Pipelining*

Maurizio Palesi

* Adapted from David A. Patterson's CS252 lecture slides,

http://www.cs.berkeley/~pattrsn/252S98/index.html

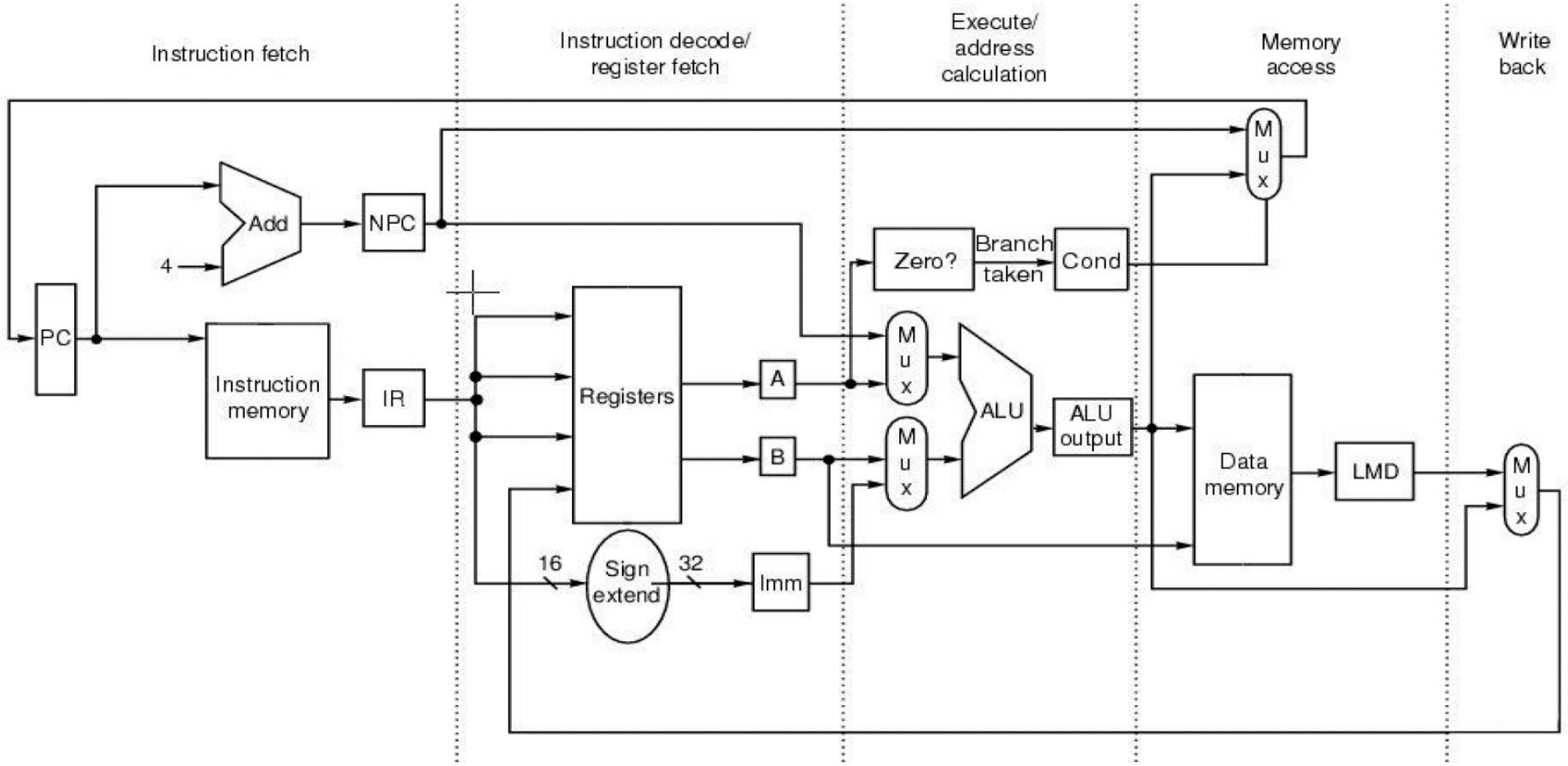
References

- John L. Hennessy and David A. Patterson, *Computer Architecture a Quantitative Approach*, second edition, Morgan Kaufmann
  - ➔Chapter 3

Basic Steps of Execution in DLX

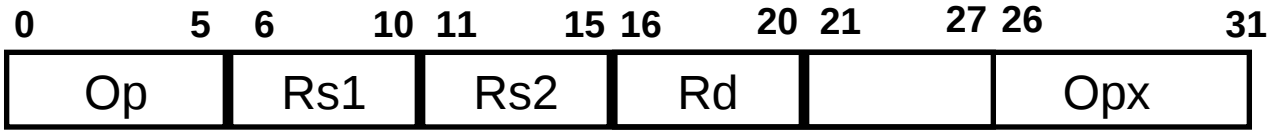
- 1. Instruction fetch step (IF)
- 2. Instruction decode/register fetch step (ID)
- 3. Execution/effective address step (EX)
- 4. Memory access/branch completion step (MEM)
- 5. Register write-back step (WB)

DLX Datapath



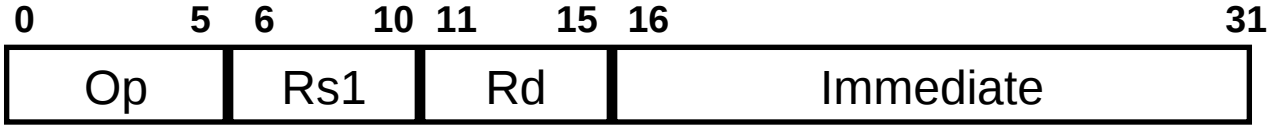
DLX Instruction Format

Register-Register – Register ALU operations



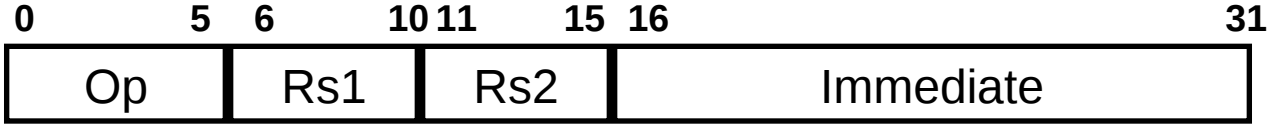
R-Type

Register-Immediate – Load/Store, ALU immed, Jump



Branch

I-Type

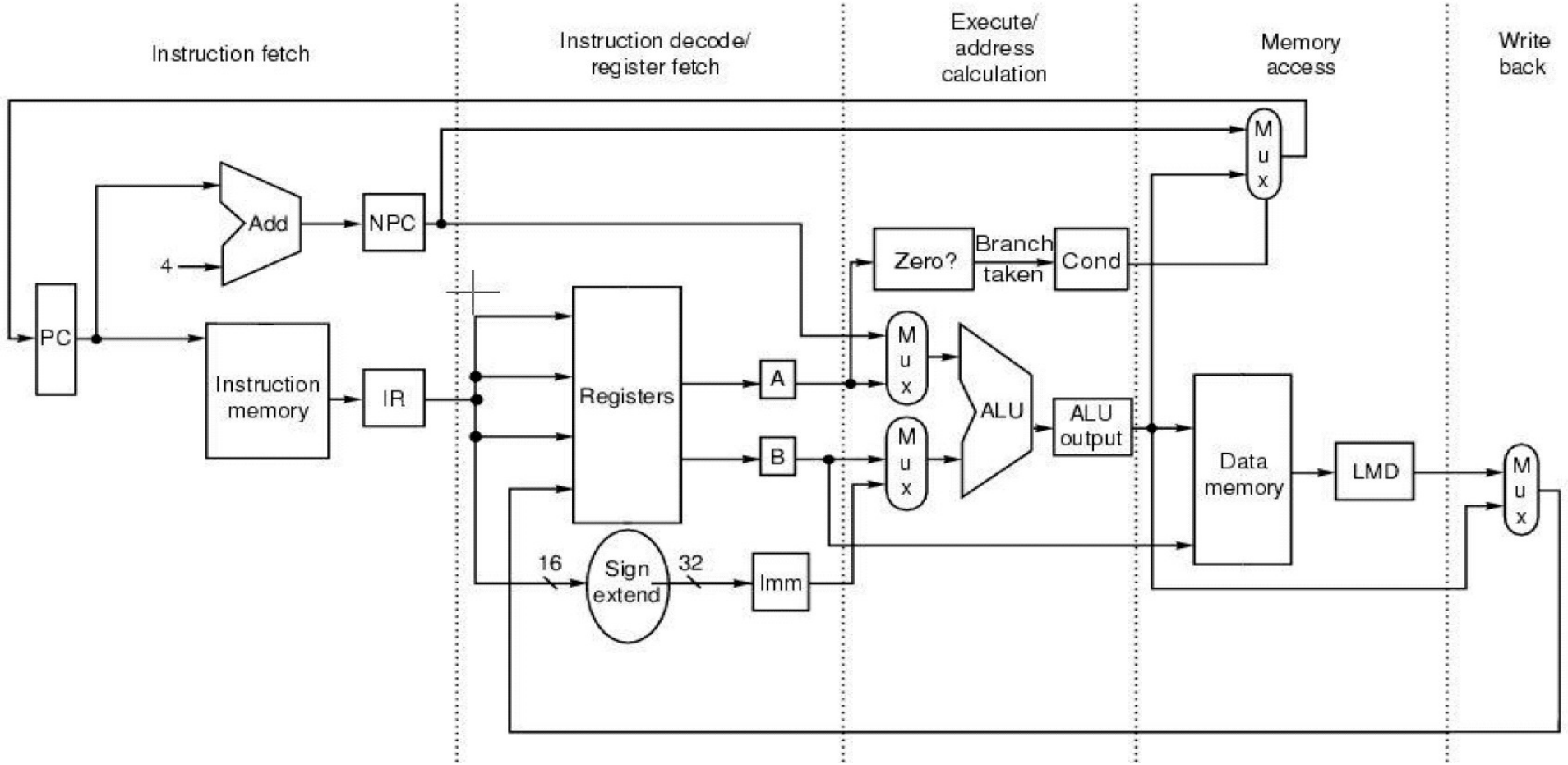


Jump / Call

J-Type

Basic Steps of Execution

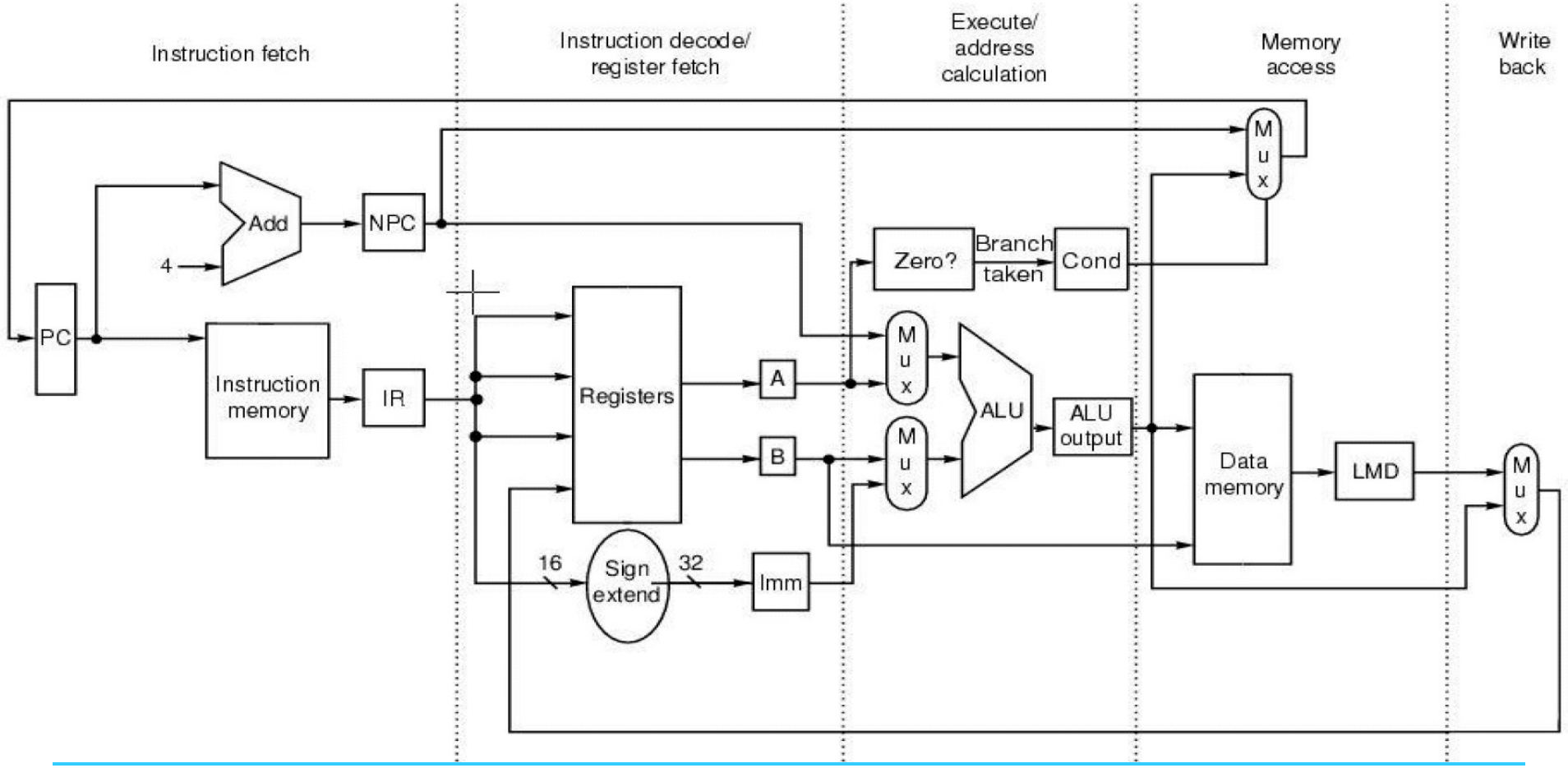
1. Instruction fetch (IF) IR ← Mem[PC], NPC ← PC + 4



Basic Steps of Execution

2. Instruction decode/register fetch (ID)

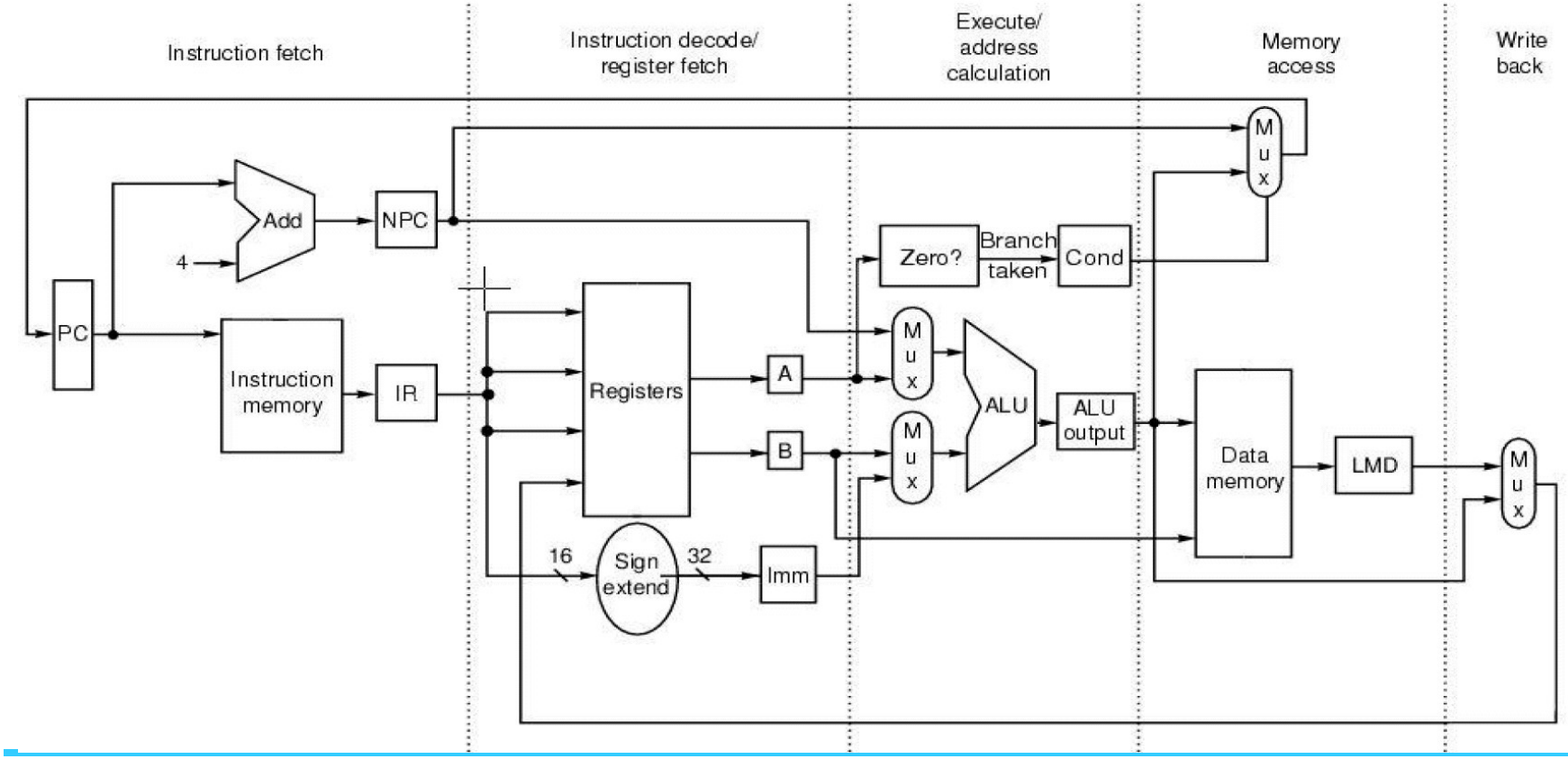
A ← Regs[IR(6:10)], B ← Regs[IR(11:15)]
Imm ← $(IR(16)^{16}\#IR(16:31))$



Basic Steps of Execution

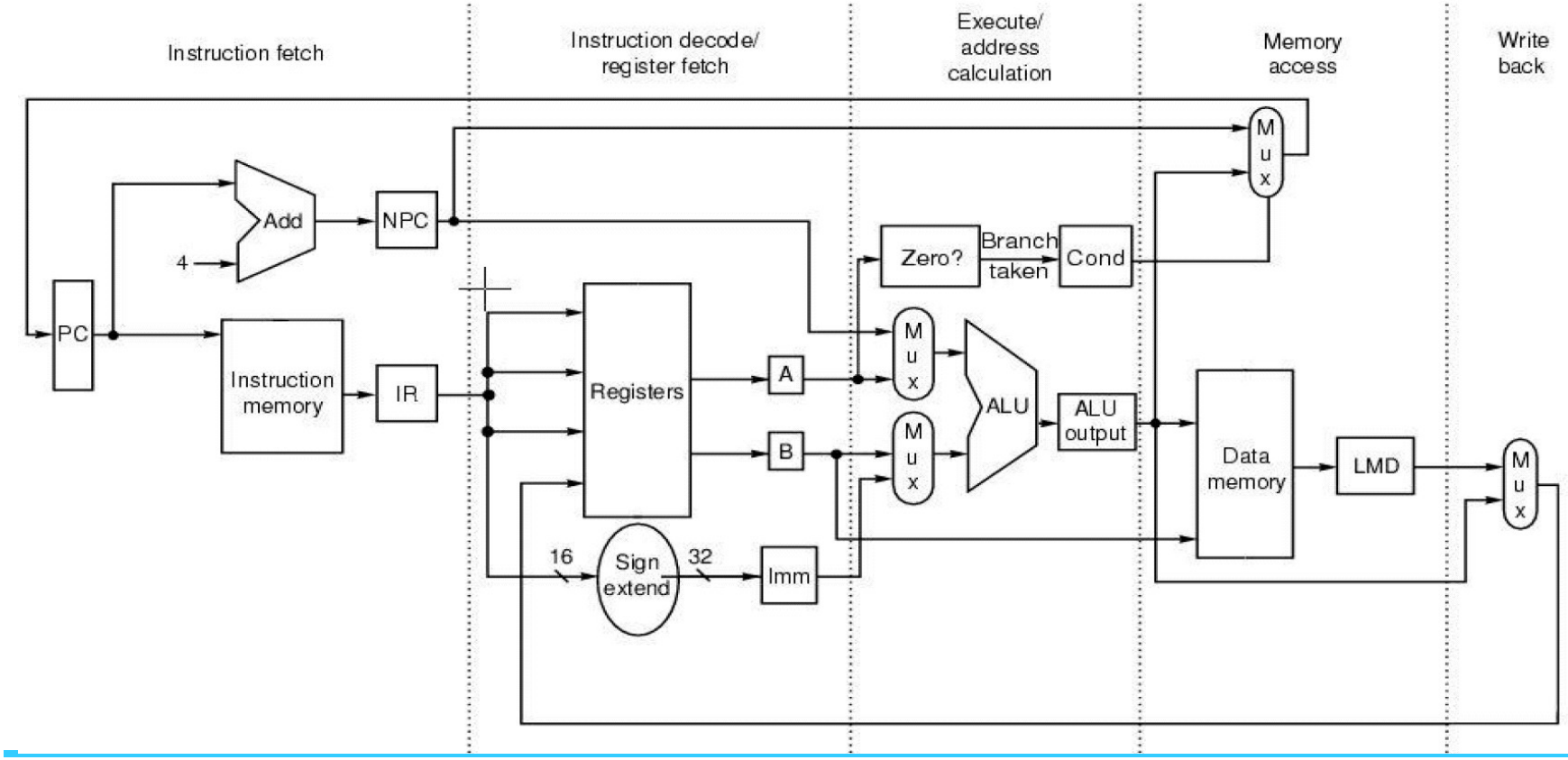
3. Execution/effective adddress (EX)

ALU output ← A + Imm
ALU output ← A func B
ALU output ← A op Imm
ALU output ← NPC + Imm; Cond ← A op 0



Basic Steps of Execution

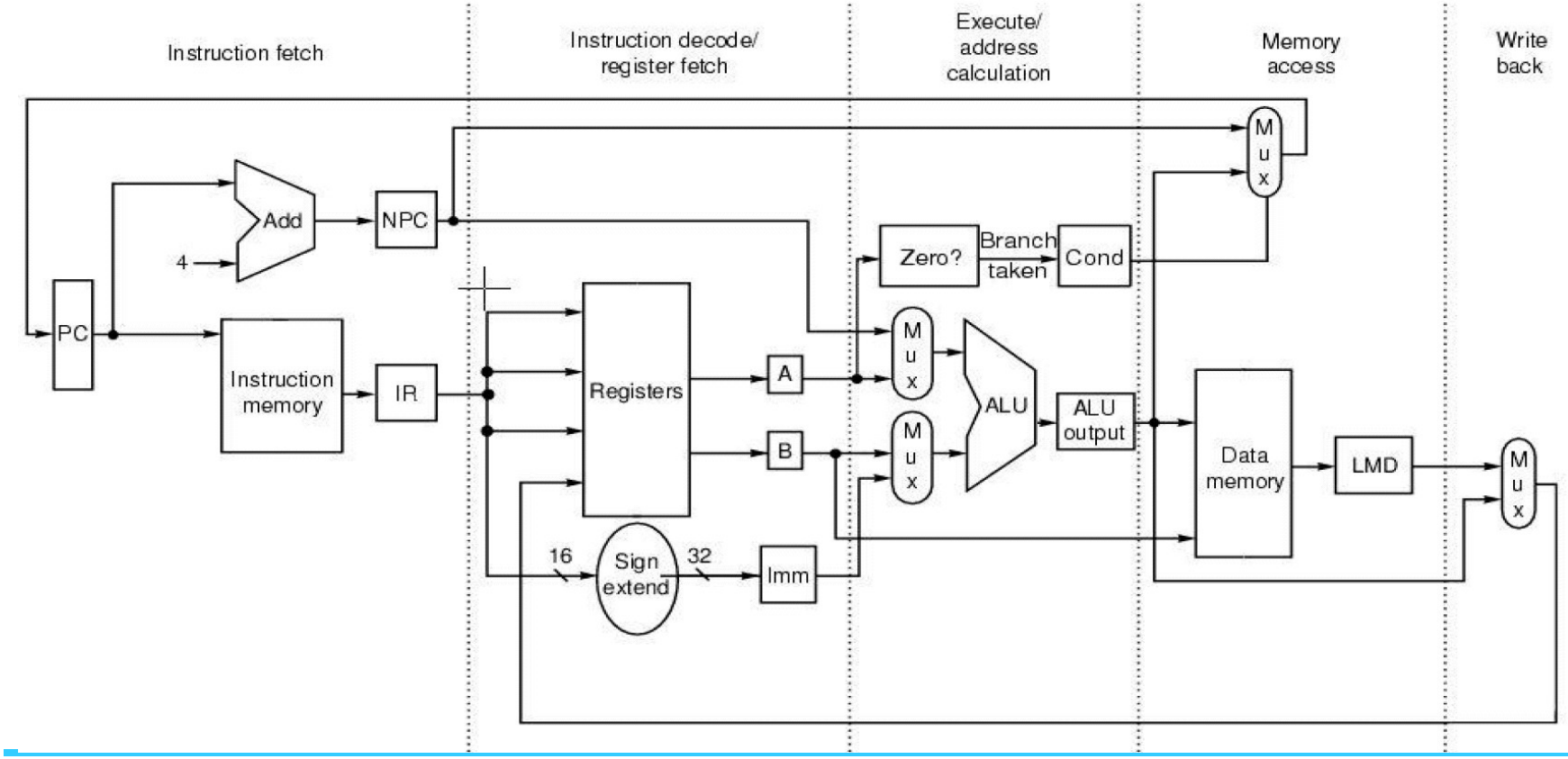
4. Memory access/Branch completion (MEM)
   - PC ← NPC, LMD ← Mem[ALU output] or Mem[ALU output] ← B
   - if (cond) PC ← ALU ouput



Basic Steps of Execution

5. Register write-back (WB)

Regs[IR(16:20)] ← ALU output
Regs[IR(11:15)] ← ALU output
Regs[IR(11:15)] ← LMD

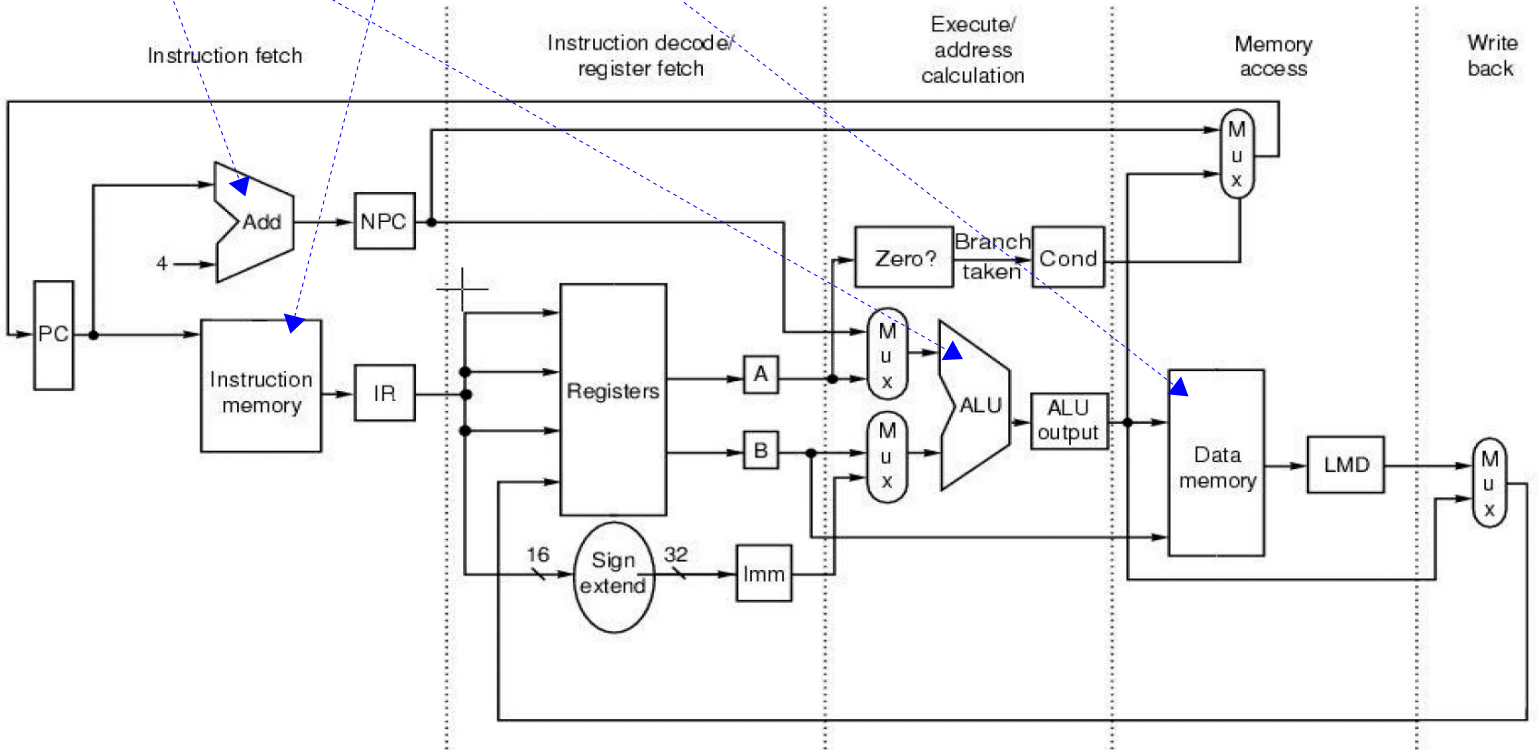


Discussion

- Branch & Store instructions: 4 cycles
- All other instructions: 5 cycles
- Example
  - Branch 12%, Store 5%
  - CPI = (12% + 5%)*4 + 83%*5 = 4.83
- Improvement
  - Complete ALU instruction during MEM
  - Ex., 47% ALU instructions
    - CPI = (12% + 5% + 47%)*4 + 36%*5 = 4.36
    - Speedup = 4.83/4.36 = 1.1
- Any other attempts to decrease CPI may increase the clock cycle time

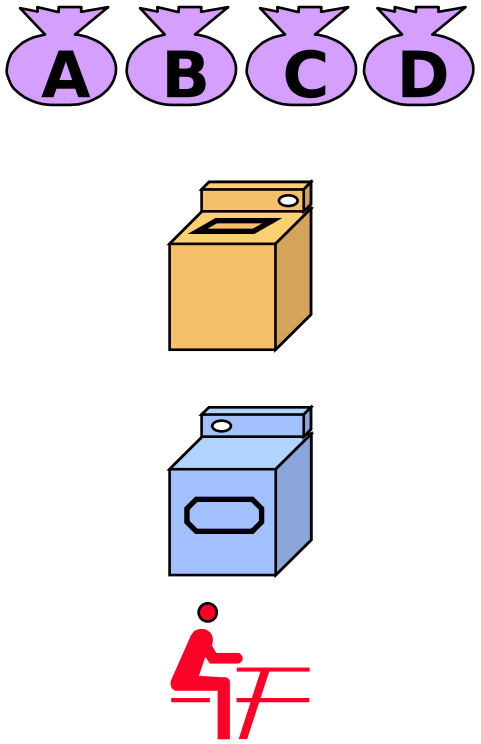
Multicycle Implementation

- A simple FSM could be used to implement the control logic
  - ➔ Microcode control could be used for a much more complex machine
- Hardware redoundancies
  - ➔ Two ALU
  - ➔ Separate instruction and data memories

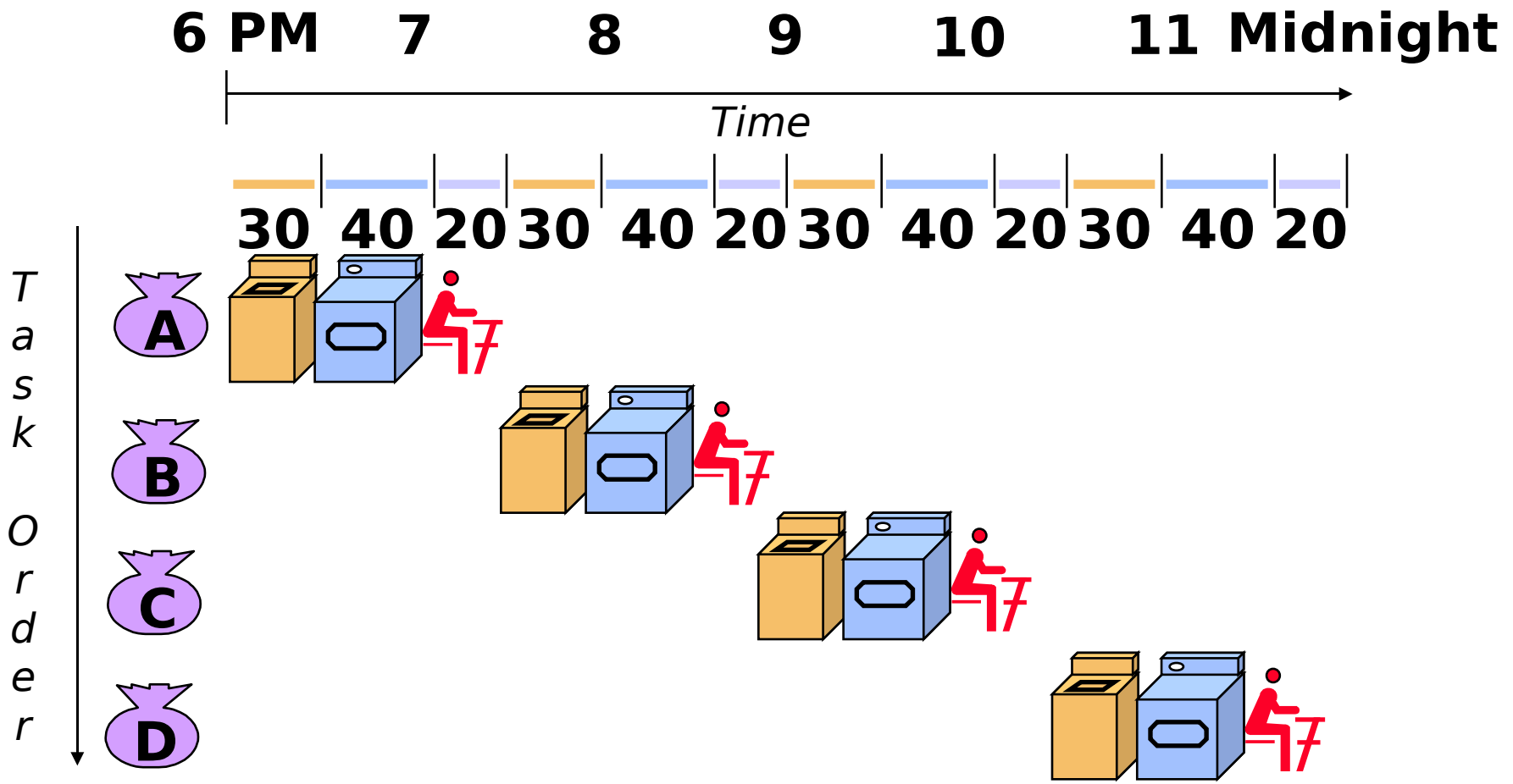


Pipelining: Its Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes

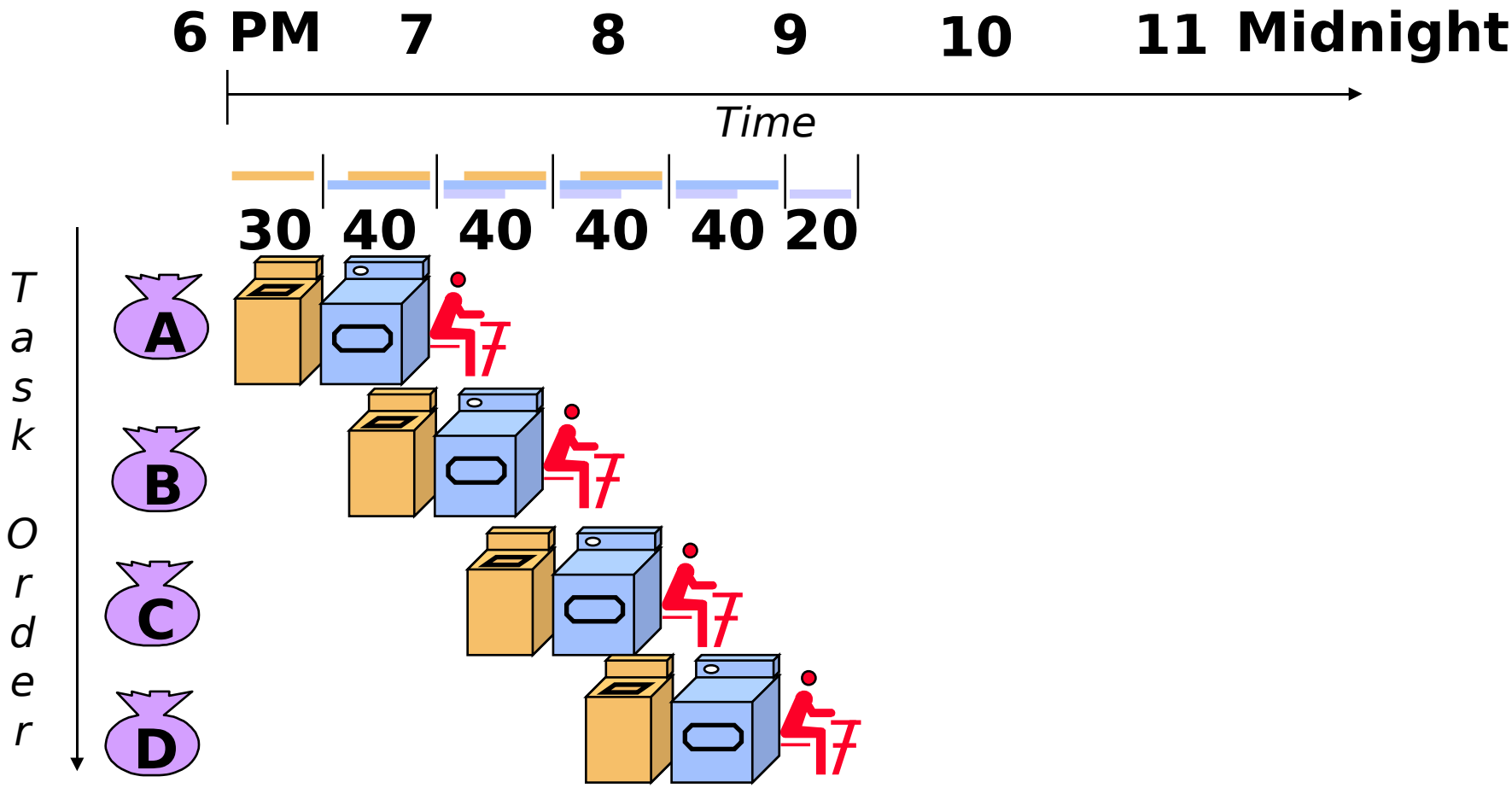


Sequential Laundry



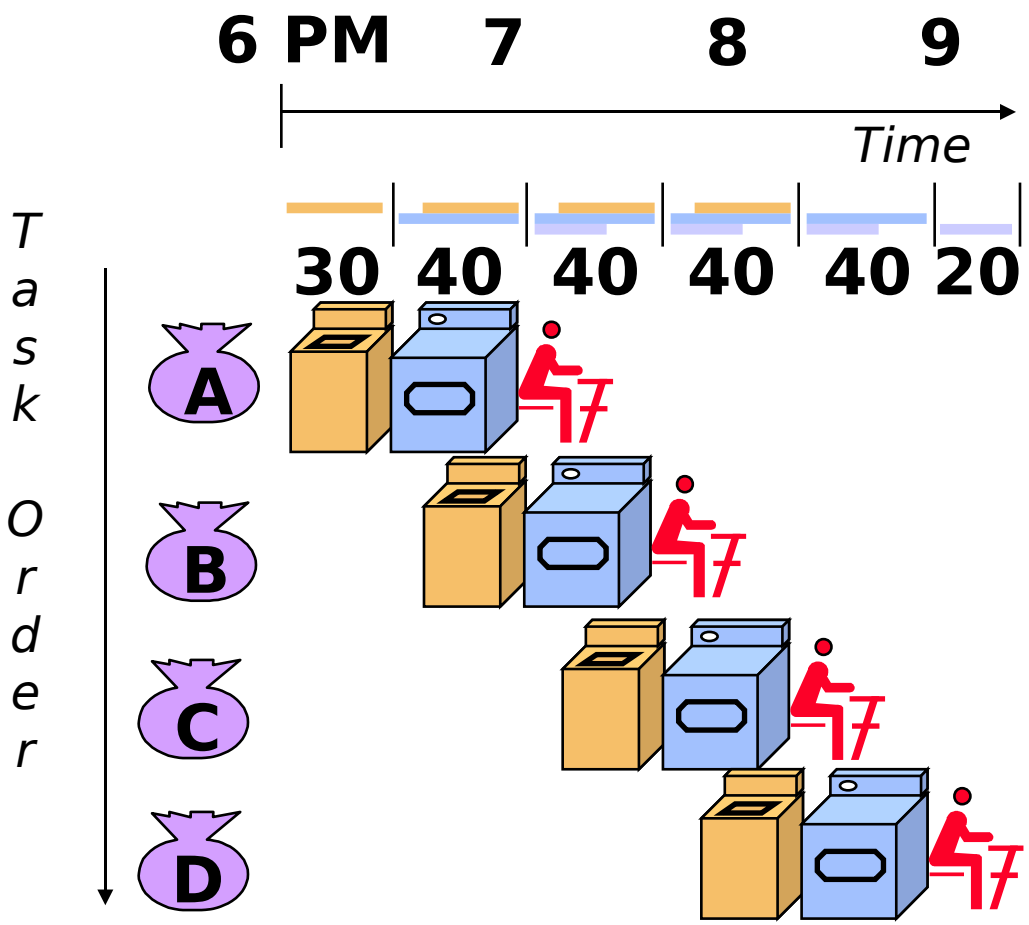
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



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

Pipelining

- Technique for having multiple instructions execute in an overlapped manner
- Assembly Line
- Throughput
  - ➔Determined by how often an instruction exits the pipeline
- Time to move one step is machine cycle time
  - ➔Determined by the slowest step in the pipeline
  - ➔Often it is clock cycle though clocks may have multiple phases
- Length of Pipe - No of stages
  - ➔Determine latency

Pipeline Performance

- Under ideal conditions
  - Time per instruction = Time per instruction on non-pipelined machine/Number of pipe stages
  - Speedup = Number of stages
- But…
  - Stages are not balanced
  - Overhead (10%)

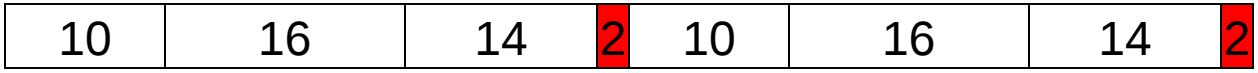
Basic Performance Issues

- Pipeline increases instruction throughput
  - ➔It does not decrease the time of execution of any single instruction
    - ✓It may increase it!
- Stage imbalance may yield further inefficiencies
- Overheads due to
  - ➔Register and latches adding to delays and clock skew

Pipeline Example: 3 stages

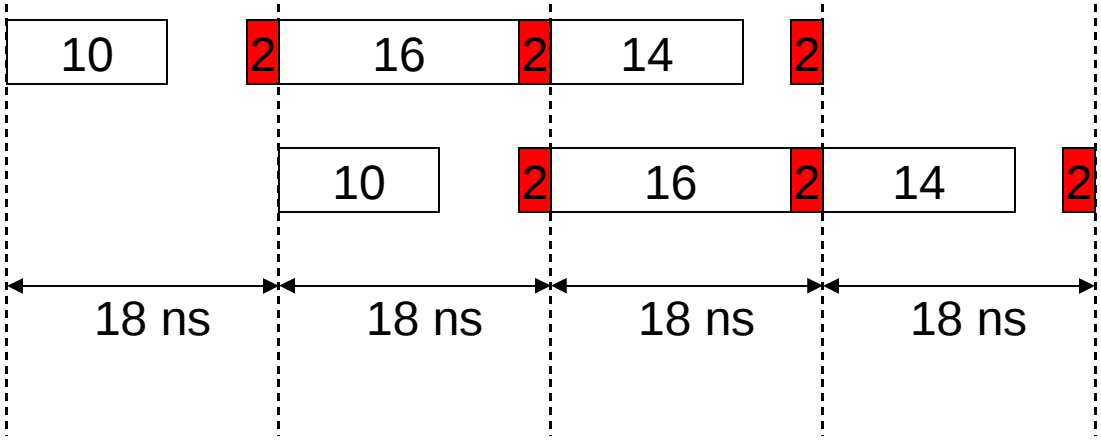
Assume 2 nsec latch delay

Unpipelined (mono cycle)



Latency = 42 nsec, Throughput = 1/42

Pipelined



Latency = (3 x 18) = 54 nsec, Throughput = 1/18 (2.3x, not 3x)

CPU Time

$$\text{CPU Time} = \#\,\text{Instructions} \times \text{CPI} \times T_{CK}$$

Strategy	# instructions	CPI	Tck
Single long clock cycle	=	1	big
Multiple cycles	=	many	small
Pipelining	=	1 (ideal)	small

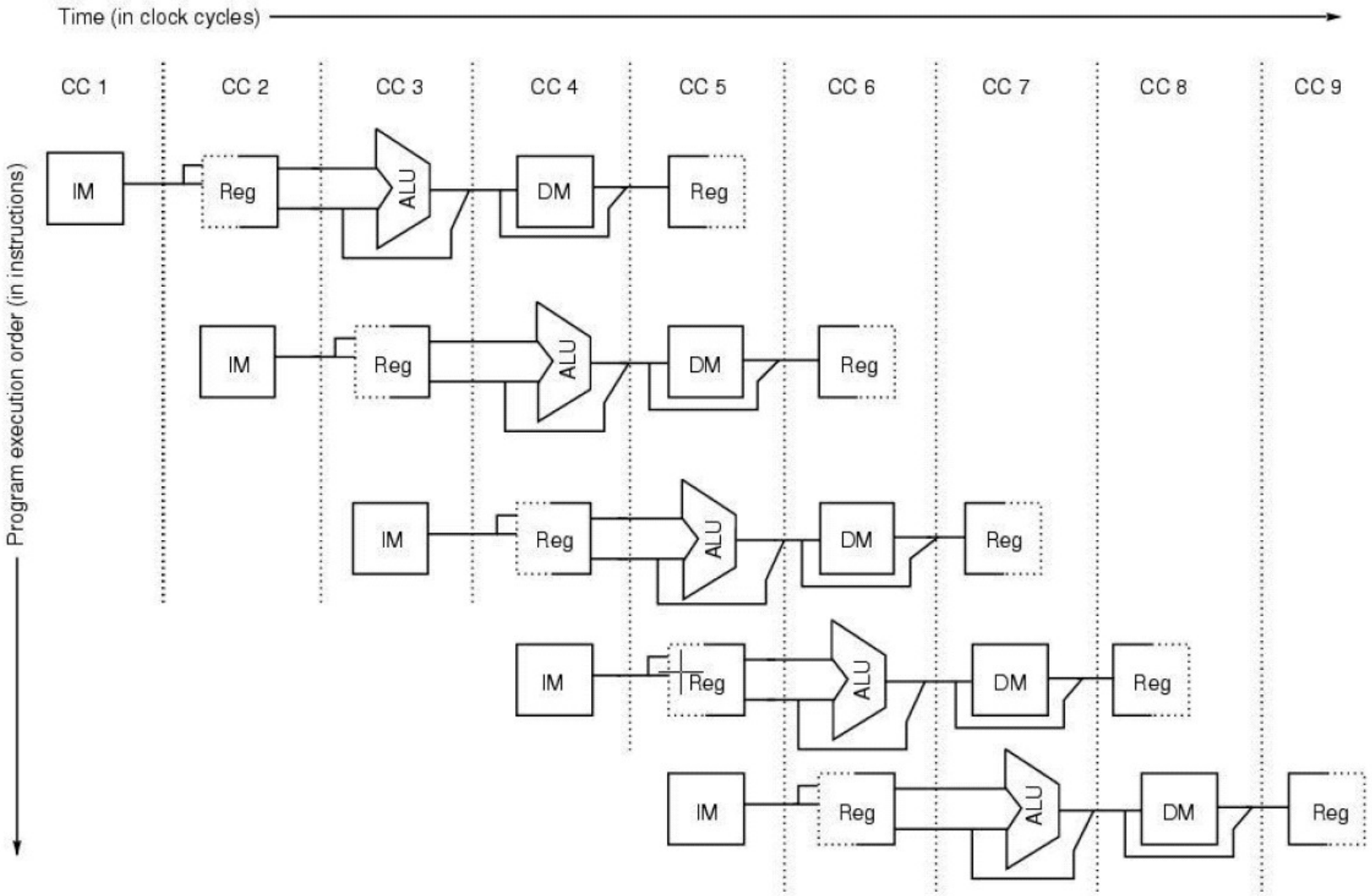
Basic Pipeline DLX

	Clock cycle							
Instruction number	1	2	3	4	5	6	7	8
Instruction i	IF	ID	EX	MEM	WB			
Instruction i+1		IF	ID	EX	MEM	WB		
Instruction i+2			IF	ID	EX	MEM	WB	
Instruction i+3				IF	ID	EX	MEM	WB
Instruction i+4					IF	ID	EX	MEM

Making Pipeline Work

- Determine what happens at each clock tick
- Assure no resource conflicts
  - ➔Can not use one ALU for address calculation and ALU functions
- All operations in a pipe stage must complete in one clock cycle
  - ➔May have to elongate the clock cycle to accommodate this

DLX Pipeline: Datapath



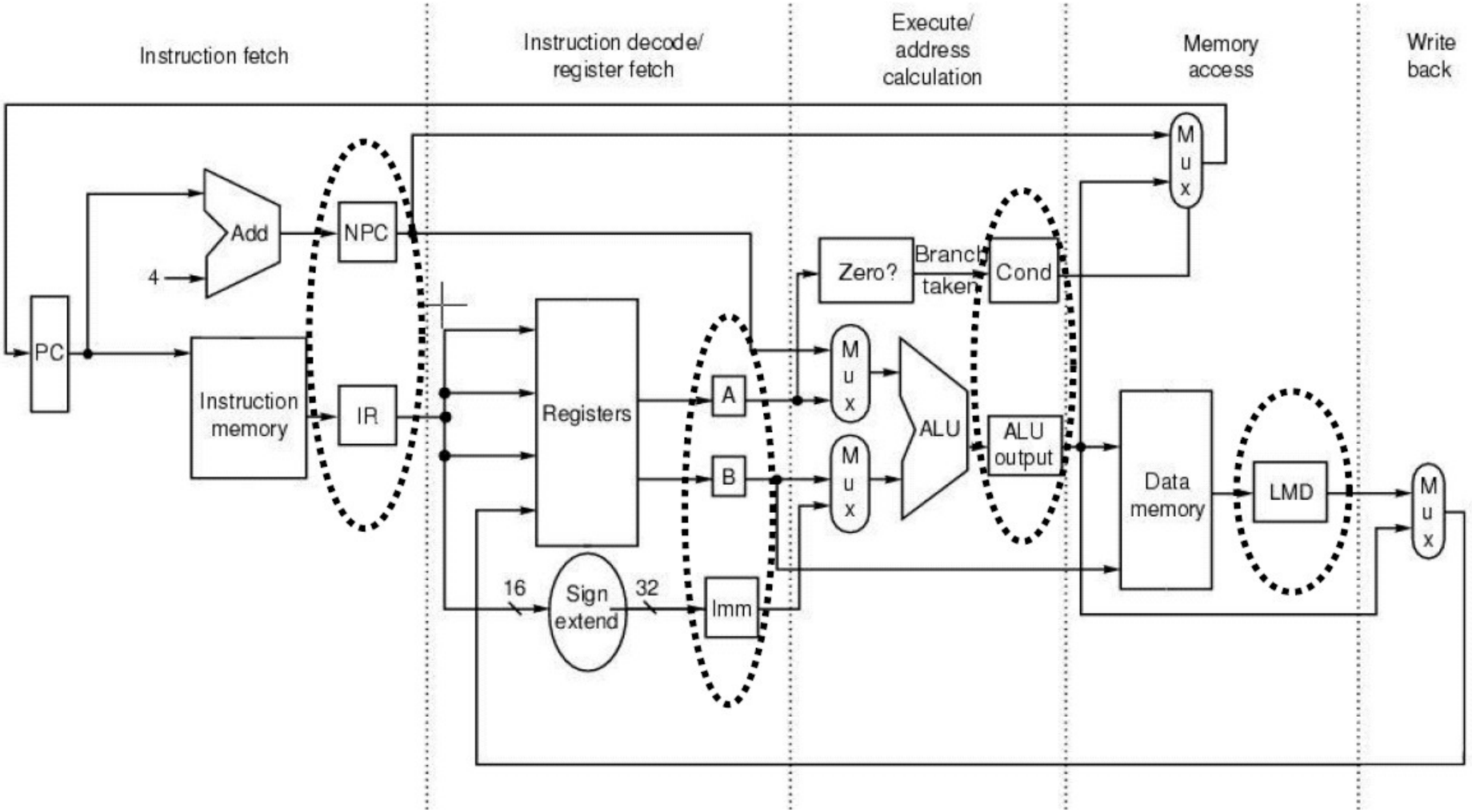
Pipeline Structure

- Major functional units are used in different cycles
- Three observations
  - ➔Basic datapath uses separate instruction and data memory
    - ✓Have separate instruction and data caches
    - ✓Memory bandwidth increases in a pipelined system
  - ➔Register file is used in two stages - ID and WB
  - ➔To start a new instruction every clock we must increment and store the PC every clock cycle during IF stage
    - ✓What happens when branches occur

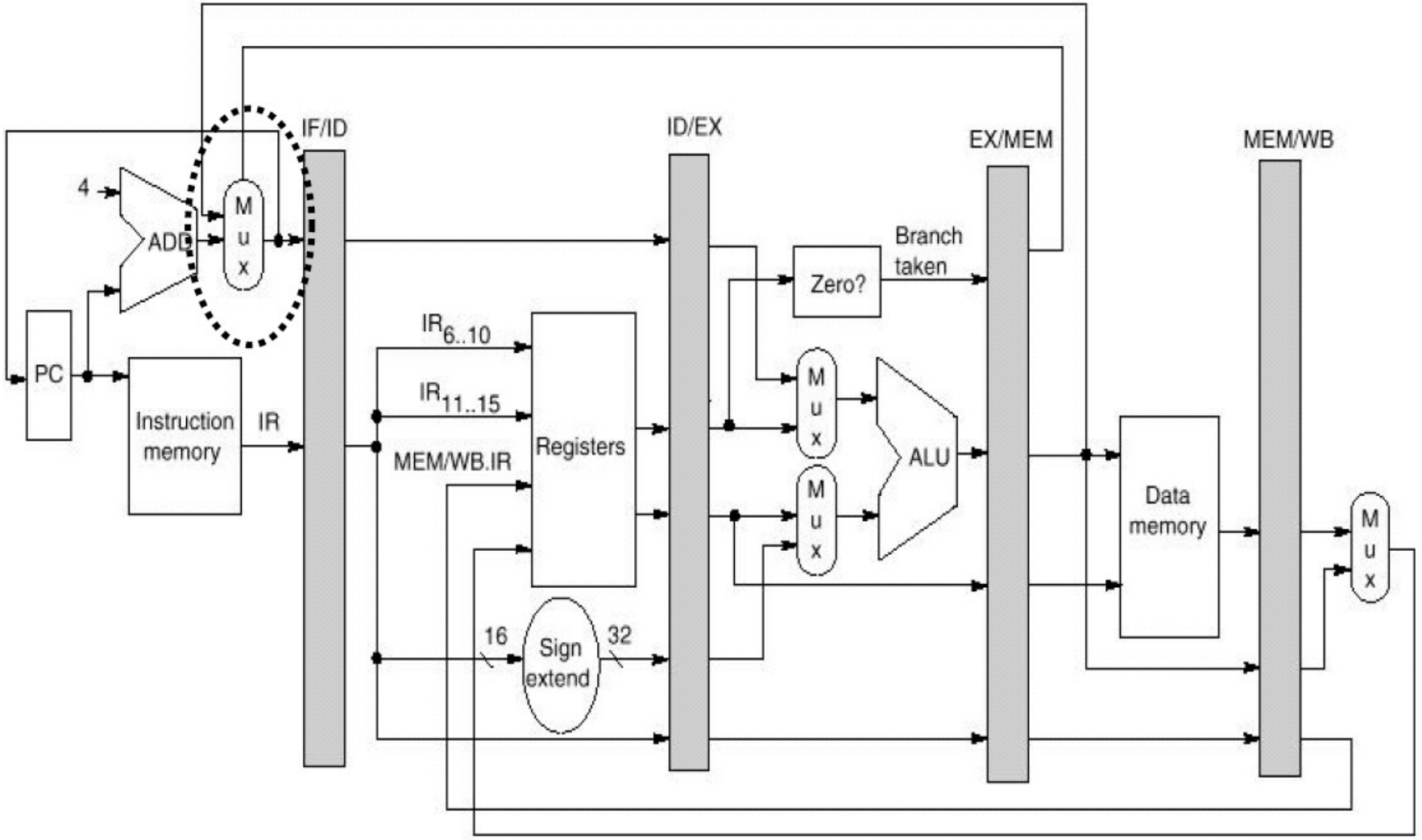
Pipeline Structure

- Every pipe stage is active in every cycle
  - ➔ Values passed from one stage to next must be placed in registers
  - ➔ Use pipeline registers or pipeline latches between stages
- Registers used to transfer information from one stage to the next
- Pipeline registers carry both control and data from stage to stage
- Values may have to be copied from one to the next stage if it is needed at a later stage
- An instruction is active in exactly one stage of the pipe at any moment

DLX Datapath w/o Pipelining

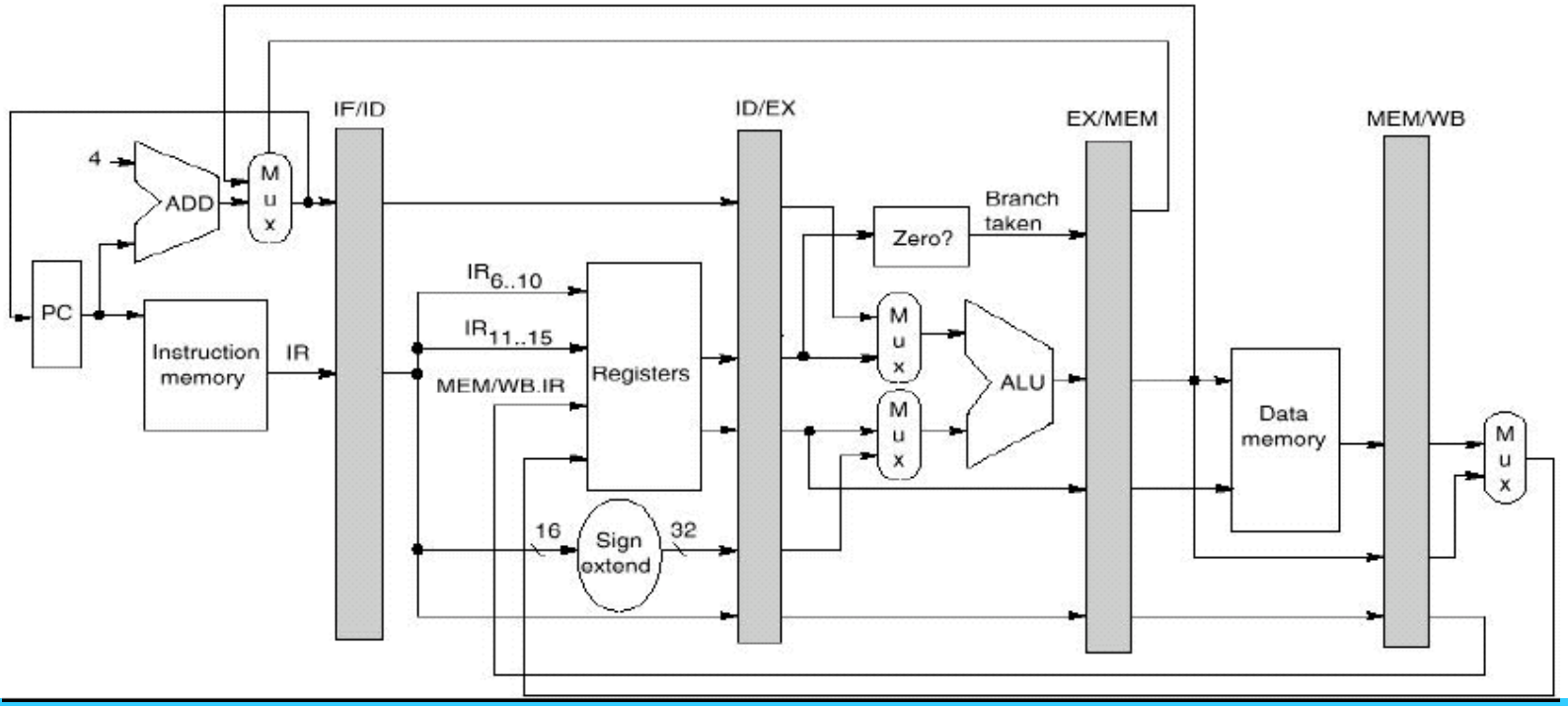


DLX Pipeline Stages



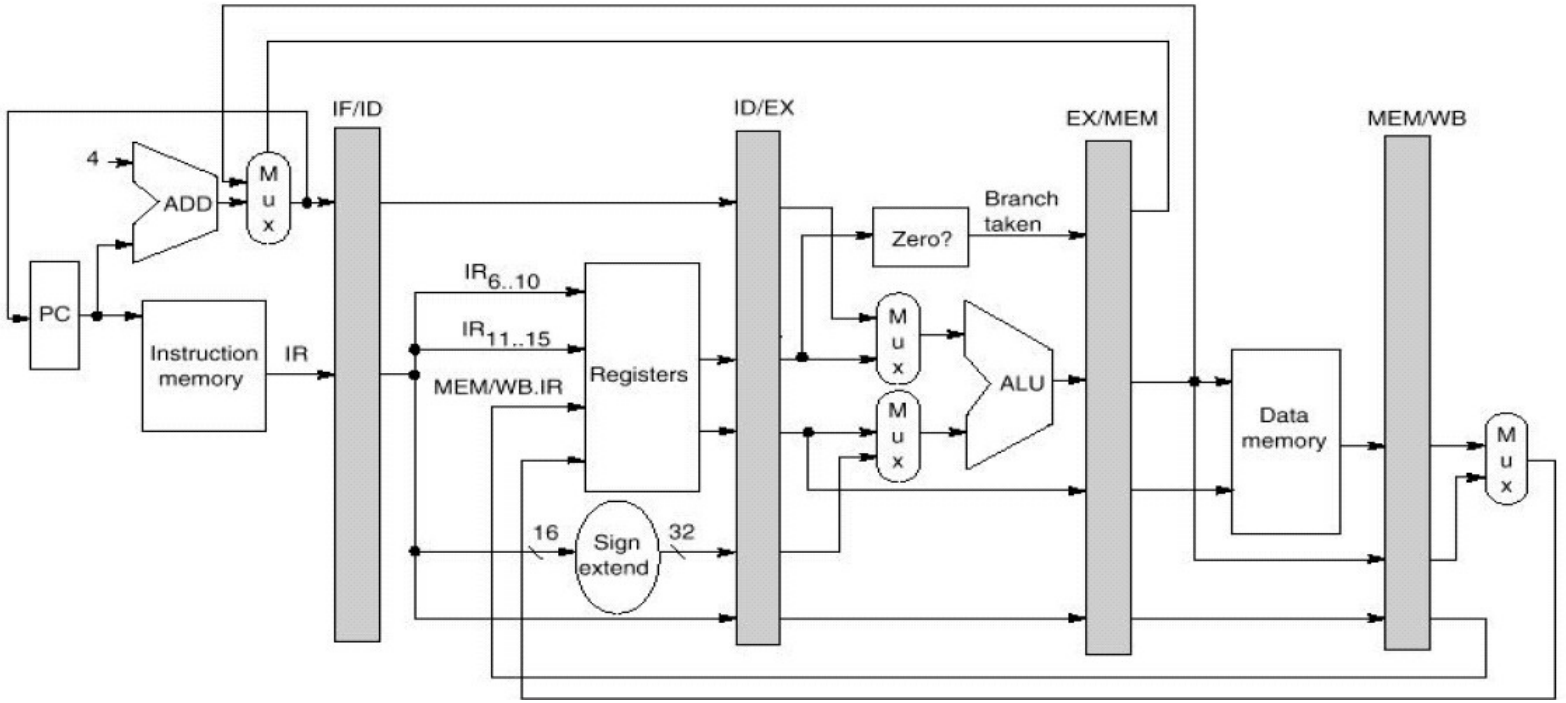
Events in DLX Pipe

- IF Stage
  - IF/ID.IR ← Mem[PC]
  - IF/ID.NPC, PC ← if (EX/MEM.Opcode == Branch && EX/MEM.cond) ← EX/MEM.ALU output else ← PC+4
- ID Stage
  - ID/EX.A ← Regs[IF/ID.IR(6:10)]; ID/EX.B ← Regs[IF/ID.IR(11:15)]; ID/EX.NPC ← IF/ID.NPC;
  - ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR(16))^{16}#IF/ID.IR(16:31)



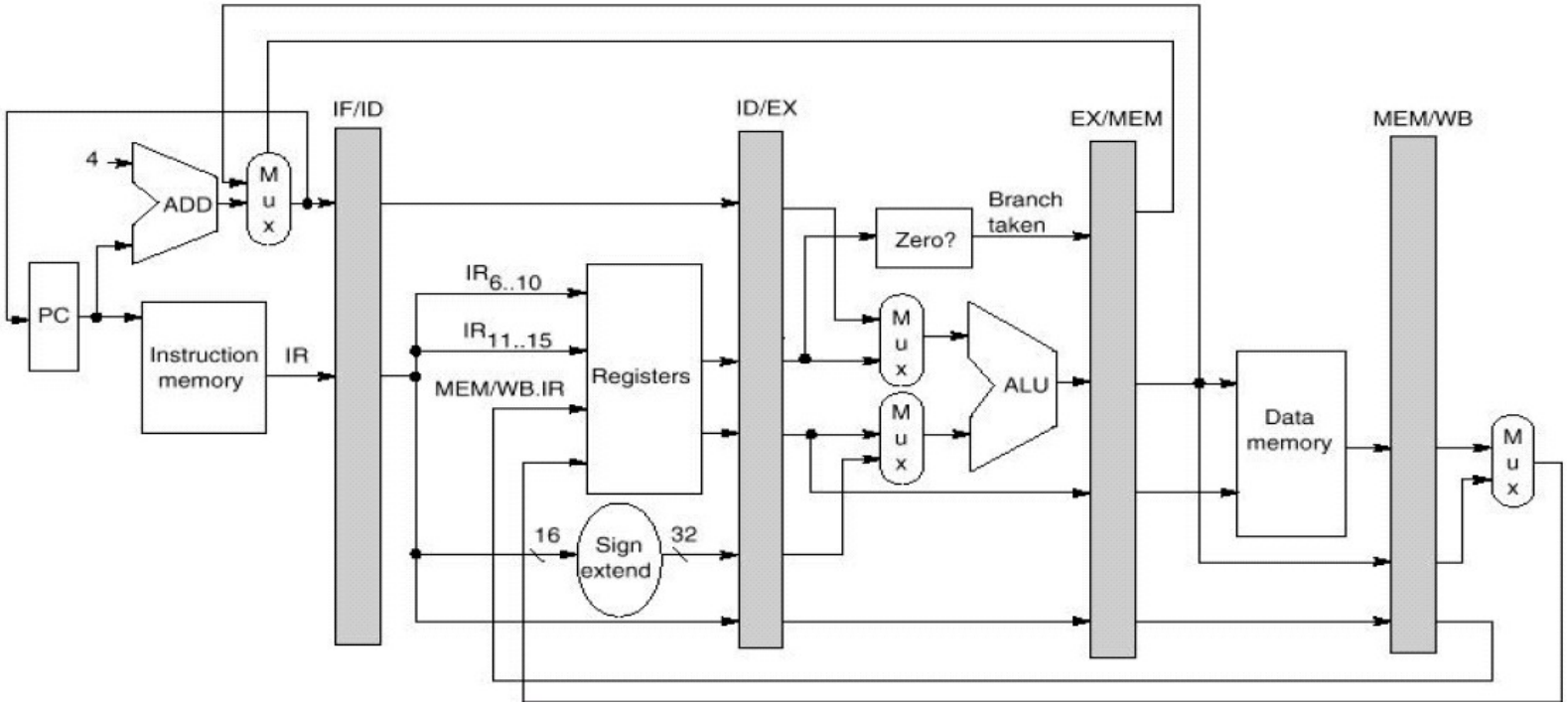
Events in DLX Pipe

	ALU instruction	Load or store instruction	Branch instruction
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput← ID/EX.A *op* ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm; EX/MEM.cond ← 0;	EX/MEM.IR← ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.cond ← 0; EX/MEM.B← ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC+ID.EX.Imm; EX/MEM.cond ← (ID/EX.A *op* 0);



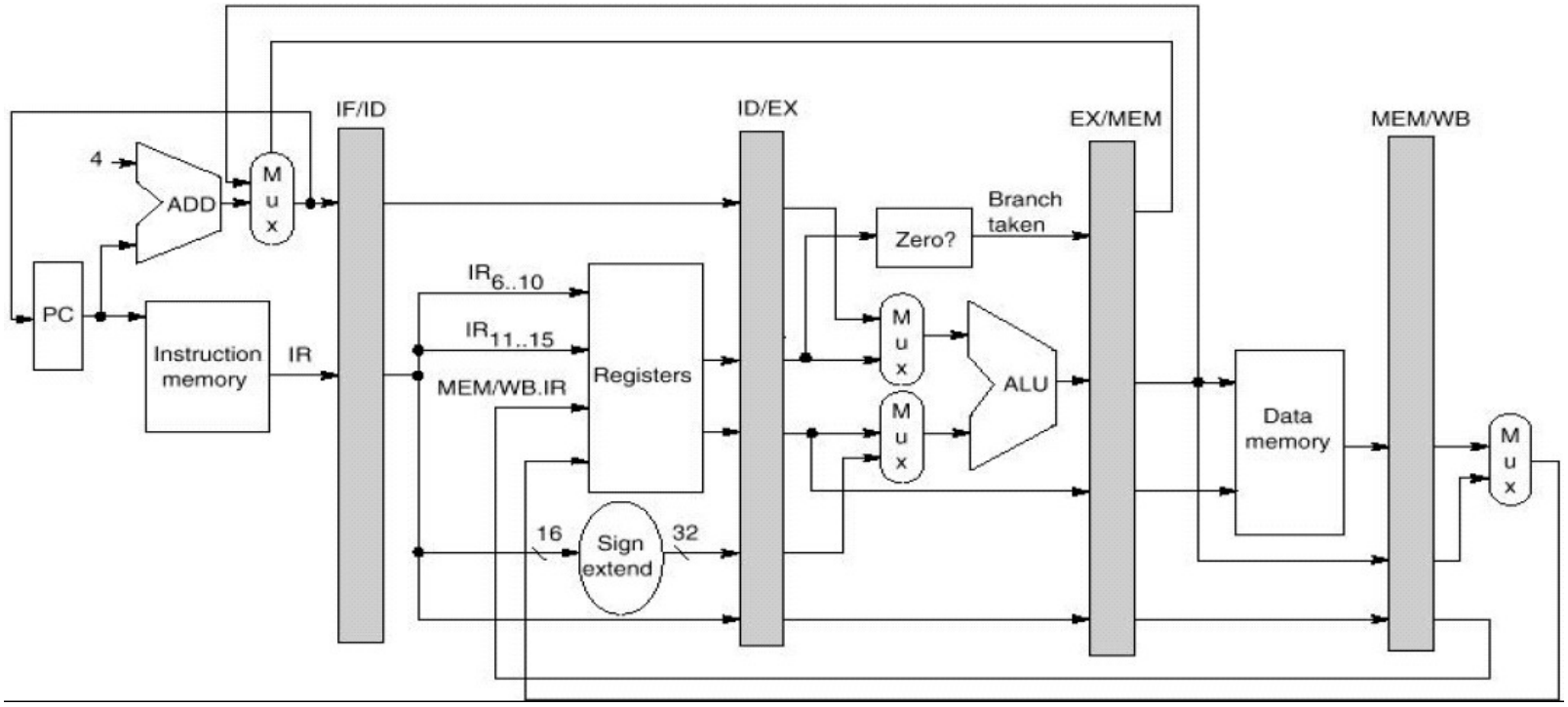
Events in DLX Pipe

MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;



Events in DLX Pipe

WB	Regs[MEM/WB.IR$_{16..20}$] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.ALUOutput;	Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.LMD;



Pipeline Hazards: Major Hurdles

- Structural Hazards
  - ➔Resource conflicts
- Data Hazards
  - ➔Data dependencies
- Control Hazards
  - ➔Branches and other instructions that change PC

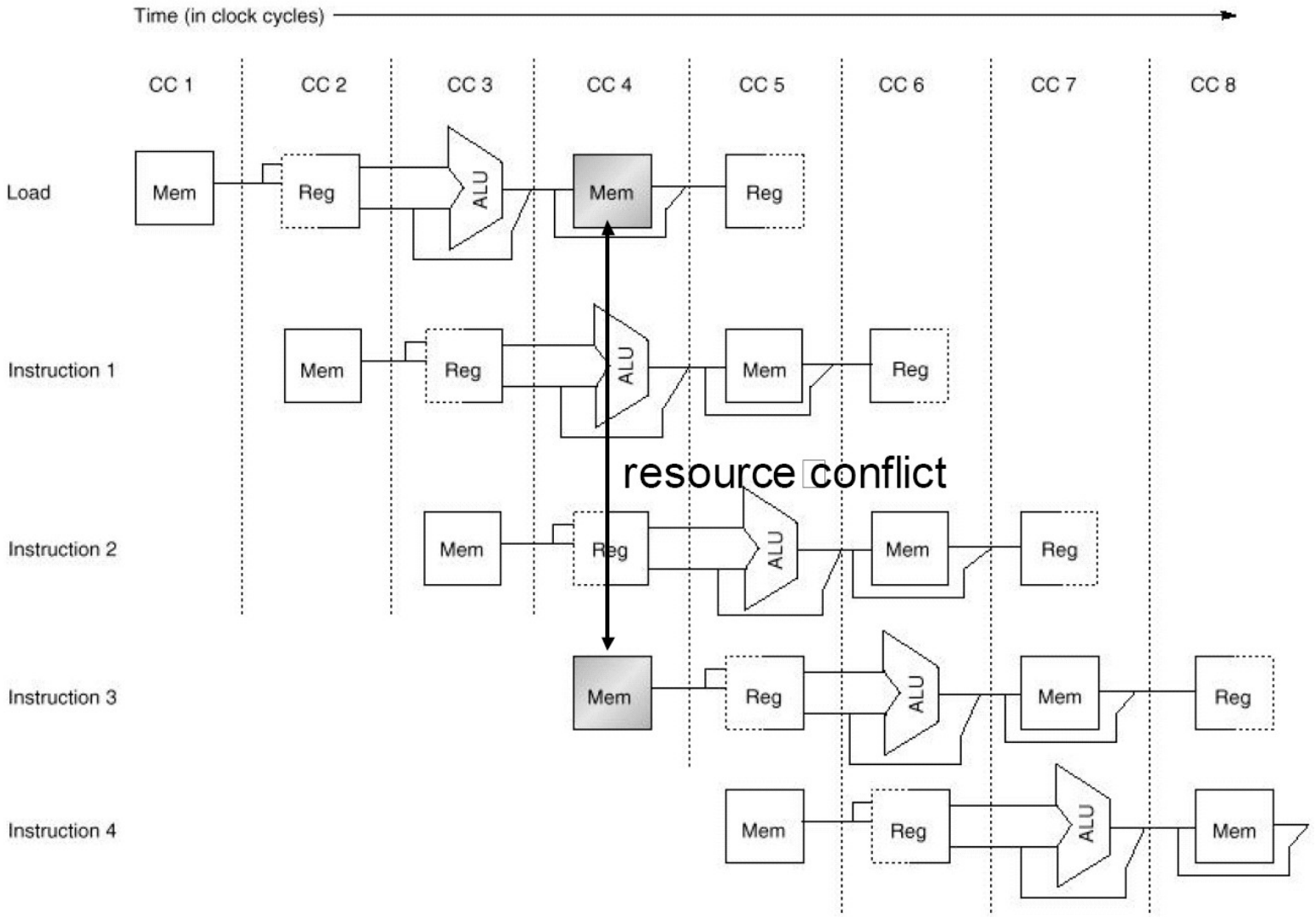
Pipeline Hazards: Major Hurdles

- Structural Hazards
  - ➔Resource conflicts
- Data Hazards
  - ➔Data dependencies
- Control Hazards
  - ➔Branches and other instructions that change PC

Structural Hazard

- Some combination of instructions result in resource conflicts
  - ➔Some functional units are not fully pipelined
  - ➔Some resource is not duplicated enough
    - ✓Register file write port
    - ✓Single memory pipeline for instruction and data
- Stall pipeline for one cycle
  - ➔Also called a *Bubble*

Structural Hazard

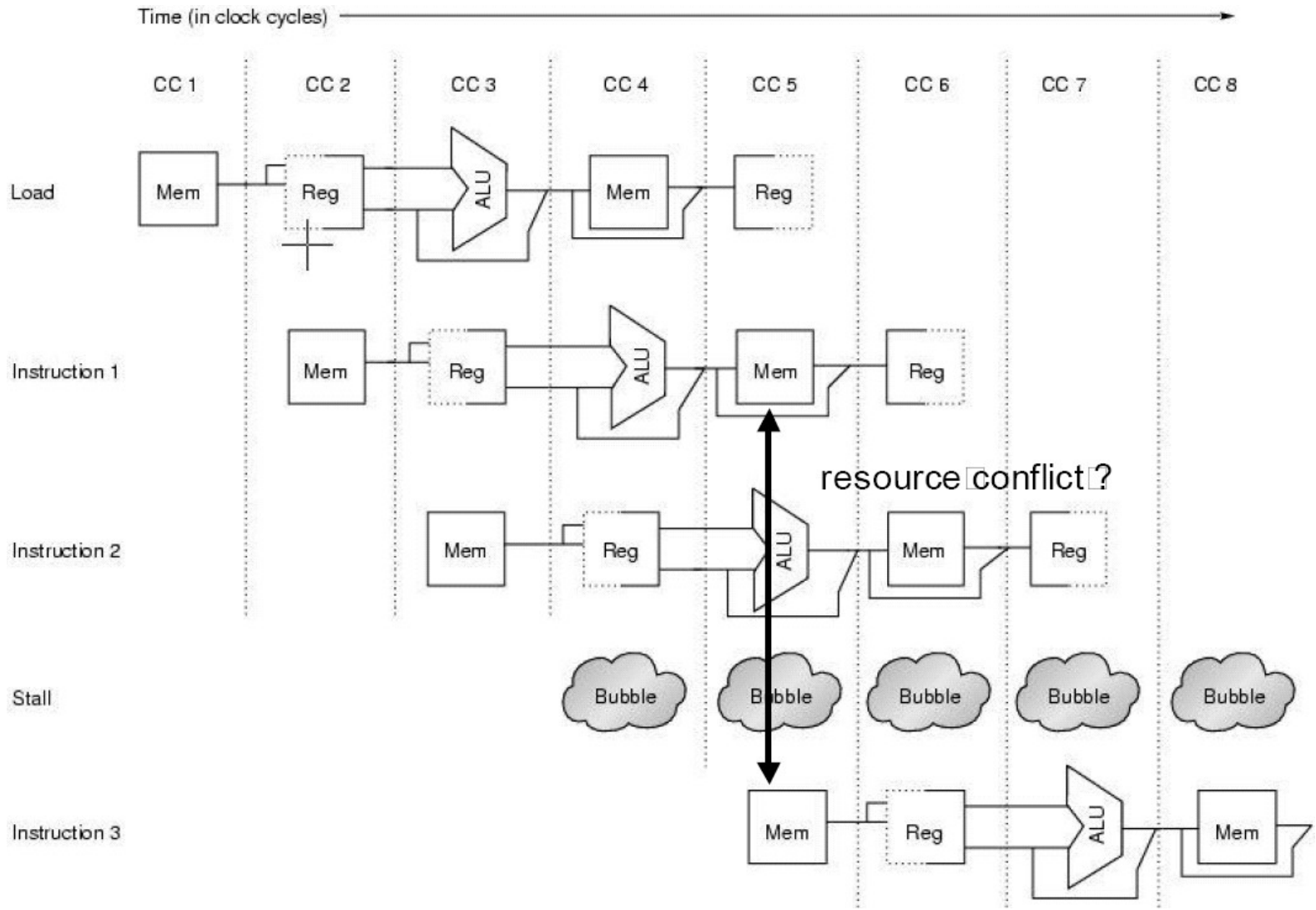


Pipeline Stall

- Single memory for data and instructions

	Clock cycle								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Load instruction	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				**Stall**	IF	ID	EX	MEM	WB
Instruction i+4						IF	ID	EX	MEM

Structural Hazard and Bubbles



Solutions to Structural Hazard

- Resource Duplication
  - ➔ Example
    - ✓ Separate I and D caches for memory access conflict
    - ✓ Time-multiplexed or multiport register file for register file access conflict

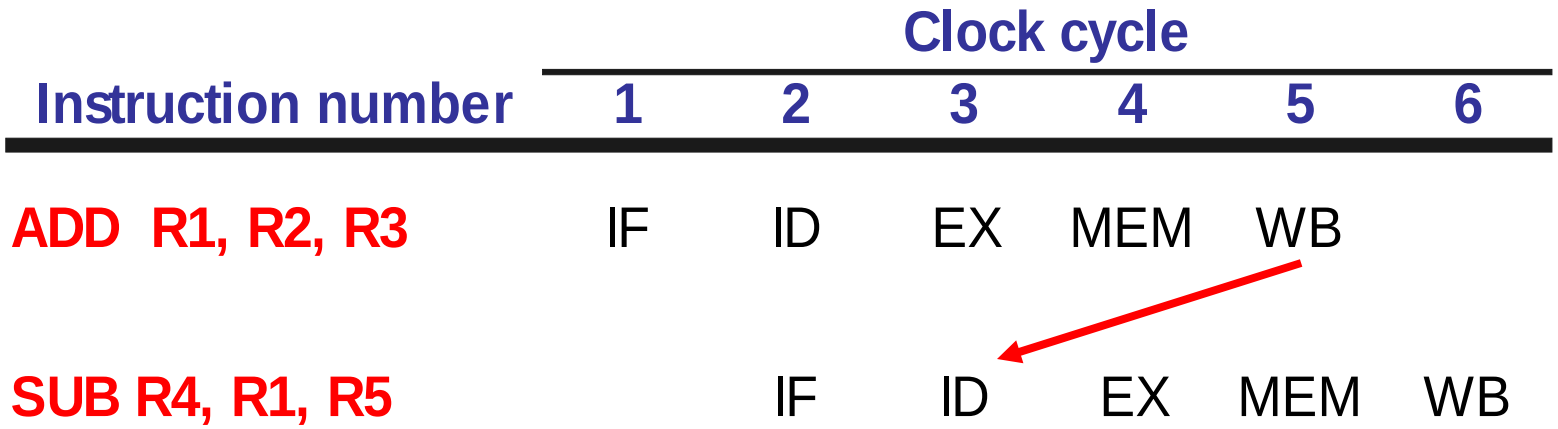
Pipeline Hazards: Major Hurdles

- Structural Hazards
  - ➔Resource conflicts
- Data Hazards
  - ➔Data dependencies
- Control Hazards
  - ➔Branches and other instructions that change PC

Data Hazards

- Order of access to operands is changed by the pipeline vs the normal order

ADD R1, R2, R3
SUB R4, R1, R5



Types of Data Hazard

- **RAW** (read after write)
  - ➔ *j* tries to read a source before *i* writes it, so *j* may get wrong value
- **WAR** (write after read)
  - ➔ *j* tries to write a destination before it is read by *i*
  - ➔ As reads are early this can not happen in our examples
  - ➔ Autoincrement addressing can create it
- **WAW** (write after write)
  - ➔ *j* tries to write an operand before it is written by *i*. Wrong value may remain in the register
  - ➔ Occurs in pipes which write in more than one stage

Data Hazard

- WAR

SW 0(R1),R2	IF	ID	EX	MEM1	**MEM2**	WB
ADD R2,R3,R4		IF	ID	EX	**WB**	

- WAW

LW R1,0(R2)	IF	ID	EX	MEM1	MEM2	**WB**
ADD R1,R2,R3		IF	ID	EX	**WB**	

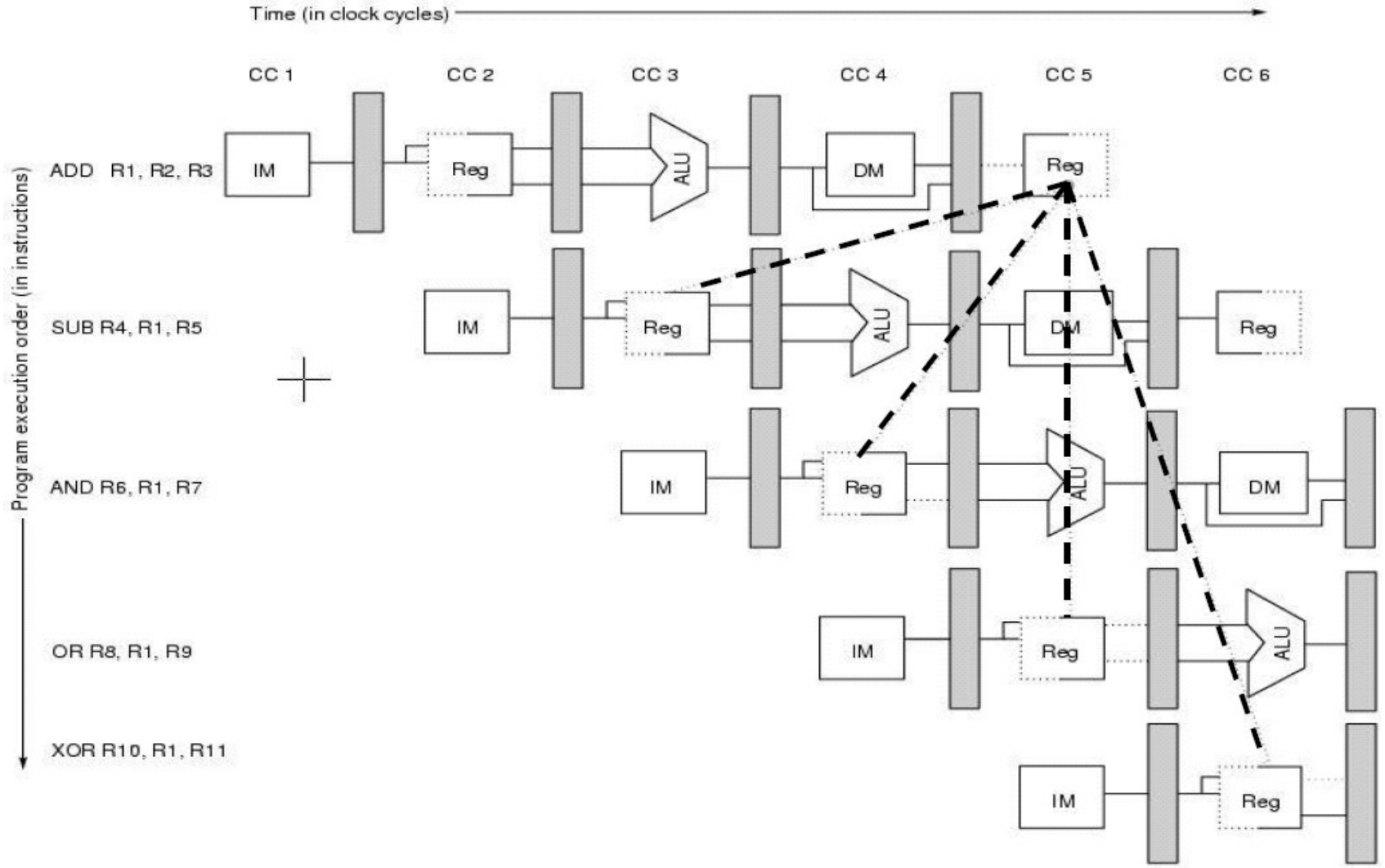
Solutions to Data Hazard

- (Internal) Forwarding
  - ➔Extra hardware
- Freezing the pipeline
  - ➔Stalls/bubbles; pipeline interlock
- Compiler (instruction) scheduling
  - ➔Delay slot

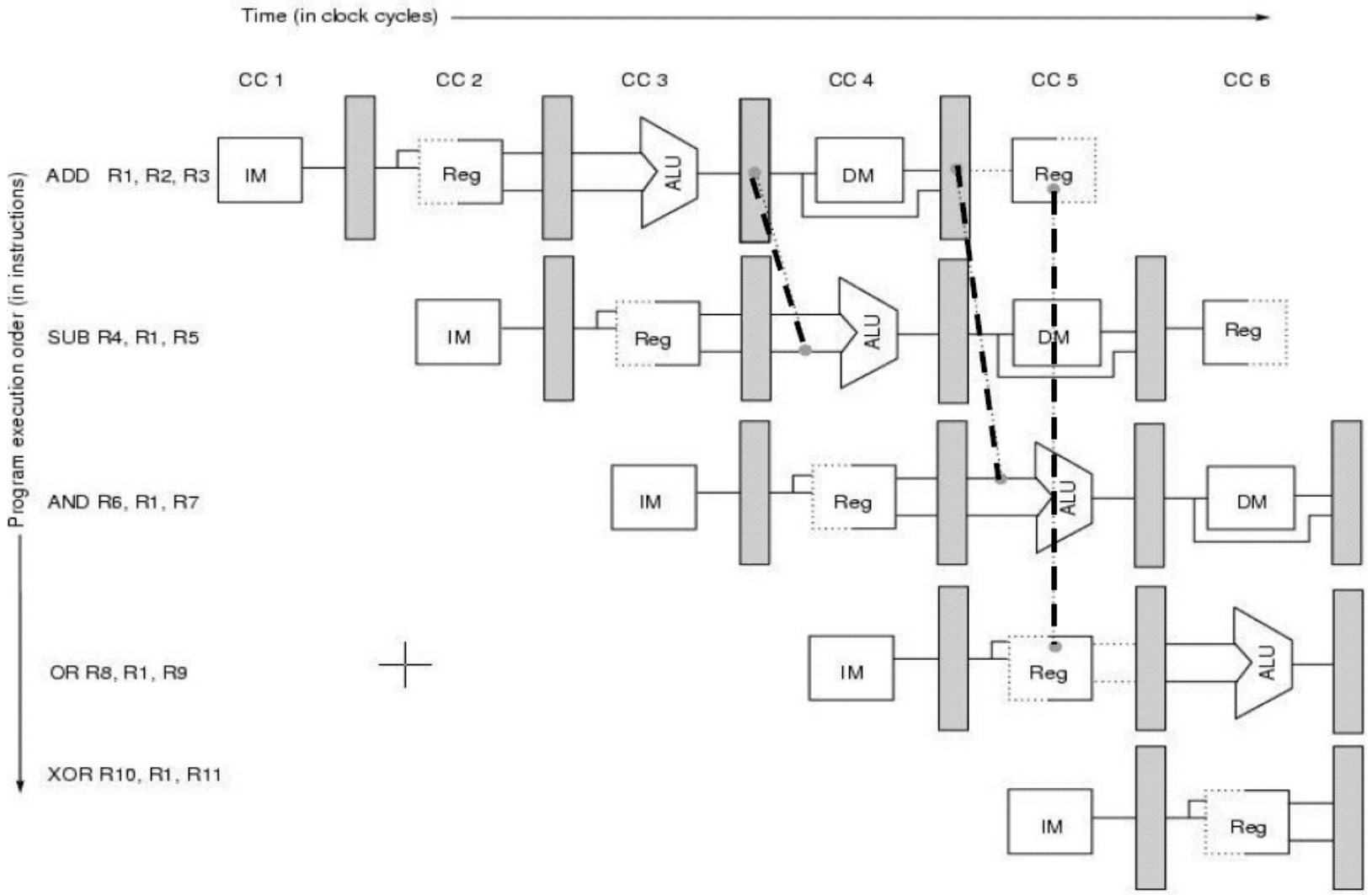
Data Hazard

```
ADD     R1, R2, R3
SUB     R4, R5, R1
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10, R1, R11
```

Data Hazard

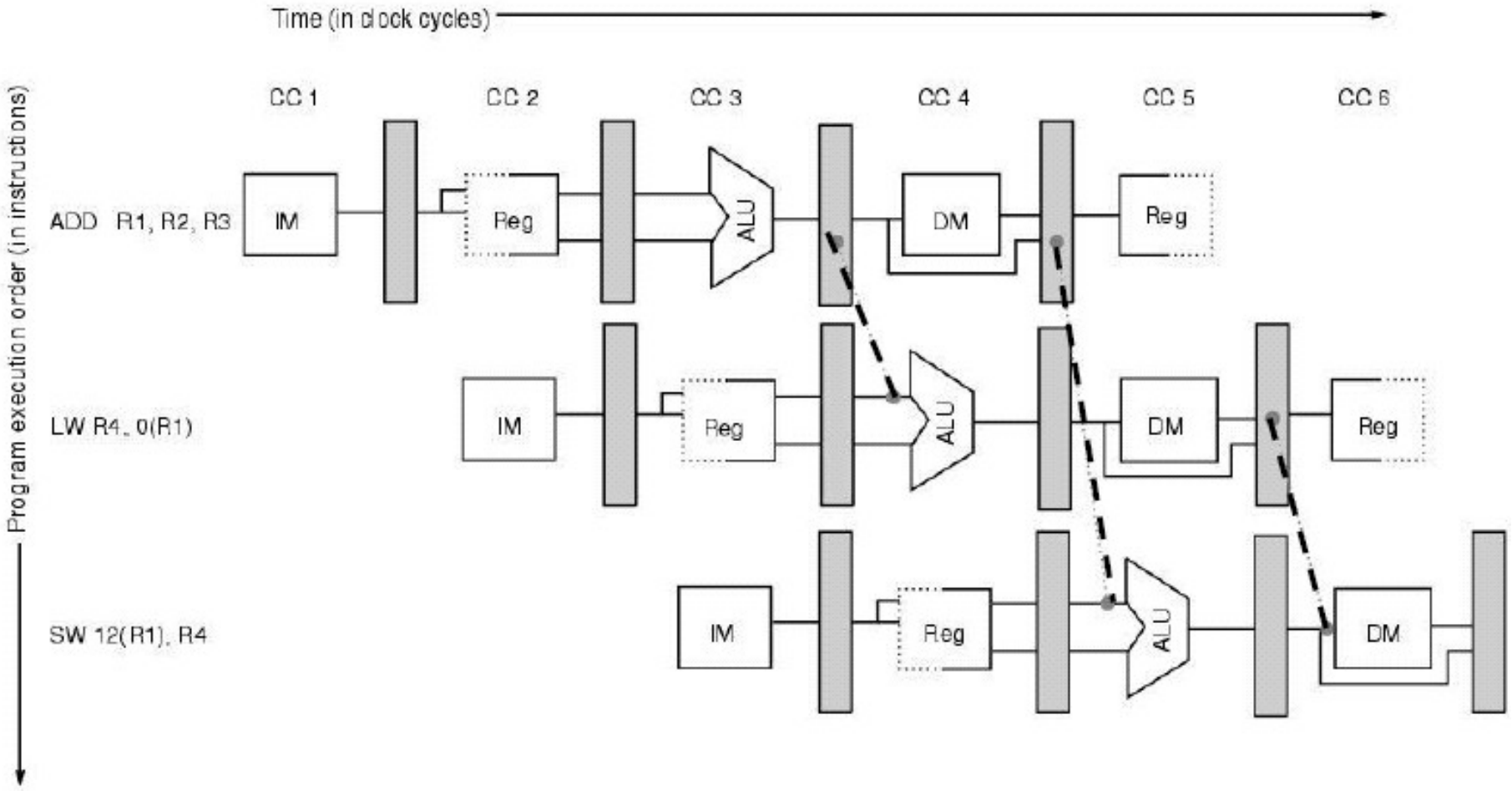

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
Program execution order (in instructions)
ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM

Forwarding Paths

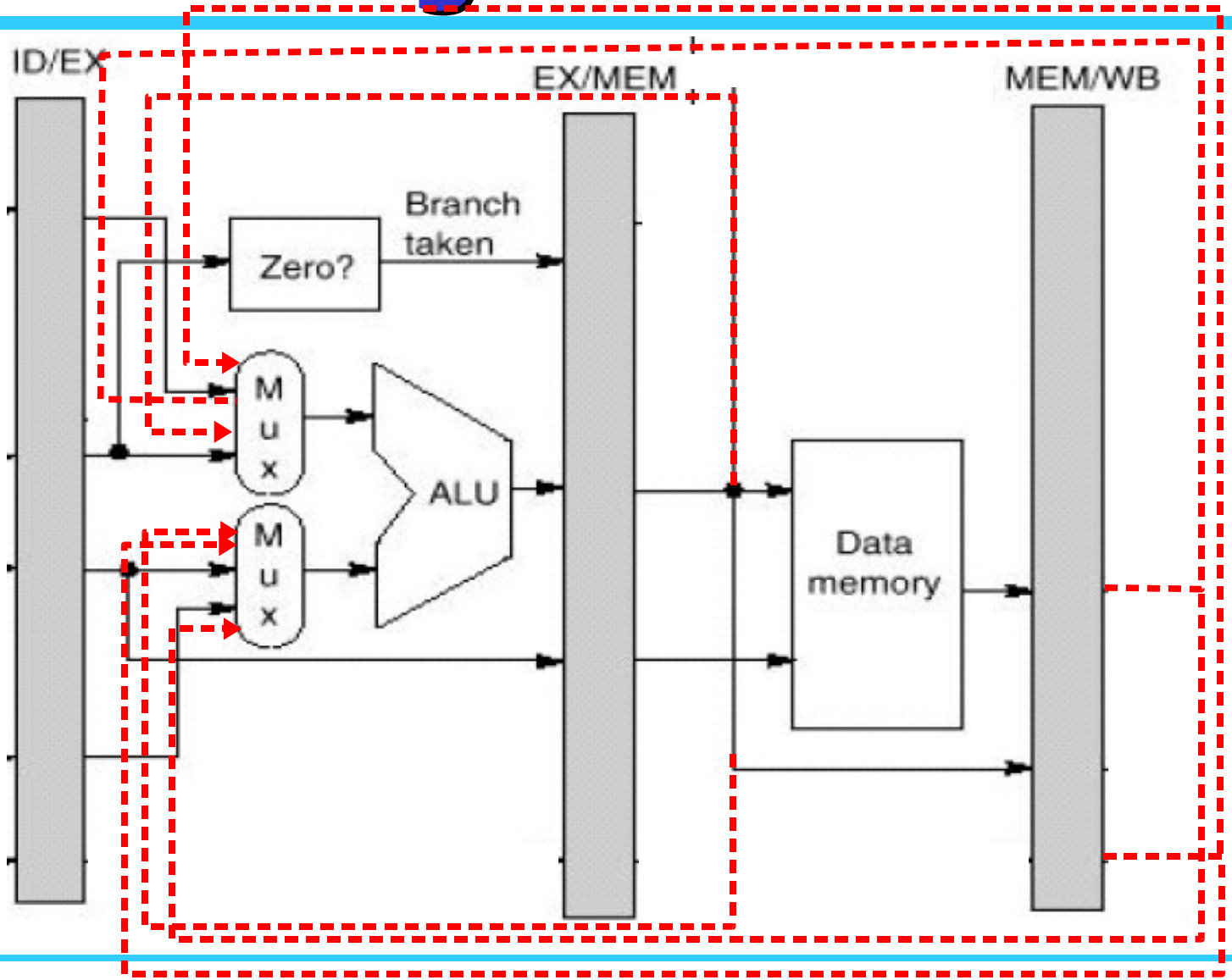


Another Example

```
ADD     R1, R2, R3
LW      R4, 0(R1)
SW      12(R1), R4
```



Forwarding



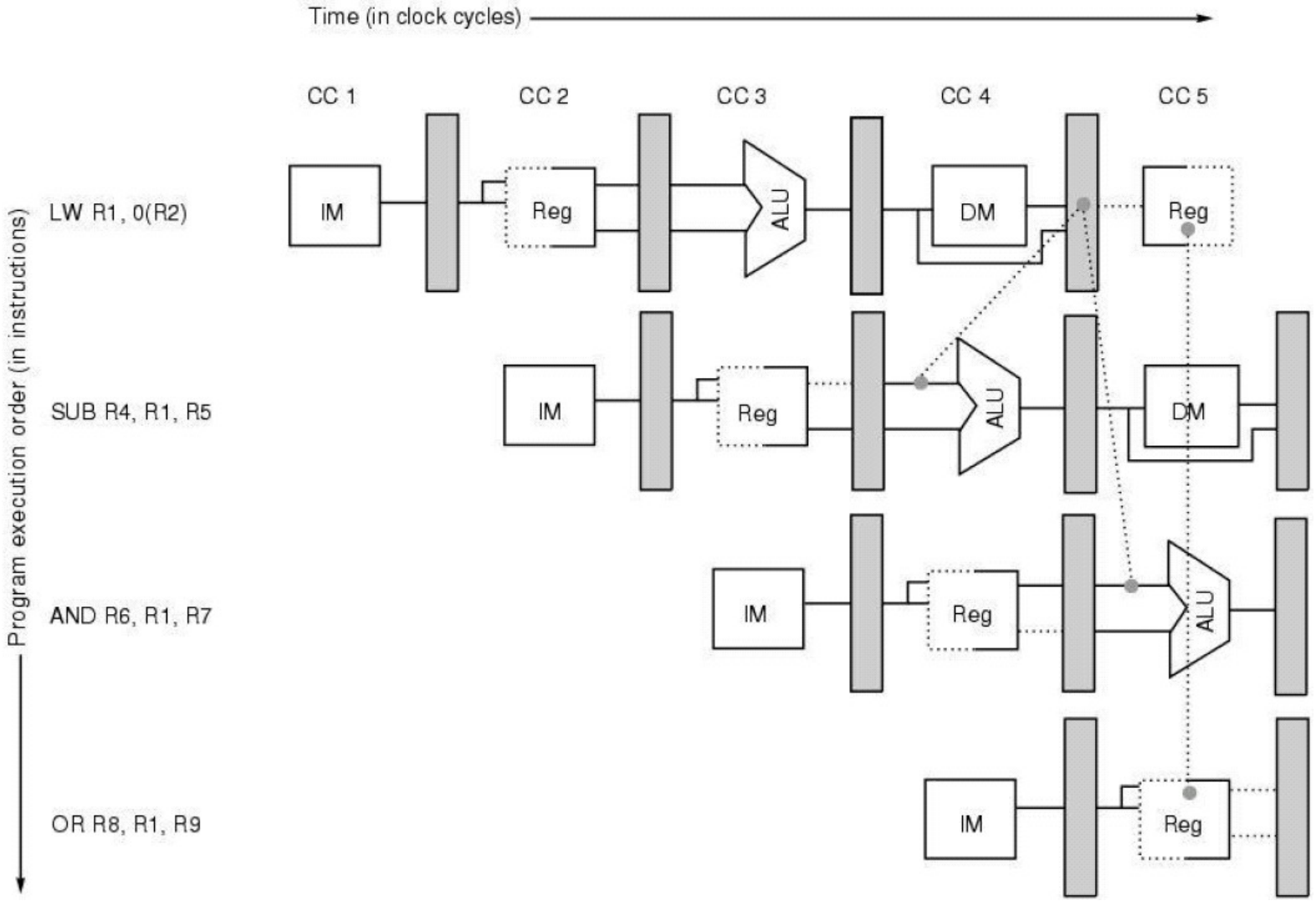
Data Hazards Requiring Stalls

- Situations were forwarding is not possible
  - ➔Load interlock

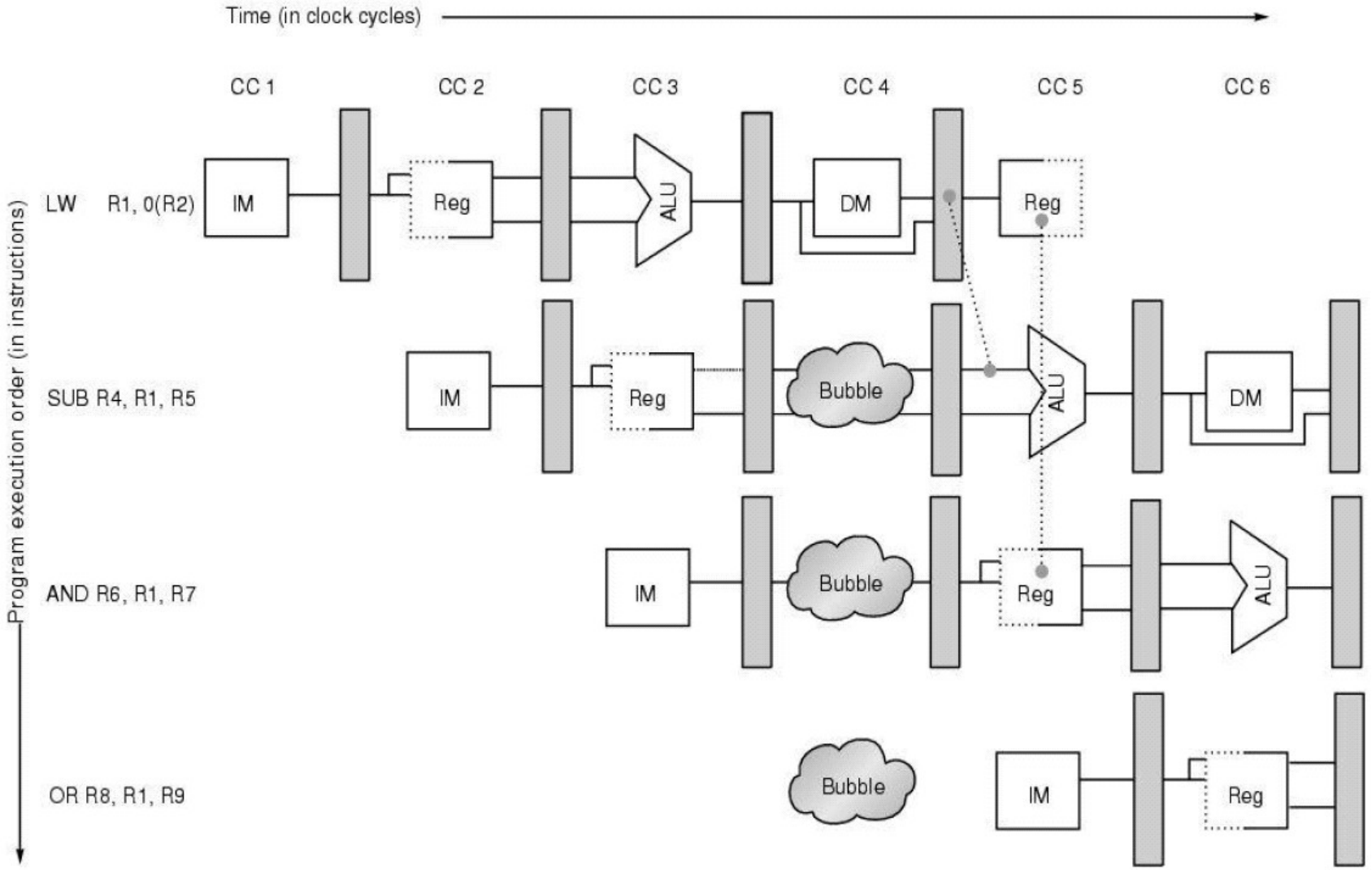
```
LW      R1, 0(R2)
SUB     R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
```

- Pipeline interlock hardware
  - ➔Detects Hazard
  - ➔Stalls the pipe until hazard is cleared

Load cannot Bypass Results to SUB



Pipeline Interlock Solution



Load Interlocks

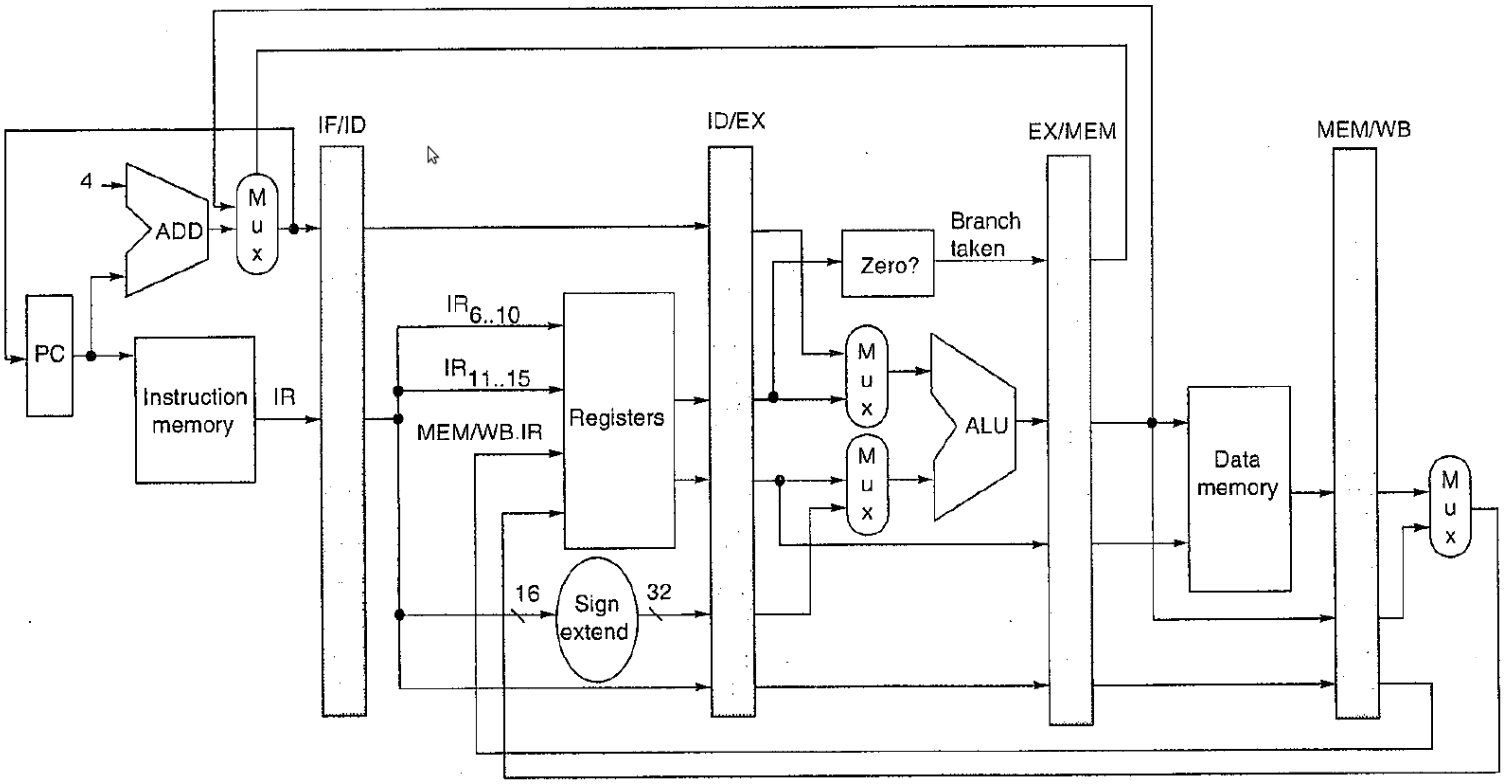
Opcode field of ID/EX ID/EX.IR(0:5)	Opcode field of IF/ID IF/ID.IR(0:5)	Matching operand fields
Load	Reg-reg ALU	ID/EX.IR(11:15)==IF/ID.IR(6:10)
Load	Reg-reg ALU	ID/EX.IR(11:15)==IF/ID.IR(11:15)
Load	Load, Store, ALU imm, Branch	ID/EX.IR(11:15)==IF/ID.IR(6:10)

Pipeline Hazards: Major Hurdles

- Structural Hazards
  - ➔Resource conflicts
- Data Hazards
  - ➔Data dependencies
- Control Hazards
  - ➔Branches and other instructions that change PC

Control Hazards

- Can cause a greater performance loss than do data hazards
- Recall that if instruction *i* is a *taken branch*
  - ➔ PC is not changed until the end of MEM



Stalling the Pipeline

- The simplest method to deal with branches is stall the pipeline
- We do not want to stall the pipeline until we know that the instruction is a branch
  - ➔ The stall does not occur until after the ID stage

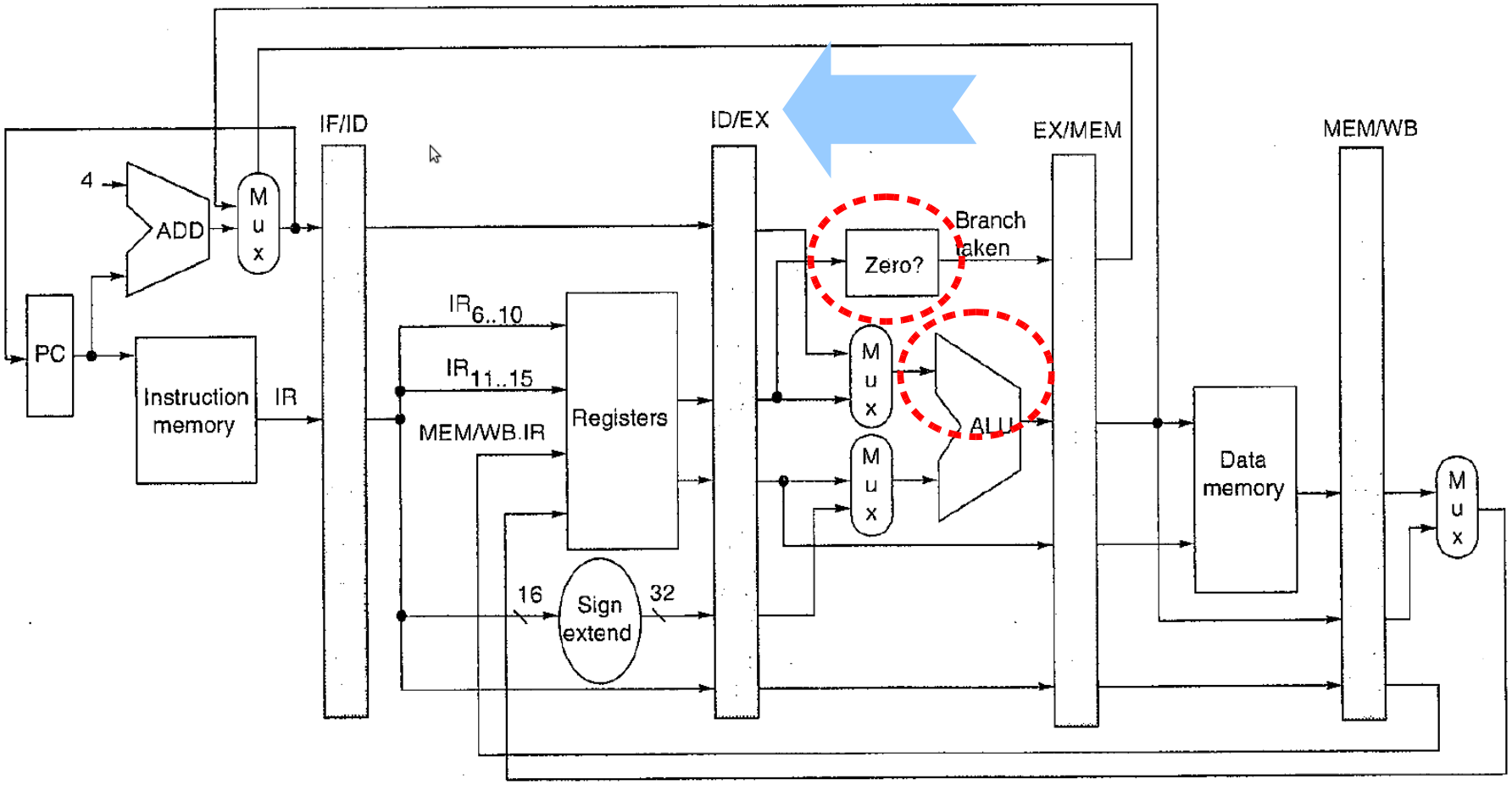
Branch instruction	IF	ID	EX	MEM	WB					
Branch successor		IF	*stall*	*stall*	IF	ID	EX	MEM	WB	
Branch successor + 1						IF	ID	EX	MEM	WB
Branch successor + 2							IF	ID	EX	MEM
Branch successor + 3								IF	ID	EX
Branch successor + 4									IF	ID
Branch successor + 5										IF

3 cycles penalty

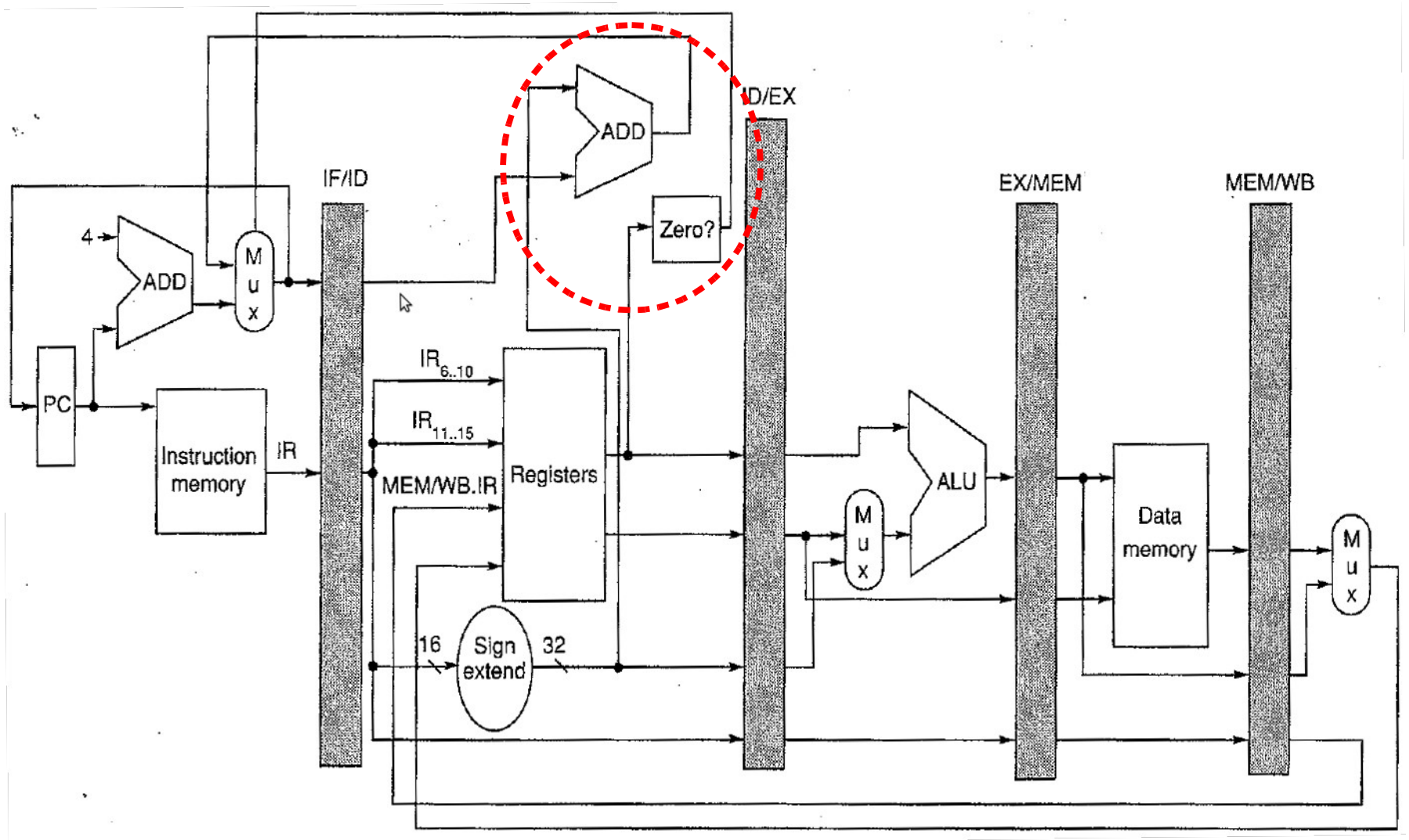
Reducing the Branch Penalty

- The number of clock cycles in a branch stall can be reduced in two steps
  - ➔Find out whether the branch is taken or not taken earlier in the pipeline
  - ➔Compute the taken PC (i.e., the address of the branch target) earlier

Reducing the Branch Penalty



Reducing the Branch Penalty



Reducing the Branch Penalty

- With a separate adder and a branch decision made during ID, there is only 1 clock cycle stall on branches

Branch instruction	IF	ID	EX	MEM	WB			
Branch successor		IF	IF	ID	EX	MEM	WB	
Branch successor + 1		1 cycle penalty		IF	ID	EX	MEM	WB
Branch successor + 2					IF	ID	EX	MEM
Branch successor + 3						IF	ID	EX
Branch successor + 4							IF	ID
Branch successor + 5								IF

Reducing the Branch Penalty

- *Predict-not-taken* scheme
- Treat every branch as not taken
  - ➔Allowing the hw to continue as if the branch were not executed
    - ✓ If the branch is taken, it needs to turn the fetched instruction into a nop

Branch not taken

Untaken branch instr.	IF	ID	EX	MEM	WB				
Instruction I + 1		IF	ID	EX	MEM	WB			
Instruction I + 2			IF	ID	EX	MEM	WB		
Instruction I + 3				IF	ID	EX	MEM	WB	
Instruction I + 4					IF	ID	EX	MEM	WB

Branch taken

Taken branch instr.	IF	ID	EX	MEM	WB				
Instruction I + 1		IF	*idle*	*idle*	*idle*	*idle*			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB