Computer Arithmetic: The Addition, subtraction, multiplication and division are the four basic arithmetic operations. These operations addition, subtraction, multiplication, and division can be performed on the following types of data:

1. Fixed-point binary data (Signed Magnitude and Signed 2's complement representation)
2. Floating-point binary data
3. Binary-coded decimal (BCD) data

Addition and Subtraction: There are three ways of representing negative fixed-point binary numbers:

1. Signed-magnitude
2. Signed one's complement
3. Signed two's complement.

Most computers use the signed-2's complement representation when performing arithmetic operations with integers. For floating-point operations, most computers use the signed-magnitude representation for the Mantissa.

Addition and Subtraction with signed-magnitude Data: Consider the magnitude of the two numbers by A and B. When the signed numbers are added or subtracted, there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the following table.

TABLE 10-1 Addition and Subtraction of Signed-Magnitude Numbers

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

Addition (Subtraction) Algorithm: When the signs of A and B are identical (different), add the two magnitudes and attach the sign of A to the result. When the signs of A and B are different (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. If the two magnitudes are equal/ subtract B from A and make the sign of the result positive.

Hardware Implementation: Addition and Subtraction with Signed-Magnitude Data Hardware Design 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.

The addition of A plus B is done through the parallel adder. 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 1's complement of B is applied to the adder, the input carry is 1, and output S = A + B +1. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction, A - B. The S (sum) output of the adder is applied to the input of the A register.

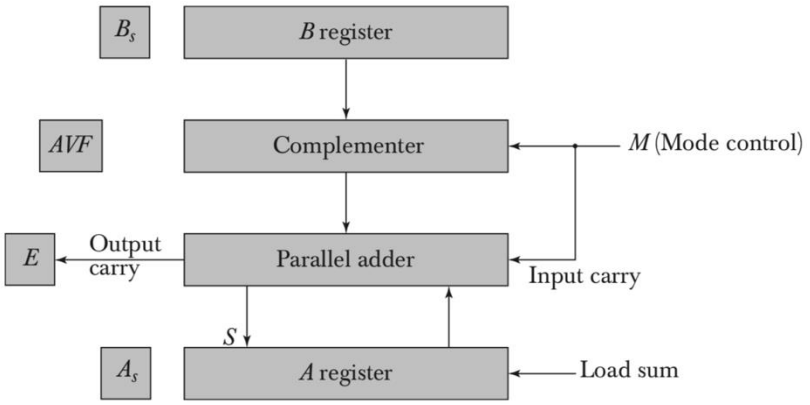


Figure 10-1 Hardware for signed-magnitude addition and subtraction.

Hardware Algorithm:

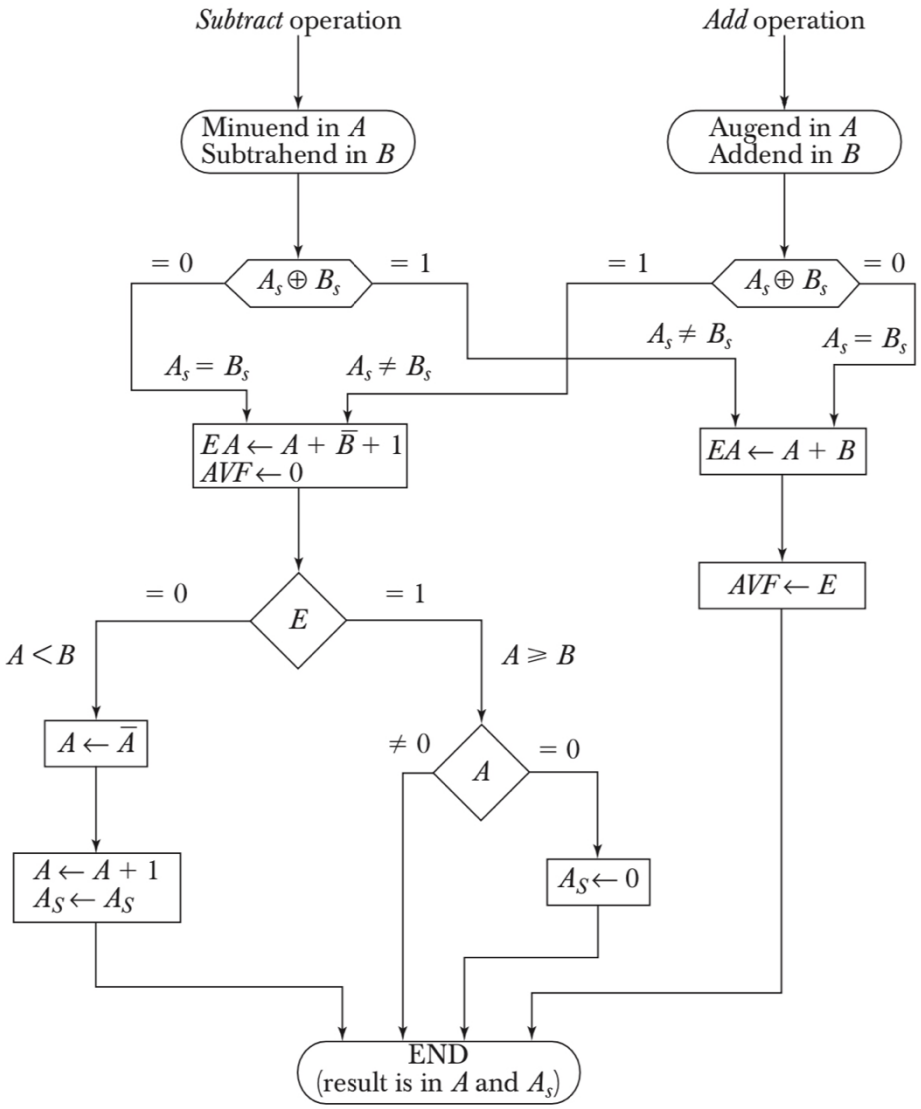


Figure 10-2 Flowchart for add and subtract operations.

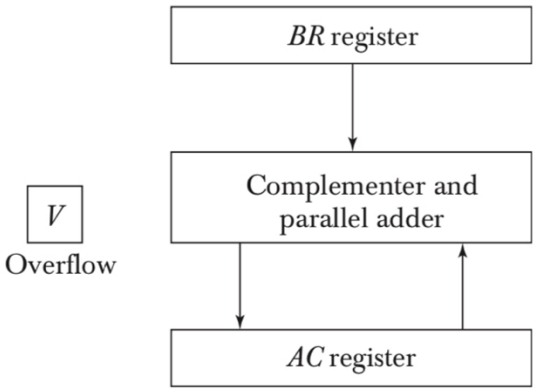
The flowchart for the addition and subtraction is presented in above figure.

- The two signs ***As* and *Bs*** are compared by an exclusive-OR gate.
- If the output of the gate is 0, the signs are identical; if it is 1, the signs are different.
- For an *add* operation, identical signs indicate that the magnitudes be added.
- For a *subtract* operation, different signs indicate that the magnitudes be subtracted.
- The magnitudes are added with a micro-operation EA ←A+B, where EA is a register that combines E and A.
- The carry in E after the addition indicates an overflow if it is equal to 1.
- If E = 1, then A ≥ B.
  - However, if A = 0, then the sign is made positive.
- If E = 0, then A < B and 2's complement of A is stored back into A and sign for A is complemented.

Addition and Subtraction with Signed-2's Complement Algorithm: The addition of two numbers in signed 2's complement form consists of adding the numbers with signed 2's complement data. The subtraction is performed by adding 2's complement of the subtrahend to the minuend.

When two numbers of n digits are added and sum occupies n+1 digits, we say that an overflow is occurred. An overflow can be detected by observing the carry into the sign bit position and carry out of the sign bit position. When the two carriers are applied to exclusive-OR gates, the overflow is detected when the output of the gate is equal to 1. The register configuration for the hardware implementation is as shown below.

Hardware for signed-2's complement addition and subtraction.



In this case, unlike signed magnitude data sign bits are not separated from the rest of the registers. The left most 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 (flowchart) for adding and subtracting two binary numbers in signed 2's complement representation is as shown below.

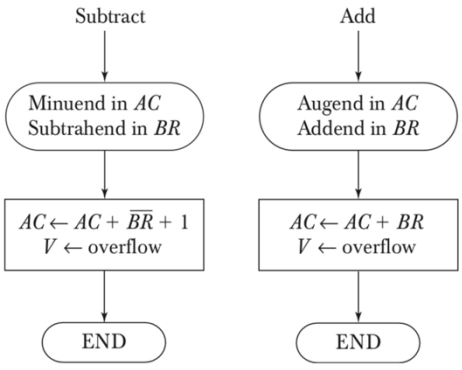


Figure 10-4 Algorithm for adding and subtracting numbers in signed-2's complement representation.

The sum is obtained by adding the contents of AC and BR (including their sign bits). The overflow bit is set to 1 if there is an overflow, else it is set to 0. The subtraction is obtained by adding the content of AC to the 2‘s complement of BR.

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

23	10111	Multiplicand
19	× 10011	Multiplier
	10111	
	10111	
	00000 +	
	00000	
	10111	
437	110110101	Product

This process looks at successive bits of the multiplier, least significant bit first. If the multiplier bit is 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 numbers. Finally, the numbers are added and their sum produces the product.

The sign of the product is determined from the signs of the multiplicand and multiplier. If they are same, the sign of the product is positive. If they are different, the sign of the product is negative.

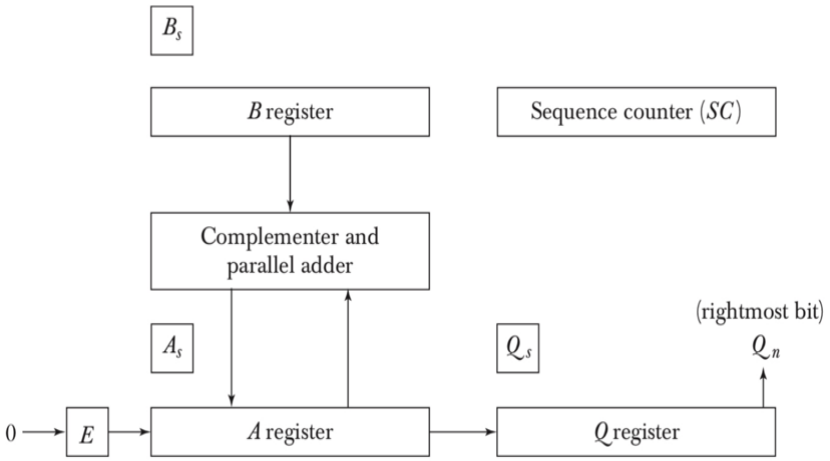
Multiplication is performed as shown below:

1. Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.
2. The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, then the partial product is the multiplicand.
3. The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
4. The multiplication of two n-bit binary integers results in a product of up to 2n bits in length.

Hardware implementation for Multiplication with Signed Magnitude Data: Initially the multiplier is stored in Q register and its sign is Q_s. The sequence counter is initially set to a number which is equal to number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the contents of the counter reaches zero, the product is formed and the process stops.

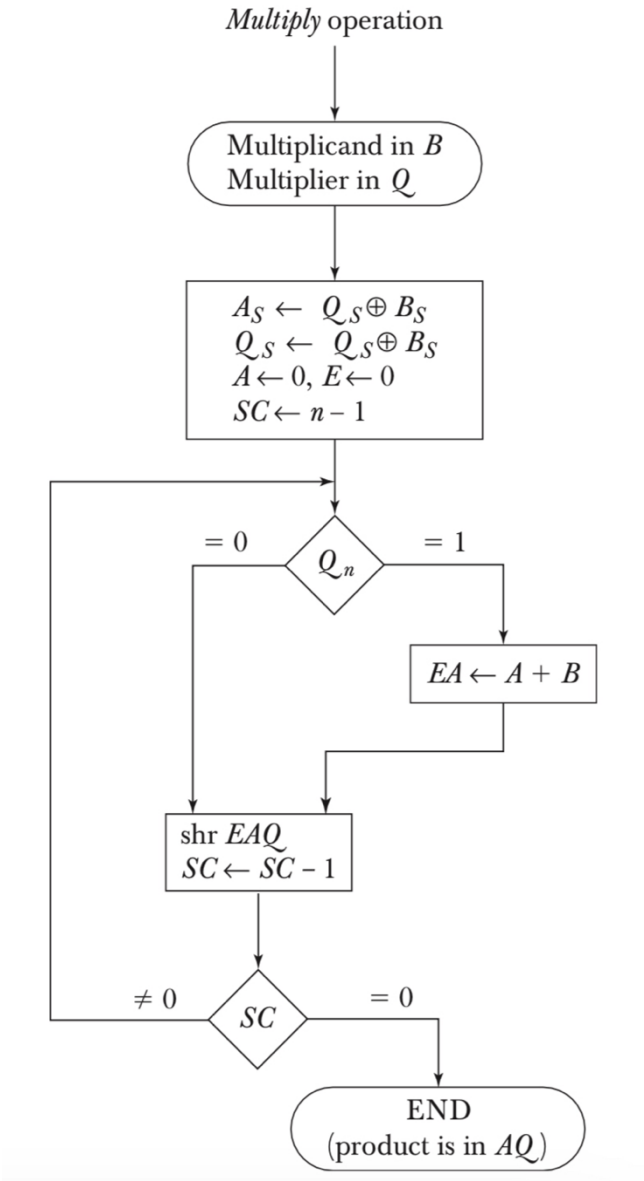
Initially, the multiplicand is in register B and the multiplier is in Q. The sum of A and B forms a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to right. This shift will be denoted by shr EAQ. The least significant bit of A is shifted into 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, pursuing 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.

Figure 10-5 Hardware for multiply operation.



Hardware Algorithm: The following flowchart represents hardware multiply algorithm.

Figure 10-6 Flowchart for multiply operation.



Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s, 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 Q_n is tested. If it is a 1, the multiplicand in B is added to the present partial product in A. If it is a 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. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q , with A holding the most significant bits and Q holding the least significant bits.

Example: 23 * 19 = 437

TABLE 10-2 Numerical Example for Binary Multiplier

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 $AQ = 0110110101$				

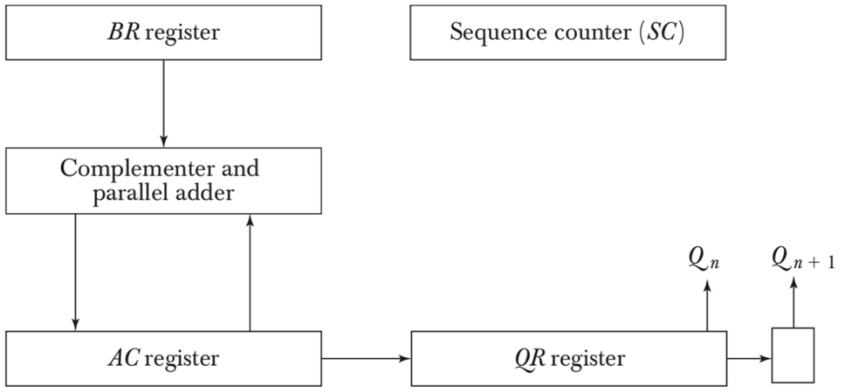
Multiplication with Signed-2's Complement data (Booth's algorithm): Booth algorithm multiplies binary integers in signed 2's complement representation. If the numbers are represented in signed 2's complement then we can multiply them by using Booth algorithm.

Booth algorithm needs examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand added to the partial product, subtracted from the partial product, or left unchanged by the following rules:

1. The multiplicand is subtracted from the partial product when we get the first least significant 1 in a string of 1's in the multiplier.(when $Q_nQ_{n+1}=10$)
2. The multiplicand is added to the partial product when we get the first Q (provided that there was a previous 1) in a string of 0's in the multiplier.(when $Q_nQ_{n+1}=01$)
3. The partial product does not change when the multiplier bit is the same as the previous multiplier bit.(when $Q_nQ_{n+1}=11$ or 00)

Hardware for Booth Algorithm: Booth Algorithm use registers 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.

Figure 10-7 Hardware for Booth algorithm.



Flow Chart for Booth Algorithm:

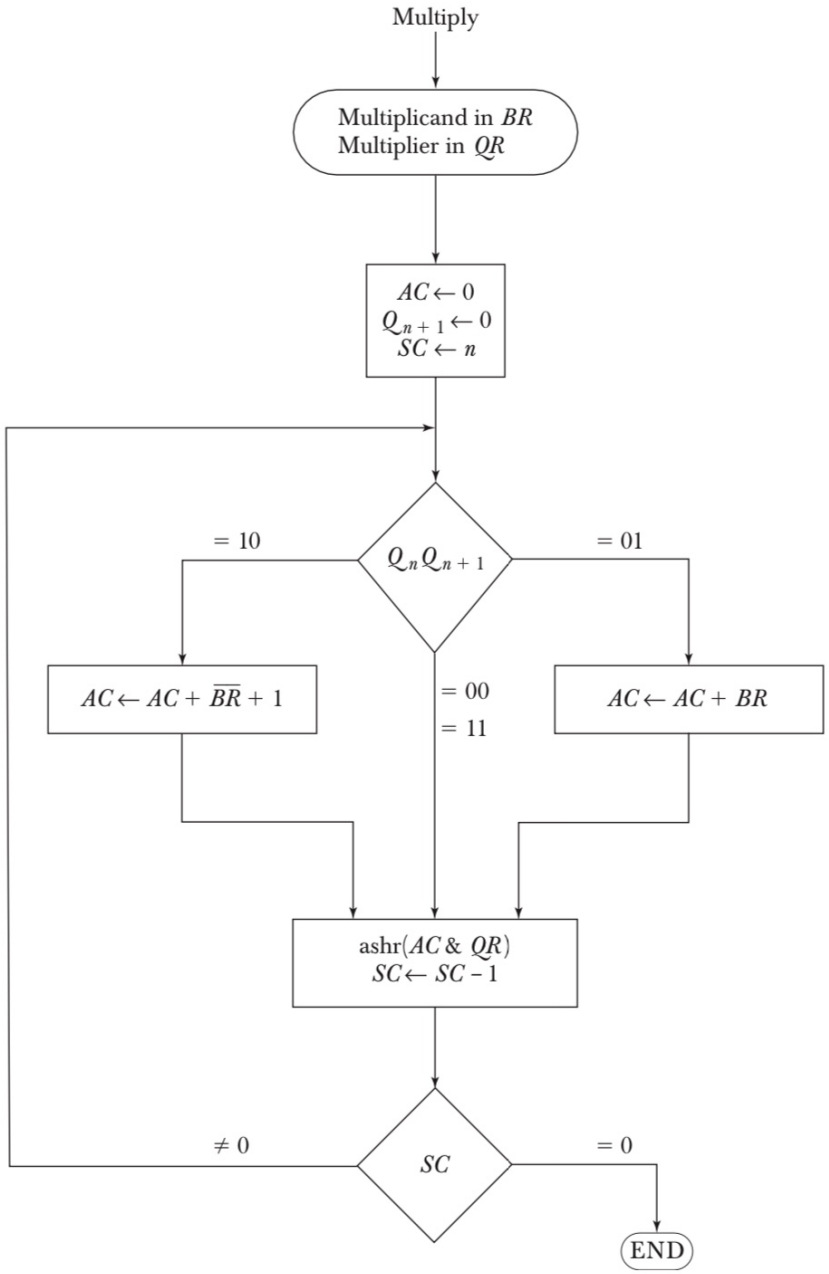


Figure 10-8 Booth algorithm for multiplication of signed-2's complement numbers.

Initially, multiplicand is in BR and multiplier is in QR. 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 int the multiplier. The two bits of the multiplier in Q_n and Q_{n+1} are inspected. If the two bits are equal to 10, it performs a subtraction of the multiplicand from the partial product in AC. If the two bits are equal to 01, it performs addition of the multiplicand to the partial product in AC. When the two bits are equal, the partial product does not change. 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.

Example: (-9) * (-13) = +117

TABLE 10-3 Example of Multiplication with Booth Algorithm

Q_nQ_{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

Division: The divisor is stored in the B register and the double length dividend is stored in registers A and Q. In division algorithms there is a possibility of occurrence of divide overflow. The divide overflow occurs if the high order half bits of the dividend is greater than or equal to the divisor. If there is no overflow, the dividend is shifted left and the divisor is subtracted by adding its 2's complement value. The relative magnitude is available in E. If E=1, a quotient bit 1 is inserted in Q_n, and the partial remainder is shifted left, and repeat the process. If E=0, the quotient Q_n remains 0. The value of B is then added to restore the partial remainder in A. the partial remainder is shifted to the left and the process is repeated again. Finally the quotient is in Q and remainder is in A.

Hardware implementation for division operation is similar to multiplication operation.

Hardware Algorithm (Restore Method): The dividend is in A and Q and the divisor in B. The sign of the result is transferred into Q_s to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient.

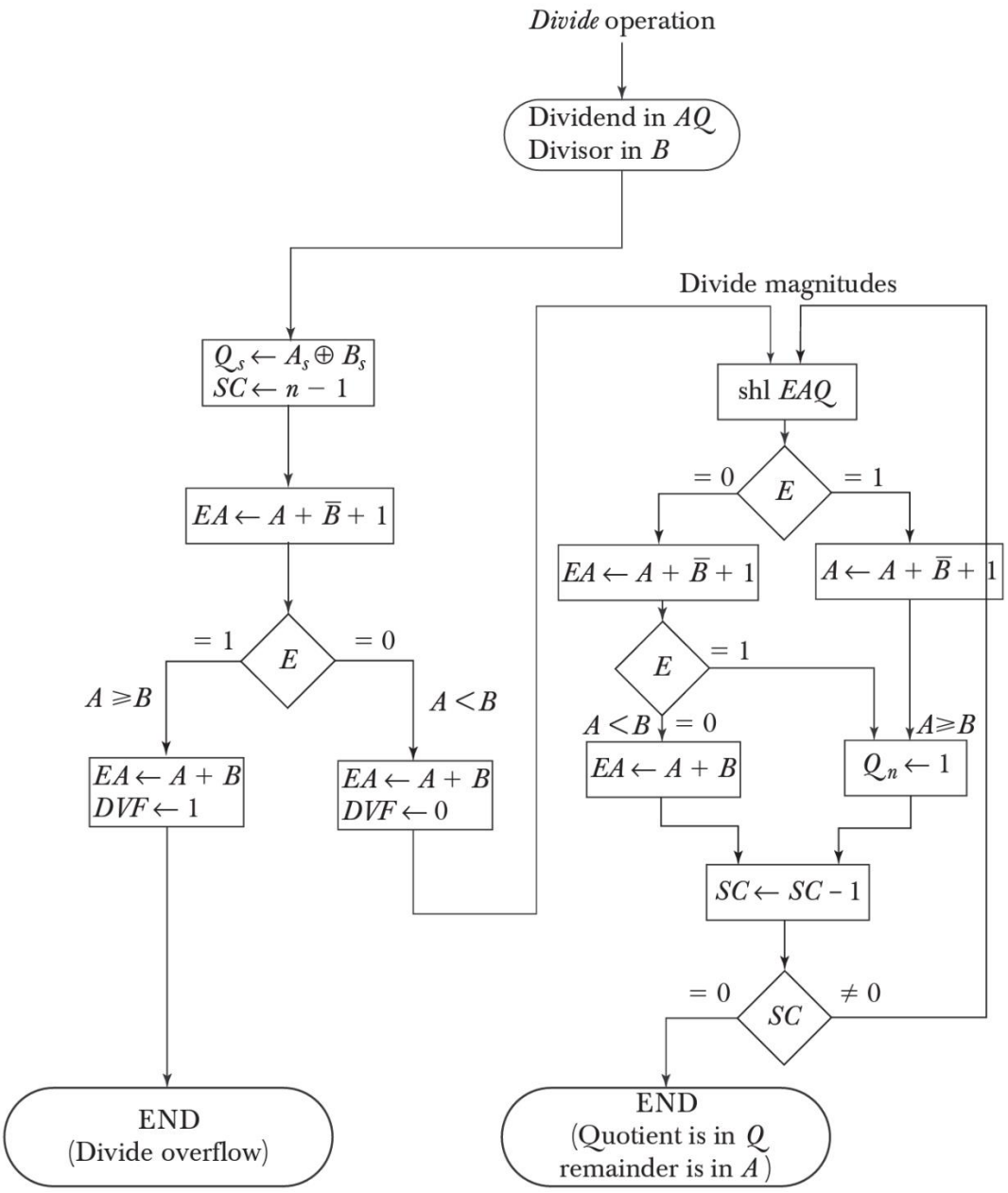
A divide-overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A. If A>=B, the divide-overflow flip-flop DVF is set and the operation is terminated prematurely. If A < B, no divide overflow occurs so the value of the dividend is restored by adding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

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, B must be subtracted from A and 1 inserted into Q_n for the quotient bit.

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>=B; therefore, Q_n is set to 1. If E = 0, it signifies that A < B and the original number is restored by adding B to A.

This process is repeated again with register A holding the partial remainder. After n-1 times, the quotient magnitude is formed in register Q and the remainder is found in register A. The quotient sign is in Q_s and the sign of the remainder in A_s is the same as the original sign of the dividend.

Figure 10-13 Flowchart for divide operation.



Comparison and Non-Restoring Method: In the non-restoring method, B is not added if the difference is negative, instead the negative difference is shifted.

In Comparison method, A and B are compared prior to the subtraction operation. Then if A≥B, B is subtracted from A. If A < B nothing is done.

Example: 448 / 17 = Quotient 26, Remainder 6

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 10-12 Example of binary division with digital hardware.

Basic Computer Organization and Design

Instruction Codes : 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. 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. Every computer has its own unique instruction set.

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 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. Consider a computer with 64 distinct operations, one of them being an ADD operation. The operation code consists of six bits, with a 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. The following represents the general format of an instruction code.

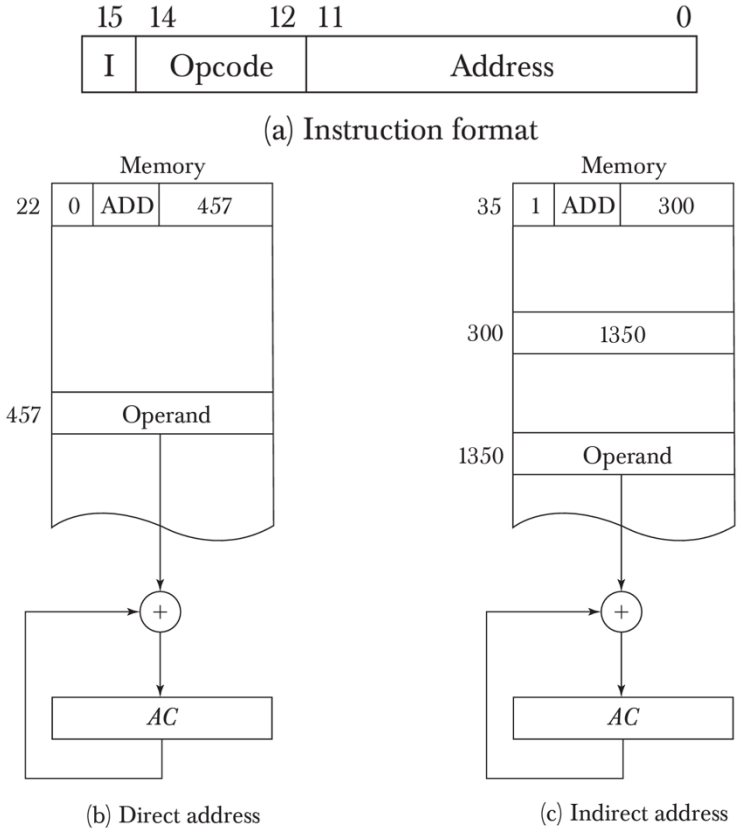


Figure 5-2 Demonstration of direct and indirect address.

Each instruction specifies the operation noted in the operation code. There are two basic types of addressing modes. Where I=0 indicates direct addressing and I=1 indicates indirect addressing.

Direct Addressing: It uses direct address of operands. i.e. address part of the instruction specifies a memory address where the data is stored. It is placed in address 22 in memory. The I bit is 0, so the instruction is recognized as a direct address instruction. The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457. The control finds the operand in memory at address 457 and adds it to the content of AC.

Indirect Addressing: It uses indirect address of operands. i.e. address part of the instruction specifies another memory location where the data is stored. The instruction in address 35 has a mode bit I = 1. Therefore, it is recognized as an indirect address instruction. The address part is the binary equivalent of 300. The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC.

The indirect address instruction needs two references to memory to fetch an operand. The first reference is needed to read the address of the operand; the second is for the operand itself.

Computer Register : 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 following table lists different computer registers and their function.

TABLE 5-1 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

The memory unit has a capacity of 4096 words and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand. This leaves three bits for the operation, part of the instruction and a bit to specify a direct or indirect address.

The data register (DR) holds the operand read from memory. The accumulator (AC) register is a general purpose processing register. The instruction read from memory is placed in the instruction register (IR). The temporary register (TR) is used for holding temporary data during the processing.

The memory address register (AR) has 12 bits since this is the width of a memory address. The program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed. The PC goes through a counting sequence and causes the computer to read sequential instructions previously stored in memory. Instruction words are read and executed in sequence

unless a branch instruction is encountered. A branch instruction calls for a transfer to a nonconsecutive instruction in the program. The address part of a branch instruction is transferred to PC to become the address of the next instruction. To read an instruction, the content of PC is taken as the address for memory and a memory read cycle is initiated. PC is then incremented by one, so it holds the address of the next instruction in sequence.

Two registers are used for input and output. The input register (INPR) receives an 8-bit character from an input device. The output register (OUTR) holds an 8-bit character for an output device.

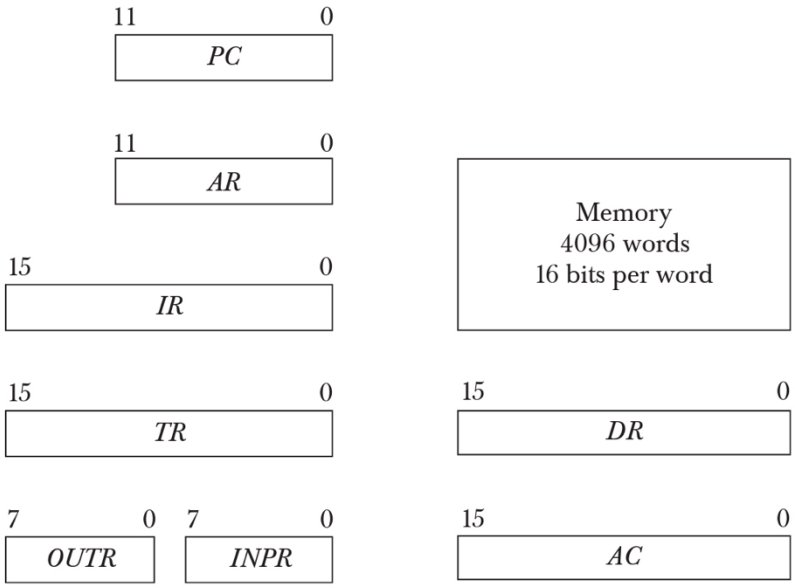
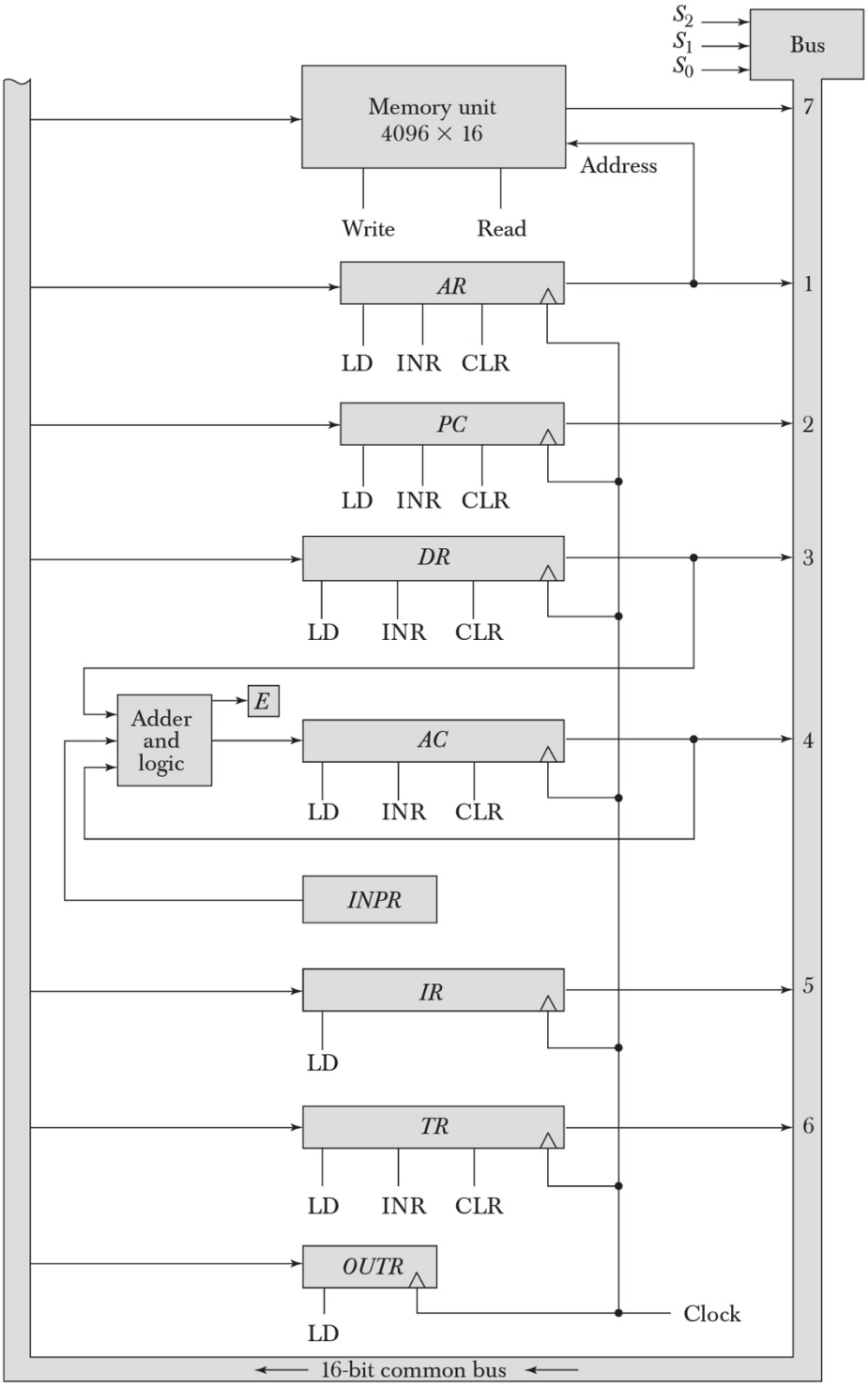


Figure 5-3 Basic computer registers and memory.

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. A more efficient scheme for transferring information in a system with many registers is to use a common bus.

1. Memory Unit : Memory Unit 4096 * 16 represents that it has a capacity of 4096 words and 1 word= 16 bits. Among these 12 bits specify address of an operand, 3 bits specify operation code, and 1 bit specify addressing mode (i.e. direct or indirect). The control inputs of memory unit are read and write. When Read input is enabled, the contents of memory unit are transferred to common bus. When Write input is enabled, the contents are transferred from bus to the memory unit.
2. Address Register (AR) : Address Register contains address for the memory. It contains 12 bit memory address. Their controls inputs are Load (LD), increment (INR), and Clear (CLR).
3. Program Counter (PC): Program Counter contains address of the next instruction to be fetched from memory. It consists of 12 bits. It allows the computer to read the instructions from the memory in sequential manner. To hold the next instruction it is incremented by 1. Its control inputs are Load (LD), Increment (INR), and Clear (CLR).
4. Data Register (DR): Data Register contains data read from memory. It consists of 16 bits. Its control inputs are Load (LD), Increment (INR) and Clear (CLR).
5. Accumulator (AC): Accumulator contains temporary operands and results of the ALU. It consists of 16 bits. Its control inputs are Load (LD), Increment (INR), and Clear (CLR).
6. Instruction Register (IR): Instruction Register contains an instruction which is currently being executed. It consists of 16 bits. Its control input is Load (LD).

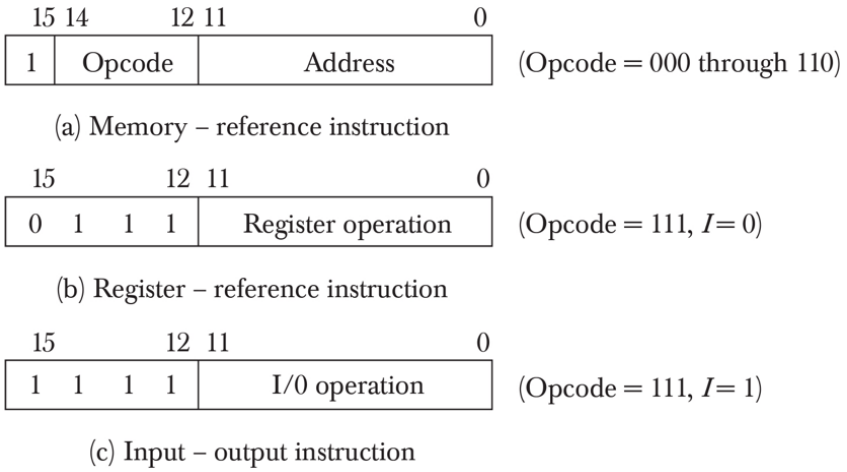
7. Temporary Register (TR): Temporary Register contains temporary data. It consists of 16 bits. Their control inputs are Load (LD), Increment (INR), and Clear (CLR).
8. Input Register (INPR): Input Register contains data read from the input device. It consists of 8 bits.
9. Output Register (OUTR): Output Register contains the data to be sent to the output device. It consists of 8 bits. Its control input is Load (LD). It receives information from the bus.
10. Flip Flop (E): A flip flop is a storage device which is capable of storing 1 bit of information. It contains the end carry out of the all arithmetic operations performed.
11. Adder and Logic Circuit: It consists of 3 sets of inputs. First set of 16 bit inputs coming from the AC. Second set of 16 bit inputs coming from the DR. Third set of 8 bi inputs coming from the INPR.



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. The number along each output shows the decimal equivalent of the required binary selection. For example, 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. The lines from the common bus are connected to the inputs of each register and the data inputs of the memory. The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and $S_2S_1S_0 = 111$.

Computer Instructions : A computer instruction is a binary code that specifies a sequence of micro operations 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 micro operations.

Figure 5-5 Basic computer instruction formats.



A basic computer has three types of instructions, with each of 16 bit long. They are

1. Memory Reference Instruction : A Memory Reference Instruction uses 12 bits to specify an address, 3 bits to specify the operation code and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address. The following are different types of memory reference instructions.

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

2. Register Reference Instruction : Register Reference Instructions are recognized by the operation code 111 with a 0 in the leftmost bit of the instruction. The remaining 12 bits represent the type of register reference instruction. The following are different types of 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

3. Input – Output Instruction : An Input-Output instruction is recognized by the operation code 111 with a 1 in the left most bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation. The following are different types of Input-Output 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 Cycle : The process of executing a program by allowing each instruction through a cycle in a sequential manner is known as Instruction Cycle. Each instruction cycle is divided into sub cycles or sub phases. To execute an instruction each instruction should pass these sub phases. 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.

1. Fetch Phase : 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 phase can be specified by the following register transfer statements.

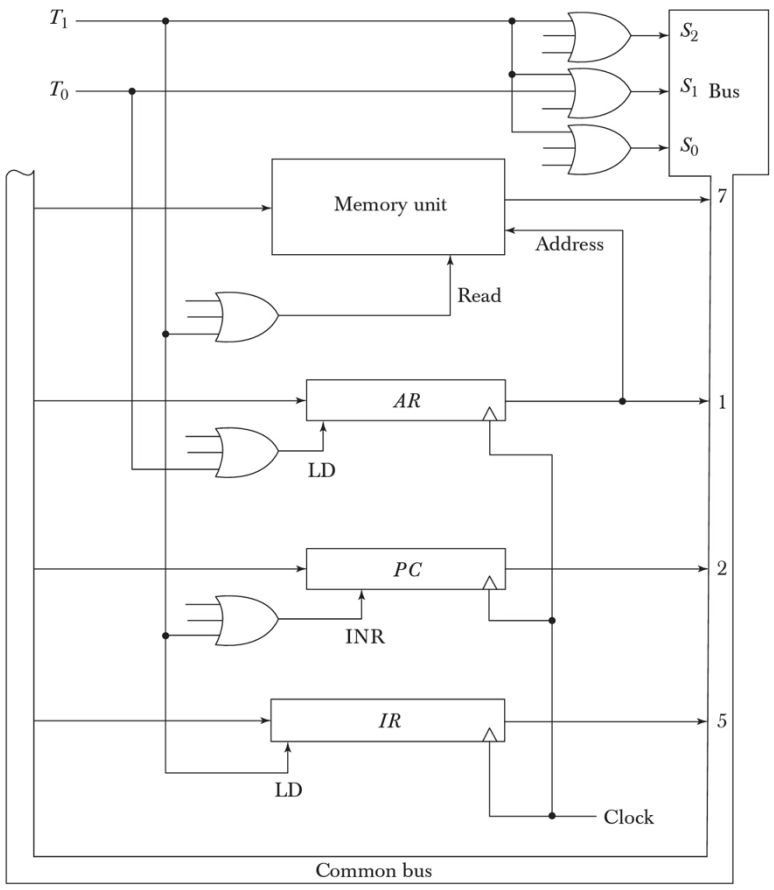
$$T_0: \quad AR \leftarrow PC$$

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

At timing signal T_0, address of the instruction is transferred from program counter to address register. At timing signal T_1, the instruction register is loaded with instruction read from the memory and program counter is incremented by 1.

Register Transfers for the Fetch Phase :

Figure 5-8 Register transfers for the fetch phase.



To provide the data path for the transfer of PC to AR, apply timing signal T_0 to achieve the following connection:

- Place the content of PC onto the bus by making the bus selection inputs $S_2S_1S_0$ equal to 010.
- Transfer the content of the bus to AR by enabling the LD input of AR.

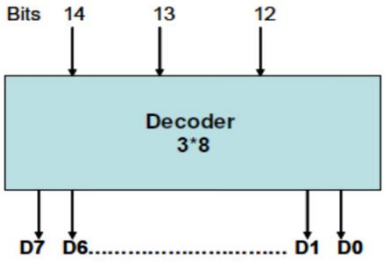
In order to implement second statement it is necessary to use timing signal T1 to provide the following connections in the bus system.

- Enable the read input of memory.
- Place the content of memory onto the bus by making $S_2S_1S_0 = 111$.
- Transfer the content of the bus to IR by enabling the LD input of IR.
- Increment PC by enabling the INR input of PC.

2. Decode Phase : During this phase, the decoding of an instruction is performed at timing signal T2. To decode

- A 3 X 8 decoder is used to decode the 12 – 14 bits of instruction register as shown below.
- The 0 – 11 bits of instruction register are loaded into address register
- The 15th bit of instruction register is loaded into the addressing mode.
- The decoding phase micro operations are,

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



3. Determining 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. Before executing the instruction type of instruction must be determined. This will be done as shown below:
   - If the 7th bit of the output of the decoder i.e. D7 = 1, then it represents register reference instruction or input – output instruction.
   - If the 7th bit of the output of the decoder i.e. D7 = 0, then it represents memory reference instruction.
   - If the 7th bit of the output of the decoder i.e. D7 = 1 and addressing mode (I) = 0, then it represents a register reference instruction.
   - If the 7th bit of the output of the decoder i.e. D7 = 1 and addressing mode (I) = 1, then it represents a input-output instruction.

4. Execution Phase : The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T3. This can be symbolized as follows:

   $D_7' I T_3$: $AR \leftarrow M[AR]$

   $D_7' I T_3$: 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. 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 $T_0 = 1$.

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. The control functions and microoperations for the register-reference instructions are listed below.

The first seven register-reference instructions perform clear, complement, circular shift, and increment microoperations on the AC or E registers. The next four instructions cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing PC once again. The condition control statements must be recognized as part of the control conditions. The AC is positive when the sign bit in AC(15) = 0; it is negative when AC(15) = 1. The content of AC is zero (AC = 0) if all the flip-flops of the register are zero. The HLT instruction clears a start-stop flip-flop S and stops the

sequence counter from counting. To restore the operation of the computer, the start–stop flip-flop must be set manually.

TABLE 5-3 Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (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:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow$ 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

Block Diagram or Flowchart of Instruction Cycle :

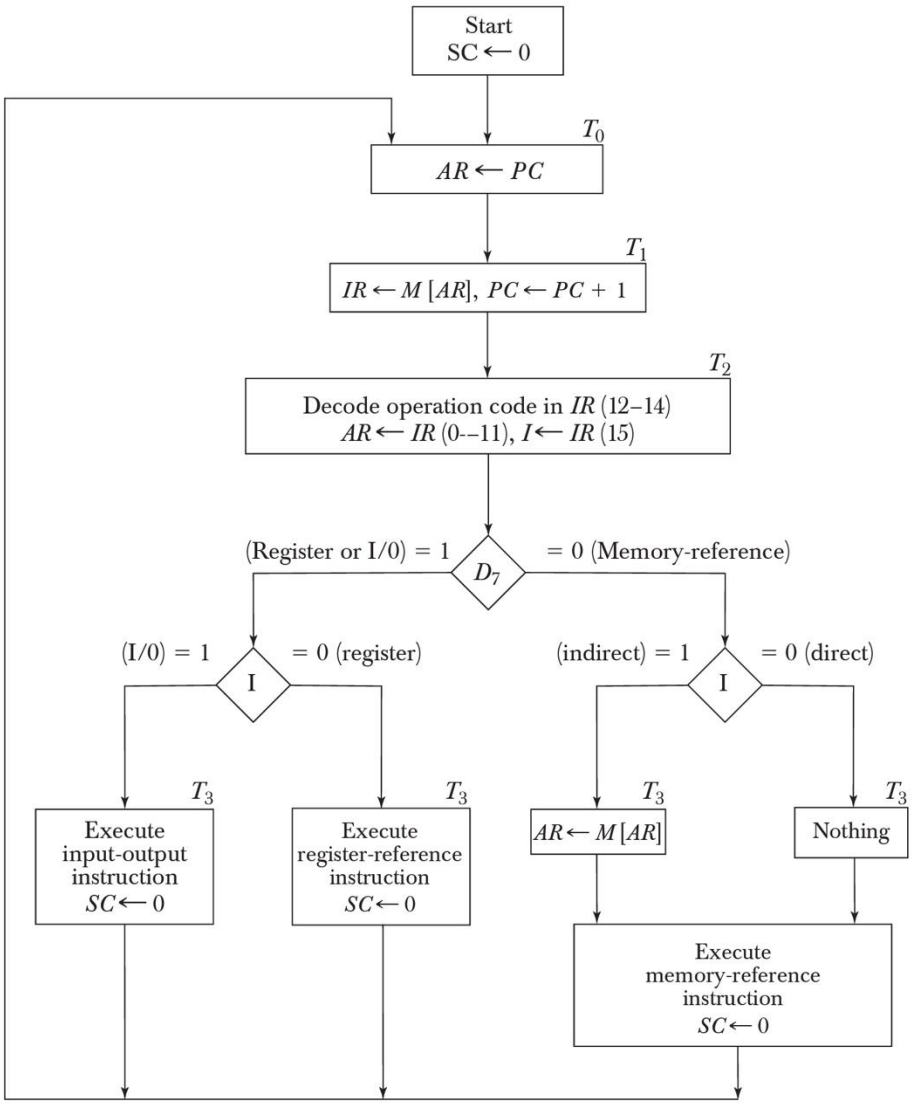


Figure 5-9 Flowchart for instruction cycle (initial configuration).

Memory – Reference Instructions : The following table lists the seven memory reference instructions. If the decoded output is D_0, AND operation is performed. Similarly, for the decoded output D_1, ADD operation will be performed and so on.

TABLE 5-4 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 $PC \leftarrow PC + 1$

AND : 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 micro operations that execute this instruction are:

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

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

The control function for this instruction uses the operation decoder D_0 since this output of the decoder is active when the instruction has an AND operation whose binary code value is 000. Two timing signals are needed to execute the instruction. The clock transition associated with timing signal T_4 transfers the operand from memory into DR. The clock transition associated with the next timing signal T_5 transfers to AC the result of the AND logic operation between the contents of DR and AC. The same clock transition clears SC to 0, transferring control to timing signal T_0 to start a new instruction cycle.

ADD : 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 microoperations needed to execute this instruction are

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

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

The same two timing signals, T_4 and T_5, are used again but with operation decoder D_1 instead of D_0, which was used for the AND instruction. After the instruction is fetched from memory and decoded, only one output of the operation decoder will be active, and that output determines the sequence of microoperations that the control follows during the execution of a memory reference instruction.

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

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

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

Here, read the memory word into DR first and then transfer the content of DR into AC.

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. In general, 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 to branche (or jump) 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.

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

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. 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 back into memory. This is done with the following sequence of microoperations:

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

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

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

Control Flowchart: A flowchart showing all micro operations for the execution of the seven memory-reference instructions is shown in below. The control functions are indicated on top of each box. The microoperations that are performed during time T_4, T_5, or T_6 depend on the operation code value. This is indicated in the flowchart by six different paths, one of which the control takes after the instruction is decoded.

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 T_0 to start the next instruction cycle.

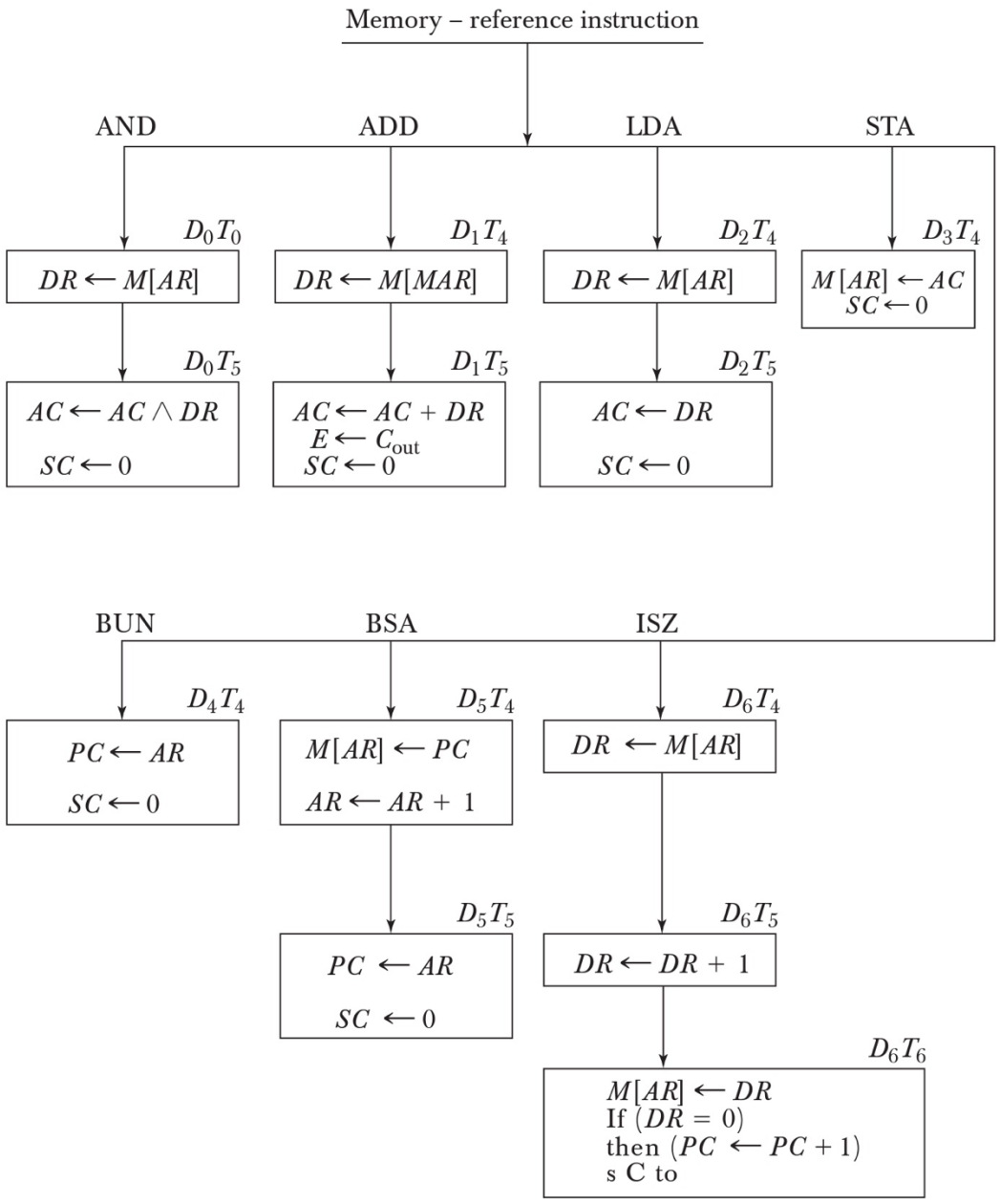


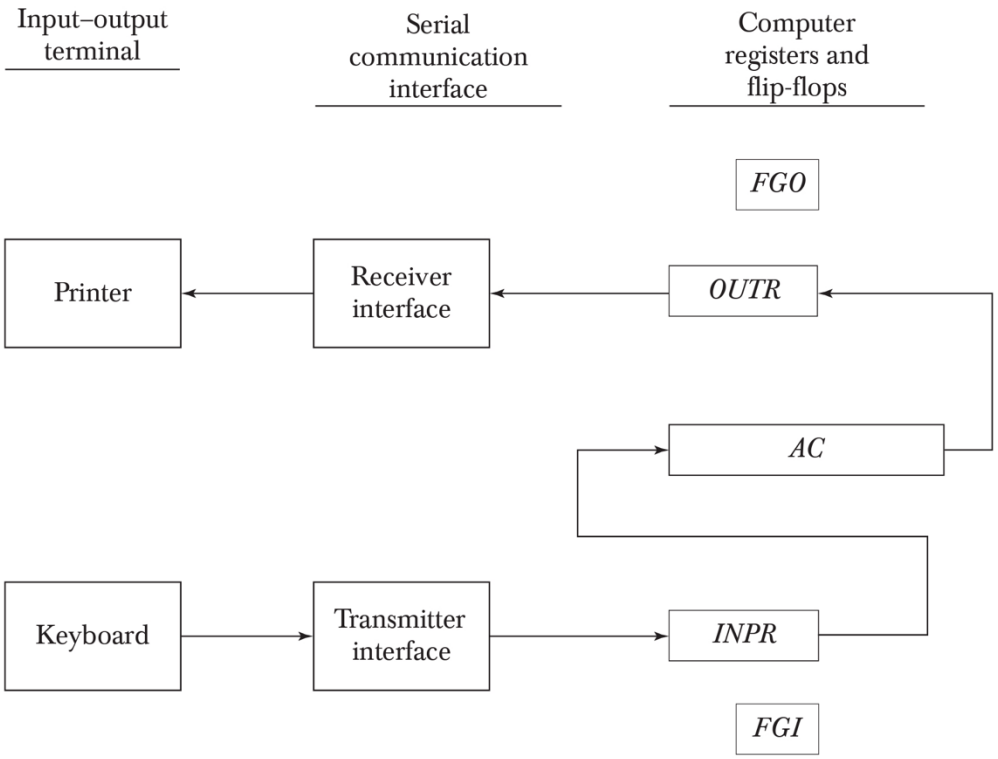
Figure 5-11 Flowchart for memory-reference instructions.

Input –Output and Interrupt : Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. 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 interfaces serially and with the AC in parallel. The communication interfaces are

- The transmitter interface receives serial information from the keyboard and transmits it to INPR.
- The receiver interface receives information from OUTR and sends it to the printer serially.

The input–output configuration is shown below.

Figure 5-12 Input–output configuration.



The data is transferred from keyboard to INPR in the following manner:

1. Initially, the input flag FGI is cleared to 0.
2. The input flag is set to 1 (FGI = 1) so that the data received from keyboard can be transmitted to input register INPR
3. The data in the input register INPR remains same till the input flag is 1.
4. The data gets transferred from INPR to accumulator AC, when FGI = 1.
5. Once the transfer is completed FGI is set to 0 and INPR starts receiving input.

The data is transferred from OUTR to printer in the following manner:

1. Initially, the output flag FGO is set to 1.
2. The computer checks the flag bit; if it is 1, the information present in the AC is transferred to OUTR.
3. Once the data transfer is completed FGI is set to 0.
4. Printer accepts data and prints it
5. Again the flag bit is set to 1

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. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input–output instructions are listed below.

TABLE 5-5 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}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	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

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 the above table 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 enable flip-flop IEN.

Interrupt : An interrupt is a routine which causes disturbance to normal flow of execution. When interrupt is occurred, 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 IEN can be set and cleared with two instructions.

- When IEN is cleared to 0 (with the Interrupt Enable Off instruction), the flags cannot interrupt the computer.
- When IEN is set to 1 (with the Interrupt Enable On instruction), the computer can be interrupted.

An interrupt flip-flop R is included which checks whether R = 0 or 1.

When R = 0

- The computer goes through an instruction cycle
- During the execution phase of the instruction cycle, if IEN is 1control checks the flag bits. If both are 0, it indicated that neither the input nor the output registers are ready to transfer information.
- So control continues with the next instruction cycle.
- At the end of the execute phase, control checks the value of R and if it is equal to 1, control goes to the interrupt cycle instead of instruction cycle.

When R = 1

- The return address available in PC is stored in M[0] (Memory Stack)
- Control inserts 1 into PC and clears IEN (IEN ← 0) and R (R ← 0) to avoid other interruptions.
- Once the information transfer is completed control return back to the instruction cycle and continue with the execution of next instruction.

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

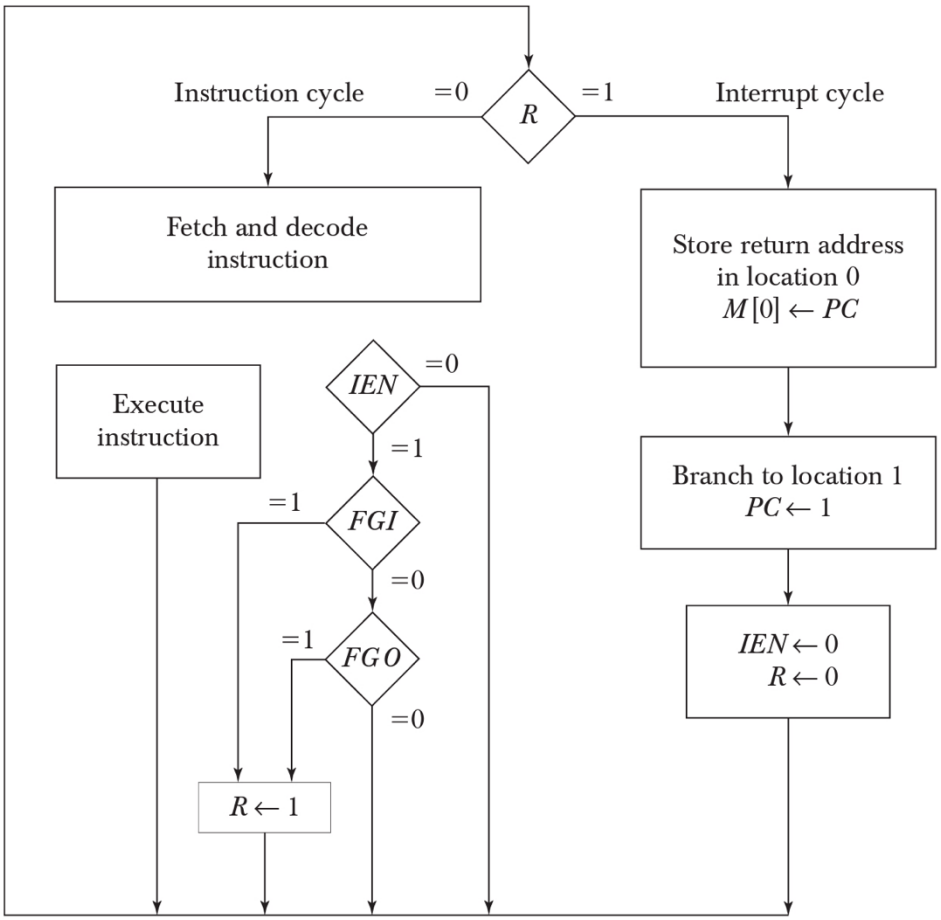


Figure 5-13 Flowchart for interrupt cycle.

Complete Computer Description : The final flowchart of the instruction cycle, including the interrupt cycle for the basic computer, is shown below.

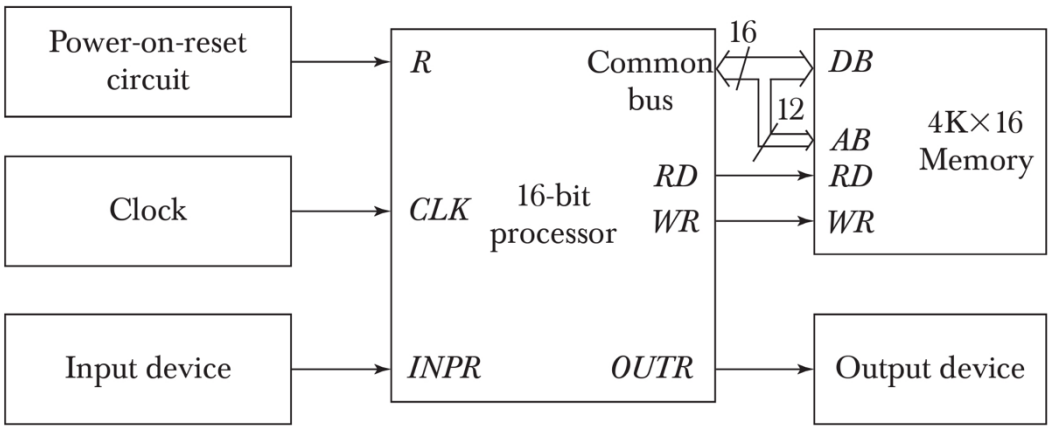


Figure 5-15 Flow chart for hypothetic basic computer.

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 according to the flowchart of memory reference instructions. If the instruction is one of the register-reference instructions, it is executed with one of the microoperations related to register reference instructions. If it is an input–output instruction, it is executed with one of the microoperations related to input-output instructions.

The following diagram represents flowchart for computer operation.

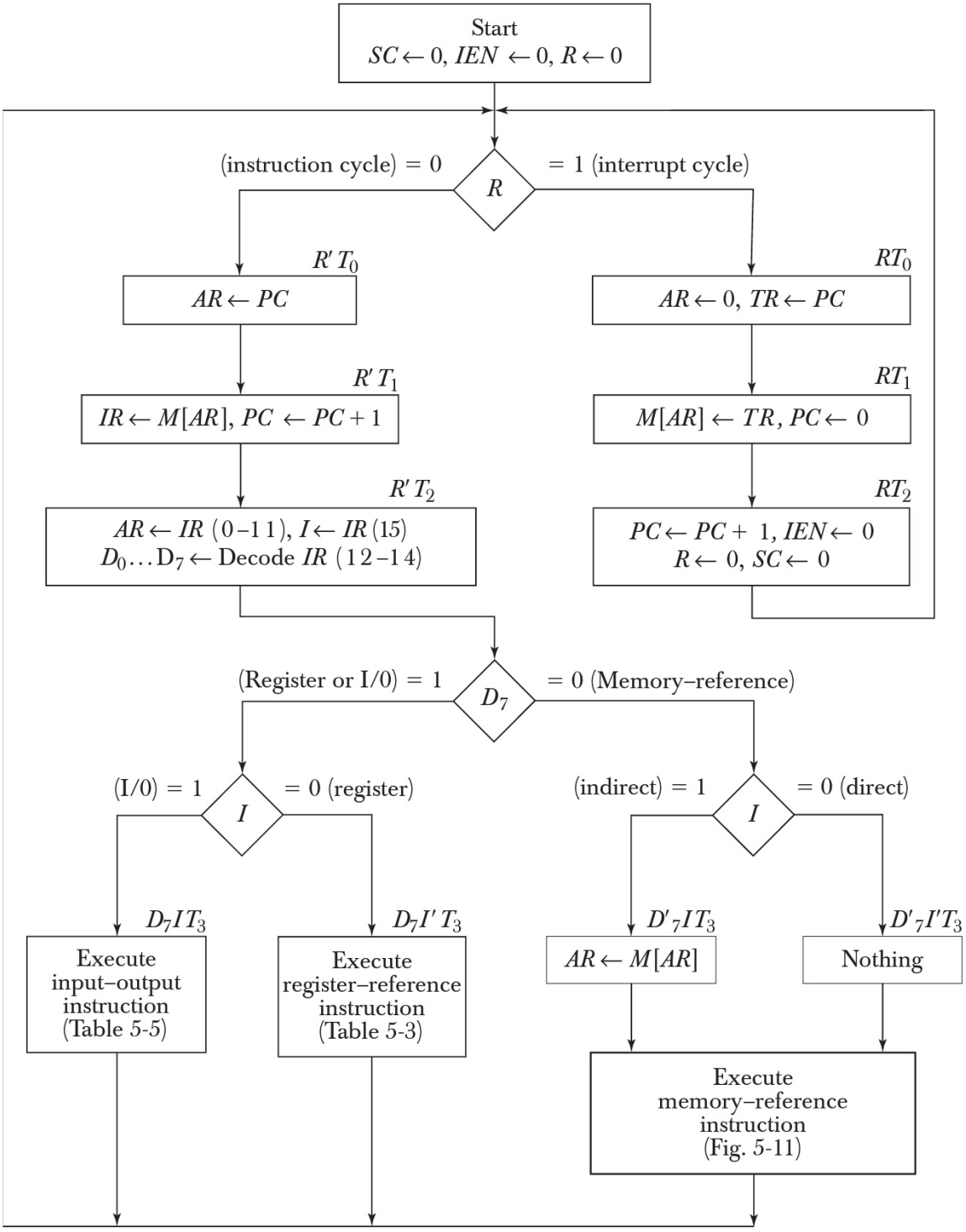


Figure 5-16 Flowchart for computer operation.

The following diagram represents control functions and microoperations for basic computer.

TABLE 5-6 Control Functions and Microoperations for the Basic Computer

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:		
$T_0T_1'T_2'(IEN)(FGI + FGO)$:		$R \leftarrow 1$
	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:	$AC \leftarrow AC \wedge DR, \quad SC \rightarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR, \quad E \leftarrow C_{out} \rightarrow SC \leftarrow 0$
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:	$M[AR] \leftarrow DR, \quad$ if $(DR = 0)$ then $(PC \leftarrow PC + 1), SC \leftarrow 0$
Register–reference:		
	$D_7I'T_3 = r$ (common to all register-reference instruct ions	
	$IR(i) = B_i \; (i = 0, 1, 2, \ldots, 11)$	
	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:	$IEN \leftarrow 0$