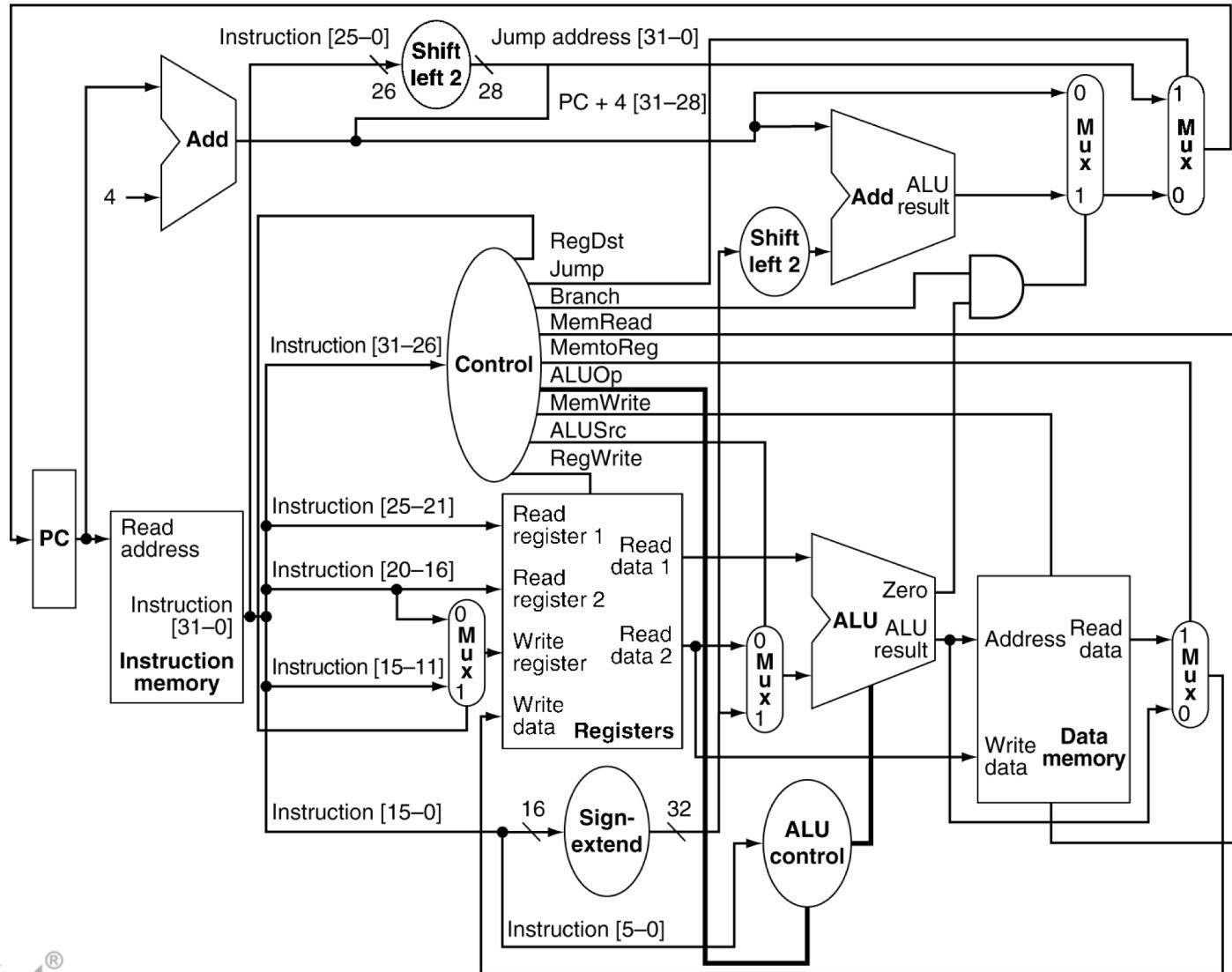




Topic 7

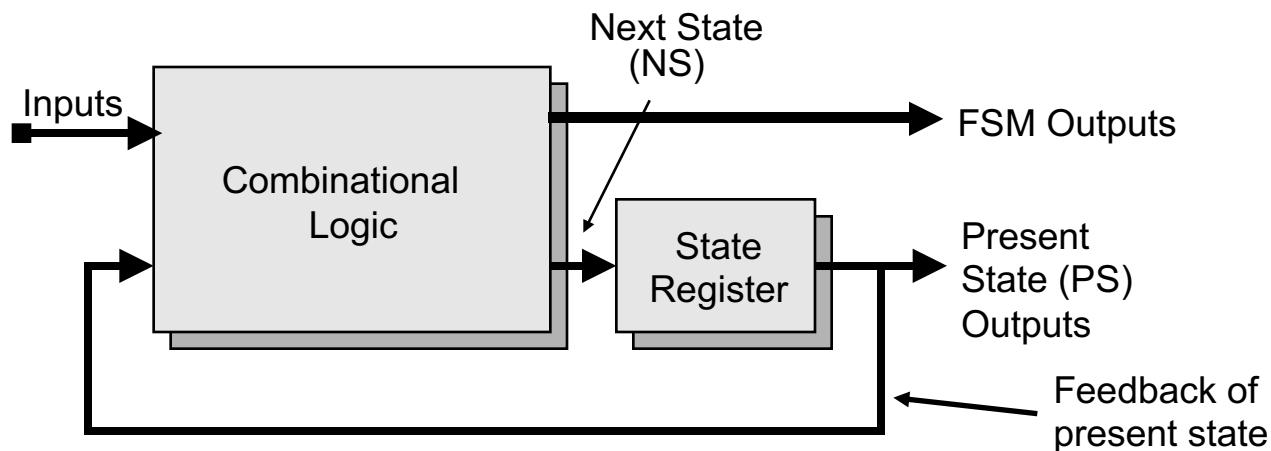
Pipelined Processor

Single Cycle Implementation



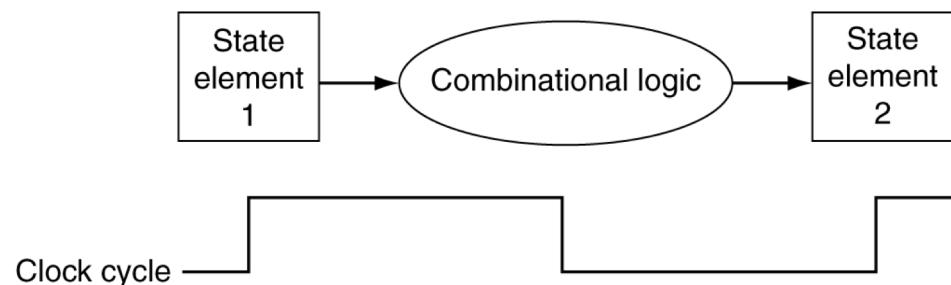
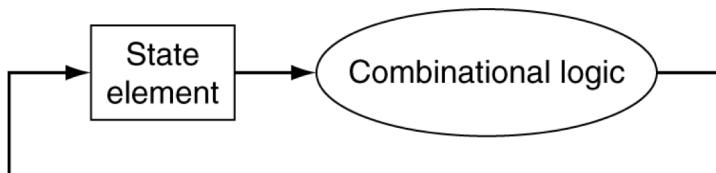
State Machine

- Finite State Machine

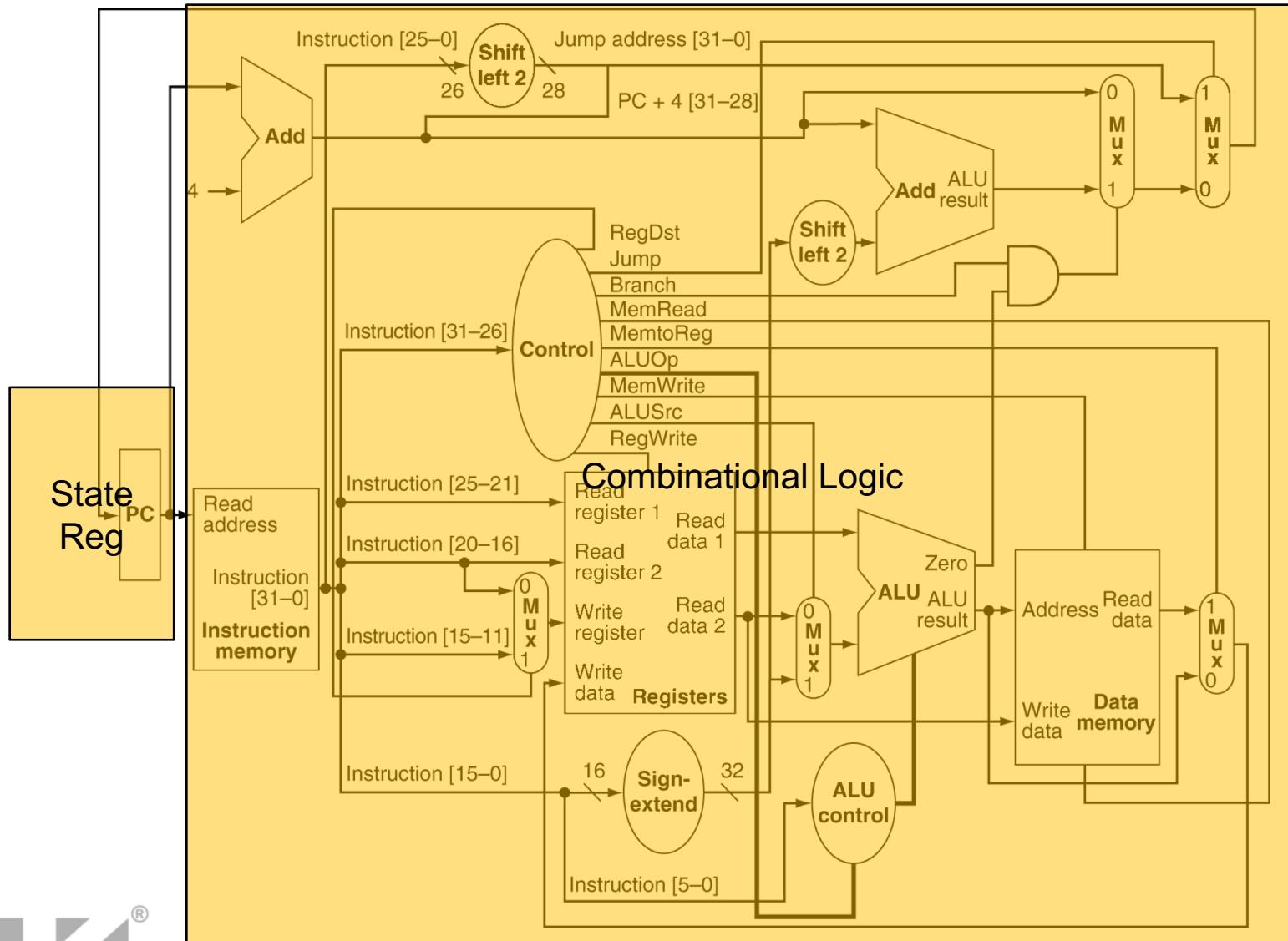


Clocking Methodology for FSM

- 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



Single Cycle Implementation



Performance Issues

- Longest delay determines clock period
 - Critical path: load instruction
 - Instruction memory → register file → ALU → data memory → 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 time for major components is
 - 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

many operations are

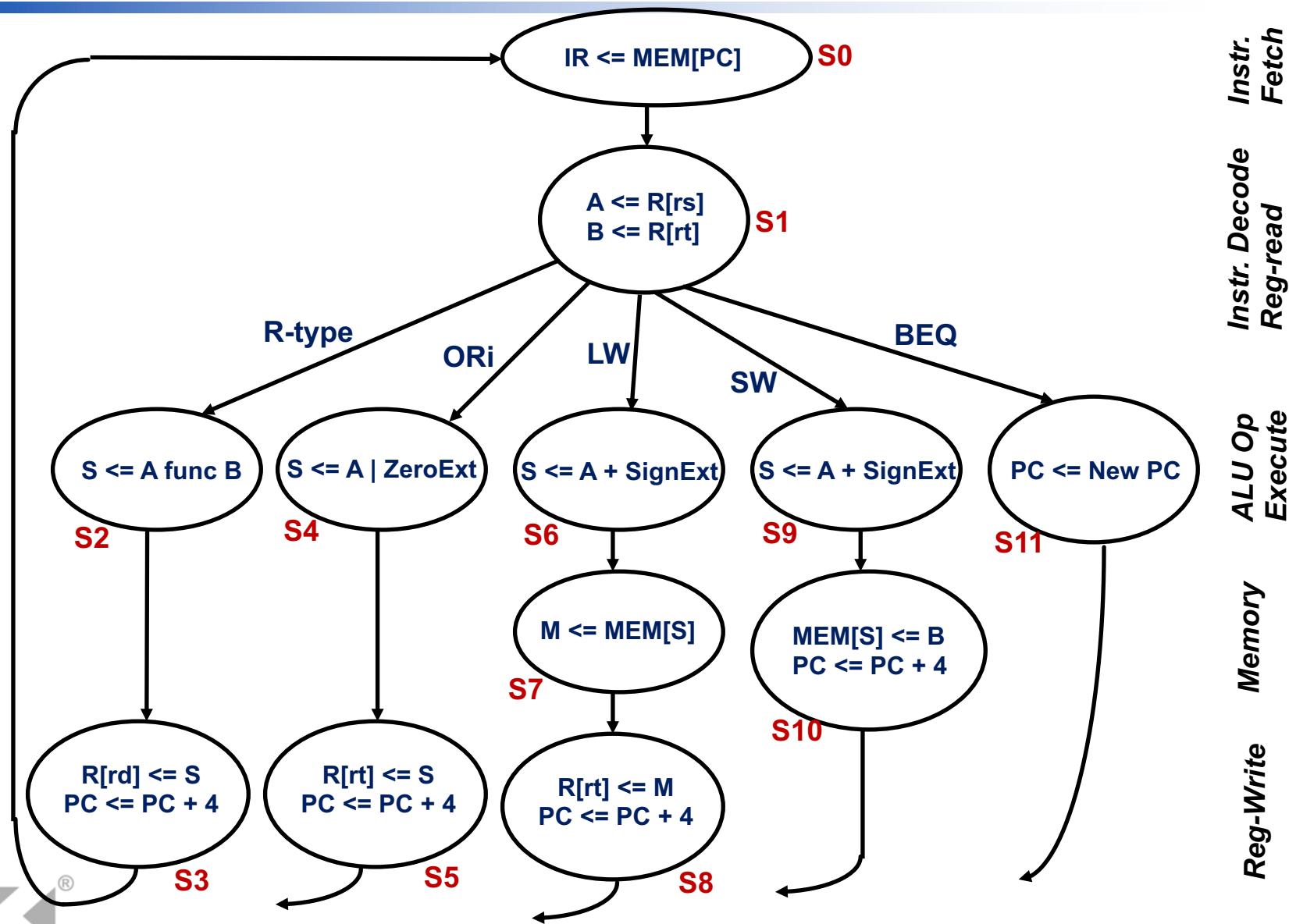
Performance Consideration

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

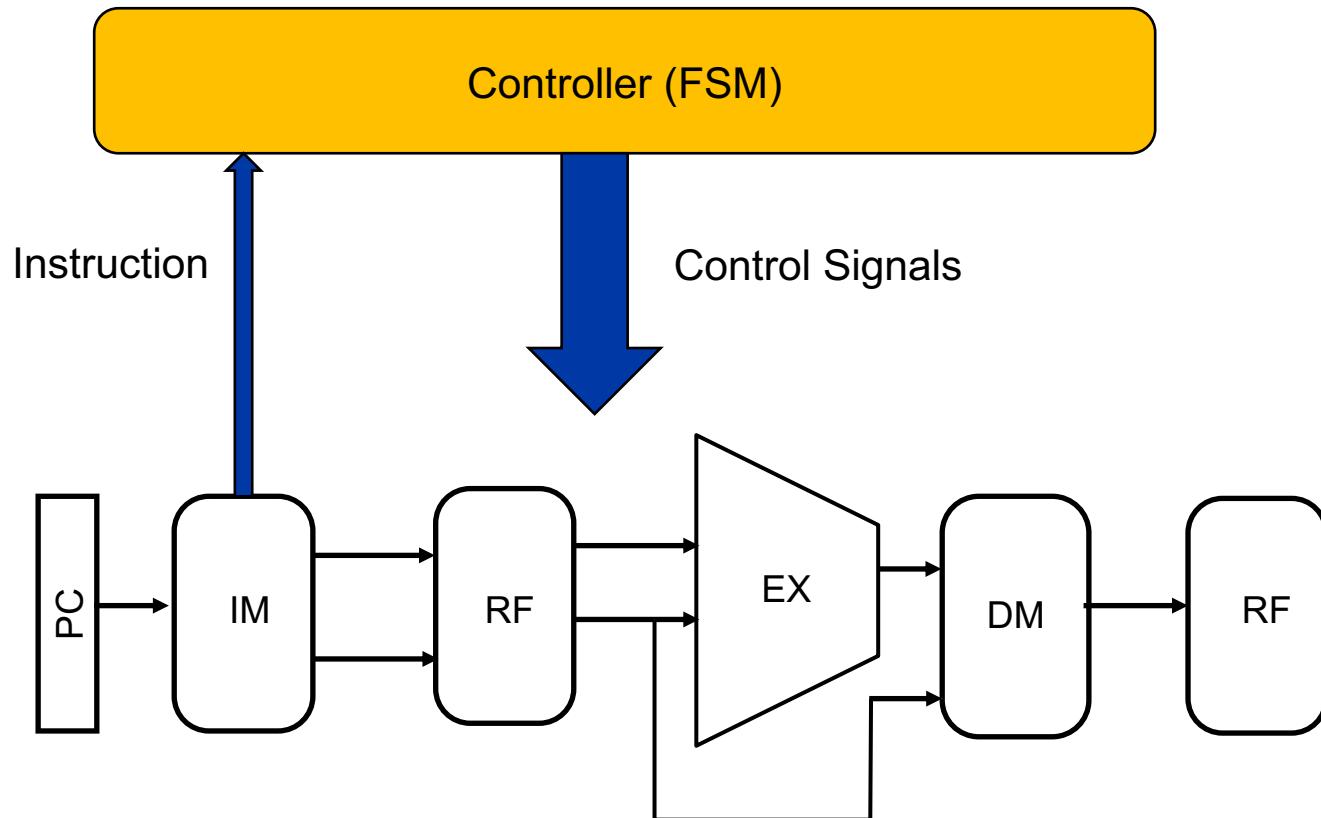
Improvement – Multi-cycle CPU

- Each instruction takes multiple cycles to execute
 - Clock cycle time is reduced
 - Different instructions take different clock cycles to complete
 - Slower instructions take more cycles
 - Overall execution time is shorter

Multi-cycle CPU – FSM



Multi-cycle Implementation



Performance Consideration

- Assume 100 instructions are executed
 - 30% are loads
 - 15% are stores
 - 40% are R format instructions
 - 15% are branches

Instr.	State	Instr. fetch	Register read	ALU op	Data Memory	Register write	Total# of cycles
lw		x	x	x	x	x	5
sw		x	x	x	x		4
R-format		x	x	x		x	4
beq		x	x	x			3

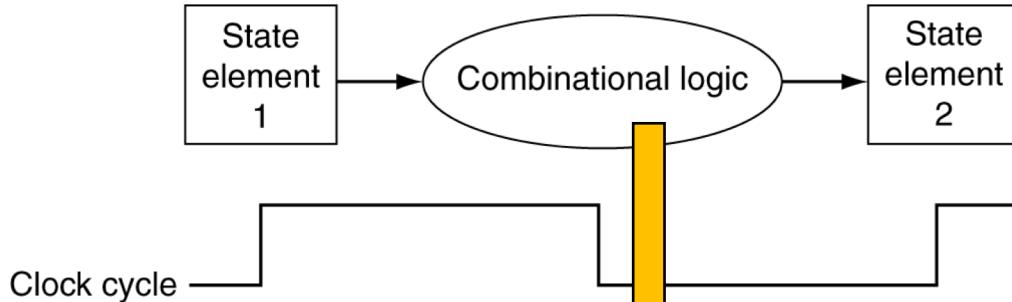
- How to decide clock cycle time?
- Execution time using multi-cycle processor?

Performance Issue

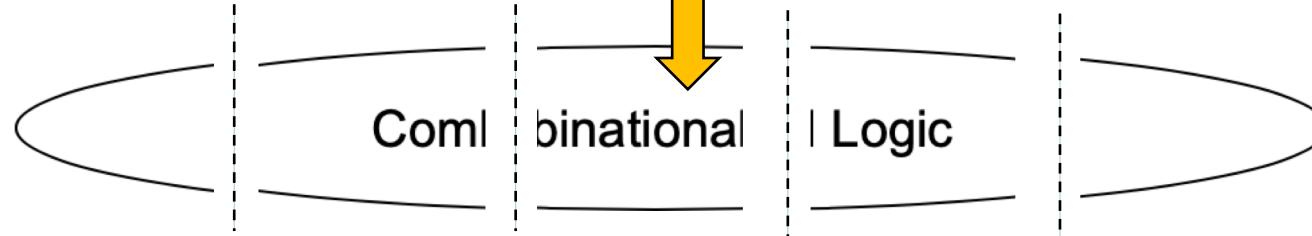
- Many components are still doing nothings and waiting – waste of time and resources

Improvement – Pipelining

- Divide the combination logic into smaller pieces



Single Cycle

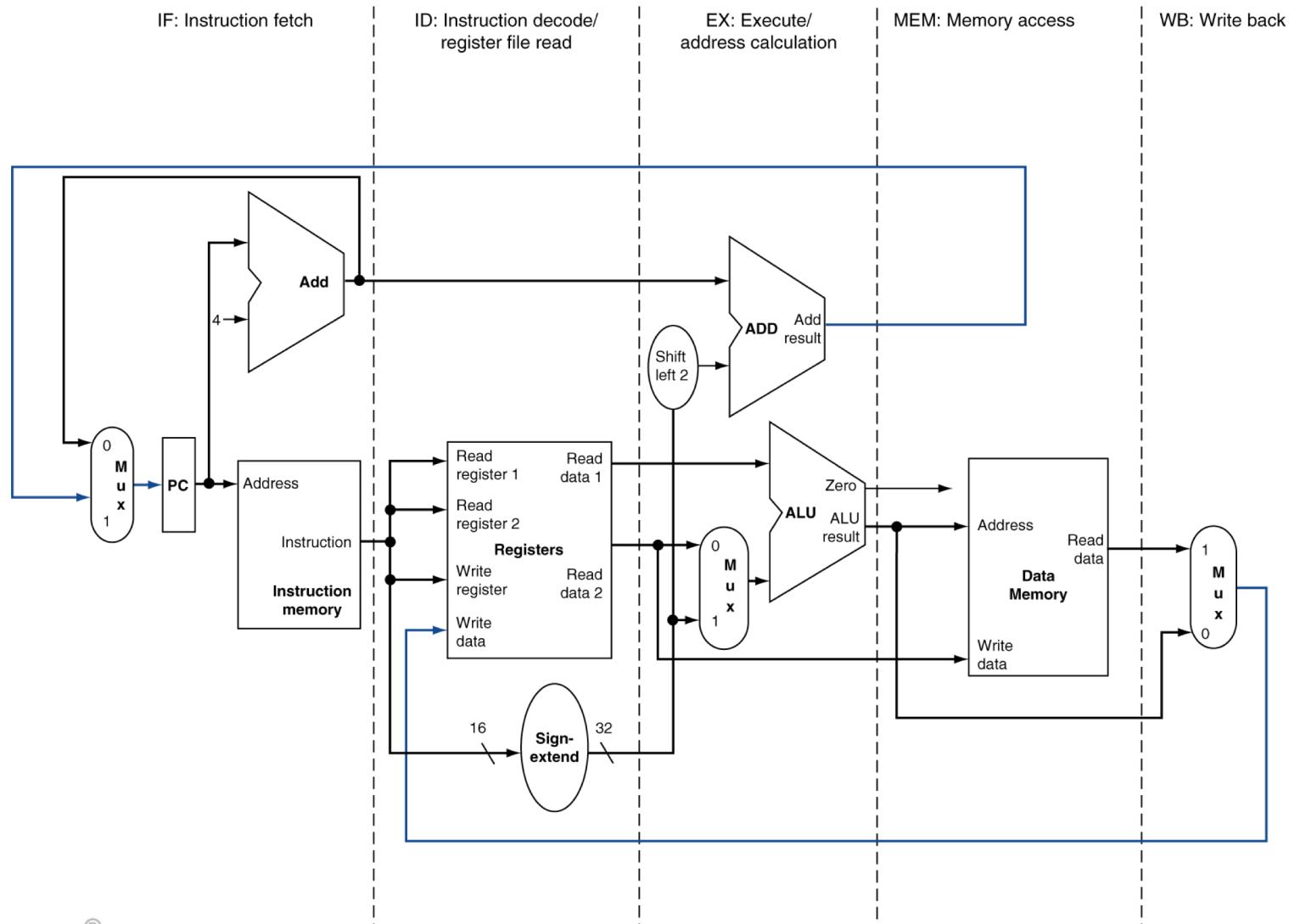


Pipelining



- Each piece is finished in shorter time

MIPS Pipelined Datapath

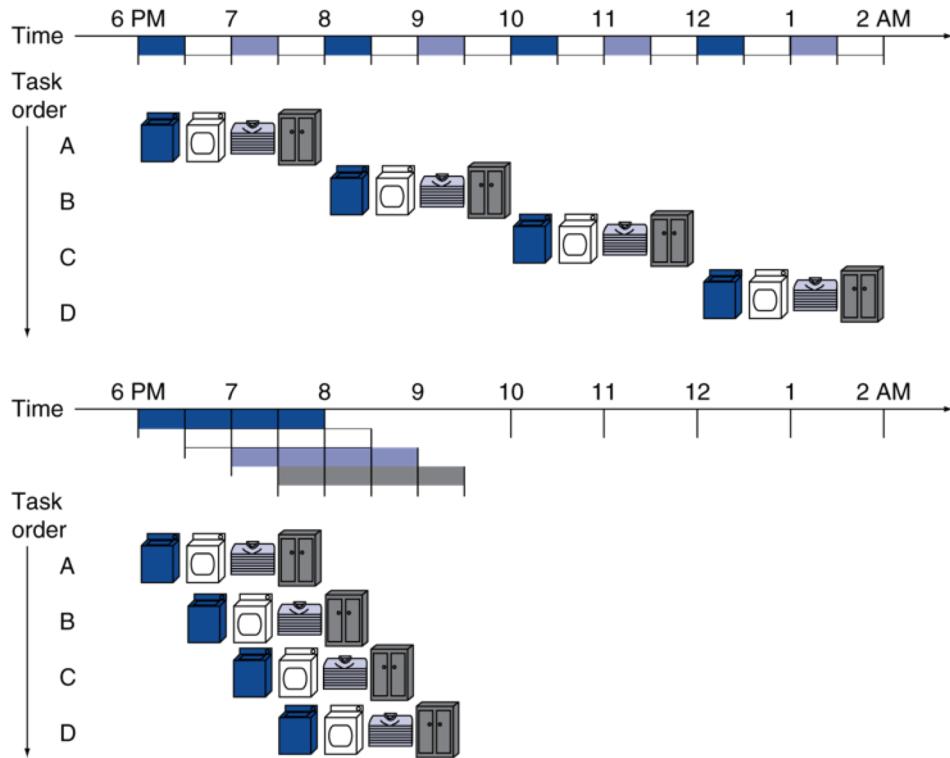


MIPS Pipeline

- 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

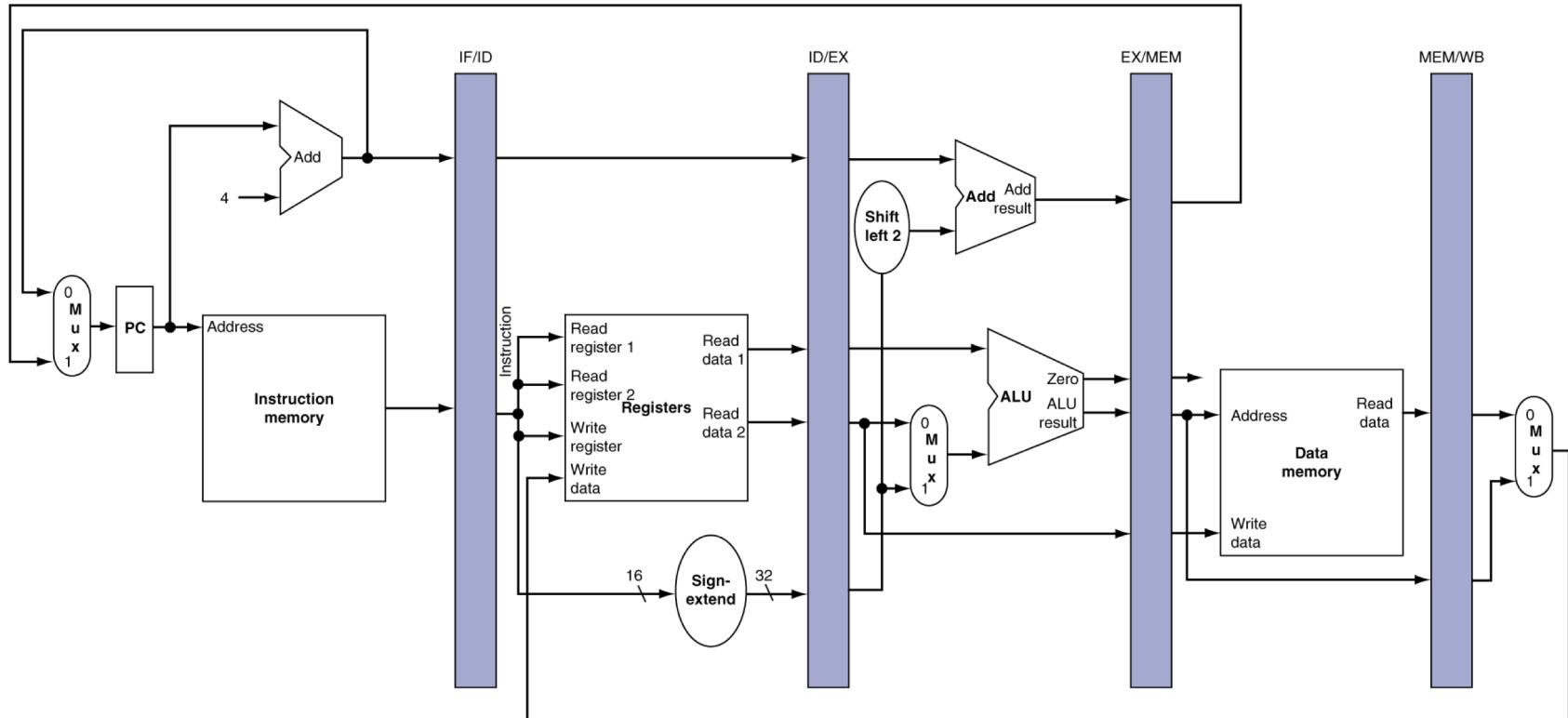
- Pipelined laundry: overlapping execution
 - Parallelism improves performance



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

Pipeline registers

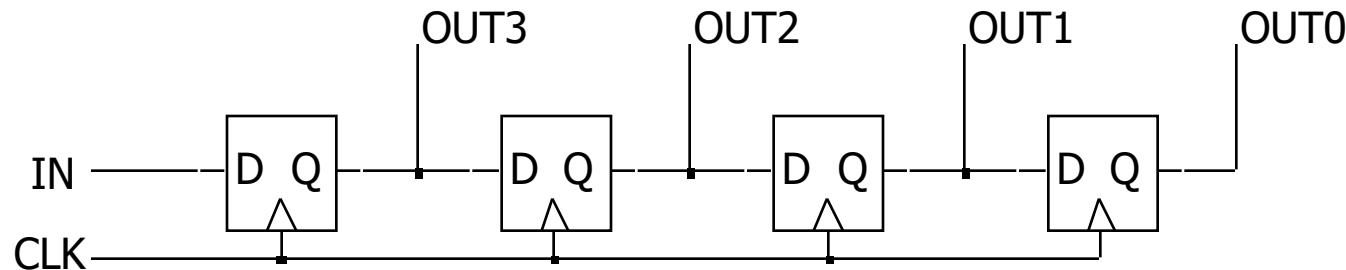
- Need registers between stages
 - To hold information produced in previous cycle



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

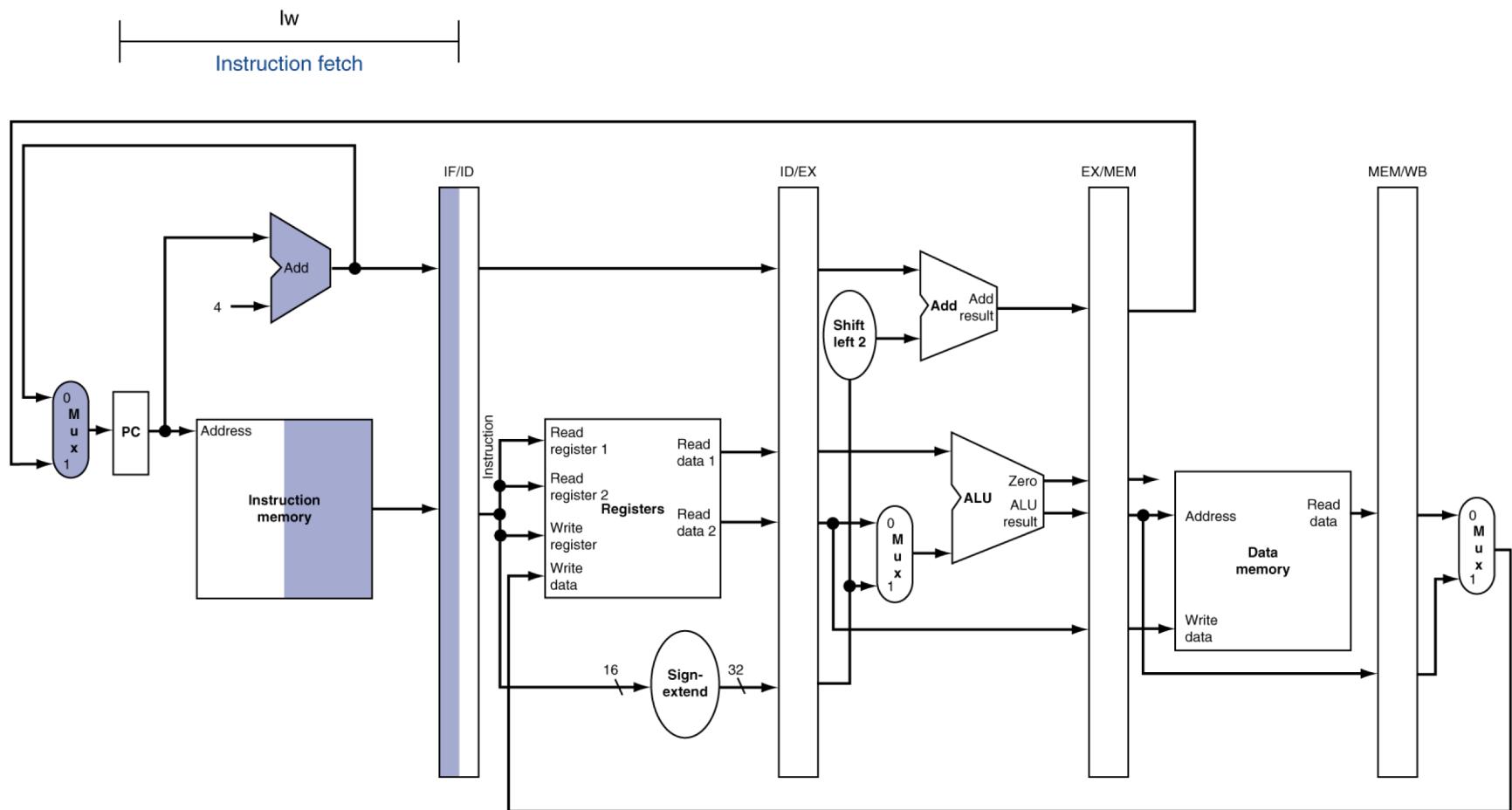


	IN	OUT(3:0)
Initial value:	0	0110
rising edge:	0	0011
rising edge:	0	0001
rising edge:	0	0000
rising edge:	1	1000
rising edge:	0	0100

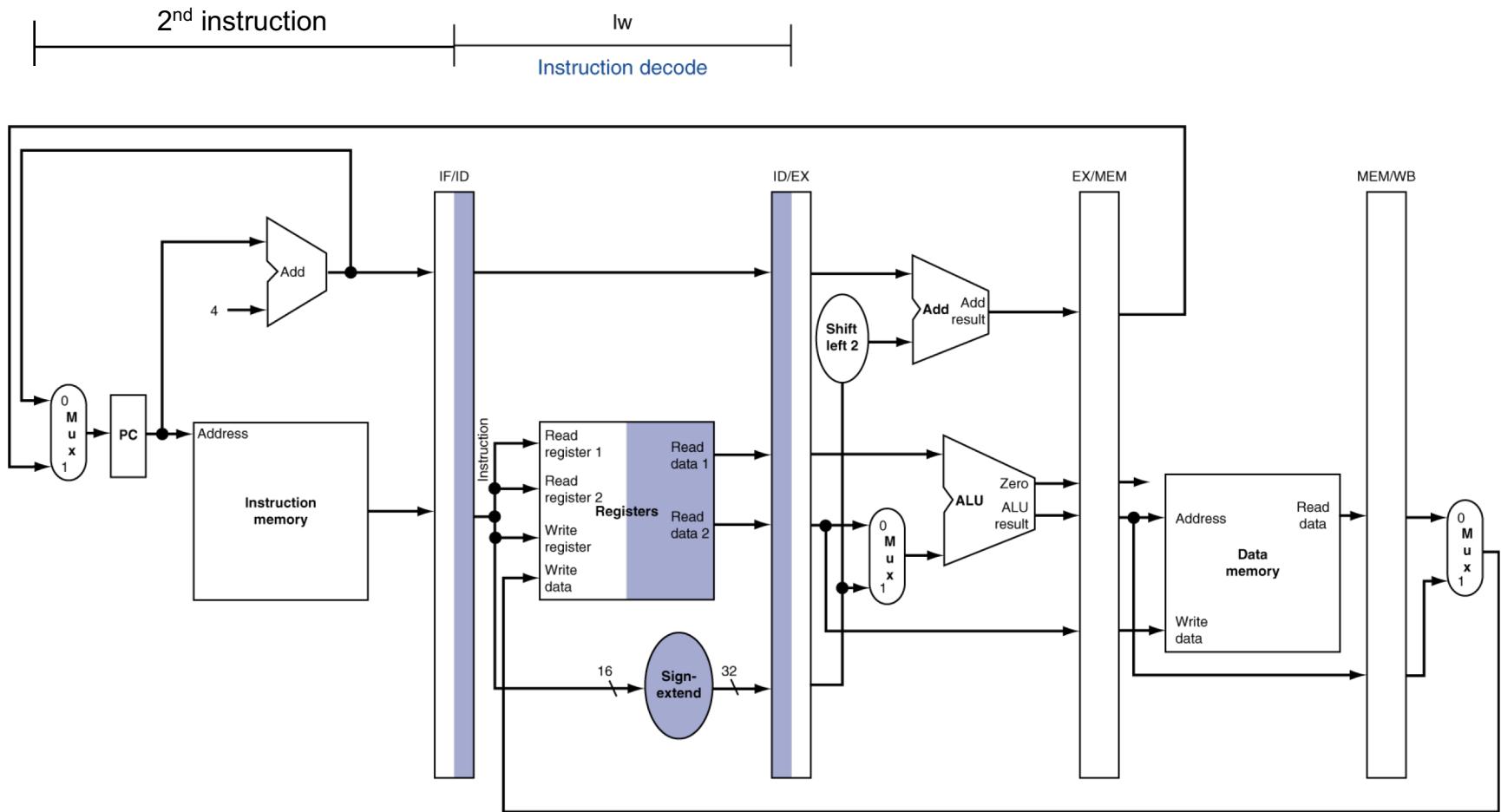
Pipeline Operation

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

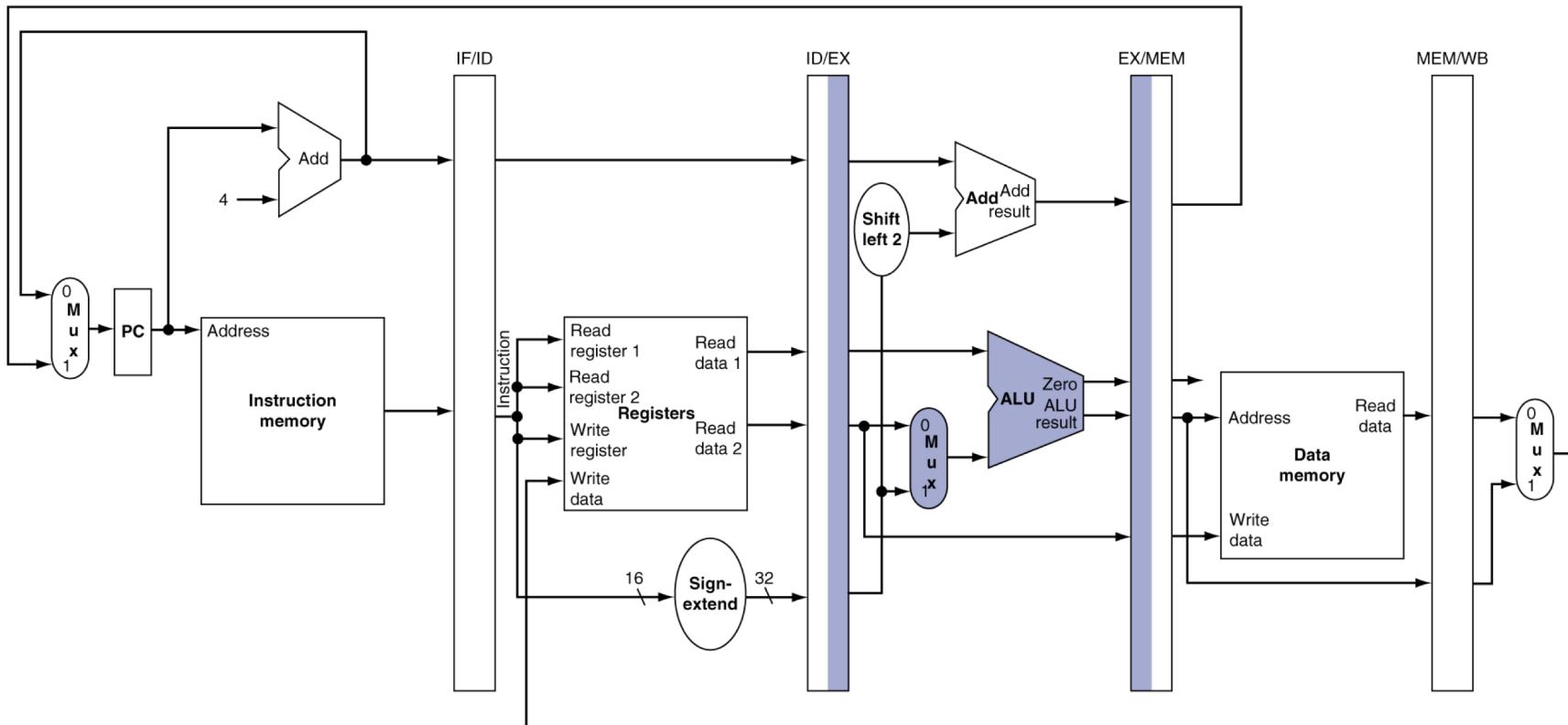
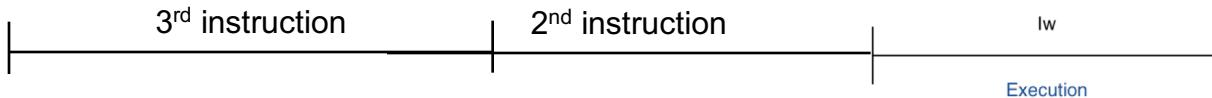
IF for Load, Store, ...



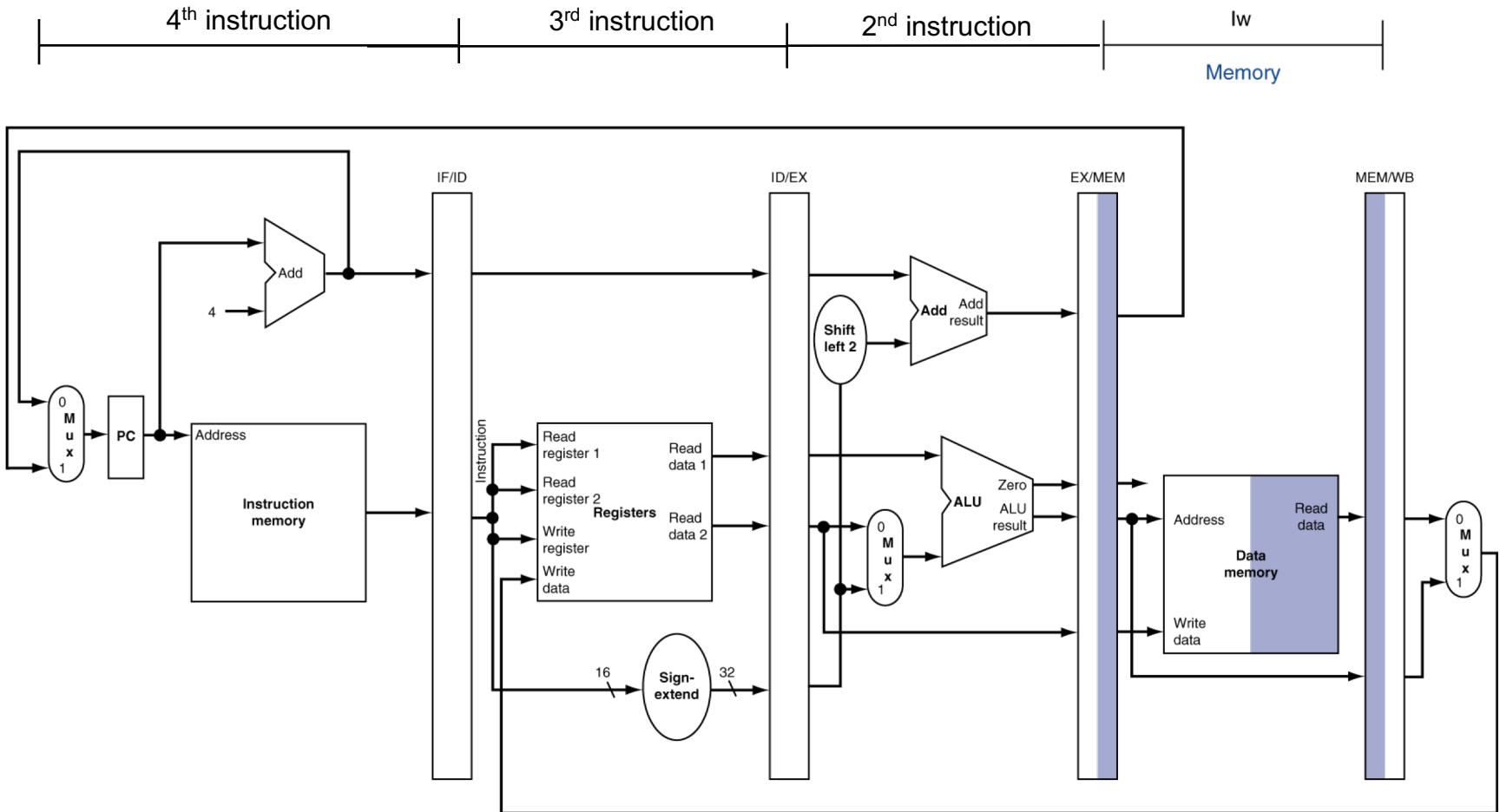
ID for Load, Store, ...



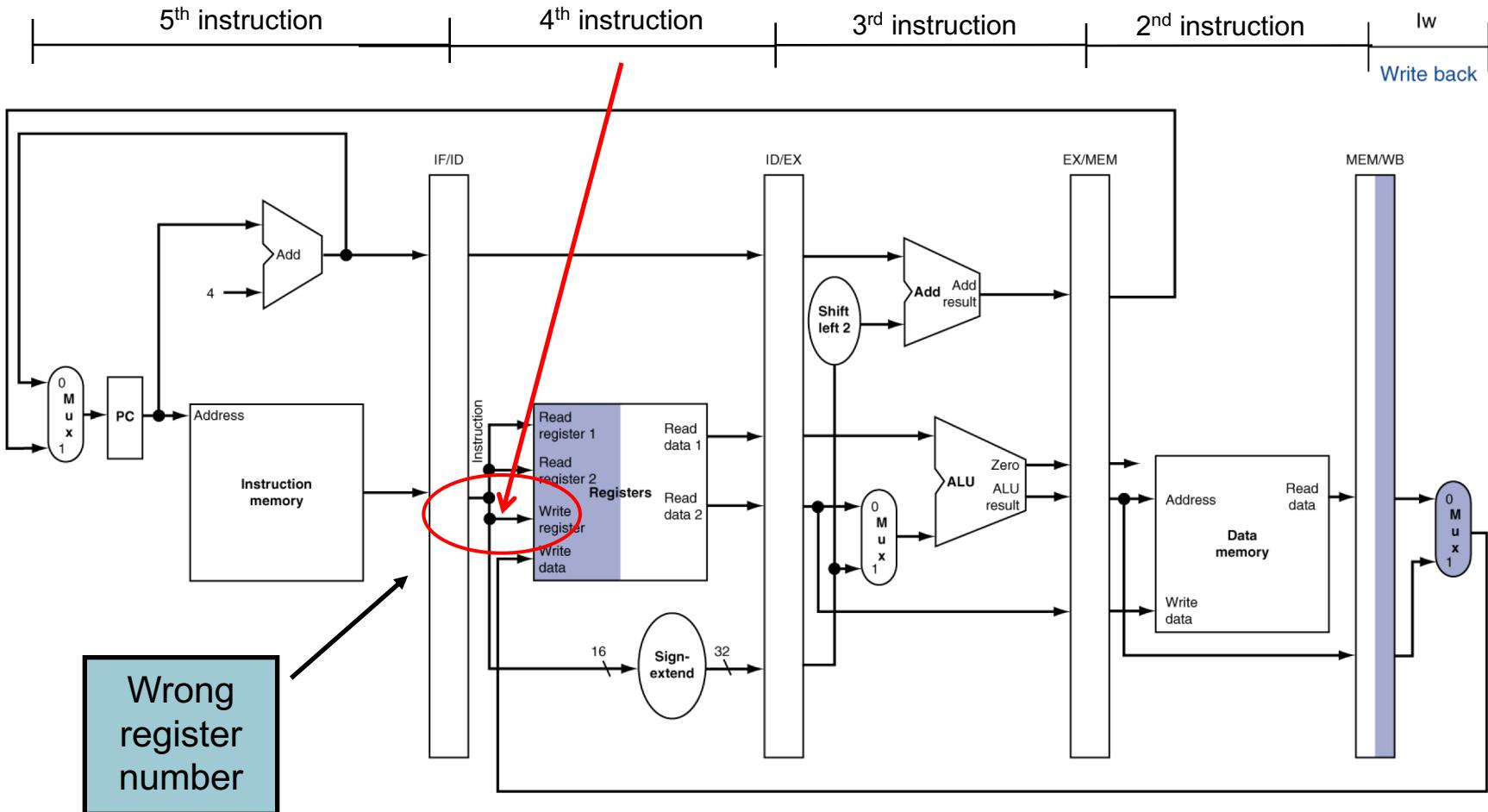
EX for Load



MEM for Load

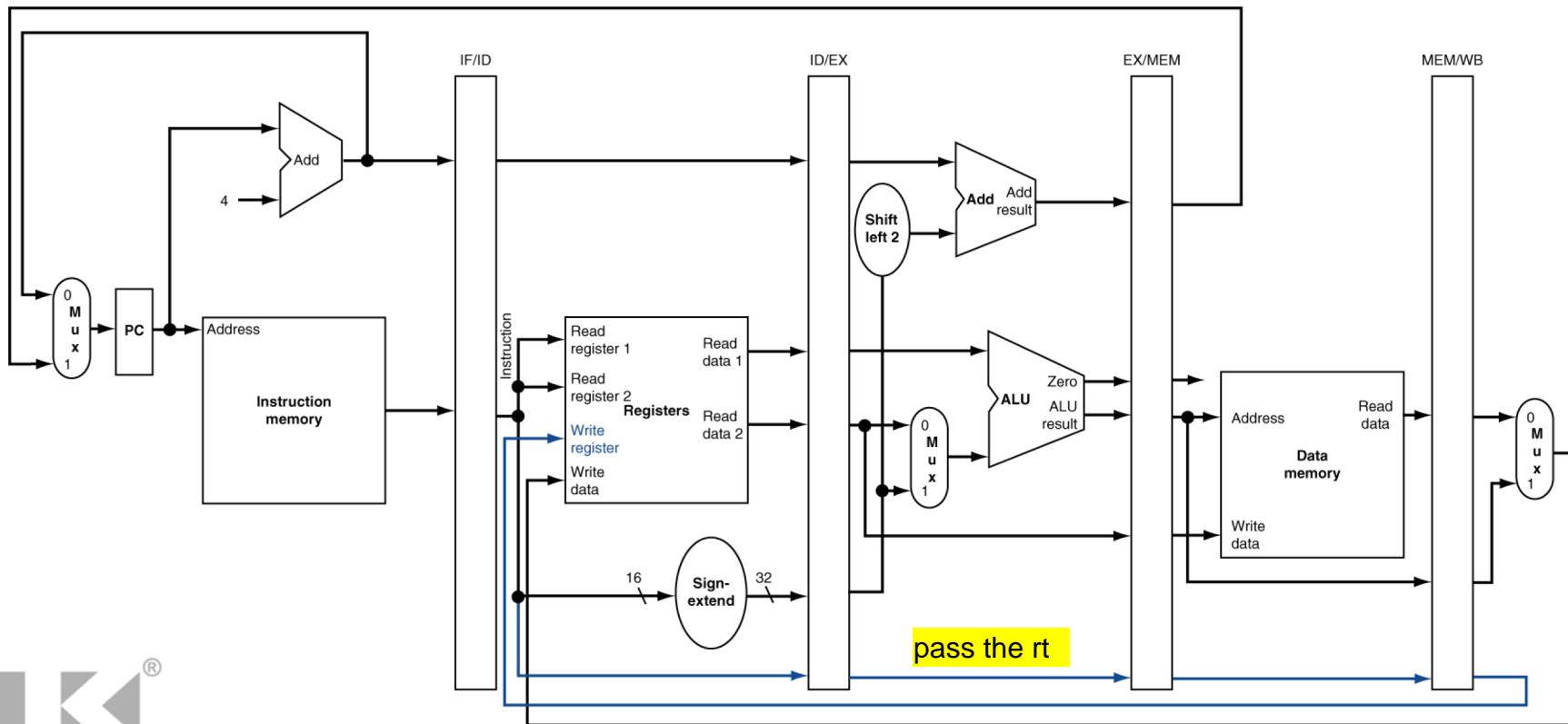


WB for Load

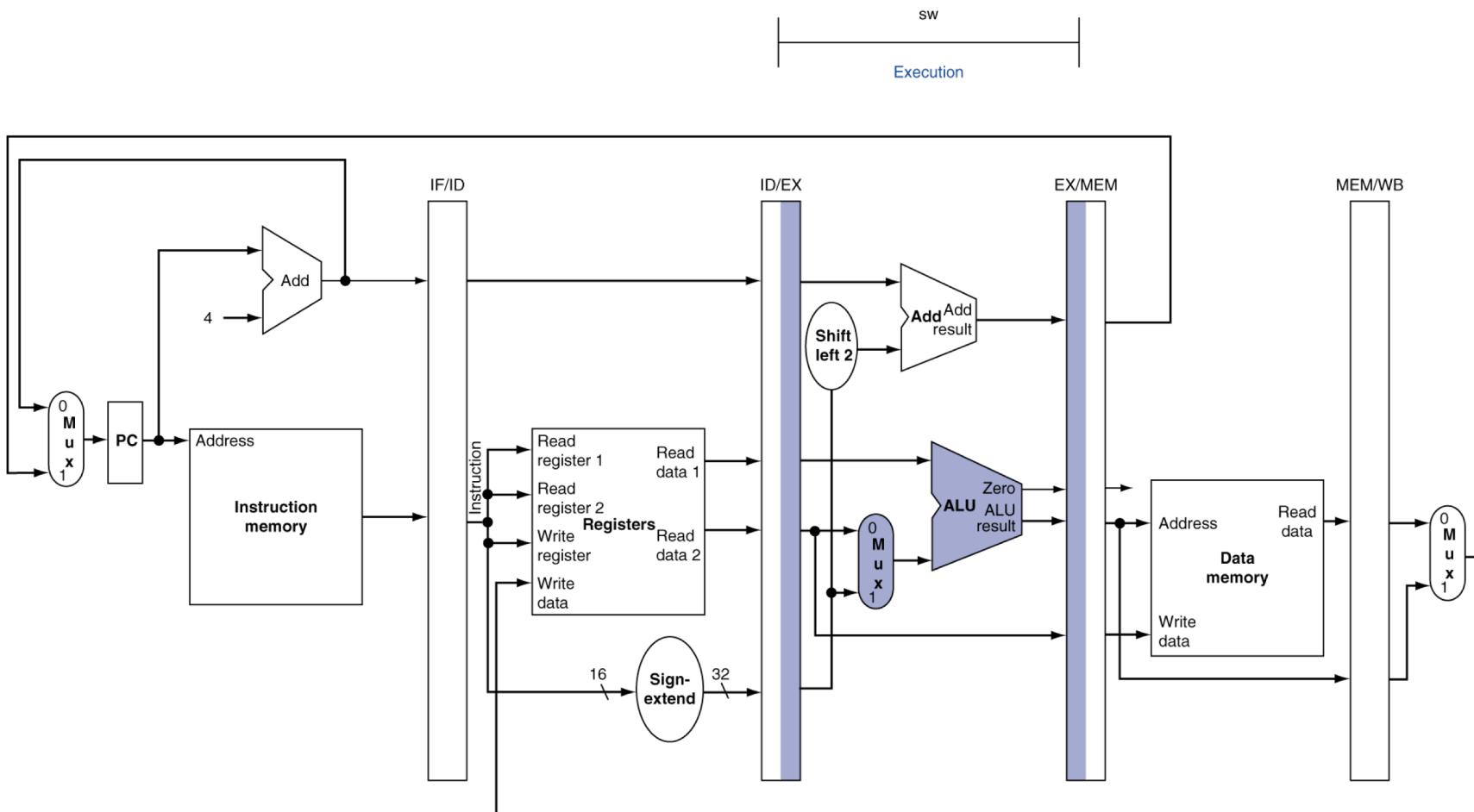


Corrected Datapath for Load

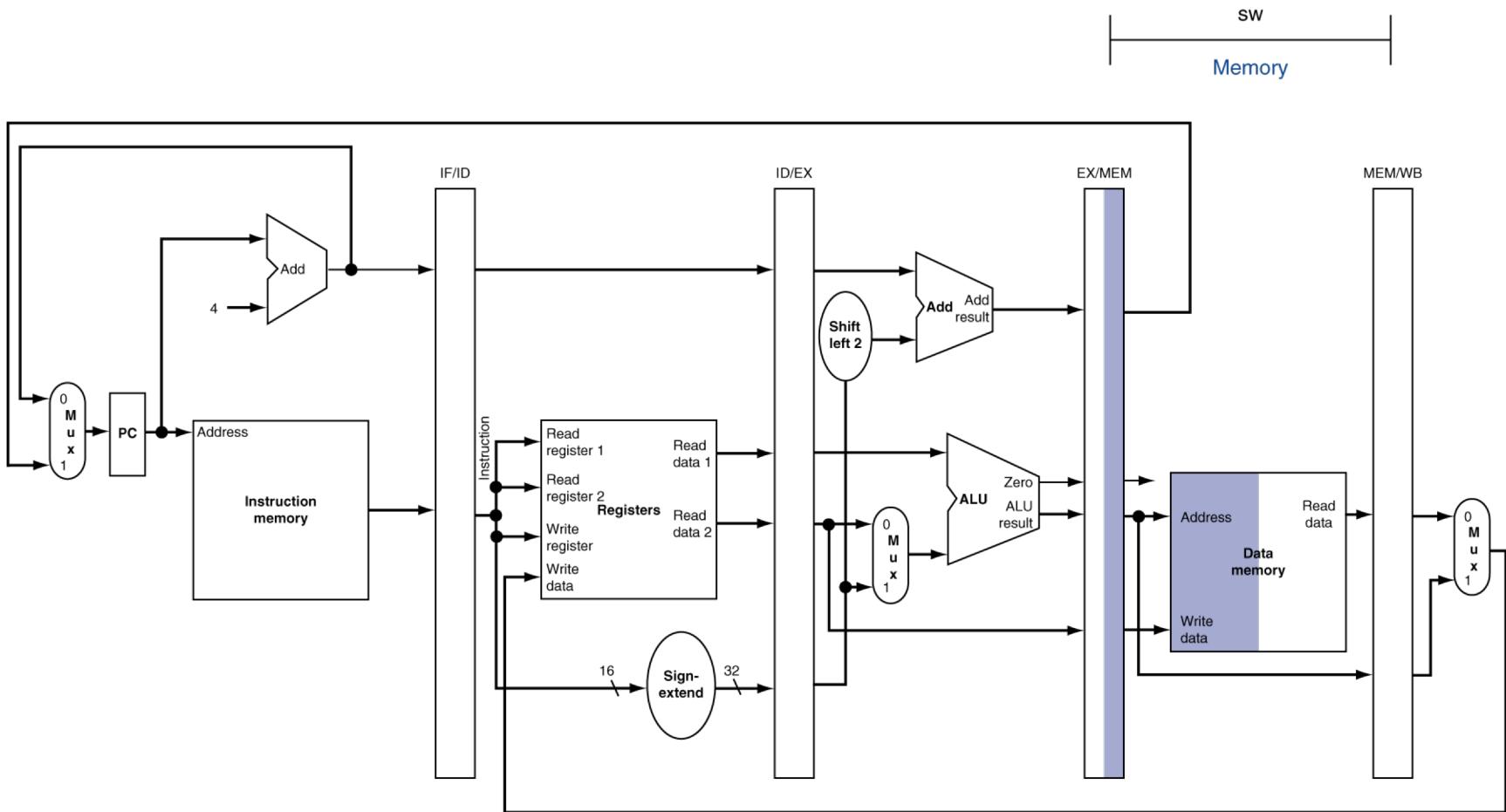
- Pass alive signals along through the pipeline
- Has to write/read register file at the same time
 - Writing reg in first half of clock
 - Reading reg in second half of clock



EX for Store



MEM for Store

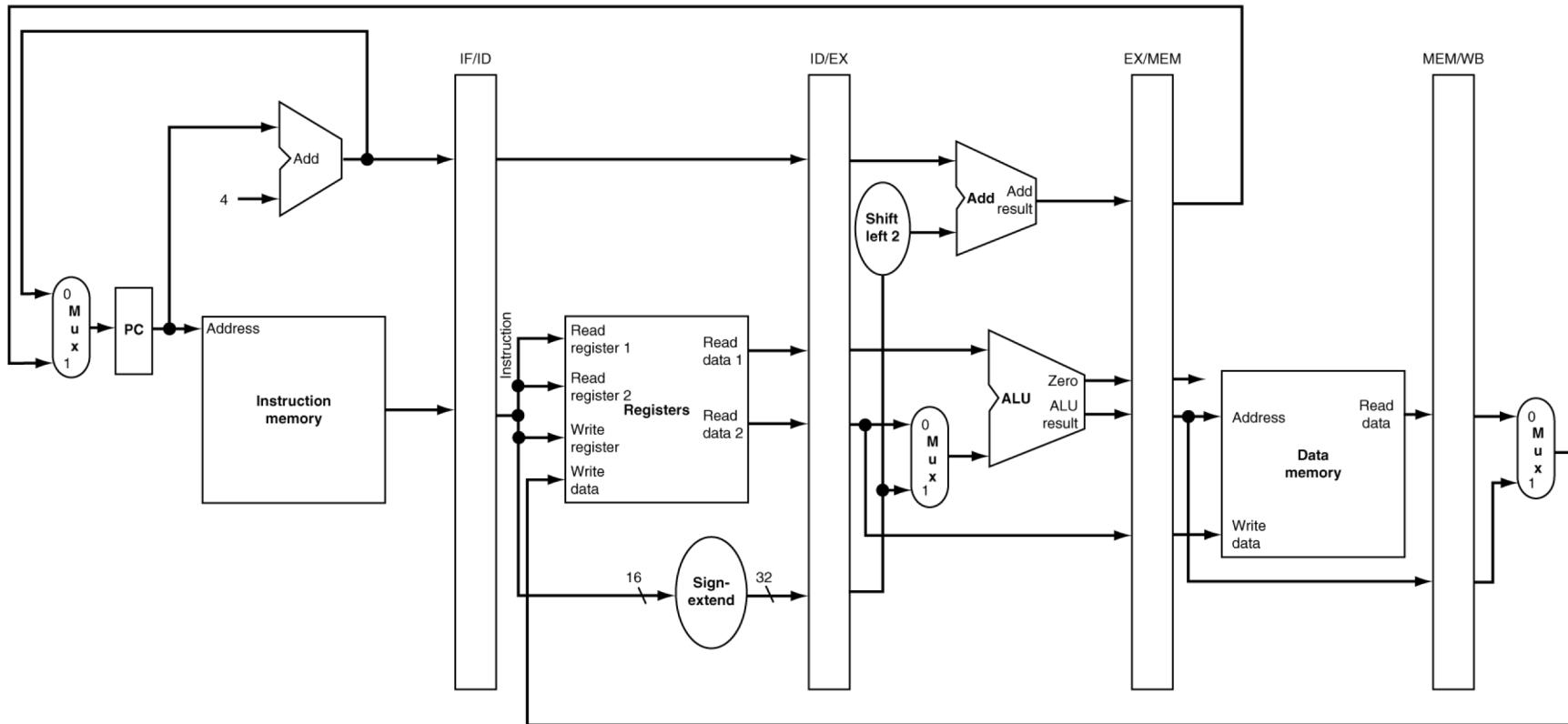


WB for Store

No operation

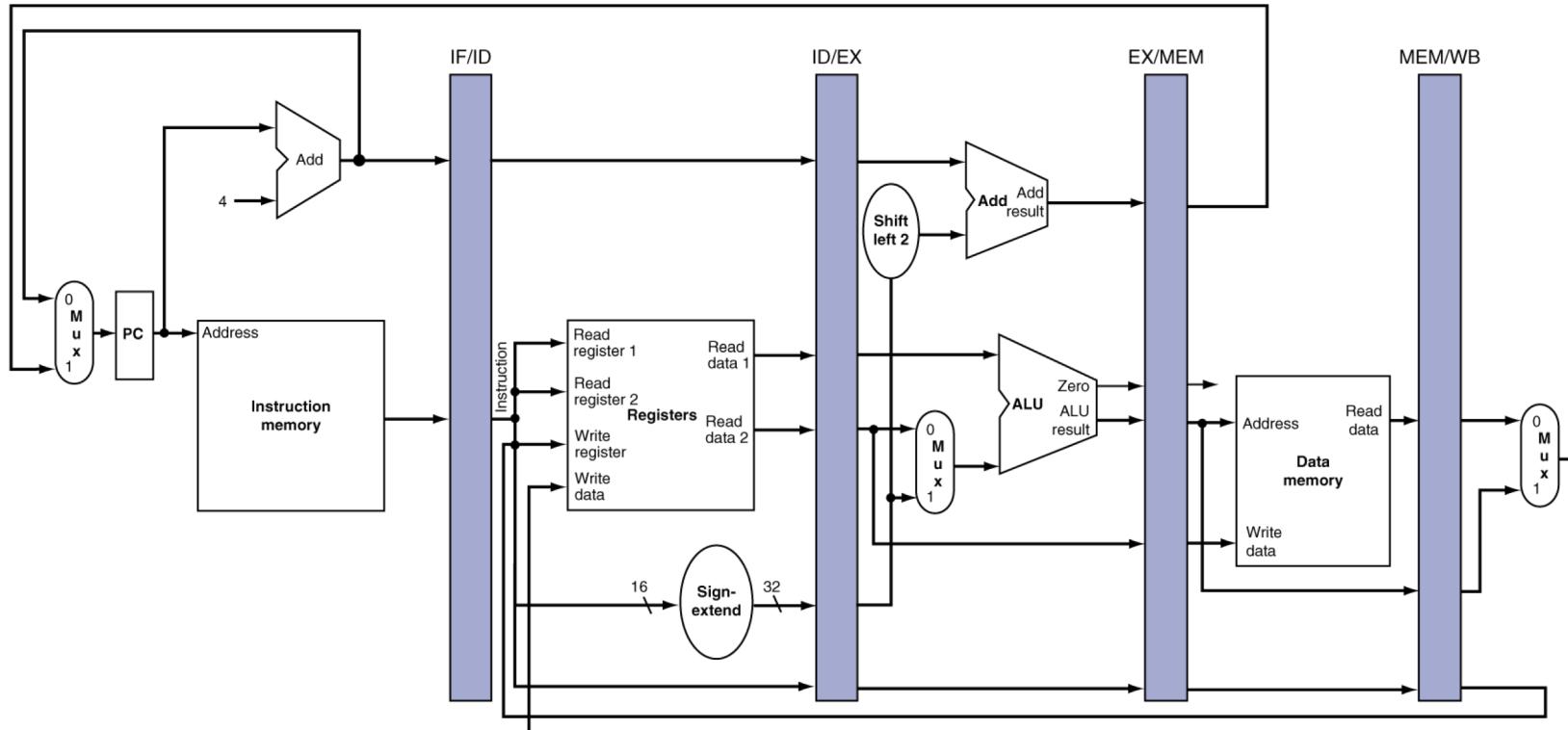
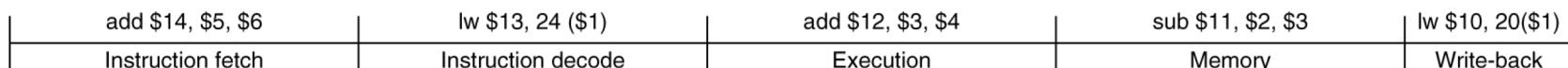
still have to go through the 5th stage

SW
Write-back



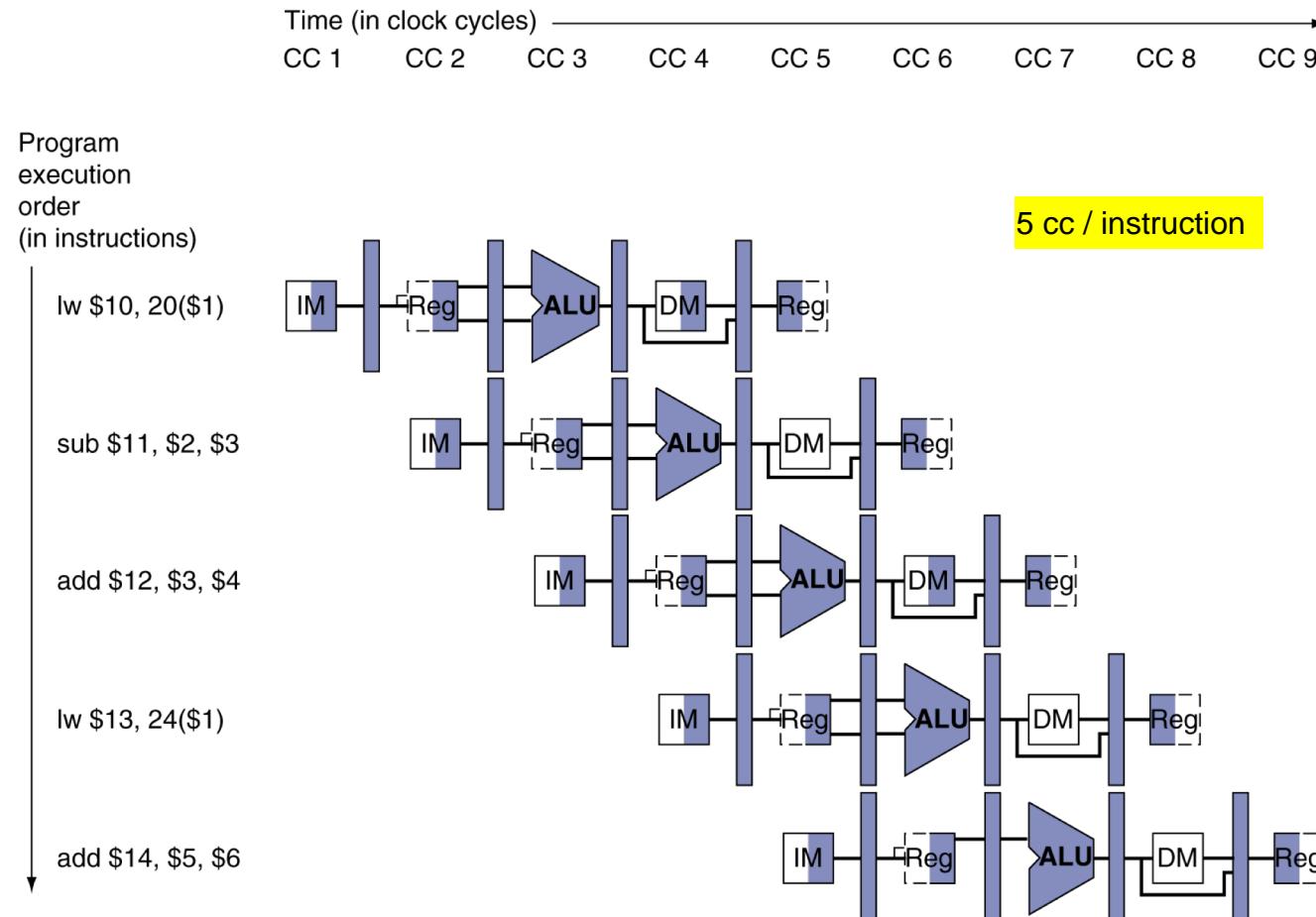
Single-Cycle Pipeline Diagram

State of pipeline in a given cycle



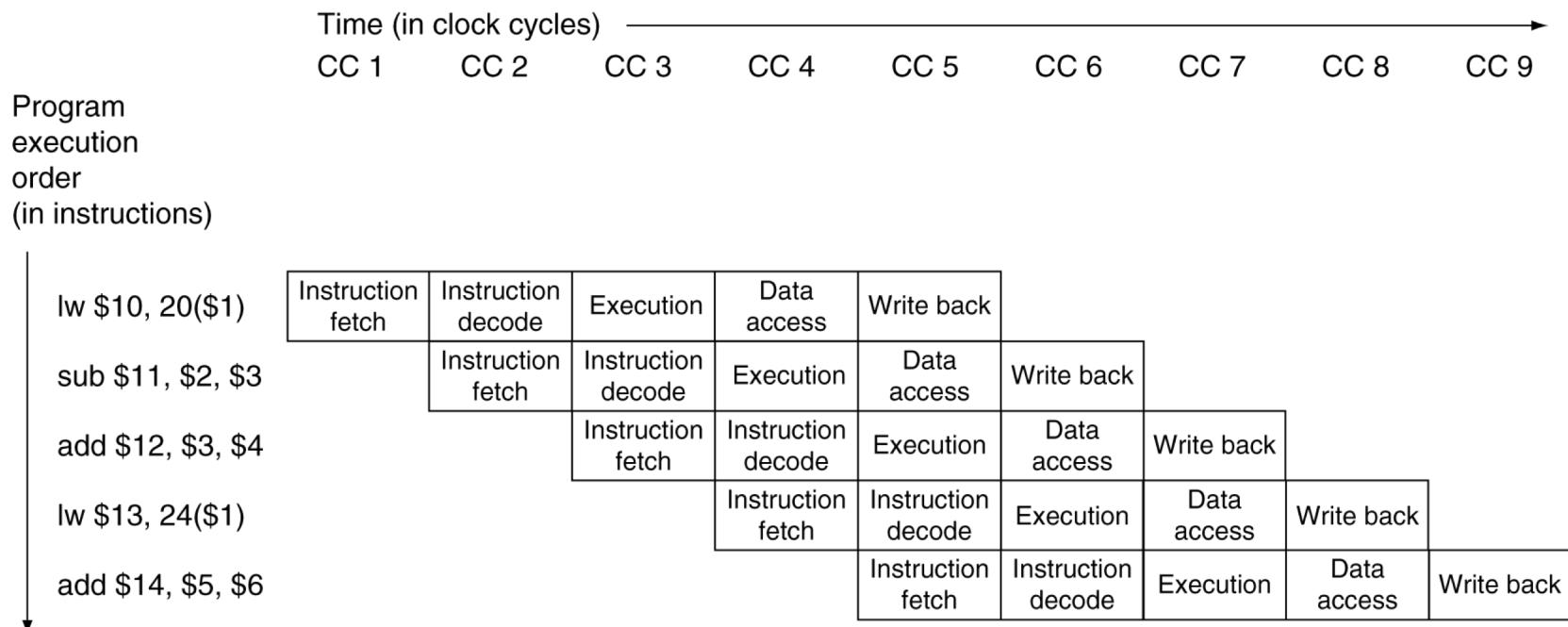
Multi-Cycle Pipeline Diagram

- Another way showing resource usage

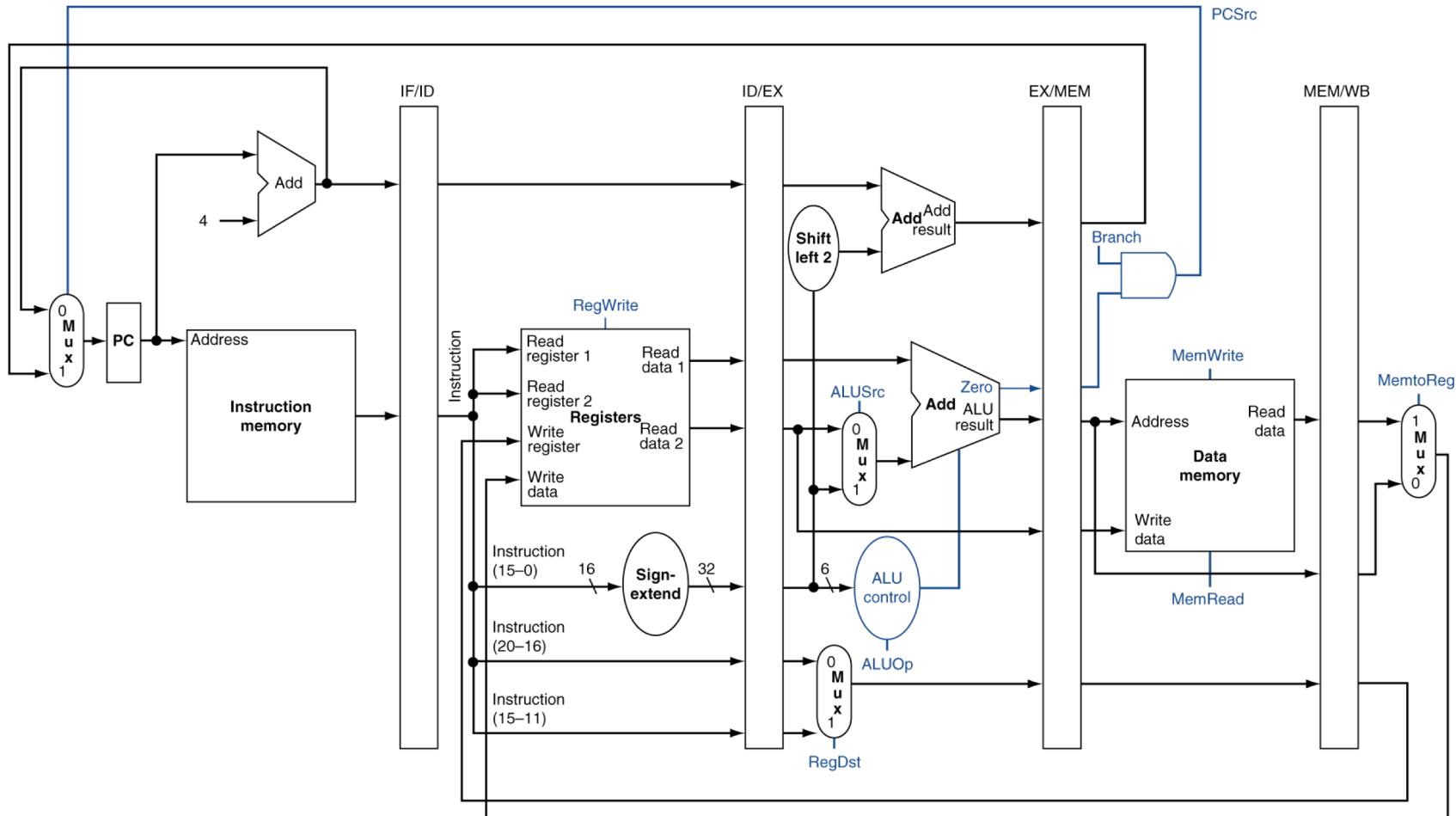


Multi-Cycle Pipeline Diagram

Traditional form

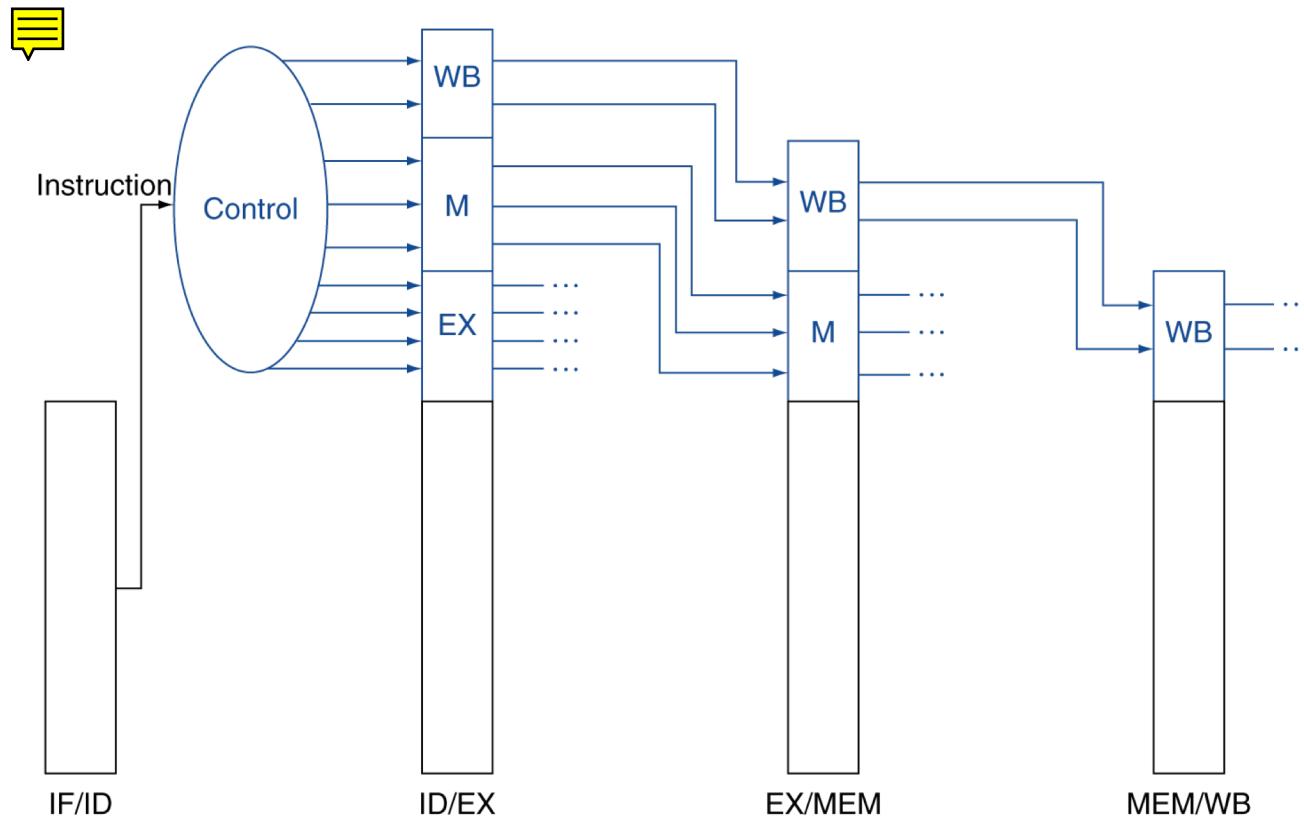


Pipelined Control (Simplified)

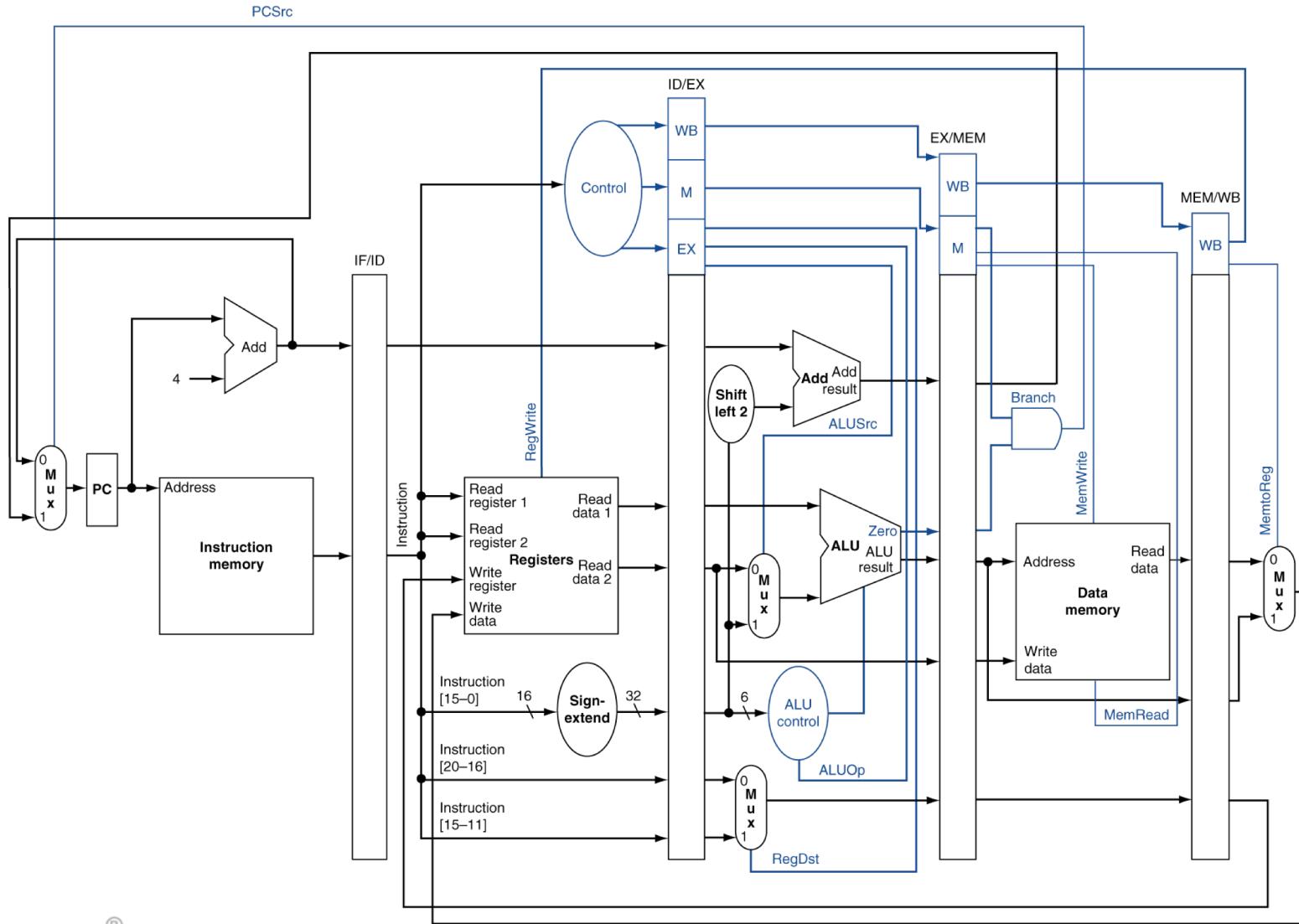


Pipelined Control

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



Pipelined Control



Pipelined Control

- Control signals derived from instruction

Inst.	EX			Mem			WB	
	ALU Src	Reg Dst	ALU Op	Mem Write	Mem Read	Branch	Memto Reg	Reg Write
add								
addi								
lw								
sw								
beq								

Pipeline Summary

The BIG Picture

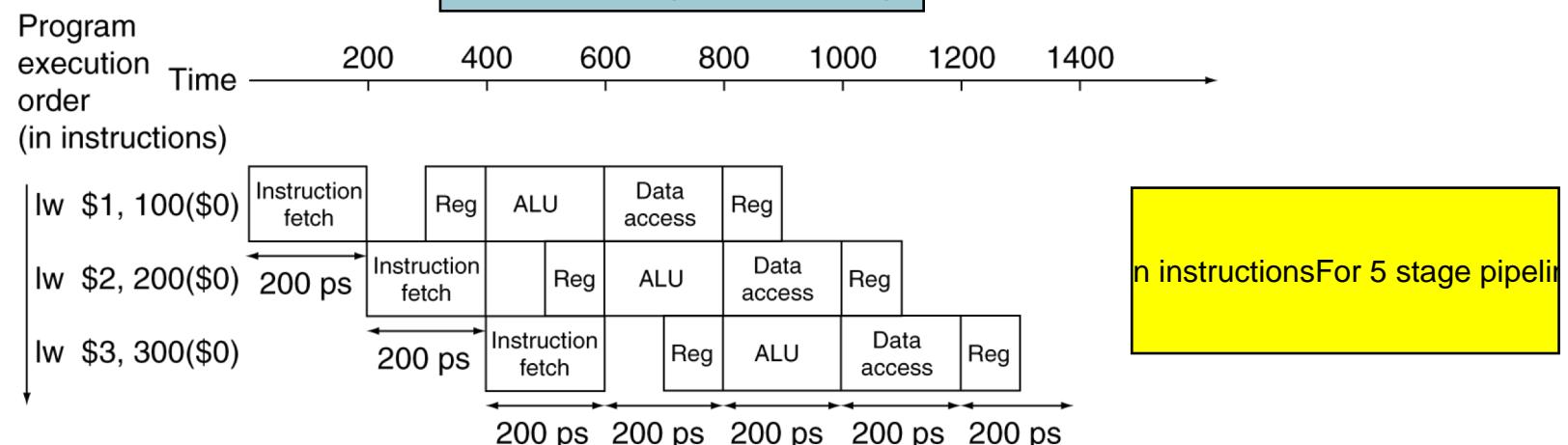
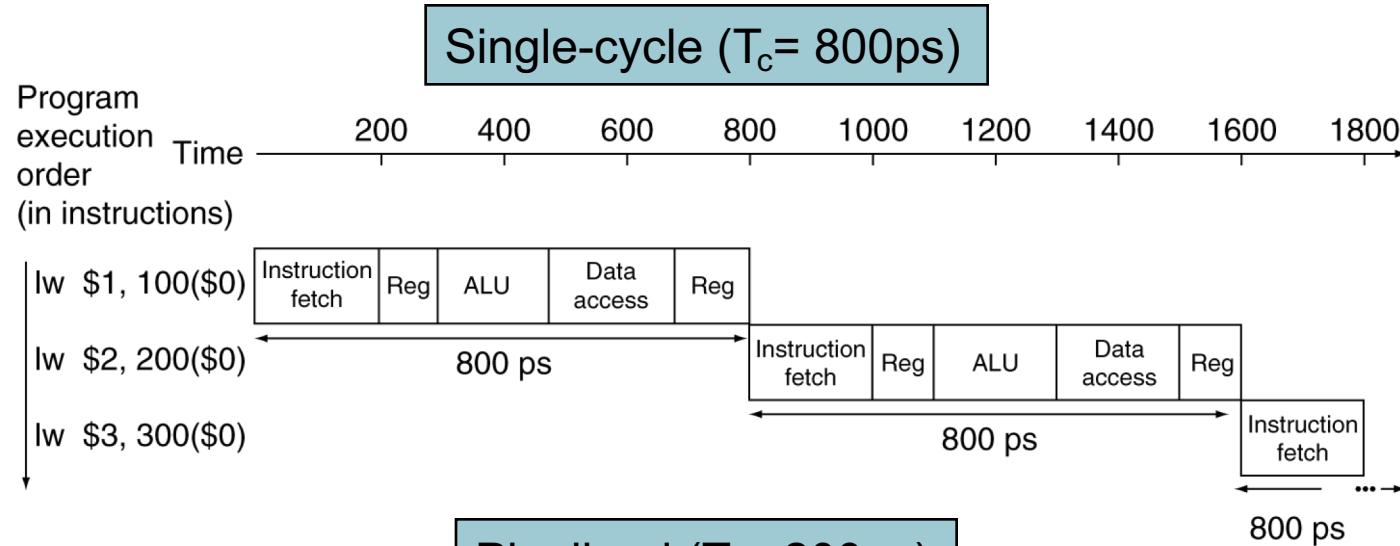
- Pipelining improves performance by increasing instruction throughput
 - Executes multiple instructions in parallel
 - Each instruction has the same latency
- Subject to hazards
 - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

Pipeline Performance

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

- If all stages are balanced
 - i.e., all take the same time
 - Time between instructions_{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