CCSYA

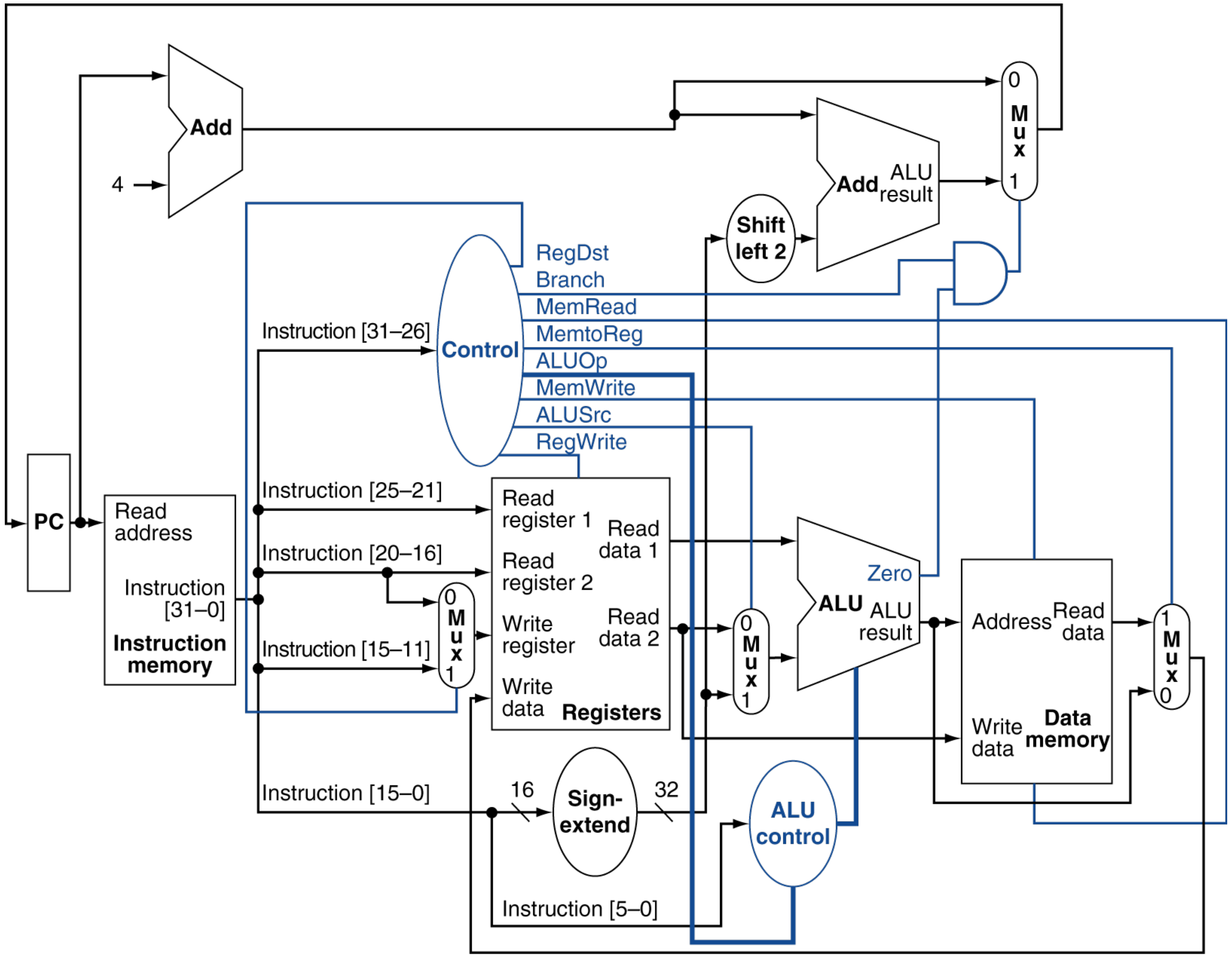
The Processor

Pipelining – Part I

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

Single-cycle Datapath

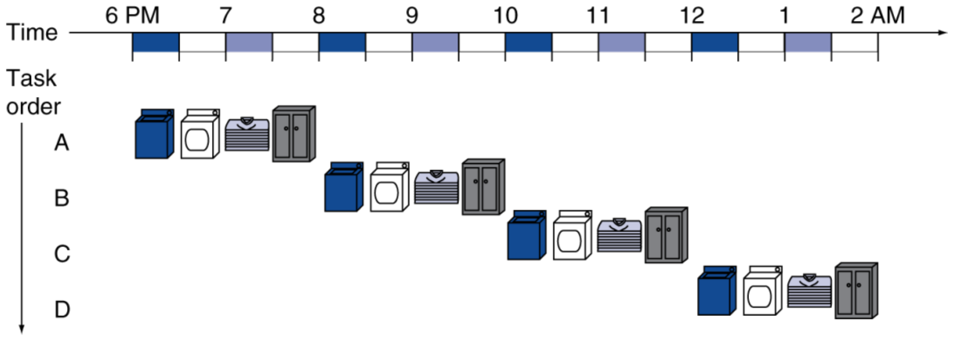


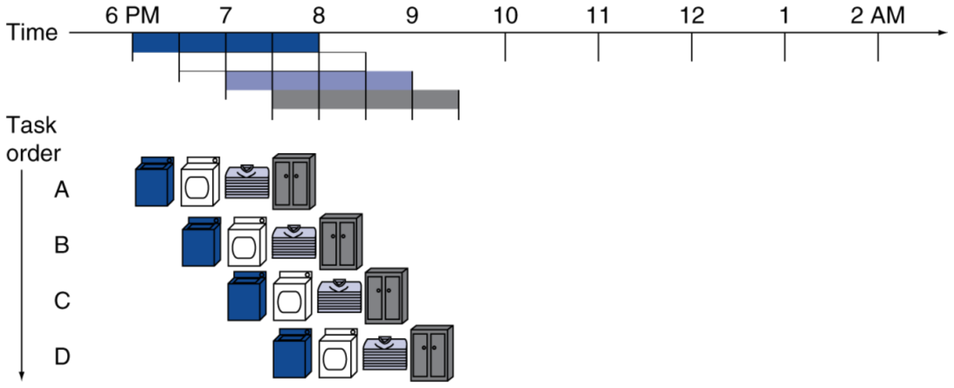
Pipelining

- **Implementation technique in which multiple instructions are overlapped in execution**
  - Instructions are executed in steps and each step takes one clock cycle
  - Different instructions may take a different number of steps to execute
  - The pipelined clock cycle is determined by the worst-case step
- **Pipelining improves performance by increasing instruction throughput**
  - As opposed to decreasing the average instruction execution time
- **Nearly universal today**

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

Pipelining

- **Exploits the potential parallelism among instructions**
  - This parallelism is often called **instruction-level parallelism** (ILP)

- **There are two methods for increasing the potential amount of ILP:**
  - **Increase the number of steps** (depth) of the pipeline to overlap more instructions (and potentially reduce the clock cycle)
  - **Replicate the functional units** of the computer so that it can launch multiple instructions in every pipeline stage

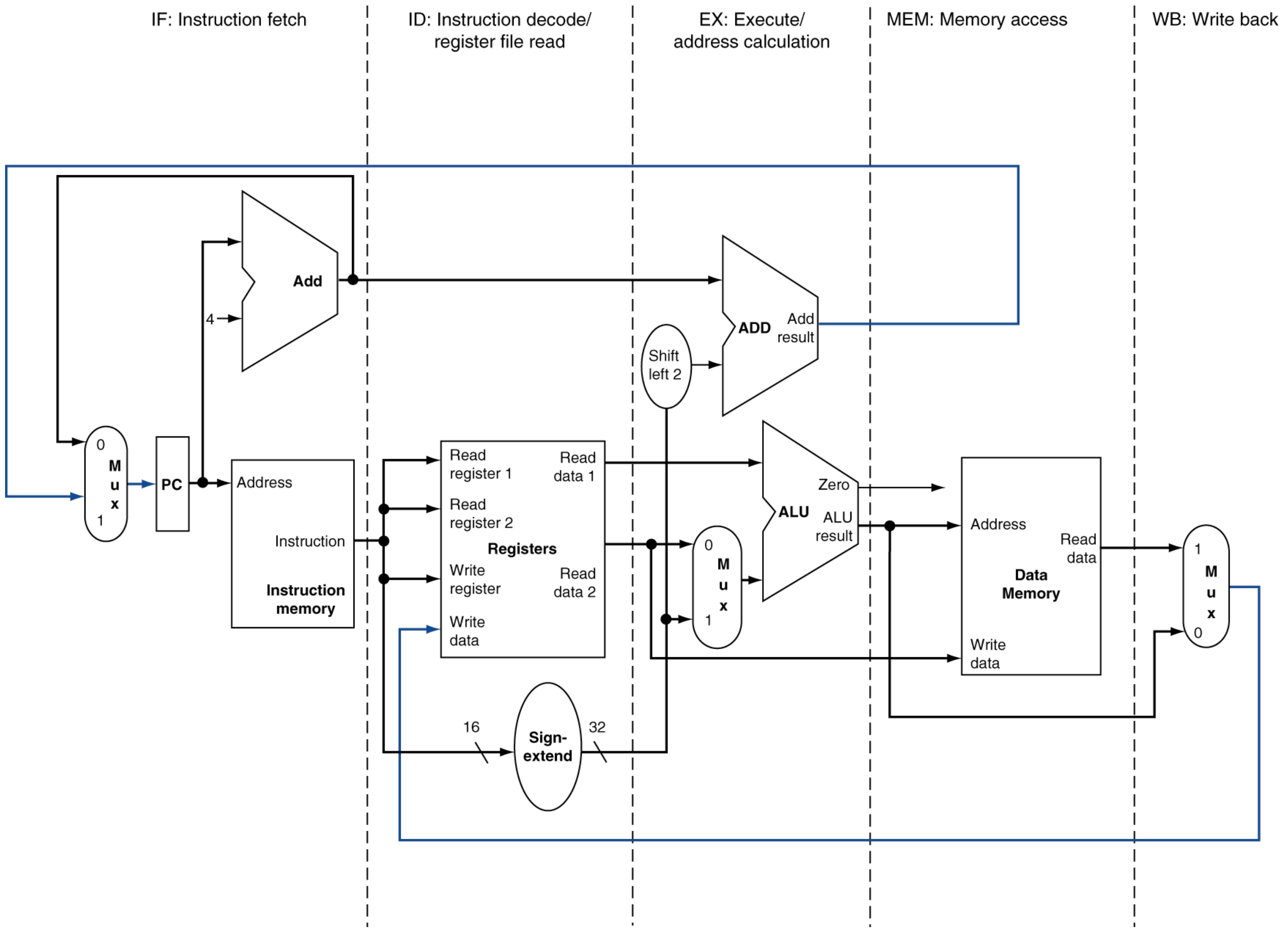
- **In what follows, we will not consider the case of using several functional units per stage**

MIPS Pipeline

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

- **These five steps correspond roughly to the way the data path is drawn**
  - Instructions and data move generally from left to right through the five stages as they complete execution

MIPS Pipelined Datapath



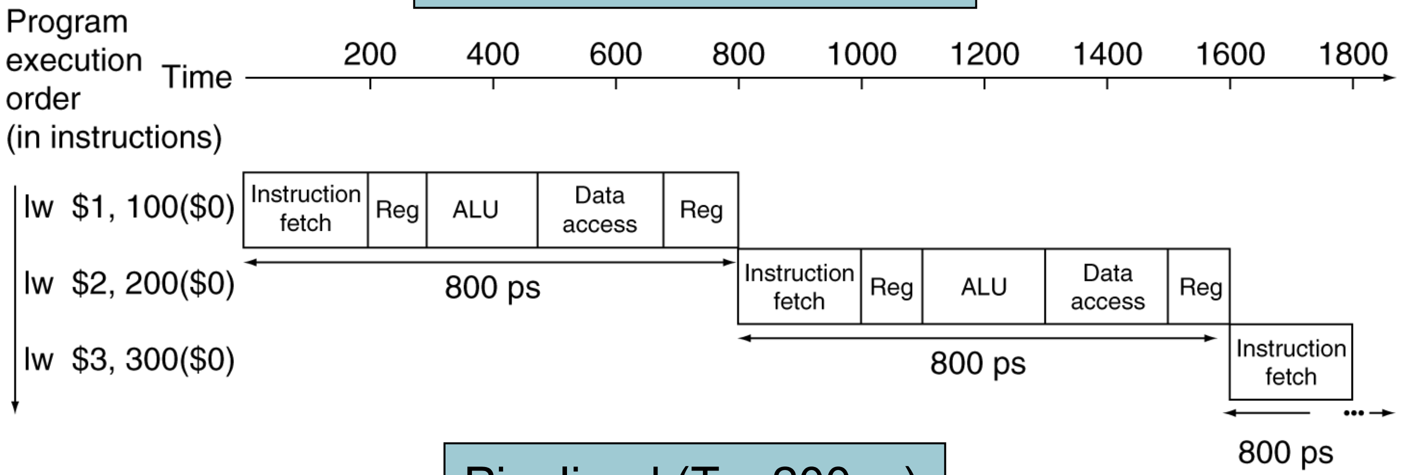
Pipeline Performance

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

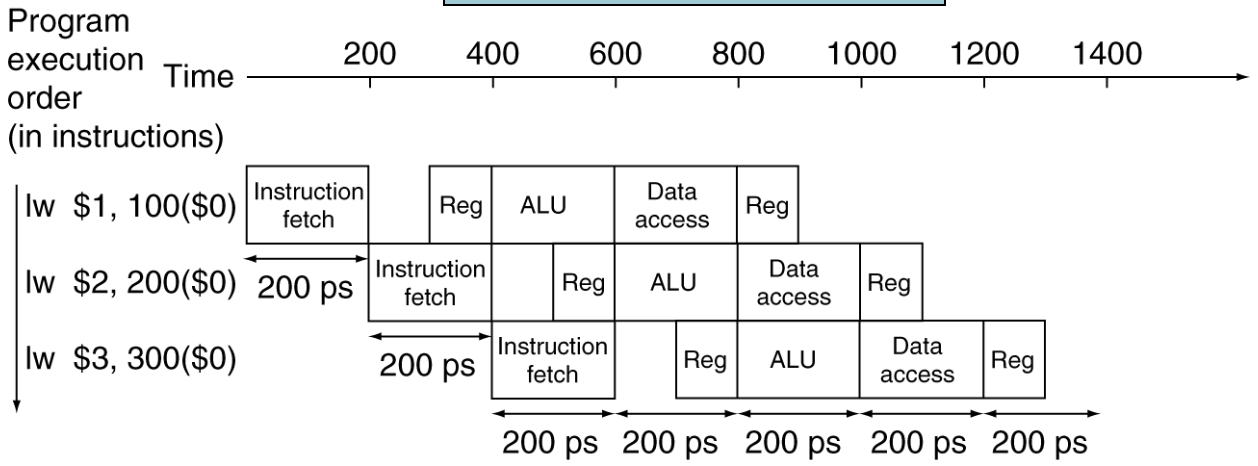
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100ps	200ps	200ps	100ps	800ps
sw	200ps	100ps	200ps	200ps		700ps
R-format	200ps	100ps	200ps		100ps	600ps
beq	200ps	100ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Pipeline Speedup

- **If all stages are balanced (ideal condition)**
  - i.e., all take the same time

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

- **If not balanced, speedup is less**
  - Moreover, pipelining involves some overhead, the source of which will be clearer shortly

- **Speedup due to increased throughput**
  - Latency (time for each instruction) does not decrease
  - But instruction throughput is the important metric because real programs execute billions of instructions

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

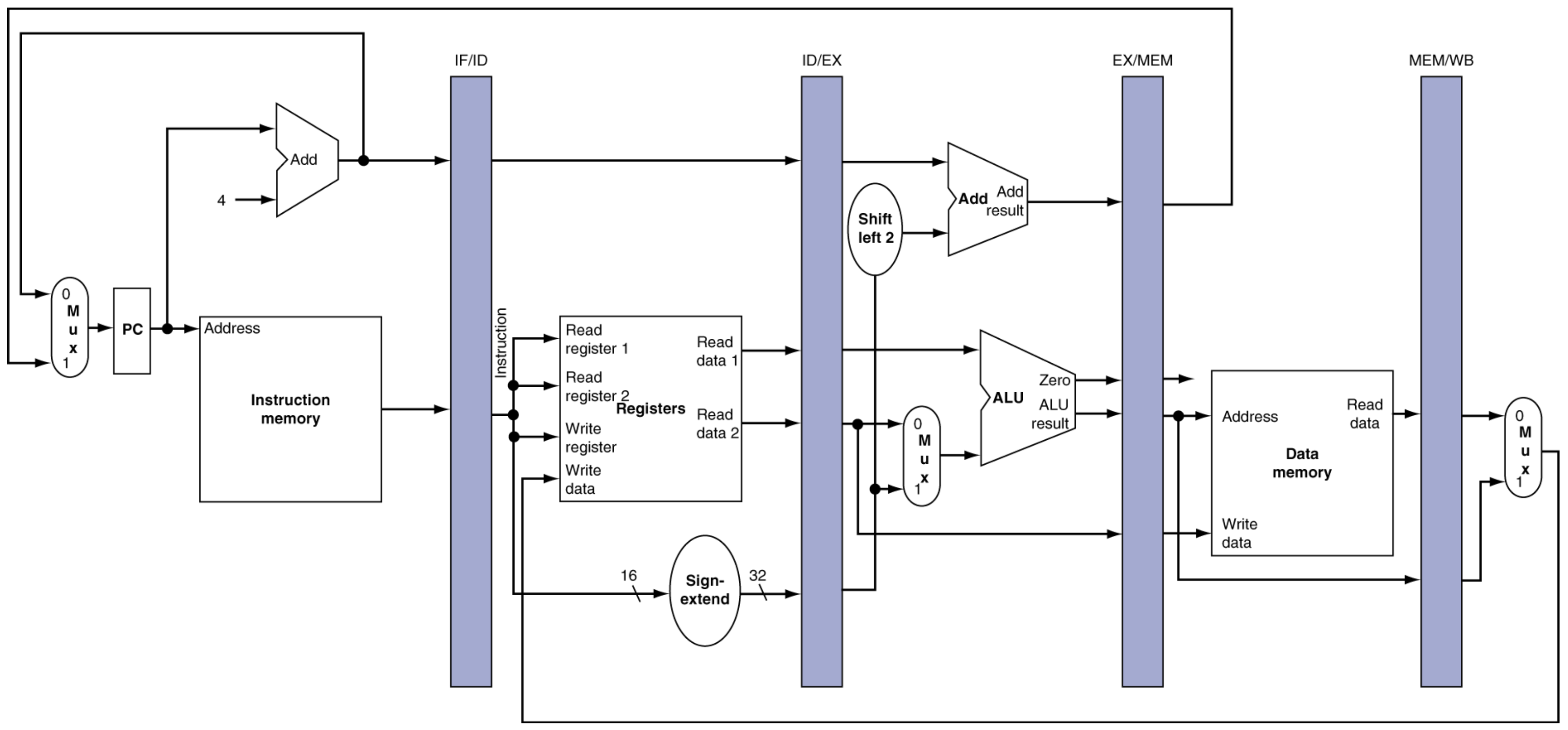
Implementing a Pipelined Datapath

- **All instructions advance during each clock cycle from one pipeline stage to the next**

- **Therefore, we need pipeline registers to hold data**
  - So that portions of a single datapath can be shared during instruction execution

- **The registers are named for the two stages separated by that register**
  - IF/ID, ID/EX, EX/MEM, MEM/WB
  - Notice that there is no pipeline register at the end of the write-back stage

Pipeline registers

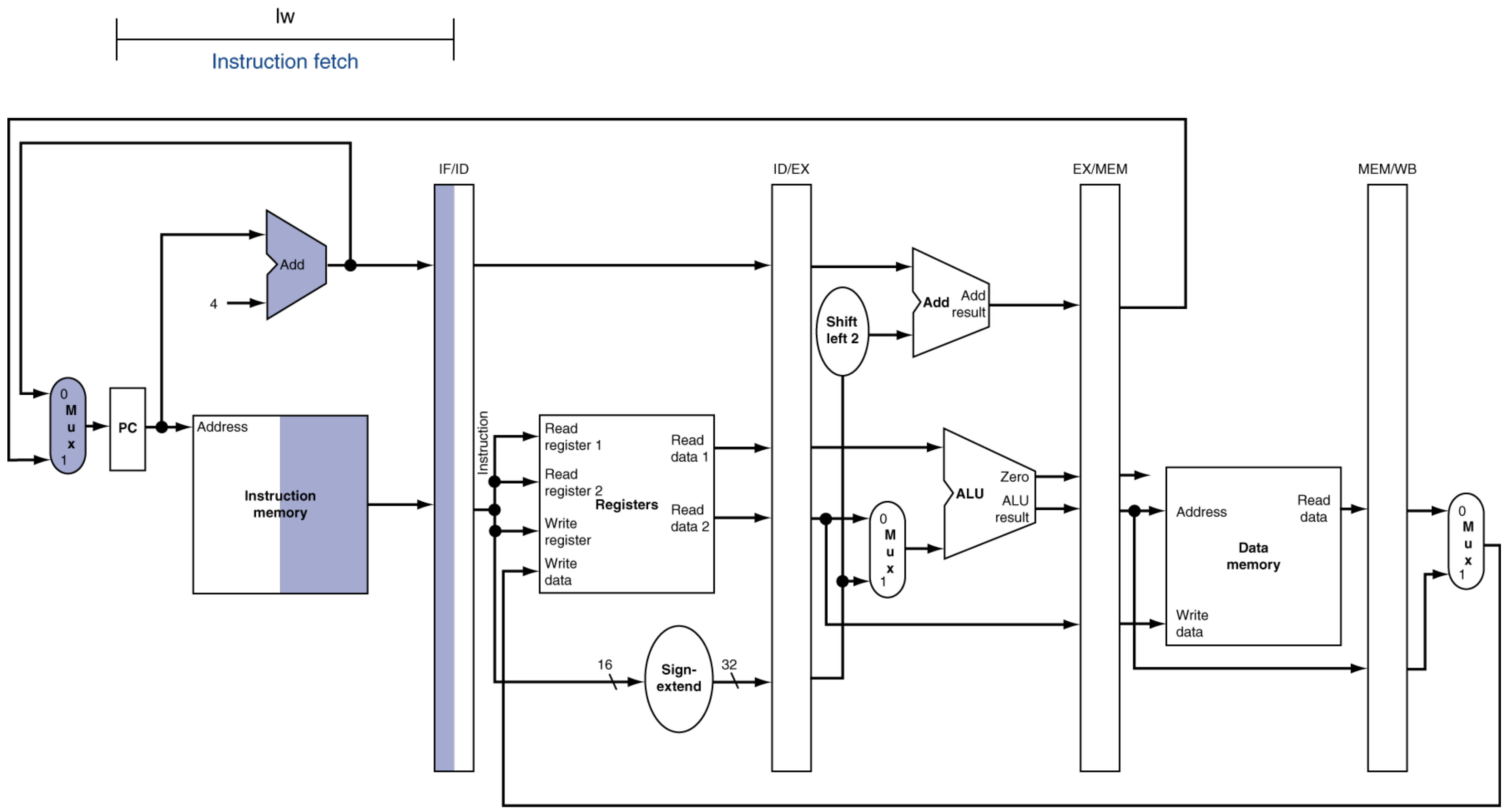


Pipeline Operation

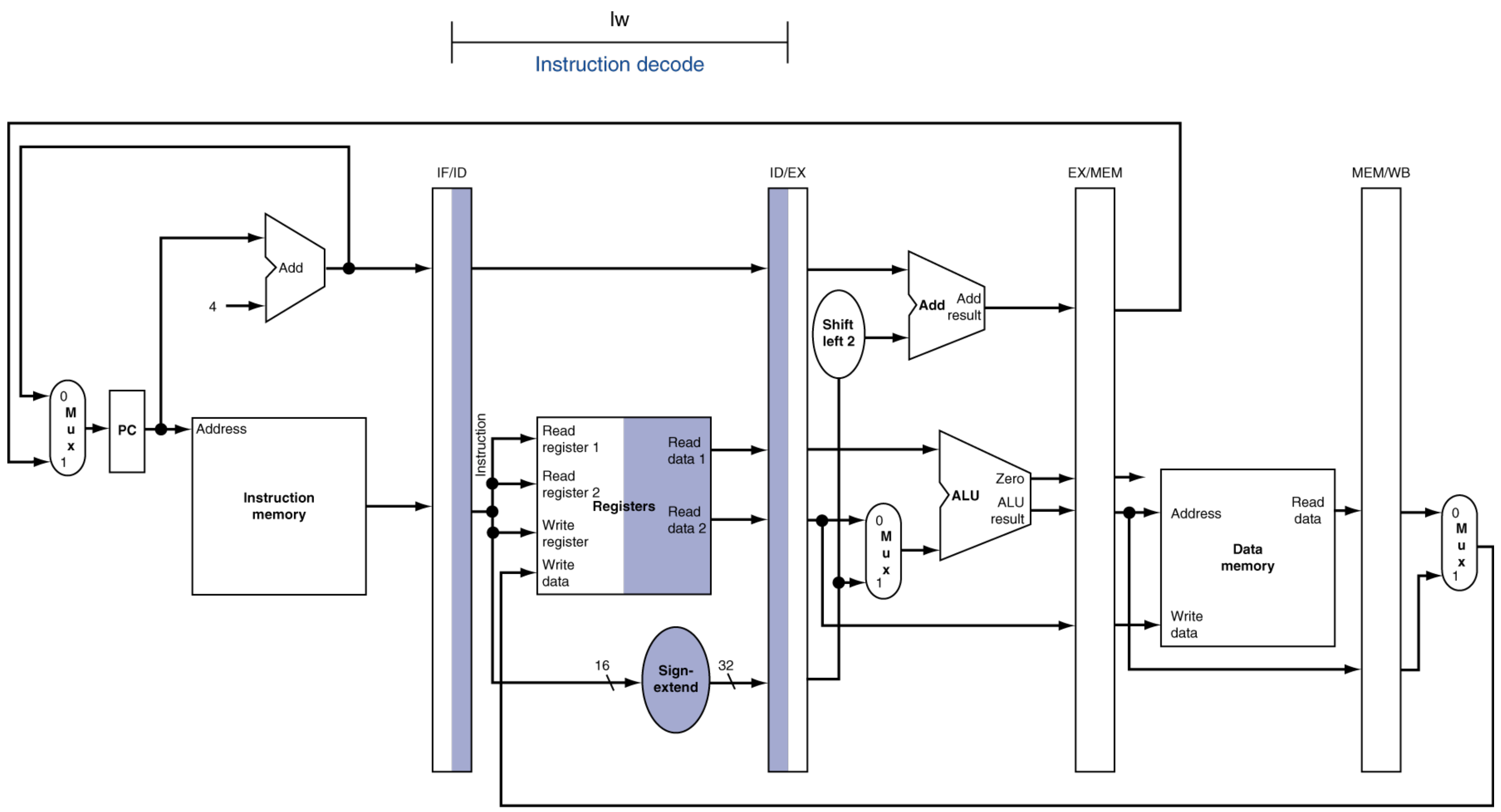
- **“Single-clock-cycle” pipeline diagram**
  - Shows pipeline usage in a single cycle
  - Highlight the **right half** of registers or memory when they are being **read** and highlight the **left half** when they are being **written**

- **We’ll look at “single-clock-cycle” diagrams for load and store**
  - This walk-through of the load instruction shows that any information needed in a later pipe stage must be passed to that stage via a pipeline register
  - Walking through a store instruction shows the similarity of instruction execution, as well as passing the information for later stages

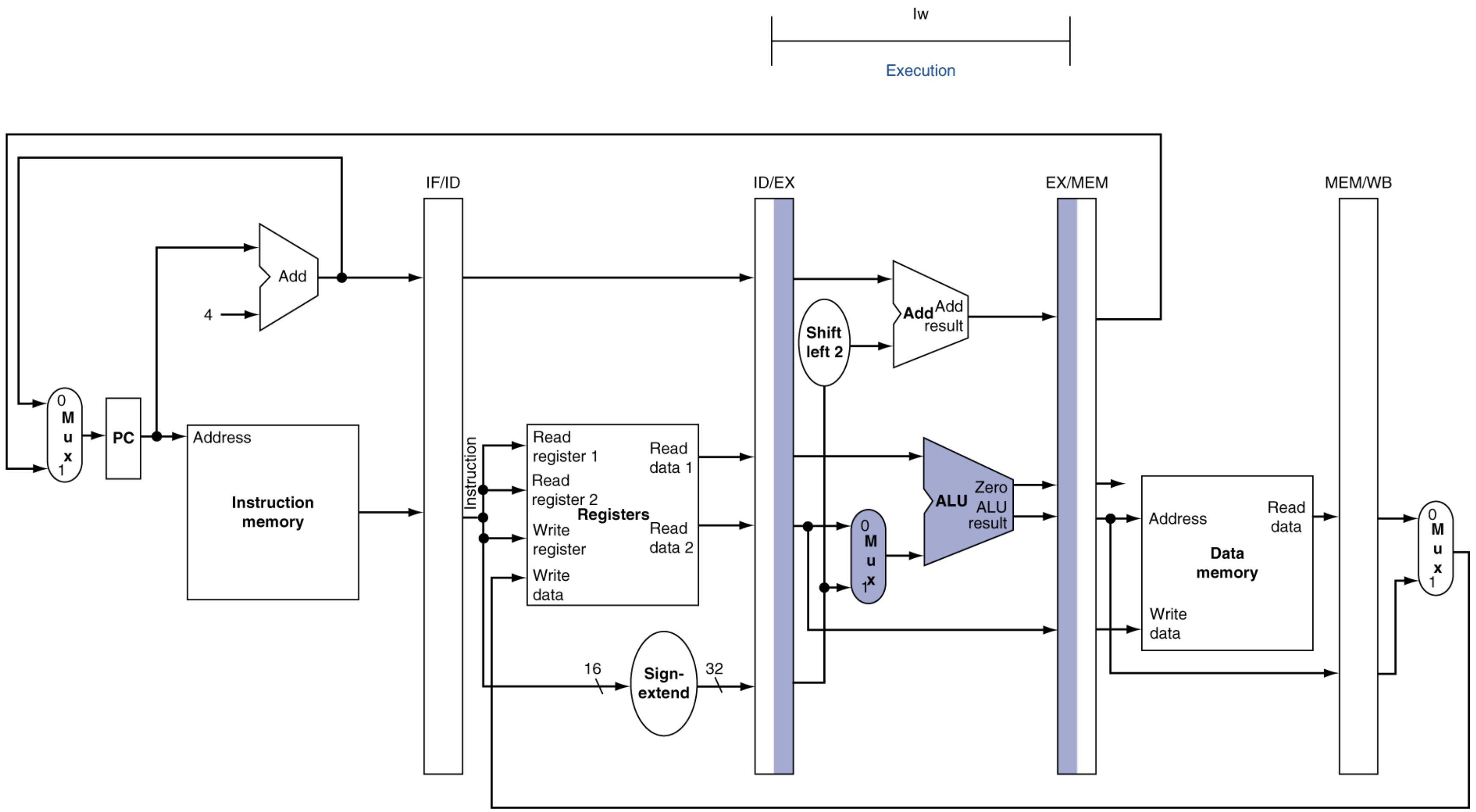
IF for Load, Store, …



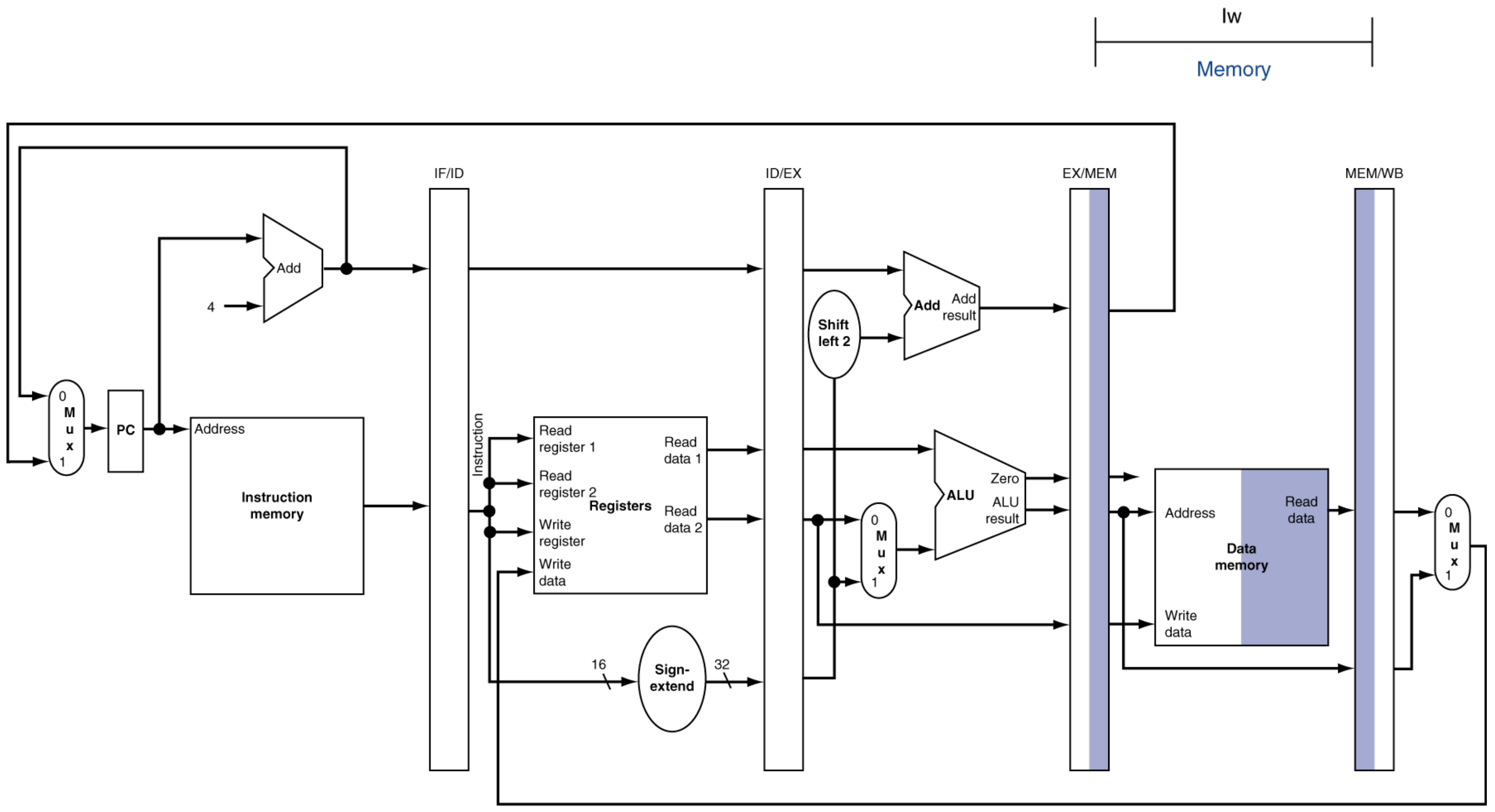
ID for Load, Store, …



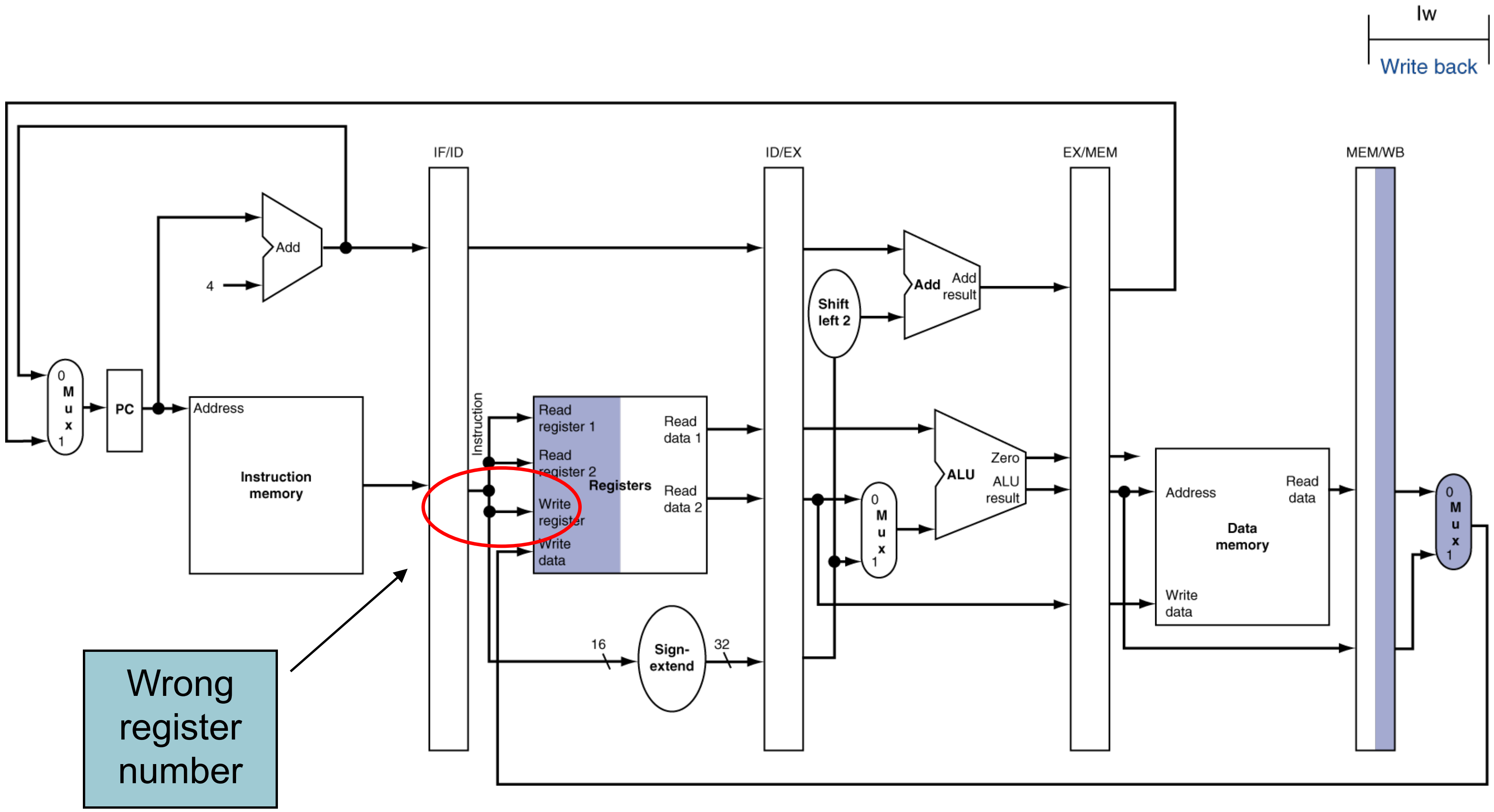
EX for Load


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0 M u x 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
0 M u x 1
Address
Data memory
Write data
Read data
0 M u x 1

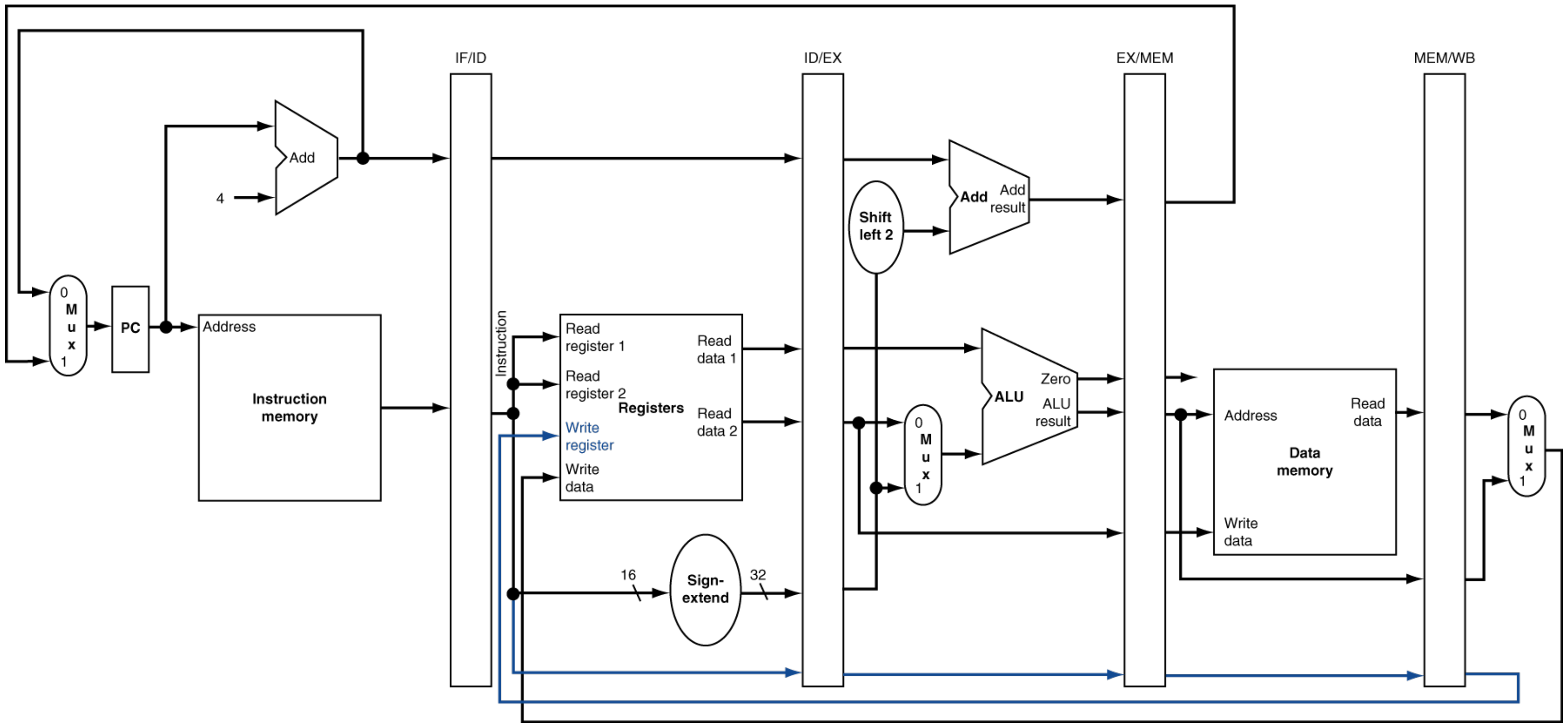
MEM for Load


lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
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
Write data
Read data

WB for Load

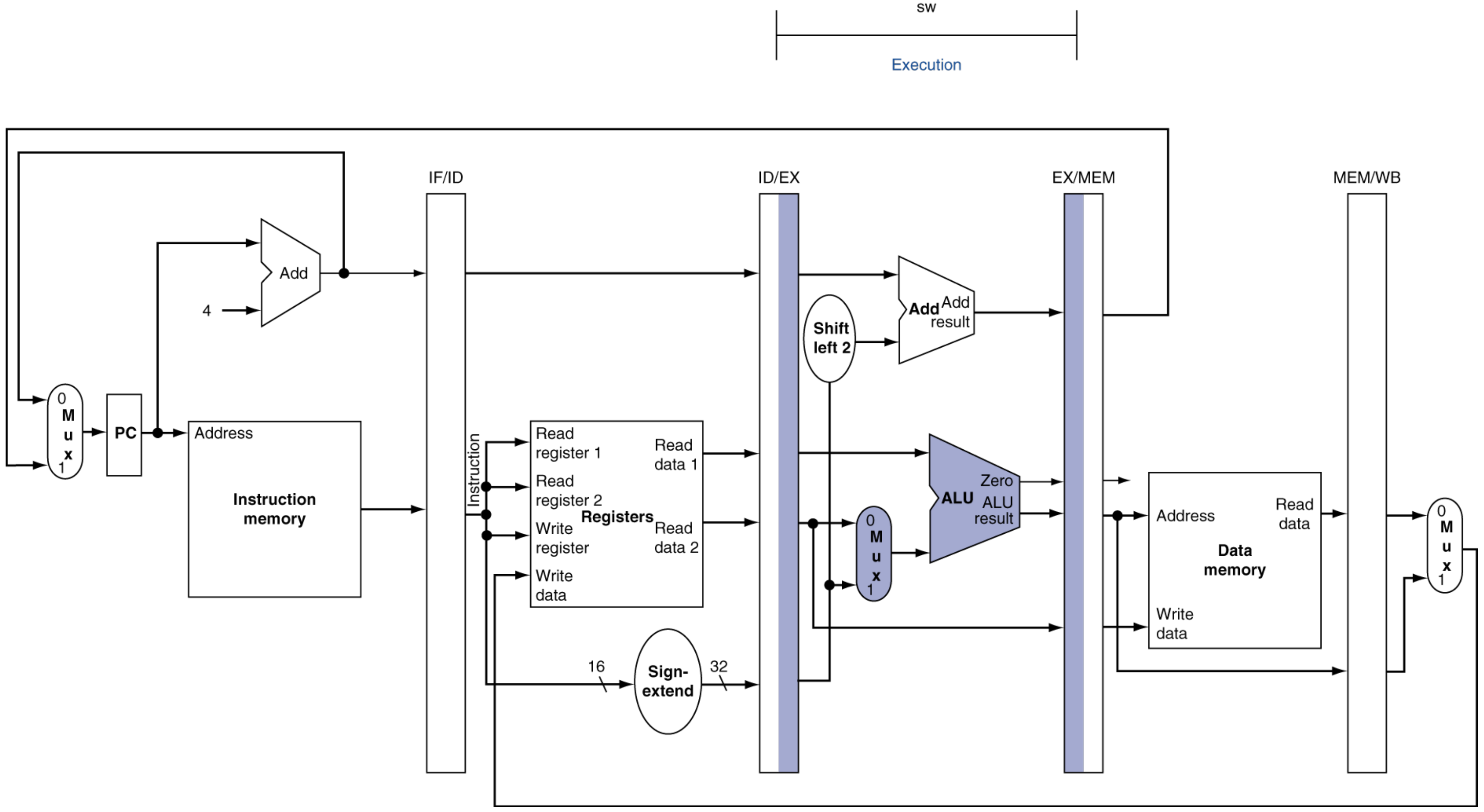

lw
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
Add
Add result
Shift left 2
0 M u x 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
0 M u x 1
ALU
Zero
ALU result
Address
Read data
Data memory
Write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

Corrected Datapath for Load

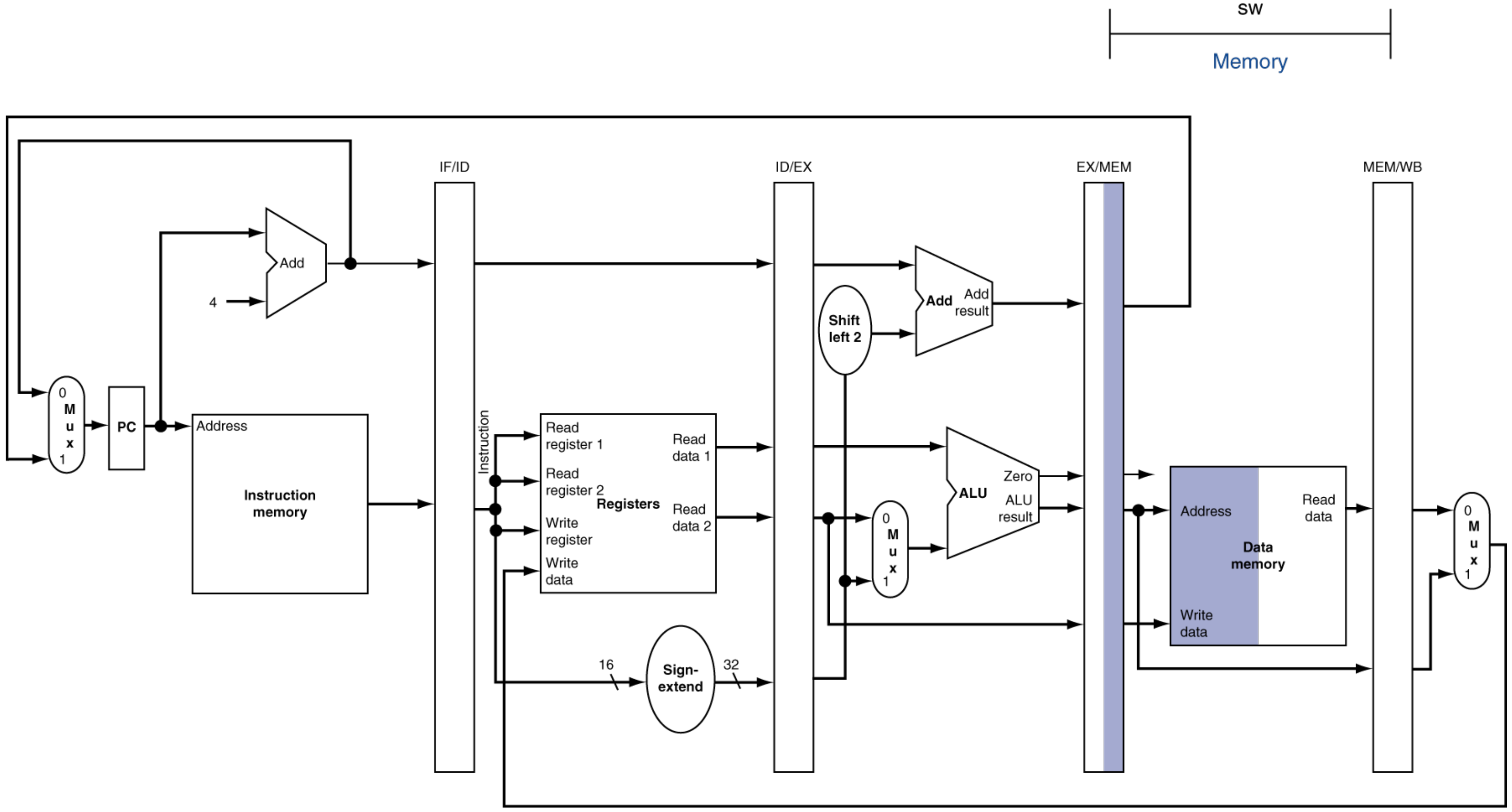


- **Need to preserve the destination register number**
  - Pass the register number from the ID/EX through EX/MEM to the MEM/WB pipeline register for use in the WB stage

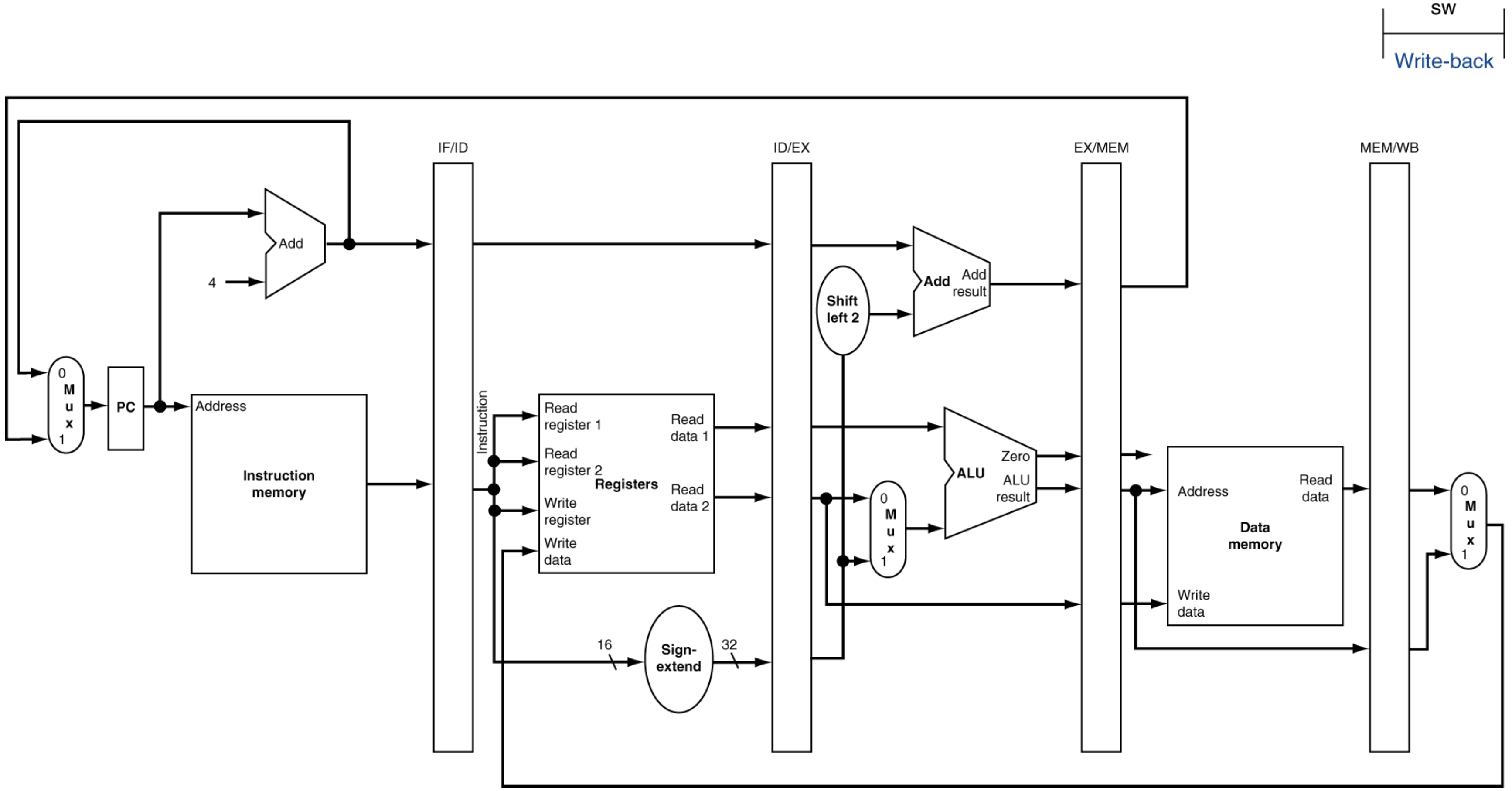
EX for Store


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

MEM for Store



WB for Store



SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
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
0
M
u
x
1
Address
Read data
Data memory
Write data
0
M
u
x
1

Multiple instructions

- **Consider the following five-instruction sequence:**

```
lw  $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw  $13, 24($1)
add $14, $5, $6
```

- **Multi-clock-cycle pipeline diagram**
  - Time advances from left to right
  - Program execution advances from the top to the bottom

Multi-Cycle Pipeline Diagram

- **Traditional form**

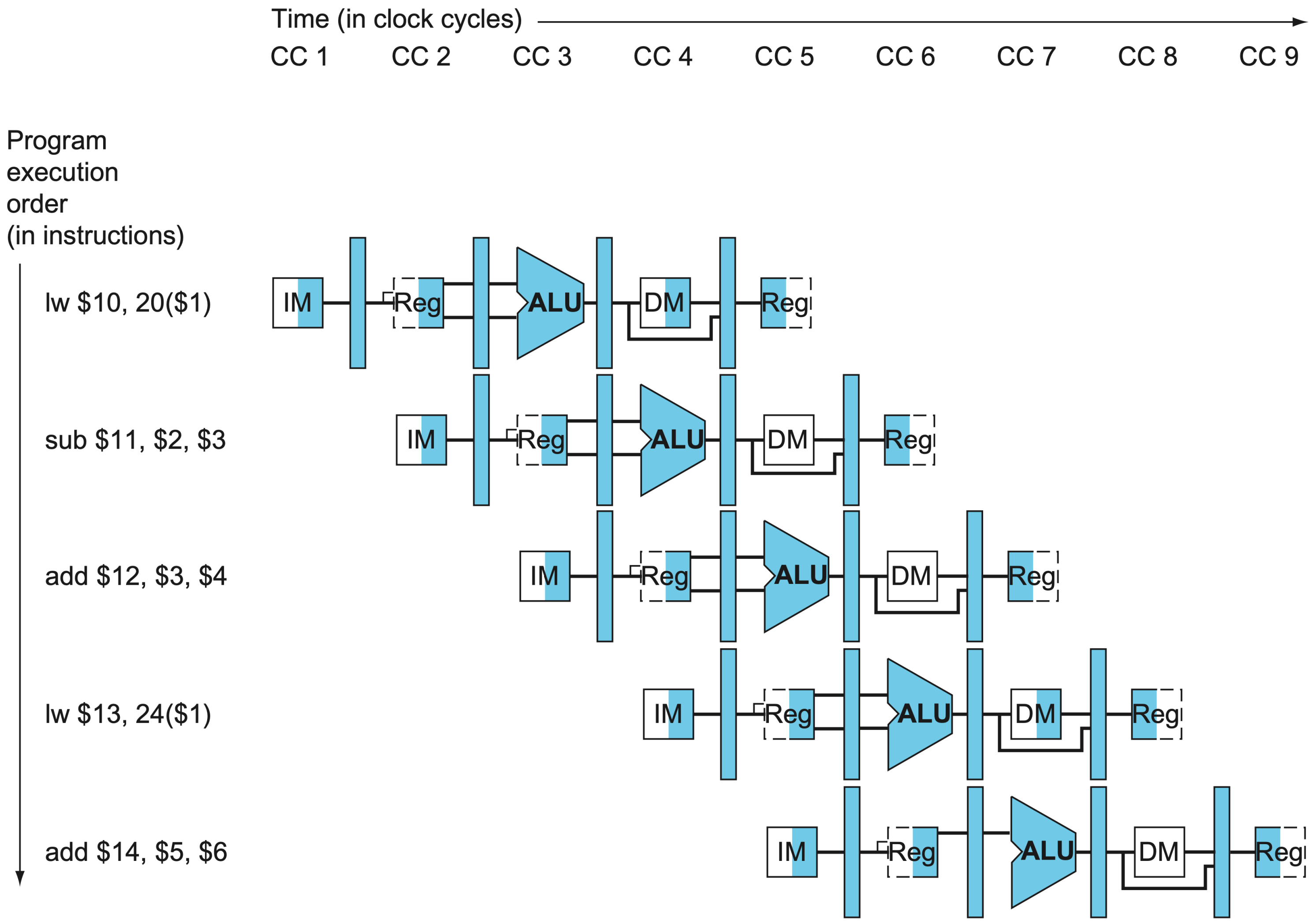
Time (in clock cycles) →

Program execution order (in instructions) ↓

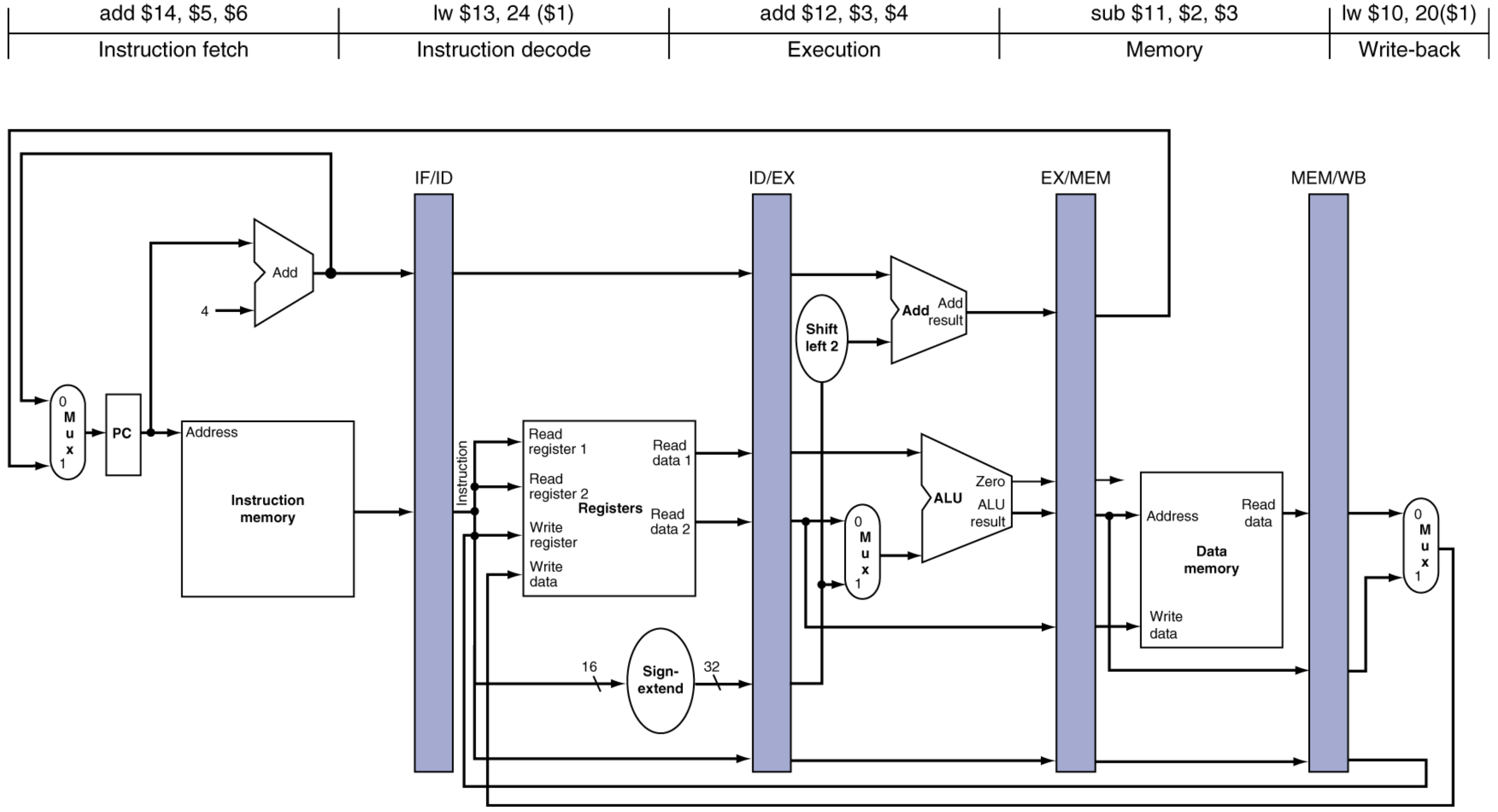
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw \$10, 20(\$1)	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
sub \$11, \$2, \$3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
add \$12, \$3, \$4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
lw \$13, 24(\$1)				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
add \$14, \$5, \$6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

Multi-Cycle Pipeline Diagram

- **Form showing resource usage**



Single-Cycle Pipeline Diagram (CC5)



Pipeline Control Lines

- **We borrow as much as we can from the control for the simple datapath**
  - In particular, we use the same ALU control logic, branch logic, destination-register-number multiplexor, and control lines

- **To specify control for the pipeline, we need only to set the control values during each pipeline stage**
  - No control is required for the IF and ID stages
  - Therefore, we can divide the control lines into three groups according to the pipeline stage

Pipeline Control Lines

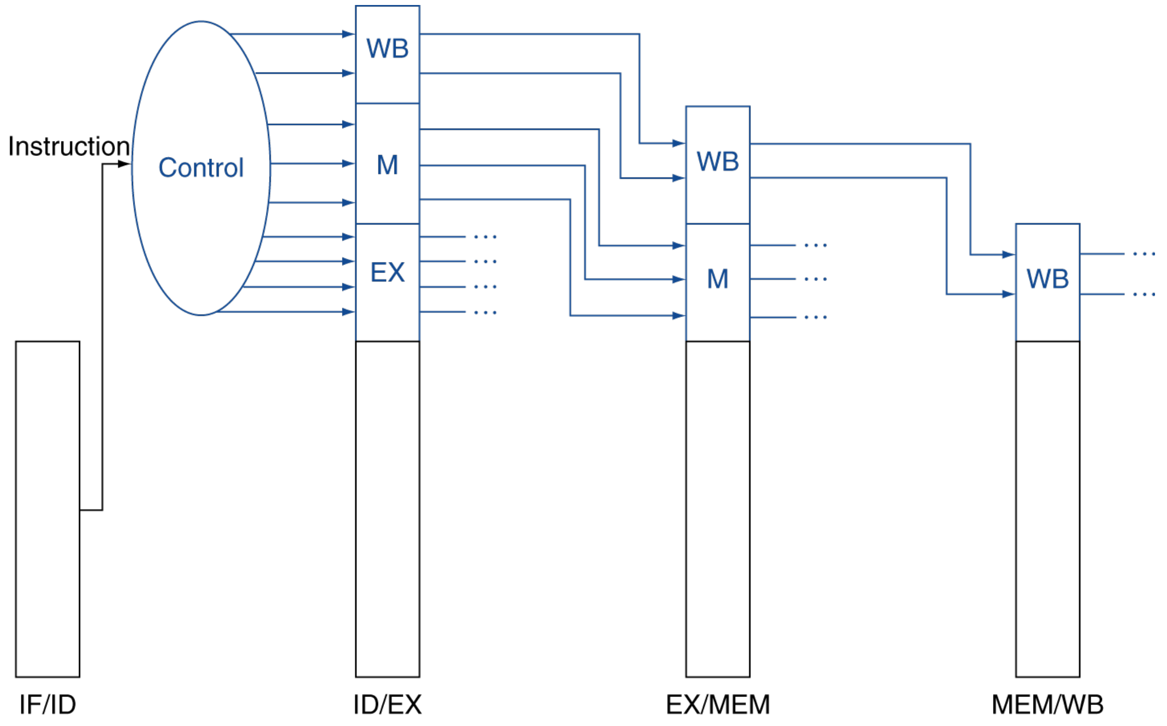
- Set the nine control lines to these values in each stage for each instruction

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

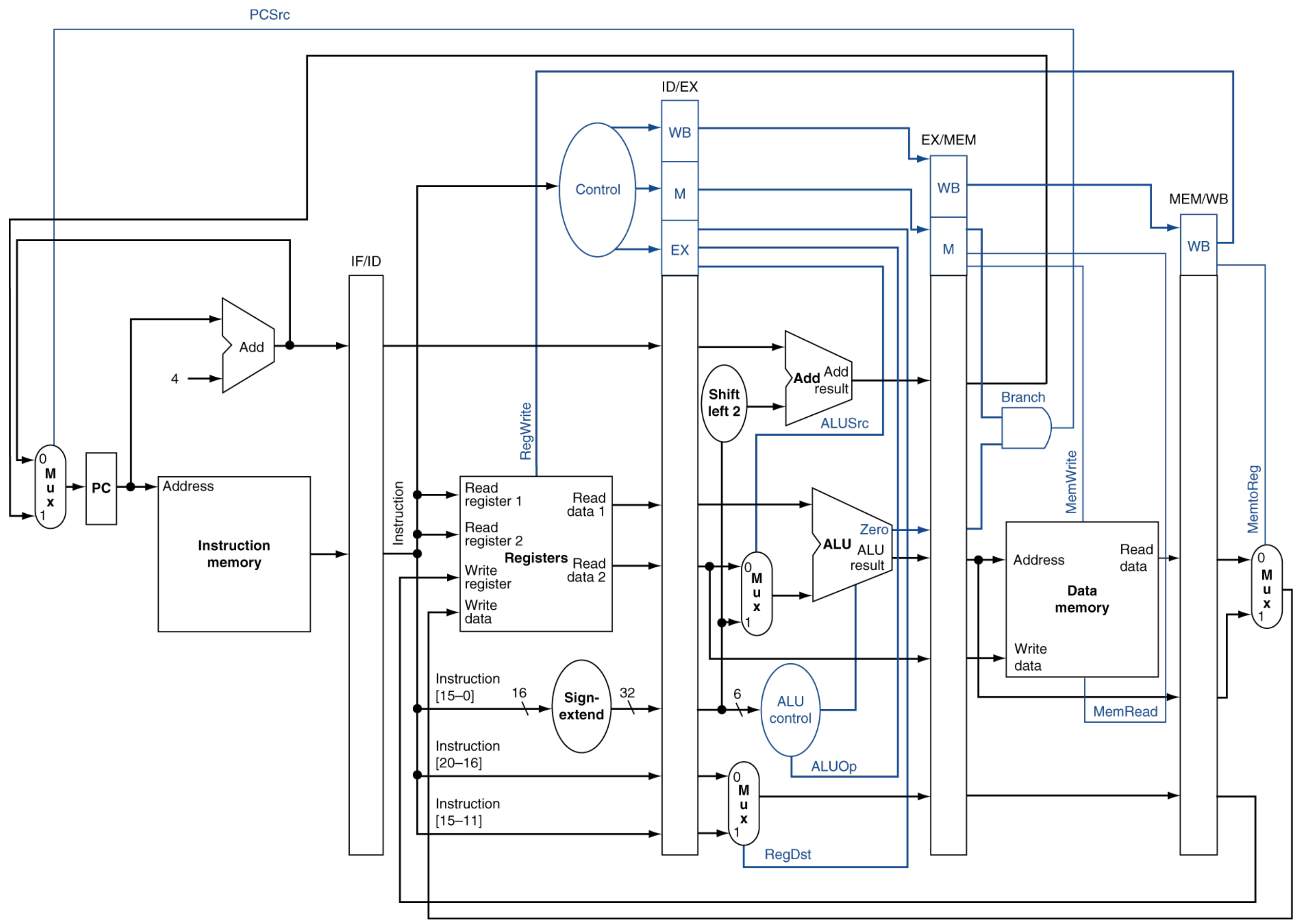
- The simplest way to do this is to **extend the pipeline registers** to include control information

Pipelined Control

- **Control signals derived from instruction**
  - As in single-cycle implementation



Pipelined Control



Pipeline Summary

- **Pipelining improves performance by increasing instruction throughput**
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- **Instruction set design affects complexity of pipeline implementation**
- **Subject to hazards**
  - Structural, data, control

Hazards

- **Situations that prevent starting the next instruction in the next cycle**
- **Structural 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