Computer Architecture:
MIPS Pipeline Datapath

Hossein Asadi (asadi@sharif.edu)
Department of Computer Engineering
Sharif University of Technology
Spring 2024

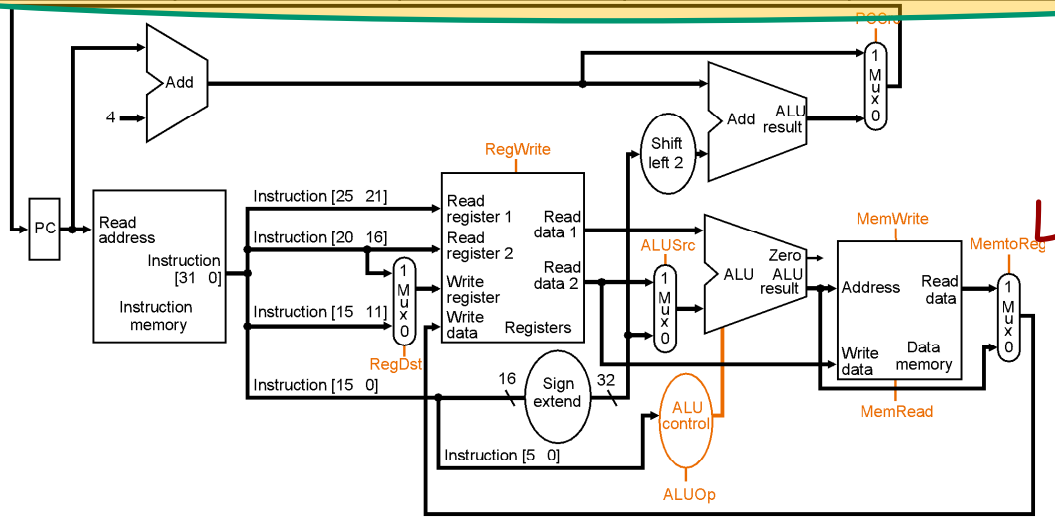
Copyright Notice

- Some Parts (text & figures) of this Lecture adopted from following:
  - D.A. Patterson and J.L. Hennessy, "Computer Organization and Design: the Hardware/Software Interface" (MIPS), 6th Edition, 2020.
  - J.L. Hennessy and D.A. Patterson, "Computer Architecture: A Quantitative Approach", 6th Edition, Nov. 2017.
  - "Intro to Computer Architecture" handouts, by Prof. Hoe, CMU, Spring 2009.
  - "Computer Architecture & Engineering" handouts, by Prof. Kubiatowicz, UC Berkeley, Spring 2004.
  - "Intro to Computer Architecture" handouts, by Prof. Hoe, UWisc, Spring 2021.
  - "Computer Arch I" handouts, by Prof. Garzarán, UIUC, Spring 2009.

Quick Reminder from Previous Lecture

Single-Cycle CPU Clock Cycle Time

	I-cache	Decode, R-Read	ALU	PC update	D-cache	R-Write	Total
R-type	1	1	.9	-	-	.8	3.7
Load	1	1	.9	-	1	.8	4.7
Store	1	1	.9	-	1	-	3.9
beq	1	1	.9	.1	-	-	3.0



Clock cycle time
= 4.7 + setup + hold

Load on critical path

Setup time?

Hold time?

Critical path?

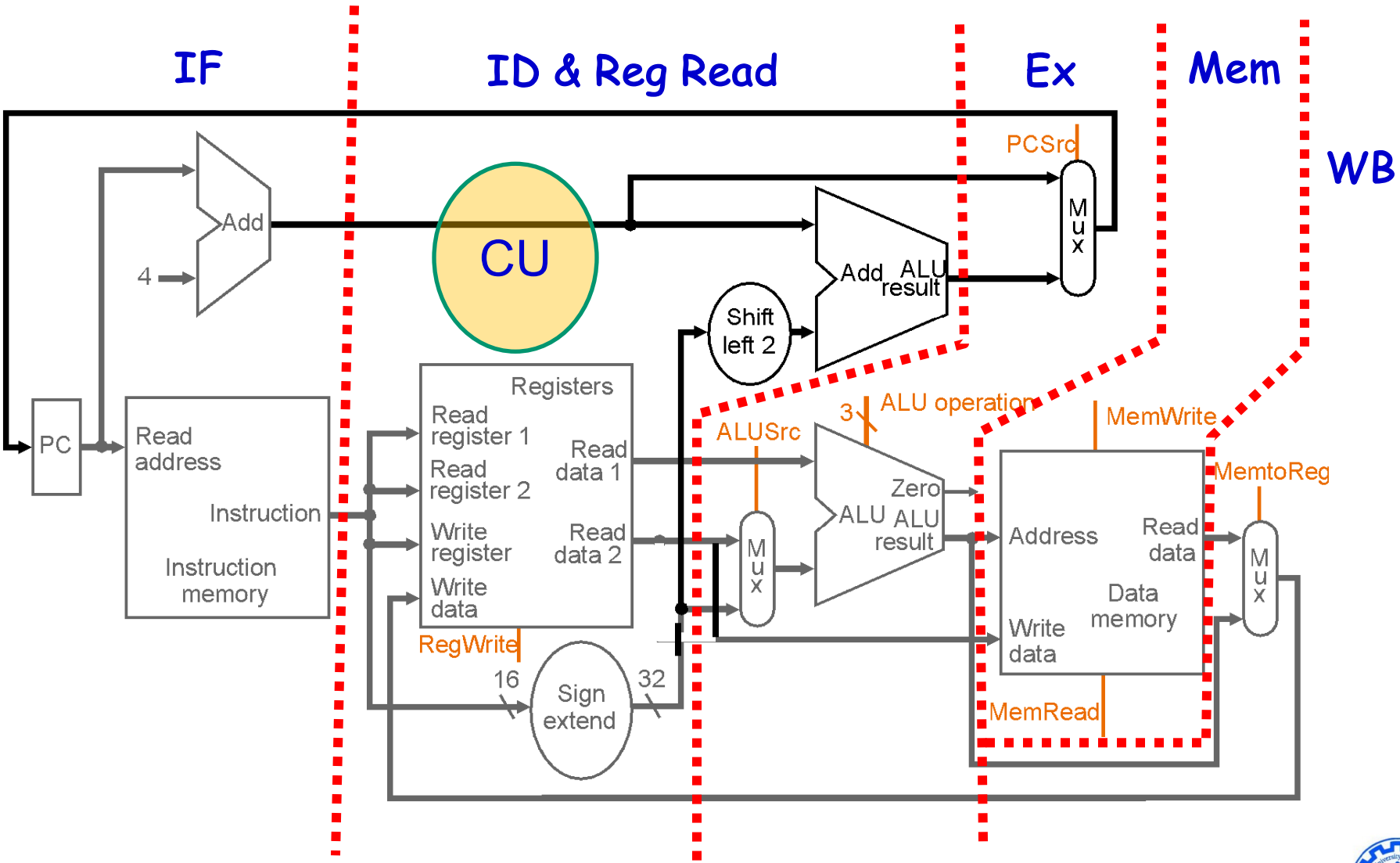
Multicycle Implementation

Goal: Balance amount of work done each cycle

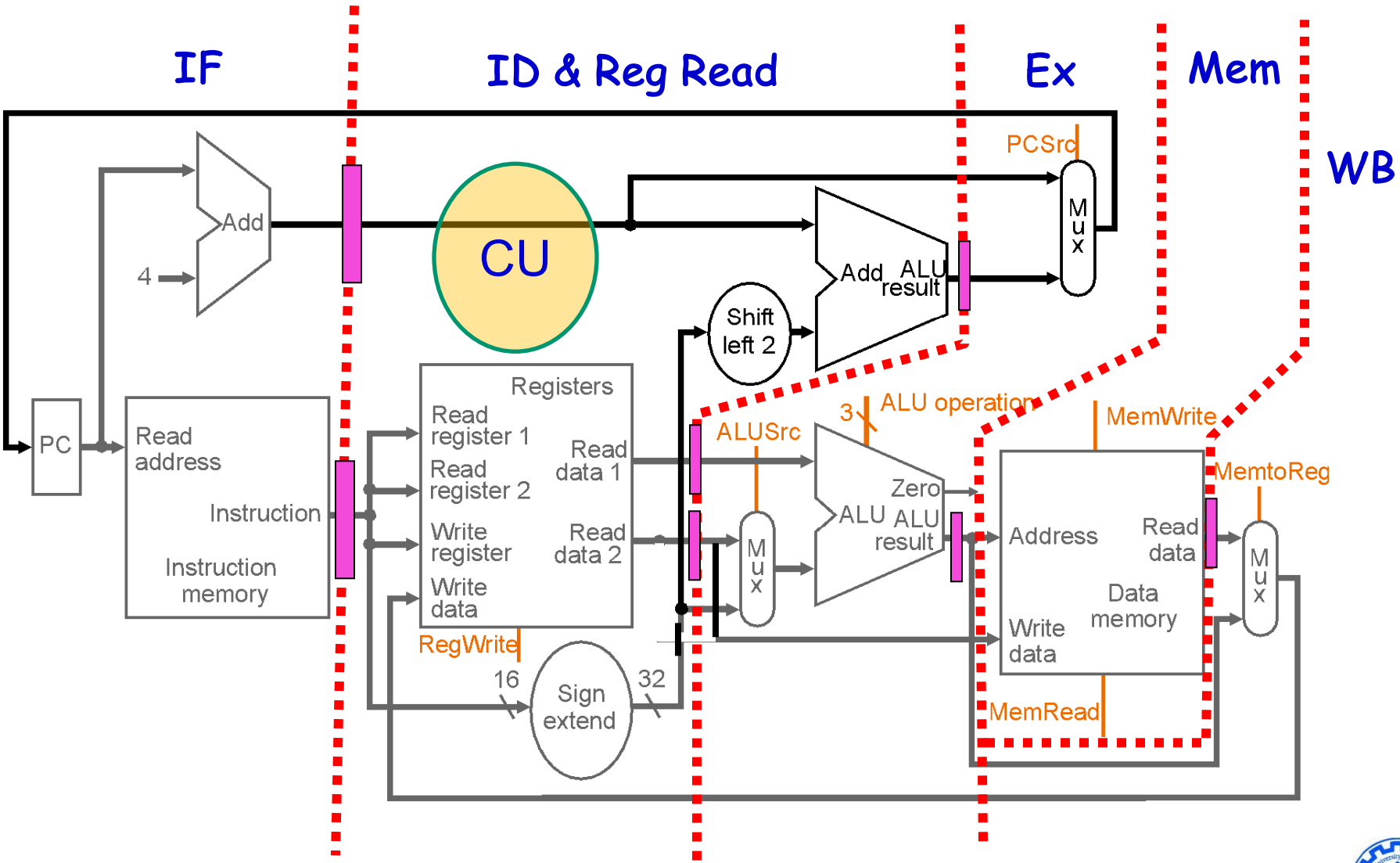
	I cache	Decode, R-Read	ALU	PC update	D cache	R-Write	Total
R-type	1	1	.9	-	-	.8	3.7
Load	1	1	.9	-	1	.8	4.7
Store	1	1	.9	-	1	-	3.9
beq	1	1	.9	.1	-	-	3.0

- Load needs 5 cycles
- Store and R-type need 4
- beq needs 3

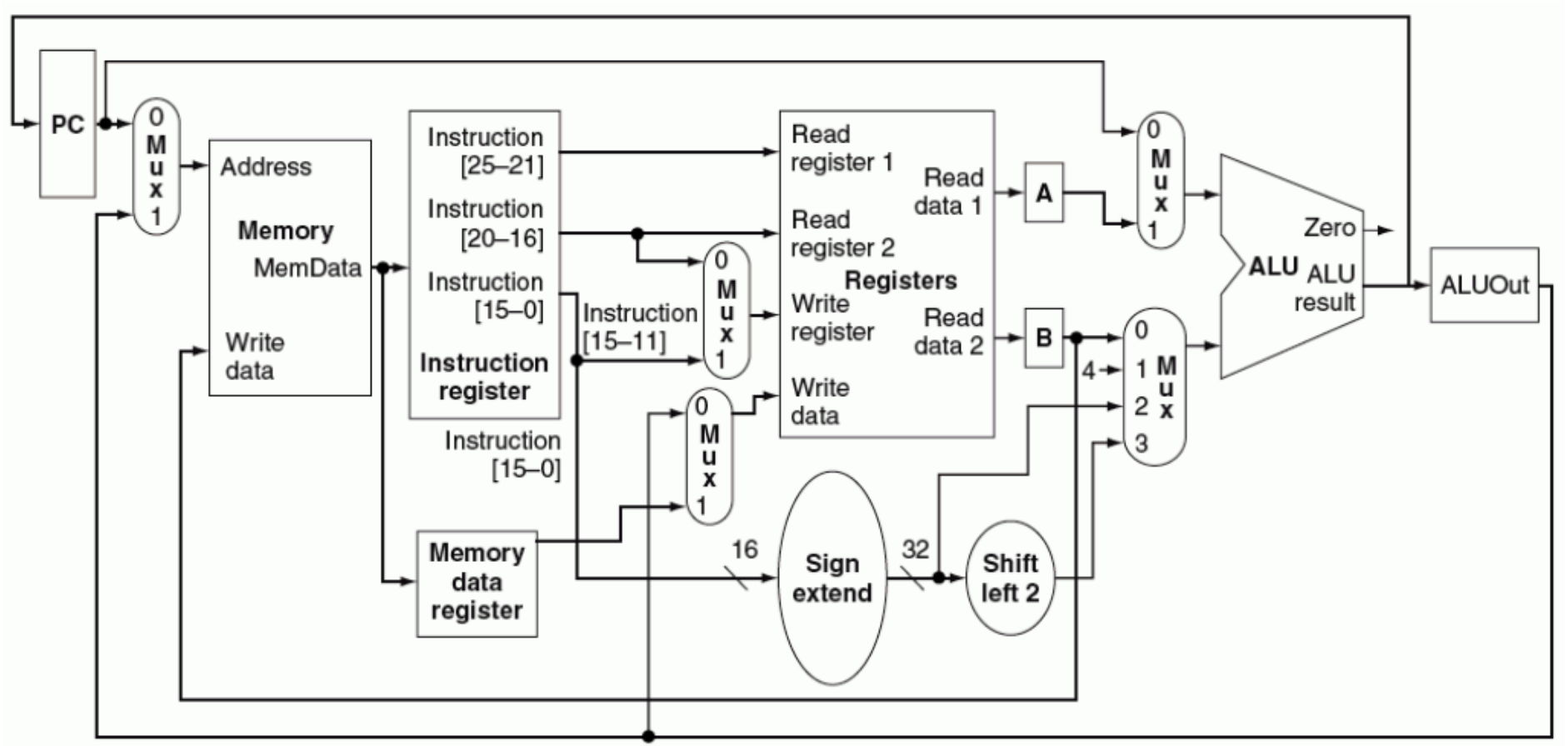
Partitioning Single-Cycle Design



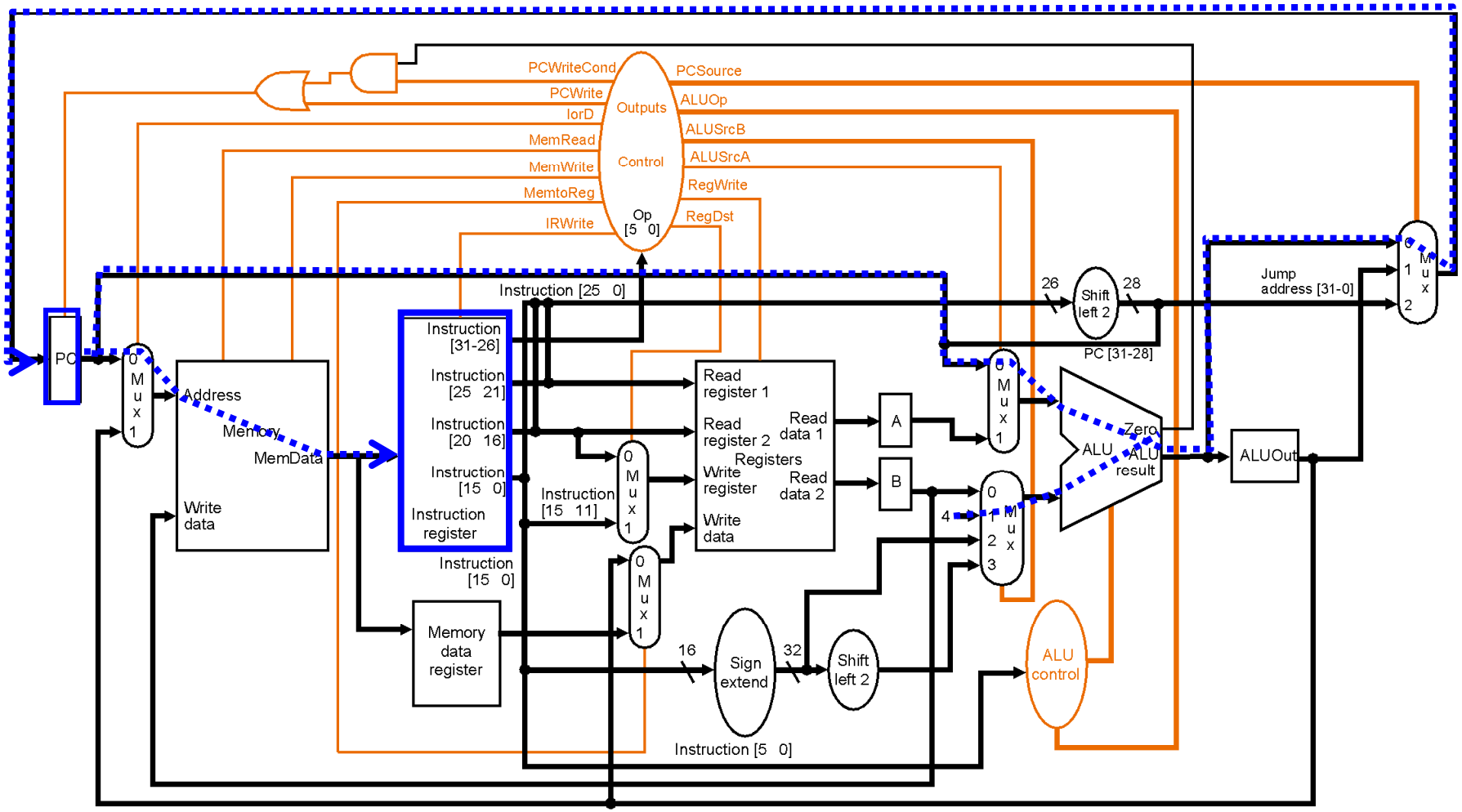
Where to Add Registers?



Multicycle Datapath (cont.)



Cycle 1: Instruction Fetch



Datapath:

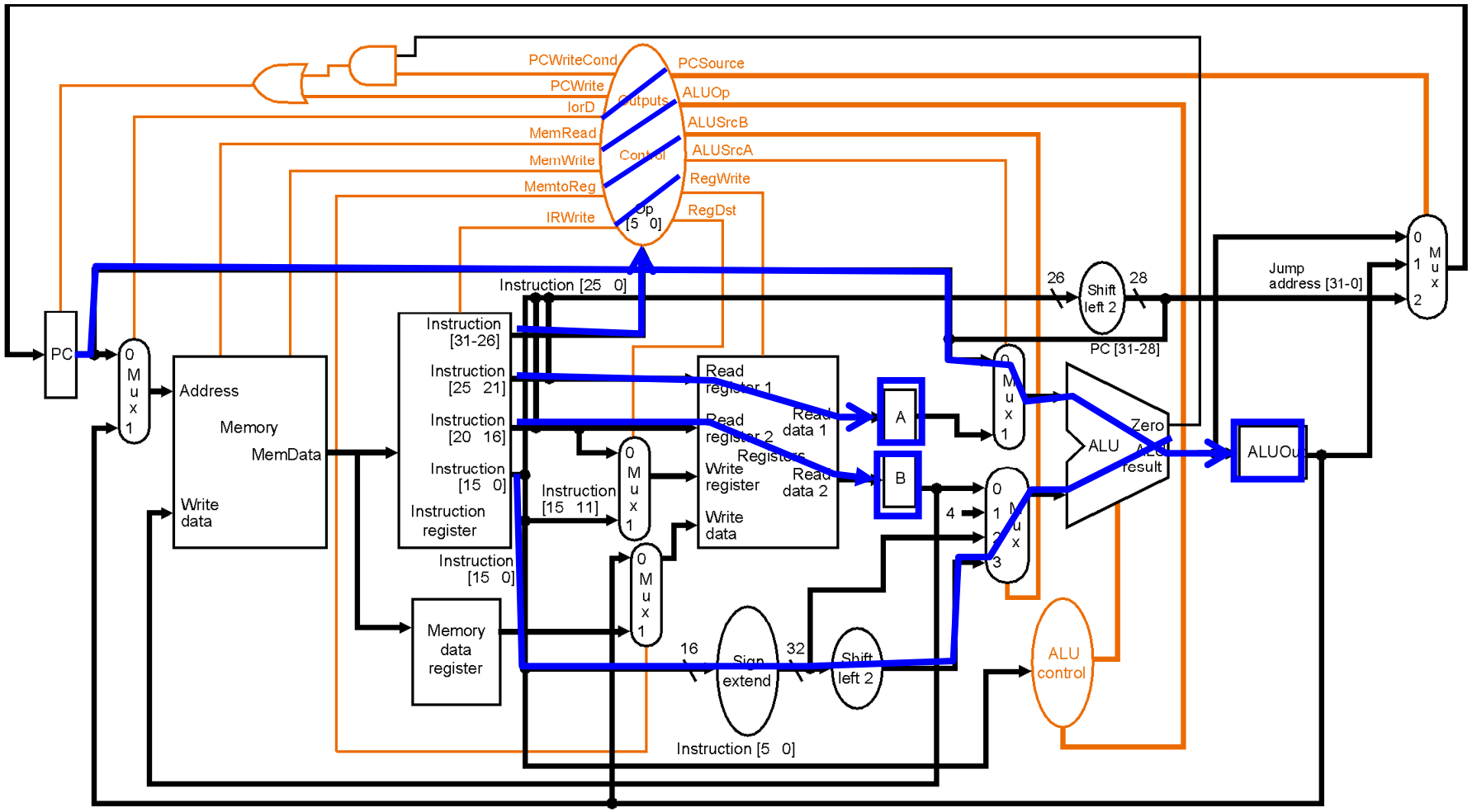
IR <= Mem[PC]

PC <= PC + 4

Control:

IorD=0, MemRead=1, MemWrite=0,
IRwrite=1, ALUsrcA=0, ALUsrcB=01,
PCWrite=1, ALUop=00, PCsource=00

Cycle 2: ID & RF Cycle

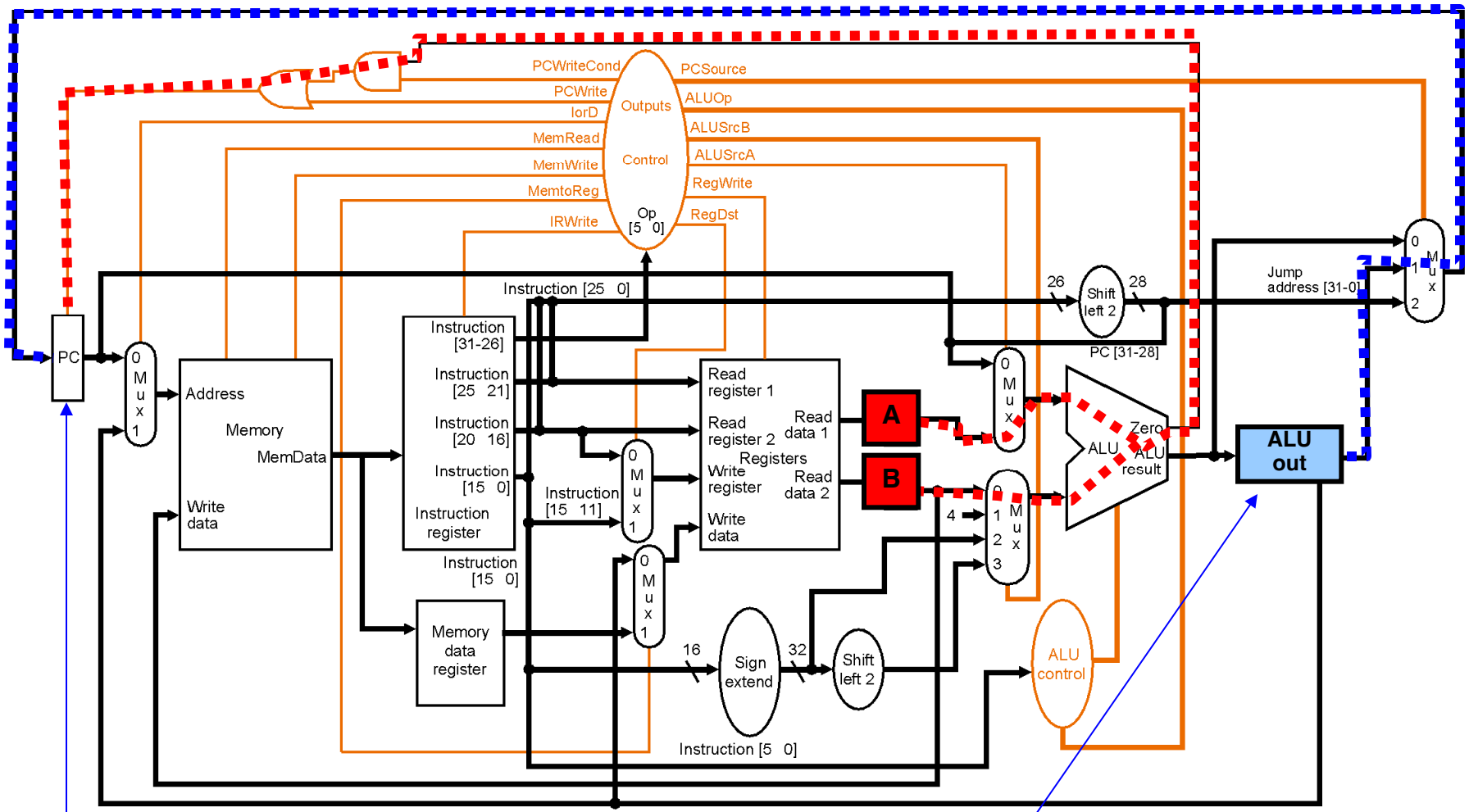


A <= GPR[IR[25-21]]

B <= GPR[IR[20-16]]

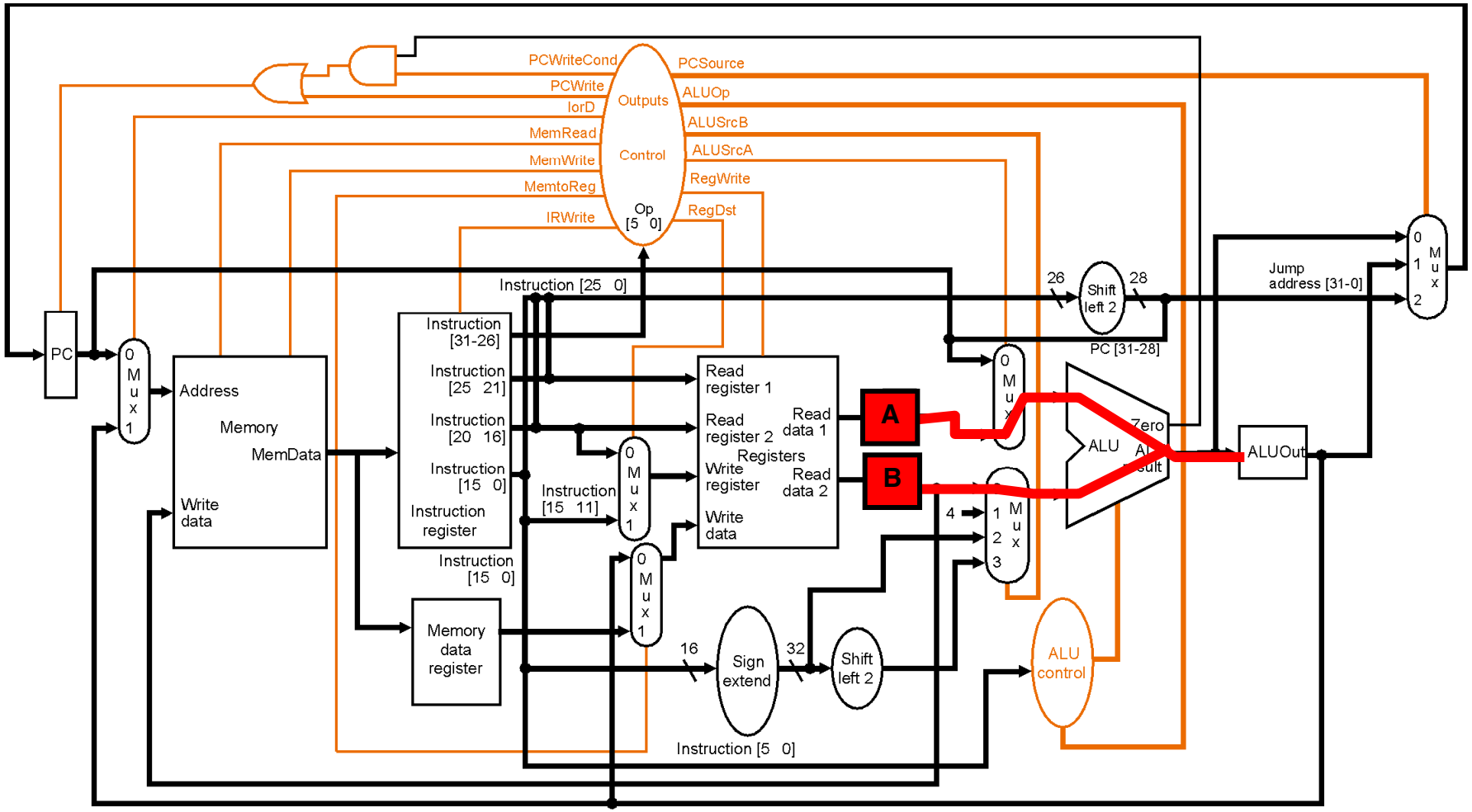
ALUout <= PC + (sign-extend (IR[15-0]) << 2)

Cycle 3 for beq: Execute



- In cycle 1, PC was incremented by 4
- In cycle 2, ALUout was set to branch target
- This cycle, we conditionally update PC: if (A==B) PC=ALUout

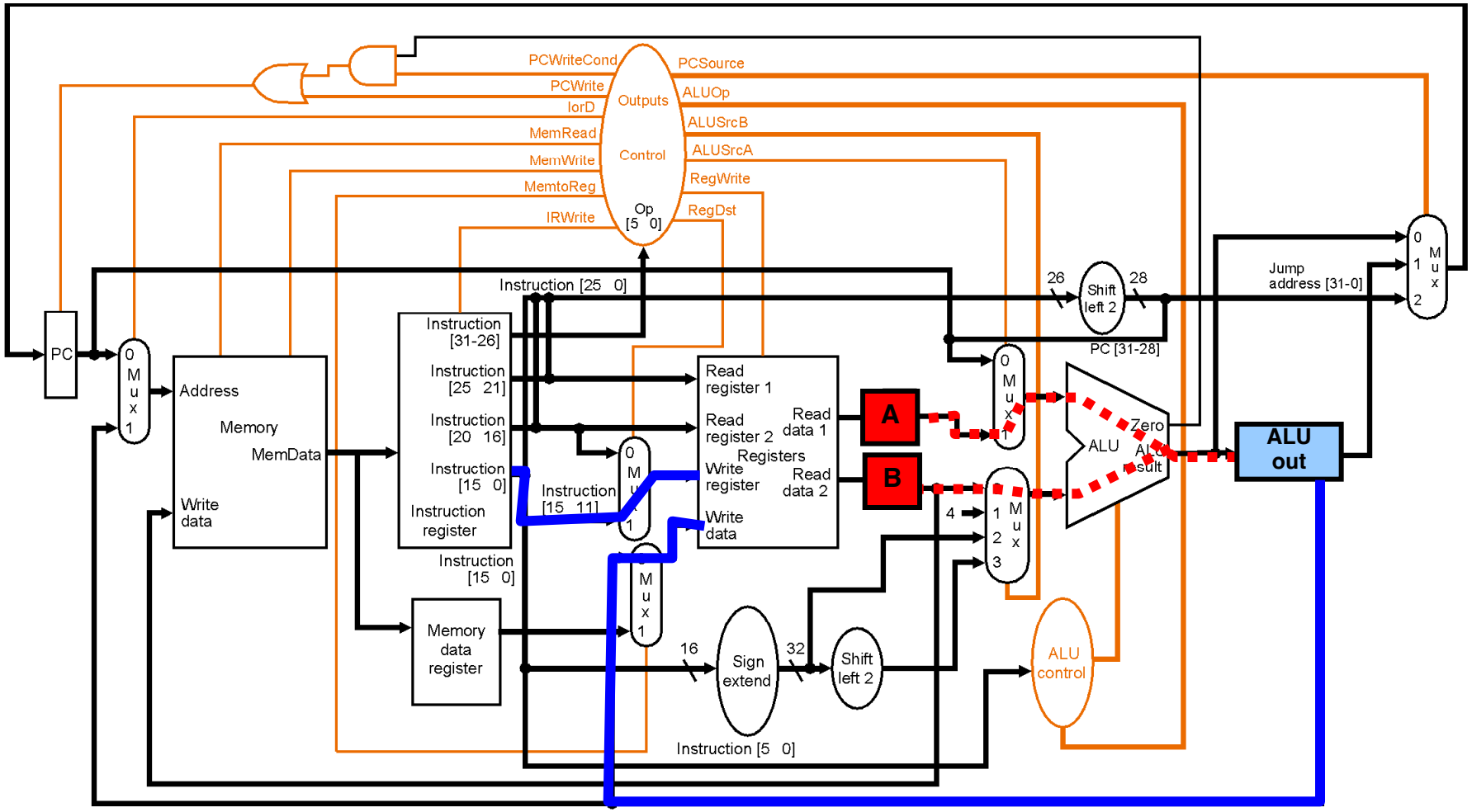
R-Type Execution



Cycle 3: **`ALUout = A op B`**

Cycle 4: `GPR[IR[15-11]] = ALUout`

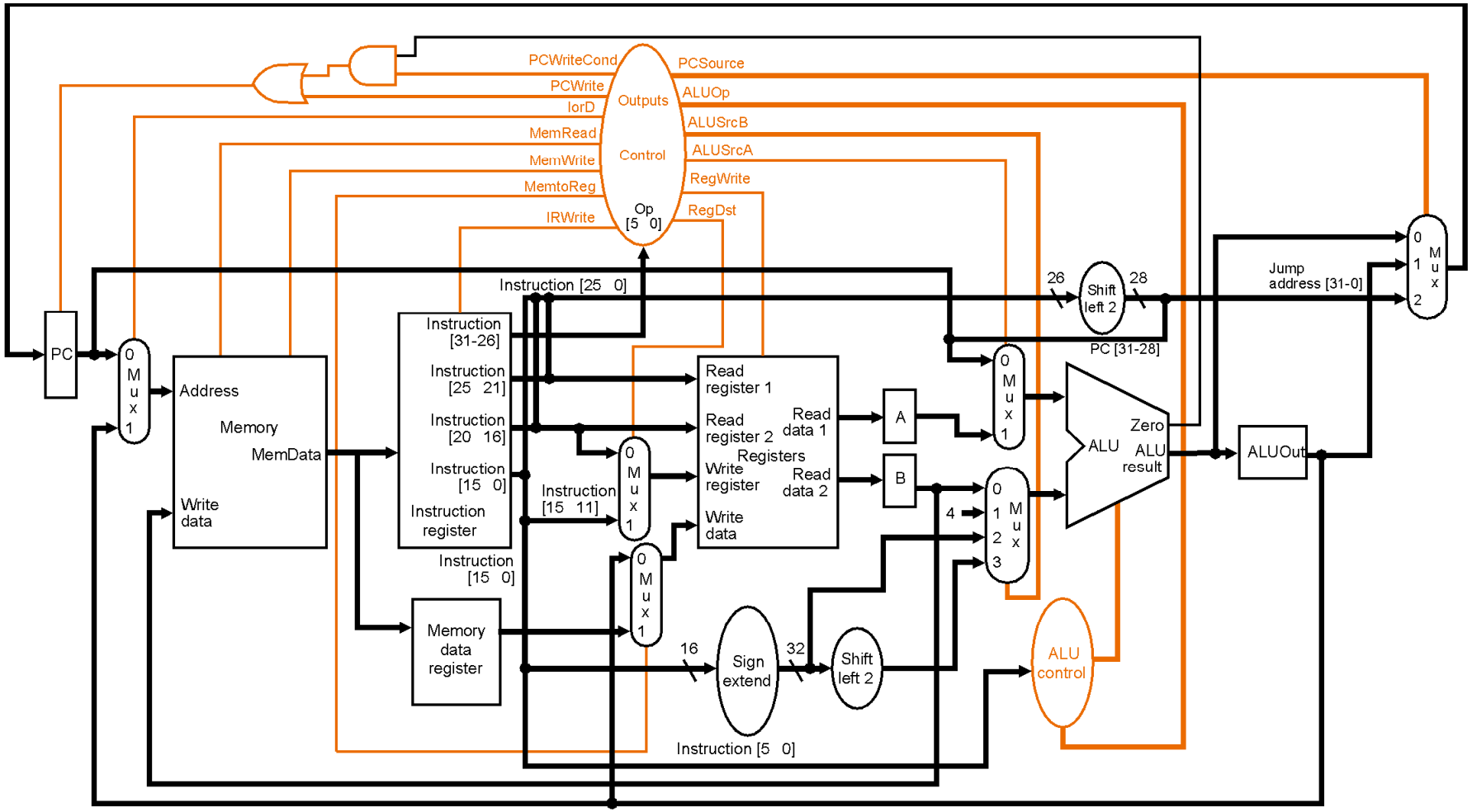
R-Type Execution & WB



Cycle 3: `ALUout = A op B`

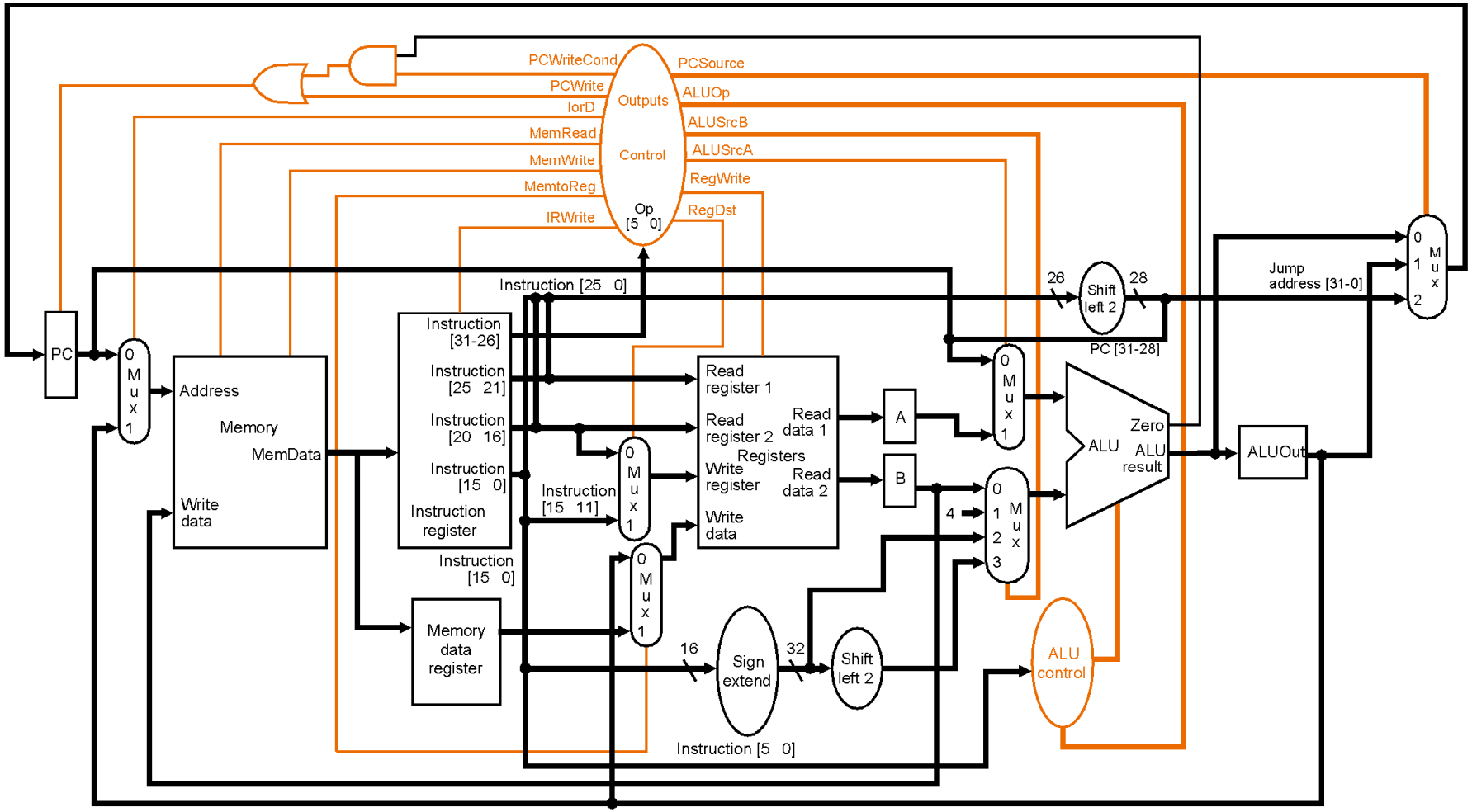
Cycle 4: **`GPR[IR[15-11]] = ALUout`**

Cycle 3 for lw/sw: Address Computation



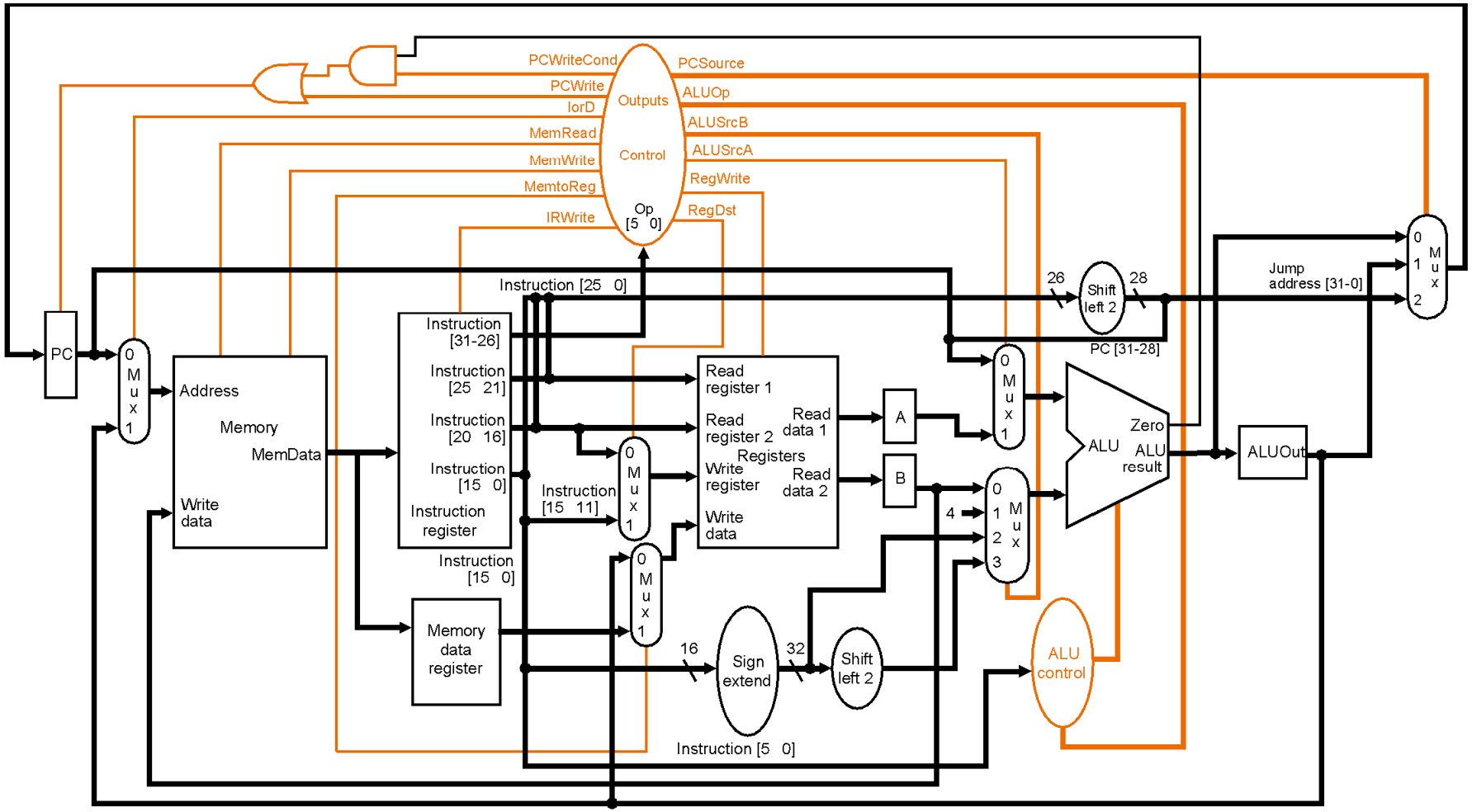
ALUout = A + sign-extend(IR[15-0])

Cycle 4 for Store: Memory Access



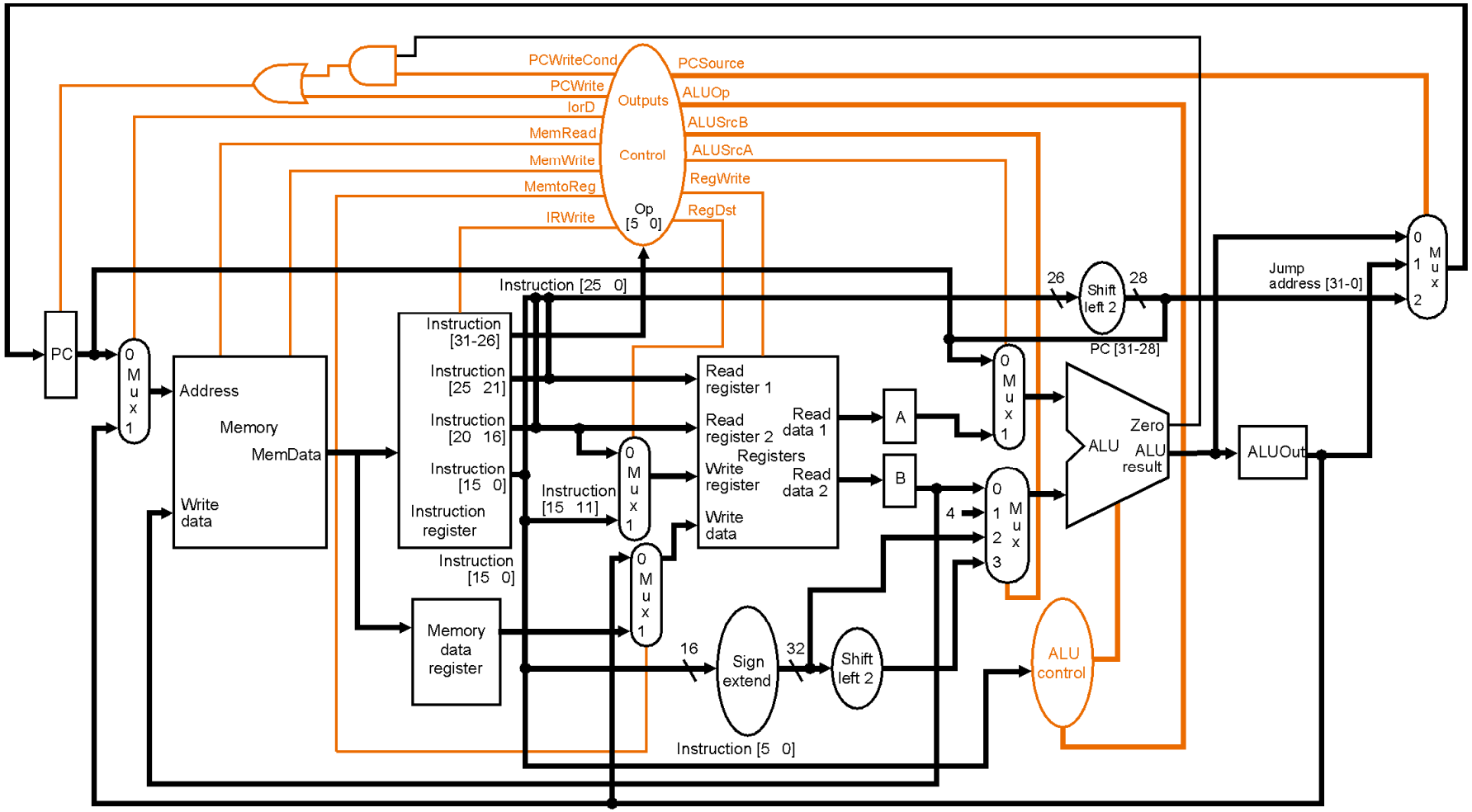
Memory[ALUout] = B

Cycle 4 for Load: Memory Access



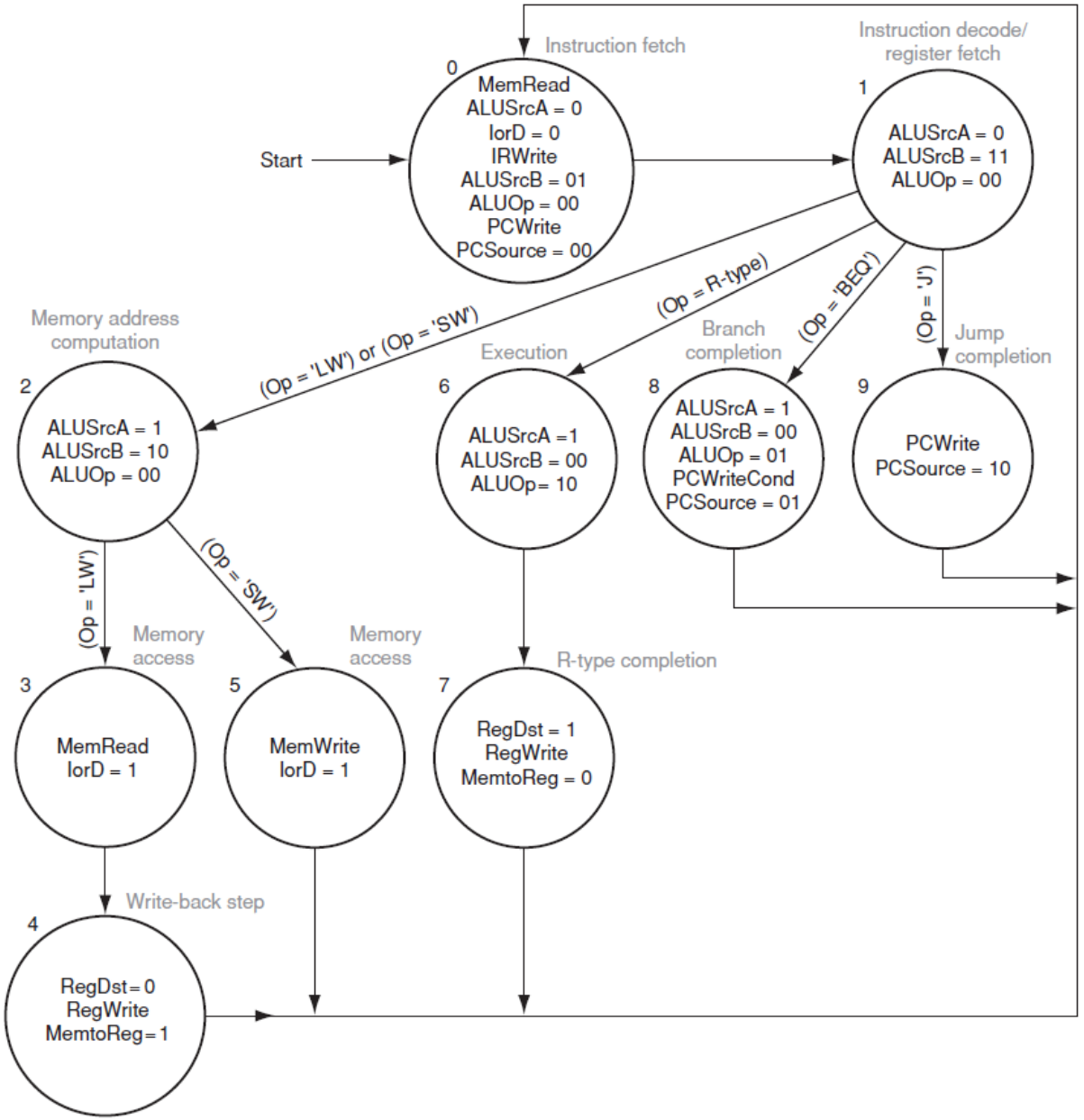
MDR = Memory[ALUout]

Cycle 5 for load: WriteBack



GPR[IR[20-16]] = MDR

Complete FSM



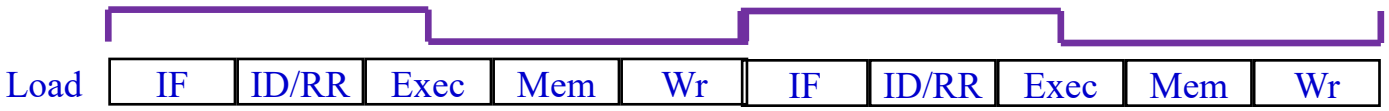
Our Lectur Today

Topics Covered Today

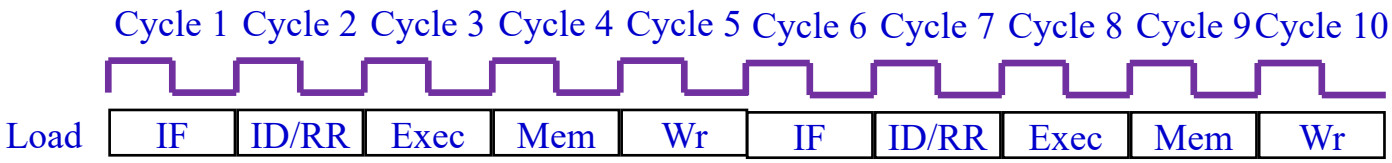
- Basics of Pipelined Design
- MIPS Pipelined Datapath
- Pipelined Performance
- Hazards in Pipelined Design
  - Structural hazards
  - Data hazards
  - Control hazards

Instruction Latencies and Throughput

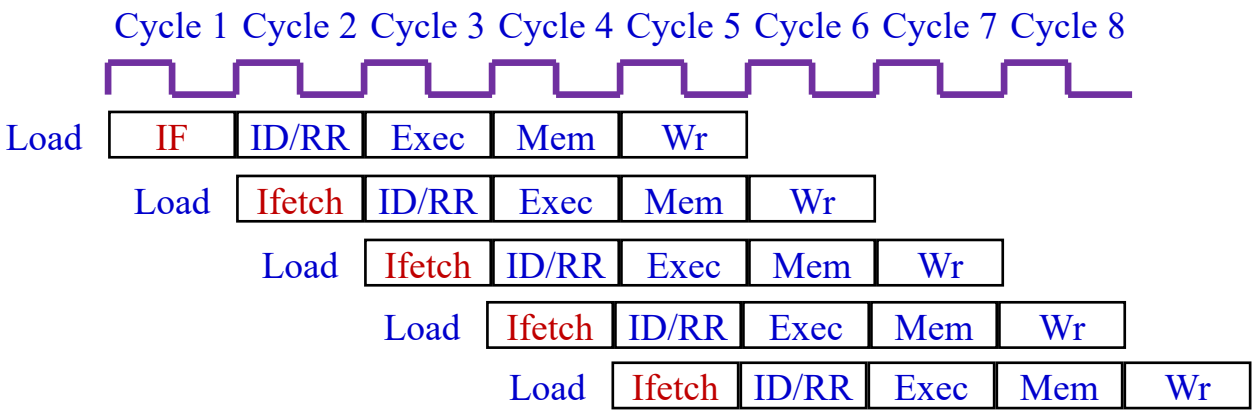
- **Single-Cycle CPU**



- **Multiple Cycle CPU**



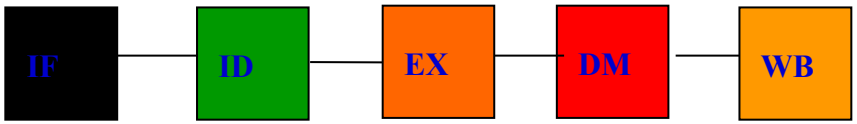
- **Pipelined CPU**



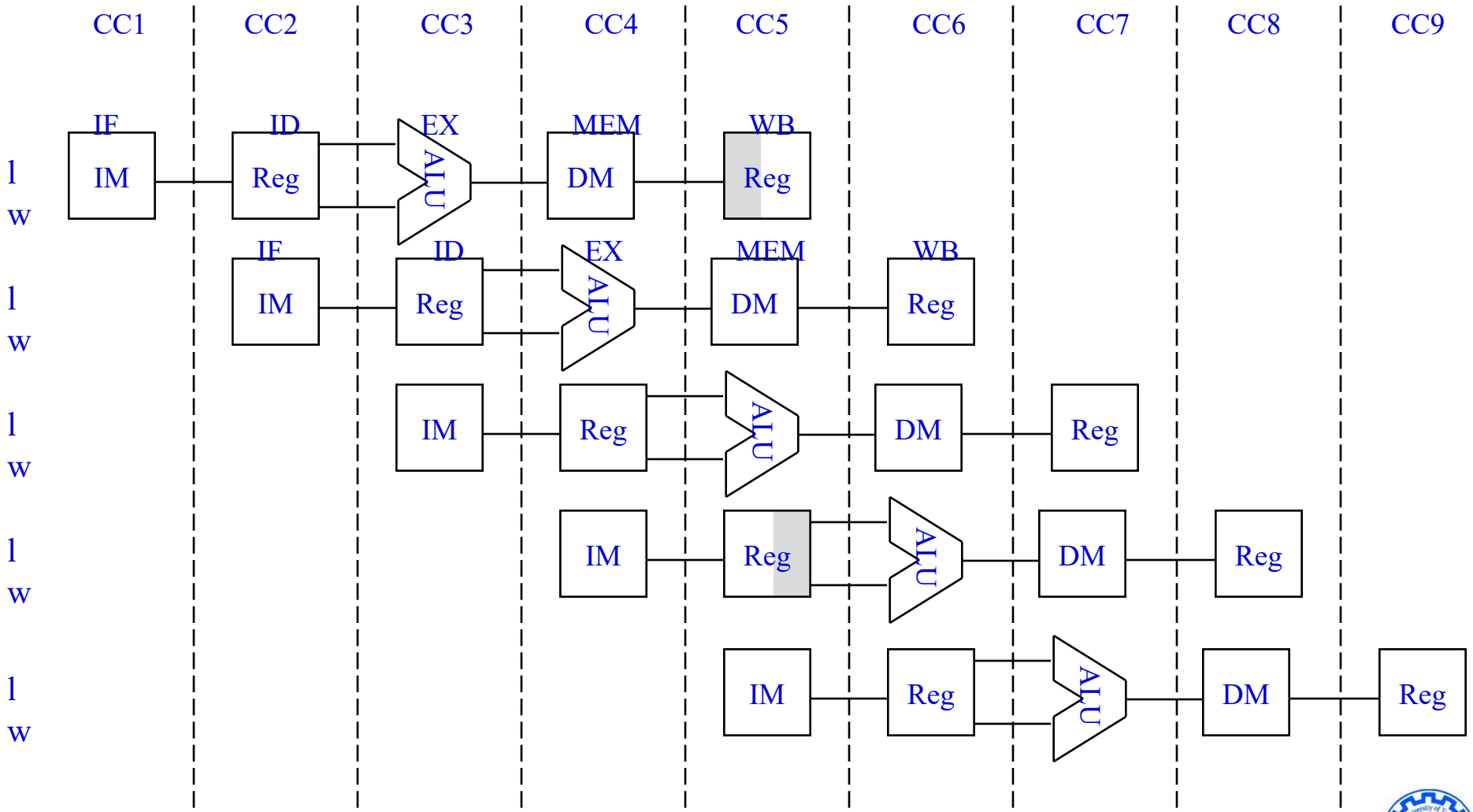
Introduction of Pipeline

Machine assembly line

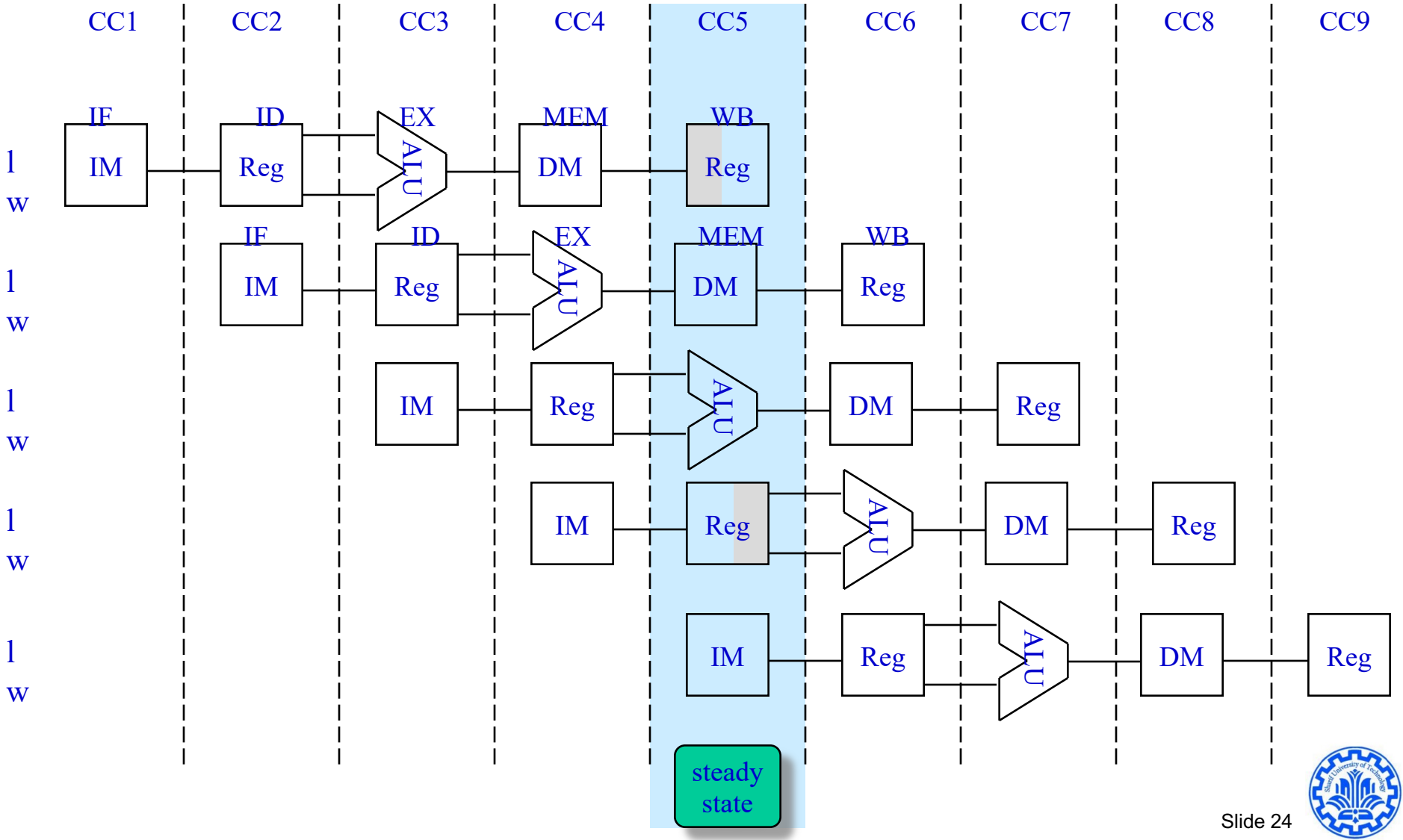
Datapath of pipeline



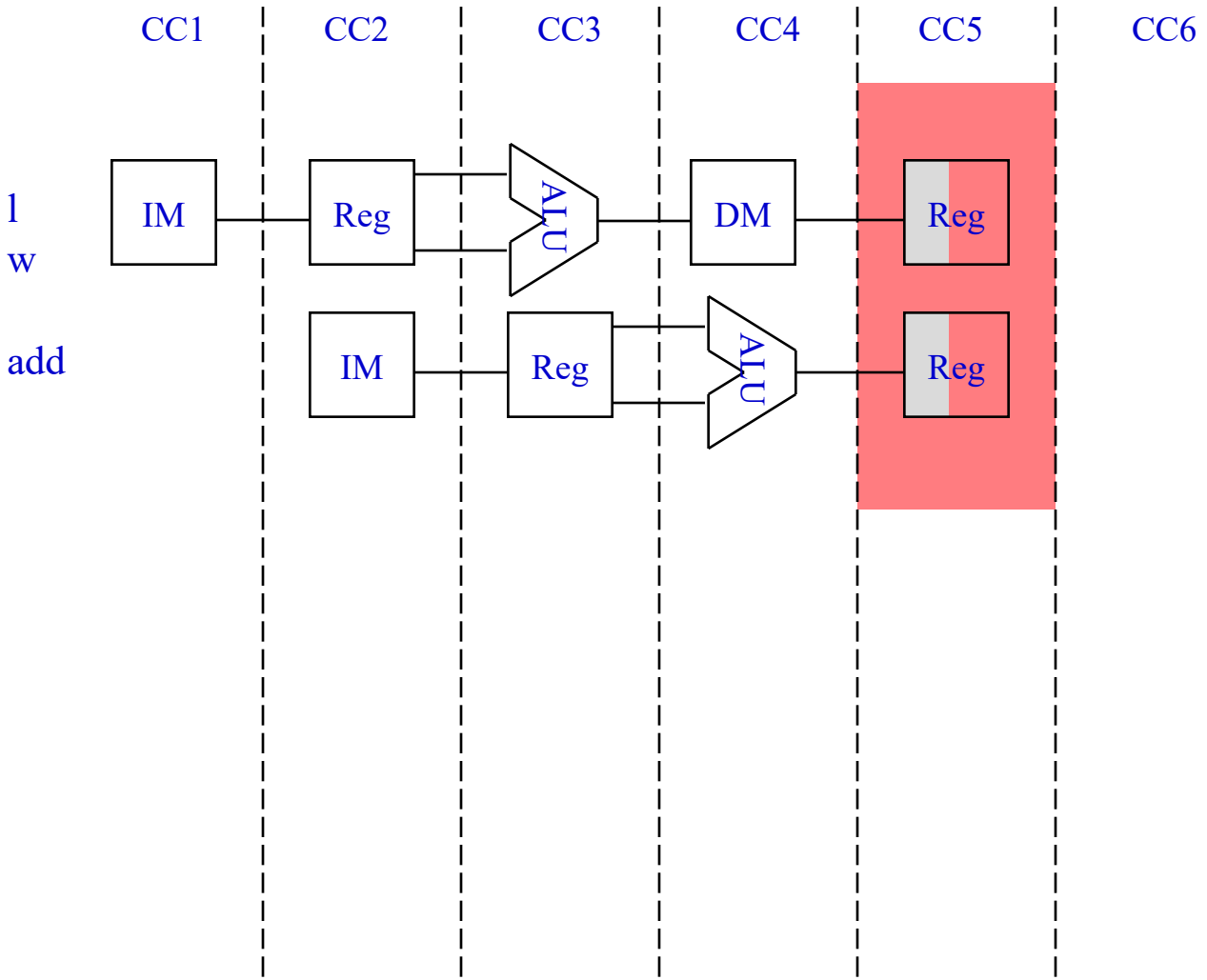
Execution in a Pipelined Datapath



Execution in a Pipelined Datapath

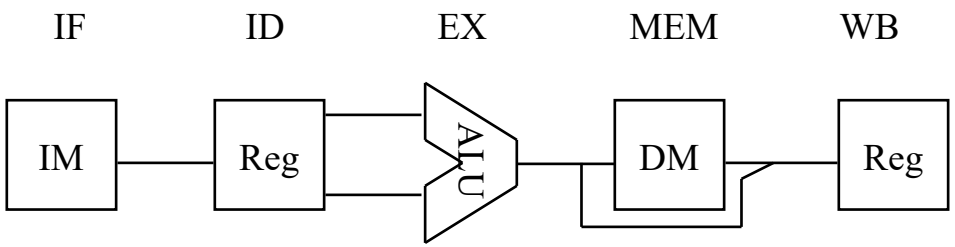


Mixed Instructions in Pipeline



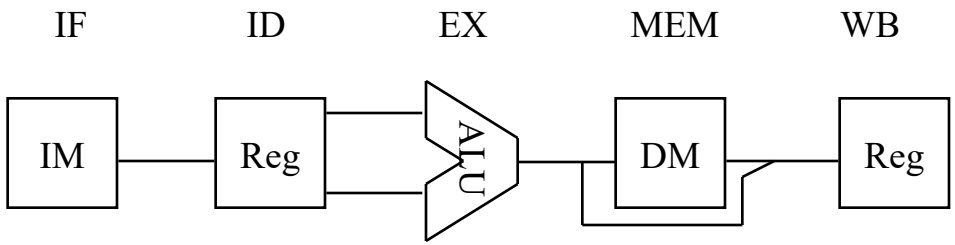
Pipeline Principles

- Principles
  - All instructions that share a pipeline must have same *stages* in same *order*
    - ➔ *add* does nothing during Mem stage
    - ➔ *sw* does nothing during WB stage



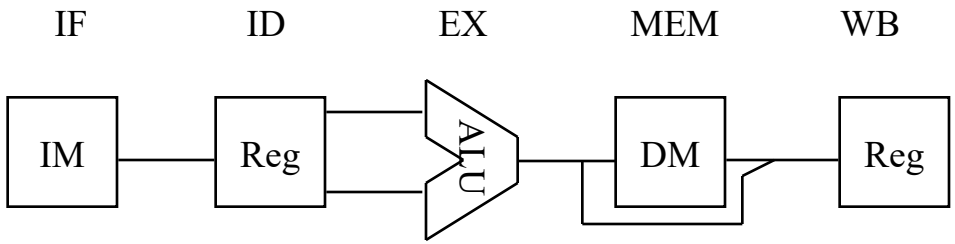
Pipeline Principles (cont.)

- Principles
  - All intermediate values must be latched each cycle
  - No functional block reuse for an instruction
    - E.g. need 2 adders and ALU (like in single-cycle)

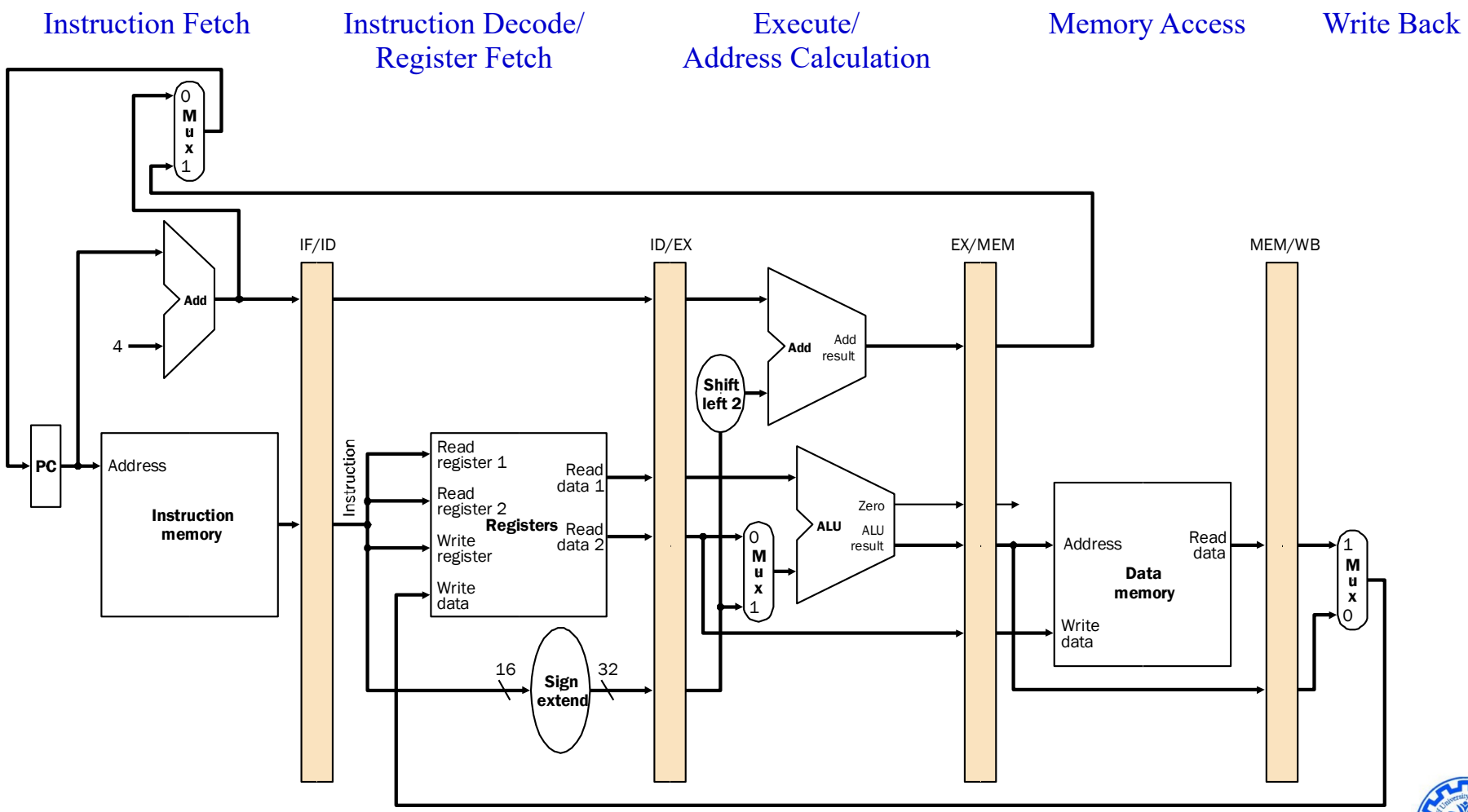


Pipeline Principles

- Question:
  - What if we want to bypass a stage for an instruction?

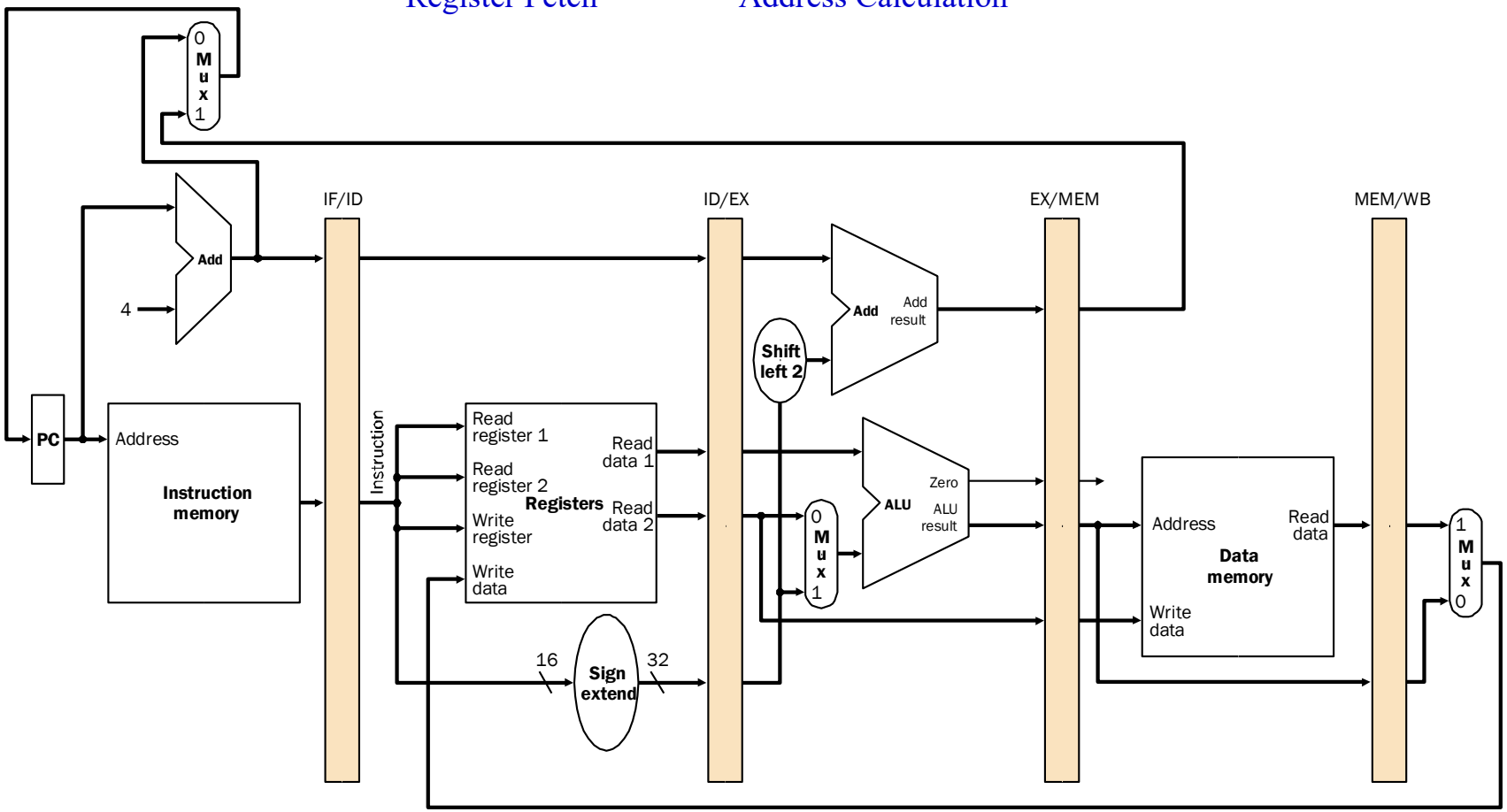


Pipeline in Execution



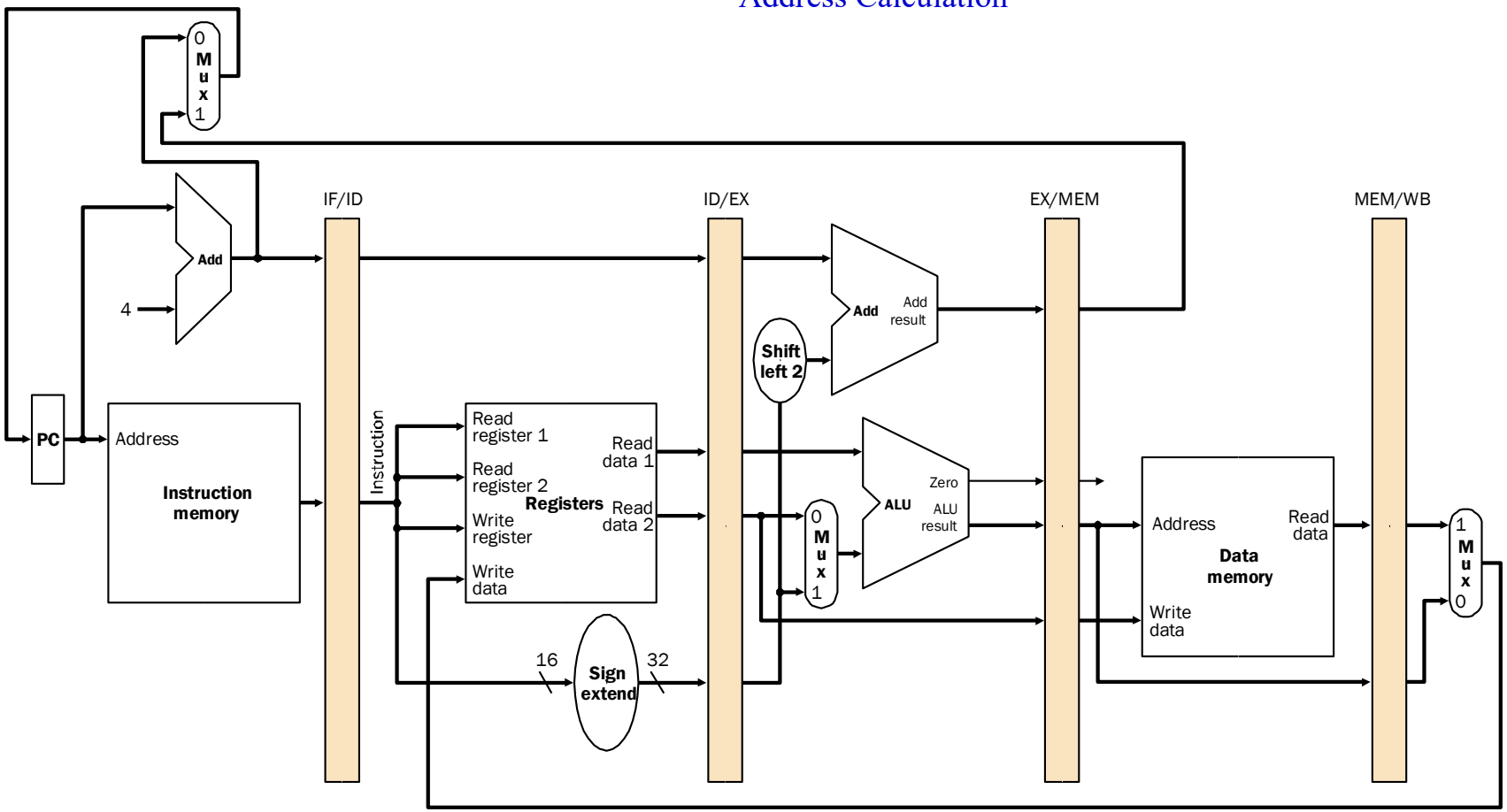
Pipeline in Execution

add $10, $1, $2 | Instruction Decode/ Register Fetch | Execute/ Address Calculation | Memory Access | Write Back



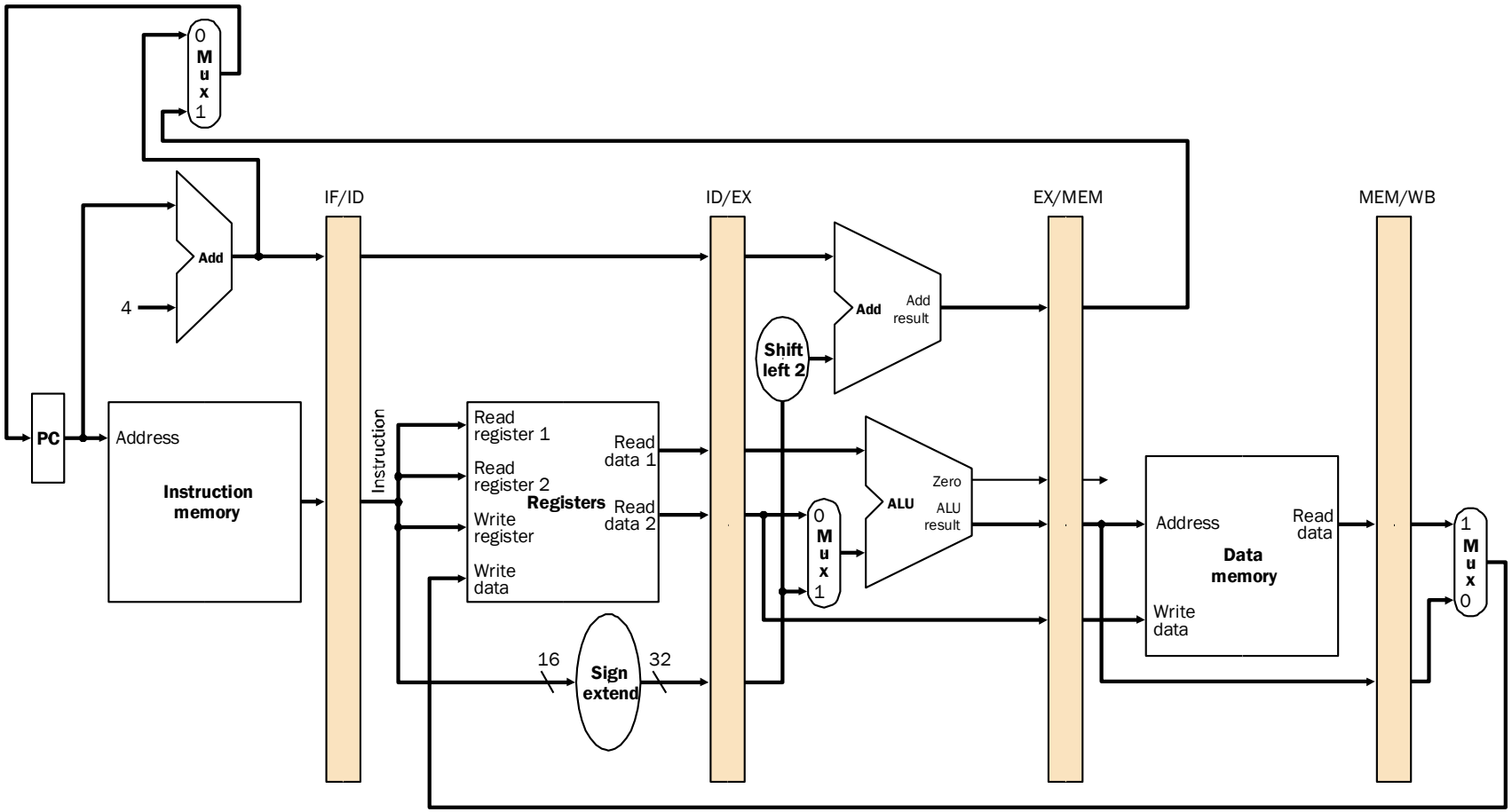
Pipeline in Execution

lw $12, 1000($4) **add $10, $1, $2** Execute/ Address Calculation Memory Access Write Back



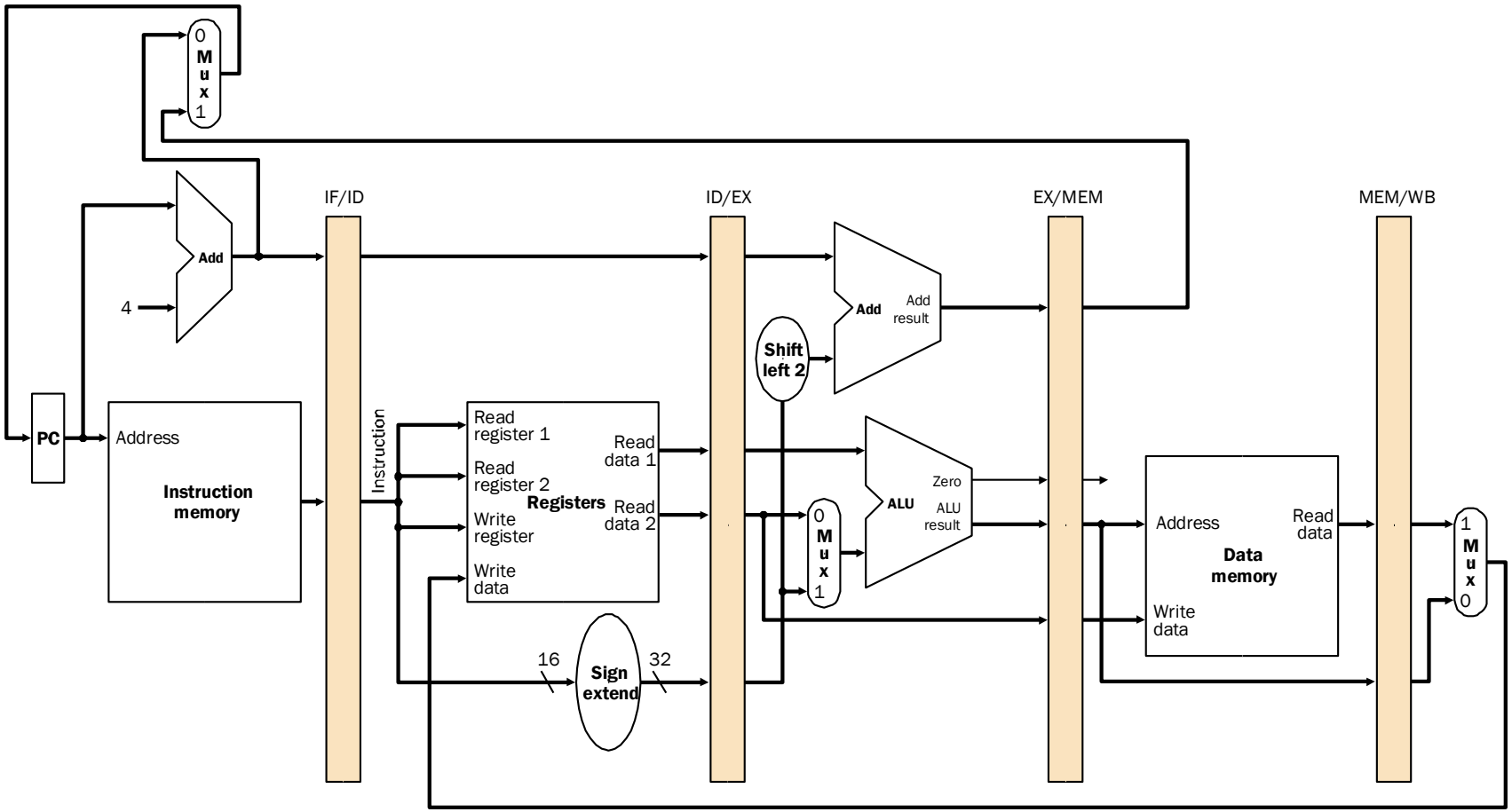
Pipeline in Execution

sub $15, $4, $1 **lw $12, 1000($4)** **add $10, $1, $2** Memory Access Write Back



Pipeline in Execution

Instruction Fetch | **sub $15, $4, $1** | **lw $12, 1000($4)** | **add $10, $1, $2** | Write Back



Pipelining Performance

- ET = IC * CPI * CT
- Achieve High *throughput*
  - Without reducing instruction *latency*
- Example:
  - A CPU that takes 5ns to execute an instruction pipelined into 5 equal stages
    - Latch between each stage has a delay of 0.25 ns

1. Min. clock cycle time of this arch?
2. Max. speedup that can be achieved by this arch (compared to single cycle arch)?

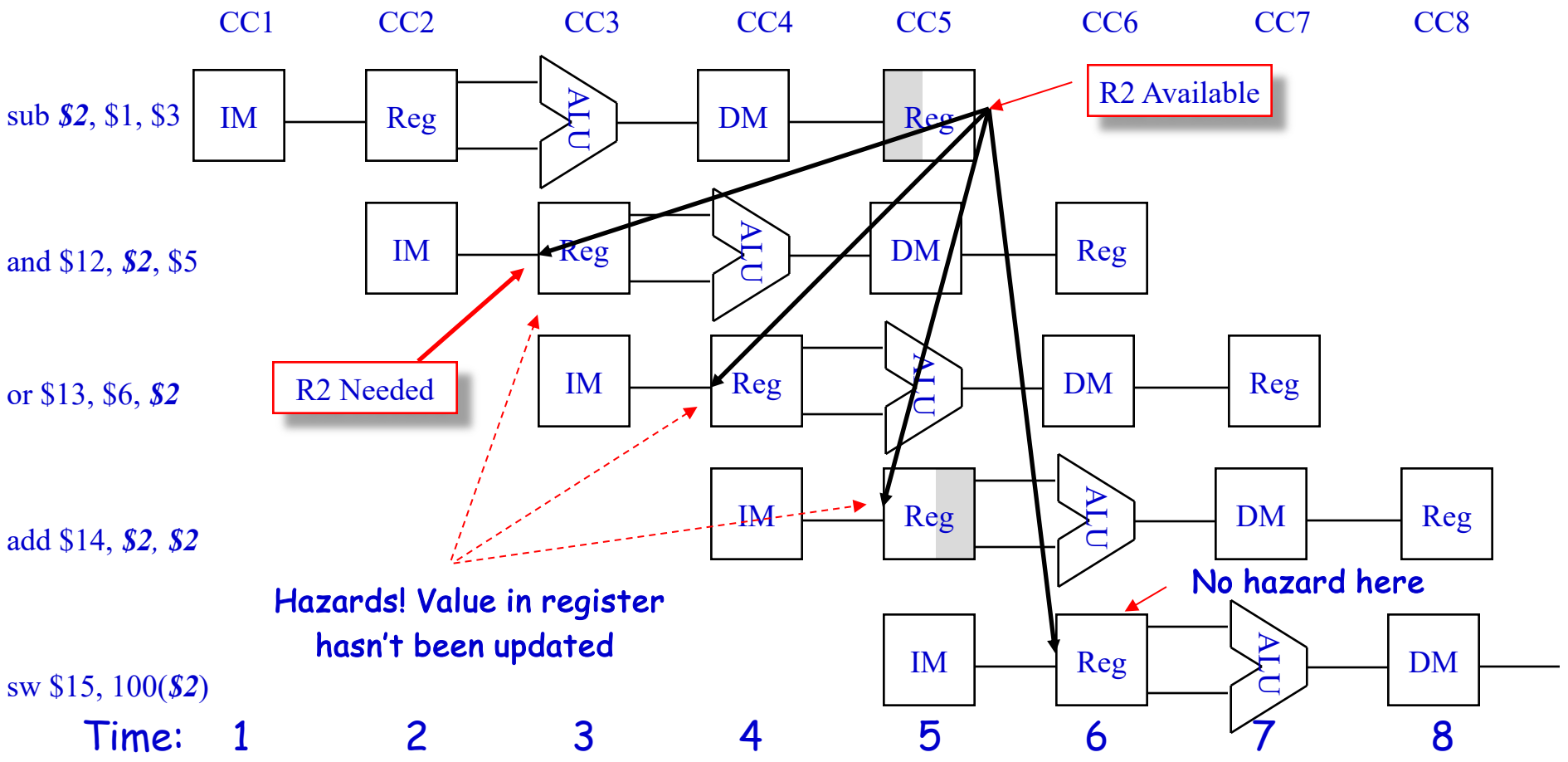
Issues With Pipelining

- **Pipelining Creates Potential Hazards**
  - What happens if two instructions need a same structure?
    - Structure hazard
  - What happens when an instruction needs result of another instruction?
    - Data hazard
  - What happens on a branch?
    - Control hazard

Structural Hazards

- How?
  - Two instructions require use of a given hardware resource at same time
- Access to Memory
  - Separate instruction and data caches
- Access to Register File
  - Multiple port register file

Data Hazards



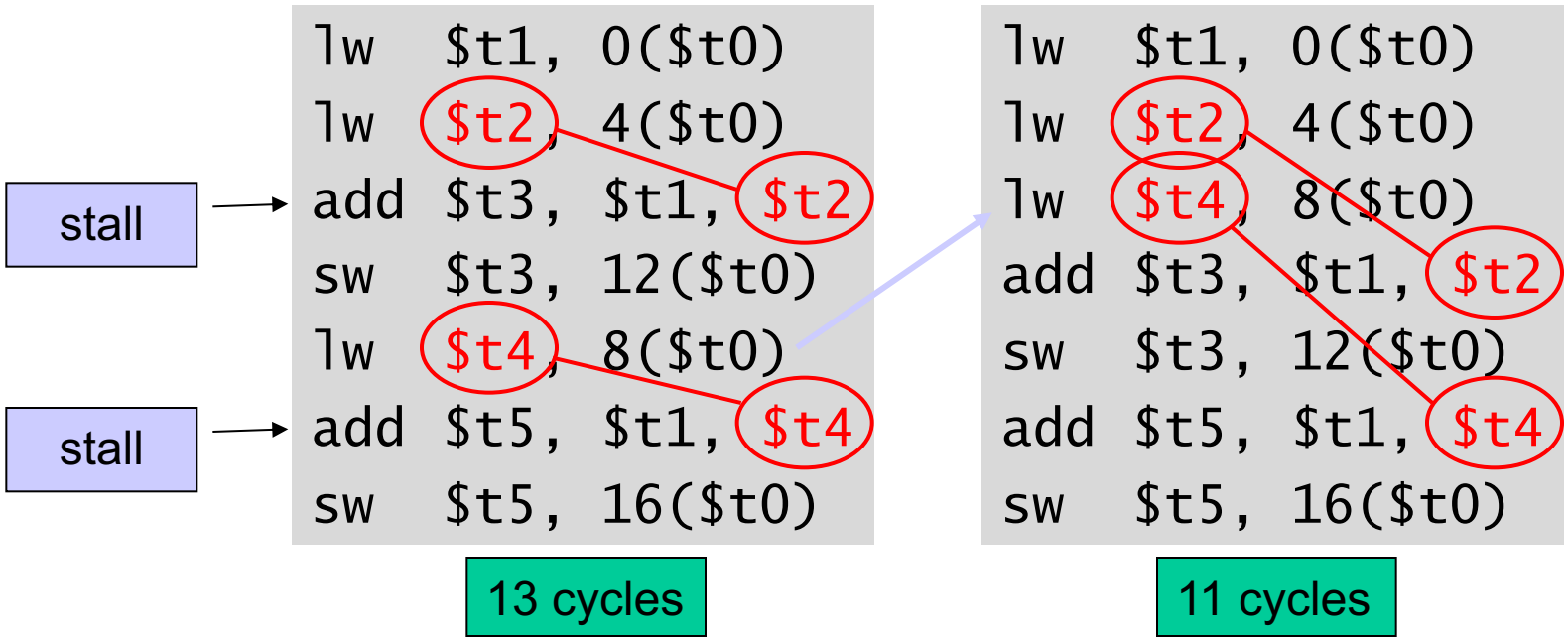
Data Hazard: result needed before it is available

Handling Data Hazards

- SW Solution
  - Insert independent instructions
  - No-ops instructions
  - Code reordering
- HW Solutions
  - Insert bubbles (i.e. stall pipeline)
  - Data forwarding
  - Latch-based GPR

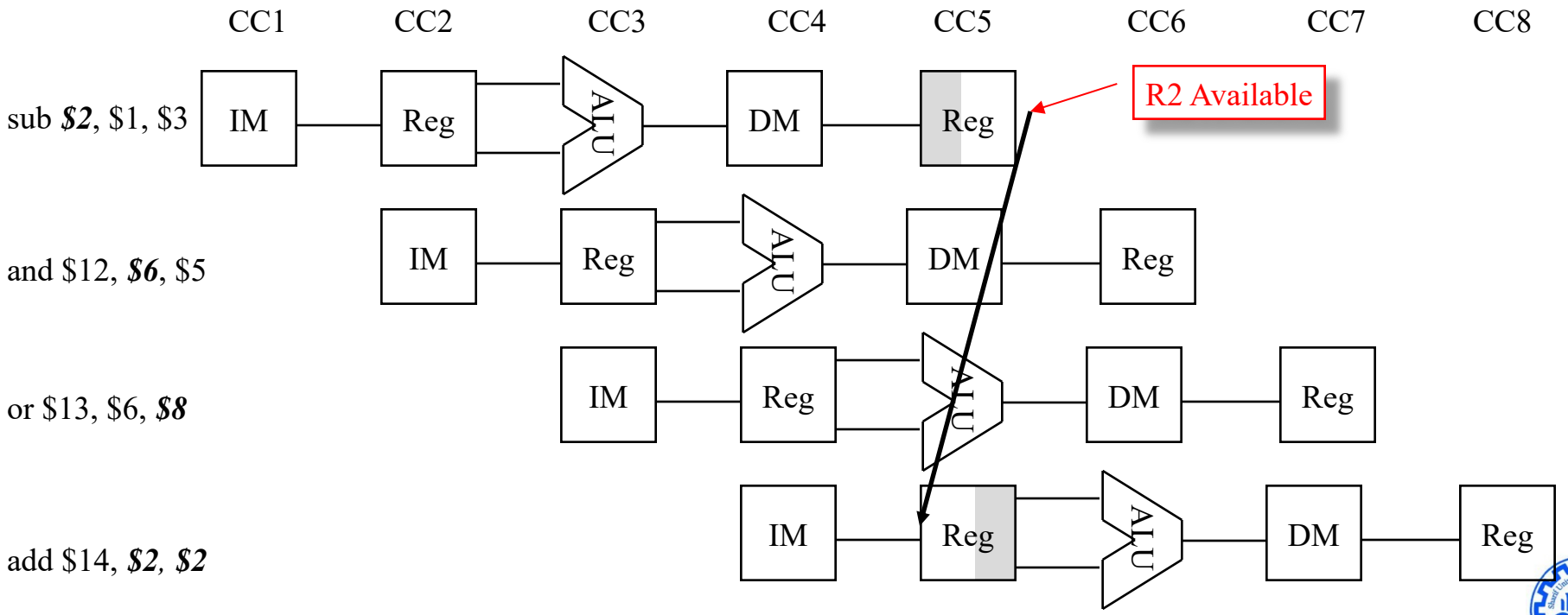
Dealing With Data Hazards

- Reorder code to avoid use of load result in next instruction
- C code for `A = B + E; C = B + F;`

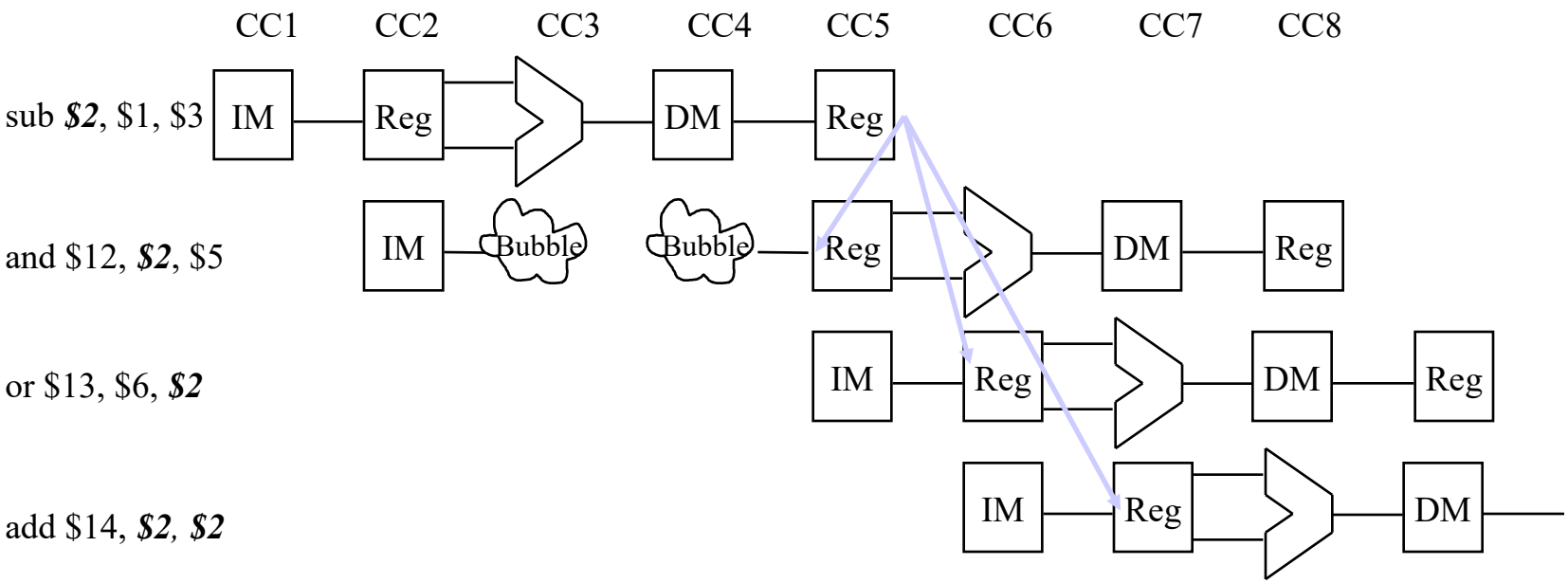


Dealing With Data Hazards

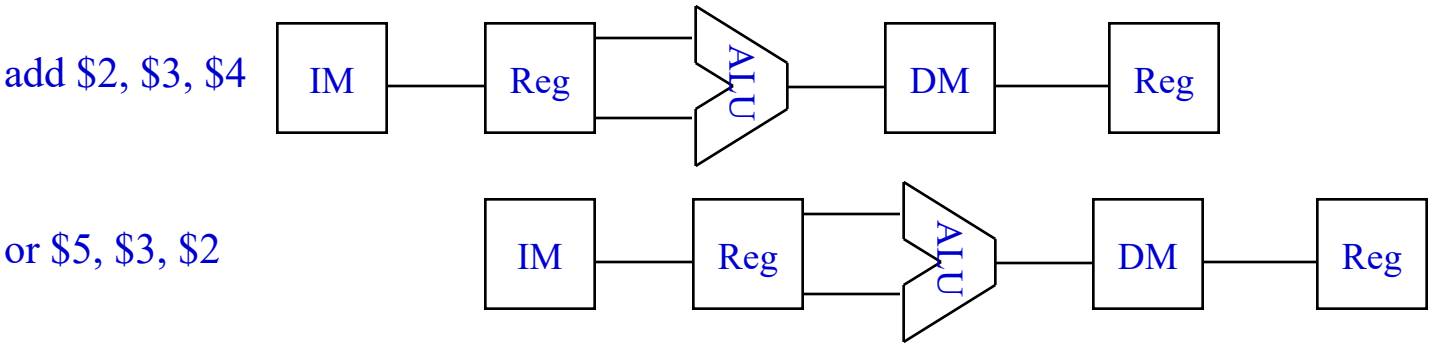
- Using Transparent Register File
  - Use latches rather than flip-flops in RF
    - First half-cycle : register loaded
    - Second half-cycle: new value is read into pipeline state



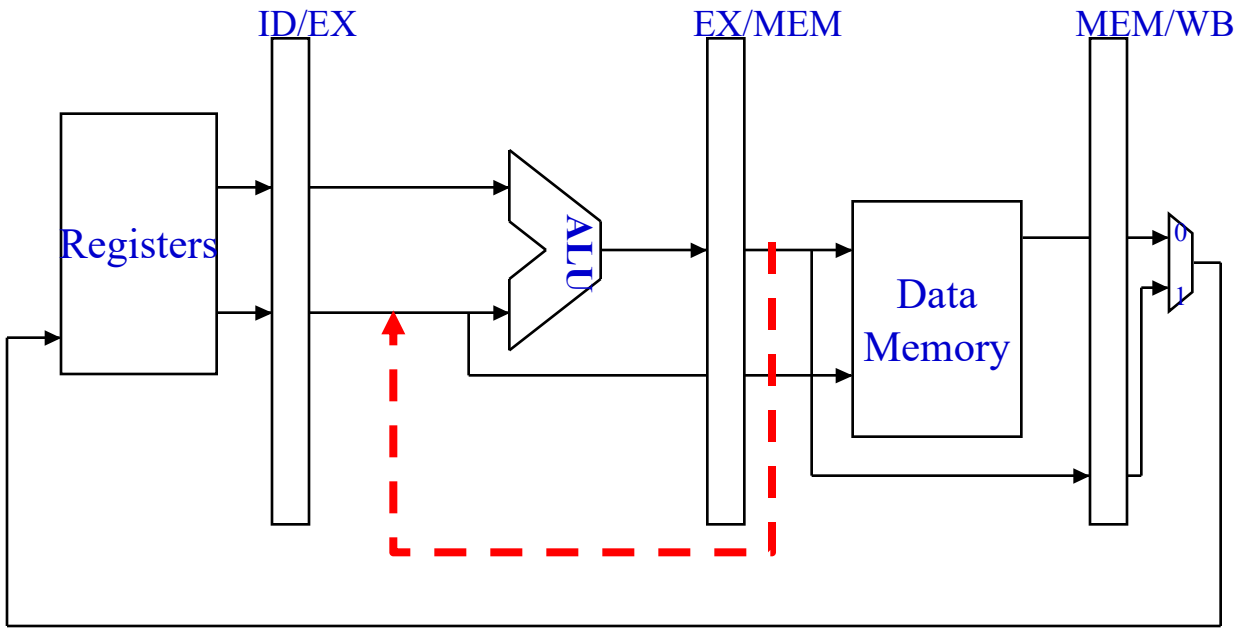
Handling Data Hazards in Hardware: Stall pipeline



Reducing Data Hazards Through Forwarding



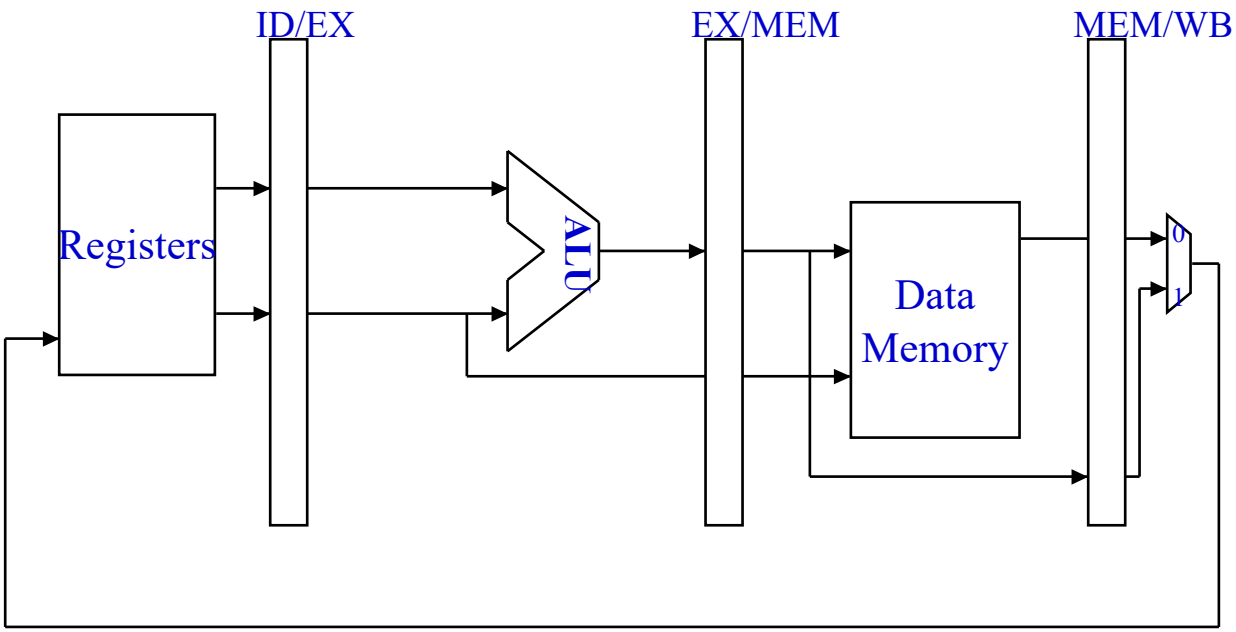
Avoid stalling by forwarding ALU output from "add" to ALU input for "or"



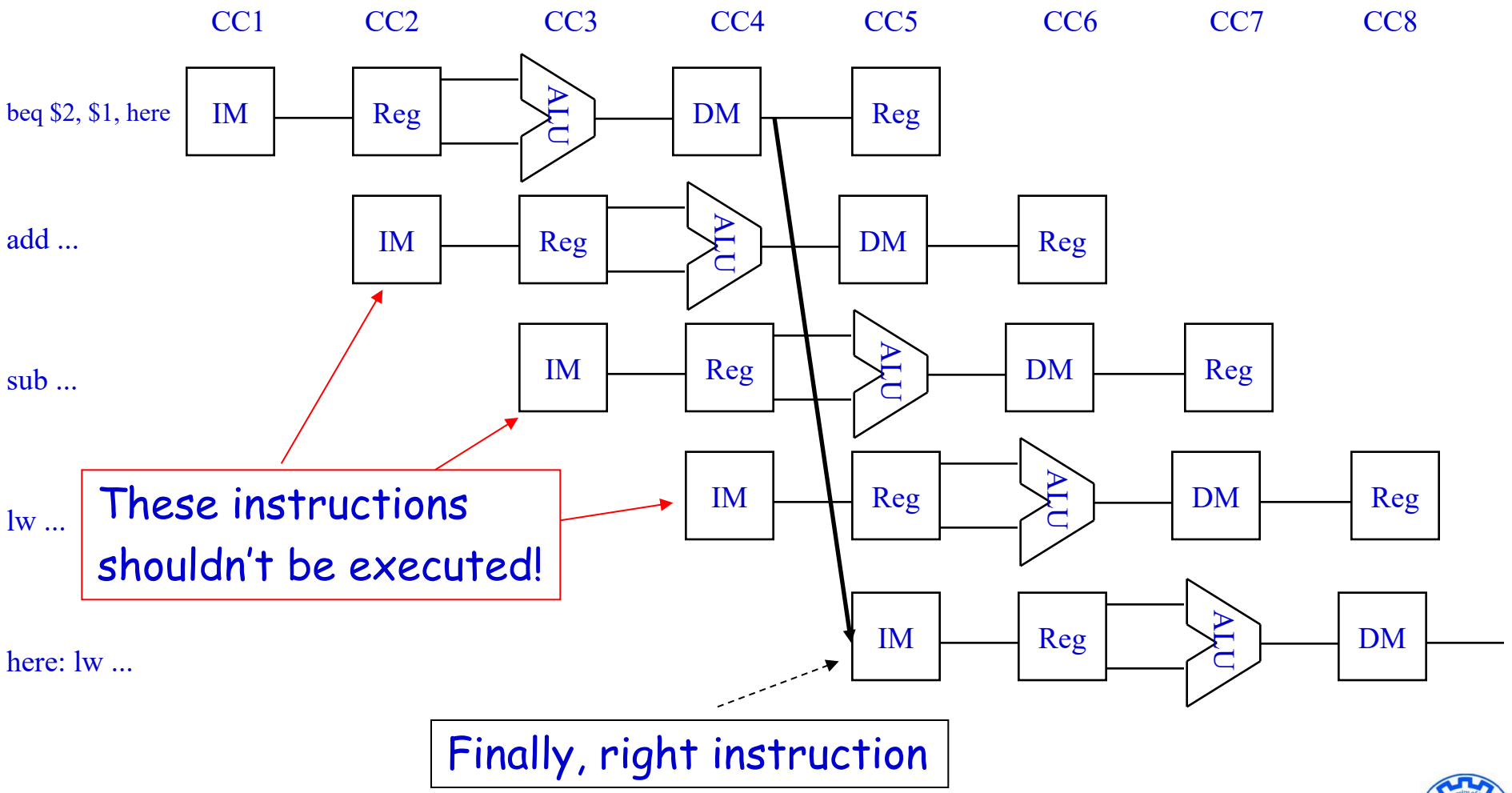
Practice

- Consider the following code:
  - What types of data forwarding are required?

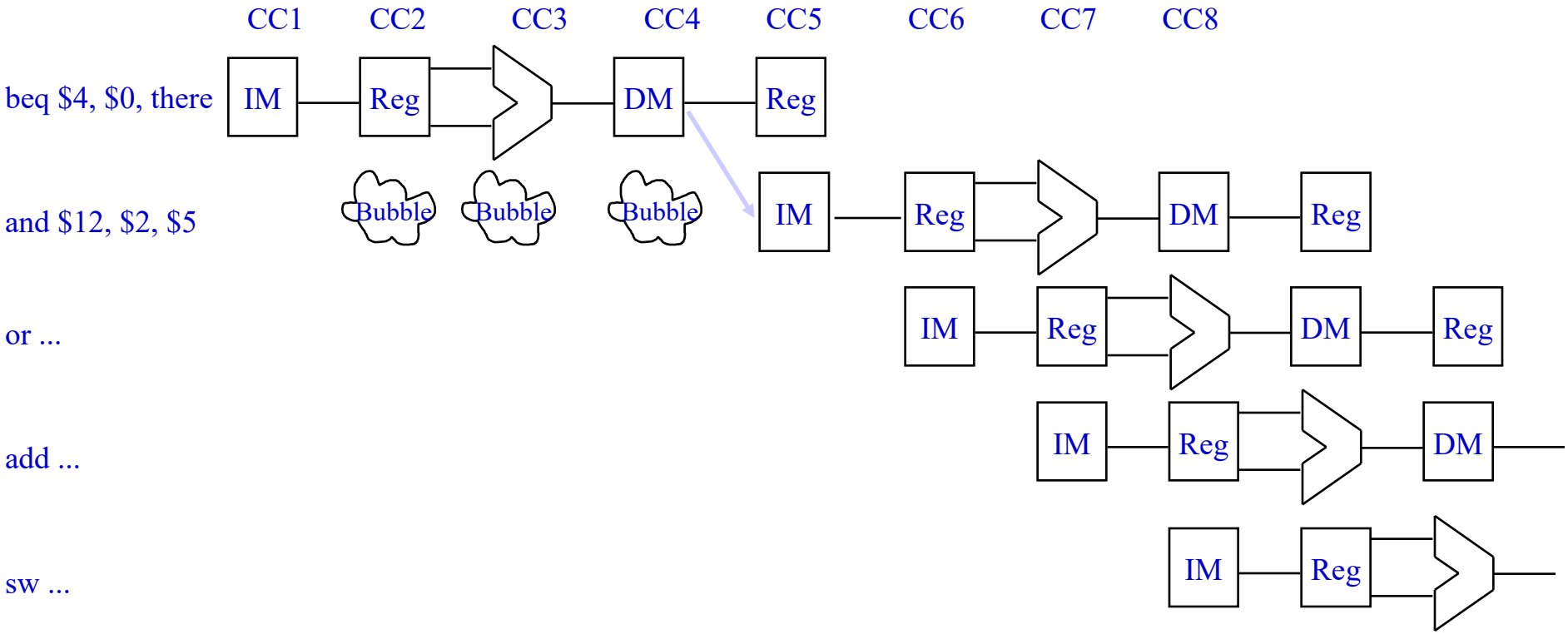
sub ***$2***, $1, $3
and $12, ***$2***, $5
or $13, $6, ***$2***
add $14, ***$2***, ***$2***
sw $15, 100(***$2***)



Control or Branch Hazard

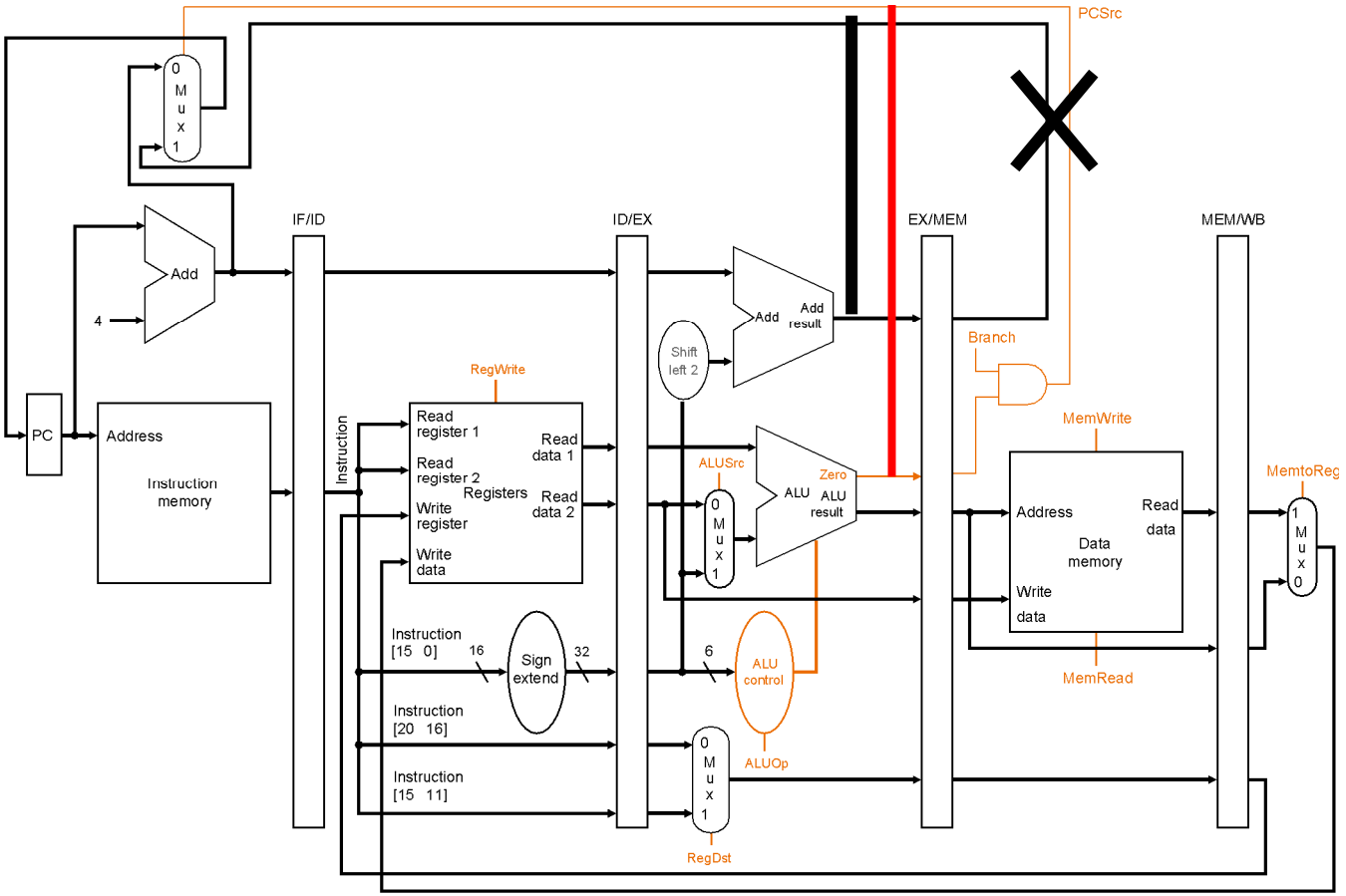


Stalling for Branch Hazards

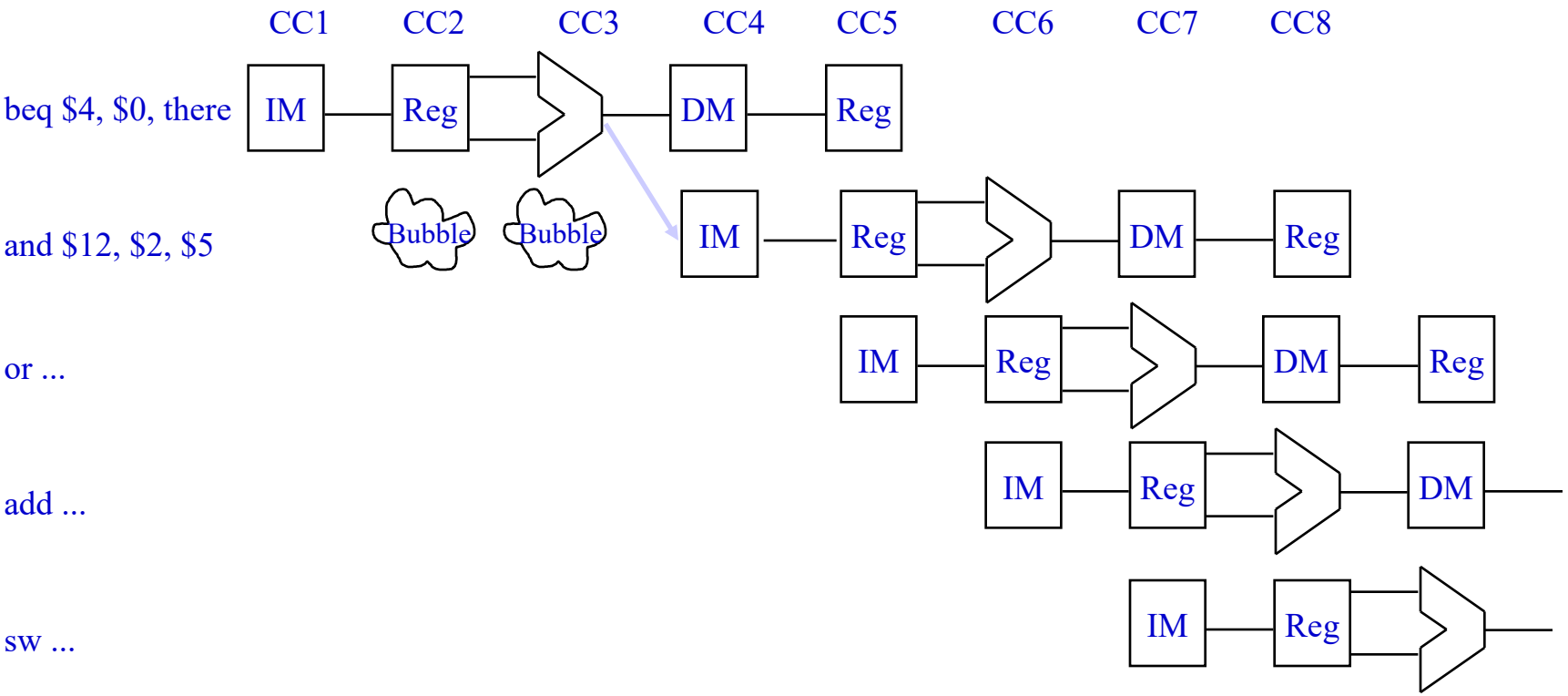


Reducing Branch Delay

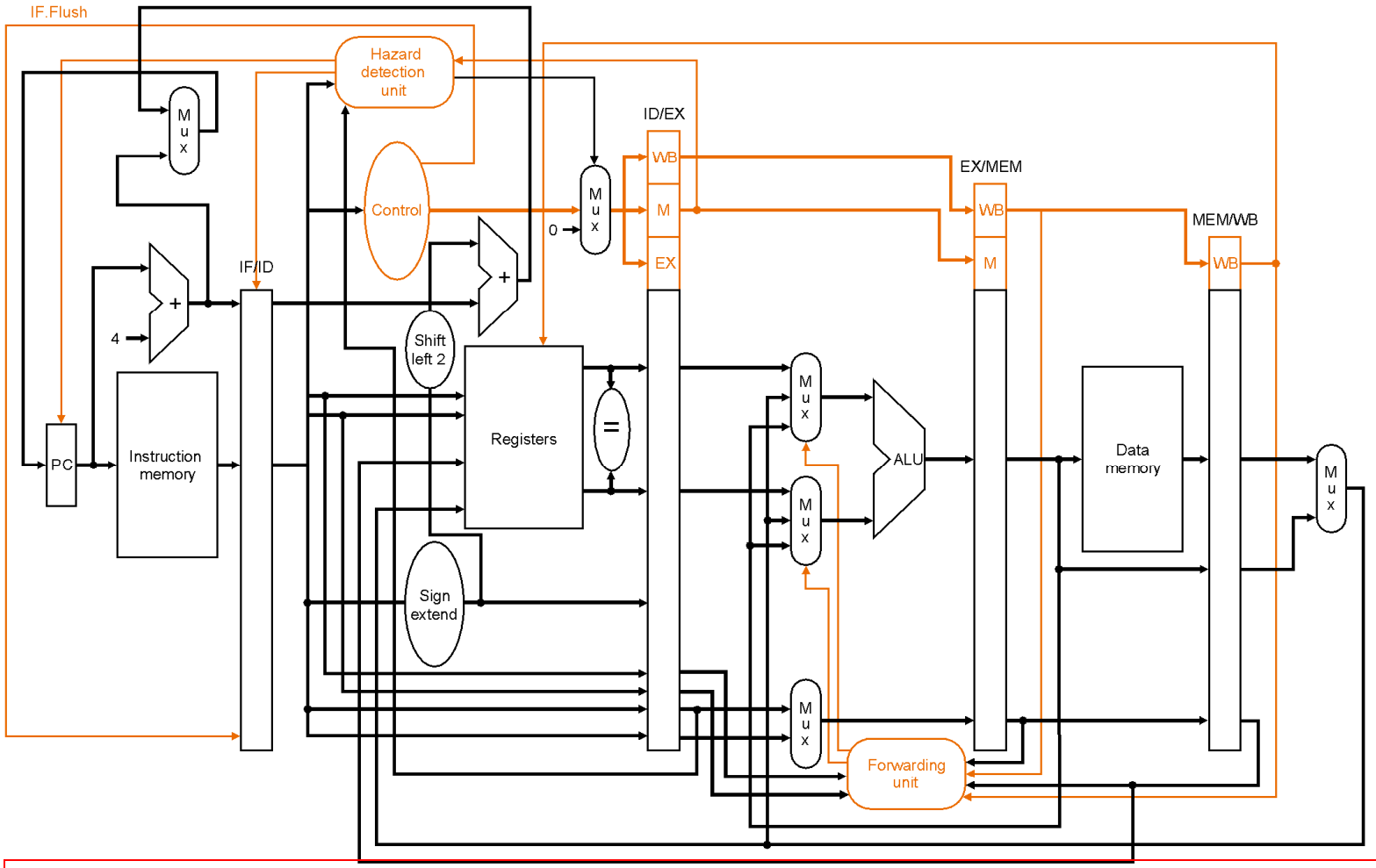
It's easy to reduce stall to 2-cycles



Stalling for Branch Hazards

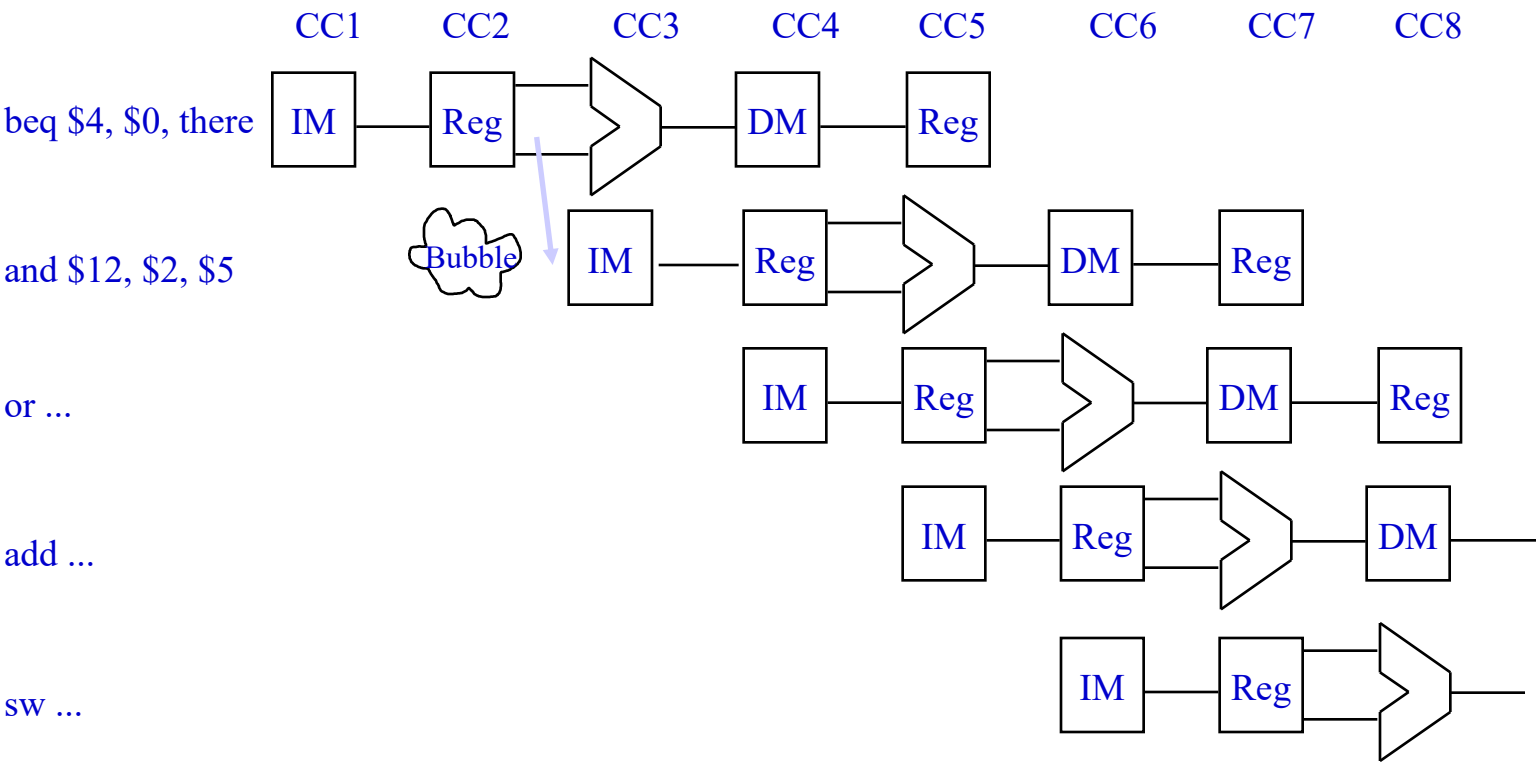


One-Cycle Branch Misprediction Penalty



- Target computation & equality check in ID phase

Stalling for Branch Hazards



Practice

- Which program has more IC?
- Which one has less bubbles?
- Which one runs faster?
- How many clock cycles?

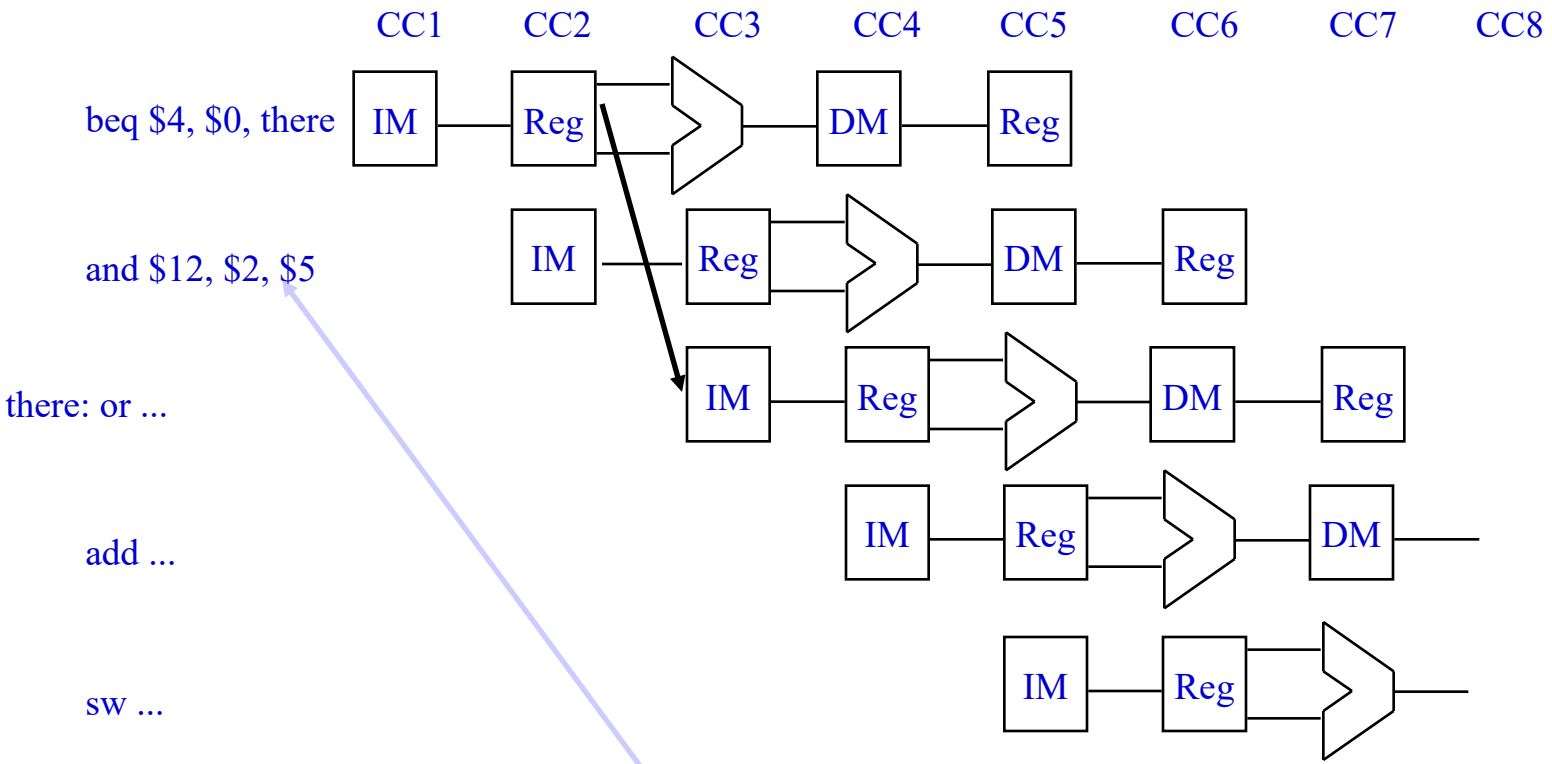
```
        move $s0,$zero
        li $s1, 100
L1:     beq $s0, $s1, L2
        add $s2,$s2,$s0
        addi $s0,$s0,1
        j L1
L2:
```

```
        move $s0,$zero
        li $s1, 100
L1:     add $s2,$s2,$s0
        addi $s0,$s0,1
        bne $s0, $s1, L1
L2:
```


Eliminating Branch Stall

- SPARC and MIPS
  - Use a single *branch delay slot* to eliminate single-cycle stalls after branches
  - Instruction after a conditional branch is *always executed* in those machines
    - Regardless of whether branch is taken or not!

Branch Delay Slot



Branch delay slot instruction (next instruction after a branch) is executed even if the branch is taken.

Filling Branch Delay Slot

```
add  $5, $3, $7
sub  $6, $1, $4
and  $7, $8, $2
beq $6, $7, there
nop   /* branch delay slot */
add  $9, $1, $2
sub  $2, $9, $5
...
there:
mult $2, $10, $11
```

More-Realistic Branch Prediction

- **Static** Branch Prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- **Dynamic** Branch Prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue trend
    - When wrong ➔ stall while re-fetching & update history

Correlated Branches

- Two Types of branch
  - No correlation
  - Simple correlation
  - Complex correlation

Code A

```
for (i=0; i++; i<n)
   sum+=i;

for (j=0; j++; j<m)
   sum+=2*j;
```

Code B

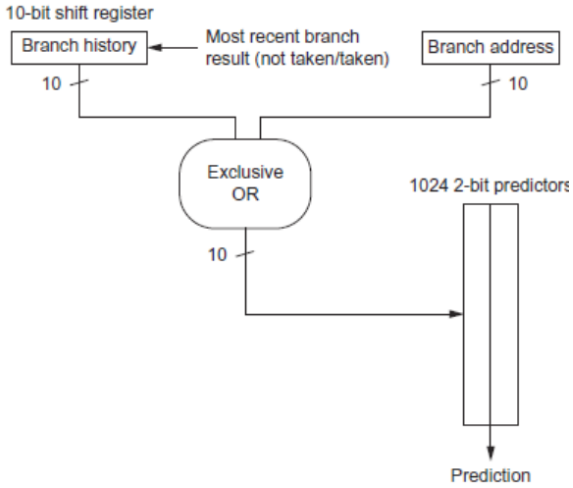
```
for (i=0; i++; i<n)
   for (j=0; j++; j<i)
      sum+=i+j;
```

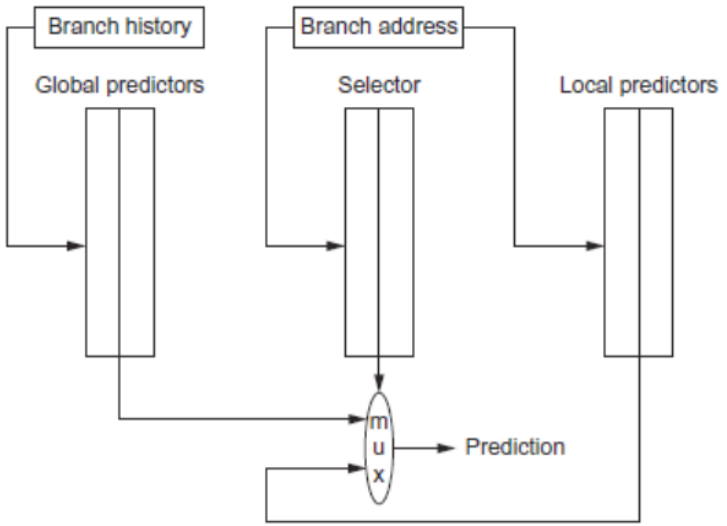
Code C

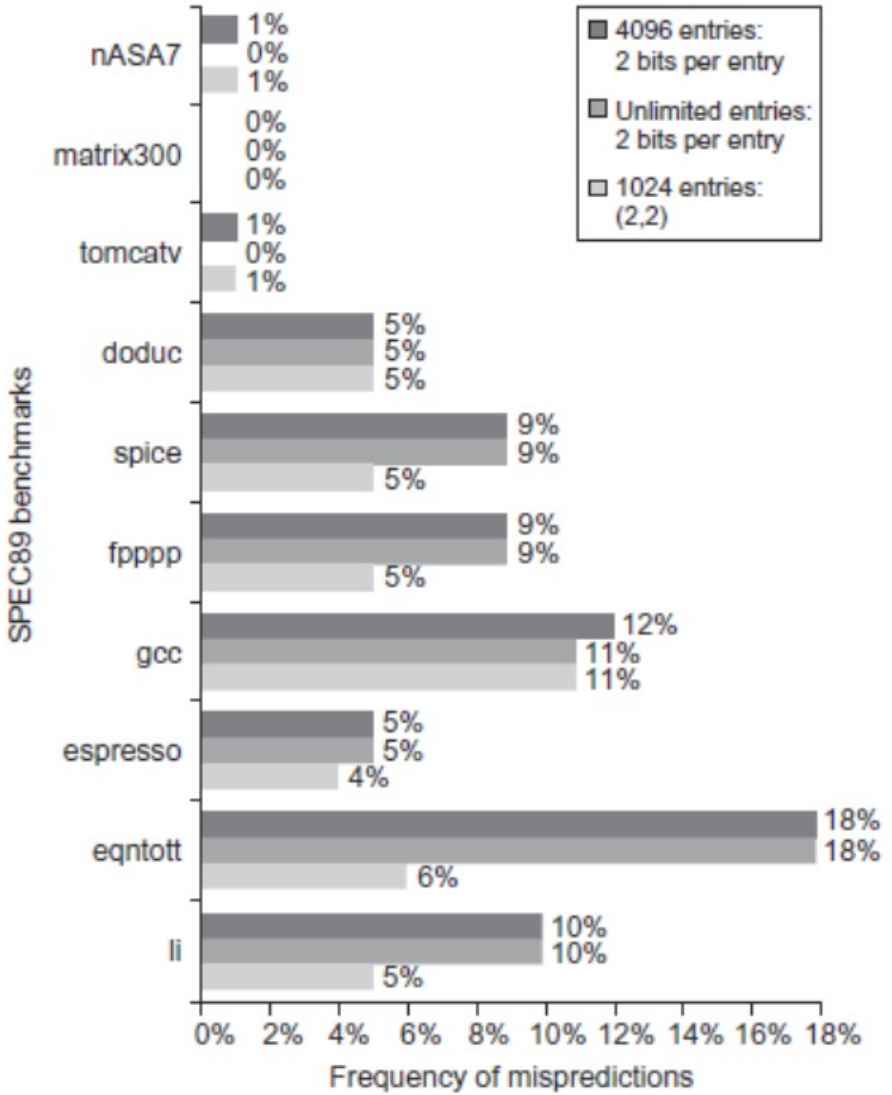
```
If (aa==2)
   aa=0;
If (bb==2)
   bb=0;
if (aa!=bb)
   cc = 1;
```

Branch Prediction

- Different Configurations ➔ Different mispredictions







Compiler Techniques

- Compiler Techniques to Mitigate Branch Misspenalty
  - Loop unrolling
  - Macro (instead of procedure call)

```
for (i=0; i++; i<n)
    for (j=0; j++; j<2)
        sum+=i+j;
```

Thanks for Your Attention!