بسم الله الرحمن الرحیم

Advanced Computer Architecture
Chapter 4: Processor and pipline

Teacher

Fatemeh Daraee

f_daraei@semnan.ac.ir

https://fdaraei.profile.semnan.ac.ir

Introduction

CPU performance factors

- Instruction count
  - Determined by ISA and compiler
- CPI and Cycle time
  - Determined by CPU hardware

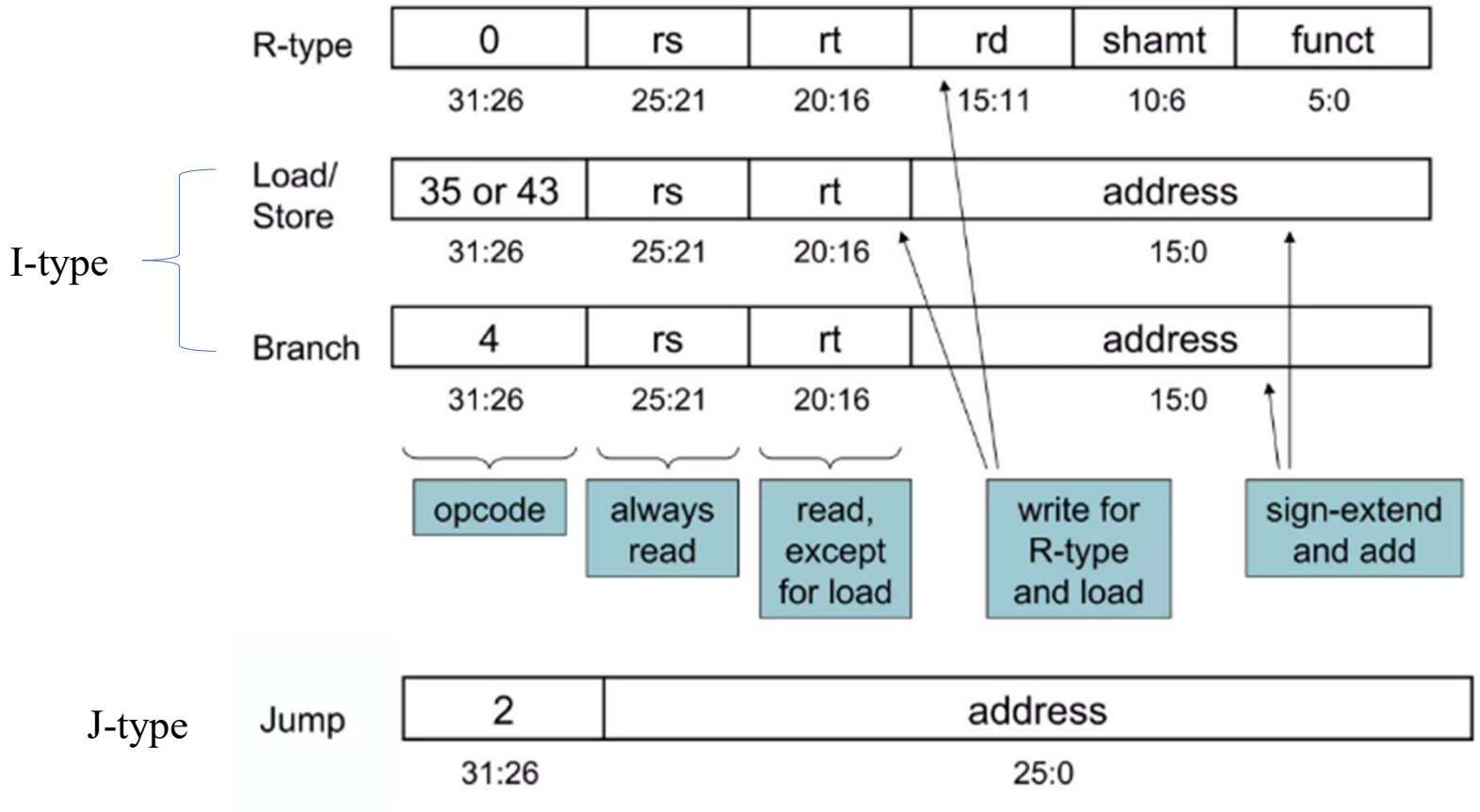
We will examine two MIPS implementations

- A simplified version
- A more realistic pipelined version

Simple subset, shows most aspects

- Memory reference: lw, sw
- Arithmetic/logical: add, sub, and, or, slt
- Control transfer: beq, j

MIPS Instruction Format



Instruction Execution

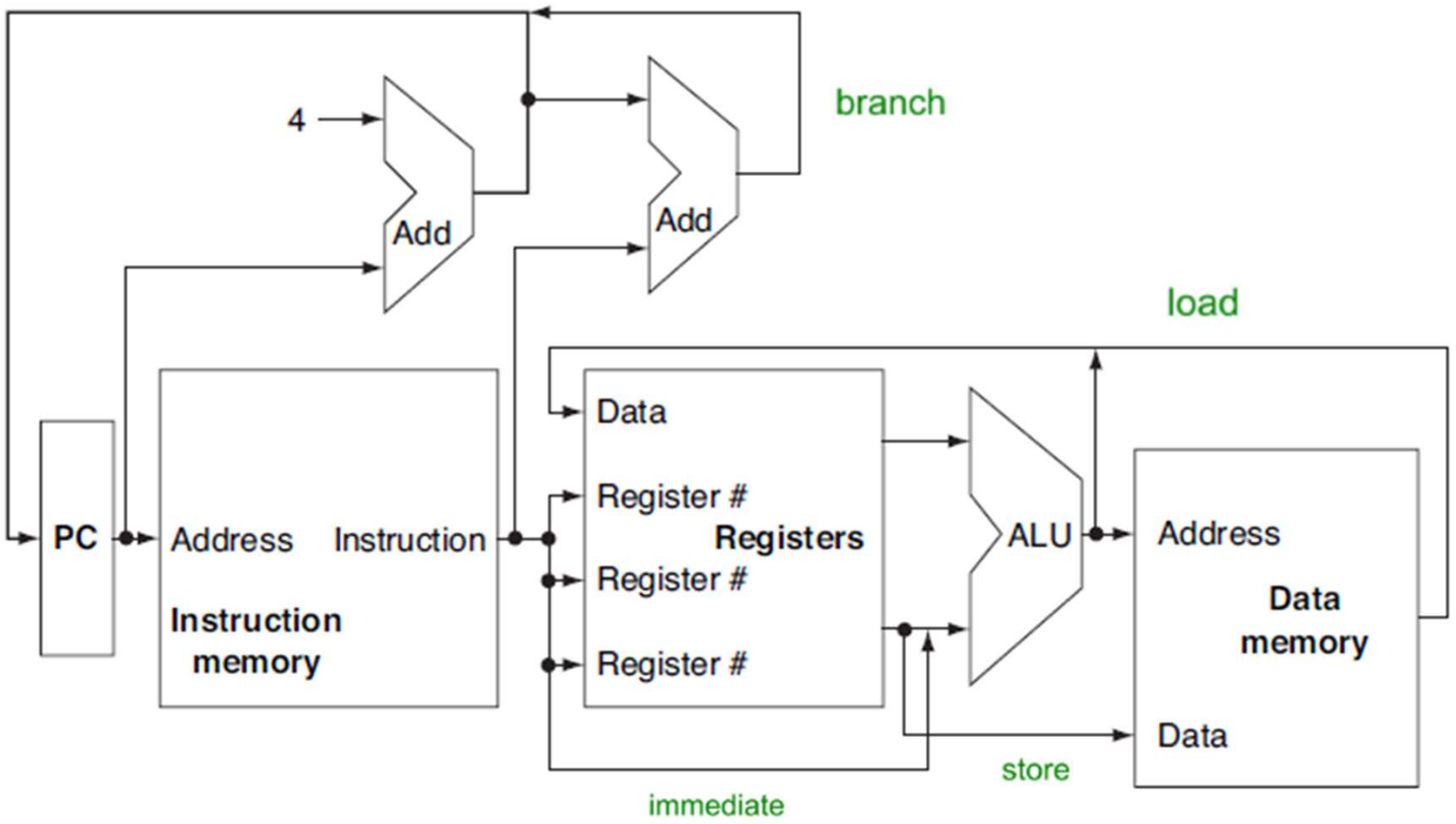
PC → instruction memory, fetch instruction

Register numbers → register file, read registers

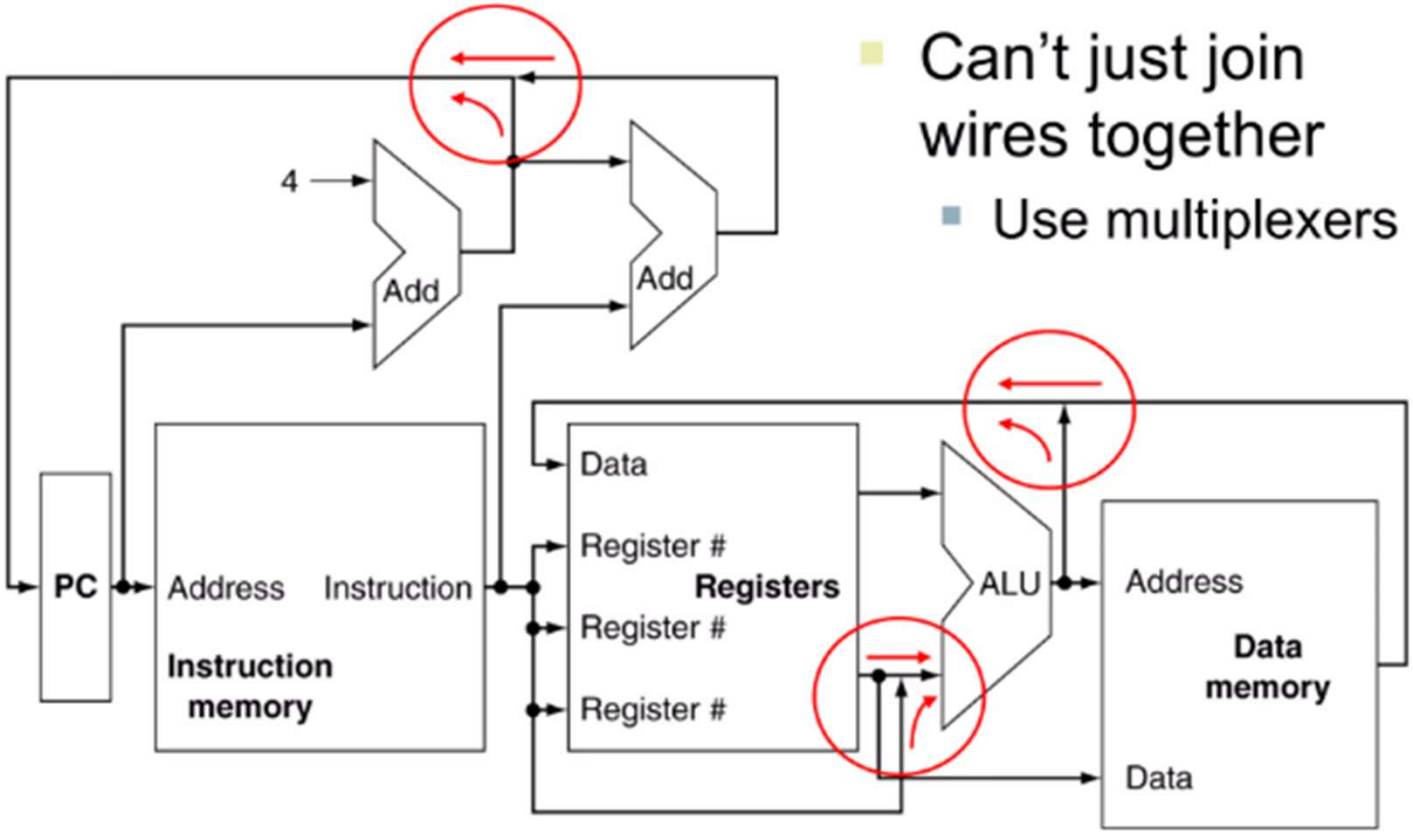
Depending on instruction class:

- Use ALU to calculate:
  - Arithmetic result
  - Memory address for load/store
  - Branch target address
- Access data memory for load/store
- PC ← target address or PC + 4

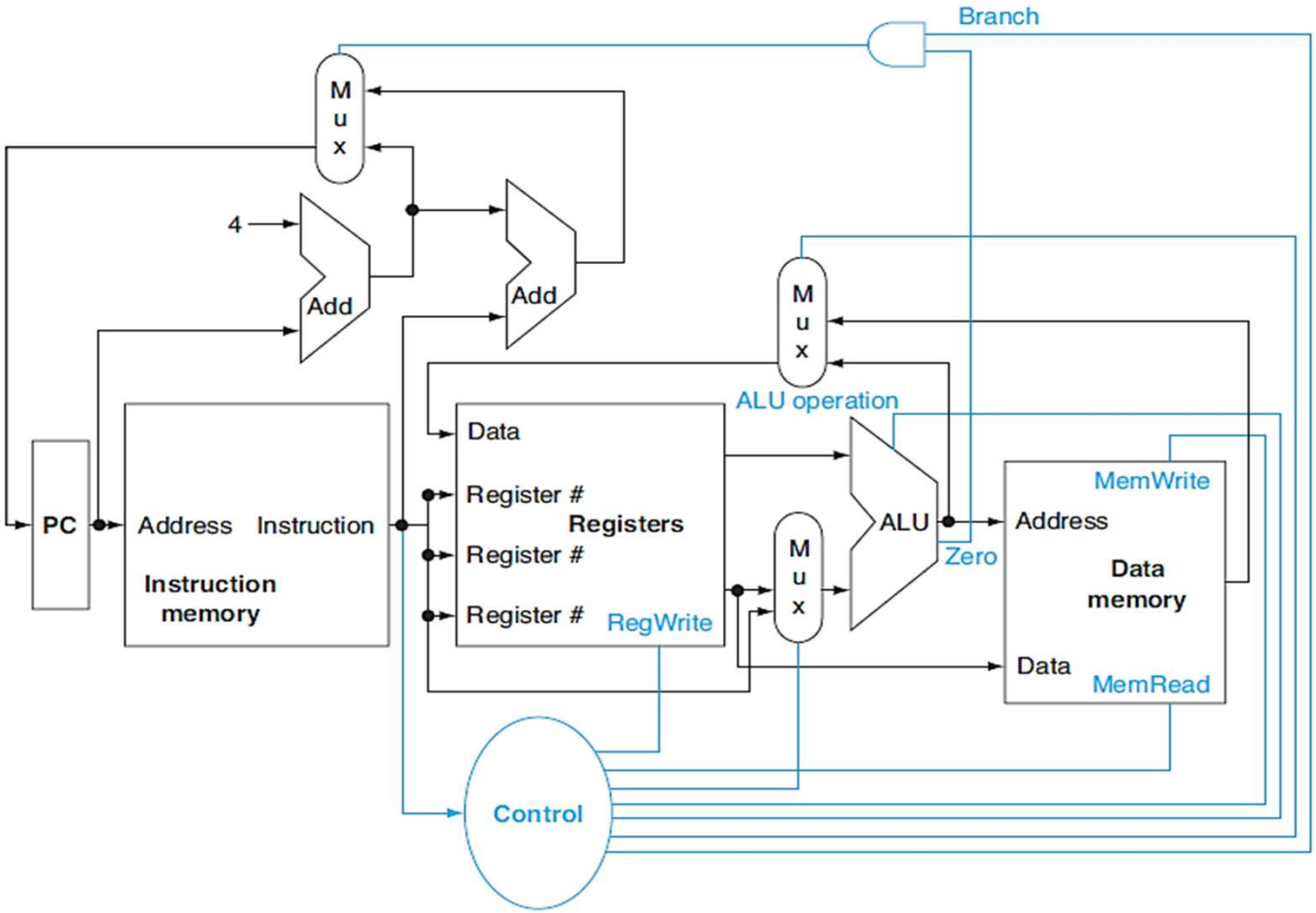
CPU Overview



Multiplexers



Control

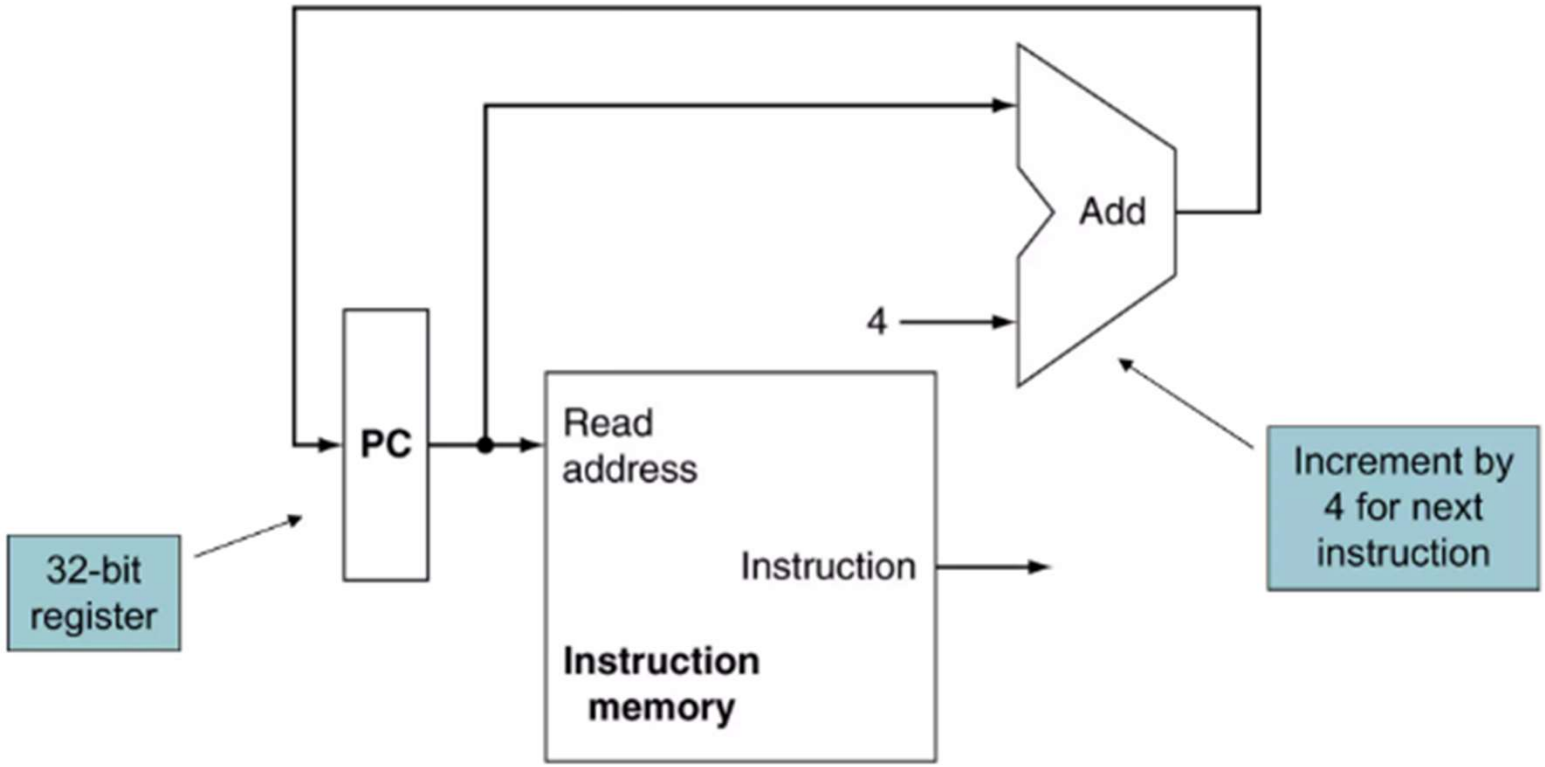

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Building a Datapath

Datapath

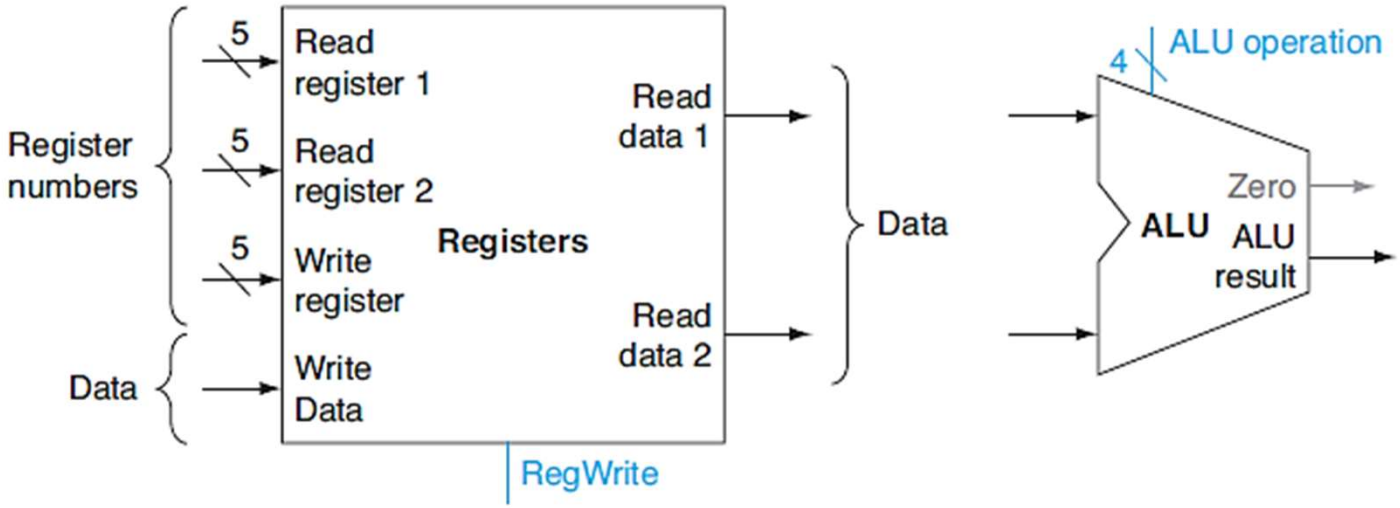
- Elements that process data and addresses in the CPU
  - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

Instruction Fetch



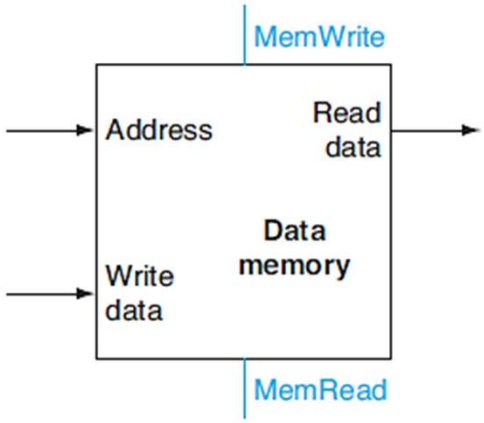
R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

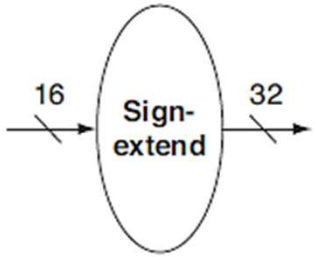


Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- **Load**: Read memory and update register
- **Store**: Write register value to memory



a. Data memory unit



b. Sign extension unit

Branch Instructions

Semnan University

Read register operands

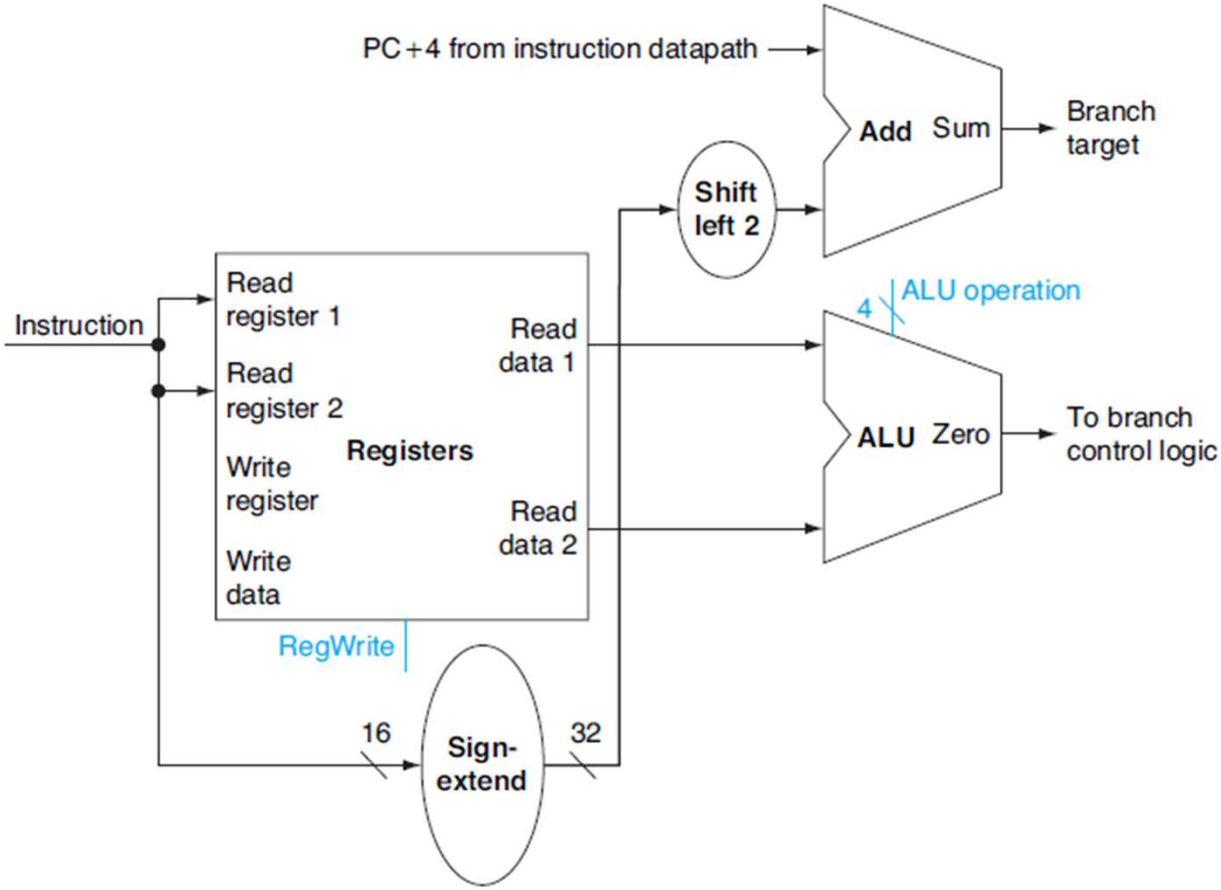
Compare operands

- Use ALU, subtract and check Zero output

Calculate target address

- Sign-extend displacement
- Shift left 2 places (word displacement)
- Add to PC + 4
  - Already calculated by instruction fetch

Branch Instructions



Composing the Elements

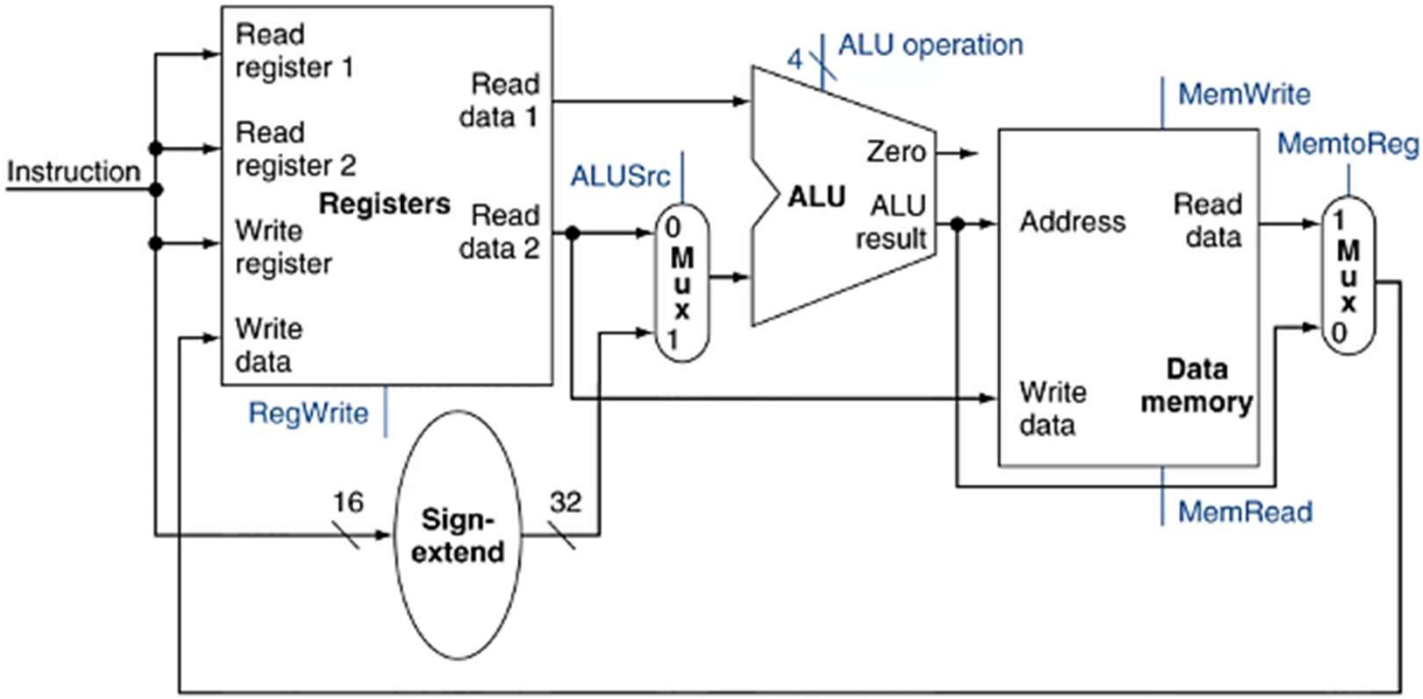
Semnan University

First-cut data path does an instruction in one clock cycle

- Each datapath element can only do one function at a time
- Hence, we need separate instruction and data memories

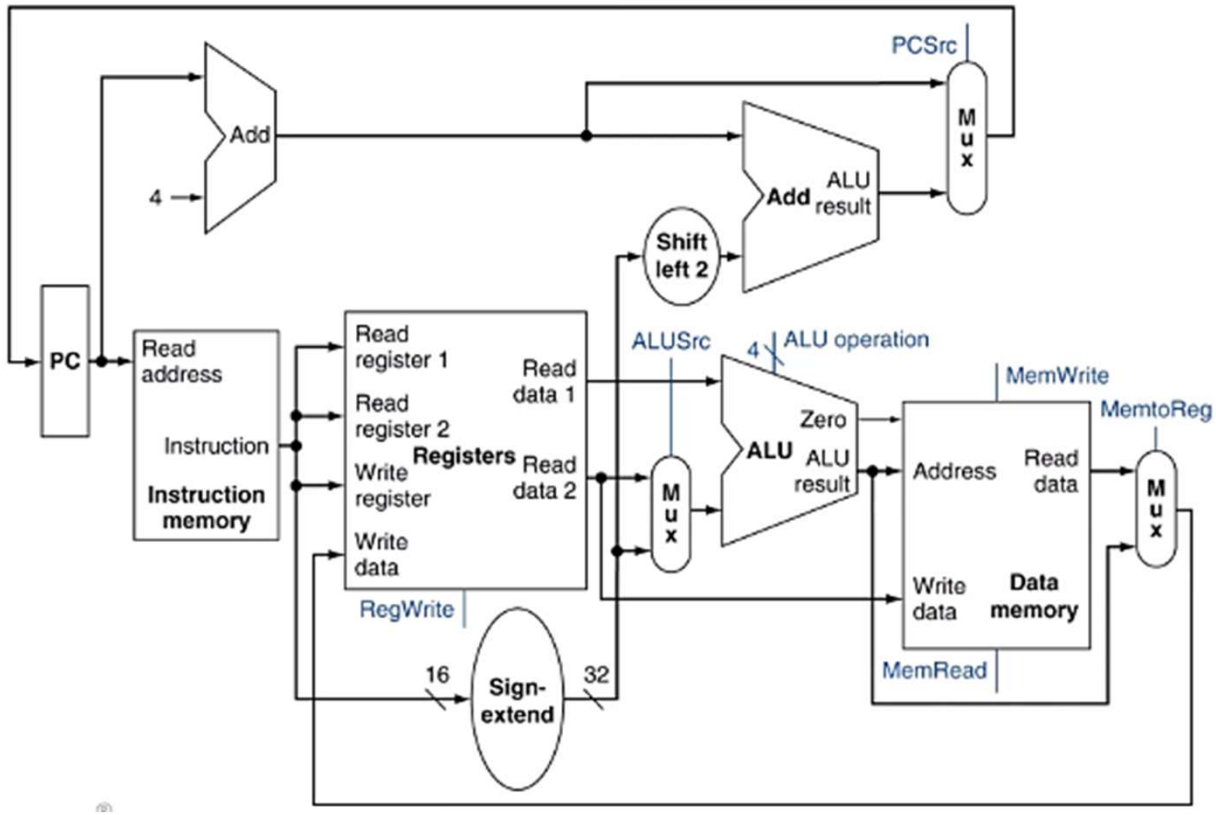
Use multiplexers where alternate data sources are used for different instructions

R-type/Load/Store/Datapath



Semnan University

Full Datapath



ALU Control

Semnan University

ALU used for

- Load/Store: F = add
- Branch: F = subtract
- R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

ALU Control

Assume 2-bit ALUOp derived from opcode

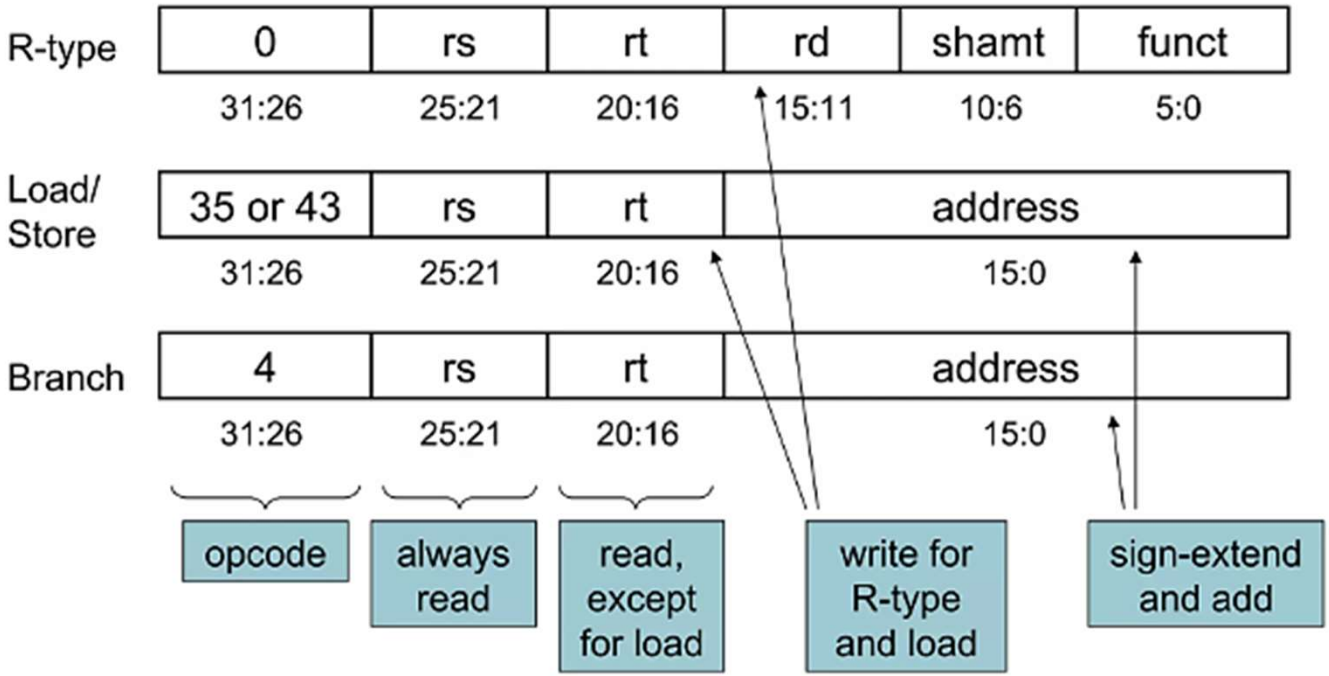
- Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

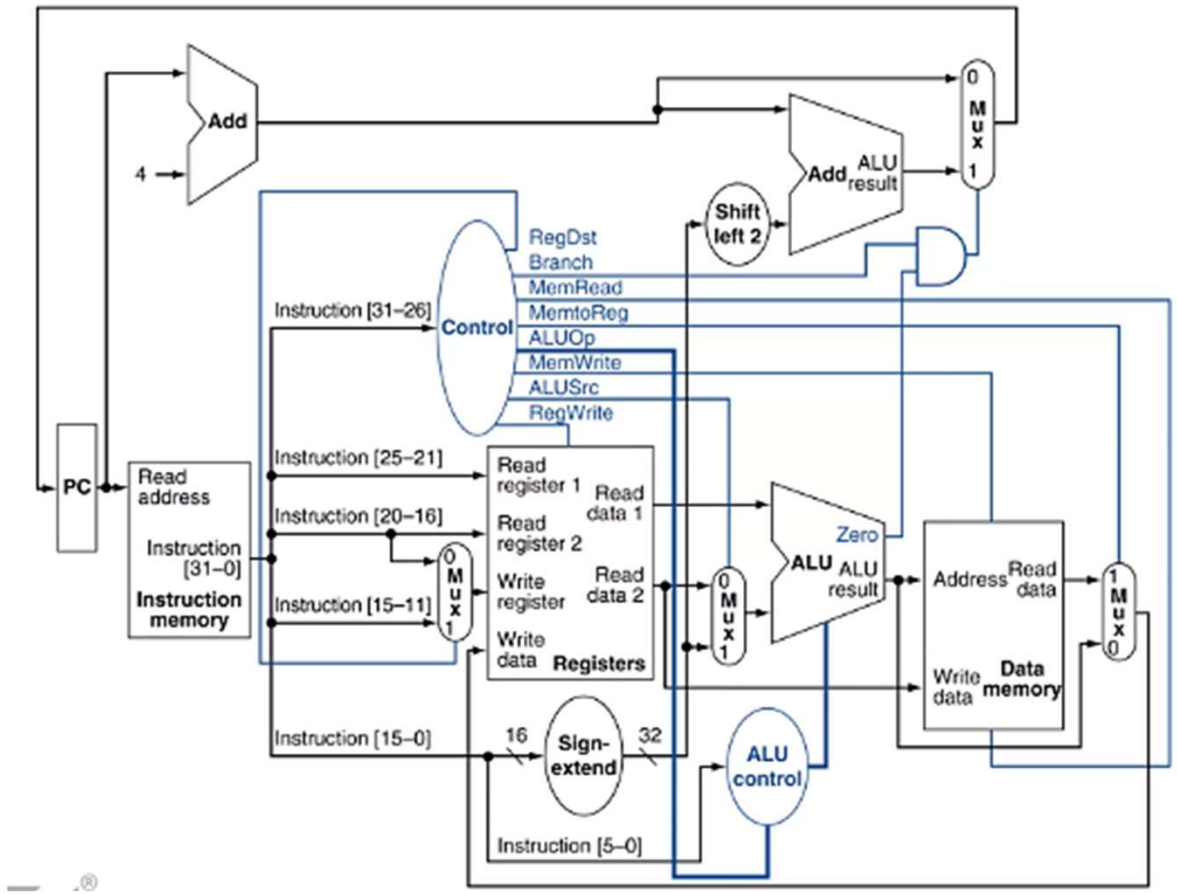
Semnan University

The Main Control Unit

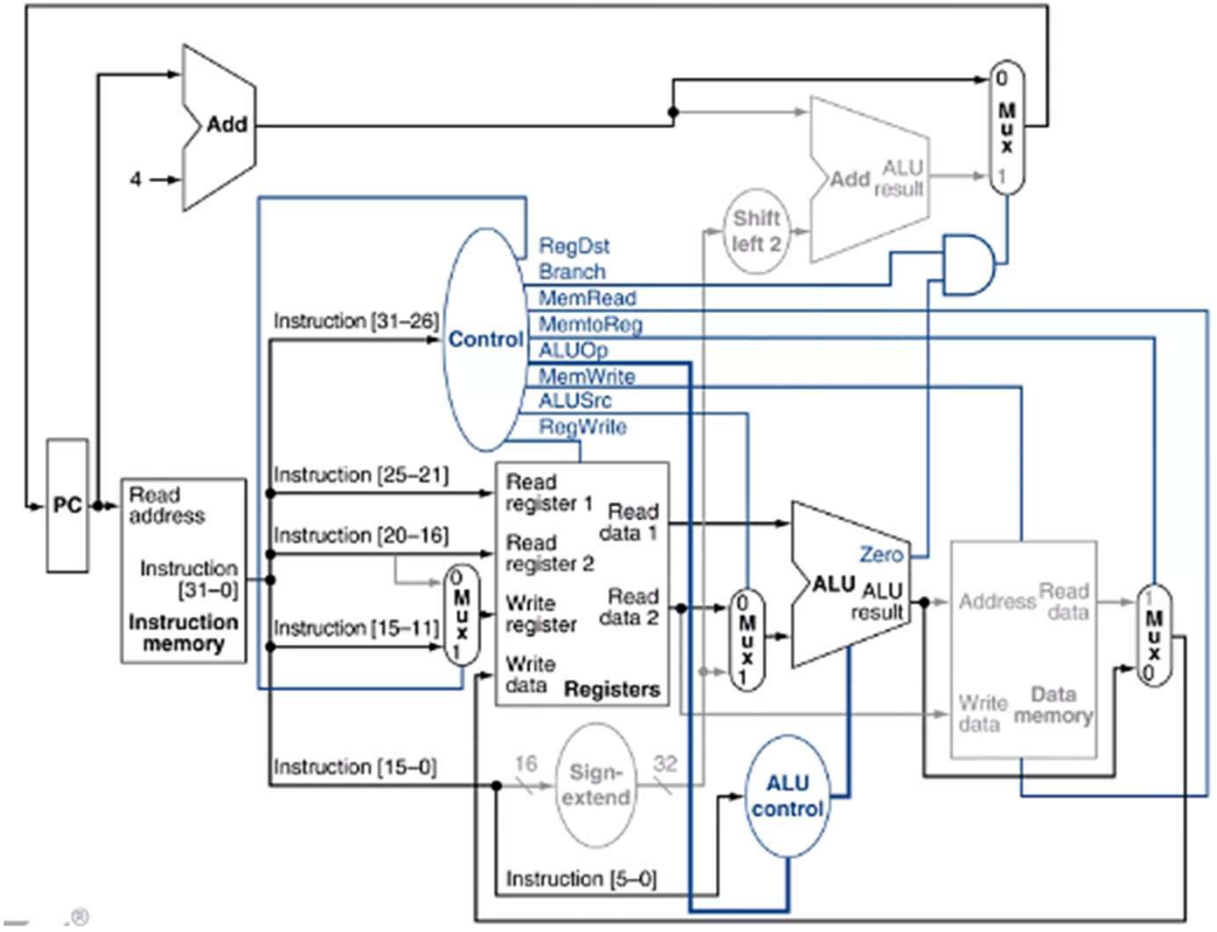
- Control signals derived from instruction



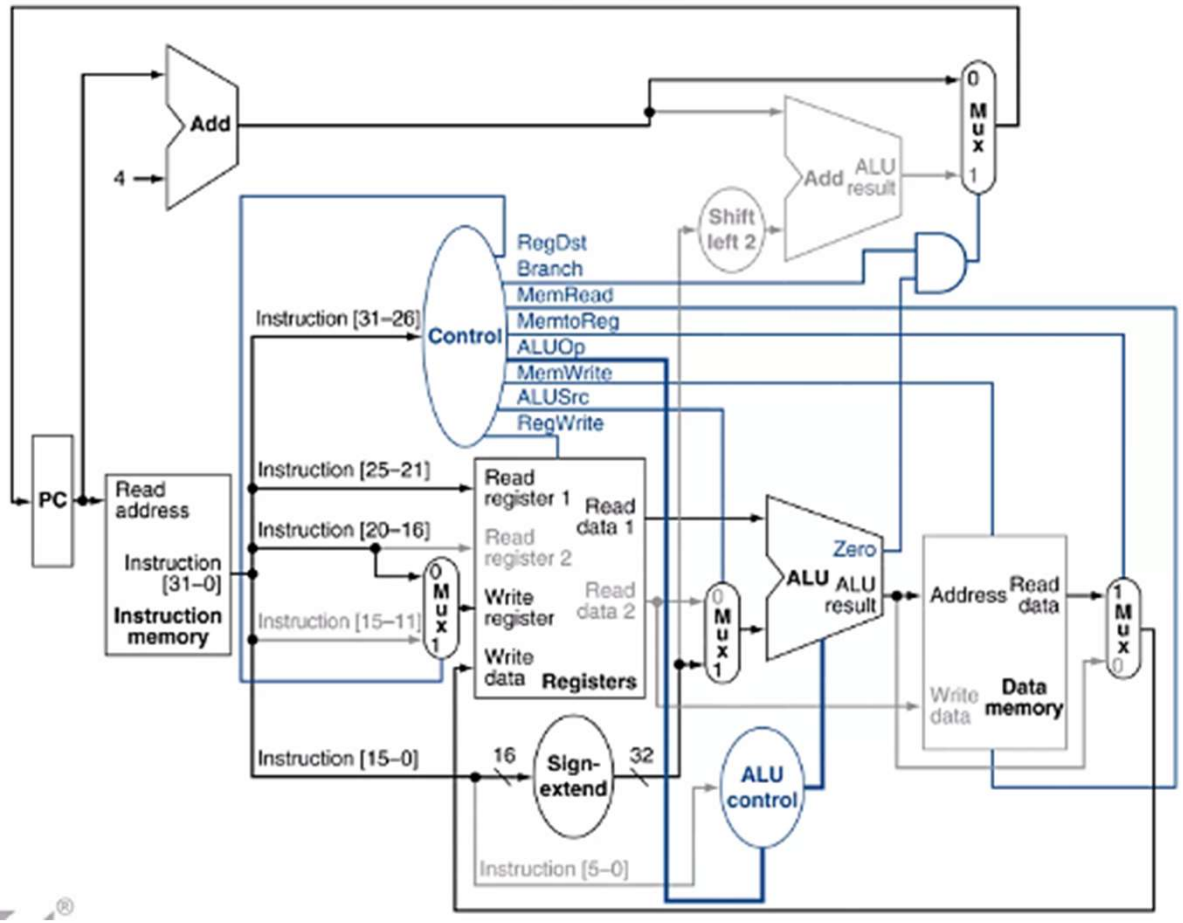
Datapath with Control



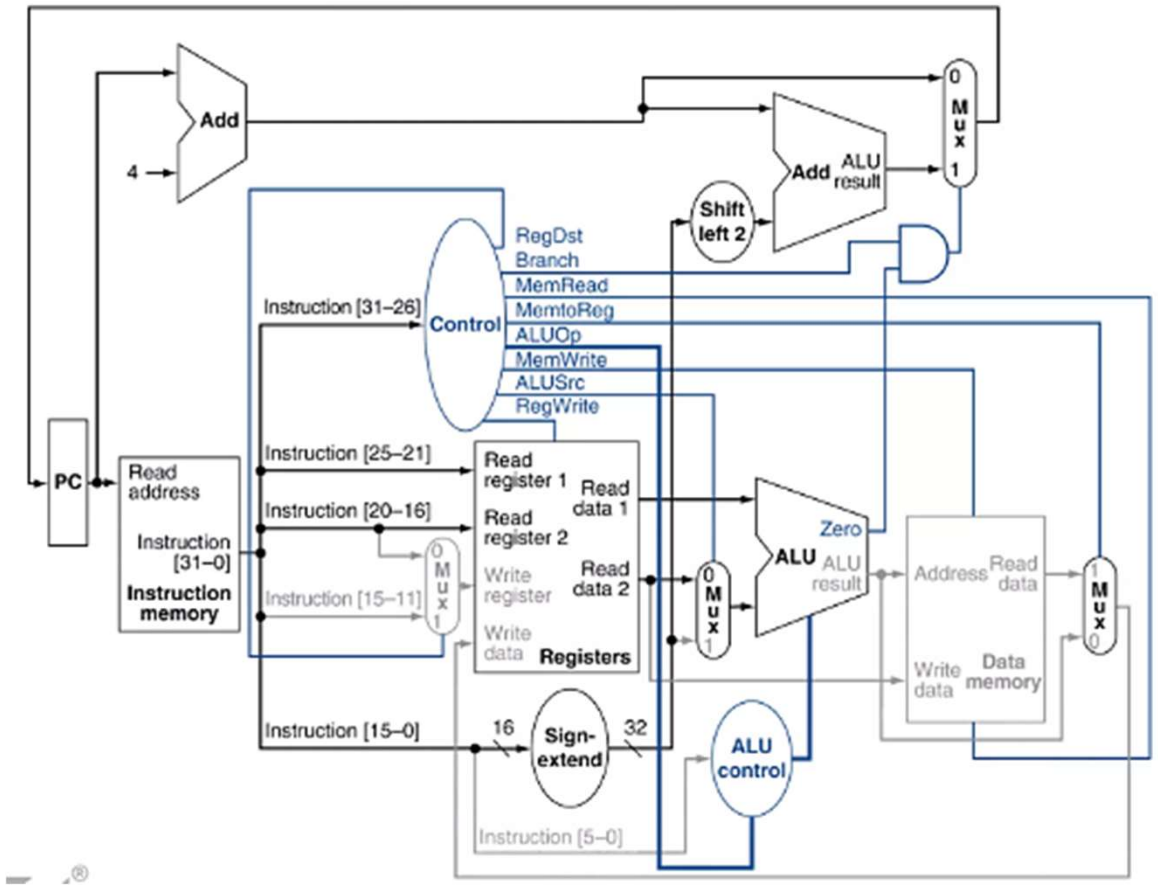
R-Type Instructions



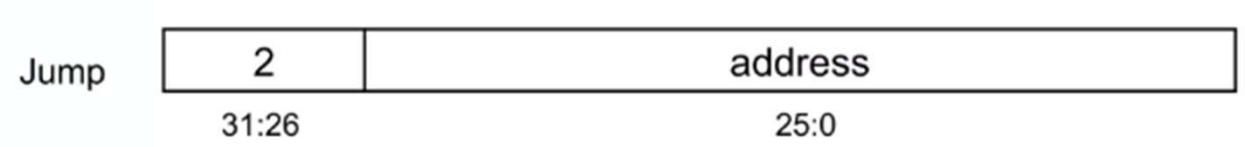
Load Instructions



Branch-on-equal Instruction



Implementing Jumps



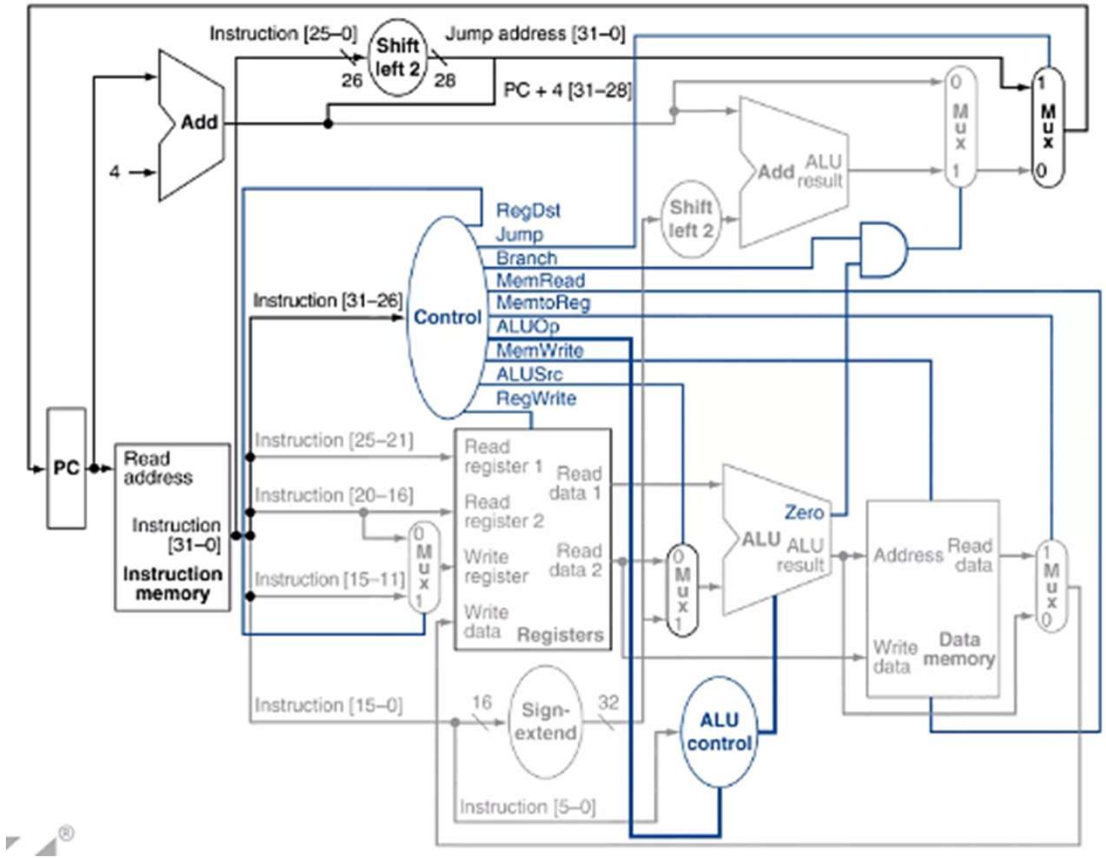
Jump uses word address

Update PC with concatenation of:

- Top 4 bits of old PC
- 26-bit jump address
- 00

Need an extra control signal decoded from opcode

Datapath with Jumps Added



Performance Issues

Longest delay determines clock period

- Critical path: load instruction
- Instruction memory → register file → ALU → data memory → register file

Not feasible to vary period for different instructions

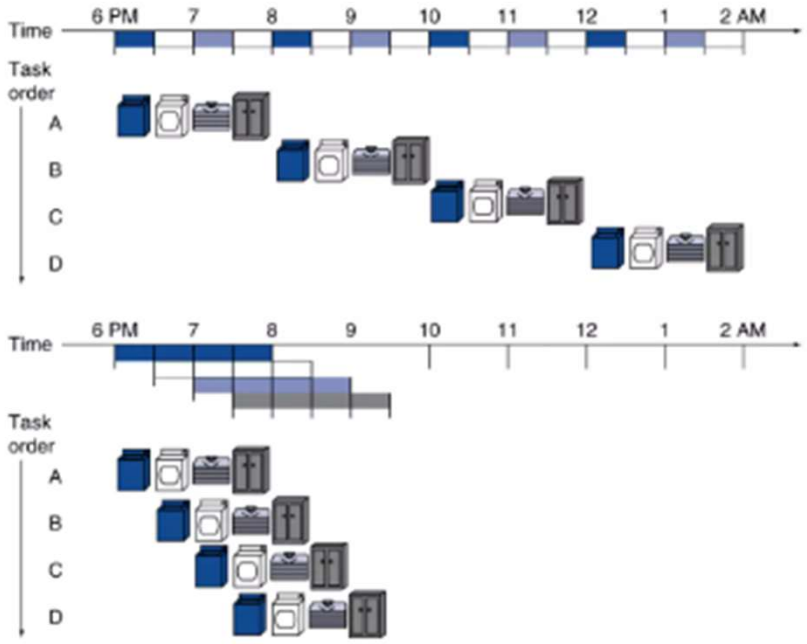
Violates design principle

- Making the common case fast

We will improve performance by pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup
    = 8/3.5 = 2.3
- Non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

MIPS Pipeline

Semnan University

- Five stages, one step per stage:
  - **IF:** Instruction fetch from memory
  - **ID:** Instruction decode & register read
  - **EX:** Execute operation or calculate address
  - **MEM:** Access memory operand
  - **WB:** Write result back to register

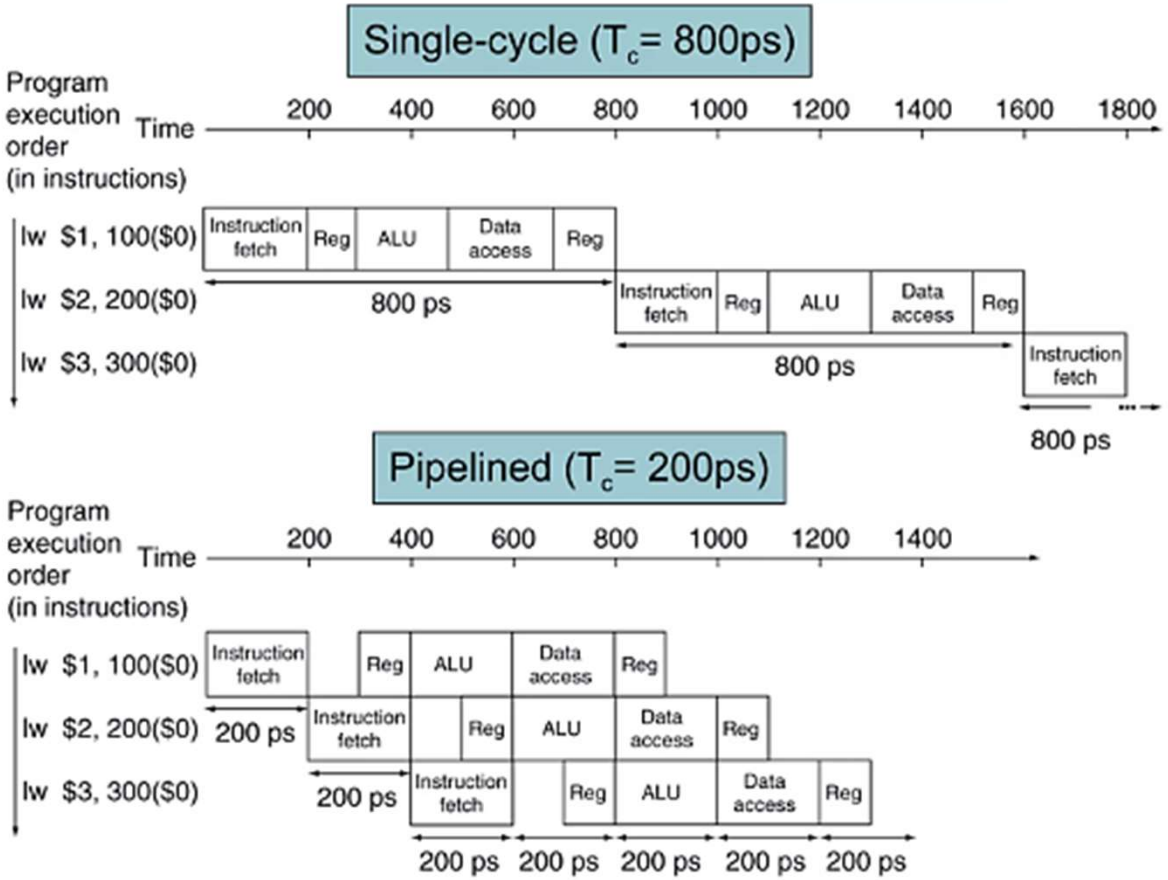
Pipeline Performance

Semnan University

- Assume time for stages is:
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance



Pipeline Speedup

Semnan University

- If all stages are balanced
  - i.e., all take the same time

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining and ISA Design

MIPS ISA designed for pipelining

- **All instructions are 32-bits**
  - Easier to fetch and decode in one cycle
  - c.f. x86: 1- to 17-byte instructions
- **Few and regular instruction formats**
  - Can decode and read registers in one step
- **Load/store addressing**
  - Can calculate address in 3rd stage, access memory in 4th stage
- **Alignment of memory operands**
  - Memory access takes only one cycle

Hazards

Situations that prevent starting the next instruction in the next cycle

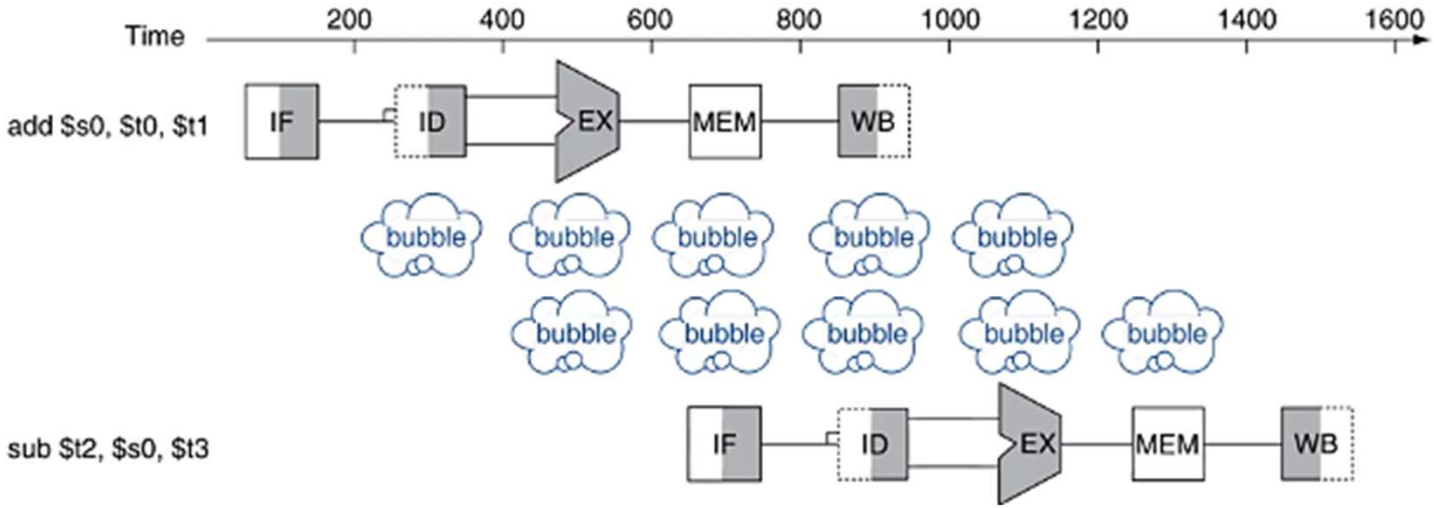
- **Structure hazards**
  - A required resource is busy
- **Data hazard**
  - Need to wait for previous instruction to complete its data read/write
- **Control hazard**
  - Deciding on control action depends on previous instruction

Structure Hazards

- **Conflict for use of a resource**
  - In MIPS pipeline with a single memory
    - Load/store requires data access
    - Instruction fetch would have to **stall** for that cycle
      - Would cause a pipeline “bubble”
- **Hence, pipelined datapaths require separate instruction/data memories**
  - Or separate instruction/data caches

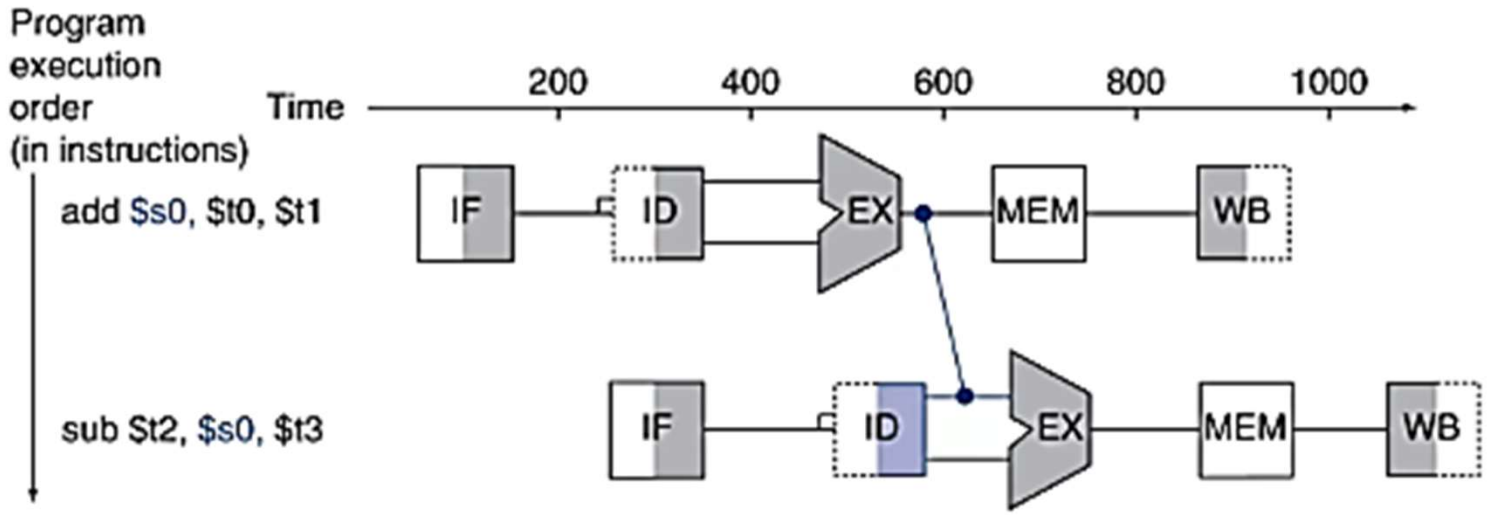
Data Hazards

- **An instruction depends on completion of data access by a previous instruction**
  - add $s0, $t0, $t1
  - sub $t2, $s0, $t3



Forwarding (aka Bypassing)

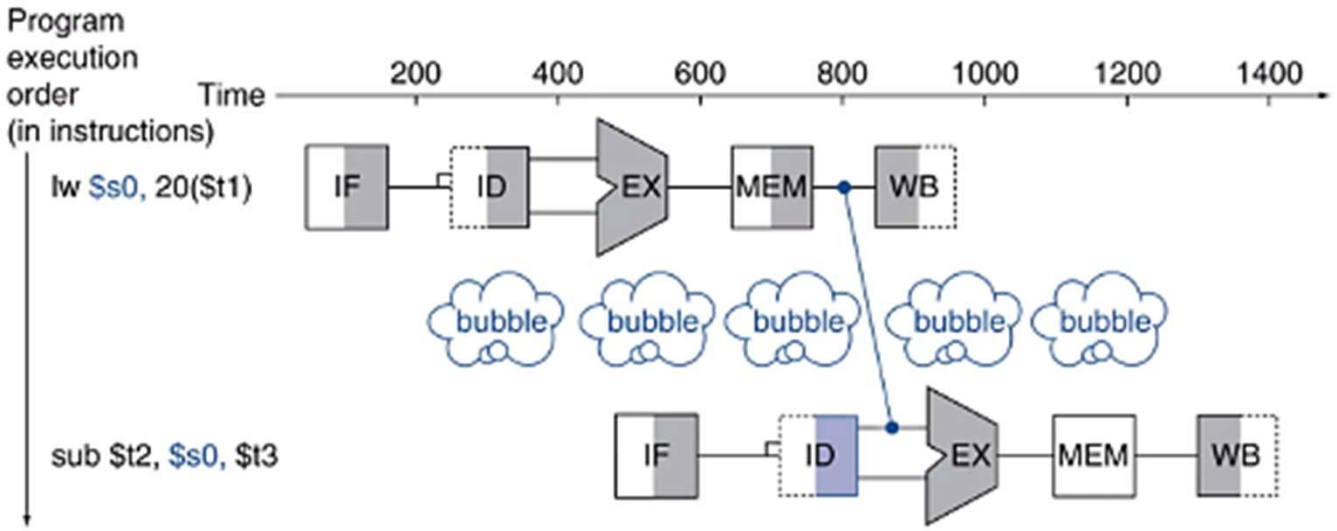
- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



Load-Use Data Hazard

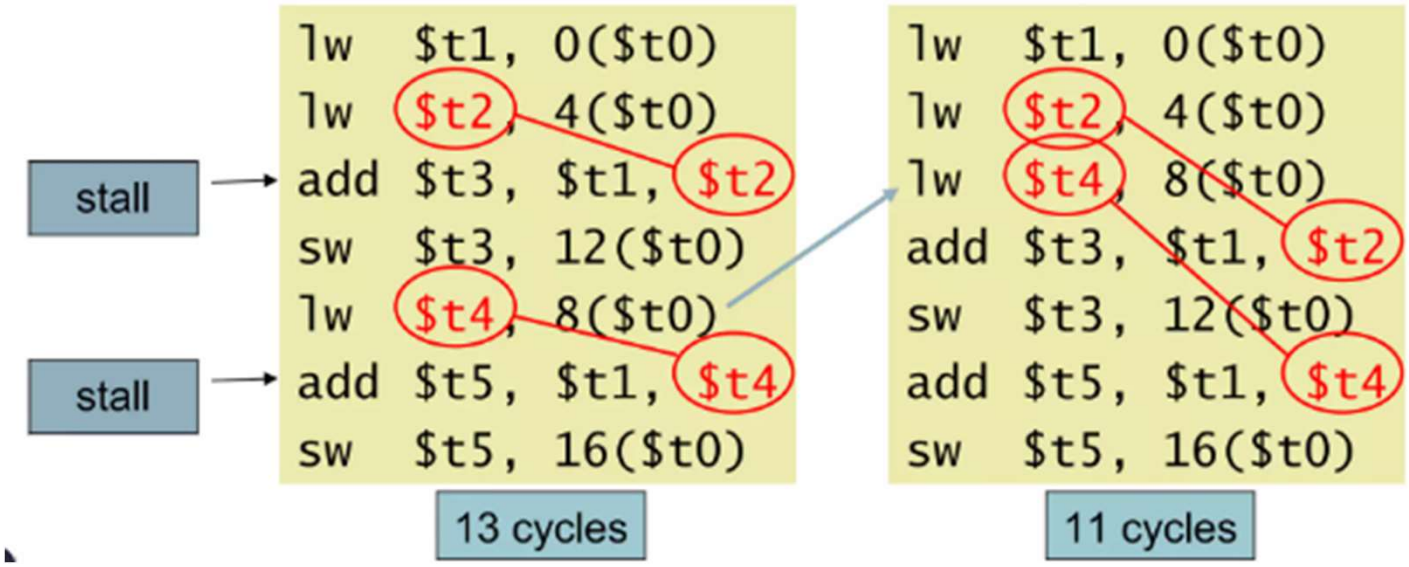
Can’t always avoid stalls by forwarding

- If value not computed when needed
- Can’t forward backward in time!



Code Scheduling to Avoid Stall

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



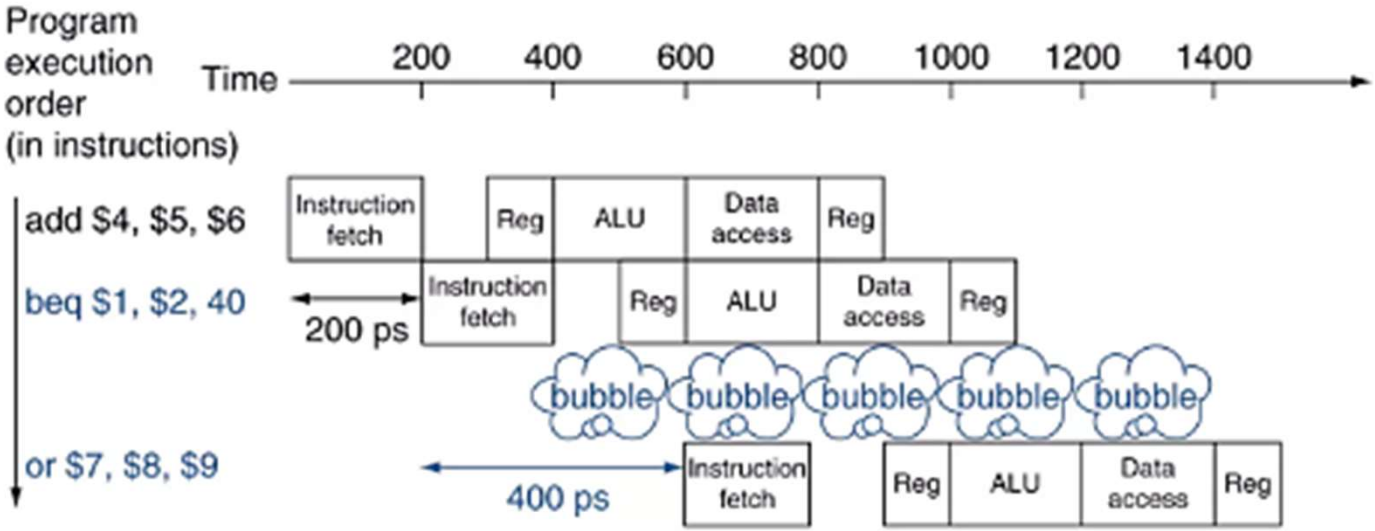
Control Hazards

Semnan University

- **Branch determines flow of control**
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- **In MIPS pipeline**
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



Branch Prediction

Longer pipelines can't readily determine branch outcome early

- Stall penalty becomes unacceptable

Predict outcome of branch

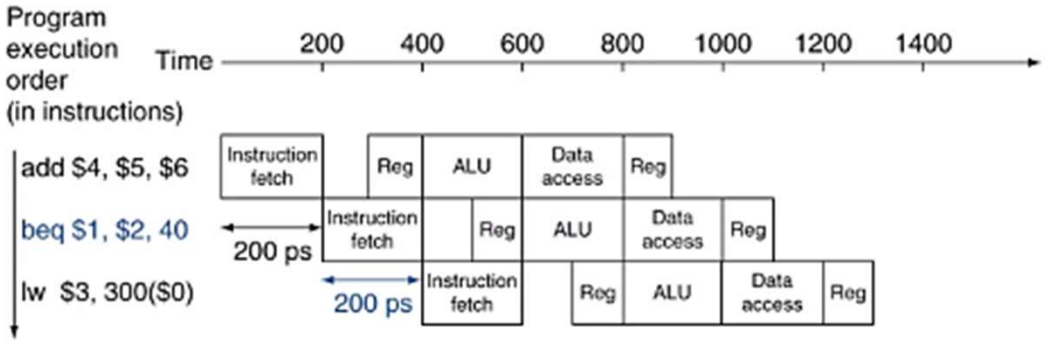
- Only stall if prediction is wrong

In MIPS pipeline

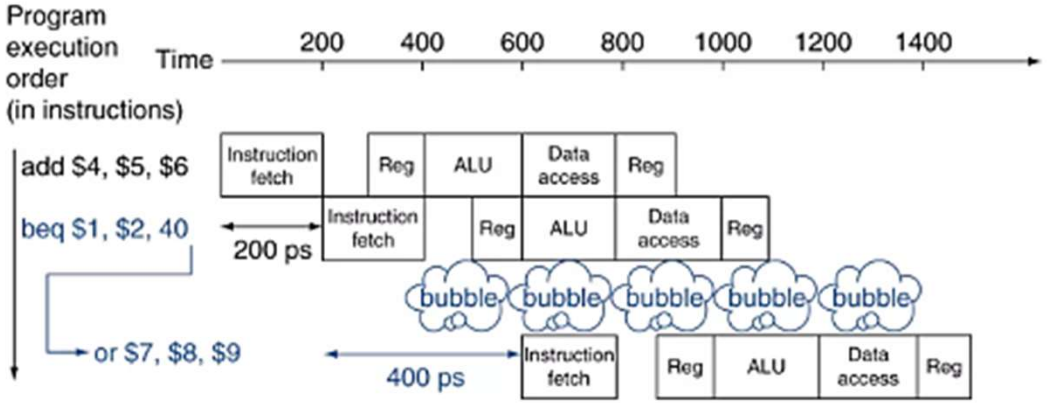
- Can predict branches not taken
- Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

Semnan University

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 the trend
  - When wrong, stall while re-fetching, and update history

Pipeline Summary

Pipelining improves performance by increasing instruction throughput

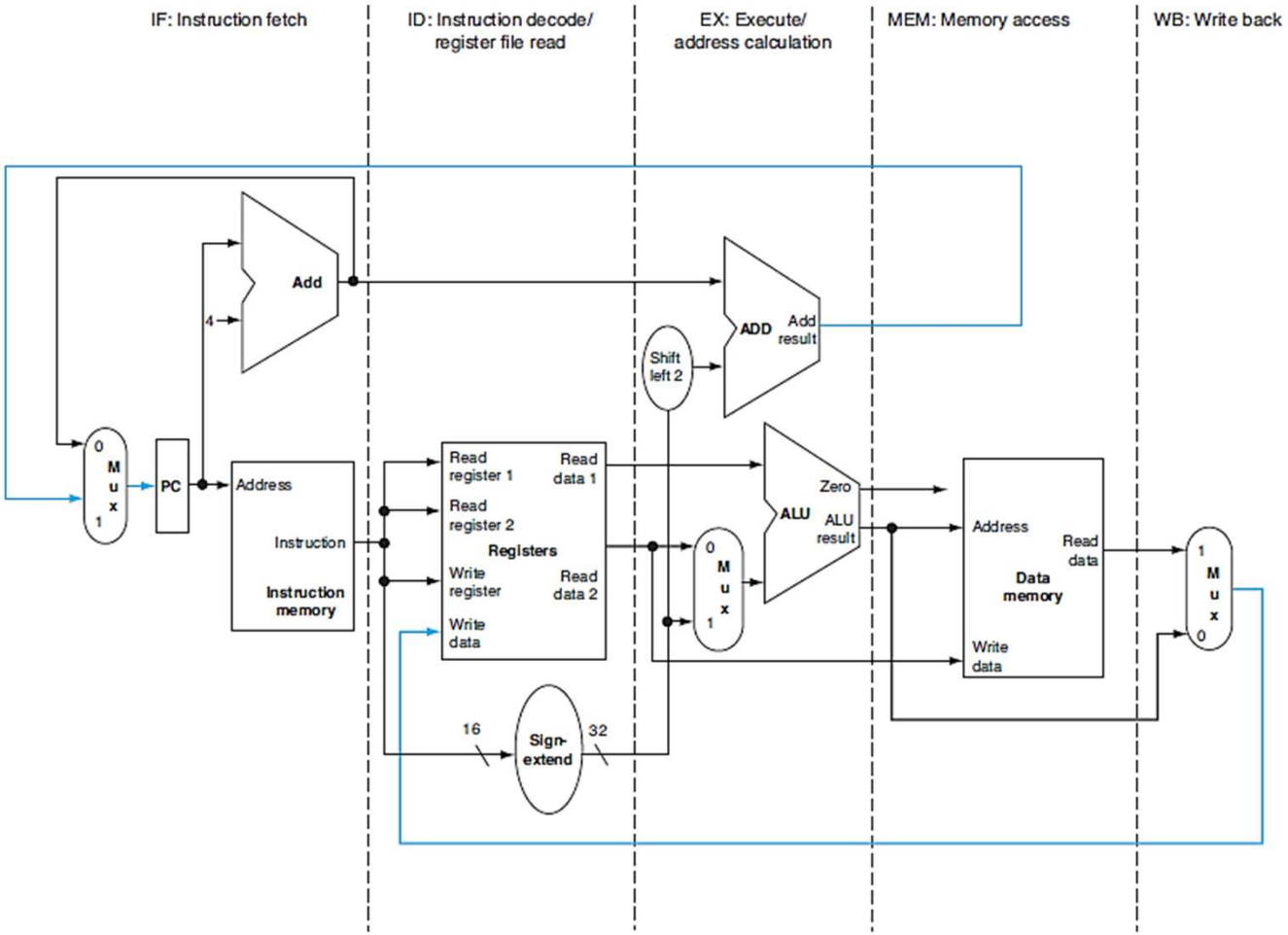
- Executes multiple instructions in parallel
- Each instruction has the same latency

Subject to hazards

- Structure, data, control

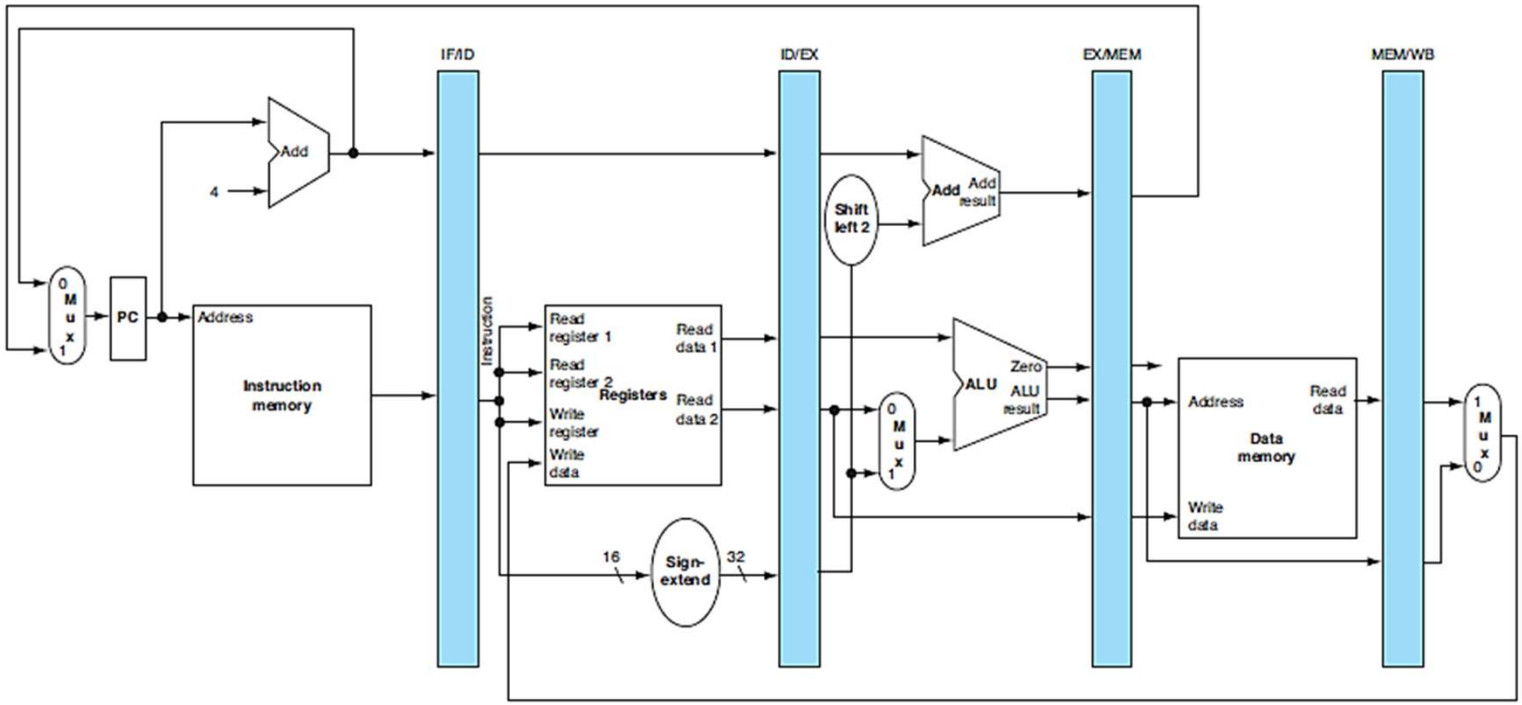
Instruction set design affects complexity of pipeline implementation

MIPS Pipelined Datapath



Pipeline Registers

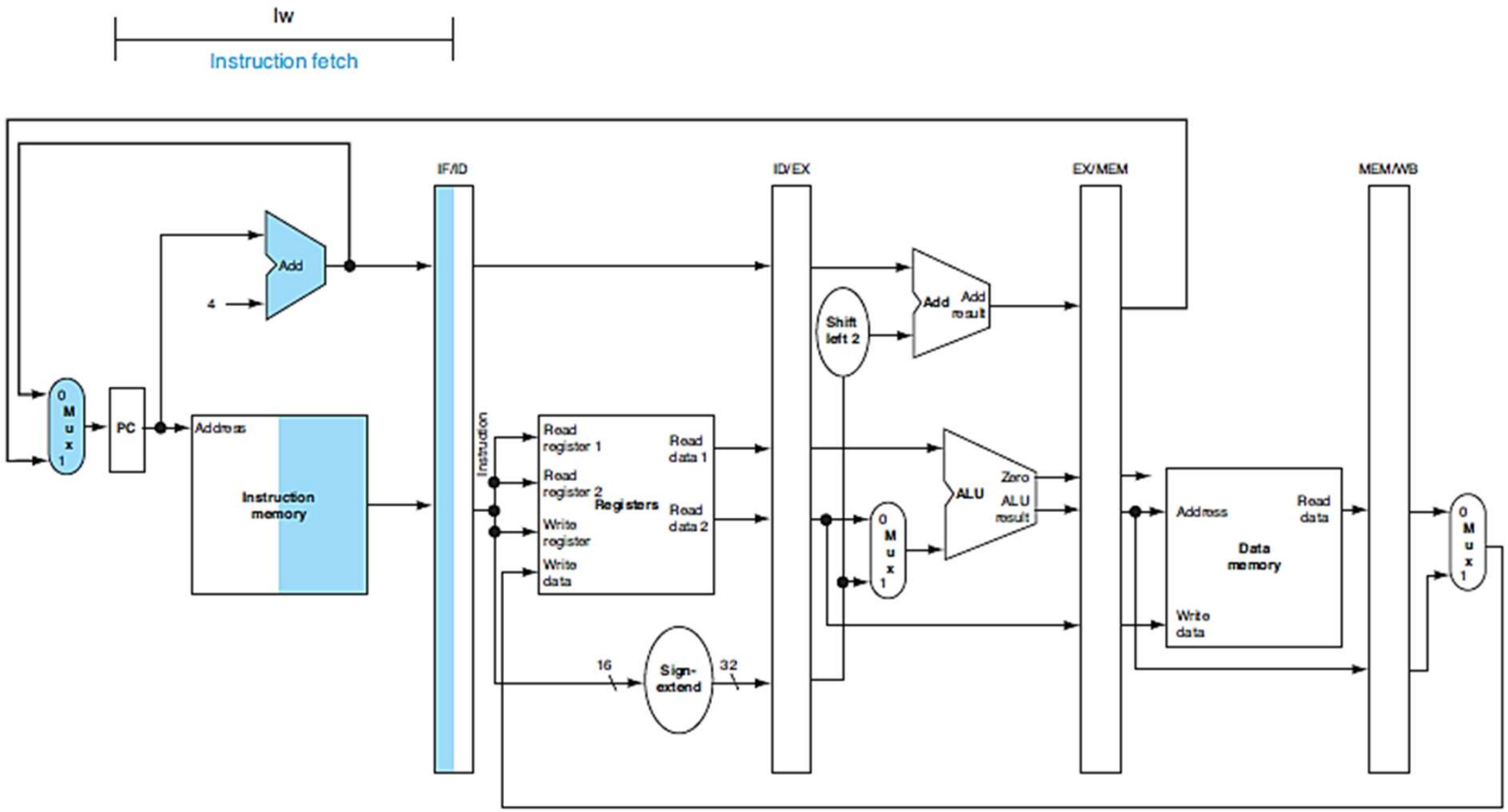
- Need registers between stages
  - To hold information produced in previous cycle



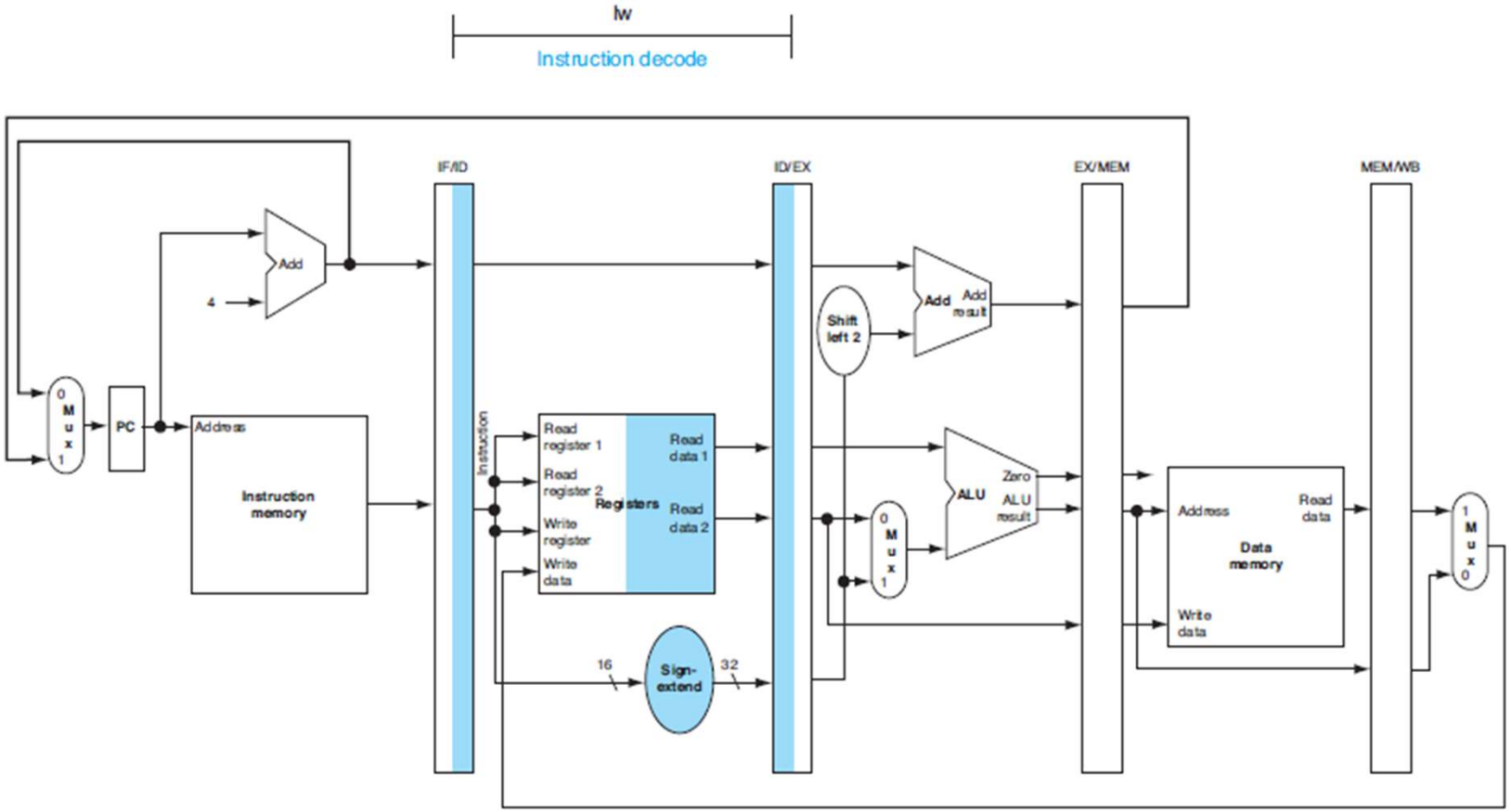
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlights resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

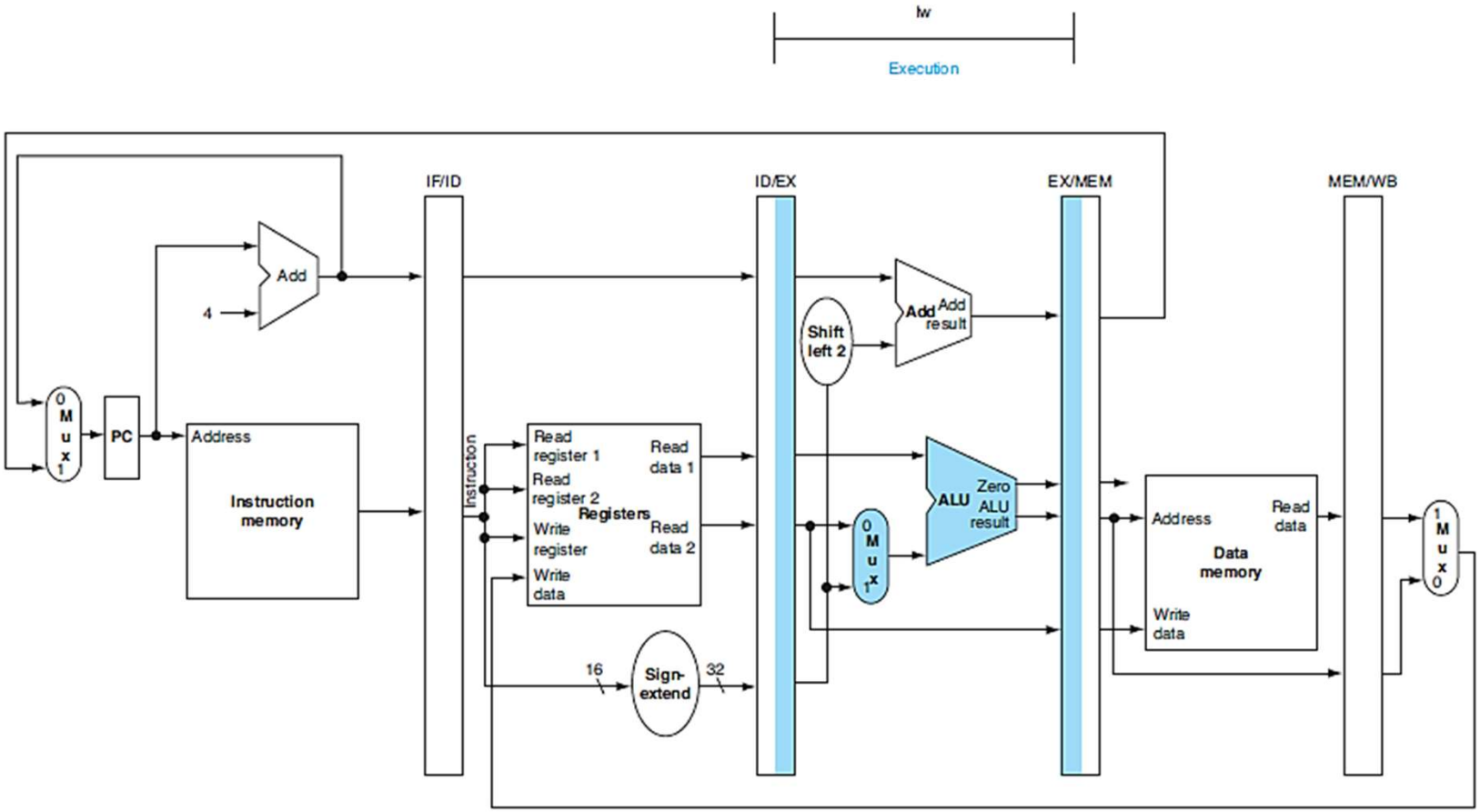
IF for Load, Store, …



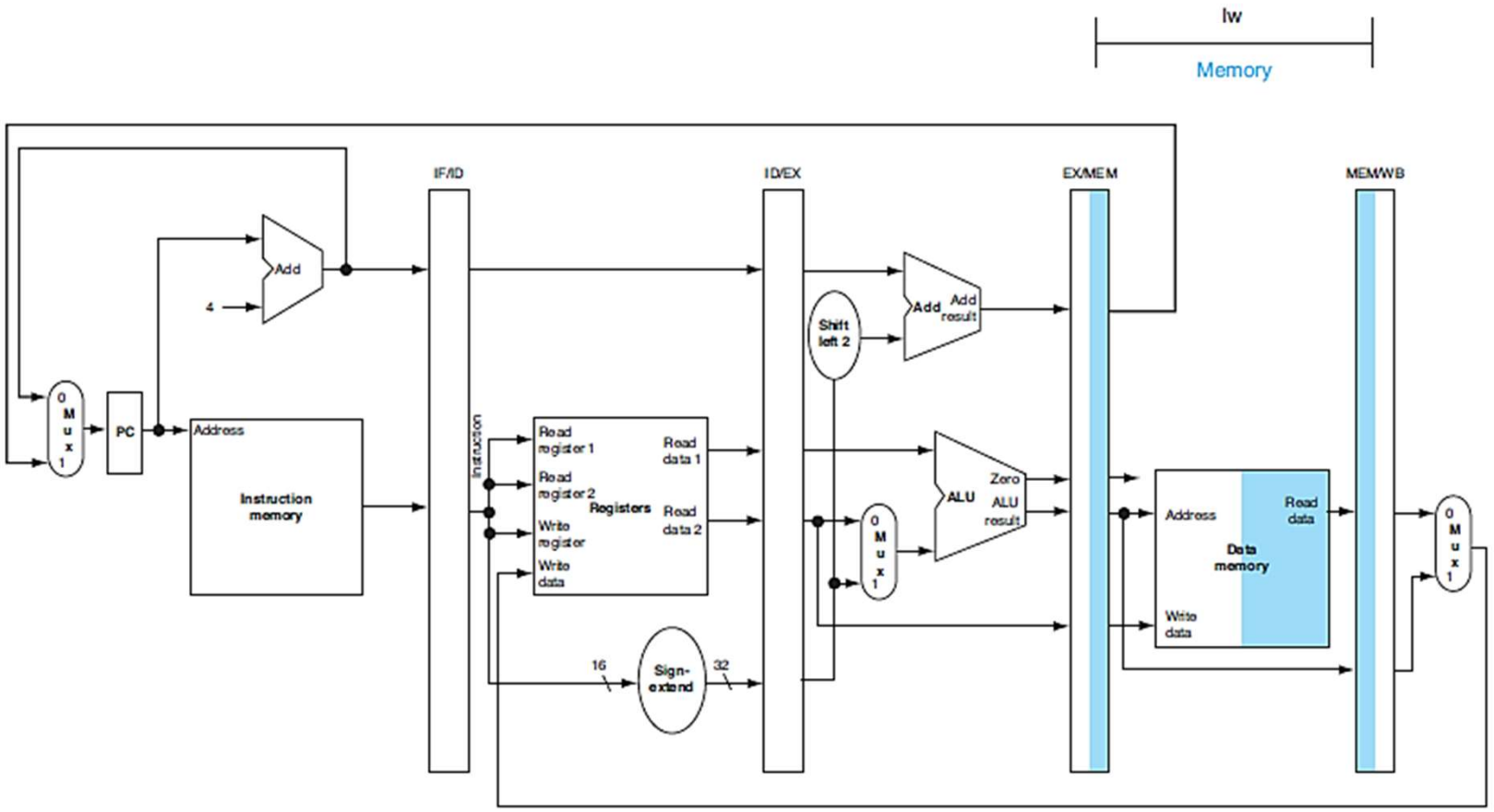
ID for Load, Store, …



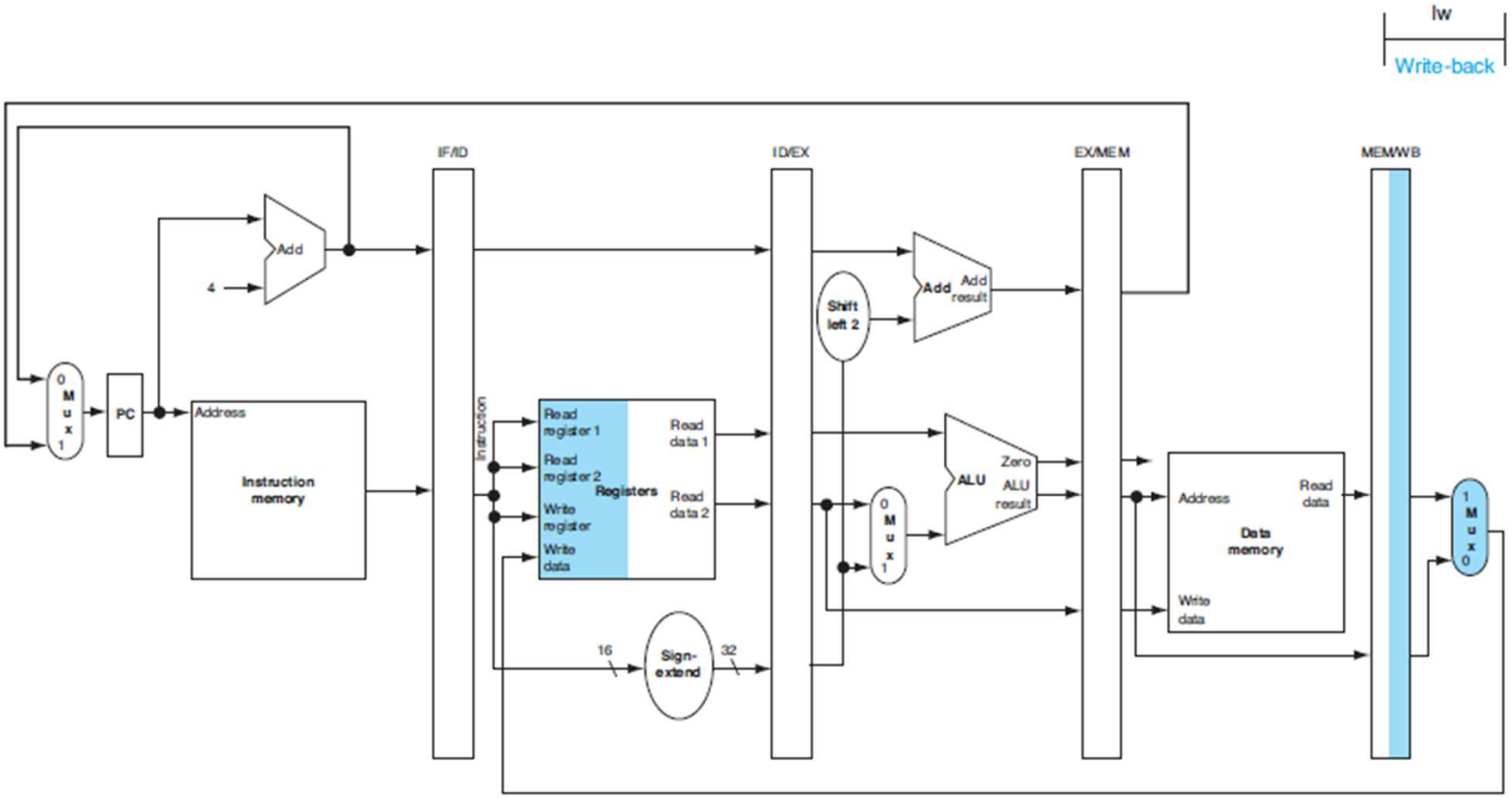
EX for Load


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32

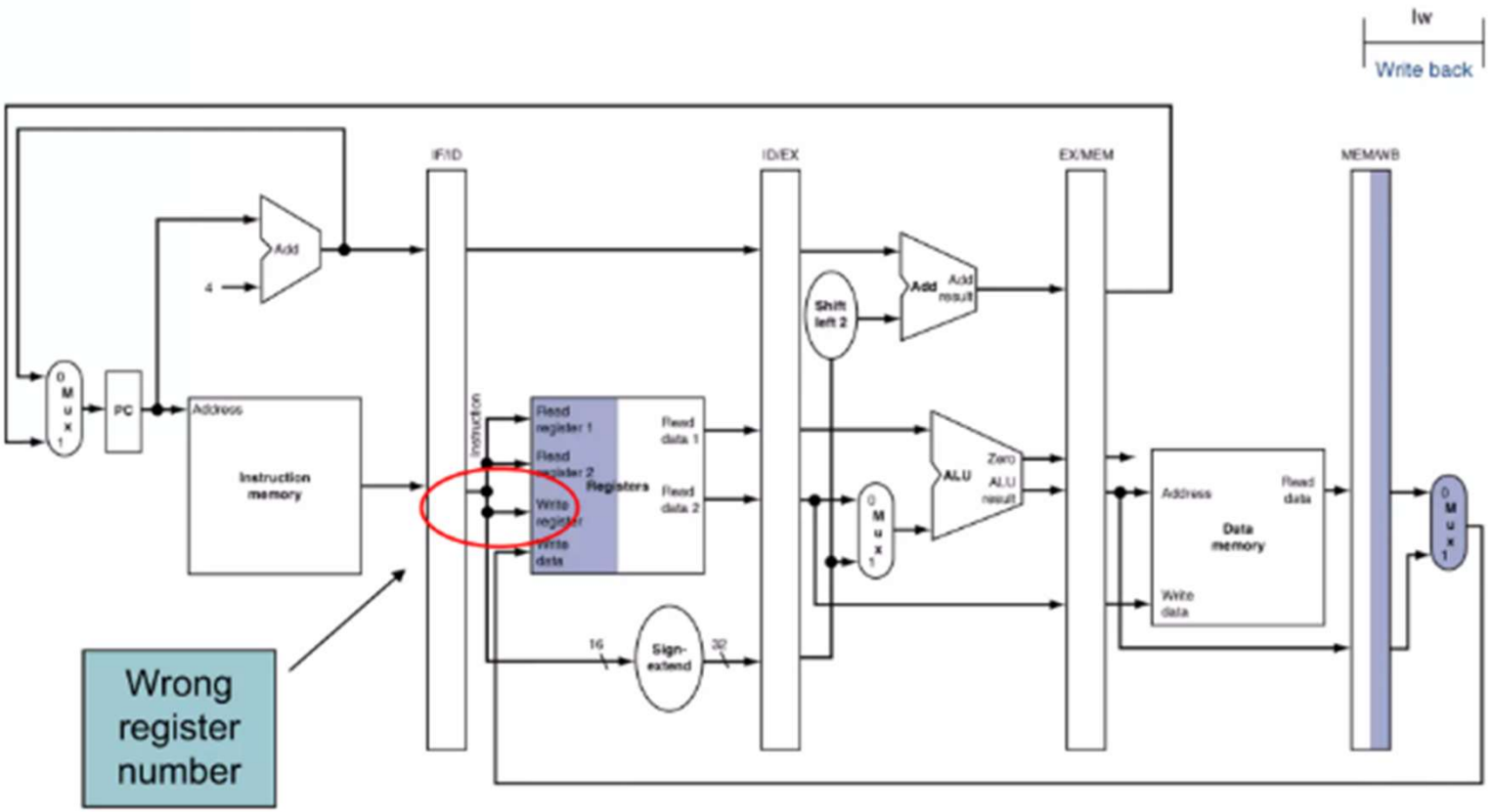
MEM for Load


lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

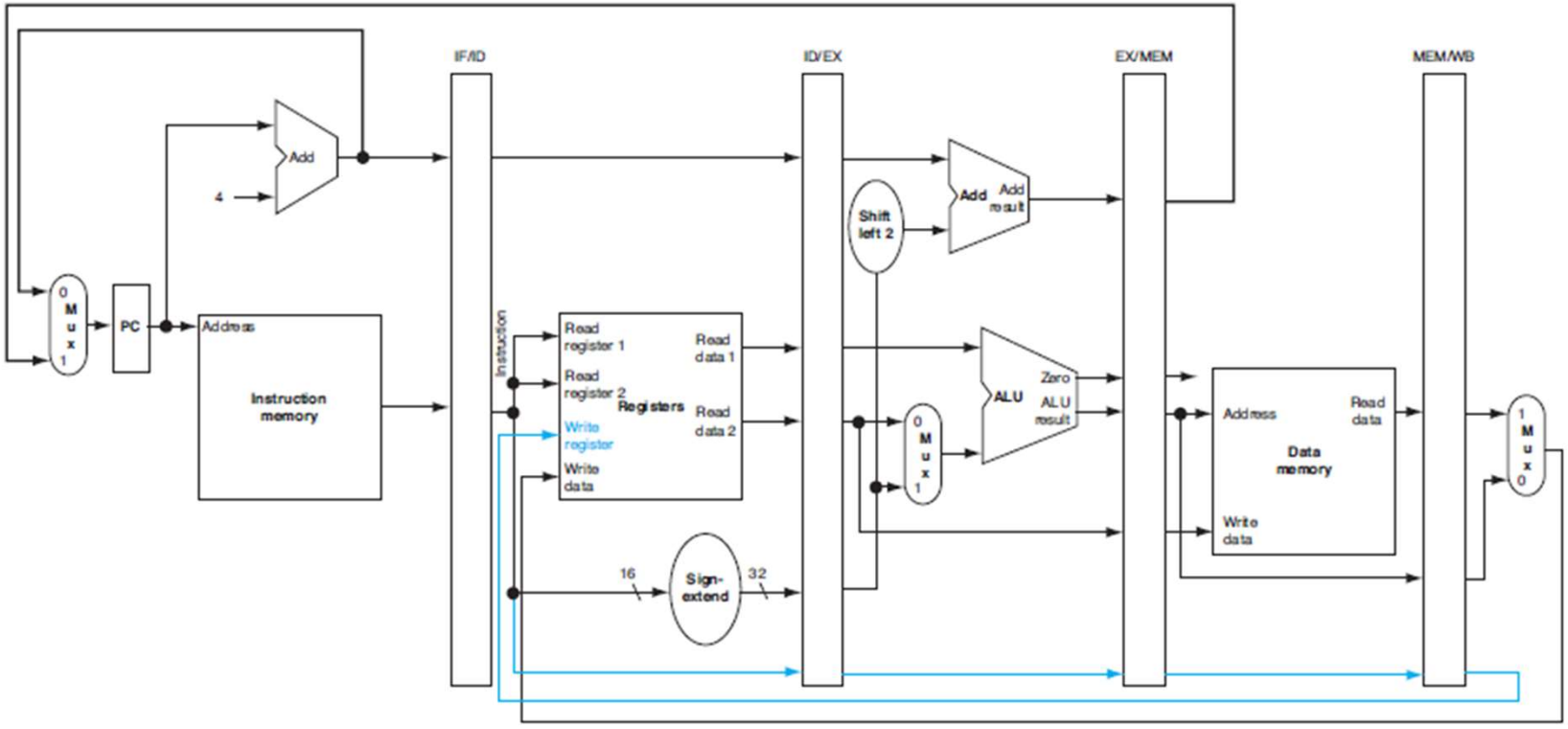
WB for Load


lw
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32

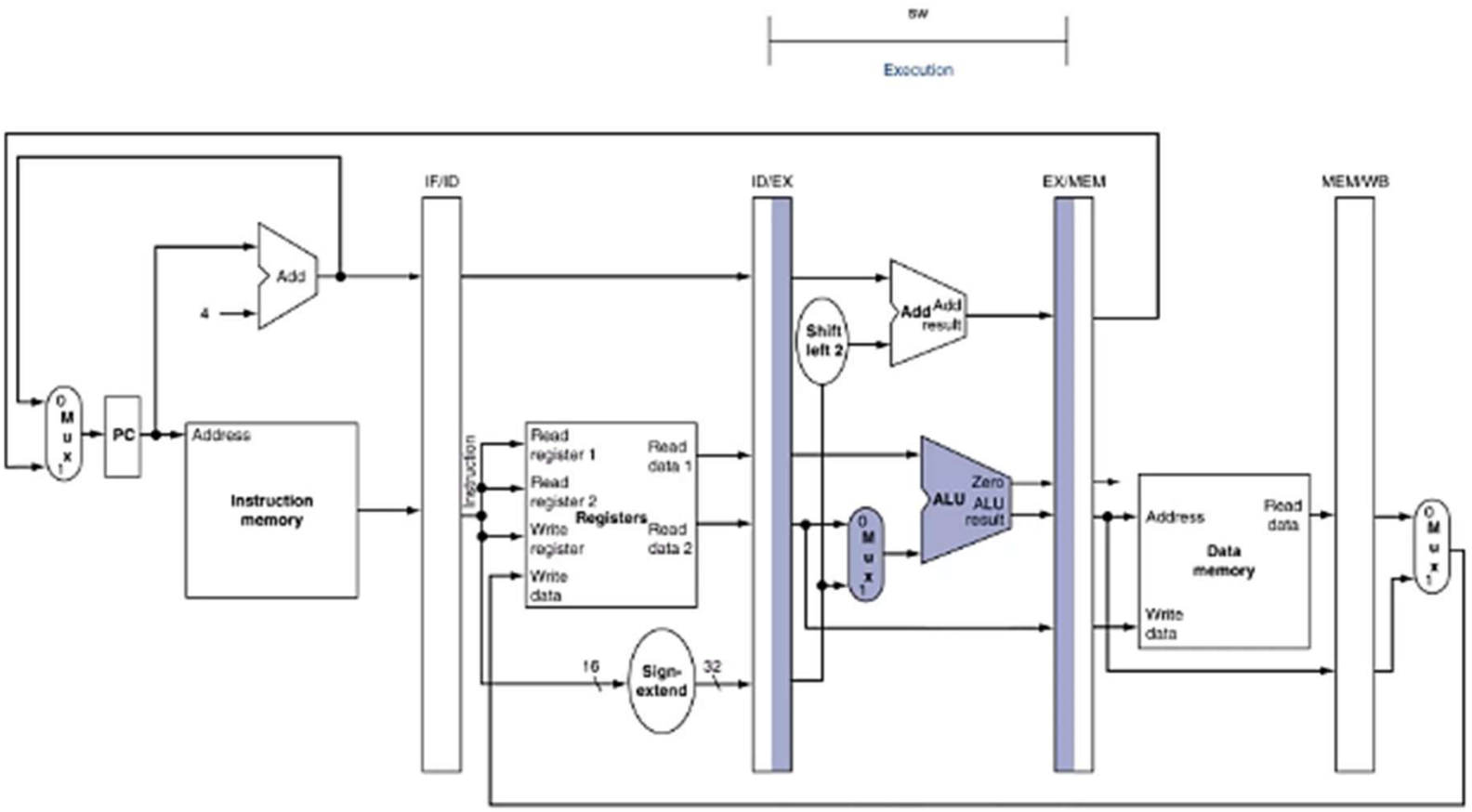
WB for Load


lw
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Registers
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
16
Sign-extend
32
Wrong register number

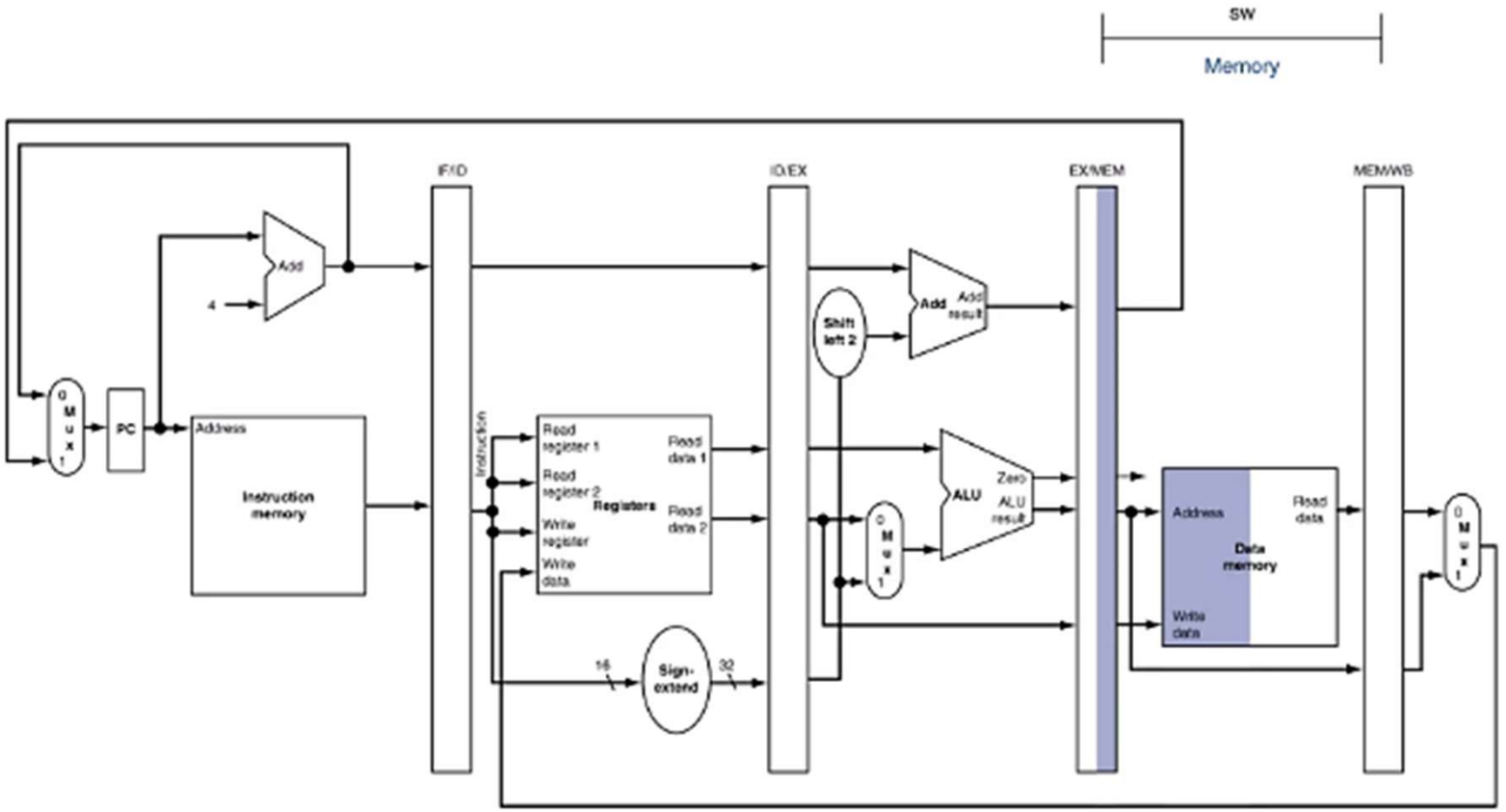
Corrected Datapath For Load



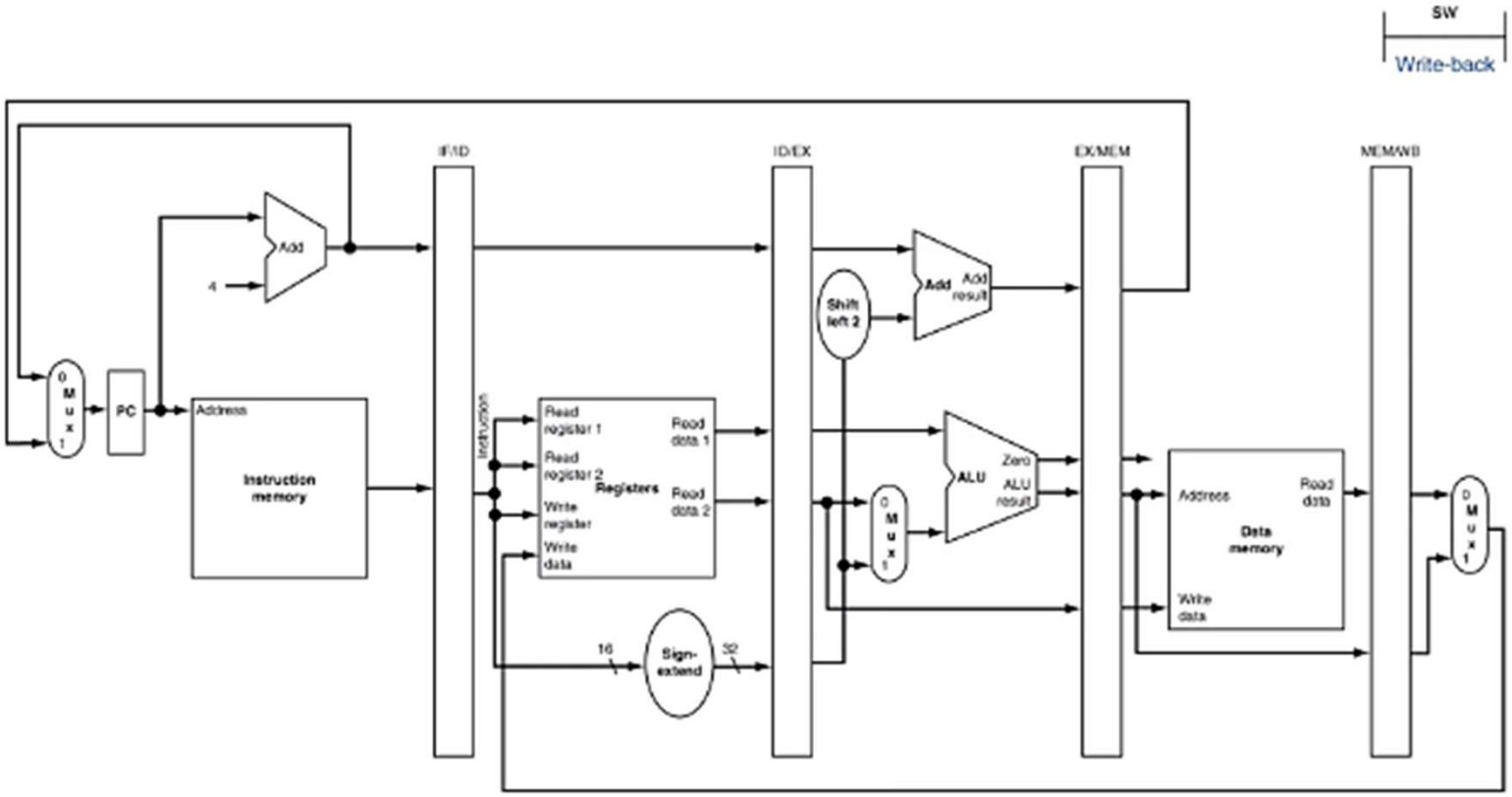
EX For Store


sw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

MEM For Store

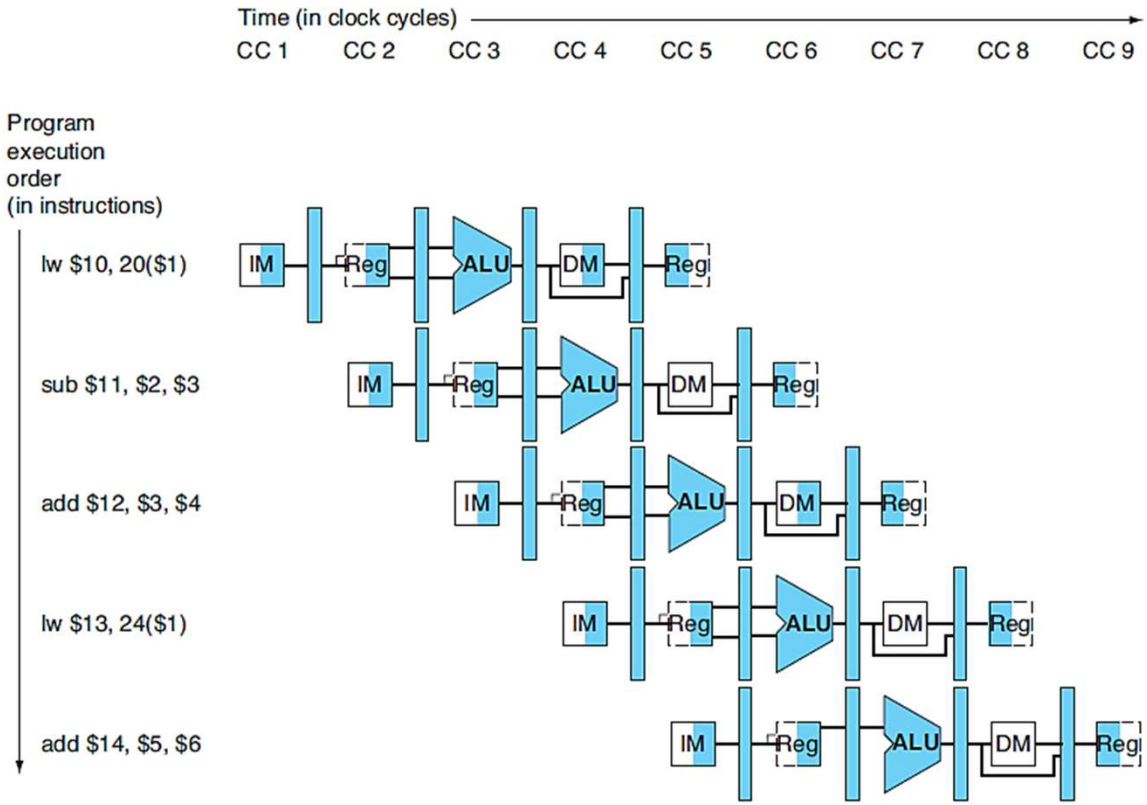


WB For Store


SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

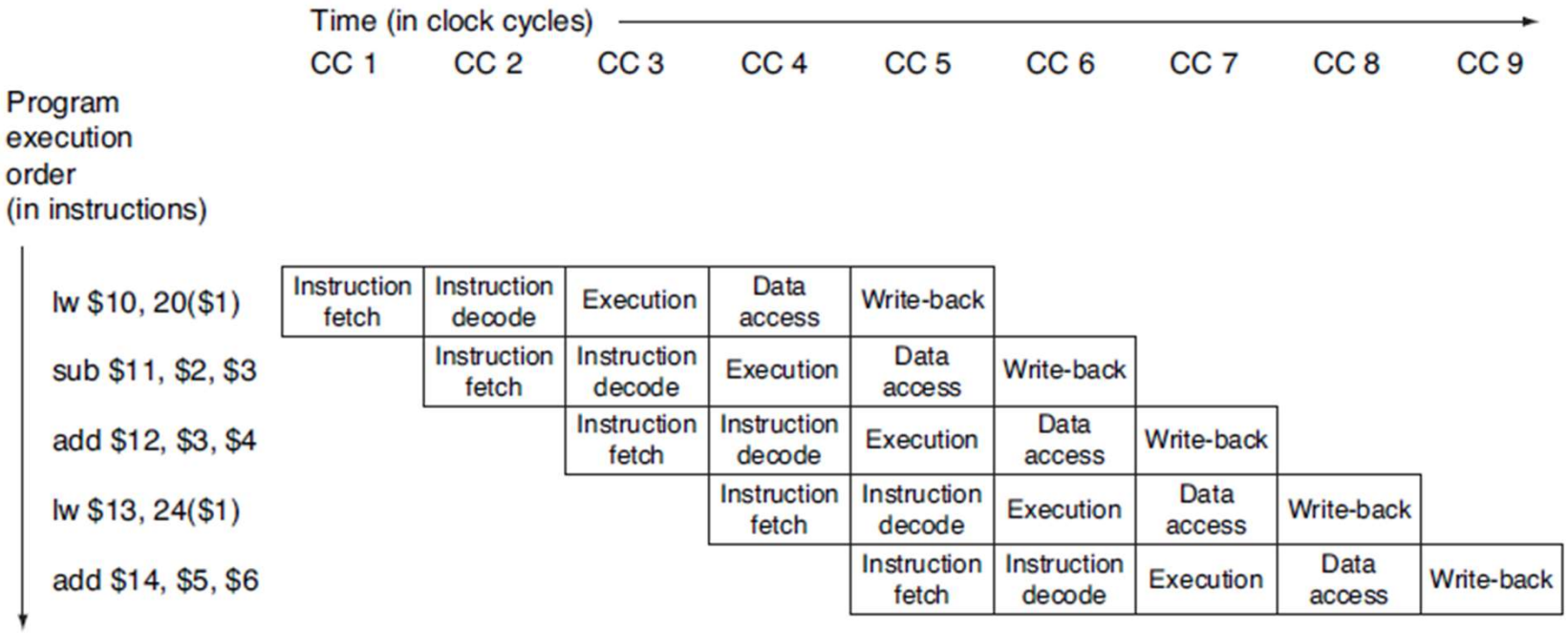
Multi-Cycle Pipeline Diagram

Form showing resource usage



Multi-Cycle Pipeline Diagram

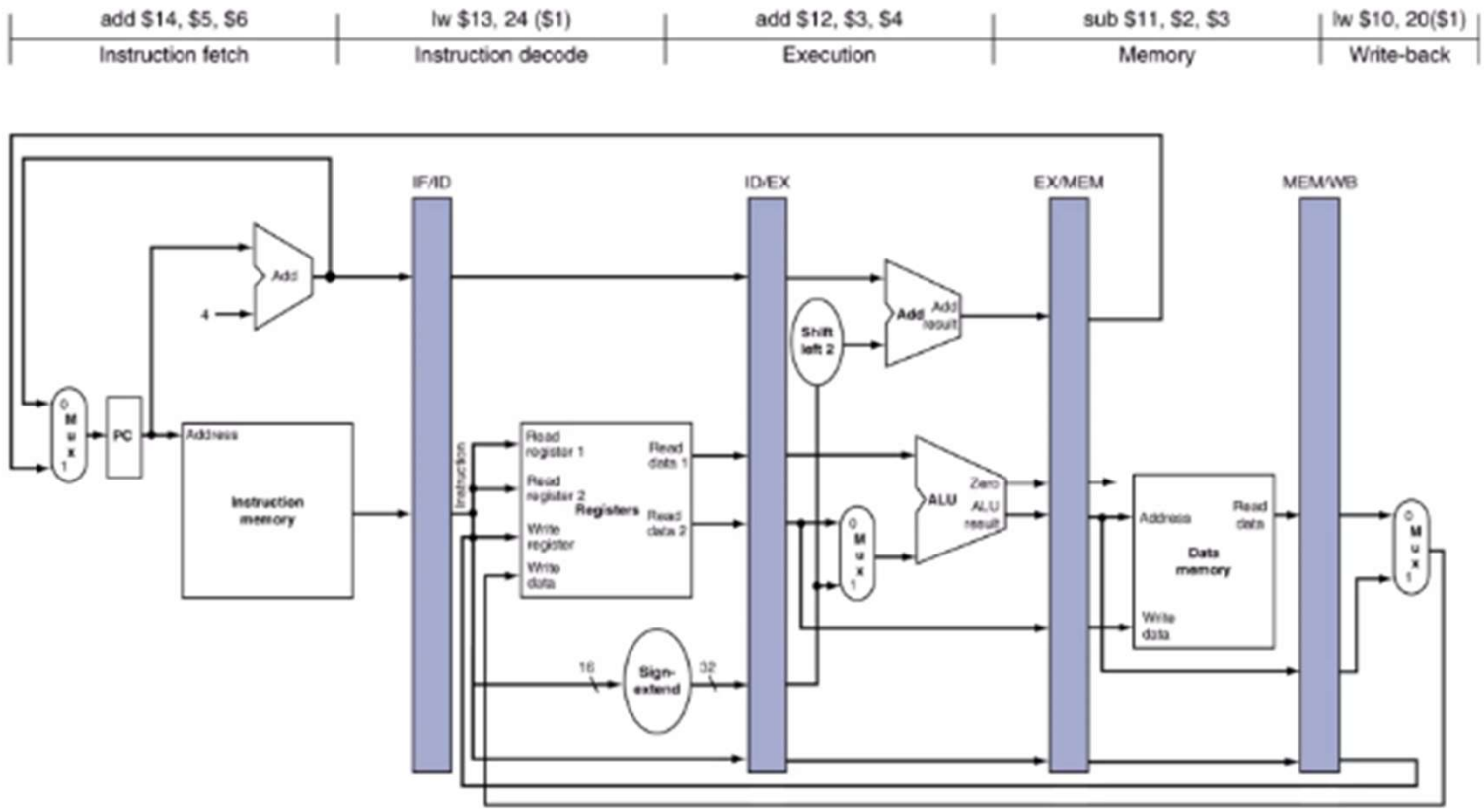
Traditional Form



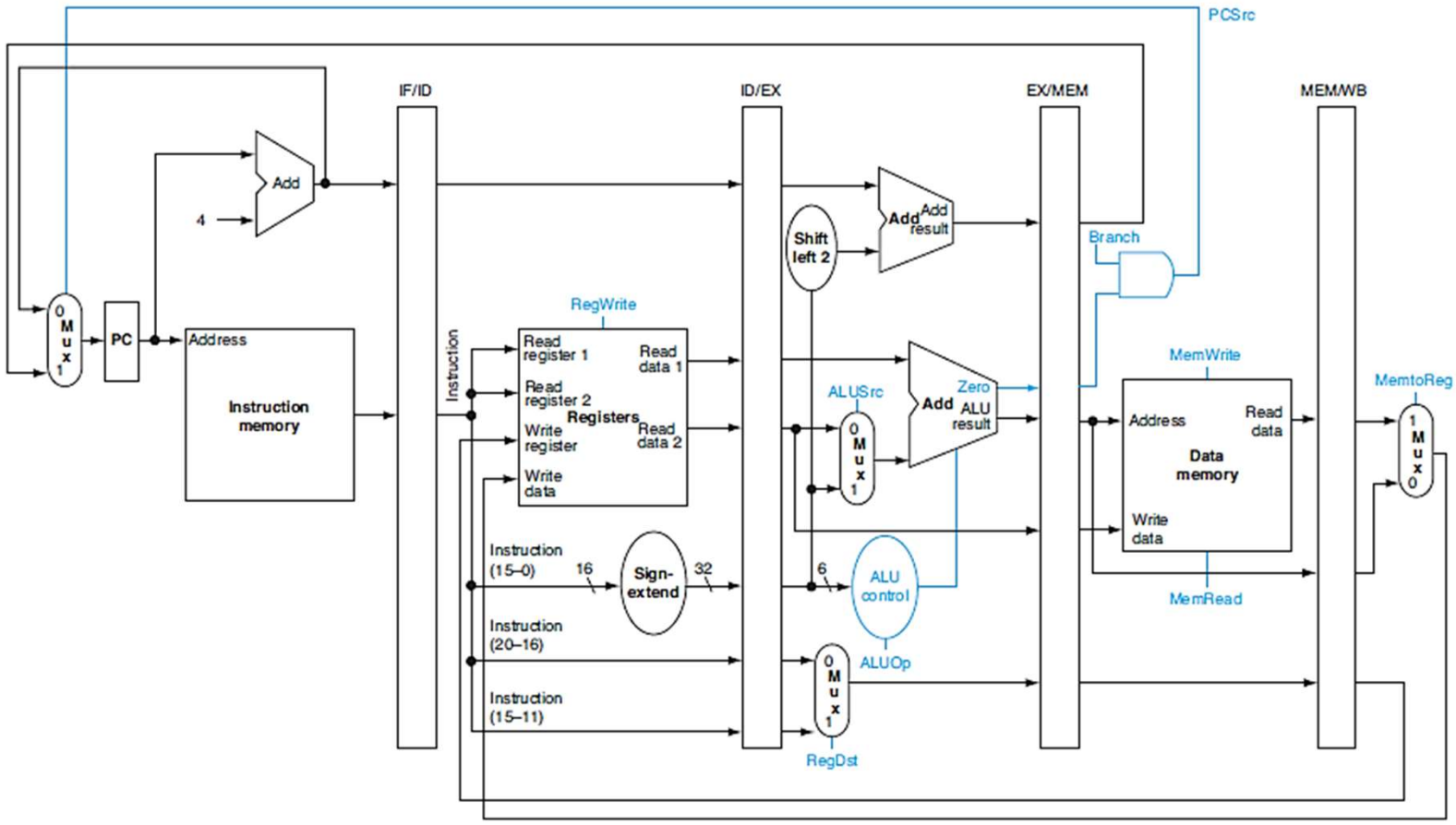
Multi-Cycle Pipeline Diagram

Semnan University

- State of pipeline in a given cycle



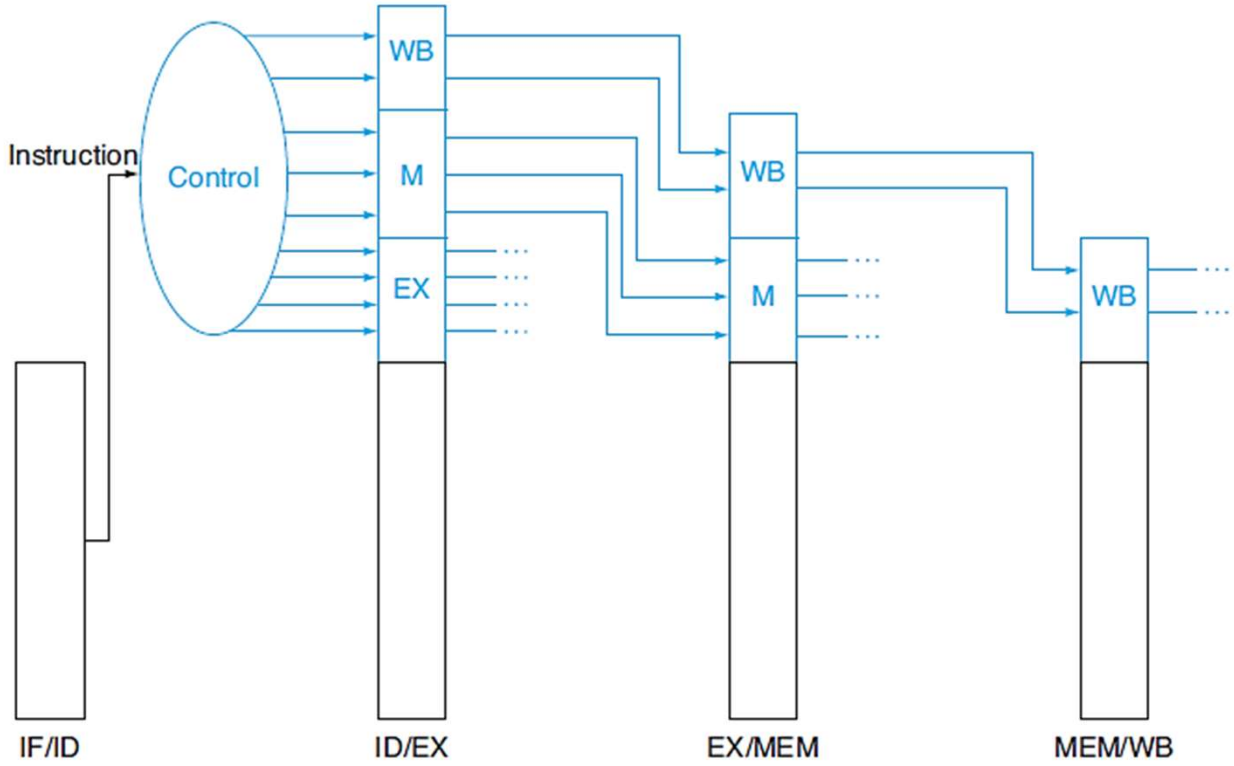
Pipeline Control



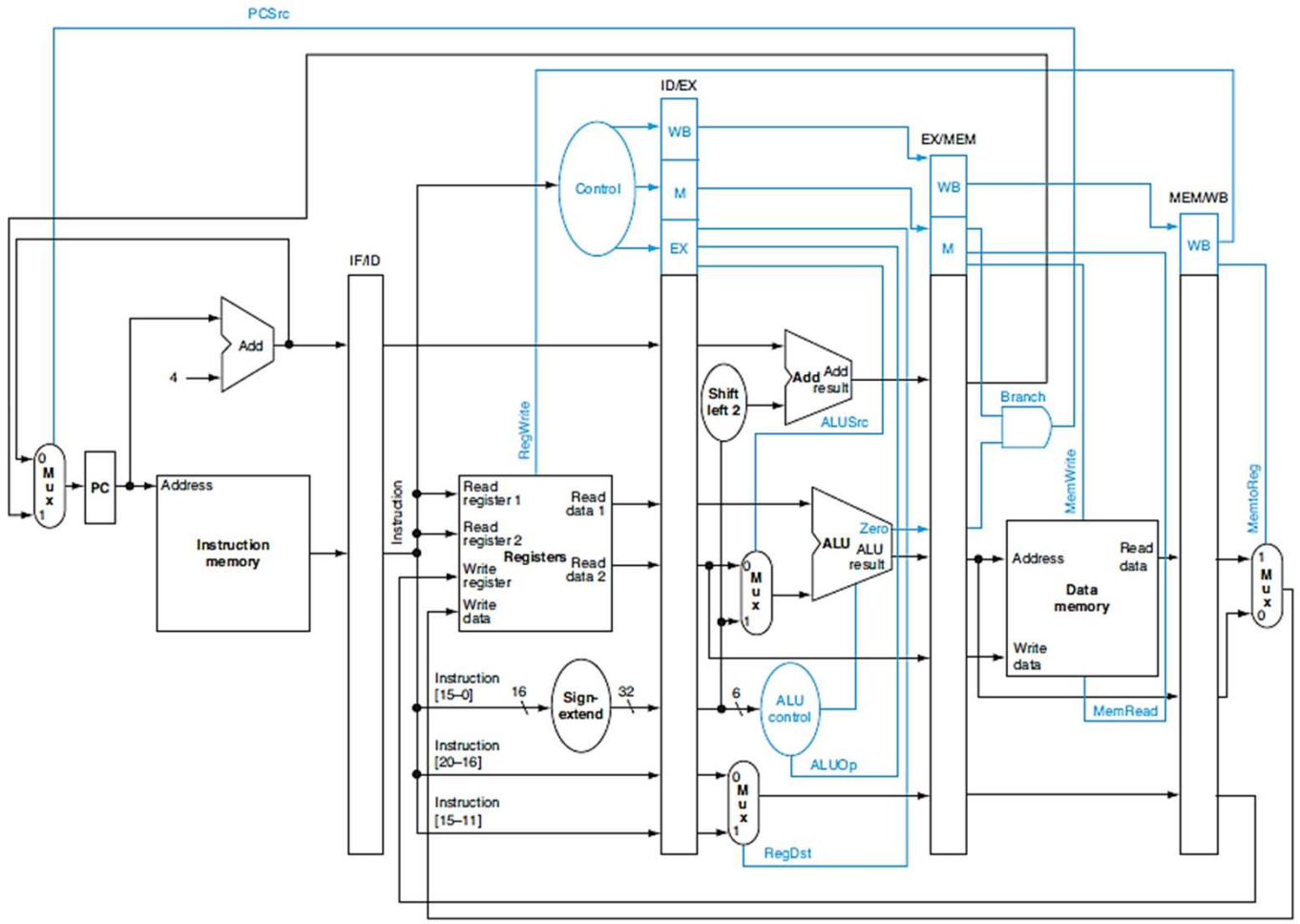
Pipelined Control

Control signals derived from instruction

As in single-cycle implementation



Pipeline Control



Data Hazards in ALU Instructions

Semnan University

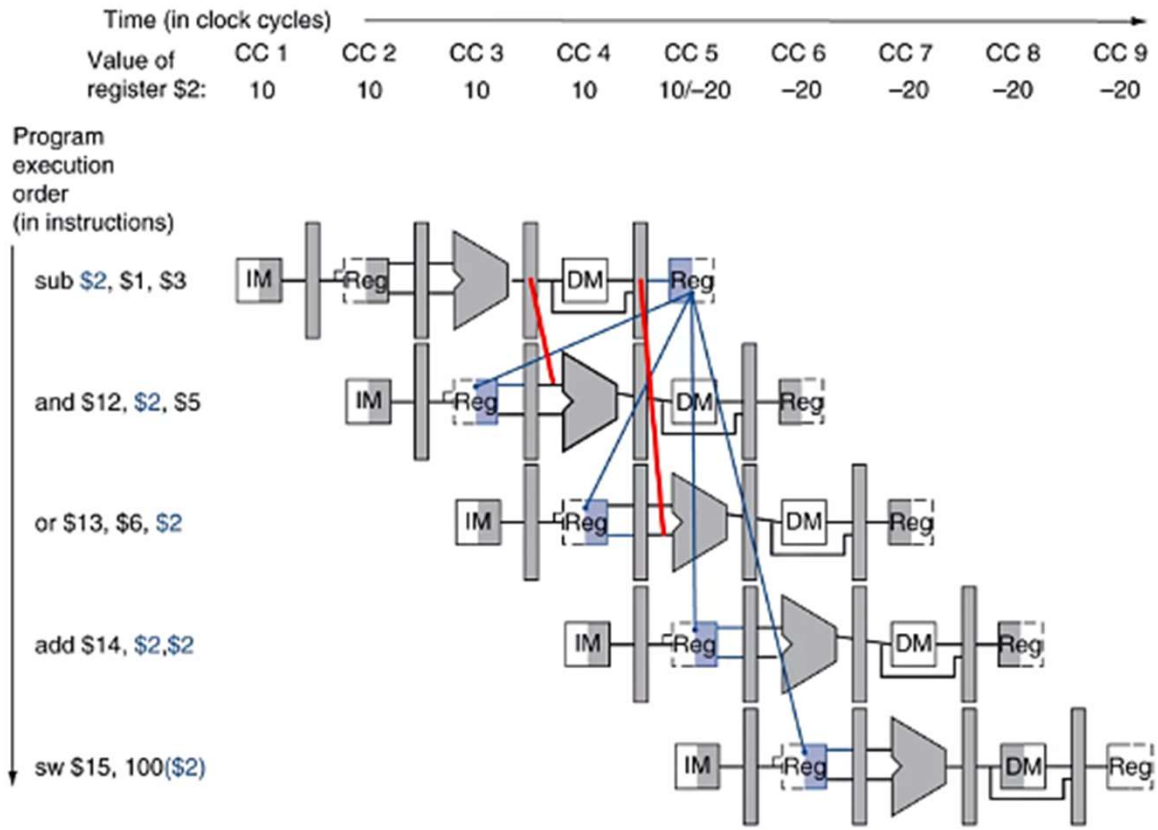
Consider this sequence:

```
sub   $2, $1,$3     # Register $2 written by sub
and   $12,$2,$5     # 1st operand($2) depends on sub
or    $13,$6,$2     # 2nd operand($2) depends on sub
add   $14,$2,$2     # 1st($2) & 2nd($2) depend on sub
sw    $15,100($2)   # Base ($2) depends on sub
```

We can resolve hazards with forwarding

- How do we detect when to forward?

Dependencies and Forwarding



Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when

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

Fwd from EX/MEM pipeline reg

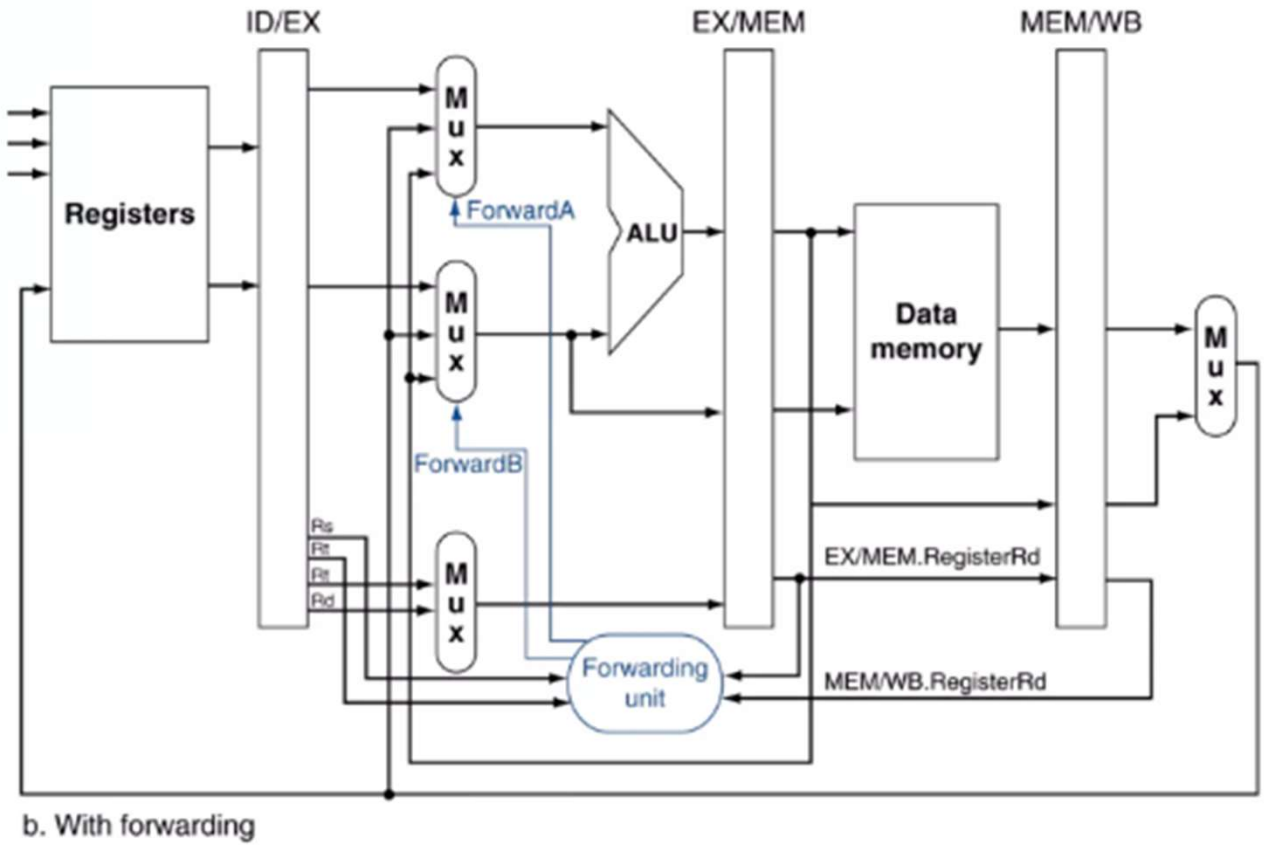
Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

Semnan University

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0,
  - MEM/WB.RegisterRd ≠ 0

Forwarding Path



b. With forwarding

Forwarding Conditions

Semnan University

EX hazard

- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 10

- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 10

MEM hazard

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 01
- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 01

Double Data Hazard

- Consider the sequence
- add $1, $1, $2
- add $1, $1, $3
- add $1, $1, $4
- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only forward if EX hazard condition isn't true

Revised Forwarding Condition

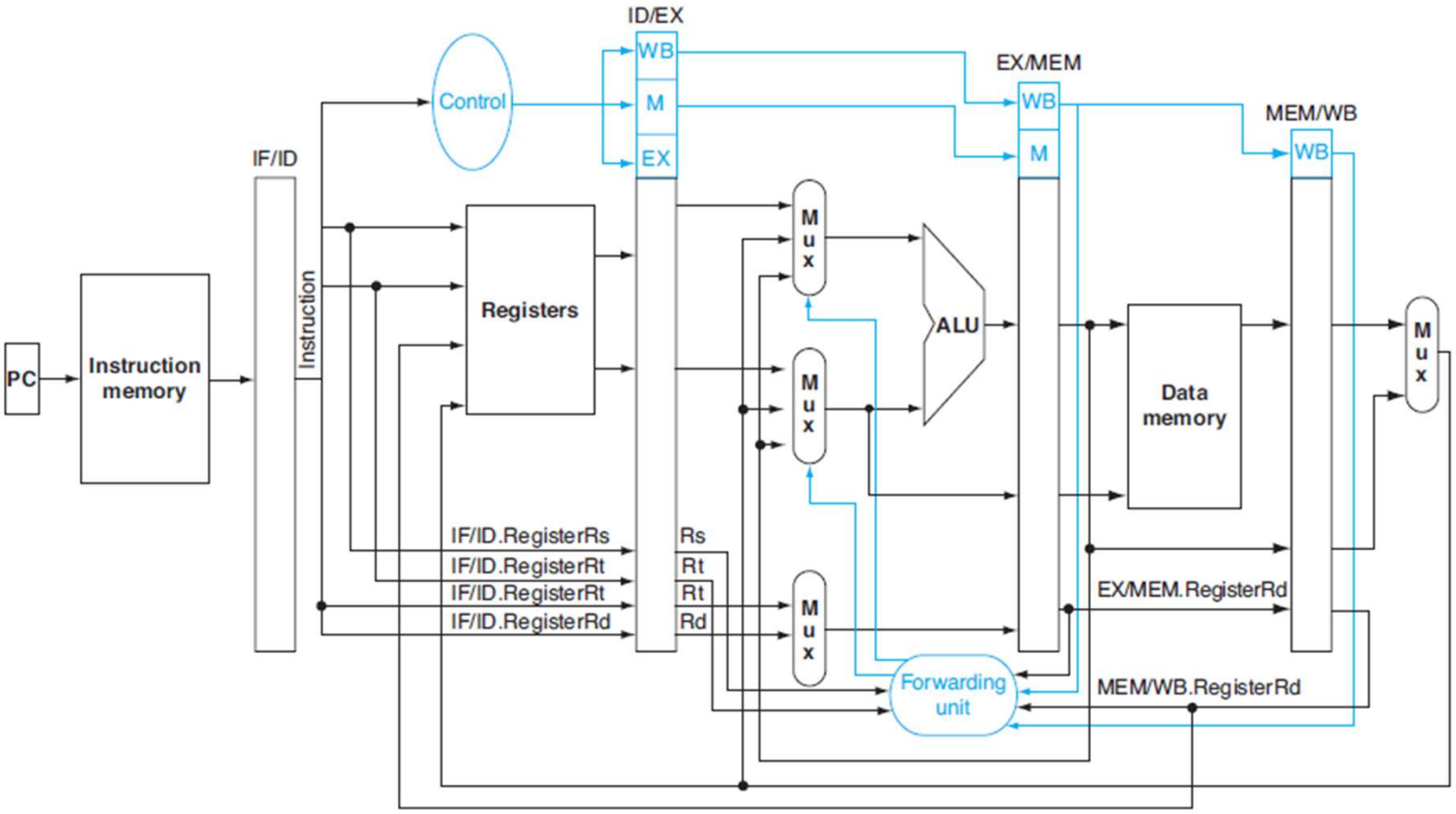
Semnan University

MEM hazard

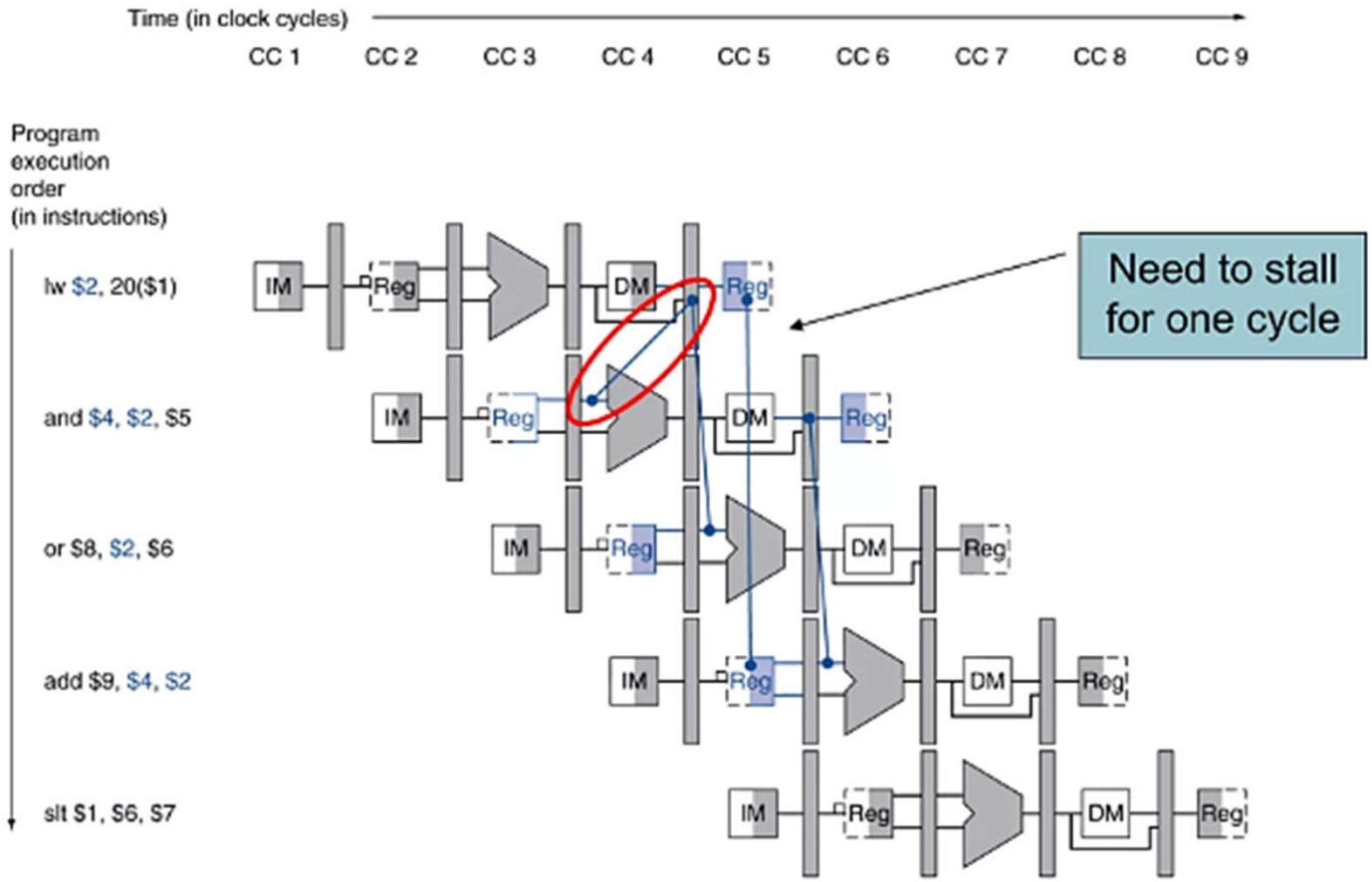
- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
  and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 01

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
  and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
  and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 01

Datapath with Forwarding



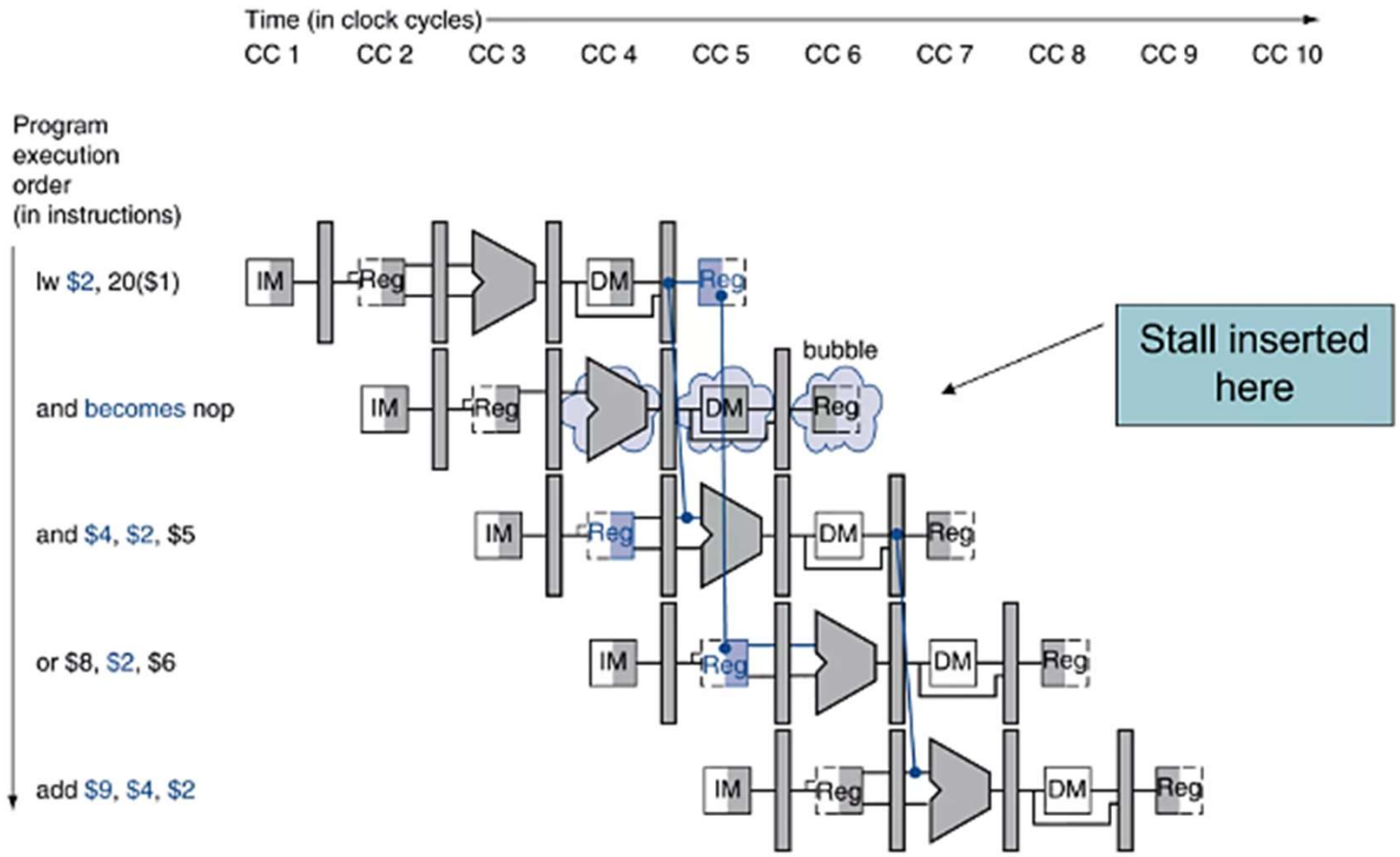
Load-Use Data hazard



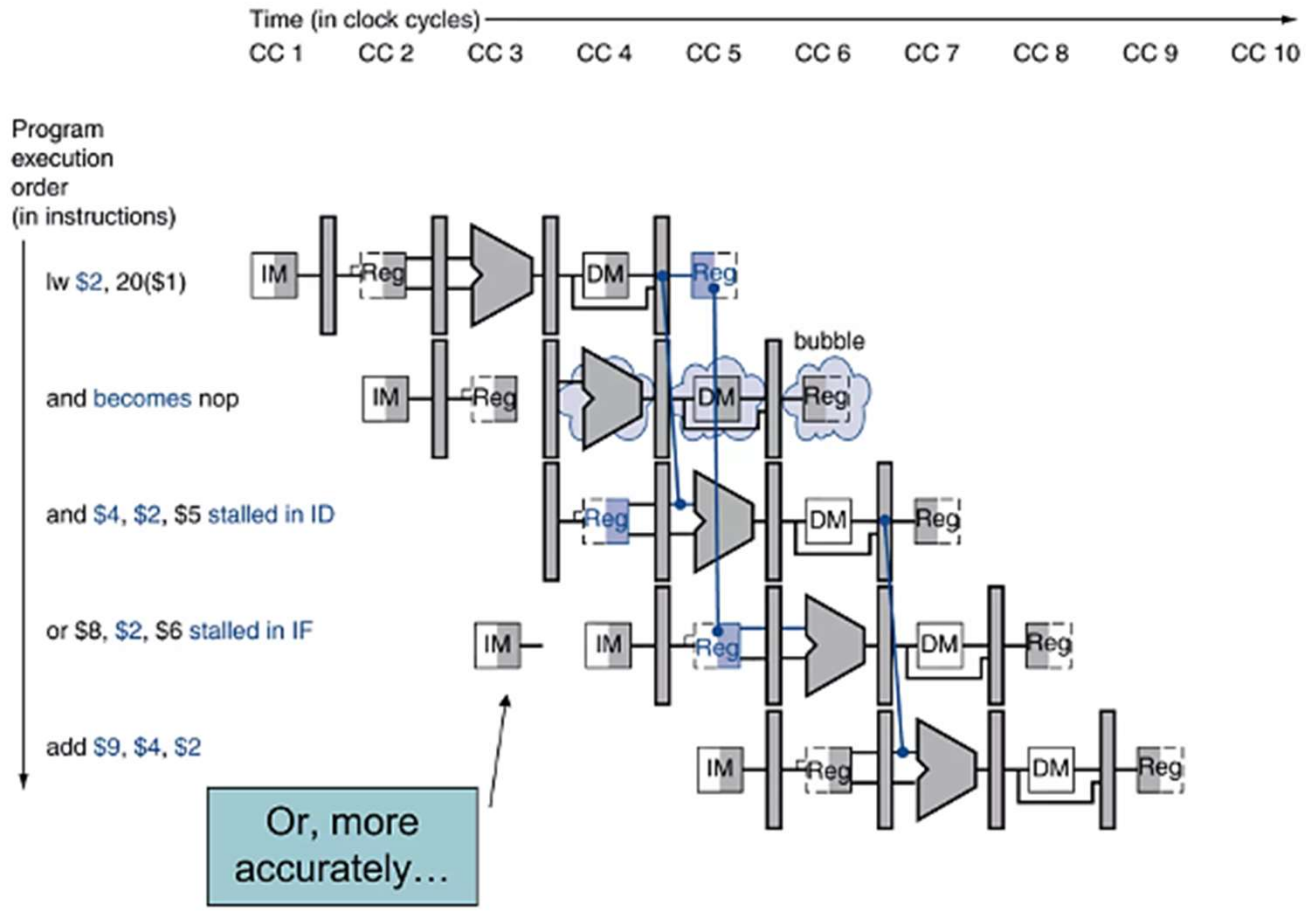
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do no p (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
- 1-cycle stall allows MEM to read data for 1w
  - Can subsequently forward to EX stage

Stall/Bubble in the Pipline



Stall/Bubble in the Pipline



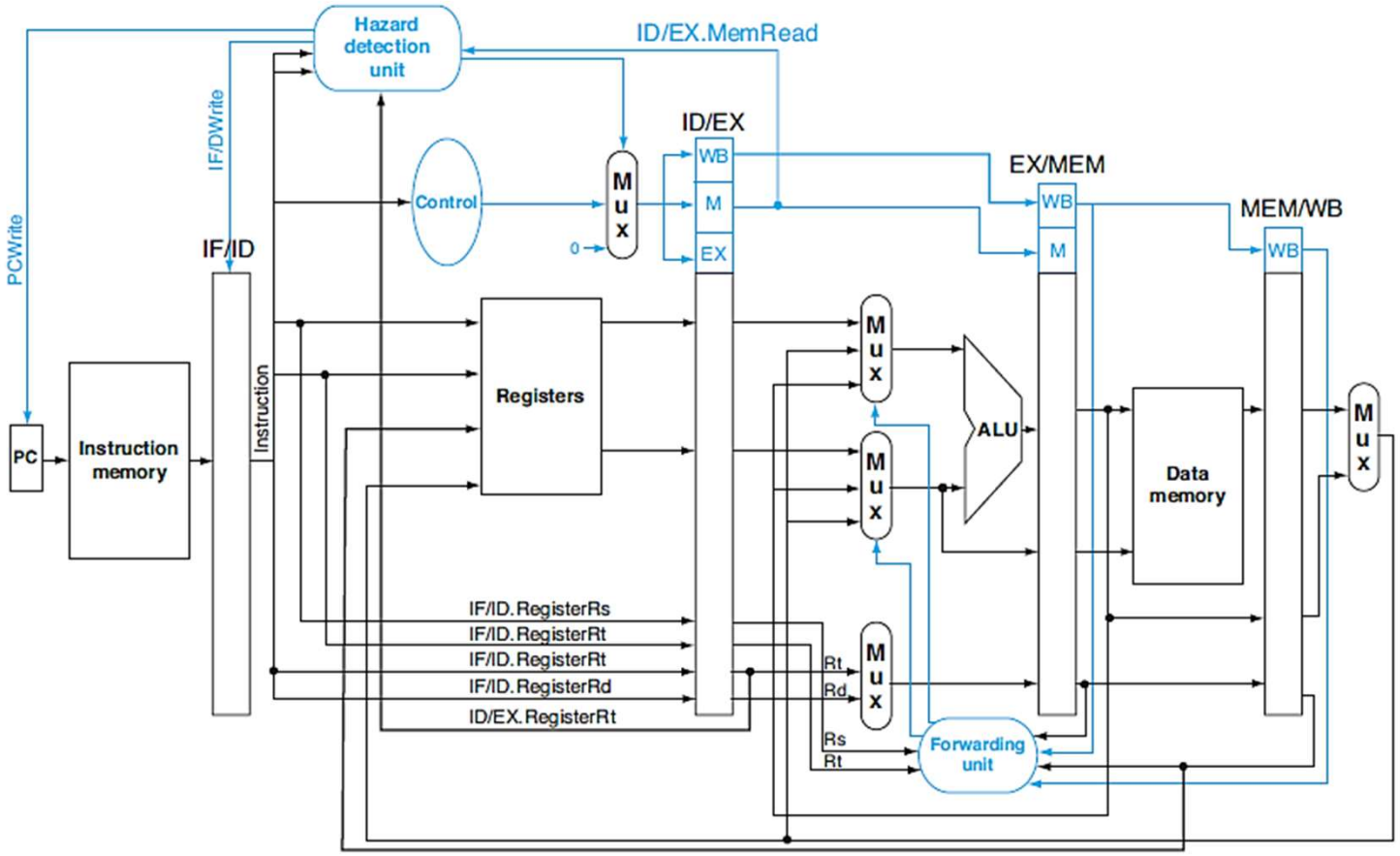
Load-Use Hazard Detection

Semnan University

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and

  ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble

Datapath with Hazard detection



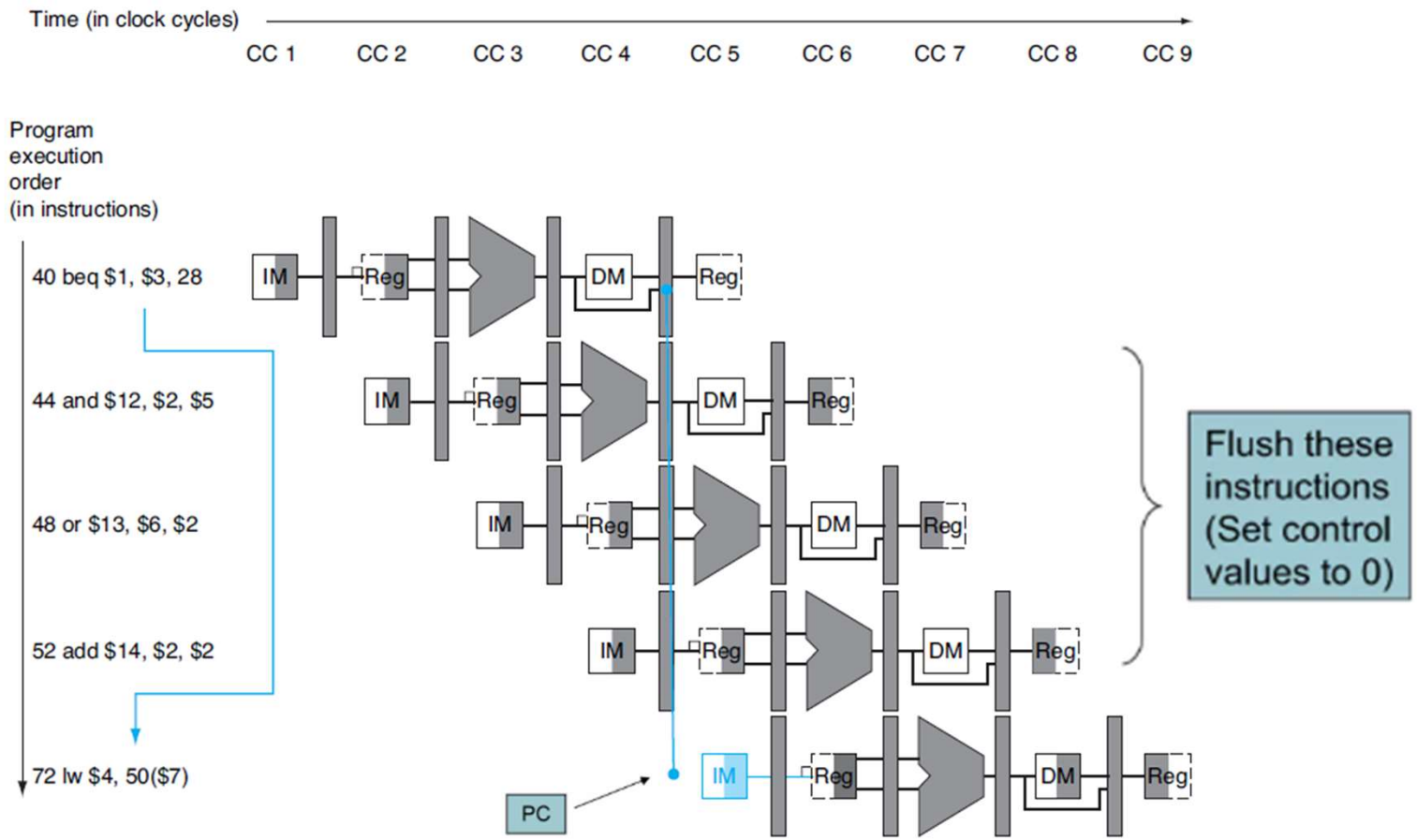
Stalls and Performance

- Stalls reduce performance
  - But are required to get correct results

- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Branch Hazards

- If branch outcome determined in MEM

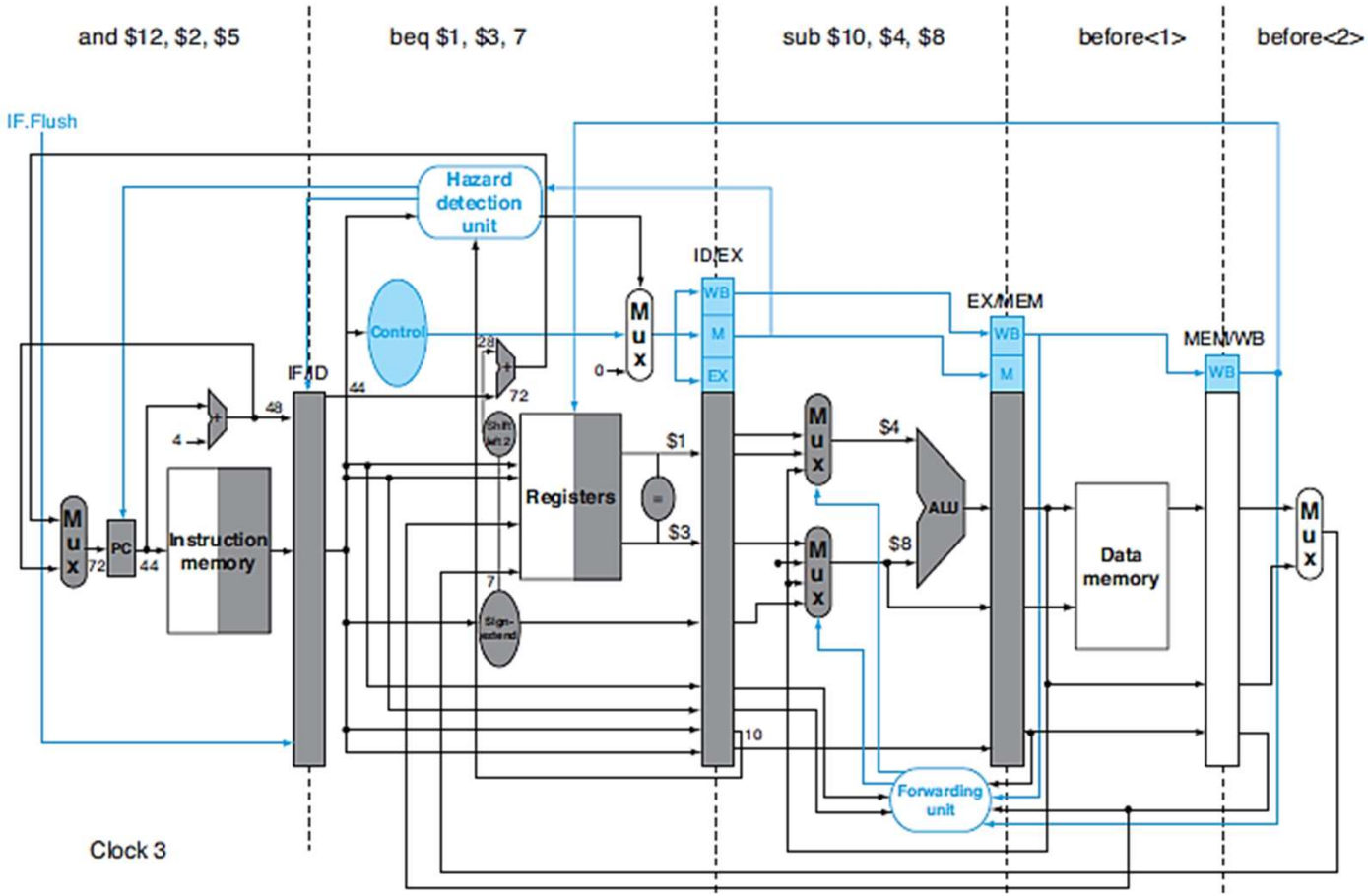


Reducing Branch Delay

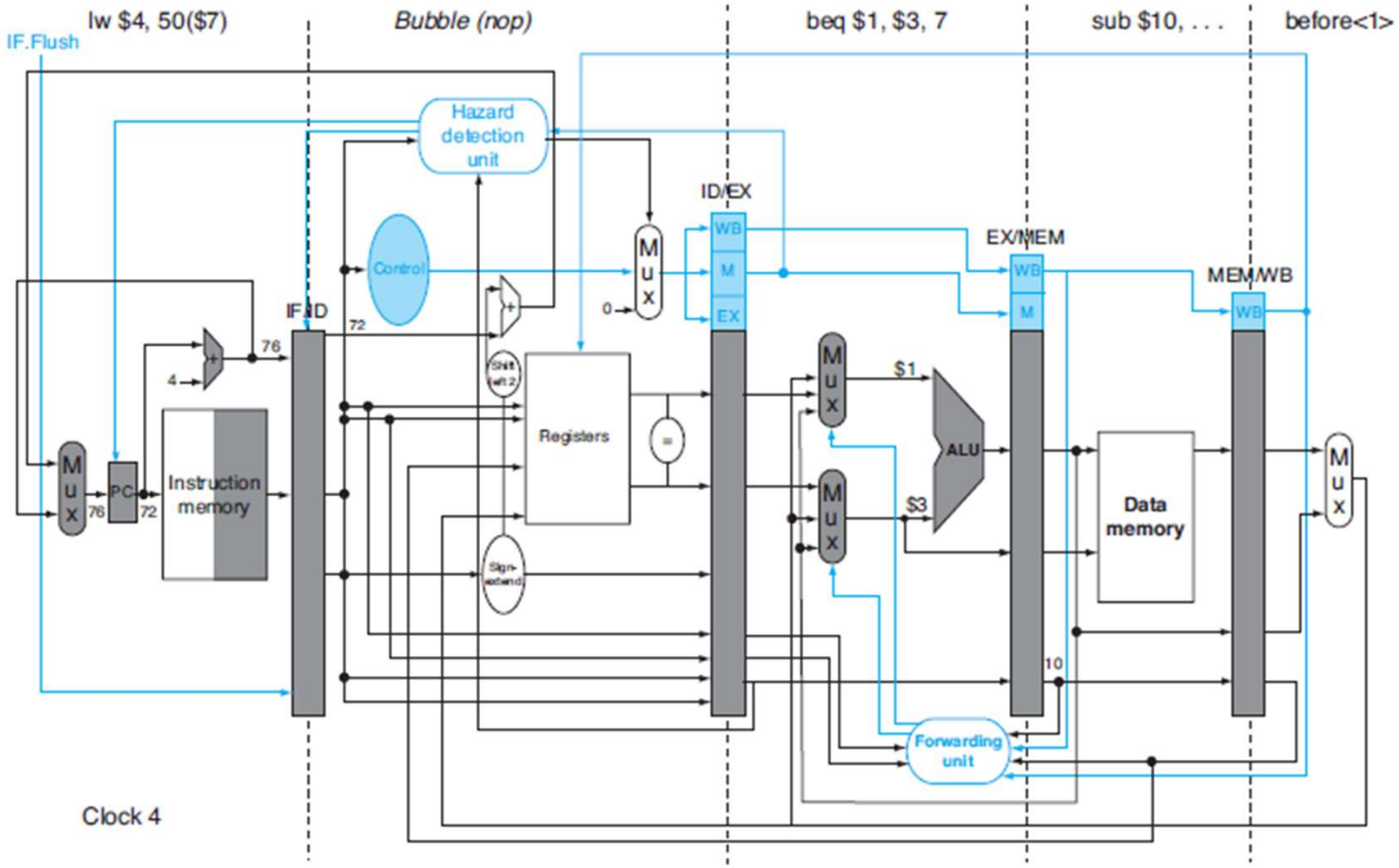
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
        .
36:   sub   $10, $4, $8
40:   beq   $1,  $3, 7
44:   and   $12, $2, $5
48:   or    $13, $2, $6
52:   add   $14, $4, $2
56:   slt   $15, $6, $7
        ...
72:   lw    $4, 50($7)
```

Example: Branch taken

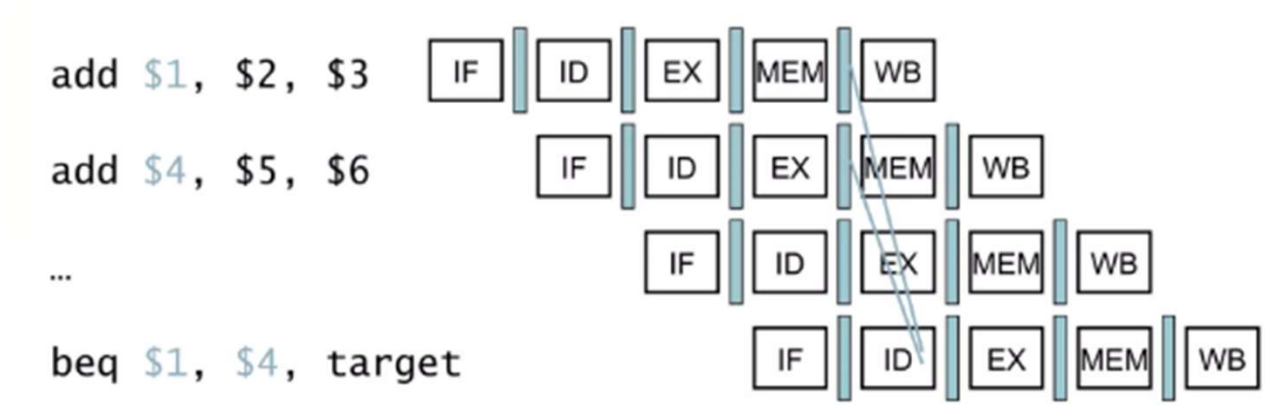


Example: Branch taken



Data Hazards for Branches

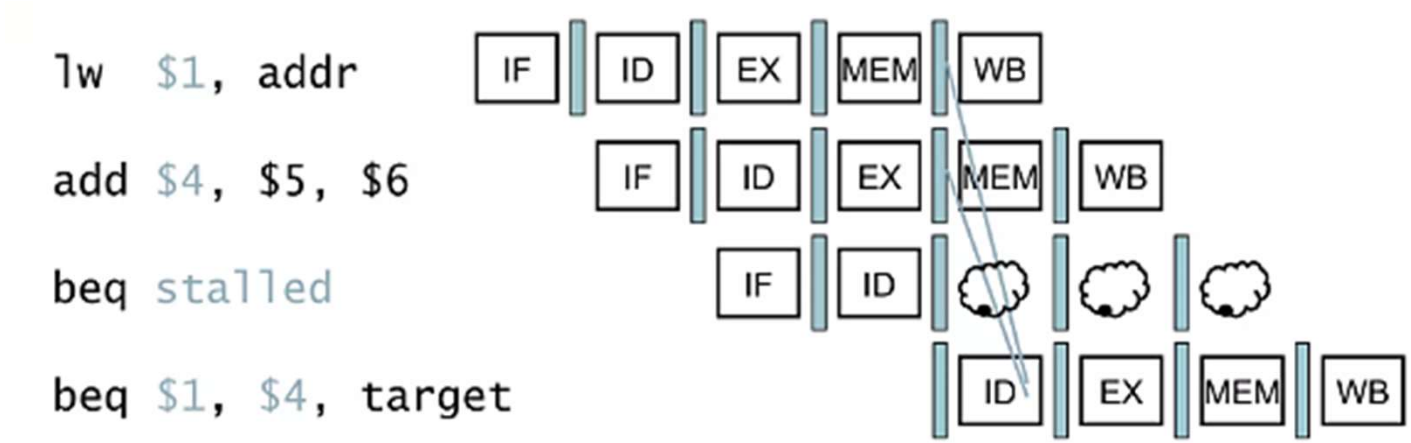
- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction



- Can resolve using Forwarding

Data Hazards for Branches

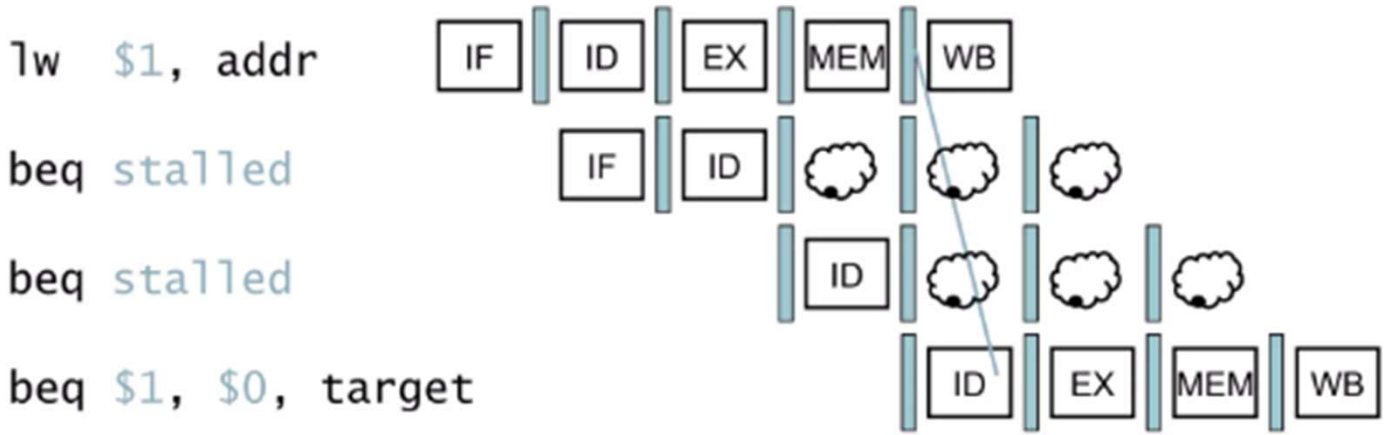
- If a comparison register is a destination preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle



Data Hazards for Branches

Semnan University

- If a comparison register is a destination immediately preceding load instruction
  - Need 2 stall cycles



Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards

- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Concluding Remarks

Semnan University

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall