Computer Aritnmetic:

Introduction:

- Arithmetic instructions in digital computers manipulate data to produce results necessary for the solution of computational problems.
- These instructions perform arithmetic calculations and are responsible for the bulk of activity involved in processing data in a computer.

The four basic arithmetic operations are **addition, subtraction, multiplication and division.** From these four bulk operations, it is possible to formulate other arithmetic functions and solve scientific problems by means of numerical analysis methods.

- An arithmetic processor is the part of a processor unit that executes arithmetic operations. The data type assumed to reside in **processor registers** during the execution of an arithmetic instruction is specified in the definition of the instruction. A:n arithmetic instruction may specify binary or decimal data, and in each case the data may be in fixed-point or floating-point form.
- We must be thoroughly familiar with the sequence of steps to be followed in order to carry out the operation and achieve a correct result. The solution to any problem that is stated by a finite number of well-defined procedural steps is called an **algorithm**.
- Usually, an algorithm will contain a number of procedural steps which are dependent on results of previous steps. A convenient method for presenting algorithms is a **flowchart**.

Addition and Subtraction:

- As we have discussed, there are three ways of representing negative fixed-point binary numbers: **signed-magnitude**, **signed-1's complement**, or **signed-2's complement**. Most computers use the signed-2's complement representation when performing arithmetic operations with integers.

i. Addition and Subtraction with Signed-Magnitude Data:

When the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table shown below.

		Subtract Magnitudes		
Operation	Add Magnitudes	When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

Algorithm: (Addition with Signed-Magnitude Data)

i. When the signs of A and B are identical ,add the two magnitudes and attach the sign of A to the result.

ii. When the signs of A and B are different, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B.

iii. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

Algorithm: (Subtraction with Signed-Magnitude Data)

i. When the signs of A and B are different, add the two magnitudes and attach the sign of A to the result.

ii. When the signs of A and B are identical, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B.

iii. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

The two algorithms are similar except for the sign comparison. The procedure to be followed for identical signs in the addition algorithm is the same as for different signs in the subtraction algorithm, and vice versa.

Hardware Implementation:

To implement the two arithmetic operations with hardware, it is first necessary that the two numbers be stored in registers.

i. Let A and B be two registers that hold the magnitudes of the numbers, and A_S and B_S be two flip-flops that hold the corresponding signs.
ii. The result of the operation may be transferred to a third register: however, a saving is achieved if the result is transferred into A and A_S. Thus A and A_S together form an accumulator register.

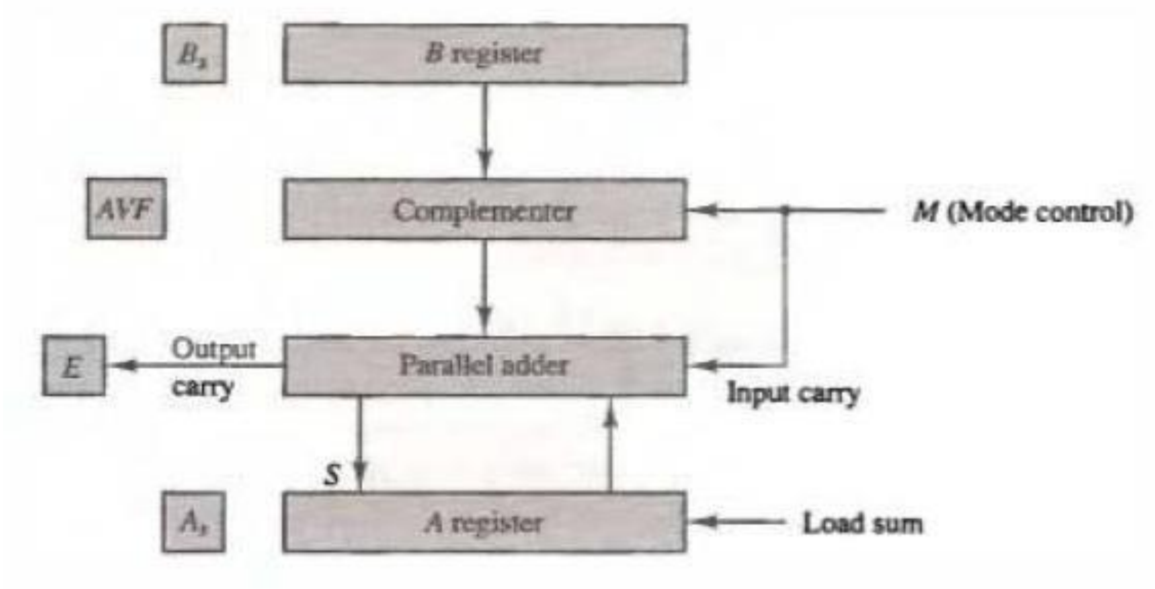
Consider now the hardware implementation of the algorithms above.

- First, **a parallel-adder** is needed to perform the microoperation A + B.
- Second, **a comparator circuit** is needed to establish if A > B, A = B, or A < B.
- Third, **two parallel-subtractor circuits** are needed to perform the microoperations A - B and B - A. The sign relationship can be determined from an exclusive-OR gate with A_S and B_S as inputs.

The below figure shows a block diagram of the hardware for implementing the addition and subtraction operations. It consists of registers A and B and sign flip-flops A_S and B_S.

- Subtraction is done by adding A to the 2' s complement of B. The output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitudes of the two numbers.
- The add-overflow flip-flop AVF holds the overflow bit when A and B are added.

Figure (i): Hardware for addition and subtraction with Signed-Magnitude Data



The complementer provides an output of B or the complement of B depending on the state of the mode control M.

- ❖ When M = 0, the output of B is transferred to the adder, the input carry is 0, and the output of the adder is equal to the sum A + B.
- ❖ When M= 1, the l's complement of B is applied to the adder, the input carry is 1, and output $S = A + \overline{B} + 1$. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction A - B.

Hardware Algorithm

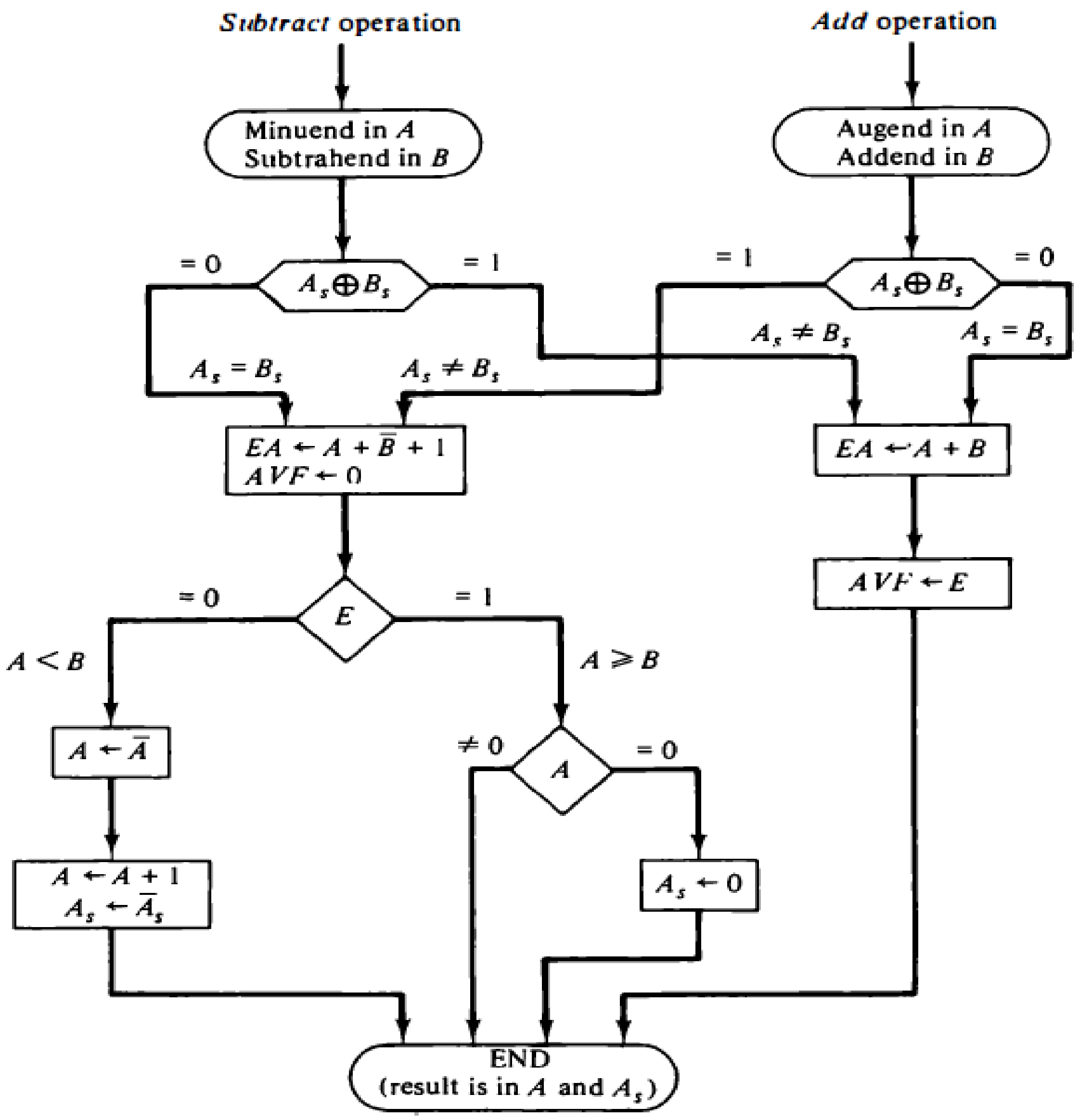
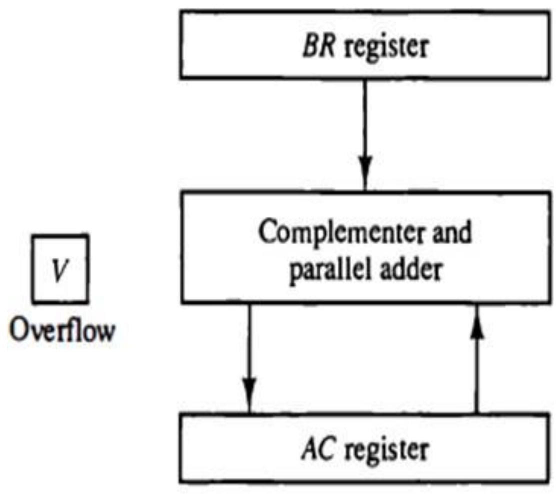


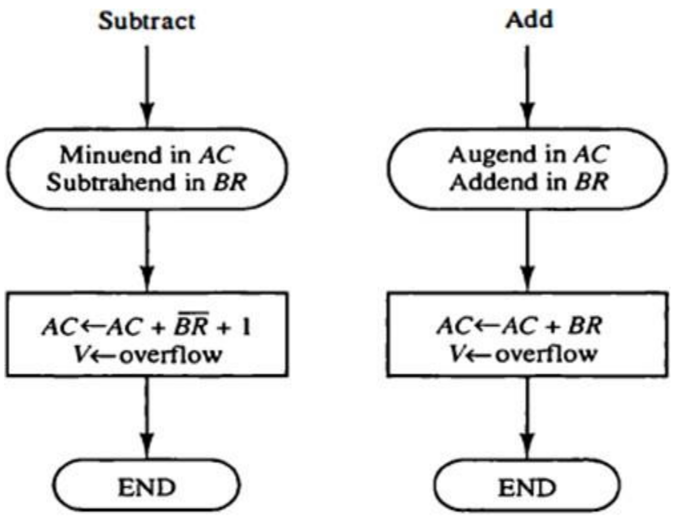
Figure (j): Flowchart for add and subtract operations

ii. Addition and Subtraction with Signed-2's Complement Data

- The register configuration for the hardware implementation is shown in the below Figure(a). We name the A register AC (accumulator) and the B register BR. The leftmost bit in AC and BR represent the sign bits of the numbers. The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder. The overflow flip-flop V is set to 1 if there is an overflow. The output carry in this case is discarded.
- The algorithm for adding and subtracting two binary numbers in signed-2' s complement representation is shown in the flowchart of Figure(b). The sum is obtained by adding the contents of AC and BR (including their sign bits). The overflow bit V is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise. The subtraction operation is accomplished by adding the content of AC to the 2's complement of BR.
- Comparing this algorithm with its signed-magnitude counterpart, we note that it is much simpler to add and subtract numbers if negative numbers are maintained in signed-2' s complement representation.



Figure(a): Hardware for addition & subtraction of 2's complement numbers



Figure(b): Algorithm for adding & subtracting of 2's complement numbers

Multiplication Algorithms:

Multiplication of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive **shift** and **adds** operations. This process is best illustrated with a numerical example.

```
 23       10111     Multiplicand
 19     × 10011     Multiplier
         -------
          10111   }
         10111    }
        00000   + }  Partial
       00000      }  Products
      10111       }
      ---------
437   110110101     Product
```

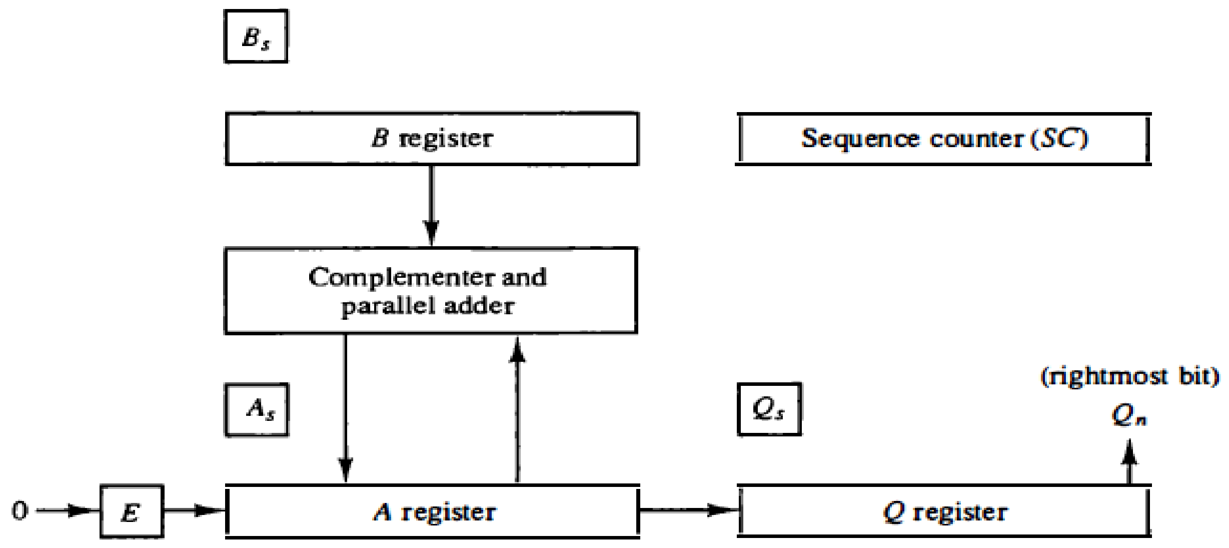
The process of multiplication:

- It consists of looking at successive bits of the multiplier, least significant bit first.
- If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down.
- The numbers copied down in successive lines are shifted one position to the left from the previous number.
- Finally, the numbers are added and their sum forms the product.

The sign of the product is determined from the signs of the multiplicand and multiplier. If they are alike, the sign of the product is **positive**. If they are unlike, the sign of the product is **negative**.

Hardware Implementation for Signed-Magnitude Data

→ The registers A, B and other equipment are shown in Figure (a). The multiplier is stored in the Q register and its sign in Qs. The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.



Figure(k): Hardware for multiply operation.

- ➔ Initially, the **multiplicand** is in register B and the **multiplier** in Q, Their corresponding signs are in Bs and Qs, respectively
- ➔ The sum of A and B forms a **partial product** which is transferred to the EA register.
- ➔ Both partial product and multiplier are shifted to the right. This shift will be denoted by the statement shr EAQ to designate the right shift.
- ➔ The least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right.

In this manner, the rightmost flip-flop in register Q, designated by Q_n, will hold the bit of the multiplier, which must be inspected next.

Hardware Algorithm:

→Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs, respectively. The signs are compared, and both A and Q are set to correspond to the sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier.

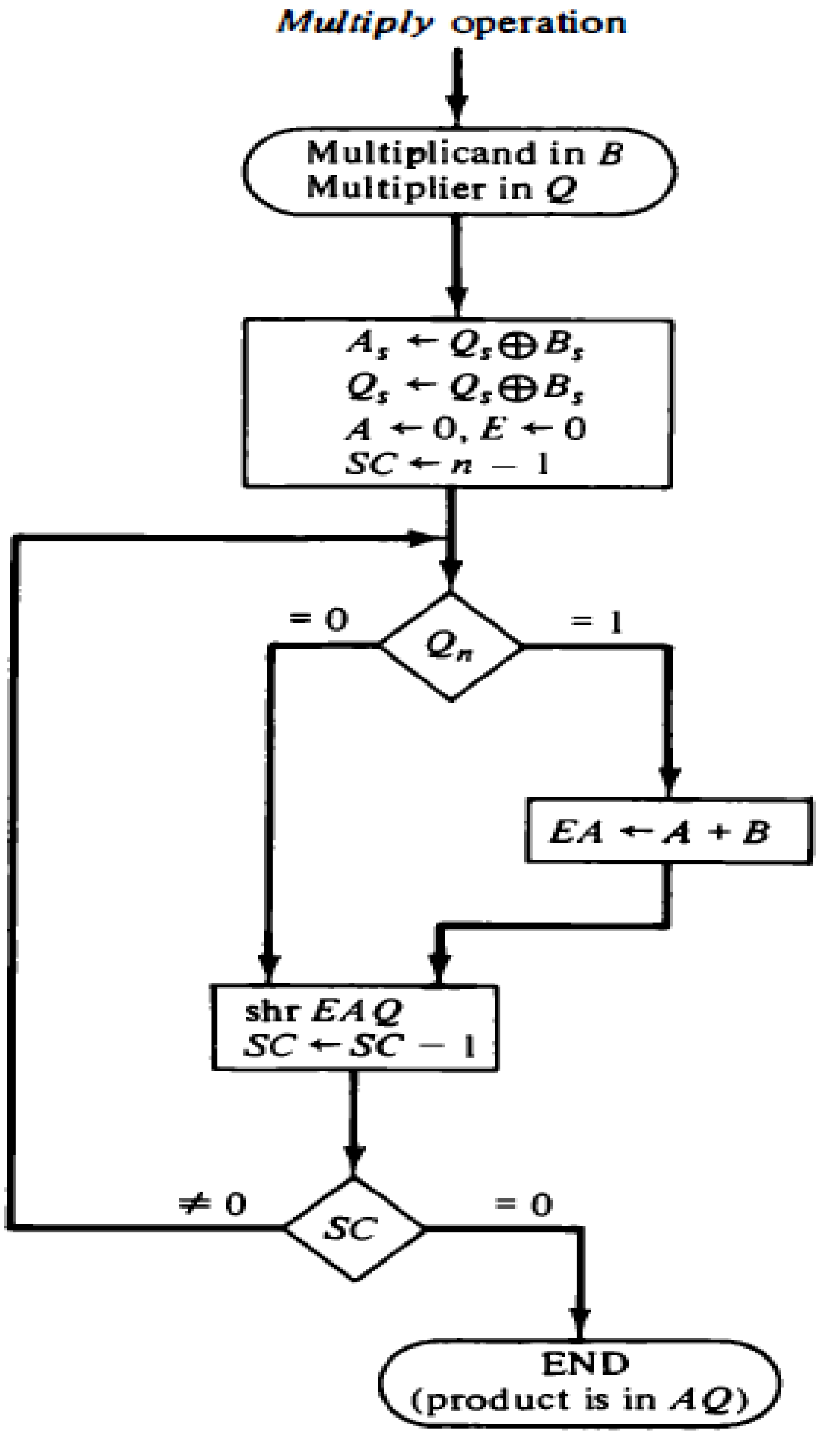
→After the initialization, the low-order bit of the multiplier in Qn is tested.

i. If it is 1, the multiplicand in B is added to the present partial product in A .
ii. If it is 0 , nothing is done. Register EAQ is then shifted once to the right to form the new partial product.

→The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC = 0.

→The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

A flowchart of the hardware multiply algorithm is shown in the below figure (l).



Figure(l): Flowchart for multiply operation.

Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

Figure (m): Numerical Example of multiplication

Booth Multiplication Algorithm:(multiplication of 2's complement data):

→Booth algorithm gives a procedure for multiplying binary integers in signed-2's complement representation.

→Booth algorithm requires examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the **following rules**:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of O's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

Hardware implementation of Booth algorithm Multiplication:

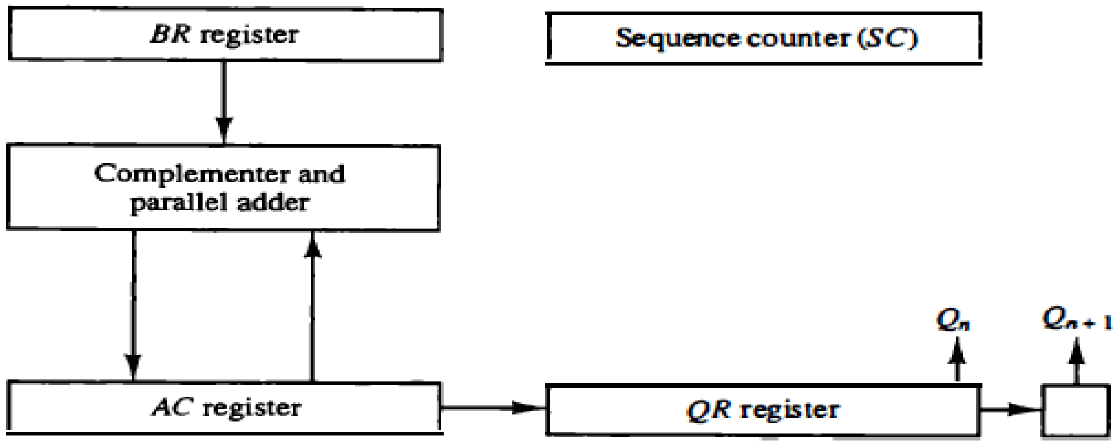


Figure (n): Hardware for Booth Algorithm

The hardware implementation of Booth algorithm requires the register configuration shown in figure (n). This is similar addition and subtraction hardware except that the sign bits are not separated from the rest of the registers. To show this difference, we rename registers A, B, and Q, as AC, BR, and QR, respectively. Q_n designates the least significant bit of the multiplier in register

QR. An extra flip-flop Q_{n+1}, is appended to QR to facilitate a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in Figure (o).

Hardware Algorithm for Booth Multiplication:

→AC and the appended bit Q_{n+1} are initially cleared to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Q_n and Q_{n+1} are inspected.

- **i.** If the two bits are equal to 10, it means that the first 1 in a string of 1's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.
- **ii.** If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC.
- **iii.** When the two bits are equal, the partial product does not change.
- **iv.** The next step is to shift right the partial product and the multiplier (including bit Q_{n+1}). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged. The sequence counter is decremented and the computational loop is repeated n times.

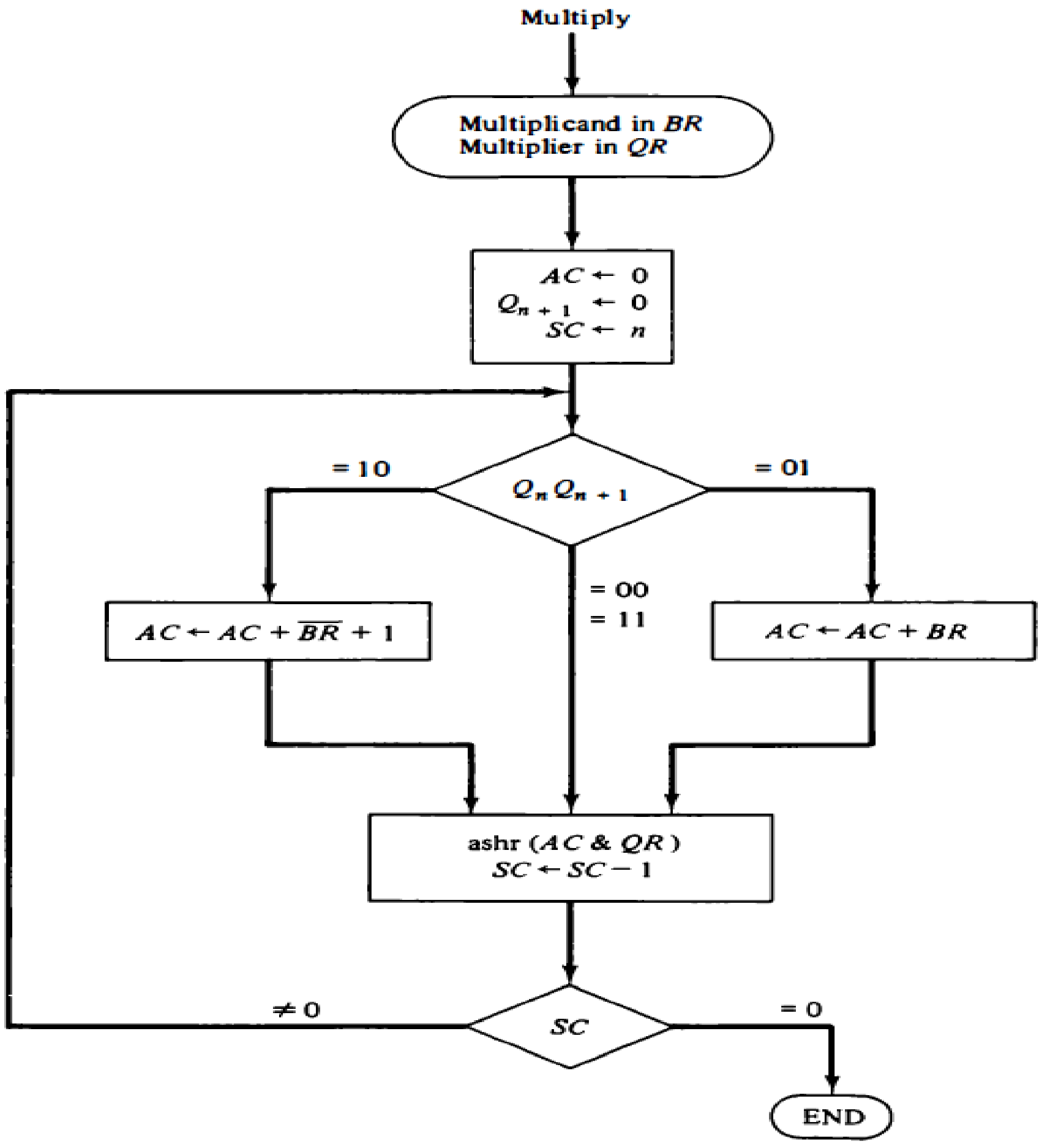
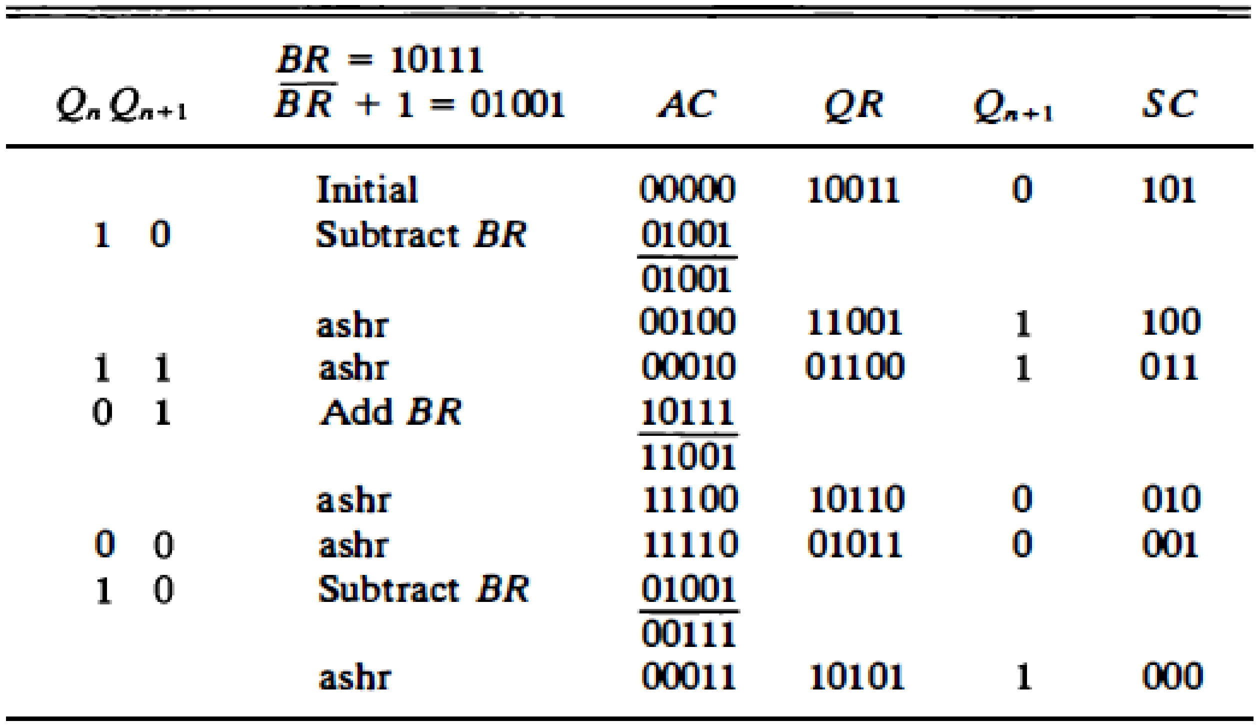


Figure (o): Booth Algorithm for multiplication of 2's complement numbers

Example: multiplication of (- 9) x (- 13) = + 117 is shown below. Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC and QR and is positive.



Q_n Q_{n+1}	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract *BR*	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add *BR*	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract *BR*	01001			
		00111			
	ashr	00011	10101	1	000

Figure (p): Example of Multiplication with Booth Algorithm.

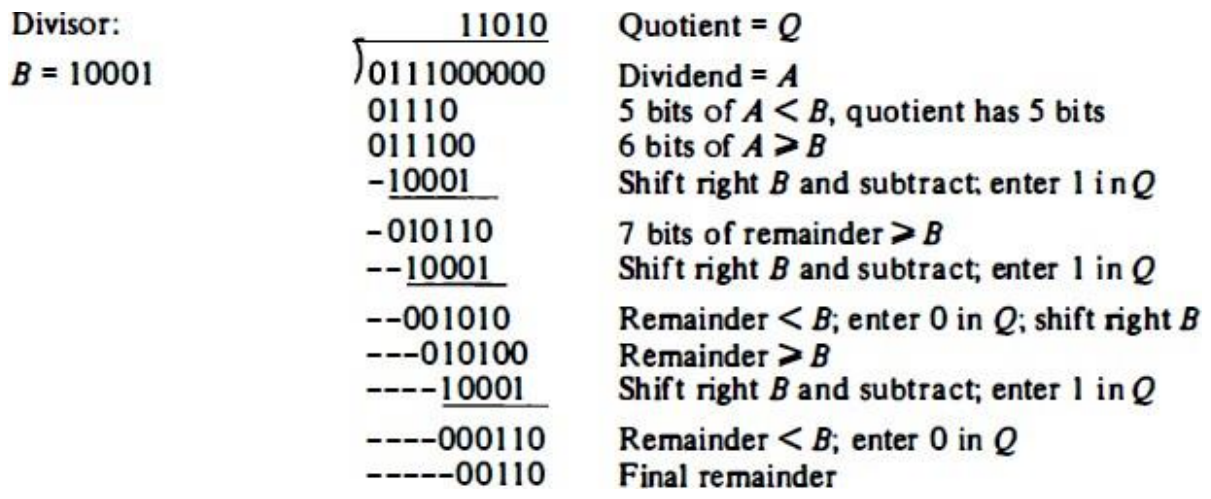
Division Algorithms:

- Division of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations.

The division process is illustrated by a numerical example in the below figure (q).

- The divisor B consists of five bits and the dividend A consists of ten bits. The five most significant bits of the dividend are **compared** with the divisor. Since the 5-bit number is smaller than B, we try again by taking the sixth most significant bits of A and compare this number with B. The 6-bit number is greater than B, so we place a 1 for the quotient bit. The divisor is then shifted once to the right and subtracted from the dividend.

- The difference is called a **partial remainder** because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor.
- If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder.
- If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.



```
Divisor:              11010         Quotient = Q
B = 10001           )0111000000     Dividend = A
                     01110          5 bits of A < B, quotient has 5 bits
                     011100         6 bits of A > B
                     -10001         Shift right B and subtract; enter 1 in Q
                     -010110        7 bits of remainder > B
                     --10001        Shift right B and subtract; enter 1 in Q
                     --001010       Remainder < B; enter 0 in Q; shift right B
                     ---010100      Remainder > B
                     ----10001      Shift right B and subtract; enter 1 in Q
                     ----000110     Remainder < B; enter 0 in Q
                     -----00110     Final remainder
```

Figure (q): Example of Binary Division

Hardware Implementation for Signed-Magnitude Data:

The hardware for implementing the division operation is identical to that required for multiplication.

- ✓ The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. The information about the relative magnitude is available in E.
- ✓ If E = 1, it signifies that A≥B. A quotient bit 1 is inserted into Q, and the partial remainder is shifted to the left to repeat the process.
- ✓ If E = 0, it signifies that A < B so the quotient in Qn remains a 0. The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed.
- ✓ Note that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and the final remainder is in A.

The sign of the quotient is determined from the signs of the dividend and the divisor. If the two signs are alike, the sign o f the quotient is **plus**. If they are unalike, the sign is **minus**. The sign of the remainder is the same as the sign of the dividend.

Divide Overflow

- ❑ The division operation may result in a quotient with an overflow. This is not a problem when working with paper and pencil but is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length.
- ❑ To see this, consider a system that has 5-bit registers. We use one register to hold the divisor and two registers to hold the dividend. From the example shown in the above, we note that the quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor. The quotient is to be stored in a standard 5-bit register, so the overflow bit will require one more flip-flop for storing the sixth bit.
- ❑ This divide-overflow condition must be avoided in normal computer operations because the entire quotient will be too long for transfer into a memory unit that has words of standard length, that is, the same as the length of registers.
- ❑ This condition detection must be included in either the hardware or the software of the computer, or in a combination of the two.

When the dividend is twice as long as the divisor,

i. A divide-overflow condition occurs if the high-order half bits of the dividend constitute a number greater than or equal to the divisor.
ii. A division by zero must be avoided. This occurs because any dividend will be greater than or equal to a divisor which is equal to zero. Overflow condition is usually detected when a special flip-flop is set. We will call it a divide-overflow flip-flop and label it **DVF**.

Hardware Algorithm:

1. The dividend is in A and Q and the divisor in B . The sign of the result is transferred into Qs to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient.

2. A divide-overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A. If $A \geq B$, the divide-overflow flip-flop DVF is set and the operation is terminated prematurely. If $A < B$, no divide overflow occurs so the value of the dividend is restored by adding B to A.

3. The division of the magnitudes starts by shifting the dividend in AQ to the left with the high-order bit shifted into E. If the bit shifted into E is 1, we know that $EA > B$ because EA consists of a 1 followed by n-1 bits while B consists of only n -1 bits. In this case, B must be subtracted from EA and 1 inserted into Qn for the quotient bit.

4. If the shift-left operation inserts a 0 into E, the divisor is subtracted by adding its 2's complement value and the carry is transferred into E . If $E = 1$, it signifies that $A \geq B$; therefore, Qn is set to 1 . If $E = 0$, it signifies that $A < B$ and the original number is restored by adding B to A . In the latter case we leave a 0 in Qn.

This process is repeated again with registers EAQ. After n times, the quotient is formed in register Q and the remainder is found in register A

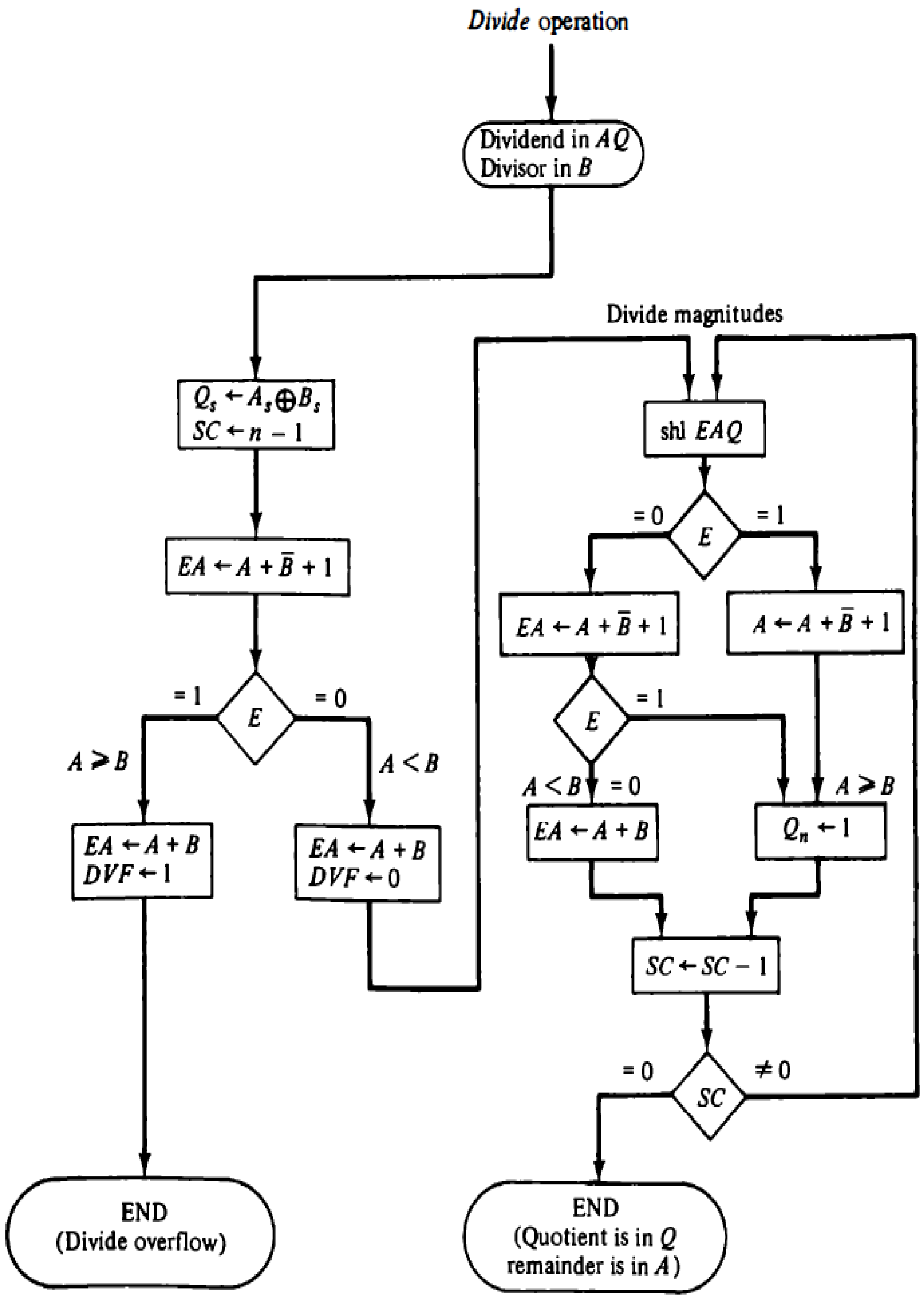


Figure (r): Flowchart for Divide operation

Divisor B = 10001, $\overline{B} + 1$ = 01111

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure (s): Example of Binary Division

Basic Computer Organization and Design

Instruction Codes:

The general purpose digital computer is capable of executing various micro-operations and also can be instructed as to what specific sequence of operations it must perform. The user of a computer can control the process by using a program.

- A **program** is a set of instructions that specify the operations, operands, and the sequence by which processing has to occur.
- 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.
- 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.
- 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.
- The **operation part** of an instruction code specifies the operation to be performed. 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.

Stored Program Organization

The simplest way to organize a computer is to have **one processor register** and an **instruction code format with two parts**. The first part specifies the operation to be performed and the second specifies an address.

- The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

The below figure shows this type of organization.

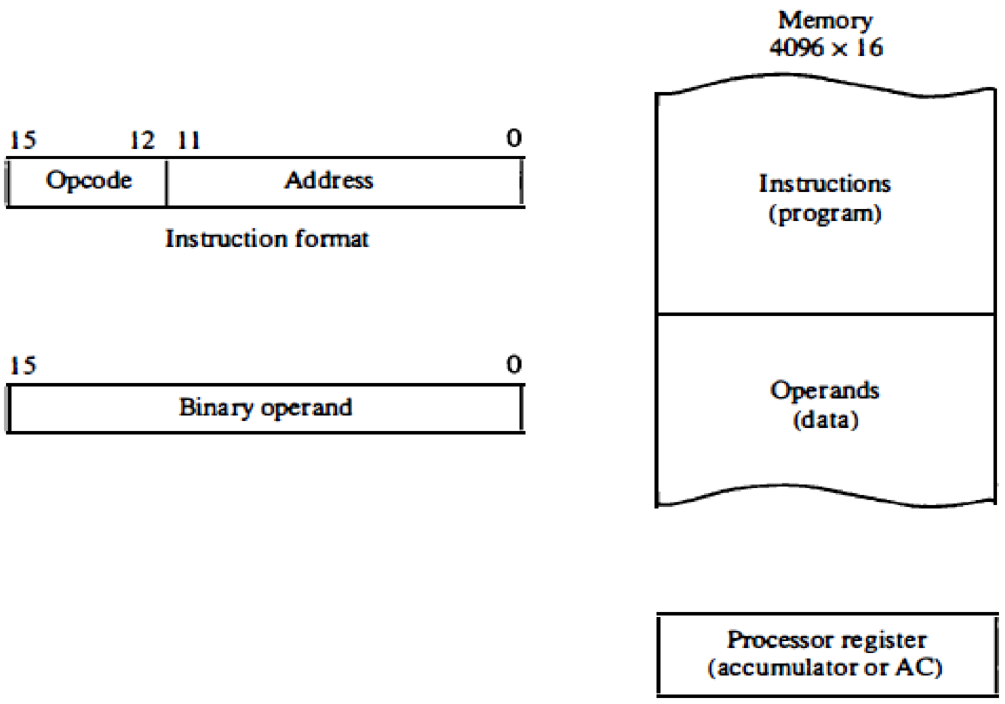


Figure (k): Stored program organization

Instructions are stored in one section of memory and data in another. **EX:** A memory unit with 4096 words, we need 12 bits to specify an address since $2^{12} = 4096$. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.

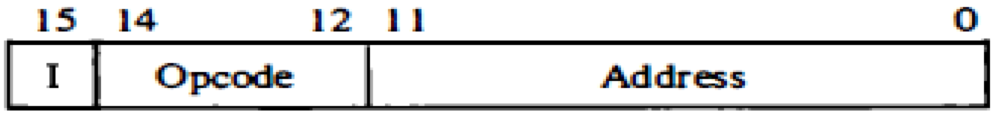
The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code.

- Computers that have a single-processor register usually assign to it the name **accumulator** and label it AC . The operation is performed with the memory operand and the content of AC .
- If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes. For example, operations such as clear AC, complement AC, and increment AC operate on data stored in the AC register. They do not need an operand from memory.

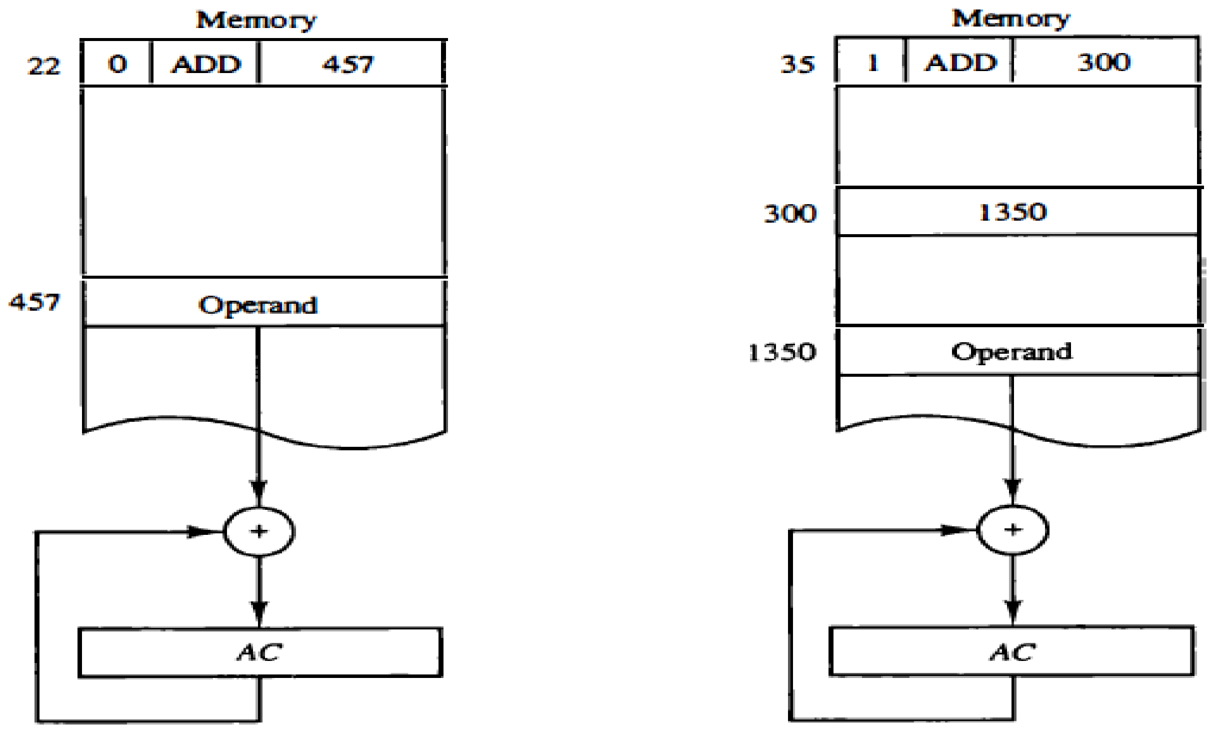
Indirect Address

- _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**.

- When 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 instruction is said to **an indirect address**. One bit of the instruction code can be used to distinguish between a direct and an indirect address.
- An **effective address** is the address of the operand.



(a) Instruction format



(b) Direct address (c) Indirect address

Figure (l): Demonstration of direct and indirect address.

Computer Registers:

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.

The registers available in the computer are shown in the below figure (m) and table (f), a brief description of their function and the number of bits that they contain also given.

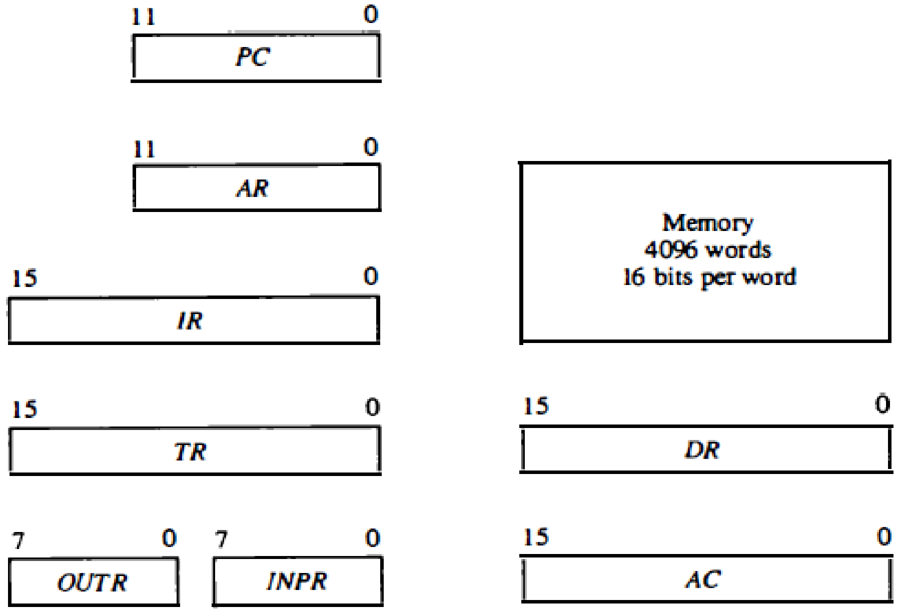


Figure (m): Basic computer registers and memory.

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

Table (f): List of Registers for the Basic computer.

Common Bus System:

- The basic computer has eight registers, a memory unit, and a control unit. Paths must be provided to transfer information from one register to another and between memory and registers.
- The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers. A more efficient scheme for transferring information in a system with many registers is to use a **common bus**.

The connection of the registers and memory of the basic computer to a common bus system is shown in the below figure (n).

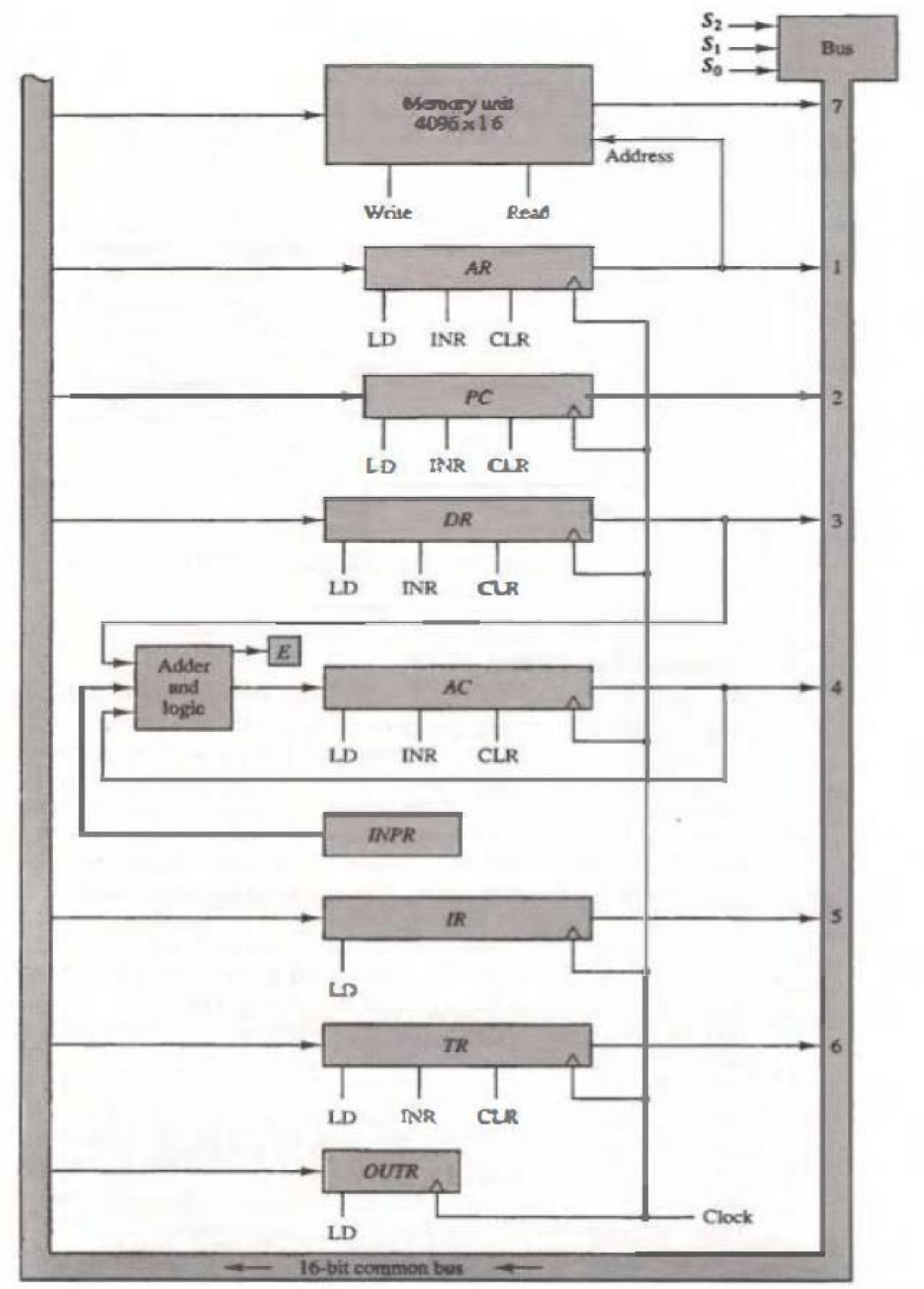


Figure (n): Basic computer registers connected to a common bus.

- The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S_2, S_1, and S_0.

For example1, the number along the output of DR is 3. The 16-bit outputs of DR are placed on the bus lines when $S_2S_1S_0 = 011$ since this is the binary value of decimal 3.
For example2, The memory places its 16-bit output onto the bus when the read input is activated and $S_2S_1S_0 = 111$.

- 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 rnicrooperations

$$DR \leftarrow AC \text{ and } AC \leftarrow DR$$

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.

Computer Instructions:

The basic computer has **three types of instruction code formats**,

1. Memory-reference instruction.
2. Register-reference instruction.
3. An input-output instruction.

Each format has 16 bits. The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.

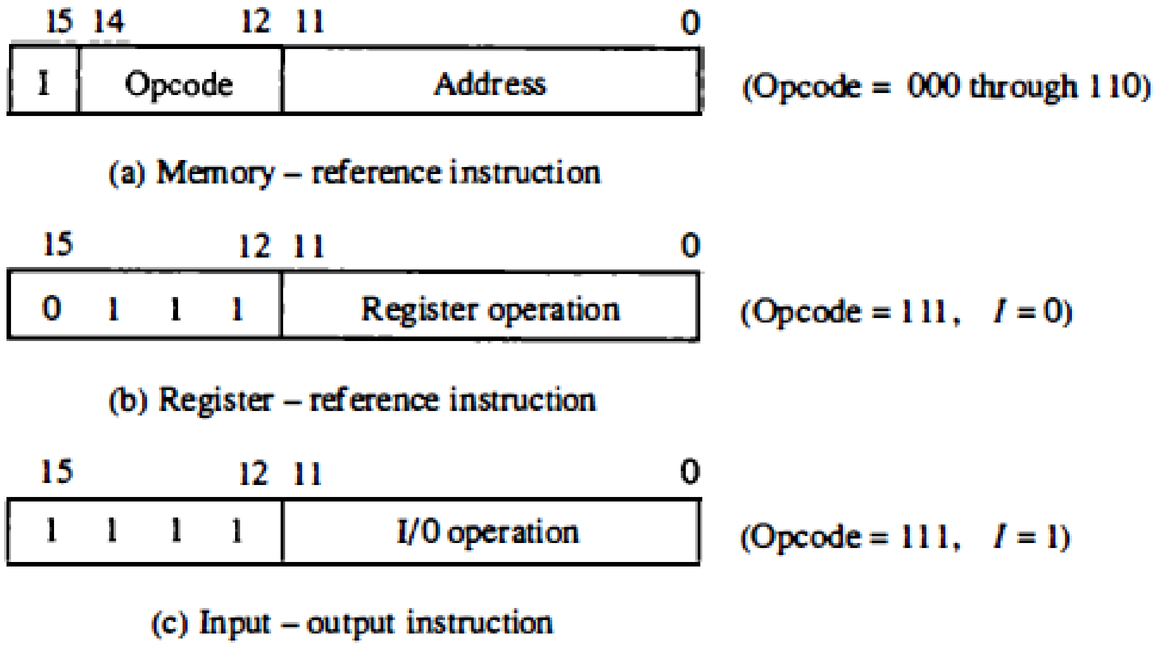


Figure (n): Basic computer instruction formats

The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction.

- If the three opcode bits in positions 12 to 14 are not equal to 111, the instruction is a **memory-reference type** and the bit in position 15 is taken as the addressing mode I. A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I. I = 0 for direct address and I = 1 for indirect address.
- If the 3-bit opcode = 111, control then inspects the bit in position 15. If this bit = 0, the instruction is a **register-reference type**. These instructions use 16 bits to specify an operation.
- If the bit I = 1, the instruction is **an input-output type**. These instructions also use all 16 bits to specify an operation.

The instructions for the computer are listed in Table (g, h, i).

	Hexadecimal code		
Symbol	$I = 0$	$I = 1$	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

Table (g): Memory-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

Table (h): Register-reference 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

Table (i): Input-output instructions

The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction. By using the hexadecimal equivalent we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits.

A) memory-reference instruction has an address part of 12 bits. The address part is denoted by three x's and stand for the three hexadecimal digits corresponding to the 12-bit address. The last bit of the instruction is designated by the symbol I.

i. When I = 0, the last four bits of an instruction have a hexadecimal digit equivalent from 0 (000) to 6 (110) since the last bit is 0.
ii. When I = I, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 (1000) to E (1110) since the last bit is I.

B) Register-reference instructions use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7. The other three hexadecimal digits give the binary equivalent of the remaining 12 bits.
C) The input-output instructions also use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimal F.

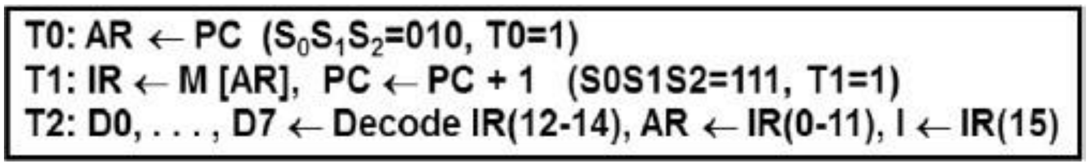
Instruction Set Completeness
A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable. The set of instructions are said to be **complete** if the computer includes a sufficient number of instructions 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.

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 **subcycles 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 rnicrooperations for the fetch and decode phases can be specified by the following register transfer statements.



T0: AR ← PC ($S_0S_1S_2$=010, T0=1)
T1: IR ← M [AR], PC ← PC + 1 (S0S1S2=111, T1=1)
T2: D0, . . . , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)

Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T_0. The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T_1. At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program. At time T_2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR. Note that SC is incremented after each clock pulse to produce the sequence T_0, T_1, and T_2.

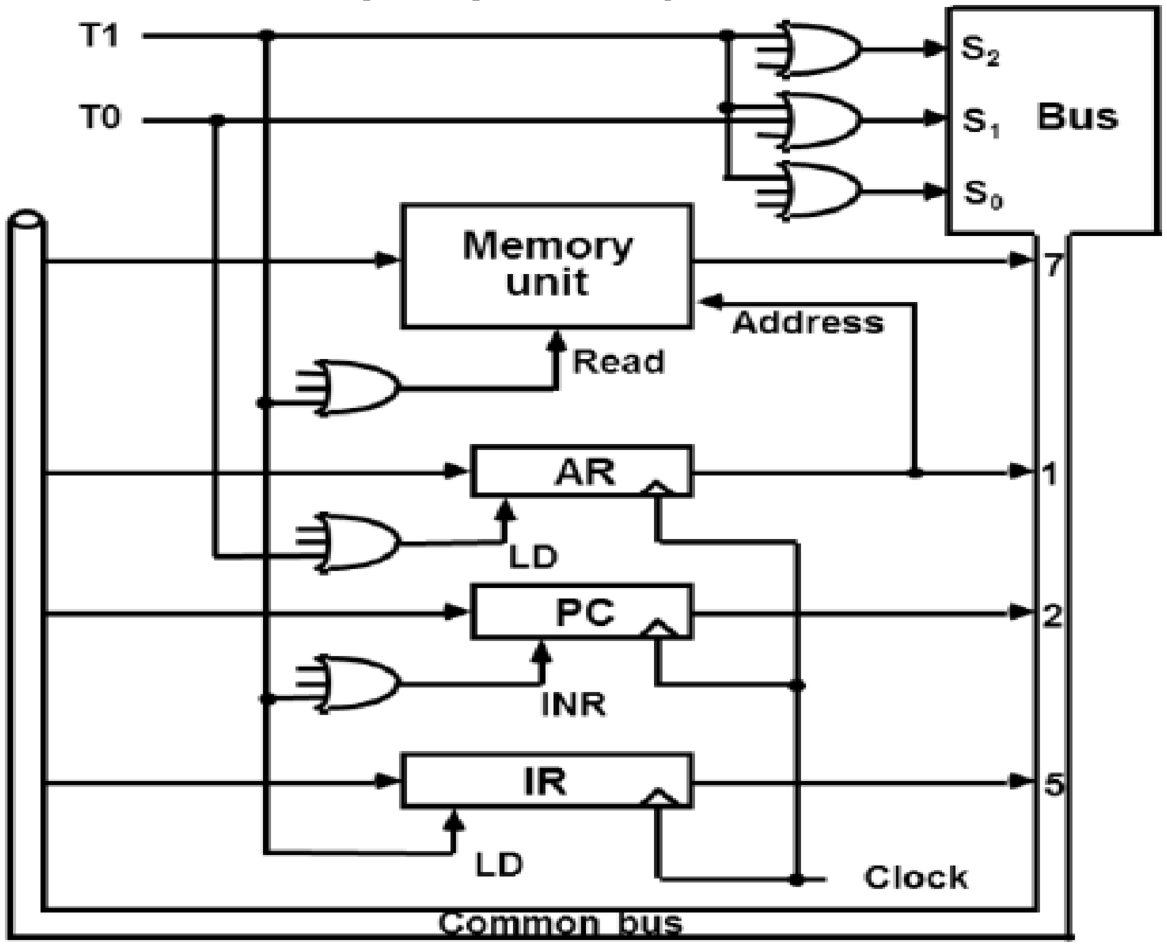


Figure (o): Register transfers for the fetch phase

The above Figure (o) shows how the first two register transfer statements are implemented in the bus system. To provide the data path for the transfer of PC to AR we must apply timing signal T_0 to achieve the following connection:

1. Place the content of PC onto the bus by making the bus selection inputs S_2 S_1 S_0 equal to 010.
2. Transfer the content of the bus to AR by enabling the LD input of AR.

The next clock transition initiates the transfer from PC to AR since $T_0 = 1$.
In order to implement the second statement

T1: IR←M[AR], PC ← PC + 1

It is necessary to use timing signal T_1 to provide the following connections in the bus system.
1. Enable the read input of memory.
2. Place the content of memory onto the bus by making $S_2 S_1 S_0 = 111$.
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

Determine the Type of Instruction

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.

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

$$\begin{aligned} D_7' I T_3&: \quad AR \leftarrow M[AR] \\ D_7' I' T_3&: \quad \text{Nothing} \\ D_7 I' T_3&: \quad \text{Execute a register-reference instruction} \\ D_7 I T_3&: \quad \text{Execute an input–output instruction} \end{aligned}$$

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 T4. 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 $T_0 = 1$.

The flowchart of Figure (p) presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding

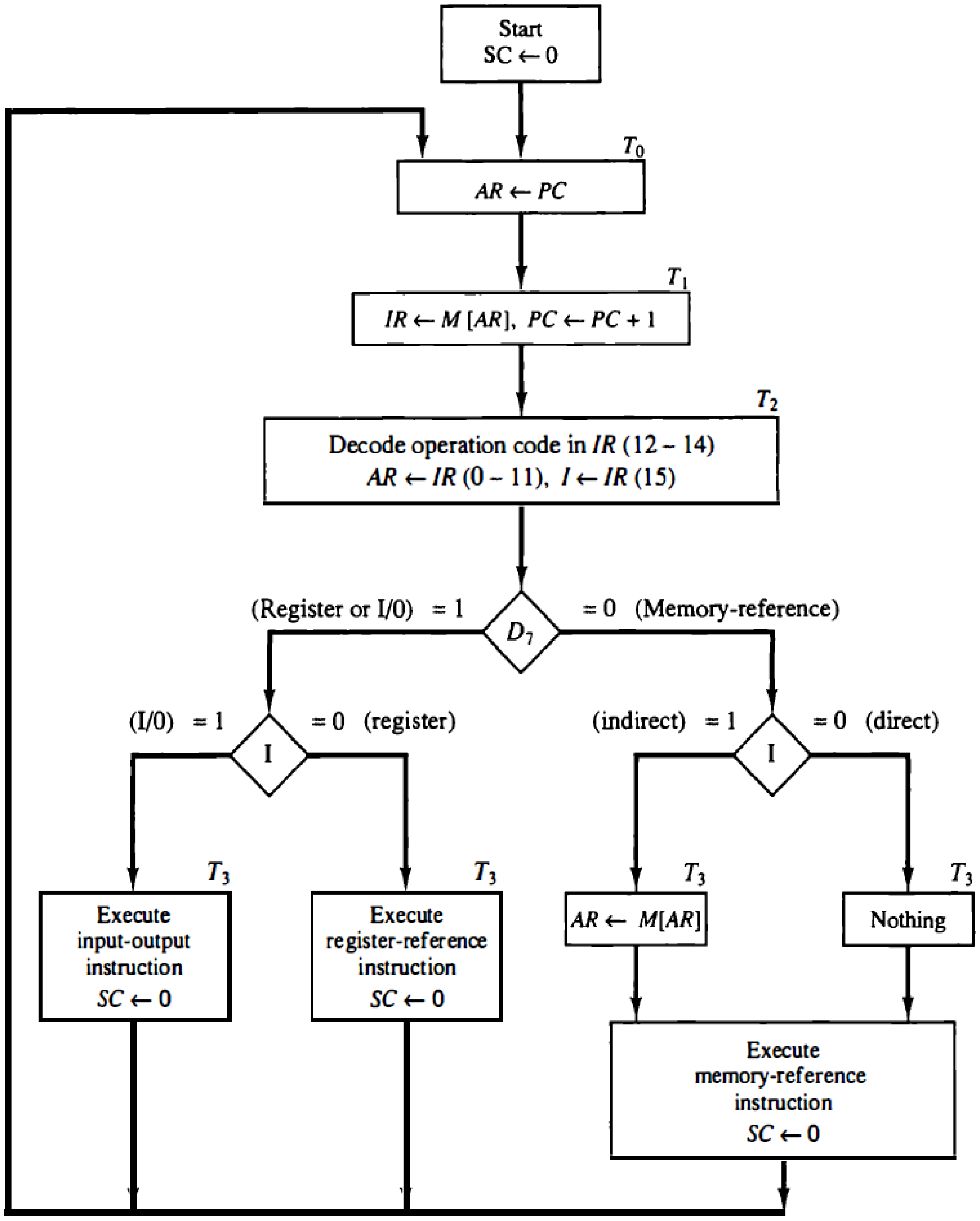

Start
SC ← 0
T0
AR ← PC
T1
IR ← M [AR], PC ← PC + 1
T2
Decode operation code in IR (12 – 14)
AR ← IR (0 – 11), I ← IR (15)
(Register or I/0) = 1
D7
= 0 (Memory-reference)
(I/0) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC ← 0
T3
AR ← M[AR]
T3
Nothing
Execute memory-reference instruction SC ← 0

Figure (p): Flowchart for instruction cycle (initial configuration).

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 .

- Each control function needs the Boolean relation D_7 I' T_3, which we designate for convenience by the symbol r . The control function is distinguished by one of the bits in

IR(0-11). By assigning the symbol B_i to bit i of IR, all control functions can be simply denoted by rB_i.

- **For example**, the instruction CLA has the hexadecimal code 7800, which gives the binary equivalent 0111 1000 0000 0000.
  - i. The first bit is a zero and is equivalent to I'.
  - ii. The next three bits constitute the operation code and are recognized from decoder output D7.
  - iii. Bit 11 in IR is 1 and is recognized from B_{11}.

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

$D_7 I' T_3 = r$ (common to all register-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:	$AC \leftarrow AC + 1$	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

<u>Table (j):</u> Execution of Register-Reference Instructions

Memory-Reference Instructions:

- ✓ The below Table (k) lists the seven memory-reference instructions. 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.
- ✓ The effective address of the instruction is in the address register AR and was placed there during timing signal T2 when I = 0, or during timing signal T3 when I = 1. The execution of the memory-reference instructions starts with timing signal T4.

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

<u>Table (k):</u> Memory-Reference Instructions

<u>AND</u> : 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 rnicrooperations that execute this instruction are:

D_0T_4: DR ← M[AR]
D_0T_5: AC ← AC Λ DR, SC ← 0

ADD : 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,., is transferred to the E (extended accumulator) flip-flop. The rnicrooperations needed to execute this instruction are:
D_1T_4 : DR ← M[AR]
D_1T_5: AC ← AC + DR, E ← Cout, SC ← 0

LDA: Load to AC
This instruction transfers the memory word specified by the effective address to AC . The rnicrooperations needed to execute this instruction are:
D_2T_4: DR ← M[AR]
D_2T_5: AC ← DR, SC ← 0

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 microoperation:

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

BUN: Branch Unconditionally
- This instruction transfers the program to the instruction specified by the effective address.
- 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 rnicrooperation:

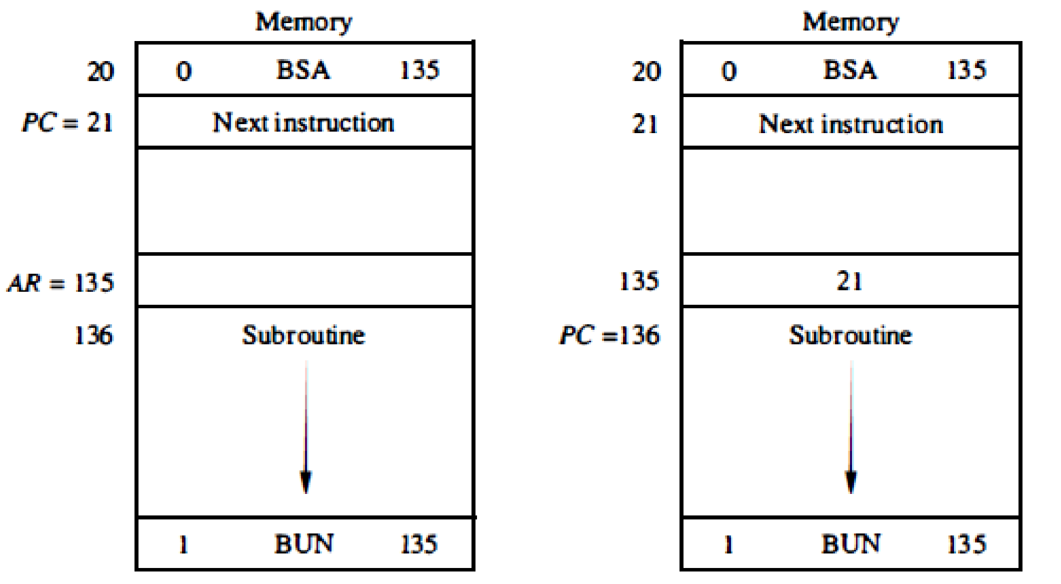
BSA: Branch and Save Return Address
This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) into a memory location specified by the effective address. The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine.

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

BSA: Branch and Save Return Address
EX:
The BSA instruction is assumed to be in memory at address 20. The I bit is 0 and the address part of the instruction has the binary equivalent of 135. After the fetch and decode phases, PC contains 21, which is the address of the next instruction in the program (referred to as the return address). AR holds the effective address 135. This is shown in part (a) of the figure. The BSA instruction performs the following numerical operation:

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

The result of this operation is shown in part (b) of the figure. The return address 21 is stored in memory location 135 and control continues with the subroutine program starting from address 136. The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine. When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21. When the BUN instruction is executed, the effective address 21 is transferred to PC. The next instruction cycle finds PC with the value 21, so control continues to execute the instruction at the return address.



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

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:

$$D_5T_4: \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$$
$$D_5T_5: \quad PC \leftarrow AR, \quad SC \leftarrow 0$$

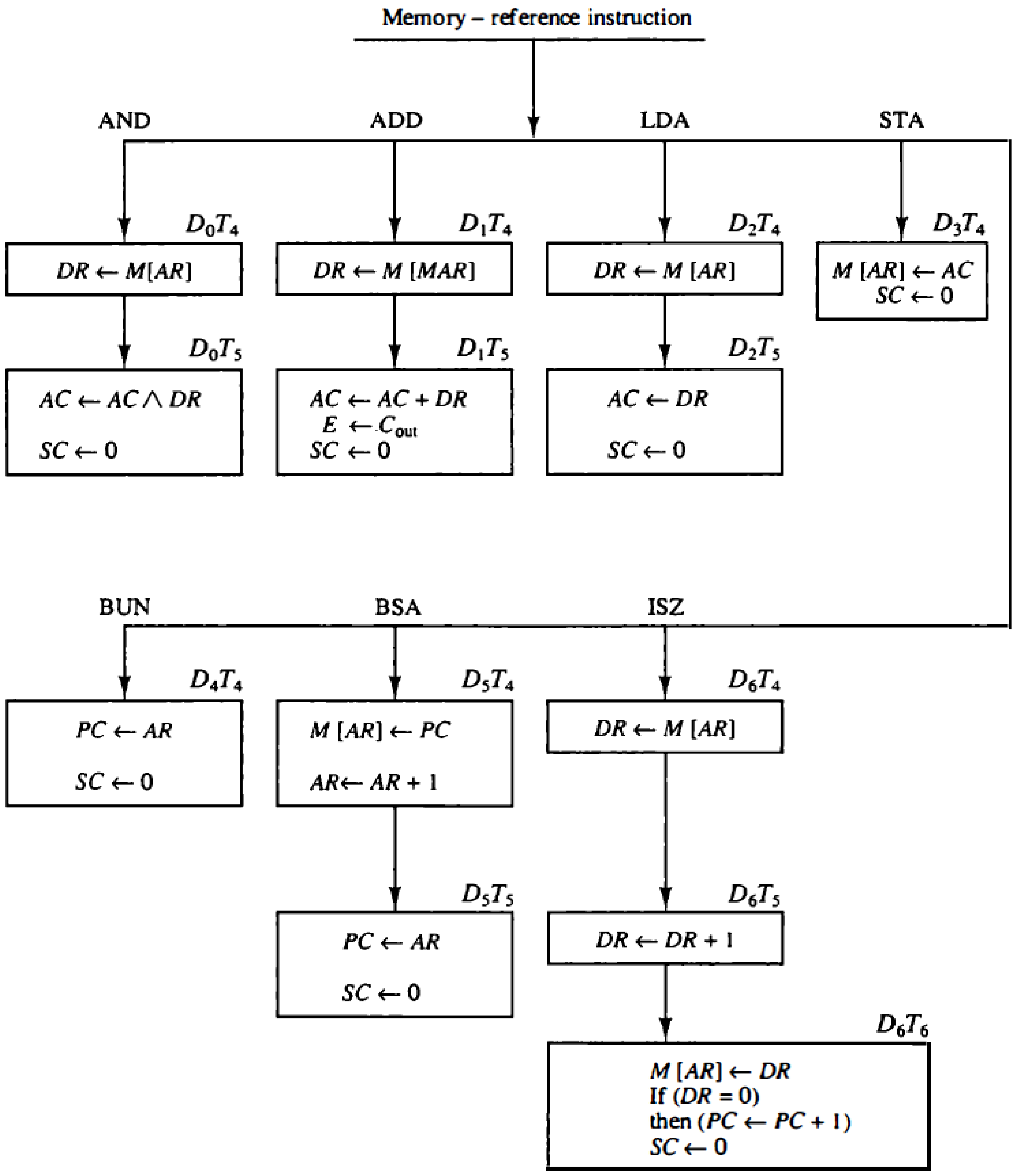
Timing signal T4 initiates a memory write operation, places the content of PC onto the bus, and enables the INR input of AR . The memory write operation is completed and AR is incremented by the time the next clock transition occurs. The bus is used at T5 to transfer the content of AR to PC.

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.

$$D_6T_4: \quad DR \leftarrow M[AR]$$
$$D_6T_5: \quad DR \leftarrow DR + 1$$
$$D_6T_6: \quad M[AR] \leftarrow DR, \quad \text{if } (DR = 0) \text{ then } (PC \leftarrow PC + 1), \quad SC \leftarrow 0$$

A flowchart showing all microoperations for the execution of the seven memory-reference instructions is shown in Figure (q). The control functions are indicated on top of each box. The microoperations that are performed during time T4, T5, or T6, depend on the operation code value. The sequence counter SC is cleared to 0 with the last timing signal in each case. This causes a transfer of control to timing signal T0 to start the next instruction cycle.



<u>Figure (q):</u> Flowchart for Memory-reference instructions

Input-Output and Interrupt:

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

The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR . These two registers communicate with a communication interface serially and with the AC in parallel.

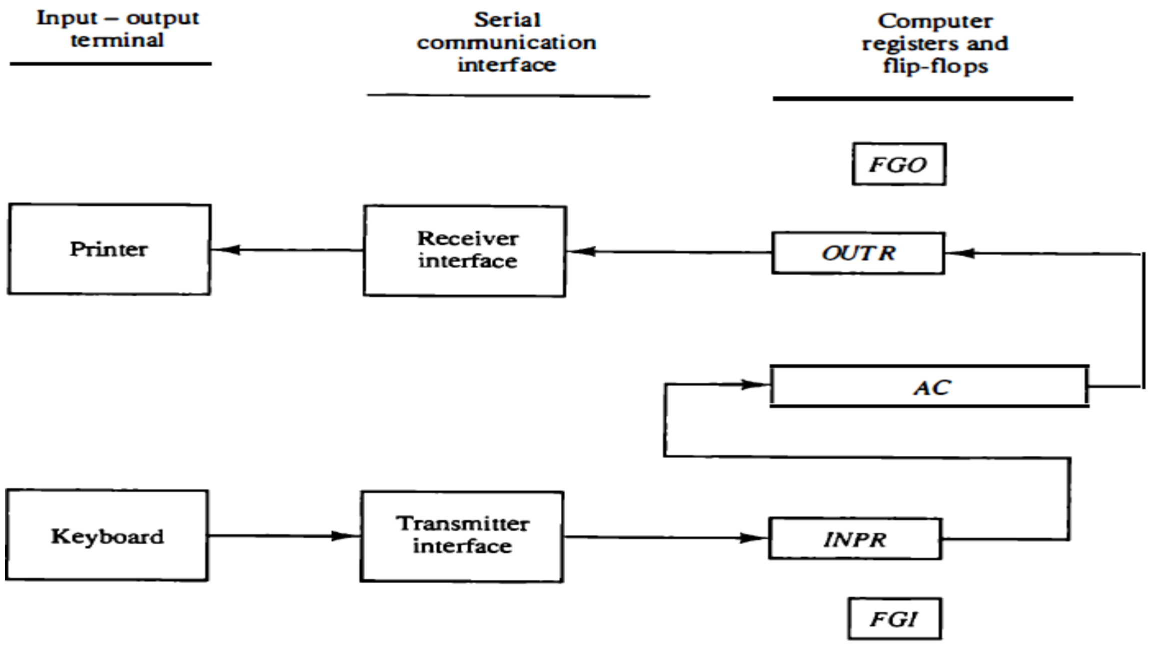


Figure (r): Input-Output configuration

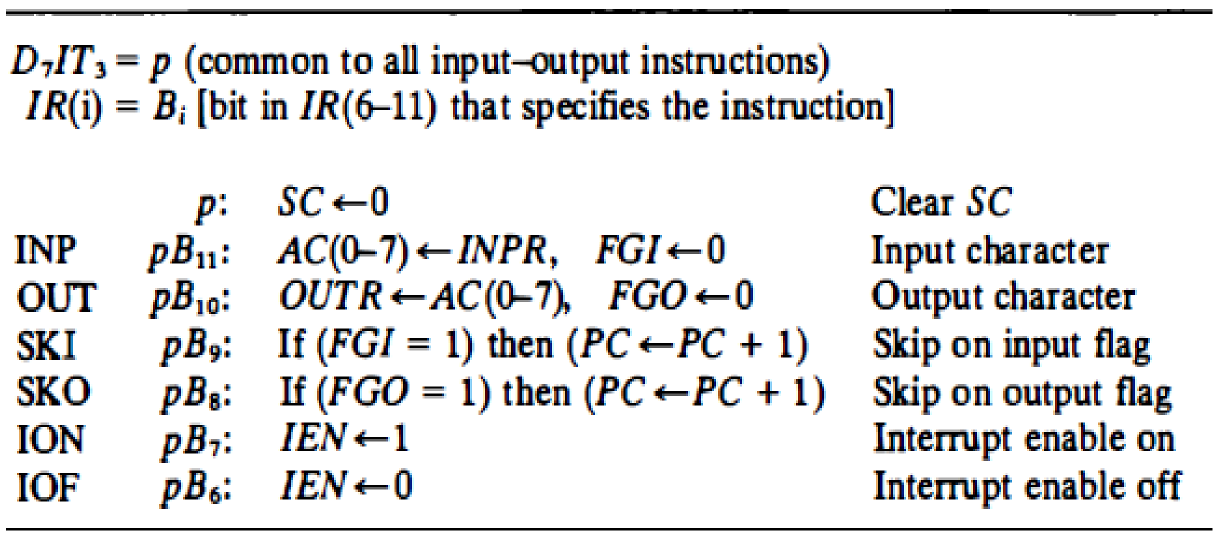
The process of information transfer is as follows: Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR cannot be changed by striking 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.

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

Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility.

Input-output instructions have an operation code 1111 and are recognized by the control when D7 = 1 and I = 1. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input-output instructions are listed in Table (l).



$D_7IT_3 = p$ (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

Table (l): Input-Output instructions

Program Interrupt

The process of communication discussed so far 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 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 1µs. 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.
- 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.
- The interrupt enable flip-flop lEN can be set and cleared with two instructions (IOF and ION instructions).

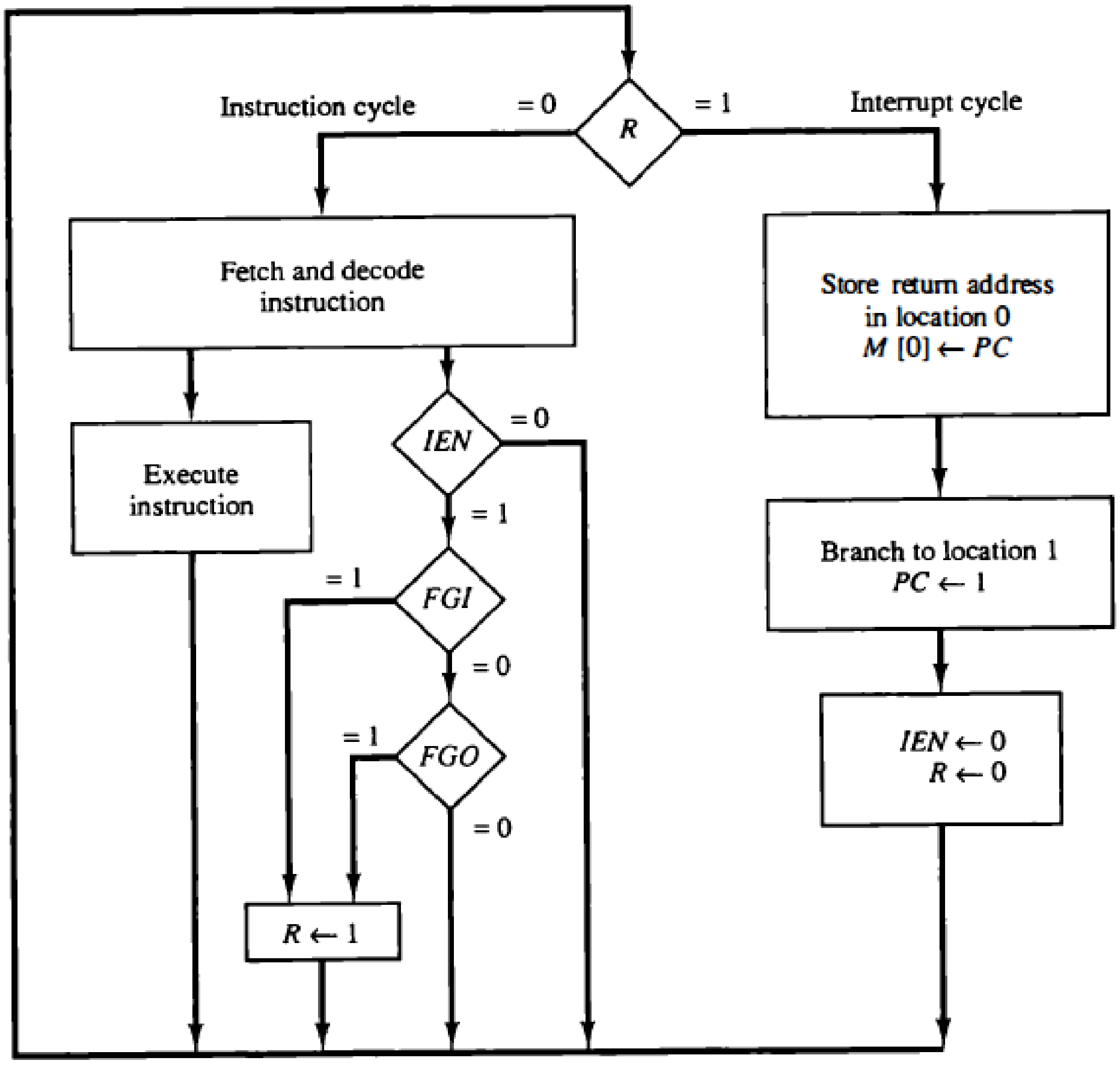


Figure (s): Flowchart for interrupt cycle

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Figure (s).

- An interrupt flip-flop R is included in the computer. When R = 0, the computer goes through an instruction cycle.
- During the execute phase of the instruction cycle lEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle.
- If lEN is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while lEN = 1, flip-flop R is set to 1.
- At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

The interrupt cycle is a hardware implementation of a **branch and save return address(BSA)** operation.

EX:

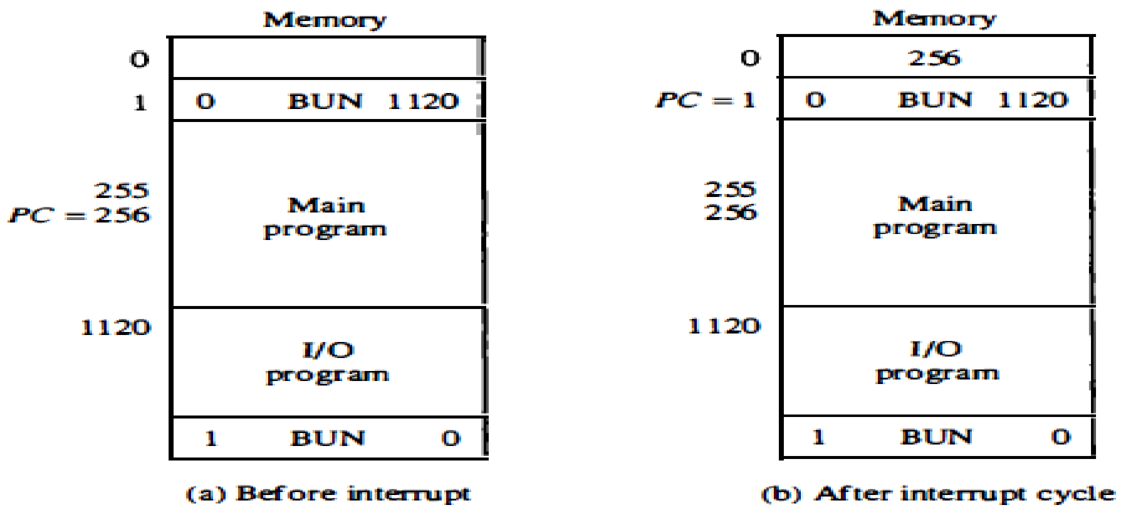


Figure (t): Demonstration of Interrupt Cycle

An example that shows what happens during the interrupt cycle is shown in Figure (t). Suppose that an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in PC. The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Figure (a).

When control reaches timing signal T0 and finds that R = 1, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. The branch instruction at address 1 causes the program to transfer to the input-output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the program returns to the location where it was interrupted. This is shown in Figure (b).

Interrupt Cycle

The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop R is equal to 1. This flip-flop is set to 1 if lEN = 1 and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T0, T1 or T2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement:

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

During the first timing signal AR is cleared to 0, and the content of PC is transferred to the temporary register TR. With the second timing signal, the return address is stored in memory at location 0 and PC is cleared to 0. The third timing signal increments PC to 1, clears lEN and R, and control goes back to T0 by clearing SC to 0. The beginning of the next instruction cycle has the condition R' T0 and the content of PC is equal to 1. The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

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

Complete Computer Description:

The final flowchart of the instruction cycle, including the interrupt cycle for the basic computer, is shown in the below figure (u). The interrupt flip-flop R may be set at any time during the indirect or execute phases. Control returns to timing signal T_0 after SC is cleared to 0.

- If R = 1, the computer goes through an interrupt cycle.
- If R = 0, the computer goes through an instruction cycle.

If the instruction is one of the memory-reference instructions, the computer first checks if there is an indirect address and then continues to execute the decoded instruction. If the instruction is one of the register-reference instructions, it will be executed. If it is an input-output instruction, it will be executed.

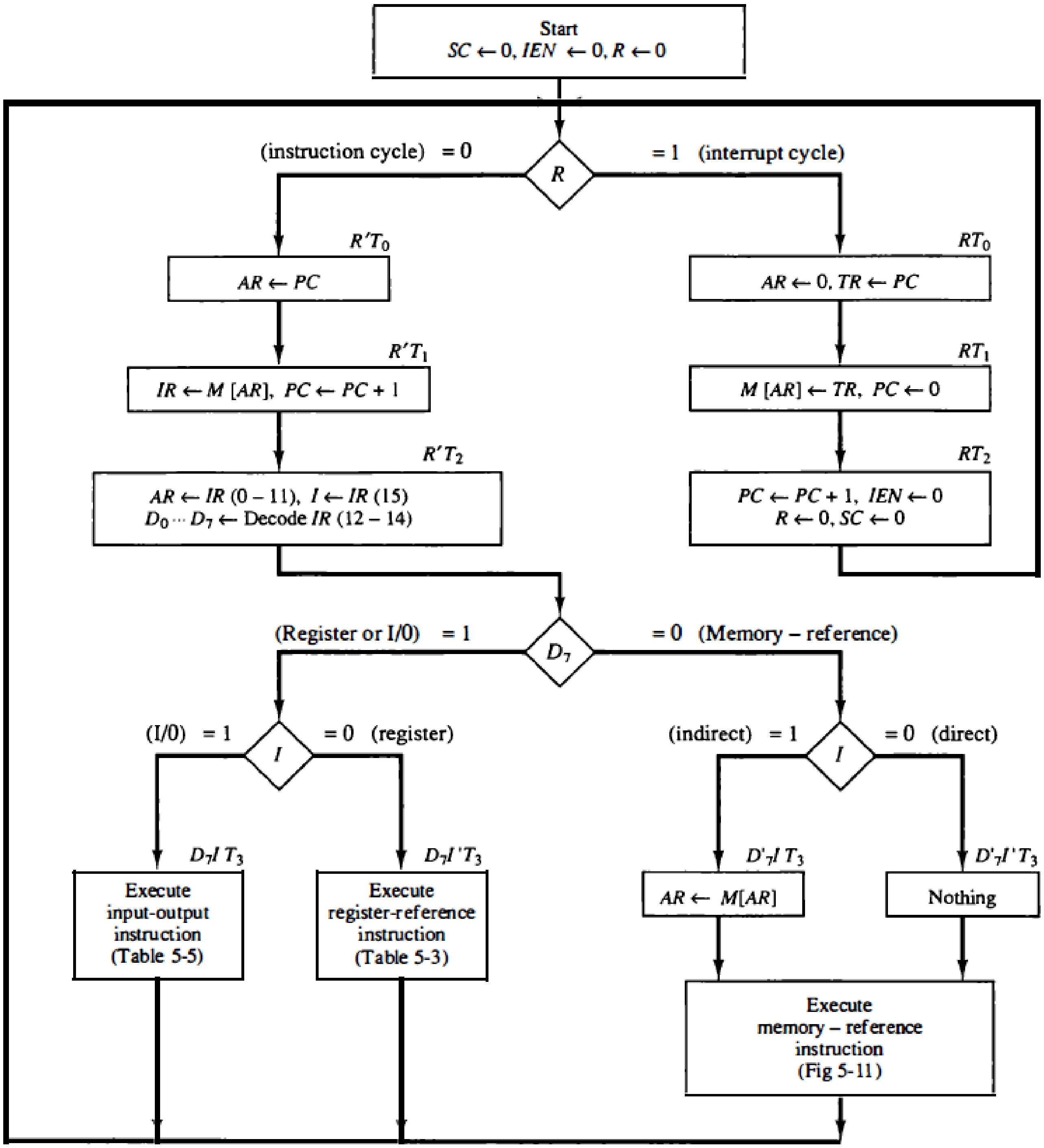
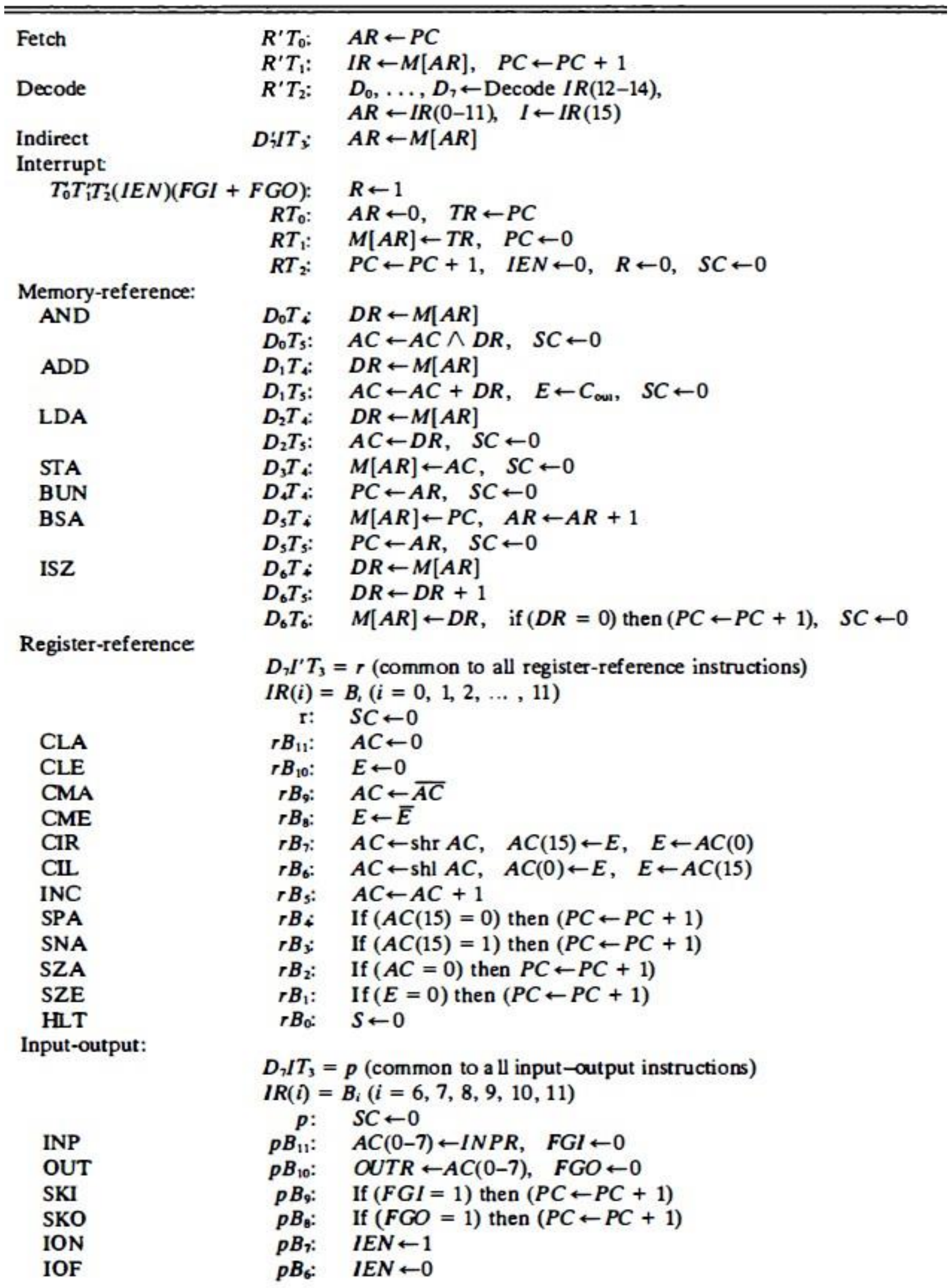


Figure (u): Flowchart for computer operation



Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR], \quad PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$,
		$AR \leftarrow IR(0\text{–}11), \quad I \leftarrow IR(15)$
Indirect	$D_7'IT_3$:	$AR \leftarrow M[AR]$
Interrupt:		
$T_0'T_1'T_2'(IEN)(FGI + FGO)$:		$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0, \quad TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR, \quad PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1, \quad IEN \leftarrow 0, \quad R \leftarrow 0, \quad SC \leftarrow 0$
Memory-reference:		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR, \quad SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR, \quad E \leftarrow C_{out}, \quad SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR, \quad SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC, \quad SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR, \quad SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR, \quad SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1), \quad SC \leftarrow 0$
Register-reference:		
	$D_7I'T_3 = r$ (common to all register-reference instructions)	
	$IR(i) = B_i \ (i = 0, 1, 2, \ldots, 11)$	
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow \overline{AC}$
CME	rB_8:	$E \leftarrow \overline{E}$
CIR	rB_7:	$AC \leftarrow$ shr $AC, \quad AC(15) \leftarrow E, \quad E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow$ shl $AC, \quad AC(0) \leftarrow E, \quad E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Input-output:		
	$D_7IT_3 = p$ (common to all input–output instructions)	
	$IR(i) = B_i \ (i = 6, 7, 8, 9, 10, 11)$	
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{–}7) \leftarrow INPR, \quad FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7), \quad FGO \leftarrow 0$
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$

Table (m): Control functions and microoperations for the Basic computer

Instead of using a flowchart, we can describe the operation of the computer with a list of register transfer statements. This is done by accumulating all the control functions and microoperations in one table, as shown in the below Table (m).

The register transfer statements in this table describe in a concise form the internal organization of the basic computer. They also give all the information necessary for the design of the logic circuits of the computer.

A register transfer language is useful not only for describing the internal organization of a digital system but also for specifying the logic circuits needed for its design.