UNIT 3 – ARITHMETIC AND LOGIC UNIT

Introduction

- ALU is that part of the computer that actually performs arithmetic and logical operations on data
- All of the other elements of the computer system—control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out.
- An ALU and indeed, all electronic components in the computer, are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

ALU Inputs and Outputs

- Figure 3.1 indicates how the ALU is interconnected with the rest of the processor
- Operands for arithmetic and logic operations are presented to the ALU in registers, and the results of an operation are stored back in registers
- These registers are temporary storage locations within the processor that are connected by signal paths to the ALU
- The ALU may also set flags as the result of an operation. For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored.

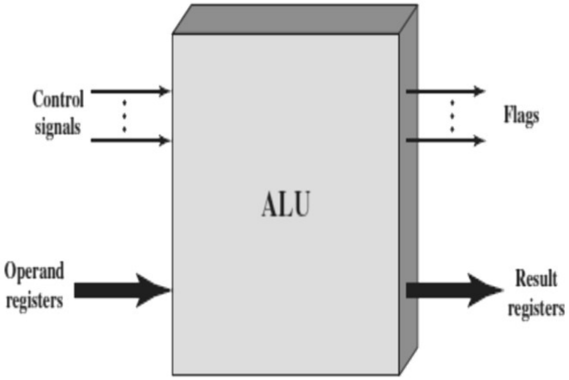


Fig 3.1 ALU Inputs and Outputs

Hardware for Addition and Subtraction

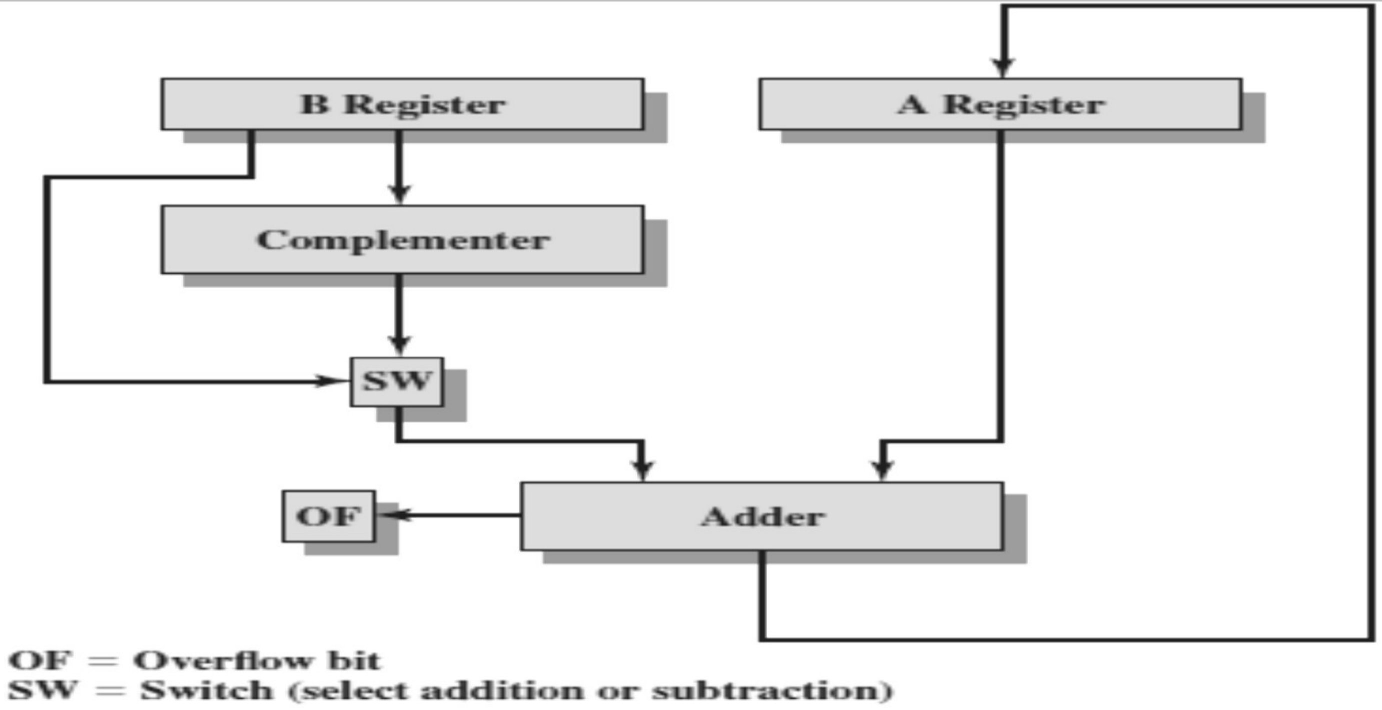


Fig 3.2 Block Diagram of Hardware for Addition and Subtraction

Hardware for Addition and Subtraction

- The central element is a binary adder, which is presented two numbers for addition and produces a sum and an overflow indication
- The binary adder treats the two numbers as unsigned integers
- For addition, the two numbers are presented to the adder from two registers, designated in this case as A and B registers
- The result may be stored in one of these registers or in a third
- The overflow indication is stored in a 1-bit overflow flag (0 = no overflow; 1 = overflow)
- For subtraction, the subtrahend (B register) is passed through a twos complementer so that

its twos complement is presented to the adder.

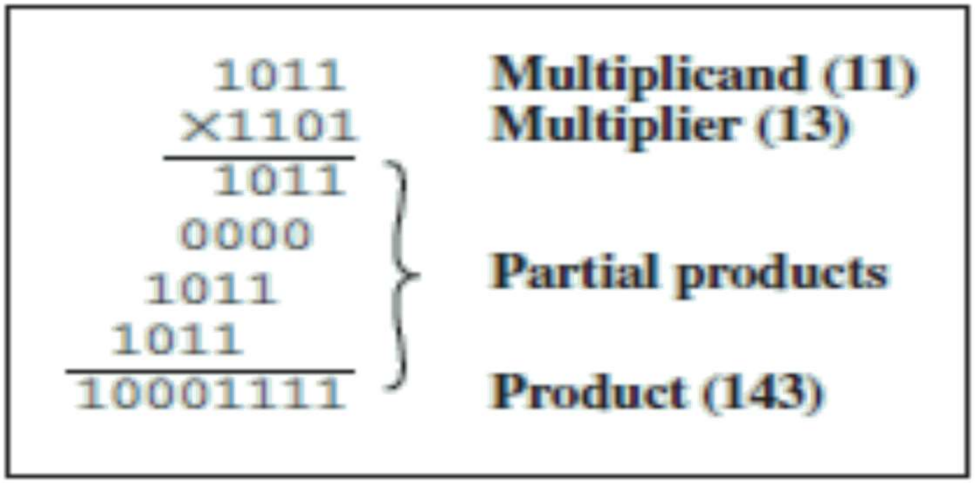
Multiplication

- Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software
- A wide variety of algorithms have been used in various computers
- We begin with simpler problem of multiplying two unsigned (nonnegative) integers, and then we look at one of the most common techniques for multiplication of numbers in twos complement representation

Multiplication of Unsigned Integers

- Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.
- The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
- The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
- The multiplication of two *n*-bit binary integers results in a product of up to 2*n* bits in length (e.g., 11 * 11 = 1001).

Multiplication of Unsigned Integers



```
    1011     Multiplicand (11)
   ×1101     Multiplier (13)
   -----
    1011  ⎫
   0000   ⎪
  1011    ⎬  Partial products
 1011     ⎪
--------  ⎭
10001111     Product (143)
```

Figure 3.3 Multiplication of Unsigned Integers

Example 2

Multiply 15 and 6

Hardware Implementation of Unsigned Binary Integer

- Compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient
- First, we can perform a running addition on the partial products rather than waiting until the end. This eliminates the need for storage of all the partial products; fewer registers are needed.
- Second, we can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required

Hardware Implementation of Unsigned Binary Integer

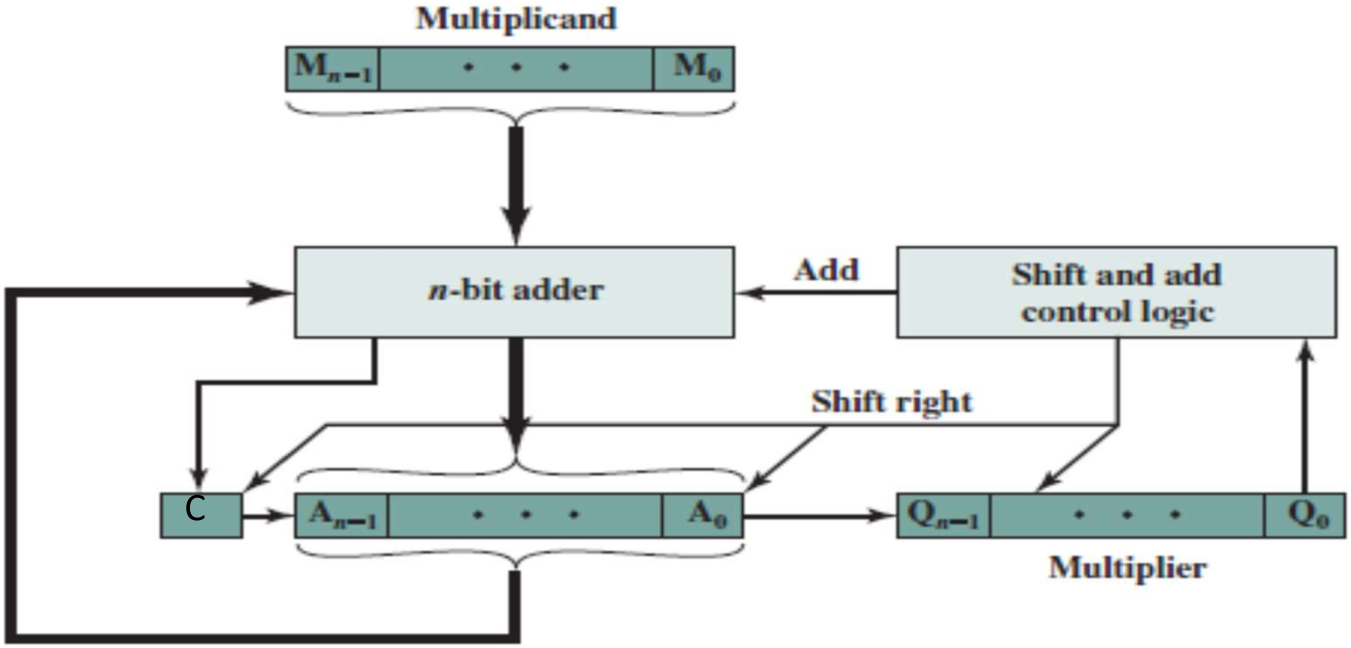
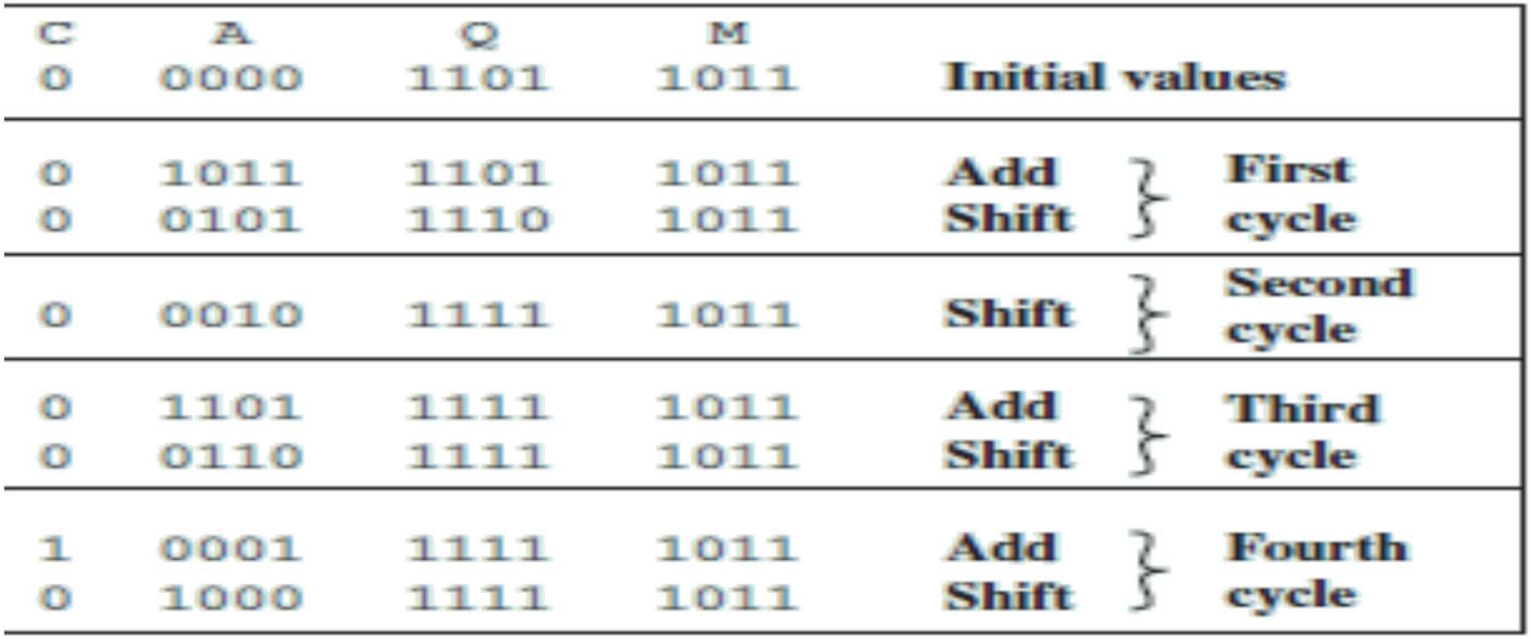


Figure 3.4 Block Diagram

Hardware Implementation of Unsigned Binary Integer



C	A	Q	M		
0	0000	1101	1011	Initial values	
0	1011	1101	1011	Add	First cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second cycle
0	1101	1111	1011	Add	Third cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth cycle
0	1000	1111	1011	Shift	

Figure 3.5 Example

Hardware Implementation of Unsigned Binary Integer

- The multiplier and multiplicand are loaded into two registers (Q and M)
- A third register, the A register, is also needed and is initially set to 0
- There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition

Operation of the Multiplier

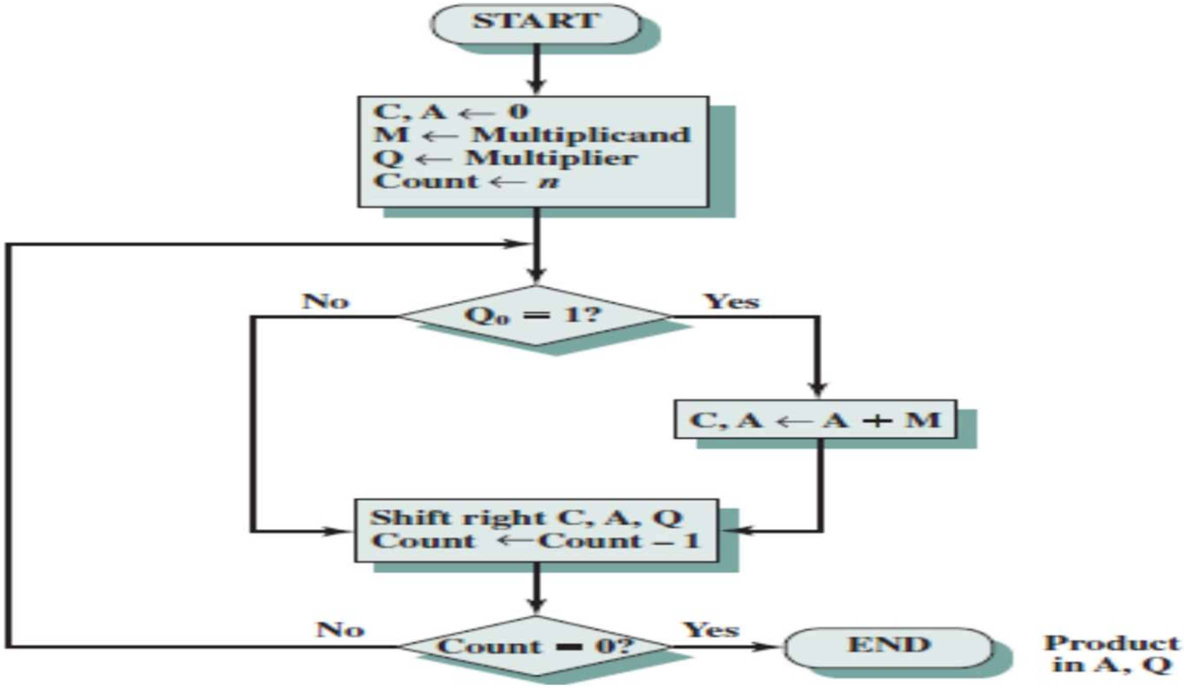
- Control logic reads the bits of the multiplier one at a time
- If Q0 is 1, then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow
- Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into An-1, A0 goes into Qn-1, and Q0 is lost
- If Q0 is 0, then no addition is performed, just the shift. This process is repeated for each bit of the original multiplier
- The resulting 2n-bit product is contained in the A and Q registers
- Note that on the second cycle, when the multiplier bit is 0, there is no add operation.

Example 2

Multiply 15 and 6

Operation of the Multiplier

Figure 3.6 Flowchart



2’s Complement Multiplication

- We have seen that addition and subtraction can be performed on numbers in twos complement notation by treating them as unsigned integers
- If these numbers are considered to be unsigned integers, then we are adding

    9 (1001) + 3 (0011) = 12 (1100)

- As twos complement integers, we are adding

    - 7(1001) + 3 (0011) = - 4(1100)

2’s Complement Multiplication

- Unfortunately, this simple scheme will not work for multiplication
- We multiplied 11 (1011) X 13 (1101) = 143(10001111)
- If we interpret these as twos complement numbers, we have

-5(1011) X - 3 (1101) = - 113 (10001111)

This example demonstrates that straightforward multiplication will not work if both the multiplicand and multiplier are negative

2’s Complement Multiplication

- Recall that any unsigned binary number can be expressed as a sum of powers of 2. Thus,

  $1101 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 2^3 + 2^2 + 2^0$

- Further, the multiplication of a binary number by 2^n is accomplished by shifting that number to the left n bits.

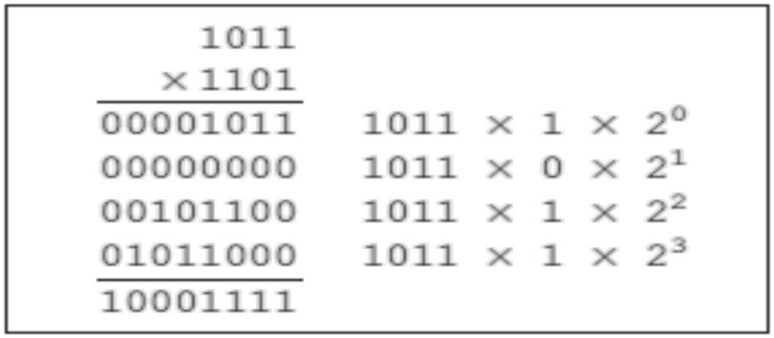
- This technique is used to make the generation of partial products by multiplication explicit

- The only difference is that it recognizes that the partial products should be viewed as 2n-bit numbers generated from the n-bit multiplicand

2’s Complement Multiplication

- Thus, as an unsigned integer, the 4-bit multiplicand 1011 is stored in an 8-bit word as 00001011

- Each partial product (other than that for 2^0) consists of this number shifted to the left, with the unoccupied positions on the right filled with zeros (e.g., a shift to the left of two places yields 00101100).



```
     1011
   × 1101
 --------
 00001011    1011 × 1 × 2^0
 00000000    1011 × 0 × 2^1
 00101100    1011 × 1 × 2^2
 01011000    1011 × 1 × 2^3
 --------
 10001111
```

Figure 3.7 Multiplication of Two Unsigned 4-bit Integers yielding an 8-bit result

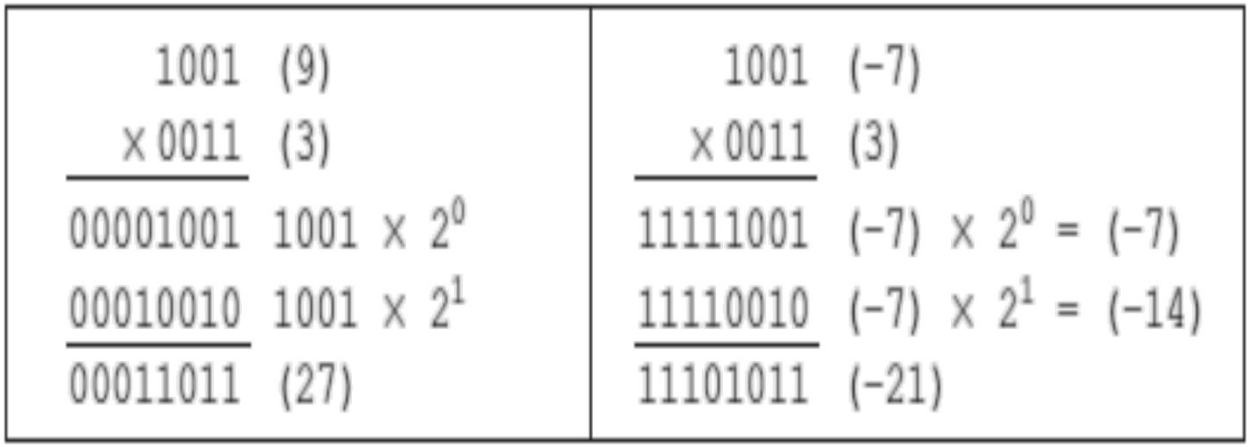
Example 2

Multiply 15 and 6

2’s Complement Multiplication

- Now we can demonstrate that straightforward multiplication will not work if the multiplicand is negative
- The problem is that each contribution of the negative multiplicand as a partial product must be a negative number on a 2n-bit field; the sign bits of the partial products must line up
- This is demonstrated in Figure 3.8, which shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of 9 * 3 = 27 proceeds simply
- However, if 1001 is interpreted as the twos complement value - 7, then each partial product must be a negative twos complement number of 2*n* (8) bits, as shown in Figure 3.8b
- Note that this is accomplished by padding out each partial product to the left with binary 1s.

2’s Complement Multiplication



1001 (9)	1001 (-7)
x 0011 (3)	x 0011 (3)
00001001 1001 x 2^0	11111001 (-7) x 2^0 = (-7)
00010010 1001 x 2^1	11110010 (-7) x 2^1 = (-14)
00011011 (27)	11101011 (-21)

Figure 3.8 Comparison of Multiplication of Unsigned and Twos Complement Integers

2’s Complement Multiplication

- If the multiplier is negative, straightforward multiplication also will not work
- The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place.

For example, the 4-bit decimal number - 3 is written 1101 in twos complement. If we simply took partial products based on each bit position, we would have the following correspondence:

$1101 = (1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0) = -(2^3 + 2^2 + 2^0)$

In fact, what is desired is $-(2^1 + 2^0)$. So this multiplier cannot be used directly in the manner we have been describing.

2’s Complement Multiplication

- Solution:
- First solution is both multiplier and multiplicand can be converted to positive numbers, perform the multiplication, and then take the twos complement of the result if and only if the sign of the two original numbers differed
- Implementers have preferred to use techniques that do not require this final transformation step
- Second solution is to use Booth’s algorithm
- This algorithm also has the benefit of speeding up the multiplication process, relative to a more straightforward approach

Booth’s Algorithm

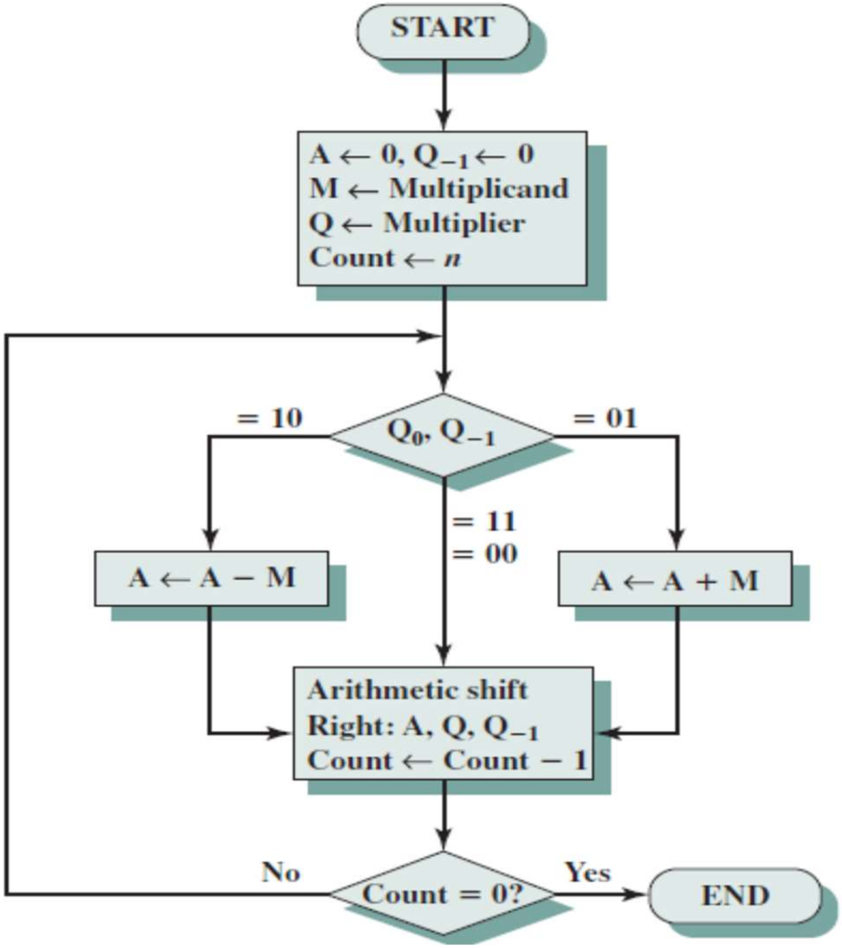


Figure 3.9 Booth’s Algorithm for 2’s Complement Multiplication

Booth’s Algorithm

1) The multiplier and multiplicand are placed in the Q and M registers, respectively
2) There is also a 1-bit register placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1}
3) The results of the multiplication will appear in the A and Q registers
4) Initially ,A and Q_{-1} are initialized to 0
5) The control logic scans the bits of the multiplier one at a time. Now, as each bit is examined, the bit to its right is also examined
6) If the two bits are the same (1–1 or 0–0), then all of the bits of the A, Q, and Q-1 registers are shifted to the right 1 bit.

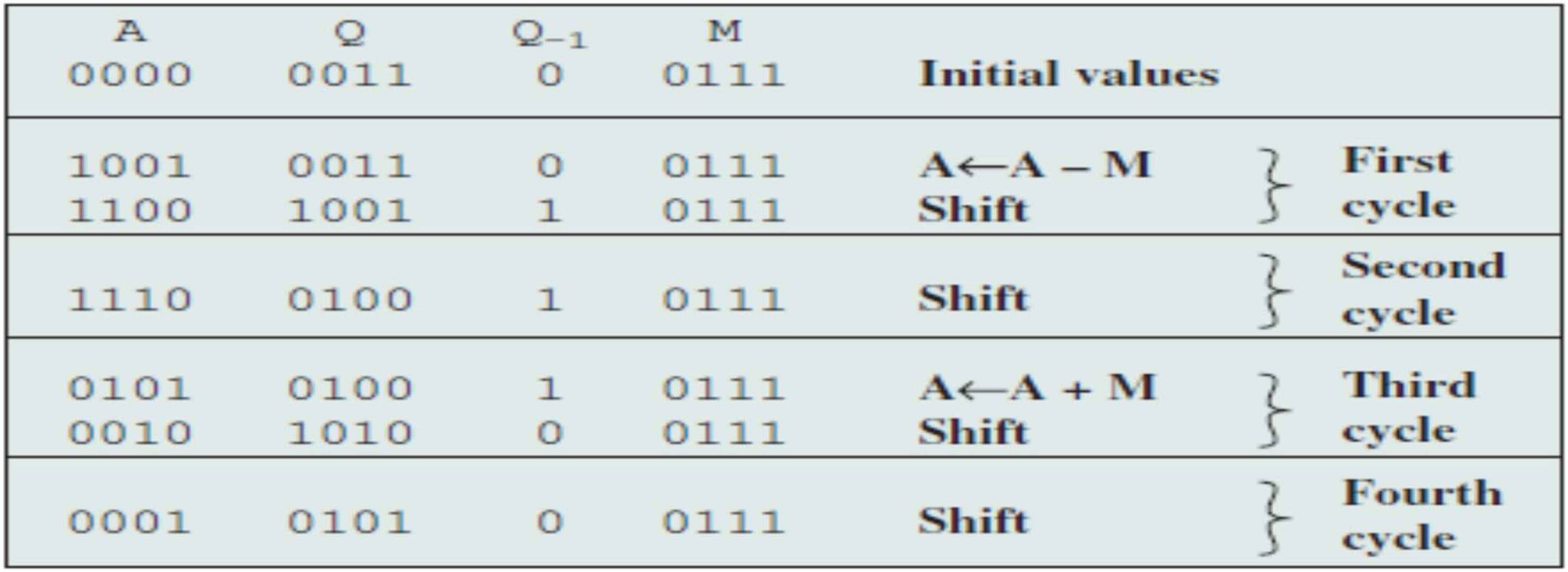
Booth’s Algorithm

7) If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits are 0–1 or 1–0

8) In either case, the right shift is such that the leftmost bit of A, namely A_{n-1}, not only is shifted into A_{n-2}, but also remains in A_{n-1}

9) This is required to preserve the sign of the number in A and Q. It is known as an arithmetic shift, because it preserves the sign bit

Booth’s Algorithm



A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A←A – M	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A←A + M	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle

Figure 3.10 Example of Booth’s Algorithm(7*3)

Booth’s Algorithm

For (-7)x(3)

Booth’s Algorithm

For 7x(-3)

Booth’s Algorithm

For (-7)x(-3)

Booth’s Algorithm

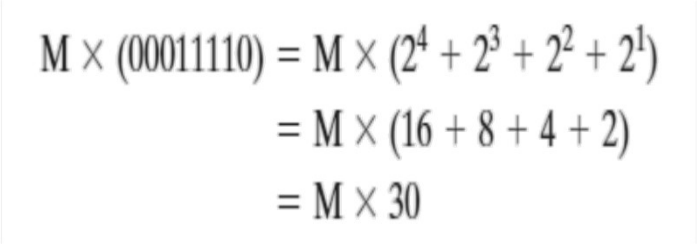
For (5)x(-4)

Booth’s Algorithm

For (-5)x(-4)

Why does Booth’s algorithm works?

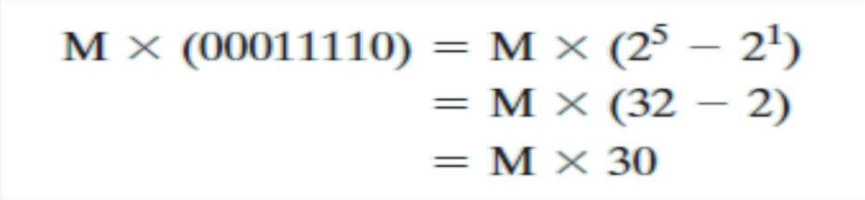
- Consider a positive multiplier consisting of one block of 1s surrounded by 0s (e.g., 00011110)
- As we know, multiplication can be achieved by adding appropriately shifted copies of the multiplicand:



$$\begin{aligned} M \times (00011110) &= M \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= M \times (16 + 8 + 4 + 2) \\ &= M \times 30 \end{aligned}$$

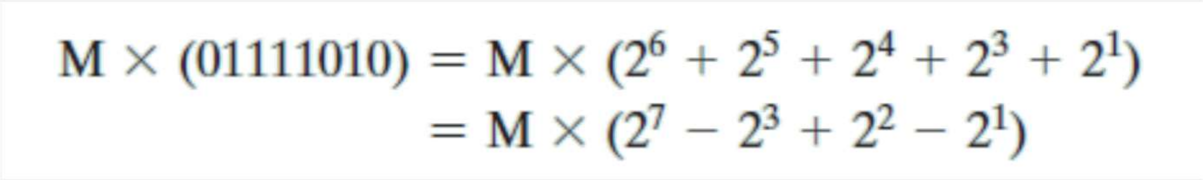
- The number of such operations can be reduced to two if we observe that $2^n + 2^{n-1} + \ldots + 2^{n-K} = 2^{n+1} - 2^{n-K}$

Why does Booth's algorithm works?



$$\begin{aligned} M \times (00011110) &= M \times (2^5 - 2^1) \\ &= M \times (32 - 2) \\ &= M \times 30 \end{aligned}$$

- So the product can be generated by one addition and one subtraction of the multiplicand
- This scheme extends to any number of blocks of 1s in a multiplier, including the case in which a single 1 is treated as a block.



$$\begin{aligned} M \times (01111010) &= M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= M \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

Why does Booth’s algorithm works?

- Booth’s algorithm conforms to this scheme by performing a subtraction when the first 1 of the block is encountered (1–0) and an addition when the end of the block is encountered (0–1)
- The same scheme works for negative multiplier also

Division

- First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number
- Until this event occurs, 0s are placed in the quotient from left to right
- When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a partial remainder
- From this point on, the division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor
- As before, the divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted

Unsigned Binary Division

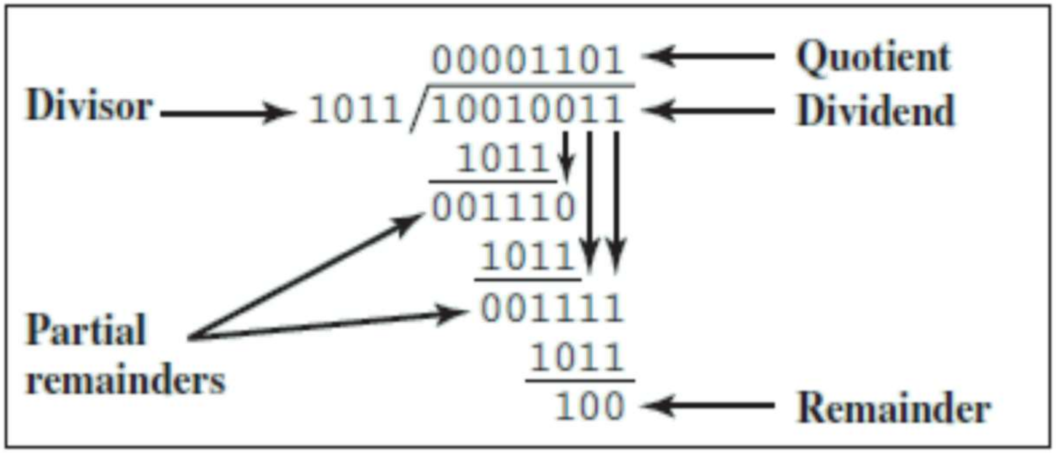
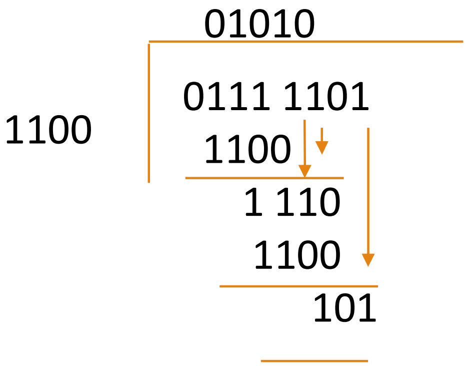


Figure 3.11 Example of Division of Unsigned Binary Integers

Unsigned Binary Division

125/12



01010

1100

0111 1101

1100

1 110

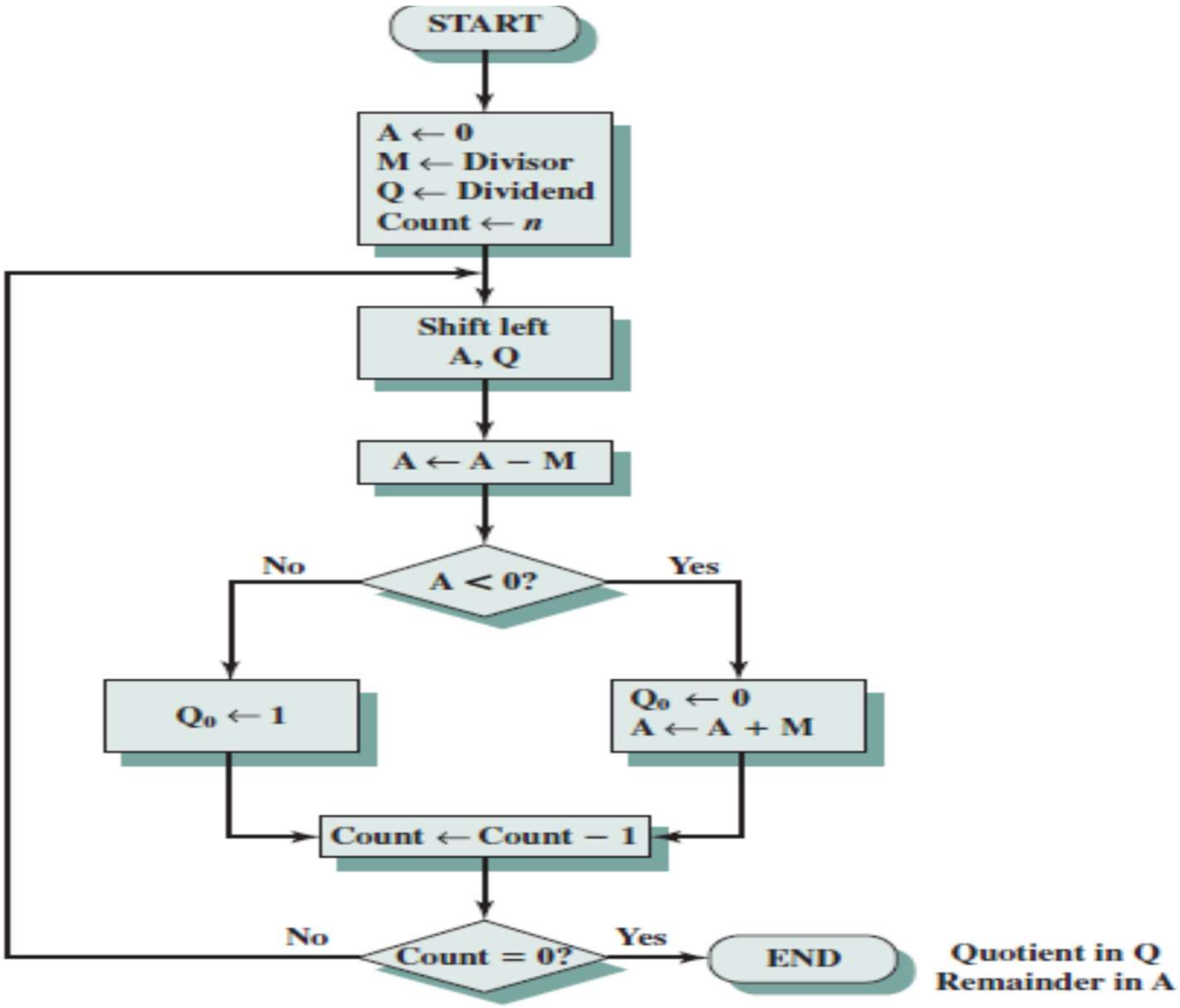
1100

101

Example of Division of Unsigned Binary Integers

Unsigned Binary Division

Figure 3.12 Flow Chart of unsigned binary division



Unsigned Binary Division

- Figure 3.12 shows a machine algorithm that corresponds to the long division process
- The divisor is placed in the M register, the dividend in the Q register
- At each step, the A and Q registers together are shifted to the left 1 bit
- M is subtracted from A to determine whether A divides the partial remainder
- If it does, then Q0 gets a 1 bit. Otherwise, Q0 gets a 0 bit and M must be added back to A to restore the previous value
- The count is then decremented, and the process continues for *n* steps
- At the end, the quotient is in the Q register and the remainder is in the A register.

Unsigned Binary Division

Example: Dividend=11 and Divisor=3

n	M	A	Q	Operation
4	0011	0000	1011	initialize
	0011	0001	011_	shift left AQ
	0011	1110	011_	A=A-M
	0011	0001	0110	Q_0=0 And restore A
3	0011	0010	110_	shift left AQ
	0011	1111	110_	A=A-M
	0011	0010	1100	Q_0=0 And restore A

Unsigned Binary Division

n	M	A	Q	Operation
2	0011	0101	100_	shift left AQ
	0011	0010	100_	A=A-M
	0011	0010	1001	Q_0=1
1	0011	0101	001_	shift left AQ
	0011	0010	001_	A=A-M
	0011	0010	0011	Q_0=1

Unsigned Binary Division

Example: Dividend=14 and Divisor=4

n	M	A	Q	Operation
4	0100	0000	1110	initialize
	0100	0001	111_	shift left AQ
	0100	1101	111_	A=A-M
	0100	0001	1110	Q_0=0 And restore A
3	0100	0011	110_	shift left AQ
	0100	1111	110_	A=A-M
	0100	0011	1100	Q_0=0 And restore A

Unsigned Binary Division

n	M	A	Q	Operation
2	0100	0111	100_	shift left AQ
	0100	0011	100_	A=A-M
	0100	0011	1001	Q_0=1
1	0100	0111	001_	shift left AQ
	0100	0011	001_	A=A-M
	0100	0011	0011	Q_0=1

Signed Binary Division

- This process can, with some difficulty, be extended to negative numbers
- We give here one approach for twos complement numbers as shown in example in previous slide.
- The algorithm assumes that the divisor V and the dividend D are positive and that

  |V|< |D|
- If |V| = |D|, then the quotient Q = 1 and the remainder R = 0
- If |V|>|D|, then Q = 0 and R = D.

Twos Complement Division Algorithm

1. Load the twos complement of the divisor into the M register; that is, the M register

contains the negative of the divisor

Load the dividend into the A, Q registers

The dividend must be expressed as a 2n-bit positive number. Thus, for example, the 4-bit 0111 becomes 00000111.

2. Shift A, Q left 1 bit position

3. Perform A $\leftarrow$A - M. This operation subtracts the divisor from the contents of A.

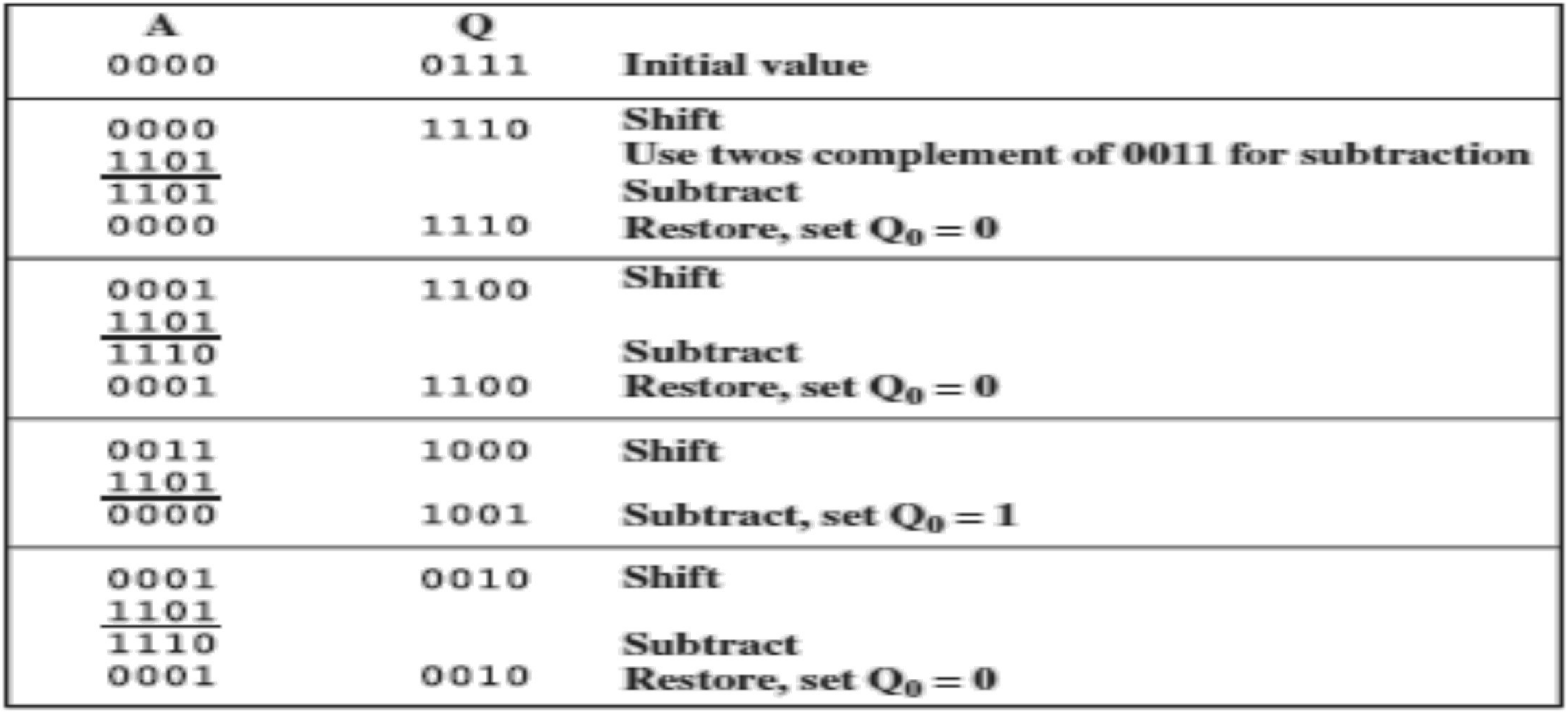
4. a. If the result is nonnegative (most significant bit of A = 0), then set $Q_0 \leftarrow 1$.

Twos Complement Division Algorithm

b. If the result is negative (most significant bit of A = 1), then set $Q_0 \leftarrow 0$ and restore the previous value of A.

5. Repeat steps 2 through 4 as many times as there are bit positions in Q.

6. The remainder is in A and the quotient is in Q.



A	Q	
0000	0111	Initial value
0000	1110	Shift
1101		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
1101		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

Figure 3.13 Example of Restoring Twos Complement Division (7/3)

Twos Complement Division Algorithm

- To deal with negative numbers, we recognize that the remainder is defined by

  D = *Q* * *V* + *R*, *t*hat is, the remainder is the value of *R* needed for the preceding equation to be valid

- Consider the following examples of integer division with all possible combinations

  of signs of *D* and *V*:

  D = 7 V = 3 → Q = 2 R = 1

  D = 7 V = -3 → Q = -2 R = 1

  D = -7 V = 3 → Q = -2 R = -1

  D = -7 V = -3 → Q = 2 R = -1

Twos Complement Division Algorithm

- Note that (- 7)/(3) and (7)/(- 3) produce different remainders
- We see that the magnitudes of Q and R are unaffected by the input signs and that the signs of Q and R are easily derived from the signs of D and V
- Specifically
  - sign(R) = sign(D) and
  - sign(Q) = sign(D) * sign(V)
- Hence, one way to do twos complement division is to convert the operands into unsigned values and, at the end, to account for the signs by complementation where needed.

Fixed Point Number

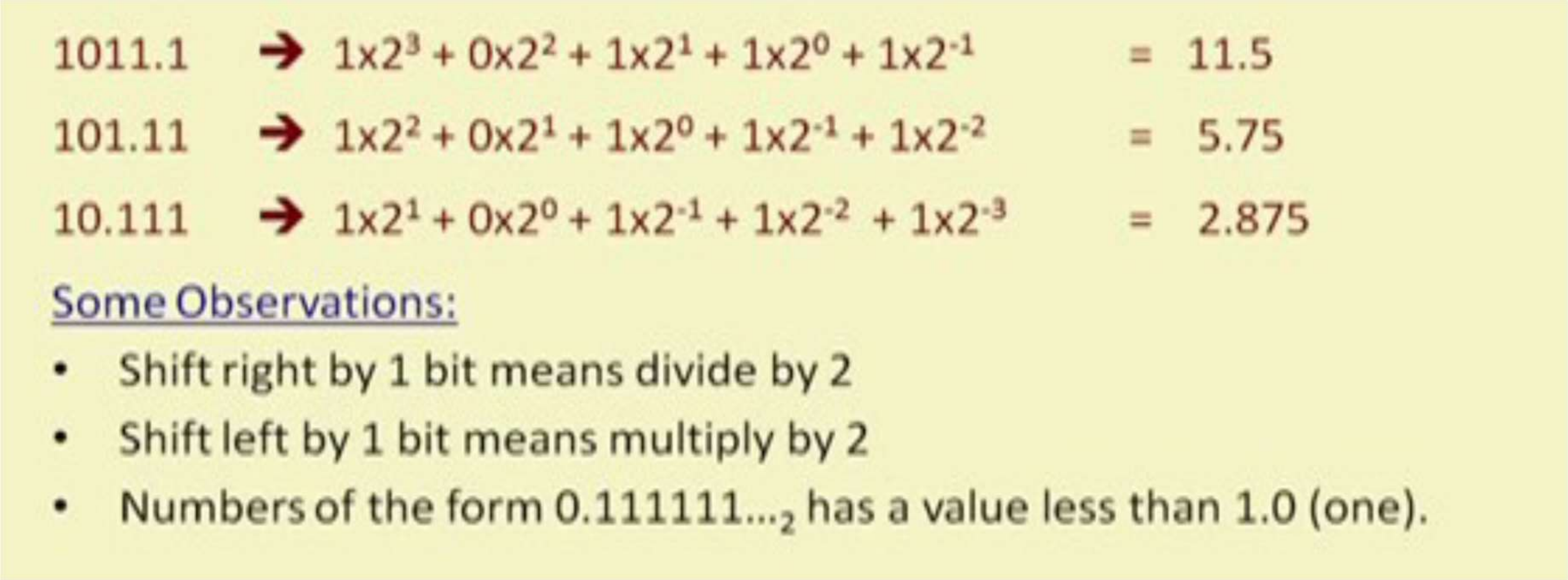
- A binary number with fractional part corresponds to the decimal number
- A binary number with fractional part

  B=b_{n-1},b_{n-2}…….b_1b_0.$b_{-1}b_{-2}$……..b_{-m} corresponds to the decimal D=$\sum_{i=-m}^{n-1} b_i\, 2^i$

- Also called fixed-point numbers
  - The position of the radix point is fixed
- If the radix point is allowed to move, then it is a floating-point number
- With a fixed-point notation it is possible to represent a range of positive and negative integers centered on or near 0

Fixed Point Number

- Some Examples



$1011.1 \rightarrow 1x2^{3} + 0x2^{2} + 1x2^{1} + 1x2^{0} + 1x2^{-1} = 11.5$

$101.11 \rightarrow 1x2^{2} + 0x2^{1} + 1x2^{0} + 1x2^{-1} + 1x2^{-2} = 5.75$

$10.111 \rightarrow 1x2^{1} + 0x2^{0} + 1x2^{-1} + 1x2^{-2} + 1x2^{-3} = 2.875$

Some Observations:

- Shift right by 1 bit means divide by 2
- Shift left by 1 bit means multiply by 2
- Numbers of the form $0.111111..._2$ has a value less than 1.0 (one).

Limitations of Representation

- In the fractional part, we can only represent numbers Of the form x/ 2^k exactly
- Other numbers have repeating bit representations (i.e. never converge)
- Examples:
  - ¾=0.11
  - 7/8 =0.111
  - 5/8 =0.101
  - 1/3 = 0.10101010101 [01]….
  - 1/5 = 0.001100110011 [0011]…..
  - 1/10 = 0.0001100110011 [0011]…
- More the number of bits, more accurate is the representation.

Floating Point Number

- This approach has limitations
  - Very large numbers cannot be represented, nor can very small fractions
  - Lacks flexibility
  - Furthermore, the fractional part of the quotient in a division of two large numbers could be lost.
- Solution: using Floating Point representation
- For decimal numbers, we get around this limitation by using scientific notation
  - 976,000,000,000,000 can be represented as $9.76 * 10^{14}$
  - 0.0000000000000976 can be represented as $9.76 * 10^{-14}$
  - What we have done, in effect, is dynamically to slide the decimal point to a convenient location and use the exponent of 10 to keep track of that decimal point

IEEE Floating Point Representation

- The IEEE (Institute of Electrical and Electronic Engineers) is an international organization that has designed specific binary formats for storing floating point numbers.
- The IEEE defines two different formats with different precisions: single and double precision
- Single precision is used by float variables and double precision is used by double variables.

IEEE Floating Point Representation

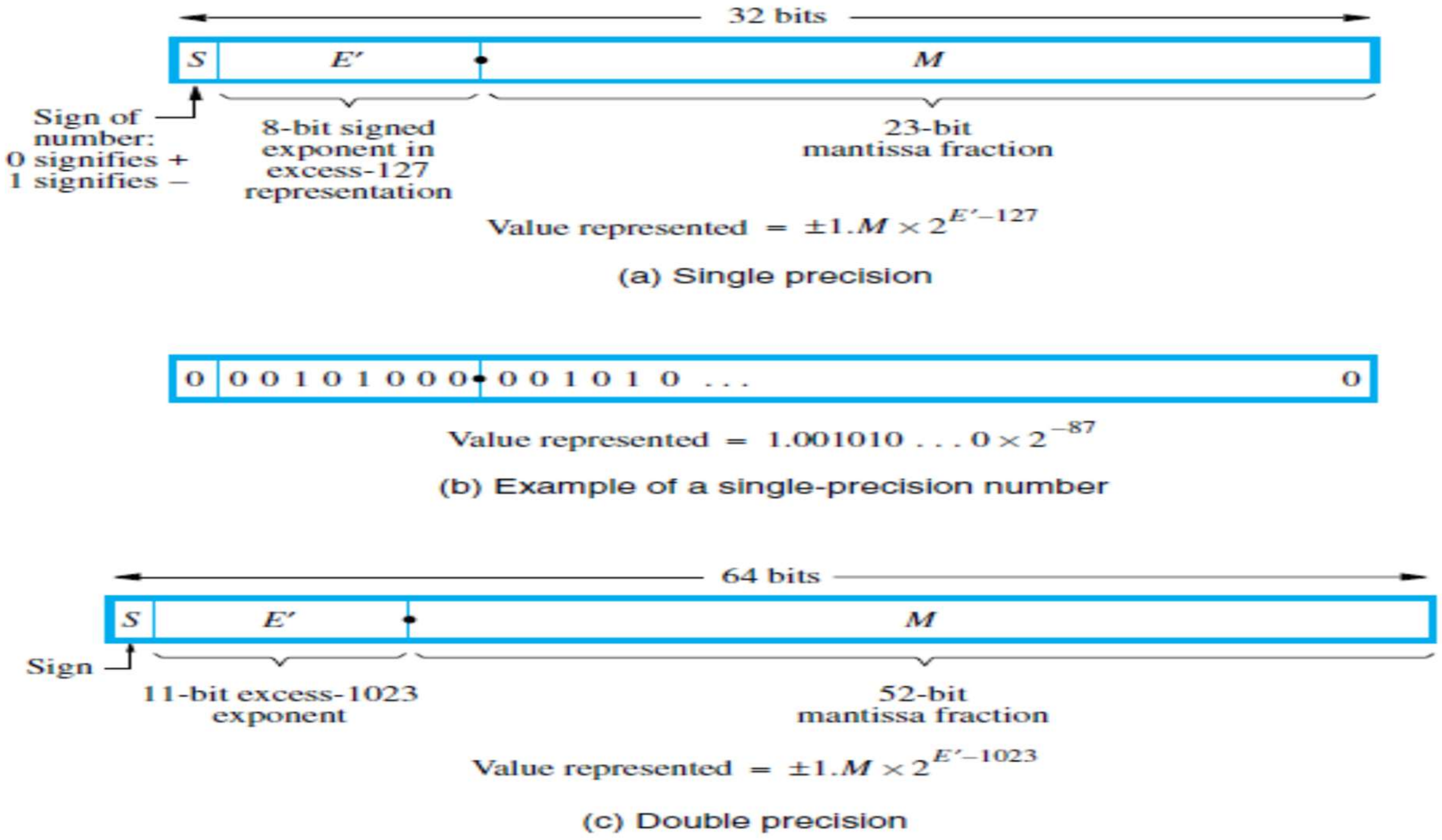
- We can represent the number in the form:

  $\pm MxB^{\pm E}$

- This number can be stored in a binary word with three fields:
  - Sign: plus or minus indicating whether the number is positive or negative
  - Mantissa M or Significand S
  - Exponent E which weights the number by power of 2

- The base B is implicit and need not be stored because it is the same for all numbers

IEEE Floating Point Representation


32 bits
S
E′
M
Sign of number: 0 signifies + 1 signifies −
8-bit signed exponent in excess-127 representation
23-bit mantissa fraction
0
0 0 1 0 1 0 0 0
0 0 1 0 1 0 . . .
0
64 bits
S
E′
M
Sign
11-bit excess-1023 exponent
52-bit mantissa fraction

Value represented $= \pm 1.M \times 2^{E'-127}$

(a) Single precision

Value represented $= 1.001010 \ldots 0 \times 2^{-87}$

(b) Example of a single-precision number

Value represented $= \pm 1.M \times 2^{E'-1023}$

(c) Double precision

IEEE Floating Point Representation

- The full 24-bit string, *B*, of significant bits, called the *mantissa*, always has a leading 1, with the binary point immediately to its right
- Therefore, the mantissa

$B = 1.M = 1.b_{-1}b_{-2} \ldots b_{-23}$ has the value

$V(B) = 1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-23} \times 2^{-23}$

- By convention, when the binary point is placed to the right of the first significant bit, the number is said to be *normalized*
- Note that the base, 2, of the scale factor and the leading 1 of the mantissa are both fixed
- They do not need to appear explicitly in the representation

IEEE Floating Point Representation

- The number of significant digits depends on the number of bits in M
  - 7 significant digits for 24bit mantissa (23 bits + 1 implied bit).
- The range of the number depends on the number of bits in E.
  - 10^{-38} to 10^{38} for 8-bit exponent.
- Normalized Representation-
- Assume that the actual exponent of the number is EXP (i.e. number is M x 2^{EXP})
- Permissible range of E’: 1 ≤ E’ ≤254 (the all-0 and all-1 patterns are not allowed)
- Encoding of the E:
  - The exponent is encoded as a biased value: E’ =EXP + BIAS

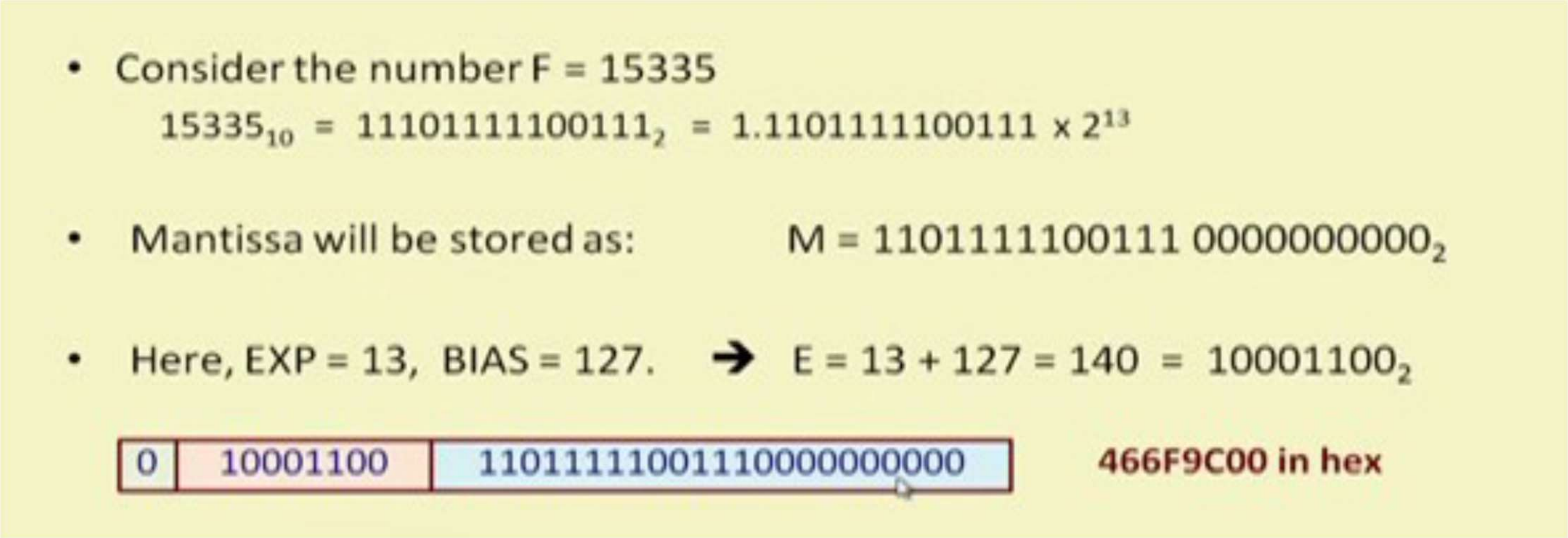
    where BIAS is 127 (2^{8-1} — 1) for single-precision, and BIAS is 1023 (2^{11-1} — 1) for double-precision.

IEEE Floating Point Representation

- Encoding Of the mantissa M:

- The mantissa is coded with an implied leading 1 (i.e. in 24 bits).
  - M 1 .xxxx...x — Here, xxxx...x denotes the bits that are actually stored for the mantissa
  - We get the extra leading bit for free
  - When xxxx...x = 0000...0, M is minimum (=1.0).
  - When xxxx...x = 11111...1, M is maximum (=2.0-$^{\varepsilon}$).

IEEE Floating Point Representation



- Consider the number F = 15335

  $15335_{10} = 11101111100111_2 = 1.1101111100111 \times 2^{13}$

- Mantissa will be stored as: $M = 1101111100111\ 0000000000_2$

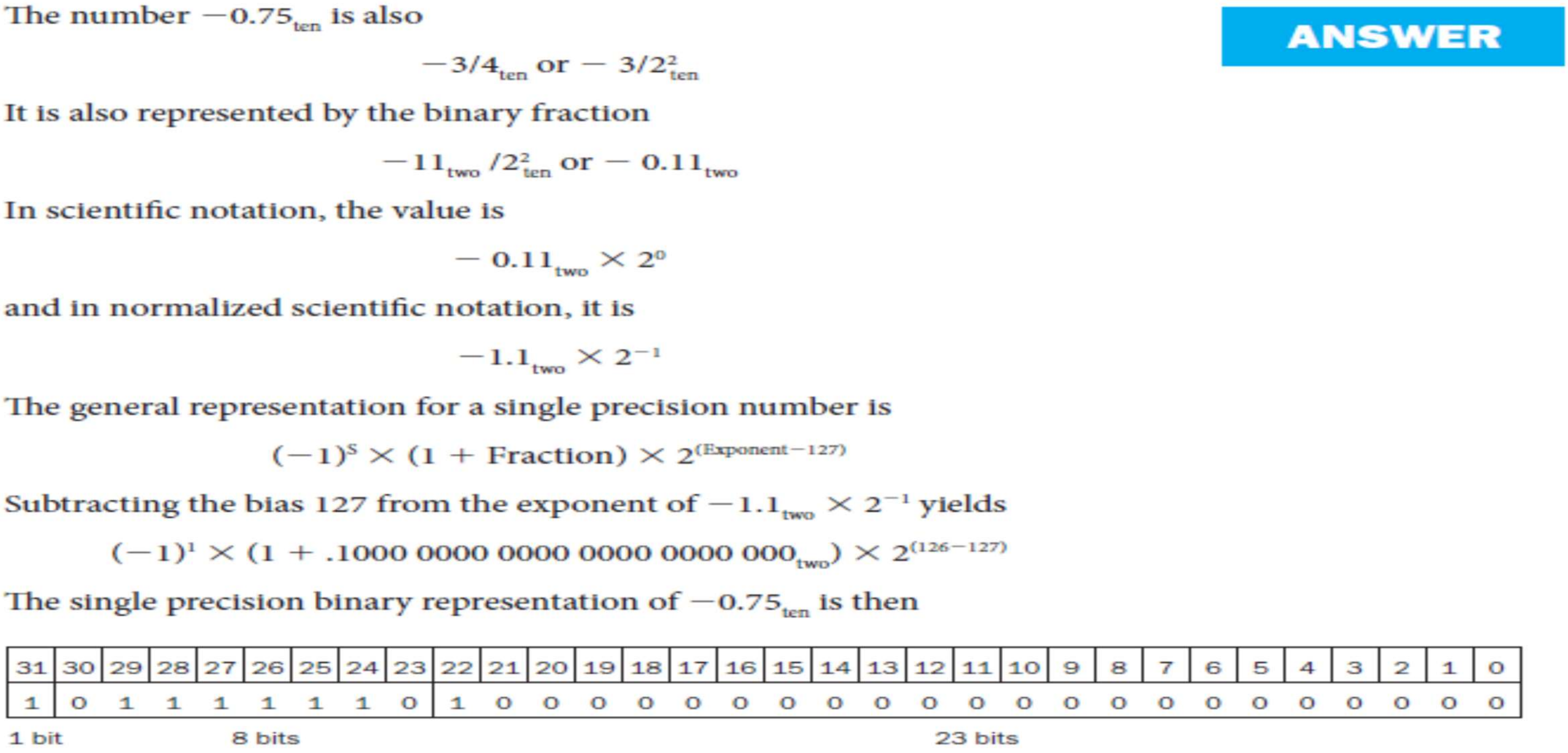
- Here, EXP = 13, BIAS = 127. → $E = 13 + 127 = 140 = 10001100_2$

0	10001100	11011111001110000000000

466F9C00 in hex

Show the IEEE 754 binary representation of the number -0.75$_{ten}$ in single and double precision.

ANSWER



The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2_{ten}$$

It is also represented by the binary fraction

$$-11_{two}/2^2_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

Subtracting the bias 127 from the exponent of $-1.1_{two} \times 2^{-1}$ yields

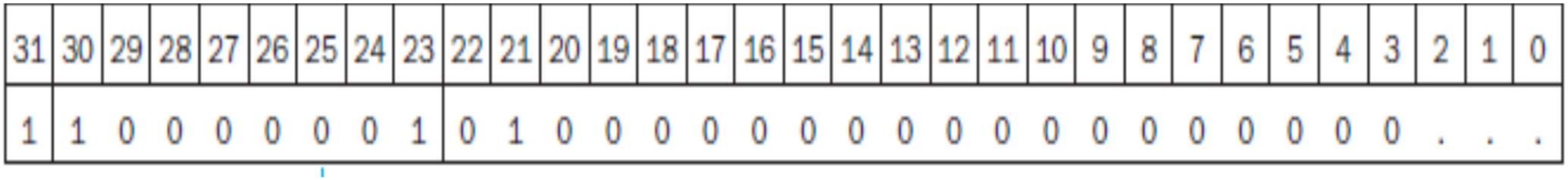
$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126-127)}$$

The single precision binary representation of -0.75_{ten} is then

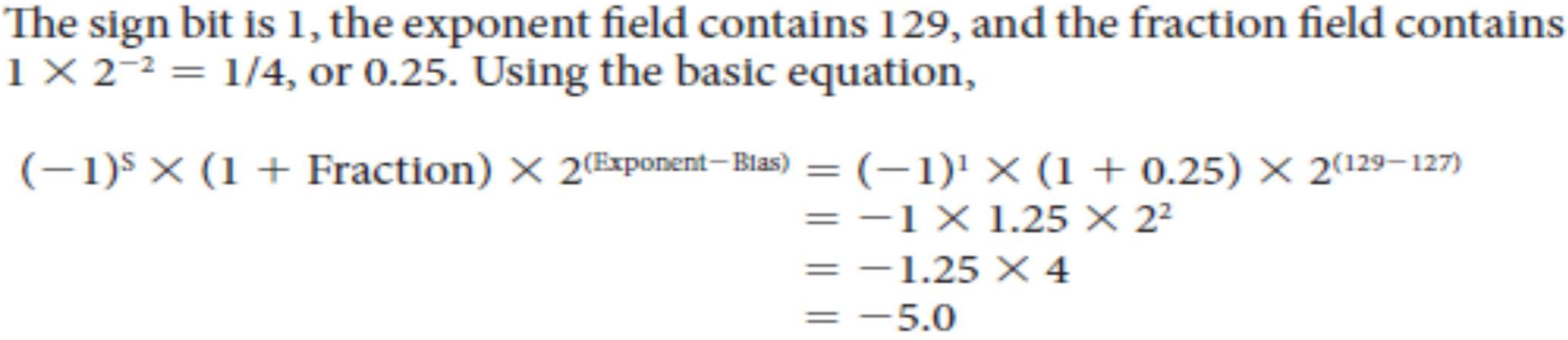
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

What decimal number is represented by this single precision float?



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.



The sign bit is 1, the exponent field contains 129, and the fraction field contains $1 \times 2^{-2} = 1/4$, or 0.25. Using the basic equation,

$$\begin{aligned}(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129-127)} \\ &= -1 \times 1.25 \times 2^2 \\ &= -1.25 \times 4 \\ &= -5.0\end{aligned}$$