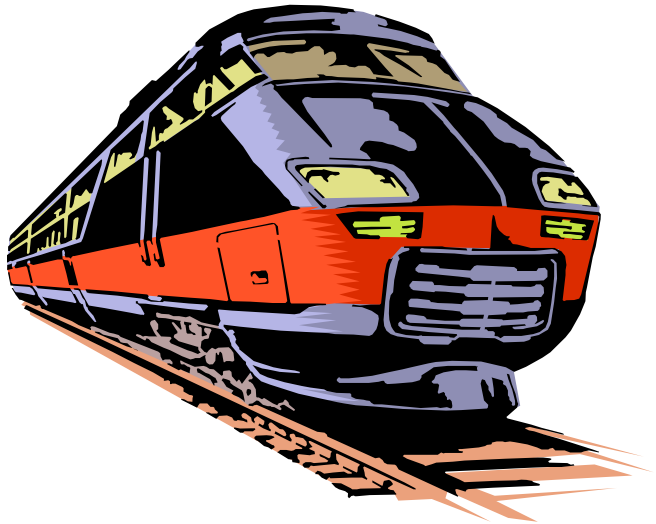
A single-cycle MIPS processor

- As previously discussed, an instruction set architecture is an *interface* that defines the hardware operations that are available to software.
- Any instruction set can be implemented in many different ways.
  - In a basic single-cycle implementation all operations take the same amount of time—a single cycle.
  - In a pipelined implementation, a processor can overlap the execution of several instructions, **potentially leading to big performance gains**.



Single-cycle implementation

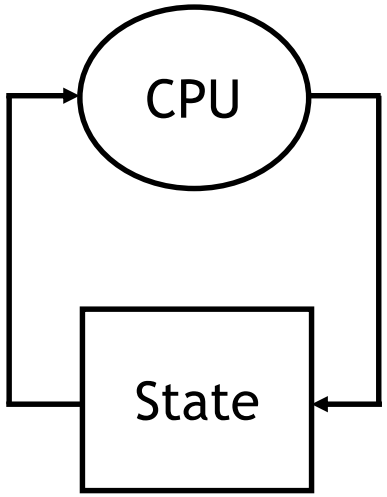
- In lecture, we will describe the implementation a simple MIPS-based instruction set supporting just the following operations.

Arithmetic:	add	sub	and	or	slt
Data Transfer:	lw	sw			
Control:	beq	j			

- Today we'll build a single-cycle implementation of this instruction set.
  - We will discuss three types of instructions (R-type, I-Type, J-Type) separately.
  - All instructions will execute in the same amount of time; this will determine the clock cycle time for our performance equations.
  - We'll explain the datapath first, and then make the control unit.

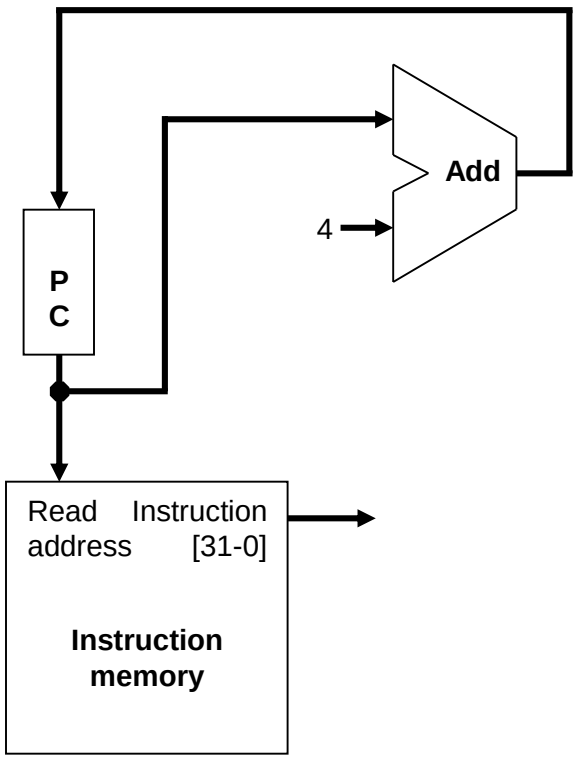
Computers are state machines

- A computer is just a big fancy state machine.
  - Registers, memory, hard disks and other storage form the state.
  - The processor keeps reading and updating the state, according to the instructions in some program.



Instruction fetching

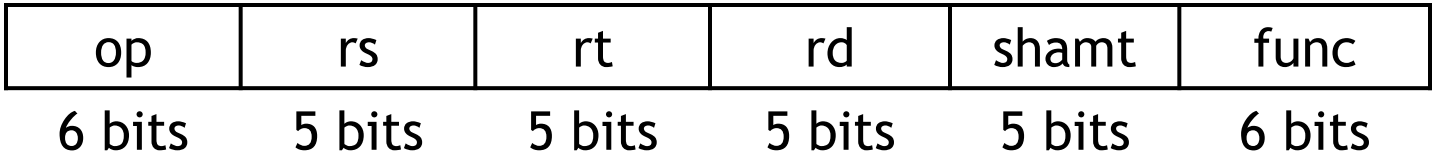
- It's easier to use a Harvard architecture at first, with programs and data stored in *separate* memories.
  - For today, we will assume you cannot write to the instruction memory.
  - Pretend it's already loaded with a program, which doesn't change while it's running.



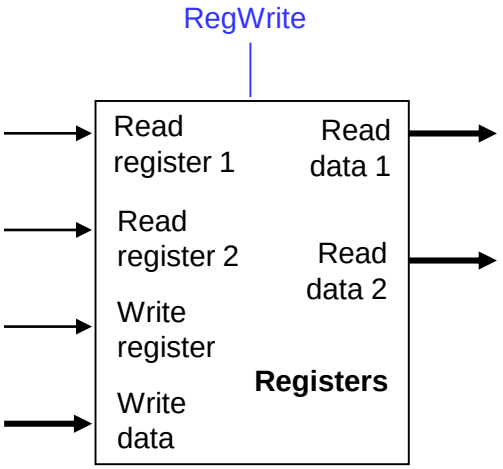
- The CPU is always in an infinite loop, fetching instructions from memory and executing them.
- The program counter or PC register holds the address of the current instruction.
- MIPS instructions are each four bytes long, so the PC should be incremented by four to read the next instruction in sequence.

Decoding instructions (R-type)

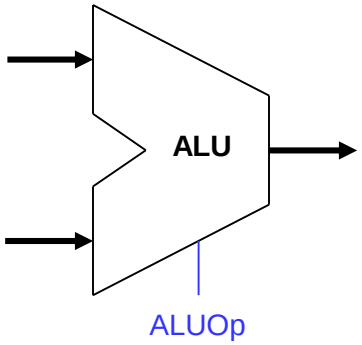
- We have seen encodings of MIPS instructions as 32-bit values
- *Example*: R-type instructions



- Our register file stores thirty-two 32-bit values
  - Each register specifier is 5 bits long
  - You can read from two registers at a time
  - RegWrite is 1 if a register should be written

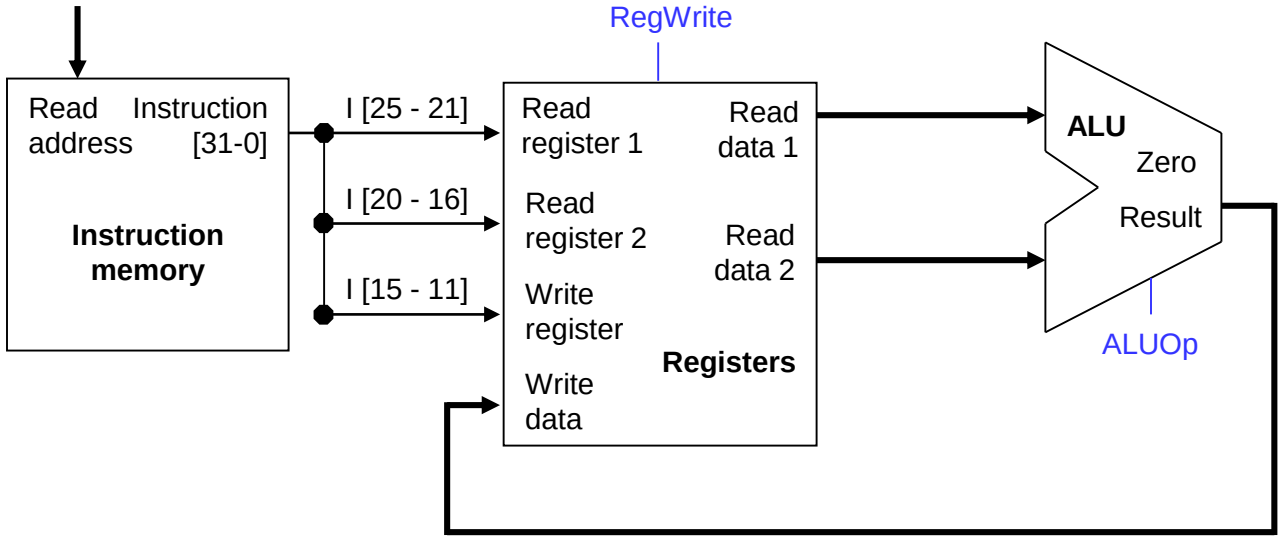


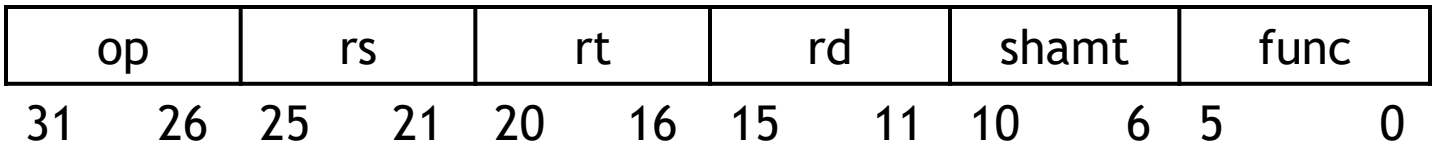
- Opcode determines ALUOp



Executing an R-type instruction

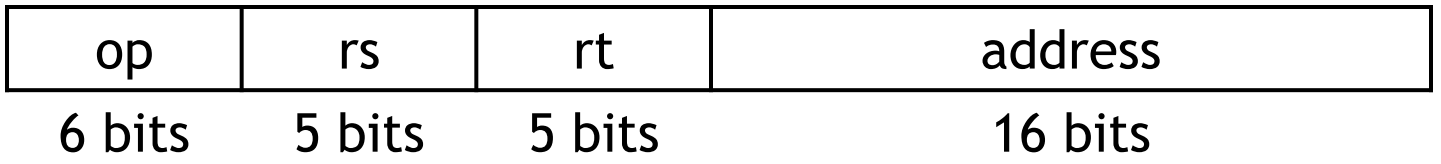
1. Read an instruction from the instruction memory.
2. The source registers, specified by instruction fields rs and rt, should be read from the register file.
3. The ALU performs the desired operation.
4. Its result is stored in the destination register, which is specified by field rd of the instruction word.

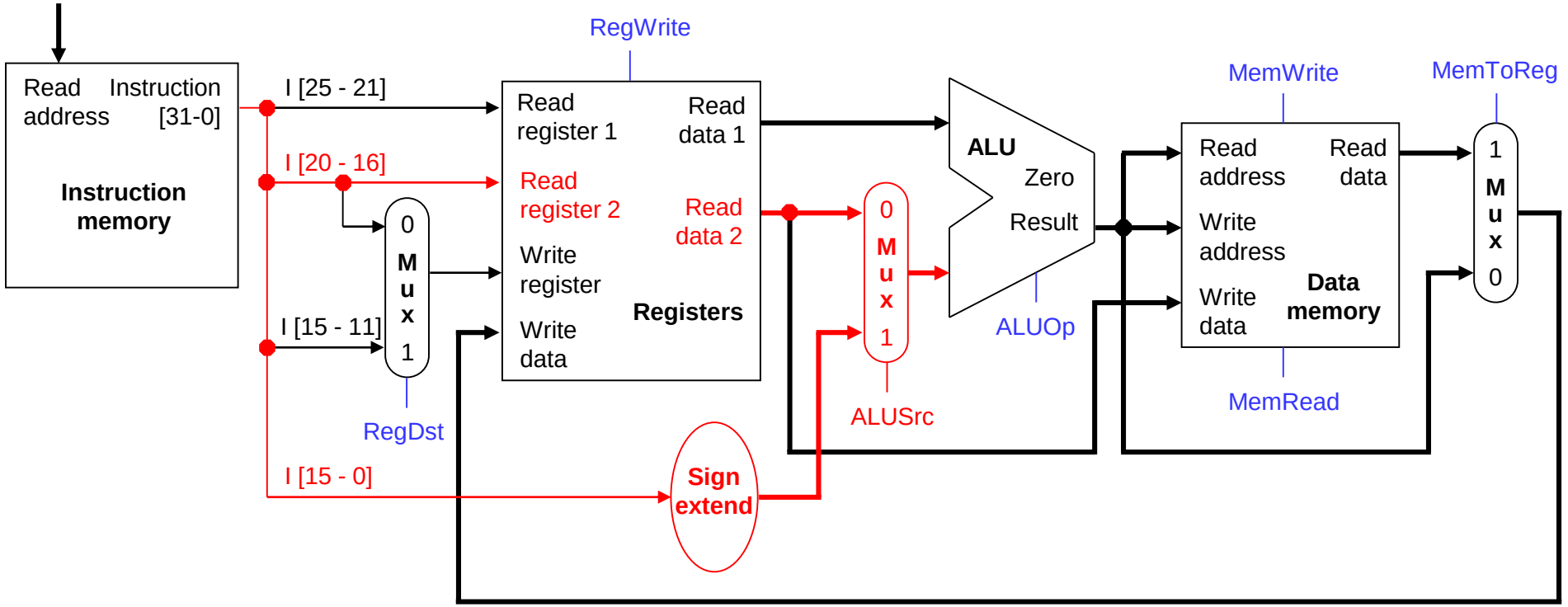




Decoding I-type instructions

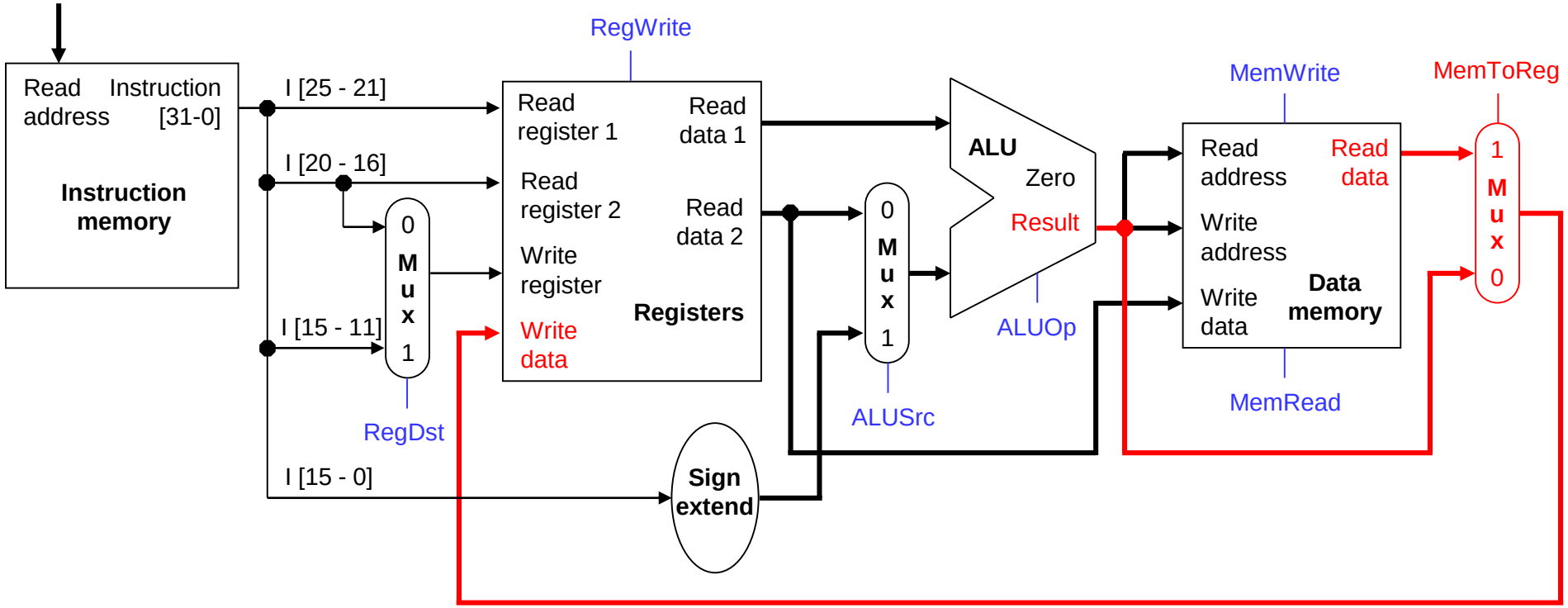
- The lw, sw and beq instructions all use the I-type encoding
  - rt is the *destination* for lw, but a *source* for beq and sw
  - address is a 16-bit signed constant (can be ALU source, sign-extended)





MemToReg

- The register file's "Write data" input has a similar problem. It must be able to store *either* the ALU output of R-type instructions, *or* the data memory output for lw.
- We add a mux, controlled by MemToReg, to select between saving the ALU result (0) or the data memory output (1) to the registers.



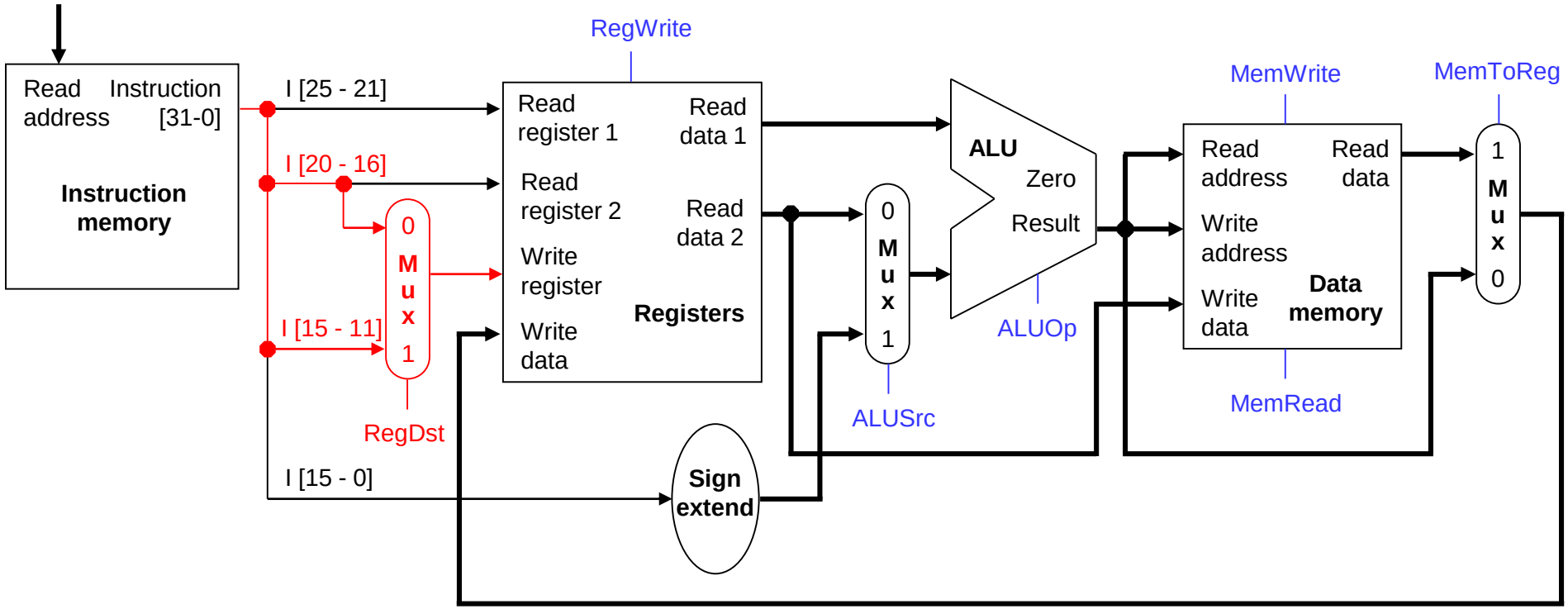
RegDst

- A final annoyance is the destination register of lw is in *rt* instead of rd.

op	rs	rt	address

lw $rt, address($rs)

- We'll add one more mux, controlled by RegDst, to select the destination register from either instruction field rt (0) or field rd (1).

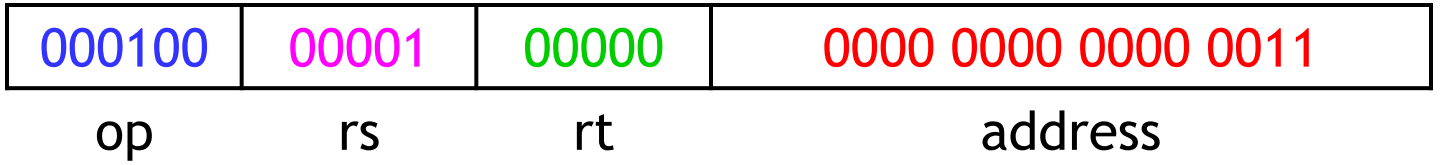


Branches

- For branch instructions, the constant is not an address but an *instruction offset* from the next program counter to the desired address

```
        beq   $at, $0, L
        or    $v1, $v0, $0
        add   $v1, $v1, $v1
        j     Somewhere
L:      add   $v1, $v0, $v0
```

- The target address L is three *instructions* past the or, so the encoding of the branch instruction has 0000 0000 0000 0011 for the address field

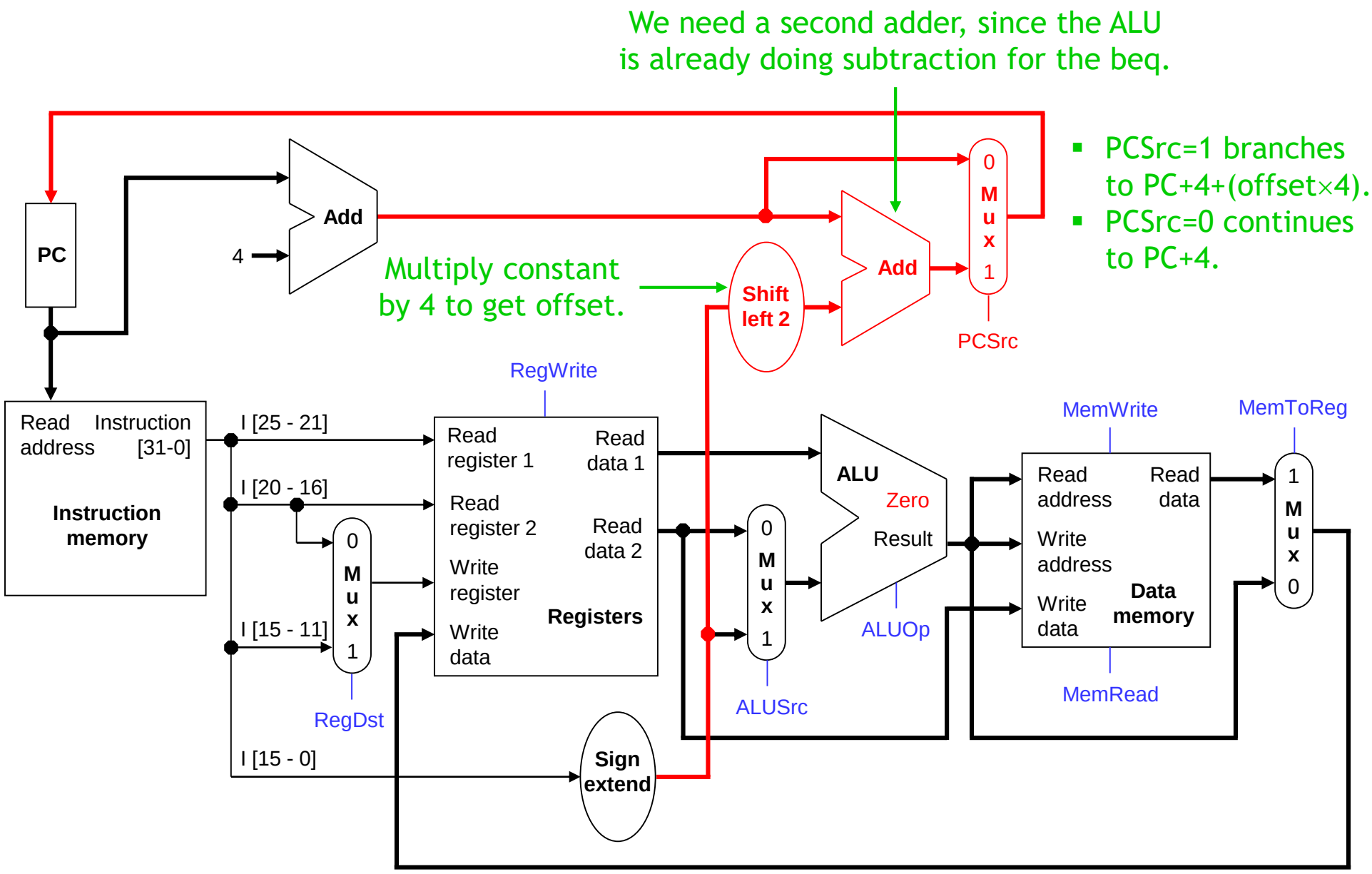


- Instructions are four bytes long, so the actual memory offset is 12 bytes

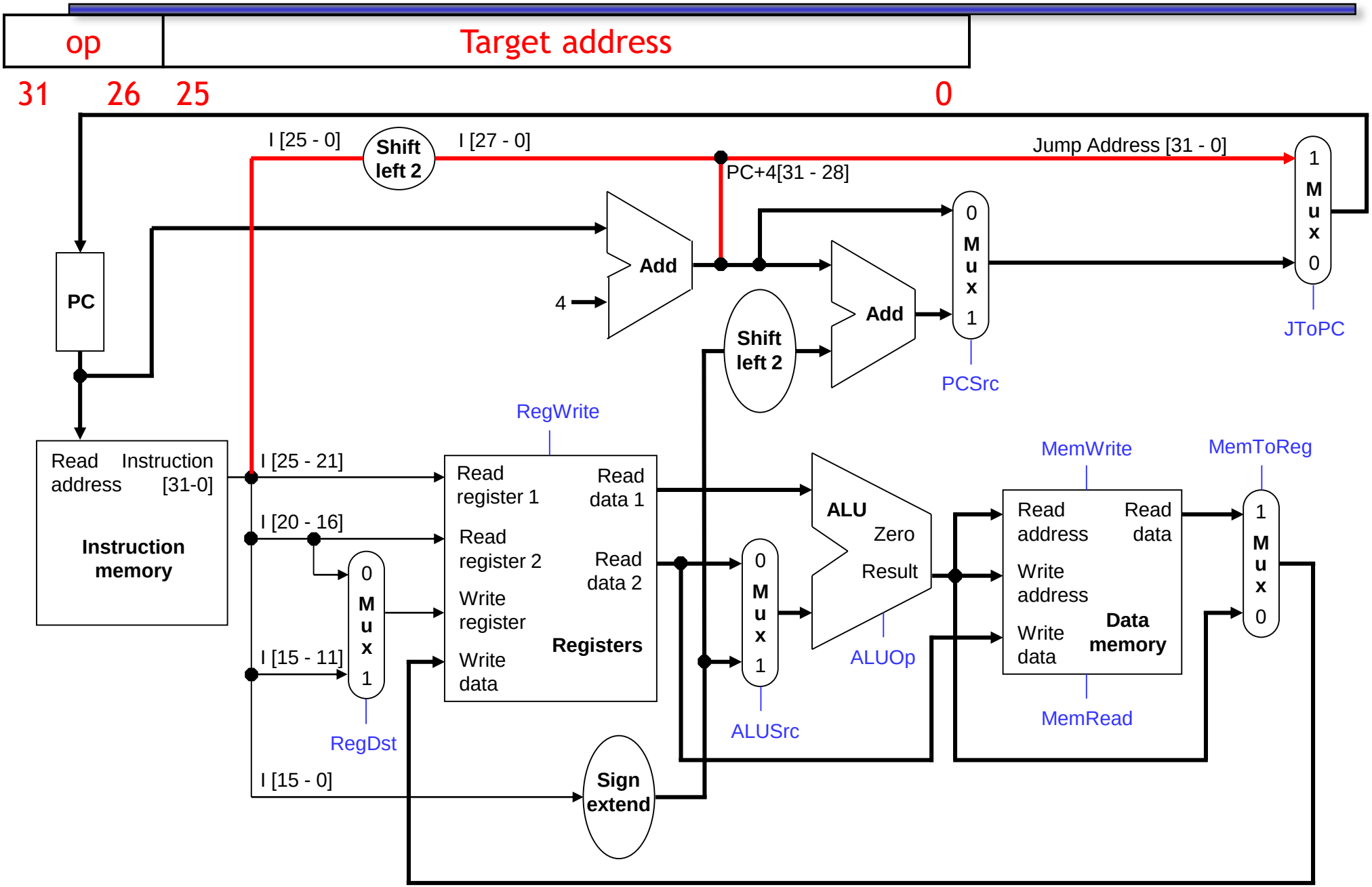
The steps in executing a beq

1. Fetch the instruction, like beq $at, $0, offset, from memory
2. Read the source registers, $at and $0, from the register file
3. Compare the values (e.g., by XORing them in the ALU)
4. If the XOR result is 0, the source operands were equal and the PC should be loaded with the target address, PC + 4 + (offset x 4)
5. Otherwise the branch should not be taken, and the PC should just be incremented to PC + 4 to fetch the next instruction sequentially

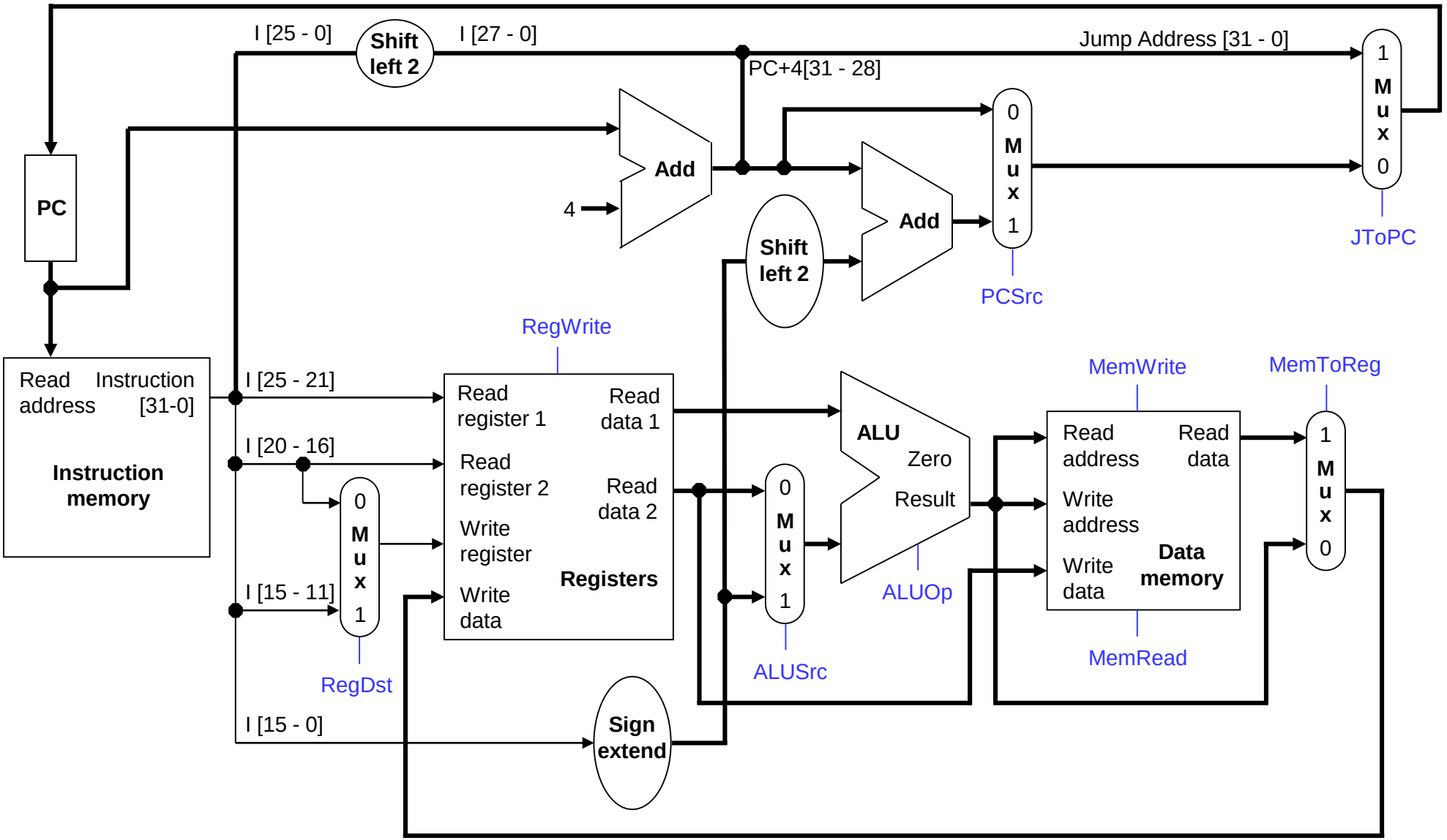
Branching hardware



Executing an J-type (j and jal) instruction



The final datapath

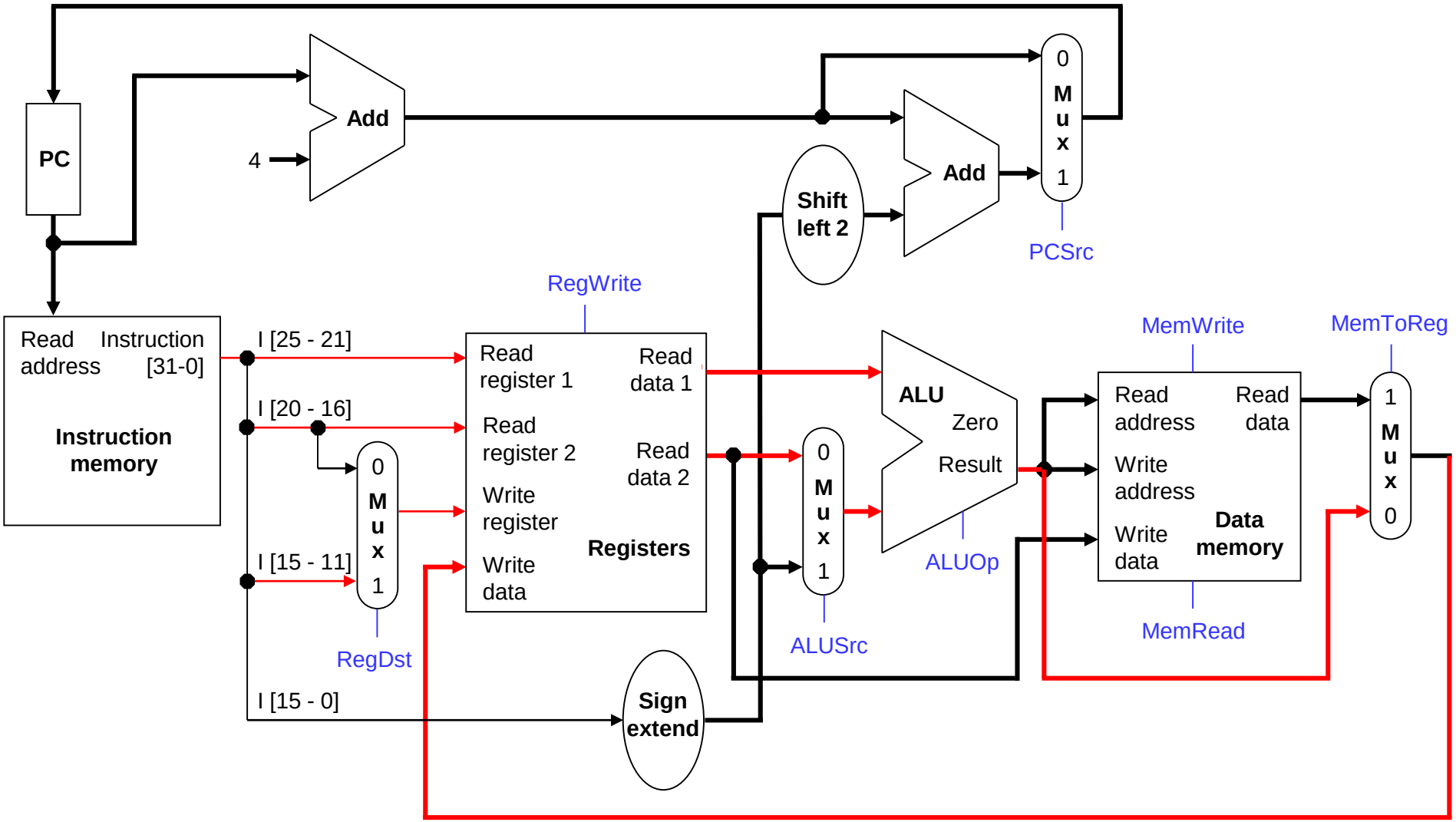


Control path

- The control unit is responsible for setting all the control signals so that each instruction is executed properly.
  - The control unit's input is the 32-bit instruction word.
  - The outputs are values for the blue control signals in the datapath.
- Most of the signals can be generated from the instruction opcode alone, and not the entire 32-bit word.
- To illustrate the relevant control signals, we will show the route that is taken through the datapath by R-type, lw, sw and beq instructions.

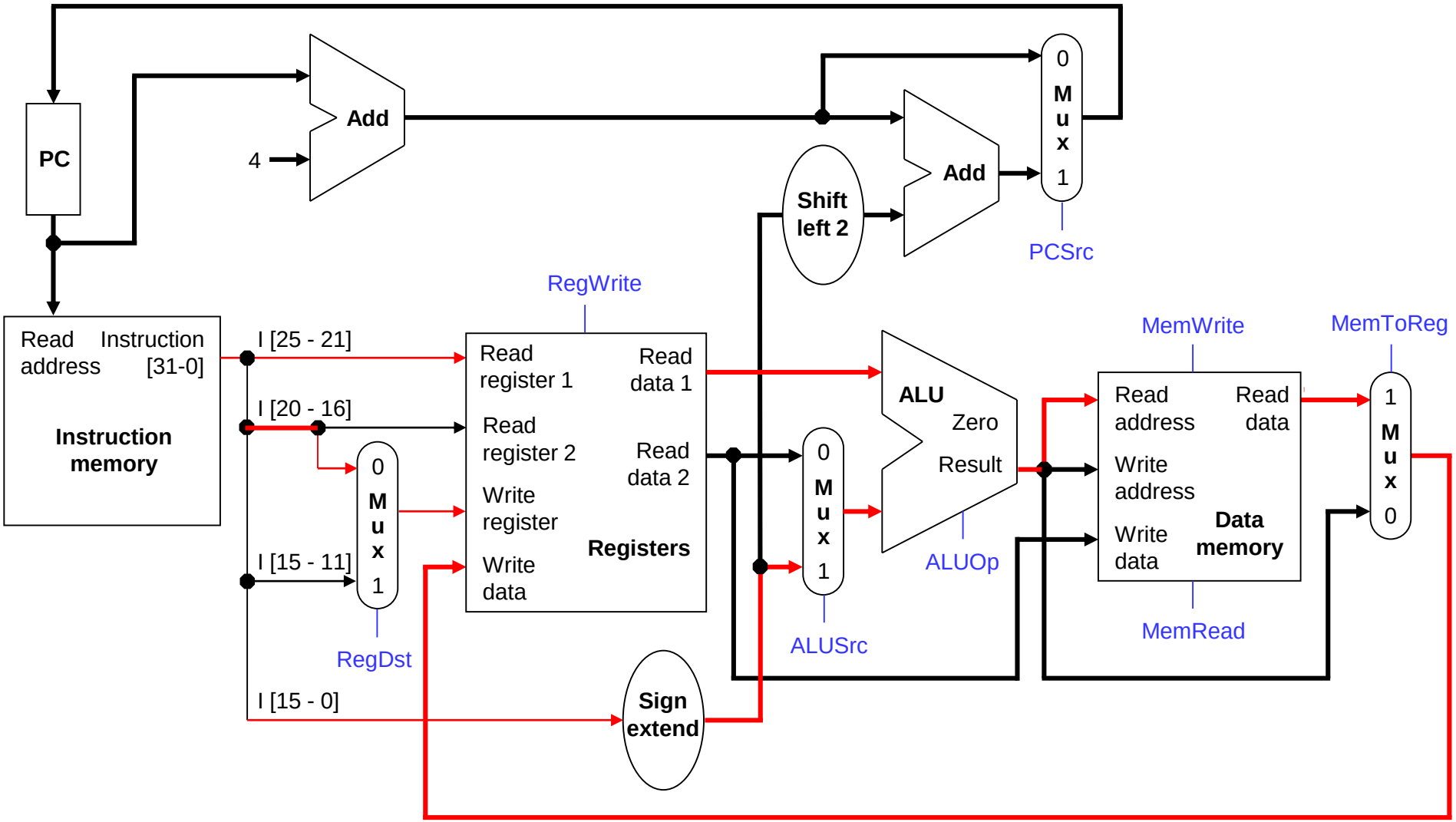
R-type instruction path

- The R-type instructions include add, sub, and, or, and slt.
- The ALUOp is determined by the instruction's "func" field.



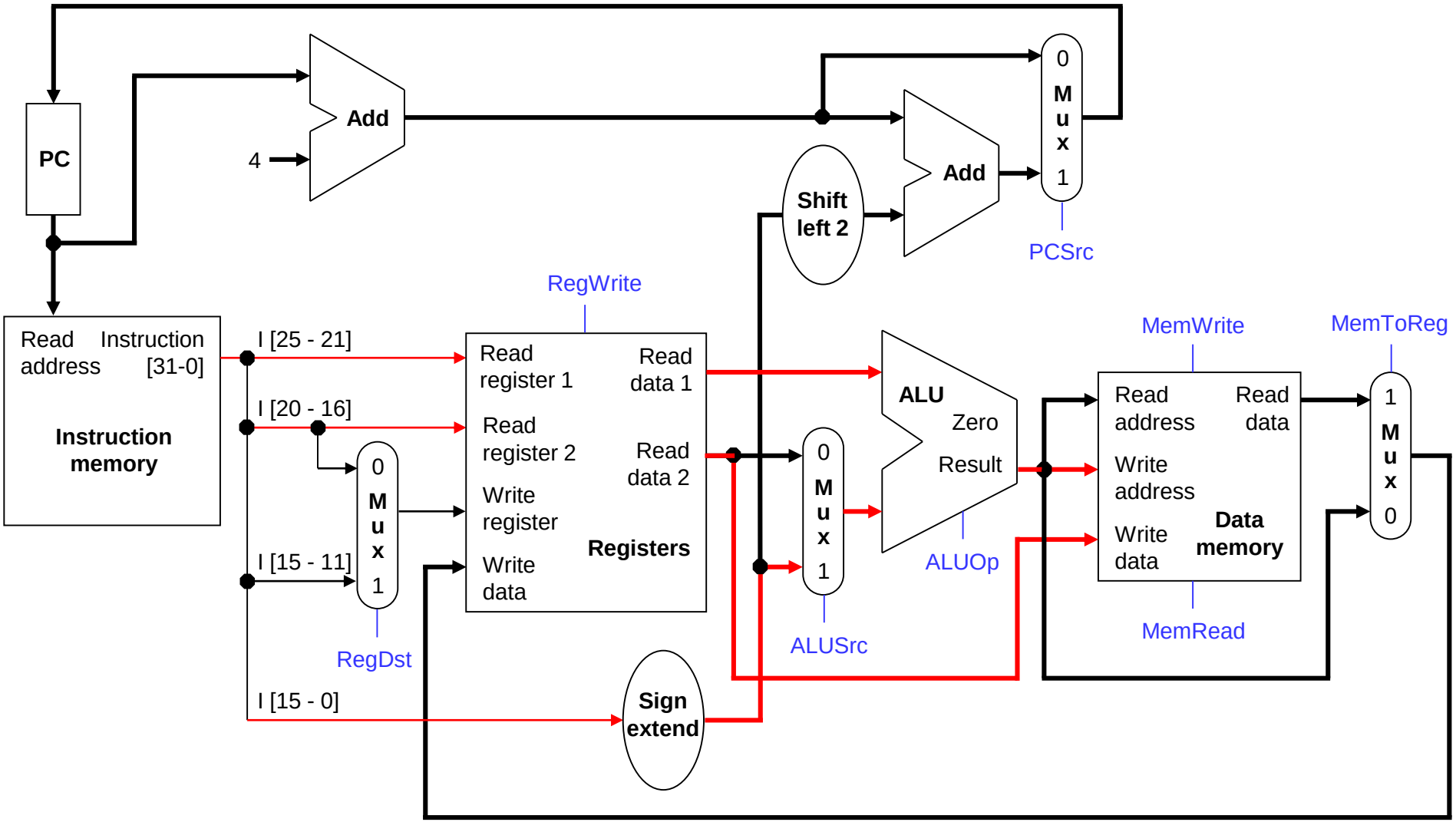
lw instruction path

- An example load instruction is lw $t0, -4($sp).
- The ALUOp must be 010 (add), to compute the effective address.



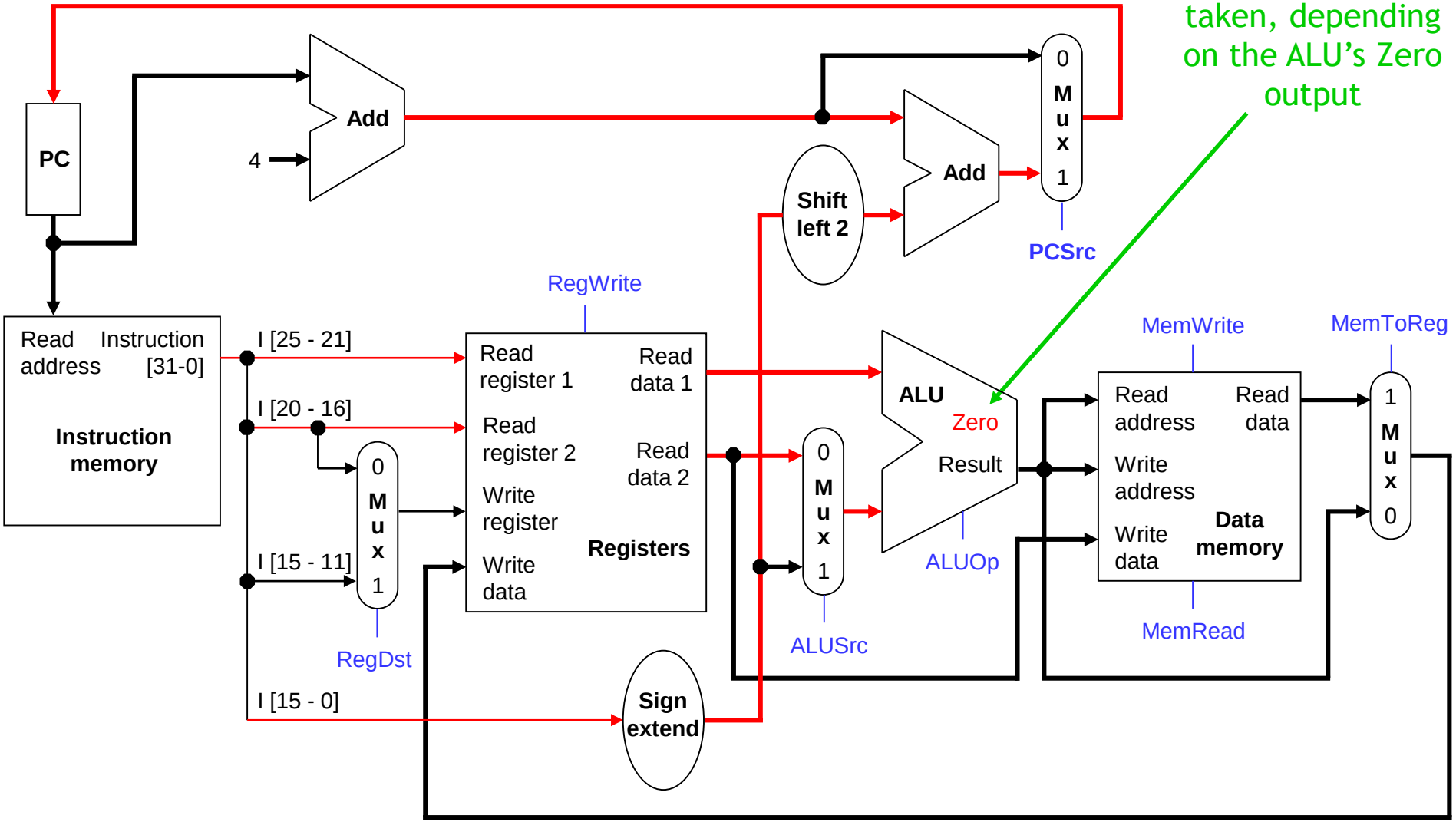
sw instruction path

- An example store instruction is sw $a0, 16($sp).
- The ALUOp must be 010 (add), again to compute the effective address.

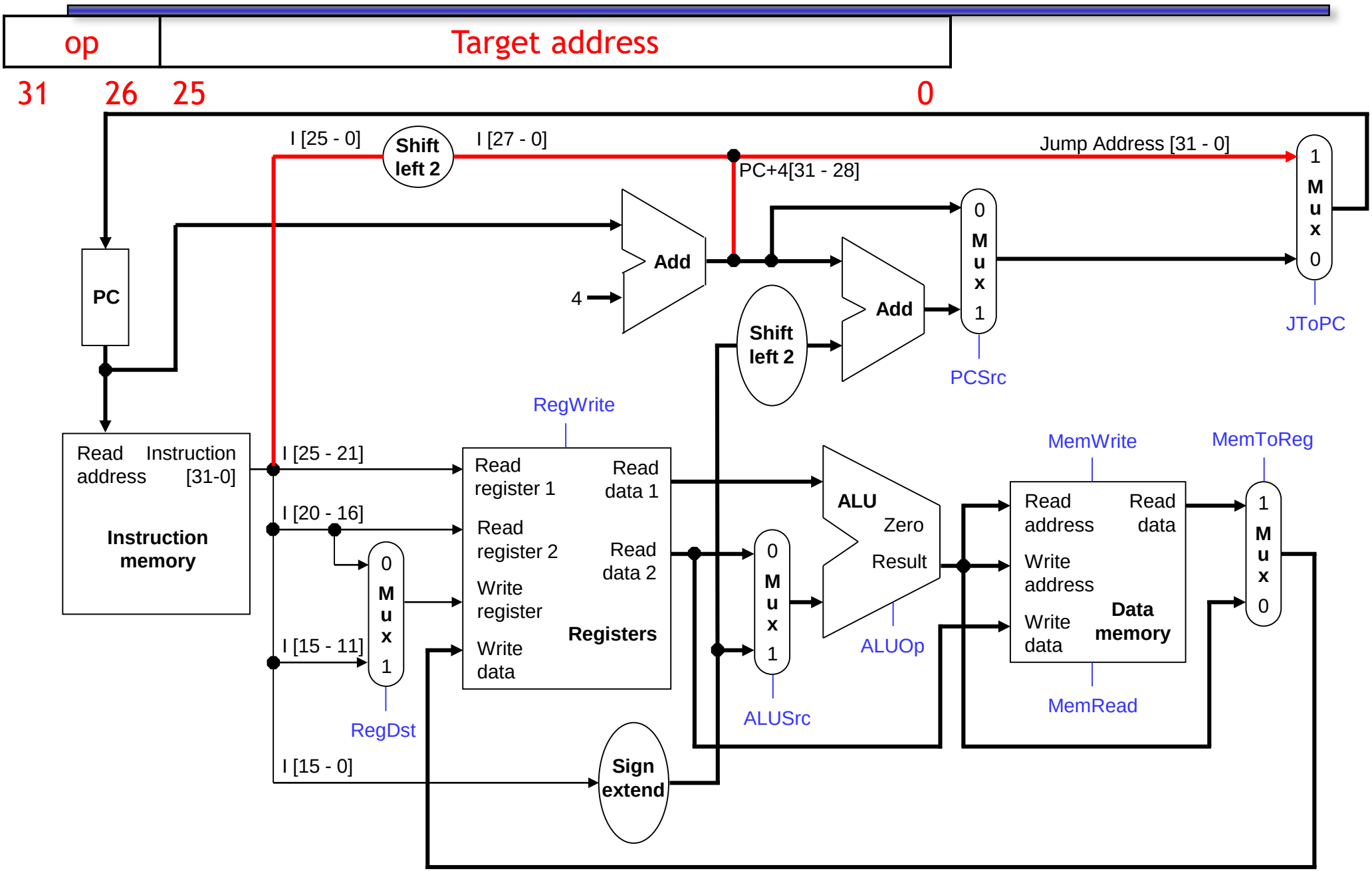


beq instruction path

- One sample branch instruction is beq $at, $0, offset.
- The ALUOp is 110 (subtract), to test for equality.



J-type (j and jal) instruction path



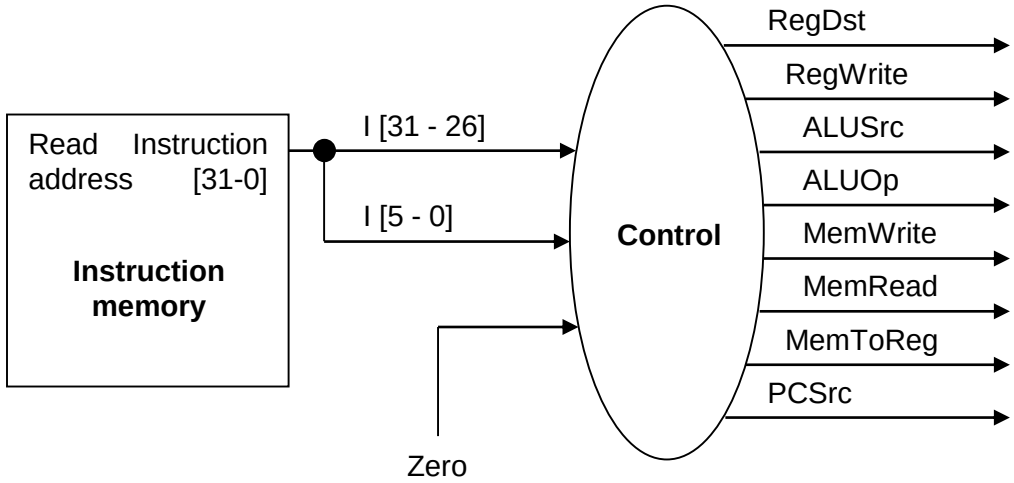
Control signal table

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

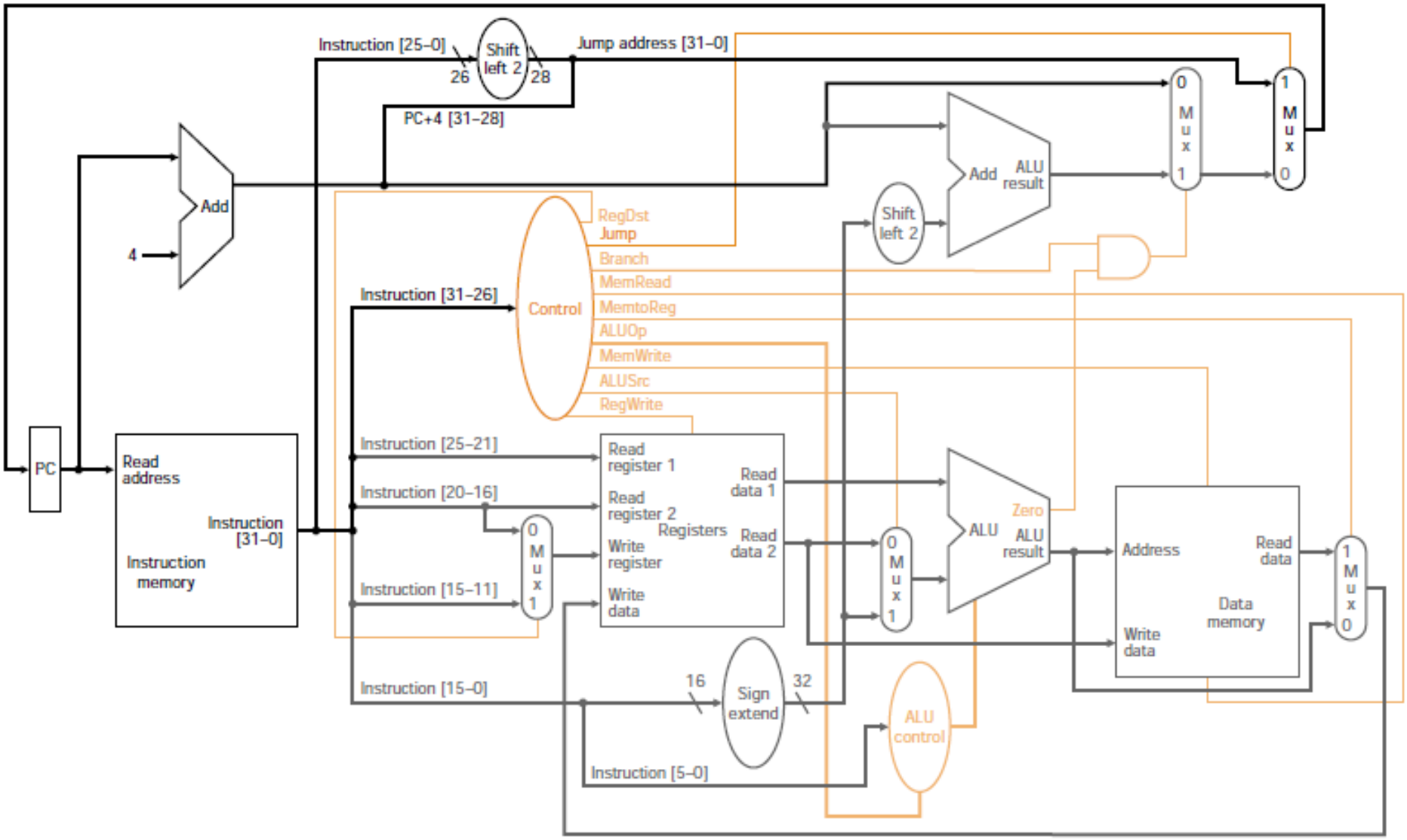
- sw and beq are the only instructions that do not write any registers.
- lw and sw are the only instructions that use the constant field. They also depend on the ALU to compute the effective memory address.
- ALUOp for R-type instructions depends on the instructions' func field.
- The PCSrc control signal (not listed) should be set if the instruction is beq *and* the ALU's Zero output is true.

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 on the previous page.
- 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.



Top level block diagram for MIPS CPU



Summary

- A datapath contains all the functional units and connections necessary to implement an instruction set architecture.
  - For our single-cycle implementation, we use two separate memories, an ALU, some extra adders, and lots of multiplexers.
  - MIPS is a 32-bit machine, so most of the buses are 32-bits wide.
- The control unit tells the datapath what to do, based on the instruction that's currently being executed.
  - Our processor has ten control signals that regulate the datapath.
  - The control signals can be generated by a combinational circuit with the instruction's 32-bit binary encoding as input.