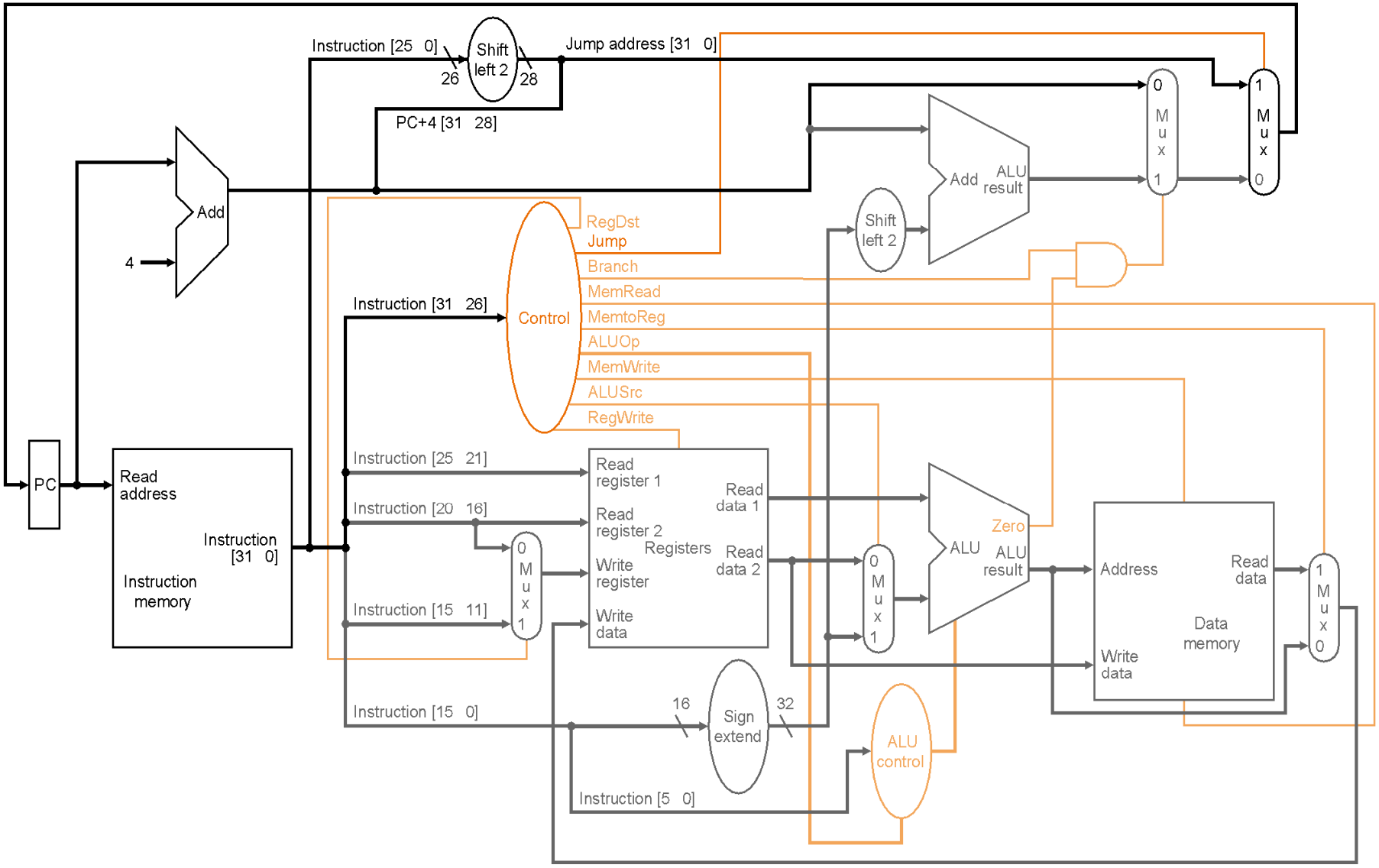
The Processor: Multicycle Implementation

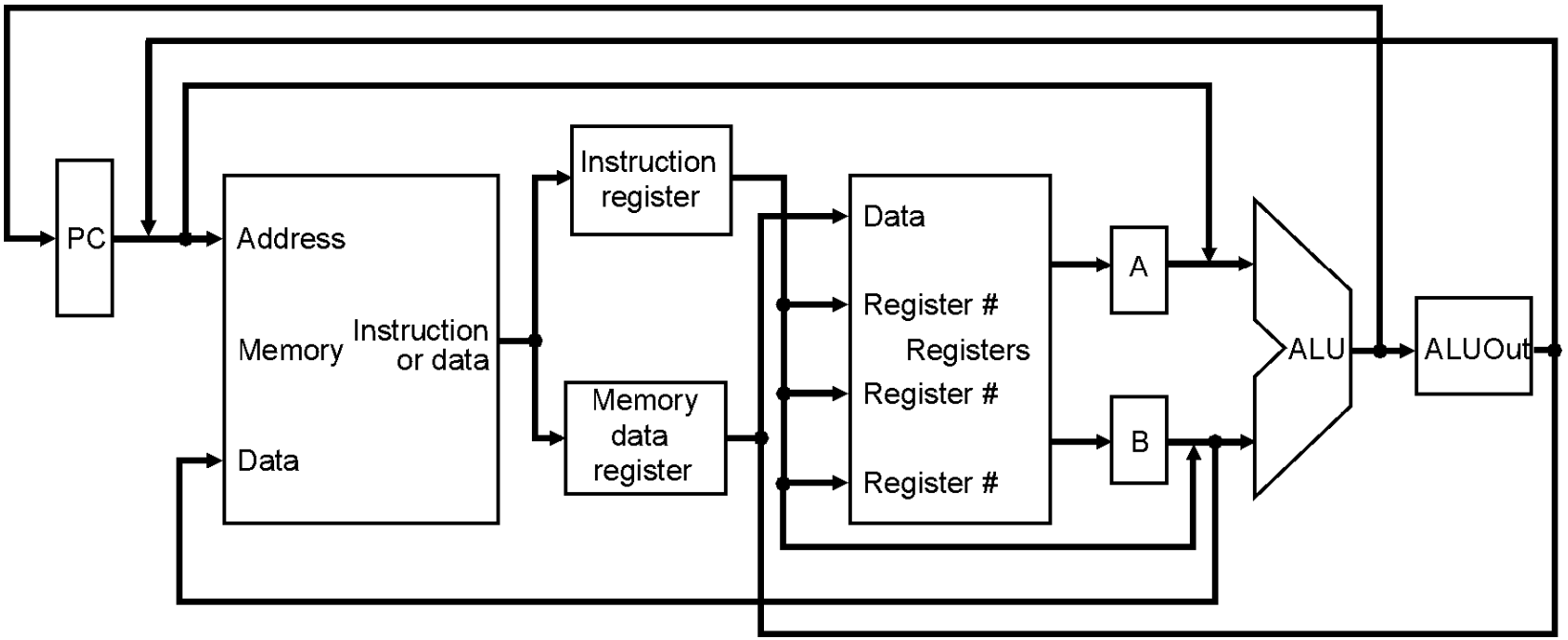
Reminder: Single-Cycle Datapath



Multicycle Approach

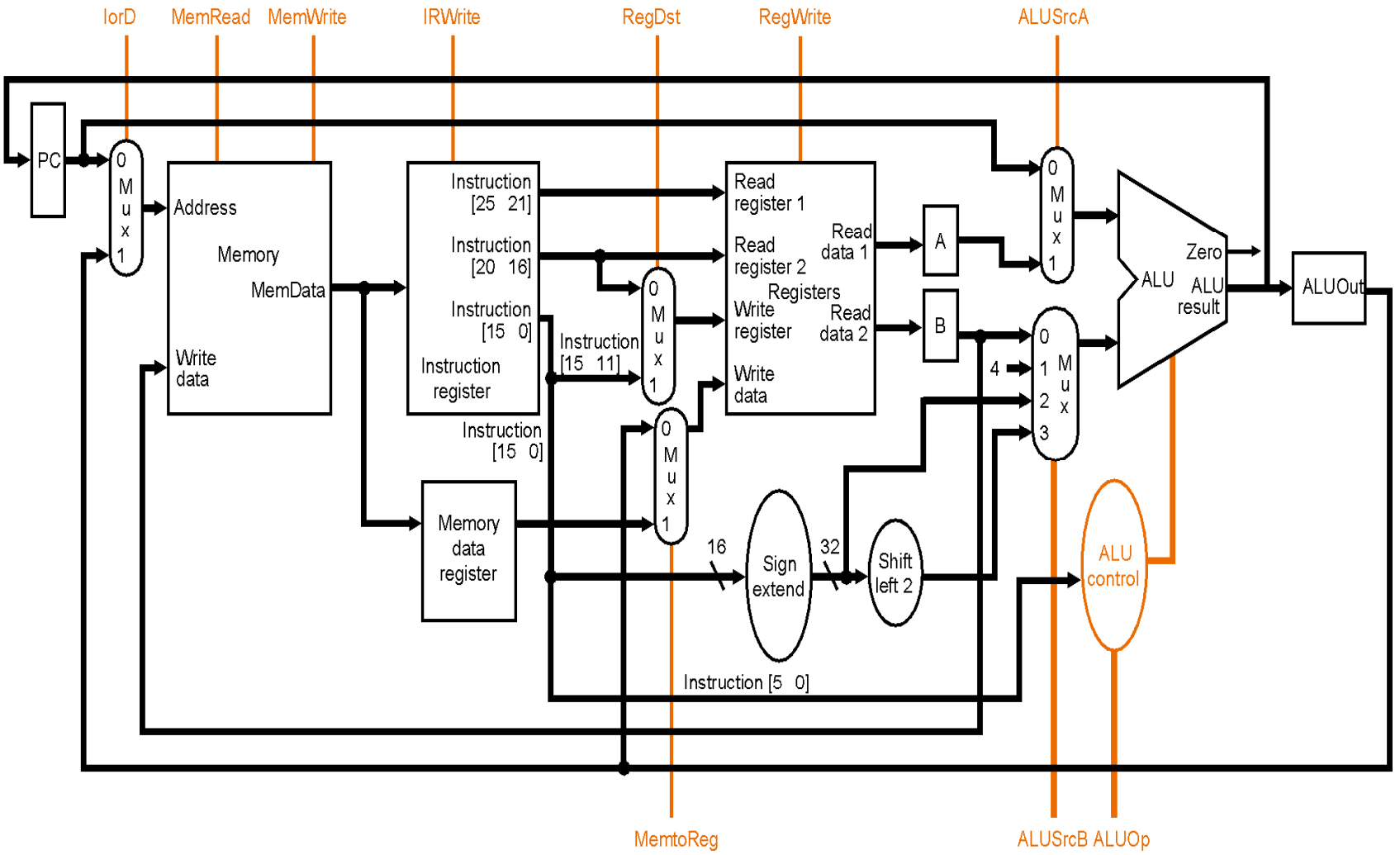
- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work in cycles
  - only one major functional unit is used in each cycle
- At the end of each cycle
  - store values for use in later cycles
  - introduce additional "internal" registers
- We will be recycling functional units
  - ALU used to compute branch address and to increment PC
  - The same memory used for instruction and data
- We will use finite state machine for control

Multicycle Implementation



- buffer registers between functional units which will be updated every clock cycle
- IR: Instruction register
- MDR: Memory Data register
- A and B registers to hold register operand values read from register file
- ALUOut: to hold output of ALU

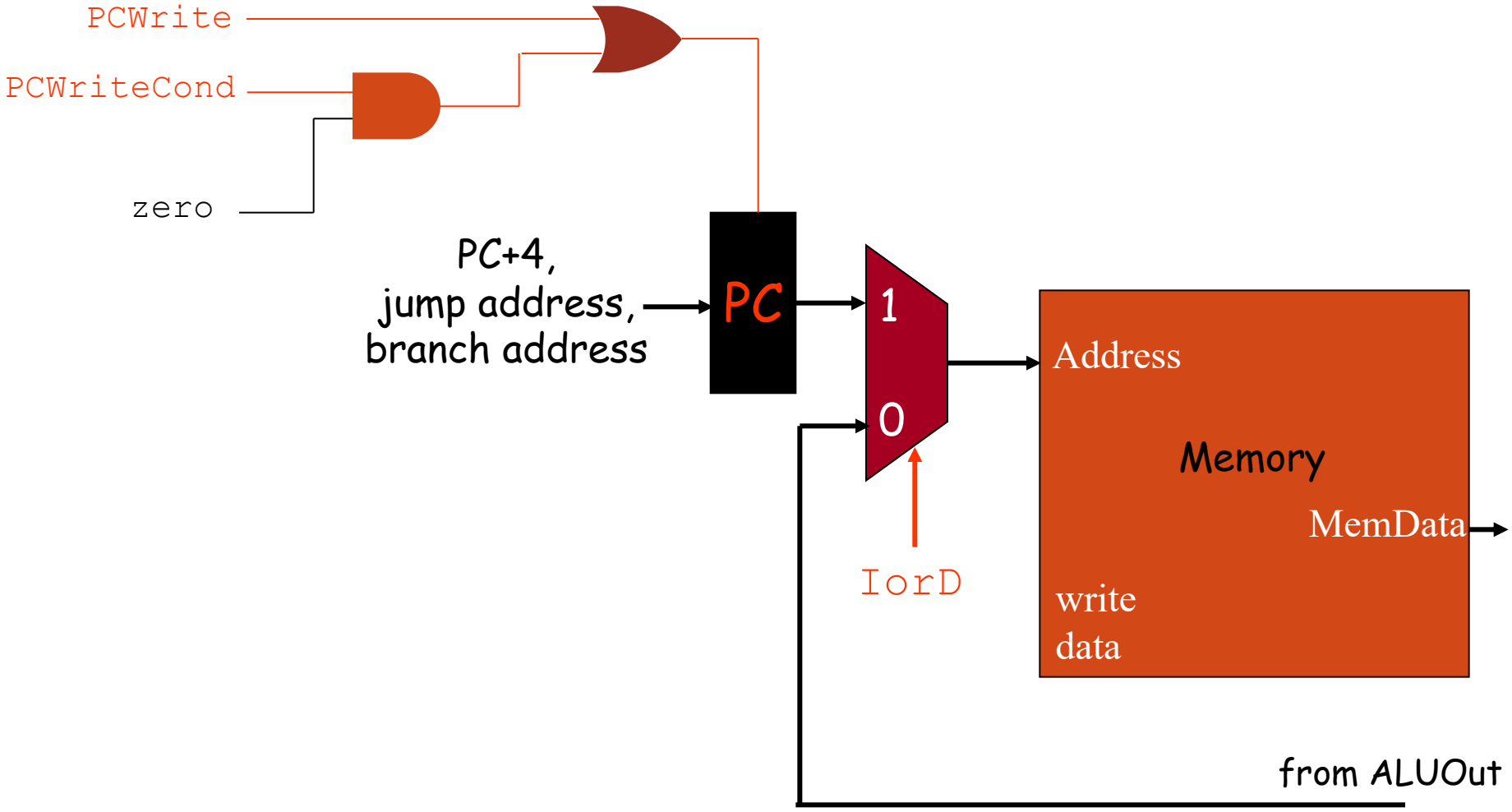
Datapath with Control Signals



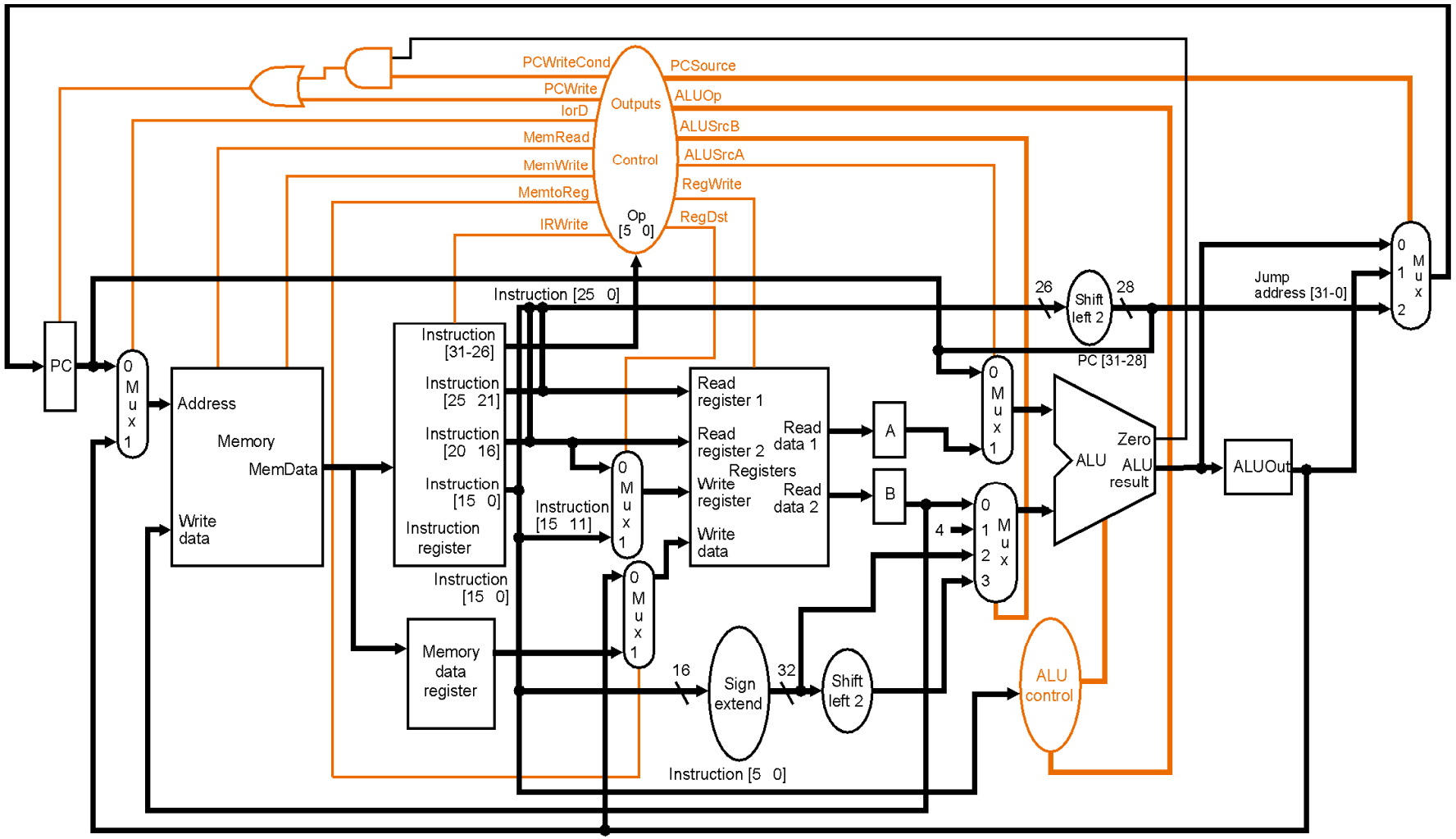
Handling PC

- Multiplexor for selecting the source for the `PC`
  - ALU output after `PC+4`
  - ALU output after branch target address calculation
  - New address with jump instructions (26 bits of `IR`)
- Control signals for writing `PC`
  - Unconditionally after a normal instruction and jump - `PCWrite`
  - Conditionally overwritten if a branch is taken
    - When `PCWriteCond` and `zero` are asserted, the branch address at the output of ALU is written into the PC.

Handling PC


PCWrite
PCWriteCond
zero
PC+4,
jump address,
branch address
PC
1
0
IorD
Address
Memory
MemData
write
data
from ALUOut

Complete Datapath & Control



Five Execution Steps

1. Instruction Fetch
2. Instruction Decode and Register Fetch
3. Execution, Memory Address Computation, or Branch Completion
4. Memory Access or R-type instruction completion
5. Write-back step

INSTRUCTIONS TAKE 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get the instruction and put it in IR.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL

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

  The ALU is used for incrementing PC

- Control signals

```
MemRead = ? IRWrite = ? PCWrite = ?
IorD = ? ALUSrcA = ?
ALUSrcB = ?? ALUOp = ??
PCSource = ??
```

Step 2: Instruction Decode & Register Fetch

1. Read registers `rs` and `rt` in case we need them
2. 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 yet

```
ALUSrcA = ?        (PC)
ALUSrcB = ??       (sign-extended & shifted
                   offset)
ALUOp = ??         (add)
```

Step 3 (Instruction Dependent)

- ALU performs for one of the four functions

1. Memory Reference:

```
ALUOut <= A + sign-extend(IR[15-0]);
ALUSrcA = ?, ALUSrcB = ??, ALUOp = ??
```

2. R-type:

```
ALUOut <= A op B;
ALUSrcA = ?, ALUSrcB = ??, ALUOp = ??
```

3. Branch:

```
if (A == B) PC <= ALUOut;
ALUSrcA = ?, ALUSrcB = ??, ALUOp = ??,
PCWriteCond = ?, PCSource = ??
```

4. Jump:

```
PC <= PC[31:28]||(IR[25:0]<<2)
PCWrite = ?, PCSource = ??
```

Step 4 (R-type or Memory-Access)

1. Load and store instructions access memory
   - `lw: MDR <= Memory[ALUOut];`
     `MemRead = ?, IorD = ?`
   - `sw: Memory[ALUOut] <= B;`
     `MemWrite = ?, IorD = ?`

2. R-type instructions finish

   ```
   Reg[IR[15-11]] <= ALUOut;
   RegDst = ?, RegWrite = ?,
   MemtoReg = ?
   ```

Write-Back Step

- Memory read completion step
- `Reg[IR[20-16]] <= MDR;`

```
RegDst = ?, RegWrite = ?, MemtoReg = ?
```

What about all the other instructions?

Summary of the Steps

Step Name	Actions for R-type	Actions for memory-access	Action for branch	Action for jump
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 calculations, 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)`

Memory access / R-type completion	`Reg[IR[15-11]] <= ALUOut;`	`Load:`
		`MDR <= Memory[ALUOut];`
		`Store:`
		`Memory[ALUOut] <= B;`		

Memory read completion		`Reg[IR[20-16]] <= MDR;`		

Instructions in Multicycle Implementation

- Loads: 5, Stores: 4, ALU Instructions: 4, Branch: 3, Jump: 3
- Example: From SPECINT2000, instruction mix:
  - 25 % loads (1% load byte + 24% load word)
  - 10% stores (1% store byte + 9% store word)
  - 11 % branches (6% beq + 5% bne)
  - 2 % jumps (1% jal + 1% jr)
  - 52% ALU operations
- CPI = formula = 4.12

Example

- How many cycles does 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 exactly is happening during the 8th cycle of execution?
- In what cycle does the actual addition of `$t2` and `$t3` take place?

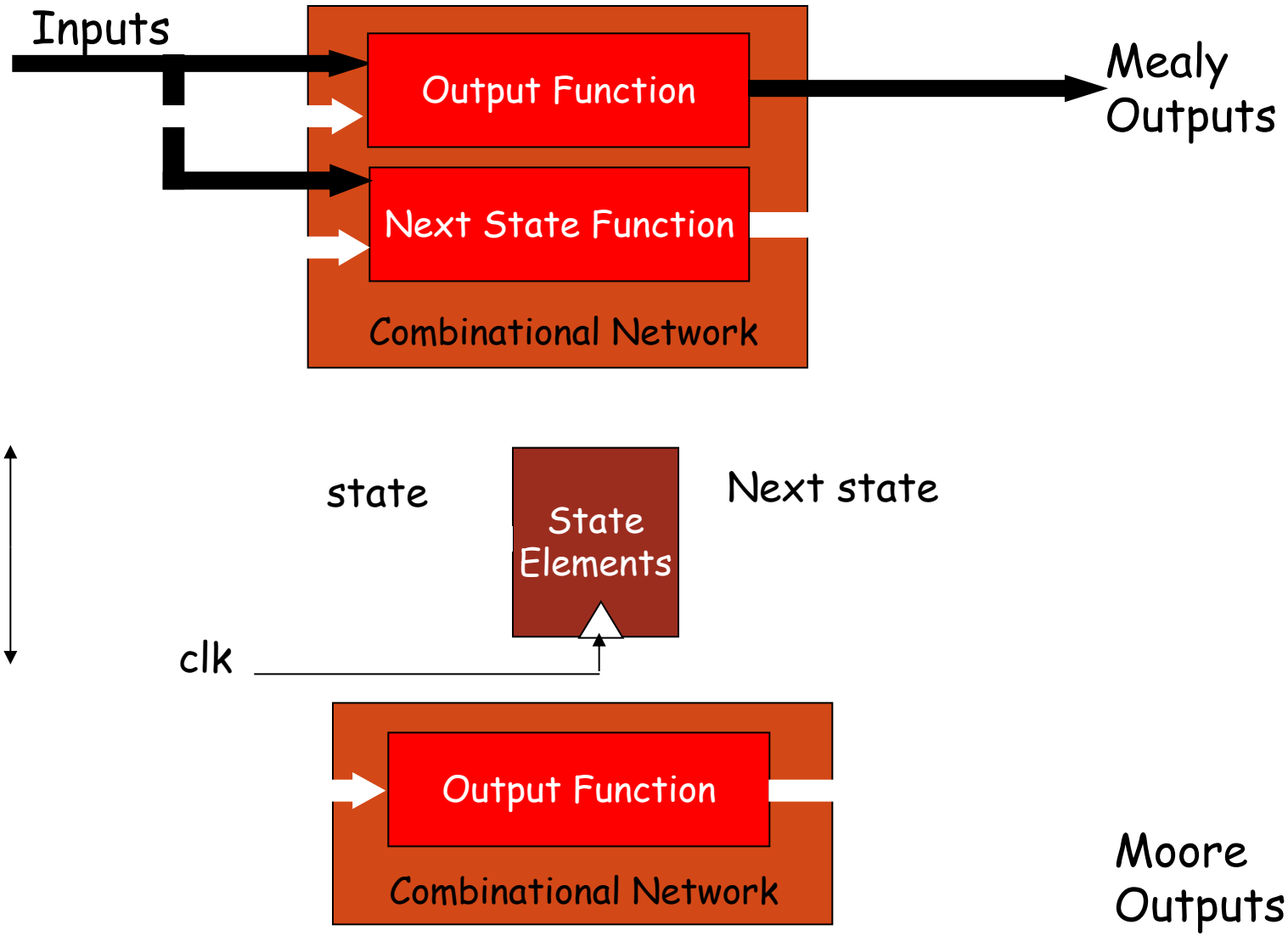
Implementing the Control

- A control unit that will
  - assert control signals at the right time
  - determine the next step
- Value of control signals is dependent upon:
  - which instruction is being executed
  - which step is being performed (implies sequential logic)
- Two specification methods
  - State diagrams
  - Microprogramming
- Implementation can be derived from this specification

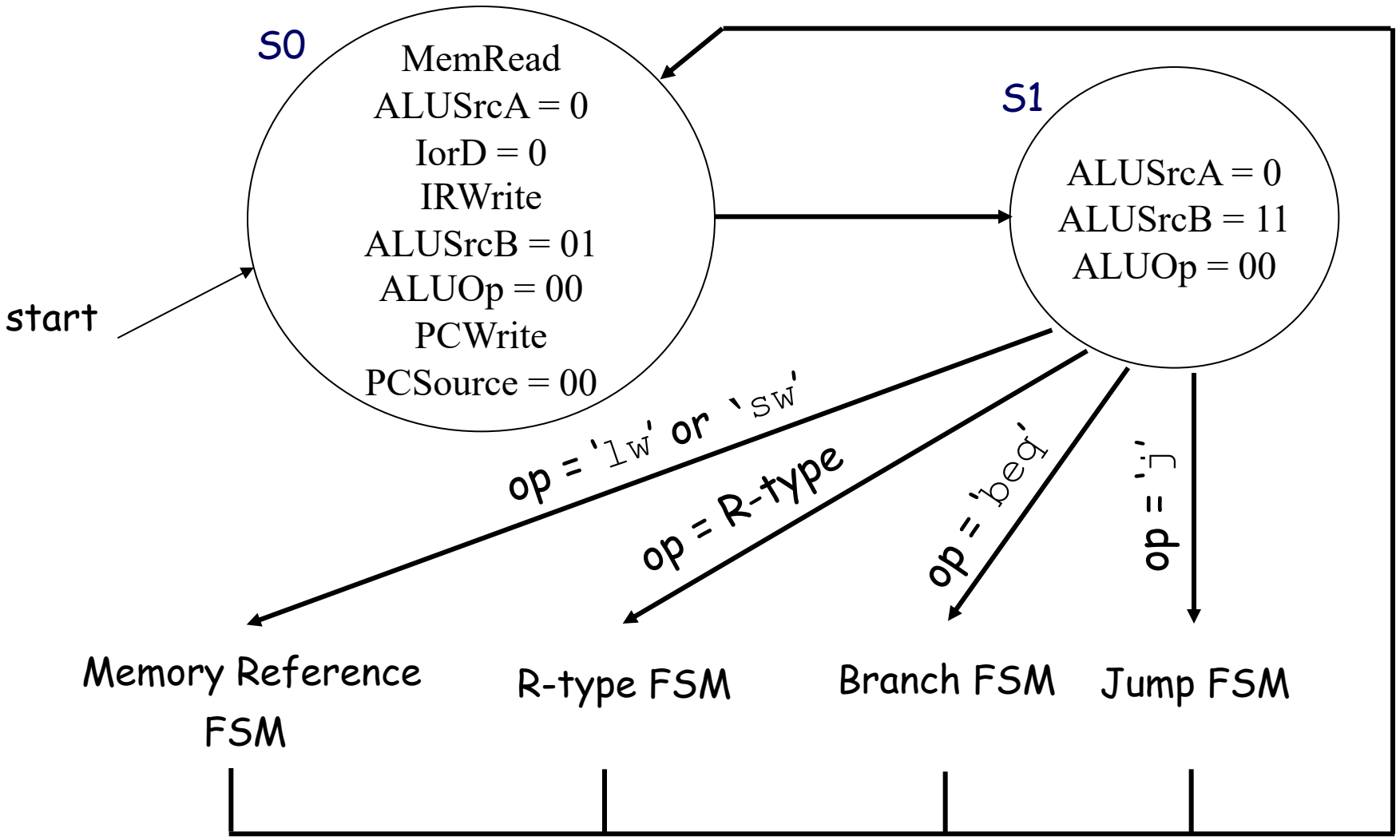
Review: Finite State Machines

- Finite state machines:
  - a finite set of states and
  - next state function (determined by current state and input)
  - output function (determined by current state and possibly input)
  - We'll use a Moore machine (output based only on current state)

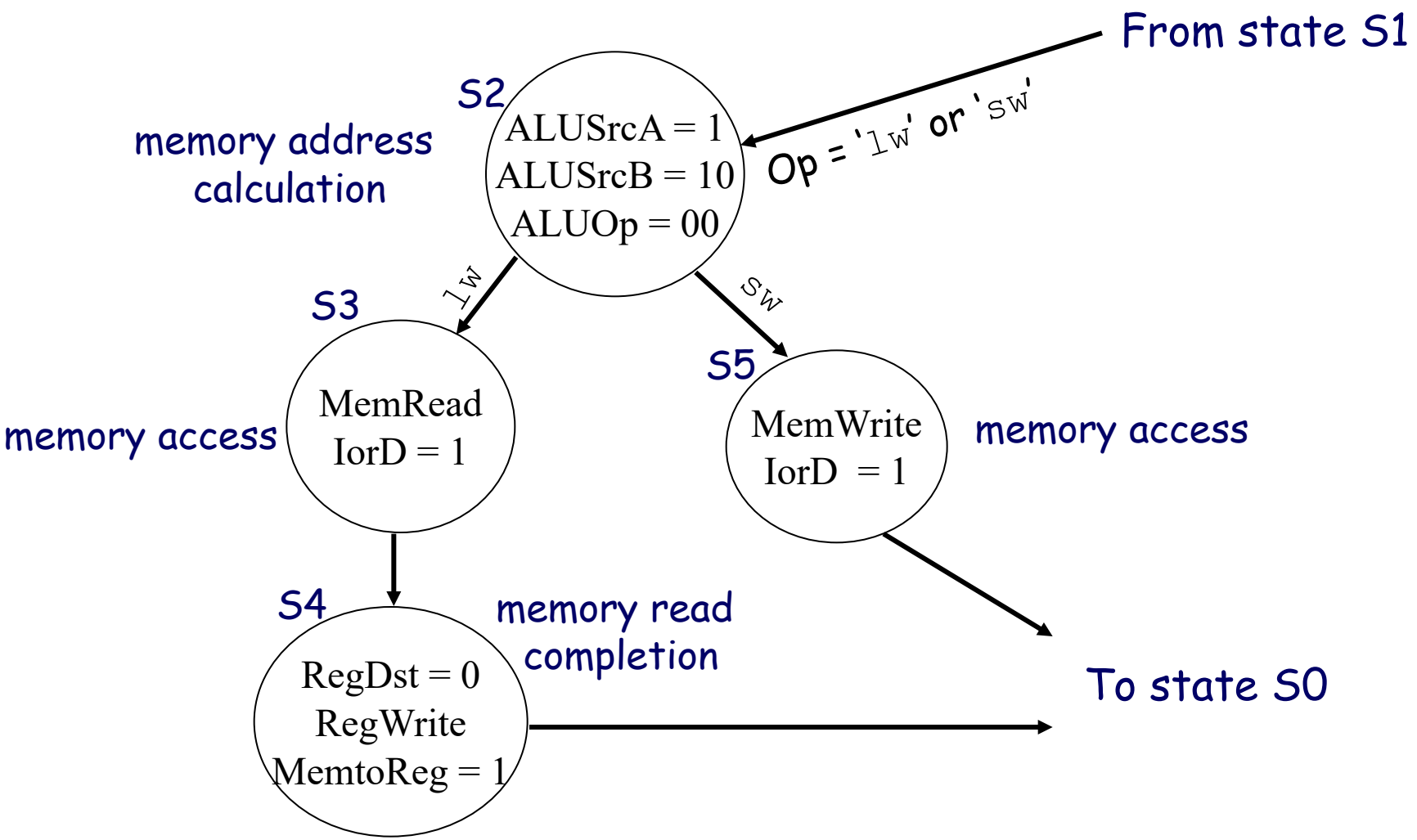
Finite State Machines



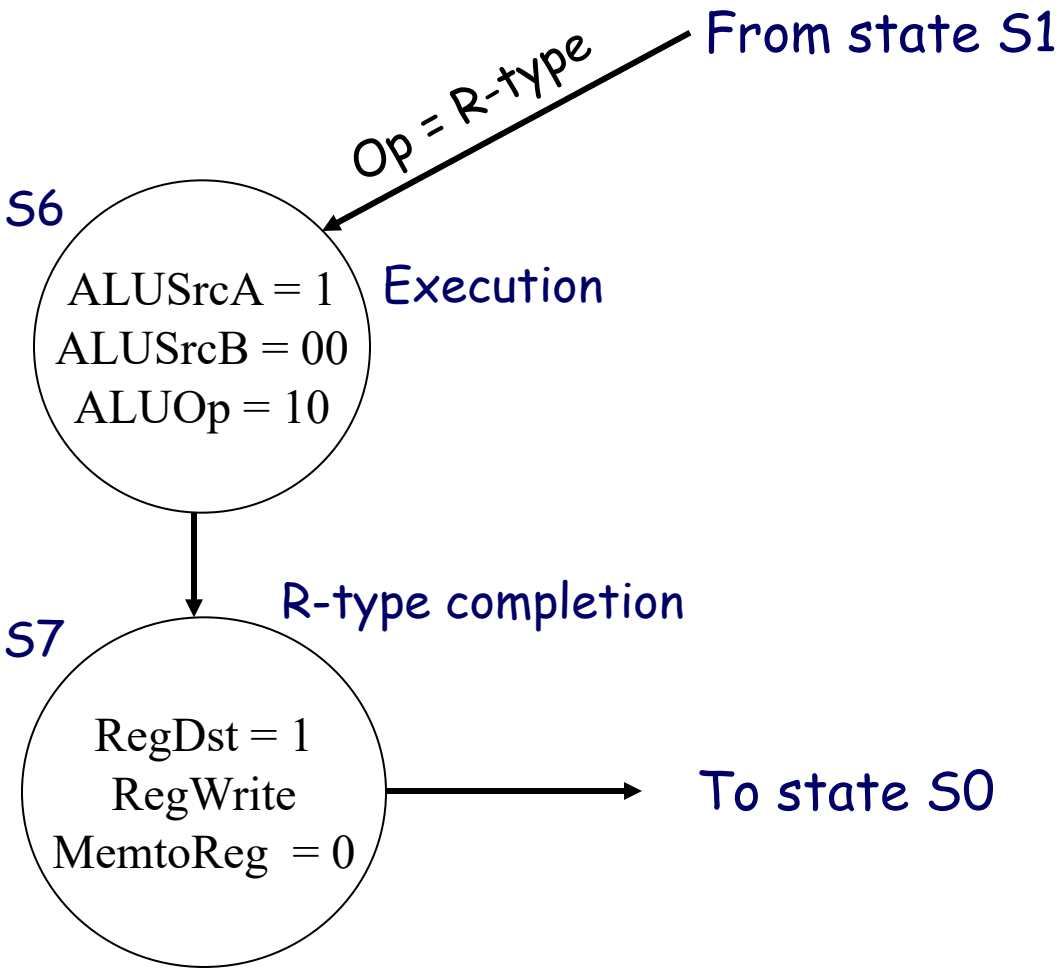
State Diagram for Fetch & Decode



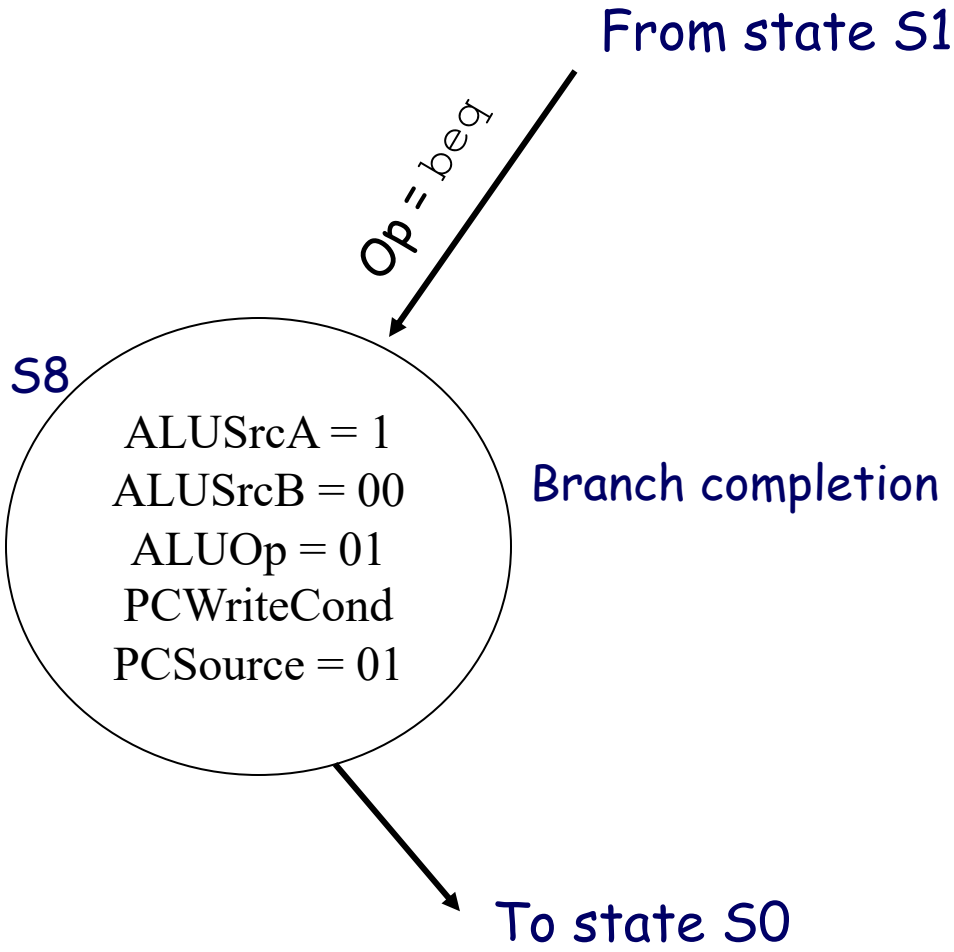
State Diagram for Memory-Access



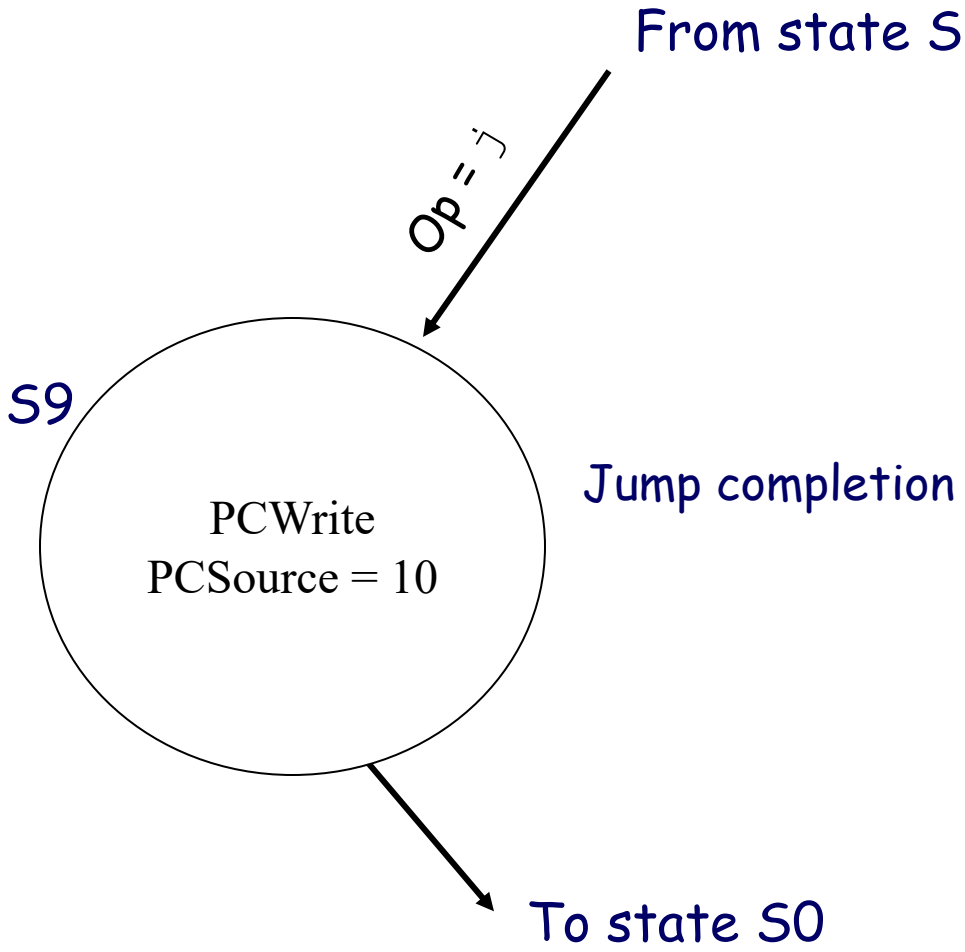
State Diagram for R-type



State Diagram for Branch

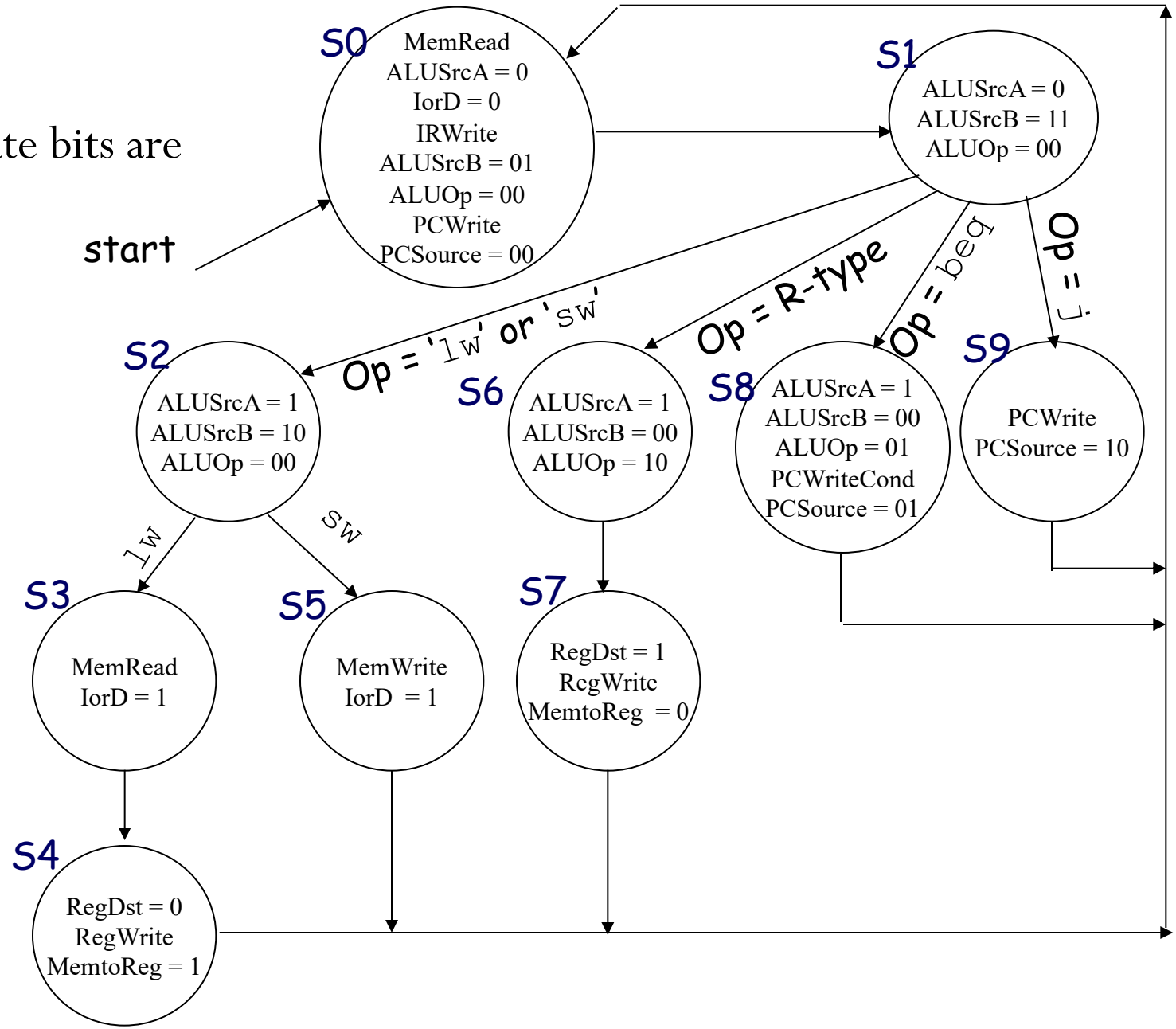


State Diagram for Jump

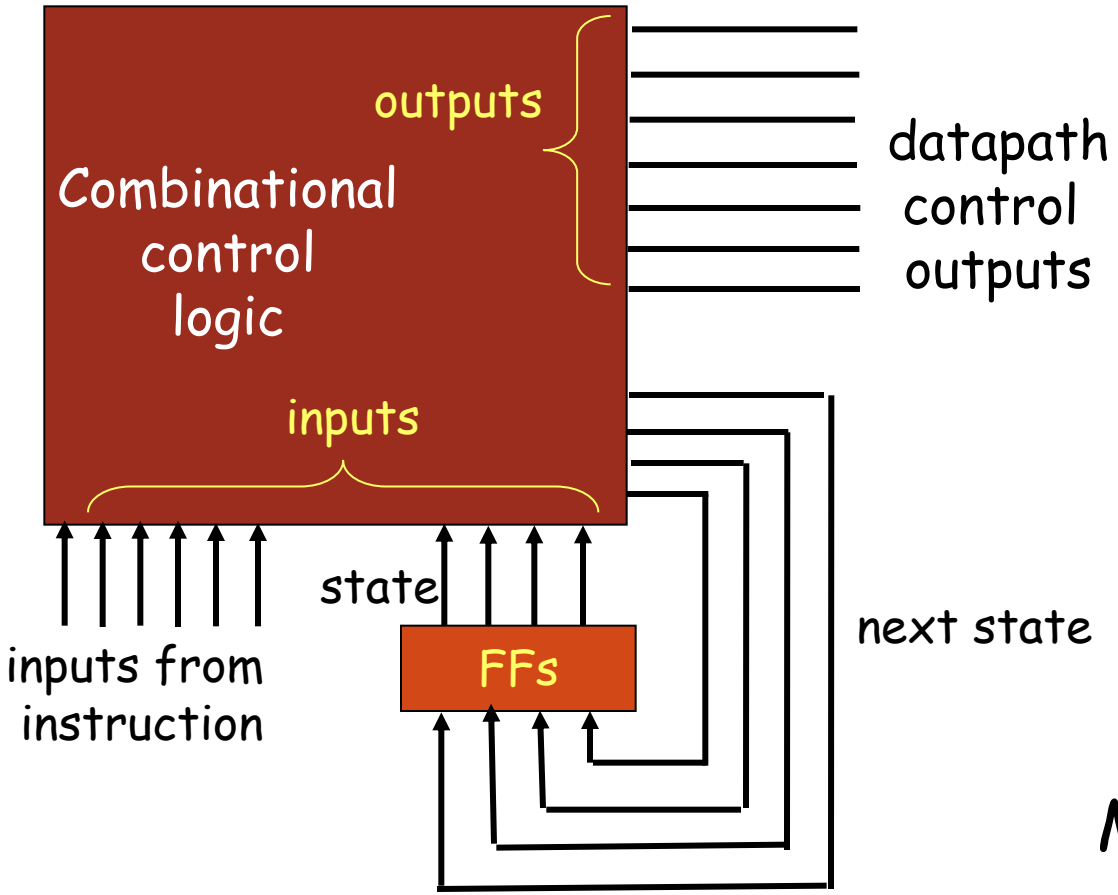


Complete State Diagram

How many state bits are needed?

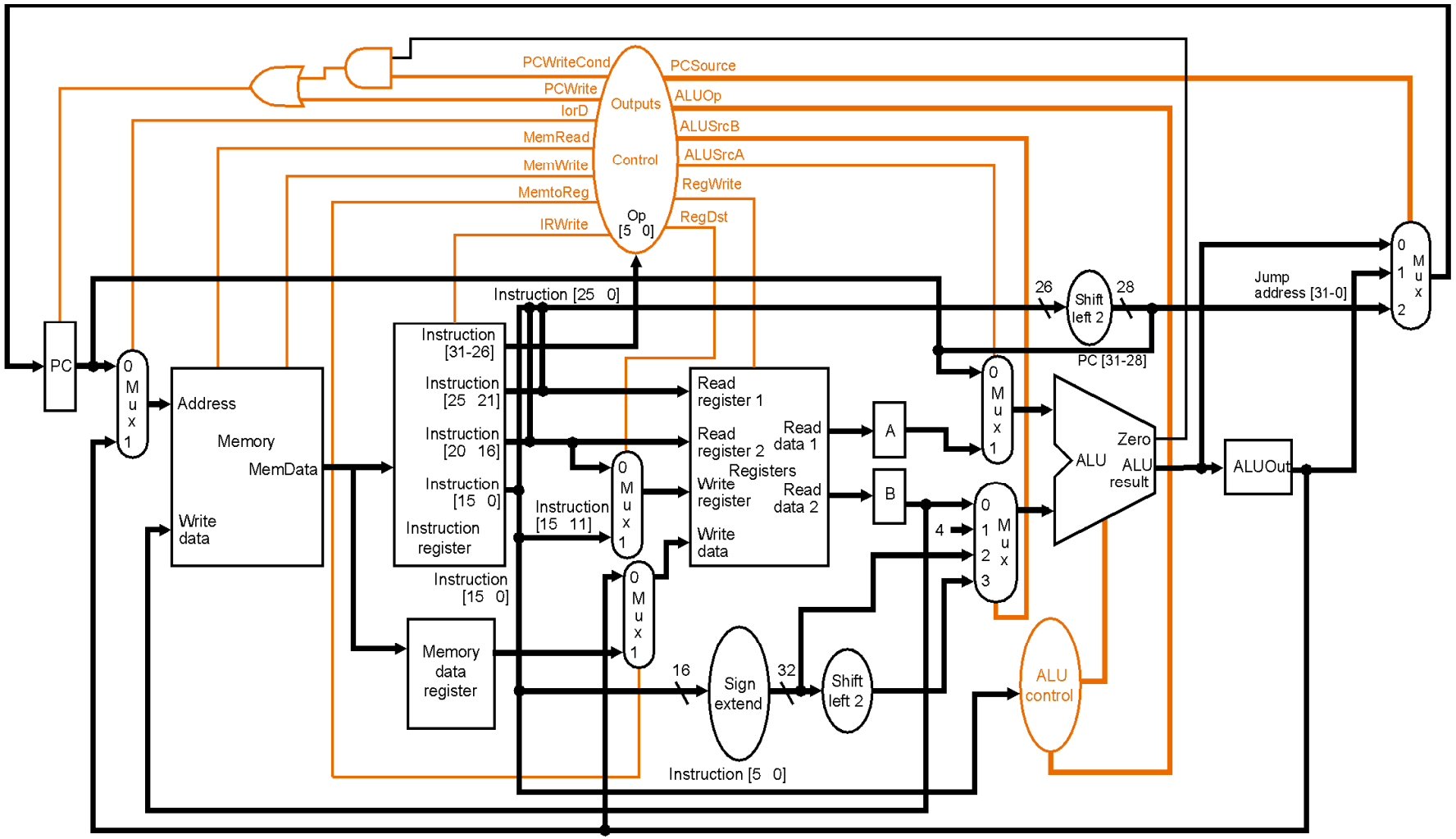


Implementing the Control



Moore or Mealy machine?

Complete Datapath & Control



Exceptions or Interrupts

- Unintentional change of normal flow of instructions
- Exception is an unexpected event from within the processor, e.g arithmetic overflow.
- An interrupt may come from outside of the processor.
- These terms are observed to be used interchangeably (IA32 uses interrupts for both)
- Our simplified MIPS architecture can generate two exceptions:
  - Arithmetic overflow
  - Undefined instruction.

Interrupt or Exception?

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the OS from the user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

When an Exception Occurs 1/2

- *Exception program counter* (EPC)
  - PC of the current instruction (i.e. PC-4) in EPC
- Transfer the control to OS at some specified address.
- The OS does what needs to be done, it either
  - Stops the execution of the program and reports an error
  - Provides a service to the user program (e.g a predefined action to an overflow) and return to the point of origin of exception using EPC

When an Exception Occurs 2/2

- The OS must know the reason for the exception
- An exception is handled using two approaches:
  1. cause register contains the reason for exception (MIPS).
  2. Vectored interrupt: For each interrupt type, a separate vector is stored and each interrupt vector targets at the appropriate piece of code.

Vectored Interrupts

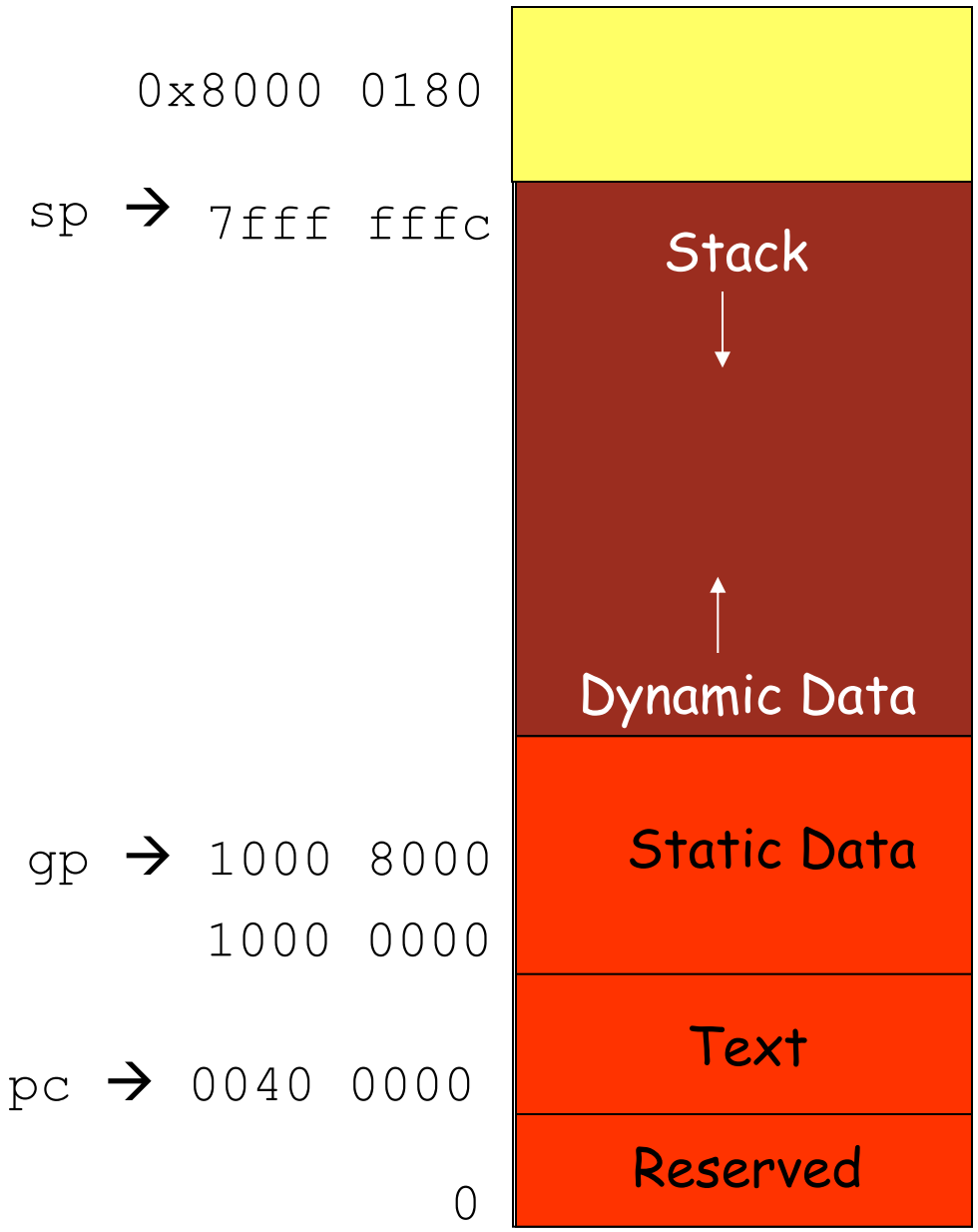
For each cause of the interrupt, the control goes to a different location in memory to handle the exception

Exception type	exception vector address (in hex)
Undefined instruction	0xC0000000
Arithmetic overflow	0xC0000020

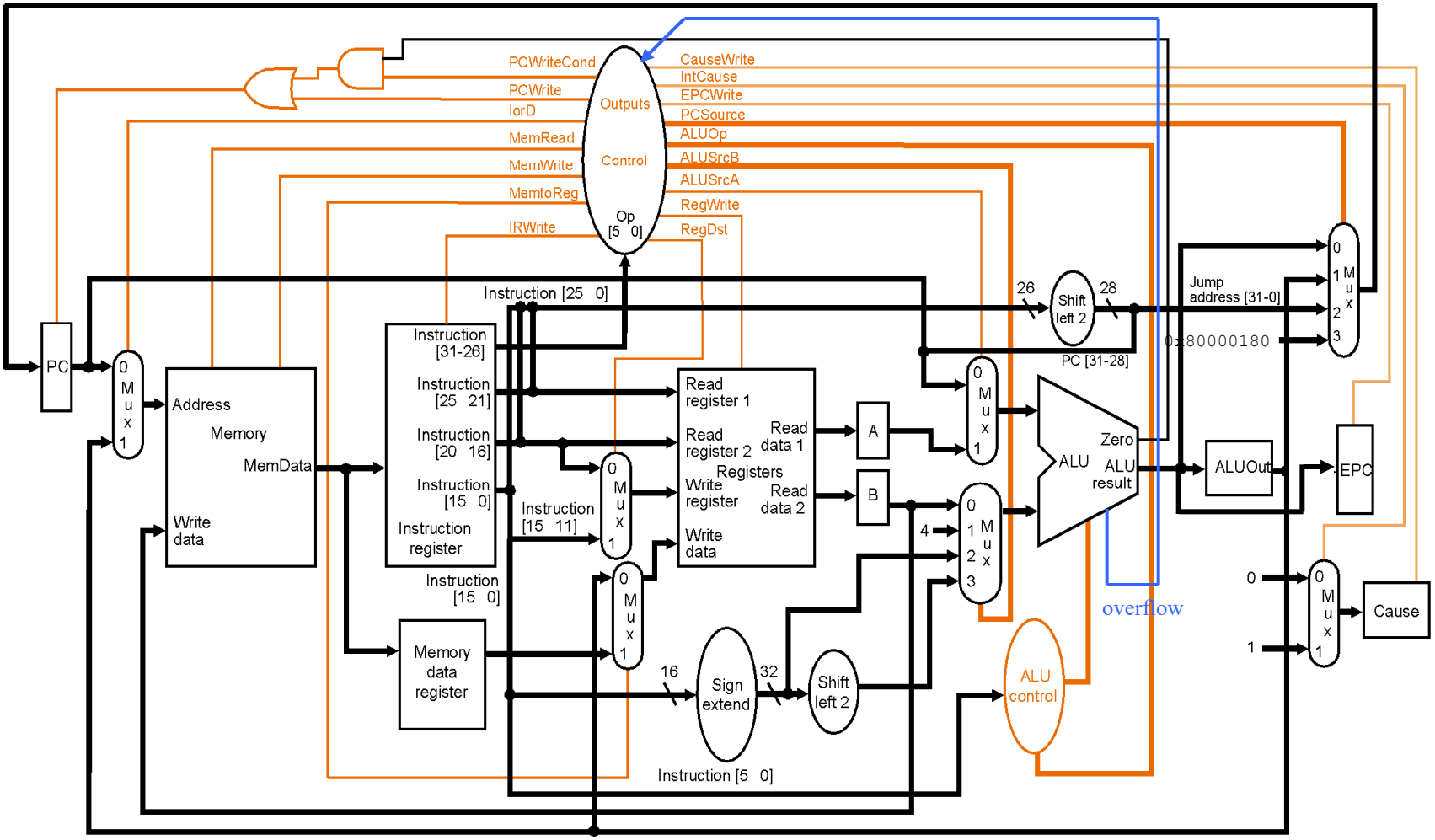
How MIPS Handles Exceptions

- Registers
  - `EPC`
  - `Cause` register: individual bits are used to encode the cause of exceptions (e.g. 0 in the LSB for undefined instruction 1 for overflow)
- Control Signals
  - `CauseWrite`
  - `EPCWrite`
  - `IntCause`: 1-bit signals to set the appropriate bit of `Cause` register
- Exception address:
  - `0x8000 0180` (SPIM uses `0x8000 0080`)

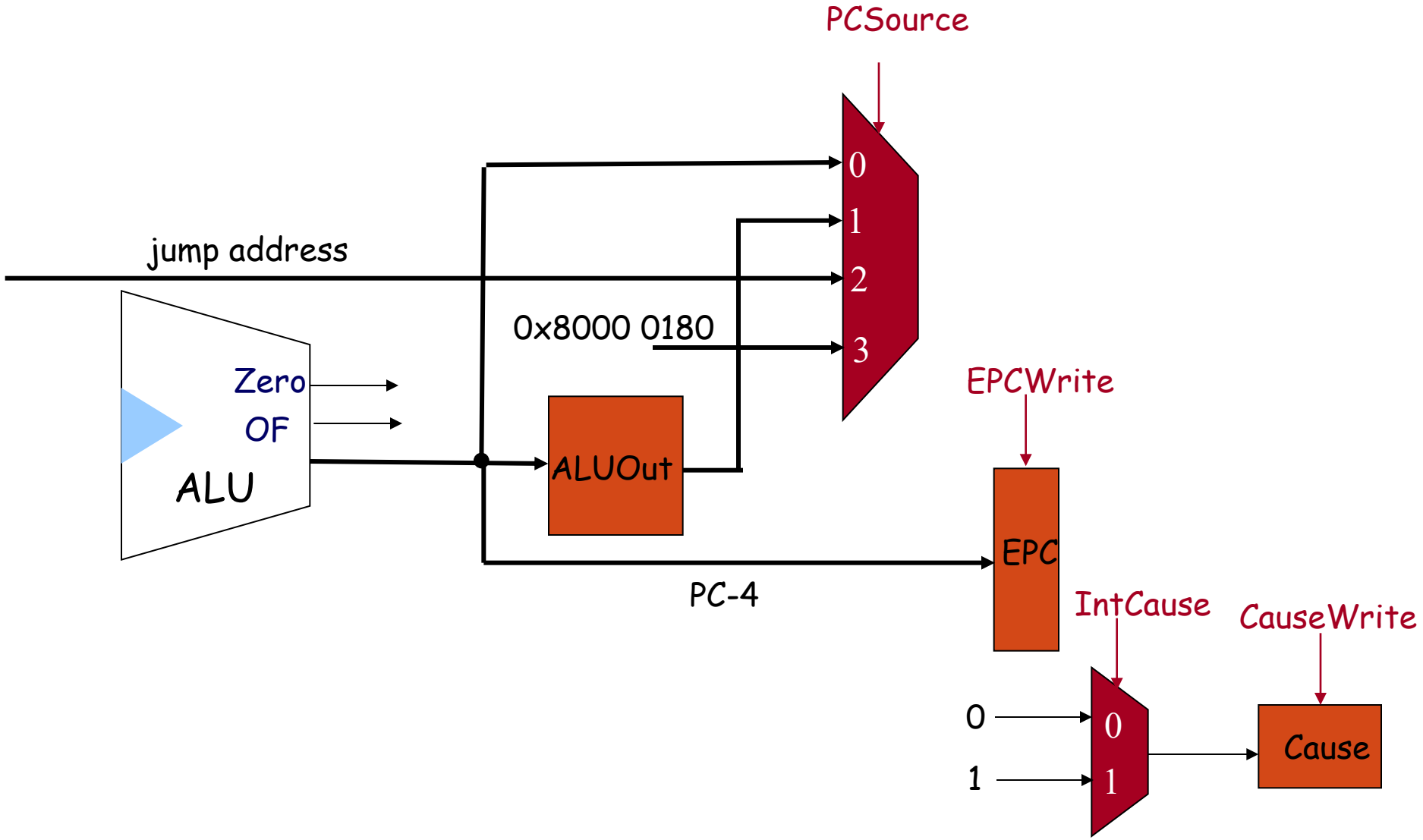
MIPS Memory Allocation



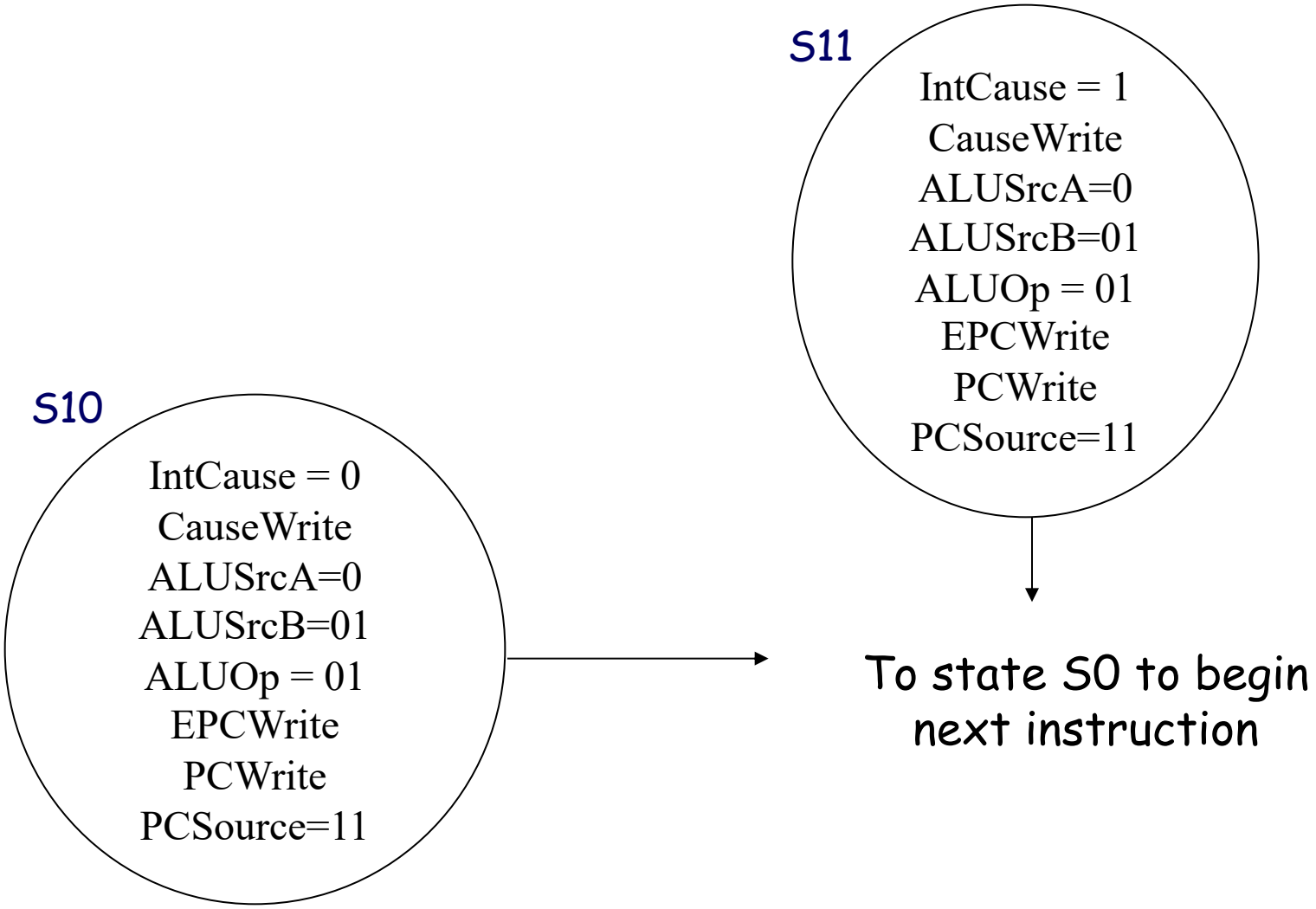
The Multicycle Datapath with Exception Handling



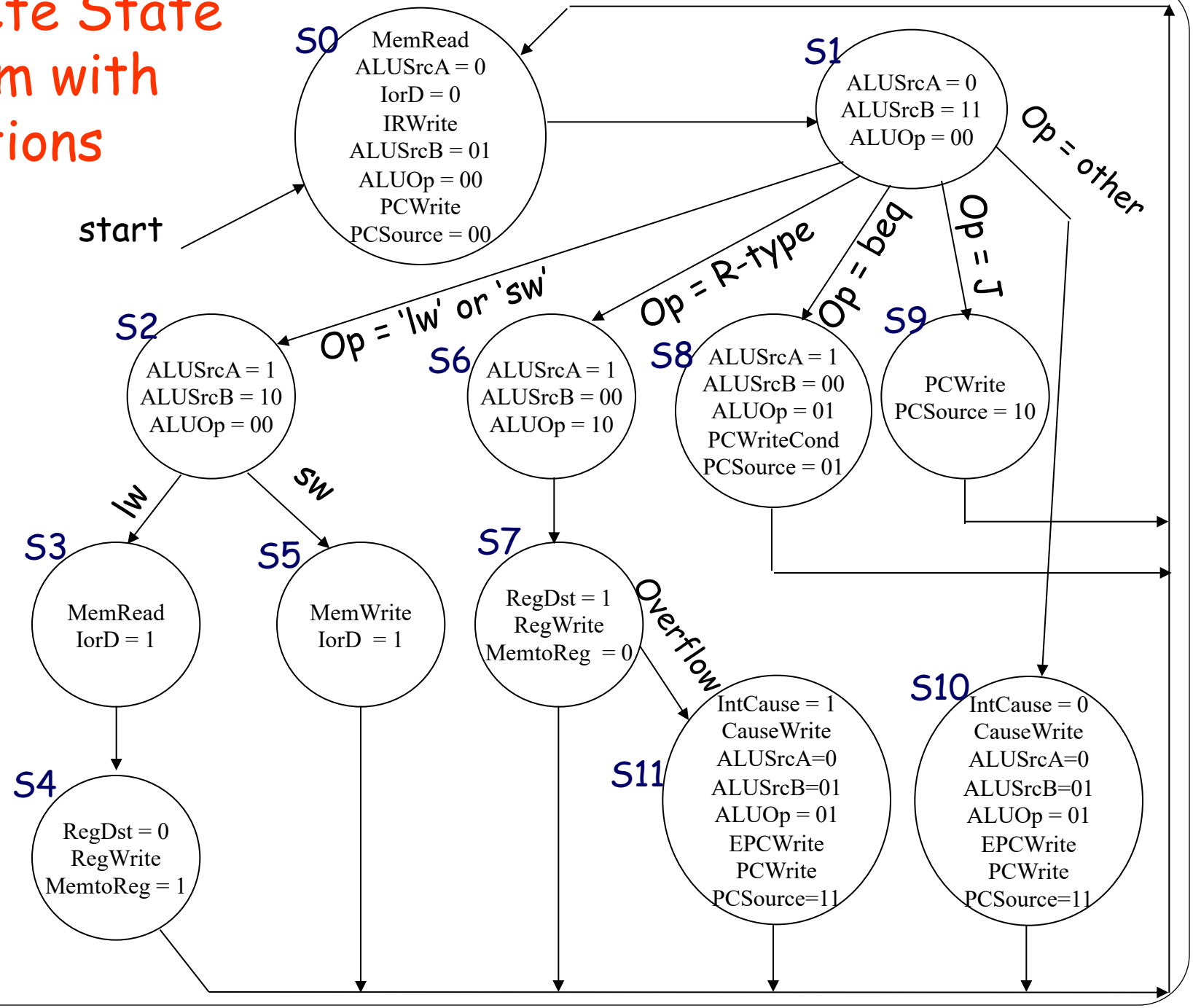
What is New?


PCSource
0
1
2
3
jump address
0x8000 0180
Zero
OF
ALU
ALUOut
EPCWrite
EPC
PC-4
IntCause
CauseWrite
0
1
0
1
Cause

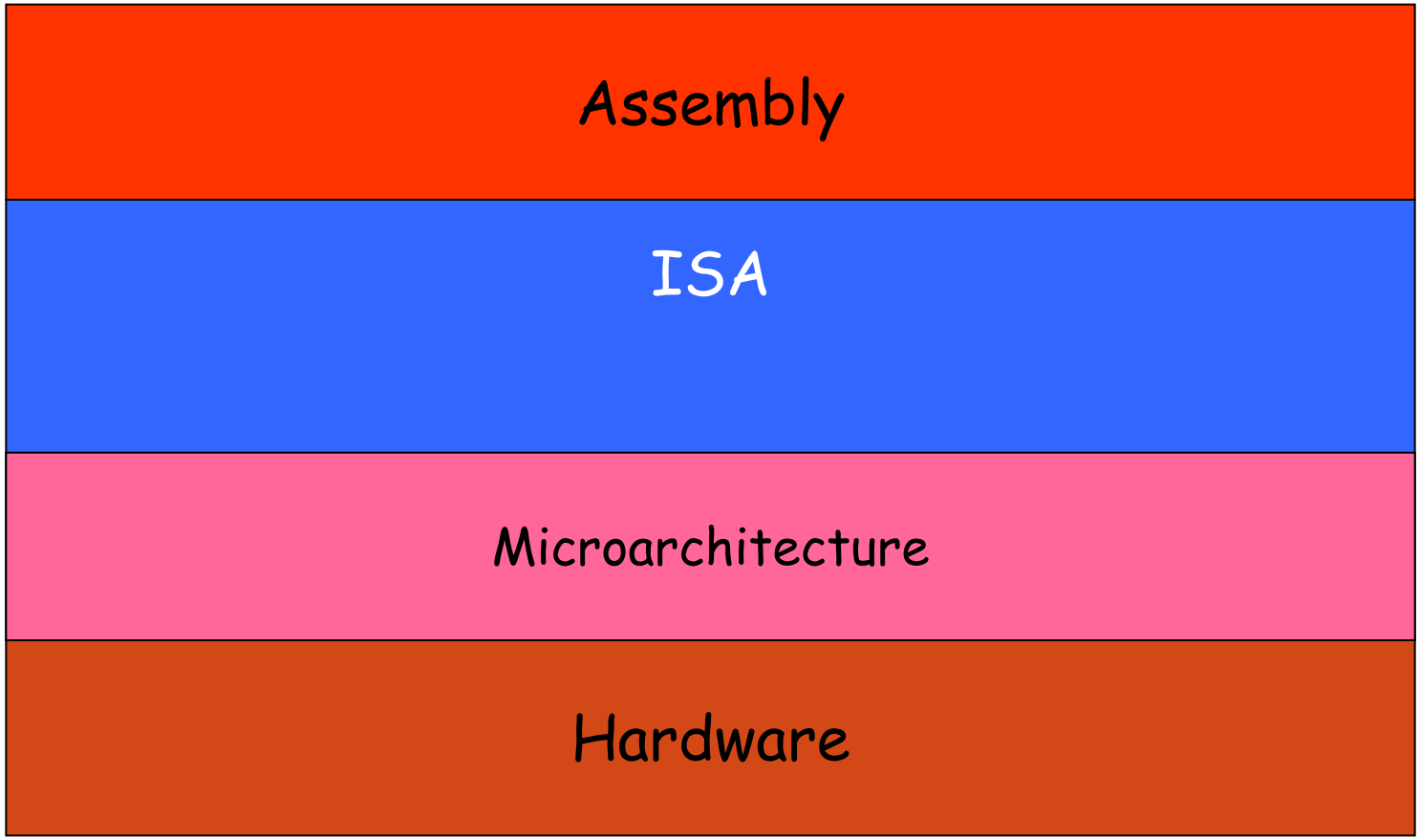
Exception States



Complete State Diagram with Exceptions

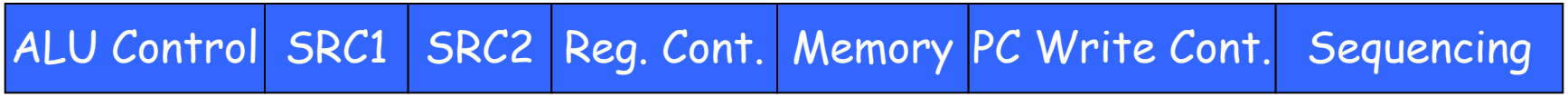


Hardware/Software Interface



Control with Microprogramming

- Simpler instructions to implement (macro) instructions
- Microinstructions are kept in ROM
  - addressable
- Not visible to programmers
- Microinstruction format



Control signals are directly embedded in microinstructions

Fields of Microinstructions

Field name	Function of field
ALU Control	Specify the operation being performed by the ALU during this clock, the result is always written in ALUOut
SRC1	Specify the source for the first ALU operand
SRC2	Specify the source for the second ALU operand
Register Control	Specify read or write for the register file, and the source of the value for a write
Memory	Specify read or write, and the source for the memory. For a read specify the destination register
PCWrite Control	Specify the writing of the PC
Sequencing	Specify how to choose the next microinstruction

Choosing Next Microinstruction

1. Next instruction
   - **`Seq`** (sequential behavior)
2. Branch to the microinstruction that starts execution of the next MIPS instruction
   - **`Fetch`** is the label of the initial microinstruction
3. Next microinstruction is chosen based on the input (dispatching, dispatch tables)
   - In MIPS, two dispatch tables: one from S1 and the other from S2.
   - **`Dispatch i`**

Microprogram for the Control Unit

	Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
0	Fetch	Add	PC	4		Read PC	ALU	Seq
1		Add	PC	Extshft	Read			Dispatch 1
2								
3								
4								
5								
6								
7								
8								
9								

Dispatch Tables

Microcode dispatch table 1		
Opcode field	Opcode name	Value
000000 (0)	R-format	Rformat1
000010 (2)	jmp	JUMP1
000100 (4)	beq	BEQ1
100011 (35)	lw	Mem1
101011 (43)	sw	Mem1

Microcode dispatch table 2		
Opcode field	Opcode name	value
100011 (35)	lw	LW2
101011 (43)	sw	SW2

Microprogram for the Control Unit

	Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
0	Fetch	Add	PC	4		Read PC	ALU	Seq
1		Add	PC	Extshft	Read			Dispatch 1
2	Mem1	Add	A	Extend				Dispatch 2
3	LW2					Read ALU		Seq
4					Write MDR			Fetch
5	SW2					Write ALU		Fetch
6	Rformat1	Func code	A	B				Seq
7					Write ALU			Fetch
8	BEQ1	Sub	A	B			ALUOut-cond	Fetch
9	JUMP1						Jump address	Fetch

Implementing Microprogram