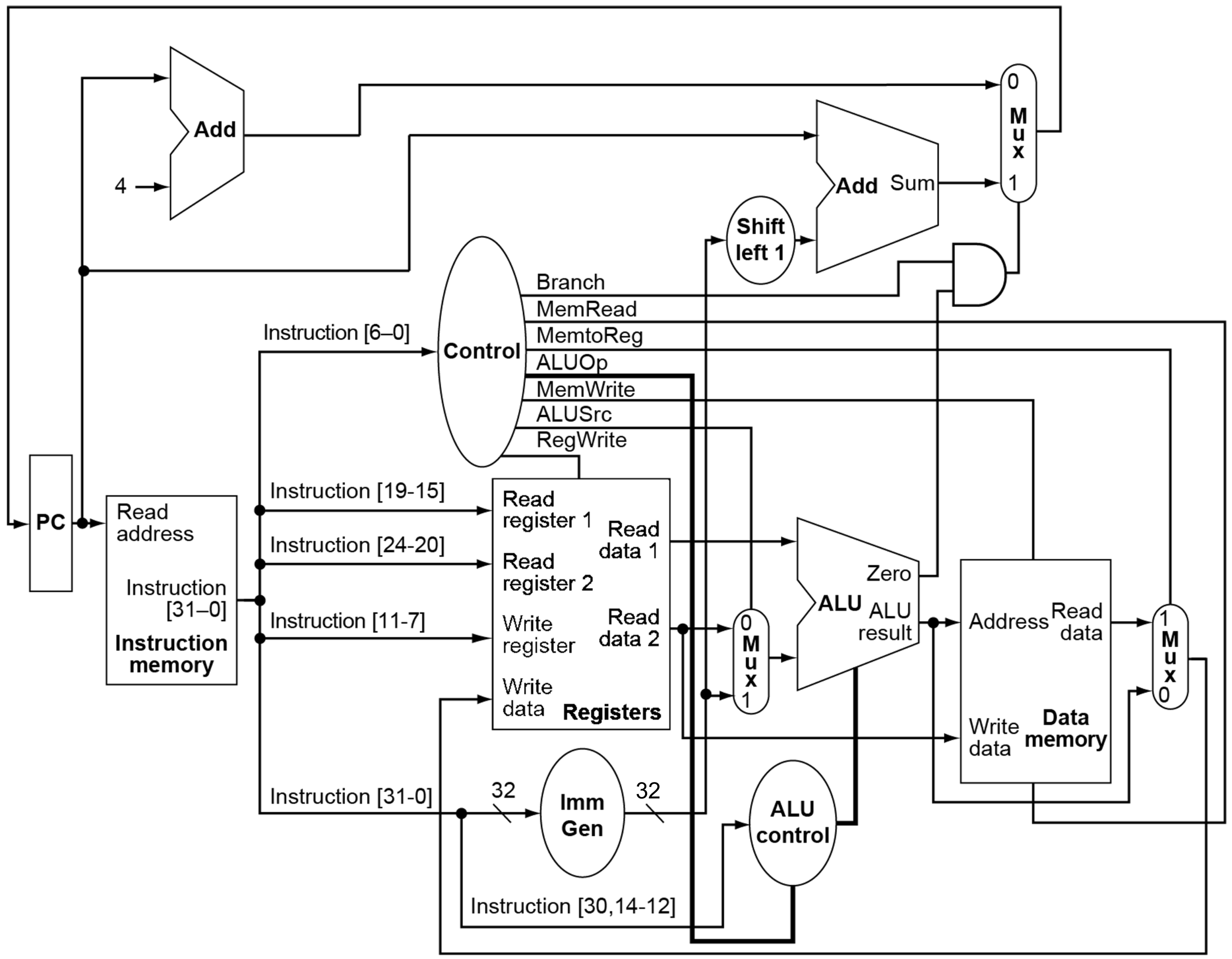
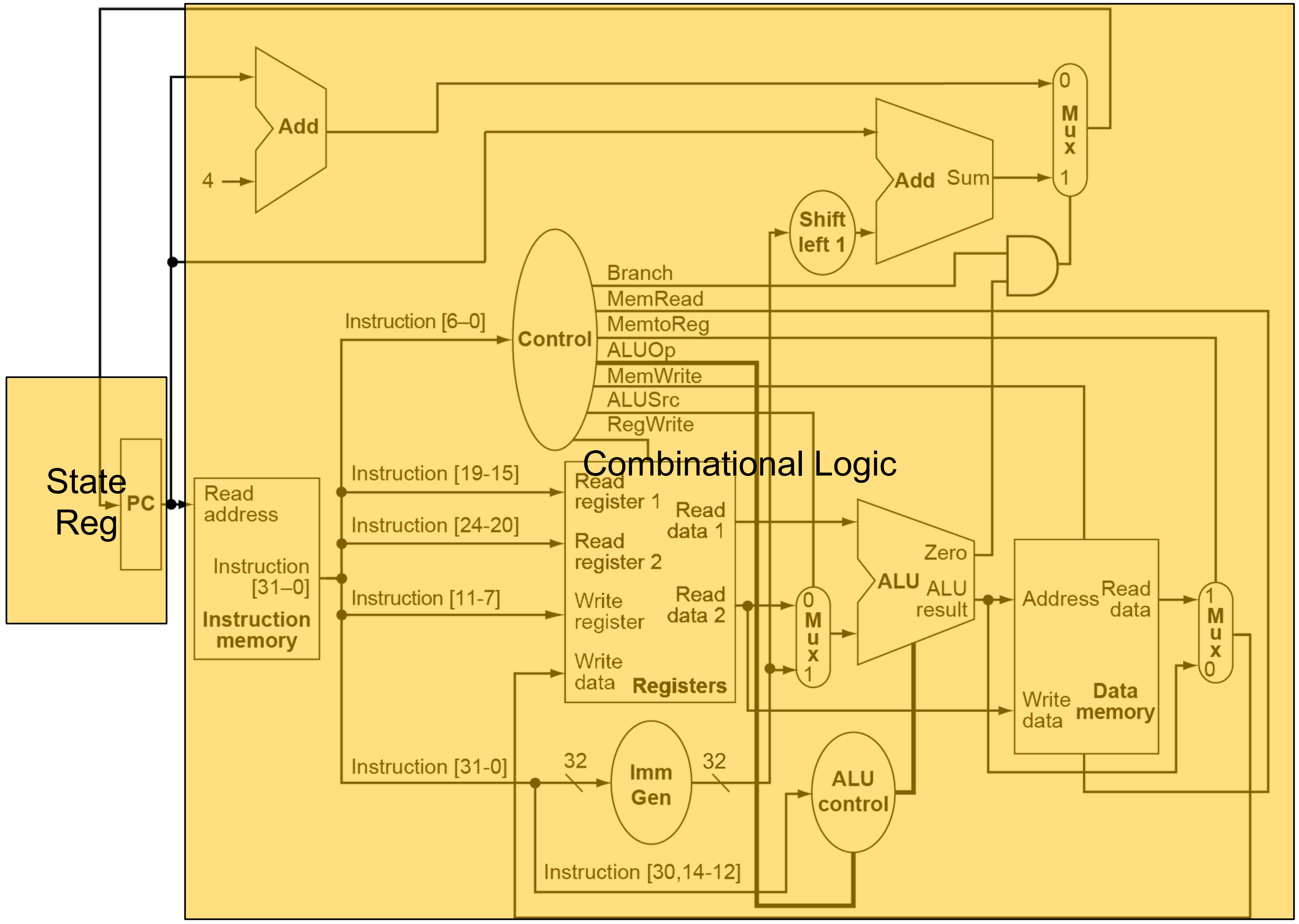
Topic 6

Pipelined Processor

Single Cycle Implementation

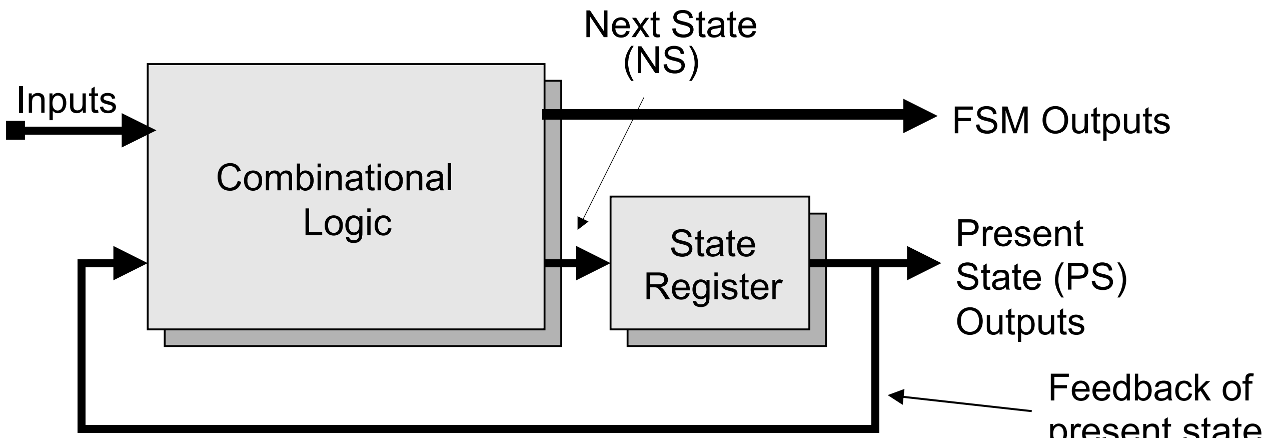


Single Cycle Implementation



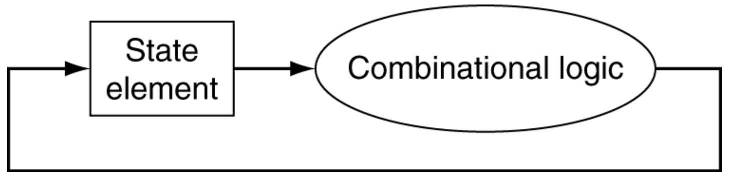
State Machine

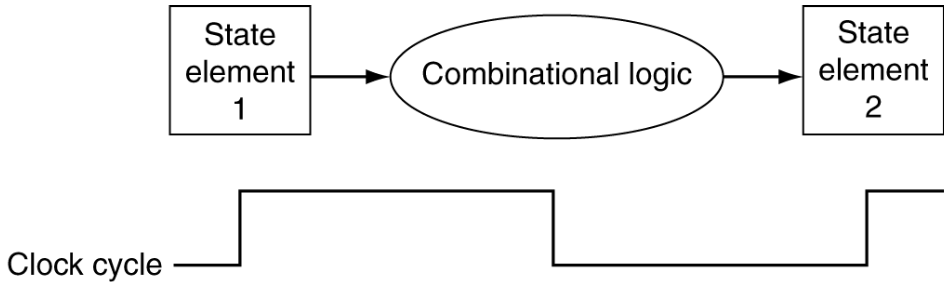
- Finite State Machine



Clocking Methodology for FSM

- Combinational logic transforms data during clock cycles
  - Between clock edges
- Clock cycles should be
  - Long enough to allow combinational logic completes computation
    - Longest delay determines clock period
  - Short enough to ensure acceptable performance and to capture small changes on external inputs





Performance Consideration

- Longest instruction determines the clock cycle time
  - Critical path: load instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file (plus MUXes)
- Not feasible to vary period for different instructions
  - Unless using multi-cycle design
- Many components are doing nothings and waiting – waste of time and resources

Performance Consideration

- Assume times for major components are
  - 100ps for register read or write
  - 200ps for accessing memory
  - 200ps for ALU operations

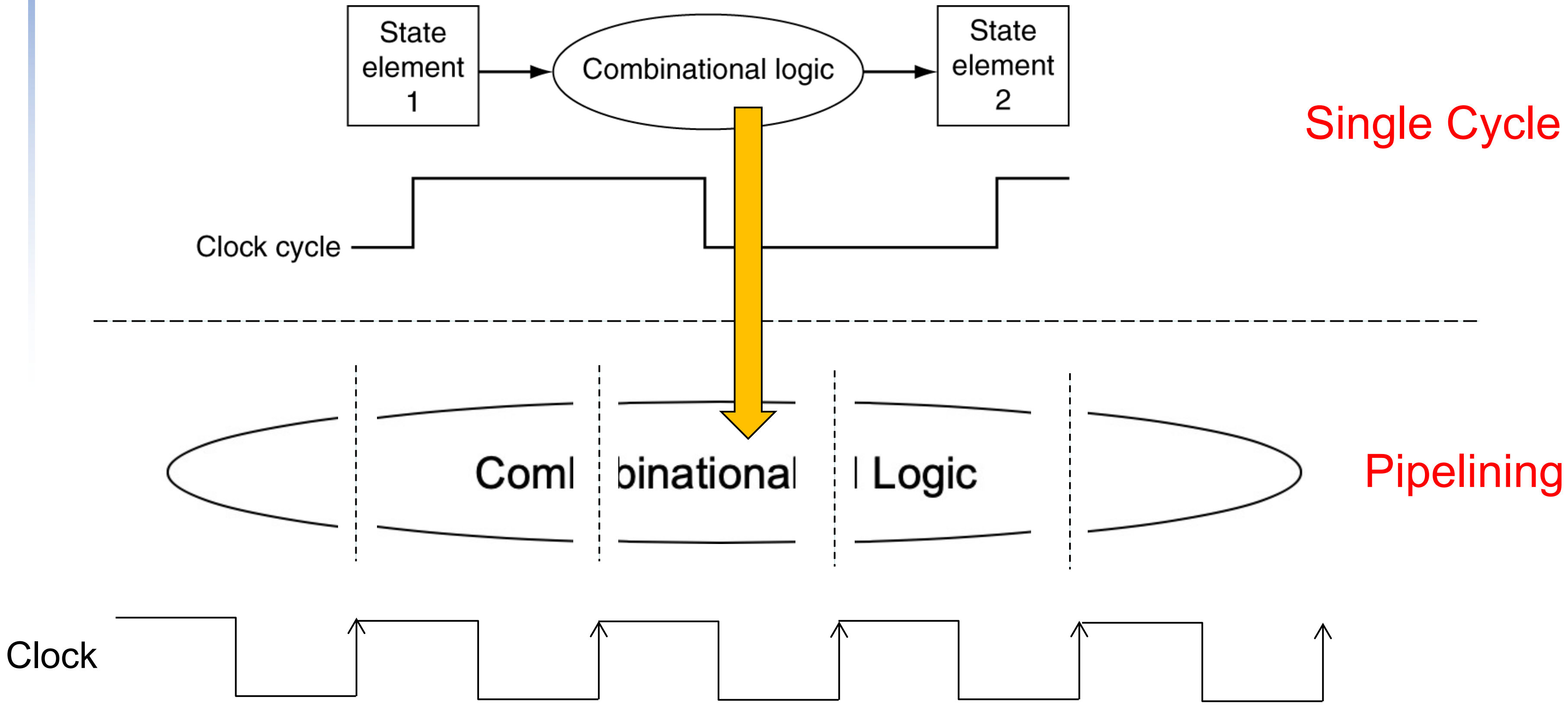
Instruction	Instr fetch	Register read	ALU op	Data Memory	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Performance Consideration

- Assume 100 instructions are executed
  - 15% are loads
  - 15% are stores
  - 40% are R format instructions
  - 30% are branches
- What’s the clock cycle time for single-cycle processor?
- Execution time using single-cycle processor?

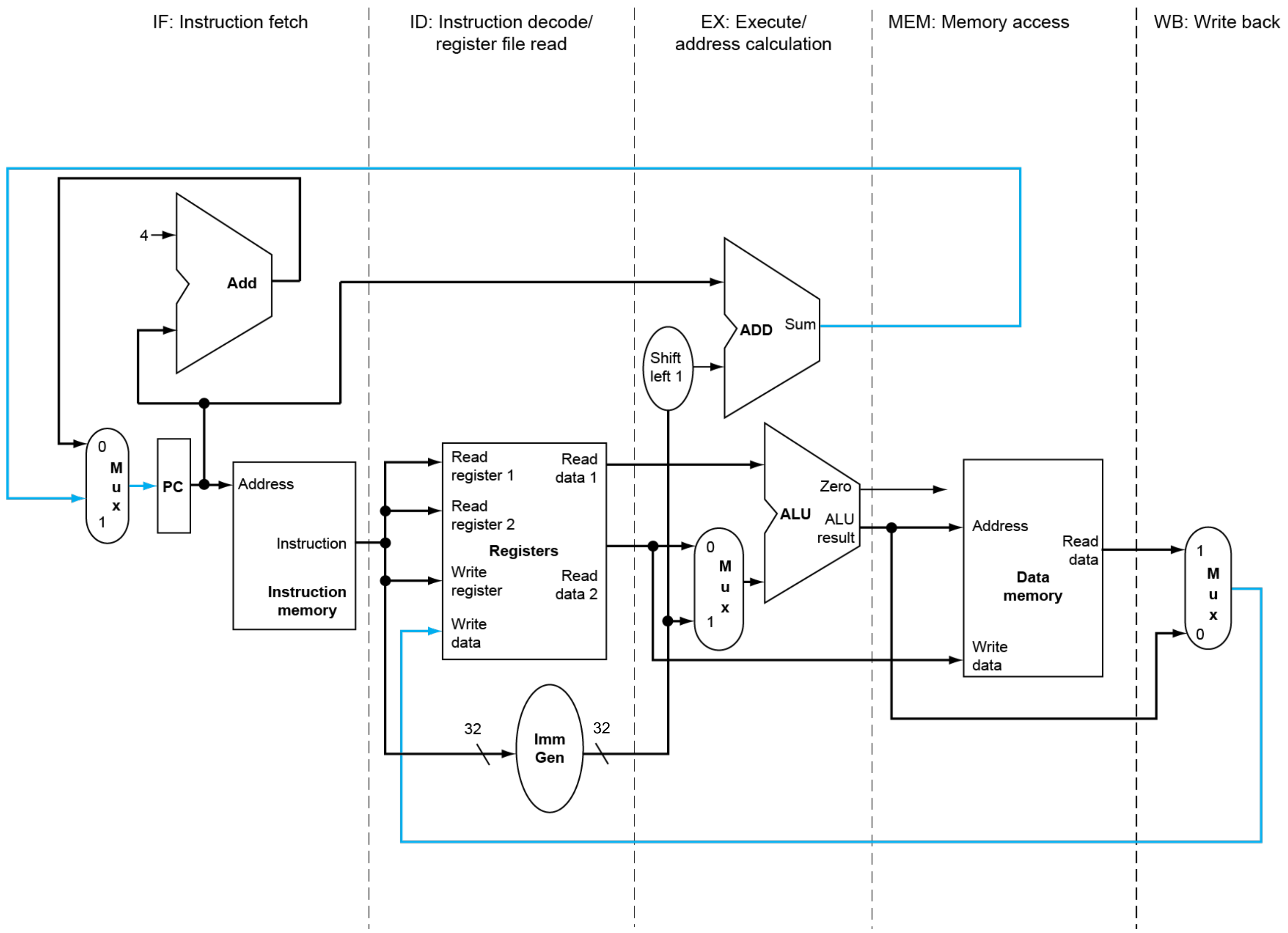
Improvement – Pipelining

- Divide the combinational logic into smaller pieces



- Each piece is finished in shorter time

RISC-V Pipelined Datapath

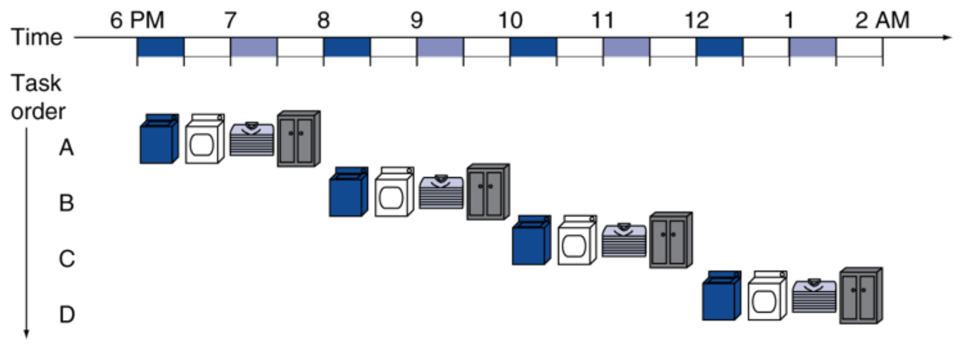


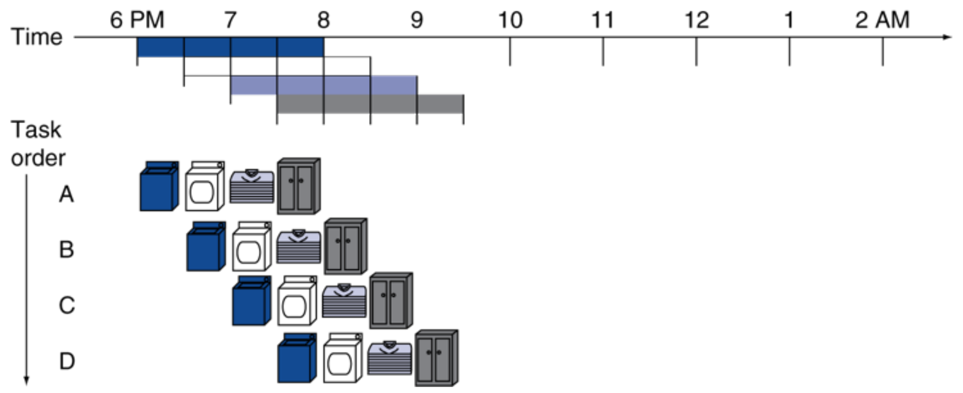
RISC-V Pipeline

- Five stages, one step per stage per cycle
  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

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

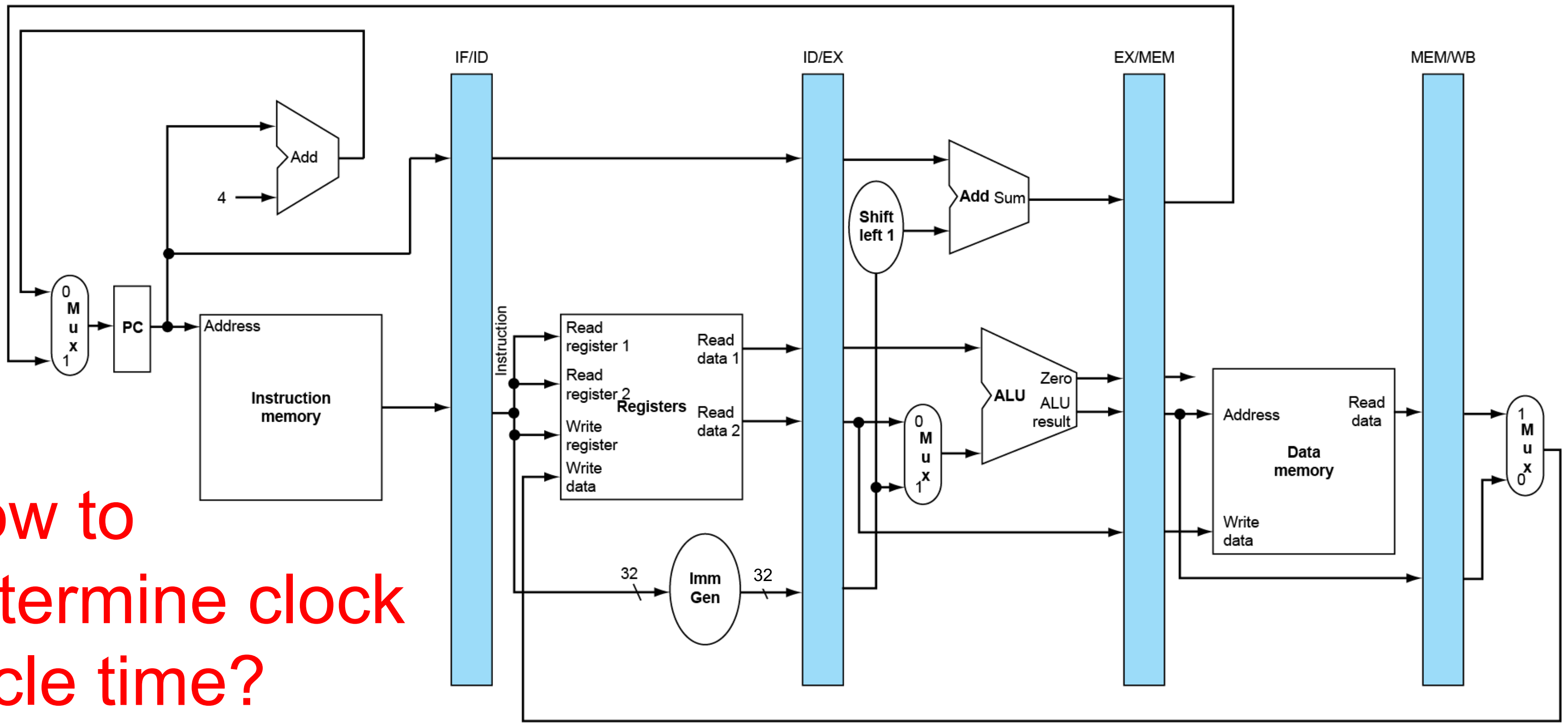




- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup ≈ 2*n/0.5*n = 4 = number of stages
  - n is number of instructions

Pipeline registers

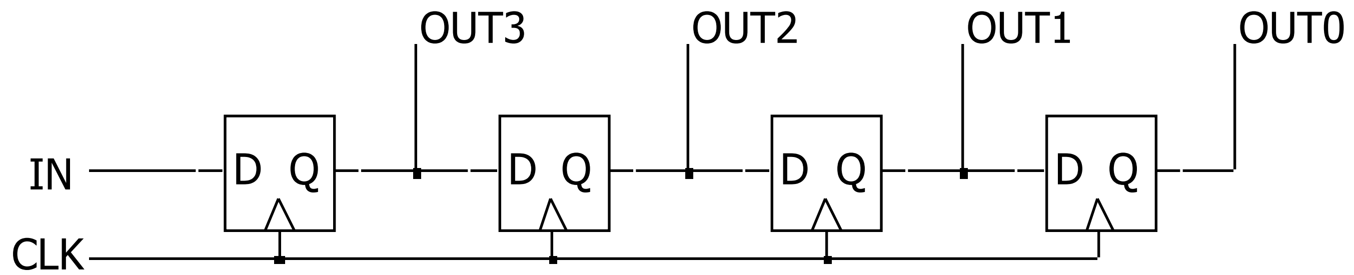
- Need registers between stages
  - To hold information produced in previous cycle



How to determine clock cycle time?

Recall the Shift Register

- Implementation:
  - Connect Q output of one flip flop to the D input of the next flip flop
  - 4-bit shift register

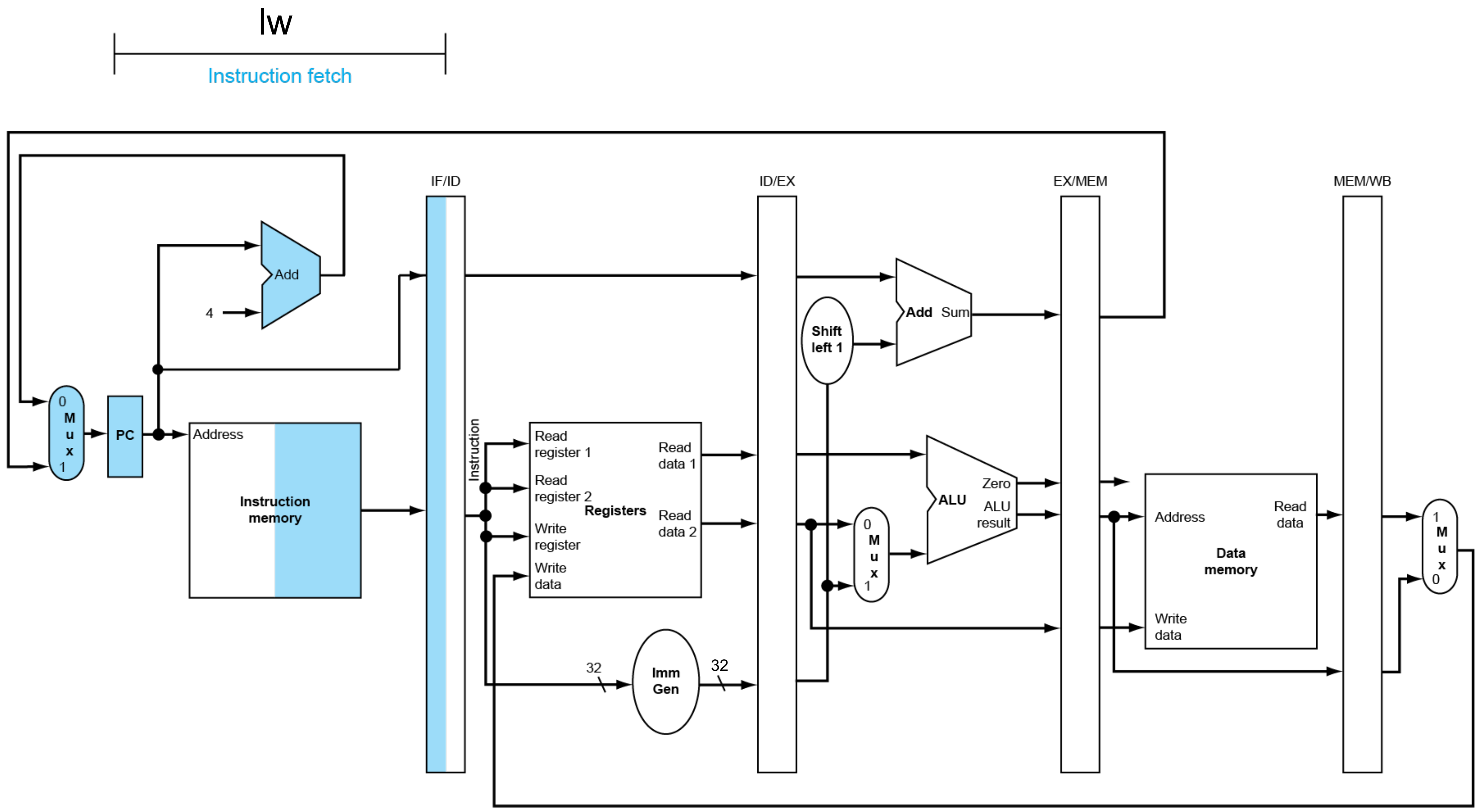


	IN	OUT(3:0)
Initial value:	0	0110
rising edge:	0	0011
rising edge:	0	0001
rising edge:	0	0000
rising edge:	1	1000
rising edge:	0	0100

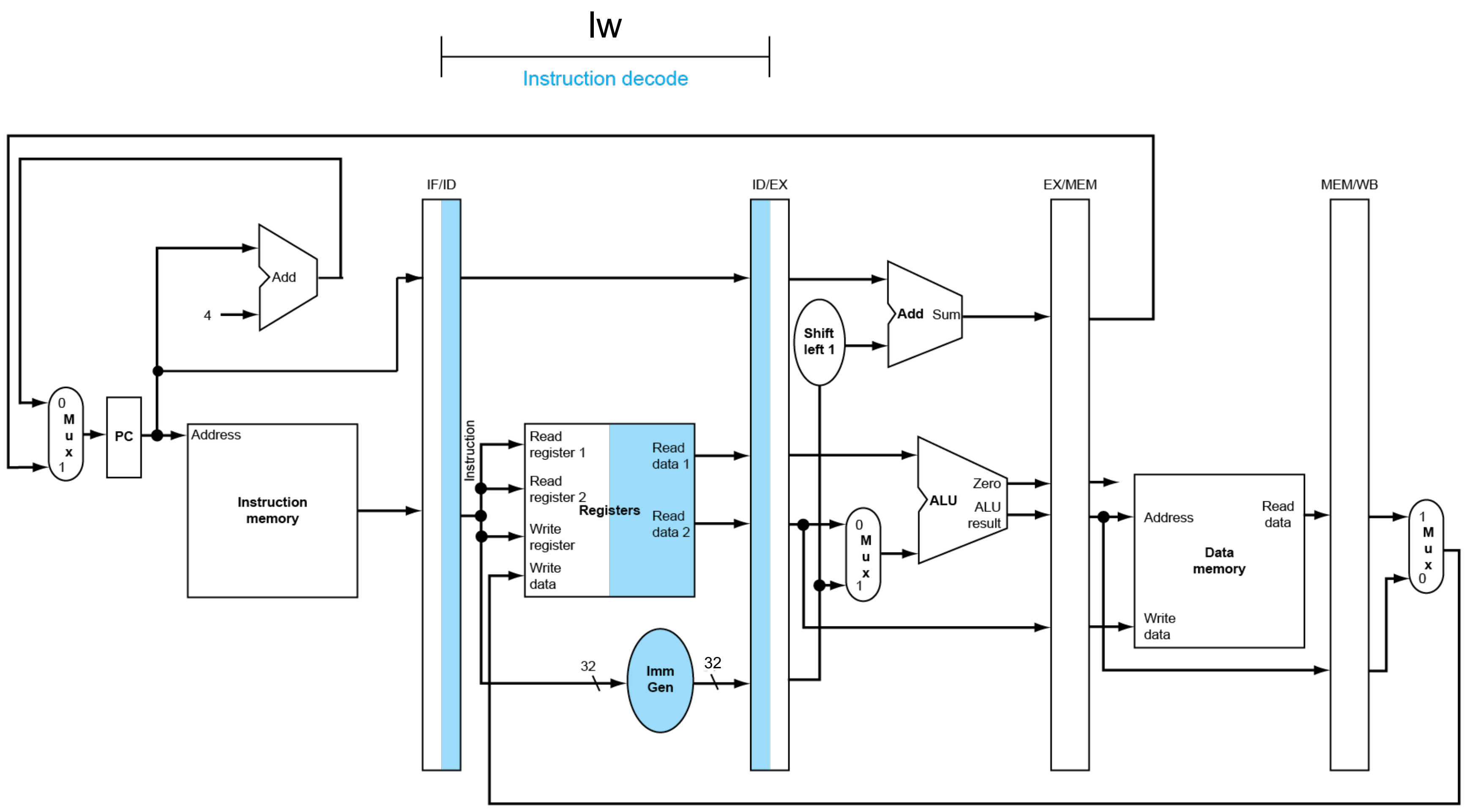
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
- Representation/illustration:
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - “multi-clock-cycle” diagram
    - Graph of operation over time

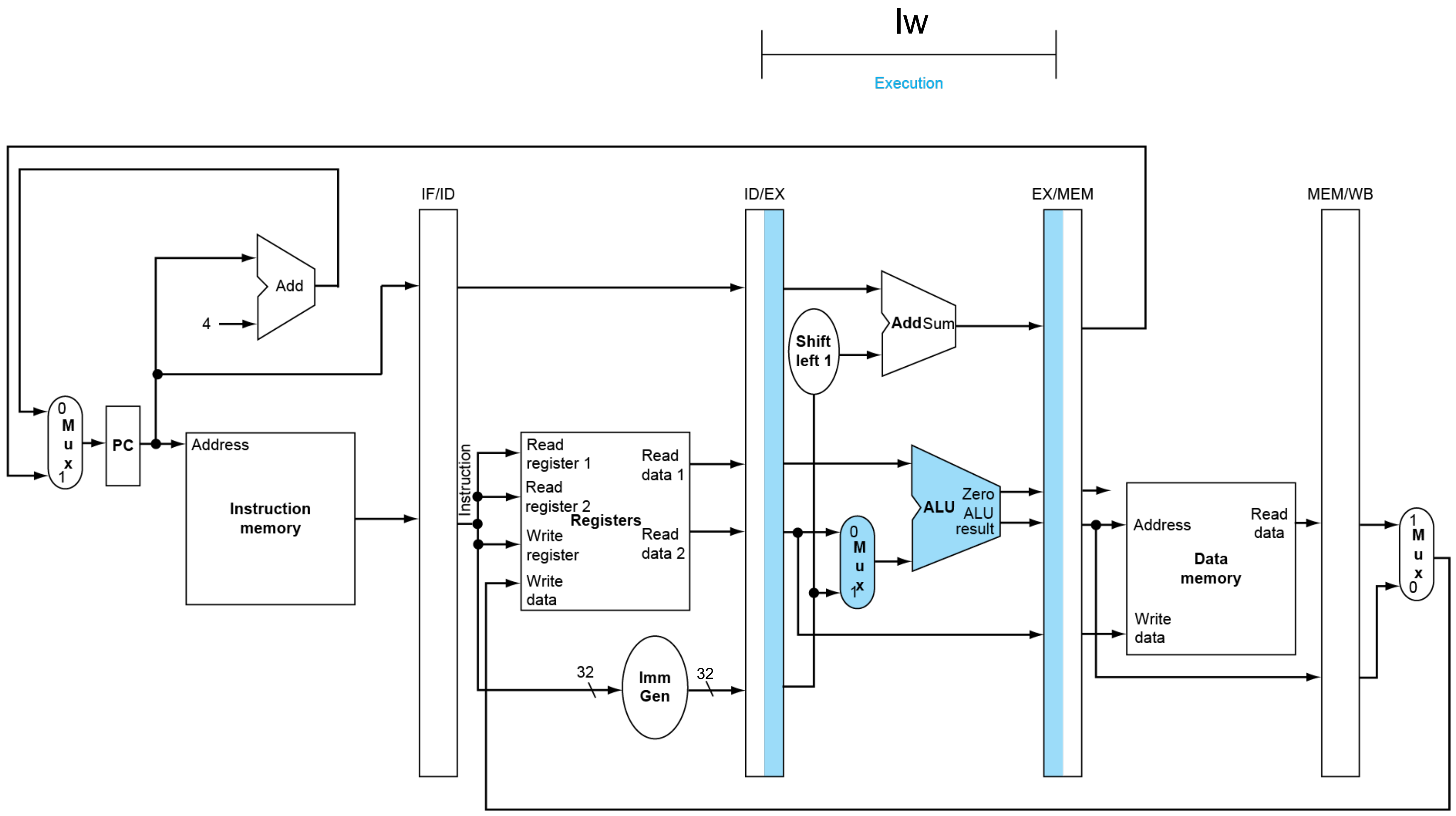
IF for Load, Store, …



ID for Load, Store, …

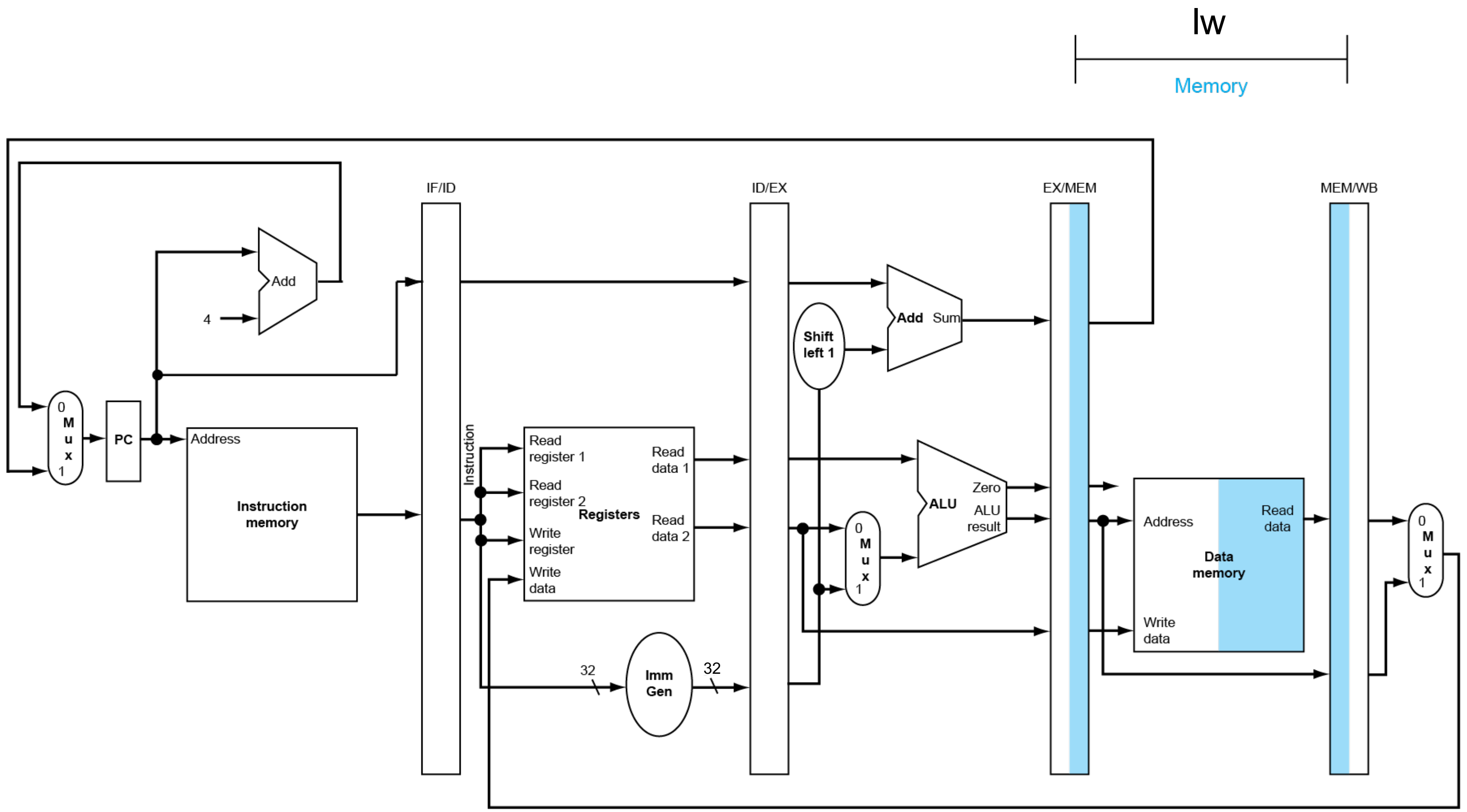


EX for Load



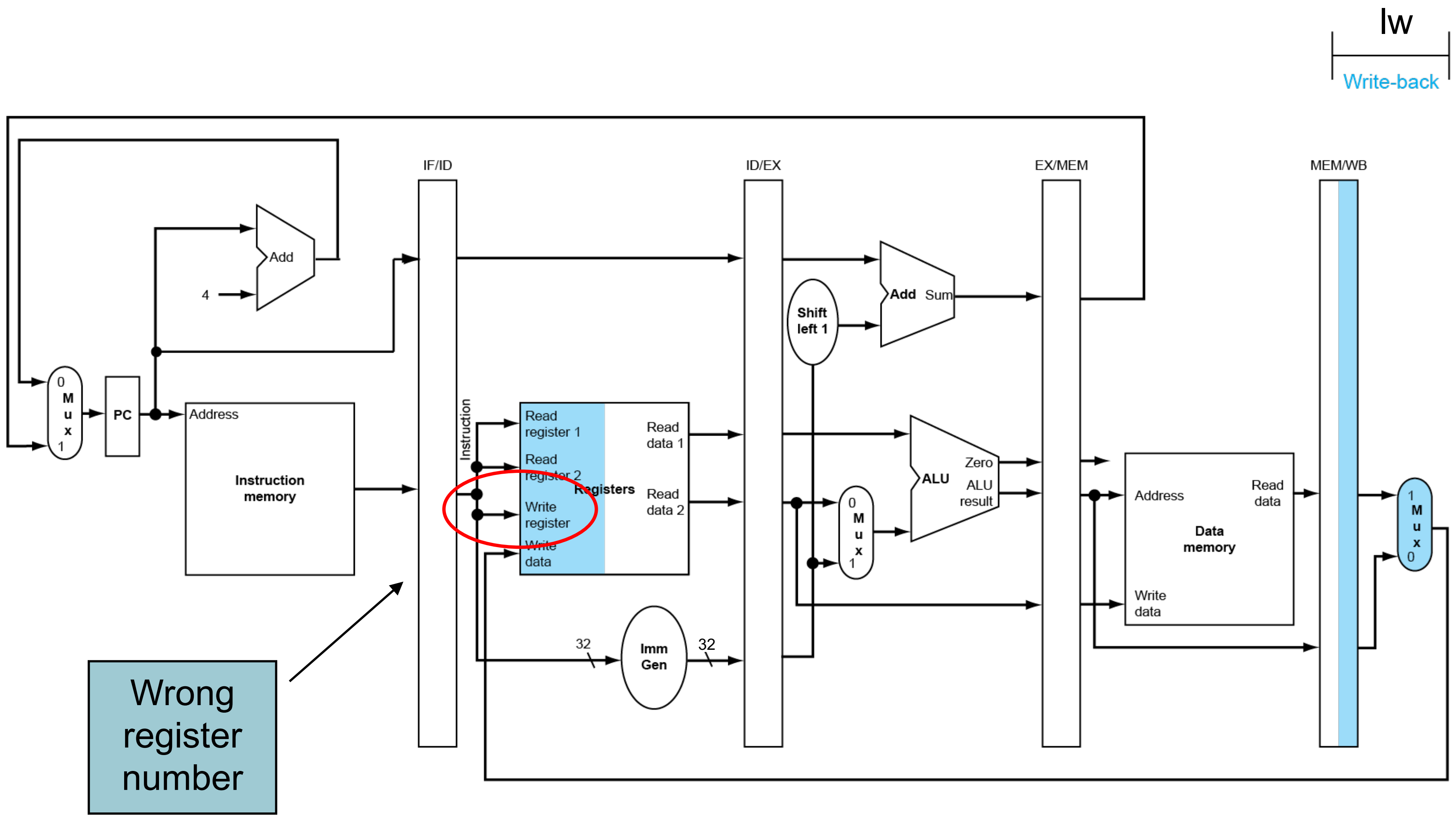
lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0 M u x 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
32
Imm Gen
32
Shift left 1
Add Sum
0 M u x 1
ALU
Zero
ALU result
Address
Data memory
Write data
Read data
1 M u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

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
32
Imm Gen
32
Shift left 1
Add Sum
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

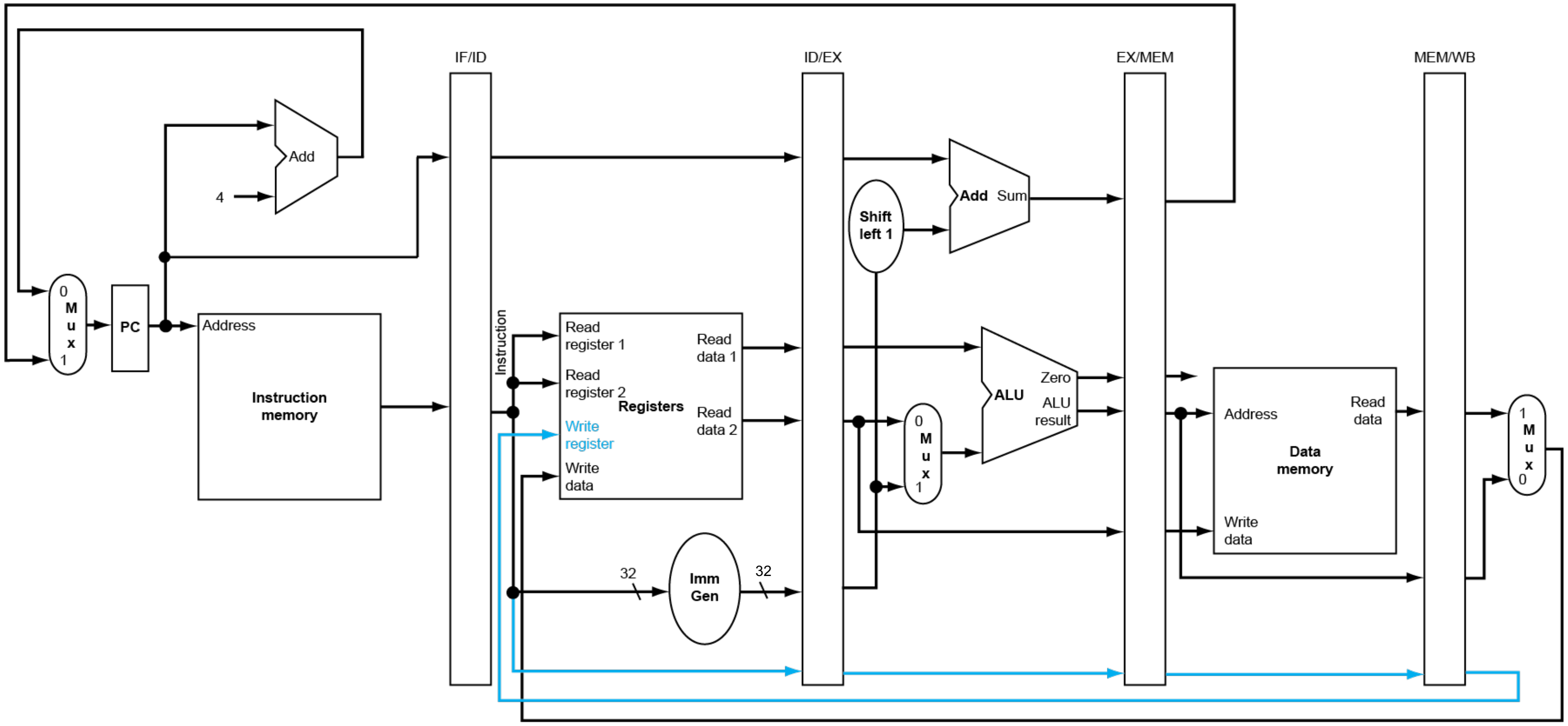
WB for Load



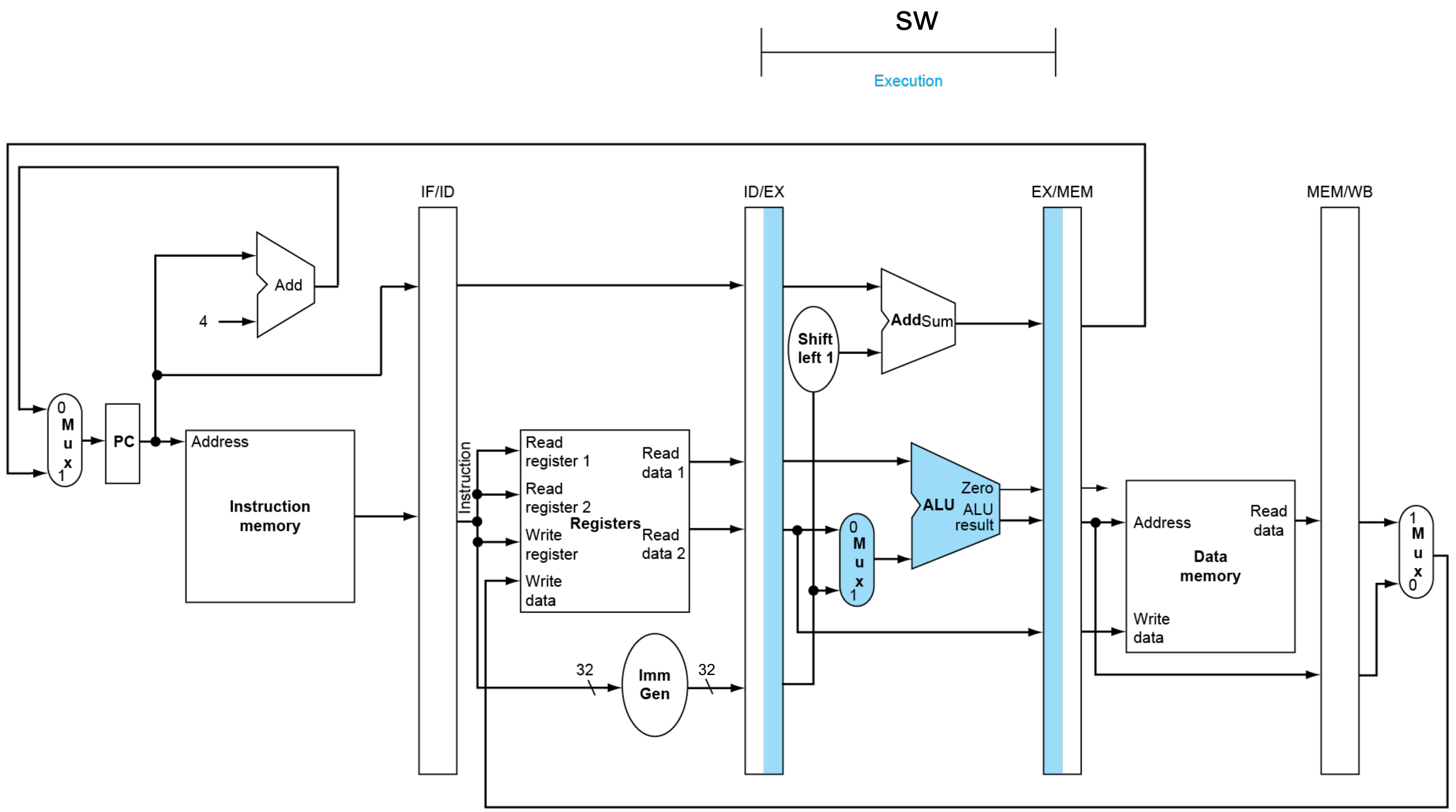
Wrong register number

Corrected Datapath for Load

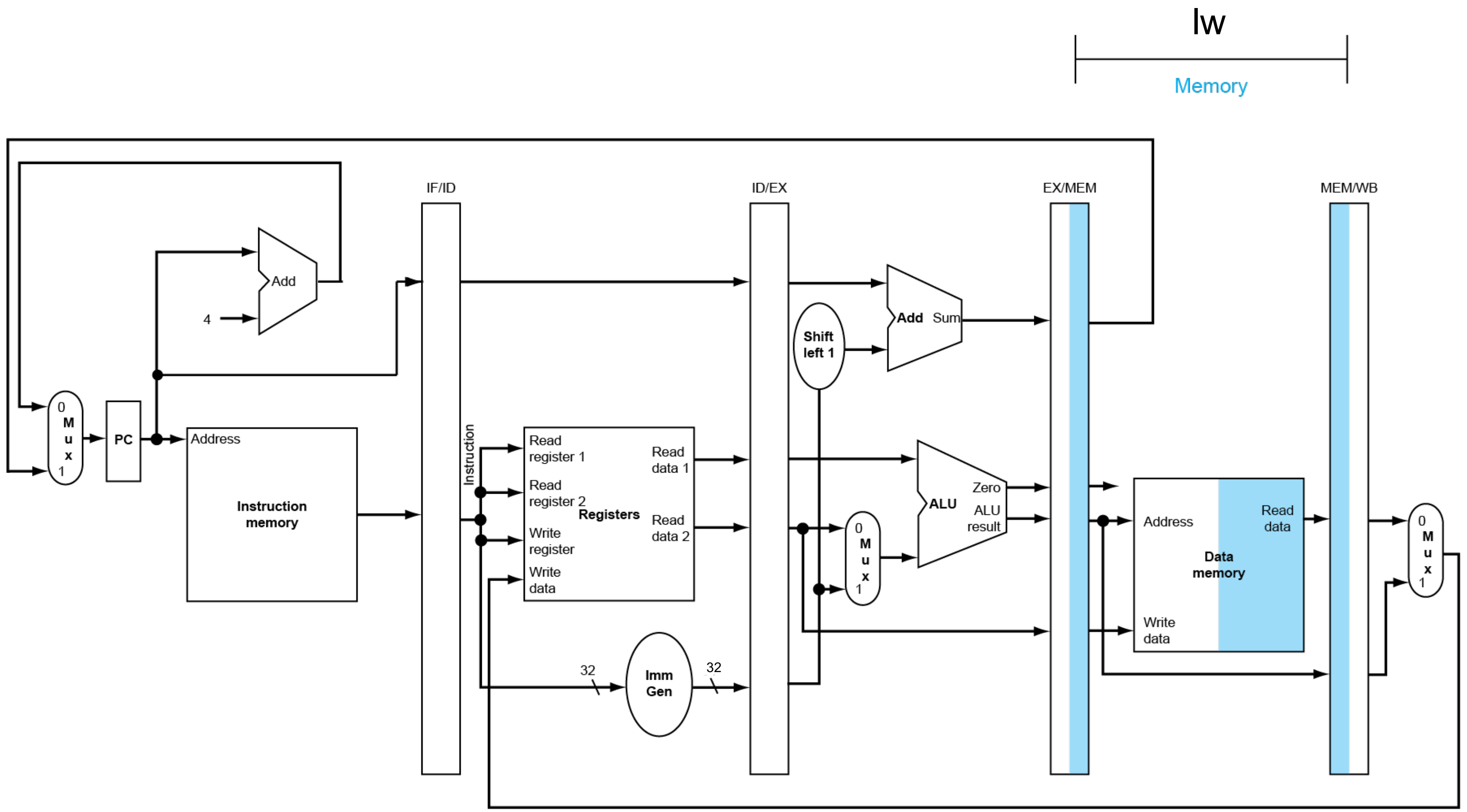
- Pass alive signals along through the pipeline
- Has to write/read register file at the same time
  - Writing reg in first half of clock
  - Reading reg in second half of clock



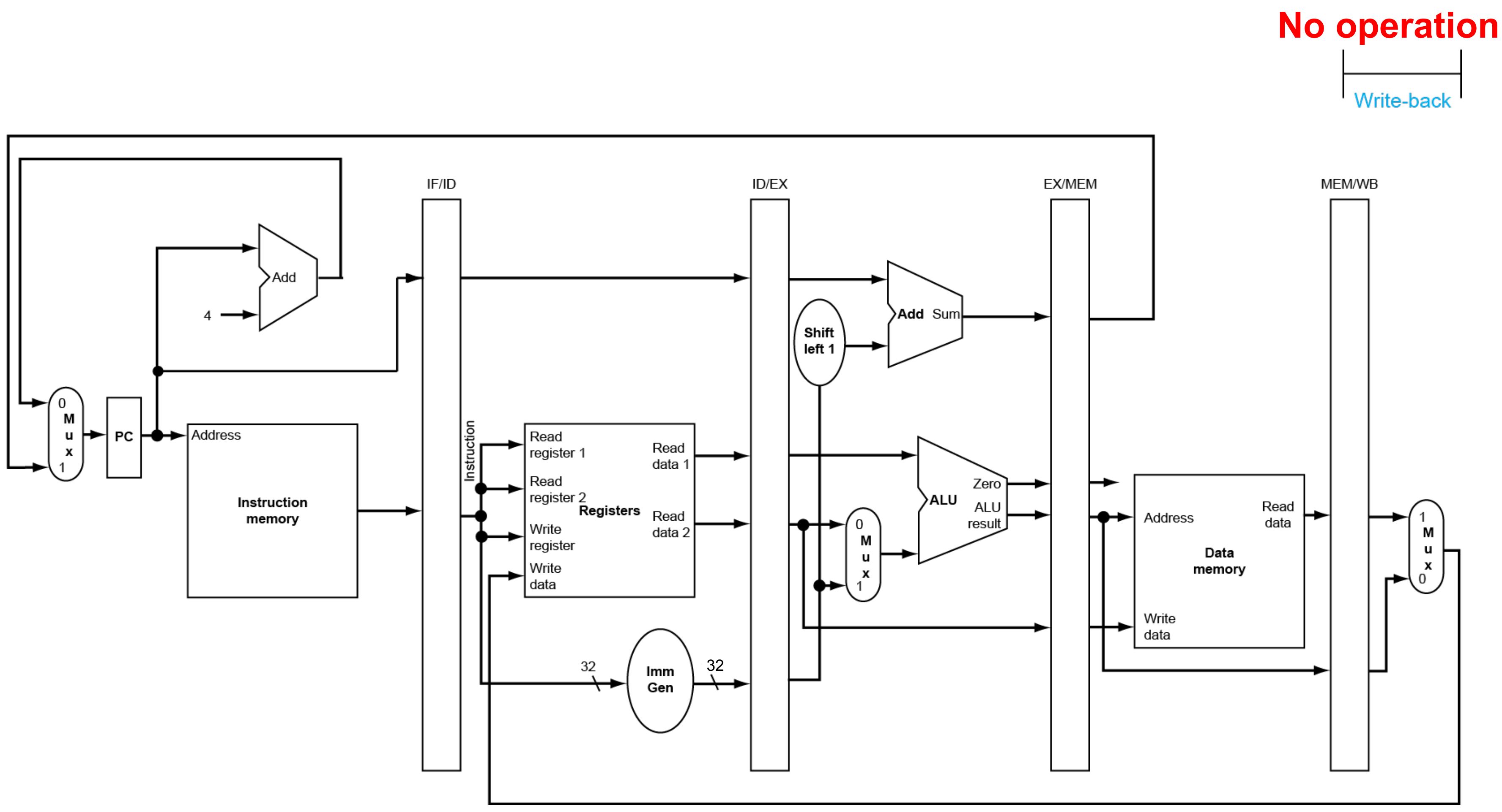
EX for Store


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
32
Imm Gen
32
Shift left 1
Add Sum
0
M
u
x
1
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
1
M
u
x
0

MEM for Store

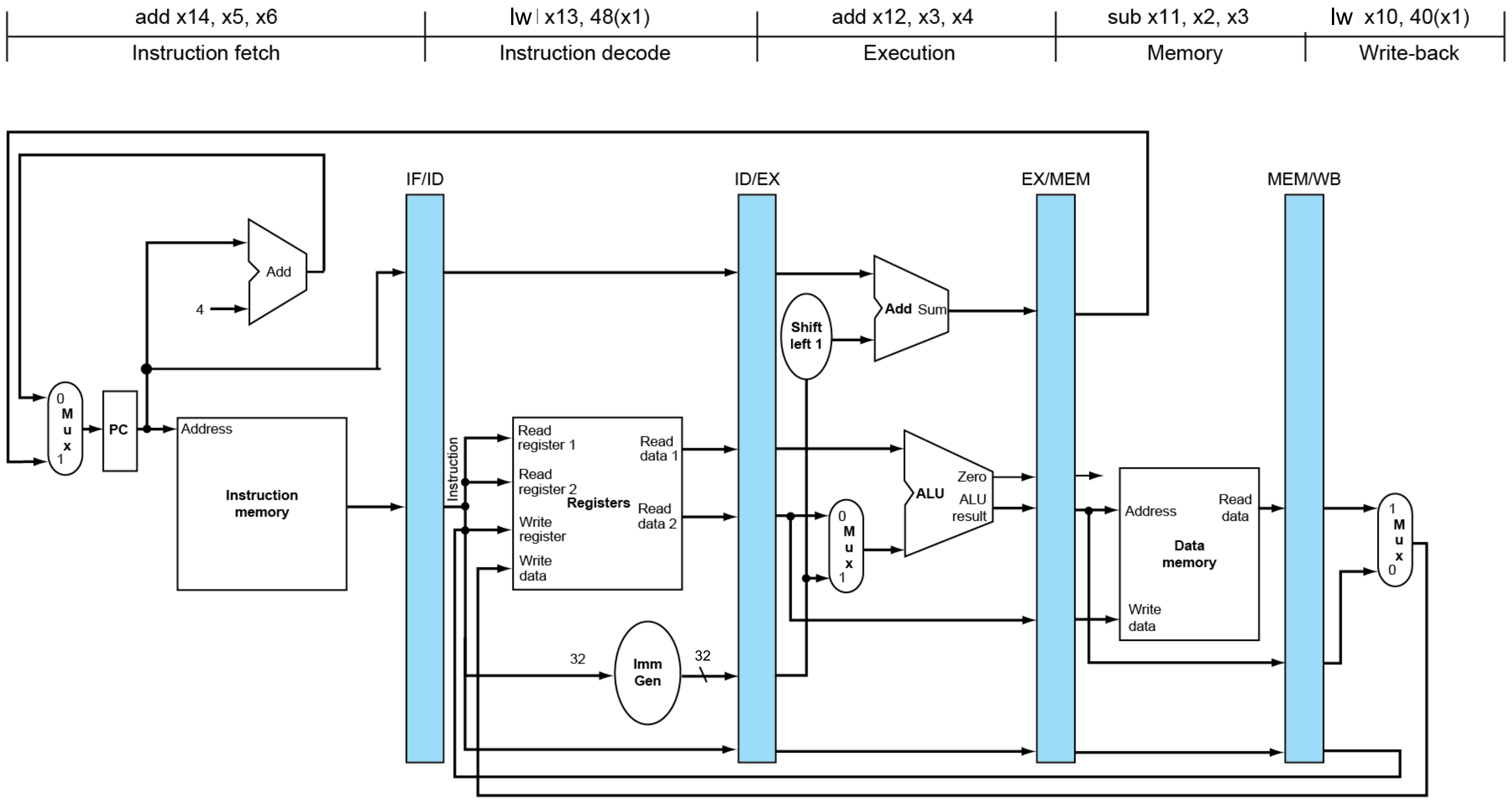


WB for Store


No operation
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Imm Gen
32
32
Shift left 1
Add Sum
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
0 Mux 1
1 Mux 0

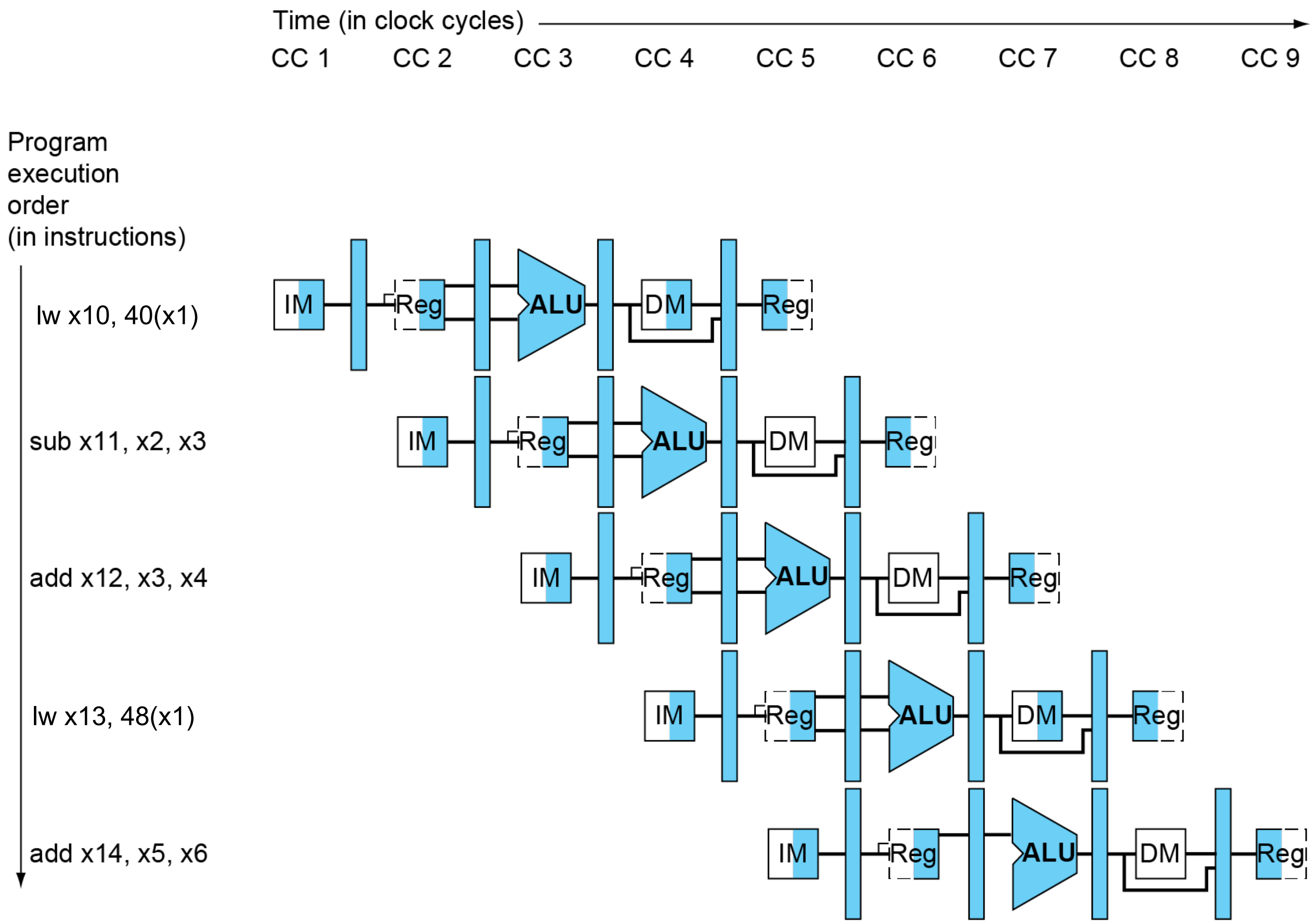
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle



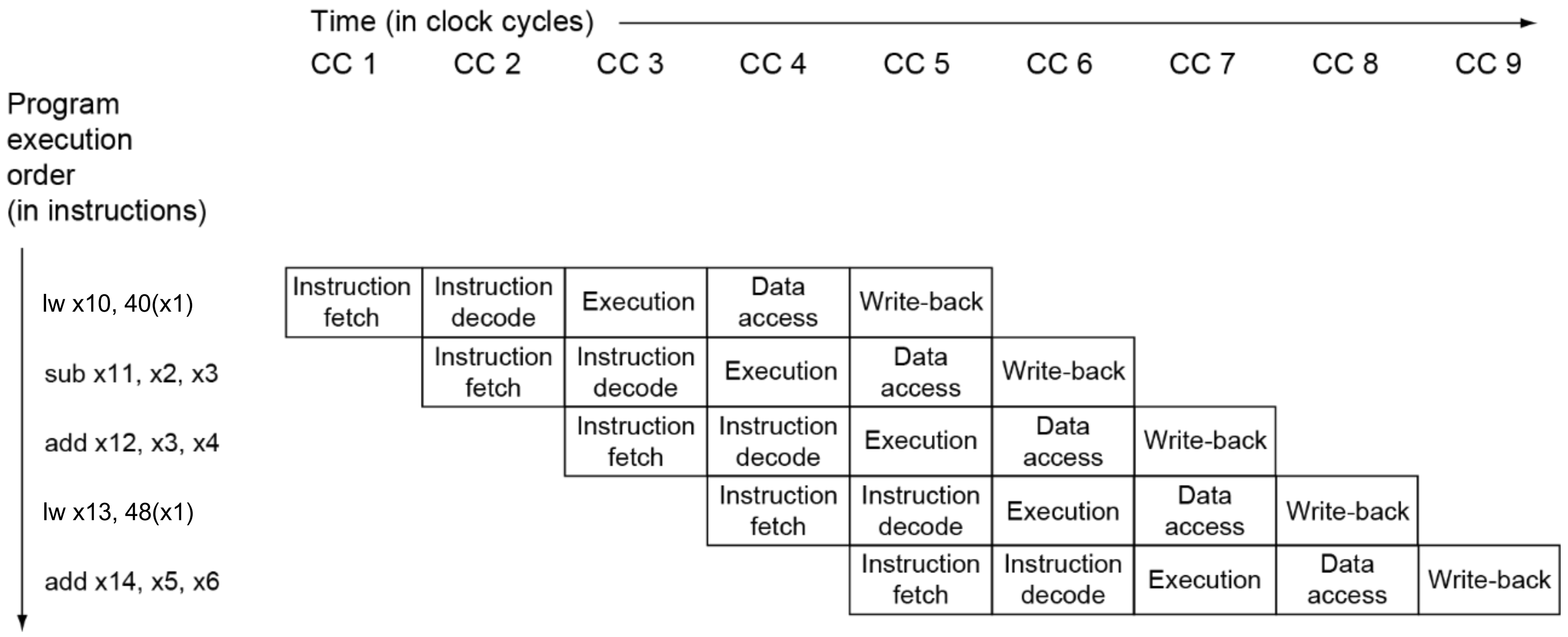
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

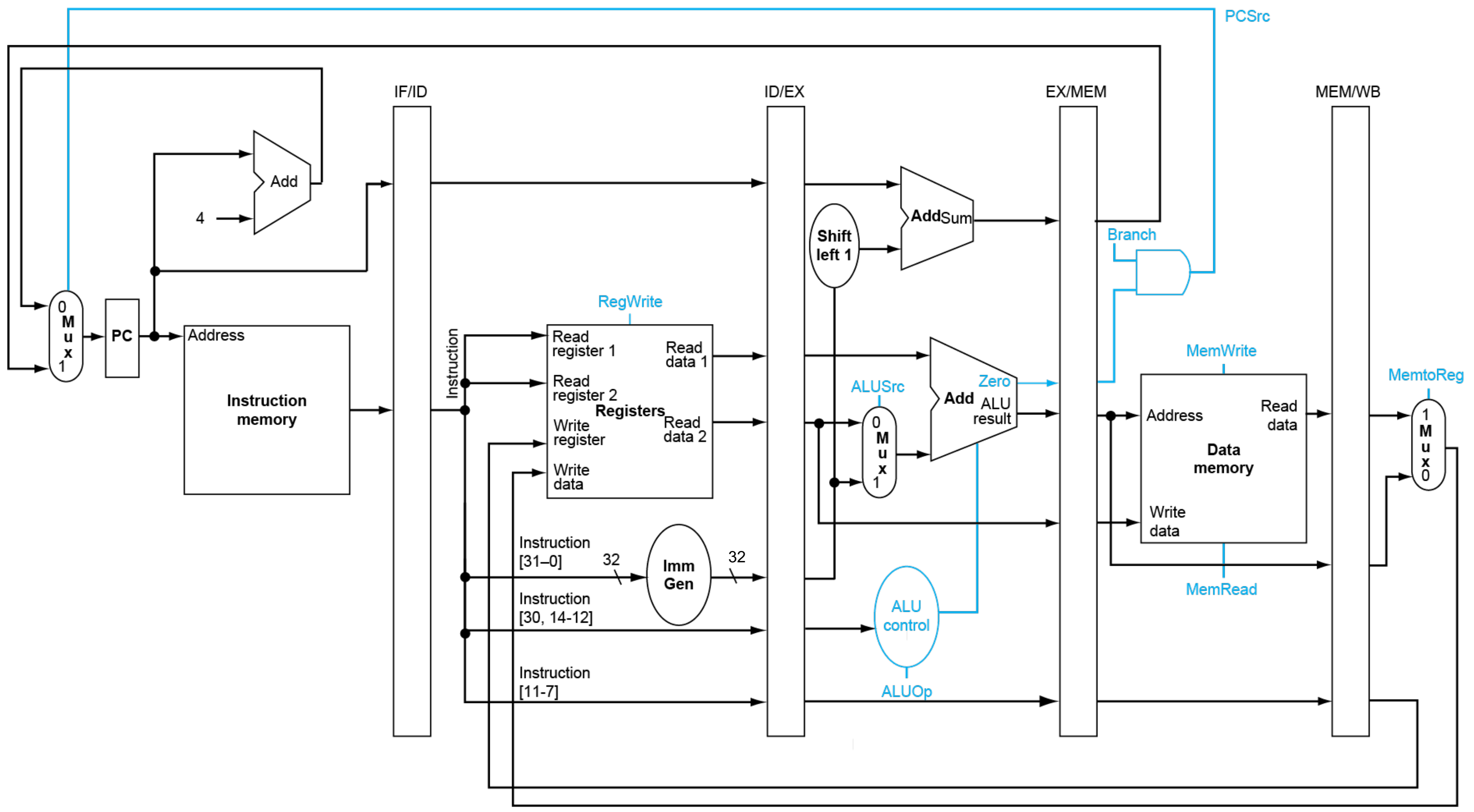
- Traditional form



Time (in clock cycles) →

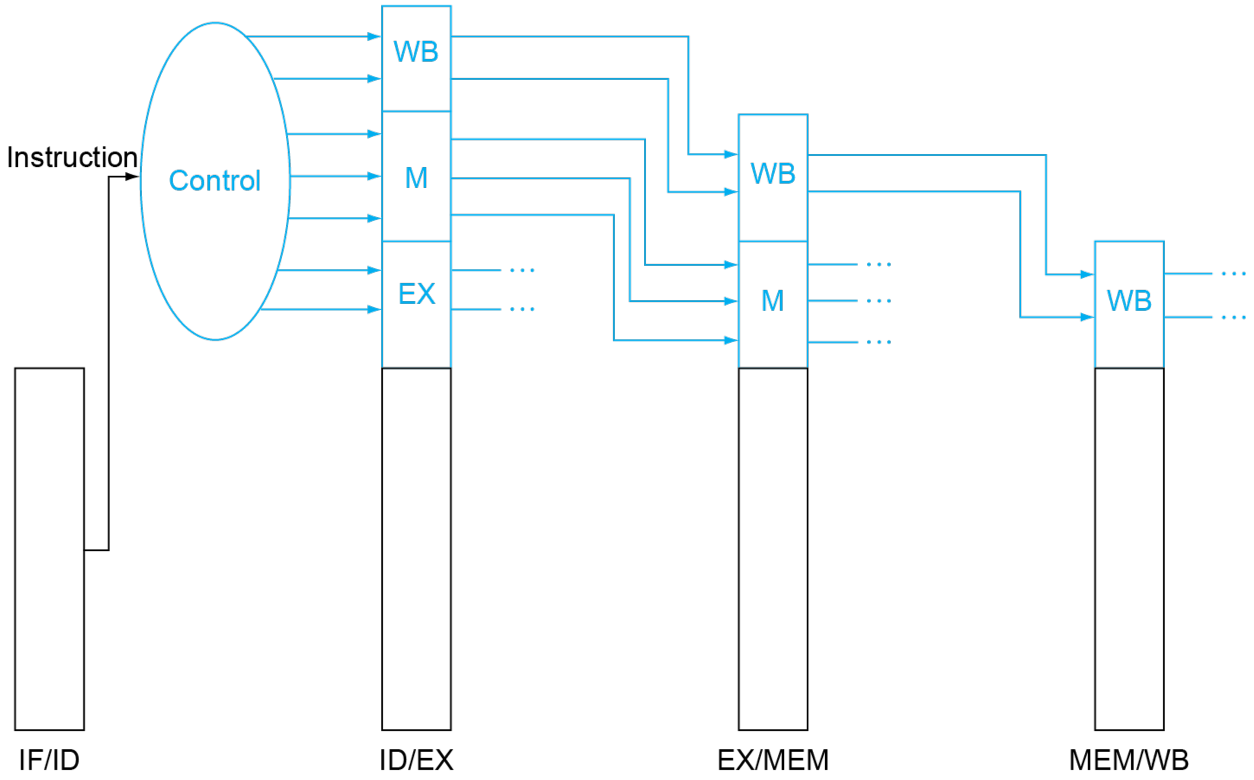
Program execution order (in instructions) ↓	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw x10, 40(x1)	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
sub x11, x2, x3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
add x12, x3, x4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
lw x13, 48(x1)				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
add x14, x5, x6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

Pipelined Control (Simplified)

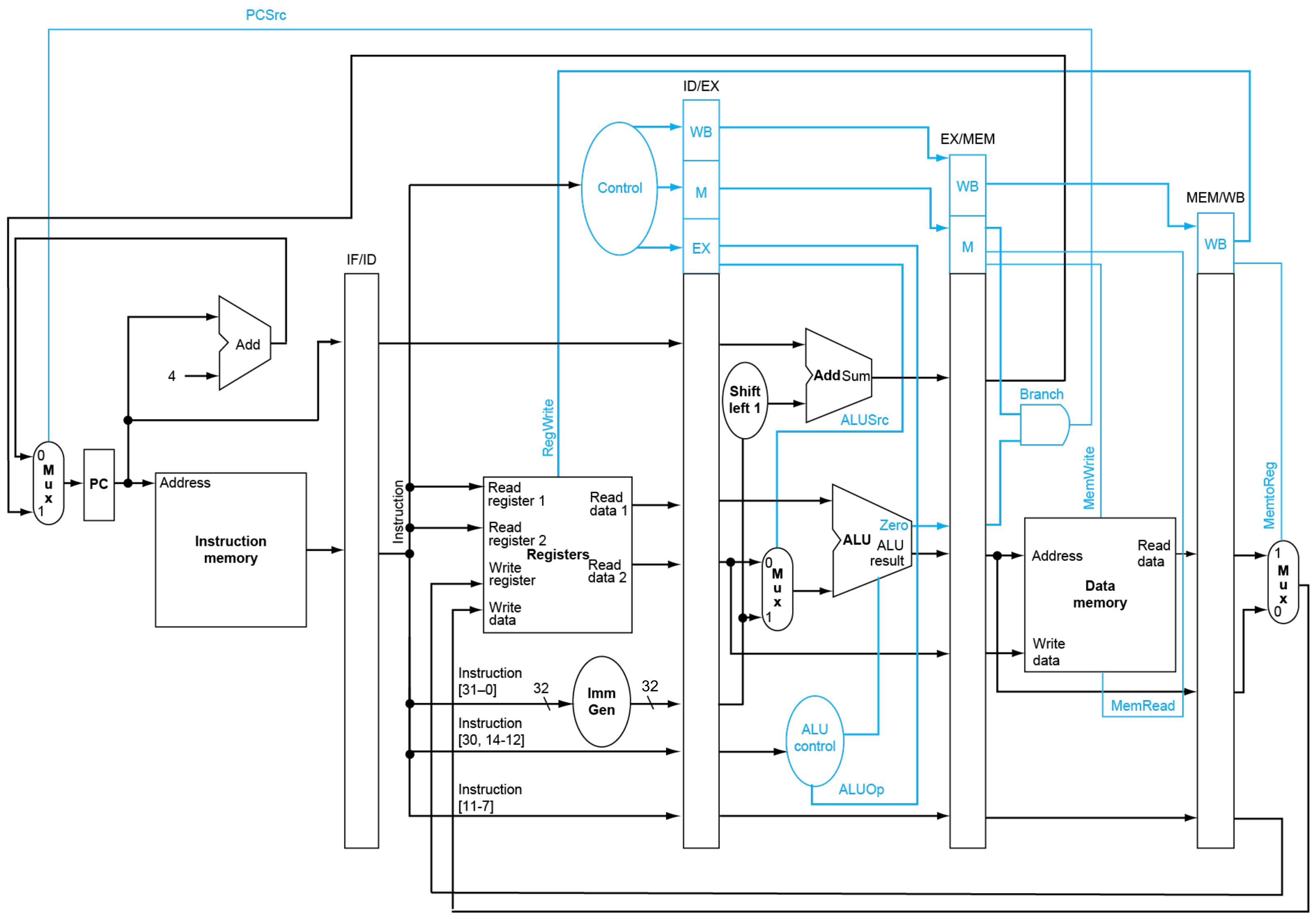


Pipelined Control

- Control signals derived from instruction
  - Passed along with corresponding instruction
  - Consumed in appropriate stages

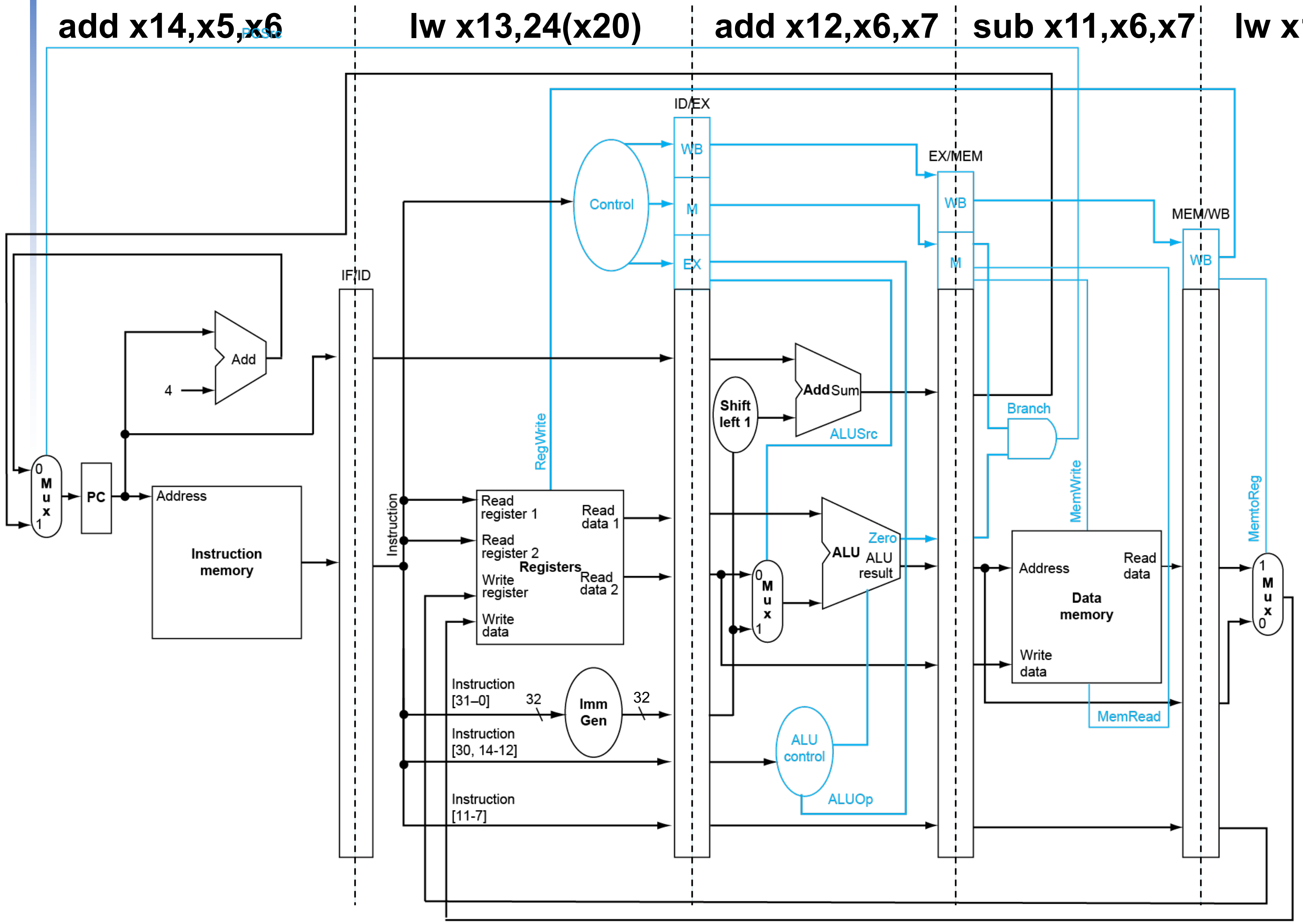


Pipelined Control



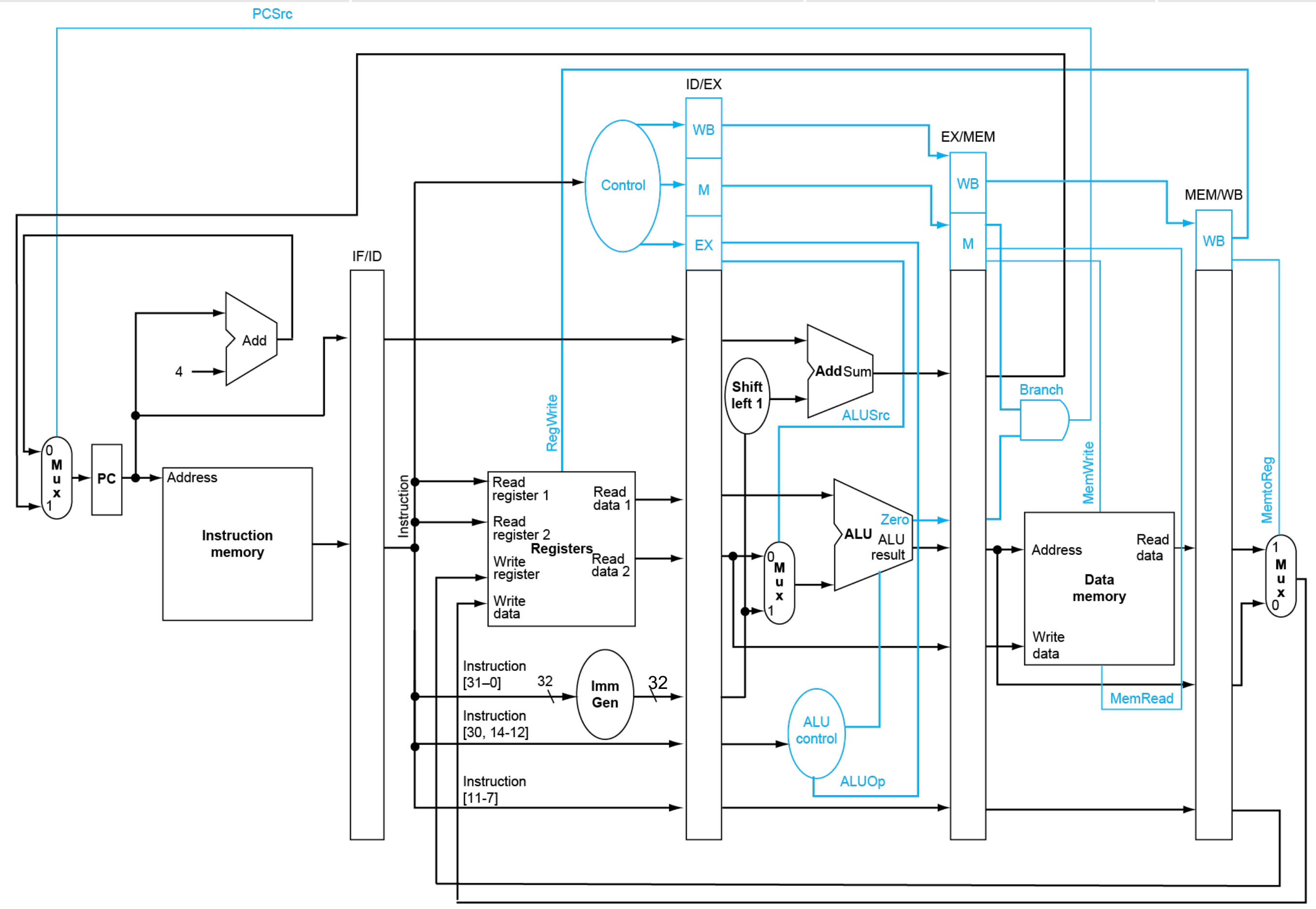
What’s happening in each stage?

add x14,x5,x6 | lw x13,24(x20) | add x12,x6,x7 | sub x11,x6,x7 | lw x10,20(x20)



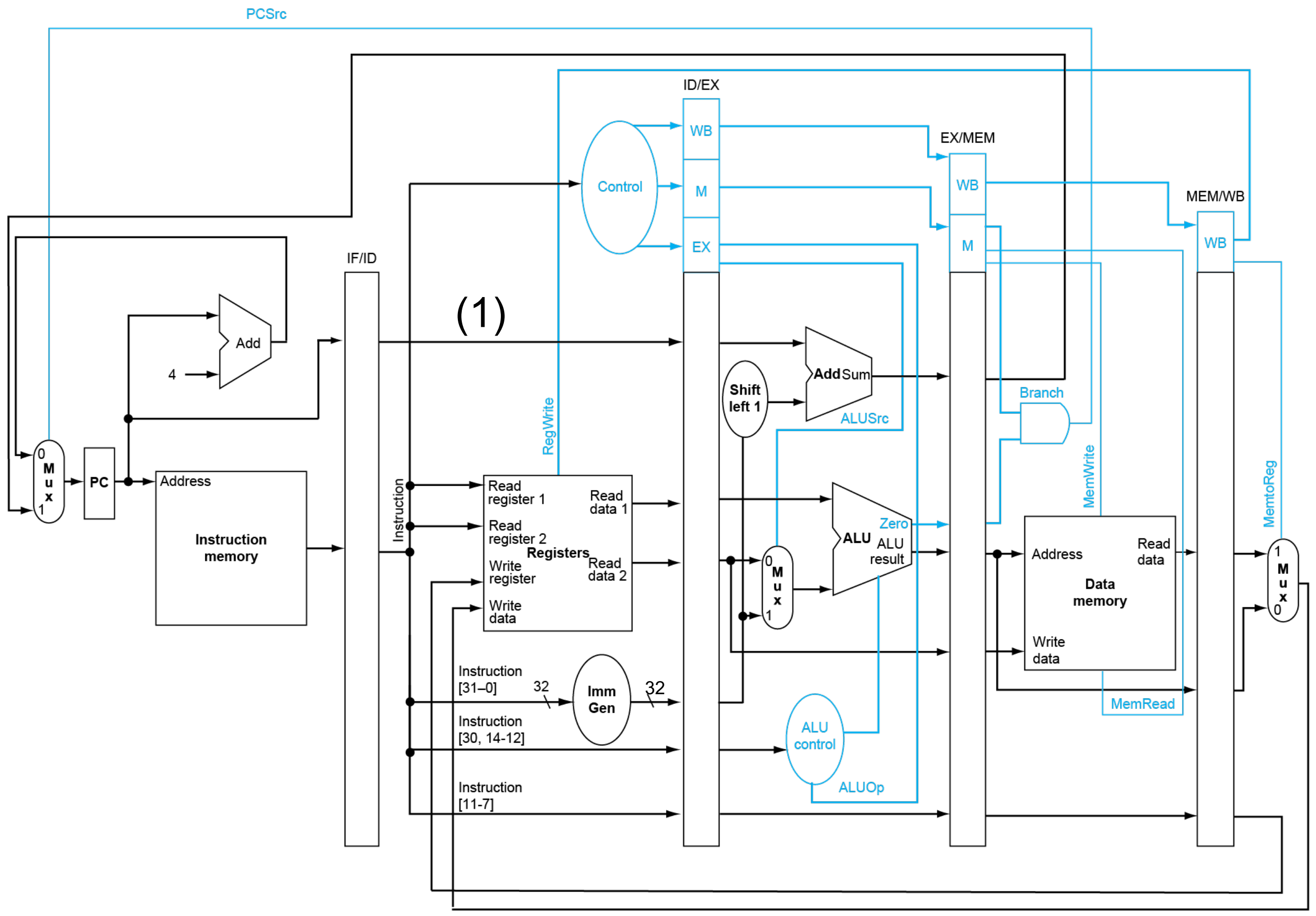
- add x14, x5, x6 # IF

PC Src	PC	IM Output	MUX0	MUX1



lw x13, 24(x20)   # ID

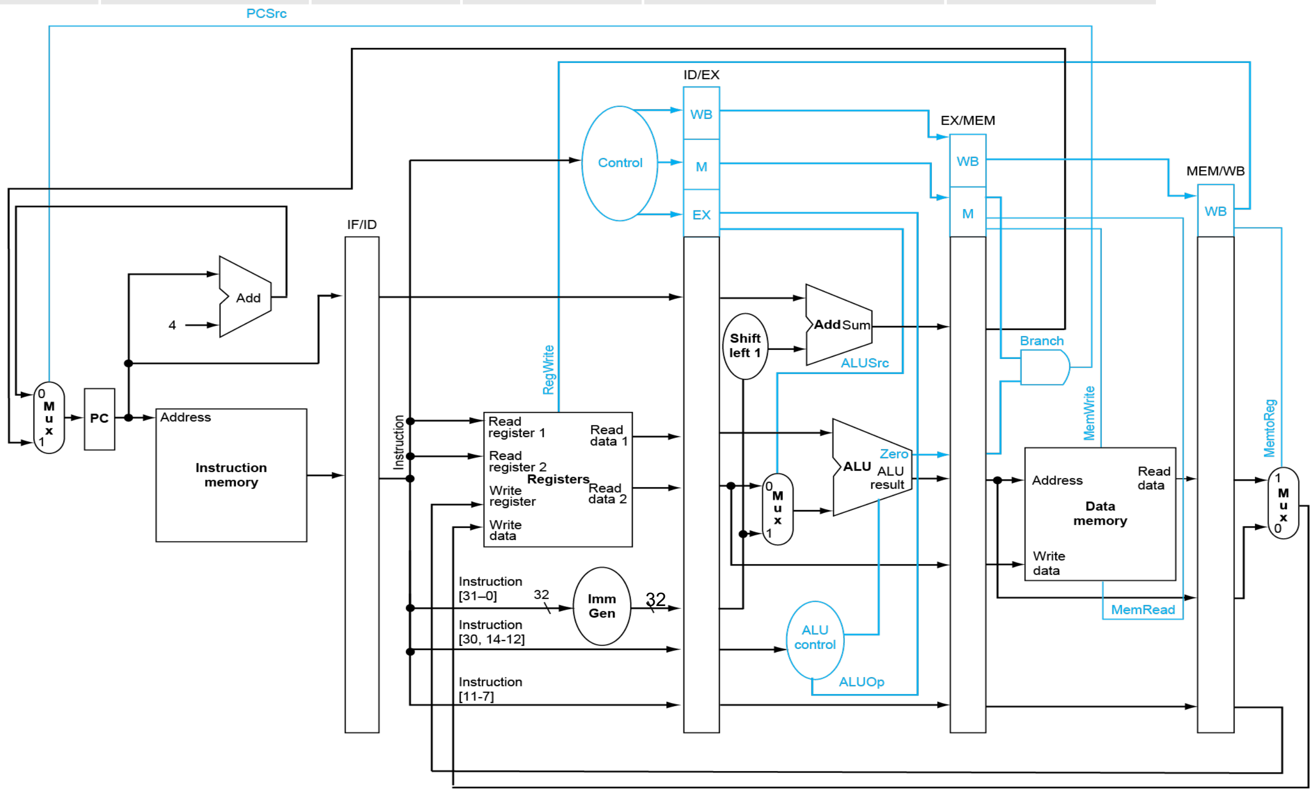
Reg Write	Rd reg1	Rd reg2	Wr reg	Wr Data	Imm Gen out	rd	(1)



add x12, x6, x7 # EX

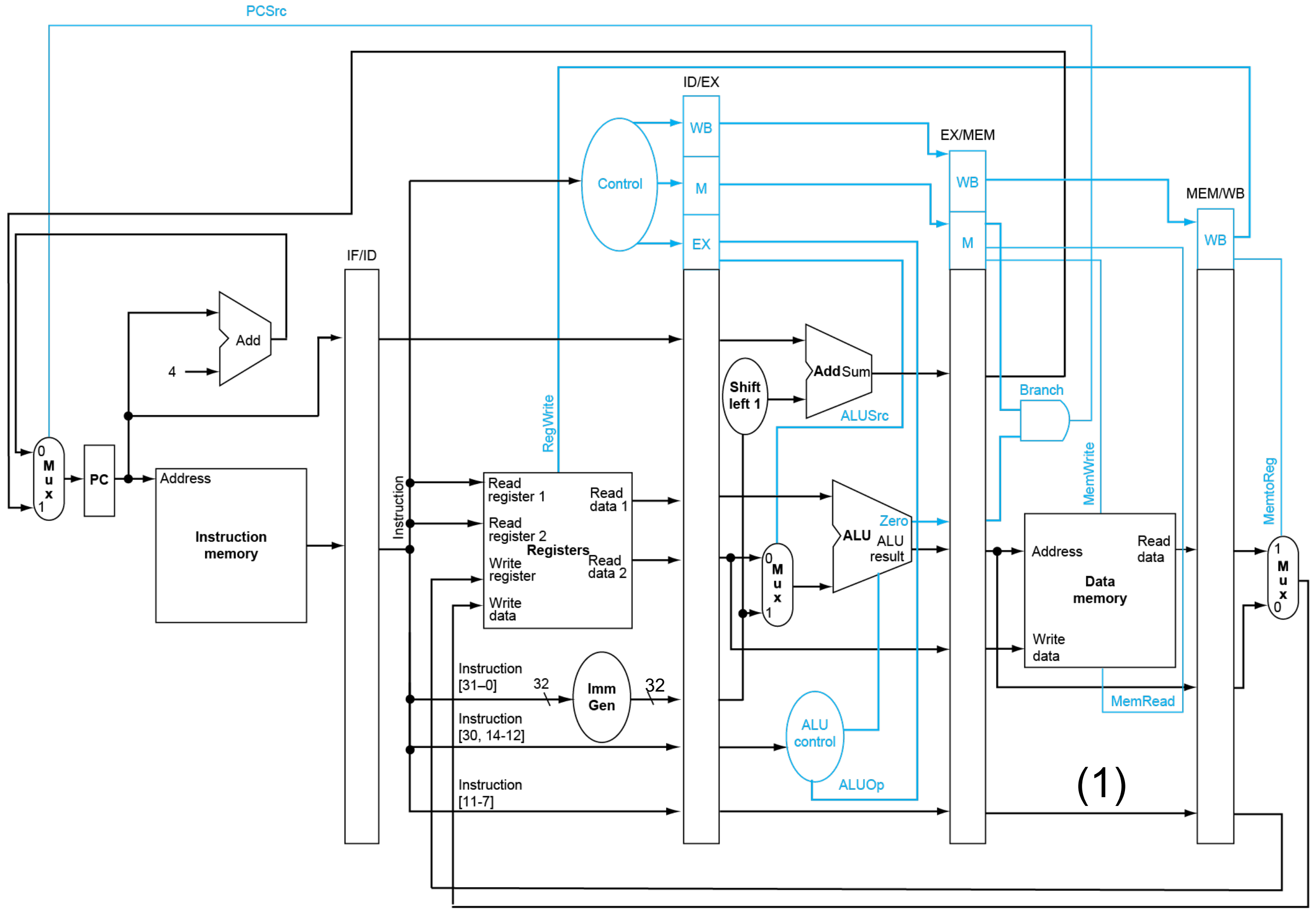
ALU Src	ADD A	ADD B	ALU MUX0	ALU MUX1	ALU A	ALU out	Zero	Mem Write	Mem Read

ALU Ctrl in	ALU Op	ALU Ctrl out	rd	Reg Write	MemtoReg	Branch



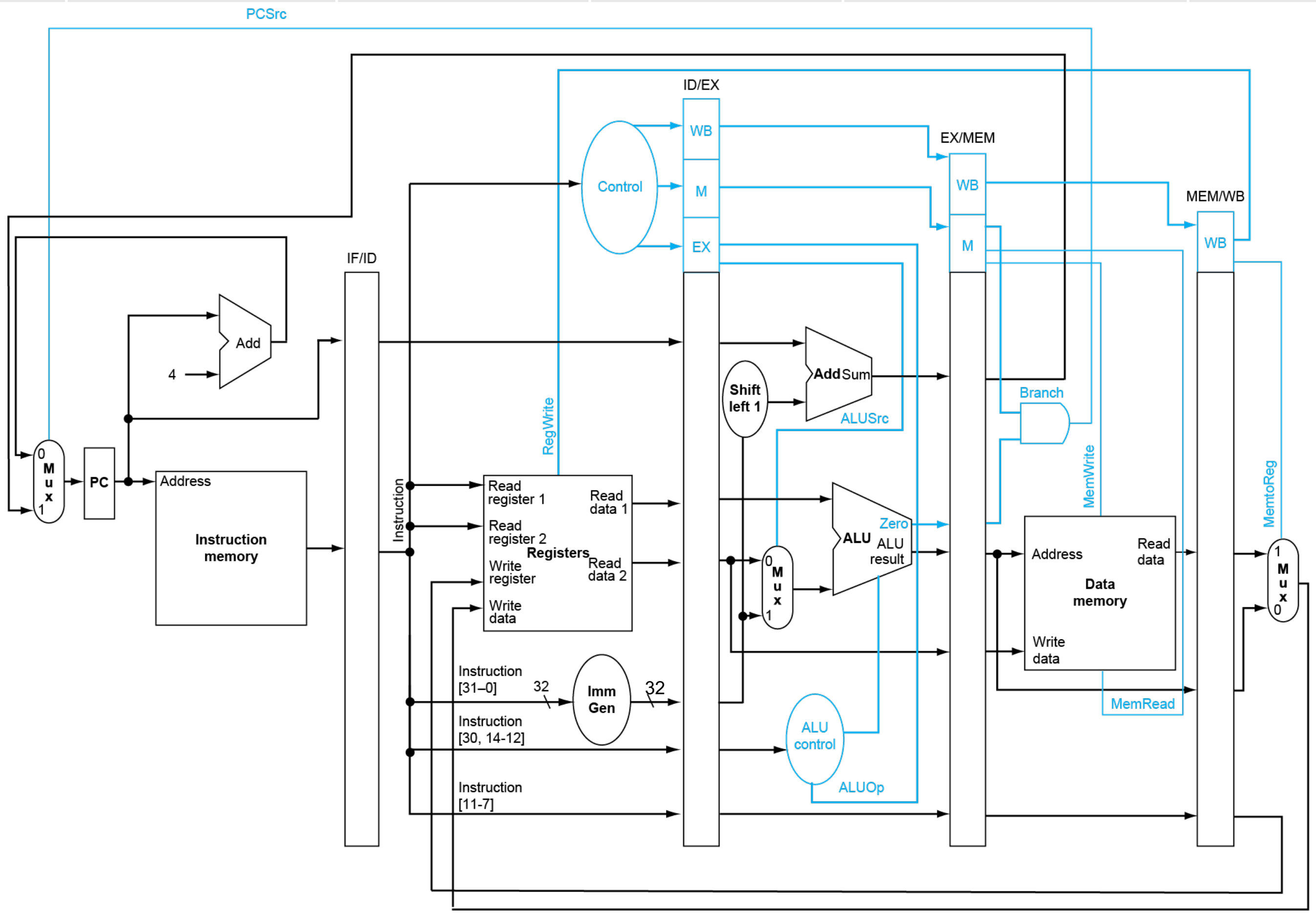
sub x11, x6, x7 # MEM

Addr	Write data	Read data	Mem Write	Mem Read	Zero	Branch	Memto Reg	Reg Write	(1)



lw x10, 20(x20) # WB

MUX0	MUX1	Write register	Write data	MemtoReg	Reg Write



Pipeline Summary

The BIG Picture

- 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

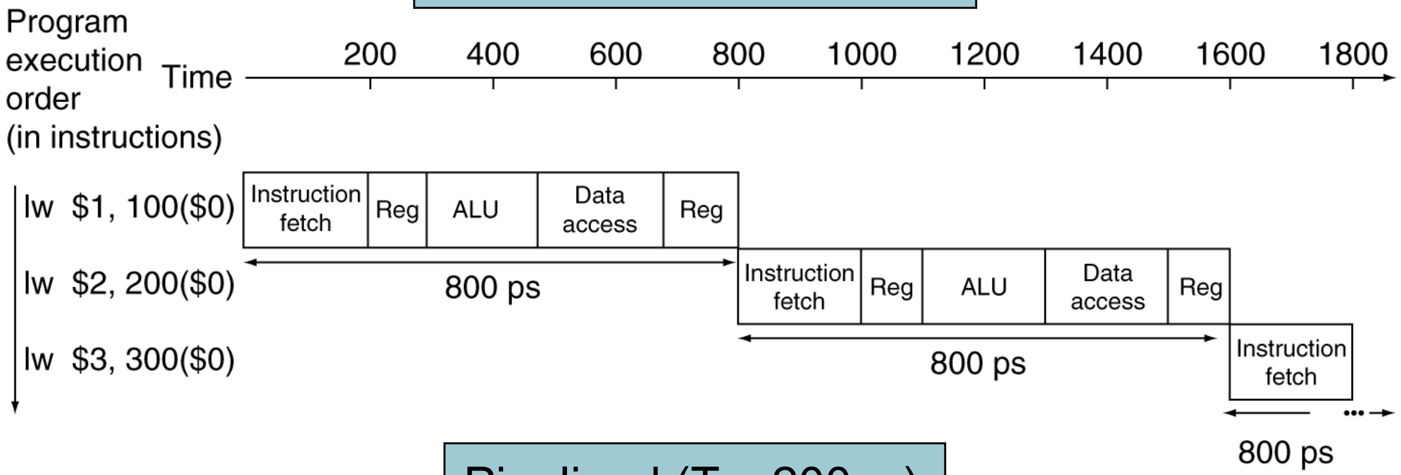
Pipeline Performance

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

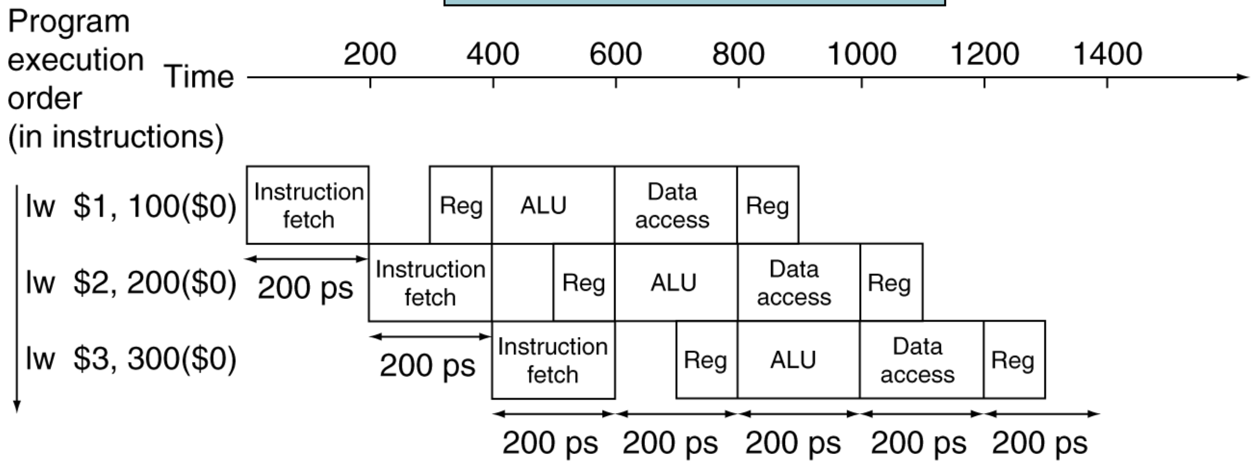
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Performance Consideration

- Assume 100 instructions are executed
  - 30% are loads
  - 15% are stores
  - 40% are R format instructions
  - 15% are branches
- Execution time using pipelined processor?

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{\text{pipelined}}$

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

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