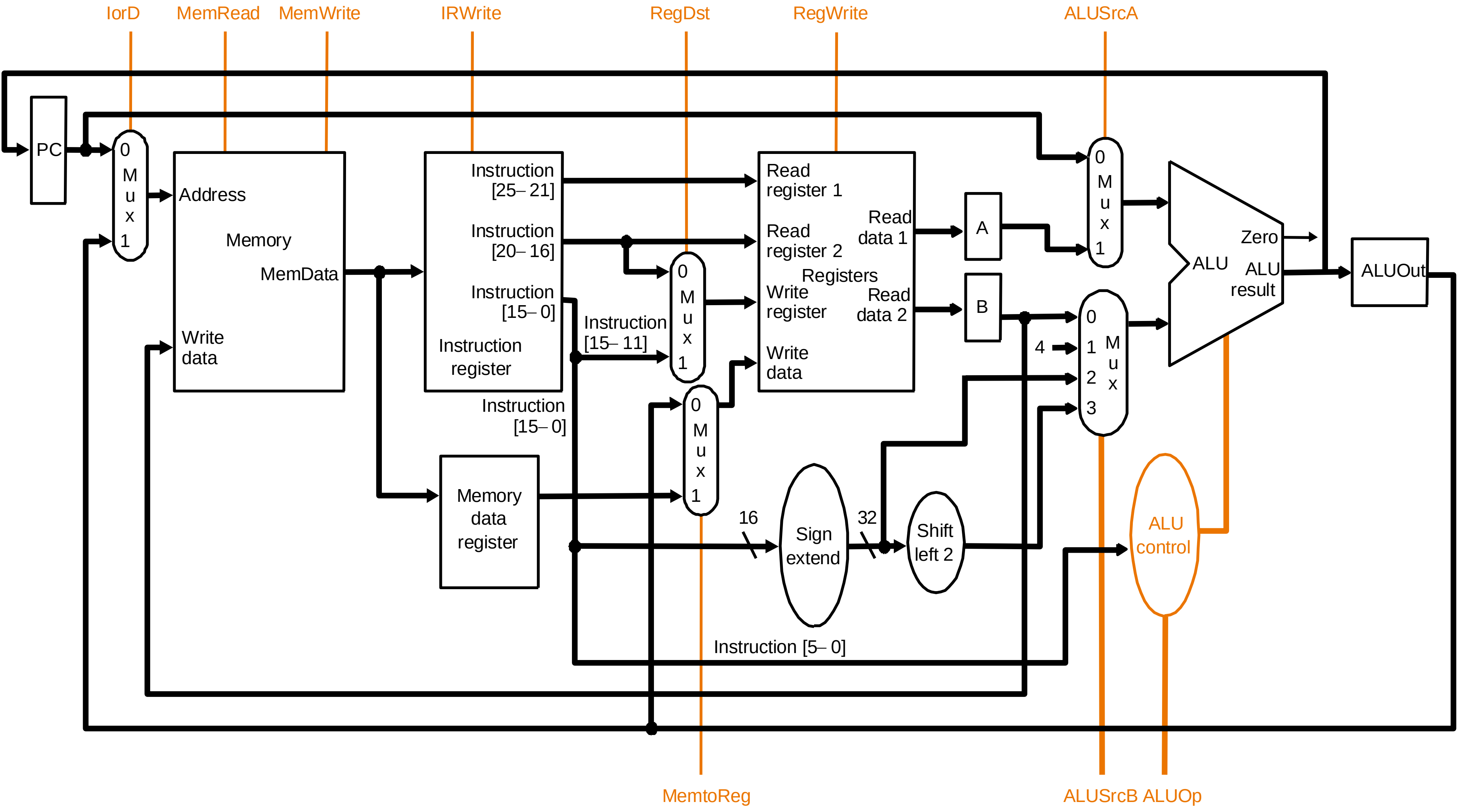
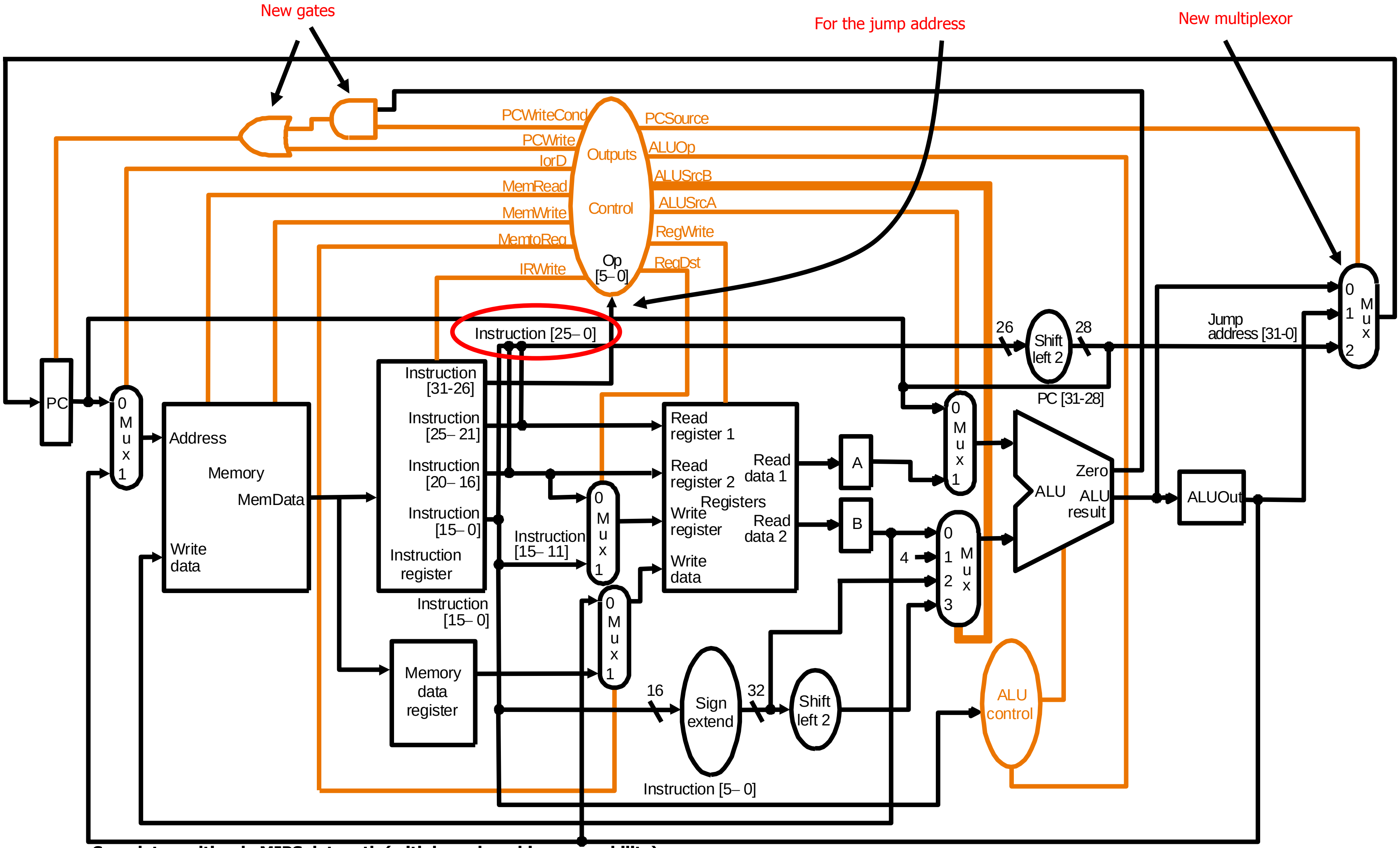
Multicycle Datapath with Control I



… with control lines and the ALU control block added – *not all* control lines are shown

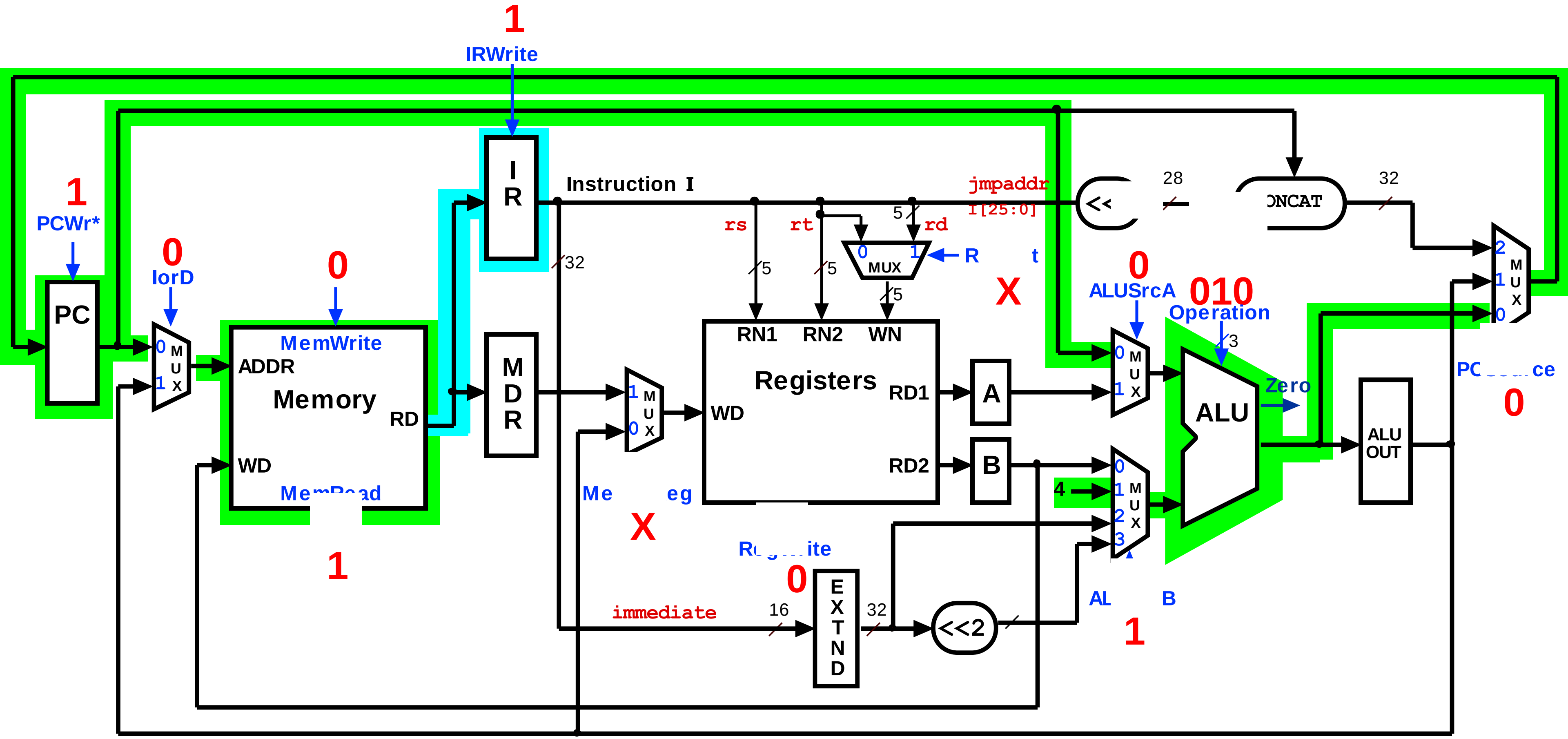
Multicycle Datapath with Control II



Complete multicycle MIPS datapath (with branch and jump capability) and showing the main control block and all control lines

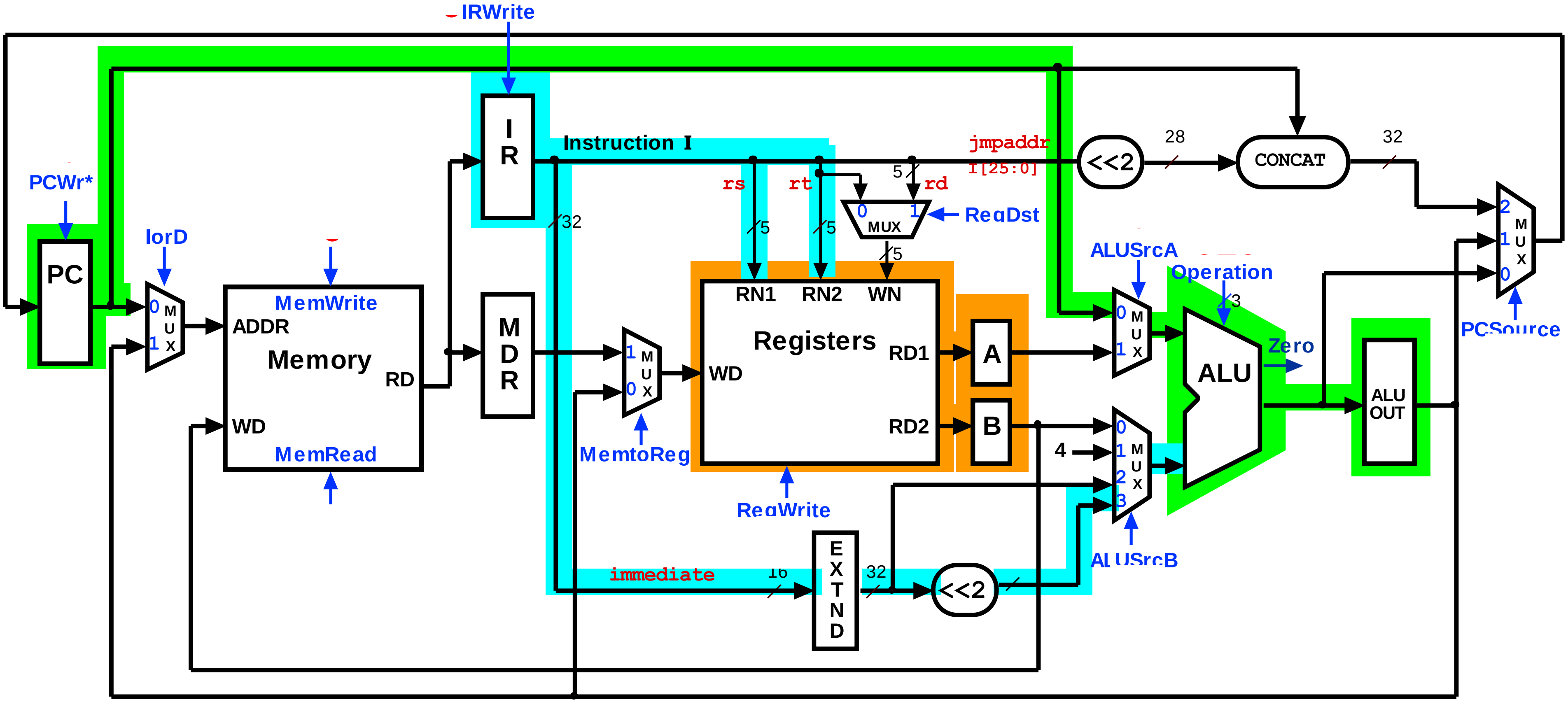
Multicycle Control Step (1): Fetch

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

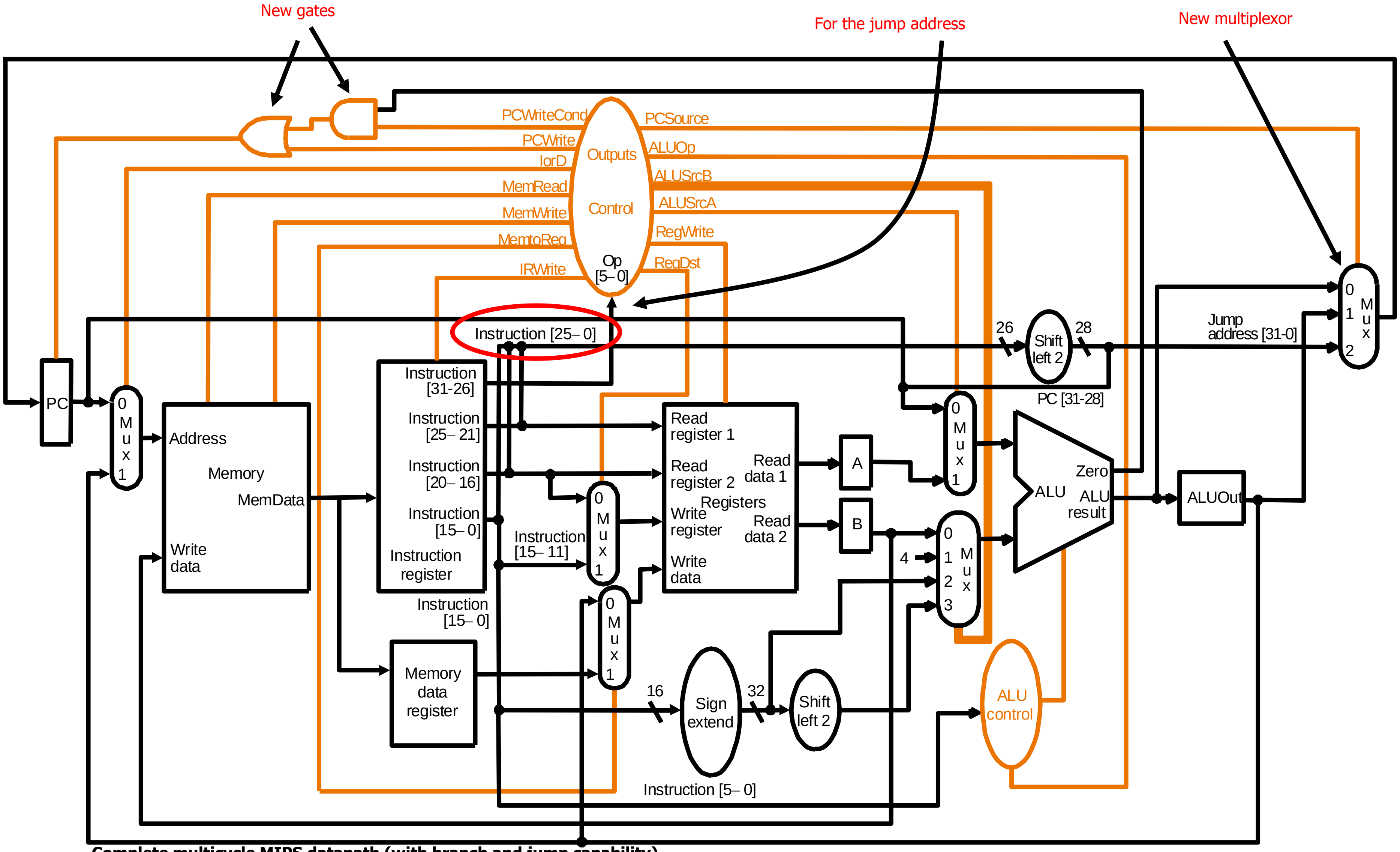


Multicycle Control Step (2): Instruction Decode & Register Fetch

A = Reg[IR[25-21]];   (A = Reg[rs])
B = Reg[IR[20-15]];   (B = Reg[rt])
ALUOut = (PC + sign-extend(IR[15-0]) << 2);



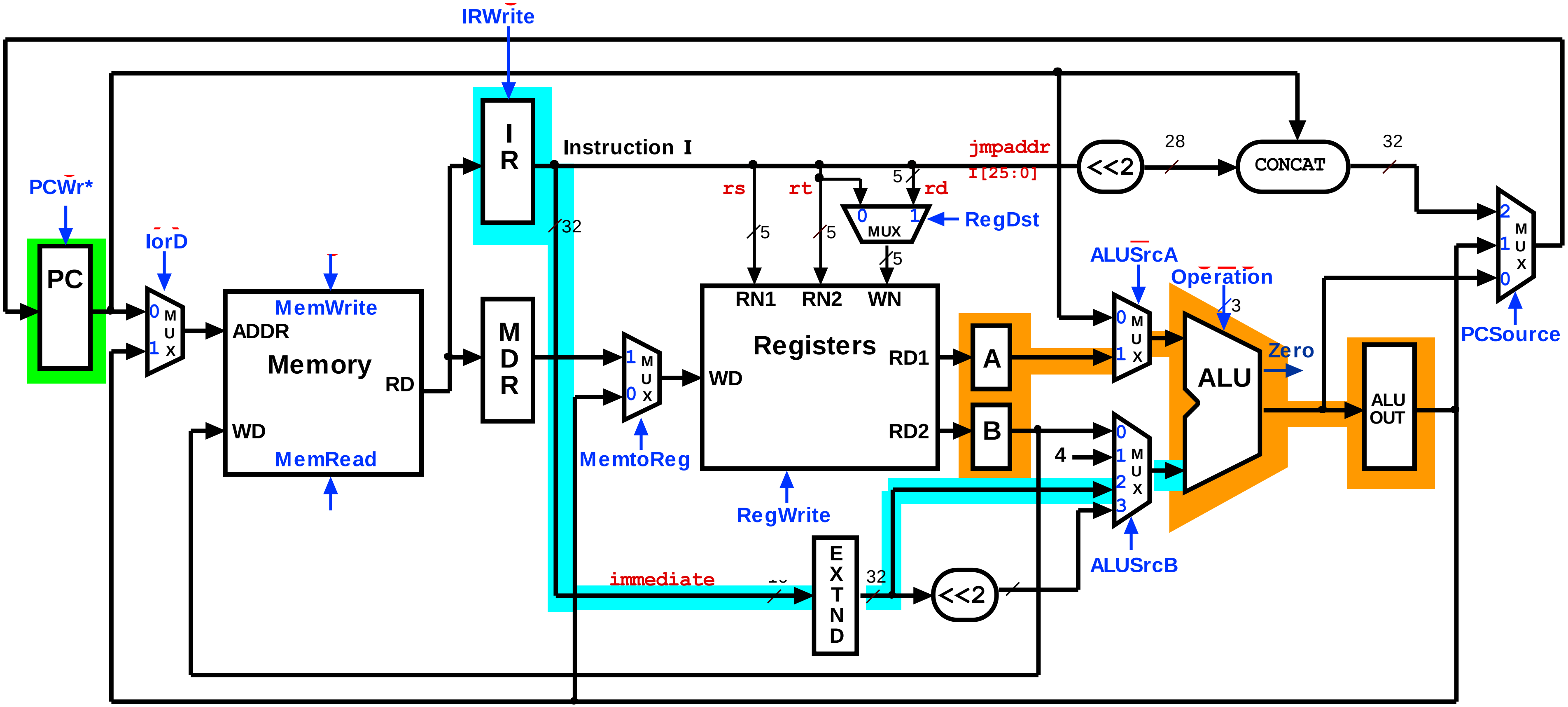
Multicycle Datapath with Control II



Complete multicycle MIPS datapath (with branch and jump capability) and showing the main control block and all control lines

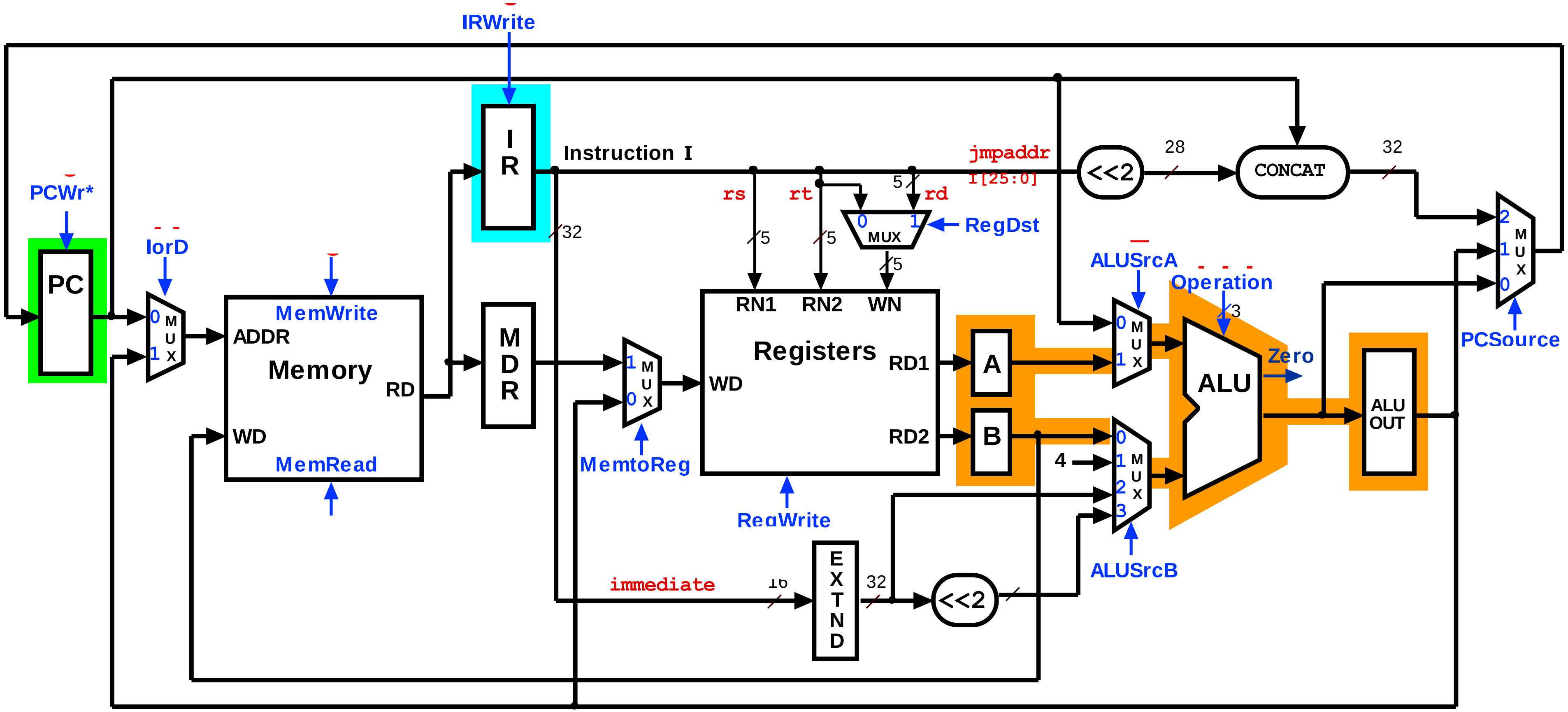
Multicycle Control Step (3): Memory Reference Instructions

ALUOut = A + sign-extend(IR[15-0]);



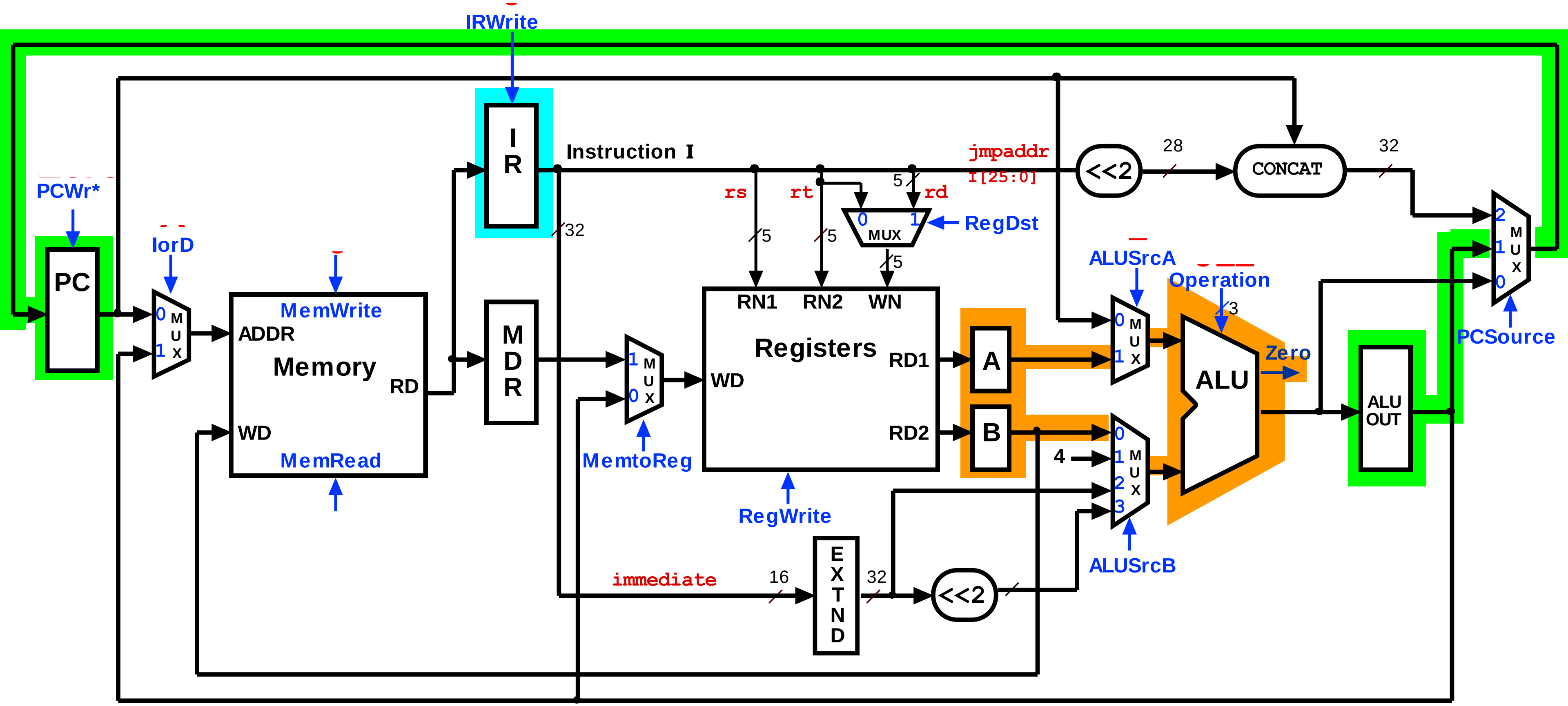
Multicycle Control Step (3): ALU Instruction (R-Type)

ALUOut = A op B;



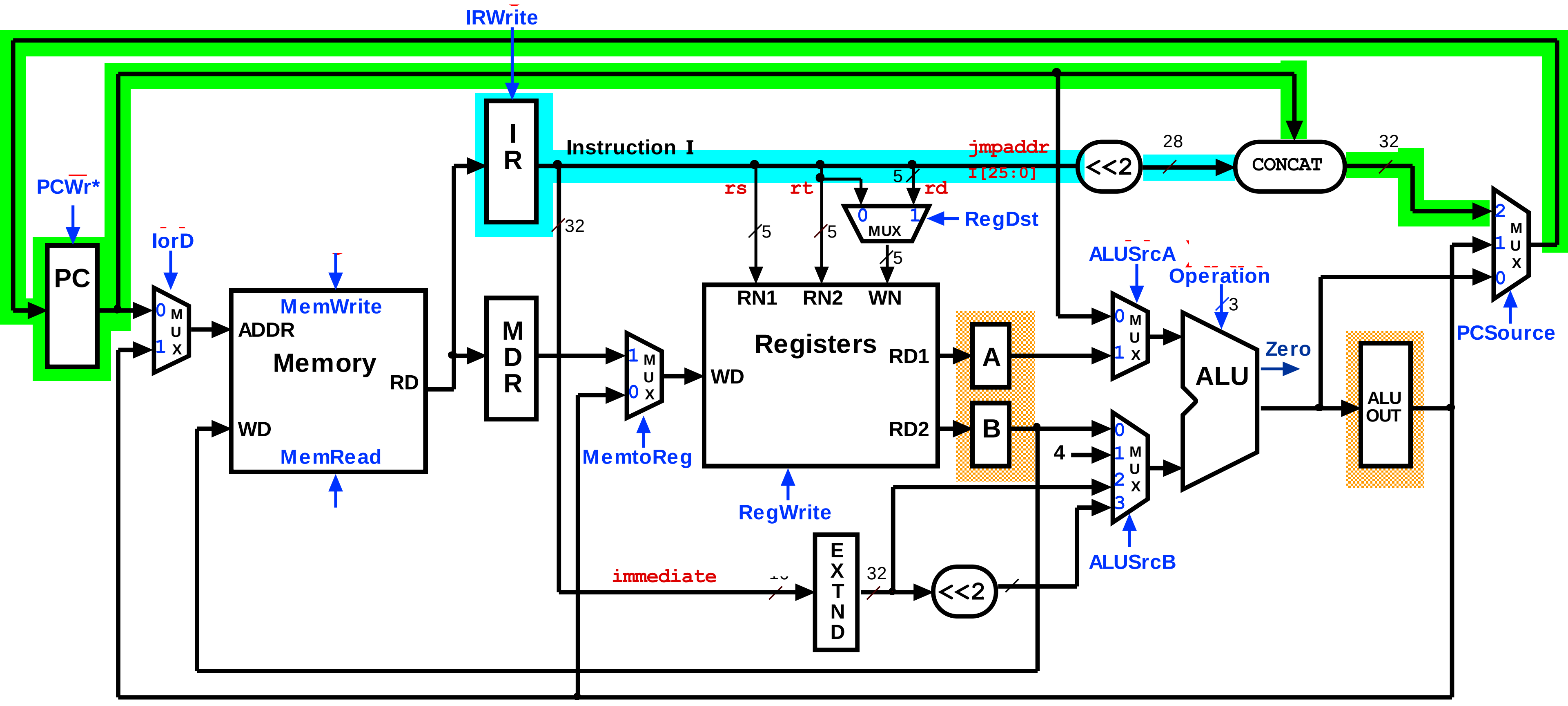
Multicycle Control Step (3): Branch Instructions

if (A == B) PC = ALUOut;



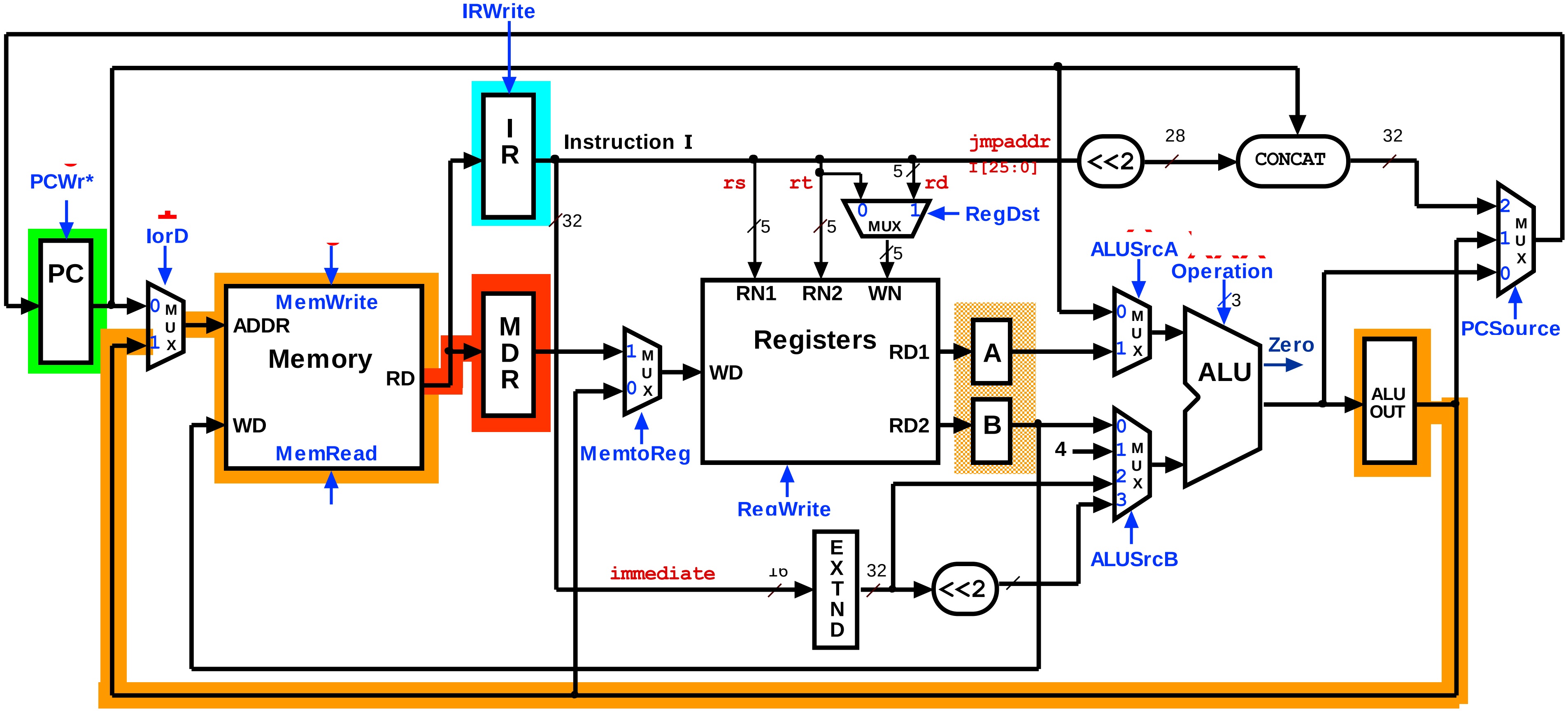
Multicycle Execution Step (3): Jump Instruction

PC = PC[21-28] concat (IR[25-0] << 2);



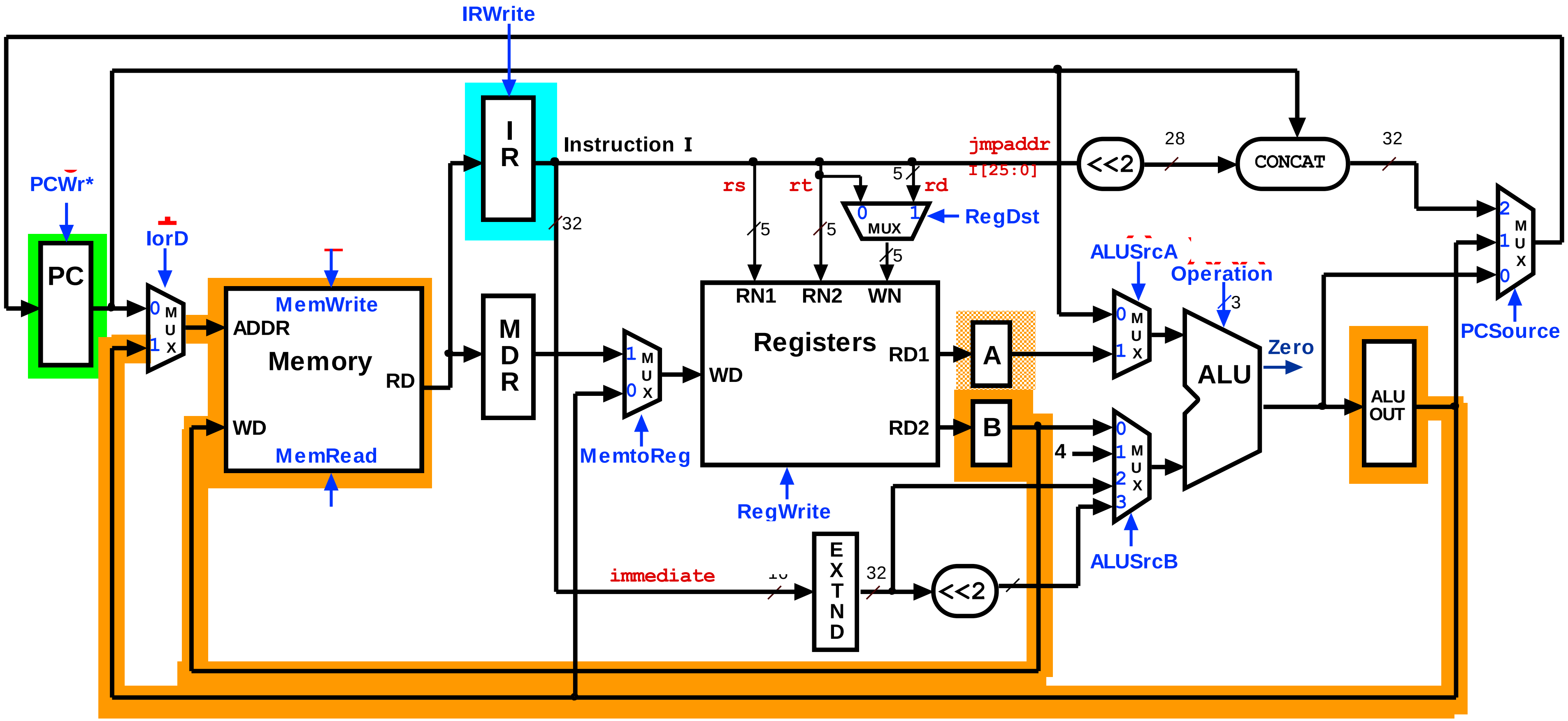
Multicycle Control Step (4): Memory Access - Read (`lw`)

MDR = Memory[ALUOut];



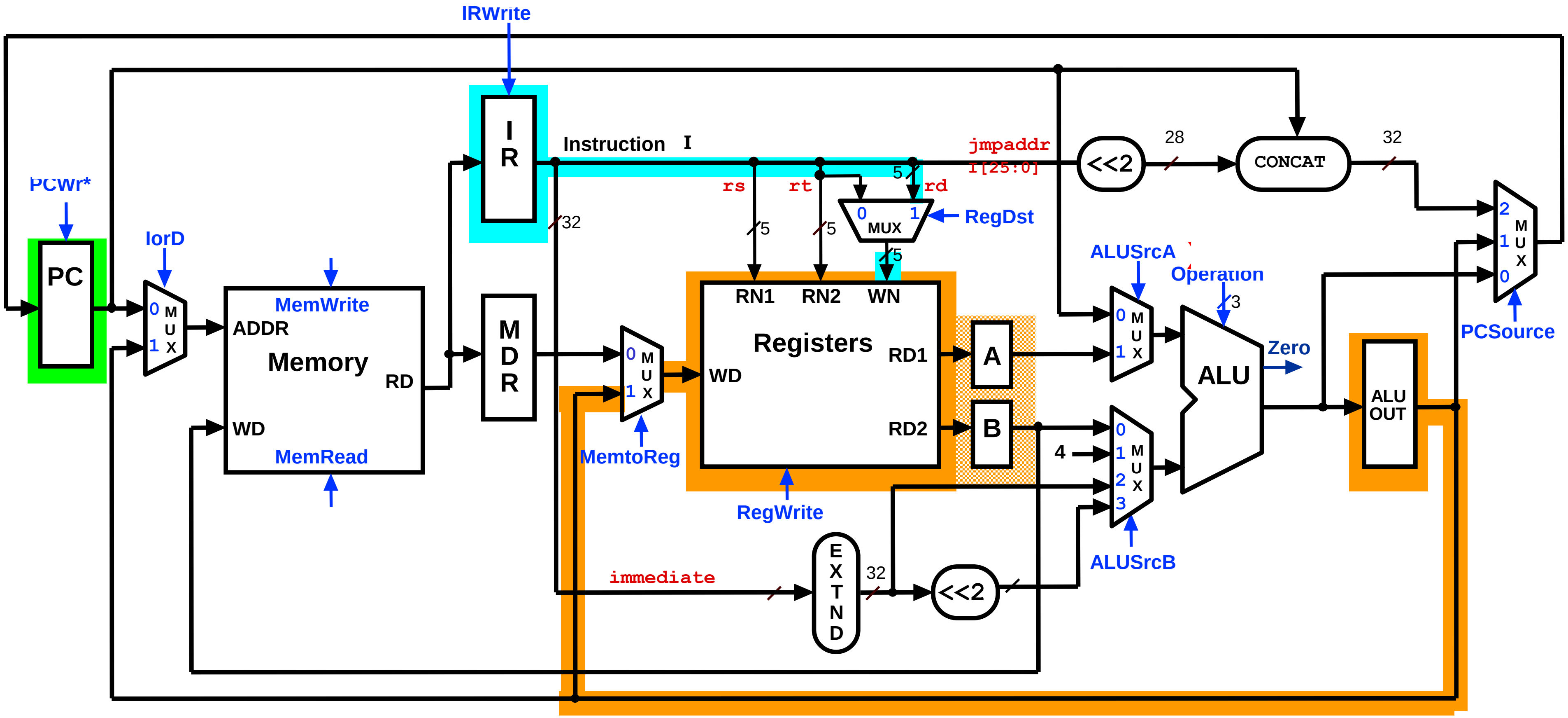
Multicycle Execution Steps (4) Memory Access - Write (sw)

Memory[ALUOut] = B;



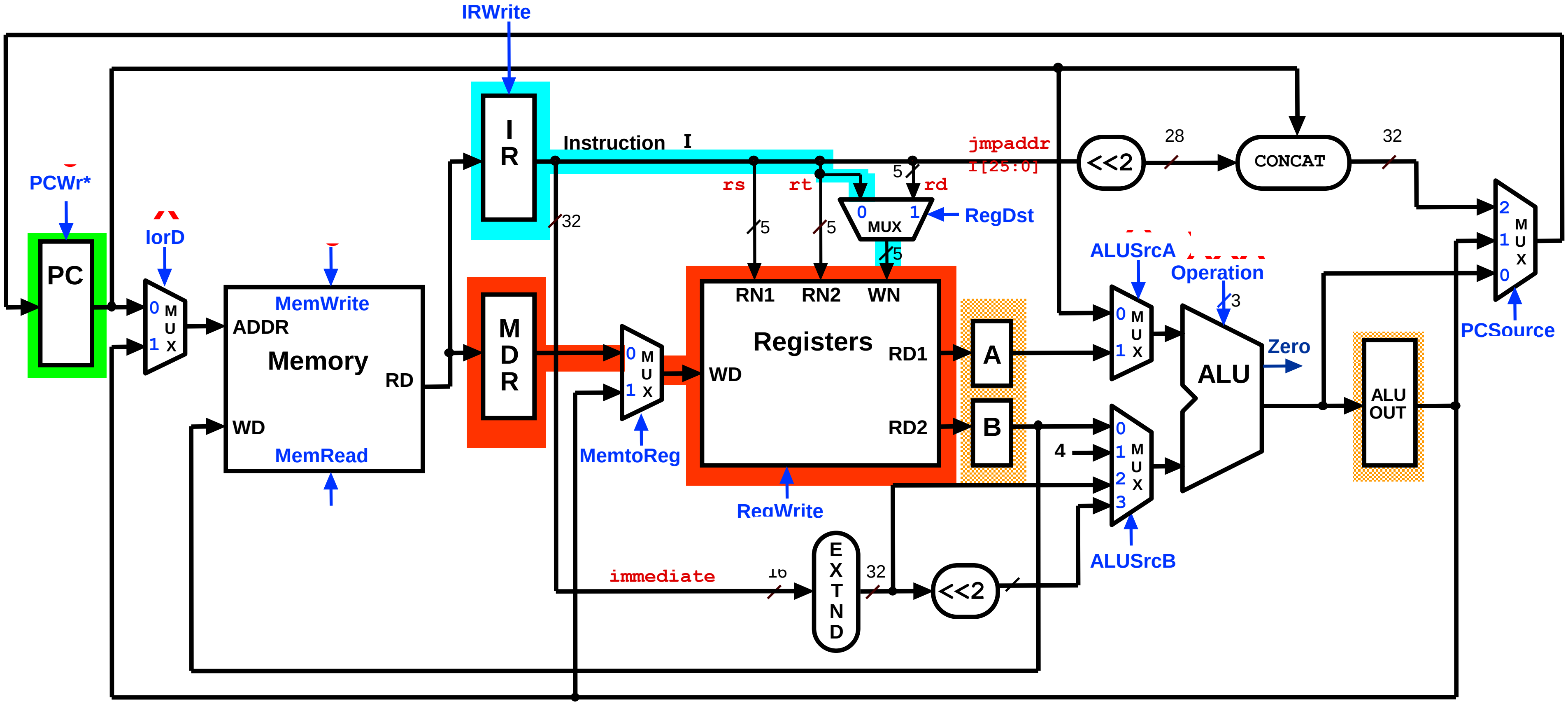
Multicycle Control Step (4): ALU Instruction (R-Type)

(Reg[Rd] = ALUOut)



Multicycle Execution Steps (5) Memory Read Completion (lw)

Reg[IR[20-16]] = MDR;



Implementing Control

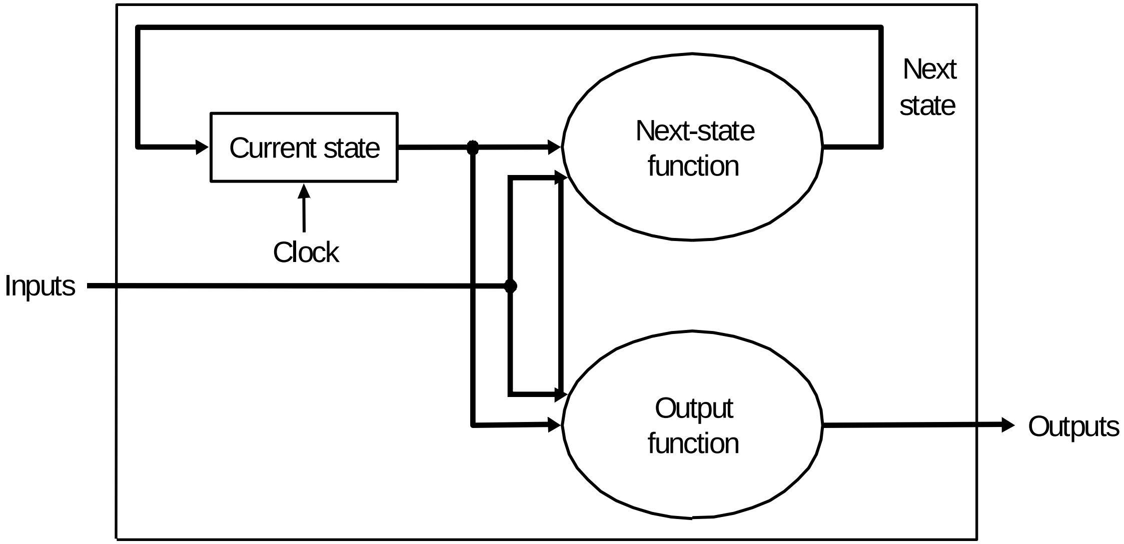
- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed

- Use the information we have accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming

- Implementation is then derived from the specification

Review: Finite State Machines

- Finite state machines (FSMs):
  - a set of states and
  - next state function, determined by current state and the input
  - output function, determined by current state and possibly input



Next state

Current state

Next-state function

Clock

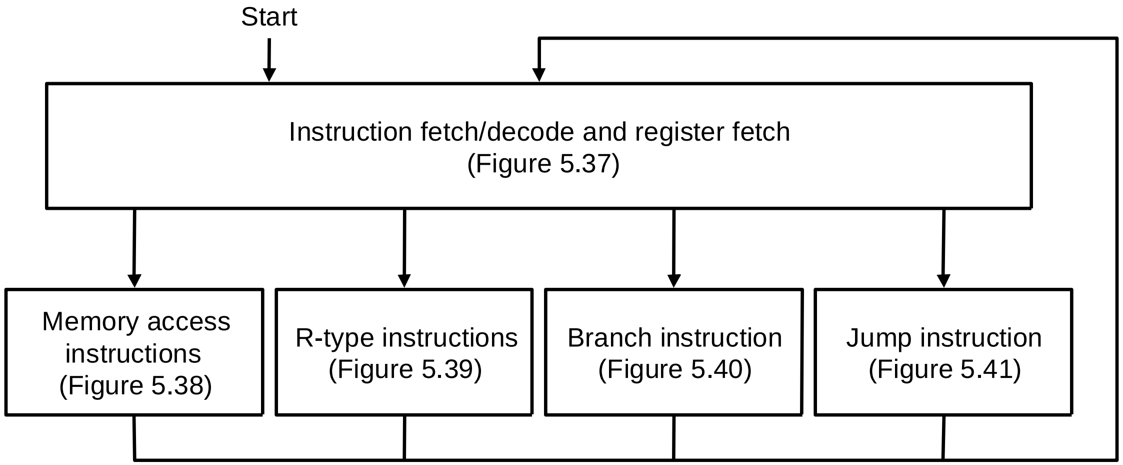
Inputs

Output function

Outputs

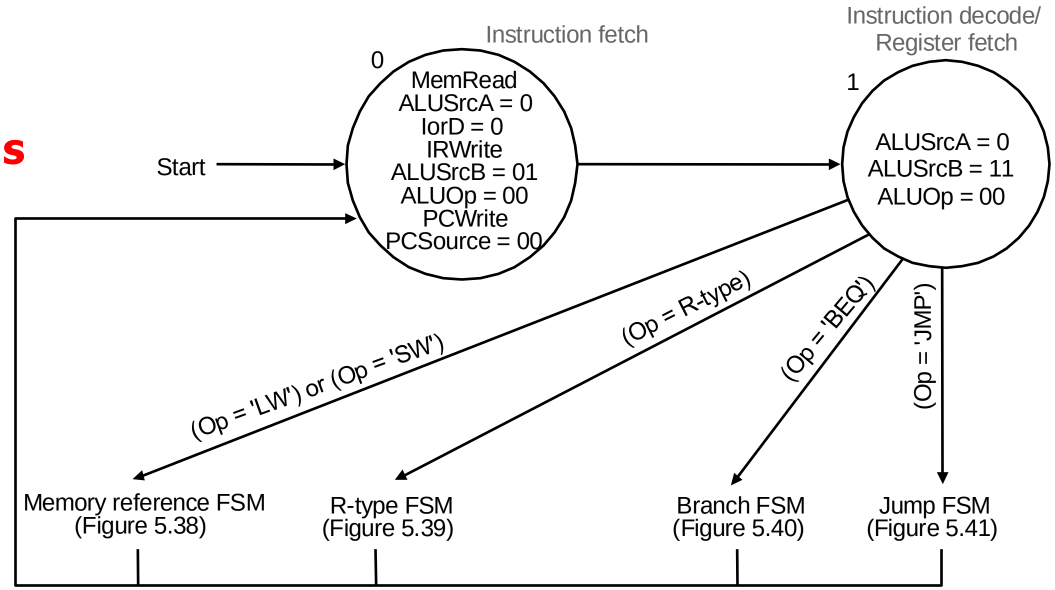
  - We'll use a *Moore machine* – output based *only* on current state

FSM Control: High-level View



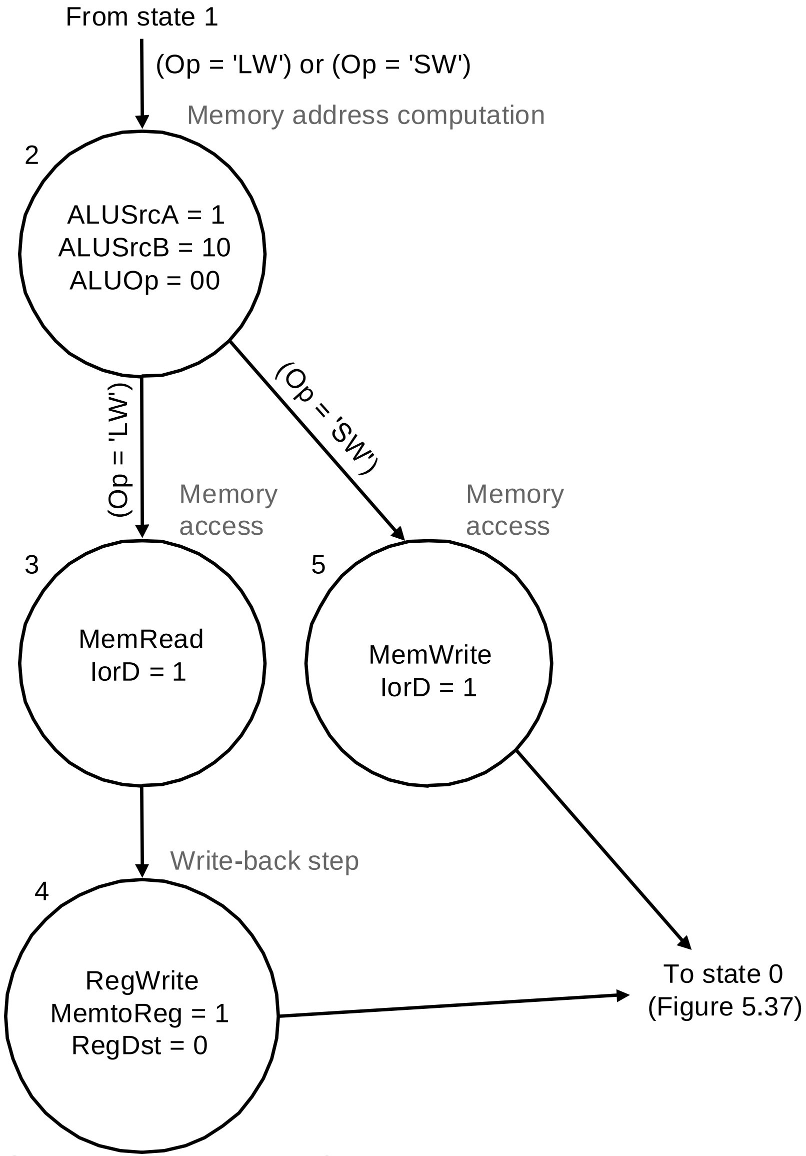
High-level view of FSM control

Asserted signals shown inside state circles



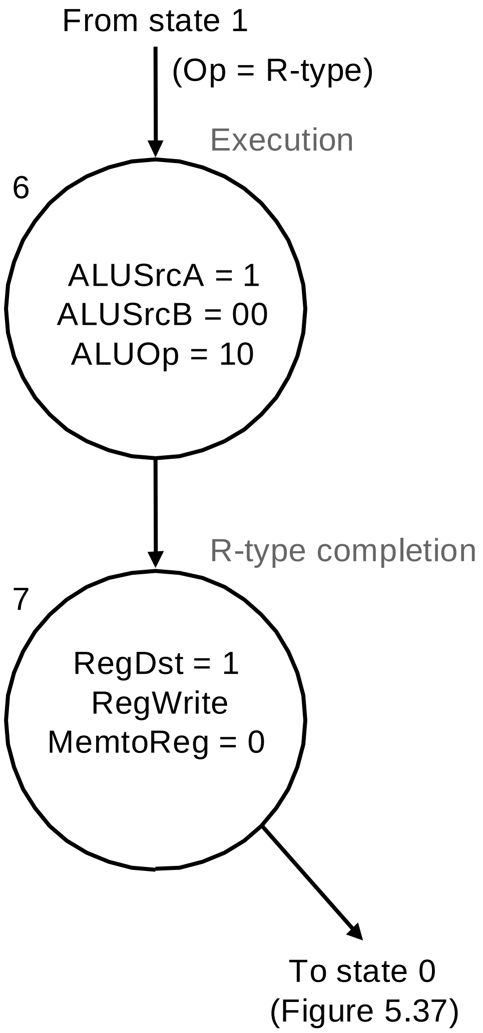
Instruction fetch and decode steps of every instruction is identical

FSM Control: Memory Reference



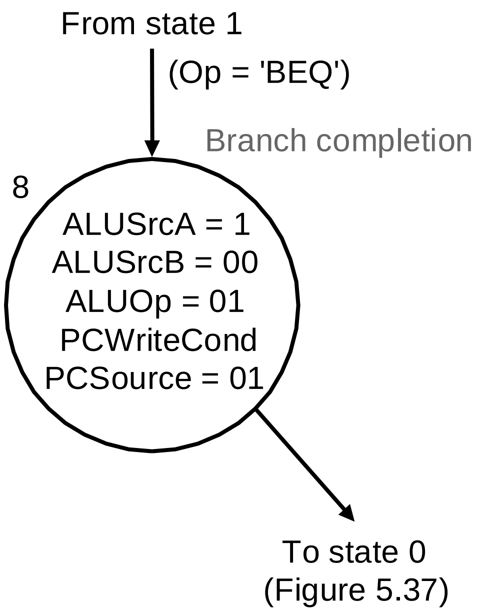
FSM control for memory-reference has 4 states

FSM Control: R-type Instruction



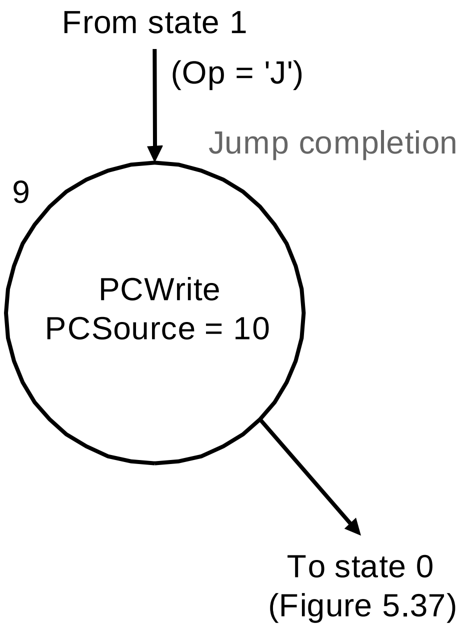
FSM control to implement R-type instructions has 2 states

FSM Control: Branch Instruction



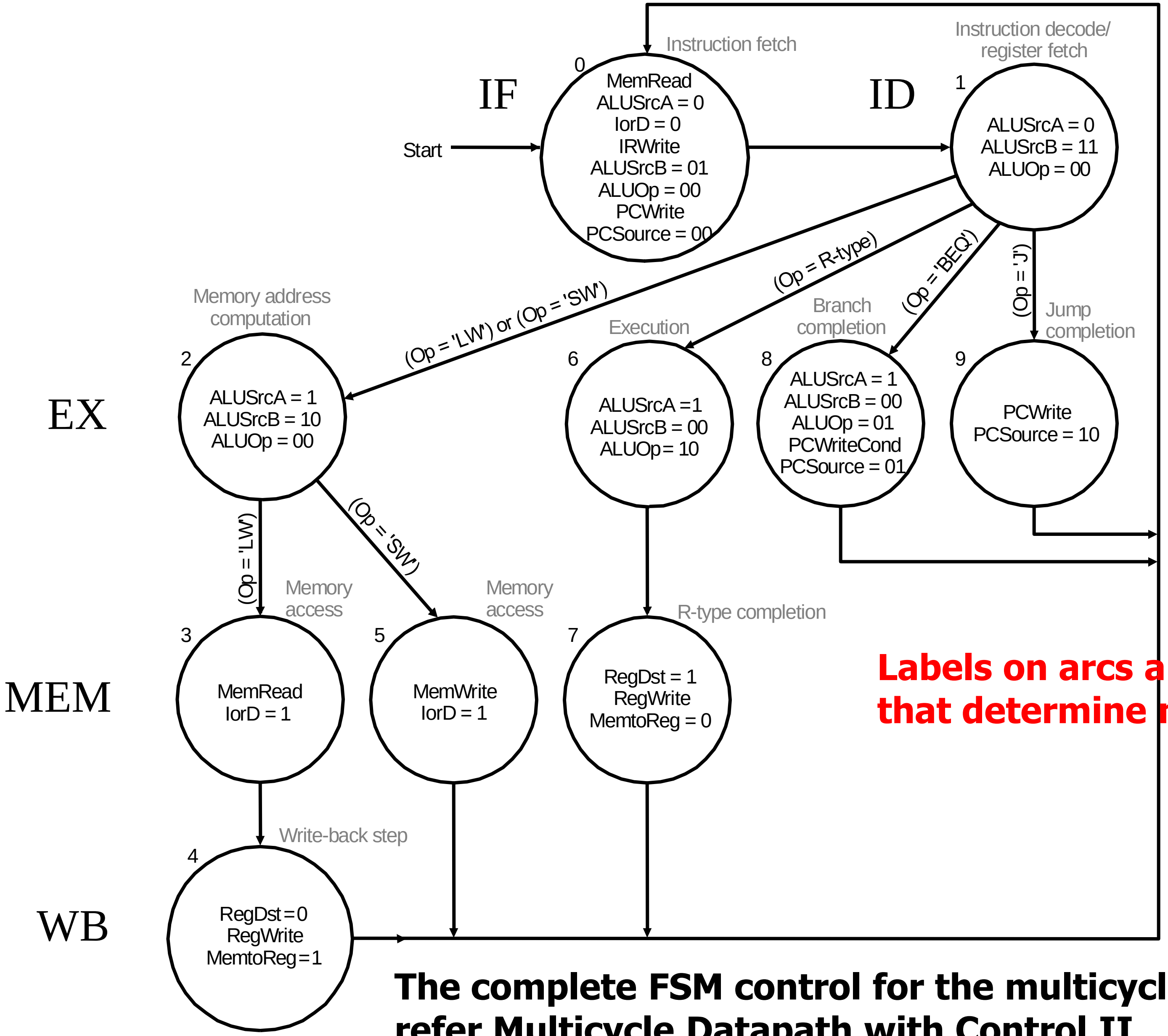
FSM control to implement branches has 1 state

FSM Control: Jump Instruction



FSM control to implement jumps has 1 state

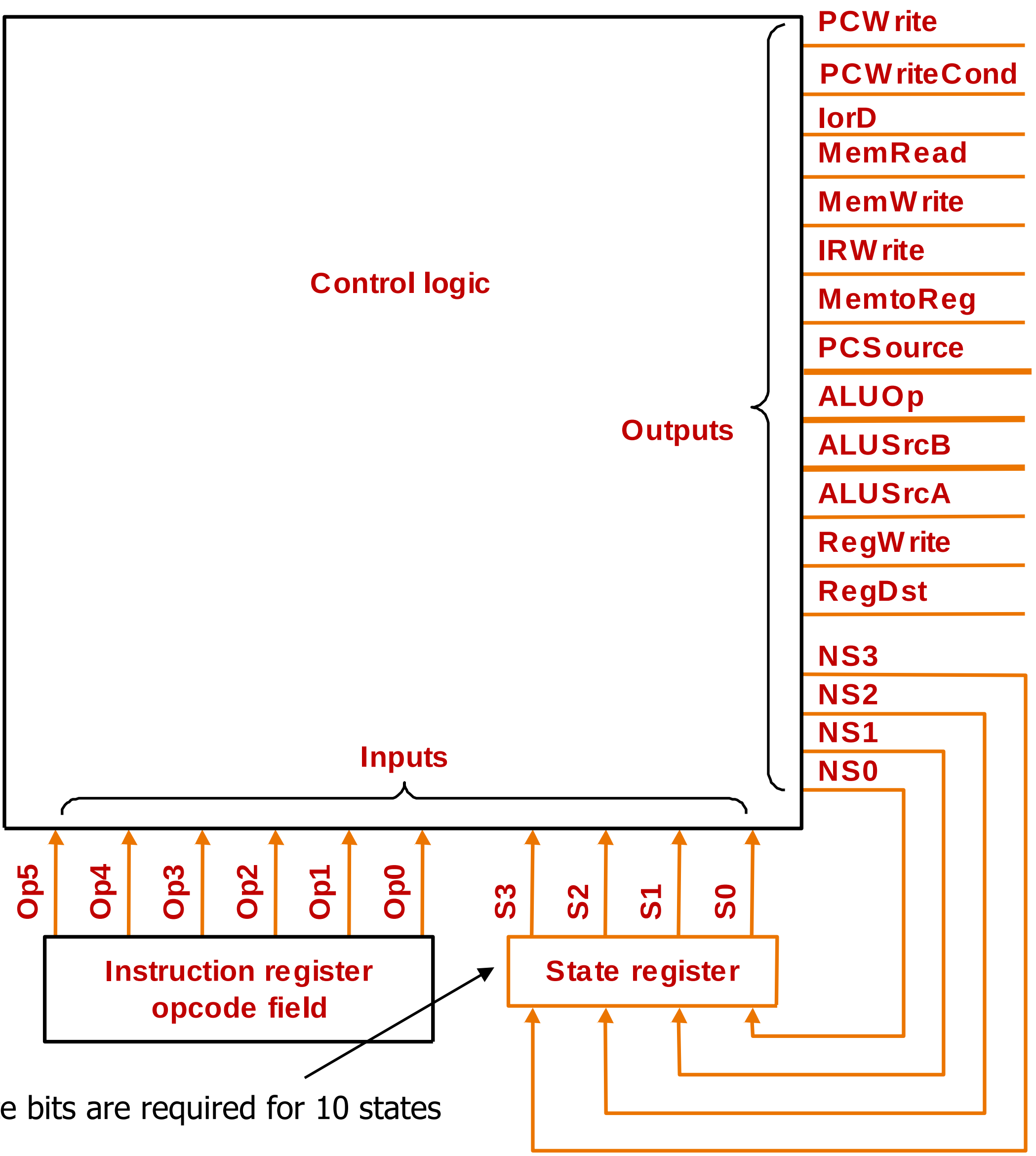
FSM Control: Complete View



Labels on arcs are conditions that determine next state

The complete FSM control for the multicycle MIPS datapath: refer Multicycle Datapath with Control II

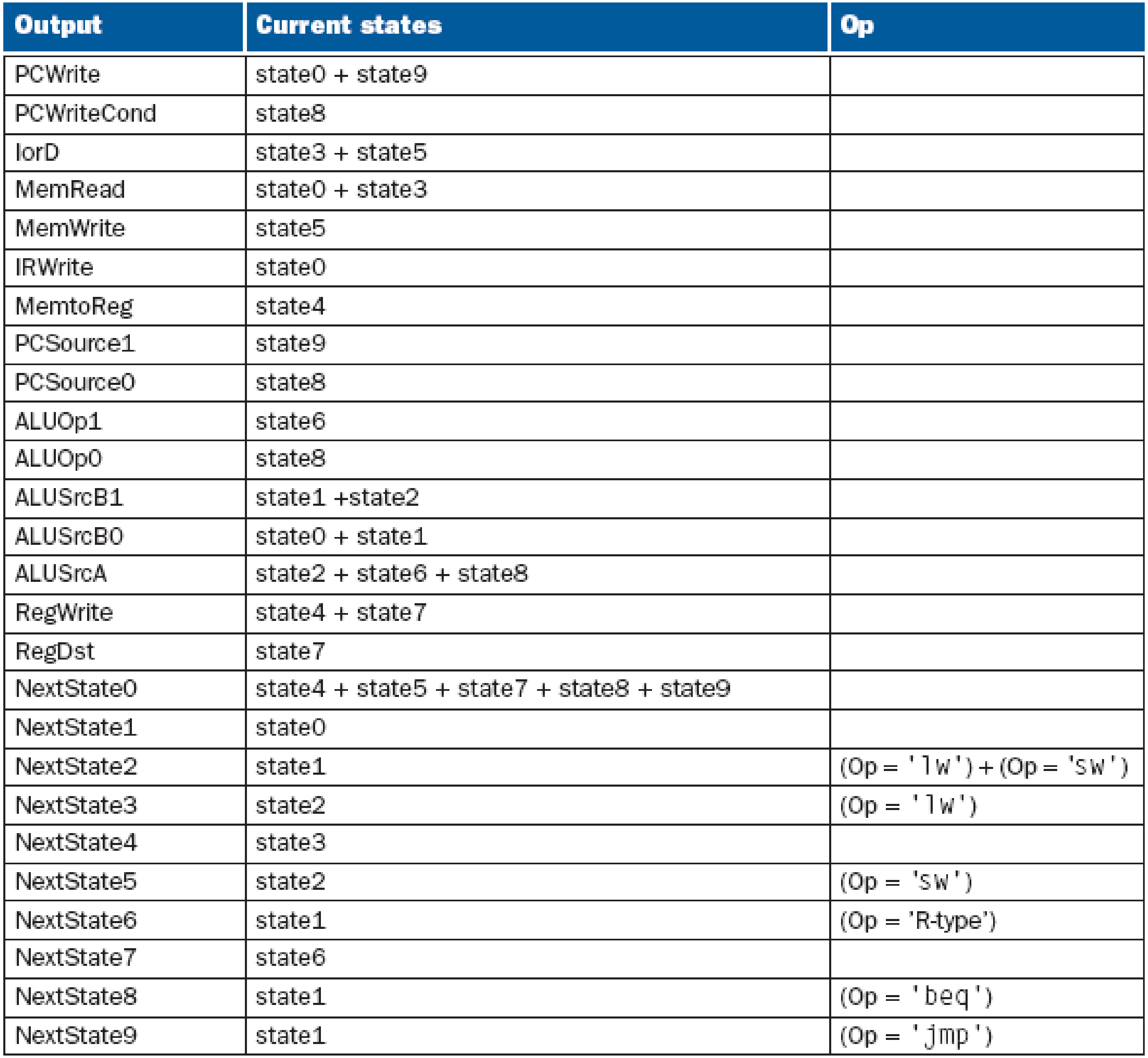
FSM Control: Implement-ation



Four state bits are required for 10 states

High-level view of FSM implementation: inputs to the combinational logic block are the current state number and instruction opcode bits; outputs are the next state number and control signals to be asserted for the current state

Logic Equation for Control Unit



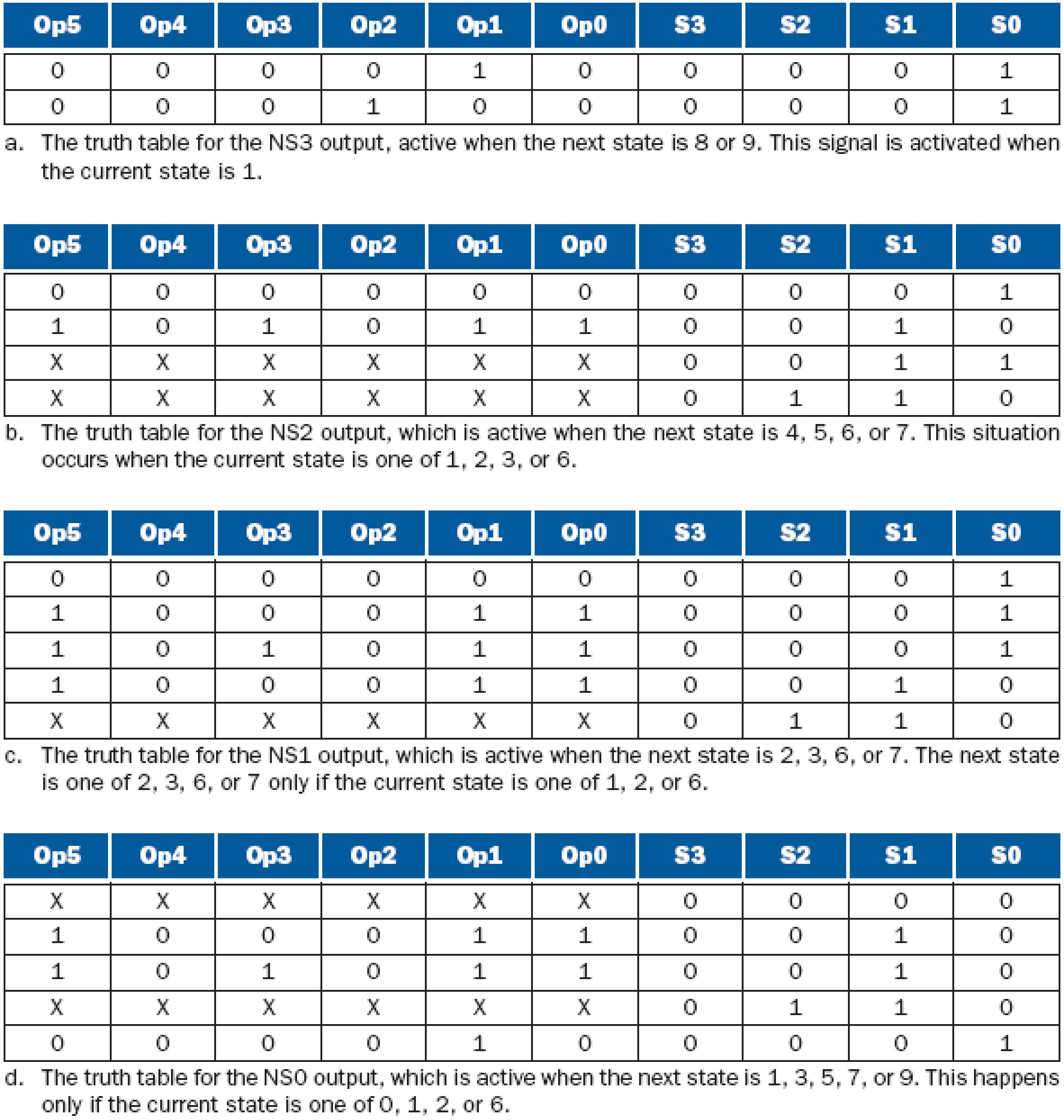
Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 +state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + 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')

Example of Combinational Circuit

- NS0 = NextState1 + NextState3 + NextState5 + NextState7 + NextState9

$$
\begin{aligned}
\mathrm{NextState1} &= \mathrm{State0} = \overline{\mathrm{S3}} \cdot \overline{\mathrm{S2}} \cdot \overline{\mathrm{S1}} \cdot \overline{\mathrm{S0}} \\
\mathrm{NextState3} &= \mathrm{State2} \cdot (\mathrm{Op}[5\text{-}0] = \mathrm{lw}) \\
&= \overline{\mathrm{S3}} \cdot \overline{\mathrm{S2}} \cdot \mathrm{S1} \cdot \overline{\mathrm{S0}} \cdot \mathrm{Op5} \cdot \overline{\mathrm{Op4}} \cdot \overline{\mathrm{Op3}} \cdot \overline{\mathrm{Op2}} \cdot \mathrm{Op1} \cdot \mathrm{Op0} \\
\mathrm{NextState5} &= \mathrm{State\ 2} \cdot (\mathrm{Op}[5\text{-}0] = \mathrm{sw}) \\
&= \overline{\mathrm{S3}} \cdot \overline{\mathrm{S2}} \cdot \mathrm{S1} \cdot \overline{\mathrm{S0}} \cdot \mathrm{Op5} \cdot \overline{\mathrm{Op4}} \cdot \mathrm{Op3} \cdot \overline{\mathrm{Op2}} \cdot \mathrm{Op1} \cdot \mathrm{Op0} \\
\mathrm{NextState7} &= \mathrm{State6} = \overline{\mathrm{S3}} \cdot \mathrm{S2} \cdot \mathrm{S1} \cdot \overline{\mathrm{S0}} \\
\mathrm{NextState9} &= \mathrm{State1} \cdot (\mathrm{Op}[5\text{-}0] = \mathrm{jmp}) \\
&= \overline{\mathrm{S3}} \cdot \overline{\mathrm{S2}} \cdot \overline{\mathrm{S1}} \cdot \mathrm{S0} \cdot \overline{\mathrm{Op5}} \cdot \overline{\mathrm{Op4}} \cdot \overline{\mathrm{Op3}} \cdot \overline{\mathrm{Op2}} \cdot \mathrm{Op1} \cdot \overline{\mathrm{Op0}}
\end{aligned}
$$

Truth Table for Next-State



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

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

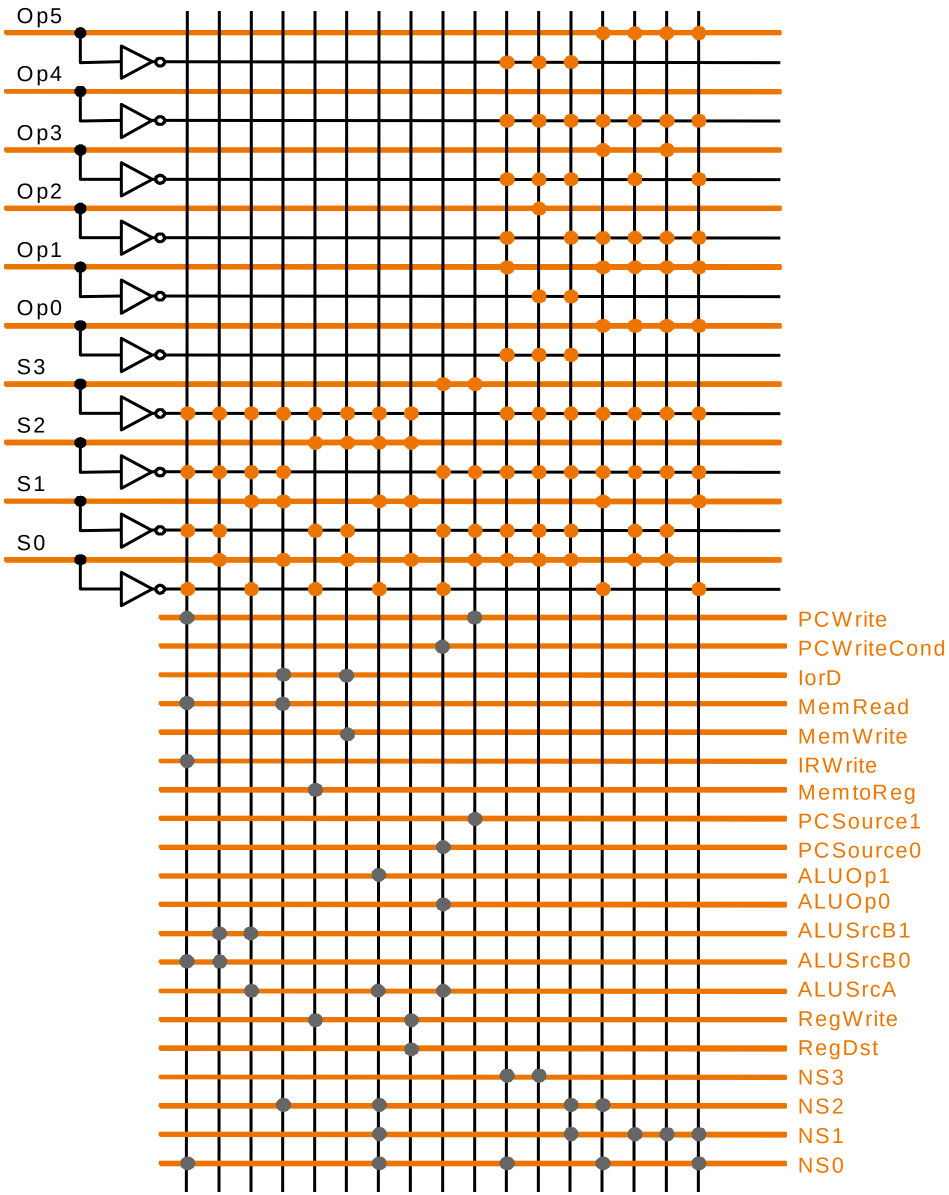
s3	s2	s1	s0
0	1	0	0
0	1	1	1

o. Truth table for RegWrite

s3	s2	s1	s0
0	1	1	1

p. Truth table for RegDst

FSM Control: PLA Implementation

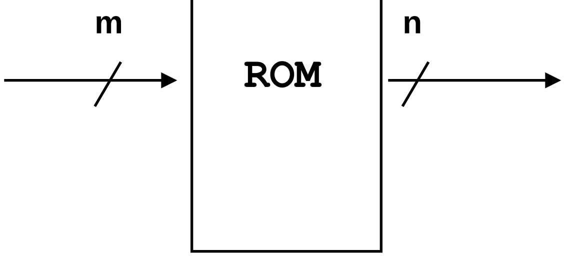


Upper half is the AND plane that computes all the products. The products are carried to the lower OR plane by the vertical lines. The sum terms for each output is given by the corresponding horizontal line

E.g., $IorD = S0.S1.\overline{S2}.\overline{S3} + S0.\overline{S1}.S2.\overline{S3}$

FSM Control: 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
  - outputs are the bits of the entry the address points to



m = 3
n = 4

address			output			
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

The size of an m-input n-output ROM is 2^m x n bits – such a ROM can be thought of as an array of size 2^m with each entry in the array being n bits

FSM Control: ROM vs. PLA

- First improve the ROM: break the table into two parts
  - 4 state bits give the 16 output signals – 2^4 x 16 bits of ROM
  - all 10 input bits give 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
- PLA size = (#inputs × #product-terms) + (#outputs × #product-terms)
  - FSM control PLA = (10x17)+(20x17) = 460 PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

Microprogramming

- Microprogramming is a method of *specifying* FSM control that resembles a programming language – textual rather graphic
  - this is appropriate when the FSM becomes very large, e.g., if the instruction set is large and/or the number of cycles per instruction is large
  - in such situations graphical representation becomes difficult as there may be thousands of states and even more arcs joining them
  - a microprogram is *specification* : *implementation* is by ROM or PLA
- A *microprogram is a sequence of microinstructions*
  - each microinstruction has eight fields (label + 7 functional)
    - Label: used to control microcode sequencing
    - ALU control: specify operation to be done by ALU
    - SRC1: specify source for first ALU operand
    - SRC2: specify source for second ALU operand
    - Register control: specify read/write for register file
    - Memory: specify read/write for memory
    - PCWrite control: specify the writing of the PC
    - Sequencing: specify choice of next microinstruction

Microprogramming

- The *Sequencing* field value determines the execution order of the microprogram
  - value *Seq* : control passes to the sequentially next microinstruction
  - value *Fetch* : branch to the first microinstruction to begin the next MIPS instruction, i.e., the first microinstruction in the microprogram
  - value *Dispatch i* : branch to a microinstruction based on control input and a dispatch table entry (called *dispatching*):
    - Dispatching is implemented by means of creating a table, called *dispatch table*, whose entries are microinstruction labels and which is indexed by the control input. There may be multiple dispatch tables – the value *Dispatch i* in the sequencing field indicates that the *i* th dispatch table is to be used

Control Microprogram

- The microprogram corresponding to the FSM control shown graphically earlier:

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

Microprogram containing 10 microinstructions

Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	Rformat1
000010	`jmp`	JUMP1
000100	`beq`	BEQ1
100011	`lw`	Mem1
101011	`sw`	Mem1

Dispatch Table 1

Dispatch ROM 2		
Op	Opcode name	Value
100011	`lw`	LW2
101011	`sw`	SW2

Dispatch Table 2

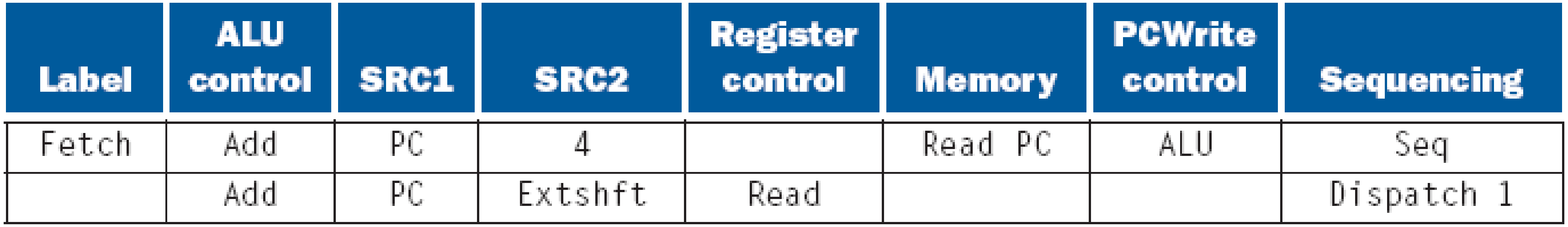
Field name	Values for field	Function of field with specific value
Label	Any string	Used to specify labels to control microcode sequencing. Labels that end in a 1 or 2 are used for dispatching with a jump table that is indexed based on the opcode. Other labels are used as direct targets in the microinstruction sequencing. Labels do not generate control signals directly but are used to define the contents of dispatch tables and generate control for the Sequencing field.
ALU control	`Add`	Cause the ALU to add.
	`Subt`	Cause the ALU to subtract; this implements the compare for branches.
	`Func code`	Use the instruction's funct field to determine ALU control.
SRC1	`PC`	Use the PC as the first ALU input.
	`A`	Register A is the first ALU input.
SRC2	`B`	Register B is the second ALU input.
	`4`	Use 4 for the second ALU input.
	`Extend`	Use output of the sign extension unit as the second ALU input.
	`Extshft`	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, putting the data into registers A and B.
	`Write ALU`	Write the register file using the rd field of the IR as the register number and the contents of ALUOut as the data.
	`Write MDR`	Write the register file using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	`Read PC`	Read memory using the PC as address; write result into IR (and the MDR).
	`Read ALU`	Read memory using ALUOut as address; write result into MDR.
	`Write ALU`	Write memory using the ALUOut as address; contents of B as the data.
PCWrite control	`ALU`	Write the output of the ALU into the PC.
	`ALUOut-cond`	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	`Jump address`	Write the PC with the jump address from the instruction.
Sequencing	`Seq`	Choose the next microinstruction sequentially.
	`Fetch`	Go to the first microinstruction to begin a new instruction.
	`Dispatch i`	Dispatch using the ROM specified by i (1 or 2).

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 ALUOut as the data.

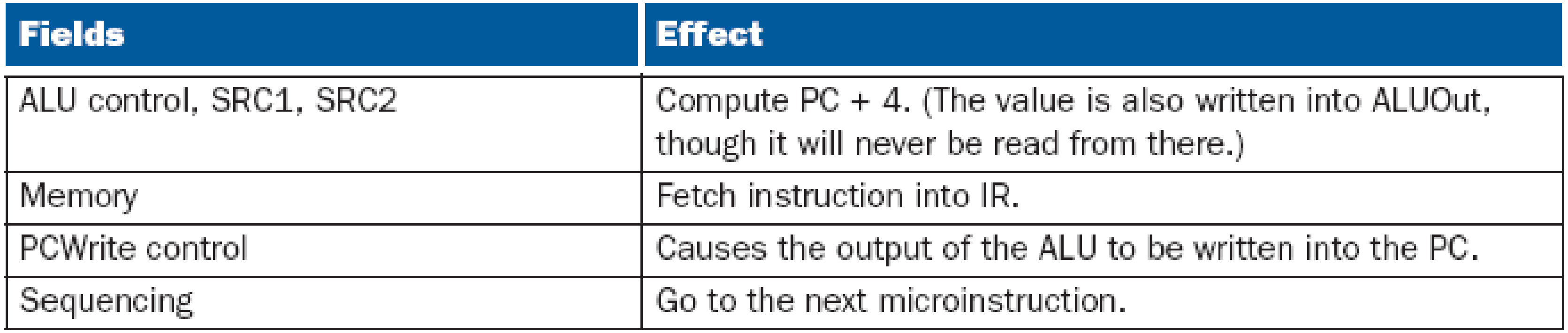
	Write MDR	RegWrite,
		RegDst = 0,
		MemtoReg = 1
			Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.

Memory	Read PC	MemRead, lorD = 0, IRWrite	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using 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.

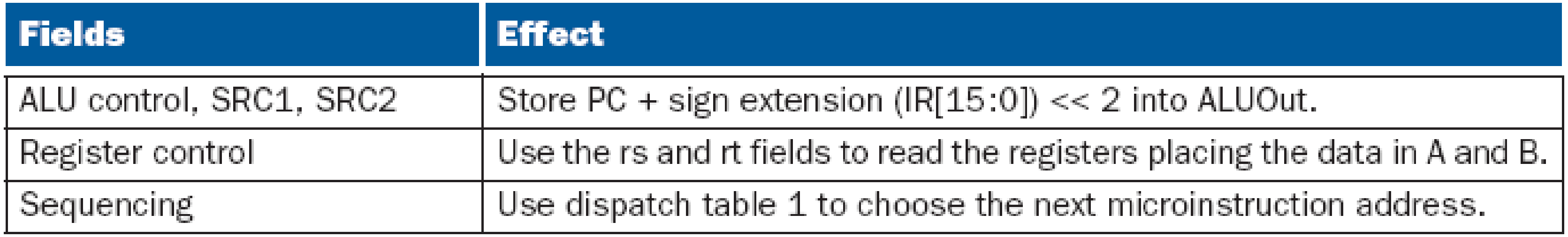
Fetch & Decode



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

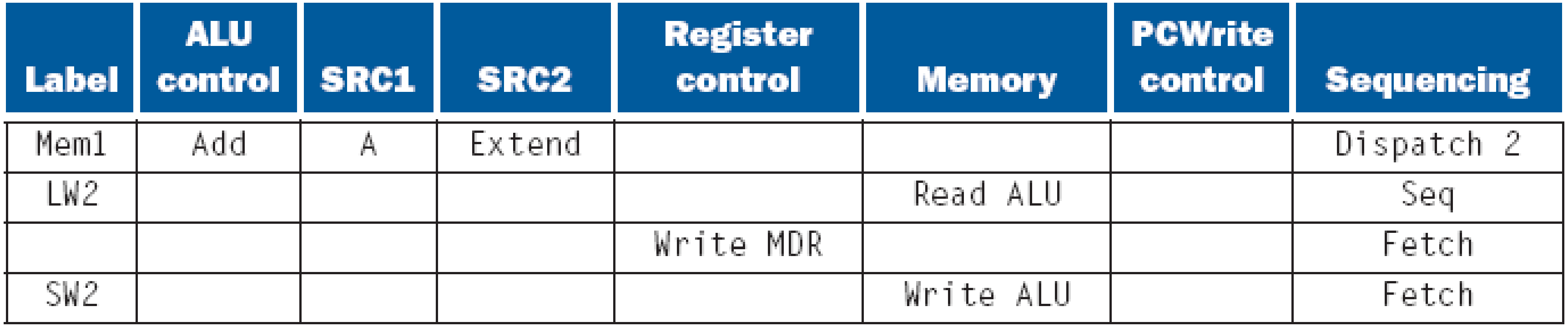


Fields	Effect
ALU control, SRC1, SRC2	Compute PC + 4. (The value is also written into ALUOut, though it will never be read from there.)
Memory	Fetch instruction into IR.
PCWrite control	Causes the output of the ALU to be written into the PC.
Sequencing	Go to the next microinstruction.

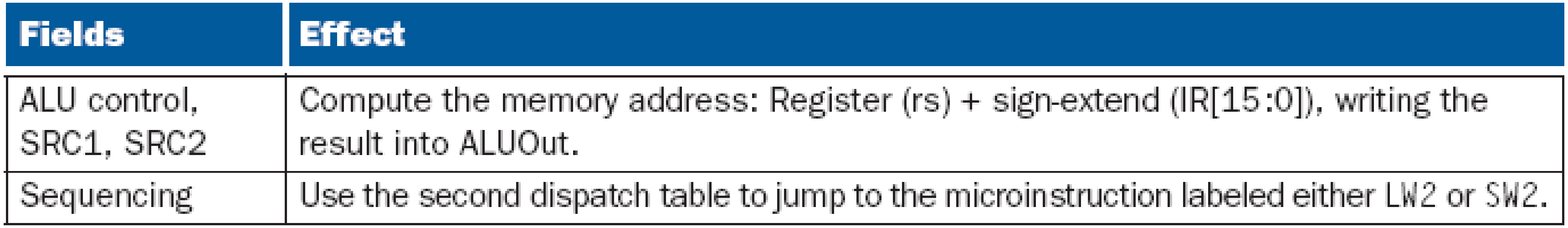


Fields	Effect
ALU control, SRC1, SRC2	Store PC + sign extension (IR[15:0]) << 2 into ALUOut.
Register control	Use the rs and rt fields to read the registers placing the data in A and B.
Sequencing	Use dispatch table 1 to choose the next microinstruction address.

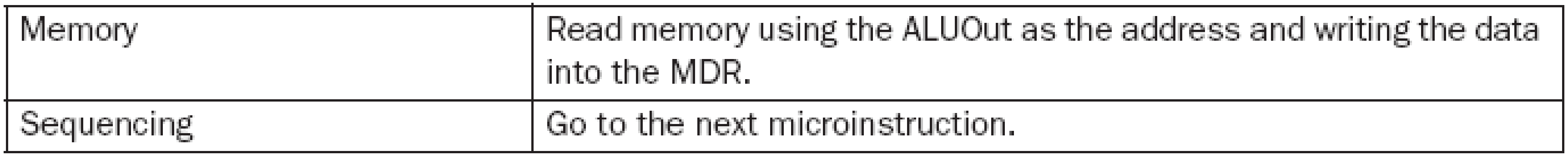
Memory Reference



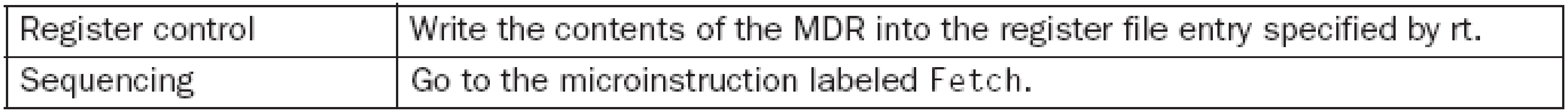
Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch



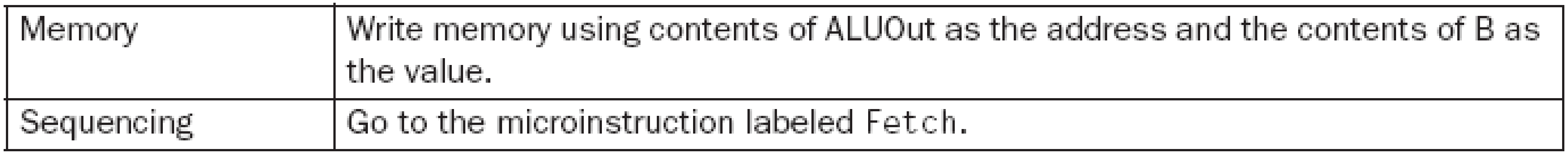
Fields	Effect
ALU control, SRC1, SRC2	Compute the memory address: Register (rs) + sign-extend (IR[15:0]), writing the result into ALUOut.
Sequencing	Use the second dispatch table to jump to the microinstruction labeled either LW2 or SW2.



Memory	Read memory using the ALUOut as the address and writing the data into the MDR.
Sequencing	Go to the next microinstruction.

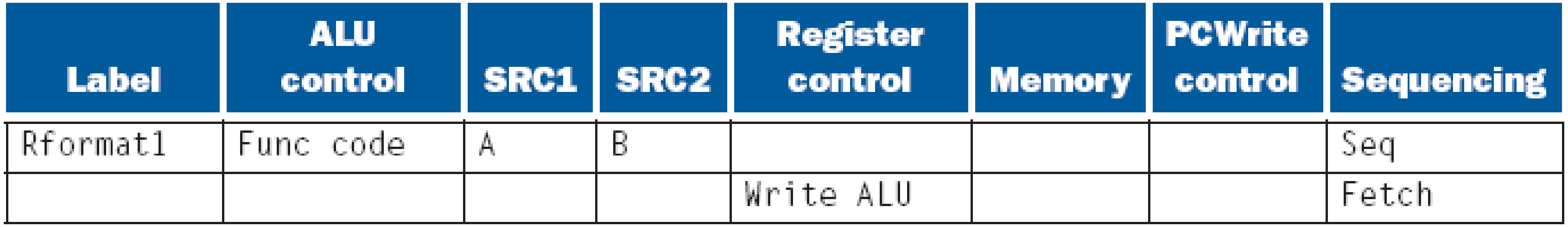


Register control	Write the contents of the MDR into the register file entry specified by rt.
Sequencing	Go to the microinstruction labeled Fetch.

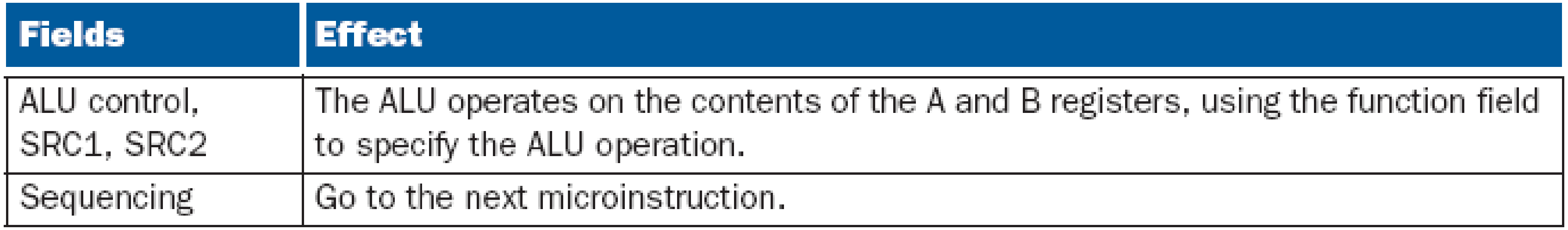


Memory	Write memory using contents of ALUOut as the address and the contents of B as the value.
Sequencing	Go to the microinstruction labeled Fetch.

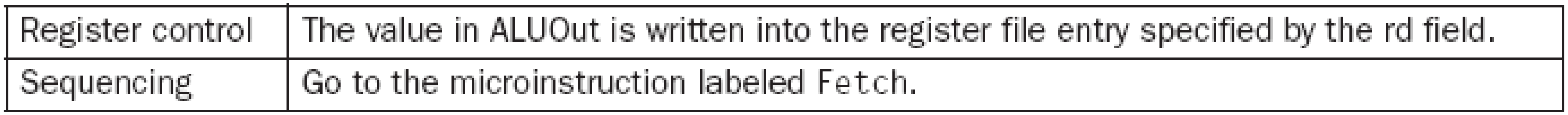
R-Type



Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch

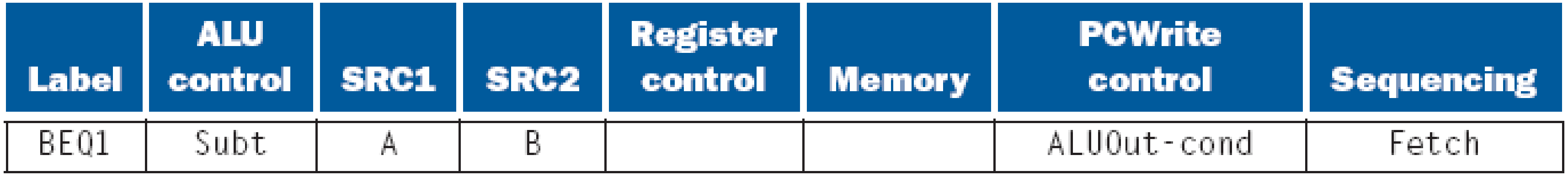


Fields	Effect
ALU control, SRC1, SRC2	The ALU operates on the contents of the A and B registers, using the function field to specify the ALU operation.
Sequencing	Go to the next microinstruction.

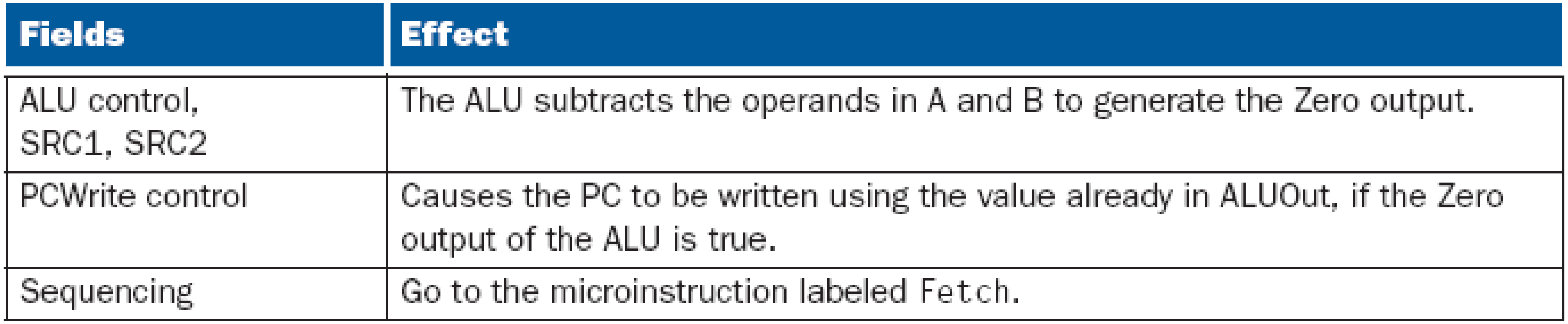


Register control	The value in ALUOut is written into the register file entry specified by the rd field.
Sequencing	Go to the microinstruction labeled Fetch.

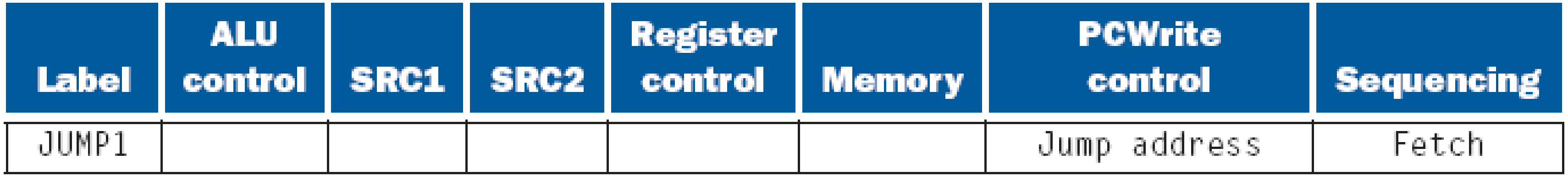
Beq & Jump



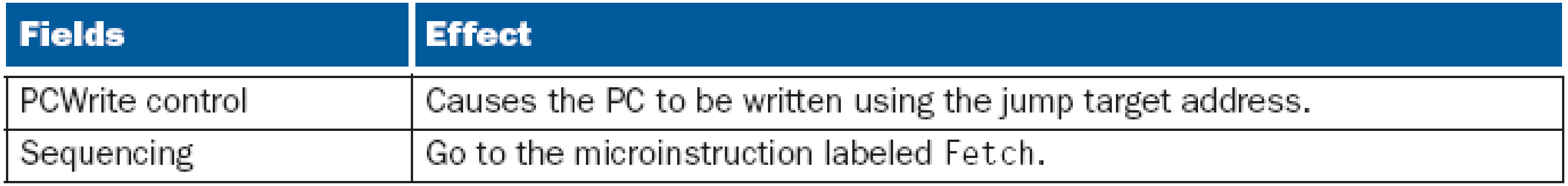
Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
BEQ1	Subt	A	B			ALUOut-cond	Fetch



Fields	Effect
ALU control, SRC1, SRC2	The ALU subtracts the operands in A and B to generate the Zero output.
PCWrite control	Causes the PC to be written using the value already in ALUOut, if the Zero output of the ALU is true.
Sequencing	Go to the microinstruction labeled Fetch.

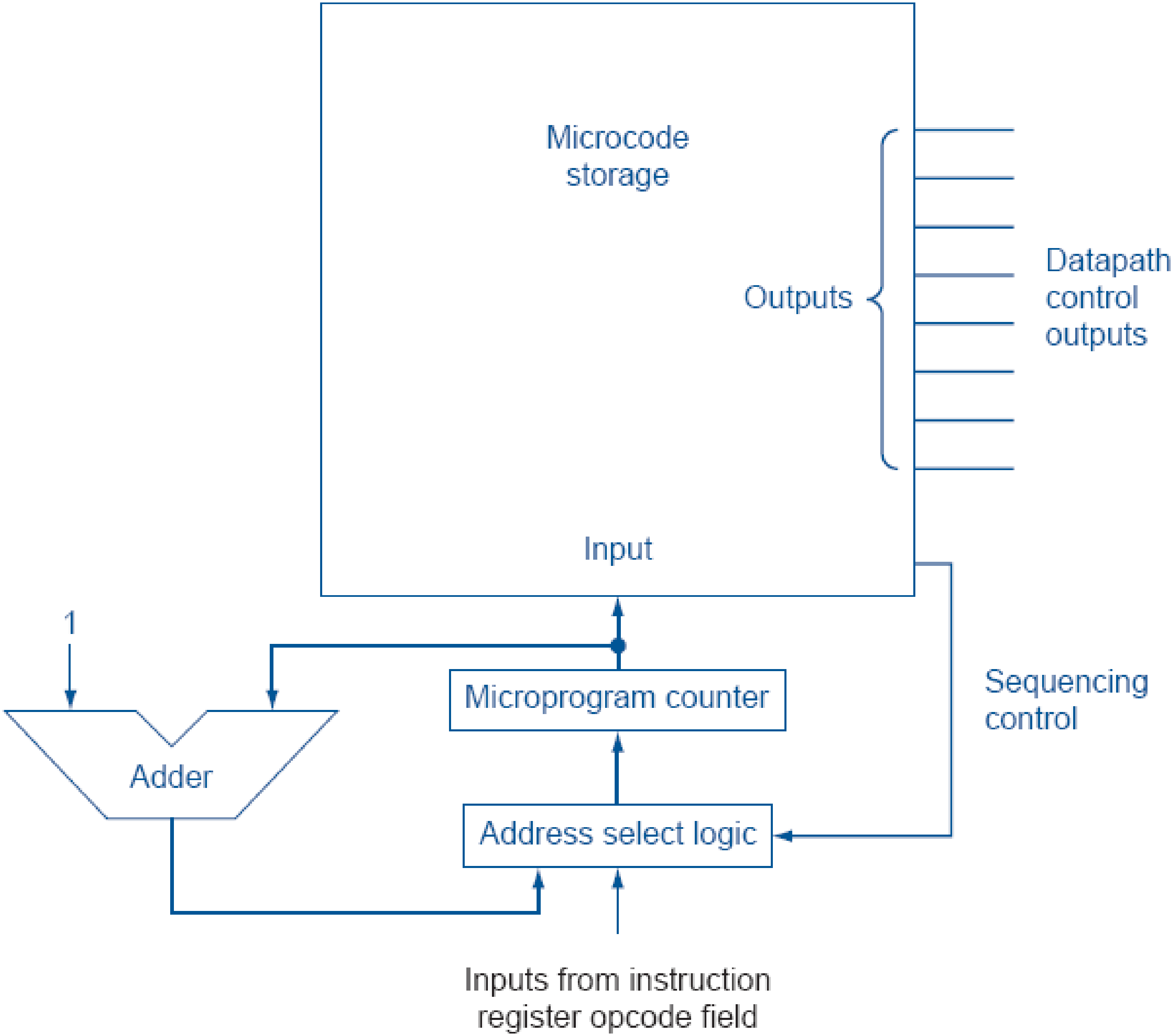


Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
JUMP1						Jump address	Fetch

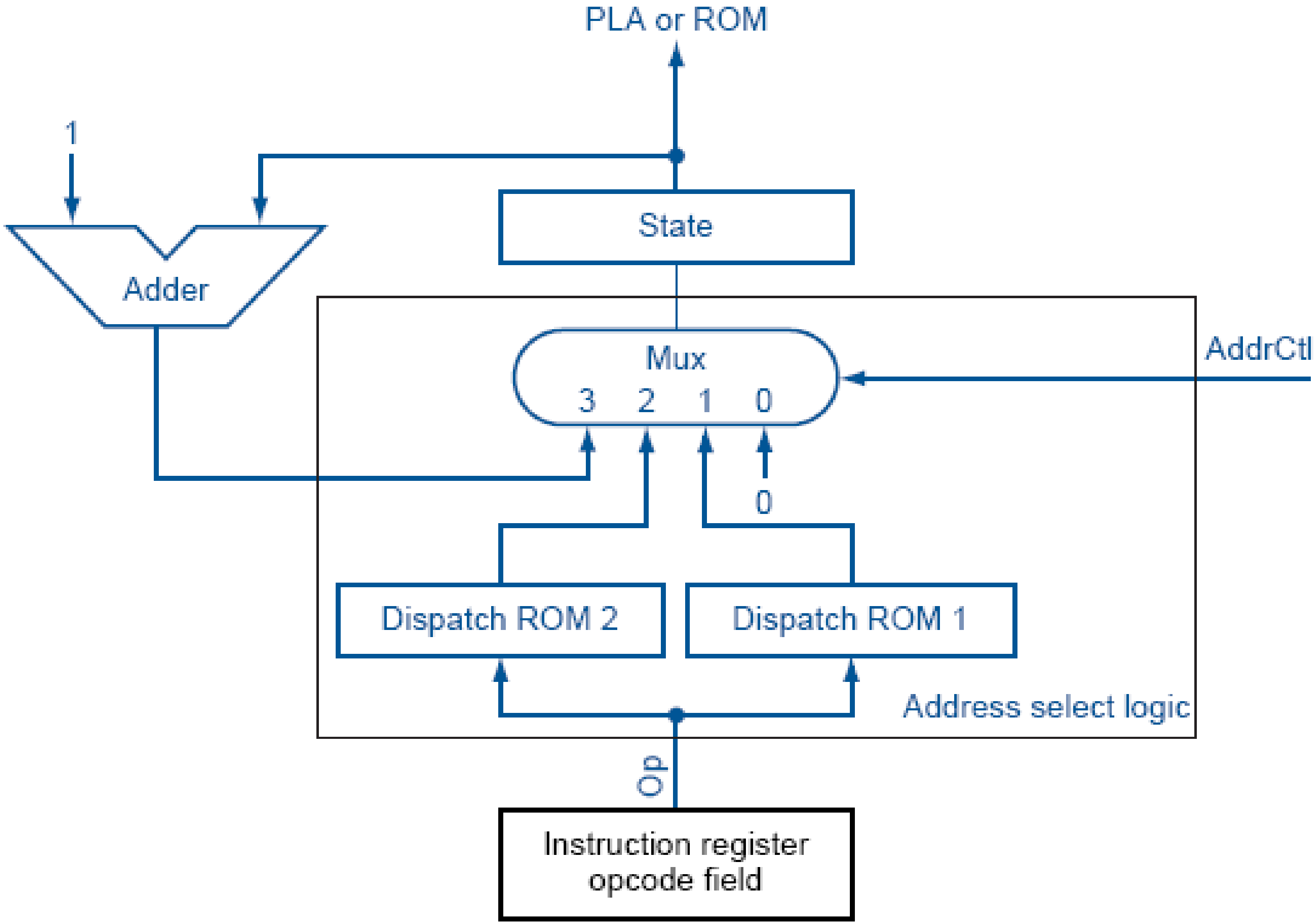


Fields	Effect
PCWrite control	Causes the PC to be written using the jump target address.
Sequencing	Go to the microinstruction labeled Fetch.

Implementing the Microprogram



Implementing the Address Select Logic



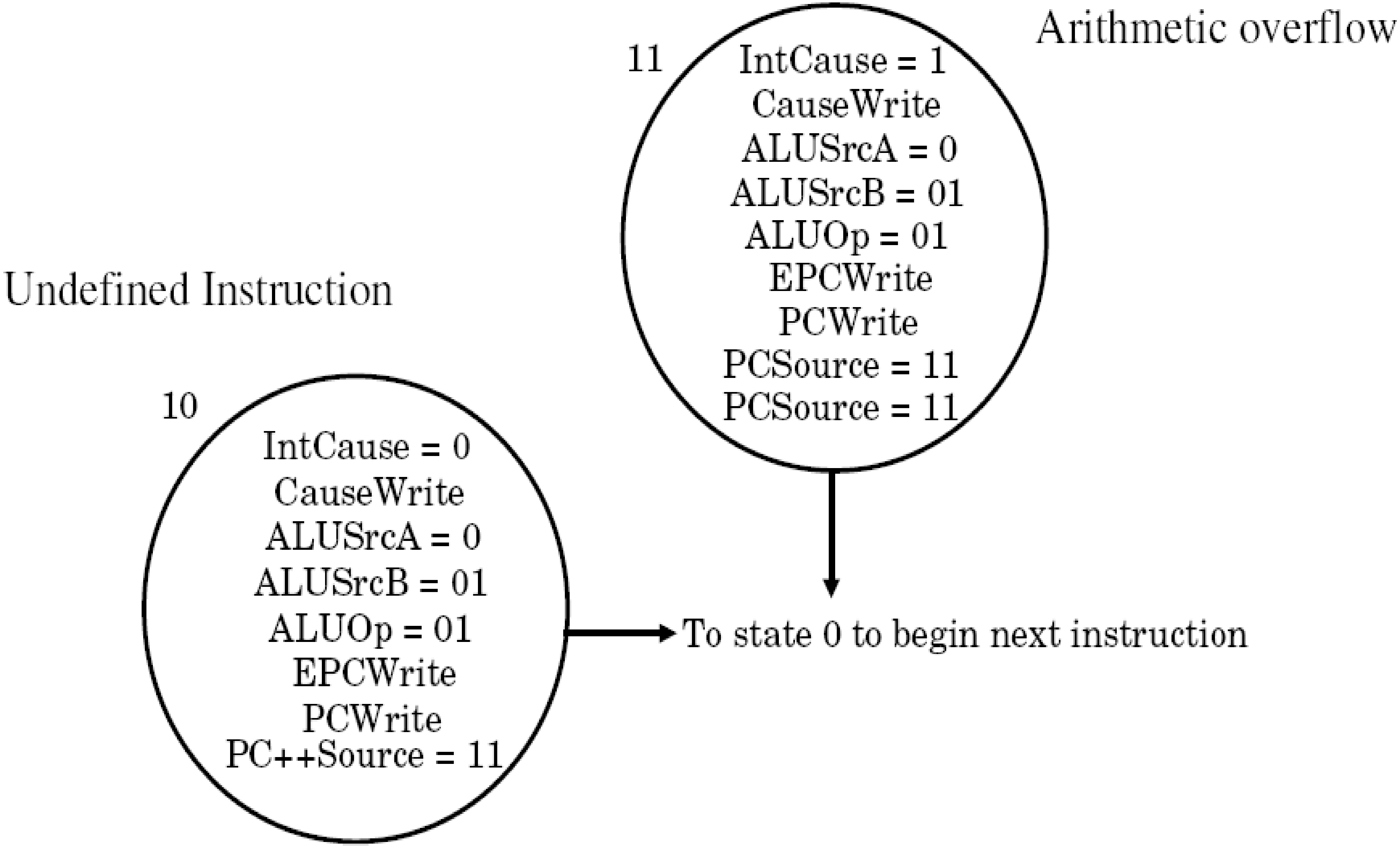
Microcode: Trade-offs

Specification advantages

  - easy to design and write
  - typically manufacturer designs architecture and microcode in parallel

- Implementation advantages
  - easy to change since values are in memory (e.g., off-chip ROM)
- Implementation disadvantages
  - control is implemented nowadays on same chip as processor so the advantage of an off-chip ROM does not exist
  - ROM is not fast
  - there is little need to change the microcode as general-purpose computers are used far more nowadays than computers designed for specific applications

Exception Handling



Arithmetic overflow

11

IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
PCSource = 11

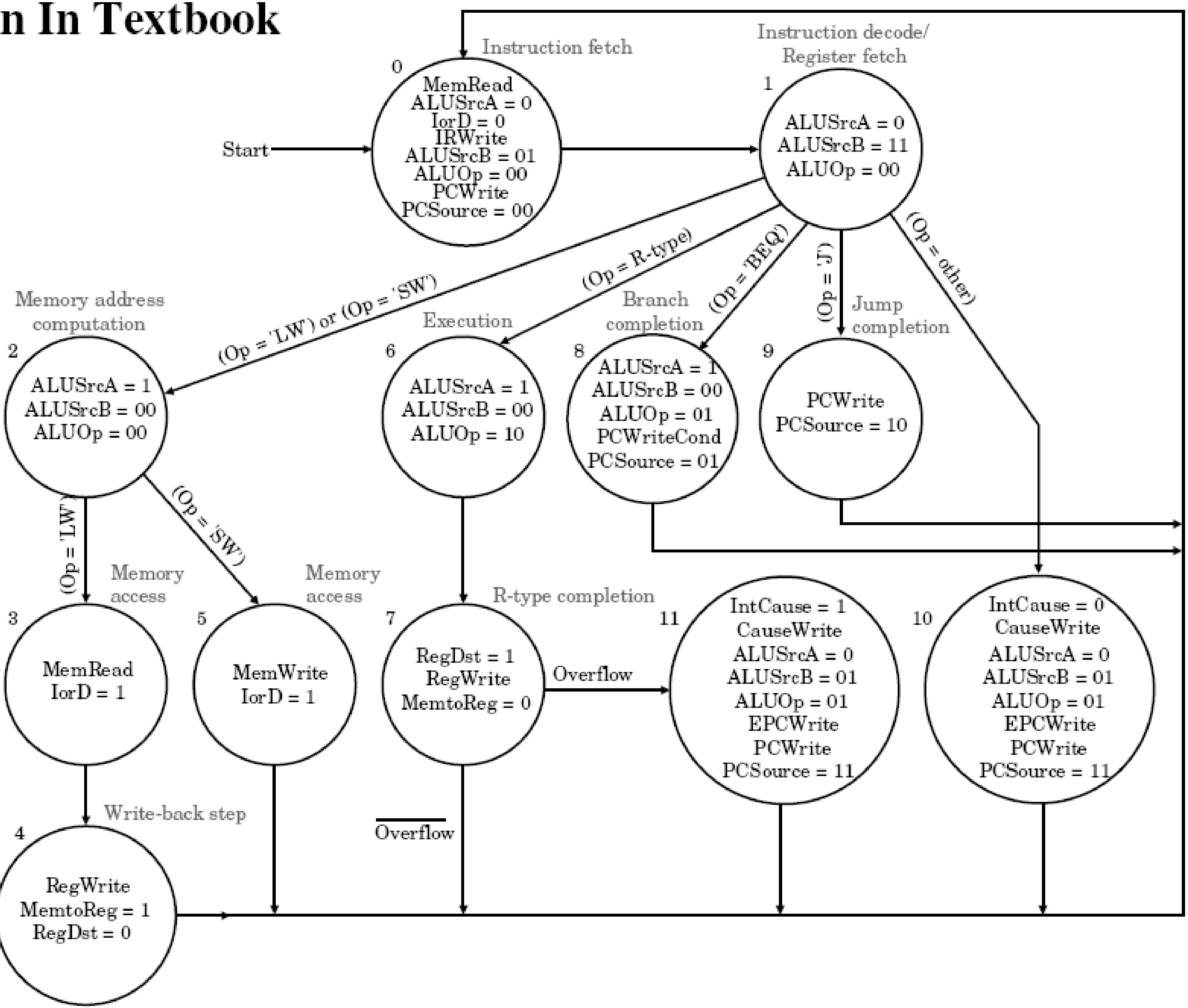
Undefined Instruction

10

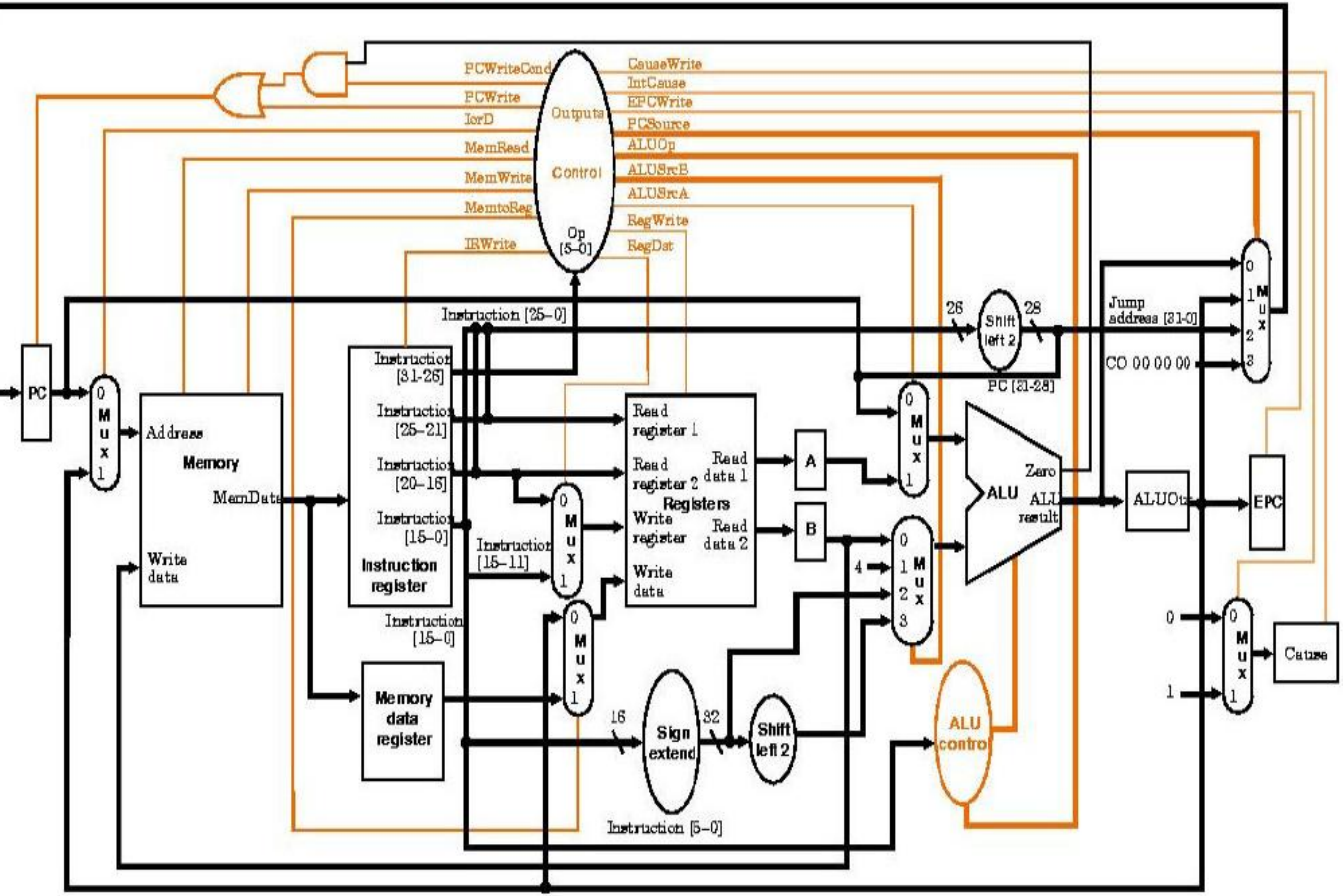
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PC++Source = 11

To state 0 to begin next instruction

Version In Textbook



Required Hardware



Summary

- Multicycle datapaths offer two great advantages over single-cycle
  - functional units can be reused within a single instruction if they are accessed in different cycles – reducing the need to replicate expensive logic
  - instructions with shorter execution paths can complete quicker by consuming fewer cycles
- Modern computers, in fact, take the multicycle paradigm to a higher level to achieve greater instruction throughput:
  - *pipelining* (next topic) where multiple instructions execute simultaneously by having cycles of different instructions overlap in the datapath
  - *the MIPS architecture was designed to be pipelined*