WWW.MWFTR.COM

EECE 417 Computer Systems Architecture

Department of Electrical and Computer Engineering
Howard University

Charles Kim

Spring 2007

Computer Organization and Design (3rd Ed)

-The Hardware/Software Interface

by

David A. Patterson
John L. Hennessy

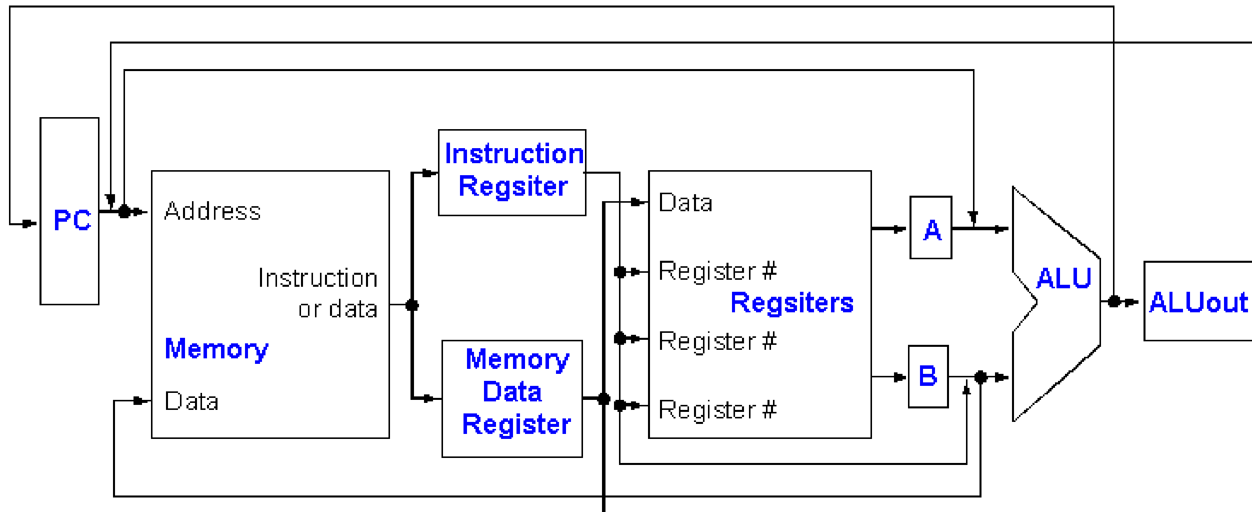
Chapter Five

The Processor: Detapath and Control

Part B

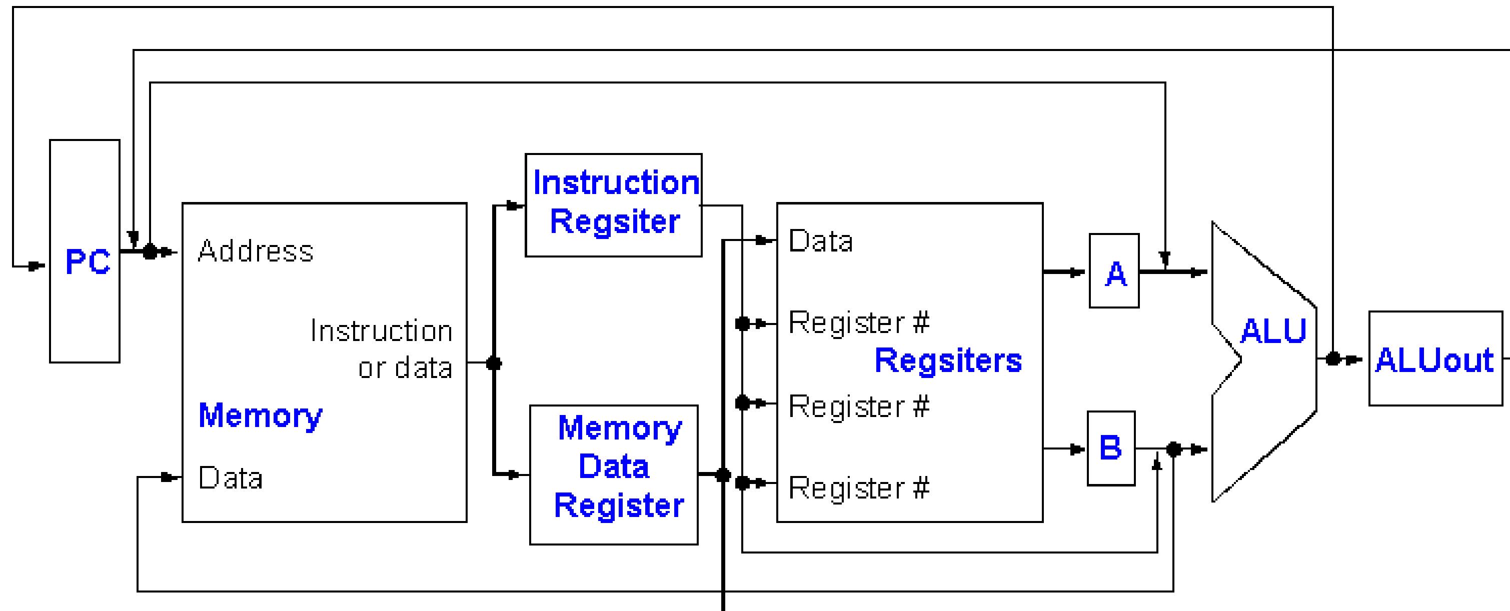
Multi-Cycle Implementation

- A Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath:



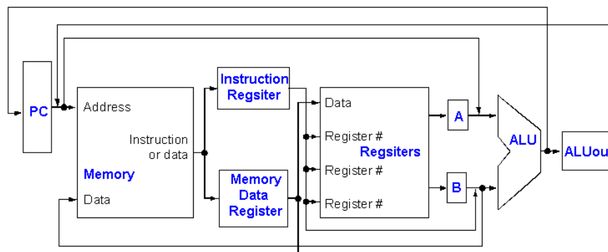
Overview of Multi-Cycle Approach (1)

- Overview
  - *step*: 1 clock cycle in execution
  - Functional units allowed to be used more than once per instruction (as long as it is used on different clock cycle)
  - Single memory unit (instruction and data)
  - Single ALU (rather than ALU and two ADDs)



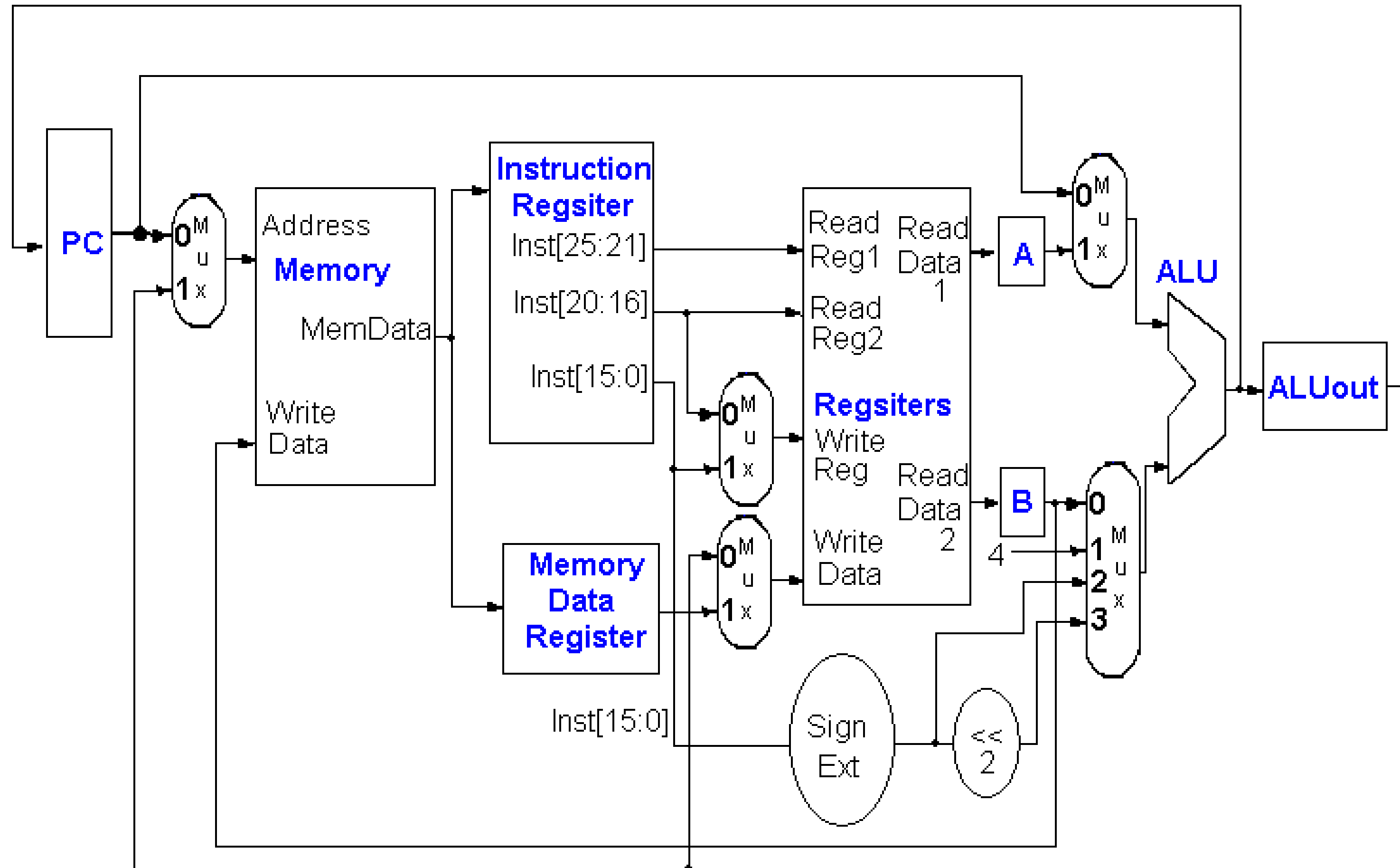
Overview of Multi-Cycle Approach (2)

- Overview (-continued)
  - One of more registers are *added* after every functional unit to hold the output until the value is used in a subsequent clock cycle
  - At the end of clock, all data that is used in the subsequent clock cycle must be stored in a *state element*.
    - Data used by subsequent instructions in a later clock cycle is stored into one of:
      - Register File
      - PC
      - Memory
    - Data used by the same instruction in a later clock must be stored into one of the *added registers*
  - Added Temporary Registers
    - IR (Instruction Register): save for instruction read
    - MDR(Memory Data Register): save for data read
    - A (Register A): register operand read from register file
    - B (Register B): register operand read from register file
    - ALUout (ALU output register): hold the output of ALU



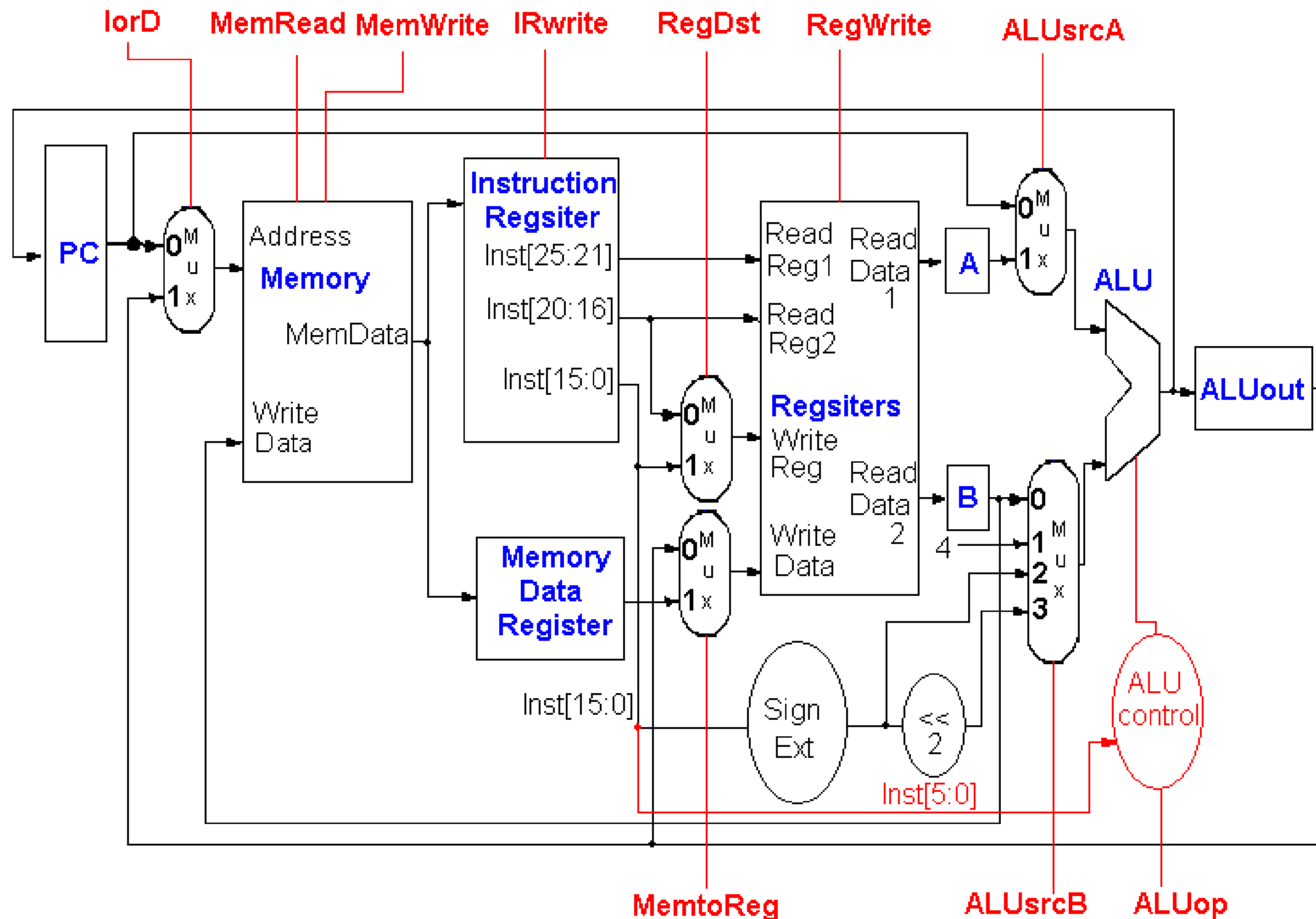
Multiplxor for Multi-Cycle Approach

- Single Memory –Mem (from ALUout)/Inst (from PC))
- IR needs to hold the instruction until the end of execution
- Single ALU

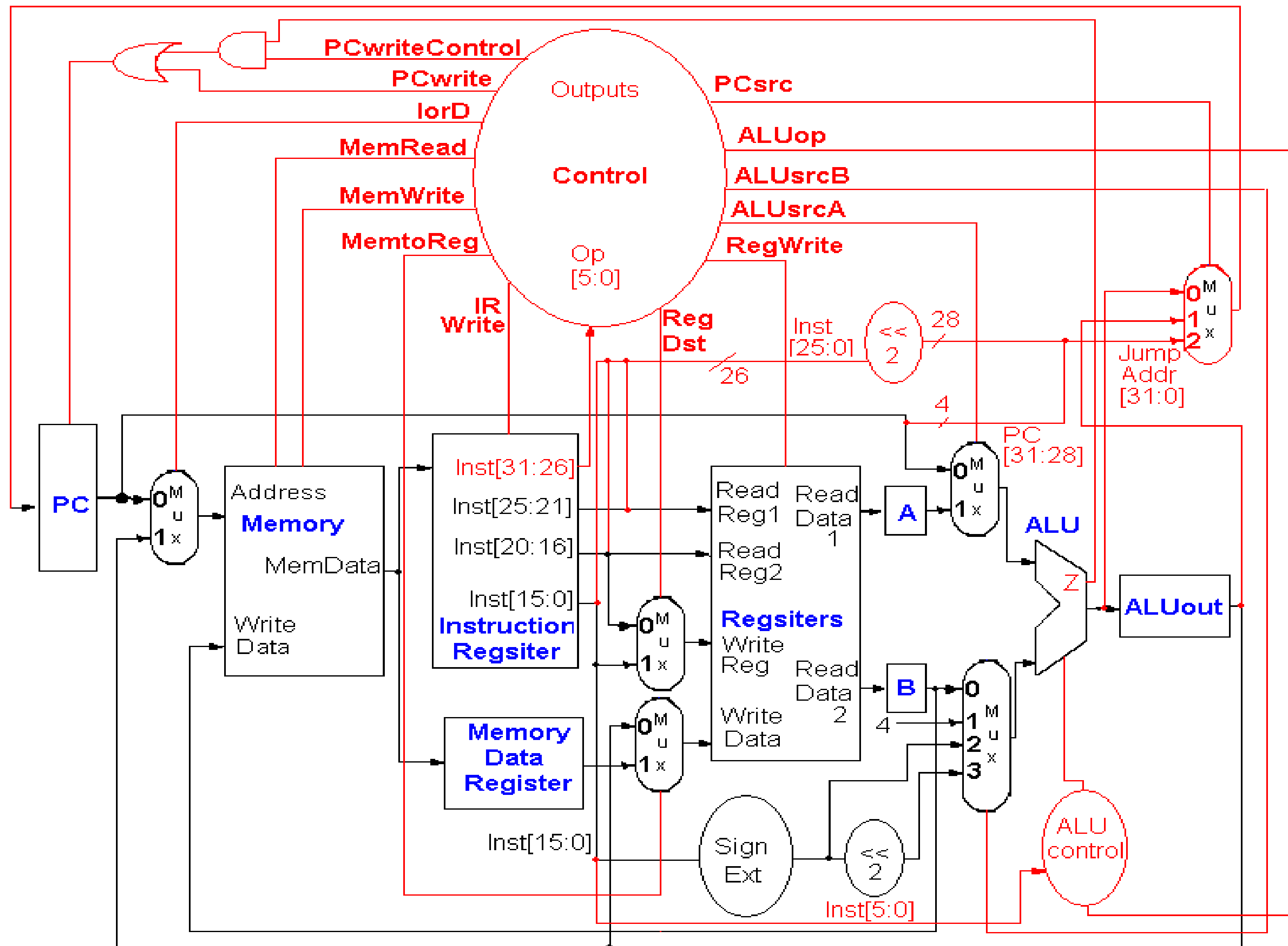


Control Signals for Multi-Cycle Datapath

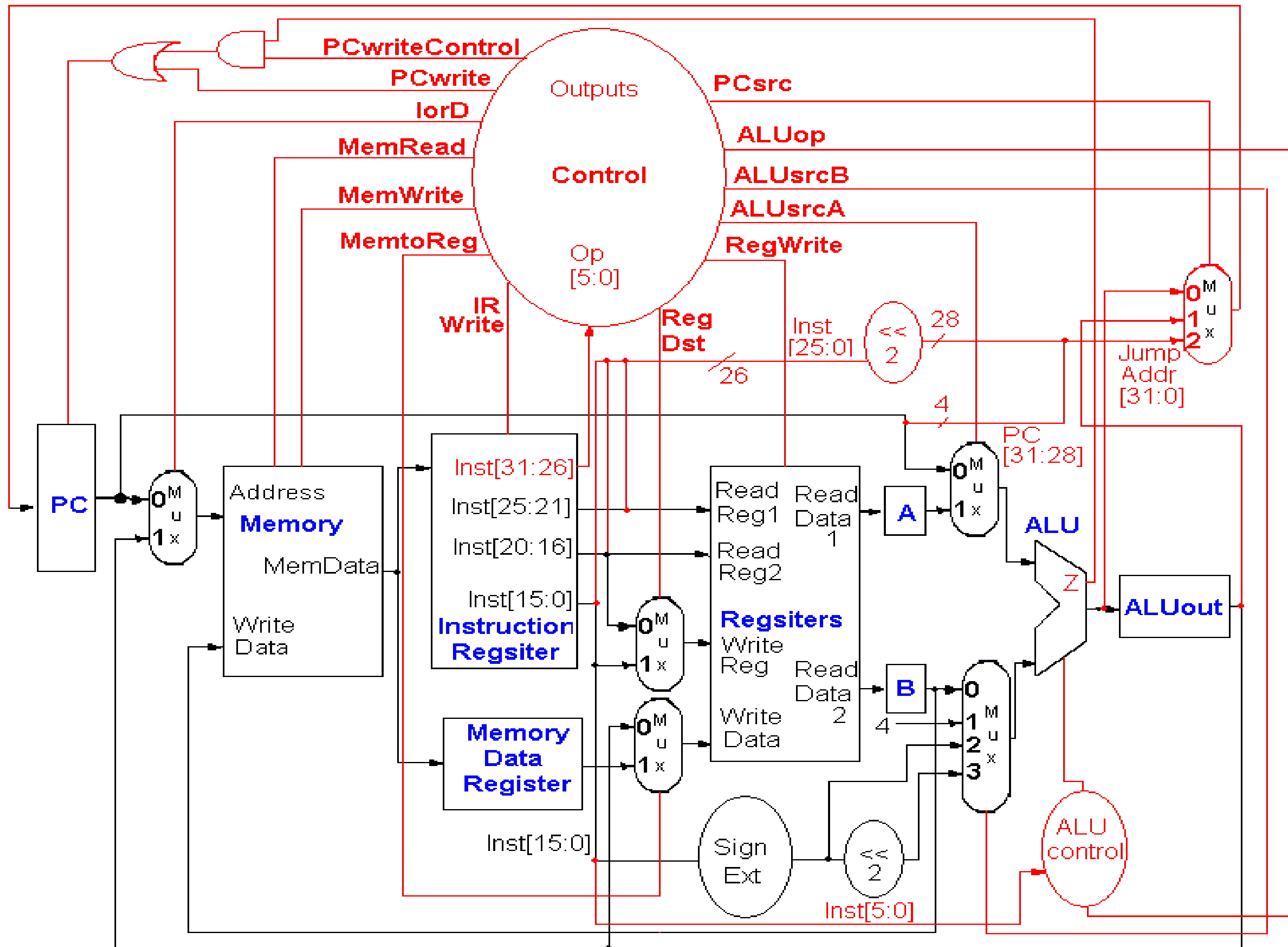
- Jump instr is still not included -PC
- Controlled branch is not included – PC



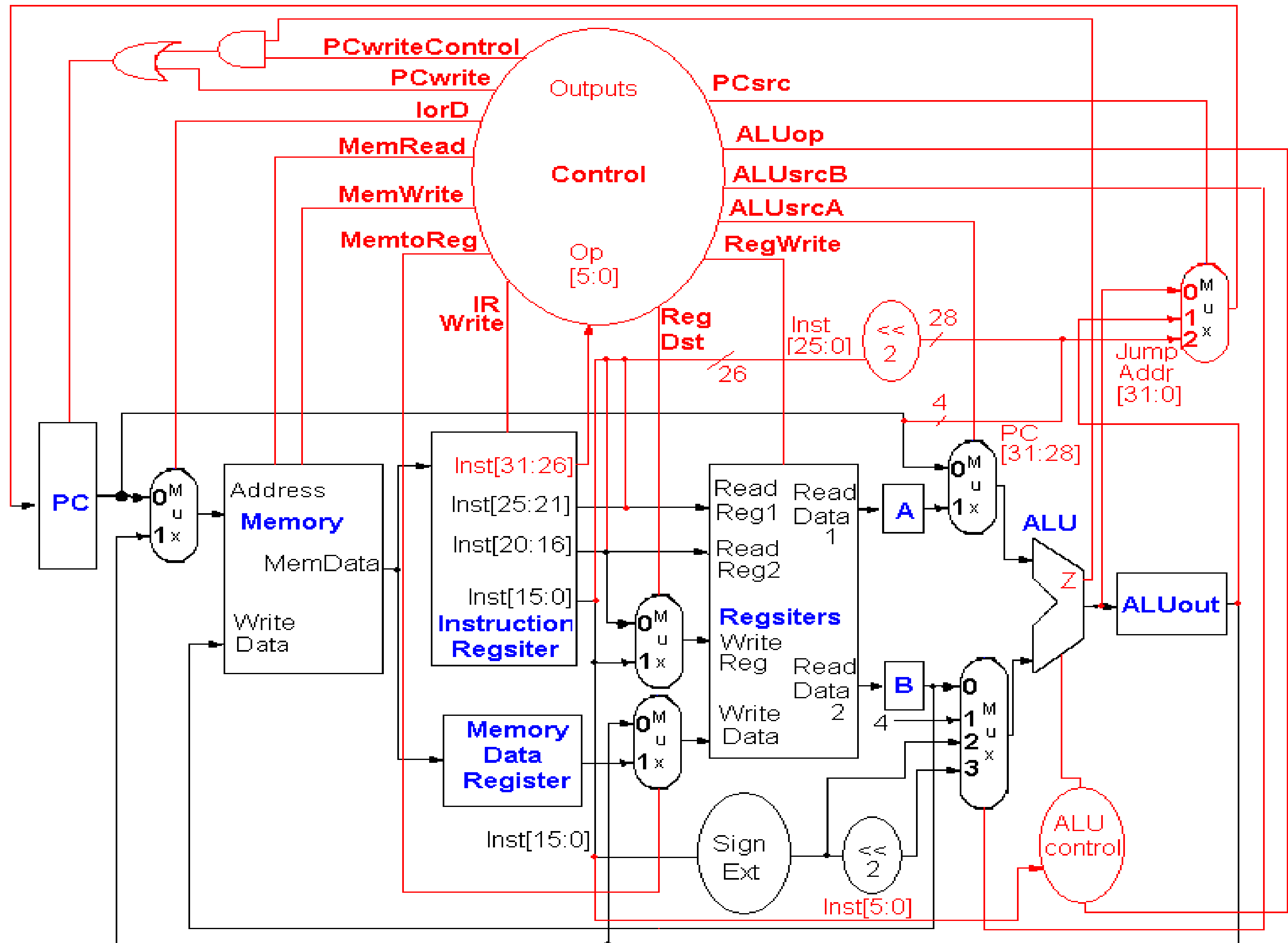
Complete Multipath Datapath and Control unit



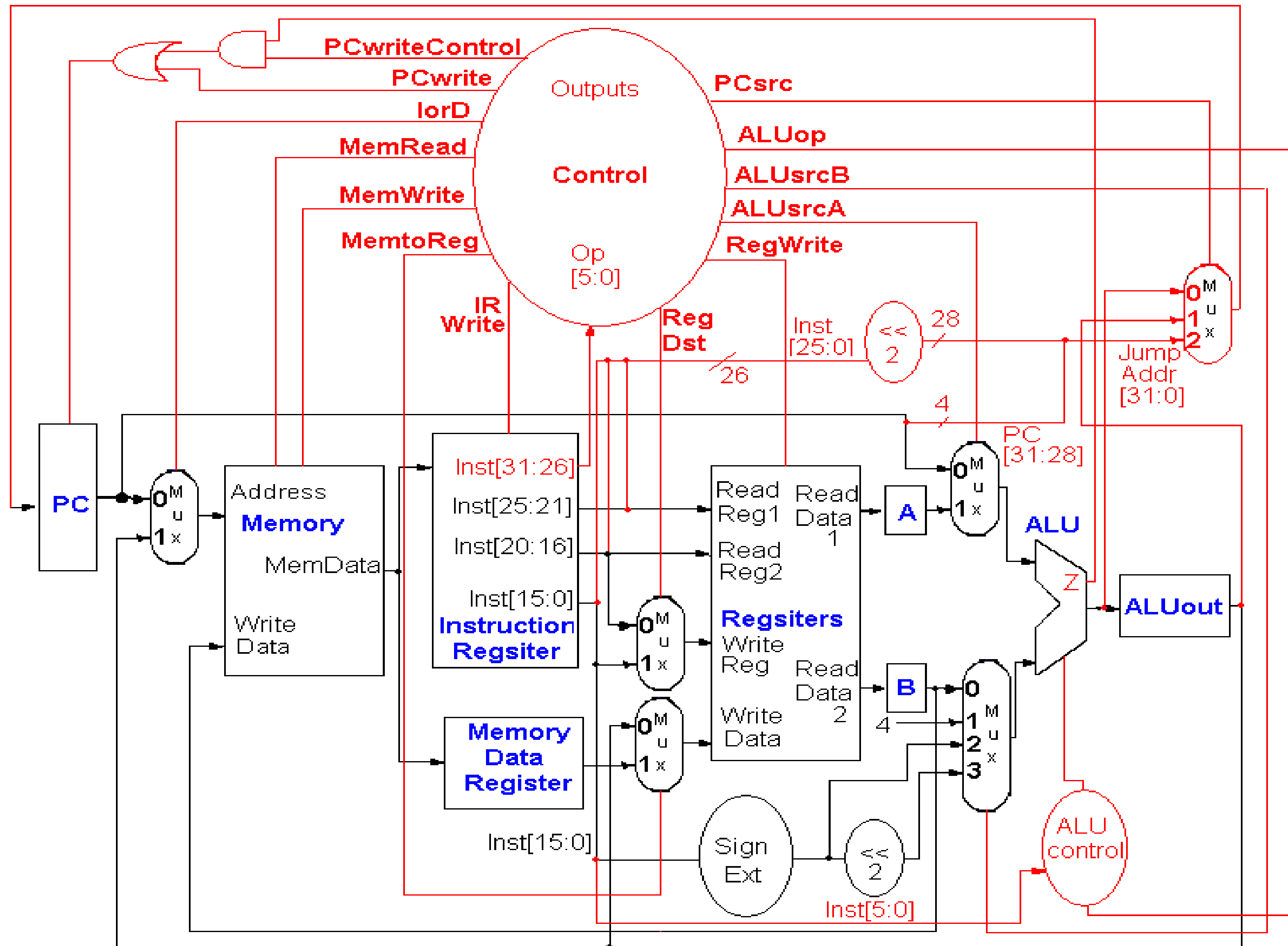
Control Signal Explanation (Tables. P.324)



Action of Control Signals (1-bit control signals)



Action of Control Signals (2-bit signals)



Instructions from ISA perspective – Clock Cycle

- What should happen in each clock cycle of the multicycle execution?
- Consider each instruction from perspective of ISA.
- Example:
  - The add instruction changes a register.
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC.
  - New value is the sum (“op”) of two registers.
  - Registers specified by bits 25:21 and 20:16 of the instruction

```
Reg[Memory[PC][15:11]] <=
Reg[Memory[PC][25:21]] op
                     Reg[Memory[PC][20:16]]
```

  - In order to accomplish this we must break up the instruction.
    (kind of like introducing variables when programming)

Breaking down an instruction

- ISA definition of arithmetic:

```
Reg[Memory[PC][15:11]] <= Reg[Memory[PC][25:21]]  op
                              Reg[Memory[PC][20:16]]
```

- Could break down to:
  - `IR <= Memory[PC]`
  - `A <= Reg[IR[25:21]]`
  - `B <= Reg[IR[20:16]]`
  - `ALUOut <= A op B`
  - `Reg[IR[20:16]] <= ALUOut`

- We forgot an important part of the definition of arithmetic!
  - `PC <= PC + 4`

Idea behind multicycle approach

- We define each instruction from the ISA perspective
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)

- Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)

- Finally try and pack as much work into each step
  (avoid unnecessary cycles)
  while also trying to share steps where possible
  (minimizes control, helps to simplify solution)

- Result: Our book's multicycle Implementation!

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;`

Which instruction needs this?

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) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
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		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

Simple Questions

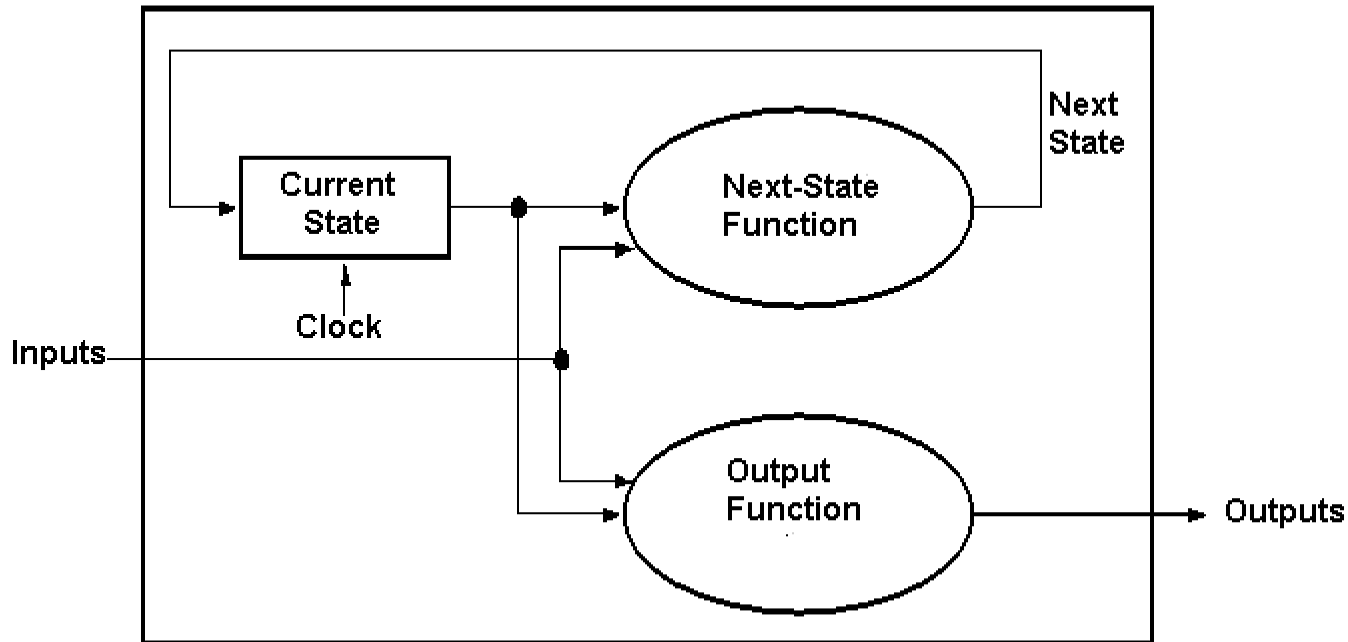
- Loads (5), Stores(4), ALU instructions (4), Branches(3), Jumps(3)
- How many cycles will it take to execute this code?

```
        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?

Specification of the Multicycle Control – Finite State Machine (FSM) approach

- Finite state machines-A Sequential Logic Function
  - 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)
  - If the output function can depend on both the current state and the current input, the machine is called a *Mealy machine*.

FSM Example - controlling a traffic light

- Our example concerns the control of a traffic light at an intersection of a north-south route and an east-west route. For simplicity, we will consider only the green and red lights. We want the lights to cycle no faster than 30 seconds in each direction, so we will use a 0.033 Hz clock so that the machine cycles between states at no faster than once every 30 seconds.
- There are two output signals:
  - ***NSlite****:* When this signal is asserted, the light on the north-south road is green; when this signal is deasserted the light on the north-south road is red.
  - ***EWlite****:* When this signal is asserted, the light on the east-west road is green; when this signal is deasserted the light on the east-west road is red.
- There are two inputs: NScar and EWcar.
  - ***NScar****:* Indicates that a car is over the detector placed in the roadbed in front of the light on the north-south road (going north or south).
  - ***EWcar****:* Indicates that a car is over the detector placed in the roadbed in front of the light on the east-west road (going east or west).

FSM Example - continued

- The traffic light should change from one direction to the other only if a car is waiting to go in the other direction; otherwise, the light should continue to show green in the same direction as the last car that crossed the intersection.
- To implement this simple traffic light we need two states:
  - ***NSgreen**:* The traffic light is green in the north-south direction.
  - ***EWgreen**:* The traffic light is green in the east-west direction.

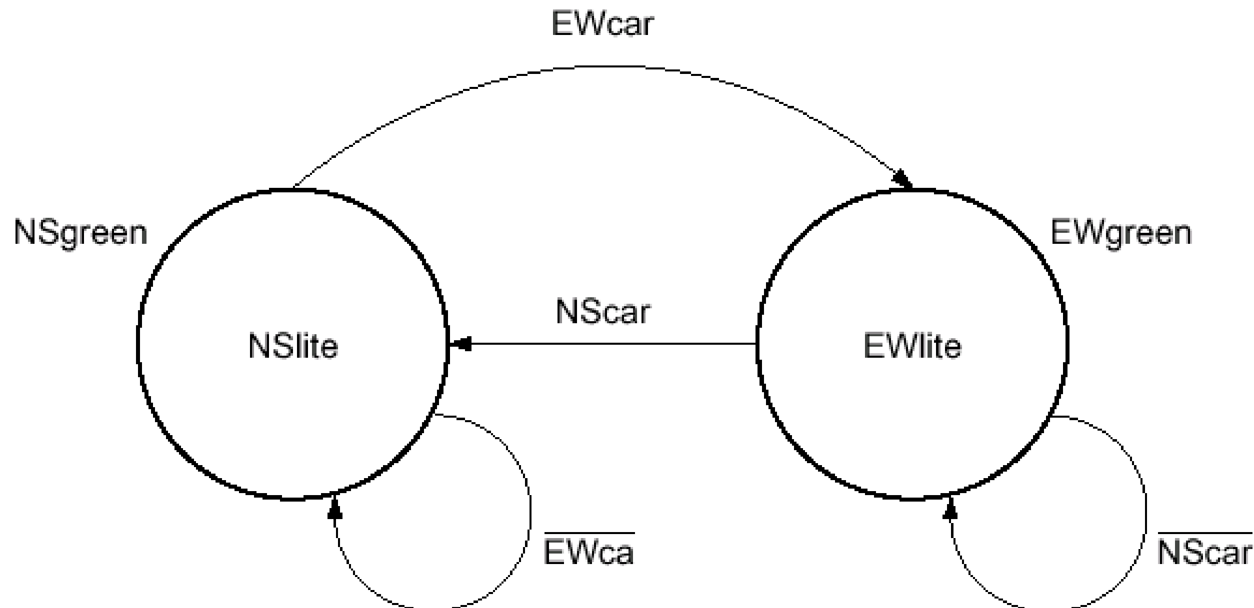
Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

FSM Example - Continued

- the output function:

Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

- Graphical Representation



FSM Example – Verilog version

The next-state function would be given as

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

where CurrentState is the contents of the state register (0 or 1) and NextState is the output of the next-state function that will be written into the state register at the end of the clock cycle.

The output function is also simple:

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

```
module TrafficLite (EWCar,NSCar,EWLite,NSLite,clock);
   input EWCar, NSCar,clock;
output EWLite,NSLite;
reg state;
initial state=0;  //set initial state
//following two assignments set the output, which is based only on the state
variable
assign NSLite = ~ state; //NSLite on if state = 0;
assign EWLite = state; //EWLite on if state =1
always @(posedge clock) // all state updates on a positive clock edge
   case (state)
      0: state = EWCar; //change state only if EWCar
      1: state = NSCar; //change state only if NSCar
   endcase
endmodule
```

Review: finite state machines

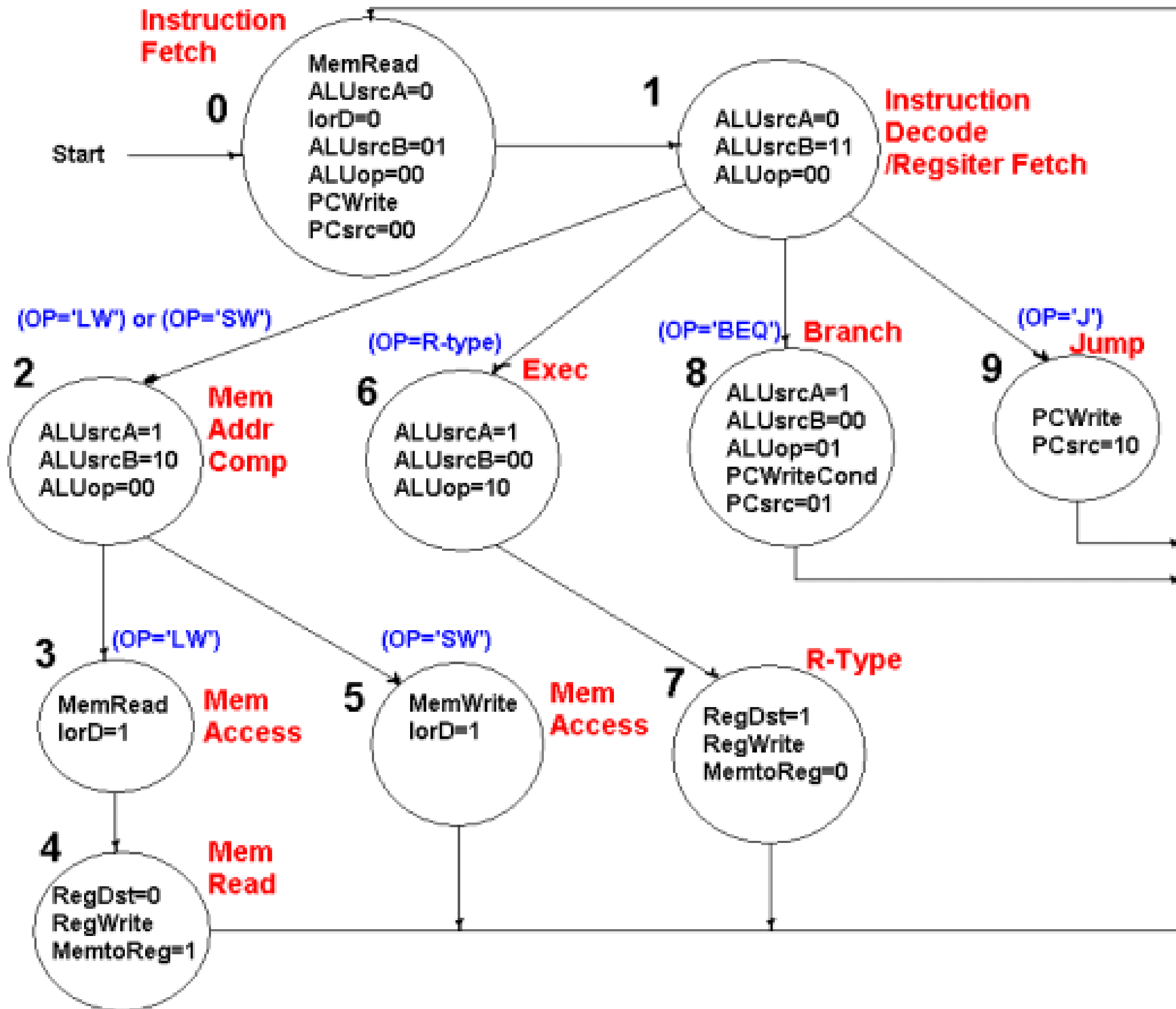
- Example:

  Appendix B. 37 *A friend would like you to build an "electronic eye" for use as a fake security device. The device consists of three lights lined up in a row, controlled by the outputs Left, Middle, and Right, which, if asserted, indicate that a light should be on. Only one light is on at a time, and the light "moves" from left to right and then from right to left, thus scaring away thieves who believe that the device is monitoring their activity. Draw the graphical representation for the finite state machine used to specify the electronic eye. Note that the rate of the eye's movement will be controlled by the clock speed (which should not be too great) and that there are essentially no inputs.*

FSM - Implementing the Control

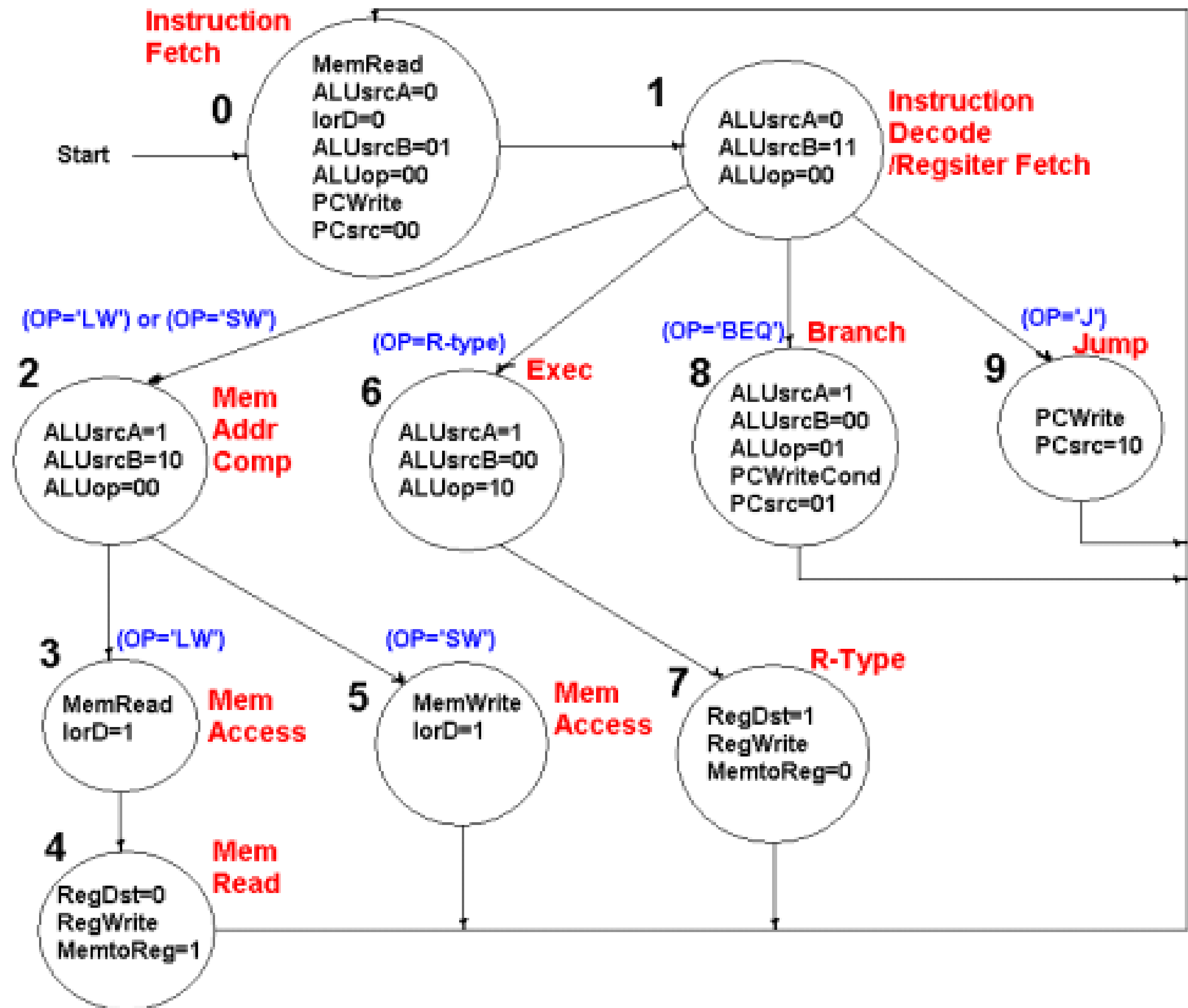
- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived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

Graphical Specification of FSM



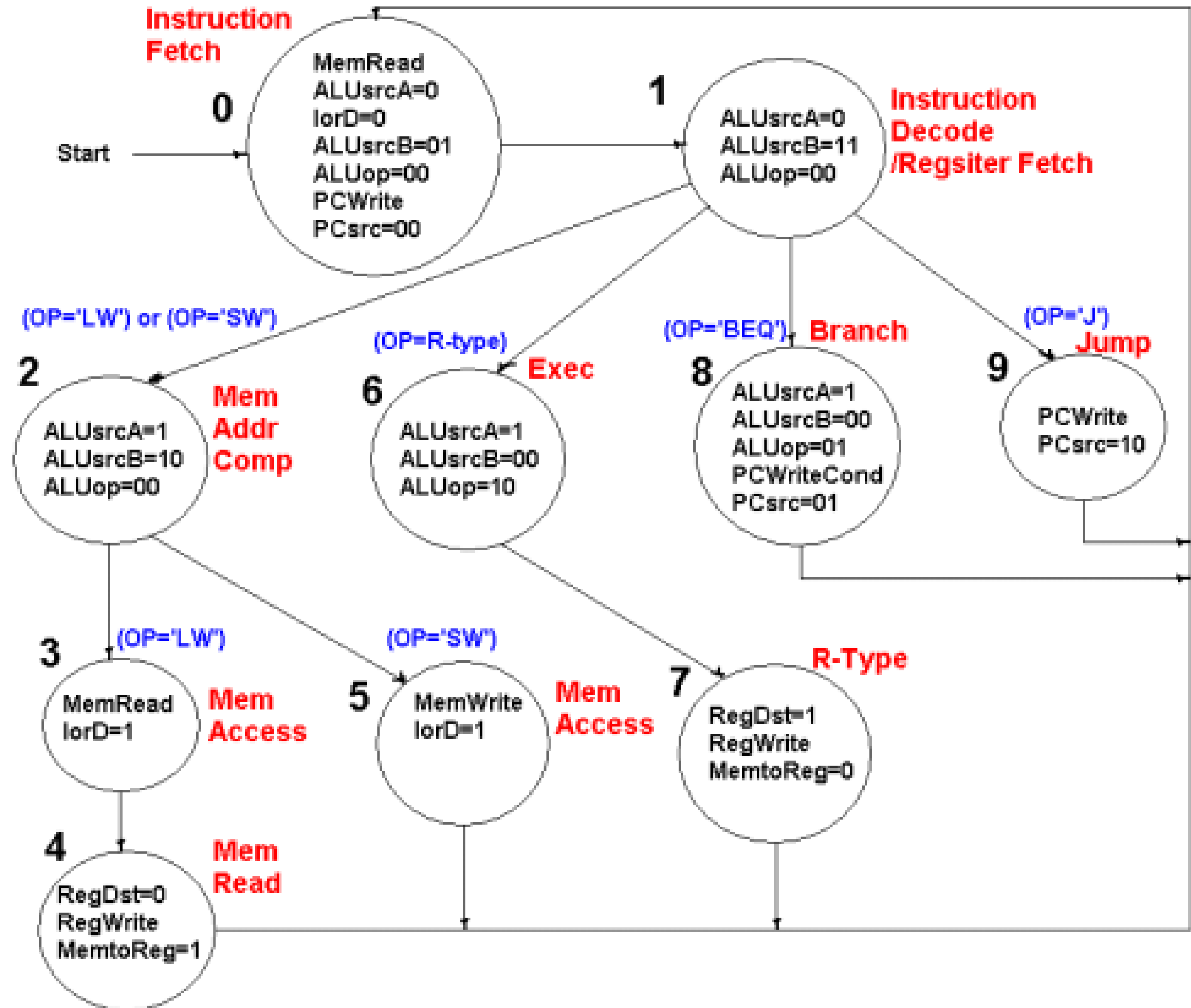
Graphical Specification of FSM-Instruction Fetch and Decode

- State 0
- State 1
- 4 Classes of instruction
  - Mem ref
  - R-type
  - Beq
  - Jump



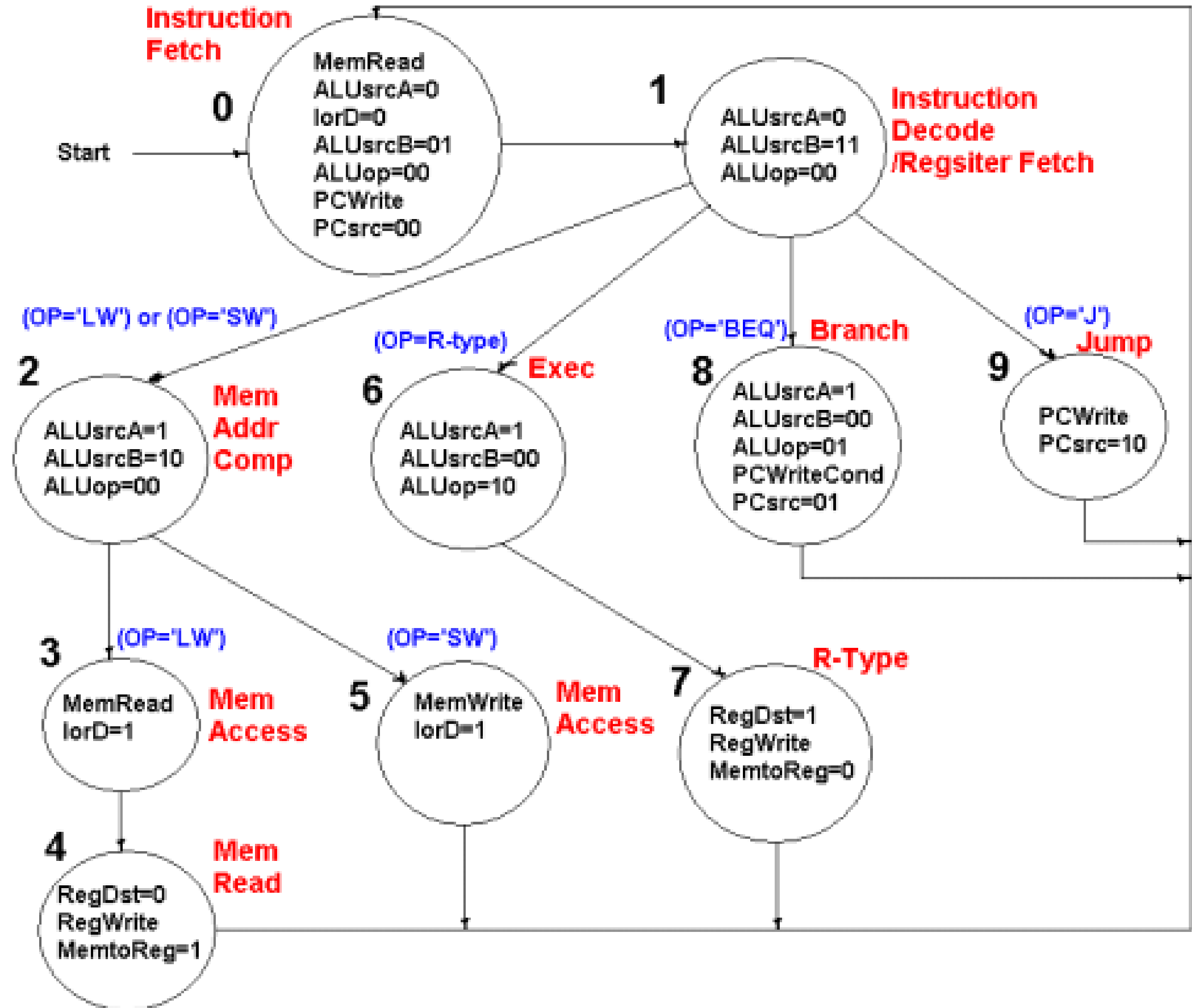
Graphical Specification of FSM-Mem Reference Instructions

- 4 States
- State 2
  - Compute Memory Address
  - In1: A Reg
  - In2: Sign-Ext
  - Out: ALUout
- State 3
  - LW
  - Mem Addr from ALU
  - State 4
    - **Write to Reg**
- State 5
  - SW
  - Mem Addr from ALU



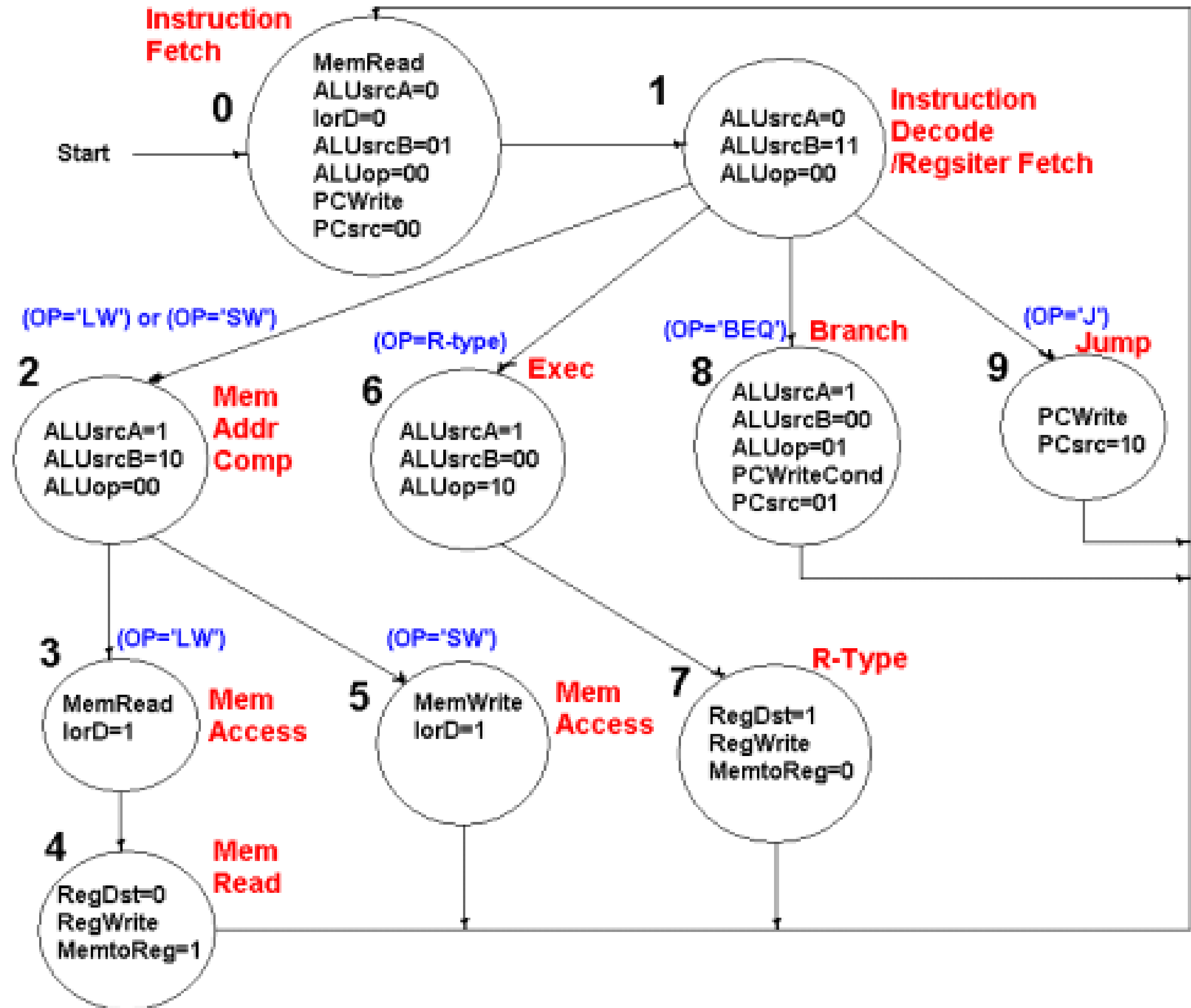
Graphical Specification of FSM- R-type Instructions

- 2 States
- State 6
  - In: 2 Regs
- State 7
  - Register file to write
  - Rd as destination
  - ALUout is the source of the value to write into the register file



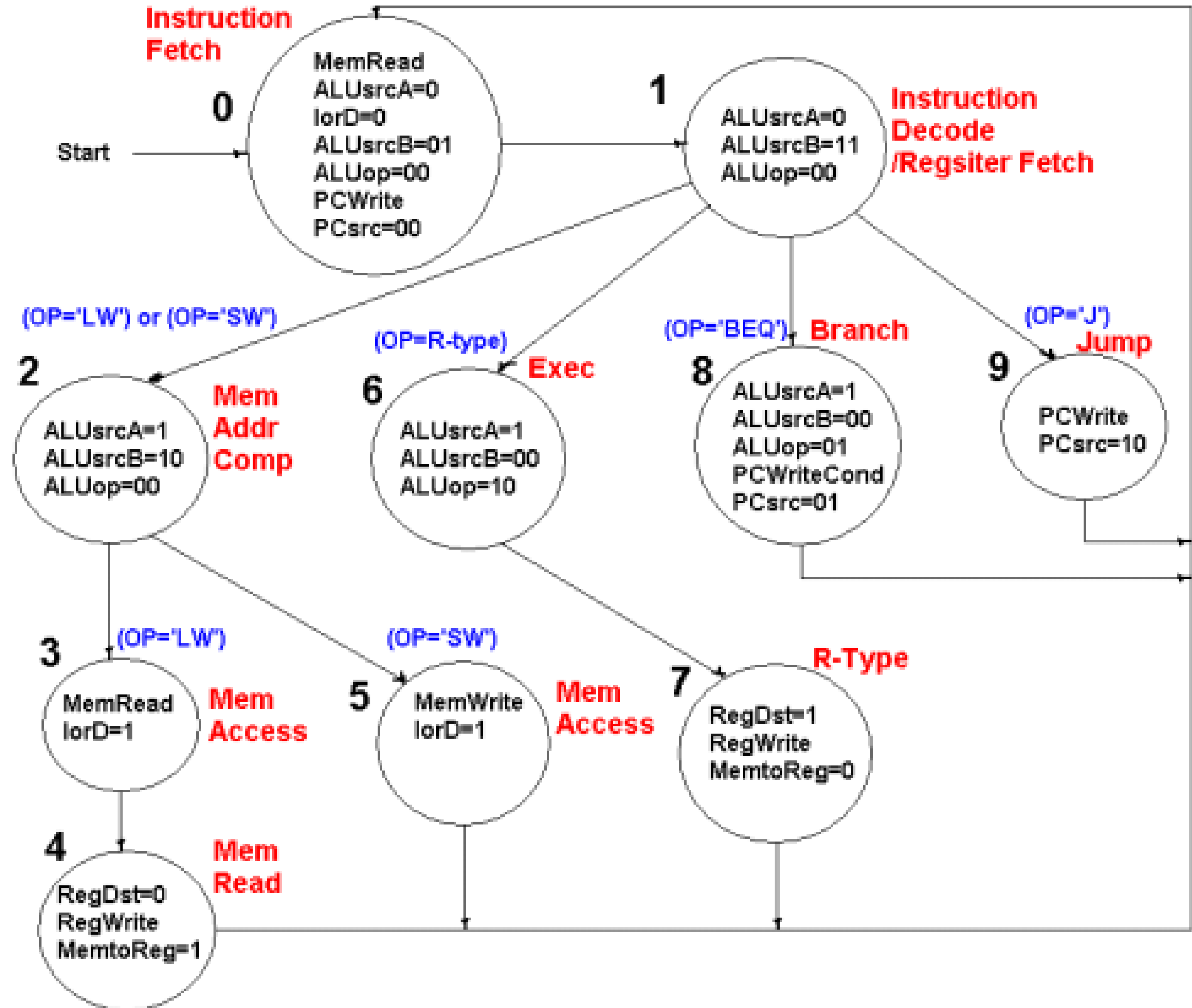
Graphical Specification of FSM- beq Instructions

- 1 State
- State 8
  - Compare reg A and reg B (read Zero output of ALU)
  - PC set by the the ALUout

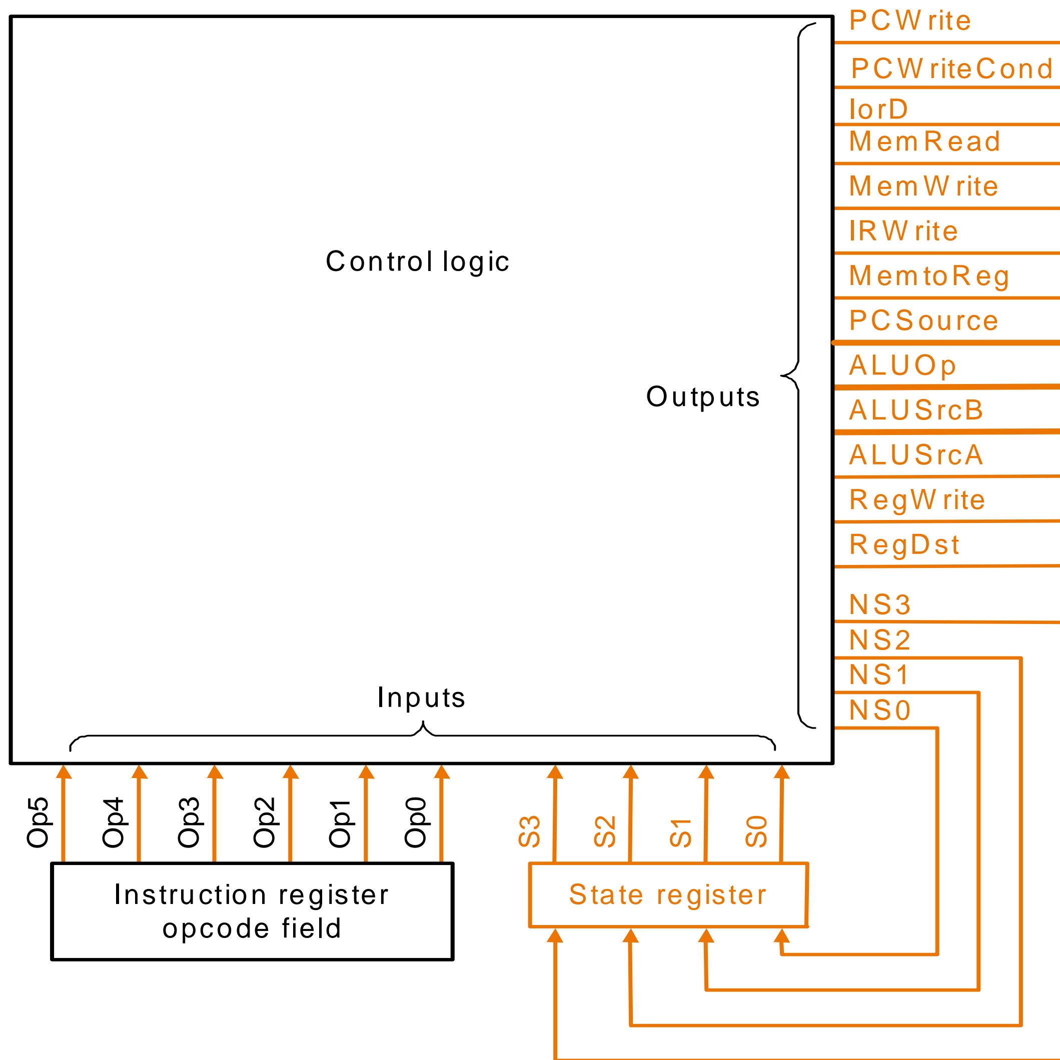


Graphical Specification of FSM- Jump Instruction

- 1 State
- State 9
  - Lower 26 bits of IR
  - 00 for lower order bits
  - Concatenated with upper 4bits of PC



Finite State Machine Control – Moore Machine



state 6 0110_{two}

$$\overline{S3} \cdot S2 \cdot S1 \cdot \overline{S0}$$

Logic Equations

Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	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')

$$\text{NextState1} = \text{State0} = \overline{S3}\cdot\overline{S2}\cdot\overline{S1}\cdot\overline{S0}$$

$$\begin{aligned}\text{NextState3} &= \text{State2}\cdot(\text{Op}[5\text{-}0] = \text{lw})\\ &= \overline{S3}\cdot\overline{S2}\cdot S1\cdot\overline{S0}\cdot Op5\cdot\overline{Op4}\cdot\overline{Op3}\cdot\overline{Op2}\cdot Op1\cdot Op0\end{aligned}$$

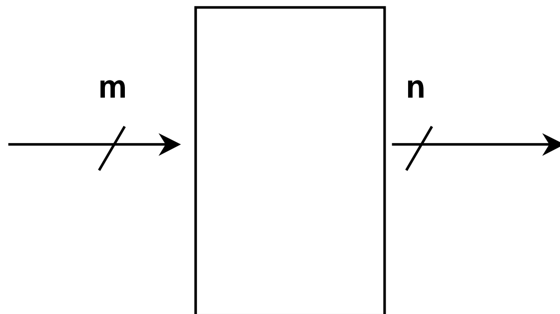
$$\begin{aligned}\text{NextState5} &= \text{State 2}\cdot(\text{Op}[5\text{-}0] = \text{sw})\\ &= \overline{S3}\cdot\overline{S2}\cdot S1\cdot\overline{S0}\cdot Op5\cdot\overline{Op4}\cdot Op3\cdot\overline{Op2}\cdot Op1\cdot Op0\end{aligned}$$

$$\text{NextState7} = \text{State6} = \overline{S3}\cdot S2\cdot S1\cdot\overline{S0}$$

$$\begin{aligned}\text{NextState9} &= \text{State1}\cdot(\text{Op}[5\text{-}0] = \text{jmp})\\ &= \overline{S3}\cdot\overline{S2}\cdot\overline{S1}\cdot S0\cdot\overline{Op5}\cdot\overline{Op4}\cdot\overline{Op3}\cdot\overline{Op2}\cdot Op1\cdot\overline{Op0}\end{aligned}$$

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.
  - our outputs are the bits of data that the address points to.



0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

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

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} x 20 = 20K bits (and a rather unusual size)
- Rather wasteful, since for lots of the entries, the outputs are the same
  — i.e., opcode is often ignored

Truth Tables for Control Signals

s3	s2	s1	s0
0	0	0	0
1	0	0	1

a. Truth table for PCWrite

s3	s2	s1	s0
1	0	0	0

b. Truth table for PCWriteCond

s3	s2	s1	s0
0	0	1	1
0	1	0	1

c. Truth table for IorD

s3	s2	s1	s0
0	0	0	0
0	0	1	1

d. Truth table for MemRead

s3	s2	s1	s0
0	1	0	1

e. Truth table for MemWrite

s3	s2	s1	s0
0	0	0	0

f. Truth table for IRWrite

s3	s2	s1	s0
0	1	0	0

g. Truth table for MemtoReg

s3	s2	s1	s0
1	0	0	1

h. Truth table for PCSource1

s3	s2	s1	s0
1	0	0	0

i. Truth table for PCSource0

s3	s2	s1	s0
0	1	1	0

j. Truth table for ALUOp1

s3	s2	s1	s0
1	0	0	0

k. Truth table for ALUOp0

s3	s2	s1	s0
0	0	0	1
0	0	1	0

l. Truth table for ALUSrcB1

s3	s2	s1	s0
0	0	0	0
0	0	0	1

m. Truth table for ALUSrcB0

s3	s2	s1	s0
0	0	1	0
0	1	1	0
1	0	0	0

n. Truth table for ALUSrcA

s3	s2	s1	s0
0	1	0	0
0	1	1	1

o. Truth table for RegWrite

s3	s2	s1	s0
0	1	1	1

p. Truth table for RegDst

Truth Tables for Next-States

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	0	1

a. The truth table for the NS3 output, active when the next state is 8 or 9. This signal is activated when the current state is 1.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	0	1	1
X	X	X	X	X	X	0	1	1	0

b. The truth table for the NS2 output, which is active when the next state is 4, 5, 6, or 7. This situation occurs when the current state is one of 1, 2, 3, or 6.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	0	1
1	0	1	0	1	1	0	0	0	1
1	0	0	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0

c. The truth table for the NS1 output, which is active when the next state is 2, 3, 6, or 7. The next state is one of 2, 3, 6, or 7 only if the current state is one of 1, 2, or 6.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
X	X	X	X	X	X	0	0	0	0
1	0	0	0	1	1	0	0	1	0
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0
0	0	0	0	1	0	0	0	0	1

d. The truth table for the NS0 output, which is active when the next state is 1, 3, 5, 7, or 9. This happens only if the current state is one of 0, 1, 2, or 6.

Truth Table for 16 Control Signals

Outputs	Input values (S[3–0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

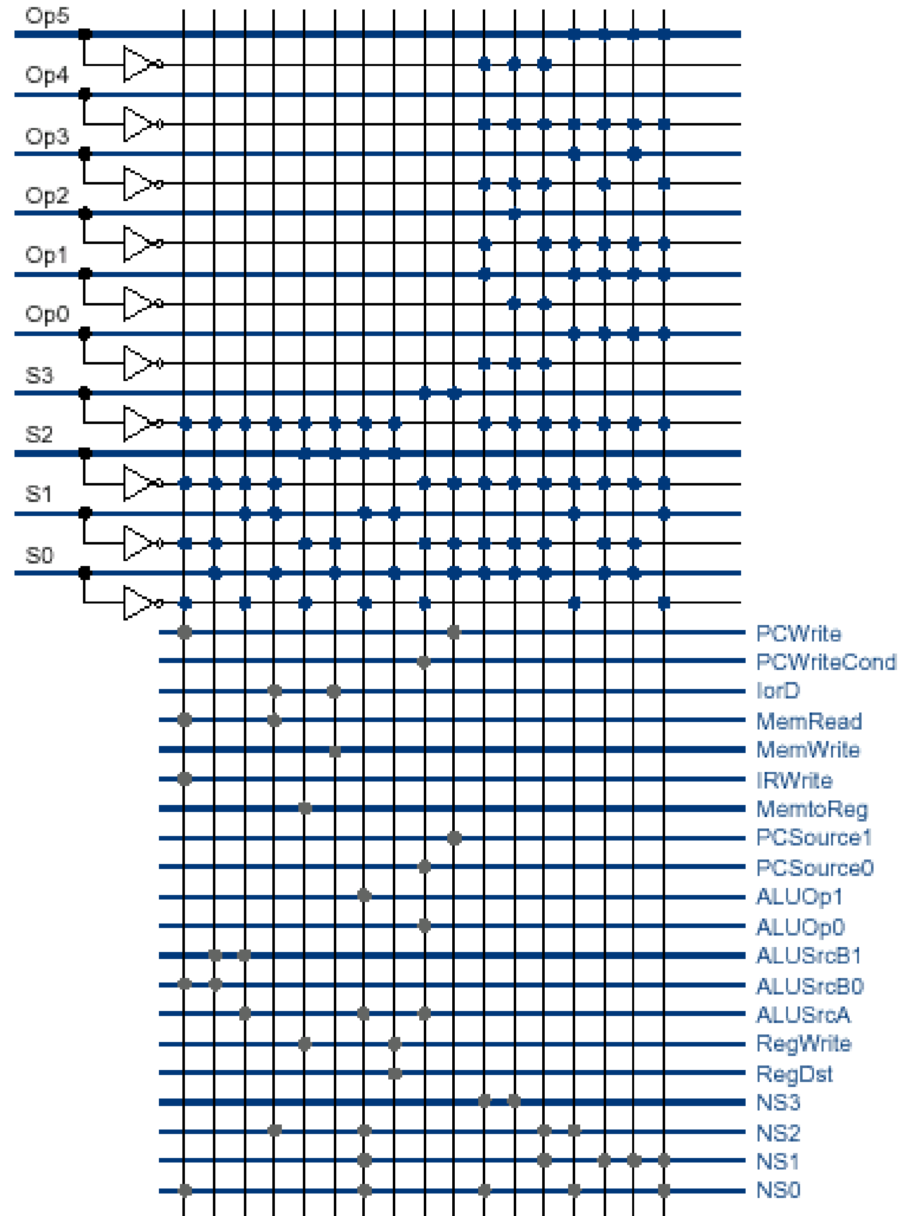
Tables for ROM implementation

Lower 4 bits of the address	Bits 19–4 of the word
0000	1001010000001000
0001	0000000000011000
0010	0000000000010100
0011	0011000000000000
0100	0000001000000010
0101	0010100000000000
0110	0000000001000100
0111	0000000000000011
1000	0100000010100100
1001	1000000100000000

	Op [5–0]					
Current state S[3–0]	000000 (R-format)	000010 (jmp)	000100 (beq)	100011 (lw)	101011 (sw)	Any other value
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	illegal
0010	XXXX	XXXX	XXXX	0011	0101	illegal
0011	0100	0100	0100	0100	0100	illegal
0100	0000	0000	0000	0000	0000	illegal
0101	0000	0000	0000	0000	0000	illegal
0110	0111	0111	0111	0111	0111	illegal
0111	0000	0000	0000	0000	0000	illegal
1000	0000	0000	0000	0000	0000	illegal
1001	0000	0000	0000	0000	0000	illegal

PLA Implementation

- 17 unique Minterms
  - 10 depends only on current states
  - 7 on combination of O field and current-state bits
- Total Size of PLA
  - (#inx#minterm) + (#outx#minterm) = (10x17)+(20x17) = 51(



ROM vs PLA

- Break up the table into two parts
  - — 4 state bits tell you the 16 outputs, 2^4 x 16 bits of ROM
  - — 10 bits tell you the 4 next state bits, 2^{10} x 4 bits of ROM
  - — Total: 4.3K bits of ROM
- PLA is much smaller
  - — can share product terms
  - — only need entries that produce an active output
  - — can take into account don't cares
- Size is (#inputs $\times$ #product-terms) + (#outputs $\times$ #product-terms)

  For this example = (10x17)+(20x17) = 510 PLA cells

- PLA cells usually about the size of a ROM cell (slightly bigger)