DATA TYPES

•*Binary information is stored in **memory** or **processor registers.***

• *Registers contain either **data** or **control information.***

• *Data are **numbers** and other **binary-coded information***

• *Control information is a bit or a group of bits used to specify the sequence of command signals* **needed for manipulation of the data in other registers**

Data types found in the registers of digital computers may be Classified as being one of the following categories:

1) ***Numbers used in arithmetic computations***
2) ***Letters of the alphabet used in data processing***
3) ***Other discrete symbols used for specific purpose***

Number Systems

- All types of data, except binary numbers, are represented in computer registers in binary-coded form.
- This is because registers are made up of flip-flops and flip-flops are two-state devices that can store only l's and O's.
- ***Base or Radix r system*** *: uses distinct symbols for r digits*
- ***Most common number system*** *:Decimal, Binary, Octal,*

  Hexadecimal
- ***Decimal System/Base****-10 System*
- *Composed of 10 symbols or numerals(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0)*
- The string of digits 724.5 is interpreted to represent the as
- $7 \times 10^2 + 2 \times 10^1 + 4 \times 10^{\circ} + 5 \times 10^{-1}$
- ***Binary System/Base****-2 System*
- *Composed of 10 symbols or numerals(0, 1)*
- *Bit = Binary digit*
- ***Hexadecimal System/Base****-16 System*
- *Composed of 16 symbols or numerals(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D,*
- *E, F)*

Number Systems

- ***Binary-to-Decimal Conversions***
- ***101101***
- $1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$
- ***Decimal-to-Binary Conversions***
- *Repeated division*
- *37 / 2 = 18 remainder 1 (binary number will end with 1) :* ***LSB***
- *18 / 2 = 9 remainder 0*
- *9 / 2 = 4 remainder 1*
- *4 / 2 = 2 remainder 0*
- *2 / 2 = 1 remainder 0*
- *1 / 2 = 0 remainder 1 (binary number will start with 1) :* ***MSB***

Number Systems

Octal to Decimal:

- $(736.4)_8 = 7 \text{ X } 8^2 + 3 \text{ X } 8^1 + 6 \text{ X } 8^{\circ} + 4\text{X}8^{-1}$
- $= 7\text{X}64 + 3\text{X}8 + 6\text{X}1 + 4/8 = (478.5)_{10}$

Hex-to-Decimal Conversion:

$2AF_{16} = (2 \times 16^2) + (10 \times 16^1) + (15 \times 16^O)$

$= 512_{10} + 160_{10} + 15_{10}$

$= 687_{(10)}$

- ***Decimal-to-Hex Conversion***
- *423_{10} / 16 = 26 remainder 7 (Hex number will end with 7) :* ***LSB***
- *26_{10} / 16 = 1 remainder 10*
- *1_{10} / 16 = 0 remainder 1 (Hex number will start with 1) :* ***MSB***
- *Read the result upward to give an answer of 42310 = $1A7_{16}$*

Number Systems

- **41.6875**
- **Integer = 41** **Fraction = 0.6875**
- **41** **0.6875**
- **20 |1** **2**
- **10 | 0** **1.3750**
- **5 | 0** **x 2**
- **2 |1** **0.7500**
- **1 |0** **x 2**
- **0 | 1** **1.5000**

x 2

1 .0000

- **(41) 10 = $(101001)_2$** **(0.6875) 10 = $(0.101\ 1)_2$**
- **(41.6875)10 = $(101001.1011)_2$**
- ***Hex-to-Binary Conversion***

9F216 = 9 F 2

= 1001 1111 0010

= 100111110010_2

- ***Binary-to-Hex Conversion***

1 1 1 0 1 0 0 1 1 0_2 = 0 0 1 1 1 0 1 0 0 1 1 0

3 A 6

= $3A6_{16}$

Number Systems

TABLE 3-3 Binary-Coded Decimal (BCD) Numbers

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

TABLE 3-2 Binary-Coded Hexadecimal Numbers

Hexadecimal number	Binary-coded hexadecimal	Decimal equivalent	
0	0000	0	Code for one hexadecimal digit
1	0001	1	
2	0010	2	
3	0011	3	
4	0100	4	
5	0101	5	
6	0110	6	
7	0111	7	
8	1000	8	
9	1001	9	
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 3-1 Binary-Coded Octal Numbers

Octal number	Binary-coded octal	Decimal equivalent	
0	000	0	Code for one octal digit
1	001	1	
2	010	2	
3	011	3	
4	100	4	
5	101	5	
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	

Number Systems

- ***Binary-Coded-Decimal Code***

  Each digit of a decimal number is represented by its binary equivalent

  8 7 4 ***(Decimal)***

  1000 0111 0100 ***(BCD)***

- *Only the four bit binary numbers from 0000 through 1001 are used*
- *Comparison of BCD and Binary*
- $137_{10} = 10001001_2$ ***(Binary) - require only 8 bits***
- 137_{10} *= 0001 0011 0111BCD* ***(BCD) - require 12 bits***

Fixed-Point Representation

- Positive integers, including zero, can be represented as unsigned numbers.
- However, to represent negative integers, we need a notation for negative values.
- In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign.
- Because of hardware limitations, computers must represent everything with 1's and 0's, including the sign of a number.
- As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number.
- The convention is to make the sign bit equal to 0 for positive and to 1 for negative.

Fixed-Point Representation

- In addition to the sign, a number may have a binary (or decimal) point.
- The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers.
- The representation of the binary point in a register is complicated by the fact that it is characterized by a position in the register.
- **There are two ways of specifying the position of the binary point in a register**: by giving it a **fixed position** or by employing a **floating-point** representation.
- The fixed-point method assumes that the binary point is always fixed in one position.

Fixed-Point Representation

- The two positions most widely used are:

  1) a binary point in the extreme left of the register to make the stored number a fraction, and

  2) a binary point in the extreme right of the register to make the stored number an integer.
- In either case, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer.

Integer Representation

***Signed numbers*:**

- When an integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number.
- When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of three possible ways:

    1. Signed-magnitude representation
    2. Signed-1's complement representation
    3. Signed 2's complement representation

Signed numbers

- The signed-magnitude representation of a negative number consists of the

  magnitude and a negative sign.
- In the other two representations, the negative number is represented in either the 1's or 2's complement of its positive value.
- As an example, consider the signed number 14 stored in an 8-bit register.
- +14 is represented by a sign bit of 0 in the leftmost position followed by the binary equivalent of 14: 00001110.
- Note that each of the eight bits of the register must have a value and therefore 0's must be inserted in the most significant positions following the sign bit.
- Although there is only one way to represent +14, there are three different ways to represent —14 with eight bits.

  In signed-magnitude representation 1 0001110

  In signed-1's complement representation 1 1110001

  In signed-2's complement representation 1 1110010

Signed numbers

- The signed-magnitude representation of —14 is obtained from +14 by complementing only the sign bit.
- The signed-1's complement representation of —14 is obtained by complementing all the bits of +14, including the sign bit.
- The signed-2's complement representation is obtained by taking the 2's complement of the positive number, including its sign bit.

Arithmetic Addition

- The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic.
- If the signs are the same, we add the two magnitudes and give the sum the common sign.
- If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude.
- **For example**, (+25) + (-37) = -(37 - 25) = -12 and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result.
- This is a process that requires the comparison of the signs and the magnitudes and then performing either addition or subtraction.

+6	0000 0110	-6	1111 1010
+13	$\underline{\text{0000 1101}}$	+13	$\underline{\text{0000 1101}}$
+19	00010011	+7	00000111

+6	0000 0110	-6	11111010
-13	$\underline{\text{1111 0011}}$	-13	$\underline{\text{11110011}}$
-7	1111 1001	-19	11101101

- In each of the four cases, the operation performed is always addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2's complement form.

Arithmetic Subtraction

- Subtraction of two signed binary numbers when negative numbers are in 2's subtraction complement form is very simple and can be stated as follows:
- Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit).
- A carry out of the sign bit position is discarded.
- This procedure stems from the fact that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed.
- This is demonstrated by the following relationship:

$$(\pm A) - (+B) = (\pm A) + (-B)$$

$$(\pm A) - (-B) = (\pm A) + (+B)$$

- Consider the subtraction of (-6) - (-13) = +7. In binary with eight bits this is written as 11111010 - 11110011.
- The subtraction is changed to addition by taking the 2' s complement of the subtrahend (- 13) to give (+ 13).
- In binary this is 11111010 + 00001101 = 100000111. Removing the end carry, we obtain the correct answer 00000111 (+7).

Character Representation

- An alphanumeric character set is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, such as $, + , and = .
- Such a set contains between 32 and 64 elements (if only uppercase letters are included) or between 64 and 128 (if both uppercase and lowercase letters are included).
- In the first case, the binary code will require six bits and in the second case, seven bits.
- The standard alphanumeric binary code is the ASCII (American Standard Code for Information Interchange), which uses seven bits to code 128 characters.

ASCCI

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Floating-Point Representation

- The floating-point representation of a number has two parts.
- The first part represents a signed, fixed-point number called the mantissa.
- The second part designates the position of the decimal (or binary) point and is called the exponent
- The fixed-point mantissa may be a fraction or an integer.
- **For example**, the decimal number +6132.789 is represented in floating-point with a fraction and an exponent as follows:

Fraction	Exponent
+0.6132789	+04

Floating-Point Representation

- The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction.
- This representation is equivalent to the scientific notation is

  $+0.6132789 \times 10^{+4}$.

- Floating-point is always interpreted to represent a number in the following form:

  $m \times r^{e}$

- Only the mantissa m and the exponent e are physically represented in the register (including their signs).
- The radix r and the radix-point position of the mantissa are always assumed.

Floating-Point Representation

- A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent.
- **For example**, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

Fraction	Exponent
01001110	000100

- The fraction has a 0 in the leftmost position to denote positive. The binary point
- of the fraction foUows the sign bit but is not shown in the register. The exponent
- has the equivalent binary number +4.

Floating-Point Representation

- The floating-point number is equivalent to

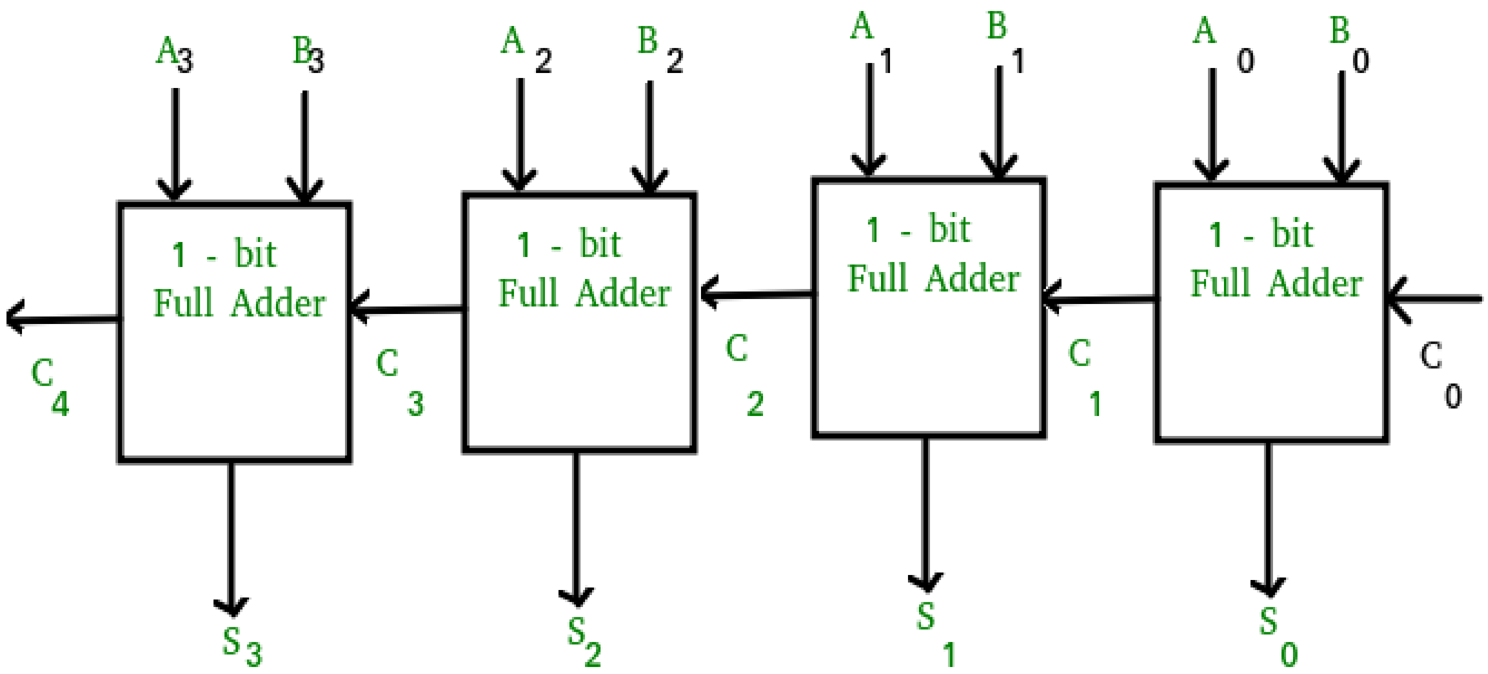
  $m \times 2^e = +(.1001110)_2 \times 2^{+4}$

- A floating-point number is said to be normalized if the most significant digit of the mantissa is nonzero.
- For example, the decimal number 350 is normalized but 00035 is not.
- Regardless of where the position of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero.
- **For example**, the 8-bit binary number 00011010 is not normalized because of the three leading 0' s .
- The number can be normalized by shifting it three positions to the left and discarding the leading 0's to obtain 11010000.
- The three shifts multiply the number by $2^3 = 8$.
- To keep the same value for the floating-point number, the exponent must be subtracted by 3.
- Normalized numbers provide the maximum possible precision for the floating-point number.
- A zero cannot be normalized because it does not have a nonzero digit.

Ripple Carry Adder

- **Motivation behind Carry Look-Ahead Adder :**
  In ripple carry adders, for each adder block, the two bits that are to be added are available instantly.
- However, each adder block waits for the carry to arrive from its previous block.
- So, it is not possible to generate the sum and carry of any block until the input carry is known. The i^{th} block waits for the $i^{th}-1$ block to produce its carry. So there will be a considerable time delay which is carry propagation delay.
- Consider the above 4-bit ripple carry adder.
- The sum is produced by the corresponding full adder as soon as the input signals are applied to it.
- But the carry input is not available on its final steady state value until carry is available at its steady state value.

Ripple Carry Adder



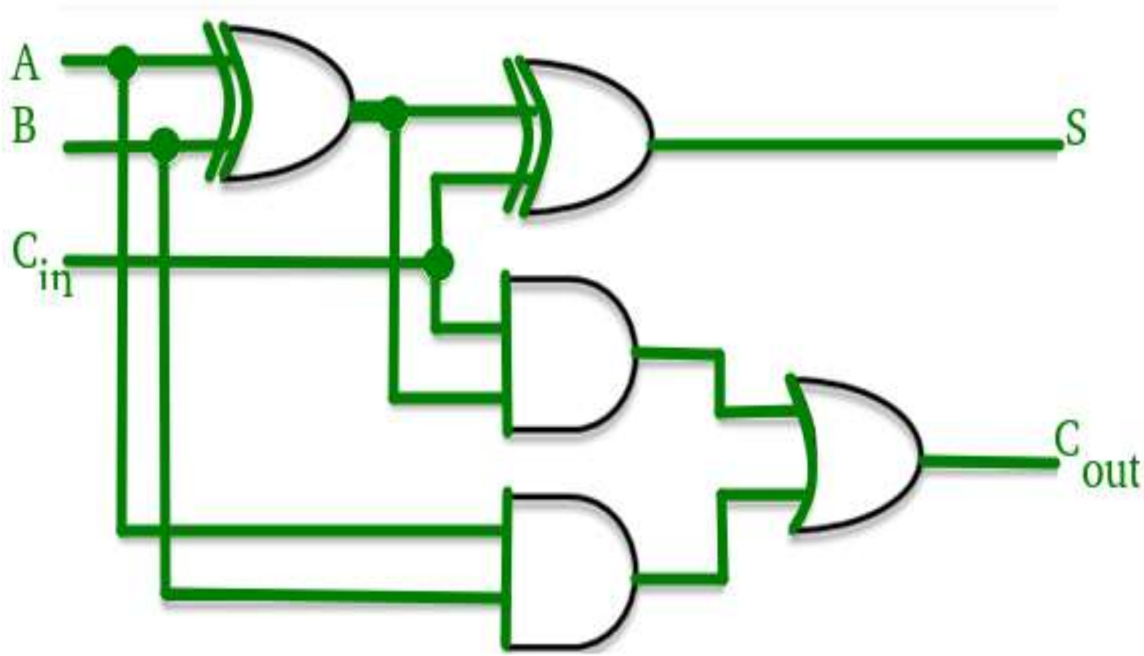
Ripple Carry Adder

- Similarly depends on and on .
- Therefore, though the carry must propagate to all the stages in order that output and carry settle their final steady-state value.
- The propagation time is equal to the propagation delay of each adder block, multiplied by the number of adder blocks in the circuit.
- **For example**, if each full adder stage has a propagation delay of 20 nanoseconds, then will reach its final correct value after 60 (20 × 3) nanoseconds.
- The situation gets worse, if we extend the number of stages for adding more number of bits.

Carry Look-ahead Adder :

- A carry look-ahead adder reduces the propagation delay by introducing more complex hardware.
- In this design, the ripple carry design is suitably transformed such that the carry logic over fixed groups of bits of the adder is reduced to two-level logic.
- Let us discuss the design in detail.

Carry Look-ahead Adder :



Carry Look-ahead Adder

- Consider the full adder circuit shown above with corresponding truth table. We define two variables as **'carry generate G_i '** and **'carry propagate P_i'** then,
- **$P_i = A_i + B_i$(ExclusiveOR)**
- **$G_i = A_i B_i$**
-
- The sum output and carry output can be expressed in terms of carry generate **G_i** and carry propagate P_i as
- **$S_i = P_i + C_i$** (ExclusiveOR)
- **$C_{i+1} = G_i + P_i C_i$**
  where **G_i** produces the carry when both **A_i** , **B_i** are 1 regardless of the input carry. **P_i** is associated with the propagation of carry from **C_i** to **C_{i+1}** .

Carry Look-ahead Adder

- **Advantages and Disadvantages of Carry Look-Ahead Adder :**
  Advantages –
  1) The propagation delay is reduced.
  2) It provides the fastest addition logic.
- **Disadvantages –**

  1) The Carry Look-ahead adder circuit gets complicated as the number of variables increase.

  2)The circuit is costlier as it involves more number of hardware.

Booth Multiplication Algorithm

- Booth algorithm gives a procedure for multiplying binary integers in signed-2's complement representation.
- It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$.
- For example, the binary number 001 110 (+ 14) has a string of 1's from 2^3 to 2^1 (k = 3, m = 1).
- The number can be represented as $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$.
- Therefore, the multiplication M x 14, where M is the multiplicand and 14 the multiplier, can be done as $M \times 2^4 - M \times 2^1$.

Booth Multiplication Algorithm

- Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

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		$\underline{10111}$		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		$\underline{10111}$		
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		$\underline{10111}$		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

Booth Multiplication Algorithm

TABLE 10-3 Example of Multiplication with Booth Algorithm

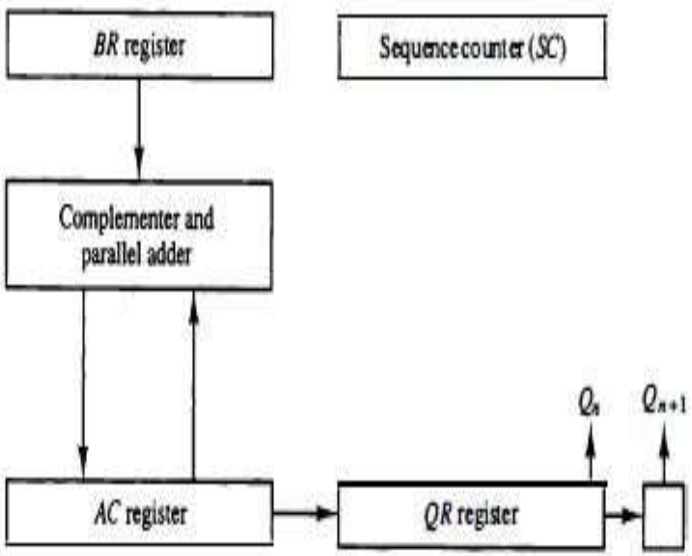
$Q_n\ Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111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

Booth Multiplication Algorithm

- Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift micro-operations.
- The multiplication of two binary numbers can be done with one micro-operation by means of a combinational circuit that forms the product bits all at once.
- This is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array.

Booth Multiplication Algorithm

Figure 10-7 Hardware for Booth algorithm.



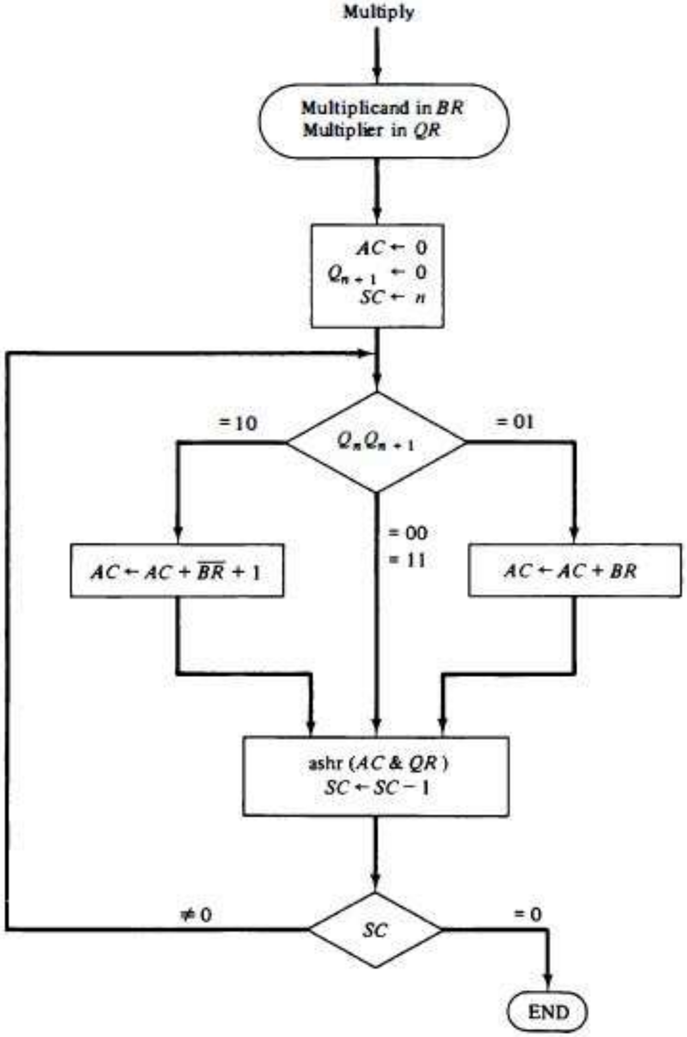
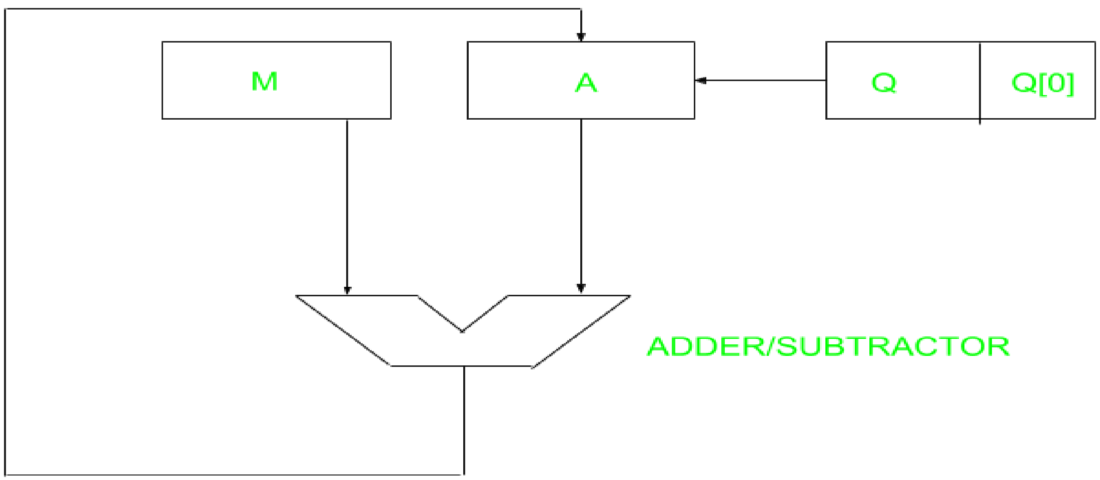


Figure 10-8 Booth algorithm for multiplication of signed-2's complement

Restoring Division Algorithm For Unsigned Integer

- A division algorithm provides a quotient and a remainder when we divide two number.
- They are generally of two type **slow algorithm and fast algorithm**. Slow division algorithm are restoring, non-restoring, non-performing restoring, SRT algorithm.
- Will be performing restoring algorithm for unsigned integer.
- Restoring term is due to fact that value of register A is restored after each iteration.
- Here, register Q contain quotient and register A contain remainder.
- Here, n-bit dividend is loaded in Q and divisor is loaded in M.
- Value of Register is initially kept 0 and this is the register whose value is restored during iteration due to which it is named Restoring.

Restoring Division Algorithm For Unsigned Integer



Restoring Division Algorithm For Unsigned Integer

- **Let's pick the step involved**:
- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend).
- **Step-2:** Then the content of register A and Q is shifted left as if they are a single unit.
- **Step-3:** Then content of register M is subtracted from A and result is stored in A.
- **Step-4:** Then the most significant bit of the A is checked if it is 0 the least significant bit of Q is set to 1 otherwise if it is 1 the least significant bit of Q is set to 0 and value of register A is restored i.e the value of A before the subtraction with M.
- **Step-5:** The value of counter n is decremented.
- **Step-6:** If the value of n becomes zero we get of the loop otherwise we repeat from step 2.
- **Step-7:** Finally, the register Q contain the quotient and A contain remainder.
- **Examples:**
- Perform Division Restoring Algorithm
- Dividend = 11 .
- Divisor = 3.

Restoring Division Algorithm For Unsigned Integer

N	M	A	Q	OPERATION
4	00011	00000	1011	initialize
	00011	00001	011_	shift left AQ
	00011	11110	011_	A=A-M
	00011	00001	0110	Q[0]=0 And restore A
3	00011	00010	110_	shift left AQ
	00011	11111	110_	A=A-M
	00011	00010	1100	Q[0]=0
2	00011	00101	100_	shift left AQ
	00011	00010	100_	A=A-M
	00011	00010	1001	Q[0]=1
1	00011	00101	001_	shift left AQ
	00011	00010	001_	A=A-M
	00011	00010	0011	Q[0]=1

Restoring Division Algorithm For Unsigned Integer

- Remember to restore the value of A most significant bit of A is 1.
- As that register Q contain the quotient, i.e. 3 and register A contain remainder 2

Non-Restoring Division For Unsigned Integer

- In earlier post Restoring Division learned about restoring division.
- Now, here perform Non-Restoring division, it is less complex than the restoring one because simpler operation are involved i.e. addition and subtraction, also now restoring step is performed.
- In the method, rely on the sign bit of the register which initially contain zero named as A.

Non-Restoring Division For Unsigned Integer

- **Let's pick the step involved:**
- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Check the sign bit of register A
- **Step-3:** If it is 1 shift left content of AQ and perform A = A+M, otherwise shift left AQ and perform A = A-M (means add 2's complement of M to A and store it to A)
- **Step-4:** Again the sign bit of register A
- **Step-5:** If sign bit is 1 Q[0] become 0 otherwise Q[0] become 1 (Q[0] means least significant bit of register Q)
- **Step-6:** Decrements value of N by 1
- **Step-7:** If N is not equal to zero go to **Step 2** otherwise go to next step
- **Step-8:** If sign bit of A is 1 then perform A = A+M
- **Step-9:** Register Q contain quotient and A contain remainder
- **Examples:** Perform Non_Restoring Division for Unsigned Integer
- Dividend =11
- Divisor =3
- -M =11101

Non-Restoring Division For Unsigned Integer

- Quotient = 3 (Q)   Remainder = 2 (A)

N	M	A	Q	ACTION
4	00011	00000	1011	Start
		00001	011_	Left shift AQ
		11110	011_	A=A-M
3		11110	0110	Q[0]=0
		11100	110_	Left shift AQ
		11111	110_	A=A+M
2		11111	1100	Q[0]=0
		11111	100_	Left Shift AQ
		00010	100_	A=A+M
1		00010	1001	Q[0]=1
		00101	001_	Left Shift AQ
		00010	001_	A=A-M
0		00010	0011	Q[0]=1

Floating-Point Arithmetic Operations

- Many high-level programming languages have a facility for specifying floating point numbers.
- Any computer that has a compiler for such high-level programming language must have a provision for handling floating-point arithmetic operations.
- The operations are quite often included in the internal hardware.
- If no hardware is available for the operations, the compiler must be designed with a package of floating-point software subroutines.
- Although the hardware method is more expensive, it is so much more efficient than the software method that floating-point hardware is included in most computers and is omitted only in very small ones

Floating-Point Arithmetic Operations

- A floating point number in computer registers consists of two parts: a Mantissa M and an exponent e.
- The two parts represent a number obtained from multiplying m times a radix r raised to the value of e; thus

$$M \times r^{e}$$

The mantissa may be a fraction or an integer.

- **For example**, assume a fraction representation and a radix 10.
- The decimal number 537.25 is represented in a register with m = 53725 and e = 3 and is interpreted to represent the floating-point number

$$.53725 \times 10^{3}$$

Floating-Point Arithmetic Operations

- A floating-point number i s normalized if the most significant digit o f the mantissa is nonzero.
- In this way the mantissa contains the maximum possible number of significant digits.
- A zero cannot be normalized because it does not
- have a nonzero digit.
- It is represented in floating-point by all 0' s in the mantissa and exponent.

Floating-Point Arithmetic Operations

- Adding or subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas.
- The alignment is done by shifting one mantissa while its exponent is adjusted until it is equal to the other exponent.
- Consider the sum of the following floating-point numbers:

$$\begin{array}{r} .5372400 \times 10^{2} \\ +\ .1580000 \times 10^{-1} \end{array}$$

- It is necessary that the two exponents be equal before the mantissas can be added.
- We can either shift the first number three positions to the left, or shift the second number three positions to the right.
- When the mantissas are stored in registers, shifting to the left causes a loss of most significant digits.

Floating-Point Arithmetic Operations

- The usual alignment procedure is to shift the mantissa that has the smaller exponent to the right by a number of places equal to the difference between the exponents.
- After this is done, the mantissas can be added.

$$
\begin{array}{r}
.\ 5372400 \times 10^{2} \\
\underline{+\ .\ 0001580 \times 10^{2}} \\
\underline{.5373980 \times 10^{2}}
\end{array}
$$

1.Check the exponent.

2. Align the mantissas.

3. Add or subtract the mantissas.

4. Normalize the result.

Floating-Point Arithmetic Operations

- When two numbers are **subtracted**,the result may contain most significant zeros as shown in the following

example:

$$\begin{array}{r} \mathbf{.56780 \times 10^5} \\ \underline{\mathbf{-\ .56430 \times 10^5}} \\ \mathbf{.00350 \times 10^5} \end{array}$$

- A floating-point number that has a 0 in the most significant position of the mantissa is said to have an underflow.
- To normalize a number that contains an underflow, it is necessary to shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position.
- In the example above, it is necessary to shift left twice to obtain .35000 X 10^3.
- Floating-point multiplication and division do not require an alignment of the mantissas.