Basic Computer Organization and Design

Objective

- To introduce a basic computer and show how its operation can be specified with register transfer statements.

Introduction

- The organization of the computer is defined by
  -its internal registers,
  -the timing and control structure, and
  -the set of instructions that it uses.
- The design of computer is then carried out in detail.

Introduction

- The internal organization of a digital system is defined by the sequence of microoperations it performs on data stored in its registers.
- The general purpose digital computer is capable of executing various microoperations and, in addition, can be instructed as to what **specific sequence of operations it must perform.**
- The user of a computer can control the process by means of a **program**.

Program

- A program is a set of instructions that specify the **operations operands**, and the sequence by which processing has to occur.
- The data processing task may be altered by specifying a new program with different instructions or specifying the same instructions with different data.

Stored Program Concept

- A **computer instruction is a binary code** that specifies a sequence of microoperations for the computer.
- Instruction codes together with data are stored in memory.
- The computer reads each instruction from memory and places it in a control register.
- The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of microoperations.
- Every computer has its own unique instruction set.
- The ability to store and execute instructions, **the stored program concept,** is the most important property of a general-purpose computer.

Instruction Code = OPCODE+DATA

- An instruction code is a group of bits that instruct the computer to perform a specific operation.
- It is usually divided into parts, each having its own particular interpretation.

Operation Code (OPCODE)

- The most basic part of an instruction code is its operation part.
- The **operation code** of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement.
- This operation must be performed on some data stored in processor registers or in memory.
- An instruction code must therefore **specify not only the operation but also the registers or the memory words where the operands are to be found**, as well as the register or memory word where the result is to be stored.
- The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer.
- The operation code must consist of at least n bits for a given 2^n (or less) distinct operations.

DATA

- Memory words can be specified in instruction codes by their address.
- Processor registers can be specified by assigning to the instruction another binary code of k bits that specifies one of 2^K registers.
- There are many variations for arranging the binary code of instructions, and **each computer has its own particular instruction code format.**

Instruction Code Format

- There are many variations for arranging the binary code of instructions, and **each computer has its own particular instruction code format.**
- Instruction code formats are conceived by computer designers who specify the architecture of the computer

Computer with 64-Distinct Operations

The operation code consists of six bits: $\mathbf{2^6 = 64}$.

One of them being an ADD operation.

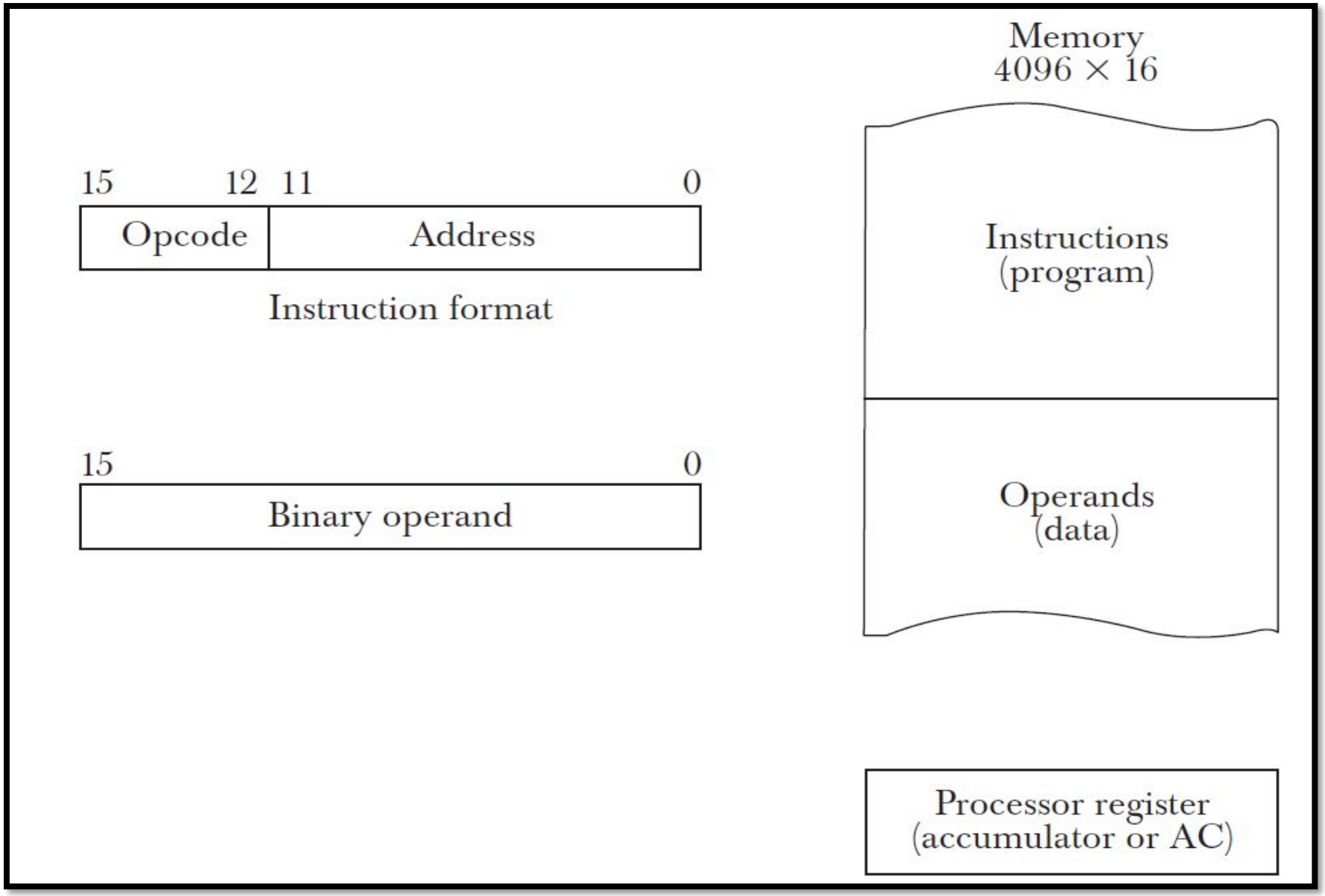
Bit configuration 110010 assigned to the ADD operation.

When this operation code is decoded in the control unit, the computer issues control signals to read an operand from memory and add the operand to a processor register.

Computer Operation (Instruction) Vs Microoperation

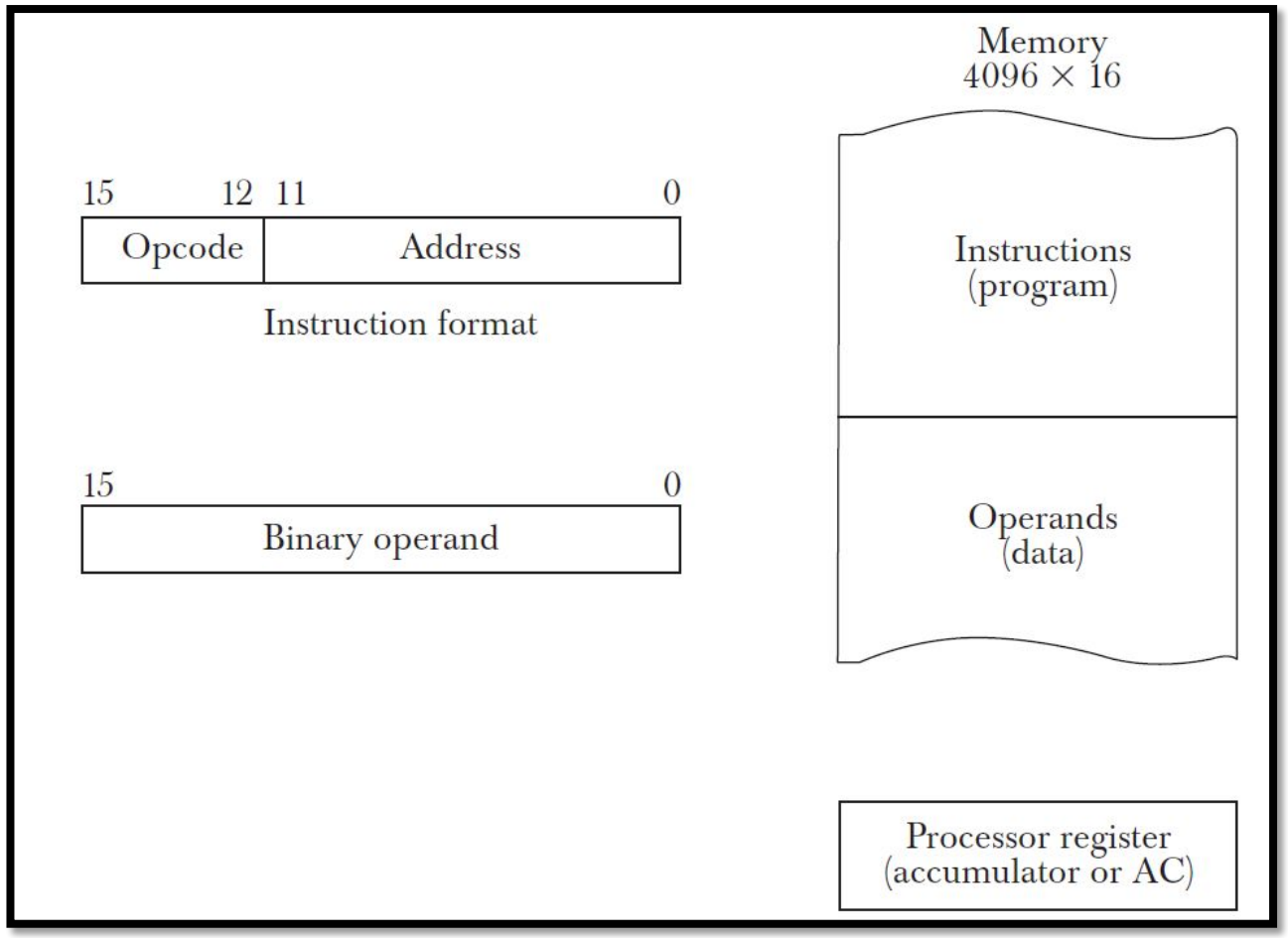
- An operation is part of an instruction stored in computer memory.
- It is a binary code that tells the computer to perform a specific operation.
- The control unit receives the instruction from memory and interprets the operation code bits.
- It then issues a sequence of control signals to initiate microoperations in internal computer registers.
- **For every operation code, the control issues a sequence of microoperations needed for the hardware implementation of the specified operation.**
- For this reason, an operation code is sometimes called a macrooperation because it specifies a set of micro-operations.

Stored Program Organization



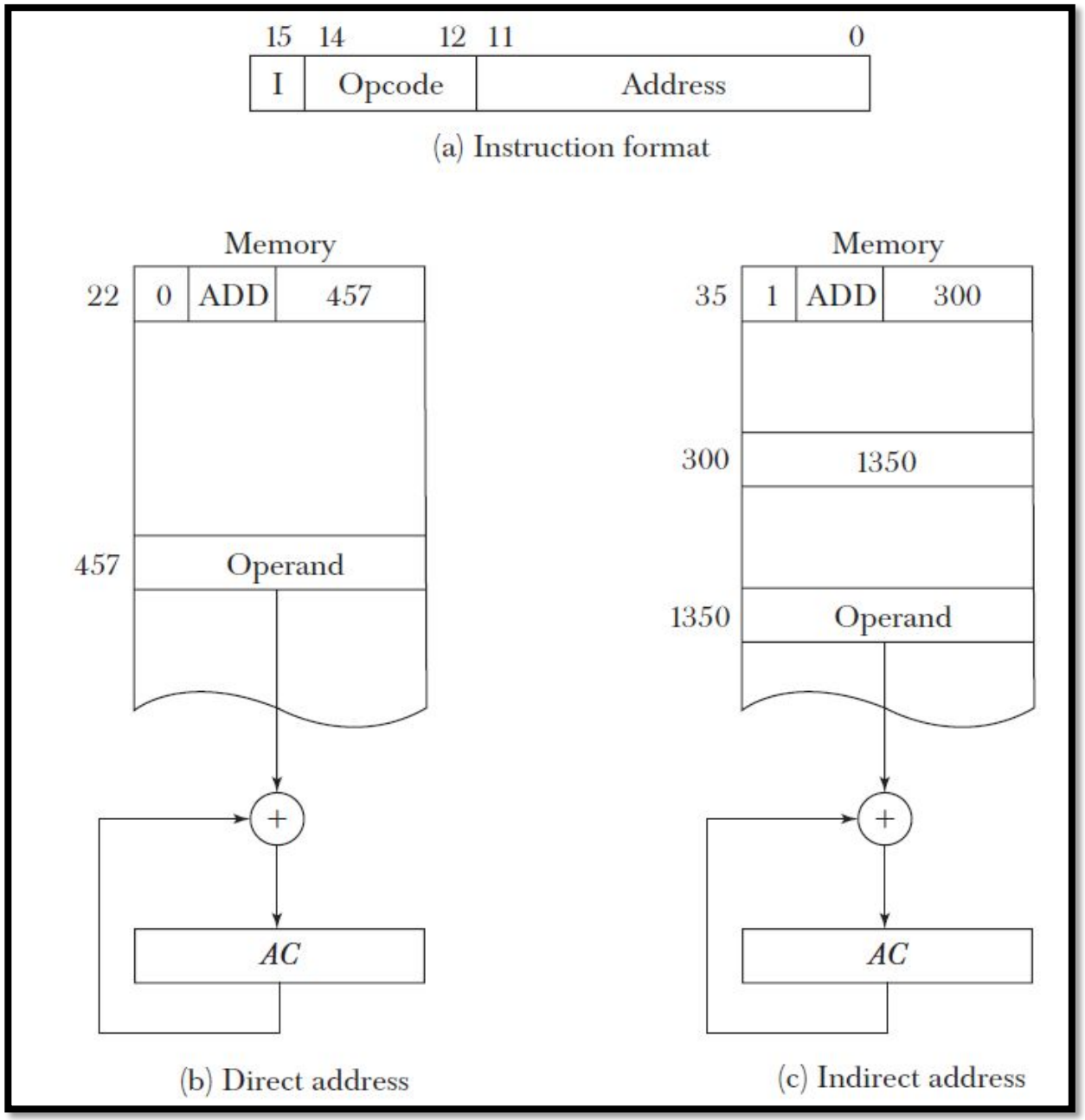
(REFER NOTES)

Immediate Addressing Mode



- It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand.
- When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand.

Direct and Indirect Addressing Mode



When the second part of an instruction code specifies an operand, the instruction is said to have an **immediate operand.**

When the second part specifies the address of an operand, the instruction is said to have a **direct address.**

This is in contrast to a third possibility called **indirect address**, where the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found.

The memory word that holds the address of the operand in an indirect address instruction is used as a pointer to an array of data.

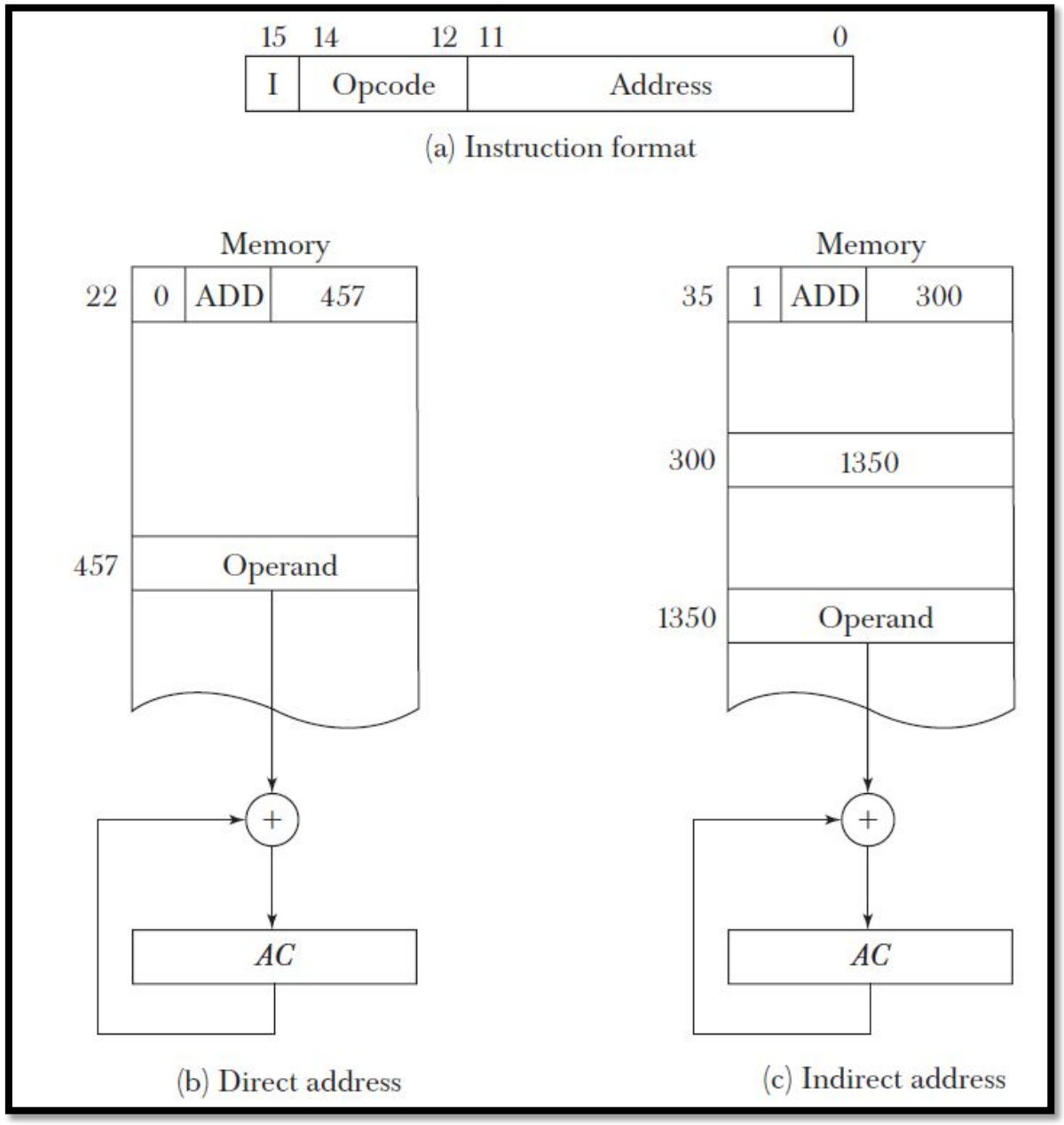
The pointer could be placed in a processor register instead of memory.

One bit of the instruction code can be used to distinguish between a direct and an indirect address.

I- indirect address mode bit

(REFER NOTES)

Effective Address



We define the *effective address* to be the address of the operand in a computation-type instruction or the target address in a branch-type instruction.

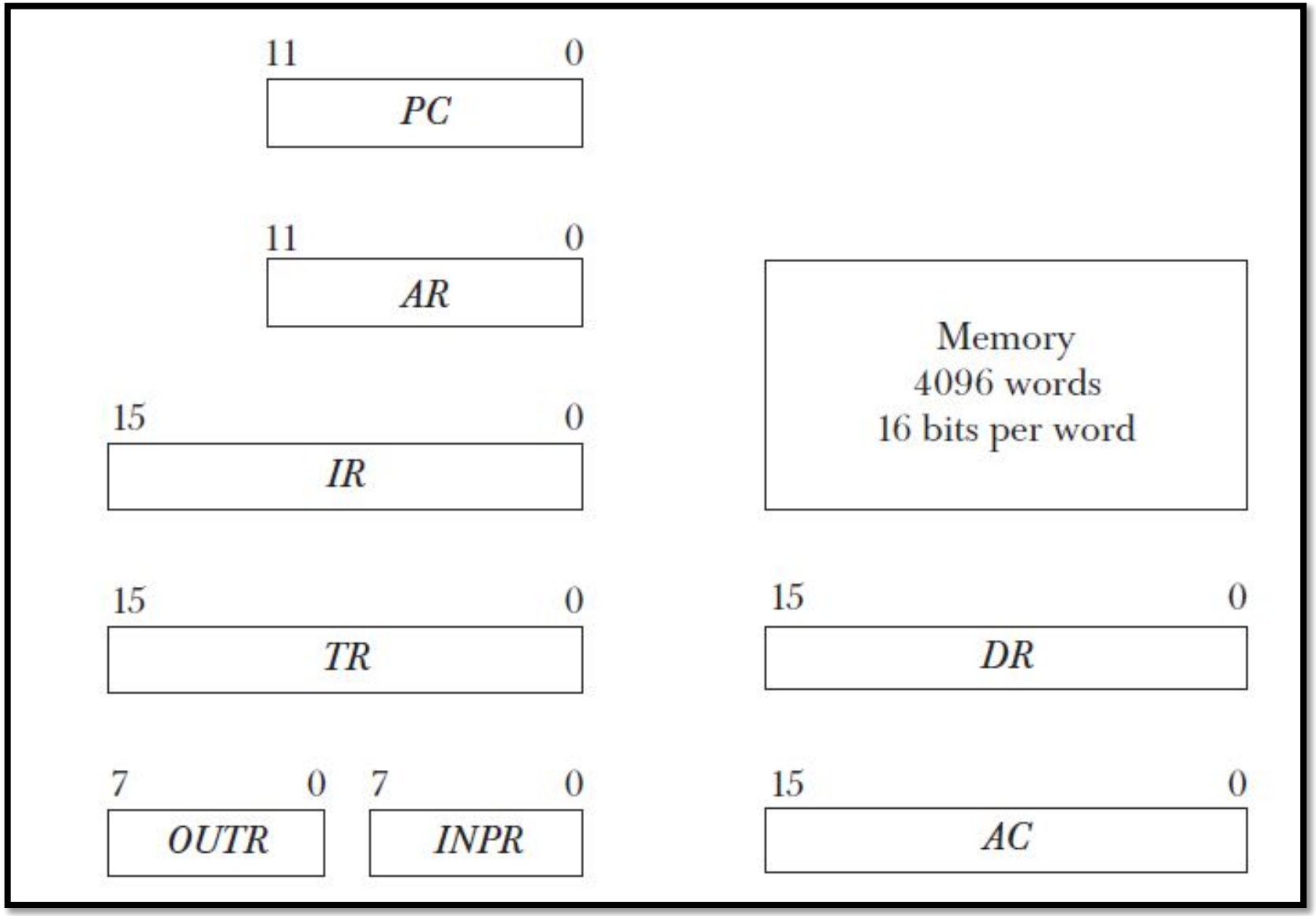
Thus the effective address in the instruction of Figure (b) is 457 and in the instruction of Figure (c) is 1350.

Computer Registers

Introduction

- Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time.
- The control reads an instruction from a specific address in memory and executes it.
- It then continues by reading the next instruction in sequence and executes it, and so on.
- This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed.
- It is also necessary to provide a register in the control unit for storing the instruction code after it is read from memory.
- **The computer needs processor registers for manipulating data and a register for holding a memory address.**

Basic computer registers and memory

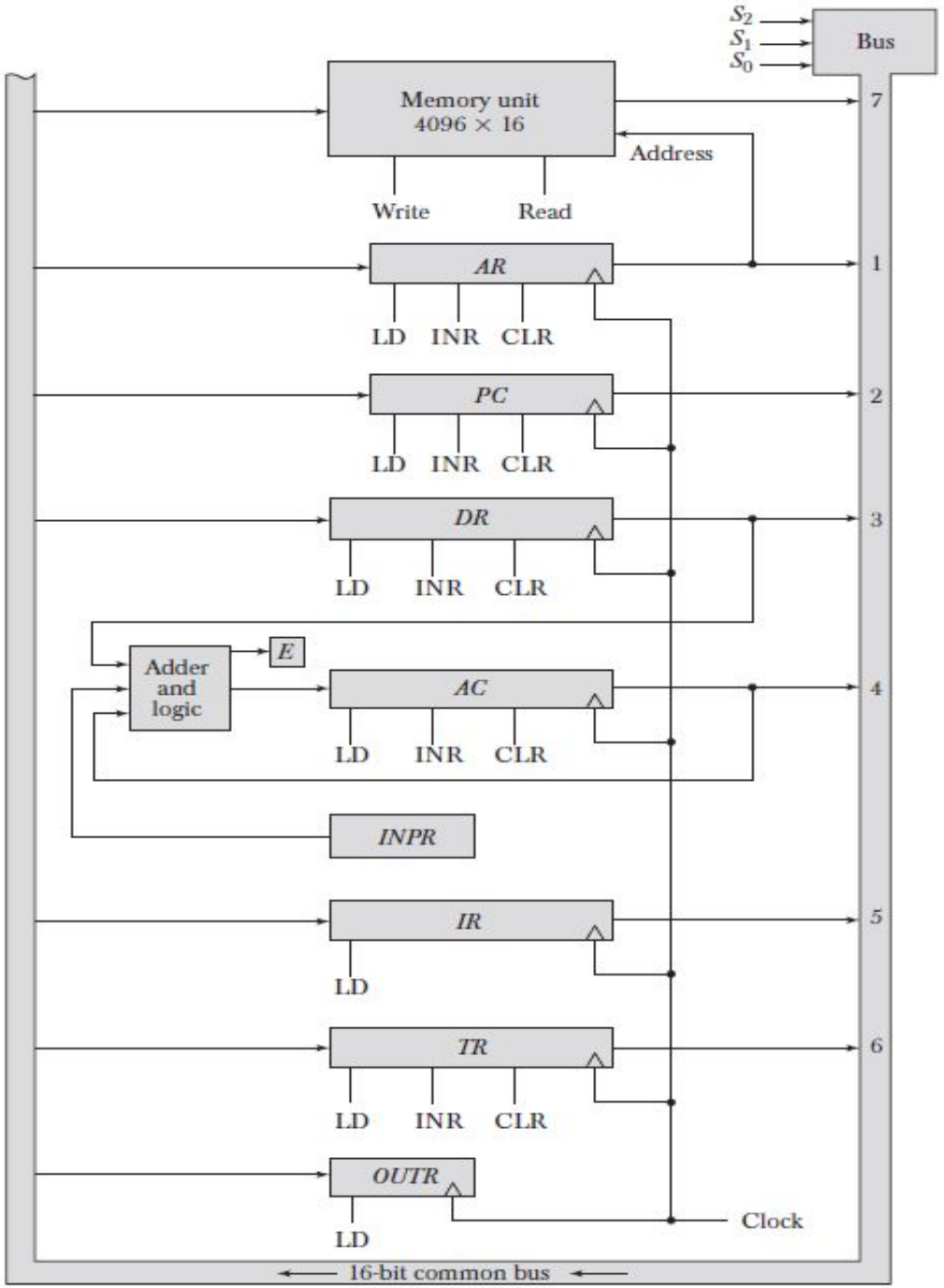


(REFER NOTES)

List of Registers for the Basic Computer

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

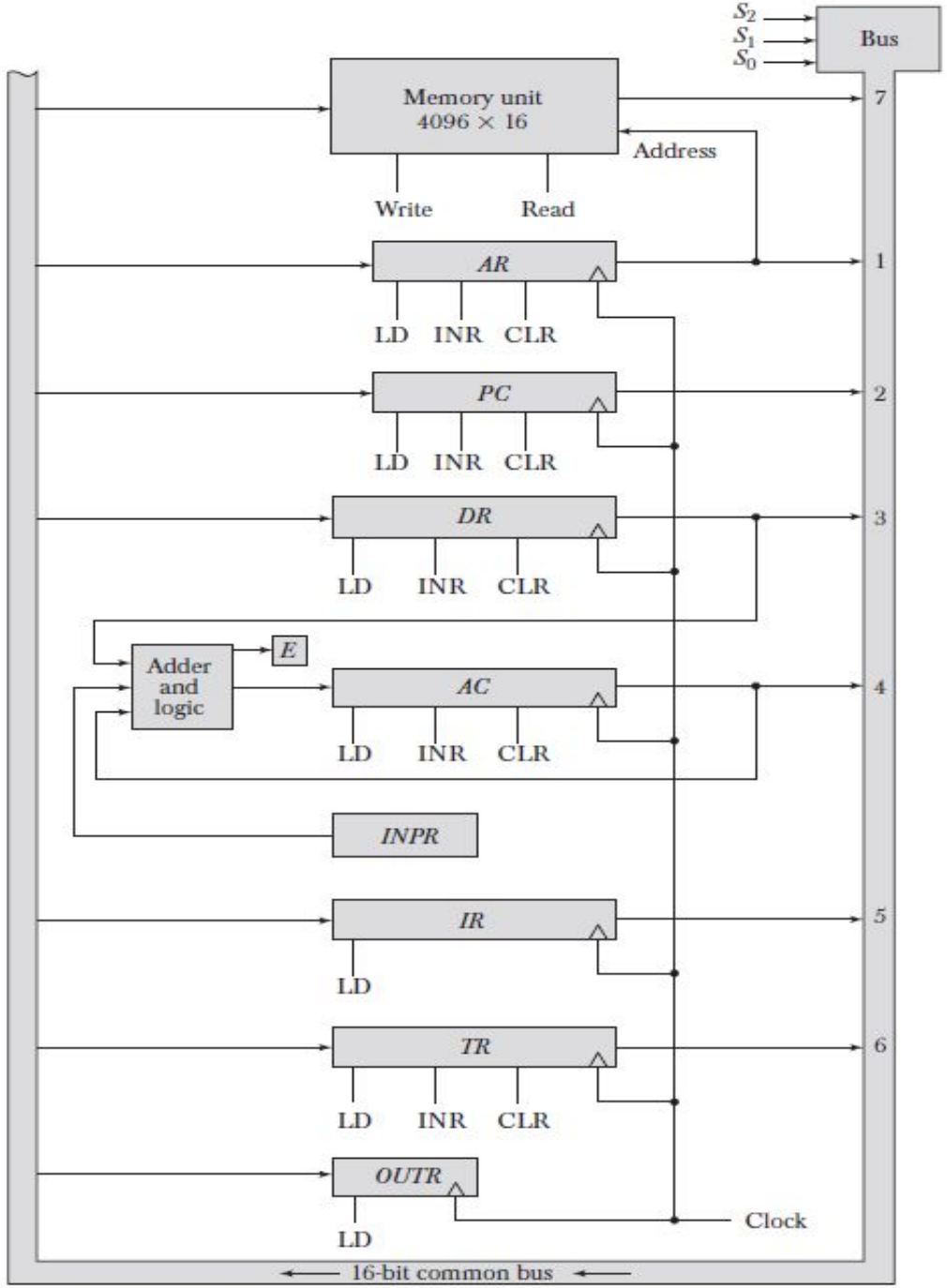
Common Bus System



NOTE:

- *E* (extended *AC* bit).
- LD (load), INR (increment), and CLR (clear).
- This type of register is equivalent to a binary counter with parallel load and synchronous clear.

(REFER NOTES)



Note that the content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle.

The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into *AC*.

For example, the two microoperations

DR ← AC and AC ← DR

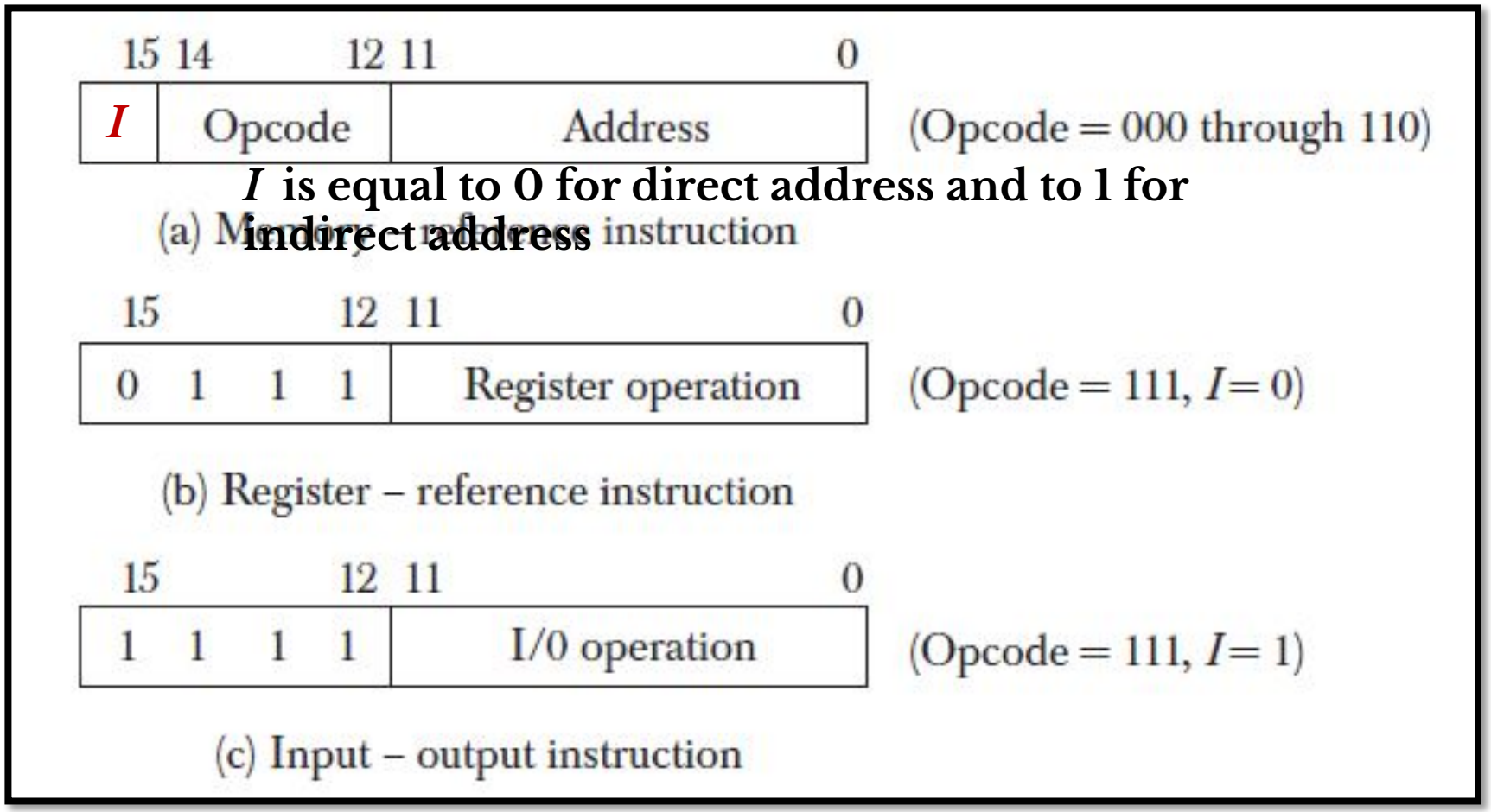
can be executed at the same time.

This can be done by placing the content of *AC* on the bus (with $S_2S_1S_0 = 100$), enabling the LD (load) input of *DR*, transferring the content of DR through the adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle.

The two transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle.

Computer Instructions

Basic computer instruction formats



I **is equal to 0 for direct address and to 1 for indirect address**

Basic Computer Instructions

The total number of instructions chosen for the basic computer is equal to 25.

Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	Description
AND	0xxx	8xxx	AND memory word to *AC*
ADD	1xxx	9xxx	Add memory word to *AC*
LDA	2xxx	Axxx	Load memory word to *AC*
STA	3xxx	Bxxx	Store content of *AC* in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero

Basic Computer Instructions

CLA	7800	Clear *AC*
CLE	7400	Clear *E*
CMA	7200	Complement *AC*
CME	7100	Complement *E*
CIR	7080	Circulate right *AC* and *E*
CIL	7040	Circulate left *AC* and *E*
INC	7020	Increment *AC*
SPA	7010	Skip next instruction if *AC* positive
SNA	7008	Skip next instruction if *AC* negative
SZA	7004	Skip next instruction if *AC* zero
SZE	7002	Skip next instruction if *E* is 0
HLT	7001	Halt computer

Basic Computer Instructions

INP	F800	Input character to *AC*
OUT	F400	Output character from *AC*
SKI	F200	Skip on input flag
SKO	F100	Skip on output flag
ION	F080	Interrupt on
IOF	F040	Interrupt off

Instruction Set

- A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable.

The **set of instructions are said to be complete** if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions
2. Instructions for moving information to and from memory and processor registers
3. Program control instructions together with instructions that check status conditions
4. Input and output instructions

(REFER NOTES)

Instructions (Explanation)

- There is one arithmetic instruction, ADD, and two related instructions, complement AC(CMA) and increment AC(INC).
- With these three instructions we can add and subtract binary numbers when negative numbers are in signed-2's complement representation.
- The circulate instructions, CIR and CIL, can be used for arithmetic shifts as well as any other type of shifts desired.
- Multiplication and division can be performed using addition, subtraction, and shifting.
- There are three logic operations: AND, complement AC(CMA), and clear AC(CLA).
- The AND and complement provide a NAND operation. It can be shown that with the NAND operation it is possible to implement all the other logic operations with two variables.
- Moving information from memory to AC is accomplished with the load AC(LDA) instruction.
- Storing information from AC into memory is done with the store AC(STA) instruction.
- The branch instructions BUN, BSA, and ISZ, together with the four skip instructions, provide capabilities for program control and checking of status conditions.
- The input (INP) and output (OUT) instructions cause information to be transferred between the computer and external devices.

Note

- Although the set of instructions for the basic computer is complete, it is not efficient because frequently used operations are not performed rapidly.
- An efficient set of instructions will include such instructions as subtract, multiply, OR, and exclusive-OR.
- These operations must be programmed in the basic computer.

Timing and Control

Timing and Control

- The timing for all registers in the basic computer is controlled by a master clock generator.
- The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit.
- The clock pulses do not change the state of a register unless the register is enabled by a control signal.
- The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

Types of Control Organization

There are two major types of control organization:

1. Hardwired control.
2. Microprogrammed control.

Hardwired control

- In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits.
- It has the advantage that it can be optimized to produce a fast mode of operation.

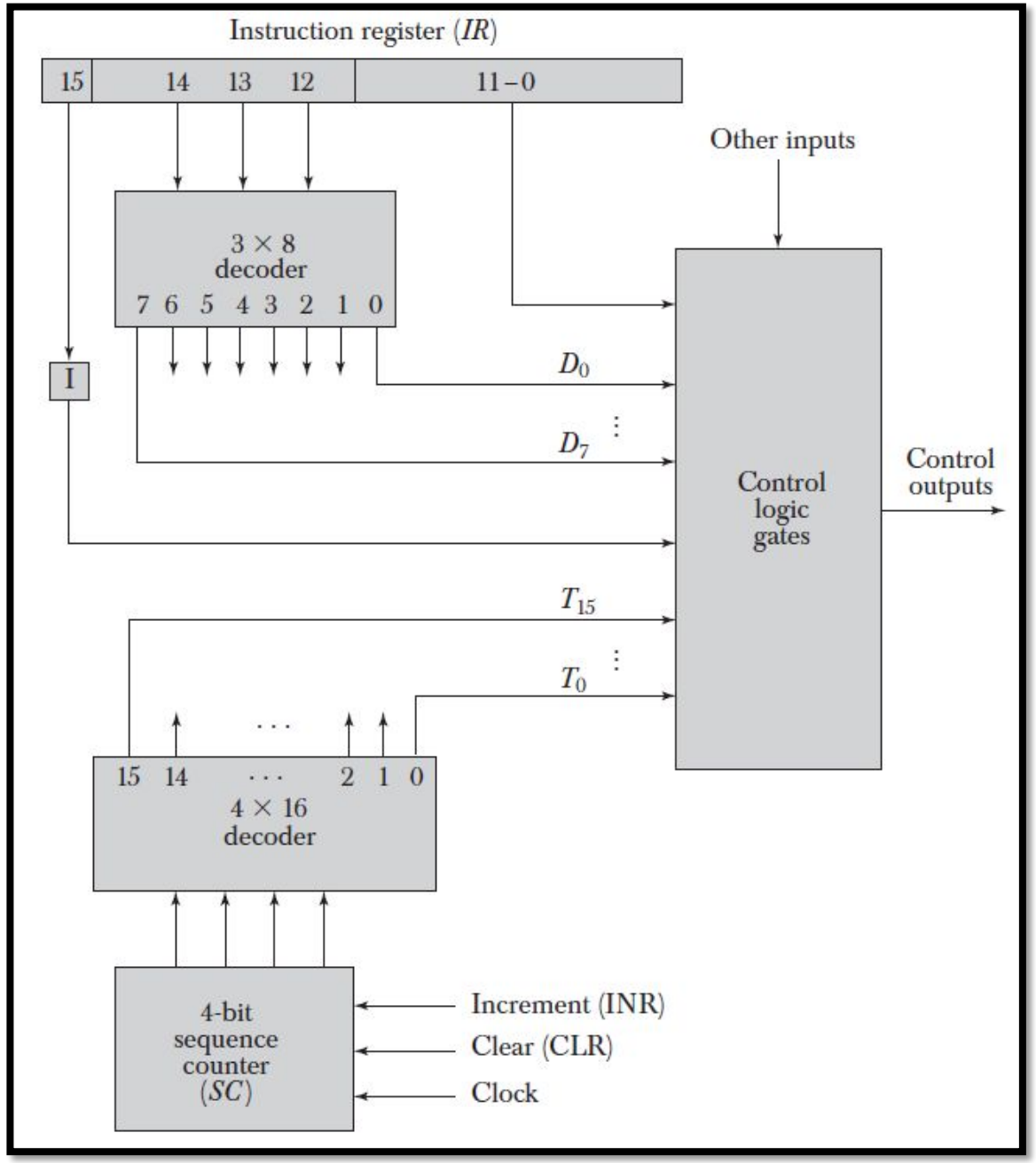
Microprogrammed control

- In the microprogrammed organization, the control information is stored in a control memory.
- The control memory is programmed to initiate the required sequence of microoperations.

Hardwired Vs Microprogrammed

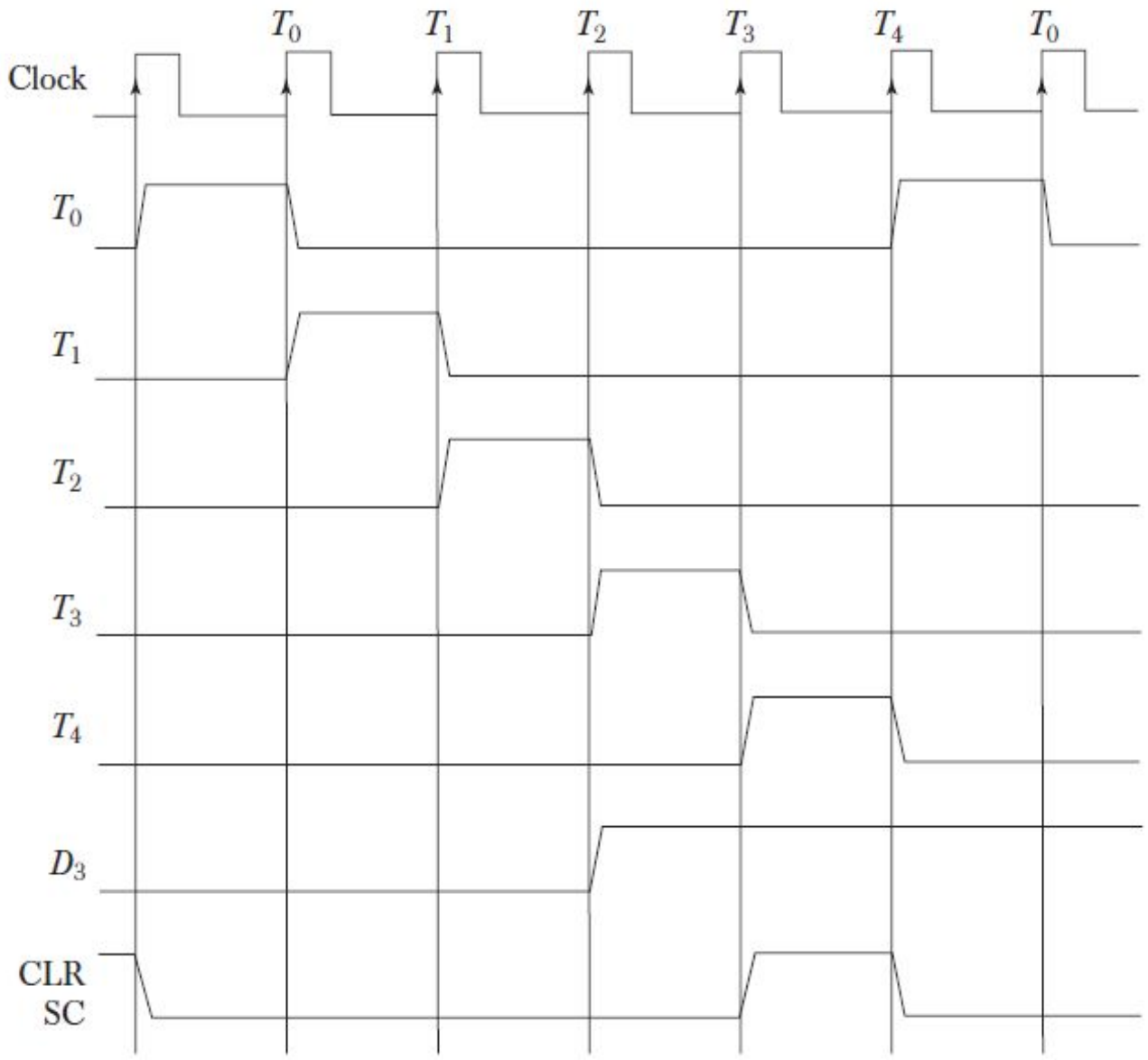
- A hardwired control, as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed.
- In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory.

Control unit of basic computer (A hardwired control)



(REFER NOTES)

Example of control timing signals



Consider the case where *SC* is incremented to provide timing signals T_0, T_1, T_2, T_3, and T_4 in sequence.

At time T_4, *SC* is cleared to 0 if decoder output D_3 is active.

This is expressed symbolically by the statement

D_3T_4: SC ← 0 *SC* is cleared when $D_3T_4 = 1$.

The sequence counter *SC* responds to the positive transition of the clock.

Initially, the CLR input of *SC* is active.

The first positive transition of the clock clears *SC* to 0, which in turn activates the timing signal T_0 out of the decoder.

T_0 is active during one clock cycle.

The positive clock transition labeled T_0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T_0.

SC is incremented with every positive clock transition, unless its CLR input is active.

This produces the sequence of timing signals T_0, T_1, T_2, T_3, T_4 , and so on, as shown in the diagram.

NOTE

- **A memory read or write cycle will be initiated with the rising edge of a timing signal.**
- **It will be assumed that a memory cycle time is less than the clock cycle time.**
- According to this assumption, a memory read or write cycle initiated by a timing signal will be completed by the time the next clock goes through its positive transition.
- The clock transition will then be used to load the memory word into a register.
- This timing relationship is not valid in many computers because the memory cycle time is usually longer than the processor clock cycle.
- In such a case it is necessary to provide wait cycles in the processor until the memory word is available.
- We will assume that a wait period is not necessary in the basic computer.

Timing relationship between the clock transition and the timing signals

- The register transfer statement

$$T_0: \text{AR} \leftarrow \text{PC}$$

specifies a **transfer of the content of PC into AR if timing signal T_0 is active**. T_0 is active during an entire clock cycle interval.

- During this time the content of PC is placed onto the bus (with $S_2S_1S_0 = 010$) and the LD (load) input of AR is enabled.
- The actual transfer does not occur until the end of the clock cycle when the clock goes through a positive transition.
- This same positive clock transition increments the sequence counter SC from 0000 to 0001.
- The next clock cycle has T_1 active and T_0 inactive.

Instruction Cycle
(Instruction Execution Cycle)

Instruction Cycle

- A program residing in the memory unit of the computer consists of a sequence of instructions.
- The program is executed in the computer by going through a cycle for each instruction.
- Each instruction cycle in turn is subdivided into a sequence of sub cycles or phases.
- **In the basic computer each instruction cycle consists of the following phases:**

1. Fetch an instruction from memory.

2. Decode the instruction.

3. Read the effective address from memory if the instruction has an indirect address.

4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction.

This process continues indefinitely unless a HALT instruction is encountered.

Fetch and Decode

- Initially, the program counter *PC* is loaded with the address of the first instruction in the program.
- The sequence counter *SC* is cleared to 0, providing a decoded timing signal T_0.
- After each clock pulse, *SC* is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2, and so on.
- The microoperations for the fetch and decode phases can be specified by the following register transfer statements:

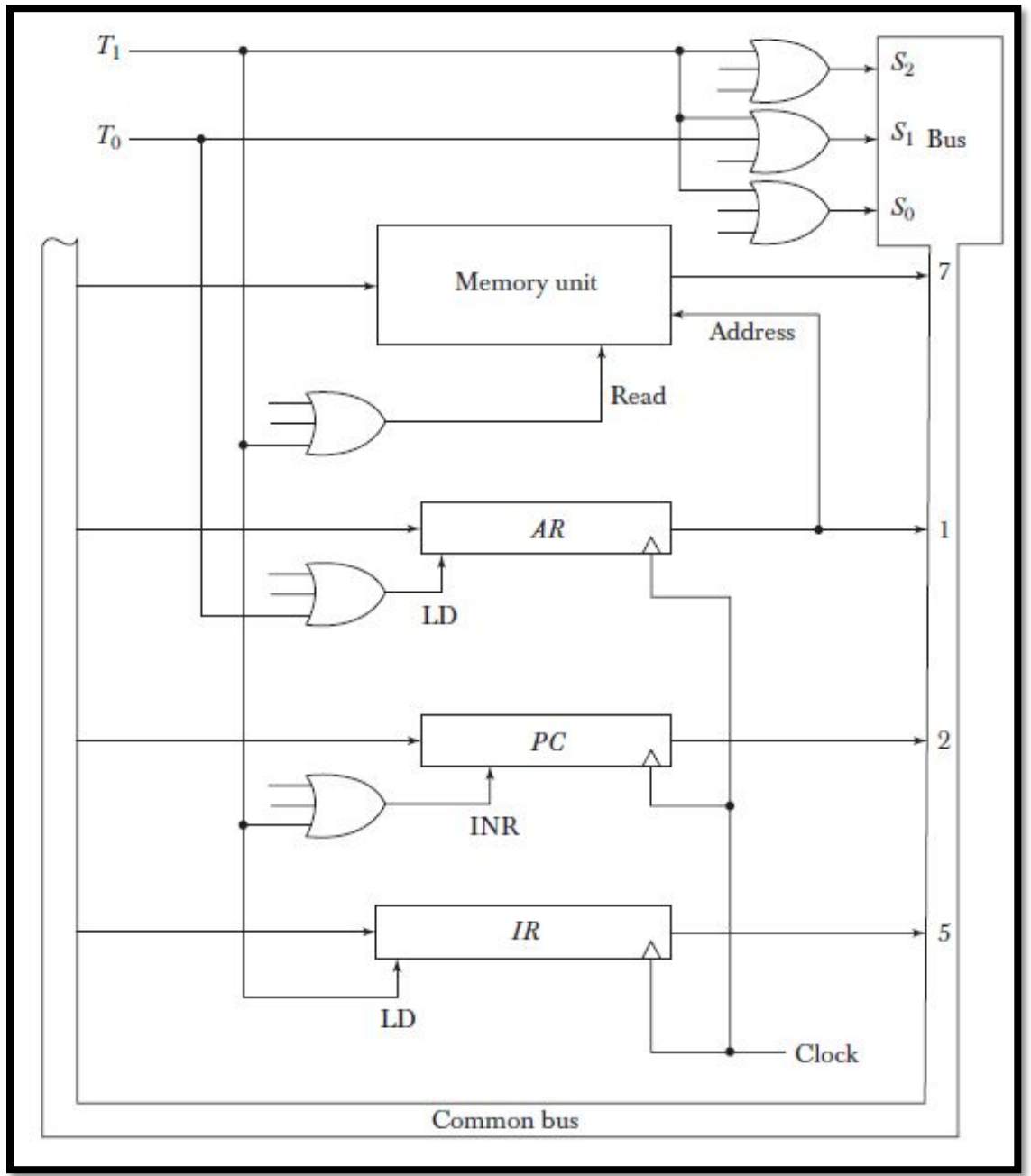
$$T_0: \quad AR \leftarrow PC$$

$$T_1: \quad IR \leftarrow M[AR],\ PC \leftarrow PC + 1$$

$$T_2: \quad D_0, \ldots, D_7 \leftarrow \text{Decode } IR(12\text{–}14),\ AR \leftarrow IR(0-11),\ I \leftarrow IR(15)$$

(REFER NOTES)

Register transfers for the fetch phase

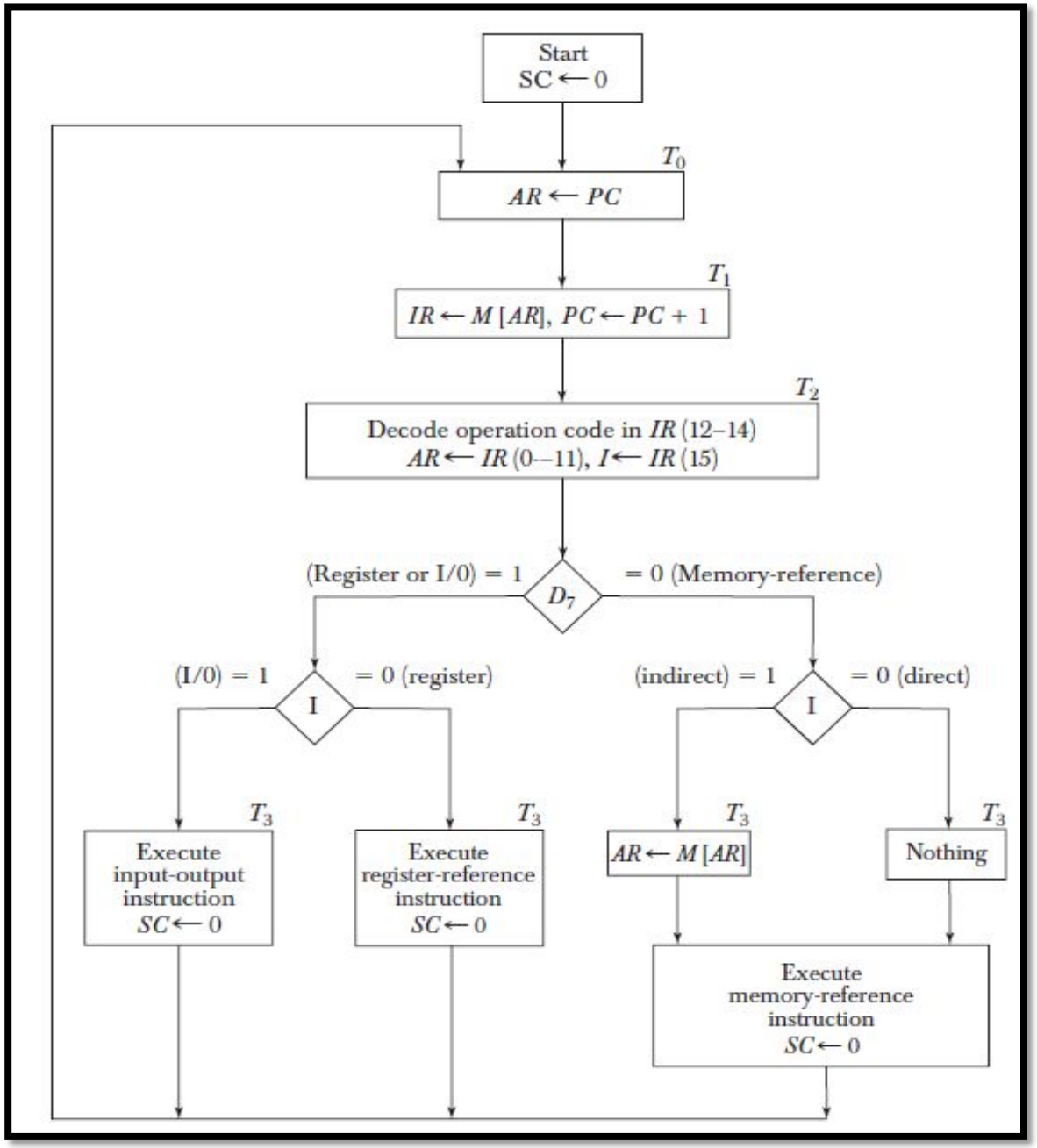


T_0: $AR \leftarrow PC$

T_1: $IR \leftarrow M[AR],\ PC \leftarrow PC + 1$

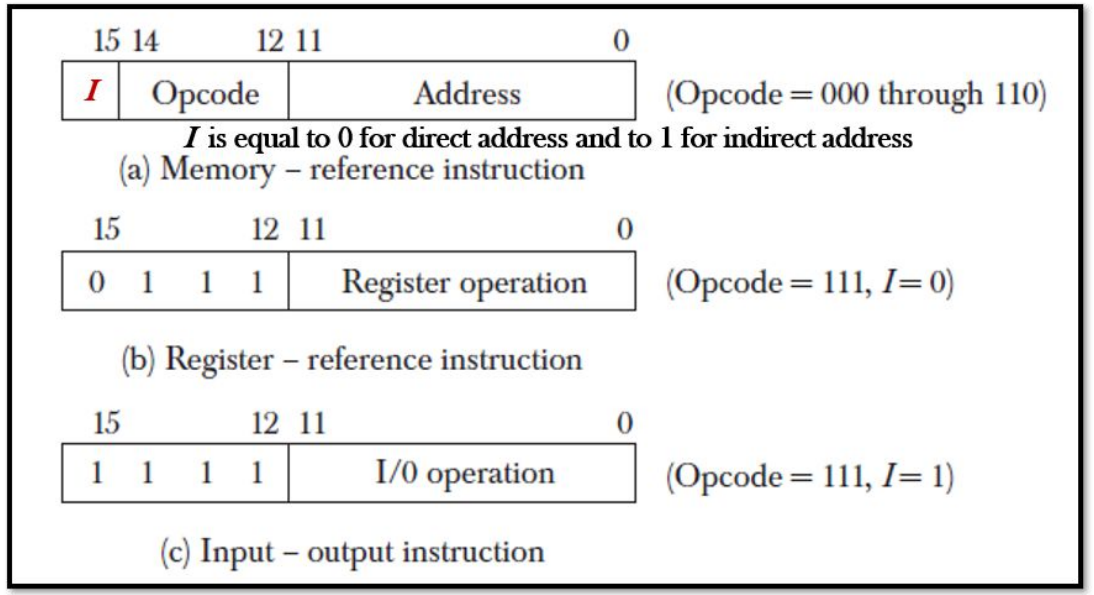
(REFER NOTES)

Flowchart for instruction cycle (Initial Configuration)

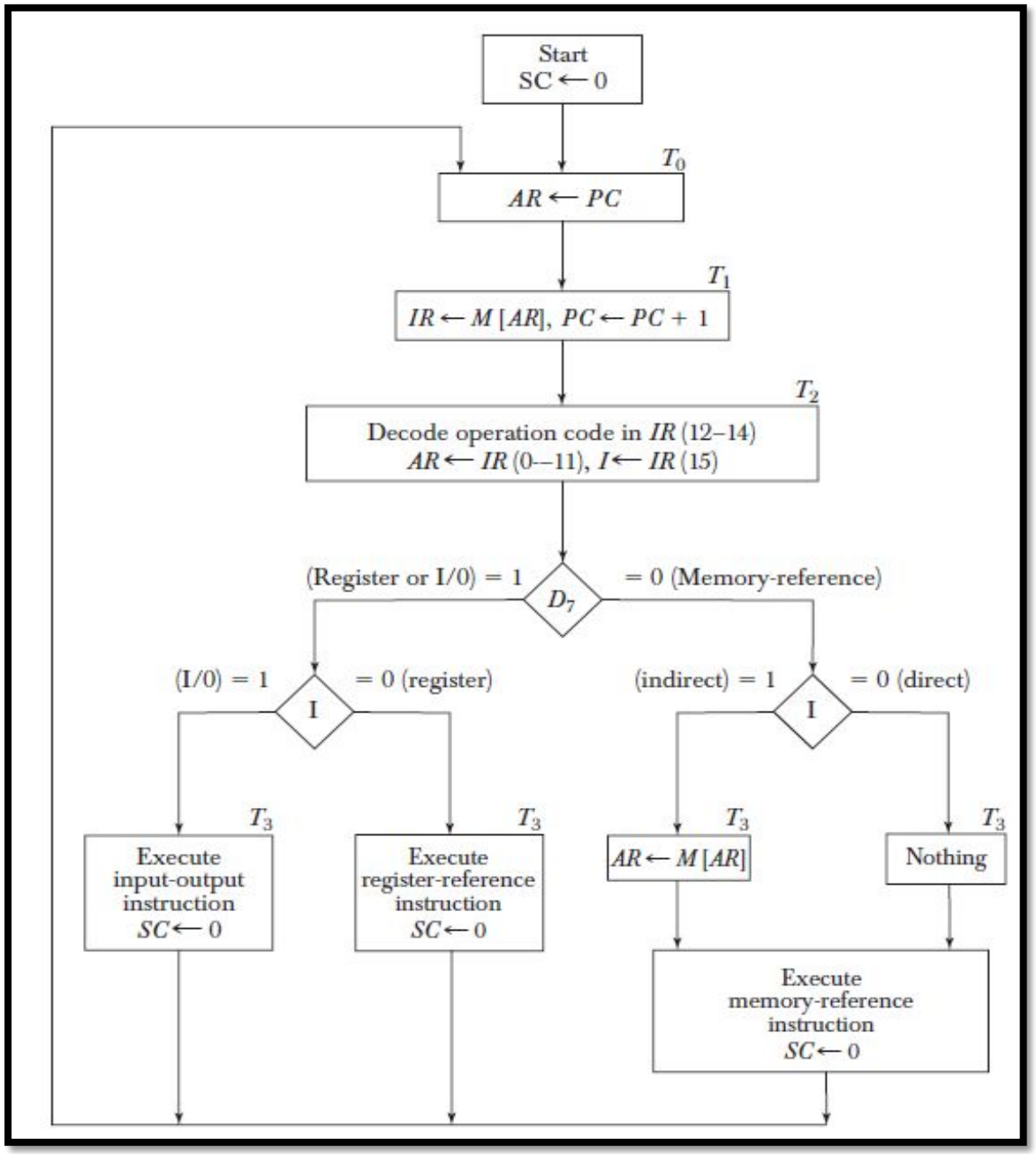


- The timing signal that is active after the decoding is T_3.
- During time T_3, the control unit determines the type of instruction that was just read from memory.
- The flowchart presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.

The three possible instruction types available in the basic computer are



Determine the Type of Instruction



Decoder output D_7 is equal to 1 if the operation code is equal to binary 111.

If D_7 =1, the instruction must be a register-reference or input–output type.

If D_7 = 0, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction.

Control then inspects the value of the first bit of the instruction, which is now available in flip-flop *I*.

If D_7 = 0 and I =1, we have a memory reference instruction with an indirect address.

It is then necessary to read the effective address from memory.

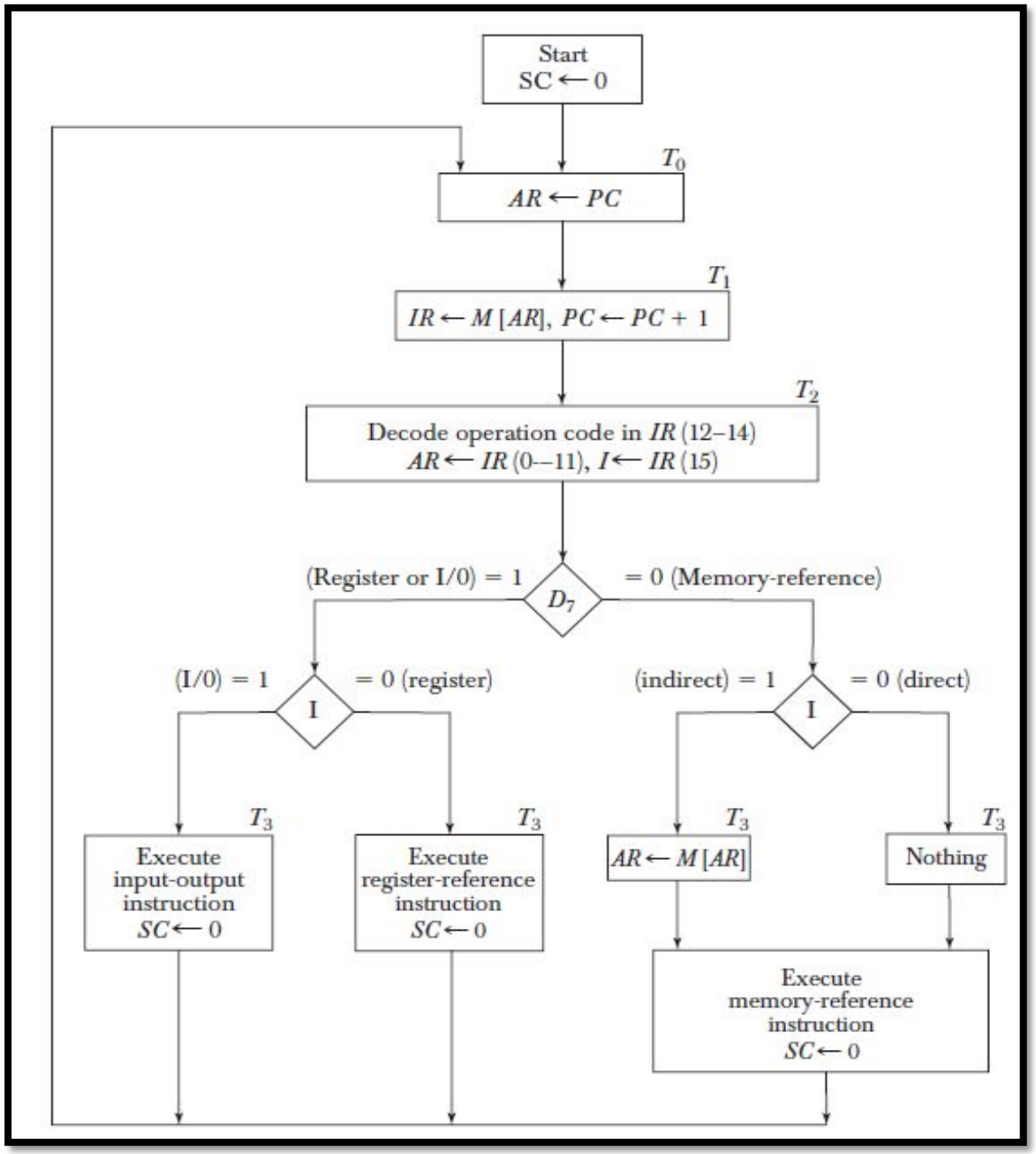
The microoperation for the indirect address condition can be symbolized by the register transfer statement

$AR \leftarrow M[AR]$

Initially, *AR* holds the address part of the instruction. This address is used during the memory read operation. The word at the address given by *AR* is read from memory and placed on the common bus.

The LD input of *AR* is then enabled to receive the indirect address that resided in the 12 least significant bits of the memory word.

Determine the Type of Instruction



The three instruction types are subdivided into four separate paths.

The selected operation is activated with the clock transition associated with timing signal T_3.

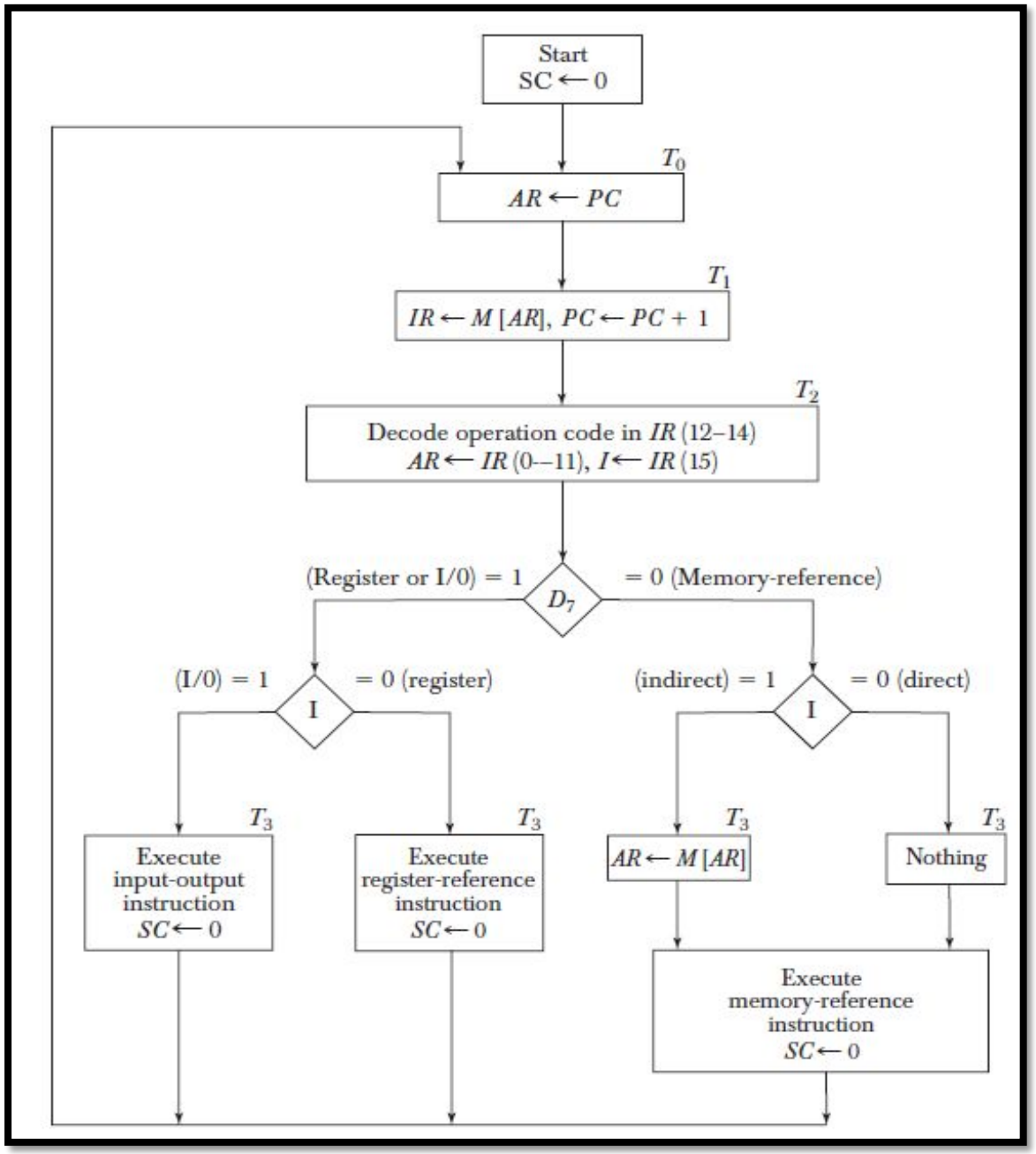
This can be symbolized as follows:

$$D_7' I T_3: \; AR \leftarrow M[AR]$$
$$D_7' I' T_3: \quad Nothing$$
$$D_7 I' T_3: Execute\ a\ register - reference\ Instruction$$
$$D_7 I T_3: Execute\ an\ input - output\ Instruction$$

When a memory-reference instruction with I =0 is encountered, it is not necessary to do anything since the effective address is already in AR.

However, the sequence counter SC must be incremented when $D_7' T_3$ = 1, so that the execution of the memory-reference instruction can be continued with timing variable T_4.

Determine the Type of Instruction



A register-reference or input-output instruction can be executed with the clock associated with timing signal T_3.

After the instruction is executed, *SC* is cleared to 0 and control returns to the fetch phase with $T_3 = 1$.

Note that the sequence counter *SC* is either incremented or cleared to 0 with every positive clock transition.

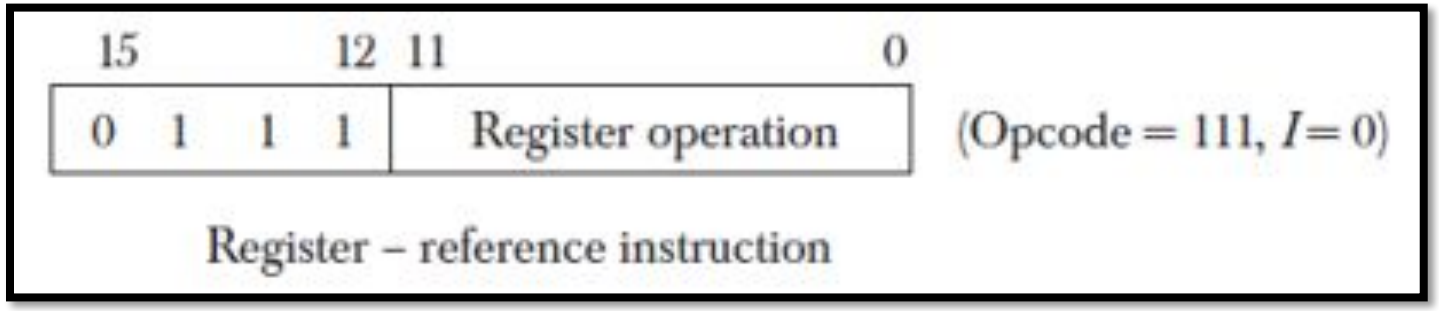
We will adopt the convention that if *SC* is incremented, we will not write the statement *SC* ← *SC* +1, but it will be implied that the control goes to the next timing signal in sequence.

When *SC* is to be cleared, we will include the statement *SC* ← 0.

RTL Interpretation of Instructions

Register-Reference Instructions

Register-Reference Instructions



Register-reference instructions are recognized by the control when D_7 =1 and I =0.

These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions.

These 12 bits are available in IR (0–11). They were also transferred to AR during time T_2.

Register-Reference Instructions

CLA	7800	Clear *AC*
CLE	7400	Clear *E*
CMA	7200	Complement *AC*
CME	7100	Complement *E*
CIR	7080	Circulate right *AC* and *E*
CIL	7040	Circulate left *AC* and *E*
INC	7020	Increment *AC*
SPA	7010	Skip next instruction if *AC* positive
SNA	7008	Skip next instruction if *AC* negative
SZA	7004	Skip next instruction if *AC* zero
SZE	7002	Skip next instruction if *E* is 0
HLT	7001	Halt computer

Execution of Register-Reference Instructions

$D_7 I' T_3 = r$ (common to all registcr-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC^* \rightarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

(REFER NOTES)

CLA	7800	Clear AC

For example, the instruction CLA has the hexadecimal code 7800 , which gives the binary equivalent 0111 1000 0000 0000.

The first bit is a zero and is equivalent to I.

The next three bits constitute the operation code and are recognized from decoder output D_7.

Bit 11 in IR is 1 and is recognized from B_{11}.

The control function that initiates the microoperation for this instruction is $D_7 I' T_3 B_{11} = r B_{11}$.

The execution of a register-reference instruction is completed at time T_3.

The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T_0.

Execution of Register-Reference Instructions

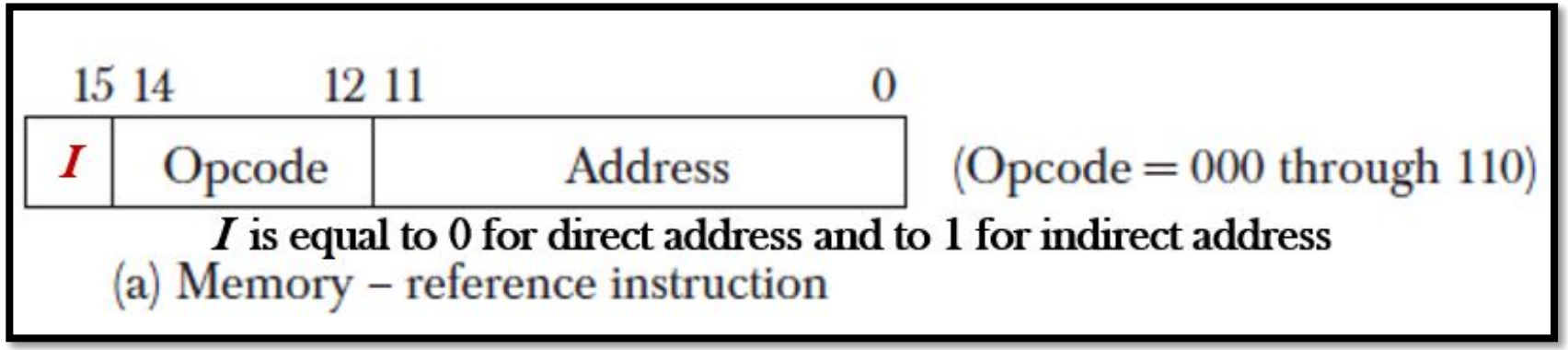
$D_7 I' T_3 = r$ (common to all registcr-reference instructions)
$IR(i) = B_i$ [bit in $IR(0–11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC^* \rightarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

(REFER NOTES)

Memory-Reference Instructions

Memory-Reference Instructions



I is equal to 0 for direct address and to 1 for indirect address

(a) Memory – reference instruction

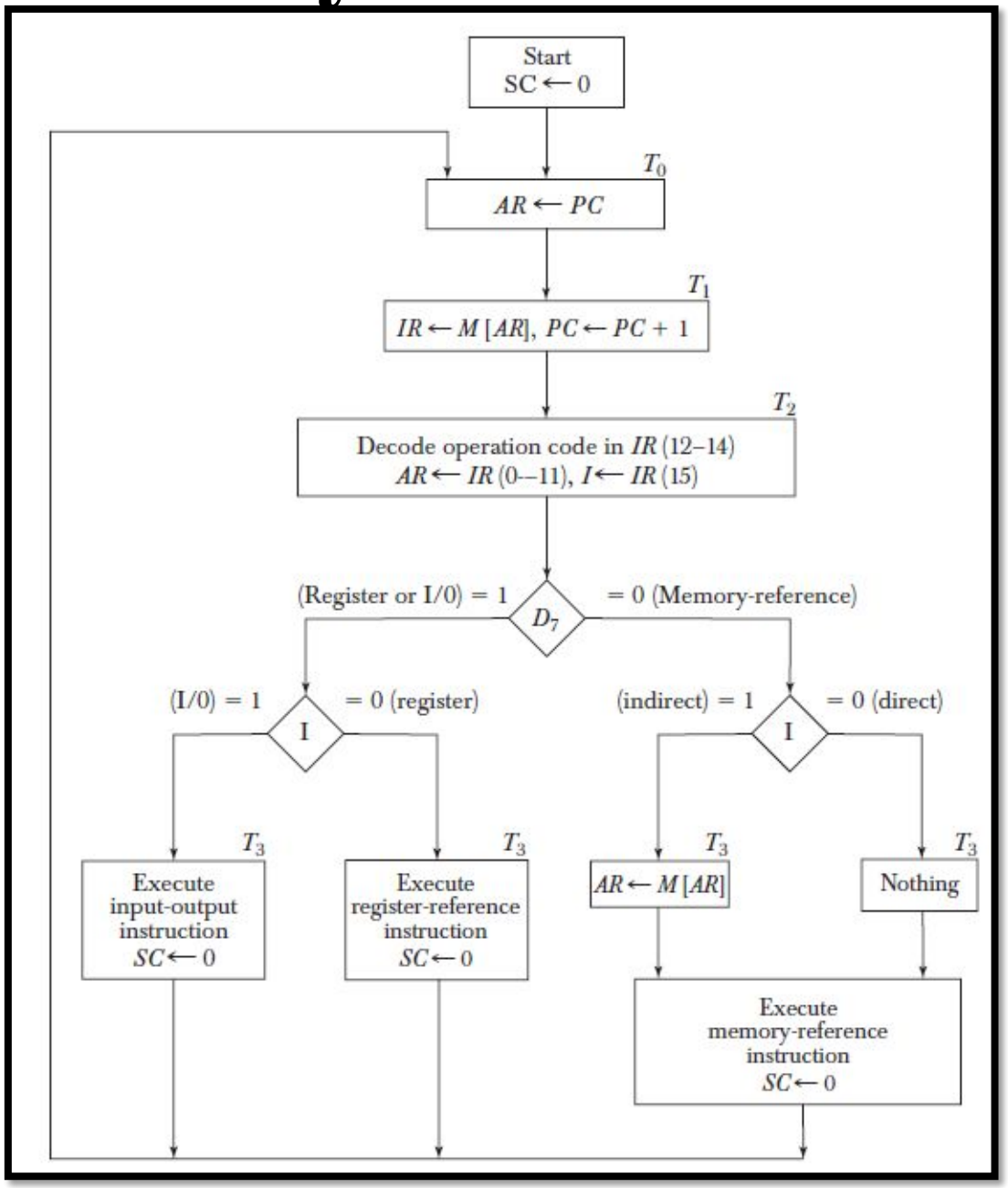
Symbol	Hexadecimal code		Description
	I = 0	*I* = 1	
AND	0xxx	8xxx	AND memory word to *AC*
ADD	lxxx	9xxx	Add memory word to *AC*
LDA	2xxx	Axxx	Load memory word to *AC*
STA	3xxx	Bxxx	Store content of *AC* in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero

Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1,$ If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

The decoded output D_i for $i = 0, 1, 2, 3, 4, 5,$ and 6 from the operation decoder that belongs to each instruction is included in the table.

Memory-Reference Instructions



The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when I =0, or during timing signal T_3 when I =1.

The execution of the memory- reference instructions starts with timing signal T_4.

The symbolic description of each instruction is specified in the table in terms of register transfer notation.

The actual execution of the instruction in the bus system will require a sequence of microoperations.

This is because data stored in memory cannot be processed directly.

The data must be read from memory to a register where they can be operated on with logic circuits.

AND to AC

- This is an instruction that performs the AND logic operation on pairs of bits in *AC* and the memory word specified by the effective address.
- The result of the operation is transferred to AC. The microoperations that execute this instruction are:

$$D_0T_4: \quad DR \leftarrow M[AR]$$

$$D_0T_5: \quad AC \leftarrow AC \wedge DR, \quad SC \leftarrow 0$$

(REFER NOTES)

ADD to AC

- This instruction adds the content of the memory word specified by the effective address to the value of *AC*.
- The sum is transferred into *AC* and the output carry *C*out is transferred to the *E* (extended accumulator) flip-flop.
- The

instru

$$D_1T_4: \quad DR \leftarrow M[AR]$$

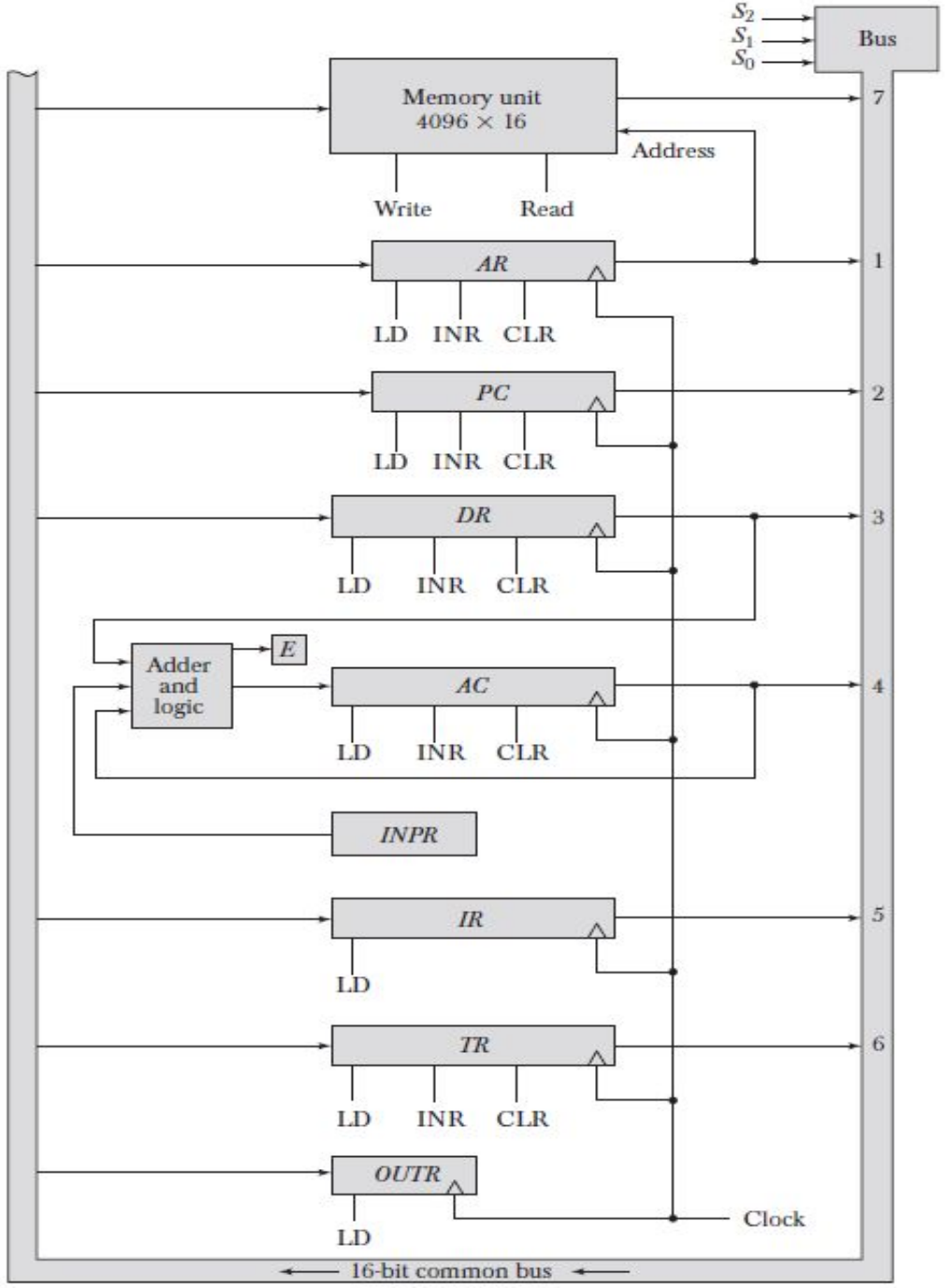
$$D_1T_5: \quad AC \leftarrow AC + DR, \quad E \leftarrow C_{out}, \quad SC \leftarrow 0$$

LDA: Load to AC

- This instruction transfers the memory word specified by the effective address to *AC*.
- The microoperations needed to execute this instruction are

$$D_2T_4: \quad DR \leftarrow M[AR]$$

$$D_2T_5: \quad AC \leftarrow DR, \quad SC \leftarrow 0$$



NOTE:

The reason for not connecting the bus to the inputs of *AC* is the delay encountered in the adder and logic circuit.

It is assumed that the time it takes to read from memory and transfer the word through the bus as well as the adder and logic circuit is more than the time of one clock cycle.

By not connecting the bus

STA: Store AC

- This instruction stores the content of AC into the memory word specified by the effective address.
- Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one mic

$$D_3T_4: \quad M[AR] \leftarrow AC, \quad SC \leftarrow 0$$

BUN: Branch Unconditionally

- This instruction transfers the program to the instruction specified by the effective address.
- Remember that *PC* holds the address of the instruction to be read from memory in the next instruction cycle.
- *PC* is incremented at time T_1 to prepare it for the address of the next instruction in the program sequence.
- The BUN instruction allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally.
- The instruction is executed with one microoperation:

$$D_4T_4: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

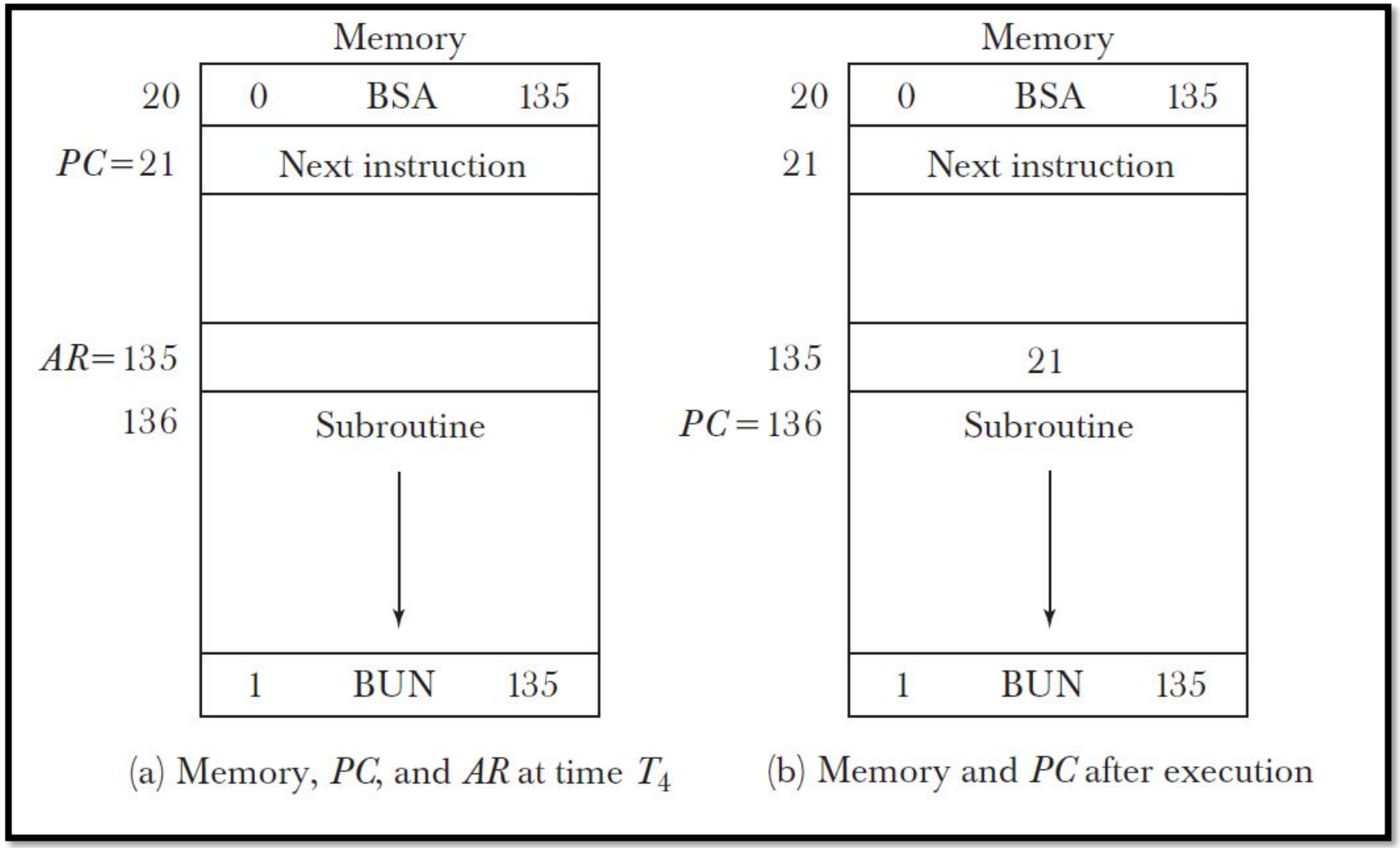
The effective address from *AR* is transferred through the common bus to *PC.*
Resetting *SC* to 0 transfers control to T_0 .
The next instruction, is then fetched and executed from the memory address given by the new value in *PC.*

BSA: Branch and Save Return Address

- This instruction is useful for branching to a portion of the program called a subroutine or procedure.
- When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in *PC*) into a memory location specified by the effective address.
- The effective address plus one is then transferred to *PC* to serve as the address of the first instruction in the subroutine.
- This operation was specified with the following register transfer:

$$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$$

Example of BSA instruction execution



(a) Memory, *PC*, and *AR* at time T_4 (b) Memory and *PC* after execution

$$M[135] \leftarrow 21, \qquad PC \leftarrow 135 + 1 = 136$$

(REFER NOTES)

Subroutine Call

- The BSA instruction performs the function usually referred to as a subroutine call.
- The indirect BUN instruction at the end of the subroutine performs the function referred to as a subroutine return.
- **In most commercial computers, the return address associated with a subroutine is stored in either a processor register or in a portion of memory called a stack.**

BSA: Branch and Save Return Address

- It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer.
- To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

$$D_5T_4: \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$$

$$D_5T_5: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

- Timing signal T_4 initiates a memory write operation, places the content of PC onto the bus, and enables the INR input of AR.
- The memory write operation is completed and AR is incremented by the time the next clock transition occurs.
- The bus is used at T_5 to transfer the content of AR to PC.

ISZ: Increment and Skip if Zero

- This instruction increments the word specified by the effective address, and if the incremented value is equal to 0, *PC* is incremented by 1.
- The programmer usually stores a negative number (in 2's complement) in the memory word.
- As this negative number is repeatedly incremented by one, it eventually reaches the value of zero.
- At that time *PC* is incremented by one in order to skip the next instruction in the program.
- Since it is not possible to increment a word inside the memory, it is necessary to read the word into *DR*, increment *DR*, and store the word bac
- Thi

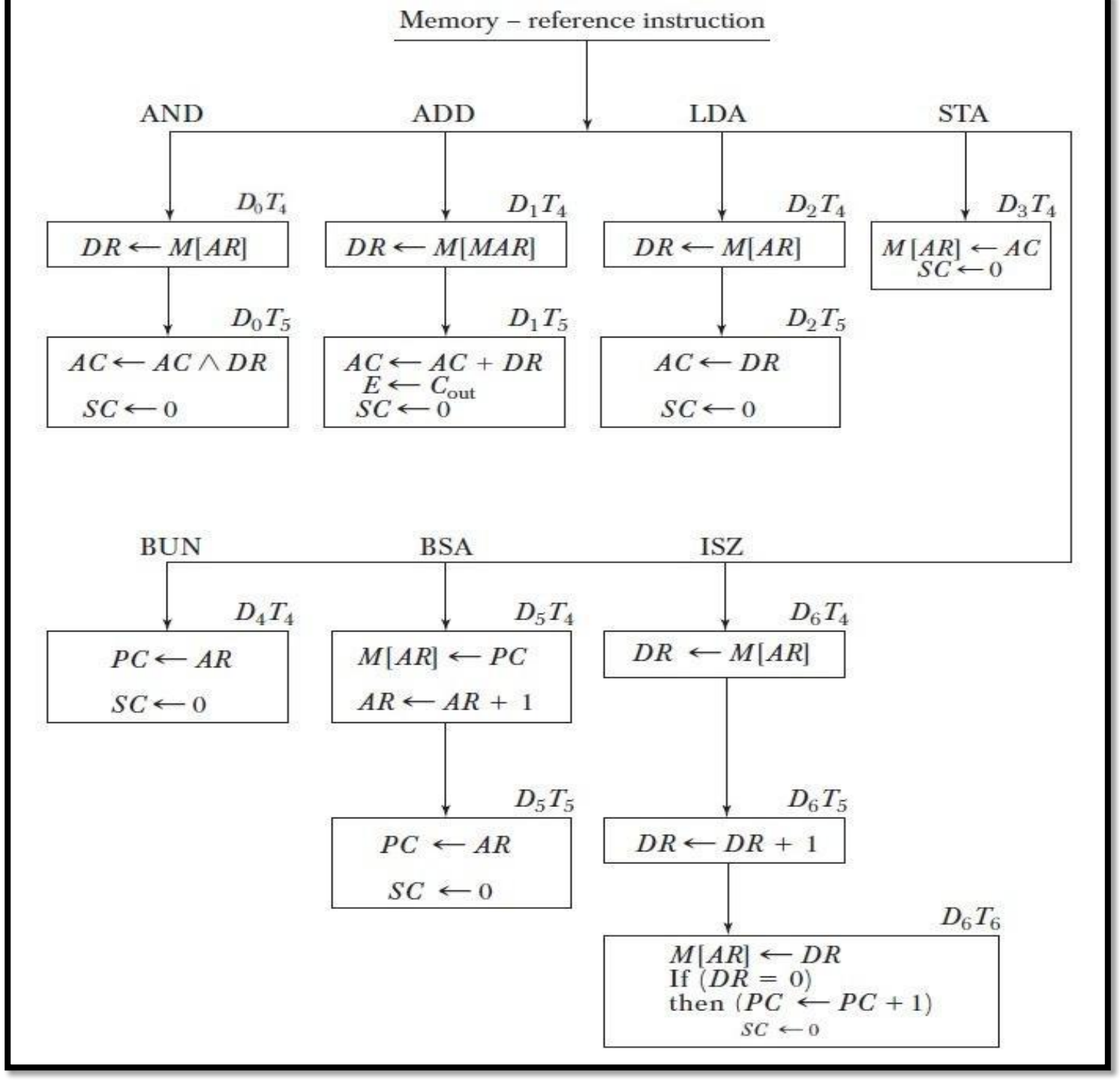
$$D_6T_4: \quad DR \leftarrow M[AR]$$

$$D_6T_5: \quad DR \leftarrow DR + 1$$

$$D_6T_6: \quad M[AR] \leftarrow DR, \quad \text{if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), \quad SC \leftarrow 0$$

Control Flowchart

- Note that we need only seven timing signals to execute the longest instruction (ISZ).
- The computer can be designed with a 3-bit sequence counter.
- The reason for using a 4-bit counter for *SC* is to provide additional timing signals for other instructions which may take more than eight timing signals.



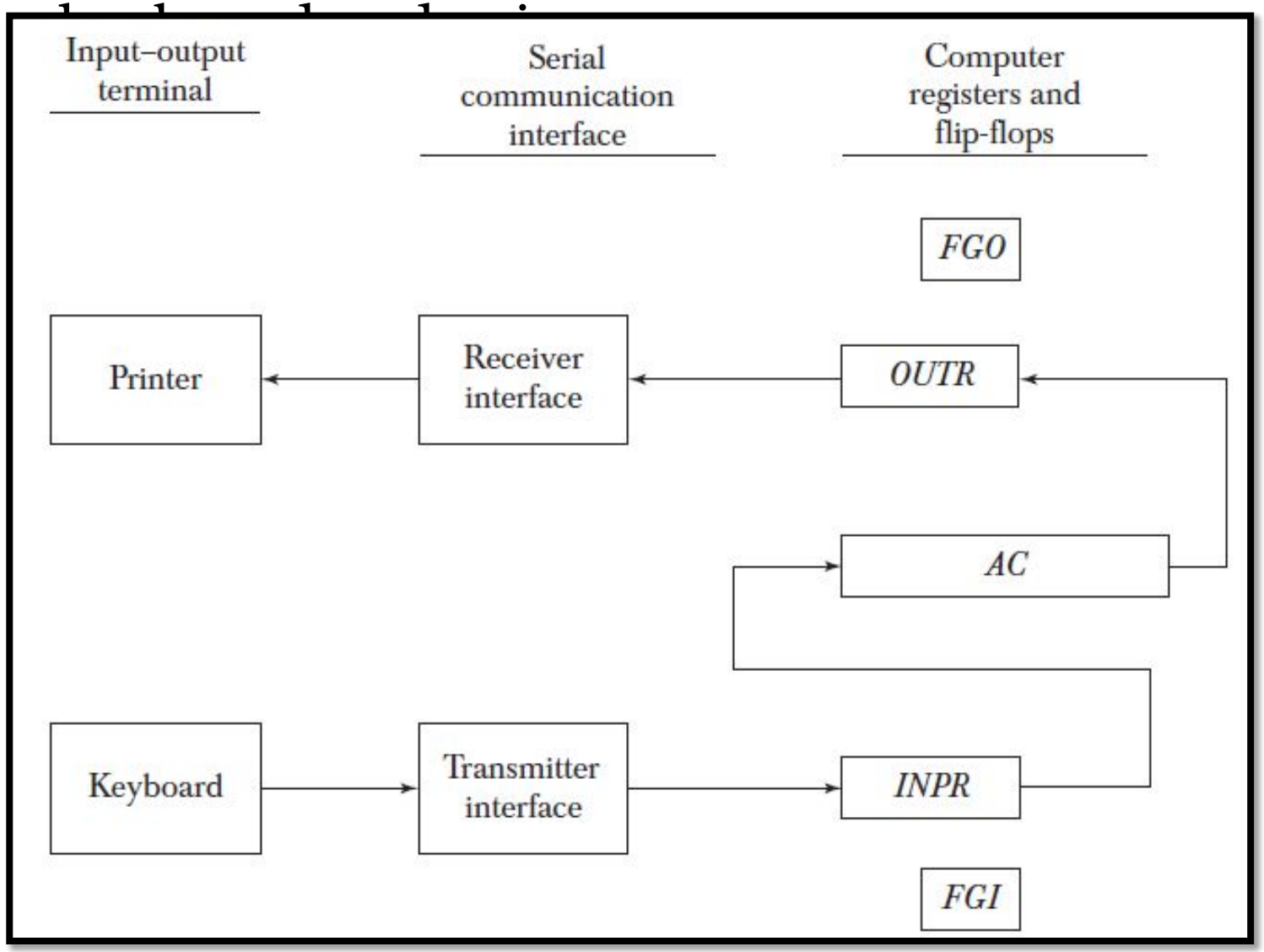
Input–Output and Interrupt

Introduction

- A computer can serve no useful purpose unless it communicates with the external environment.
- Instructions and data stored in memory must come from some input device.
- Computational results must be transmitted to the user through some output device.
- Commercial computers include many types of input and output devices.

Input–Output Configuration

- To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with



(REFER NOTES)

Role of FGI and FGO

- The flag is needed to synchronize the timing rate difference between the input device and the computer.

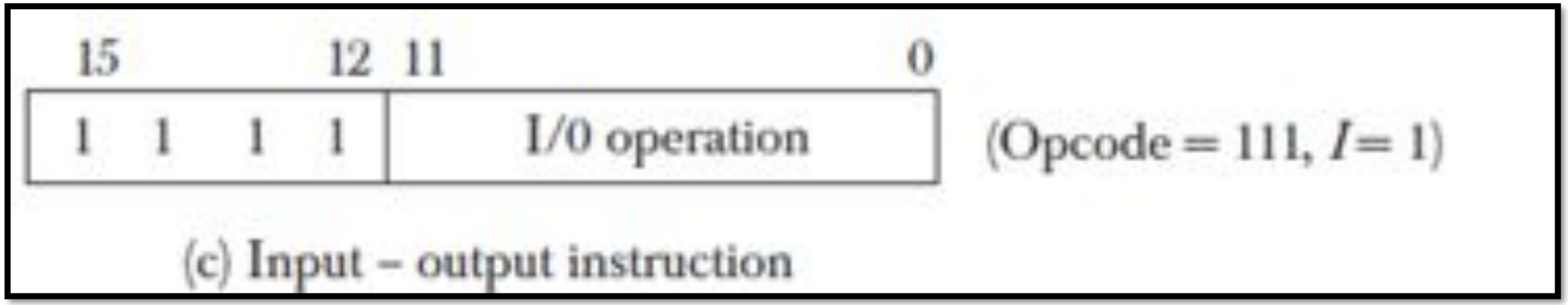
Input

- The process of information transfer is as follows.
- Initially, the input flag *FGI* is cleared to 0.
- When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into *INPR* and the input flag *FGI* is set to 1.
- As long as the flag is set, the information in *INPR* cannot be changed by striking another key.
- The computer checks the flag bit; if it is 1, the information from *INPR* is transferred in parallel into *AC* and *FGI* is cleared to 0.
- Once the flag is cleared, new information can be shifted into *INPR* by striking another key.

Output

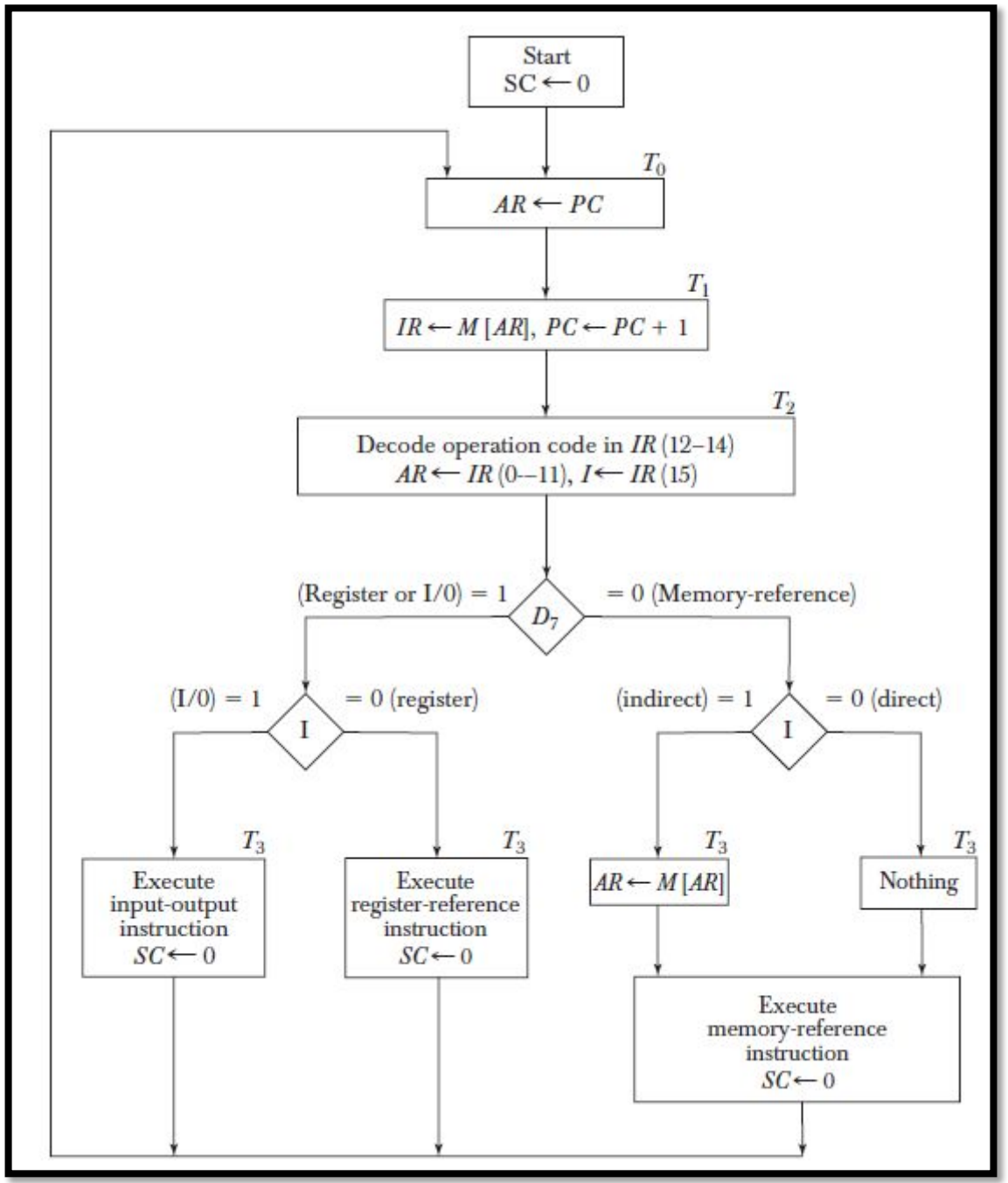
- The output register *OUTR* works similarly but the direction of information flow is reversed.
- Initially, the output flag *FGO* is set to 1. The computer checks the flag bit; if it is 1, the information from *AC* is transferred in parallel to *OUTR* and *FGO* is cleared to 0.
- The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets *FGO* to 1.
- The computer does not load a new character into *OUTR* when *FGO* is 0 because this condition indicates that the output device is in the process of printing the character.

Input–Output Instructions



(c) Input – output instruction

Input–Output Instructions



Input–Output Instructions

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR, \quad FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC\,(0–7), \quad FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

Input–Output Instructions (Description)

- The INP instruction transfers the input information from *INPR* into the eight low-order bits of *AC* and also clears the input flag to 0.
- The OUT instruction transfers the eight least significant bits of *AC* into the output register *OUTR* and clears the output flag to 0.
- The next two instructions in check the status of the flags and cause a skip of the next instruction if the flag is 1. The instruction that is skipped will normally be a branch instruction to return and check the flag again. The branch instruction is not skipped if the flag is 0. If the flag is 1, the branch instruction is skipped and an input or output instruction is executed.
- The last two instructions set and clear an interrupt

Limitation of Programmed Control Transfer

- The process of communication just described is referred to as **programmed control transfer.**
- The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer.
- **The difference of information flow rate between the computer and that of the input–output device makes this type of transfer inefficient.**
- To see why this is inefficient, consider a computer that can go through an instruction cycle in 1 μs.
- Assume that the input–output device can transfer information at a maximum rate of 10 characters per second.
- This is equivalent to one character every 100,000 μs.
- Two instructions are executed when the computer checks the flag bit and decides not to transfer the information.
- This means that at the maximum rate, the computer will check the flag 50,000 times between each transfer.
- The computer is wasting time while checking the flag instead of doing some other useful processing task.

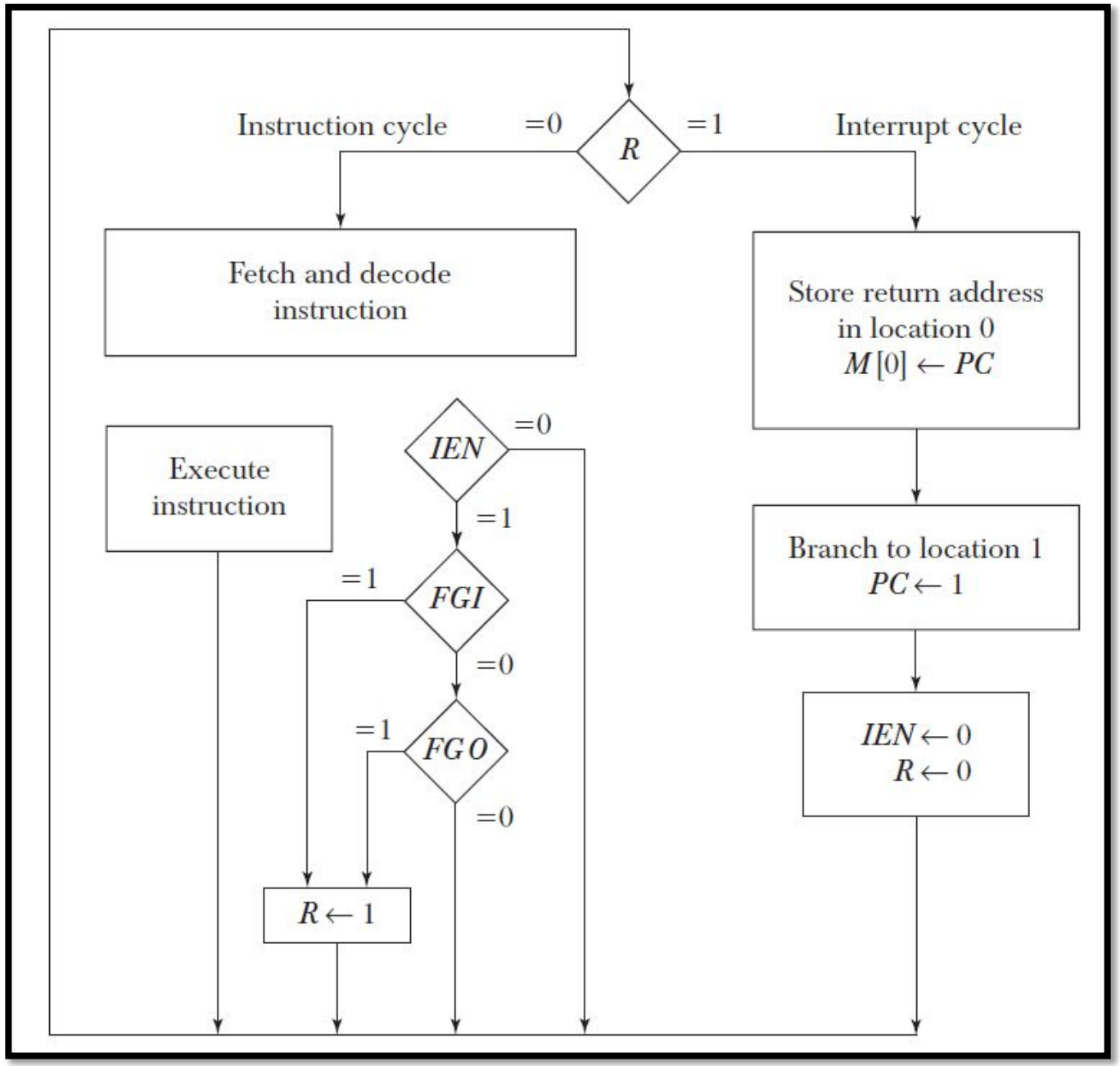
Program Interrupt

- An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer.
- In the meantime the computer can be busy with other tasks.
- This type of transfer uses the interrupt facility.
- While the computer is running a program, it does not check the flags.
- However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set.
- The computer deviates momentarily from what it is doing to take care of the input or output transfer.
- It then returns to the current program to continue what it was doing before the interrupt.

Program Interrupt

- The interrupt enable flip-flop *IEN* can be set and cleared with two instructions.
- When *IEN* is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer.
- When *IEN* is set to 1 (with the ION instruction), the computer can be interrupted.
- These two instructions provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.

Flowchart for Interrupt Cycle

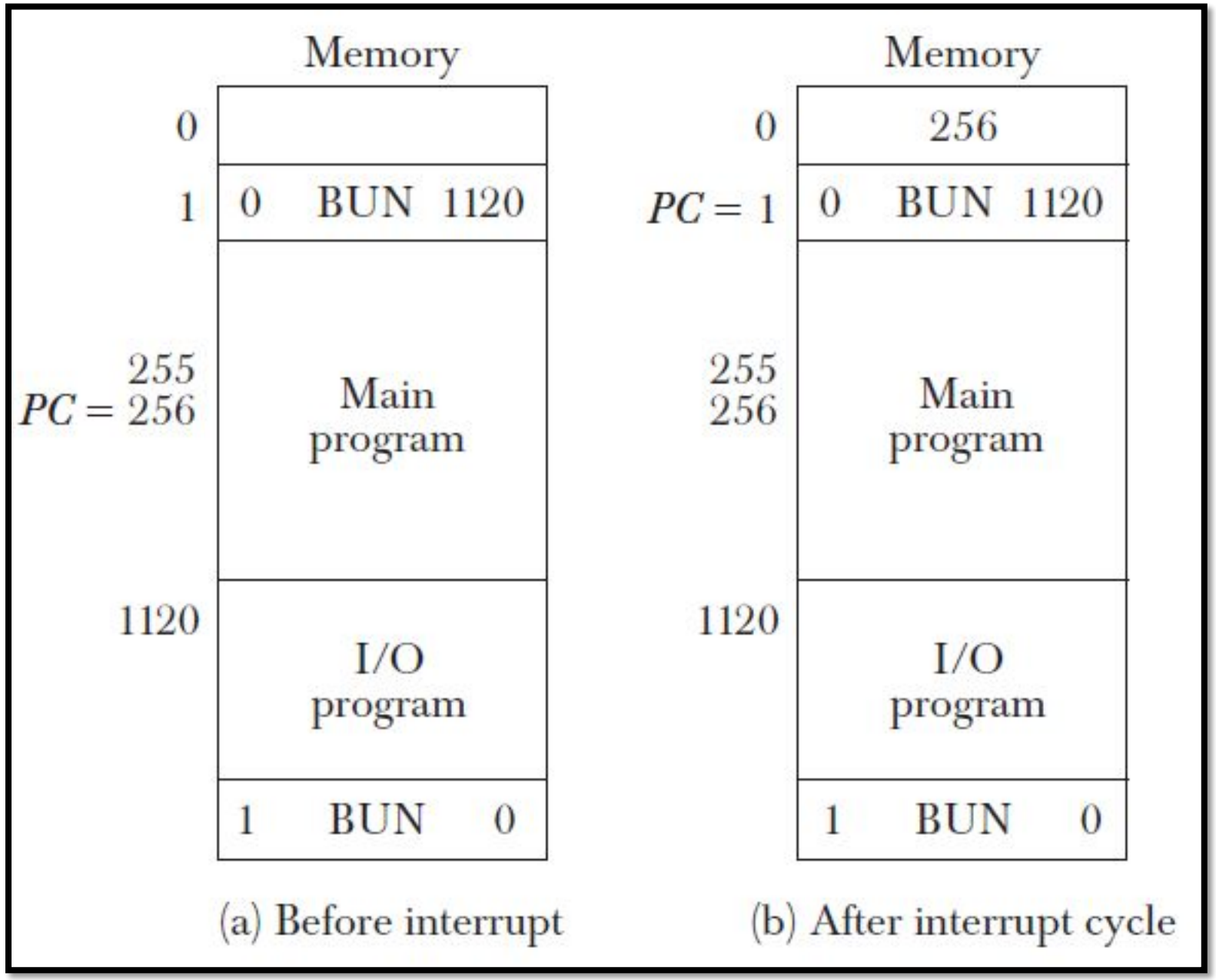


(REFER NOTES)

Interrupt Cycle

- The interrupt cycle is a hardware implementation of a branch and save return address operation.
- The return address available in *PC* is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted.
- This location may be a processor register, a memory stack, or a specific memory location.
- Here we choose the memory location at address 0 as the place for storing the return address.
- Control then inserts address 1 into *PC* and clears *IEN* and *R* so that no more interruptions can occur until the interrupt request from the flag has been serviced.

Demonstration of the interrupt cycle



(REFER NOTES)

Interrupt Cycle

- We are now ready to list the register transfer statements for the interrupt cycle.
- The interrupt cycle is initiated after the last execute phase if the interrupt flipflop R is equal to 1.
- This flip-flop is set to 1 if *IEN* 1 and either *FGI* or *FGO* are equal to 1.
- This can happen with any clock transition except when timing signals T_0, T_1, or T_2 are active.
- The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement:

$$T_0' T_1' T_2' (IEN)(FGI + FGO): \quad R \leftarrow 1$$

The symbol + between *FGI* and *FGO* in the control function designates a logic OR operation. This is ANDed with *IEN* and $T_0' T_1' T_2'$.

Modified fetch and decode phase

T_0: $AR \leftarrow PC$

T_1: $IR \leftarrow M[AR],\ PC \leftarrow PC + 1$

T_2: $D_0, \ldots, D_7 \leftarrow \text{Decode } IR(12\text{–}14),\ AR \leftarrow IR(0-11),\ I \leftarrow IR(15)$

$R'T_0$: $AR \leftarrow PC$

$R'T_1$: $IR \leftarrow M[AR],\ PC \leftarrow PC\ 1$

$R'T_2$: $D_0, \ldots, D_7 \leftarrow$Decode $IR(12\text{-}14),\ AR \leftarrow IR(011),\ I \leftarrow IR\ (15)$

(REFER NOTES)

Interrupt Cycle

- The interrupt cycle stores the return address (available in *PC*) into memory location 0, branches to memory location 1, and clears *IEN*, *R*, and *SC* to 0.
- This can be done with the following sequence of micr

$$
\begin{aligned}
RT_0: &\quad AR \leftarrow 0, \quad TR \leftarrow PC \\
RT_1: &\quad M[AR] \leftarrow TR, \quad PC \leftarrow 0 \\
RT_2: &\quad PC \leftarrow PC + 1, \quad IEN \leftarrow 0, \quad R \leftarrow 0, \quad SC \leftarrow 0
\end{aligned}
$$

(REFER NOTES)