

CSE 141 – Computer Architecture

Fall 2005

Lecture 9

Multi-cycle CPU Part 2

Pramod V. Argade
October 24, 2005

Announcements

- **Reading Assignment**

- Chapter 5. The Processor: Datapath and Control
Sections 5.1 - 5.5, 5.7 (on CD)

- **Homework None**

- **Midterm**

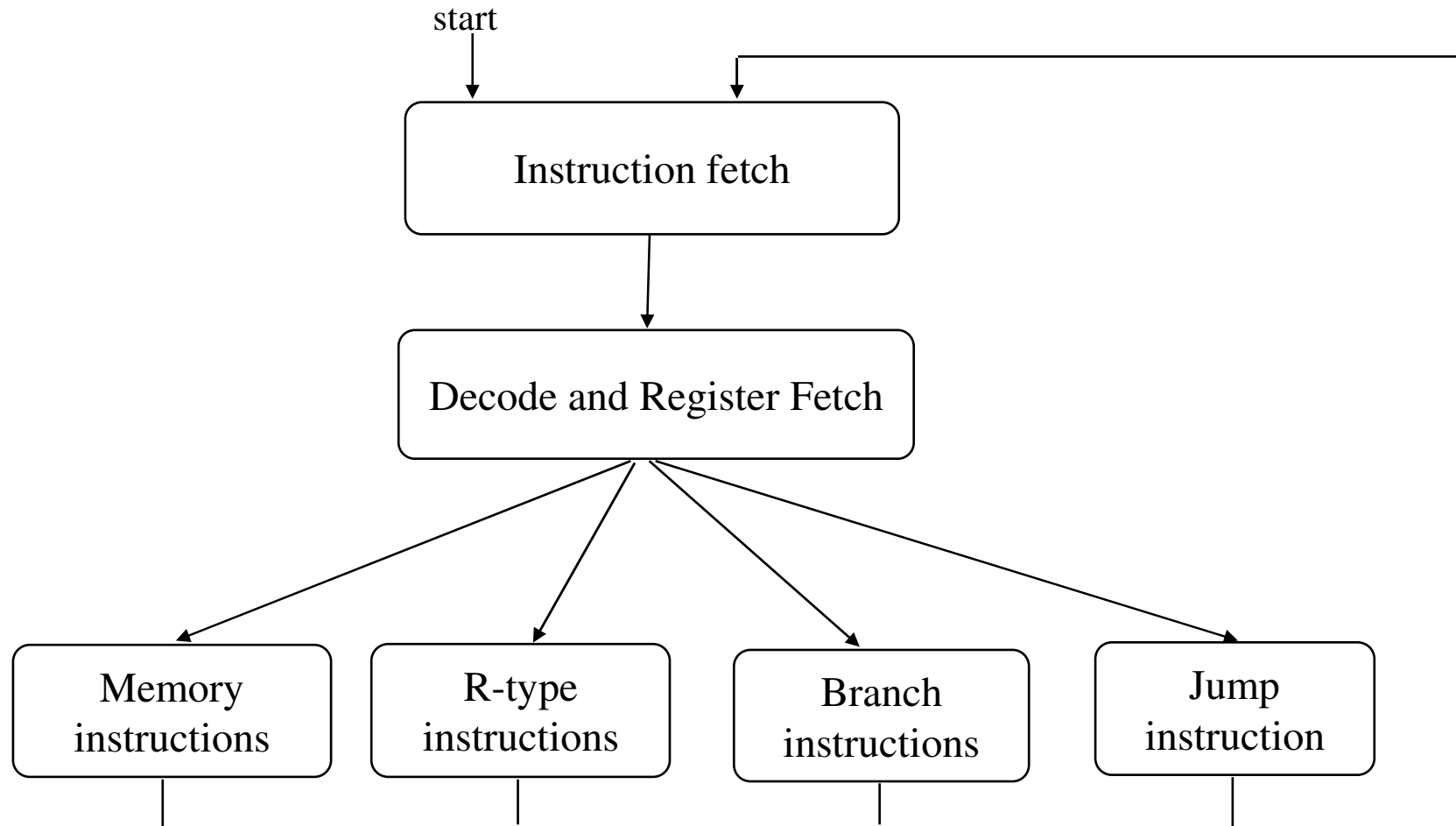
When: Mon., October 31st

Topic: Chapters 1 - 5, lecture material and HW topics
Closed book, no calculators

Course Schedule

Lecture #	Date	Day	Lecture Topic	Quiz Topic	Homework Due
1	9/26	Monday	Introduction, Ch. 1	-	-
2	9/28	Wednesday	ISA, Ch. 2	-	-
3	10/3	Monday	Arithmetic Part 1, Ch. 4	ISA	#1
4	10/5	Wednesday	Arithmetic Part 2, Ch. 4	-	-
5	10/10	Monday	Performance, Ch. 3	Arithmetic	#2
6	10/12	Wednesday	Single cycle CPU, Ch. 5	-	-
7	10/17	Monday	Single cycle CPU, Ch. 5	Performance	#3
8	10/19	Wednesday	Multi-cycle CPU, Ch. 5	-	-
9	10/24	Monday	Multi-cycle CPU, Ch. 5	Single Cycle CPU	#4
10	10/26	Wednesday	Review for the Midterm	-	-
	10/31	Monday	Mid-term Exam	-	-
11	11/2	Wednesday	Exceptions, Ch. 5 and Pipelining, Ch. 6	-	-
12	11/7	Monday	Pipelining, Ch. 6	-	-
13	11/9	Wednesday	Data and control hazards, Ch. 6	-	-
14	11/14	Monday	Data and control hazards, Ch. 6	Pipeline Hazards	#5
15	11/16	Wednesday	Memory & cache design, Ch. 7	-	-
16	11/21	Monday	Memory & cache design, Ch. 7	Cache	#6
17	11/23	Wednesday	Virtual Memory & cache design, Ch. 7	-	-
18	11/28	Monday	Course Review	-	-
	12/8	Thursday	Final Exam 7 - 10 PM		

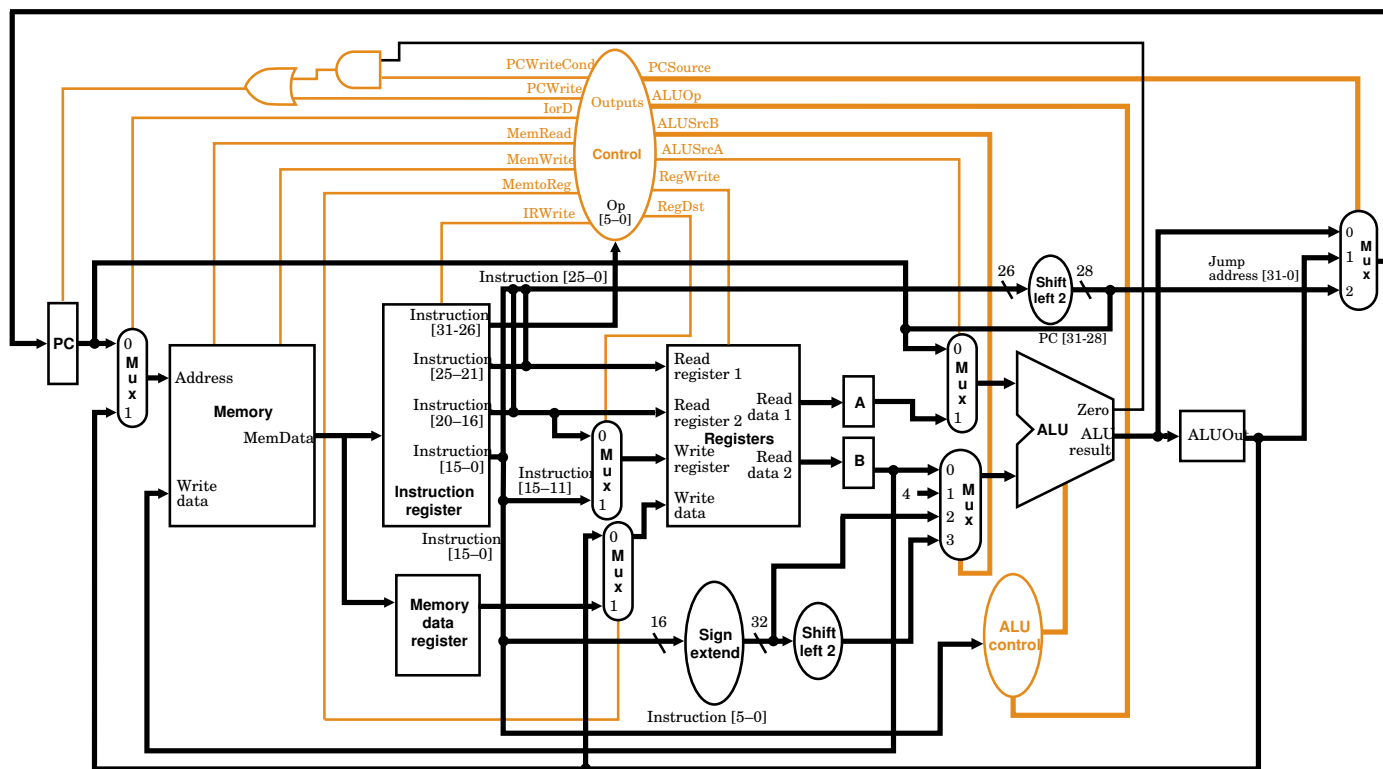
Multi-cycle CPU: Control FSM



State 1. Instruction Fetch Cycle (Instruction Independent)

$IR = \text{Memory}[PC]$

$PC = PC + 4$ (may not be final value of PC)

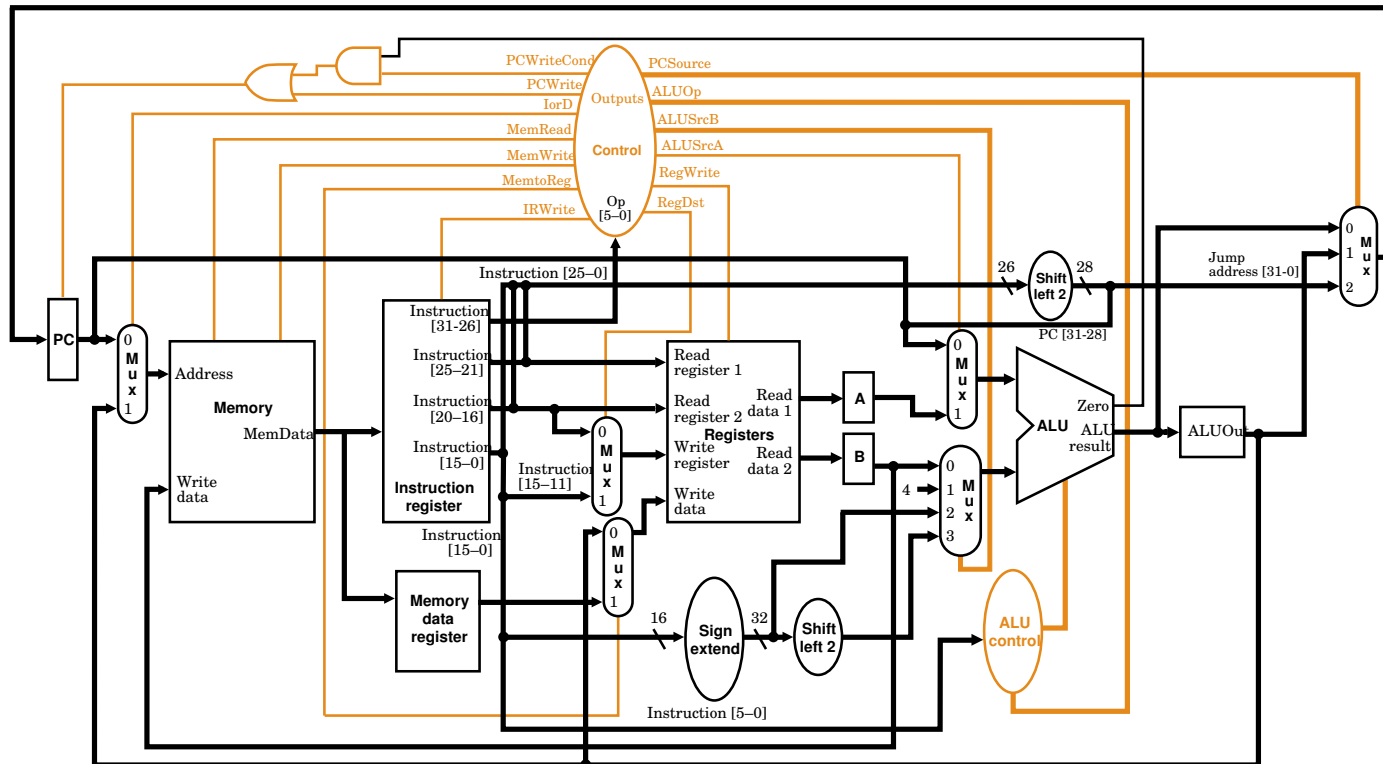


State 2. Instruction Decode and Reg. Fetch Cycle (Instruction Independent)

A = Register[IR[25-21]]

B = Register[IR[20-16]]

ALUOut = PC + (sign-extend (IR[15-0]) << 2) [May not be used]



First two states of the FSM

Instruction Fetch, *state 0*

IR = Memory[PC]

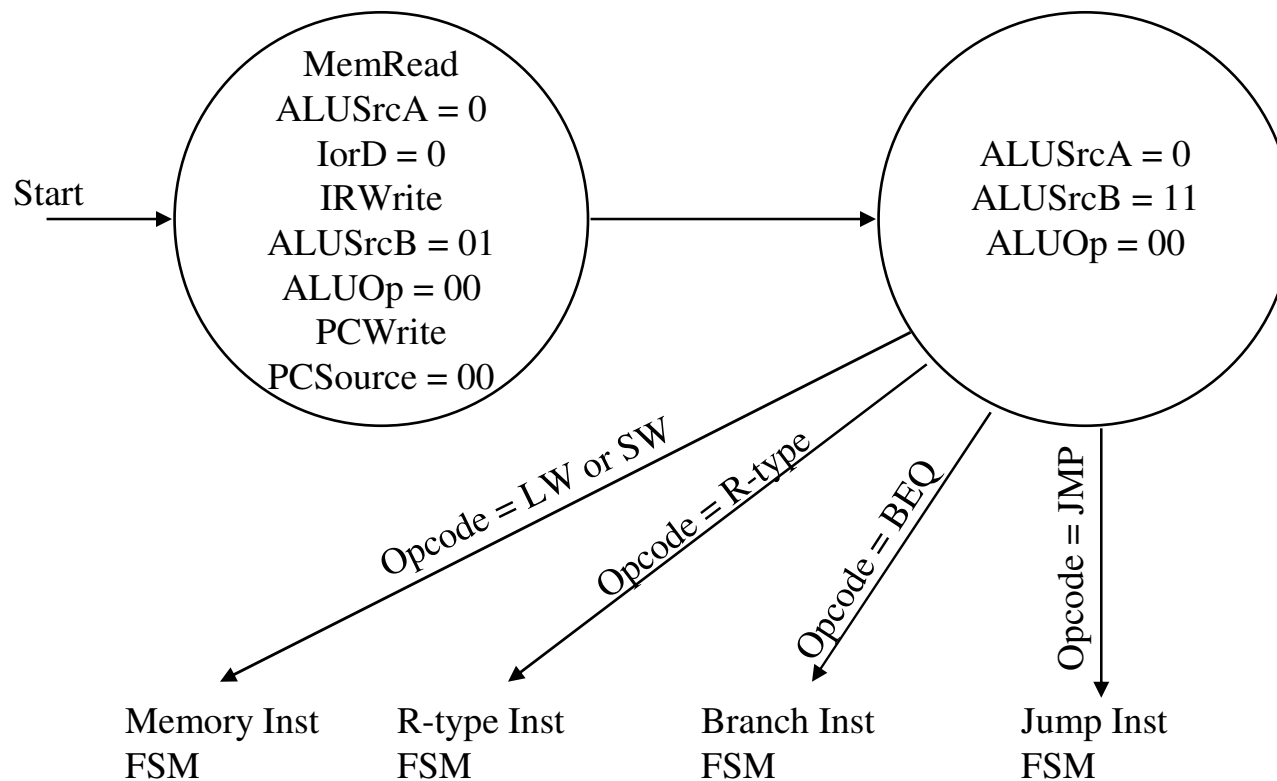
PC = PC + 4

Instruction Decode/ Register Fetch, *state 1*

A = Register[IR[25-21]]

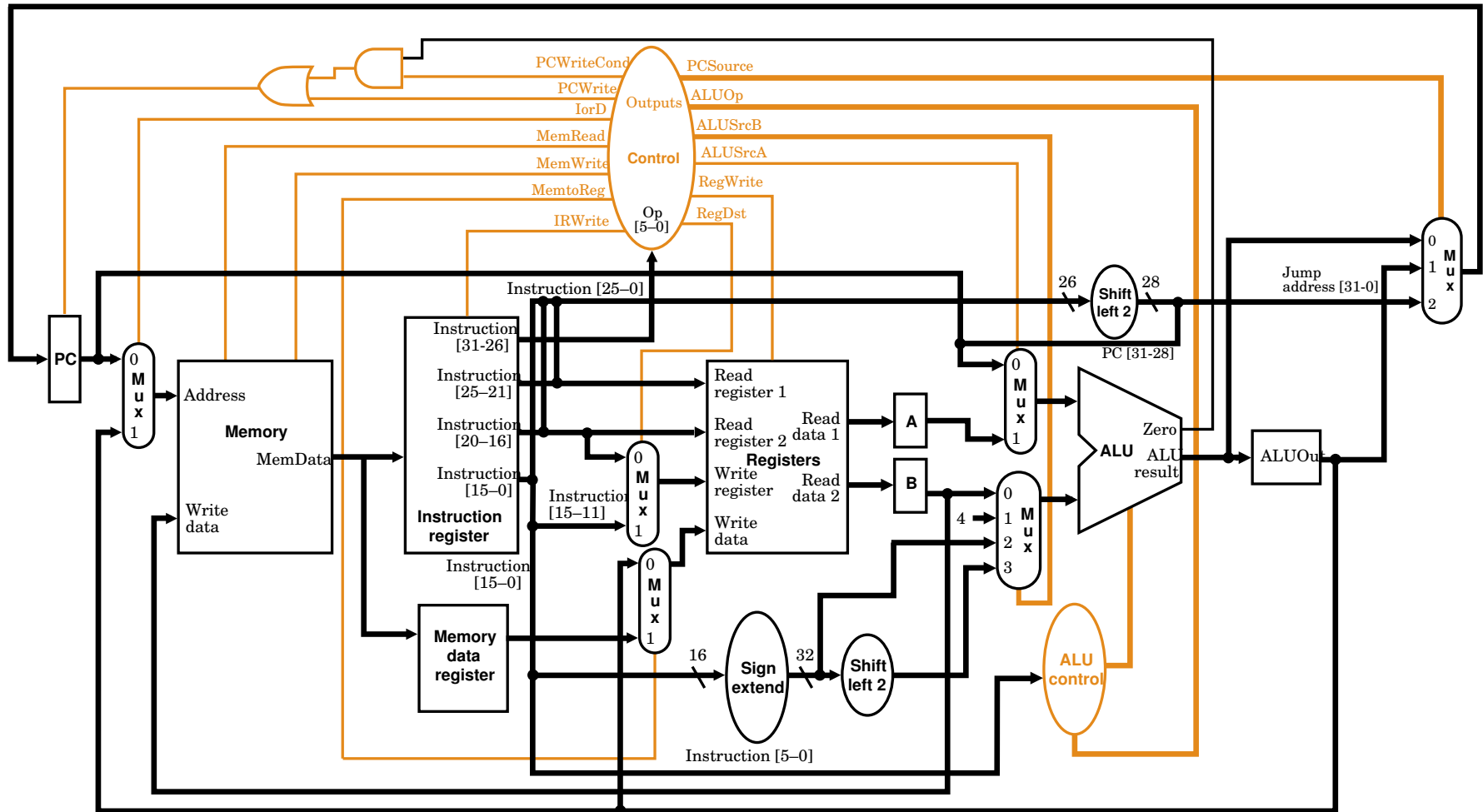
B = Register[IR[20-16]]

ALUOut = PC + (sign-extend (IR[15-0]) << 2)

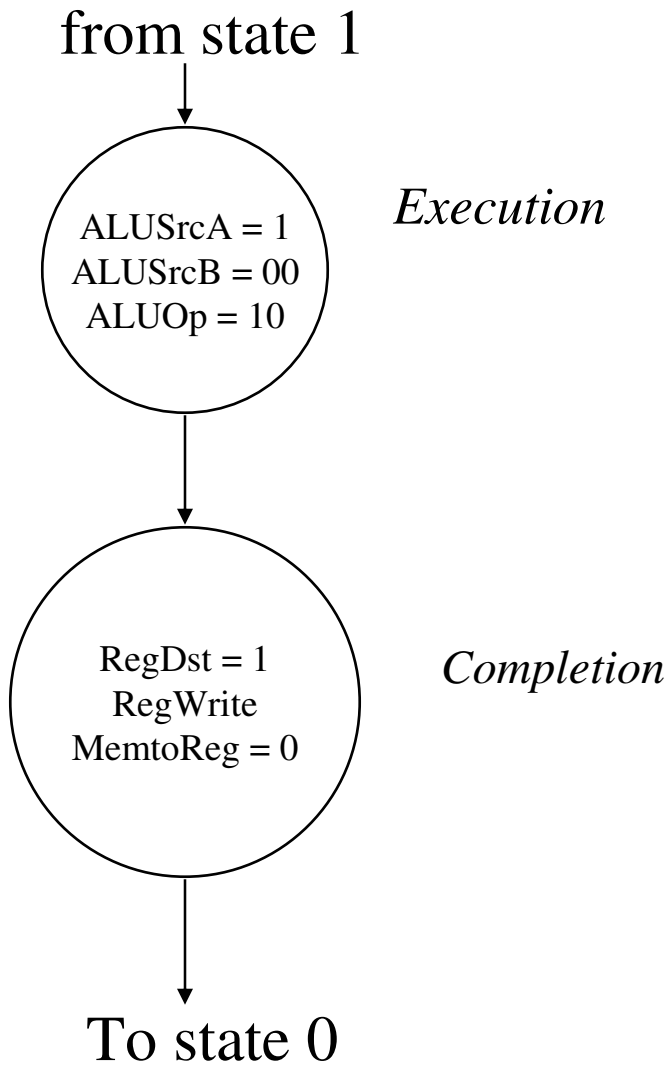


State 3. R-type Execution

$$\text{ALUOut} = A \text{ op } B$$

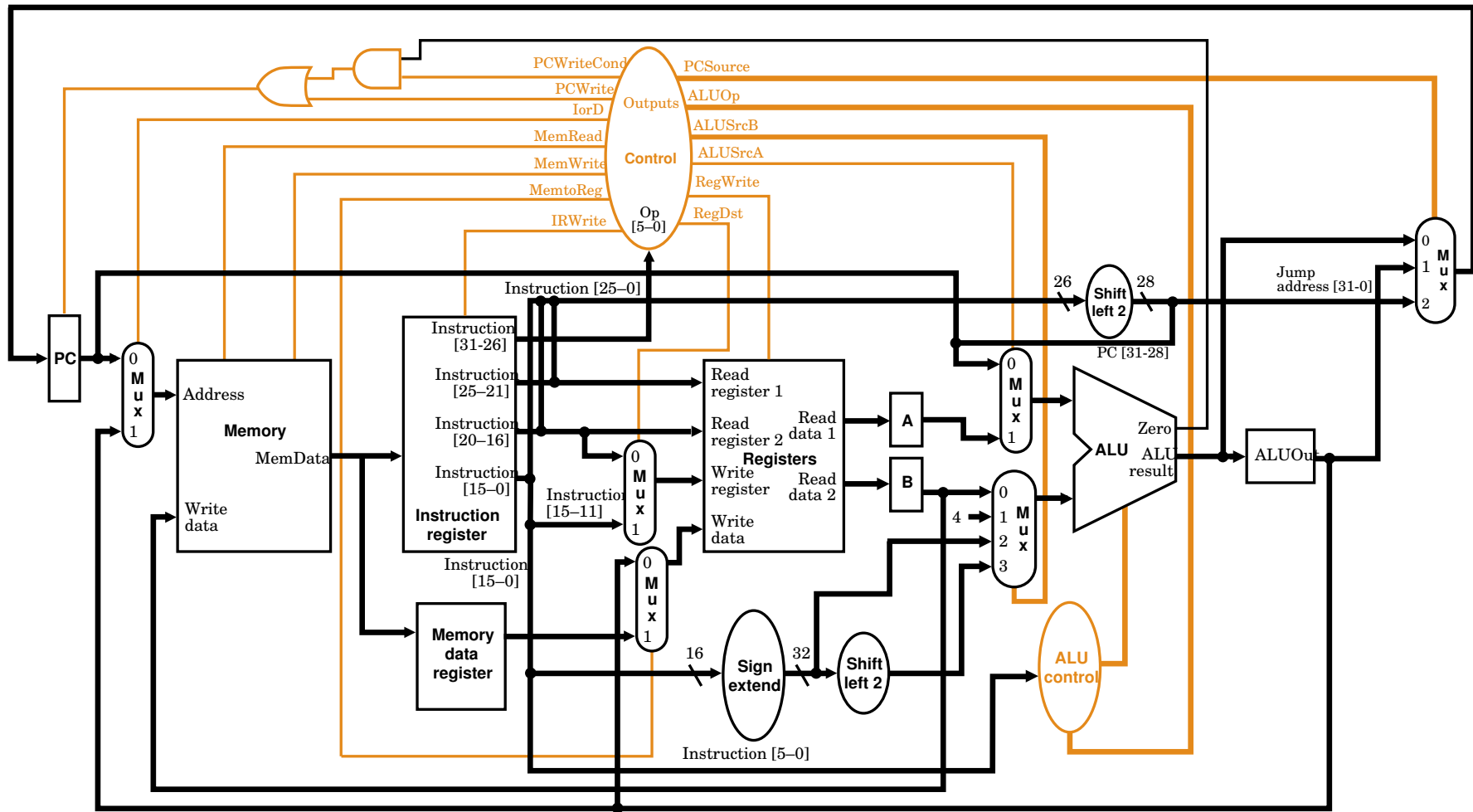


States for R-type Instructions

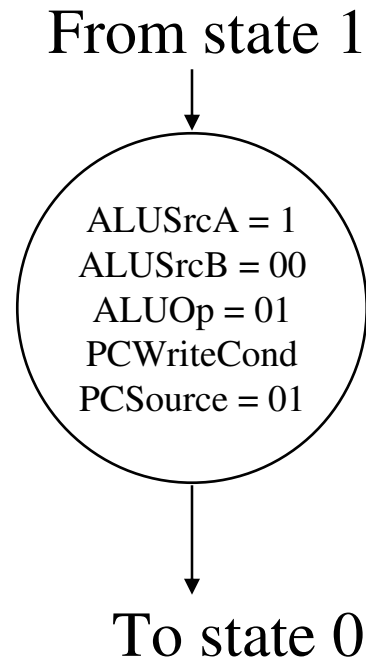


State 3. Branch completion

if (A == B) PC = ALUOut

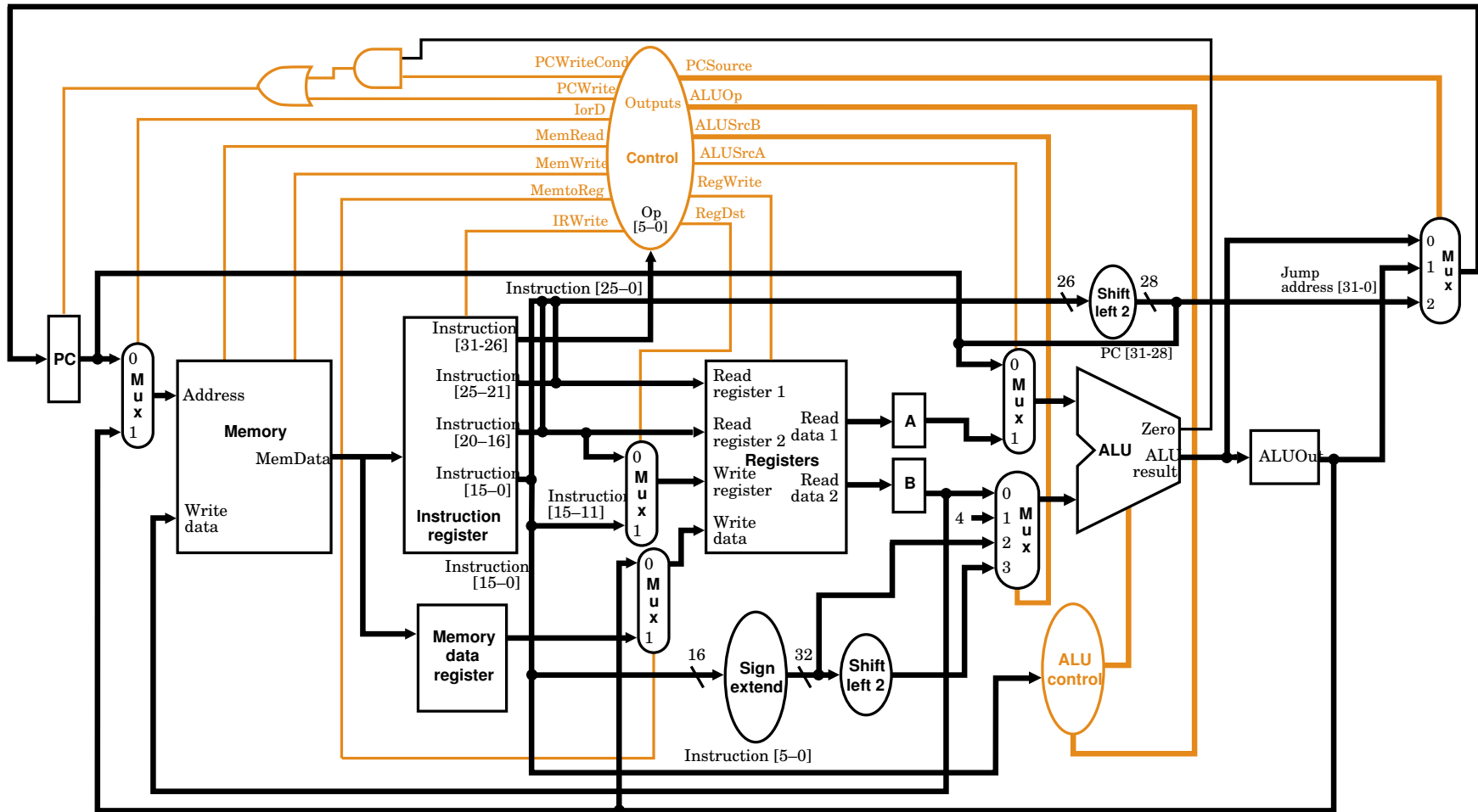


State for BEQ Instruction



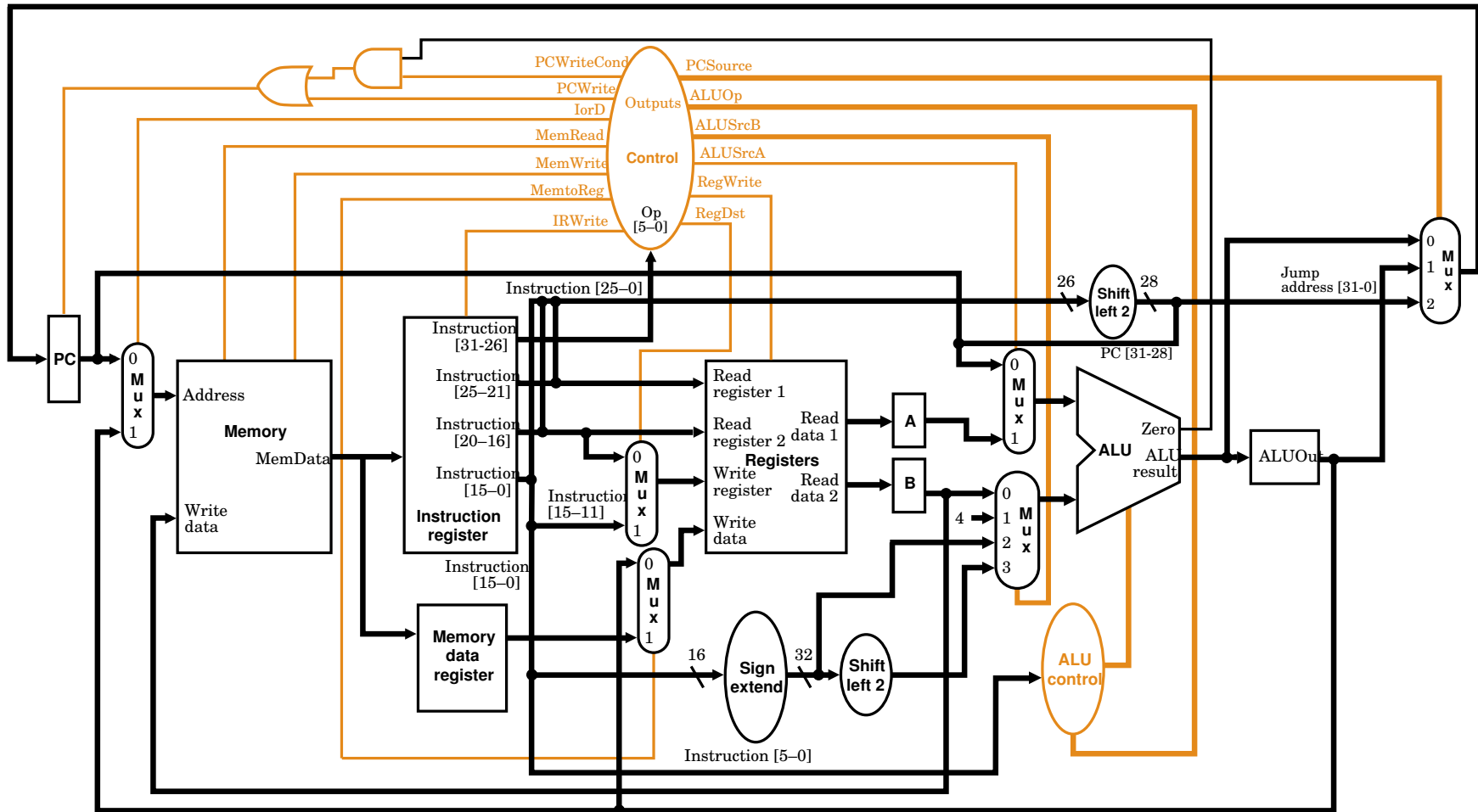
State 3. Memory address computation

$$\text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0])$$

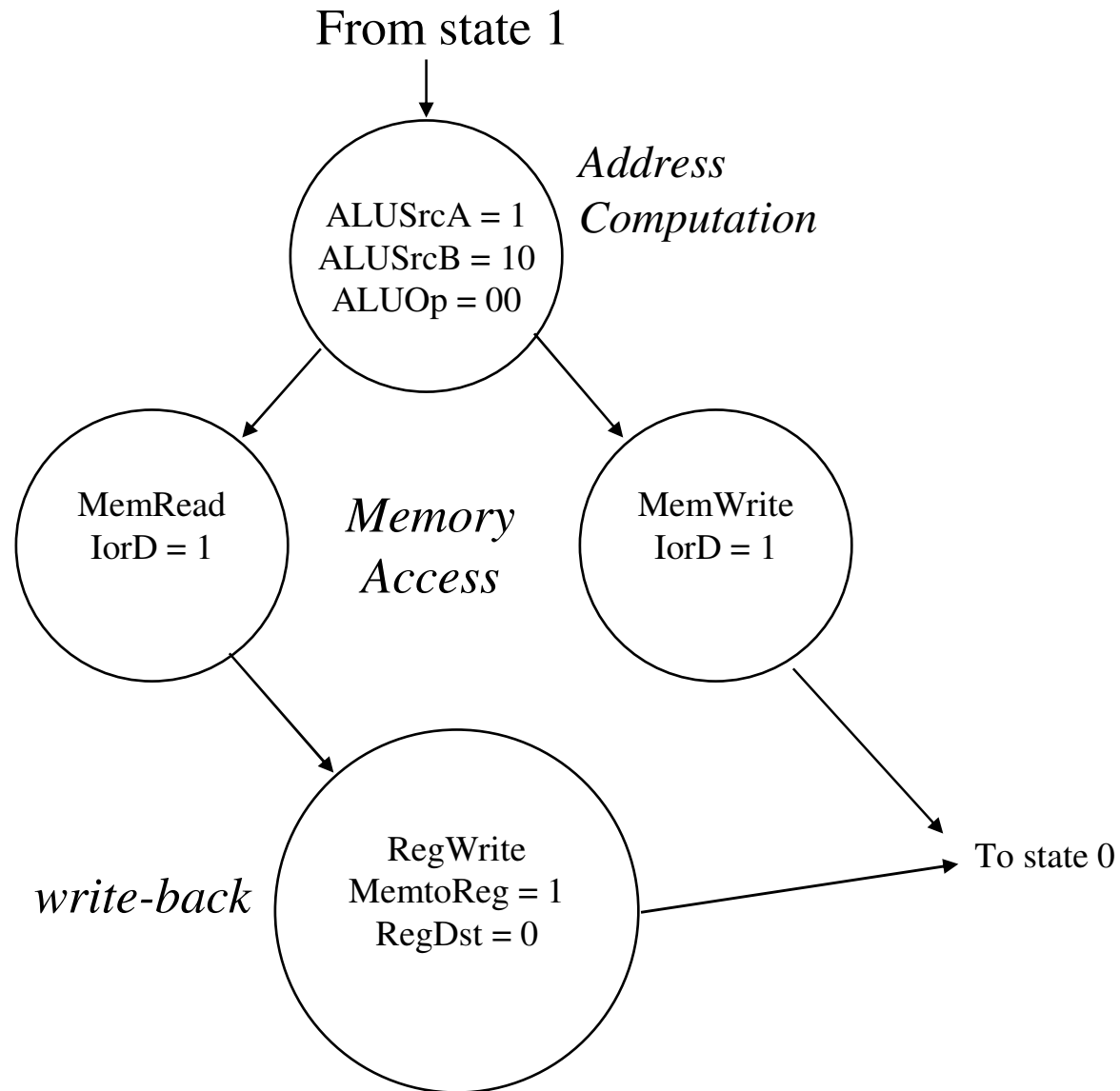


State 5. Memory Write-Back

$\text{Reg}[\text{IR}[20-16]] = \text{MDR}$ [Completes in this cycle]

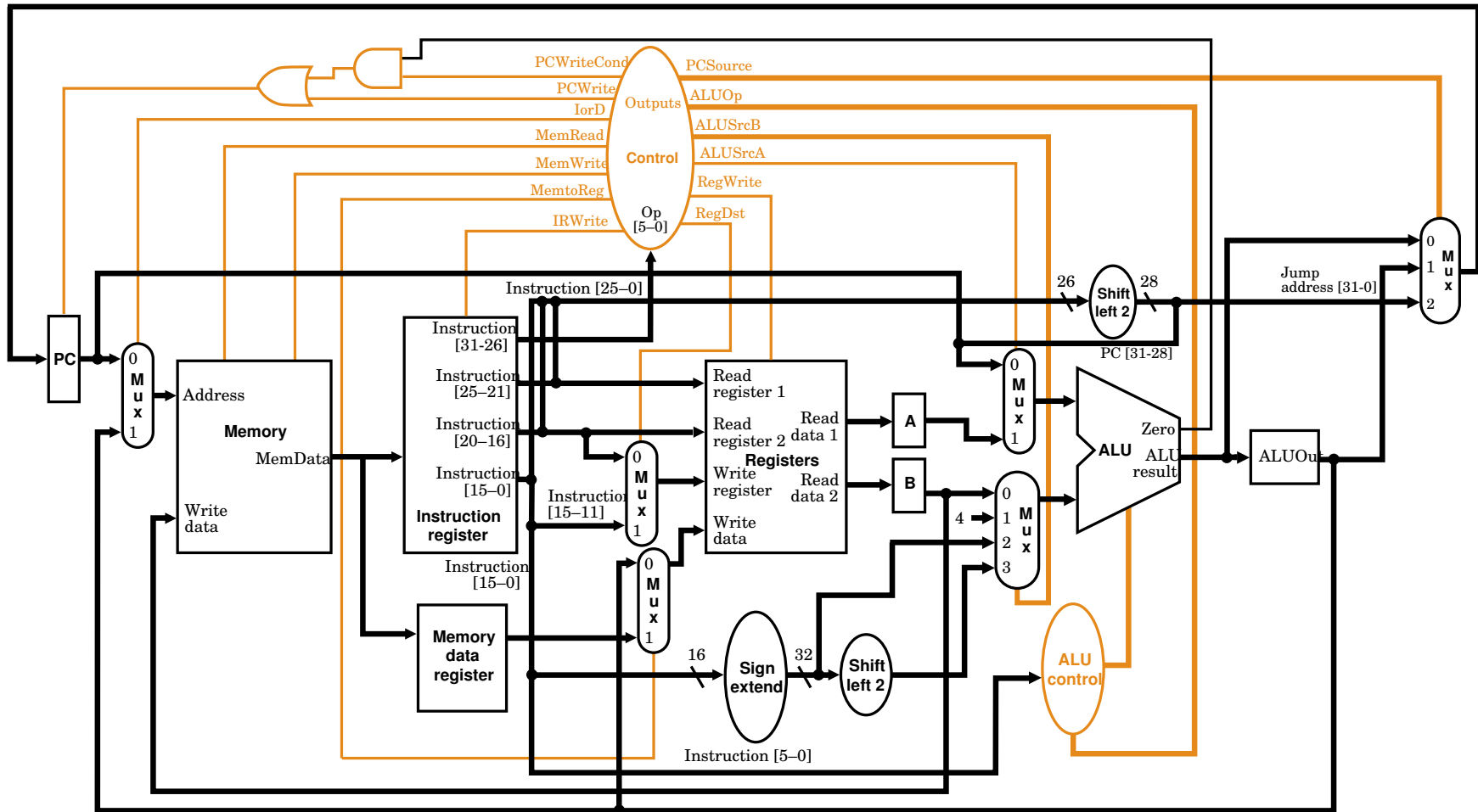


States for Memory Instructions

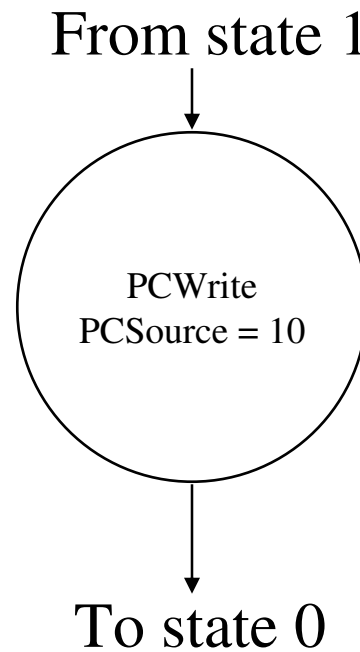


State 2: Jump Instruction Completion

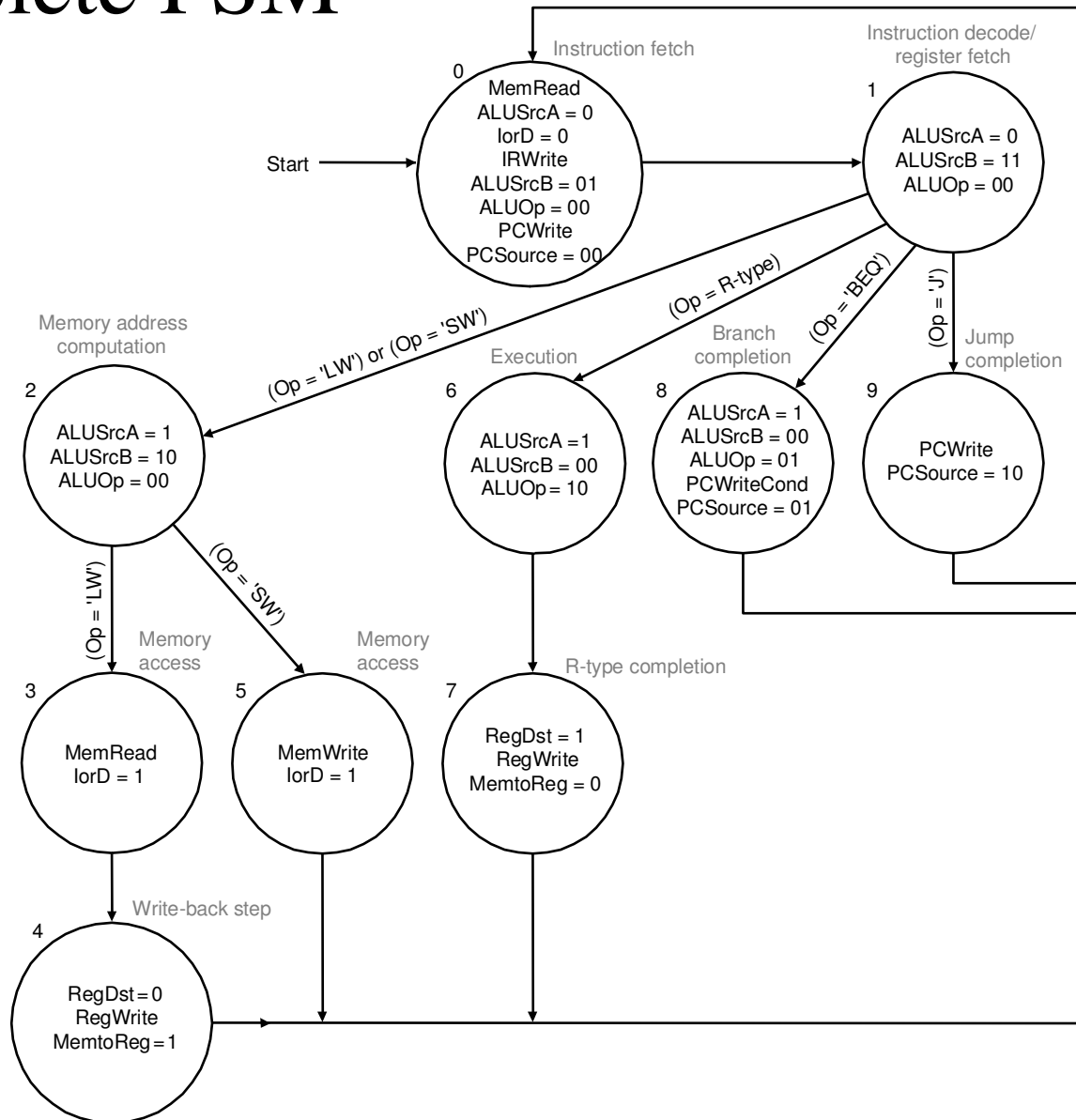
$$PC = PC[31-28] \mid (inst[25-0] \ll 2)$$



JMP Instruction



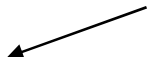
Complete FSM



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 taken
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?
- Assume 20% loads, 10% stores, 50% R-type, 20% branches, what is the CPI?

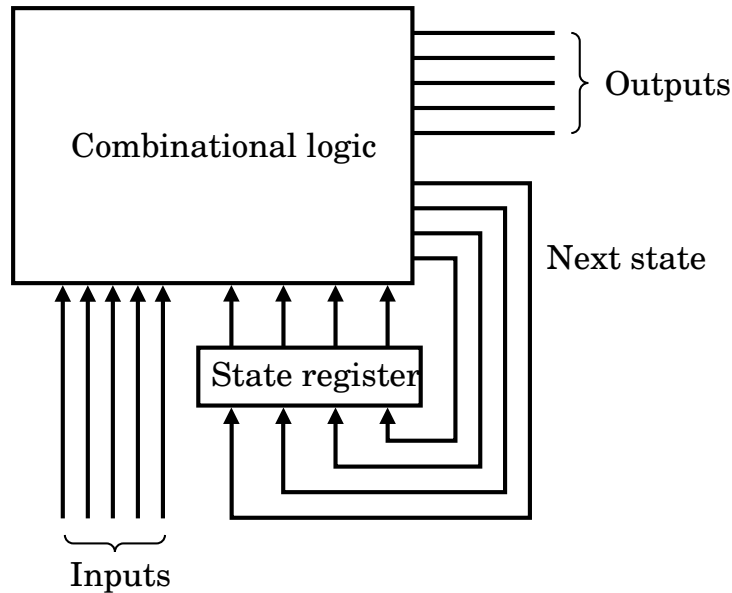
Multicycle CPU Key Points

- Performance gain achieved from variable-time instructions
 - Reduces the cycle time
- Execution Time = #Instructions * CPI * cycle time
 - Average CPI is reduced and so is cycle time
- Required very few new state elements
- More complex control signals
- Control requires FSM

Logic Equations for Control Unit

Output	Current State	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
lrd	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOP0	state8	
ALUSrcB1	state1 + state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op='lw') + (Op='sw')
NextState3	state2	(Op='lw')
NextState4	state3	
NextState5	state2	(Op='sw')
NextState6	state1	(Op='R-type')
NextState7	state6	
NextState8	state1	(Op='beq')
NextState9	state1	(Op='jmp')

Multicycle CPU: Control

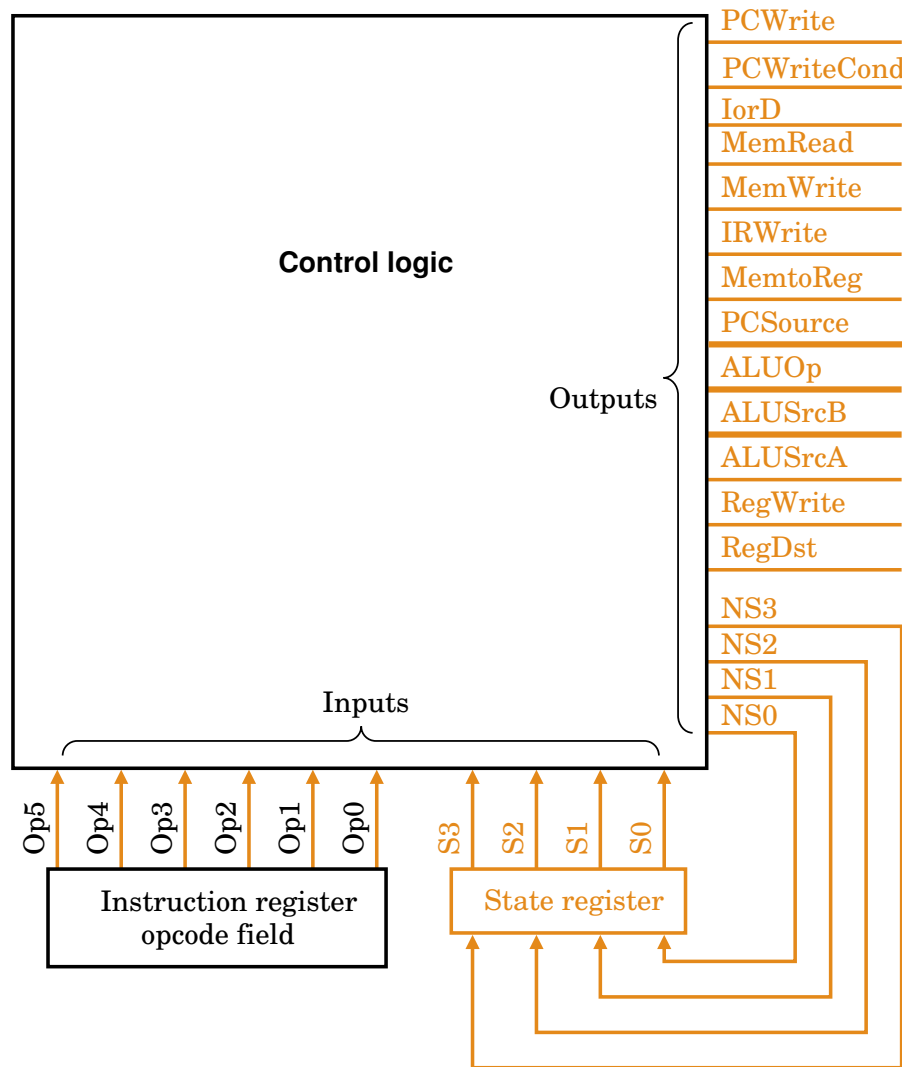


```
If (State == Instruction Fetch)
{
  IRWrite = 1;
  // All other signals are 0;
  State = Operand Fetch;
}
```

```
If (State == Execute &&
    InstructionOpCode == BEQ )
{
  // Set up signals for BEQ...
}
```

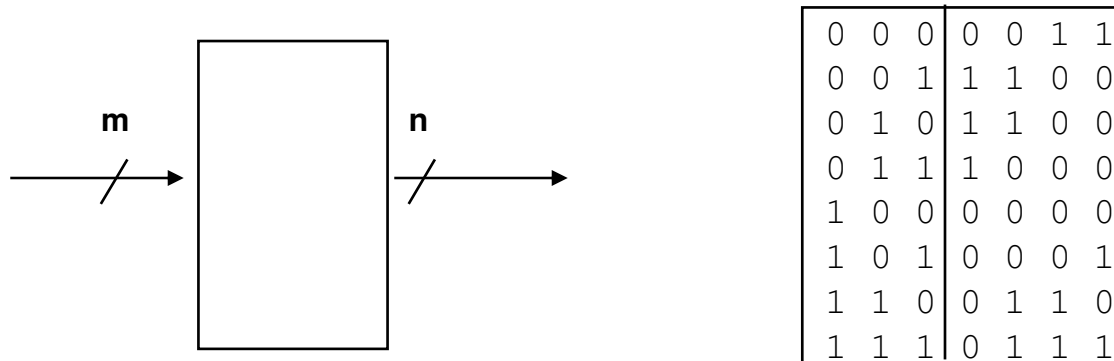
$$\begin{aligned} \text{ControlOutput} &= f(\text{State}, \text{OpCode}) \\ \text{NextState} &= f(\text{State}, \text{OpCode}) \end{aligned}$$

Finite State Machine for Control



FSM: ROM Implementation

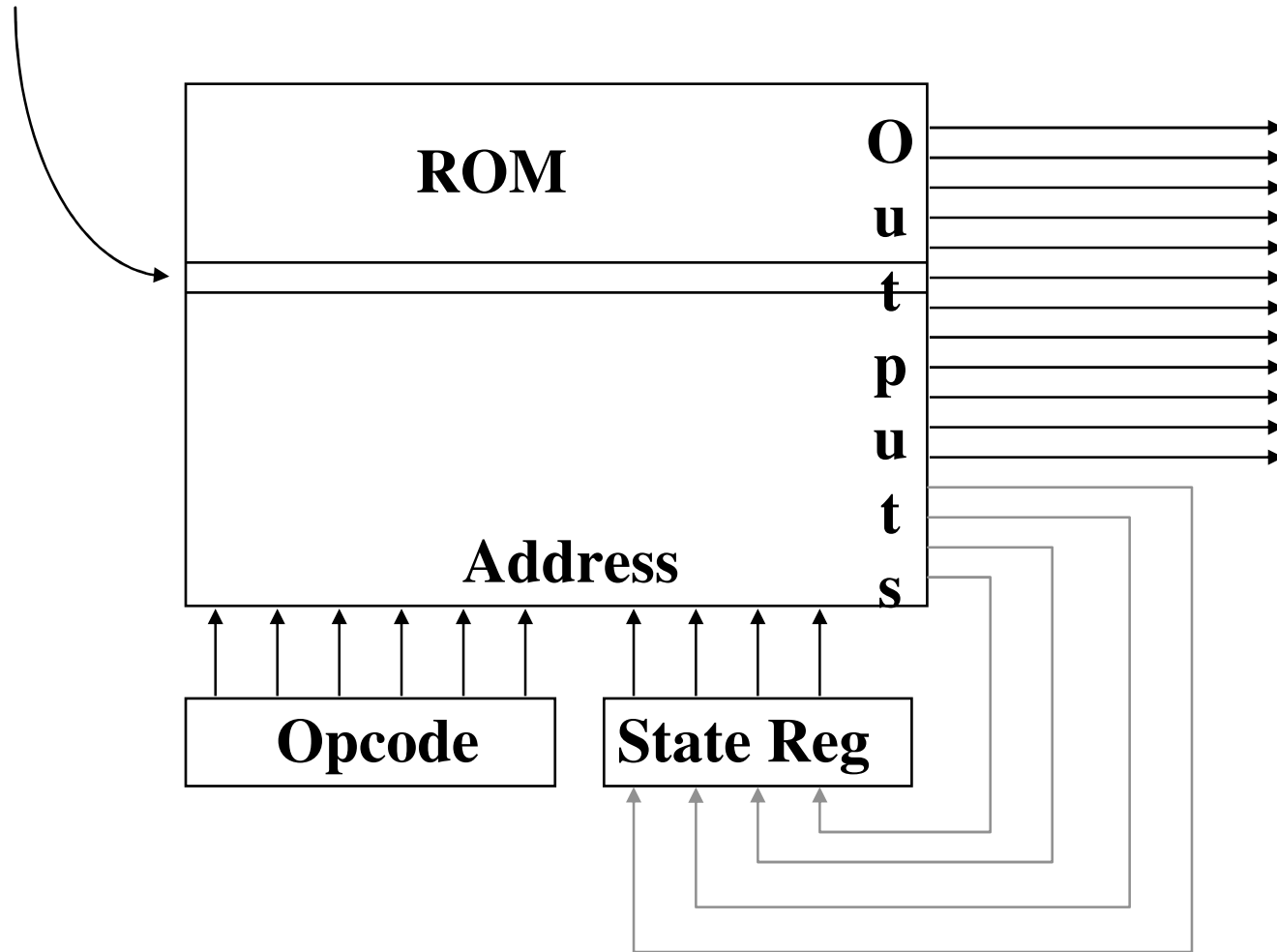
- ROM = "Read Only Memory"
 - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
 - if the address is m -bits, we can address 2^m entries in the ROM.
 - ROM outputs are the bits of data that the address points to
 - ROM outputs are the control and next state signals



2^m is the "height", and n is the "width"

Implementing a control FSM with ROM

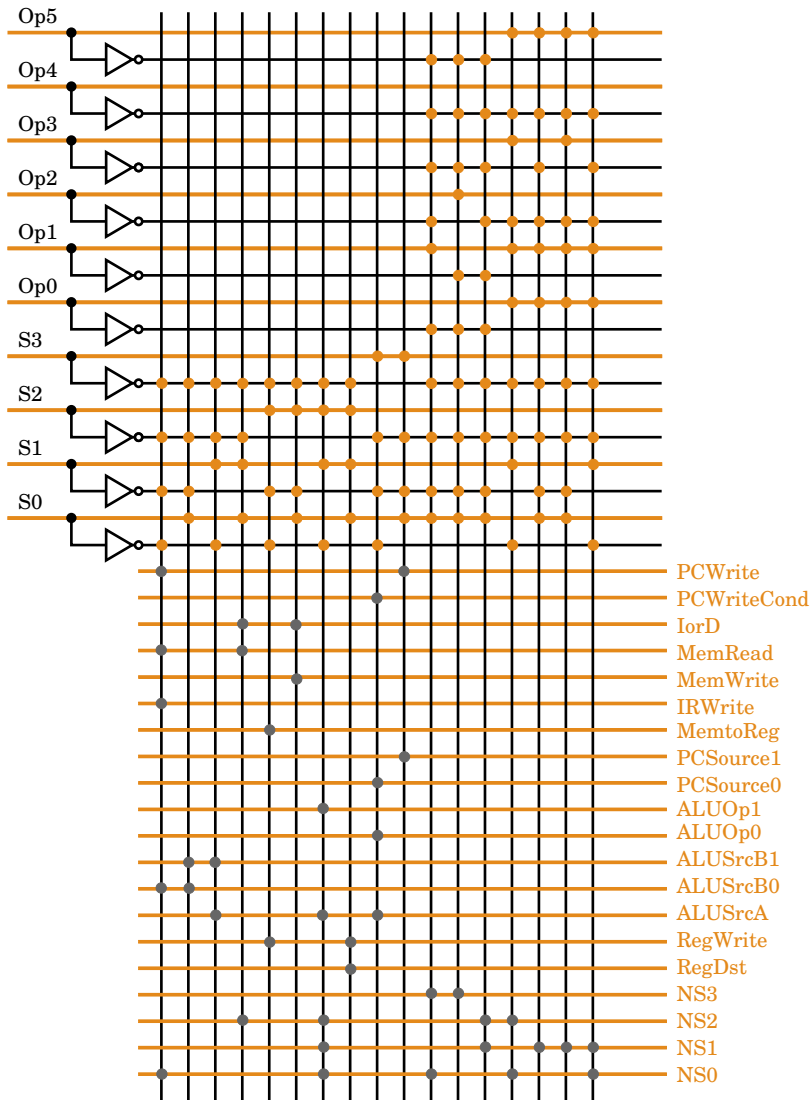
Each line in the ROM contains control signal outputs (an operation), and next-state outputs (branch destination)



FSM: ROM Implementation

- How many inputs are there?
6 bits for opcode, 4 bits for state = 10 address lines
(i.e., $2^{10} = 1024$ different addresses)
- How many outputs are there?
16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is $2^{10} \times 20 = 20\text{K}$ bits (and a rather unusual size)
- Very wasteful, since for lots of the entries, the outputs are the same
— i.e., opcode is often ignored

FSM: Programmed Logic Array (PLA)



Examples:

PCWrite = state0 + state9

PCWriteCond = state8

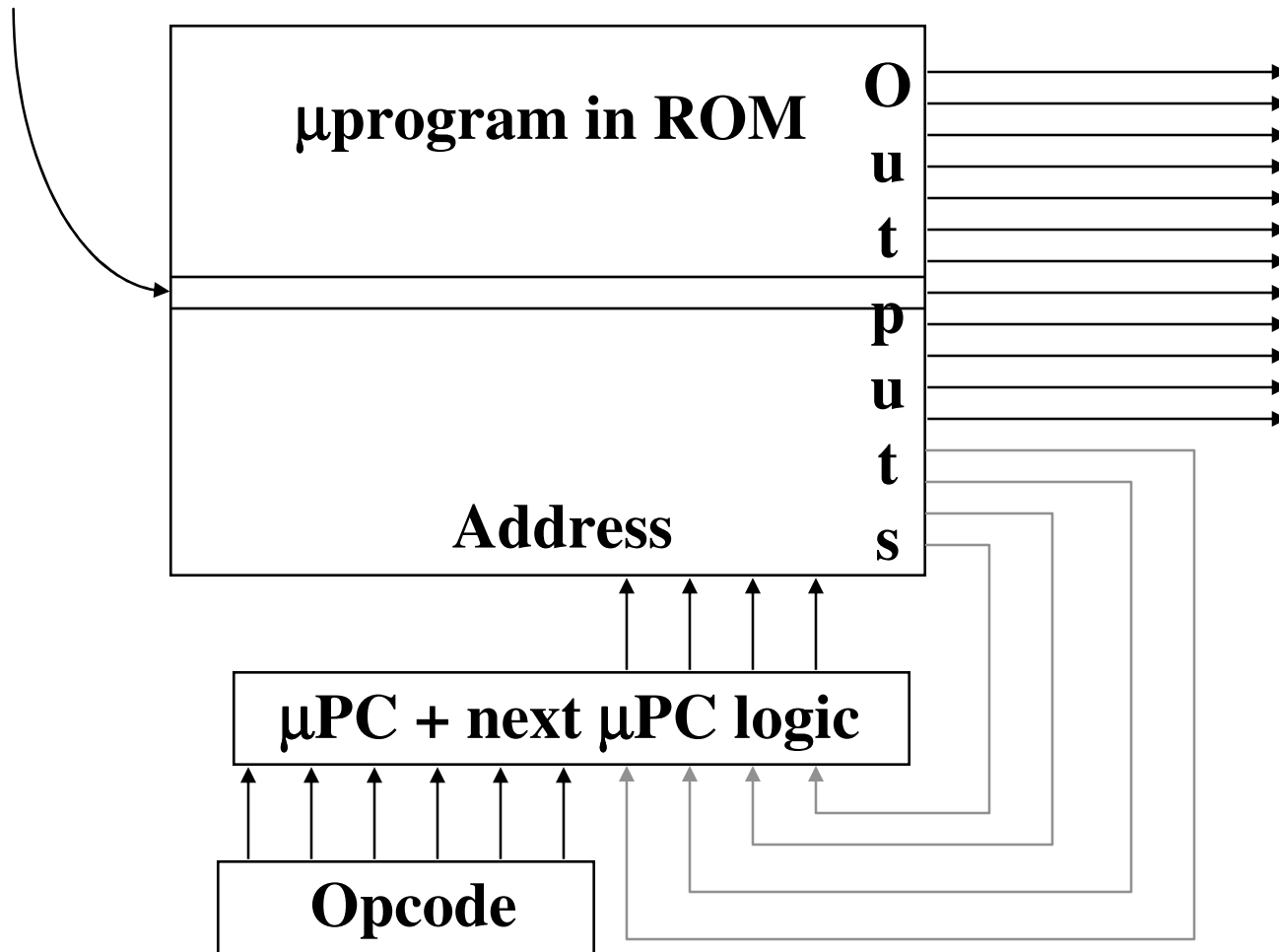
IorD = state3 + state5

The Problem with FSMs as control sequencers

- They get unmanageable quickly as they grow.
 - Hard to specify
 - Impractical to manipulate
 - Error prone
 - Difficult to visualize
- MIPS-32 instruction set contains over 100 instructions!
 - In one implementation instructions take 1 - 20 cycles
 - Graphical representation would be impractical
- Solution: Microprogramming
 - Uses ideas from programming
 - Think of the set of control signals that must be asserted in a state as an instruction to be executed by the data path.
 - These low level instructions are called “microinstructions”

Implementing a control FSM with a microprogram

Each line in the ROM is now a microprogram instruction, corresponding to a FSM state, with an operation (control signals) and branch destination (next state info).



Microprogramming

- Each microinstruction typically specifies control information
 - Must also specify sequencing information
 - What micro-instruction should be executed next?
- Control signals are grouped into “fields”
 - Make it impossible to write inconsistent microinstruction
- Microcode subroutines reuse code
 - Return address stack is provided in the control unit
- Translated by a program to control logic
- If a microprogram is fundamentally the same as the FSM, what’s the big deal?
 - Easier to specify (program), visualize, and manipulate.
 - Allows us to think about the control symbolically
 - Easier to maintain, fix bugs
 - Attractive choice for large and complex control

Microinstruction Fields

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, lorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

First six fields control the datapath & the Sequencing field specifies how to select the next microinstruction

Microprogramming

- Selection of next address of microinstruction
 - Sequential (Seq): Increment the current address
 - Branch (Fetch): Branch to the microinstruction that begins next MIPS instruction
 - Dispatch: Select next instruction based on control unit input
 - Implemented by a table indexed by control unit input
 - There may be multiple dispatch tables

A Microprogram

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch:	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1:	Add	A	Extend				Dispatch 2
LW2:					Read ALU		Seq
				Write MDR			Fetch
SW2:					Write ALU		Fetch
Rformat1:	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1:	Subt	A	B			ALUOut-cond	Fetch
JUMP1:						Jump address	Fetch

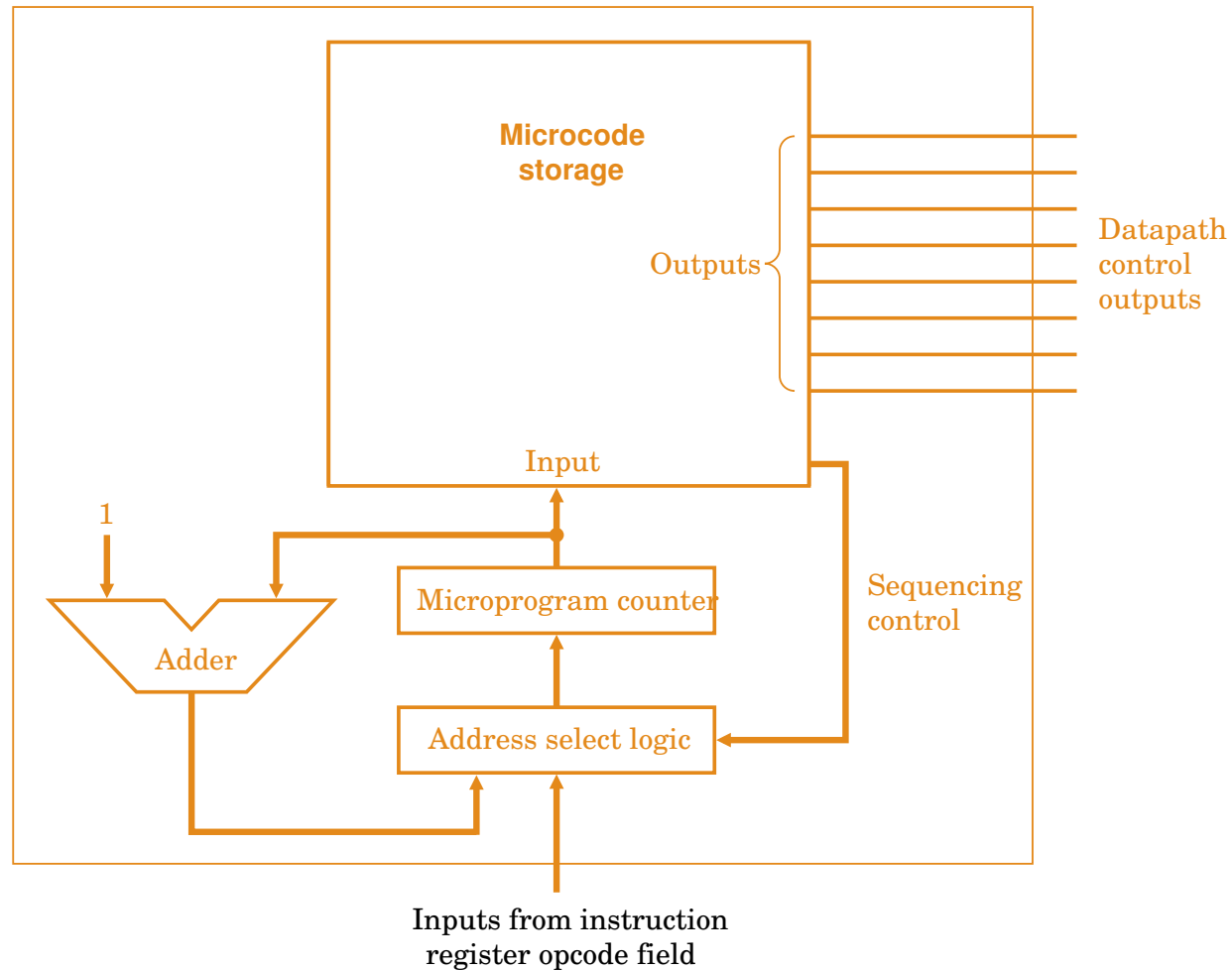
Dispatch Table 1 is used to select one of four microinstruction sequences

- OPCODE field is used to dispatch
- Mem1, Rformat1, BEQ1, JUMP1

Dispatch Table 2 is used to select one of two microinstruction sequences

- OPCODE field is used to dispatch
- LW2, SW2

Microprogram Implementation



Multi-cycle CPU: Summary

- Instructions take variable number of cycles to complete
 - Reduces (CPI * Cycle time)
 - Improves performance
- Efficient use of HW elements
 - Blocks are reused
- Control is harder
- Possible to make “common case” faster
 - Improved implementation techniques help
 - Unlike single-cycle CPU
- Microprogramming can simplify (conceptually) CPU control generation
 - A microprogram is a small program inside the CPU that executes the individual instructions of the “real” program.

Announcements

- **Reading Assignment**

- Chapter 5. The Processor: Datapath and Control
Sections 5.1 - 5.5, 5.7 (on CD)

- **Homework None**

- **Midterm**

When: Mon., October 31st

Topic: Chapters 1 - 5, lecture material and HW topics
Closed book, no calculators