# Computer Architecture & Assembly Language 14:332:331

## Lecture 6

## The Processor

**Naghmeh Karimi**

**Fall 16**

Adapted from *Computer Organization and Design*, *5th Edition*, Patterson & Hennessy, © 2013, Elsevier, *and Computer Organization and Design*, *4th Edition*, Patterson & Hennessy, © 2008, Elsevier and Mary Jane Irwin's slides from Penn State University.

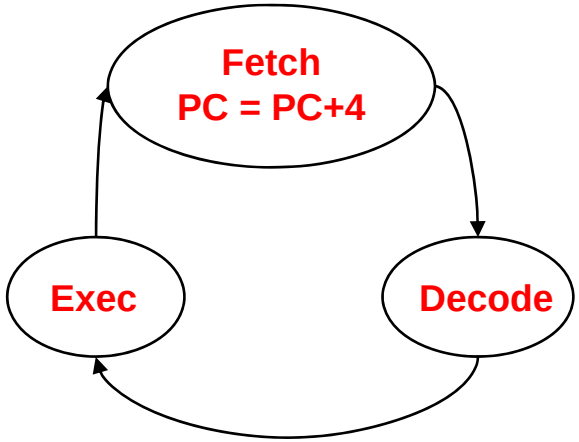
# Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

# The Processor: Datapath & Control

- Generic implementation:
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction

- All instructions (except `j`) use the ALU after reading the registers

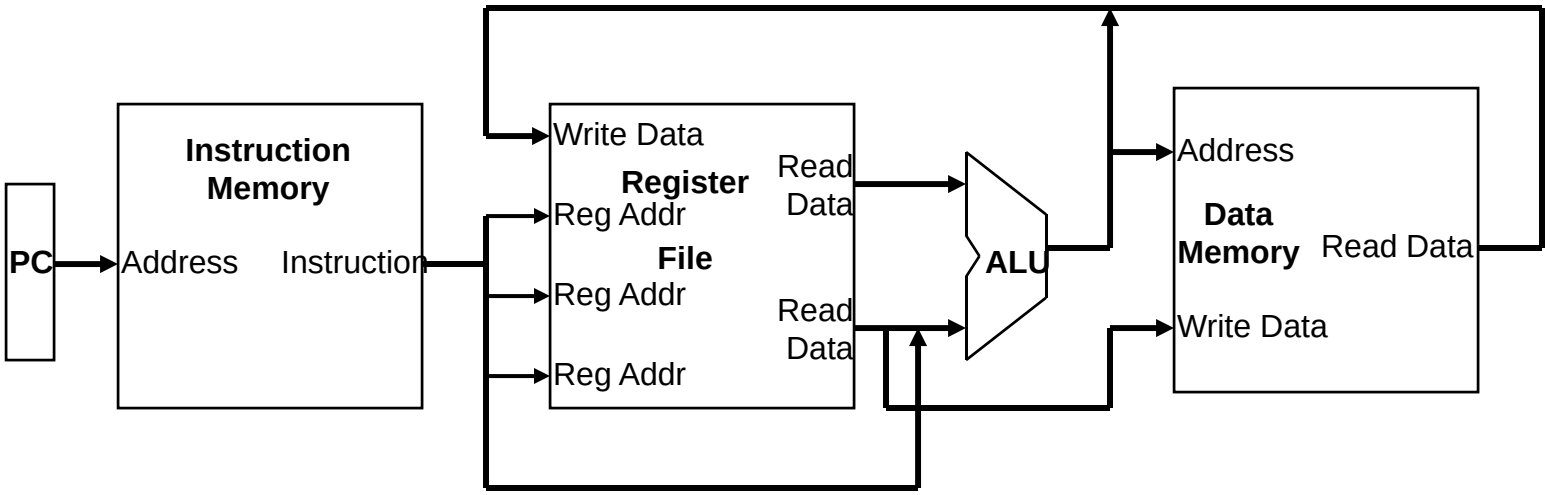


# Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → 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 ← target address or PC + 4

# Abstract Implementation View

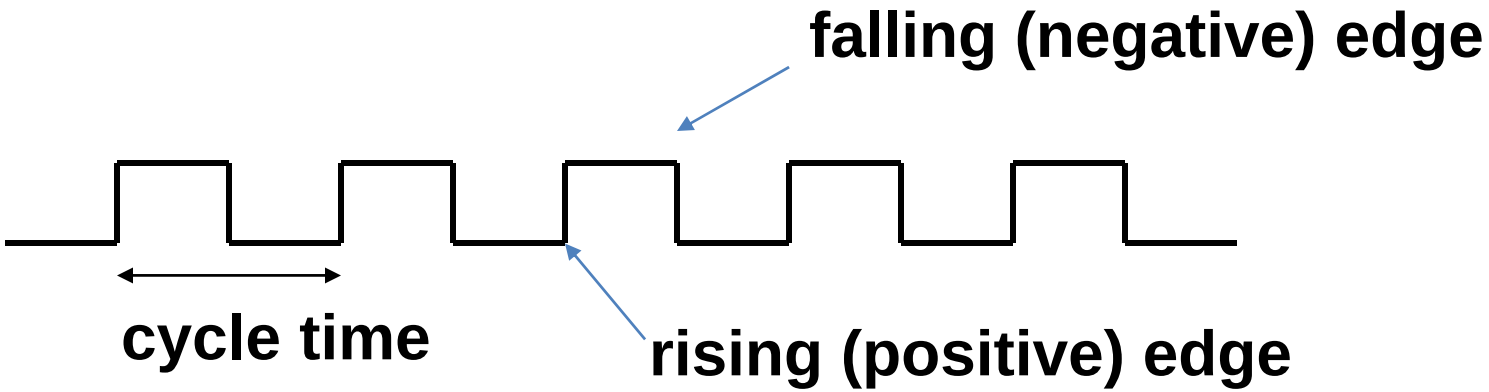
- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)




- Single cycle operation
- Split memory (Harvard) model - one memory for instructions and one for data

# Clocking Methodologies

- Clocking methodology defines when signals can be read and when they can be written




clock rate = 1/(cycle time)

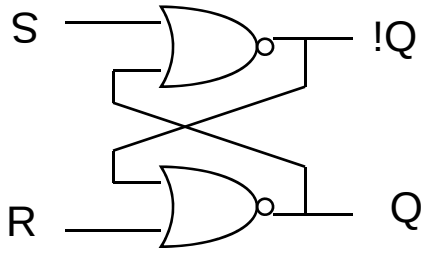
e.g., 10 nsec cycle time = 100 MHz clock rate

1 nsec cycle time = 1 GHz clock rate

- State element design choices
  - level sensitive latch
  - master-slave and edge-triggered flipflops

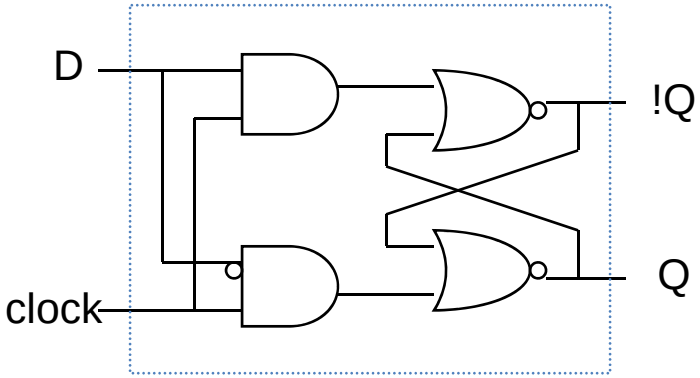
# Review: State Elements

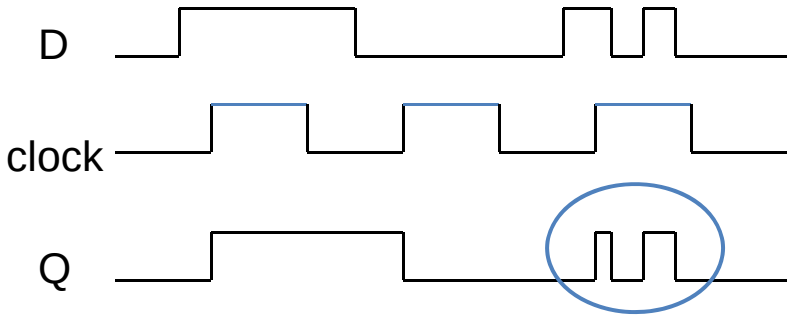
- Set-reset latch




| S | R | Q(t+1) | !Q(t+1) |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 0 | 0 | Q(t) | !Q(t) |
| 1 | 1 | 0 | 0 |

- Level sensitive D latch

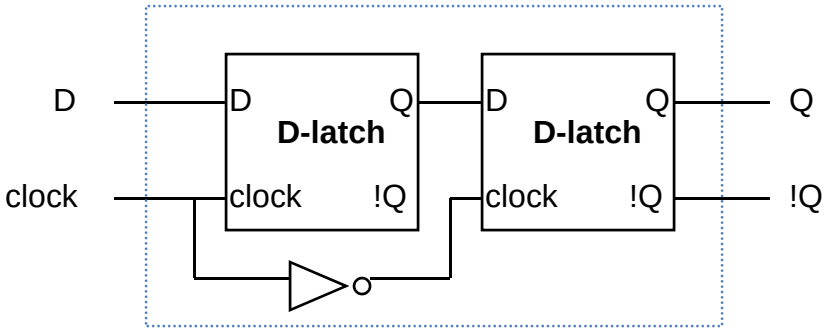


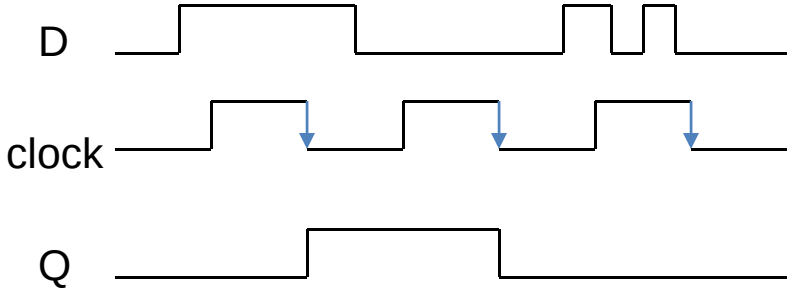




  - latch is transparent when clock is high (copies input to output)

# Review: State Elements, con't

- Solution is to use flipflops that change state (Q) only on clock edge (master-slave)



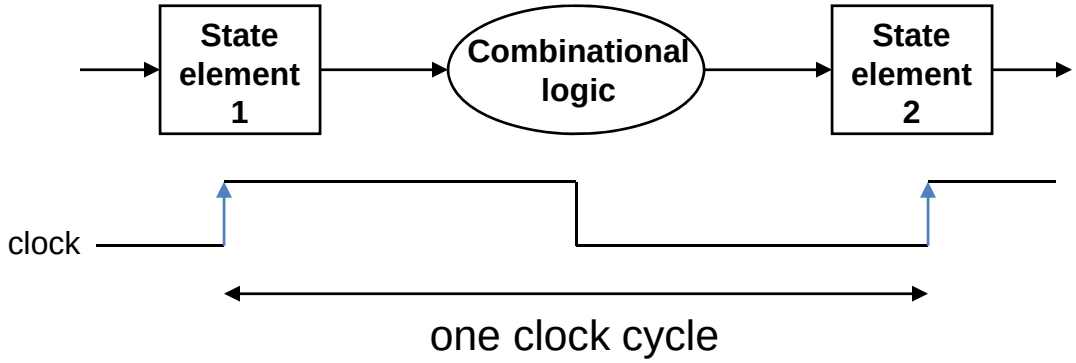

– Master (first D-latch) copies the input when the clock is high (the slave (second D-latch) is locked in its memory state and the output does not change)

– Slave copies the master when the clock goes low (the master is now locked in its memory state so changes at the input are not loaded into the master D-latch)

– Must have the clock cycle time long enough to accommodate the worst case delay path

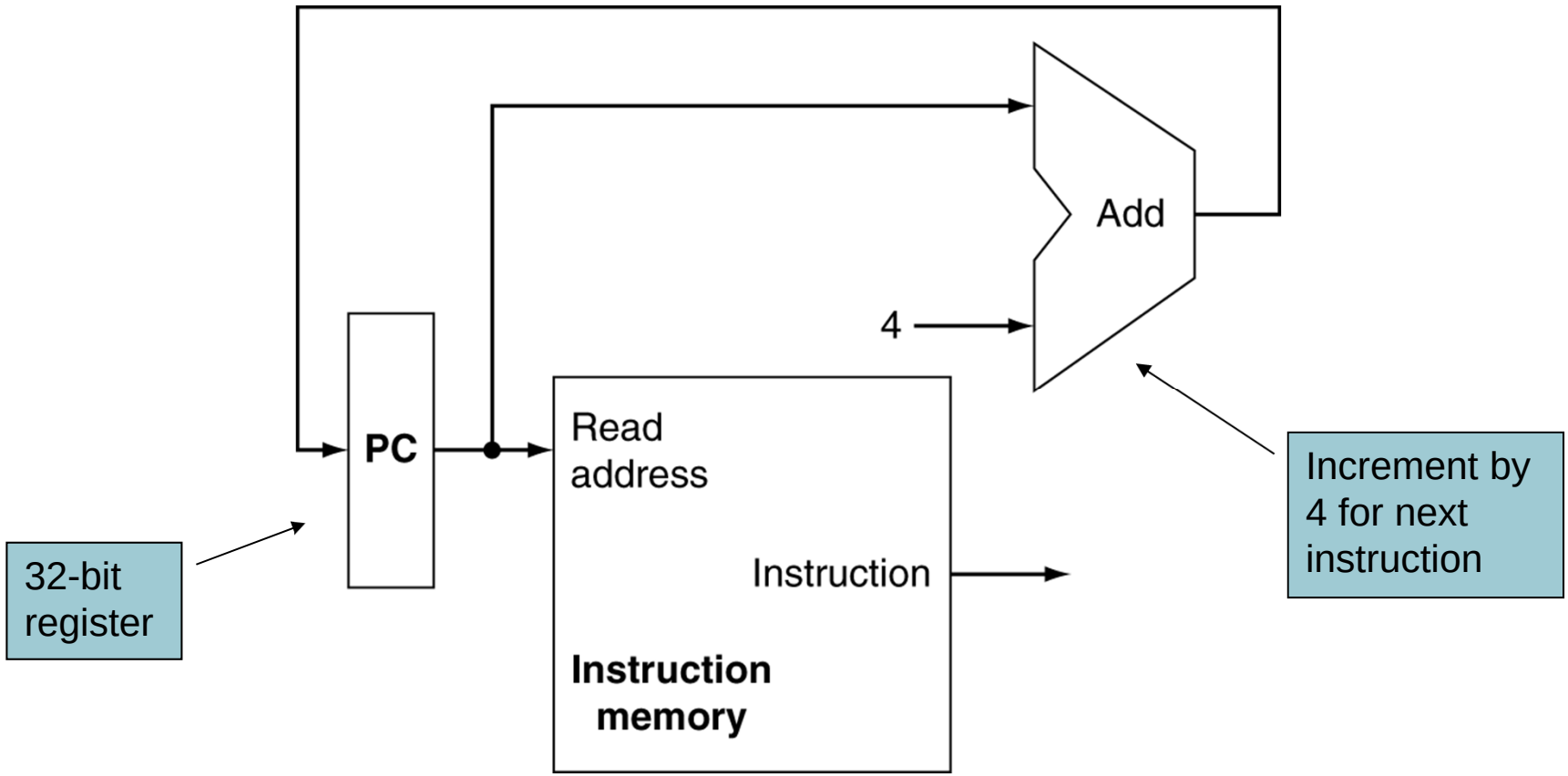
# Our Implementation

- An edge-triggered methodology
- Typical execution
  - read contents of some state elements
  - send values through some combinational logic
  - write results to one or more state elements



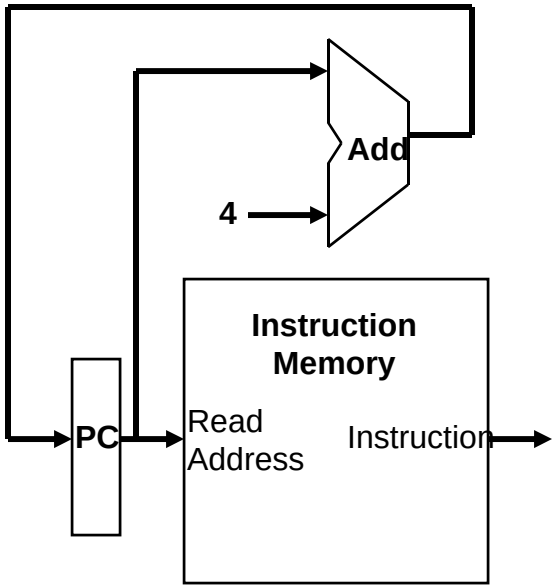

- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs

# Instruction Fetch



# Fetching Instructions

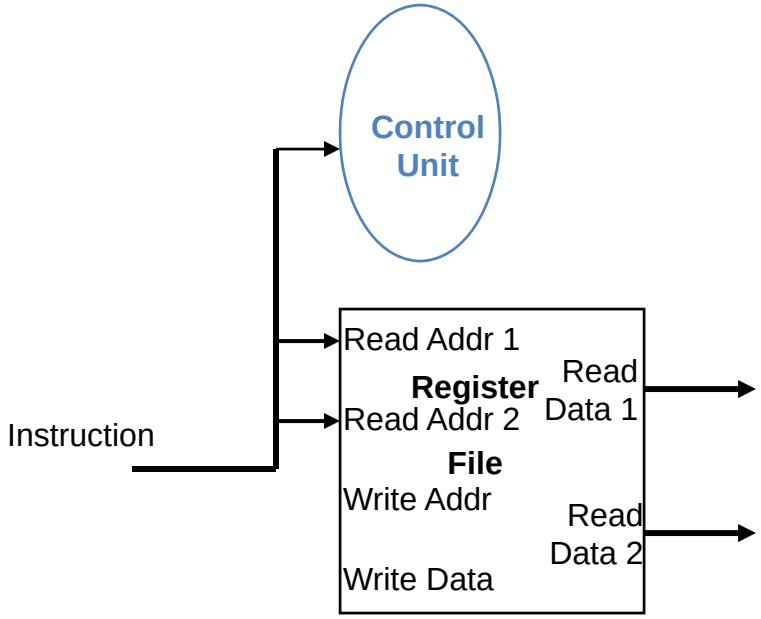
- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC to hold the address of the next instruction




  - PC is updated every cycle, so it does not need an explicit write control signal
  - Instruction Memory is read every cycle, so it doesn't need an explicit read control signal

# Decoding Instructions

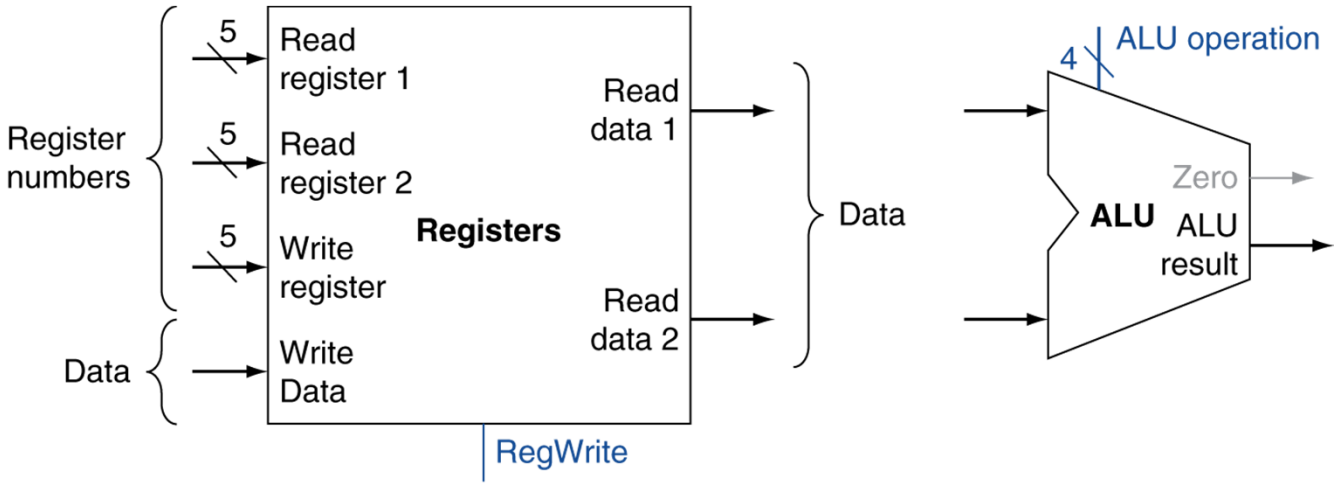
- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit




- **reading two values from the Register File**
  - **Register File addresses are contained in the instruction**

# R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation (indicated by op and funct)
- Write register result

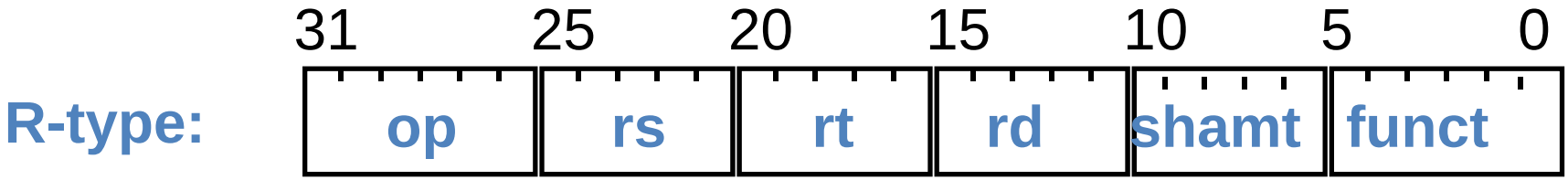



a. Registers

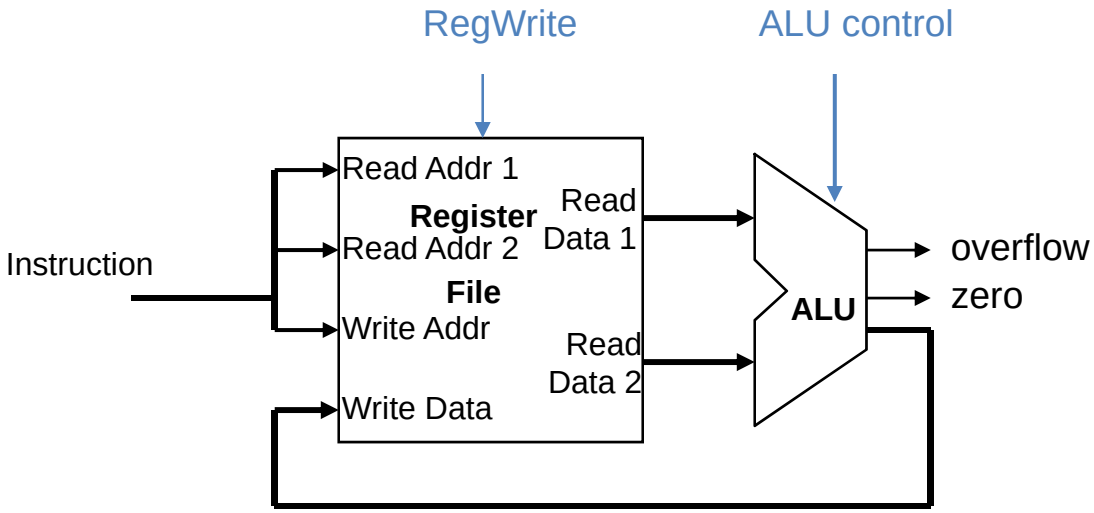
b. ALU

# Executing R Format Operations

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



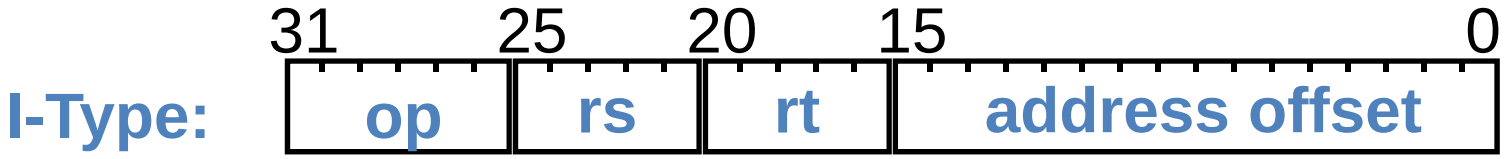

  - perform the indicated (by op and funct) operation on values in rs and rt
  - store the result back into the Register File (into location rd)




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

# Executing Load/Store Operations

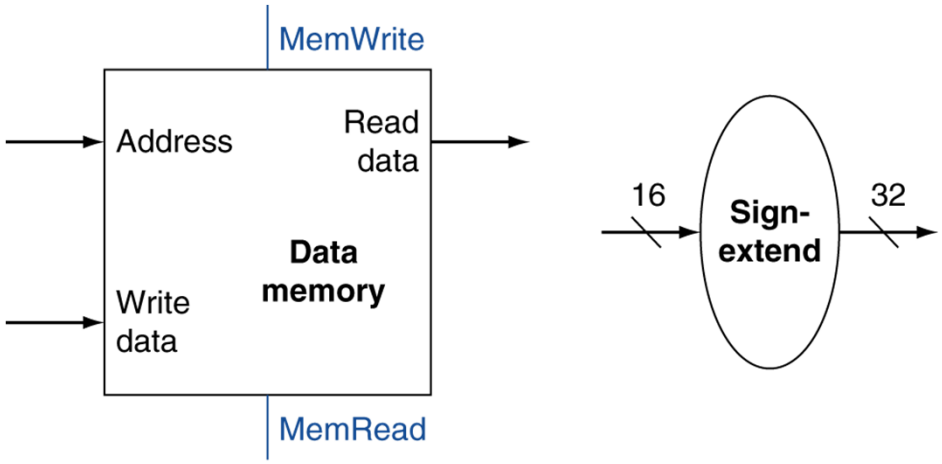
- Load and store operations




  - compute a memory address by adding the base register (in rs) to the 16-bit signed offset field in the instruction
    - base register was read from the Register File during decode
    - offset value in the low order 16 bits of the instruction must be sign extended to create a 32-bit signed value
  - store value, read from the Register File during decode, must be written to the Data Memory
  - load value, read from the Data Memory, must be stored in the Register File

# Load/Store Instructions

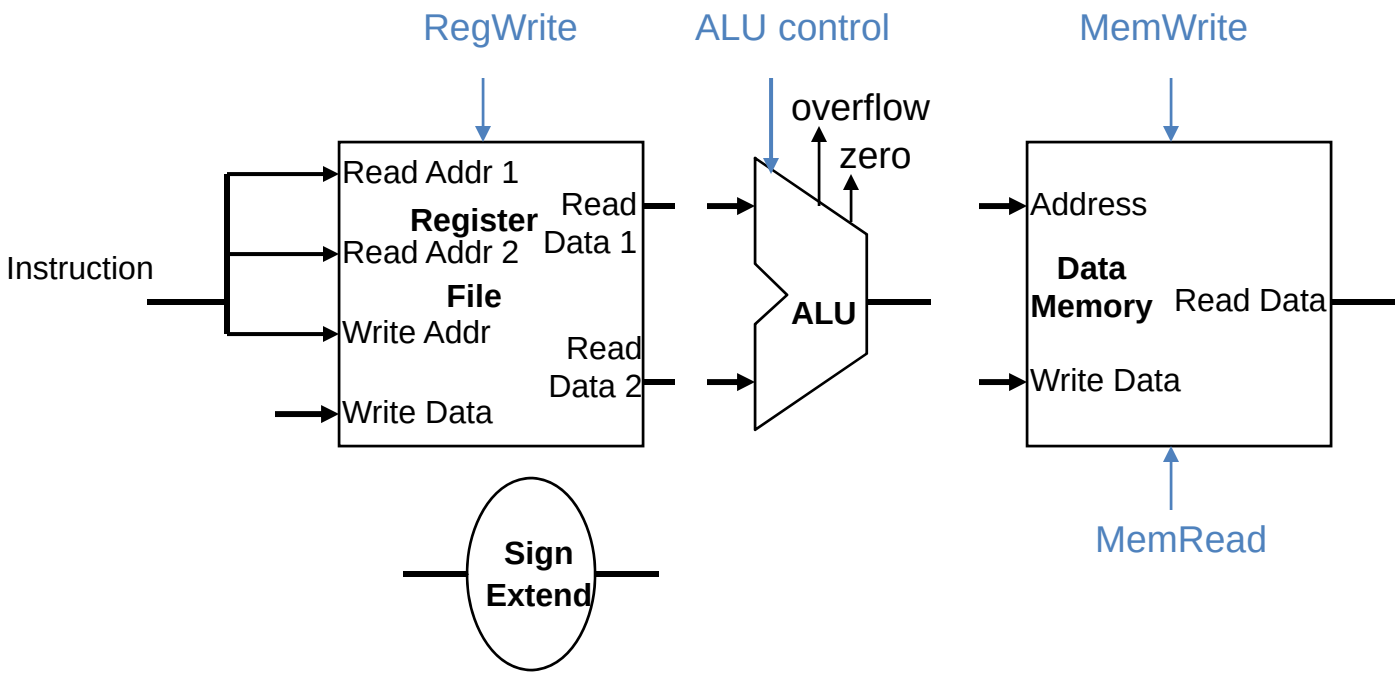
- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory




a. Data memory unit

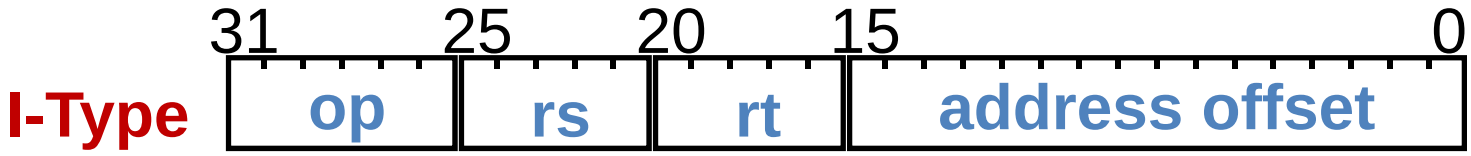
b. Sign extension unit

# Executing Load/Store Operations, con'd



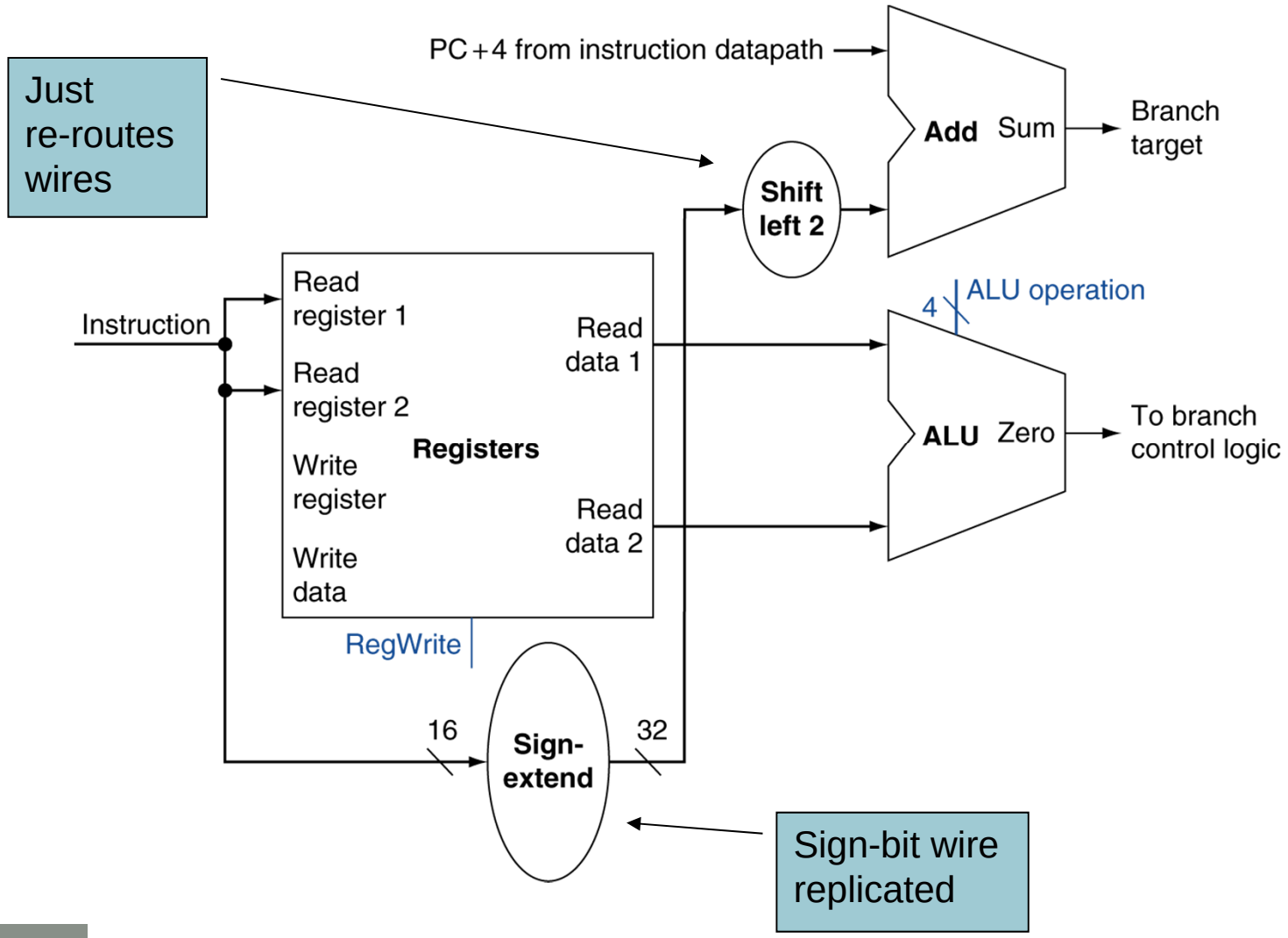
# Executing Branch Operations

- Branch operations have to

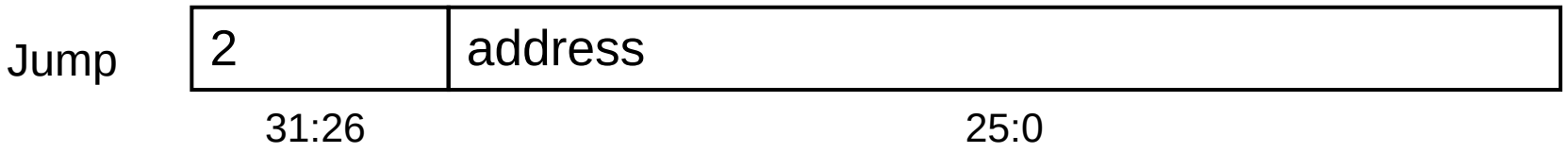



  - compare the operands read from the Register File during decode (rs and rt values) for equality (`zero` ALU output)
  - compute the branch target address by adding the updated PC to the sign extended16-bit signed offset field in the instruction
    - "base register" is the updated PC
    - offset value in the low order 16 bits of the instruction must be sign extended to create a 32-bit signed value and then shifted left 2 bits to turn it into a word address

# Branch Instructions

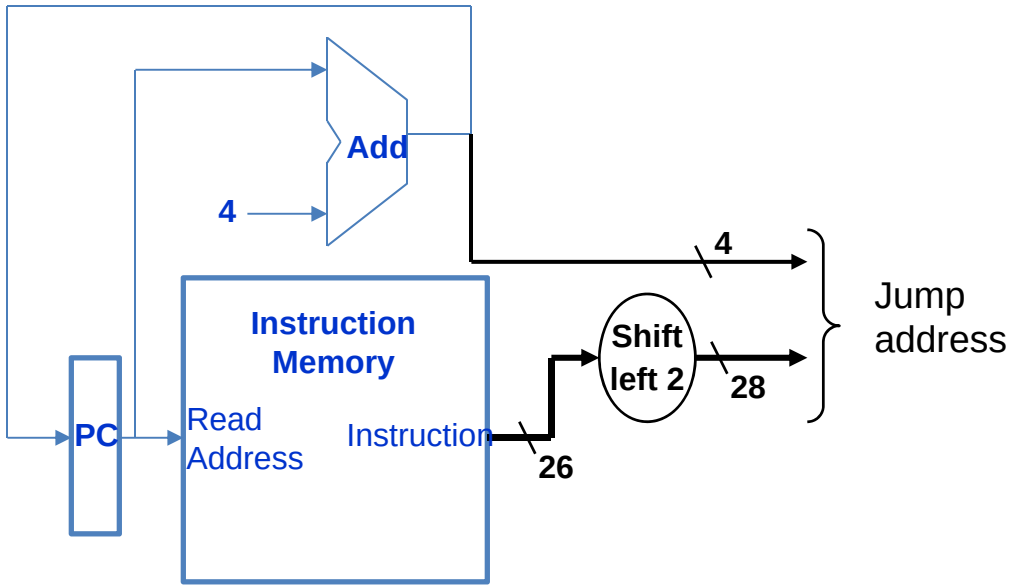


# Implementing Jumps




- 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

# Executing Jump Operations

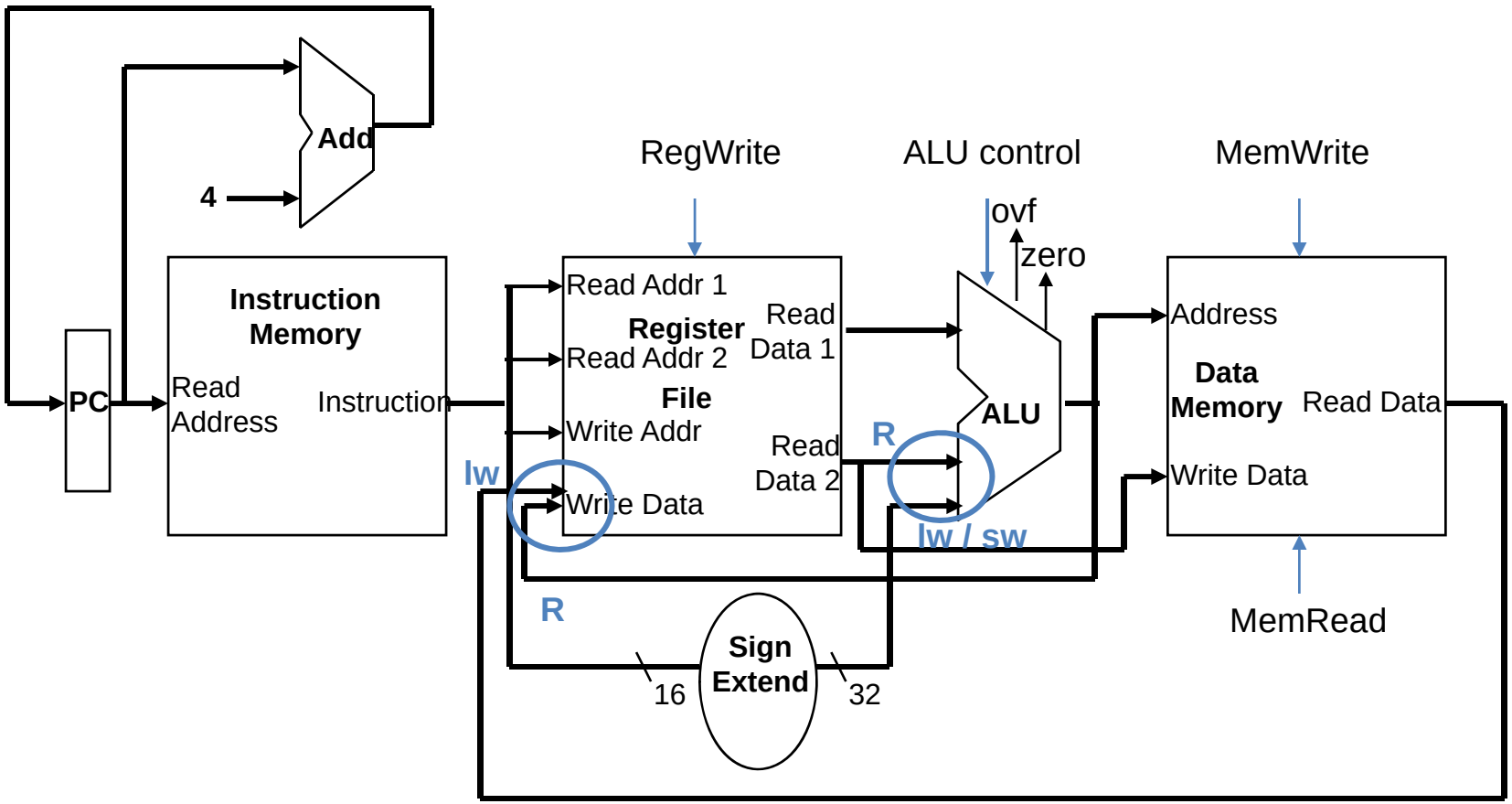


# Our Simple Control Structure

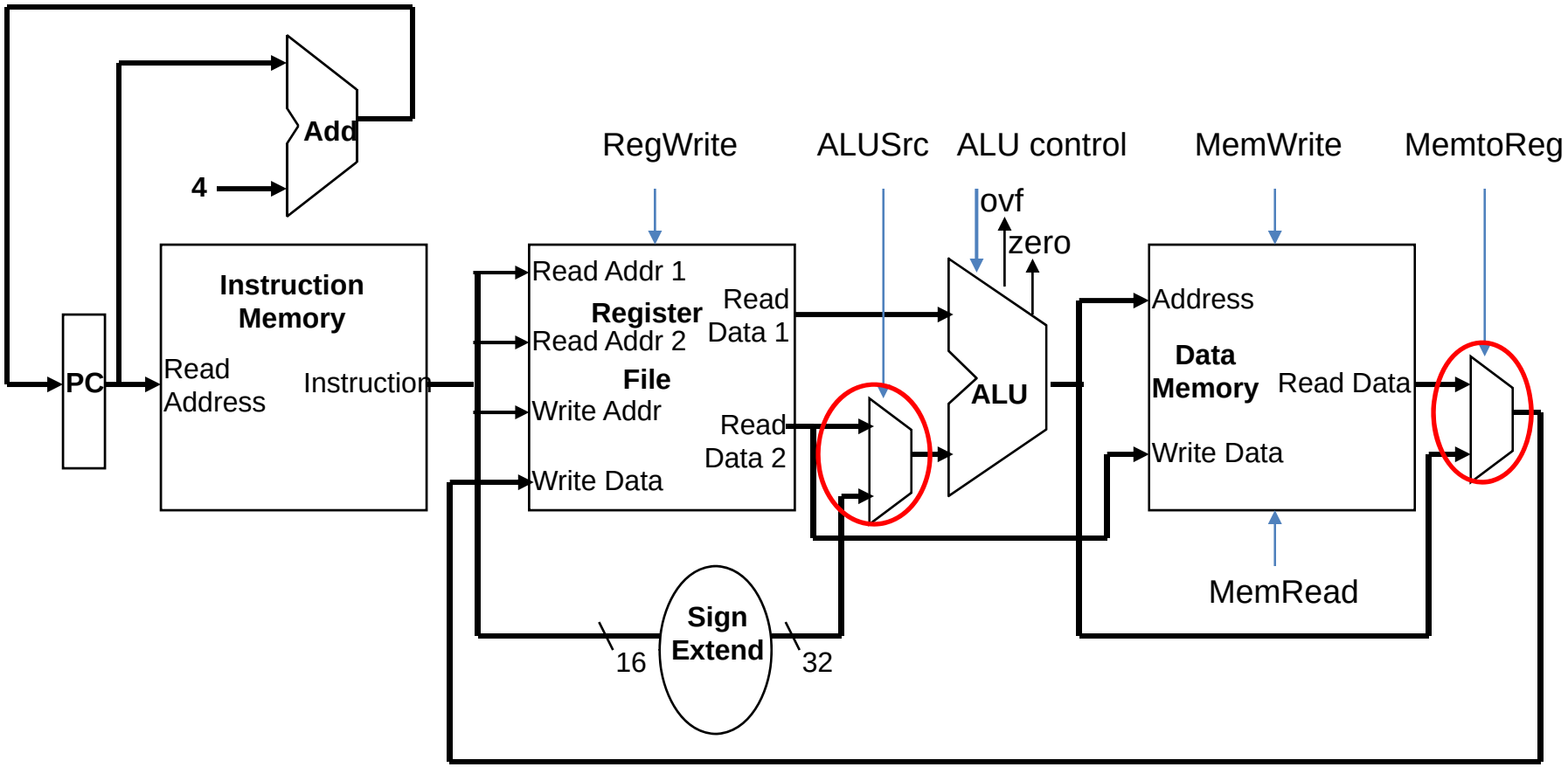
- We wait for everything to settle down
  - ALU might not produce “right answer” right away
  - we use write signals along with the clock edge to determine when to write (to the Register File and the Data Memory)
- Cycle time determined by length of the longest path

We are ignoring some details like register setup and hold times

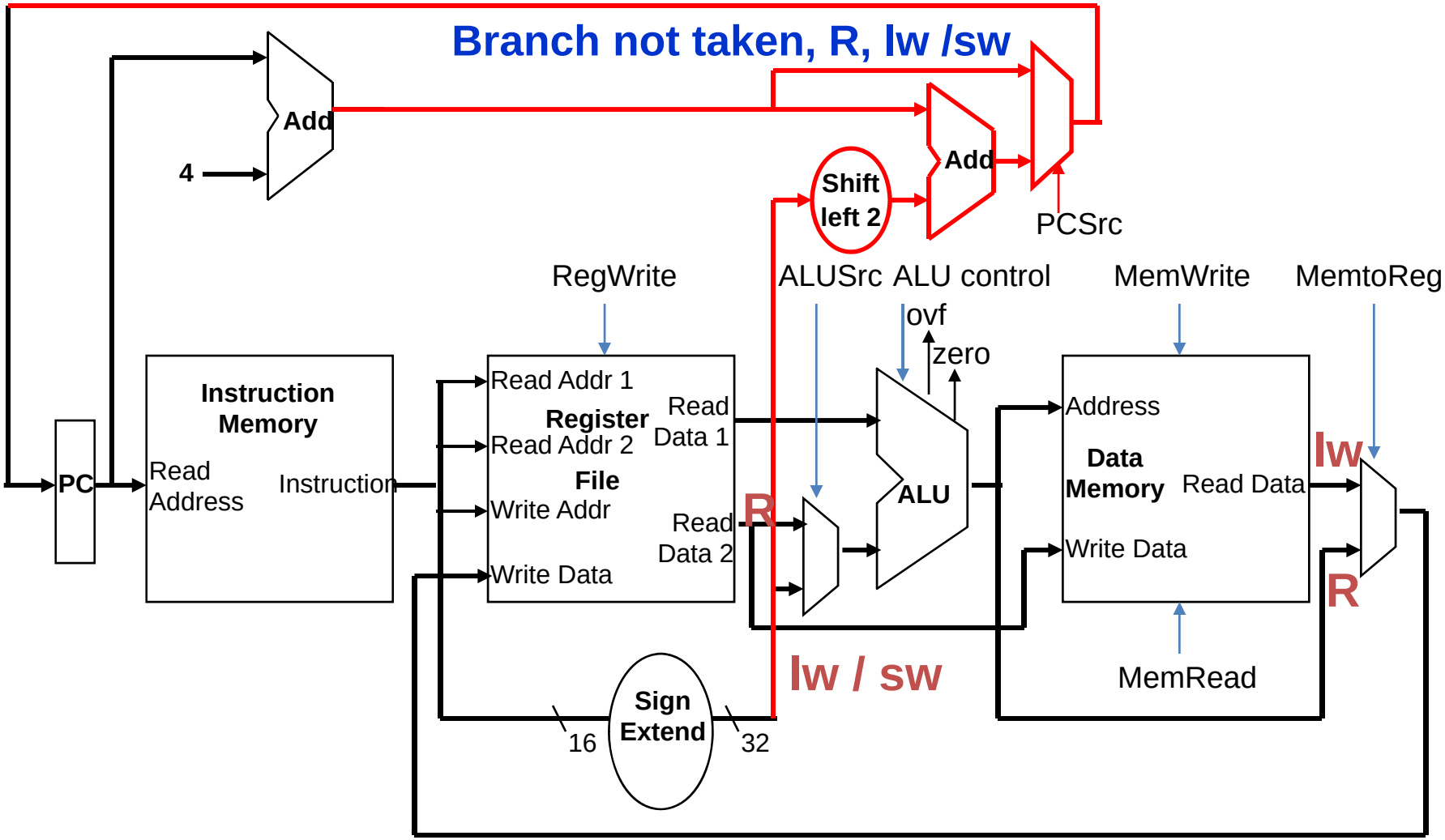
# Fetch, R, and Memory Access Portions



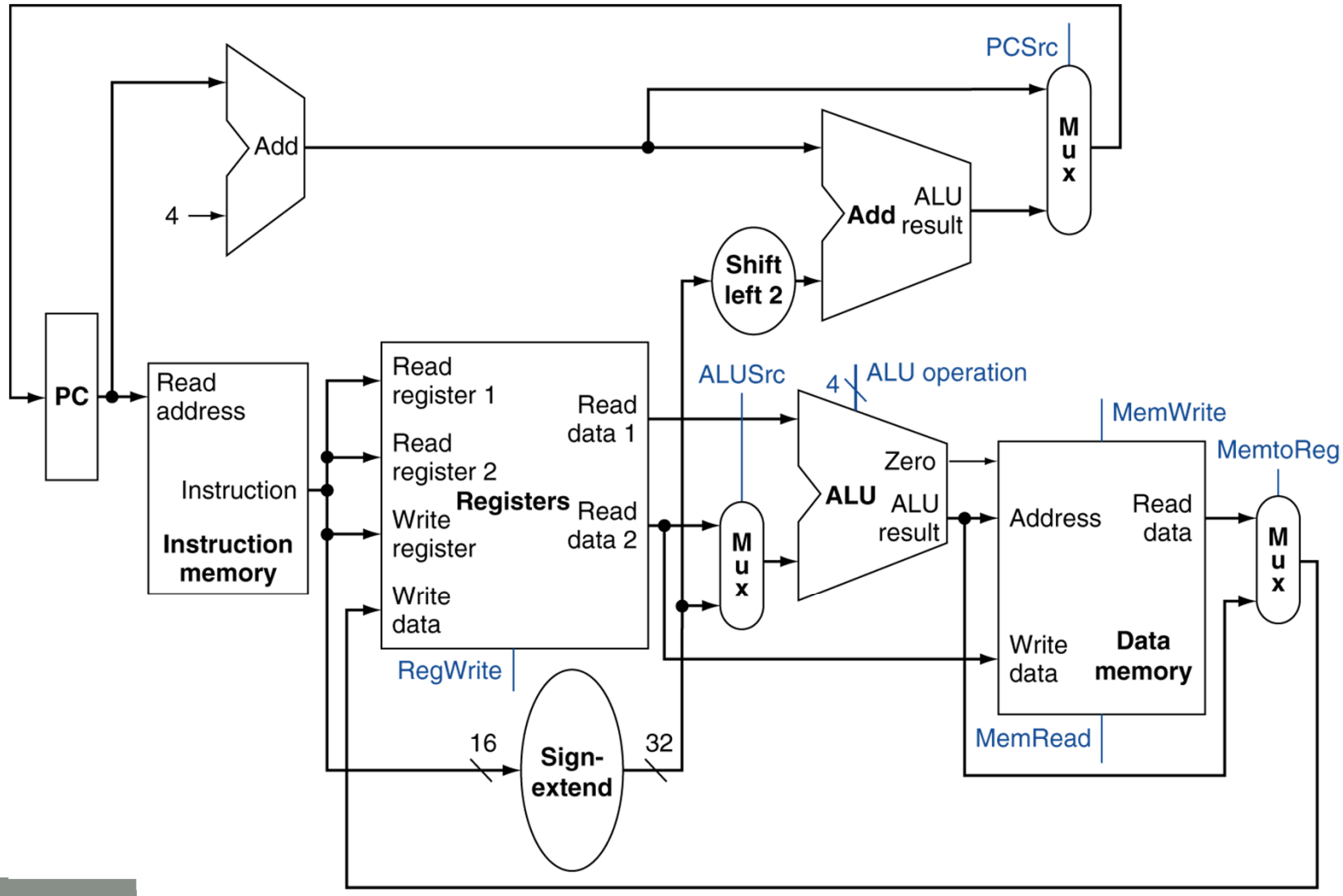
# Multiplexor Insertion



# Adding the Branch Portion



# Full Datapath



# Review

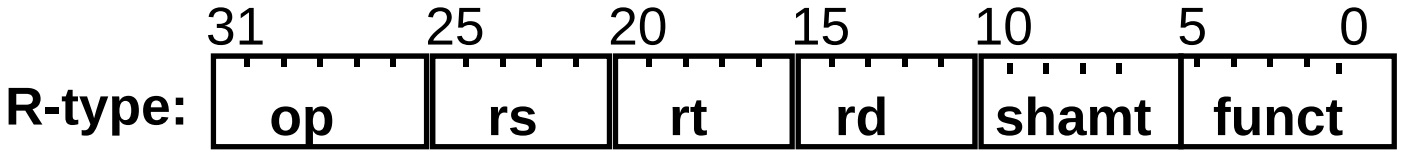
- Split memory (Harvard) model - single cycle operation
- Simplified to contain only the instructions:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Sequential components (PC, RegFile, Memory) are edge triggered
  - state elements are written on every clock cycle; if not, need explicit write control signal
    - write occurs only when **both** the write control is asserted and the clock edge occurs

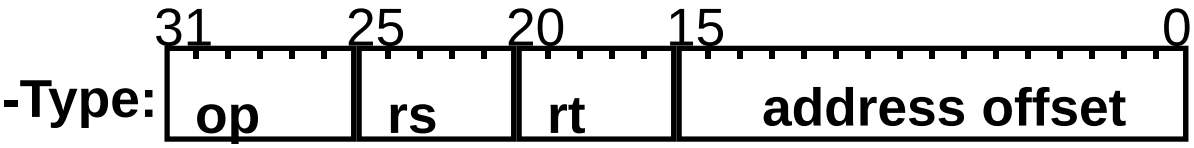
# Creating a Single Datapath from the Parts

- Assemble the datapath segments, add control lines as needed, and design the control path
- Fetch, decode and execute each instructions in one clock cycle – single cycle design
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., why we have a separate Instruction Memory and Data Memory)
  - to share datapath elements between two different instruction classes will need multiplexors at the input of the shared elements with control lines to do the selection
- Cycle time is determined by length of the longest path

# Adding the Control

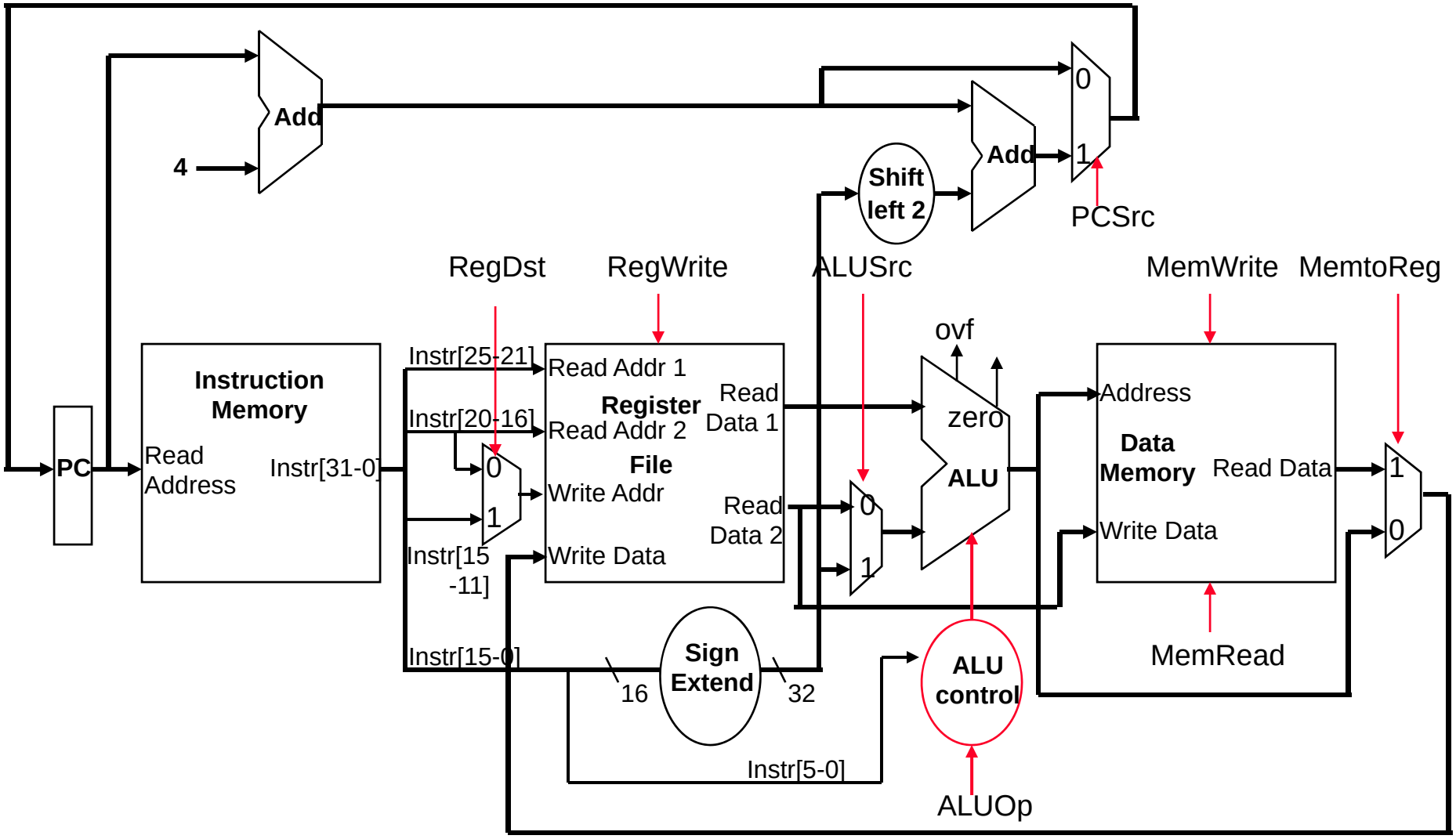
- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction



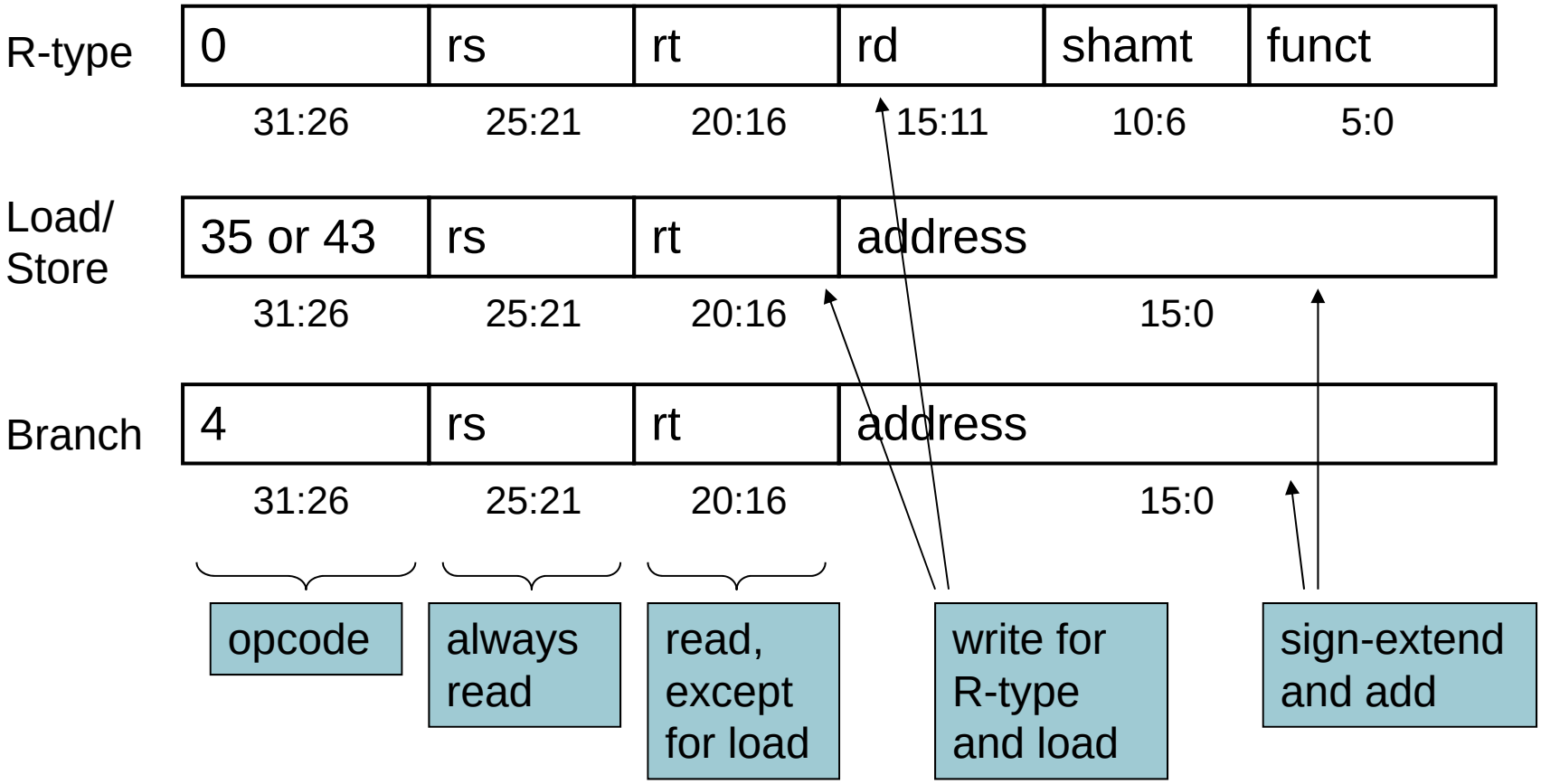

- **Observations**
  - op field always in bits 31-26
  - addr of two registers to be read are **always** specified by the **rs** and **rt** fields (bits 25-21 and 20-16)
  - addr. of register to be written is in one of **two** places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions
  - base register for lw and sw always in rs (bits 25-21)
  - offset for beq, lw, and sw always in bits 15-0

# (Almost) Complete Single Cycle Datapath

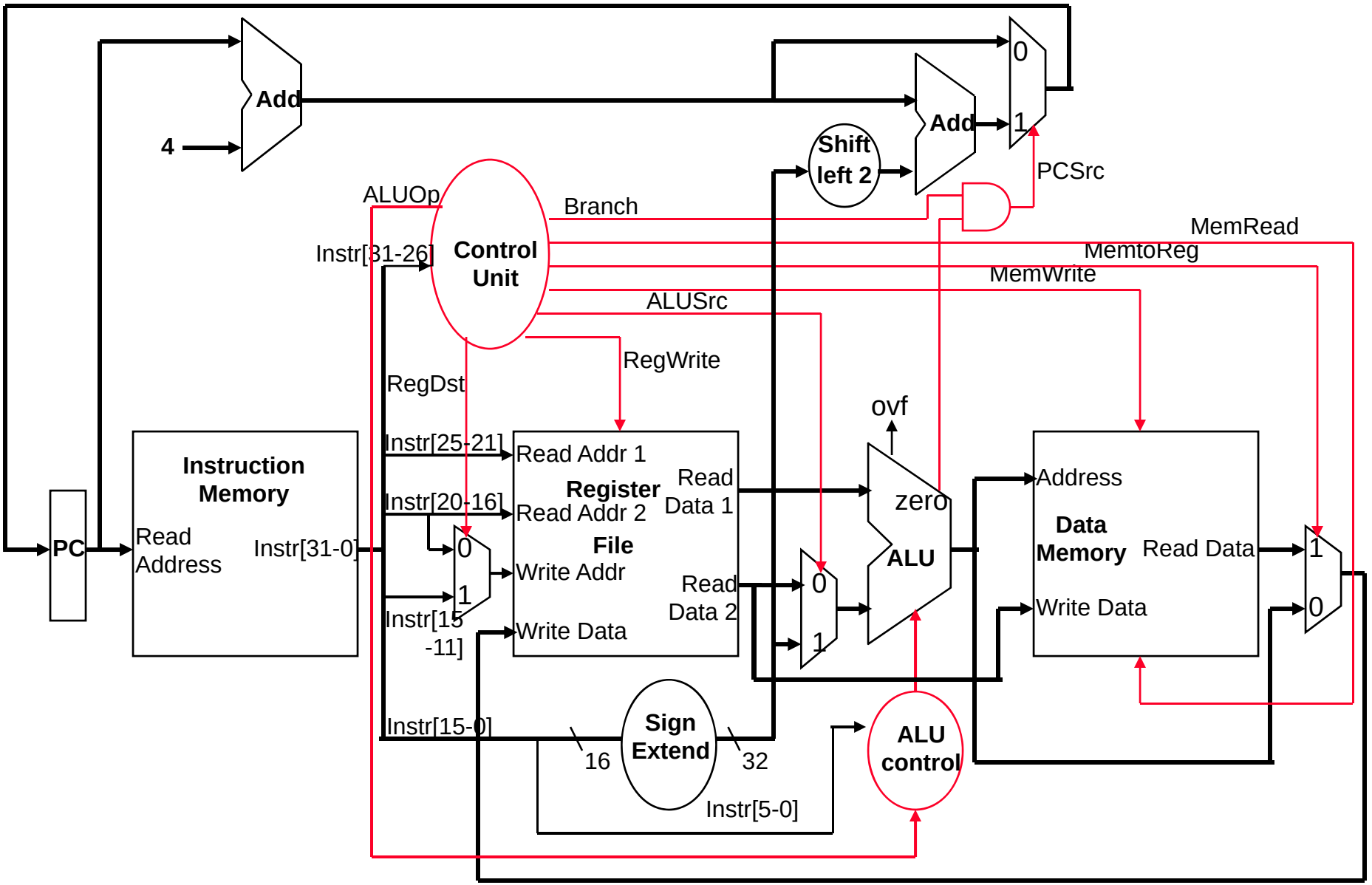


# The Main Control Unit

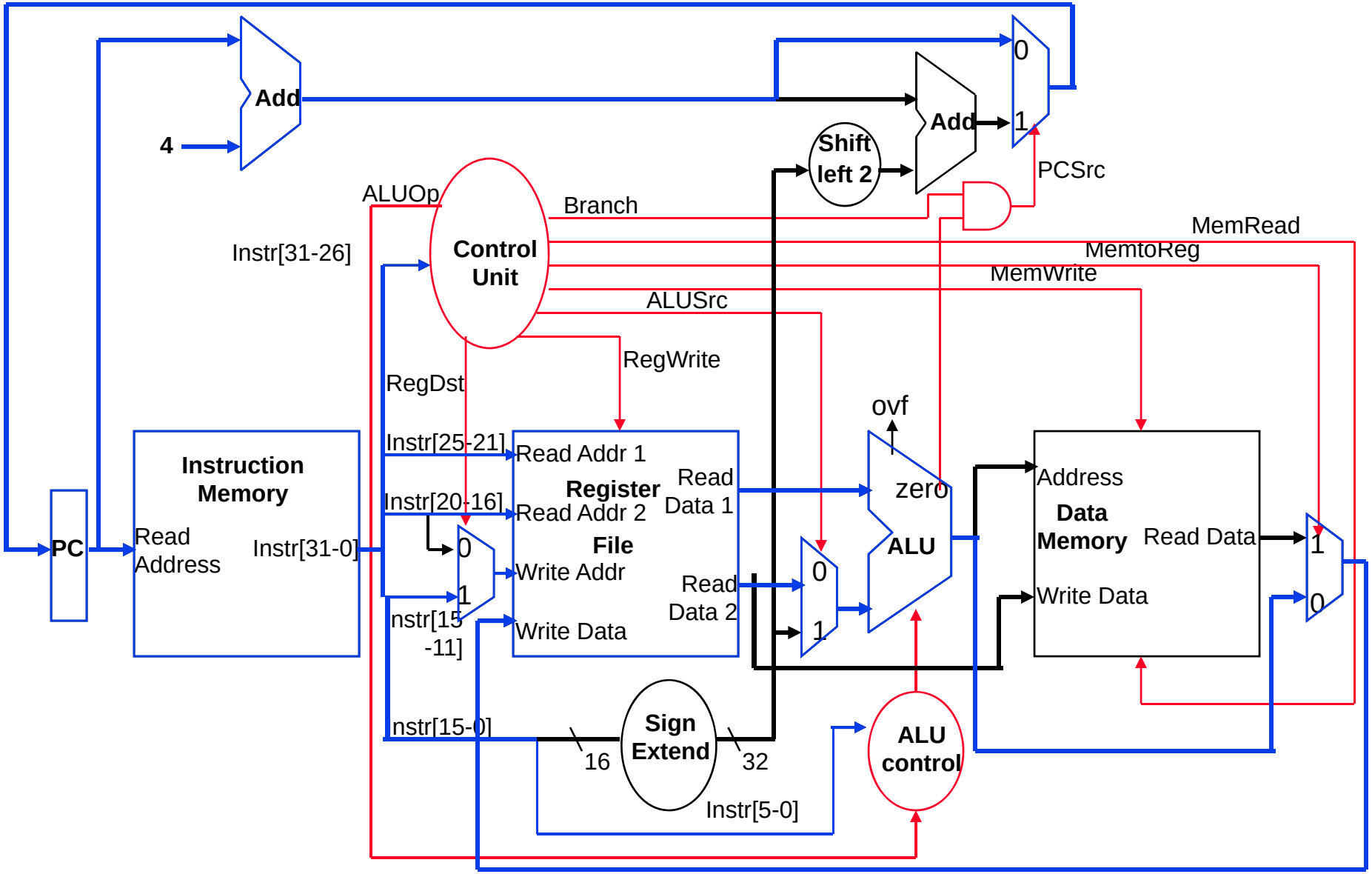
- Control signals derived from instruction



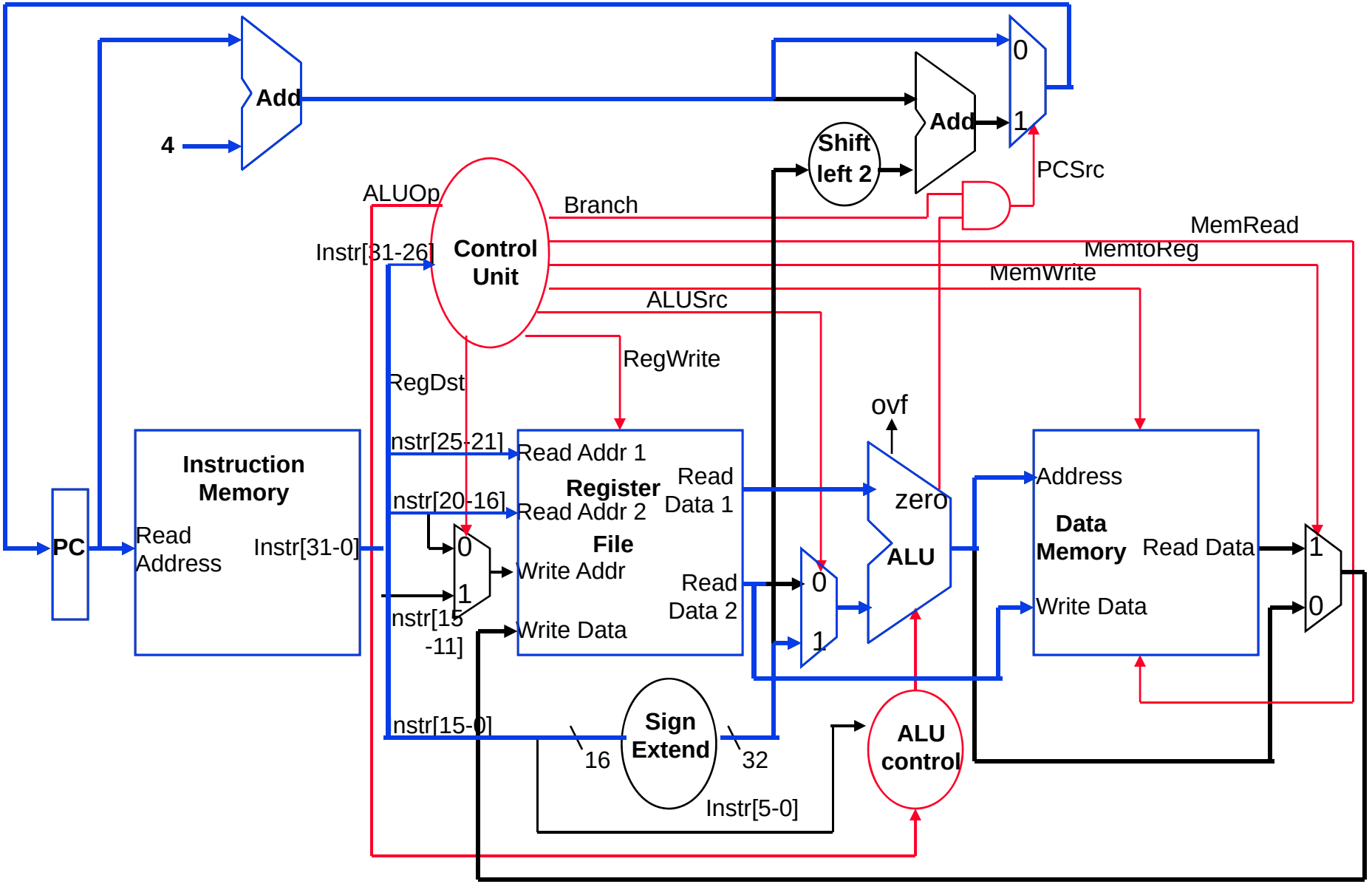
# (Almost) Complete Datapath with Control Unit



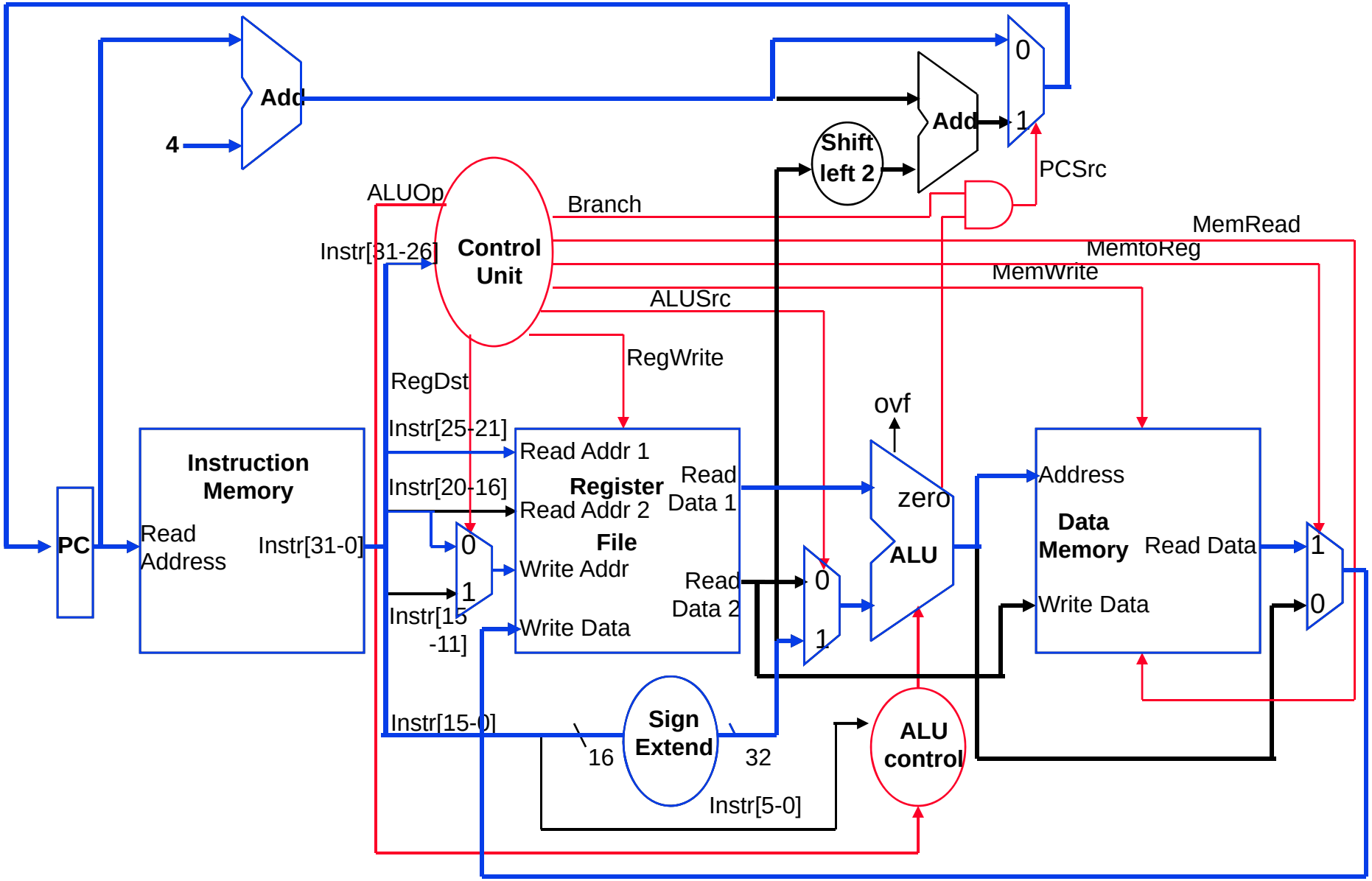
# R-type Instruction Data/Control Flow



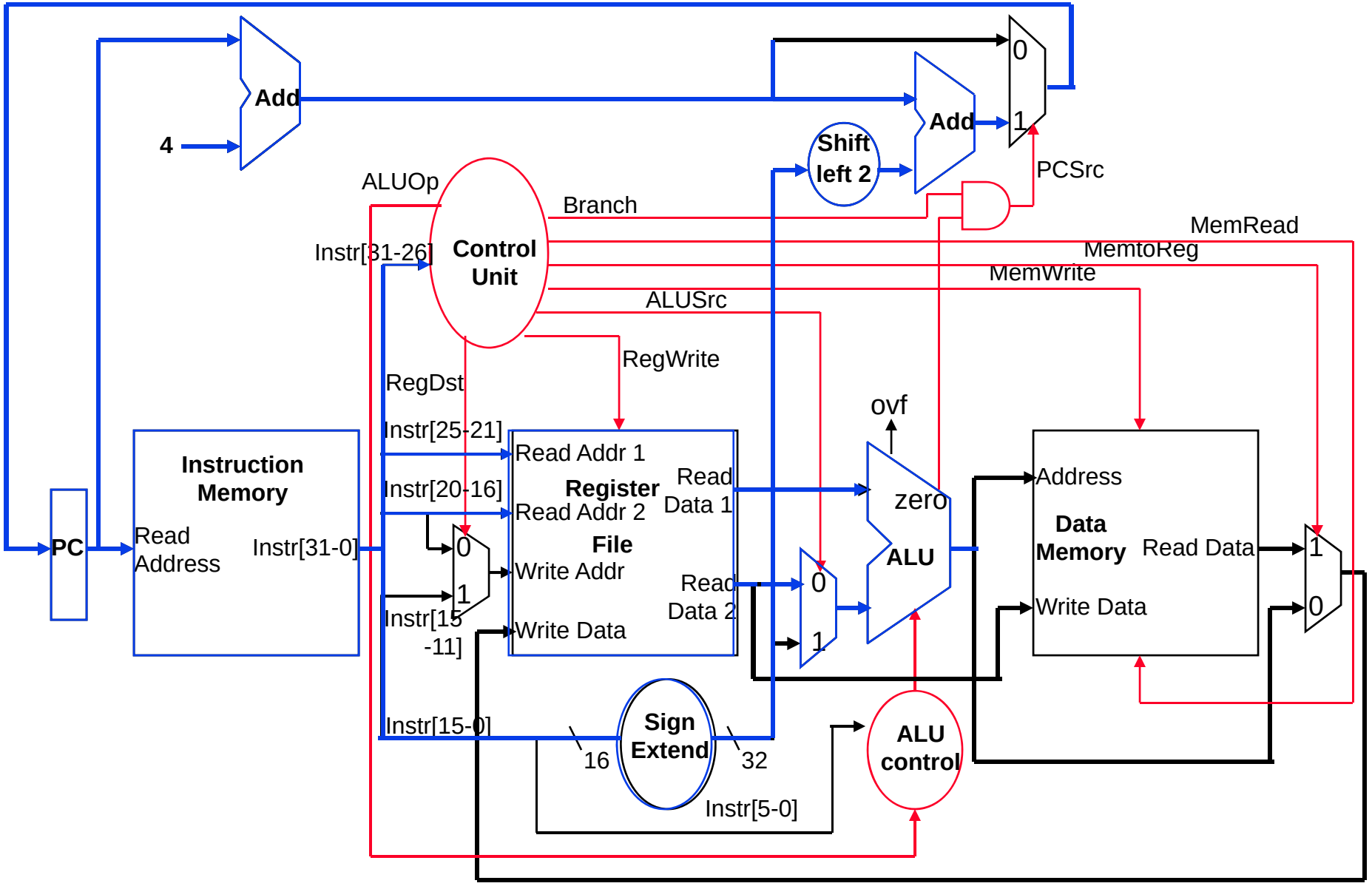
# Store Word Instruction Data/Control Flow



# Load Word Instruction Data/Control Flow



# Branch Instruction Data/Control Flow



# ALU Control



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

| ALU control | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | set-on-less-than |
| 1100 | NOR |

# ALU Control

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

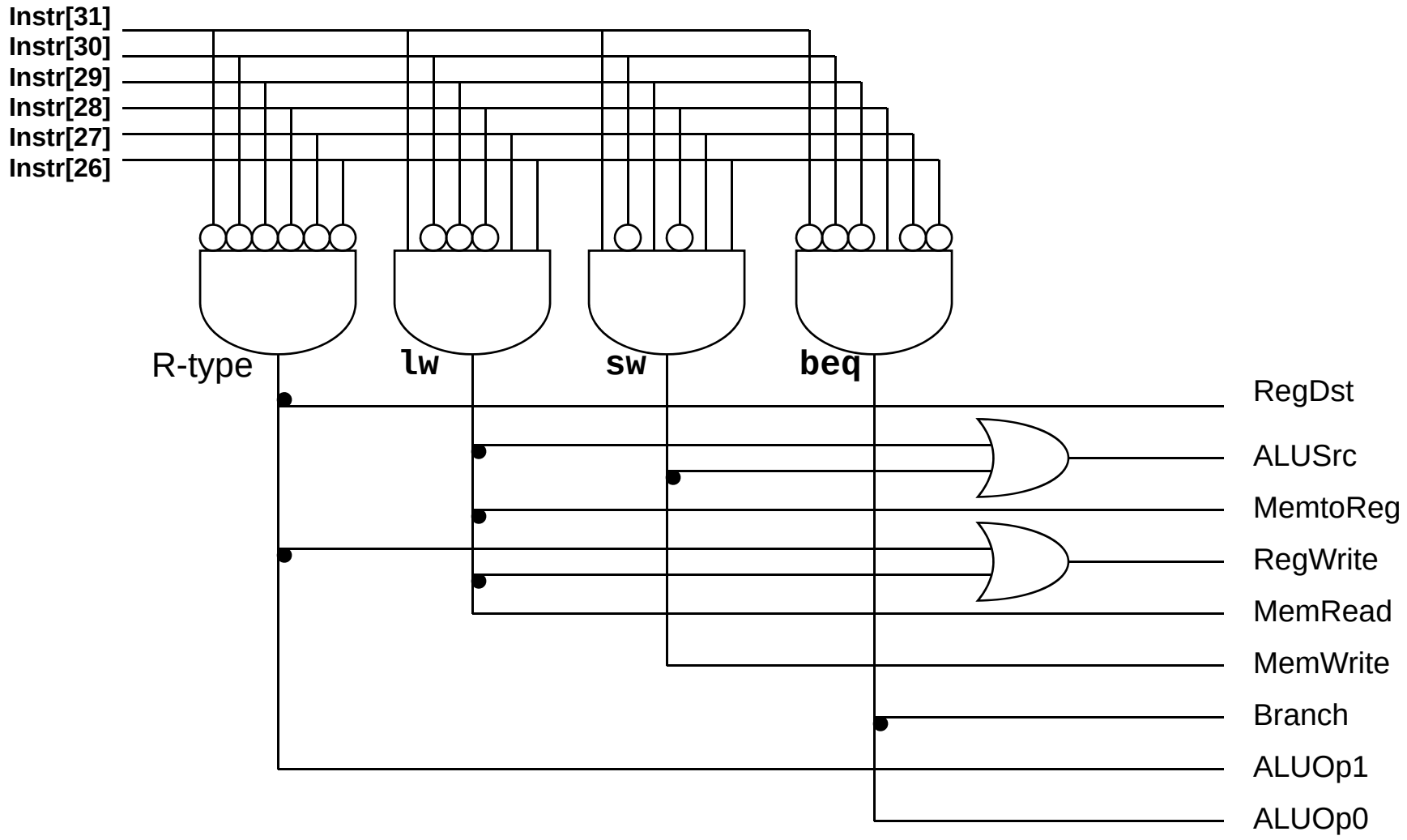
| opcode | ALUOp | Operation | funct | ALU function | ALU control |
|---|---|---|---|---|---|
| lw | 00 | load word | XXXXXX | add | 0010 |
| sw | 00 | store word | XXXXXX | add | 0010 |
| beq | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| | | subtract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

# Main Control Unit

| Instr | RegDst | ALUSrc | MemReg | RegWr | MemRd | MemWr | Branch | ALUOp1 | ALUOp0 |
|---|---|---|---|---|---|---|---|---|---|
| R-type **000000** | 1 | 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| lw **100011** | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sw **101011** | X | 1 | X | 0 | X | 1 | 0 | 0 | 0 |
| beq **000100** | X | 0 | X | 0 | X | 0 | 1 | 0 | 1 |

# Control Unit Logic

- From the truth table can design the Main Control logic

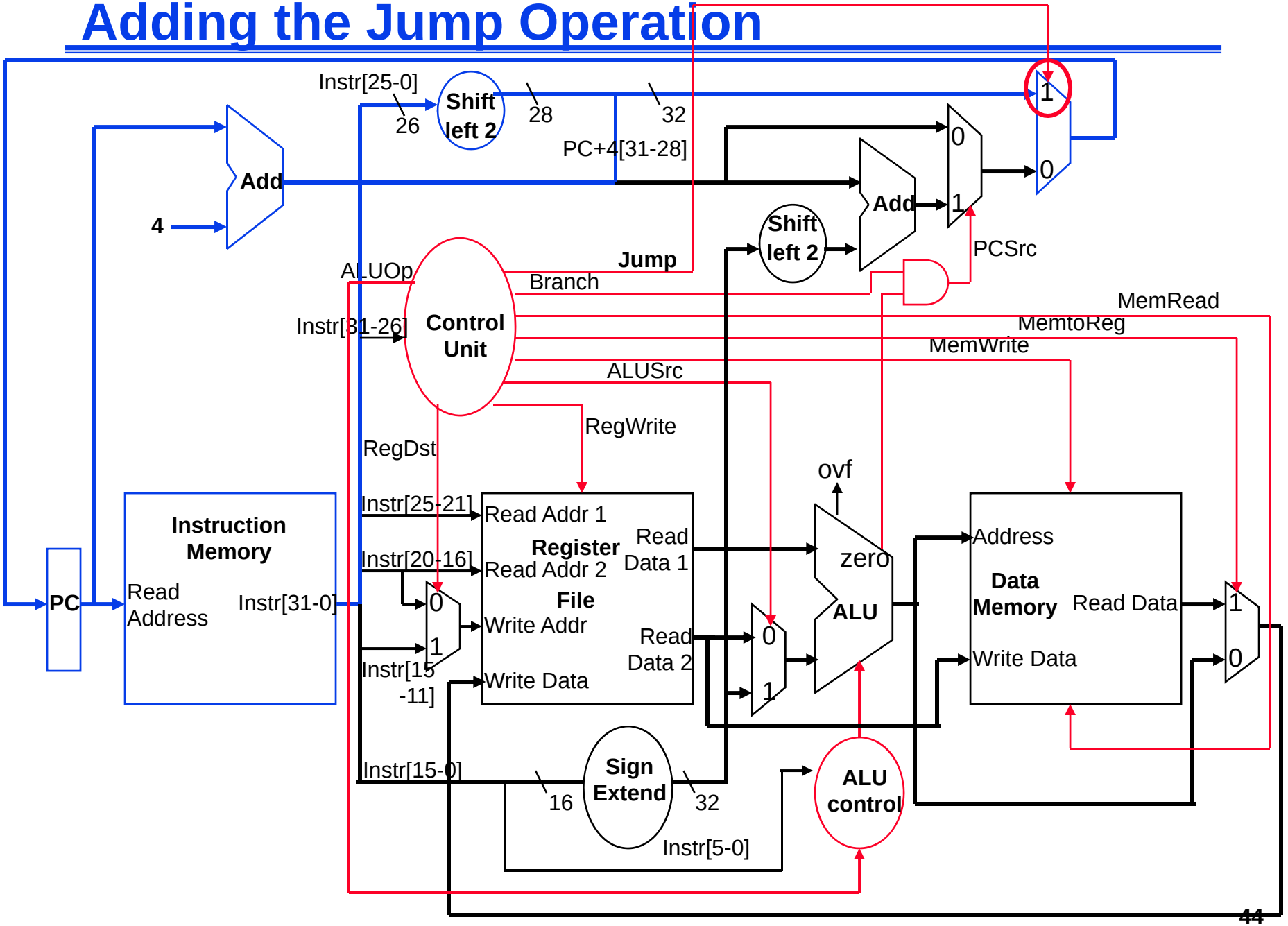


# ALU Control Truth Table

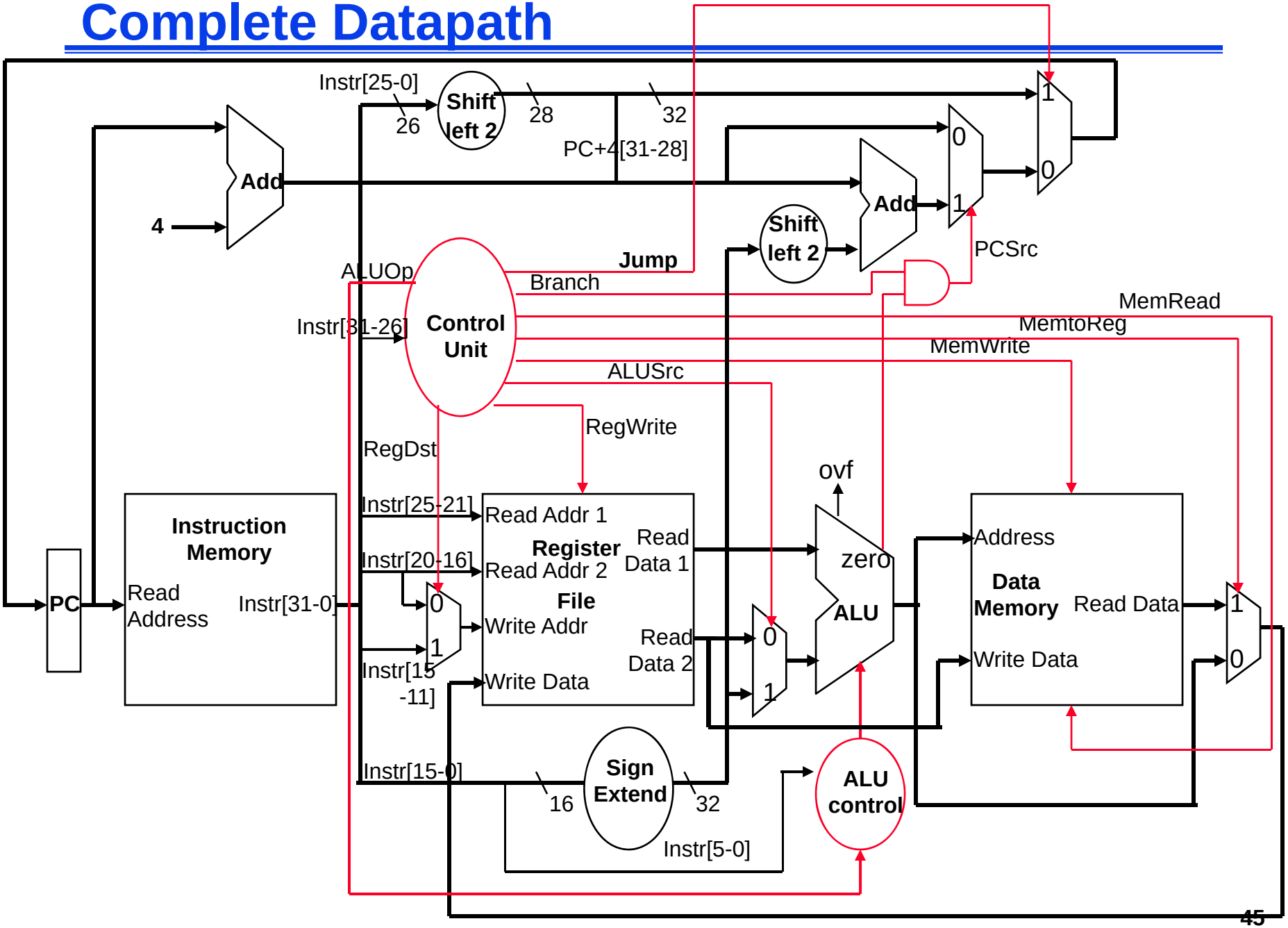
| F5 | F4 | F3 | F2 | F1 | F0 | ALUOp1 | ALUOp0 | Op2 | Op1 | Op0 |
|---|---|---|---|---|---|---|---|---|---|---|
| X | X | X | X | X | X | 0 | 0 | 0 | 1 | 0 |
| X | X | X | X | X | X | 0 | 1 | 1 | 1 | 0 |
| X | X | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| X | X | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| X | X | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| X | X | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| X | X | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

- Can make use of more don't cares
  - since ALUOp does not use the encoding 11
  - since F5 and F4 are always 10
- Logic comes from the K-maps …

# Adding the Jump Operation



# Complete Datapath



# Performance Issues

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

# Instruction Critical Paths

- Calculate cycle time assuming negligible delays (for muxes, control unit, sign extend, PC access, shift left 2, wires) except:
  - Instruction and Data Memory (2ns)
  - ALU and adders (2ns)
  - Register File access (reads or writes) (1ns)

| Instr. | I Mem | Reg Rd | ALU Op | D Mem | Reg Wr | Total |
|---|---|---|---|---|---|---|
| R-type | 2 | 1 | 2 | | 1 | 6 |
| load | 2 | 1 | 2 | 2 | 1 | 8 |
| store | 2 | 1 | 2 | 2 | | 7 |
| Beq | 2 | 1 | 2 | | | 5 |
| jump | 2 | | | | | 2 |

# Question:

- **We wish to add the instruction addi to the single-cycle datapath. Add any necessary datapaths and control signals.**

| Control Signal | Setting | Control Signal | Setting |
|---|---|---|---|
| RegDst | | ALUOp1 | |
| Jump | | AlUOp0 | |
| Branch | | MemWrite | |
| MemRead | | ALUSrc | |
| MemtoReg | | RegWrite | |