Chapter 4

The Processor

Introduction

- The performance of a computer is determined by three key factors:
  1. Instruction Count (IC)
  2. Clock Cycles per Instruction (CPI)
  3. Clock Cycle Time (T_c)

Determined by algorithm, ISA and Compiler

Determined by CPU H/W

$$\text{Execution Time (seconds/program)} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

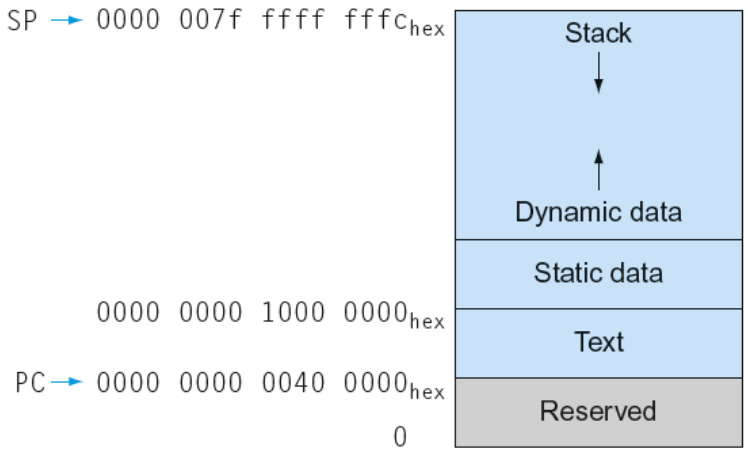
- Chapter 2 & 3
  - The compiler, algorithm and the instruction set architecture determine the instruction count required for a given program
- Chapter 4 :
  - How implementation of the processor determines both clock cycle time and the CPI.

Introduction

- We will examine two LEGv8 implementations
  - A simplified version (single-cycle datapath)
    - One instruction per cycle
    - Every instruction begins execution on one clock edge and complete execution on the next clock edge
    - Not efficient (why?)
  - A more realistic pipelined version
    - Designing instruction set for pipelining
    - Pipelined datapath and control
    - Pipeline hazards
- Simple ISA subset, shows most aspects
    - Memory reference: `LDUR`, `STUR`
    - Arithmetic/logical: `ADD`, `SUB`, `AND`, `OR`
    - Control transfer: `CBNZ`, `B`

Instruction Execution

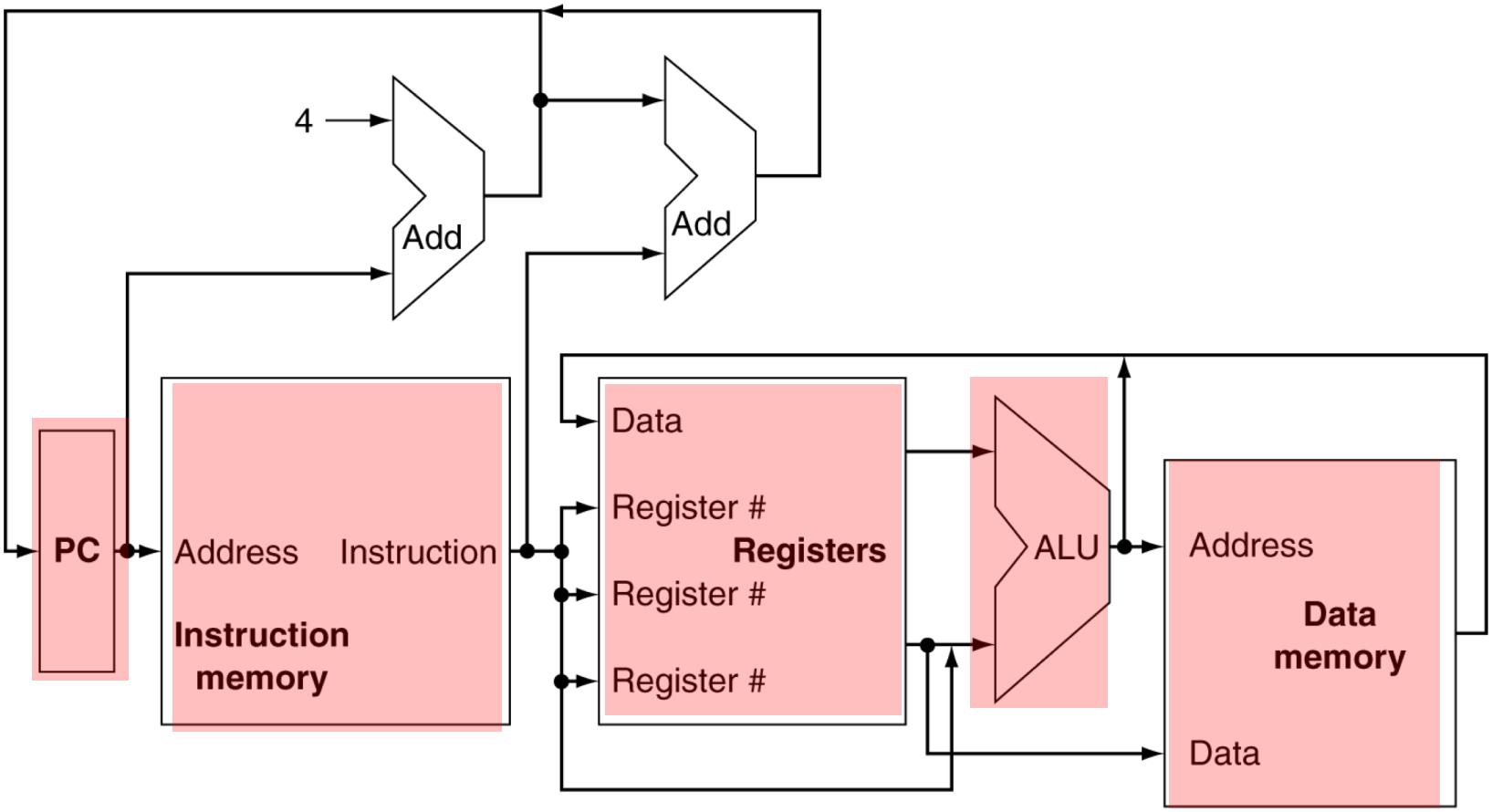
- PC → instruction in memory, and fetch it
- Register numbers → register file, read registers



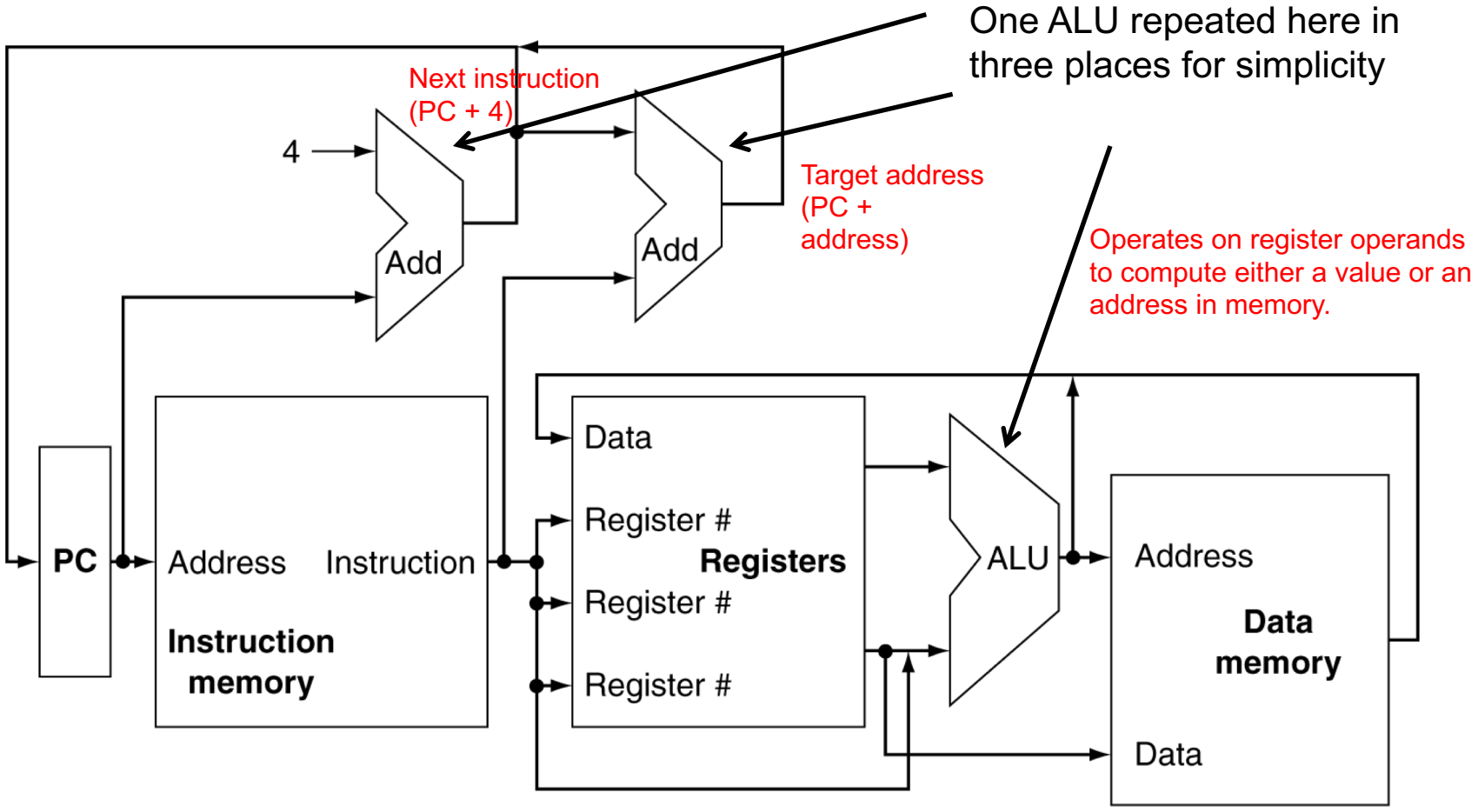
Instruction Execution

- Depending on instruction type
  - Use ALU to:
    - Perform Arithmetic result (ADD X0, X1, X2)
    - Calculate the memory address for load/store using base and offset addresses (LDUR X0, [X2, #4])
    - Calculate the branch target address (CBZ X0, exit)
      - Calculate PC + target address or PC + 4
        - For branches after comparison or simply go to the next instruction address (PC + 4) in unconditional B

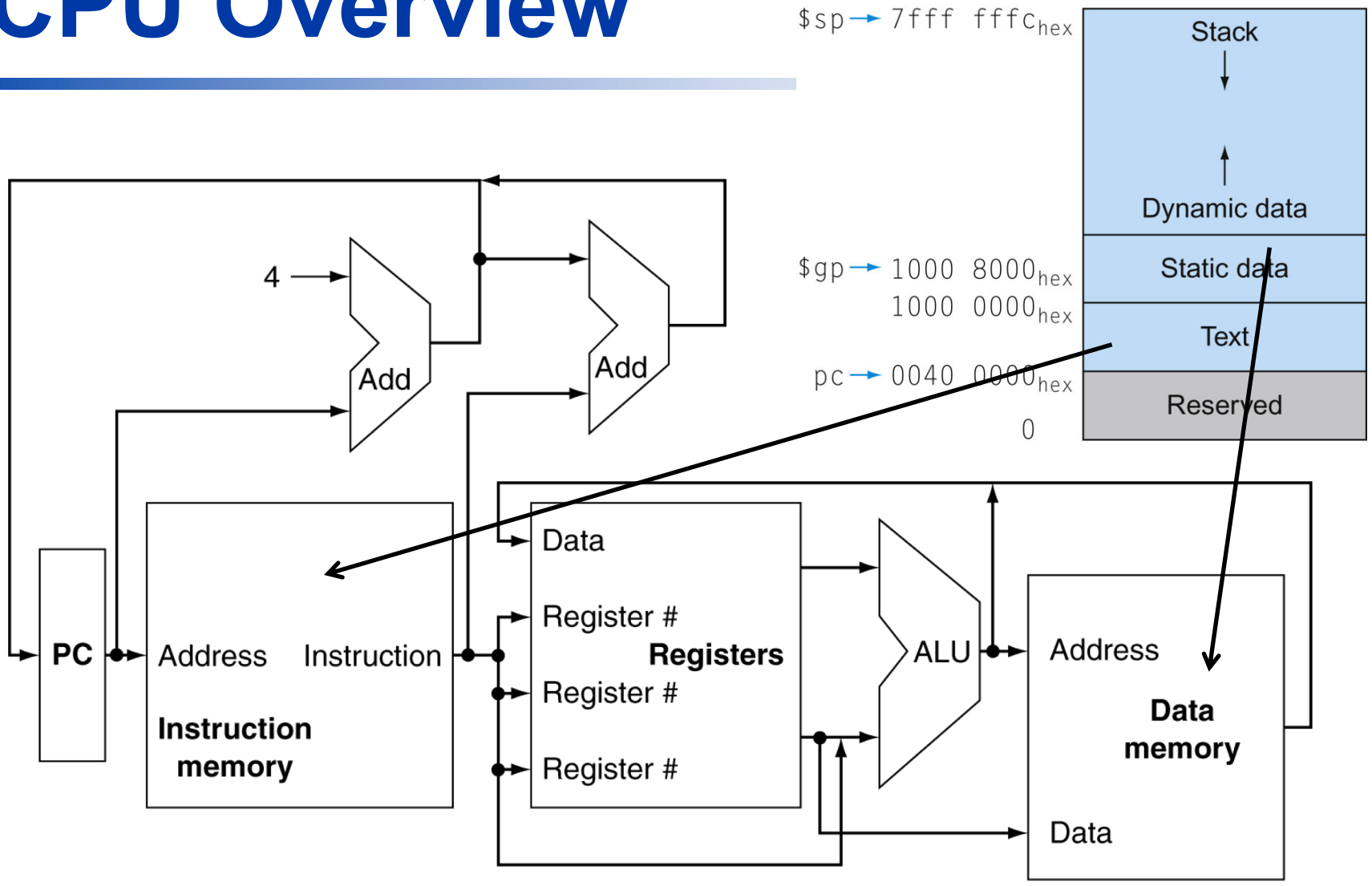
CPU Overview



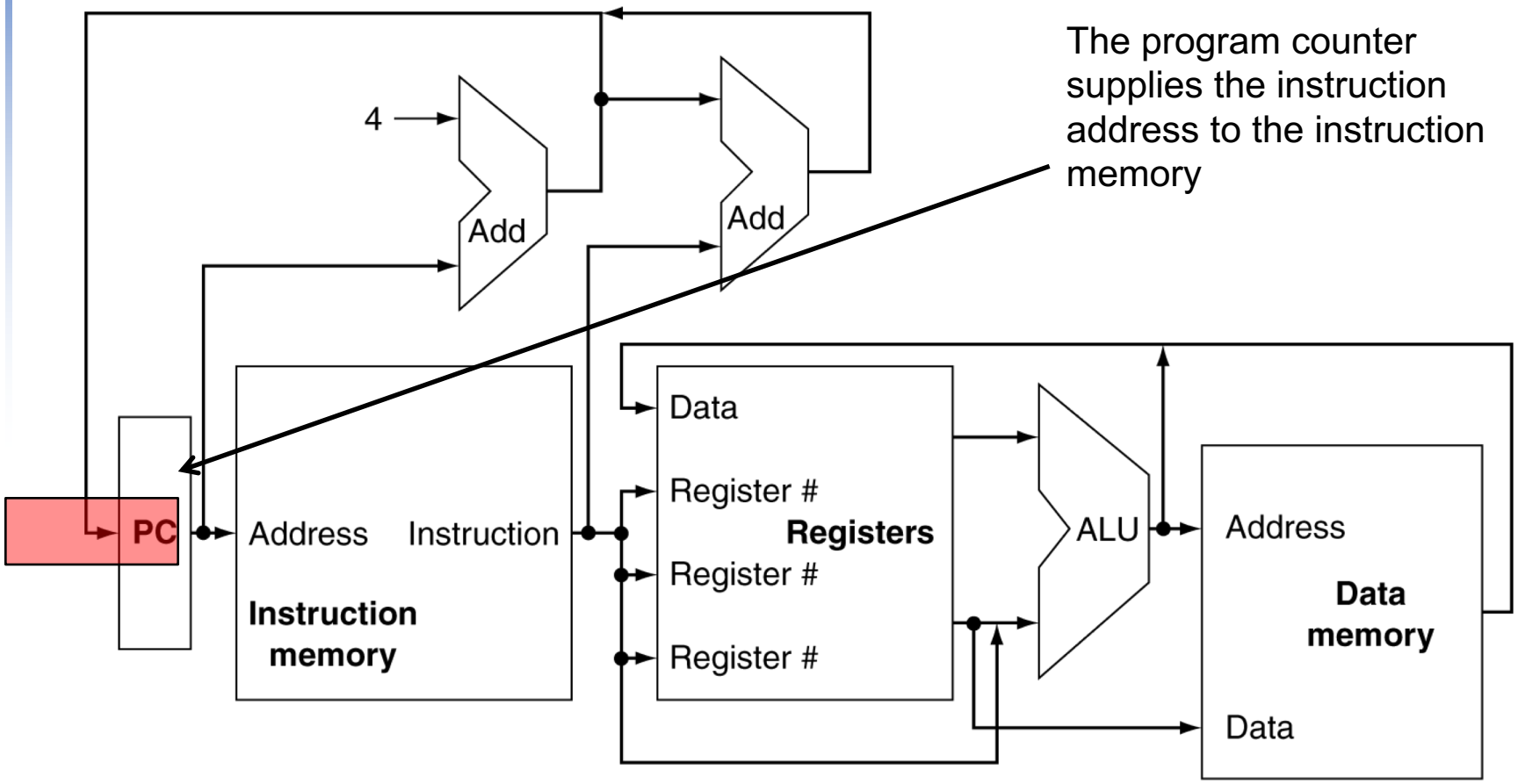
CPU Overview



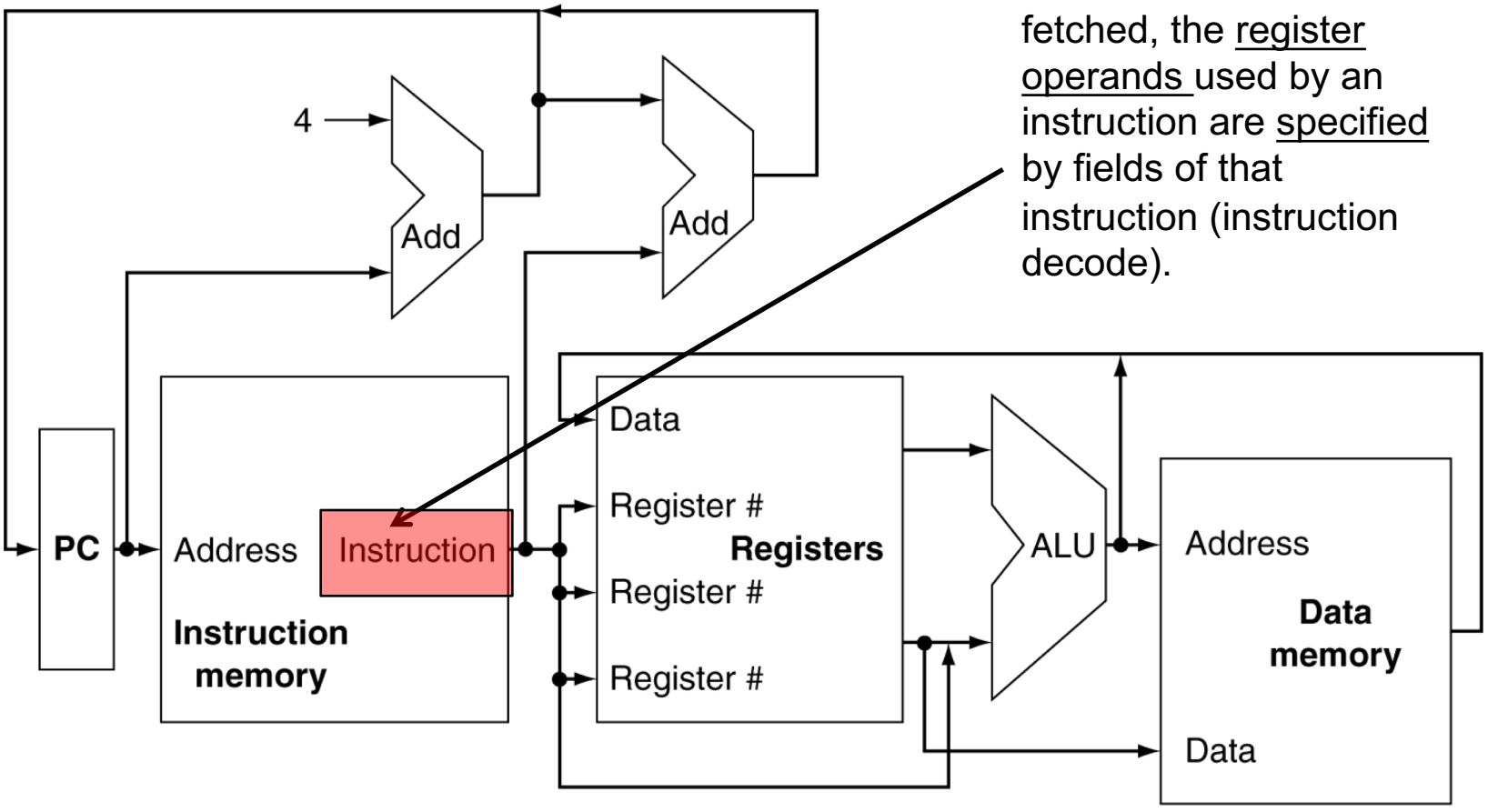
CPU Overview



CPU Overview



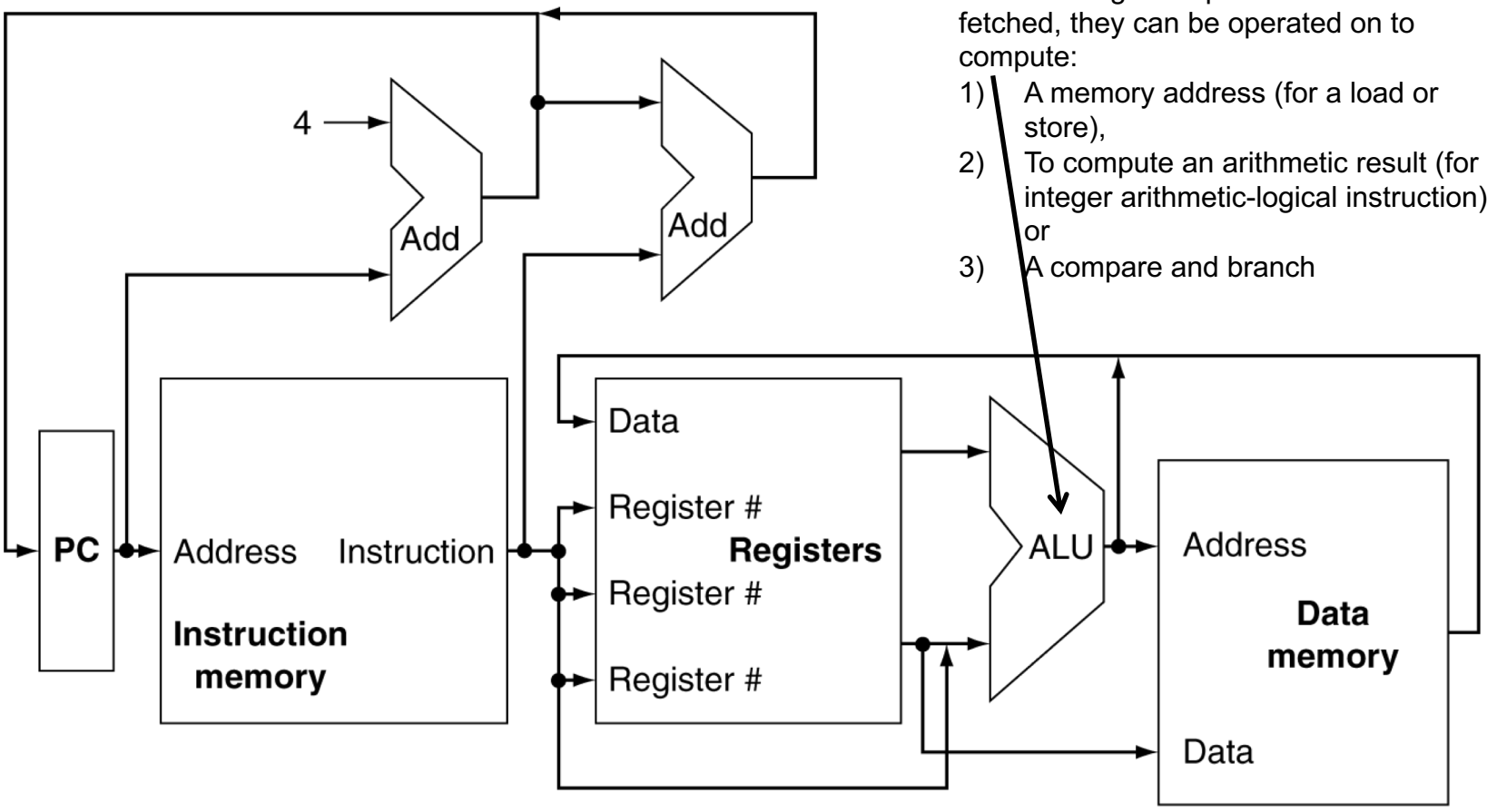
CPU Overview



After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction (instruction decode).

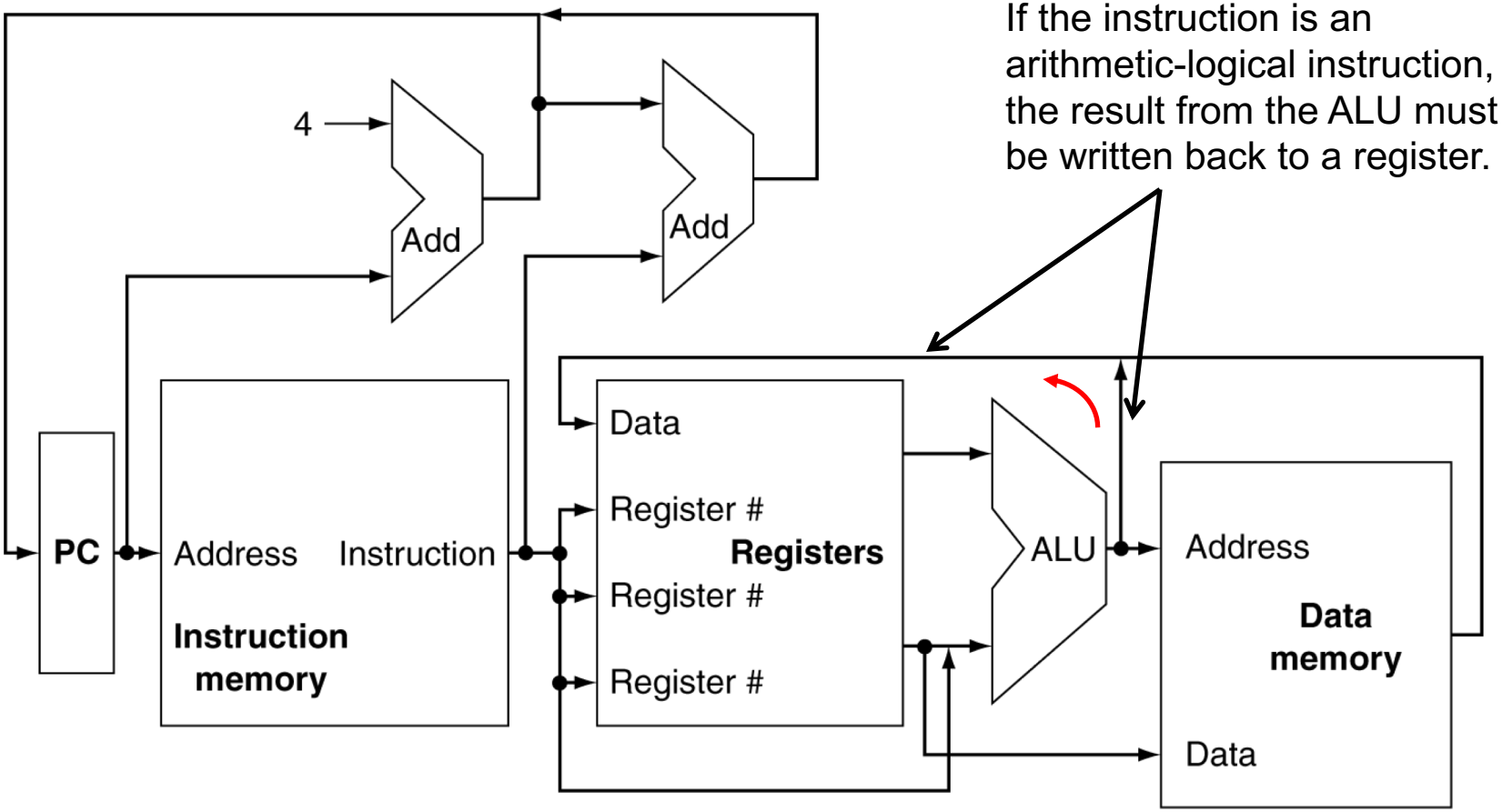
LDUR X9,[X22,#64]

CPU Overview



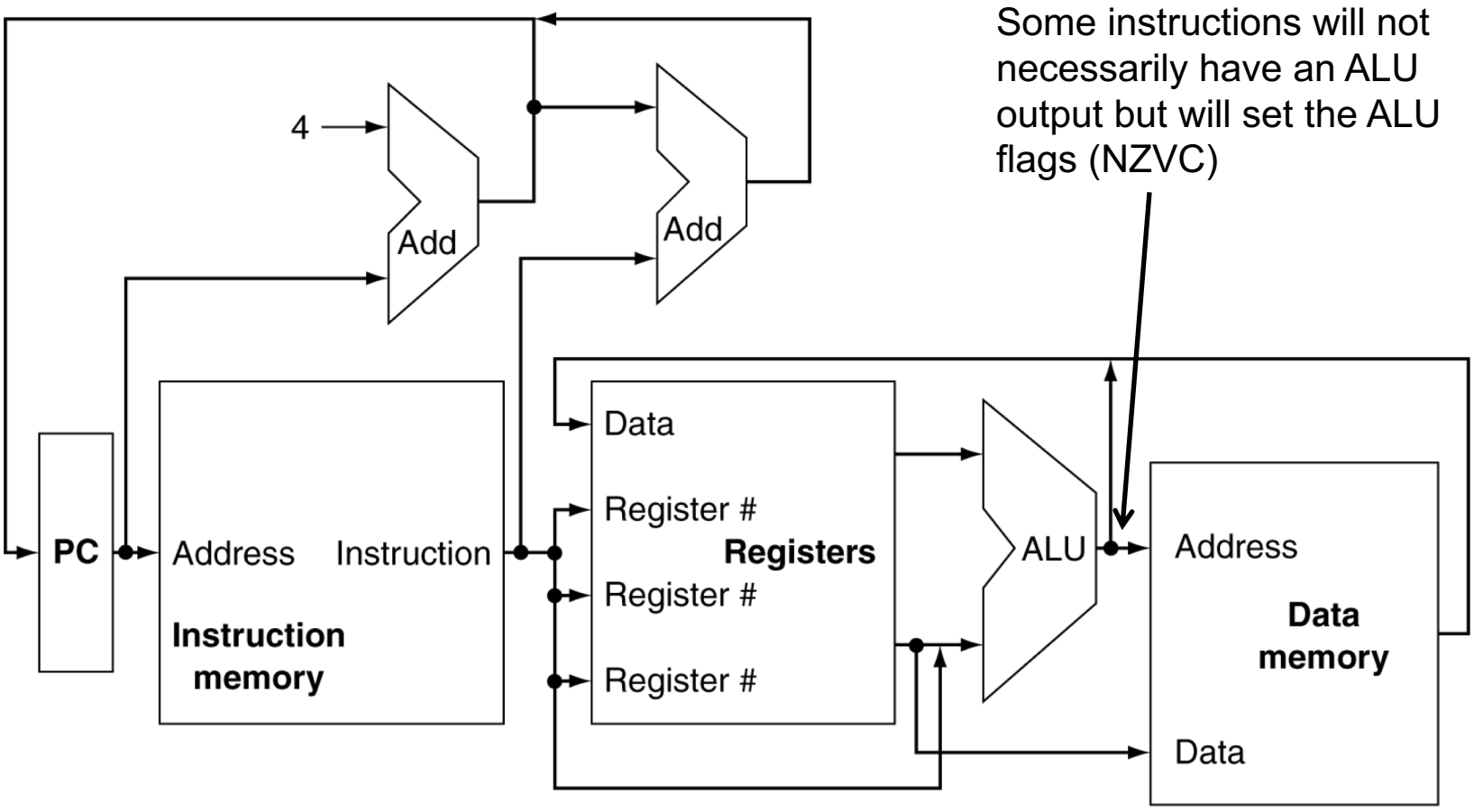

```
1) LDUR   X9,[X22,#64]
2) ADD    X0, X1, X2
3) CBZ    X0, L1
```

CPU Overview




```
ADD  X0, X1, X2
OR   X9, X10, X11
```

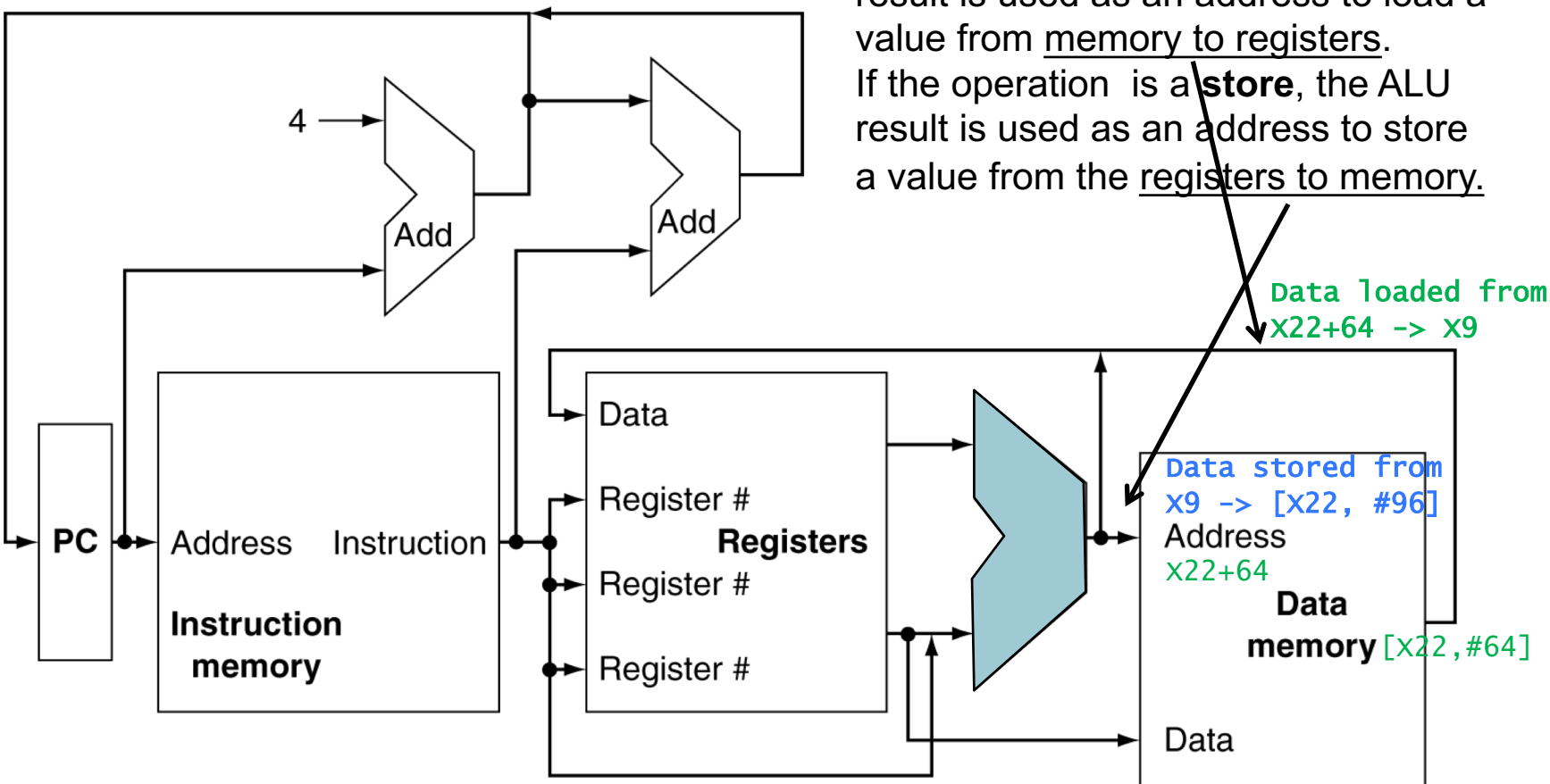
CPU Overview



CMP X9, X10

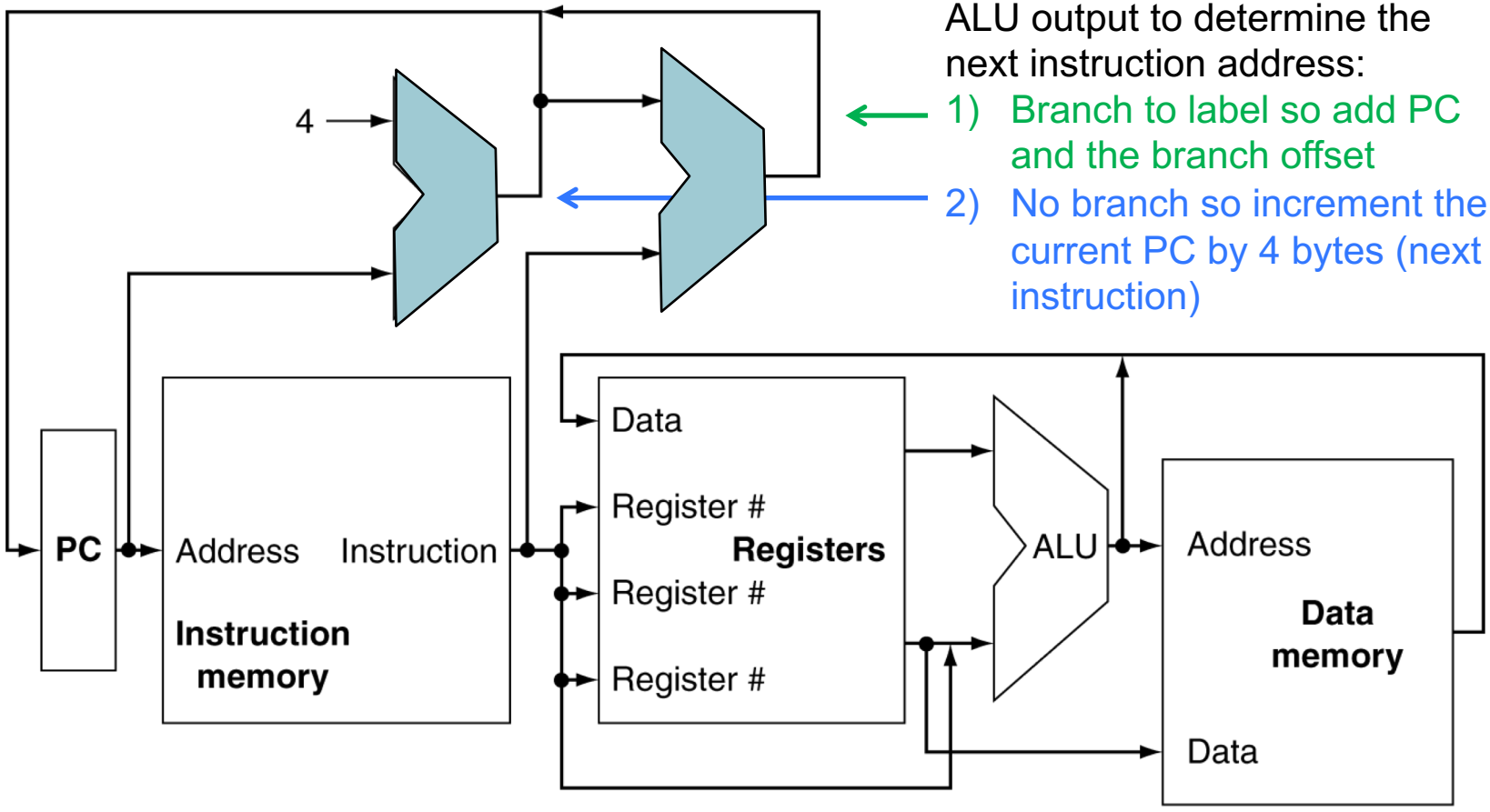
CPU Overview

If the operation is a **load**, the ALU result is used as an address to load a value from memory to registers.
If the operation is a **store**, the ALU result is used as an address to store a value from the registers to memory.




```
LDUR   X9,[X22,#64]
STUR   X9,[X22,#96]
```

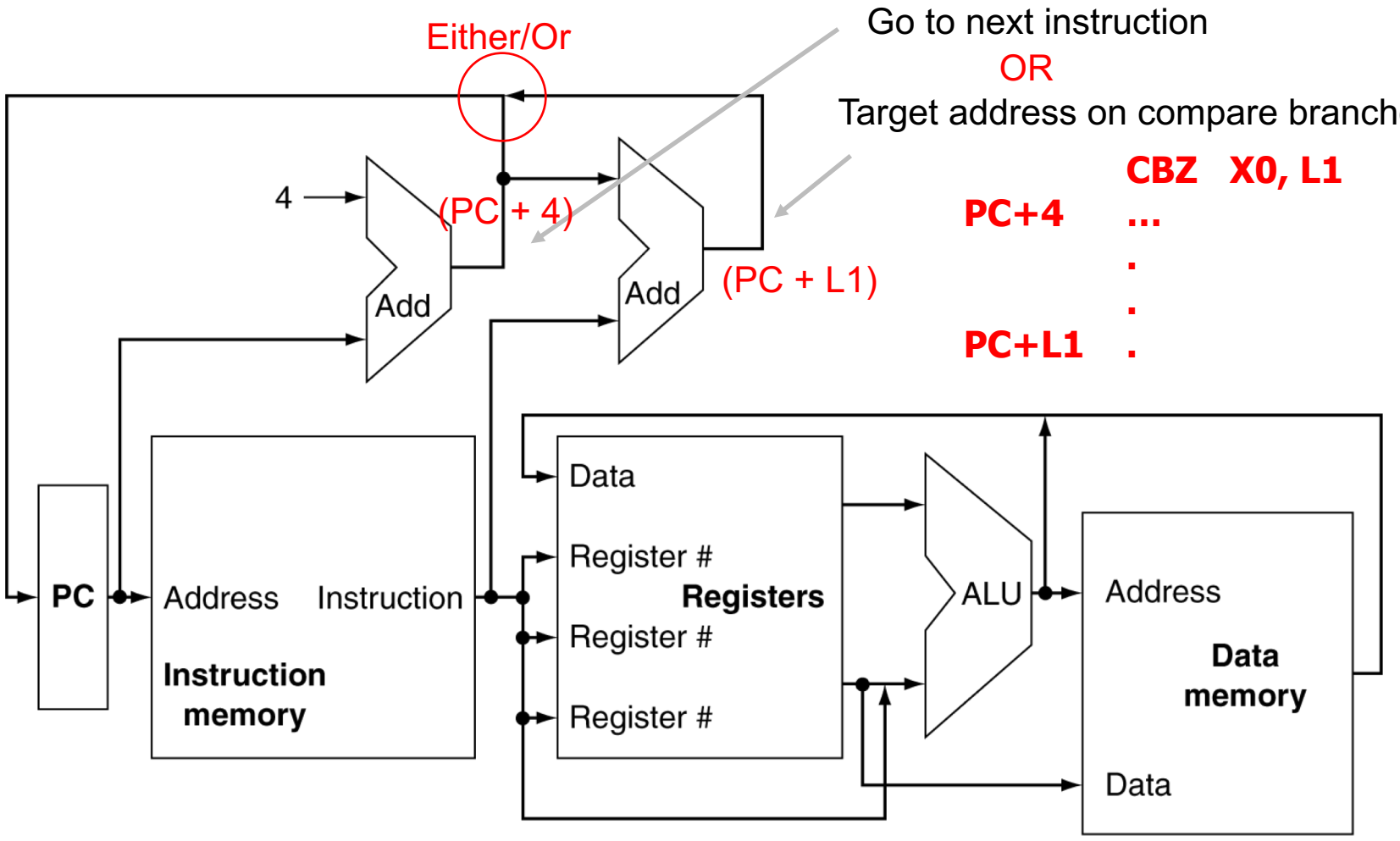
CPU Overview



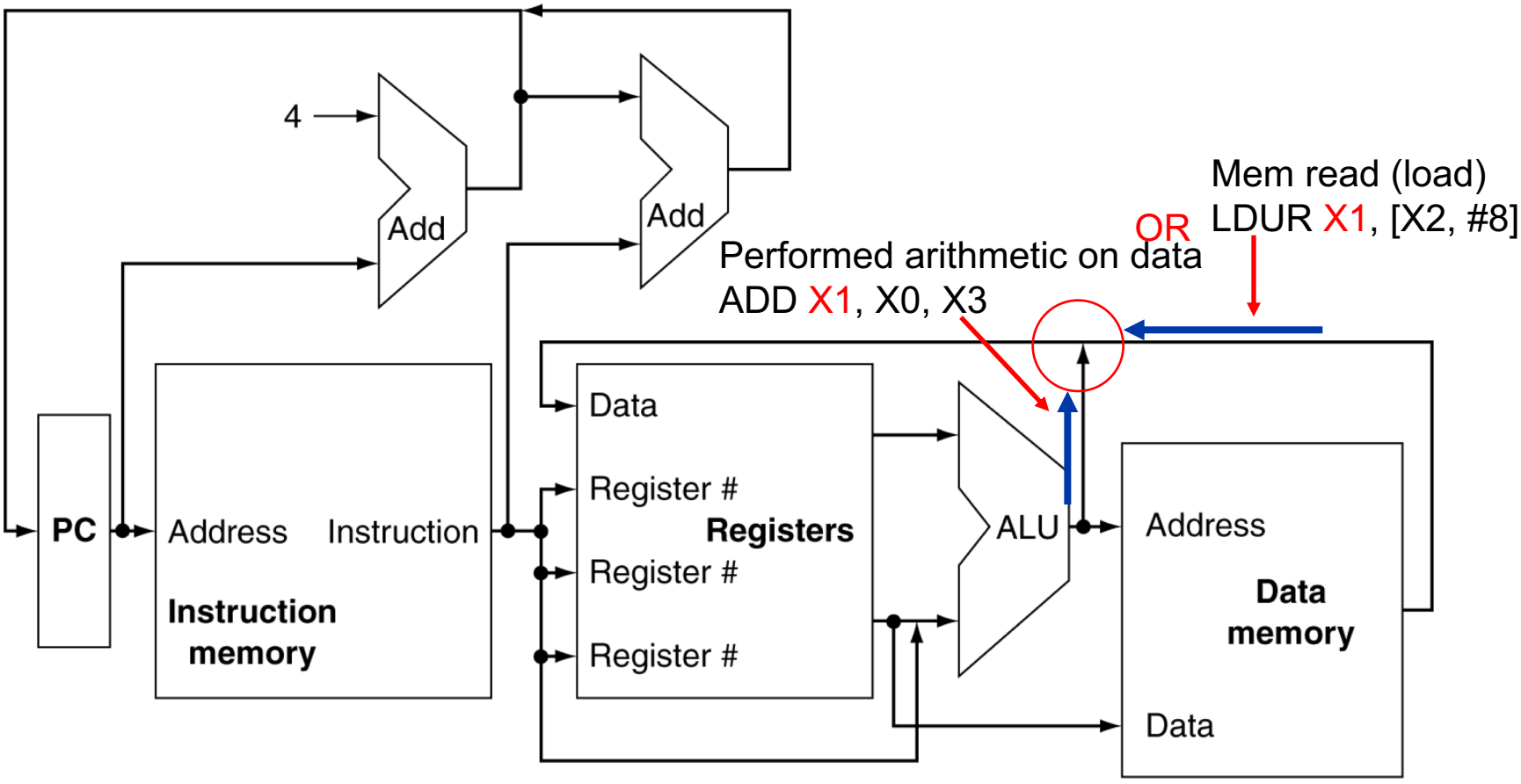
Branches require the use of the ALU output to determine the next instruction address:

1) Branch to label so add PC and the branch offset
2) No branch so increment the current PC by 4 bytes (next instruction)

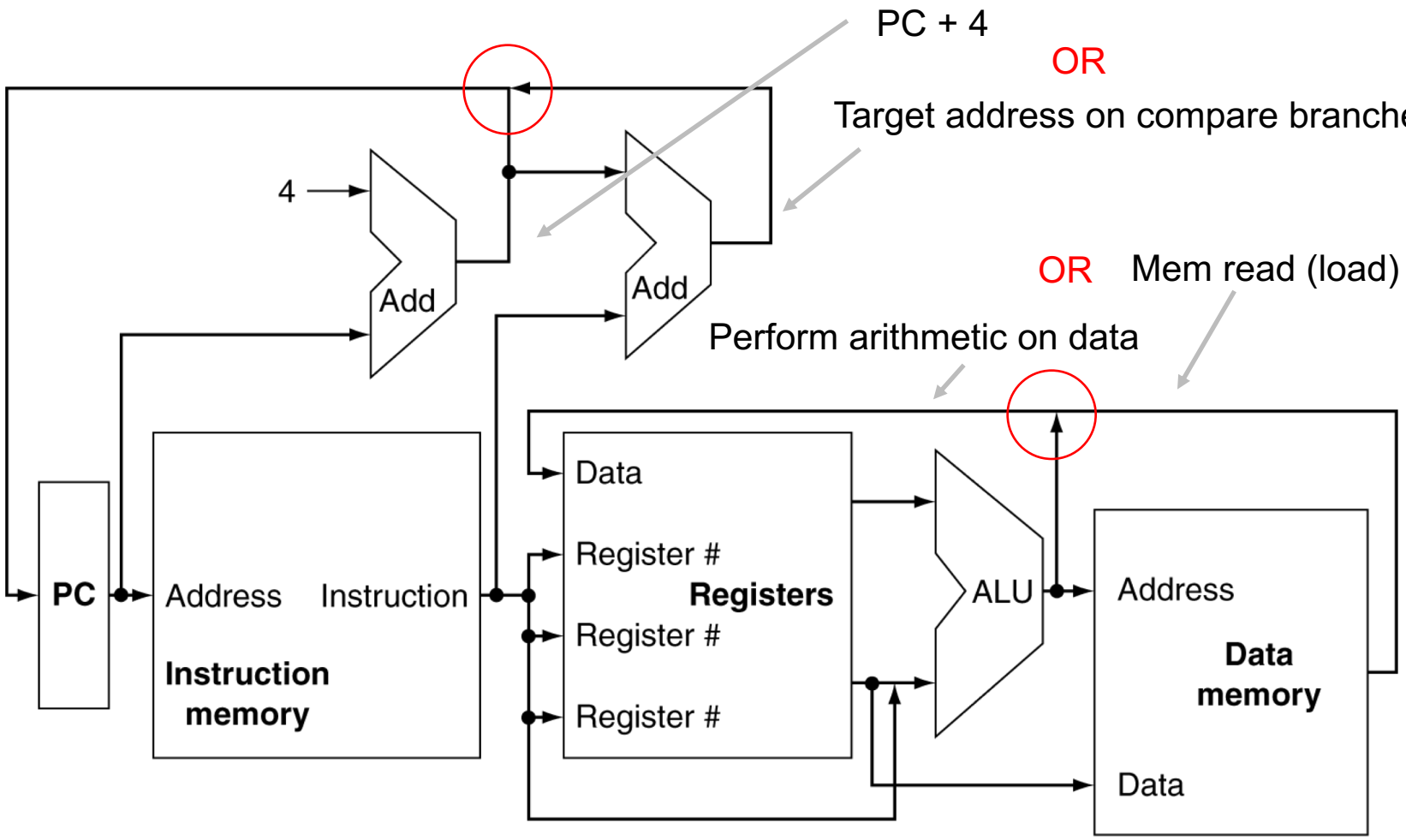
CPU Overview



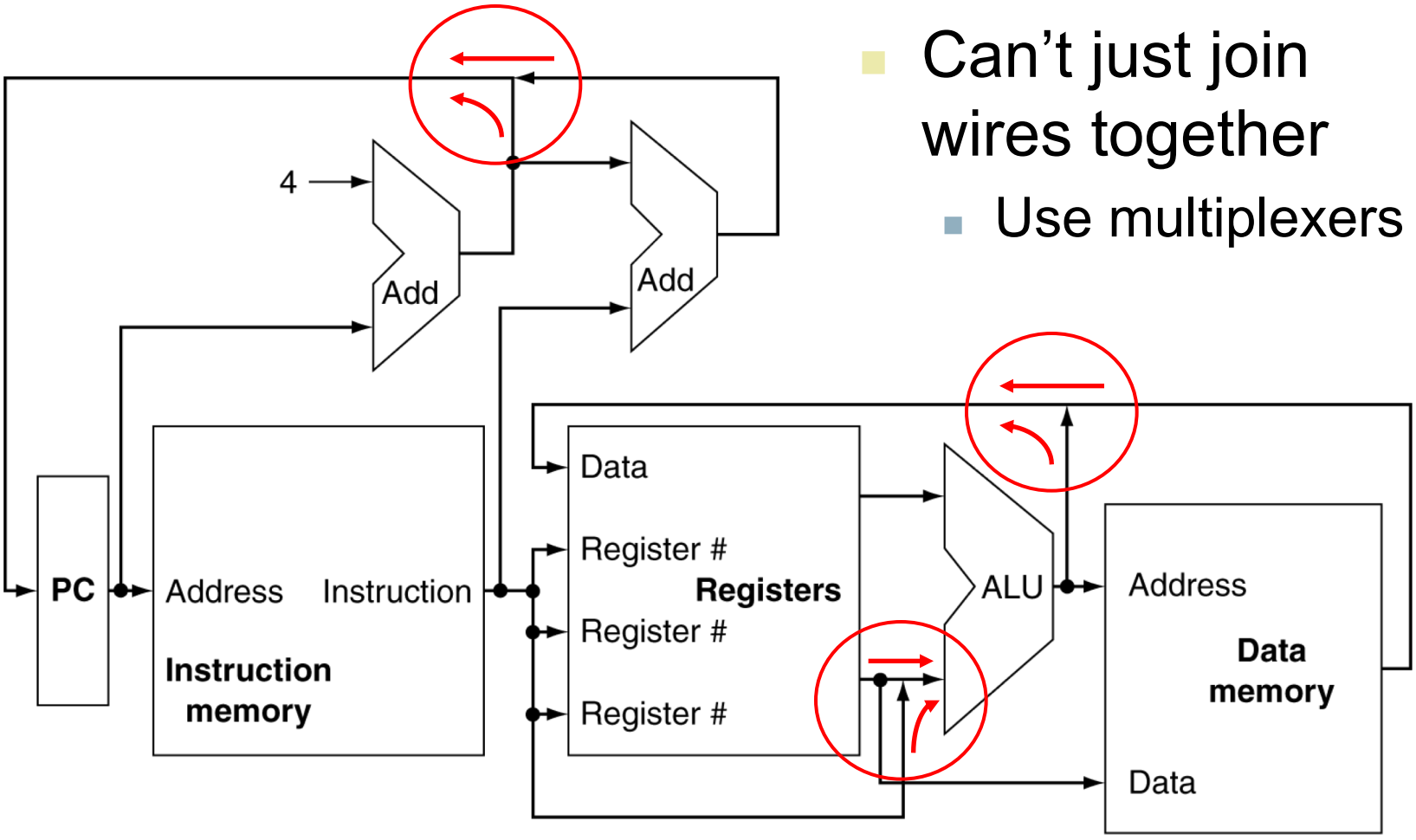
CPU Overview



CPU Overview

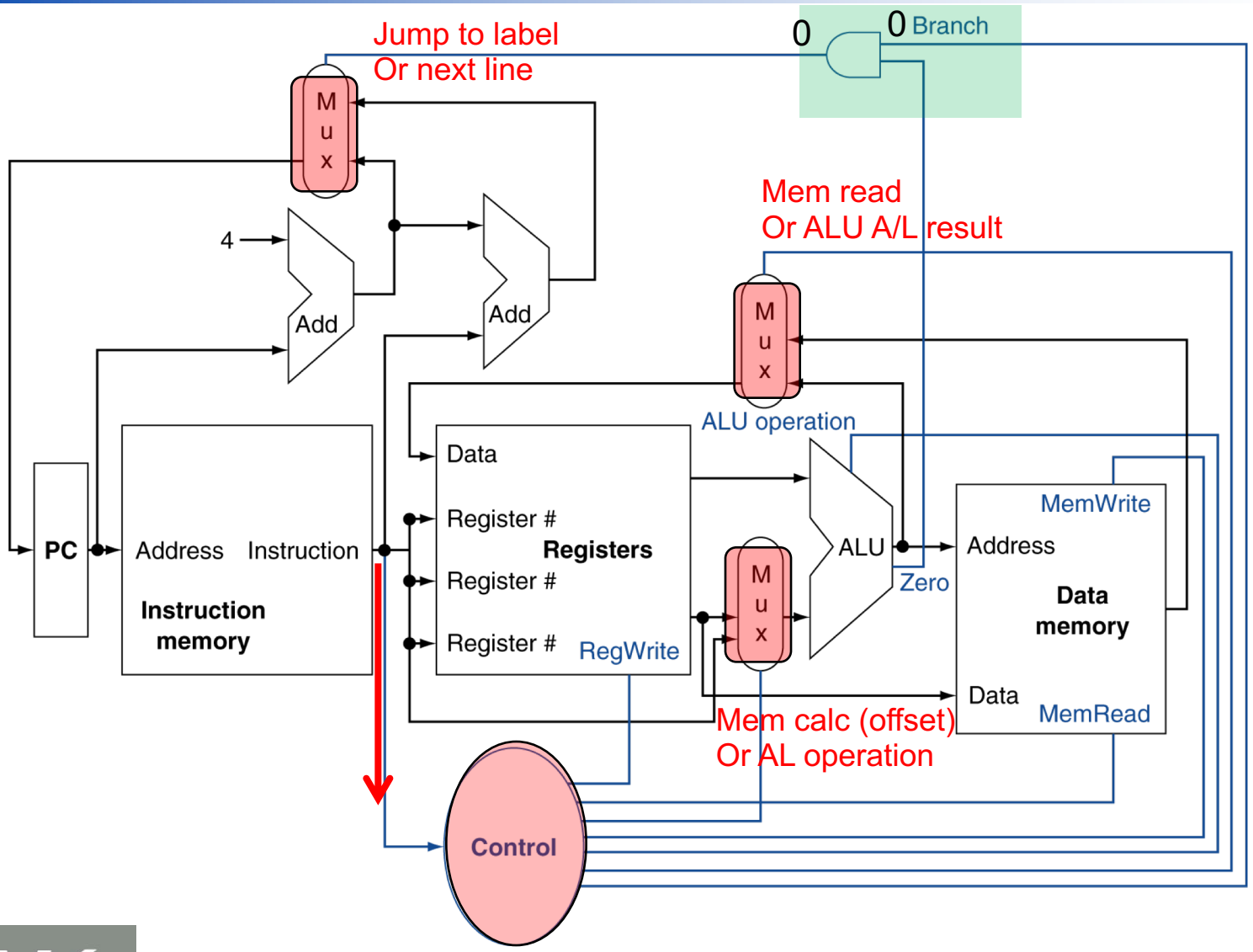


Multiplexers

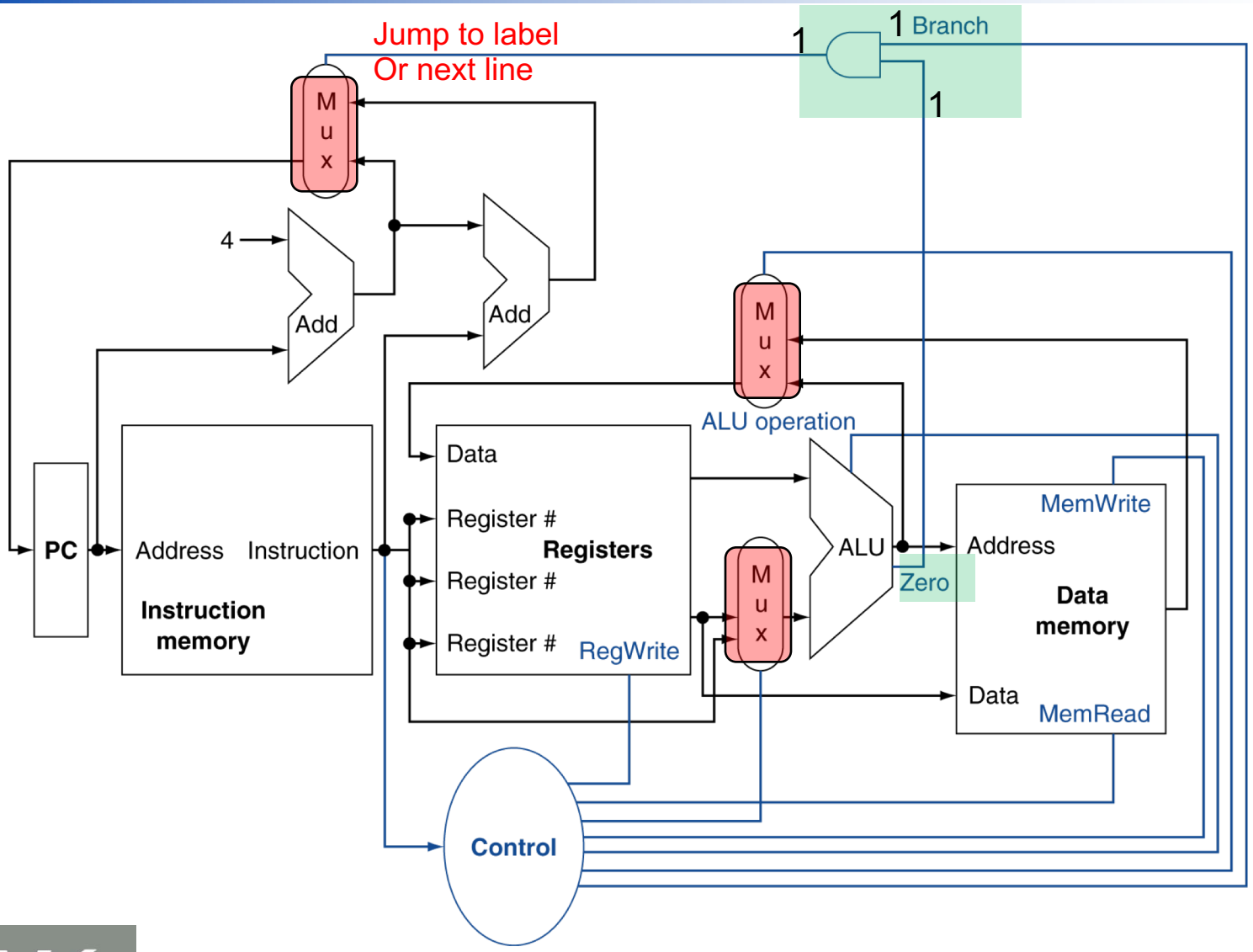


- Can't just join wires together
  - Use multiplexers

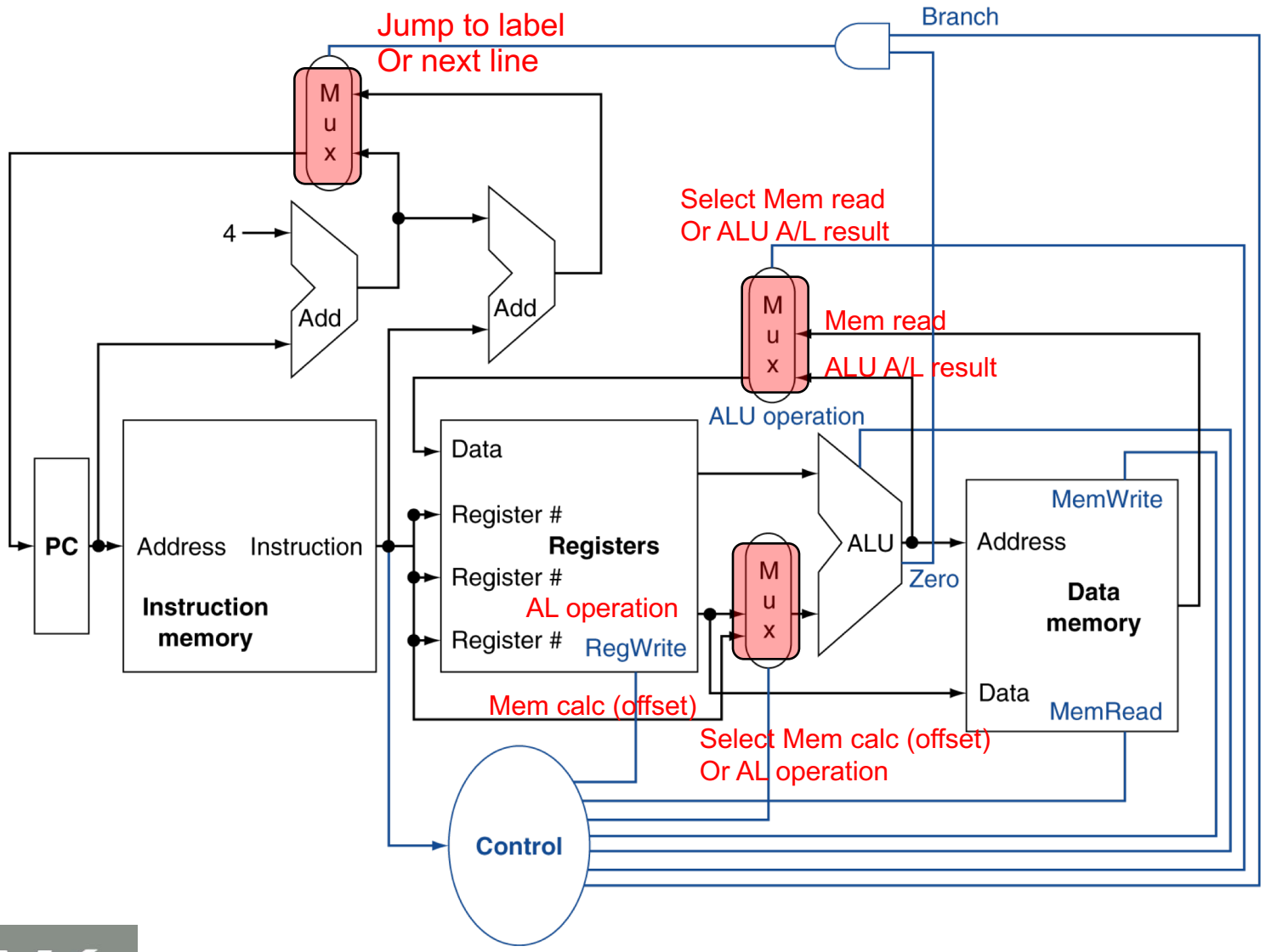
Control


Jump to label
Or next line
0
0 Branch
M
u
x
4
Add
Add
Mem read
Or ALU A/L result
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Mem calc (offset)
Or AL operation
Control

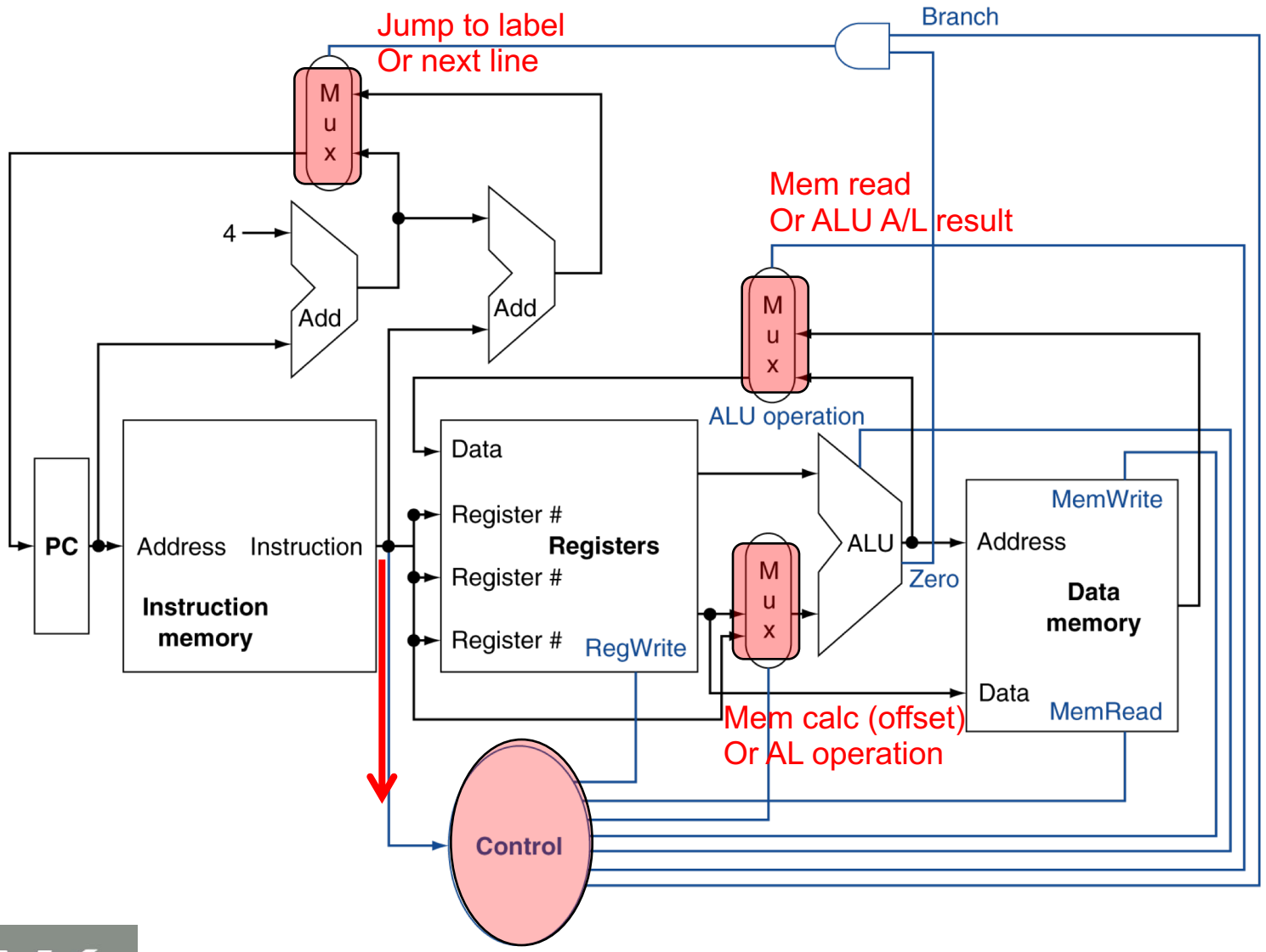
Control


Jump to label
Or next line
1
1 Branch
1
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Control


Jump to label
Or next line
Branch
M
u
x
4
Add
Add
Select Mem read
Or ALU A/L result
M
u
x
Mem read
ALU A/L result
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
AL operation
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Mem calc (offset)
Select Mem calc (offset)
Or AL operation
Control

Control


Jump to label
Or next line
Branch
M
u
x
4
Add
Add
Mem read
Or ALU A/L result
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Mem calc (offset)
Or AL operation
Control

Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a LEGv8 datapath incrementally
  - Refining the overview design

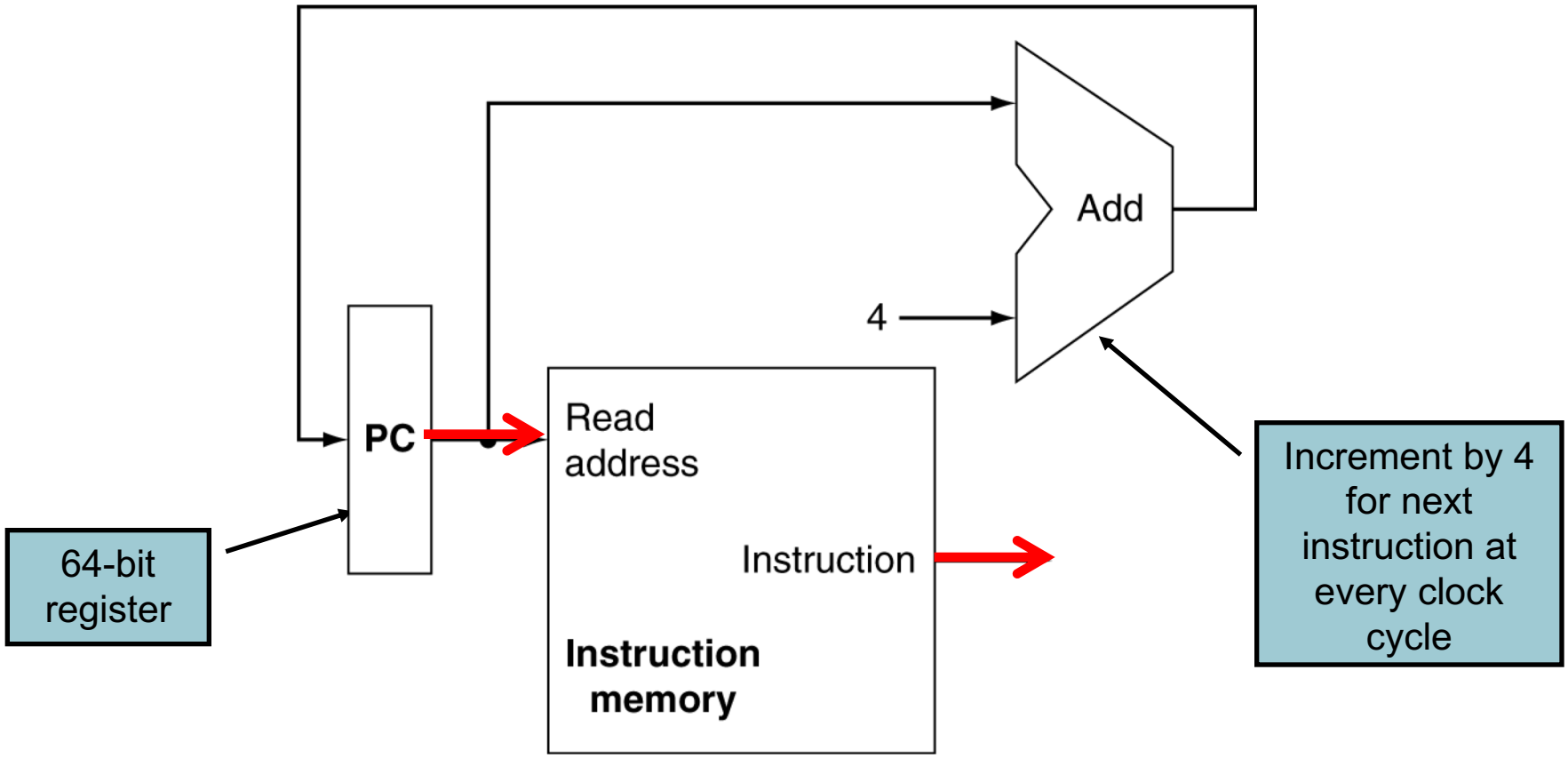
LEGv8 Instruction Formats

LEGv8

Name	Format	Example					Comments
ADD	R	1112	3	0	2	1	ADD X1, X2, X3
SUB	R	1624	3	0	2	1	SUB X1, X2, X3
ADDI	I	580	100		2	1	ADDI X1, X2, #100
SUBI	I	836	100		2	1	SUBI X1, X2, #100
LDUR	D	1986	100	0	2	1	LDUR X1, [X2, #100]
STUR	D	1984	100	0	2	1	STUR X1, [X2, #100]

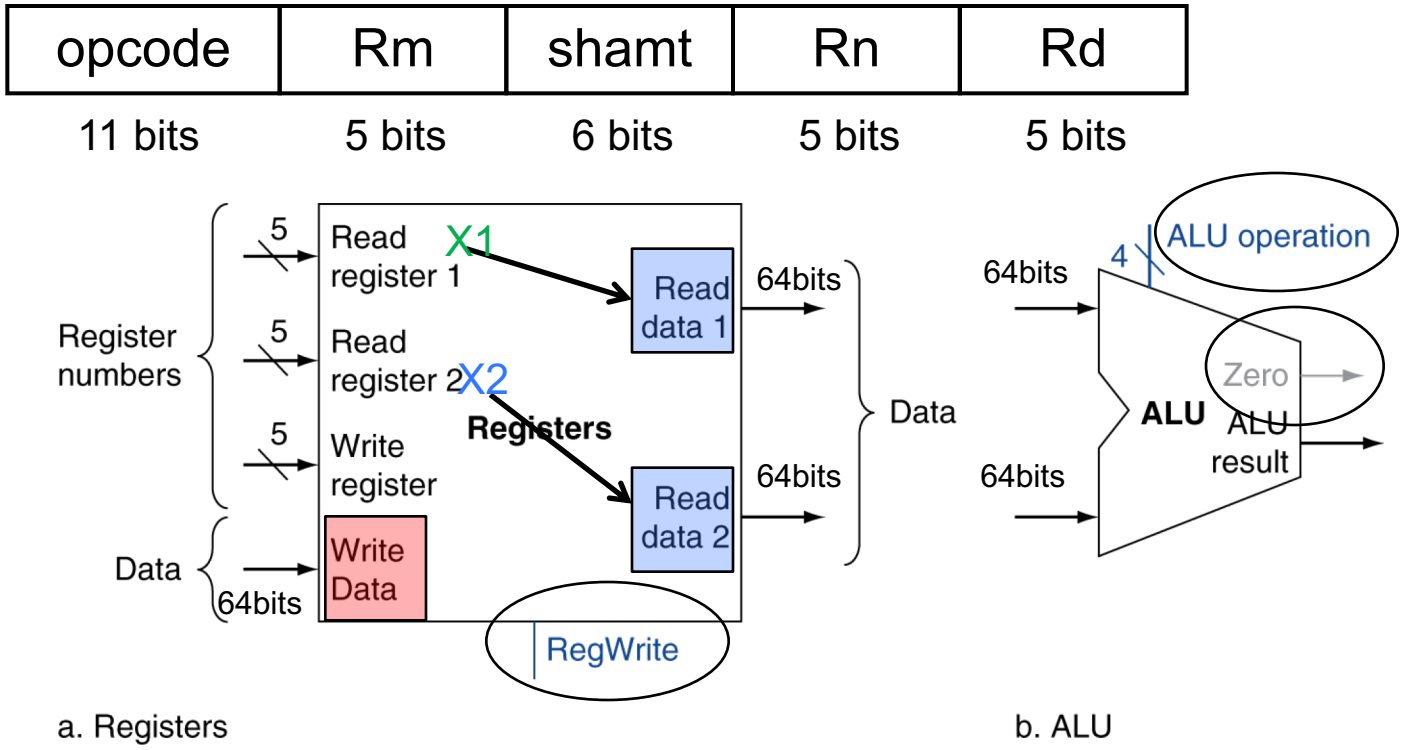
Name	Fields							Comments
Field size		6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format
B-format	B	opcode	address					Unconditional Branch format
CB-format	CB	opcode	address				Rt	Conditional Branch format
IW-format	IW	opcode	immediate				Rd	Wide Immediate format

Instruction Fetch



R-Format Instructions

1. Read two register operands ADD X0, X1, X2
   - assume those registers are loaded with data from memory already
2. Perform arithmetic/logical operation
3. Write register result



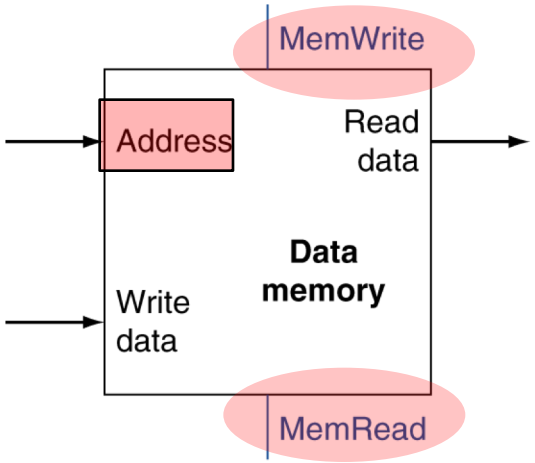
Load/Store Instructions

- Read register operands
- Calculate address using 9-bit offset
  - Use ALU, but first sign-extend offset
- LDUR: Read memory and update register
- STUR: Write register value to memory

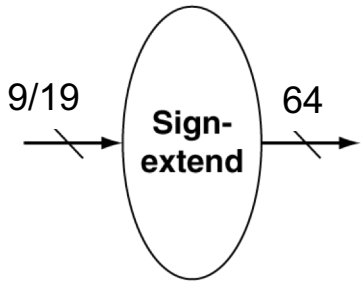
Rt Rn address

LDUR X1, [X2, offset_value]

STUR X1, [X2, offset_value]



a. Data memory unit



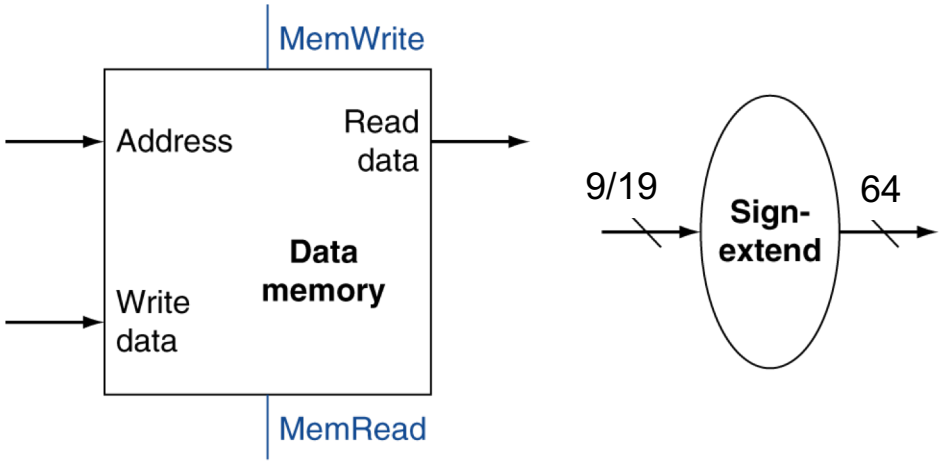
b. Sign extension unit

Load/Store Instructions

- Read register operands
- Calculate address using 9-bit offset
  - Use ALU, but first sign-extend offset
- LDUR: Read memory and update register
- STUR: Write register value to memory

Rt Rn address

LDUR X1, [X2, offset_value]
STUR X1, [X2, offset_value]
CBZ X15, Exit



a. Data memory unit

b. Sign extension unit

Conditional Branch Instruction

- Compare or subtract operands
  - Read register operands.
  - Use ALU: Subtract and check destination register.
- Branch
  - Either,
    - Calculate target address,
    - Or use PC + 4

SUB X0, X1, X2

CBZ X0, offset

Next instruction at PC + 4

.

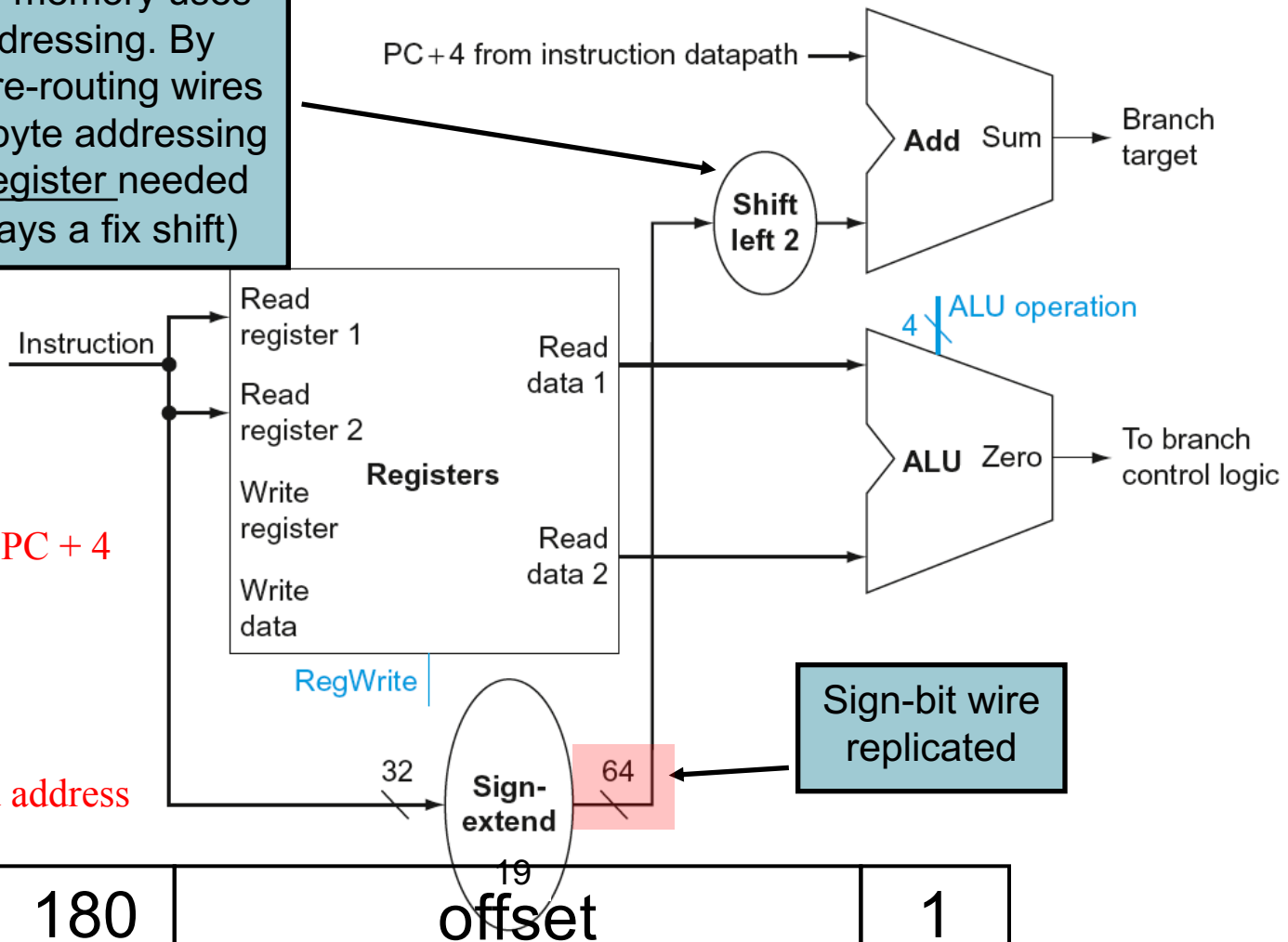
.

.

Branch target address (PC + offset)

Conditional Branch Instruction

Instruction memory uses word addressing. By physically re-routing wires convert to byte addressing (no shift register needed since always a fix shift)



SUB X0, X1, X2

CBZ X0, offset

Next instruction at PC + 4

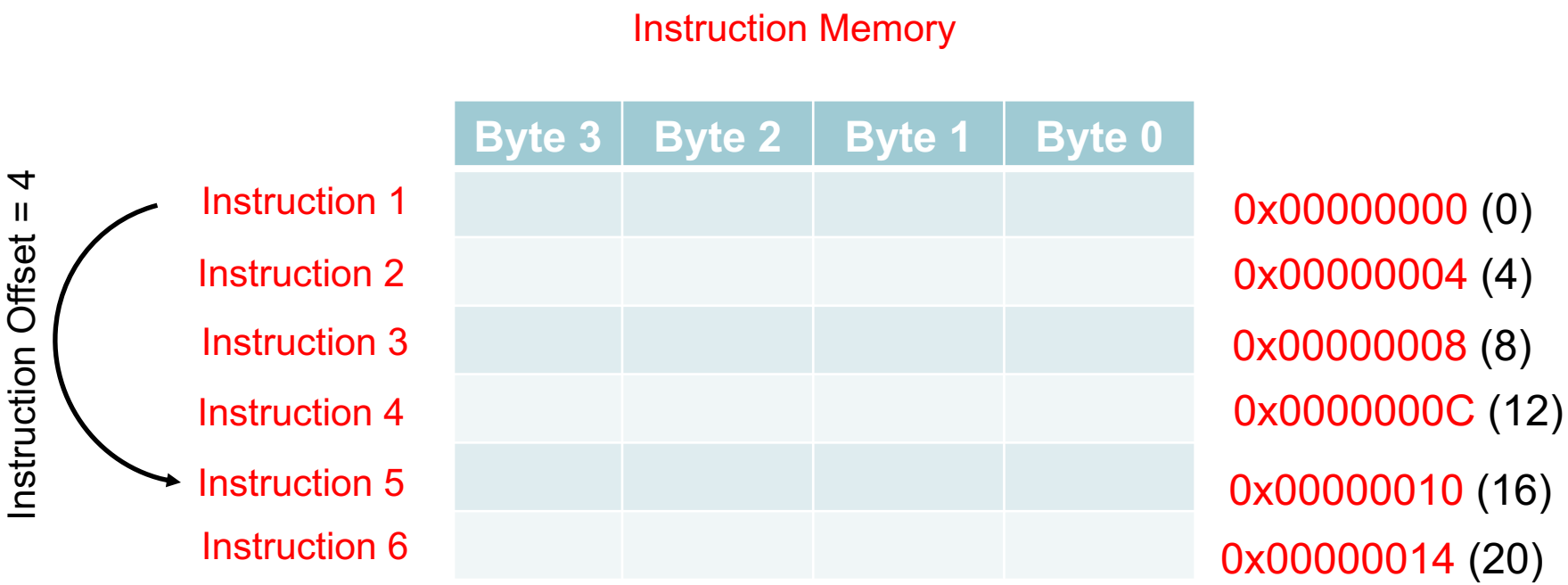
.

.

.

Branch offset word address

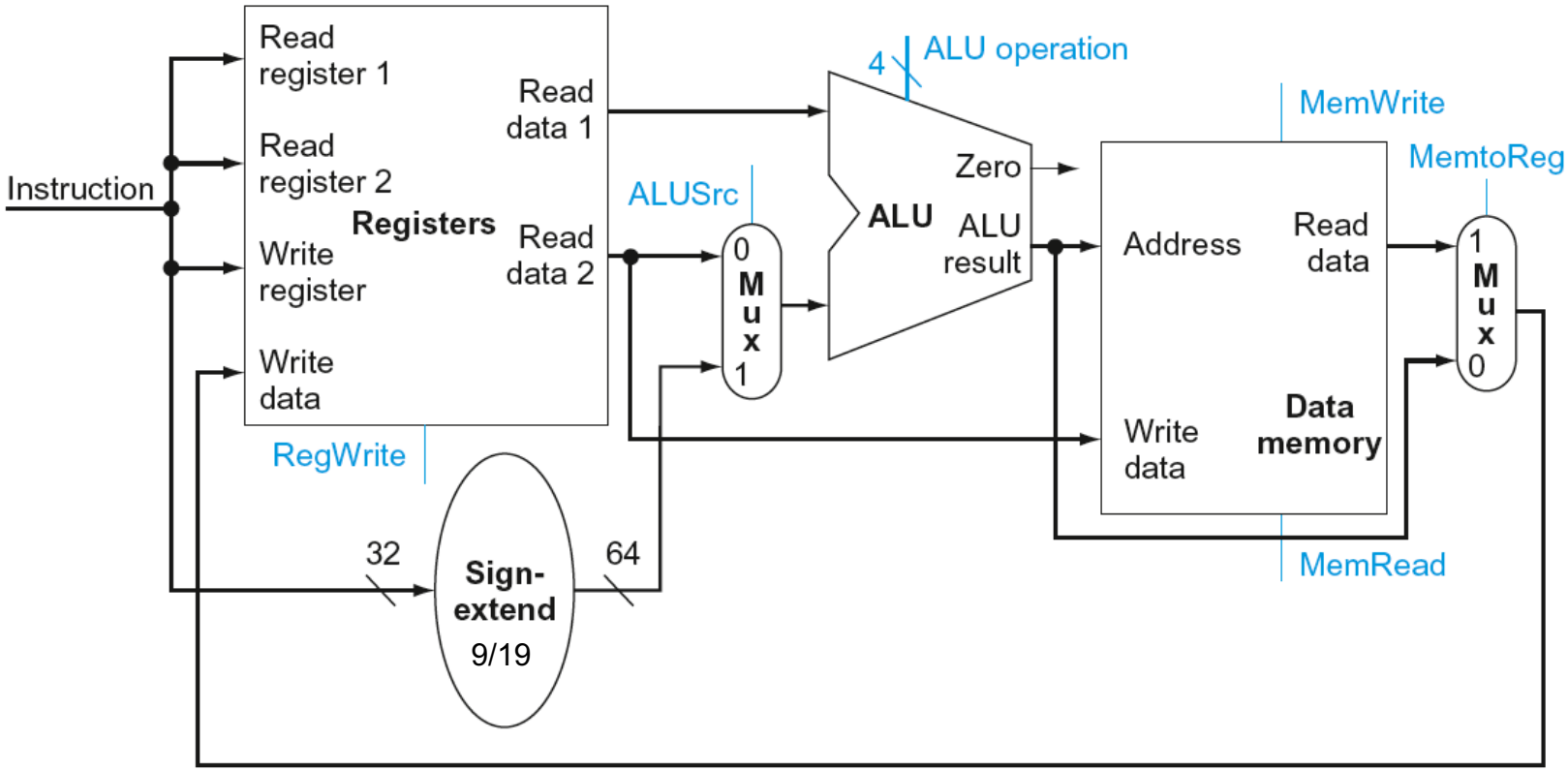
Conditional Branch Instruction



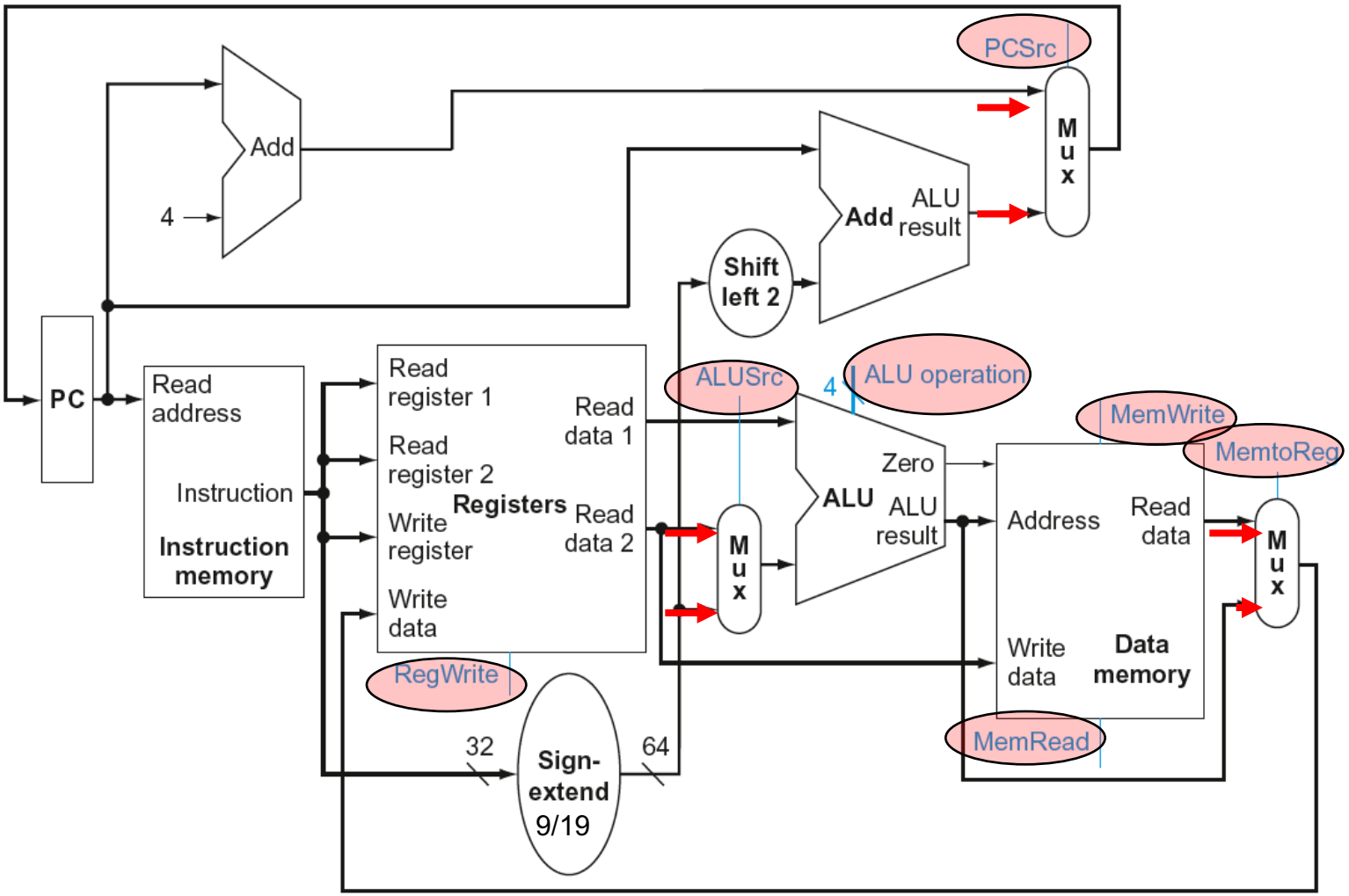
Composing the Elements

- First-cut data path makes an attempt to do an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories (using separate data and instruction caches for example0
- Use multiplexers where alternate data sources are used for different instructions

R-Type/D-Type(ldr/str) Datapath



R-Type/D-Type/CB-Type Datapath



ALU Control

- ALU used for
  - D-type Load/Store: Function = add
  - CB-type CBZ: Function = subtract
  - R-type: Function depends on opcode

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	pass input b
1100	NOR

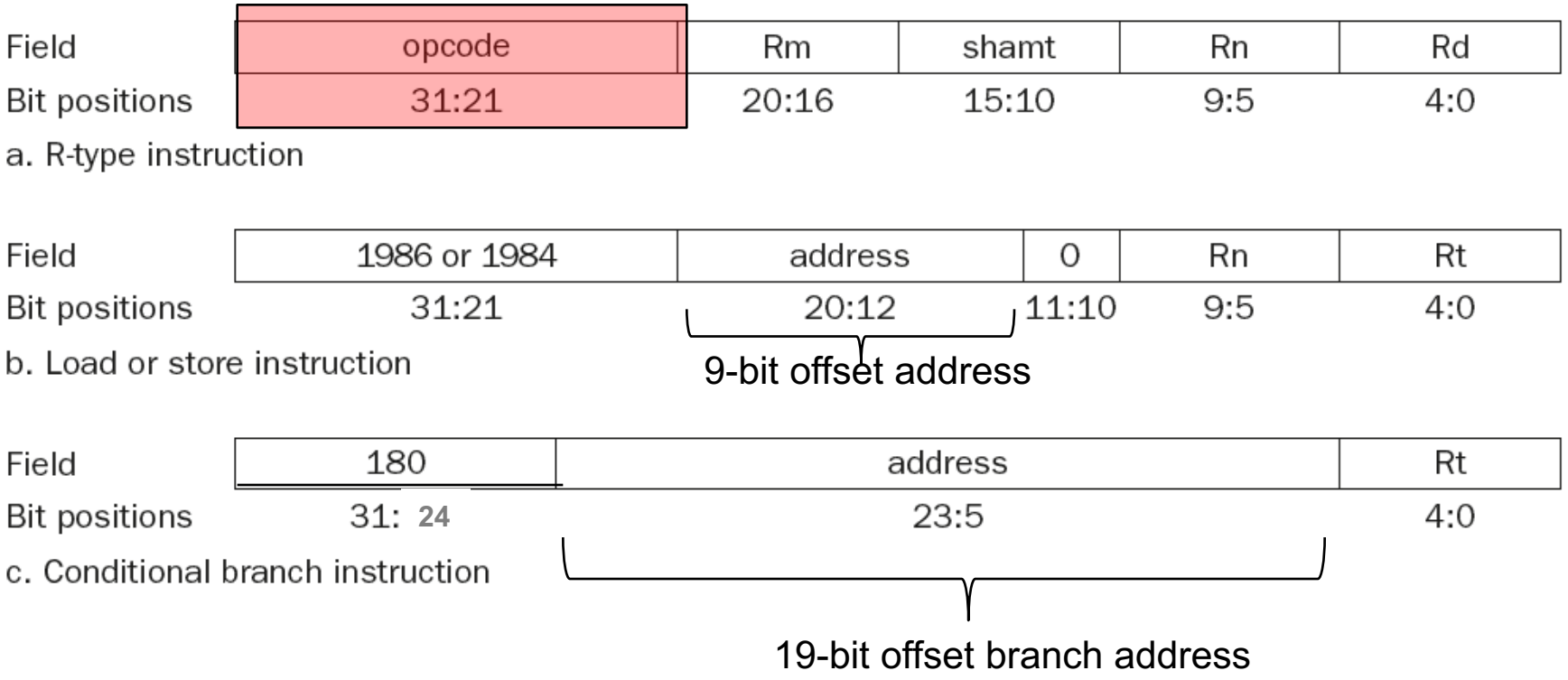
ALU Control

- Assume 2-bit ALUOp control lines derived from opcode and used as an input to the ALU control code generator.

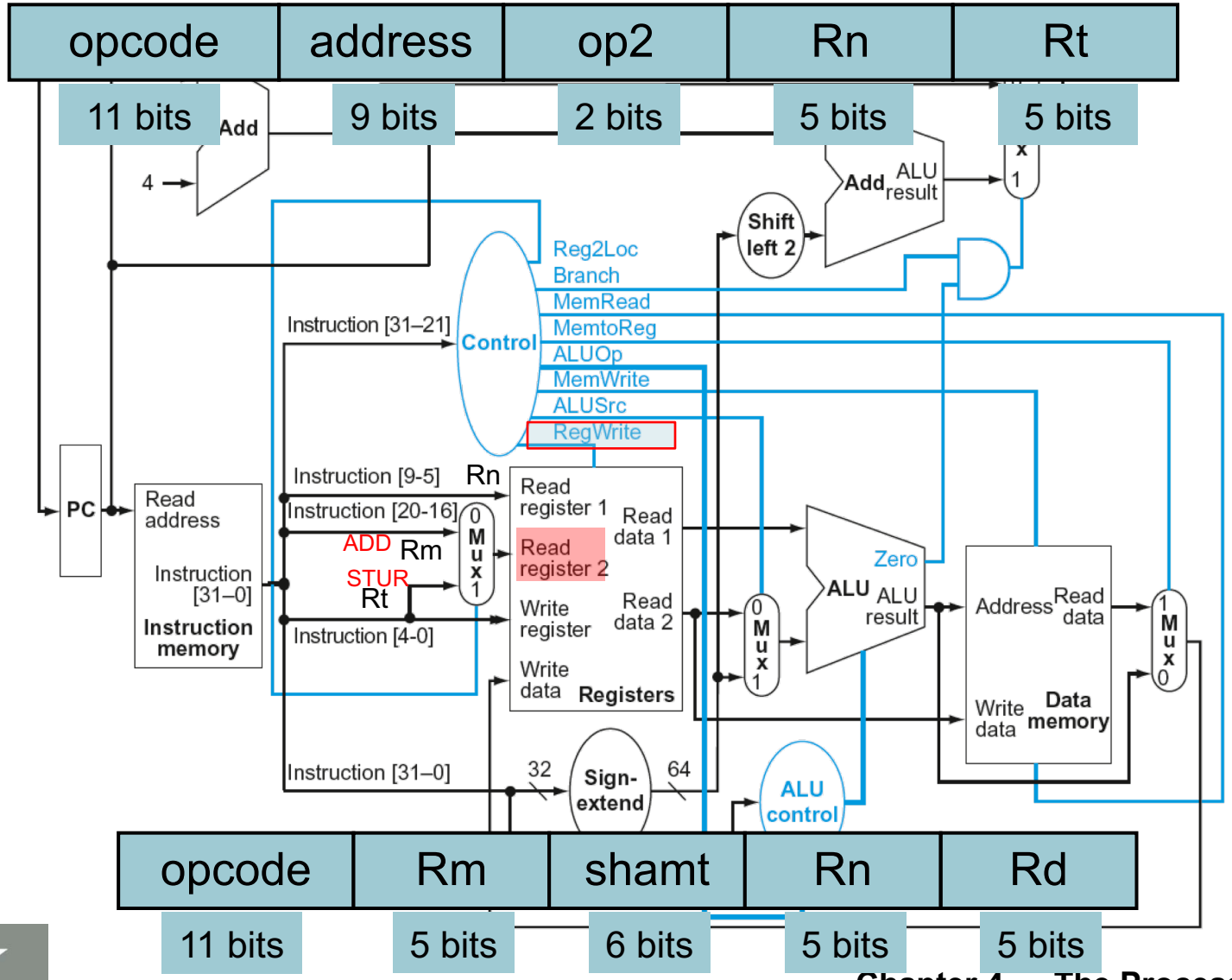
opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111
R-type	10	add	10001011000	add	0010
		subtract	11001011000	subtract	0110
		AND	10001010000	AND	0000
		ORR	10101010000	OR	0001

The Main Control Unit

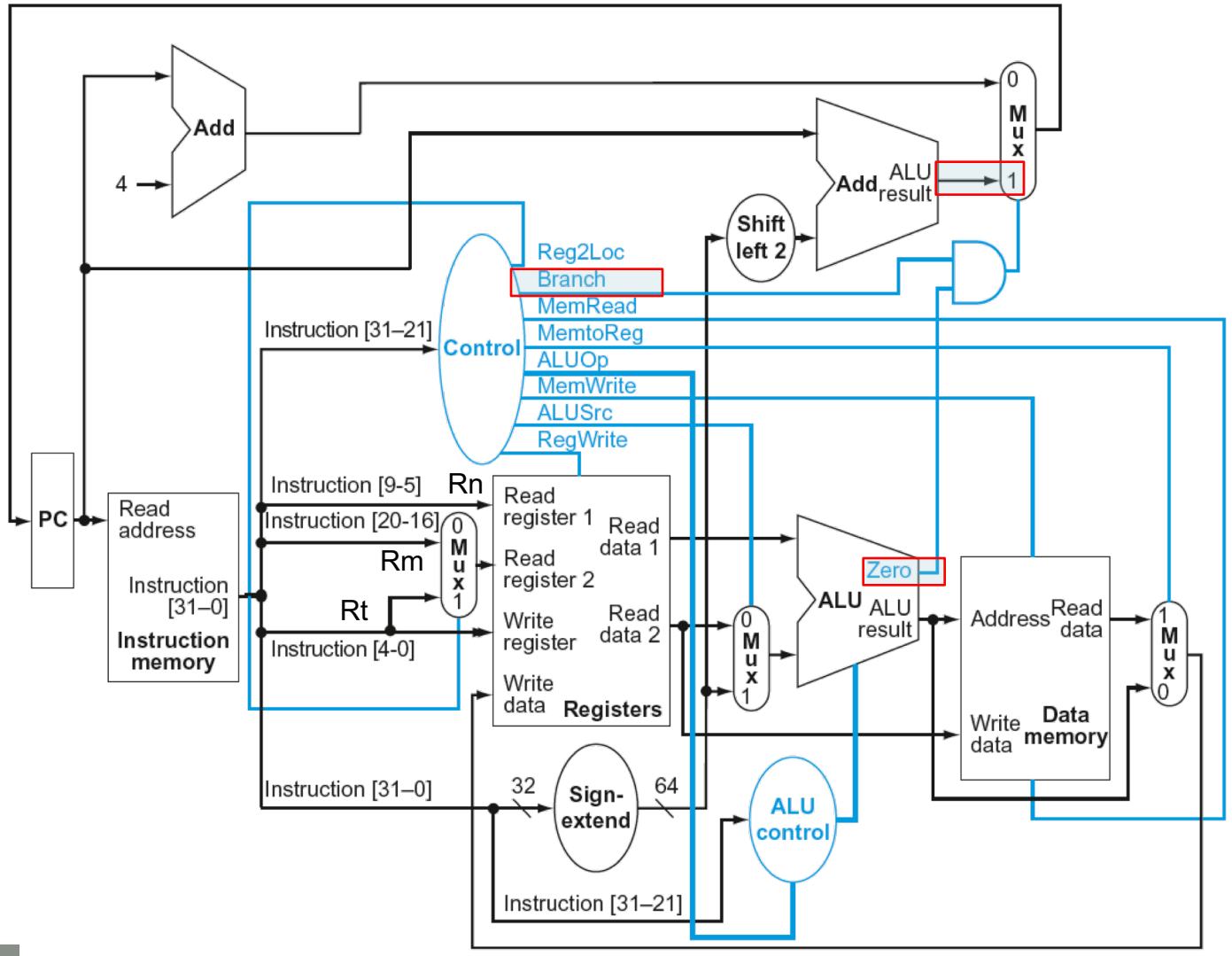
- Control signals derived from instruction



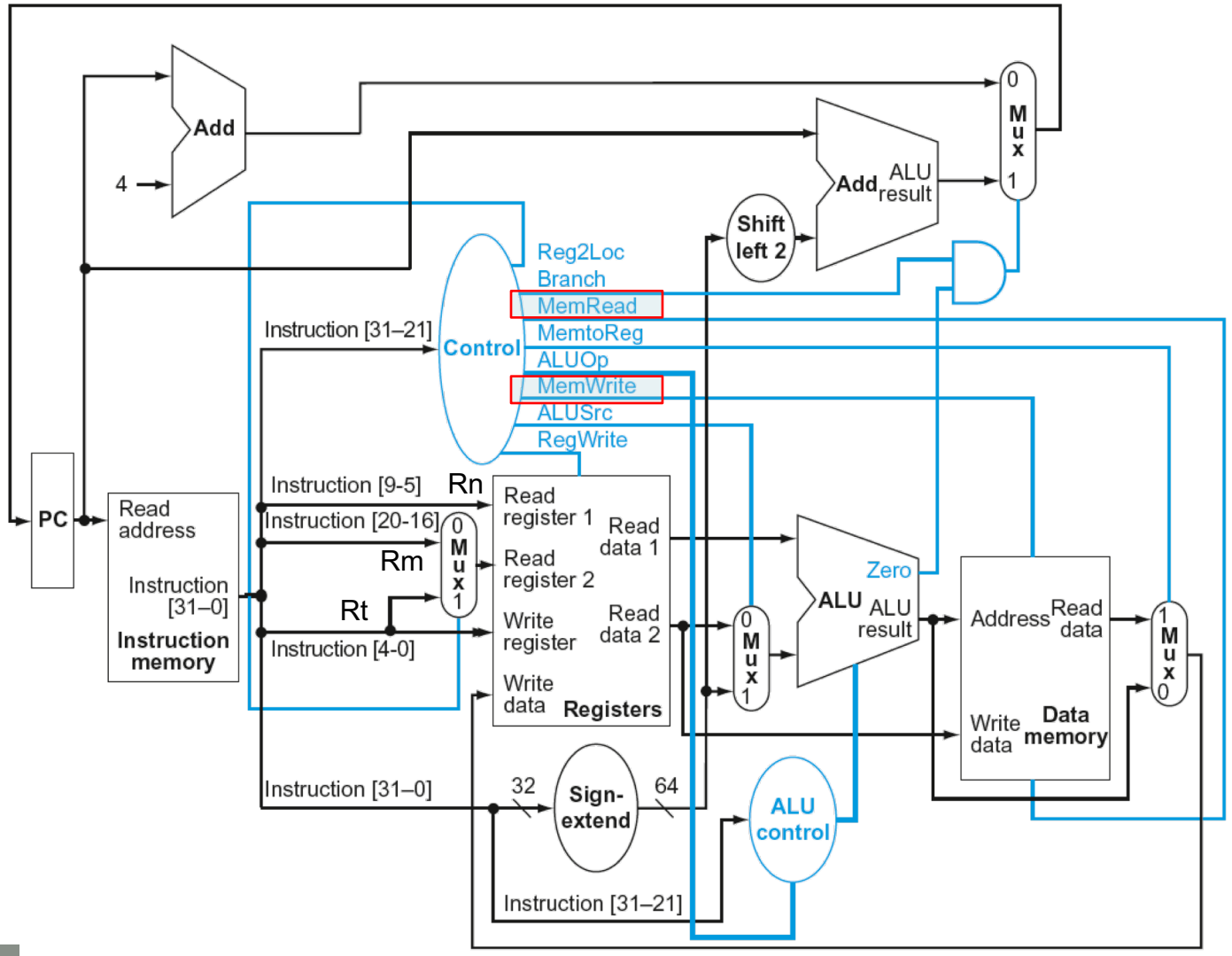
Datapath With Control



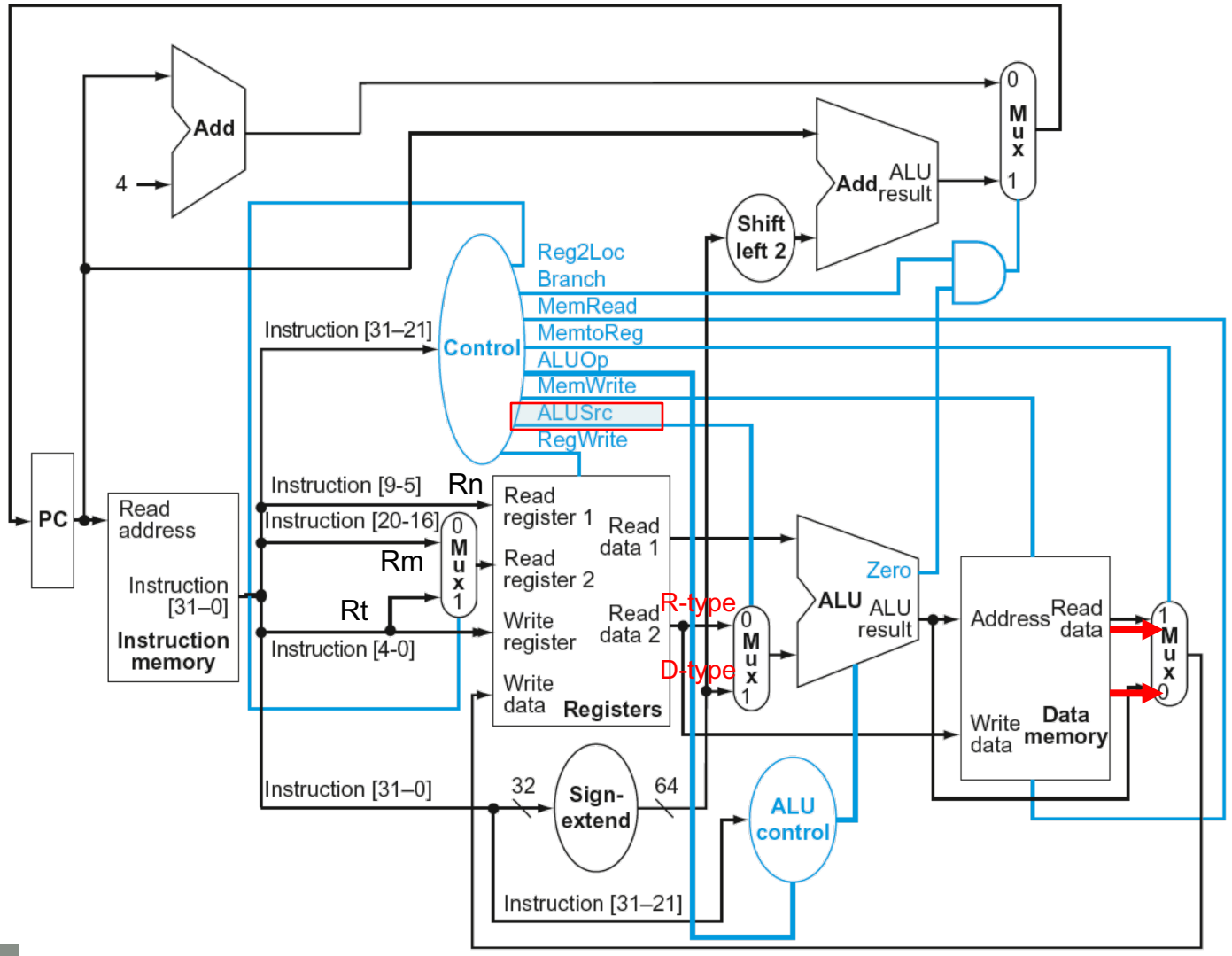
Datapath With Control



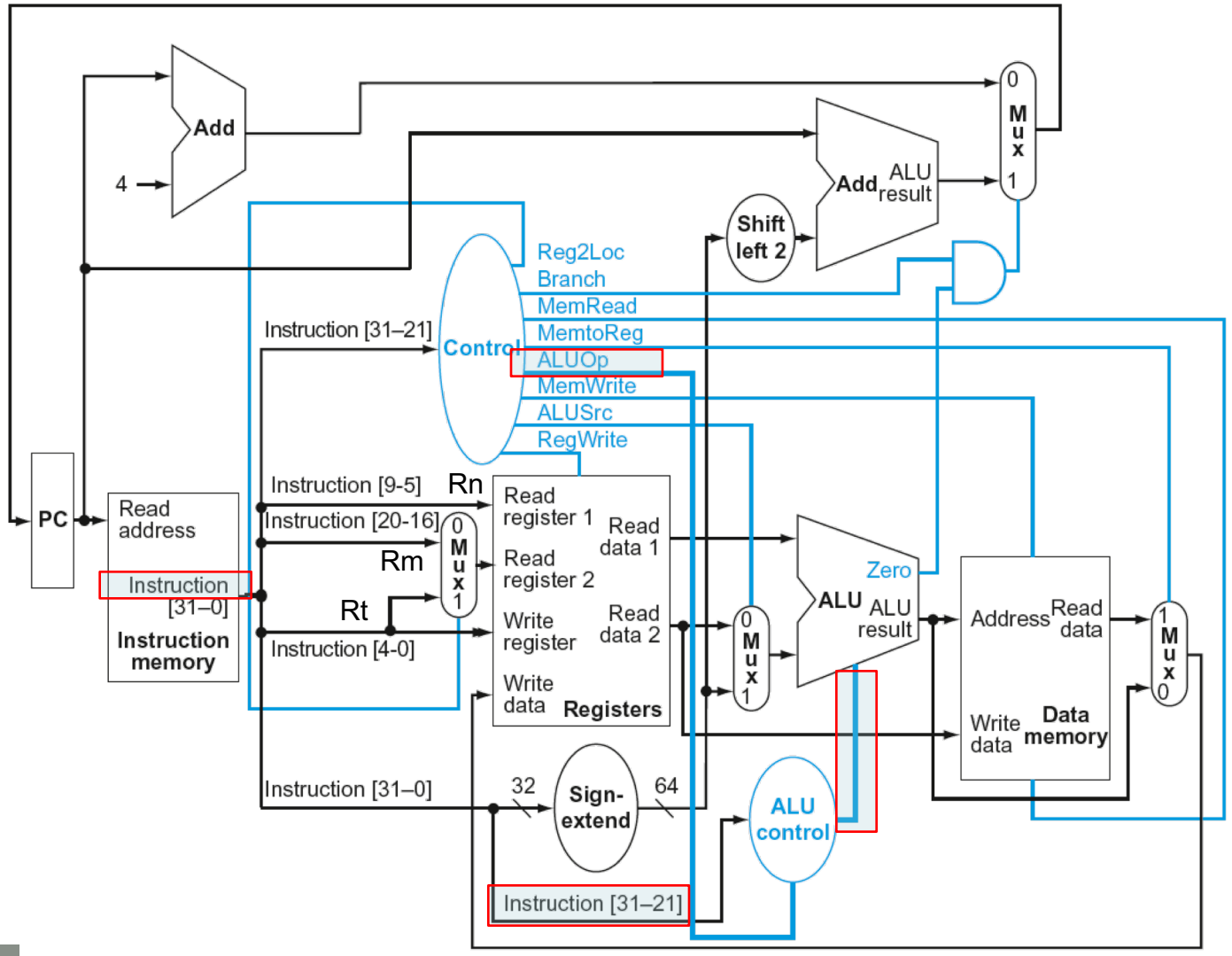
Datapath With Control



Datapath With Control

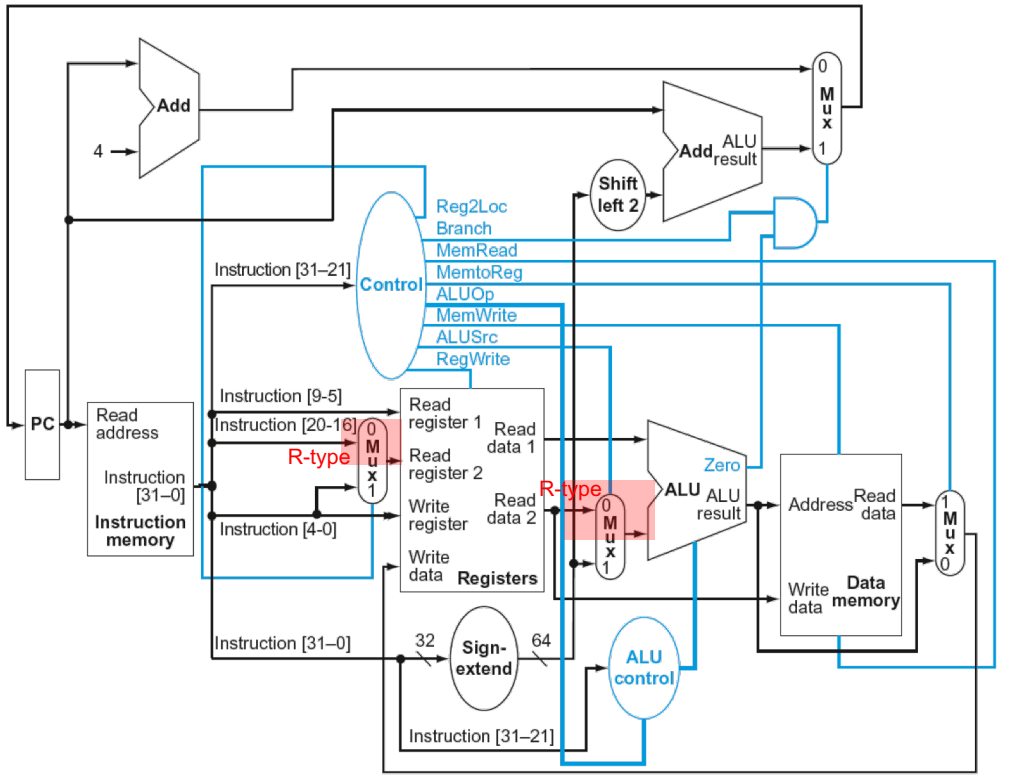


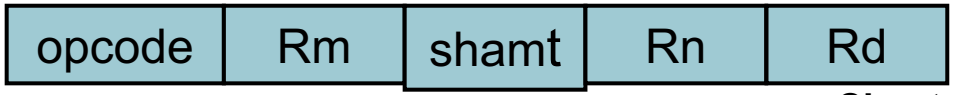
Datapath With Control



Opcode Determines Control Lines

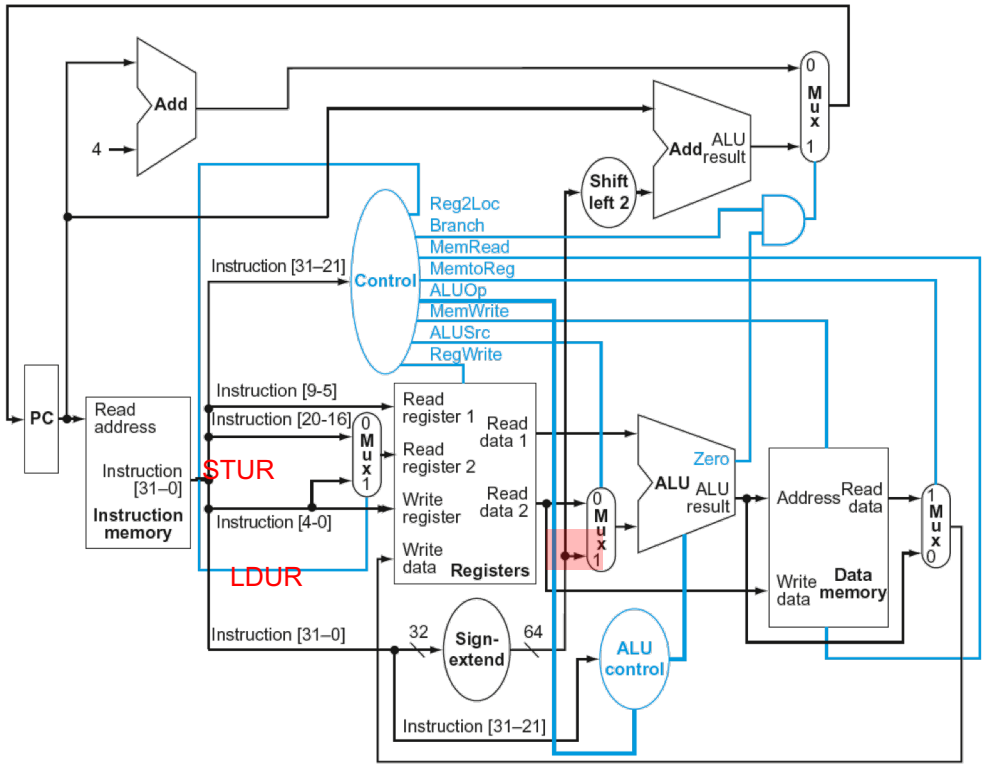
Instruction	Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	0	0	0	1	0	0	0	1	0
LDUR	X	1	1	1	1	0	0	0	0
STUR	1	1	X	0	0	1	0	0	0
CBZ	1	0	X	0	0	0	1	0	1





Opcode Determines Control Lines

Instruction	Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	0	0	0	1	0	0	0	1	0
LDUR	X	1	1	1	1	0	0	0	0
STUR	1	1	X	0	0	1	0	0	0
CBZ	1	0	X	0	0	0	1	0	1

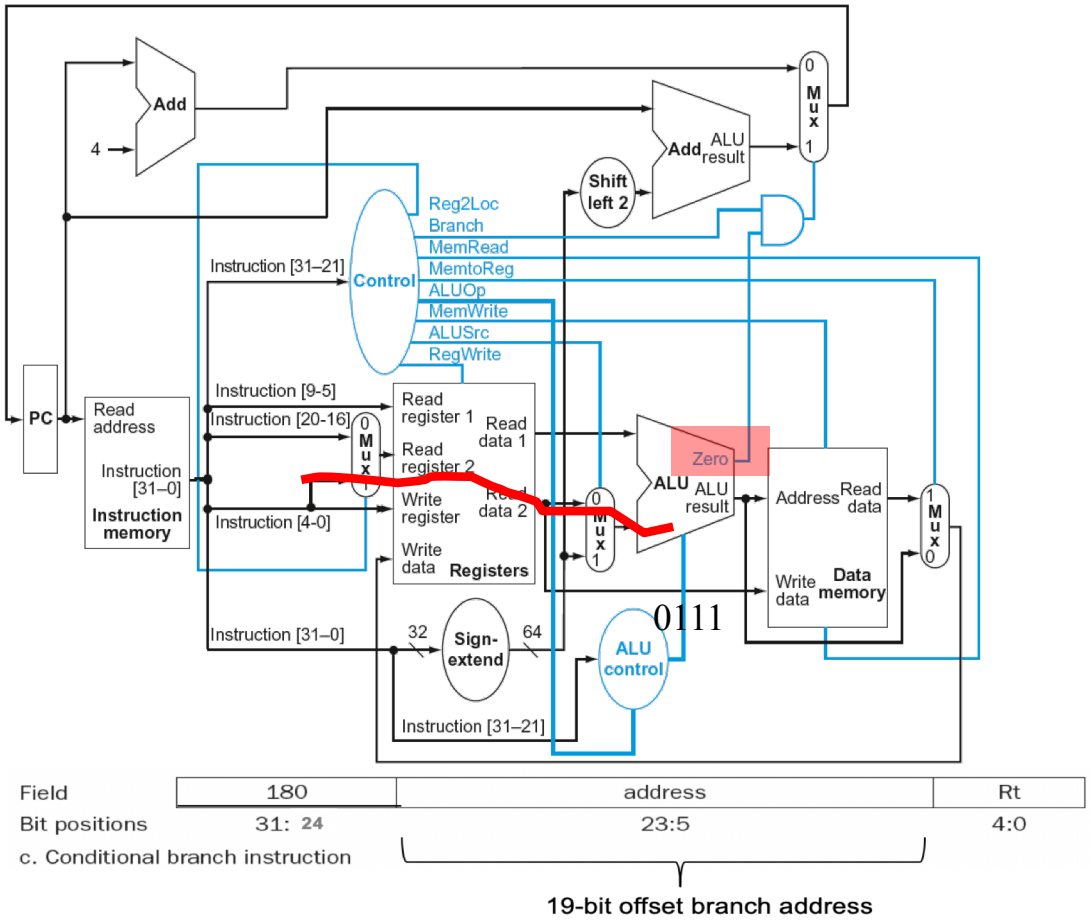



```
LDUR   X9,[X22,#64]
STUR   X9,[X22,#64]
```

opcode	address	op2	Rn	Rt

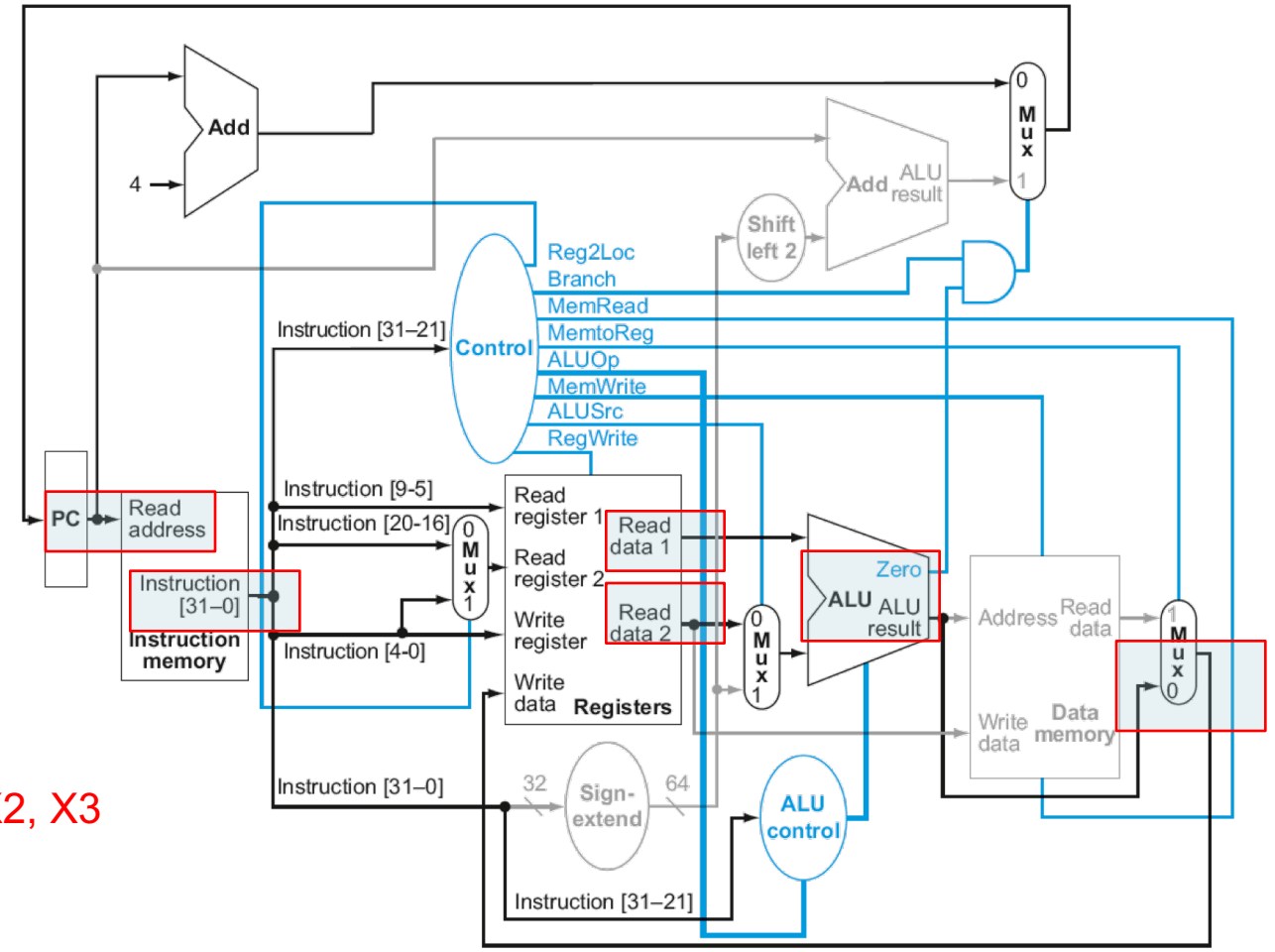
Opcode Determines Control Lines

Instruction	Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	0	0	0	1	0	0	0	1	0
LDUR	X	1	1	1	1	0	0	0	0
STUR	1	1	X	0	0	1	0	0	0
CBZ	1	0	X	0	0	0	1	0	1



R-Type Instruction

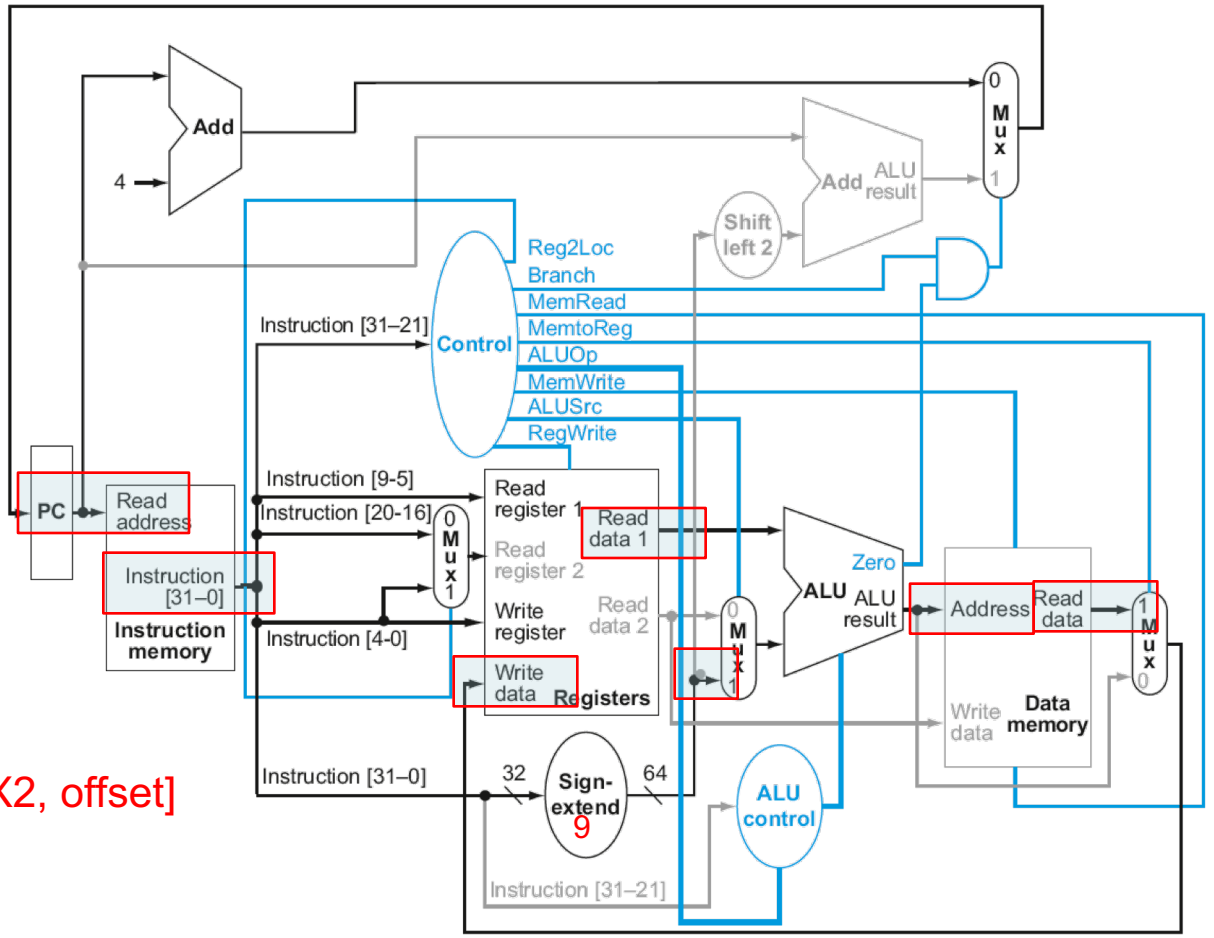
opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits



ADD X1, X2, X3

D-Type (Load Instruction)

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

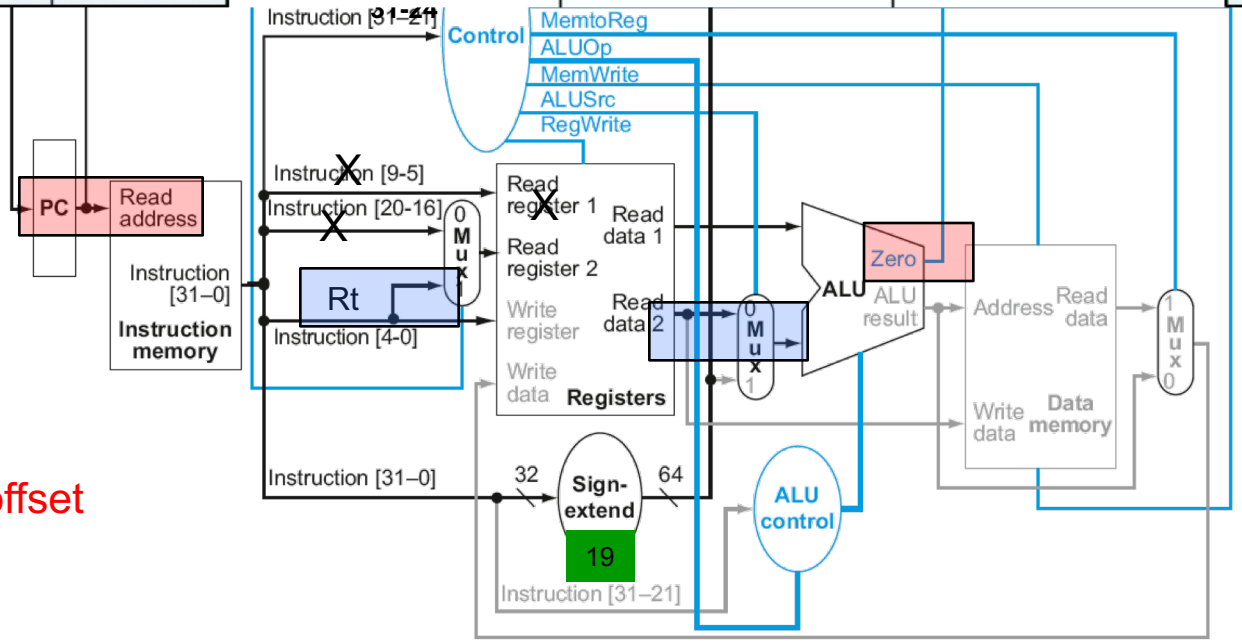


LDUR X1, [X2, offset]

CBZ Instruction

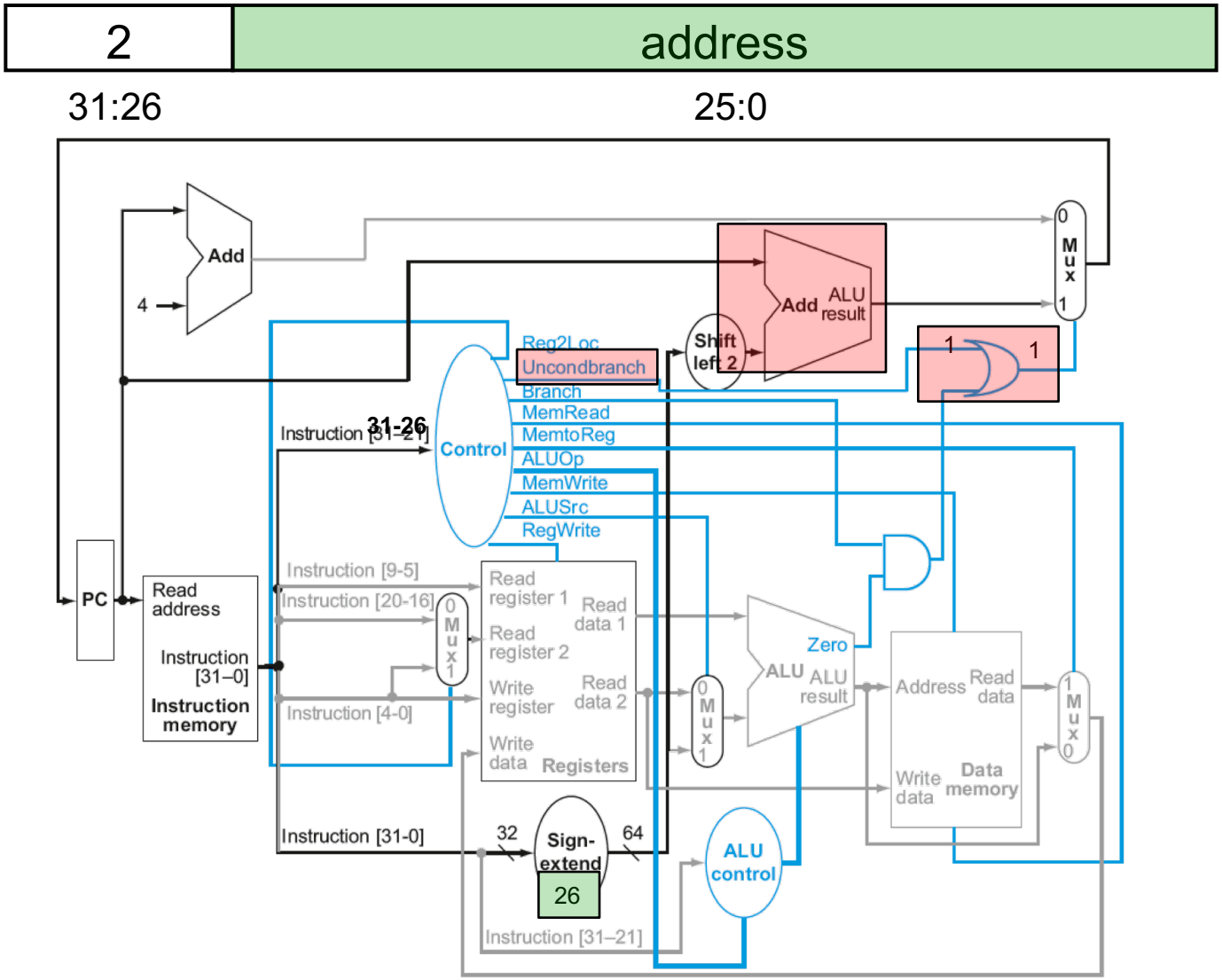
180	Exit	11
8 bits	19 bits	5 bits

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111

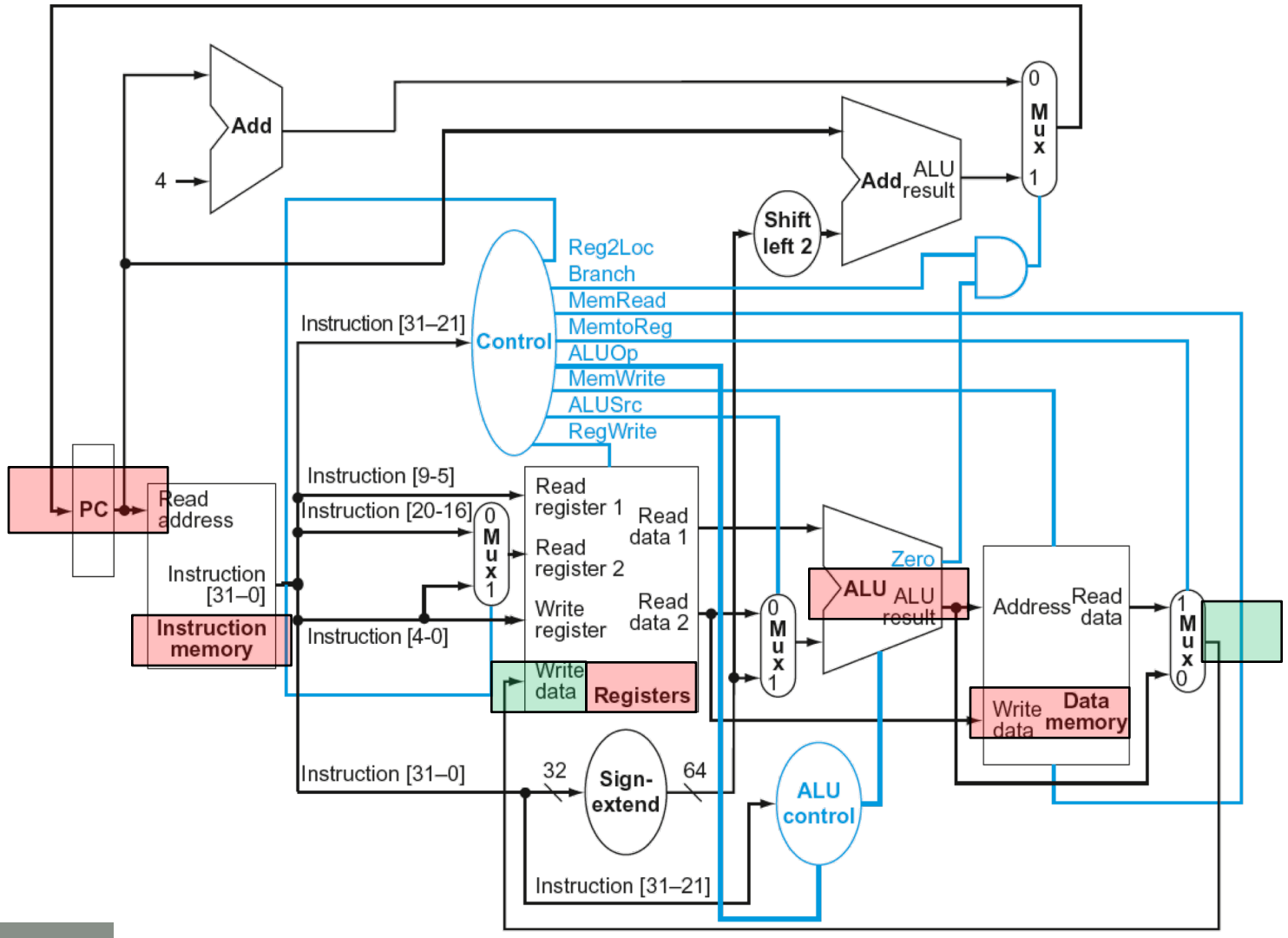


CBZ X11, offset

Datapath With B-type Added

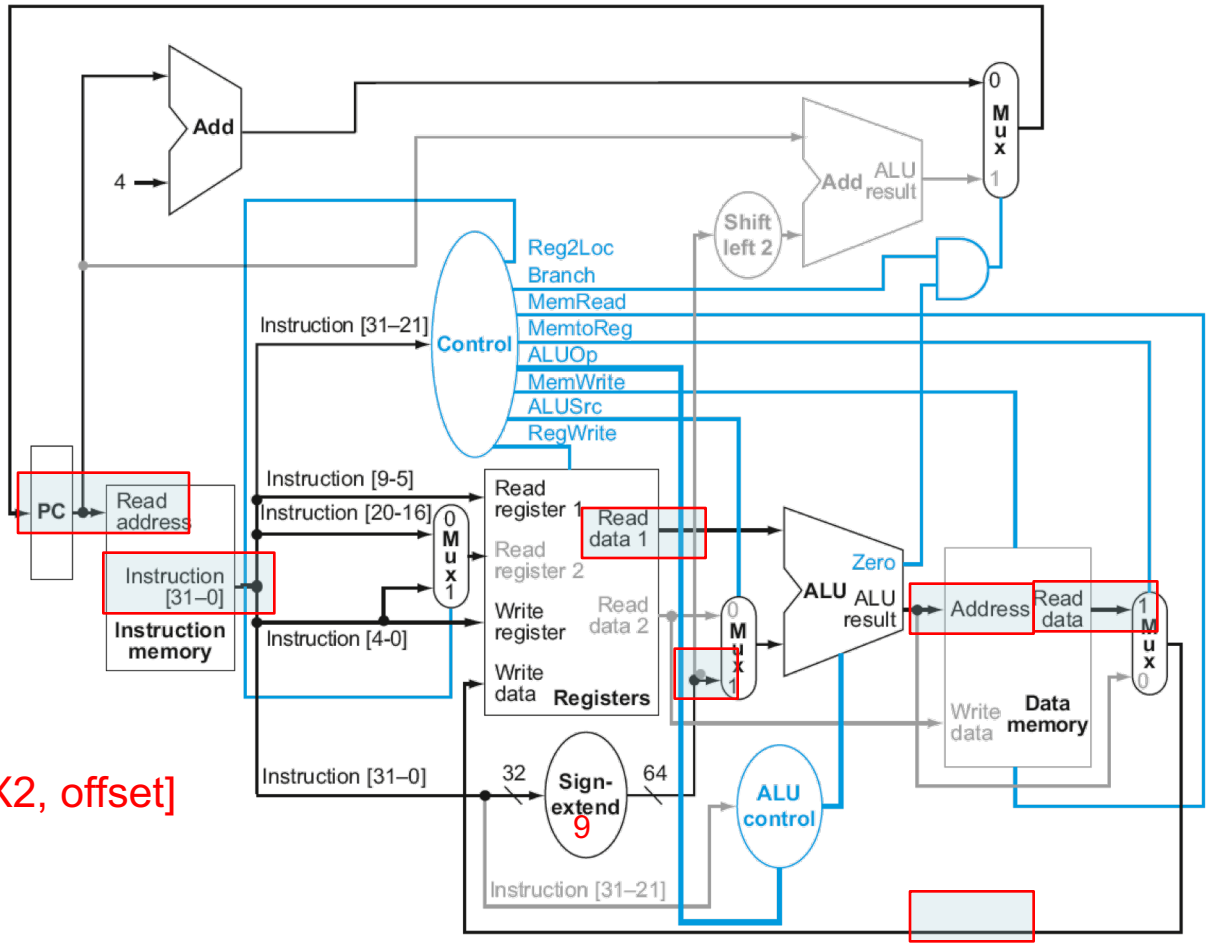


Performance Issues



D-Type (Load Instruction)

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits



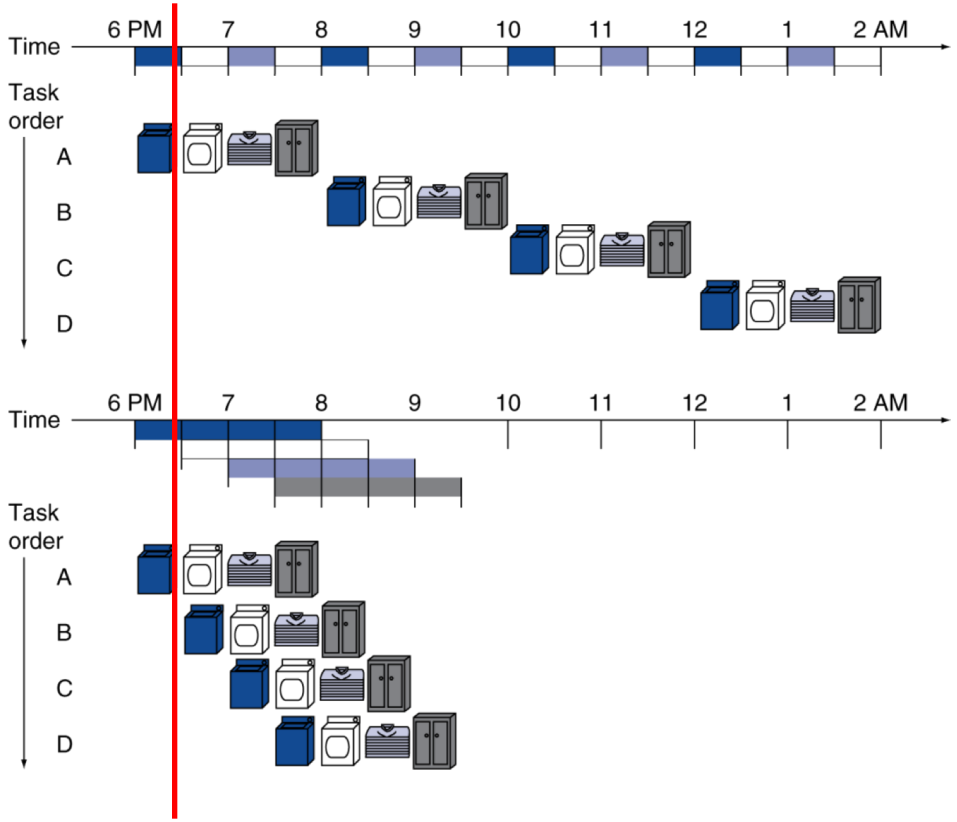
LDUR X1, [X2, offset]

Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction fetch → read register file → ALU → read data from memory → write to register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



LEGv8 Pipeline

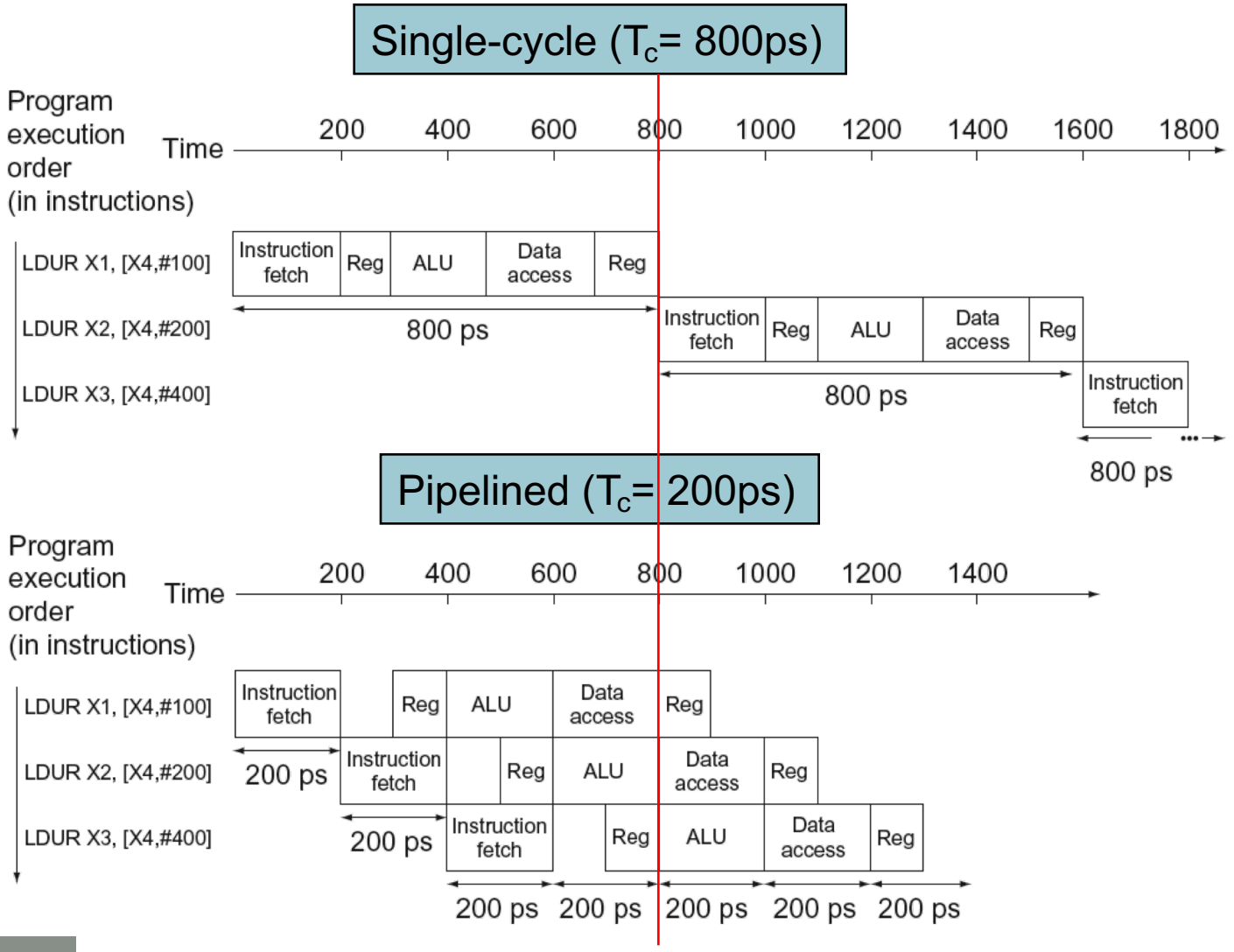
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipeline Performance

- For these typical LEGv8 instructions:
  - Let's assume time taken for each stage is as follows:
    - 100ps for a register read or register write
    - 200ps for other stages

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
LDUR	200ps	100 ps	200ps	200ps	100 ps	800ps
STUR	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
CBZ	200ps	100 ps	200ps			500ps

Pipeline Performance



Pipeline Performance

Single-cycle (T_c= 800ps)

So in the single-cycle we need to stretch the clock period to 800ps to accommodate the LDUR instruction:

Clock Rate = (1 / Clock Period) = (1 / 800 ps) = [1 / (800 x 10^{-12})] = 1.25 GHz (or 1.25 billions cycles per second.

Pipelined (T_c= 200ps)

Versus in the pipelined design our clock period can be about 200 ps to accommodate the same load instruction:

Clock Rate = (1 / Clock Period) = (1 / 200 ps) = [1 / (200 x 10^{-12})] = 5 GHz (or 5 billions cycles per second.

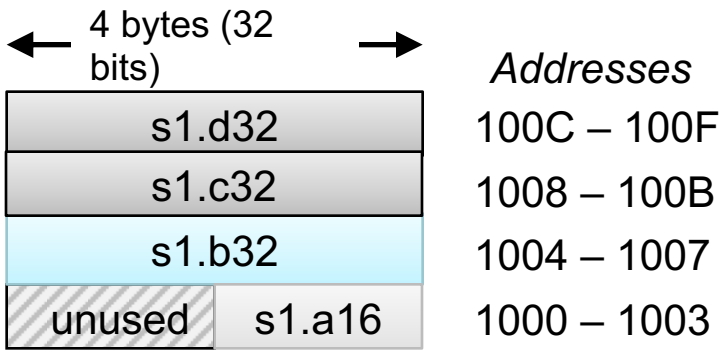
Pipelining and ISA Design

- LEGv8 ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
  - Few and regular instruction formats
    - Can decode and read registers in one step

Name	Fields							Comments
Field size		6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format
B-format	B	opcode	address					Unconditional Branch format
CB-format	CB	opcode	address				Rt	Conditional Branch format
IW-format	IW	opcode	immediate				Rd	Wide Immediate format

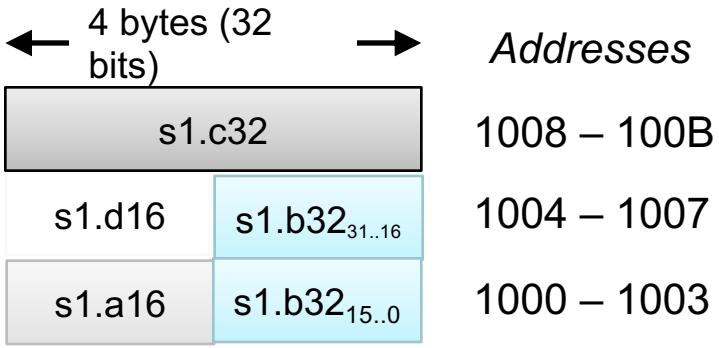
Pipelining and ISA Design

- Alignment of memory operands in LEGv8
  - Memory access takes only one cycle



Optimized for speed (default)

The entire s1.b32 instruction is at 1004 address in memory and therefore, can be accessed in one clock cycle.



Optimized to conserve memory

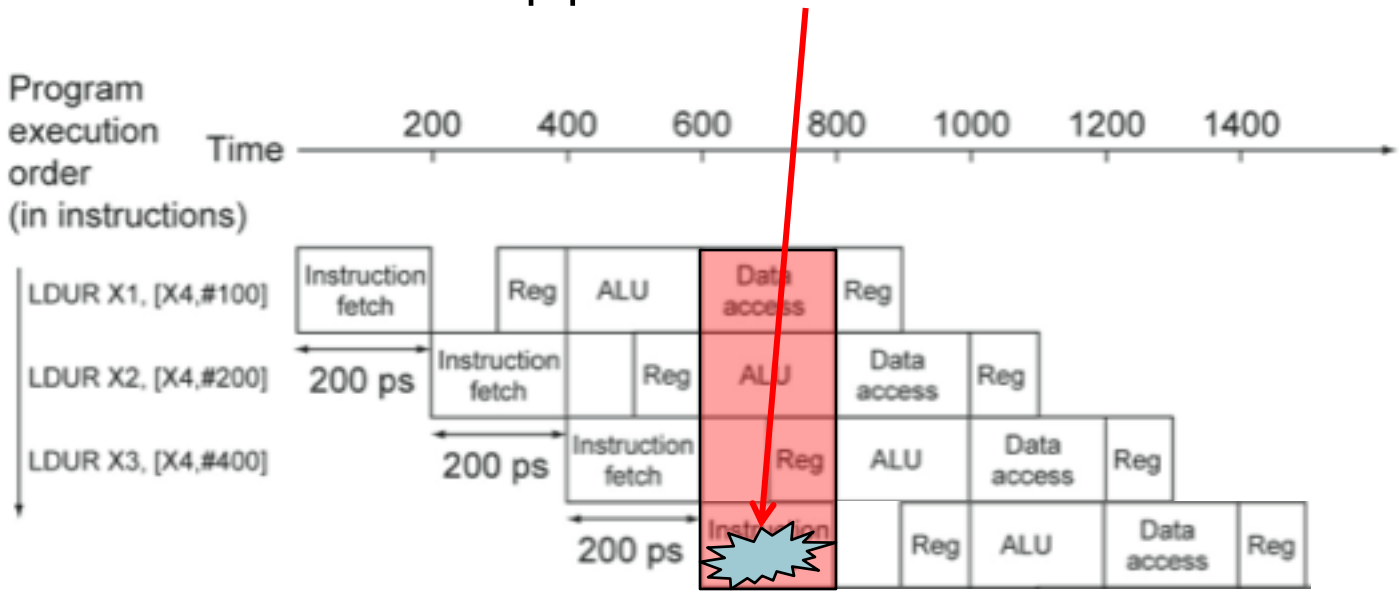
First half of the s1.b32 is at 1000 and the second half is at 1004 memory location so requires two memory accesses.

Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure 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

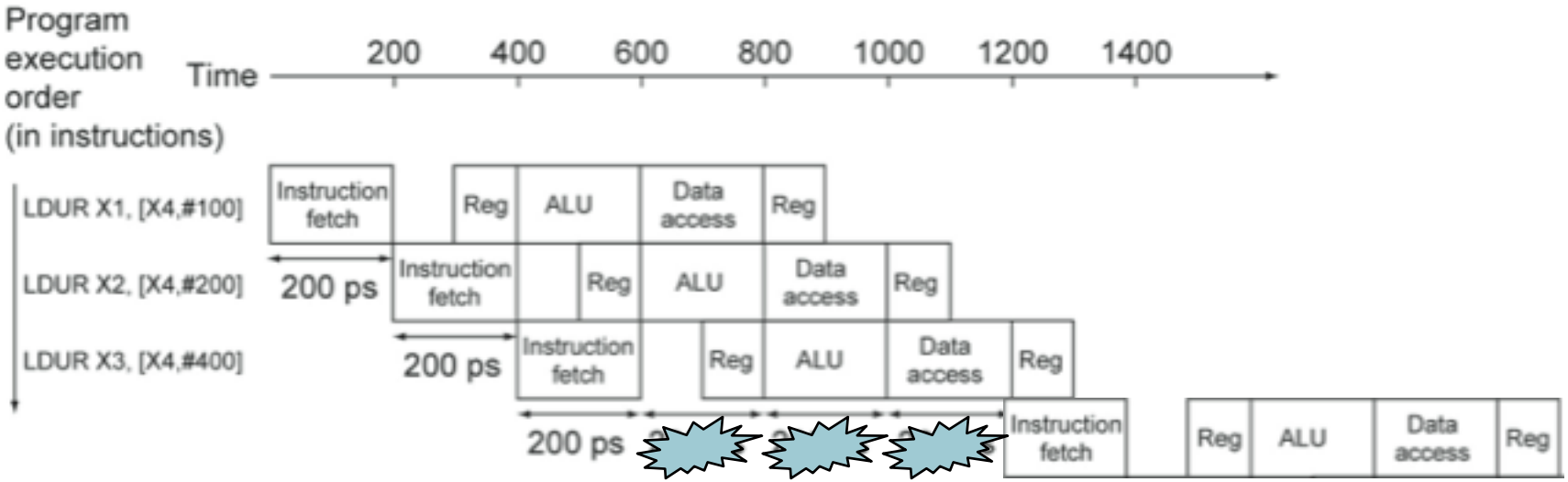
Structure Hazards

- Conflict for use of a resource
- In LEGv8 pipeline with a single memory
  - Load requires instruction/data memory access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”



Structure Hazards

- Conflict for use of a resource
- In LEGv8 pipeline with a single memory
  - Load requires instruction/data memory access
  - Instruction fetch would have to *stall* for 600 ps
    - Would cause three pipeline “bubbles”

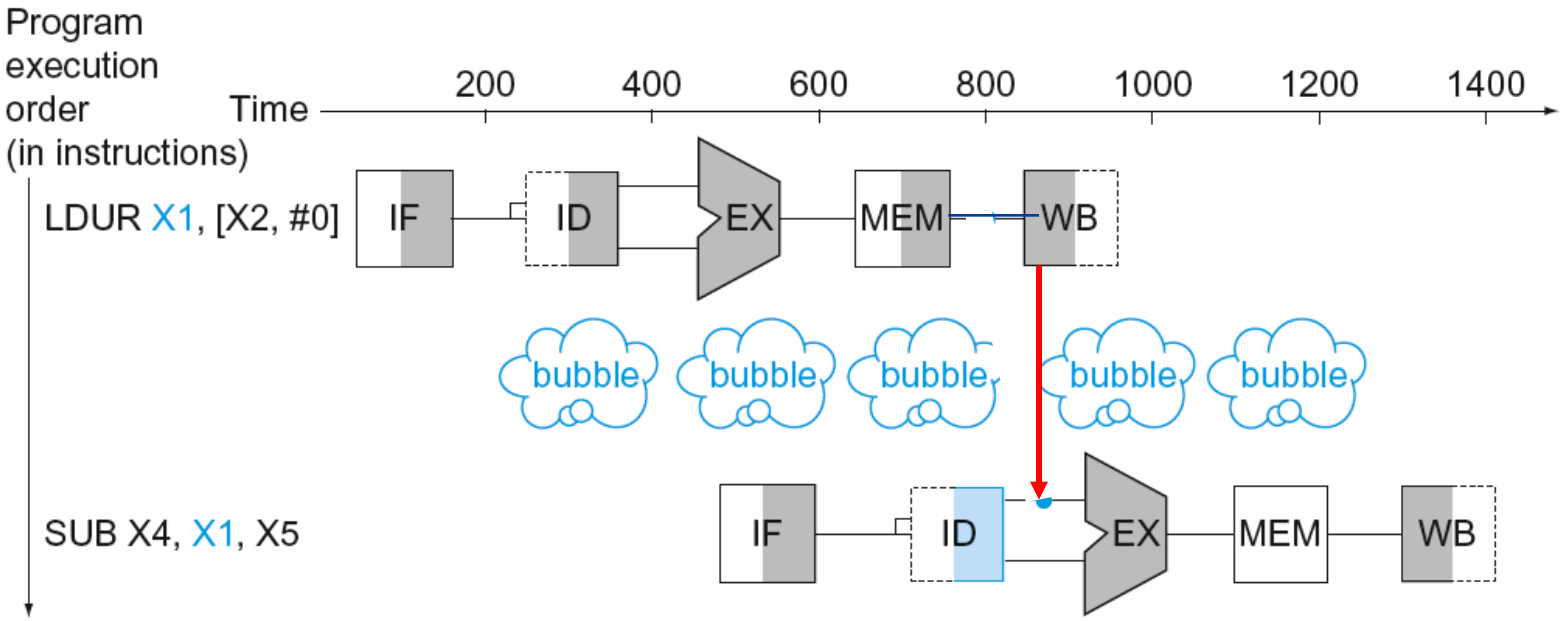


Structure Hazards

- Conflict for use of a resource
- In LEGv8 pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

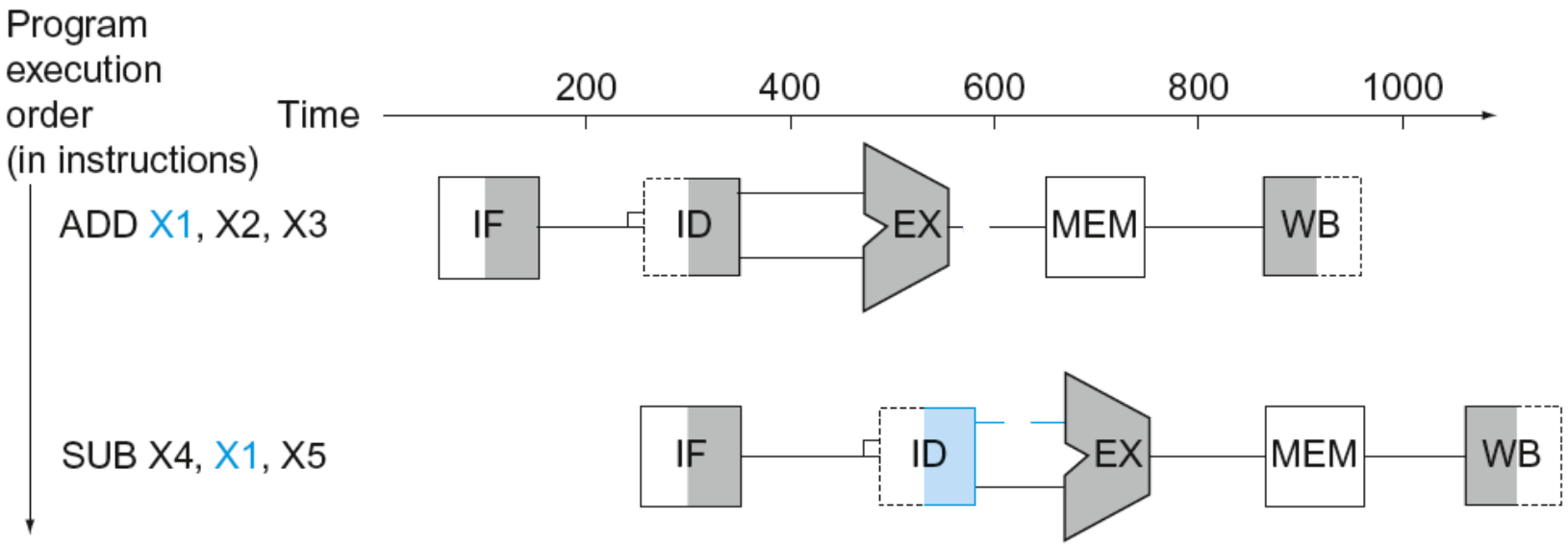
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - `LDUR X1, [X2, #0]`
  - `SUB  X4, X1, X5`



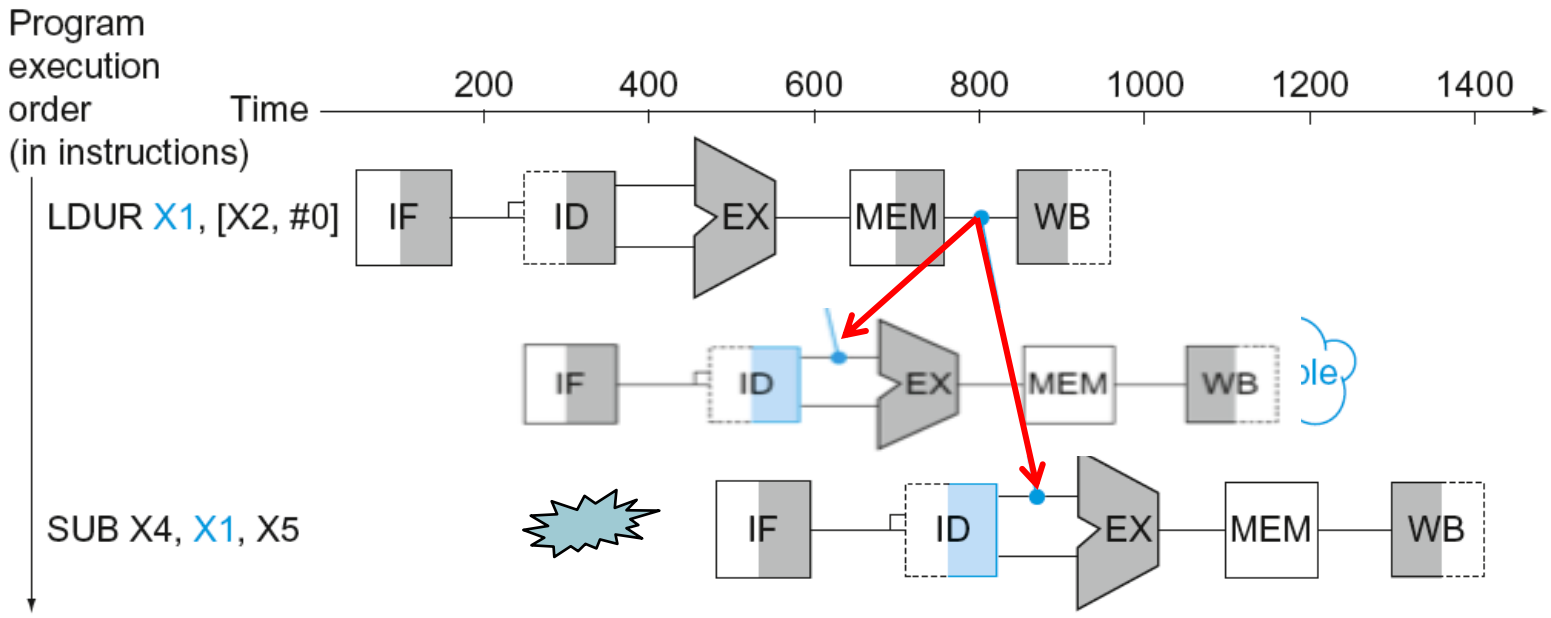
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath



Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; D = B + F;`