ECE-425: Chapter-4: Processor

Prof: Mohamed El-Hadedy

Email: mealy@cpp.edu

Office: 909-869-2594

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

Instruction Execution

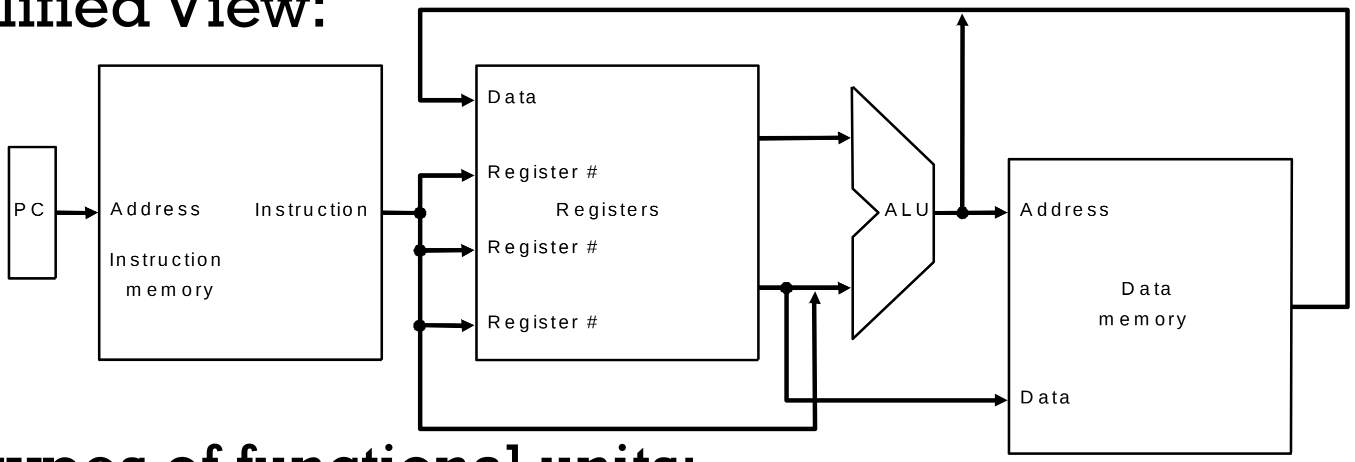
- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
    - Use ALU to calculate
        - Arithmetic result
        - Memory address for load/store
        - Branch target address
    - Access data memory for load/store
    - PC ← target address or PC + 4

The Processor: Datapath & Control

- **MIPS implementation is simplified to contain only:**
  - memory-reference instructions: lw, sw
  - arithmetic-logical instructions: add,sub,and,or,slt
  - control flow instructions: beq, j
- **Implementation:**
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do

More Implementation Details

- Simplified View:

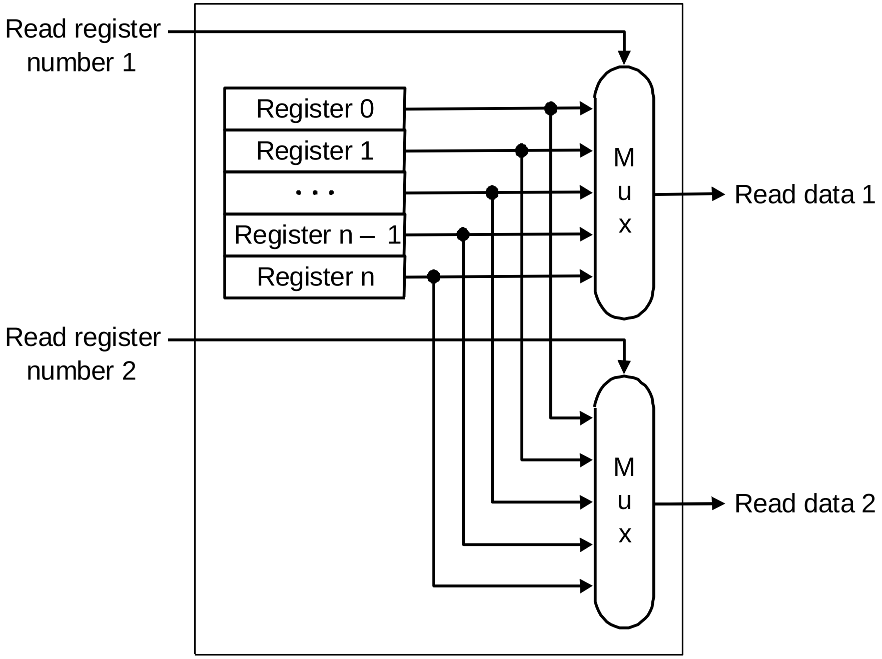


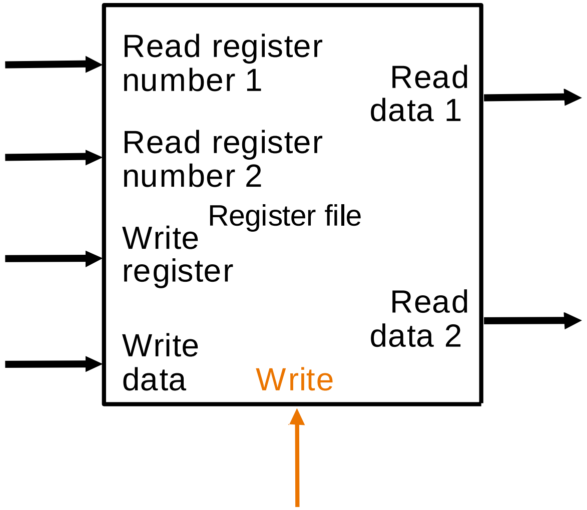
Two types of functional units:

  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

Register File

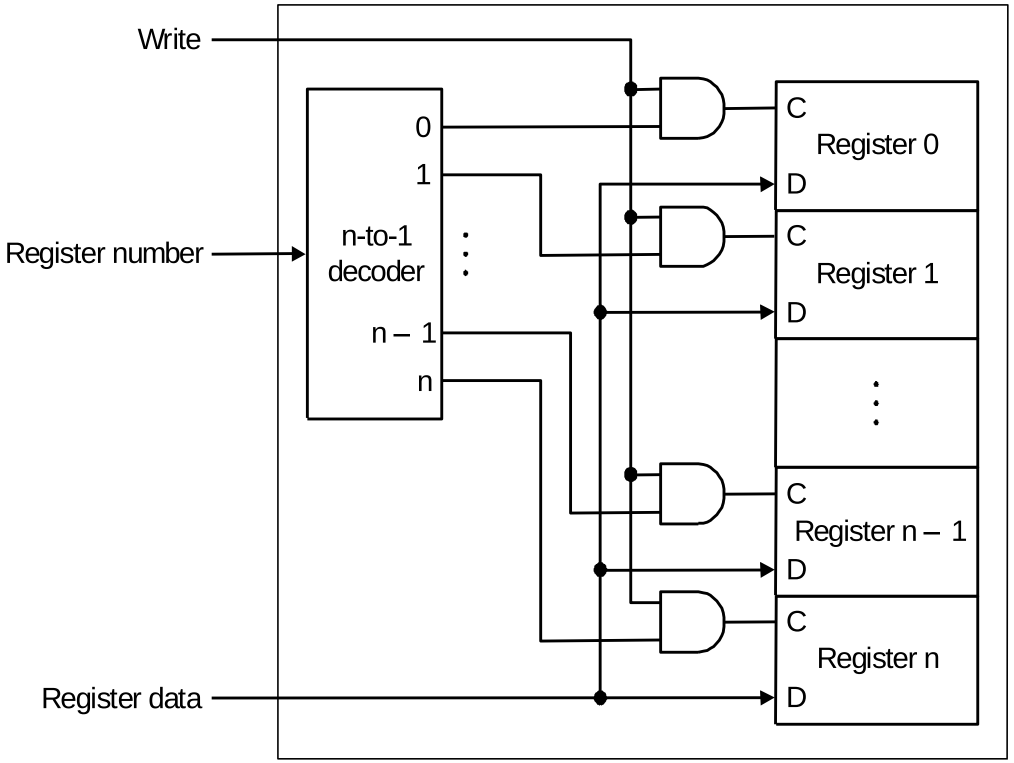
- Built using D flip-flops





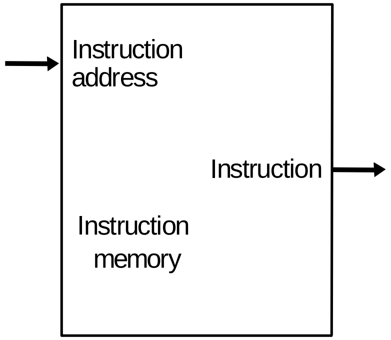
Register File

- Note: we still use the clock to determine when to write

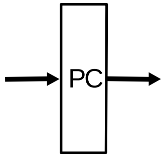


Simple Implementation

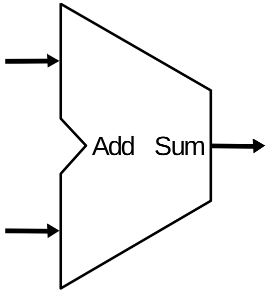
- Include the functional units we need for each instruction



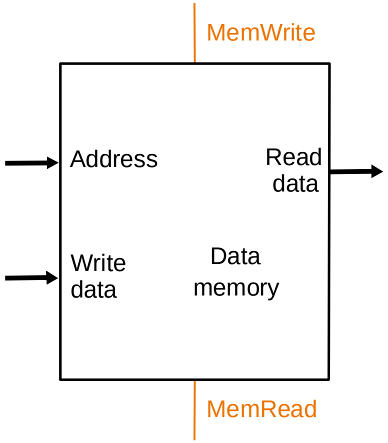
a. Instruction memory



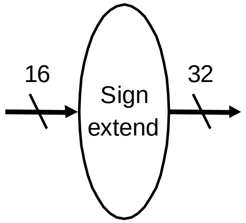
b. Program counter



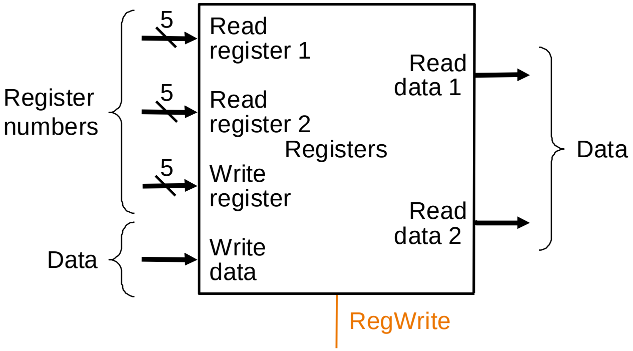
c. Adder



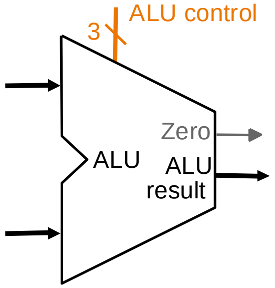
a. Data memory unit



b. Sign-extension unit



a. Registers



b. ALU

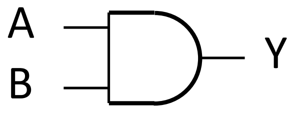
Why do we need this stuff?

Logic Design Basics

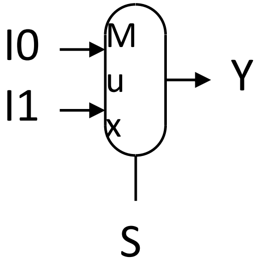
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

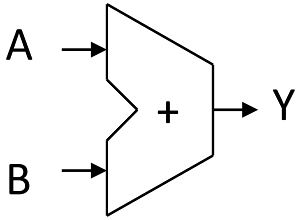
- AND-gate
  - Y = A & B



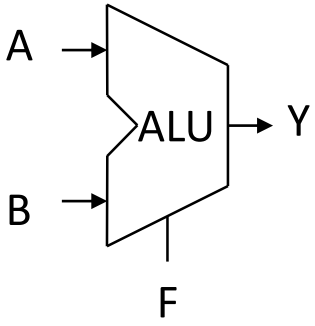
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

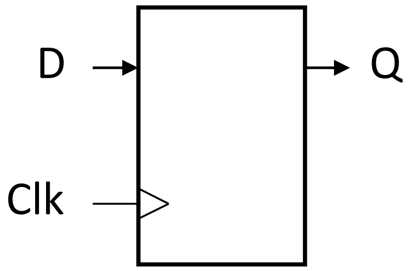


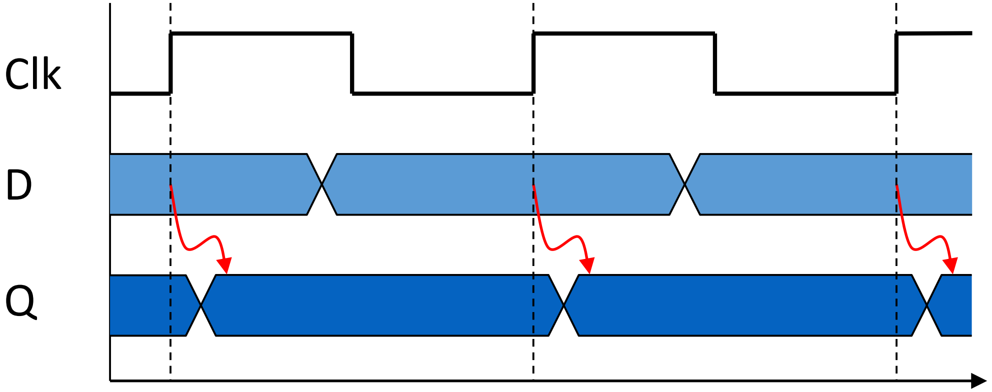
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

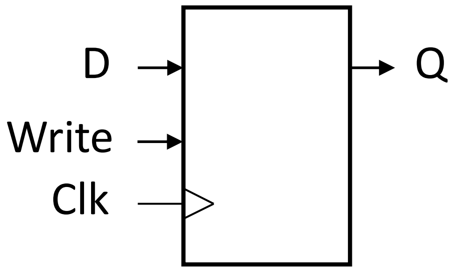
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

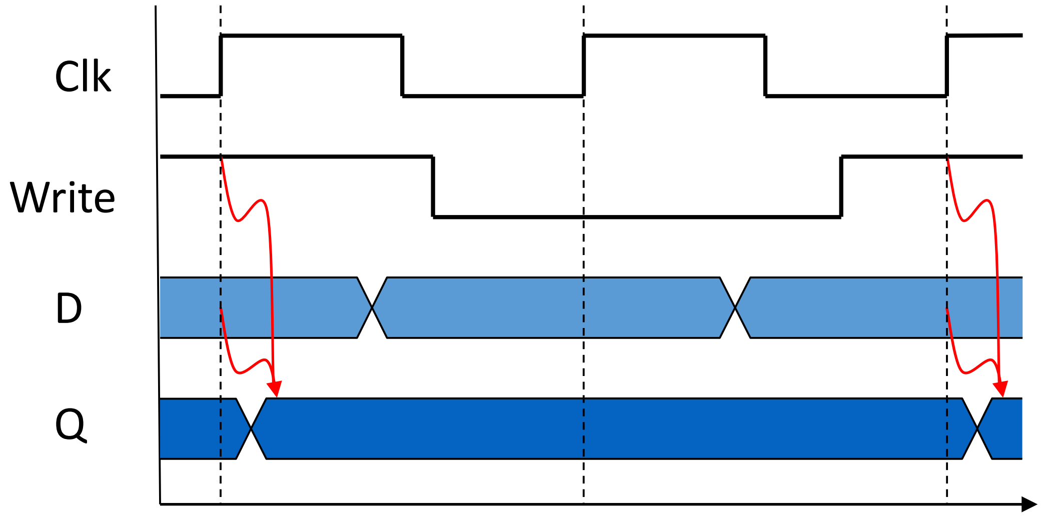




Sequential Elements

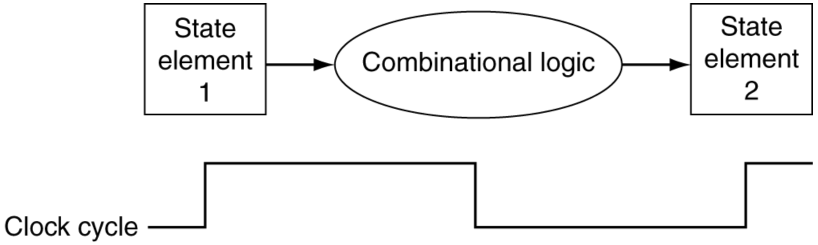
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

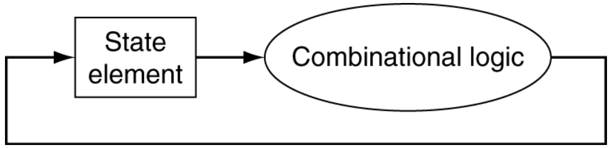




Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

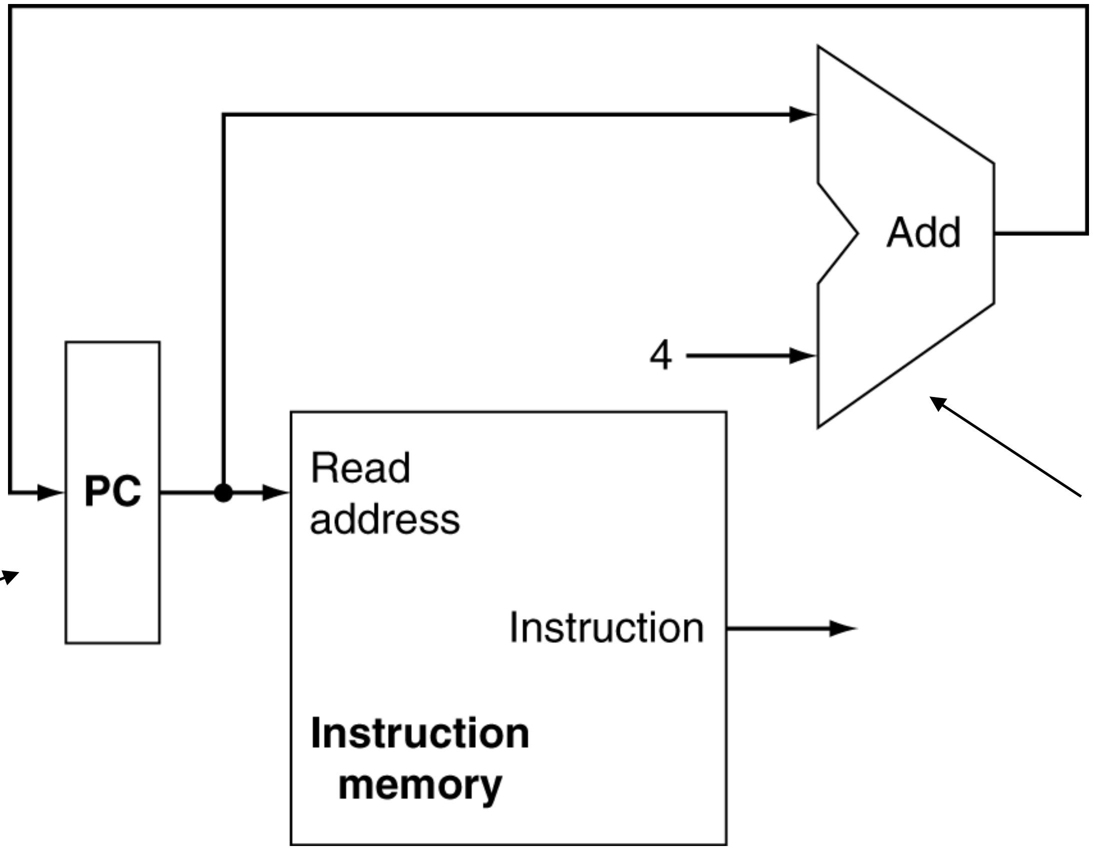




Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

Instruction Fetch

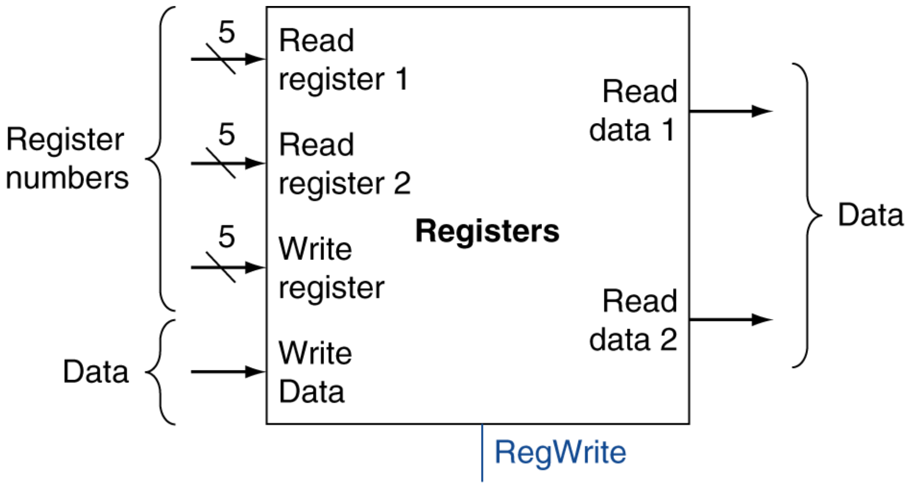


32-bit register

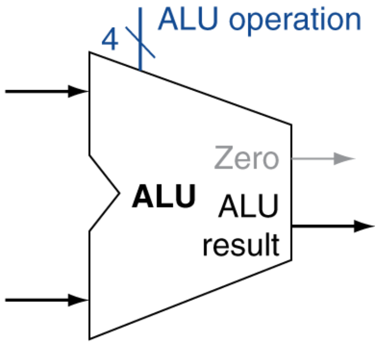
Increment by 4 for next instruction

R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



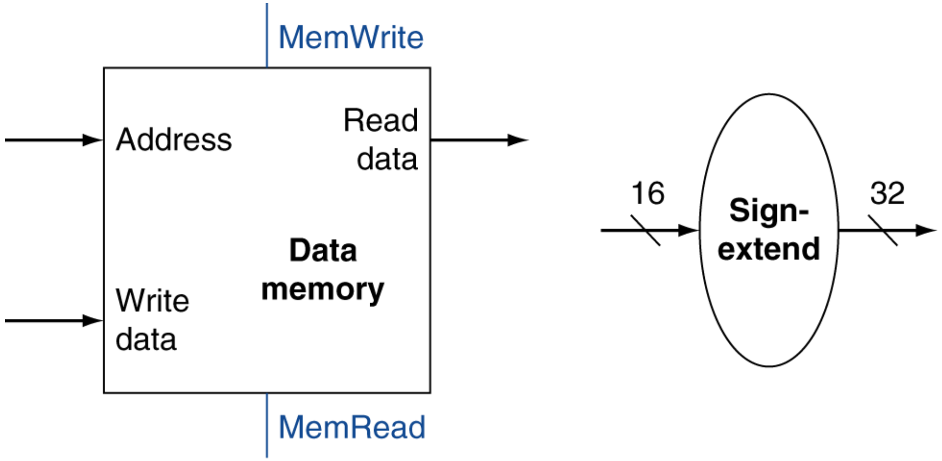
a. Registers



b. ALU

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



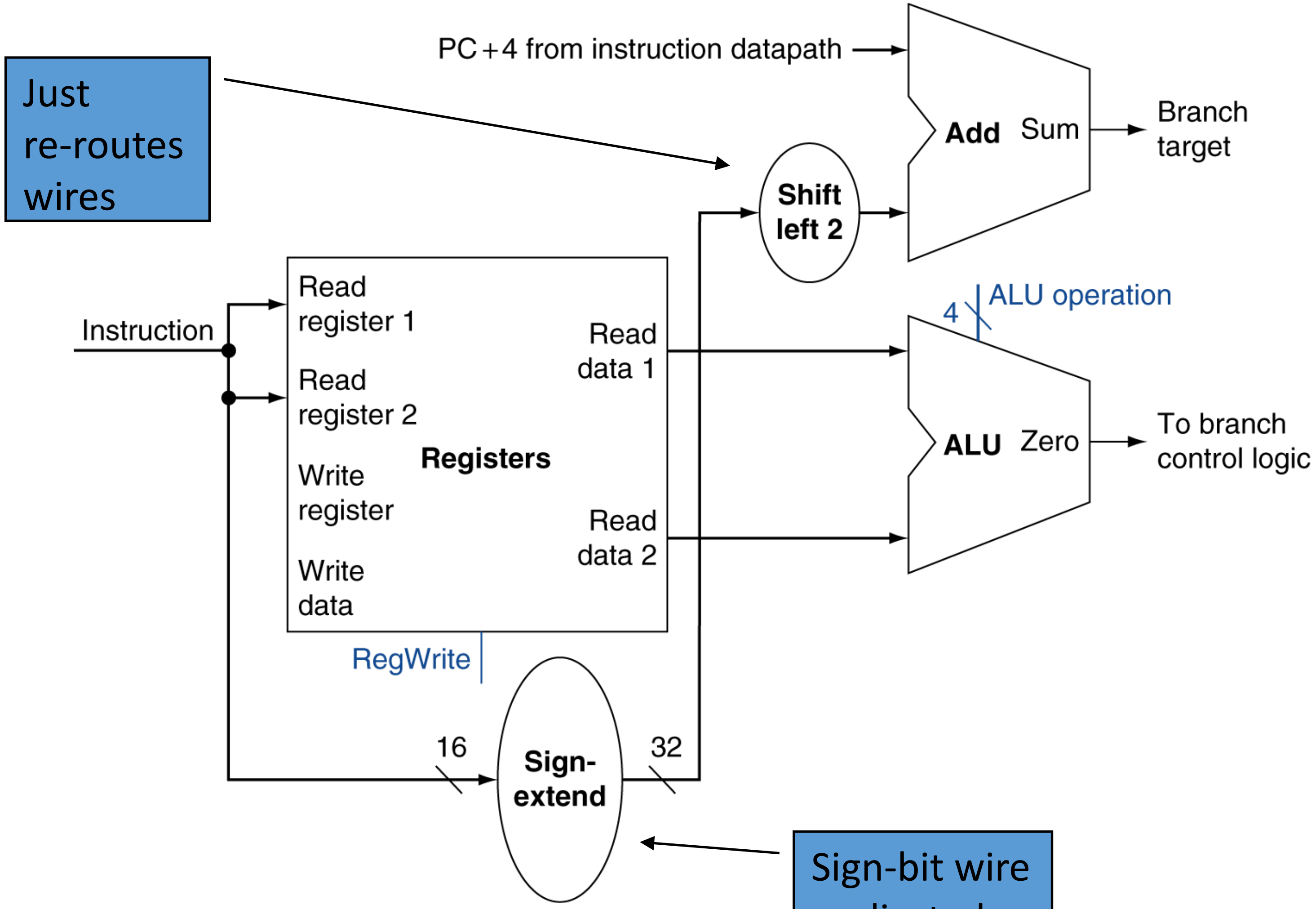
a. Data memory unit

b. Sign extension unit

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Branch Instructions



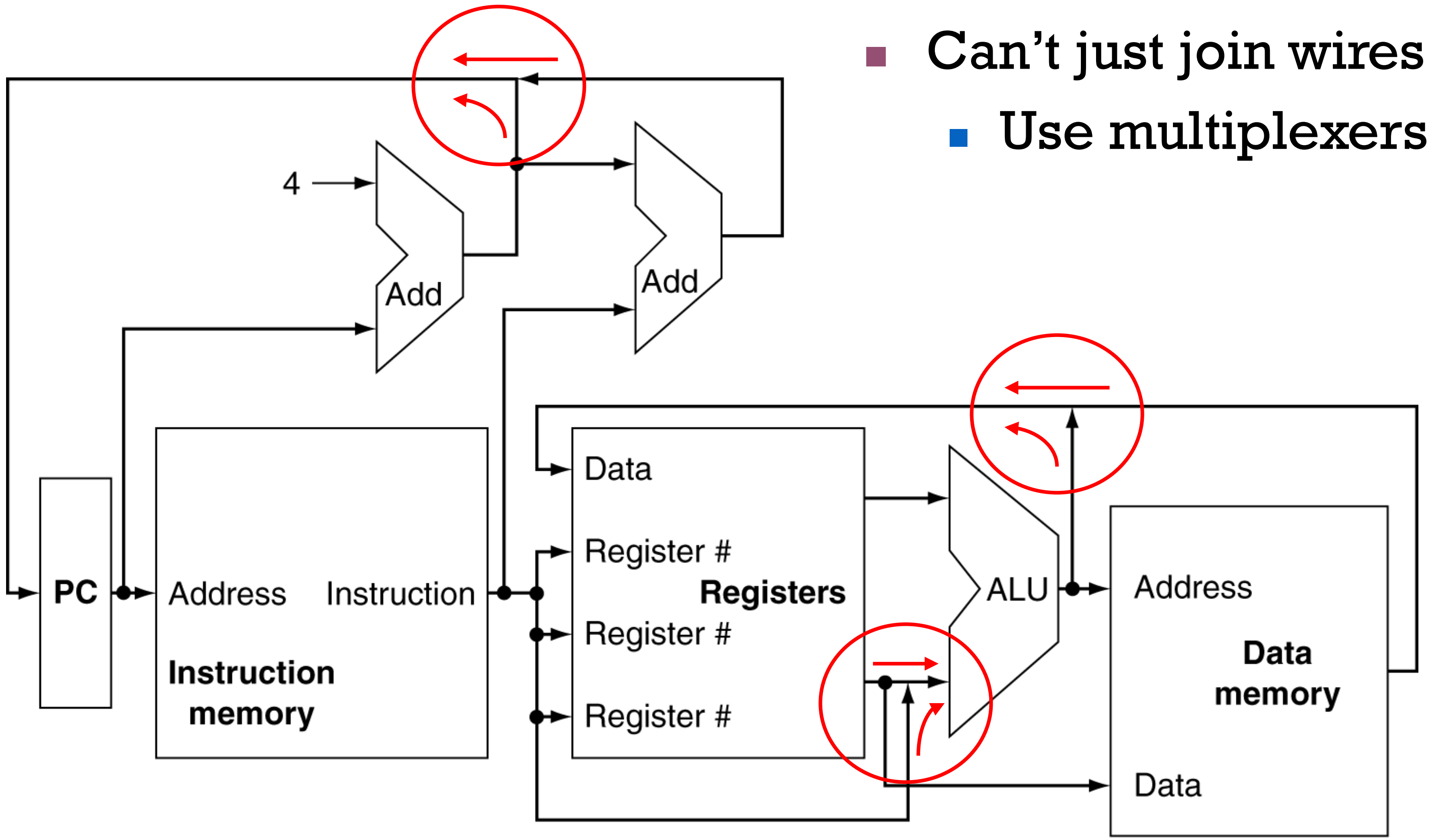
Just re-routes wires

Sign-bit wire replicated

Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

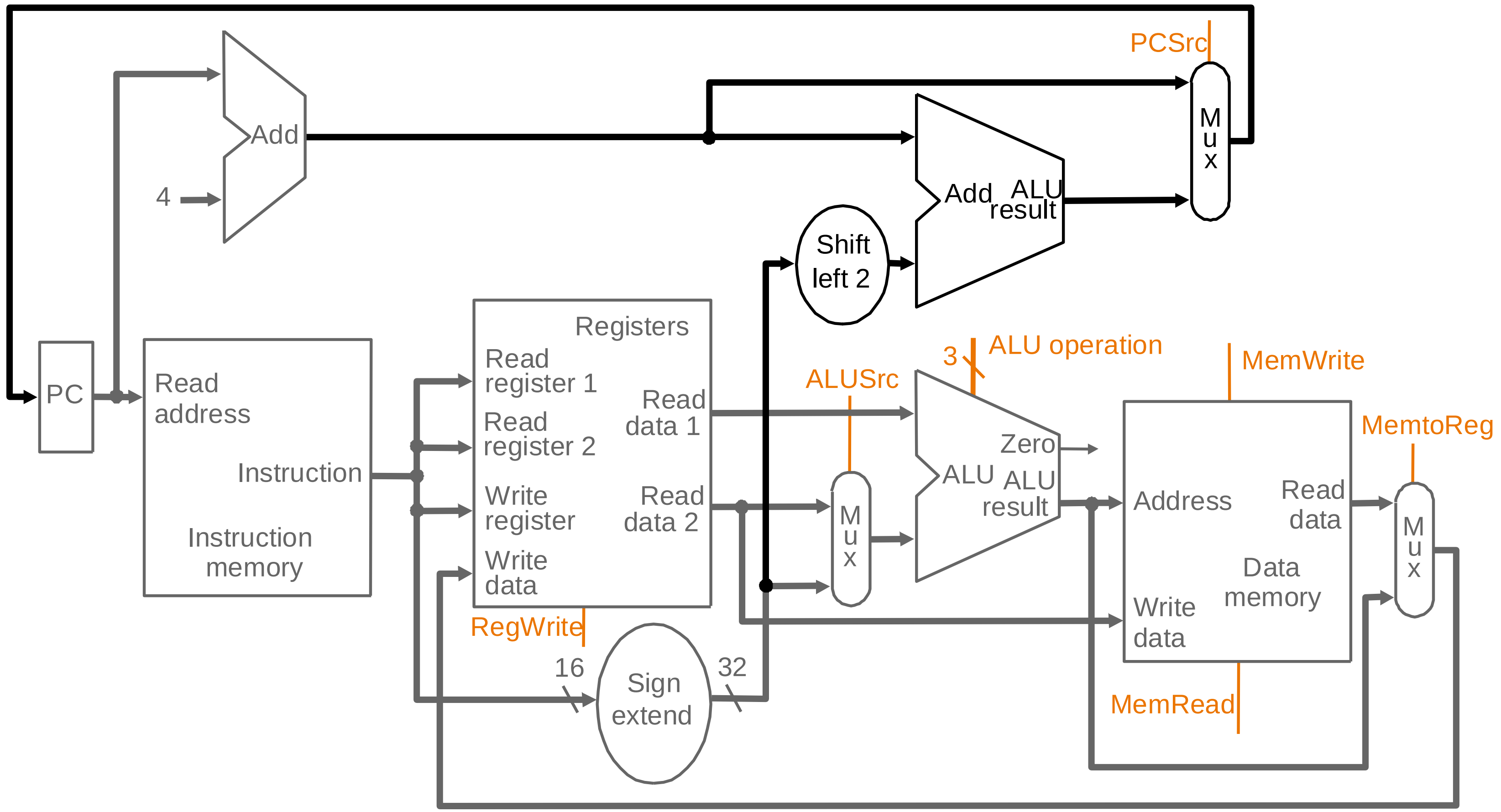
Multiplexers



- Can’t just join wires together
  - Use multiplexers

Building the Data path

- Use multiplexers to connect them together



Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexer inputs)
- Information comes from the 32 bits of the instruction
- Example:

add $8, $17, $18   Instruction Format:

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

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

Control

- What should the ALU do with this instruction
- Example: lw $1, 100($2)

35	2	1	100
op	rs	rt	16 bit offset

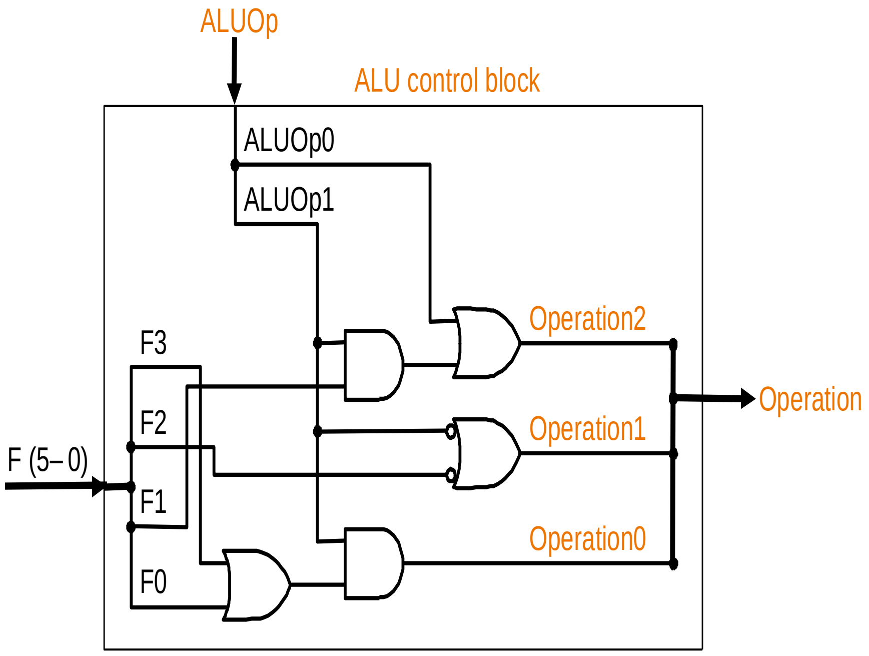
- ALU control input

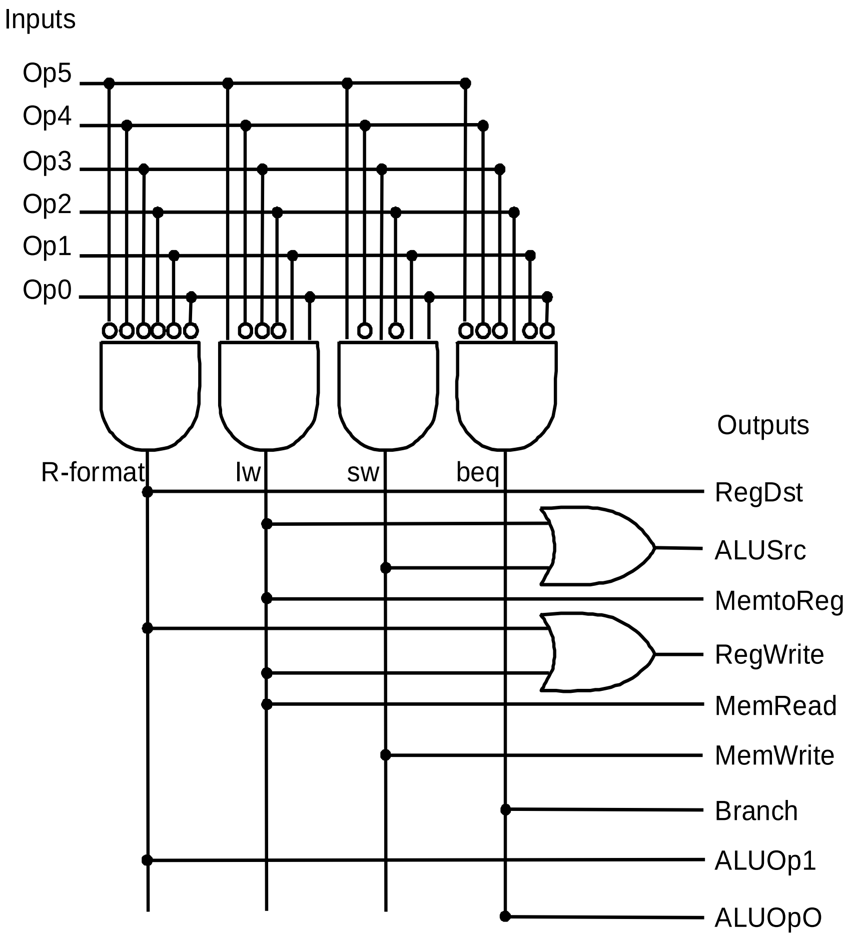
000	AND
001	OR
010	add
110	subtract
111	set-on-less-than

For load word and store word instructions, we use the ALU to compute the memory address by addition.

Control

- Simple combinational logic (truth tables)





Control

- Must describe hardware to compute 3-bit ALU control input
  - given instruction type

    00 = lw, sw

    01 = beq,

    11 = arithmetic

    ALUOp
    computed from instruction type

  - function code for arithmetic
- Describe it using a truth table (can turn into gates):

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

For branch equal, the ALU must perform as a subtraction

Generate the 4-bit ALU control input using a small control unit that has an inputs the function field of the instruction and a 2-bit control field, which we call ALUOp. ALUOp indicates whether the operation to be performed should be add(00) for loads and stores, subtract(01) for beq, or determined by the operation encoded in the funct field (10). The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations.

ALU Control

- For the R-type instructions, the ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than), depending on the value of the 6-bit funct (or function) field in the low-order bits of the instruction.
- **R Format: (Check slide no.20, Lecture 3, Chaper-2)**

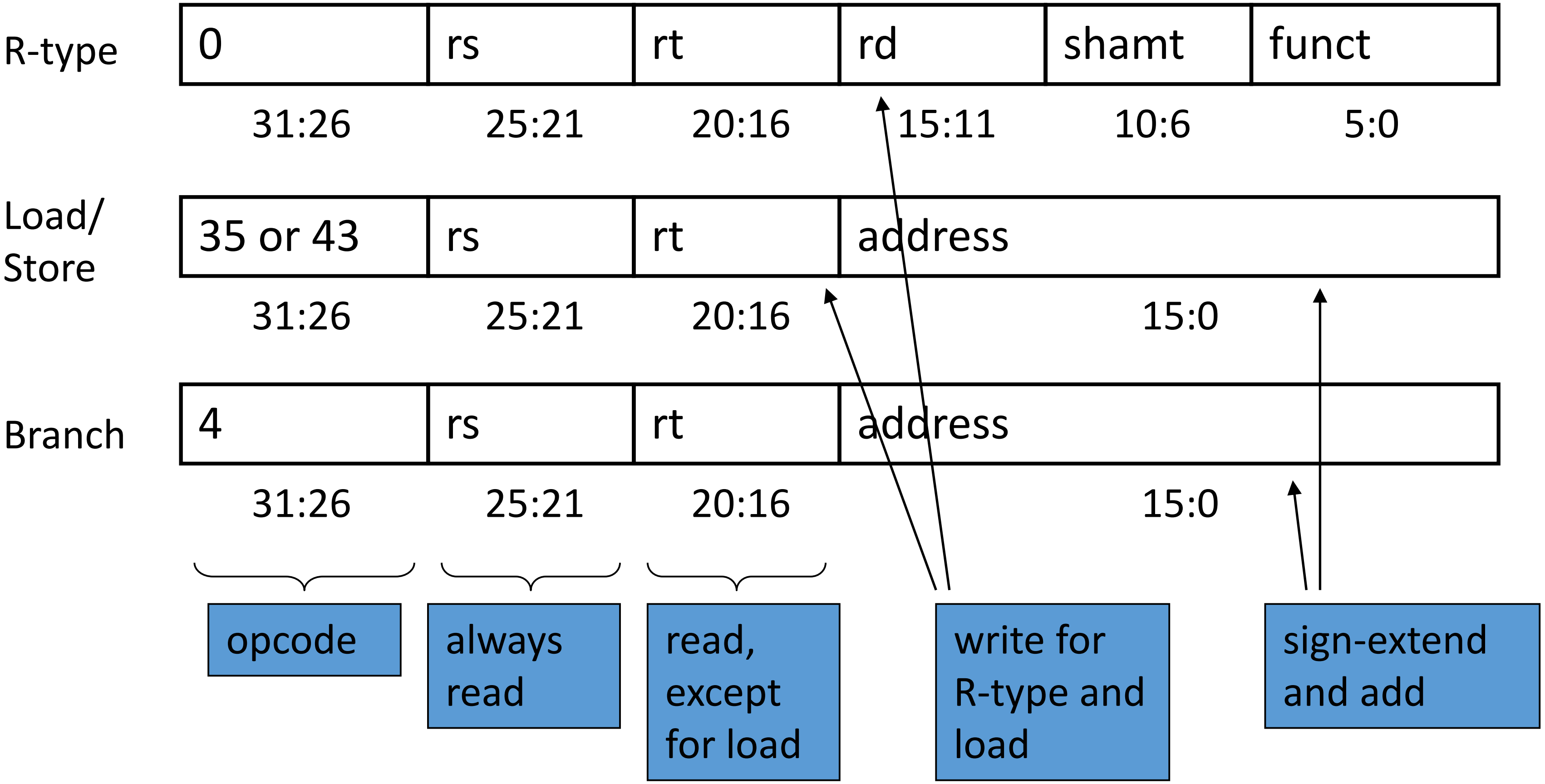
opcode	RS	RT	RD	Shift (shmat)	Funct
6-bits	5-bits	5-bits	5-bits	5-bits	6- bits

Shift(shmat): used with the shift and rotate instructions, this is the amount by which the source operand RS is rotated/shifted. This field is 5-bits long (6 to 10).

Funct: For the instructions that share an opcode, the funct parameter contains the necessary control codes to differentiate the different instructions. Example: Opcode 0x00 access the ALU, and the funct selects which ALU function to use (lets say 0x20 refers to adding specifically)

The Main Control Unit

- Control signals derived from instruction



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	address		
	31:26	25:21	20:16	15:0		
Branch	4	rs	rt	address		
	31:26	25:21	20:16	15:0		

opcode

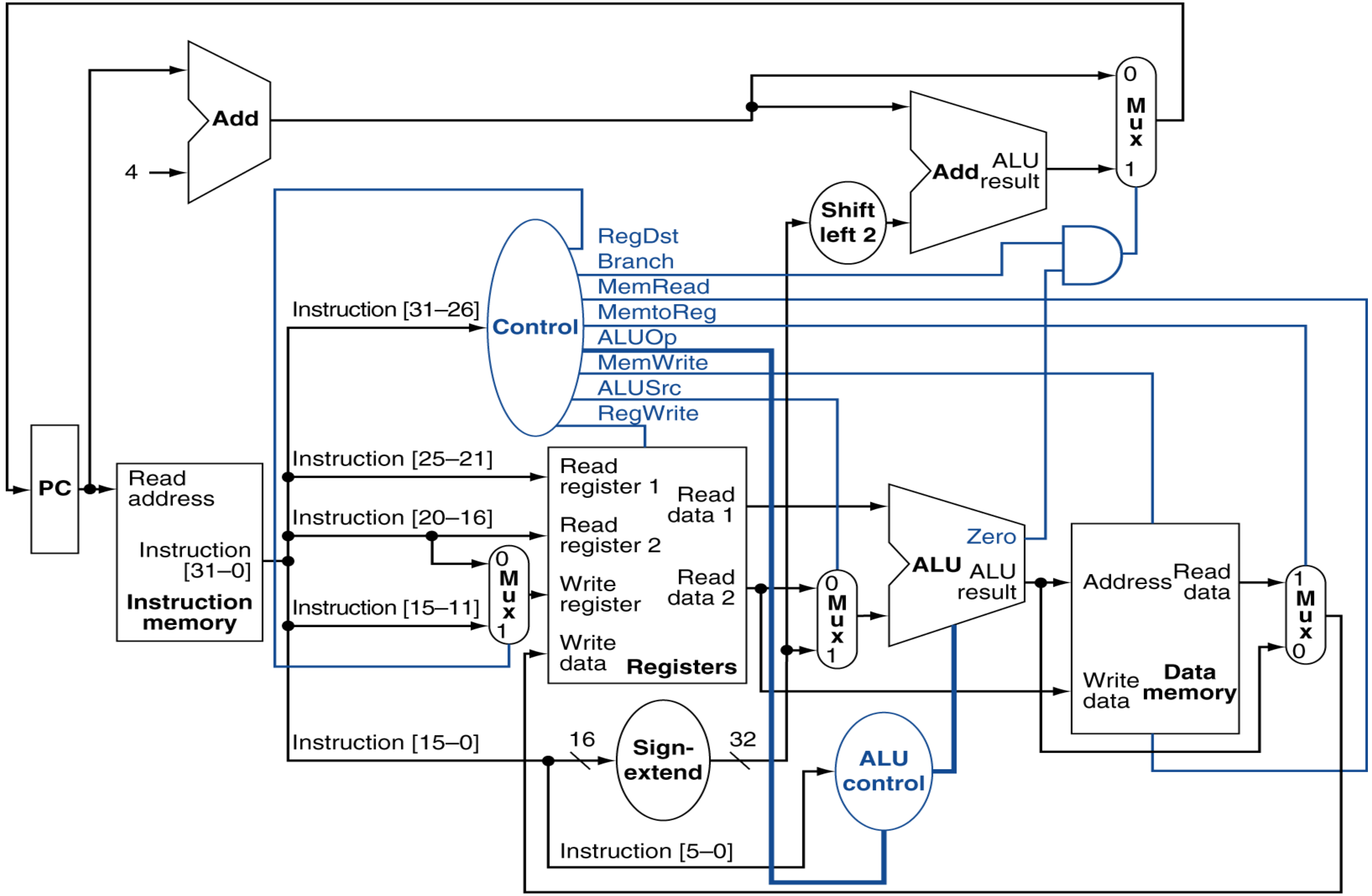
always read

read, except for load

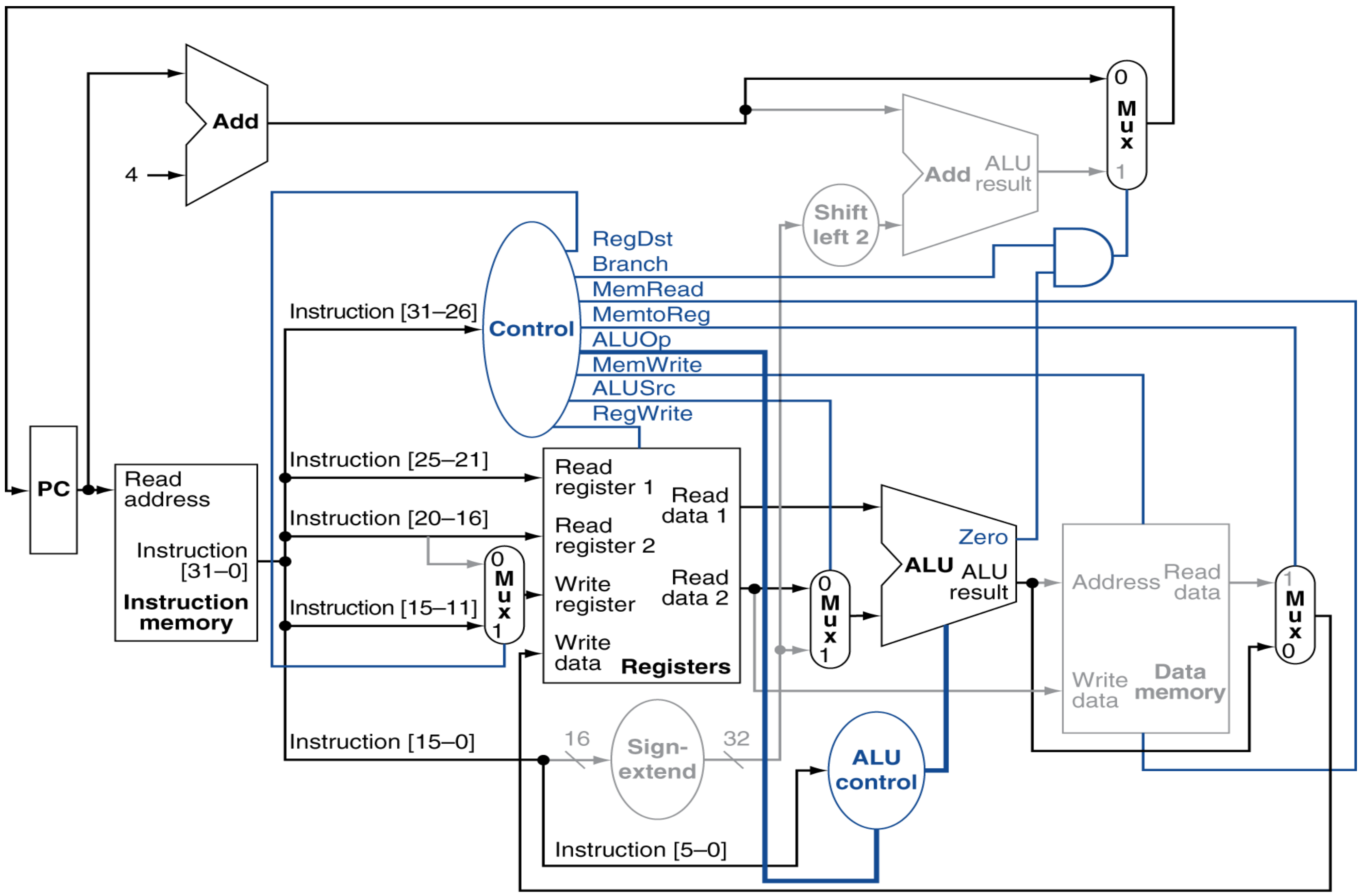
write for R-type and load

sign-extend and add

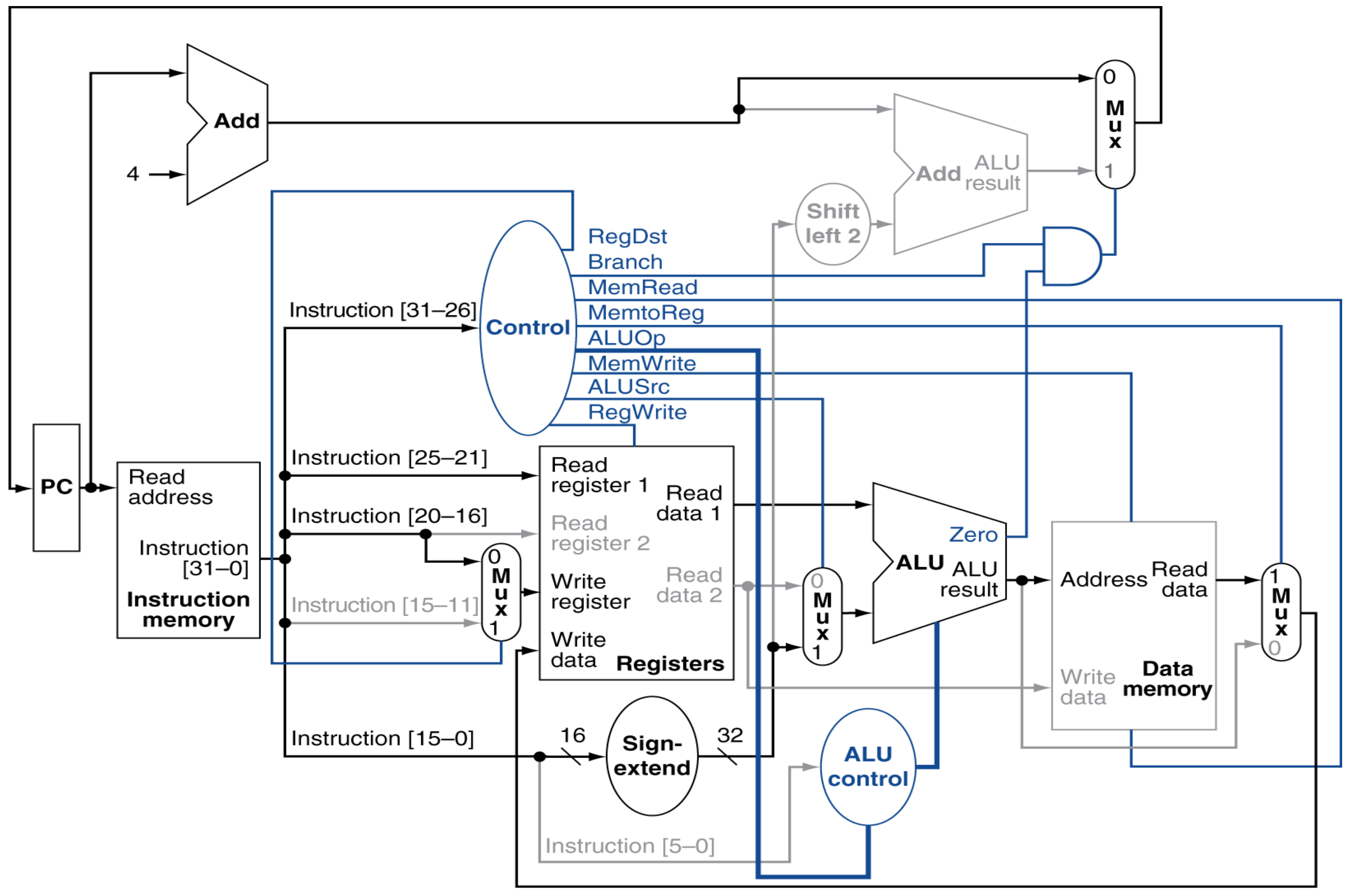
Datapath With Control



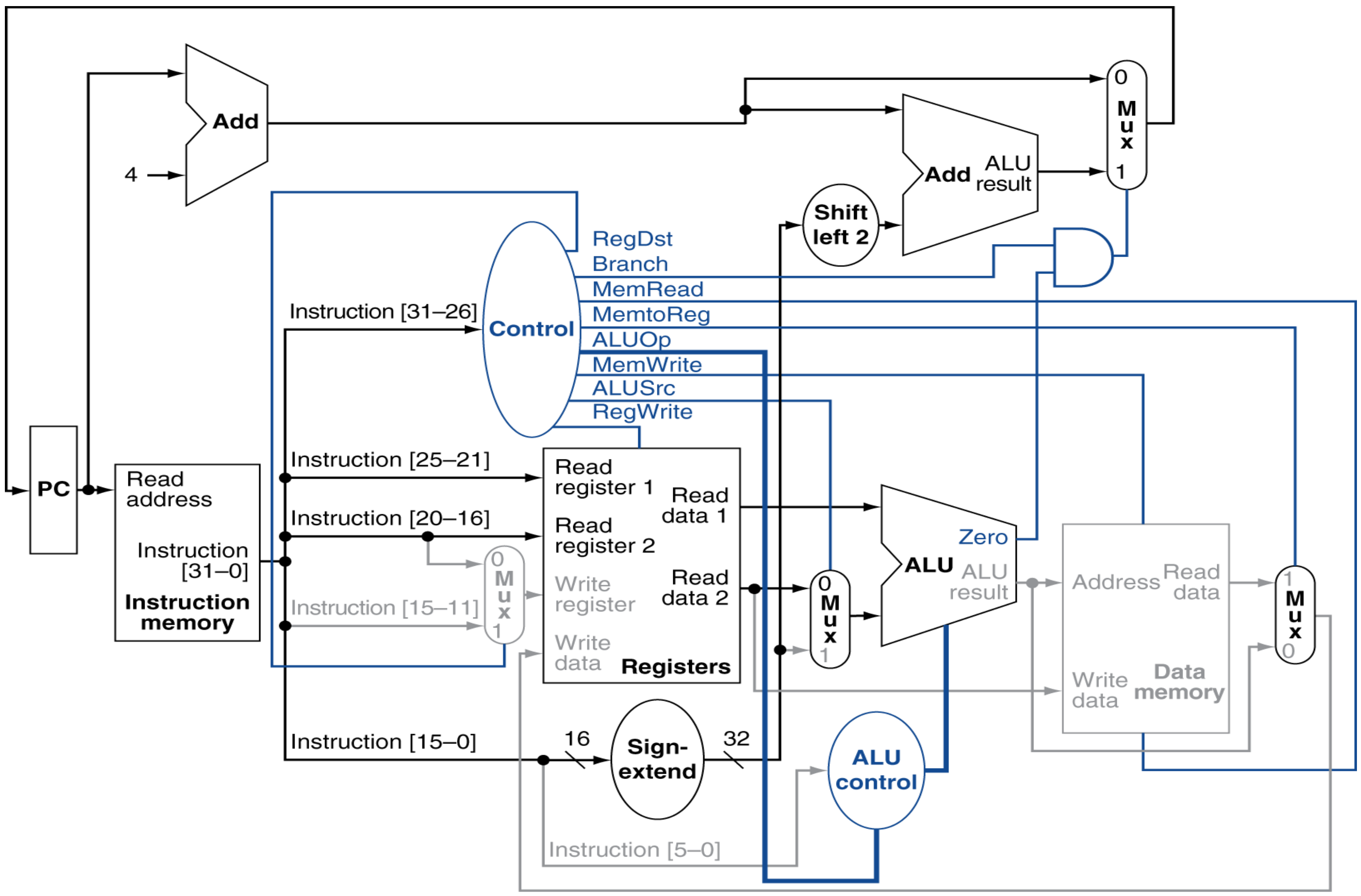
R-Type Instruction



Load Instruction



Branch-on-Equal Instruction

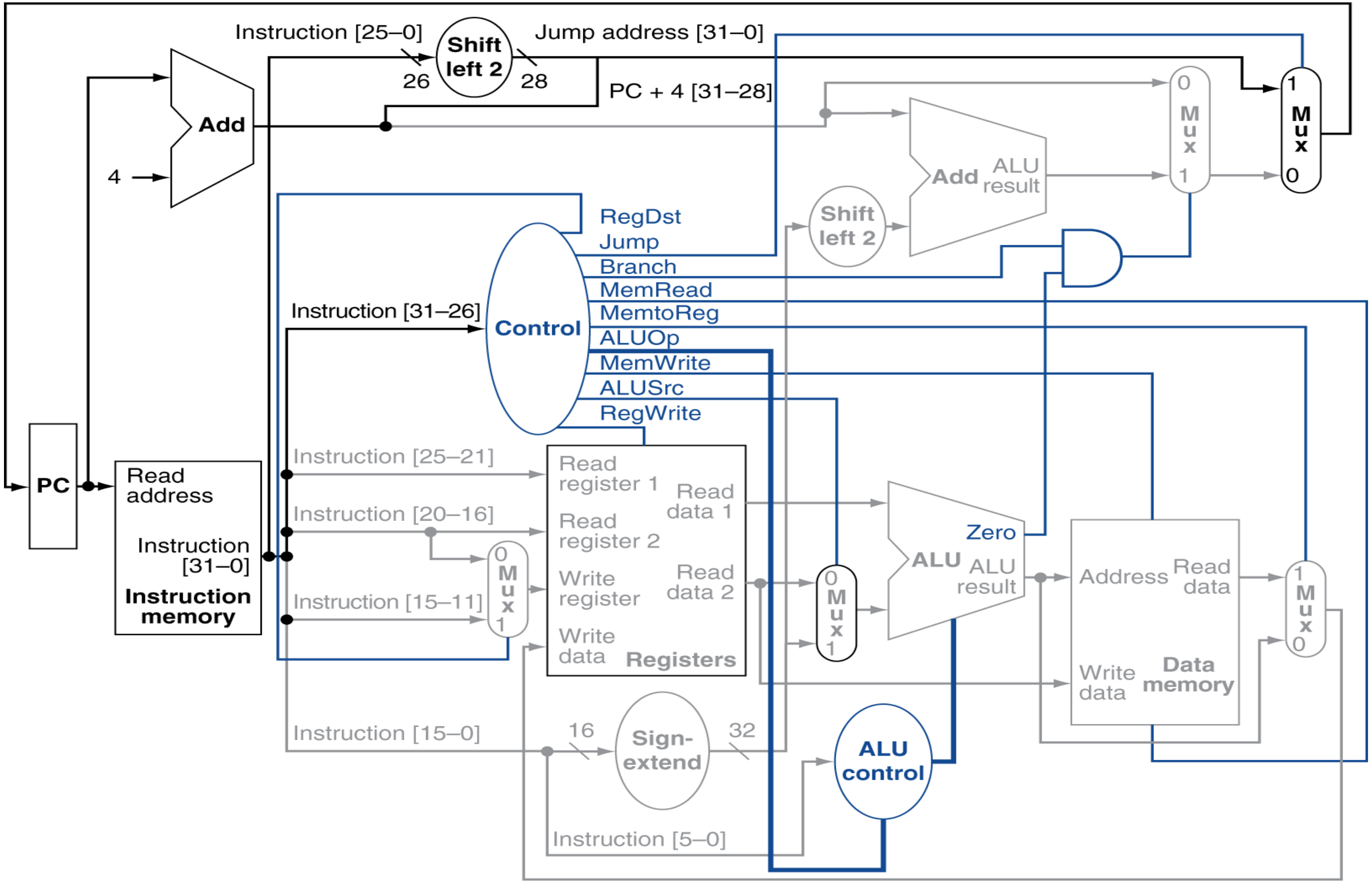


Implementing Jumps

Jump		address
	31:26	25:0

- 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

Datapath With Jumps Added

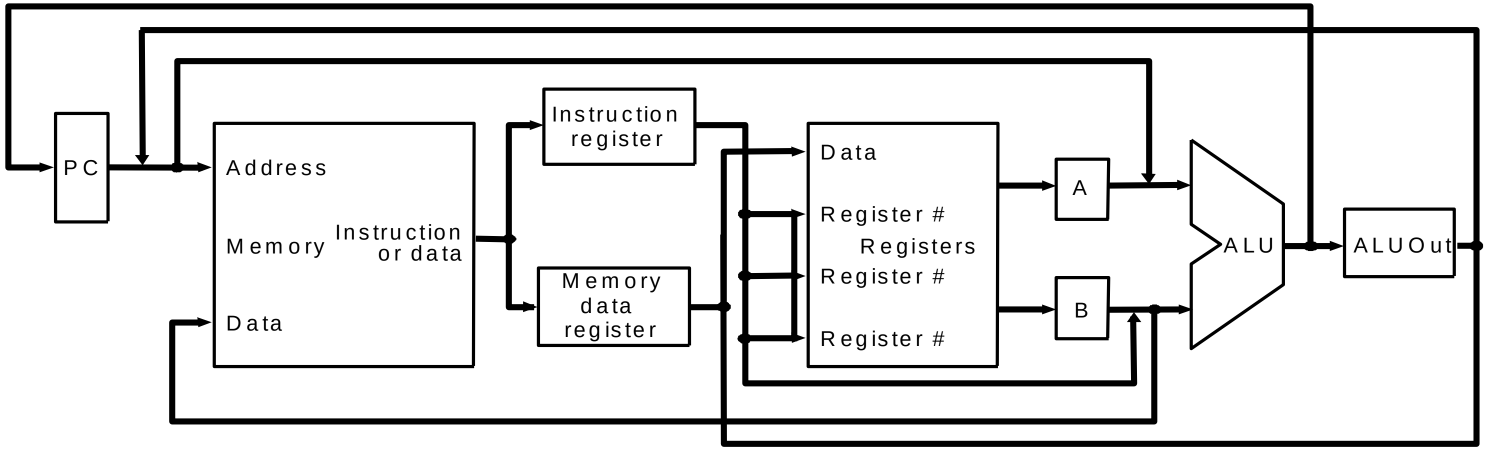


Performance Issues

- 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

Where we are headed

- Single Cycle Problems:
  - what if we had a more complicated instruction such as floating point?
- One Solution:
  - use a "smaller" cycle time
  - have different instructions take different numbers of cycles
  - a "multicycle" datapath:

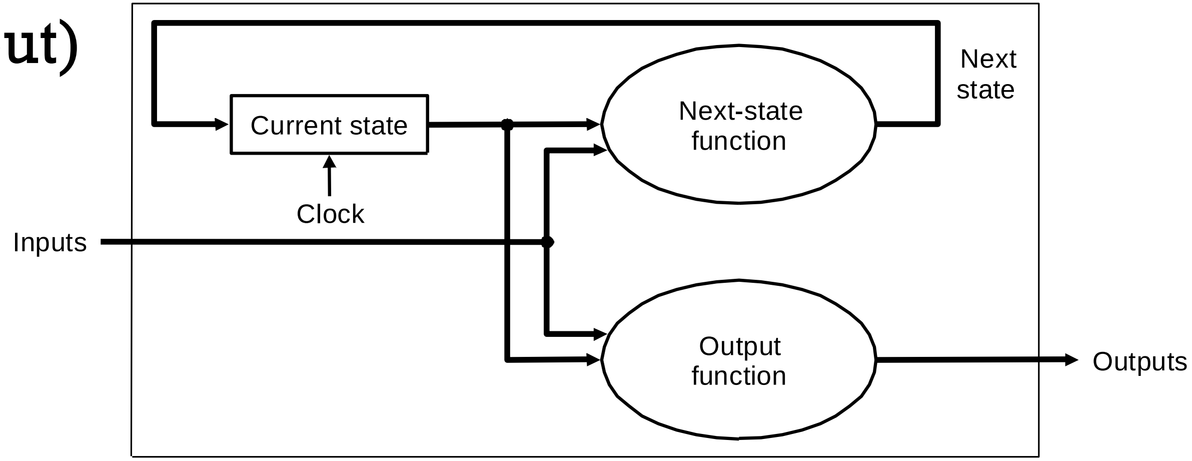


Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control

Review: finite state machines

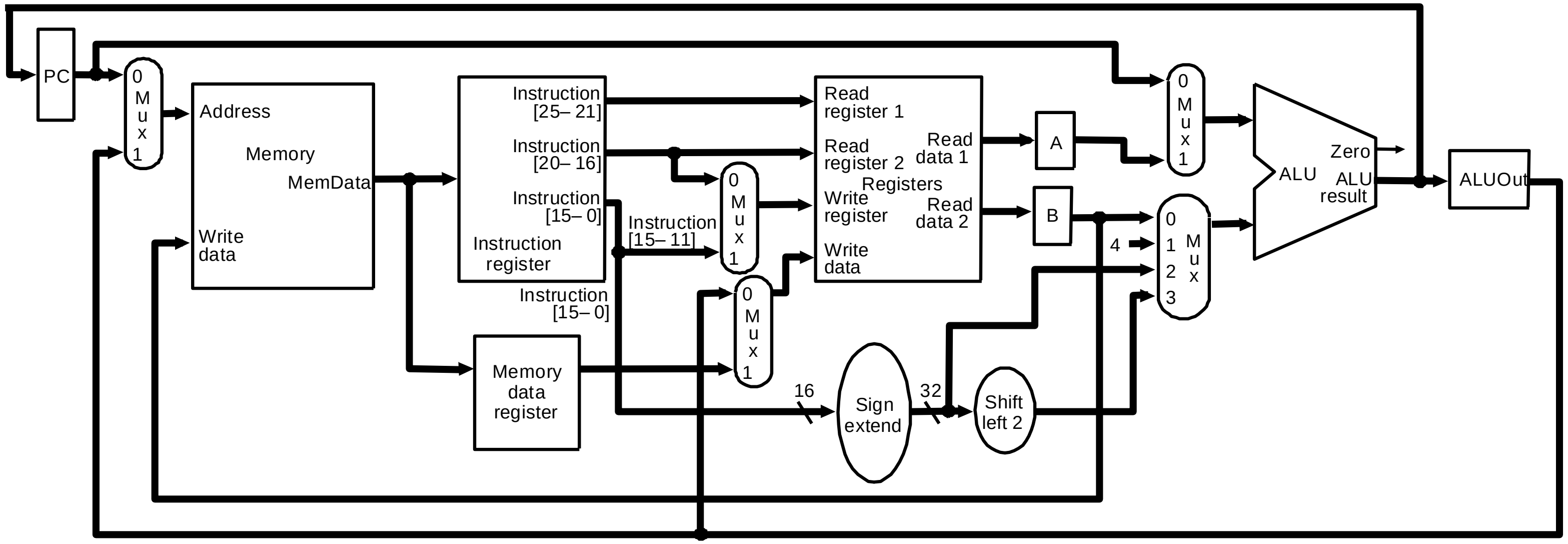
- Finite state machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)



  - We'll use a Moore machine (output based only on current state)

Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional "internal" registers



Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR = Memory[PC];
PC = PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut= PC +(sign-extend(IR[15-0])<< 2);
```

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:

  ALUOut = A + sign-extend(IR[15-0]);

- R-type:

  ALUOut = A op B;

- Branch:

  if (A==B) PC = ALUOut;

Step 4 (R-type or memory-access)

- Loads and stores access memory

```
MDR = Memory[ALUOut];
        or
Memory[ALUOut] = B;
```

- R-type instructions finish

```
Reg[IR[15-11]] = ALUOut;
```

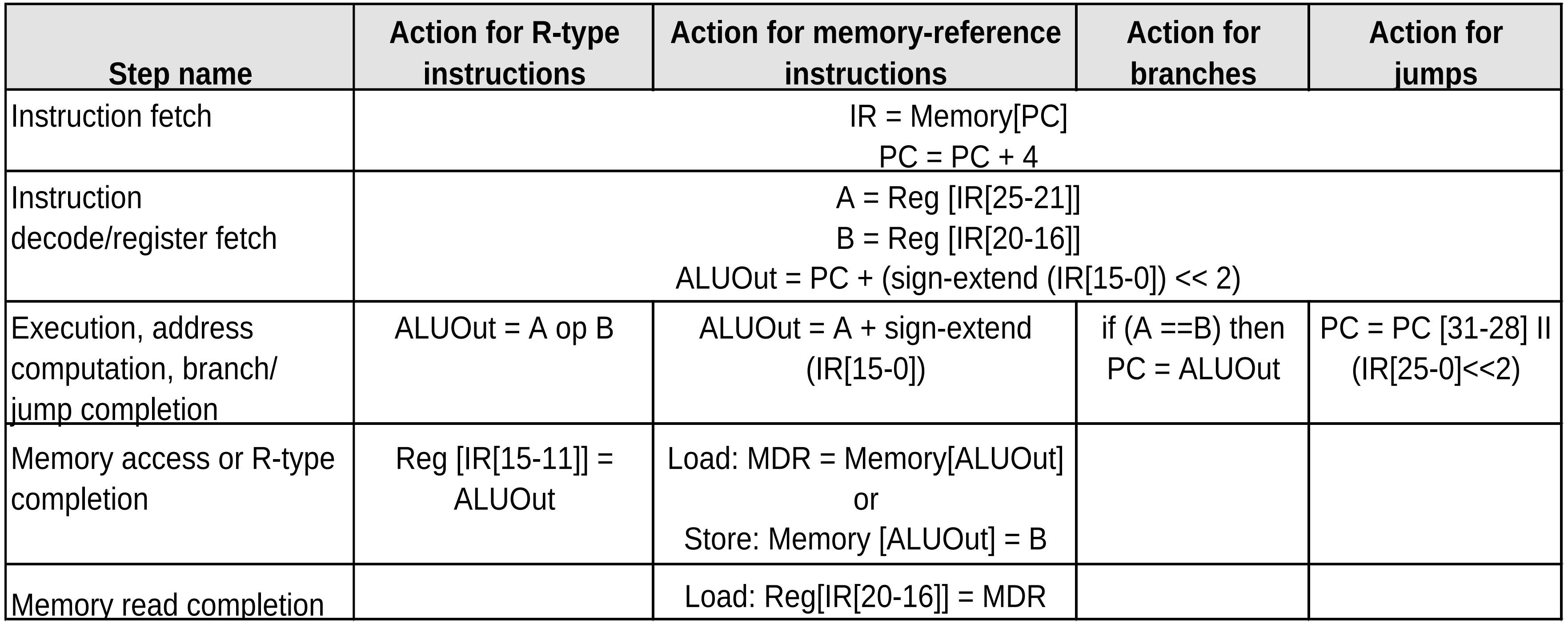
The write actually takes place at the end of the cycle on the edge

Write-back step

- Reg[IR[20-16]]= MDR;

What about all the other instructions?

Summary



Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps

Instruction fetch	IR = Memory[PC]
	PC = PC + 4			

Instruction decode/register fetch	A = Reg [IR[25-21]]
	B = Reg [IR[20-16]]
	ALUOut = PC + (sign-extend (IR[15-0]) << 2)			

Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Simple Questions

- How many cycles will it take to execute this code?

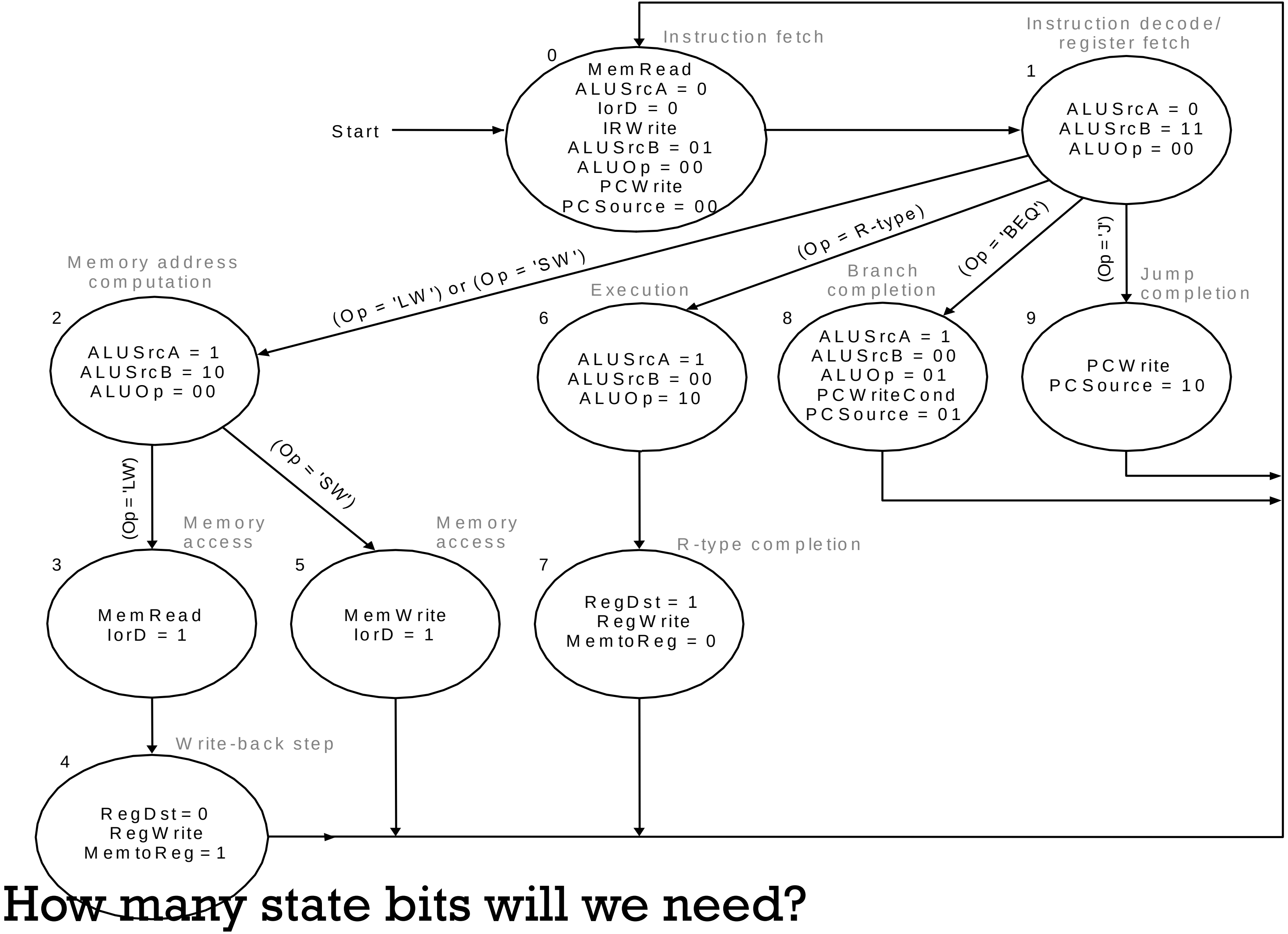
```
            lw $t2, 0($t3)
            lw $t3, 4($t3)
            beq $t2, $t3, Label     #assume not
            add $t5, $t2, $t3
            sw $t5, 8($t3)
Label:      ...
```

- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of $t2 and $t3 takes place?

Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

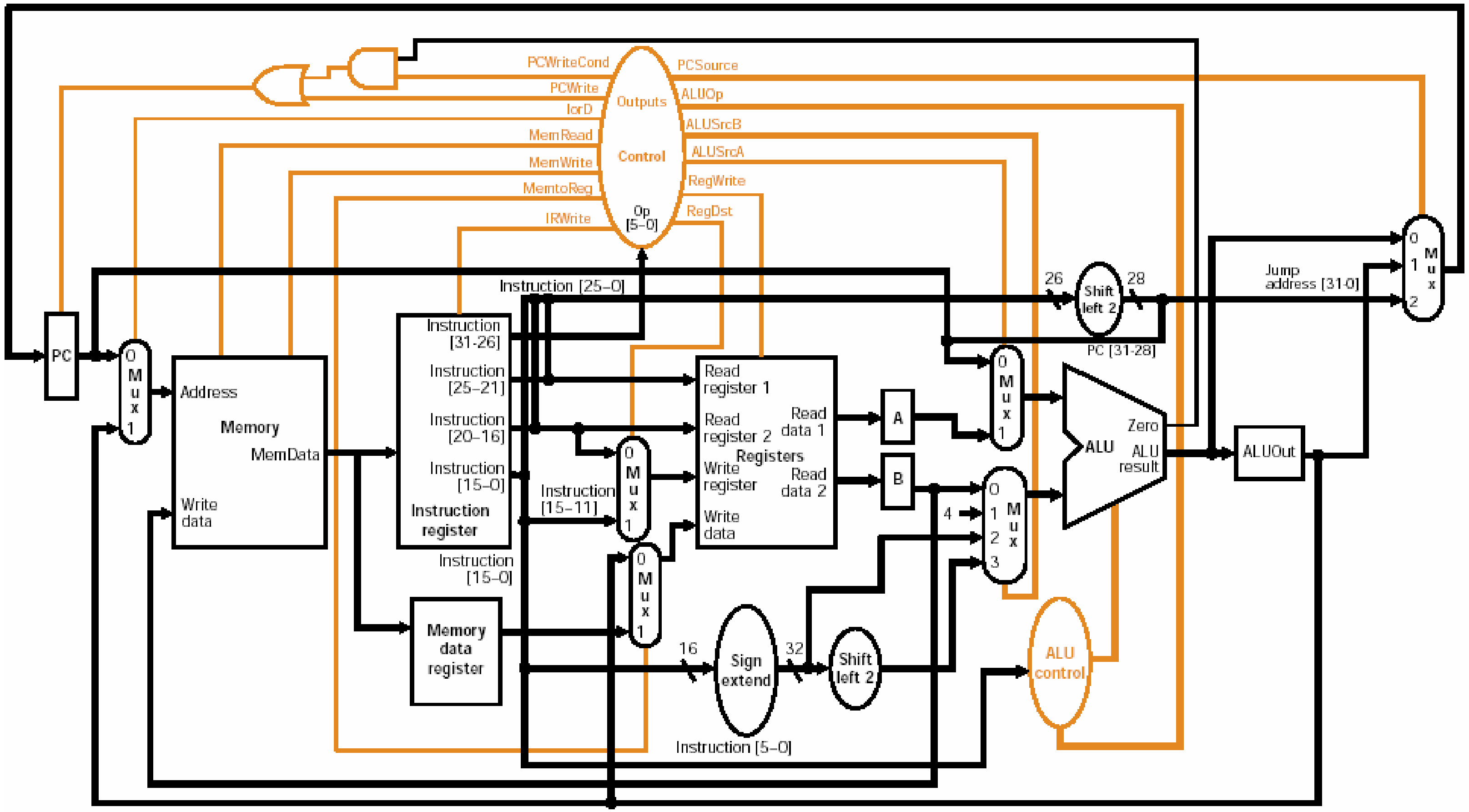
Graphical Specification of FSM



- How many state bits will we need?

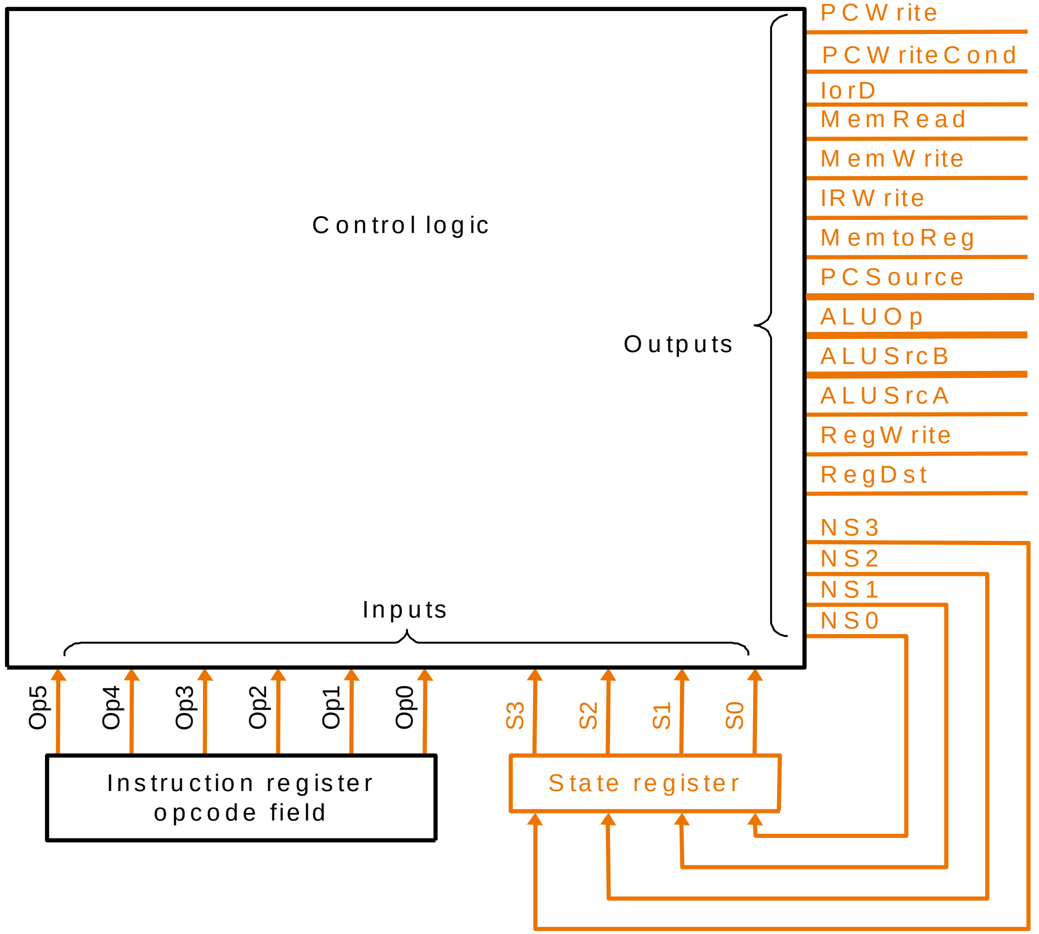
Multiple Cycle Datapath

- Miminizes Hardware: 1 memory, 1 adder



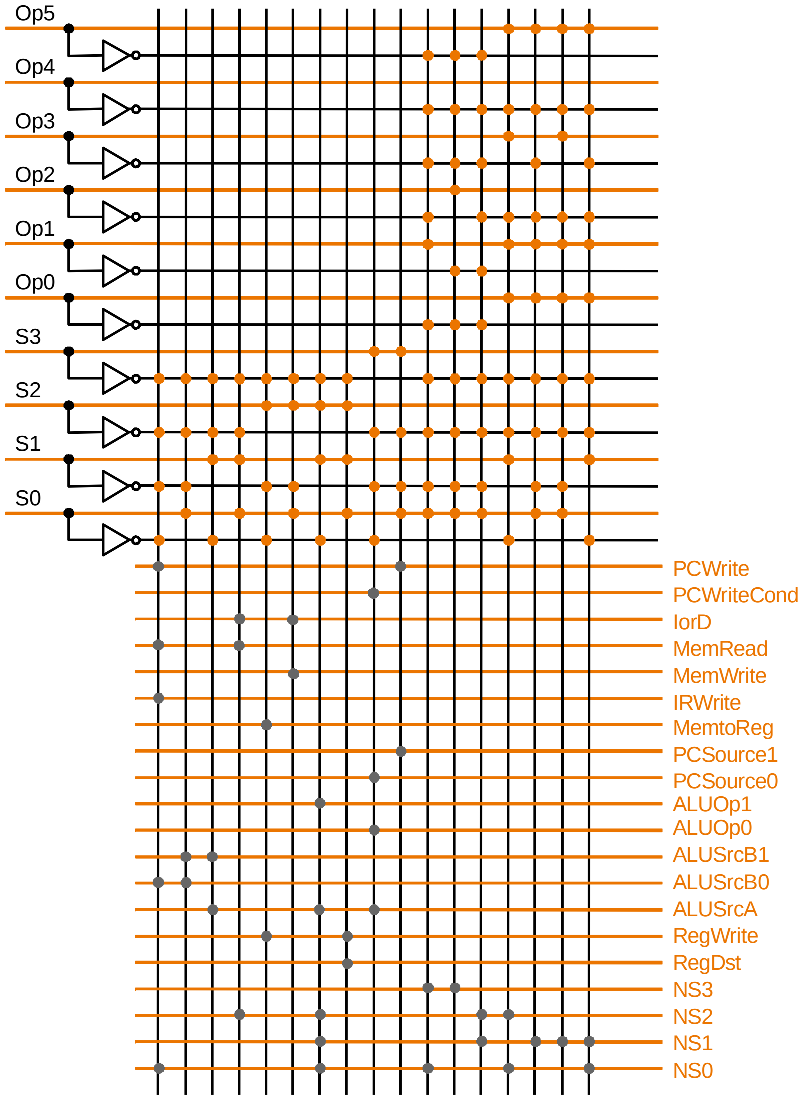
Finite State Machine for Control

- Implementation:



PLA Implementation

- If I picked a horizontal or vertical line could you explain it?



AND Gates

OR Gates

Summary

- Disadvantages of the Single Cycle Processor
  - Long cycle time
  - Cycle time is too long for all instructions except the Load
- Multiple Cycle Processor:
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
- Partition datapath into equal size chunks to minimize cycle time
  - ~10 levels of logic between latches
- Follow same 5-step method for designing "real" processor

Summery

- Control is specified by finite state diagram
- Control is more complicated with:
  - complex instruction sets
  - restricted datapaths
- Simple Instruction set and powerful datapath => simple control
  - could try to reduce hardware
  - rather go for speed => many instructions at once!

Reference:

- **Book:** D. A. Patterson and J. L. Hennessy, **Computer Organization and Design: The Hardware/ Software Interface**, 5th Edition, San Mateo, CA: Morgan and Kaufmann. ISBN: 1-55860-604-1
- https://www.mips.com/
- https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html
- Professor El-Naga ECE-425 notes