CS / EE 320
Computer Organization and Assembly Language
Spring 2025
Lecture 14

Shahid Masud

Topics: Building a Single Cycle MIPS CPU

Important Announcement

- **Midterm Exam next week**

Topics

- Review Basic ALU Design (see Lecture 13 slides)
- Scale 1 Bit ALU to 32 Bits
- Building blocks of MIPS CPU
- Architecture of simple instructions in Single Cycle MIPS
  - J jump instruction
  - ADD instruction
  - ADDI instruction
  - BEQ instruction
  - LW load word instruction
  - SW store word instruction

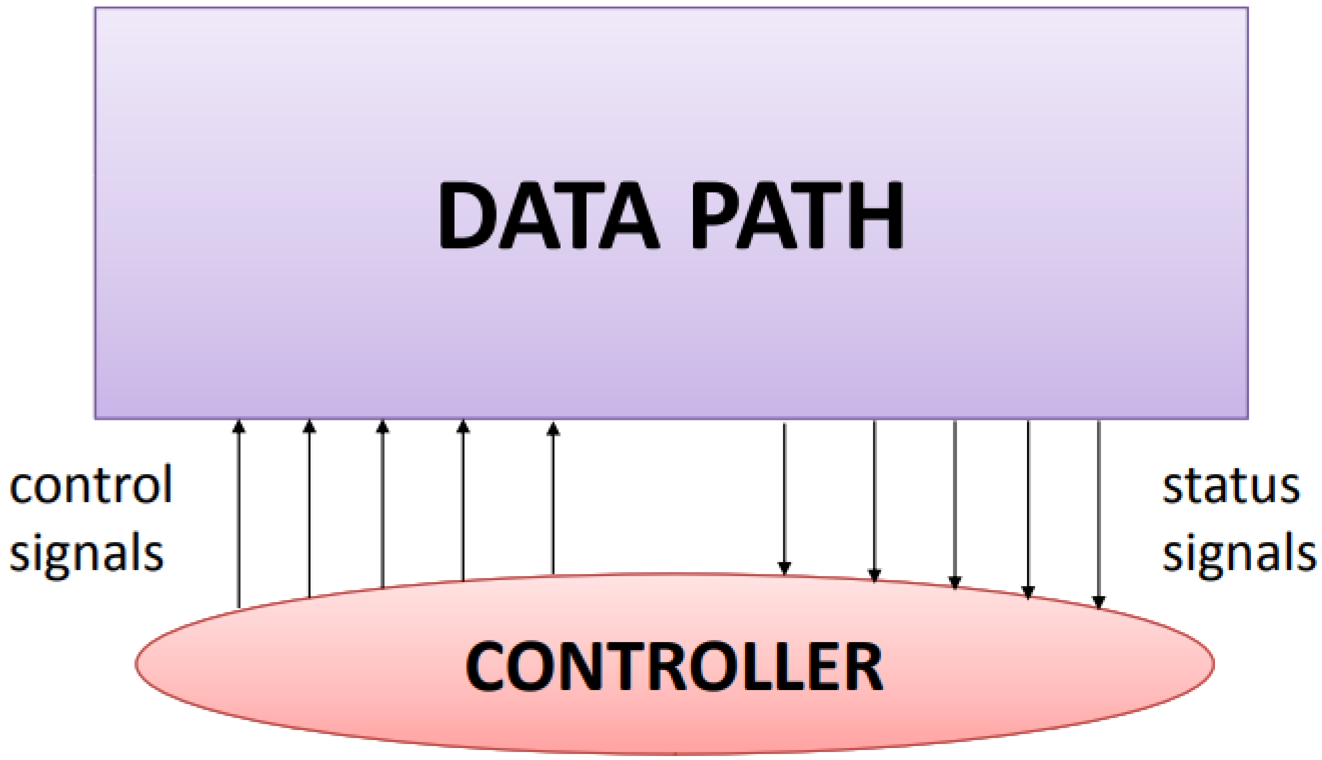
As much as possible, before the quiz

- Putting together a single cycle Simple MIPS CPU Architecture
- **Quiz 3 today**
- **Midterm Exam Next Week**

Instruction Execution

- Instruction **Fetch** → Instruction **Decode** → Instruction **Execute**
- PC $\rightarrow$ instruction memory, fetch instruction
- Register numbers $\rightarrow$ register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
- Access data memory for load/store
- PC $\leftarrow$ target address or PC + 4 after current **Fetch**

CPU consists of Data path and Control path



Building a Datapath

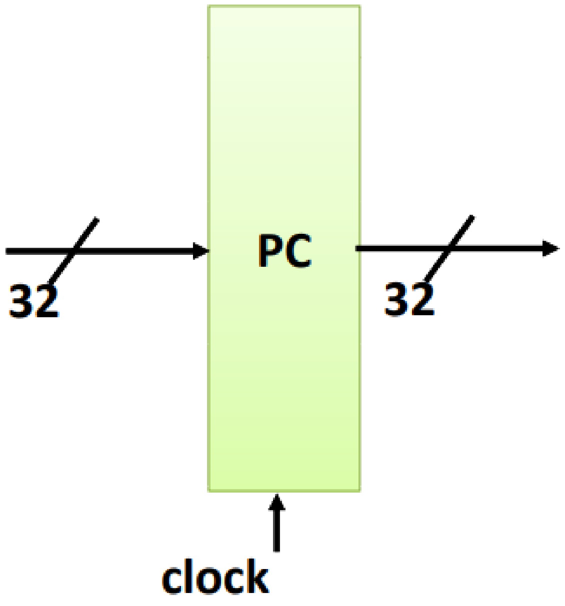
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the basic design
- We will start with a **Single Cycle MIPS Processor**

Components for Building MIPS CPU

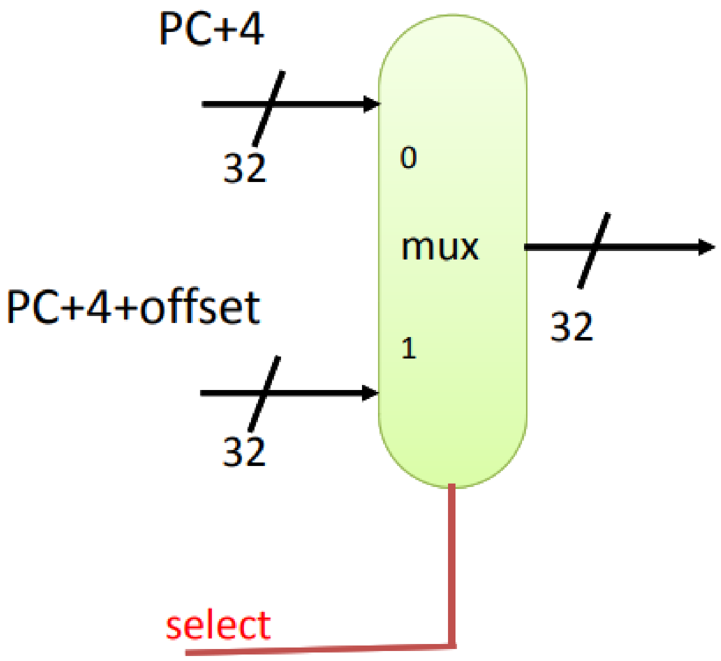
- Register
- Adder
- ALU
- Multiplexer
- Register file
- Program memory
- Data memory
- Bit manipulation components

MIPS Components - 1

MIPS Components - Register

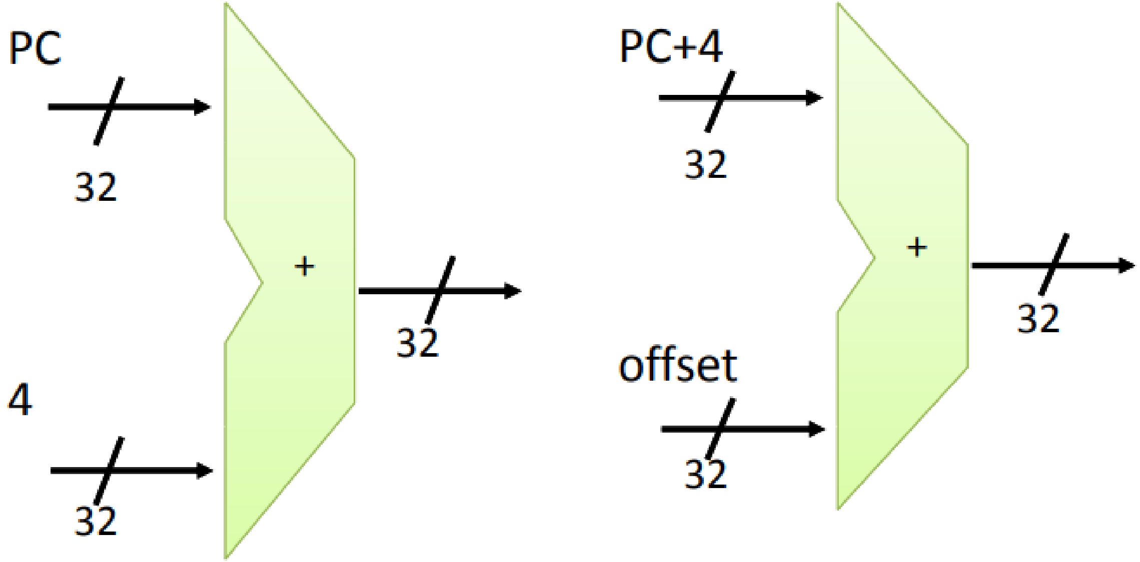


MIPS components - Multiplexers

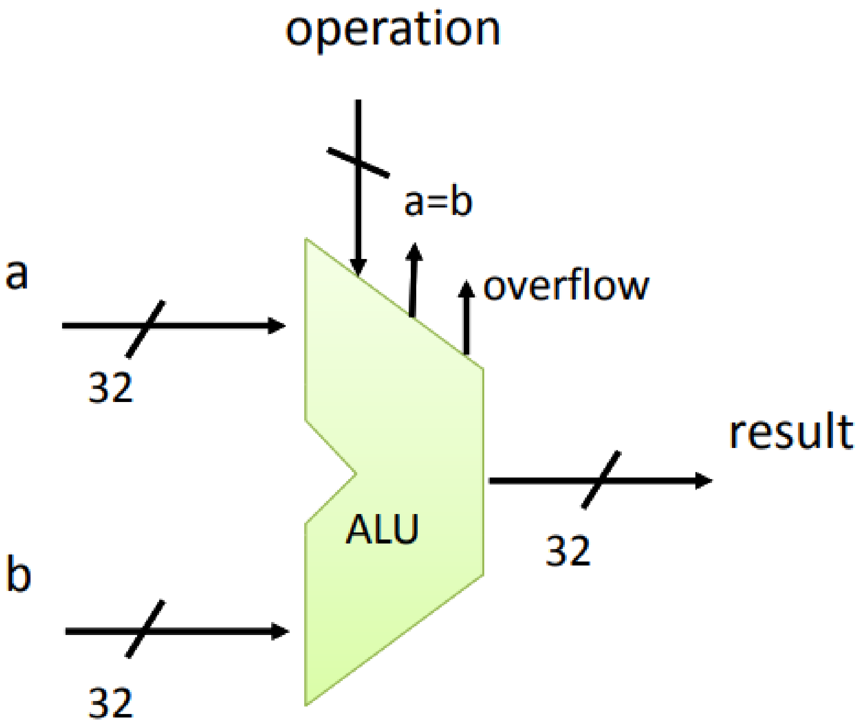


MIPS Components - 2

MIPS Components - Adder

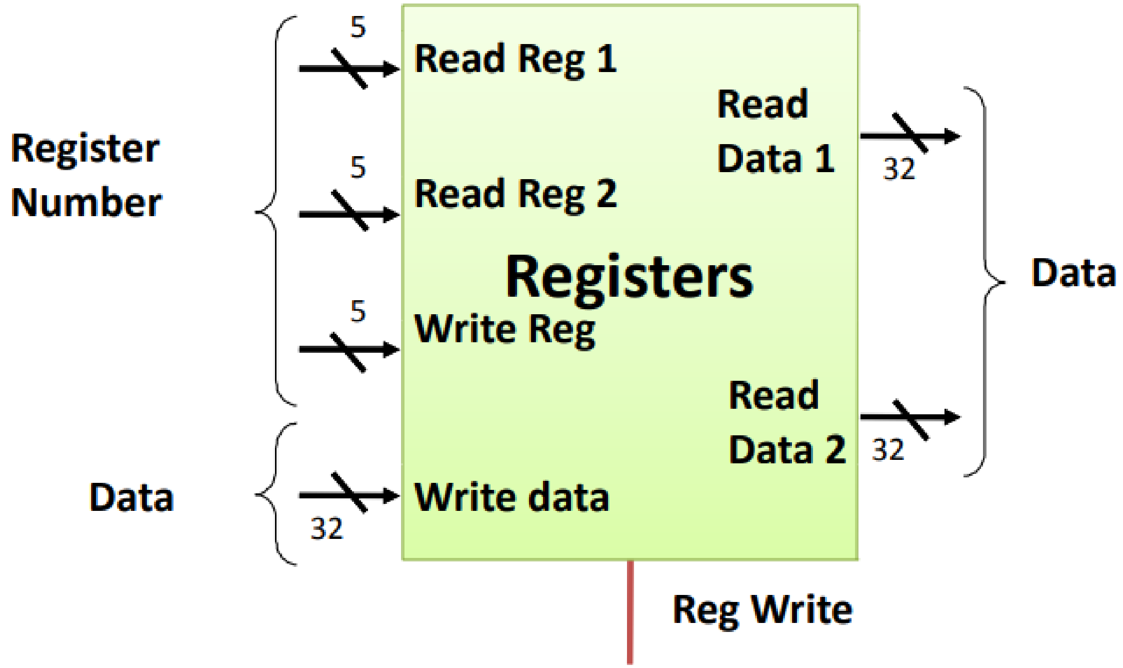


MIPS Components - ALU

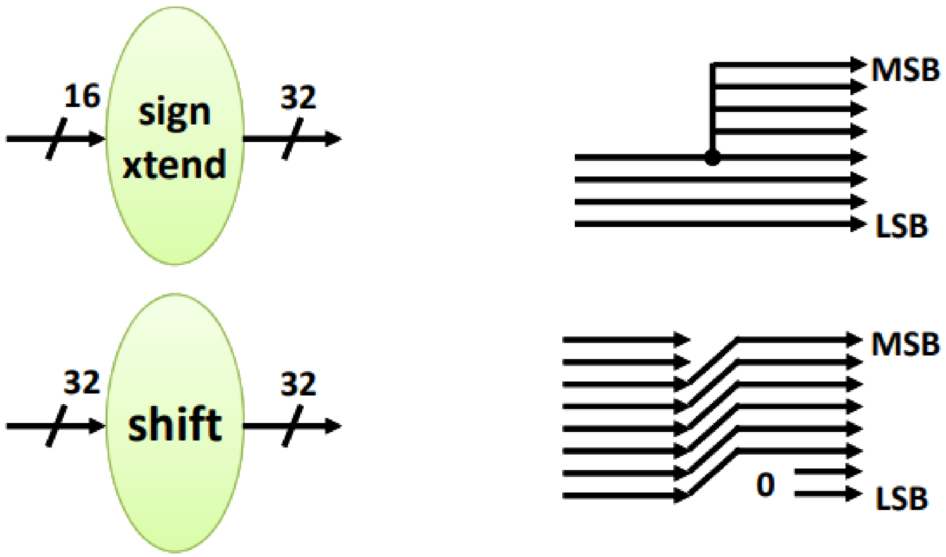


MIPS Components - 3

MIPS Components - register file

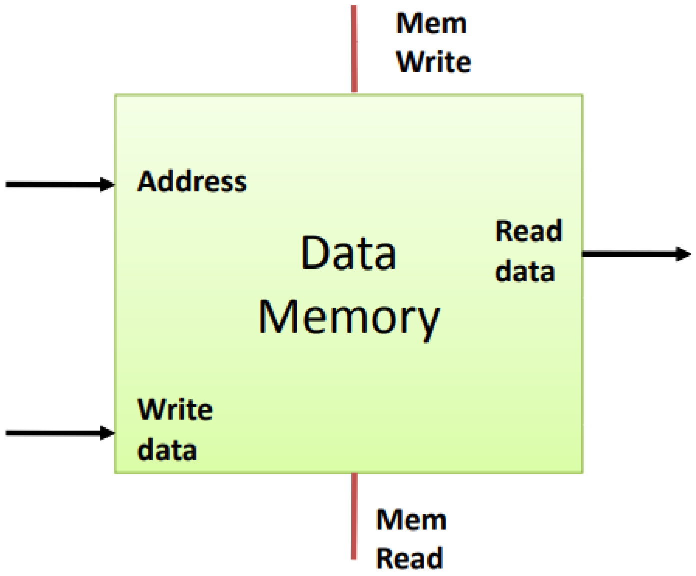


MIPS Components - Bit manipulation circuits

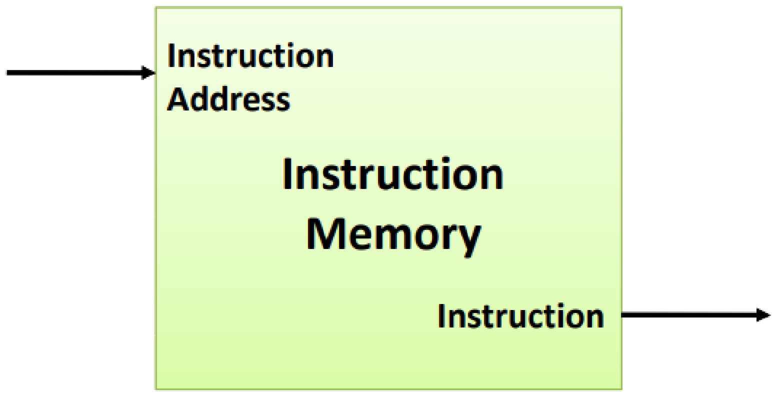


MIPS Components - 4

MIPS Components -Data memory



MIPS Components: Program memory



Build MIPS Data path – Step by Step

Build the datapath step by step as follows

- Start with R - class instructions
- Include other instructions one by one
- Identify control signals
- Interconnect datapath and controller

Two Components for R format CPU

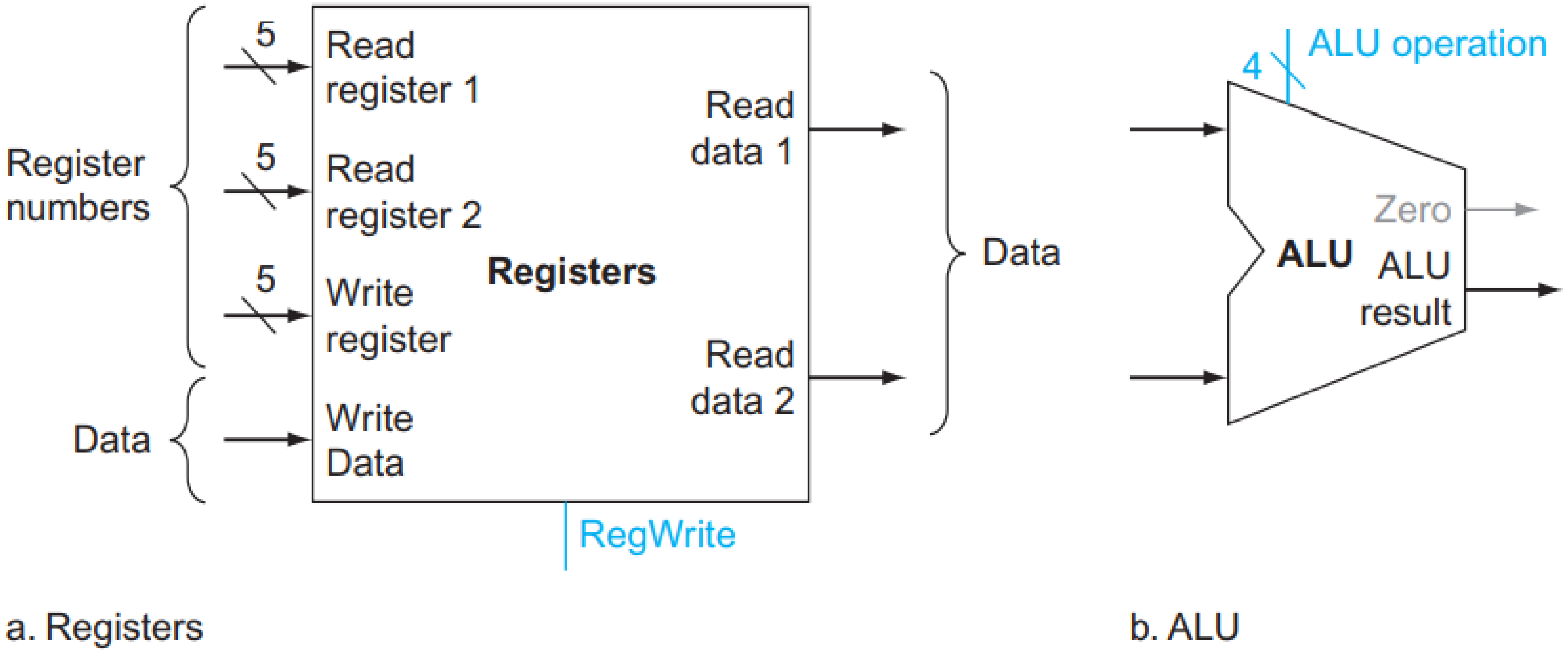
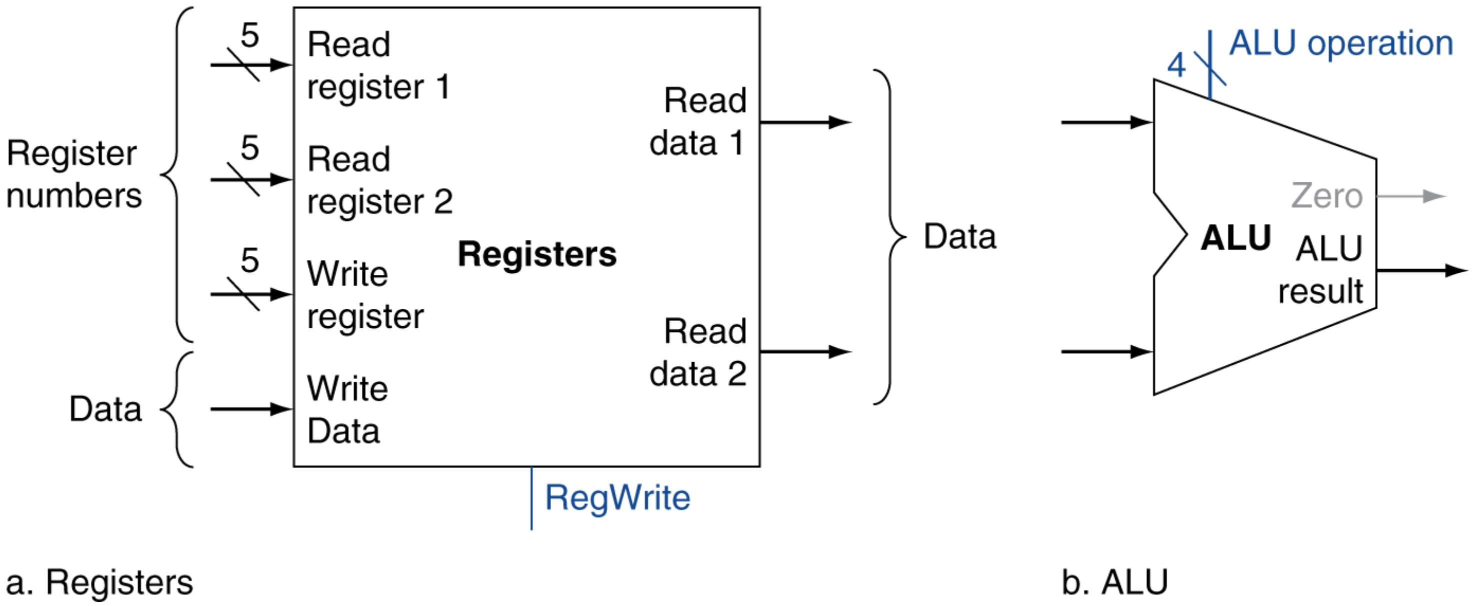


FIGURE 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section B.8 of Appendix B. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to

R-Format Instructions

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



MIPS Subset of R format to start implementation

- Arithmetic - logic instructions
  - **add, sub, and, or, slt**
- Memory reference instructions
  - **lw, sw**
- Control flow instructions
  - **beq, j**

Develop Data path for add, sub, and, or, slt

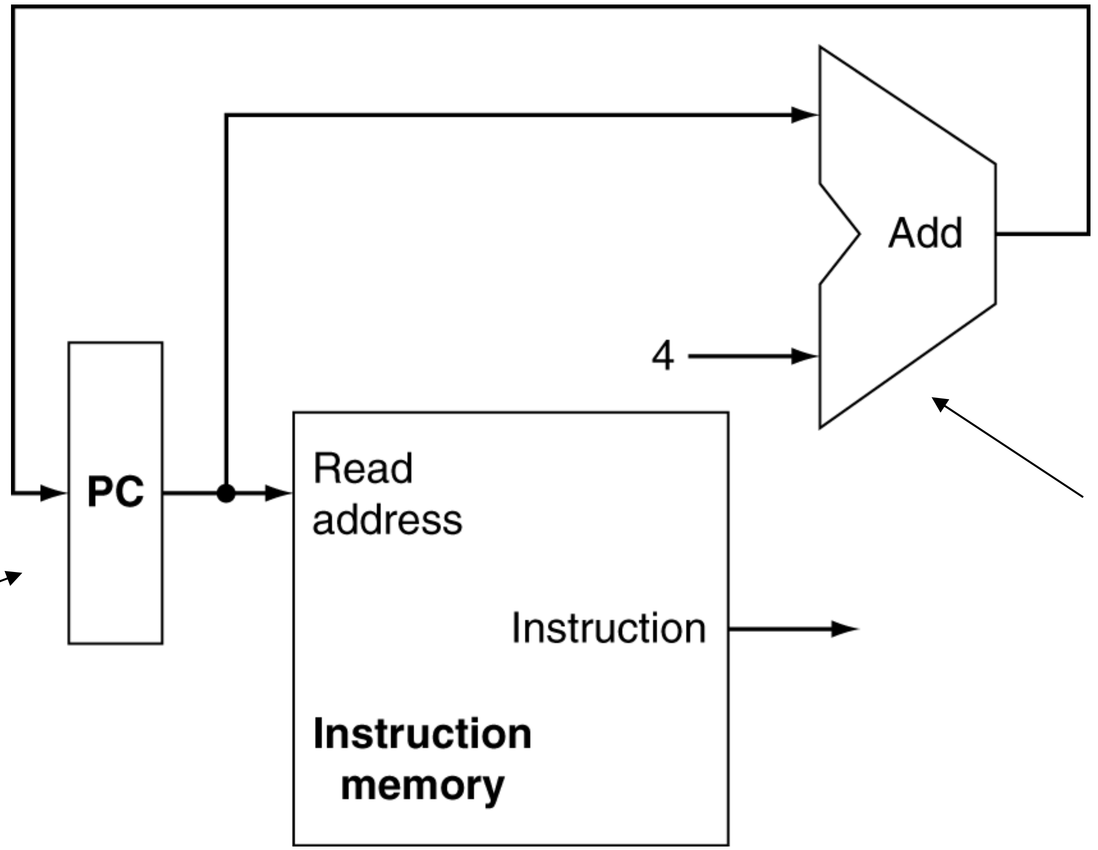
- fetch instruction
- address the register file
- pass operands to ALU
- pass result to register file
- increment PC

actions required

Format: add $t0, $s1, $s2

op	rs	rt	rd	shamt	funct
000000	10001	10010	01000	00000	100000

Instruction Fetch – PC Register operation

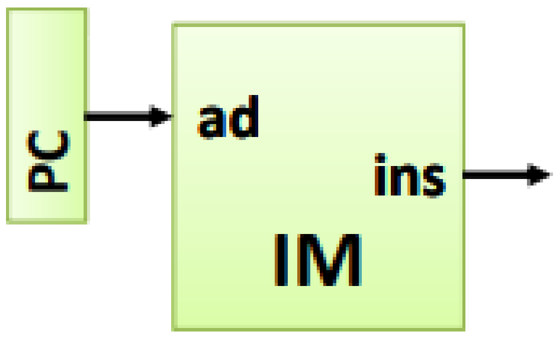


32-bit register

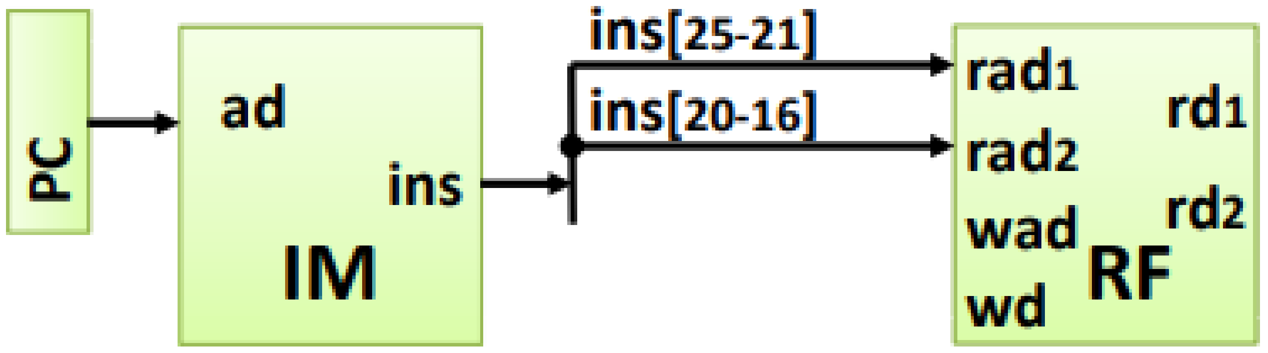
Increment by 4 for next instruction

Basic CPU Operations - 1

Fetching Instruction

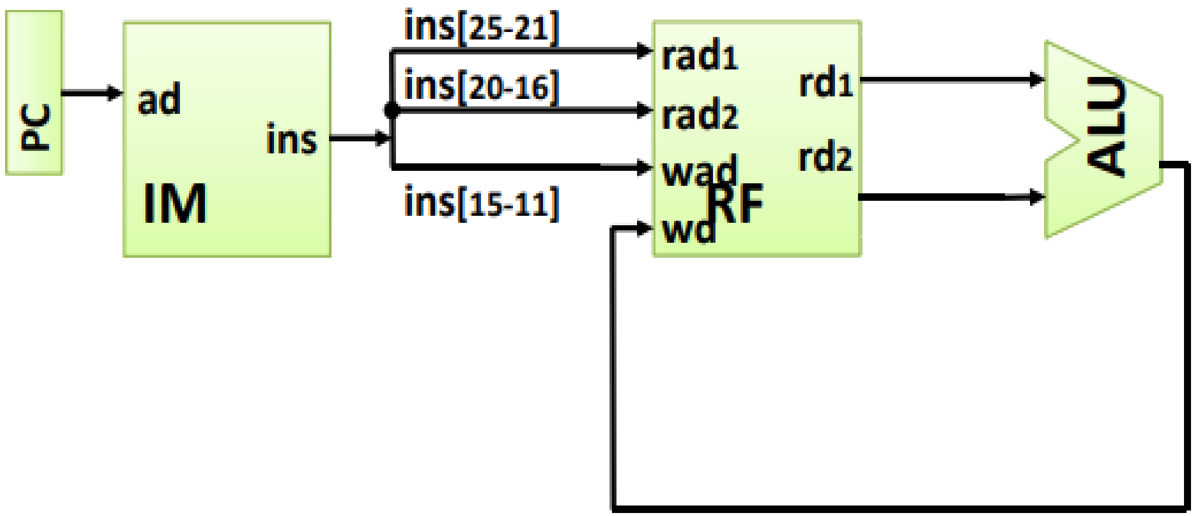


Addressing Register File

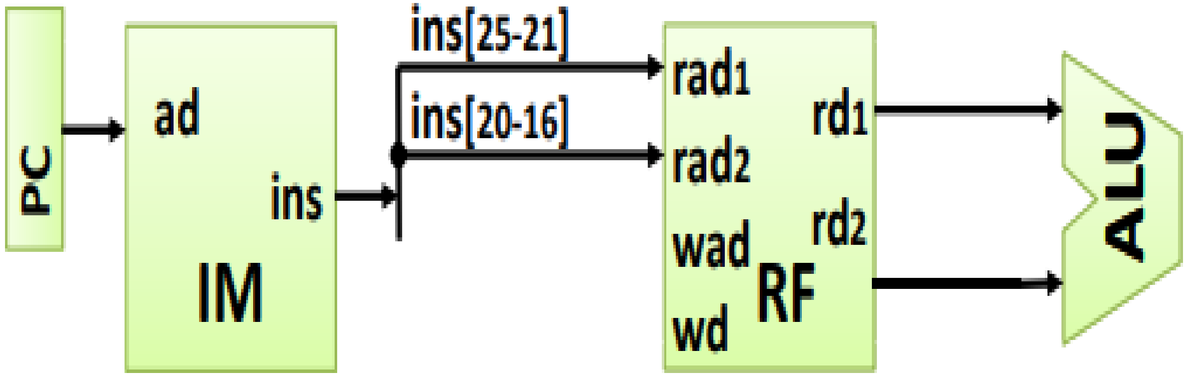


Basic CPU Operations - 2

Passing the Result to Register File

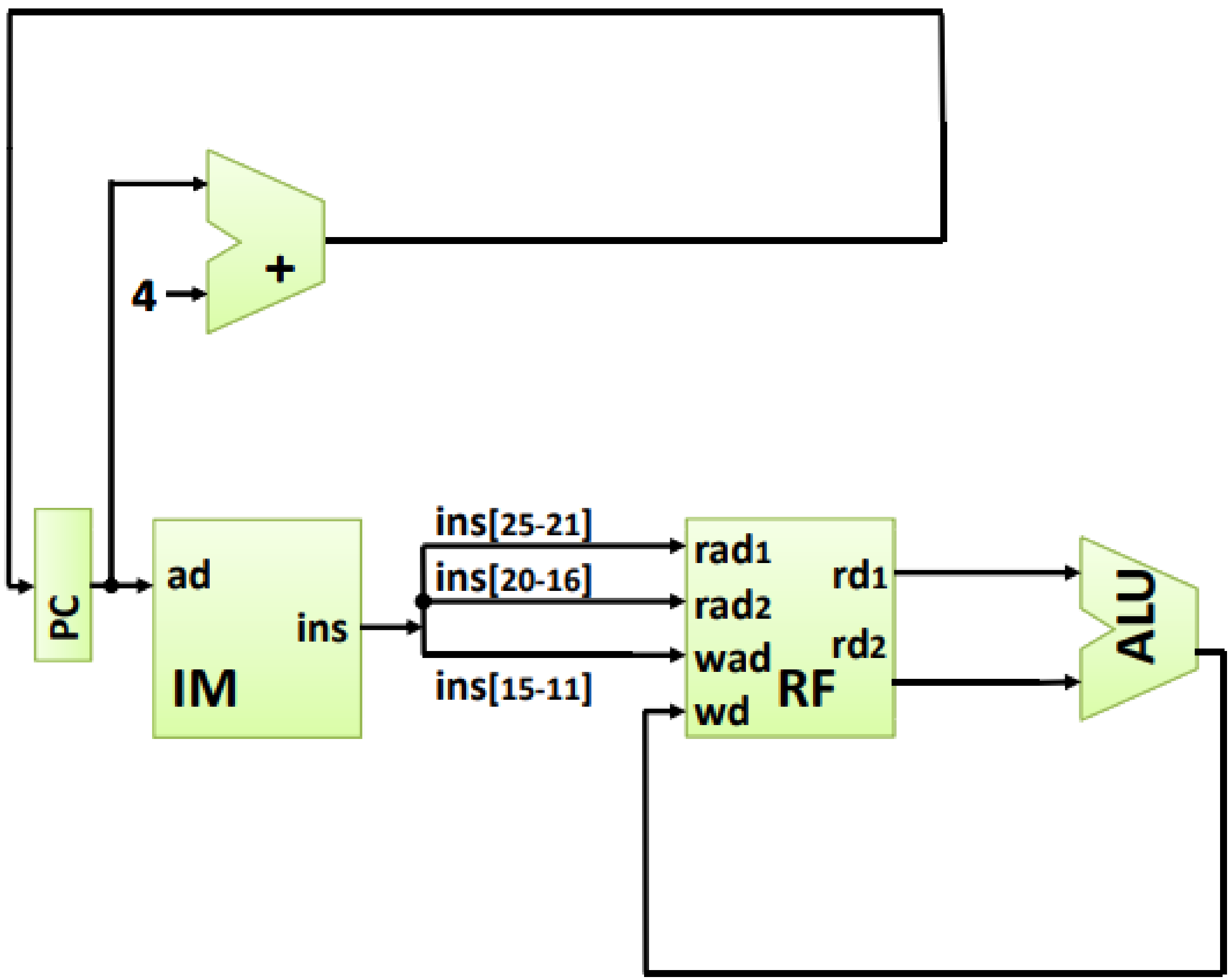


Passing Operands to ALU

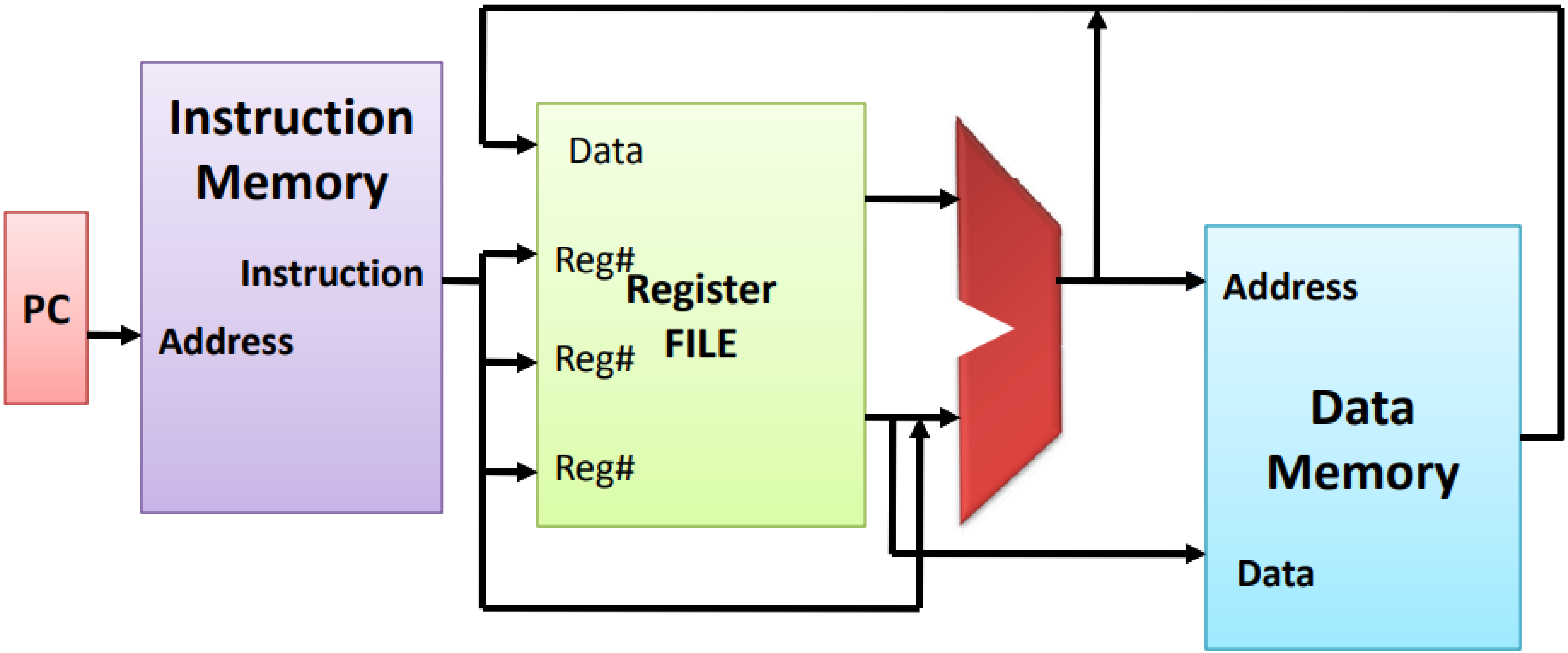


Basic CPU Operations - 3

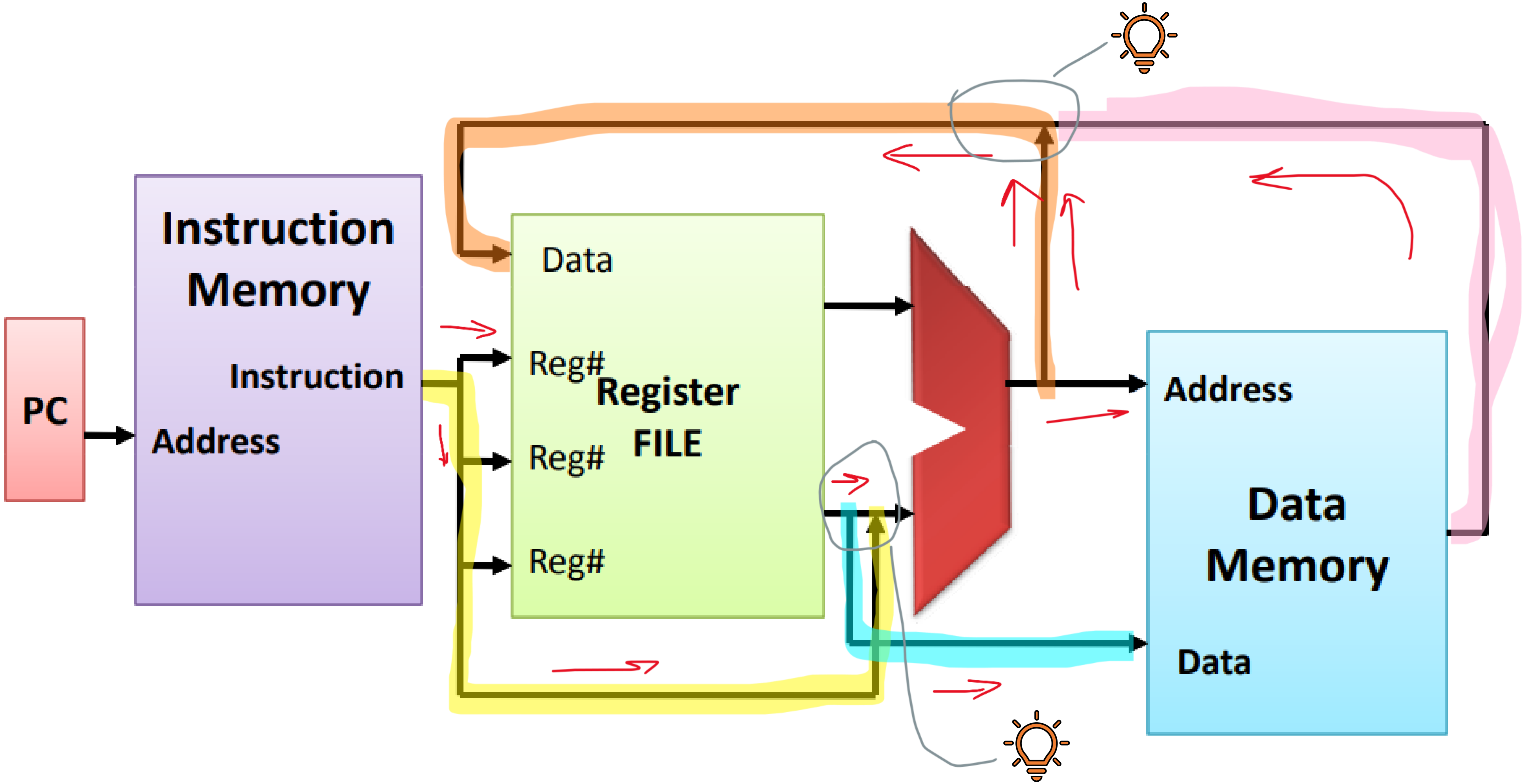
Incrementing PC



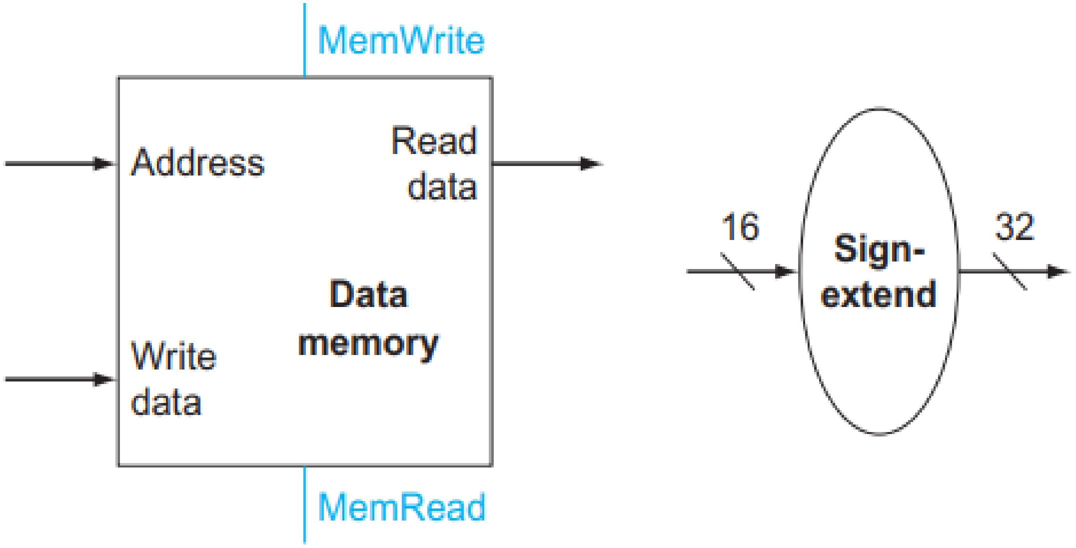
Overview of Simple CPU Design for R Format



Overview of Simple CPU Design for R Format



Two Components for **lw** and **sw** instructions



a. Data memory unit

b. Sign extension unit

FIGURE 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the sign extension unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in Chapter 5. The sign extension unit has a 16-bit input that is sign-extended into a 32-bit result appearing on the output (see Chapter 2). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See Section B.8 of Appendix B for further discussion of how real memory chips work.

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

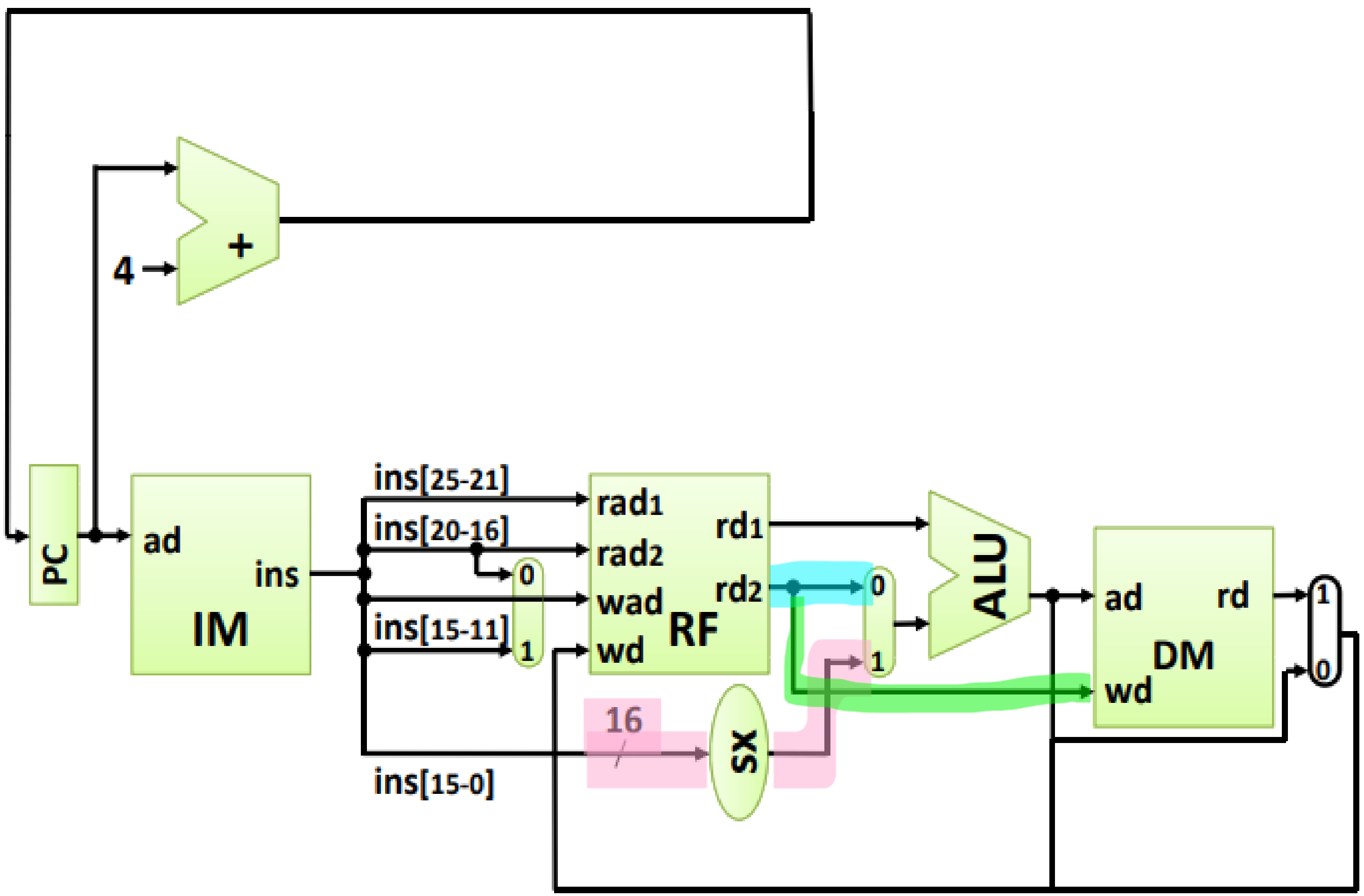
Implement the “lw” Instruction

Load and Store instructions

- format : I
- Example: lw $t0, 32($s2)

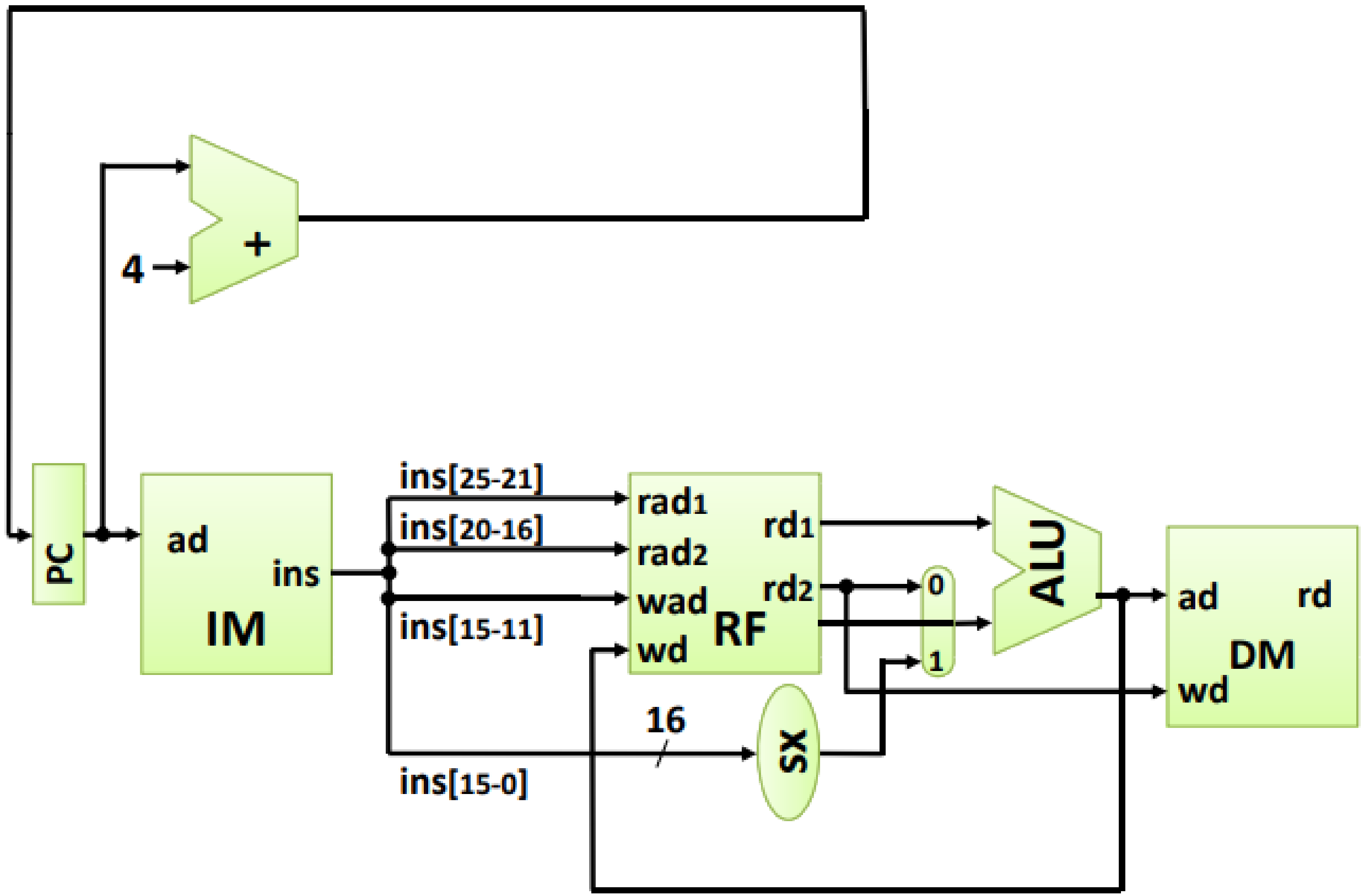
35	18	9	32
op	rs	rt	16 bit number

Adding “lw” Instruction



Implement the “sw” Instruction

Adding “sw” Instruction



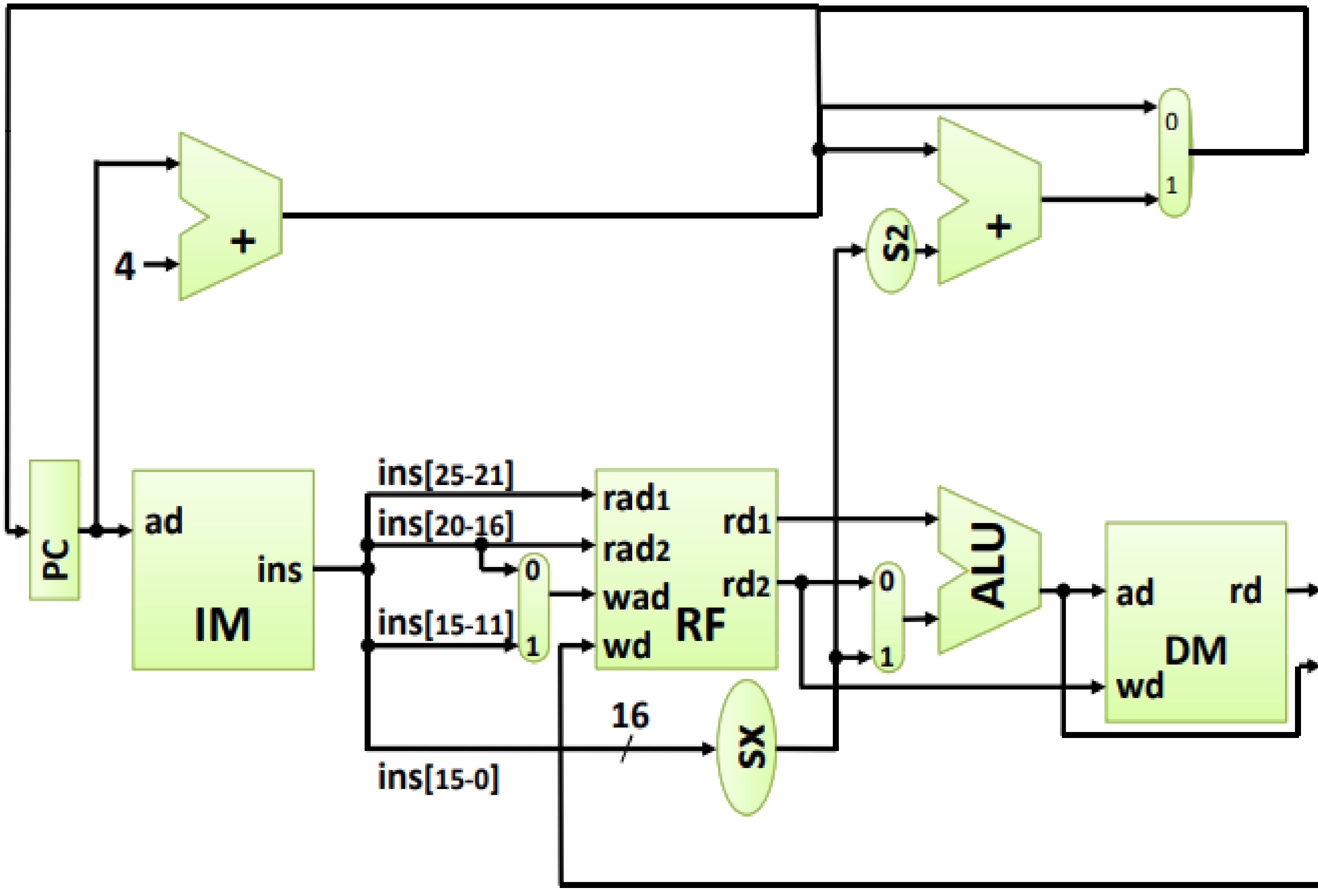
Implement "**beq**" Instruction

Format of beq instruction

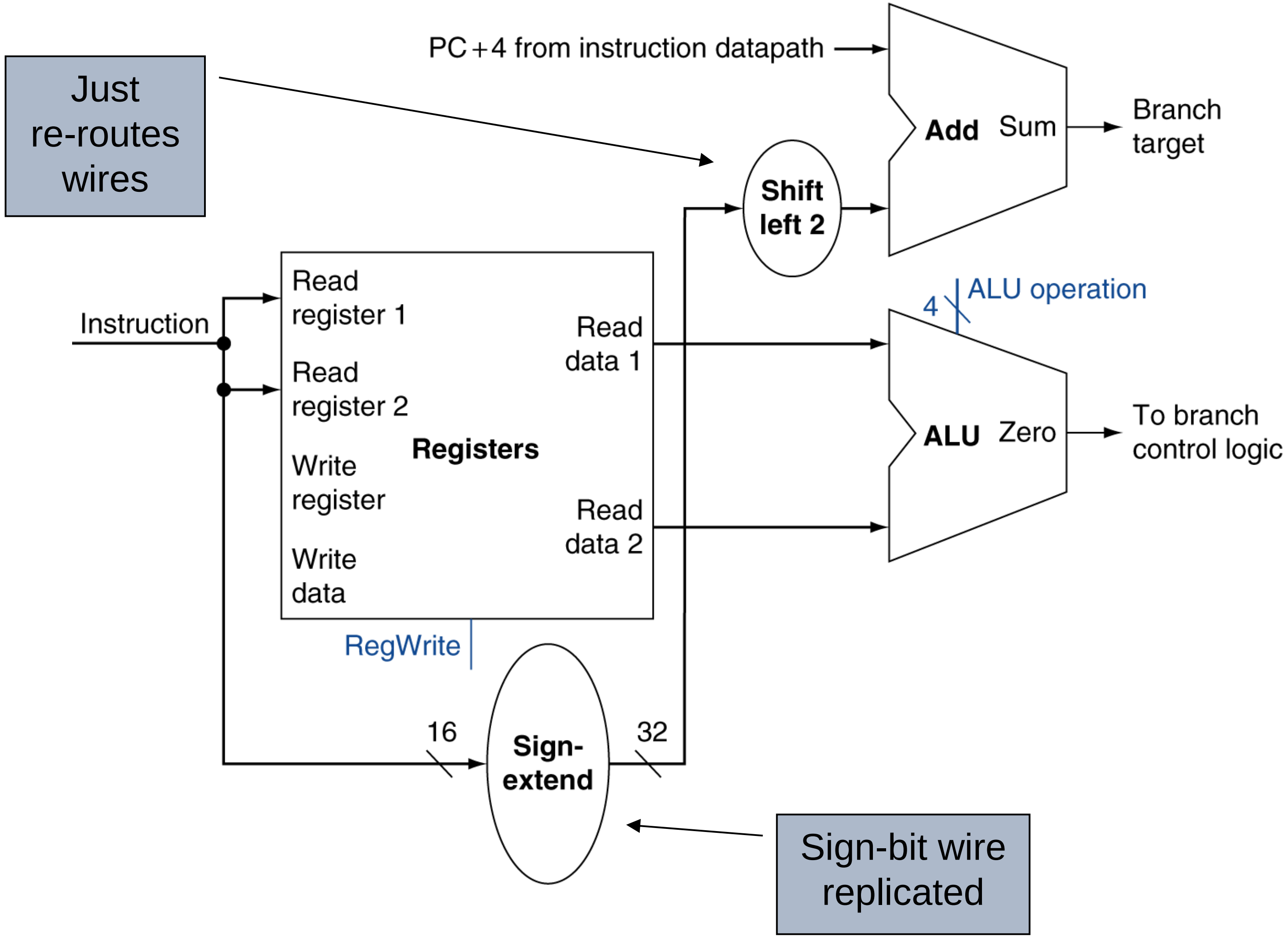
- beq I - format

op	rs	rt	16 bit number

Adding "beq" Instruction



Branch Instructions



Implementing **Jumps**

	31:26	25:0
Jump	2	address

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

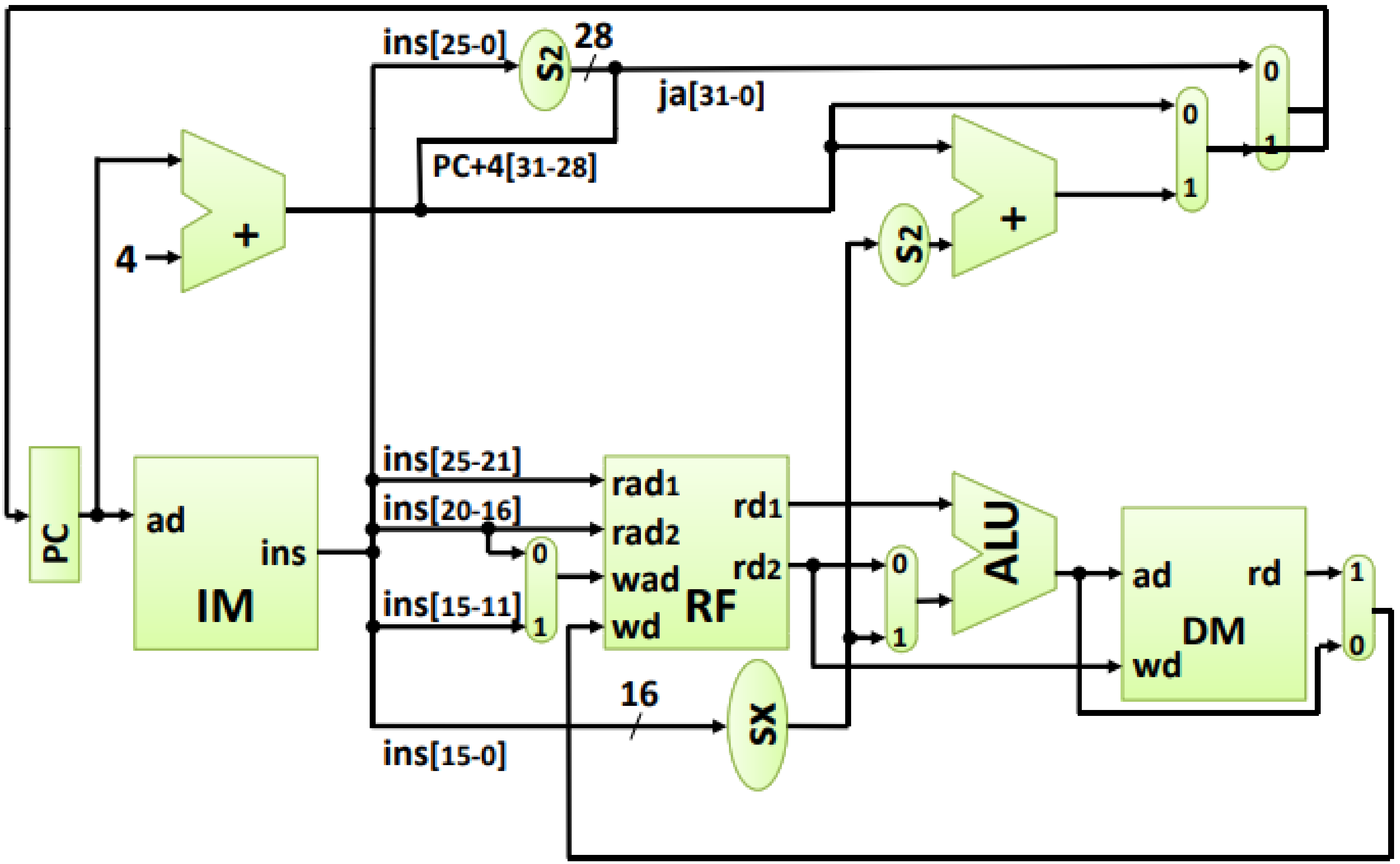
Implement Jump "j" instruction

Adding "j" Instruction

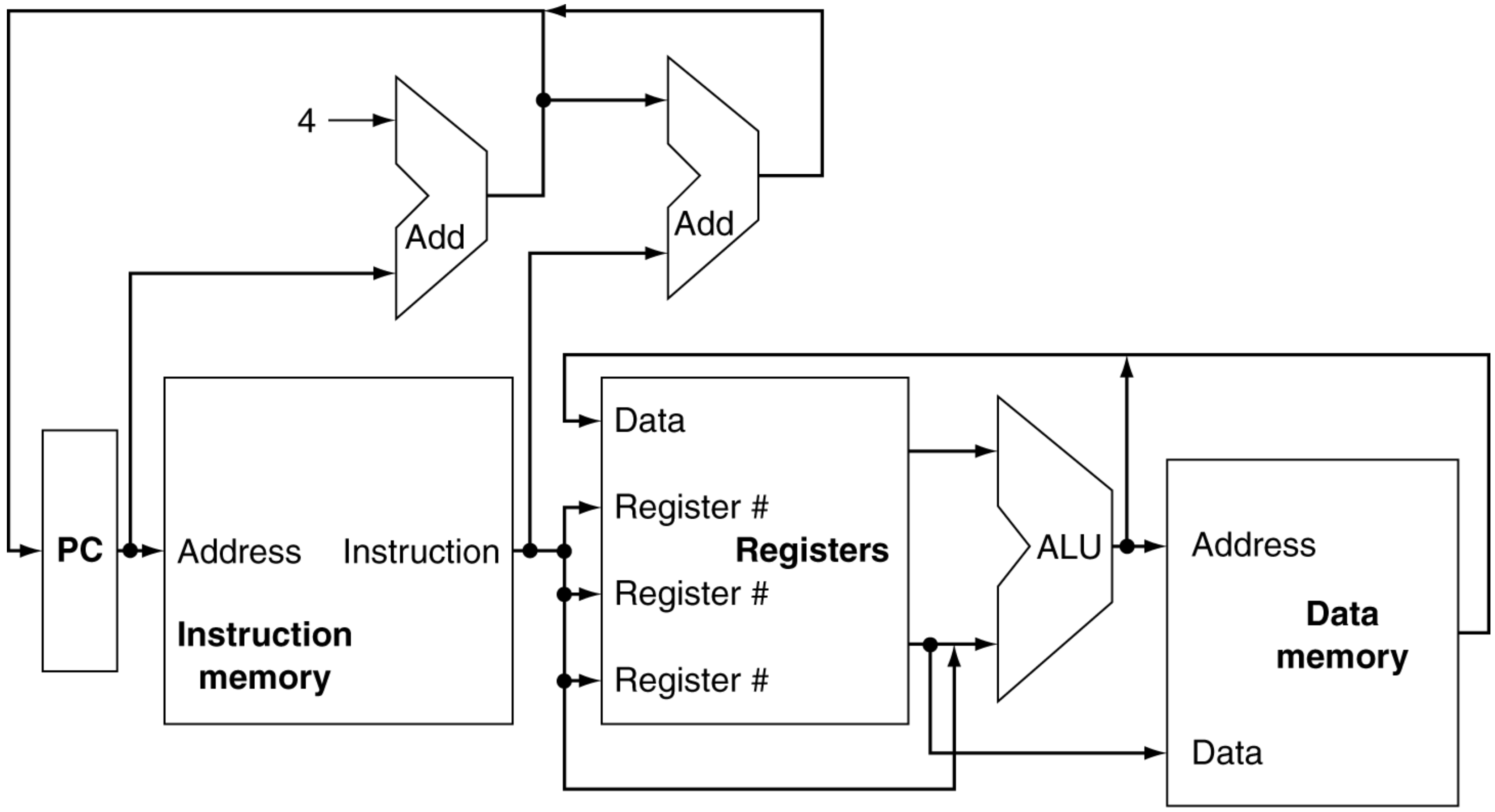
Format of jump instruction

- j J - format

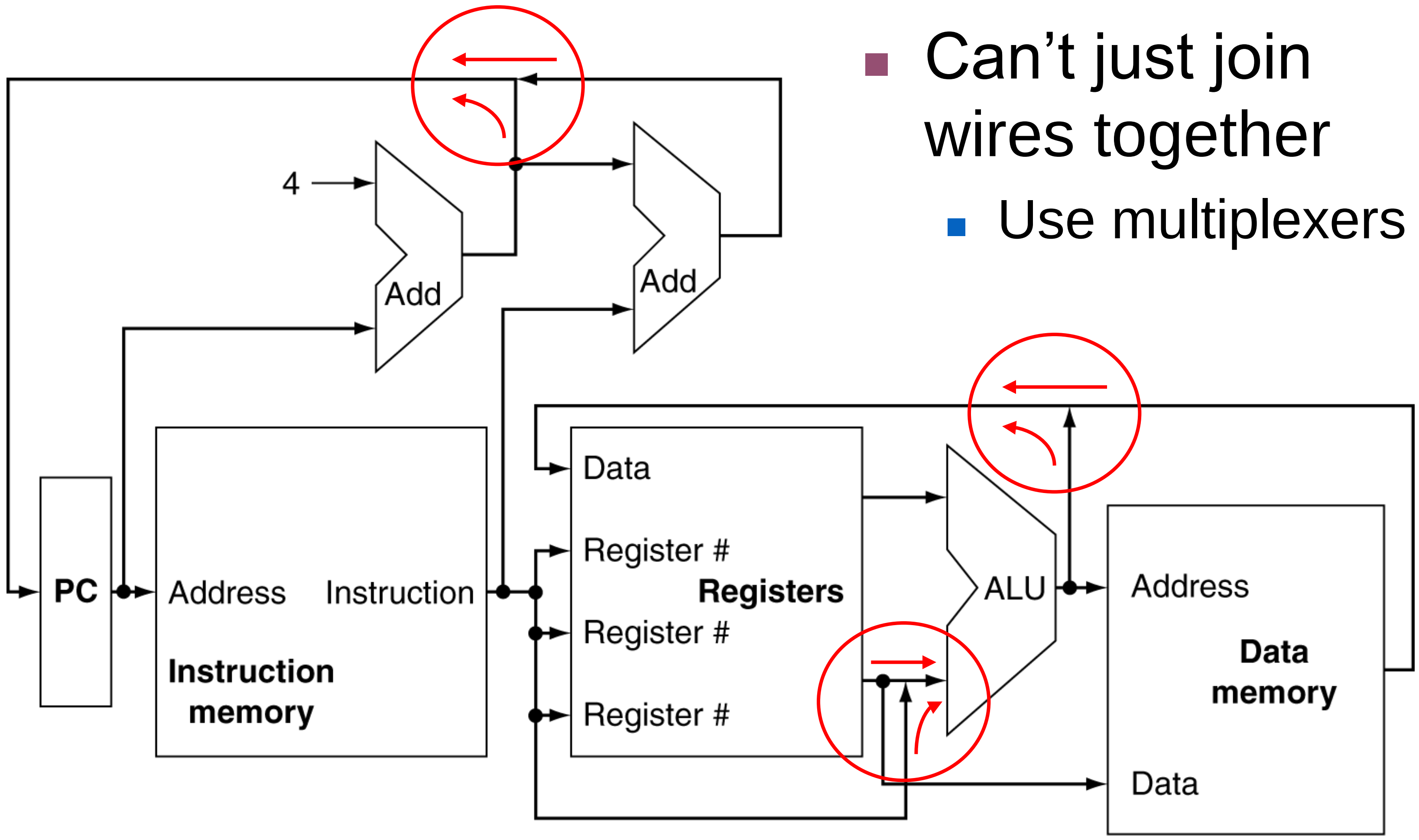
op	26 bit number



Basic CPU Overview



Multiplexers



- Can't just join wires together
  - Use multiplexers

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Data path for Branch Instruction

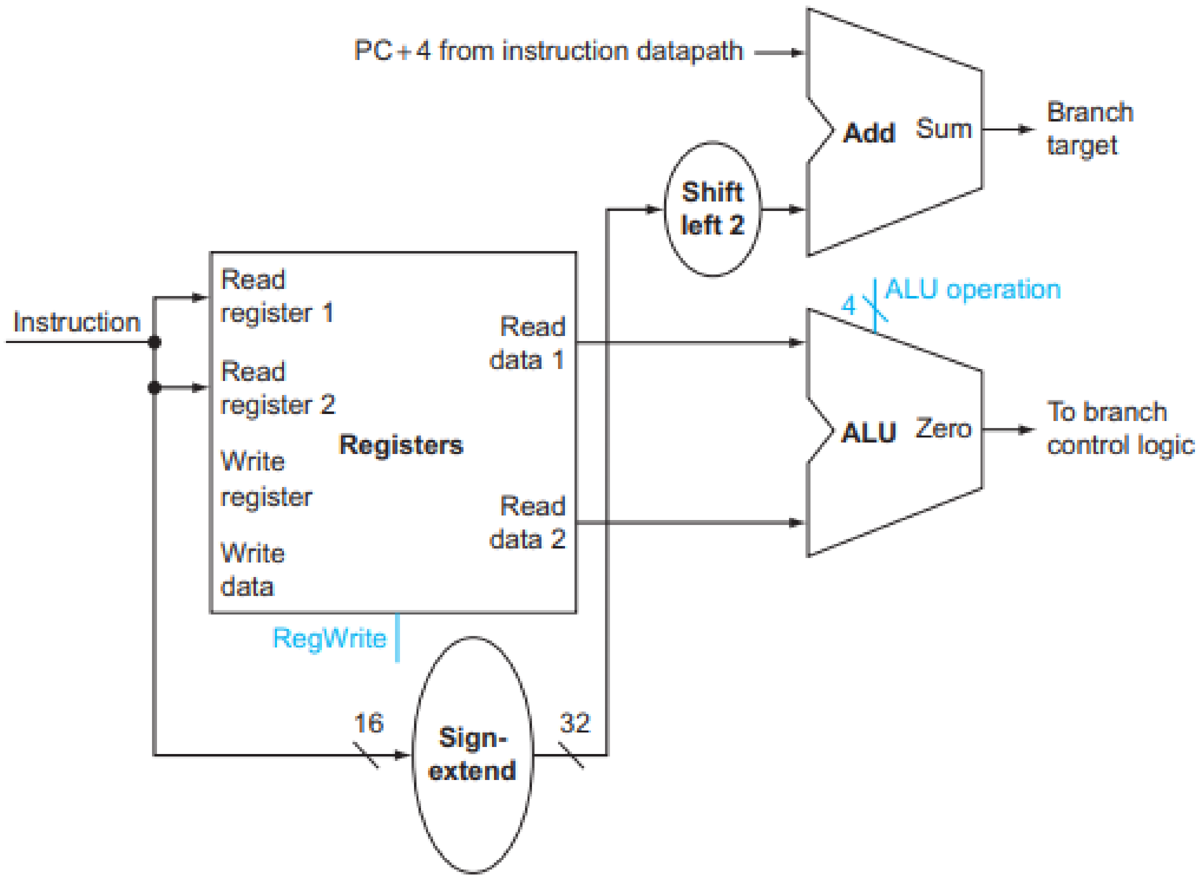
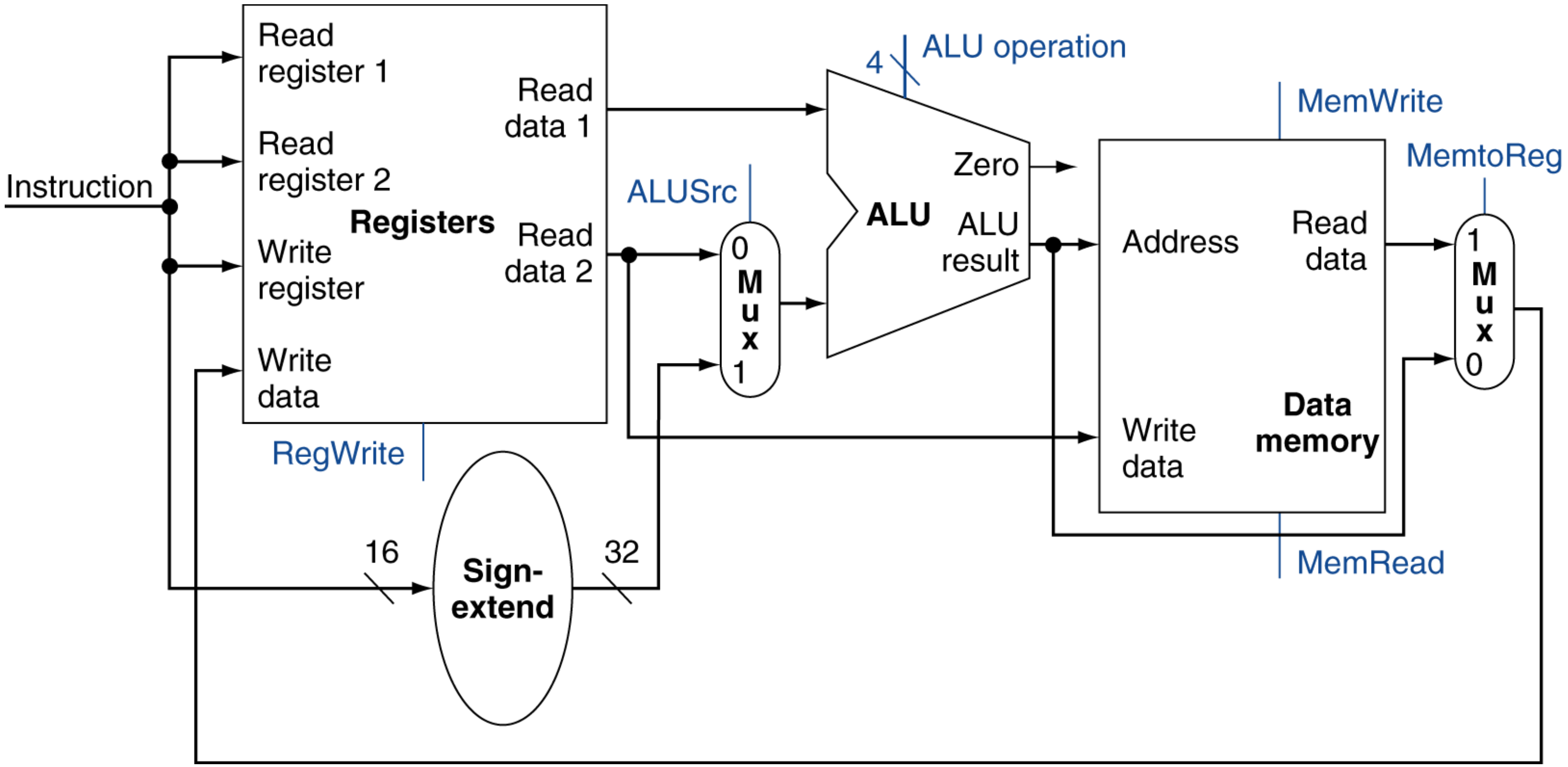


FIGURE 4.9 The datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the incremented PC and the sign-extended, lower 16 bits of the instruction (the branch displacement), shifted left 2 bits. The unit labeled *Shift left 2* is simply a routing of the signals between input and output that adds 00_{two} to the low-order end of the sign-extended offset field; no actual shift hardware is needed, since the amount of the "shift" is constant. Since we know that the offset was sign-extended from 16 bits, the shift will throw away only "sign bits." Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

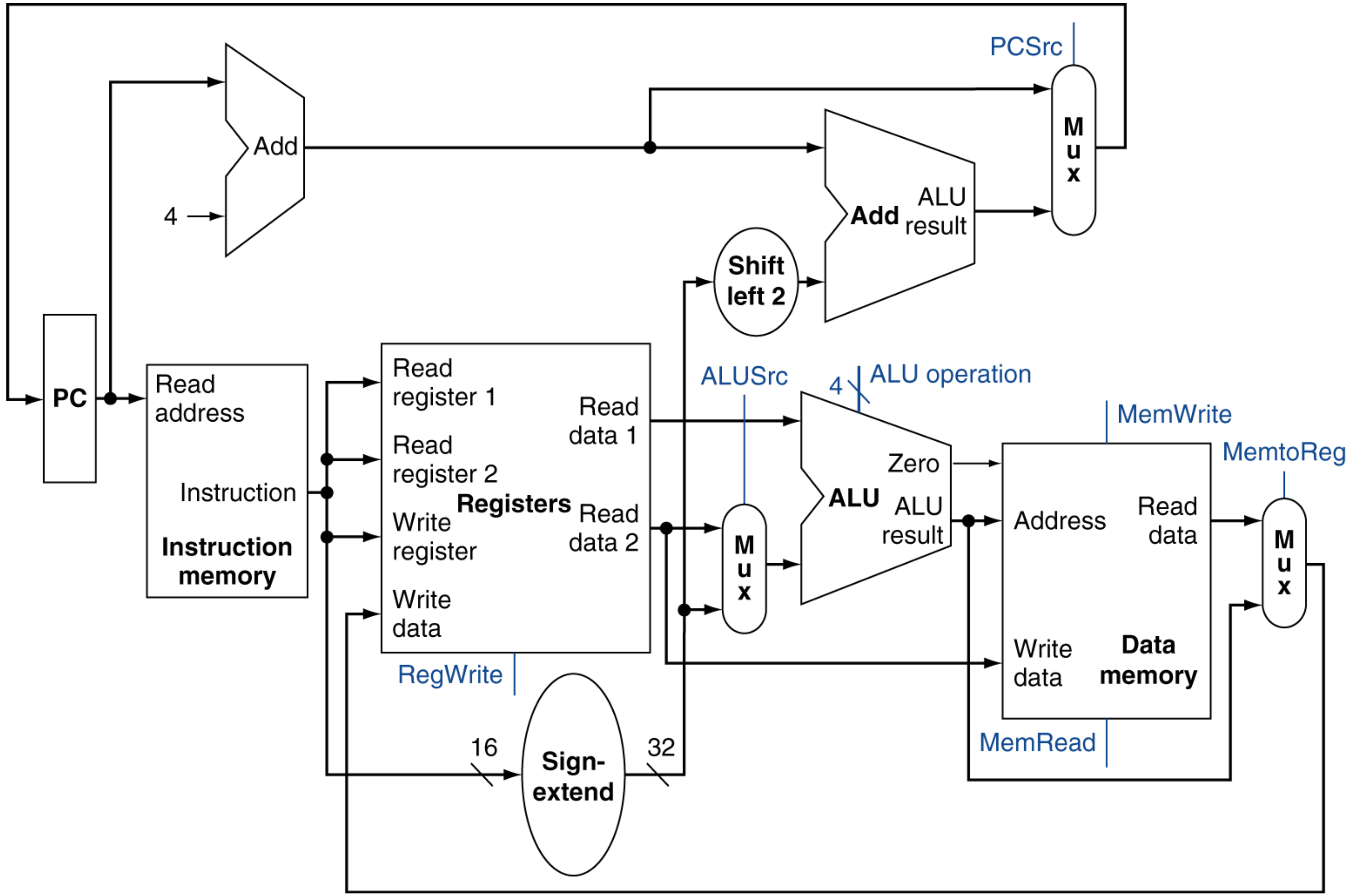
Composing the Elements

- First-cut data path does 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
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



Data path for Memory Instructions

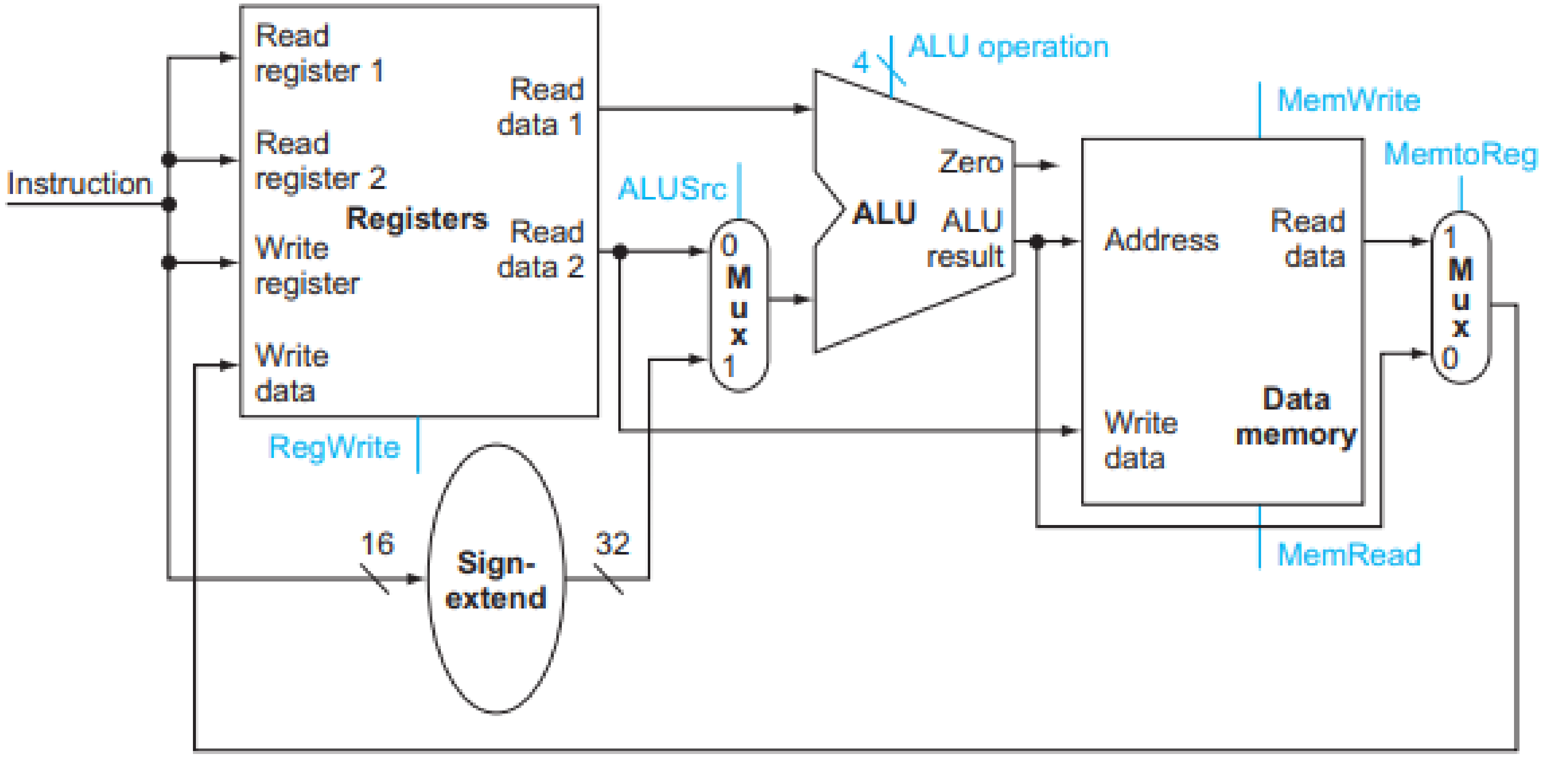
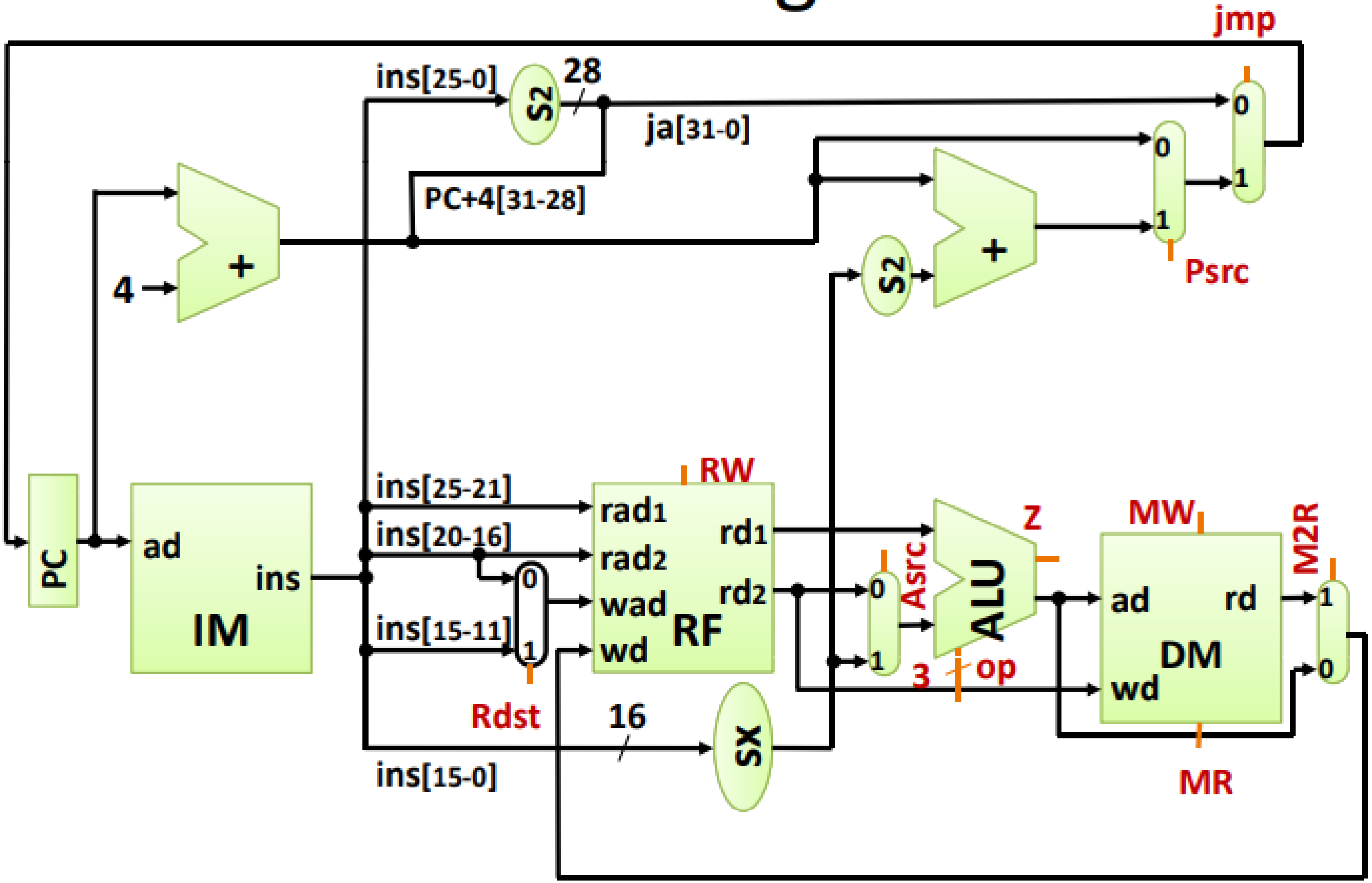


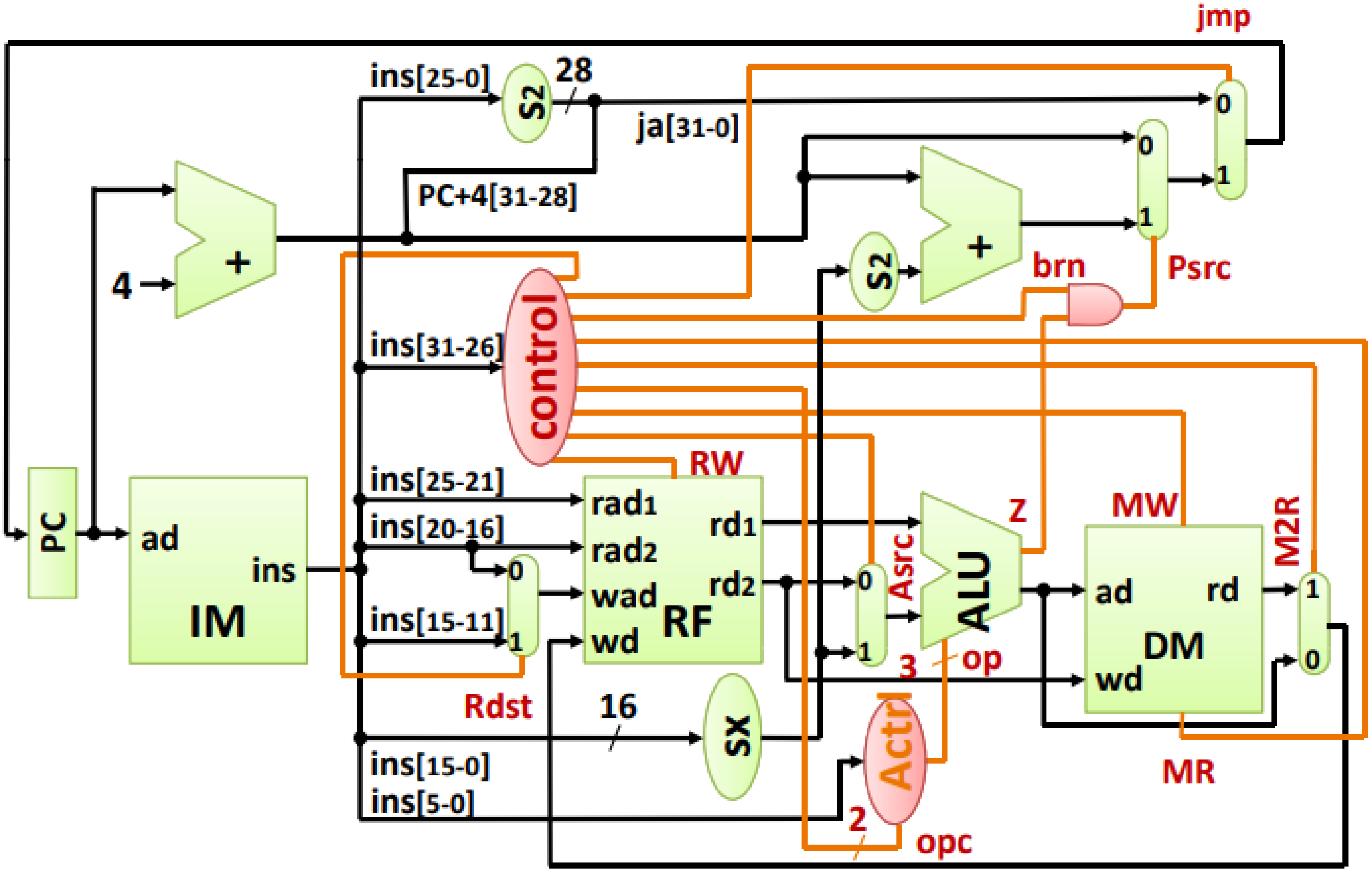
FIGURE 4.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example.

Identify Control Signals for Simple CPU

Control signals



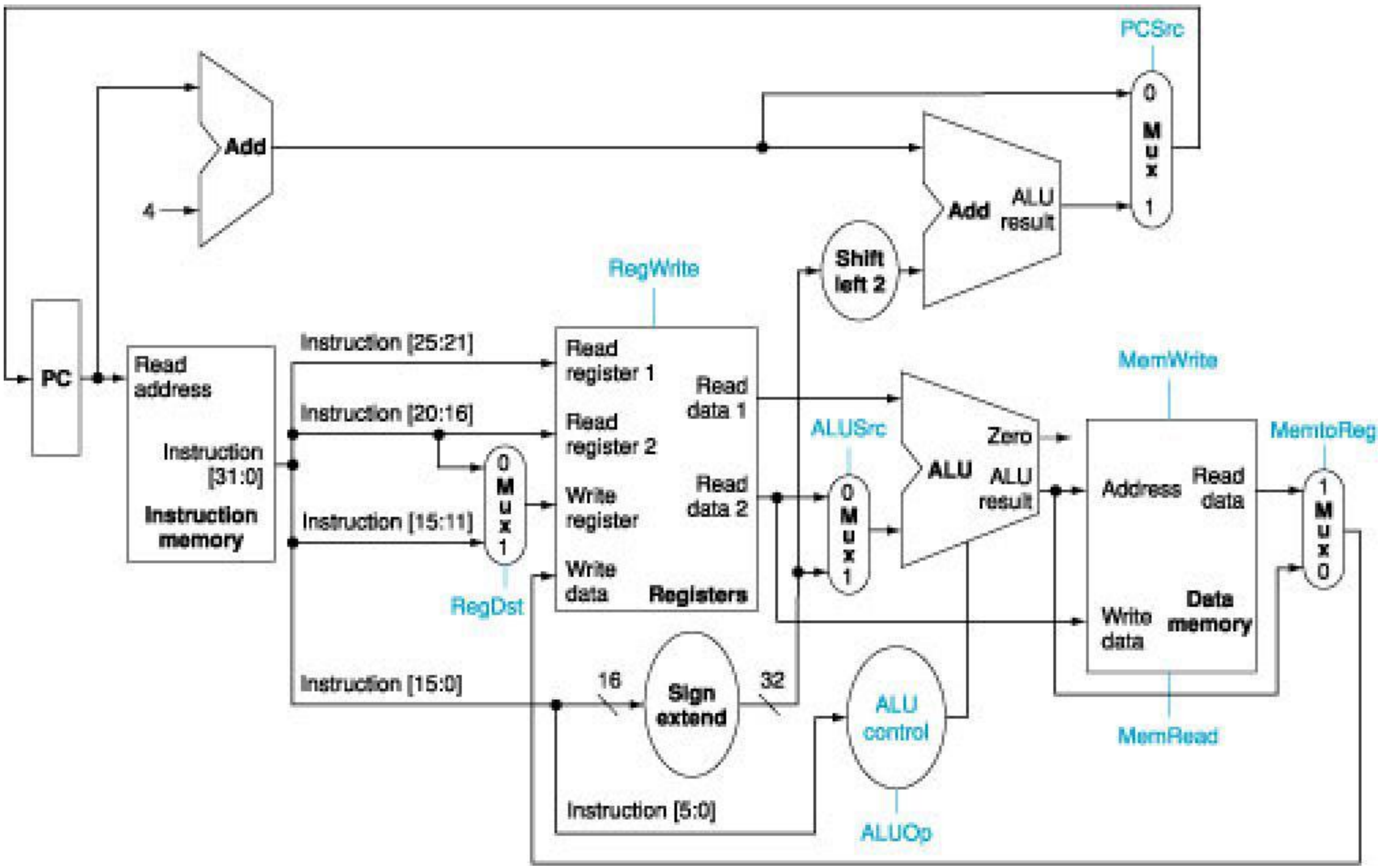
Combining Data path and Control path signals



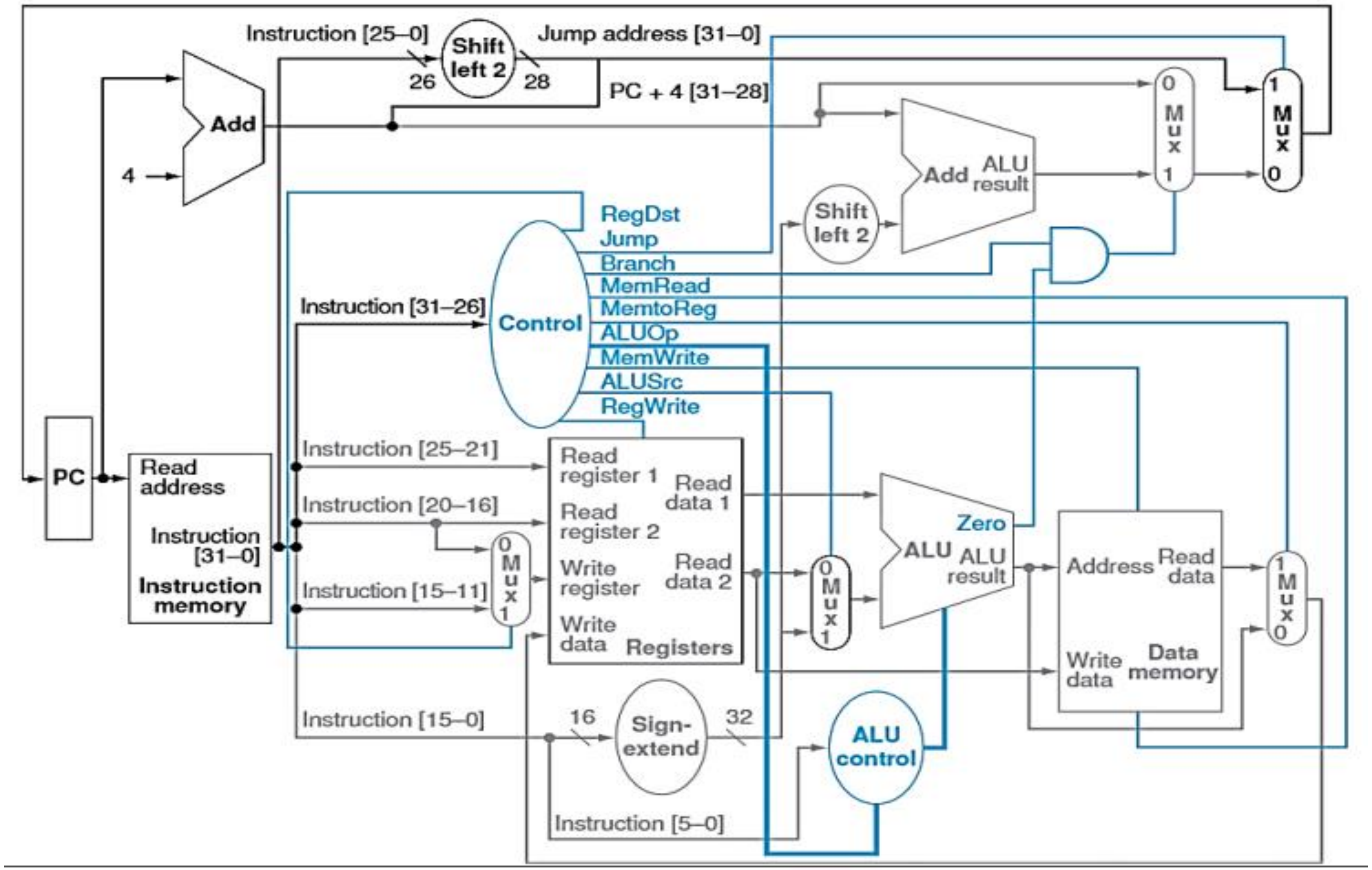
Control words in Simple MIPS CPU

Instruction	Opcode	Rdst	RW	Asrc	MW	MR	M2R	Brn	Jmp
Rtype	000000	1	1	0	0	0	0	0	0
Sw	101011	X	0	1	1	0	X	0	0
Lw	100011	0	1	1	0	1	1	0	0
Beq	000100	X	0	0	0	0	X	1	0
J	000010	X	0	X	0	0	X	X	1

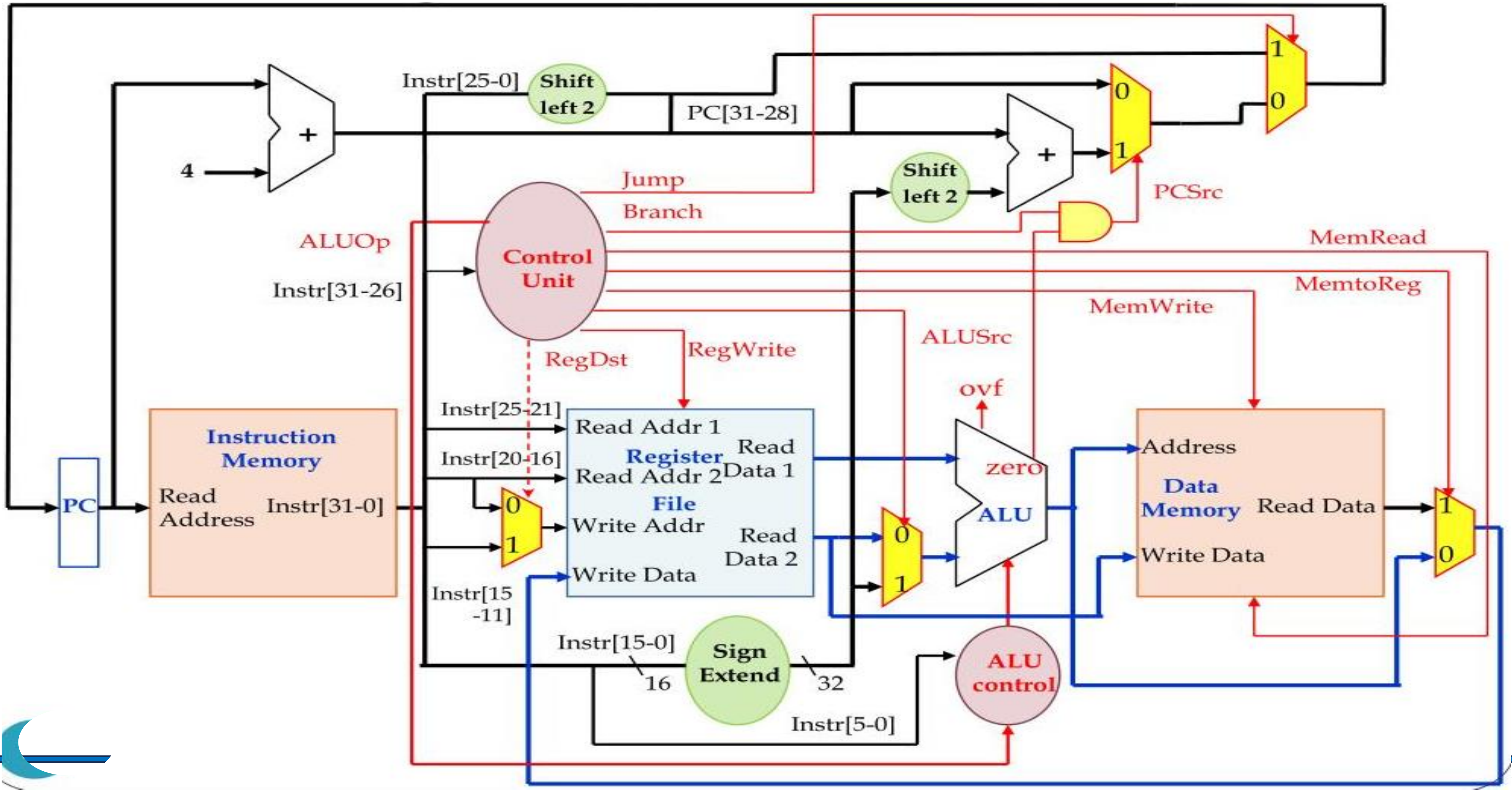
Control Signals needed in MIPS (P&H book style)



Data path with Control Signals in simple MIPS

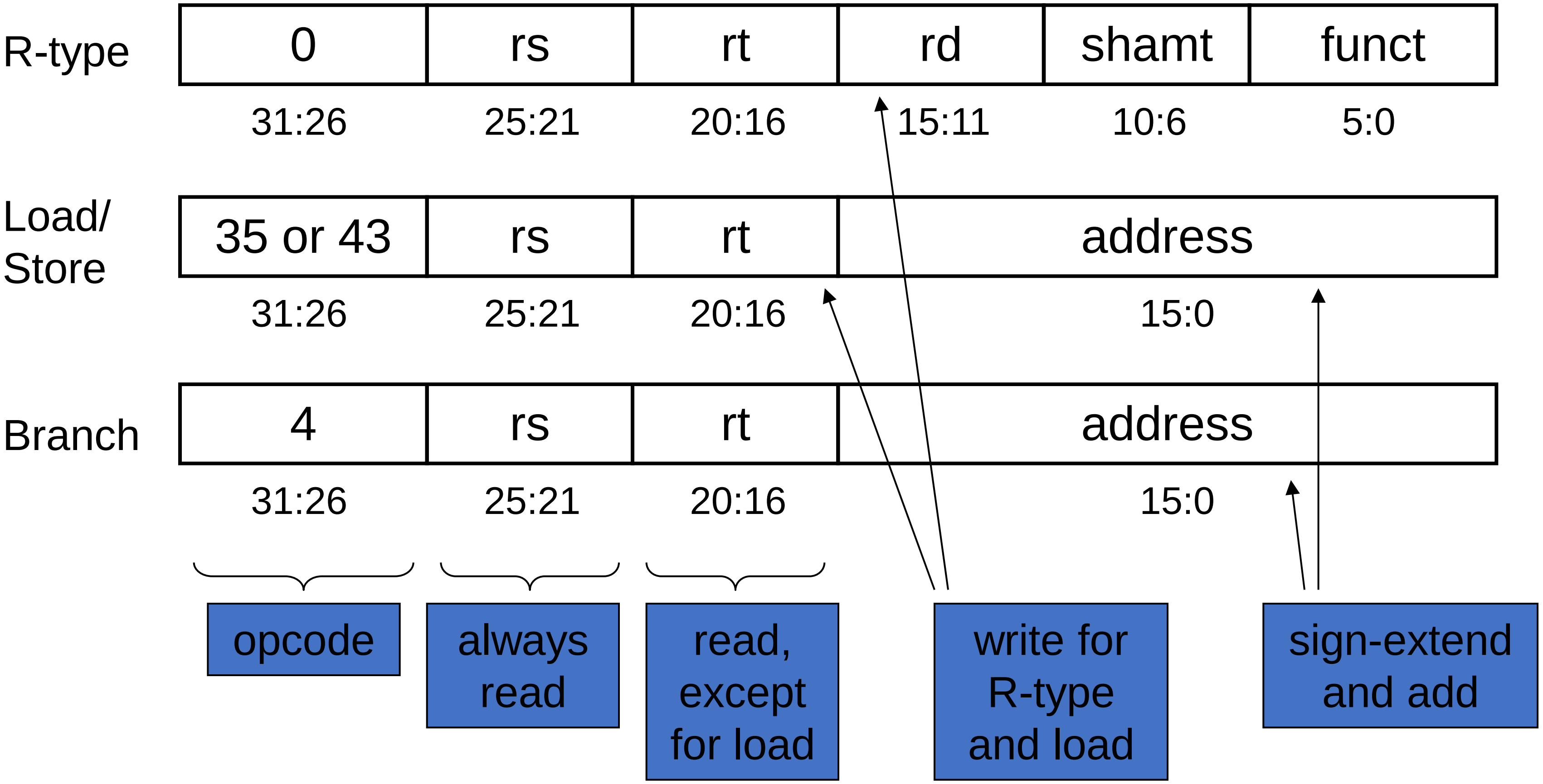


Another (clear) View of Control Signals



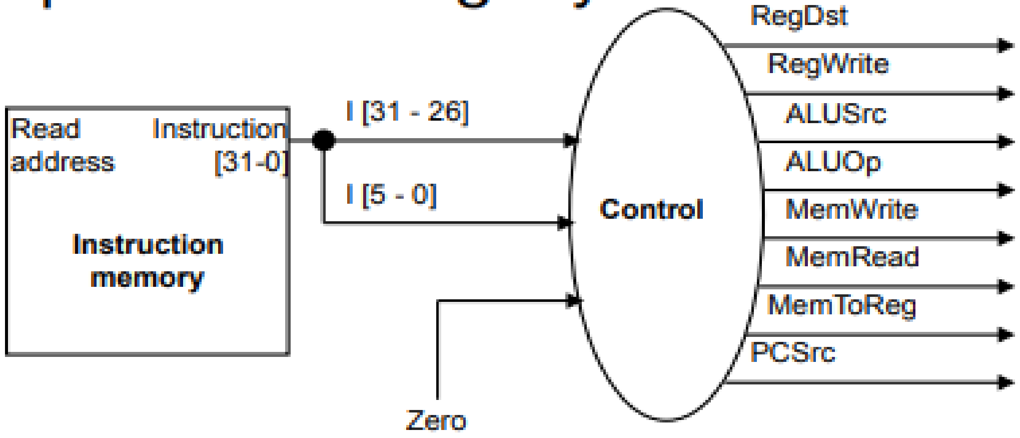
The Main Control Unit

- Control signals derived from instruction

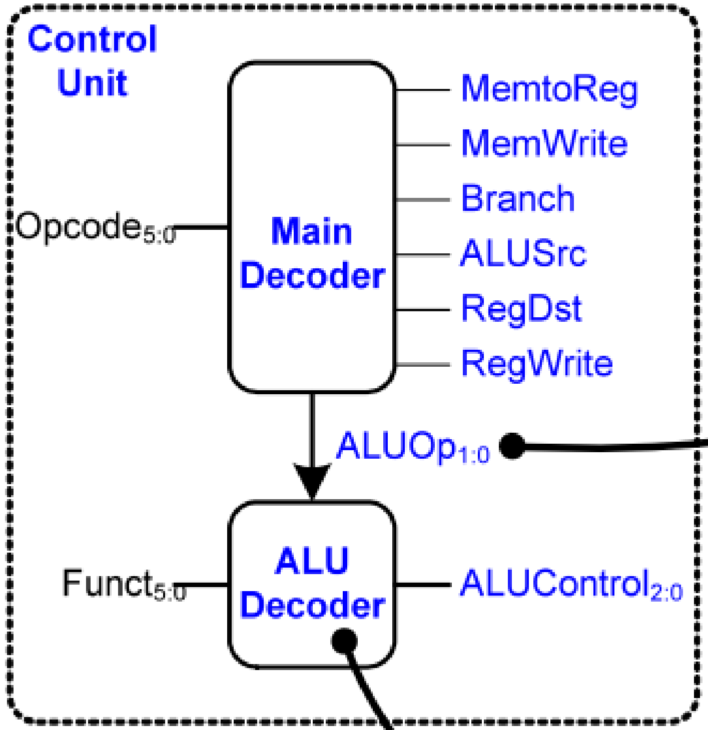


Generating Control Signals

- The control unit needs 13 bits of inputs
  - ❖ Six bits make up the instruction's opcode
  - ❖ Six bits come from the instruction's func field
  - ❖ It also needs the Zero output of the ALU
- The control unit generates 10 bits of output, corresponding to the signals mentioned earlier
- You can build the actual circuit by using big K-maps, big Boolean algebra, or big circuit design programs
- The textbook presents a slightly different control unit



Control words with ALU Op codes



ALUOp$_{1:0}$	Meaning
0 0	Add
0 1	Subtract
1 0	Look at Funct
1 1	*not used*

ALUOp$_{1:0}$	Funct	ALUControl$_{2:0}$
0 0	x	010 (Add)
x 1	x	110 (Subtract)
1 x	100000 (add)	010 (Add)
1 x	100010 (sub)	110 (Subtract)
1 x	100100 (and)	000 (And)
1 x	100101 (or)	001 (Or)
1 x	101010 (slt)	111 (SLT)

ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

Control words for different Instructions

Control signal table

Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X