UNIT - III

Data Representation

Data types, Complements, Fixed Point Representation, Floating Point Representation.

Computer Arithmetic

Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating – point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.to CFG.

Unit-III

Part-1: MICROPROGRAMMED CONTROL

Contents:

- Control memory
- Address Sequencing
- Microprogram Example
- Design of Control Unit

<u>Introduction:</u>

- The function of the control unit in a digital computer is to initiate sequence of microoperations.
- Control unit can be implemented in two ways
  - Hardwired control
  - Microprogrammed control

<u>Hardwired Control:</u>

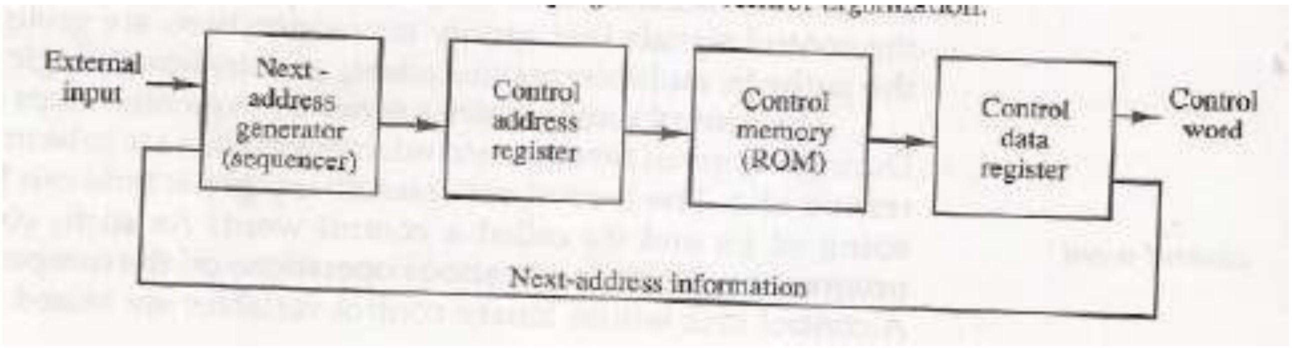
- When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be *hardwired.*
- The key characteristics are
  - oHigh speed of operation
  - oExpensive
  - oRelatively complex
  - oNo flexibility of adding new instructions
- Examples of CPU with hardwired control unit are Intel 8085, Motorola 6802, Zilog 80, and any RISC

CPUs.<u>Microprogrammed Control:</u>

- Control information is stored in control memory.
- Control memory is programmed to initiate the required sequence of micro-operations.
- The key characteristics are
  - oSpeed of operation is low when compared with hardwired
  - oLess complex
  - oLess expensive
  - oFlexibility to add new instructions
- Examples of CPU with microprogrammed control unit are Intel 8080, Motorola 68000 and any CISC CPUs.

<u>1. Control Memory:</u>

- The control function that specifies a microoperation is called as ***control variable.***
- When control variable is in one binary state, the corresponding microoperation is executed. For the other binary state the state of registers does not change.
- The active state of a control variable may be either 1 state or the 0 state, depending on the application.
- For bus-organized systems the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and arithmetic logic units.
- ***Control Word:*** The control variables at any given time can be represented by a string of 1's and 0's called a control word.
- All control words can be programmed to perform various operations on the components of the system.
- ***Microprogram control unit:*** A control unit whose binary control variables are stored in memory is called a microprogram control unit.
- The control word in control memory contains within it a *microinstruction.*
- The microinstruction specifies one or more micro-operations for the system.
- A sequence of microinstructions constitutes a *microprogram*.
- The control unit consists of control memory used to store the microprogram.
- Control memory is a permanent i.e., read only memory (ROM).
- The general configuration of a micro-programmed control unit organization is shown as block diagram below.



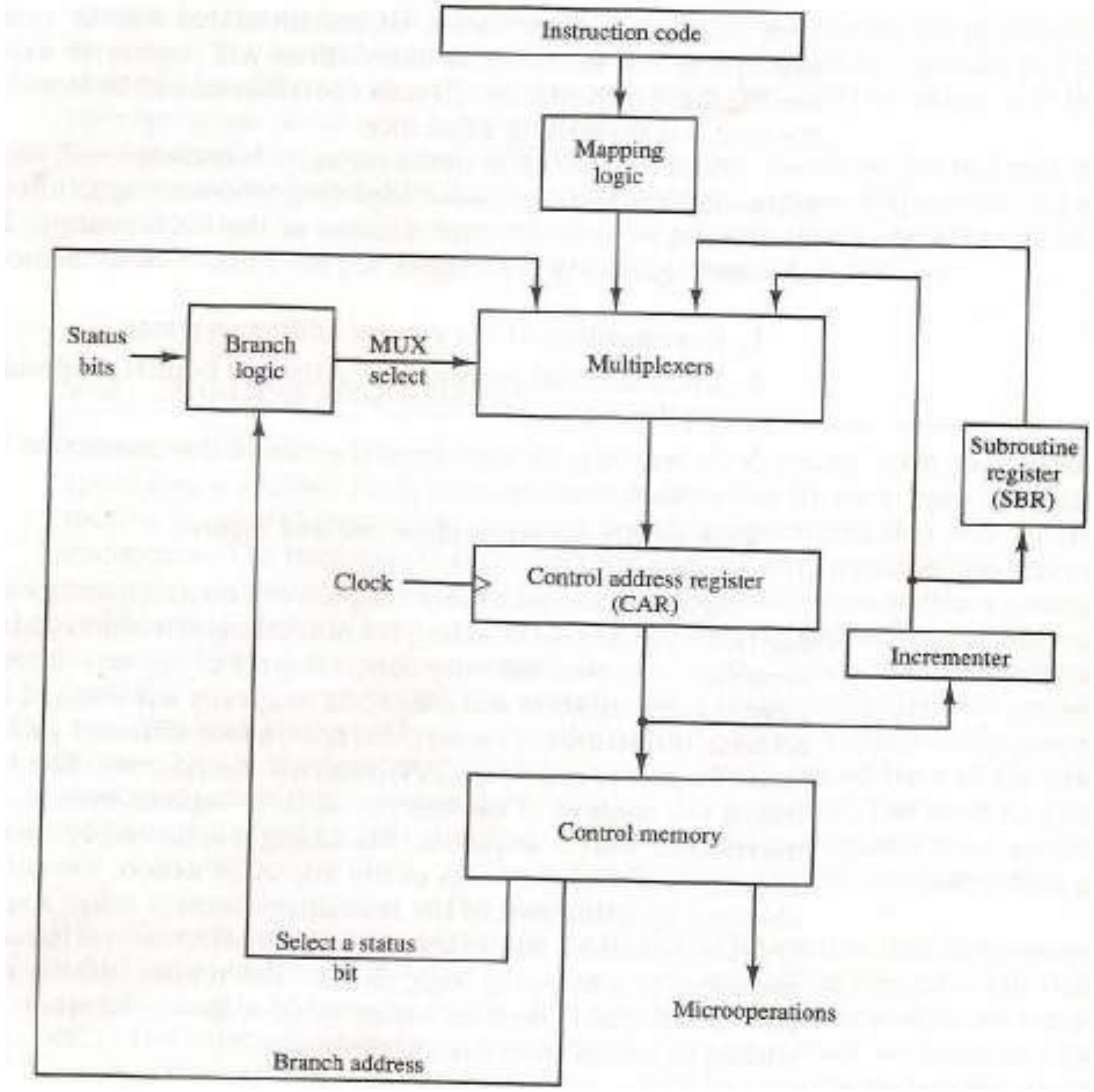
- The control memory is ROM so all control information is permanently stored.
- The control memory address register (CAR) specifies the address of the microinstruction and the control data register (CDR) holds the microinstruction read from memory.
- The next address generator is sometimes called a microprogram sequencer. It is used to generate the next micro instruction address.
- The location of the next microinstruction may be the one next in sequence or it may be located somewhere else in the control memory.
- So it is necessary to use some bits of the present microinstruction to control the generation of the address of the microinstruction.
- Sometimes the next address may also be a function of external input conditions.
- The control data register holds the present microinstruction while next address is computed and read from memory. The data register is times called a *pipeline register.*

- A computer with a microprogrammed control unit will have two separate memories: a main memory and a control memory
- The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations
- These microinstructions generate the microoperations to:
    - fetch the instruction from main memory
    - evaluate the effective address
    - execute the operation
    - return control to the fetch phase for the next instruction

2. Address Sequencing:

- Microinstructions are stored in control memory in groups, with each group specifying a *routine.*
- Each computer instruction has its own microprogram routine to generate the microoperations.
- The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another
- Steps the control must undergo during the execution of a single computer instruction:
    - Load an initial address into the CAR when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine – IR holds instruction
    - The control memory then goes through the routine to determine the effective address of the operand – AR holds operand address
    - The next step is to generate the microoperations that execute the instruction by considering the opcode and applying a *mapping process.*
        - *The transformation of the instruction code bits to an address in control memory where the routine of instruction located is referred to as mapping process.*
    - After execution, control must return to the fetch routine by executing an unconditional branch
- In brief the address sequencing capabilities required in a control memory are:
    - Incrementing of the control address register.
    - Unconditional branch or conditional branch, depending on status bit conditions.
    - A mapping process from the bits of the instruction to an address for control memory.
    - A facility for subroutine call and return.

- The below figure shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.



- The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained.
- In the figure four different paths form which the control address register (CAR) receives the address.
  - The incrementer increments the content of the control register address register by one, to select the next microinstruction in sequence.
  - Branching is achieved by specifying the branch address in one of the fields of the microinstruction.
  - Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
  - An external address is transferred into control memory via a mapping logic circuit.
  - The return address for a subroutine is stored in a special register, that value is used when the micoprogram wishes to return from the subroutine.

Conditional Branching:

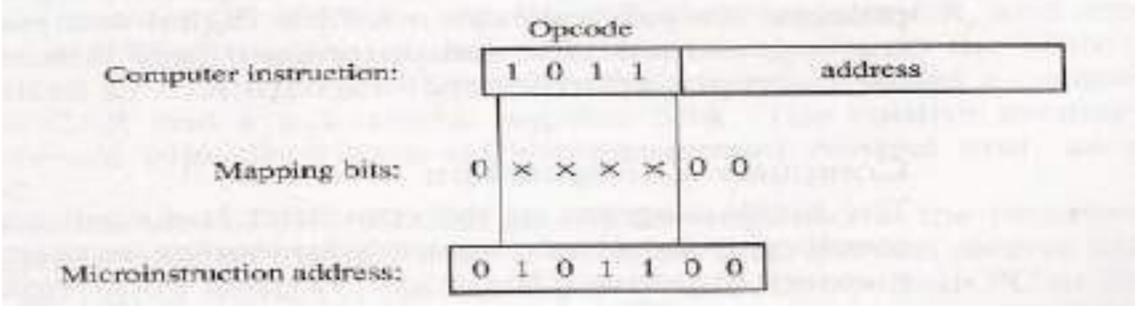
- Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
- The status conditions are special bits in the system that provide parameter information such as the carry-outof an adder, the sign bit of a number, the mode bits of an instruction, and i/o status conditions.
- The status bits, together with the field in the microinstruction that specifies a branch address, control the branch logic.

- The branch logic tests the condition, if met then branches, otherwise, increments the CAR.
- If there are 8 status bit conditions, then 3 bits in the microinstruction are used to specify the condition and provide the selection variables for the multiplexer.
- For unconditional branching, fix the value of one status bit to be one load the branch address from control memory into the CAR.

Mapping of Instruction:

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine is located.
- The status bits for this type of branch are the bits in the opcode.
- Assume an opcode of four bits and a control memory of 128 locations. The mapping process converts the 4-bit opcode to a 7-bit address for control memory shown in below figure.

Figure 7-3 Mapping from instruction code to microinstruction address.



- Mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register.
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.

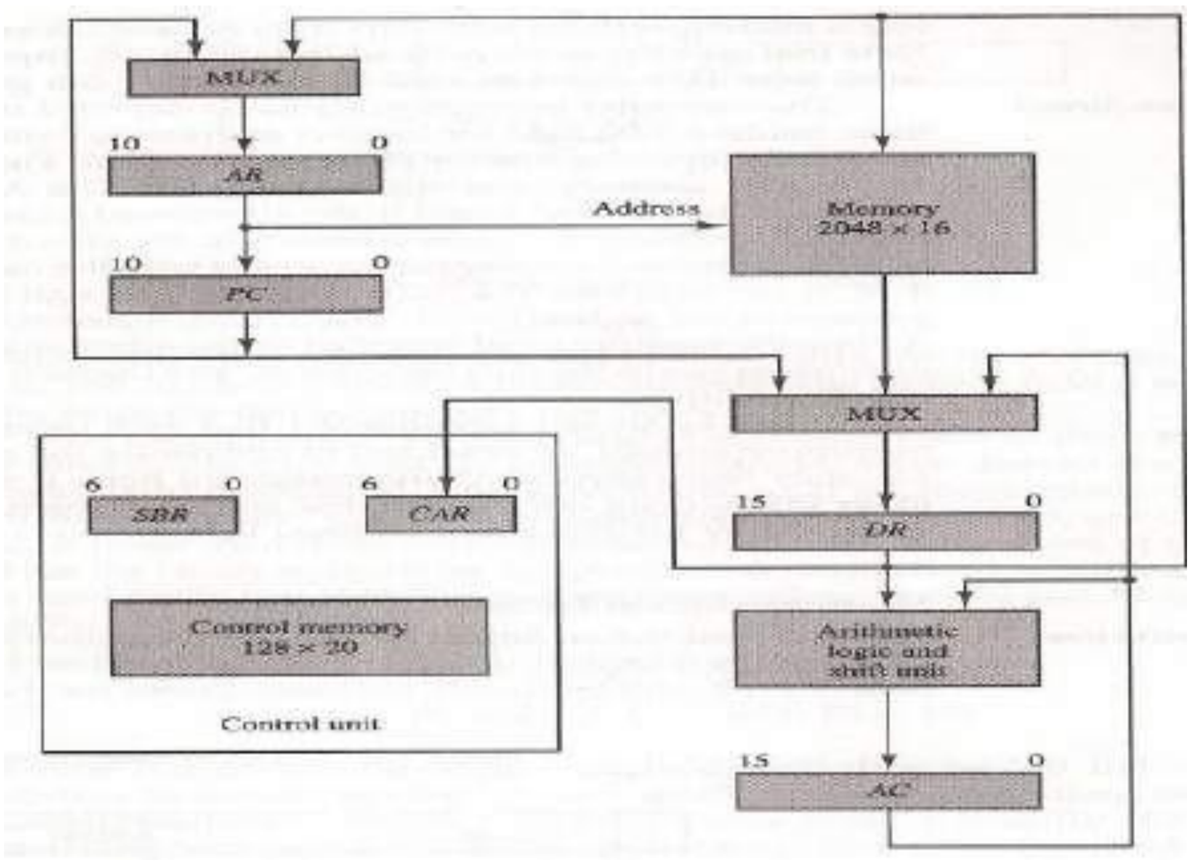
Subroutines:

- Subroutines are programs that are used by other routines to accomplish a particular task and can be called from any point within the main body of the microprogram.
- Frequently many microprograms contain identical section of code.
- Microinstructions can be saved by employing subroutines that use common sections of microcode.
- Microprograms that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return.
- A subroutine register is used as the source and destination for the addresses

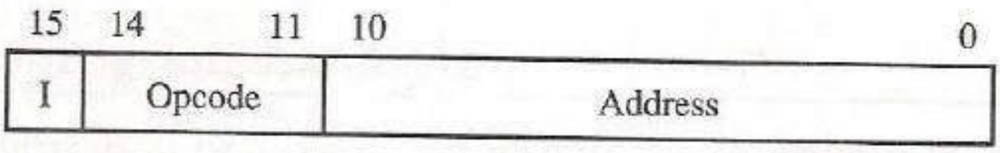
3. <u>Microprogram Example:</u>

- The process of code generation for the control memory is called *microprogramming.*
- The block diagram of the computer configuration is shown in below figure.
- Two memory units:
    - Main memory – stores instructions and data
    - Control memory – stores microprogram
- Four processor registers
    - Program counter – PC
    - Address register – AR
    - Data register – DR
    - Accumulator register - AC
- Two control unit registers
    - Control address register – CAR
    - Subroutine register – SBR

- Transfer of information among registers in the processor is through MUXs rather than a bus.



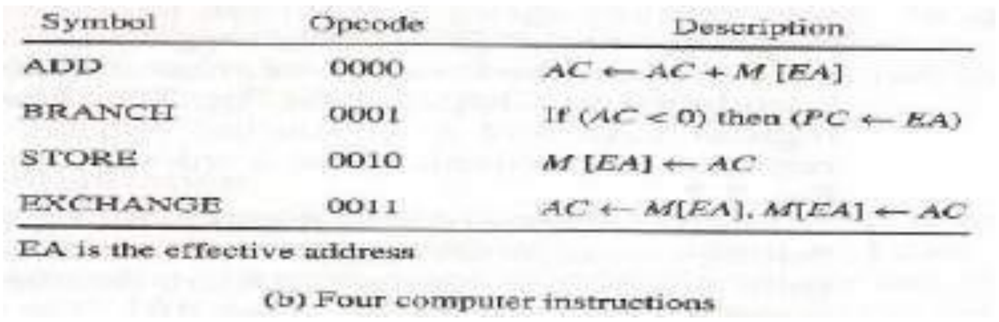
- The computer instruction format is shown in below figure.



(a) Instruction format

- Three fields for an instruction:
  - 1-bit field for indirect addressing
  - 4-bit opcode
  - 11-bit address field

- The example will only consider the following 4 of the possible 16 memory instructions



Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions

- The microinstruction format for the control memory is shown in below figure.

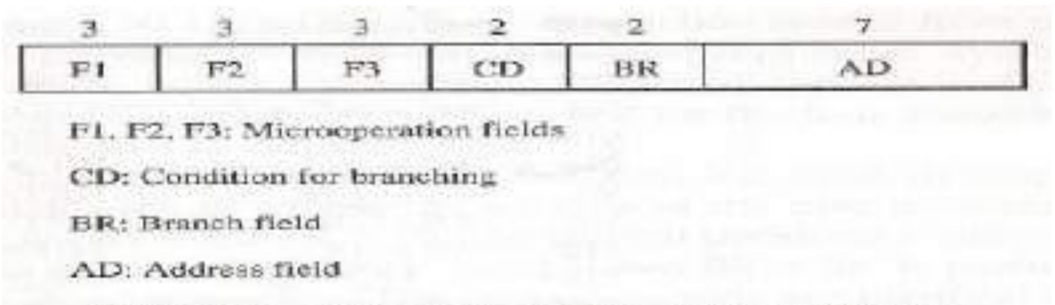
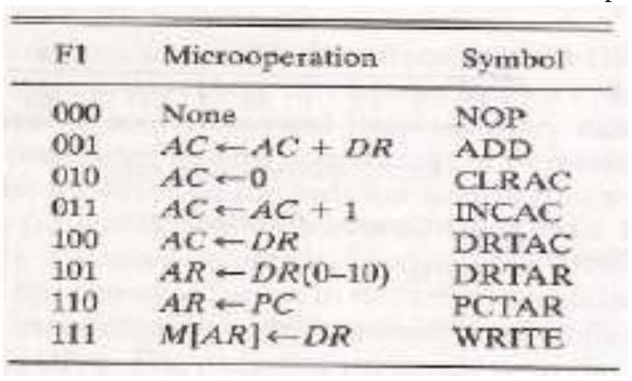
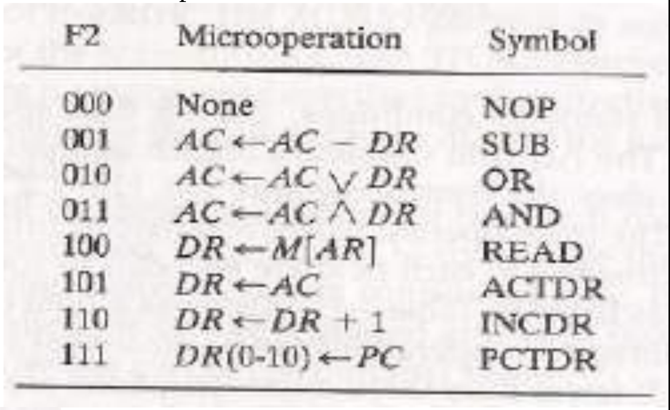

3
3
3
2
2
7
F1
F2
F3
CD
BR
AD
F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

Figure 7-6 Microinstruction code format (20 bits).

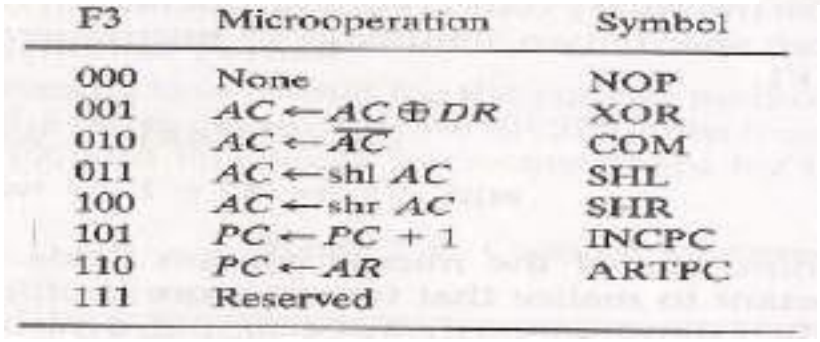
- The microinstruction format is composed of 20 bits with four parts to it
    - Three fields F1, F2, and F3 specify microoperations for the computer [3 bits each]
    - The CD field selects status bit conditions [2 bits]
    - The BR field specifies the type of branch to be used [2 bits]
    - The AD field contains a branch address [7 bits]
- Each of the three microoperation fields can specify one of seven possibilities.
- No more than three microoperations can be chosen for a microinstruction.
- If fewer than three are needed, the code 000 = NOP.
- The three bits in each field are encoded to specify seven distinct microoperations listed in below table.



F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE



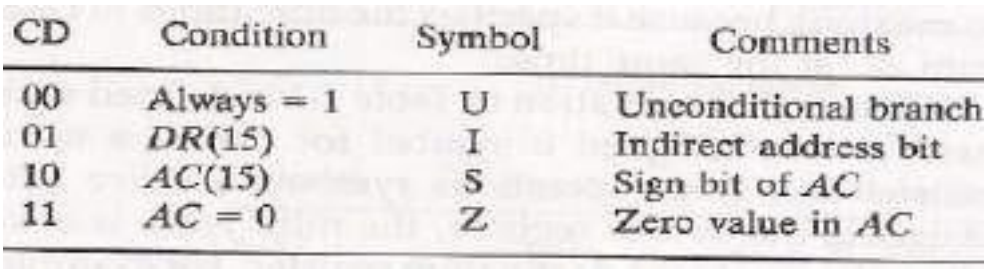
F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR



F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow \text{shl } AC$	SHL
100	$AC \leftarrow \text{shr } AC$	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

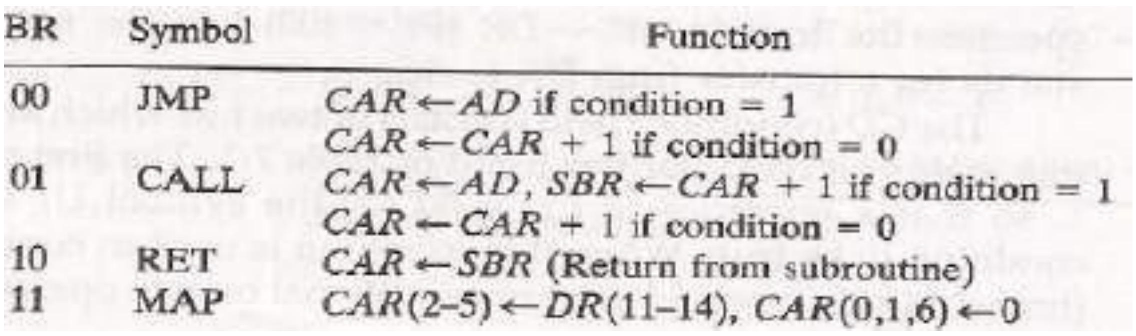
- Five letters to specify a transfer-type microoperation
    - First two designate the source register
    - Third is a 'T'
    - Last two designate the destination register

  AC ← DR F1 = 100 = DRTAC
- The condition field (CD) is two bits to specify four status bit conditions shown below



CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

- The branch field (BR) consists of two bits and is used with the address field to choose the address of the next microinstruction.

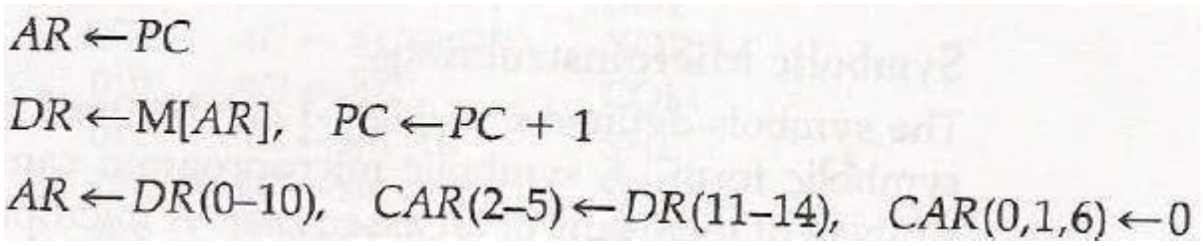


BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

- Each line of an assembly language microprogram defines a symbolic microinstruction and is divided into five parts
  1. The label field may be empty or it may specify a symbolic address. Terminate with a colon (:).
  2. The microoperations field consists of 1-3 symbols, separated by commas. Only one symbol from each field. If NOP, then translated to 9 zeros
  3. The condition field specifies one of the four conditions
  4. The branch field has one of the four branch symbols
  5. The address field has three formats
     a. A symbolic address – must also be a label
     b. The symbol NEXT to designate the next address in sequence
     c. Empty if the branch field is RET or MAP and is converted to 7 zeros
- The symbol ORG defines the first address of a microprogram routine.
- ORG 64 – places first microinstruction at control memory 1000000.

Fetch Routine:

- The control memory has 128 locations, each one is 20 bits.
- The first 64 locations are occupied by the routines for the 16 instructions, addresses 0-63.
- Can start the fetch routine at address 64.
- The fetch routine requires the following three microinstructions (locations 64-66).
- The microinstructions needed for fetch routine are:

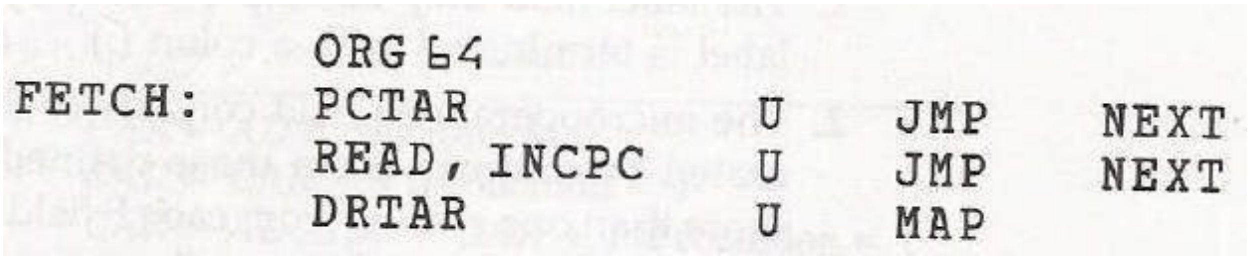


$AR \leftarrow PC$

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

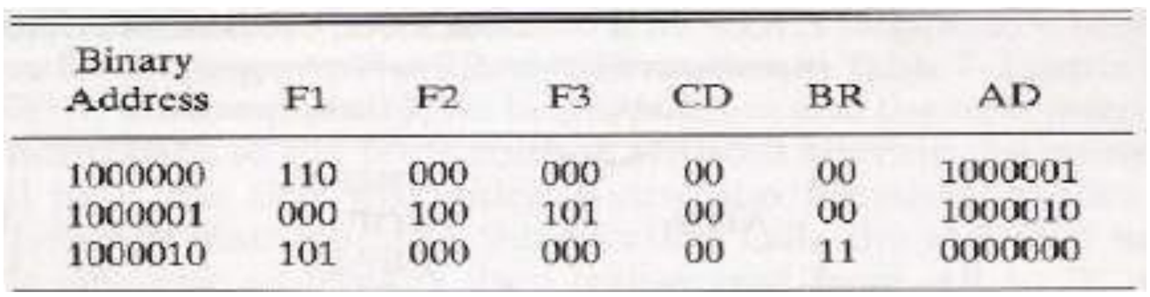
$AR \leftarrow DR(0\text{–}10), \quad CAR(2\text{–}5) \leftarrow DR(11\text{–}14), \quad CAR(0,1,6) \leftarrow 0$

- It's Symbolic microprogram:



```
        ORG 64
FETCH:  PCTAR         U   JMP   NEXT
        READ, INCPC   U   JMP   NEXT
        DRTAR         U   MAP
```

- It's Binary microprogram:



Binary Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

4. Design of control Unit:

- The control memory out of each subfield must be decoded to provide the distinct microoperations.
- The outputs of the decoders are connected to the appropriate inputs in the processor unit.
- The below figure shows the three decoders and some of the connections that must be made from their outputs.

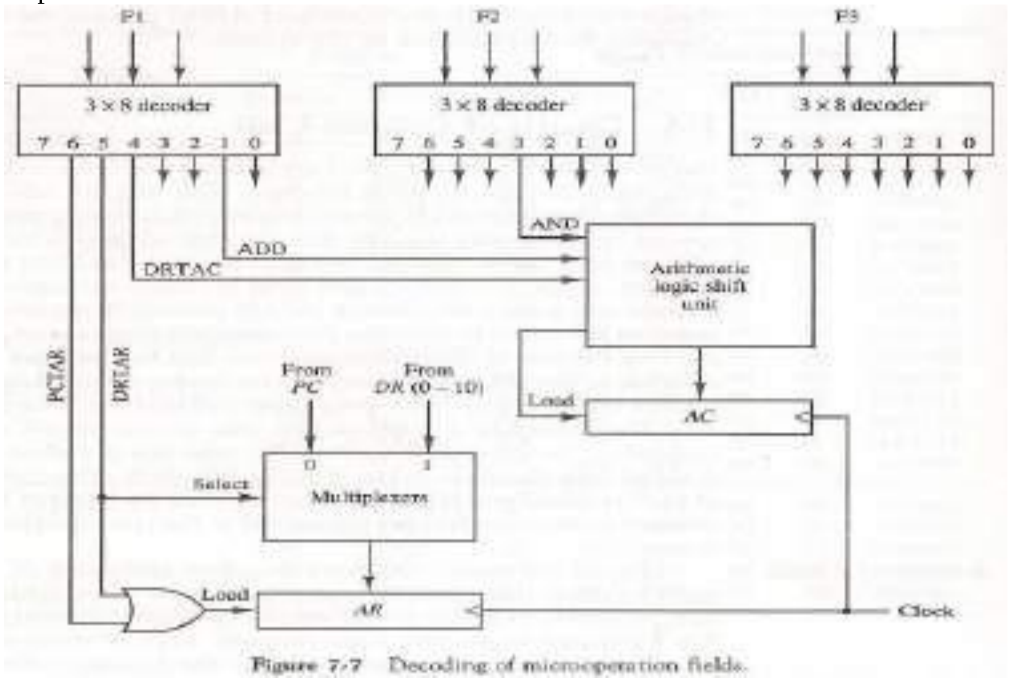


Figure 7-7 Decoding of microoperation fields.

- The three fields of the microinstruction in the output of control memory are decoded with a 3x8 decoder to provide eight outputs.
- Each of the output must be connected to proper circuit to initiate the corresponding microoperation as specified in previous topic.
- When F1 = 101 (binary 5), the next pulse transition transfers the content of DR (0-10) to AR.
- Similarly, when F1= 110 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR). As shownin Fig, outputs 5 and 6 of decoder *F1* are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR.
- The multiplexers select the information from DR when output 5 is active and from PC when output 5 is inactive.
- The transfer into AR occurs with a clock transition only when output 5 or output 6 of the decoder is active.
- For the arithmetic logic shift unit the control signals are instead of coming from the logical gates, now these inputs will now come from the outputs of AND, ADD and DRTAC respectively.

Microprogram Sequencer:

- The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address.
- The address selection part is called a microprogram sequencer.
- The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.
- The next-address logic of the sequencer determines the specific address source to be loaded into the control address register.
- The block diagram of the microprogram sequencer is shown in below figure.
- The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it.
- There are two multiplexers in the circuit.
  - The first multiplexer selects an address from one of four sources and routes it into control address register *CAR*.
  - The second multiplexer tests the value of a selected status bit and the result of the test is applied toan input logic circuit.
- The output from CAR provides the address for the control memory.

- The content of CAR is incremented and applied to one of the multiplexer inputs and to the subroutine register *SBR.*
- The other three inputs to multiplexer come from
  - The address field of the present microinstruction
  - From the out of SBR
  - From an external source that maps the instruction
- The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer.
- If the bit selected is equal to 1, the T variable is equal to 1; otherwise, it is equal to 0.
- The *T* value together with two bits from the BR (branch) field goes to an input logic circuit.
- The input logic in a particular sequencer will determine the type of operations that are available in the unit.

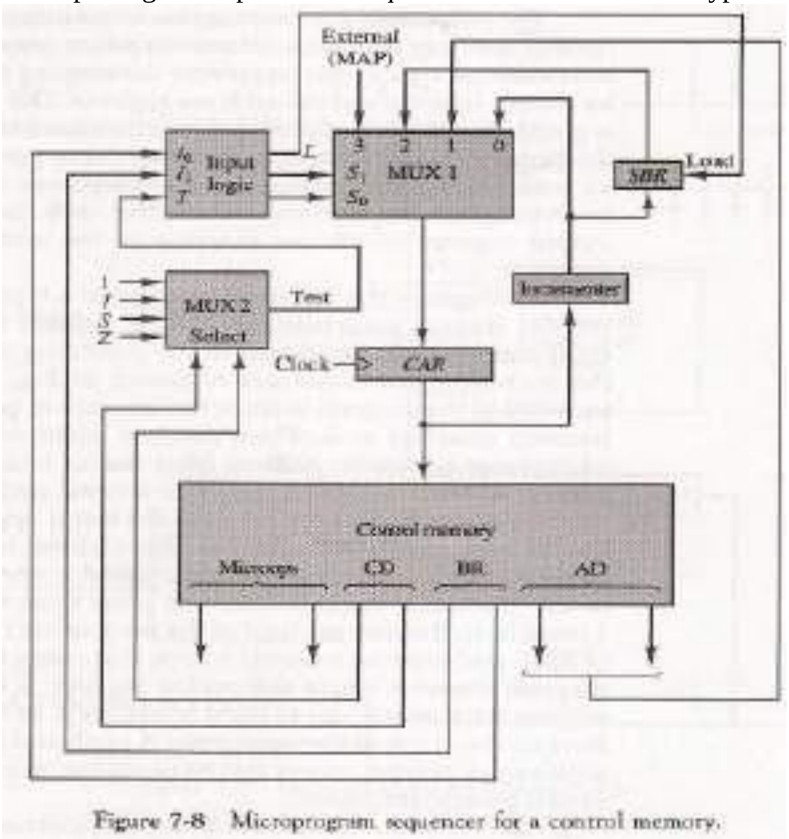
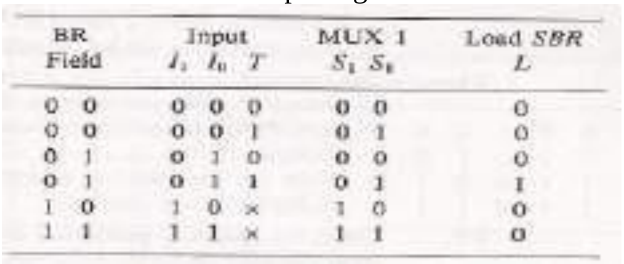


Figure 7-8 Microprogram sequencer for a control memory.

- The input logic circuit in above figure has three inputs I_0, I_1, and T, and three outputs, S_0, S_1, and L.
- Variables S_0 and S_1 select one of the source addresses for CAR. Variable L enables the load input in SBR.
- The binary values of the selection variables determine the path in the multiplexer.
- For example, with S_1, S_0 = 10, multiplexer input number 2 is selected and establishes transfer path from SBR to CAR.
- The truth table for the input logic circuit is shown in Table below.



BR Field		Input I_1	I_0	T	MUX 1 S_1	S_0	Load SBR L
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	0	1	1	0	1	1
1	0	1	0	×	1	0	0
1	1	1	1	×	1	1	0

- Inputs I_1 and I_0 are identical to the bit values in the BR field.
- The bit values for S_1 and S_0 are determined from the stated function and the path in the multiplexer that establishes the required transfer.
- The subroutine register is loaded with the incremented value of CAR during a call microinstruction (BR = 01)provided that the status bit condition is satisfied (T = 1).
- The truth table can be used to obtain the simplified Boolean functions for the input logic circuit:

  $S_1=I_1$
  $S_0=I_1I_0+I'$
  $_1TL = I'_1T$
  I_0

UNIT – III

PART – II - COMPUTER ARITHMETIC

Addition, subtraction, multiplication are the four basic arithmetic operations. Using these operations other arithmetic functions can be formulated and scientific problems can be solved by numerical analysis methods.

Arithmetic Processor:

It is the part of a processor unit that executes arithmetic operations. The arithmetic instructions definitions specify the data type that should be present in the registers used . The arithmetic instruction may specify binary or decimal data and in each case the data may be in fixed-point or floating point form. Fixed point numbers may represent integers or fractions. The negative numbers may be in signed- magnitude or signed- complement representation. The arithmetic processor is very simple if only a binary fixed point add instruction is included. It would be more complicated if it includes all four arithmetic operations for binary and decimal data in fixed and floating point representations.

Algorithm:

Algorithm can be defined as a finite number of well defined procedural steps to solve a problem. Usually, an algorithm will contain a number of procedural steps which are dependent on results of previous steps. A convenient method for presenting an algorithm is a flowchart which consists of rectangular and diamond –shaped boxes. The computational steps are specified in the rectangular boxes and the decision steps are indicated inside diamond-shaped boxes from which 2 or more alternate path emerge.

ADDITION AND SUBTRACTION:

3 ways of representing negative fixed point binary numbers:

1. **Signed-magnitude representation----** used for the representation of mantissa for floating pointoperations by most computers.
2. **Signed-1's complement**
3. **Signed -2's complement**—Most computers use this form for performing arithmetic operation withintegers

Addition and subtraction algorithm for signed-magnitude data

Let the magnitude of two numbers be A & B. When signed numbers are added or subtracted, there are 4 different conditions to be considered for each addition and subtraction depending on the sign of the numbers. The conditions are listed in the table below. The table shows the operation to be performed with magnitude(addition or subtraction) are indicated for different conditions.

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

The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

The algorithm for addition and subtraction (from the table above):

Addition Algorithm:

When the signs of A and B are identical, add two magnitudes and attach the sign of A to the result. When the sign 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 sign of A if A < B. If the two magnitudes are equal, subtract B from A and make te sign of the result positive.

Subtraction algorithm:

When the signs of A and B are different, add two magnitudes and attach the sign of A to the result. When the sign 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 sign of A if A < B. If the two magnitudes are equal, subtract B from A and make te sign of the result positive.

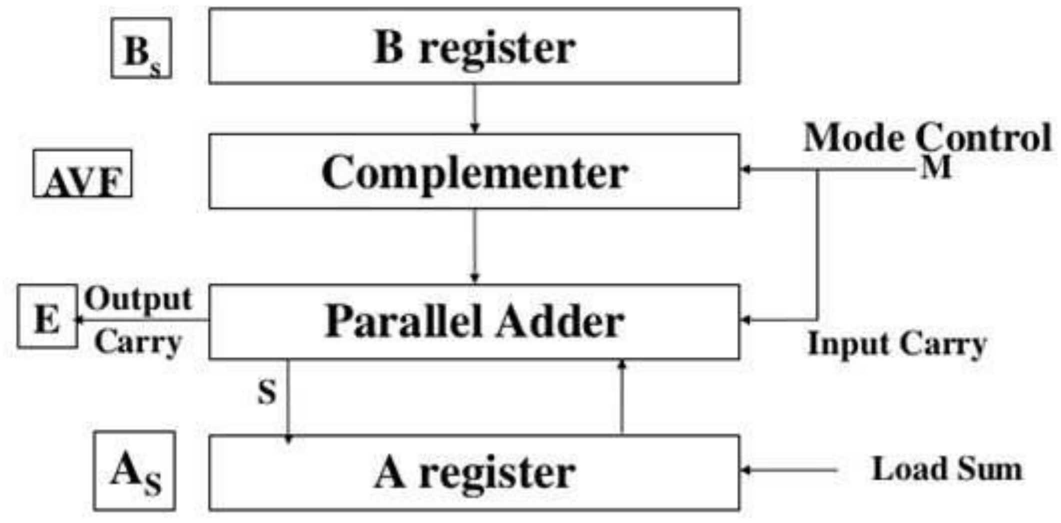
Hardware Implementation:

Let A and B are two registers that hold the numbers.
A_S and B_S are 2, flip-flops that hold sign of corresponding numbers.
The result is stored In A and A_S .and thus they form Accumulator register.We need to perform micro operation, A+ B and hence a parallel adder.
A comparator is needed to establish if A> B, A=B, or A<B.
We need to perform micro operations A-B and B-A and hence two parallel subtractor. An exclusive OR gate can be used to determine the sign relationship, that is, equal or not.
Thus the hardware components required are a magnitude comparator, an adder, and twosubtractors.

Reduction of hardware by using different procedure:

1. We know subtraction can be done by complement and add.
2. The result of comparison can be determined from the end carry after the subtraction.

We find An adder and a complementer can do subtraction and comparison if 2’s complement is used for subtraction.



Hardware forsigned-magnitude addition and subtractionAVF Add overflow

flip flop. It hold the overflow bit when A & B are added.

Flip flop E—Output carry is transferred to E. It can be checked to see the relative magnitudes of the twonumbers.
A-B = A +(-B)= Adding a and 2's complement of B.
The A register provides other micro operations that may be needed when the sequence of steps in thealgorithm is specified.
The complementer Passes the contents of B or the complement of B to the Parallel Adder depending on the state of the mode control B. It consists of EX-OR gates and the parallel adder consists of full adder circuits. The M signal is also applied to the input carry of the adder.
When input carry M=0, the sum of full adder is A +B. When M=1, S = A + B' +1= A – B Hardware algorithm:

Flow Chart for Add and Subtract operations:
The EX-OR gate provides 0 as output when the signs are identical. It is 1 when the signs aredifferent.
A + B is computed for the following and the sum is stored in EA:
1. When the signs are same and addition operation is required.
2. When the signs are different and subtract operation is required.

The carry in E after addition indicates an overflow if it is 1 and it is transferred to AVF, theaddoverflow flag
A-B = A+ B'+1 computed for the following:
1. When the signs are different and addition operation is required.
2. When the signs are same and subtract operation is required.

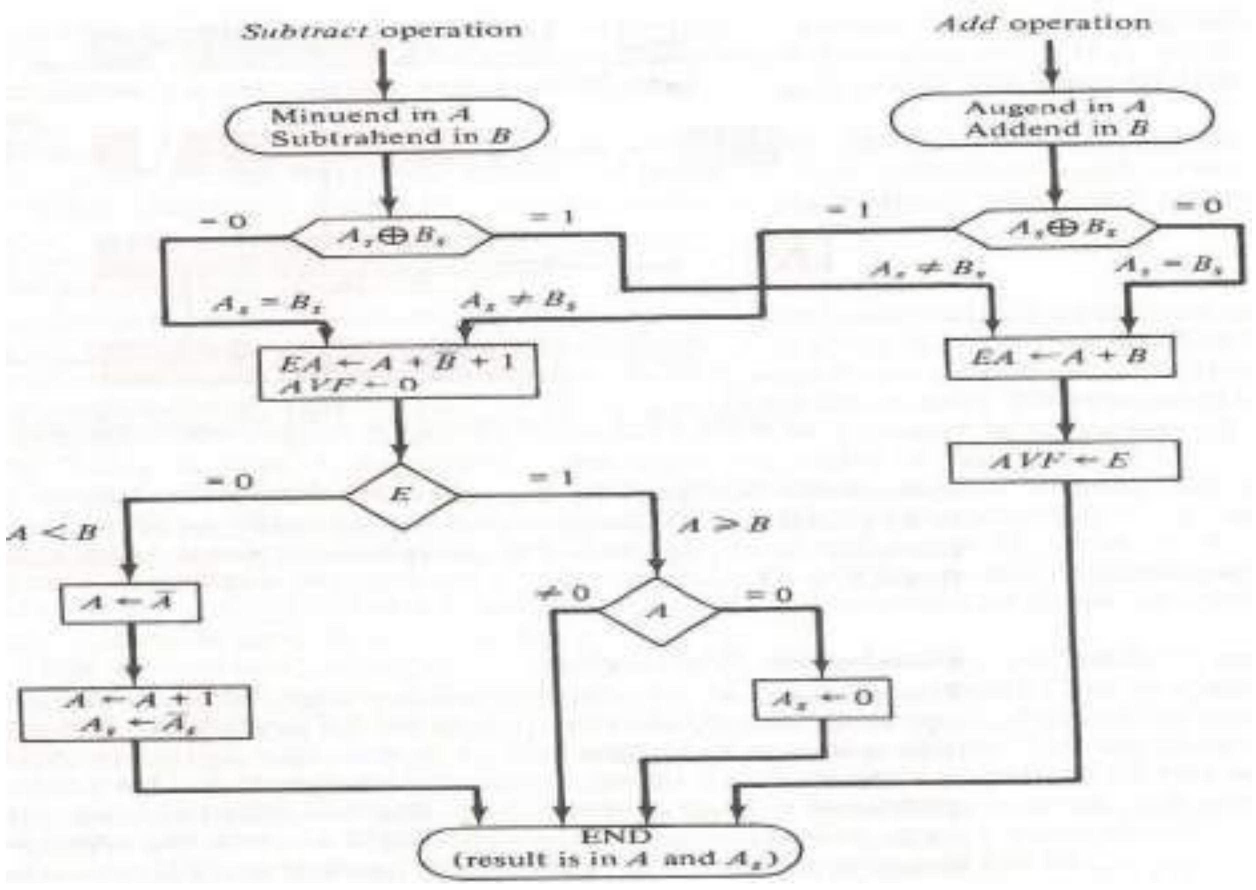
No overflow can occur if the numbers are subtracted and hence AVF is cleared to Zero.[the subtraction

of 2 n-digit un signed numbers M-N (N≠0) in base r can be done as follows:
1. Add minuend M to thee r's complement of the subtrahend N. This performs M-N +rn .
2. If M ≥ N, The sum will produce an end carry rnwhich is discarded, and what is left is the result M-N.
3. If M< N, the sum does not produce an end carry and is equal to rn–(N-M), which is the r's complement of the sum and place a negative sign in front.] in A.

A 1 in E indicates that A ≥ B and the number in A is the correct result.
If this number in A is zero, the sign A_S must be made positive to avoid a negative zero.
A 0 in E indicates that A< B. For this case it is necessary to take the 2's complement of the value

In the algorithm shown in flow chart, it is assumed that A register has circuits for micro operations complement and increment. Hence two complement of value in A is obtained in 2, micro operations. In other paths of the flow chart, the sign of the result is the same as the sign of A, so no change in A_S is required.

However When A < B, the sign of the result is the complement of original sign of A.
Hence The complement of A_S stored in A_S.
Final Result: A_S A



Flow chart for ADD and Subtract operations

Addition and Subtraction with signed-2's complement Data.:
Arithmetic Addition:

This method does not need a comparison or subtraction but only addition and complementation. The procedure is as below:

1. Represent the negative numbers in 2's complement form.

2. Add the two numbers including the sign bits and discard any carry out of sign bit position.

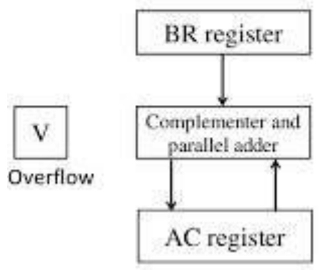
3. The overflow bit V is set to 1 if there is a carry into sign bit and no carry out of sign bit or ifthere is a no carry into sign bit and a carry out of sign bit. Otherwise it is set to zero.

4. If the result is negative, take the 2's complement of the result to get a correct negative result.

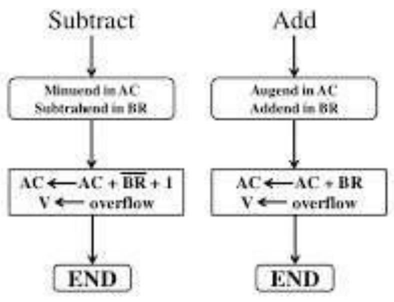
Arithmetic Subtraction:

1. Represent the negative numbers in 2's complement form.
2. Take the 2's complement of the subtrahend including the sign bit and add it to the minuendincluding the sign bit.
3. The overflow bit V is set to 1 if there is a carry into sign bit and no carry out of sign bit or if thereis a no carry into sign bit and a carry out of sign bit. Otherwise it is set to zero.
4. Discard the carry out of the sign bit position.

Note: A subtraction operation can be changed to an addition operation if the sign of the subtrahendis changed.



Hardware for Signed 2/s complement for addition/ subtractioin



Algorithm For Adding And Subtracting Numbers In Signed -2's Complement Representation

<u>MULTIPLICATION ALGORITHM:</u>

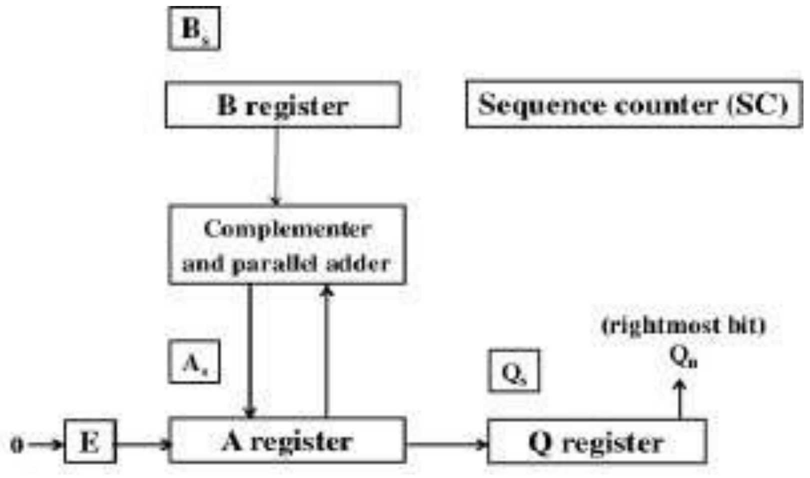
Hardware implementation of multiplication of numbers in signed – magnitude form:

register. A adder is provided to add two binary numbers and the partial product is accumulated in a

1. Instead of shifting the multiplicand to the left, the partial product is shifted to the right, which result in leaving the partial product and the multiplicand in the required relative positions.

2. When the corresponding bit of the multiplier is zero, there is no need to add all zeros to the partial product, since it will not alter it's value.

The hardware consists of 4 flipflops, 3 registers, one sequence counter , an adder andcomplementer.



Hardware For Multiply OperationQ register&Q_s

flip flop: contains multiplier & Its sign
Sequence counter: It is set to a value equal to the number of bits in the multiplier
B Register& B_s flipflop: It contains the multiplicand,& its sign
A Register, E Flip flop: Initialized to ' 0'. A_s denotes sign of partial product
EA Register: hold partial product, with carry generated in addition being shifted to E .
Qn: Rightmost bit of the multiplier; AQ : will contain the final product.

As **AQ** represent product register, both A_s $Q_s$represent the sign of the partial product or product.

The number to be multiplied are stores in memory as n bit sign magnitude numbers and whentransferred to register msb bit go to sign flipflop and remaining n-1 bits go to registers. Hence SC is initially set to n-1.

Let the lower order bit of the multiplier in $Q_n$tested.

If it is 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 it's new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops whenSC = 0.

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

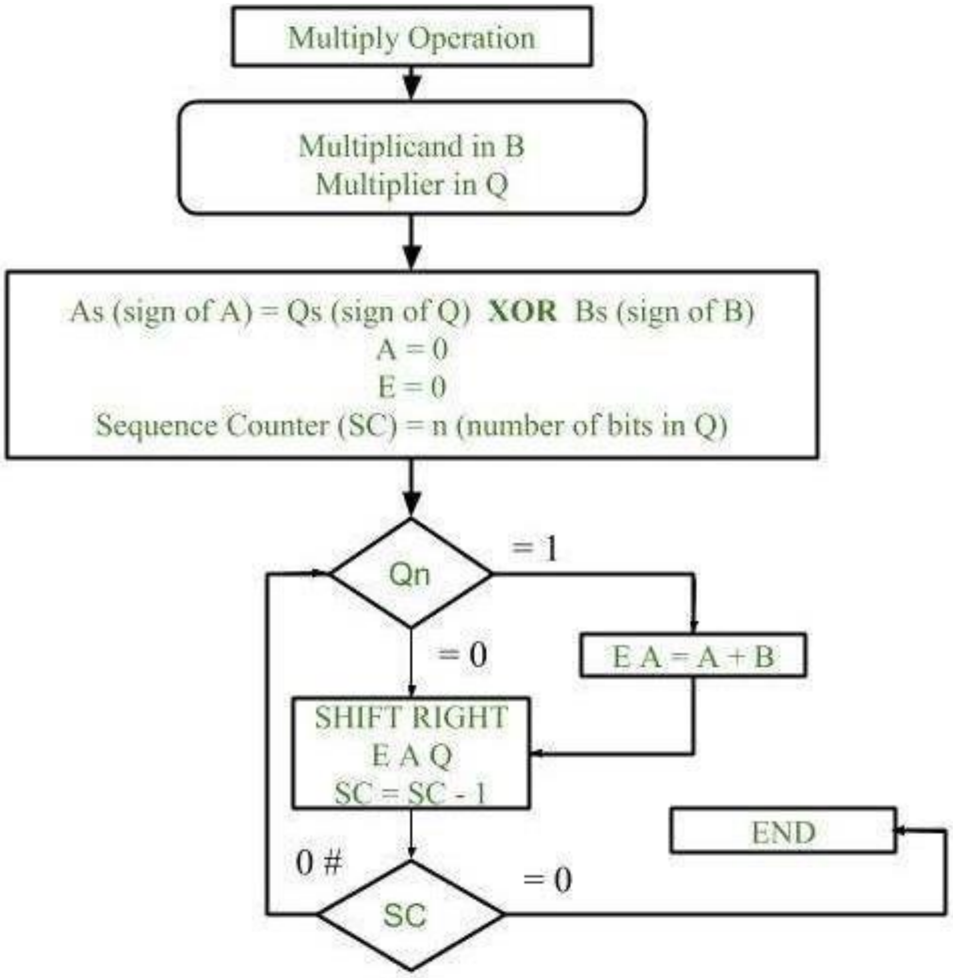


Fig: Flowchart for multiply operation

Multiplicand B= 10111	E	A	Q	SC
Multiplier in Q Qn =1;add B First Partial Product Shift Right EAQ	0 0 0	00000 10111 10111 01011	10011 11001	101 100
Qn =1;add B Second Partial Product	 1	10111 00010		
Shift Right EAQ	0	10001	01100	011
Qn =0; Shift Right EAQ	0	10001	01100	011
Qn =0; Shift Right EAQ	0	01000	10110	010

Qn =1;add B Fifth Partial Product Shift Right EAQ	 0 0	10111 11011 01101	 10101	 000
Final Product in AQ AQ = 0110110101				

Numerical Example for the above algorithm

Booth Multiplication Algorithm:

Multiplication of signed- 2's complement integers:

This algorithm uses the following facts.

1. A string of 0's in the multiplier requires no addition but just shifting.
2. A string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as 2^{k+1} - $2^{m.}$

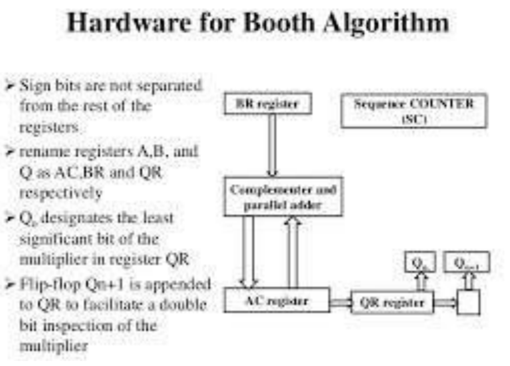
Example: Consider the binary number: 001110 (+14)

The number has a string of 1's from 2^3 to 2^1 . Hence k = 3 and m= 1. As other bits are 0's, the number can be represented as 2^{k+1} - 2^m = 2^4 – 2^1 = 16-2 = 14. Therefore the multiplication M * 14 , whereM is the multiplicand and 14 the multiplier can be done as Mx 2^4 –M x $2^{1.}$

This can be achieved by shifting binary multiplicand M four times to the left and subtracting Mshifted left once which is equal to (Mx 2^4 –M x $2^{1.}$).

Shifting and addition/subtraction rules for multiplicand in Booth's Algorithm:

1. The multiplicand is subtracted from the partial product upon encountering the first least significand 1 in a string of I'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 0's in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit



Hardware for Booth Algorithm

- Sign bits are not separated from the rest of the registers
- 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
- Flip-flop Qn+1 is appended to QR to facilitate a double bit inspection of the multiplier

Hardware Implementation of Booth Algorithm

Note: Sign bit is not separated from register. QR register contains the multiplier register and $Q_n$representthe least significant bit of the multiplier in QR. Q_{n+1} is an extra flip flop appended to QR to facilitate a double bit inspection of the multiplier.

AC register and appended Q_{n+1} are initially cleared to 0.

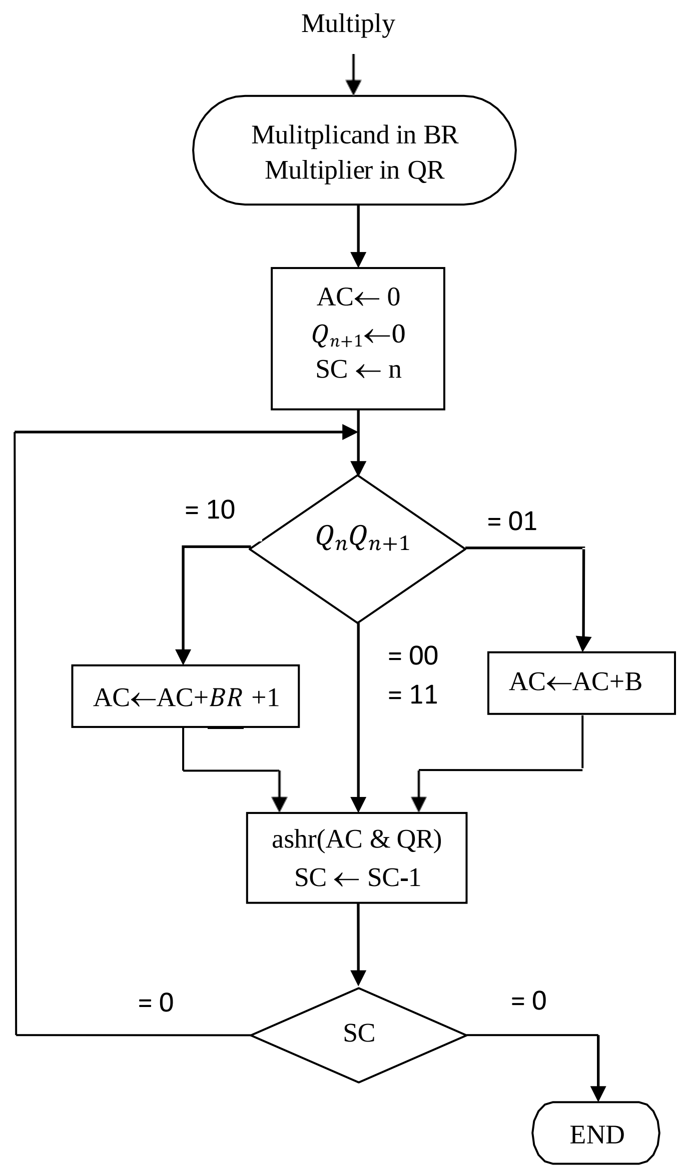
Sequence counter Sc is set to the number n which is equal to the number of bits of bits In themultiplier.

Q_nQ_{n+1} are to successive bits in the multiplier

QnQn+1	BR = 1011 ,BR'+1 = 01001	AC	QR	Qn+1	SC
10	Initial	00000	10011	0	101

	Subtract BR ashr	01001 01001 00100	 11001	 1	 100
11	ashr	00010	01100	1	011
01	Add BR ashr	10111 11001 11100	 10110	 0	 010
00	ashr	11110	01011	0	001
10	Subtract BR Ashr	01001 00111 00011	 10101	 1	 000

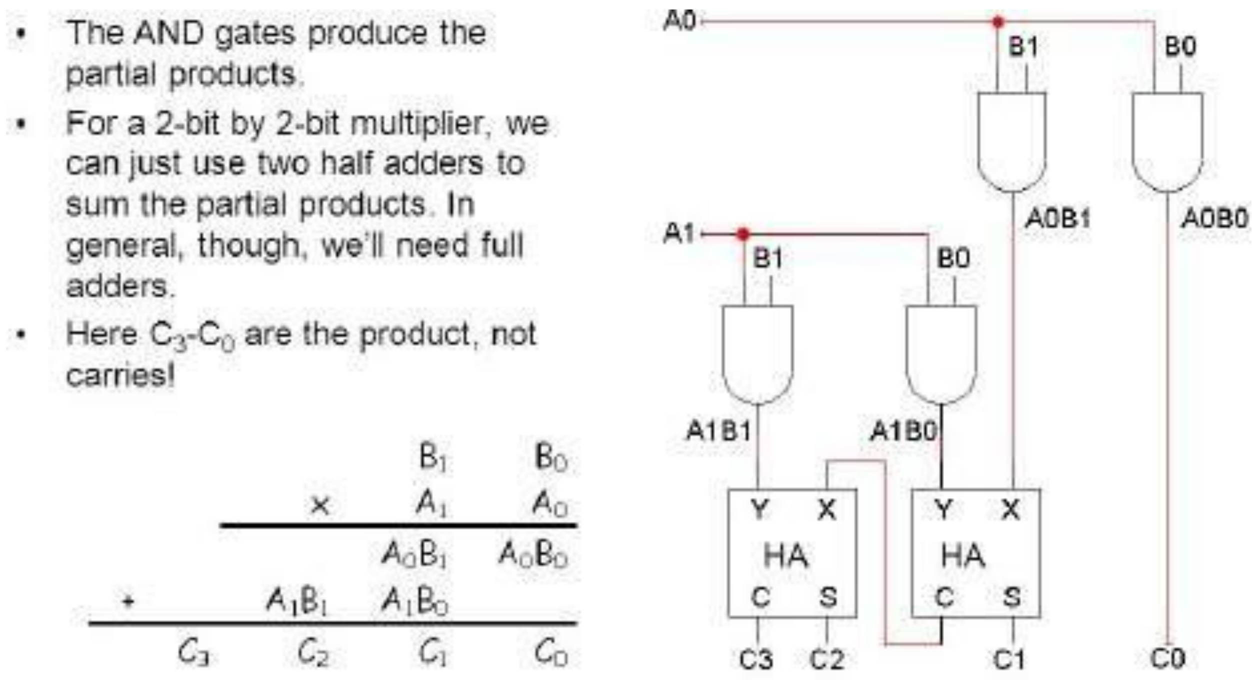
Example for multiplication using Booth algorithm



Algorithm in flowchart for multiplication of signed 2's complement numbers

Array Multiplier:
2 -bit by 2- bit Array Multiplier:

Multiplicand bits are b_1 and b_0 .Multiplier bits are a_1 and a_0 .The first partial product is obtained by multiplying a_0 by b_1 b_0 . The bit multiplication is implemented by AND gate. First partial product is made by two AND gates. Second partial product is made by two AND gates. The two partial products are addedwith two half adder circuits.



- The AND gates produce the partial products.
- For a 2-bit by 2-bit multiplier, we can just use two half adders to sum the partial products. In general, though, we'll need full adders.
- Here C_3-C_0 are the product, not carries!

Fig: 2-bit by 2-bit array multiplierCombinational

circuit binary multiplier:
A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there bits in the multiplier. The binary output in each level of the AND gates is added in parallel with the partial product of the previous level to form a ne partial product. The last level produces the product. For j multiplier andk multiplicand bits, we need j*k AND Gates and (j-1)*k bit adders to ptoduce a product of j+k bits.
4- bit by 3-bit Array Multiplier:

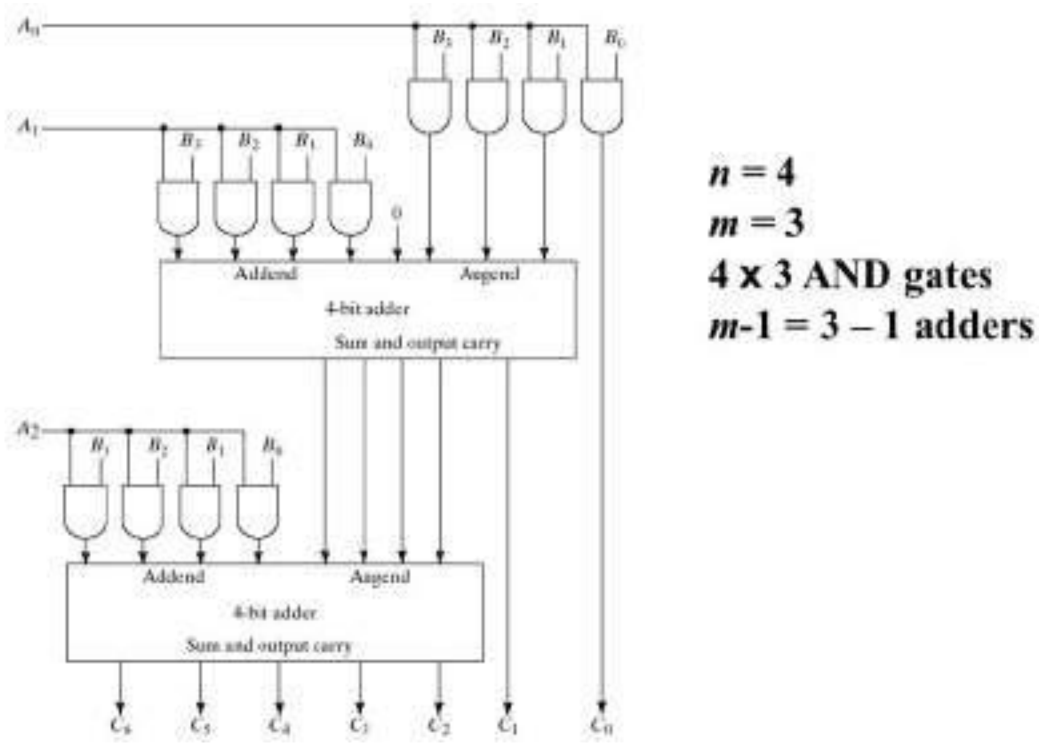
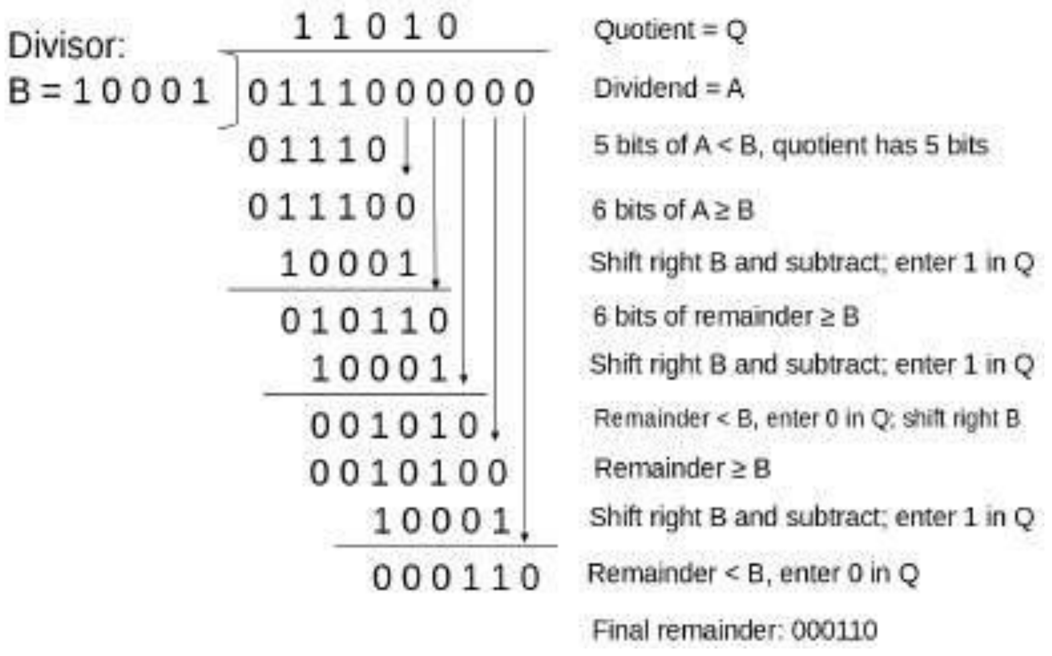


Fig: 4- bit by 3-bit Array Multiplier

DIVISION ALGORITHMS:

Division Process for division of fixed point binary number in signed –magnitude representation:



```
                         11010        Quotient = Q
Divisor:
B = 10001       01110000000           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              6 bits of remainder ≥ B
                   10001              Shift right B and subtract; enter 1 in Q
                   001010             Remainder < B, enter 0 in Q; shift right B
                   0010100            Remainder ≥ B
                      10001           Shift right B and subtract; enter 1 in Q
                      000110          Remainder < B, enter 0 in Q
                                      Final remainder: 000110
```

Fig: Example of Binary Division

Let dividend A consists of 10 bits and divisor B consists of 5 bits.

1. Compare the 5 most significant bits of the dividend with that of divisor.

2. If the 5 bit number is smaller than divisor B, then take 6 bits of the dividend and compare with the 5 bit divisor.

3. The 6 bit number is greater than divisor B. Hence place a 1 for the quotient bit in the sixth position above the dividend. Shift the divisor once to the right and subtracted from the dividend. The difference is called partial remainder.

4. Repeat the process with the partial remainder and divisor. If the partial remainder is equal or 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 small than the divisor, then the quotient bit is zero and no subtraction is needed. The divisor is shifted once to the right in any case,.

Hardware Implementation of division for signed magnitude fixed point numbers:

To implement division using a digital computer, the process is changed slightly forconvenience.

1. Instead of shifting the divisor to the right, the dividend or the partial remainder, is shifted tothe left so as to leave the two numbers in the required relative position.

2. Subtraction may be achieved by adding A (dividend)to the 2's complement of B(divisor). Theinformation about the relative magnitude is then available from end carry.

3. Register EAQ is now shifted to the left with 0 inserted into Qn and the previous value of E islost..

4. The divisor is stored in B register and the double length dividend is stored in registers A andQ.

5. The dividend is shifted to the left and the divisor is subtracted by adding it's 2's complement value.

6. If E= 1, it signifies that $A \geq B$.A quotient bit is inserted into Qnand the partial remainder is shifted to the left to repeat the process.

7. If E = 0, it signifies that $A < B$ so the quotient Qn remains 0(inserted during the shift). 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 5 quotient bits are formed.

8. At the end Q contains the quotient and A the remainder. If the sign of dividend and divisor arealike, the quotient is positive and if unalike, it is negative. The sign of the remainder is the same as dividend.

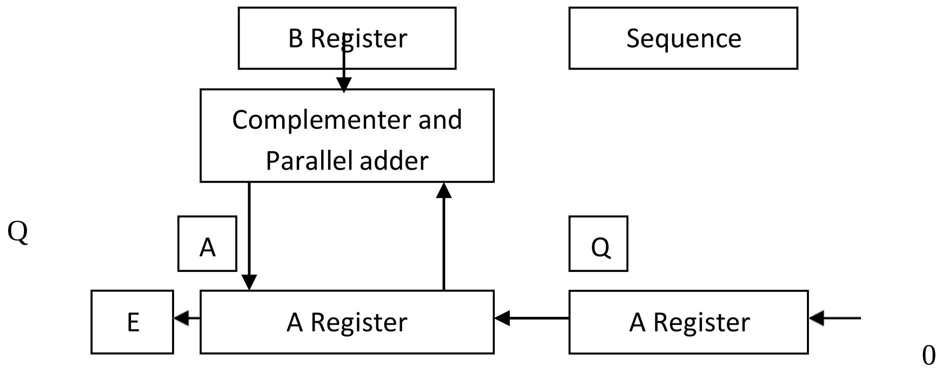


Fig: Hardware for implementing division of fixed point signed- Magnitude Numbers

UNIT 3 - DATA REPRESENTATION

DATA TYPES:

- Registers contain either data or control information
- Control information is a bit or group of bits used to specify the sequence of command signals needed for data manipulation
- Data are numbers and other binary-coded information that are operated on
- Possible data types in registers:
  - o Numbers used in computations
  - o Letters of the alphabet used in data processing
  - o Other discrete symbols used for specific purposes
- All types of data, except binary numbers, are represented in binary-coded form
- A number system of *base*, or *radix*, *r* is a system that uses distinct symbols for *r* digits
- Numbers are represented by a string of digit symbols
- The string of digits 724.5 represents the quantity

  $7 \times 10^{2} + 2 \times 10^{1} + 4 \times 10^{0} + 5 \times 10^{-1}$
- The string of digits 101101 in the binary number system represents the quantity

  $1 \times 2^{5} + 0 \times 2^{4} + 1 \times 2^{3} + 1 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0} = 45$
- $(101101)_{2} = (45)_{10}$
- We will also use the octal (radix 8) and hexidecimal (radix 16) number systems

  $(736.4)_{8} = 7 \times 8^{2} + 3 \times 8^{1} + 6 \times 8^{0} + 4 \times 8^{-1} = (478.5)_{10}$

  $(F3)_{16} = F \times 16^{1} + 3 \times 16^{0} = (243)_{10}$
- Conversion from decimal to radix *r* system is carried out by separating the number into its integer and fraction parts and converting each part separately
- Divide the integer successively by *r* and accumulate the remainders
- Multiply the fraction successively by *r* until the fraction becomes zero

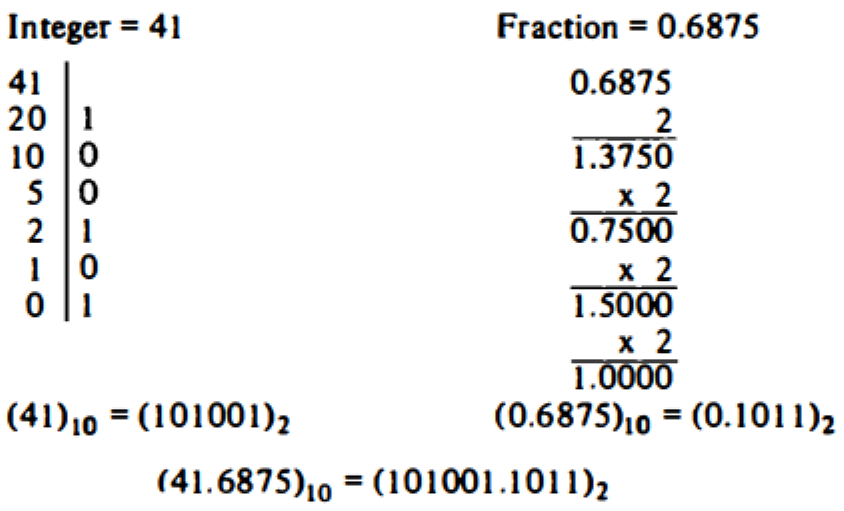


Fig: conversion of decimal 41.6875 to binary

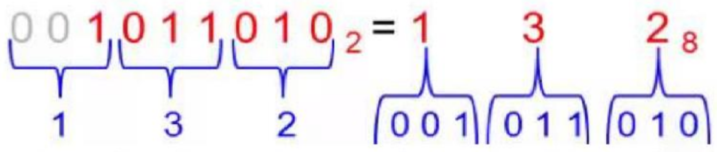
- Each octal digit corresponds to three binary digits
- Each hexadecimal digit corresponds to four binary digits
- Rather than specifying numbers in binary form, refer to them in octal or hexadecimal and reduce the number of digits by 1/3 or ¼, respectively

Binary ↔ Octal ↔ Hex Shortcut

Because binary, octal, and hex number systems are all powers of two (which is the reason we use them) there is a relationship that we can exploit to make conversion easier.

$$1011010_2 = 132_8 = 5A_H$$

To convert directly between binary and octal, group the binary bits into sets of 3 (because $2^3 = 8$). You may need to pad with leading zeros.



To convert directly between binary and hexadecimal number systems, group the binary bits into sets of 4 (because $2^4 = 16$). You may need to pad with leading zeros.

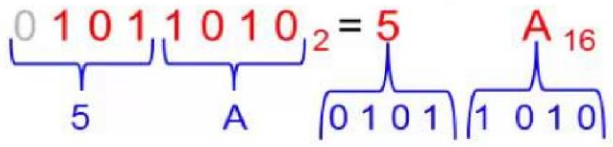
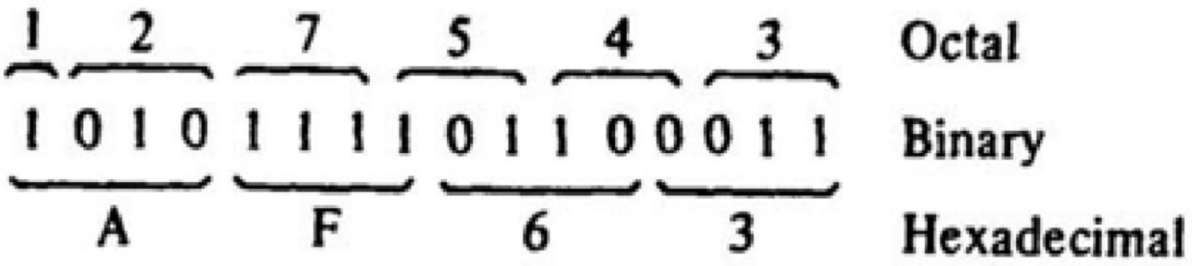


Fig: Binary, Octal and Hexadecimal Conversion



Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	↑
1	001	1	
2	010	2	Code
3	011	3	for one
4	100	4	octal
5	101	5	digit
6	110	6	
7	111	7	↓
10	001 000	8	
11	001 001	9	
12	001 010	10	
24	010 100	20	
62	110 010	50	
143	001 100 011	99	
370	011 111 000	248	

Table: Binary-Coded Octal Numbers

Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	↑
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	Code
7	0111	7	for one
8	1000	8	hexadecimal
9	1001	9	digit
A	1010	10	
B	1011	11	
C	1100	12	
D	1101	13	
E	1110	14	
F	1111	15	↓
14	0001 0100	20	
32	0011 0010	50	
63	0110 0011	99	
F8	1111 1000	248	

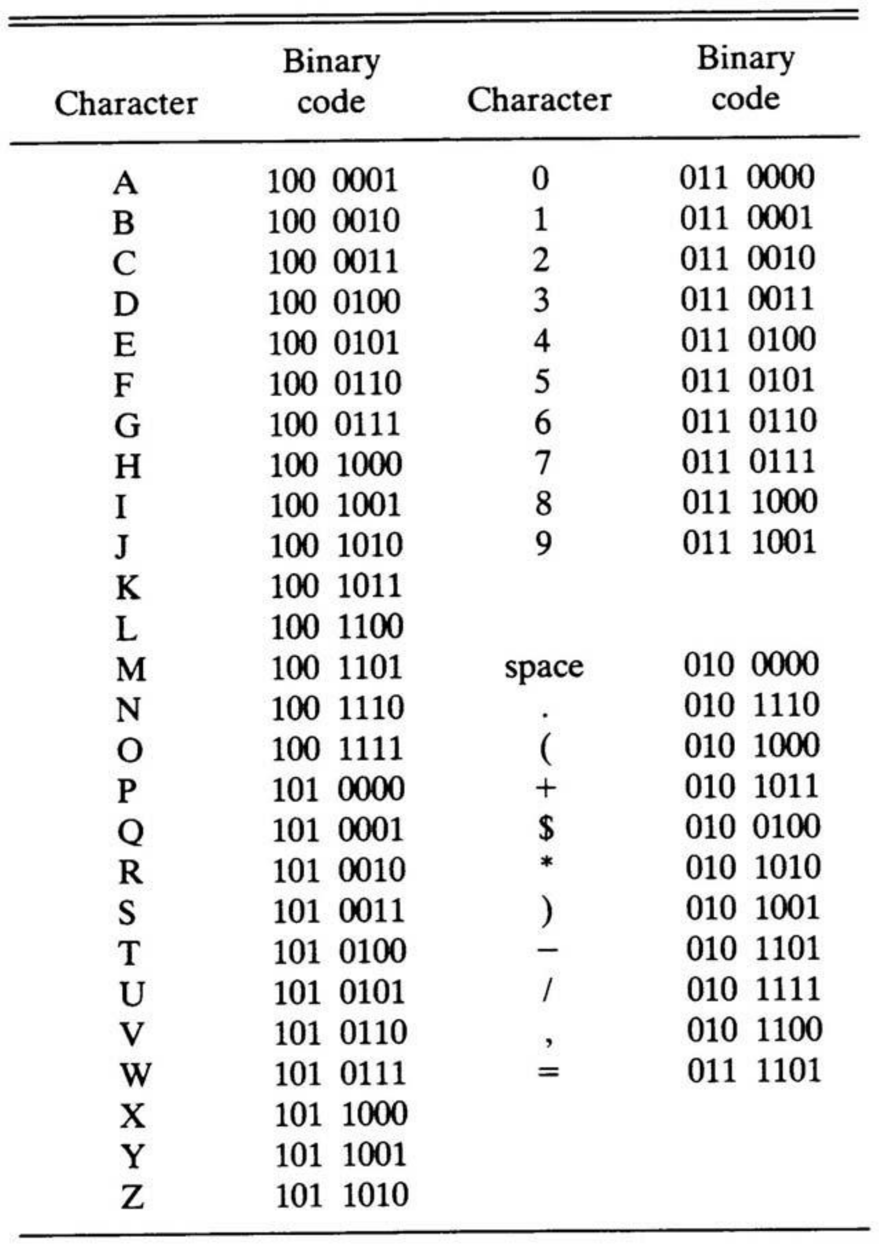
Table: Binary-Coded Hexadecimal Numbers

•A binary code is a group of *n* bits that assume up to 2^n distinct combinations
• A four bit code is necessary to represent the ten decimal digits – 6 are unused
• The most popular decimal code is called *binary-coded decimal* (BCD)
• BCD is different from converting a decimal number to binary
• For example 99, when converted to binary, is 1100011
• 99 when represented in BCD is 1001 1001

Decimal number	Binary-coded decimal (BCD) number	
0	0000	↑
1	0001	
2	0010	
3	0011	Code
4	0100	for one
5	0101	decimal
6	0110	digit
7	0111	
8	1000	
9	1001	↓
10	0001 0000	
20	0010 0000	
50	0101 0000	
99	1001 1001	
248	0010 0100 1000	

•The standard alphanumeric binary code is ASCII
• This uses seven bits to code 128 characters
• Binary codes are required since registers can hold binary information only

TABLE 3-4 American Standard Code for Information Interchange (ASCII)



Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

COMPLEMENTS :

• Complements are used in digital computers for simplifying subtraction and logical manipulation
• Two types of complements for each base *r* system: *r*'s complement and (*r* – 1)'s complement
• Given a number *N* in base *r* having *n* digits, the (*r* – 1)'s complement of *N* is defined as $(r^n - 1) - N$
• For decimal, the 9's complement of *N* is $(10^n - 1) - N$
• The 9's complement of 546700 is 999999 – 546700 = 453299

□□The 9's complement of 453299 is 999999 – 453299 = 546700
• For binary, the 1's complement of *N* is $(2^n - 1) - N$
• The 1's complement of 1011001 is 1111111 – 1011001 = 0100110

- The 1's complement is the true complement of the number – just toggle all bits
- The *r*'s complement of an *n*-digit number *N* in base *r* is defined as $r^n - N$
- This is the same as adding 1 to the $(r - 1)$'s complement
- The 10's complement of 2389 is 7610 + 1 = 7611
- The 2's complement of 101100 is 010011 + 1 = 010100
- Subtraction of unsigned *n*-digit numbers: $M - N$
  - o Add *M* to the *r*'s complement of *N* – this results in $M + (r^n - N) = M - N + r^n$
  - o If $M \geq N$, the sum will produce an end carry r^n which is discarded
  - o If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the *r*'s complement of $(N - M)$. To obtain the answer in a familiar form, take the *r*'s complement of the sum and place a negative sign in front.

Example: 72532 – 13250 = 59282. The 10's complement of 13250 is 86750.

M = 72352
10's comp. of N <u>= +86750</u>
Sum = 159282
Discard end carry <u>= -100000</u>
Answer = 59282
Example for M < N: 13250 – 72532 = -59282
M = 13250
10's comp. of N <u>= +27468</u>
Sum = 40718
No end carry
Answer = -59282 (10's comp. of 40718)
Example for X = 1010100 and Y = 1000011
X = 1010100
2's comp. of Y <u>= +0111101</u>
Sum = 10010001
Discard end carry <u>= -10000000</u>
Answer X – Y = 0010001
Y = 1000011

2's comp. of X <u>= +0101100</u>
Sum = 1101111
No end carry
Answer = -0010001 (2's comp. of 1101111)

<u>FIXED-POINT REPRESENTATION</u> :

- Positive integers and zero can be represented by unsigned numbers
- Negative numbers must be represented by signed numbers since + and – signs are not available, only 1's and 0's are
- Signed numbers have msb as 0 for positive and 1 for negative – msb is the sign bit
- Two ways to designate binary point position in a register
  - o Fixed point position
  - o Floating-point representation
- Fixed point position usually uses one of the two following positions
  - o A binary point in the extreme left of the register to make it a fraction
  - o A binary point in the extreme right of the register to make it an integer
  - o In both cases, a binary point is not actually present
- The floating-point representations uses a second register to designate the position of the binary point in

the first register

When an integer is positive, the msb, or sign bit, is 0 and the remaining bits represent the magnitude

- When an integer is negative, the msb, or sign bit, is 1, but the rest of the number can be represented in one of three ways
  - o Signed-magnitude representation
  - o Signed-1's complement representation
  - o Signed-2's complement representation
- Consider an 8-bit register and the number +14
  - o The only way to represent it is 00001110
- Consider an 8-bit register and the number –14
  - o Signed magnitude: 1 0001110
  - o Signed 1's complement: 1 1110001
  - o Signed 2's complement: 1 1110010
- Typically use signed 2's complement
- Addition of two signed-magnitude numbers follow the normal rules
  - o If same signs, add the two magnitudes and use the common sign
  - o Differing signs, subtract the smaller from the larger and use the sign of the larger magnitude
  - o Must compare the signs and magnitudes and then either add or subtract
- Addition of two signed 2's complement numbers does not require a comparison or subtraction – only addition and complementation
  - o Add the two numbers, including their sign bits
  - o Discard any carry out of the sign bit position
  - o All negative numbers must be in the 2's complement form
  - o If the sum obtained is negative, then it is in 2's complement form

+6 00000110 -6 11111010 +13 00001101 +13 00001101
+19 00010011 +7 00000111
+6 00000110 -6 11111010
-13 11110011 -13 11110011
-7 11111001 -19 11101101

- Subtraction of two signed 2's complement numbers is as follows
  - o Take the 2's complement form of the subtrahend (including sign bit)
  - o Add it to the minuend (including the sign bit)
  - o A carry out of the sign bit position is discarded
- An *overflow* occurs when two numbers of *n* digits each are added and the sum occupies *n* + 1 digits
- Overflows are problems since the width of a register is finite
- Therefore, a flag is set if this occurs and can be checked by the user
- Detection of an overflow depends on if the numbers are signed or unsigned
- For unsigned numbers, an overflow is detected from the end carry out of the msb
- For addition of signed numbers, an overflow cannot occur if one is positive and one is negative – both have to have the same sign
- An overflow can be detected if the carry into the sign bit position and the carry out of the sign bit position are not equal

+70 0 1000110 -70 1 0111010
+80 0 1010000 -80 1 0110000
+150 1 0010110 -150 0 1101010

The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit

- A 4-bit decimal code requires four flip-flops for each decimal digit
- This takes much more space than the equivalent binary representation and the circuits required to

perform decimal arithmetic are more complex

- Representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary
- Either signed magnitude or signed complement systems
- The sign of a number is represented with four bits
  - o 0000 for +
  - o 1001 for –
- To obtain the 10’s complement of a BCD number, first take the 9’s complement and then add one to the least significant digit
- Example: (+375) + (-240) = +135

0 375 $(0000\ 0011\ 0111\ 1010)_{BCD}$

+9 760 $(1001\ 0111\ 0110\ 0000)_{BCD}$

0 135 $(0000\ 0001\ 0011\ 0101)_{BCD}$

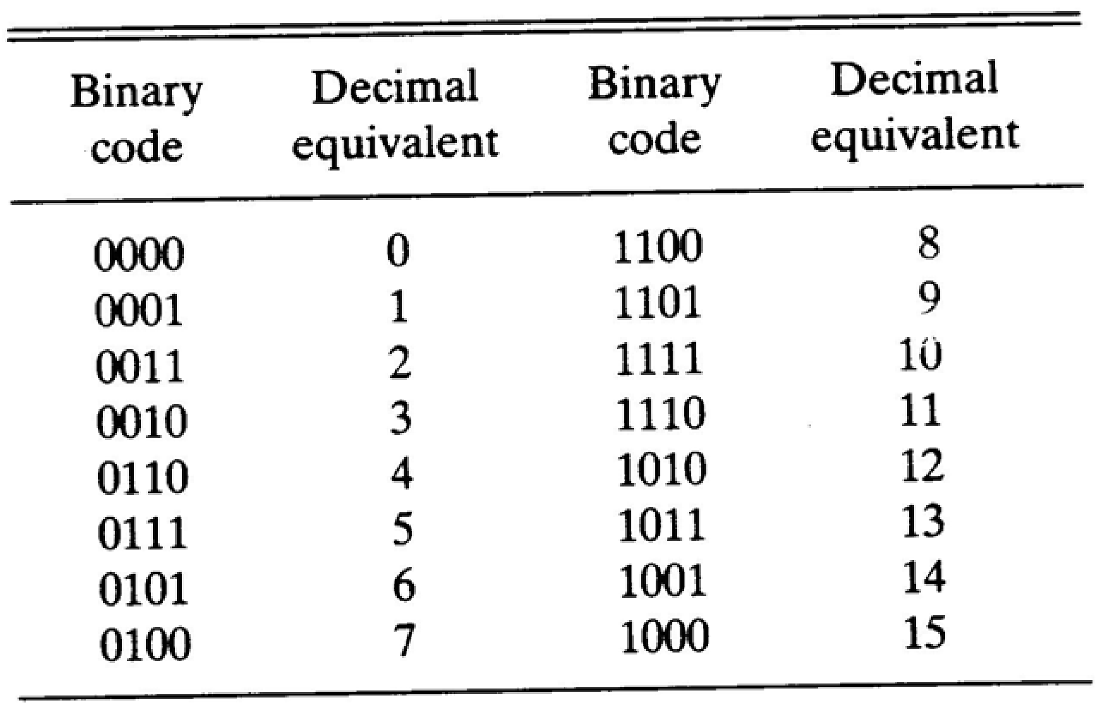
<u>FLOATING-POINT REPRESENTATION</u> :

- The floating-point representation of a number has two parts
- The first part represents a signed, fixed-point number – the *mantissa*
- The second part designates the position of the binary point – the *exponent*
- The mantissa may be a fraction or an integer
- Example: the decimal number +6132.789 is
  - o Fraction: +0.6123789
  - o Exponent: +04
  - o Equivalent to $+0.6132789 \times 10^{+4}$
- A floating-point number is always interpreted to represent $m \times r^{e}$
- Example: the binary number +1001.11 (with 8-bit fraction and 6-bit exponent)
  - o Fraction: 01001110
  - o Exponent: 000100
  - o Equivalent to $+(.1001110)_2 \times 2^{+4}$
- A floating-point number is said to be *normalized* if the most significant digit of the mantissa is nonzero
- The decimal number 350 is normalized, 00350 is not
- The 8-bit number 00011010 is not normalized
- Normalize it by fraction = 11010000 and exponent = -3
- Normalized numbers provide the maximum possible precision for the floating-point number

Other Binary Codes

- Digital systems can process data in discrete form only
- Continuous, or analog, information is converted into digital form by means of an analog-to-digital converter
- The reflected binary or *Gray code,* is sometimes used for the converted digital data
- The Gray code changes by only one bit as it sequences from one number to the next
- Gray code counters are sometimes used to provide the timing sequences that control the operations in a digital system

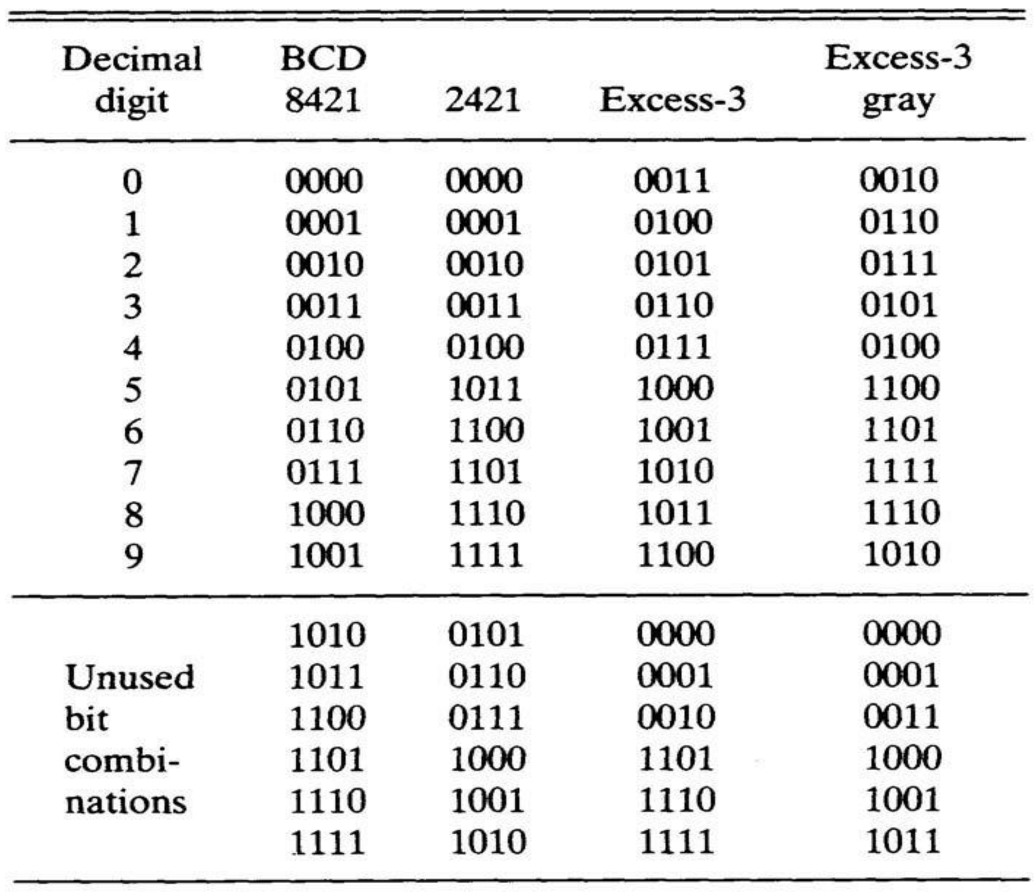
TABLE 3-5 4-Bit Gray Code



Binary code	Decimal equivalent	Binary code	Decimal equivalent
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

- Binary codes for decimal digits require a minimum of four bits
- Other codes besides BCD exist to represent decimal digits

TABLE 3-6 Four Different Binary Codes for the Decimal Digit



Decimal digit	BCD 8421	2421	Excess-3	Excess-3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
	1010	0101	0000	0000
Unused bit combi- nations	1011	0110	0001	0001
	1100	0111	0010	0011
	1101	1000	1101	1000
	1110	1001	1110	1001
	1111	1010	1111	1011

• The 2421 code and the excess-3 code are both *self-complementing*
• The 9's complement of each digit is obtained by complementing each bit in the code
• The 2421 code is a *weighted code*
• The bits are multiplied by indicated weights and the sum gives the decimal digit
• The excess-3 code is obtained from the corresponding BCD code added to 3

Error Detection Codes

• Transmitted binary information is subject to noise that could change bits 1 to 0 and vice versa
• An *error detection code* is a binary code that detects digital errors during transmission
• The detected errors cannot be corrected, but can prompt the data to be retransmitted
• The most common error detection code used is the *parity bit*

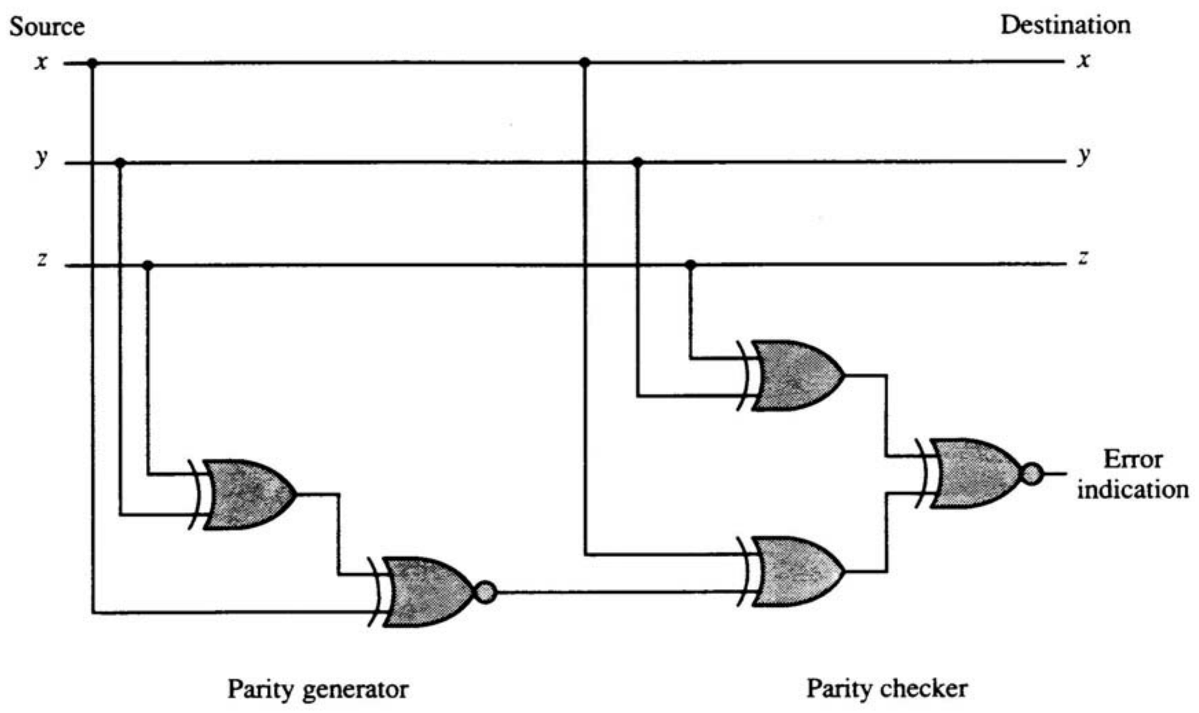
A parity bit is an extra bit included with a binary message to make the total number of 1's either odd or even

TABLE 3-7 Parity Bit Generation

Message *xyz*	*P*(odd)	*P*(even)
000	1	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	0	1

The P(odd) bit is chosen to make the sum of 1's in all four bits odd
• The even-parity scheme has the disadvantage of having a bit combination of all 0's
• Procedure during transmission:
  o At the sending end, the message is applied to a *parity generator*
  o The message, including the parity bit, is transmitted
  o At the receiving end, all the incoming bits are applied to a *parity checker*
  o Any odd number of errors are detected
• Parity generators and checkers are constructed with XOR gates (odd function)
• An odd function generates 1 iff an odd number if input variables are 1

Figure 3-3 Error detection with odd parity bit.


Source
x
y
z
Destination
x
y
z
Error indication
Parity generator
Parity checker