

ECE260: Fundamentals of Computer Engineering

Datapath for a Simplified Processor

James Moscola
Dept. of Engineering & Computer Science
York College of Pennsylvania



Introduction

- What we've done so far:
 - Examined CPU performance factors
 - Instruction count, clock rate, and cycle per instruction (CPI)
 - Examined the instruction set for MIPS and how high-level languages are compiled into machine code
 - Examined how computers do arithmetic
- Now, we'll look inside the processor to see how it's implemented
 - Start by looking at a simplified MIPS processor
 - Includes only a basic subset of instructions (lw, sw, add, sub, and, or, slt, beq, j)
 - Later, we'll look at a more realistic MIPS processor
 - Includes more instructions and a **pipeline**

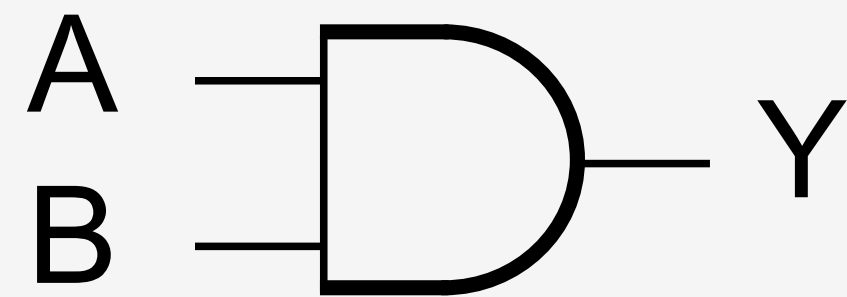
Logic Design Basics

- All information is encoded in binary
 - Low voltage = 0, High voltage = 1
 - One wire per bit
 - Multi-bit data encoded on multi-wire buses (e.g. 32-bit values travel on 32-bit buses)
- Two different types of logic elements in hardware designs
 - **Combinational elements** (a.k.a. combinational logic)
 - Operate on data values
 - Output is a function of input (same input always produces same output)
 - **State elements** (a.k.a. sequential logic)
 - Store information (e.g. registers and data memory)
 - Can be written/updated to update state

Combinational Elements

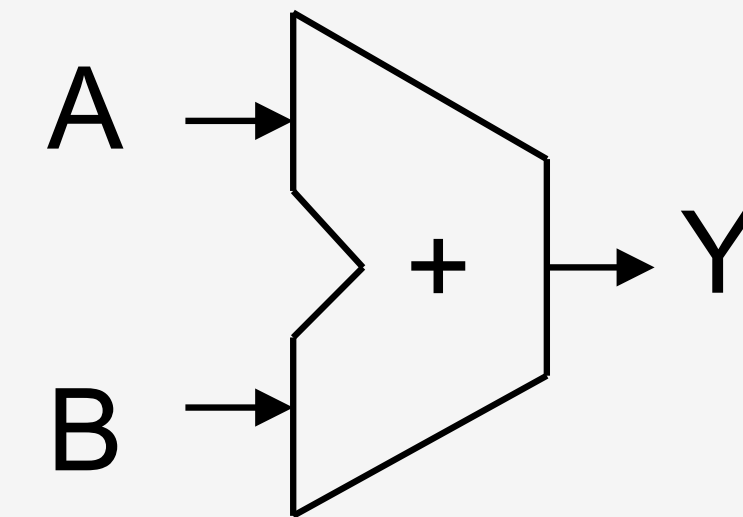
AND Gate

$$Y = A \& B$$



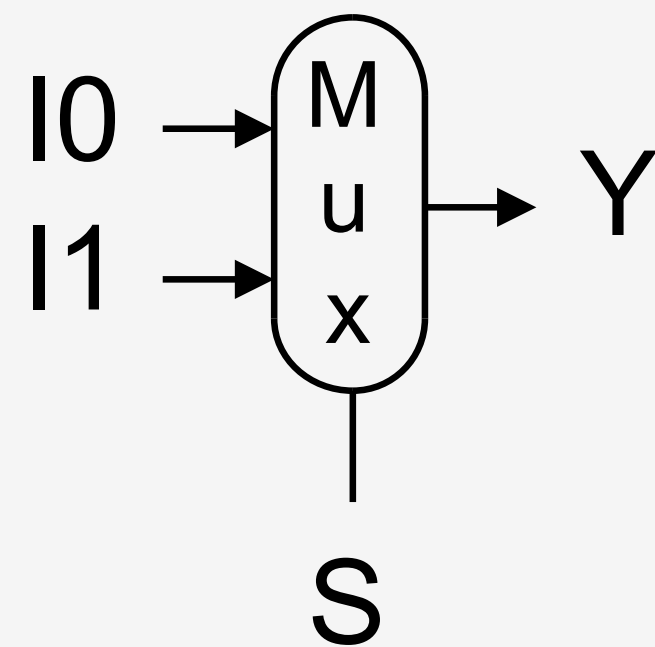
Adder

$$Y = A + B$$



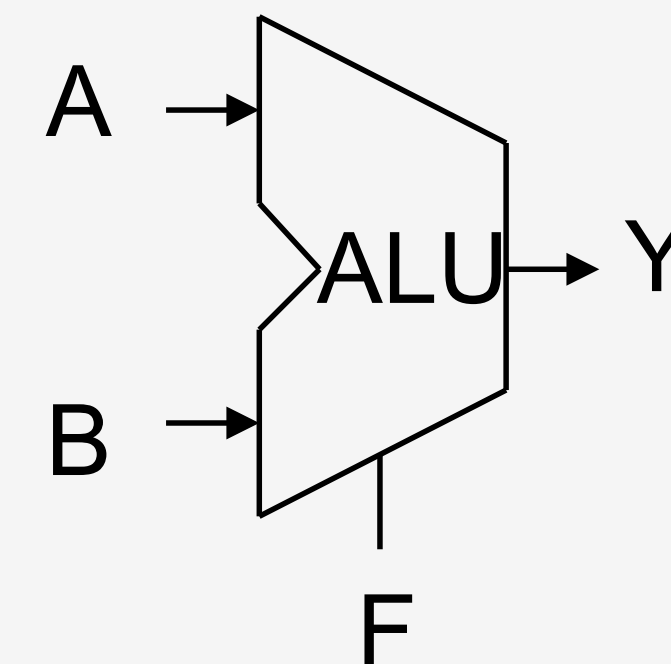
Multiplexer

$$Y = S ? I1 : I0$$



Arithmetic Logic Unit

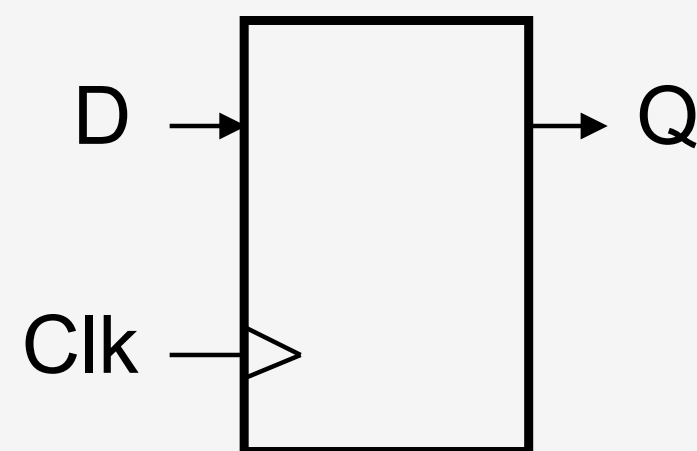
$$Y = F(A, B)$$



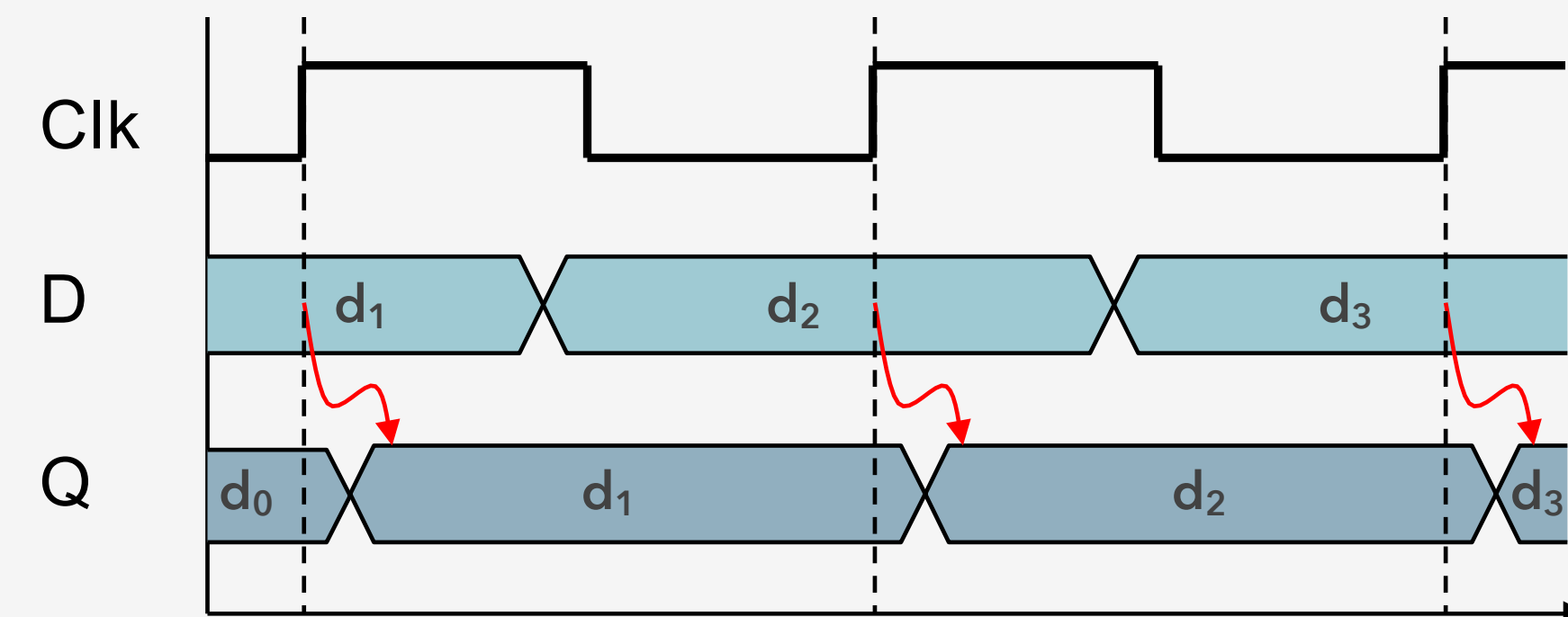
Sequential Elements (State Elements)

- A **register** stores data in a circuit (i.e. it maintains the state of a computation)
 - Uses a clock signal to determine when to update the stored value
 - An **edge-triggered** register updates its contents on a clock edge
 - **Positive edge-triggered** registers update when the clock changes from 0 to 1
 - **Negative edge-triggered** registers update when the clock changes from 1 to 0

A basic storage element, a positive edge-triggered D-type flip-flop; it can store a *single bit* of data
Inputs are D and Clk; Output is Q



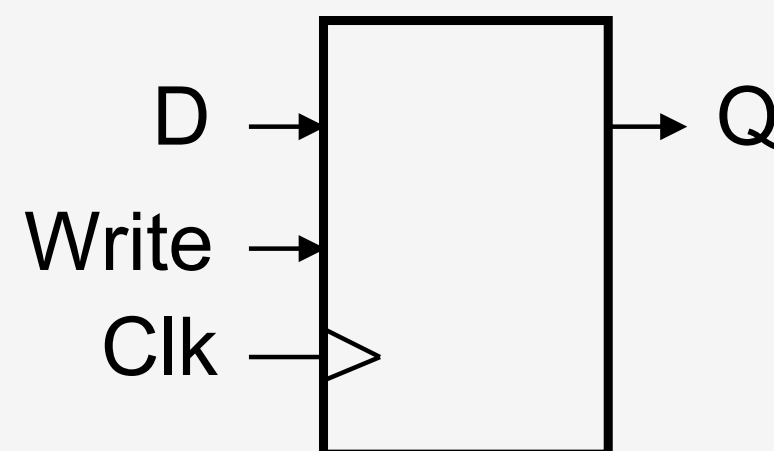
Shortly after a positive clock edge, the input D is stored in the flip-flop and it becomes visible on the output Q.



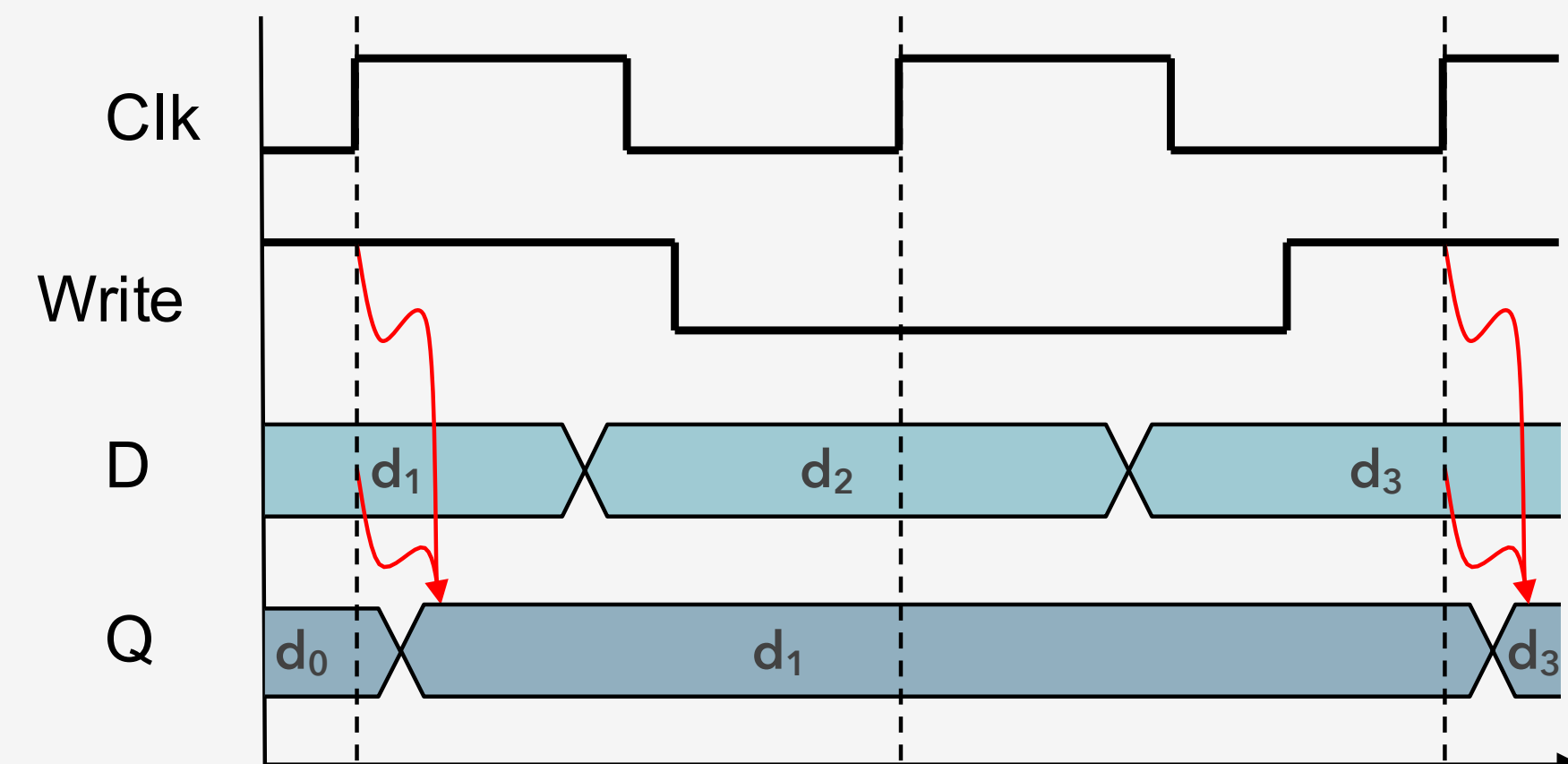
Sequential Elements with More Control

- The clock signal is *ALWAYS* ticking and may not always want to store new data in a register
- A register with write control allows data to be stored only when directed
 - Only updates register contents on clock edge AND when a **write enable** input is 1
 - Typically used when a stored value is still required

A positive edge-triggered D-type flip-flop with write enable input

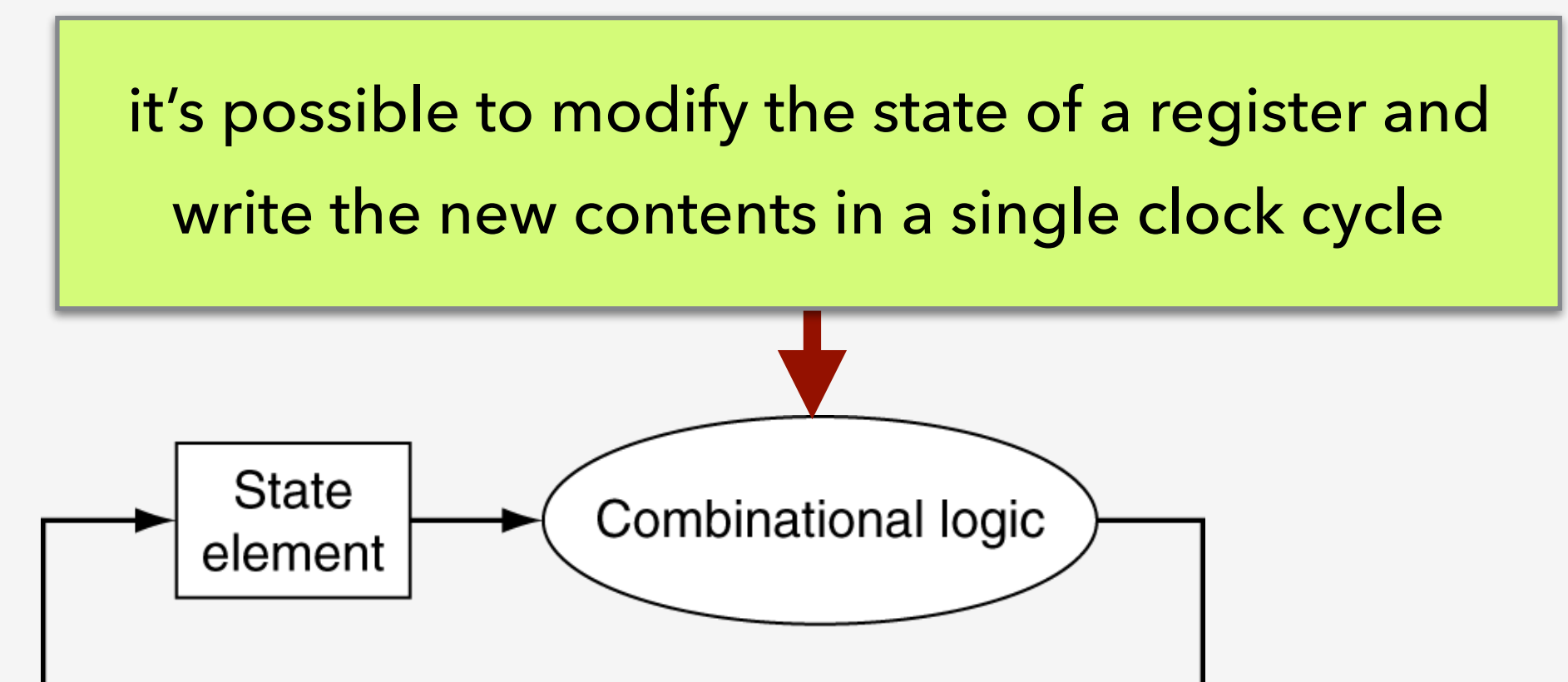
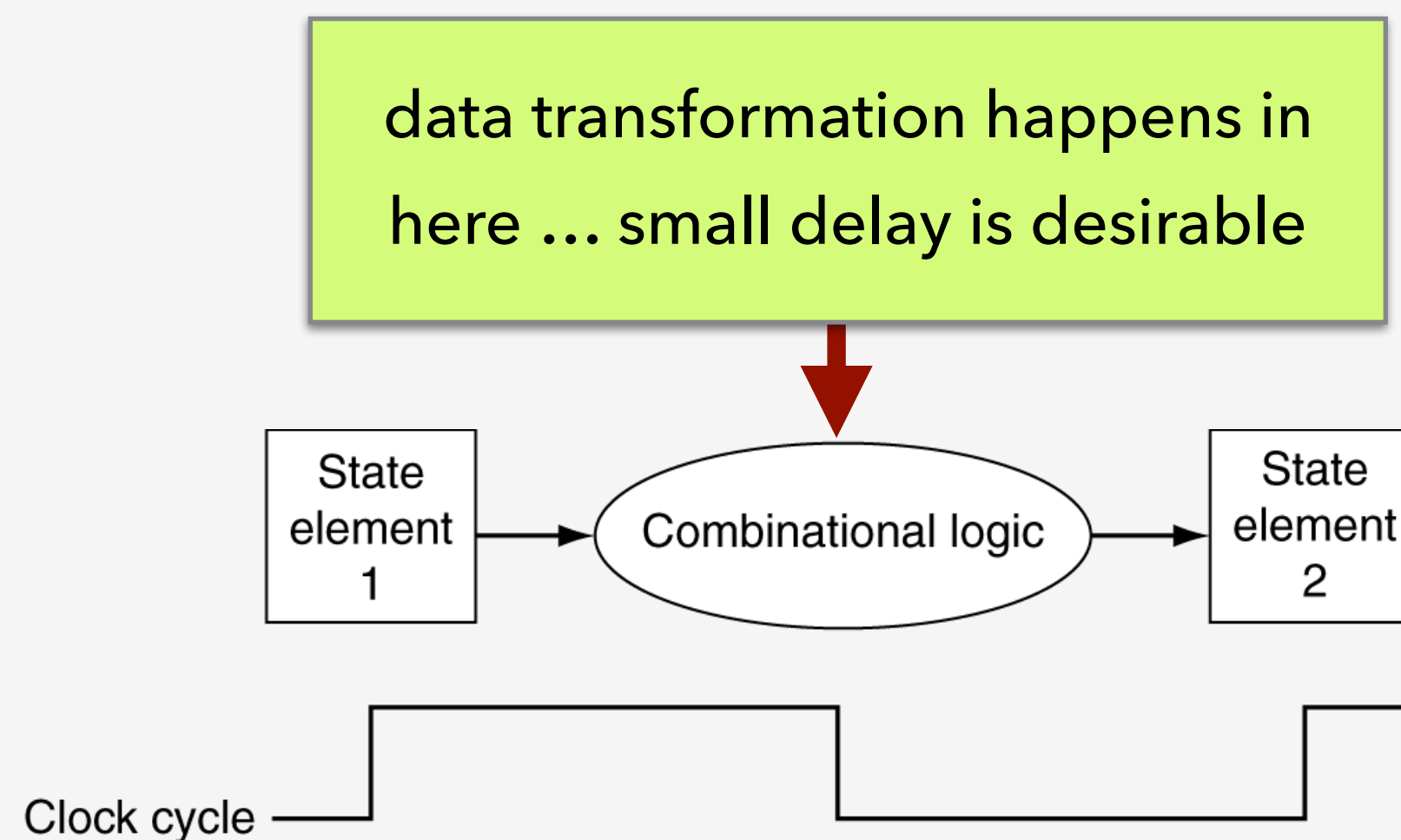


Shortly after a positive clock edge, the input D is stored in the flip-flop IF the write enable was asserted at the clock edge



Clocking Methodology

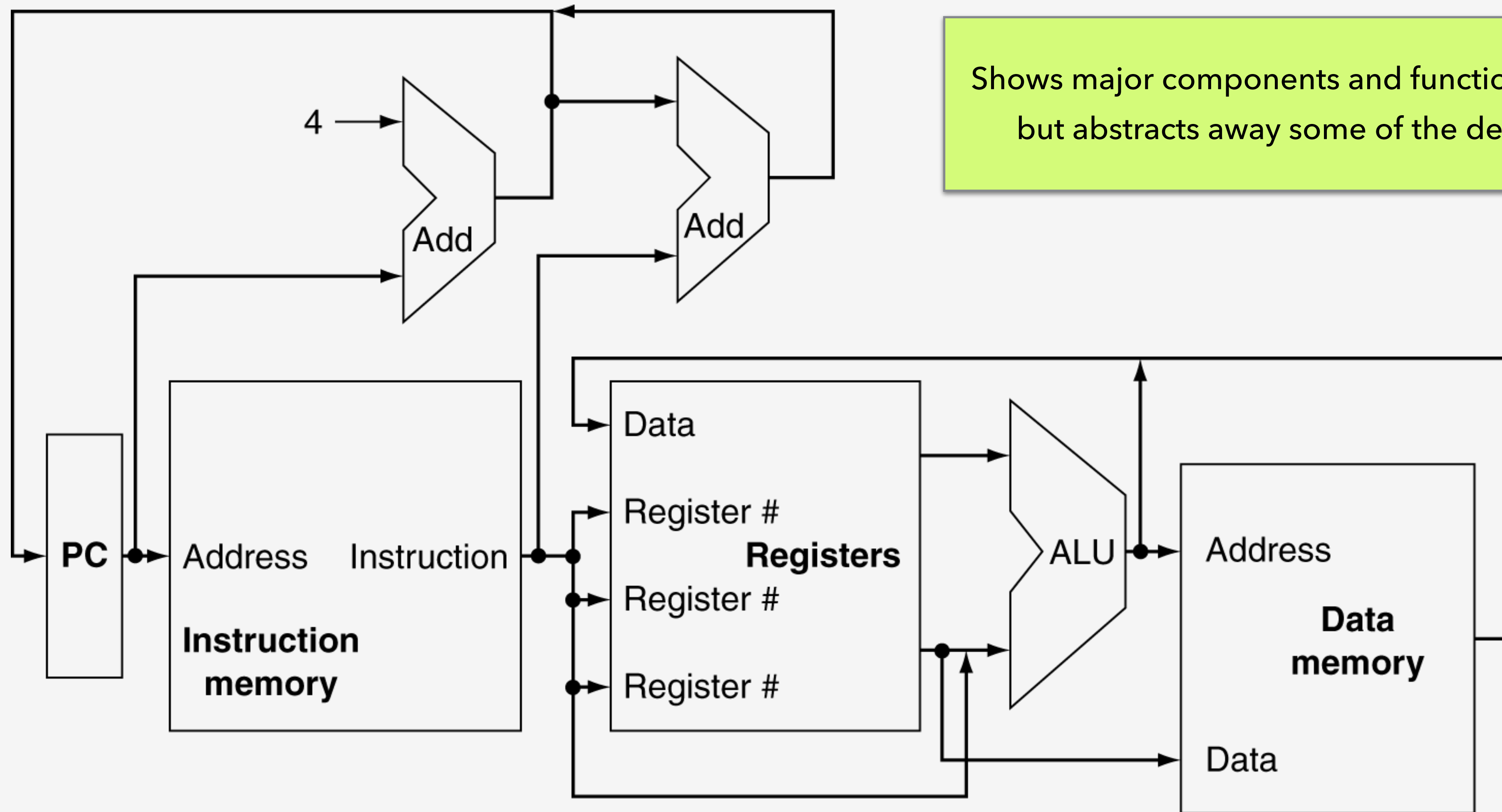
- Combinational logic transforms data during clock cycles
 - Between clock edges
 - Receives input data from state elements, sends new output to state elements
 - Combinational logic introduces delay (takes some time for data to pass through an AND gate!)
 - Total delay through combinational logic is sum of delay through all elements (and wire) on path
 - Longest combination logic delay determines clock period for **entire** processor!!



Executing Instructions on a Processor

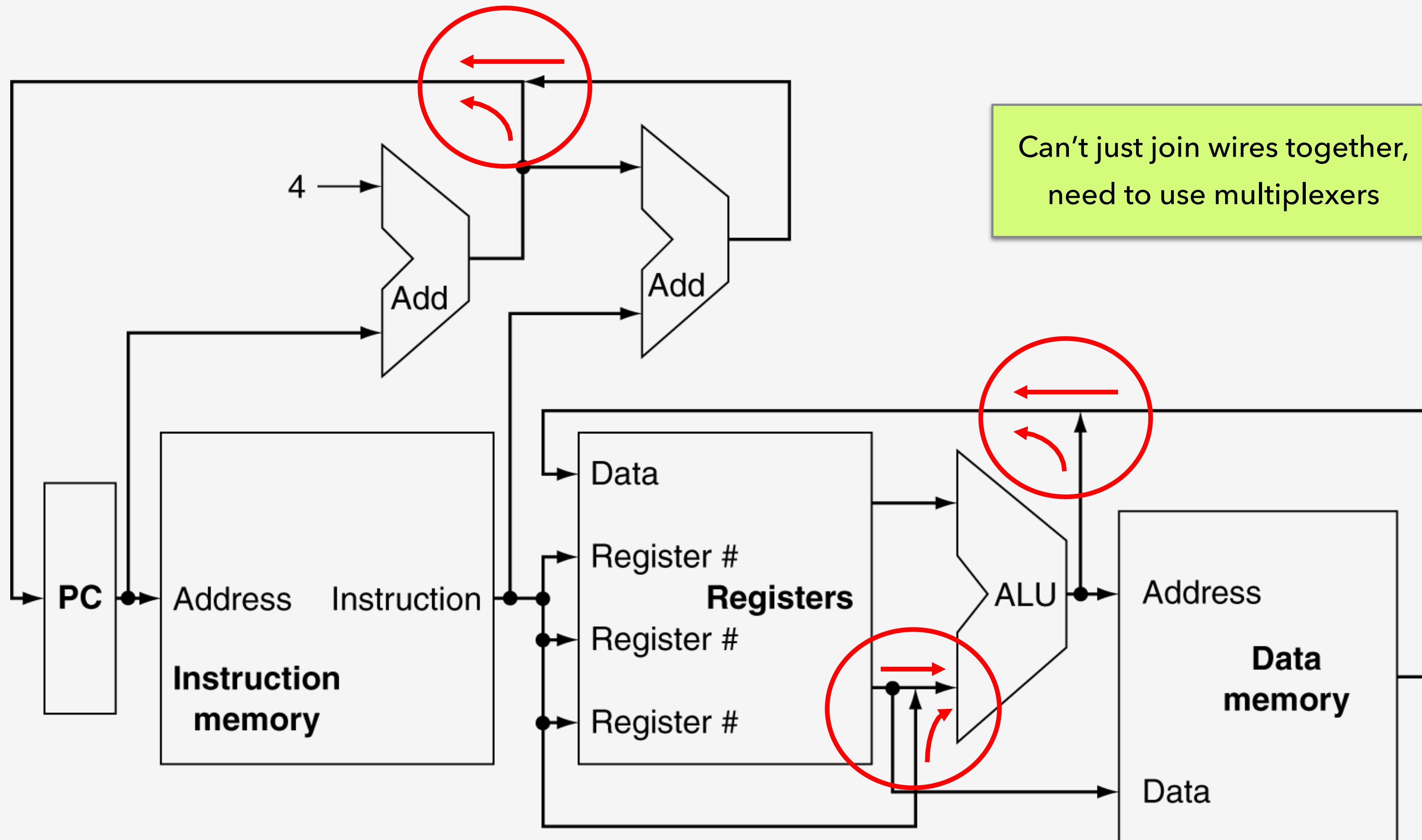
- For ***all instructions***, the first two steps are the same:
 - Use the Program Counter (PC) to access program memory and fetch an instruction
 - Read the source registers (one or two) to be used for the instruction – encoded into instruction
- Depending on instruction class ...
 - Use the ALU to compute one of the following:
 - An arithmetic result (e.g. add, sub)
 - A memory address for a load/store (e.g. lw, sw)
 - A branch target address (e.g. beq)
 - Access data memory for load/store (only done for load/store instruction)
 - Update the PC
 - $PC = PC + 4$ OR $PC = \text{some branch target address}$

CPU Overview



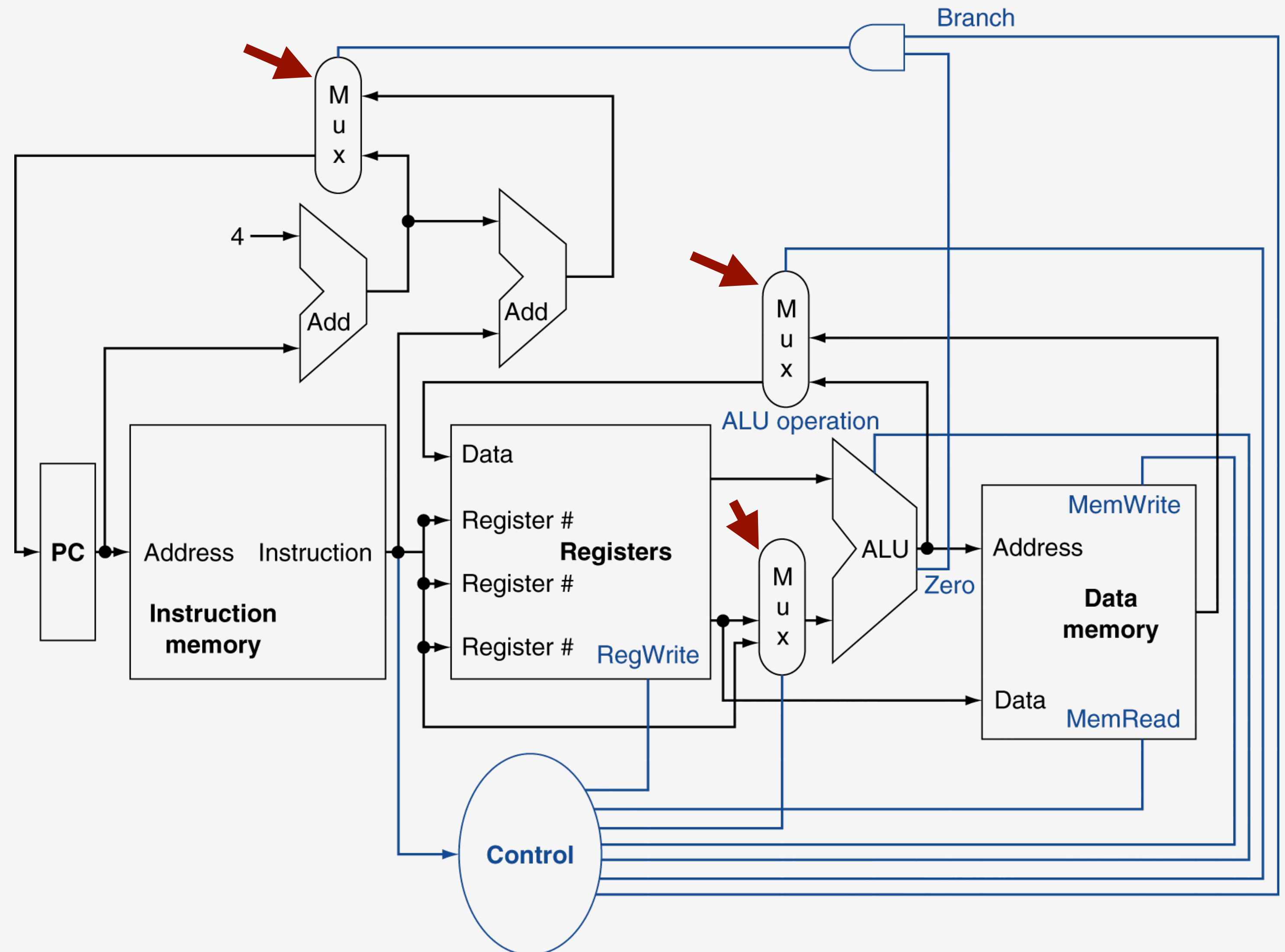
Shows major components and functionality,
but abstracts away some of the detail

CPU Overview (continued)



CPU Overview with Multiplexers and Control

- Multiplexers use a control line to select an output from multiple different inputs
 - Three multiplexers added for this simplified version of MIPS
- Control uses information from the instruction word to assert/deassert various control lines
 - Steers data through muxes
 - Write enables for registers
 - Read/Write enable for memory
 - Specify ALU operation

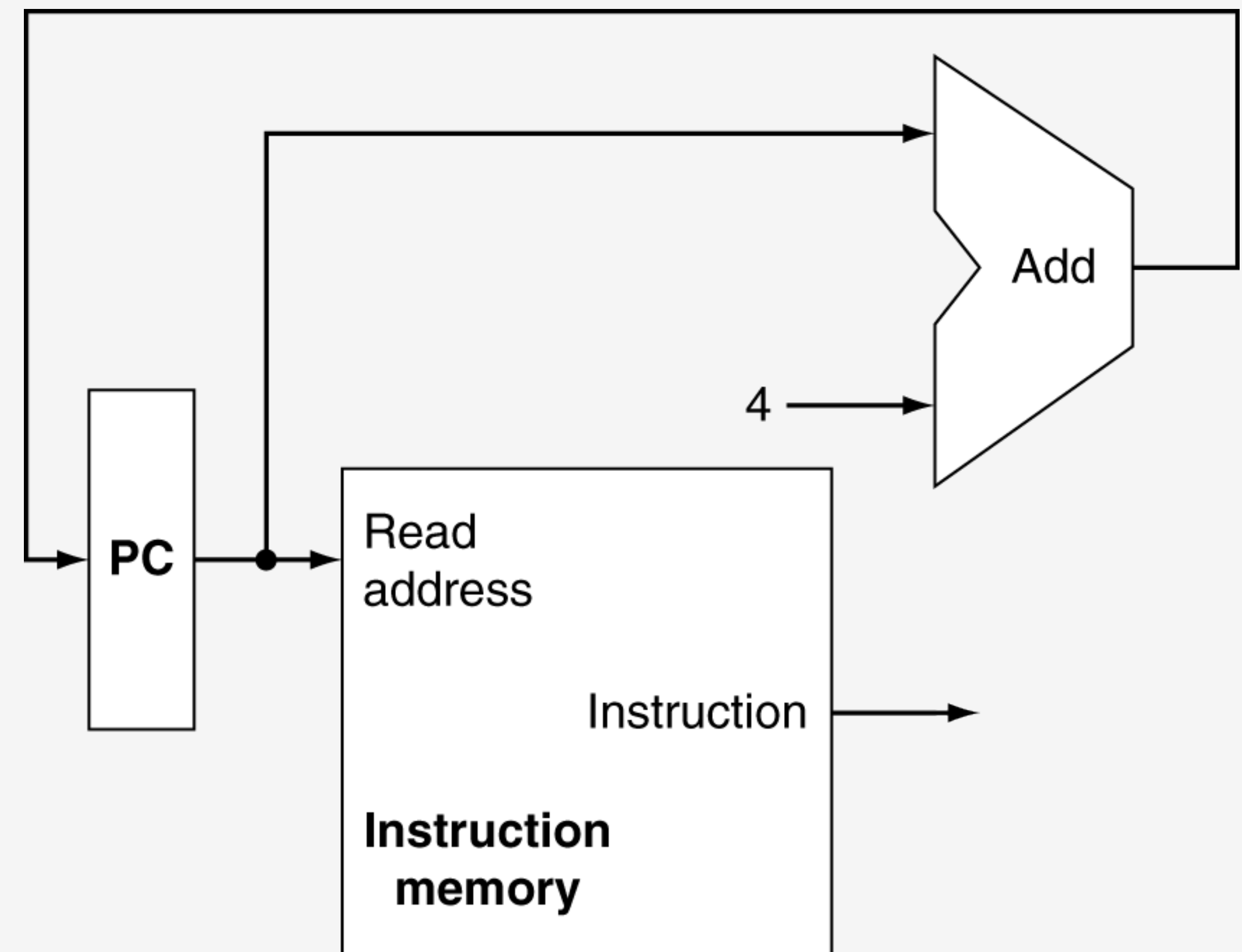


Building a Datapath

- **Datapath element** is a unit used to operate on or store data within a processor
 - Processor datapath is made up of multiple datapath elements
 - Registers, ALUs, multiplexers, memories, etc.
- We will incrementally build a datapath for a simplified MIPS processor
 - Examine how each datapath element is used
 - Examine which datapath elements are used for which instructions

Beginning of Datapath – Instruction Fetch

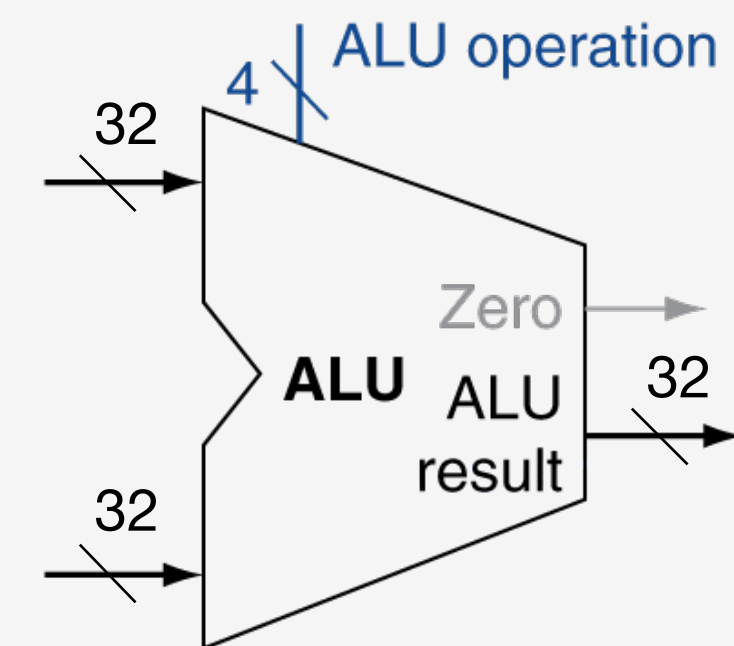
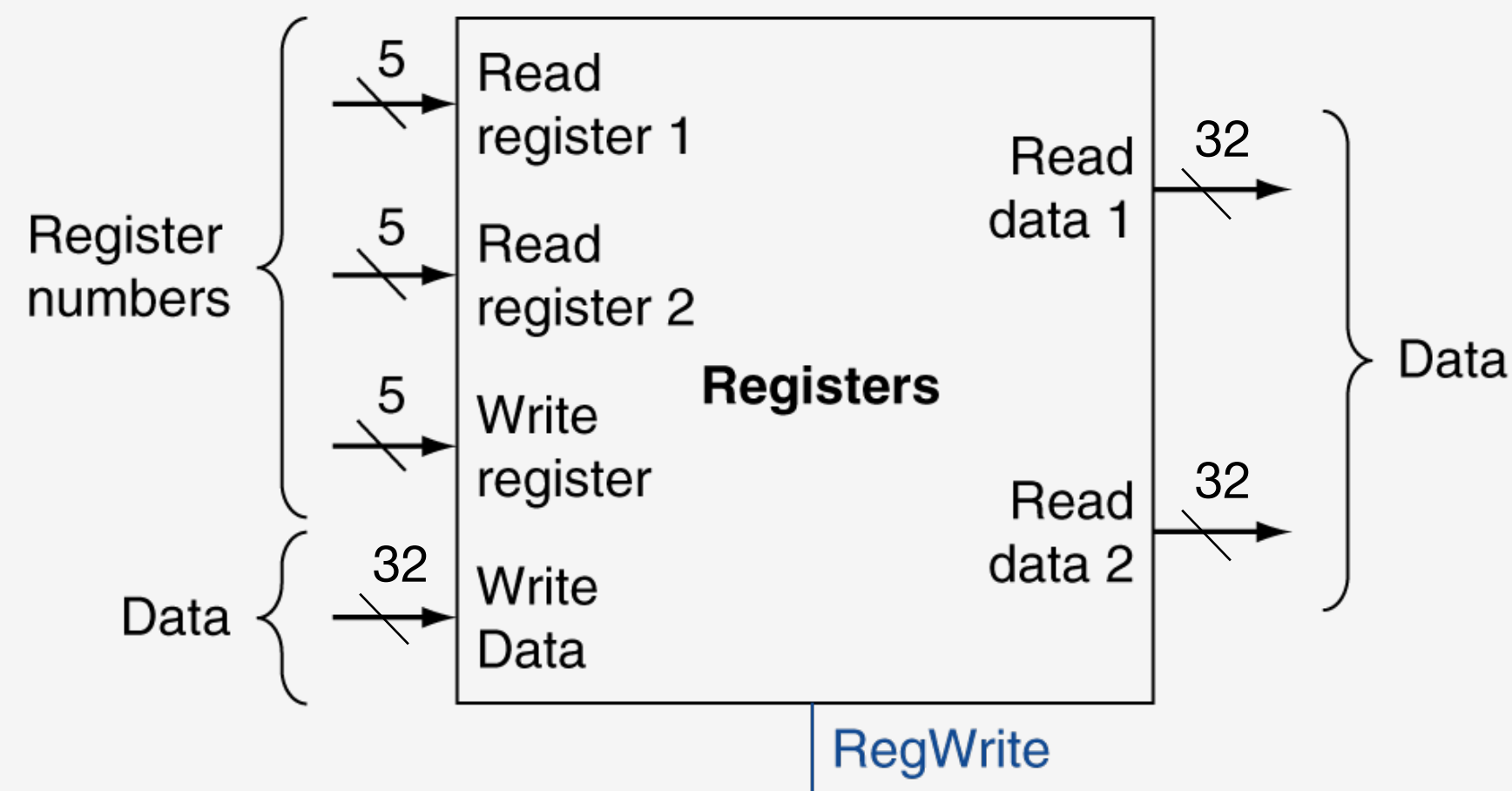
- Recall first step for all instructions:
 - Use the Program Counter (PC) to access program memory and fetch an instruction
 - **Need a Program Counter (PC)**
 - 32-bit register
 - Points to program memory
 - **Need a memory**
 - Stores program
 - **Need an ALU (or simple adder) to increment the PC**
 - Increments PC by 4 to point to next instruction



Supporting the R-Type Instruction Format

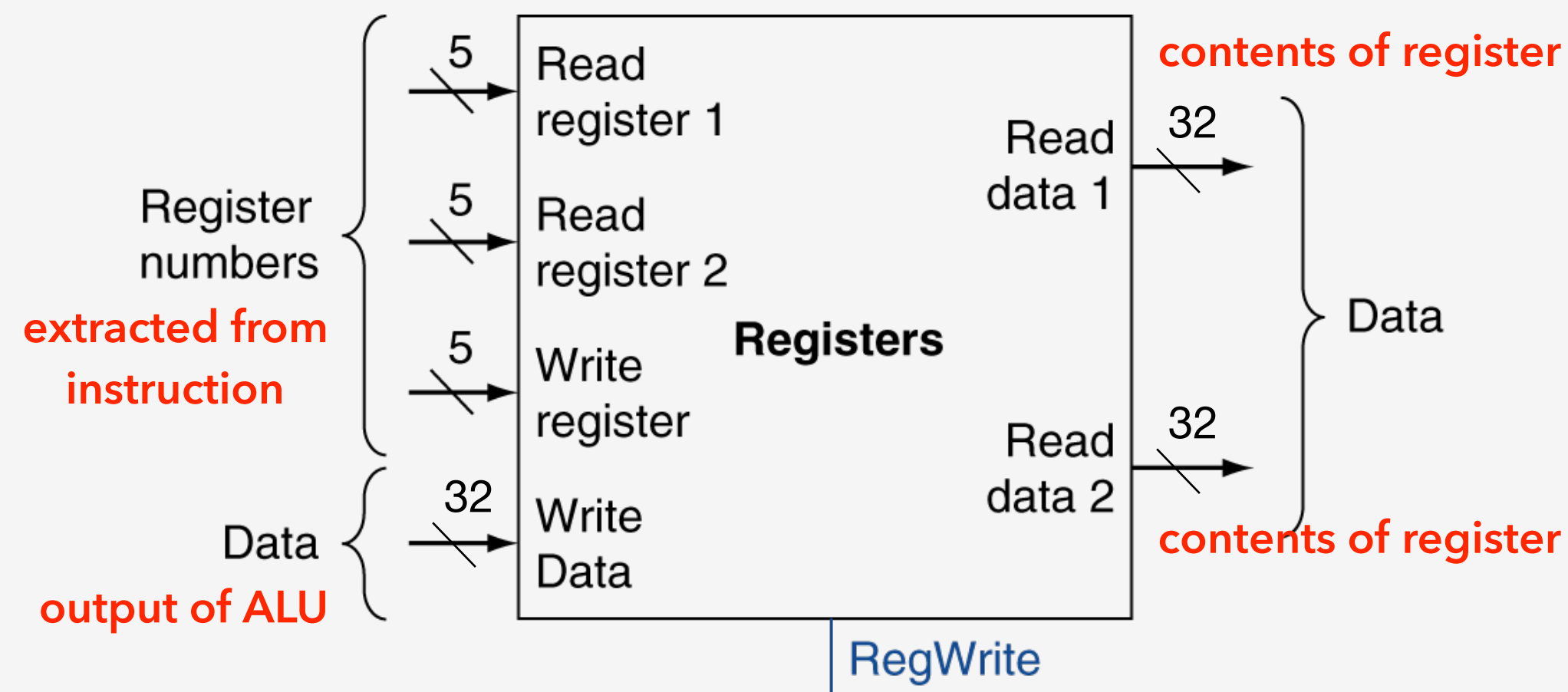
- For R-Type instructions, read two source registers, perform an ALU operation, and write the result to destination register
- Need a collection of registers, called a **register file**
- Need an ALU to perform arithmetic ops
- A control signal specifies operation the to perform

Example Instruction:
add \$t0, \$t1, \$t2

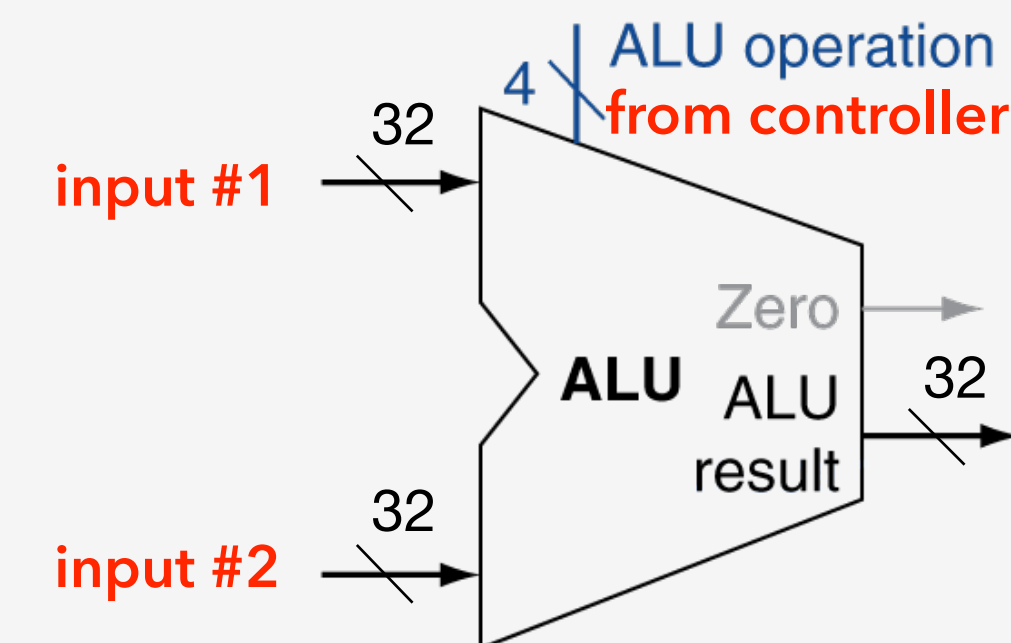


Supporting the R-Type Instruction Format (continued)

- Register File in detail:
 - Two 5-bit inputs specify source registers
 - Values stored in the specified source registers are output on "Read data 1/2"
 - One 5-bit input specifies destination register
 - Input "Write Data" is stored in "Write register" when write enable (RegWrite) is asserted



- ALU in detail:
 - Has two 32-bit inputs, one for each source operand
 - Has a single 32-bit output for the result of the arithmetic operation
 - A control signal specifies the ALU operation to perform
 - "Zero" output is asserted anytime a result is 0



Supporting Load/Store Instructions

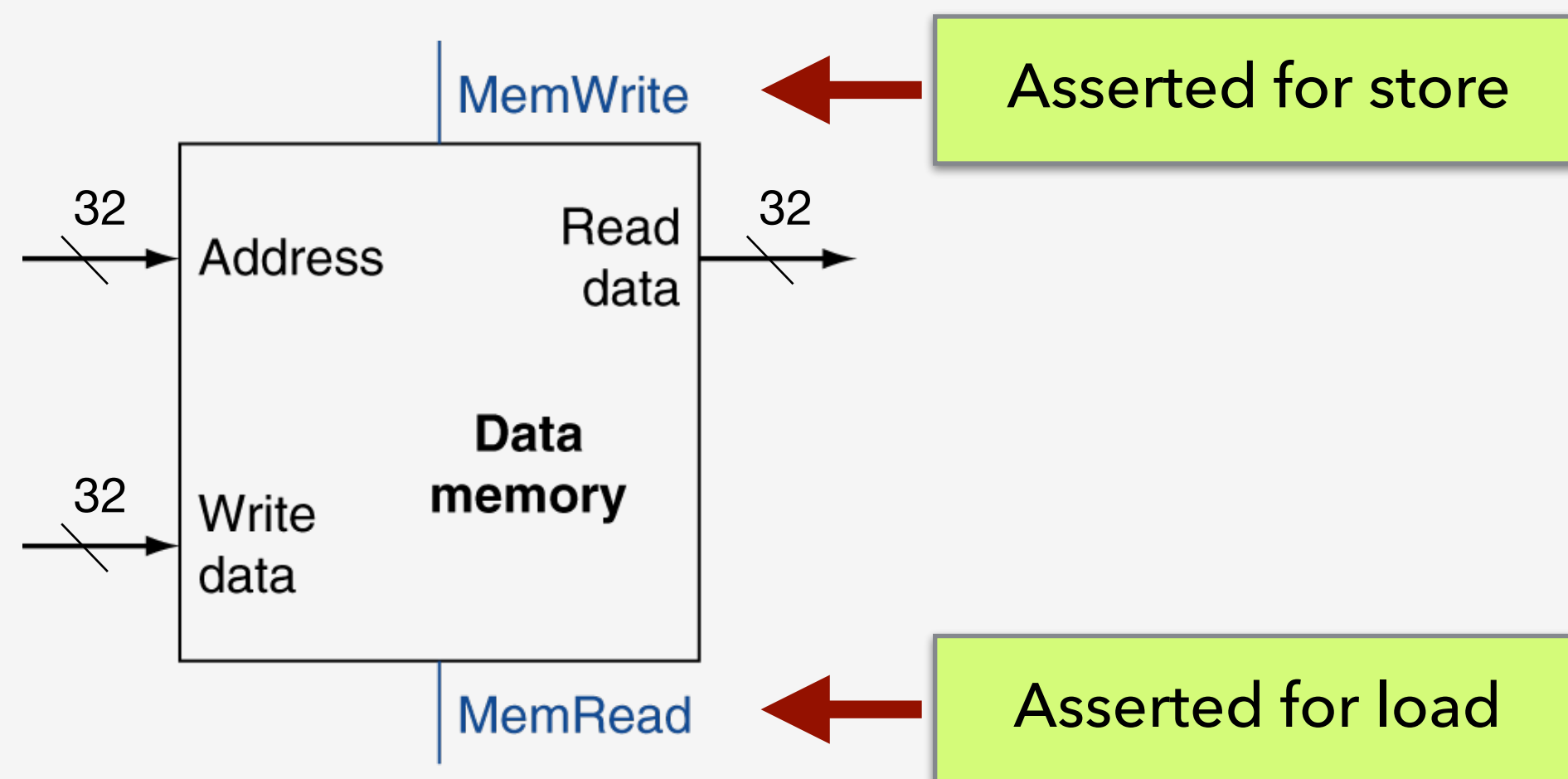
- Need a memory to write/read data to/from
- Must read register operands (two registers for a store, only one register for a load)
 - One register contains the base memory address to read/write
 - In case of store instruction, second register contains data to be written to memory
 - In case of load instruction, second register is destination for data read from memory
- Must calculate effective address using the 16-bit offset embedded in the instruction word
 - First, sign-extend the 16-bit offset (*will need a sign-extension unit!*)
 - Next, use the ALU to add the offset to the register containing the base address

Example Load Instruction:
`lw $t0, 4($t1)`

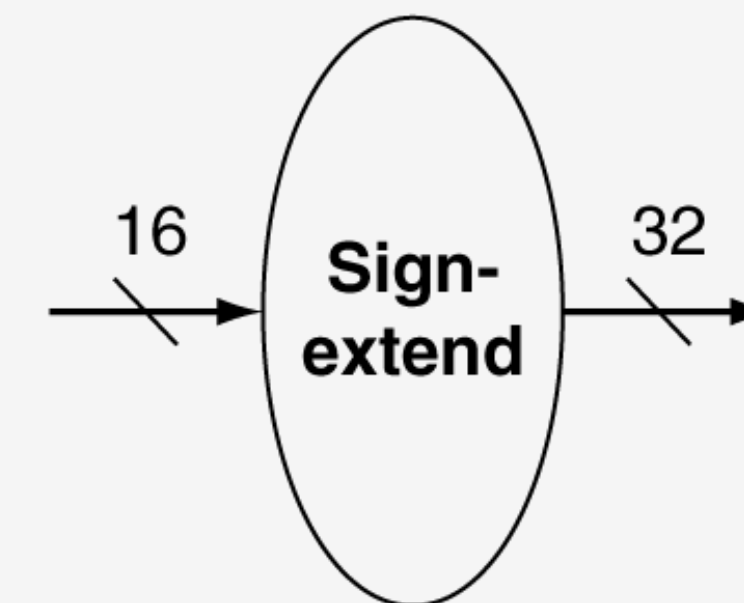
Example Store Instruction:
`sw $s5, 16($s1)`

Supporting Load/Store Instructions

- Data memory in detail:
 - Address input specifies the address that should be read/written
 - "Write data" input has data to be written during a store instruction
 - Ignored during a load instruction
 - "Read data" output has data read during load



- Sign-extension unit in detail:
 - Single 16-bit input
 - Single 32-bit output
 - Simply duplicates MSB of input (bit 15) into bits 31 down to 16 of output



Supporting Branch Instructions (only beq for now)

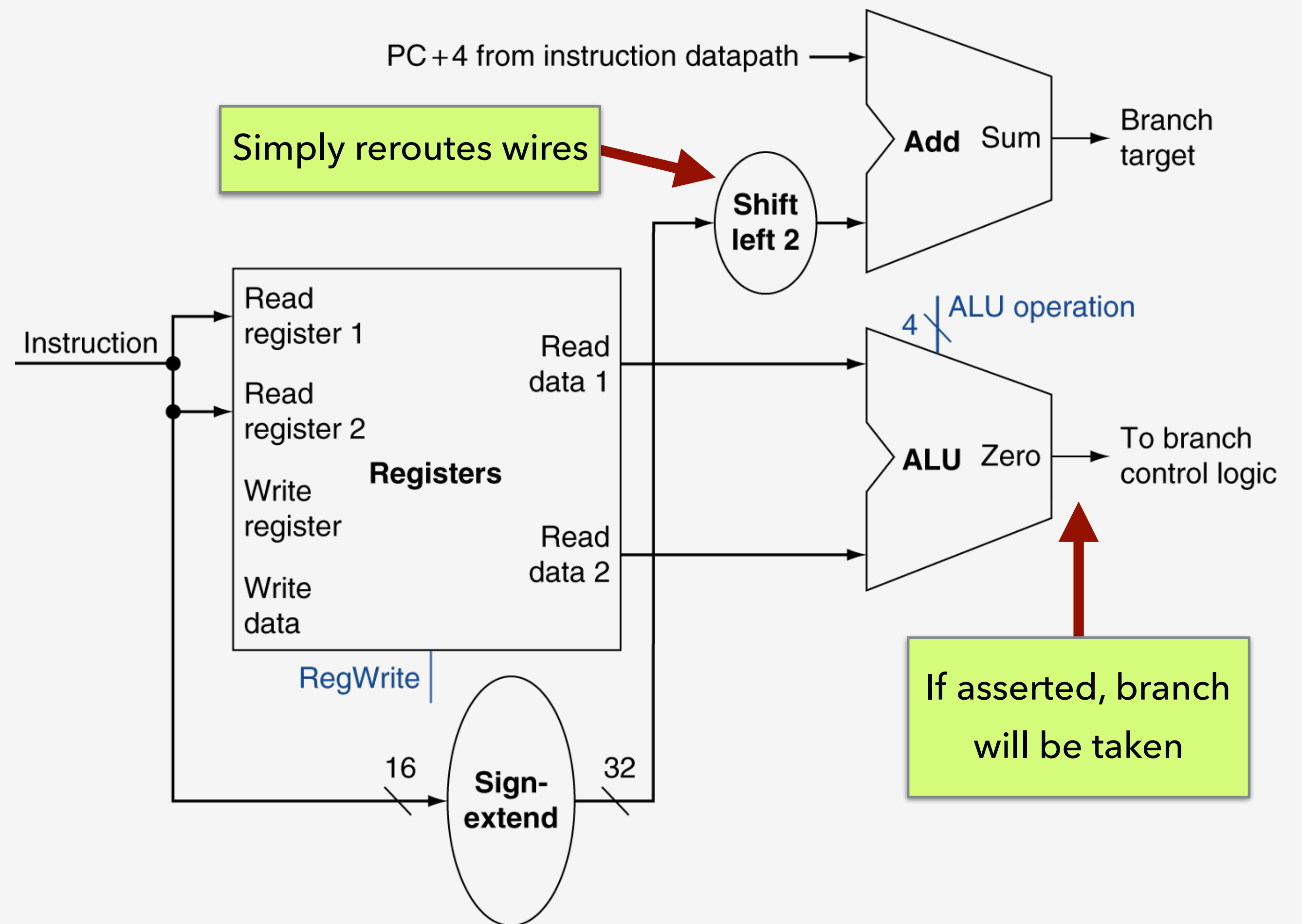
- Compares two source registers

Example Instruction:
beq \$t1, \$t2, offset

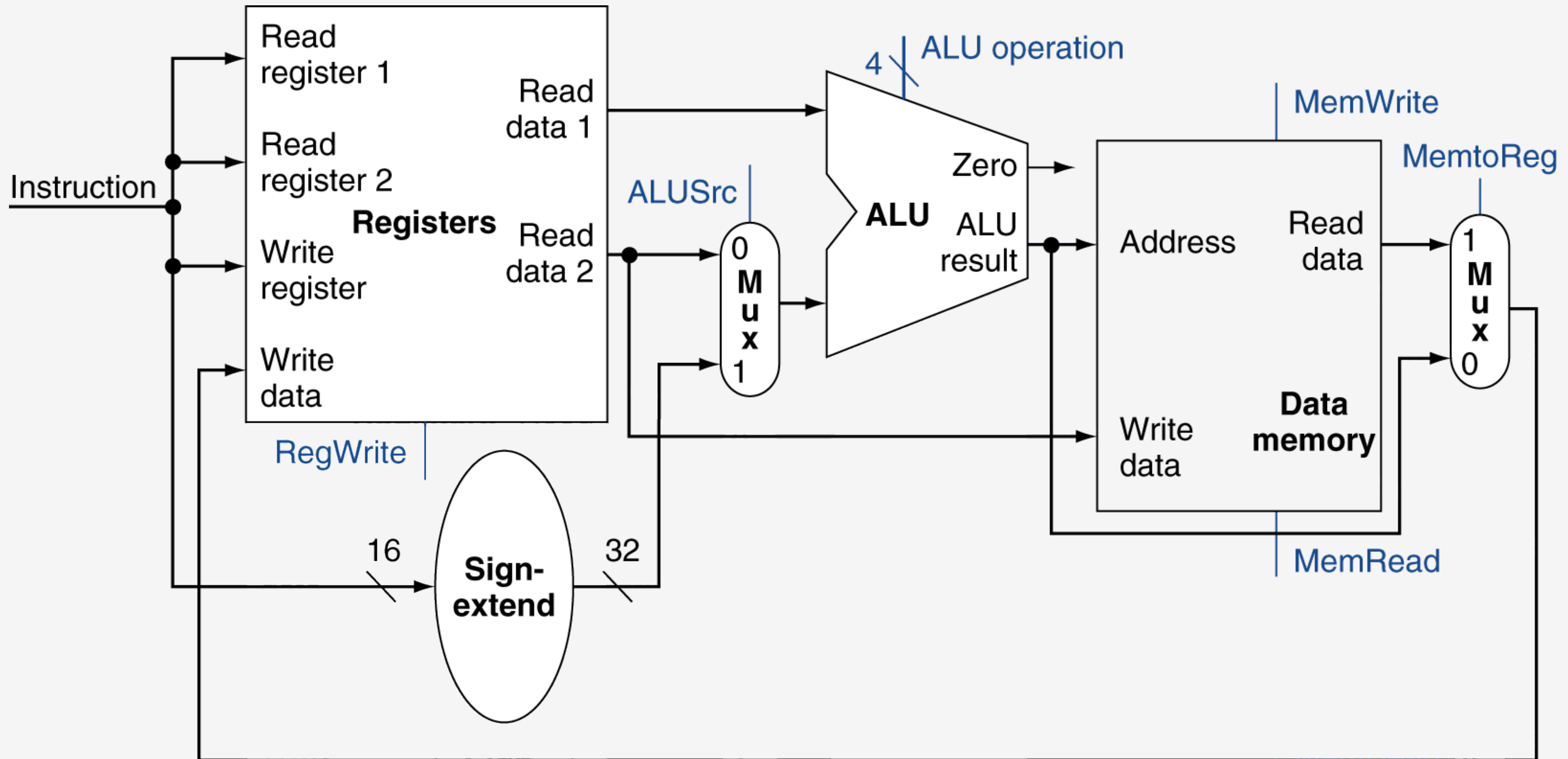
- Instruction word contains 16-bit offset

BranchAddr =
{ 14{immediate[15]}, immediate, 2'b0 }
 $PC = PC + 4 + \text{BranchAddr}$

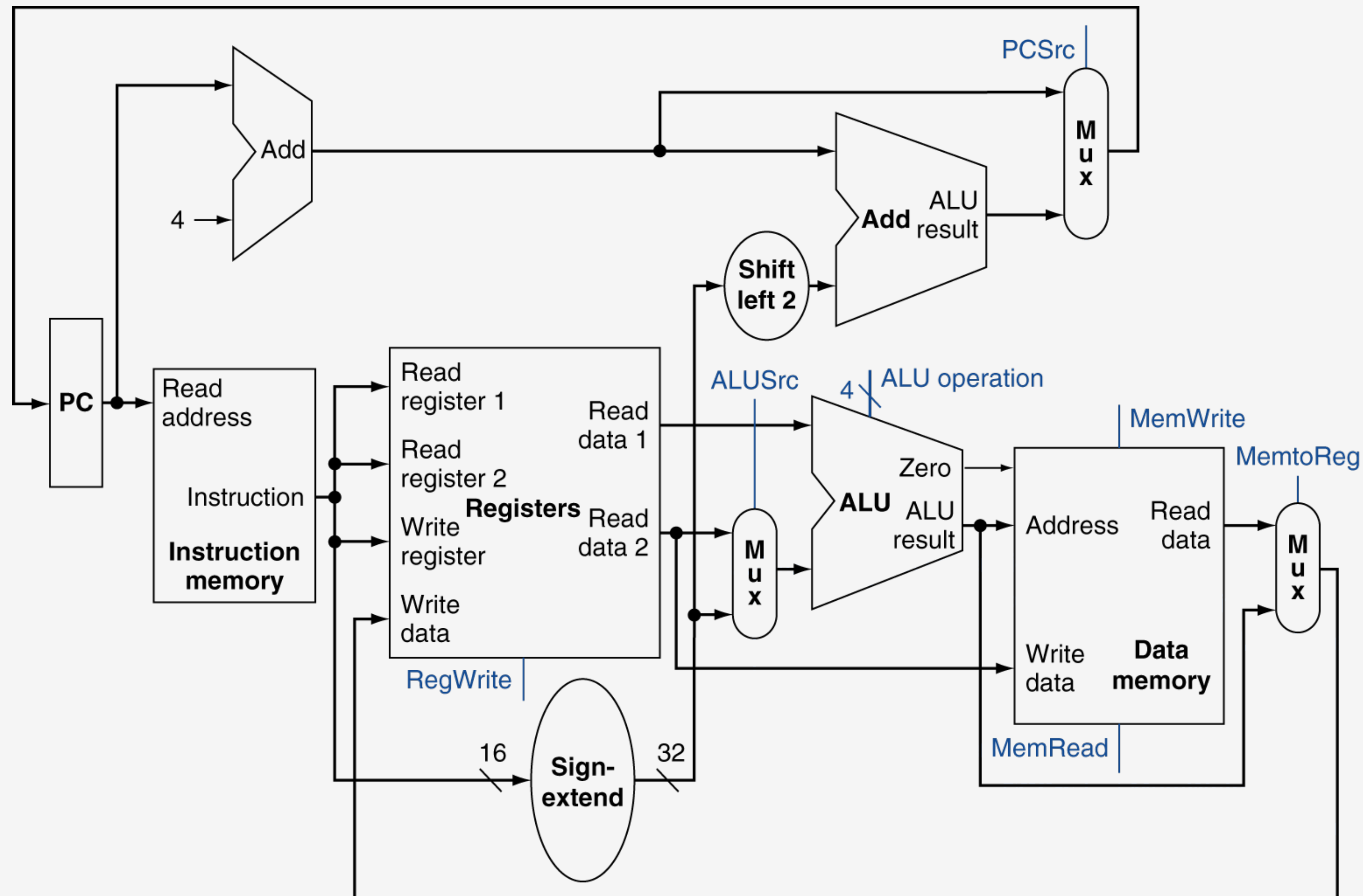
- ALU is used to determine if inputs are equal
 - Subtract sourceReg1 from sourceReg2
If result is 0, regs are equal, take branch
Otherwise, $PC = PC + 4$
- An Adder computes "Branch Target" address if case branch is taken



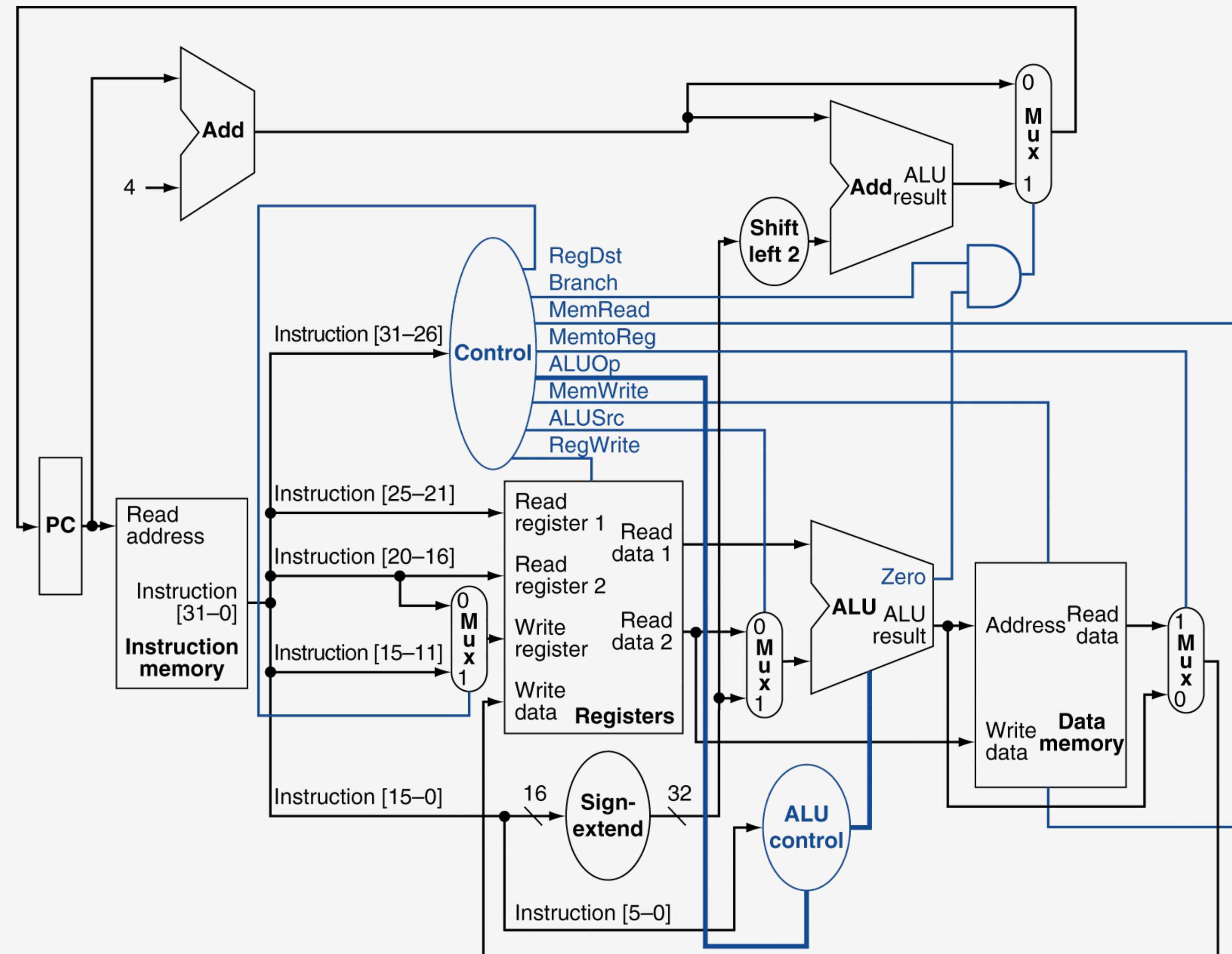
Putting it all together: R-Type/Load/Store Datapath



Complete Datapath for Simplified MIPS Processor



Complete Datapath With Control



The Main Control Unit

- Input is 6-bit opcode from 32-bit instruction
 - Control signals are all derived from the instruction
- Output includes many control signals
 - Control signals for various muxes
 - ALUOp for controlling various muxes and the ALU control unit
 - Read/Write signal to data memory
 - Write signal to register file
 - Branch instruction signal

- Bits 31:26 => opcode
- Bits 25:21 => source reg for all three ops
- Bits 20:16 => dest reg for load, source reg for store, branch, and R-type
- Bits 15:11 => dest reg for R-type
- Bits 15:0 => sign extend and add for branch and load/store

| | | | | | | |
|-------------------|----------|-------|-------|---------------|-------|-------|
| R-Type | 0 | rs | rt | rd | shamt | funct |
| | 31:26 | 25:21 | 20:16 | 15:11 | 10:6 | 5:0 |
| Load/Store | 35 or 43 | rs | rt | 16-bit offset | | |
| I-Type | 31:26 | 25:21 | 20:16 | 15:0 | | |
| Branch | 4 | rs | rt | 16-bit offset | | |
| I-Type | 31:26 | 25:21 | 20:16 | 15:0 | | |

Controlling the ALU

- ALU needs to perform different operations depending on the instruction
 - For load/store instructions: ALU performs add
 - For branch instructions: ALU performs subtract
 - For R-type instructions: ALU performs operation specified in funct field (AND, OR, add, sub, ...)
- 4-bit "ALU Operation" input to ALU tells ALU what operation to perform
 - Need an ALU control unit to generate the 4-bit input to the ALU

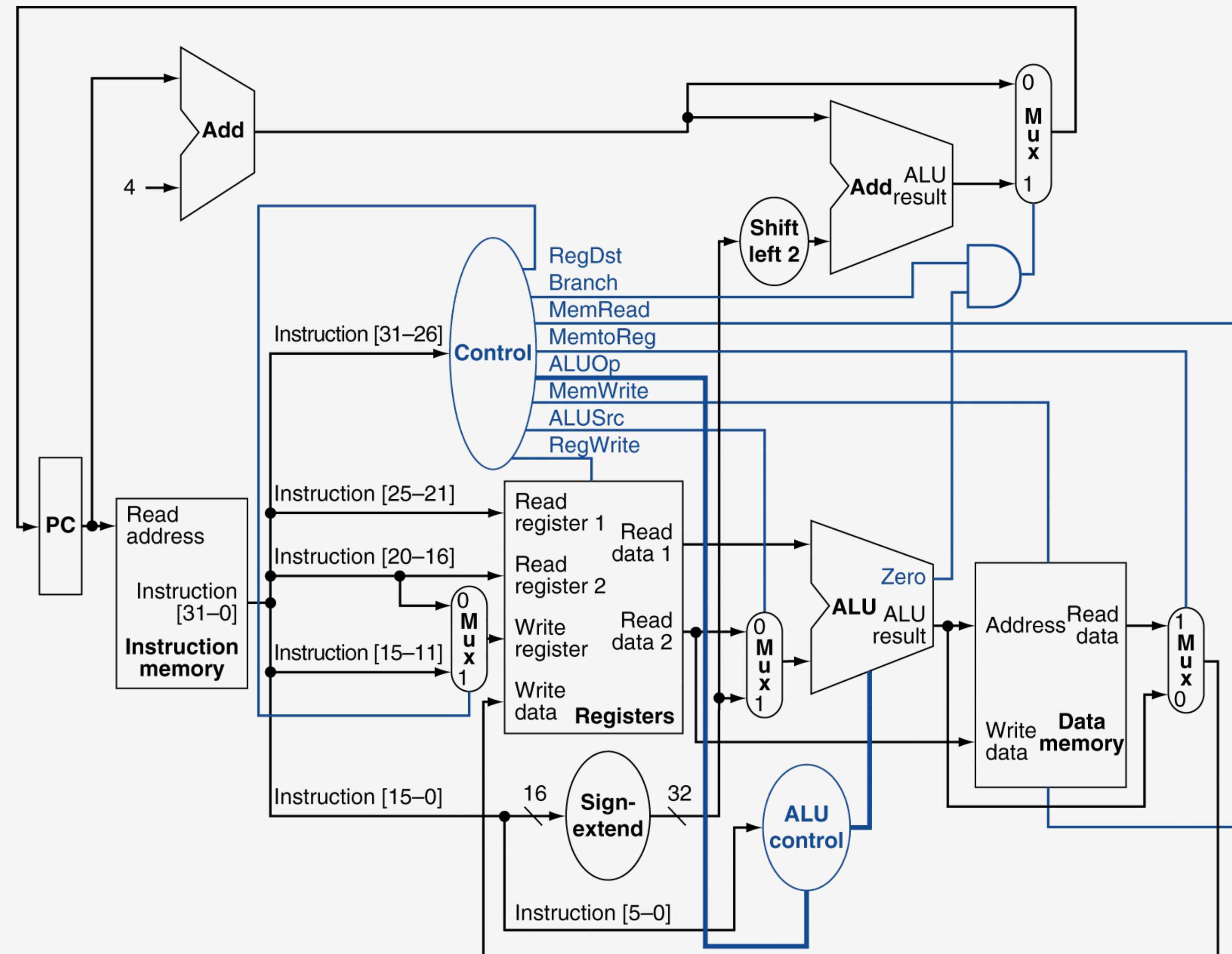
| ALU control lines | Function |
|-------------------|------------------|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | set on less than |
| 1100 | NOR |

The ALU Control Unit

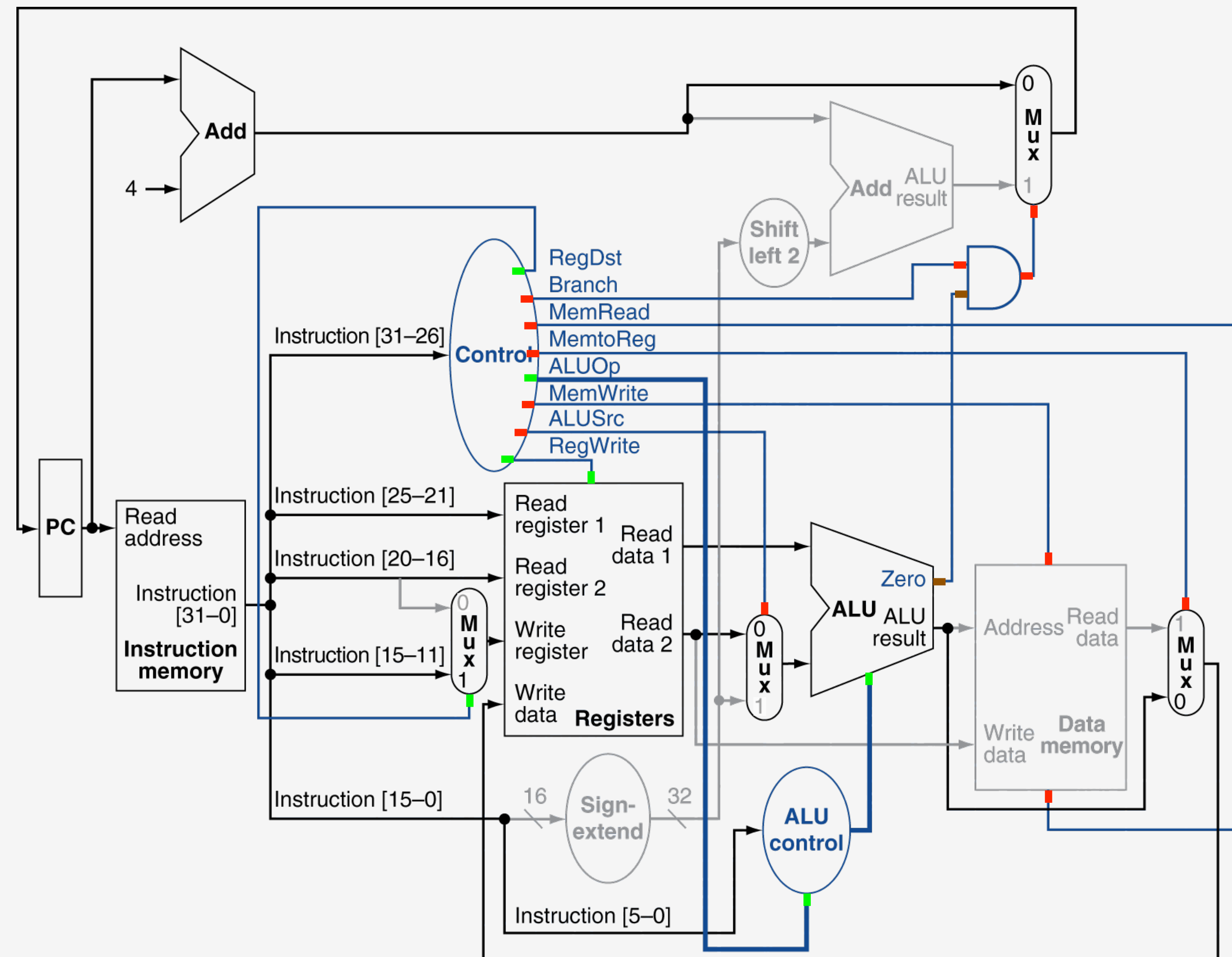
- Generates the 4-bit “ALU Operation” input for the ALU
- Inputs include:
 - 6-bit funct field directly from instruction word; used for R-type instructions only
 - Ignored for load/store/branch instructions since they don’t have a funct field
 - 2-bit ALUOp value that is generated by a main control unit
 - ALUOp is derived from the opcode in the main control unit

| Instruction opcode | ALUOp | Instruction operation | Function code | Desired ALU action | ALU control input |
|--------------------|-------|-----------------------|---------------|--------------------|-------------------|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | AND | 0000 |
| R-type | 10 | OR | 100101 | OR | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

Complete Datapath With Control (again)



Datapath During R-Type Instruction

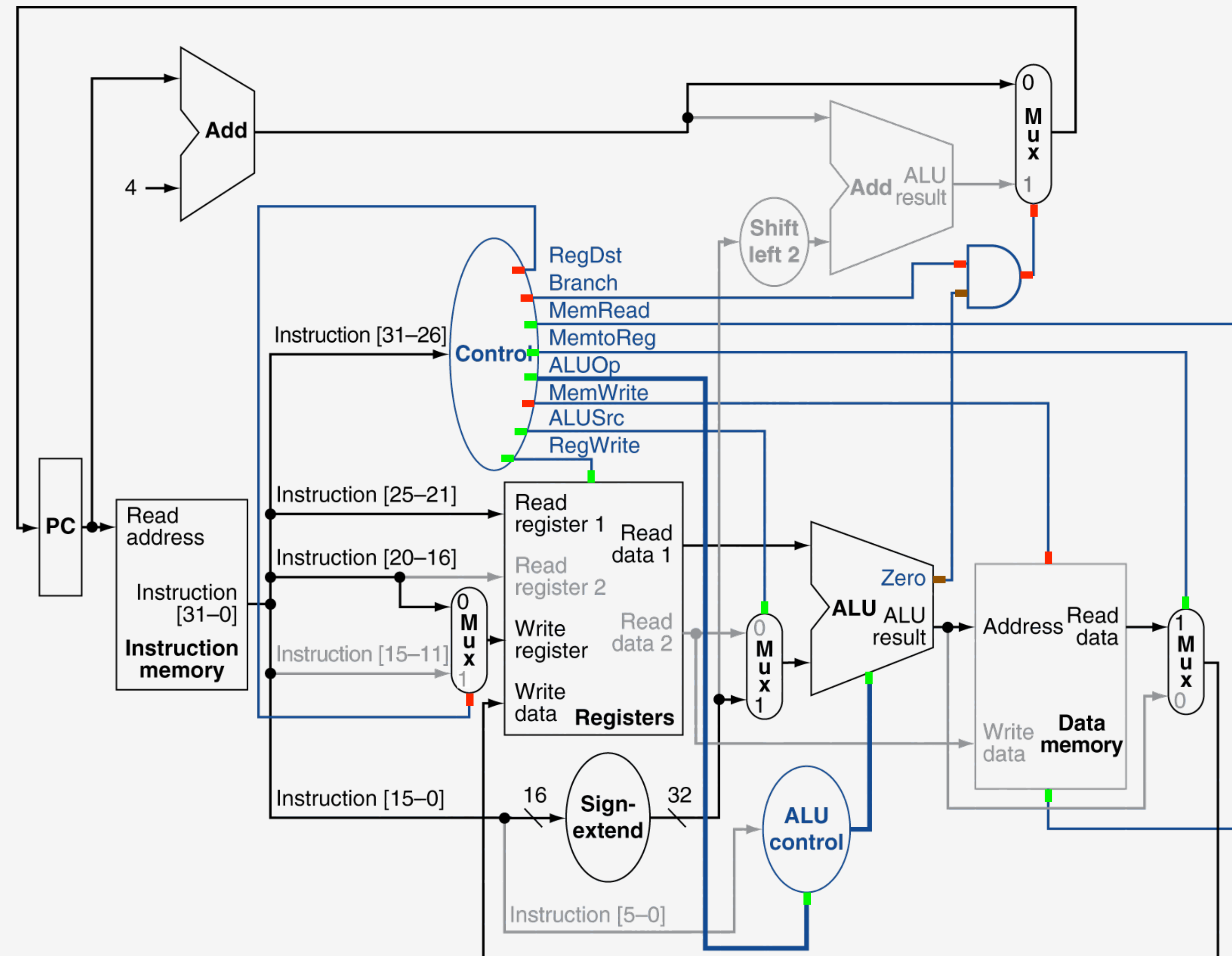


Darker colored paths and hardware units are in use. Lighter colored paths are not in use.

Control Lines:

- asserted
- not asserted
- don't care

Datapath During Load Instruction

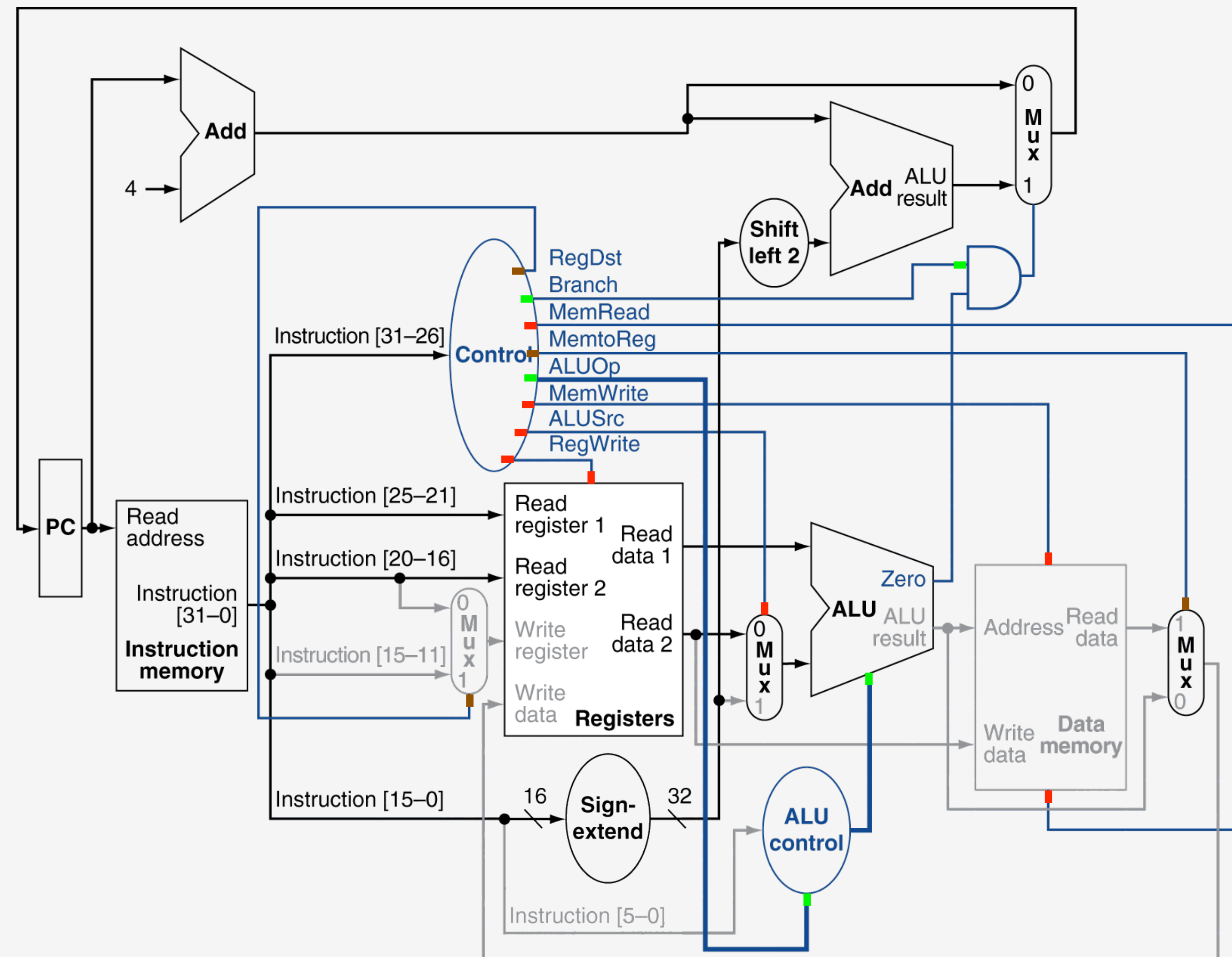


Darker colored paths and hardware units are in use. Lighter colored paths are not in use.

Control Lines:

- Green line: asserted
- Red line: not asserted
- Brown line: don't care

Datapath During Branch-on-Equal Instruction



Darker colored paths and hardware units are in use. Lighter colored paths are not in use.

Control Lines:
— asserted
— not asserted
— don't care

Zero output of ALU may or may not be asserted.