

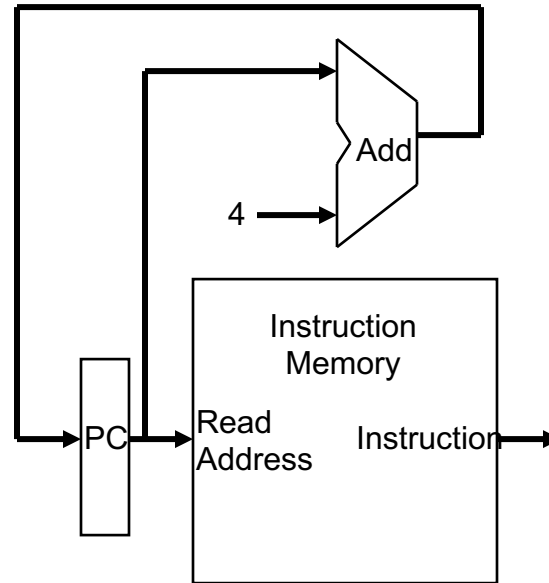
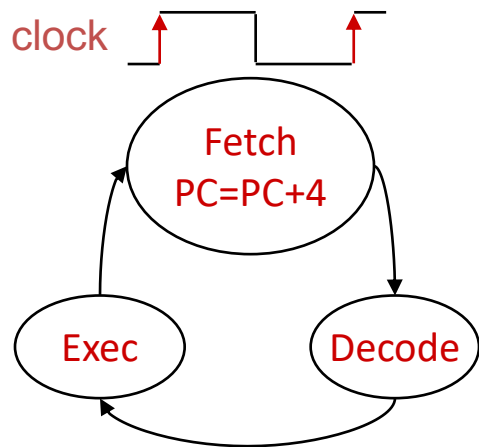
# **CprE 381: Computer Organization and Assembly Level Programming**

Processor Design

Henry Duwe  
Electrical and Computer Engineering  
Iowa State University

# Review: Fetching Instructions

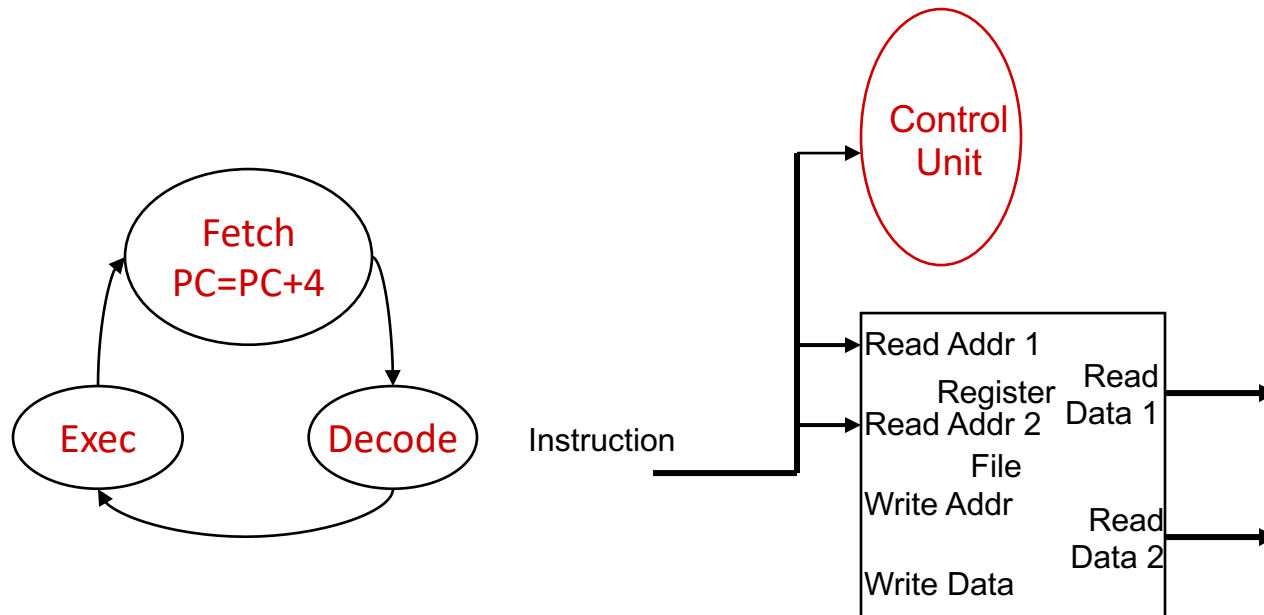
- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC value to be the address of the next (sequential) instruction



- PC is updated every clock cycle, so it does not need an explicit write control signal – just a clock signal
- Reading from the Instruction Memory is a combinational activity

# Review: Decoding Instructions

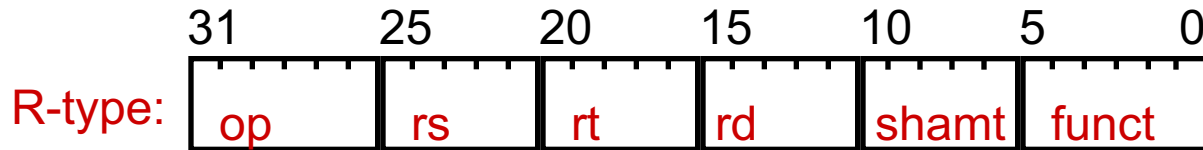
- Decoding instructions involves
  - Sending the fetched instruction's **opcode** and **function** field bits to the control unit



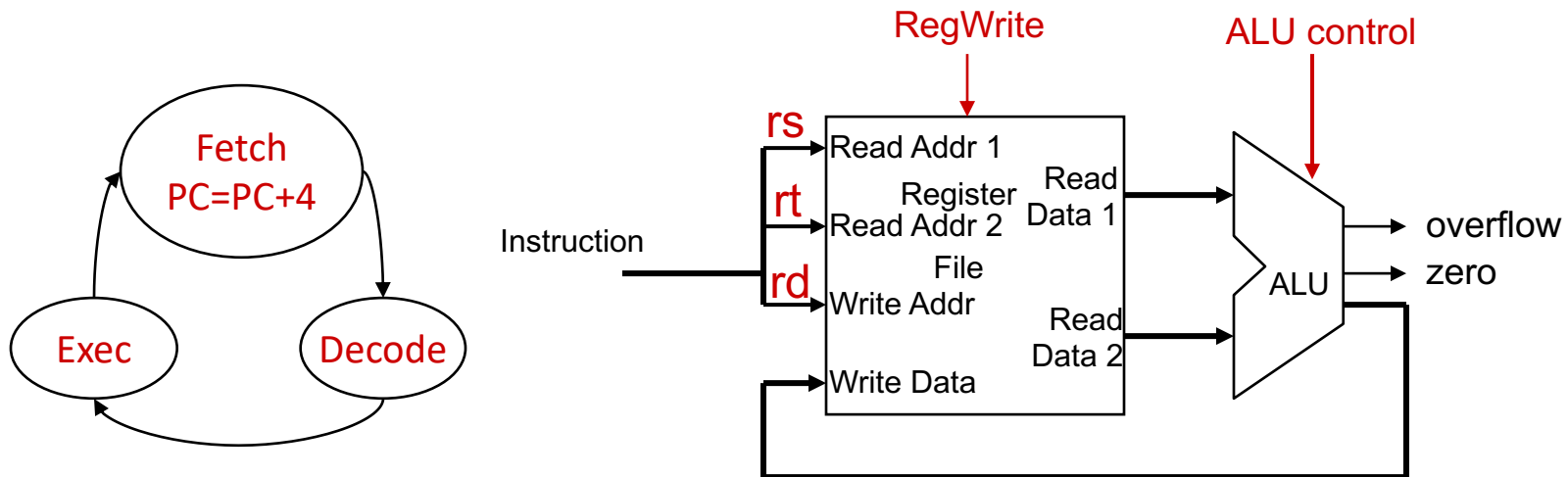
- Reading two values from the Register File
  - Register File addresses are contained in the instruction

# Review: Executing R Format Operations

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



- Perform operation (**op** and **funct**) on values in **rs** and **rt**
- Store the result back into the Register File (into location **rd**)



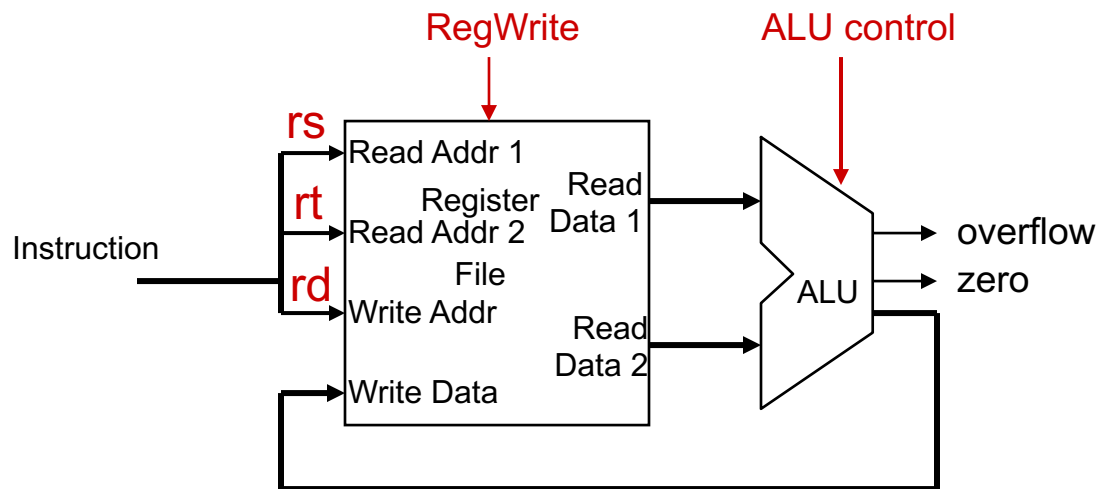
- Note that Register File is not written every cycle (e.g. **sw** or **jr**), so we need an explicit write control signal for the Register File

# Consider the `slt` Instruction

- Remember the R format instruction `slt`

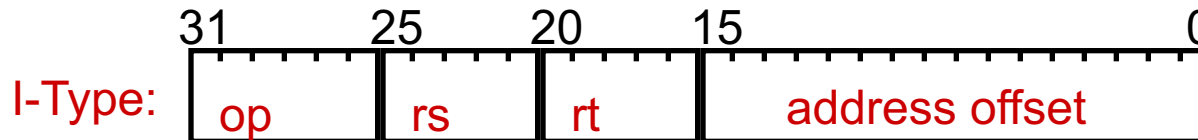
```
slt $t0, $s0, $s1    # if $s0 < $s1  
                      # then $t0 = 1  
                      # else $t0 = 0
```

- Where does the 1 (or 0) come from to store into `$t0` in the Register File at the end of the execute cycle?



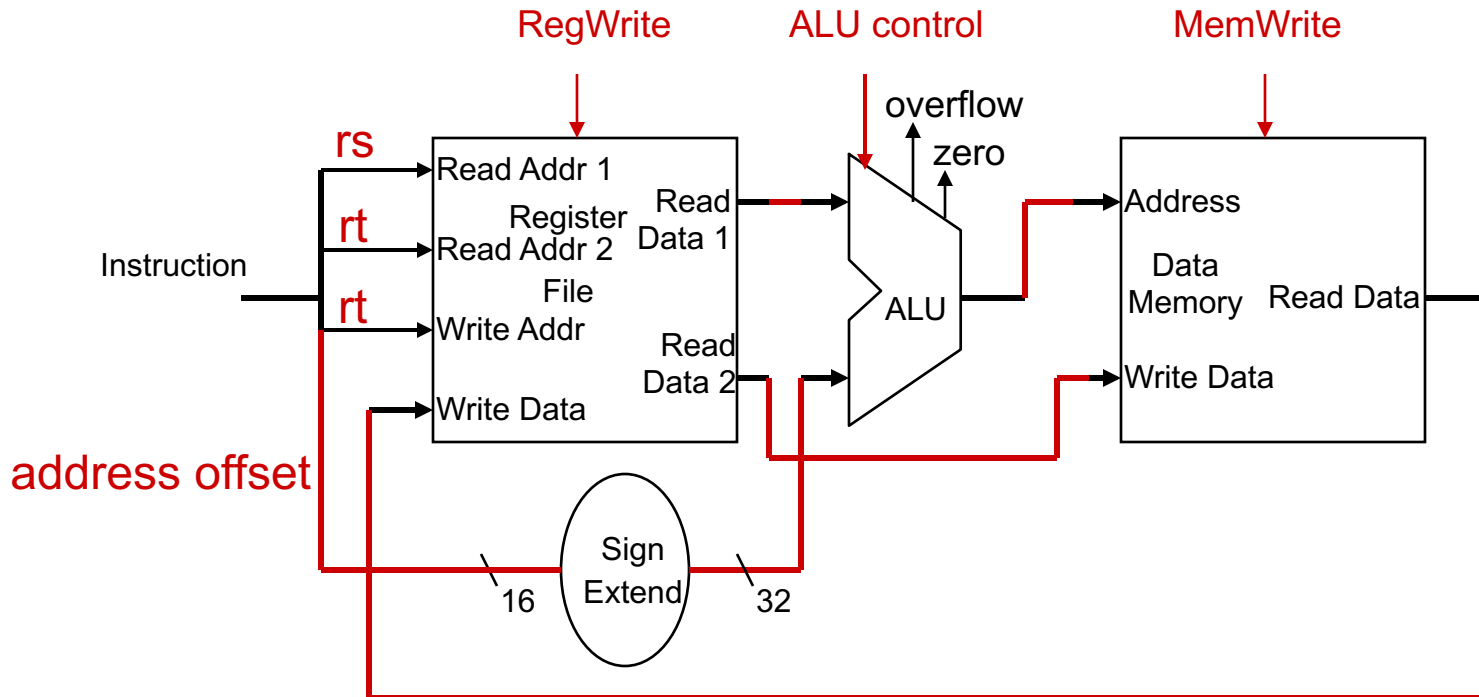
# Review: Executing Load and Store Operations

- Load and store operations have to:



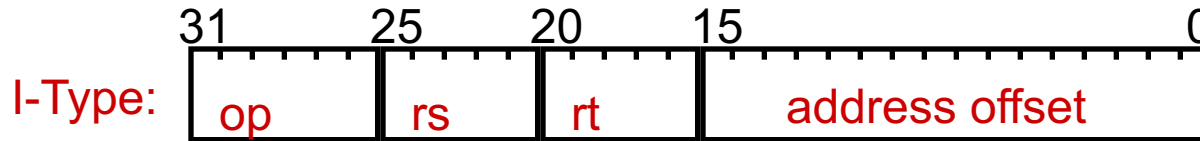
- Compute a memory address by adding the base register (in **rs**) to the 16-bit signed offset field in the instruction
  - Base register was read from the Register File during decode
  - Offset value in the low order 16 bits of the instruction must be sign extended to create a 32-bit signed value
- **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

# Executing Load / Store Operations (cont.)



# Executing Branch Operations

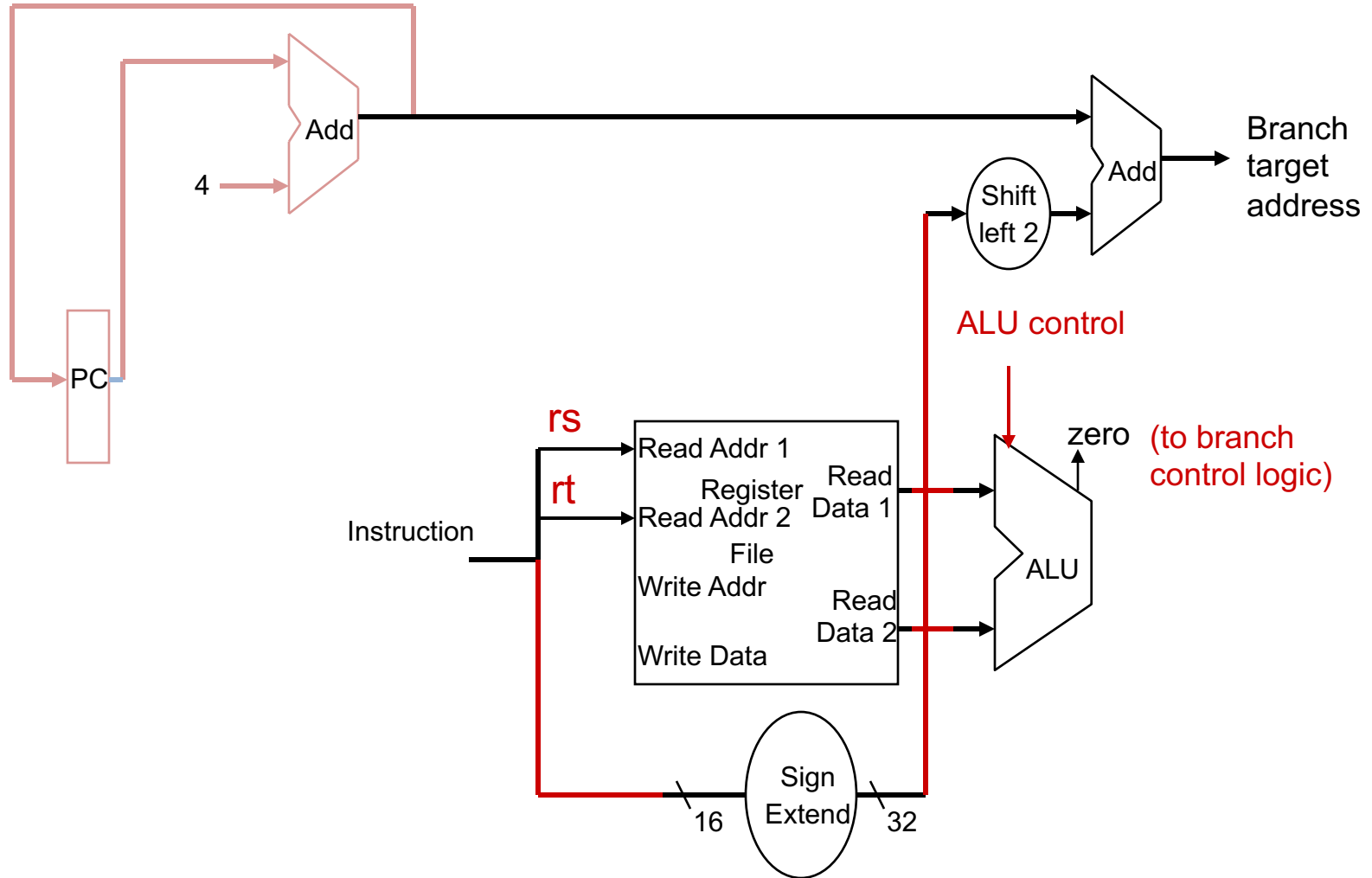
- Branch operations have to



- 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 extended 16-bit signed offset field in the instruction
  - The “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



# Executing Branch Operations (cont.)

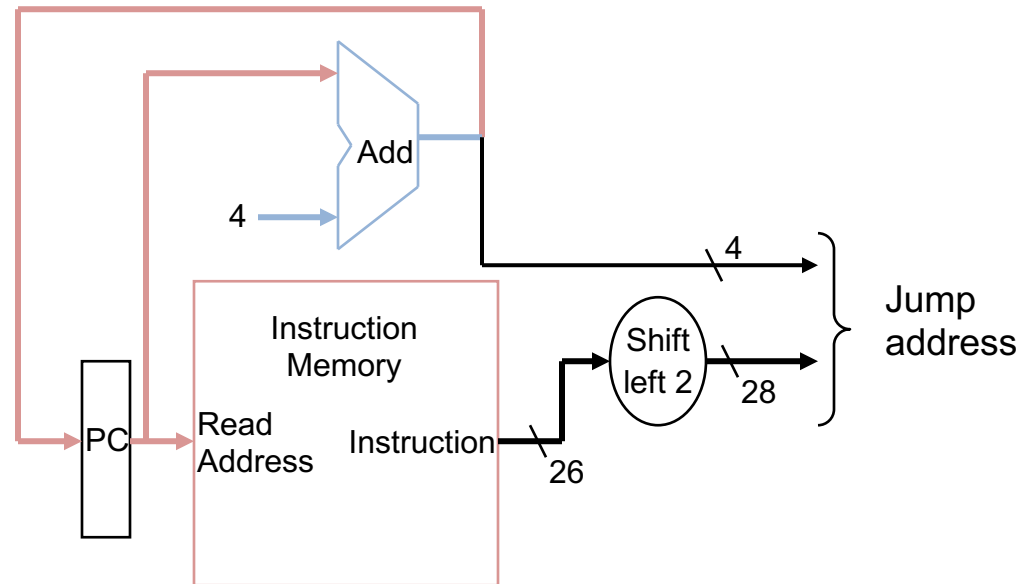


# Executing Jump Operations

- Jump operations have to



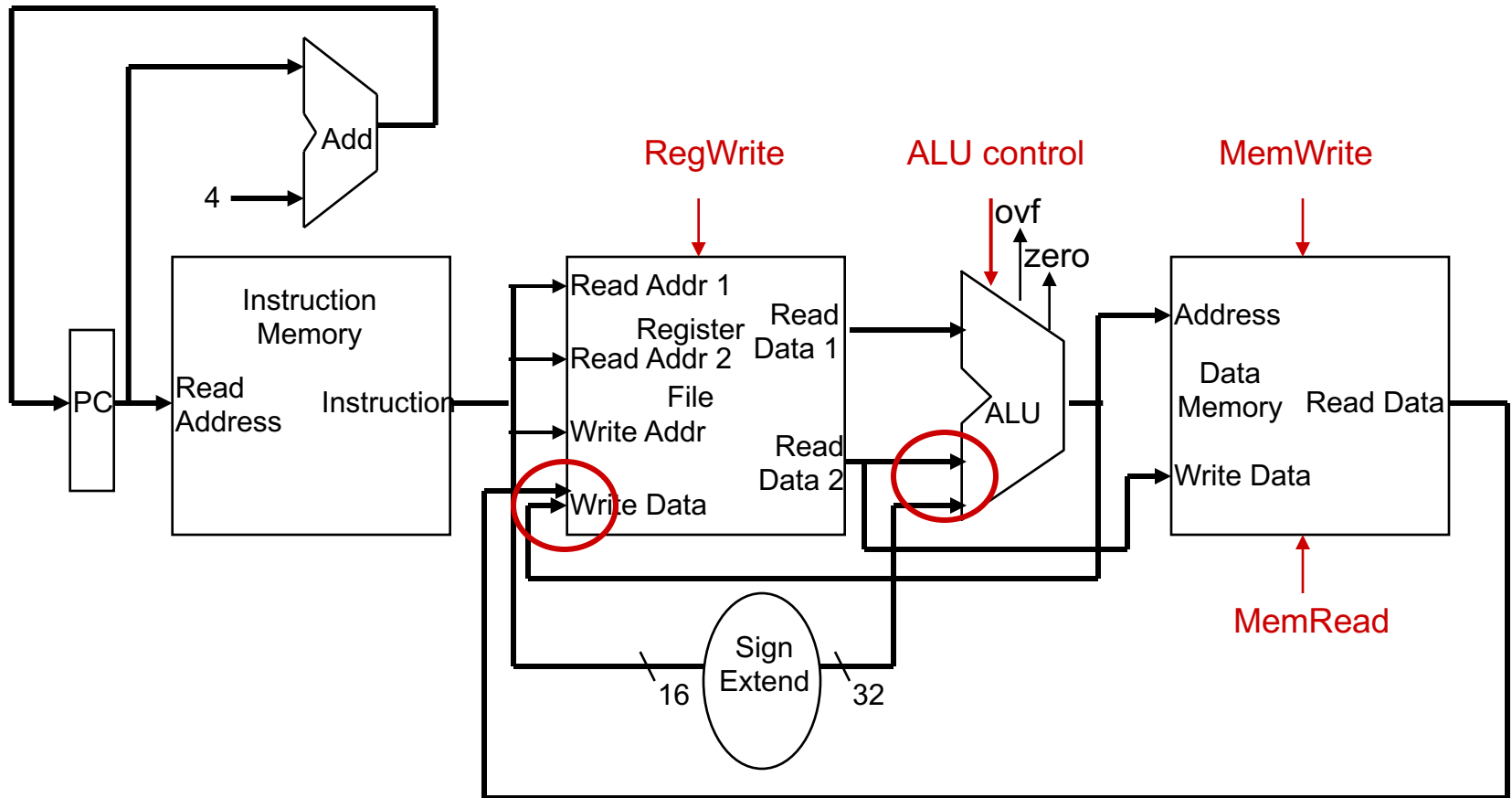
- Replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits



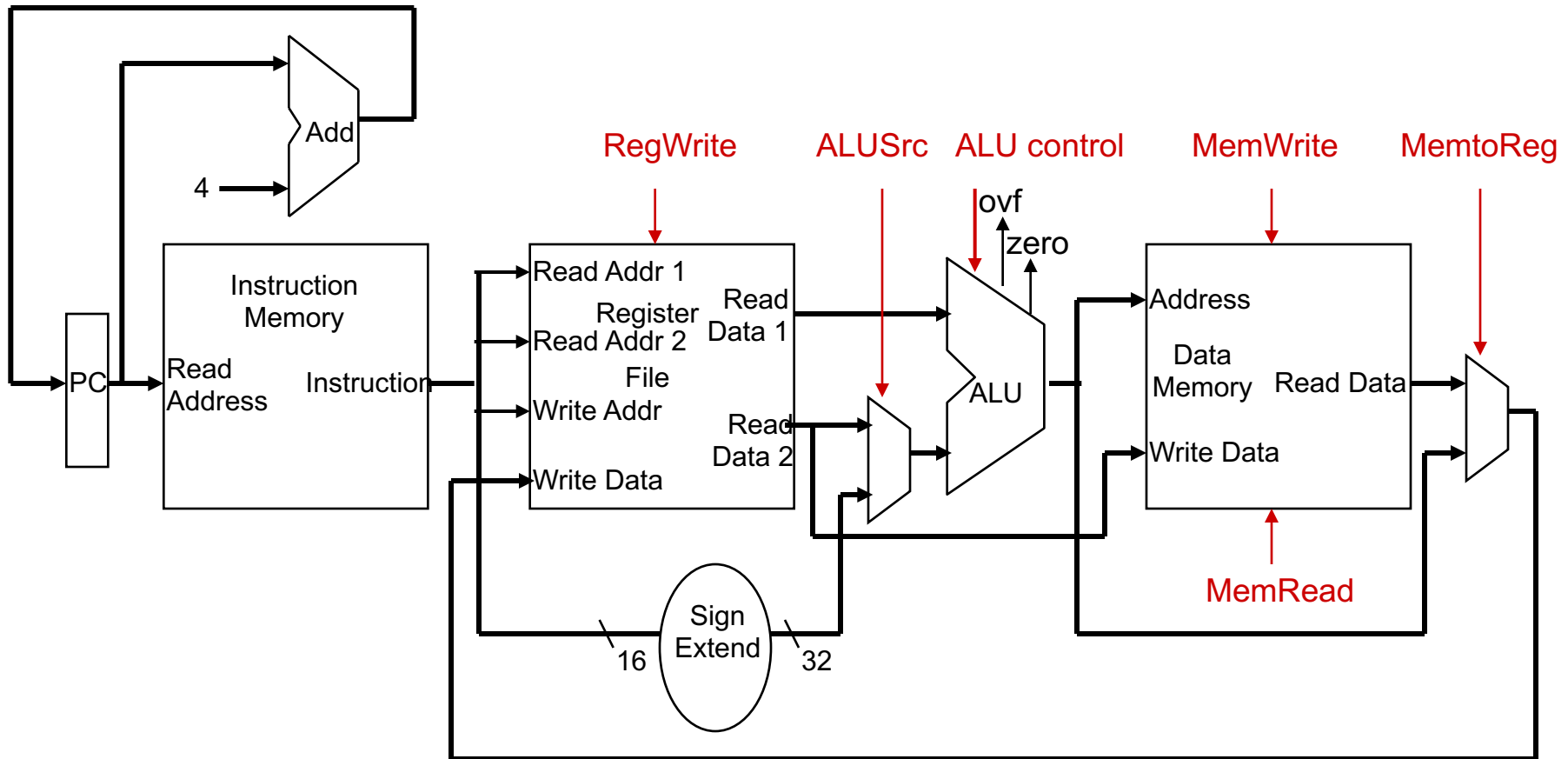
# Creating a Single Datapath from the Parts

- Assemble the datapath elements, add control lines as needed, and design the control path
- Fetch, decode and execute each instructions in one clock cycle – **single cycle** design
  - One instruction can't use same resource/structure twice (ergo Harvard split memory architecture)
  - Two different instructions 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

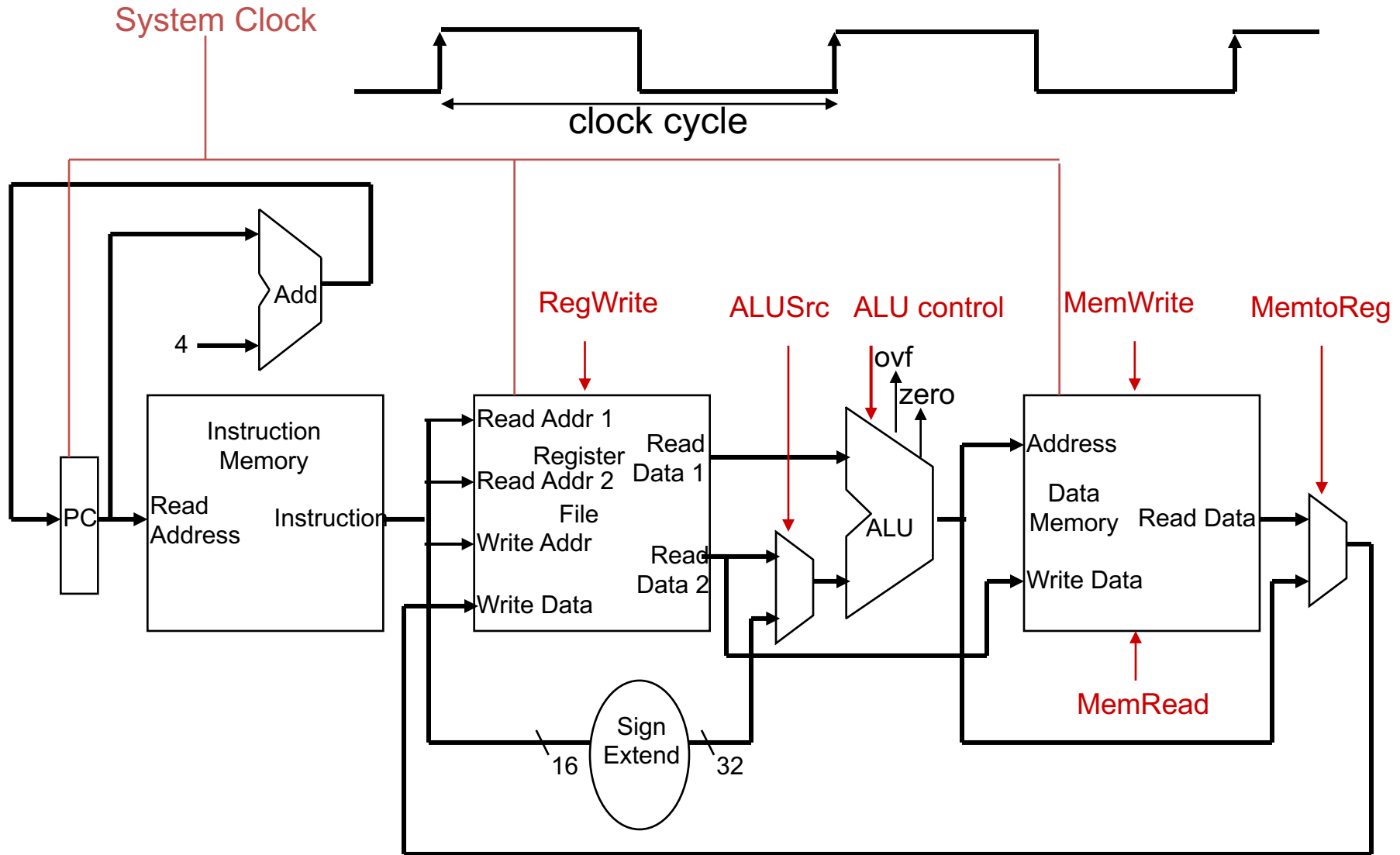
# Fetch, R, and Memory Access Portions



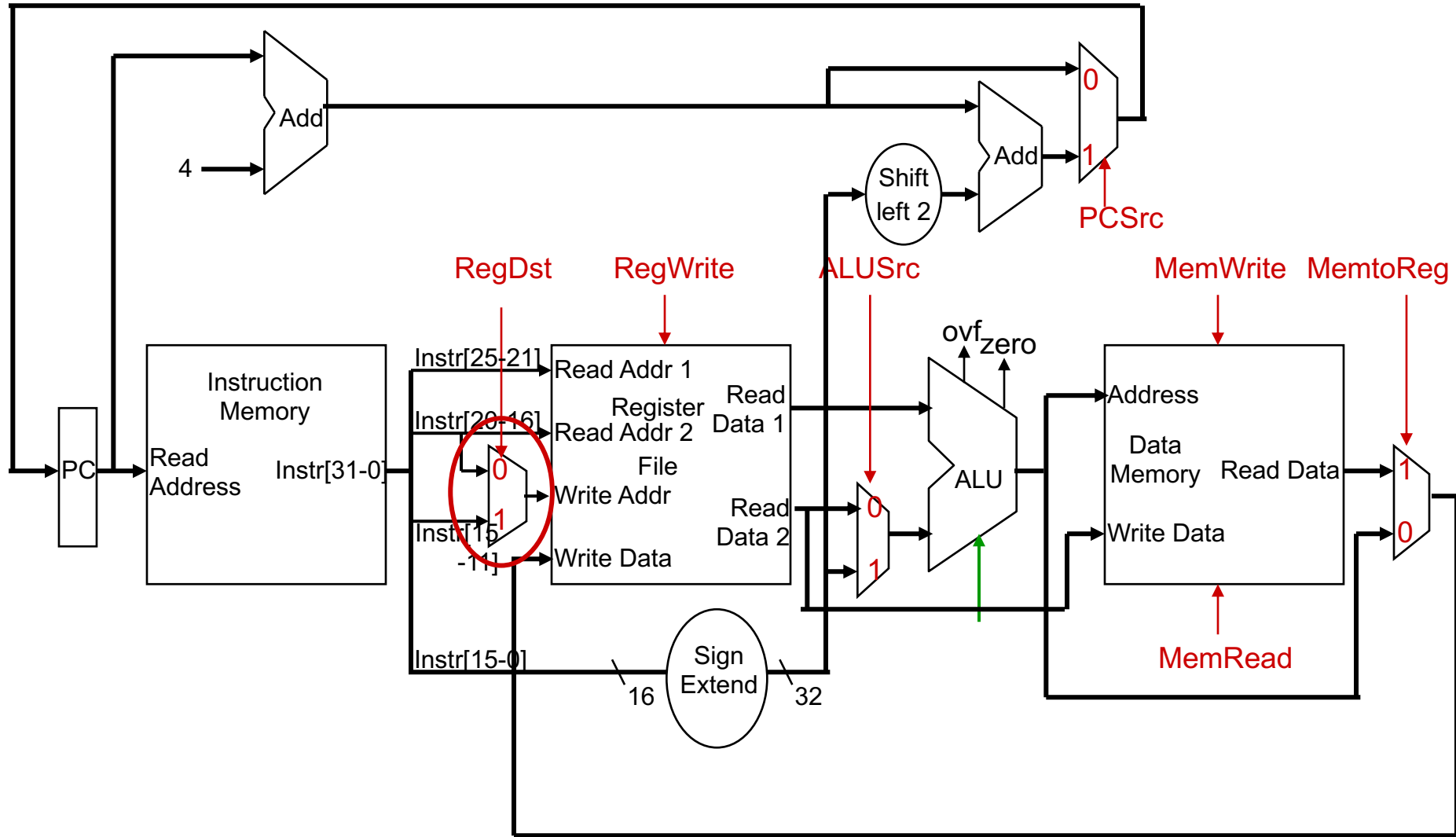
# Multiplexor Insertion



# Clock Distribution

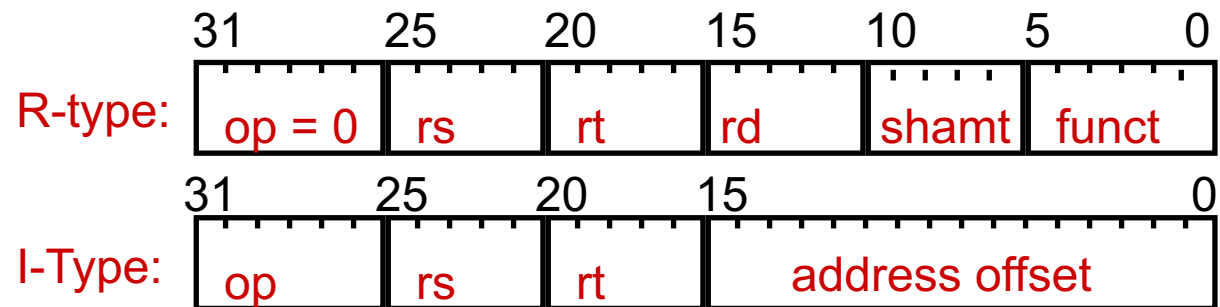


# (Almost) Complete Single Cycle Processor



# Adding the Control

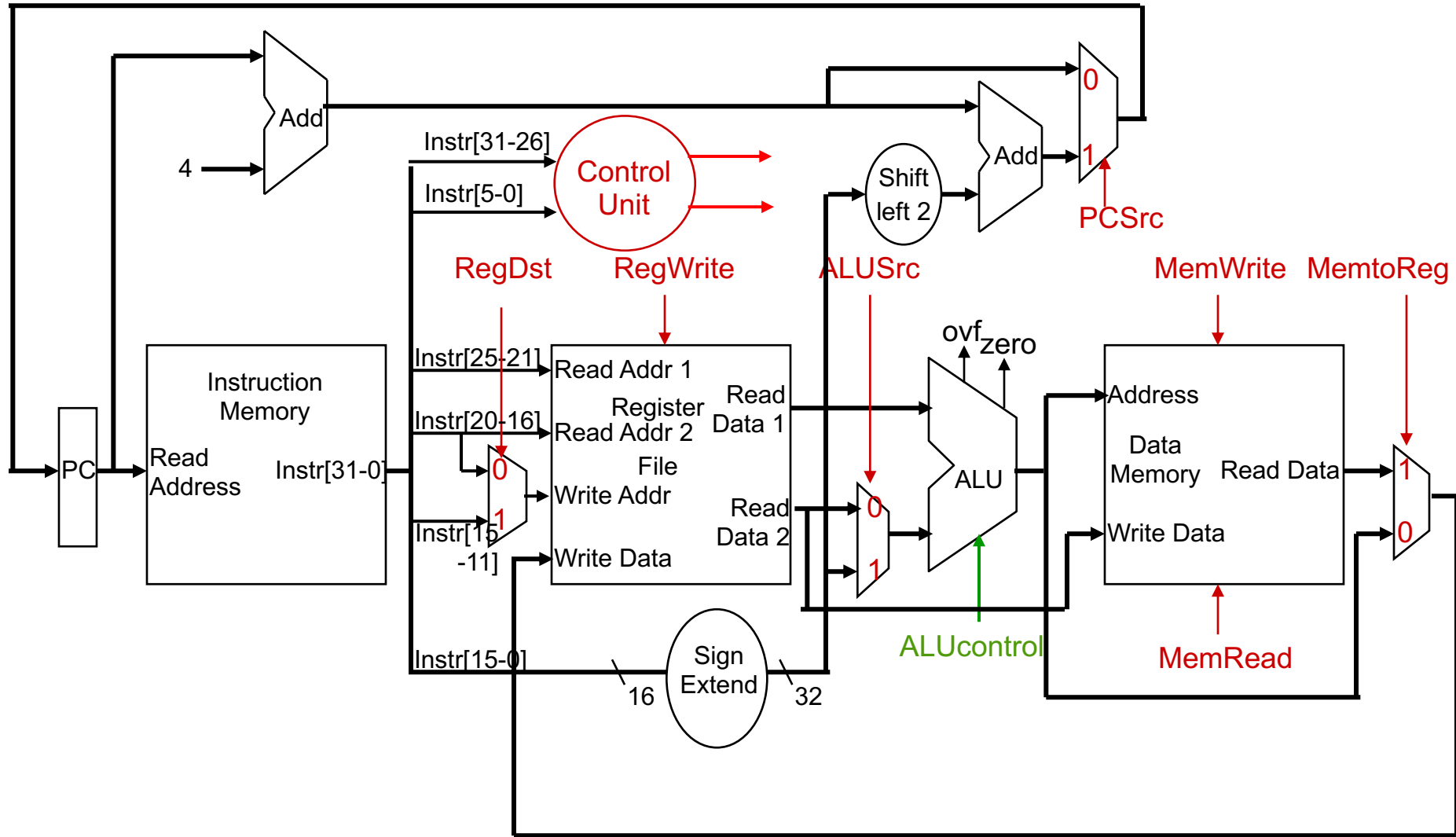
- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction



- Observations
  - op field always in bits 31-26
  - When op field is 0, funct field used from bits 5-0
  - Operand locations regular – rs always 25-20, rt always 20-15, etc.



# (Almost) Complete Single Cycle Processor



# ALU Control

- ALU's operation based on instruction type and function code:


ALU control input	Function
0000	and
0001	or
0010	xor
0011	nor
0110	add
1110	subtract
1111	set on less than

- Notice that we are using **different** encodings than in the book (and **different** than you have chosen for your project)

# ALU Control (cont.)

Four truth tables

- So describe with a truth table

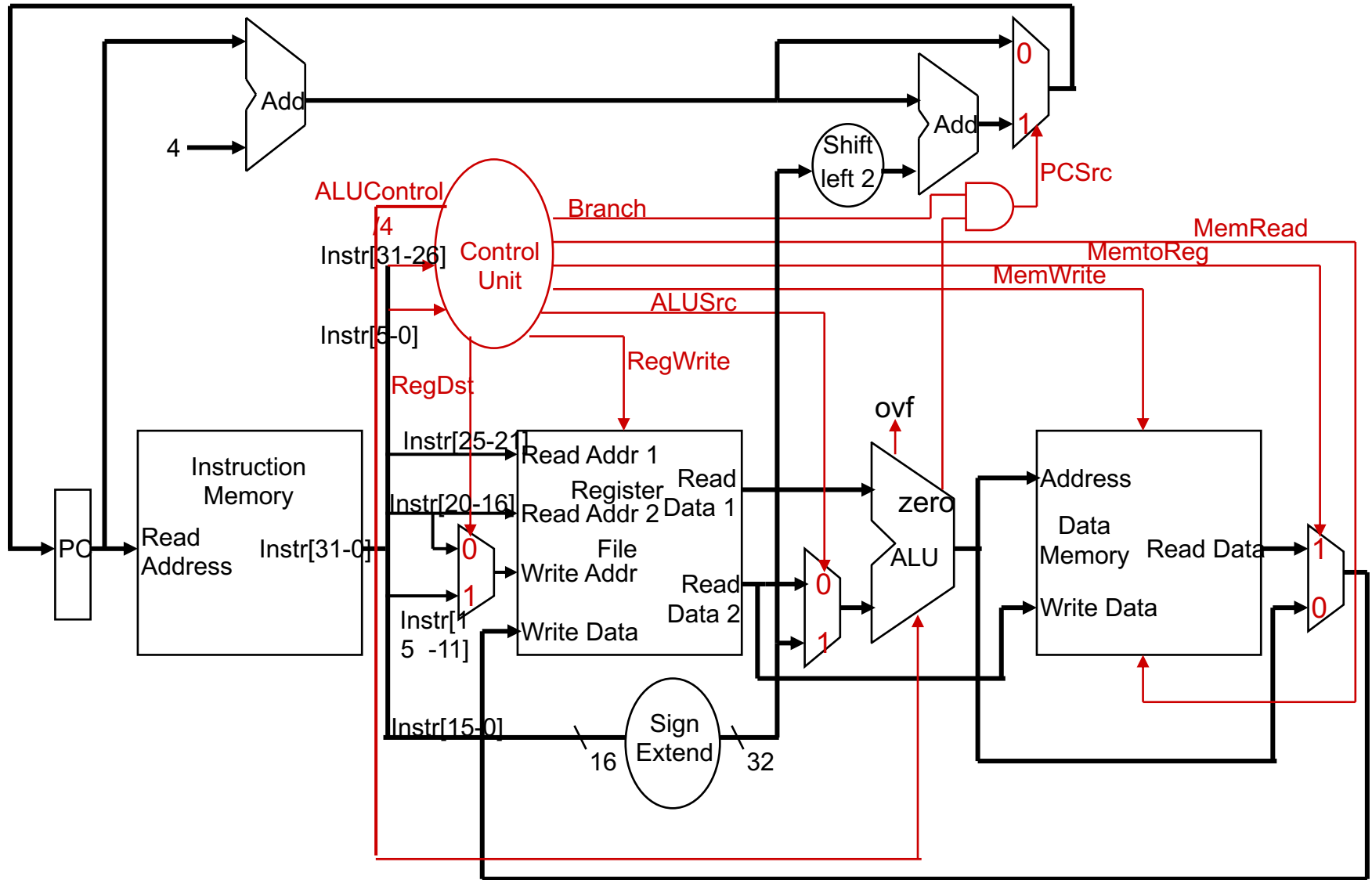


Instr op	Opcode	funct	action	ALUcontrol
lw	100011	xxxxxx	add	0110
sw	101011	xxxxxx	add	0110
beq	000100	xxxxxx	subtact	1110
add	000000	100000	add	0110
subt	000000	100010	subtract	1110
and	000000	100100	and	0000
or	000000	100101	or	0001
xor	000000	100110	xor	0010
nor	000000	100111	nor	0011
slt	000000	101010	slt	1111



12 inputs

# (Almost) Complete Datapath w/ Control

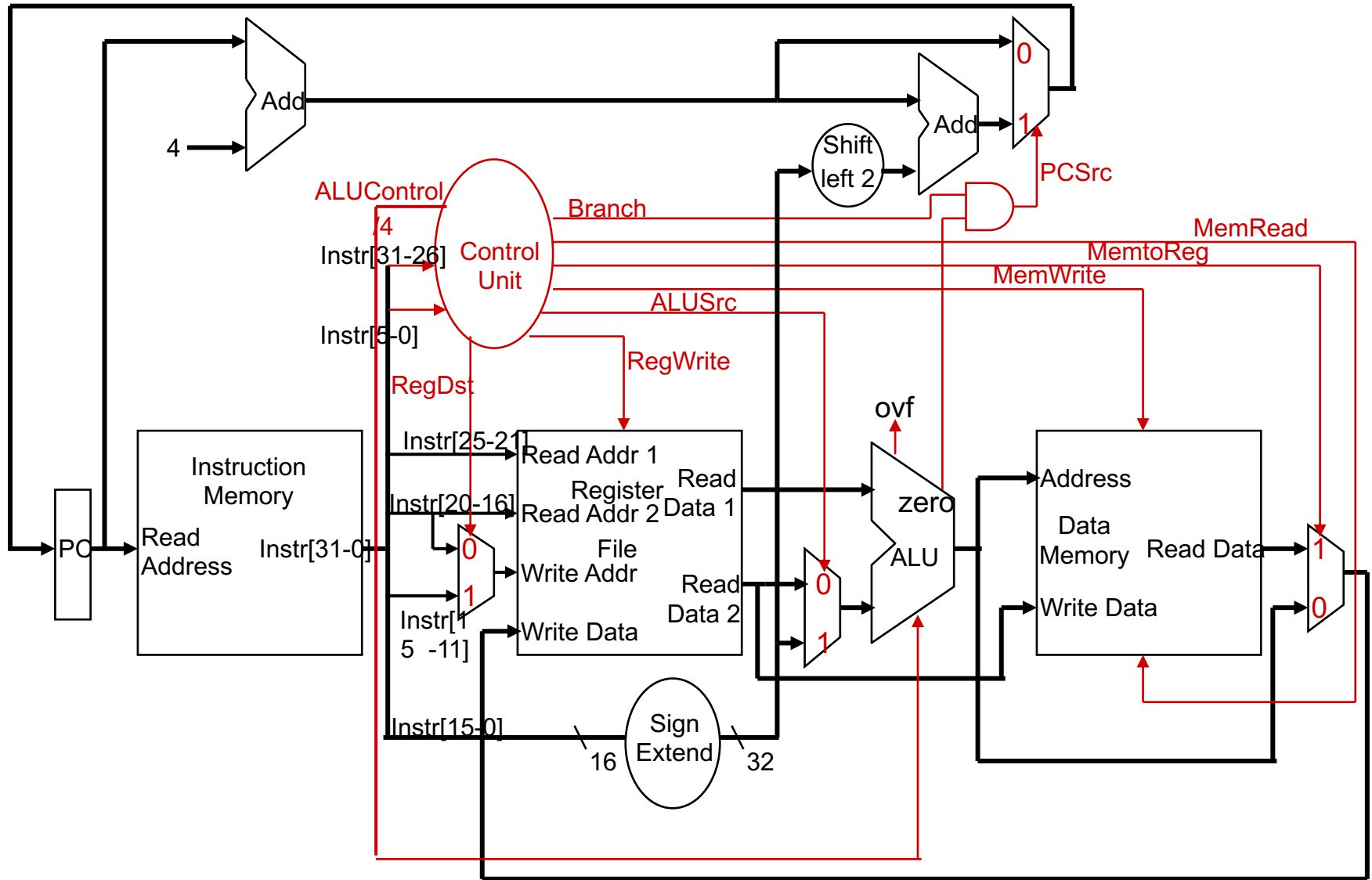


# Main Control Unit

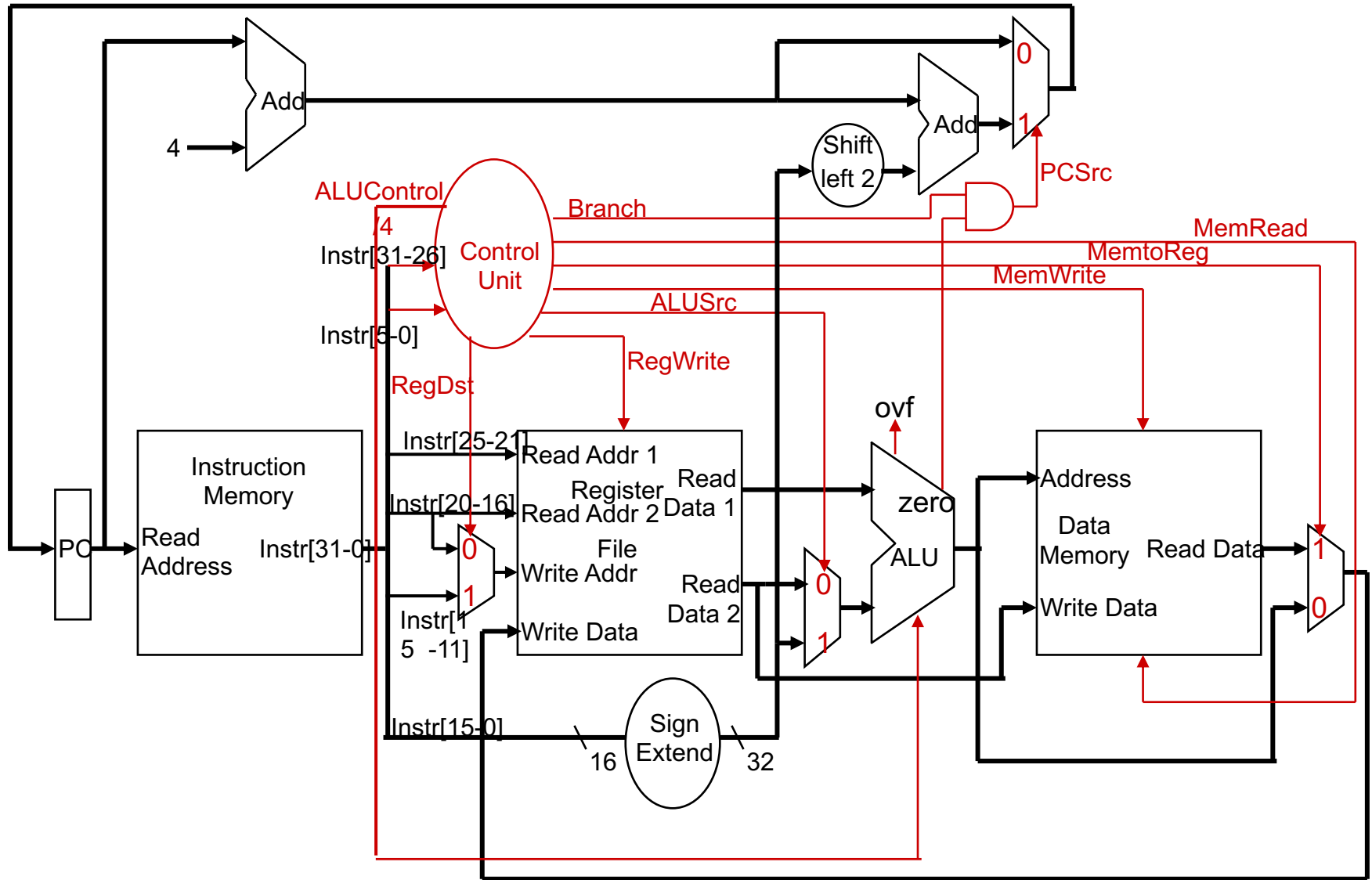
Instr	ALU control	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch
<b>R-type</b> 000000								
<b>lw</b> 100011								
<b>sw</b> 101011								
<b>beq</b> 000100								

- Note that a multiplexor whose control input is 0 has a definite action, even if it is not used in performing the operation

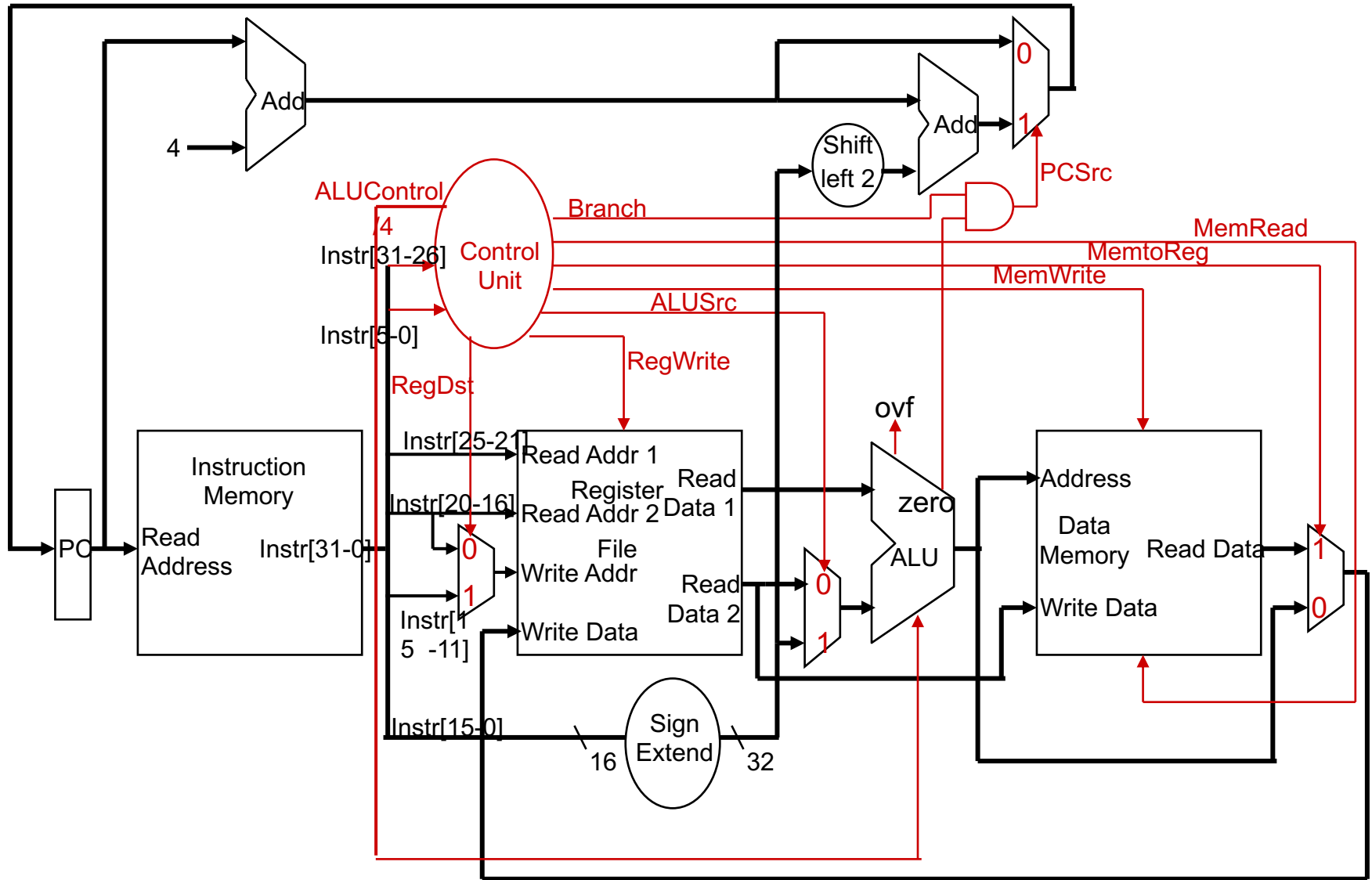
# R-Type Instruction Data/Control Flow



# Iw Instruction Data/Control Flow

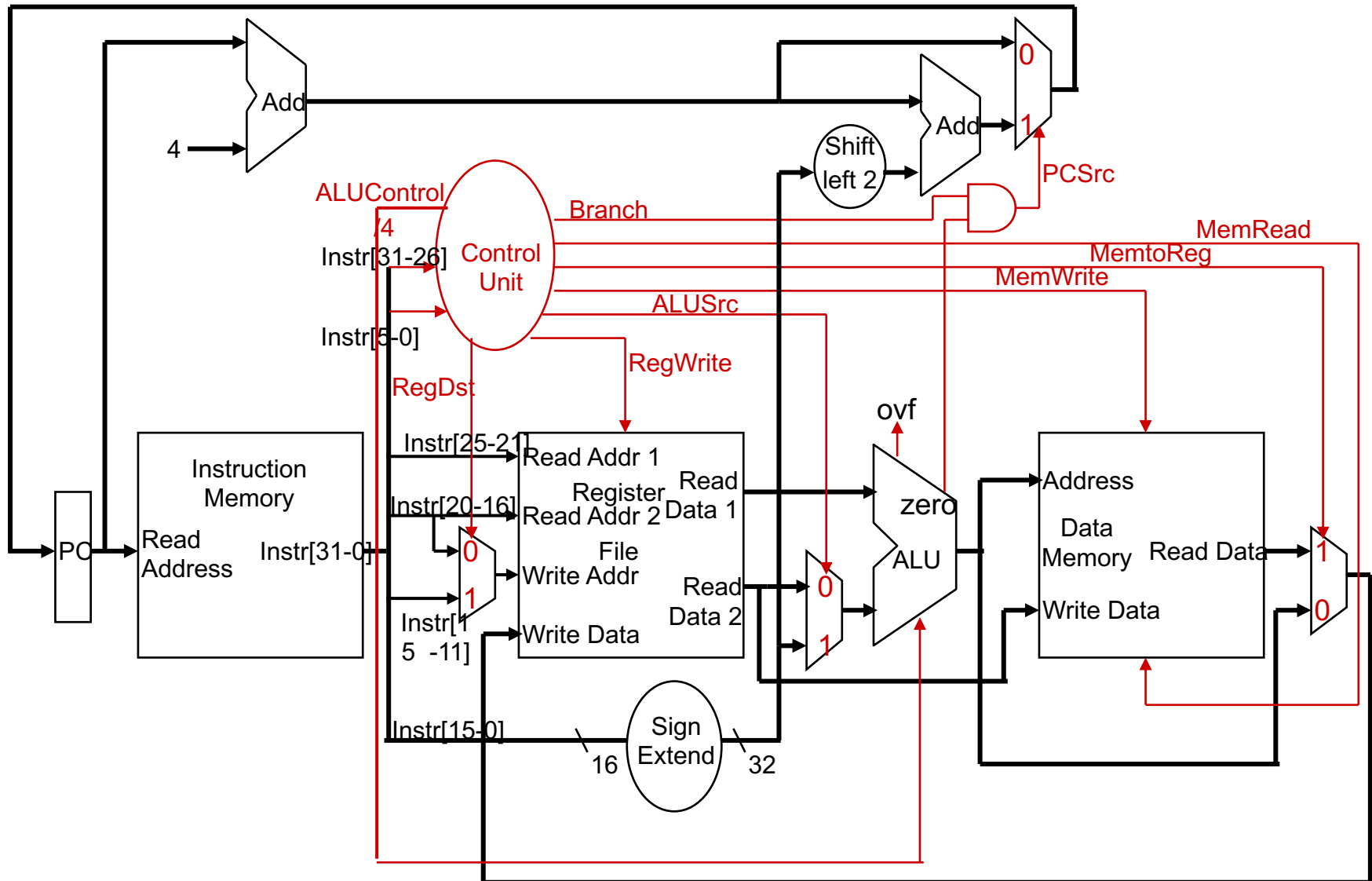


# sw Instruction Data/Control Flow





# beq Instruction Data/Control Flow



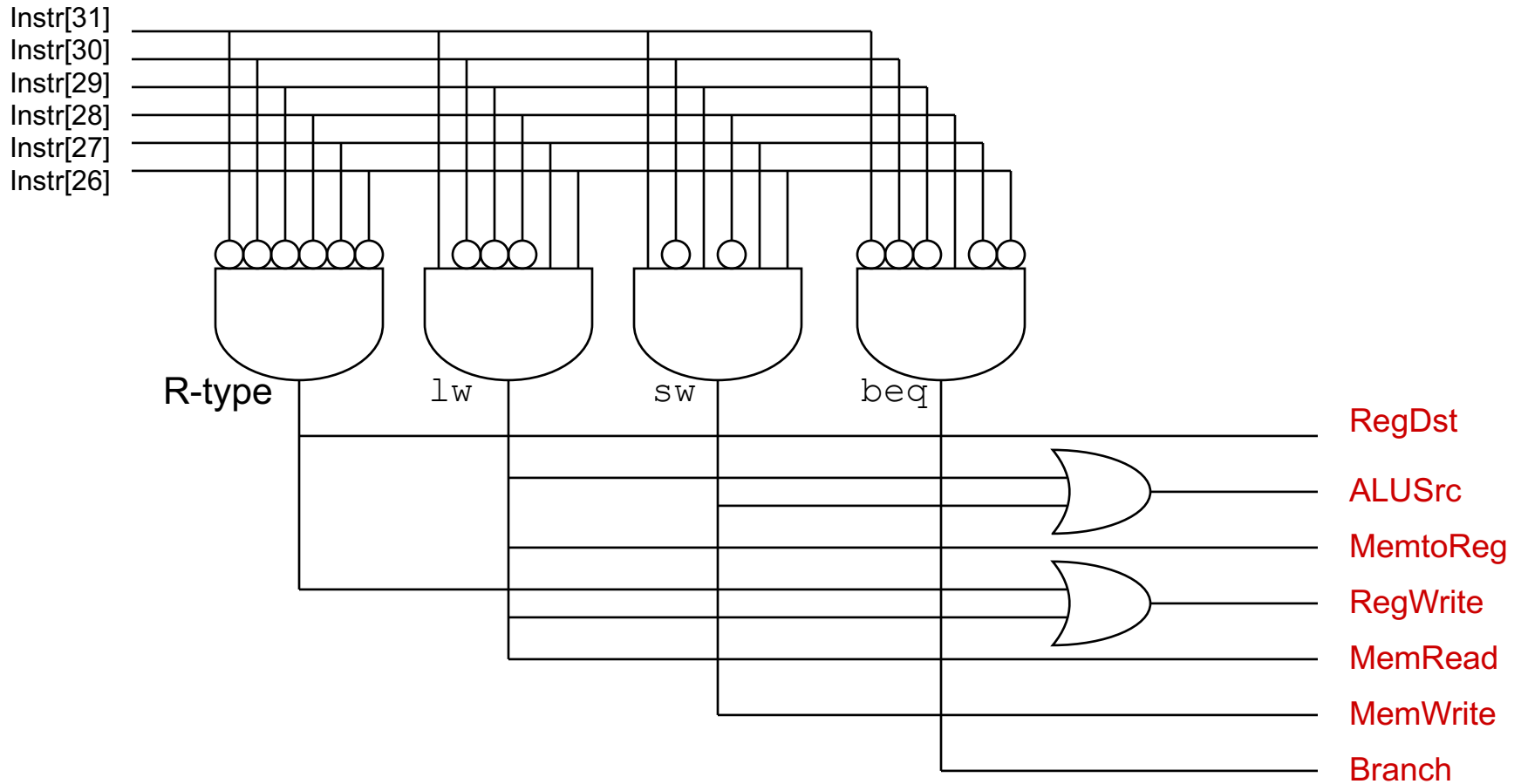
# Main Control Unit (cont.)

Instr	ALU control	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch
<b>R-type</b> 000000	Depends on Funct	1	0	0	1	0	0	0
<b>lw</b> 100011	0110	0	1	1	1	1	0	0
<b>sw</b> 101011	0110	X	1	X	0	0	1	0
<b>beq</b> 000100	1110	X	0	X	0	0	0	1

- Setting of the MemRd signal (for R-type, sw, beq) depends on the memory design (could have to be 0 or could be a X (don't care))

# Control Unit Logic

- From the truth table can design the Main Control logic

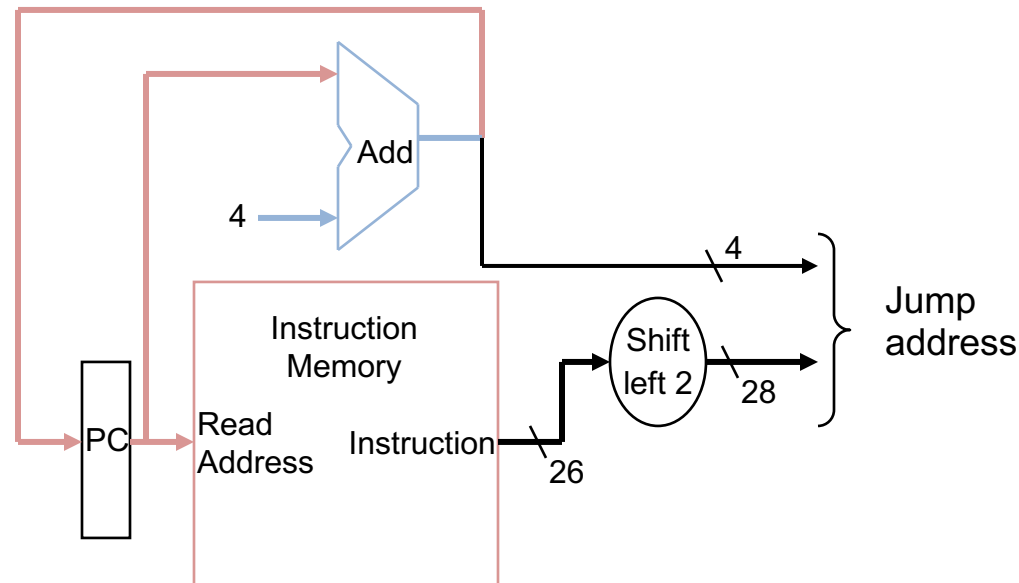


# Review: Executing Jump Operations

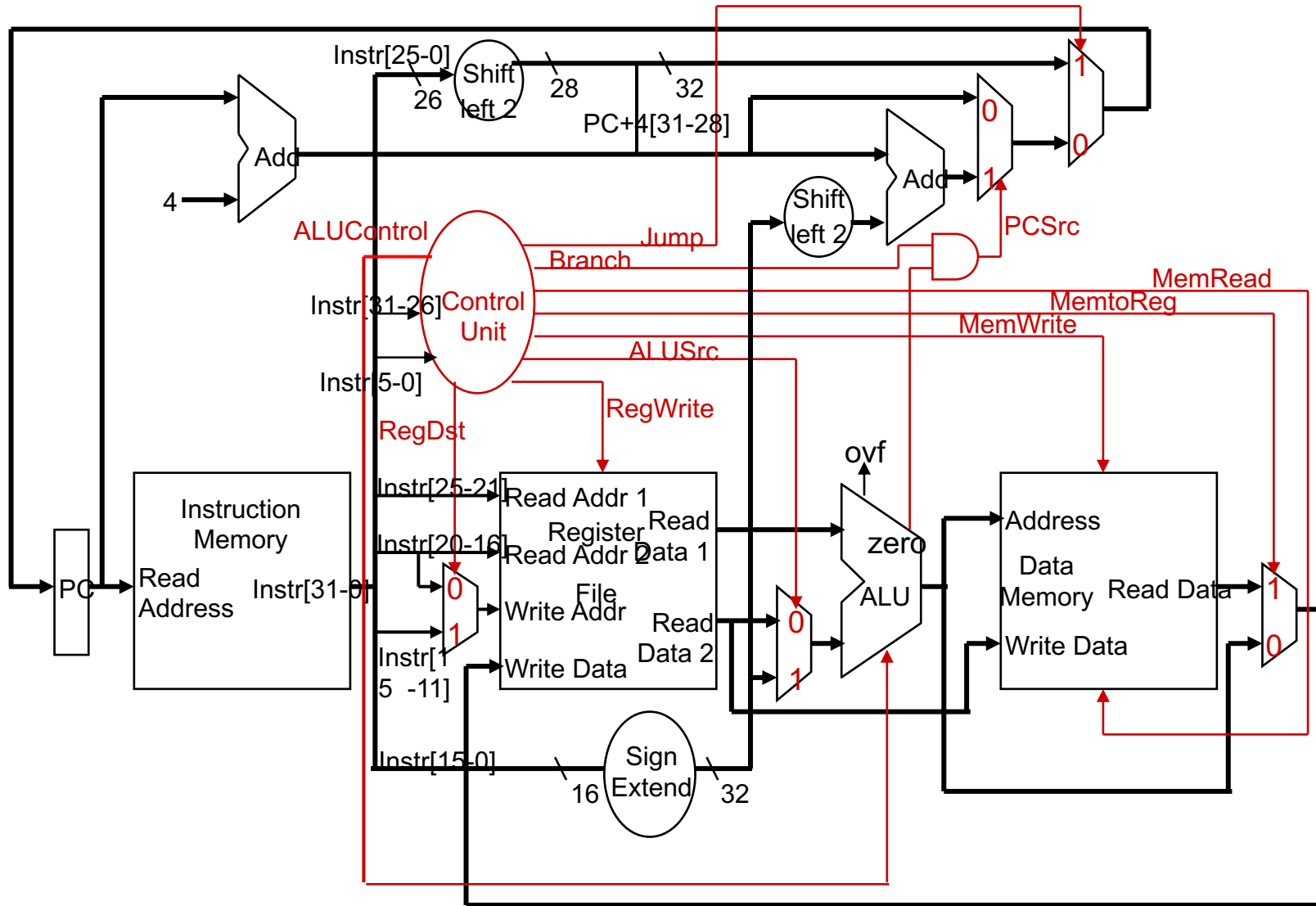
- Jump operations have to



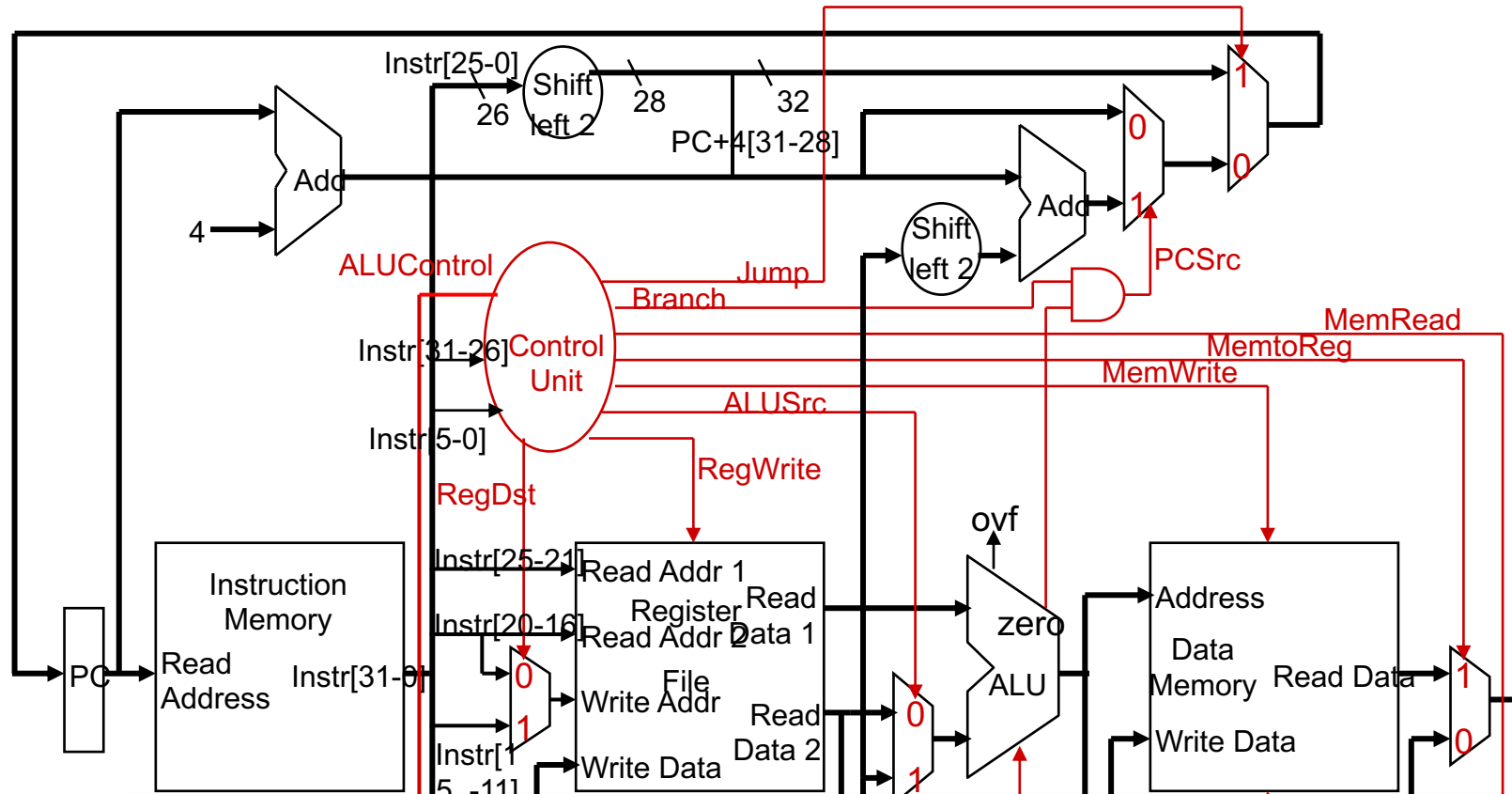
- Replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits



# Adding the Jump Instruction



# Adding the Jump Instruction



**In-class Assessment!**  
**Access Code: Heigh-ho**

**Note: sharing access code to those outside of classroom or using access while outside of classroom is considered cheating**

# Main Control Unit (cont.)

Instr	ALU control	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	Jump
<b>R-type</b> 000000	Depends on Funct	1	0	0	1	0	0	0	0
<b>lw</b> 100011	0110	0	1	1	1	1	0	0	0
<b>sw</b> 101011	0110	X	1	X	0	0	1	0	0
<b>beq</b> 000100	1110	X	0	X	0	0	0	1	0
<b>j</b> 000010	X	X	X	X	0	0	0	X	1

- Setting of the MemRd signal (for R-type, *sw*, *beq*) depends on the memory design

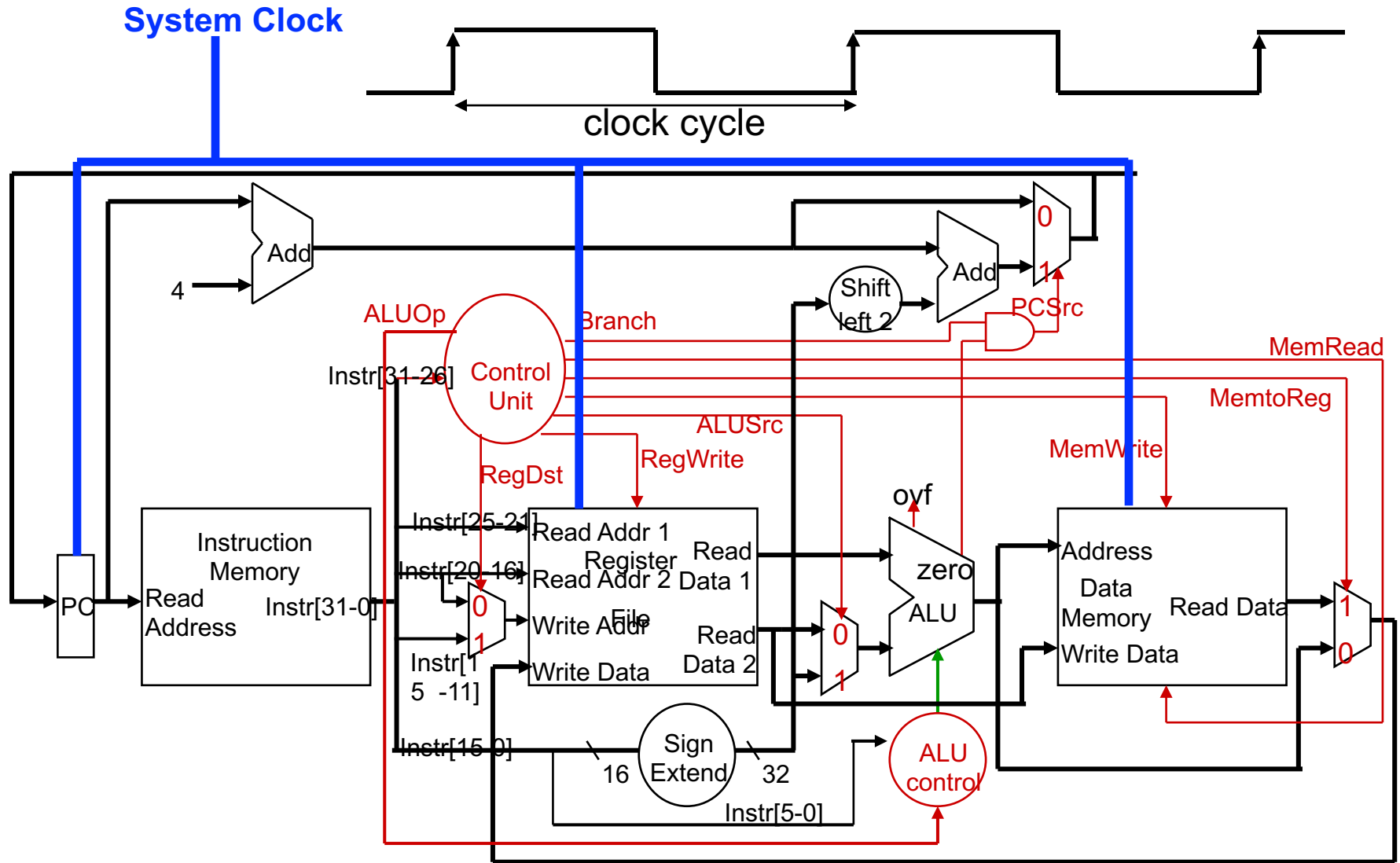
# Model Truth Table in VHDL

Use selected signal statement (data flow)

```
-- input : 5-bit addr; output: 32-bit sel
with addr select
    sel <= x"00000001" when b"00000",
          x"00000002" when b"00001",
          ... -- more cases
          x"80000000" when b"11111";
```



# Clock Distribution

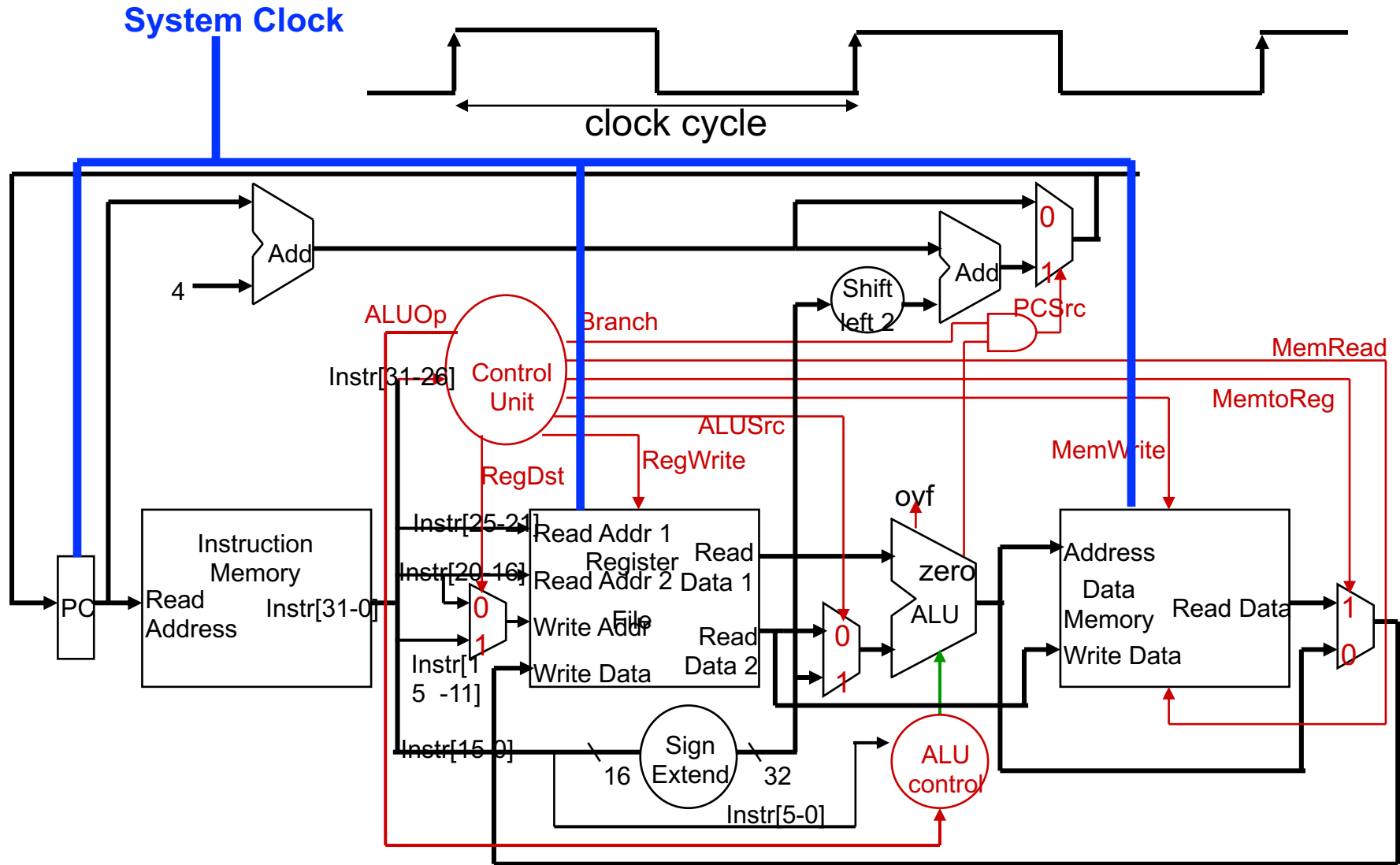


# Operation

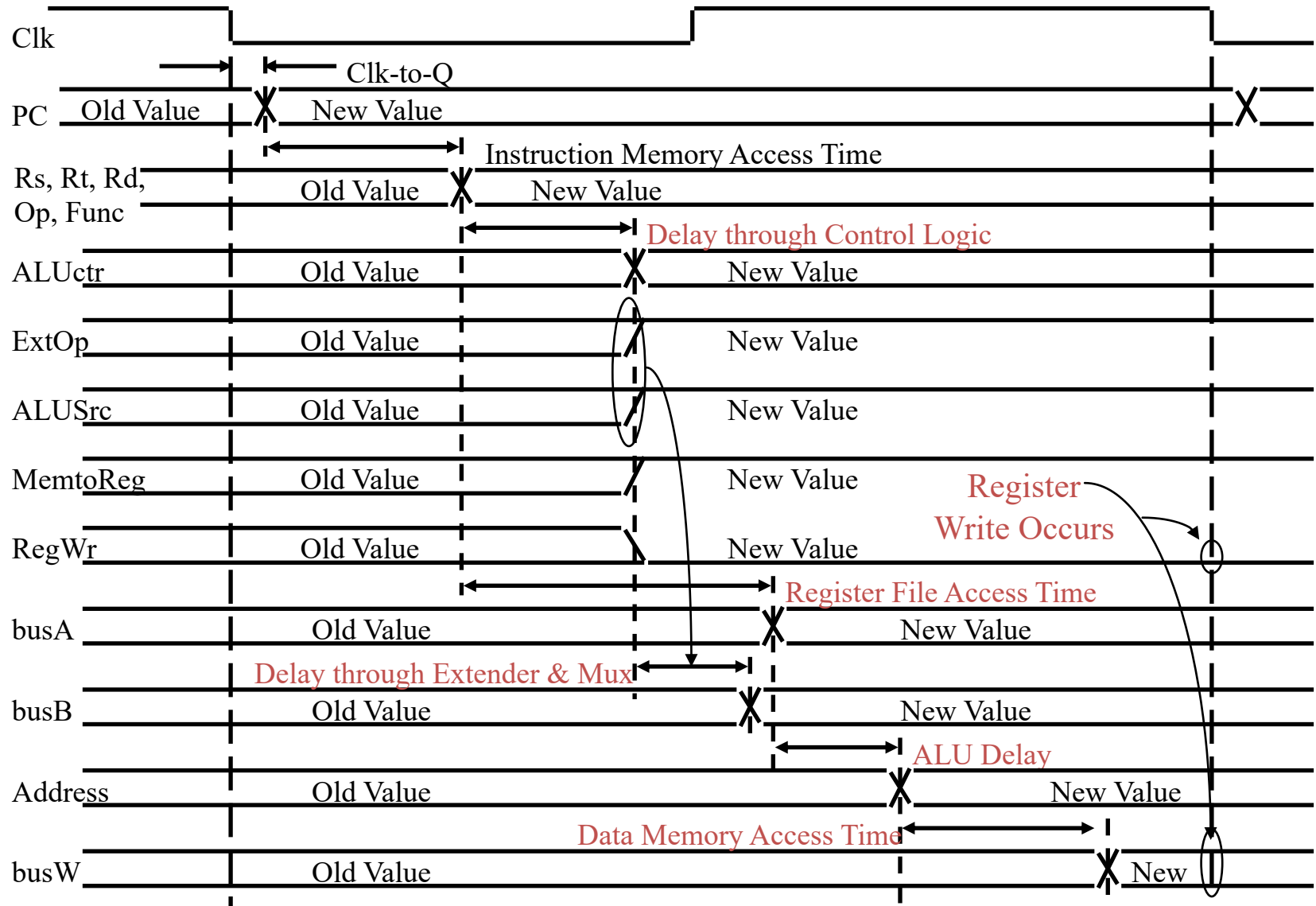
- We wait for everything to settle down
  - ALU might not produce “final answer” right away
  - Memory and RegFile reads are combinational (as are ALU, adders, muxes, shifter, signextender)
  - Use write signals along with the clock edge to determine when to write to the sequential elements (to the PC, to the Register File and to the Data Memory)
- The clock cycle time is determined by the logic delay through the longest path

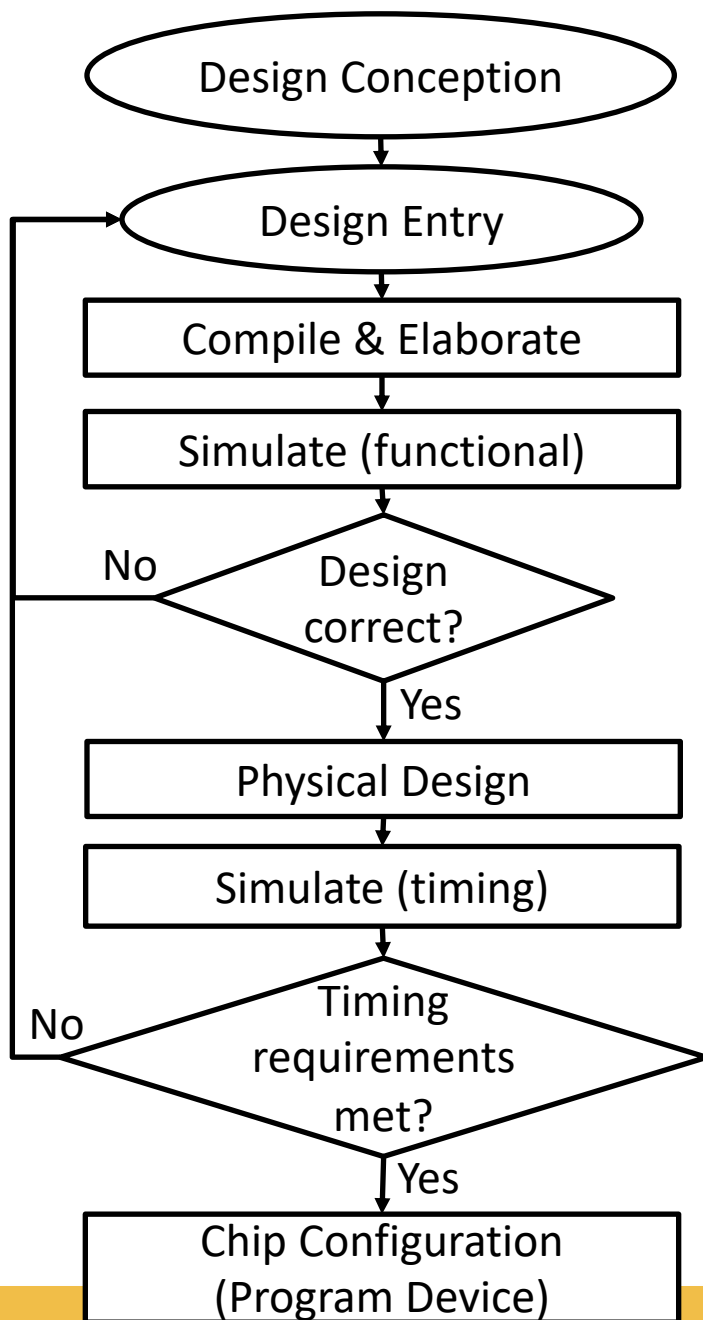
**We are ignoring some details like register setup and hold times**

# Worst Case Timing (Load Instruction)



# Worst Case Timing (Load Instruction)



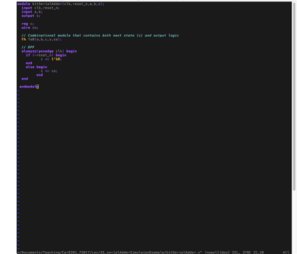


**Tools:**



**ASMs**

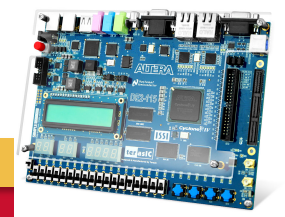
**HDLs**  
(VHDL)



**ModelSim**

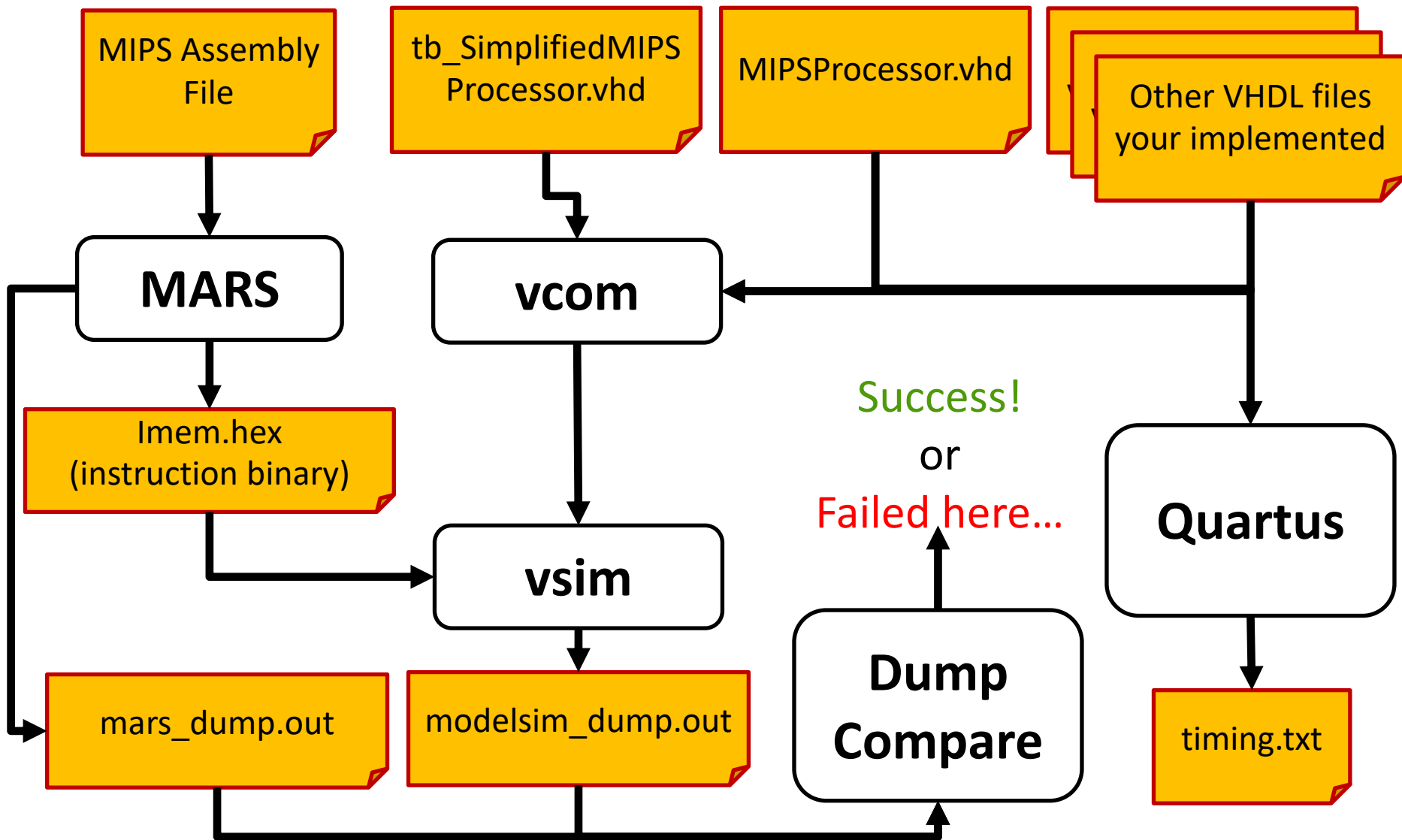


**ModelSim**



[ MODIFIED Figure 2.35 from the 281 textbook ]

# Automated Testing Framework



# Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)