Module 4

NUMBERS, ARITHMETIC OPERATIONS AND CHARACTERS

Bit- binary digit

Binary number- string of bits representing number

Character code- string of bits representing a text character

1's- Complement: complement every bit. Change 0 to 1 and 1 to 0

3=0011, -3=1100 (equivalent to subtracting number from 2^n-1)

2's- Complement: add 1 to 1's complement of the number (same as subtracting number from 2^n)

Number representation

1101000101 (b_9b_8.....b_0) – total 10 bits (n-bits)

=> b_i

where 0<=i<=9 i.e, 0<=i<=n-1

Range- 0 to 2^{10}-1

i.e, 0 to from 2^n-1

MSB=1 for negative number, 0 for positive number

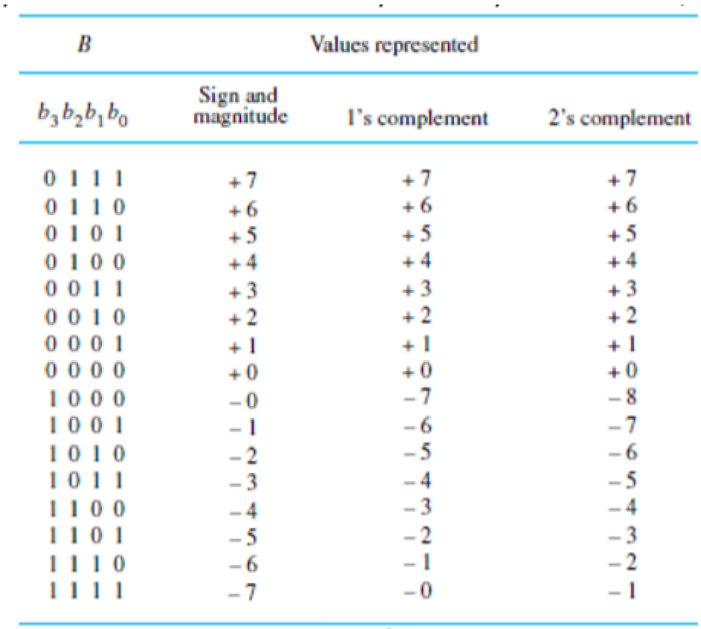
Sign and magnitude representation: 0101 (+5), 1101 (-5)

1's- Complement representation: for positive numbers, 1's complement representation is same as the original binary number represented in sign and magnitude form. For negative numbers, 1's- complement representation, find 1's complement of the number.

2's- Complement representation: for positive numbers, 2's complement representation is same as the original binary number represented in sign and magnitude form. For negative numbers, 2's- complement representation, find 2's complement of the number.

NOTE: Number representation in 1's or 2's complement representation is different from performing 1's and 2's complement. 1's and 2's complement representation may or may not require you to perform the 1's and 2's complement operation based on whether the number is positive or not. If

the number to be represented is positive, do not perform the operation, else perform the operation.



B	Values represented		
$b_3b_2b_1b_0$	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.

2's- complement has only one representation for 0

Addition of positive numbers

Start from right side (LSB)

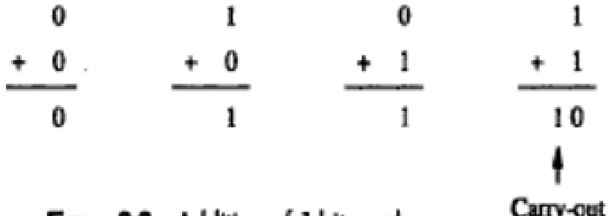


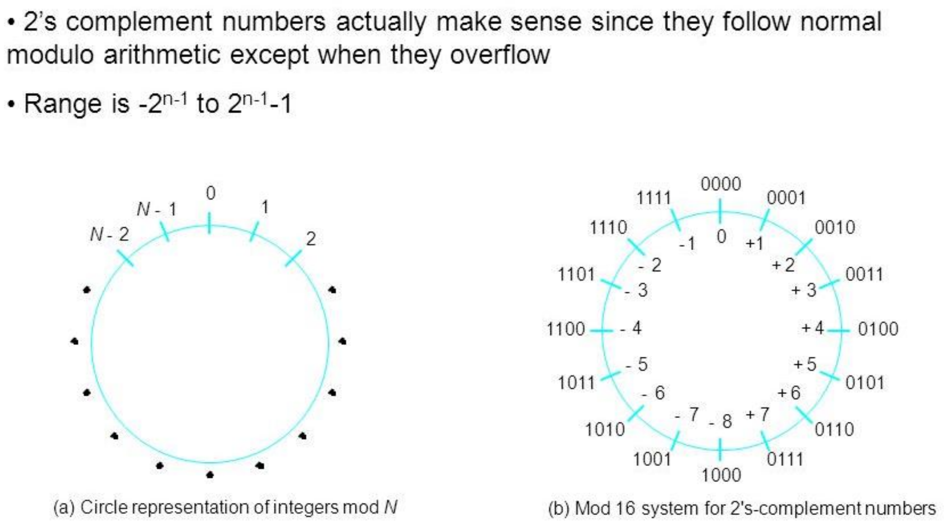
Figure 2.2 Addition of 1-bit numbers.

Addition and Subtraction of signed numbers

(a+b) mod 16

Eg. (7+4) mod 16 => locate 7 on circle and move 4 units in clockwise direction to arrive at answer 11.

2’s Complement



- 2’s complement numbers actually make sense since they follow normal modulo arithmetic except when they overflow
- Range is -2^{n-1} to $2^{n-1}-1$

(a) Circle representation of integers mod N

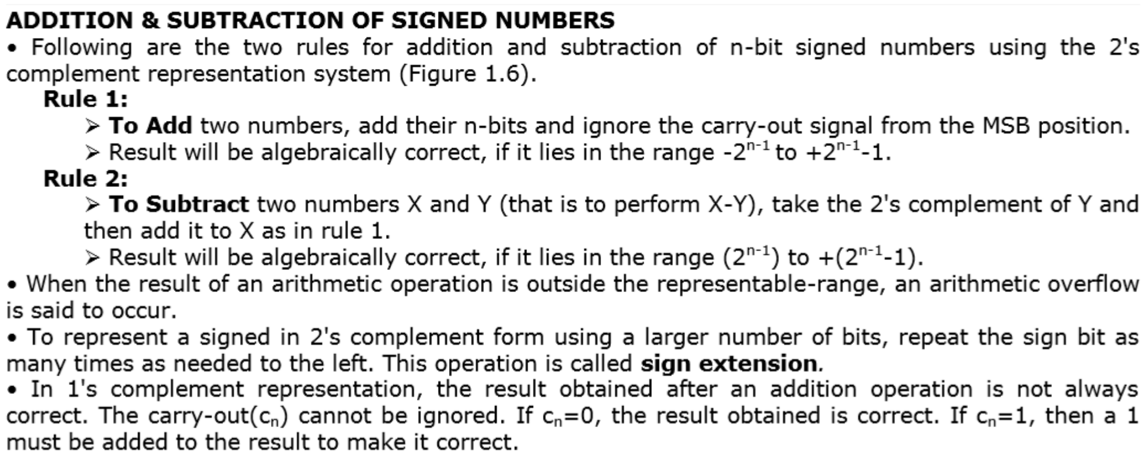
(b) Mod 16 system for 2's-complement numbers

To add +7 to -3:

2’s complement of +7=0111

2’s complement of -3=1101.

Locate 0111, then move 1101 (i.e, 13) steps in clockwise direction to arrive at 0100 (+4) result.



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.

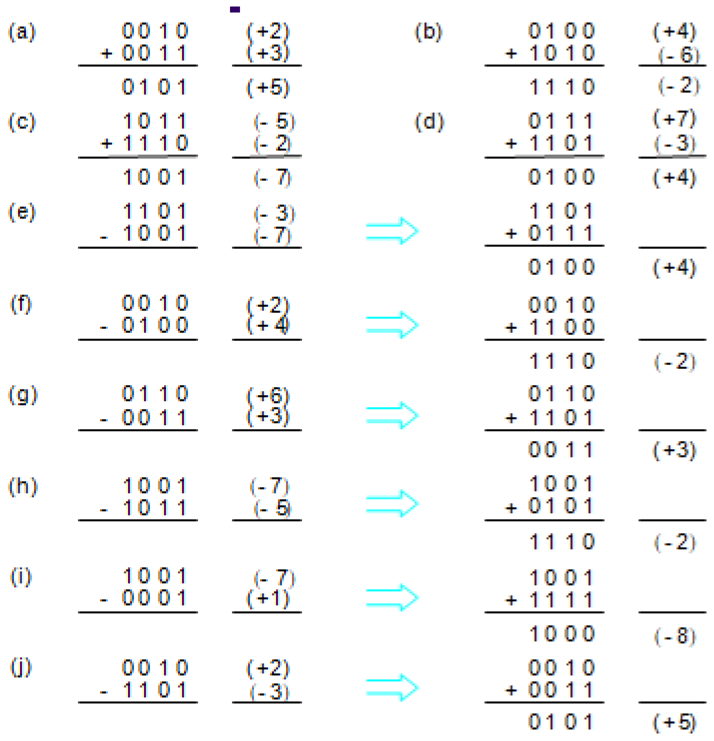
Remember, the operands must be represented in 2’s complement representation

. Only then the addition or subtraction should be performed. The answer calculated will be in the 2’s complement representation.

Assume operands are X and Y. These may be positive or negative numbers.

For addition (X+Y): Find 2’s complement representation of X (assume A), find 2’s complement representation of Y (assume B). Add A and B. Result is in 2’s complement representation.

For subtraction (X-Y): Find 2’s complement representation of X (assume A), find 2’s complement representation of Y (assume B). Find 2’s complement (not representation) of B (assume B’’) . Add A and B’’. Result is in 2’s complement representation.



```
(a)    0010    (+2)        (b)    0100    (+4)
     + 0011    (+3)             + 1010    (-6)
     ------    ----             ------    ----
       0101    (+5)               1110    (-2)

(c)    1011    (-5)        (d)    0111    (+7)
     + 1110    (-2)             + 1101    (-3)
     ------    ----             ------    ----
       1001    (-7)               0100    (+4)

(e)    1101    (-3)               1101
     - 1001    (-7)    ⇒        + 0111
     ------    ----             ------    ----
                                  0100    (+4)

(f)    0010    (+2)               0010
     - 0100    (+4)    ⇒        + 1100
     ------    ----             ------    ----
                                  1110    (-2)

(g)    0110    (+6)               0110
     - 0011    (+3)    ⇒        + 1101
     ------    ----             ------    ----
                                  0011    (+3)

(h)    1001    (-7)               1001
     - 1011    (-5)    ⇒        + 0101
     ------    ----             ------    ----
                                  1110    (-2)

(i)    1001    (-7)               1001
     - 0001    (+1)    ⇒        + 1111
     ------    ----             ------    ----
                                  1000    (-8)

(j)    0010    (+2)               0010
     - 1101    (-3)    ⇒        + 0011
     ------    ----             ------    ----
                                  0101    (+5)
```

Overflow:

Overflow occurs when result of arithmetic operation is outside the representable range.

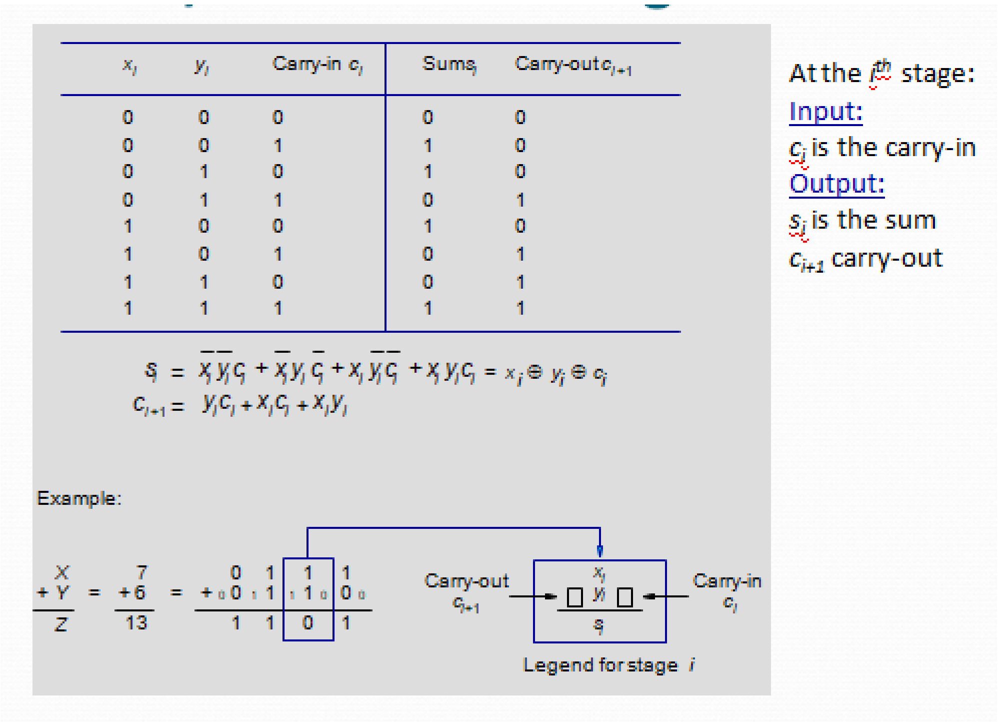
Overflow occurs when X, Y both have same sign but result has different sign.

```
            0111                          1000
   5        0101                 -7       1001
   3        0011                 -2       1100
  ---       ----                 ---      -----
  -8        1000                  7       10111

Overflow                        Overflow

            0000                          1111
   5        0101                 -3       1101
   2        0010                 -5       1011
  ---       ----                 ---      -----
   7        0111                 -8       11000

No overflow                     No overflow
```

Characters:

Represented as ASCII codes

ADDITION AND SUBTRACTION OF SIGNED NUMBERS



x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

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

Example:

$$\begin{array}{r} X \\ +\,Y \\ \hline Z \end{array} = \begin{array}{r} 7 \\ +\,6 \\ \hline 13 \end{array} = \begin{array}{rrrr} 0 & 1 & \boxed{1} & 1 \\ +\,_0 0 & _1 1 & _1 1 & _0 0\,_0 \\ \hline 1 & 1 & 0 & 1 \end{array}$$

Legend for stage i

At the i^{th} stage:

Input:

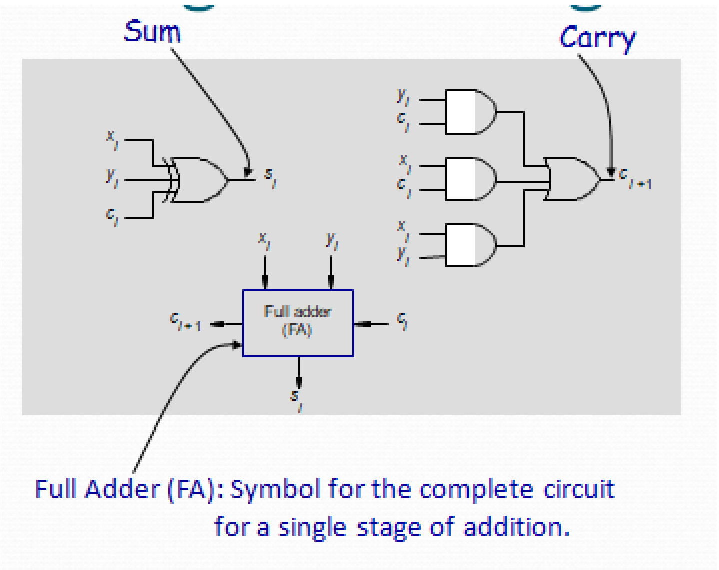
c_i is the carry-in

Output:

s_i is the sum

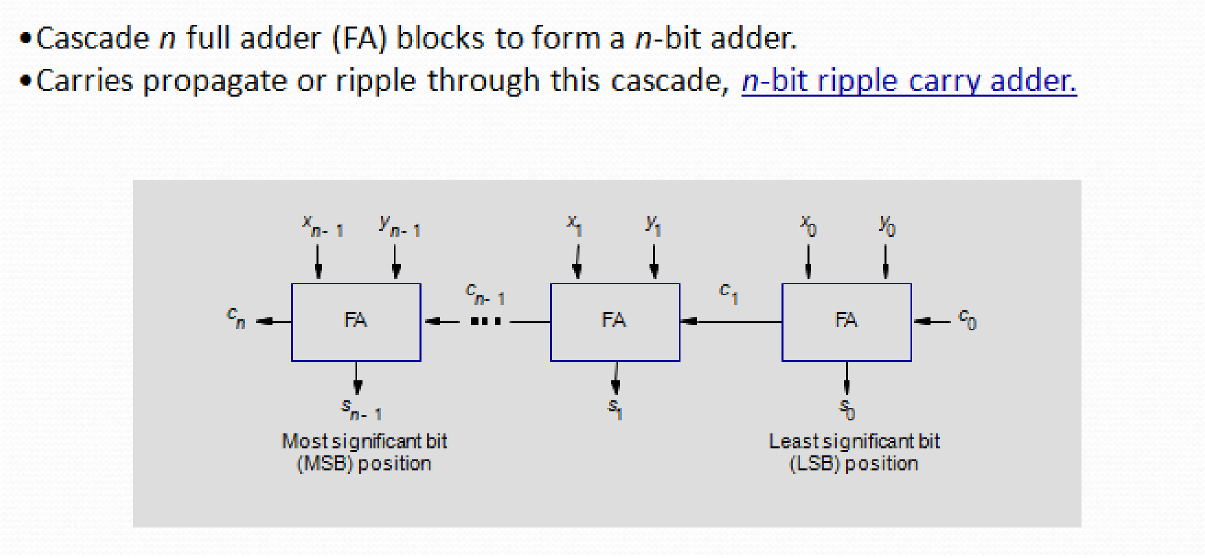
c_{i+1} carry-out

Single stage logic:



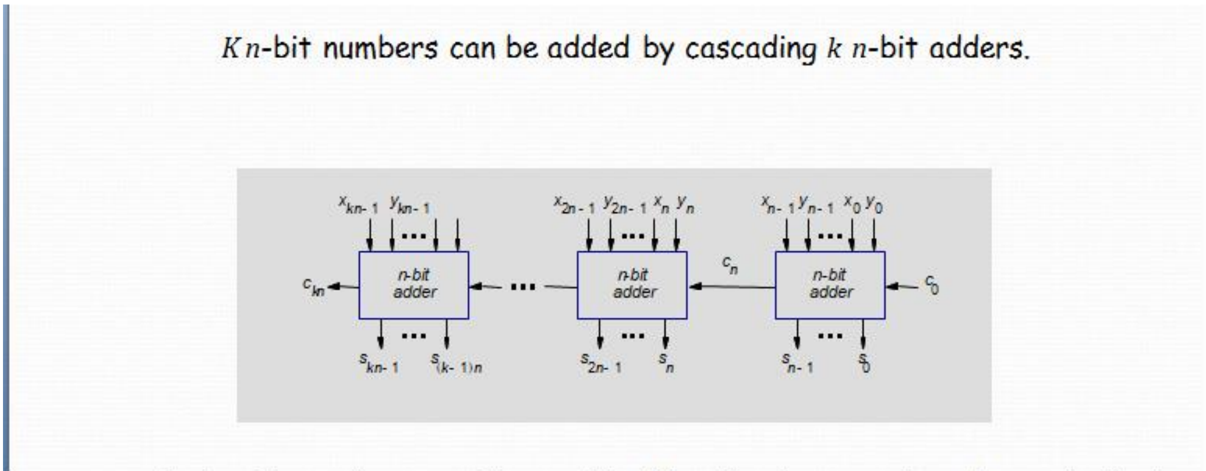
Full Adder (FA): Symbol for the complete circuit for a single stage of addition.

N-Bit Ripple carry adder:



- Cascade *n* full adder (FA) blocks to form a *n*-bit adder.
- Carries propagate or ripple through this cascade, *n-bit ripple carry adder.*

K n-bit adders



K n-bit numbers can be added by cascading *k n*-bit adders.

Interconnecting k adders to form an adder capable of handling input numbers that are kn bits long

Addition/Subtraction Logic Unit

Addition:

n-bit adder used to add 2's complement numbers X and Y.

x_{n-1} and y_{n-1} bits are sign bits. C_n overflow bit ignored.

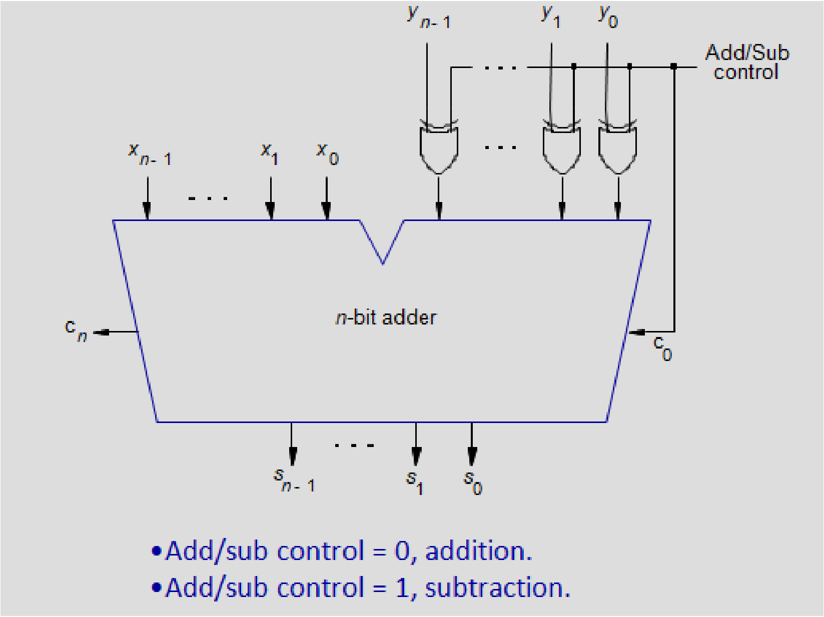
Circuit is added to detect arithmetic overflow.

$$Overflow = x_{n-1}y_{n-1}\bar{s}_{n-1} + \bar{x}_{n-1}\bar{y}_{n-1}s_{n-1}$$

$$Overflow = c_n \oplus c_{n-1}$$

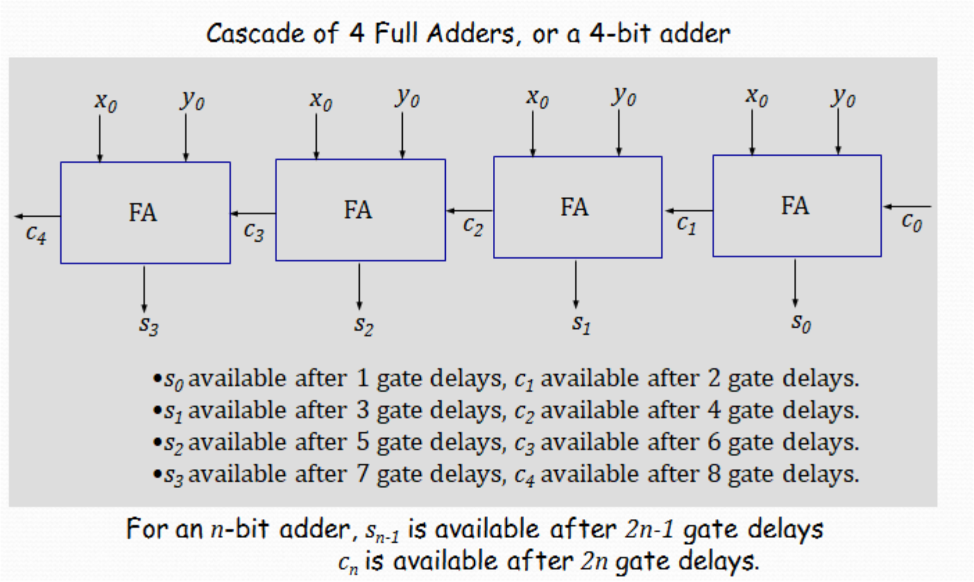
Subtraction:

X – *Y* is equivalent to adding 2's complement of *Y* to *X*



- Add/sub control = 0, addition.
- Add/sub control = 1, subtraction.

Computing gate delay:


Cascade of 4 Full Adders, or a 4-bit adder

- s_0 available after 1 gate delays, c_1 available after 2 gate delays.
- s_1 available after 3 gate delays, c_2 available after 4 gate delays.
- s_2 available after 5 gate delays, c_3 available after 6 gate delays.
- s_3 available after 7 gate delays, c_4 available after 8 gate delays.

For an *n*-bit adder, s_{n-1} is available after *2n-1* gate delays
c_n is available after *2n* gate delays.

DESIGN OF FAST ADDERS

Drawback of ripple-carry adder: Delay in developing outputs s_0 through s_{n-1} and c_n.

Reduce delay by:

- Using fastest electronic-technology in implementing ripple-carry design

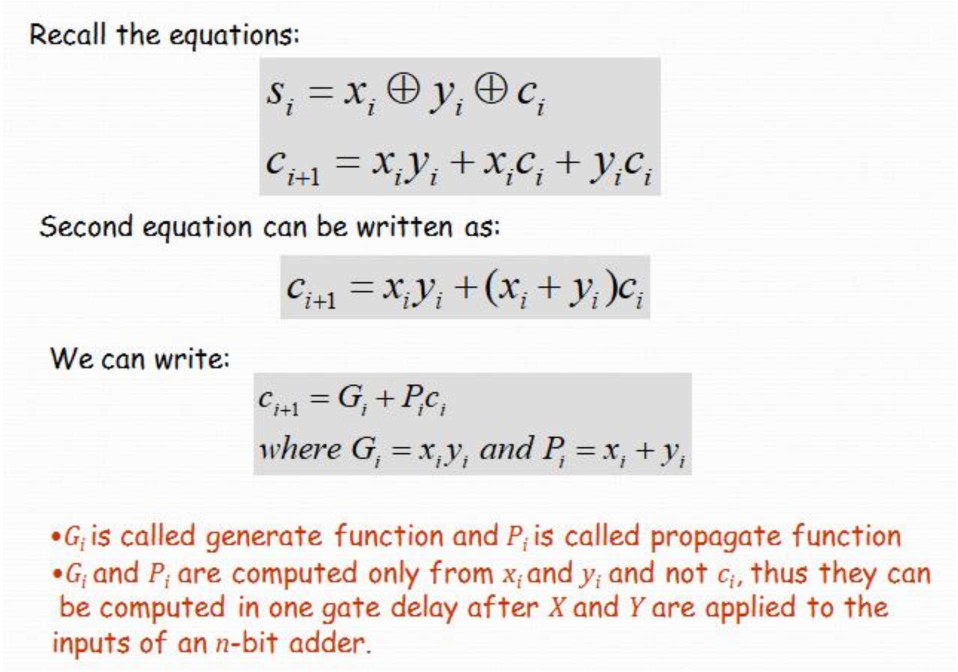
  Or

- Use augmented logic-gate network structure

Carry-Lookahead adder (CLA)

Improving speed of addition will improve speed of all other arithmetic operations.

CLA improves speed by reducing carry propagation delay. It calculates carry signal in advance, based on input signals instead of waiting for them to ripple through the adders.



Recall the equations:

$$s_i = x_i \oplus y_i \oplus c_i$$

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

Second equation can be written as:

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

We can write:

$$c_{i+1} = G_i + P_i c_i$$

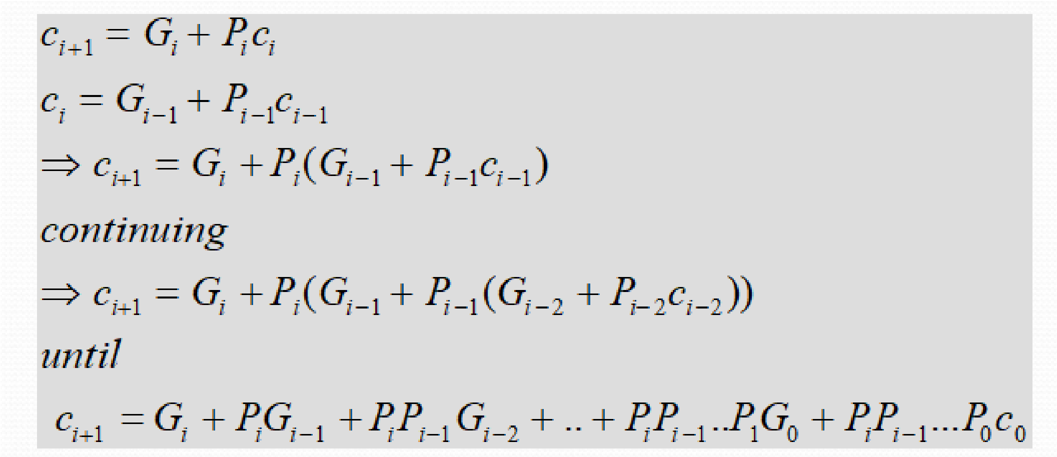
$$where\ G_i = x_i y_i\ and\ P_i = x_i + y_i$$

- G_i is called generate function and P_i is called propagate function
- G_i and P_i are computed only from x_i and y_i and not c_i, thus they can be computed in one gate delay after X and Y are applied to the inputs of an n-bit adder.

G_i: implemented as AND gate

P_i: implemented as XOR gate not OR gate (because if x_i, y_i= 1, then, G_i=1 so it does not matter if P_i is 0 or 1

All G_i and P_i functions can be formed independently and in parallel.



$$c_{i+1} = G_i + P_i c_i$$

$$c_i = G_{i-1} + P_{i-1} c_{i-1}$$

$$\Rightarrow c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1} c_{i-1})$$

$$continuing$$

$$\Rightarrow c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1}(G_{i-2} + P_{i-2} c_{i-2}))$$

$$until$$

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + .. + P_i P_{i-1} .. P_1 G_0 + P_i P_{i-1} ... P_0 c_0$$

c_{i+1} is given in terms of c_0.

Consider 4-bit CLA:

$c_1 = G_0 + P_0 c_0$

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

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

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

Gate delays:

$$s_i = x_i \oplus y_i \oplus c_i$$

$S_i = P_i$ XOR c_i

$$c_{i+1} = G_i + P_i c_i$$

P_i : 1 gate delay for P_i (XOR)

G_i : 1 gate delay for G_i (AND)

total 1 gate delay for both as they are parallel

c_{i+1} : 1 gate delay for G_i, P_i

1 gate delay for $P_i c_i$ (AND)

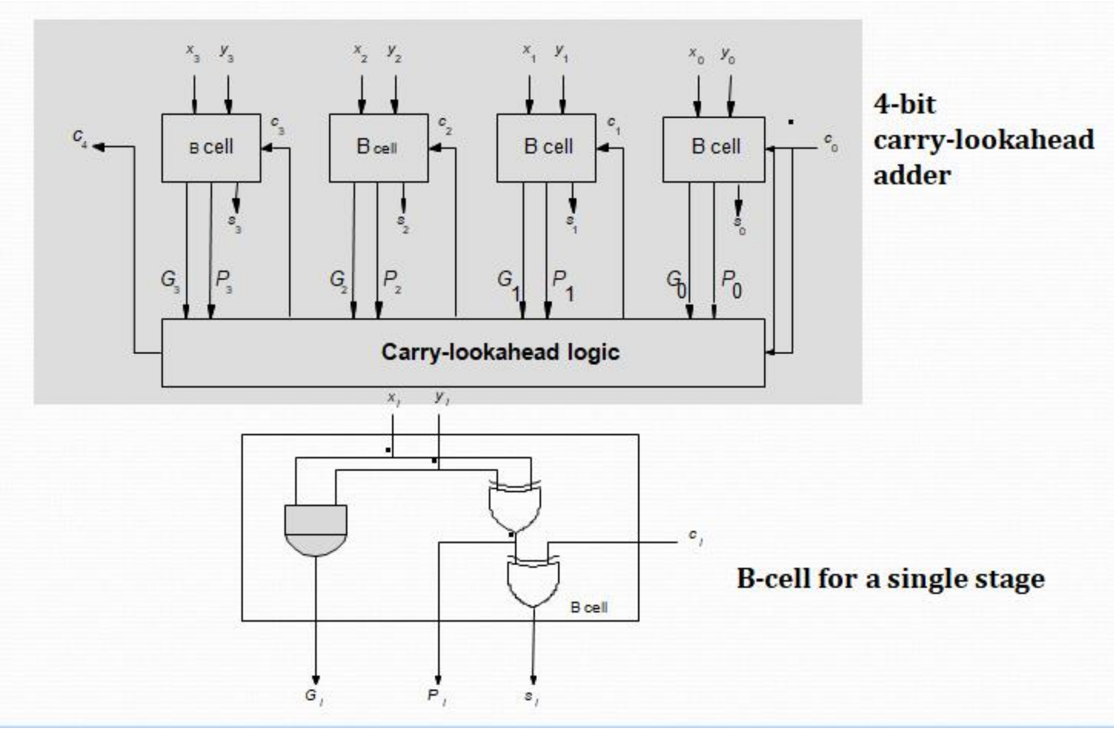
1 gate delay for $G_i + P_i c_i$ (OR)

total 3 gate delays for c_{i+1}

s_i : 3 gate delays for c_i

1 gate delay for P_i XOR c_i

total 4 gate delays for s_i



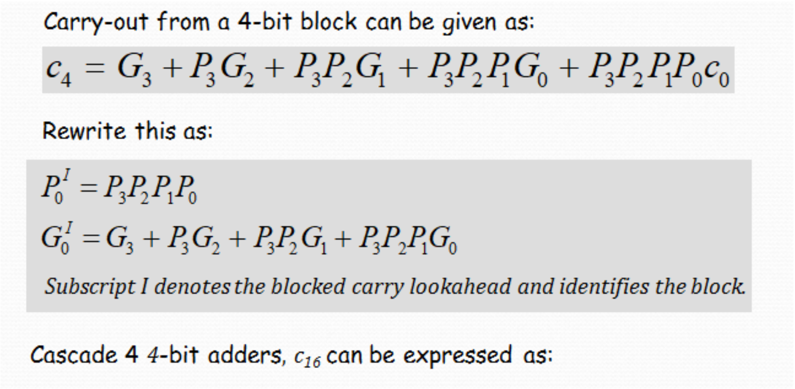
4-bit carry-lookahead adder

B-cell for a single stage

Performing *n*-bit addition in 4 gate delays independent of *n* is good only theoretically because of fan-in constraints.

For a 4-bit adder (*n=4*) fan-in of 5 is required which is Practical limit for most gates.

In order to add operands longer than 4 bits, we can cascade 4-bit Carry-Lookahead adders. Cascade of Carry-Lookahead adders is called Blocked Carry-Lookahead adder.



Carry-out from a 4-bit block can be given as:

$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

Rewrite this as:

$$P_0^I = P_3P_2P_1P_0$$

$$G_0^I = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

Subscript I denotes the blocked carry lookahead and identifies the block.

Cascade 4 4-bit adders, c_{16} can be expressed as:

$$C_{16} = G^I_3 + P^I_3G^I_2 + P^I_3P^I_2G^I_1 + P^I_3P^I_2P^I_1G^I_0 + P^I_3P^I_2P^I_1P^I_0c_0$$

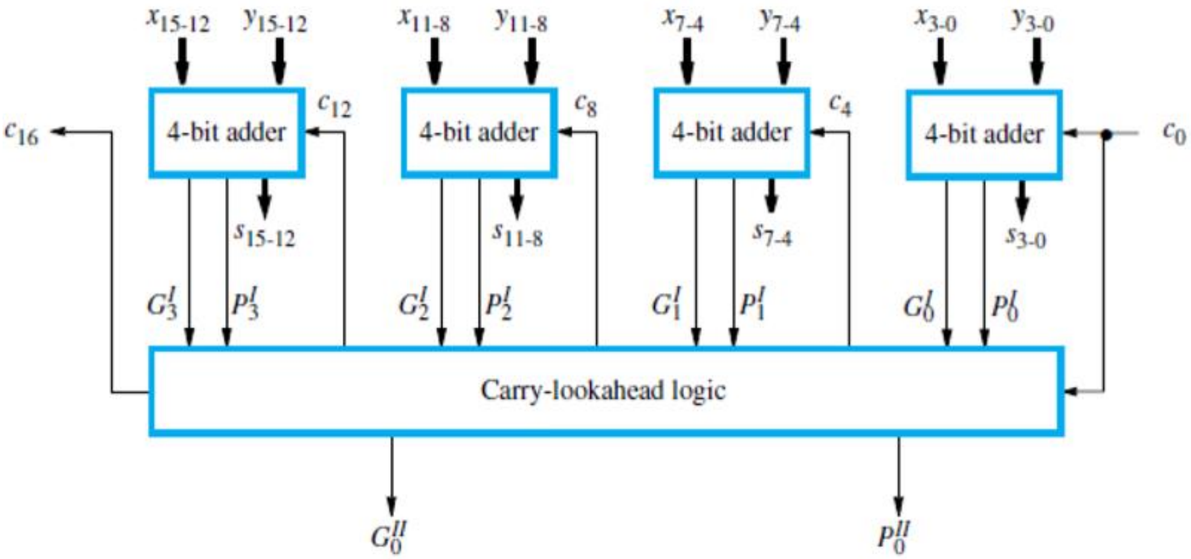
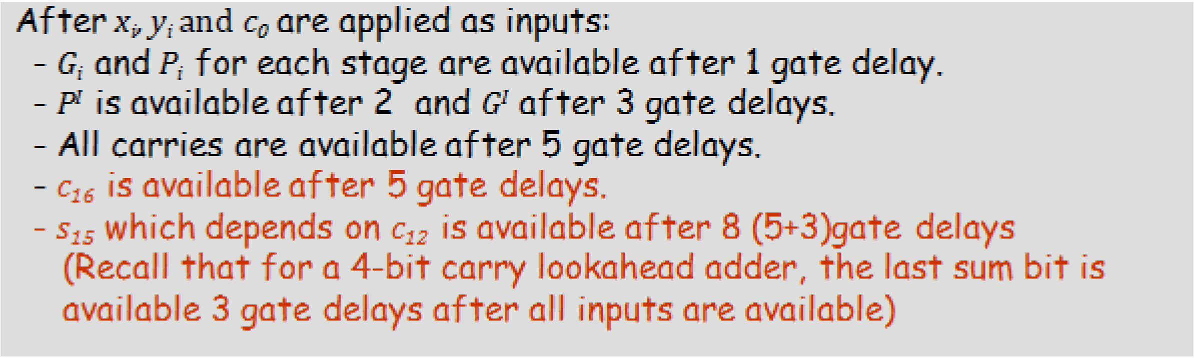


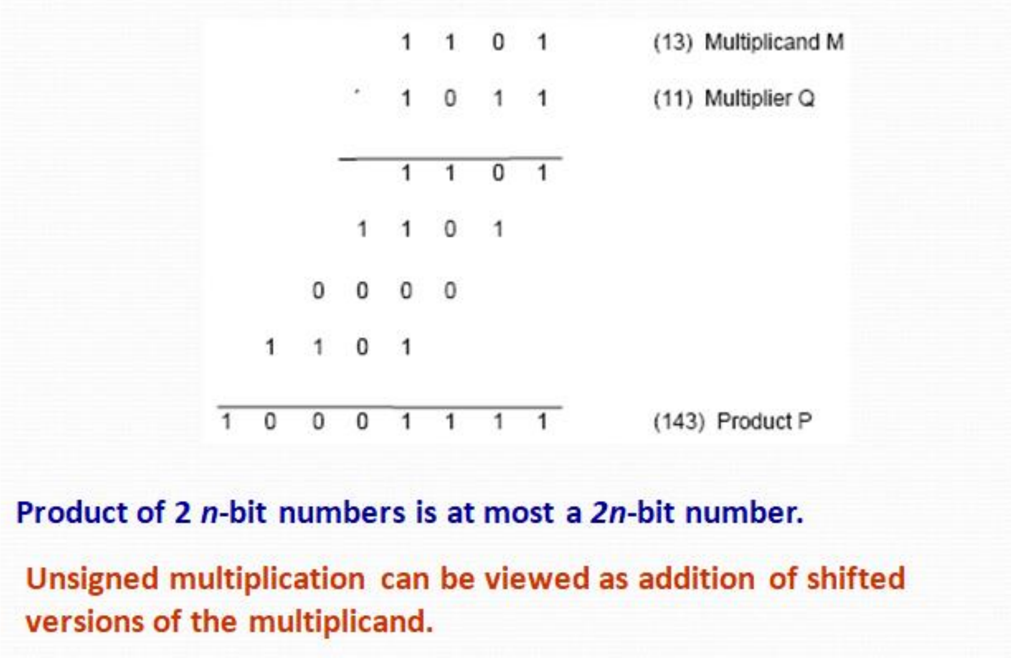
Figure 9.5 A 16-bit carry-lookahead adder built from 4-bit adders (see Figure 9.4*b*).



After x_i, y_i and c_0 are applied as inputs:

- G_i and P_i for each stage are available after 1 gate delay.
- P^I is available after 2 and G^I after 3 gate delays.
- All carries are available after 5 gate delays.
- c_{16} is available after 5 gate delays.
- s_{15} which depends on c_{12} is available after 8 (5+3)gate delays (Recall that for a 4-bit carry lookahead adder, the last sum bit is available 3 gate delays after all inputs are available)

MULTIPLICATION OF POSITIVE NUMBERS



```
            1  1  0  1      (13)  Multiplicand M
            1  0  1  1      (11)  Multiplier Q
         ____________
            1  1  0  1
         1  1  0  1
      0  0  0  0
   1  1  0  1
______________________
1  0  0  0  1  1  1  1      (143)  Product P
```

Product of 2 *n*-bit numbers is at most a 2*n*-bit number.

Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand.

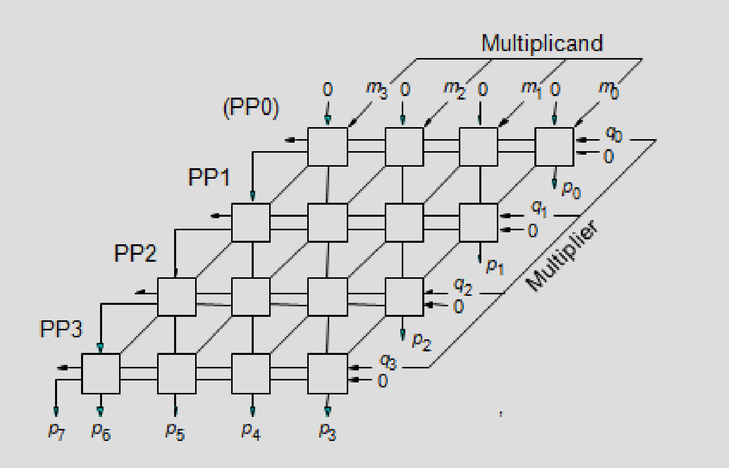
Value of the partial product at the start stage is 0

If multiplier bit is 1, the multiplicand is entered in appropriate position to be added to partial product

If multiplier bit is 0, then 0s are entered.

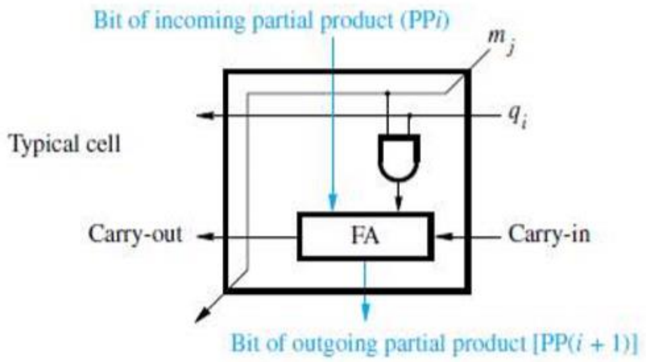
Combinatorial Array multiplication:

Combinatorial array multiplier

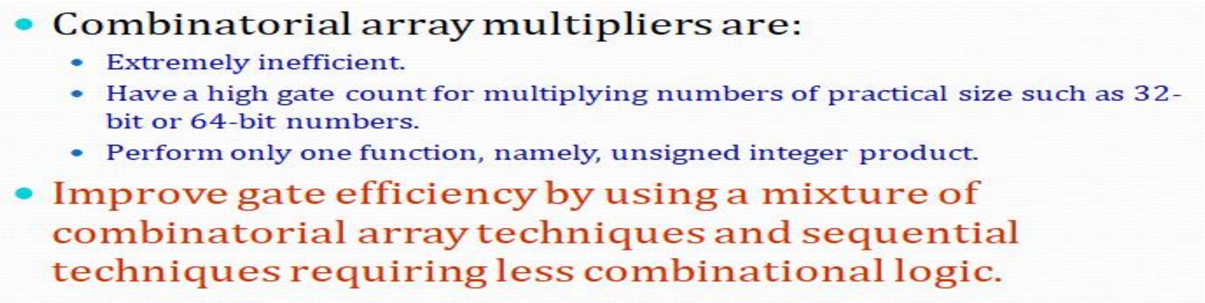


Product is: $p_7, p_6, ..p_0$

Multiplicand is shifted by displacing it through an array of adders.

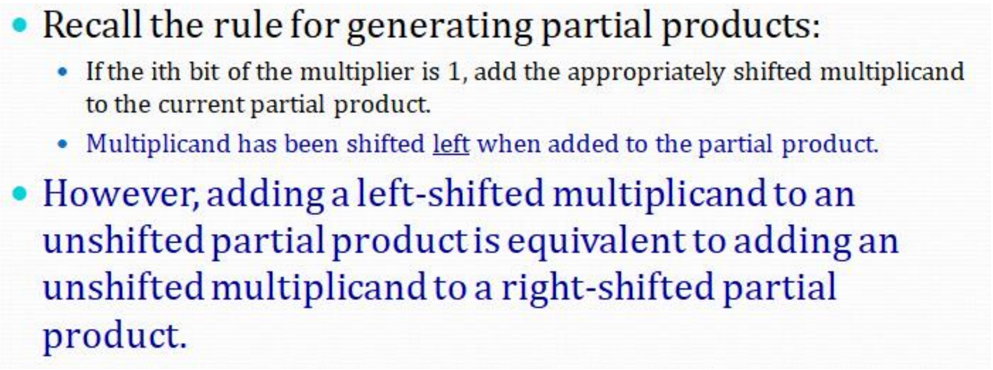

Bit of incoming partial product (PPi)
m_j
q_i
Typical cell
Carry-out
FA
Carry-in
Bit of outgoing partial product [PP(i + 1)]

(b) Array implementation

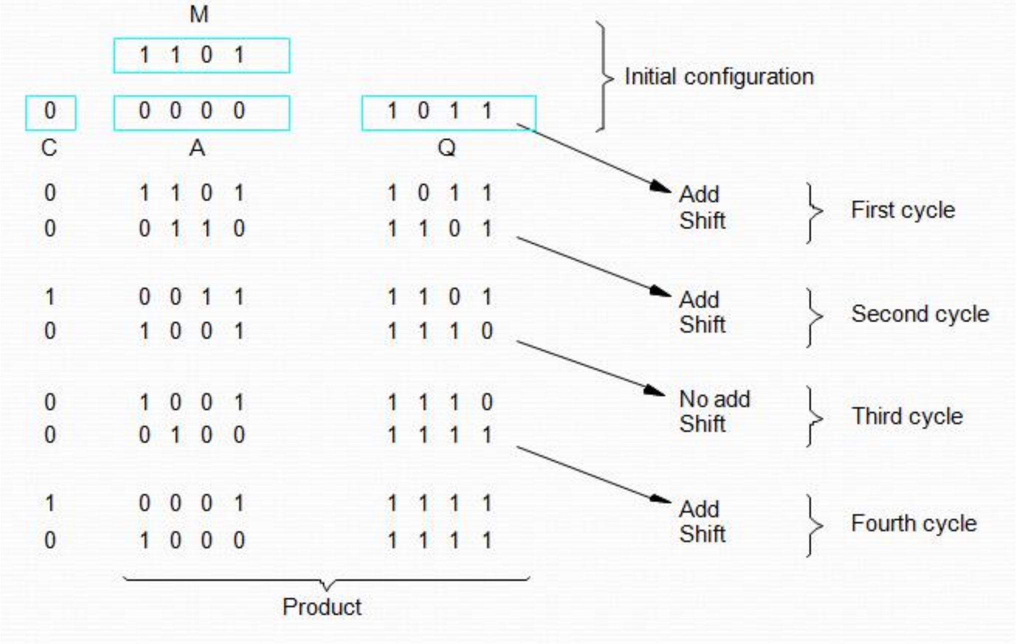


- Combinatorial array multipliers are:
  - Extremely inefficient.
  - Have a high gate count for multiplying numbers of practical size such as 32-bit or 64-bit numbers.
  - Perform only one function, namely, unsigned integer product.
- Improve gate efficiency by using a mixture of combinatorial array techniques and sequential techniques requiring less combinational logic.

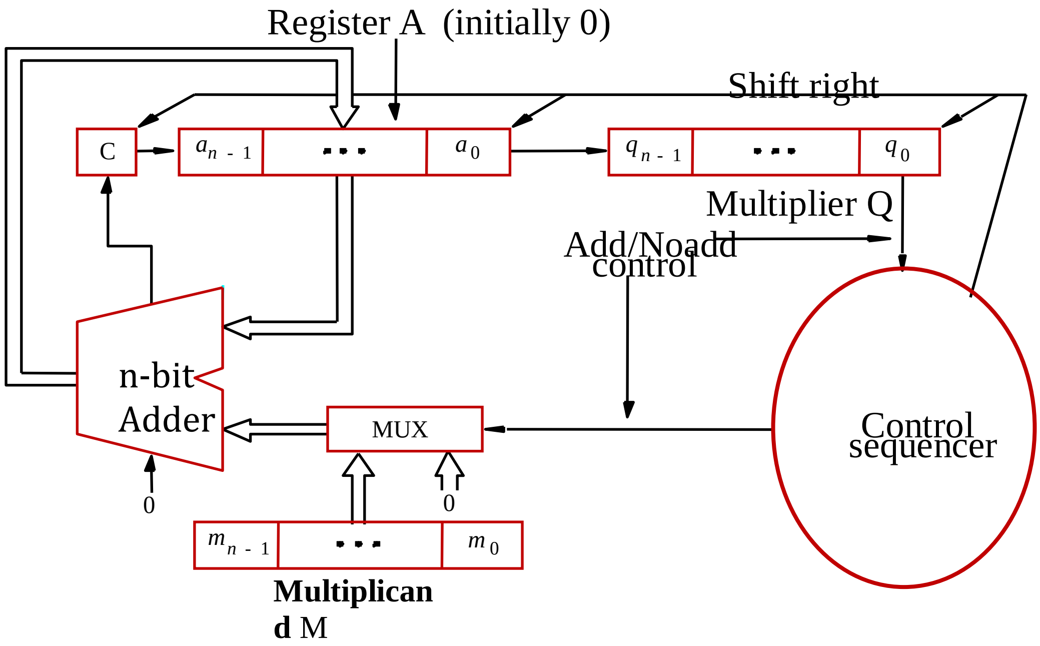
Sequential multiplication:



- Recall the rule for generating partial products:
  - If the ith bit of the multiplier is 1, add the appropriately shifted multiplicand to the current partial product.
  - Multiplicand has been shifted left when added to the partial product.
- However, adding a left-shifted multiplicand to an unshifted partial product is equivalent to adding an unshifted multiplicand to a right-shifted partial product.



C	A	Q		
	M 1 1 0 1			Initial configuration
0	0 0 0 0	1 0 1 1		
0	1 1 0 1	1 0 1 1	Add	First cycle
0	0 1 1 0	1 1 0 1	Shift	
1	0 0 1 1	1 1 0 1	Add	Second cycle
0	1 0 0 1	1 1 1 0	Shift	
0	1 0 0 1	1 1 1 0	No add	Third cycle
0	0 1 0 0	1 1 1 1	Shift	
1	0 0 0 1	1 1 1 1	Add	Fourth cycle
0	1 0 0 0	1 1 1 1	Shift	

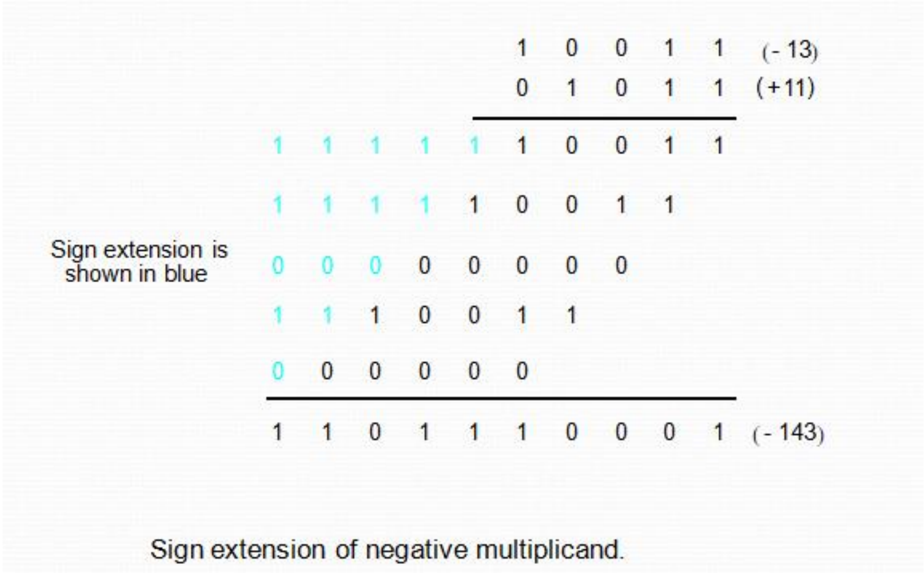


Sequential circuit binary multiplier/ register configuration for hardware arrangement for sequential multiplication

SIGNED OPERAND MULTIPLICATION

For positive multiplier and negative multiplicand (i.e, -AxB), perform sign extension on multiplicand while multiplication.

For negative multiplier, for 2s complement of both multiplier and multiplicand and perform sign extension while multiplication.

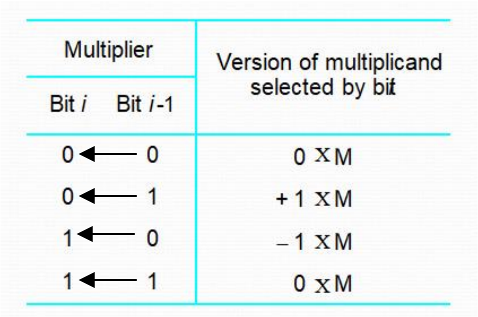


```
                                 1  0  0  1  1   (- 13)
                                 0  1  0  1  1   (+11)
                               ---------------
                  1  1  1  1  1  1  0  0  1  1
                  1  1  1  1  1  0  0  1  1
Sign extension is
 shown in blue    0  0  0  0  0  0  0  0
                  1  1  1  0  0  1  1
                  0  0  0  0  0  0
                 -----------------------------
                  1  1  0  1  1  1  0  0  0  1   (- 143)
```

Sign extension of negative multiplicand.

BOOTH ALGORITHM

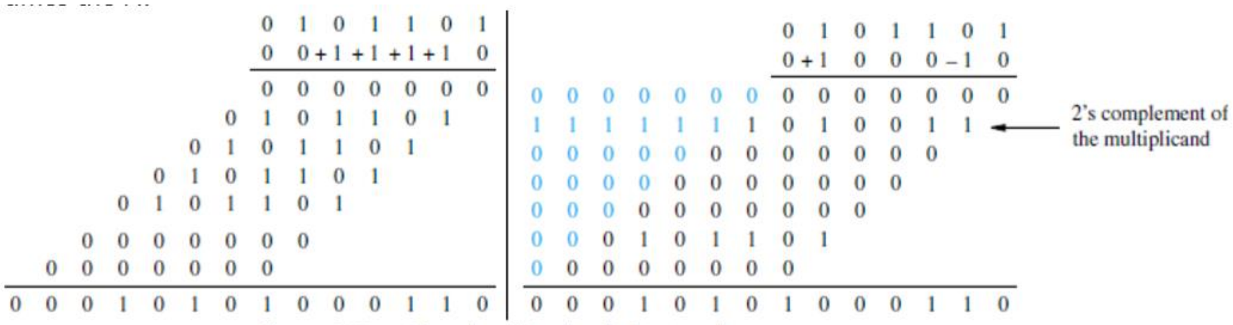
It generates 2n-bit product and treats both positive and negative 2’s-complement n-bit operands uniformly.



Multiplier		Version of multiplicand selected by bit
Bit *i*	Bit *i*-1	
0 ←	0	0 X M
0 ←	1	+ 1 X M
1 ←	0	− 1 X M
1 ←	1	0 X M

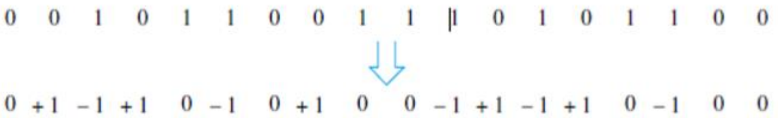
Multiply by -1 means take 2’s complement of multiplicand

Multiply by +1 means take the multiplicand as it is.



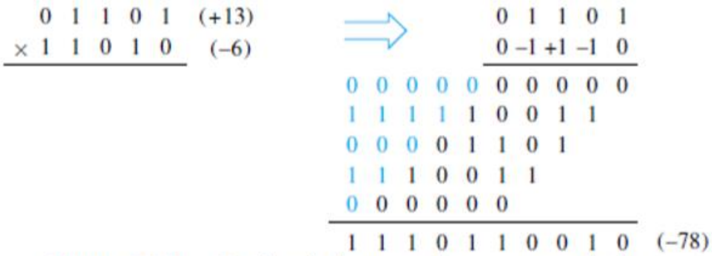
```
              0 1 0 1 1 0 1  |                0 1 0 1 1 0 1
              0 0+1+1+1+1 0  |                0+1 0 0 0-1 0
              -------------  |                -------------
              0 0 0 0 0 0 0  |  0 0 0 0 0 0 0 0 0 0 0 0 0 0
            0 1 0 1 1 0 1    |  1 1 1 1 1 1 1 0 1 0 0 1 1      ← 2's complement of
          0 1 0 1 1 0 1      |  0 0 0 0 0 0 0 0 0 0 0 0          the multiplicand
        0 1 0 1 1 0 1        |  0 0 0 0 0 0 0 0 0 0 0
      0 1 0 1 1 0 1          |  0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0            |  0 0 0 1 0 1 1 0 1
  0 0 0 0 0 0 0              |  0 0 0 0 0 0 0 0
---------------------------  |  ---------------------------
0 0 0 1 0 1 0 1 0 0 0 1 1 0  |  0 0 0 1 0 1 0 1 0 0 0 1 1 0
```

Figure 9.9 Normal and Booth multiplication schemes.



```
0  0  1  0  1  1  0  0  1  1 |1  0  1  0  1  1  0  0
                    ⇓
0 +1 −1 +1  0 −1  0 +1  0  0 −1 +1 −1 +1  0 −1  0  0
```

Figure 9.10 Booth recoding of a multiplier.



```
  0 1 1 0 1  (+13)               0  1  1  0  1
× 1 1 0 1 0  (−6)      ⇒         0 −1 +1 −1  0
-----------                      -------------
                       0 0 0 0 0 0  0  0  0  0
                       1 1 1 1 1 0  0  1  1
                       0 0 0 0 1 1  0  1
                       1 1 1 0 0 1  1
                       0 0 0 0 0 0
                       -----------------------
                       1 1 1 0 1 1  0  0  1  0  (−78)
```

Figure 9.11 Booth multiplication with a negative multiplier.

- Attractive feature: This algorithm achieves some efficiency in the number of addition required when the multiplier has a few large blocks of 1s.

FAST MULTIPLICATION

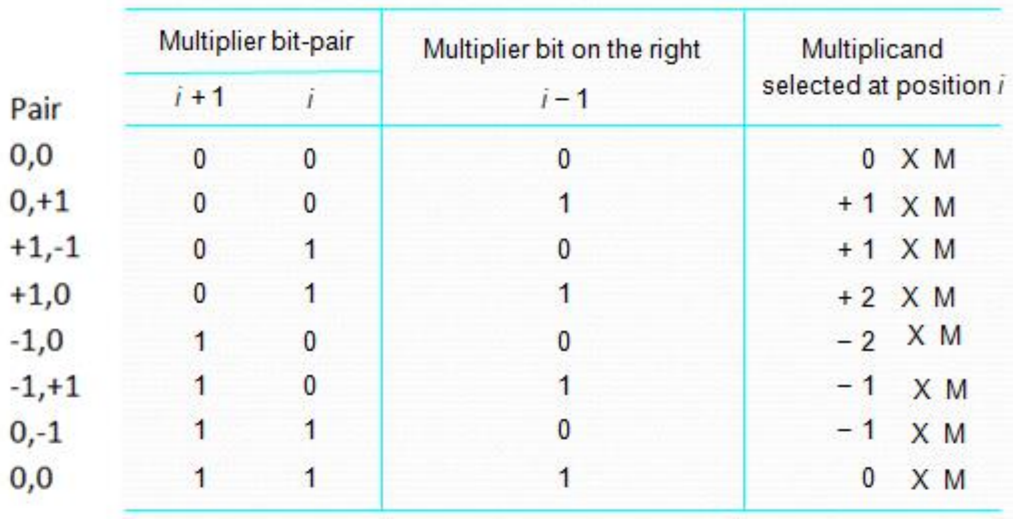
Two techniques:

Bit-pair recoding of multiplier- halves maximum number of summands

Carry-save addition- reduces time needed to add the summands

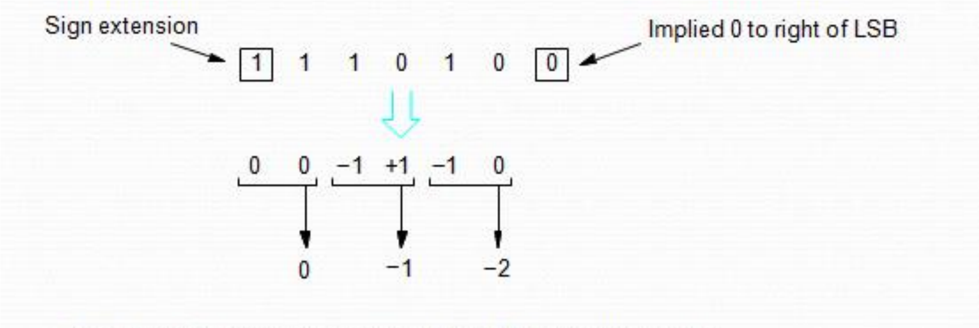
Bit-pair recoding

halves maximum number of summands (versions of multiplicand) that must be added to n/2 for n-bit operands.

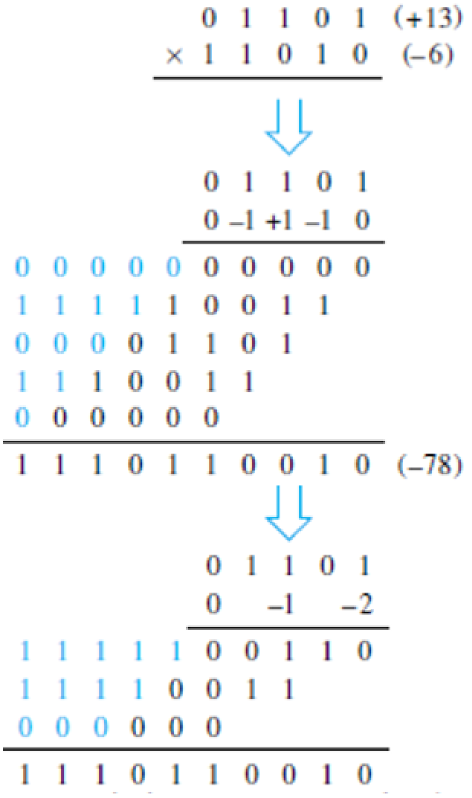


Pair	Multiplier bit-pair $i+1$	 i	Multiplier bit on the right $i-1$	Multiplicand selected at position i
0,0	0	0	0	0 X M
0,+1	0	0	1	+ 1 X M
+1,-1	0	1	0	+ 1 X M
+1,0	0	1	1	+ 2 X M
-1,0	1	0	0	− 2 X M
-1,+1	1	0	1	− 1 X M
0,-1	1	1	0	− 1 X M
0,0	1	1	1	0 X M

(b) Table of multiplicand selection decisions



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



```
          0  1  1  0  1   (+13)
        × 1  1  0  1  0   (−6)
        ---------------
          0  1  1  0  1
          0 −1 +1 −1  0
        ---------------
0  0  0  0  0  0  0  0  0  0
1  1  1  1  1  0  0  1  1
0  0  0  0  1  1  0  1
1  1  1  0  0  1  1
0  0  0  0  0  0
-----------------------------
1  1  1  0  1  1  0  0  1  0   (−78)

          0  1  1  0  1
          0    −1    −2
        ---------------
1  1  1  1  1  0  0  1  1  0
1  1  1  1  0  0  1  1
0  0  0  0  0  0
-----------------------------
1  1  1  0  1  1  0  0  1  0
```

Multiplication requiring only $n/2$ summands.

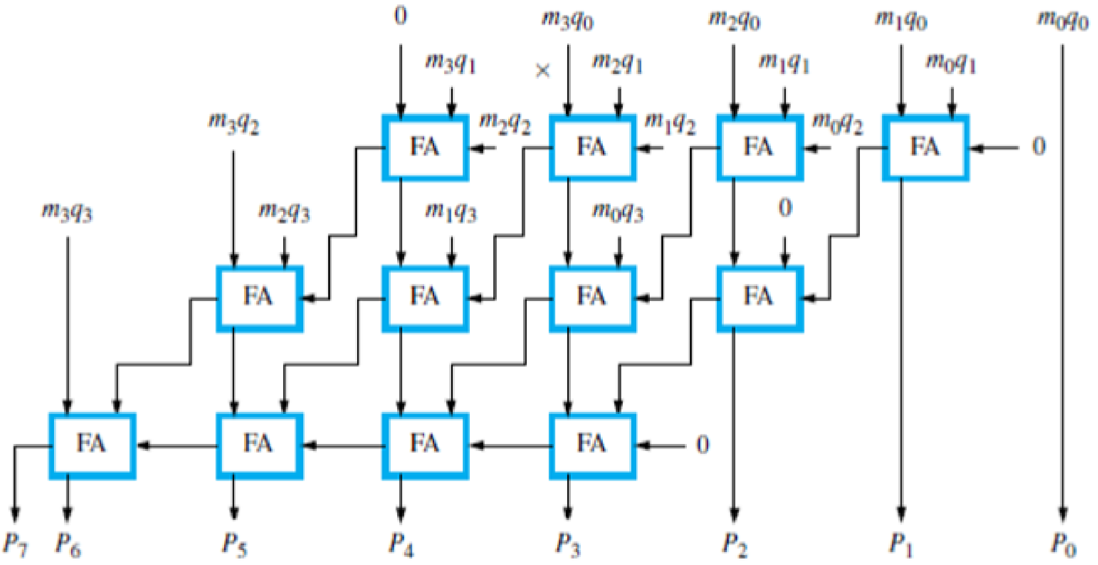
Carry-save addition

reduces time needed to add the summands

Instead of carries ripple along rows, they are saved and introduced into next row at correct weighted positions.

Delay is lesser than ripple-carry adder. Because s and c vector from each row are produced in parallel in one full adder delay

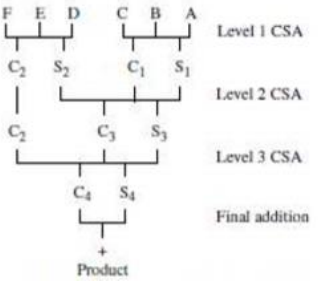
More efficient- grouping summands

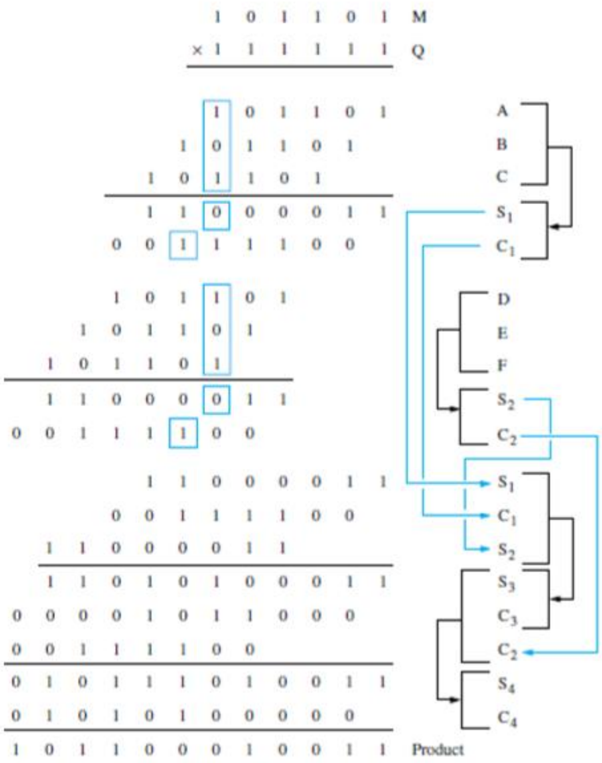


(b) Carry-save array

```
                  1   0   1   1   0   1     (45)   M
              ×   1   1   1   1   1   1     (63)   Q
              ---------------------------
                  1   0   1   1   0   1            A
              1   0   1   1   0   1                B
          1   0   1   1   0   1                    C
      1   0   1   1   0   1                        D
  1   0   1   1   0   1                            E
1 0   1   1   0   1                                F
---------------------------------------
0 1   1   0   0   0   1   0   0   1   1   (2,835) Product
```

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





```
                  1  0  1  1  0  1        M
                × 1  1  1  1  1  1        Q
                -----------------
                  1  0  1  1  0  1        A
               1  0  1  1  0  1           B
            1  0  1  1  0  1              C
            -----------------------
            1  1  0  0  0  0  1  1        S1
      0  0  1  1  1  1  0  0              C1

      1  0  1  1  0  1                    D
   1  0  1  1  0  1                       E
1  0  1  1  0  1                          F
--------------------------
1  1  0  0  0  0  1  1                    S2
0  0  1  1  1  1  0  0                    C2

            1  1  0  0  0  0  1  1        S1
         0  0  1  1  1  1  0  0           C1
   1  1  0  0  0  0  1  1                 S2
   ---------------------------------
   1  1  0  1  0  1  0  0  0  1  1        S3
0  0  0  0  1  0  1  1  0  0  0           C3
0  0  1  1  1  1  0  0                    C2
------------------------------------
0  1  0  1  1  1  0  1  0  0  1  1        S4
0  1  0  1  0  1  0  0  0  0  0           C4
------------------------------------
1  0  1  1  0  0  0  1  0  0  1  1        Product
```

INTEGER DIVISION

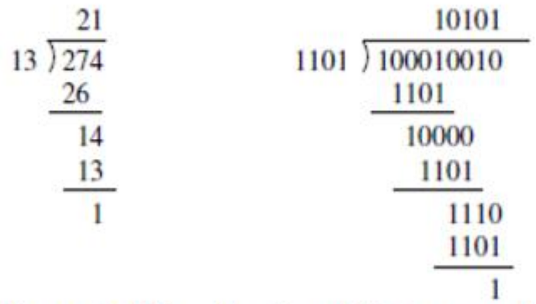


Figure 9.22 Longhand division examples.

NON-RESTORING DIVISION

- Procedure:

Step 1: Do the following n times

i) If the sign of A is 0, shift A and Q left one bit position and subtract M from otherwise, shift A and Q left and add M to A (Figure 9.23).

ii) 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).

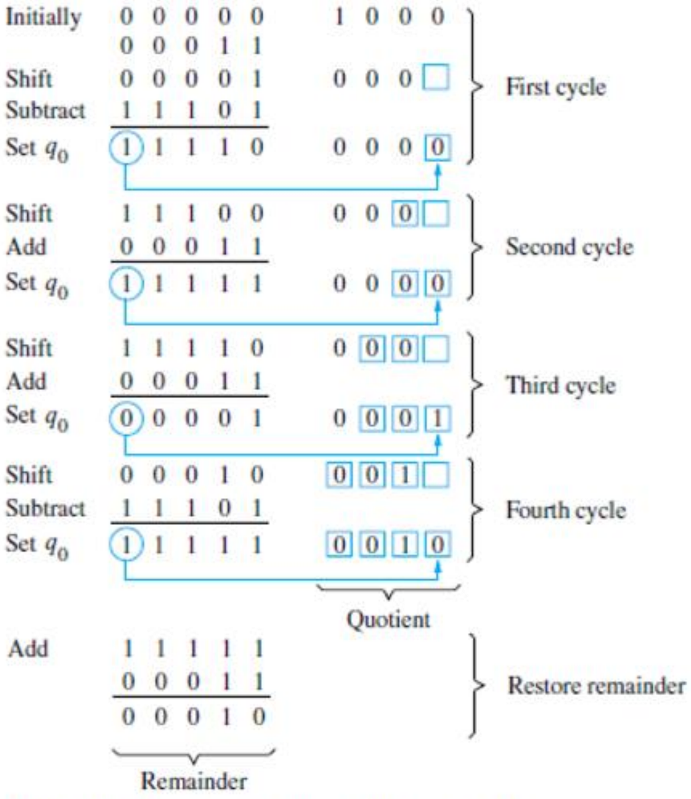


Figure 9.25 A non-restoring division example.

- 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.

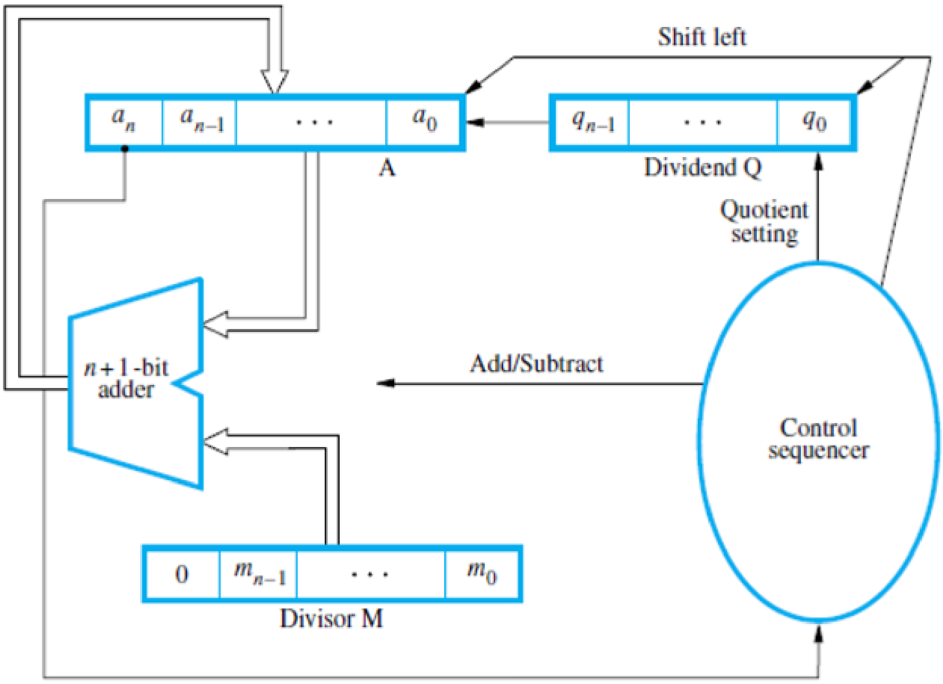


Figure 9.23 Circuit arrangement for binary division.

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.

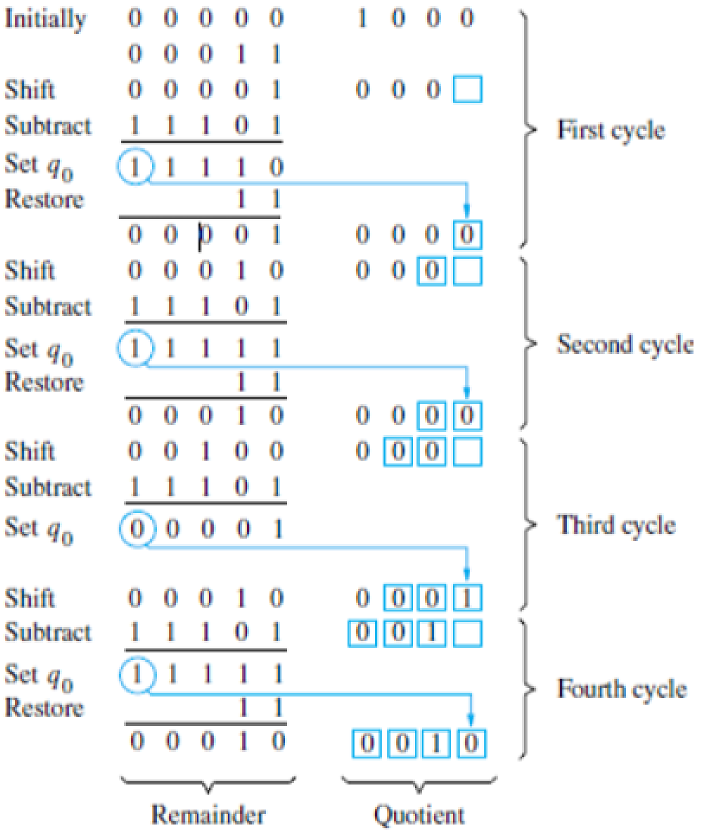


Figure 9.24 A restoring division example.