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 $33,800 in 2026)
- 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 for each instruction or use it in different ways, e.g.,
  - add and addi use the ALU but not the data memory;
  - ld, sd, add, and addi require the ALU to perform an addition whereas beq and sub require the ALU to perform a subtraction

- How do we know which elements to use or what operation to have them perform? The information is encoded in the instruction itself!

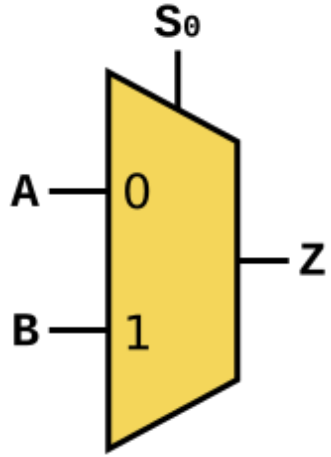
What control signals do we need?

Our data path consists of four major components

1. Instruction memory and PC register
2. Register file
3. ALU
4. Data memory

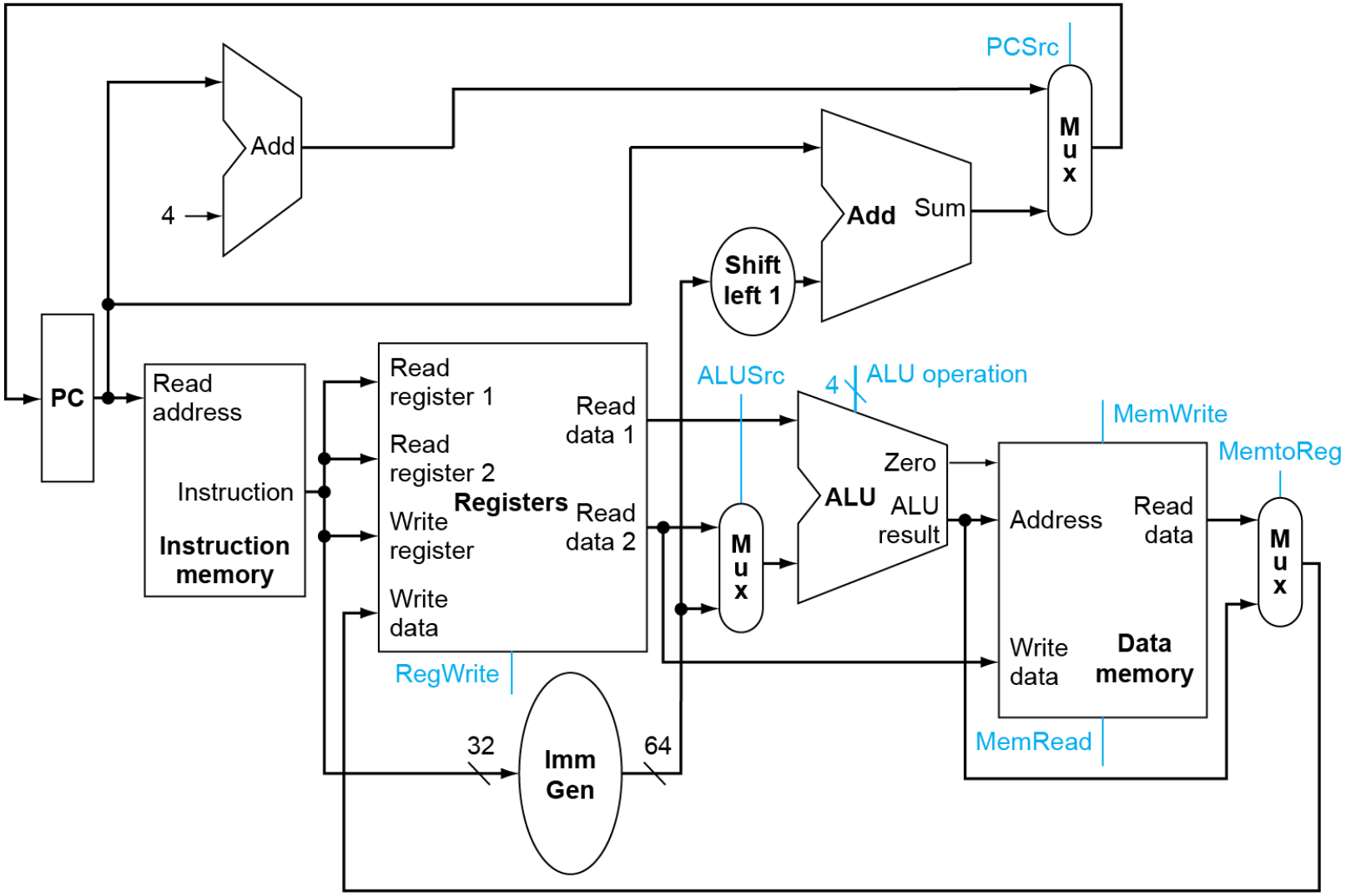
For each component, we're going to consider what control signals it needs to perform the correct operation for the instruction on the correct data

Warmup: Consider the multiplexer below. What value must the select input S_0 have for the output Z to have the same value as input B?

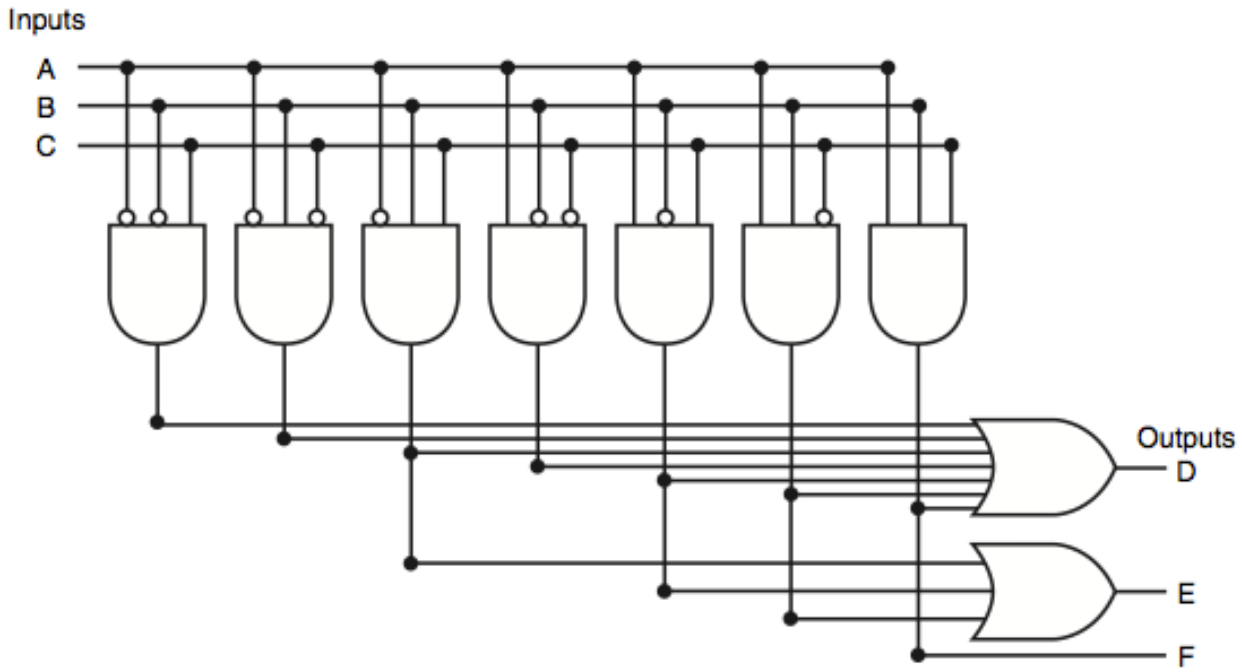


A. 0
B. 1
C. 2
D. It depends on the value of A
E. It depends on the value of both A and B

Blue inputs are some of the control signals
Today, we're going to hook some logic up to them

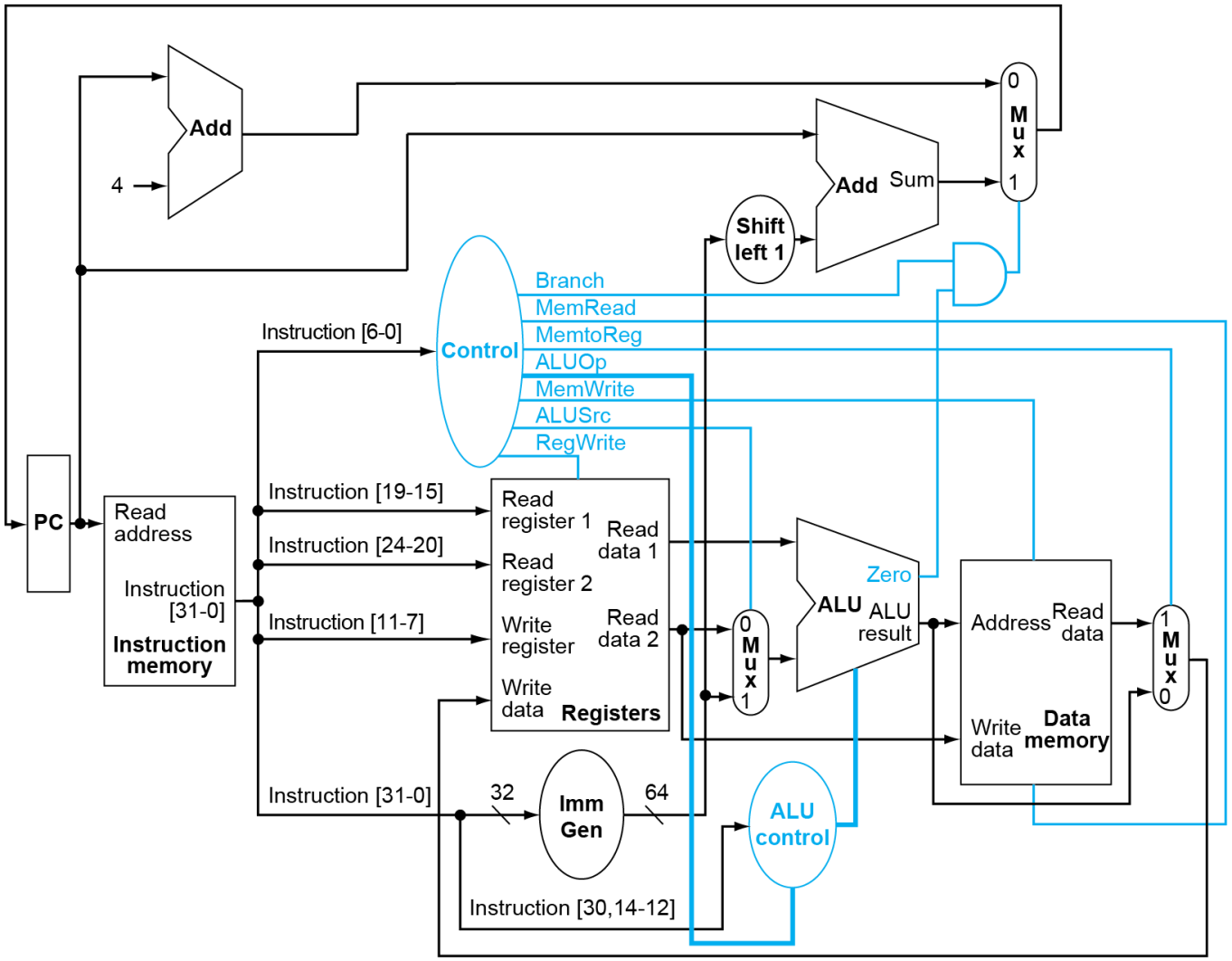


Recall: PLAs

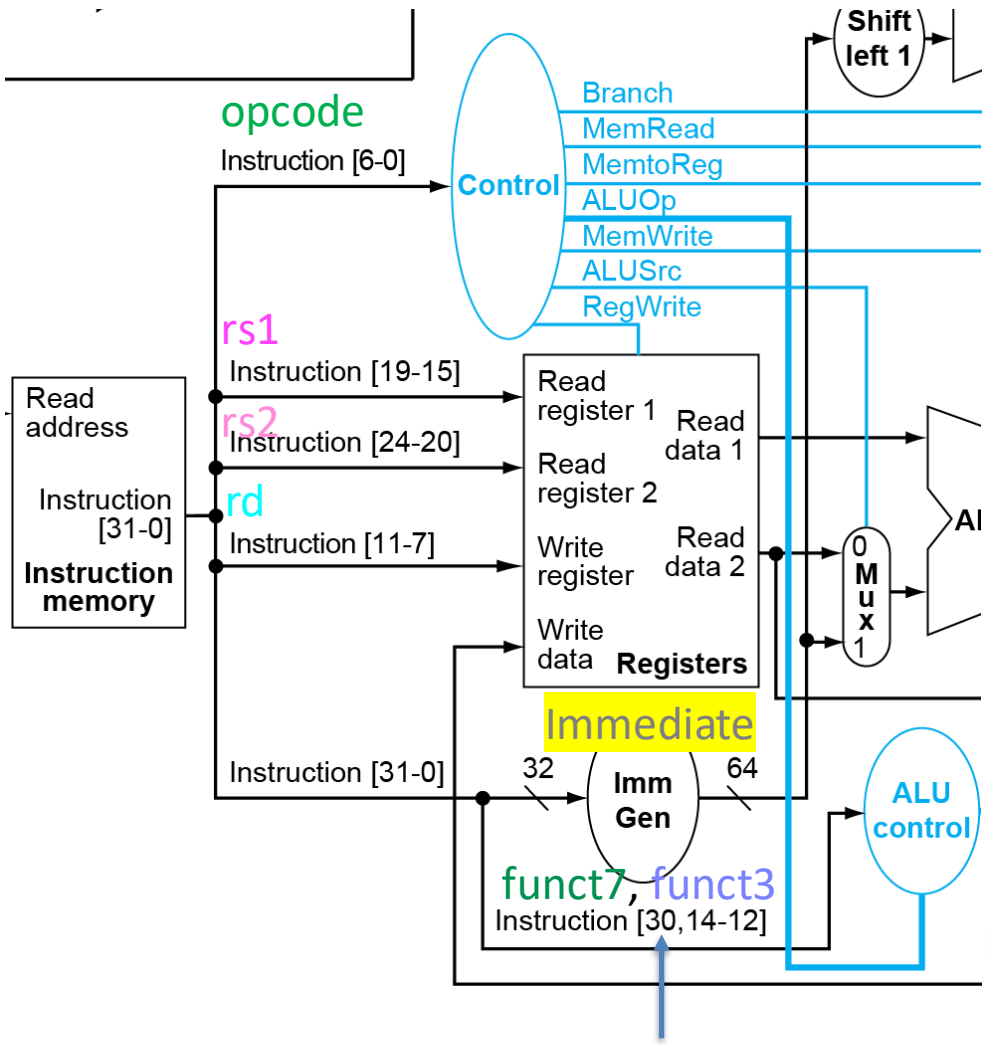


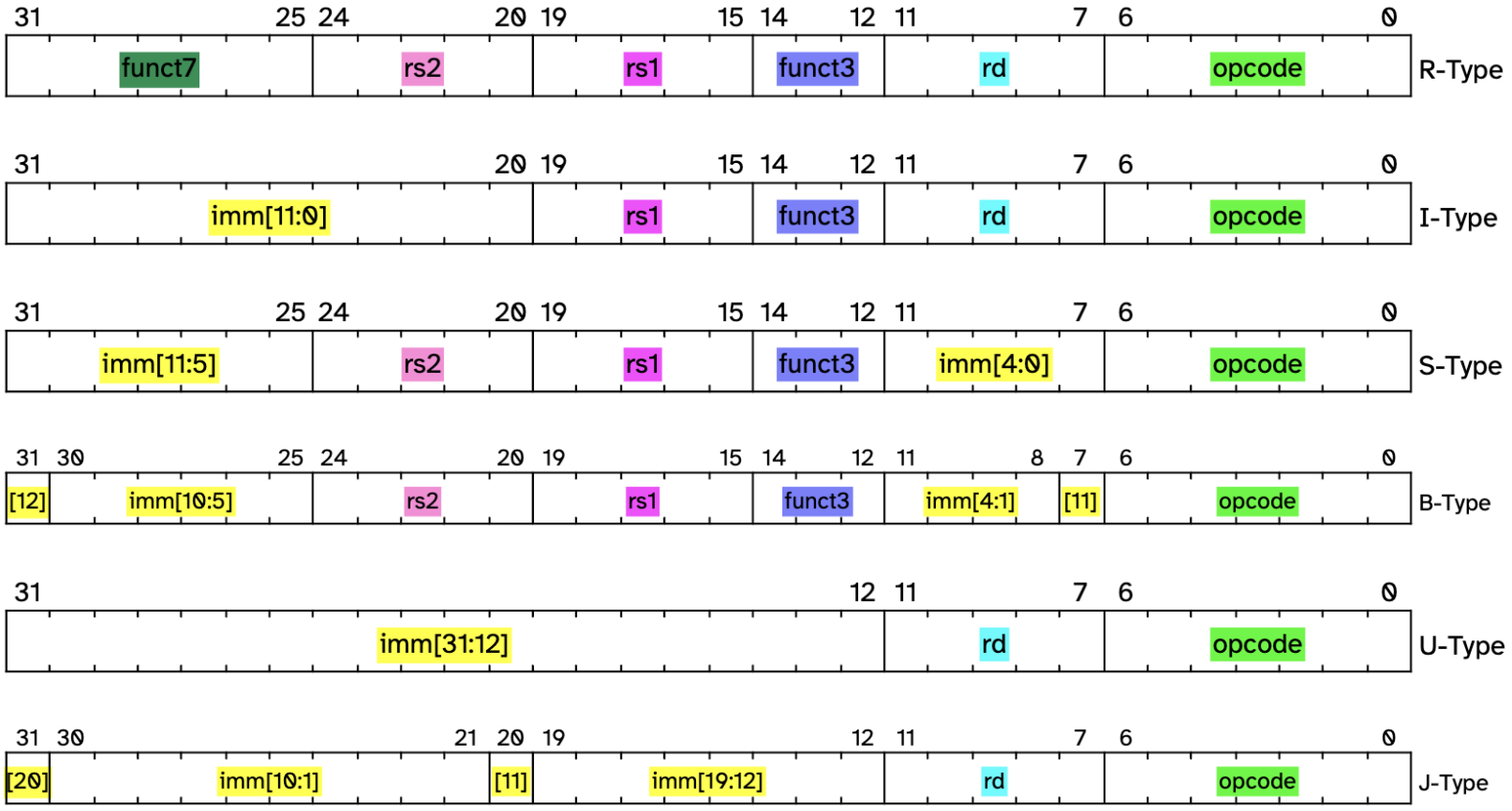
- Derived from truth table using sum of products
- Allow us to encode arbitrary functions
- **Used to derive control signals in the data path from bits in the instruction**

Datapath With Two Control Units



Splitting instructions into fields





The diagram shows all of funct3 but only bit 30 of funct7 because this implementation supports only a handful of instructions
In general, *all* of funct7 (bits 31–25) are needed by the ALU control

When the processor is executing the instruction

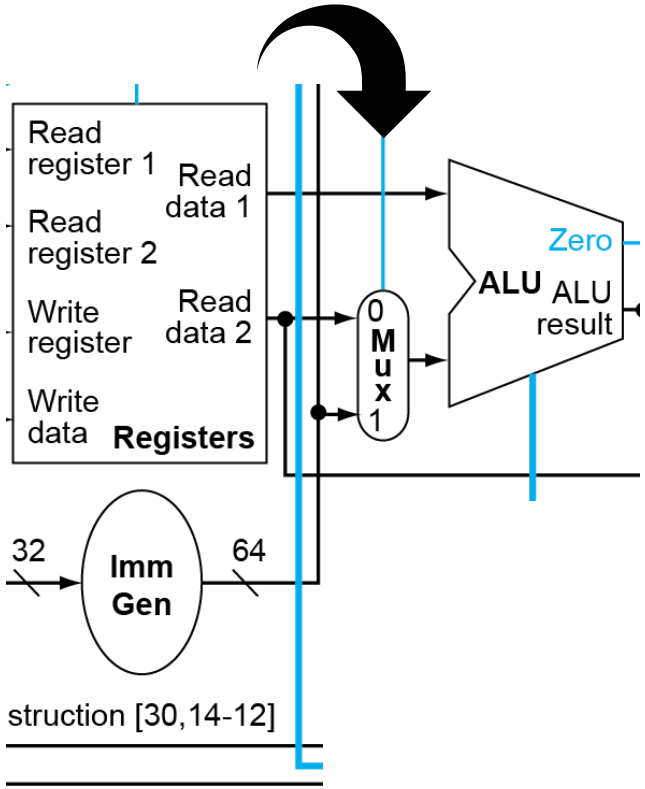
```
or t0, t1, t2
```

what value does the select input to the multiplexer need to have?

A. 0

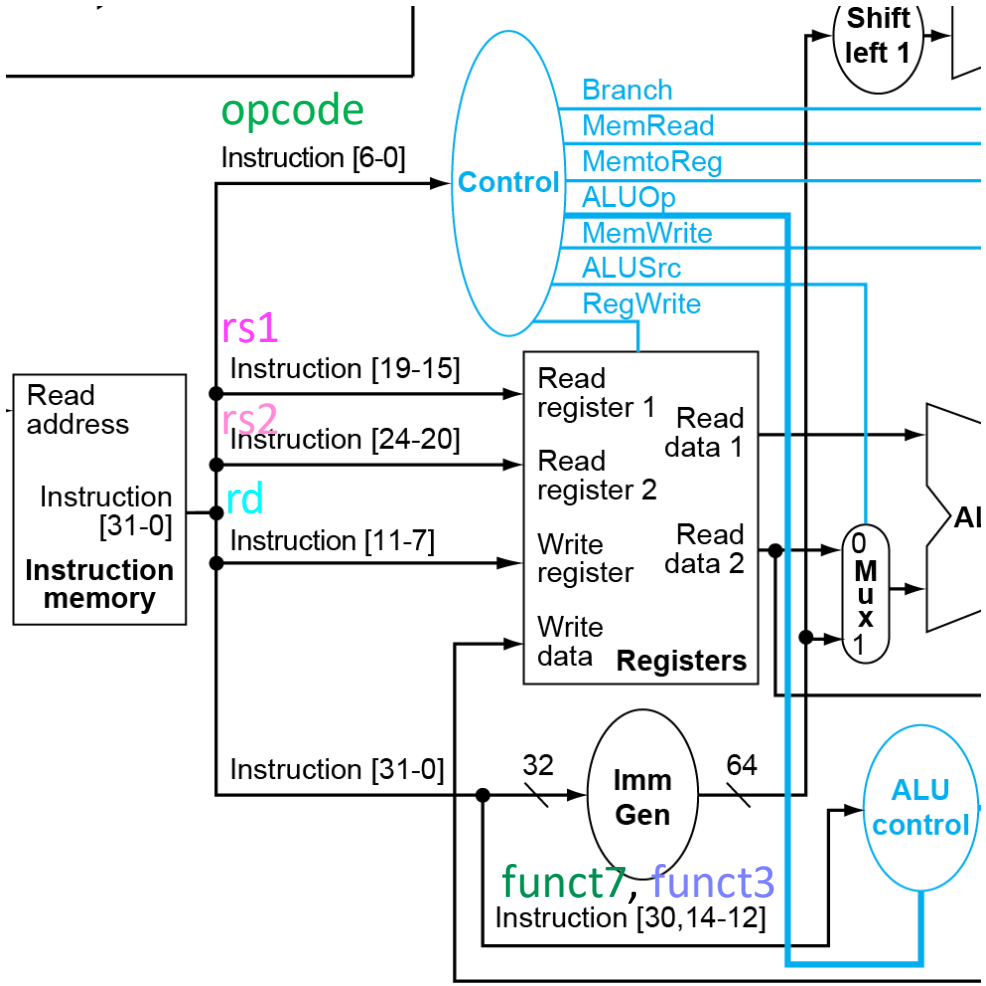
B. 1

C. X (a “don’t care”)



The Main Control Unit

Control signals derived from instruction opcode



Input: 7-bit opcode

Output:

- One 2-bit ALUOp signal to control the ALU
- Six 1-bit signals to control multiplexers, the register file and the data memory

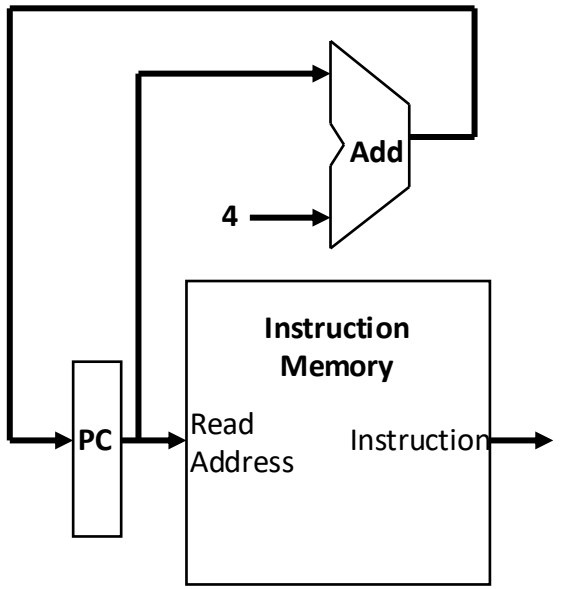
Let’s derive some control signals!

- We’ll tackle this systematically by following the data as it flows through the data path

- Recall that every instruction has 3 steps to be executed:
  1. Fetch the instruction from instruction memory and update the PC
  2. Decode the instruction and get the operands for the ALU out of the register file or from an immediate field in the instruction
  3. Execute the instruction and store the result back into the register file

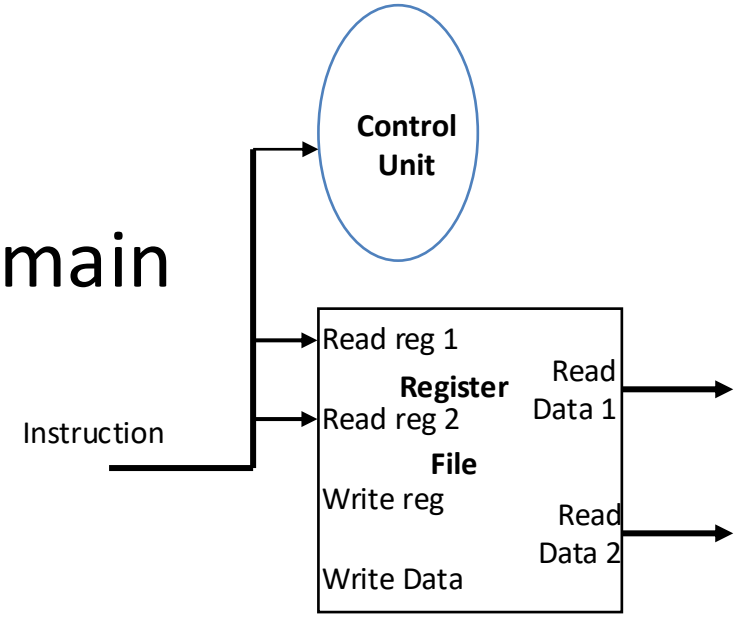
Fetching Instructions

- Read instruction from instruction memory
- Update 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 for the instruction memory or PC**



Decoding Instructions

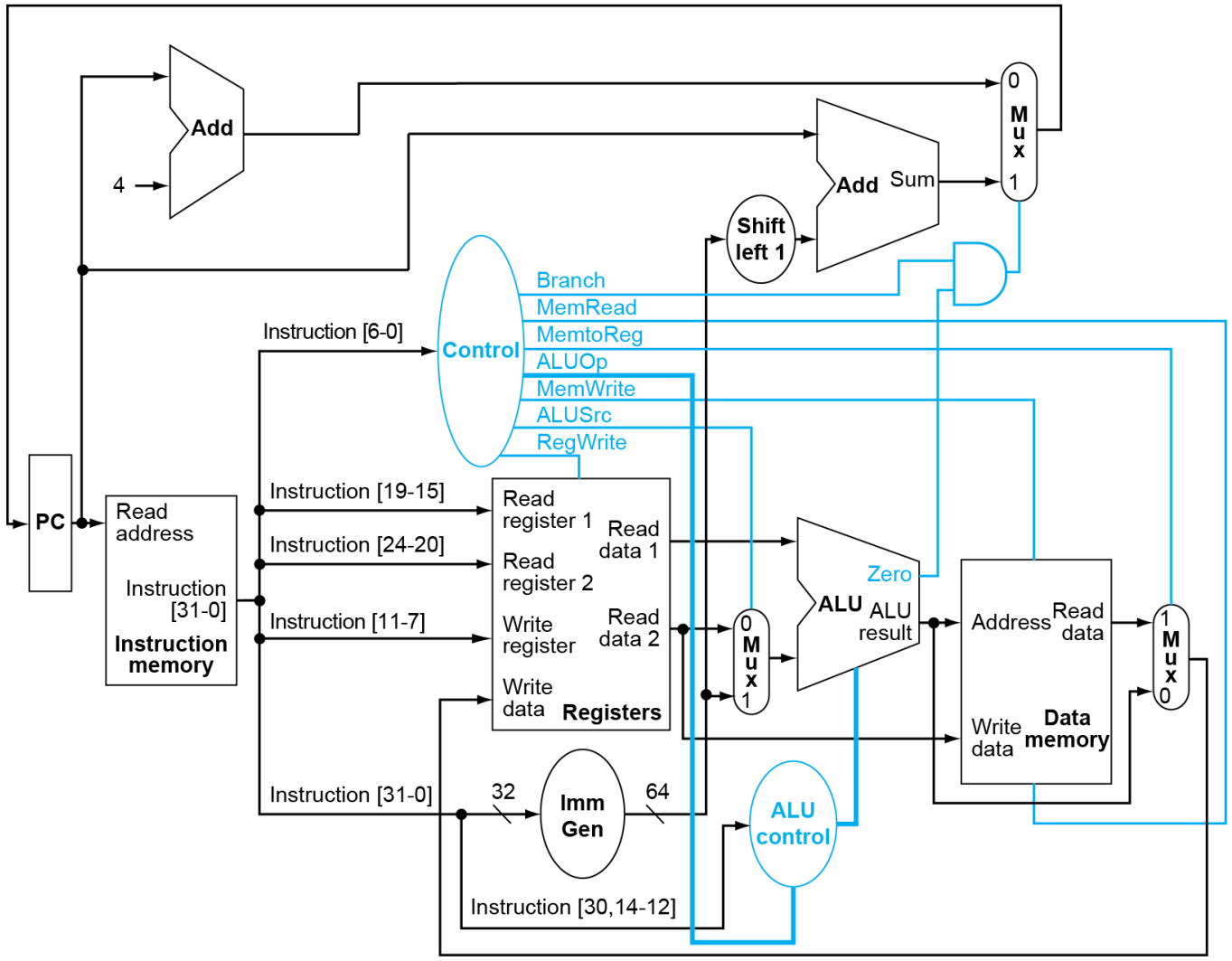
- Send fetched instruction's opcode to the main control unit
- Read two values from the register file
- Register numbers are contained in the instruction (fields rs1 and rs2)
- **Decode happens for every instruction; this is where control signals are produced**



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



When the processor is executing the instruction

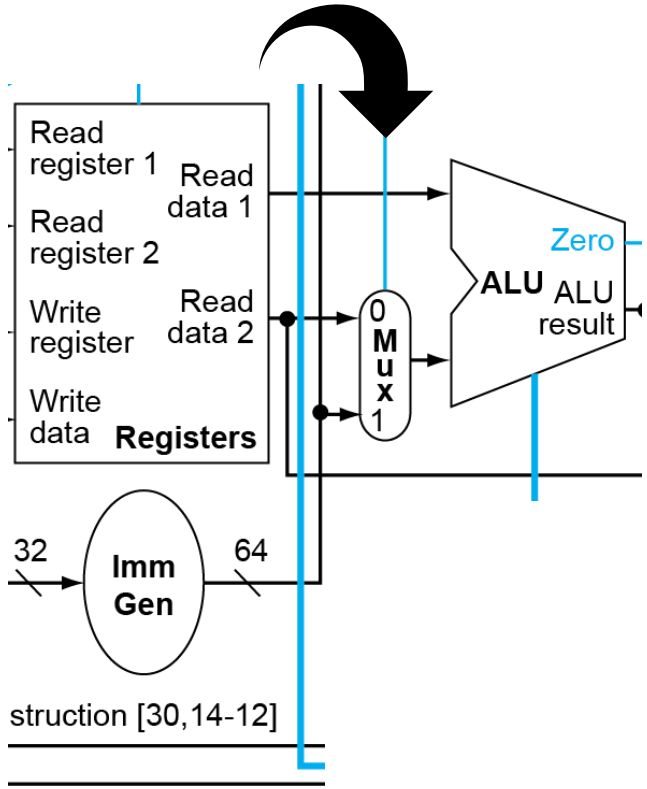
```
sd s0, 24(sp)
```

what value does the select input to this multiplexer need to have?

A. 0

B. 1

C. X (a “don’t care”)



For load/store, our ALU operation will be

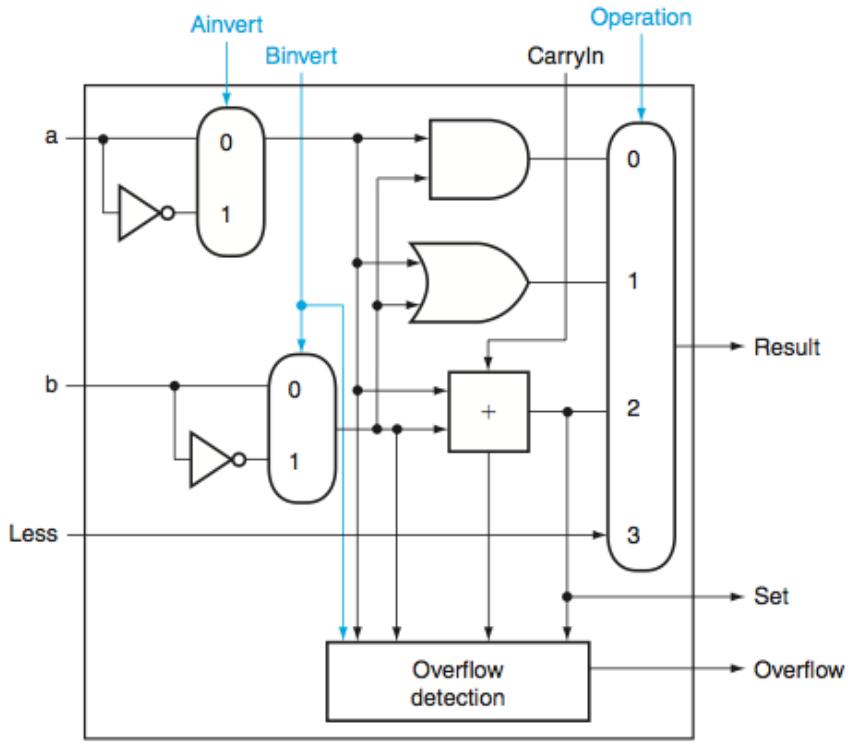
A. Add

B. And

C. Set less than

D. Subtract

E. None of the above

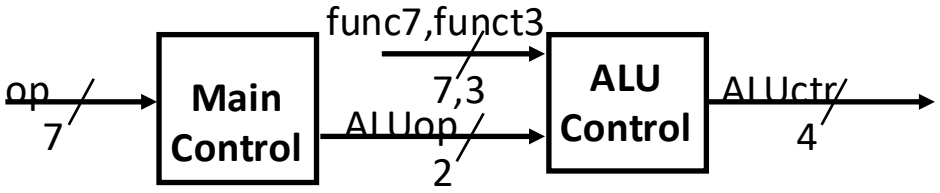


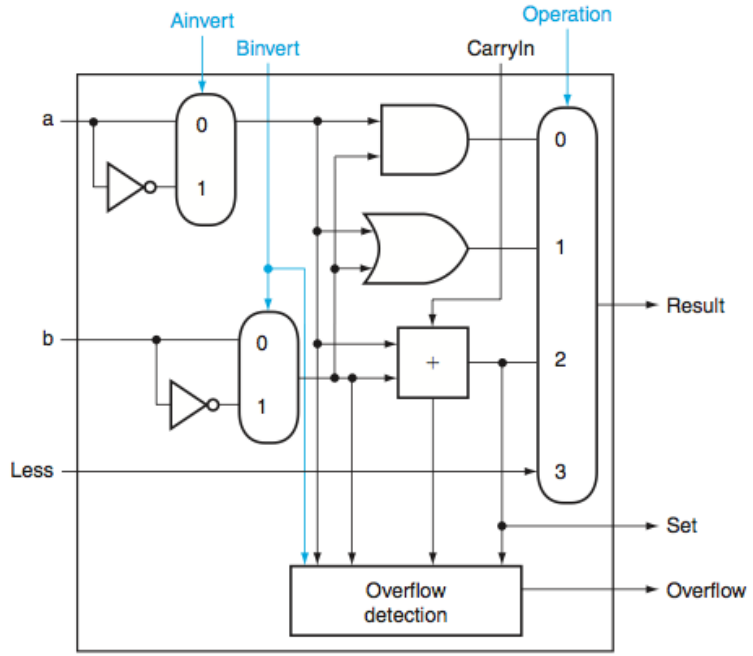

```
ld t0, 8(t1)
```

ALU Control Unit

- Combinational logic (the main control unit) derives 2-bit ALUOp signal from opcode
- ALU Control Unit takes ALUOp and the instruction's funct7 and funct3 fields as inputs and derives a 4-bit ALU control signal

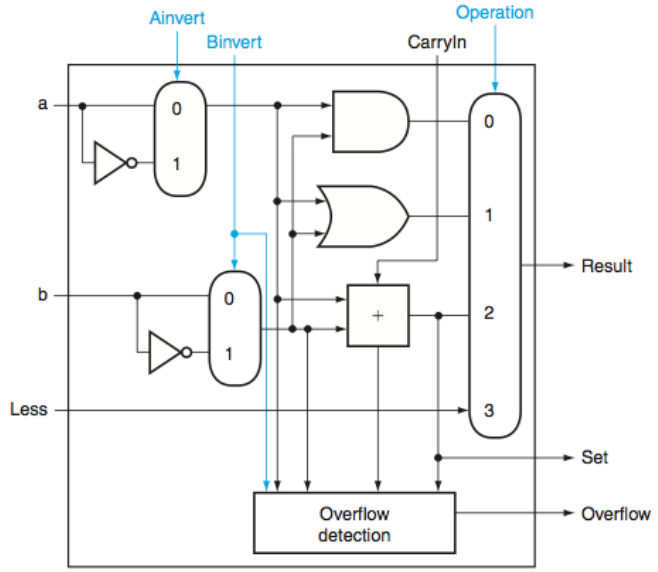
opcode	ALUOp	Operation	ALU function
ld	00	load word	add
sd	00	store word	add
beq	01	branch equal	subtract
R-type	10	arithmetic/logic	depends on funct7/funct3





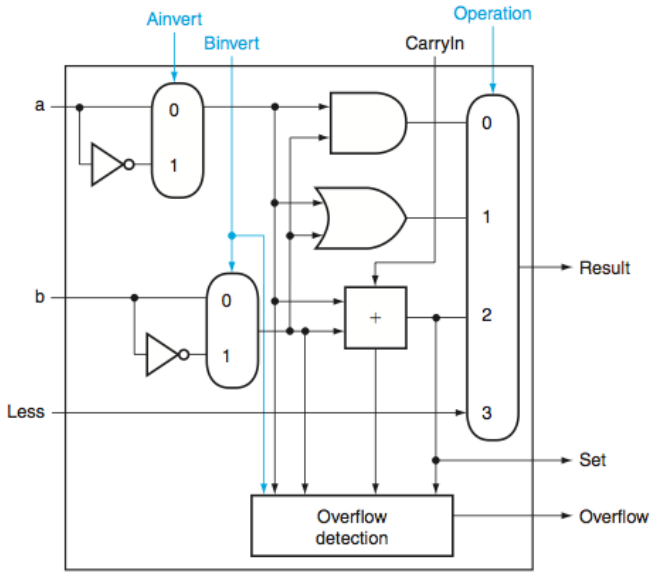
ALU Control signal

- ALU used for
  - Load/Store: op = add
  - Branch: op = subtract
  - R-type: op depends on funct7/funct3 fields



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 Unit

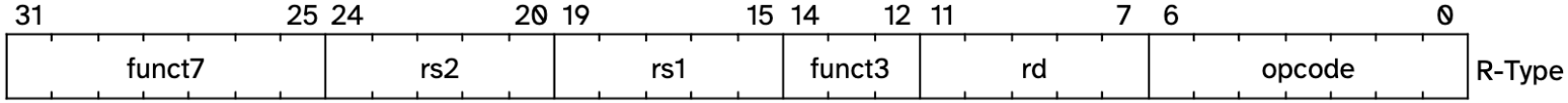


Takes as input 2-bit ALUop (derived from opcode) and 10 bits of funct fields; outputs 4 bits

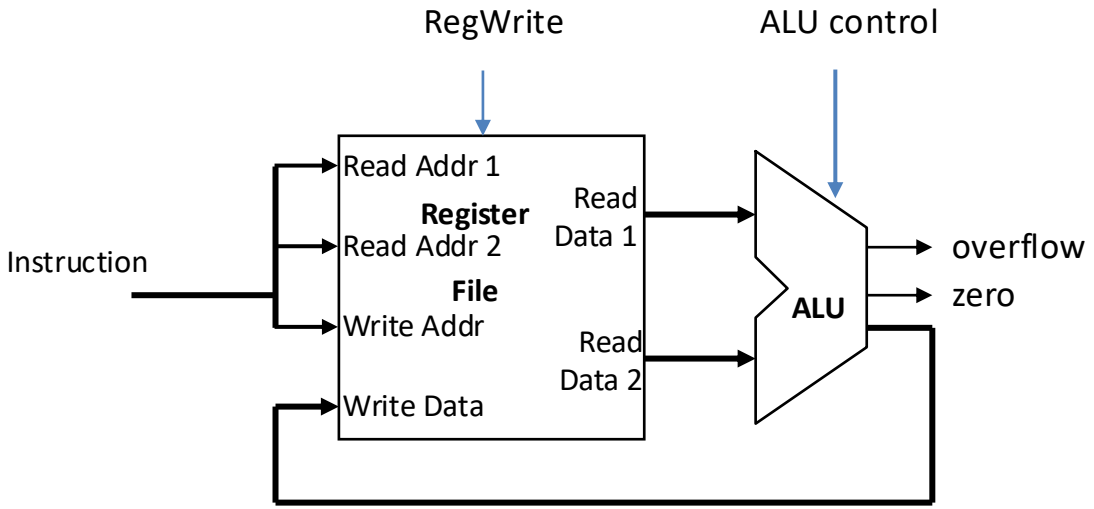
Instruction	ALUOp	funct7, funct3	ALU function	Ainvert	Binvert	ALU operation
load dword	00 (add)	XXXXXXX, XXX	add	0	0	10 (add)
store dword	00 (add)	XXXXXXX, XXX	add	0	0	10 (add)
branch equal	01 (subtract)	XXXXXXX, XXX	subtract	0	1	10 (add)
add	10 (r-type)	0000000, 000	add	0	0	10 (add)
subtract		0100000, 000	subtract	0	1	10 (add)
AND		0000000, 111	AND	0	0	00 (and)
OR		0000000, 110	OR	0	0	01 (or)
set-on-less-than		0000000, 010	set-on-less-than	0	1	11 (less)

Executing R Format Operations

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



  - perform operation (specified by funct7/funct3) on values in rs1 and rs2
  - store the result back into the Register File (into location rd)

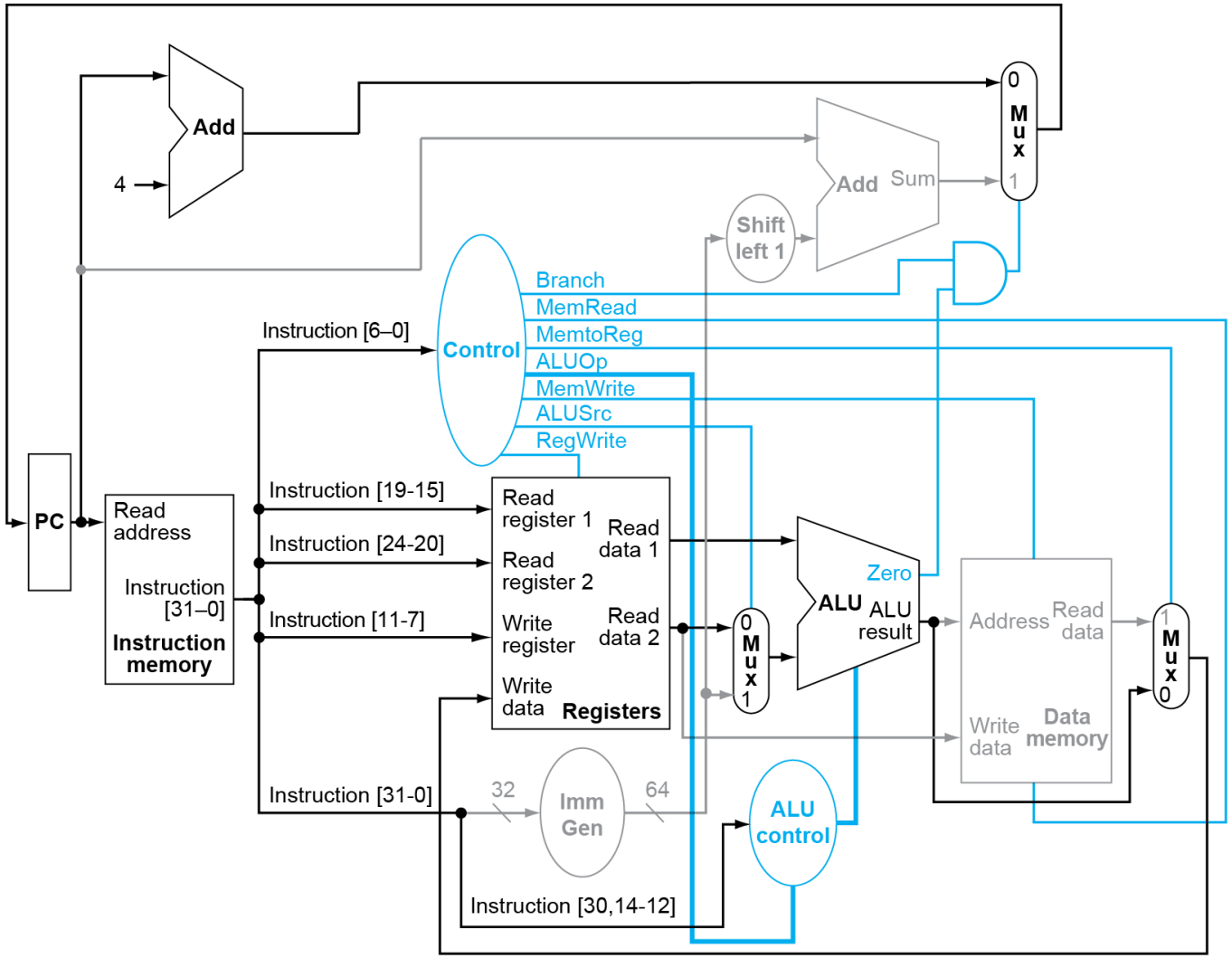


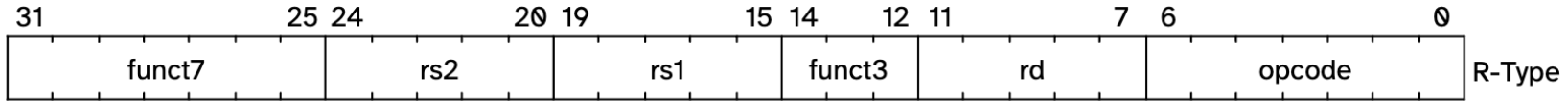
Note that Register File is not written every cycle (e.g., **`sd`**), so we need an explicit write control signal for the Register File

R-Type Instruction

ALUOp = 00 (add)
ALUOp = 01 (subtract)
ALUOp = 10 (R-type)

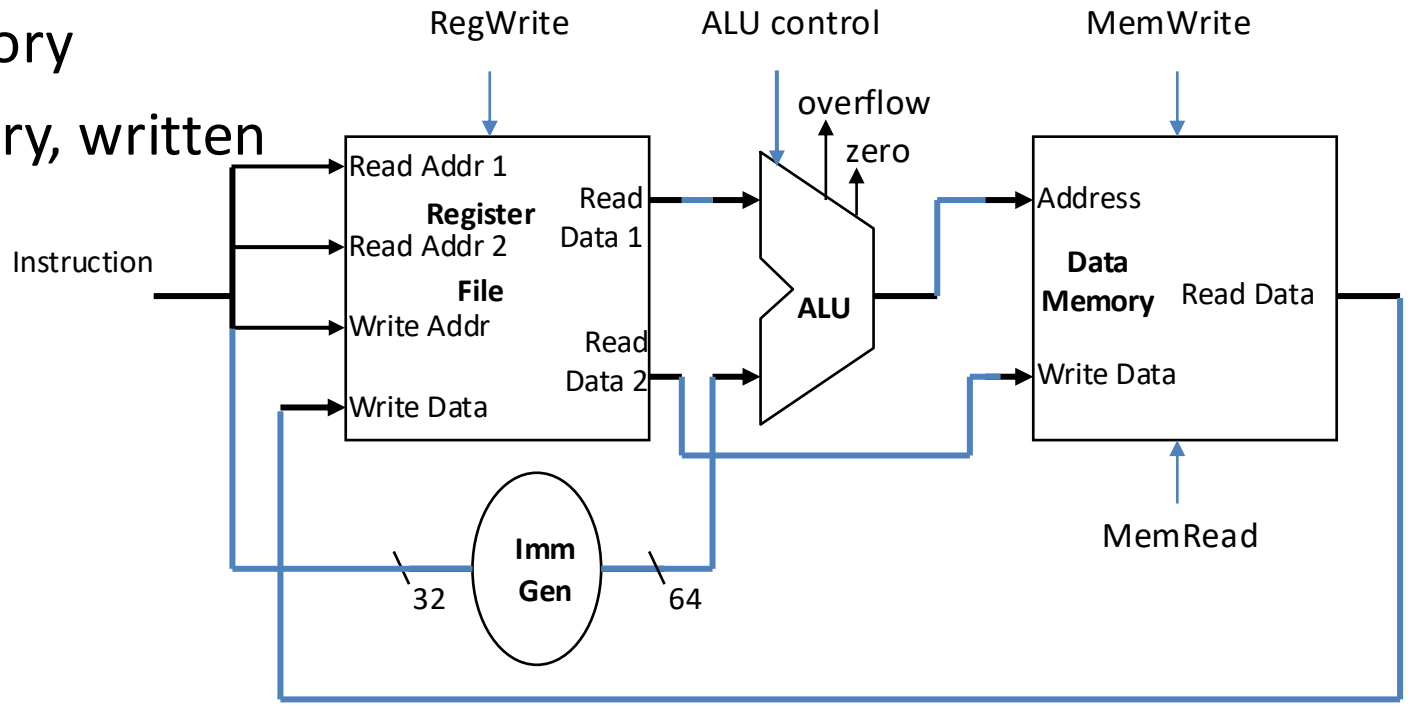
Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	

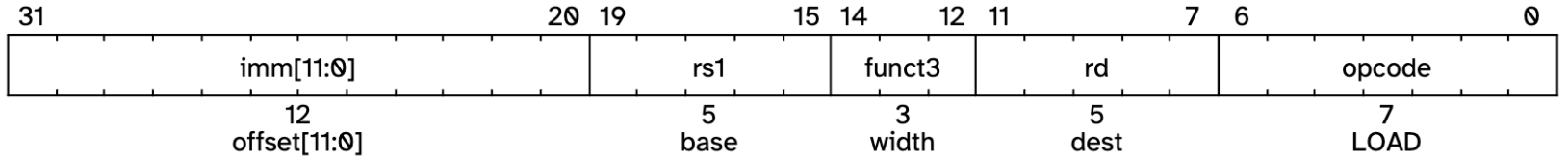


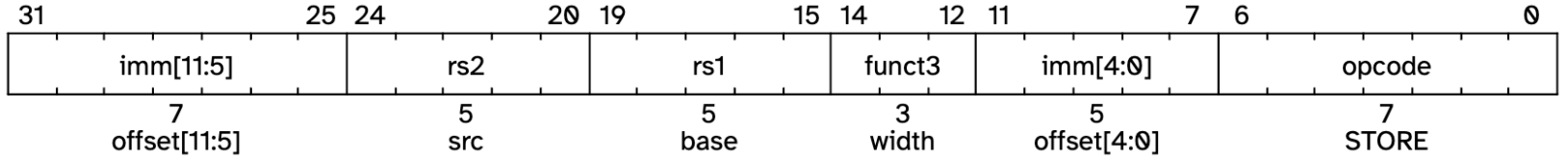


Executing Load and Store Operations

- compute memory address by adding base register to 12-bit signed-extended offset field
- store value written to the Data Memory
- load value read from the Data Memory, written to the Register File



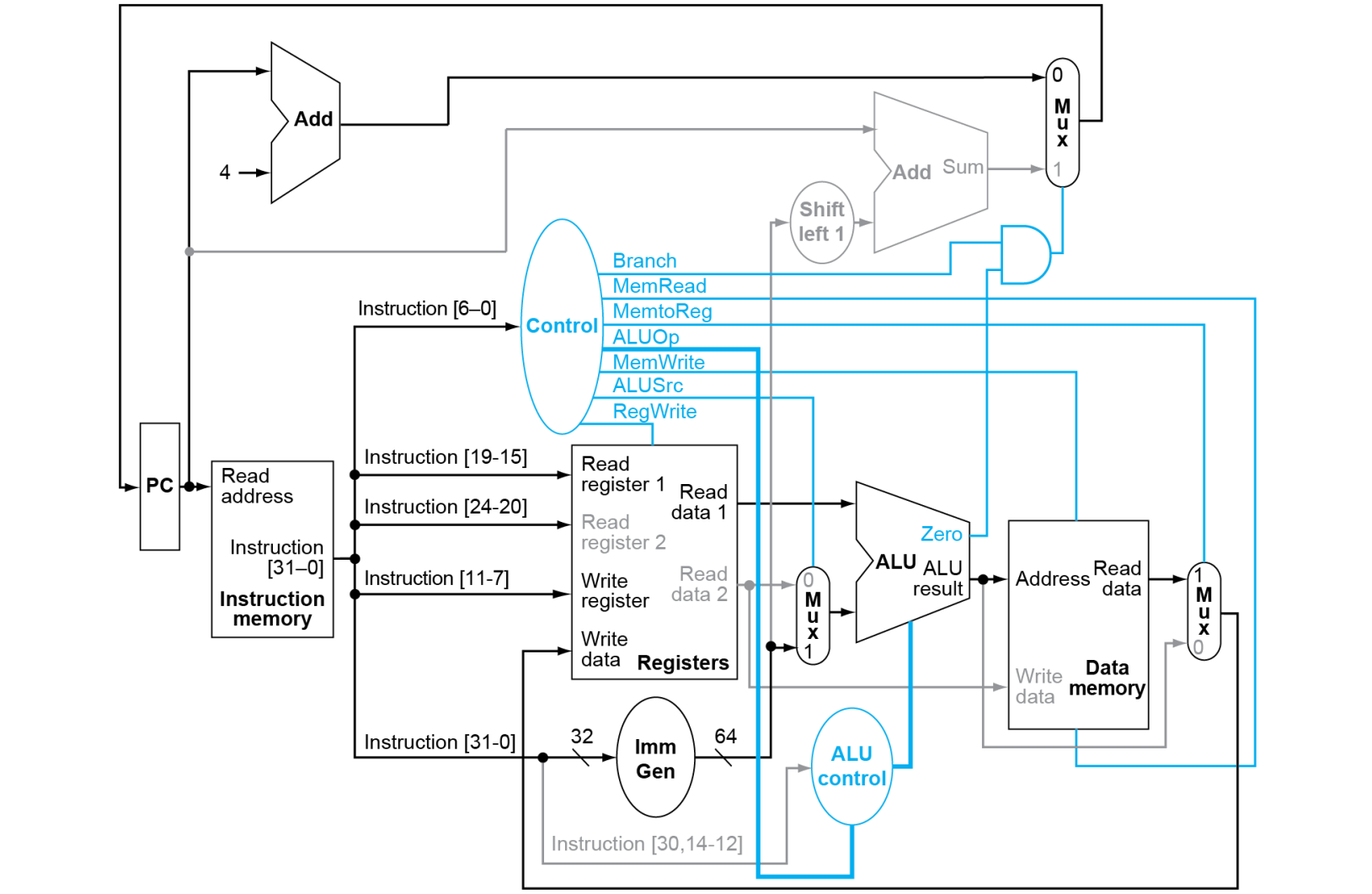




Load Instruction

ALUOp = 00 (add)
ALUOp = 01 (subtract)
ALUOp = 10 (R-type)

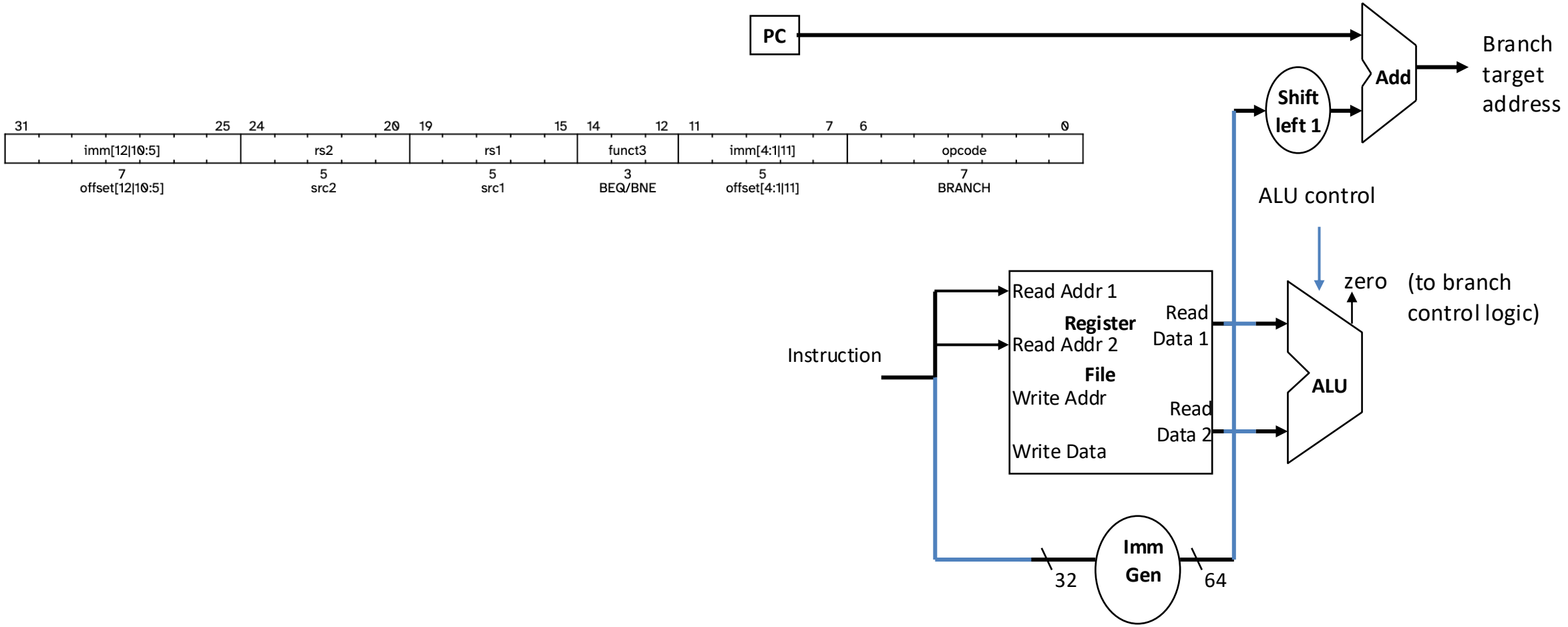
Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	



Executing Branch Operations

Branch operations involve

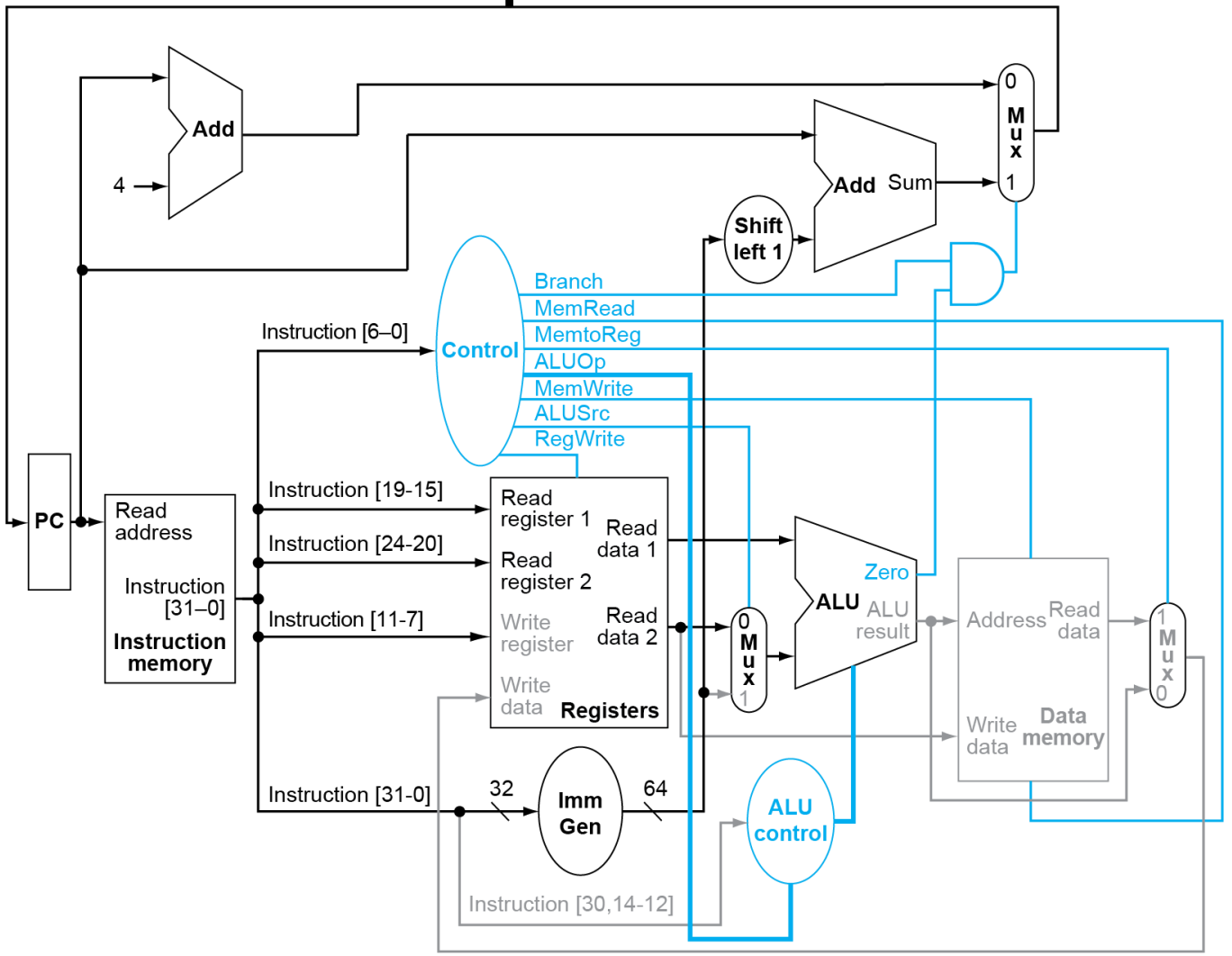
- compare the operands read from the Register File during decode for equality (`zero` ALU output)
- compute the branch target address by adding the PC to the 12-bit sign-extended and shifted offset field in the instruction

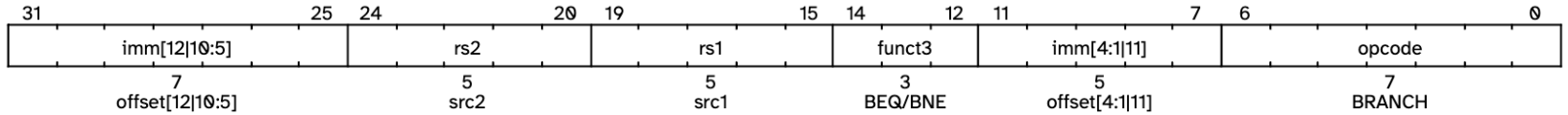


Branch-on-Equal Instruction

ALUOp = 00 (add)
ALUOp = 01 (subtract)
ALUOp = 10 (R-type)

Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	



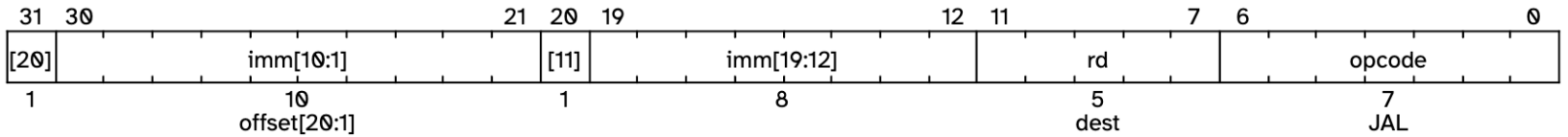


Control Truth Table

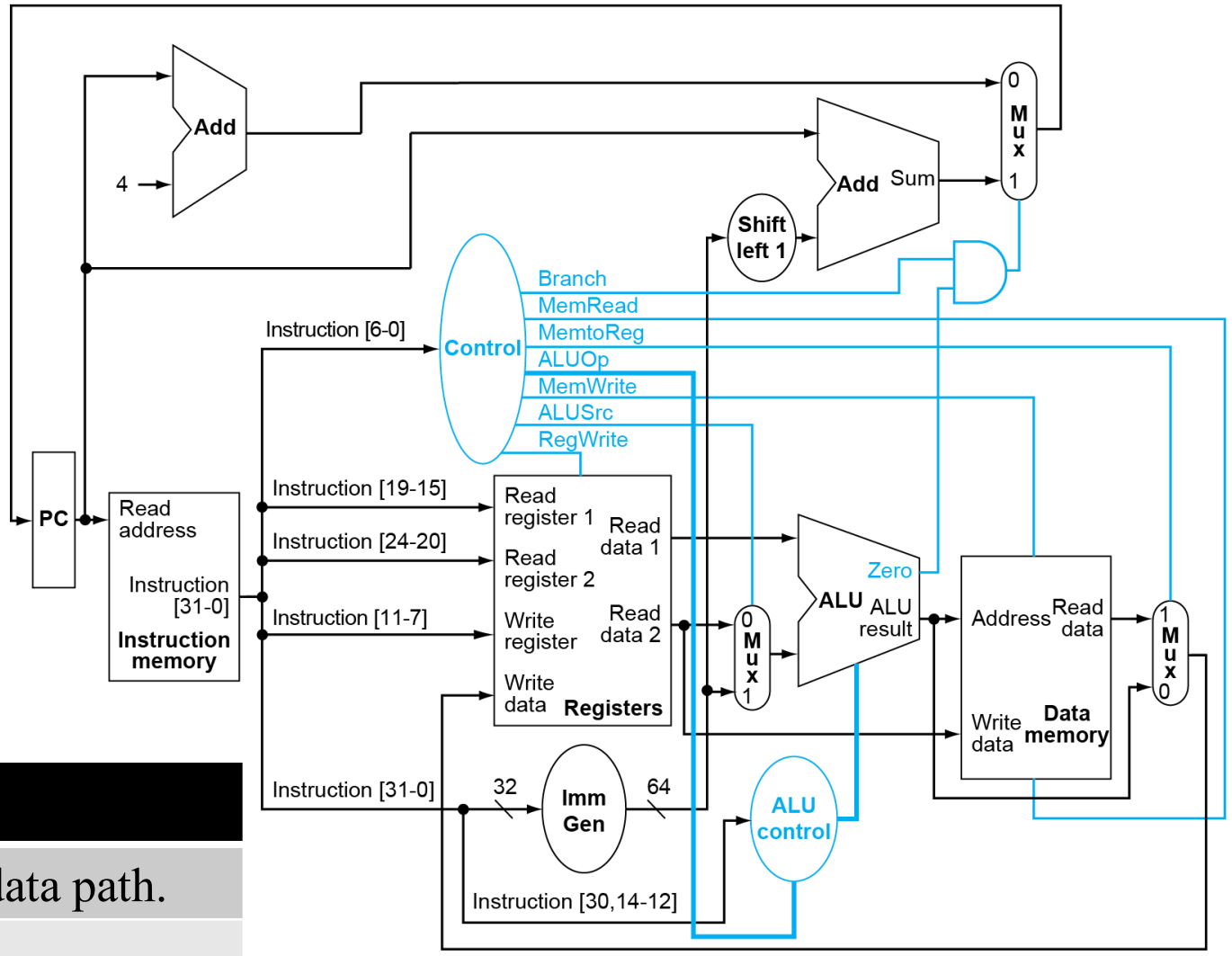
	R-type	ld	sd	beq
Signals\Opcode	0110011	0000011	0100011	1100011
Branch	0	0	0	1
MemRead	0	1	0	0
MemtoReg	0	1	X	X
ALUOp	10	00	00	01
MemWrite	0	0	1	0
ALUSrc	0	1	1	0
RegWrite	1	1	0	0

ALUOp = 00 (add)
ALUOp = 01 (subtract)
ALUOp = 10 (R-type, use funct7/funct3 to determine ALUCtrl signals)

Implementing Jump and Link (JAL)



- jal uses 20 bit, sign-extended and shifted-by-1 offset
- Computation:
  offset = SignExtend(immediate) << 1
  pc = pc + offset

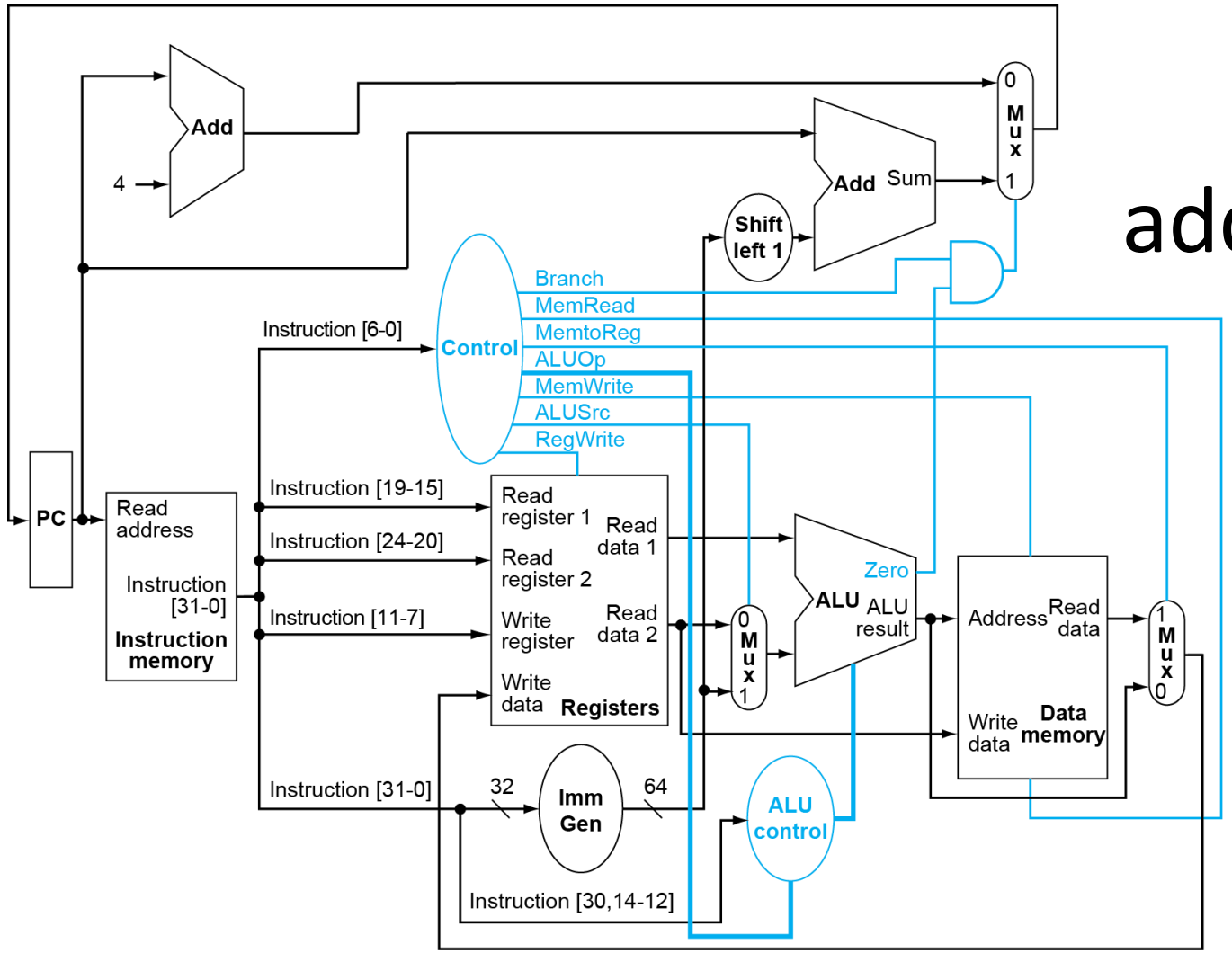


Do we need to modify our design to do jump and link?

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

addi x10, x11, 6

Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	



ALUOp = 00 (add)
ALUOp = 01 (subtract)
ALUOp = 10 (R-type)

Reading

- Next lecture: Pipelining