# In-Class Exercise – add

- Write an RTL description for add that uses four short clock cycles
- Write a single cycle RTL description
- What components does the datapath for the instruction require? Remember, each component can only be used once per clock cycle.
- What are the inputs, outputs, and control signals for each of these components?
- What signals need to be connected to the inputs of each component?

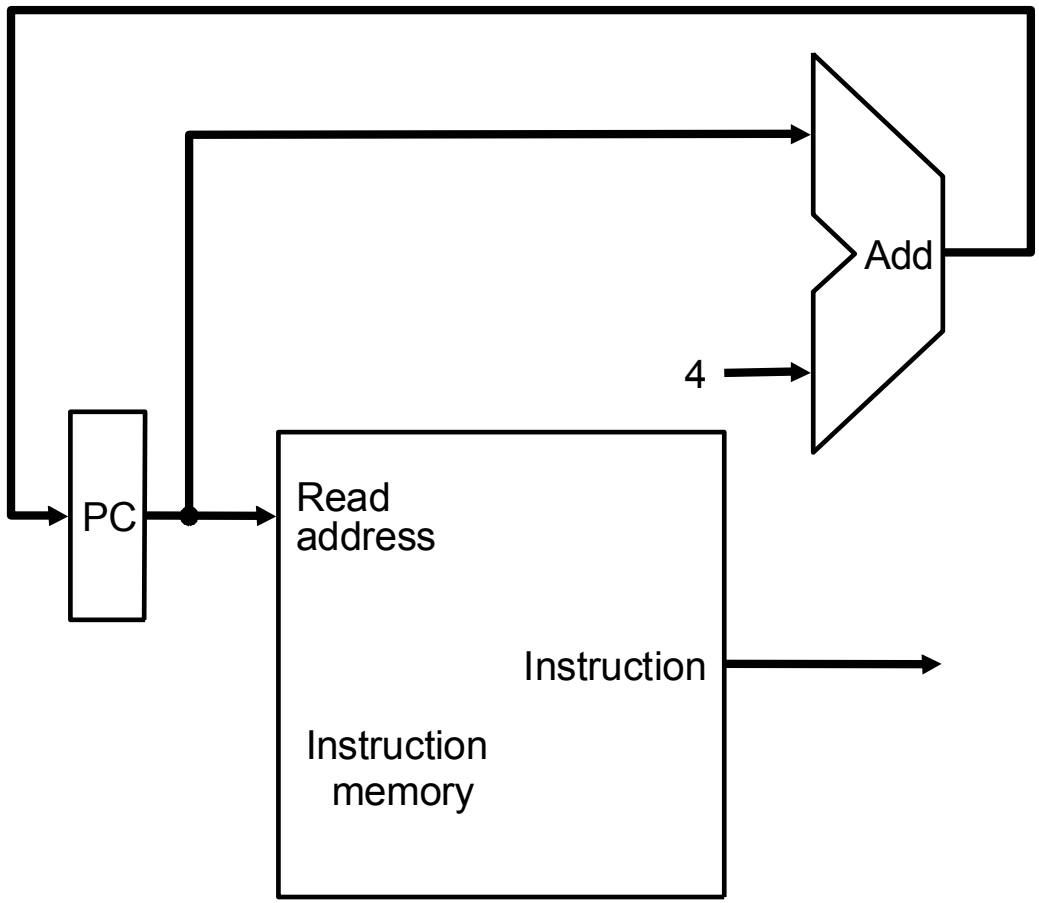
# R-type Instructions

```
if ((Mem[PC])[15-11] == 0 && (Mem[PC])[5-0] == 32) then
      PC = PC + 4
      Reg[(Mem[PC])[15-11]] =
          Reg[(Mem[PC])[25-21]] op Reg[(Mem[PC])[20-16]]
```

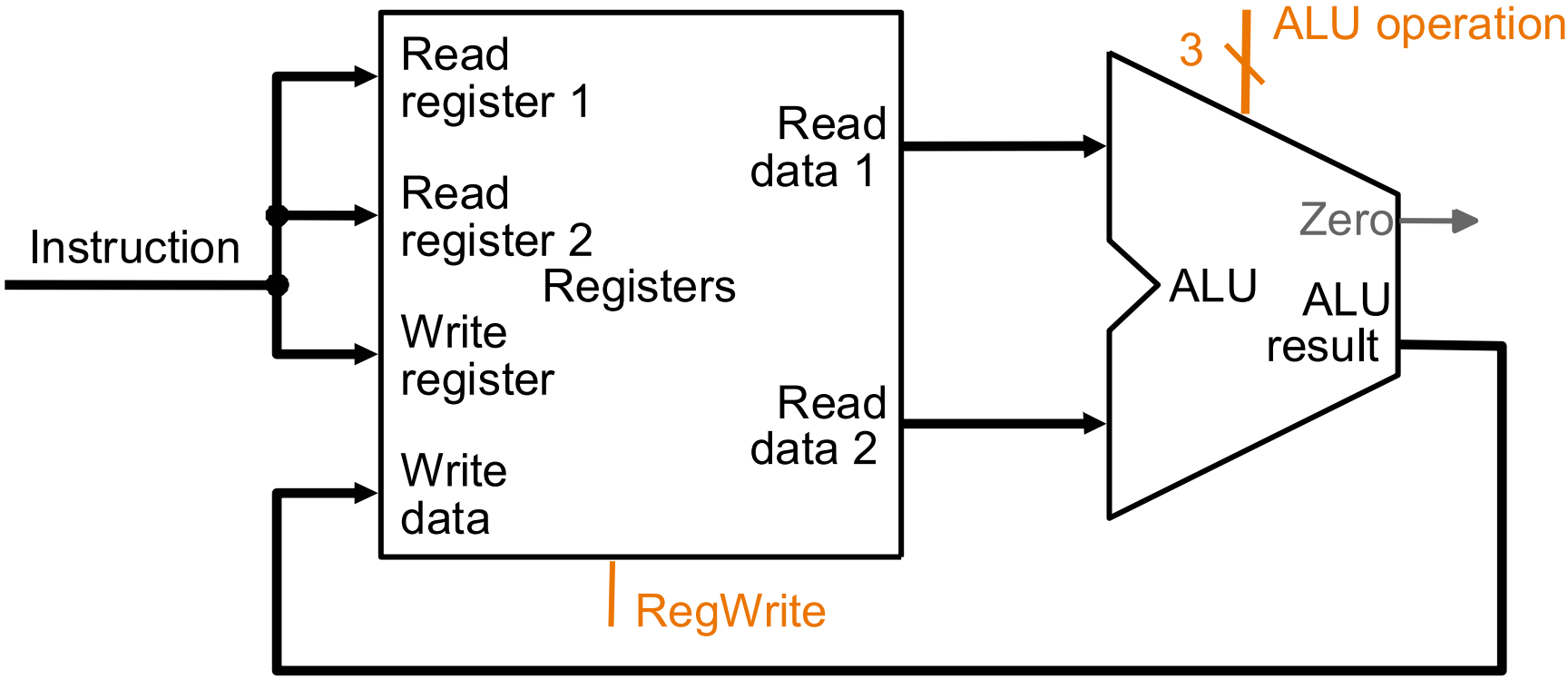
Components:

- For fetch and increment
  - PC – 32-bit register, written every cycle
  - Mem – 32-bit address input, 32-bit data output, read every cycle
  - Adder – two 32-bit inputs, one 32-bit output
- For execution
  - Reg – three 5-bit address inputs, one 32-bit data input, two 32-bit data outputs, read every cycle, only written during certain instructions (as we'll see later)
  - ALU – two 32-bit inputs, one 32-bit output, n-bit control

# Single-cycle datapath for fetch and increment PC



# Single-cycle datapath for R-type instructions (like *add*)



# In-Class Exercise – lw

- Write an RTL description for lw that uses five short clock cycles.
  - Use “sign-extend(16-bit quantity)” before adding the offset.
- Write a single cycle RTL description
- What components does the datapath for the instruction require?
- What are the inputs, outputs, and control signals for each of these components?
- What signals need to be connected to the inputs of each component?

# In-Class Exercise – sw

- Write an RTL description for sw that uses four short clock cycles
- Write a single cycle RTL description
- What components does the datapath for the instruction require?
- What are the inputs, outputs, and control signals for each of these components?
- What signals need to be connected to the inputs of each component?

# Memory-Reference Instructions

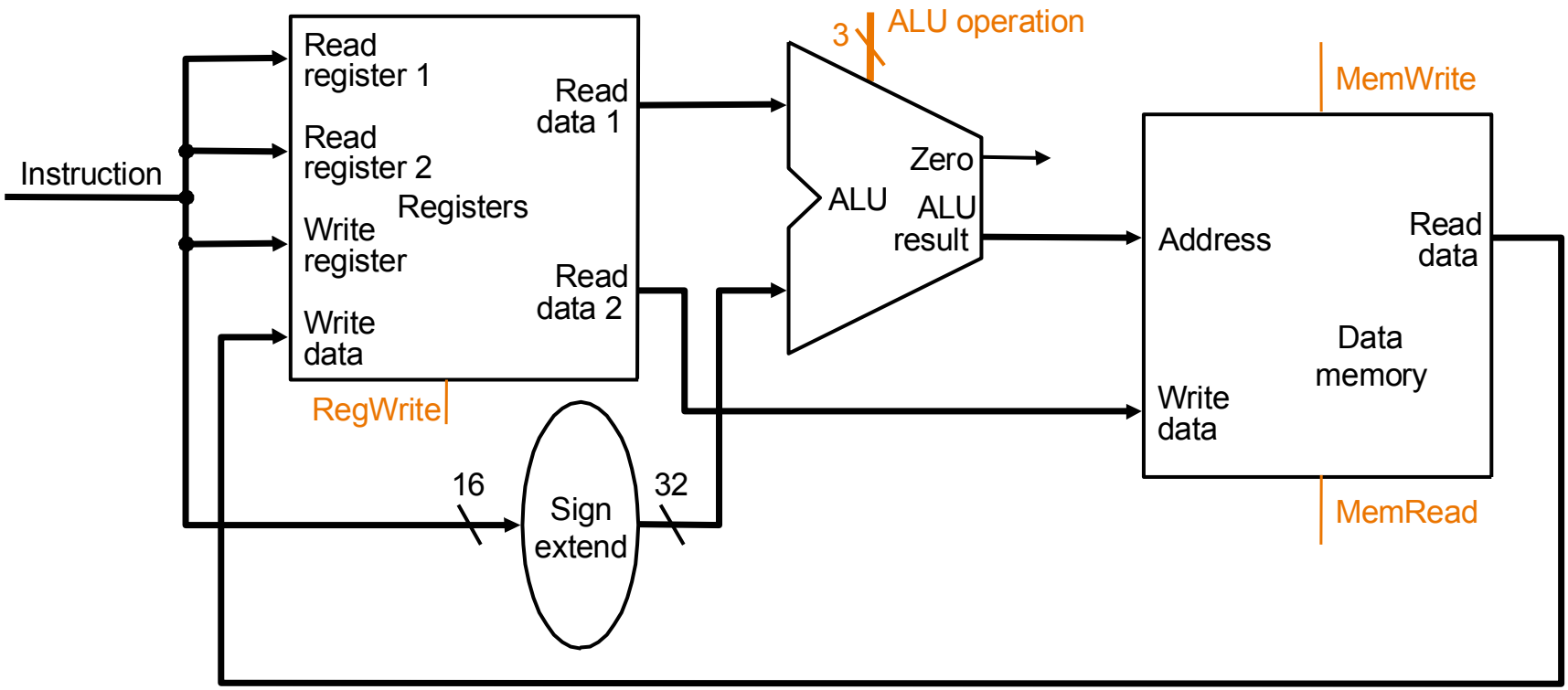
```
if ((Mem[PC])[31-26] == 35) then
        PC = PC + 4
        Reg[(Mem[PC])[15-11]] =
            Mem[Reg[(Mem[PC])[25-21]] + sign-extend((Mem[PC])[15-0])]

if ((Mem[PC])[31-26] == 43) then
        PC = PC + 4
        Mem[Reg[(Mem[PC])[25-21]] + sign-extend((Mem[PC])[15-0])] =
            Reg[(Mem[PC])[15-11]]
```

New/modified components:

- Mem – sometimes read twice per cycle (instruction and data), data written during some cycles
  - One solution: separate instruction and data memories (uggh)
- Sign-extend – 16-bit input, 32-bit output

# Single-cycle datapath for loads and stores



# In-Class Exercise – beq

- Write an RTL description for beq that uses three short clock cycles
  - Use “Zero(arg1 – arg2)” to compare arg1 and arg2
  - Use “<< 2” to shift left two bits
  - Branch by modifying the contents of PC
- Write a single cycle RTL description
- What components does the datapath for the instruction require?
- What are the inputs, outputs, and control signals for each of these components?
- What signals need to be connected to the inputs of each component?

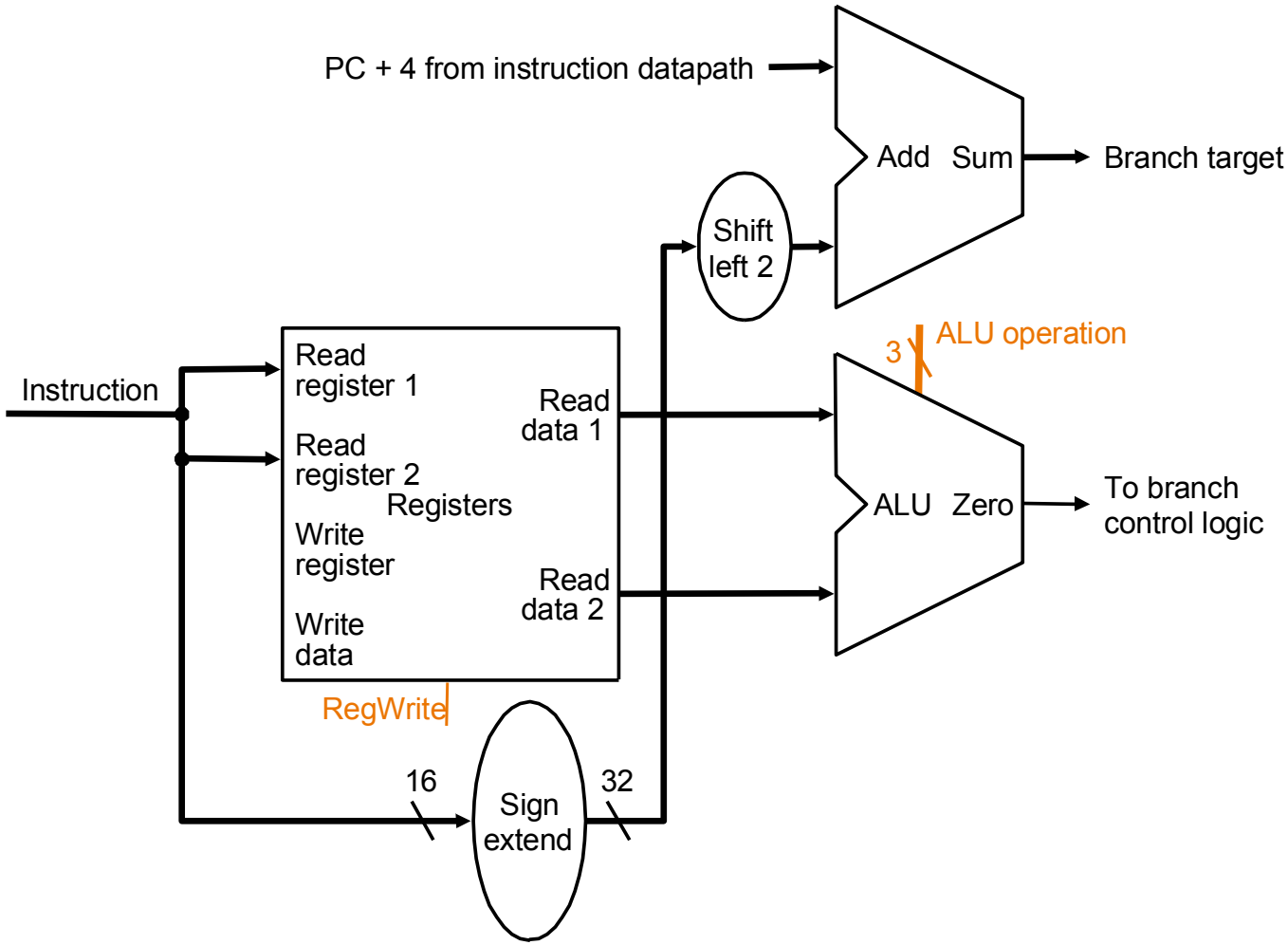
# beq instruction

```
if ((Mem[PC])[31-26] == 4) then
      if (Reg[(Mem[PC])[25-21]] == Reg[(Mem[PC])[20-16]]) then
         PC = PC + (sign-extend ((Mem[PC])[15-0]) << 2) + 4
      else
         PC = PC + 4
```

New/modified components:

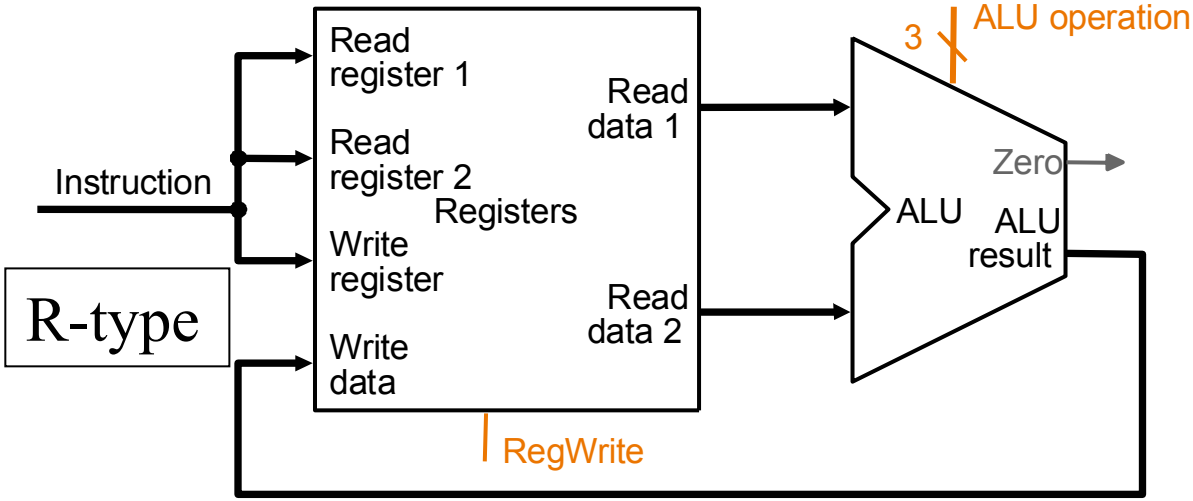
- Comparator– two 32-bit inputs, one 1-bit output (can be special function of ALU)
- Shifter – 32-bit input, 32-bit output, shifts by two every time
- Adder (for branch target address) – two 32-bit inputs, one 32-bit output

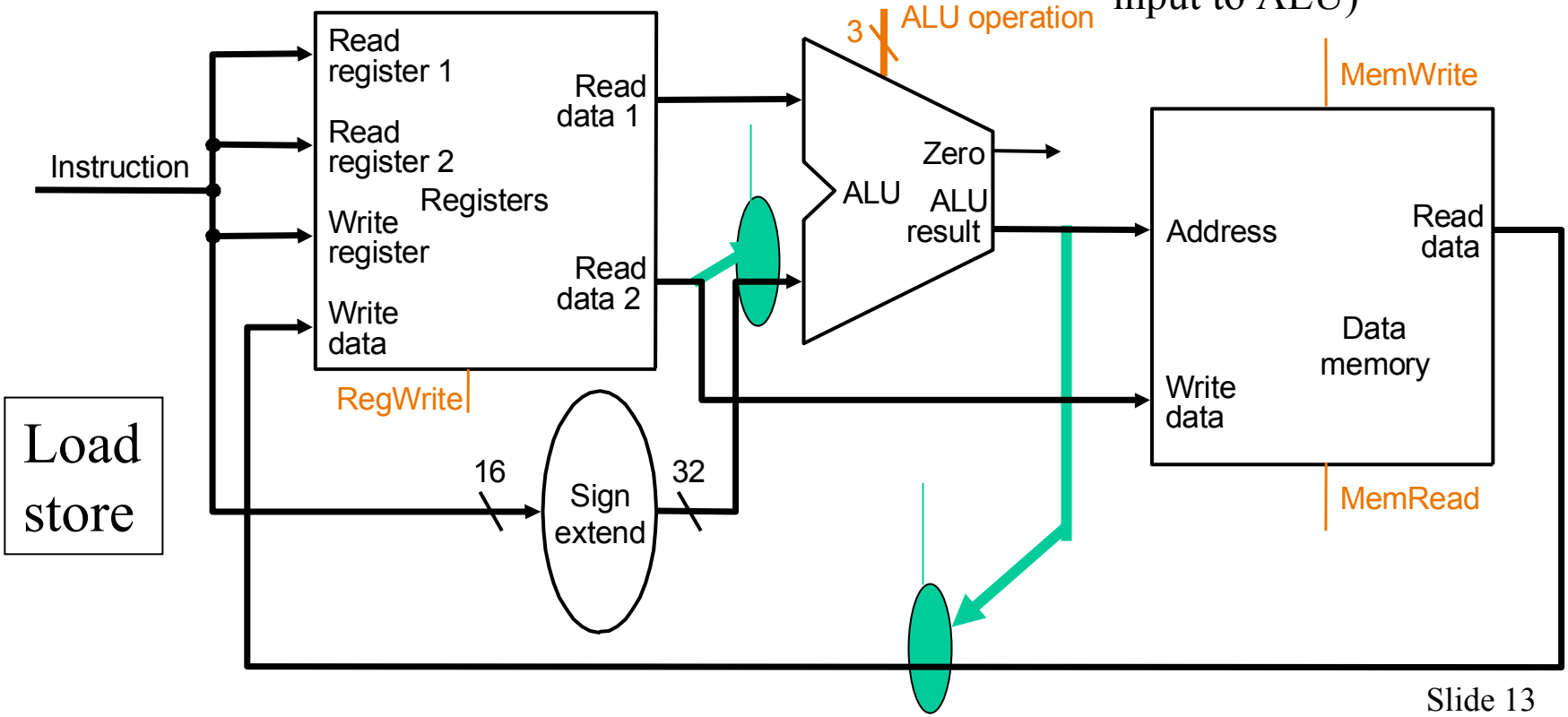
# Single-cycle datapath for branching



# Implementation

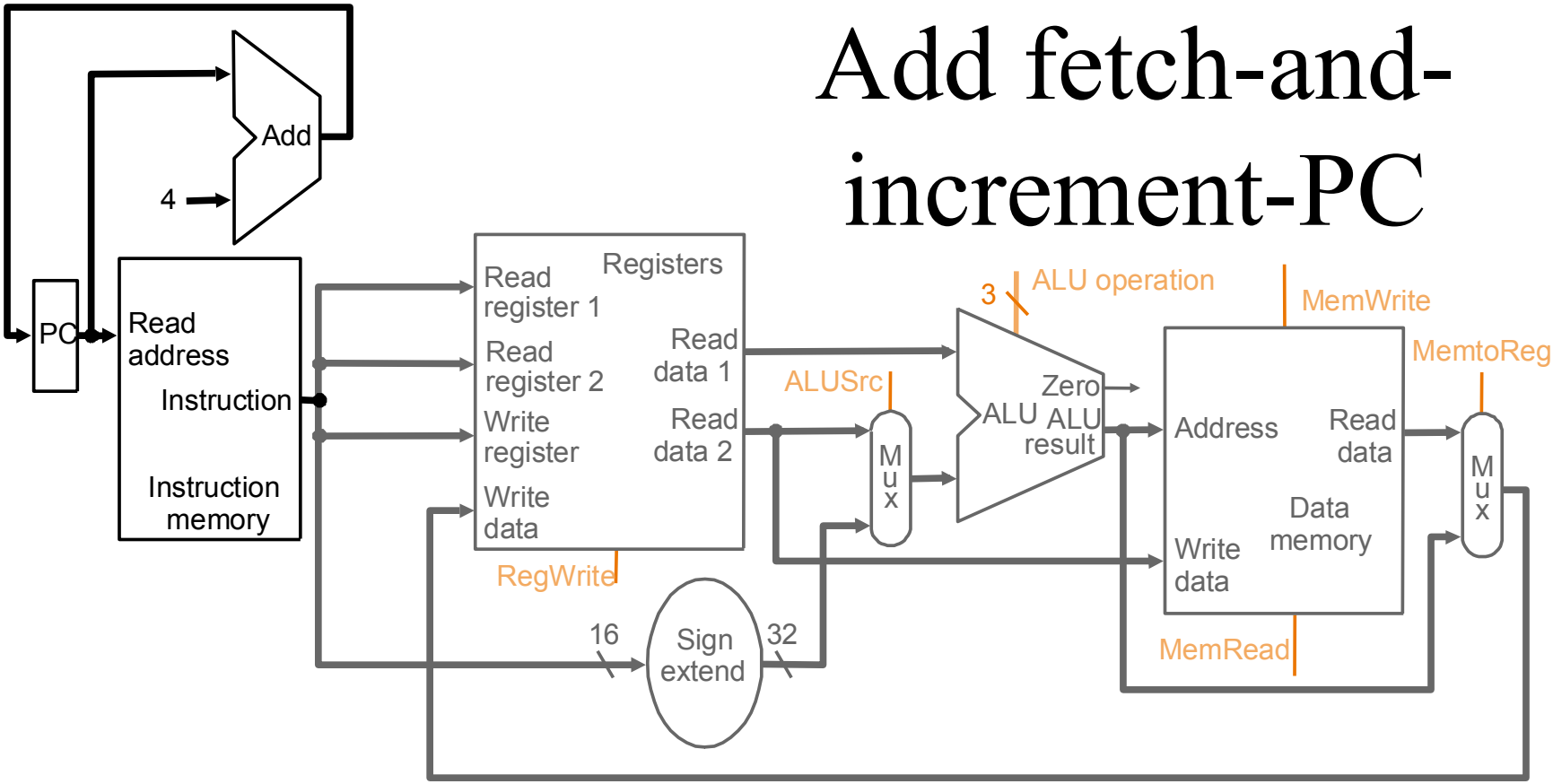
- We have now designed the single-cycle datapath for:
  - Fetch and increment PC
  - R-type instructions (like *add*)
  - Loads and stores
  - Branching instructions
    - Each of the above comes directly from the (RTL) specification of the instruction's behavior.
- Now, we combine them to a single datapath, starting with Load/Store and then:
  - Add R-type
  - Add Fetch-and-increment-PC
  - Add branching



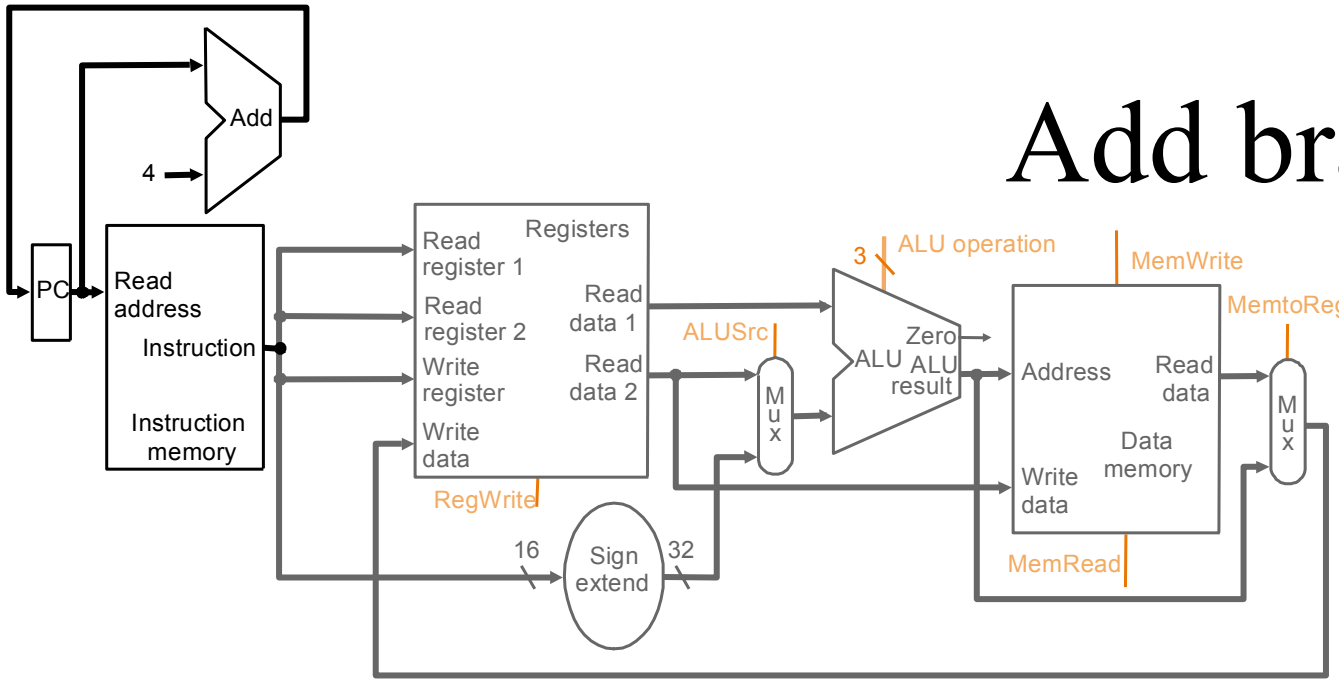

- What components can these datapaths share?
- Issue: output signals that have multiple destinations (e.g. *Read data 2)*
- Issue: input signals that have conflicting sources (e.g. second input to ALU)

# Add fetch-and-increment-PC

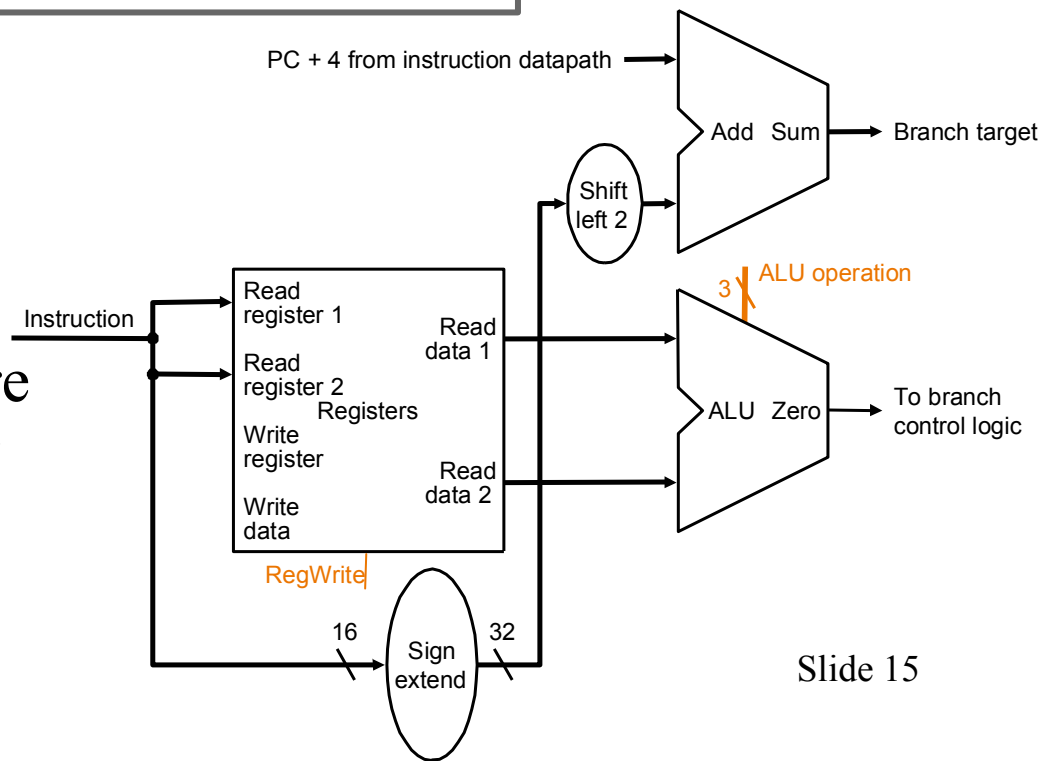



- Why two adders?
  - Why can we sometimes share an ALU, sometimes not?
    - Answer: ALU is combinational logic
    - We must increment the PC and do the ALU operation in a single cycle (in this single-cycle implementation)

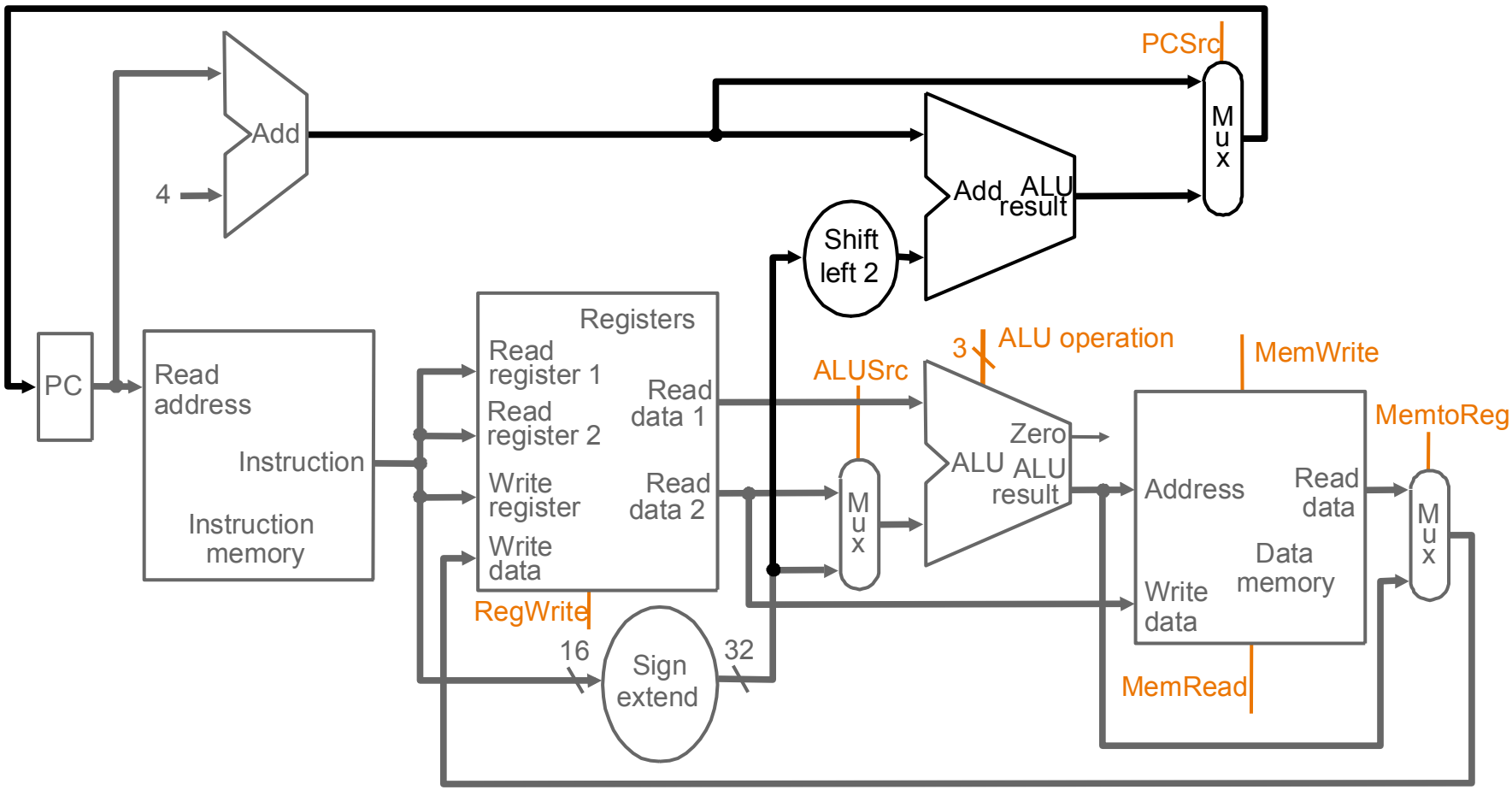
# Add branching




- How to add branching?
  - Output from *Sign-extend* branches to *Shift left 2*
  - Outputs from the *Adds* are multiplexed before going back to PC
    - See next slide for picture



# Summary: single-cycle datapath for "everything" except *jump*



# Principles of datapath design

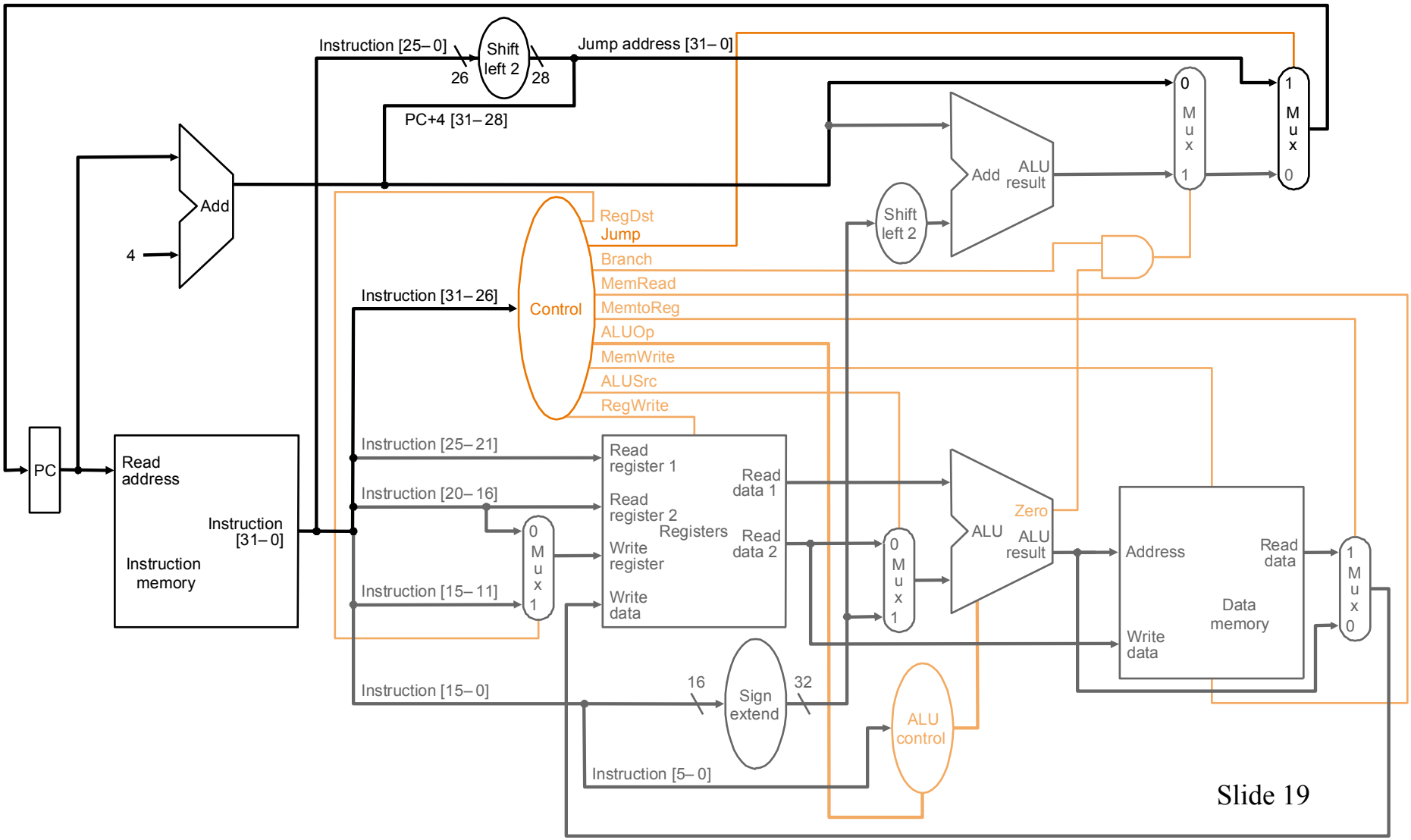
From the preceding examples, we have seen:

- Implement datapaths for related groups of instructions based on the (RTL) specification
- Share components wherever possible, but no component can be used more than once per cycle
- When an output signal has multiple destinations, just branch
- When an input signal has multiple sources, select between them by using a multiplexor

# In-Class Exercise – j

- Write an RTL description for j that uses three short clock cycles (use || to denote concatenation)
- Write a single cycle RTL description
- What components does the datapath for the instruction require?
- What are the inputs, outputs, and control signals for each of these components?
- What signals need to be connected to the inputs of each component?

# Complete Single Cycle Datapath (Fig. 5.17)



# Effects of Single Cycle Datapath Control Signals

State Elements

- RegWrite – When asserted, the register on the Write register input is written with the value on the Write data input
- MemRead – When asserted, the data memory contents designated by the address input are put on the Read data output
- MemWrite – When asserted, the data memory contents designated by the address input are replaced by the value on the Write data input

How should each of these be set for each of the following?

- R-format instructions
- lw
- sw
- beq
- j

# Effects of Single Cycle Datapath Control Signals (cont.)

Multiplexors

- RegDst – determines which field of the instruction (rt or rd) specifies the register destination number
- ALUSrc – determines the source of the second ALU operand (the second register file output or the sign-extended immediate)
- Jump and Branch – together with the Zero output of the ALU, determine the value to replace the current contents of the PC (either PC + 4, the branch target address, or the jump target address)
- MemtoReg – determines the source of the register file's Write data input (either the ALU output or the data memory)

How should each of these be set for each instruction type?

# Effects of Single Cycle Datapath Control Signals (cont.)

| ALUOp1 | ALUOp2 | ALU action |
|---|---|---|
| 0 | 0 | Add |
| 0 | 1 | Subtract |
| 1 | 0 | Determined by funct field |
| 1 | 1 | Unused (don't care) |

How should each of these be set for each instruction type?

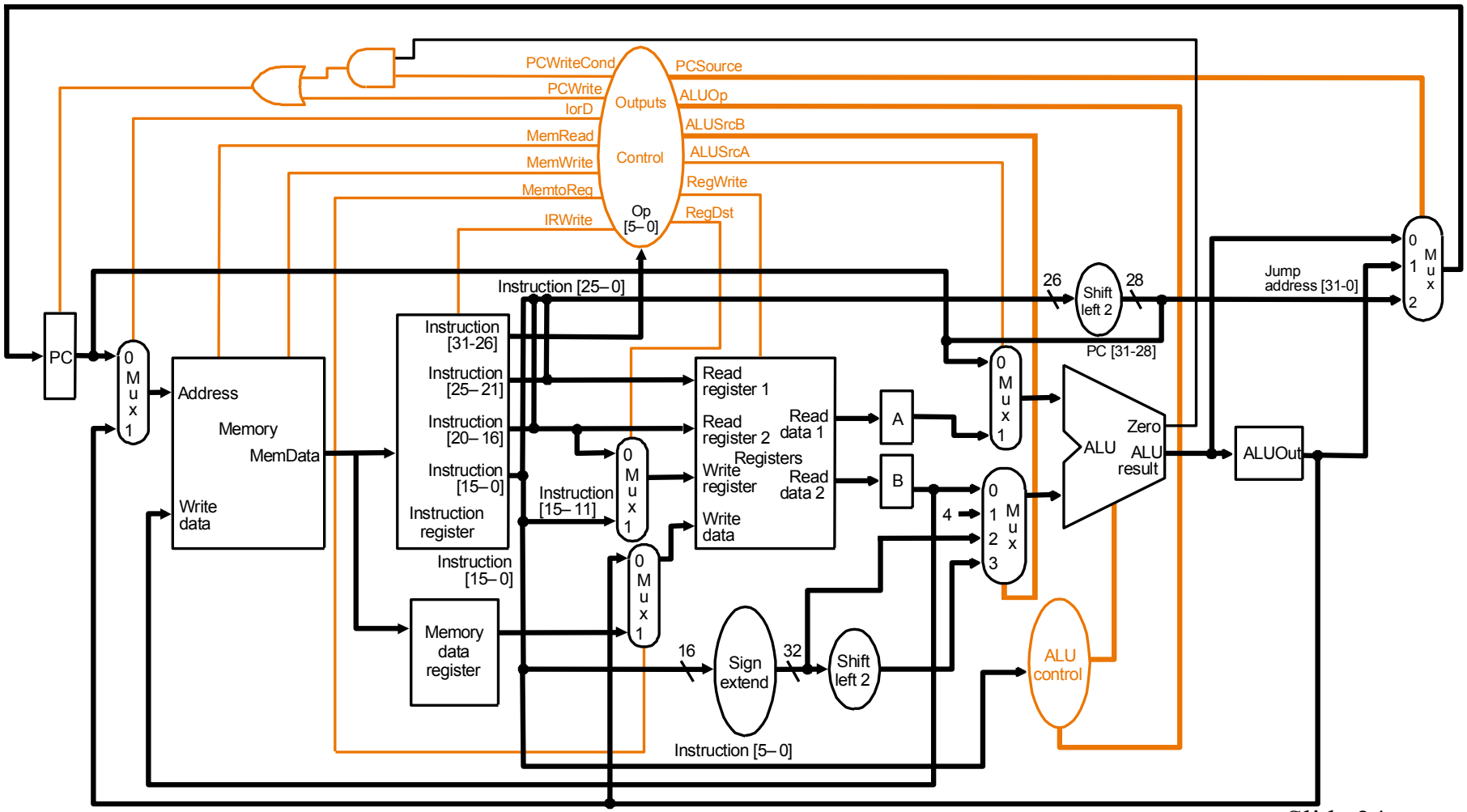
# Single Cycle Control

| | R-format | lw | sw | beq |
|---|---|---|---|---|
| RegDst | 1 | 0 | X | X |
| ALUSrc | 0 | 1 | 0 | 0 |
| MemtoReg | 0 | 1 | X | X |
| RegWrite | 1 | 1 | 0 | 0 |
| MemRead | 0 | 1 | 0 | 0 |
| MemWrite | 0 | 0 | 0 | 0 |
| Branch | 0 | 0 | 1 | 1 |
| ALUOp1 | 1 | 0 | 0 | 0 |
| ALUOp2 | 0 | 0 | 1 | 1 |

- Inputs:
  - opcode field
  - funct field
  - Zero output of ALU
- Outputs:
  - control signals
- State:
  - none

→ Combinational logic

# Multi-cycle Datapath (Fig. 5.33)



# Effects of Multi-cycle Datapath Control Signals

State Elements (new)

- IRWrite – When asserted, the IR is written
- PCWrite – When asserted, the PC is written
- PCWriteCond – When this signal and the Zero output of the ALU are asserted, the PC is written

State Elements (unchanged)

- RegWrite – When asserted, the register on the Write register input is written with the value on the Write data input
- MemRead – When asserted, the memory contents designated by the address input are put on the Read data output
- MemWrite – When asserted, the memory contents designated by the address input are replaced by the value on the Write data input

How should each of these be set for each of the instruction types?

# Effects of Multi-cycle Datapath Control Signals (cont.)

Multiplexors (new or renamed)

- IorD – determines the source of the address supplied to the memory (the PC or ALUOut)
- PCSource – determine the value to (possibly) replace the current contents of the PC (the output of the ALU, the contents of ALUOut, or the jump target address)
- ALUSrcA – determines the source of the first ALU operand (the PC or the first register file output)
- ALUSrcB – determines the source of the second ALU operand (the second register file output, the constant 4, the sign-extended immediate, or the latter left-shifted two bits)

Multiplexors (unchanged)

- RegDst – determines which field of the instruction (rt or rd) specifies the register destination number
- MemtoReg – determines the source of the register file's Write data input (either the ALU output or the data memory)



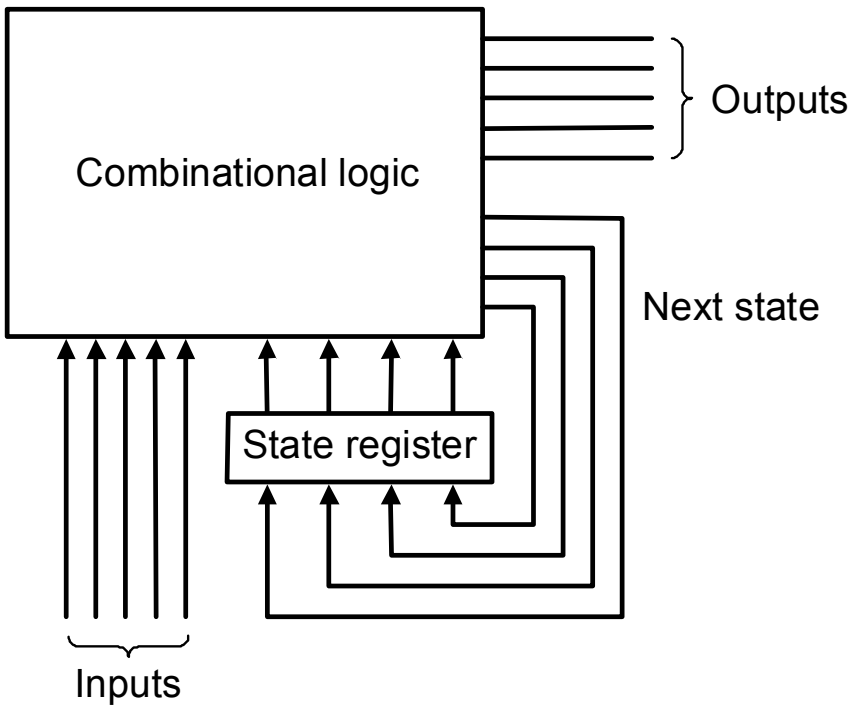
How should each of these be set for each instruction type?

# Effects of Multi-cycle Datapath Control Signals (cont.)

| ALUOp1 | ALUOp2 | ALU action |
|---|---|---|
| 0 | 0 | Add |
| 0 | 1 | Subtract |
| 1 | 0 | Determined by funct field |
| 1 | 1 | Unused (don't care) |

How should each of these be set for each instruction type?

# Multi-cycle Control

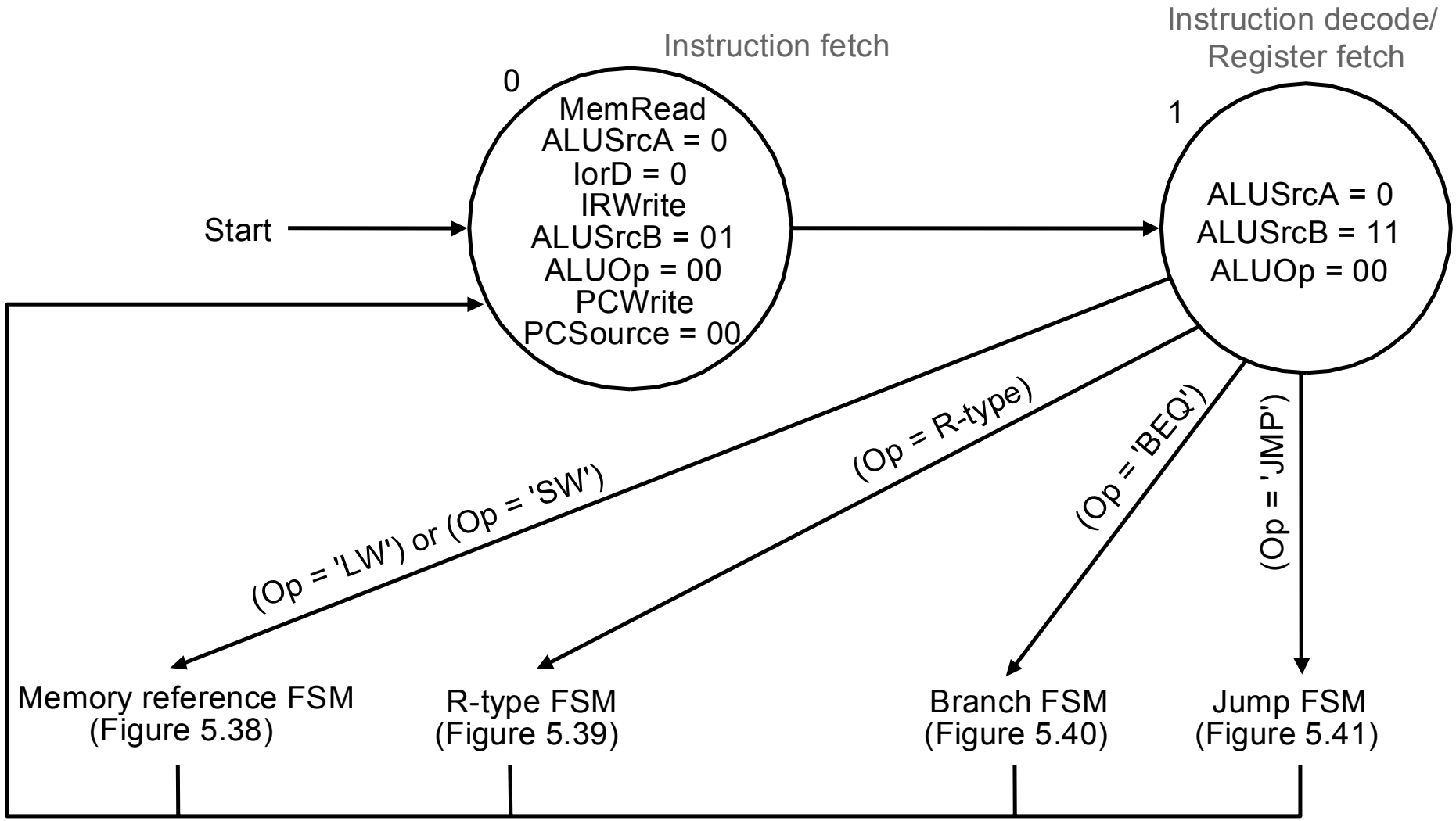



- Input:
  - opcode field
- Outputs:
  - control signals
- State:
  - Instruction type
  - Cycle number

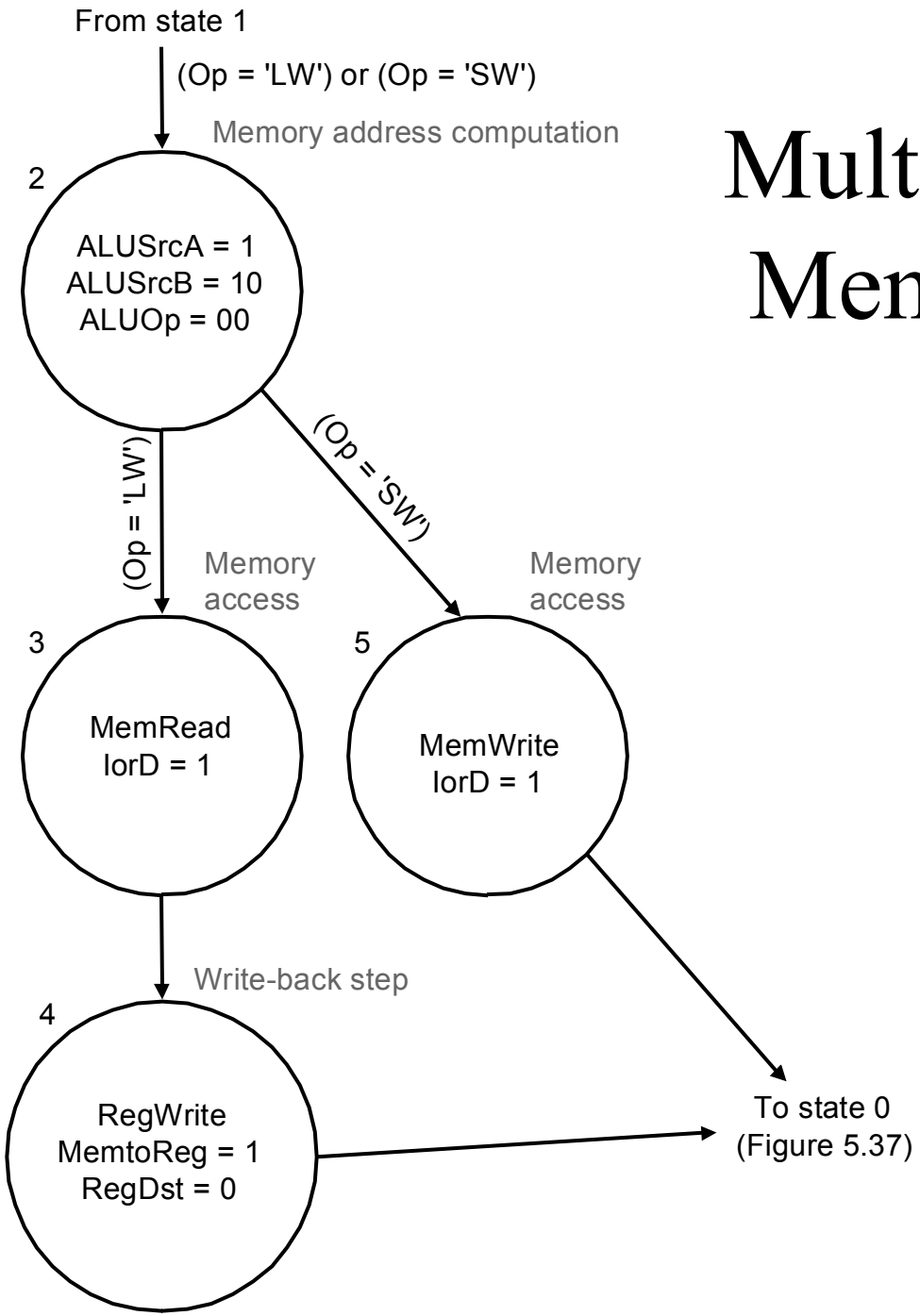
→ Sequential logic

  - FSM is one way to specify
  - ALU control and part of PC control are combinational

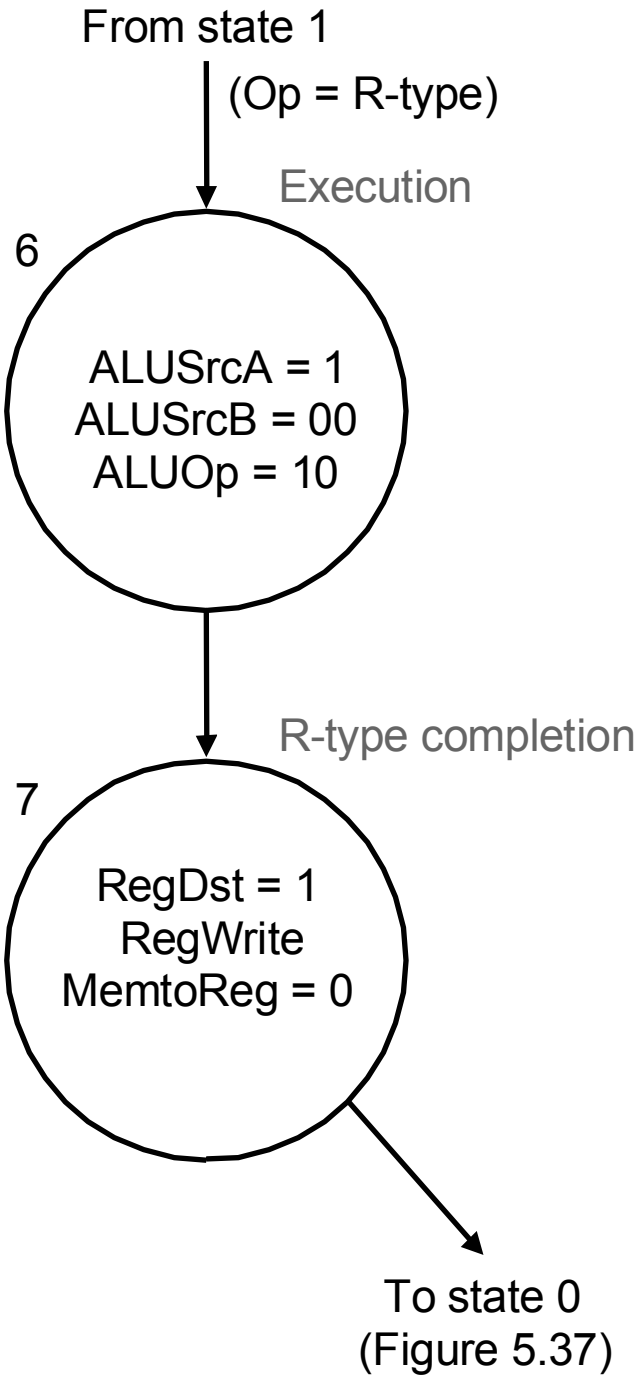
# Multi-cycle Control: Fetch and Decode



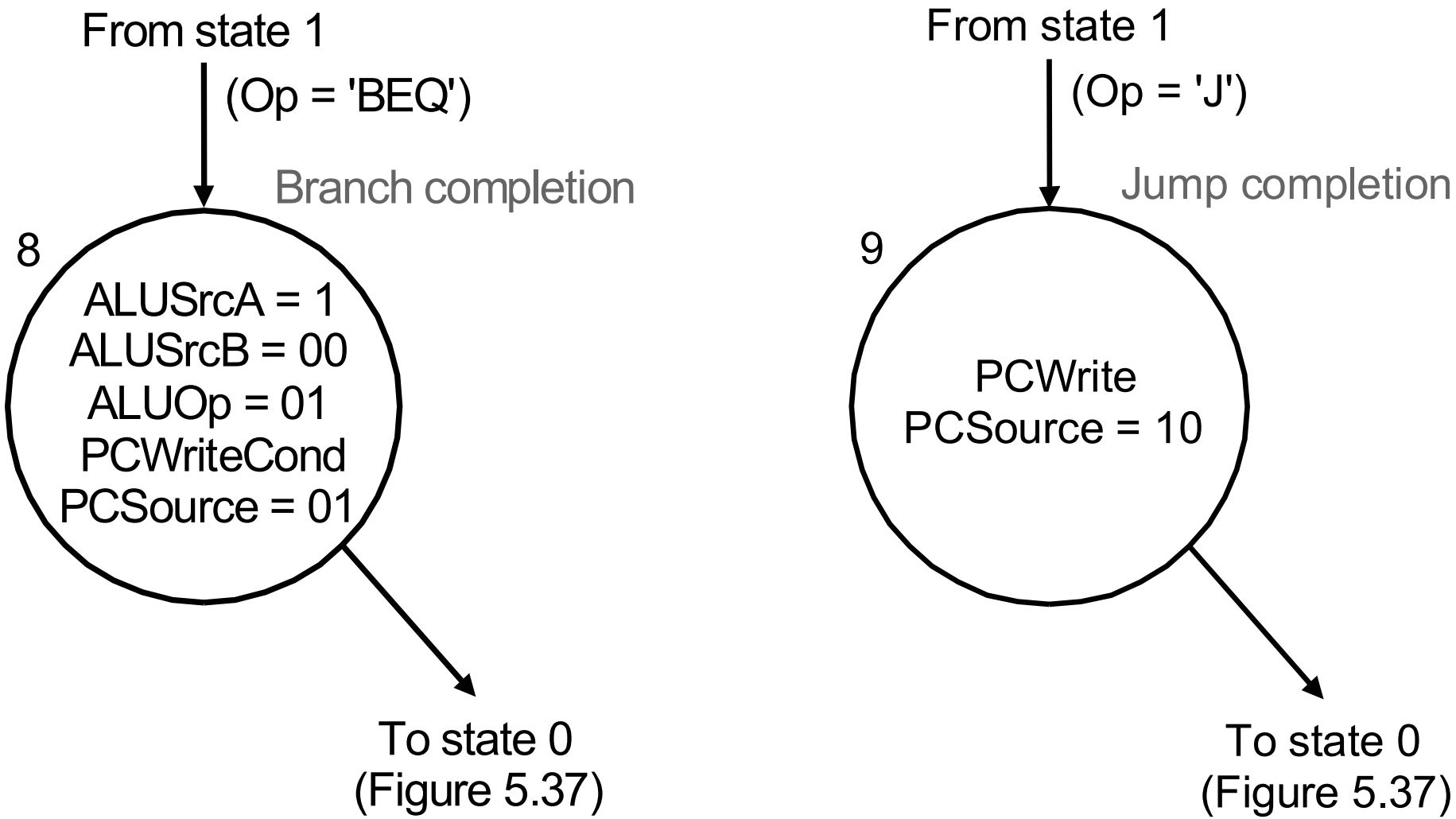
# Multi-cycle Control: Memory Reference

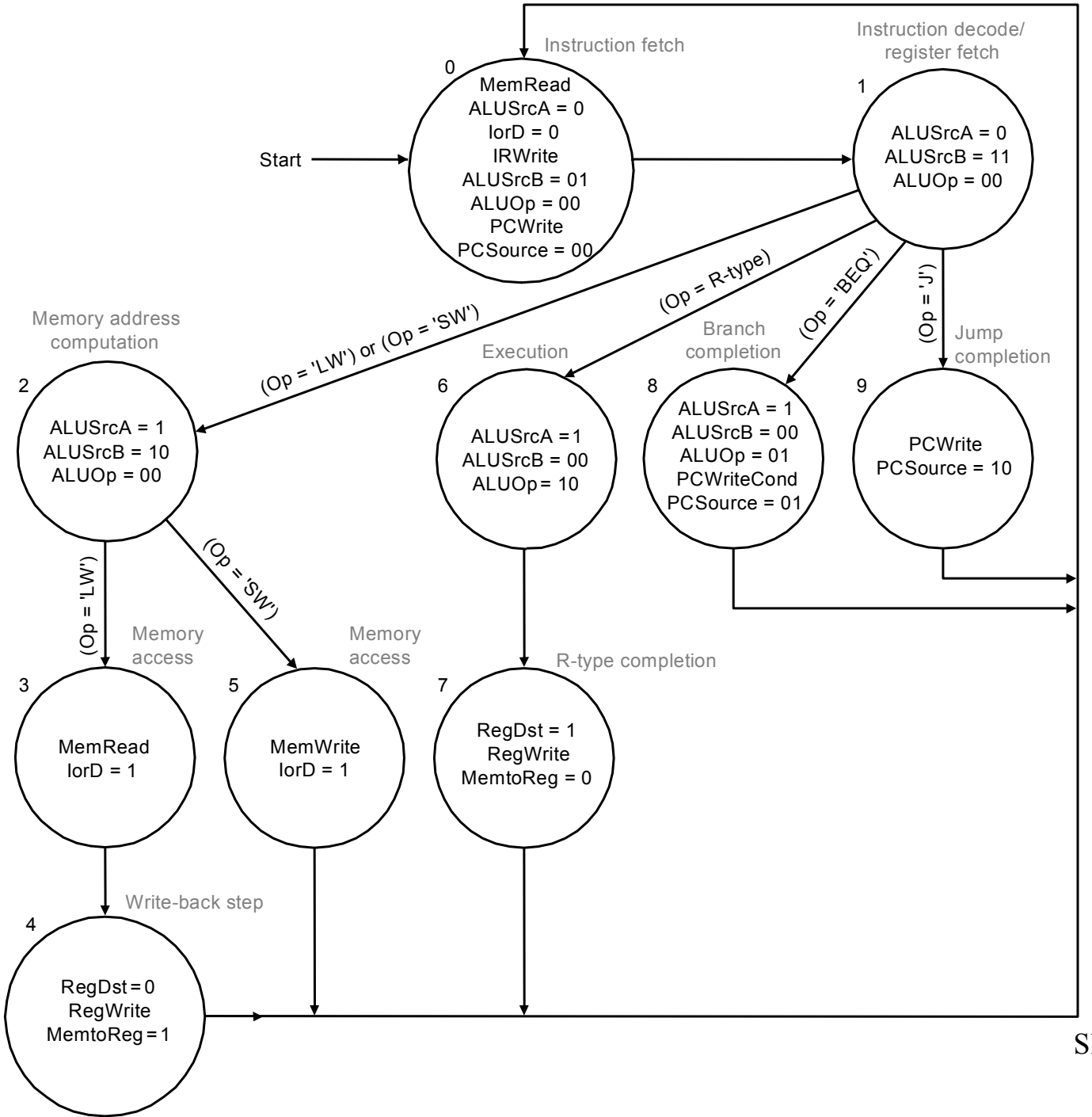


# Multi-cycle Control: R-Type



# Multi-cycle Control: Branches and Jumps




Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
Execution
Branch
completion
Jump
completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Write-back step
4
RegDst = 0
RegWrite
MemtoReg = 1

# Summary

- Register Transfer Language (RTL)
  - Describes state changes
  - Syntax is similar to programming languages
    - Should be accompanied by English description of components, their inputs, their outputs, and their control signals
  - Determines number of cycles per instruction
  - Influences clock cycle time
- Datapath
  - Block diagram of components (can be determined from RTL)
    - Combines components that have multiple non-conflicting uses
  - Shows interconnections explicitly
    - Resolves conflicts where input signals have multiple sources
  - Explicitly identifies components' control signals
    - Should be accompanied by English description of control signals
- Control (component level digital logic circuit)
  - Single-cycle: combinational logic
  - Multicycle: sequential logic (e.g. FSM), possibly with some combinational parts (e.g. ALU control, branch control)