MODULE 4: ARITHMETIC

Q.1 Perform following operation on 5 bit signed numbers using 2's complement representation system. Also indicate whether overflow has occurred.
a) (-9) + (-7)b) (+7)-(-8)
Q.2. Design a logic circuit to perform addition/subtraction of n bit number X and Y.
Q.3. Draw 4 bit carry-look-ahead adder and explain. (OR) Explain fast adder
Q.4 Explain with figure the design and working of a 16 bit carry-look-ahead adder built using 4 bit adder.
Q.5 Explain Sequential circuit for binary multiplication
Q.6 Explain Booth algorithm for binary multiplication. State its advantages and disadvantages.
Q.7 Explain Fast Multiplication Method Bit pair recoding of multipliers
Q. 8Explain the concept of carry save addition for the multiplication operation M x Q=P for 4 bit operands with diagram and suitable example.
Q. 9 Explain with figure circuit arrangement for binary division
Q.10Give an algorithm and explain with an example both restoring and non-restoring methods for integer division.
Q. 11 Explain Excess notation, Normalization and special conditions for exponents.
Q. 12 Explain IEEE standard for floating point numbers.
Q. 13With a neat diagram, explain the floating point addition/subtraction.

Problems:

1. Perform Multiplication of -13 and +09 using Booth's Algorithm.
2. Perform signed multiplication of numbers (-12) and (-11) using Booth's Algorithm.
3. Given A=10101 and B=00100 perform A/B using restoring division algorithm.
4. Perform restoring division given numbers 1000/11 show all the cycles.(Text book Problem)
5. Perform 14*-8 using Booth's algorithm
6. Apply Booth's algorithm to multiply signed numbers +13 and -6.(text book Problem)
7. Multiply the following signed 2's complement numbers using Booth's algorithm Multiplicand=$(010111)_2$ and Multiplier=$(110110)_2$
8. Perform division operation on the following unsigned numbers using the restoring method Dividend=$(101010)_2$ and Divisor=$(00100)_2$

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 the sign-and-magnitude system, negative value is obtained by changing the MSB from 0 to 1 of the corresponding positive value. For example, +5 is represented by 0101 &
-5 is represented by 1101.

• In 1's complement representation, negative values are obtained by complementing each bit of the corresponding positive number. For example, -3 is obtained by complementing each bit in 0011 to yield 1100. (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 the 2's complement system, forming the 2's complement of a number is done by subtracting that number from 2^n.
(In other words, the 2's complement of a number is obtained by adding 1 to the 1's complement of that number).

• The 2's complement system yields the most efficient way to carry out addition and subtraction operations.

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 2.1 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

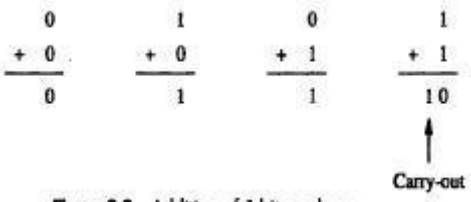


Figure 2.2 Addition of 1-bit numbers.

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.**

1) To add two numbers, add their n-bits and ignore the carry-out signal from the MSB position. The sum will be algebraically correct value as long as the answer is in the range -2^{n-1} through $+2^{n-1}-1$ (Figure 2.4).

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(cn) cannot be ignored. If cn=0, the result obtained is correct. If cn=1, then a 1 must be added to the result to make it correct.

OVERFLOW IN INTEGER ARITHMETIC

• When the result of an arithmetic operation is outside the representable-range, an arithmetic overflow is said to occur.

• For example, when using 4-bit signed numbers, if we try to add the numbers +7 and +4, the output sum S is 1011, 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.

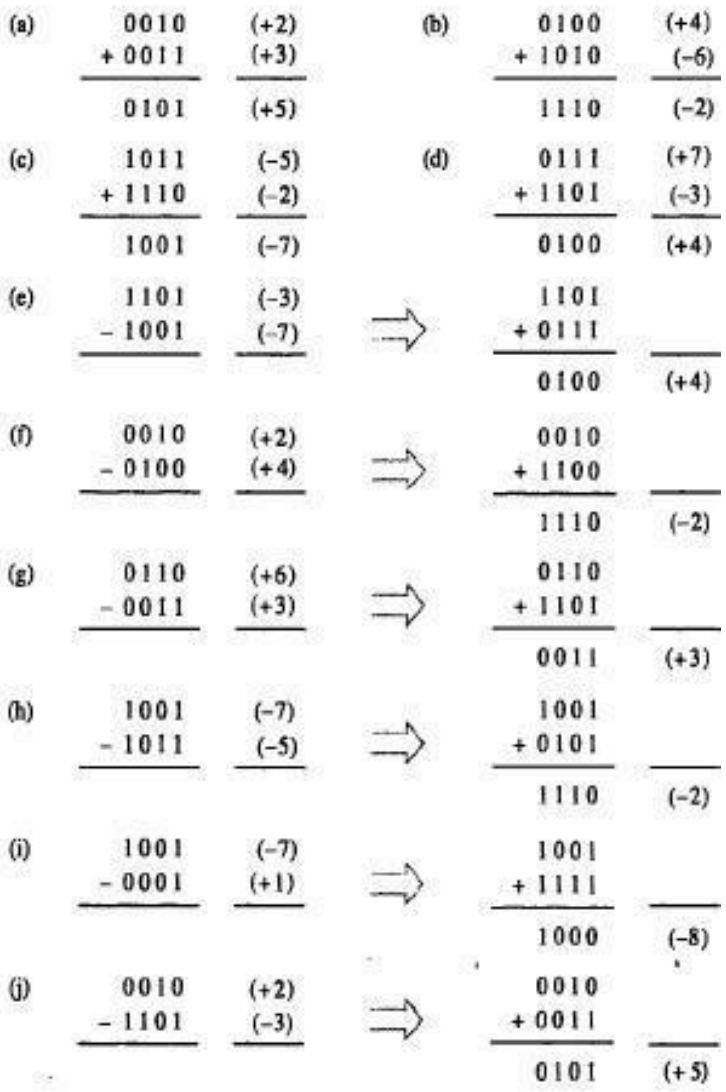


Figure 2.4 2's-complement add and subtract operations.

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.(Fig 6.2).

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:

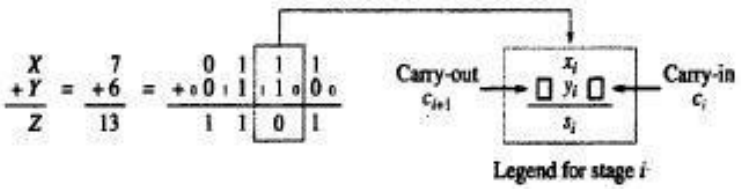
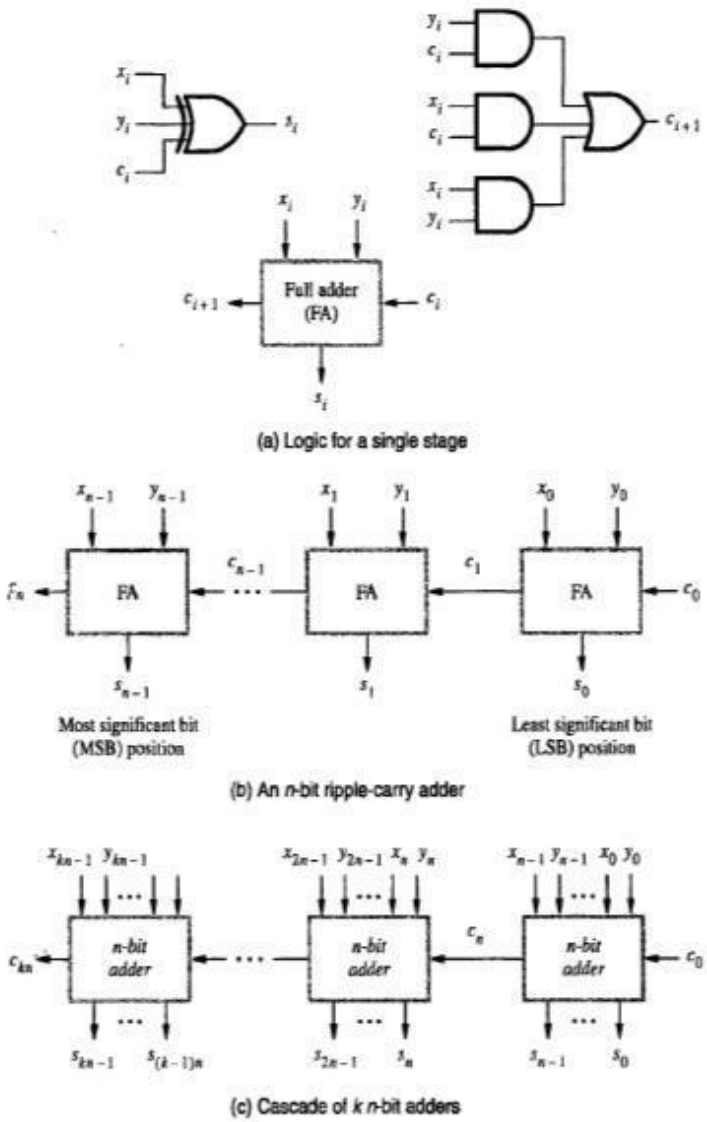


Figure 6.1 Logic specification for a stage of binary addition.



(a) Logic for a single stage

(b) An *n*-bit ripple-carry adder

(c) Cascade of *k* *n*-bit adders

Figure 6.2 Logic for addition of binary vectors.

ADDITION/SUBTRACTION LOGIC UNIT

Q.2 Design a logic circuit to perform addition/subtraction of n bit number X and Y.

- The n-bit adder can be used to add 2's complement numbers X and Y (Figure 6.3).
- 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.

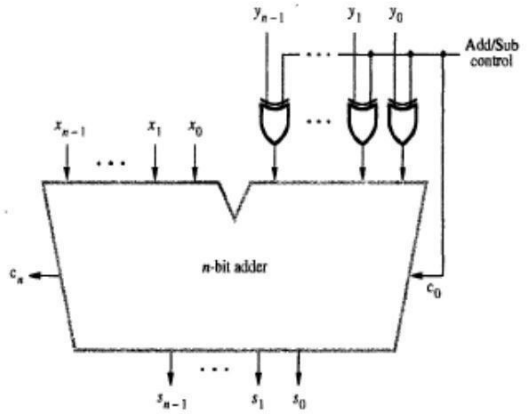


Figure 6.3 Binary addition-subtraction logic network.

DESIGN OF FAST ADDERS

Q.3. Draw 4 bit carry look ahead adder and explain. (OR) Explain fast adder

• 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:

i) Use the fastest possible electronic-technology in implementing the ripple-carry design

ii) Use an augmented logic-gate network structure

CARRY-LOOKAHEAD ADDITIONS

• The logic expression for si(sum) and ci+1(carry-out) of stage i are

$s_i = x_i + y_i + c_i$ ------(1) $c_{i+1} = x_i y_i + x_i c_i + y_i c_i$ ------(2)

• Factoring (2) into

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

we can write

$c_{i+1} = G_i + P_i C_i$ where $G_i = x_i y_i$ and $P_i = x_i + y_i$

• The expressions G_i and P_i are called generate and propagate functions (Figure 6.4).

• If $G_i = 1$, then $c_{i+1} = 1$, independent of the input carry c_i. This occurs when both x_i and y_i are 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

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

• 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

$c_1 = G_0 + P_0 c_0$

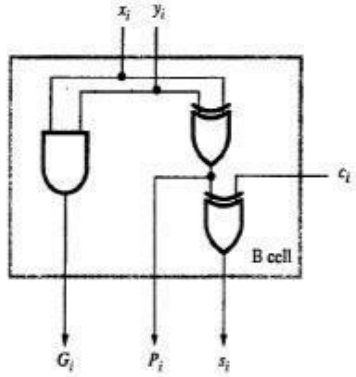
$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$

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

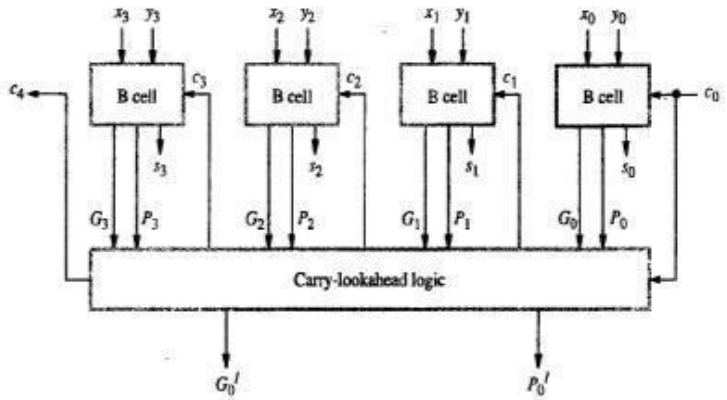
$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$

• 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.



(a) Bit-stage cell



HIGHER-LEVEL GENERATE & PROPAGATE FUNCTIONS

Q.4 Explain with figure the design and working of a 16 bit carry-look-ahead adder built using 4 bit adder.

- 16-bit adder can be built from four 4-bit adder blocks (Figure 6.5).
- 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,

  P0=P3P2P1P0

  &

  G0=G3+P3G2+P3P2G1+P3P2P1G0
- The first-level Gi and Pi functions determine whether bit stage i generates or propagates a carry, and the second level Gk and Pk functions determine whether block k generates or propagates a carry.
- Carry c_{16} is formed by one of the carry-lookahead circuits as

  c16=G3+P3G2+P3P2G1+P3P2P1G0+P3P2P1P0c0
- Conclusion: All carries are available 5 gate delays after X, Y and c0 are applied as inputs.

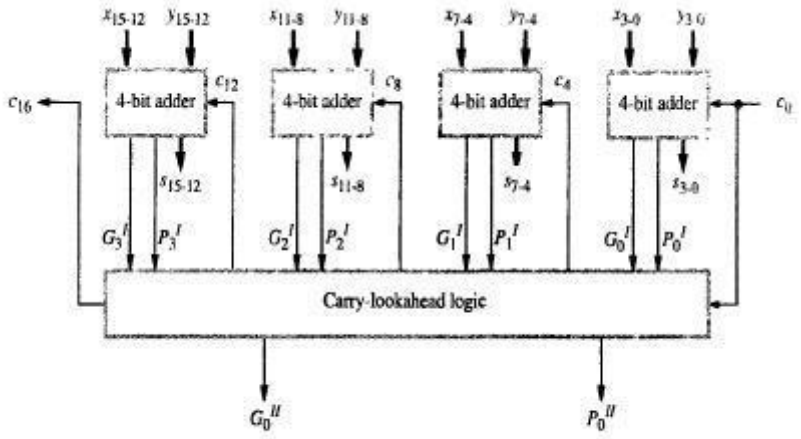
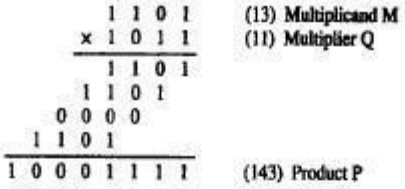
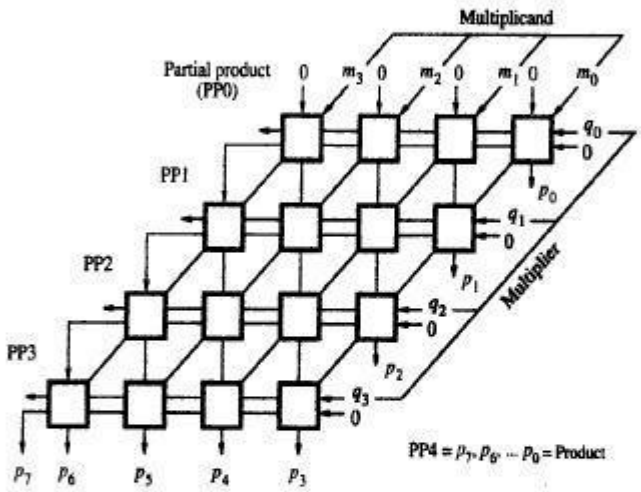


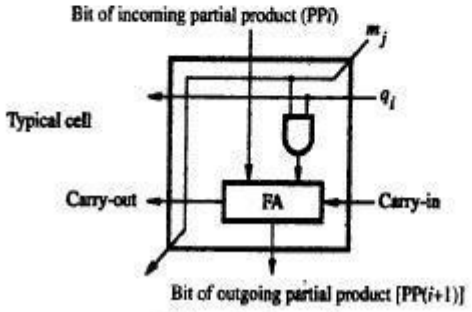
Figure 6.5 16-bit carry-lookahead adder built from 4-bit adders (see Figure 6.4b).

MULTIPLICATION OF POSITIVE NUMBERS



(a) Manual multiplication algorithm





(b) Array implementation

Figure 6.6 Array multiplication of positive binary operands.

ARRAY MULTIPLICATION using *Combinatorial array multipliers*

- We added the partial products at end.
  - Alternative would be to add the partial products at each stage.
- The AND gate in each cell determines whether a multiplicand bit mj, is added to the incoming partial-product bit, based on the value of the multiplier bit qi
- **Rules to implement multiplication are:**
  - Value of the partial product at the start stage is 0.
  - At each row, if the multiplier bit q_i =1, shift the multiplicand and add the shifted multiplicand to the current value of the partial product PP_i.to generate outgoing partial product PP(i+1).
  - If q_i =0, Pp_i is passed vertically downward unchanged.
  - Hand over the partial product to the next stage
  - PP4 is the desired product.
- **Combinatorial array multipliers are:**
  - Extremely inefficient.

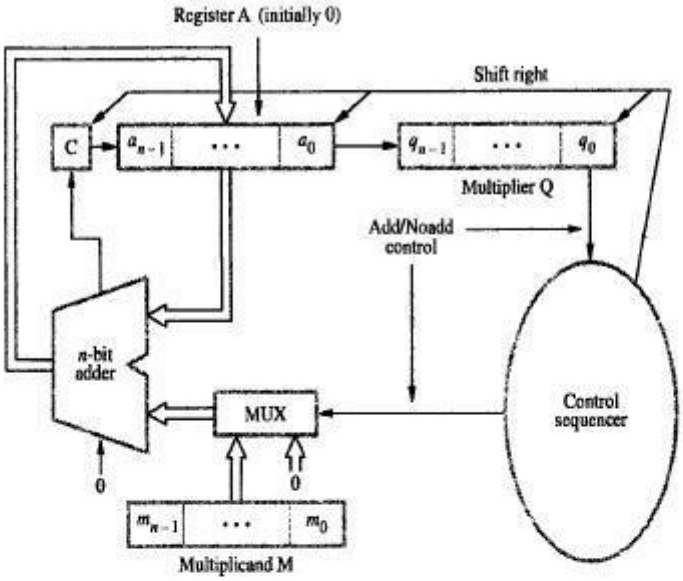
- Have a high gate count for multiplying numbers of practical size such as 32-bit or 64-bit numbers.
- Perform only one function, namely, unsigned integer product.

SEQUENTIAL CIRCUIT BINARY MULTIPLIER

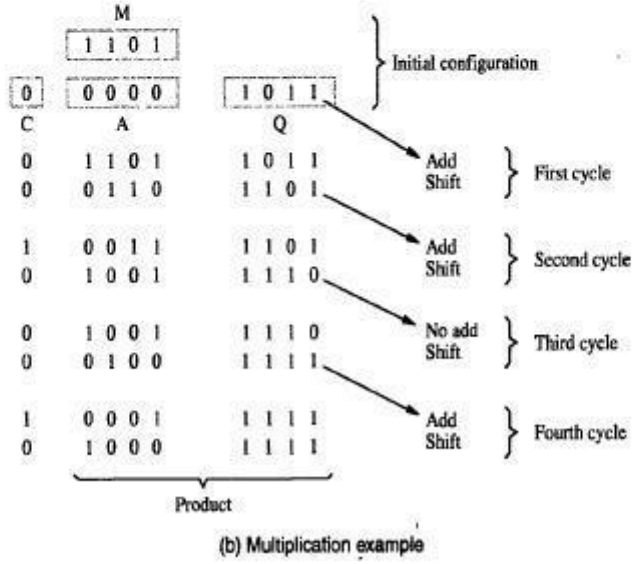
Q.5 Explain Sequential circuit for binary multiplication

• Registers A and Q combined hold PPi(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 6.7).
• Procedure for multiplication:

1) Multiplier is loaded into register Q, Multiplicand is loaded into register M and C & A are cleared to 0.
2) 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.
3) 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 6.7 Sequential circuit binary multiplier.

SIGNED OPERAND MULTIPLICATION
BOOTH ALGORITHM

Q.6 Explain Booth algorithm for binary multiplication. State its advantages and disadvantages.

• This algorithm
→ generates a 2n-bit product
→ treats both positive & negative 2's-complement n-bit operands uniformly(Figure 6.9-6.12).

• **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. Consider in a multiplication, the multiplier is positive 0011110, how many appropriately shifted versions of the multiplicand are added in a standard procedure? Add Four appropriately shifted versions of multiplicand
- The table in figure 6.12 is used for recoding multiplier and 6.10 shows an example..
- In general, in the Booth scheme, -1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left.
- If LSB of multiplier is 1 it is handled by assuming that an implied 0 lies to its right.
- Booth recoding of multiplier 0011110 is
  0 0 1 1 1 1 0 0
  0 +1 0 0 0 -1 0

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 6.12 Booth multiplier recoding table.

- 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 6.10 Booth recoding of a multiplier.

- Figure 6.9 shows and example of normal and booth multiplication for 45 and 30.

```
                    0 1 0 1 1 0 1
                    0 0+1+1+1+1 0
                    0 0 0 0 0 0 0 0
                  0 1 0 1 1 0 1
                0 1 0 1 1 0 1
              0 1 0 1 1 0 1
            0 1 0 1 1 0 1
          0 0 0 0 0 0 0
        0 0 0 0 0 0 0
```

```
  0 1 1 0 1  (+13)
x 1 1 0 1 0  (-6)
```

$\Longrightarrow$

```
          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)
```

Figure 6.11 Booth multiplication with a negative multiplier.


```
  0 1 1 0 1  (+13)
x 1 1 0 1 0  (-6)
```

$\Longrightarrow$

```
          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)
```

Figure 6.11 Booth multiplication with a negative multiplier.

FAST MULTIPLICATION
BIT-PAIR RECODING OF MULTIPLIERS
Q.7 Explain Fast Multiplication Method Bit pair recoding of multipliers

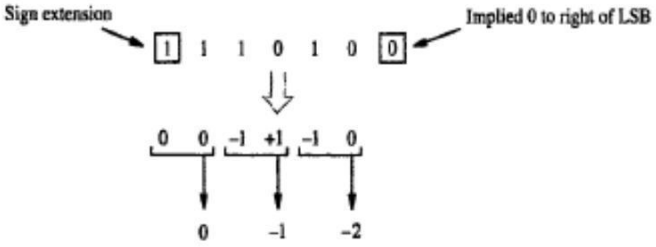
This method is

→ derived from the booth algorithm

→ reduces the number of summands by a factor of 2

Two methods used are:

1. Group the Booth-recoded multiplier bits in pairs refer to Figure 6.14a for bit pair recoding derived from Booth recoding.
   - The pair (+1 -1) is equivalent to the pair (0 +1), (-1,+1) equivalent to (0,-1),
   - (+1,0) equivalent to (0,+2) , (-1,0) equivalent to (0,-2), (0,0) equivalent to 0
2. Table of multiplicand selection decision figure 6.14b can be used for bit pair recoding.



(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 × M
0	0	1	+1 × M
0	1	0	+1 × M
0	1	1	+2 × M
1	0	0	−2 × M
1	0	1	−1 × M
1	1	0	−1 × M
1	1	1	0 × M

(b) Table of multiplicand selection decisions

Figure 6.14 Multiplier bit-pair recoding.

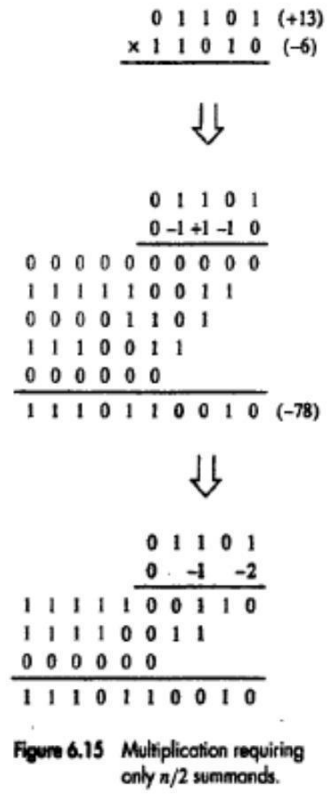
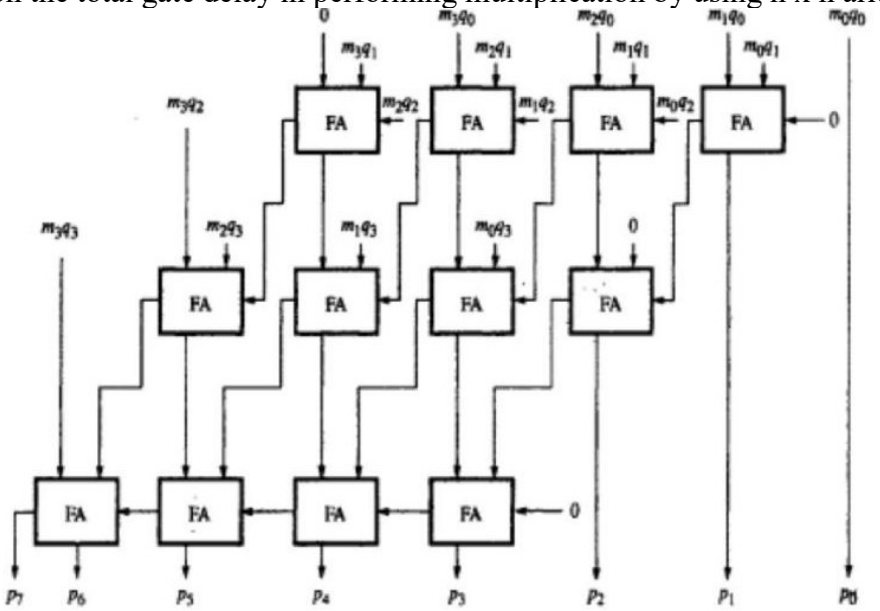


Figure 6.15 Multiplication requiring only $n/2$ summands.

CARRY-SAVE ADDITION OF SUMMANDS

Q. 8) Explain the concept of carry save addition for the multiplication operation MxQ=P for 4 bit operands with diagram and suitable example.

• Consider the array for 4*4 multiplication. (Figure 6.16 & 6.18). Needs 29 gate delays
• Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions.
• Consider the addition of many summands, we can:
• 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 Ripple Carry Adder or Carry LookAhead to produce the desired product.
• By comparison the total gate delay in performing multiplication by using n x n array is 6(n-1)-1=17



(b) Carry-save array

Figure 6.16 Ripple-carry and carry-save arrays for the multiplication operation M x Q = P for 4-bit operands.

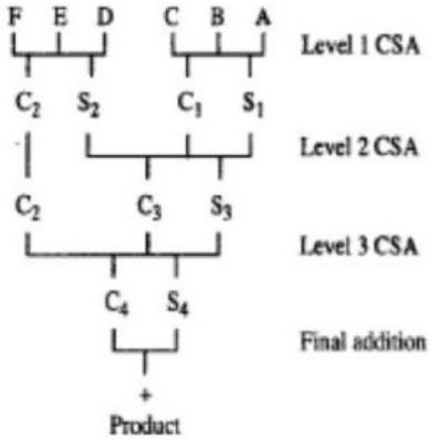


Figure 6.19 Schematic representation of the carry-save addition

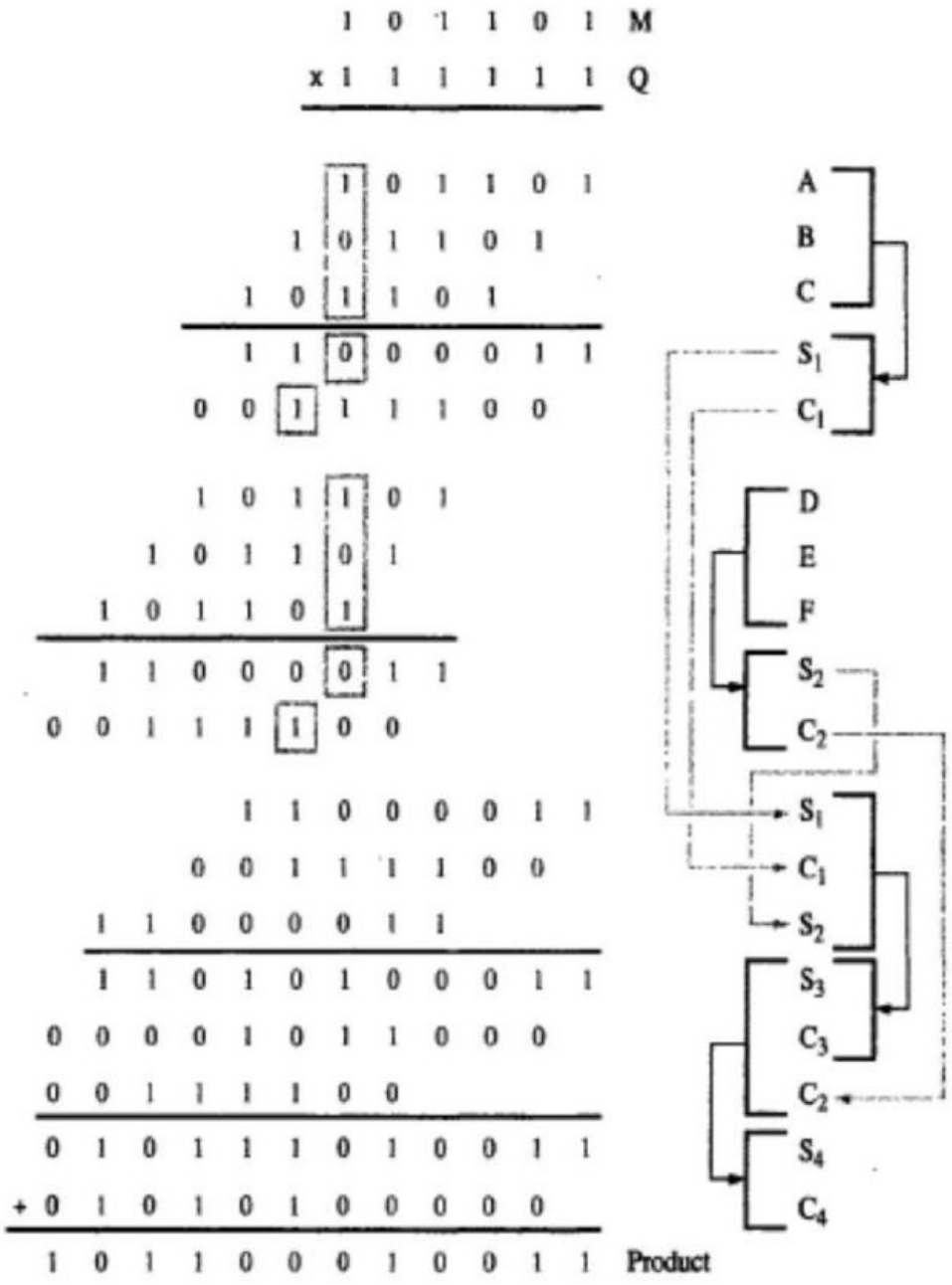


Figure 6.18 The multiplication example from Figure 6.17 performed using carry-save

INTEGER DIVISION

- **Longhand Division Steps:**
- Position the divisor appropriately with respect to the dividend and performs a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction.

Q. 9 Explain with figure circuit arrangement for binary division

- **Logic circuit arrangement for restoring 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 6.21).

- After division operation, the n-bit quotient is in register Q, and the remainder is in register A.

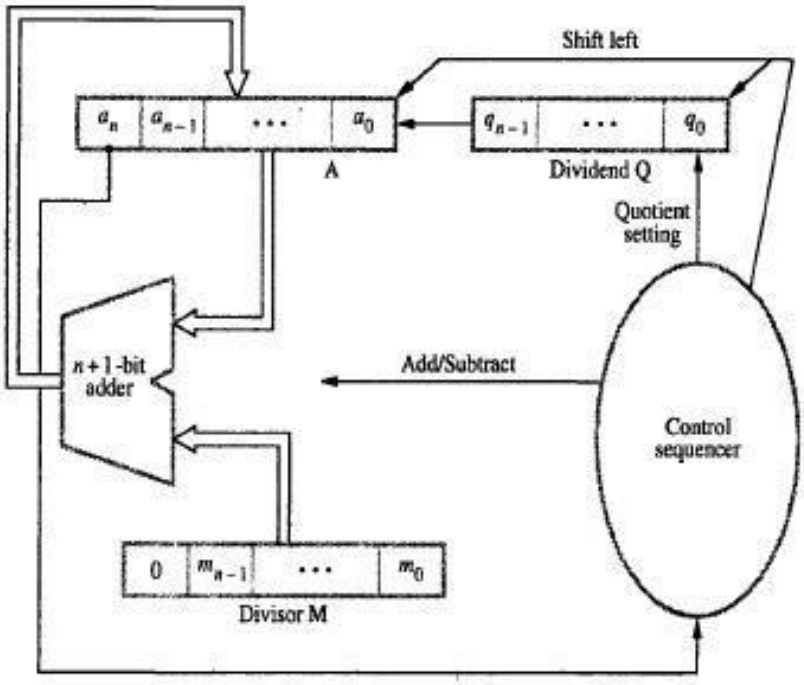


Figure 6.21 Circuit arrangement for binary division.

Q. 10 Give an algorithm and explain with an example both restoring and non-restoring methods for integer division.

RESTORING DIVISION

- **Procedure**: Do the following n times

1) Shift A and Q left one binary position (Figure 6.22).
2) Subtract M from A, and place the answer back in A.
3) If the sign of A is 1, set q0 to 0 and add M back to A(restore A). If the sign of A is 0, set q0 to 1 and no restoring done.

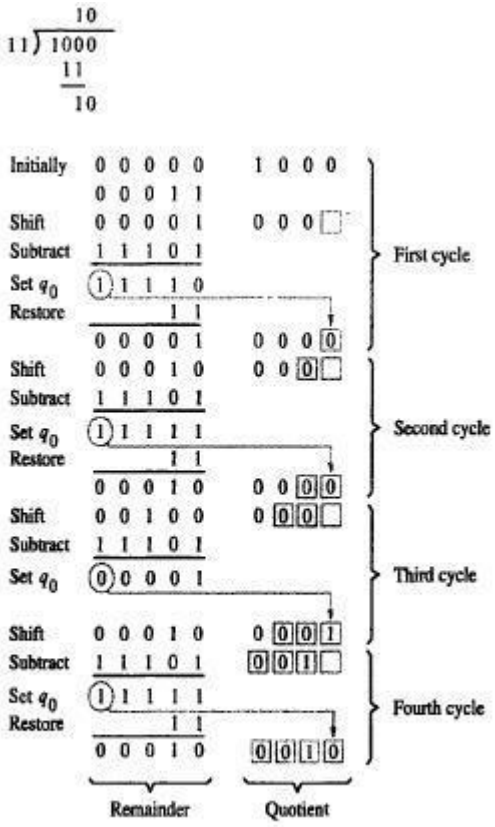


Figure 6.22 A restoring-division example.

-

c

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 A; otherwise, shift A and Q left and add M to A (Figure 6.23).

ii) Now, if the sign of A is 0, set q0 to 1; otherwise set q0 to 0.

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

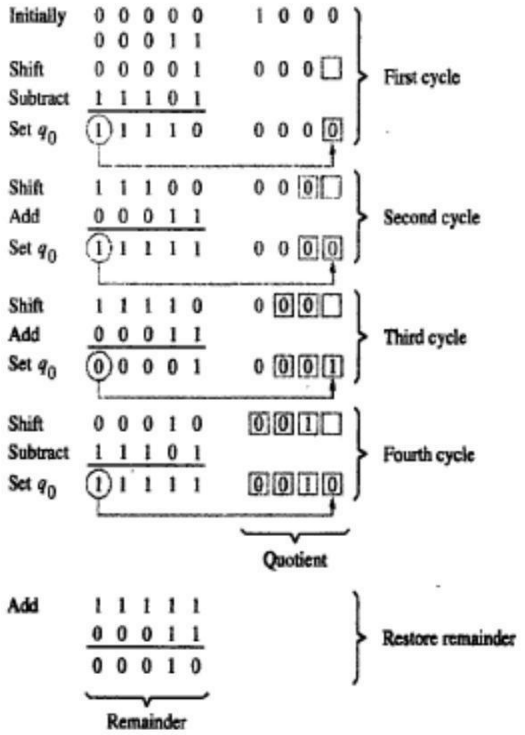


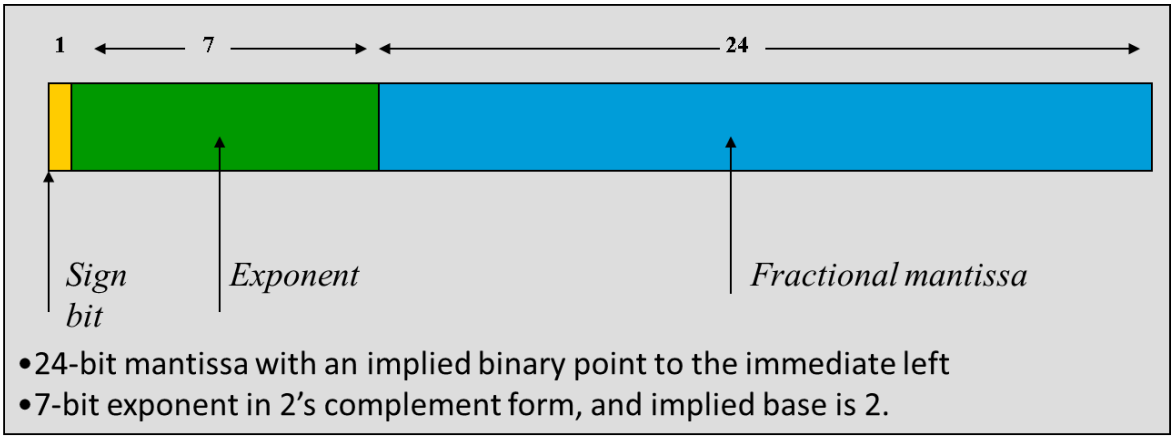
Figure 6.23 A nonrestoring-division example.

FLOATING-POINT NUMBERS & OPERATIONS

- In fixed point numbers, either the point is to the immediate right or it is to the immediate left. This is called Fixed point representation.
- Fixed point representation suffers from a drawback that these representation can only represent a finite range (and quite small) range of numbers.

A more convenient representation is the scientific representation, where the numbers are represented in the form:

$$+/- X_1.X_2X_3X_4X_5X_6X_7 * 10^{+/- Y1Y2}$$



Q.11 Explain Excess notation, Normalization and special conditions for exponents.

Excess Notation

- Rather than representing an negative exponent in 2's complement form, it turns out to be more beneficial to represent the exponent in excess notation.
- If 7 bits are allocated to the exponent, exponents can be represented in the range of -64 to +63, that is: **: -64<=e<=63**

Exponent can also be represented using the following coding called as excess-64:

$E' = E_{true} + 6$

In general, excess-p coding is represented as:

$E' = E_{true} + p$

True exponent of -64 is represented as 0
0 is represented as 64
63 is represented as 127

Normalization

Consider the number:

$$x = 0.0004056781 \times 10^{12}$$

If the number is to be represented using only 7 significant mantissa digits, the representation ignoring rounding is:

$$x = 0.0004056 \times 10^{12}$$

If the number is shifted so that as many significant digits are brought into 7 available slots:

$$x = 0.4056781 \times 10^{9} = 0.0004056 \times 10^{12}$$

Same methodology holds in the case of binary mantissas

$$0001101000(10110) \times 2^{8} = 1101000101(10) \times 2^{5}$$

Exponent of x was decreased by 1 for every left shift of x.
A number which is brought into a form so that all of the available mantissa digits are optimally used (this is different from all occupied which may not hold), is called a normalized number.

- In a base-2 representation, this implies that the MSB of the mantissa is always equal to 1.
- If every number is normalized, then the MSB of the mantissa is always 1. We can do away without storing the MSB.
- IEEE notation assumes that all numbers are normalized so that the MSB of the mantissa is a 1 and does not store this bit.
- The values of the numbers represented in the IEEE single precision notation are of the form:

$$(+,-)\ 1.M \times 2^{(E - 127)}$$

  - The hidden 1 forms the integer part of the mantissa.

Some special conditions:

1. When E'=0, then Mantissa M=0 and value exact 0 is represented.
2. When E'=255, M=0 then value ∞ is represented, where ∞ is result of dividing a number by 0.
3. When E'=0 and M≠0 denormal numbers are represented.
4. E'=255, M≠0 the value is called Not a Number (NaN).
5. E'<-126 Underflow condition.
6. E'> 127 Overflow condition

In such cases exceptions such as underflow, overflow, divide by zero, inexact and invalid are raised .

IEEE STANDARD FOR FLOATING POINT NUMBERS

Q.12 Explain IEEE standard for floating point numbers.

IEEE supports two representations:

1. 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
- 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 6.24).

2. 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.

In the IEEE representation, the exponent is in excess-127 (excess-1023)
notation.
The actual exponents represented are:

$$-126 <= E <= 127 \quad and \quad -1022 <= E <= 1023$$
$$not$$
$$-127 <= E <= 128 \quad and \quad -1023 <= E <= 1024$$

This is because the IEEE uses the exponents -127 and 128 (and -1023 and 1024), that is the actual values 0 and 255 to represent special conditions:

- Exact zero
- Infinity

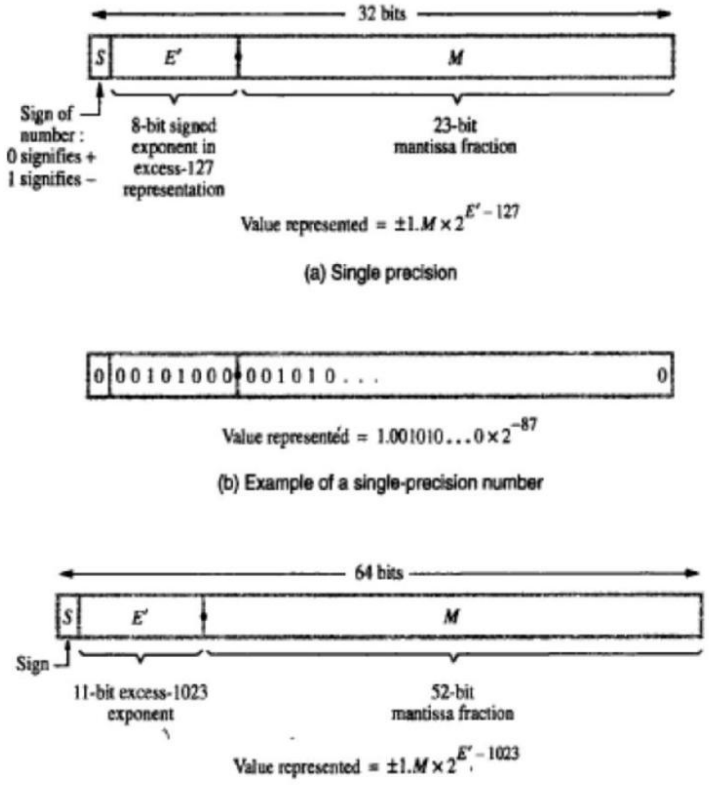


Figure 6.24 IEEE standard floating-point formats.

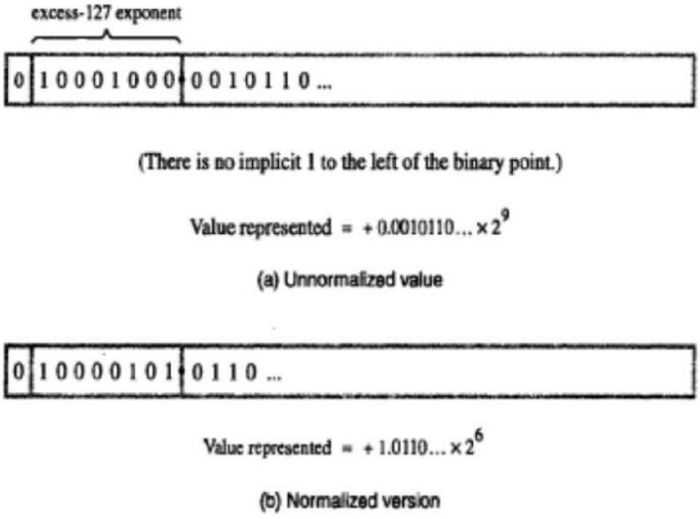


Figure 6.25 Floating-point normalization in IEEE single-precision format.

ARITHMETIC OPERATIONS ON FLOATING-POINT NUMBERS Multiply Rule

Multiplication:

$$3.1415 \times 10^8 \times 1.19 \times 10^6 = (3.1415 \times 1.19) \times 10^{(8+6)}$$

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

Division 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

Addition:

$$3.1415 \times 10^8 + 1.19 \times 10^6 = 3.1415 \times 10^8 + 0.0119 \times 10^8 = 3.1534 \times 10^8$$

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.

Guard Bits

While adding two floating point numbers with 24-bit mantissas, we shift the mantissa of the number with the smaller exponent to the right until the two exponents are equalized. This implies that mantissa bits may be lost during the right shift (that is, bits of precision may be shifted out of the mantissa being shifted).

To prevent this, floating point operations are implemented by keeping guard bits, that is, extra bits of precision at the least significant end of the mantissa.

The arithmetic on the mantissas is performed with these extra bits of precision.

After an arithmetic operation, the guarded mantissas are:

- Normalized (if necessary)
- Converted back by a process called truncation/rounding to a 24-bit mantissa.

Truncation/rounding

- **Straight chopping:**
  - The guard bits (excess bits of precision) are dropped.
- **Von Neumann rounding:**
  - If the guard bits are all 0, they are dropped.
  - However, if any bit of the guard bit is a 1, then the LSB of the retained bit is set to 1.
- **Rounding:**
  - If there is a 1 in the MSB of the guard bit then a 1 is added to the LSB of the retained bits.
  - Rounding is evidently the most accurate truncation method.

- However, Rounding requires an addition operation.
- Rounding may require a renormalization, if the addition operation de-normalizes the truncated number.
- IEEE uses the rounding method.
-

0.111111100000 rounds to 0.111111 + 0.000001 =1.000000 which must be renormalized to 0.100000

IMPLEMENTING FLOATING-POINT OPERATIONS

Q.13 With a neat diagram, explain the floating point addition/subtraction.

Step 1:

- Compare components to determine how far to shift the mantissa of the number with smaller exponent.
- The shift-count value, n, is determined by the 8 bit subtractor circuit. The magnitude of the difference $E'_A - E'_B$, or n, is sent to the SHIFTER unit.
- In step 1, the sign is sent to the swap network.
  - If sign=0, then $E'_A > E'_B$ and mantissas M_A & M_B are sent straight through SWAP network.
  - If sign=1, then $E'_A < E'_B$ and the mantissas are swapped before they are sent to SHIFTER.

Step 2: 2:1 MUX is used. The exponent of result E is tentatively determined as EA if EA>EB or EB if EA<EB

Step 3: Involves the mantissa adder/subtractor and CONTROL logic to determine whether the mantissas are to be added or subtracted. This is decided by the signs of the operands (S_A and S_B) and operation (Add or Subtract) that is to be performed on the operands.

- The CONTROL logic also determines the sign of the result, SR.
- For example, if A and B are both positive and the operation is A – B, then the mantissas are subtracted. The sign of the result, SR.
- For instance, if $E'_A \rightarrow E'_B$, then M_A – (shifted M_B) is positive and the result is positive. But $E'_A > E'_B$, then M_B – (shifted M_A) is positive and the result is negative.

Step 4 : Normalizing the result of step 3, mantissa M. The number of leading zeros in M determines the number of bit shifts, X, to be applied to M.

- The normalized value is truncated to generate the 24-bit mantissa, M_R, of the result. The value X is also subtracted from the tentative result exponent E to generate true result exponent, E_R.

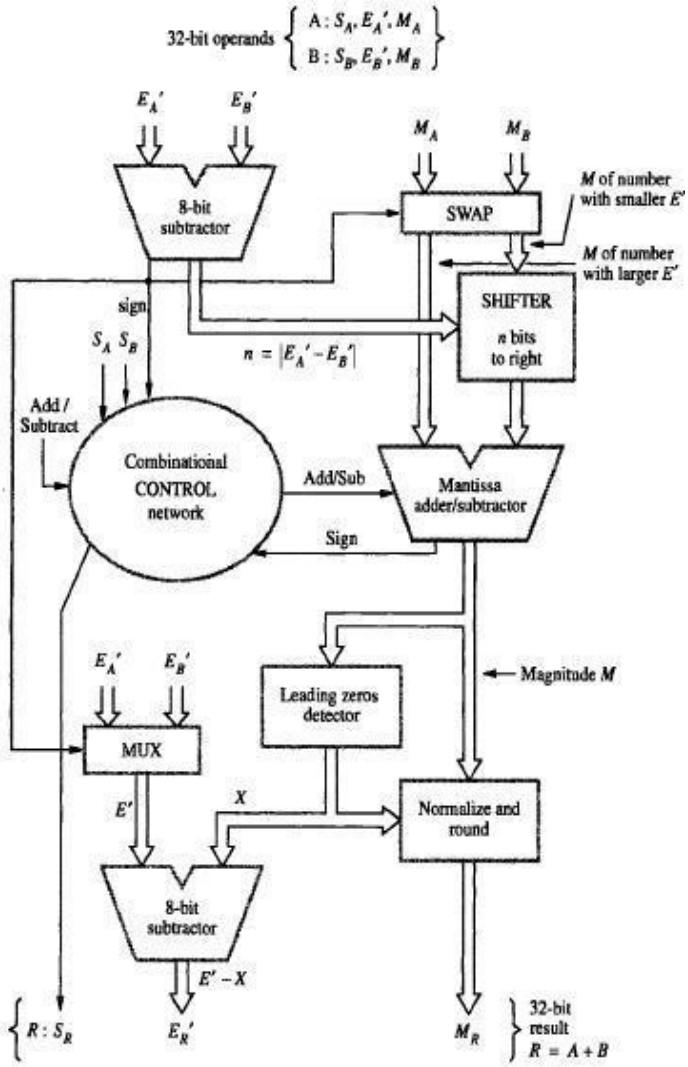


Figure 6.26 Floating-point addition-subtraction unit.