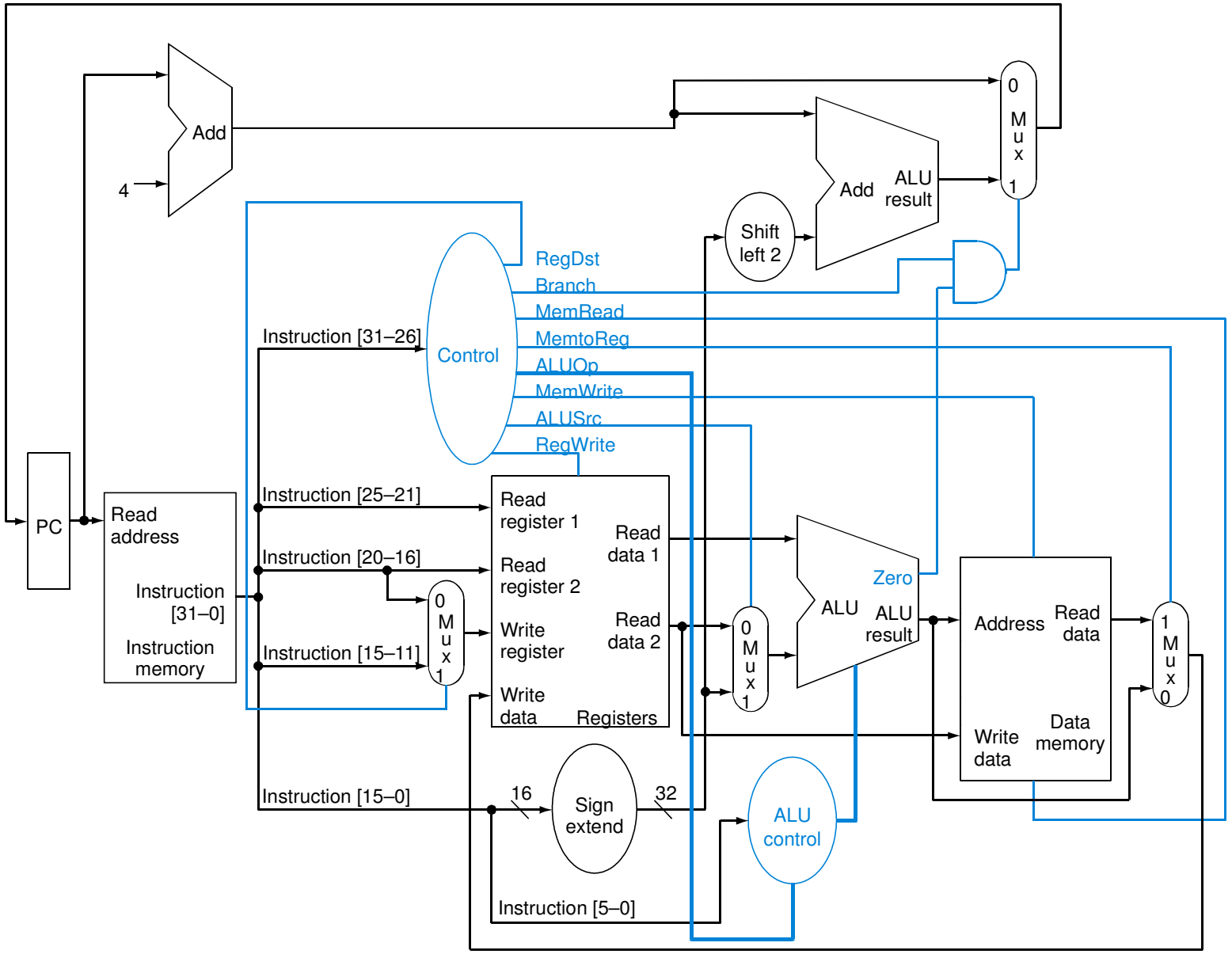

Add
4
0
M
u
x
1
Add
ALU
result
Shift
left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read
address
Instruction
[31–0]
Instruction
memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read
register 1
Read
register 2
Write
register
Write
data
Read
data 1
Read
data 2
Registers
ALU
Zero
ALU
result
Address
Read
data
Write
data
Data
memory
16
Sign
extend
32
ALU
control

Multi Cycle Datapath

Slides courtesy of Professor Tod Amon, Southern Utah University, with minor modifications by Nathan Sprague

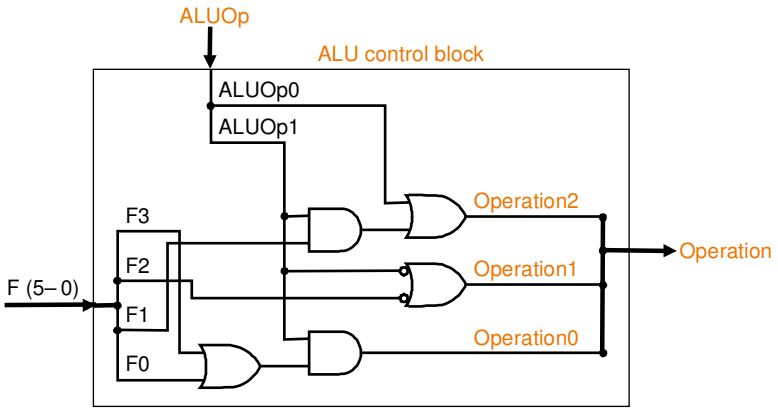
ALU control lines

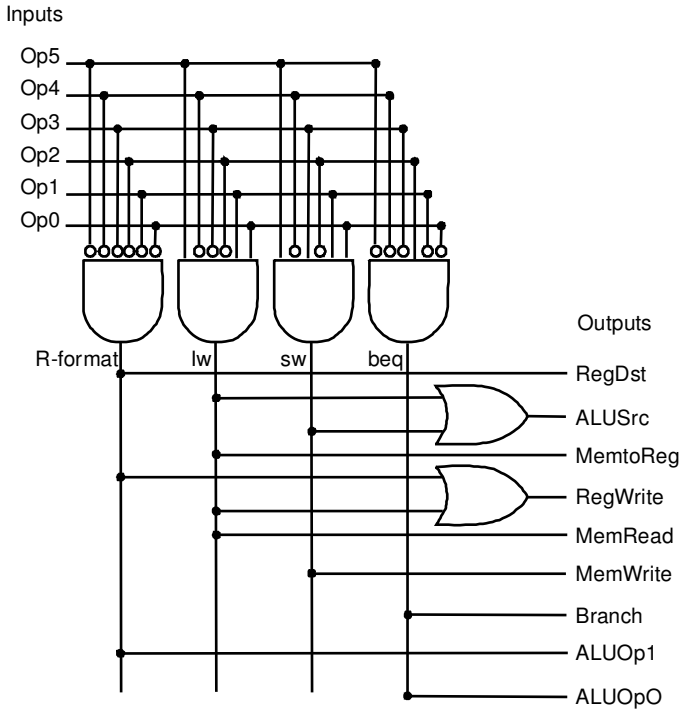
```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
1100 = NOR
```

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 5.13 The truth table for the three ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

Simple combinational logic



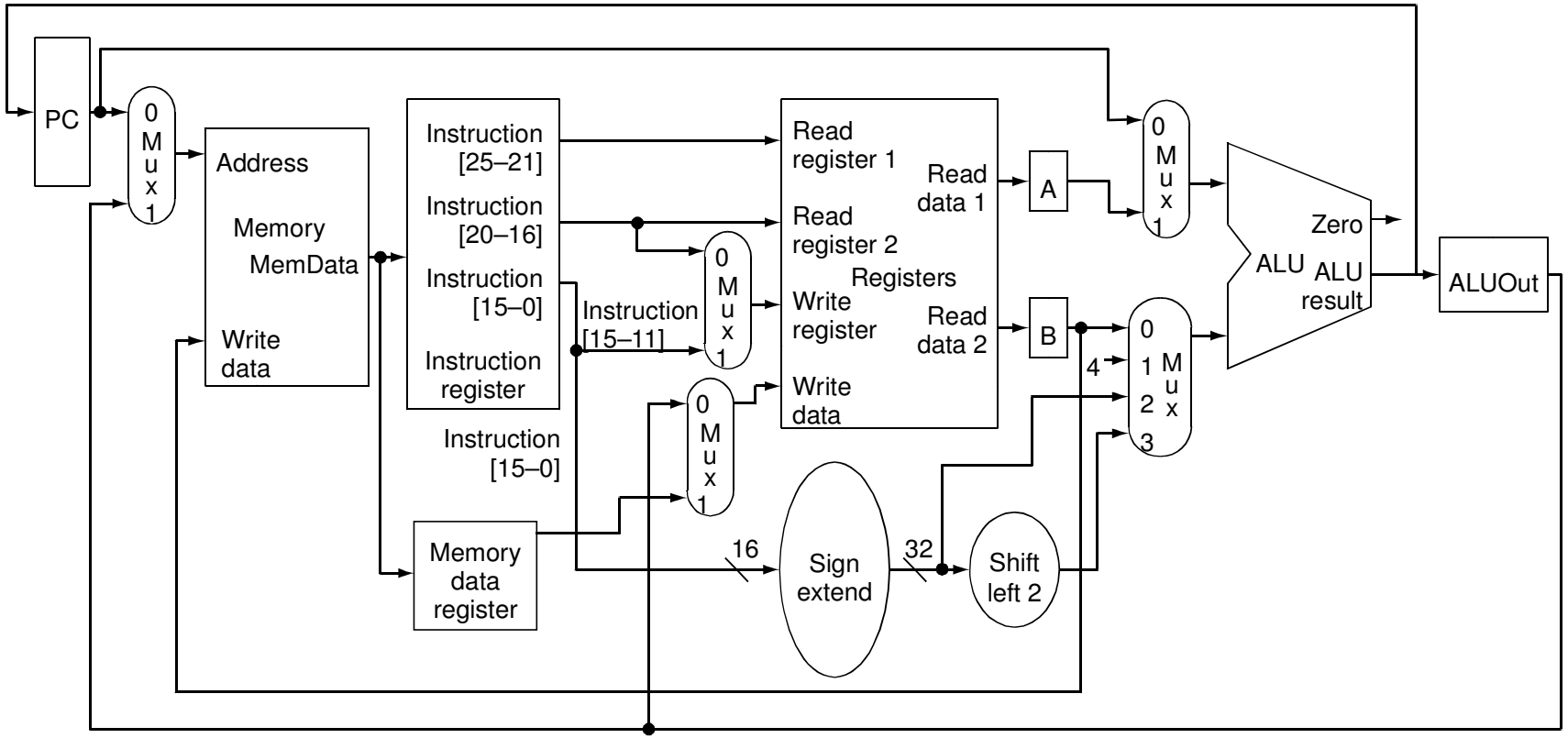


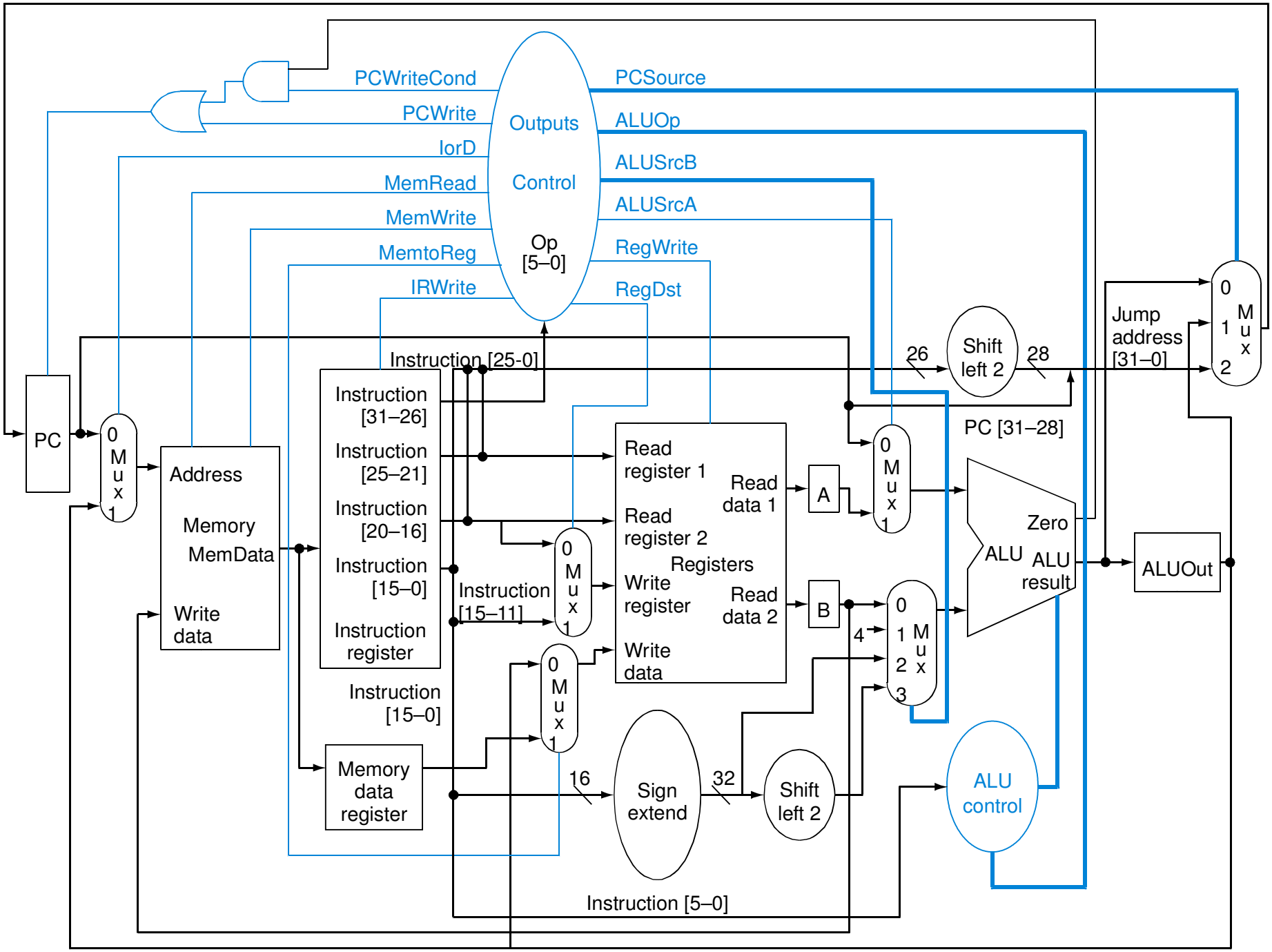
Control lines based on opcode

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Problems with single cycle implementation...

Mult-cycle design...




PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction [25-0]
Instruction
[15–11]
Instruction
[15–0]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
16
Sign
extend
32
Shift
left 2
26
Shift
left 2
28
PC [31–28]
Jump
address
[31–0]
0
1
2
3
Zero
ALU
ALU
result
ALUOut
ALU
control
Instruction [5–0]

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.

```
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

```
    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;
```

Write-back step

- `Reg[IR[20:16]] <= MDR;`

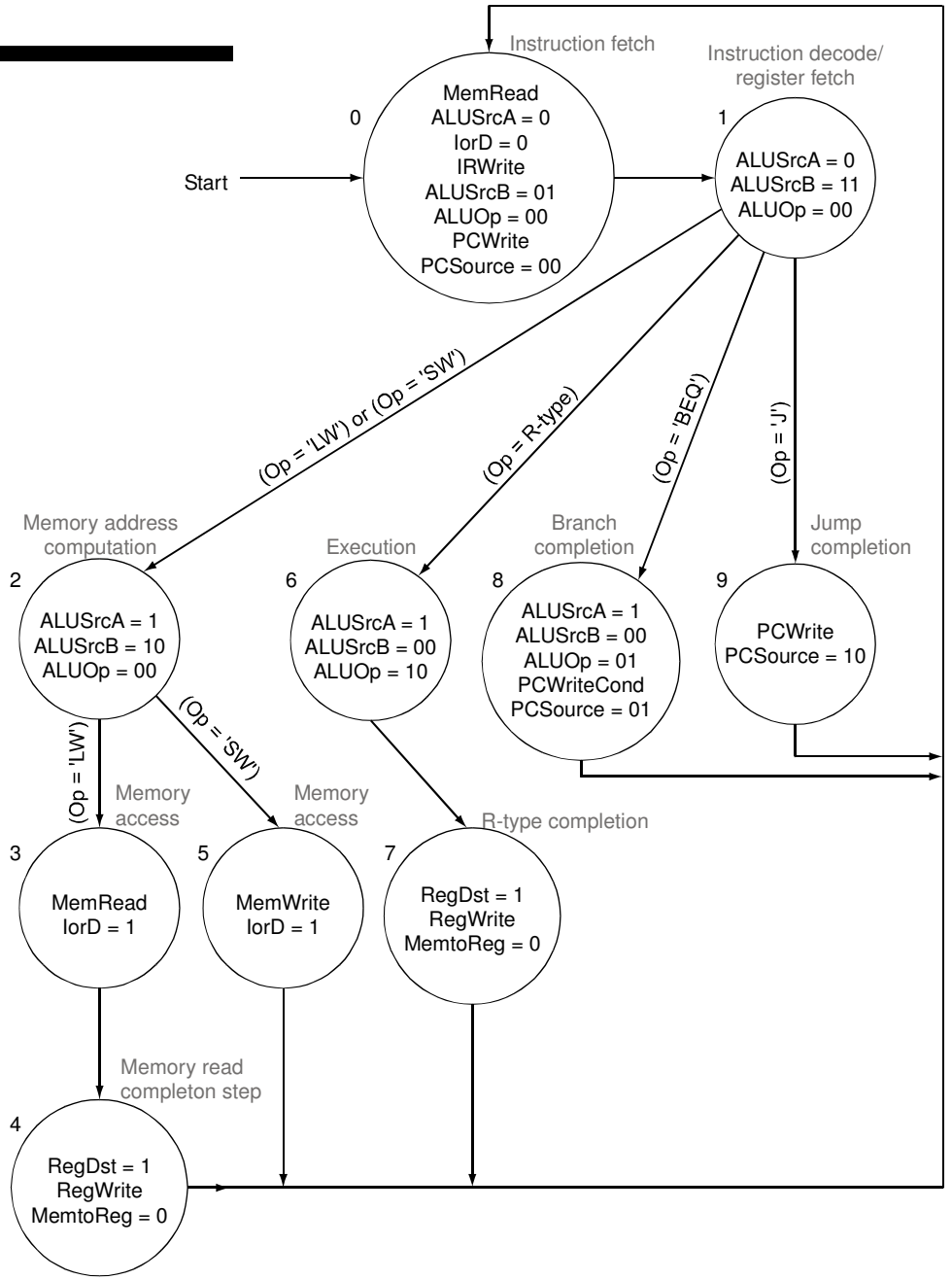
Which instruction needs this?

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

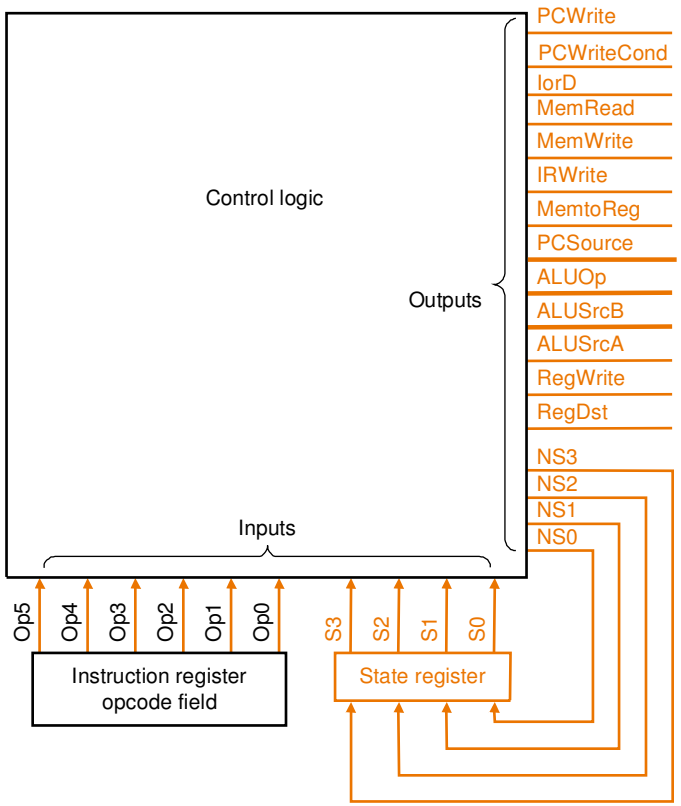
Graphical Specification of FSM

- Note:
  - don't care if not mentioned
  - asserted if name only
  - otherwise exact value
- How many state bits will we need?



Finite State Machine for Control

- Implementation:



PLA Implementation