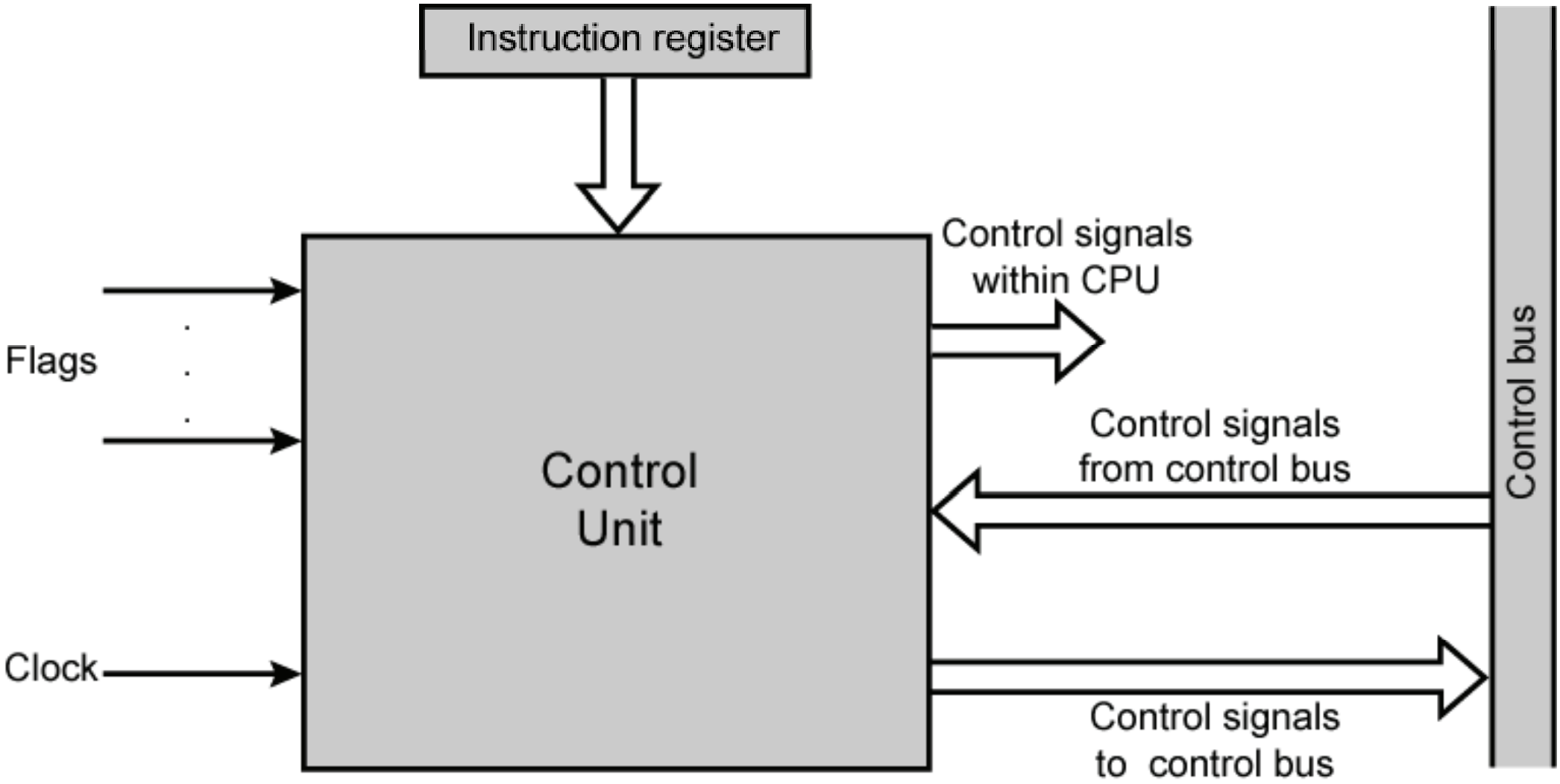
Functions of Control Unit

- Sequencing
  - Causing the CPU to step through a series of micro-operations
- Execution
  - Causing the performance of each micro-op
- Use of Control Signals to accomplish the task

Types of Control Signals

- Clock
  - One micro-instruction (or set of parallel micro-instructions) per clock cycle
- Instruction register
  - Op-code for current instruction
  - Determines which micro-instructions are performed
- Flags
  - State of CPU
  - Results of previous operations
- From control bus
  - Interrupts
  - Acknowledgements

Model of Control Unit

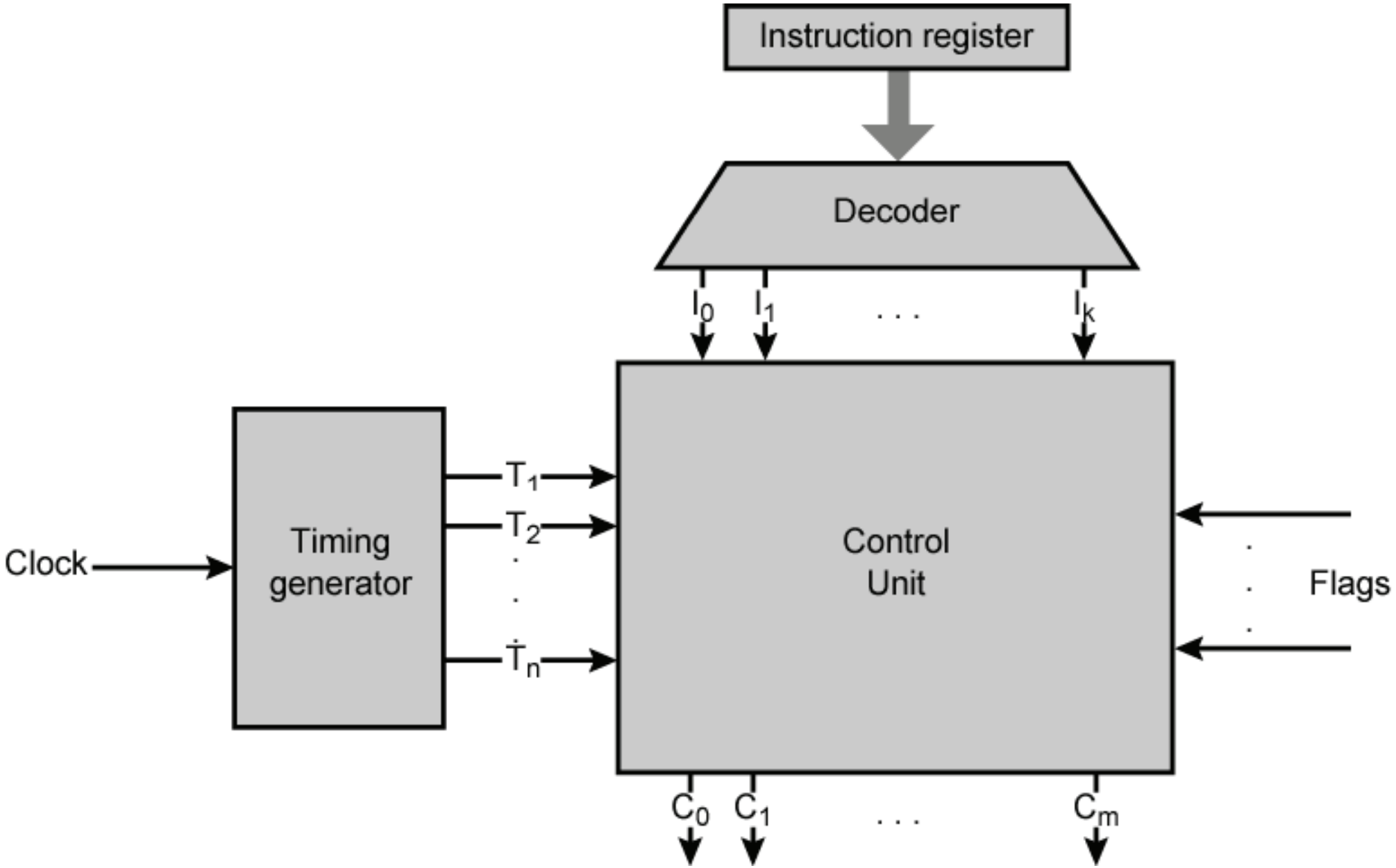


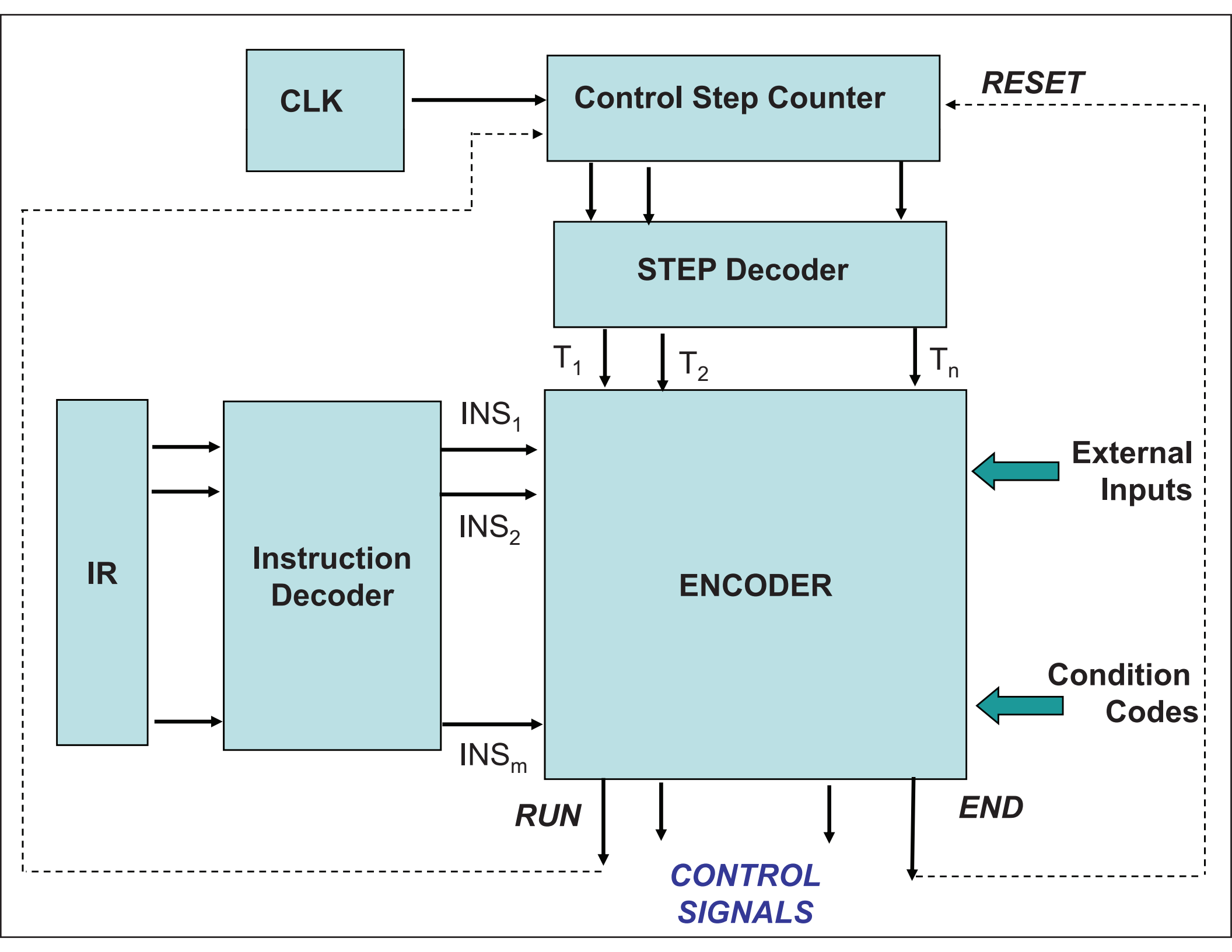
HARDWIRED CONTROL

The required control signals are determined by the following information:

- **Contents of the control Step Counter**
- **Contents of the IR**
- **Contents of the condition code flags**
- **External I/P signals, MFC, IRQ etc.**

Control Unit with Decoded Inputs




CLK
Control Step Counter
RESET
STEP Decoder
T1
T2
Tn
IR
Instruction Decoder
INS1
INS2
INSm
ENCODER
External Inputs
Condition Codes
RUN
END
CONTROL SIGNALS

By separating the decoding and encoding functions, we obtain the more detailed block diagram in Figure 7.11. The step decoder provides a separate signal line for each step, or time slot, in the control sequence. Similarly, the output of the instruction decoder consists of a separate line for each machine instruction. For any instruction loaded in the IR, one of the output lines INS_1 through INS_m is set to 1, and all other lines are set to 0. (For design details of decoders, refer to Appendix A.) The input signals to the encoder block in Figure 7.11 are combined to generate the individual control signals Y_{in}, PC_{out}, Add, End, and so on. An example of how the encoder generates the Z_{in}

For an "ADD" instruction (ISA):

1. PCout, MARin, READ, SEL #4, ADD, Zin
2.
3.
4.
5.
6. MDRout, SEL Y, ADD, Zin

For a "Branch" instruction (ISA):

1. PCout, ……, Zin
2.
3.
4. Offset (IRout), ADD, Zin
5. Zout, PCin, END

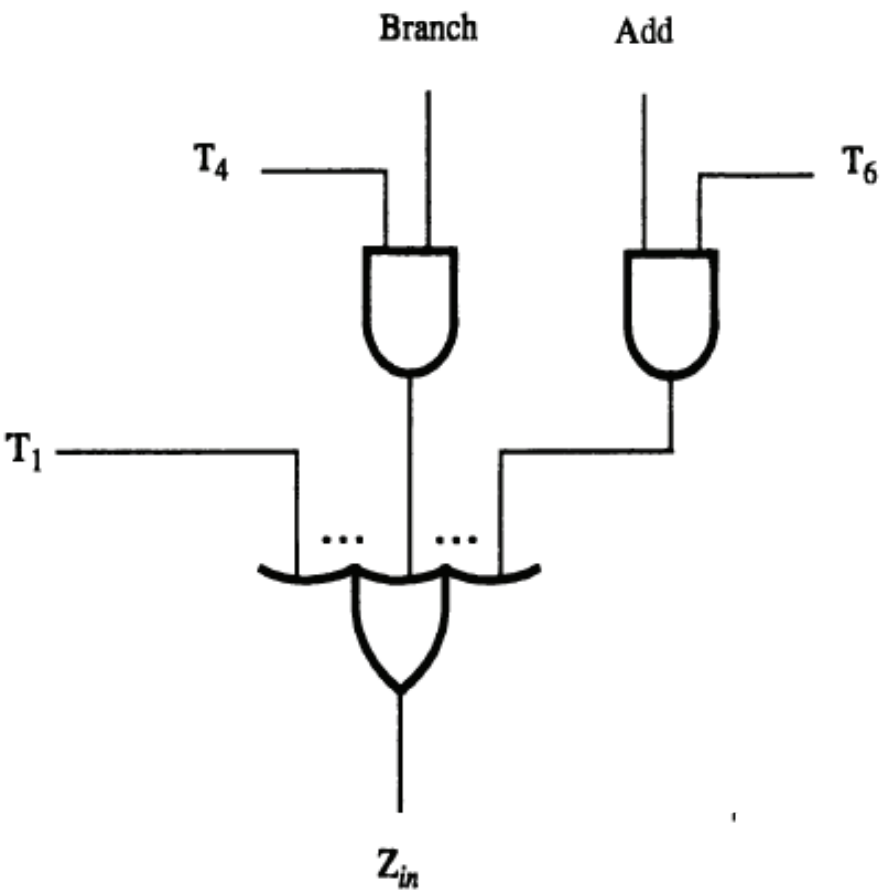
$$Z_{in} = T_1 + T_6.ADD + T_4.BR + ...$$

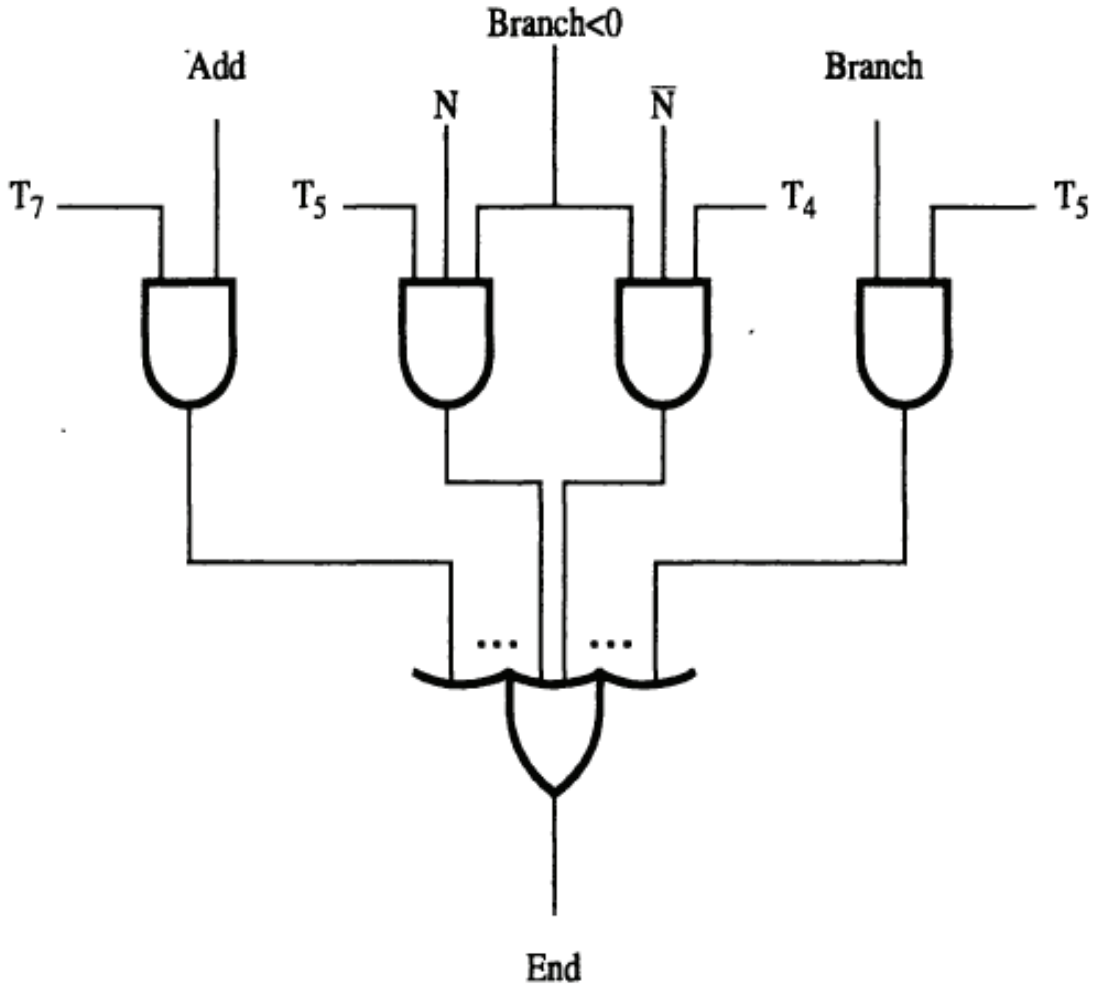
END = T_7.ADD + T_5.BR + (T_5.CF + T_4.CF').BRN +….

When RUN = 0, the counter STOPS; required from W_MFC;

Design logic mostly based on FSM (Finite State machine)

$$Z_{in} = T_1 + T_6.ADD + T_4.BR + ...$$





END = T_7.ADD + T_5.BR + (T_5.CF + T_4.CF').BRN +….

FSM – based Hardware Control Unit design

Moore type machine necessary - output signal depends on the current state.

Next state depends on the input and current state.

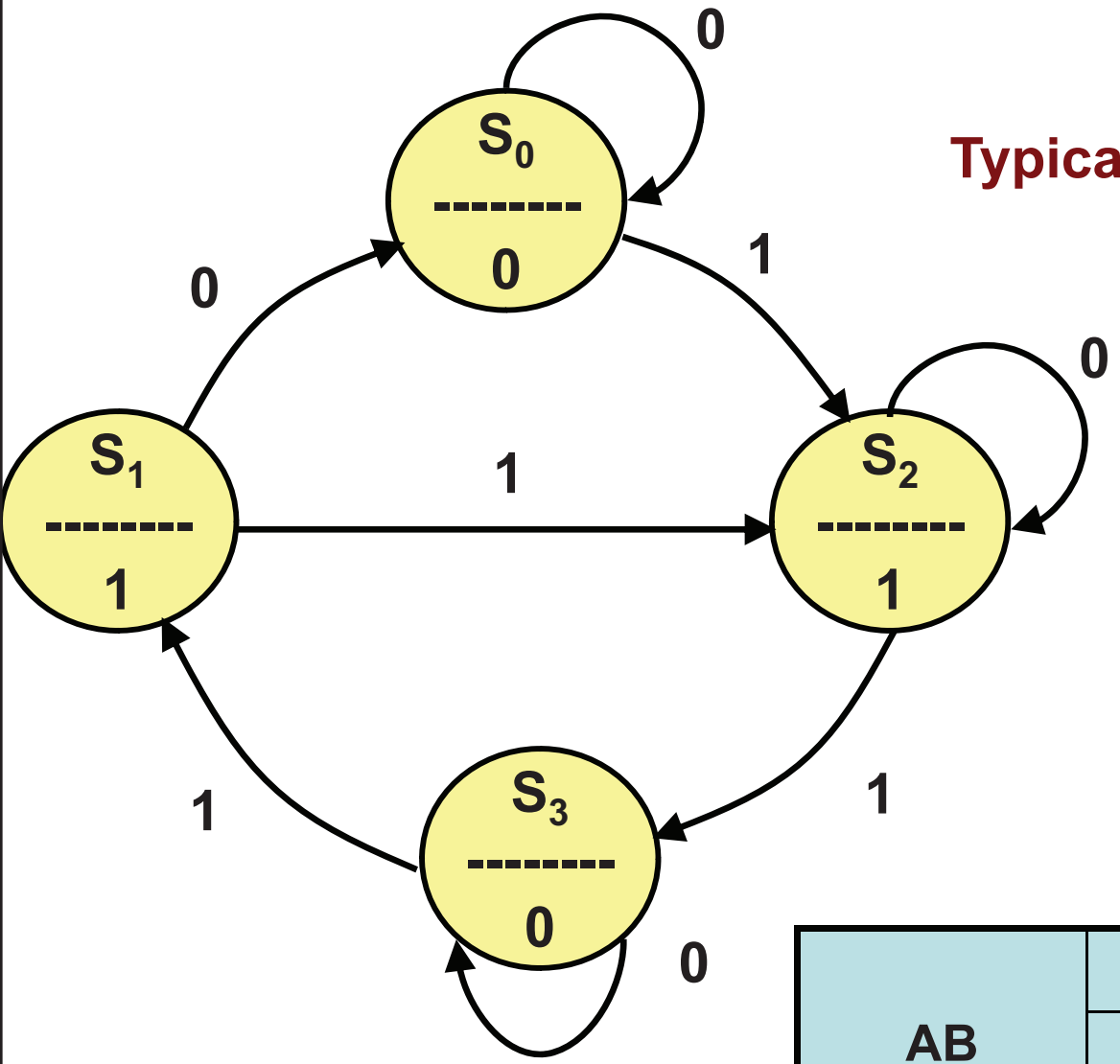
Each state generates a set of control signals.

To implement any ISA, the system sequentially changes state from one to another. Control Unit implements the steps.

For a sequence of "N" steps, there are S_0 to S_{N-1} stages.

At each stage S_i: a set of outputs $O_{i,0}....O_{i,M-1}$ are generated, depending on the S_i.

Categories of control signals: *functions for ALU, select of storage units, select of data routes (based on design).*

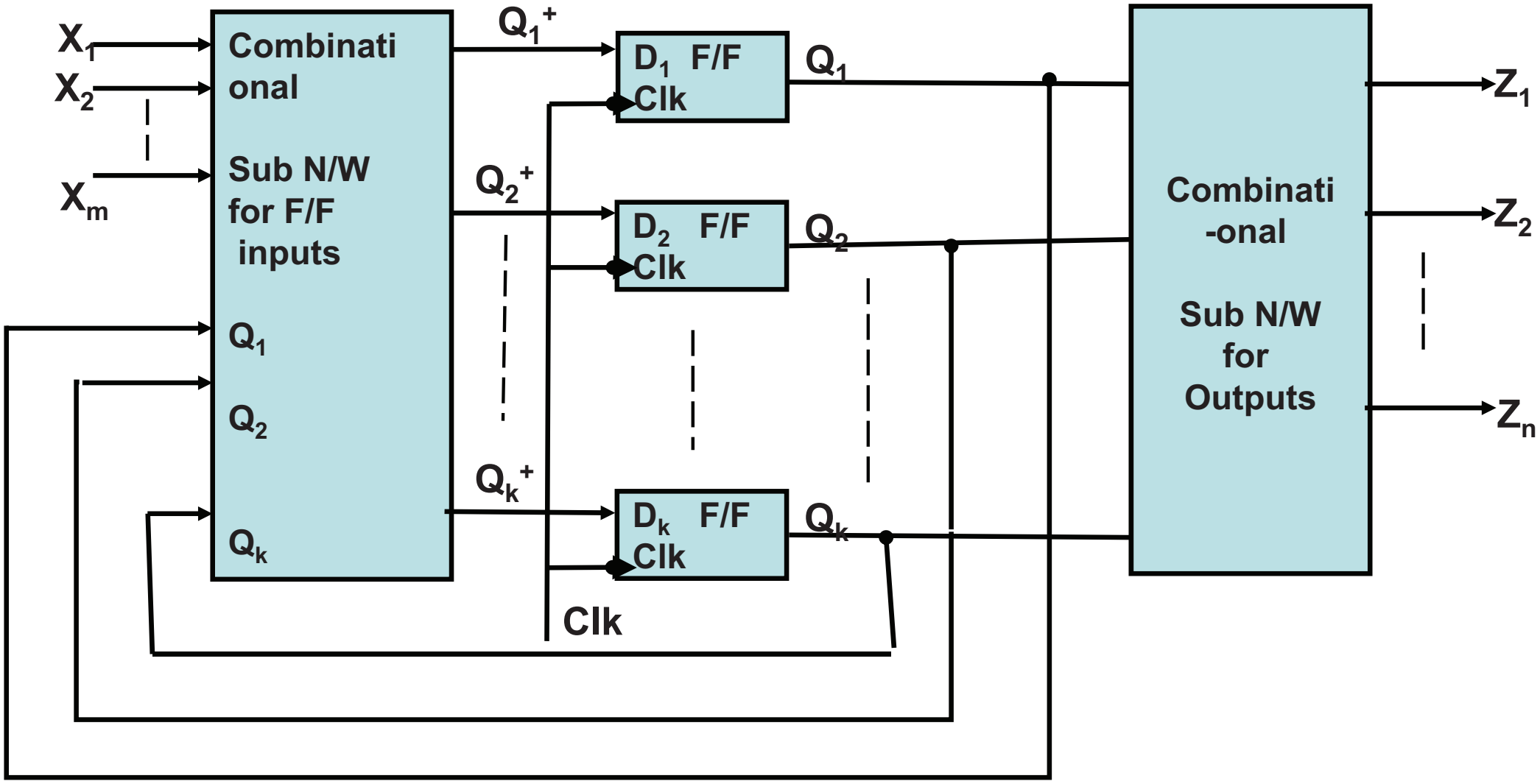


Typical Moore State Graph

Moore state table

AB	$A^+ B^+$		Z (Present output)
	X=0	X=1	
S_0 0 0	S_0 0 0	1 1 S_2	0
S_1 0 1	S_0 0 0	1 1 S_2	1
S_2 1 1	S_2 1 1	1 0 S_3	1
S_3 1 0	S_3 1 0	0 1 S_1	0

Moore network example



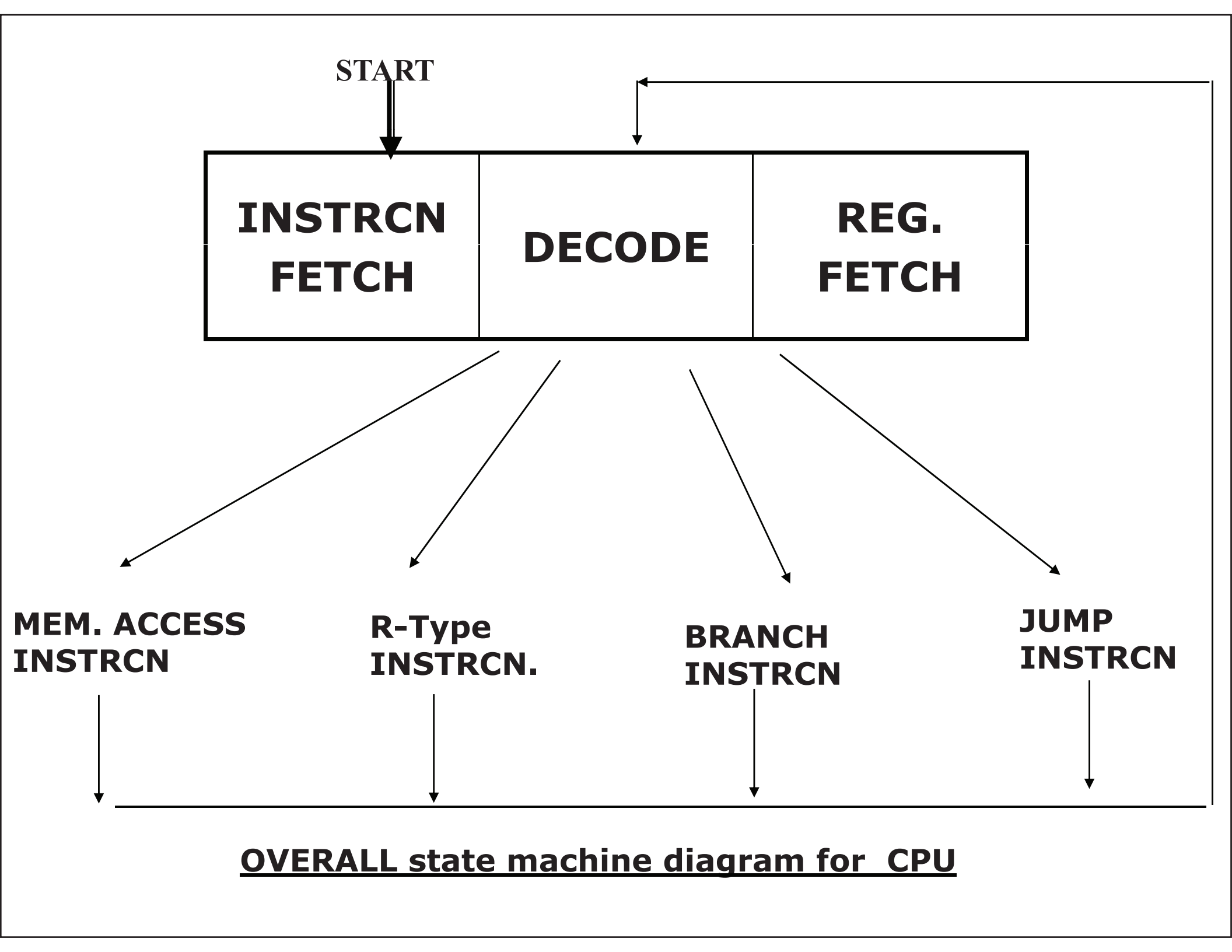
The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state.

The inputs to the combinational logic are the current state and any inputs used to determine the next state. In this case, the inputs are the instruction register opcode bits.

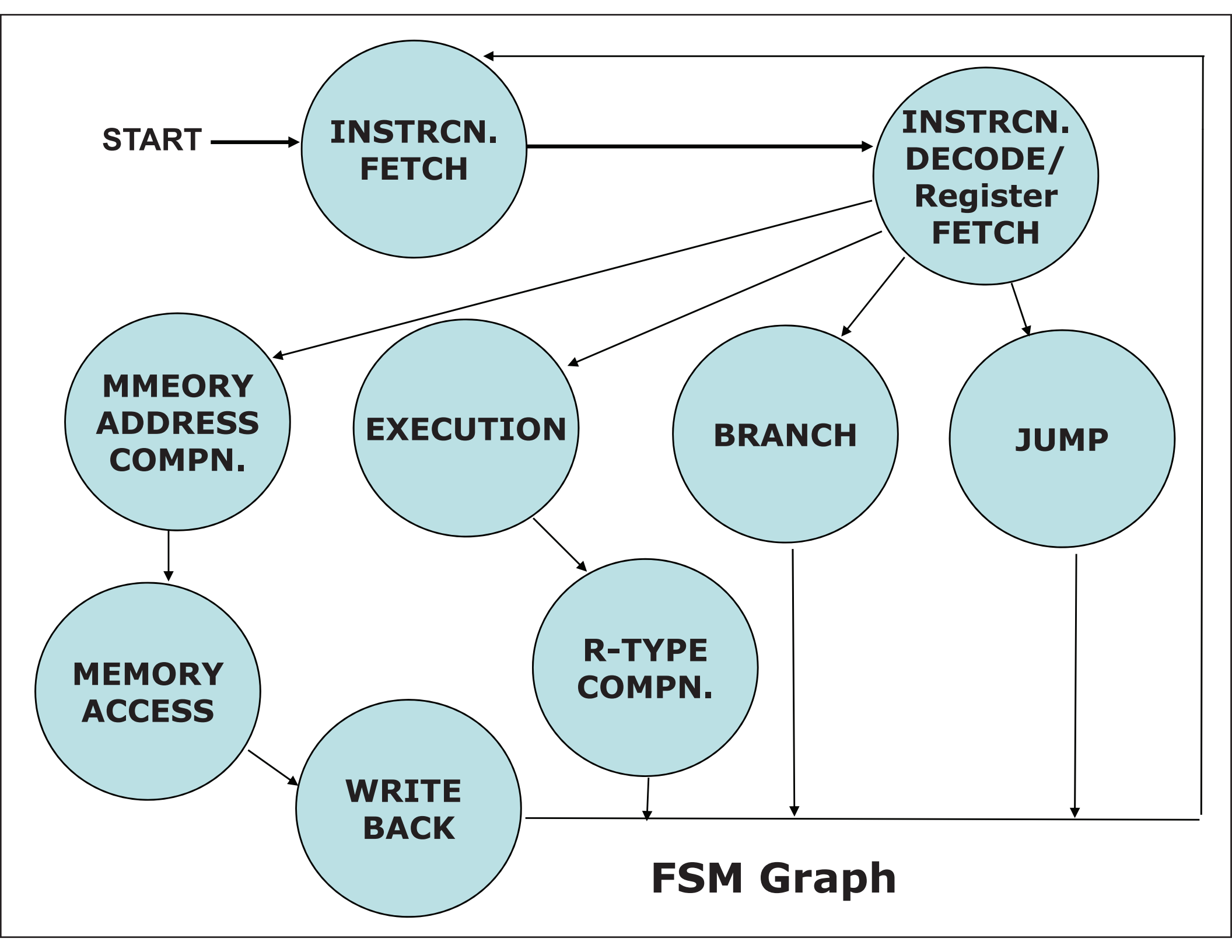
Notice that in the FSM for Hardwired Control, the outputs depend only on the current state, not on the inputs.

Identifying characteristic for a Moore machine is that the output depends only on the current state.

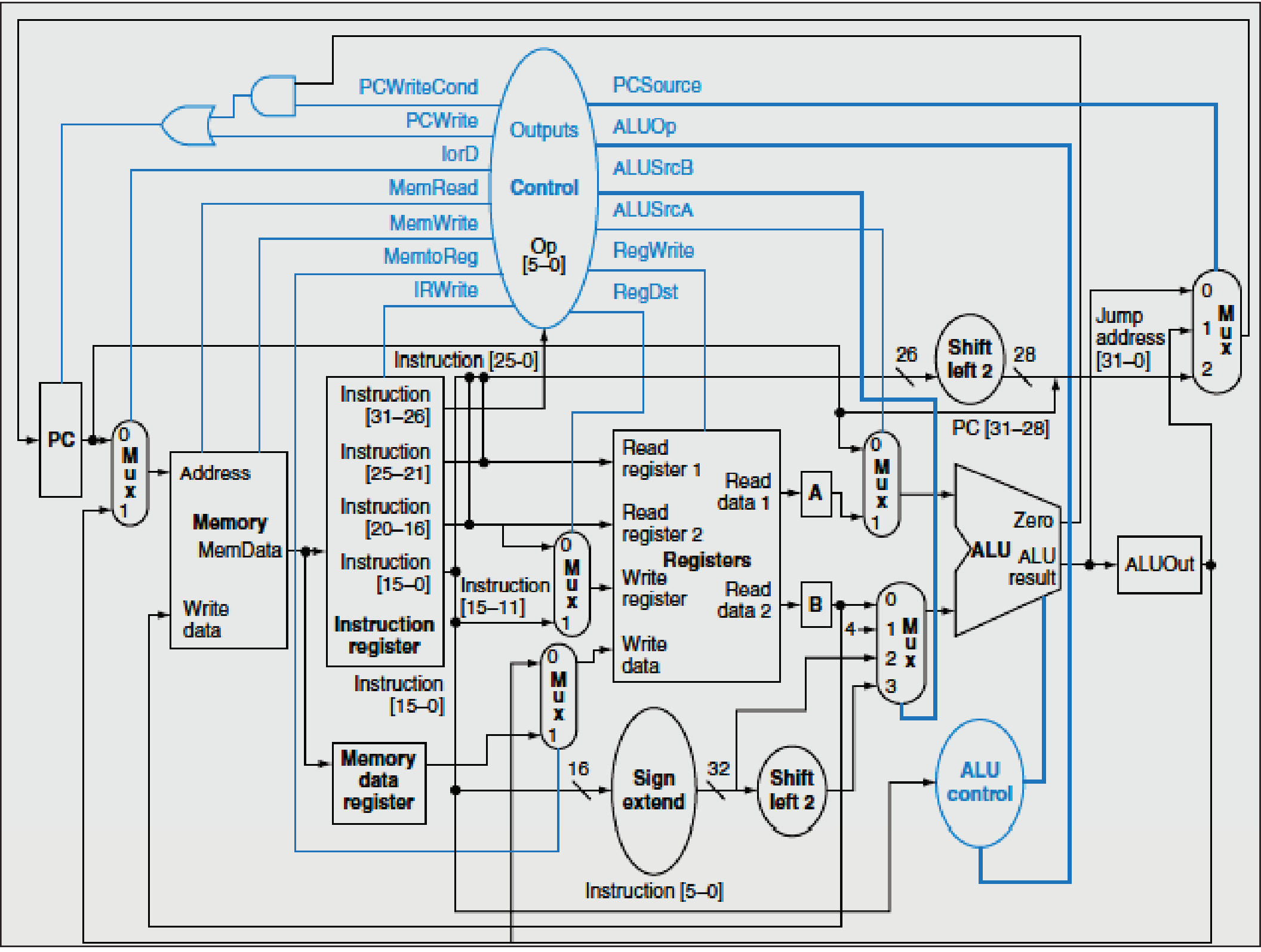
For a Moore machine, the box labeled combinational control logic can be split into two pieces. One piece has the control output and only the state input, while the other has only the next-state output.

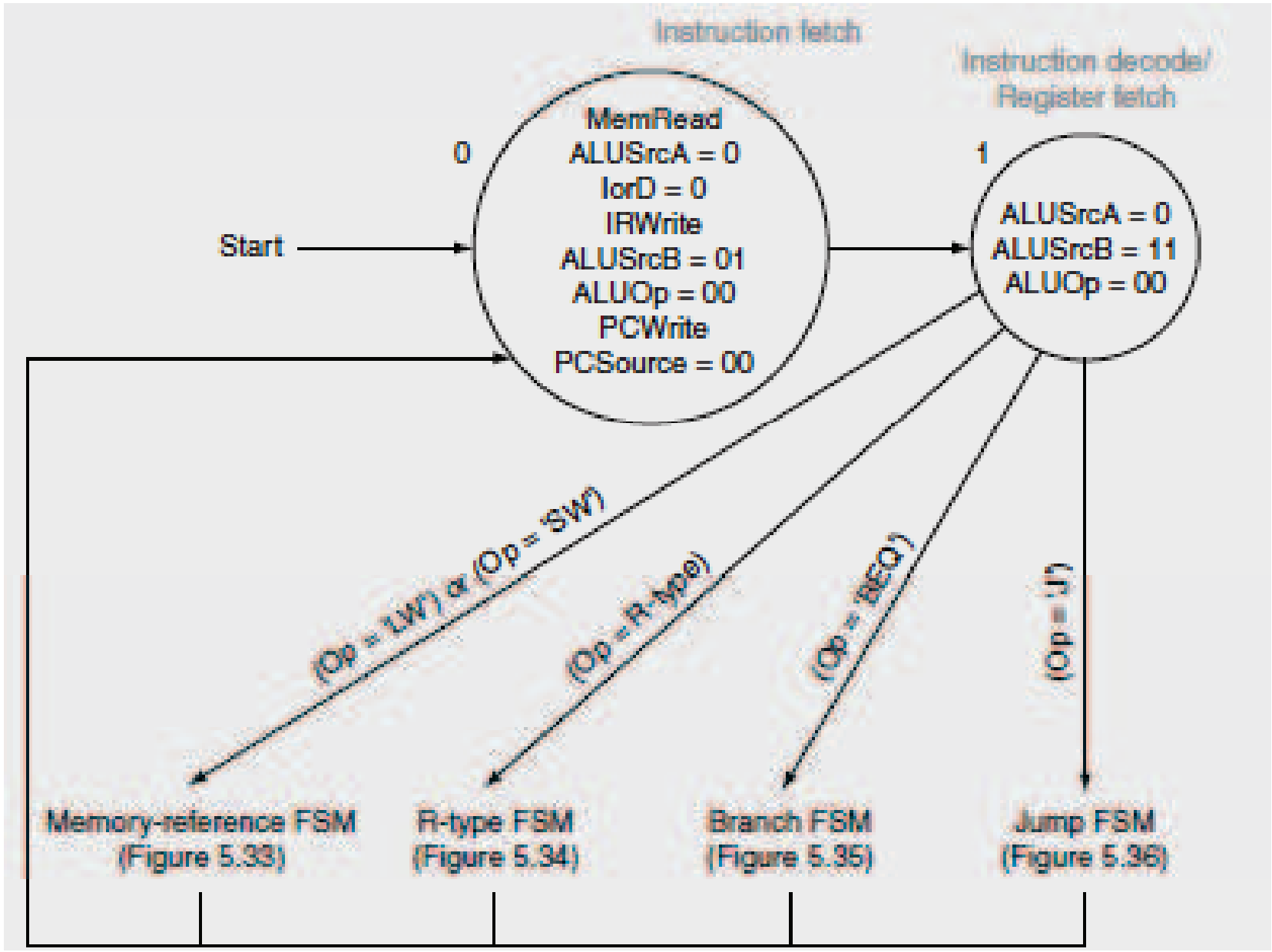


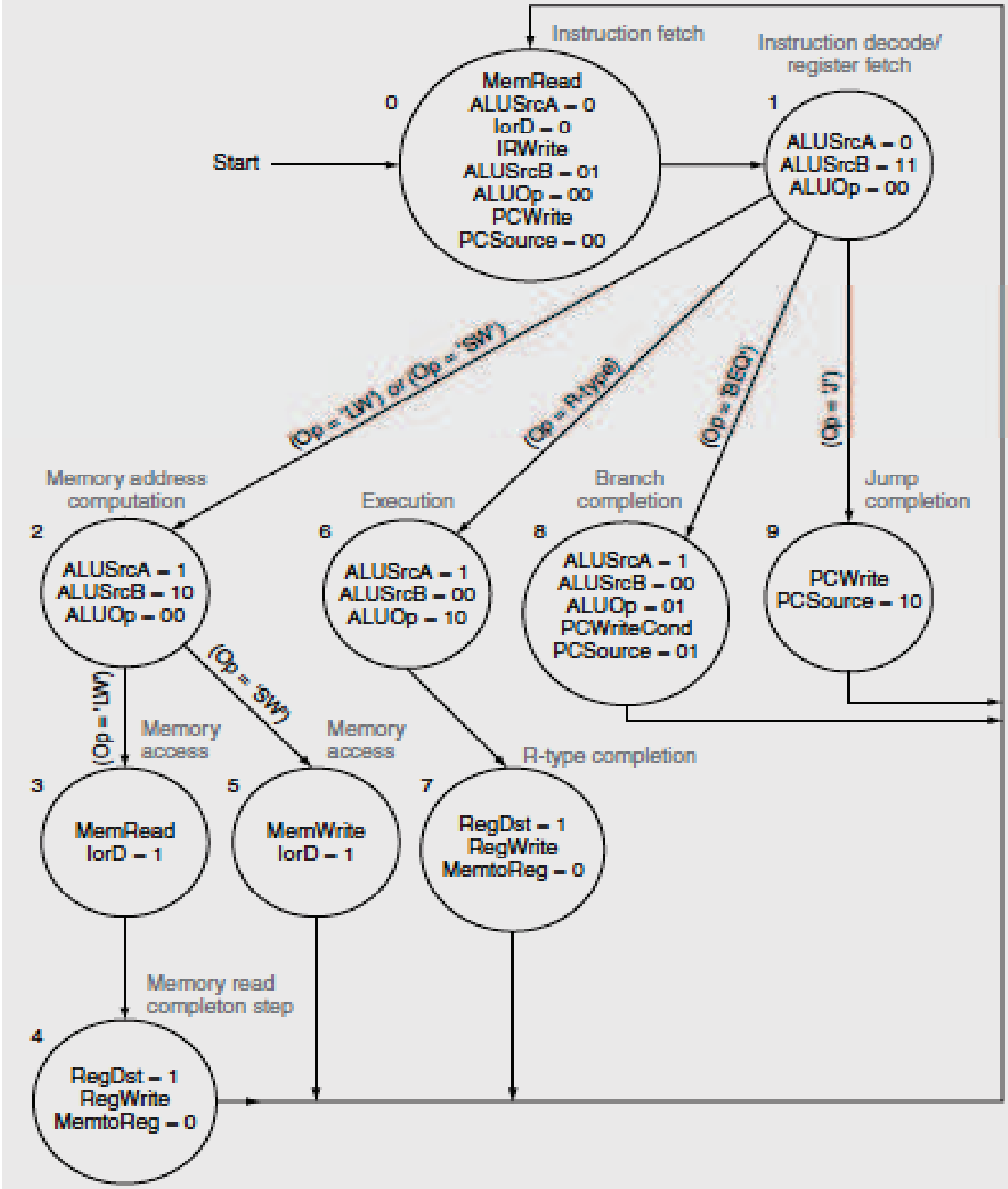
OVERALL state machine diagram for CPU

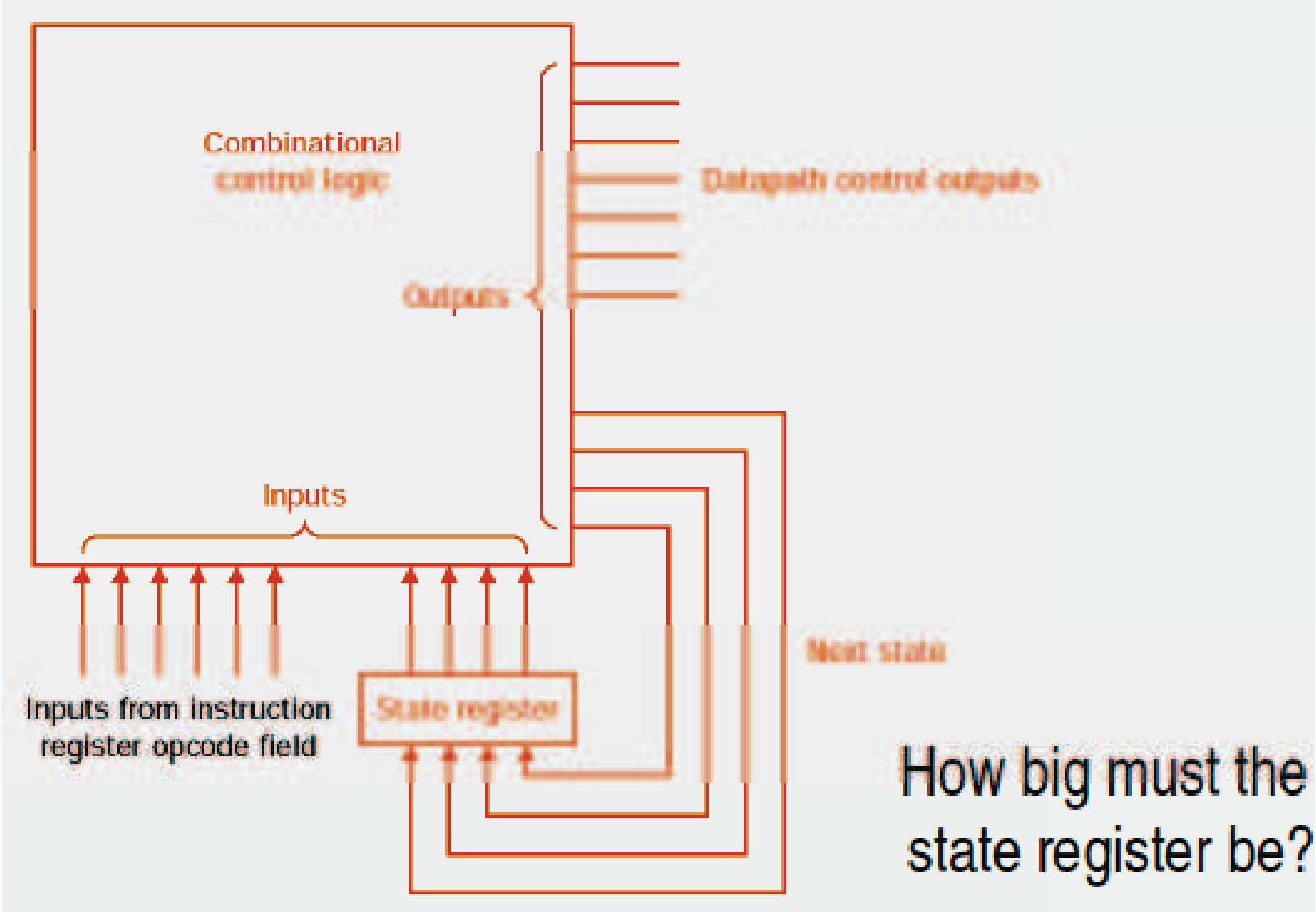

START
INSTRCN. FETCH
INSTRCN. DECODE/ Register FETCH
MMEORY ADDRESS COMPN.
EXECUTION
BRANCH
JUMP
MEMORY ACCESS
R-TYPE COMPN.
WRITE BACK

FSM Graph


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

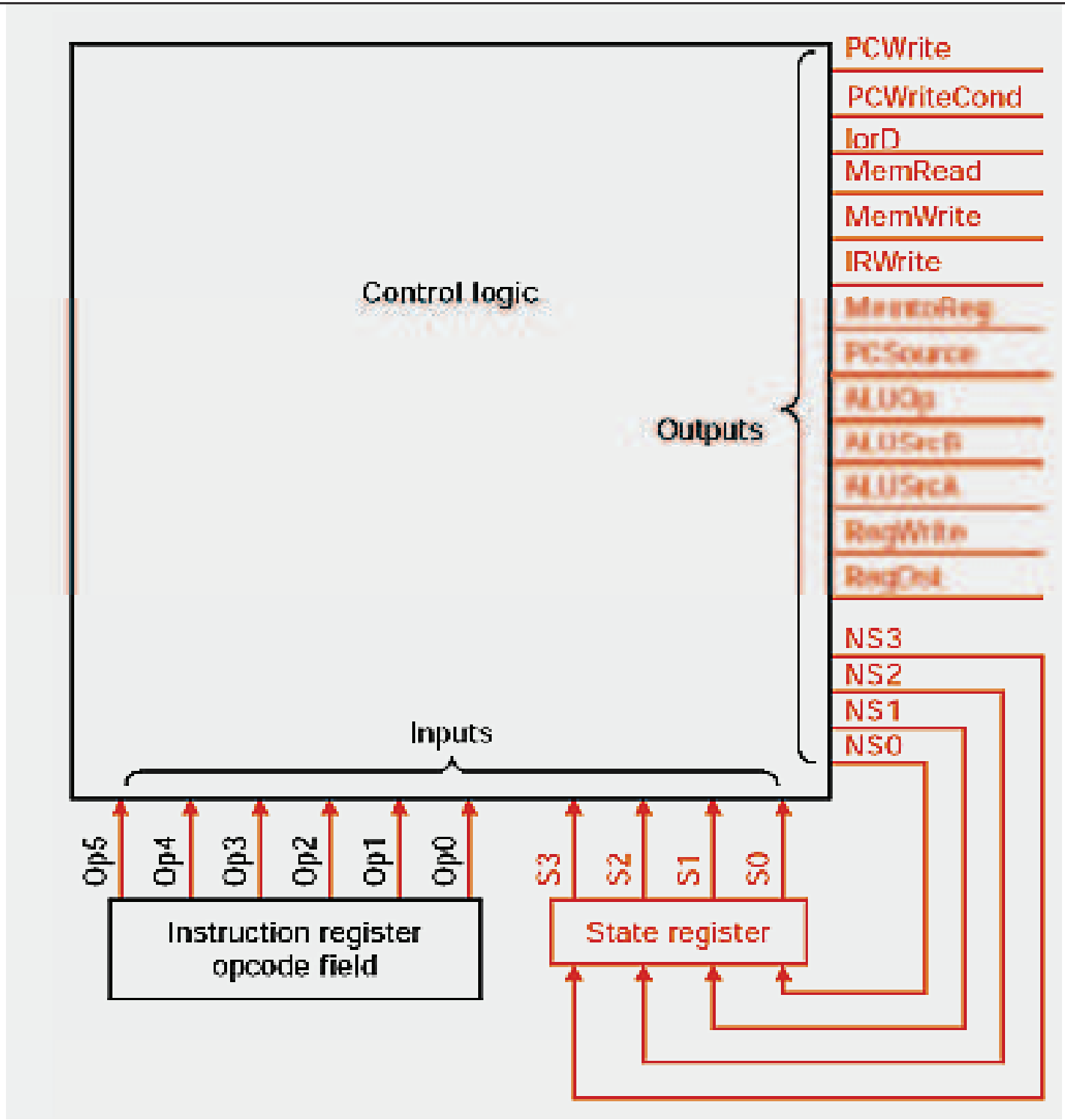

Instruction fetch
Instruction decode/
Register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
Start
(Op = LW) or (Op = SW)
(Op = R-type)
(Op = BEQ)
(Op = J)
Memory-reference FSM
(Figure 5.33)
R-type FSM
(Figure 5.34)
Branch FSM
(Figure 5.35)
Jump FSM
(Figure 5.36)


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
Memory read
completon step
4
RegDst = 1
RegWrite
MemtoReg = 0



Moore type machine **- output signal depends on the current state.**

Next state depends on the input and current state.


Control logic
Outputs
Inputs
PCWrite
PCWriteCond
IorD
MemRead
MemWrite
IRWrite
MemtoReg
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
NS3
NS2
NS1
NS0
Op5
Op4
Op3
Op2
Op1
Op0
S3
S2
S1
S0
Instruction register
opcode field
State register

Output	Current states
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
NextState1	state0
NextState2	state1
NextState3	state2
NextState4	state3
NextState5	state2
NextState6	state1
NextState7	state6
NextState8	state1 (Op = 'beq')
NextState9	state1 (Op = 'jmp')

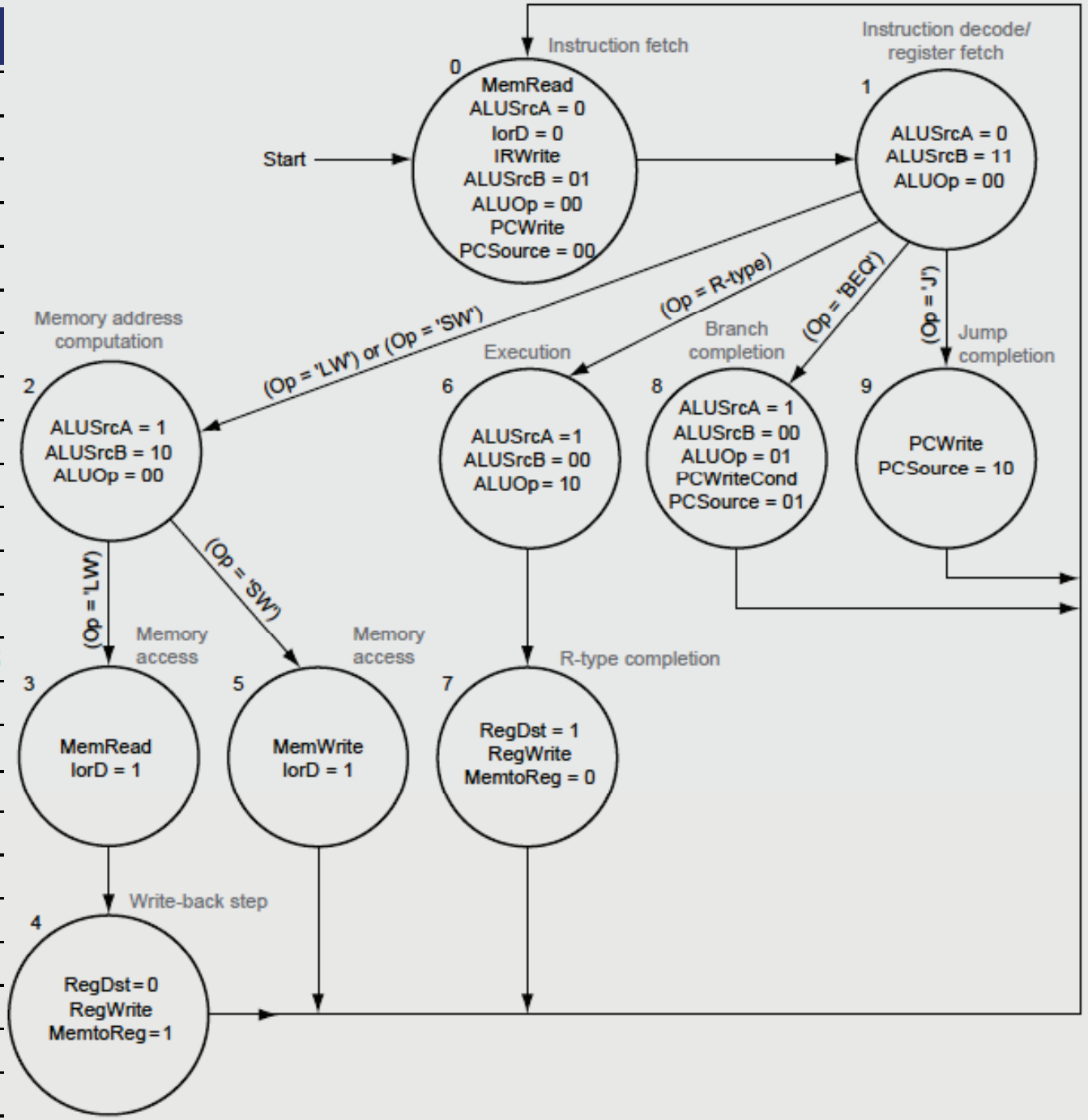


FIGURE C.3.3 The logic equations for the control unit shown in a shorthand form.

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')

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

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

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

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

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

NS0 is the logical sum of all these terms.

FIGURE C.3.3 The logic equations for the control unit shown in a shorthand form.

Break the control function into two parts:

- the next-state outputs, which depend on all the inputs,

and

- the control → signal outputs, which depend only on the current-state bits

Let's look at a ROM-based implementation, first.

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

FIGURE C.3.4 The truth tables are shown for the 16 datapath control signals that depend only on the current-state input bits, which are shown for each table. Each truth table row corresponds to 64 entries: one for each possible value of the 6 Op bits. Notice that

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

FIGURE C.3.6 The truth table for the 16 datapath control outputs, which depend only on the state inputs. The values are determined from Figure C.3.4. Although there are 16 possible values for the 4-bit state field, only 10 of these are used and are shown here. The 10 possible values are shown at the

Lower 4 bits of the address	Bits 19–4 of the word
0000	100101000001000
0001	000000000011000
0010	000000000010100
0011	001100000000000
0100	000000100000010
0101	001010000000000
0110	000000001000100
0111	000000000000011
1000	010000010100100
1001	100000100000000

FIGURE C.3.7 The contents of the upper 16 bits of the ROM depend only on the state inputs. These values are the same as those in Figure C.3.6, simply rotated 90°. This set of control words would be duplicated 64 times for every possible value of the upper 6 bits of the address.

e.g.: PCWrite is high in states 0 and 9; this corresponds to addresses with the 4 low-order bits being either 0000 or 1001. The bit will be high in the memory word independent of the inputs Op[5–0], so the addresses with the bit high are 0000000000, 0000001001, 0000010000, 0000011001, . . . , 1111110000, 1111111001.

The general form of this is XXXXXX0000 or XXXXXX1001.

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.

The truth table for next-state output bit (NS[0]).

The next-state outputs depend on the value of Op[5–0], which is the opcode field, and the current state, given by S[3–0]

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

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

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

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

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

NS0 is the logical sum of all these terms.

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.

The four truth tables for the four next-state output bits (NS[3–0]).

The next-state outputs depend on the value of Op[5–0], which is the opcode field, and the current state, given by S[3–0].

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.

	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

FIGURE C.3.8 This table contains the lower 4 bits of the control word (the NS outputs), which depend on both the state inputs, S[3–0], and the opcode, Op [5–0], which correspond to the instruction opcode. These values can be determined from Figure C.3.5. The opcode name is shown under the encoding in the heading. The 4 bits of the control word whose address is given by the current-state bits and Op bits are shown in each entry. For example, when the state input bits are 0000, the output is always 0001, independent of the other inputs; when the state is 2, the next state is don't care for three of the inputs, 3 for lw, and 5 for sw. Together with the entries in Figure C.3.7, this table specifies the contents of the control unit ROM. For example, the word at address 1000110001 is obtained by finding the

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

The entry from the top yields **0000000000011000**, while the appropriate entry in the table below is **0010**. Thus the control word at address **1000110001** is **0000000000110000010**.

The column labeled "Any other value" applies only when the Op bits do not match one of the specified opcodes.

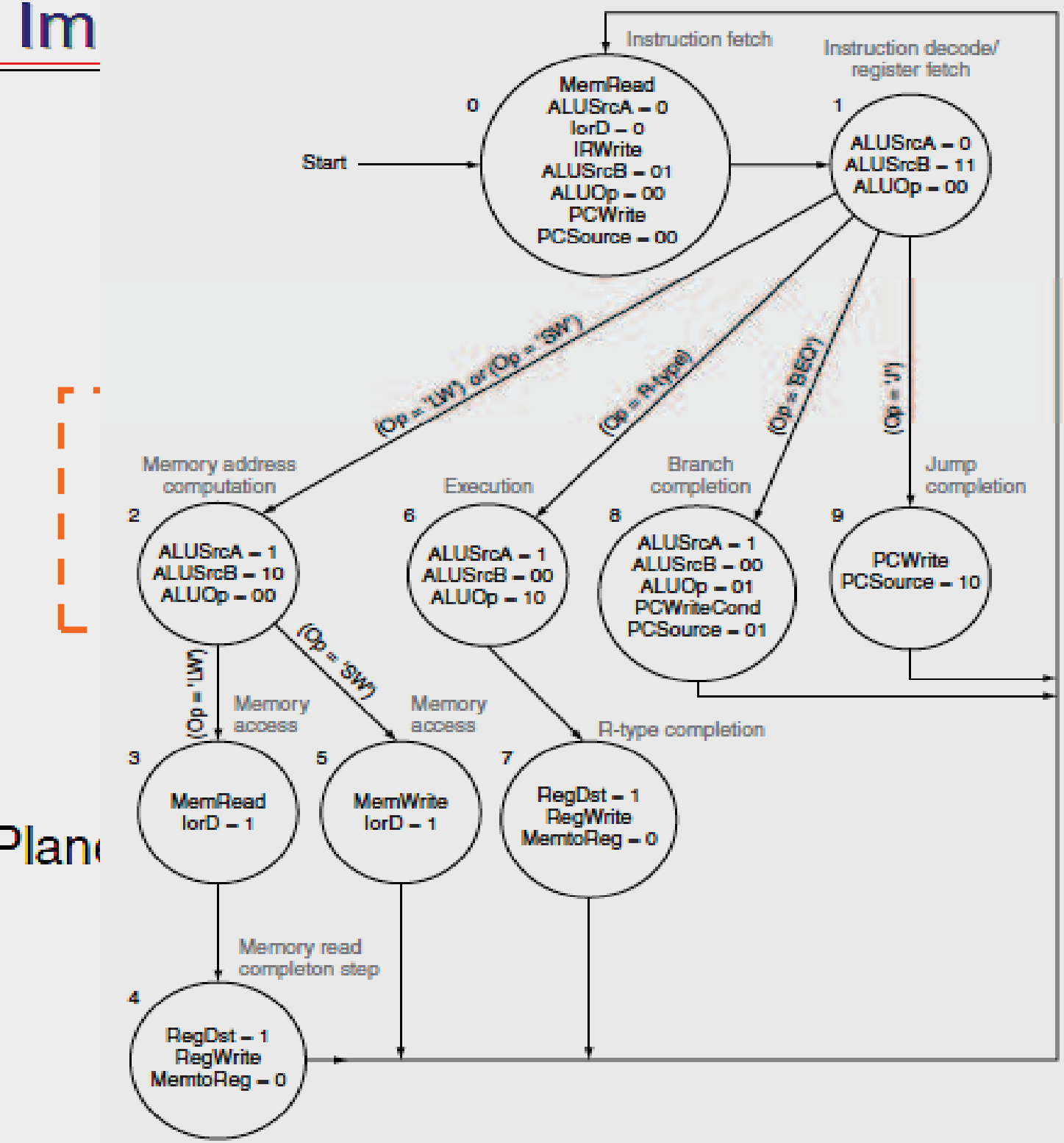
	Op [5–0]					
Current state S[3–0]	000000 (R-format)	000010 (jmp)	000100 (beq)	100011 (lw)	101011 (sw)	Any other value
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	Illegal
0010	XXXX	XXXX	XXXX	0011	0101	Illegal
0011	0100	0100	0100	0100	0100	Illegal
0100	0000	0000	0000	0000	0000	Illegal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

For example, the word at address **1000110001** is obtained by finding (i) the upper 16 bits from the table on top, using only the state input bits (**0001**) and (ii) concatenating the lower 4 bits found by using the entire address (**0001** to find the row and **100011** to find the column).

For ALU Control & simple CPU control lines – check slides:

32 - 35

PLA Im



AND-Plane

OR-Plan

Cond

TYPES of PLDS:

- **PAL - PAL devices have arrays of transistor cells arranged in a "fixed-OR, programmable-AND" plane used to implement "sum-of-products" binary logic equations**

- **PLA - The PLA (also FPLA) has a set of programmable AND gate planes, which link to a set of programmable OR gate planes, which can then be conditionally complemented to produce an output. This layout allows for a large number of logic functions to be synthesized in the sum of products (and sometimes product of sums) in canonical forms.**

- **GAL - The GAL (Generic Array Logic) was an improvement on the PAL because one device was able to take the place of many PAL devices or could even have functionality not covered by the original range. Its primary benefit, however, was that it was erasable and re-programmable making prototyping and design changes easier for engineers.**

- **A similar device called a PEEL (programmable electrically erasable logic) was introduced by the International CMOS Technology (ICT) corporation.**

- **FPGA - FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" - somewhat like a one-chip programmable breadboard.**

The most common FPGA architecture consists of an array of configurable logic blocks (CLBs), I/O pads, and routing channels. Generally, all the routing channels have the same width (number of wires). Multiple I/O pads may fit into the array – programmable using HDL.

- **CPLD – between PALs and FPGAs. Has ROM and hence non-volatile. Handles complex logics with feedback and arithmetic operations.**

- **ROM –**

- **PLC** - Automation of machinery control – a small embedded system

- **PLL ??**

Various optimizers and sequencers are used for efficient design.

Difficult to design when complex operations/instructions are necessary –
Floating point, superscalar, pipelining etc.

Correcting errors and debugging is difficult

How do you implement W(MFC) in this state machine ??

Minor modifications of the ISA requires lot of changes and redo the design.

Complex instructions may require to go through several states and signals to be generated

Many opcodes – the design may require a RISE lab./hall for generating the truth table.

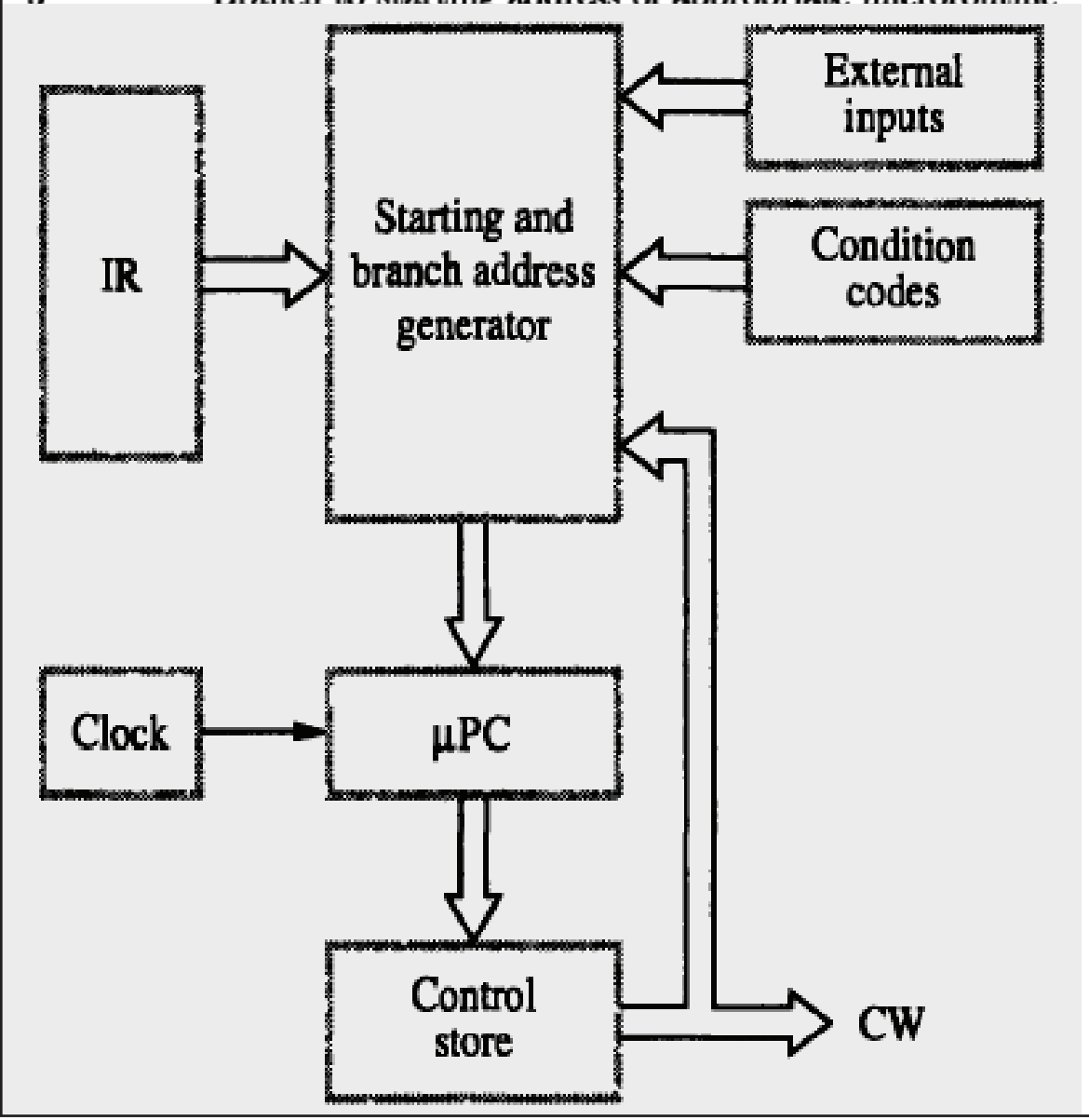
Microprogramming

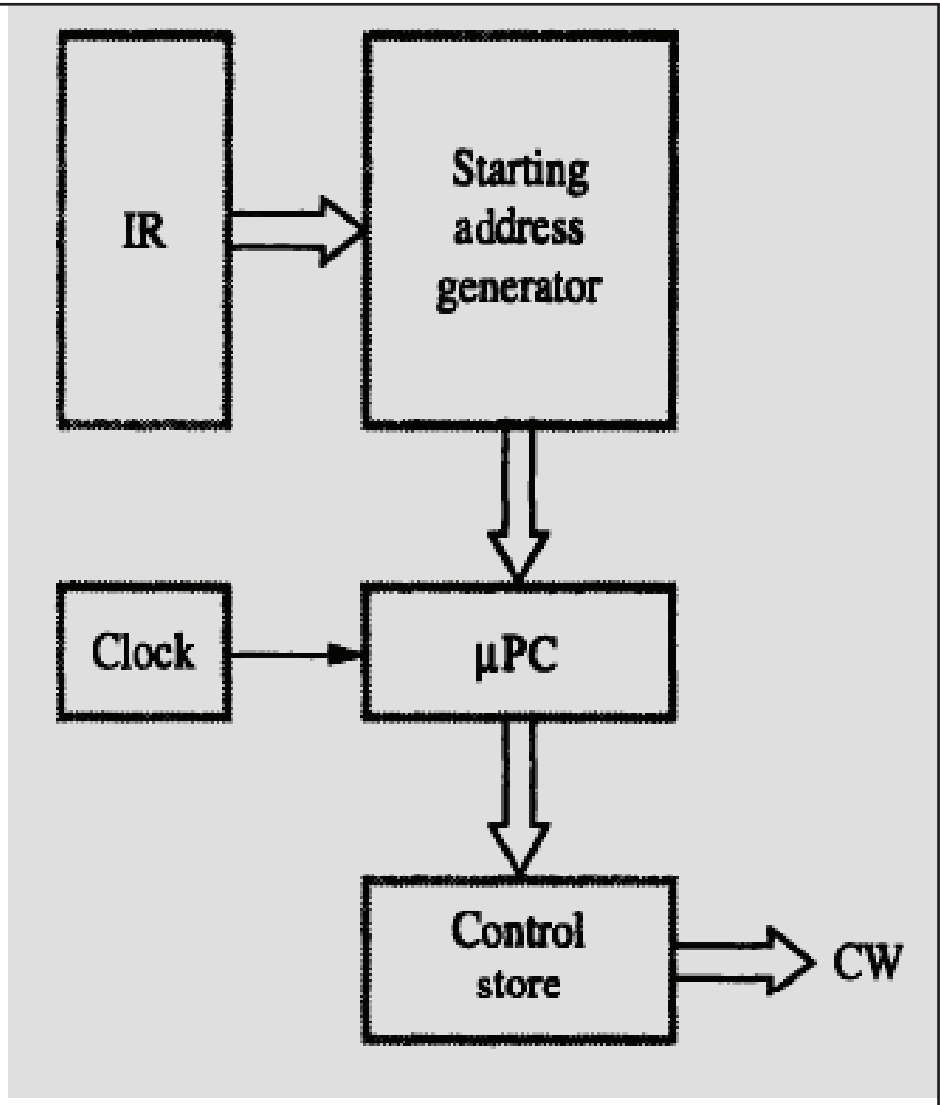
Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

← Micro-Instructions for:

instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1

Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine





The *μPC* is incremented every time a new micro-instruction is fetched from the micro-program (Control Store) memory, except in the following situations:

1. When a new instruction is loaded into the IR, the *μPC is loaded with the starting* address of the micro-routine for that instruction.

2. When a Branch microinstruction is encountered and the branch condition is satisfied, the *μPC is loaded with the branch address.*

3. When an End microinstruction is encountered, the *μPC is loaded with the address* of the first CW in the microroutine for the instruction fetch cycle (this address is 0).

Drawbacks of this simple micro-instrcn. system:

- assigning individual bits to each control signal results in long microinstructions because the number of required signals is usually large.

- only a few bits are set to 1 (for active gating) in any given microinstruction, which means the available bit space is poorly used.

Assume:

In total, 42 control signals are needed.

e.g.

- Read, Write, Select, WMFC, End;
- Add, Subtract, AND, and XOR;
- Separate signals to R_i's ; PC, IR, MAR, MDR etc.

42 bits would be needed in each microinstruction. Fortunately, the length of the microinstructions can be reduced easily. Most signals are not needed simultaneously, and many signals are mutually exclusive.

For example, only one function of the ALU can be activated at a time. The source for a data transfer must be unique because it is not possible to gate the contents of two different registers onto the bus at the same time. Read and Write signals to the memory cannot be active simultaneously.

This suggests that signals can be grouped so that all mutually exclusive signals are placed in the same group. Thus, at most one *microoperation per group is specified in any microinstruction*

For example, four bits suffice to represent the 16 available functions in the ALU.

Register output control signals can be placed in a group consisting of PC_{out}, MDR_{out}, Z_{out}, $Offset_{out}$, $R0_{out}$, $R1_{out}$, $R2_{out}$, $R3_{out}$ and $TEMP_{out}$.

***Thus, do this natural grouping (of* mutually exclusive signals) and then -**

Select anyone by a 4-bit code.

Most fields must include one inactive code for the case in which no action is required.

Grouping control signals into fields requires a little more hardware because decoding circuits must be used to decode the bit patterns of each field into individual control signals.

The cost of this additional hardware is more than offset by the reduced number of bits in each microinstruction, which results in a smaller CONTROL store.

Microinstruction

F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)	F6 (1 bit)	F7 (1 bit)	F8 (1 bit)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action	0: SelectY	0: No action	0: Continue
0001: PC_{out}	001: PC_{in}	001: MAR_{in}	0001: Sub	01: Read	1: Select4	1: WMFC	1: End
0010: MDR_{out}	010: IR_{in}	010: MDR_{in}	⋮	10: Write			
0011: Z_{out}	011: Z_{in}	011: $TEMP_{in}$					
0100: $R0_{out}$	100: $R0_{in}$	100: Y_{in}	1111: XOR				
0101: $R1_{out}$	101: $R1_{in}$		16 ALU functions				
0110: $R2_{out}$	110: $R2_{in}$						
0111: $R3_{out}$	111: $R3_{in}$						
1010: $TEMP_{out}$							
1011: $Offset_{out}$							

Only 20 bits are needed to store the patterns for the 42 signals

Microinstruction

F1	F2	F3	F4	F5

F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action
0001: PC_{out}	001: PC_{in}	001: MAR_{in}	0001: Sub	01: Read
0010: MDR_{out}	010: IR_{in}	010: MDR_{in}	⋮	10: Write
0011: Z_{out}	011: Z_{in}	011: $TEMP_{in}$		
0100: $R0_{out}$	100: $R0_{in}$	100: Y_{in}	1111: XOR	
0101: $R1_{out}$	101: $R1_{in}$		16 ALU functio	
0110: $R2_{out}$	110: $R2_{in}$			
0111: $R3_{out}$	111: $R3_{in}$			
1010: $TEMP_{out}$				
1011: $Offset_{out}$				

VERTICAL ORGANIZATION is also possible, where compact codes are generated using highly encoded schemes.

HORIZONTAL ORGANIZATION

Micro-instruction	:	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1

MICROPROGRAM SEQUENCING

Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control store.

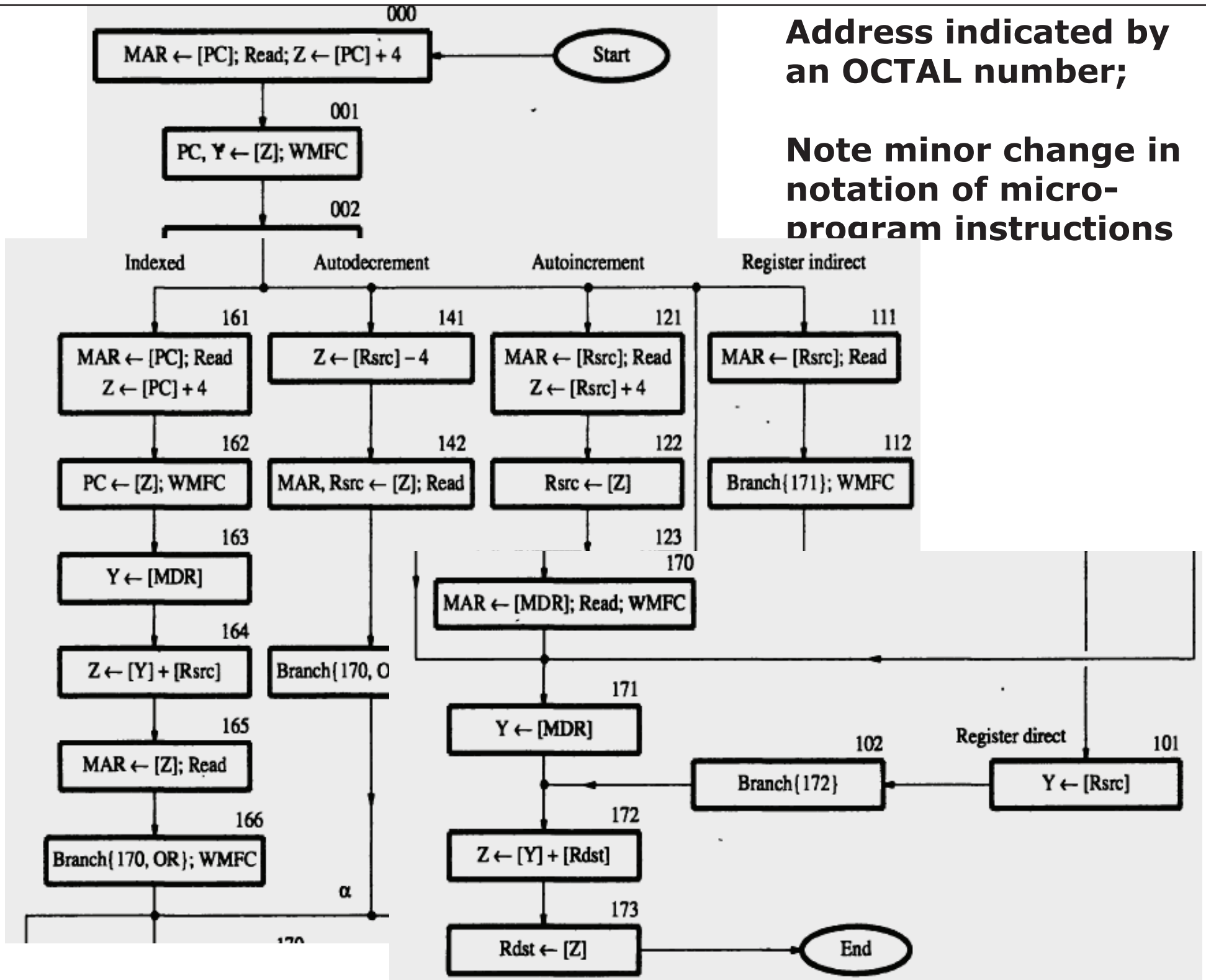
If most machine instructions involve several addressing modes, there can be many instruction and addressing mode combinations. A separate microroutine for each of these combinations would produce considerable duplication of common parts.

Its better to organize the microprogram so that the microroutines share as many common parts as possible. This requires many branch microinstructions to transfer control among the various parts.

e.g. Consider an instruction of the type:

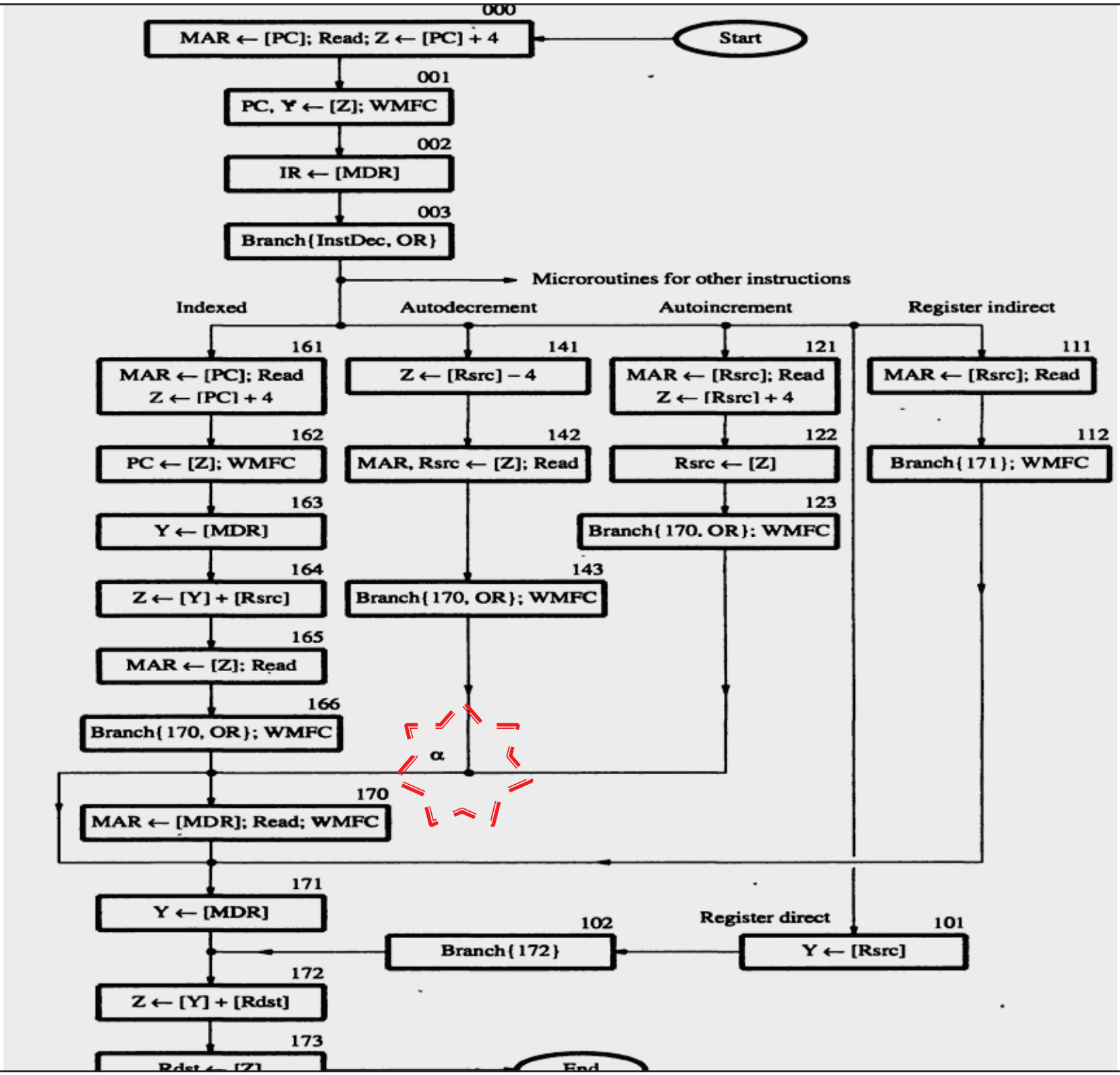
Add R_{src}, R_{dst}

Addressing modes:
register, autoincrement, autodecrement, and indexed, as well as the indirect forms of these four modes.



Address indicated by an OCTAL number;

Note minor change in notation of micro-program instructions


000
MAR ← [PC]; Read; Z ← [PC] + 4
Start
001
PC, Y ← [Z]; WMFC
002
IR ← [MDR]
003
Branch{InstDec, OR}
Microroutines for other instructions
Indexed
Autodecrement
Autoincrement
Register indirect
161
MAR ← [PC]; Read
Z ← [PC] + 4
141
Z ← [Rsrc] – 4
121
MAR ← [Rsrc]; Read
Z ← [Rsrc] + 4
111
MAR ← [Rsrc]; Read
162
PC ← [Z]; WMFC
142
MAR, Rsrc ← [Z]; Read
122
Rsrc ← [Z]
112
Branch{171}; WMFC
163
Y ← [MDR]
123
Branch{170, OR}; WMFC
164
Z ← [Y] + [Rsrc]
143
Branch{170, OR}; WMFC
165
MAR ← [Z]; Read
166
Branch{170, OR}; WMFC
α
170
MAR ← [MDR]; Read; WMFC
171
Y ← [MDR]
102
Branch{172}
Register direct
101
Y ← [Rsrc]
172
Z ← [Y] + [Rdst]
173
Rdst ← [Z]
End

Branch Address Modification using Bit-ORing

Consider the point labeled "α" *in the figure. At this point,* it is necessary to choose between actions required by direct and indirect addressing modes.

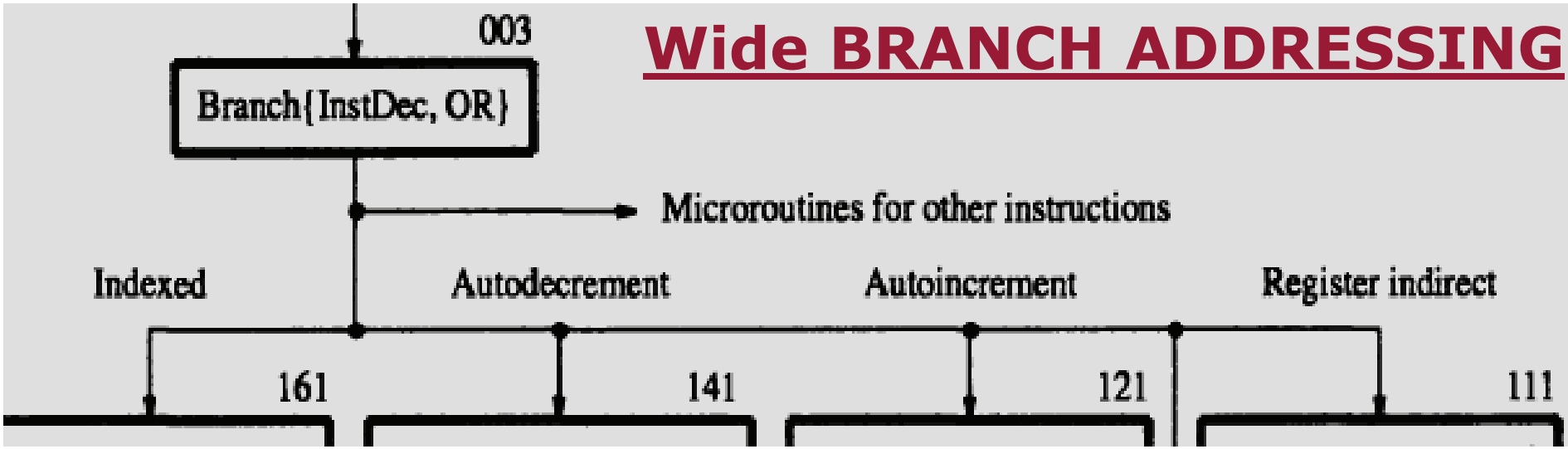
If the indirect mode is specified in the instruction, then the microinstruction in location 170 is performed to fetch the operand from the memory.

If the direct mode is specified, this fetch must be bypassed by branching immediately to location 171.

The most efficient way to bypass microinstruction 170 is to have the preceding branch microinstructions specify the address 170 and then use an OR gate to change the least significant bit of this address to 1 if the direct addressing mode is involved.

This is known as the *bit-ORing technique* for modifying *branch addresses.*

Wide BRANCH ADDRESSING



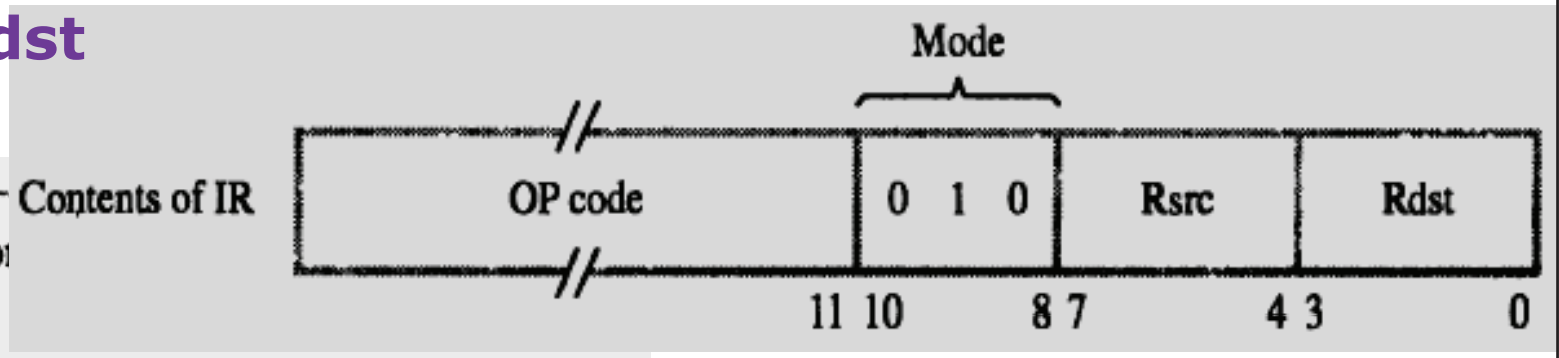
The instruction decoder {InstDec}, generates the starting address of the microroutine that implements the instruction that has just been loaded into the IR.

In our example, register IR contains the Add instruction, for which the instruction decoder generates the micro-instruction address 101, which cannot be loaded as is into the microprogram counter (*μPC*).

The bit-ORing technique can be used at this point to modify the starting address generated by the instruction decoder to reach the appropriate path.

Bit-Oring should change the address 101 to one of the five possible address values, 161, 141, 121, 101, or 111, depending on the addressing mode used in the instruction

Execute the instruction -
Add (Rsrc)+, Rdst



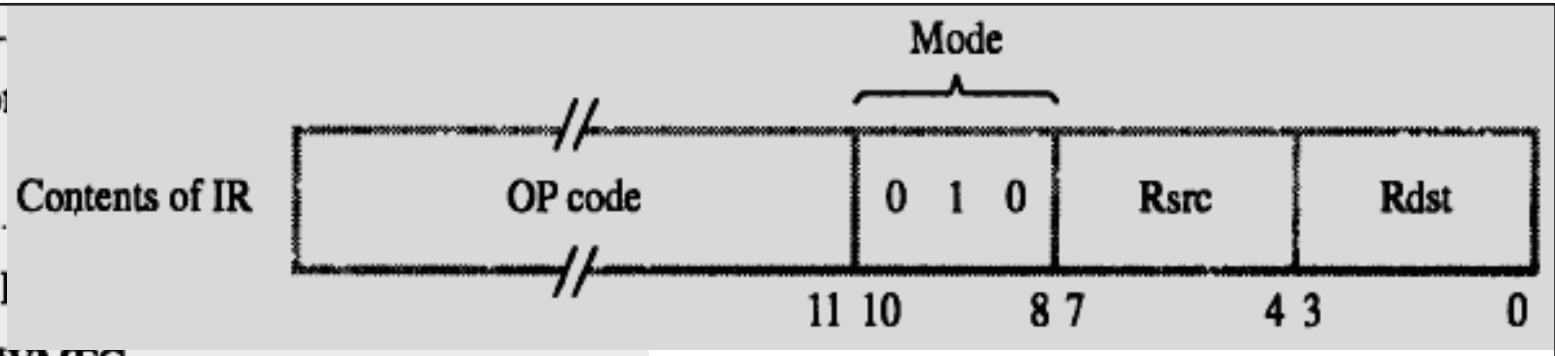
Address (octal)	Microinstruction
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch {$\mu PC \leftarrow 101$ (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	μBranch {$\mu PC \leftarrow 170$; $\mu PC_0 \leftarrow [\overline{IR_8}]$}, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Microinstruction for Add (Rsrc)+,Rdst.

The instruction has a 3-bit field to specify the addressing mode for the source operand, as above.

Bit patterns:
11, 10, 01, and 00, located in bits 10 and 9, denote the indexed, autodecrement, autoincrement, and register modes, respectively.

For each of these modes, bit 8 is used to specify the indirect version.



Address (octal)	Microinstruction
000	PC_{out}, MAR_{in},
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch {μPC ← 101 (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$}
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	μBranch {μPC ← 170; $\mu PC_0 \leftarrow [\overline{IR_8}]$}, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Microinstruction for Add (Rsrc)+,Rdst.

Add (Rsrc)+, Rdst;

IR_{10-9} for Auto-increment mode: 01;

IR_8 = 0 (no Indirect);

Thus,

$\mu PC_{5-3} = (010)_2 = (2)_8$;

Modified μPC for branching after $(003)_8 = (121)_8$;

Modified μPC for branching after $(123)_8 = (171)_8$; //Direct mode