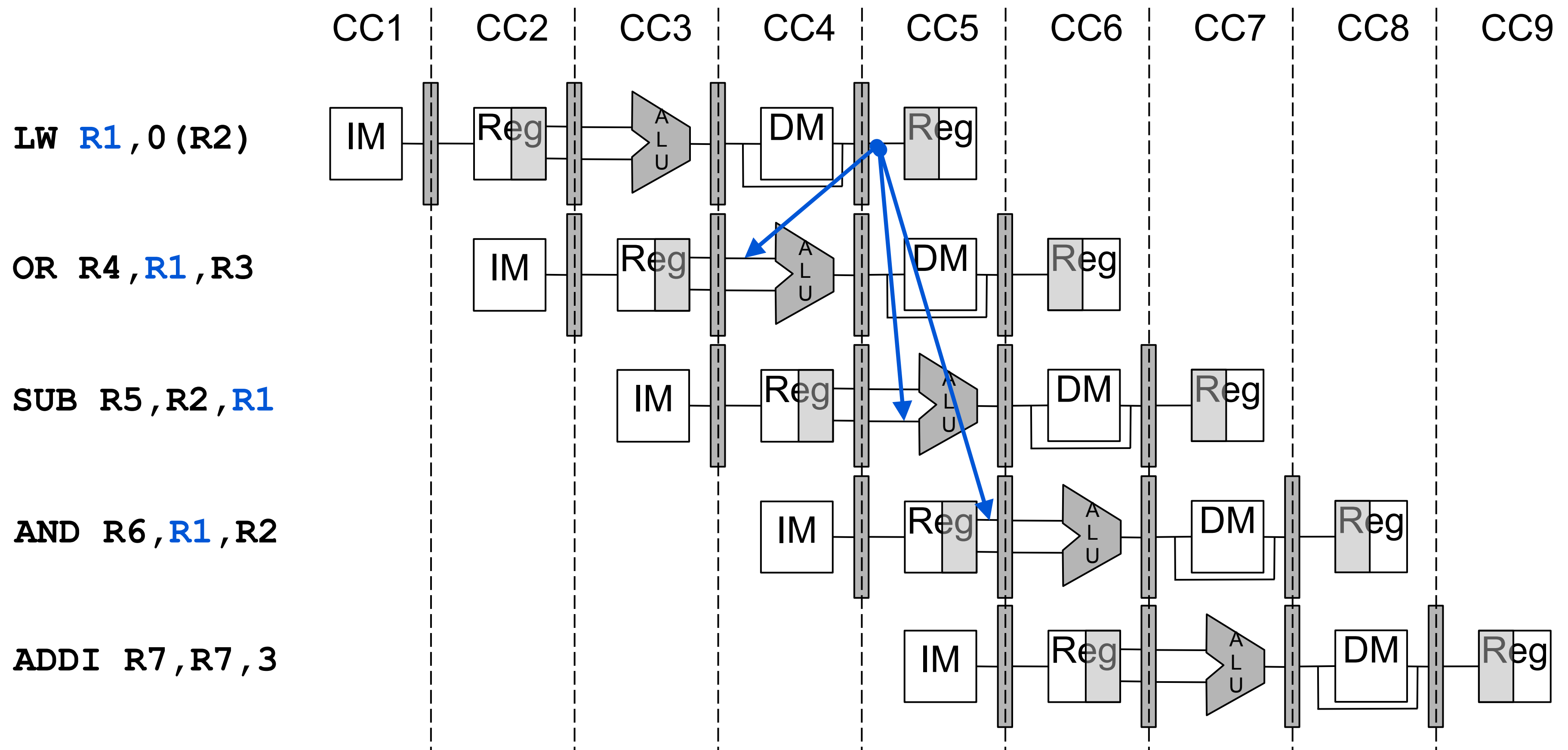
Pipelined Processor with Forwarding



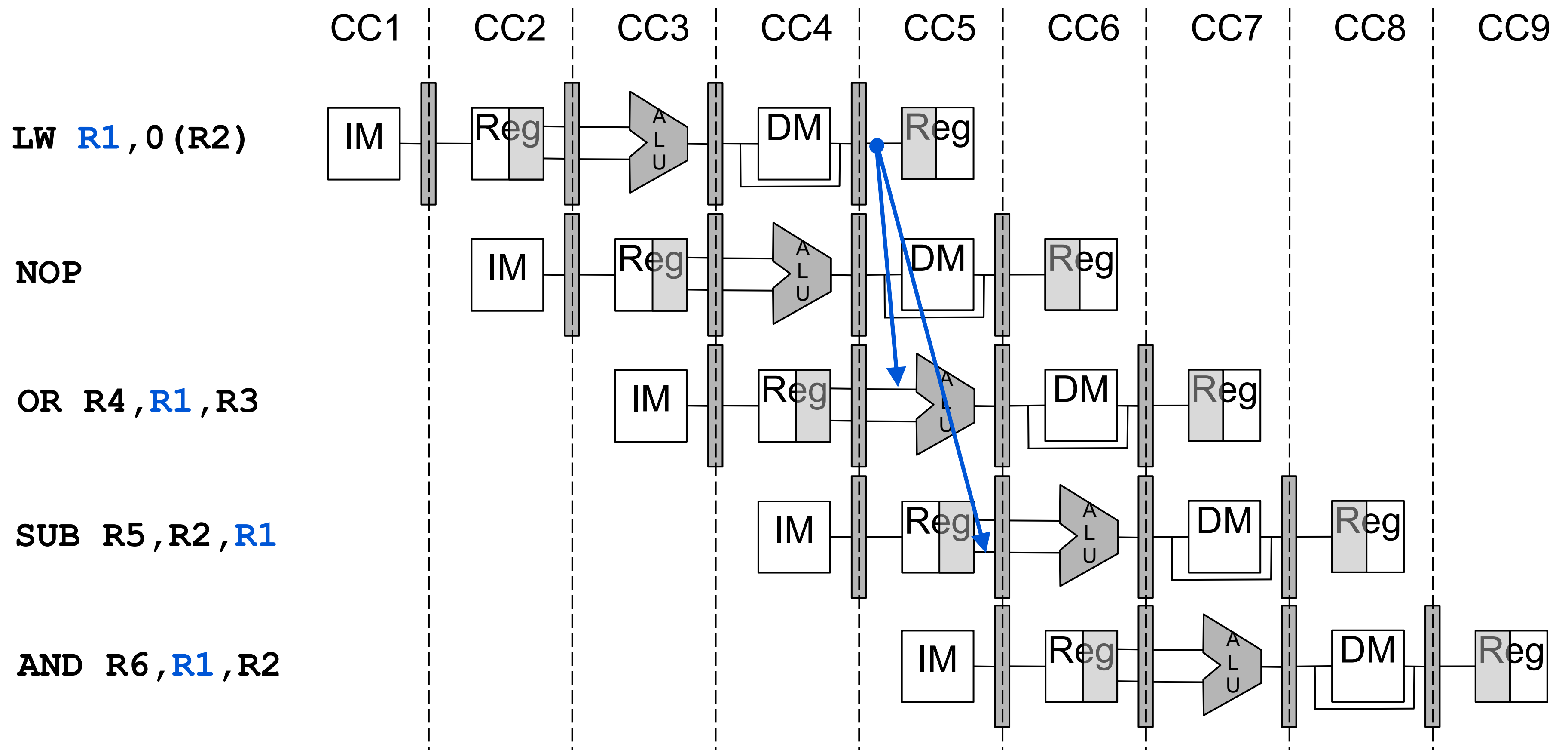
HW Forwarding (Bypassing)



Load Instructions and Forwarding



Solution 1: SW Inserts NOP Instruction



Solution 2: Delay Slots

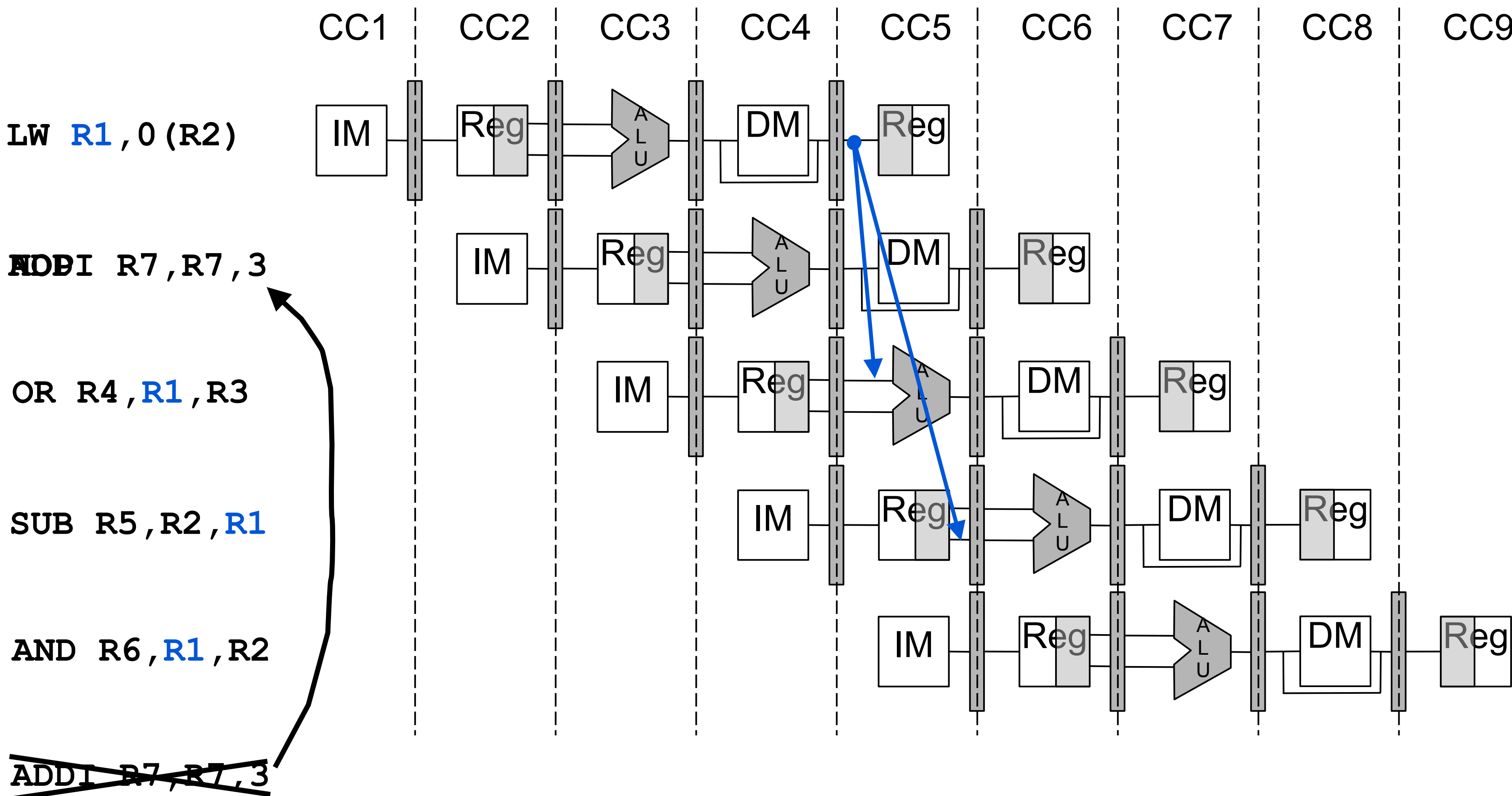
- A *delay slot* is a location in the program where the compiler is required to insert an instruction between dependent instructions

- Delay slots are defined by the ISA

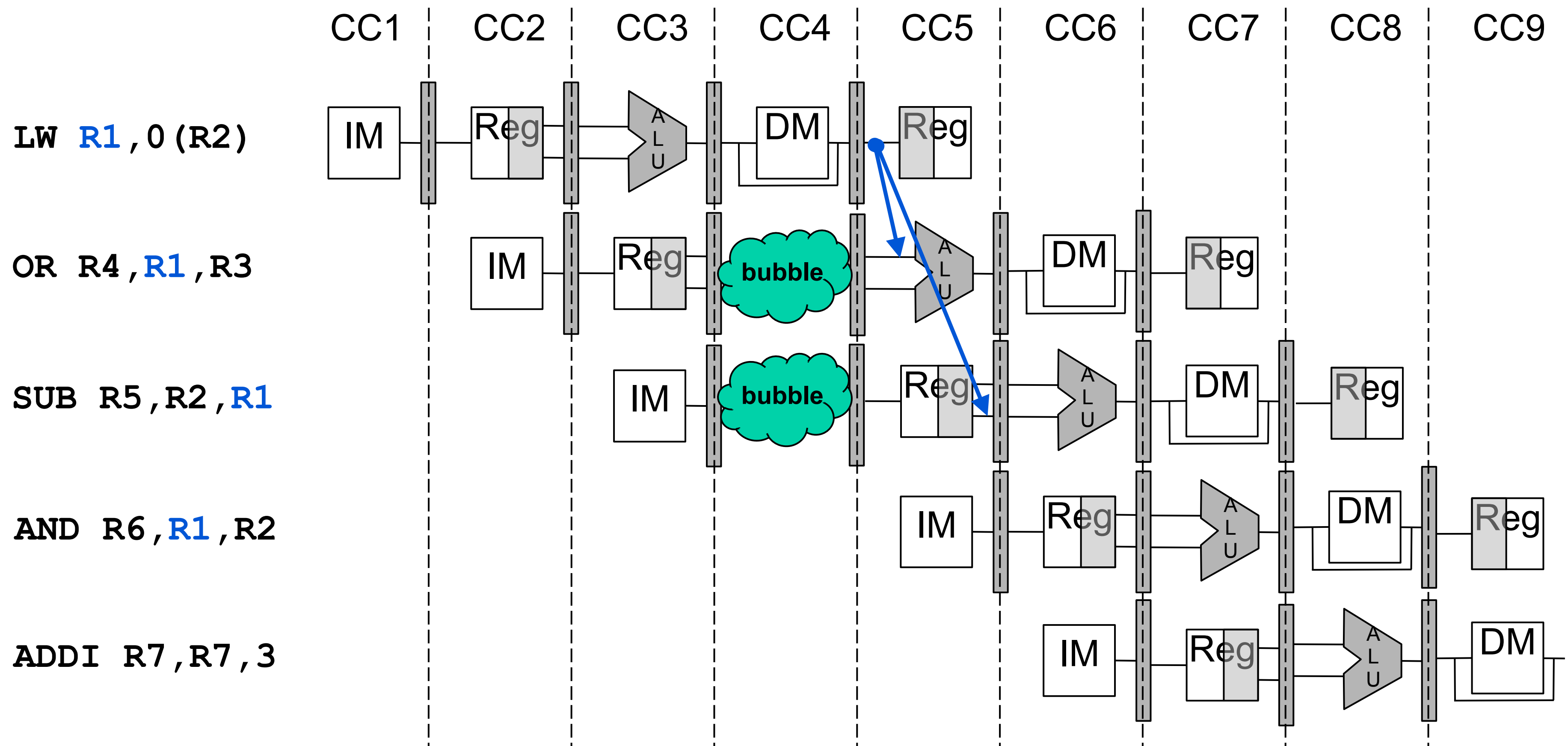
- In the worst case the compiler will fill a delay slot with a NOP

- The compiler will try to move a non-dependent instruction from elsewhere in the program into the delay slot
  - Doing so must not change the function of the program

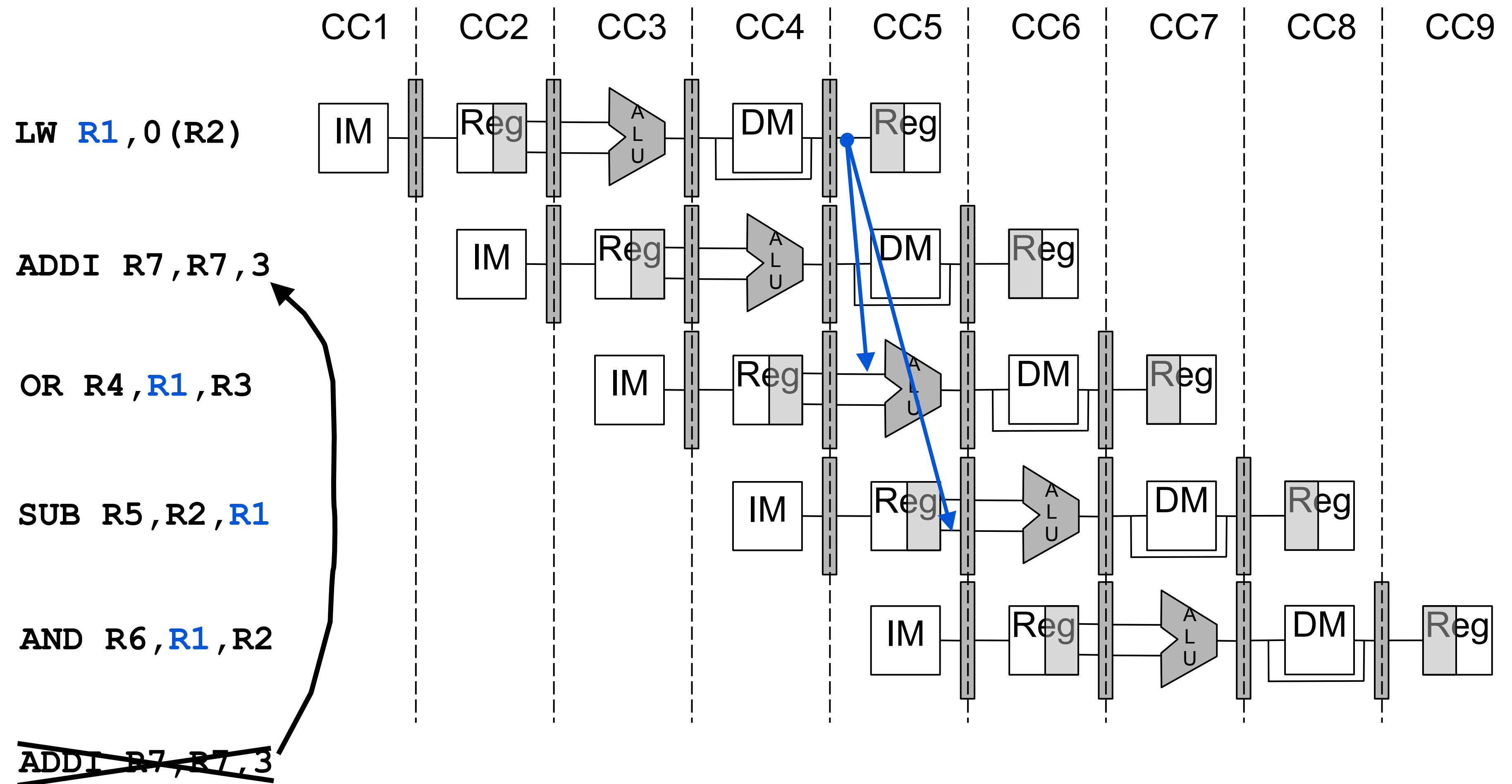
Filling the Load Delay Slot



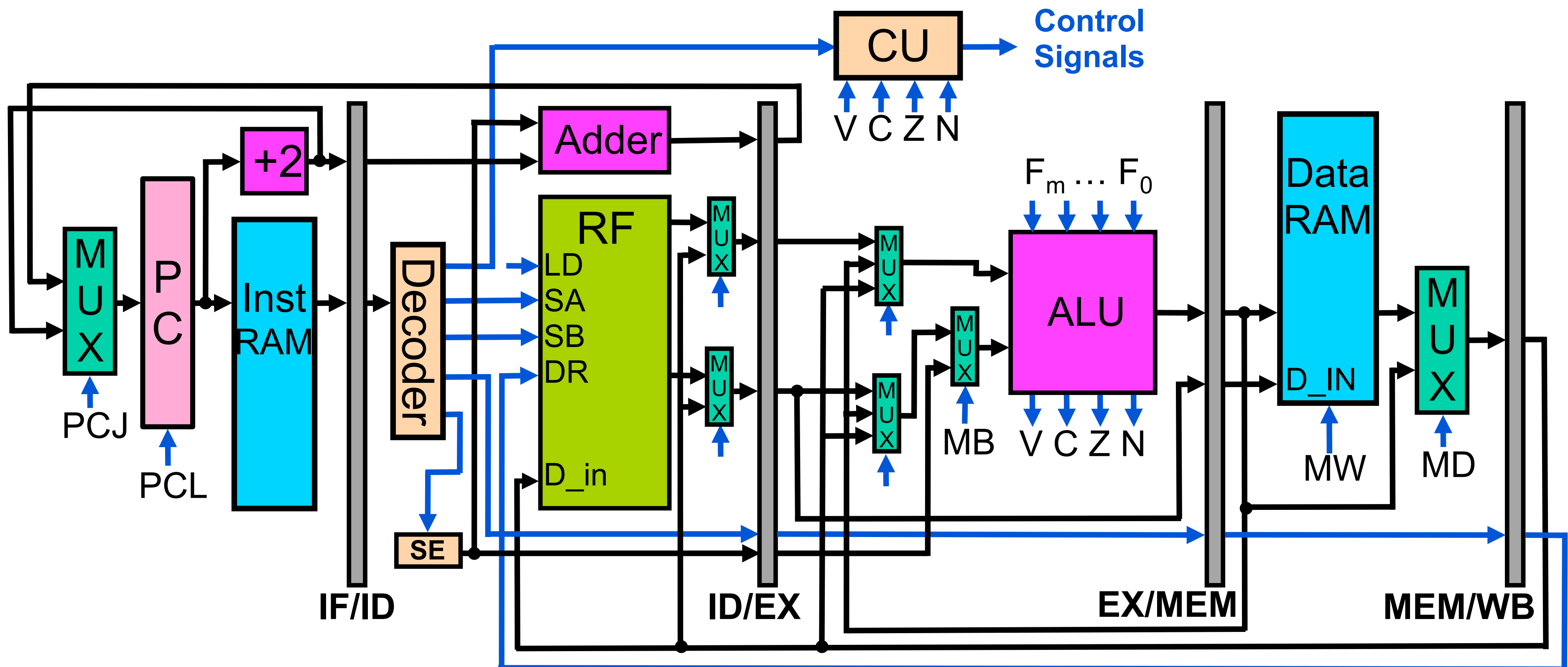
Solution 3: HW Stalls the Pipeline



Avoiding Stalls by Instruction Reordering

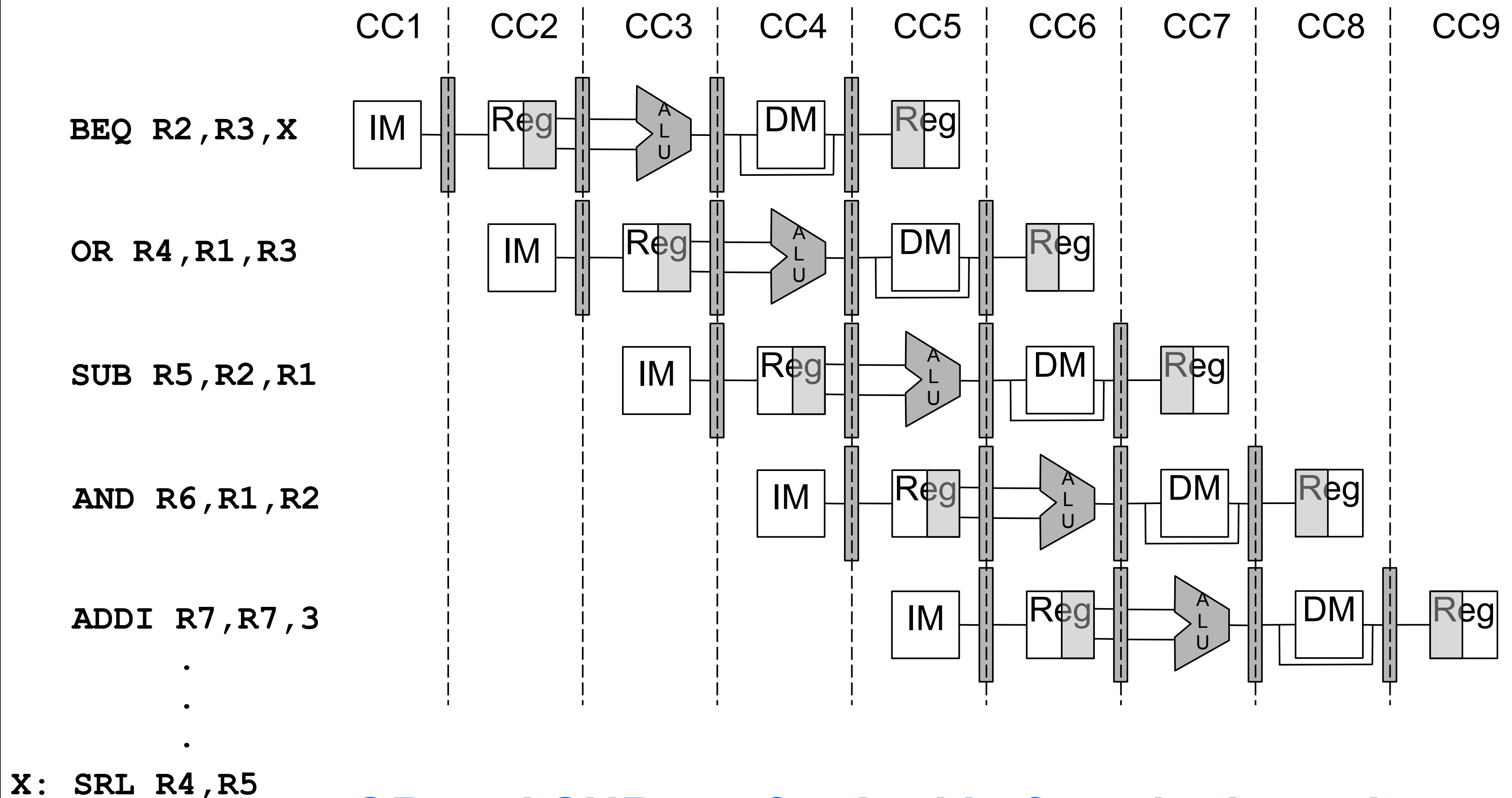


The Problem with Branches



If the condition is met, the PC is updated at the end of EX, after we've already fetched the next two instructions

The Problem with Branches

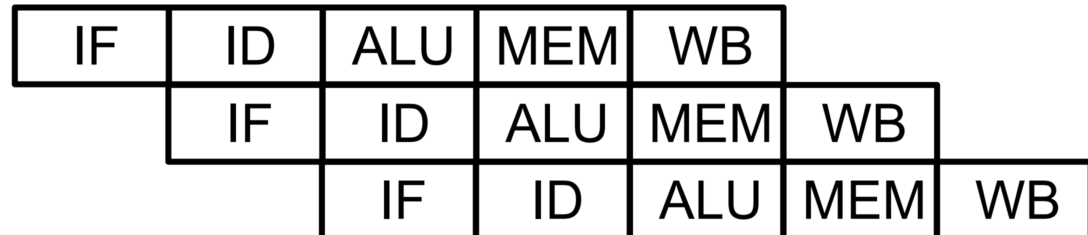


OR and SUB are fetched before the branch condition is evaluated

Control Hazard

- Occurs when instructions following a branch are fetched before the branch outcome is known

BEQ R2, R3, X
OR R4, R1, R3
SUB R5, R2, R1



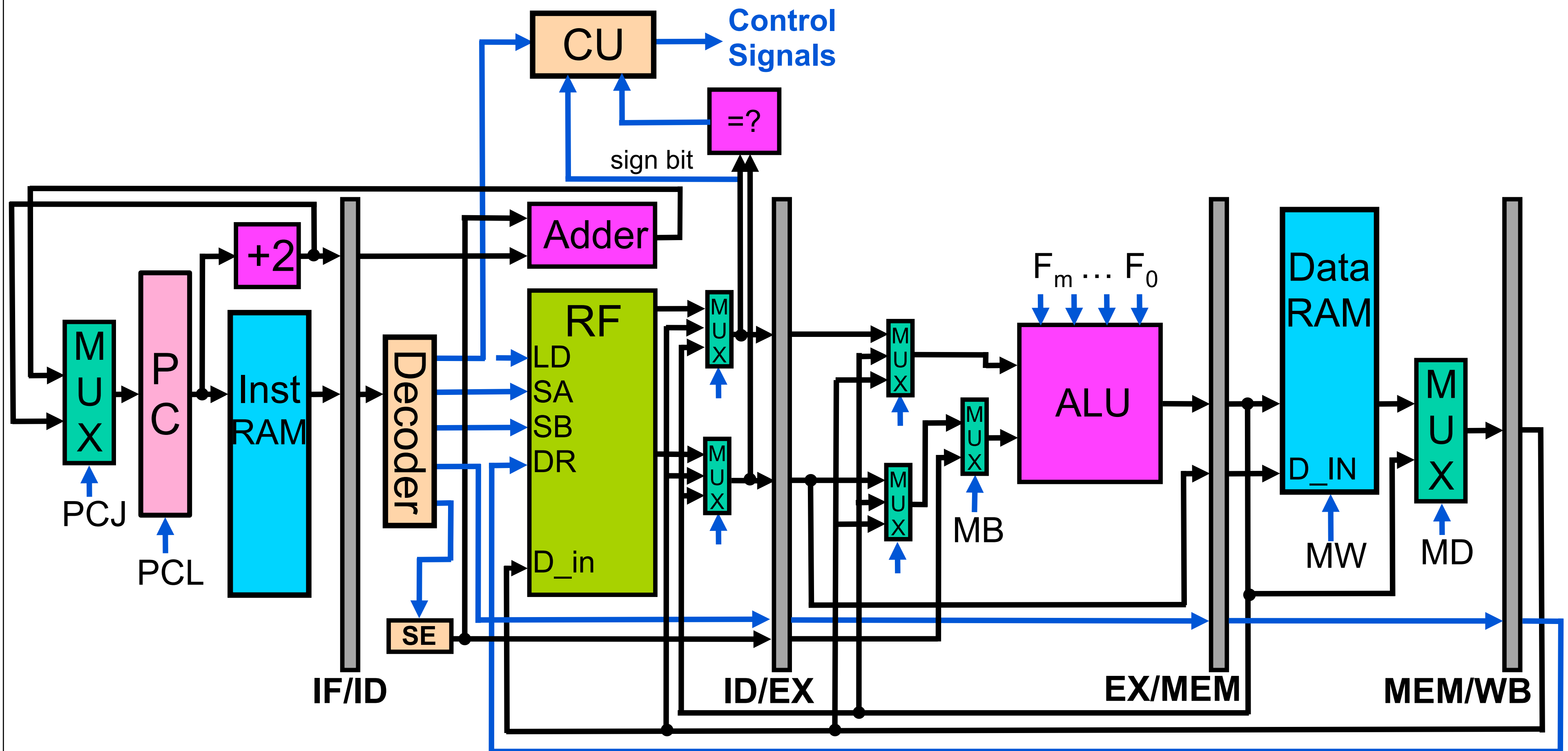
X: SRL R4, R5

- What should happen
  - If branch is not taken, next fetched instruction should be at address PC+2 (OR)
  - If branch is taken, next fetched instruction should be at address X (SRL)
- What actually happens
  - Instructions at PC+2 and PC+4 are fetched before branch outcome is known

Pipeline Modifications

- We already calculate the branch target in ID
- Put dedicated hardware to also evaluate the condition in ID
- Now only 1 instruction will follow the branch

Pipeline Modifications



Instruction	Format	OP	Operation
BEQ rt,rs,*target*	I	1000	if(R[rs] == R[rt]) PC = PC + sext({imm,1’b0})
BNE rt,rs,*target*	I	1001	if(R[rs] ≠ R[rt]) PC = PC + sext({imm,1’b0})
BGEZ rs,*target*	I	1010	if(R[rs] >= 0) PC = PC + sext({imm,1’b0})
BLTZ rs,*target*	I	1011	if(R[rs] < 0) PC = PC + sext({imm,1’b0})

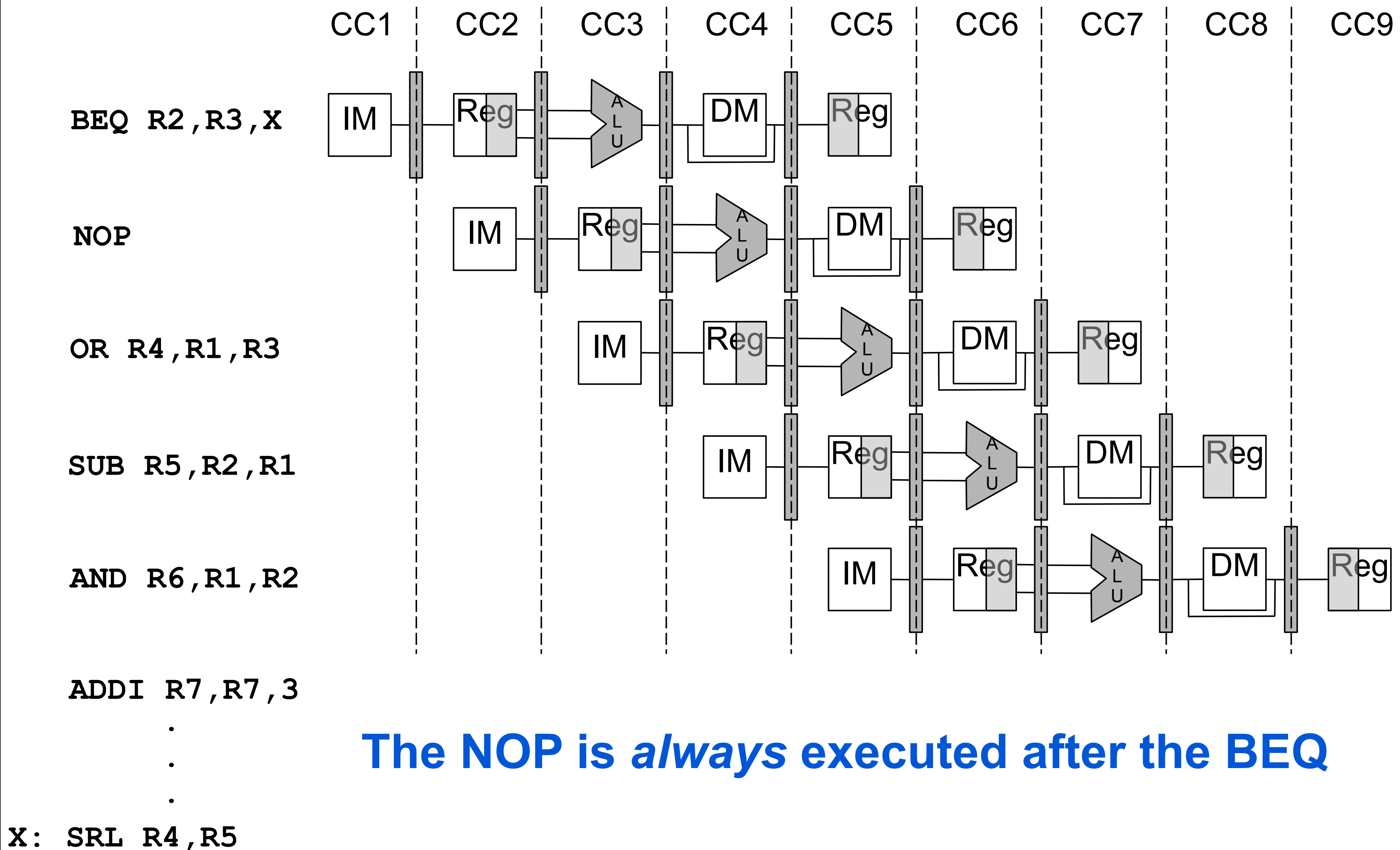
Branch Delay Slot

- **If the ISA defines a branch delay slot, the instruction immediately following a branch is *always* executed after the branch**

- **The compiler finds an instruction to put there, or puts in a NOP**

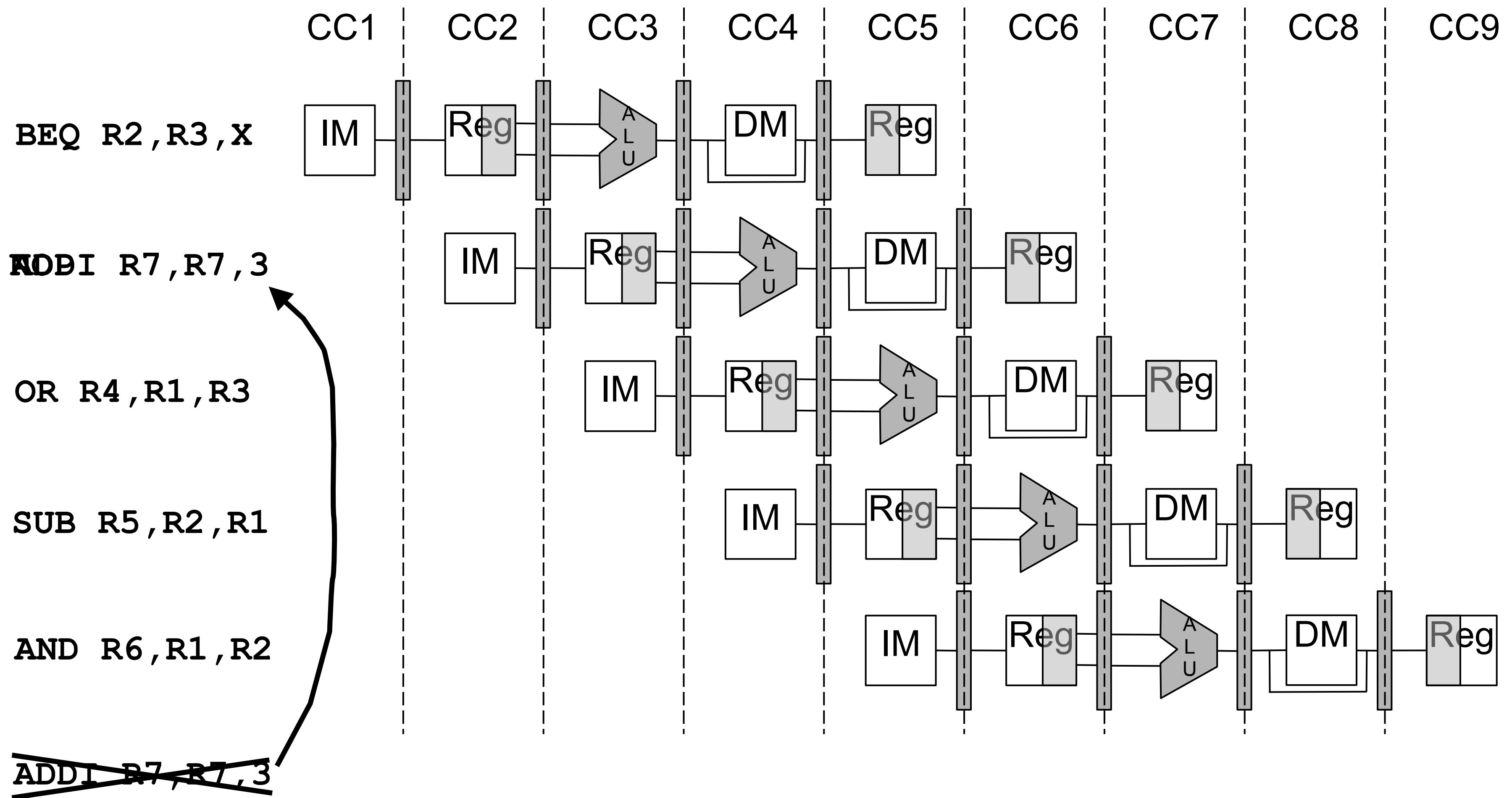
- **The hardware *must* execute the instruction immediately following the branch, regardless of whether the branch is taken or not**

Filling the Branch Delay Slot with a NOP



The NOP is *always* executed after the BEQ

Eliminating the Branch Delay



.
.
.

X: SRL R4,R5

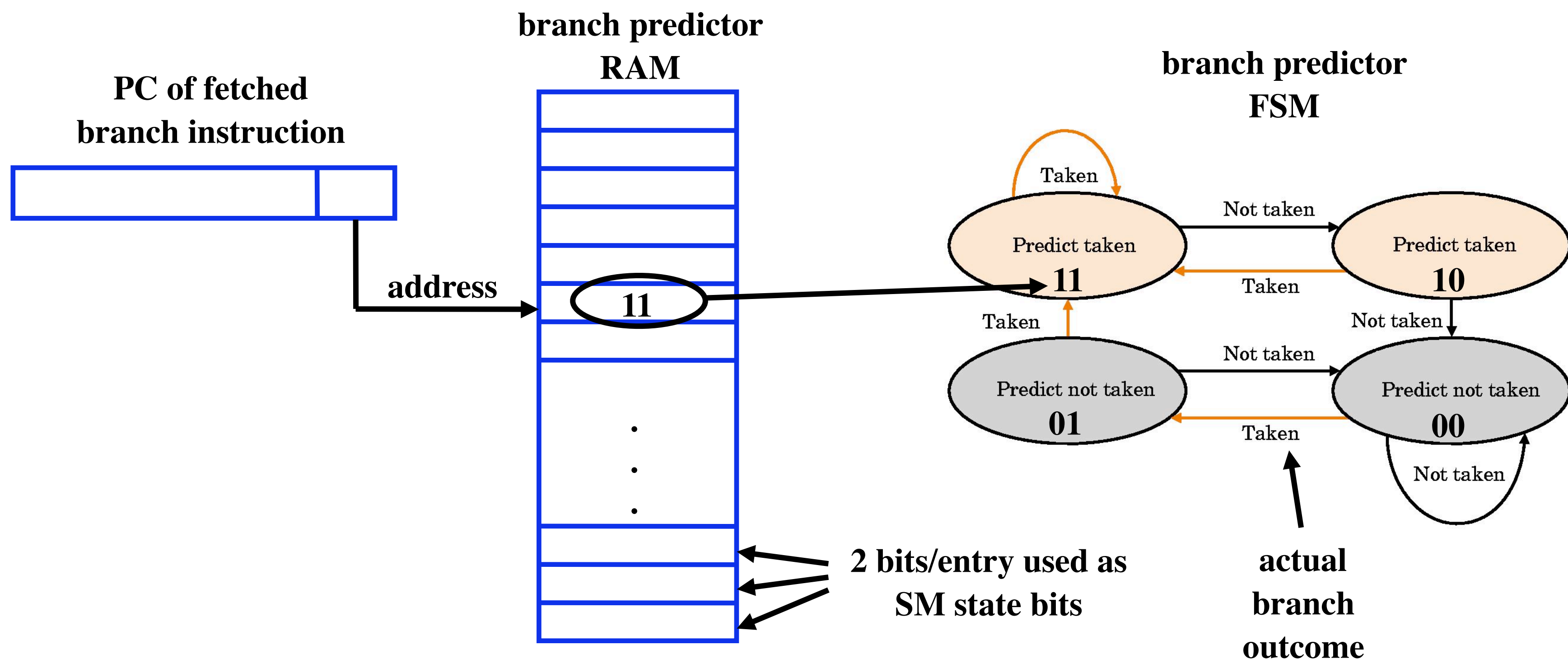
The ADDI is *always* executed after the BEQ. If the BEQ is taken, executing the ADDI must not cause incorrect behavior.

Dynamic Branch Prediction

- **When fetching the branch instruction, *predict* the branch outcome and target**
- **Fetch instructions from the predicted PC**
- **After executing the branch, verify whether the prediction was correct**
  - **If so, continue without any performance penalty**
  - **If not, undo what was done and fetch from the other direction**

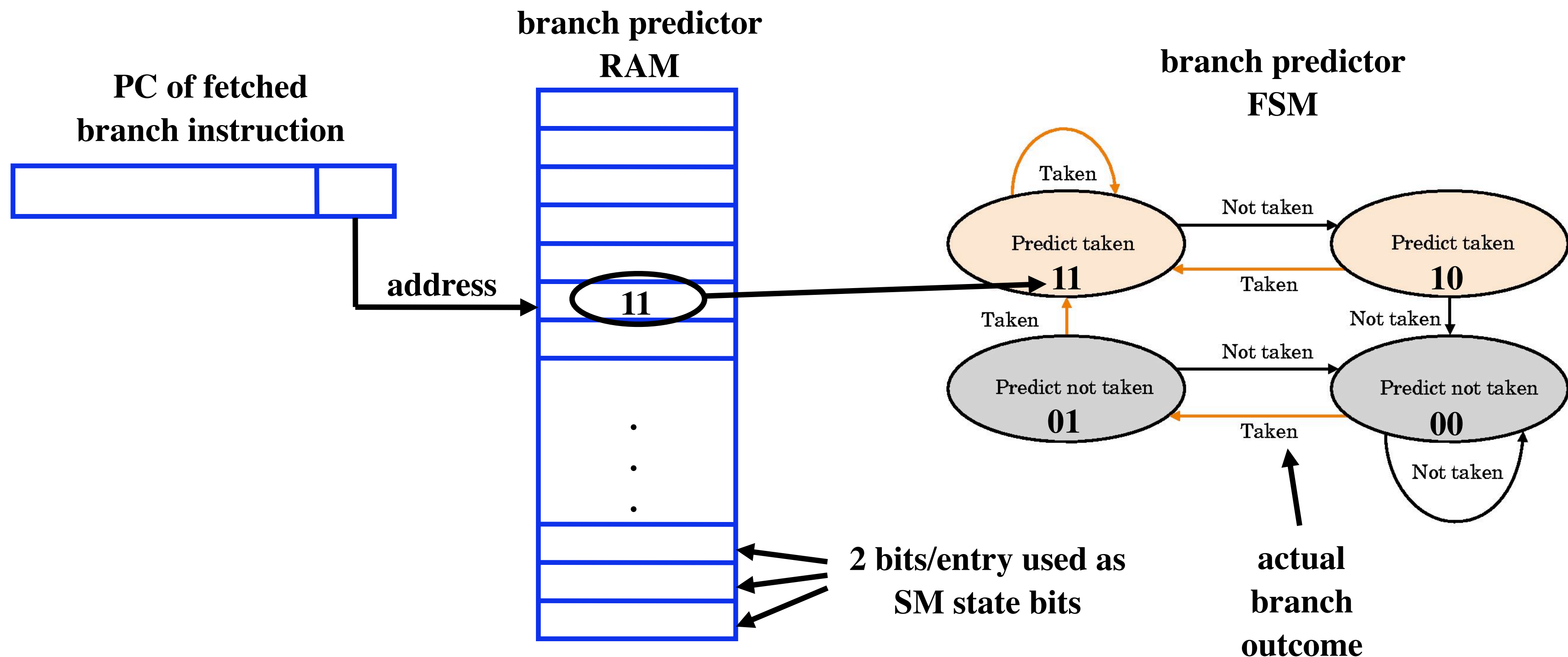
Bimodal Branch Outcome Predictor

- n location RAM + a 4 state FSM
- Low-order $\log_2 n$ PC bits of a fetched branch instruction used as the RAM address
- RAM holds state bits for n identical FSMs



Bimodal Branch Outcome Predictor

- When a branch is fetched, the RAM is read
- The prediction is based on the MSB (left bit)
- RAM state bits are updated when the branch outcome (taken / not taken) is determined in ID



Branch Target Buffer

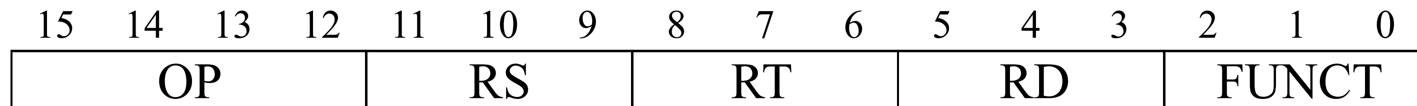
- Predicts the branch target address
- Small memory (typically 256-512 entries) addressed by the low-order branch PC bits
- Each entry holds the last target address of the branch
- When a branch is fetched, the BTB is accessed to get the target address

Pipeline Control Requirements

- Generate control signals for each stage
  - IF: PCJ
  - EX: MB, F
  - MEM: MW, MD
  - WB: LD
- Detect data hazards and insert pipeline bubbles
  - We'll assume no load delay slot
- Detect forwarding conditions and generate MUX control signals
- Assume branch delay slot defined in ISA

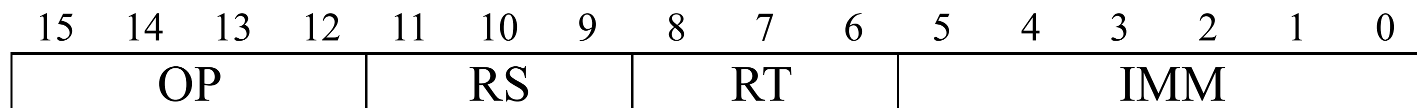
Decoding the Instruction in ID

Register to Register Format

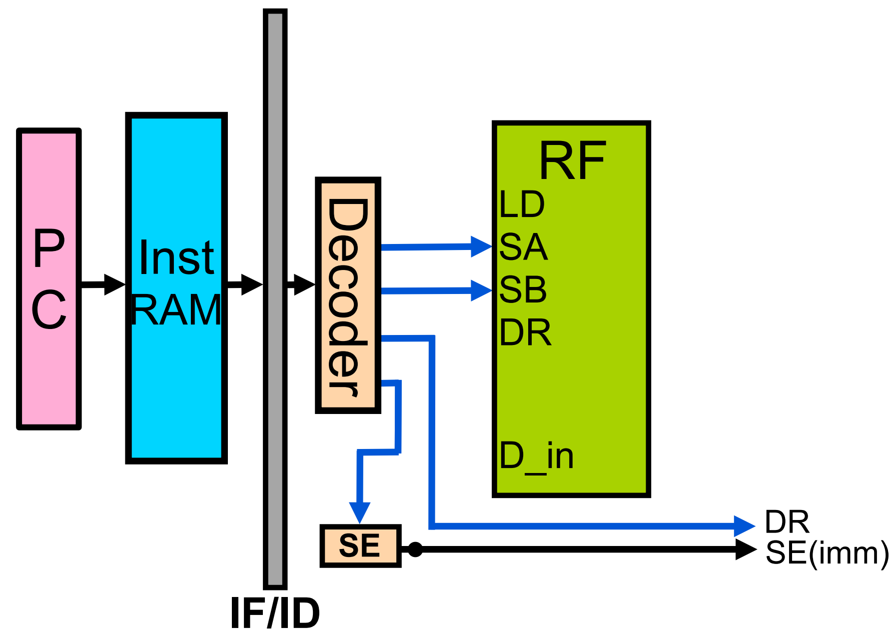


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP				RS			RT			RD			FUNCT		

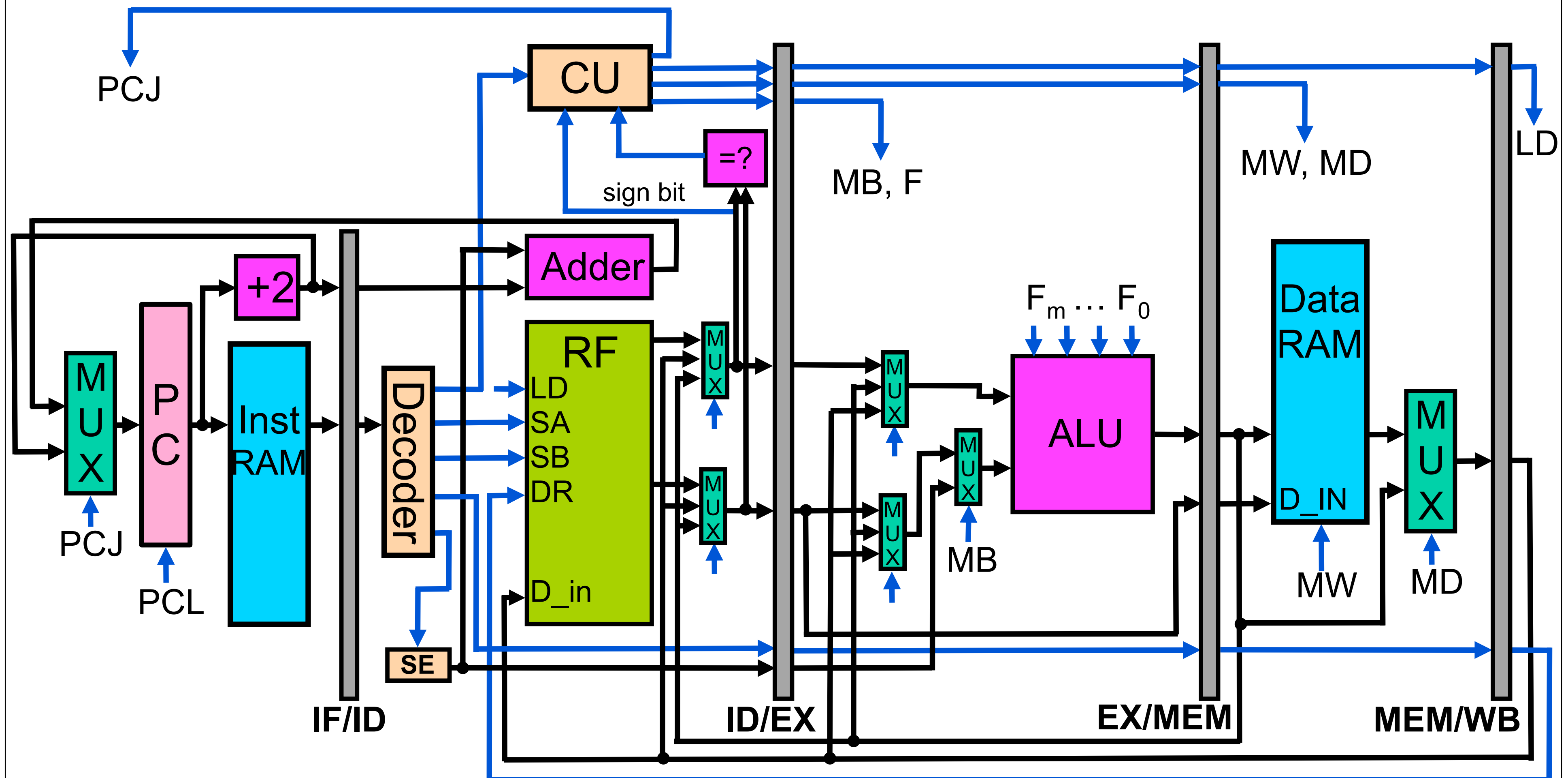
Immediate Format



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP				RS			RT			IMM					



Generating Control for Each Stage



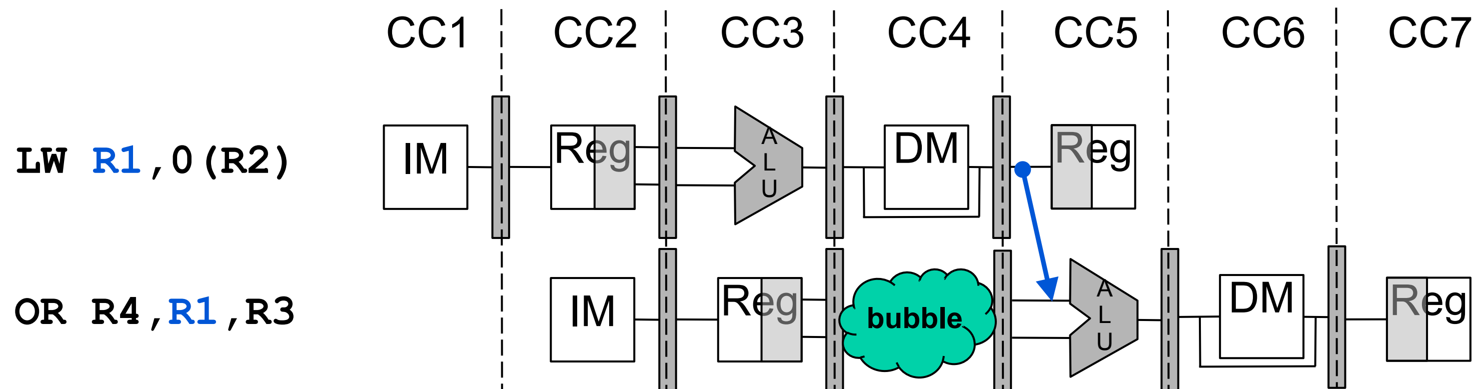
Data Hazards Requiring Bubbles

- Occur when instructions are too close together for forwarding to work

- Requires adding bubbles in the pipeline

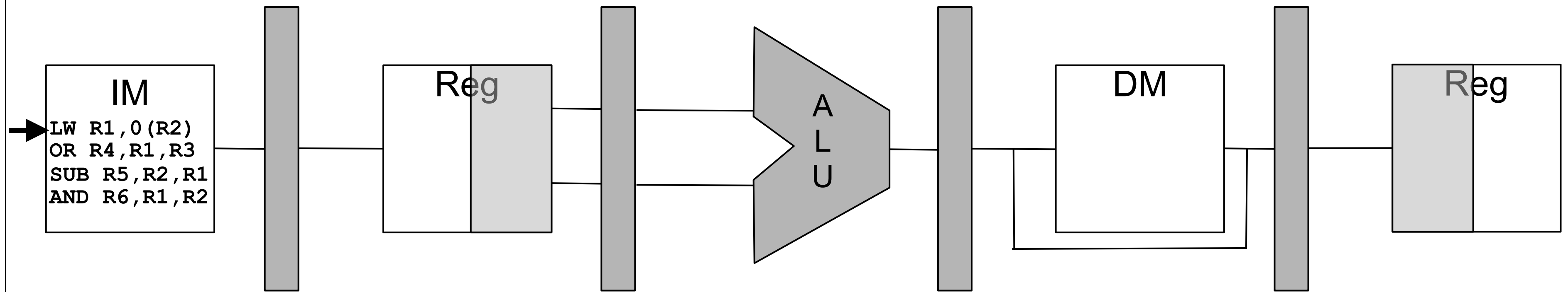
- Data hazard conditions to detect and handle
  - Load followed by R-type
  - Load followed by I-type ALU instruction
  - Load followed by Load
  - Load followed by Store (two cases)
  - Load followed by Branch
  - ALU instruction followed by Branch

Load Followed by R-type Instruction



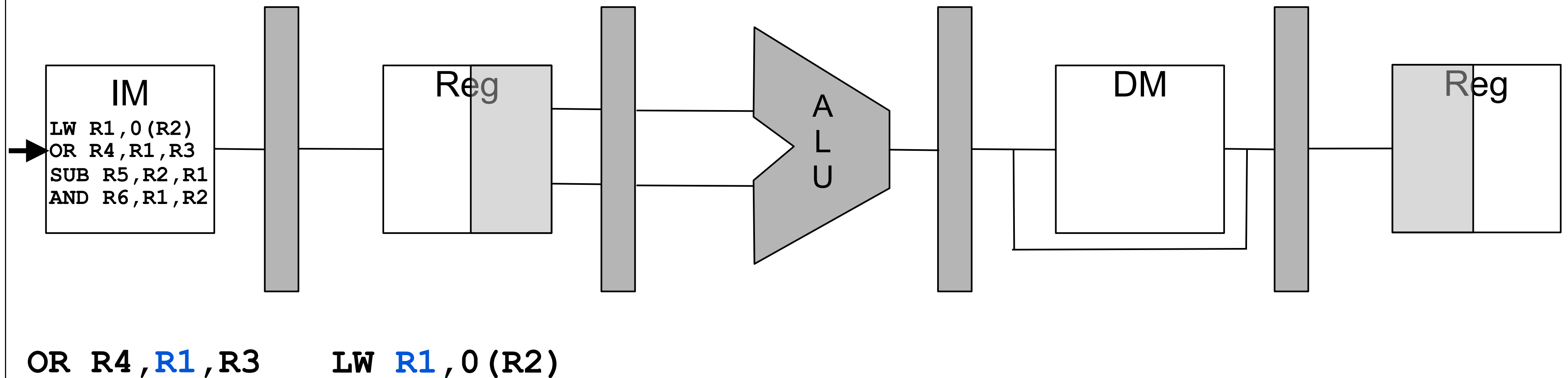

```
if (EX.Load && ID.R-type) {                 // Inst in EX is a Load and inst in ID is R-type
    if (EX.DR == (ID.SA || ID.SB)) {        // DR of Load matches SA or SB of inst in ID
        Force NOP into EX in next cycle     // Insert bubble
        Don't Load IF/ID in next cycle      // Hold instruction in ID for a cycle
        Don't Load PC in next cycle         // Hold instruction in IF for a cycle
    }
}
```

Load Followed by R-type Instruction



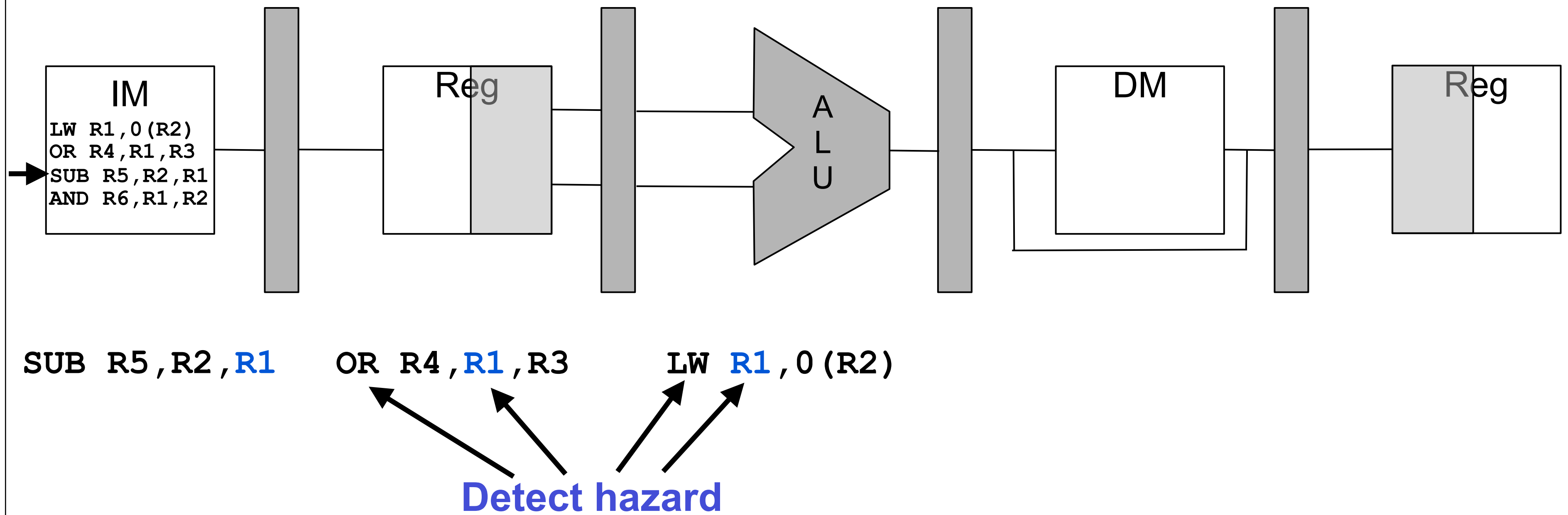
LW R1,0(R2)

Load Followed by R-type Instruction

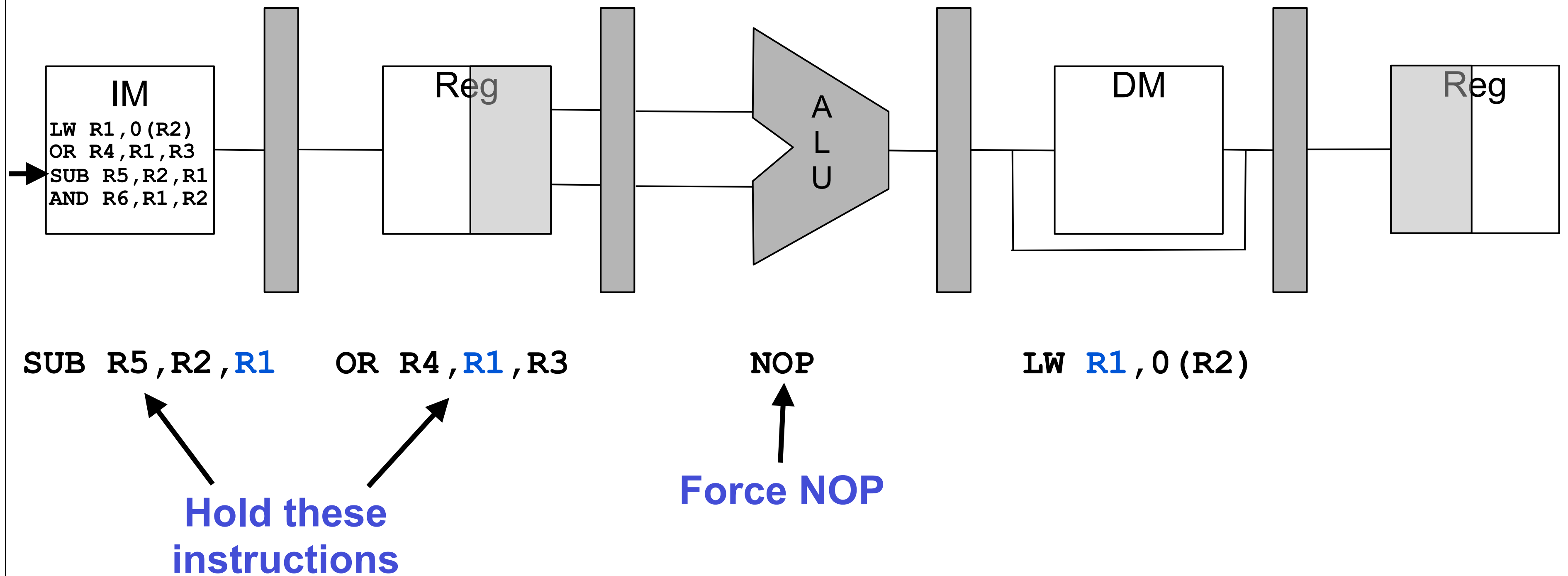


`OR R4,R1,R3`   `LW R1,0(R2)`

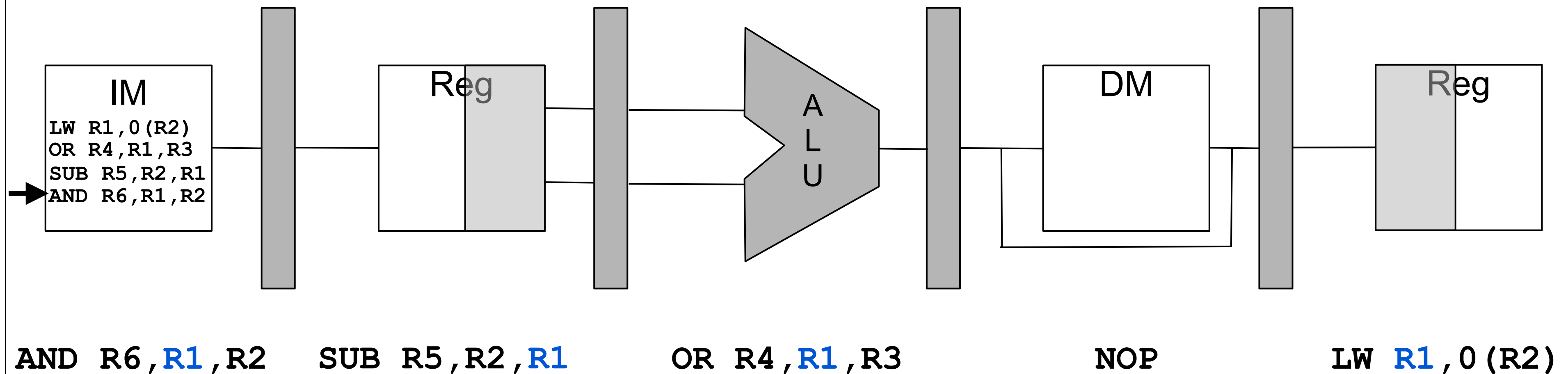
Load Followed by R-type Instruction



Load Followed by R-type Instruction

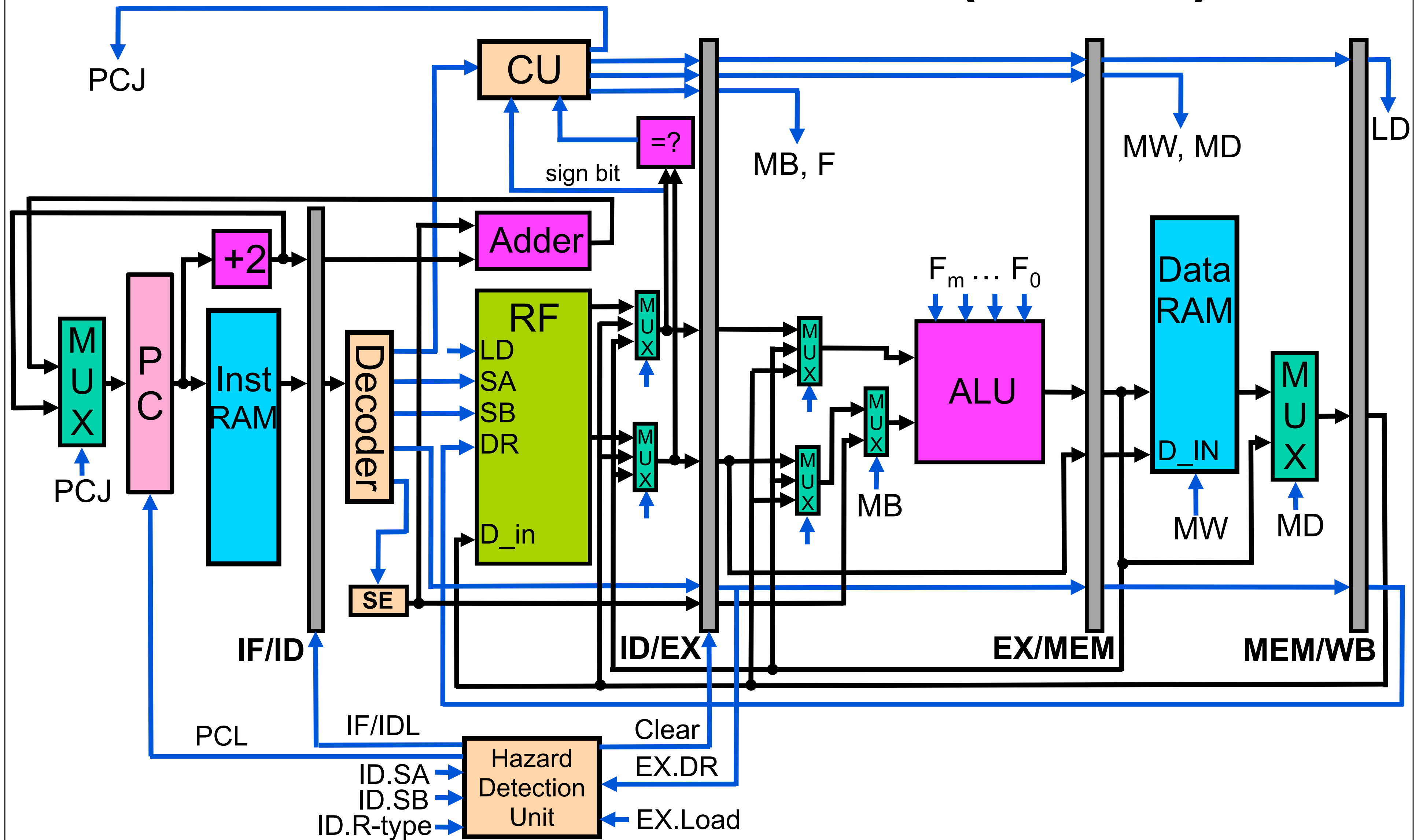


Load Followed by R-type Instruction



Pipeline continues normally
R1 value forwarded from WB to EX and ID

Hazard Detection Unit (Partial)



Before Next Class

- H&H 8-8.3

Next Time

More Pipelined Microprocessor
Caches