CPU Organization –

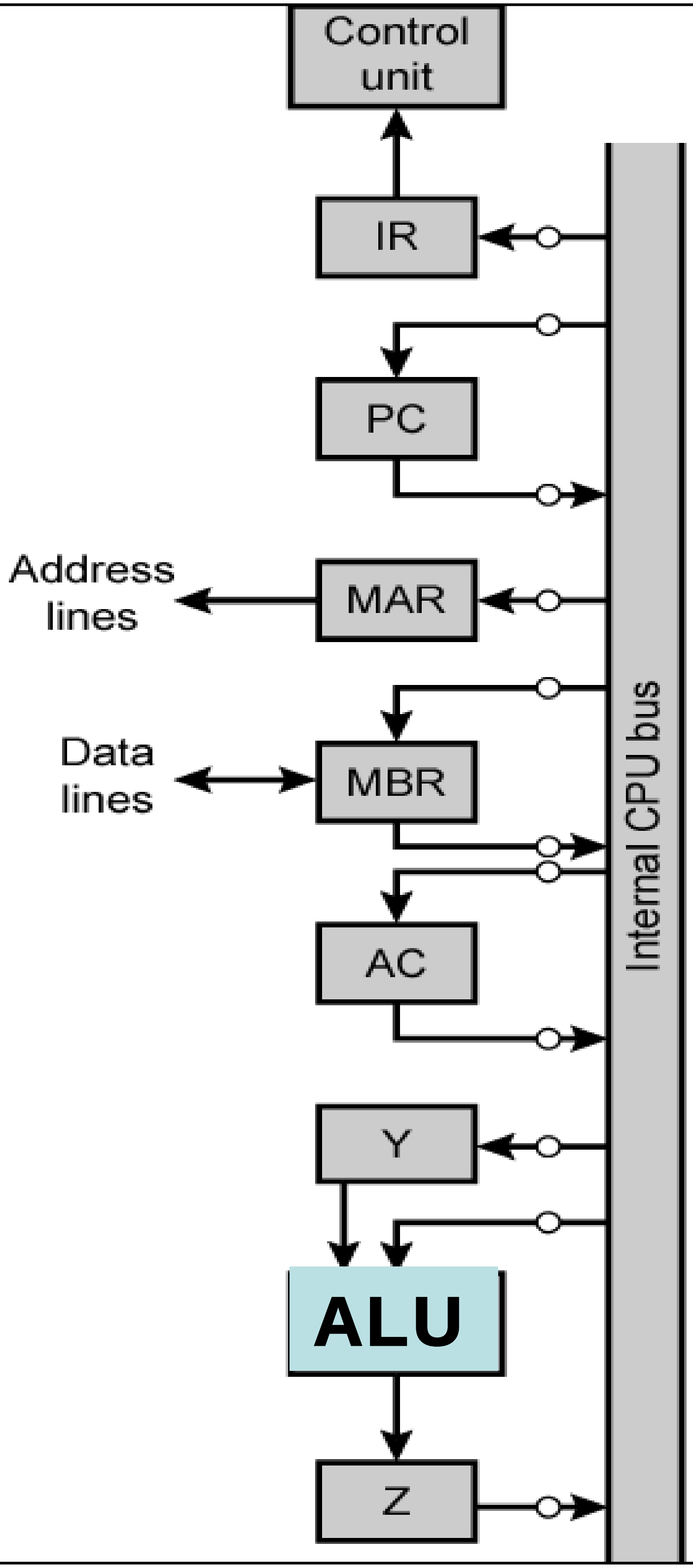
Hardware design

Vs.

Microprogramming

CPU Structure

- CPU must:
  - Fetch instructions
  - Interpret instructions
  - Fetch data
  - Process data
  - Write data



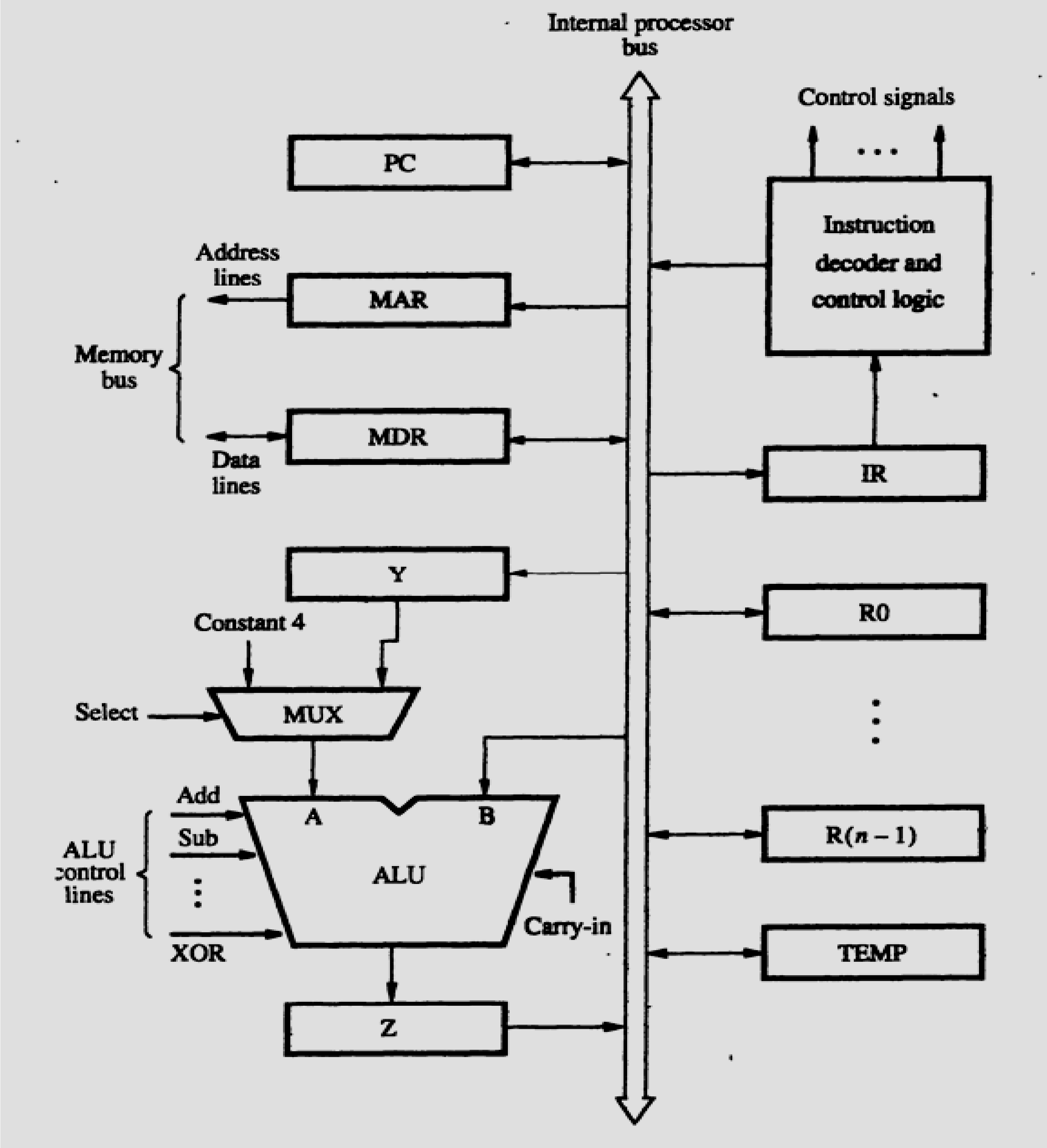
Source:
Hamacher;

Single-bus ORGN.

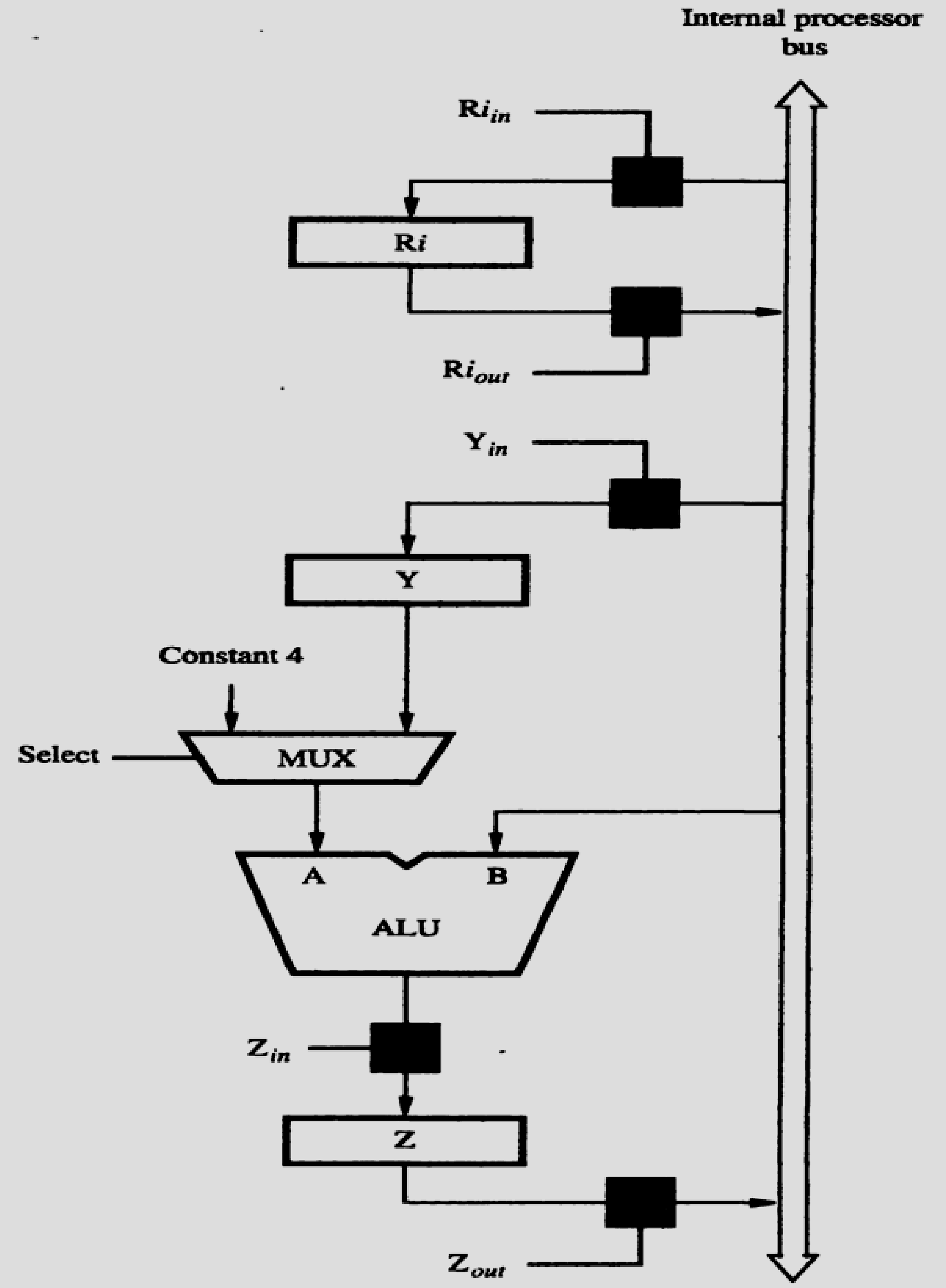
CPU always stores the results of most calculations in one special register —

typically called
“the Accumulator” of that CPU

Single Bus Architecture

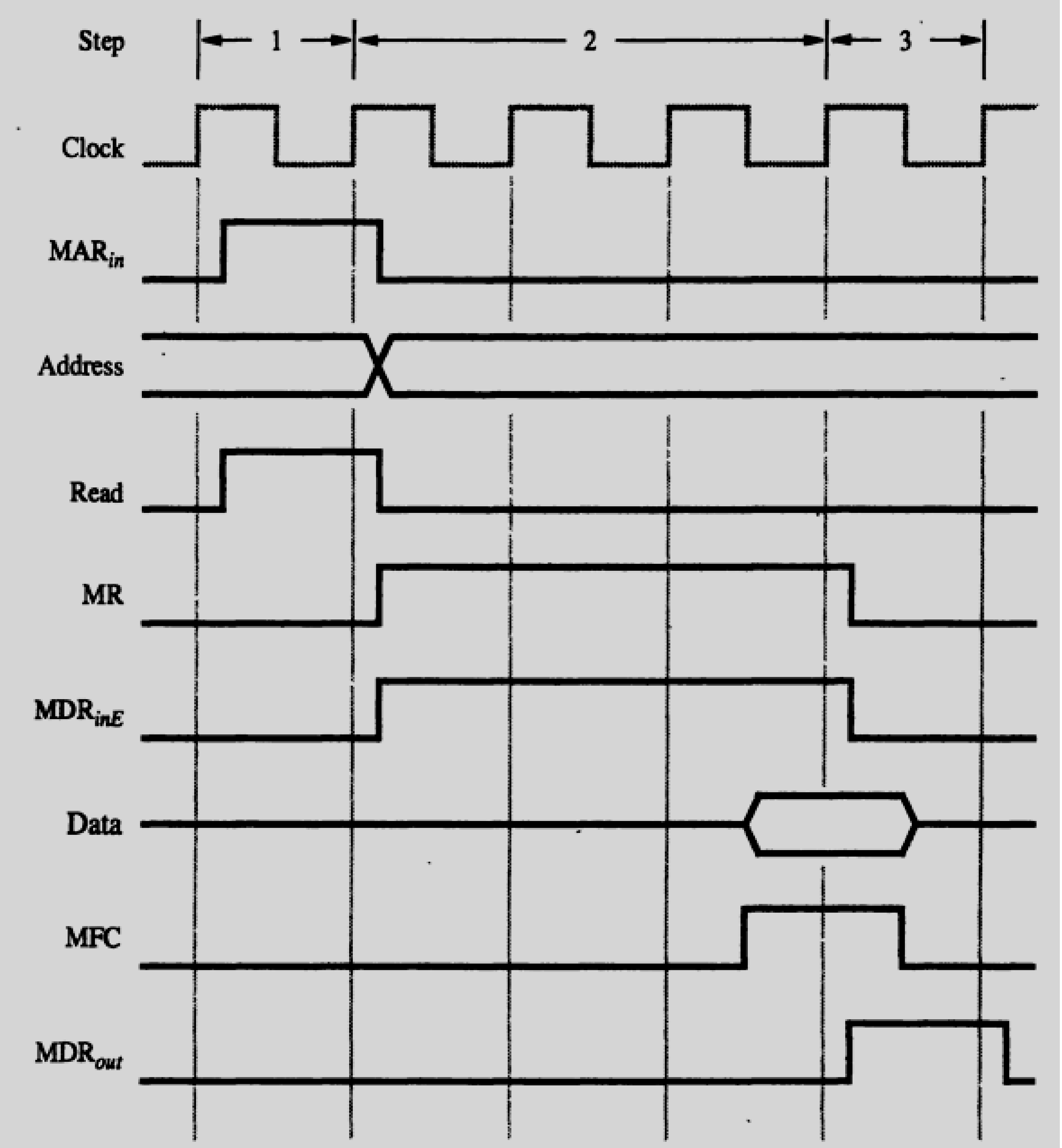


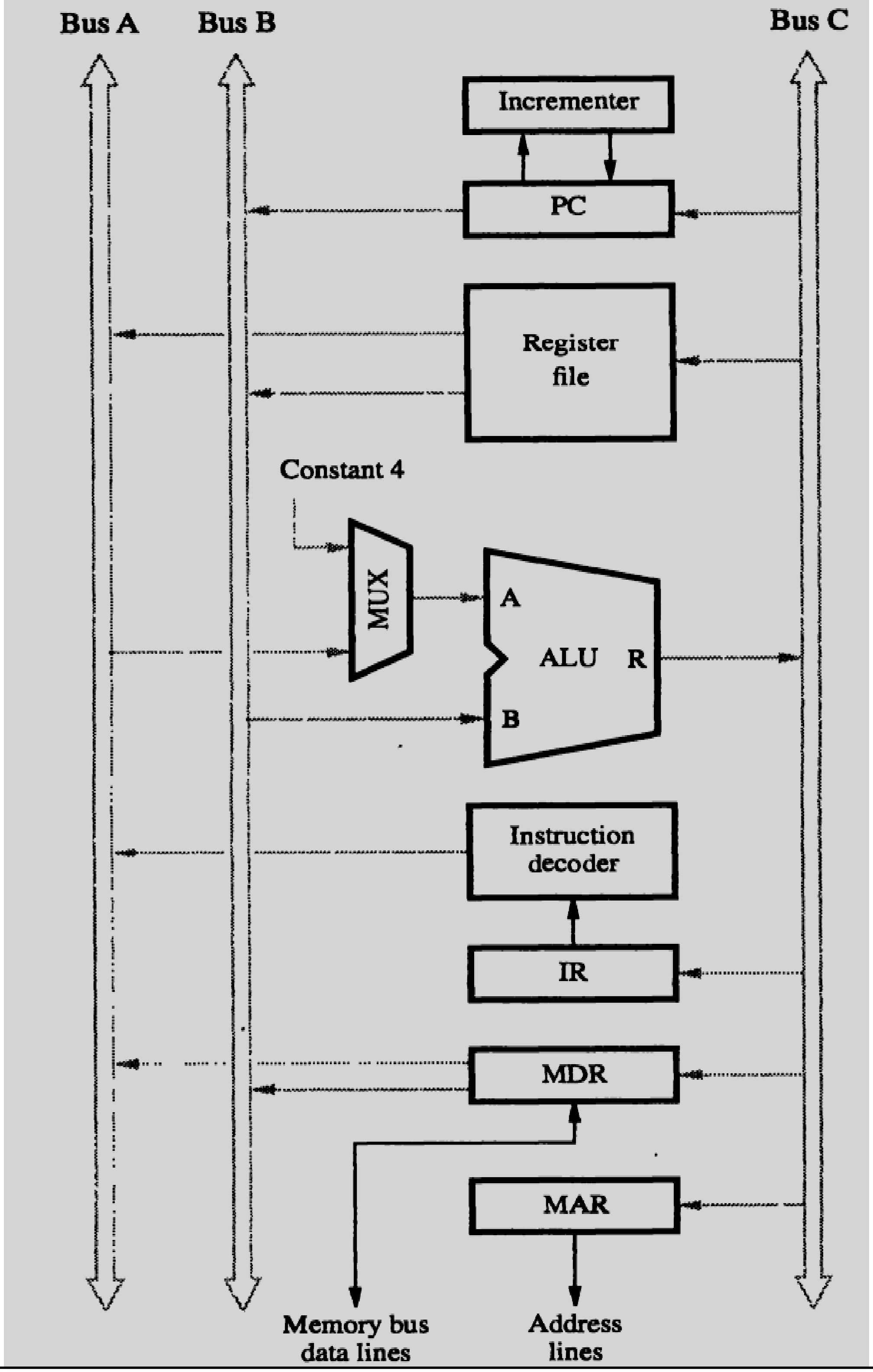
Input-Output Gating of the registers in the single-bus architecture



Timing Diagram of a:

Memory Read operation

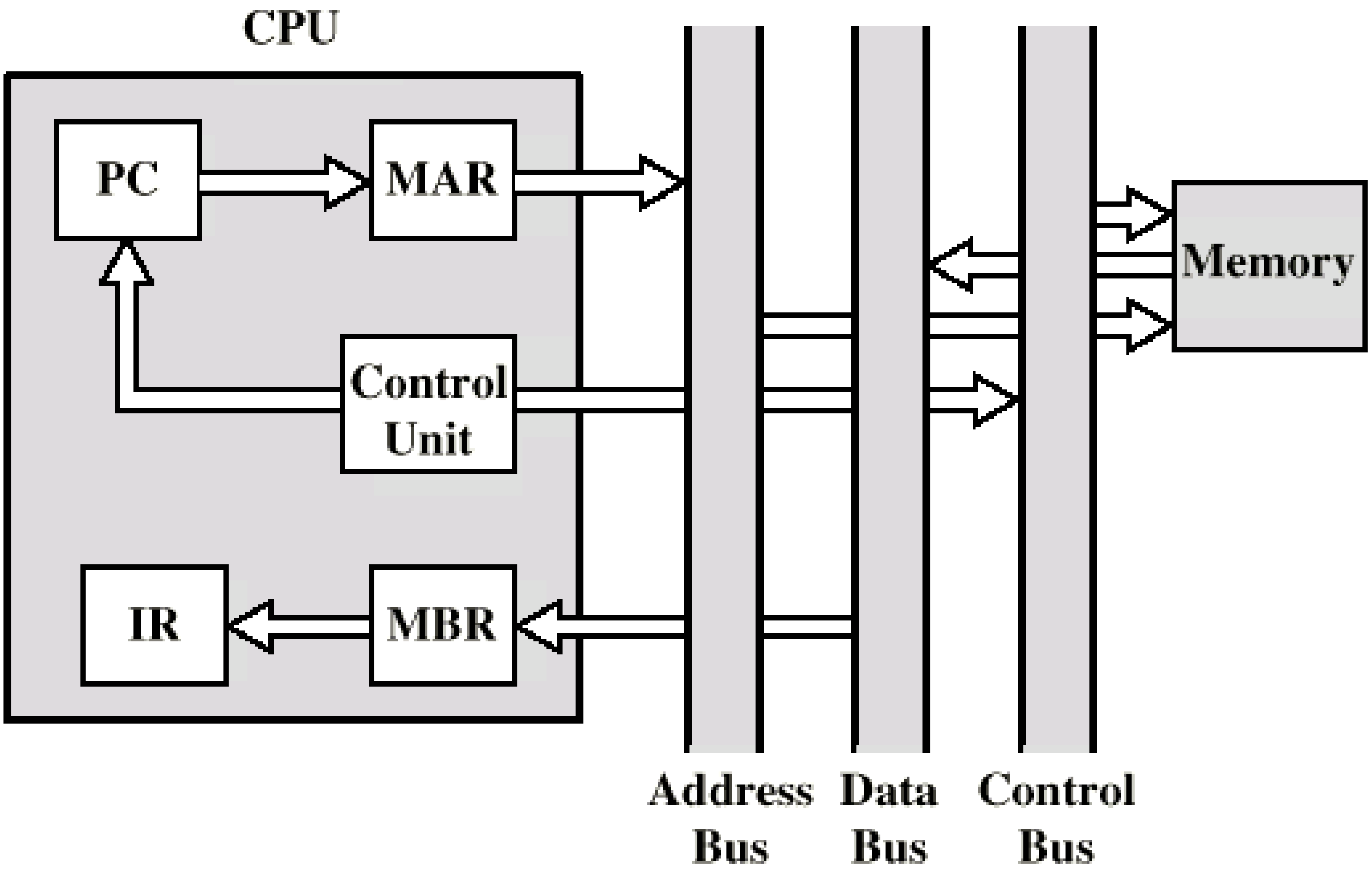




Three-bus organization of datapath;

Or CPU architecture

Data Flow (Fetch Operation)



MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Source: Stallings

MIPS (Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA).

Computer workstation systems using MIPS processors are:

SGI, MIPS Computer Systems, Inc., Whitechapel Workstations, Olivetti, Siemens-Nixdorf, Acer, Digital Equipment Corporation, NEC, and DeskStation.

Operating systems ported to the architecture include:

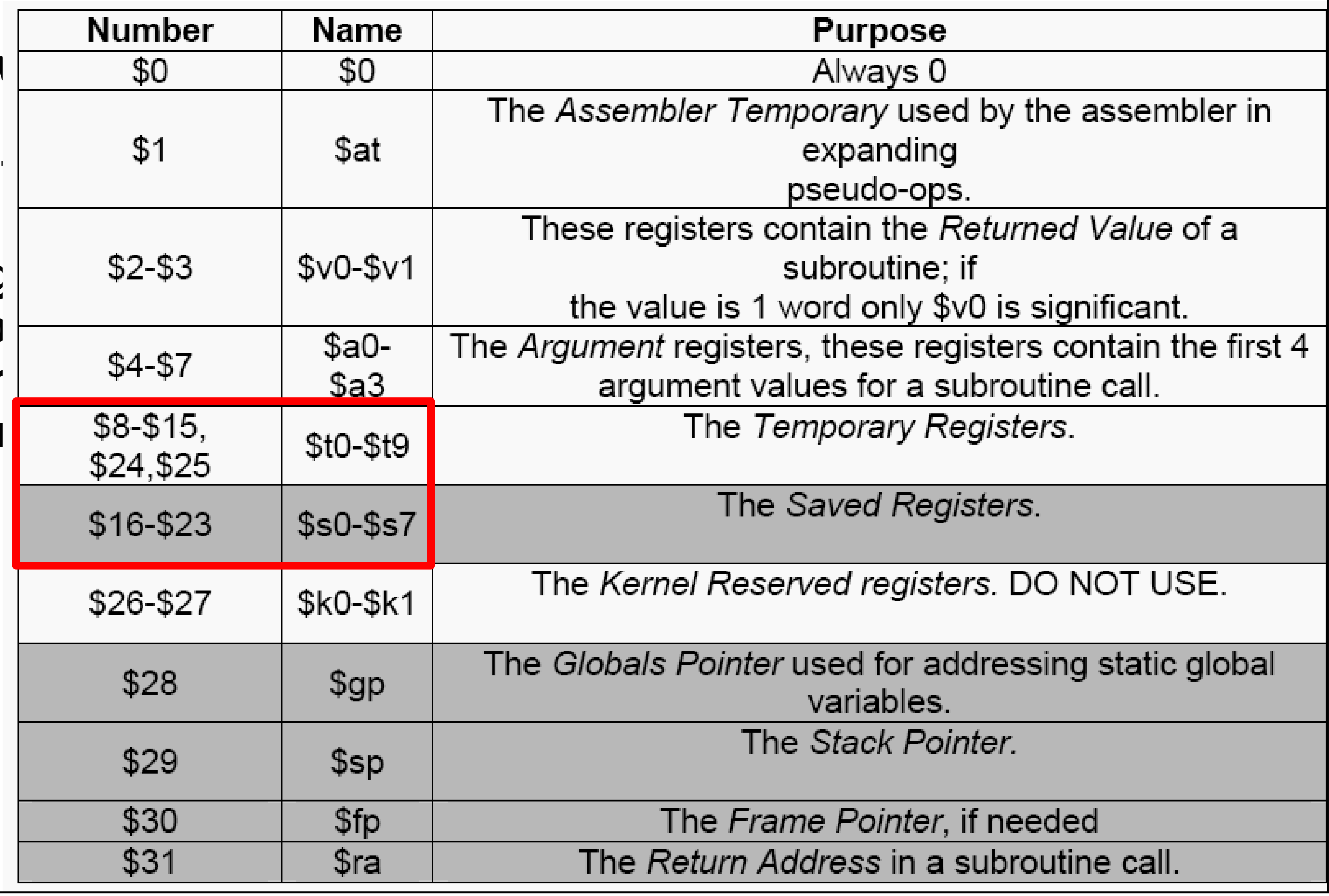
SGI's IRIX, Microsoft's Windows NT (until v4.0), Windows CE, Linux, BSD, UNIX System V, SINIX, QNX.

The R8000 (1994) was the first superscalar MIPS design, able to execute two integer or floating point and two memory instructions per cycle. The design was spread over six chips: an integer unit (with 16 KB instruction and 16 KB data caches), a floating-point unit, three full-custom secondary cache tag RAMs (two for secondary cache accesses, one for bus snooping), and a cache controller ASIC.

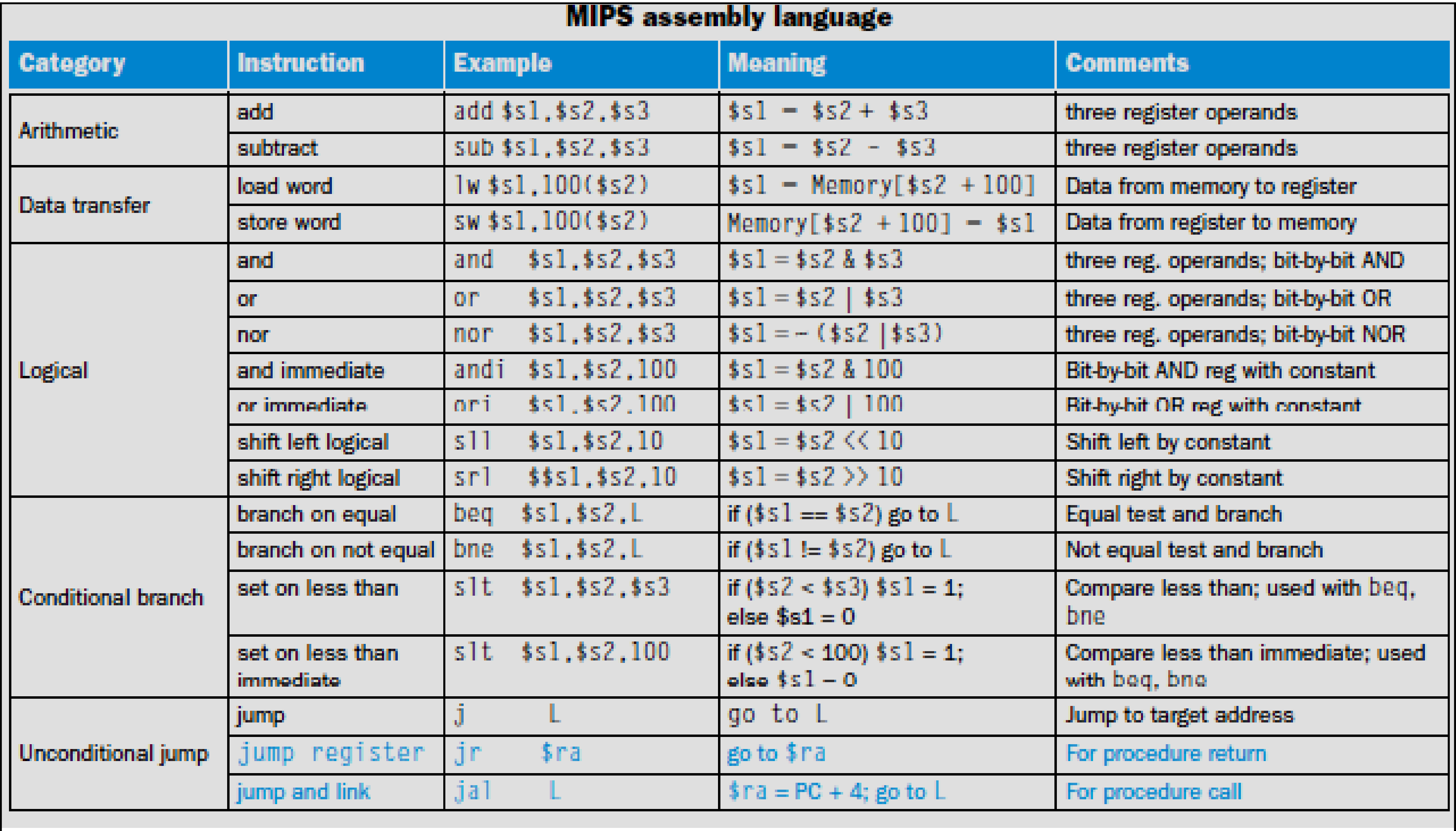
The design had two fully pipelined double precision multiply-add units, which could stream data from the 4 MB off-chip secondary cache.

MIPS instruction set

- 32 general purpose registers
- Backwards compatible
- 32 bit w
- Load-s
- 3 categ
  - –Lo
  - –Ar
  - –Ju



Number	Name	Purpose
$0	$0	Always 0
$1	$at	The *Assembler Temporary* used by the assembler in expanding pseudo-ops.
$2-$3	$v0-$v1	These registers contain the *Returned Value* of a subroutine; if the value is 1 word only $v0 is significant.
$4-$7	$a0-$a3	The *Argument* registers, these registers contain the first 4 argument values for a subroutine call.
$8-$15, $24,$25	$t0-$t9	The *Temporary Registers*.
$16-$23	$s0-$s7	The *Saved Registers*.
$26-$27	$k0-$k1	The *Kernel Reserved registers*. DO NOT USE.
$28	$gp	The *Globals Pointer* used for addressing static global variables.
$29	$sp	The *Stack Pointer*.
$30	$fp	The *Frame Pointer*, if needed
$31	$ra	The *Return Address* in a subroutine call.



MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	three register operands
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; used with beq, bne
	set on less than immediate	slt $s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than immediate; used with beq, bne
Unconditional jump	jump	j L	go to L	Jump to target address
	jump register	jr $ra	go to $ra	For procedure return
	jump and link	jal L	$ra = PC + 4; go to L	For procedure call

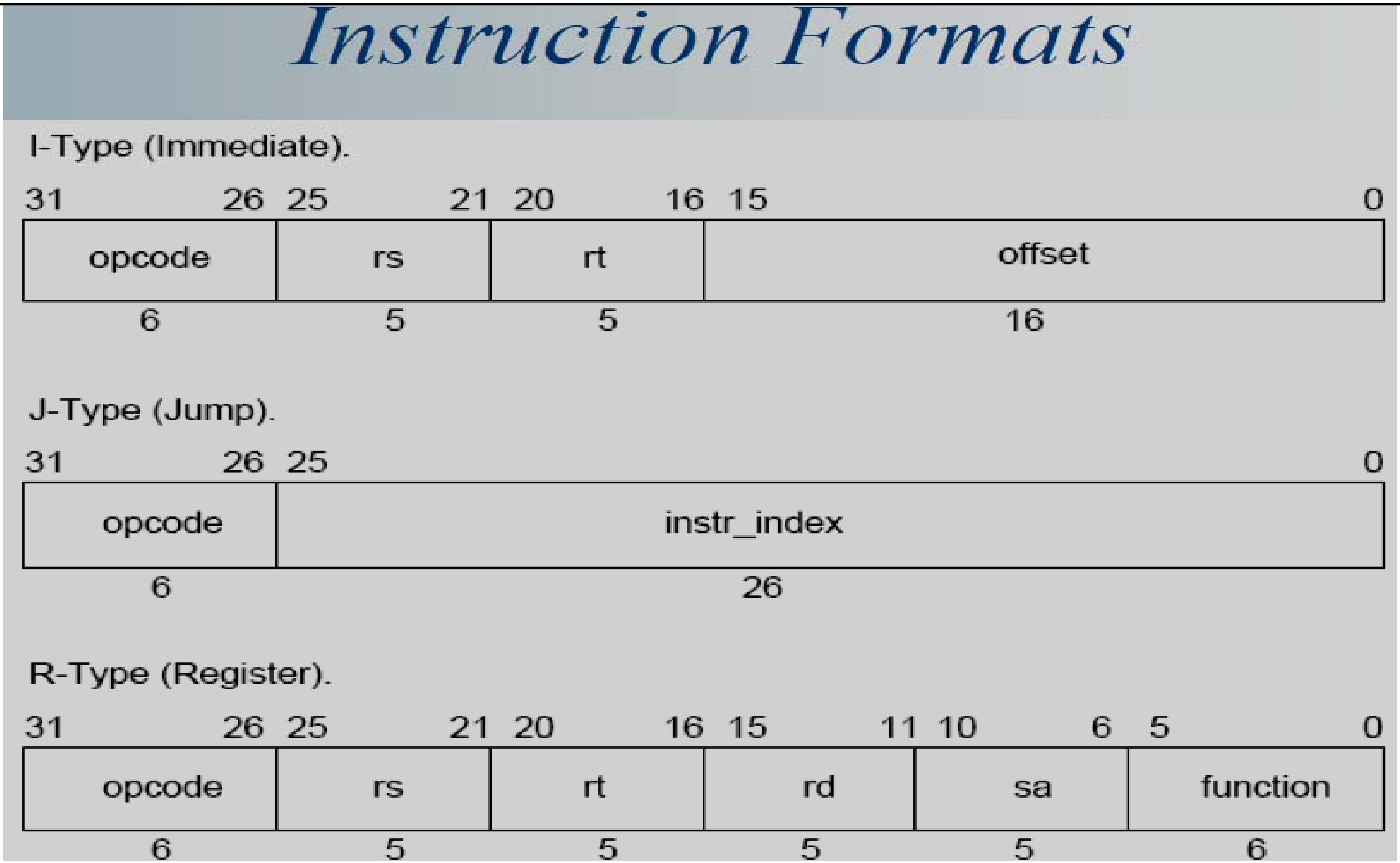
Type	-31- format (bits) -0-					
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	Target Address (26)				

All MIPS instructions are 32 bits long.

The three instruction formats:

- **R-type (Register)**
- **I-type (Immediate)**
- **J-type (Jump)**

- **The different fields are:**
  - **op: operation of the instruction (opcode)**
  - **rs, *rt*, rd: the source and destination register specifiers**
  - **shamt: shift amount (arithmetic/logical)**
  - **funct: selects the variant of the operation in the “op” field**
  - **address / immediate: address offset or immediate value**
  - **target address: target address of the jump instruction**



Instruction Formats

I-Type (Immediate).

31 26	25 21	20 16	15 0
opcode	rs	rt	offset
6	5	5	16

J-Type (Jump).

31 26	25 0
opcode	instr_index
6	26

R-Type (Register).

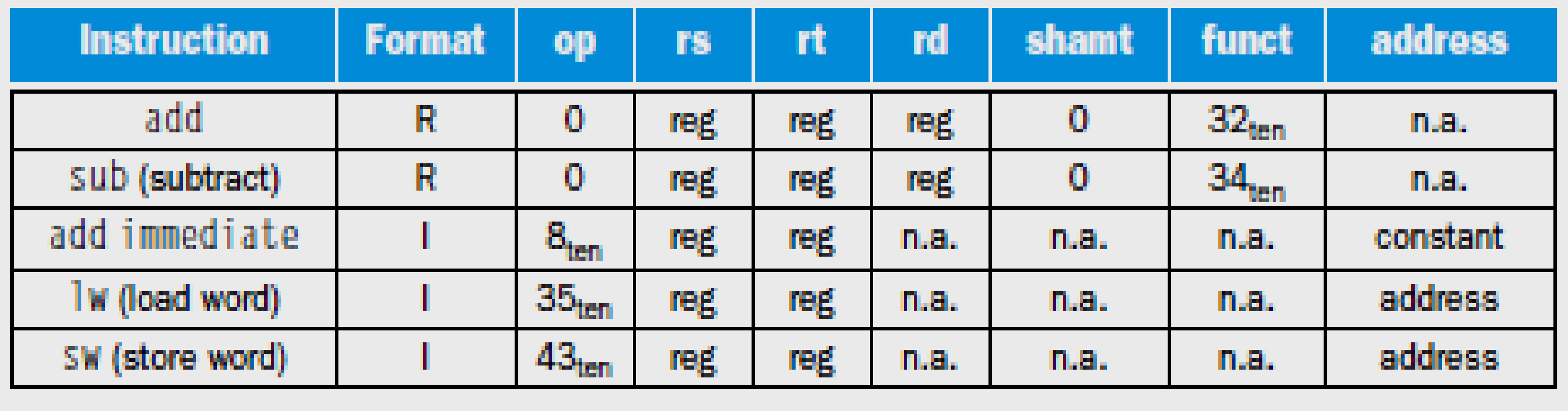
31 26	25 21	20 16	15 11	10 6	5 0
opcode	rs	rt	rd	sa	function
6	5	5	5	5	6

Every instruction : starts with a 6-bit opcode.

In addition to the opcode, R-type instructions specify three registers, a shift amount field, and a function field;

I-type instructions specify two registers and a 16-bit immediate value;

J-type instructions follow the opcode with a 26-bit jump target.



Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

MIPS Instruction encoding

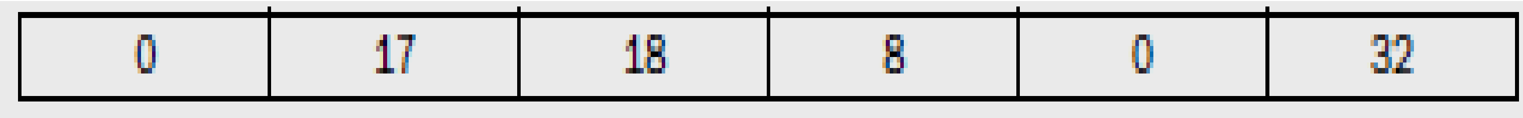
"reg" means a register number between 0 and 31,
"address" means a 16-bit address, and
"n.a." (not applicable) means this field does not appear in this format.

add and sub instructions have the same value in the op field; the hardware uses the funct field to decide the variant of the operation: add (32) or subtract (34).

In MIPS assembly language, registers $s0 to $s7 map onto registers 16 to 23, and registers $t0 to $t7 map onto registers 8 to 15.

Hence, $s0 means register 16, $s1 means register 17, $s2 means register 18, . . . , $t0 means register 8, $t1 means register 9 etc.

add $t0, $s1, $s2 →



0	17	18	8	0	32

```
lw  $t0, 32($s3)

add $t0, $s2, $t0

sw $t0, 48($s3)
```

The assignment statement:

A[300] = h + A[300];

is compiled into:

The Equivalent C-statement is:

A[12] = h + A[8];

/ <opn.> dtn, src*

```
lw   $t0, 1200($t1)       # Temporary reg $t0 gets A[300]

add $t0, $s2, $t0       # Temporary reg $t0 gets h + A[300]

sw   $t0, 1200($t1)     # Stores h + A[300] back into A[300];
```

/ <opn.> src, dtn*

op	rs	rt	rd	Addr / Shamt	Funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

MIPS Addressing Mode Summary

Multiple forms of addressing are generically called **addressing modes. The MIPS** addressing modes are the following:

1. **Register addressing, where the operand is a register**

2. **Base or displacement addressing, where the operand is at the memory location,** whose address is the sum of a register and a constant in the instruction

3. **Immediate addressing, where the operand is a constant within the instruction** itself

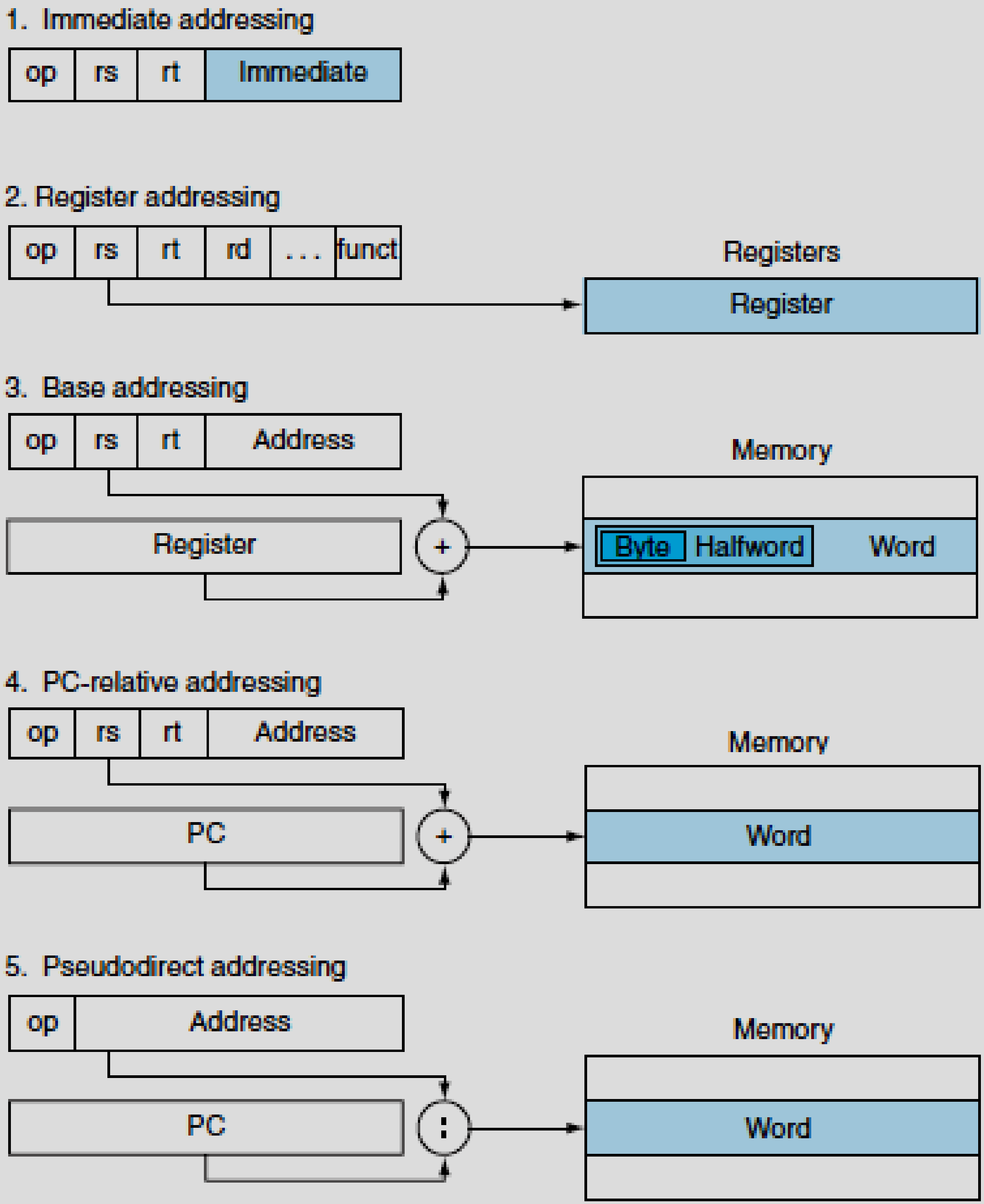
4. **PC-relative addressing, where the address is the sum of the PC and a constant** in the instruction

5. **Pseudodirect addressing, where the jump address is the 26 bits of the** instruction concatenated with the upper bits of the PC

Five MIPS addressing modes.

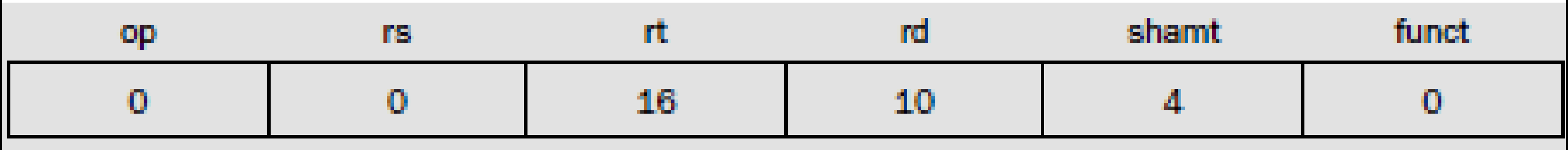
The operands are shaded in color. The operand of **mode 3** is in memory, whereas the operand for **mode 2** is a register. Note that versions of load and store access bytes, halfwords, or words. For **mode 1**, the operand is 16 bits of the instruction itself.

Modes 4 and 5 address instructions in memory, with **mode 4** adding a 16-bit address shifted left 2 bits to the PC and **mode 5** concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC.

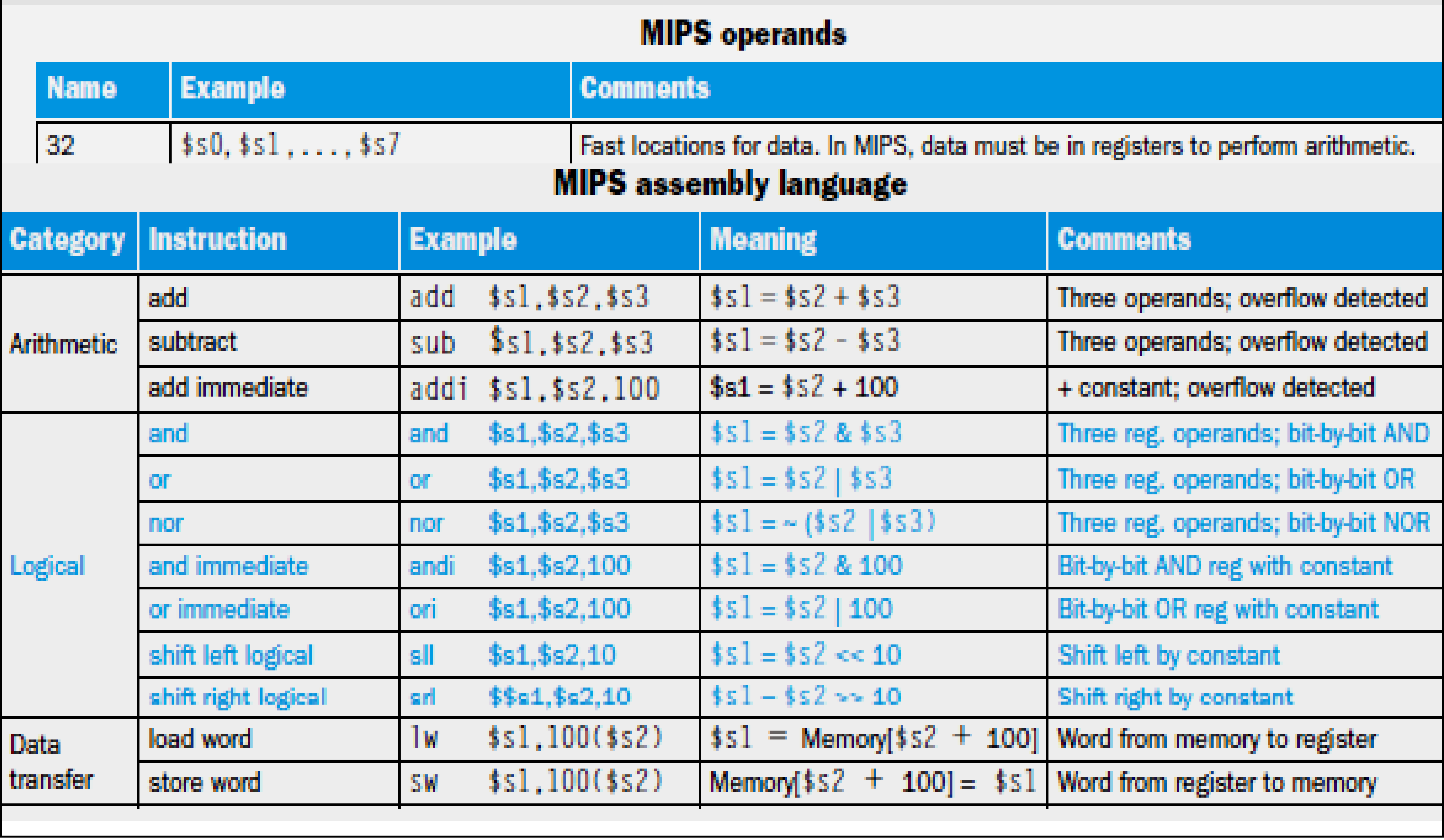


sll / srl =
Shift left / right logical

```
sll  $t2, $s0, 4
#  reg $t2 = reg $s0 << 4 bits
```



op	rs	rt	rd	shamt	funct
0	0	16	10	4	0



MIPS operands

Name	Example	Comments
32	$s0, $s1 , . . . , $s7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	three register operands
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory
Logical	and or	and $s1,$s2,$s3 or $s1,$s2,$s3	$s1 = $s2 & $s3 $s1 = $s2 | $s3	three reg. operands; bit-by-bit AND three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 |$s3)	three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,100	$s1 = $s2 & 100	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,100	$s1 = $s2 | 100	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1,$s2,L	if ($s1 != $s2) go to L	Not equal test and branch

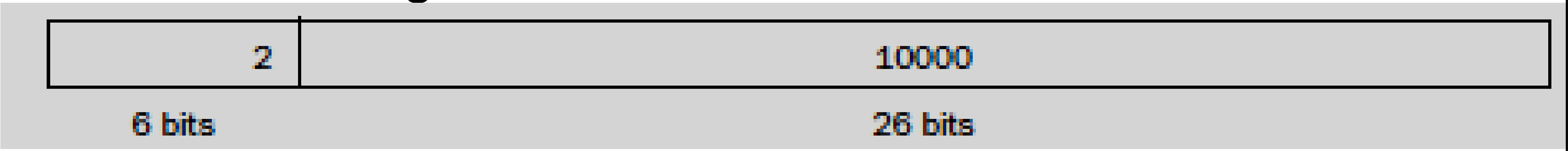
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1;
			else $s1 = 0
				Compare less than; used with beq, bne

	set on less than immediate	slt $s1,$s2,100	if ($s2 < 100) $s1 = 1;
			else $s1 = 0
				Compare less than immediate; used with beq, bne

Unconditional jump	jump	j L	go to L	Jump to target address
	jump register	jr $ra	go to $ra	For procedure return
	jump and link	jal L	$ra = PC + 4; go to L	For procedure call

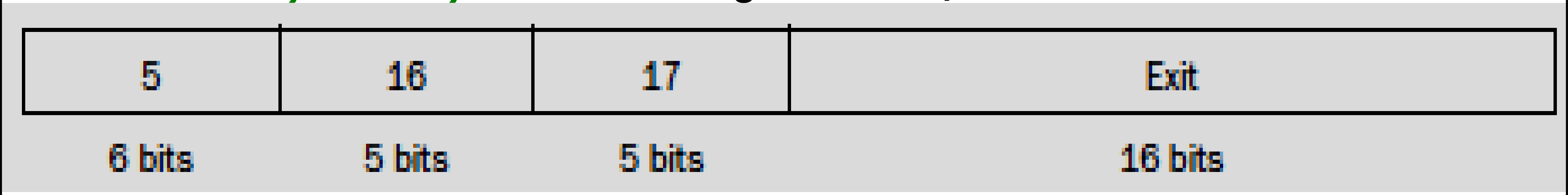
Addressing in Branches and Jumps

j 10000 # go to location 10000



2	10000
6 bits	26 bits

bne $s0, $s1, Exit # go to Exit, if $s0 <> $s1



5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

A branch instruction would calculate the following:
Program counter = Register + Branch address;

This sum allows the program to be as large as 2^{32} and still be able to use conditional branches, solving the branch address size problem. The question is then, which register?

Since the program counter (PC) contains the address of the current instruction, we can branch within +/-(2^{15}) words of the current instruction if we use the PC as the register to be added to the address. Almost all loops and *if statements are much smaller* than 2^{16} words, so the PC is the ideal choice. This form of branch addressing is called PC-relative addressing.

Addressing in Branches and Jumps

Hence, the MIPS address is actually relative to the address of the following instruction (PC + 4) as opposed to the current instruction (PC).

Like most recent computers, MIPS uses PC-relative addressing for all conditional branches because the destination of these instructions is likely to be close to the branch.

On the other hand, jump-and-link instructions invoke procedures that have no reason to be near the call, and so they normally use other forms of addressing. Hence, the MIPS architecture offers long addresses for procedure calls by using the J-type format for both jump and jump-and-link instructions.

Since all MIPS instructions are 4 bytes long, MIPS stretches the distance of the branch by having PC-relative addressing refer to the number of *words to the next* instruction *instead of the number of bytes*. Thus, the 16-bit field can branch four times as far by interpreting the field as a relative word address rather than as a relative byte address.

Similarly, the 26-bit field in jump instructions is also a word address, meaning that it represents a 28-bit byte address.
Since the PC is 32 bits, 4 bits must come from somewhere else. The *MIPS jump instruction replaces only the lower 28 bits of the PC, leaving the upper 4 bits of the PC unchanged.*

```
while (save[i] == k)   /* C-code
      i += 1;
```

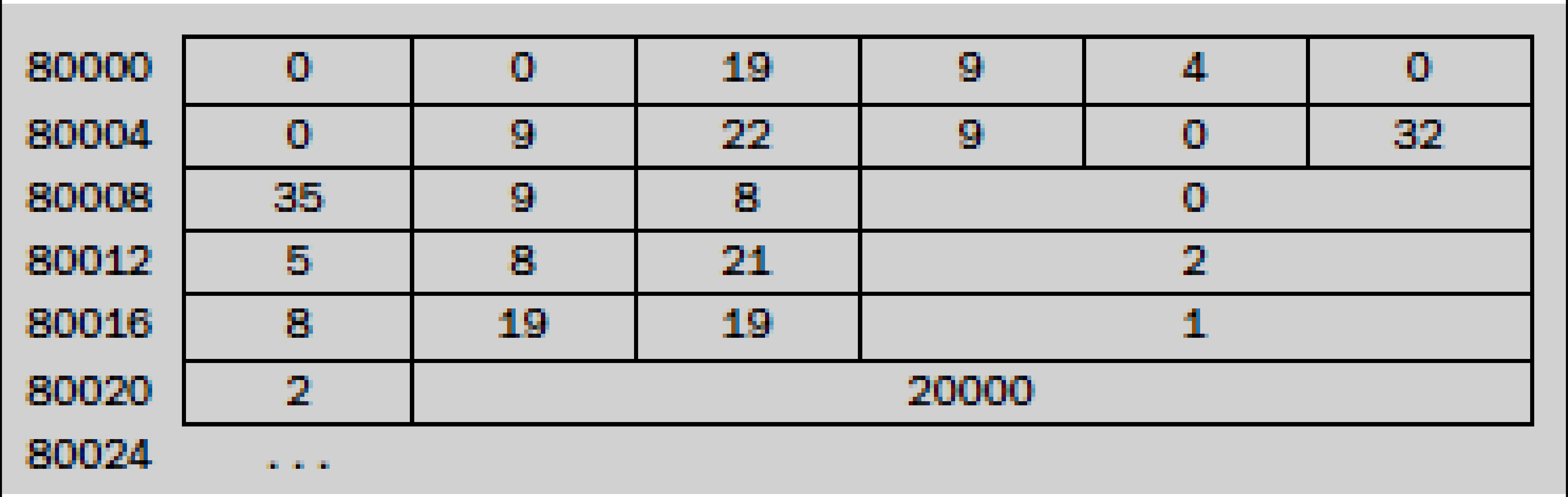
Assume that i and k correspond to registers $s3 and $s5 and the base of the array save is in $s6.

The *while loop above, compiled into this MIPS assembler code:*

```
Loop:   sll $t1, $s3, 2            # Temp reg $t1 = 4 * i
        add $t1, $t1, $s6         # $t1 = address of save[i]
        lw $t0,  0($t1)            # Temp reg $t0 = save[i]
        bne $t0, $s5, Exit        # go to Exit if save[i] <> k
         addi $s3, $s3, 1            # i = i + 1
         j Loop                     # go to Loop
Exit:
```

If we assume we place the loop starting at location 80000 in memory, what is the MIPS machine code for this loop?

```
Loop:   sll $t1, $s3, 2              # Temp reg $t1 = 4 * i
        add $t1, $t1, $s6           # $t1 = address of save[i]
        lw $t0,  0($t1)             # Temp reg $t0 = save[i]
        bne $t0, $s5, Exit          # go to Exit if save[i] < > k
         addi $s3, $s3, 1             # i = i + 1
         j Loop                      # go to Loop
Exit:
```



80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

beq $s0, $s1, L1;

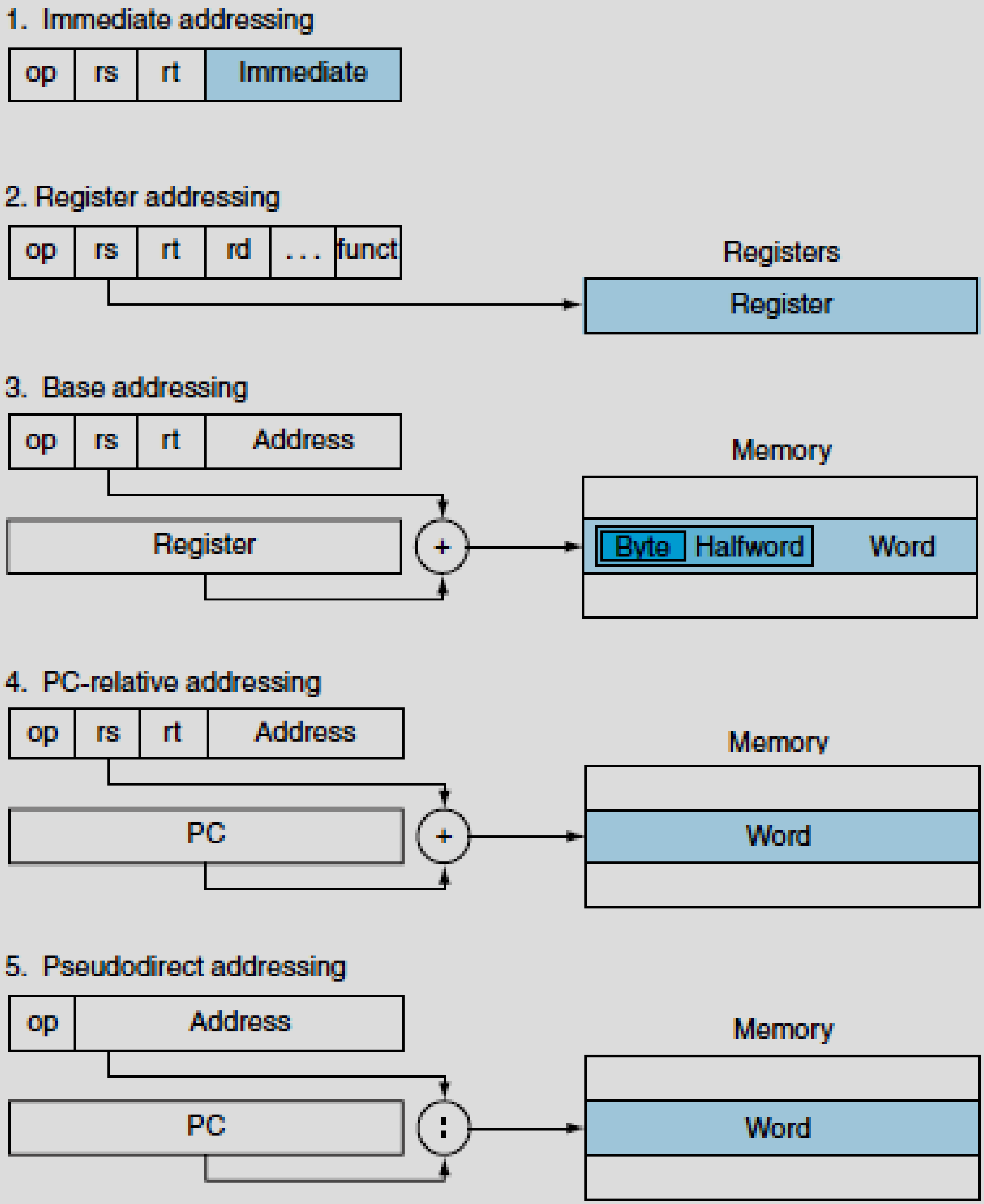
replace by a pair of instructions that offers a much greater branching distance

```
                bne $s0,  $s1, L2
                j       L1
          L2:
```

Five MIPS addressing modes.

The operands are shaded in color. The operand of **mode 3** is in memory, whereas the operand for **mode 2** is a register. Note that versions of load and store access bytes, halfwords, or words.

For **mode 1**, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with **mode 4** adding a 16-bit address shifted left 2 bits to the PC and **mode 5** concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC.



Features in a multi-cycle implementation of a MIPS processor :

- A single memory unit is used for both instructions and data.

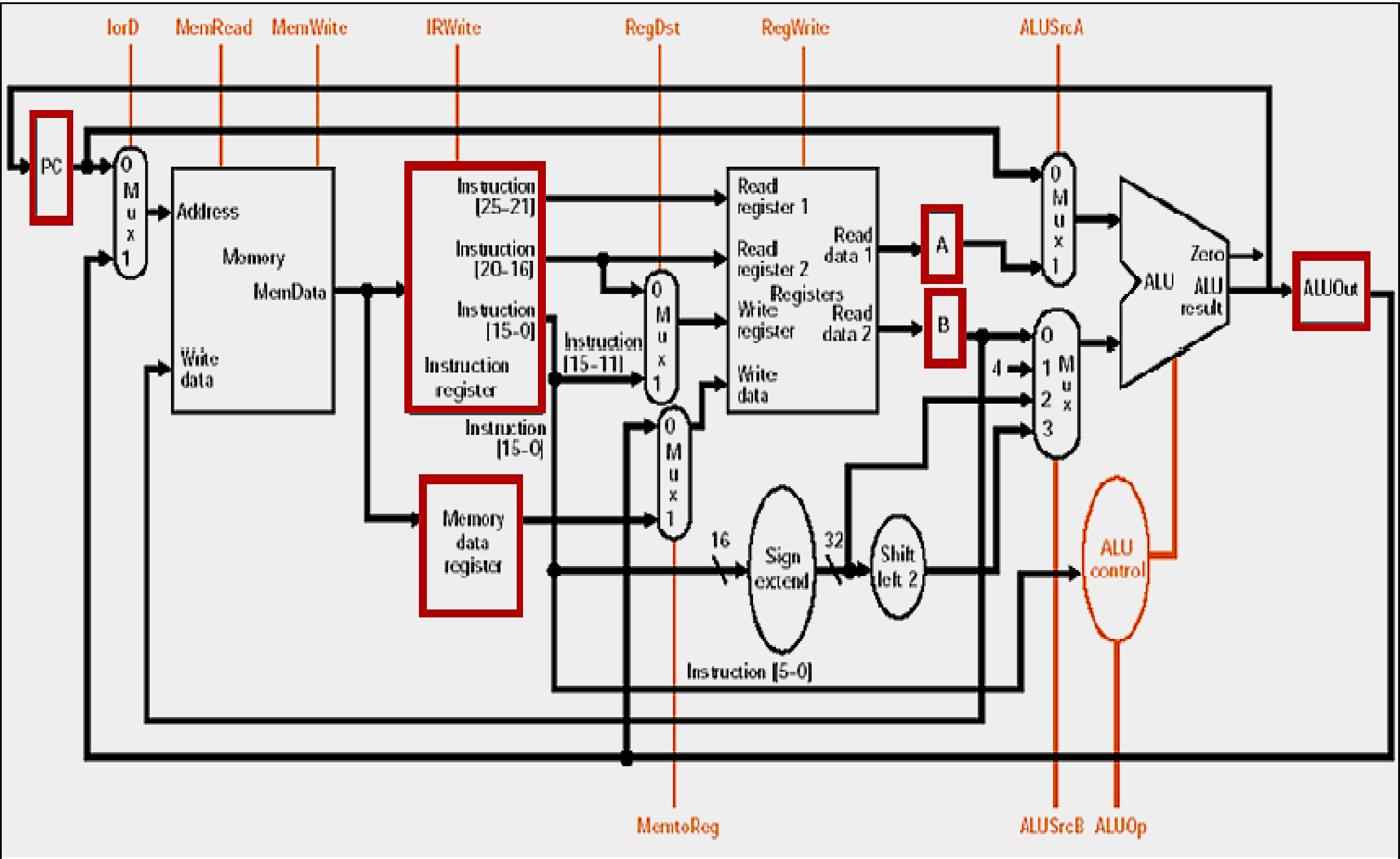
- There is a single ALU, rather than an ALU and two adders.

- One or more registers are added after every major functional unit to hold the output of that unit until the value is used in a subsequent clock cycle.

In this multi-cycle design, we assume that the clock cycle can accommodate at most one of the following operations: a memory access, a register file access (two reads or one write), or an ALU operation.

Hence, any data produced by one of these three functional units (the memory, the register file, or the ALU) must be saved, into a temporary register for use on a later cycle.

If it were not saved then the possibility of a timing race could occur, leading to the use of an incorrect value.



Source - **D. A. Patterson and J. L. Hennessy,**
major functional blocks of the CPU, for **multi-cycle datapath**
implementation of the ISA instructions (MIPS)

The following temporary registers are added to meet these requirements:

- The Instruction register (IR) and the Memory data register (MDR) are added to save the output of the memory for an instruction read and a data read, respectively. Two separate registers are used, since, both values are needed during the same clock cycle.

- The A and B registers are used to hold the register operand values read from the register file.

- The ALUOut register holds the output of the ALU.

Because several functional units are shared for different purposes, use multiplexors. For example, since one memory is used for both instructions and data, we need a multiplexor to select between the two sources for a memory address, namely, the PC (for instruction access) and ALUOut (for data access).

A single ALU must accommodate all the inputs. Major parts of the datapath now consists of:

1. An additional multiplexor is added for the first ALU input. The multiplexor chooses between the A register and the PC.

2. The multiplexor on the second ALU input is a 2-4 way multiplexor. The two additional inputs to the multiplexor are the constant 4 (used to increment the PC) and the sign-extended and shifted offset field (used in the branch address computation).

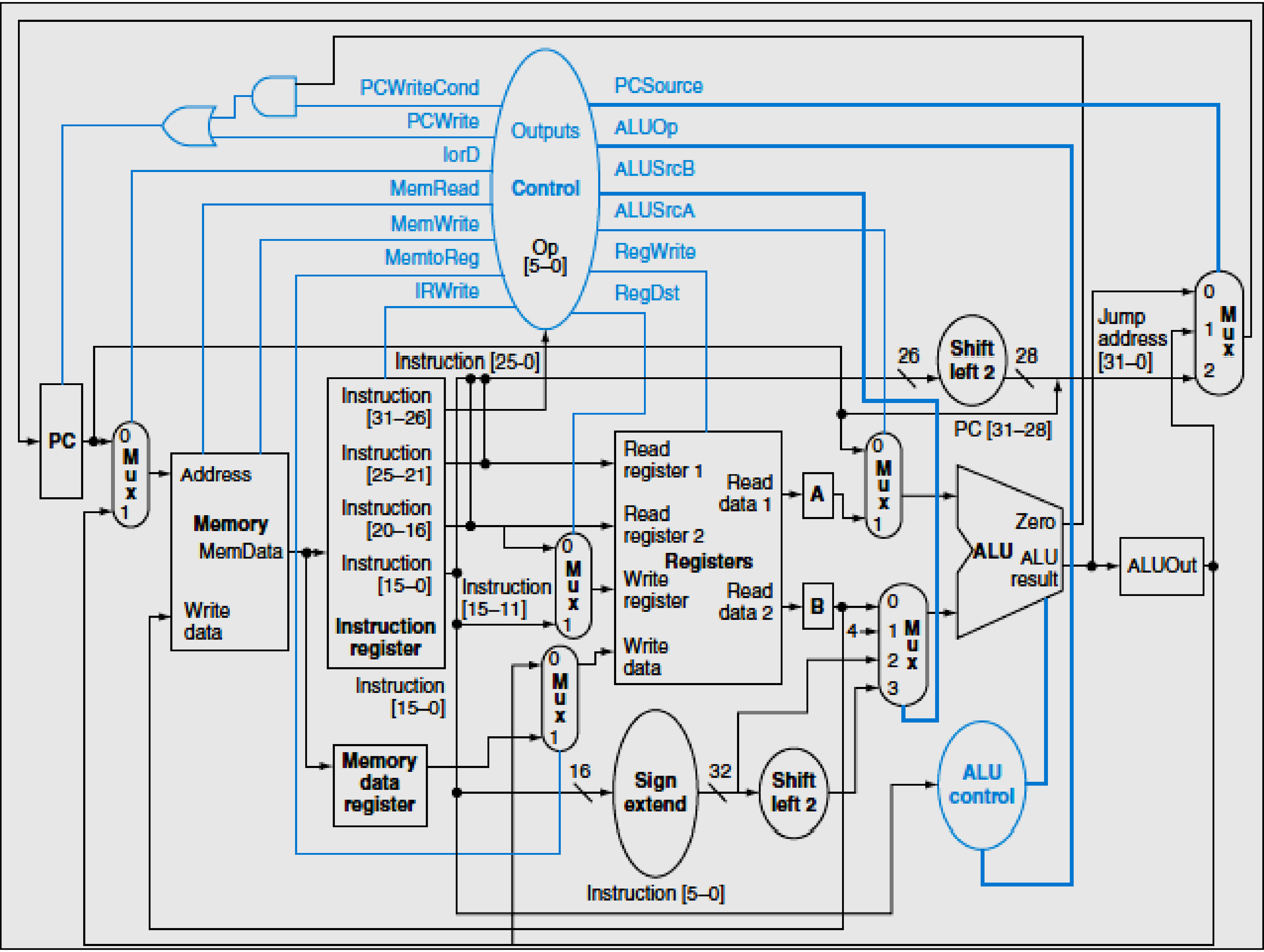
With the jump instruction and branch instruction, there are three possible sources for the value to be written into the PC:

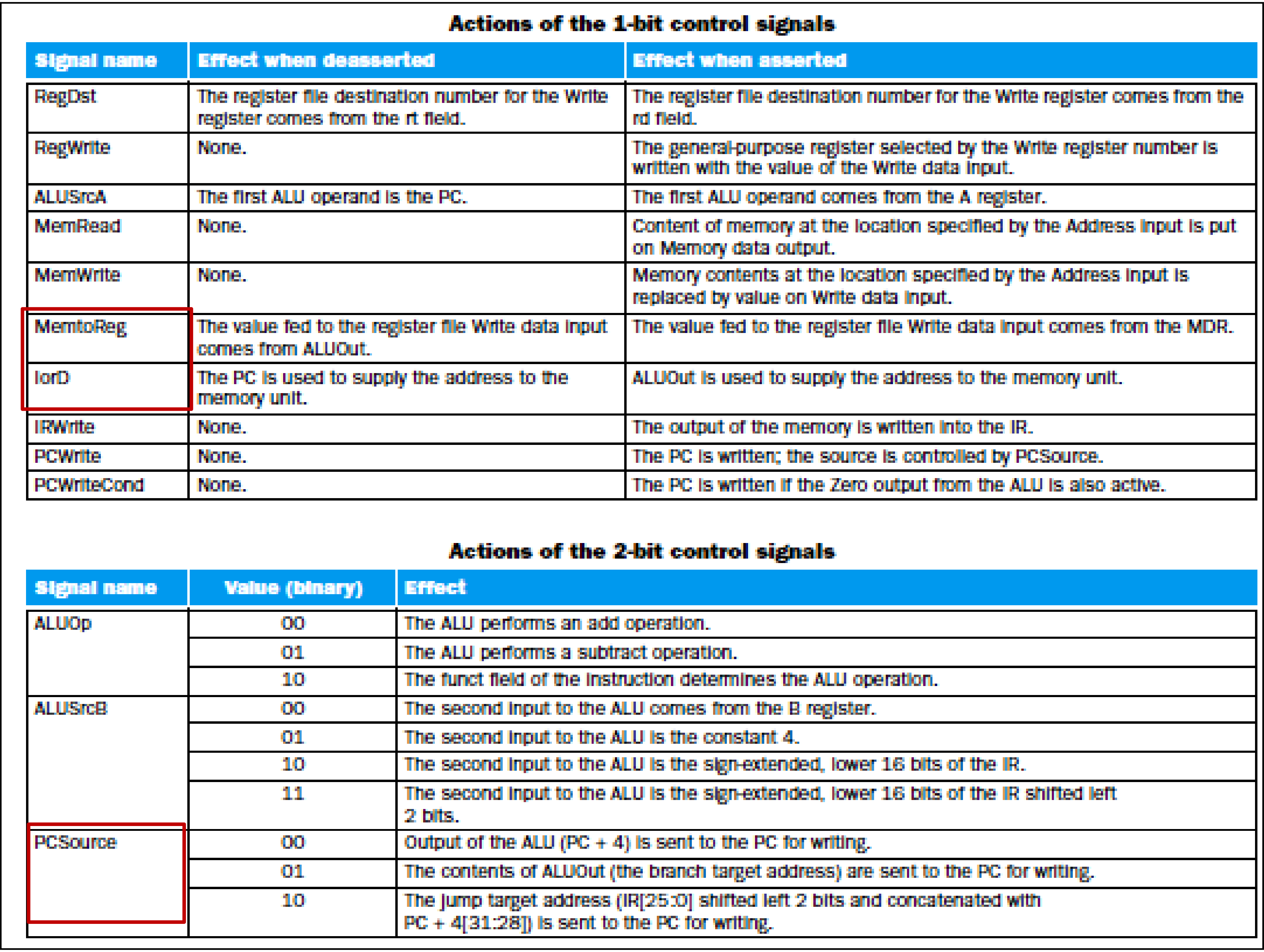
1. The output of the ALU, which is the value PC + 4 during instruction fetch. This value should be stored directly into the PC.

2. The register ALUOut, which is where we will store the address of the branch target after it is computed.

3. The lower 26 bits of the Instruction register (IR) shifted left by two and concatenated with the upper 4 bits of the incremented PC, which is the source when the instruction is a jump.

The PC is written both unconditionally and conditionally. During a normal increment and for jumps, the PC is written unconditionally. If the instruction is a conditional branch, the incremented PC is replaced with the value in ALUOut only if the two designated registers are equal.


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]
0
1
2
M
u
x
PC [31–28]
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–11]
0
M
u
x
1
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
0
M
u
x
1
Zero
ALU
ALU
result
ALUOut
0
1
2
3
M
u
x
4
Instruction
[15–0]
Memory
data
register
0
M
u
x
1
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5–0]

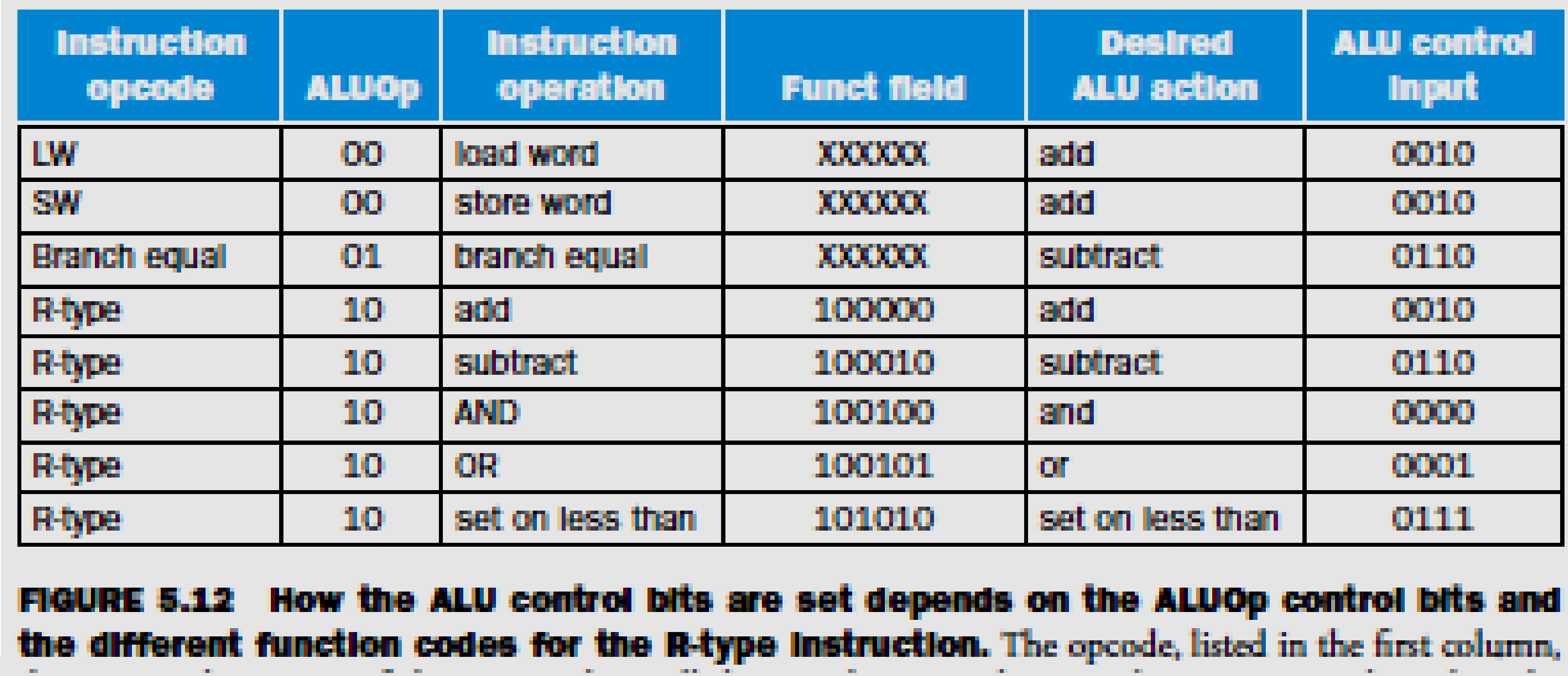


Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSource.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals

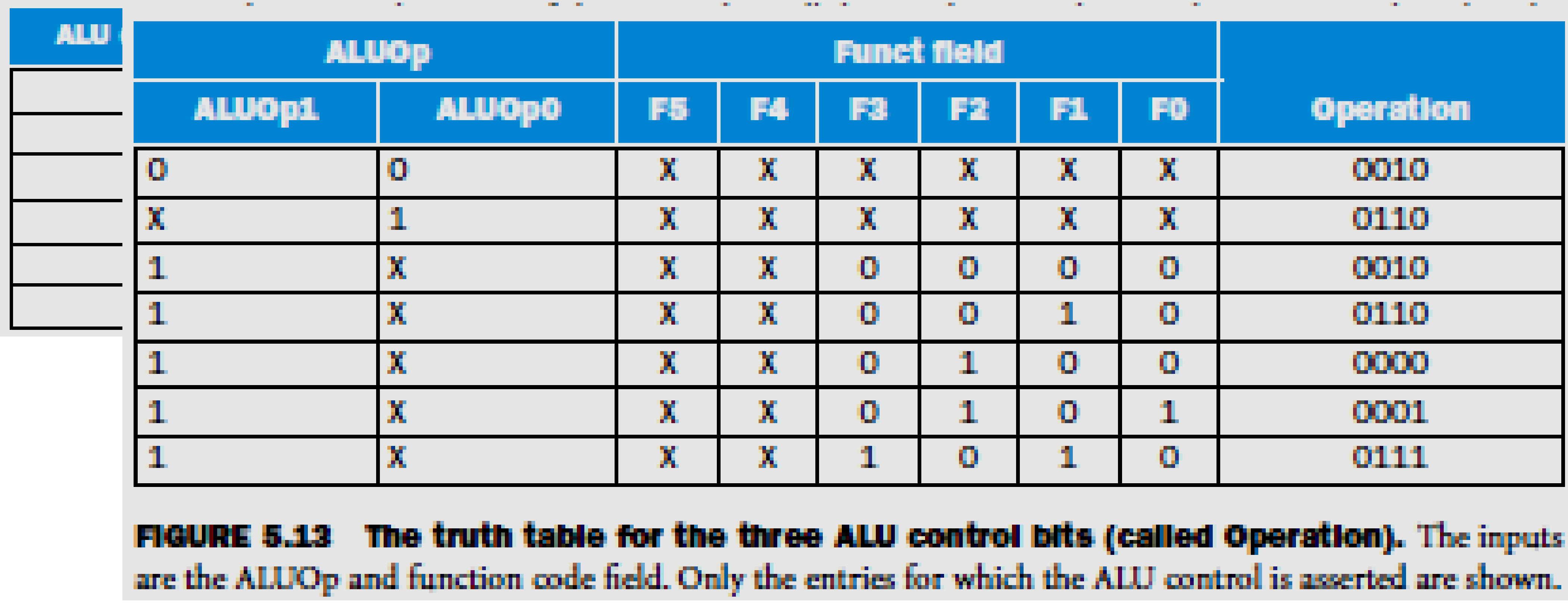
Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC + 4[31:28]) is sent to the PC for writing.



Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

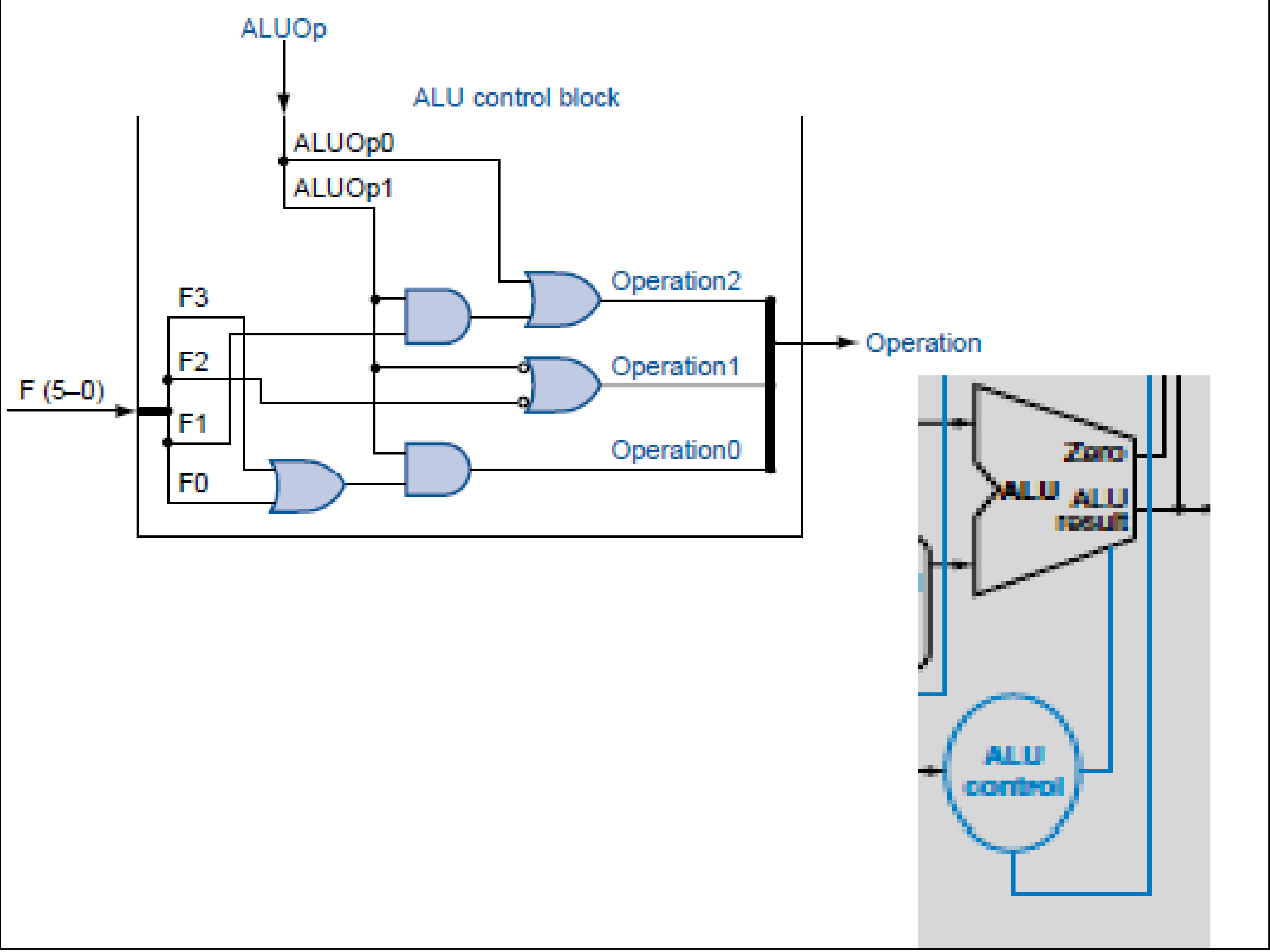
FIGURE 5.12 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column,

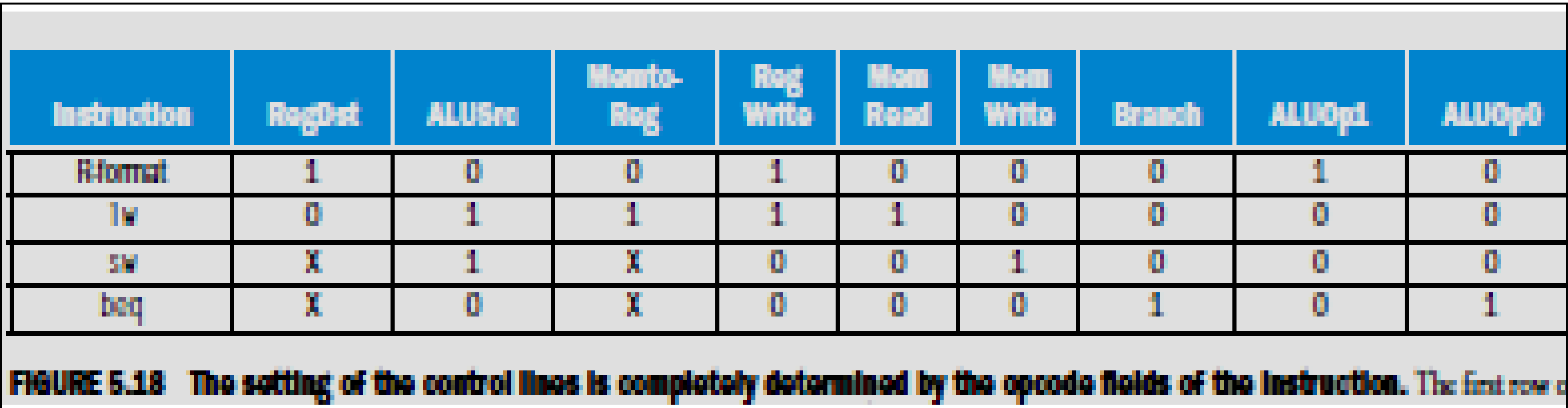
ALU



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

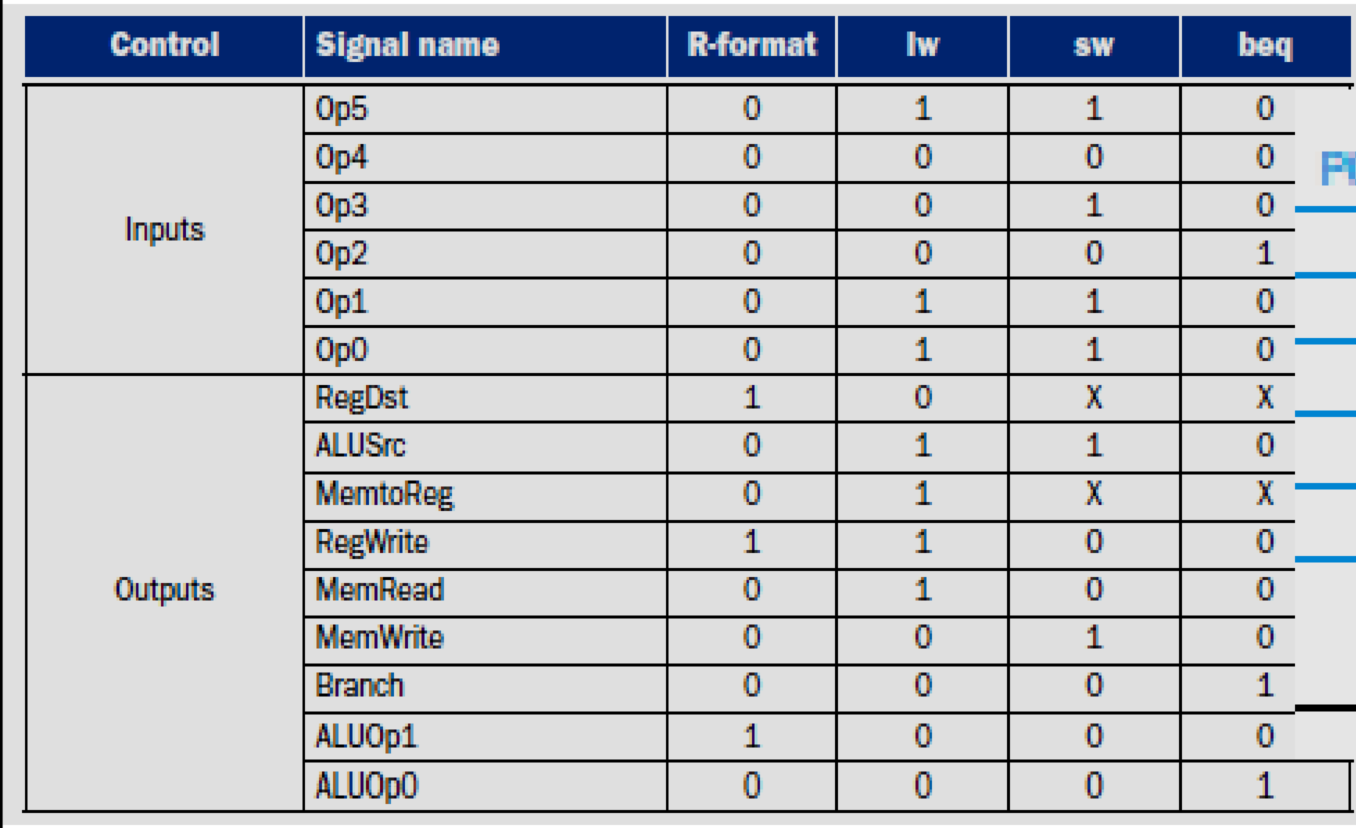
FIGURE 5.13 The truth table for the three ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown.


ALUOp
ALU control block
ALUOp0
ALUOp1
F3
F2
F1
F0
F (5–0)
Operation2
Operation1
Operation0
Operation
Zero
ALU
ALU
result
ALU
control

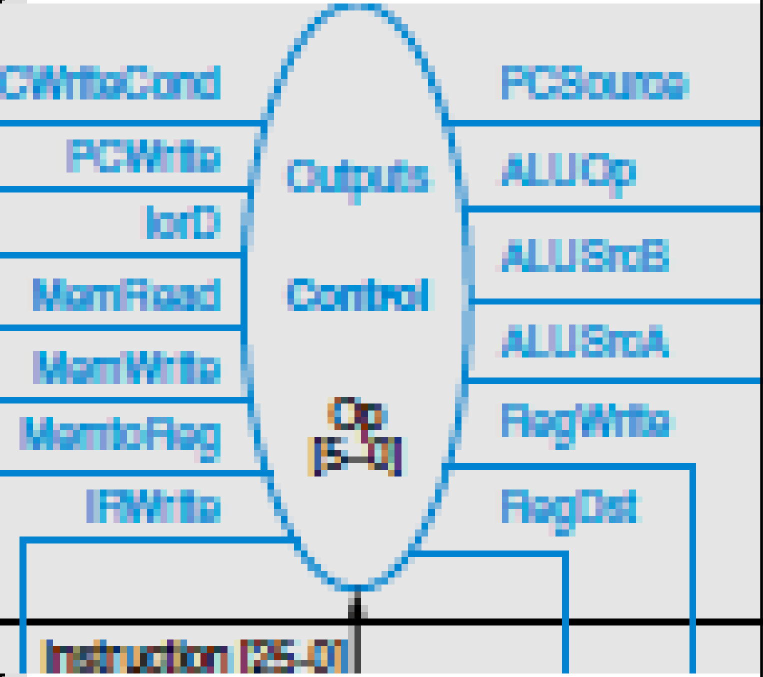


Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

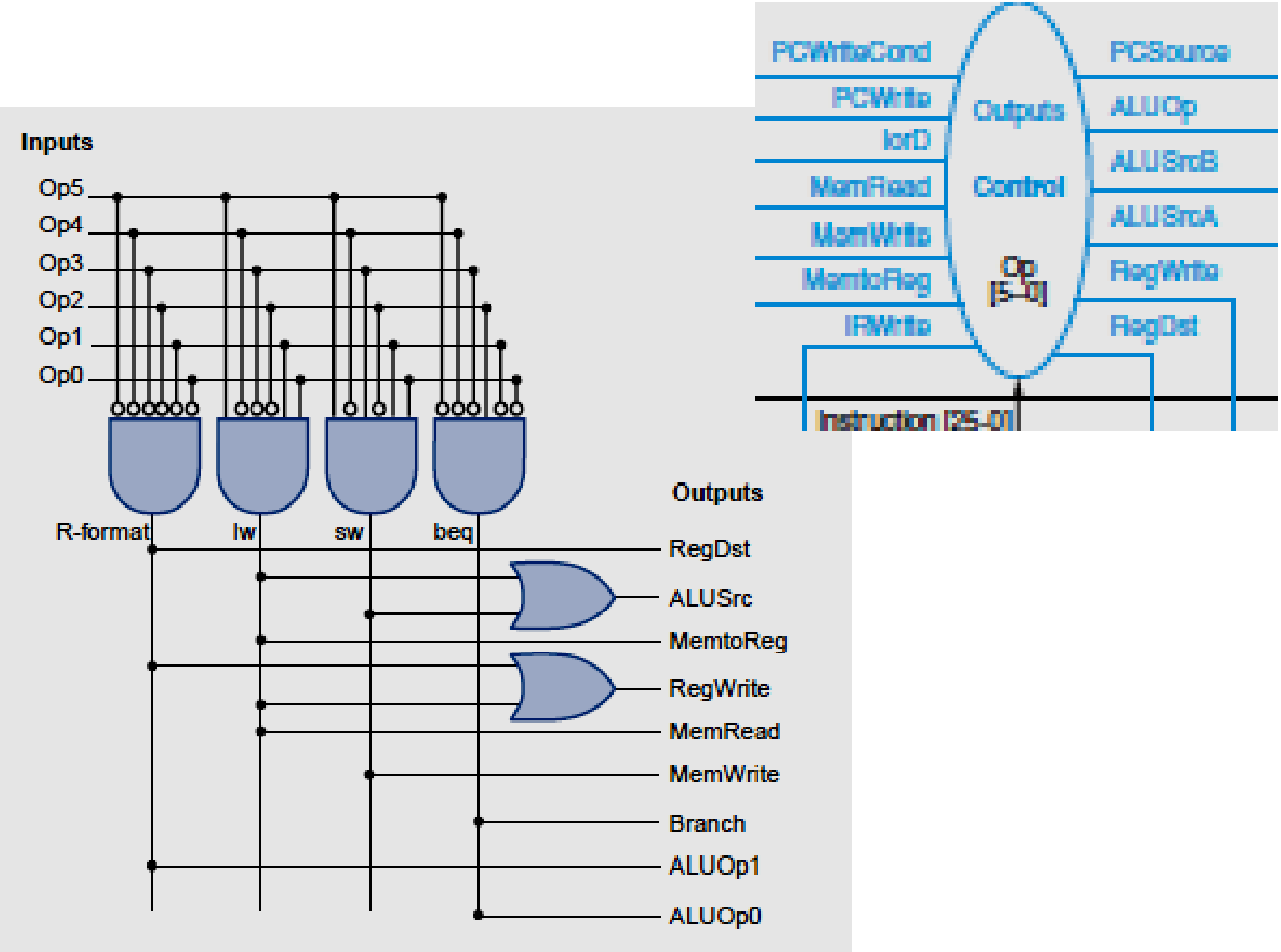
FIGURE 5.18 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row o



Control	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1



The **control function** for a **simple implementation** is completely specified by this truth table.



.2.5 The structured implementation of the control function as described

Each MIPS instruction needs from three to five of these steps:

1. **Instruction fetch step:**

Fetch the instruction from memory and compute the address of the next sequential instruction:

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

2. **Instruction decode and register fetch step**

```
A <= Reg[IR[25:21]];    // Reg. rs in opcode
B <= Reg[IR[20:16]];    //  Reg. rt in opcode
ALUOut <= PC + (sign-extend (IR[15-0]) << 2);
                        // Branch target address
```

3. **Execution, memory address computation, or branch completion**

i) Memory reference:

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

ii) Arithmetic-logical instruction (R-type):

```
ALUOut <= A op B;
```

iii) Branch:

```
if (A == B) PC <= ALUOut;
```

3. Execution, memory address computation, or branch completion

i) Memory reference:

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

ii) Arithmetic-logical instruction (R-type):

```
ALUOut <= A op B;
```

iii) Branch:

```
if (A == B) PC <= ALUOut;
```

iv) Jump:

```
// {x, y} represents concatenation of bit fields x and y
        PC <= {PC [31:28], (IR[25:0]] <<2)};
```

4. Memory access or R-type instruction completion step

Memory reference:

```
MDR <= Memory [ALUOut];  //Load
or
Memory [ALUOut] <= B;   //Store from rt to Mem.
```

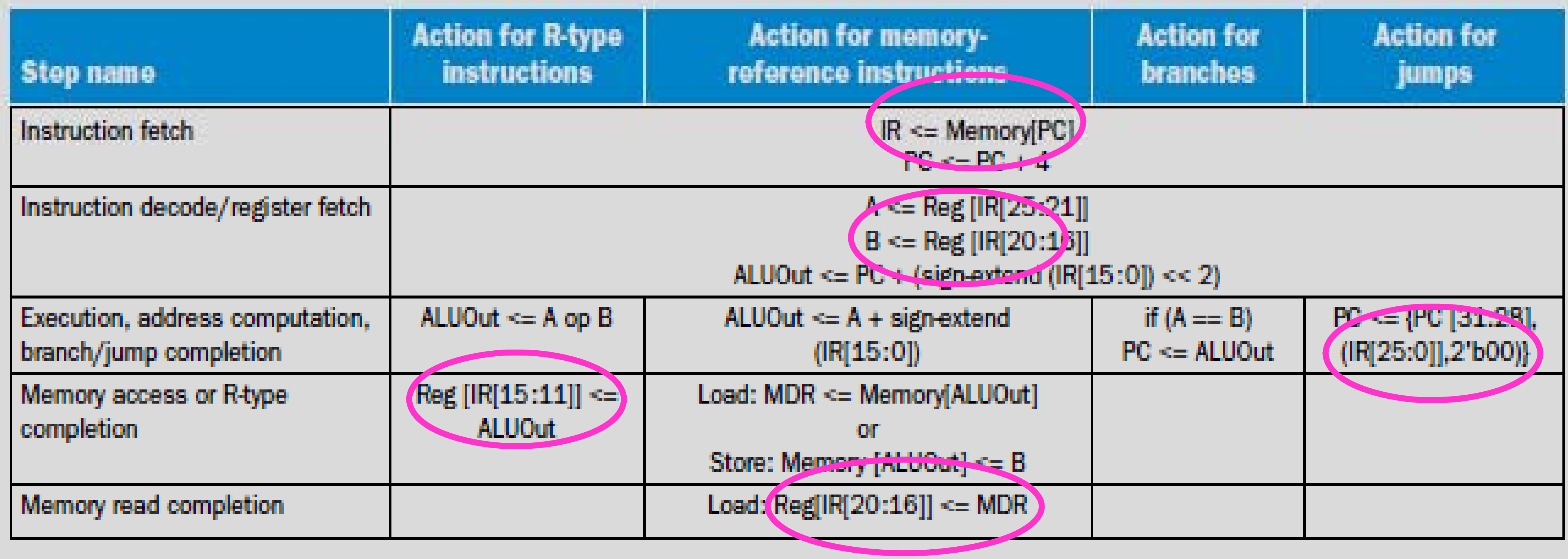
Arithmetic-logical instruction (R-type):

```
Reg[IR[15:11]] <= ALUOut;   // Reg. rd in opcode
```

5. Memory read completion step -

Load:

```
Reg[IR[20:16]] <= MDR;   //  Reg. rt
```

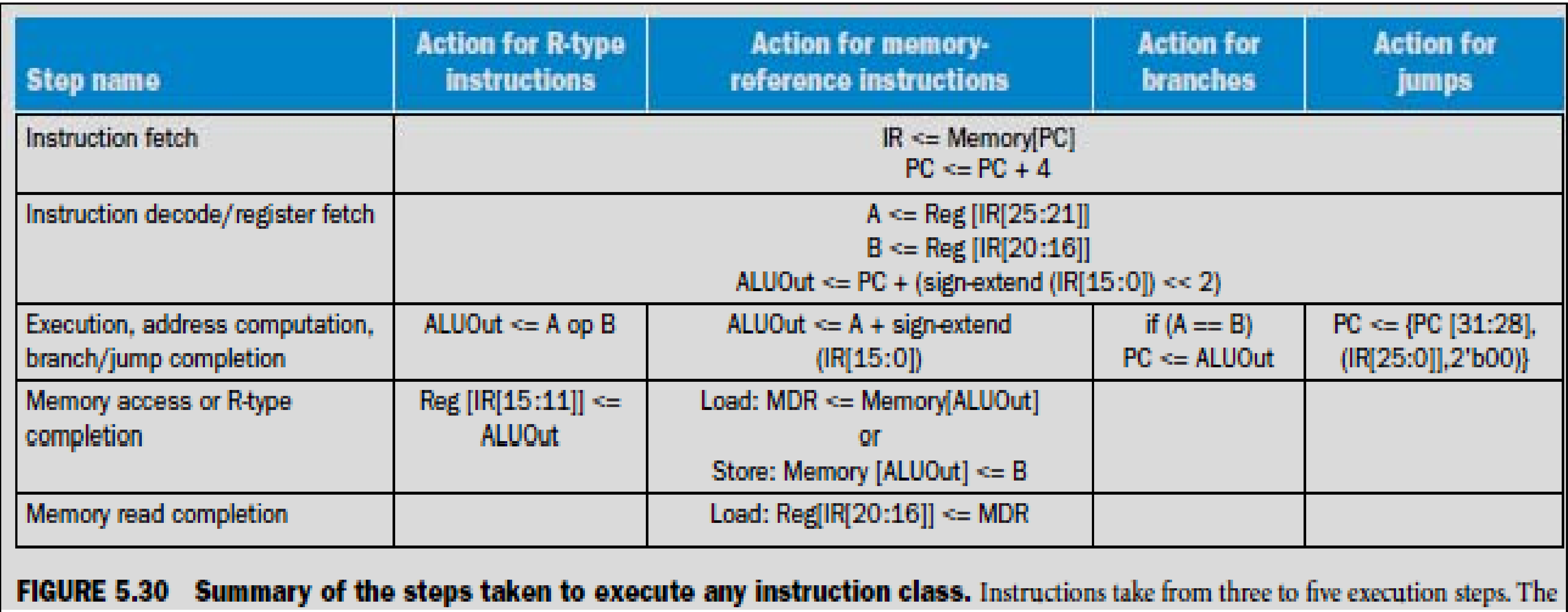


Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, 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'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The **first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class.**

In a multi-cycle implementation, a new instruction will be started as soon as the current instruction completes.

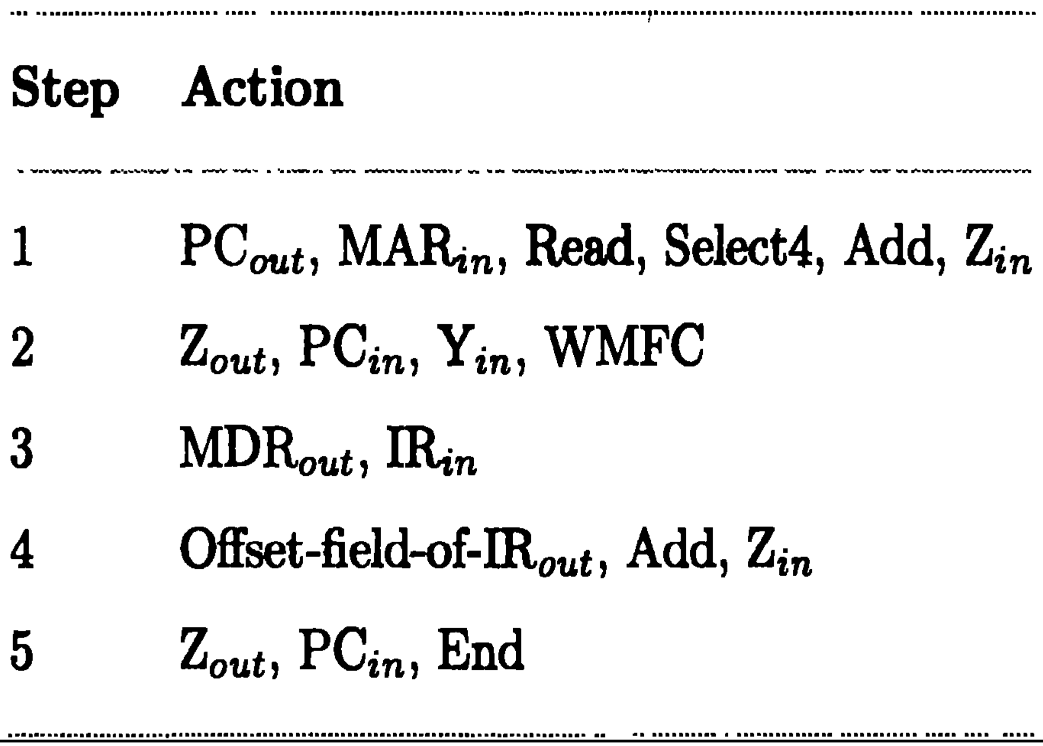
The register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.



Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
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 computation, 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'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

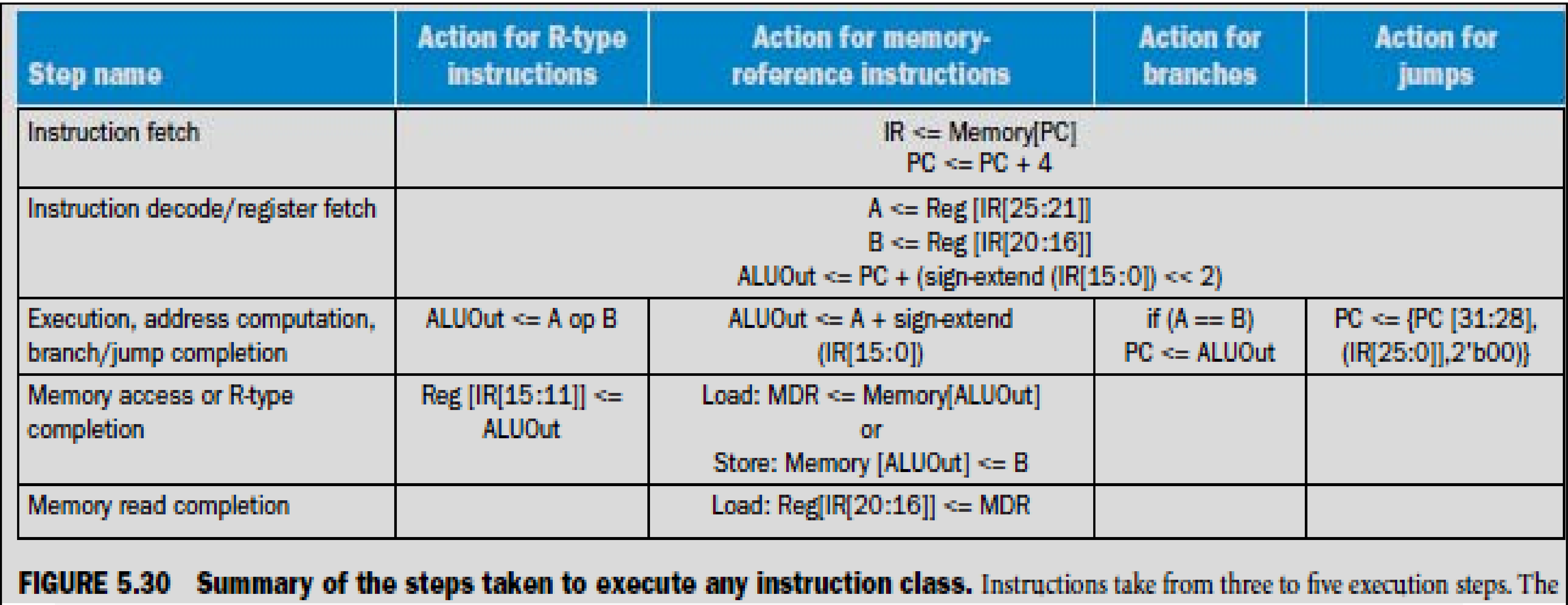
FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The

Remember this ?? →



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	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Now compare the two



Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
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 computation, 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'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The

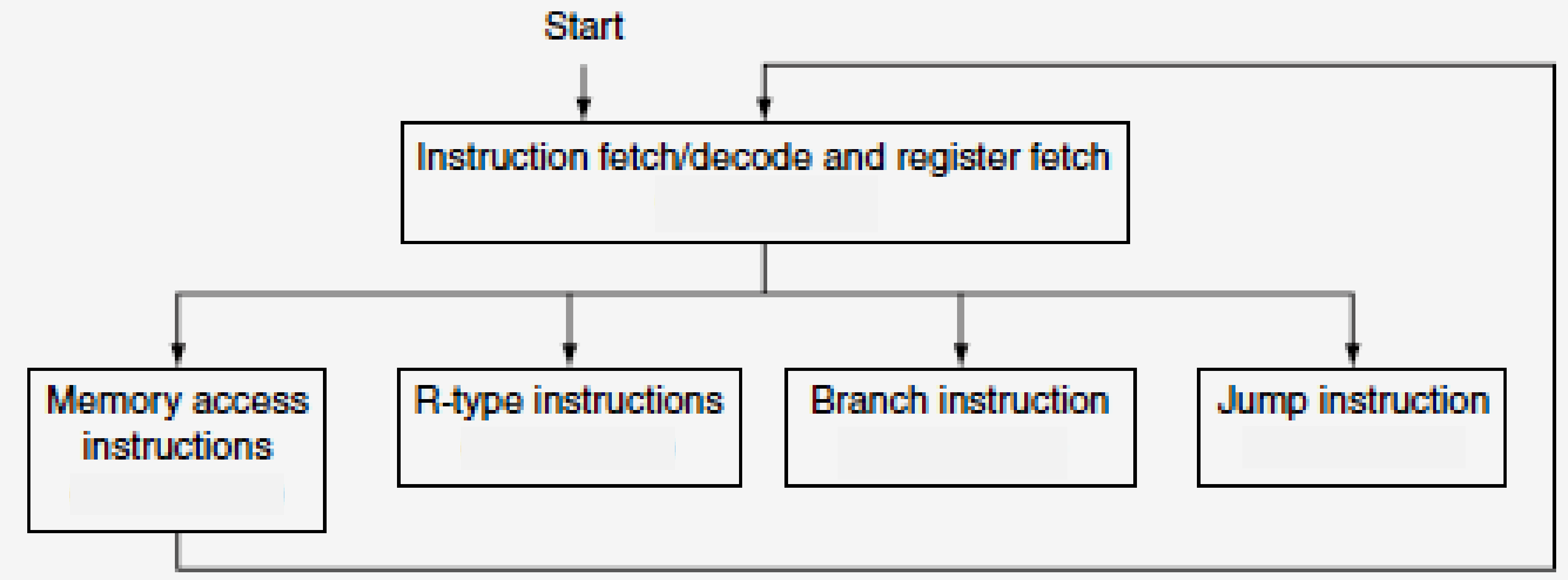

Start
Instruction fetch/decode and register fetch
Memory access instructions
R-type instructions
Branch instruction
Jump instruction

FIGURE 5.31 The high-level view of the finite state machine control. The first steps are inde-

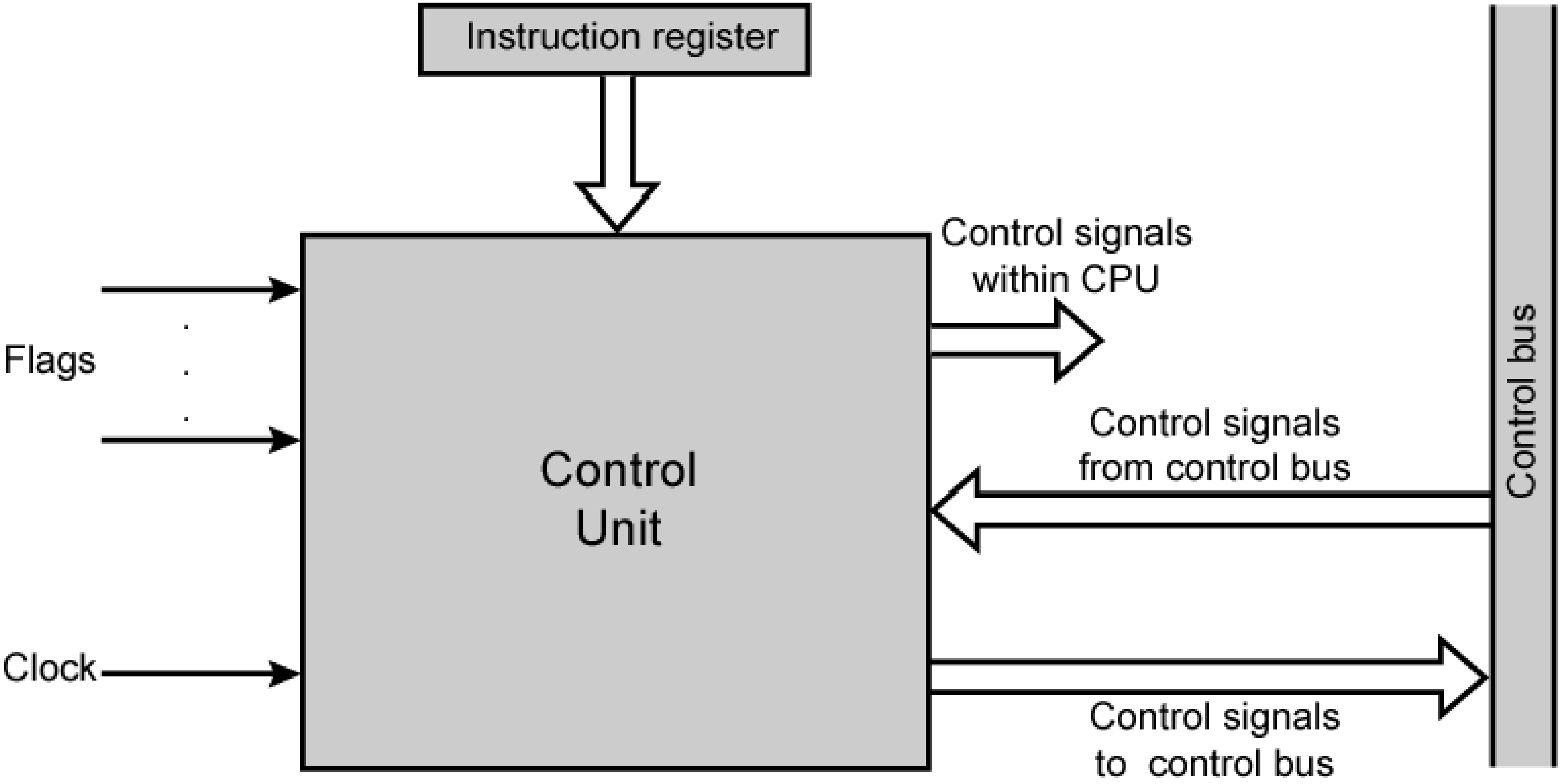
Functions of Control Unit

- 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

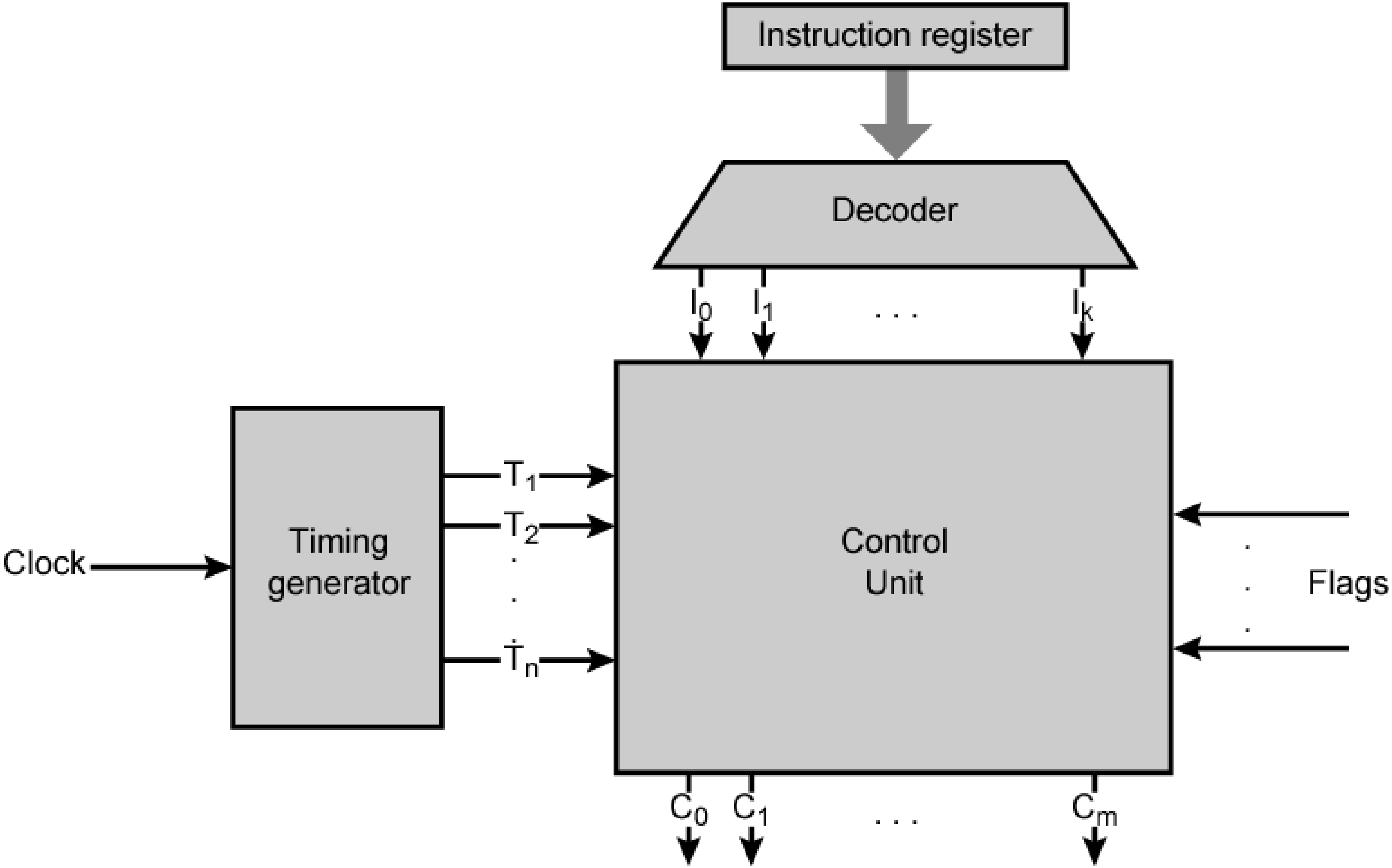

Instruction register
Control signals within CPU
Flags
. . .
Control Unit
Control signals from control bus
Control bus
Clock
Control signals to control bus

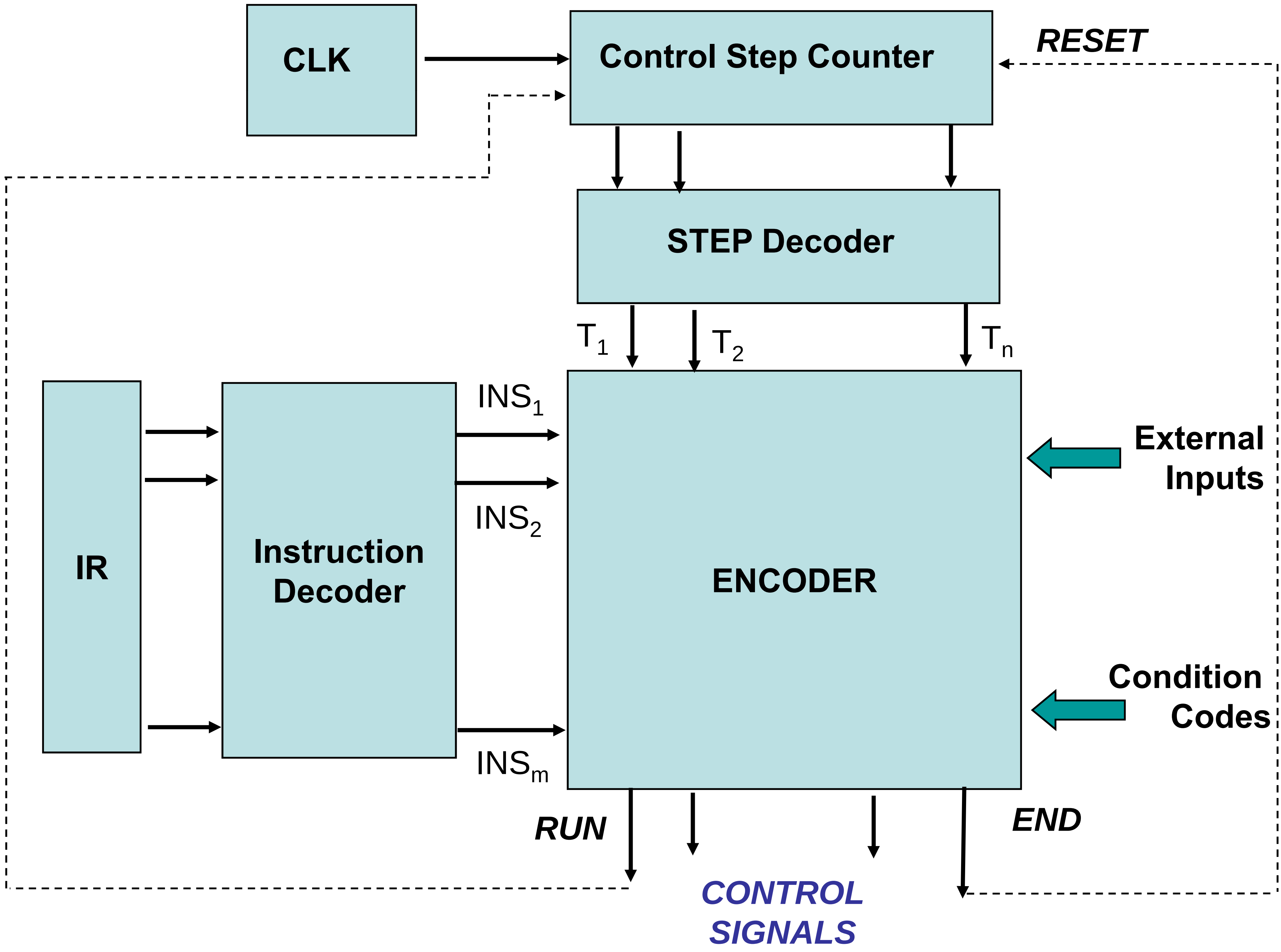
HARDWIRED CONTROL

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

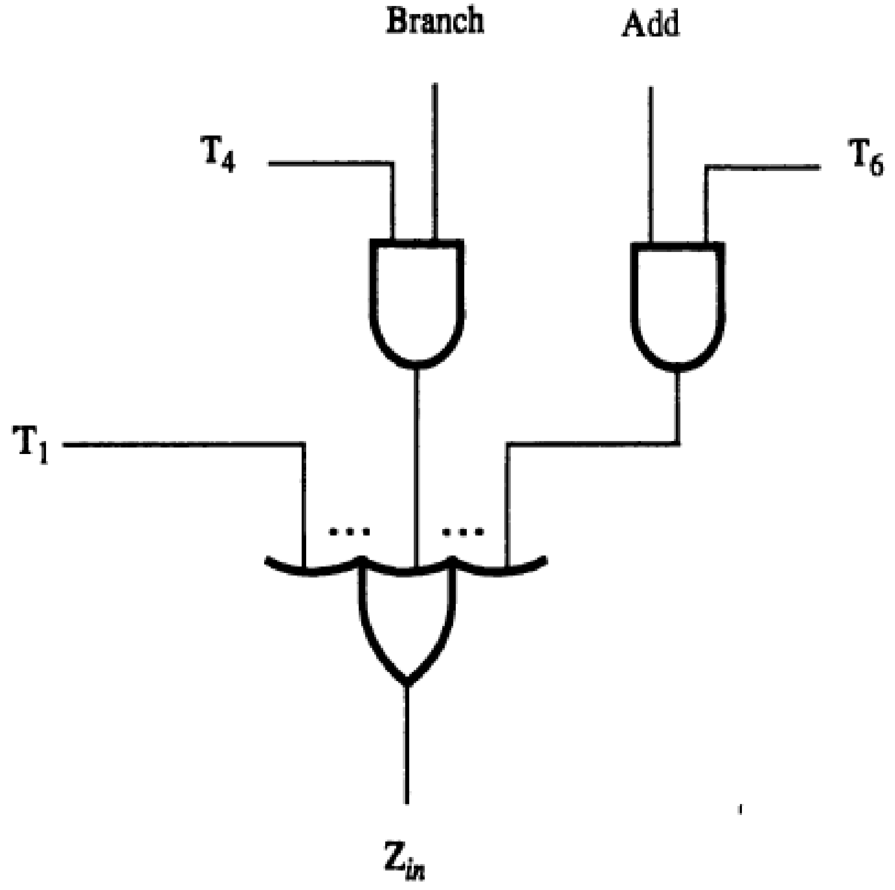
$$Z_{in} = T_1 + T_6.ADD + T_4.BR + ...$$

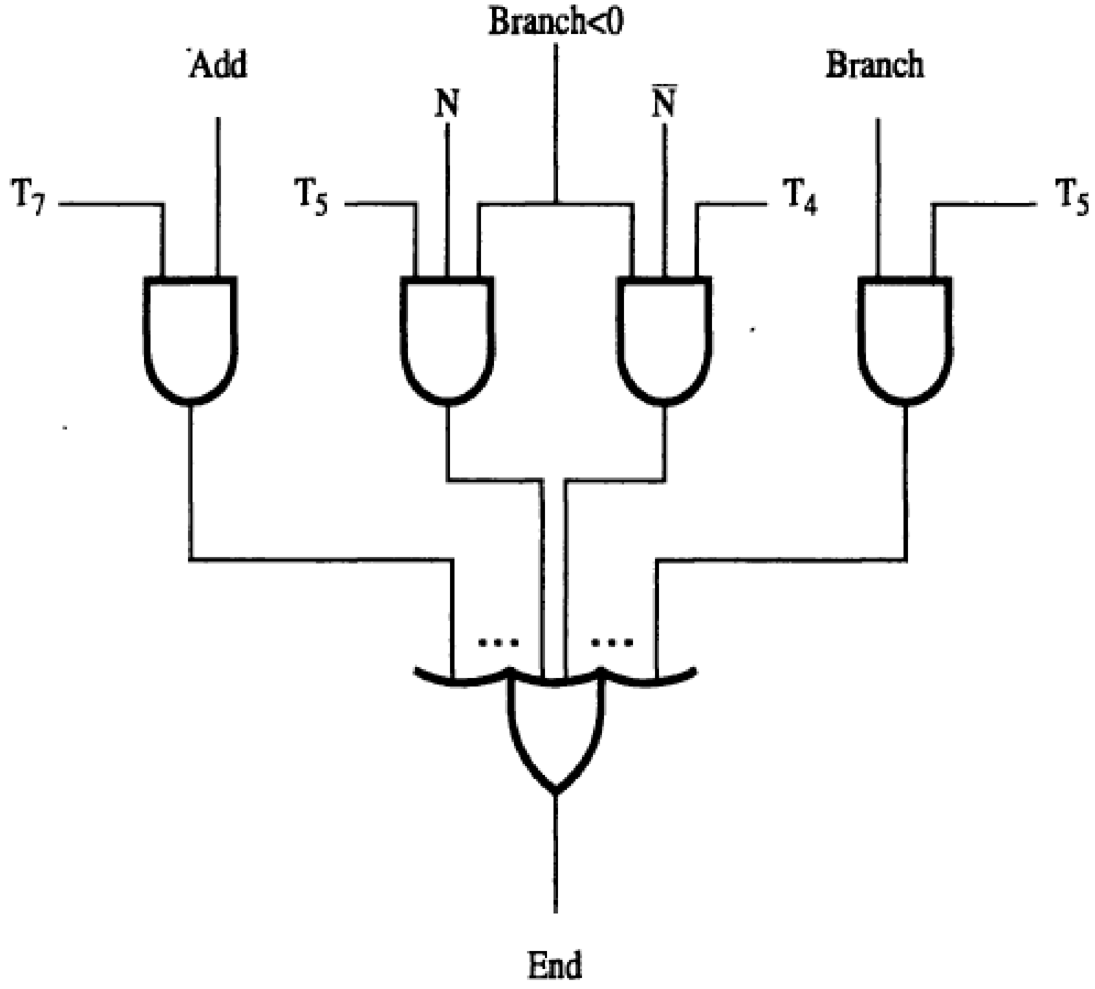
$$END = T_7.ADD + T_5.BR + (T_5.CF + T_4.CF').BRN + ...$$

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

Design logic mostly based on FSM (Finite State machine)

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





$$\boldsymbol{END = T_7.ADD + T_5.BR + (T_5.CF + T_4.CF').BRN +....}$$

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.

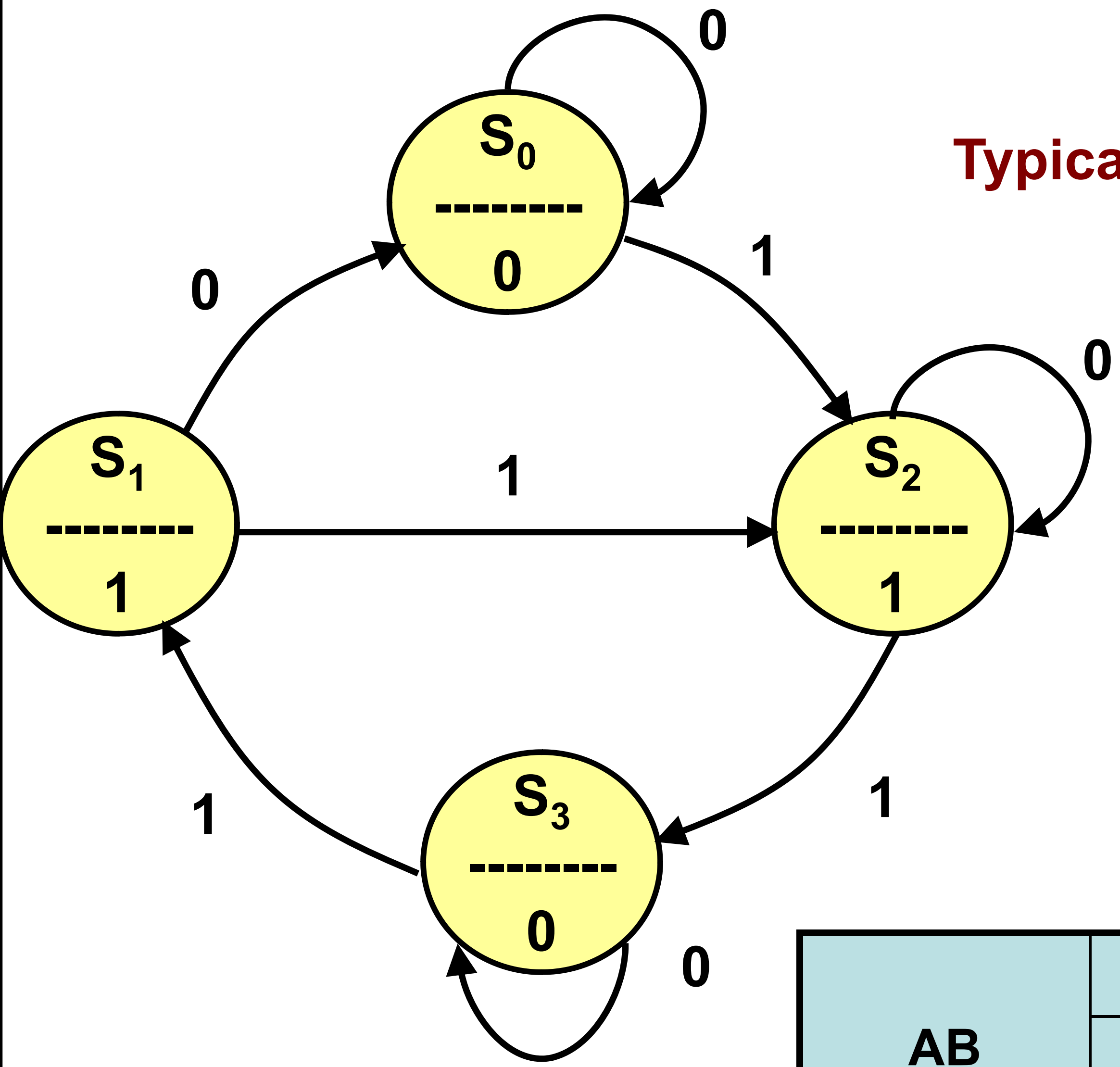
To implement any ISA, the system sequentially changes state from one to another. Control Unit implements the steps.

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).*

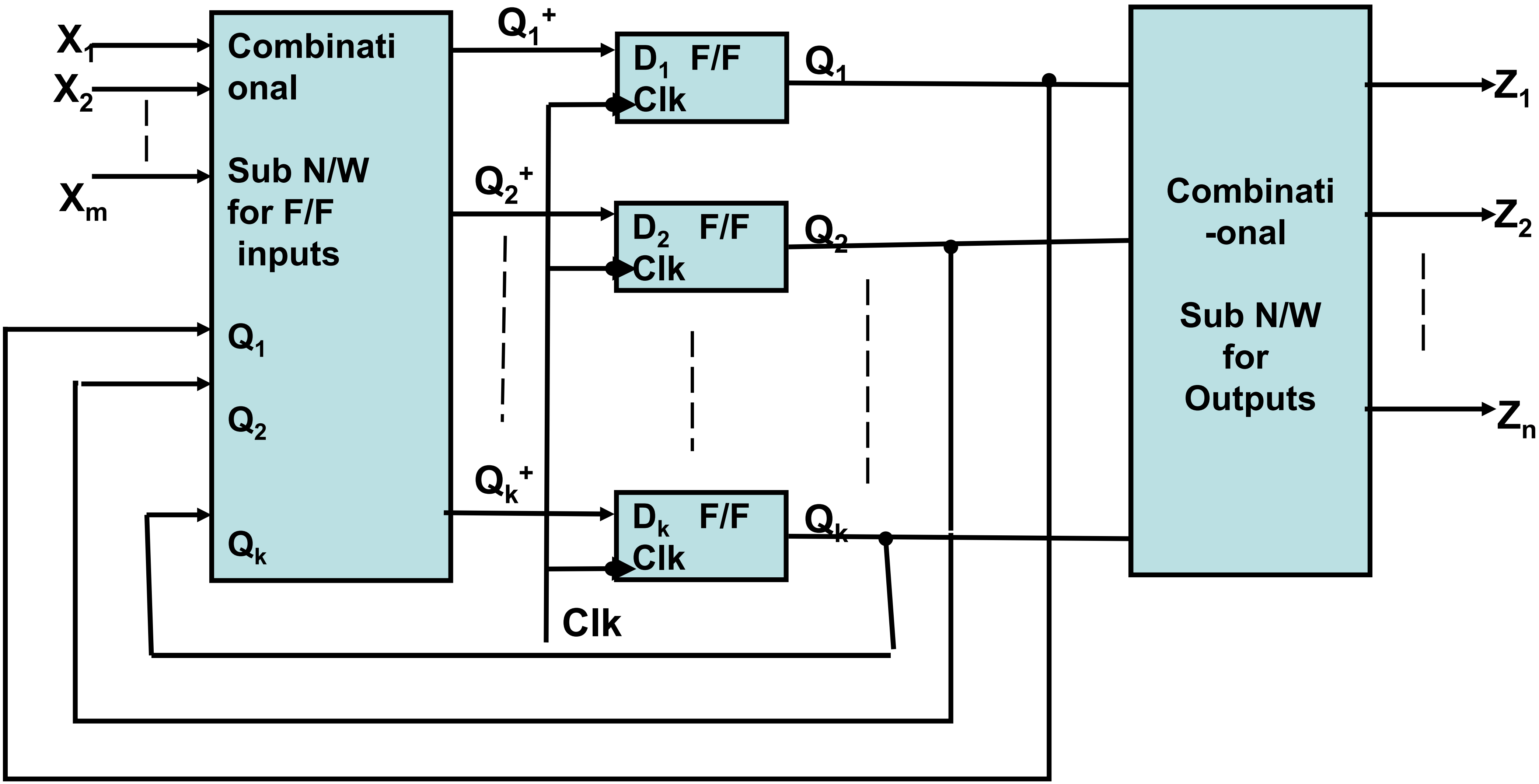
Typical Moore State Graph



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



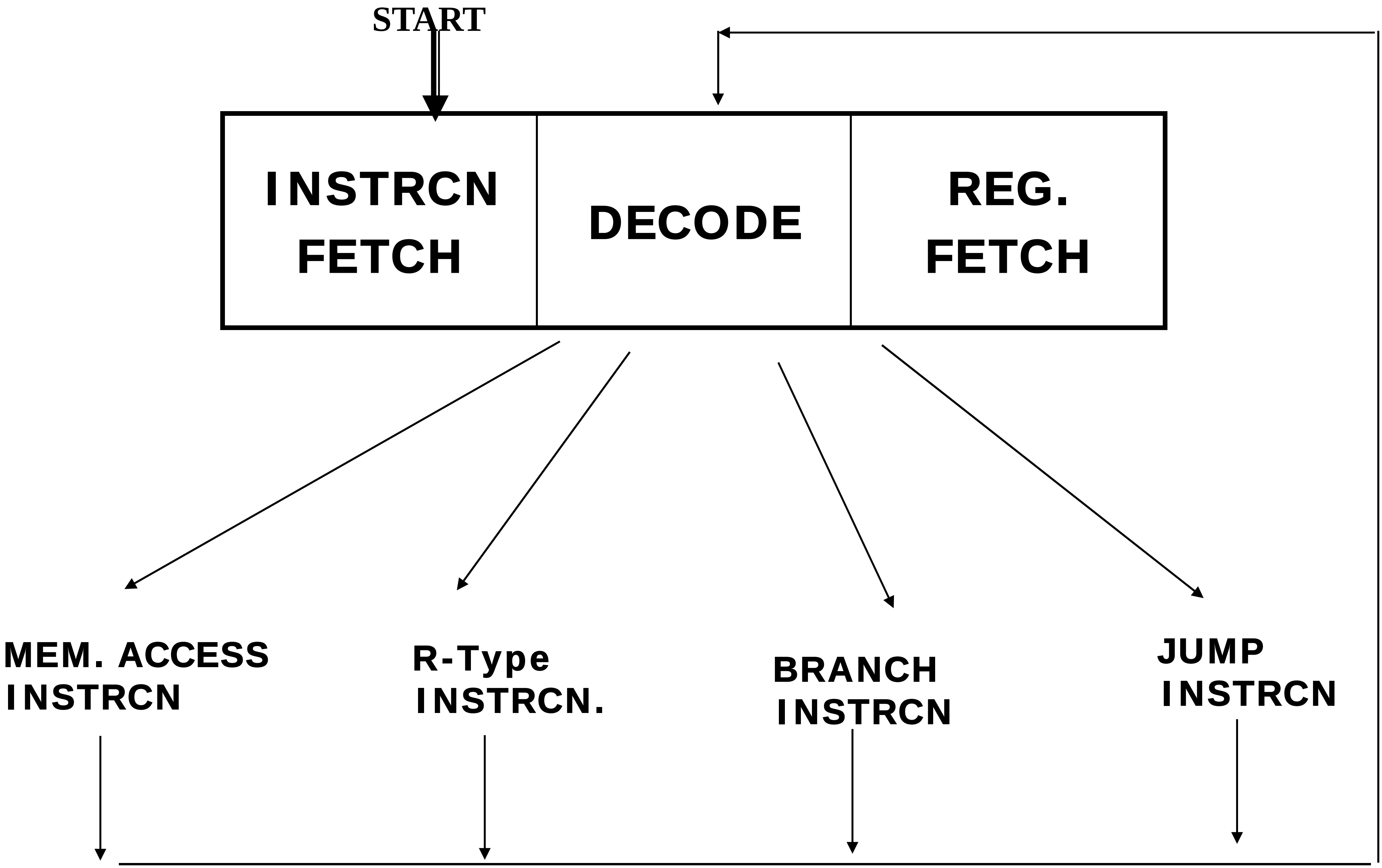
The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state.

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.



START

INSTRCN FETCH

DECODE

REG. FETCH

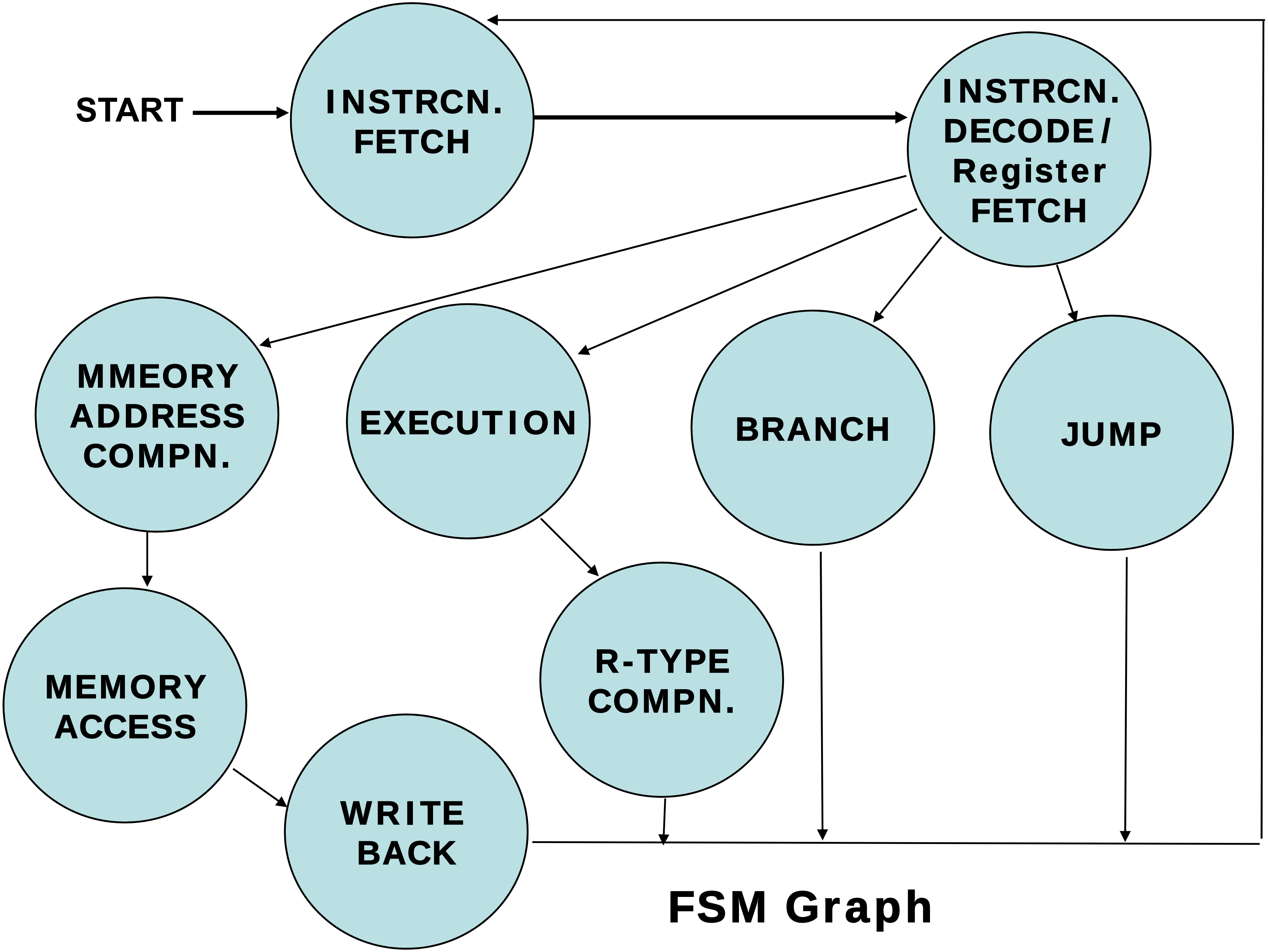
MEM. ACCESS INSTRCN

R-Type INSTRCN.

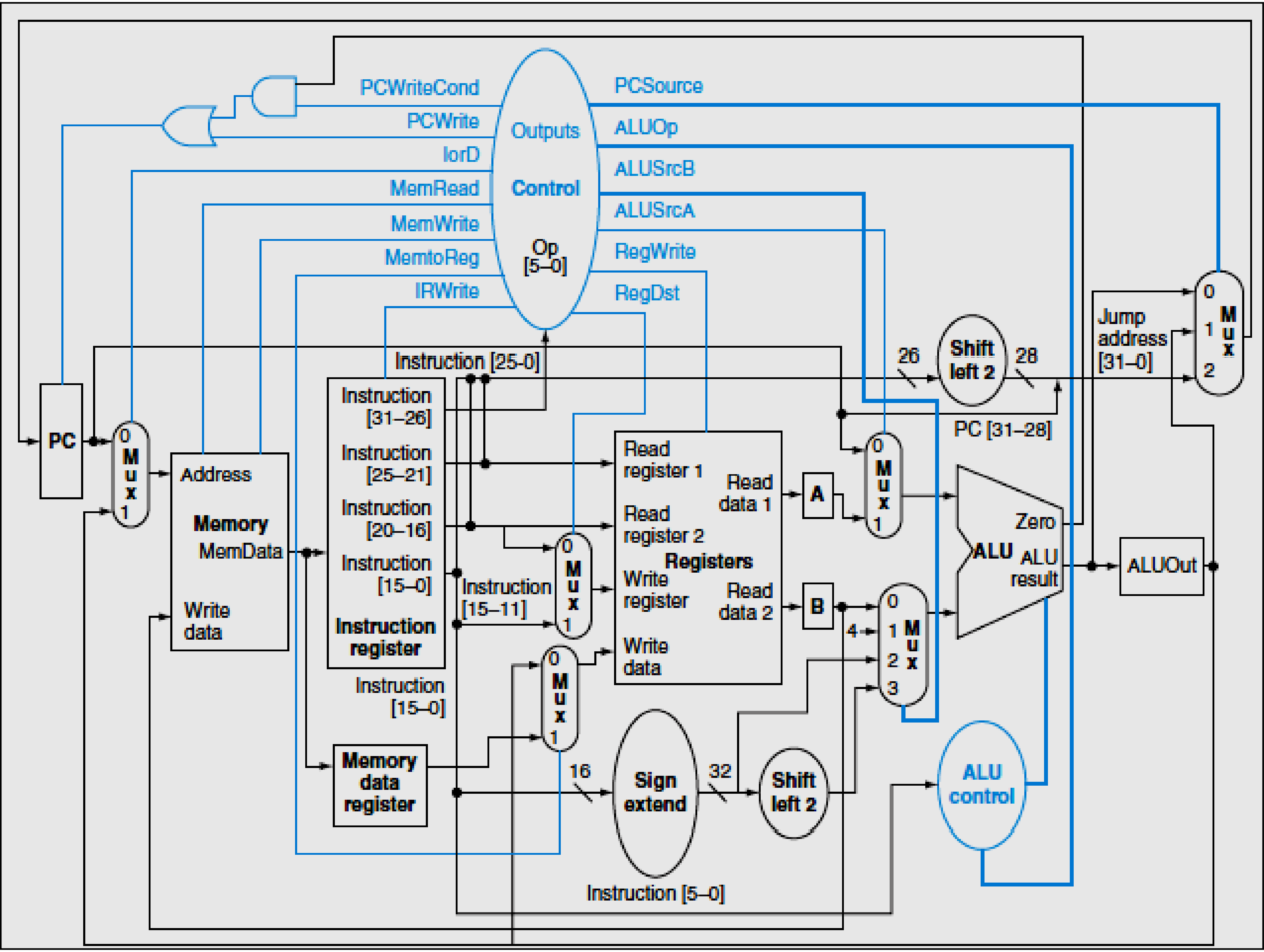
BRANCH INSTRCN

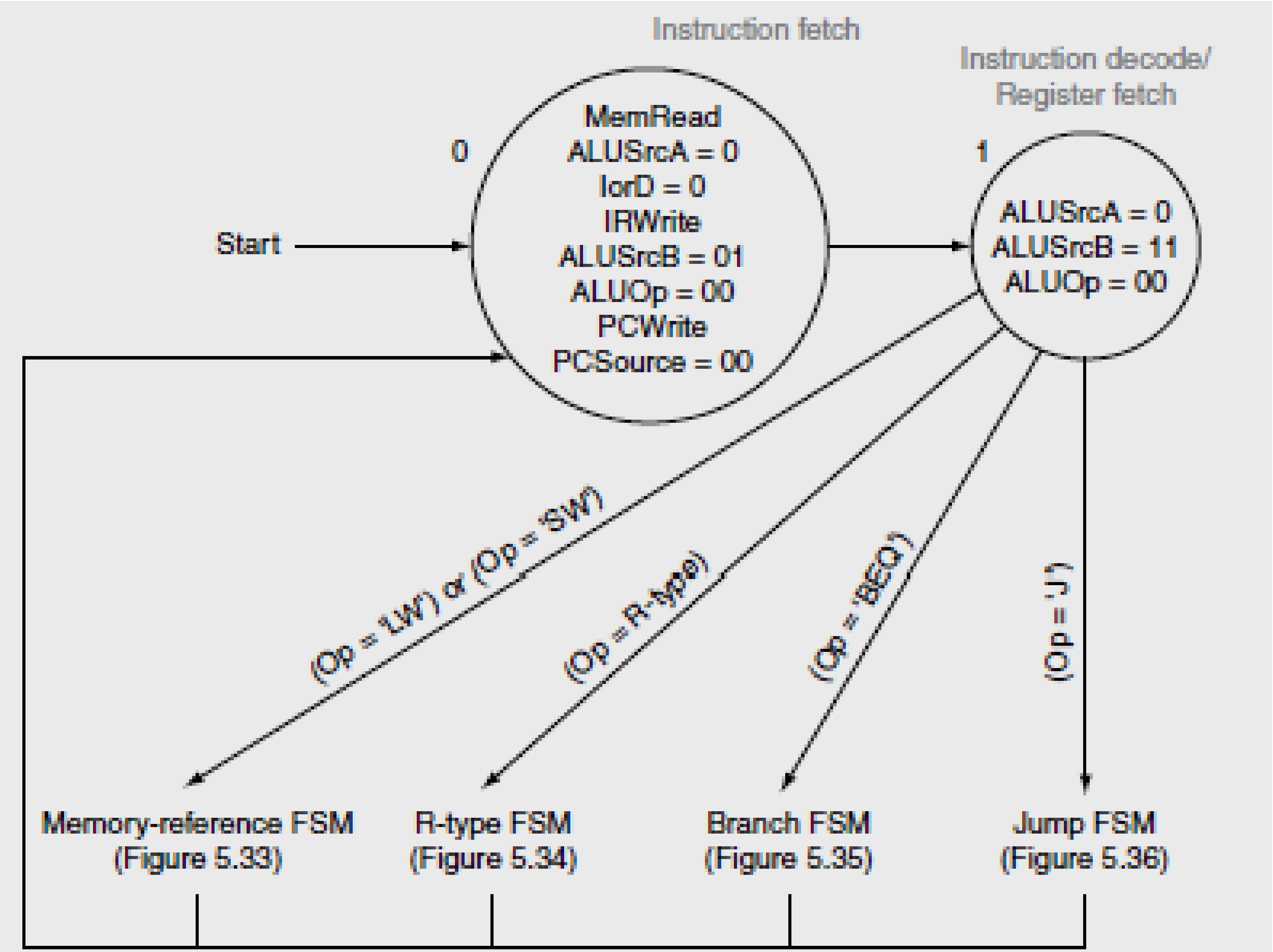
JUMP INSTRCN

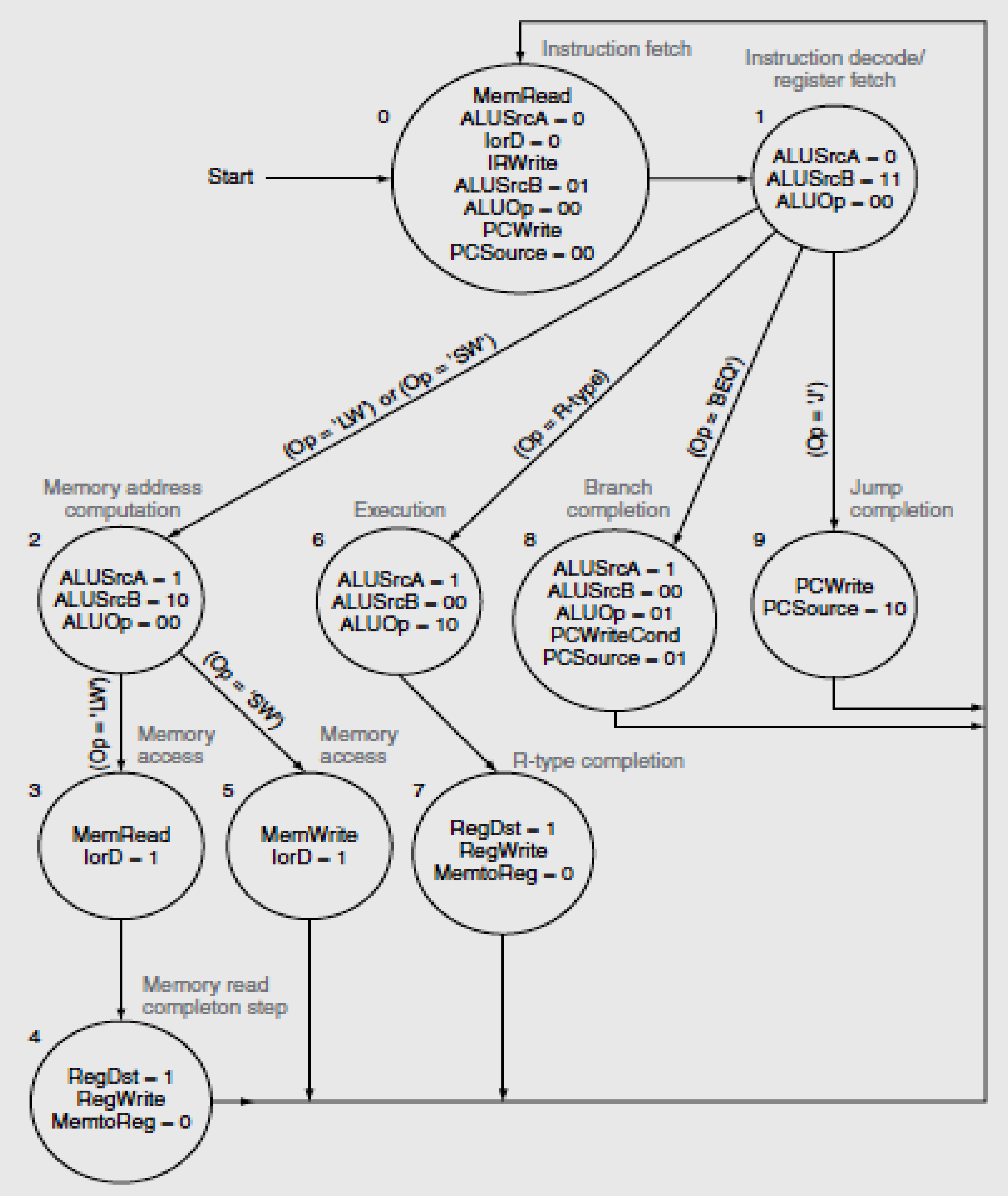
OVERALL state machine diagram for CPU

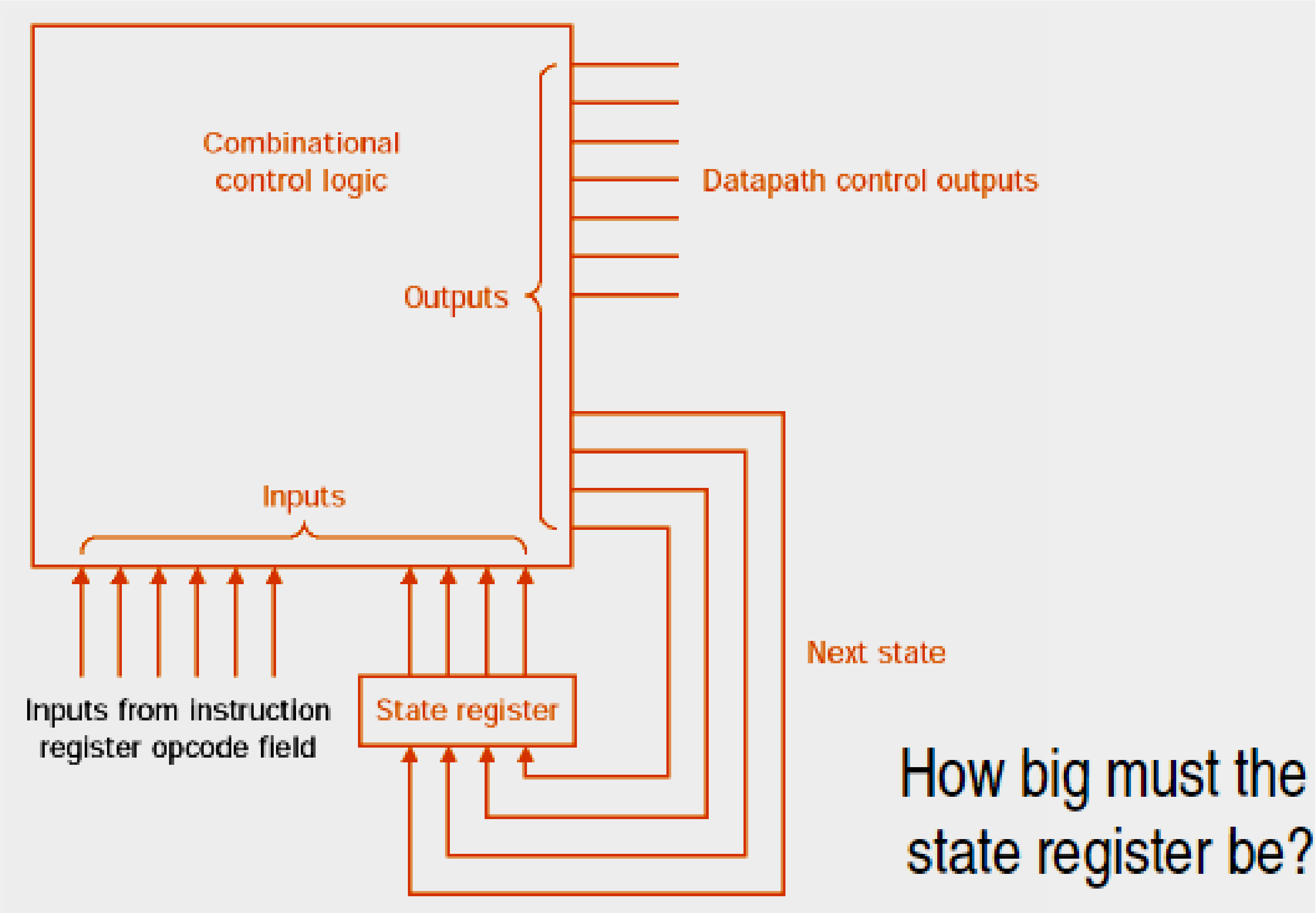

START
INSTRCN. FETCH
INSTRCN. DECODE / Register FETCH
MMEORY ADDRESS COMPN.
EXECUTION
BRANCH
JUMP
MEMORY ACCESS
R-TYPE COMPN.
WRITE BACK

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]
0
1
2
M
u
x
PC [31–28]
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–0]
Instruction
[15–11]
0
M
u
x
1
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
0
M
u
x
1
Zero
ALU
ALU
result
ALUOut
0
1
2
3
4
M
u
x
0
M
u
x
1
Memory
data
register
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5–0]


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)

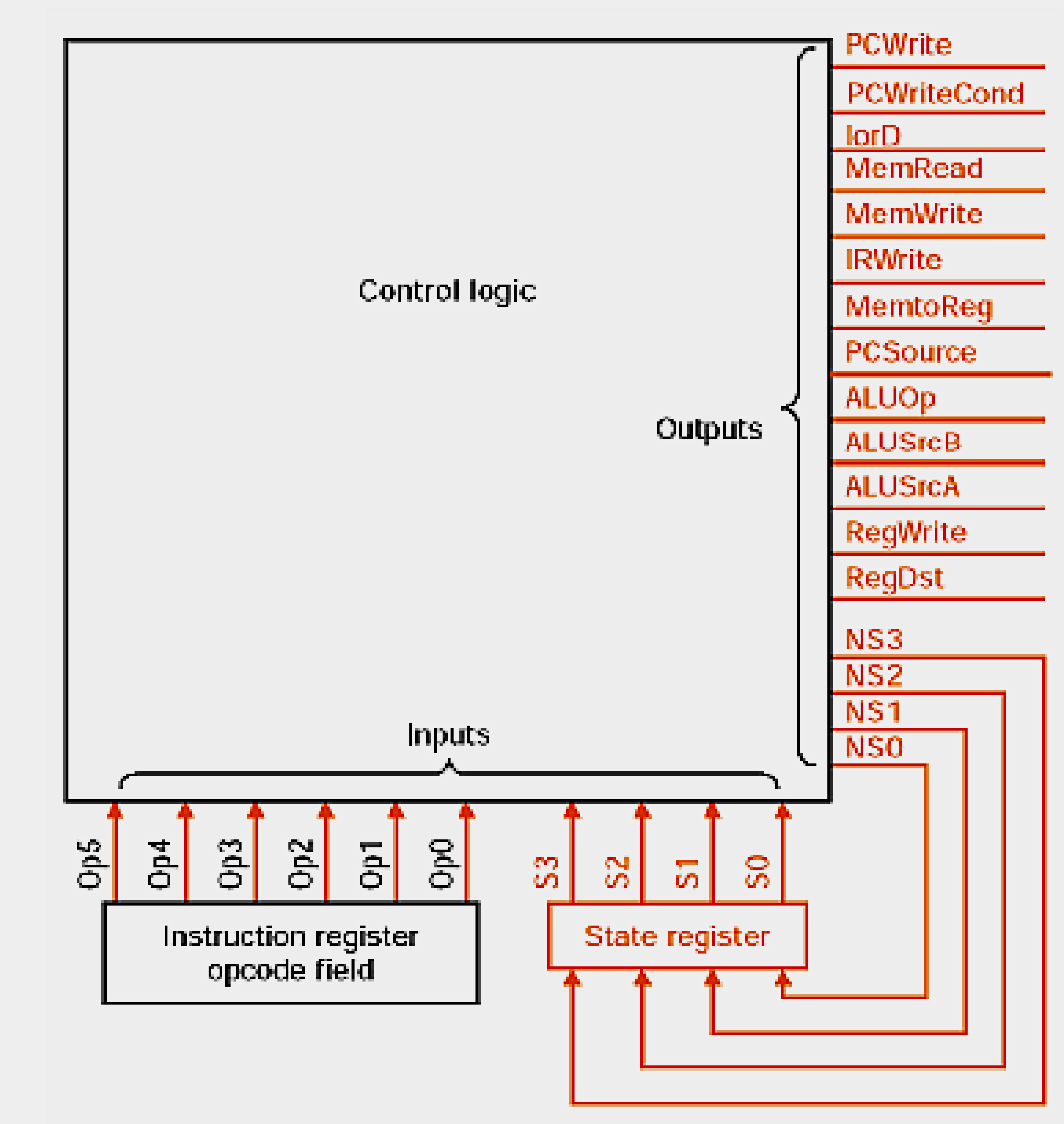

Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
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


Combinational control logic
Datapath control outputs
Outputs
Inputs
Next state
Inputs from instruction register opcode field
State register

How big must the state register be?

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

Next state depends on the input and current state.


PCWrite
PCWriteCond
IorD
MemRead
MemWrite
IRWrite
Control logic
MemtoReg
PCSource
ALUOp
Outputs
ALUSrcB
ALUSrcA
RegWrite
RegDst
NS3
NS2
NS1
NS0
Inputs
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')

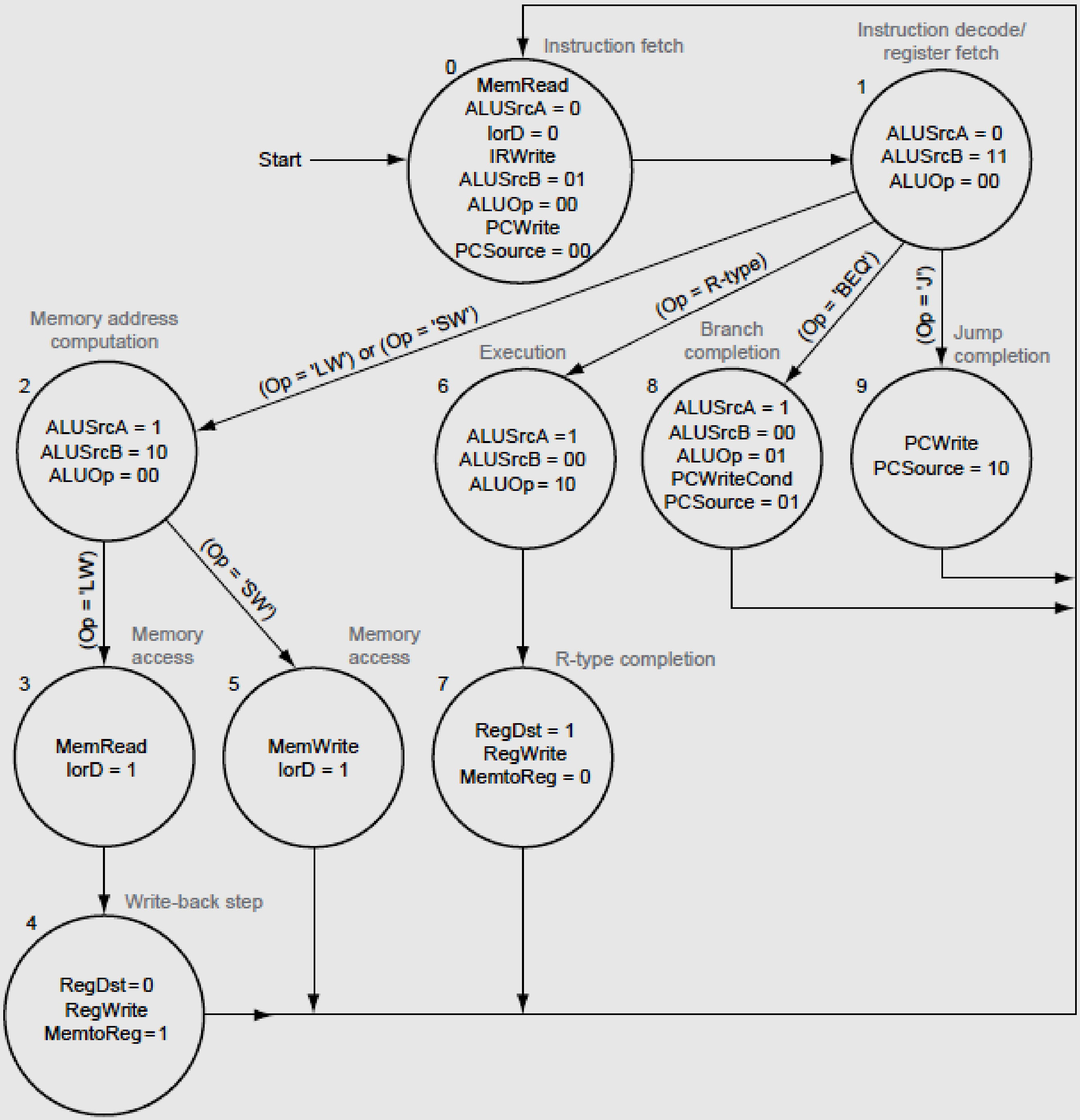


FIGURE C.3.3 The logic equations for the control unit shown in a shorthand form.

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')

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

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

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

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

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

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.

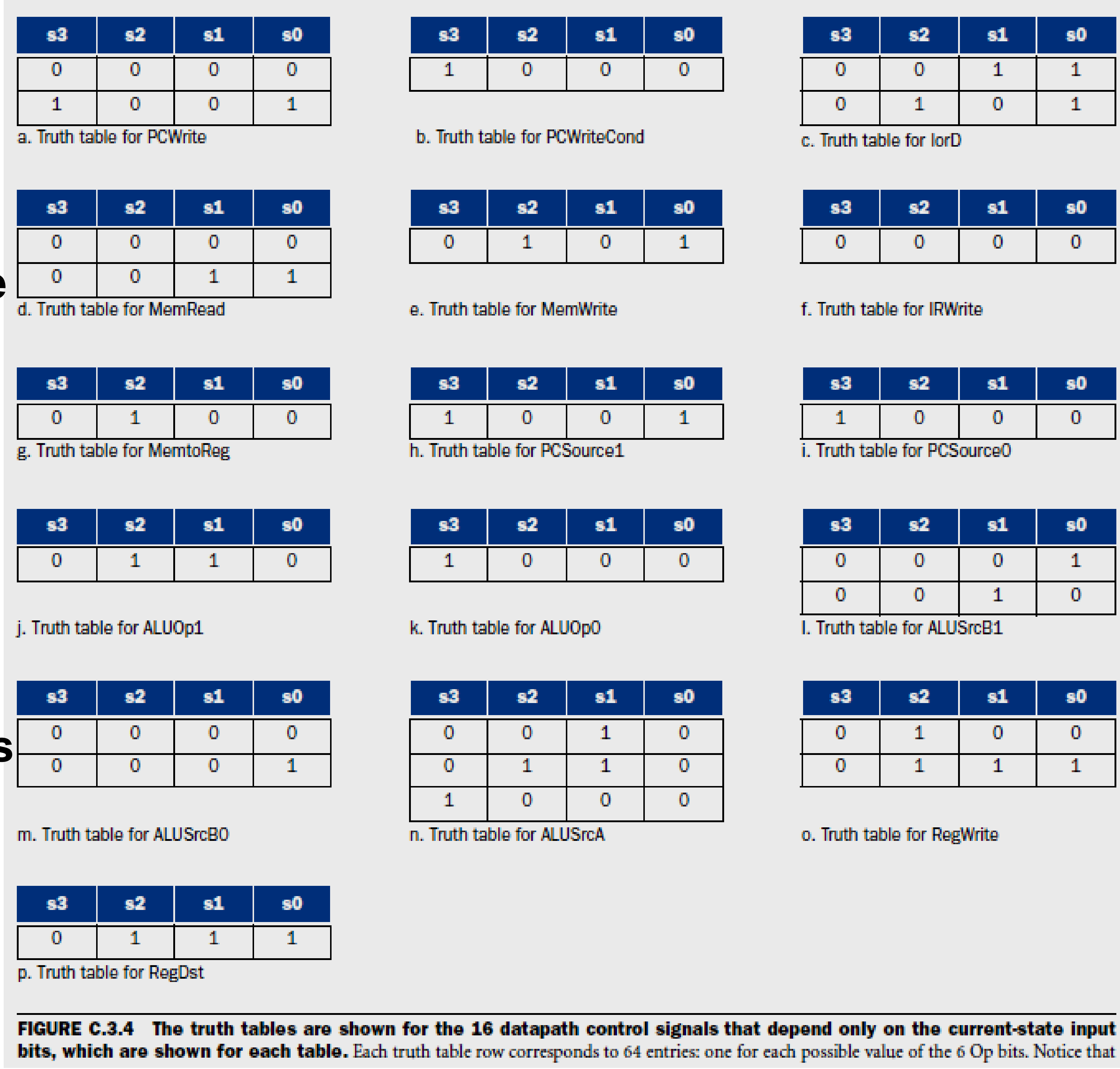
Break the control function into two parts:

- 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

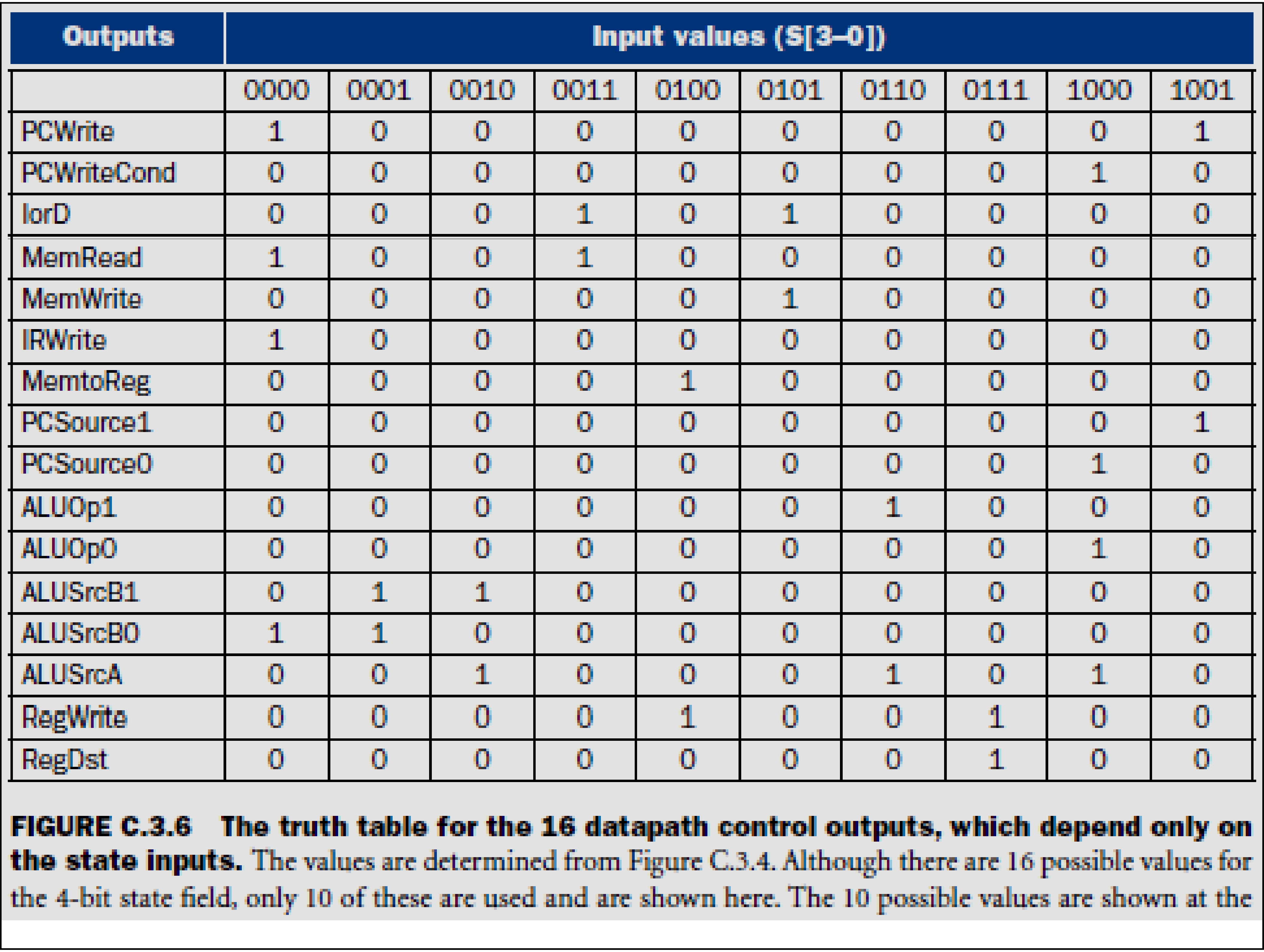
s3	s2	s1	s0
0	1	0	0
0	1	1	1

o. Truth table for RegWrite

s3	s2	s1	s0
0	1	1	1

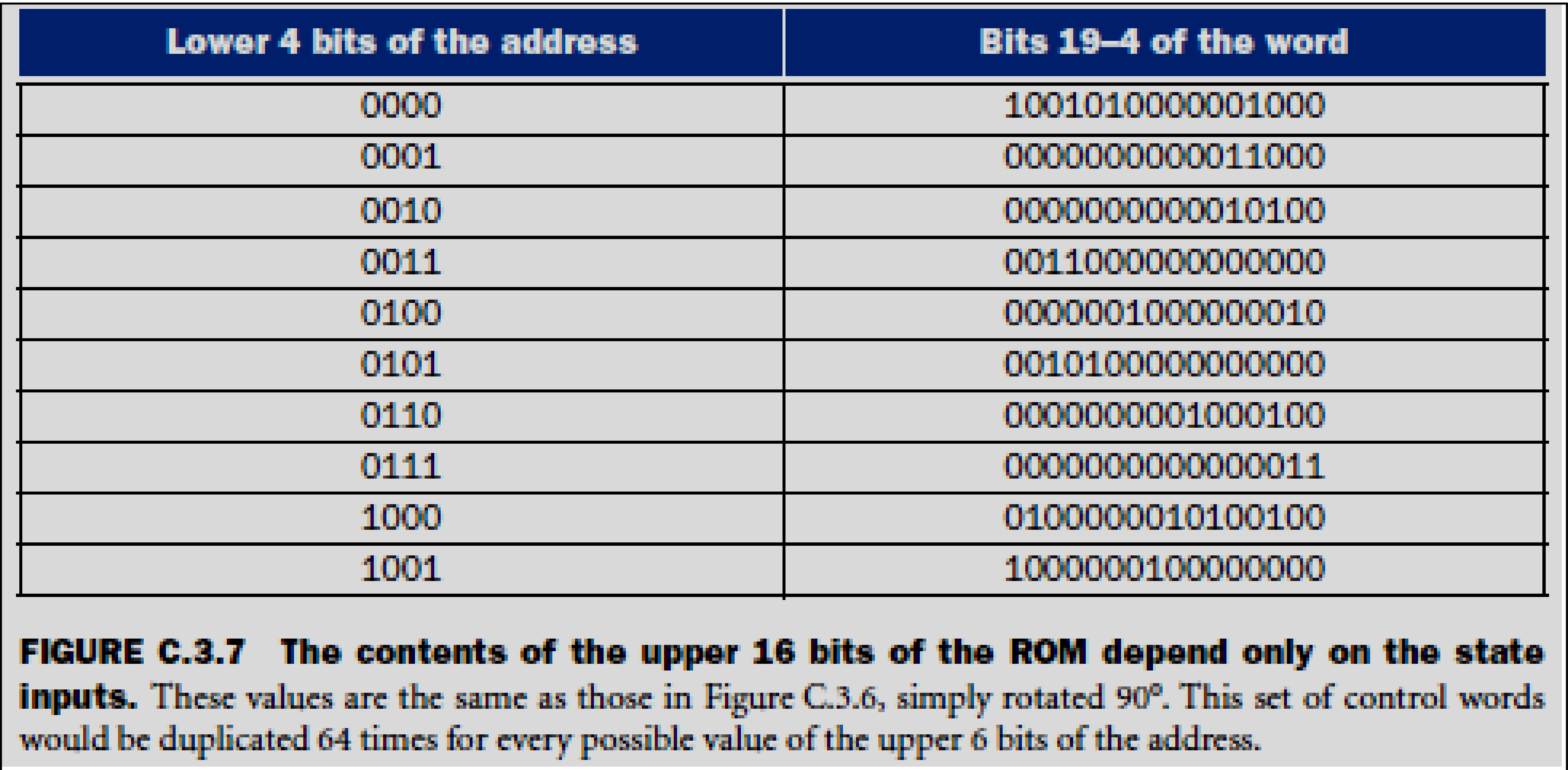
p. Truth table for RegDst

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

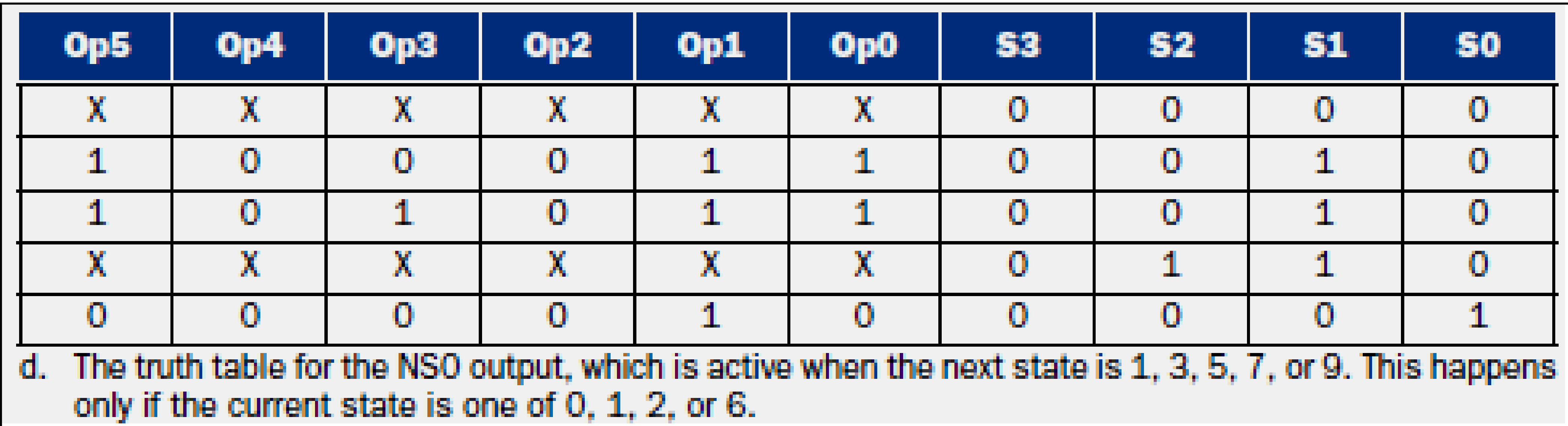


Lower 4 bits of the address	Bits 19–4 of the word
0000	1001010000001000
0001	0000000000011000
0010	0000000000010100
0011	0011000000000000
0100	0000001000000010
0101	0010100000000000
0110	0000000001000100
0111	0000000000000011
1000	0100000010100100
1001	1000000100000000

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 000000000, 0000001001, 0000010000, 0000011001, . . . , 1111110000, 1111111001.

The general form of this is XXXXXX0000 or XXXXXX1001.

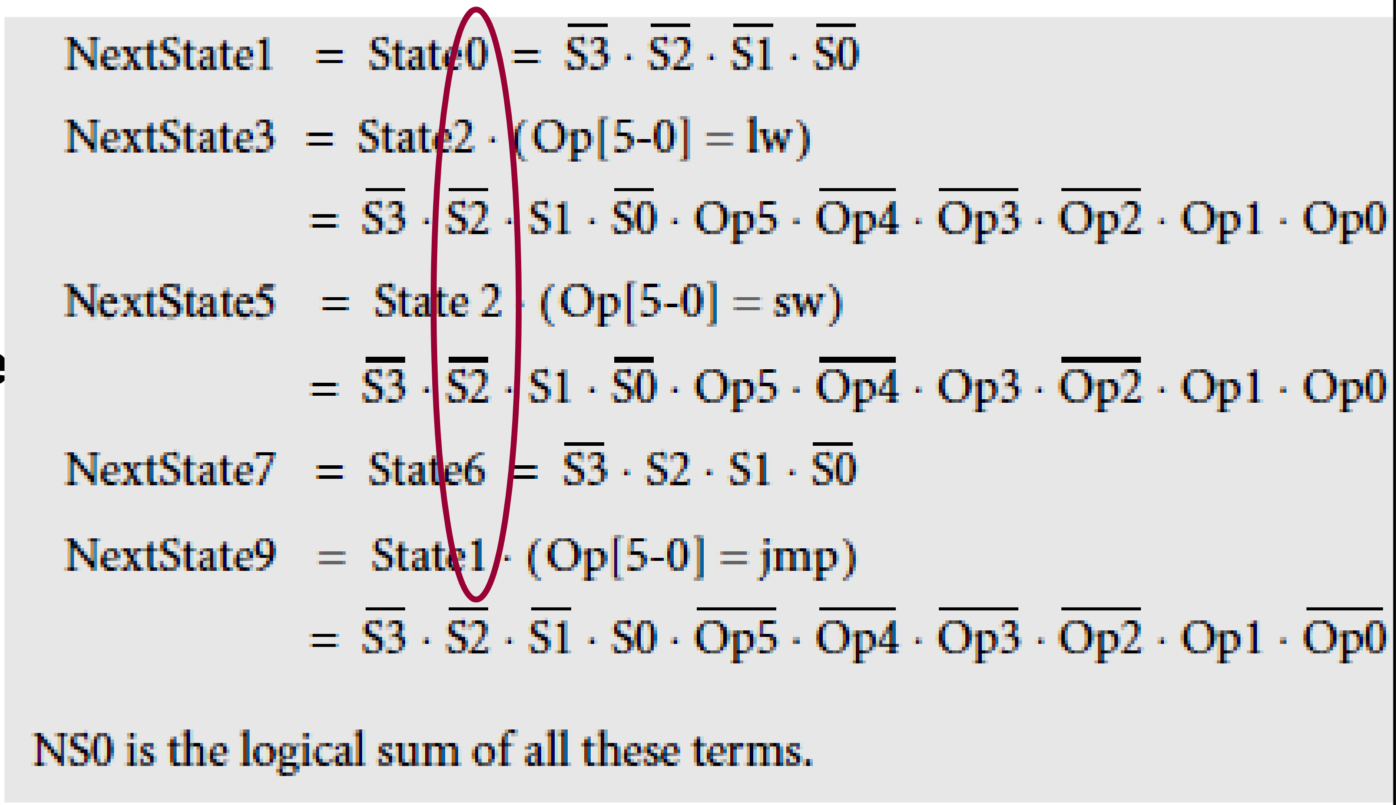


Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
X	X	X	X	X	X	0	0	0	0
1	0	0	0	1	1	0	0	1	0
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0
0	0	0	0	1	0	0	0	0	1

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]



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

$$\text{NextState3} = \text{State2} \cdot (\text{Op[5-0]} = \text{lw})$$

$$= \overline{\text{S3}} \cdot \overline{\text{S2}} \cdot \text{S1} \cdot \overline{\text{S0}} \cdot \text{Op5} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \text{Op0}$$

$$\text{NextState5} = \text{State 2} \cdot (\text{Op[5-0]} = \text{sw})$$

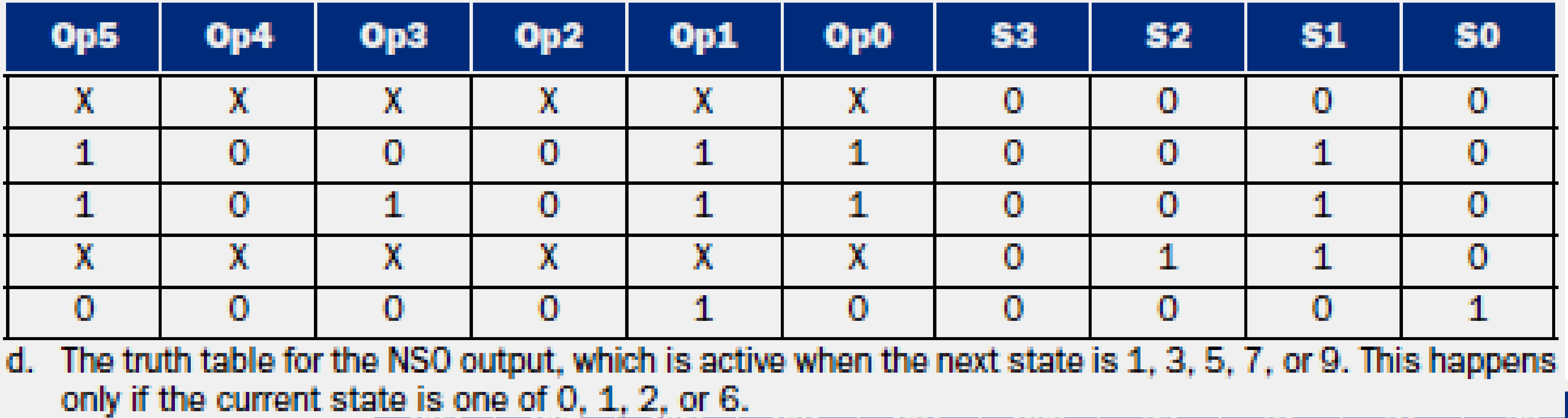
$$= \overline{\text{S3}} \cdot \overline{\text{S2}} \cdot \text{S1} \cdot \overline{\text{S0}} \cdot \text{Op5} \cdot \overline{\text{Op4}} \cdot \text{Op3} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \text{Op0}$$

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

$$\text{NextState9} = \text{State1} \cdot (\text{Op[5-0]} = \text{jmp})$$

$$= \overline{\text{S3}} \cdot \overline{\text{S2}} \cdot \overline{\text{S1}} \cdot \text{S0} \cdot \overline{\text{Op5}} \cdot \overline{\text{Op4}} \cdot \overline{\text{Op3}} \cdot \overline{\text{Op2}} \cdot \text{Op1} \cdot \overline{\text{Op0}}$$

NS0 is the logical sum of all these terms.

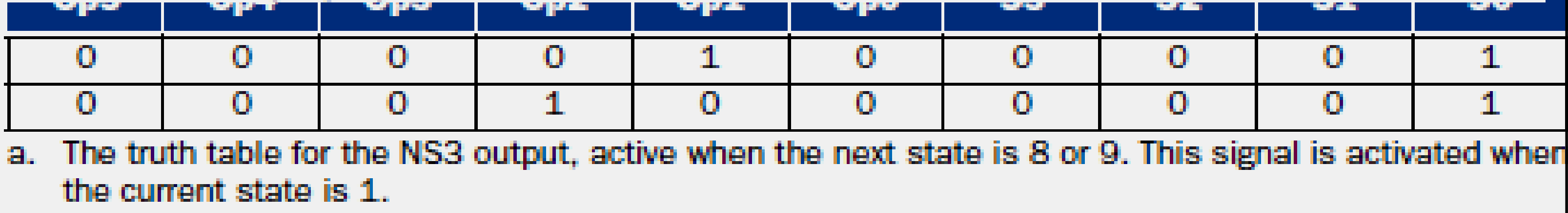


Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
X	X	X	X	X	X	0	0	0	0
1	0	0	0	1	1	0	0	1	0
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0
0	0	0	0	1	0	0	0	0	1

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.

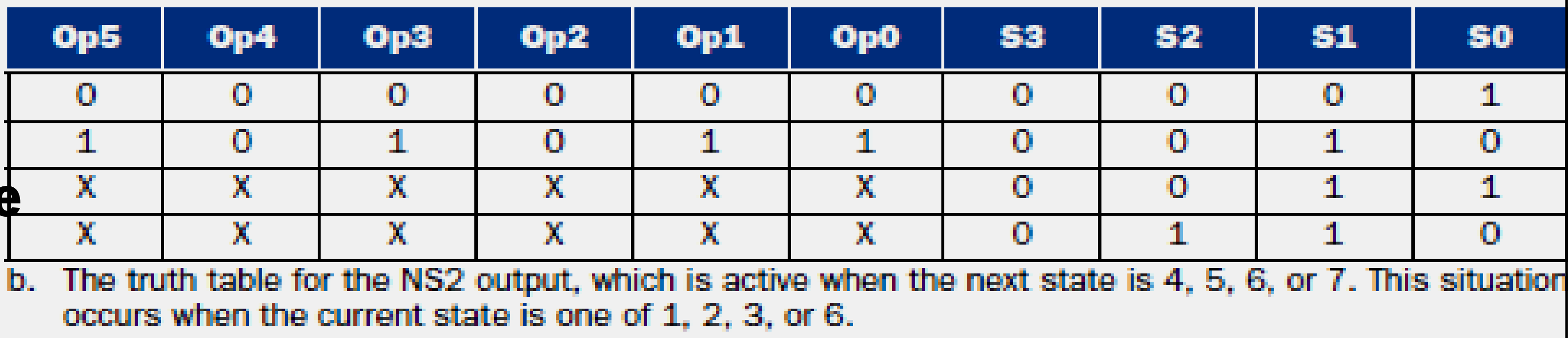
The four truth tables for the four **next-state output bits** (NS[3–0]).

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].



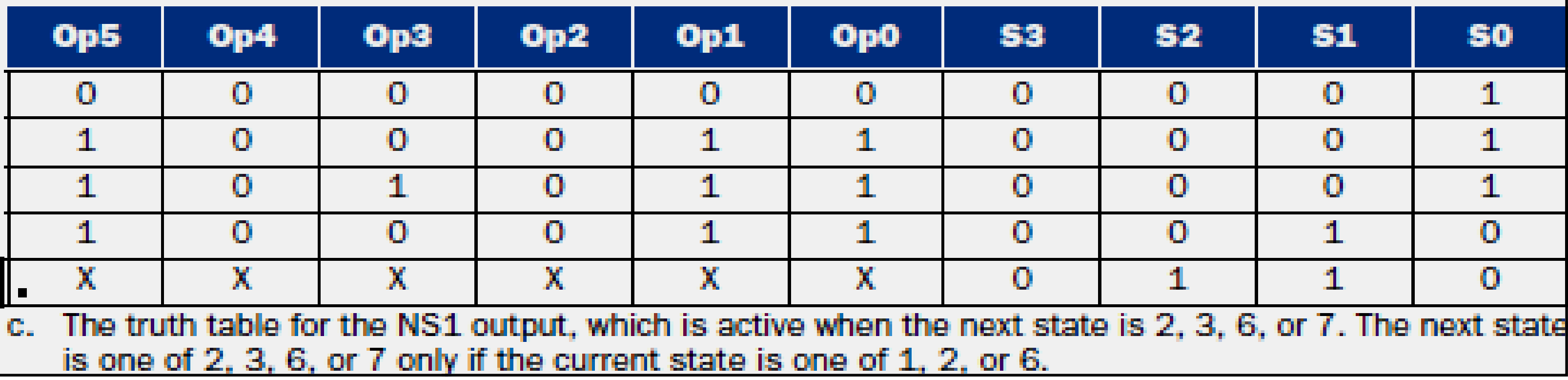
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	0	1

a. The truth table for the NS3 output, active when the next state is 8 or 9. This signal is activated wher the current state is 1.



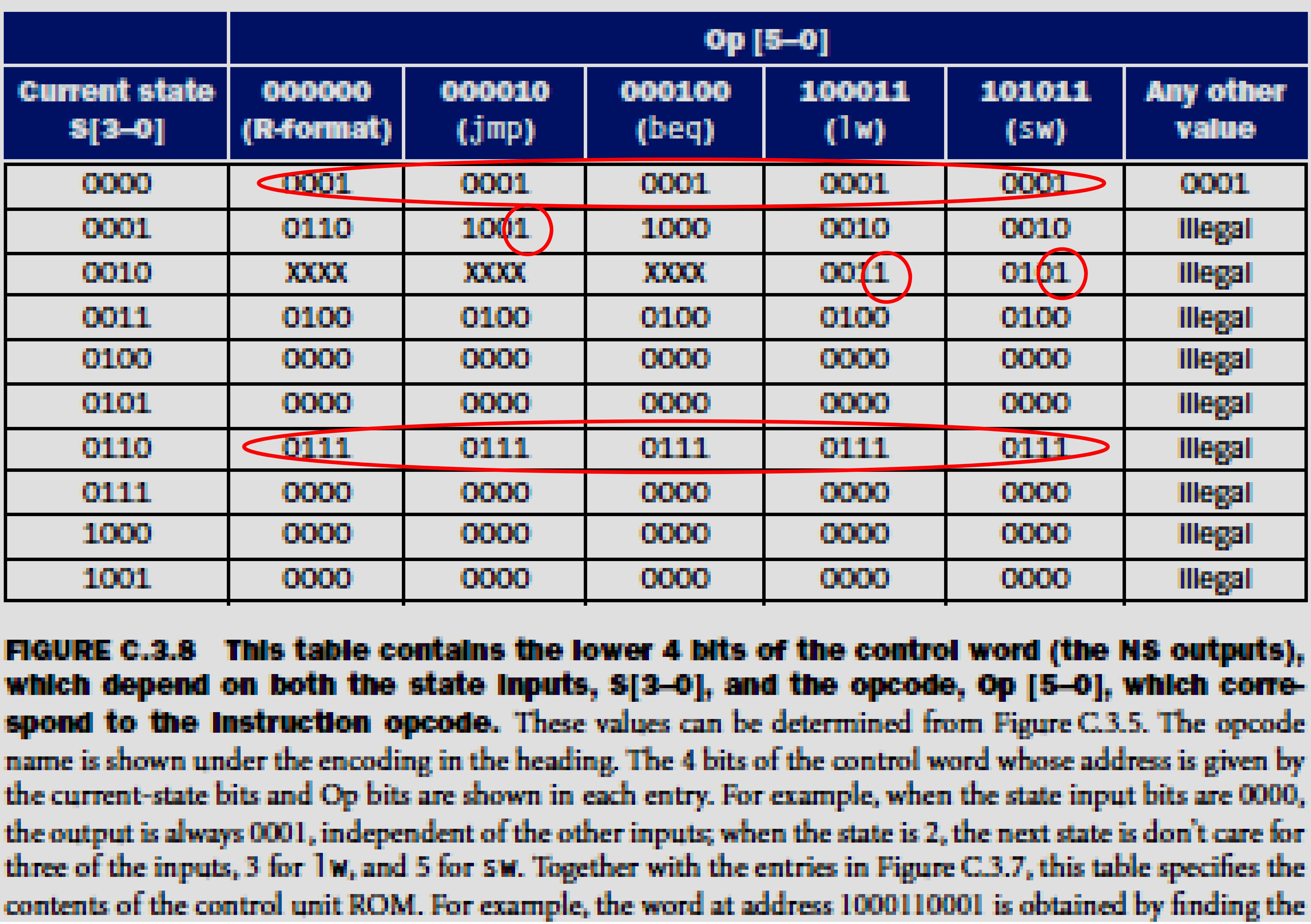
Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	1	0	1	1	0	0	1	0
X	X	X	X	X	X	0	0	1	1
X	X	X	X	X	X	0	1	1	0

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.



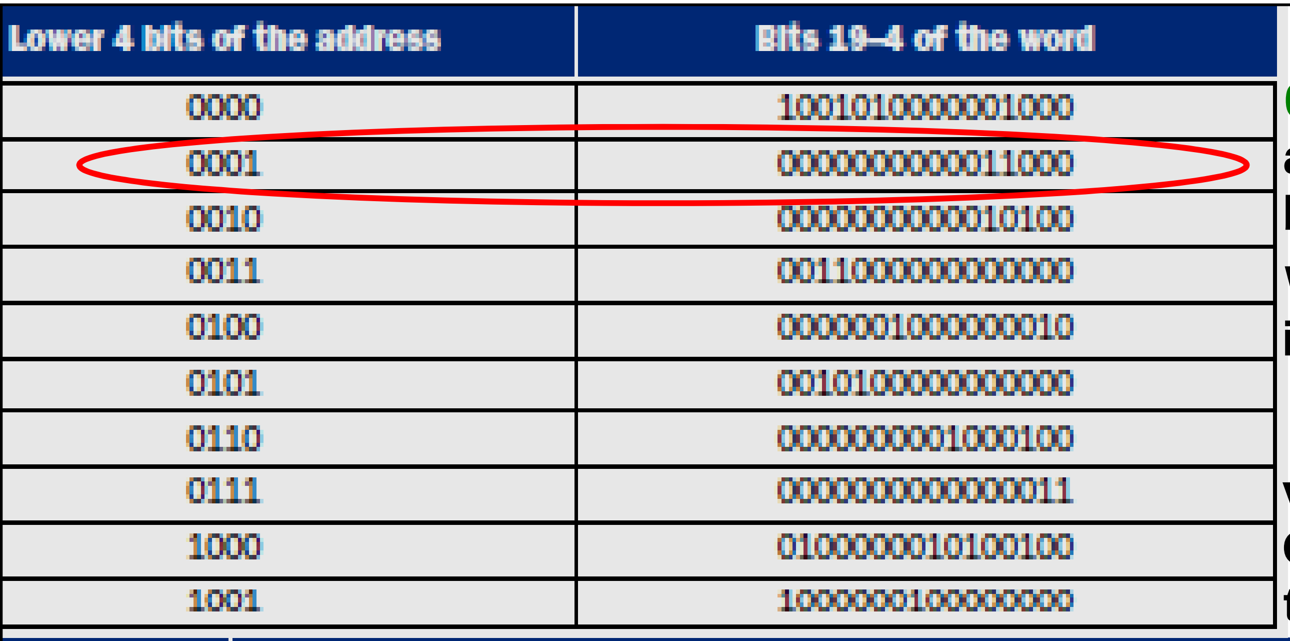
Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	0	1
1	0	1	0	1	1	0	0	0	1
1	0	0	0	1	1	0	0	1	0
X	X	X	X	X	X	0	1	1	0

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

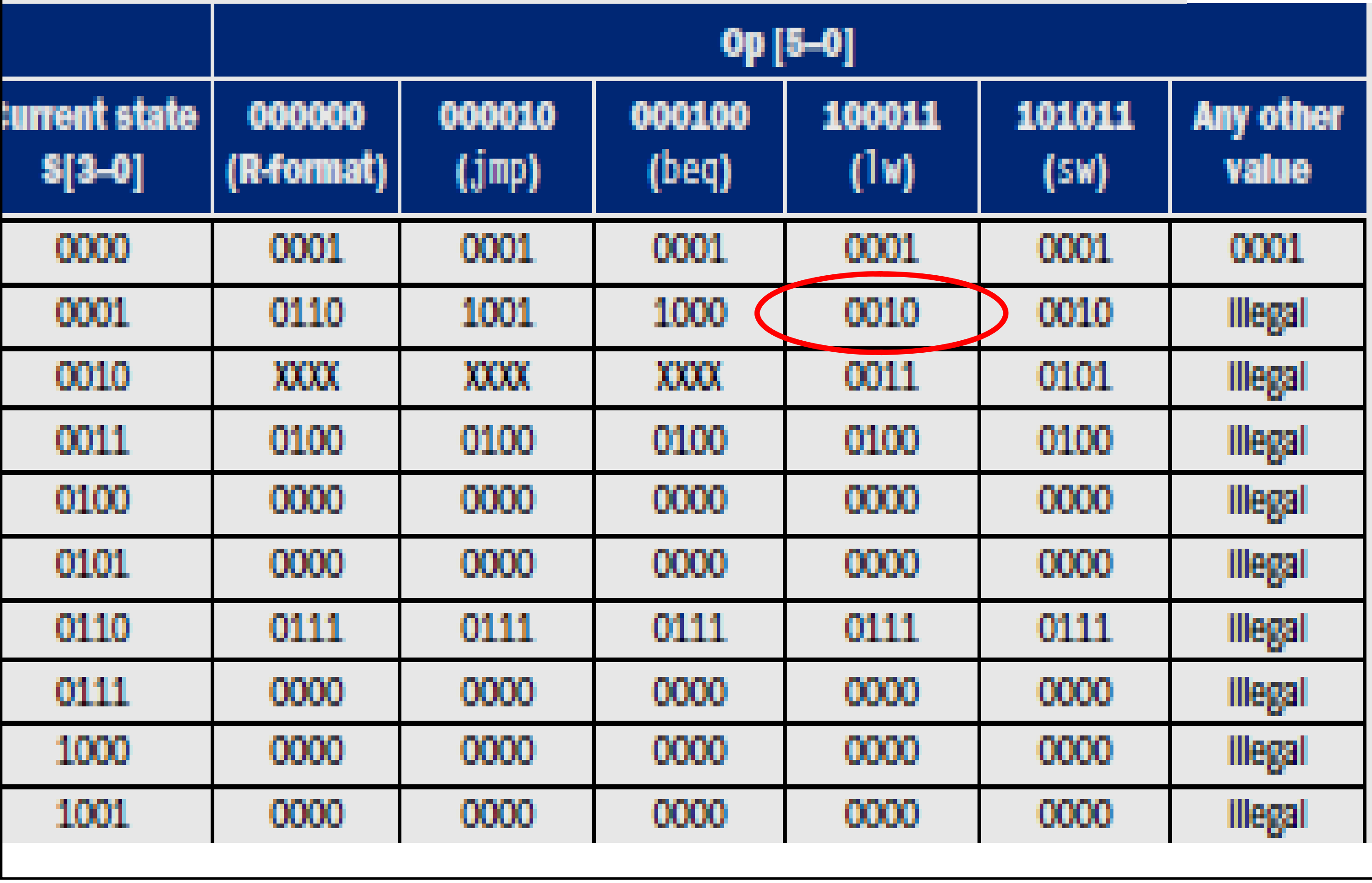
FIGURE C.3.8 This table contains the lower 4 bits of the control word (the NS outputs), which depend on both the state inputs, S[3–0], and the opcode, Op [5–0], which correspond to the instruction opcode. These values can be determined from Figure C.3.5. The opcode name is shown under the encoding in 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 00000000000110000010.

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



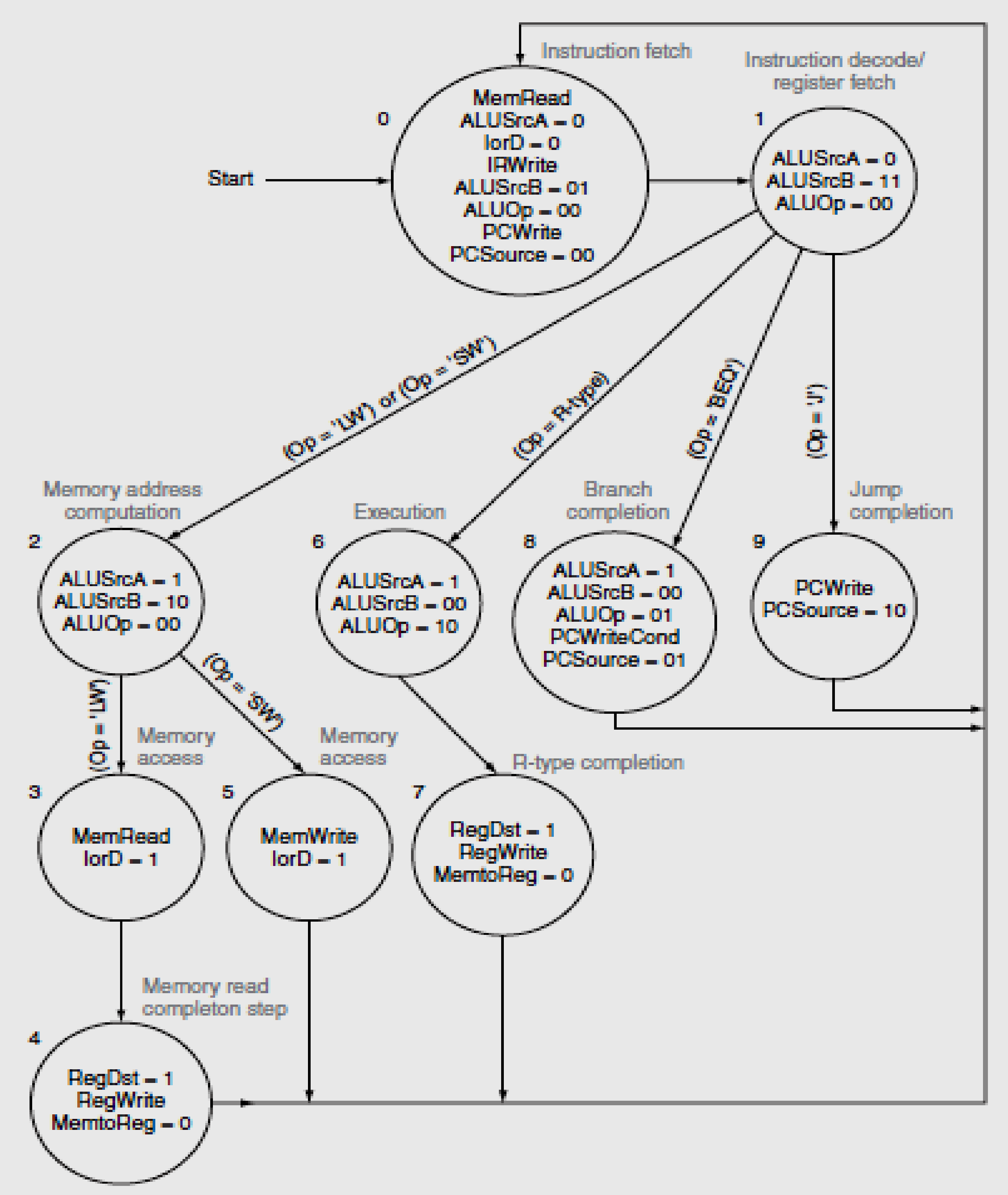
	Op [5–0]					
urrent state S[3–0]	000000 (R-format)	000010 (jmp)	000100 (beq)	100011 (lw)	101011 (sw)	Any other value
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	Illegal
0010	XXXX	XXXX	XXXX	0011	0101	Illegal
0011	0100	0100	0100	0100	0100	Illegal
0100	0000	0000	0000	0000	0000	Illegal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

Cond

ad

te

eg

e1

e0

31

30

A

e

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 ??

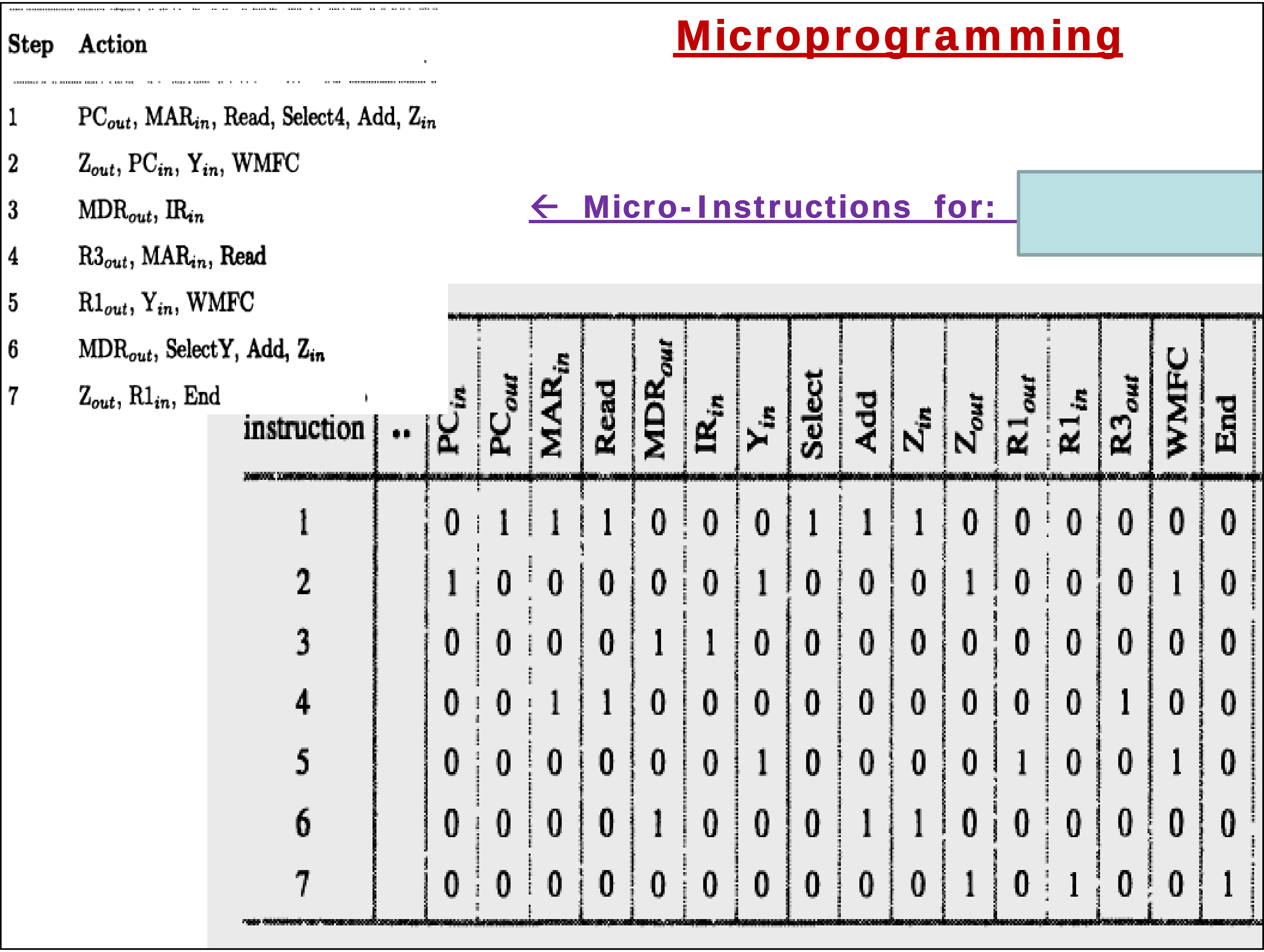
Minor modifications of the ISA requires lot of changes and redo the design.

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.

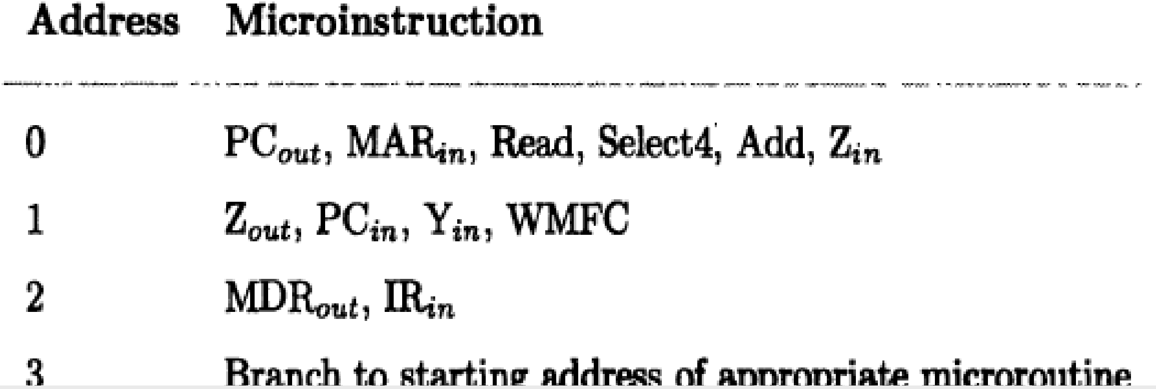
Microprogramming

← Micro-Instructions for:

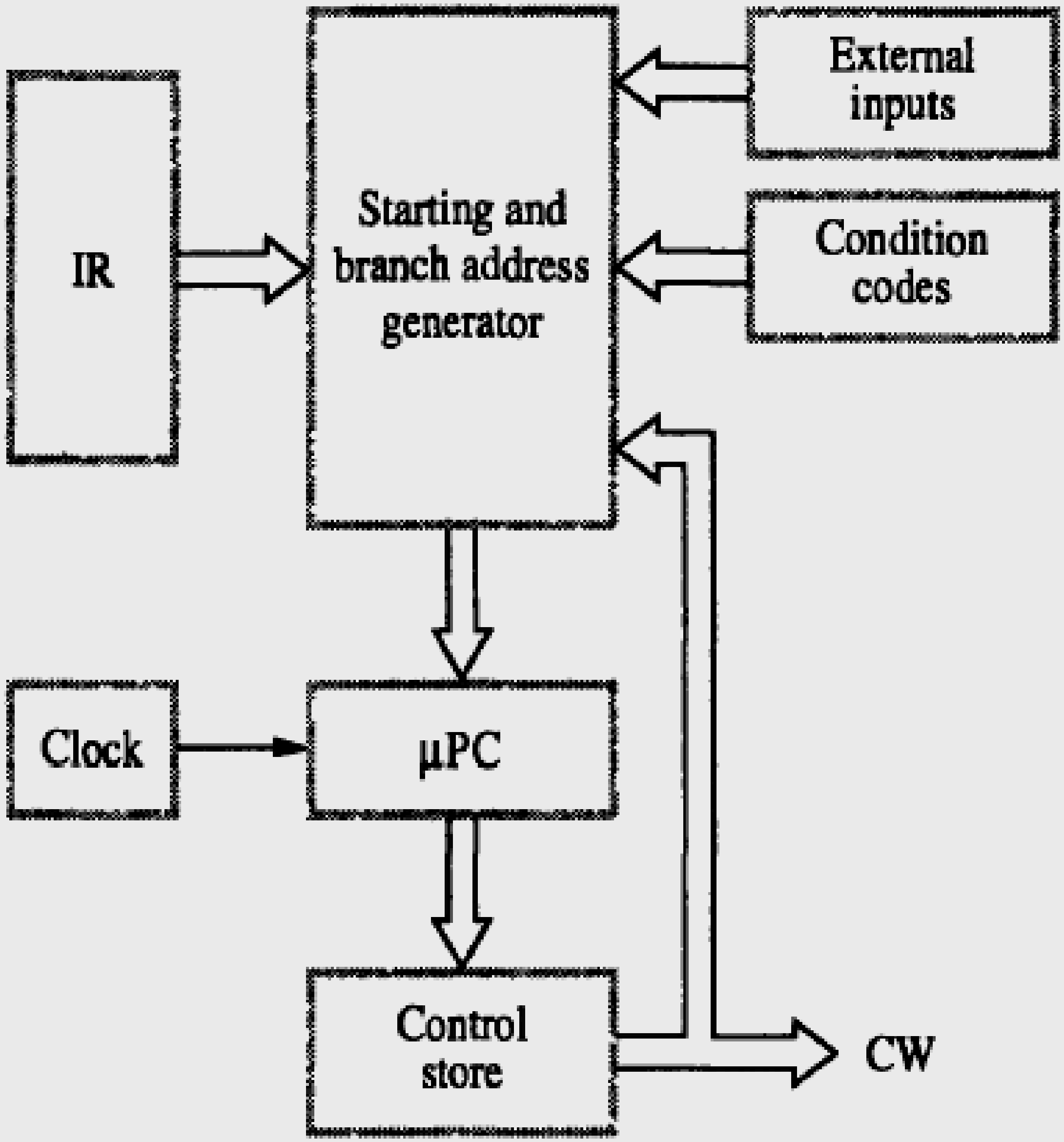


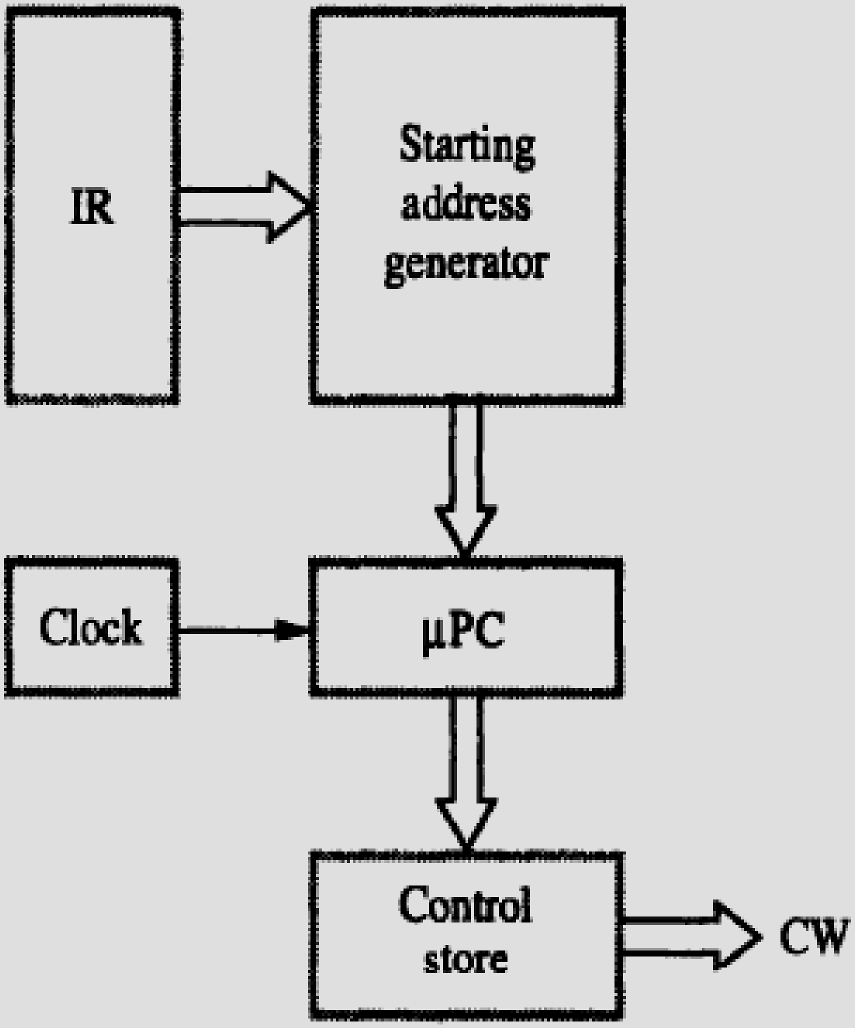
Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

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 *$TEMP_{out}$* .

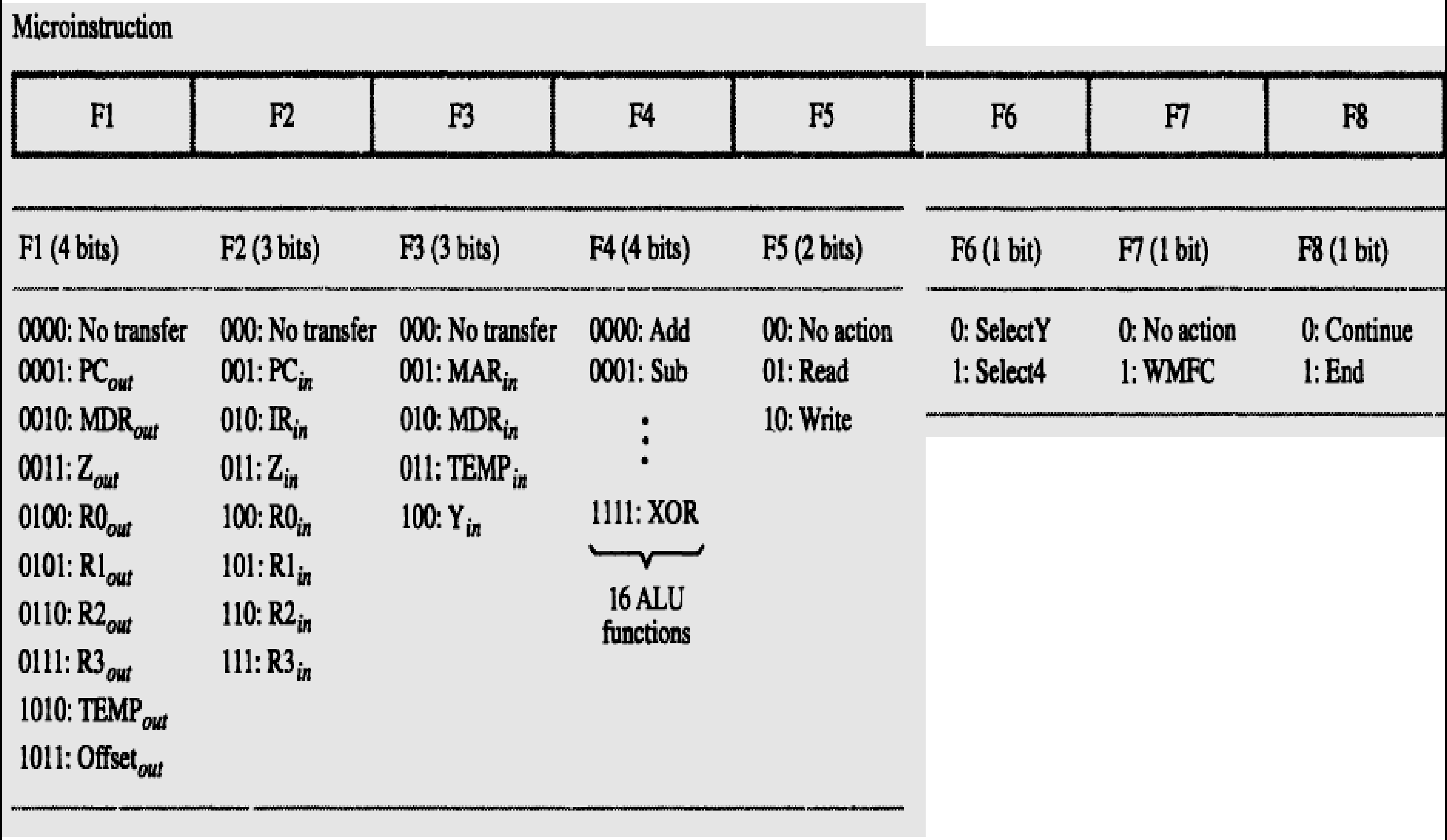
***Thus, do this natural grouping (of* mutually exclusive signals) and then -**

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.

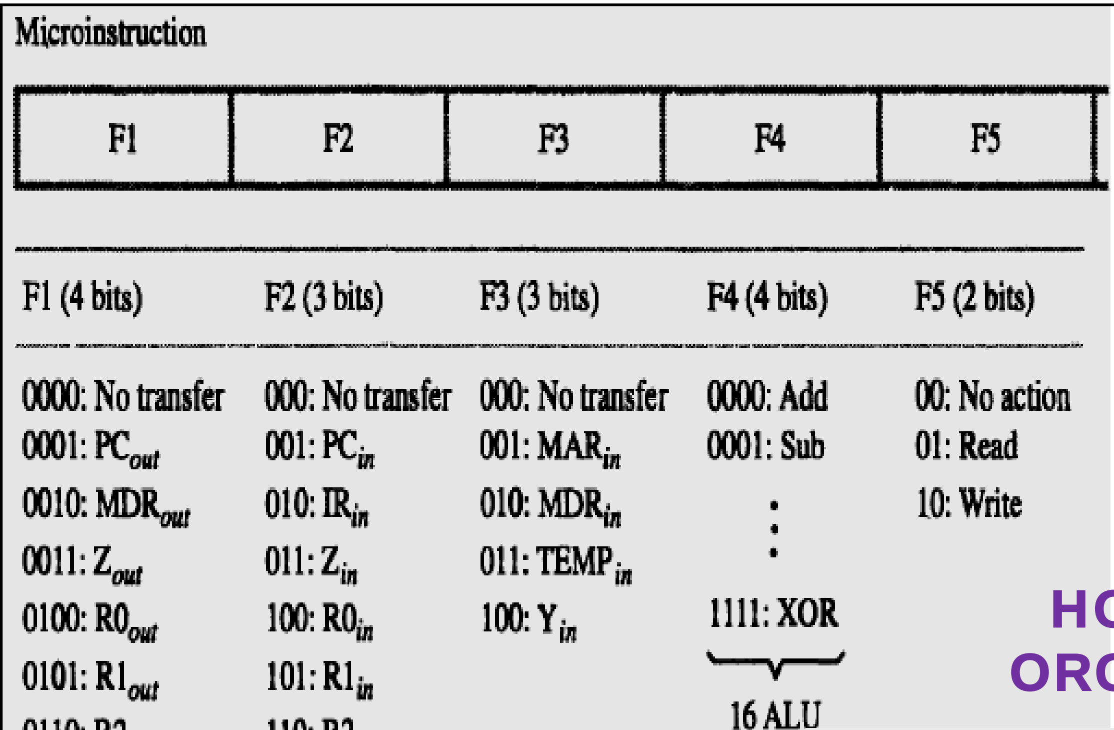


Microinstruction

F1	F2	F3	F4	F5	F6	F7	F8

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



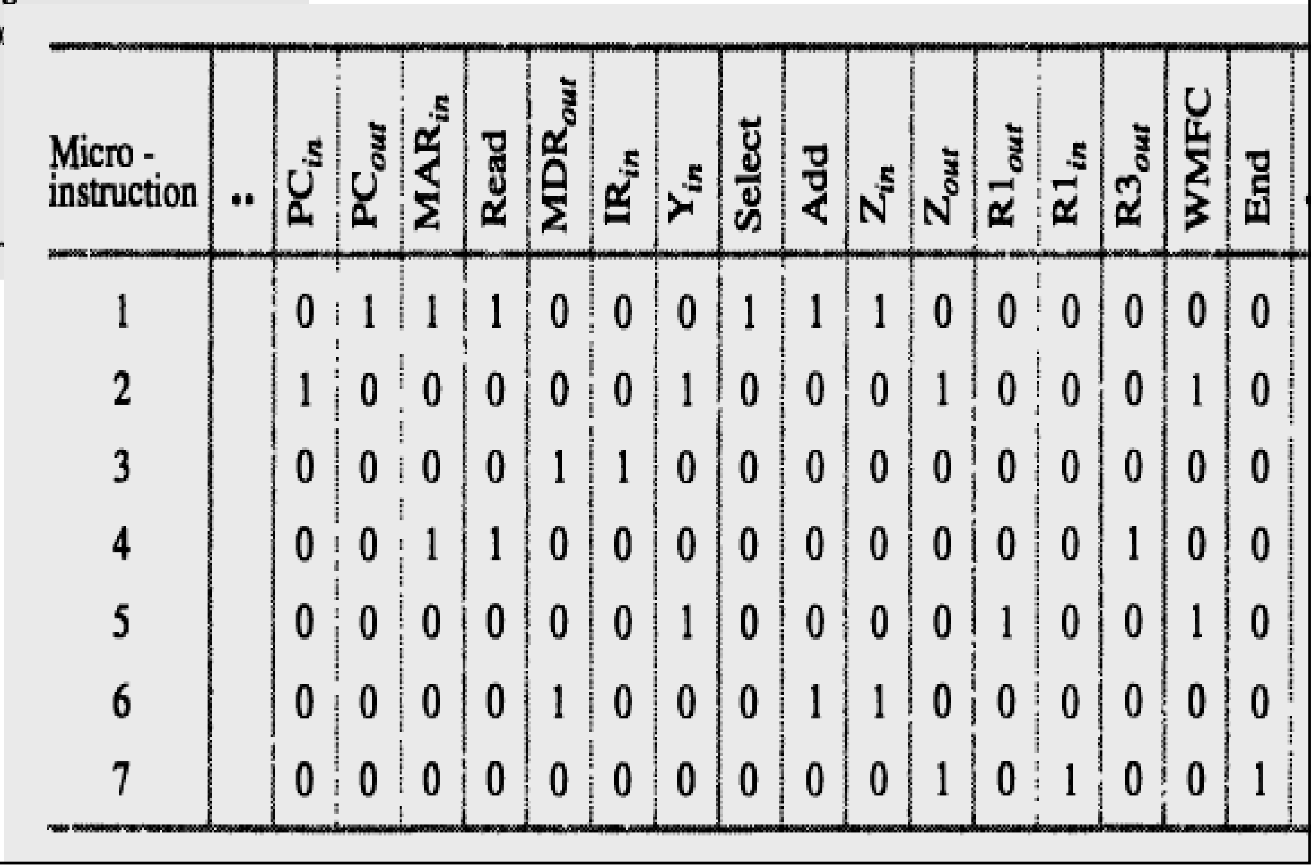
Microinstruction

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.

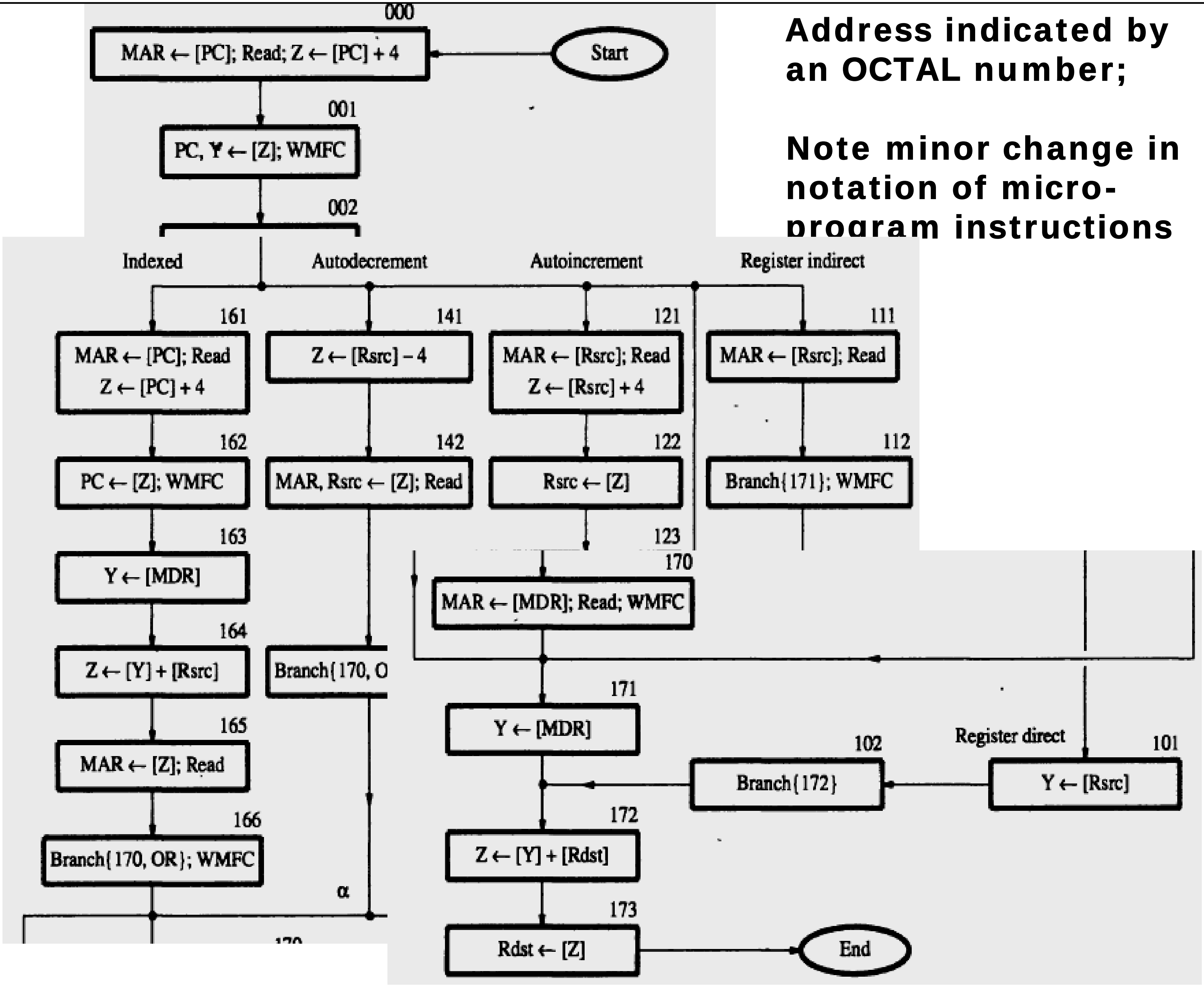
e.g. Consider an instruction of the type:

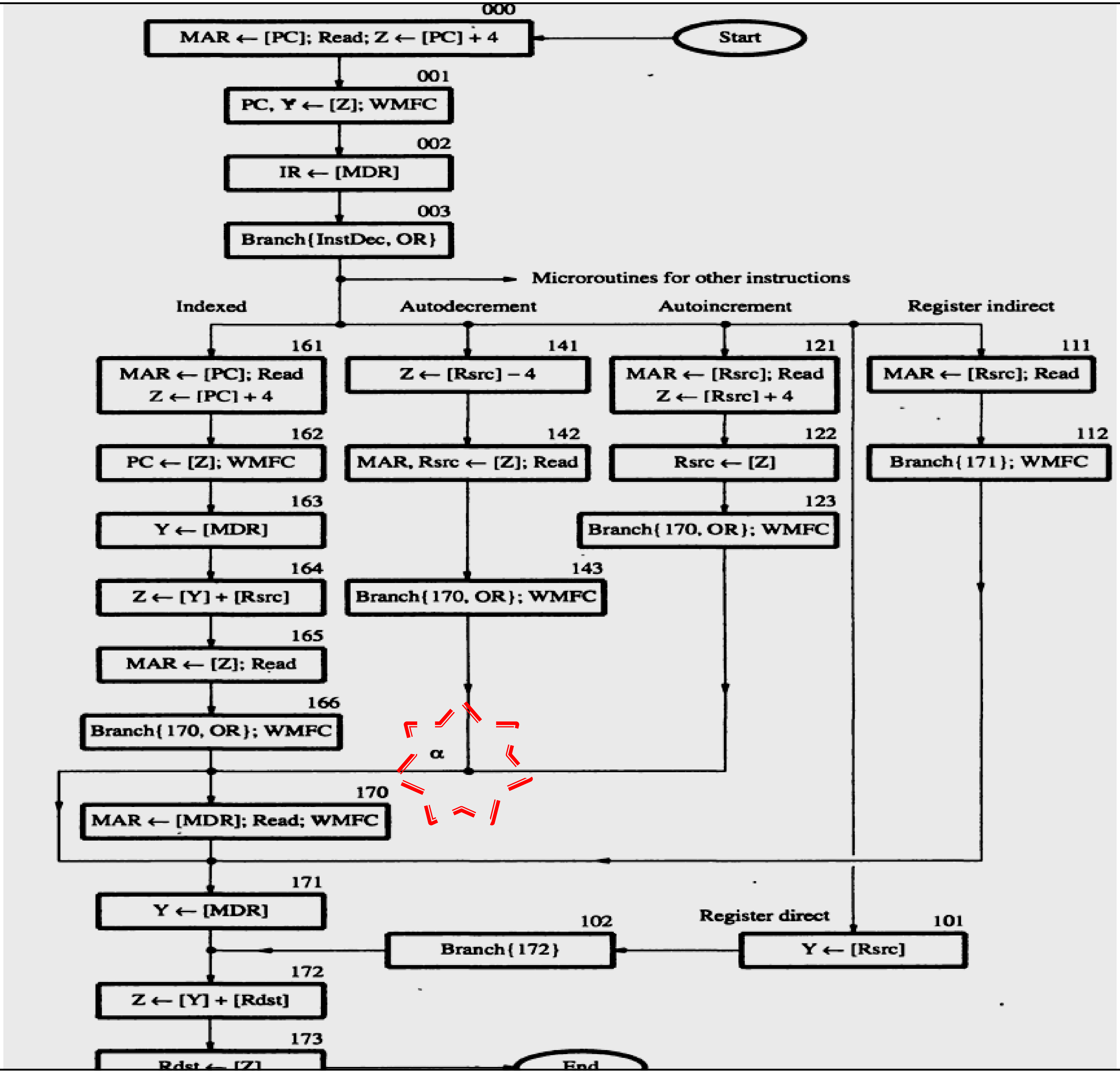
$\text{Add} \quad R_{src}, \quad R_{dst}$

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]
Register direct
102
Branch{172}
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.

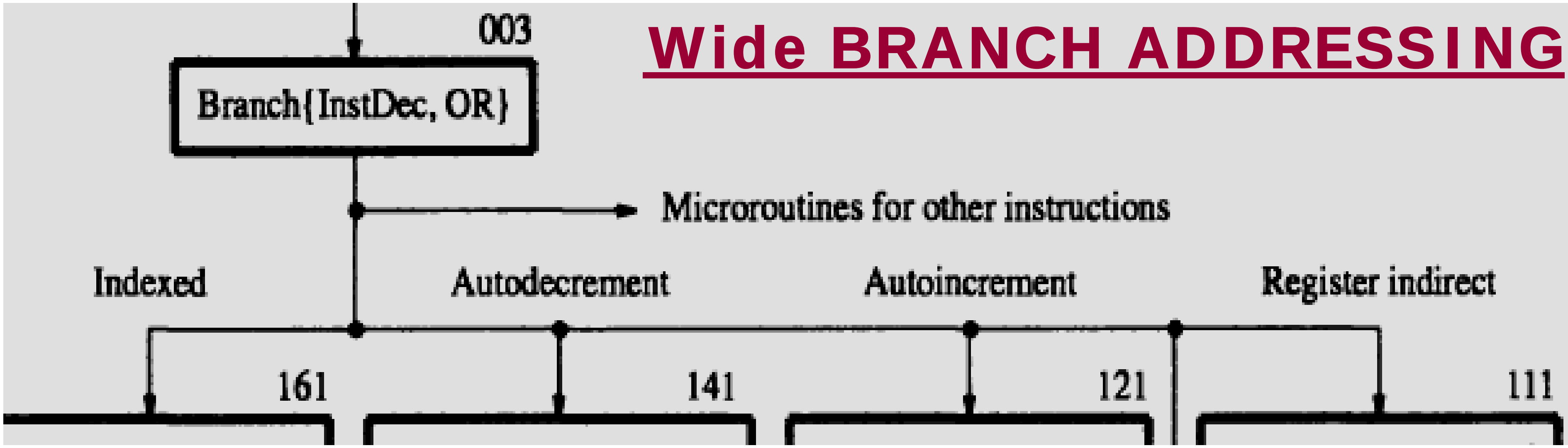
If the indirect mode is specified in the instruction, then the microinstruction in location 170 is performed to fetch the operand from the memory.

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



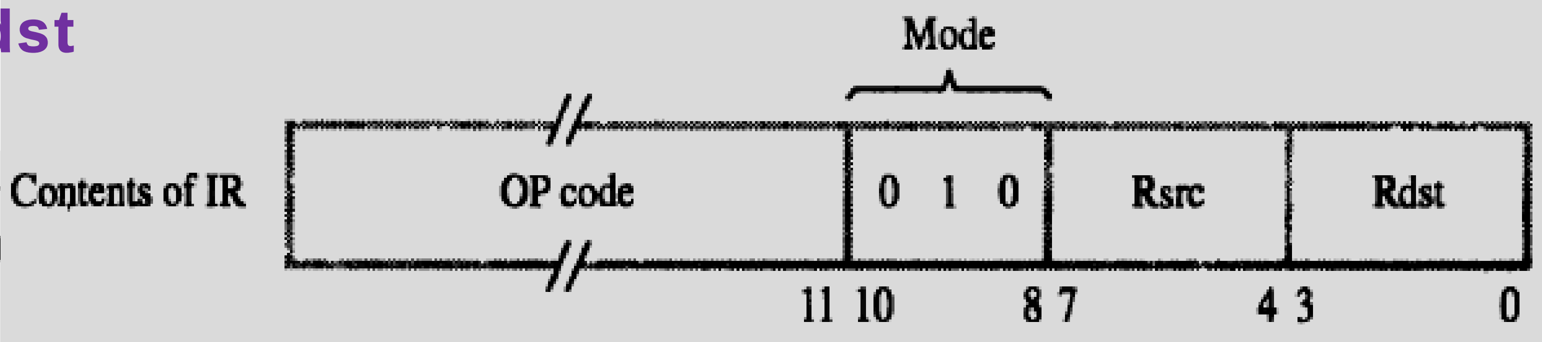
The instruction decoder {InstDec}, generates the starting address of the microroutine that implements the instruction that has just been loaded into the IR.

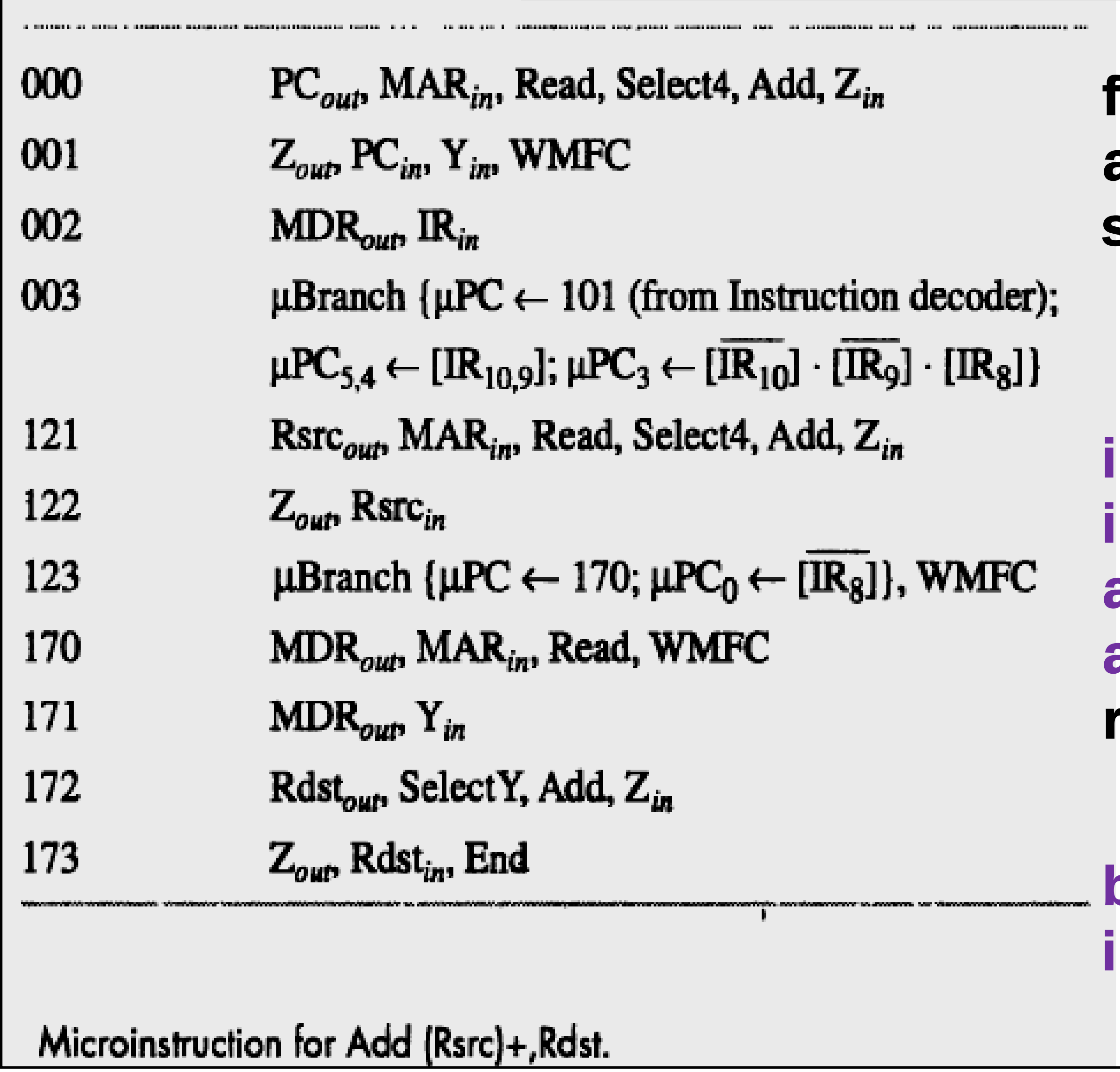
In our example, register IR contains the Add instruction, for which the instruction decoder generates the microinstruction address 101, which cannot be loaded as is into the microprogram counter (μPC).

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 {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.

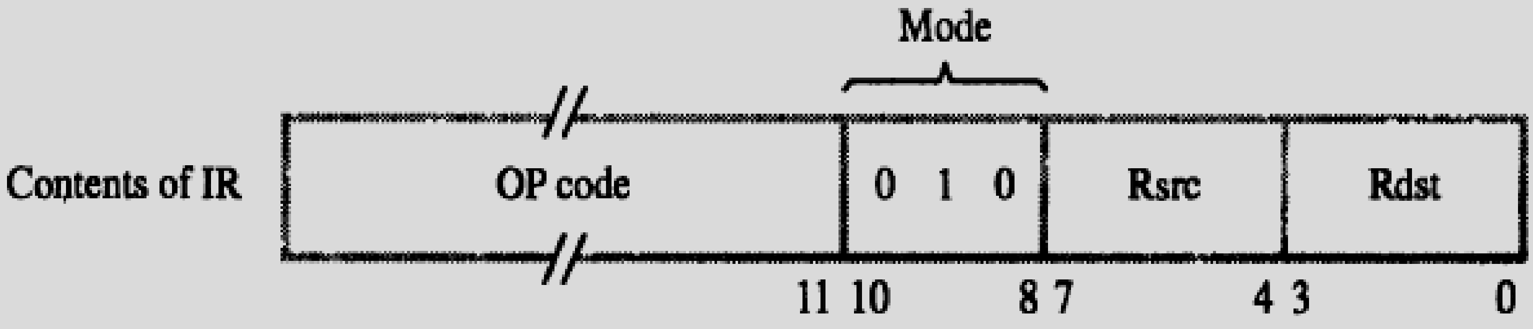
The instruction has a 3-bit field to specify the addressing mode for the source operand, as above.

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)	Microinstructio
000	PC_{out}, MAR_{in}, I
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

Micro-instruction with “next Address Field”.

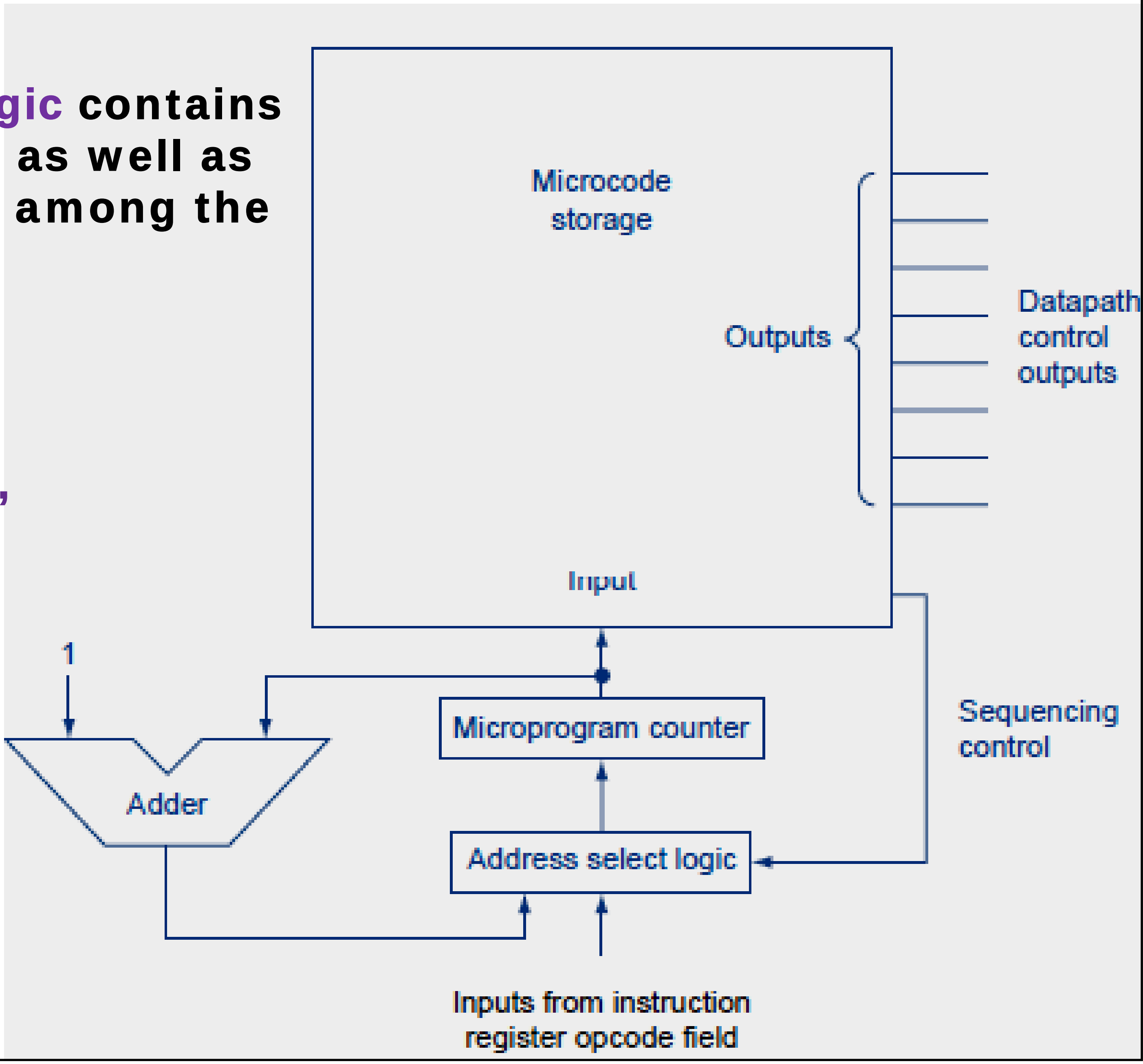
<* For self-study – in END SEM Exam. *>

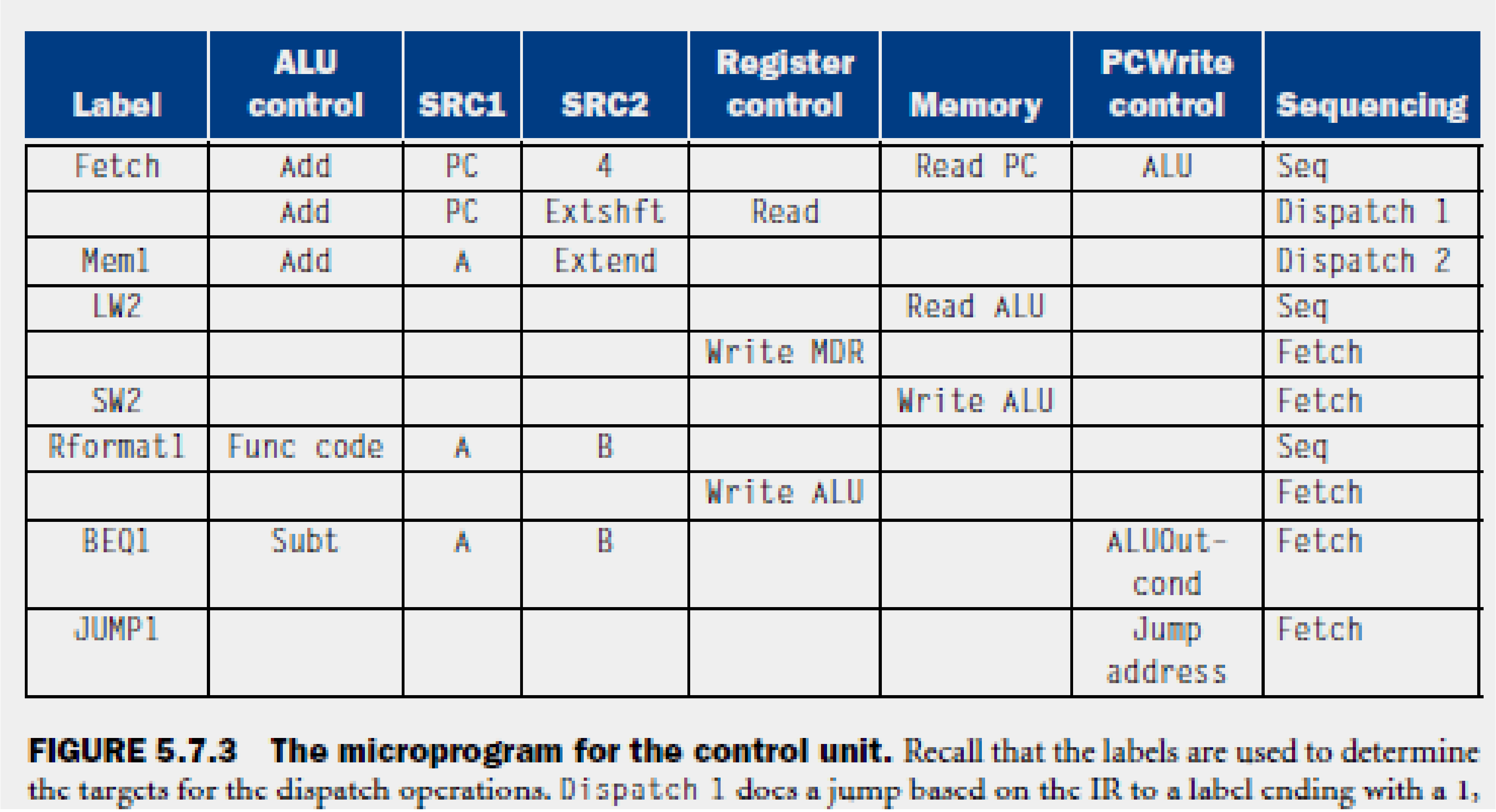
A typical implementation of a microcode controller

The selection of the next microinstruction is controlled by the sequencing control outputs from the control logic.

The address select logic contains a set of dispatch tables as well as the logic to select from among the alternative next states.

The combination of the current microprogram counter, incrementer, dispatch tables, and address select logic forms a sequencer that selects the next microinstruction.

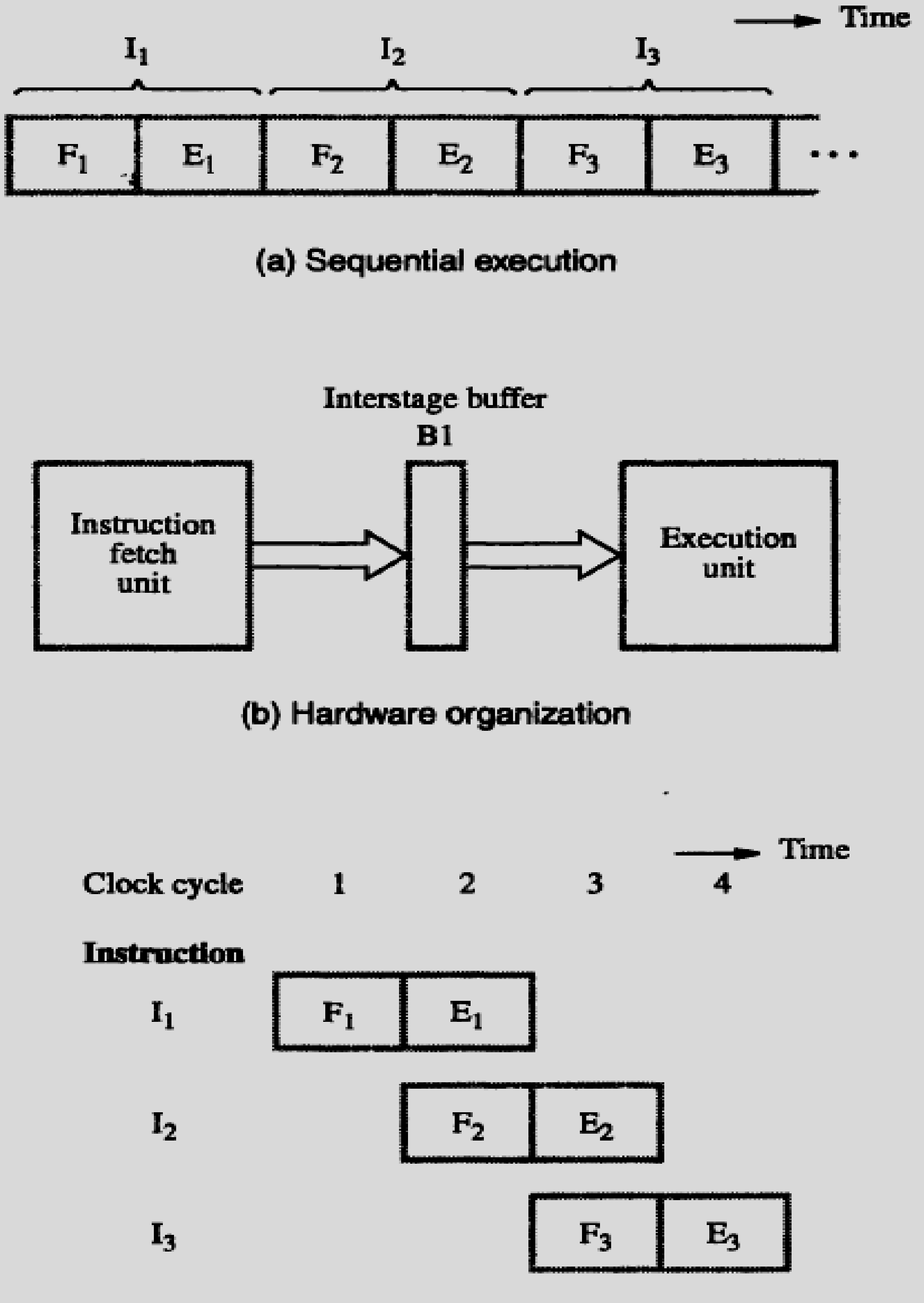




Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1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

FIGURE 5.7.3 The microprogram for the control unit. Recall that the labels are used to determine the targets for the dispatch operations. Dispatch 1 does a jump based on the IR to a label ending with a 1,

PIPELINING



(a) Sequential execution

(b) Hardware organization

(c) Pipelined execution

Hence, 4 units of time slots used;

Compared to 3*2 = 6 units of time required for a Sequential operation.

A pipelined processor may process each instruction in four steps, as follows:

F Fetch: read the instruction from the memory;

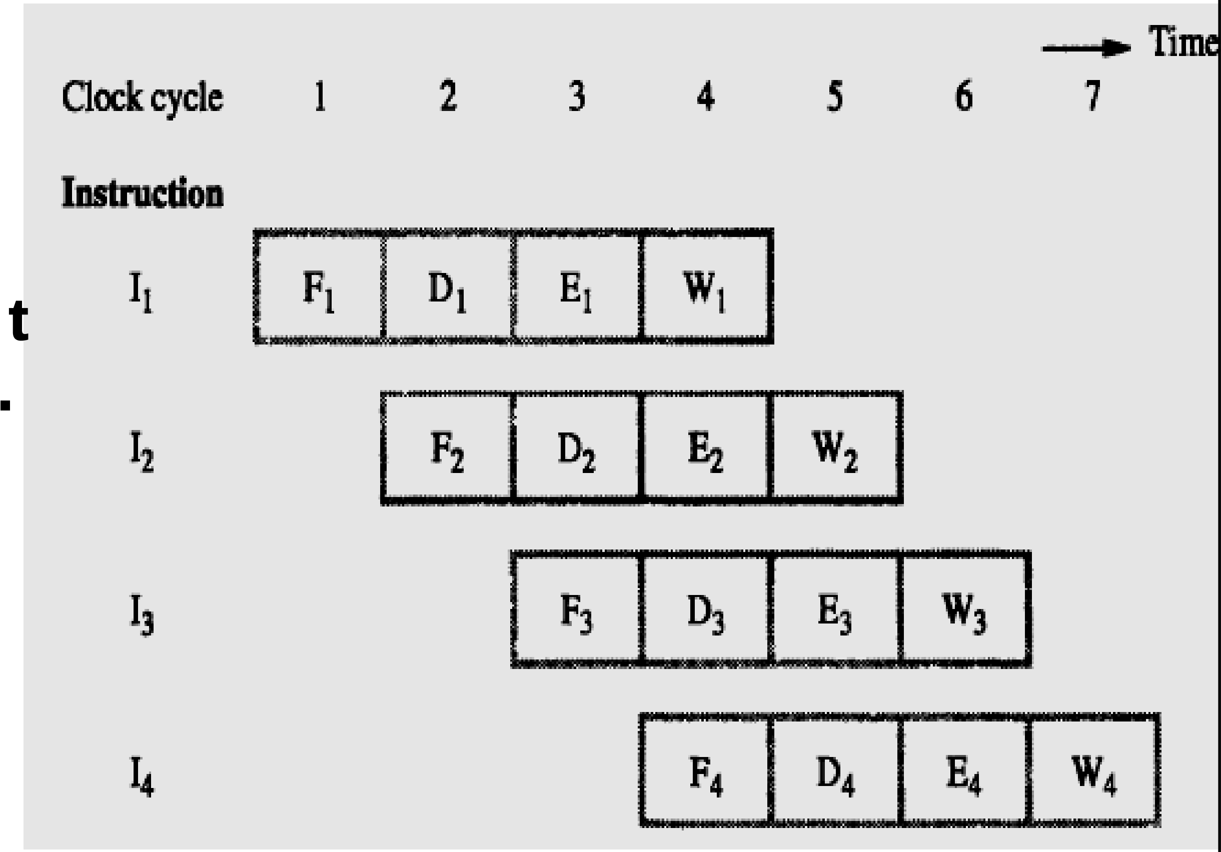
D Decode: decode the instruction and fetch the source operand(s);

E Execute: perform the operation specified by the instruction;

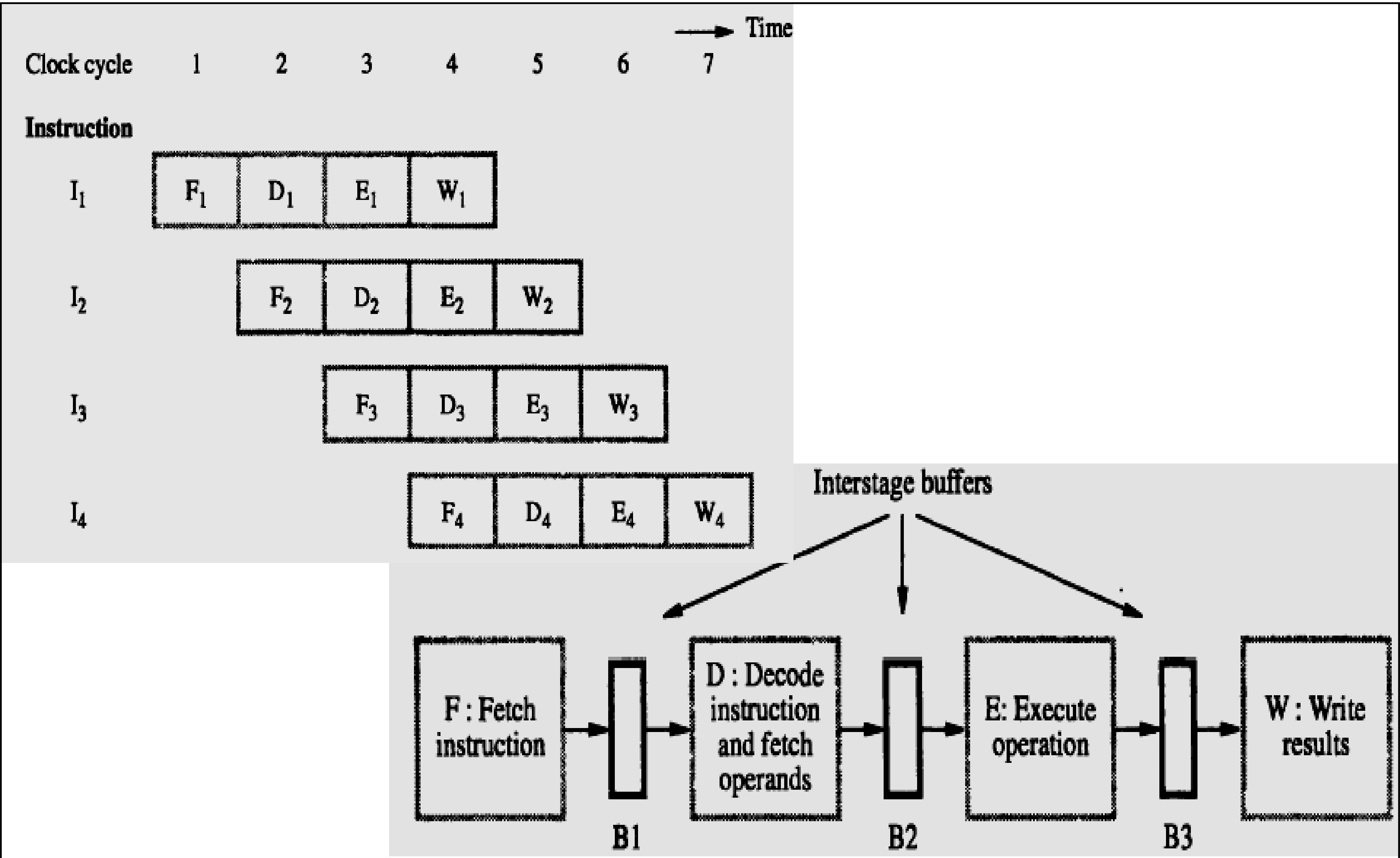
W Write: store the result in the destination location.

Here, 7 units of time slots used;

Compared to 4*4 = 16 units of time required for a Sequential operation.

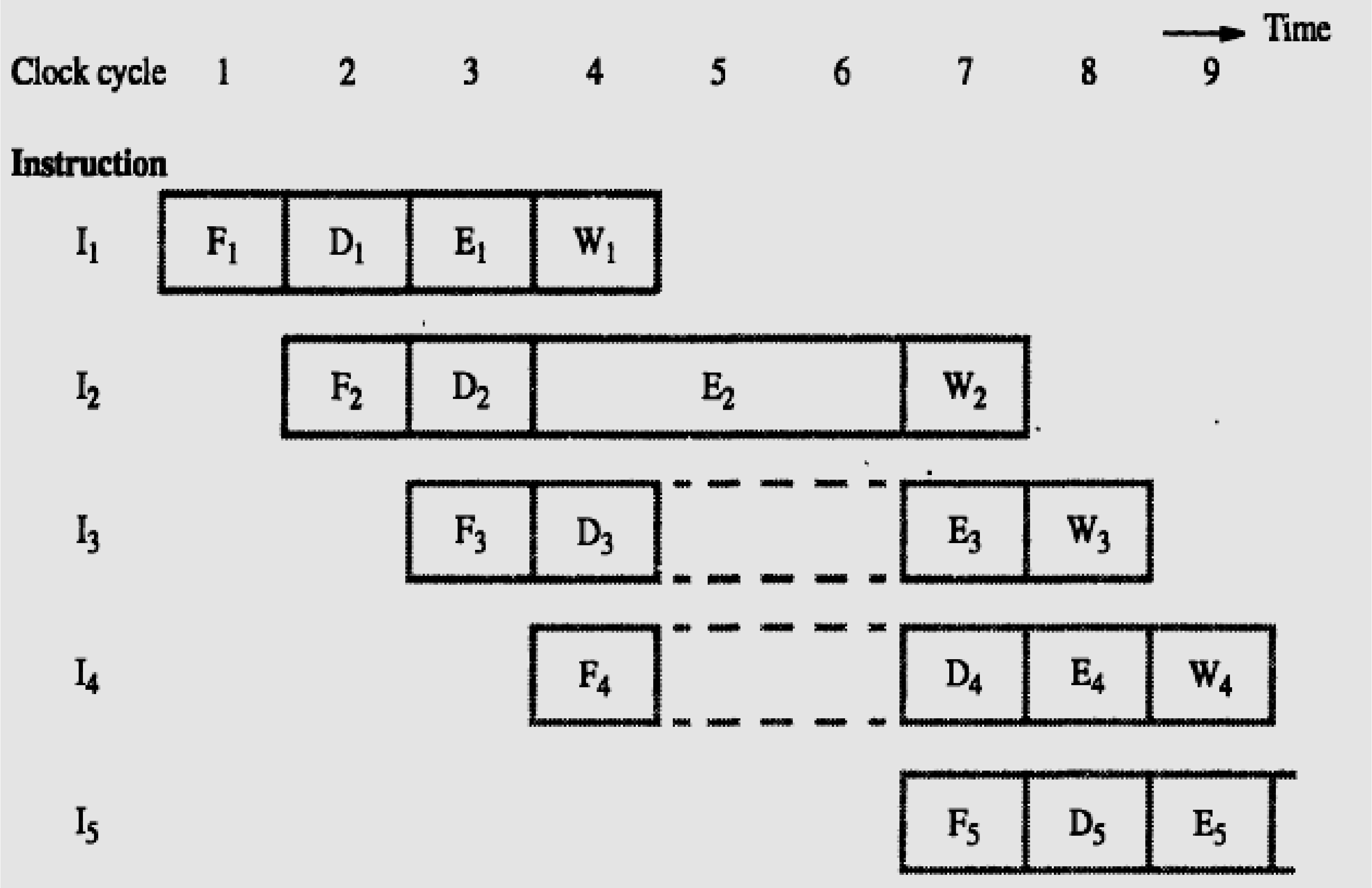


Clock cycle	1	2	3	4	5	6	7
Instruction							
I_1	F_1	D_1	E_1	W_1			
I_2		F_2	D_2	E_2	W_2		
I_3			F_3	D_3	E_3	W_3	
I_4				F_4	D_4	E_4	W_4



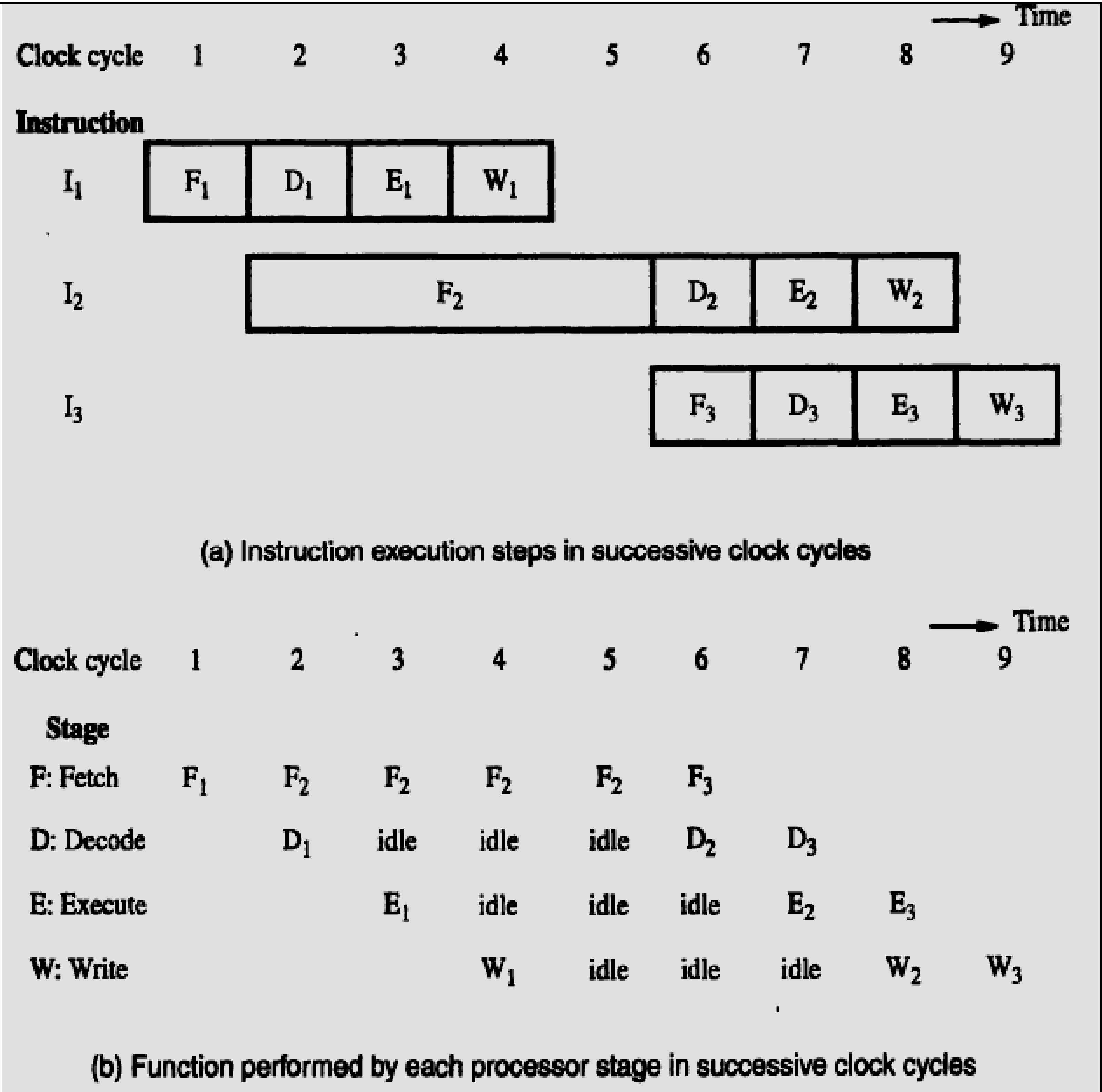
→ Time

Clock cycle	1	2	3	4	5	6	7
Instruction							
I_1	F_1	D_1	E_1	W_1			
I_2		F_2	D_2	E_2	W_2		
I_3			F_3	D_3	E_3	W_3	
I_4				F_4	D_4	E_4	W_4



A Data Hazard, due to delayed EXEC cycle

Instruction or Control Hazard



→ Time

Clock cycle	1	2	3	4	5	6	7	8	9
Instruction									
I_1	F_1	D_1	E_1	W_1					
I_2		F_2				D_2	E_2	W_2	
I_3						F_3	D_3	E_3	W_3

(a) Instruction execution steps in successive clock cycles

→ Time

Clock cycle	1	2	3	4	5	6	7	8	9
Stage									
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_3			
D: Decode		D_1	idle	idle	idle	D_2	D_3		
E: Execute			E_1	idle	idle	idle	E_2	E_3	
W: Write				W_1	idle	idle	idle	W_2	W_3

(b) Function performed by each processor stage in successive clock cycles

An Instruction or Control Hazard, also possible

due to Cache miss in W_MFC

**The Decode unit is idle in cycles 3 through 5,
the Execute unit is idle in cycles 4 through 6,
and the Write unit is idle in cycles 5 through 7.**

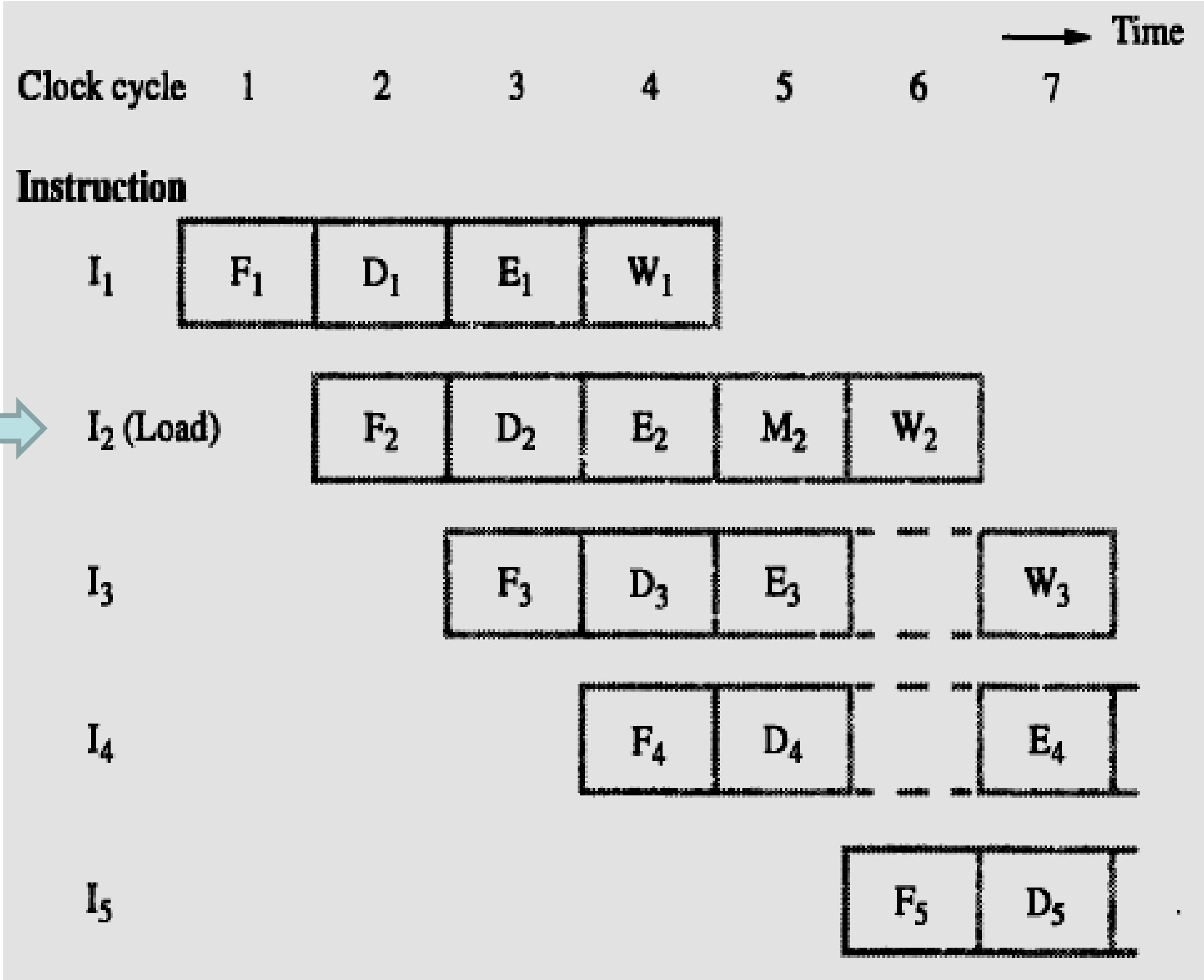
Such idle periods are called *stalls*. *They are also often referred to as bubbles* in the pipeline.

Once created as a result of a delay in one of the pipeline stages, a bubble moves downstream until it reaches the last unit.

A Structural Hazard, also possible

due conflict of usage of the same resource by two or more instructions

Load X(R1),R2



→ Time

Clock cycle	1	2	3	4	5	6	7
Instruction							
I_1	F_1	D_1	E_1	W_1			
I_2 (Load)		F_2	D_2	E_2	M_2	W_2	
I_3			F_3	D_3	E_3		W_3
I_4				F_4	D_4		E_4
I_5						F_5	D_5

Data Hazard, due to concurrent instruction dependencies

$$A \leftarrow 3 + A$$

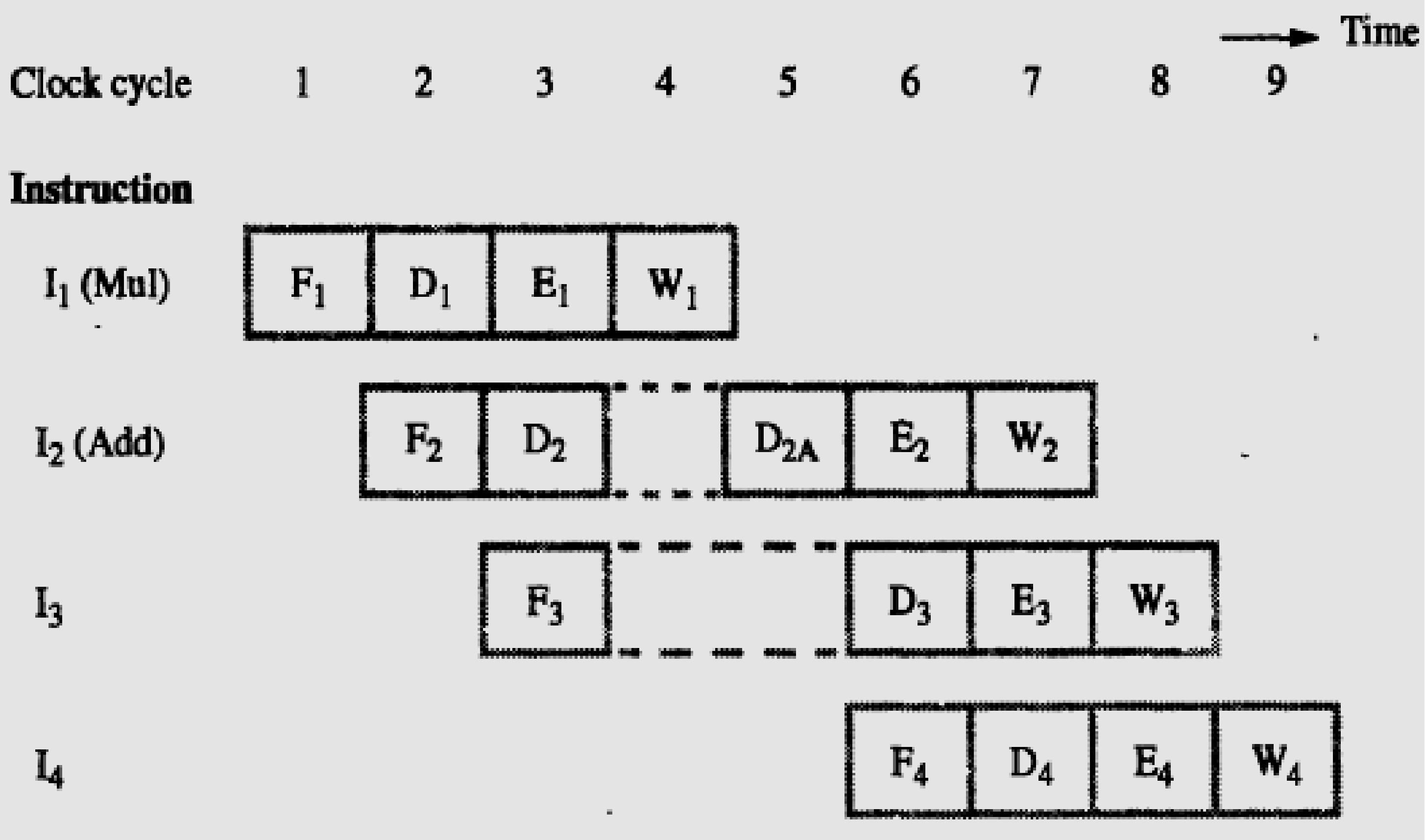
$$B \leftarrow 4 \times A$$

Assume A = 5;
Concurrent execution leads to B = 20 (incorrect)
Sequential execution leads to: B = 32 (Correct)

$$A \leftarrow 5 \times C$$

No problem of concurrency in this case: $B \leftarrow 20 + C$

Mul R2,R3,R4
Add R5,R4,R6



Clock cycle	1	2	3	4	5	6	7	8	9
Instruction									
I_1 (Mul)	F_1	D_1	E_1	W_1					
I_2 (Add)		F_2	D_2		D_{2A}	E_2	W_2		
I_3			F_3			D_3	E_3	W_3	
I_4						F_4	D_4	E_4	W_4

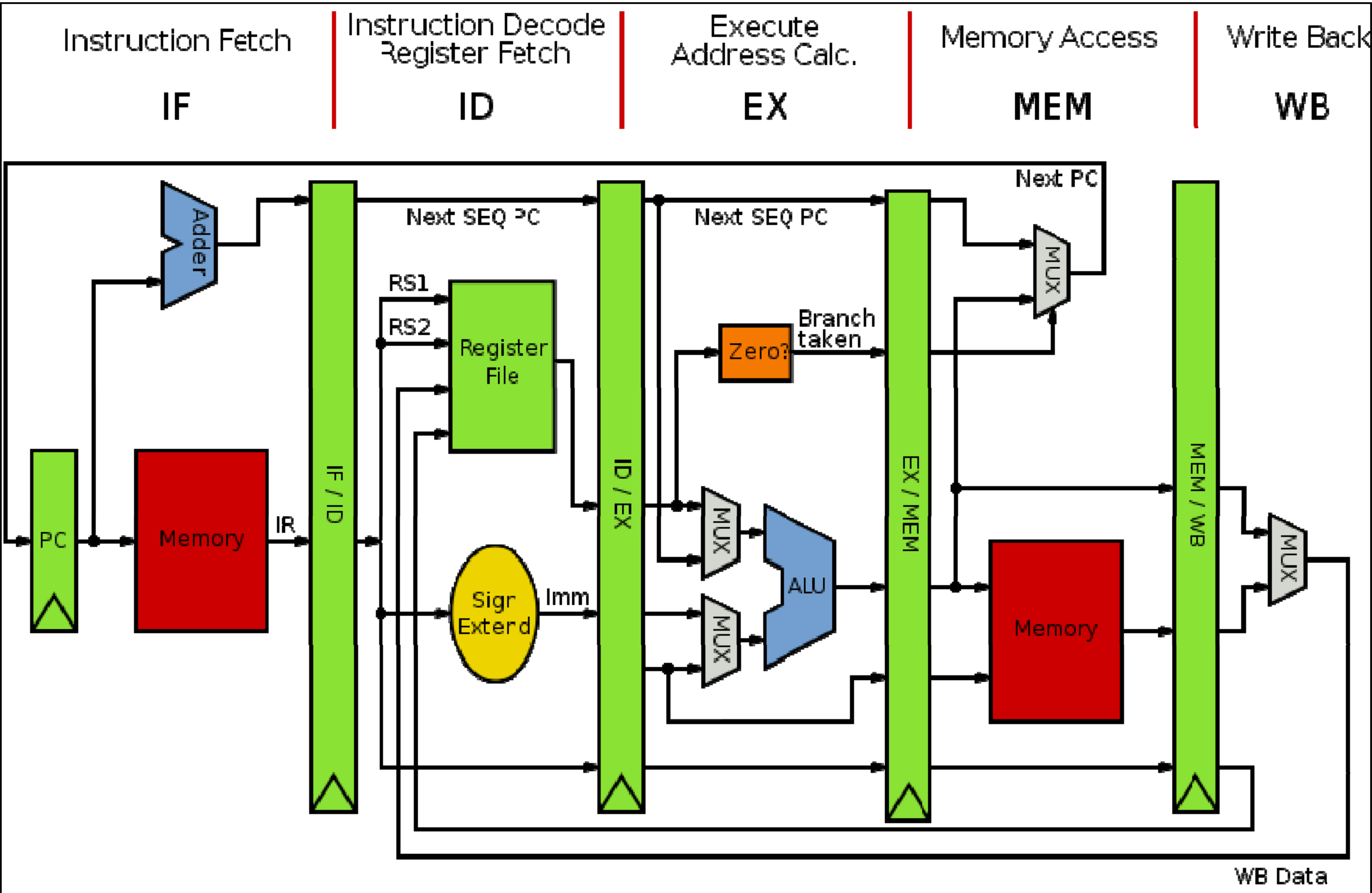
An Instructional Hazard, also possible due to

- Branching

Read:

- Pre-fetching
- Delayed Branch
- Branch Prediction
- Dispatch operation
- Performance (throughput) Gain

- Effect of Addressing modes
- Condition codes
- Datapath and Control
- Superscalar CPU
- Out of order execution



Pipelined MIPS, showing the five stages (instruction fetch, instruction decode, execute, memory access and write back

A superscalar CPU architecture implements a form of parallelism called instruction level parallelism within a single processor. It therefore allows faster CPU throughput than would otherwise be impossible at a given clock rate.

A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor. Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier.

In the Flynn Taxonomy, a superscalar processor is classified as a MIMD processor (Multiple Instructions, Multiple Data).
While a superscalar CPU is typically also pipelined, pipelining and superscalar architecture are considered different performance enhancement techniques.

The superscalar technique is traditionally associated with several identifying characteristics (within a given CPU core):
- Instructions are issued from a sequential instruction stream
- CPU hardware dynamically checks for data dependencies between instructions at run time (versus software checking at compile time)
- The CPU accepts multiple instructions per clock cycle