CSCI 210: Computer Architecture
Lecture 26: Control Path

Stephen Checkoway

Slides from Cynthia Taylor

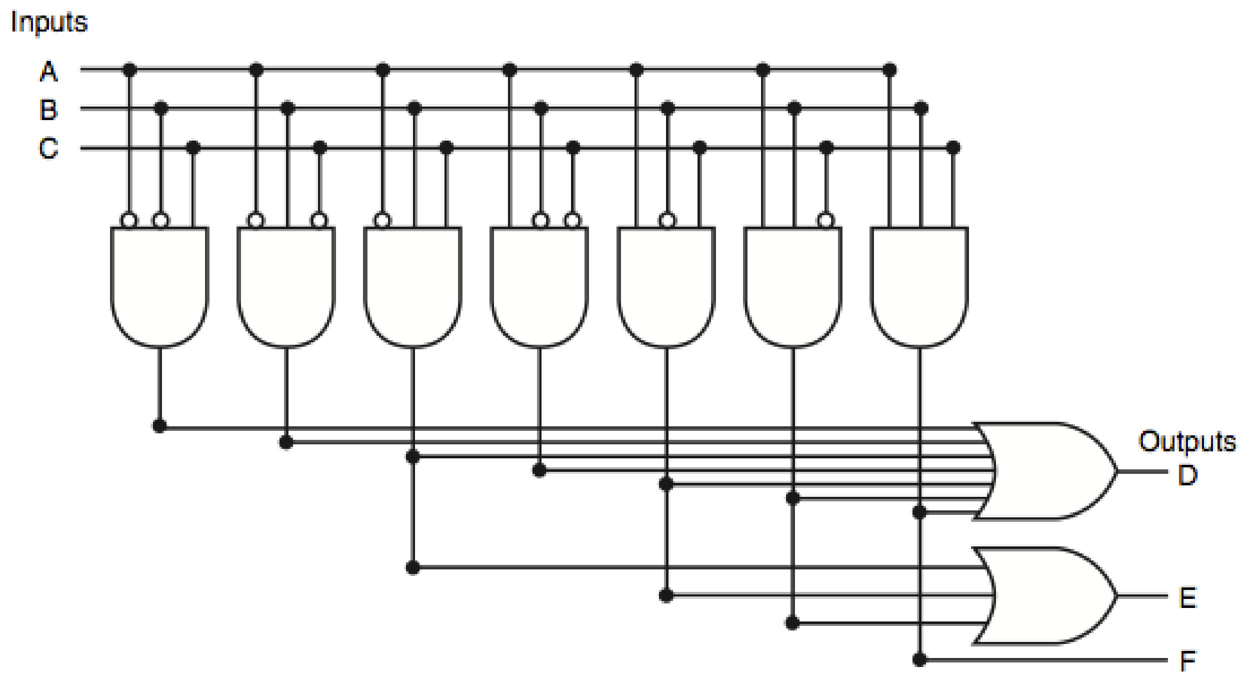
CS History: Apple Lisa

- First mass-market PC that used a graphical user interface
- Released in 1983
- Cost $9,995 (equivalent to $29,400 in 2022)
- Used the Motorola 68000 CPU, the first 32-bit CPU
- Shipped with 1 MB of RAM

Control Path

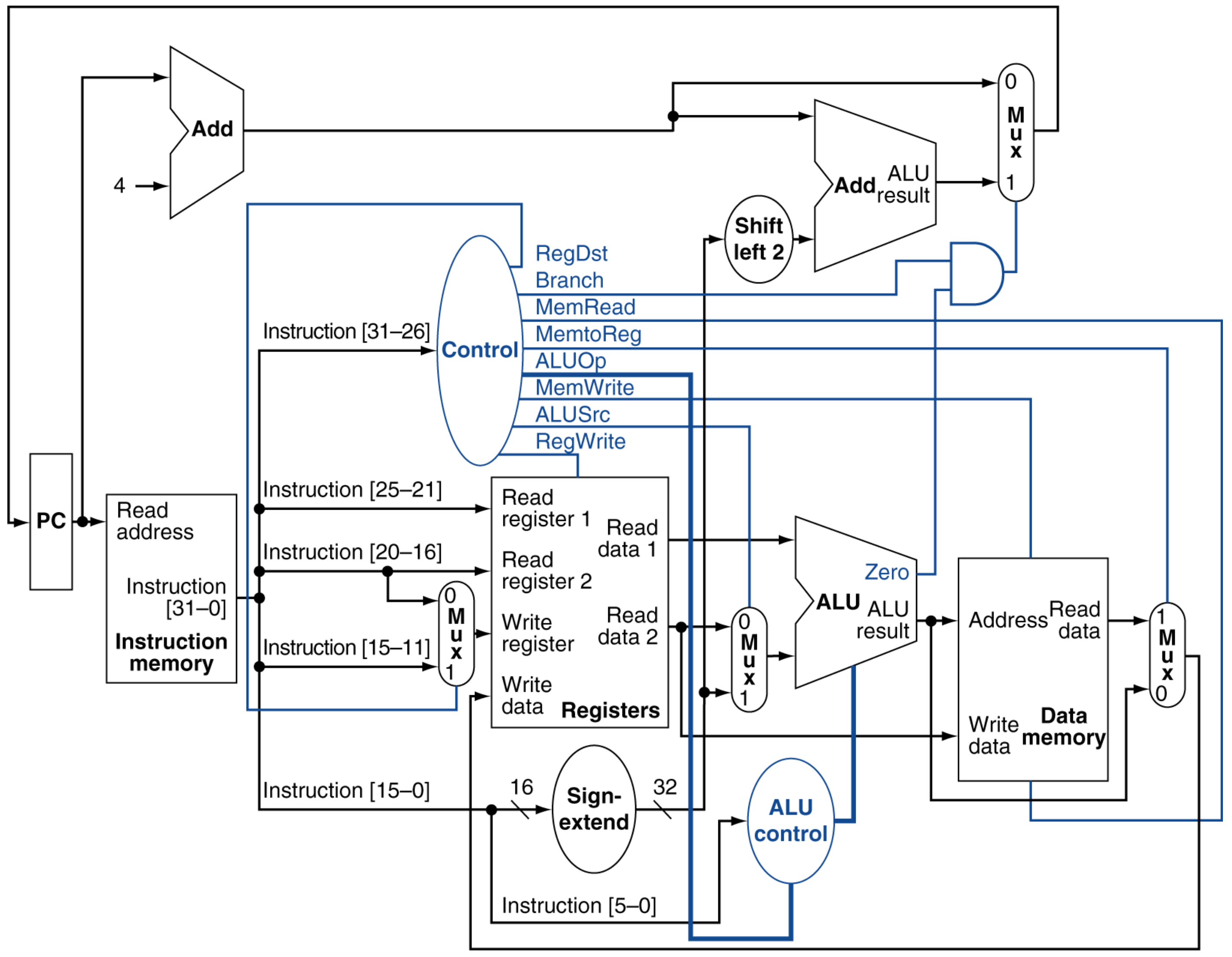
- Our data path is complicated, and we don’t use each element every time
- How do we know which elements to use?

Recall: PLAs



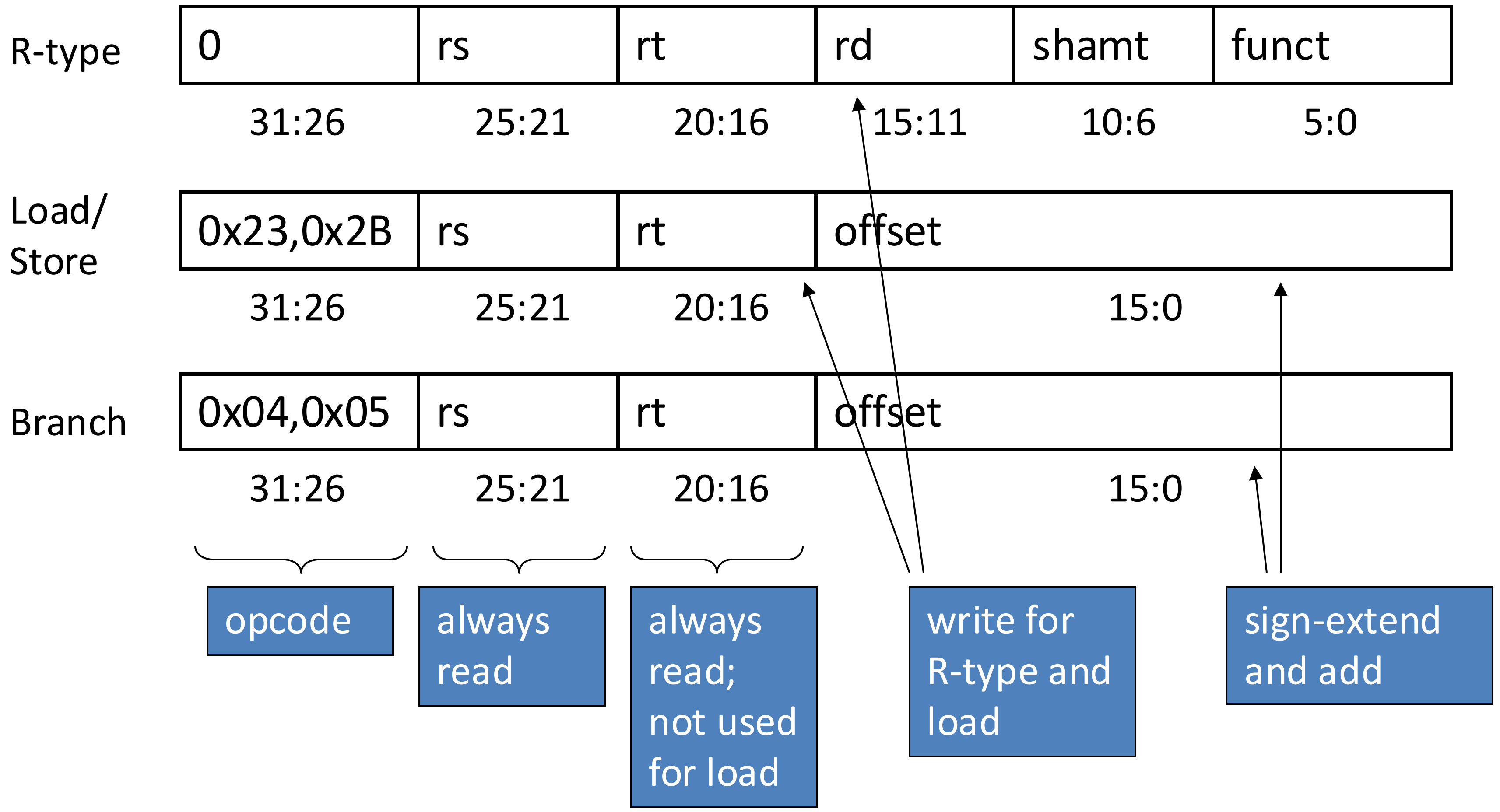
- Derived from truth table using sum of products
- Allow us to encode arbitrary functions
- Used to derive control signals in the data path

Datapath With Control



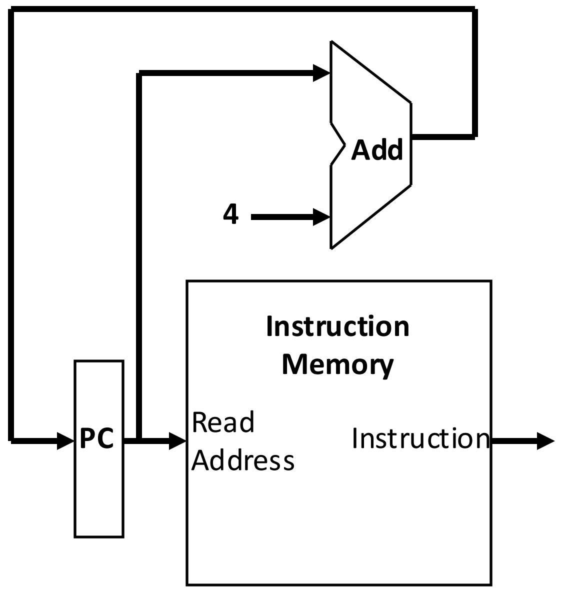
The Main Control Unit

Control signals derived from instruction opcode

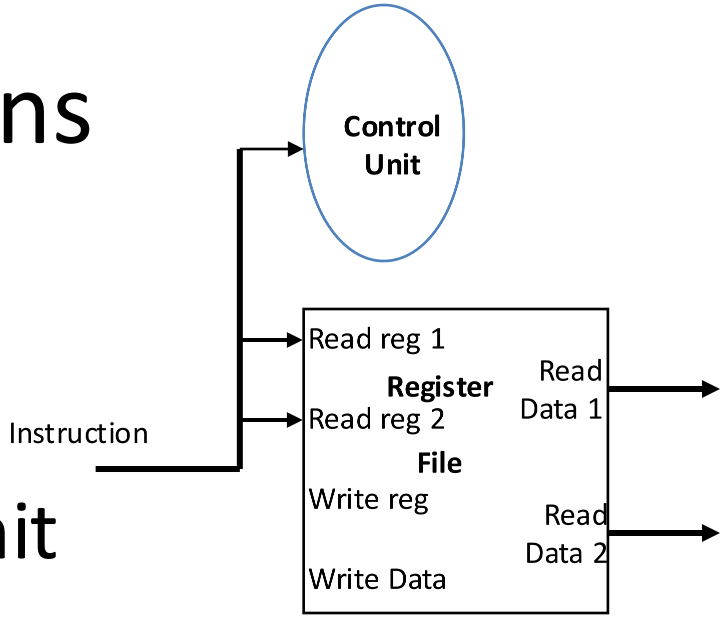


Fetching Instructions

- Read instruction from Instruction Memory
- Updating PC value to address of next (sequential) instruction
- PC is updated every clock cycle, so it does not need an explicit write control signal just a clock signal
- Read from memory each time, so we don't need an explicit control signal



Decoding Instructions



- Send fetched instruction’s opcode to the main control unit

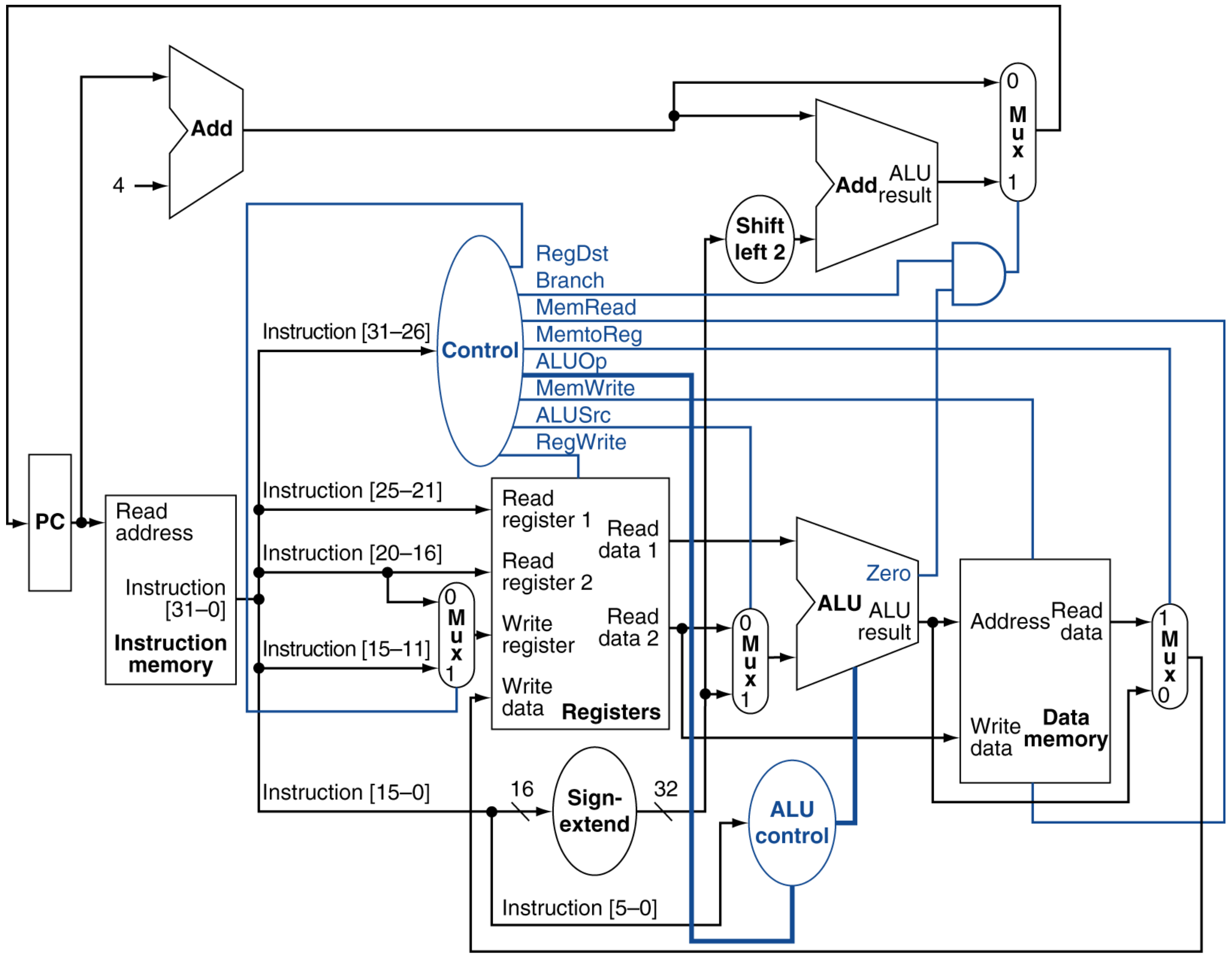
R-type	0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0
Load/ Store	0x23,0x2B	rs	rt	address		
	31:26	25:21	20:16	15:0		

- Read two values from the Register File
- Register numbers are contained in the instruction (rs and rt)

Producing control signals

After reading opcode

- Produce most control signals
- Includes the ALUOp control signal—which goes to the ALU control unit—and the ALUSrc control signal which selects the ALU's second operand



For load/store, our ALU operation will be

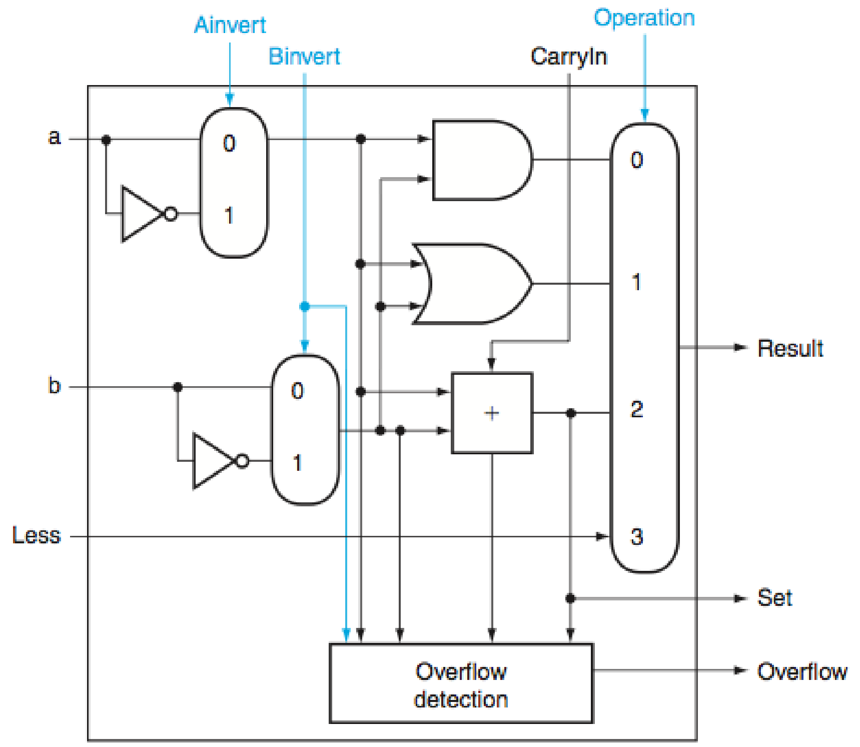
A. Add

B. And

C. Set less than

D. Subtract

E. None of the above

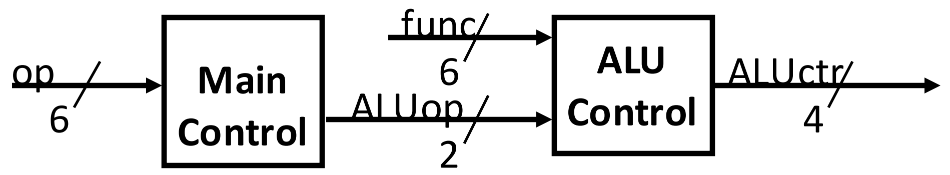


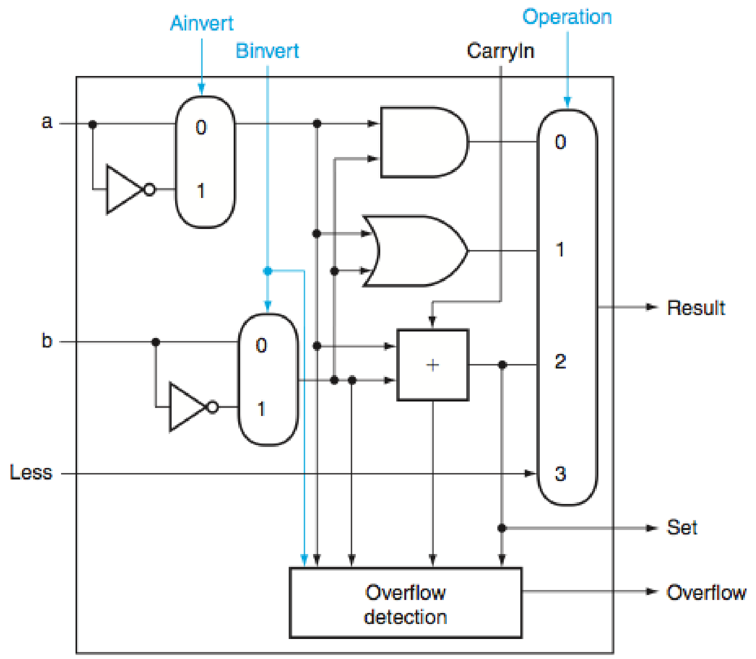
lw $t0, 4($t1)

ALU Control Unit

- Combinational logic (the main control unit) derives 2-bit ALUOp signal from opcode
- ALU Control Unit takes ALUOp and instruction funct field as inputs and derives a 4-bit ALU control signal

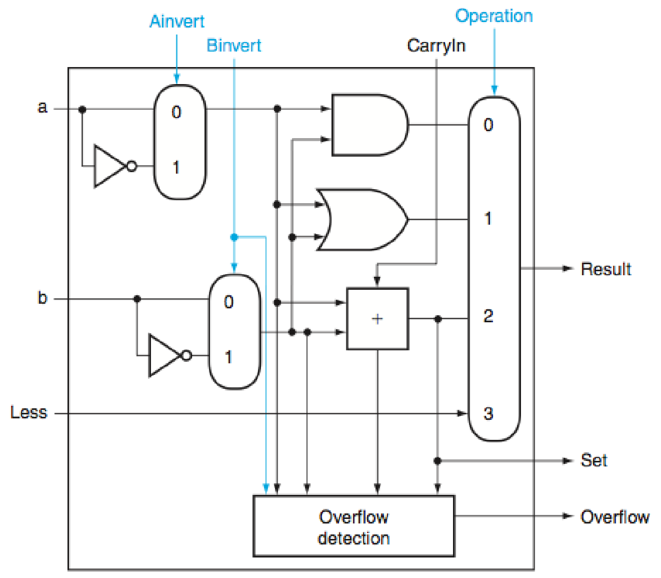
opcode	ALUOp	Operation	ALU function
lw	00	load word	add
sw	00	store word	add
beq	01	branch equal	subtract
R-type	10	arithmetic/logic	depends on funct





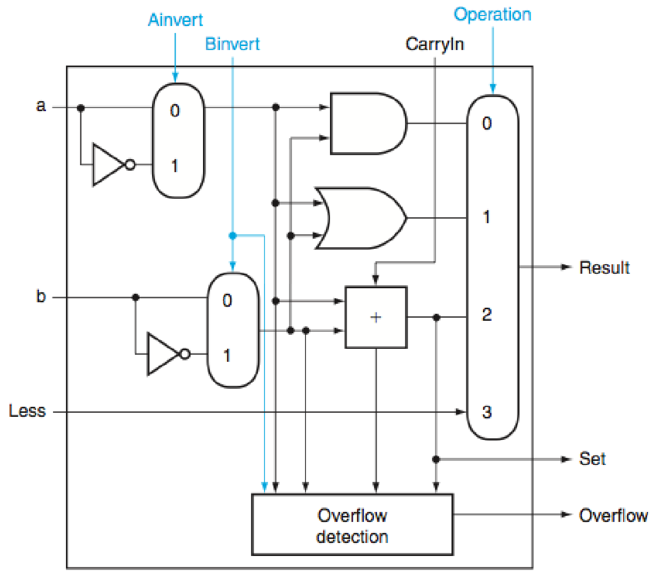
ALU Control signal

- ALU used for
  - Load/Store: op = add
  - Branch: op = subtract
  - R-type: op depends on funct field



ALU control	Function	Ainvert	Binvert/CarryIn0	Operation
0000	AND	0	0	00
0001	OR	0	0	01
0010	add	0	0	10
0110	subtract	0	1	10
0111	set-on-less-than	0	1	11
1100	NOR	1	1	00

ALU Control

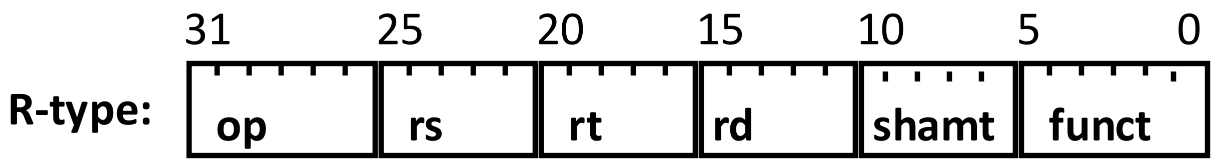


Takes as input 2-bit ALUop (derived from opcode) and 6-bit funct field; outputs 4 bits

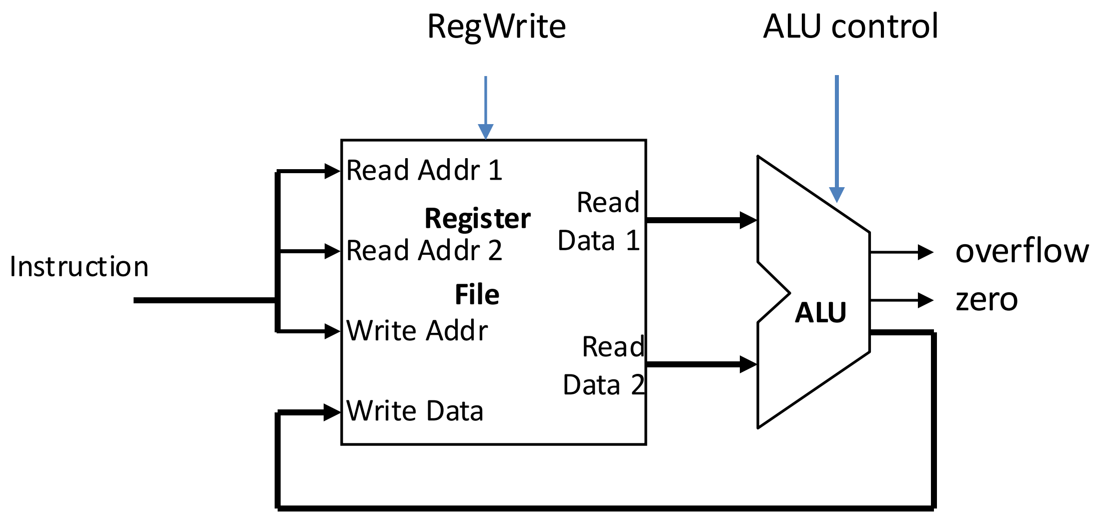
Instruction	ALUOp	funct	ALU function	Ainvert	Binvert	ALU operation
load word	00 (add)	XXXXXX	add	0	0	10 (add)
store word	00 (add)	XXXXXX	add	0	0	10 (add)
branch equal	01 (subtract)	XXXXXX	subtract	0	1	10 (add)
add	10 (r-type)	100000	add	0	0	10 (add)
subtract		100010	subtract	0	1	10 (add)
AND		100100	AND	0	0	00 (and)
OR		100101	OR	0	0	01 (or)
NOR		100111	NOR	1	1	00 (and)
set-on-less-than		101010	set-on-less-than	0	1	11 (less)

Executing R Format Operations

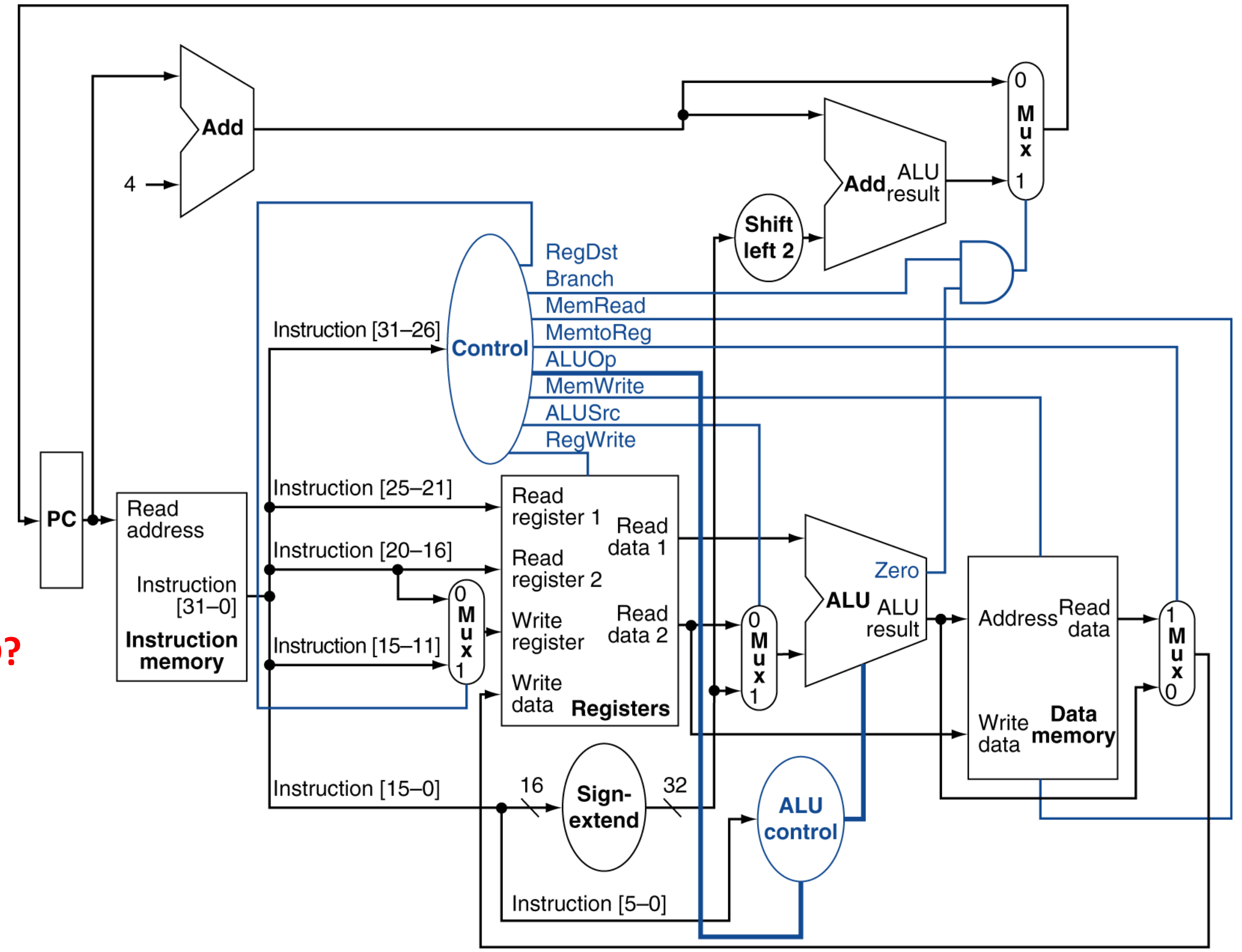
- R format operations (**`add, sub, slt, and, or`**)



  - perform operation (specified by funct) on values in rs and rt
  - store the result back into the Register File (into location rd)



Note that Register File is not written every cycle (e.g., **`sw`**), so we need an explicit write control signal for the Register File



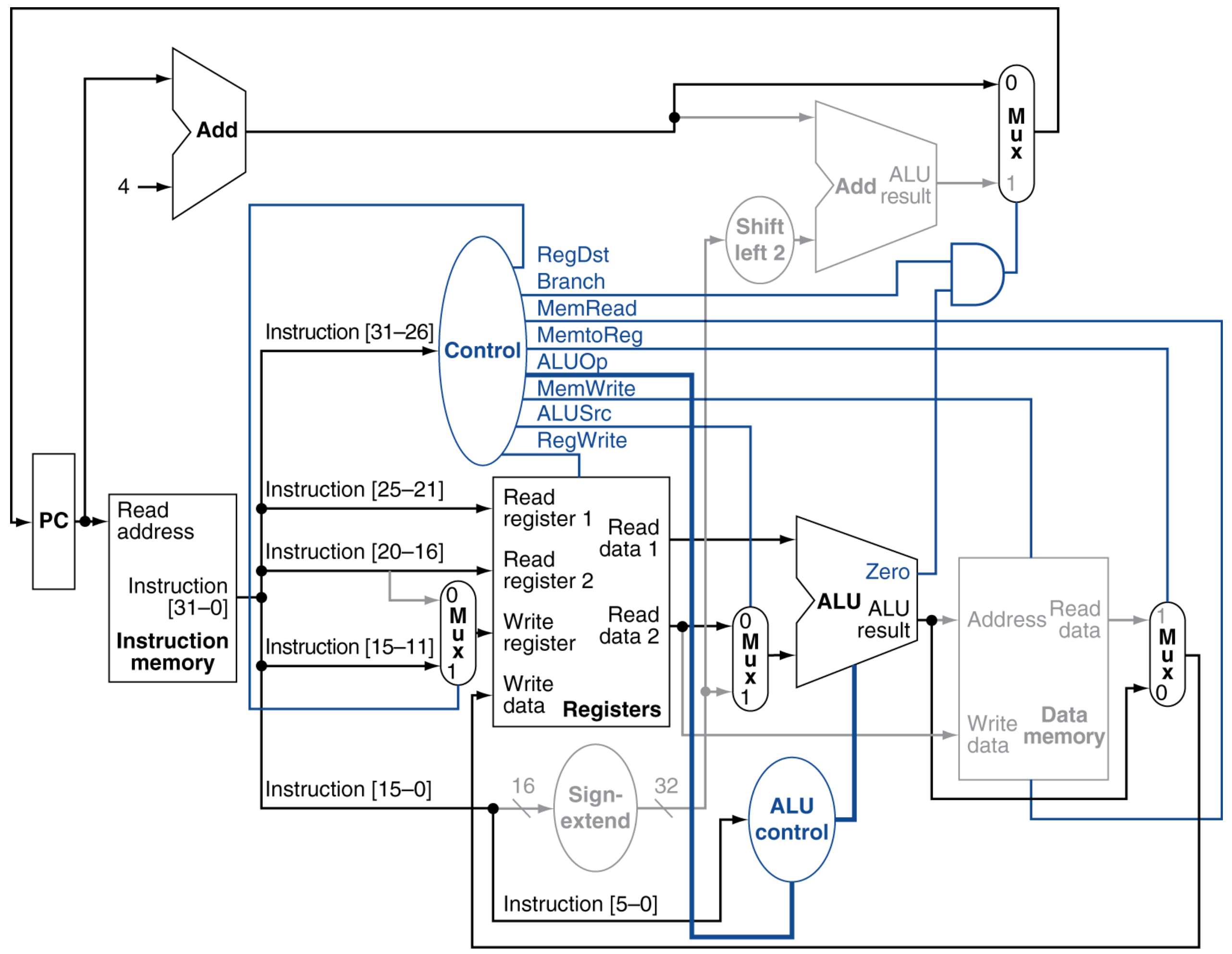
instruction control signals for ADD?

Select	RegDst	MemToReg
A	0	X
B	1	X
C	0	1
D	1	0
E	None of the above	

R-type

0	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0

R-Type Instruction



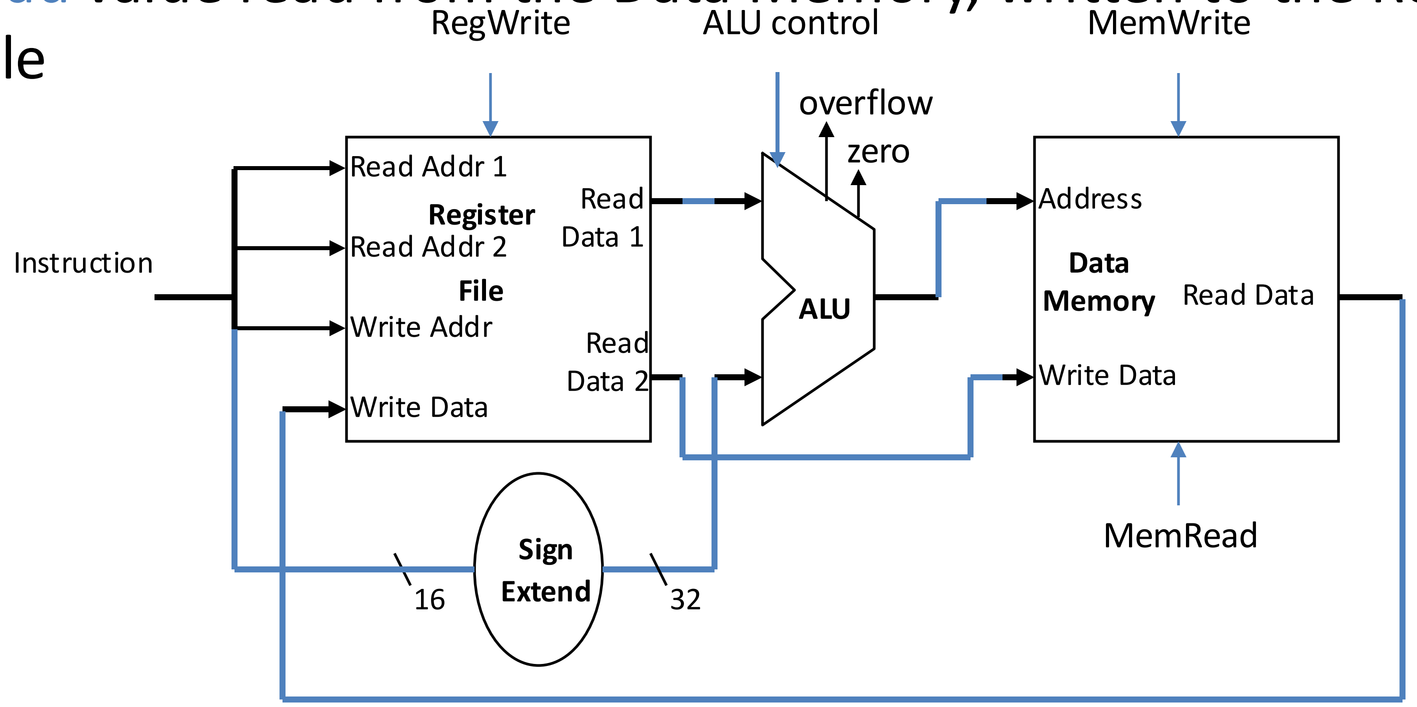
RegDst	
ALUSrc	
MemToReg	
RegWrite	

R-type

0	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0

Executing Load and Store Operations

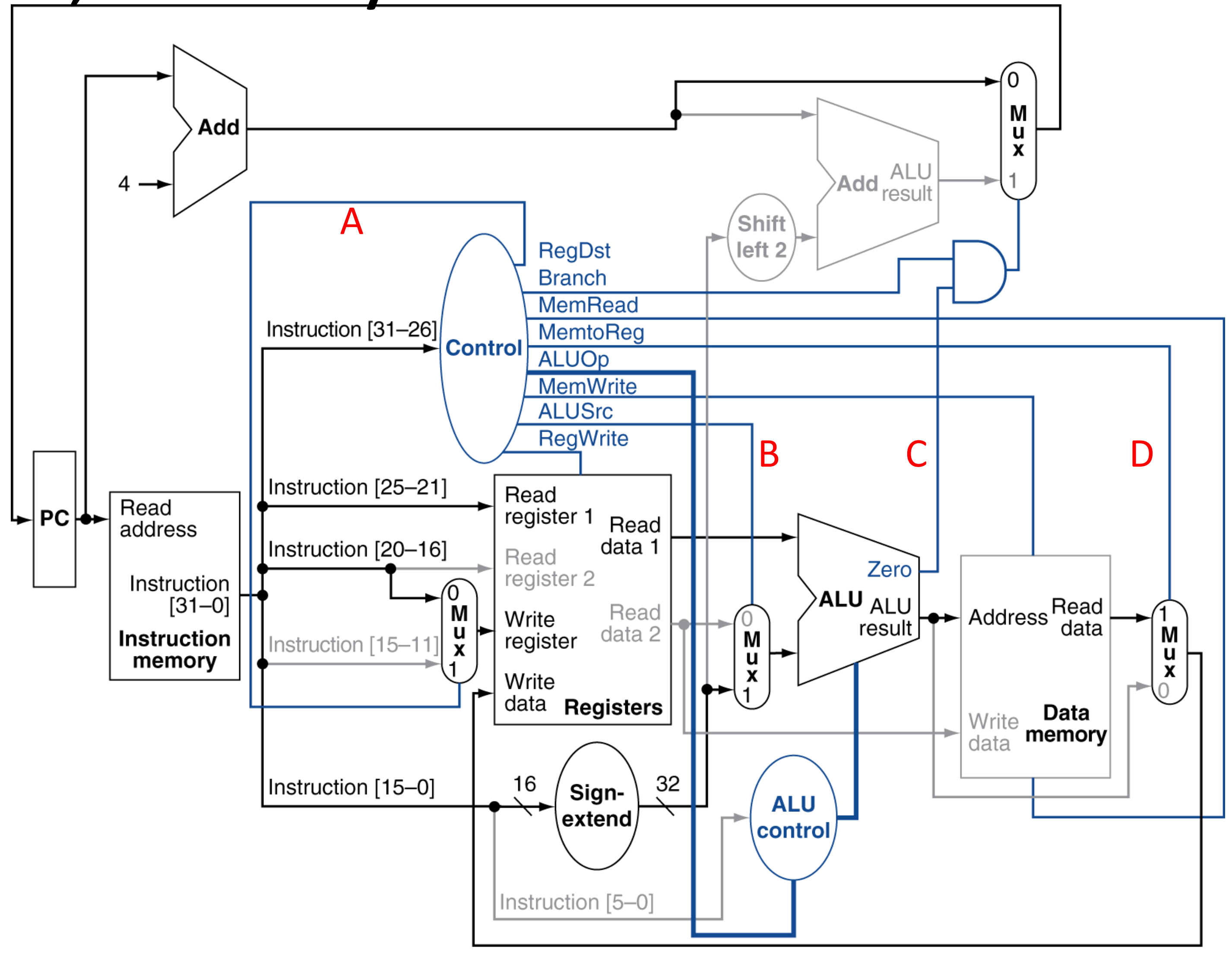
- compute memory address by adding base register to 16-bit signed-extended offset field
- store value written to the Data Memory
- load value read from the Data Memory, written to the Register File



Load/
Store

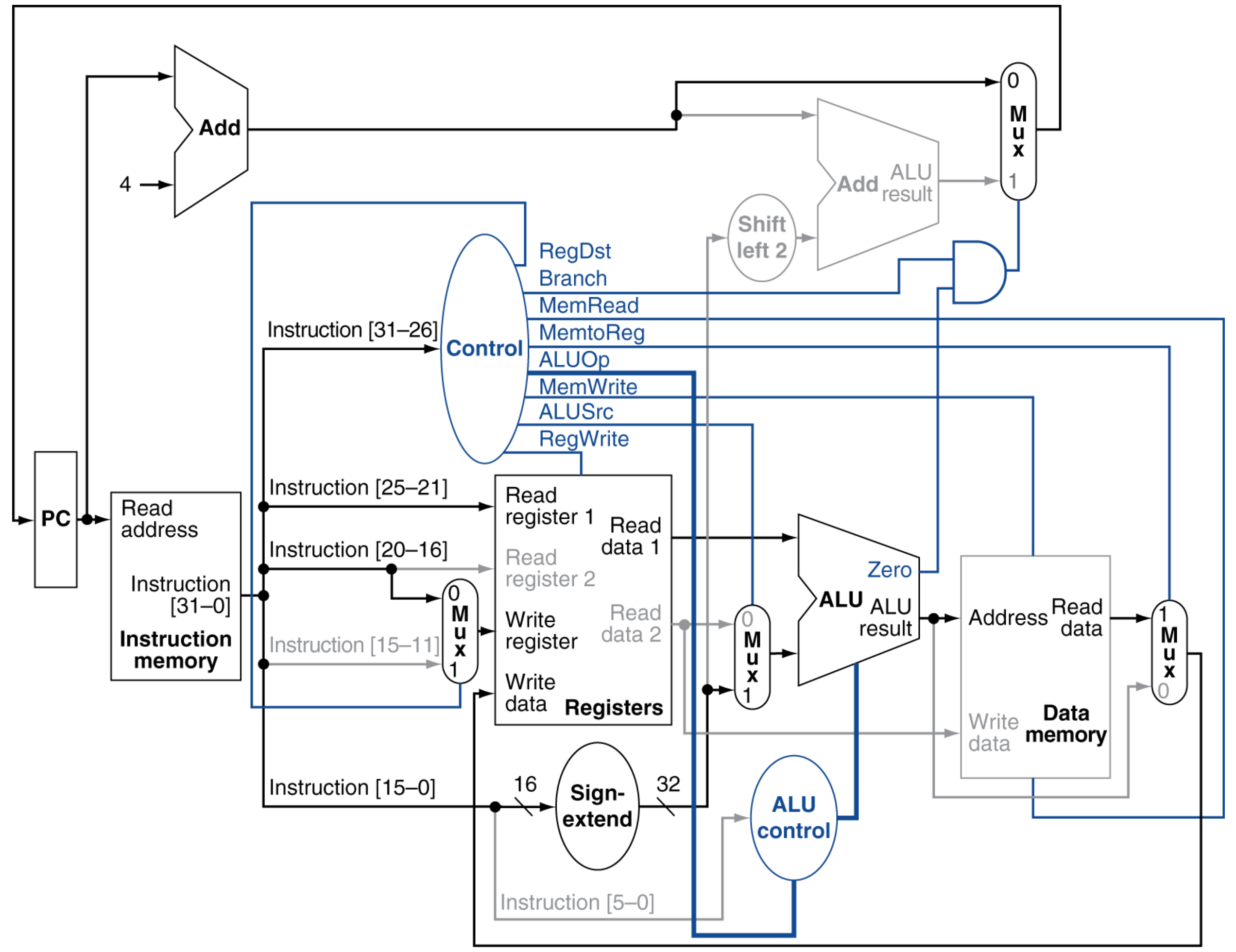
35 or 43	rs	rt	address
31:26	25:21	20:16	15:0

Which wire, if always set to 1 would break lw?



Load/ Store	35 or 43	rs	rt	address
	31:26	25:21	20:16	15:0

Load Instruction



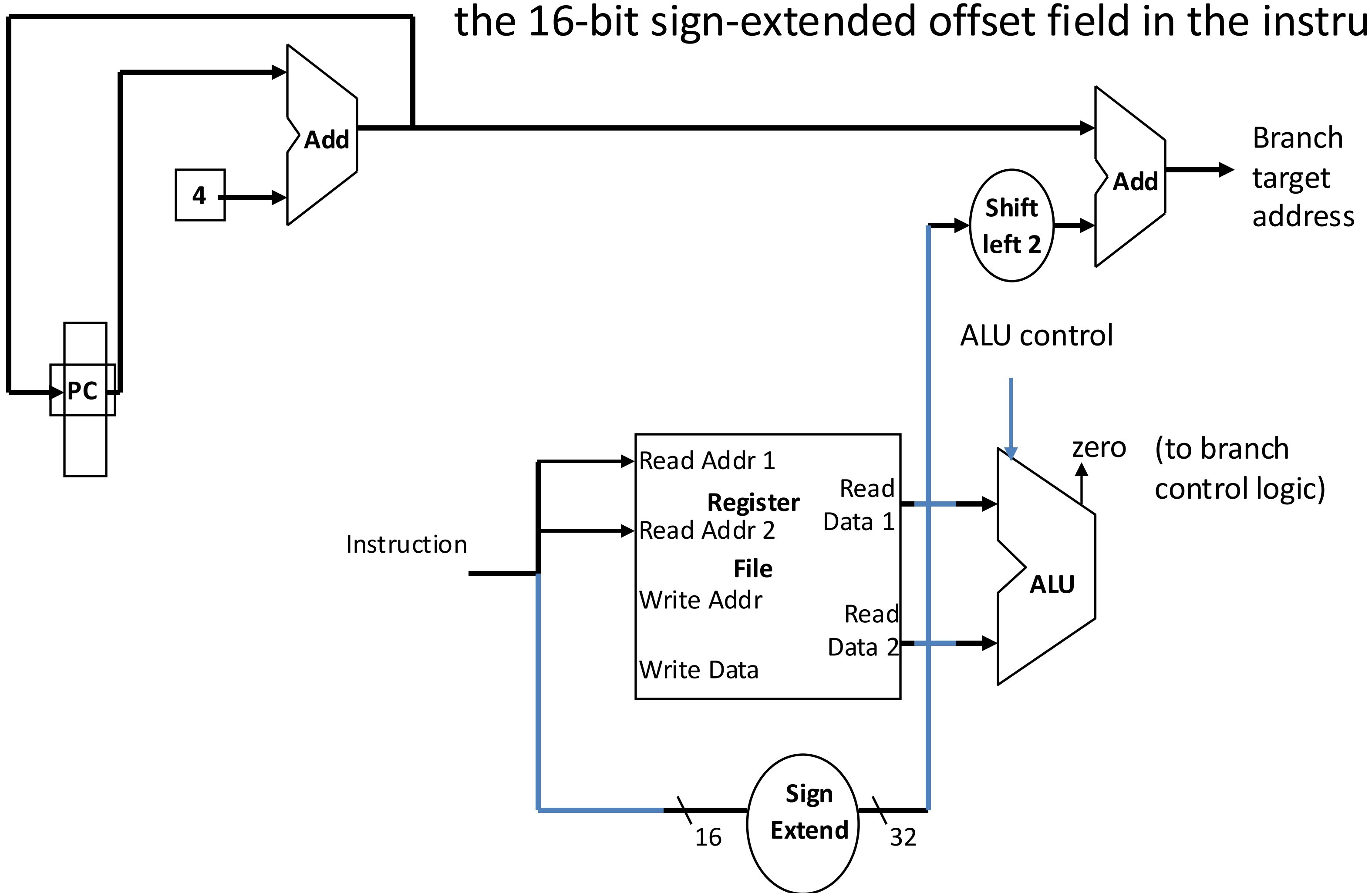
RegDest	
MemWrite	
MemRead	
MemtoReg	
RegWrite	

Load/ Store

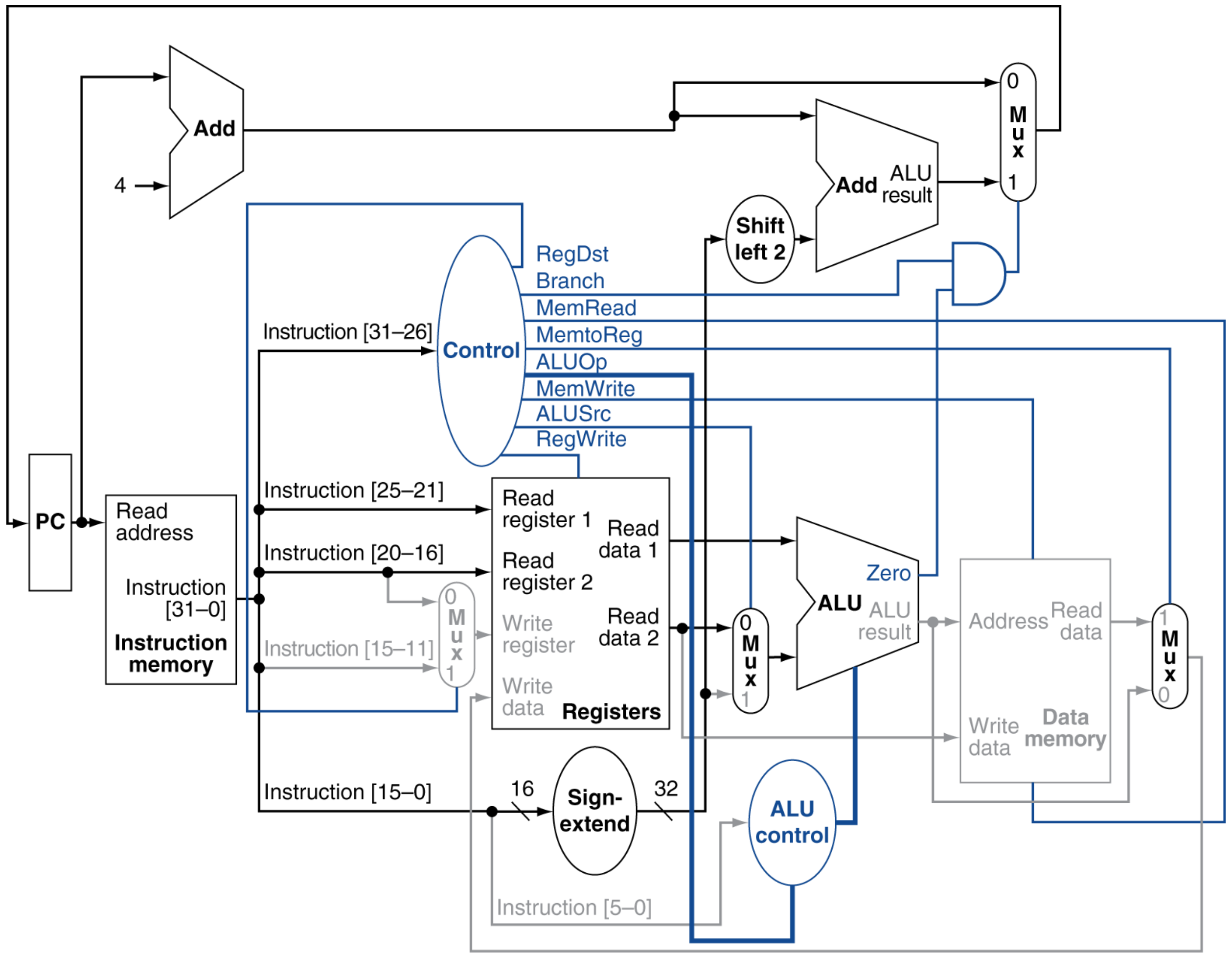
35 or 43	rs	rt	address
31:26	25:21	20:16	15:0

Executing Branch Operations

- Branch operations involve
  - compare the operands read from the Register File during decode for equality (`zero` ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit sign-extended offset field in the instruction



Branch-on-Equal Instruction



Branch	
MemWrite	
MemRead	
AluSrc	
RegWrite	

branch	35 or 43	rs	rt	address
	31:26	25:21	20:16	15:0

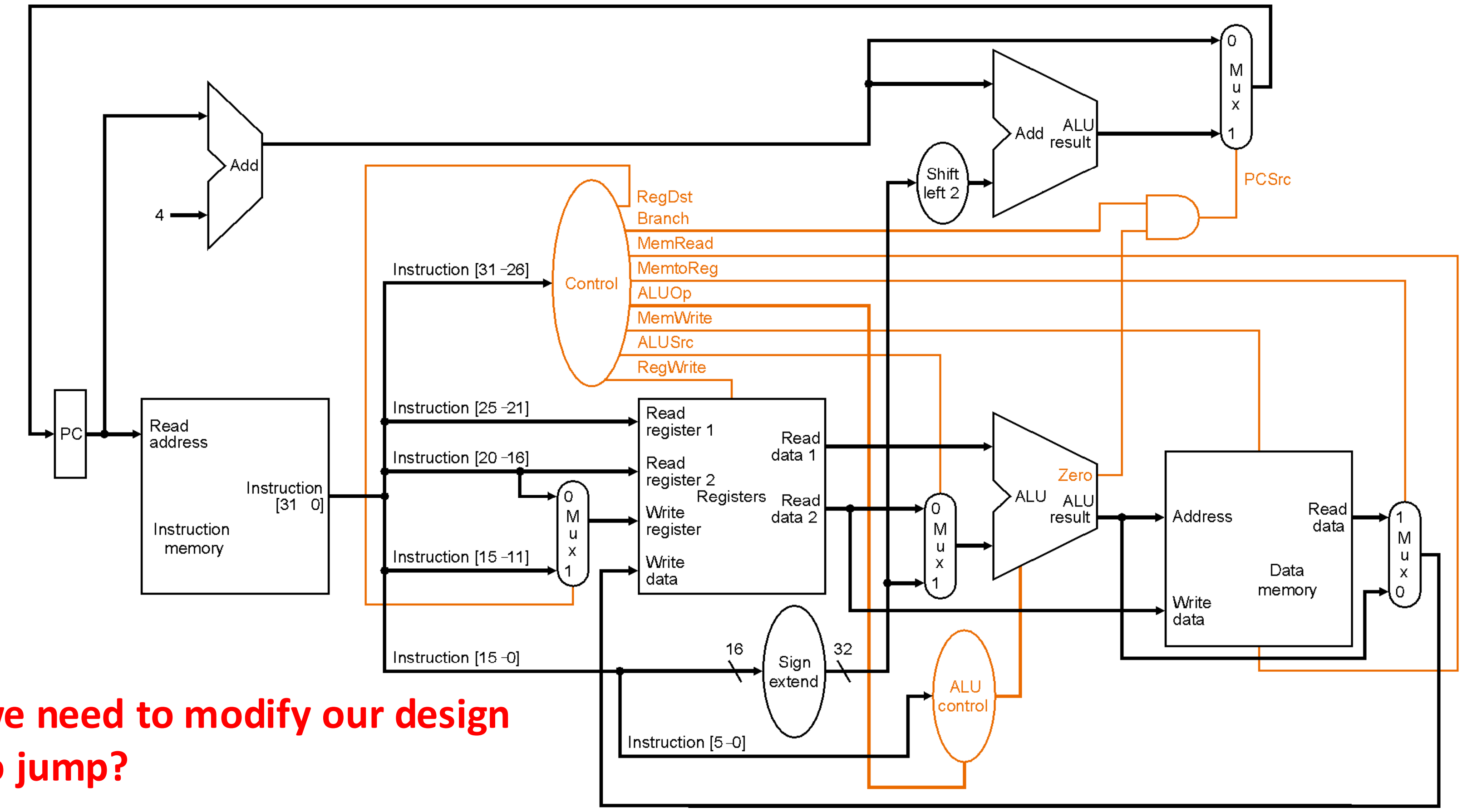
Control Truth Table

		R-format	**lw**	**sw**	**beq**
Opcode		000000	100011	101011	000100
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Implementing Jumps

Jump	2	address
	31:26	25:0

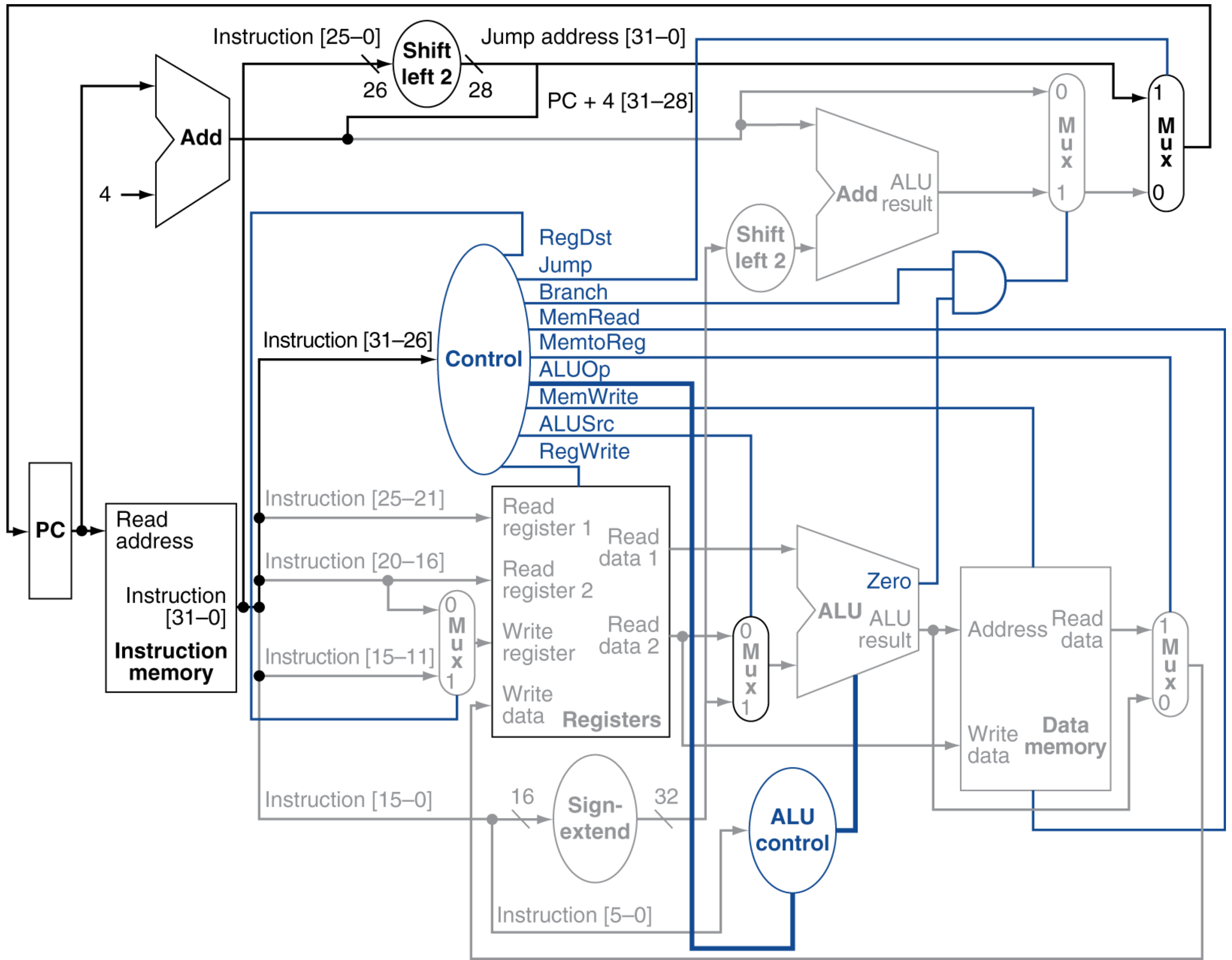
- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of PC + 4
  - 26-bit jump address
  - 00



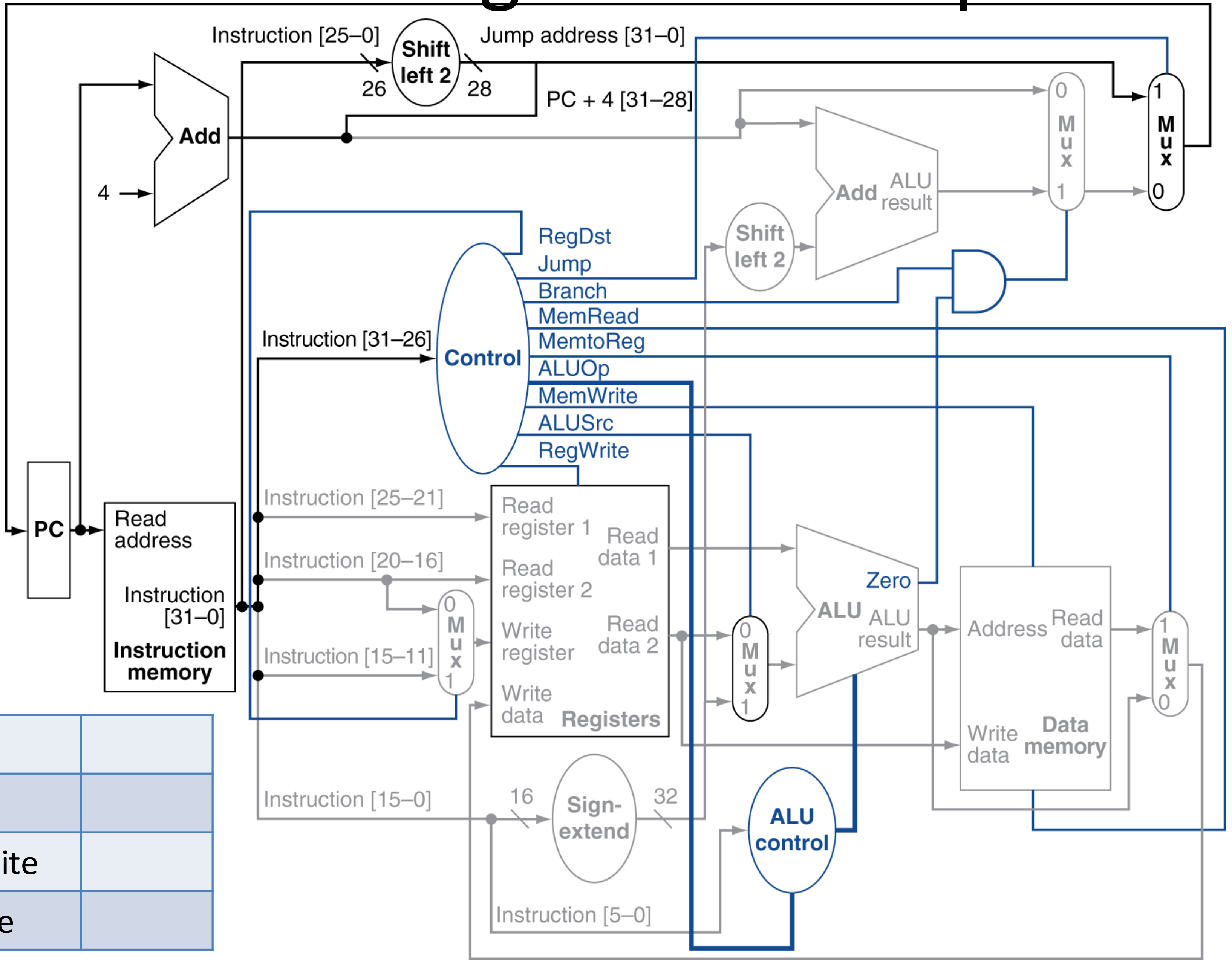
Do we need to modify our design to do jump?

Select	Best Answer
A	Yes – we need both new control and datapath.
B	Yes – we need just datapath.
C	No – but we should for better performance.
D	No – just changing control signals is fine.
E	Single cycle can’t do jump register.

Datapath With Jumps Added

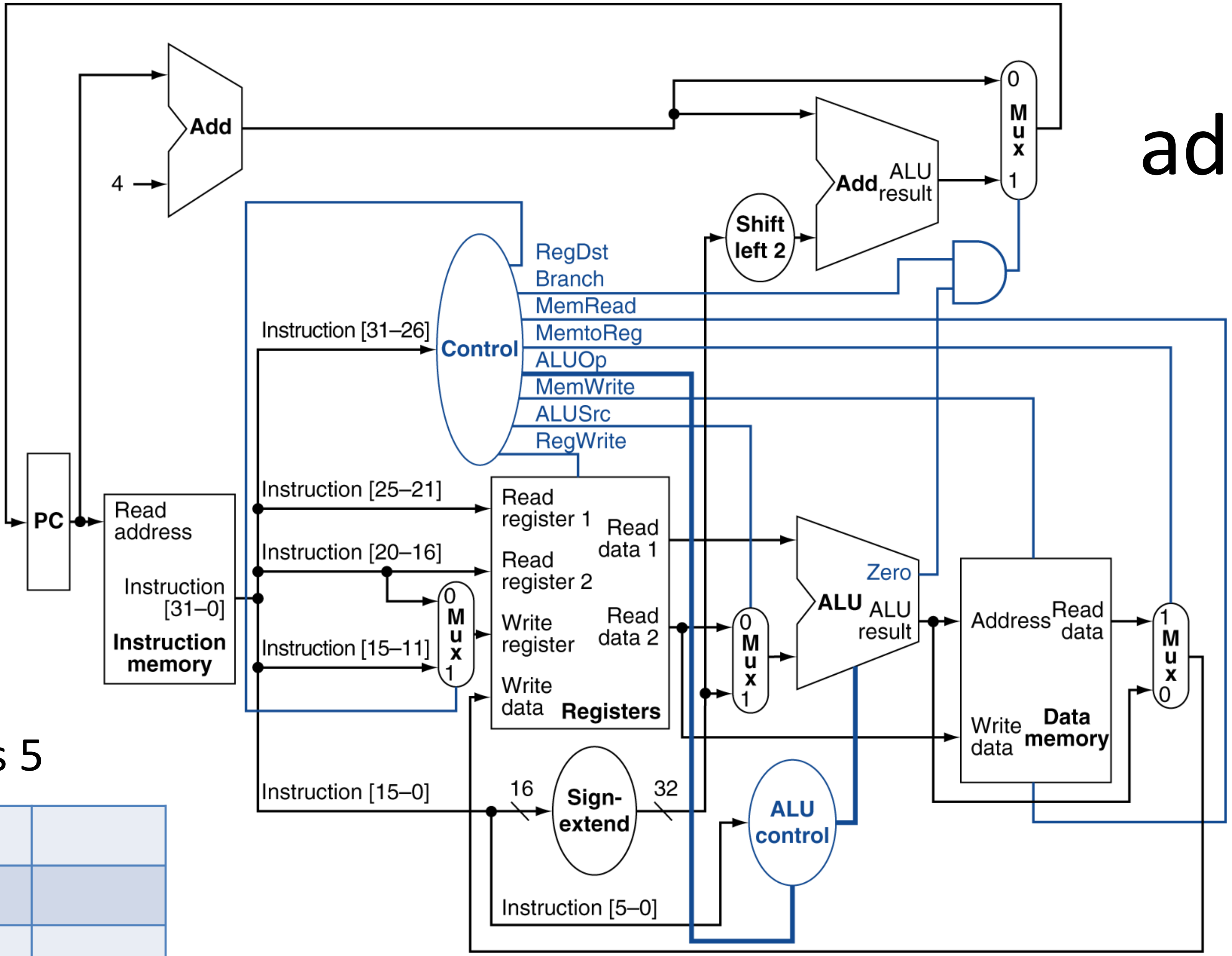


What will the Signals for Jump be?



Jump	
Branch	
MemWrite	
RegWrite	

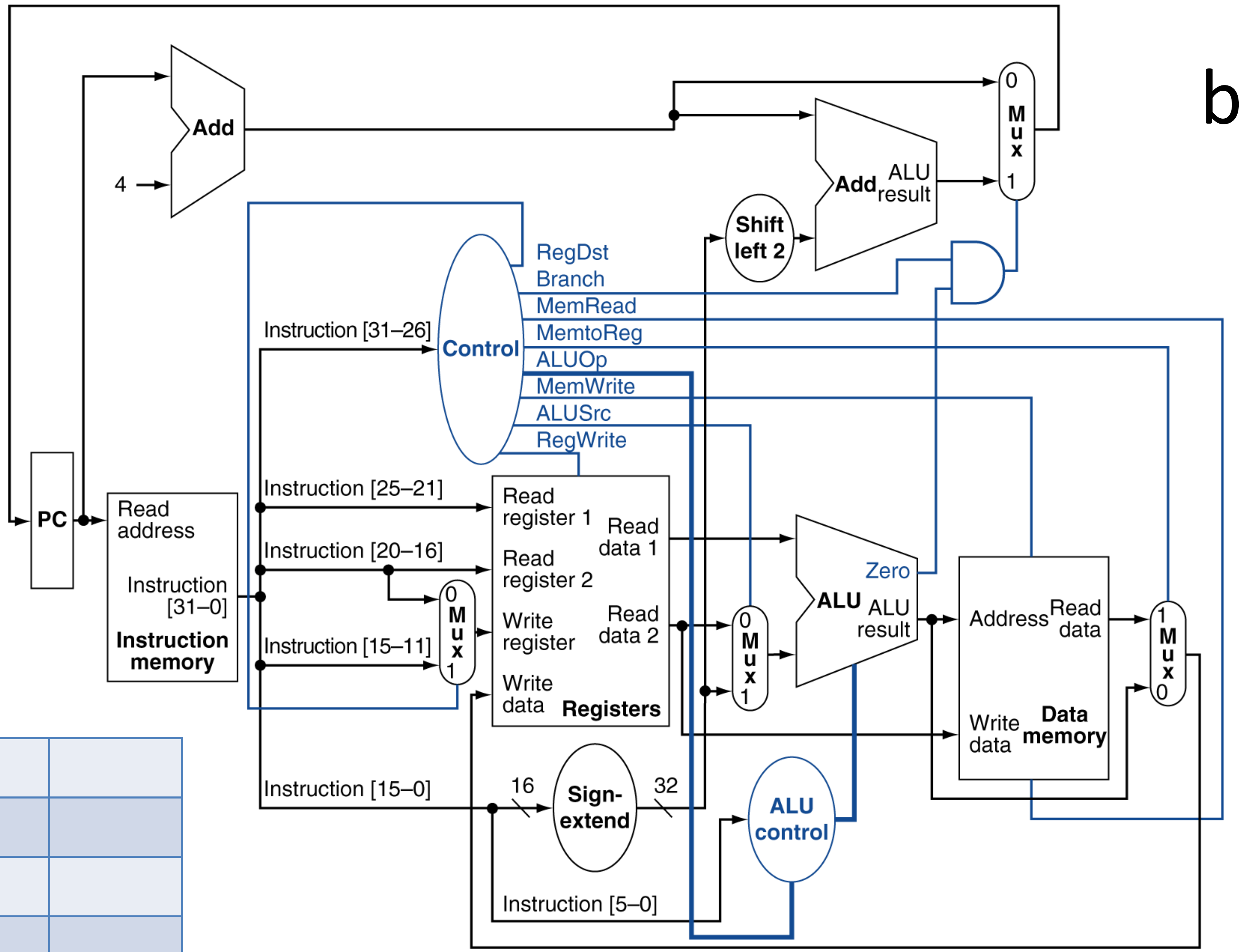
addi $t1, $t2, 6



$t2 holds 5

RegDst	
AluSrc	
MemtoReg	
RegWrite	

Op = 0x08	rs = 10	Rt = 9	Imm = 6
31:26	25:21	20:16	15:0



beq $t0, $t3, label

PC = 0x10FACE04

$t0 holds 5

$t3 holds 5

RegDst	
AluSrc	
Branch	
RegWrite	

Op = 0x04	rs = 8	Rt = 11	Imm = 0x000C
31:26	25:21	20:16	15:0