# CoE3DR4
# Computer Organization

Chapter 4

# Introduction

- Performance of a machine is determined by three factors: instruction count, clock cycle time, and clock cycles per instruction (CPI)
- The complier and ISA determine the instruction count required for a given program
- Clock cycle time and number of CPI are determined by processor implementation
- In this chapter we construct the datapath and control unit for two different implementation of MIPS instruction set

# Introduction

- The datapath is the interconnection of the components that make up the processor.
  - The datapath must provide connections for moving bits between memory, registers and the ALU.
- The control is a collection of signals that enable/disable the inputs/outputs of the various components.
- You can think of the control as the brain, and the datapath as the body.
  - the datapath does only what the brain tells it to do.
- We will implement a subset of core MIPS :
  - lw and sw
  - add, sub, and, or and slt
  - beq and j
- Effect of different implementation choices on clock rate and CPI

# Implementation overview

- Two identical first step for every instruction

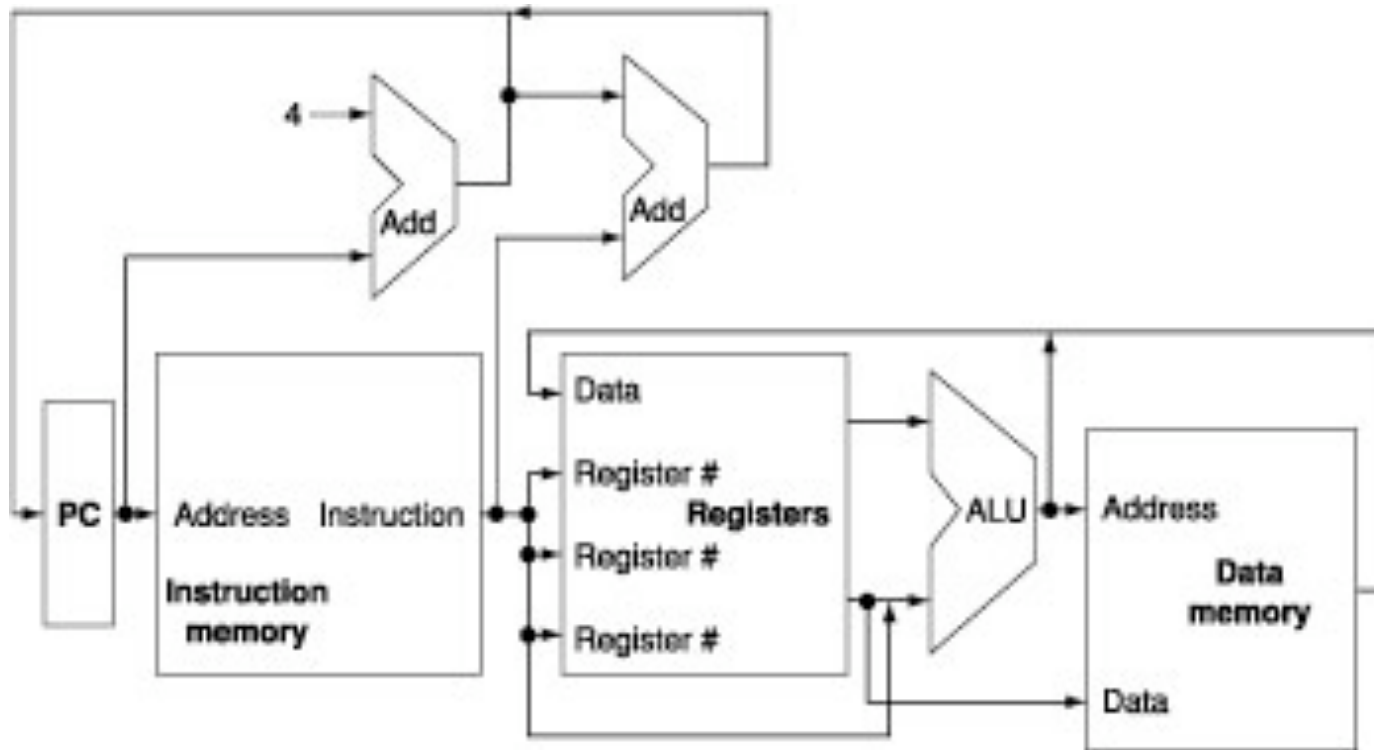
1. Send PC to memory and fetch the instruction from that location
2. Read one or two registers, selected by fields of instruction opcode (one register for lw, two for most of the other instructions)

- Perform the actions to accomplish the instruction
  - Actions are largely the same, independent of exact opcode
  - Holds for each instruction class: memory reference, arithmetic-logic, and branch

# Implementation overview

- All instruction classes (except jump) use ALU after reading registers
  - Memory reference instructions use ALU for address calculations
  - Arithmetic-logic instructions use ALU for operation execution
  - Branch instructions use ALU for comparison
- Post-ALU execution of instructions
  - Memory-reference instruction needs to access memory for load or store
  - Arithmetic-logic instruction must write data to a register
  - Branch instruction needs to change the PC for next instruction address based on comparison
- Figure 4.1
  - High level view of a MIPS implementation

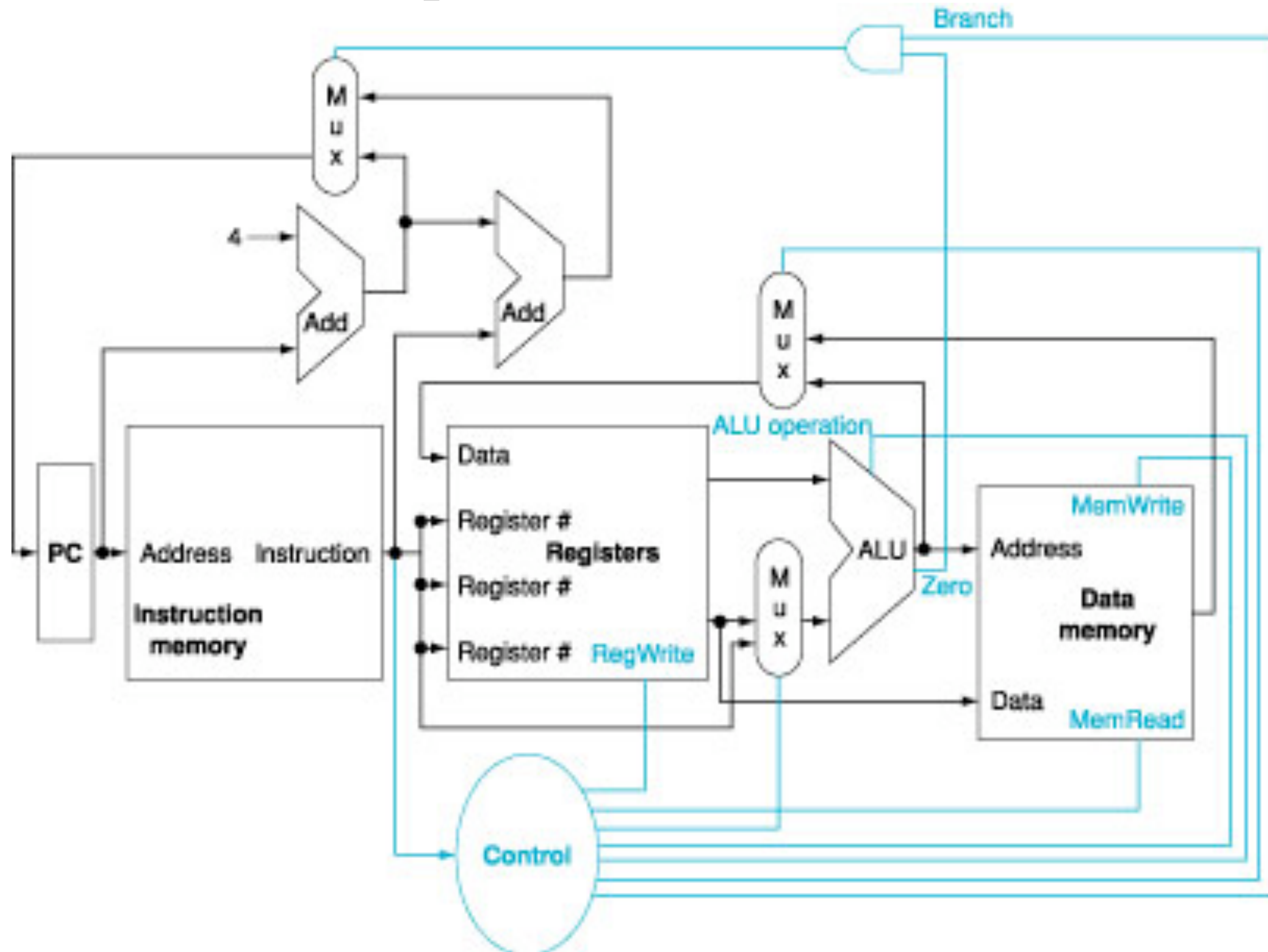
# Implementation overview



# Implementation overview

- Two important aspects of instruction execution are missing:
- 1. Data going to a particular unit is coming from two sources
  - We have to use multiplexer
  - Control lines of the multiplexers are set based on information taken from the instruction being executed
- 2. Several of the units must be controlled depending on the type of instruction
  - Example: data memory must read on a load and write on a store
  - Example: ALU must perform one of several operations
- These operations are directed by control lines that are set on the basis of various fields in the instruction

# Implementation overview



# Implementation overview

- A control unit that has the instructions as an input is used to determine how to set the control lines for the functional units and two of the multiplexers
- Third multiplexer which determines whether PC+4 or the branch destination address is written into the PC is set based on the zero output of the ALU (which is used to perform the comparison of a beq)

# Logic Design

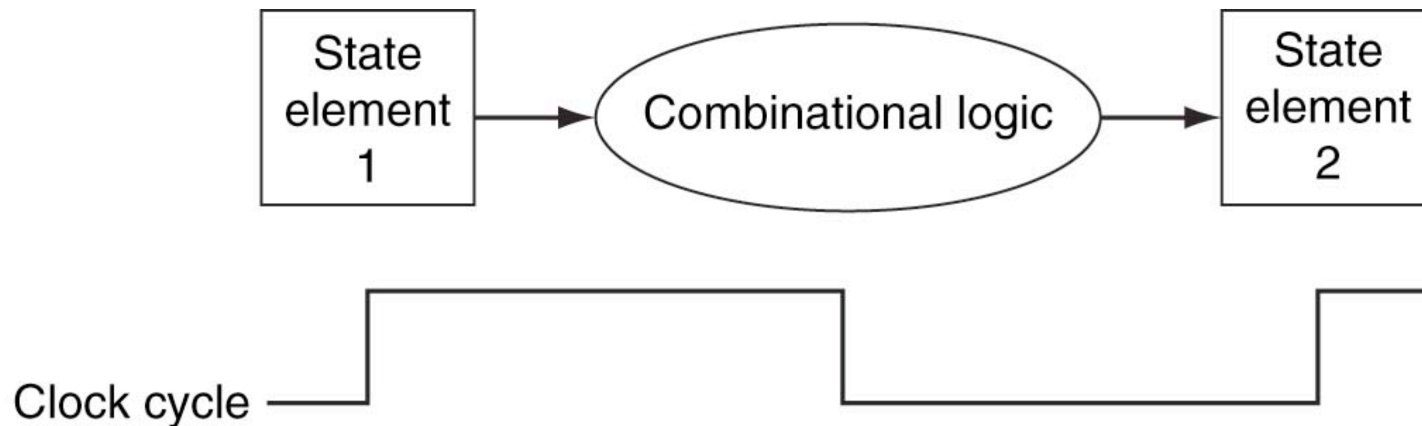
- Functional units in MIPS implementation have two different logic elements
- 1. Combinational elements
  - Elements that operate on data values
  - Output depends only on the current inputs
  - No internal storage
  - Exemplified by circuitry to perform arithmetic operations
- 2. State elements
- Elements that contain state, or internal storage
- Characterize the state of the machine
- If machine loses power but these elements can be restored, we can restart the machine from the state at which the power was lost
  - Instructions and data memories, as well as registers

# Logic Design

- State element: At least two inputs and one output
- Inputs are data value to be written into the element and the clock to determine when the data value is written
- Output is the value that was written during an earlier clock cycle
- One of the logically simplest state elements is a D-type flip flop that has exactly two inputs and one output
- Other state elements used in MIPS implementation are memories and registers
- Logic components containing state are called sequential because their output depends on both internal state and inputs

# Clocking methodology

- Defines when signals can be read or written
- Concurrency issue; read at the same time as write

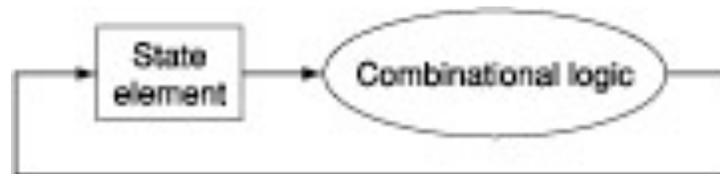


# Clocking methodology

- Edge-triggered clocking methodology
  - Any value stored in a sequential logic element are to be updated only on a clock edge
  - Any collection of combinational logic must have its inputs coming from a set of state elements and its outputs written into a set of state elements
  - Inputs are values written in a previous clock cycle
  - Outputs are values to be used in a following clock cycle
  - Length of the clock cycle is the time taken by the system to go from one element to the other
  - If a state element is not updated on every edge, an explicit write control signal is required
  - State element is updated only when the write control signal is asserted and a clock edge occurs

# Clocking methodology

- Edge-triggered clocking methodology: Read, operate, and write can be achieved in the same clock cycle
- No feedback within a single clock cycle
- All state and logic elements have 32-bit wide inputs and outputs
  – Buses are signals wider than 1 bit
  – Several buses can be combined to make a wider bus

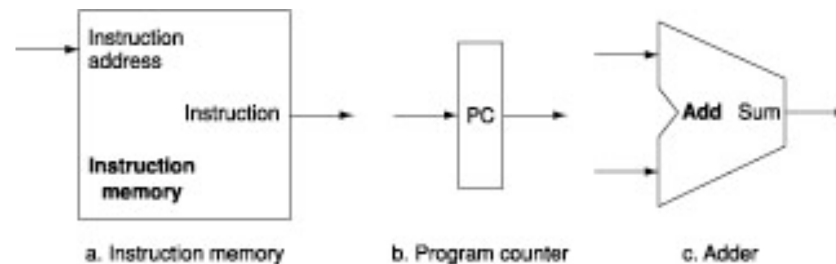


# Building a datapath

- MIPS subset implementation
- Simple implementation using a single clock cycle for every instruction
- Not practical
- Does not allow different instruction classes to take different number of clock cycles that may be shorter

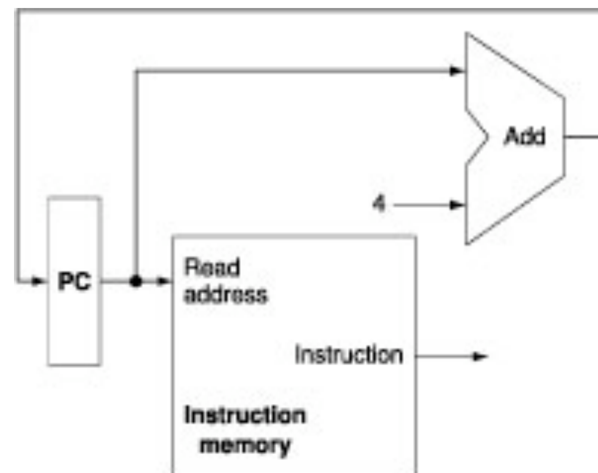
# Building a datapath

- Major components required to execute each class of MIPS instruction
  - Memory unit
    - Holds and supplies instructions given an address
  - program counter (PC)
    - Holds the address of next instruction to be executed
  - Adder
    - Makes PC point to the next instruction's address
    - Built from the ALU



# Building a datapath

- Instruction execution
  - Fetch the instruction from memory
  - Increment PC by 4



# Building a datapath

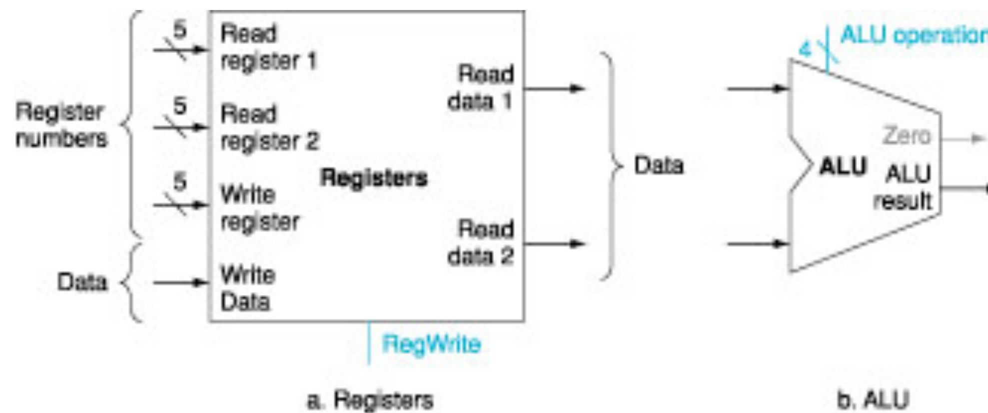
- R-format instructions: Includes arithmetic-logic instructions, such as add, sub, and, or, slt
- Read two registers, perform an operation, and write result to a third register
- Registers are stored in a structure called register file
- Register file has two read ports and one write port
- For each data word to be read from [written to] register file, we need
  - Register number to be read from [written to]
  - Output from [Input to] the register file to carry the value being read [written]

# Building a datapath

- The register file always outputs the contents of whatever register numbers are on the Read register inputs
- Writes are asserted by a write control signal, for a write to occur at a clock edge

# Building a datapath

- Four inputs: three to identify registers, one for data
- Inputs to identify registers are 5-bit wide
- Data input is 32-bits
- Two 32-bit outputs for data

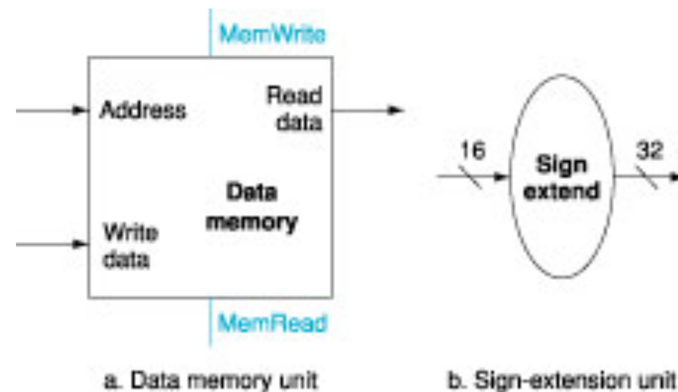


# lw and sw instructions

- General format: lw $t1, offset ($t2)
- Compute the memory address by adding the base register ($t2) to the 16-bit signed offset field
- For sw, value to be stored must be read from register file ($t1)
- For lw, value read from memory must be stored to register file in $t1
- Need the register file and ALU

# lw and sw instructions

- Also need a unit to sign extend 16-bit offset to 32-bit signed value, and a data memory unit
- Data memory
  - Must be written on sw
  - Must have both read and write control signal
  - Must have an address input
  - Must have a data input




a. Data memory unit    b. Sign-extension unit

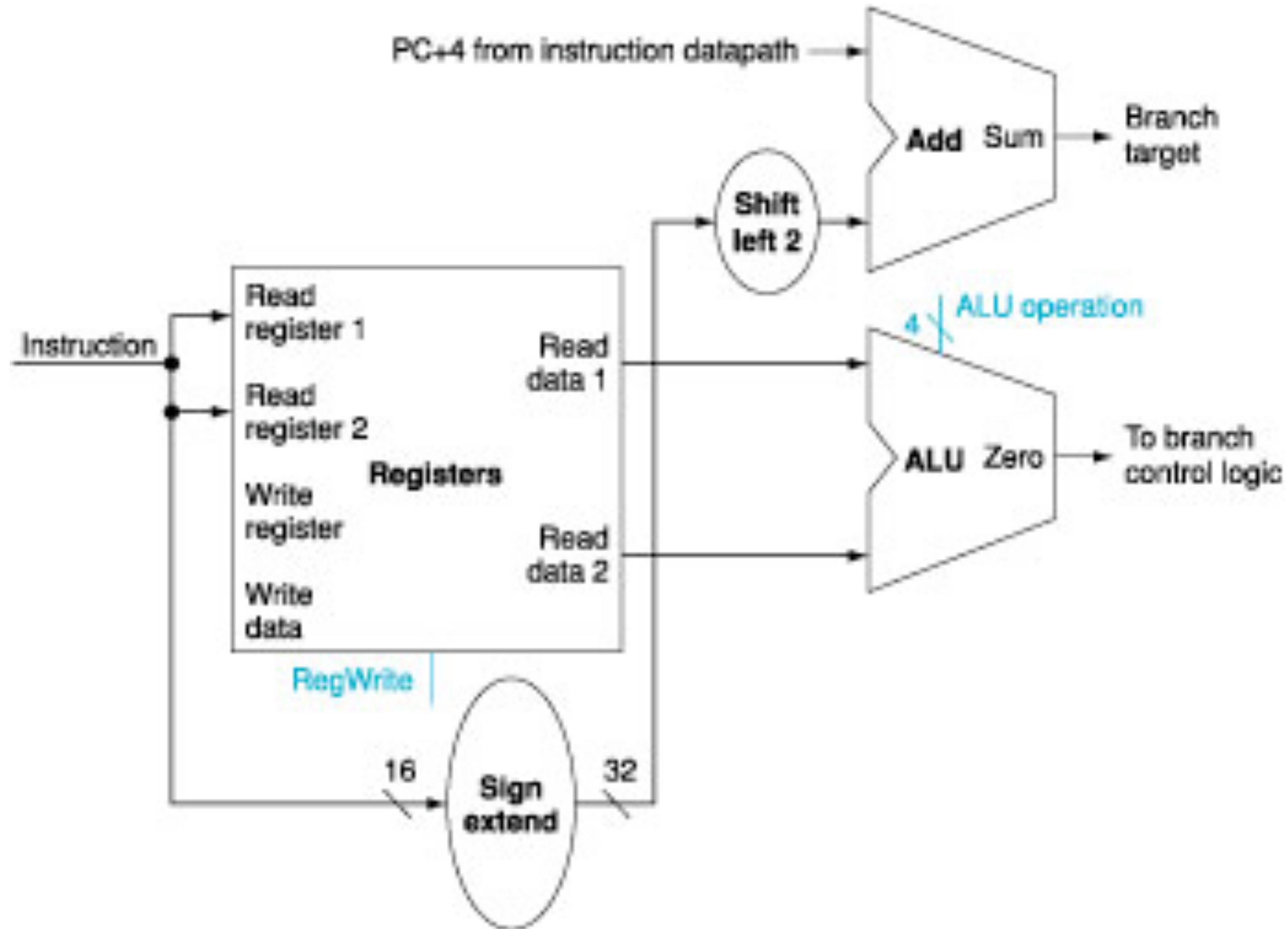
# beq instruction

- beq $t1, $t2, offset
- Three operands: two registers to compare for equality and a 16-bit offset to compute branch address relative to current instruction address
- Compute the branch address by adding sign-extended offset to PC
- Base of branch address is the address of instruction following the branch
  - Taken care of during instruction fetch
- Offset field is a word offset, increasing the effective range of offset by a factor of 4
  - This also requires the offset field to be shifted by 2 bits

# beq instruction

- Datapath has to compute the target address and compare two registers
- Equality indicated by a zero asserted out of ALU, achieved by subtracting one register from another

# beq instruction



# Jump

- Jump has J format
- Destination address is 26 bits
- It is a word address
- We shift to the left to convert to a byte address which will give us a 28 bit quantity
- Jump operates by replacing the lower 28 bits of PC with lower 26 bits of instruction shifted left by 2 bits

| op | addr |
|---|---|
| 6 bits | 26 bits |

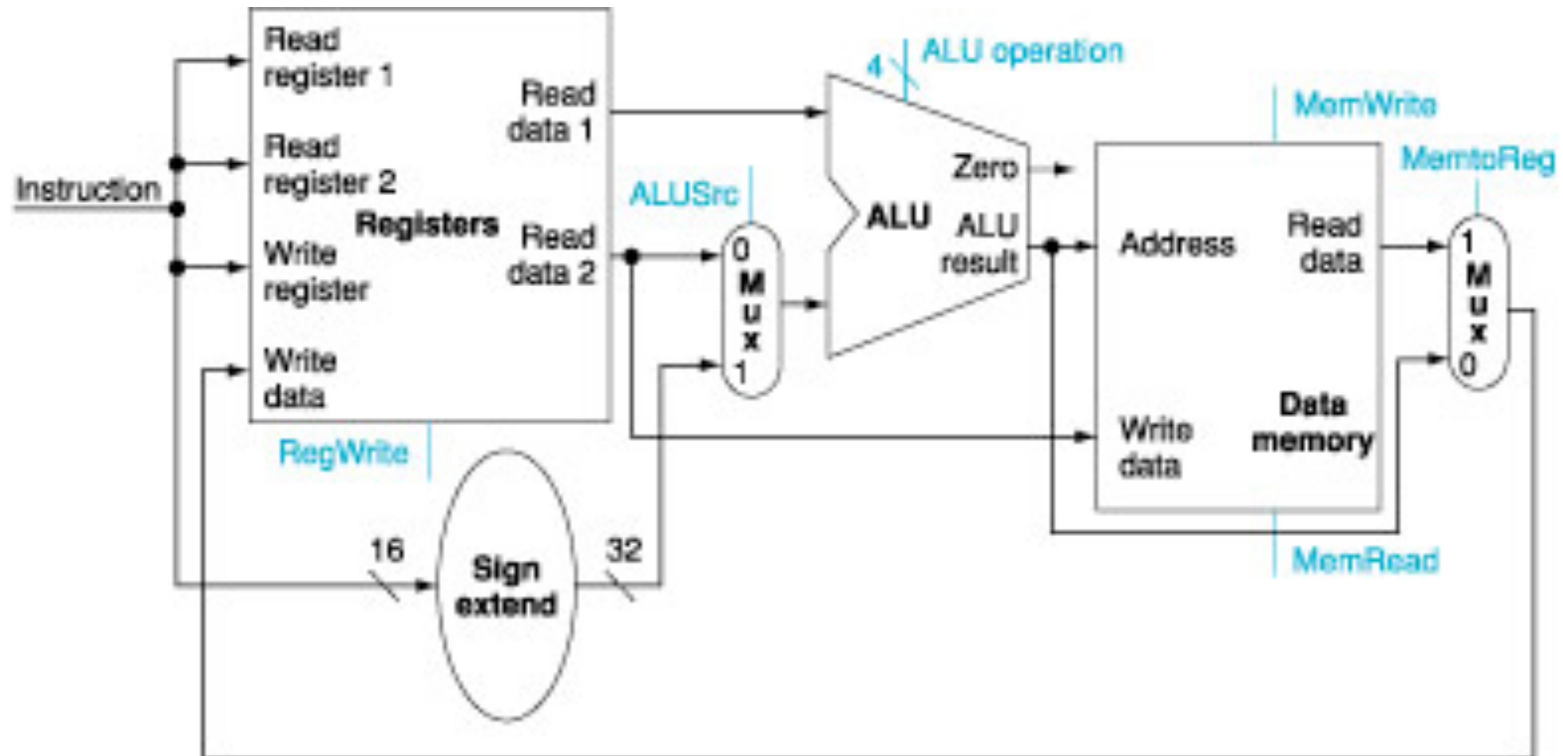
# Creating a single datapath

- Executing all instructions in a single clock cycle
  - No datapath resource can be used more than once in an instruction
  - Any element needed more than once must be duplicated
  - Instruction memory must be separated from data (separate datapath resources)
- Sharing datapath elements between different instruction classes
  - Need multiple connections to the input of an element, selected by a control signal
  - Commonly achieved by a multiplexor

# Creating a single datapath

- Key differences between R-type datapath and memory instruction datapath
  - Second input to ALU is either a register (R-type) or sign-extended 16 bit offset from the instruction
  - Value stored in destination register comes from ALU (R-type) or memory (lw)
- Problem: Combine the two datapaths using multiplexors, without duplicating the common functional units
- Solution
  - Must support two separate sources for second ALU input
  - Support two different sources for data stored in register file

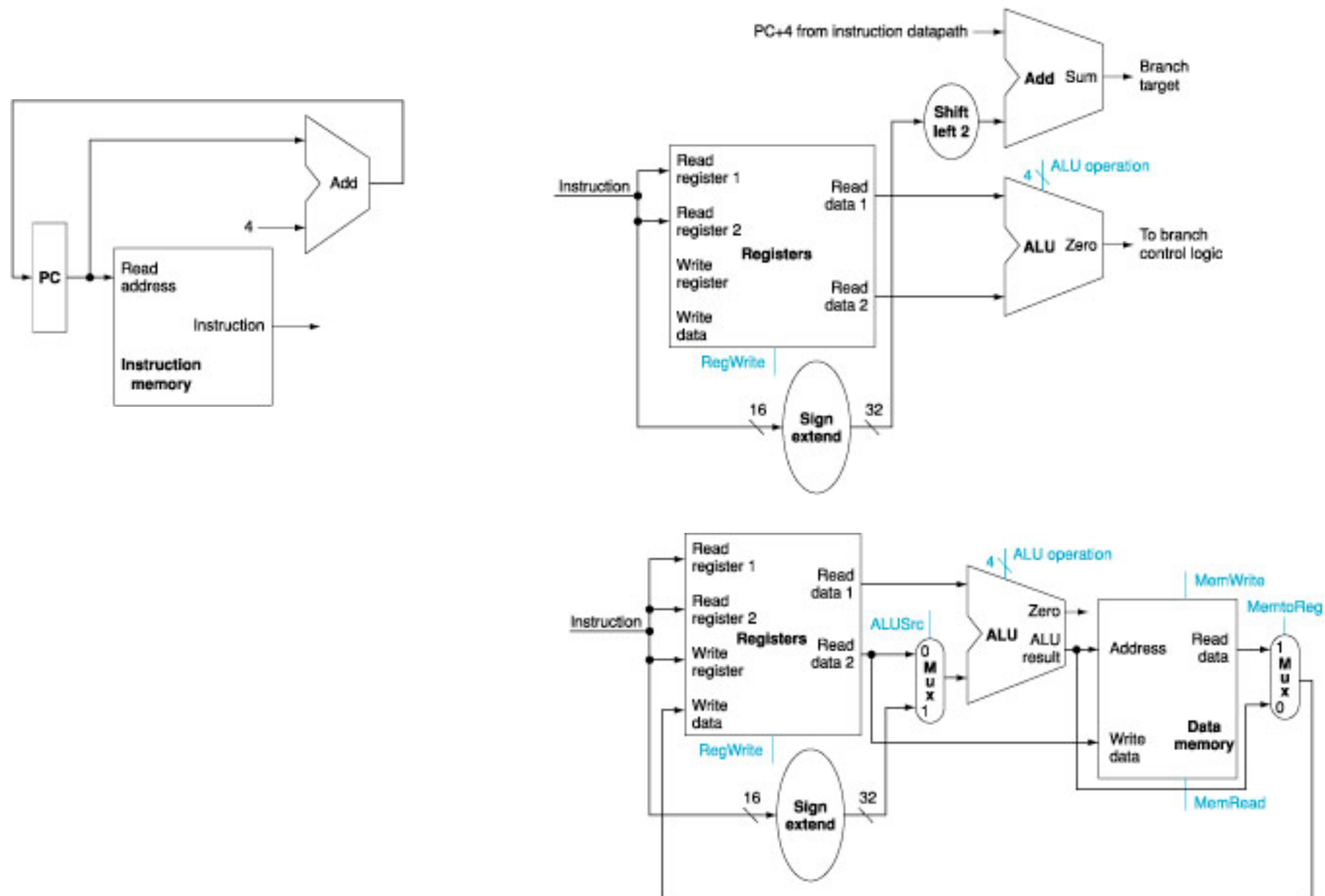
# Creating a single datapath



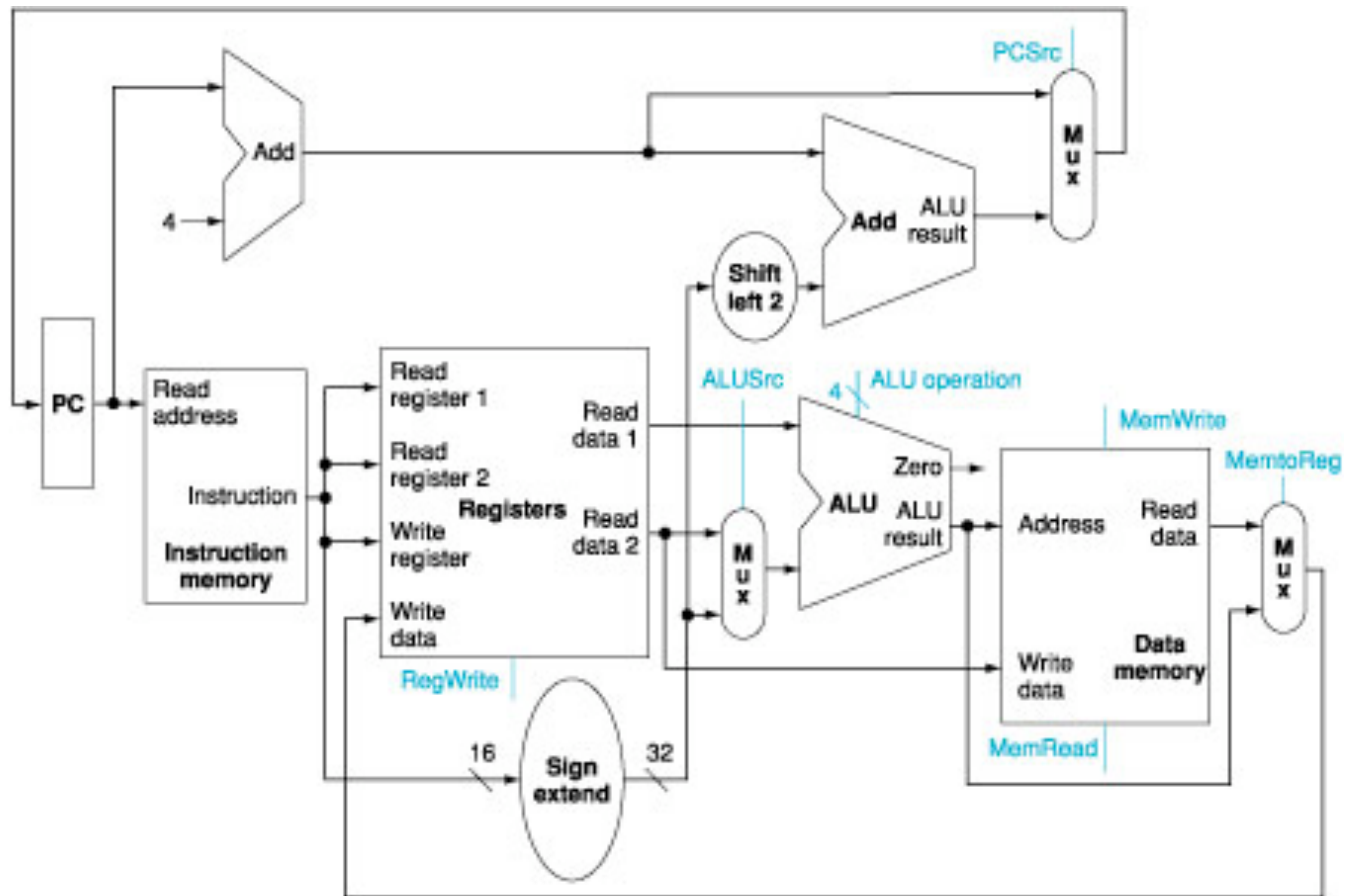
# Creating a single datapath

- Adding instruction fetch portion to the datapath
  - Separate instruction and data memory because of single cycle execution
  - Requires an adder (to increment PC) and an ALU (to execute instruction in the same clock cycle)
- Adding datapath for branches
  - Uses main ALU for comparing registers
  - Keep the adder to compute the branch target address
  - Additional multiplexor to select between target address and sequentially following address for PC

# Creating a single datapath



# Creating a single datapath



# Adding Control

- CPU = Datapath + Control
- Single-Cycle Design:
  - Instruction takes exactly one clock cycle
  - Datapath units used only once per cycle
  - Writable state updated at end of cycle
- What must be "controlled"?
  - Multiplexors (Muxes)
  - Writable state elements: Register File, Data Memory (Dmem)
    - *what about PC? Imem?*
  - ALU (which operation?)

# Processor = Datapath + Control

- Single-Cycle Design: everything happens in one clock cycle

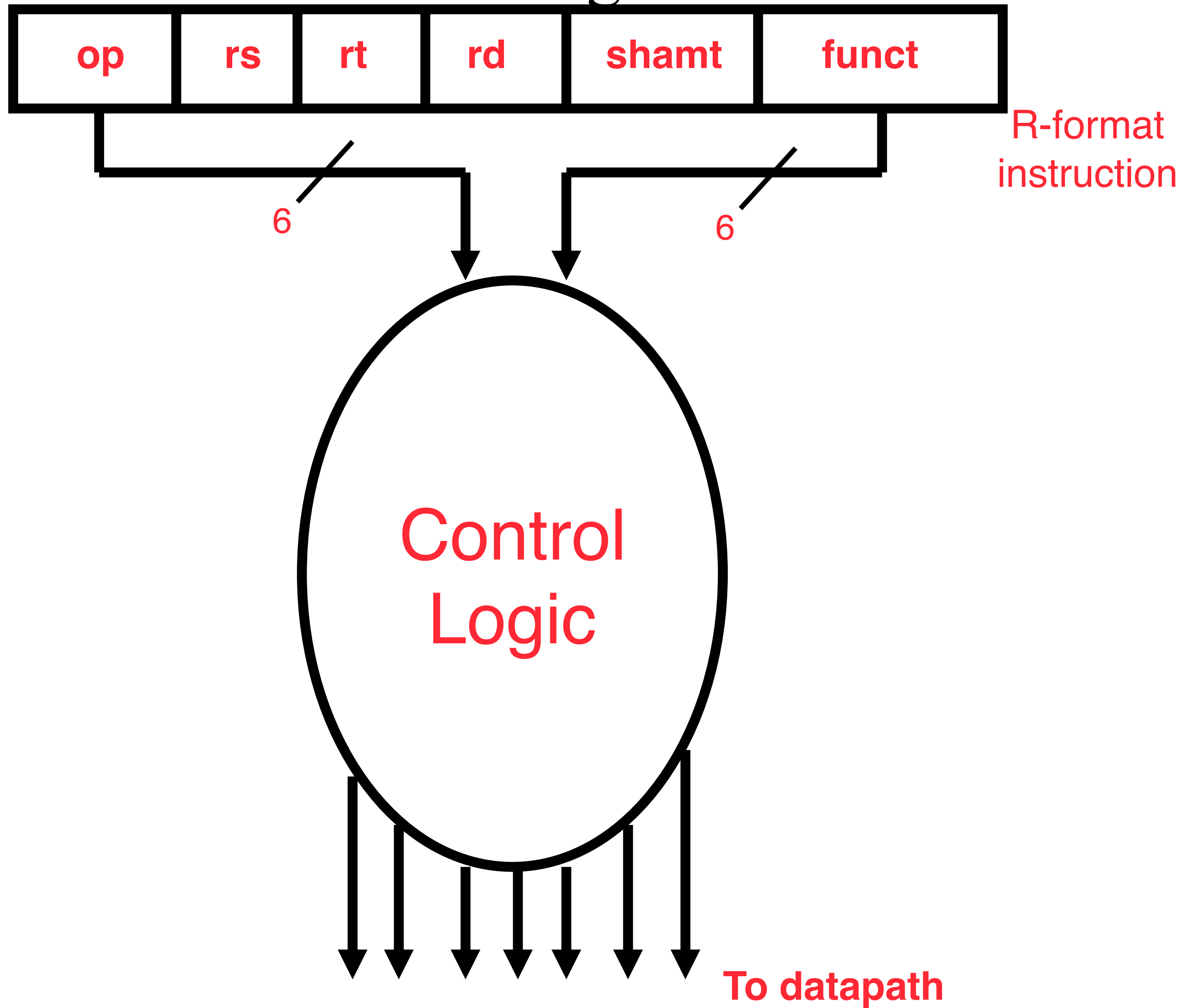
  ⇒

  until next falling edge of clock, processor just one big combinational circuit!!!

  ⇒

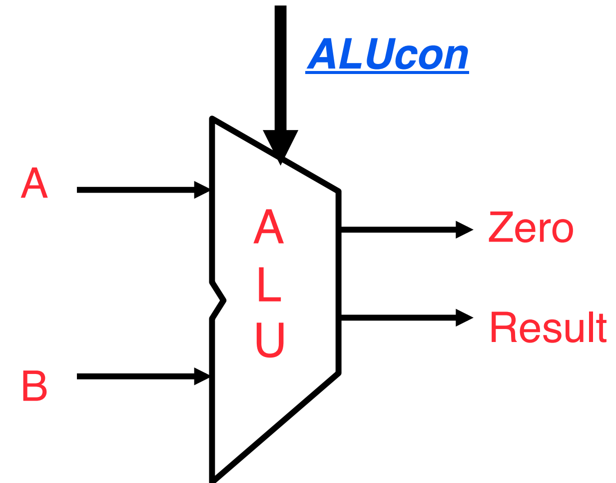
  control is just a combinational circuit
  (output, just function of inputs)
- outputs? control points in datapath
- inputs? the current instruction! (`opcode`, `funct` control everything)

# Defining Control



# Defining ALU Control

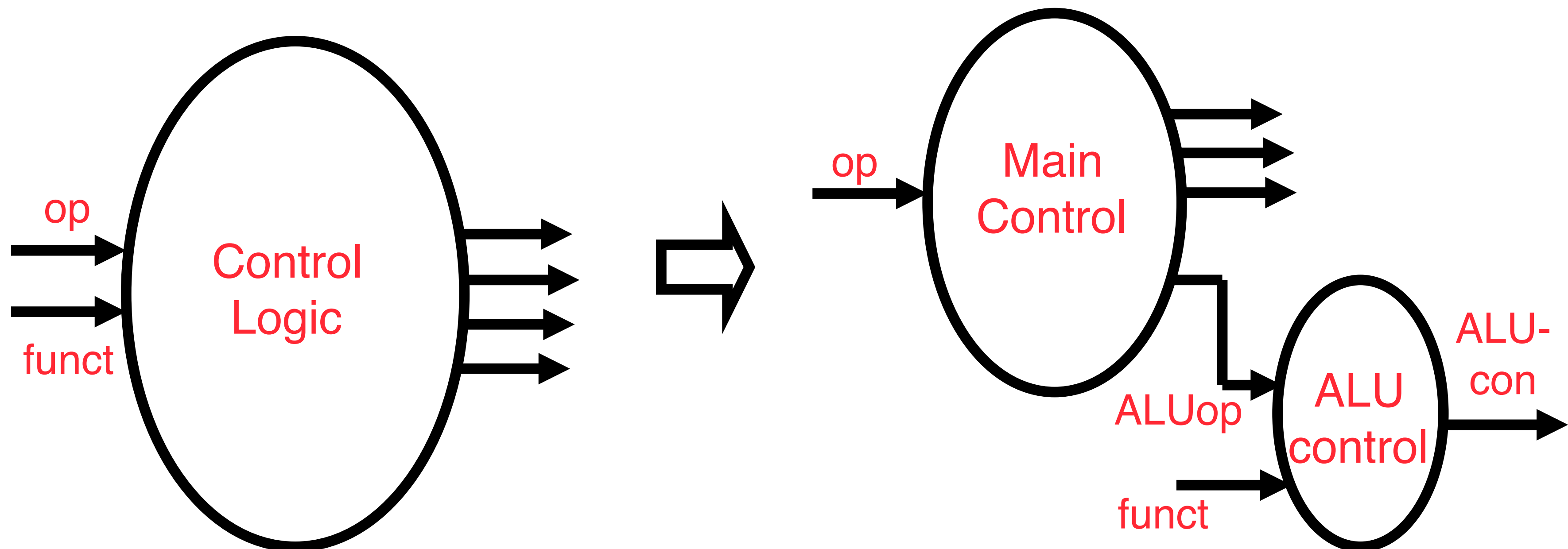
- Four control inputs (bits)
- Only six of possible 16 combinations are used




| ALUcon | ALU function | Instruction(s) supported |
|---|---|---|
| 0000 | AND | R-format (and) |
| 0001 | OR | R-format (or) |
| 0010 | add | R-format (add), lw, sw |
| 0110 | subtract | R-format (sub), beq |
| 0111 | set on less than | R-format (slt) |

# Defining Control

- Note that funct field only present in R-format instruction - funct controls ALU only
- To simplify control, define Main, ALU control separately – using multiple levels will also increase speed – important optimization technique
- ALUop inputs will be defined



# ALU control

- Depending on instruction class, ALU has to perform one of the first five functions
  - For R-type instructions, ALU does one of the five functions based on the value of 6-bit funct field in low-order bits of instruction
  - For lw and sw, ALU computes memory address by addition
  - For branch, ALU performs a subtraction

R format

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

I format

| op | rs | rt | Constant or address |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# ALU control

- Control unit for the 4-bit ALU control
  - Input from funct field of opcode and a 2-bit control signal called ALUop
  - ALUOp is generated by the main control unit
  - ALUop indicates operation

| Bits | Operation |
|---|---|
| 00 | Add for load and store |
| 01 | Subtract for beq |
| 10 | Determined by operation encoded in funct field |

  - Output of ALU control is a 4-bit signal (one of five combinations) to control the ALU

# ALU control

- Setting ALU control bits based on 2-bit ALUop control and 6-bit funct field
- Notice the relationship between ALUop and instruction type

| Instruction | ALUOp | Instruction operation | Funct field | Desired ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 010 |
| SW | 00 | store word | XXXXXX | add | 010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 110 |
| R-type | 10 | add | 100000 | add | 010 |
| R-type | 10 | subtract | 100010 | subtract | 110 |
| R-type | 10 | AND | 100100 | and | 000 |
| R-type | 10 | OR | 100101 | or | 001 |
| R-type | 10 | set on less than | 101010 | set on less than | 111 |

# ALU control

- Hierarchy of multiple decoding levels
  - Main control unit generates ALUop bits
  - ALU control unit uses ALUop bits to generate actual signals
  - Multiple levels reduce size of control unit
  - Potential increase in control unit speed

# ALU control

- Mapping from ALUop bits and funct to 3-bit ALU operation (the forth bit is always 0)
  - funct is used only when ALUop is 10
  - Create a truth table of the small number of possibilities
  - Only show those inputs for which the output is defined
- The truth table can be optimized (as above) and turned into hardware circuitry using standard algorithms

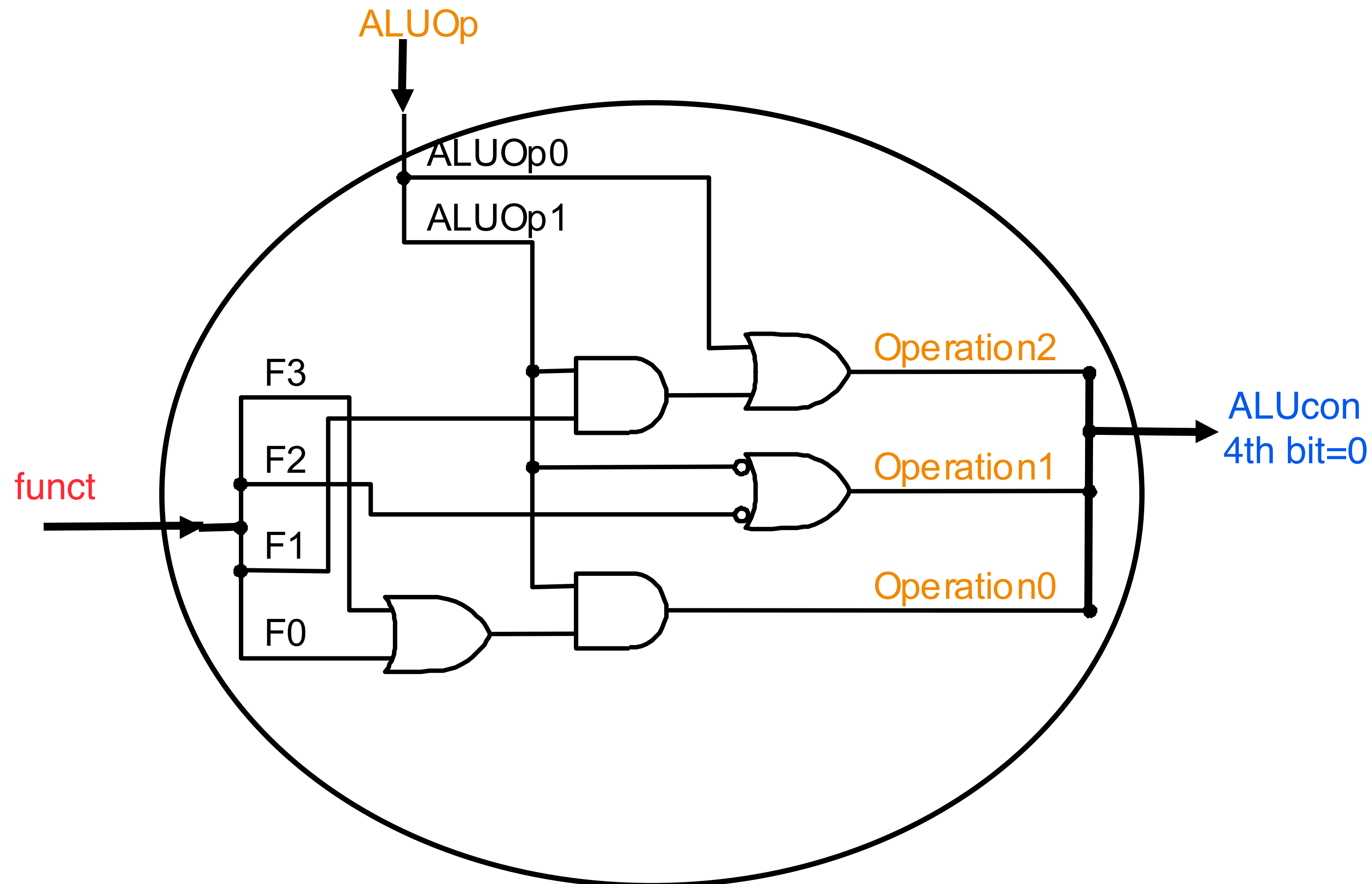
| ALUOp | | Funct field | | | | | | Operation |
|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp0 | F5 | F4 | F3 | F2 | F1 | F0 | |
| 0 | 0 | X | X | X | X | X | X | 0010 |
| X | 1 | X | X | X | X | X | X | 0110 |
| 1 | X | X | X | 0 | 0 | 0 | 0 | 0010 |
| 1 | X | X | X | 0 | 0 | 1 | 0 | 0110 |
| 1 | X | X | X | 0 | 1 | 0 | 0 | 0000 |
| 1 | X | X | X | 0 | 1 | 0 | 1 | 0001 |
| 1 | X | X | X | 1 | 0 | 1 | 0 | 0111 |

**FIGURE 5.13 The truth table for the three ALU control bits (called Operation).** The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

# Deriving ALUcon outputs

- From the truth table, output signals can be easily derived because of don’t cares.
- Operation2 (msb) = ALUOp0 OR (ALUOp1 AND F1)
- Operation1 = ALUOp1 NOR F2
- Operation0 (lsb) = ALUOp1 AND (F3 OR F0)
- ALUOp is supplied by the main control unit (to be designed) and F0-F5 are the function field of the instruction.

# Fully Minimized ALU Control



# Designing the main control unit

- Fields of instructions and control lines needed for datapath
- Instruction formats
  - R-type instruction
  - Memory instruction: I format
  - Branch instructions: I format

R format

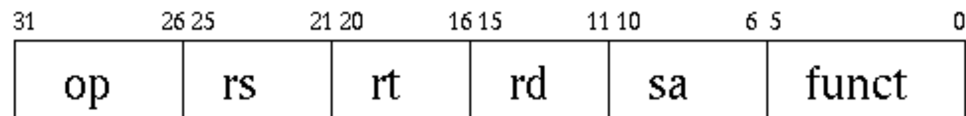
| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

I format

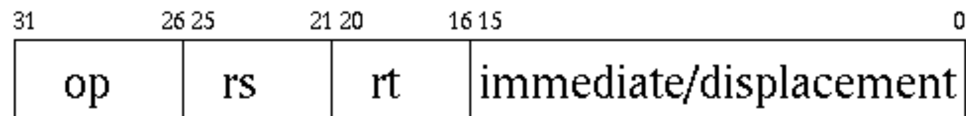
| op | rs | rt | Constant or address |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Designing the main control unit

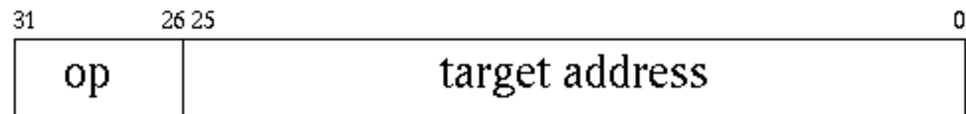
R–type format (Register)




I–type format (Immediate)



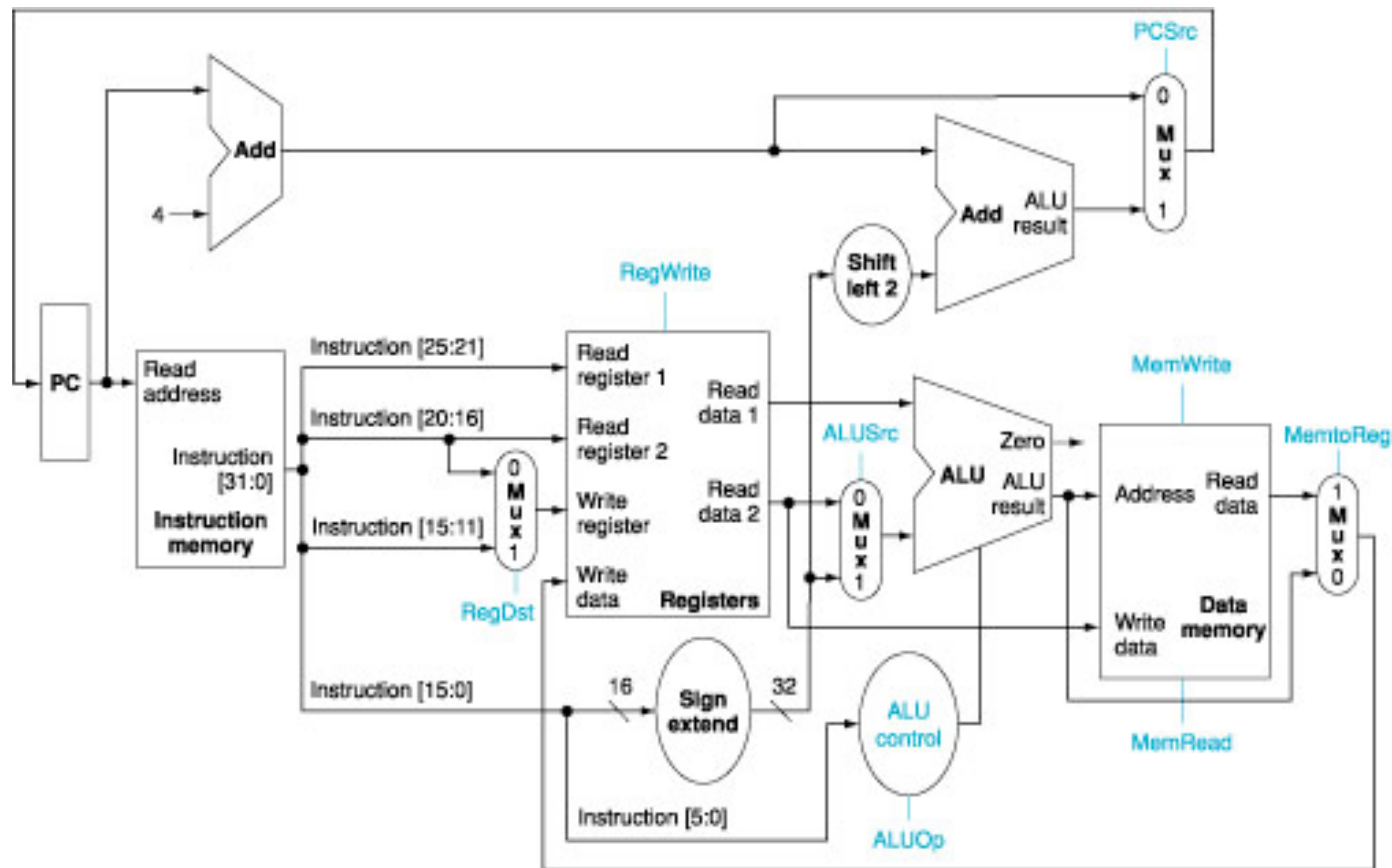

J–type format (Jump)



# Designing the main control unit

- Observations
  - Opcode is always in bits 31-26 (high-order 6 bits); refer to it as op[5-0]
  - Two registers to be read are always specified by rs and rt in bits 25-21 and 20-16 (for all R instructions, sw and beq)
  - Base register for load and store is always rs, bits 25-21
  - 16-bit offset for branch, load, and store is always in low-order 16 bits (15-0)
  - Destination register is in one of two places:
    1. For load, it is in bits 20-16 (rt)
    2. For R-type instructions, it is in bits 15-11 (rd)
    - No destination for other instructions (store and branch)
    - Add a multiplexor to select the field of instruction for register number
- Add extra multiplexor and instruction labels to the simple datapath

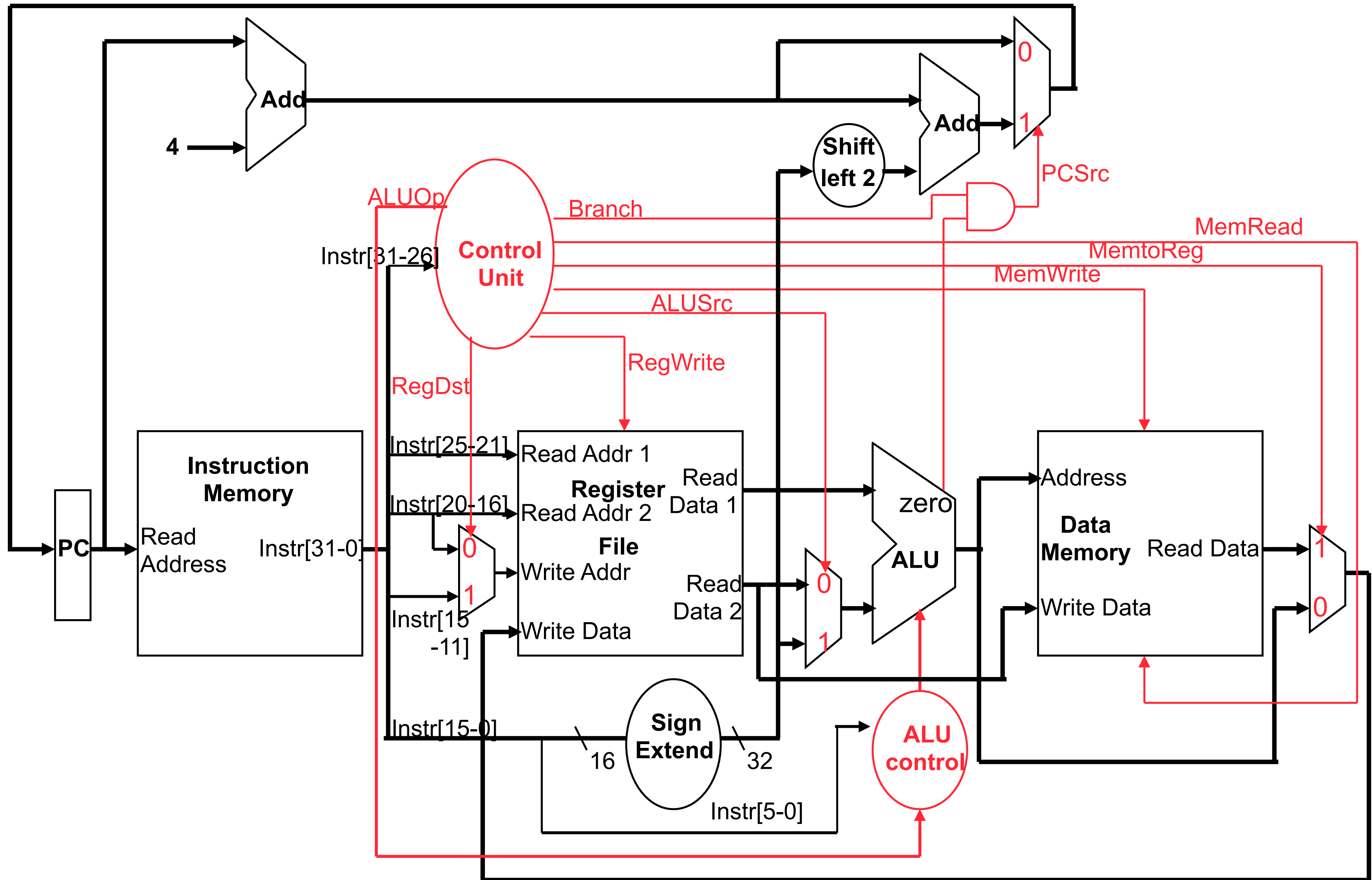
# Designing the main control unit



# Designing the main control unit

- A total of nine control signals
  - Seven above and two ALUop
  - Set on the basis of 6-bit opcode field input to control unit

# (Almost) Complete Datapath with Control Unit

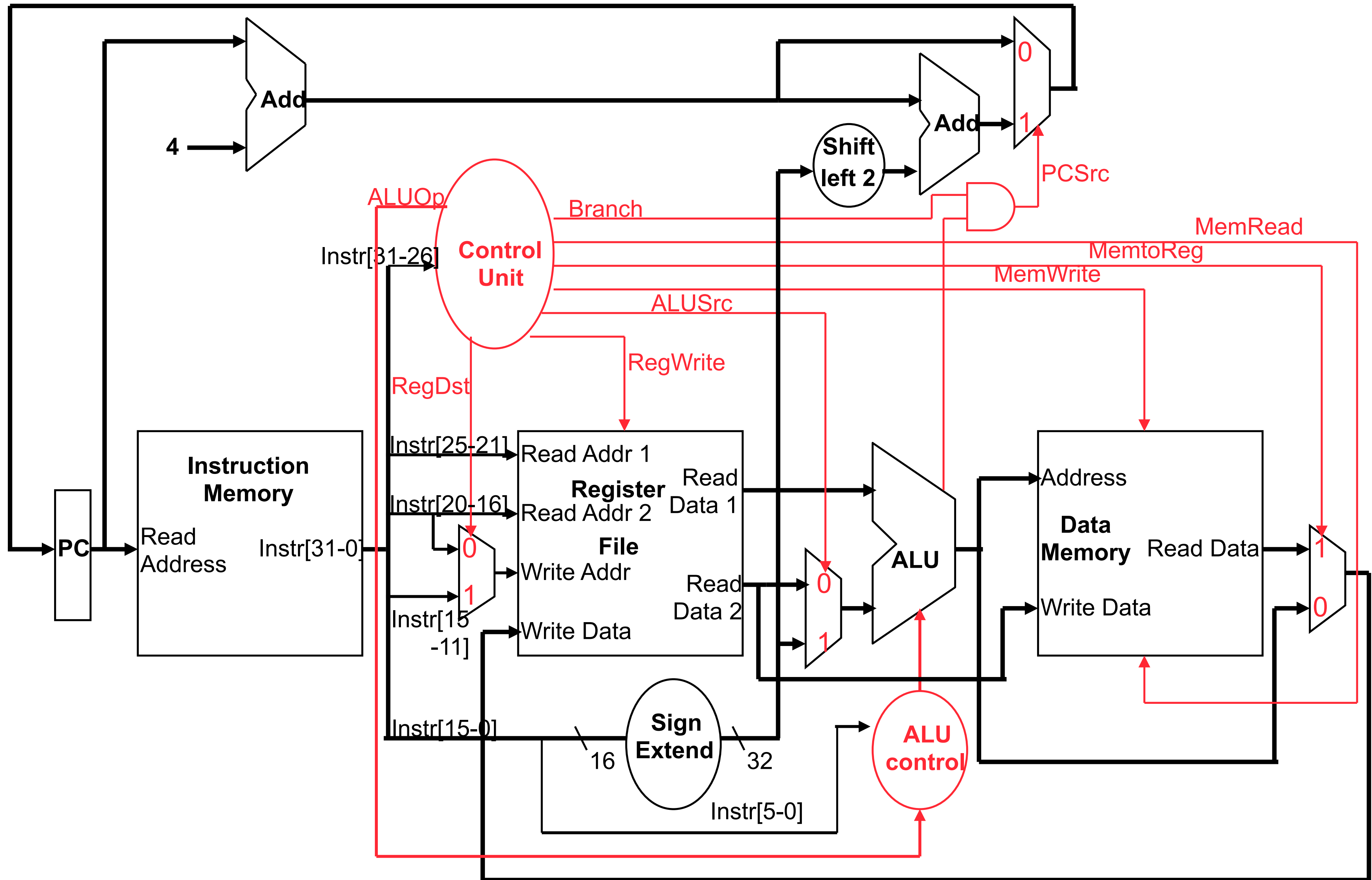


# Main Control Unit

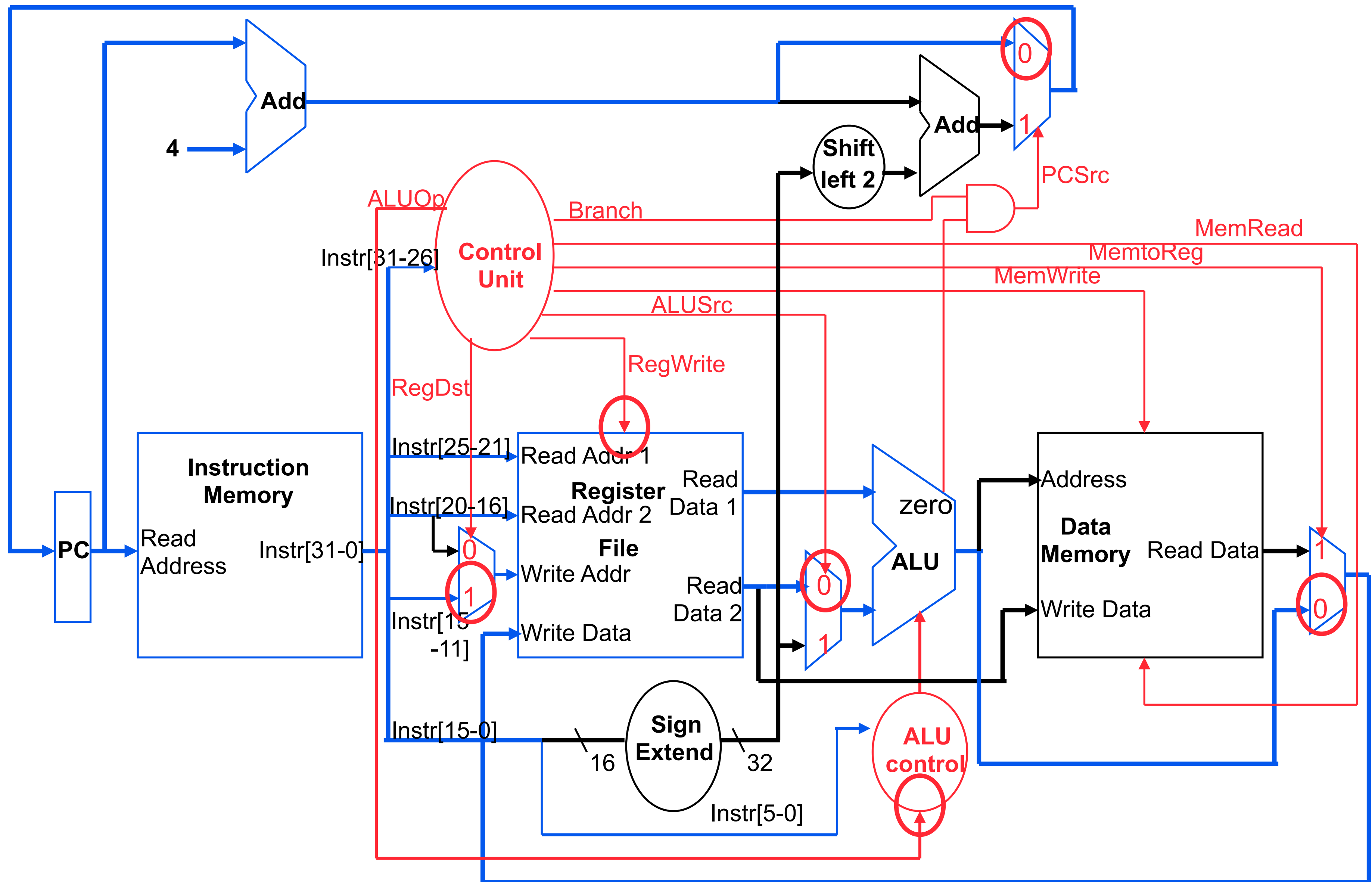
| Instr | RegDst | ALUSrc | MemReg | RegWr | MemRd | MemWr | Branch | ALUOp1 | ALUOp2 |
|---|---|---|---|---|---|---|---|---|---|
| **R-type** 000000 | | | | | | | | | |
| **lw** 100011 | | | | | | | | | |
| **sw** 101011 | | | | | | | | | |
| **beq** 000100 | | | | | | | | | |

- Completely determined by the instruction opcode field
  - Note that a multiplexor whose control input is 0 has a definite action, even if it is not used in performing the operation

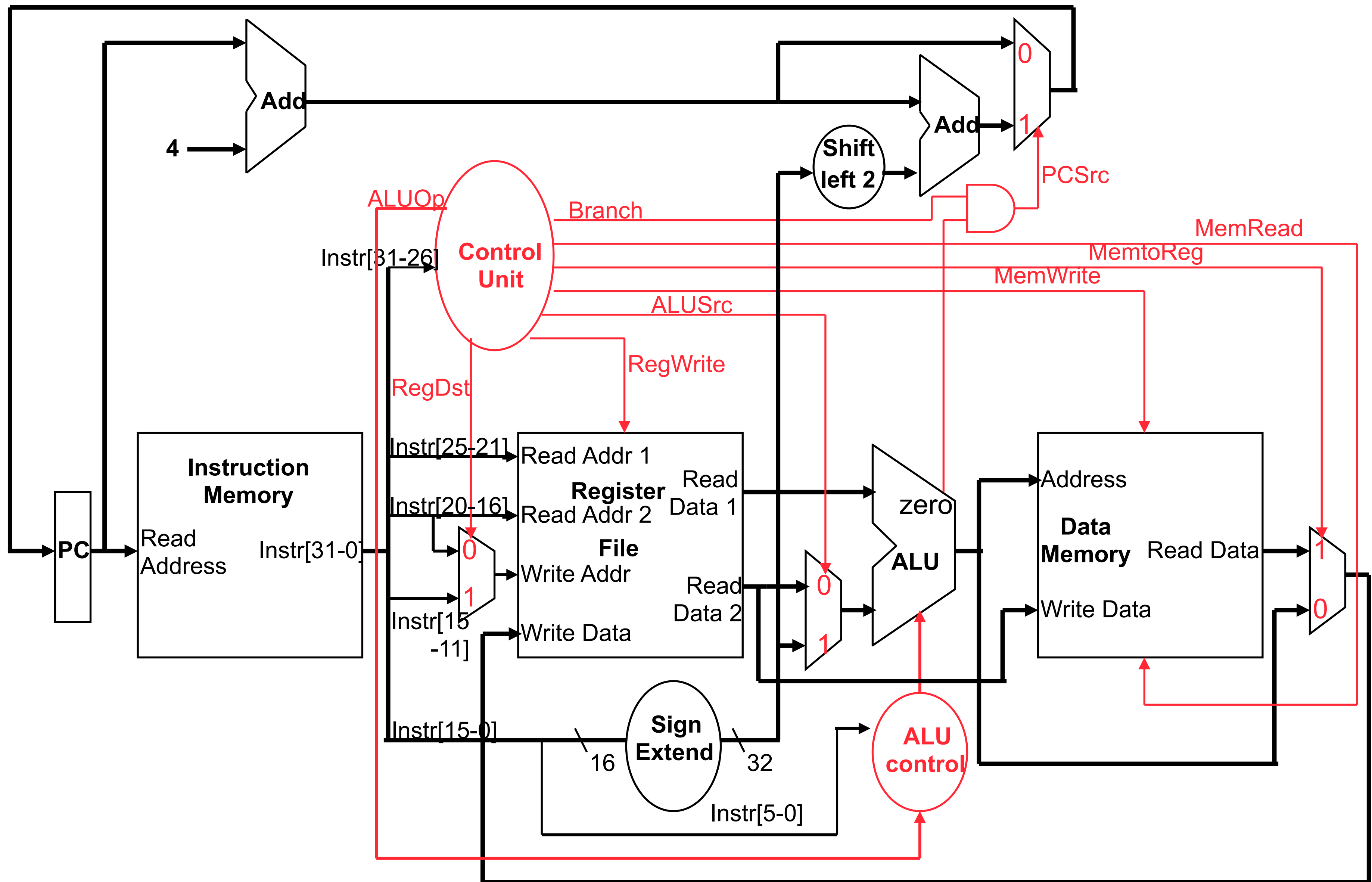
# R-type Instruction Data/Control Flow



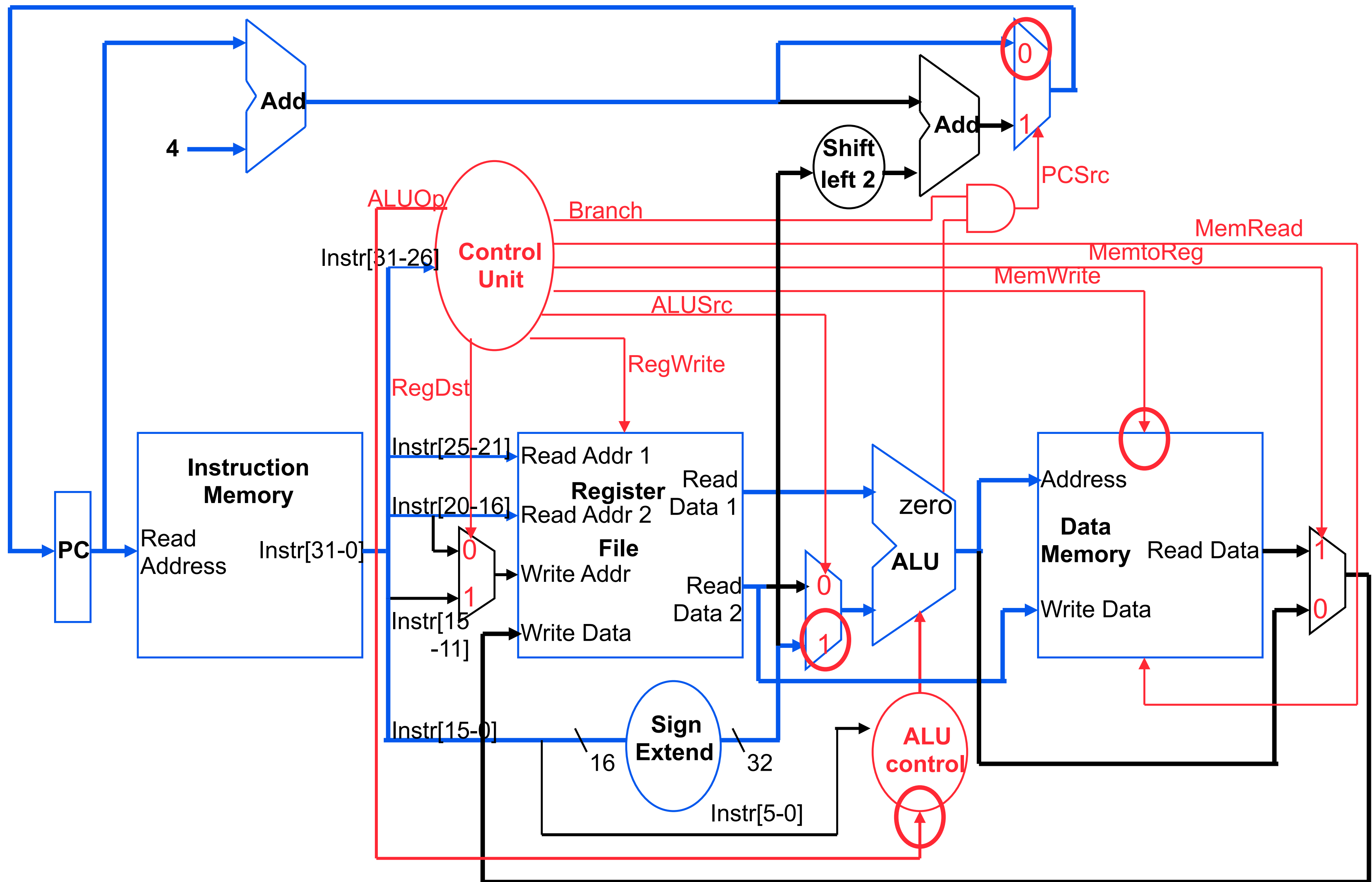
# R-type Instruction Data/Control Flow



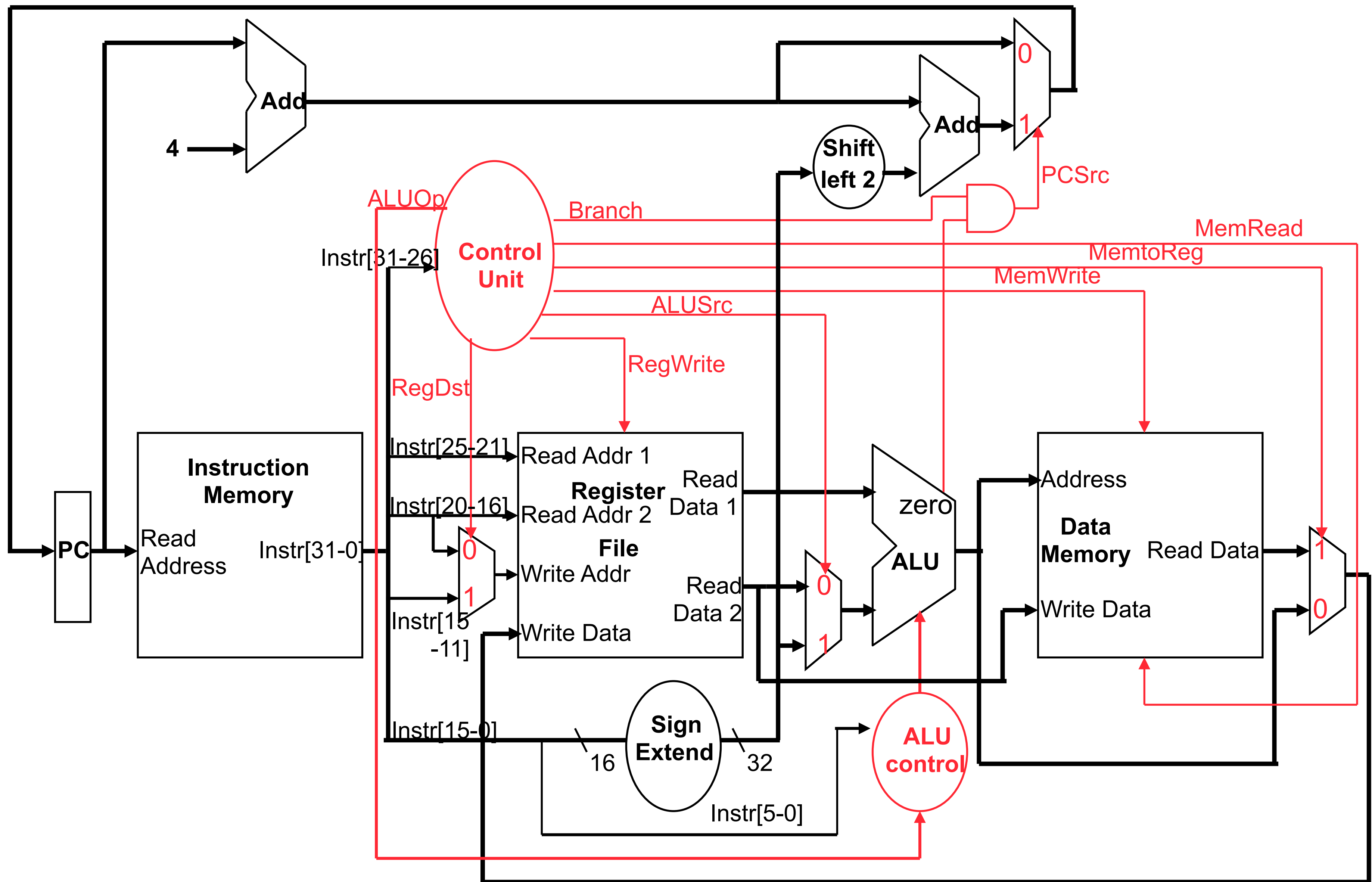
# Store Word Instruction Data/Control Flow



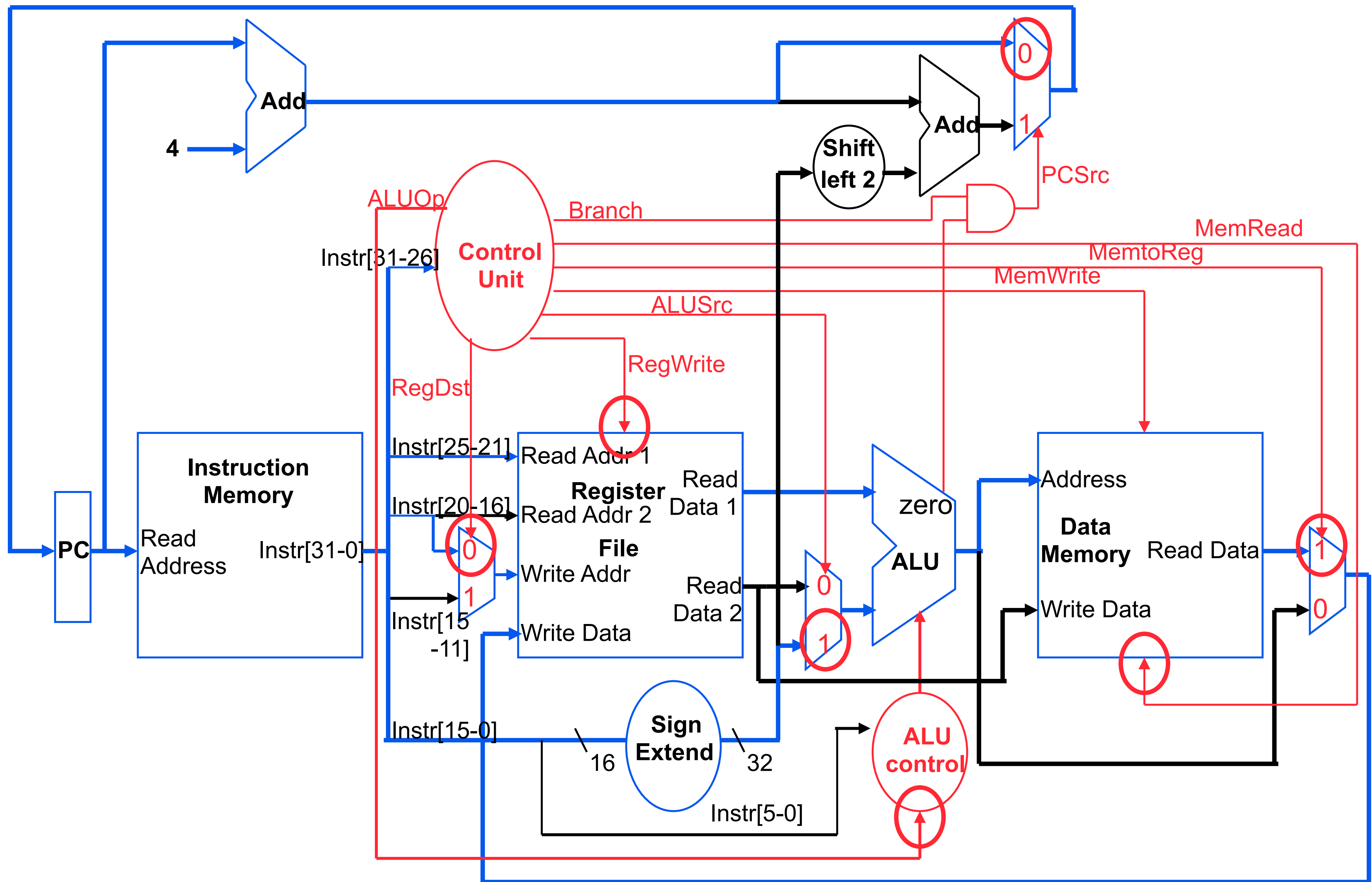
# Store Word Instruction Data/Control Flow



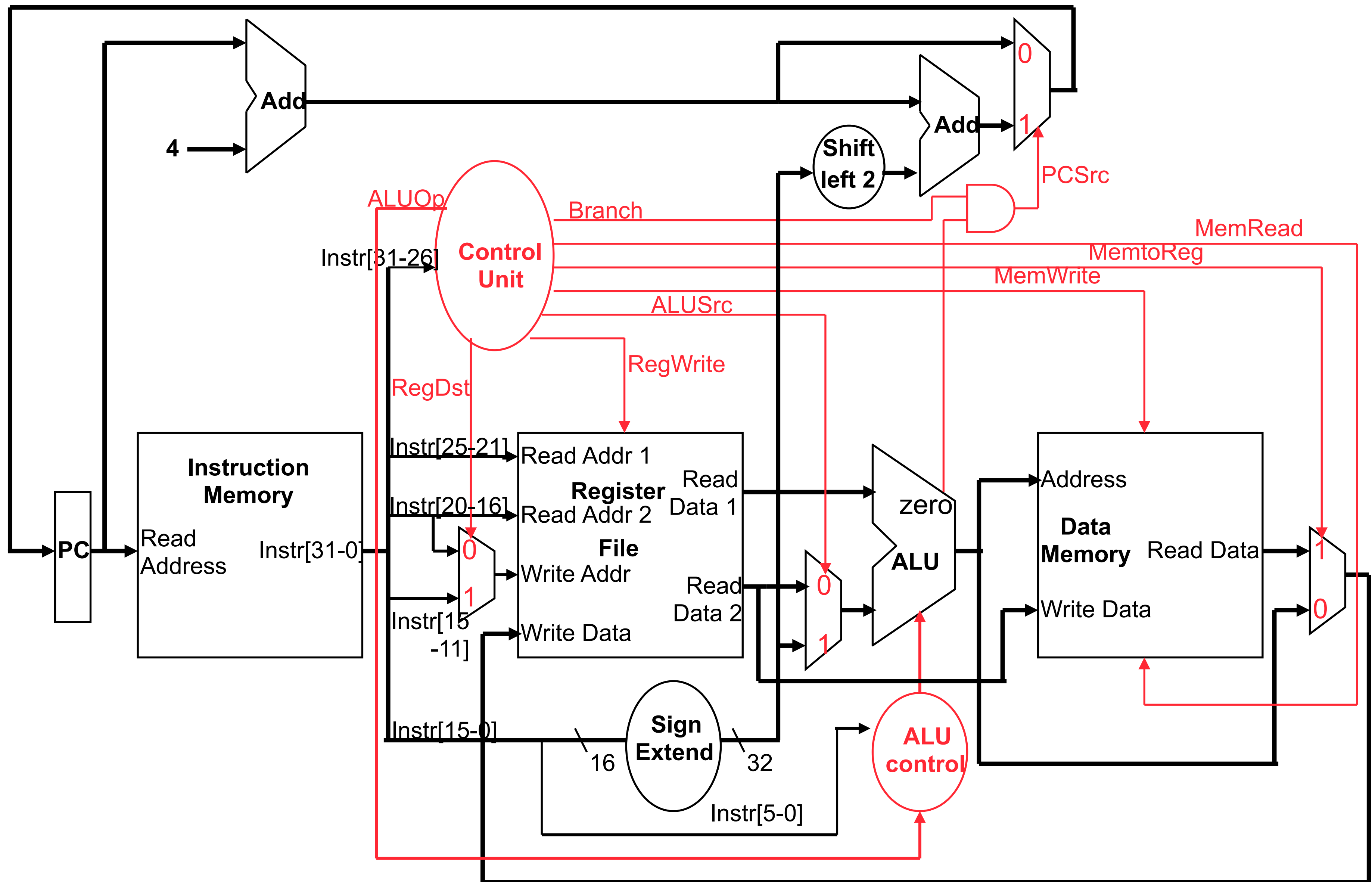
# Load Word Instruction Data/Control Flow



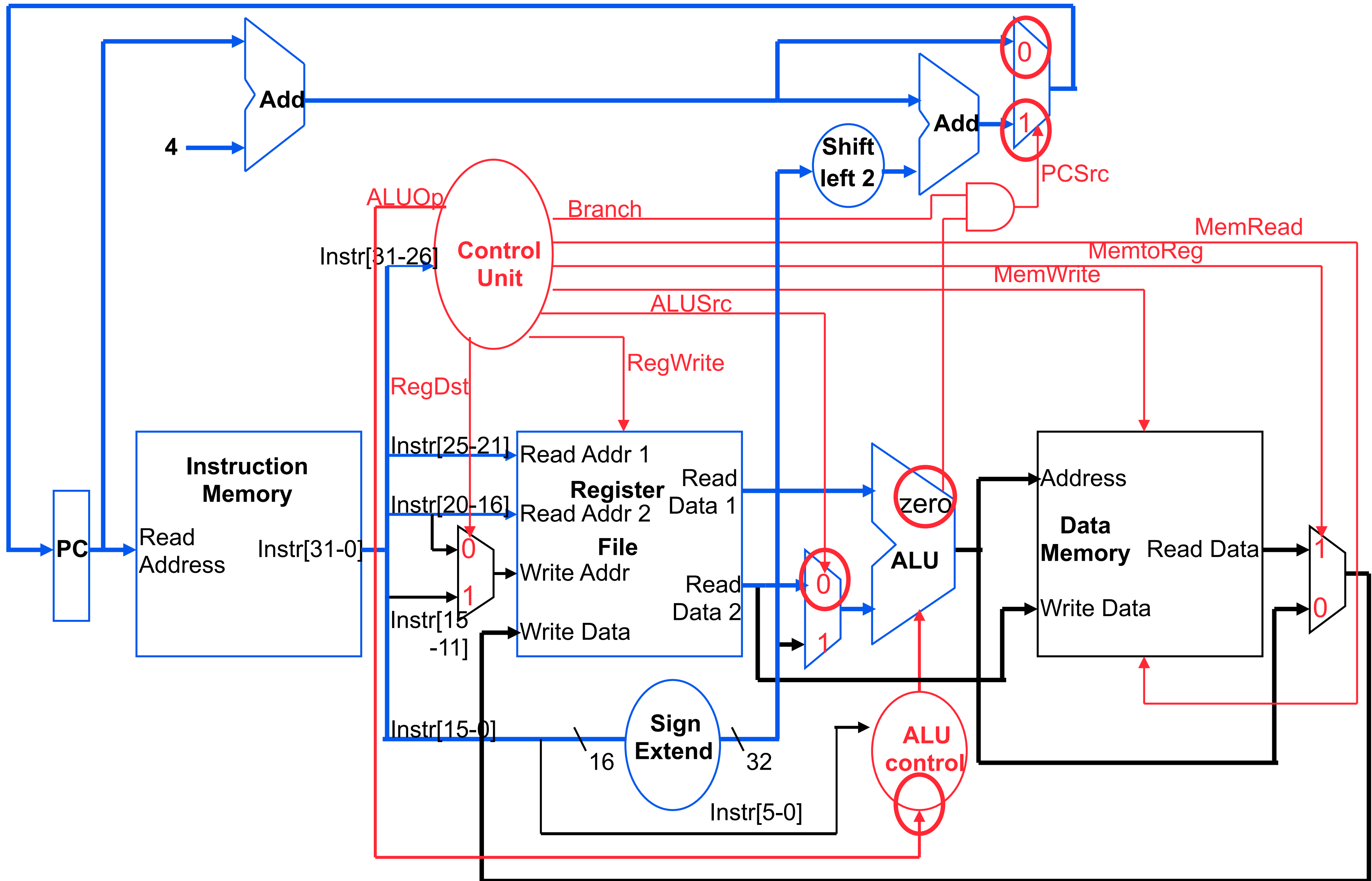
# Load Word Instruction Data/Control Flow



# Branch Instruction Data/Control Flow



# Branch Instruction Data/Control Flow



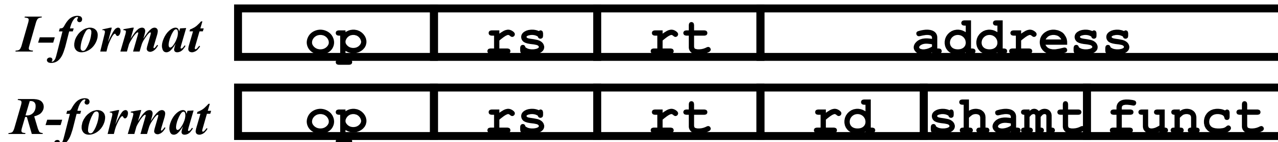
## Main Control Unit

| Instr | RegDst | ALUSrc | MemReg | RegWr | MemRd | MemWr | Branch | ALUOp1 | ALUOp0 |
|---|---|---|---|---|---|---|---|---|---|
| **R-type** 000000 | 1 | 0 | 0 | 1 | X | 0 | 0 | 1 | X |
| **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 | X | 1 |

- Completely determined by the instruction opcode field
  - Note that a multiplexor whose control input is 0 has a definite action, even if it is not used in performing the operation

# **RegDst** Control Line

- Controls a multiplexor that selects one of the fields **rt** or **rd** from an R-format or I-format instruction.
- **lw** needs to *write* to the register **rt**.



# `RegDst` usage

- `RegDst` should be
  - `0` to send `rt` to the write register # input.
  - `1` to send `rd` to the write register # input.

- `RegDst` is a function of the opcode field:
  - If instruction is `lw, RegDst` should be `0`
  - For all other instructions `RegDst` should be `1`

# `RegWrite` Control

- A 1 tells the register file to write to a register.
  - whatever register is specified by the write register # input is written with the data on the write register data inputs.
- Should be a **`1`** for arithmetic/logical instructions and for a load.
- Should be a **`0`** for load or **`beq`**.

# **ALUSrc** Control

- MUX that selects the source for the second ALU operand.
  - 0 means select the second register file output (read data 2).
  - 1 means select the sign-extended 16 bit offset (part of the instruction).
- Should be a **1** for load and store.
- Should be a **0** for everything else.

# `MemRead` Control

- A 1 tells the memory to put the content of the memory location (specified by the address lines) on the Read data output.

- Should be a **`1`** for load.
- Should be a **`0`** for everything else.

# `MemWrite` Control

- 1 means that memory location (specified by memory address lines) should get the value specified on the memory Write Data input.

- Should be a `1` for store.
- Should be a `0` for everything else.

# **MemToReg** Control

- MUX that selects the value to be stored in a register (that goes to register write data input).
  - 1 means select the value coming from the memory data output.
  - 0 means select value coming from the ALU output.
- Should be a 1 for load.
- Should be a **0** for everything else.

# `PCSrc` Control

- MUX that selects the source for the value written in to the PC register.
  - 1 means select the output of the Adder used to compute the relative address for a branch.
  - 0 means select the output of the PC+4 adder.

- Should be a **`1`** for beq if registers are equal!
- Should be a **`0`** for other instructions or if registers are different.

## `PCSrc` depends on result of ALU operation!

- This control line can't be simply a function of the instruction (all the others can).

- **`PCSrc`** should be a 1 only when:
  - **`beq`** AND **`ALU zero`** output is a 1
- Control unit generates a signal called "branch" that we can AND with the ALU zero output.

# Finalizing control

- Output: Control lines
- Input: 6-bit opcode (bits 5:0)
- Encoding for input

| Name | Opcode in decimal | Opcode in binary | | | | | |
|---|---|---|---|---|---|---|---|
| | | op5 | op4 | op3 | op2 | op1 | op0 |
| R-format | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| lw | 35 | 1 | 0 | 0 | 0 | 1 | 1 |
| sw | 43 | 1 | 0 | 1 | 0 | 1 | 1 |
| beq | 4 | 0 | 0 | 0 | 1 | 0 | 0 |

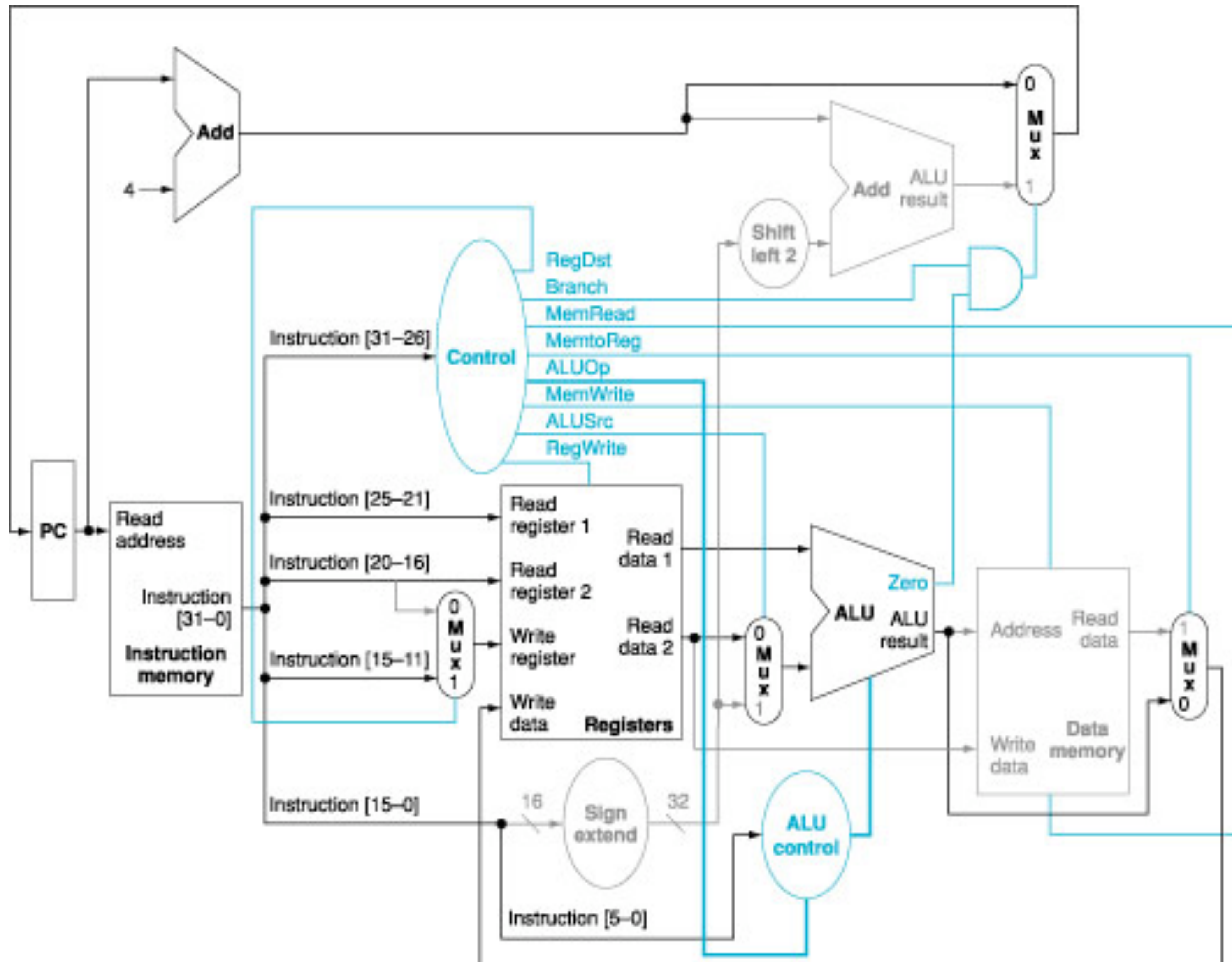
# Control function truth table

| | Signal | R-format | lw | sw | beq |
|---|---|---|---|---|---|
| Inputs | op5 | 0 | 1 | 1 | 0 |
| | op4 | 0 | 0 | 0 | 0 |
| | op3 | 0 | 0 | 1 | 0 |
| | op2 | 0 | 0 | 0 | 1 |
| | op1 | 0 | 1 | 1 | 0 |
| | op0 | 0 | 1 | 1 | 0 |
| 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 |
| | ALUOp2 | 0 | 0 | 0 | 1 |

# Operation of datapath

- Flow of three different instruction classes through datapath, with asserted control signals and active datapath elements highlighted
- R-type instruction (add $t1, $t2, $t3), executed in four steps
  - (a) Fetch instruction from memory and increment PC
  - (b) Read two registers ($t2 and $t3) from register file
  - (c) Operate on data read from register file, using funct bits (5:0) to generate ALU function
  - (d) Write the result from ALU to register file (bits 15:11 of the instruction selects the destination register)
- Implementation is combinational (not a series of four steps)
- Datapath operates in a single clock cycle

# Operation of datapath



# Operation of datapath

- Memory instruction (lw $t1, offset ( $t2 )) operate in five steps
  - (a) Fetch instruction from memory and increment PC
  - (b) Read register value ($t2) from register file
  - (c) Add the value read and sign-extended offset from lower 16-bits of instruction
  - (d) Use the sum (from ALU computed above) as address in data memory
  - (e) Write data from memory unit to register file; register destination is given by bits 20:16 ($t1)

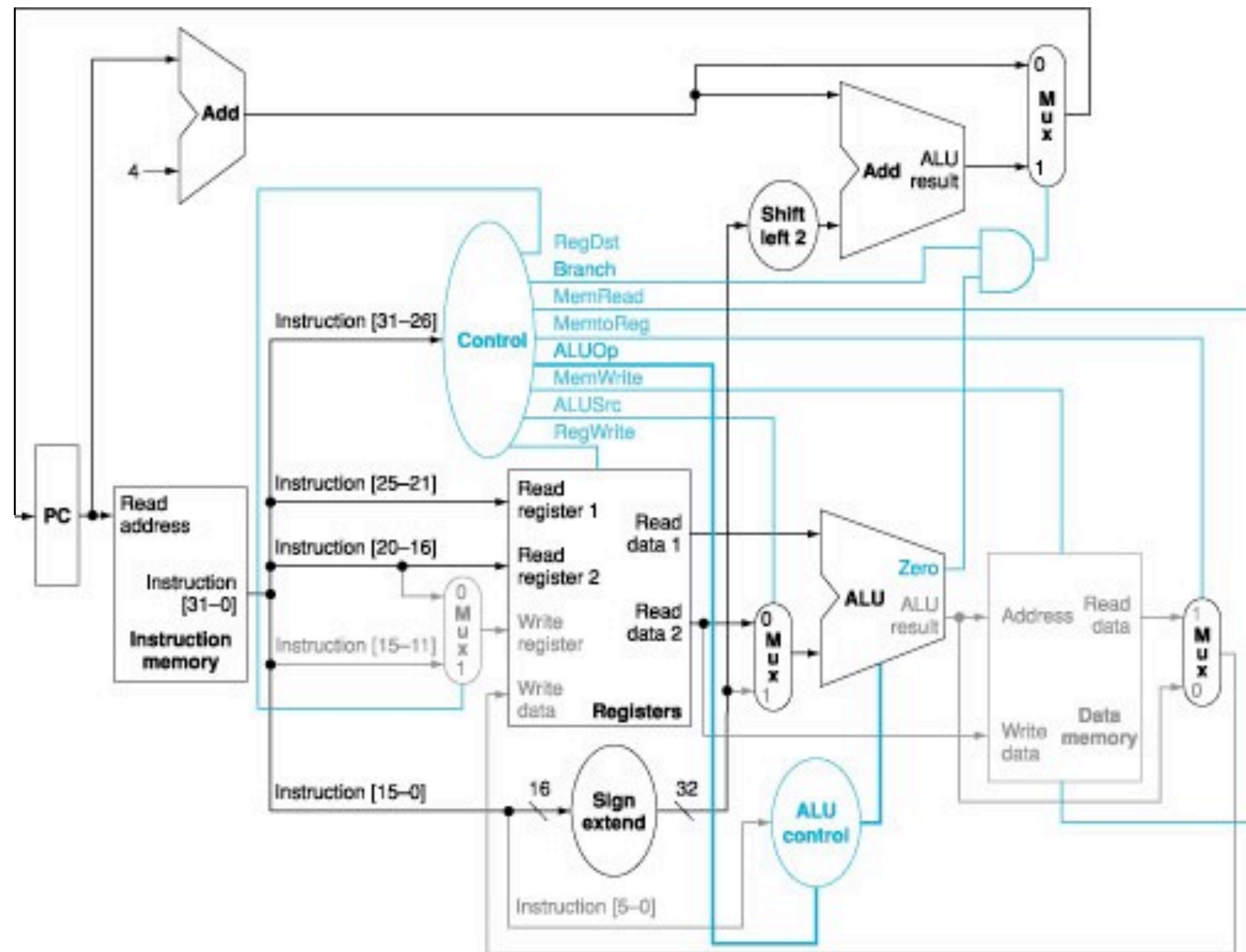
# Operation of datapath

Add
4
PC
Read address
Instruction [31–0]
Instruction memory
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Shift left 2
Add ALU result
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
16
Sign extend
32
ALU control
Instruction [5–0]
Mux

# Operation of datapath

- Branch-on-equal instruction (beq $t1, $t2, offset)
  - (a) Fetch instruction from memory and increment PC
  - (b) Read registers $t1 and $t2 from register file
  - (c) Use ALU to perform a subtract on data values read from registers; add offset (sign-extended and shifted lower 16-bits) to current value of PC (already incremented) to get branch target address
  - (d) Use zero result from ALU to decide the address to be stored in PC

# Operation of datapath



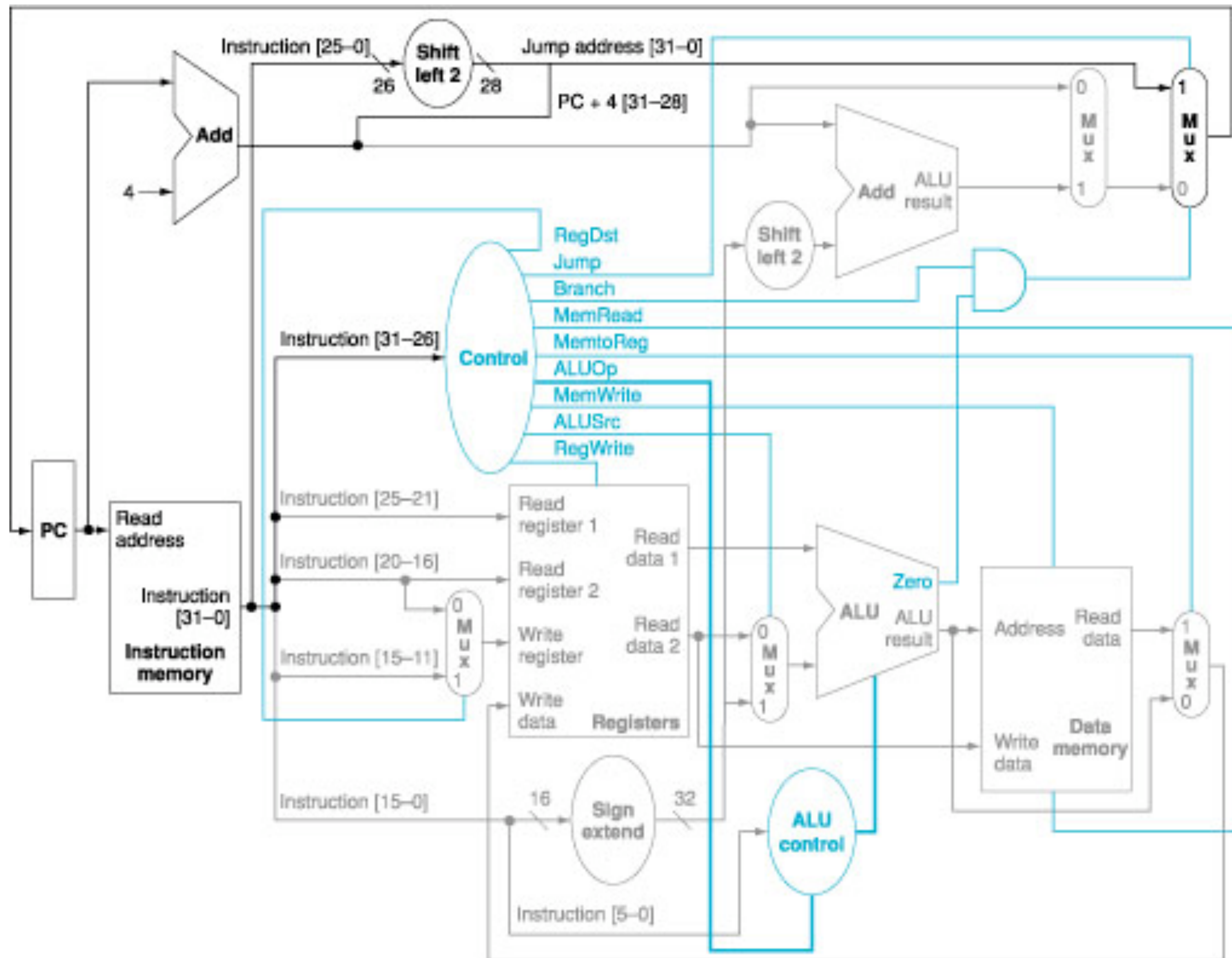
# Implementing jumps

- Jump instruction (opcode 2) format is given by

| op | addr |
|---|---|
| 6 bits | 26 bits |

- j computes target PC is not conditional
- The low-order two bits are always 00
- Next 26-bits come from the immediate field shown above
- The upper 4 bits come from the current value of PC (incremented to point to next instruction)
- Additional control signal for jump asserted only when opcode is 2

# Implementing jumps



# Inefficiency of single-cycle implementation

- Inefficiency of single-cycle implementation
  - Clock cycle must have the same length for every instruction in single cycle design
  - Clock cycle is determined by the longest possible path in the machine
  - For us, the longest path is the load instruction
    - Uses five functional units one after the other: instruction memory, register file, ALU, data memory, and register file
  - Several instructions could fit in a shorter clock cycle

# Inefficiency of single-cycle implementation

- Example: Performance of single-cycle machine
- Assume the operation time for major functional units as
  - Memory unit 200 ps
  - ALU and adders 100 ps
  - Register file (read or write) 50 ps

Assume no delay in multiplexors, control unit, PC access, sign extension, and other wires

Compare the performance of following implementations

1. Every instruction in one clock cycle of fixed length
2. Every instruction in one clock cycle of variable length clock, with clock being only as long as needed

Assume a mix of instructions as 25% loads, 10% stores, 45% R-type, 15% branches, and 5% jumps

# Inefficiency of single-cycle implementation

- CPU execution time = Instruction count x CPI x Clock cycle time
- For us, the CPI is 1, and hence
- CPU execution time = Instruction count x Clock cycle time
- Critical (longest) path for different instruction classes

| Instruction | Functional units used |
|---|---|
| R type: | Fetch Reg. access ALU Reg. access |
| Load word: | Fetch Reg. access ALU Mem. access Reg. access |
| Store word: | Fetch Reg. access ALU Mem. access |
| Branch: | Fetch Reg. access ALU |
| Jump: | Fetch |

- Compute the required length of each instruction class

| Instruction class | Instruction memory | Register read | ALU operation | Data memory | Register write | Total |
|---|---|---|---|---|---|---|
| R type | 200 | 50 | 100 | 0 | 50 | 400ps |
| Load word | 200 | 50 | 100 | 200 | 50 | 600ps |
| Store word | 200 | 50 | 100 | 200 | | 550ps |
| Branch | 200 | 50 | 100 | 0 | | 350ps |
| Jump | 200 | | | | | 200ps |