

# MODULE 4: ARITHMETIC

## NUMBERS, ARITHMETIC OPERATIONS AND CHARACTERS

### NUMBER REPRESENTATION

- Numbers can be represented in 3 formats:
  - 1) Sign and magnitude
  - 2) 1's complement
  - 3) 2's complement
- In all three formats, MSB=0 for +ve numbers & MSB=1 for -ve numbers.
- In **sign-and-magnitude system**,  
negative value is obtained by changing the MSB from 0 to 1 of the corresponding positive value.  
For ex, +5 is represented by 0101 & -5 is represented by 1101.
- In **1's complement system**,  
negative values are obtained by complementing each bit of the corresponding positive number.  
For ex, -5 is obtained by complementing each bit in 0101 to yield 1010.  
(In other words, the operation of forming the 1's complement of a given number is equivalent to subtracting that number from  $2^n-1$ ).
- In **2's complement system**,  
forming the 2's complement of a number is done by subtracting that number from  $2^n$ .  
For ex, -5 is obtained by complementing each bit in 0101 & then adding 1 to yield 1011.  
(In other words, the 2's complement of a number is obtained by adding 1 to the 1's complement of that number).
- 2's complement system yields the most efficient way to carry out addition/subtraction operations.

$B$ $b_3 b_2 b_1 b_0$	Values represented		
	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	-0	-7	-8
1 0 0 1	-1	-6	-7
1 0 1 0	-2	-5	-6
1 0 1 1	-3	-4	-5
1 1 0 0	-4	-3	-4
1 1 0 1	-5	-2	-3
1 1 1 0	-6	-1	-2
1 1 1 1	-7	-0	-1

Figure 1.3 Binary, signed-integer representations.

### **ADDITION OF POSITIVE NUMBERS**

- Consider adding two 1-bit numbers.
- The sum of 1 & 1 requires the 2-bit vector 10 to represent the value 2. We say that sum is 0 and the carry-out is 1.

0	1	0	1
+ 0	+ 0	+ 1	+ 1
0	1	1	10
			↑
			Carry-out

Figure 2.2 Addition of 1-bit numbers.

## ADDITION & SUBTRACTION OF SIGNED NUMBERS

- Following are the two rules for addition and subtraction of n-bit signed numbers using the 2's complement representation system (Figure 1.6).

### Rule 1:

- **To Add** two numbers, add their n-bits and ignore the carry-out signal from the MSB position.
- Result will be algebraically correct, if it lies in the range  $-2^{n-1}$  to  $+2^{n-1}-1$ .

### Rule 2:

- **To Subtract** two numbers X and Y (that is to perform  $X-Y$ ), take the 2's complement of Y and then add it to X as in rule 1.
- Result will be algebraically correct, if it lies in the range  $(-2^{n-1})$  to  $+(2^{n-1}-1)$ .
- When the result of an arithmetic operation is outside the representable-range, an arithmetic overflow is said to occur.
- To represent a signed in 2's complement form using a larger number of bits, repeat the sign bit as many times as needed to the left. This operation is called **sign extension**.
- In 1's complement representation, the result obtained after an addition operation is not always correct. The carry-out( $c_n$ ) cannot be ignored. If  $c_n=0$ , the result obtained is correct. If  $c_n=1$ , then a 1 must be added to the result to make it correct.

## OVERFLOW IN INTEGER ARITHMETIC

- When result of an arithmetic operation is outside the representable-range, an **arithmetic overflow** is said to occur.
- For example: If we add two numbers +7 and +4, then the output sum S is 1011( $\leftarrow 0111+0100$ ), which is the code for -5, an incorrect result.
- An overflow occurs in following 2 cases
  - 1) Overflow can occur only when adding two numbers that have the same sign.
  - 2) The carry-out signal from the sign-bit position is not a sufficient indicator of overflow when adding signed numbers.

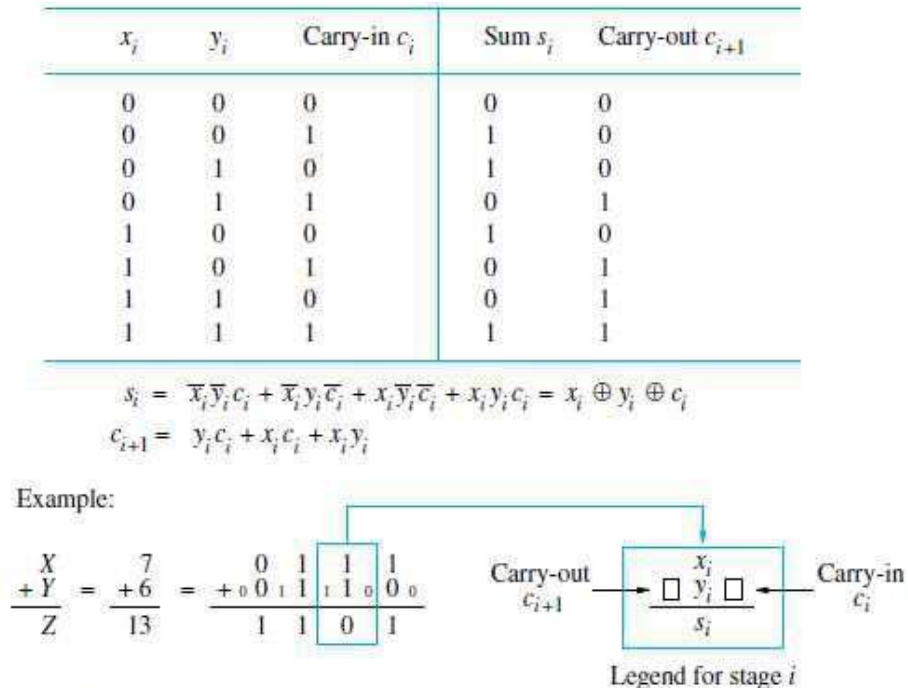
(a)	$\begin{array}{r} 0010 \\ + 0011 \\ \hline 0101 \end{array}$	$\begin{array}{r} (+2) \\ (+3) \\ \hline (+5) \end{array}$	(b)	$\begin{array}{r} 0100 \\ + 1010 \\ \hline 1110 \end{array}$	$\begin{array}{r} (+4) \\ (-6) \\ \hline (-2) \end{array}$
(c)	$\begin{array}{r} 1011 \\ + 1110 \\ \hline 1001 \end{array}$	$\begin{array}{r} (-5) \\ (-2) \\ \hline (-7) \end{array}$	(d)	$\begin{array}{r} 0111 \\ + 1101 \\ \hline 0100 \end{array}$	$\begin{array}{r} (+7) \\ (-3) \\ \hline (+4) \end{array}$
(e)	$\begin{array}{r} 1101 \\ - 1001 \\ \hline \end{array}$	$\begin{array}{r} (-3) \\ (-7) \\ \hline \end{array}$	$\Rightarrow$	$\begin{array}{r} 1101 \\ + 0111 \\ \hline 0100 \end{array}$	$\begin{array}{r} \phantom{(+7)} \\ \phantom{(-3)} \\ \hline (+4) \end{array}$
(f)	$\begin{array}{r} 0010 \\ - 0100 \\ \hline \end{array}$	$\begin{array}{r} (+2) \\ (+4) \\ \hline \end{array}$	$\Rightarrow$	$\begin{array}{r} 0010 \\ + 1100 \\ \hline 1110 \end{array}$	$\begin{array}{r} \phantom{(+2)} \\ \phantom{(+4)} \\ \hline (-2) \end{array}$
(g)	$\begin{array}{r} 0110 \\ - 0011 \\ \hline \end{array}$	$\begin{array}{r} (+6) \\ (+3) \\ \hline \end{array}$	$\Rightarrow$	$\begin{array}{r} 0110 \\ + 1101 \\ \hline 0011 \end{array}$	$\begin{array}{r} \phantom{(+6)} \\ \phantom{(+3)} \\ \hline (+3) \end{array}$
(h)	$\begin{array}{r} 1001 \\ - 1011 \\ \hline \end{array}$	$\begin{array}{r} (-7) \\ (-5) \\ \hline \end{array}$	$\Rightarrow$	$\begin{array}{r} 1001 \\ + 0101 \\ \hline 1110 \end{array}$	$\begin{array}{r} \phantom{(-7)} \\ \phantom{(-5)} \\ \hline (-2) \end{array}$
(i)	$\begin{array}{r} 1001 \\ - 0001 \\ \hline \end{array}$	$\begin{array}{r} (-7) \\ (+1) \\ \hline \end{array}$	$\Rightarrow$	$\begin{array}{r} 1001 \\ + 1111 \\ \hline 1000 \end{array}$	$\begin{array}{r} \phantom{(-7)} \\ \phantom{(+1)} \\ \hline (-8) \end{array}$
(j)	$\begin{array}{r} 0010 \\ - 1101 \\ \hline \end{array}$	$\begin{array}{r} (+2) \\ (-3) \\ \hline \end{array}$	$\Rightarrow$	$\begin{array}{r} 0010 \\ + 0011 \\ \hline 0101 \end{array}$	$\begin{array}{r} \phantom{(+2)} \\ \phantom{(-3)} \\ \hline (+5) \end{array}$

Figure 1.6 2's-complement Add and Subtract operations.

## ADDITION & SUBTRACTION OF SIGNED NUMBERS

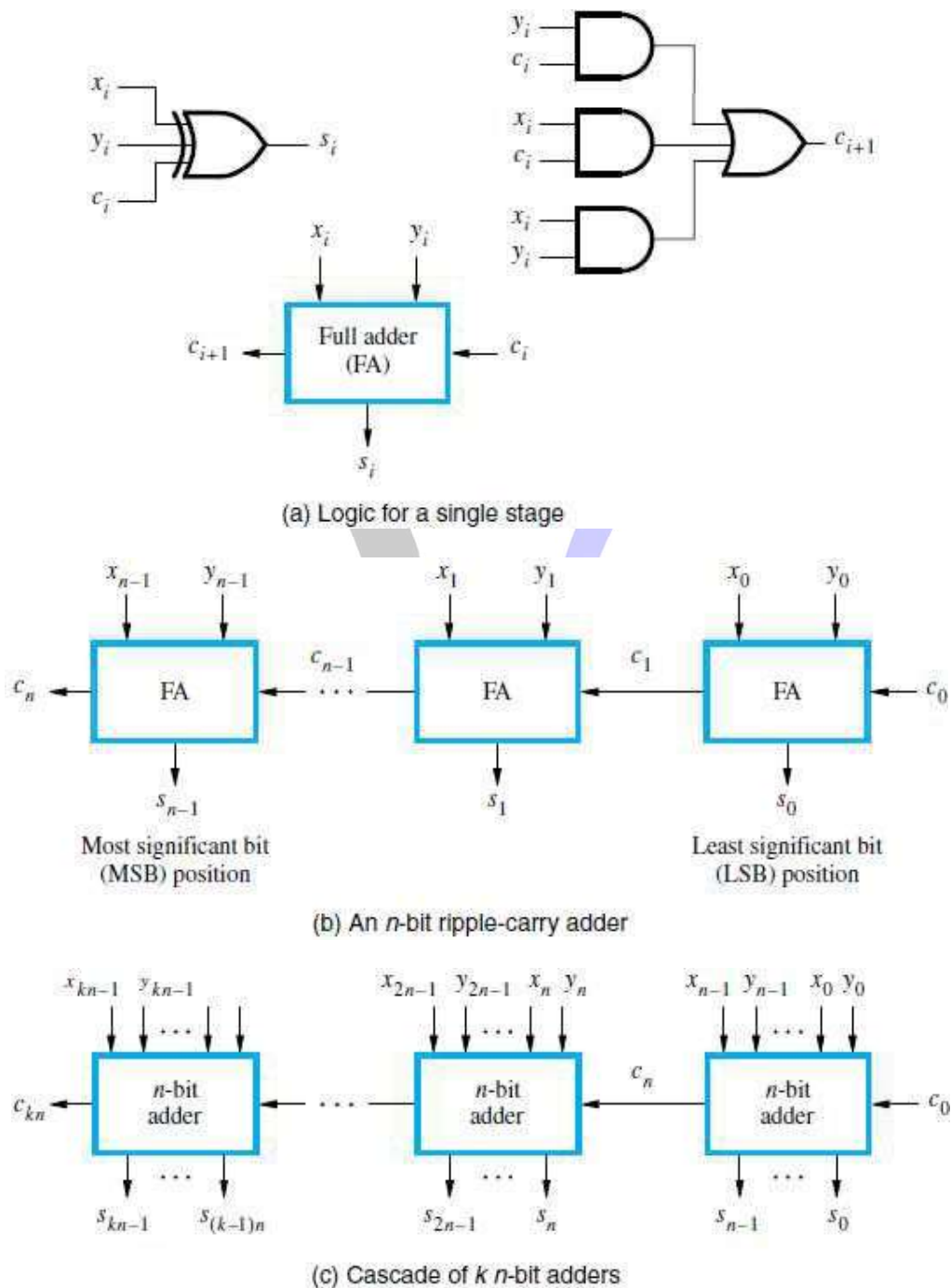
### n-BIT RIPPLE CARRY ADDER

- A cascaded connection of n full-adder blocks can be used to add 2-bit numbers.
- Since carries must propagate (or ripple) through cascade, the configuration is called an n-bit ripple carry adder (Figure 9.1).



**Figure 9.1** Logic specification for a stage of binary addition.

VTU  
Updates



**Figure 9.2** Logic for addition of binary numbers.

### ADDITION/SUBTRACTION LOGIC UNIT

- The  $n$ -bit adder can be used to add 2's complement numbers  $X$  and  $Y$  (Figure 9.3).
- **Overflow** can only occur when the signs of the 2 operands are the same.
- In order to perform the subtraction operation  $X-Y$  on 2's complement numbers  $X$  and  $Y$ ; we form the 2's complement of  $Y$  and add it to  $X$ .
- Addition or subtraction operation is done based on value applied to the Add/Sub input control-line.
- Control-line=0 for addition, applying the  $Y$  vector unchanged to one of the adder inputs.
- Control-line=1 for subtraction, the  $Y$  vector is 2's complemented.

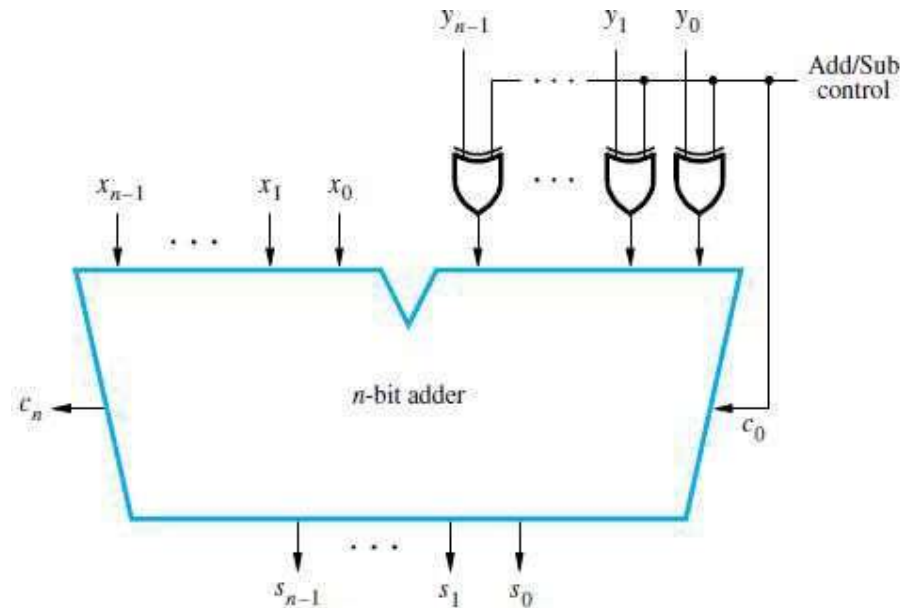


Figure 9.3 Binary addition/subtraction logic circuit.

VTU  
Updates

### DESIGN OF FAST ADDERS

- **Drawback of ripple carry adder:** If the adder is used to implement the addition/subtraction, all sum bits are available in  $2n$  gate delays.
- Two approaches can be used to reduce delay in adders:
  - 1) Use the fastest possible electronic-technology in implementing the ripple-carry design.
  - 2) Use an augmented logic-gate network structure.

## CARRY-LOOKAHEAD ADDITIONS

- The logic expression for  $s_i$  (sum) and  $c_{i+1}$  (carry-out) of stage  $i$  are

$$s_i = x_i + y_i + c_i \quad \text{----- (1)}$$

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i \quad \text{----- (2)}$$

- Factoring (2) into

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

we can write

$$c_{i+1} = G_i + P_i c_i \quad \text{where } G_i = x_i y_i \text{ and } P_i = x_i + y_i$$

- The expressions  $G_i$  and  $P_i$  are called generate and propagate functions (Figure 9.4).
- If  $G_i = 1$ , then  $c_{i+1} = 1$ , independent of the input carry  $c_i$ . This occurs when both  $x_i = 1$  or  $y_i = 1$ . Propagate function means that an input-carry will produce an output-carry when either  $x_i = 1$  or  $y_i = 1$ .
- All  $G_i$  and  $P_i$  functions can be formed independently and in parallel in one logic-gate delay.
- Expanding  $c_i$  terms of  $i-1$  subscripted variables and substituting into the  $c_{i+1}$  expression, we obtain
 
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \dots + P_i G_0 + P_i P_{i-1} \dots P_0 c_0$$
- Conclusion: Delay through the adder is 3 gate delays for all carry-bits & 4 gate delays for all sum-bits.
- Consider the design of a 4-bit adder. The carries can be implemented as
 
$$c_1 = G_0 + P_0 c_0$$

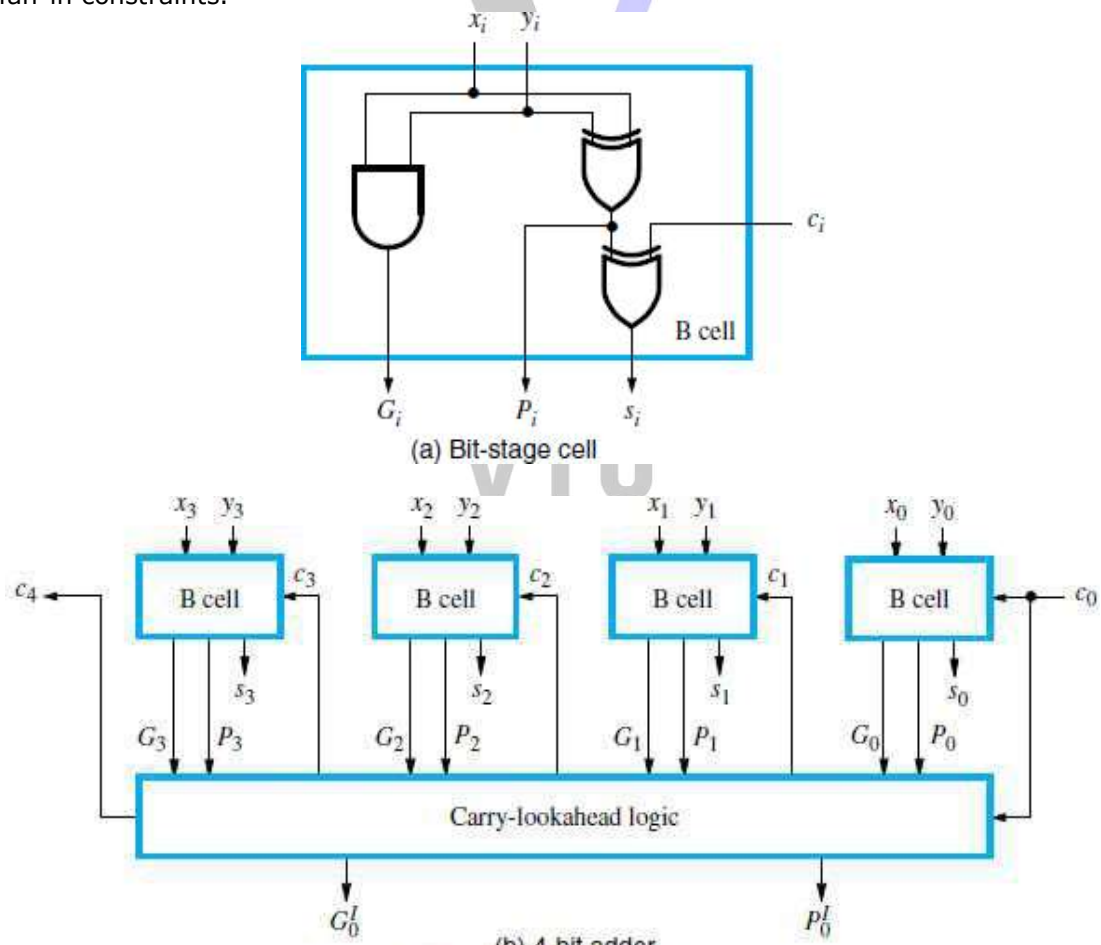
$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

The carries are implemented in the block labeled carry-lookahead logic. An adder implemented in this form is called a **Carry-Lookahead Adder**.

- Limitation: If we try to extend the carry-lookahead adder for longer operands, we run into a problem of gate fan-in constraints.



**Figure 9.4** A 4-bit carry-lookahead adder.

### HIGHER-LEVEL GENERATE & PROPAGATE FUNCTIONS

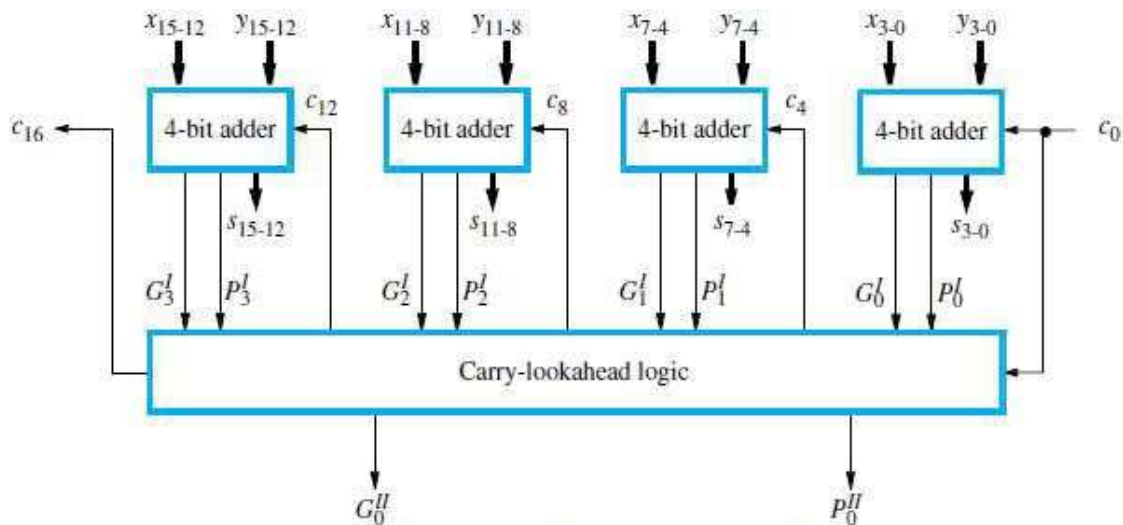
- 16-bit adder can be built from four 4-bit adder blocks (Figure 9.5).
- These blocks provide new output functions defined as  $G_k$  and  $P_k$ , where  $k=0$  for the first 4-bit block,  $k=1$  for the second 4-bit block and so on.
- In the first block,  

$$P_0 = P_3 P_2 P_1 P_0$$

$$\&$$

$$G_0 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$
- The first-level  $G_i$  and  $P_i$  functions determine whether bit stage  $i$  generates or propagates a carry, and the second level  $G_k$  and  $P_k$  functions determine whether block  $k$  generates or propagates a carry.
- Carry  $c_{16}$  is formed by one of the carry-lookahead circuits as  

$$c_{16} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$
- Conclusion: All carries are available 5 gate delays after  $X$ ,  $Y$  and  $c_0$  are applied as inputs.



**Figure 9.5** A 16-bit carry-lookahead adder built from 4-bit adders (see Figure 9.4b).

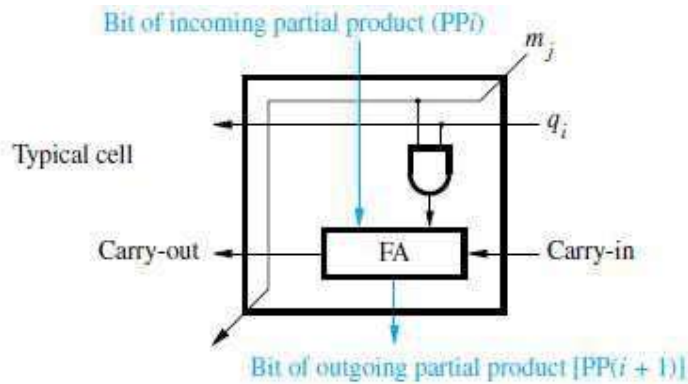
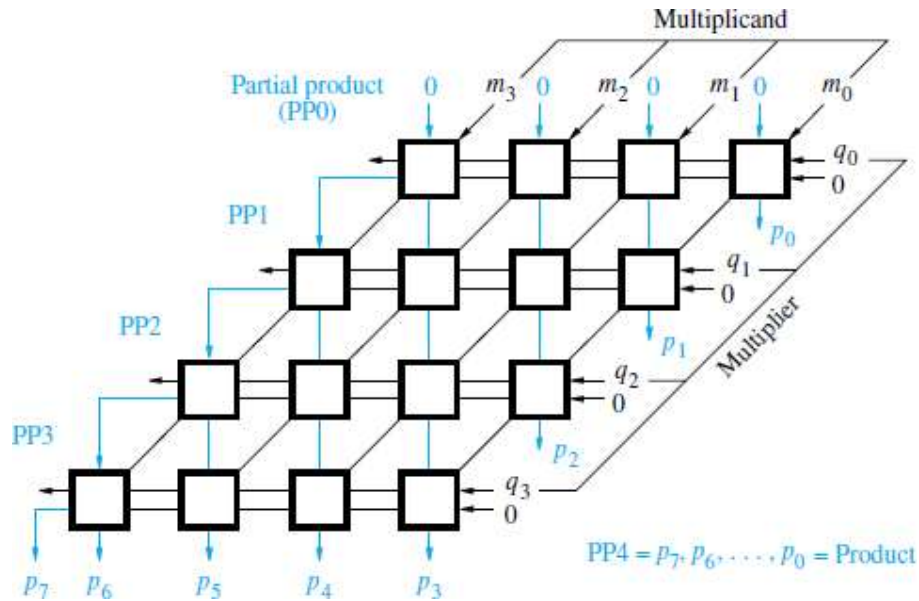
VTU  
Updates

## MULTIPLICATION OF POSITIVE NUMBERS

$$\begin{array}{r}
 \begin{array}{cccc}
 & 1 & 1 & 0 & 1 \\
 \times & 1 & 0 & 1 & 1 \\
 \hline
 & 1 & 1 & 0 & 1 \\
 & 0 & 0 & 0 & 0 \\
 & 1 & 1 & 0 & 1 \\
 \hline
 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1
 \end{array}
 \end{array}$$

(13) Multiplicand M  
(11) Multiplier Q  
(143) Product P

(a) Manual multiplication algorithm



(b) Array implementation

**Figure 9.6** Array multiplication of unsigned binary operands.

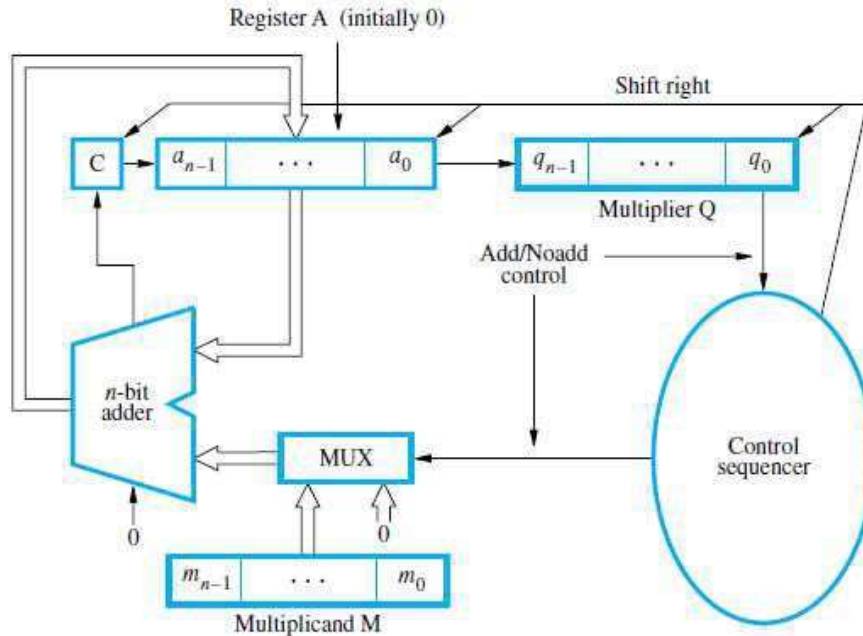
## ARRAY MULTIPLICATION

- The main component in each cell is a full adder (FA).
- The AND gate in each cell determines whether a multiplicand bit  $m_j$  is added to the incoming partial-product bit, based on the value of the multiplier bit  $q_i$  (Figure 9.6).

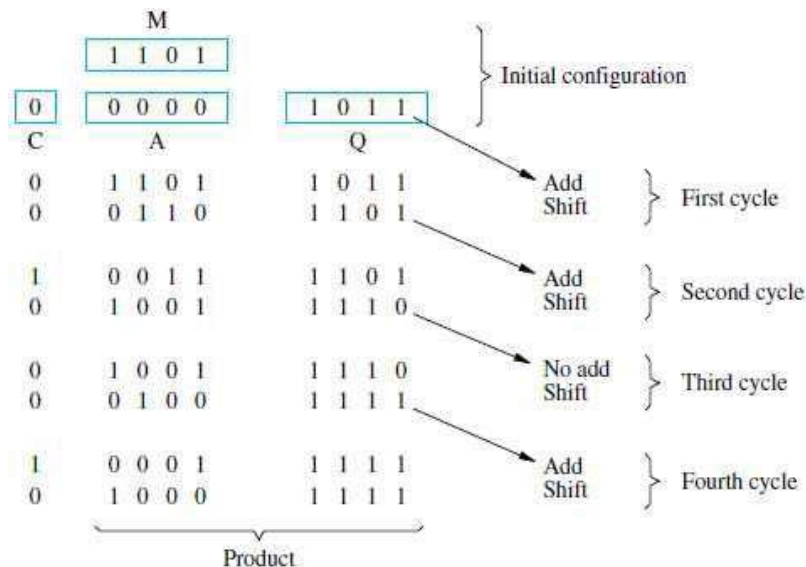


## SEQUENTIAL CIRCUIT BINARY MULTIPLIER

- Registers A and Q combined hold PP (partial product) while the multiplier bit  $q_i$  generates the signal Add/Noadd.
- The carry-out from the adder is stored in flip-flop C (Figure 9.7).
- Procedure for multiplication:
  - Multiplier is loaded into register Q, Multiplicand is loaded into register M and C & A are cleared to 0.
  - If  $q_0=1$ , add M to A and store sum in A. Then C, A and Q are shifted right one bit-position. If  $q_0=0$ , no addition performed and C, A & Q are shifted right one bit-position.
  - After n cycles, the high-order half of the product is held in register A and the low-order half is held in register Q.



(a) Register configuration



(b) Multiplication example

**Figure 9.7** Sequential circuit binary multiplier.

## SIGNED OPERAND MULTIPLICATION

### BOOTH ALGORITHM

- This algorithm
  - generates a  $2n$ -bit product
  - treats both positive & negative 2's-complement  $n$ -bit operands uniformly (Figure 9.9-9.12).
- Attractive feature: This algorithm achieves some efficiency in the number of addition required when the multiplier has a few large blocks of 1s.
- This algorithm suggests that we can reduce the number of operations required for multiplication by representing multiplier as a difference between 2 numbers.

For e.g. multiplier(Q) 14(001110) can be represented as

010000 (16)

-000010 (2)

001110 (14)

- Therefore, product  $P=M*Q$  can be computed by adding  $2^4$  times the M to the 2's complement of  $2^1$  times the M.

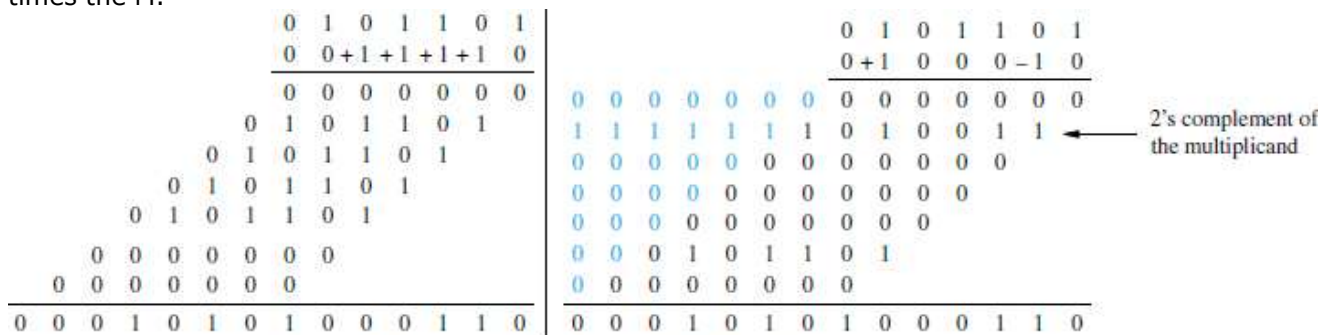


Figure 9.9 Normal and Booth multiplication schemes.

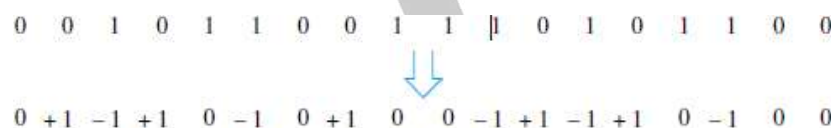


Figure 9.10 Booth recoding of a multiplier.

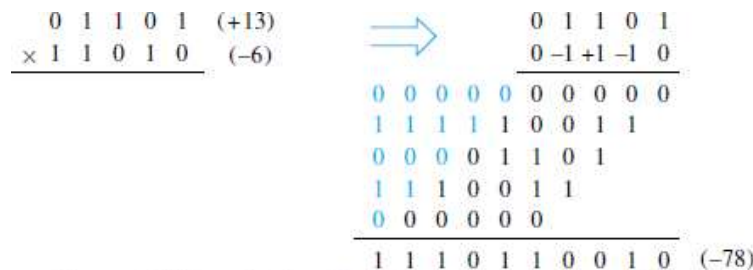


Figure 9.11 Booth multiplication with a negative multiplier.

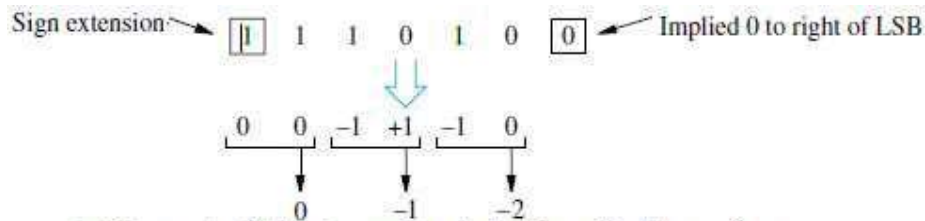
Multiplier		Version of multiplicand selected by bit $i$
Bit $i$	Bit $i-1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Figure 9.12 Booth multiplier recoding table.

## FAST MULTIPLICATION

### BIT-PAIR RECODING OF MULTIPLIERS

- This method
  - derived from the booth algorithm
  - reduces the number of summands by a factor of 2
- Group the Booth-recoded multiplier bits in pairs. (Figure 9.14 & 9.15).
- The pair (+1 -1) is equivalent to the pair (0 +1).



(a) Example of bit-pair recoding derived from Booth recoding

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position $i$
$i+1$	$i$	$i-1$	
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

(b) Table of multiplicand selection decisions

**Figure 9.14** Multiplier bit-pair recoding.

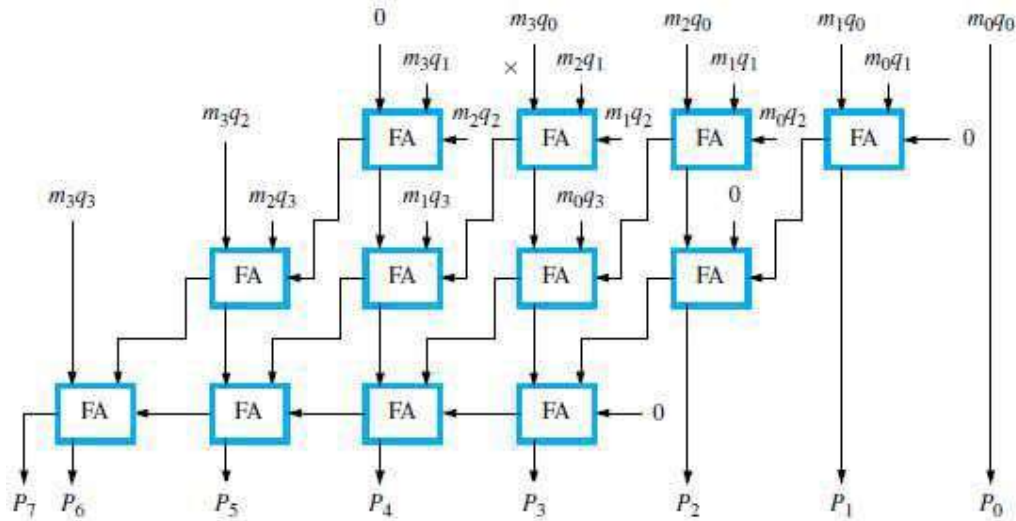
$$\begin{array}{r}
 01101 \quad (+13) \\
 \times 11010 \quad (-6) \\
 \hline
 \end{array}$$
$$\begin{array}{r}
 01101 \\
 0-1+1-10 \\
 \hline
 00000 \quad 000000 \\
 11111 \quad 10011 \\
 00001 \quad 1101 \\
 11100 \quad 11 \\
 00000 \quad 0 \\
 \hline
 1110110010 \quad (-78)
 \end{array}$$

$$\begin{array}{r}
 01101 \\
 0-1-2 \\
 \hline
 11111 \quad 100110 \\
 11111 \quad 0011 \\
 00000 \quad 0 \\
 \hline
 1110110010
 \end{array}$$

**Figure 9.15** Multiplication requiring only  $n/2$  summands.

### CARRY-SAVE ADDITION OF SUMMANDS

- Consider the array for 4\*4 multiplication. (Figure 9.16 & 9.18).
- Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions.



(b) Carry-save array

Figure 9.16 carry-save arrays for a 4 × 4 multiplier.

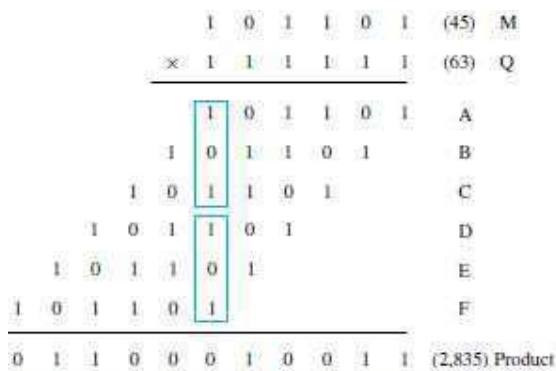


Figure 9.17 A multiplication example used to illustrate carry-save addition as shown in Figure 9.18.

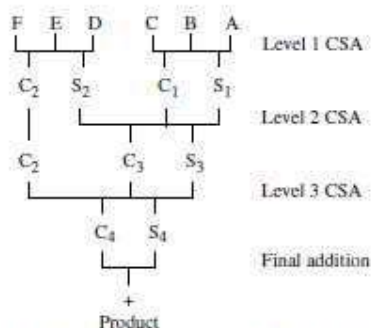


Figure 9.19 Schematic representation of the carry-save addition operations in Figure 9.18.

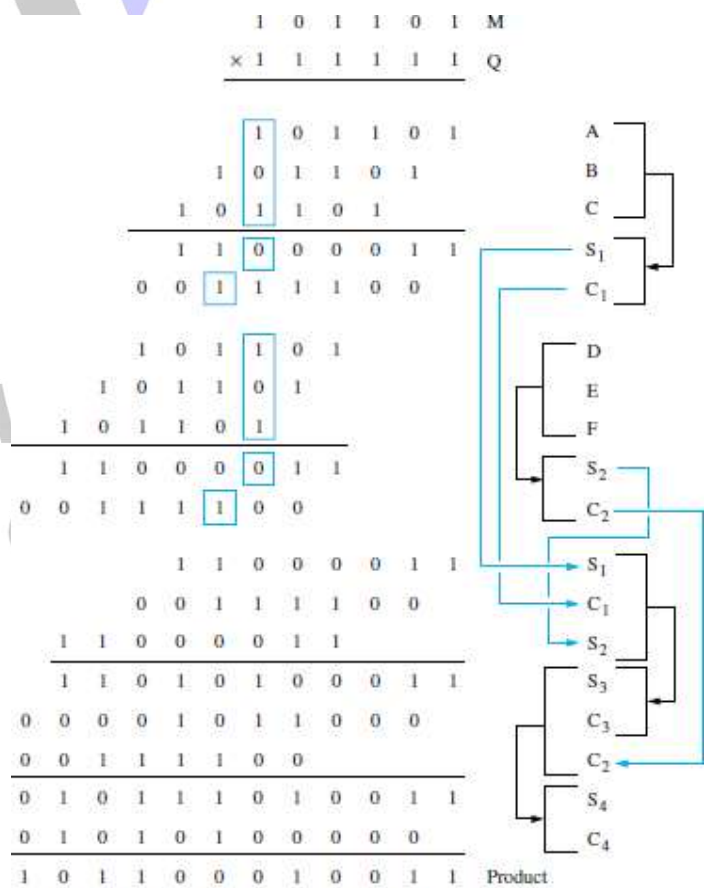


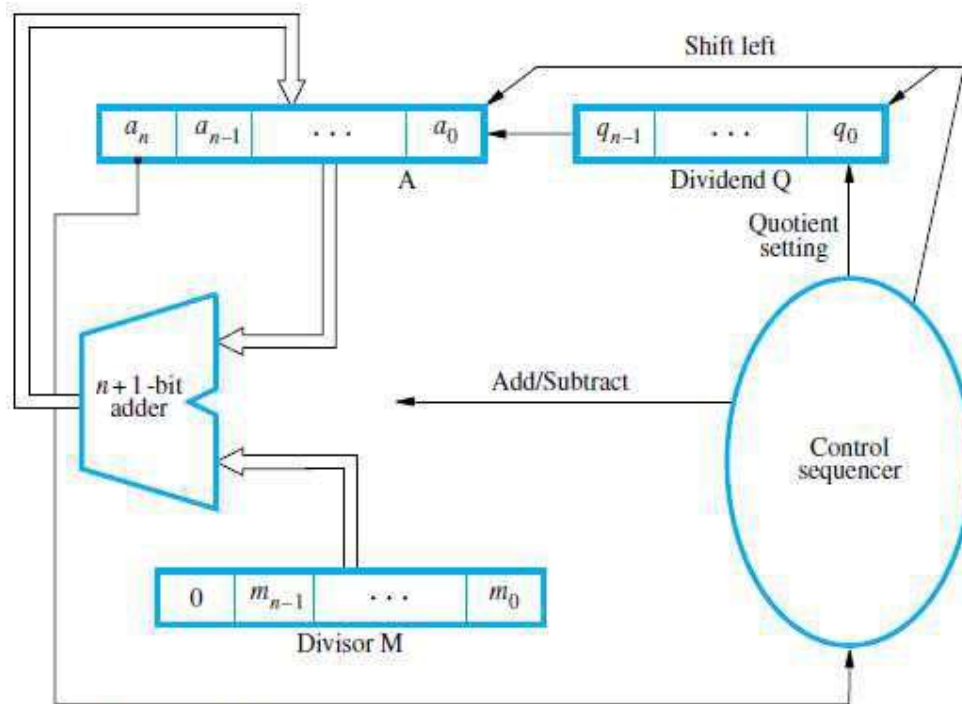
Figure 9.18 The multiplication example from Figure 9.17 performed using carry-save addition.

- The full adder is input with three partial bit products in the first row.
- Multiplication requires the addition of several summands.
- CSA speeds up the addition process.
- Consider the array for 4x4 multiplication shown in fig 9.16.
- First row consisting of just the AND gates that implement the bit products  $m_3q_0$ ,  $m_2q_0$ ,  $m_1q_0$  and  $m_0q_0$ .
- The delay through the carry-save array is somewhat less than delay through the ripple-carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.
- Consider the addition of many summands in fig 9.18.
- Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
- Group all of the S and C vectors into threes, and perform carry-save addition on them, generating a further set of S and C vectors in one more full-adder delay
- Continue with this process until there are only two vectors remaining
- They can be added in a RCA or CLA to produce the desired product.
- When the number of summands is large, the time saved is proportionally much greater.
- Delay: AND gate + 2 gate/CSA level + CLA gate delay, Eg., 6 bit number require 15 gate delay, array 6x6 require  $6(n-1)-1 = 29$  gate Delay.
- In general, CSA takes  $1.7 \log_2 k - 1.7$  levels of CSA to reduce k summands.

VTU  
Updates

## INTEGER DIVISION

- An  $n$ -bit positive-divisor is loaded into register M.  
An  $n$ -bit positive-dividend is loaded into register Q at the start of the operation.  
Register A is set to 0 (Figure 9.21).
- After division operation, the  $n$ -bit quotient is in register Q, and the remainder is in register A.



**Figure 9.23** Circuit arrangement for binary division.

	21
13	) 274
	<u>26</u>
	14
	<u>13</u>
	1

	10101
1101	) 100010010
	<u>1101</u>
	10000
	<u>1101</u>
	1110
	<u>1101</u>
	1

**Figure 9.22** Longhand division examples.

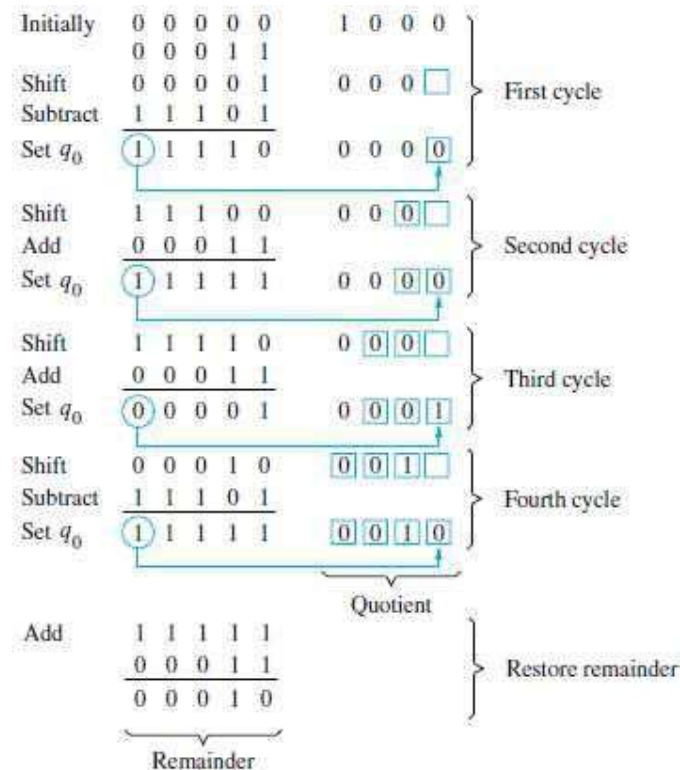
## NON-RESTORING DIVISION

- Procedure:

Step 1: Do the following n times

- If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A (Figure 9.23).
- Now, if the sign of A is 0, set  $q_0$  to 1; otherwise set  $q_0$  to 0.

Step 2: If the sign of A is 1, add M to A (restore).



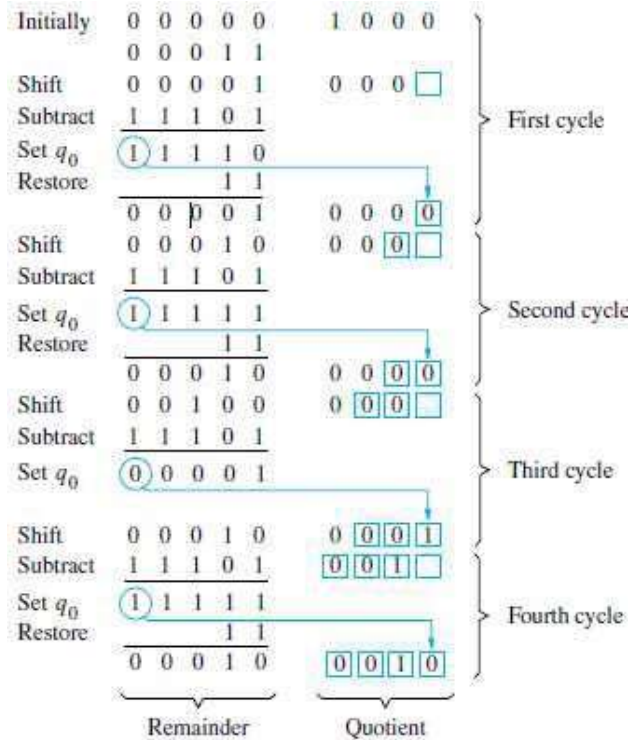
**Figure 9.25** A non-restoring division example.

VTU  
Updates



## RESTORING DIVISION

- Procedure: Do the following n times
  - 1) Shift A and Q left one binary position (Figure 9.22).
  - 2) Subtract M from A, and place the answer back in A
  - 3) If the sign of A is 1, set  $q_0$  to 0 and add M back to A (restore A).  
If the sign of A is 0, set  $q_0$  to 1 and no restoring done.



**Figure 9.24** A restoring division example.

VTU  
Updates



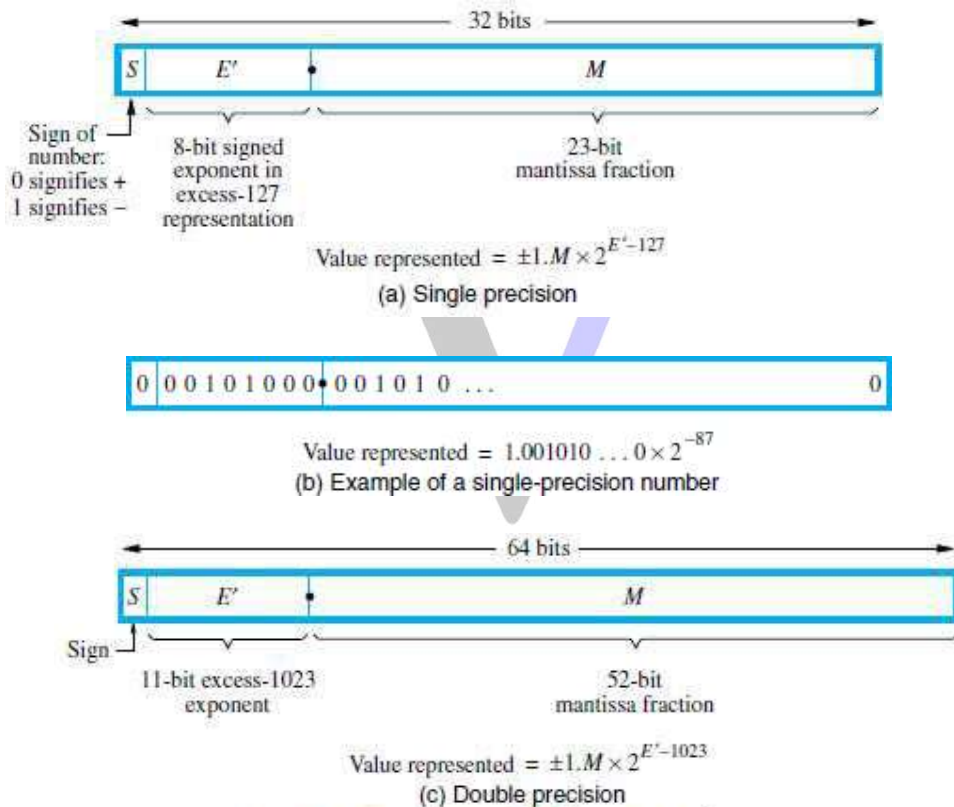
## FLOATING-POINT NUMBERS & OPERATIONS

### IEEE STANDARD FOR FLOATING POINT NUMBERS

- Single precision representation occupies a single 32-bit word.  
The scale factor has a range of  $2^{-126}$  to  $2^{+127}$  (which is approximately equal to  $10^{+38}$ ).
- The 32 bit word is divided into 3 fields: sign(1 bit), exponent(8 bits) and mantissa(23 bits).
- Signed exponent=E.

Unsigned exponent  $E' = E + 127$ . Thus,  $E'$  is in the range  $0 < E' < 255$ .

- The last 23 bits represent the mantissa. Since binary normalization is used, the MSB of the mantissa is always equal to 1. (M represents fractional-part).
- The 24-bit mantissa provides a precision equivalent to about 7 decimal-digits (Figure 9.24).
- Double precision representation occupies a single 64-bit word. And  $E'$  is in the range  $1 < E' < 2046$ .
- The 53-bit mantissa provides a precision equivalent to about 16 decimal-digits.

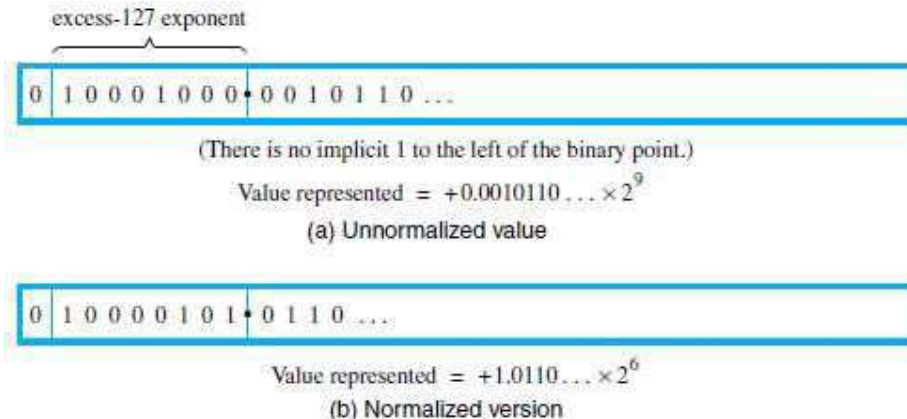


**Figure 9.26** IEEE standard floating-point formats.

# Updates

## NORMALIZATION

- When the decimal point is placed to the right of the first(non zero) significant digit, the number is said to be normalized.
- If a number is not normalized, it can always be put in normalized form by shifting the fraction and adjusting the exponent. As computations proceed, a number that does not fall in the representable range of normal numbers might be generated.
- In single precision, it requires an exponent less than -126 (underflow) or greater than +127 (overflow). Both are exceptions that need to be considered.



**Figure 9.27** Floating-point normalization in IEEE single-precision format.

## SPECIAL VALUES

- The end values 0 and 255 of the excess-127 exponent  $E'$  are used to represent special values.
- When  $E'=0$  and the mantissa fraction  $m$  is zero, the value exact 0 is represented.
- When  $E'=255$  and  $M=0$ , the value  $\infty$  is represented, where  $\infty$  is the result of dividing a normal number by zero.
- when  $E'=0$  and  $M \neq 0$ , denormal numbers are represented. Their value is  $\pm 0.M \times 2^{-126}$ .
- When  $E'=255$  and  $M \neq 0$ , the value represented is called not a number (NaN). A NaN is the result of performing an invalid operation such as  $0/0$  or  $\sqrt{0}$ .

## ARITHMETIC OPERATIONS ON FLOATING-POINT NUMBERS

### Multiply Rule

- 1) Add the exponents & subtract 127.
- 2) Multiply the mantissas & determine sign of the result.
- 3) Normalize the resulting value if necessary.

### Divide Rule

- 1) Subtract the exponents & add 127.
- 2) Divide the mantissas & determine sign of the result.
- 3) Normalize the resulting value if necessary.

### Add/Subtract Rule

- 1) Choose the number with the smaller exponent & shift its mantissa right a number of steps equal to the difference in exponents ( $n$ ).
- 2) Set exponent of the result equal to larger exponent.
- 3) Perform addition/subtraction on the mantissas & determine sign of the result.
- 4) Normalize the resulting value if necessary.

## IMPLEMENTING FLOATING-POINT OPERATIONS

- First compare exponents to determine how far to shift the mantissa of the number with the smaller exponent.
- The shift-count value  $n$ 
  - is determined by 8 bit subtractor &
  - is sent to SHIFTER unit.
- In step 1, sign is sent to SWAP network (Figure 9.26).
  - If  $\text{sign}=0$ , then  $E_A > E_B$  and mantissas  $M_A$  &  $M_B$  are sent straight through SWAP network.
  - If  $\text{sign}=1$ , then  $E_A < E_B$  and the mantissas are swapped before they are sent to SHIFTER.
- In step 2, 2:1 MUX is used. The exponent of result  $E$  is tentatively determined as  $E_A$  if  $E_A > E_B$  or  $E_B$  if  $E_A < E_B$
- In step 3, CONTROL logic
  - determines whether mantissas are to be added or subtracted.
  - determines sign of the result.
- In step 4, result of step 3 is normalized. The number of leading zeros in  $M$  determines number of bit shifts( $X$ ) to be applied to  $M$ .

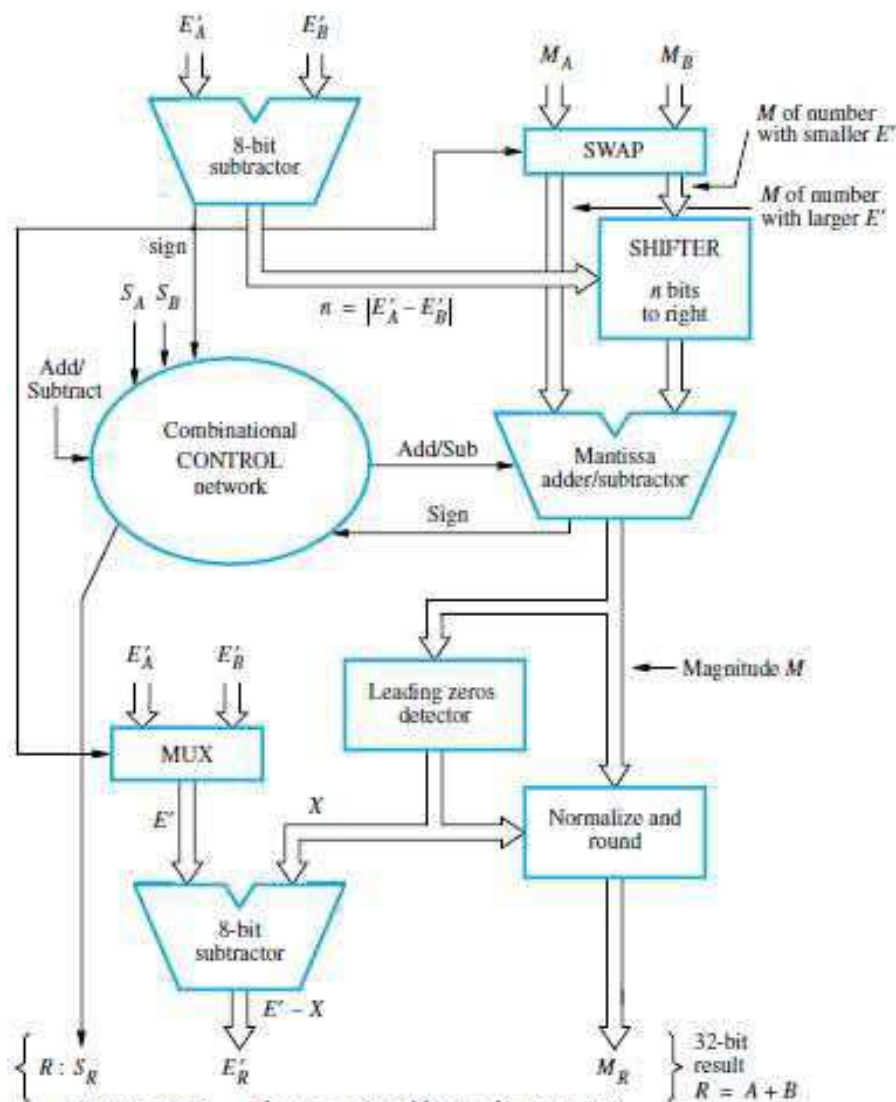


Figure 9.28 Floating-point addition-subtraction unit.

**Problem 1:**

Represent the decimal values 5, -2, 14, -10, 26, -19, 51 and -43 as signed 7-bit numbers in the following binary formats:

- (a) sign-and-magnitude
- (b) 1's-complement
- (c) 2's-complement

**Solution:**

The three binary representations are given as:

Decimal values	Sign-and-magnitude representation	1's-complement representation	2's-complement representation
5	0000101	0000101	0000101
-2	1000010	1111101	1111110
14	0001110	0001110	0001110
-10	1001010	1110101	1110110
26	0011010	0011010	0011010
-19	1010011	1101100	1101101
51	0110011	0110011	0110011
-43	1101011	1010100	1010101

**Problem 2:**

(a) Convert the following pairs of decimal numbers to 5-bit 2's-complement numbers, then add them. State whether or not overflow occurs in each case.

- a) 5 and 10                      b) 7 and 13
- c) -14 and 11                  d) -5 and 7
- e) -3 and -8

(b) Repeat Problem 1.7 for the subtract operation, where the second number of each pair is to be subtracted from the first number. State whether or not overflow occurs in each case.

**Solution:**

(a)

$$\begin{array}{rcl}
 (a) & \begin{array}{r} 00101 \\ + 01010 \\ \hline 01111 \\ \text{no overflow} \end{array} & (b) \begin{array}{r} 00111 \\ + 01101 \\ \hline 10100 \\ \text{overflow} \end{array} \quad (c) \begin{array}{r} 10010 \\ + 01011 \\ \hline 11101 \\ \text{no overflow} \end{array}
 \end{array}$$

$$\begin{array}{rcl}
 (d) & \begin{array}{r} 11011 \\ + 00111 \\ \hline 00010 \\ \text{no overflow} \end{array} & (e) \begin{array}{r} 11101 \\ + 11000 \\ \hline 10101 \\ \text{no overflow} \end{array} \quad (f) \begin{array}{r} 10110 \\ + 10011 \\ \hline 01001 \\ \text{overflow} \end{array}
 \end{array}$$

(b) To subtract the second number, form its 2's-complement and add it to the first number.

$$\begin{array}{rcl}
 (a) & \begin{array}{r} 00101 \\ + 10110 \\ \hline 11011 \\ \text{no overflow} \end{array} & (b) \begin{array}{r} 00111 \\ + 10011 \\ \hline 11010 \\ \text{no overflow} \end{array} \quad (c) \begin{array}{r} 10010 \\ + 10101 \\ \hline 00111 \\ \text{overflow} \end{array} \\
 (d) & \begin{array}{r} 11011 \\ + 11001 \\ \hline 10100 \\ \text{no overflow} \end{array} & (e) \begin{array}{r} 11101 \\ + 01000 \\ \hline 00101 \\ \text{no overflow} \end{array} \quad (f) \begin{array}{r} 10110 \\ + 01101 \\ \hline 00011 \\ \text{no overflow} \end{array}
 \end{array}$$

**Problem 3:**

Perform following operations on the 6-bit signed numbers using 2's complement representation system. Also indicate whether overflow has occurred.

010110	101011	111111
<u>+001001</u>	<u>+100101</u>	<u>+000111</u>
011001	110111	010101
<u>+010000</u>	<u>+111001</u>	<u>+101011</u>
010110	111110	100001
<u>-011111</u>	<u>-100101</u>	<u>-011101</u>
111111	000111	011010
<u>-000111</u>	<u>-111000</u>	<u>-100010</u>

**Solution:**

$\begin{array}{r} 010110 \\ + 001001 \\ \hline 011111 \end{array}$	$\begin{array}{r} (+22) \\ + (+9) \\ \hline (+31) \end{array}$	$\begin{array}{r} 101011 \\ + 100101 \\ \hline 010000 \\ \text{overflow} \end{array}$	$\begin{array}{r} (-21) \\ + (-27) \\ \hline (-48) \end{array}$	$\begin{array}{r} 111111 \\ + 000111 \\ \hline 000110 \end{array}$	$\begin{array}{r} (-1) \\ + (+7) \\ \hline (+6) \end{array}$
$\begin{array}{r} 011001 \\ + 010000 \\ \hline 101001 \\ \text{overflow} \end{array}$	$\begin{array}{r} (+25) \\ + (+16) \\ \hline (+41) \end{array}$	$\begin{array}{r} 110111 \\ + 111001 \\ \hline 110000 \end{array}$	$\begin{array}{r} (-9) \\ + (-7) \\ \hline (-16) \end{array}$	$\begin{array}{r} 010101 \\ + 101011 \\ \hline 000000 \end{array}$	$\begin{array}{r} (+21) \\ + (-21) \\ \hline (0) \end{array}$
$\begin{array}{r} 010110 \\ - 011111 \\ \hline \end{array}$	$\begin{array}{r} (+22) \\ - (+31) \\ \hline (-9) \end{array}$	$\begin{array}{r} 010110 \\ + 100001 \\ \hline 110111 \end{array}$			
$\begin{array}{r} 111110 \\ - 100101 \\ \hline \end{array}$	$\begin{array}{r} (-2) \\ - (-27) \\ \hline (+25) \end{array}$	$\begin{array}{r} 111110 \\ + 011011 \\ \hline 011001 \end{array}$			
$\begin{array}{r} 100001 \\ - 011101 \\ \hline \end{array}$	$\begin{array}{r} (-31) \\ - (+29) \\ \hline (-60) \end{array}$	$\begin{array}{r} 100001 \\ + 100011 \\ \hline 000100 \\ \text{overflow} \end{array}$			
$\begin{array}{r} 111111 \\ - 000111 \\ \hline \end{array}$	$\begin{array}{r} (-1) \\ - (+7) \\ \hline (-8) \end{array}$	$\begin{array}{r} 111111 \\ + 111001 \\ \hline 111000 \end{array}$			
$\begin{array}{r} 000111 \\ - 111000 \\ \hline \end{array}$	$\begin{array}{r} (+7) \\ - (-8) \\ \hline (+15) \end{array}$	$\begin{array}{r} 000111 \\ + 001000 \\ \hline 001111 \end{array}$			
$\begin{array}{r} 011010 \\ - 100010 \\ \hline \end{array}$	$\begin{array}{r} (+26) \\ - (-30) \\ \hline (+56) \end{array}$	$\begin{array}{r} 011010 \\ + 011110 \\ \hline 111000 \\ \text{overflow} \end{array}$			

#### Problem 4:

Perform signed multiplication of following 2's complement numbers using Booth's algorithm.

- (a) A=010111 and B=110110 (b) A=110011 and B=101100  
 (c) A=110101 and B=011011 (d) A=001111 and B=001111  
 (e) A=10100 and B=10101 (f) A=01110 and B=11000

**Solution:**

$$\begin{array}{r}
 010111 \\
 \times 110110 \\
 \hline
 \end{array}
 \quad
 \begin{array}{r}
 +23 \\
 \times -10 \\
 \hline
 -230
 \end{array}$$

sign extension

$$\begin{array}{r}
 010111 \\
 \times 0-1+10-10 \\
 \hline
 0 \\
 111111010001 \\
 000001011110 \\
 11111001011 \\
 \hline
 111100011010
 \end{array}$$

$$\begin{array}{r}
 110011 \\
 \times 101100 \\
 \hline
 \end{array}
 \quad
 \begin{array}{r}
 -13 \\
 \times -20 \\
 \hline
 260
 \end{array}$$

sign extension

$$\begin{array}{r}
 110011 \\
 \times -1+10-100 \\
 \hline
 0 \\
 00000001101 \\
 1111100110 \\
 010111011 \\
 \hline
 000100000100
 \end{array}$$

$$\begin{array}{r}
 110101 \\
 \times 011011 \\
 \hline
 \end{array}
 \quad
 \begin{array}{r}
 -11 \\
 \times 27 \\
 \hline
 -297
 \end{array}$$

sign extension

$$\begin{array}{r}
 110101 \\
 \times +10-1+10-1 \\
 \hline
 0 \\
 00000001011 \\
 1111110101 \\
 000001011 \\
 1111011011 \\
 \hline
 111011010111
 \end{array}$$

$$\begin{array}{r}
 001111 \\
 \times 001111 \\
 \hline
 \end{array}
 \quad
 \begin{array}{r}
 15 \\
 \times 15 \\
 \hline
 225
 \end{array}$$

$$\begin{array}{r}
 001111 \\
 \times 0+1000-1 \\
 \hline
 111111110001 \\
 00001111 \\
 \hline
 000011100001
 \end{array}$$

$$\begin{array}{r}
 10100(-12) \\
 \times 10101(-11) \\
 \hline
 \end{array}
 \rightarrow
 \begin{array}{r}
 10100 \\
 -11-11-1-1 \text{ (recoded multiplier)} \\
 \hline
 000001100 \\
 11110100 \\
 0001100 \\
 110100 \\
 01100 \\
 \hline
 010000100 \text{ (+132)}
 \end{array}$$

$$\begin{array}{r}
 01110(+14) \\
 \times 11000(-8) \\
 \hline
 \end{array}
 \rightarrow
 \begin{array}{r}
 01110 \\
 0-1000 \text{ (recoded multiplier)} \\
 \hline
 000000000 \\
 000000000 \\
 000000000 \\
 1110010 \\
 000000 \\
 \hline
 1110010000 \text{ (-112)}
 \end{array}$$

**Problem 5:**

Perform signed multiplication of following 2's complement numbers using bit-pair recoding method.

(a) A=010111 and B=110110

(b) A=110011 and B=101100

(c) A=110101 and B=011011

(d) A=001111 and B=001111

**Solution:**

$$\begin{array}{r} 010111 \\ \times 110110 \\ \hline \end{array}$$

$$\begin{array}{r} \phantom{010111} 010111 \\ \phantom{010111} -1 \phantom{00} +2 \phantom{00} -2 \\ \hline 111111 \phantom{00} 1010010 \\ 0000 \phantom{00} 0101110 \\ \hline 11110210101 \\ \hline 111100011010 \end{array}$$

$$\begin{array}{r} 110011 \\ \times 101100 \\ \hline \end{array}$$

$$\begin{array}{r} \phantom{110011} 110011 \\ \phantom{110011} -1 \phantom{00} -1 \phantom{00} 0 \\ \hline 0000 \phantom{00} 0001101 \\ 0000 \phantom{00} 0111011 \\ \hline 000100000100 \end{array}$$

$$\begin{array}{r} 110101 \\ \times 011011 \\ \hline \end{array}$$

$$\begin{array}{r} \phantom{110101} 110101 \\ \phantom{110101} +2 \phantom{00} -1 \phantom{00} -1 \\ \hline 000000 \phantom{00} 0001011 \\ 0000 \phantom{00} 0001011 \\ \hline 11101011 \\ \hline 111011010111 \end{array}$$

$$\begin{array}{r} 001111 \\ \times 001111 \\ \hline \end{array}$$

$$\begin{array}{r} \phantom{001111} 001111 \\ \phantom{001111} +1 \phantom{00} -1 \\ \hline 111111 \phantom{00} 110001 \\ 00001111 \\ \hline 000011100001 \end{array}$$

**VTU**  
**Updates**



### Problem 6:

Given A=10101 and B=00100, perform A/B using restoring division algorithm.

**Solution:**

Initially	0 0 0 0 0 0 (A)	1 0 1 0 1 (Q)
	0 0 0 1 0 0 (M)	
Shift	0 0 0 0 0 1	0 1 0 1 □
Subtract	1 1 1 1 0 0	
Set q0	1 1 1 1 0 1	
Restore	1 0 0	
	0 0 0 0 0 1	0 1 0 1 0
Shift	0 0 0 0 1 0	1 0 1 0
Subtract	1 1 1 1 0 0	
Set q0	1 1 1 1 1 0	
Restore	1 0 0	
	0 0 0 0 1 0	1 0 1 0 0
Shift	0 0 0 1 0 1	0 1 0 0
Subtract	1 1 1 1 0 0	
Set q0	0 0 0 0 0 1	
No restore	0 0 0 0 0 0	
	0 0 0 0 0 1	1 0 1 0 0
Shift	0 0 0 0 0 1	0 1 0 0 1
Subtract	1 1 1 1 0 0	
Set q0	1 1 1 1 1 0	
Restore	1 0 0	
	0 0 0 0 1 0	1 0 0 1 0
Shift	0 0 0 1 0 1	0 0 1 0
Subtract	1 1 1 1 0 0	
Set q0	0 0 0 0 0 1	
No restore	0 0 0 0 0 0	
	0 0 0 0 0 1	0 0 1 0 1
	remainder	quotient

### Problem 7:

Given A=10101 and B=00101, perform A/B using non-restoring division algorithm.

**Solution:**

	000000	10101	
	A	Q	
	000101		
	M		
			Initial configuration
shift	000001	0 1 0 1 □	
subtract	111011		1st cycle
	111100	0 1 0 1 0	
shift	111000	1 0 1 0 □	
add	000101		2nd cycle
	111101	1 0 1 0 0	
shift	111011	0 1 0 0 □	
add	000101		3rd cycle
	000000	0 1 0 0 1	
shift	000000	1 0 0 1 □	
subtract	111011		4th cycle
	111011	1 0 0 1 0	
shift	110111	0 0 1 0 □	
add	000101		5th cycle
	111100	0 0 1 0 0	
add	000101		
	000001		
		quotient	
	remainder		



**Problem 8:**

Represent 1259.12510 in single precision and double precision formats

**Solution:**

Step 1: Convert decimal number to binary format

$$1259_{(10)} = 10011101011_{(2)}$$

Fractional Part

$$0.125_{(10)} = 0.001$$

$$\begin{aligned} \text{Binary number} &= 10011101011 + 0.001 \\ &= 10011101011.001 \end{aligned}$$

Step 2: Normalize the number

$$10011101011.001 = 1.0011101011001 \times 2^{10}$$

Step 3: Single precision format:

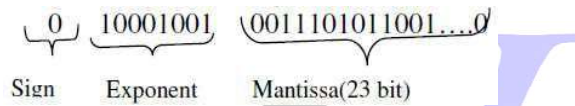
For a given number  $S=0$ ,  $E=10$  and  $M=0011101011001$

Bias for single precision format is = 127

$$E' = E + 127 = 10 + 127 = 137_{(10)}$$

$$= 10001001_{(2)}$$

Number in single precision format is given as



Step 4: Double precision format:

For a given number  $S=0$ ,  $E=10$  and  $M=0011101011001$

Bias for double precision format is = 1023

$$E' = E + 1023 = 10 + 1023 = 1033_{(10)}$$

$$= 10000001001_{(2)}$$

Number in double precision format is given as



VTU  
Updates