CS161 – Design and Architecture of Computer Systems

Single Cycle CPU Design

UNIVERSITY OF CALIFORNIA, RIVERSIDE

CPU Organization

- We will build a CPU to implement our subset of the MIPS ISA
  - Memory Reference Instructions:
    - Load Word (LW)
    - Store Word (SW)
  - Arithmetic and Logic Instructions:
    - ADD, SUB, AND, OR, SLT
  - Branch and Jump Instructions:
    - Branch if equal (BEQ)
    - Jump unconditional (J)
- These basic instructions exercise a majority of the necessary datapath and control logic for a more complete implementation

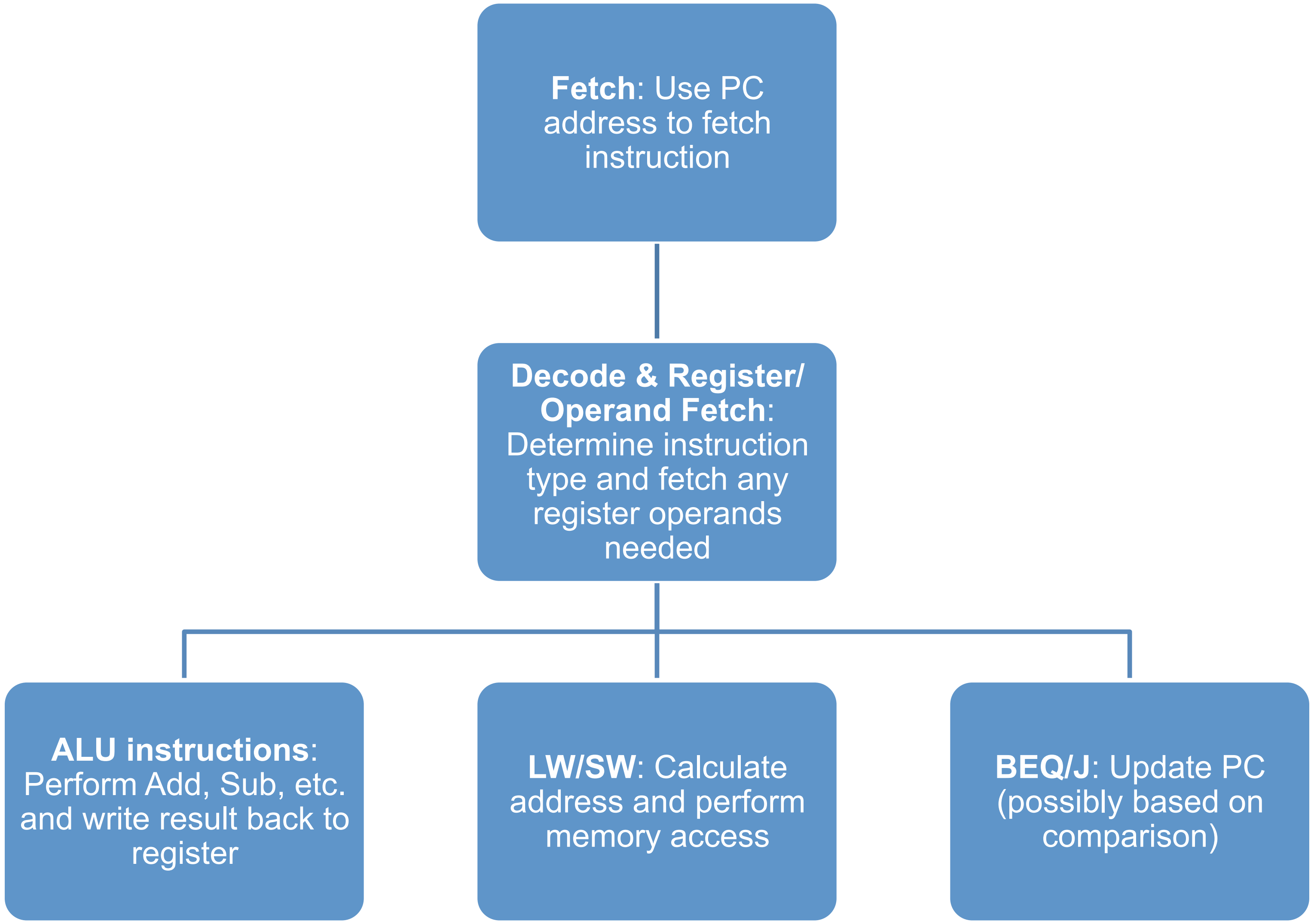
Generic Implementation

- Use program counter (PC) to supply instruction address
- Get instruction from memory
- Read registers
- Use instruction to decide exactly what to do

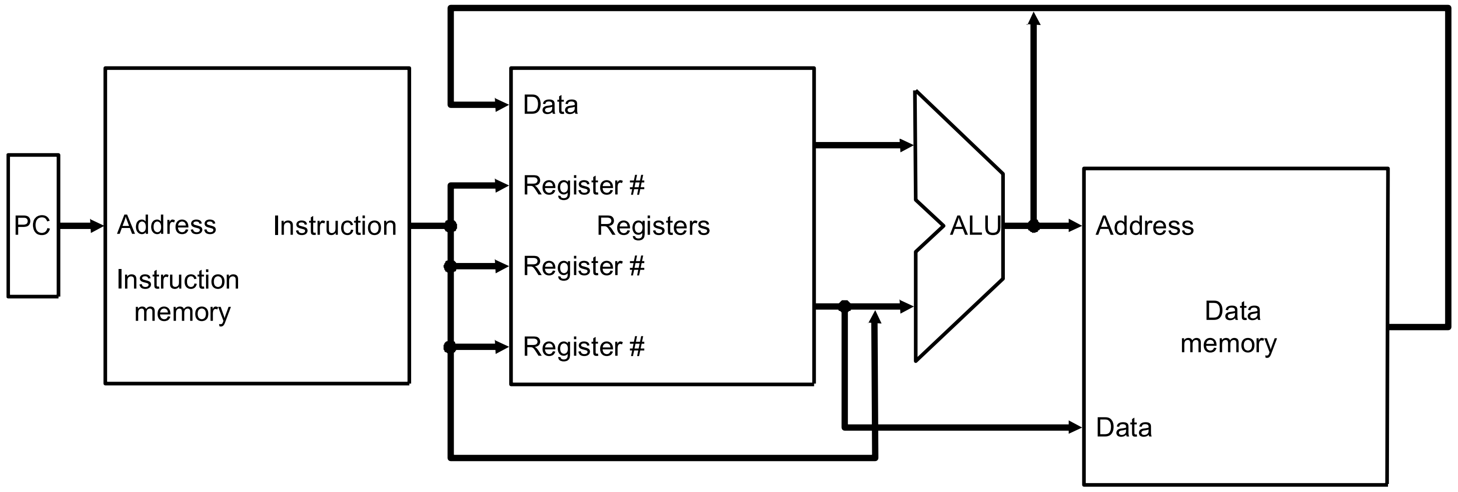
CPU Implementations

- Single-cycle CPU (CPI = 1)
  - All instructions execute in a single, long clock cycle
- Multi-cycle CPU (CPI = n)
  - Instructions can take a different number of short clock cycles to execute

Single-Cycle Datapath


Fetch: Use PC address to fetch instruction
Decode & Register/ Operand Fetch: Determine instruction type and fetch any register operands needed
ALU instructions: Perform Add, Sub, etc. and write result back to register
LW/SW: Calculate address and perform memory access
BEQ/J: Update PC (possibly based on comparison)

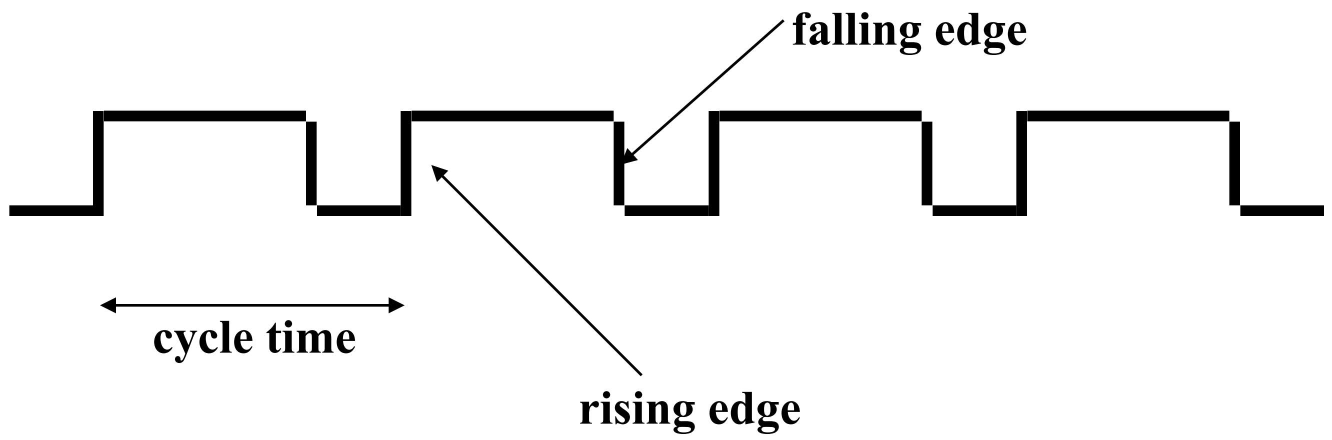
Abstract/Simplified View



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

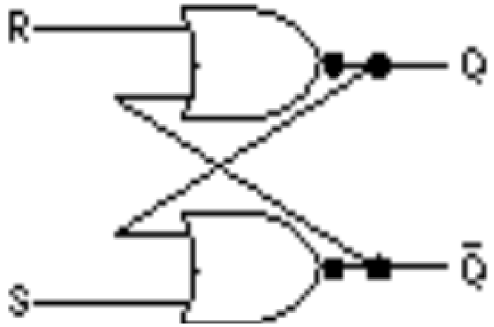
State Elements

- Unclocked vs clocked
- Clocks used in synchronous logic



Unclocked State Element

- Set-reset latch
  - Output depends on present inputs and also on past inputs



Latches and Flip-flops

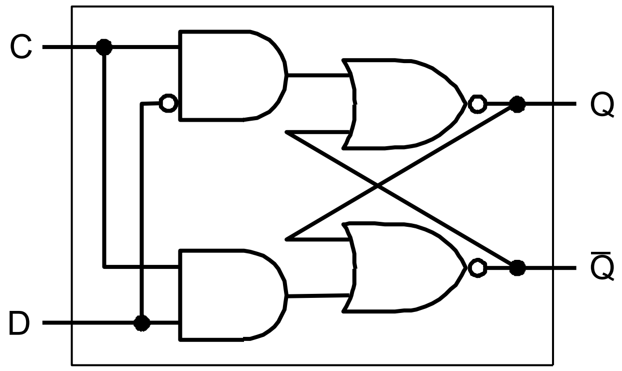
- Output is equal to the stored value inside the element (don't need to ask for permission to look at the value)
- Change of state (value) is based on the clock

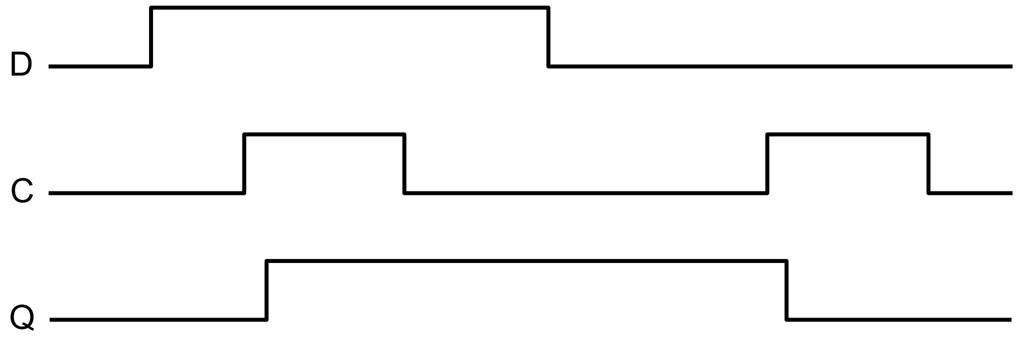
- Latches: whenever the inputs change, and the clock is asserted

- Flip-flop: state changes only on a clock edge
  (edge-triggered methodology)

D-latch

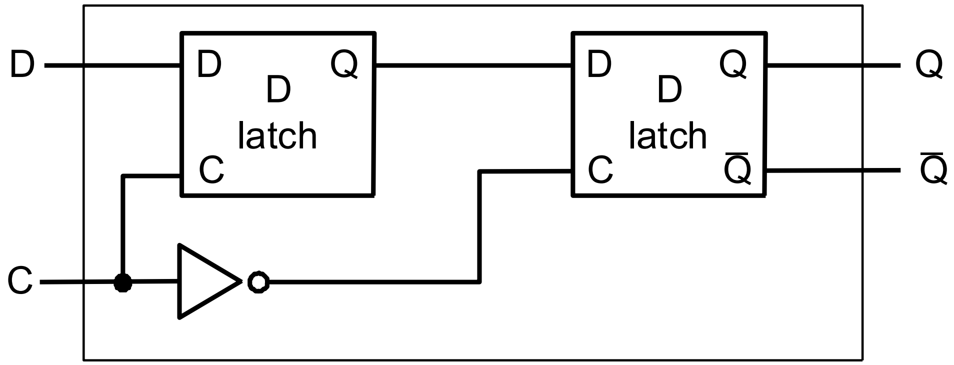
- Two inputs:
  - the data value to be stored (D)
  - the clock signal (C) indicating when to read & store D
- Two outputs:
  - the value of the internal state (Q) and it's complement

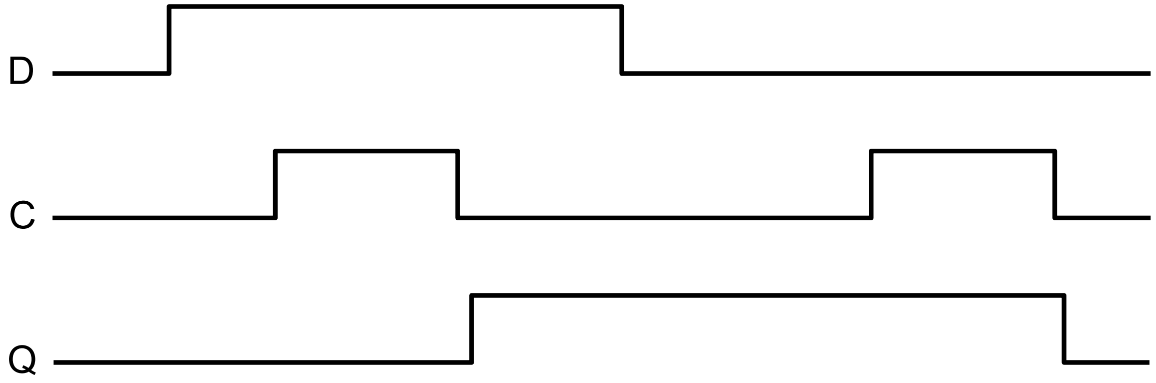




D flip-flop (Register)

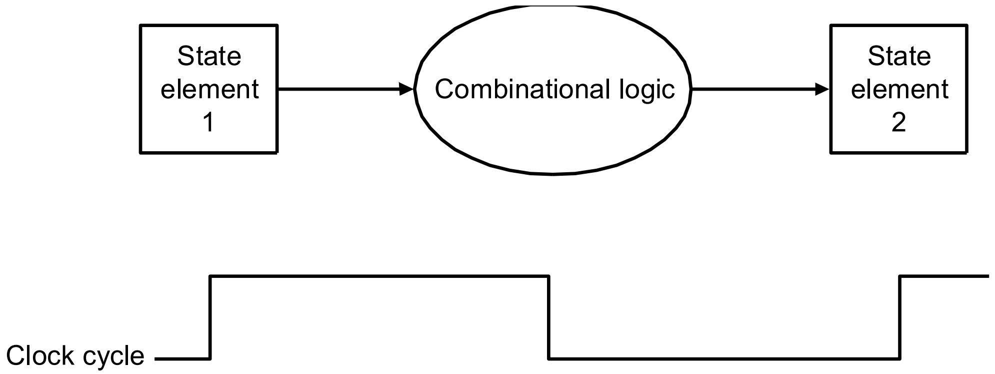
- Output changes only on the clock edge





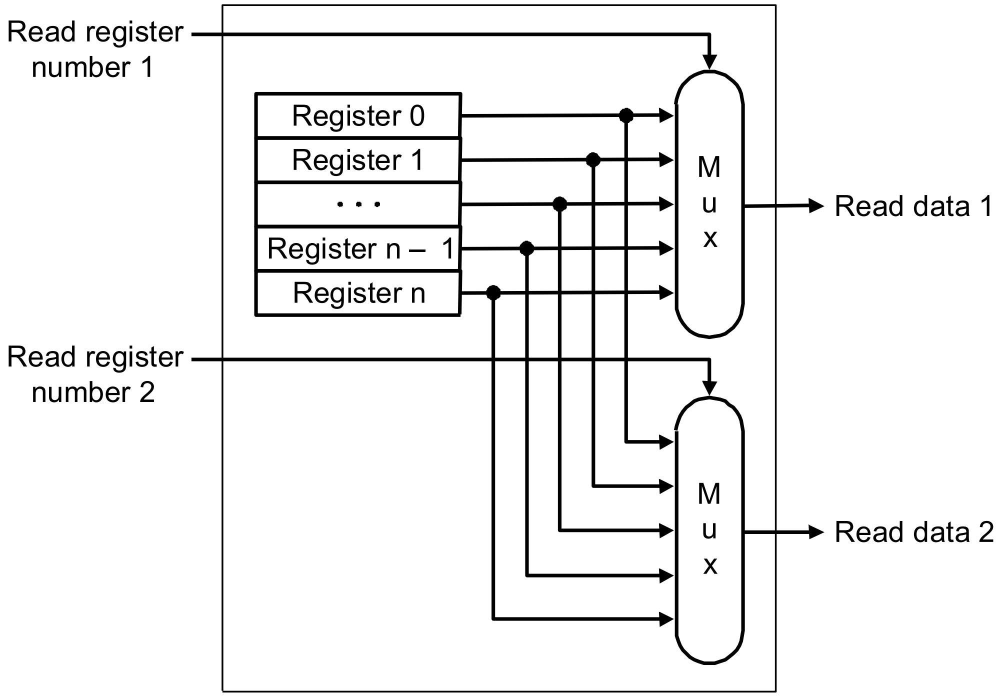
Our Implementation

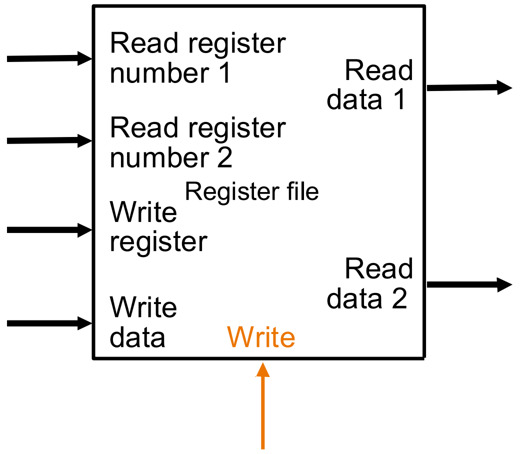
- An edge triggered methodology
- Typical execution:
  - Read contents of some state elements,
  - Send values through some combinational logic
  - Write results to one or more state elements



Register File

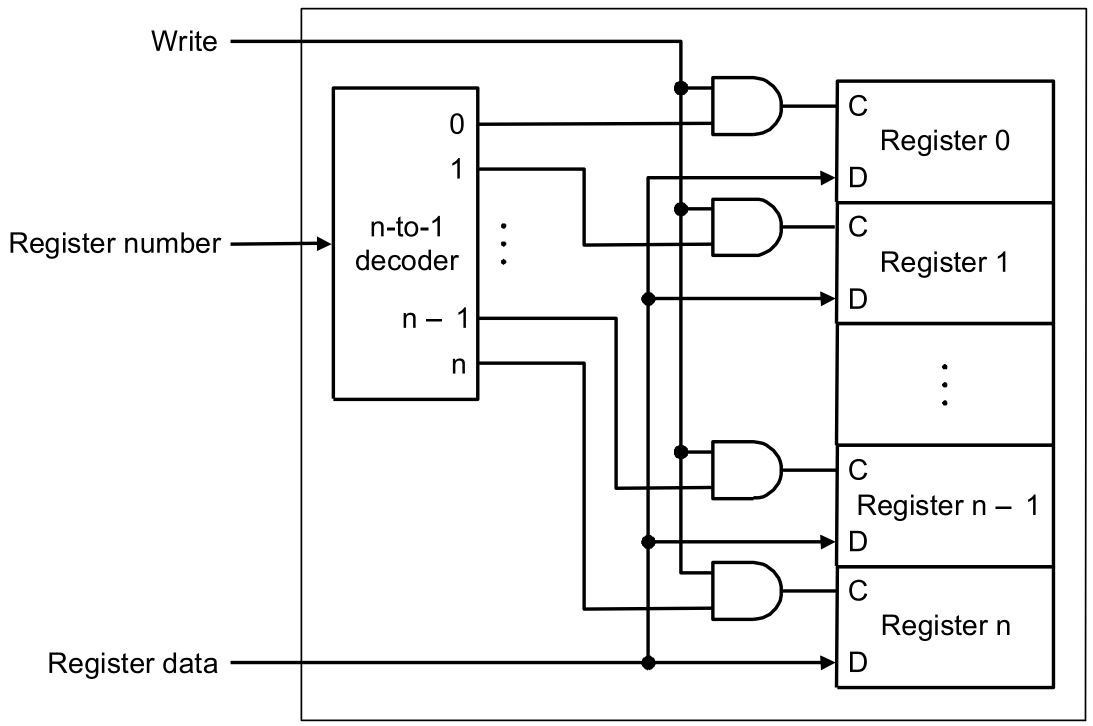
- Built using D flip-flips

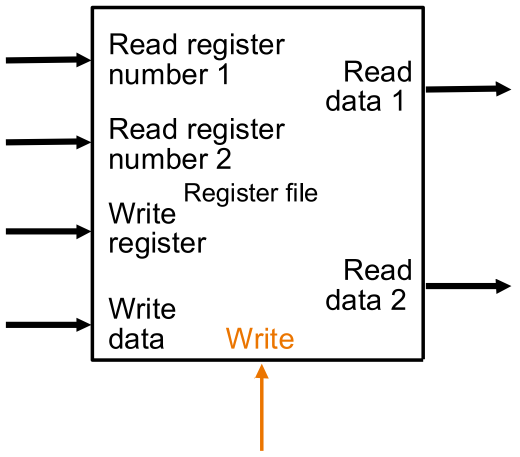




Register File

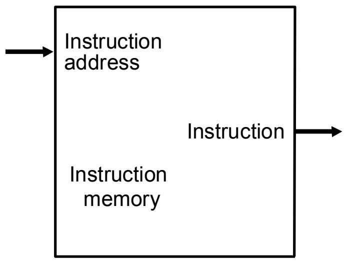
- Note: We still use the real clock to determine when to write



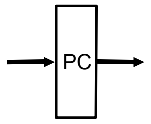


Simple Implementation

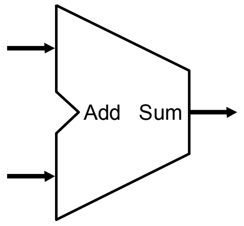
- Include the functional units we need for each instruction
- Think of this as a puzzle! ☺



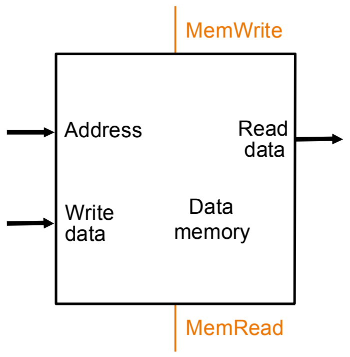
a. Instruction memory



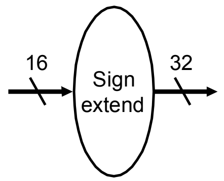
b. Program counter



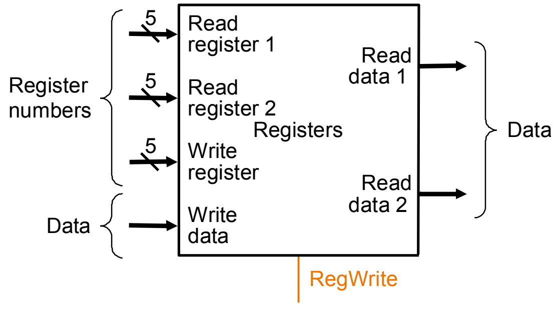
c. Adder



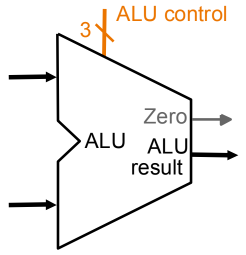
a. Data memory unit



b. Sign-extension unit



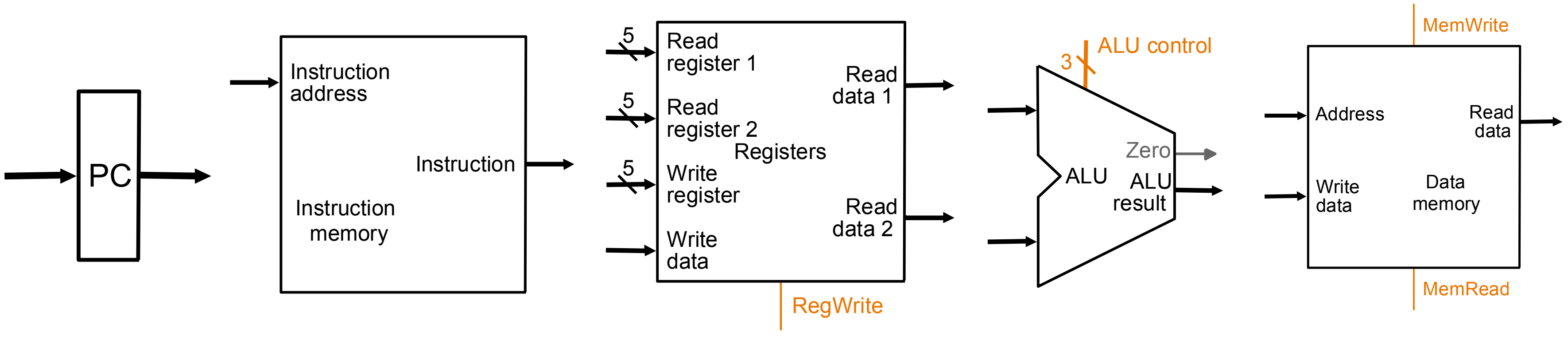
a. Registers



b. ALU

Instruction Ordering

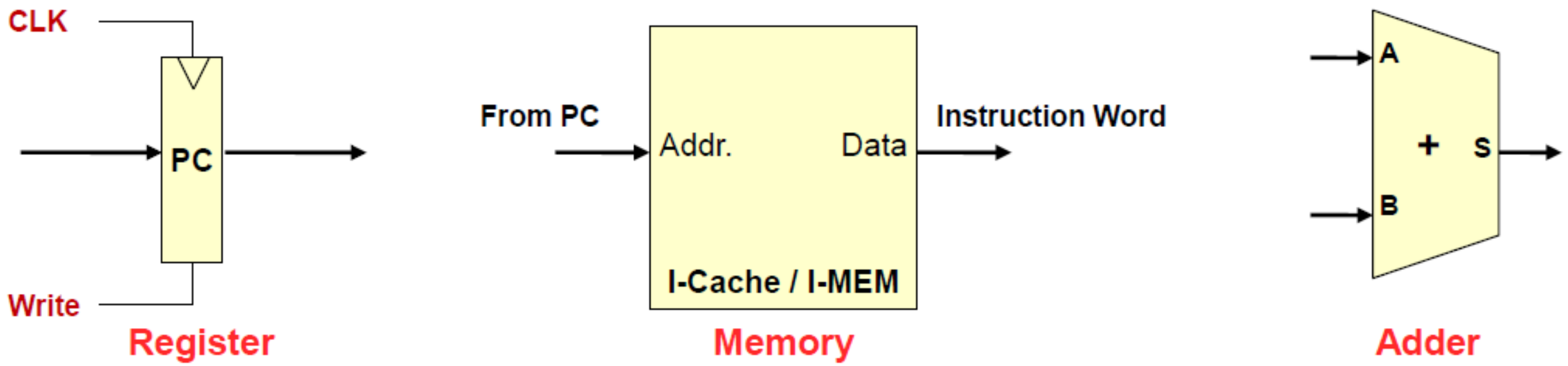
- Identify which components each instruction type would use and in what order: ALU-Type, LW, SW, BEQ



ALU-Type (ADD R3, R2, R1)	LW (LW R2, 4(R1))	SW (SW R2, 8(R1))	BEQ (BEQ R3, R1, displace)
1. PC	1. PC	1. PC	1. PC
2. I-Mem	2. I-Mem	2. I-Mem	2. I-Mem
3. Registers	3. Base. Reg.	3. Base. Reg	3. Register Access
4. ALU	4. ALU	4. ALU	4. Compare
5. WB to Reg.	5. Read Mem	5. Write Mem	5. If Zero, Update PC = PC+disp
	6. WB to Reg.		

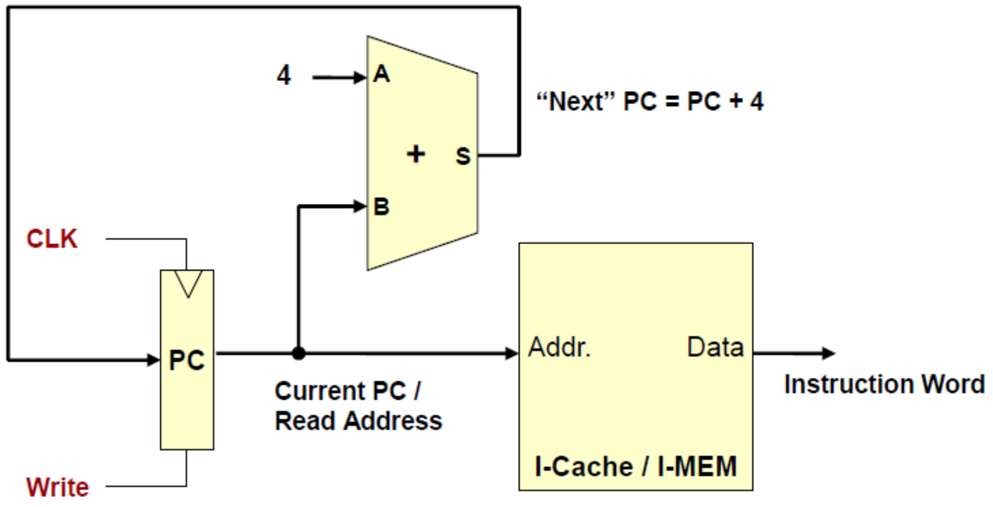
Fetch Components

- Required operations
  - Taking address from PC and reading instruction from memory
  - Incrementing PC to point at next instruction
- Components
  - PC register
  - Instruction Memory / Cache
  - Adder to increment PC value



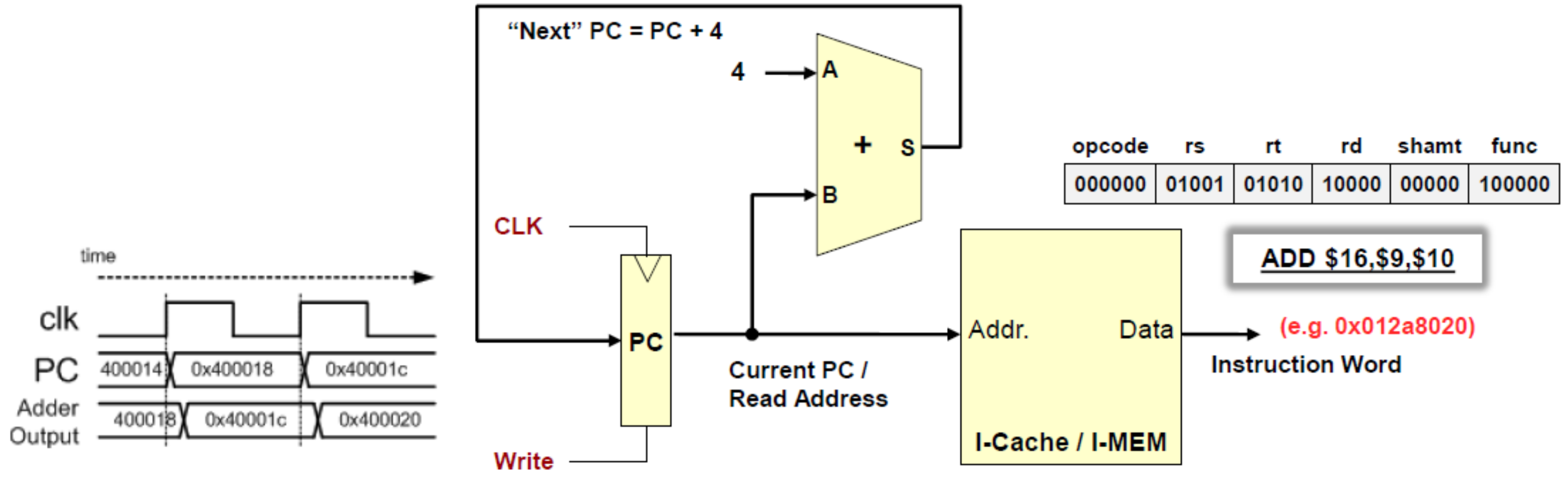
Fetch Datapath

- PC value serves as address to instruction memory while also being incremented by 4 using the adder
- Instruction word is returned by memory after some delay
- New PC value is clocked into PC register at end of clock cycle



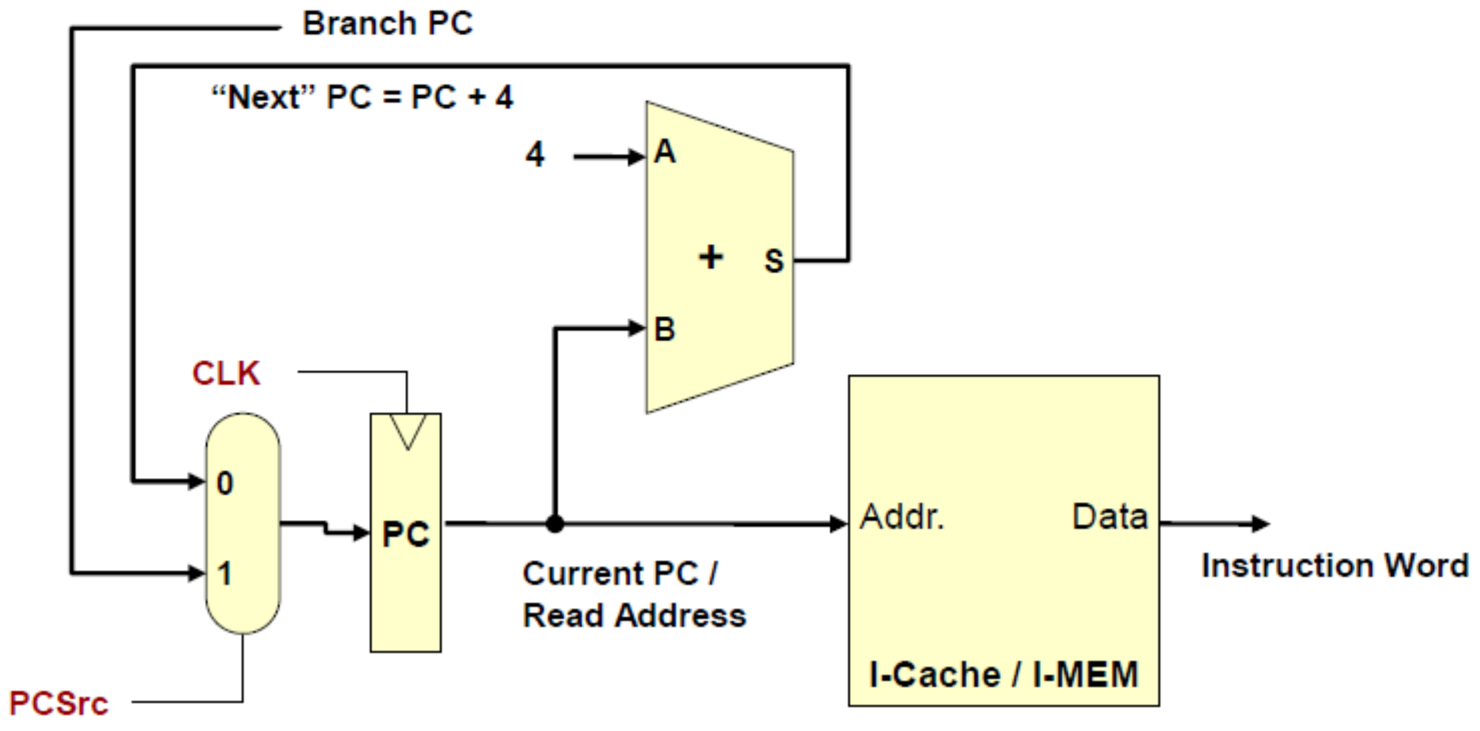
Fetch Datapath Example

- The PC and adder operation is shown
  - The PC doesn't update until the end of the current cycle
- The instruction being read out from the instruction memory
  - We have shown "assembly" syntax and the field by field machine code breakdown



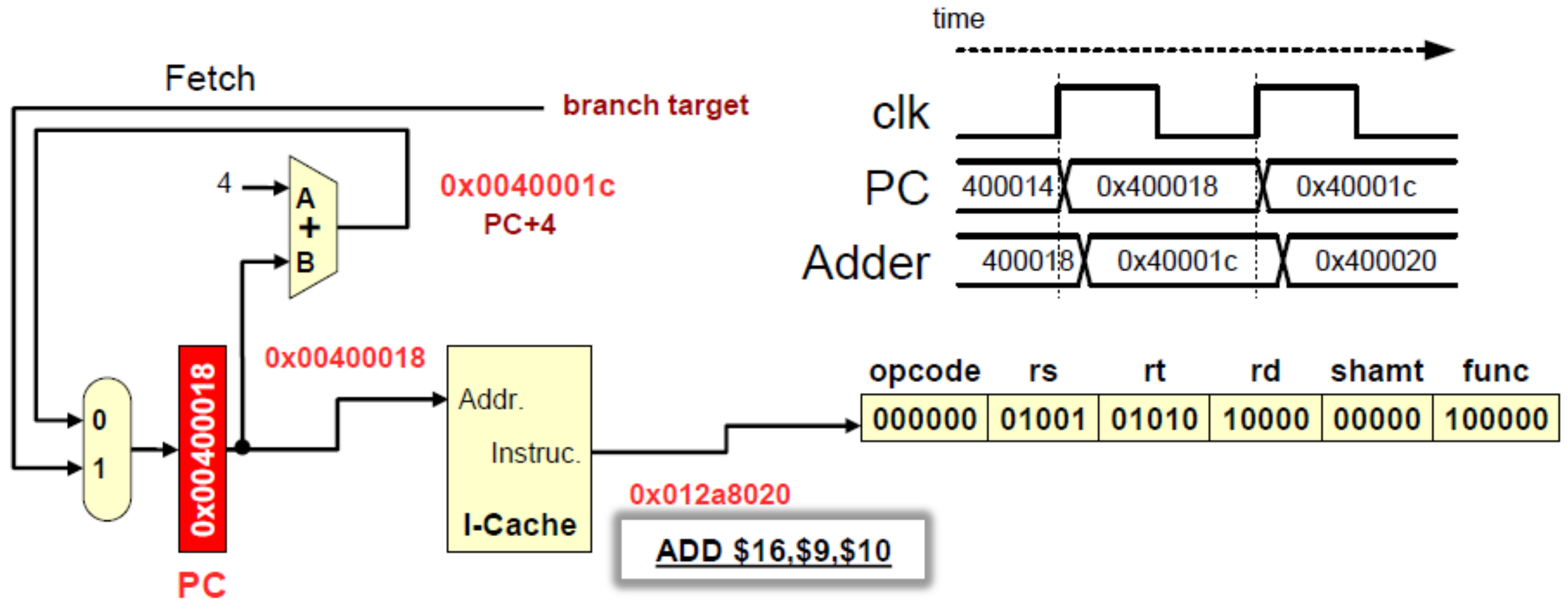
Modified Fetch Datapath

- Support for branch instruction added



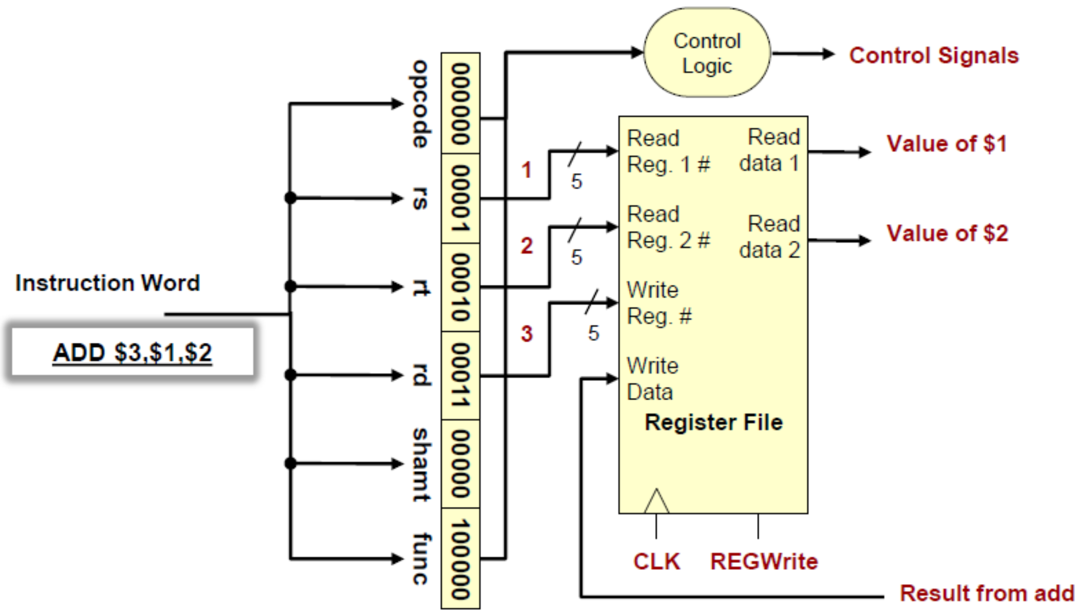
Modified Fetch Example

- Mux provides a path for branch target address



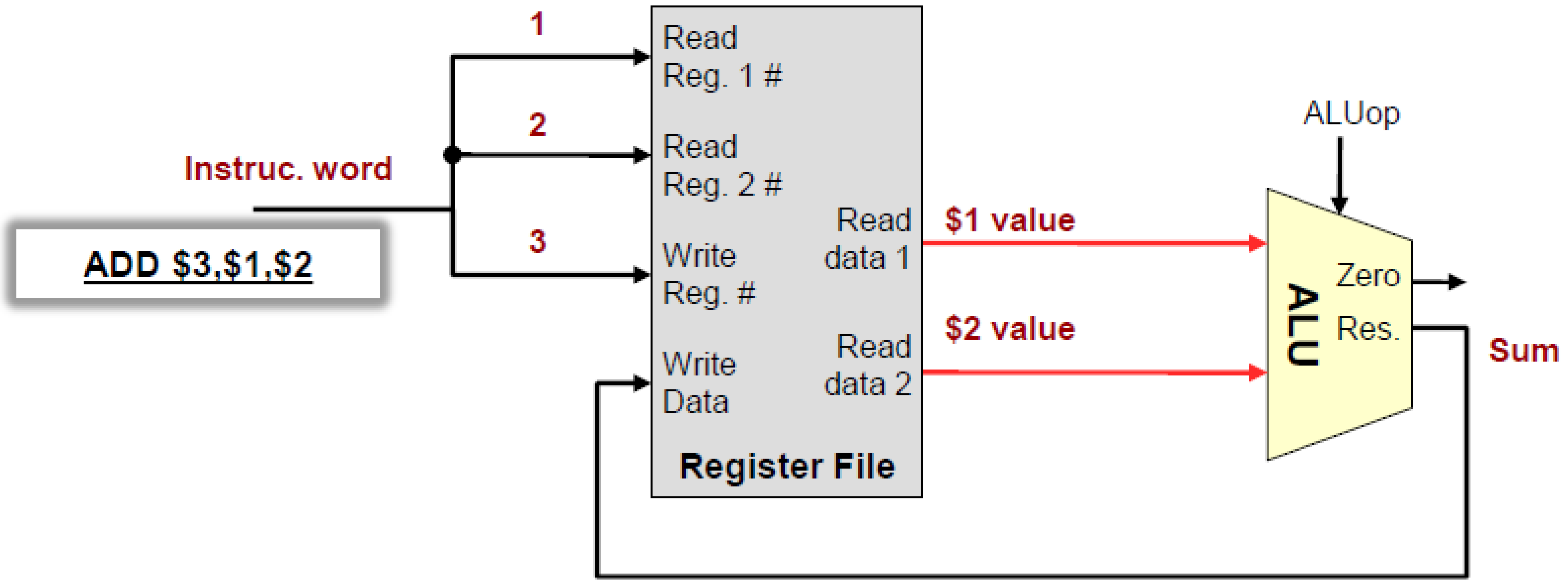
Decode

- Opcode and func. field are decoded to produce other control signals
- Execution of an ALU instruction (ADD $3,$1,$2) requires reading 2 register values and writing the result to a third
- REGWrite is an enable signal indicating the write data should be written to the specified register



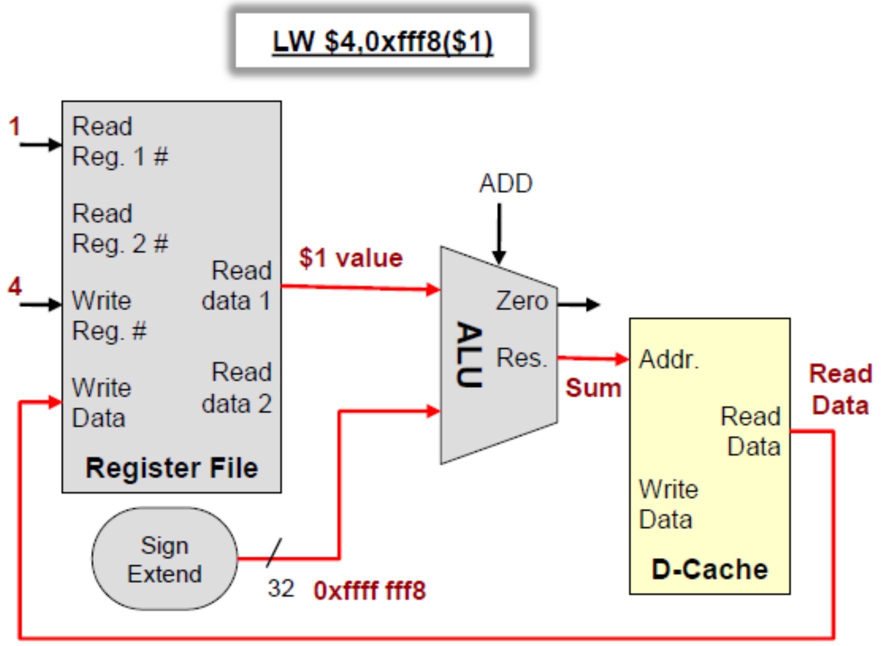
ALU Datapath

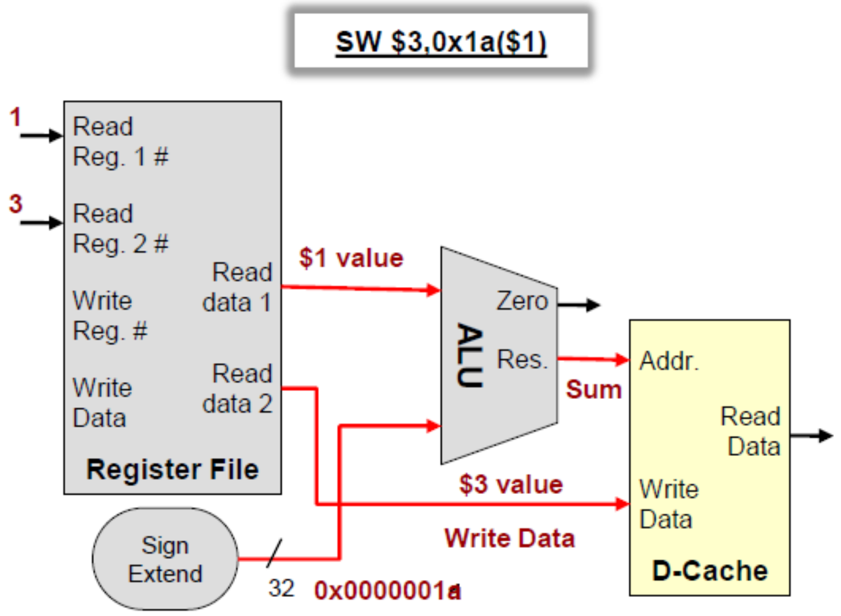
- ALU takes inputs from register file and performs the add, sub, and, or, slt operations
- Result is written back to dest. register



Memory Access Datapath

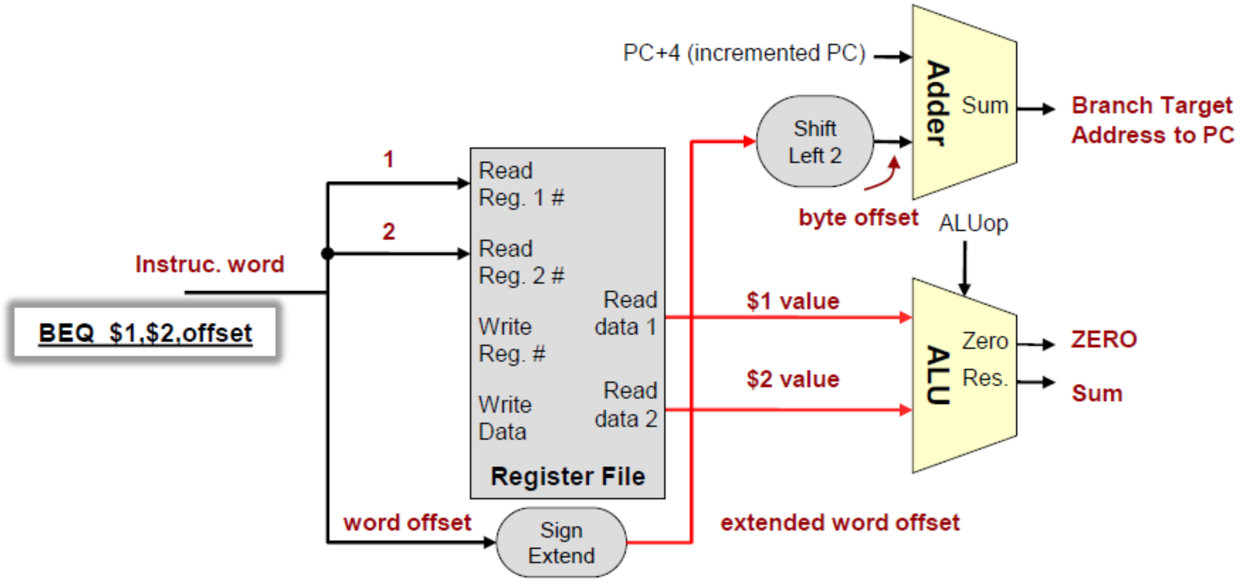
- Operands are read from register file while offset is sign extended
- ALU calculates effective address
- Memory access is performed
- If LW, read data is written back to register





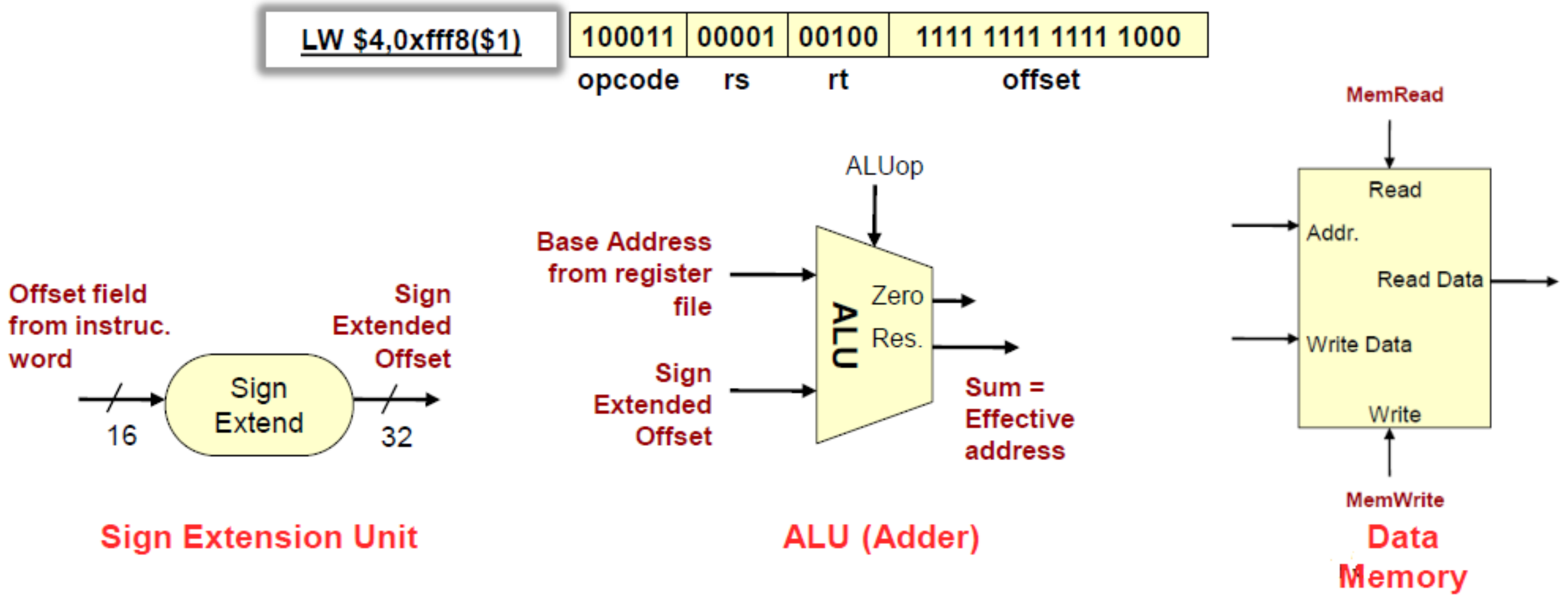
Branch Datapath

- BEQ requires…
  - ALU for comparison (examine 'zero' output)
  - Sign extension unit for branch offset
  - Adder to add PC and offset
    - Need a separate adder since ALU is used to perform comparison



Memory Access

- LW and SW require:
  - Sign extension unit for address offset
  - ALU to compute (add) base address + offset
  - Data memory

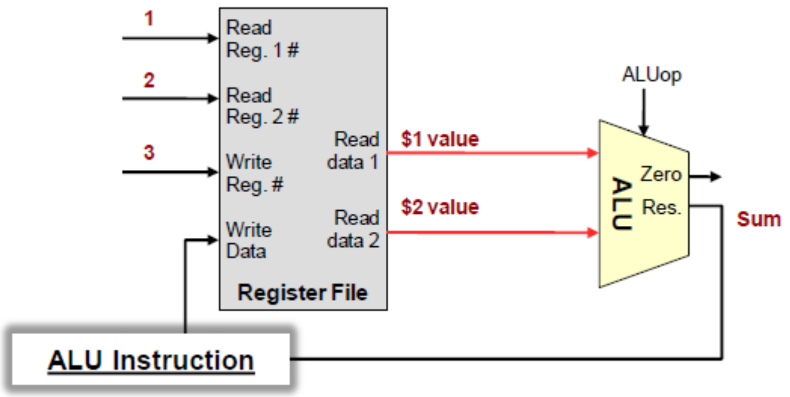


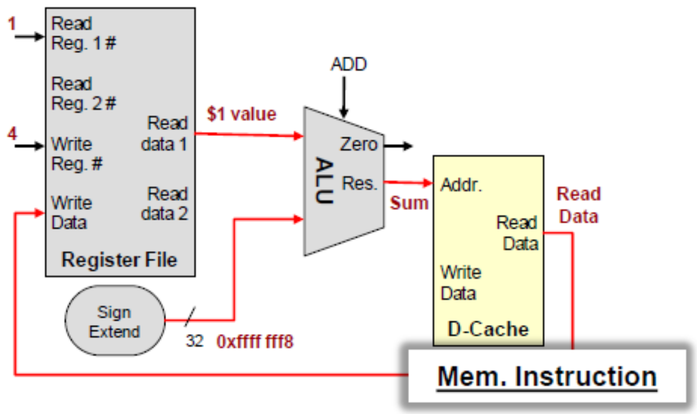
Combining Datapaths

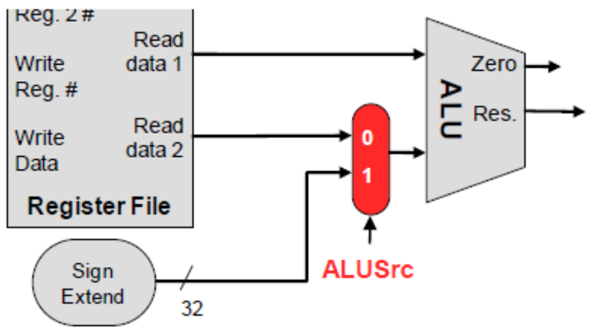
- Combine all datapaths into one
- We can have multiple options for certain inputs
- Use muxes to select appropriate input for a given instruction
- Generate control bits to control mux

ALUSrc Mux

- Mux controlling second input to ALU
  - ALU instruction provides Read Register 2 data to the 2nd input of ALU
  - LW/SW uses 2nd input of ALU as an offset to form effective address

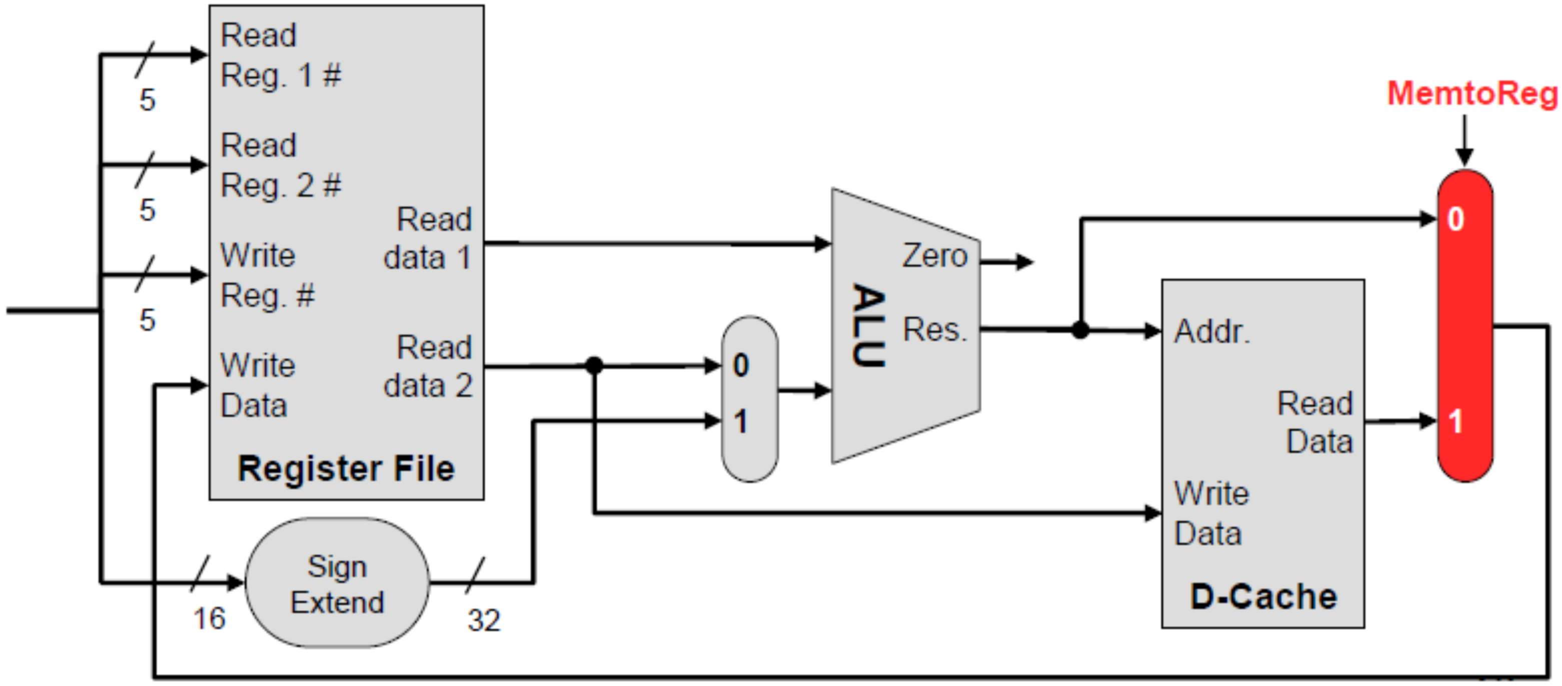






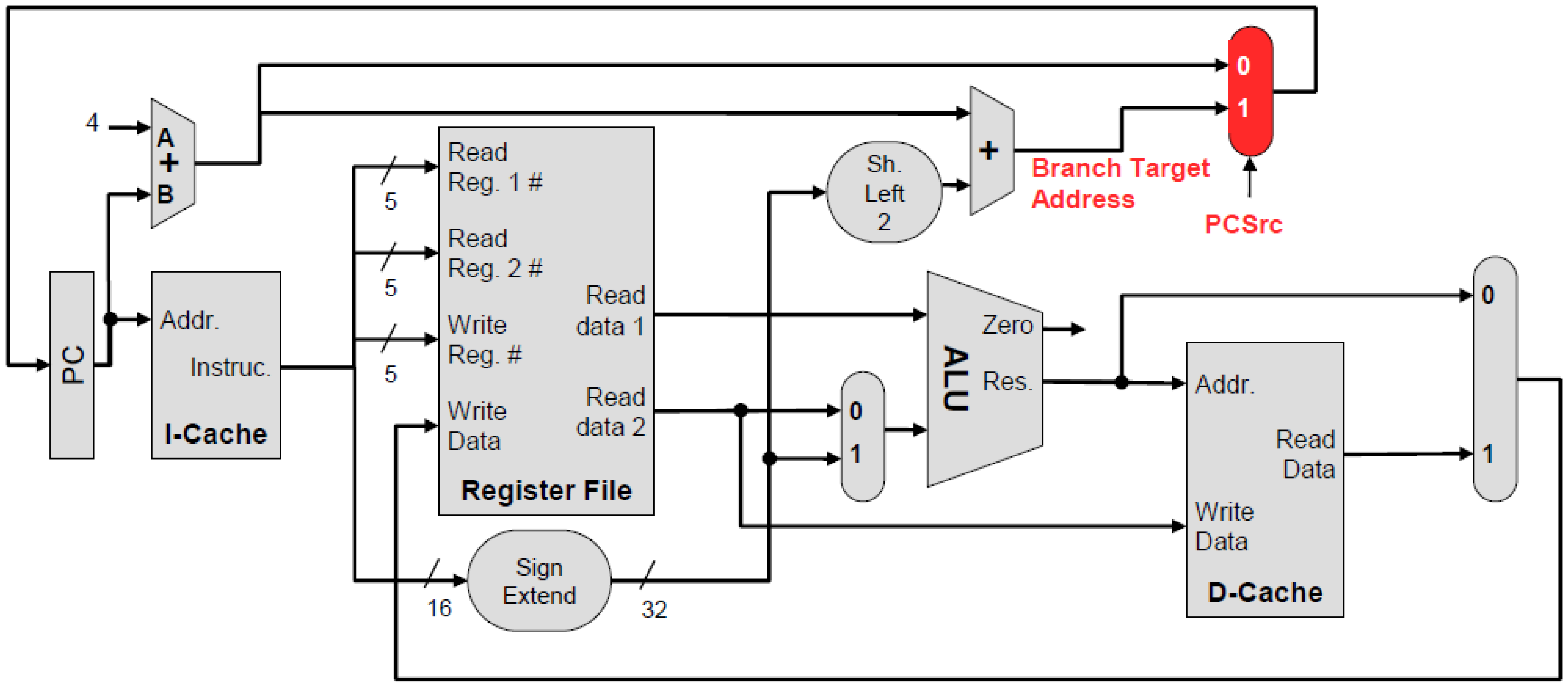
MemtoReg Mux

- Mux controlling writeback value to register file
  - ALU instructions use the result of the ALU
  - LW uses the read data from data memory



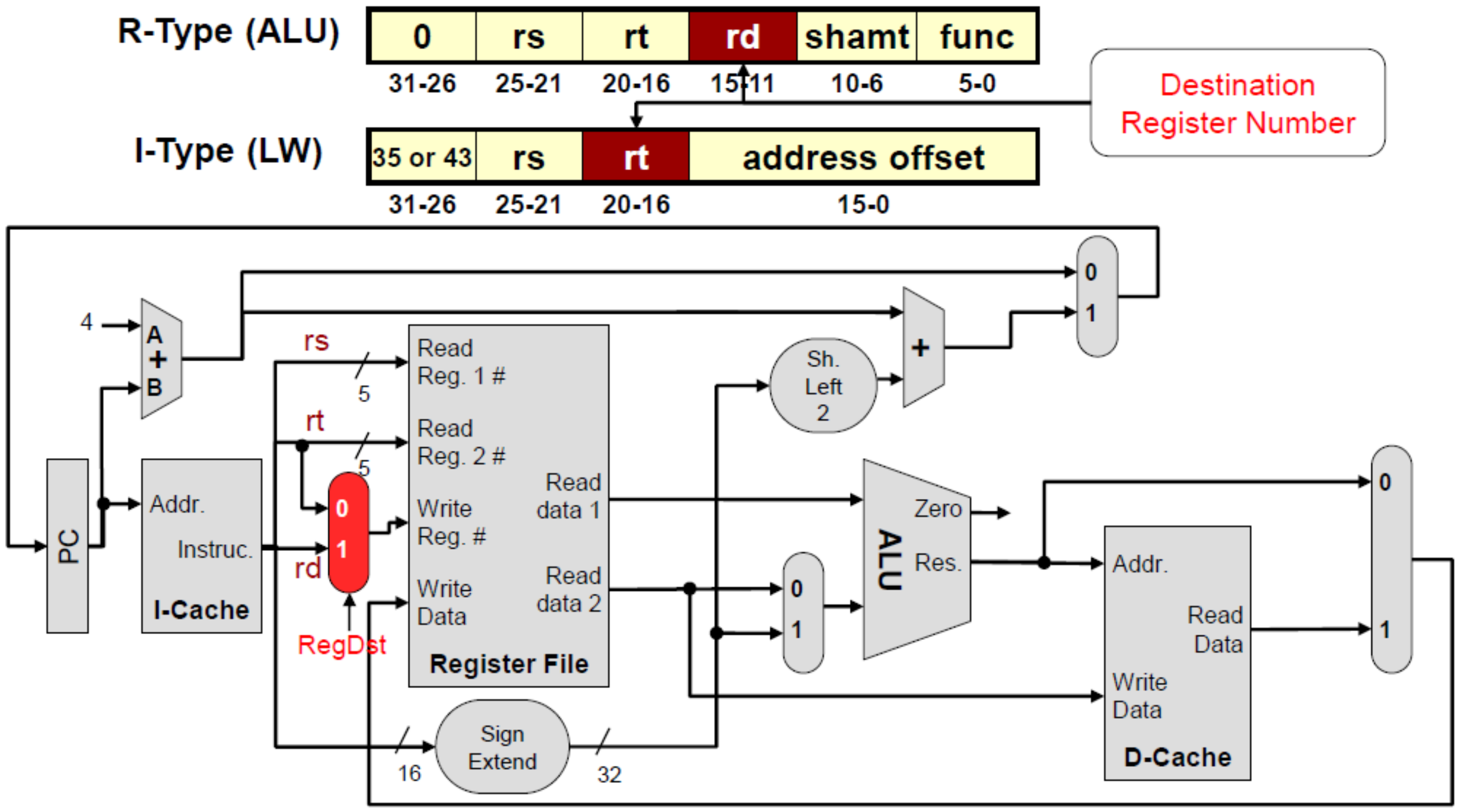
PCSrc Mux

- Next instruction can either be PC+4, or the branch target address, PC+Offset

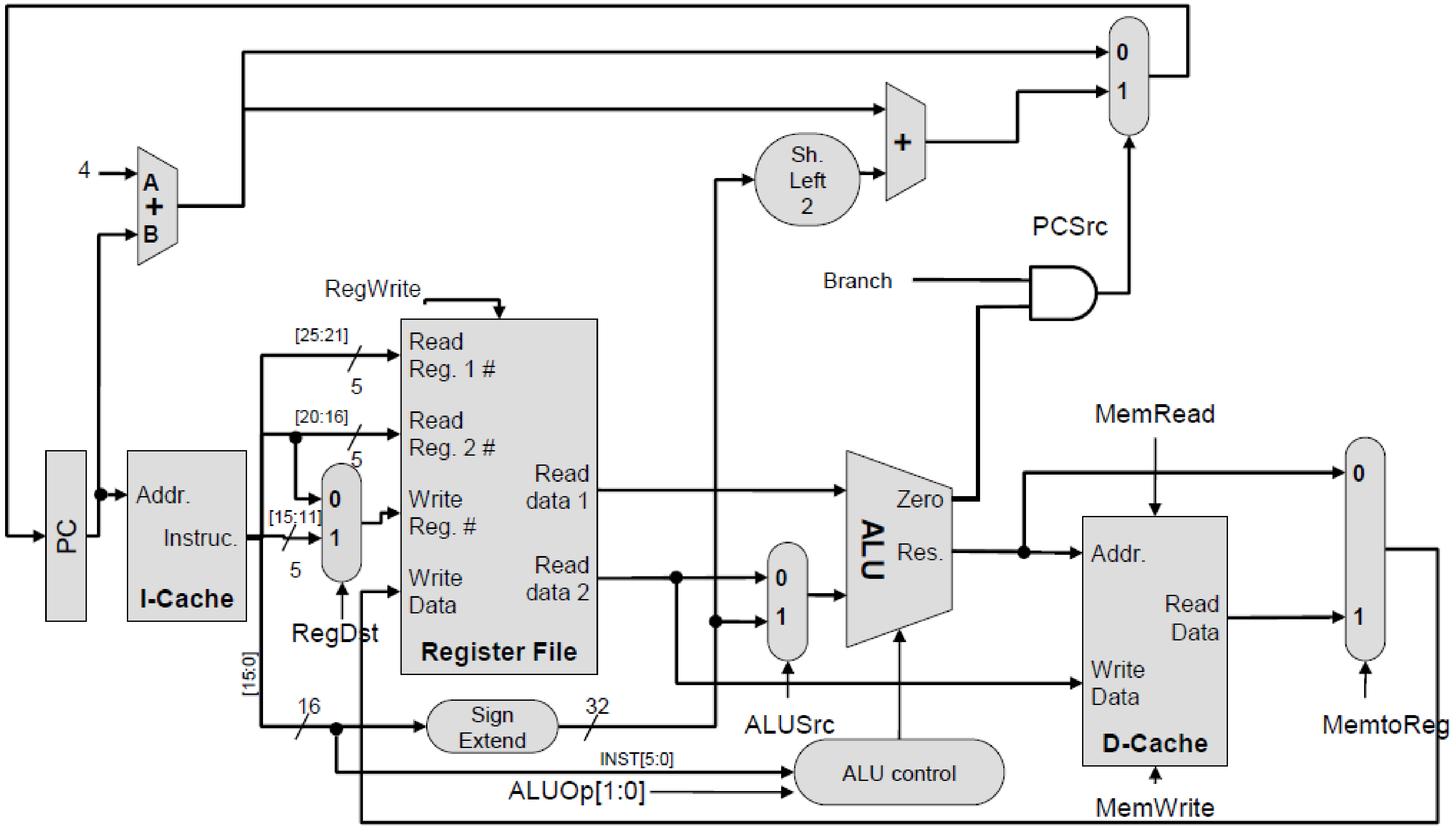


RegDst Mux

- Different destination register ID fields for ALU and LW instructions



Single Cycle CPU Datapath



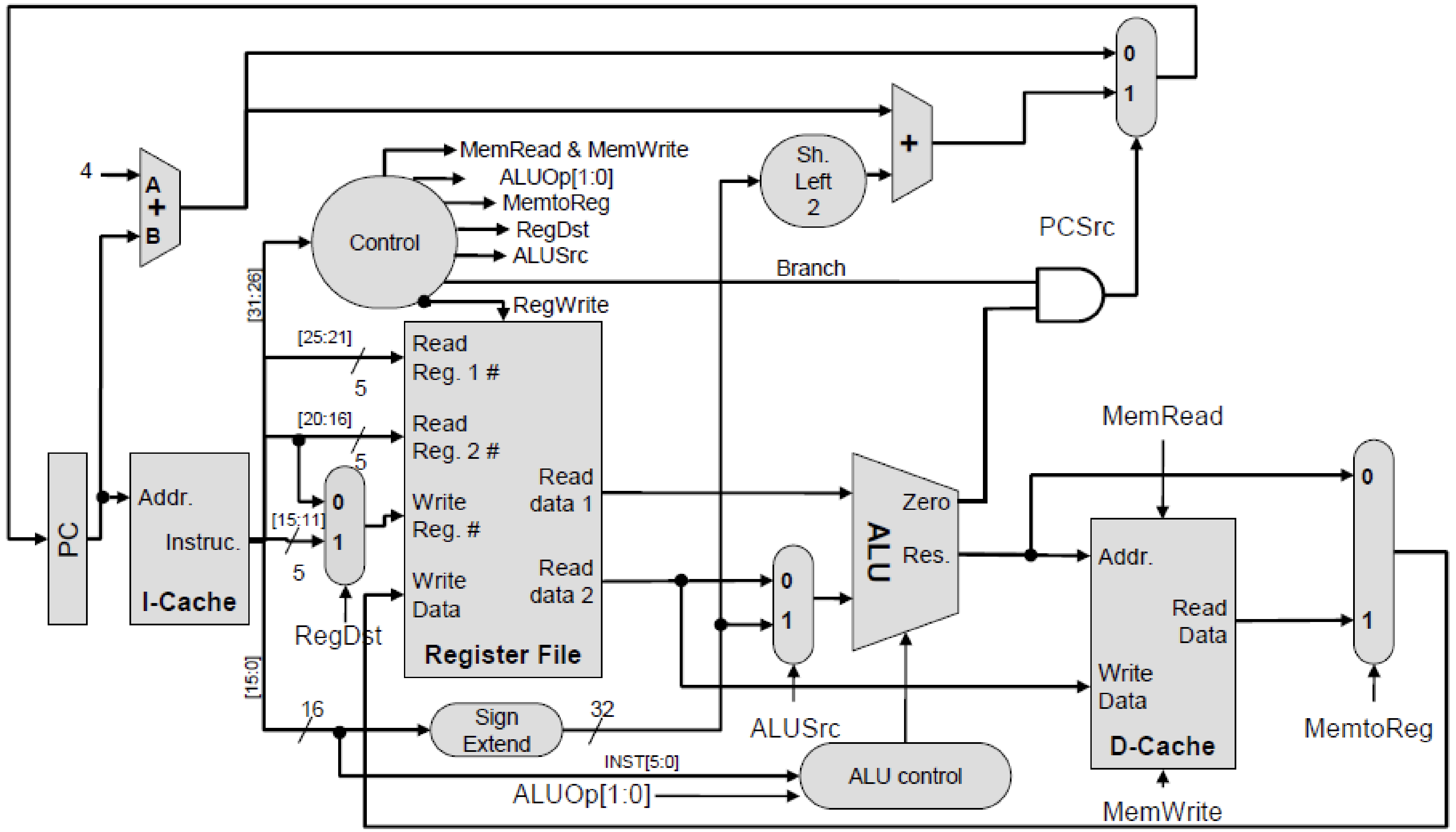
Control

- We now have data path in place, but how do we control which path an instruction will take?
- 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"?
  - Muxes
  - Writeable state elements: RF, Mem
  - ALU

Control

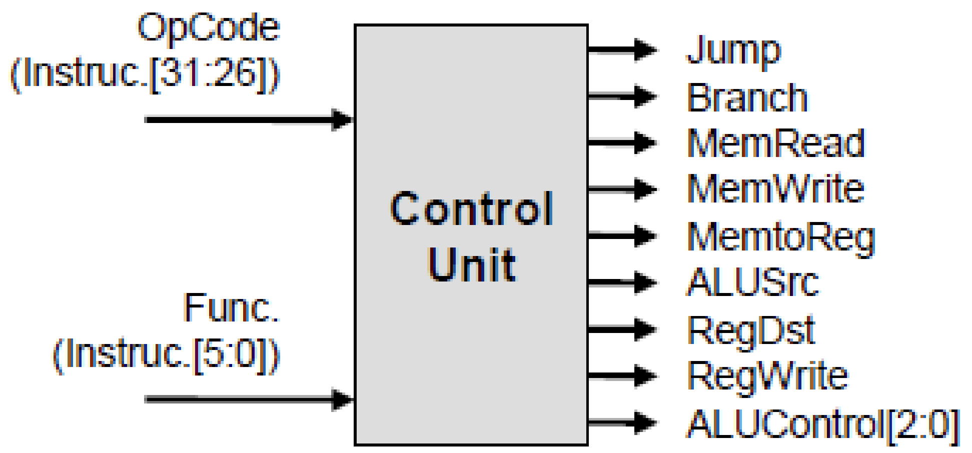
- Single-Cycle Design: everything happens in one clock cycle
  - until next falling edge of clock, processor just one big <u>combinational</u> <u>circuit</u>!!!
  - control is just a combinational circuit (output, just function of inputs)
- outputs? control points in datapath
- inputs? the current instruction! (`opcode`, `funct` control everything)

Datapath + Control



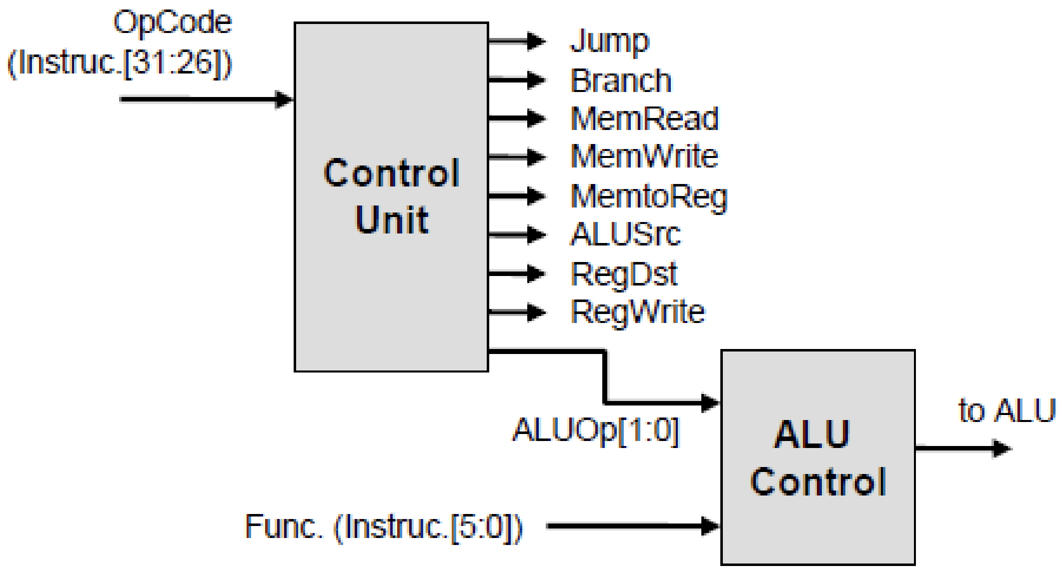
Defining Control

- Most control signals are a function of the opcode (i.e. LW/SW, R-Type, Branch, Jump)



Defining Control

- Funct field only present in R-type instruction
  - Funct controls ALU only
- To simplify control, define Main and ALU control separately

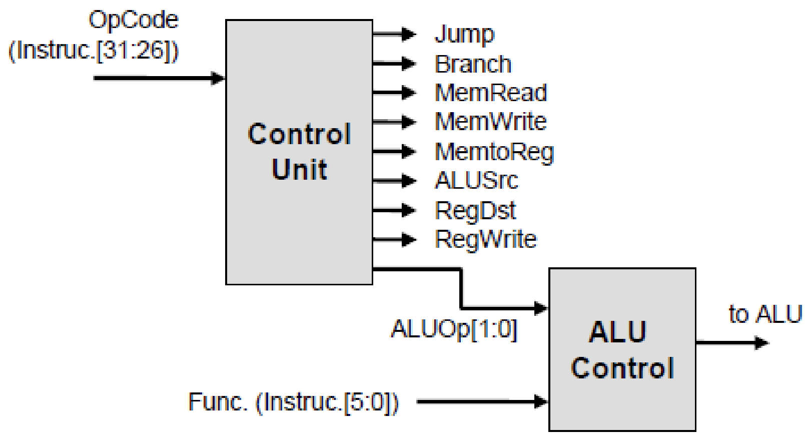


ALU Control

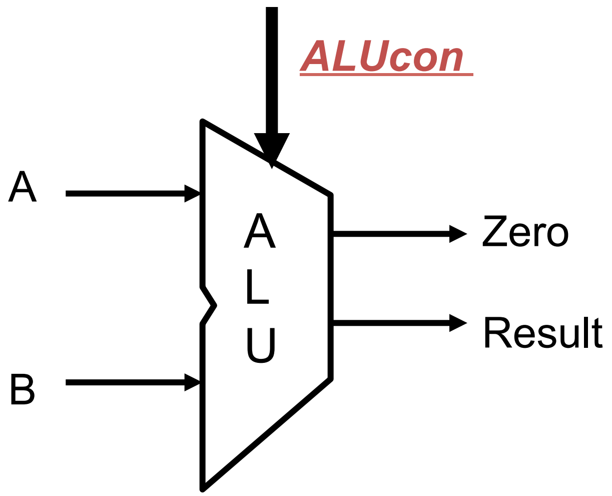
- ALU Control needs to know what instruction type it is:
  - R-Type (op. depends on func. code)
  - LW/SW (op. = ADD)
  - BEQ (op. = SUB)
- Let main control unit produce ALUOp[1:0] to indicate instruction type, then use function bits if necessary to tell the ALU what to do

Instruction	ALUOp[1:0]
LW/SW	00
Branch	01
R-Type	10

Control unit maps instruction opcode to ALUOp[1:0] encoding



ALU Control



Note: We don't use NOR.
Ignore MSB in ALUCon.

ALUCon	ALU function	Instruction 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)
1100	NOR	R-format (Nor)

ALU Control Truth Table

- ALUControl[2:0] is a function of ALUOp[1:0] and Func[5:0]

Instruc.	Instruction Operation	Desired ALU Action	ALUOp[1:0]	Func[5:0]	ALUControl
LW	Load Word	Add	00	X	010
SW	Store Word	Add	00	X	010
Branch	BEQ	Subtract	01	X	110
R-Type	AND	And	10	100100	000
R-Type	OR	Or	10	100101	001
R-Type	Add	Add	10	100000	010
R-Type	Sub	Subtract	10	100010	110
R-Type	SLT	Set on less than	10	101010	111

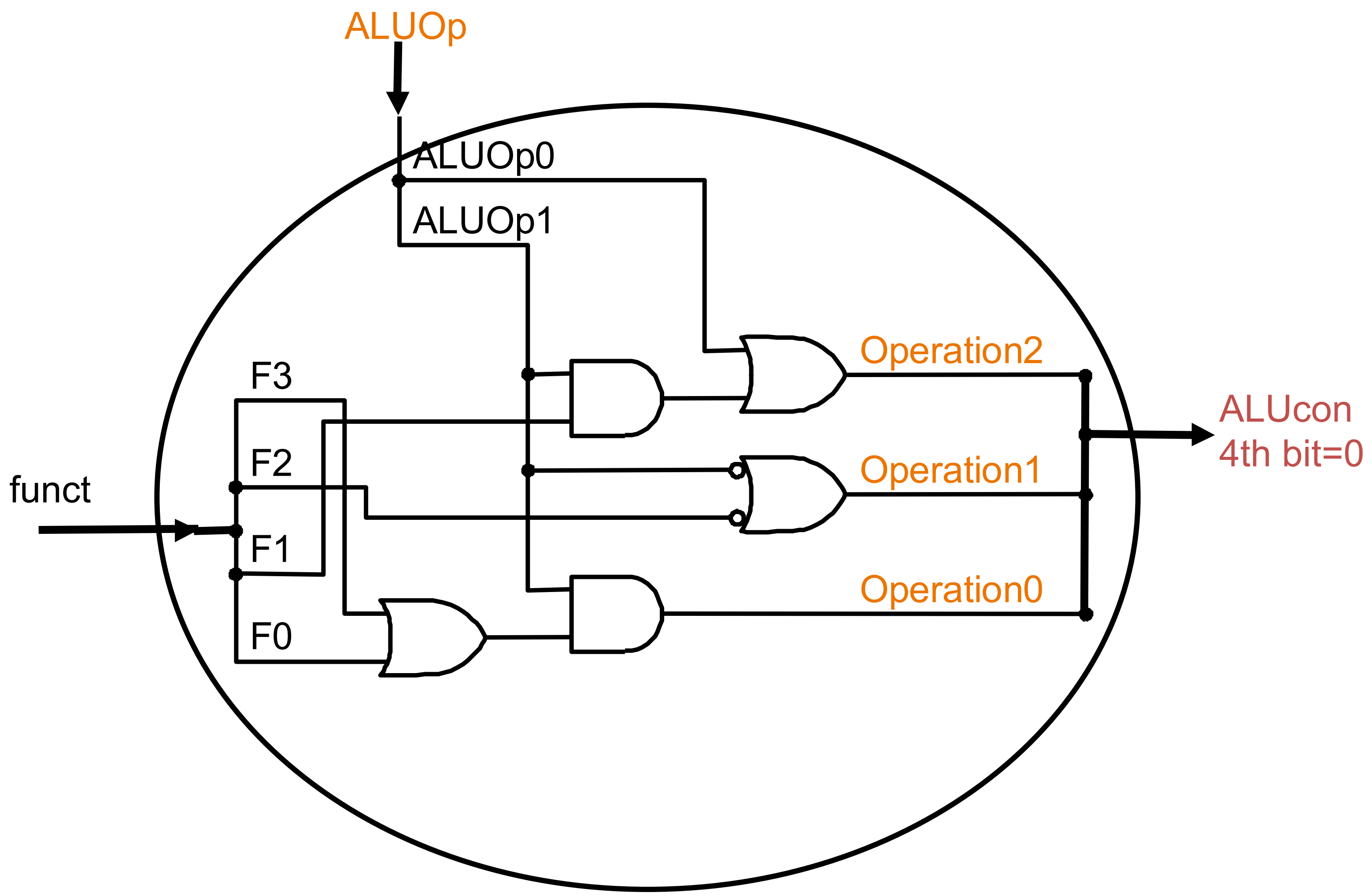
Simplified ALUControl Truth Table

- We can simplify using don't cares
- Can turn into gates

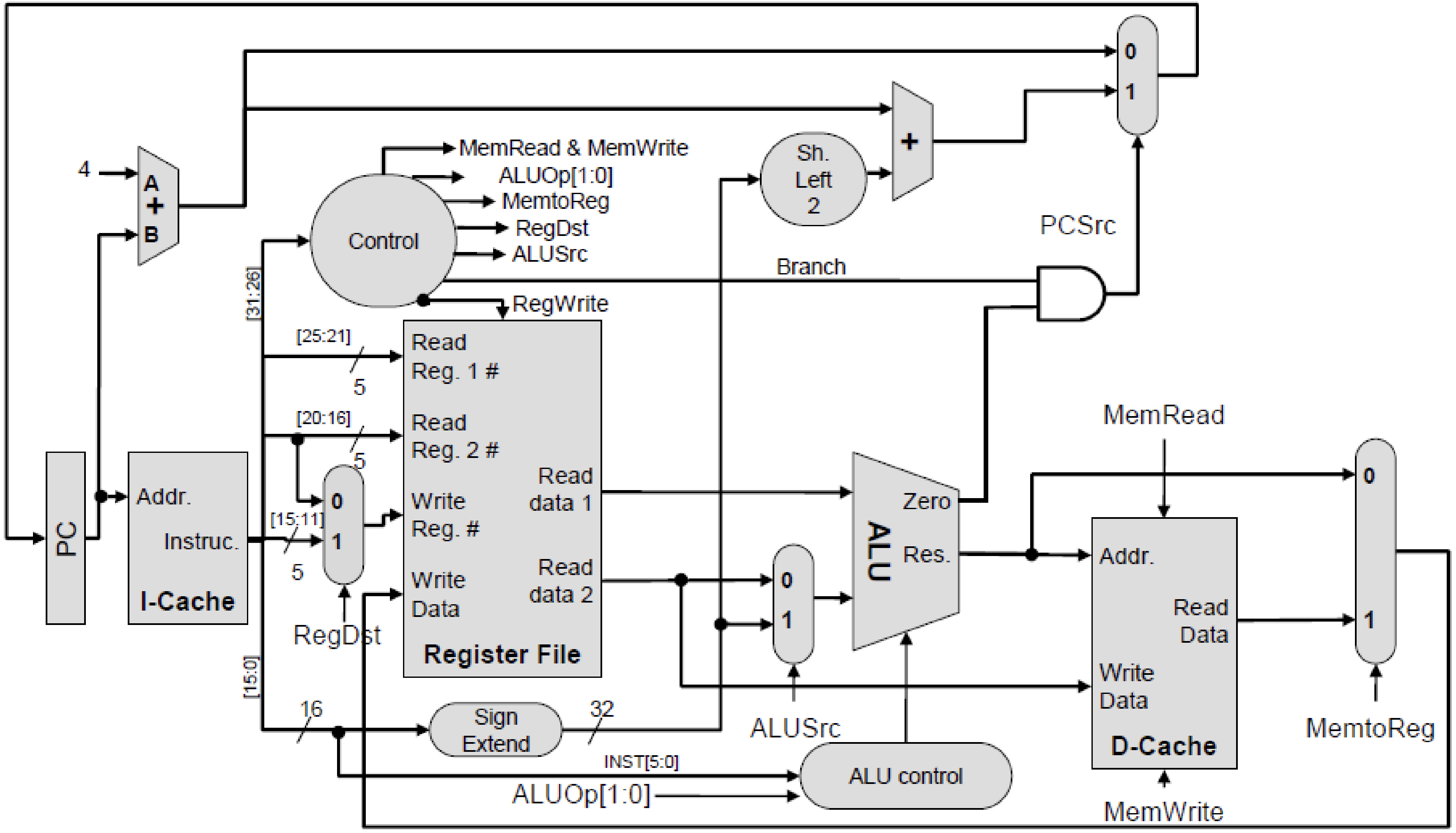
ALUOp		Funct Field						ALUCont.
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

- ALUControl2 = ALUOp0 OR (ALUOp1 AND F1)
- ALUControl1 = ALUOp1 NOR F2
- ALUControl0 = ALUOp1 AND (F3 OR F0)

Fully Minimized ALU Control

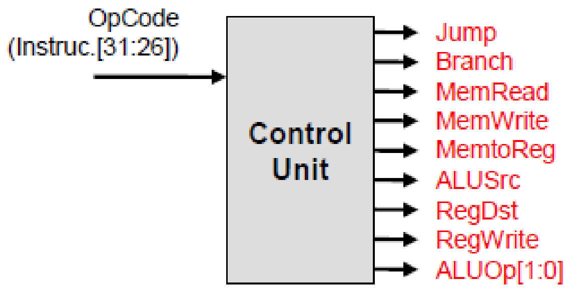


Datapath + Control

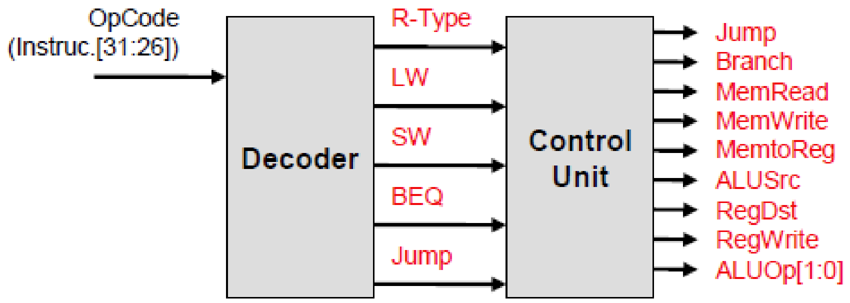


Main Control Signals

- Main control signals are function of opcode

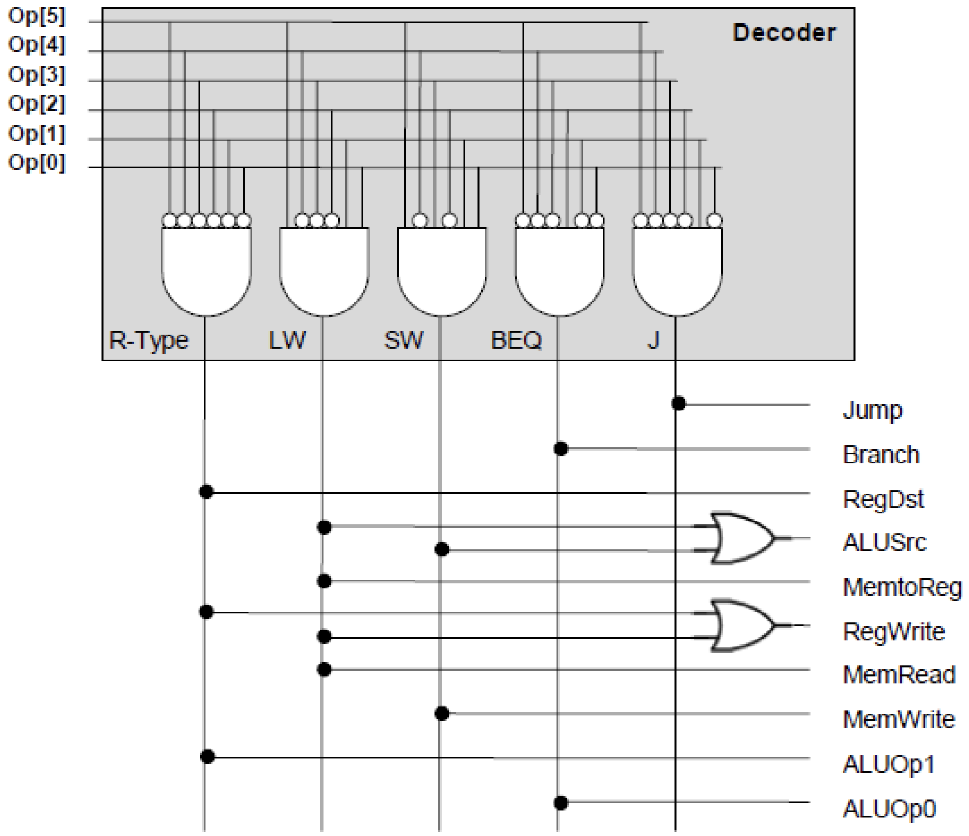


Could generate each control signal by writing full truth table of the 6-bit opcode



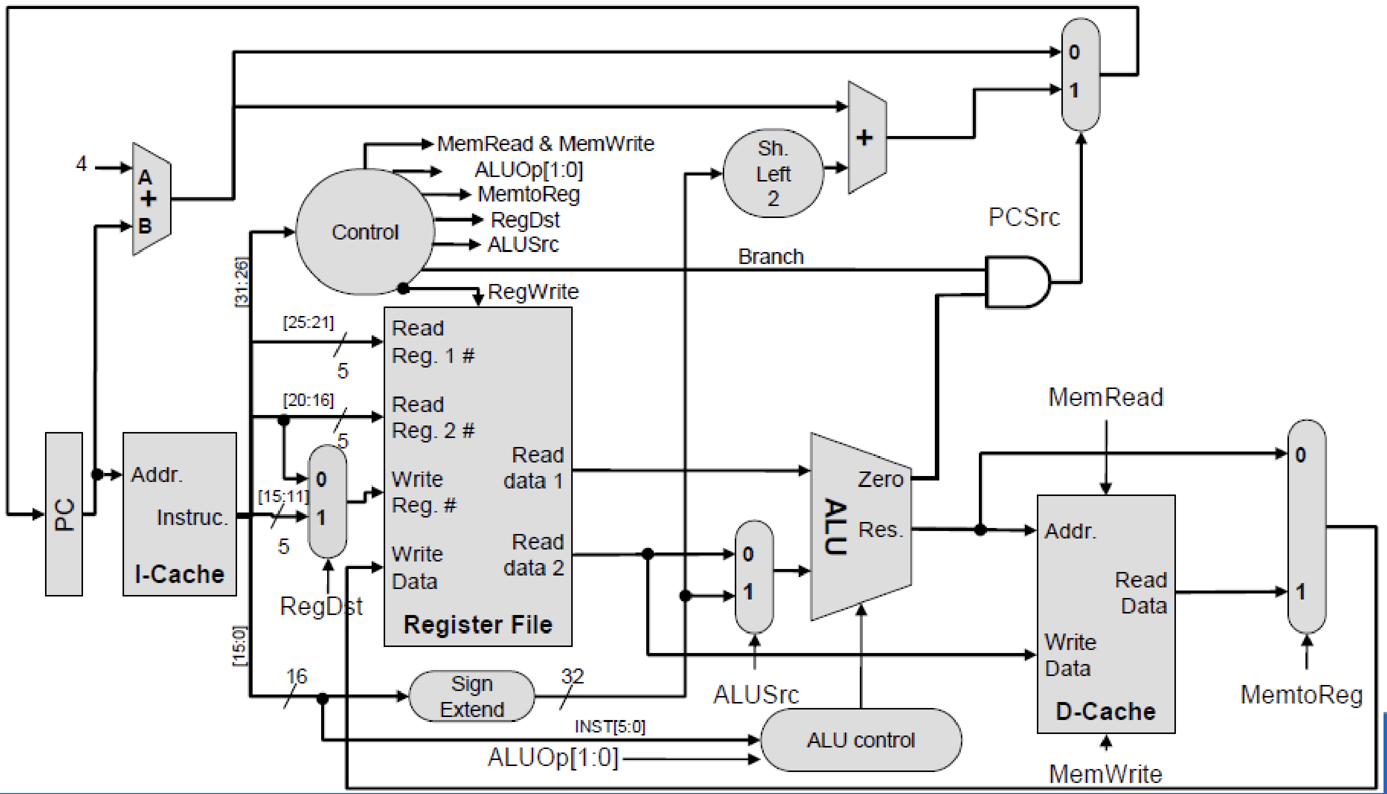
Simpler for humans to design if we decode the opcode and then use instruction signals to generate desired control signals

Main Control Signal Logic

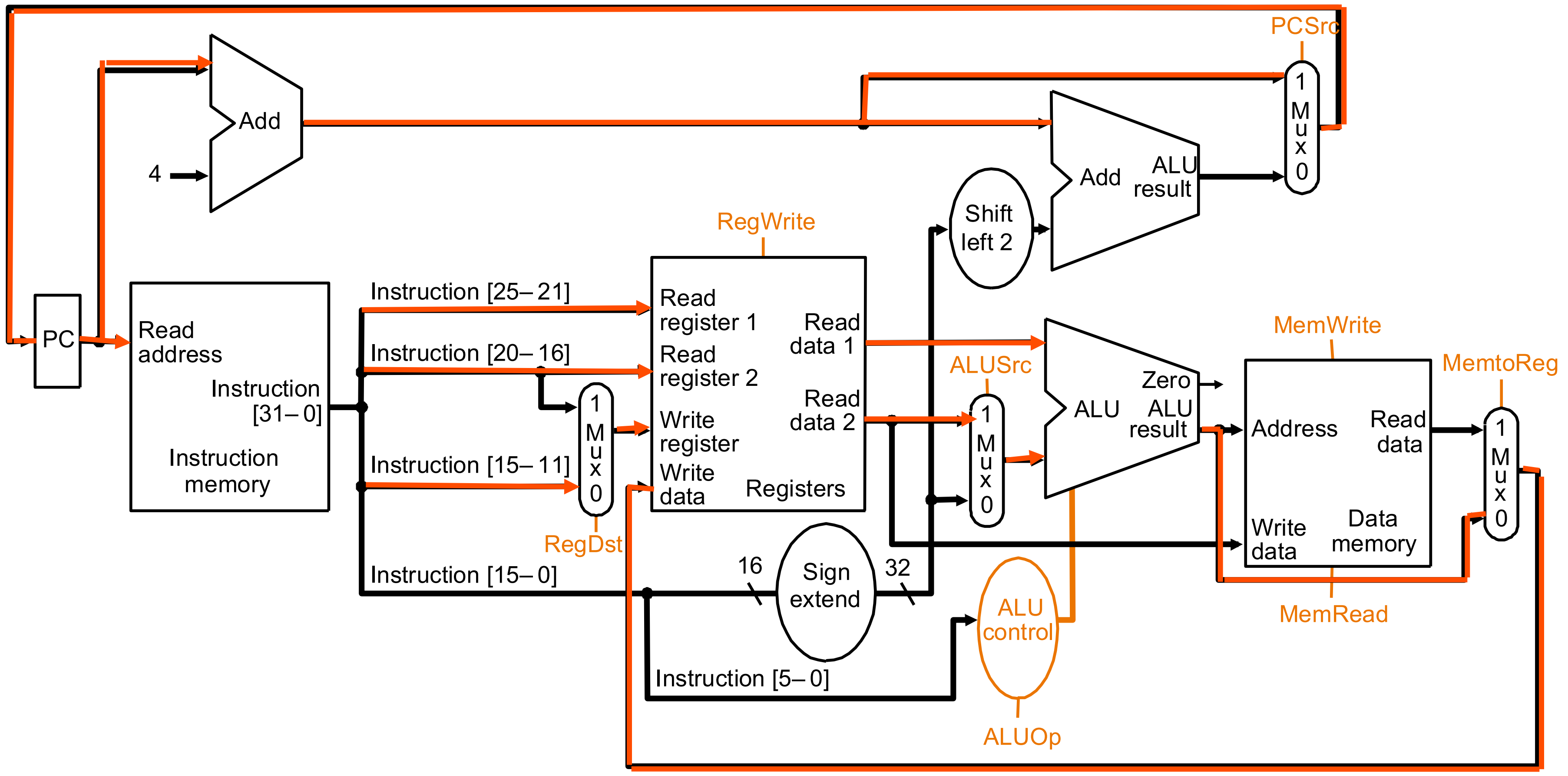


Main Control Signal Truth Table

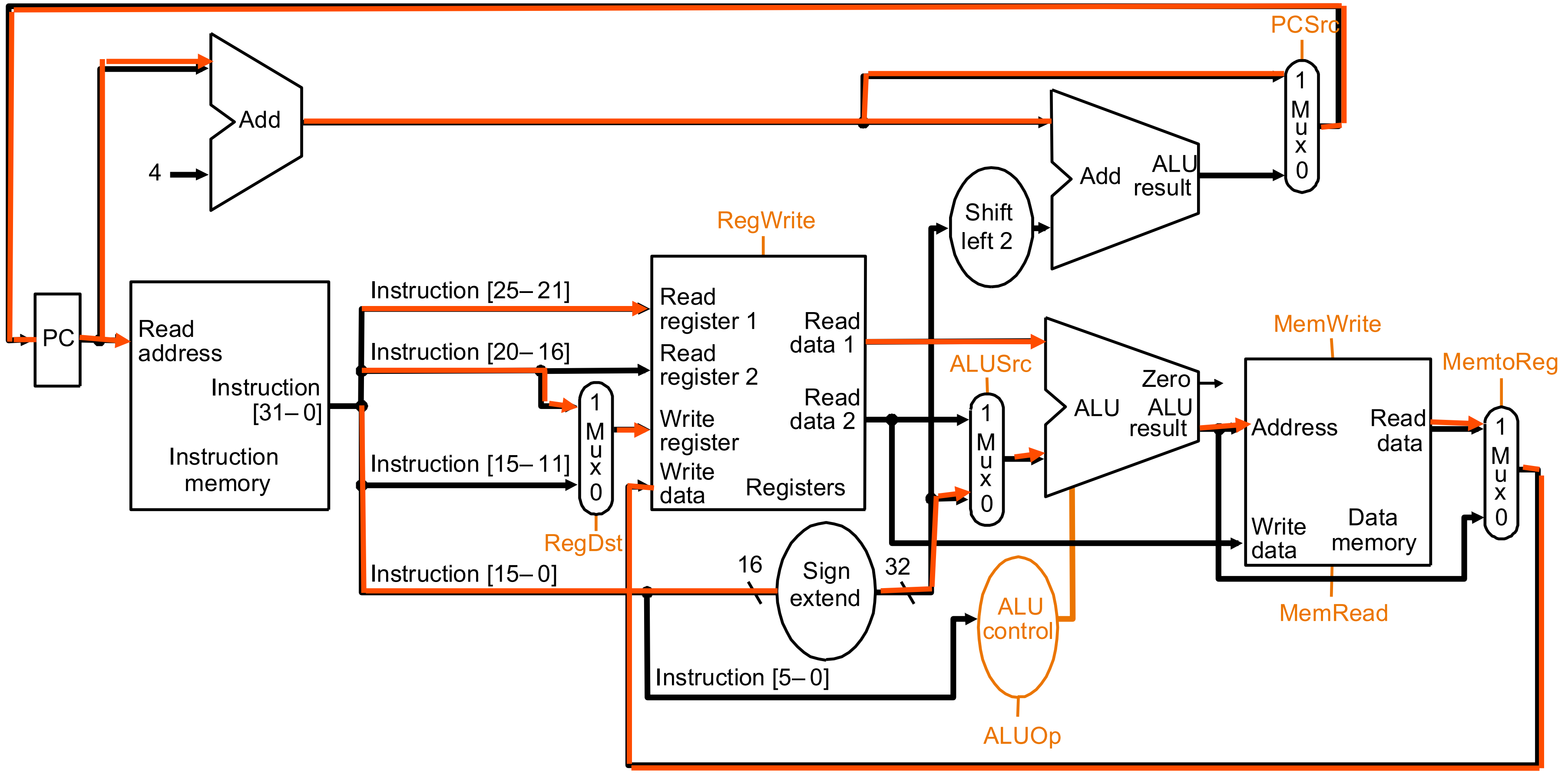
R-Type	LW	SW	BEQ	J	Jump	Branch	Reg Dst	ALU Src	Memto-Reg	Reg Write	Mem Read	Mem Write	ALU Op[1]	ALU Op[0]
1	0	0	0	0	0	0	1	0	0	1	0	0	1	0
0	1	0	0	0	0	0	0	1	1	1	1	0	0	0
0	0	1	0	0	0	0	X	1	X	0	0	1	0	0
0	0	0	1	0	0	1	X	0	X	0	0	0	0	1
0	0	0	0	1	1	0	X	X	X	0	0	0	X	X



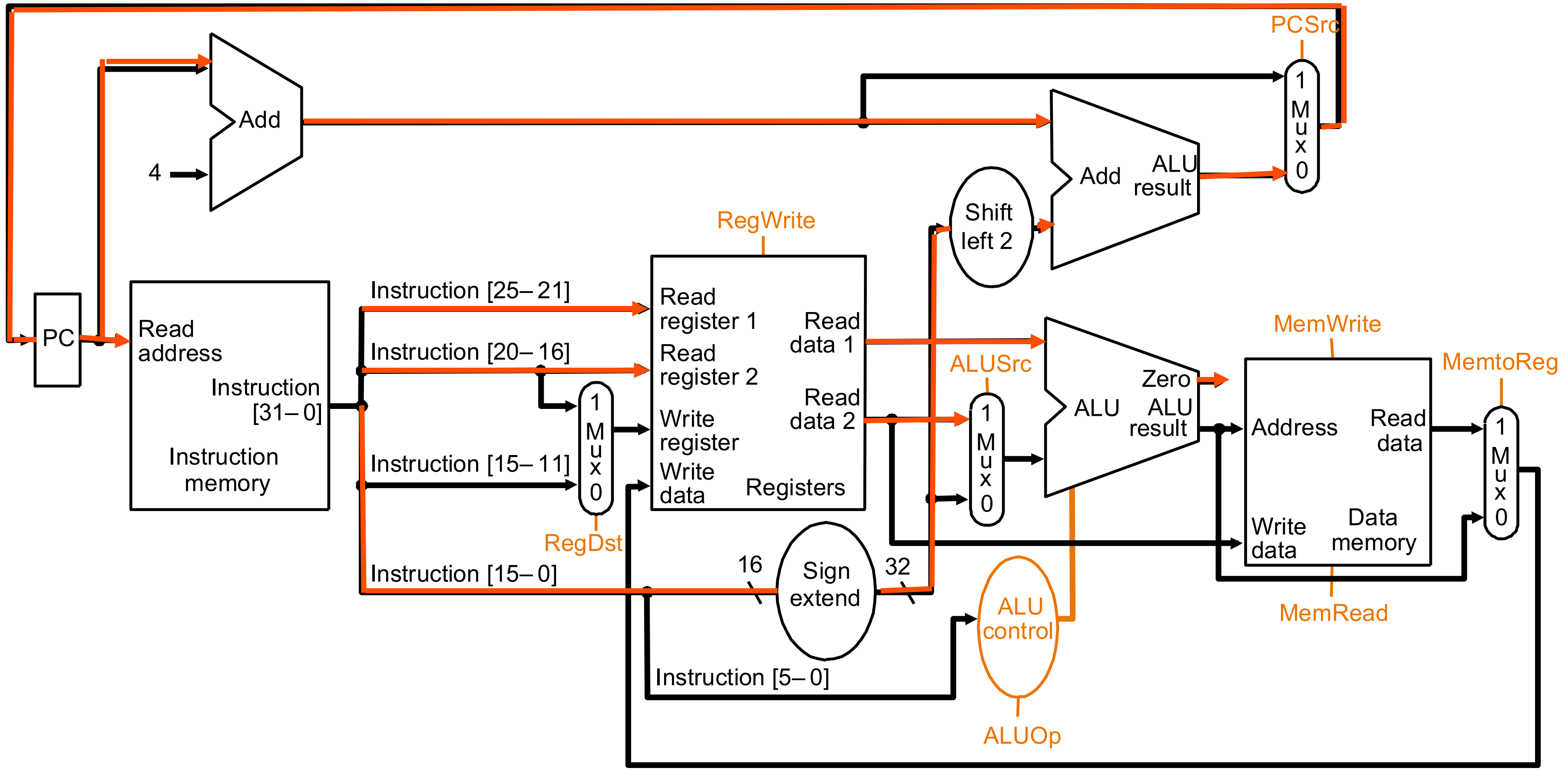
Review – R-type Instructions



Review – Lw Instruction



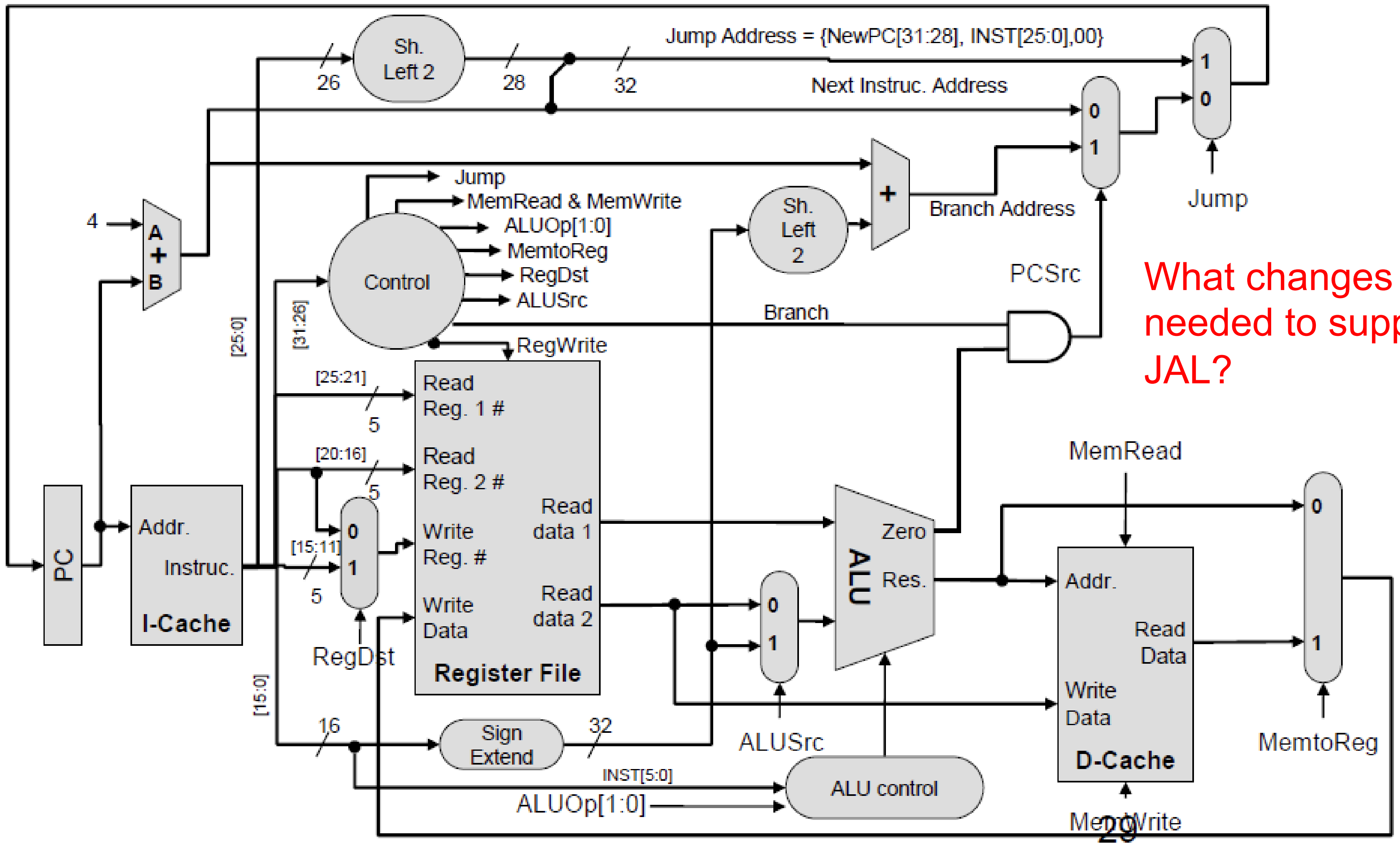
Review – Branch Instructions



Jump Instruction

- JAL (Jump and Link) for function calls
- How do we add JAL to datapath?
  - 1. Place PC+4 (return address) into $ra
  - 2. Extend RegDst mux to include 31 ($ra)
  - 3. Extend MemtoReg mux at write data input to have PC+4 as input

Jump Instruction Implementation



What changes are needed to support JAL?

Acknowledgements

- Slide sources from:
  - UCR CS161
    - Walid Najjar
    - Laxmi Bhuyan
  - USC EE457
    - Mark Redekopp
    - Gandhi Puvvada