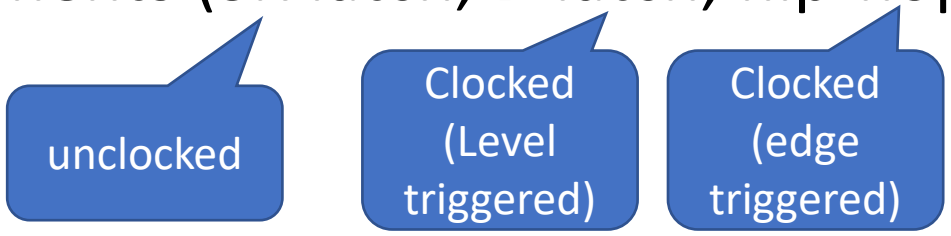
Basic Processor Implementation

Jinyang Li

What we've learnt so far

- Combinatorial logic
  - Truth table
  - ROM
- ALU
- Sequential logic
  - Clocks
  - Basic state elements (SR latch, D latch, flip-flop)



Today’s lesson plan

- Implement a basic CPU

Our CPU will be based on RISC-V instead of x86

- 3 popular ISAs now

	ISA	Key advantage	Who builds the processors?	Where are the processors used?
CISC Complex Instruction Set	x86	Fast	Intel, AMD	Server (Cloud), Desktop, Laptop, Xbox console
RISC Reduced Instruction Set	ARM	Low power (everybody can license the design from ARM Holdings for $$$)	Samsung, NVIDIA, Qualcomm, Broadcom, Huawei/HiSilicon	Phones, Tablets, Nintendo console, Raspberry Pi
	RISC-V	Open source, royalty-free	Western digital, Alibaba	Devices (e.g. SSD controllers)

RISC-V at a high level

similarities

	RISC-V	X86-64
# of registers	32	16
Memory	Byte-addressable, Little Endian	Byte-addressable, Little Endian

Why RISC-V is much simpler?

Fewer instructions	50+ (200 manual pages)	1000+ (2306 manual pages)
Simpler instruction encoding	4-byte	Variable length
Simpler instructions	• Ld/st instructions load/store memory to or from register • Other instructions take only register operands	• Instructions take either memory or register operands • Complex memory addressing modes D(B, I, S) • Prefixes modify instruction behavior

Basic RISC-V instructions

Registers: x0, x1, x2,…, x31

64-bit

Data transfer	load doubleword	`ld x5, 40(x6)`	x5=Memory[x6+40]
	store doubleword	`sd x5, 40(x6)`	Memory[x6+40]=x5
Arithmetic Logical	addition	`add x5, x6, x7`	x5 = x6 + x7
	substraction	`sub x5, x6, x7`	x5 = x6 - x7
	bit-wise and	`and x5, x6, x7`	x5 = x6 & x7
	bit-wise or	`or x5, x6, x7`	x5 = x6 | x7
Conditional Branch	Branch if equal	`beq x5, x6, 100`	If (x5==x6) go to PC+100

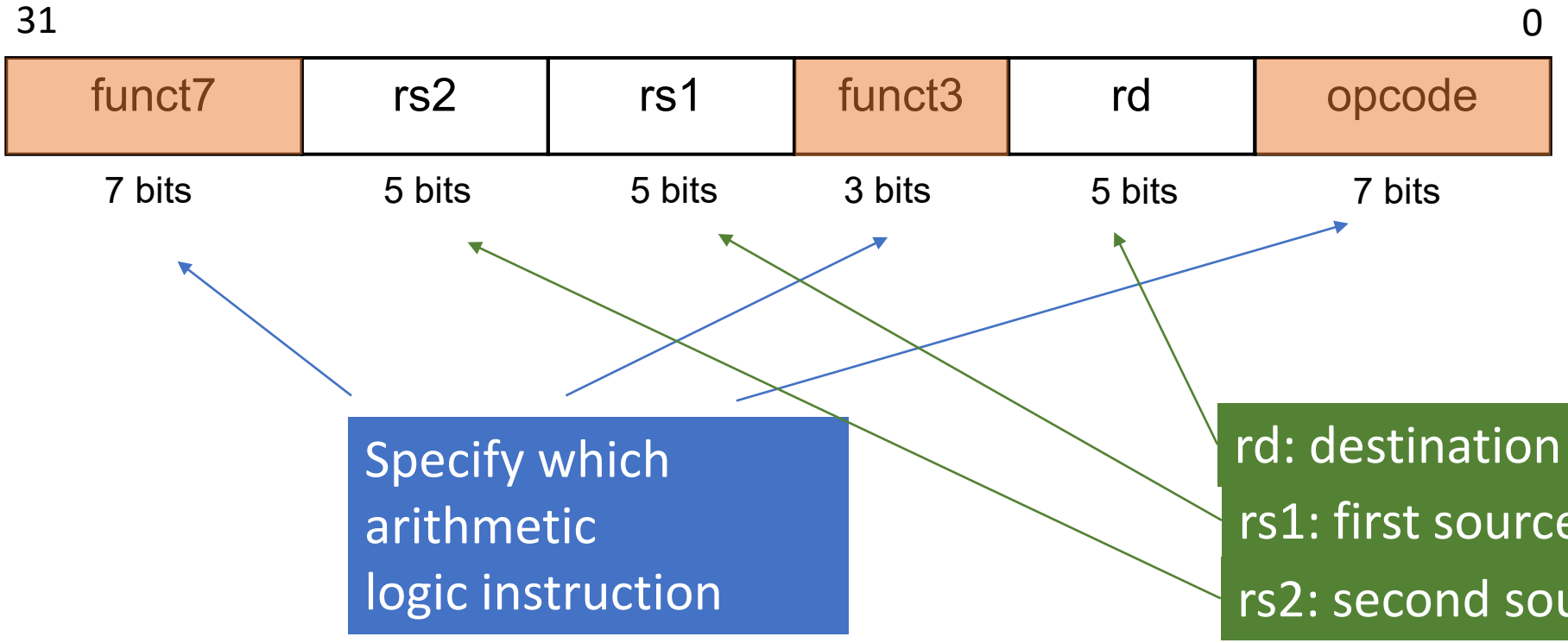
RISC-V instruction formats

- All instructions are 32-bit long
  - Several formats encoding operation code (opcode), register numbers …

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

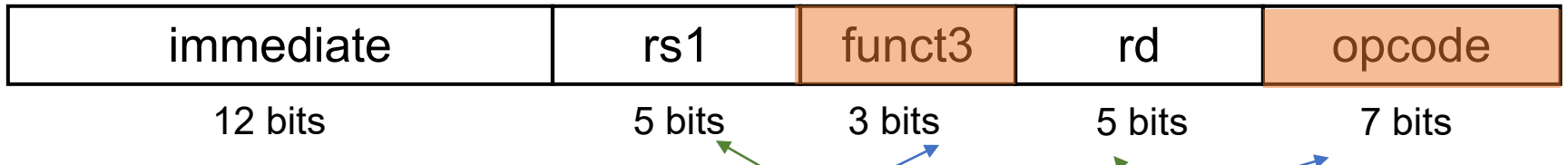
Opcode determines type of instruction

R-type: arithmetic logic instructions



	opcode	funct3	funct7
add	0110011	000	0000000
sub	0110011	000	0100000
xor	0110011	100	0000000

I-type: loads and immediate arithmetic



Specify which instruction

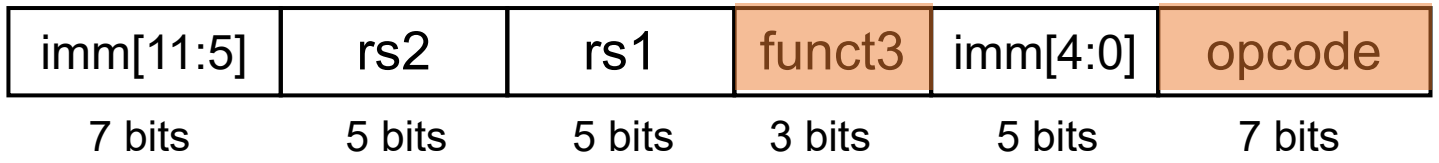
rd: destination register

rs1: first source register

Immediate: constant operand

	opcode	funct3
ld	0000011	011
addi (add immediate)	0010011	000

S-type: stores



Specify which instruction

rs1: base address register number

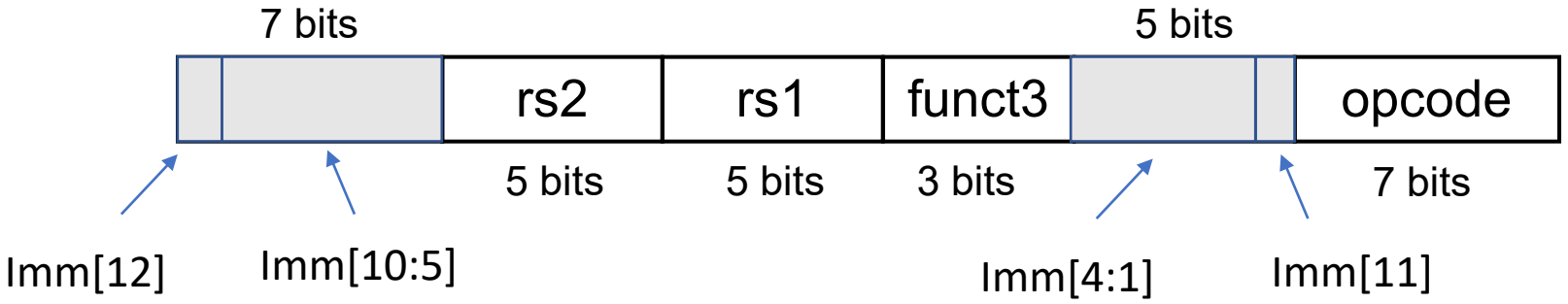
rs2: source operand register number

immediate: offset added to base address

* Split so that rs1 and rs2 fields always in the same place

	opcode	funct3
Sw (store word)	0100011	010
sd (store doubleword)	0100011	111

SB-type: conditional branch



- Immediate encodes a 12-bit signed integer offset.
- Branch address is PC+offset*2

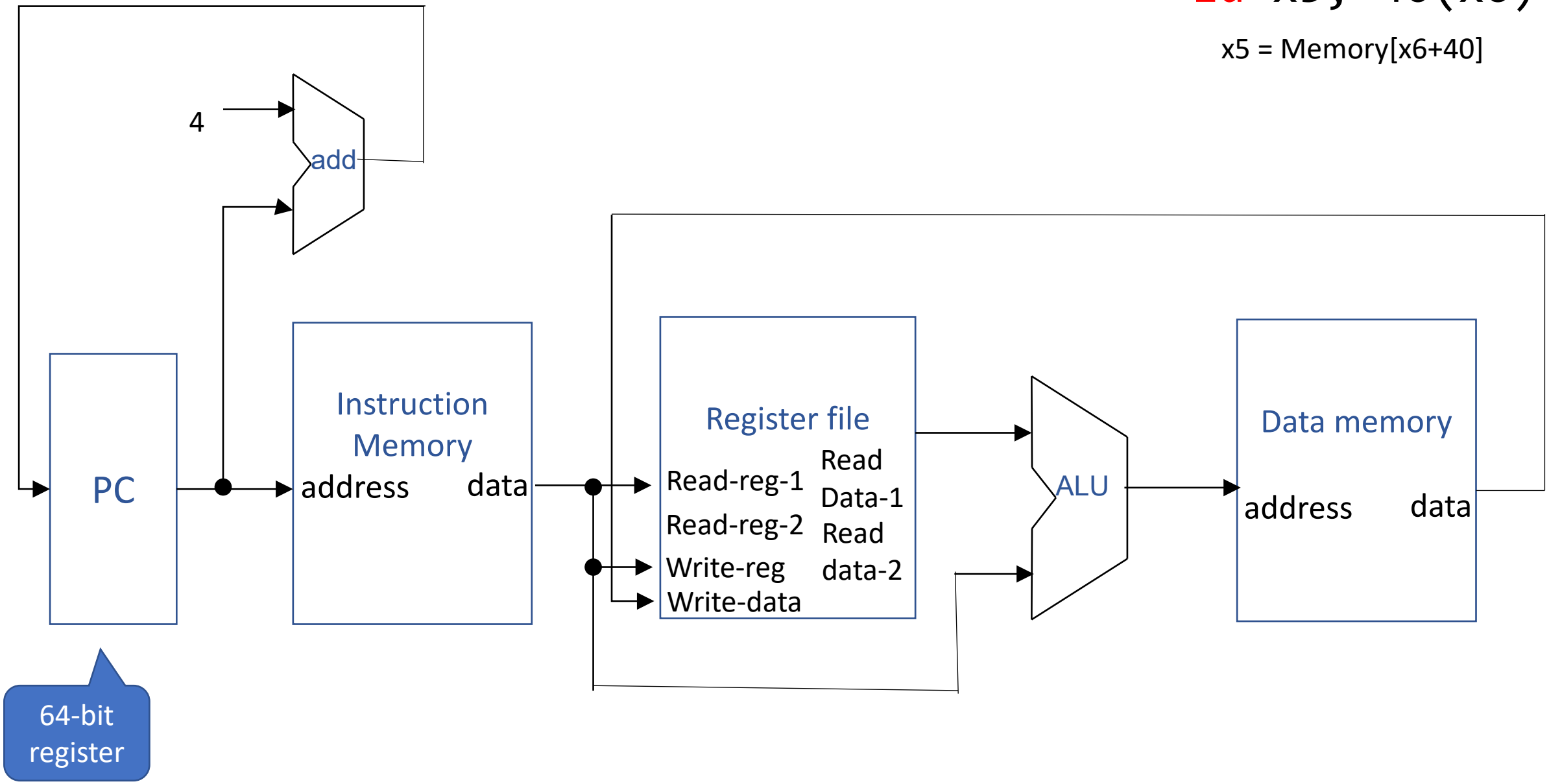
	opcode	funct3
beq	1100111	000
bne	1100111	001

Instruction Execution

- PC → fetch instruction from memory
- Register numbers → 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 ← either target address (branch) or PC + 4

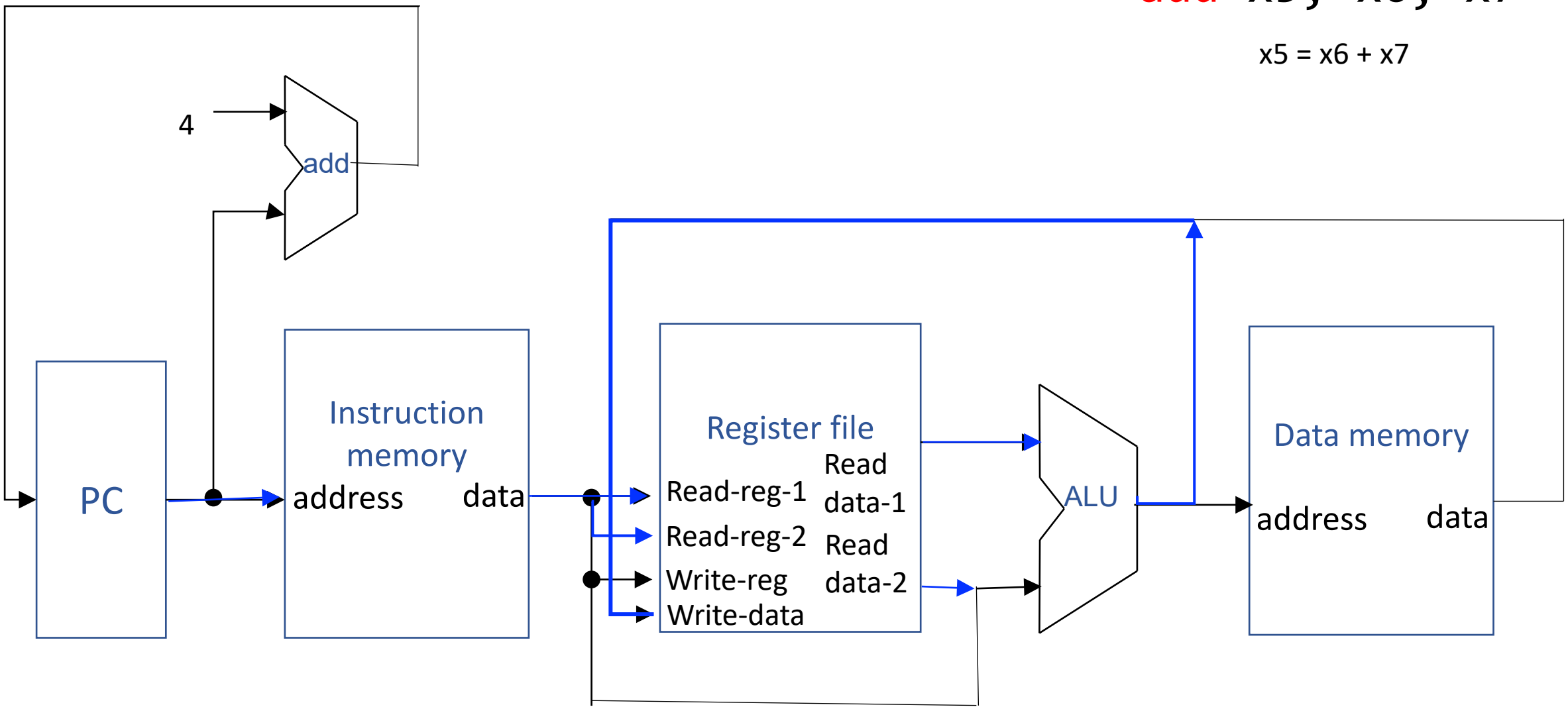
```
ld x5, 40(x6)
```

x5 = Memory[x6+40]



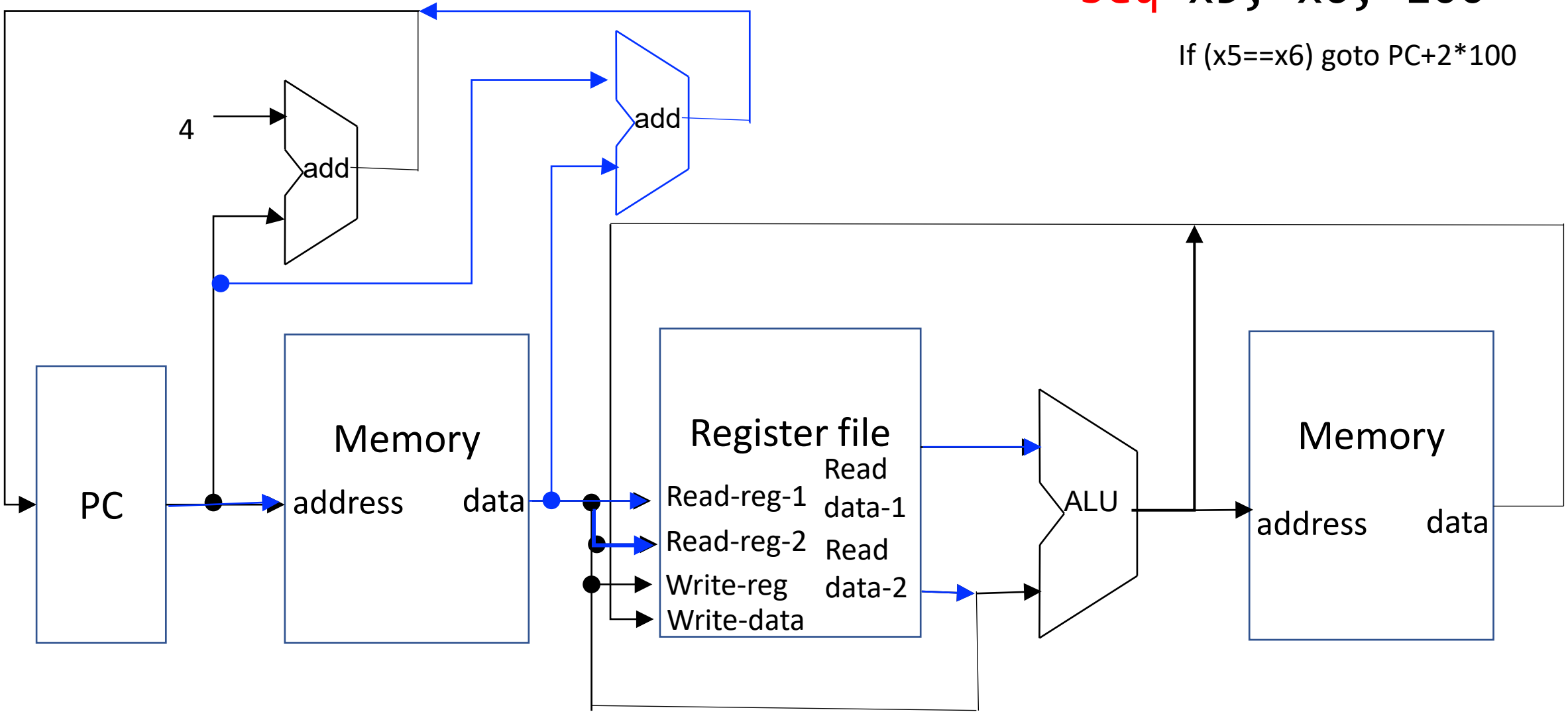
```
add x5, x6, x7
```

x5 = x6 + x7

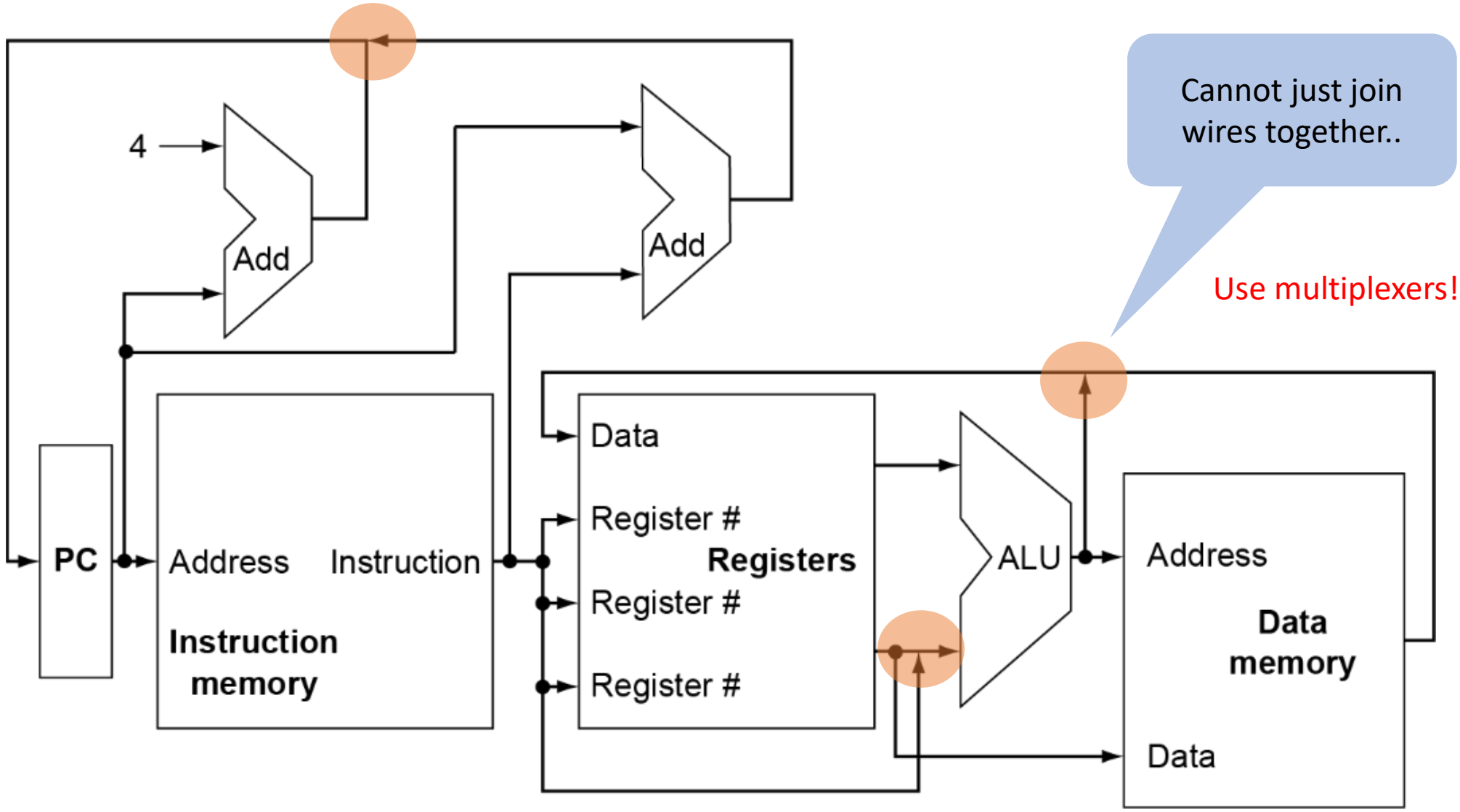


```
beq x5, x6, 100
```

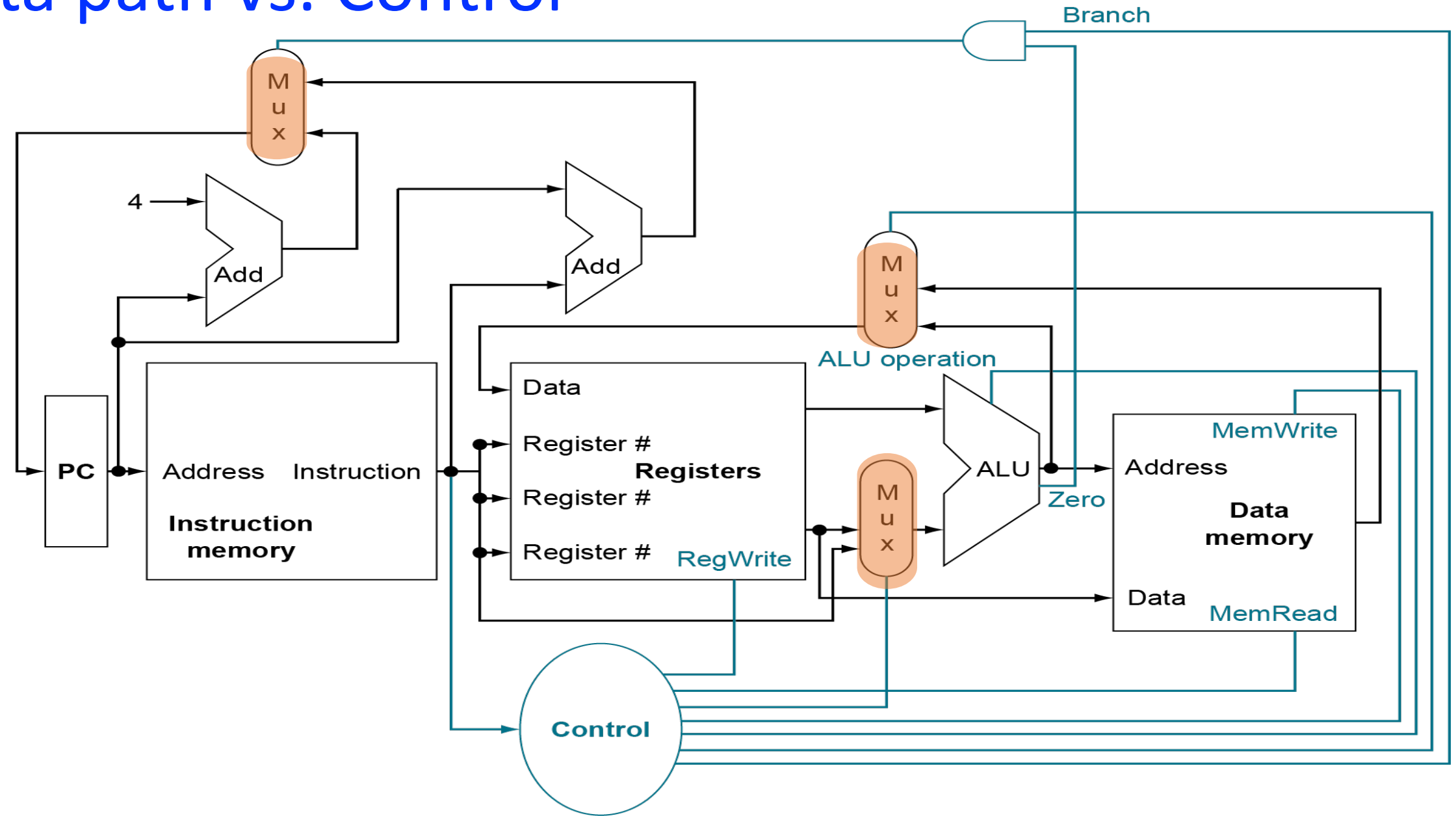
If (x5==x6) goto PC+2*100



Instruction Execution



Data path vs. Control

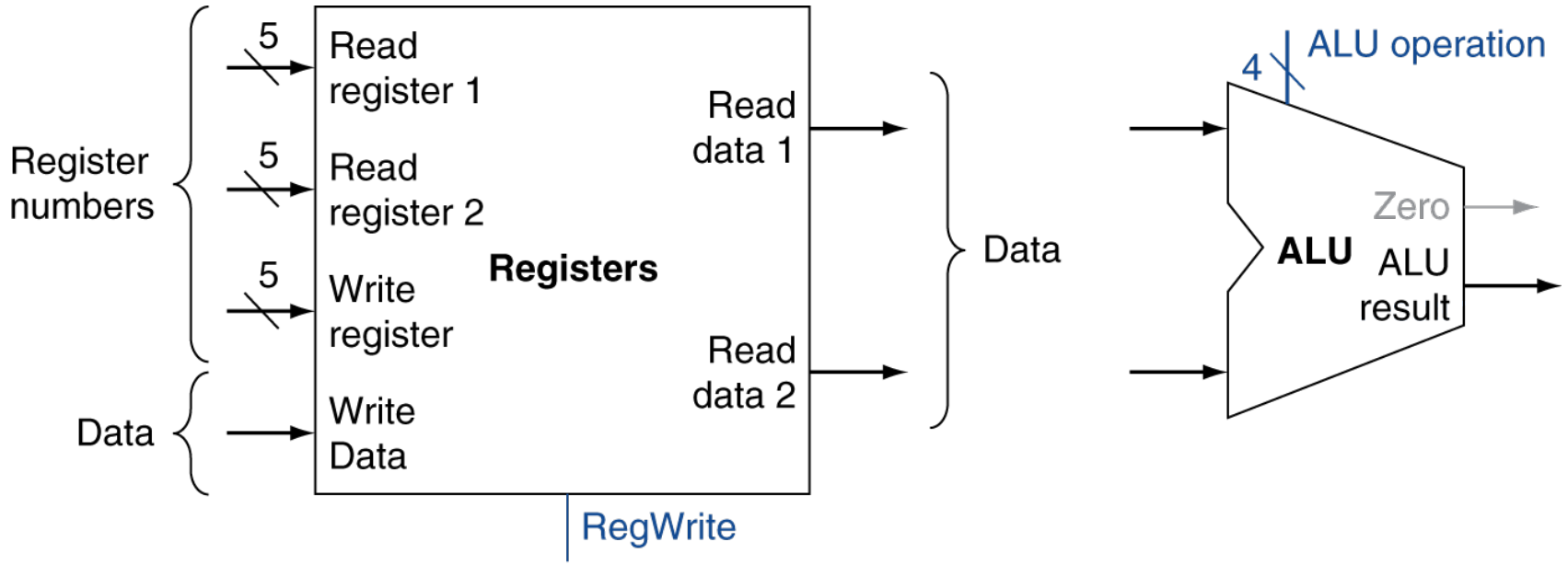


Building a Datapath

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

R-type Instructions

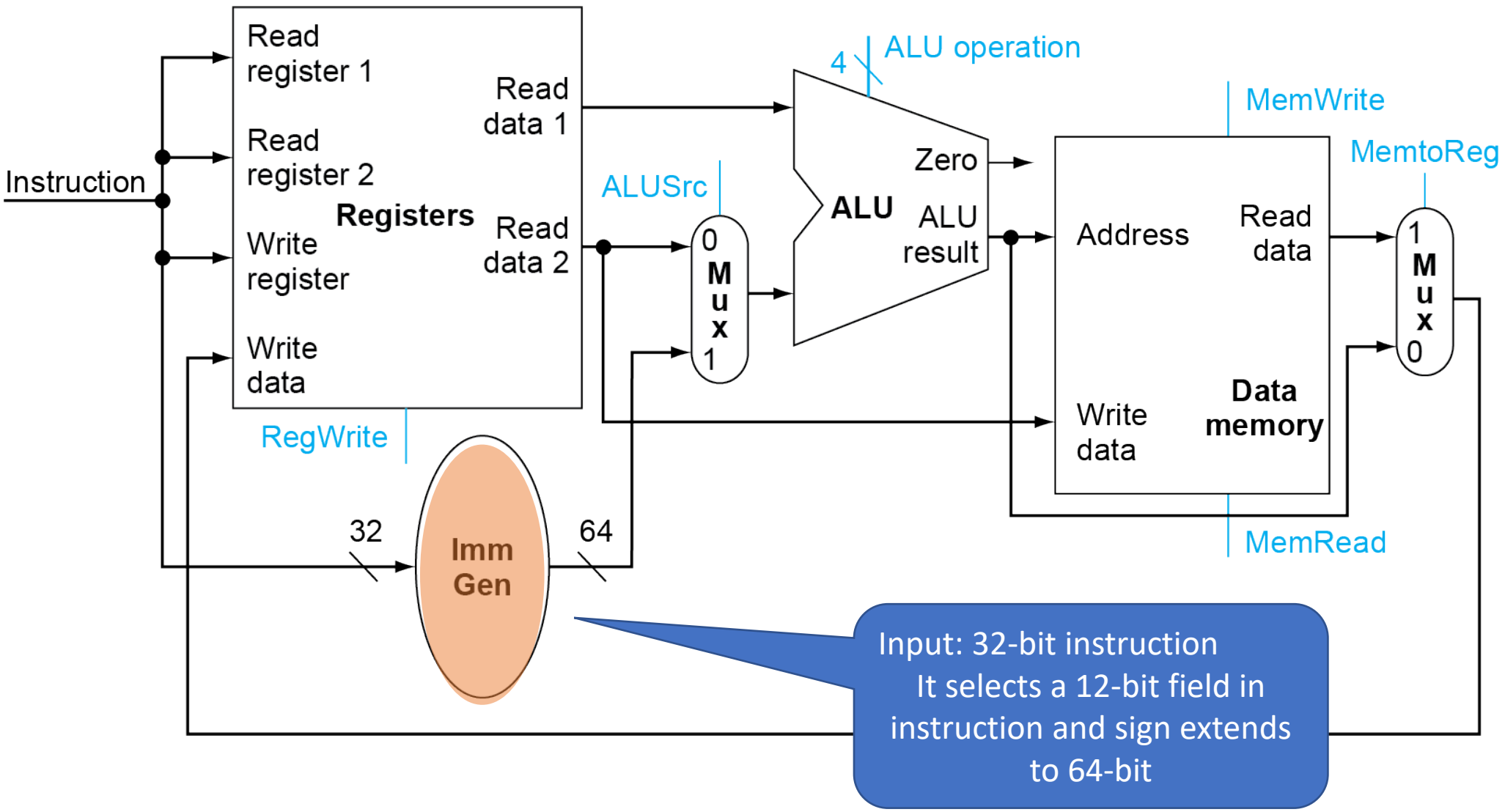
- Read two register operands
- Perform arithmetic/logical operation
- Write register result



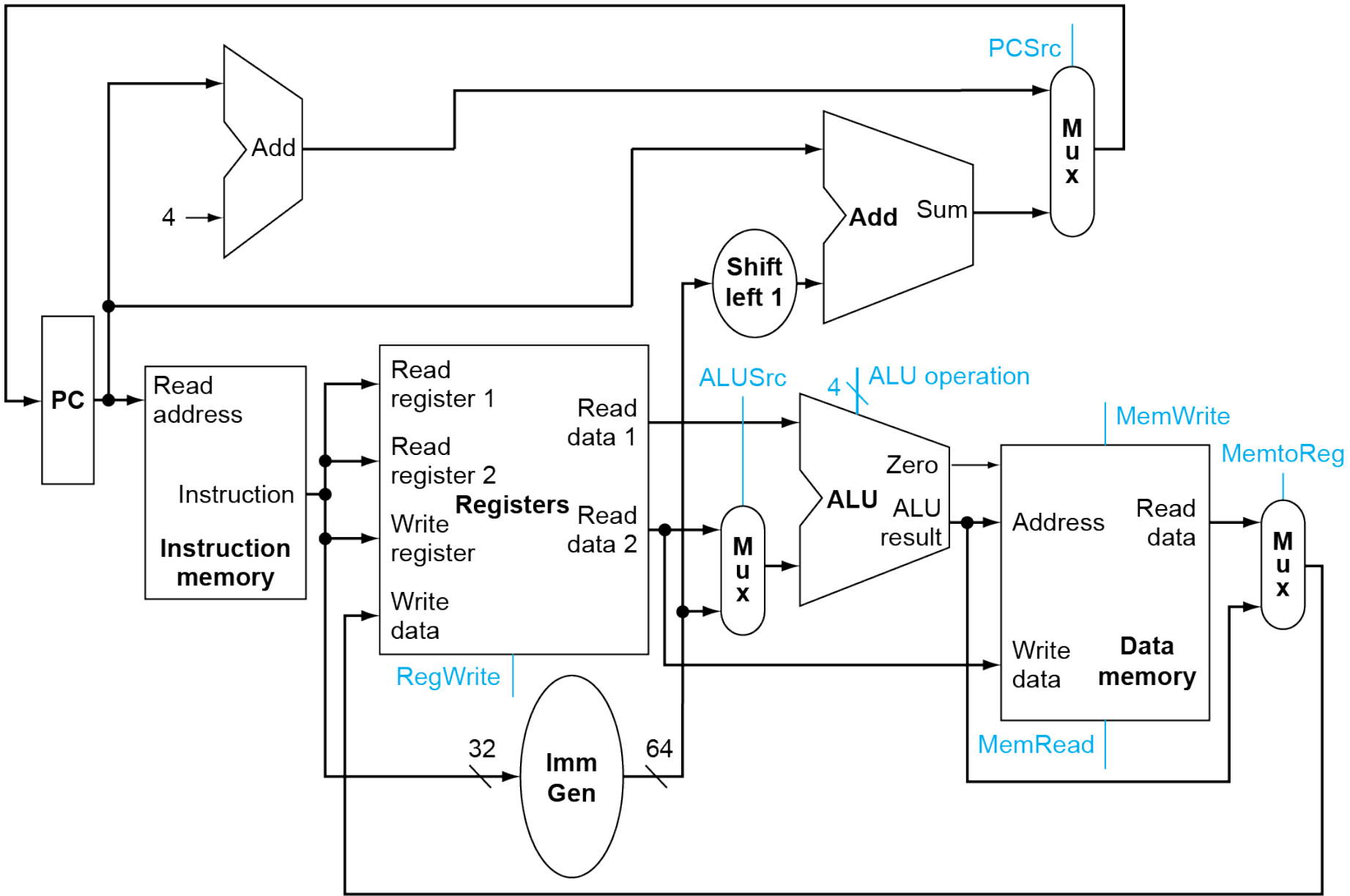
a. Registers

b. ALU

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

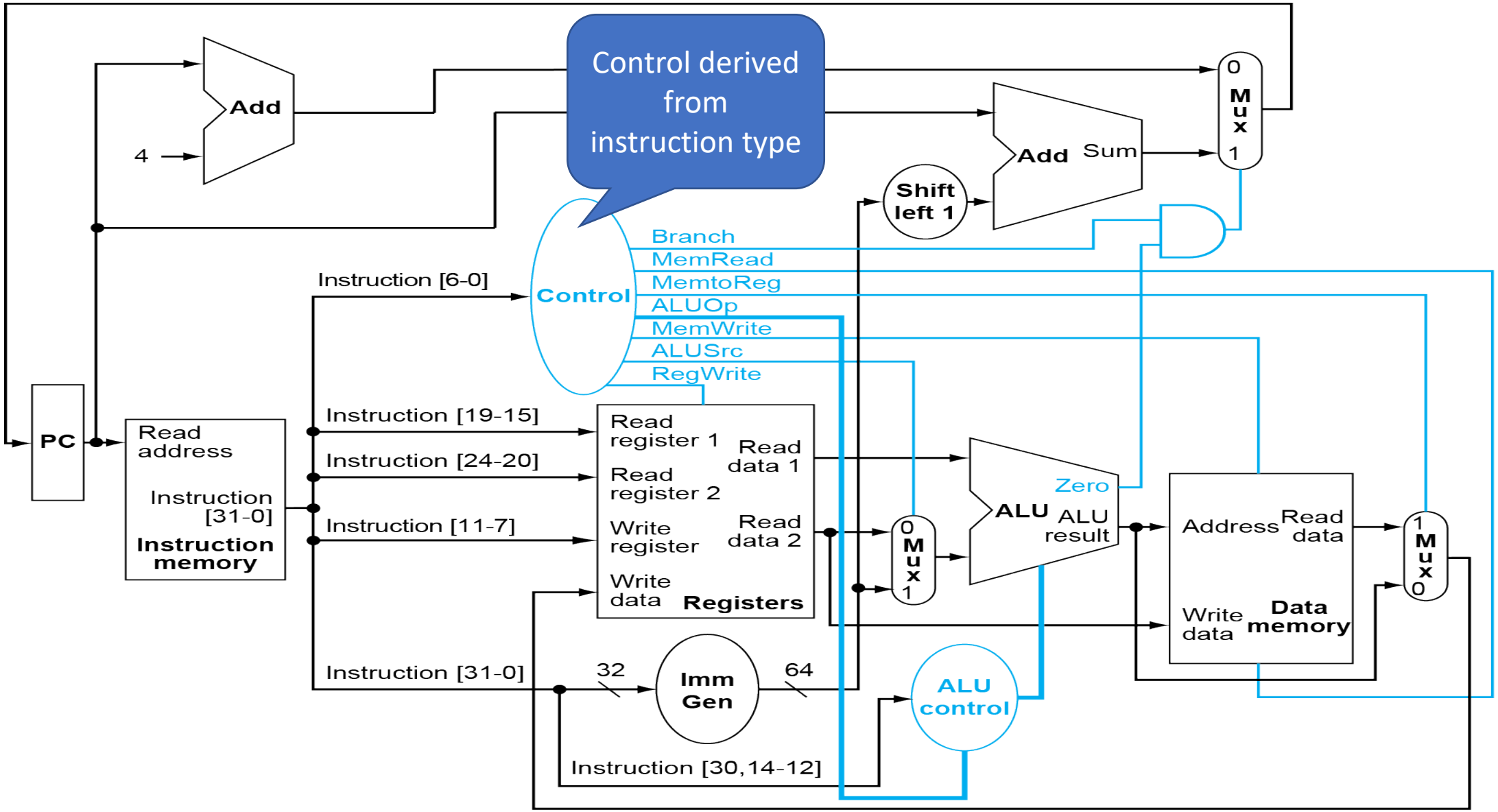
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

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

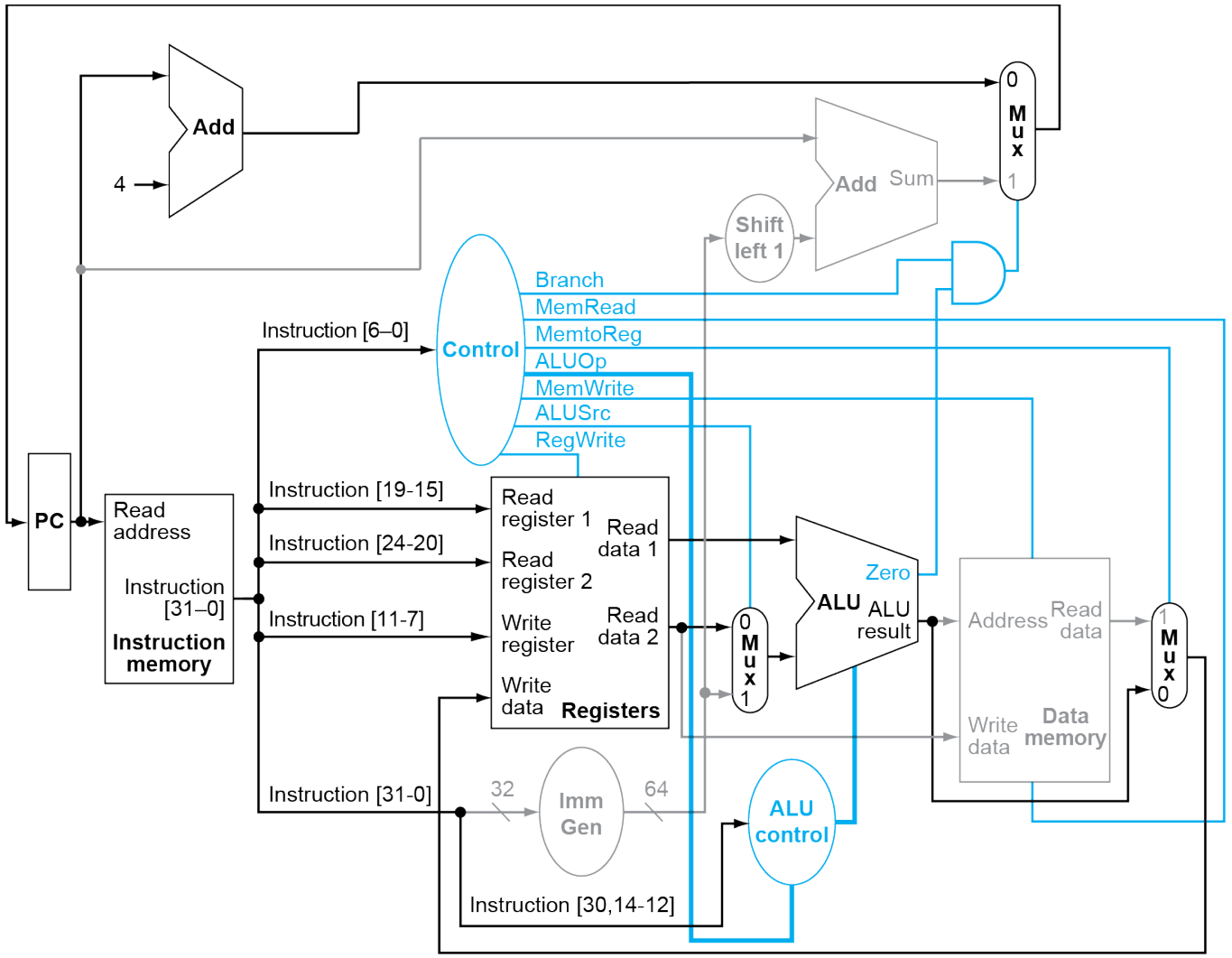
Datapath With Control



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

R-Type Inst

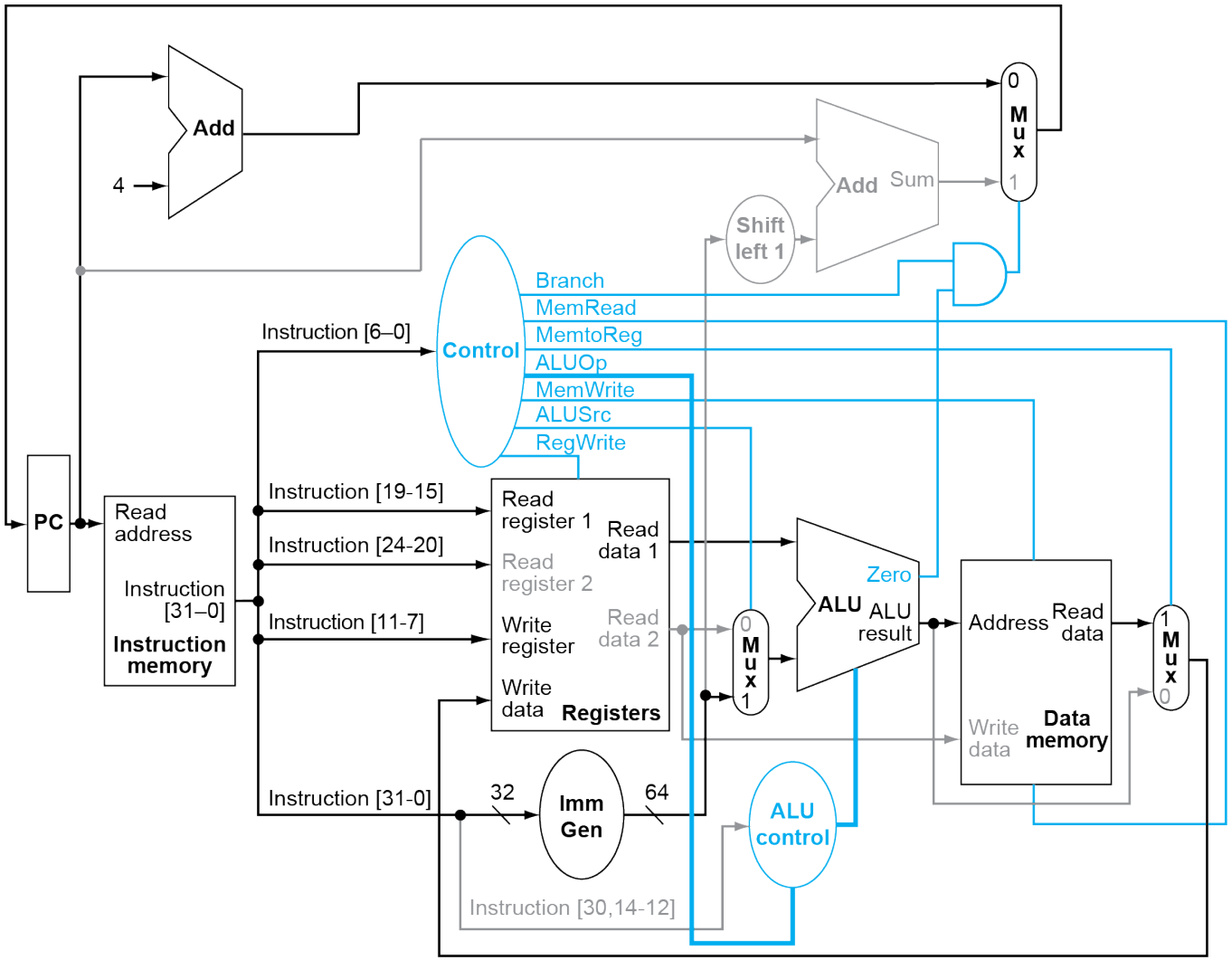
funct7	rs2	rs1	funct3	rd	opcode



Load Inst

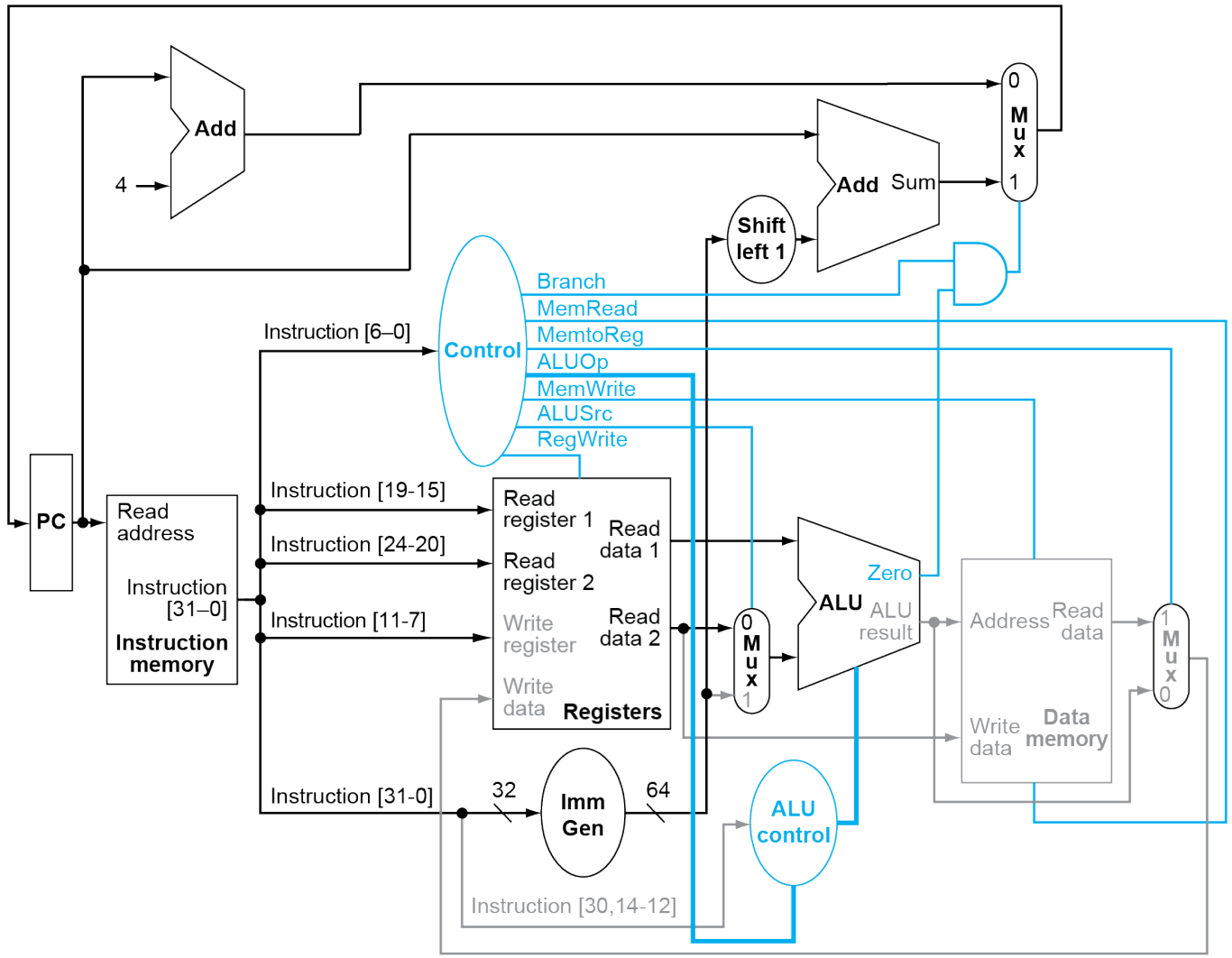
```
ld x5, 40(x6)
```

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