Basic processor Implementation

Jinyang Li

What we've learnt so far

- Combinatorial logic
  - Muxes, Decoders, …
  - ALU
- Sequential logic
  - State elements (SR latches, D-latches, Flip-flops, Register files)
  - Finite state machines
- RISC-V instructions

Today’s lesson plan

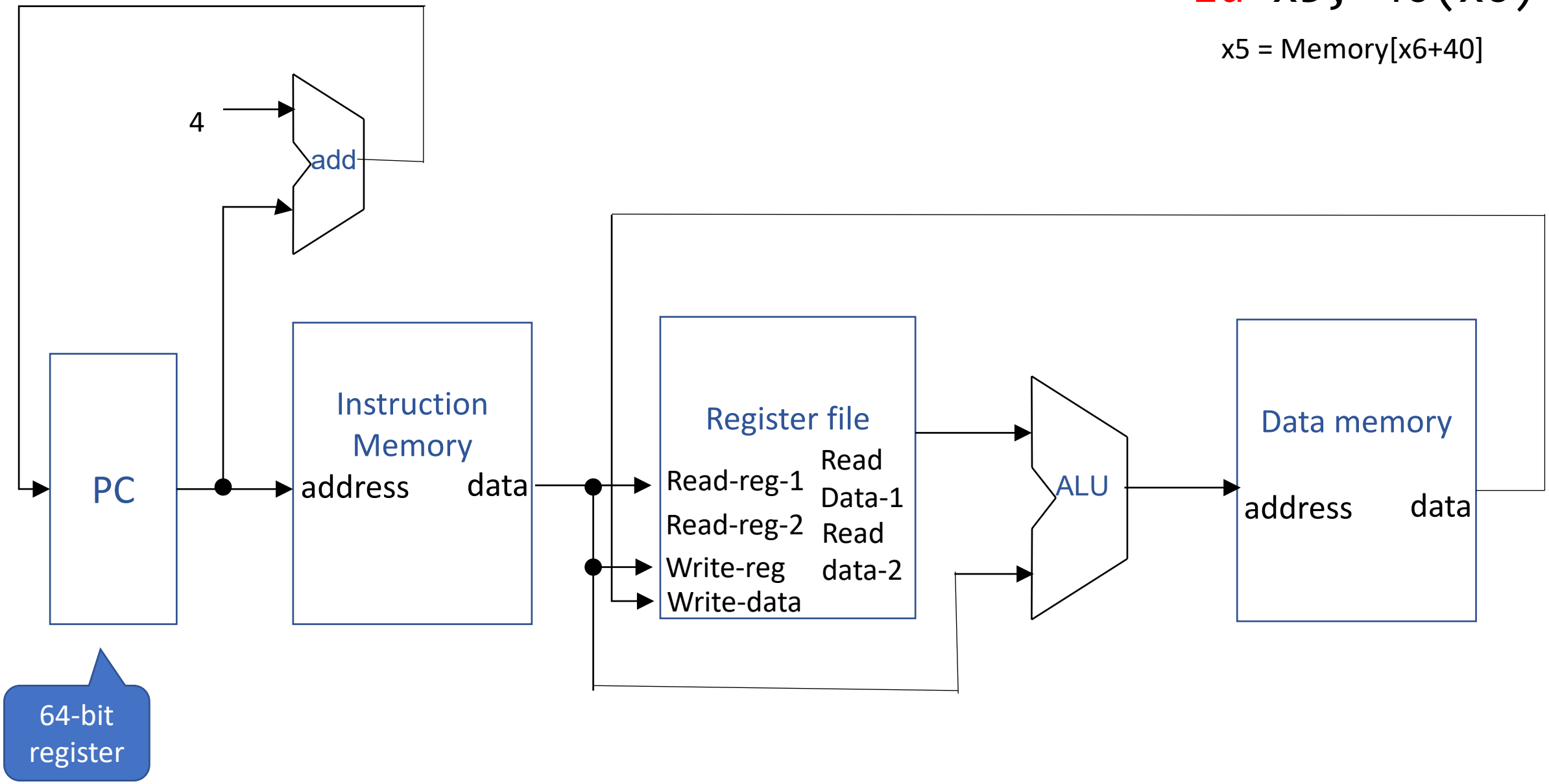
- Basic single-cycle CPU design
- Pipelining idea and challenges

RISC-V Instruction Execution

- PC $\rightarrow$ fetch instruction from memory
- Register numbers $\rightarrow$ which register to read/write in register file
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch comparison
  - Access data memory for load/store
  - PC $\leftarrow$ either target address (branch) or PC + 4

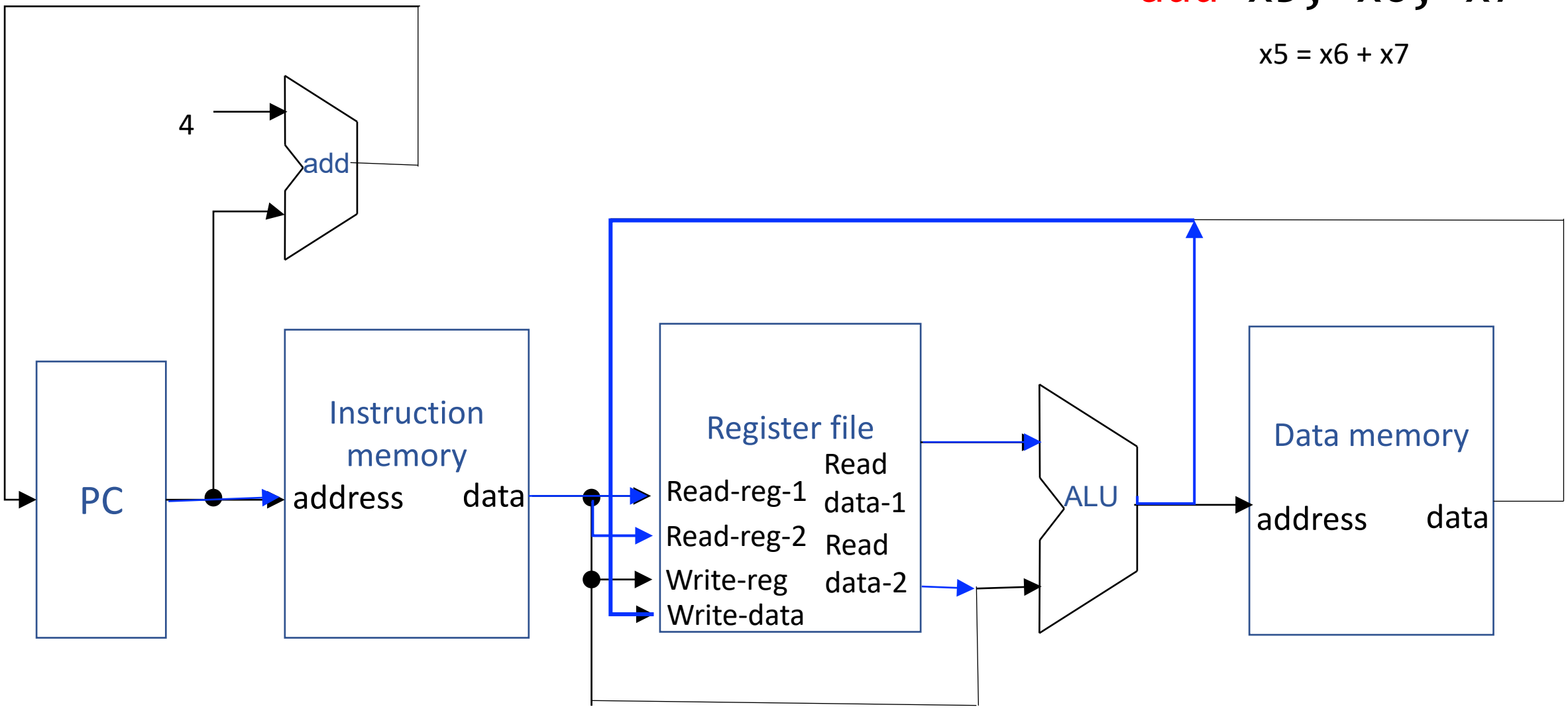
```
ld x5, 40(x6)
```

x5 = Memory[x6+40]



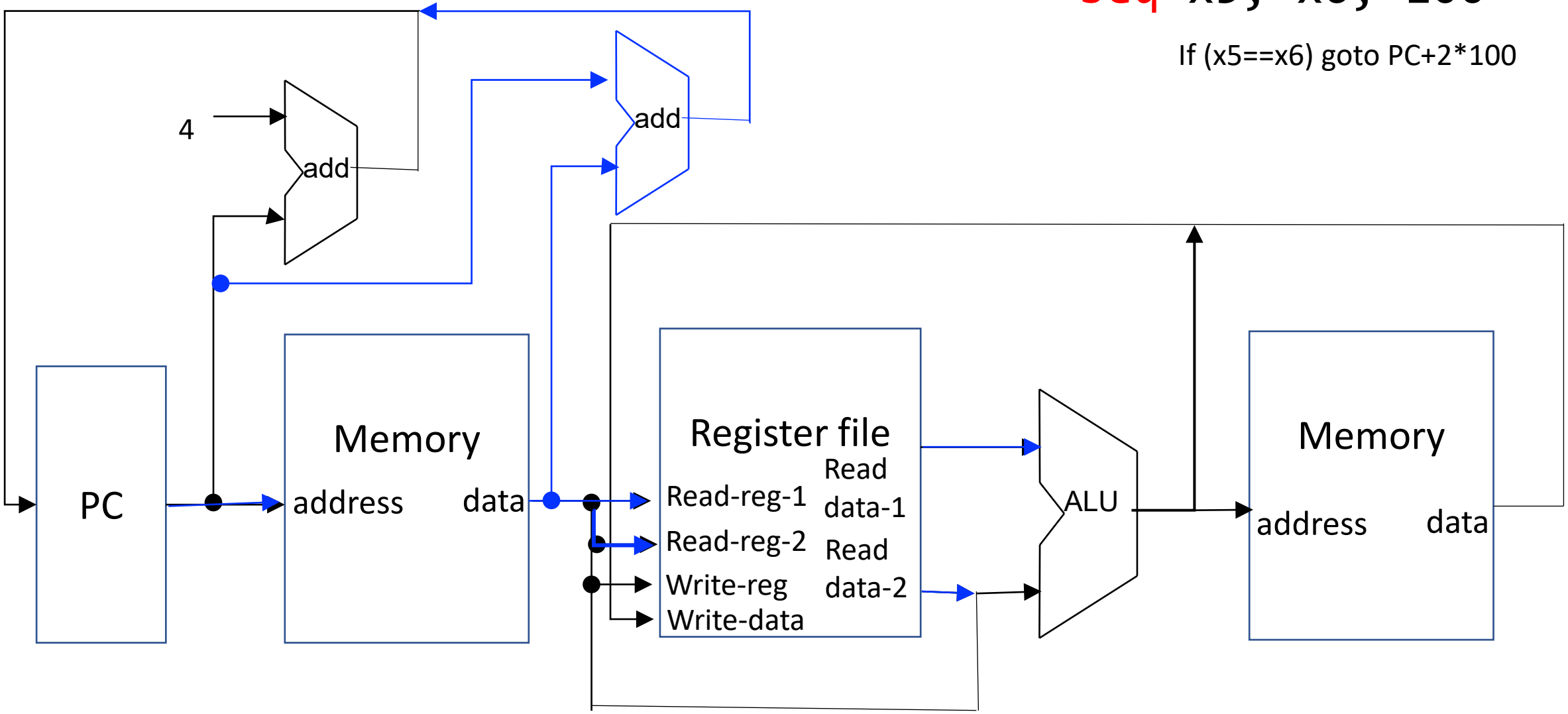
```
add x5, x6, x7
```

x5 = x6 + x7

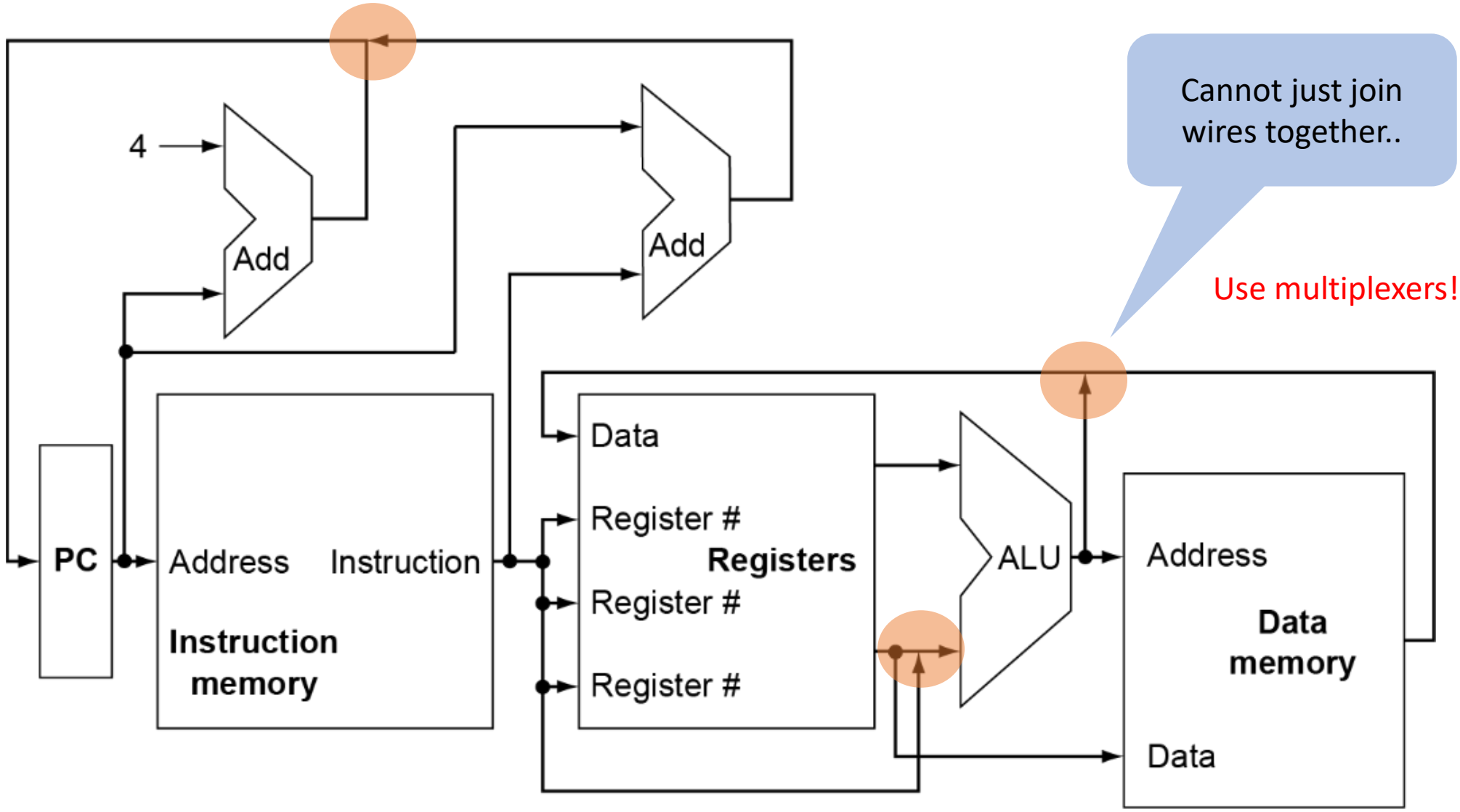


```
beq x5, x6, 100
```

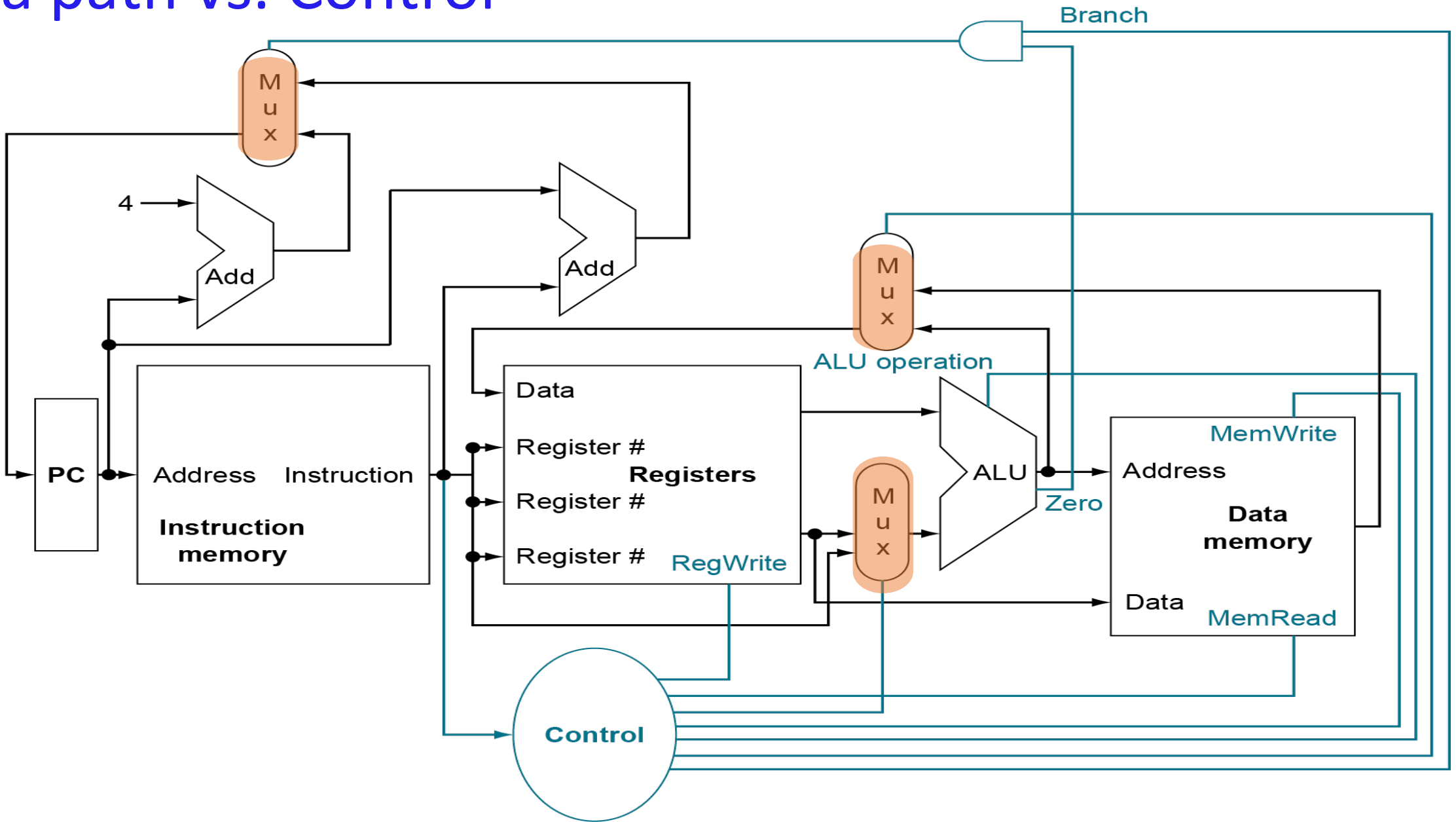
If (x5==x6) goto PC+2*100



Instruction Execution



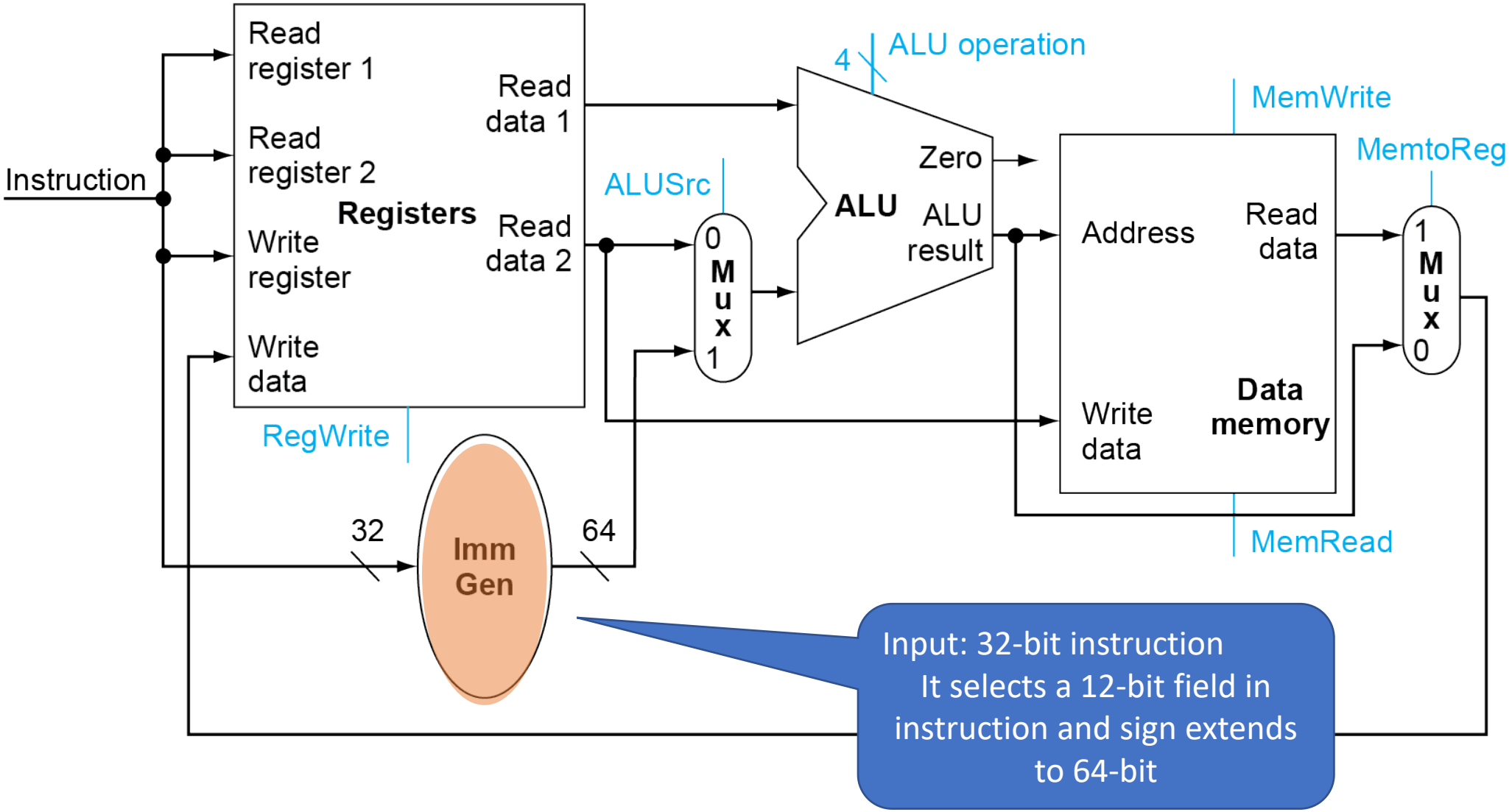
Data path vs. Control



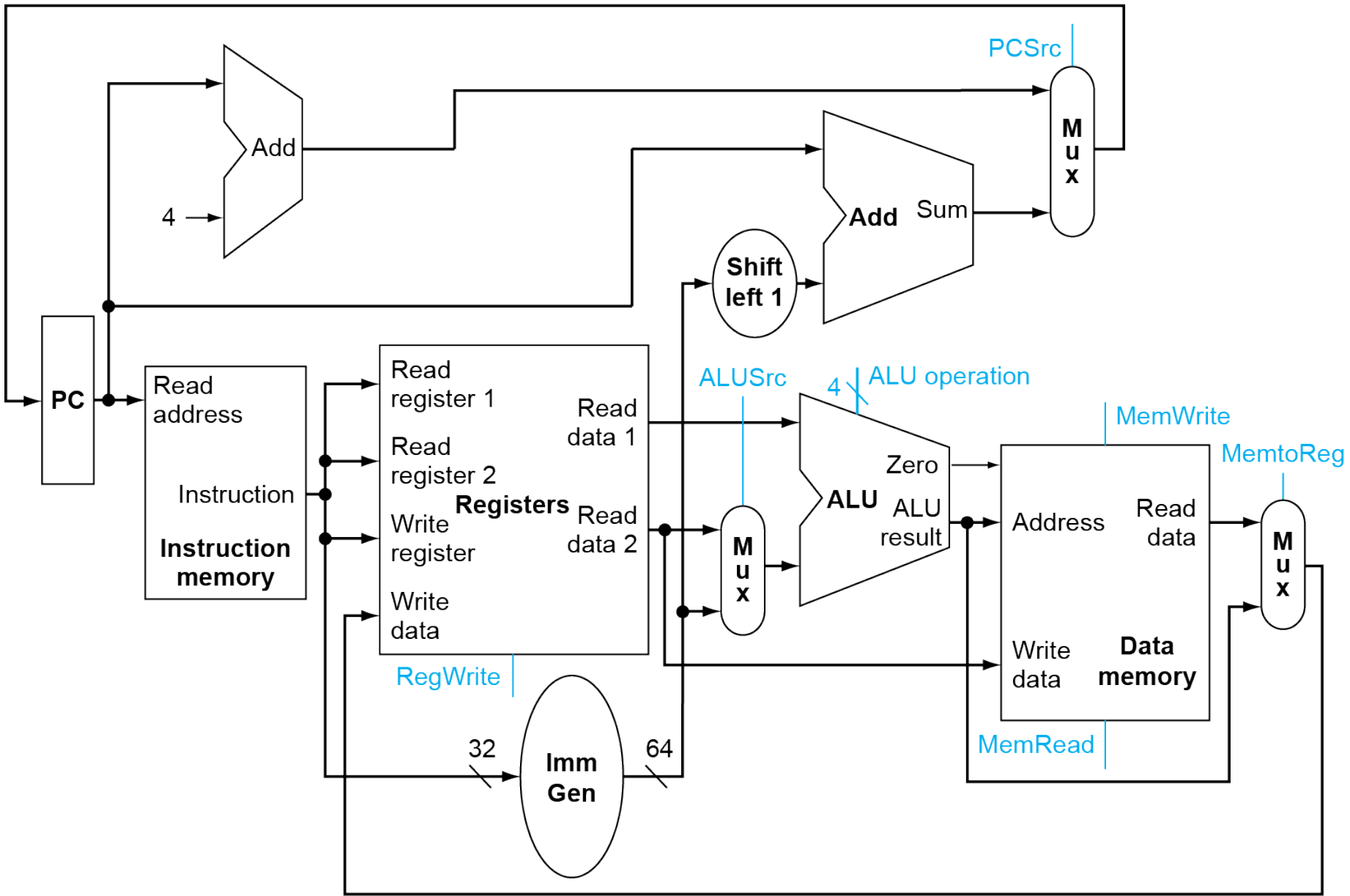
Building a Datapath

- Datapath: elements that process data and addresses in the CPU
  - Registers, ALUs, memories, …
- We will refine our overview datapath design next…

R-Type/Load/Store Datapath



Full Datapath



ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract (for comparison)
  - R-type: F depends on opcode

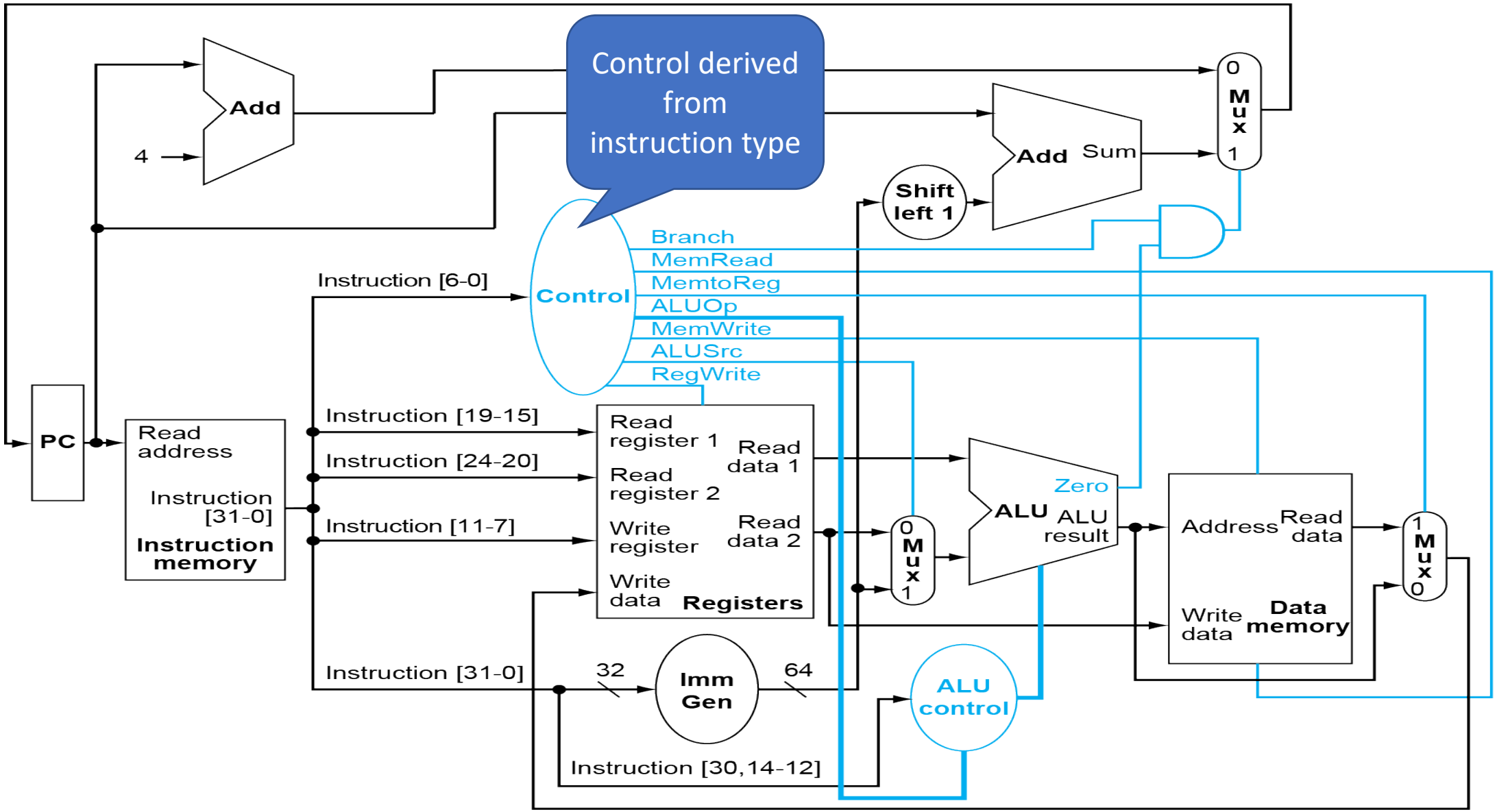
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

ALU Control

- Assume a “Control” combinatorial logic that outputs 2-bit ALUOp derived from opcode

opcode	ALU Op	Operation	func7	func3	ALU function	ALU control
ld	00	load register	xxxxxxx	xxx	add	0010
sd	00	store register	xxxxxxx	xxx	add	0010
beq	01	branch on equal	xxxxxxx	xxx	subtract	0110
R-type	10	add	000000	000	add	0010
		subtract	010000	000	subtract	0110
		AND	000000	111	AND	0000
		OR	000000	110	OR	0001

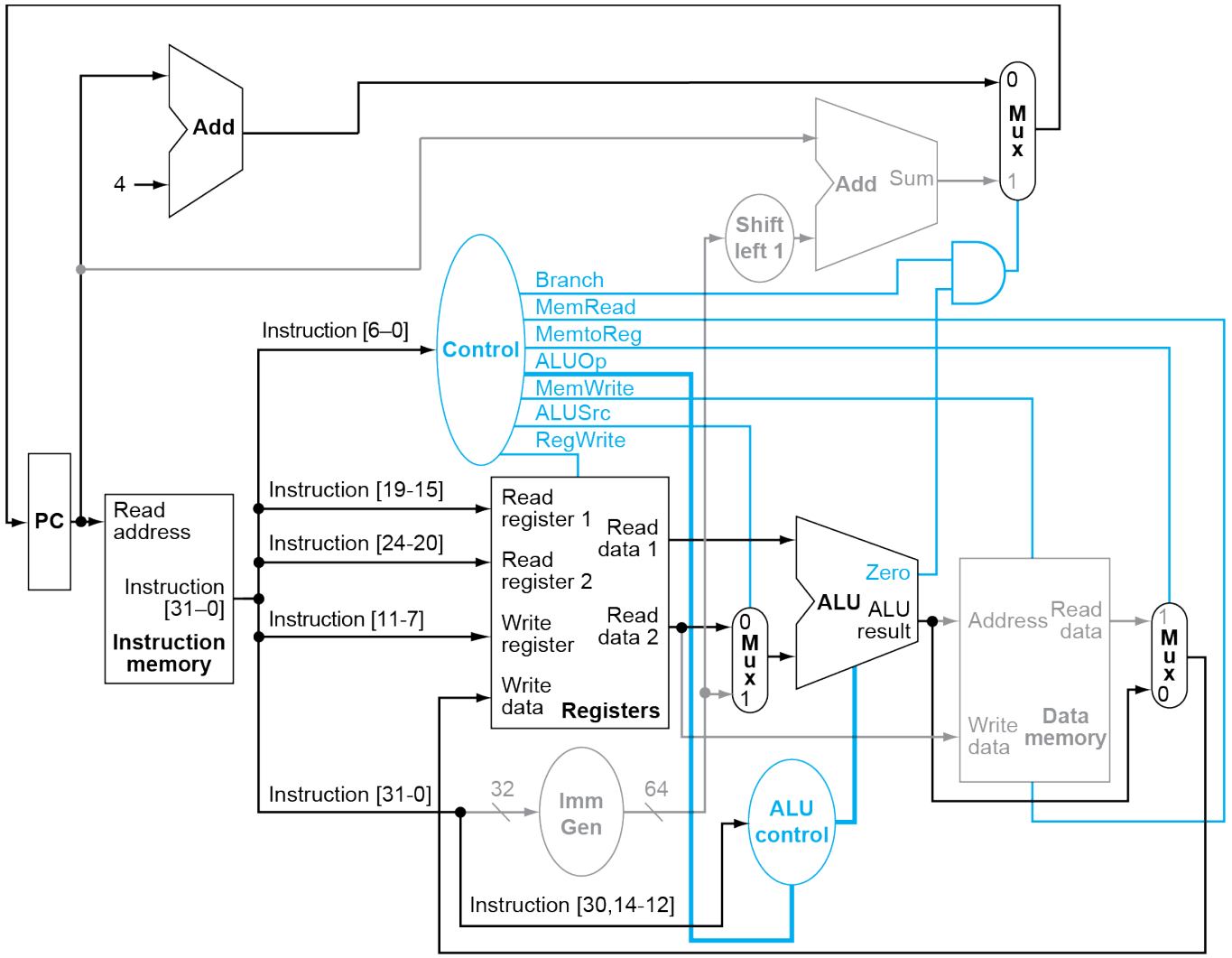
Datapath With Control



```
add x5, x6, x7
```

R-Type Inst

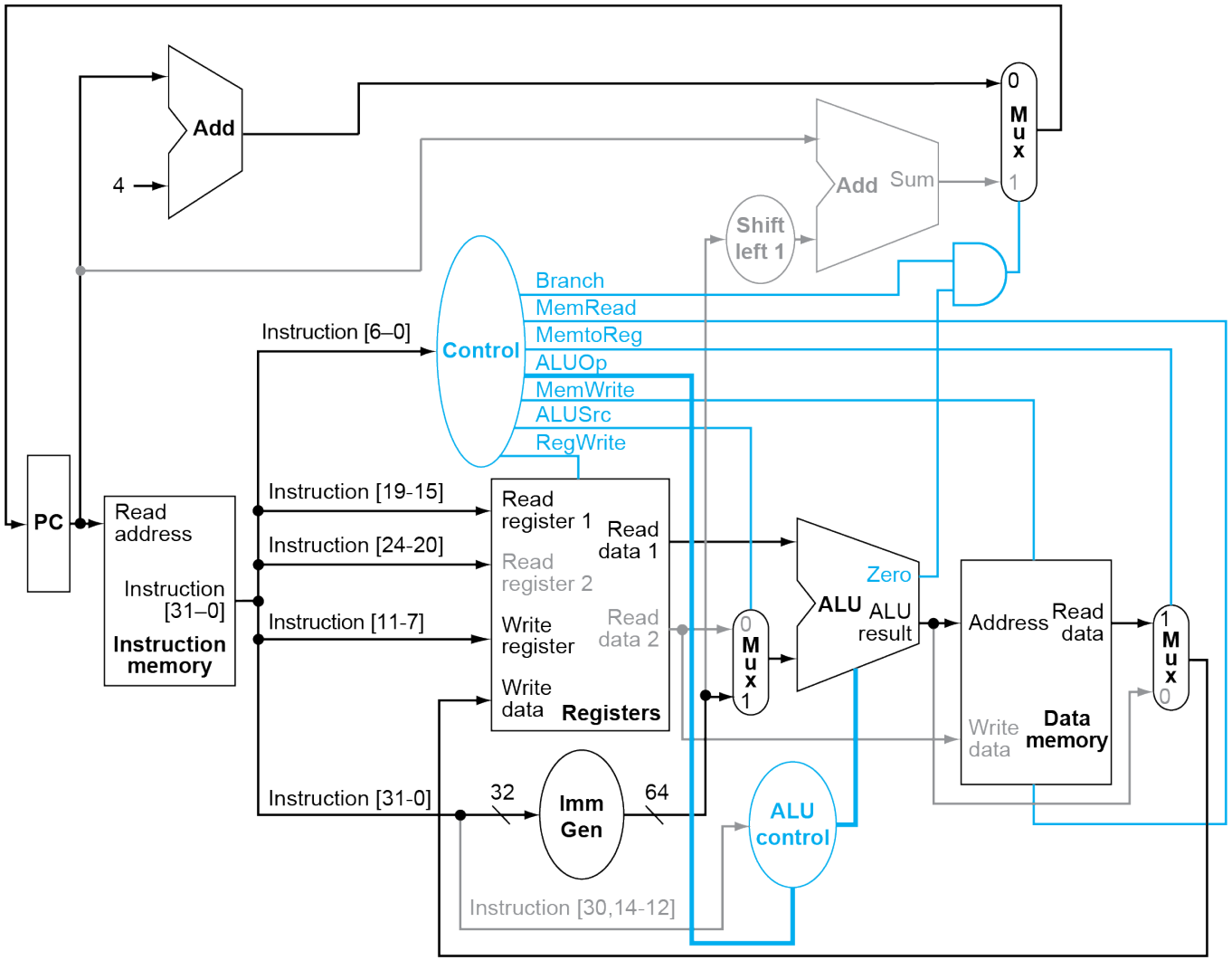
funct7	rs2	rs1	funct3	rd	opcode



```
ld x5, 40(x6)
```

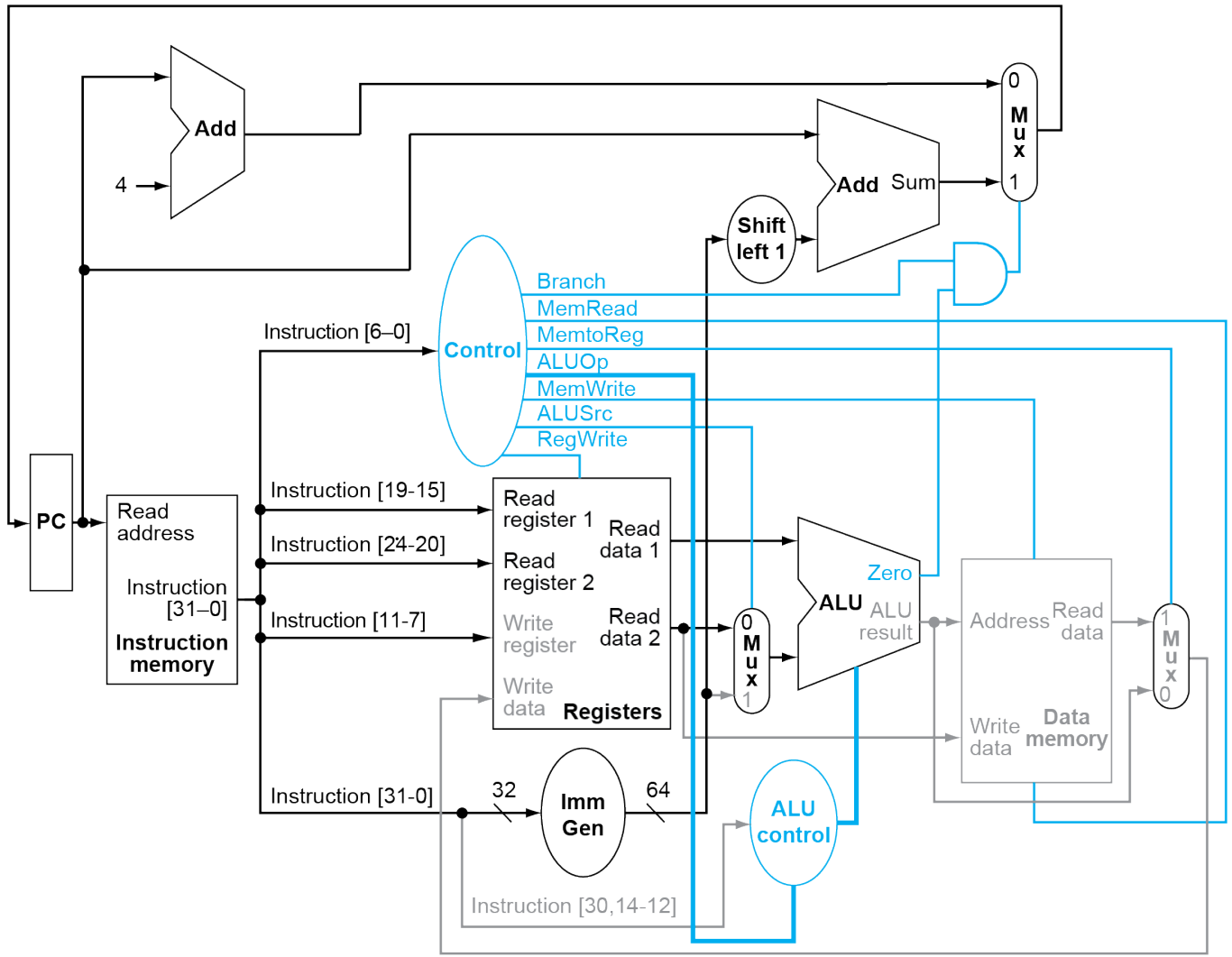
Load Inst

immediate	rs1	funct3	rd	opcode



BEQ Instruction

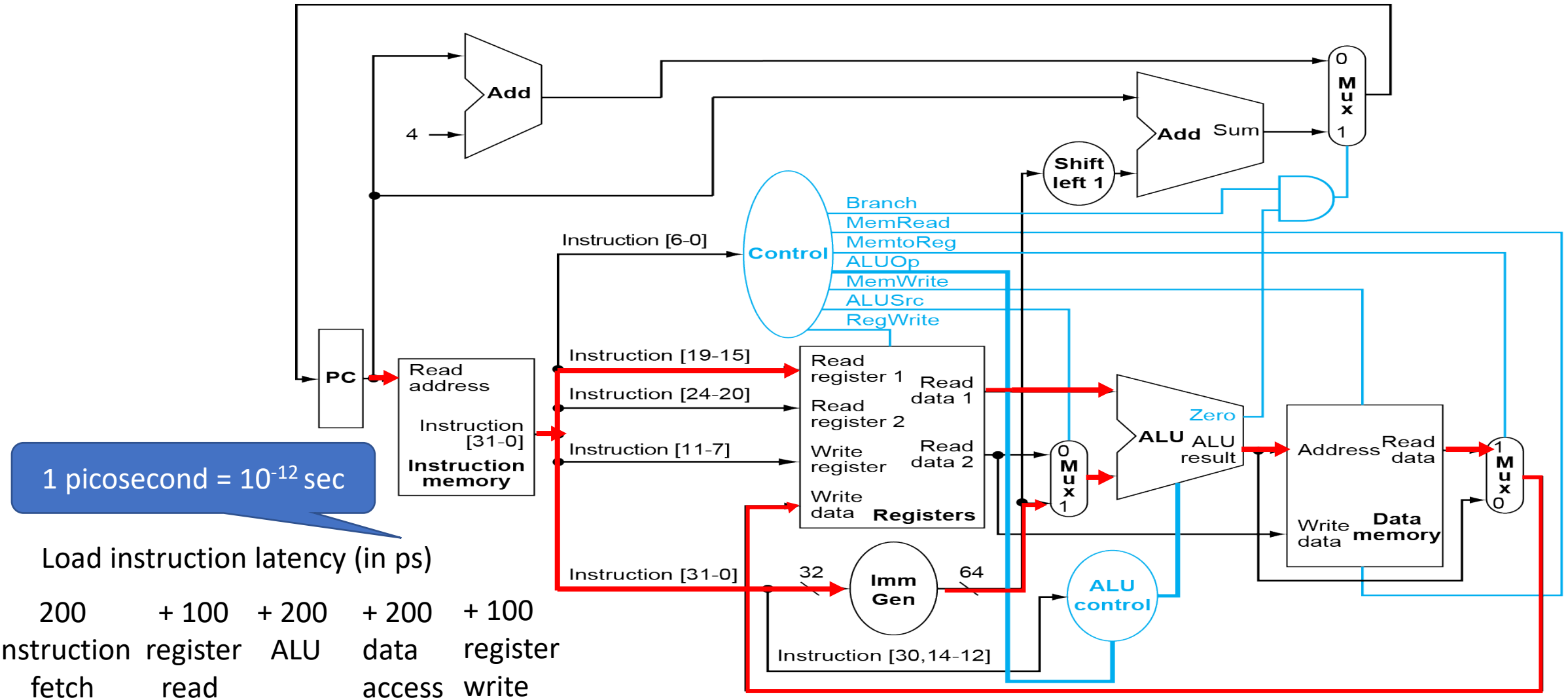
```
beq x5, x6, 100
```



Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file
- Not feasible to vary clock period for different instructions
- Next lectures: We will improve performance by pipelining

Single-cycle CPU uses a slow clock



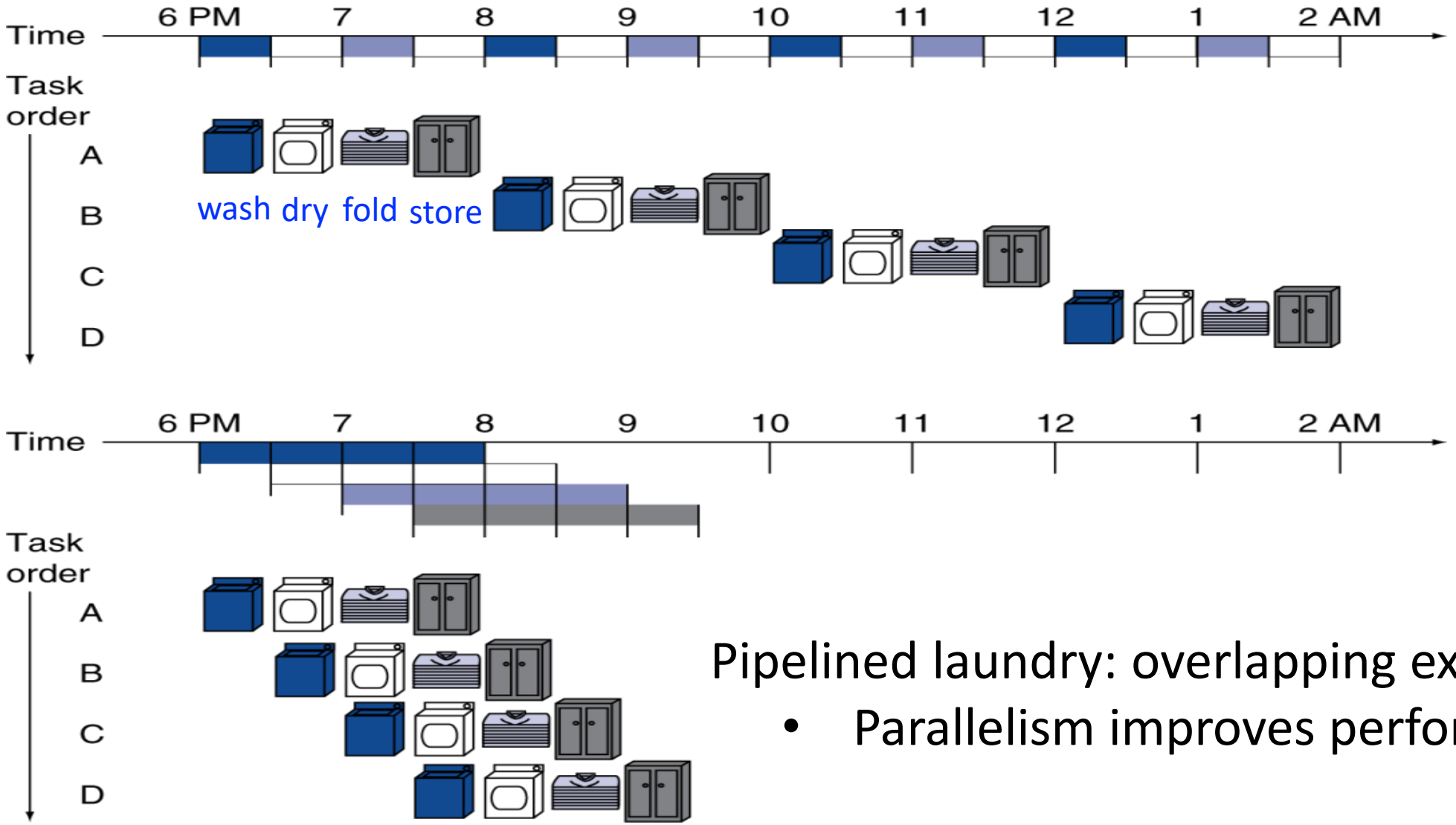
1 picosecond = 10^{-12} sec

Load instruction latency (in ps)

200	+ 100	+ 200	+ 200	+ 100
instruction fetch	register read	ALU	data access	register write

Clock cycle >= 800 ps

Pipelining: a laundry analogy



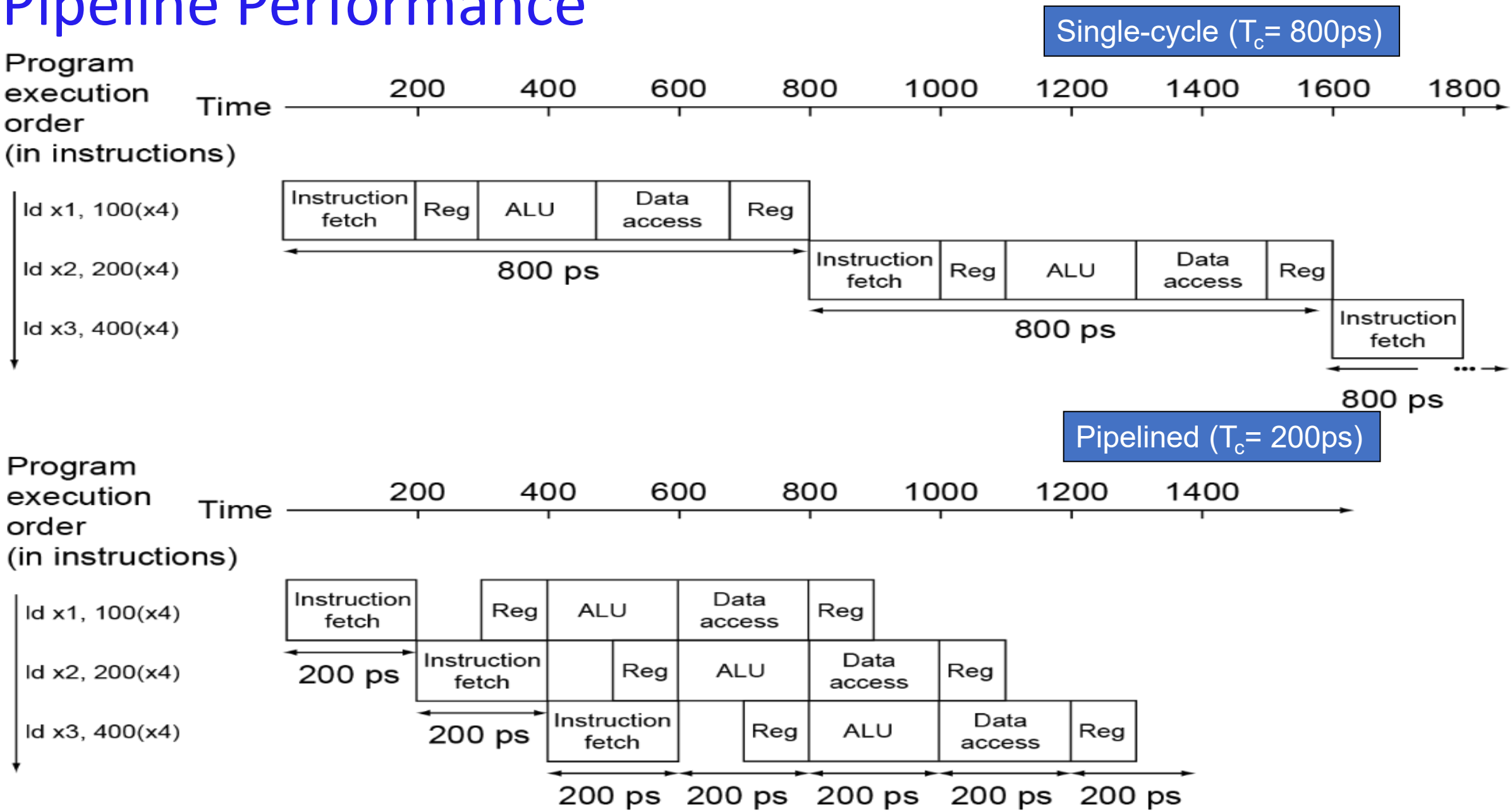
Pipelined laundry: overlapping execution

- Parallelism improves performance

RISC-V Pipeline

- Five stages:
  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

Pipeline Performance



Pipeline Speedup

- Pipelining increases throughput (instructions/sec)
  - Latency (time for each instruction) does not decrease
- If all stages are balanced (i.e., all take the same time)
  - $\text{throughput}_{\text{pipelined}} = \text{number-of-stages} * \text{throughput}_{\text{nonpipelined}}$
  - If not balanced, speedup is less

Throughput = 1/time between instructions

Pipelining and ISA Design

- RISC-V ISA is 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

Pipeline challenges: hazards

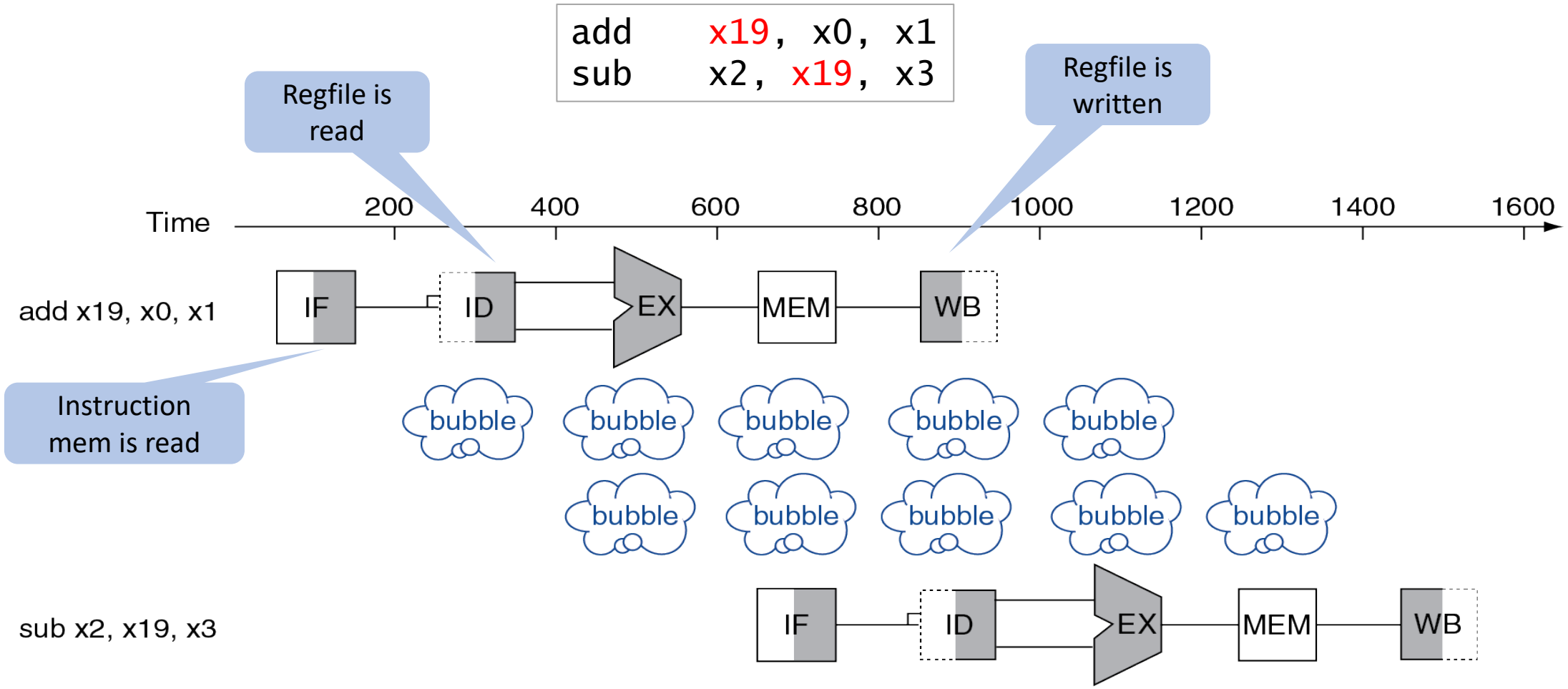
- Situations that prevent starting next instruction in the next cycle
- Structure hazard
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its write
- Control hazard
  - Which instruction to execute depends on previous instruction

Structure Hazards

- Conflict use of a single resource
- Example: Suppose CPU uses a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Solution: Use separate instruction/data memories

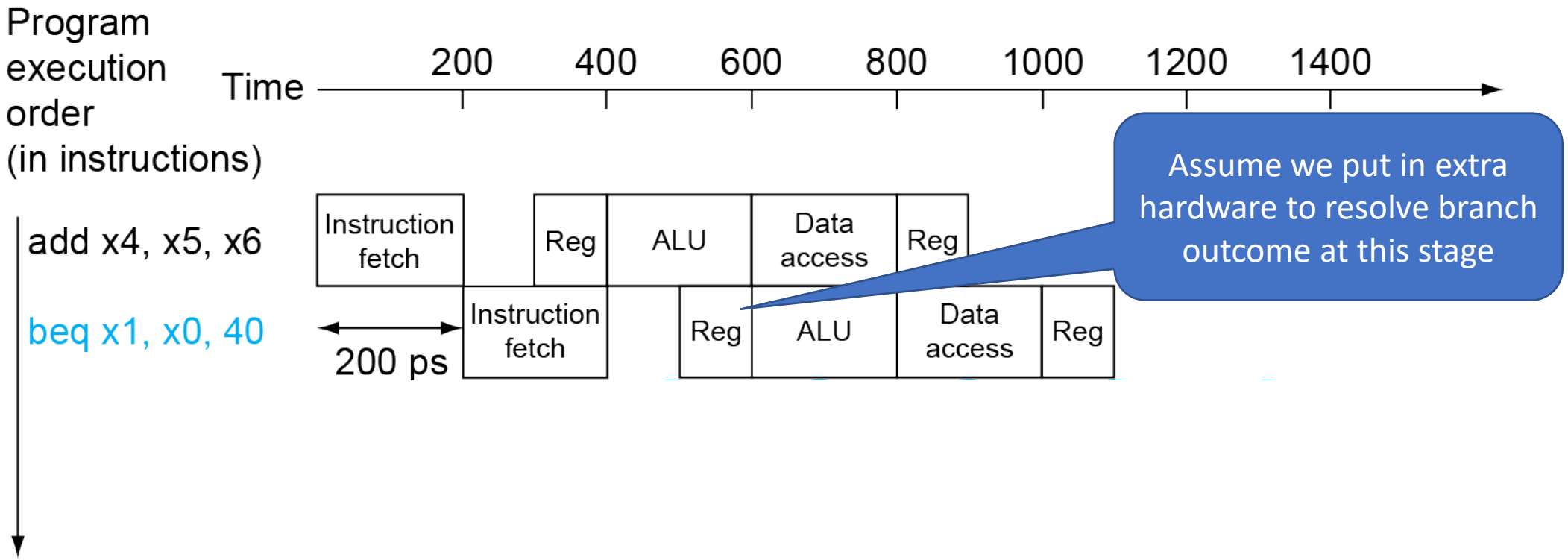
Data Hazards

- An instruction depends on the previous instruction to complete its write



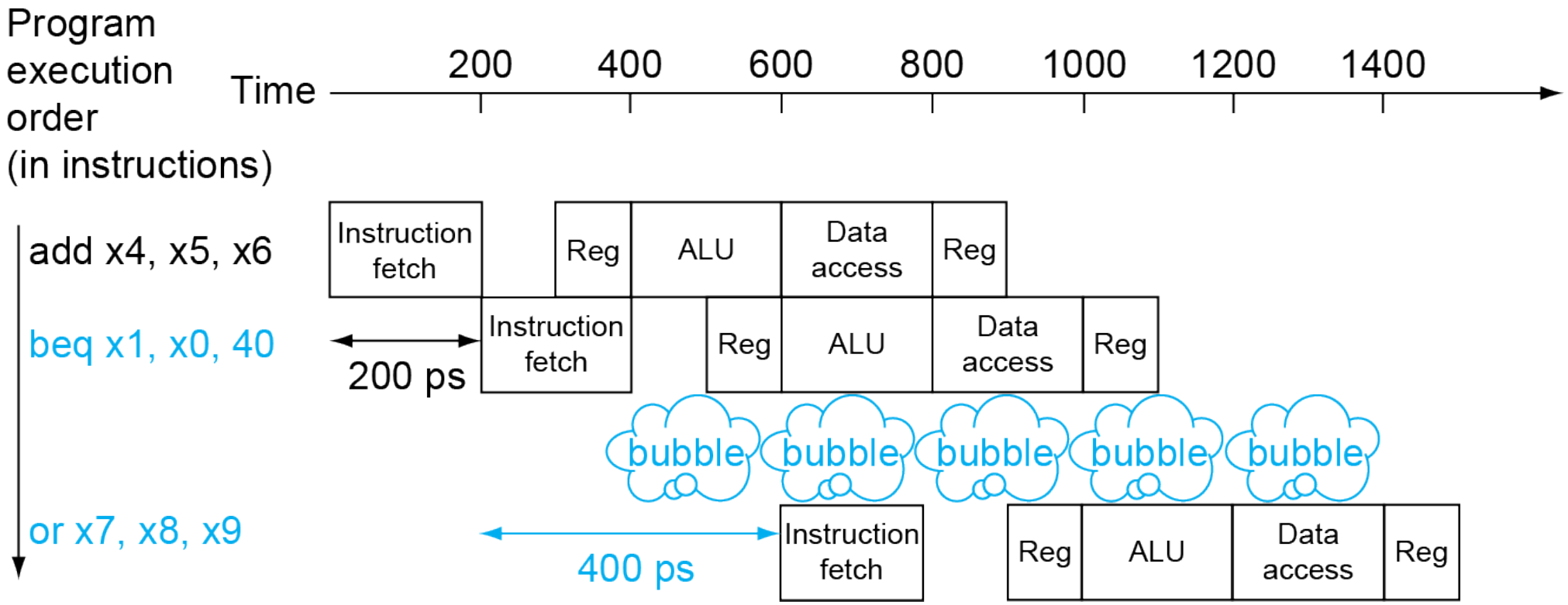
Control hazard

- Wait until branch outcome is determined before fetching next instruction



Control hazard

- Wait until branch outcome is determined before fetching next instruction



Summary

- Basic single-cycle CPU design
  - Data path vs. control path
  - Clock frequency is limited by the longest delay
- Pipelining idea and challenges
  - Parallel processing of different stages of an instruction’s execution
  - RISC-V 5-stage pipeline (IF, ID, EXE, MEM, WB)
  - Pipeline hazards: structure, data, control