Unit 7:Computer Arithmetic

Computer Arithmetic

- Data is manipulated by using the arithmetic instructions in digital computers.
- Data is manipulated to produce results necessary to give solution for the computation problems.
- The **Addition, subtraction, multiplication and division** are the four basic arithmetic operations.
- If we want then we can derive other operations by using these four operations.
- To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit.
- The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions.
- We can have **signed or unsigned negative numbers.** Fixed-point addition is the simplest arithmetic operation

- If we want to solve a problem then we use a sequence of well-defined steps.
- These steps are collectively called **algorithm.**
- To solve various problems we give algorithms. In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data.
- These instructions perform arithmetic calculations. And these instructions perform a great activity in processing data in a digital computer.
- As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods.

- A processor has an arithmetic processor(as a sub part of it) that executes arithmetic operations.
- The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction.
- Negative numbers may be in a signed magnitude or signed complement representation.
- There are three ways of representing negative fixed point - binary numbers signed magnitude, signed 1‟s complement or signed 2‟s complement.
- Most computers use the signed magnitude representation for the mantissa

The sign of a number can be represented using the **leftmost bit**:

- If bit is 0, the number is positive;
- If bit is 1, the number is negative;

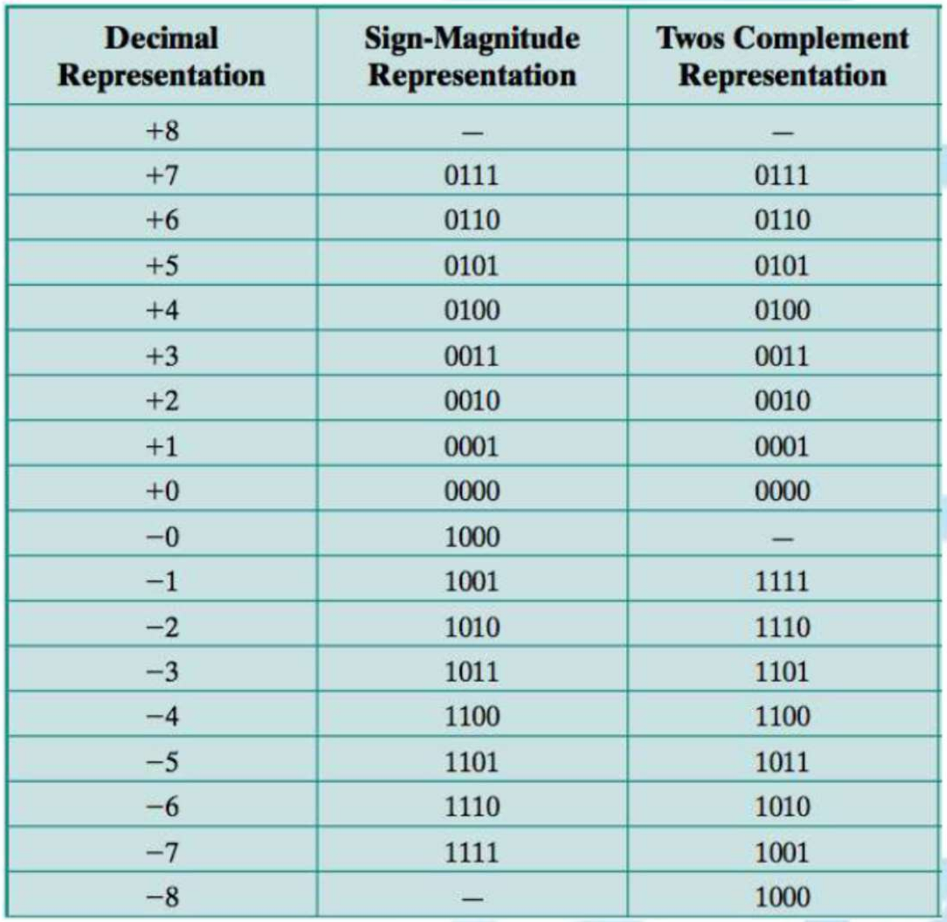
+18	= 00010010	
−18	= 10010010	(sign magnitude)

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if} \, a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if} \, a_{n-1} = 1 \end{cases}$$

Consider an n-bit integer, A, in twos complement (1/3):

- If A is positive, then the sign bit, a_{n-1}, is zero;
- Remaining bits represent number magnitude:

$$A = \sum_{i=0}^{n-2} 2^i a_i, \text{ for } A \geq 0$$



Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation
+8	—	—
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
−0	1000	—
−1	1001	1111
−2	1010	1110
−3	1011	1101
−4	1100	1100
−5	1101	1011
−6	1110	1010
−7	1111	1001
−8	—	1000

A useful illustration of twos complement:

−128	64	32	16	8	4	2	1

−128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

−128 +2 +1 = −125

- Easy in **sign-magnitude notation:**
    - move the sign bit to the new leftmost position and fill in with zeros.

+18	=	00010010	(sign magnitude, 8 bits)
+18	=	0000000000010010	(sign magnitude, 16 bits)
−18	=	10010010	(sign magnitude, 8 bits)
−18	=	1000000000010010	(sign magnitude, 16 bits)

- In this unit we will see
- 1. Fixed-point binary data in signed magnitude representation
- 2. Fixed-point binary data in signed 2’s complement representation
- 3. Floating point binary data
- 4. Binary Coded Decimal data

Addition and Subtraction with signed magnitude data

- We designate the magnitude of the two numbers by A and B.
- Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed.
- These conditions are listed in the first column of Table given.
- The other columns in the table show the actual operation to be performed with the magnitude of the numbers.
- The last column is needed to present a negative zero.
- In other words, when two equal numbers are subtracted, the result should be +0 not -0.

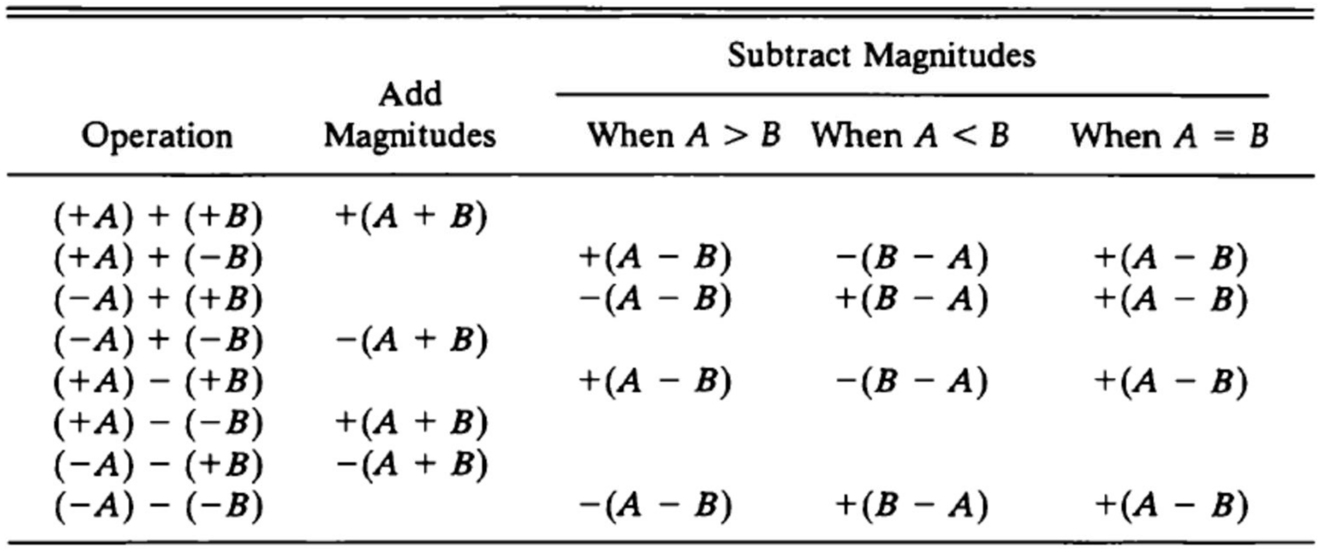
- **As display in the table, the addition algorithm states that** −
- When the signs of A and B are equal, add the two magnitudes and connect the sign of A to the output.
- When the signs of A and B are different, compare the magnitudes and subtract the smaller number from the greater number.
- The signs of the output have to be equal as A in case $A > B$ or the complement of the sign of A in case $A < B$.
- When the two magnitudes are equal, subtract B from A and modify the sign of the output to positive.

- **The subtraction algorithm states that −**
- When the signs of A and B are different, add the two magnitudes and connect the signs of A to the output.
- When the signs of A and B are the same, compare the magnitudes and subtract the smaller number from the greater number.
- The signs of the output have to be equal as A in case $A > B$ or the complement of the sign of A in case $A < B$.
- When the two magnitudes are equal, subtract B from A and modify the sign of the output to positive

Addition and subtraction with signed magnitude data

Addition: A + B ; A: Augend; B: Addend
Subtraction: A - B: A: Minuend; B: Subtrahend

TABLE 10-1 Addition and Subtraction of Signed-Magnitude Numbers



Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

- A= +3
- B=+4
- (A+B) =(3+4) =+7
- A= -5
- B= -6
- (A+B) =(5+6) = -11
- A= 17
- B= -9
- (A-B) = (17-9) = +8

- A = -3
- B=9
- (B-A) = (9-3) = +6
- A= 7
- B= -7
- (B-A) =(7-7) = +0

Addition proceeds as if the two numbers were unsigned integers:

$$\begin{array}{rcr} 1001 & = & -7 \\ +\underline{0101} & = & 5 \\ 1110 & = & -2 \end{array}$$

(a) (−7) + (+5)

Addition proceeds as if the two numbers were unsigned integers:

$$\begin{array}{rcr} 1100 & = & -4 \\ +\underline{0100} & = & 4 \\ 10000 & = & 0 \end{array}$$

(b) (−4) + (+4)

- Carry bit beyond (indicated by shading) is ignored;

Addition proceeds as if the two numbers were unsigned integers:

$$\begin{array}{rcl} 0011 & = & 3 \\ +\underline{0100} & = & 4 \\ 0111 & = & 7 \end{array}$$

(c) (+3) + (+4)

Addition proceeds as if the two numbers were unsigned integers:

$$\begin{array}{rcl} 1100 & = & -4 \\ +\underline{1111} & = & -1 \\ 11011 & = & -5 \end{array}$$

(d) (−4) + (−1)

- Carry bit beyond (indicated by shading) is ignored;

```
  0101  =  5
 +0100  =  4
  1001  =  Overflow
```

(e) (+5) + (+4)

- Two numbers of the same sign produce a different sign...

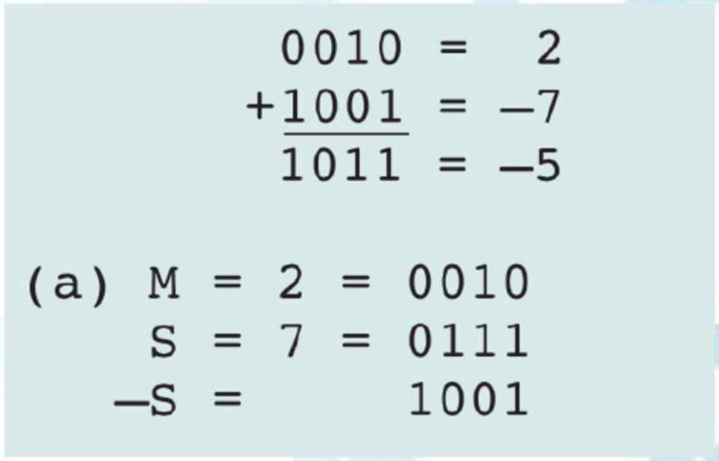
```
  1001  =  −7
 +1010  =  −6
 10011  =  Overflow
```

(f) (−7) + (−6)

- Two numbers of the same sign produce a different sign...

To subtract the **subtrahend** from the **minuend**:

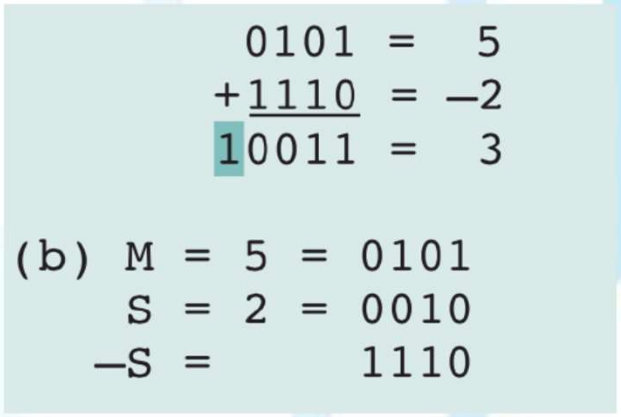
- Take the twos complement of the subtrahend **(S)** and add it to the minuend **(M)**.
  - $M + (-S)$
- *I.e.:* subtraction is achieved using addition;



```
           0010  =   2
          +1001  =  –7
           1011  =  –5

(a) M = 2 = 0010
    S = 7 = 0111
   –S =     1001
```

Subtraction is achieved using addition: $M + (-S)$



```
       0101  =   5
      +1110  =  -2
      10011  =   3

(b) M = 5 = 0101
    S = 2 = 0010
   -S =     1110
```

- Carry bit beyond (indicated by shading) is ignored;

Subtraction is achieved using addition: $M + (-S)$

```
            1011  =  —5
           +1110  =  —2
           11001  =  —7

(c) M  =  —5  =  1011
    S  =   2  =  0010
   —S  =         1110
```

- Carry bit beyond (indicated by shading) is ignored;

Subtraction is achieved using addition: $M + (-S)$

```
     0101  = 5
    +0010  = 2
     0111  = 7

(d) M =   5 = 0101
    S =  –2 = 1110
   –S =       0010
```

Subtraction is achieved using addition: $M + (-S)$

```
     0111  = 7
    +0111  = 7
     1110  = Overflow

(e) M =   7 = 0111
    S =  –7 = 1001
   –S =       0111
```

- **Overflow:** two numbers of the same sign produce a different sign;

Subtraction is achieved using addition: $M + (-S)$

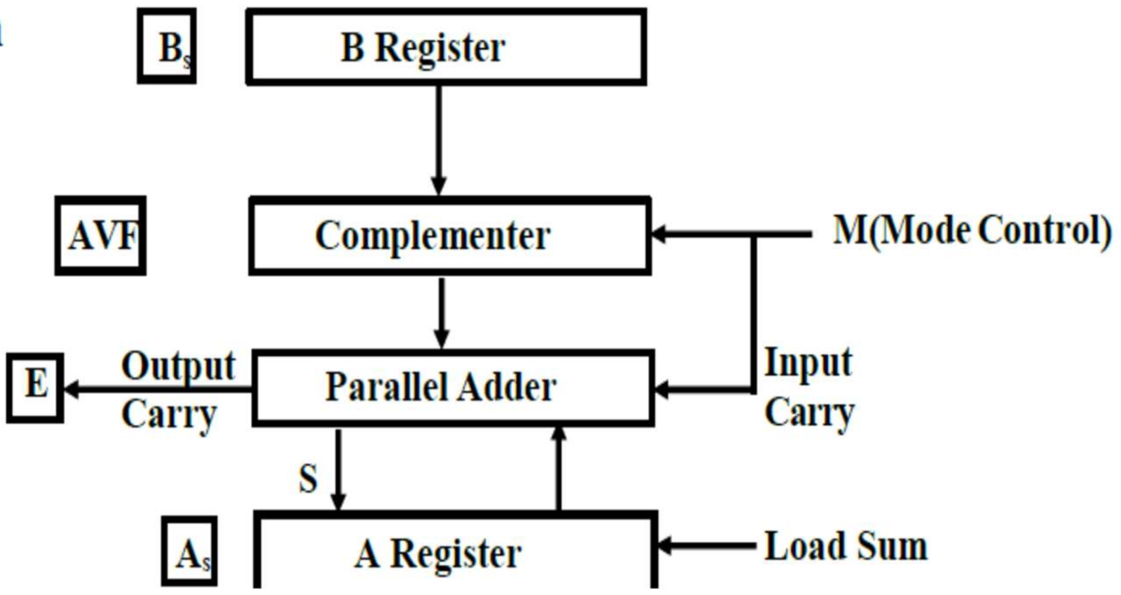
```
           1010 = –6
          +1100 = –4
          10110 = Overflow

(f) M = –6 = 1010
    S =  4 = 0100
   –S =      1100
```

- Carry bit beyond (indicated by shading) is ignored;
- **Overflow:** two numbers of the same sign produce a different sign;

Hardware Implementation

Hardware Implementation



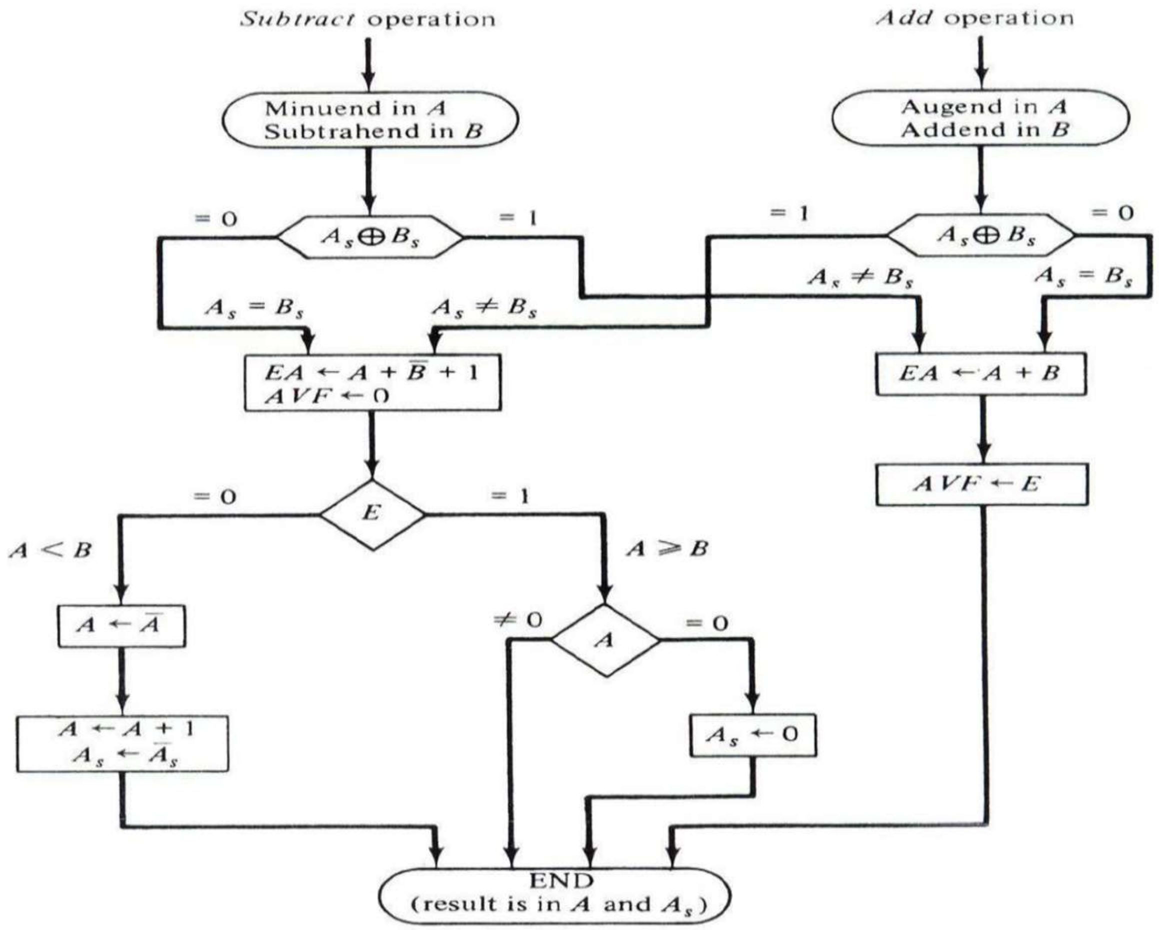


Figure 10-2 Flowchart for add and subtract operations.

- The flowchart is shown in Figure. The two signs A, and B, are compared by an exclusive-OR gate.
- If the output of the gate is 0 the signs are identical;
- If it is 1, the signs are different.
- For an add operation, identical signs dictate that the magnitudes be added.
- For a subtract operation, different signs dictate that the magnitudes be added.
- The magnitudes are added with a microoperation EA A + B, where EA is a register that combines E and A.
- The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.

- The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation.
- The magnitudes are subtracted by adding A to the 2's complemented B.
- No overflow can occur if the numbers are subtracted so AVF is cleared to 0.
- 1 in E indicates that A >= B and the number in A is the correct result. If this numbs is zero, the sign A must be made positive to avoid a negative zero. 0 in E indicates that A < B.
- For this case it is necessary to take the 2’s complement of the value in A. The operation can be done with one microoperation A A' +1.

- However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations.
- In other paths of the flowchart, the sign of the result is the same as the sign of A. so no change in A is required. However, when $A < B$, the sign of the result is the complement of the original sign of A.
- It is then necessary to complement A, to obtain the correct sign.
- The final result is found in register A and its sign in As. The value in AVF provides an overflow indication. The final value of E is immaterial.
- Figure shows a block diagram of the hardware for implementing the addition and subtraction operations.
- It consists of registers A and B and sign flip-flops As and Bs.

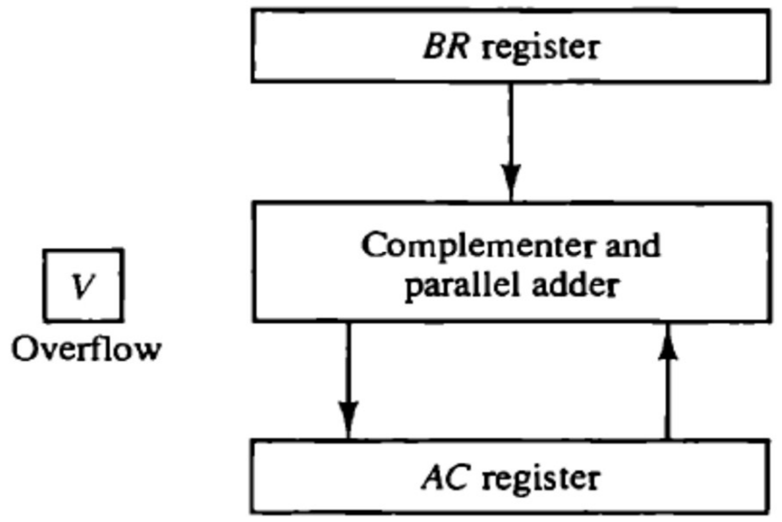
- Subtraction is done by adding A to the 2’s complement of B.
- The output carry is transferred to flip-flop E , where it can be checked to determine the relative magnitudes of two numbers.
- The add-overflow flip-flop *AVF* holds the overflow bit when A and B are added.
- The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

Addition and Subtraction with signed 2’s complement data

- As we know a negative number can be represented by 2’s complement method
- 33:- 00100001
- -33:- 11011110+1 = 11011111 (2’s complement)

Addition and Subtraction with signed 2’s complement data

Figure 10-3 Hardware for signed-2’s complement addition and subtraction.



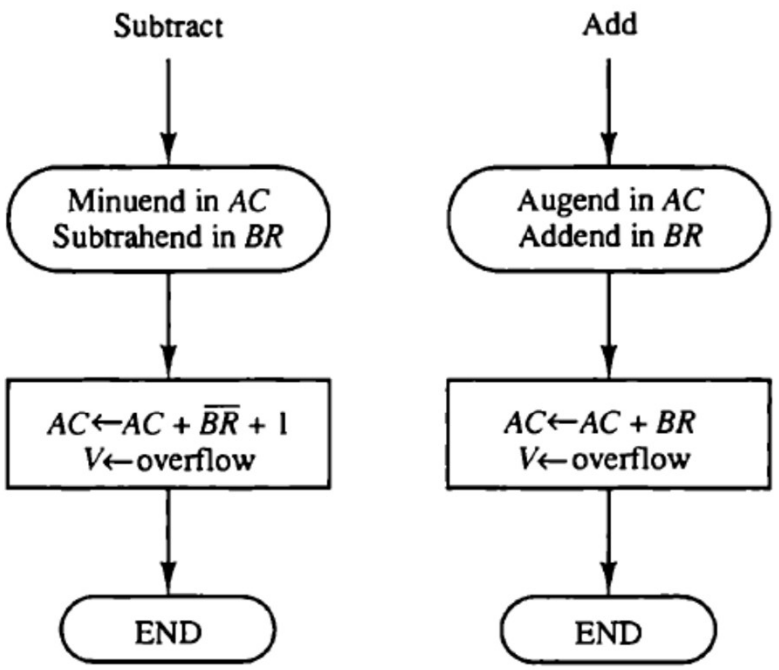
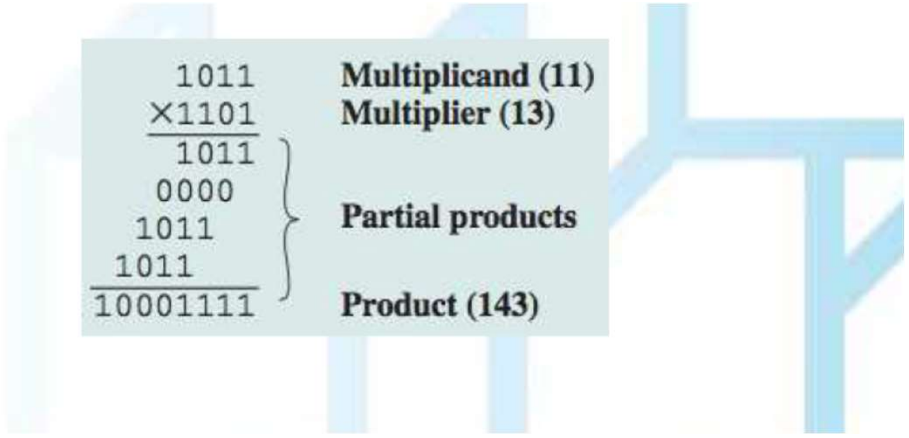


Figure 10-4 Algorithm for adding and subtracting numbers in signed-2's complement representation.

Multiplication

- Multiplication is a complex operation:
  - **Compared with addition and subtraction**
- Again lets consider multiplying for the following cases:
  - **Two unsigned numbers**
  - **Two signed (twos complement) numbers**



```
    1011     Multiplicand (11)
   ×1101     Multiplier (13)
    1011  }
   0000   }
  1011    }  Partial products
 1011     }
10001111     Product (143)
```

- Multiplication of two fixed-point binary numbers in signed magnitude representation is done with paper and pencil by a process of successive shift and adds operations.
- This process is best illustrated with a numerical example:

```
 23        10111   Multiplicand
 19    × 10011   Multiplier
 --   ---------
           10111
          10111
         00000    +
        00000
       10111
       ---------
437    110110101   Product
```

- This process looks at successive bits of the multiplier, least significant bit first.
- If the multiplier bit is 1, the multiplicand is copied as it is; otherwise, we copy zeros.
- Now we shift numbers copied down one position to the left from the previous numbers. Finally, the numbers are added and their sum produces the product

- When multiplication is implemented in a digital computer, we change the process slightly.
- Here, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers, and successively accumulate the partial products in a register.
- Second, instead of shifting the multiplicand to left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required relative positions.
- Now, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.

Figure 10-5 Hardware for multiply operation.

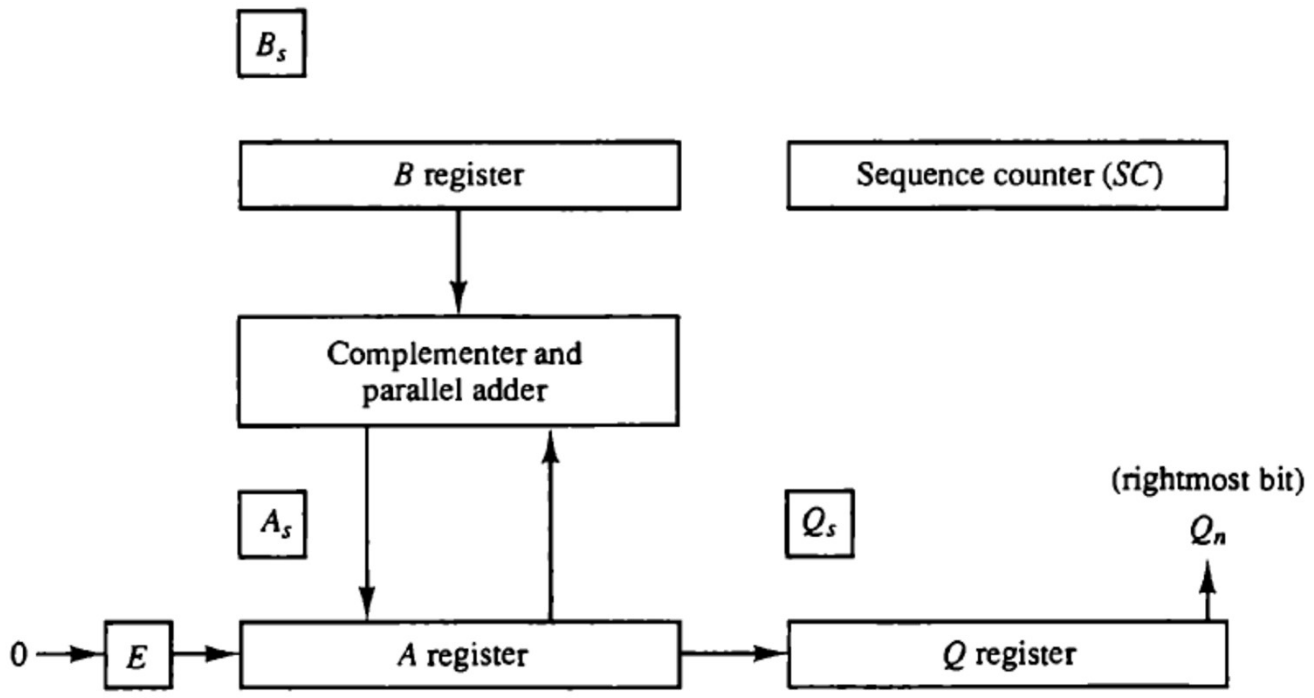
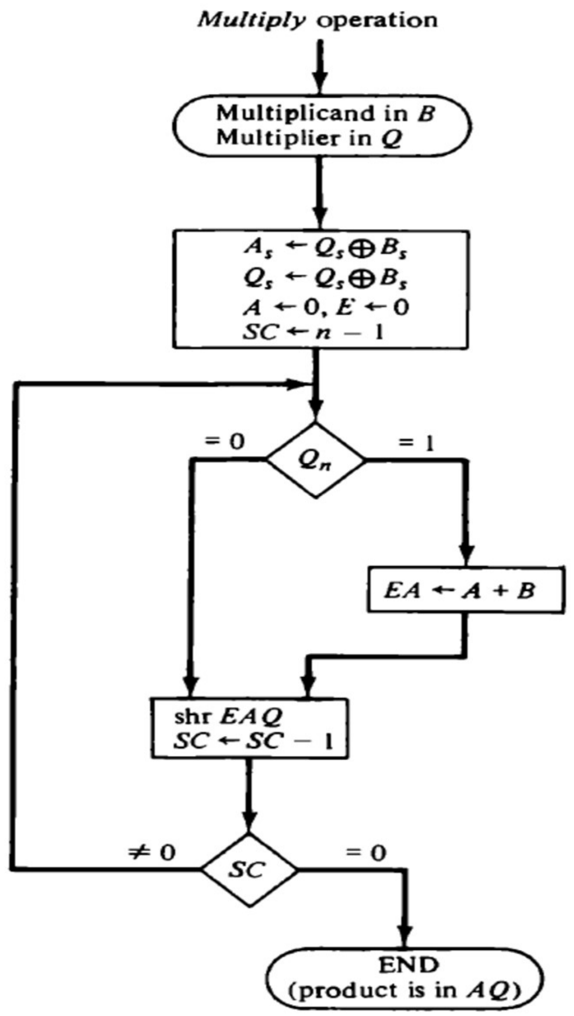


Figure 10-6 Flowchart for multiply operation.



- 1. Multiplier and multiplicand are loaded into two registers (**Q** and **B**);
- 2. A third register, the **A** register, is also needed and is initially set to 0;
- 3. There is also a 1-bit **E** register, initialized to 0:
  - which holds a potential carry bit resulting from addition.

- 4. Control logic then reads the bits of the multiplier one at a time:
- If Q_0 is 1, then:
  - the multiplicand is added to the A register...
  - and the result is stored in the A register...
  - with the *E* bit used for overflow.
  - Then all of the bits of the *E*, *A*, and *Q* registers are shifted to the right one bit:
  - So that the *C* bit goes into A_{n-1}, A_0 goes into Q_{n-1} and Q_0 is lost.
- If Q_0 is 0, then:
  - Then no addition is performed, just the shift;
  - Process is repeated for each bit of the original multiplier;
  - Resulting 2n-bit product is contained in the A and Q registers

B= 23= 10111 (multiplicand)
Q= 19= 10011(multiplier)

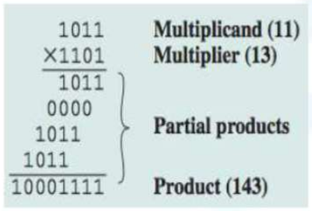
B(multiplicand)	Operations	E	A	Q(multiplier)	SC
10111	Initialization	0	00000	1001**1**	5
	Qn=1 ADD A,B		00000 10111		
		0	10111	10011	
	Shr EAQ	0	01011	1100**1**	4
	Qn=1 ADD A,B		01011 10111		
		1	00010	11001	
	Shr EAQ SC-1	0	10001	01100	3
	Qn= 0 Shr	0	01000	10110	2

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

C	A	Q	M

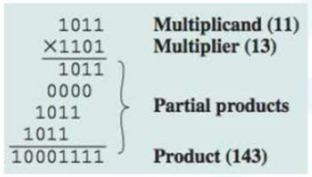
```
    1011     Multiplicand (11)
   ×1101     Multiplier (13)
    1011  ⎫
   0000   ⎬  Partial products
  1011    ⎪
 1011     ⎭
10001111     Product (143)
```

C	A	Q	M	
0	0000	1101	1011	Initial Values



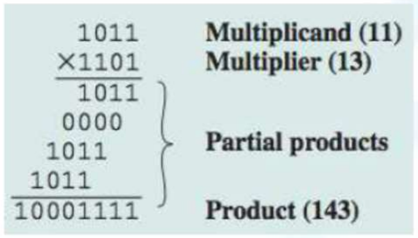
```
    1011   Multiplicand (11)
   ×1101   Multiplier (13)
    1011 ┐
   0000  │
  1011   ├ Partial products
 1011    │
10001111 ┘ Product (143)
```

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add



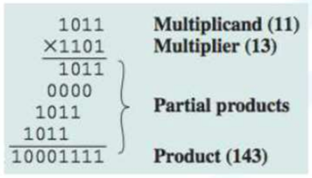
```
    1011   Multiplicand (11)
   ×1101   Multiplier (13)
    1011 ┐
   0000  │
  1011   ├ Partial products
 1011    │
10001111 ┘ Product (143)
```

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right



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

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right
0	0010	1111	1011	Shift



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

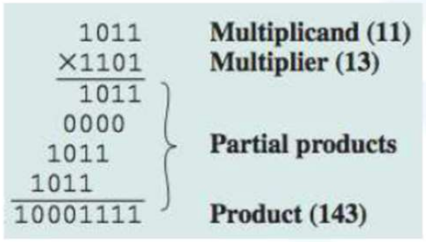
C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right
0	0010	1111	1011	Shift
0	1101	1111	1011	Add

```
    1011     Multiplicand (11)
   ×1101     Multiplier (13)
    1011  }
   0000   }  Partial products
  1011    }
 1011     }
10001111     Product (143)
```

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	0110	1111	1011	Shift

```
    1011     Multiplicand (11)
   ×1101     Multiplier (13)
    1011  }
   0000   }  Partial products
  1011    }
 1011     }
10001111     Product (143)
```

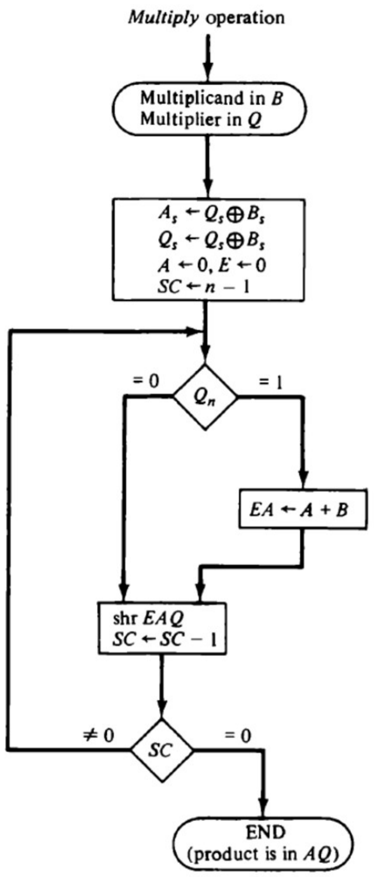
C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	0110	1111	1011	Shift
1	0001	1111	1011	Add



```
    1011     Multiplicand (11)
   ×1101     Multiplier (13)
    1011  ┐
   0000   │
  1011    ├  Partial products
 1011     ┘
10001111     Product (143)
```

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift Right
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	0110	1111	1011	Shift
1	0001	1111	1011	Add
0	1000	1111	1011	Shift

Figure 10-6 Flowchart for multiply operation.



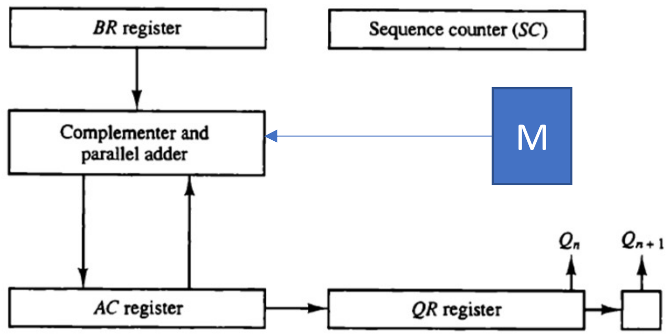
Booth’s algorithm (for signed 2’s compliment)

- Booth algorithm gives a procedure for multiplying binary integers in signed- 2‟s complement representation.
- It is also used to speed up the performance of the multiplication process. It is very efficient too.
- It operates on the fact that strings of 0‟s in the multiplier require no addition but just shifting, and a string of 1‟s in the multiplier from bit weight 2k to weight 2m can be treated as 2k+1 – 2m.
- For example, the binary number 001110 (+14) has a string 1‟s from 23 to 21 (k=3, m=1). The number can be represented as 2k+1 –2m. = 24 – 21 = 16 – 2 = 14.
- Therefore, the multiplication M X 14, where M is the multiplicand and 14 the multiplier, can be done as M X 24 – M X 21.

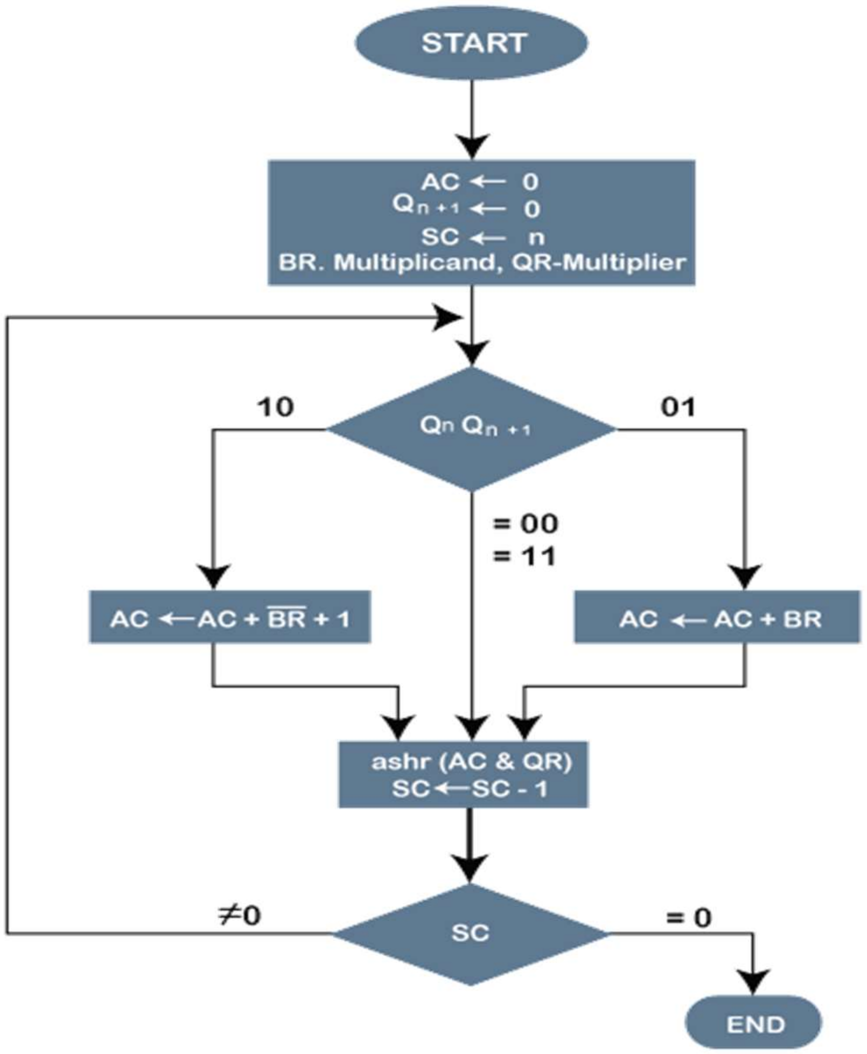
- Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Hardware Implementation of Booth’s Multiplication

Figure 10-7 Hardware for Booth algorithm.



- M= 0 Adder AC+BR
- M=1 Complementor and adder AC+BR’+1 = AC-BR



START
AC ← 0
Q_{n+1} ← 0
SC ← n
BR. Multiplicand, QR-Multiplier
$Q_n\ Q_{n+1}$
10
01
= 00
= 11
AC ← AC + $\overline{BR}$ + 1
AC ← AC + BR
ashr (AC & QR)
SC ← SC - 1
SC
≠0
= 0
END

Multiplicand=-9 = (10111)
Multiplier=-13 = (10011)

Qn	Qn+1	BR=10111 BR’+1 = 01001	AC	Qr (multiplier	Qn+1	SC
1	0	Intial	00000 01001	10011	0	5

Working of Booth’s Multiplication Algorithm

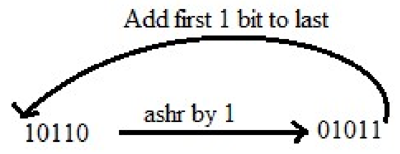
1. Set the Multiplicand and Multiplier binary bits as M and Q, respectively.
2. Initially, we set the AC and Q_{n+1} registers value to 0.
3. SC represents the number of Multiplier bits (Q), and it is a sequence counter that is continuously decremented till equal to the number of bits (n) or reached to 0.
4. A Qn represents the last bit of the Q, and the Q_{n+1} shows the incremented bit of Qn by 1.

1. On each cycle of the booth algorithm, Q_n and Q_{n+1} bits will be checked on the following parameters as follows:
   1. When two bits Q_n and Q_{n+1} are 00 or 11, we simply perform the arithmetic shift right operation (ashr) to the partial product AC. And the bits of Qn and Q_{n+1} is incremented by 1 bit.
   2. If the bits of Q_n and Q_{n+1} is shows to 01, the multiplicand bits (M) will be added to the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.
   3. If the bits of Q_n and Q_{n+1} is shows to 10, the multiplicand bits (M) will be subtracted from the AC (Accumulator register). After that, we perform the right shift operation to the AC and QR bits by 1.

1. The operation continuously works till we reached n - 1 bit in the booth algorithm.
2. Results of the Multiplication binary bits will be stored in the AC and QR registers.

1. **RSC (Right Shift Circular)**

It shifts the right-most bit of the binary number, and then it is added to the beginning of the binary bits.



2. **RSA (Right Shift Arithmetic)**

It adds the two binary bits and then shift the result to the right by 1-bit position.

Example: 0100 + 0110 => 1010, after adding the binary number shift each bit by 1 to the right and put the first bit of resultant to the beginning of the new bit.

- In the flowchart, initially, **AC** and $\mathbf{Q_{n+1}}$ bits are set to 0, and the **SC** is a sequence counter that represents the total bits set **n,** which is equal to the number of bits in the multiplier.
- There are **BR** that represent the **multiplicand bits,** and QR represents the **multiplier bits**.
- After that, we encountered two bits of the multiplier as Q_n and Q_{n+1}, where Qn represents the last bit of QR, and Q_{n+1} represents the incremented bit of Qn by 1.
- Suppose two bits of the multiplier is equal to 10; it means that we have to subtract the multiplier from the partial product in the accumulator AC and then perform the arithmetic shift operation (ashr).

- If the two of the multipliers equal to 01, it means we need to perform the addition of the multiplicand to the partial product in accumulator AC and then perform the arithmetic shift operation (**ashr**), including $\mathbf{Q_{n+1}}$.
- The arithmetic shift operation is used in Booth's algorithm to shift AC and QR bits to the right by one and remains the sign bit in AC unchanged. And the sequence counter is continuously decremented till the computational loop is repeated, equal to the number of bits (n).

Example of Booth's Algorithm for: 7×3

A	Q	Q_{-1}	M

Example of Booth's Algorithm for: 7×3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values

Example of Booth's Algorithm for: 7×3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$

Example of Booth´s Algorithm for: 7×3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right

Example of Booth´s Algorithm for: 7×3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right
1110	0100	1	0111	Arithmetic Shift Right

Example of Booth´s Algorithm for: 7×3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right
1110	0100	1	0111	Arithmetic Shift Right
0101	0100	1	0111	$A \leftarrow A + M$

Example of Booth's Algorithm for: 7×3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right
1110	0100	1	0111	Arithmetic Shift Right
0101	0100	1	0111	$A \leftarrow A + M$
0010	1010	0	0111	Arithmetic Shift Right

Example of Booth's Algorithm for: 7×3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right
1110	0100	1	0111	Arithmetic Shift Right
0101	0100	1	0111	$A \leftarrow A + M$
0010	1010	0	0111	Arithmetic Shift Right
0001	0101	0	0111	Arithmetic Shift Right

Example of Booth's Algorithm for: 7×3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right
1110	0100	1	0111	Arithmetic Shift Right
0101	0100	1	0111	$A \leftarrow A + M$
0010	1010	0	0111	Arithmetic Shift Right
0001	0101	0	0111	Arithmetic Shift Right

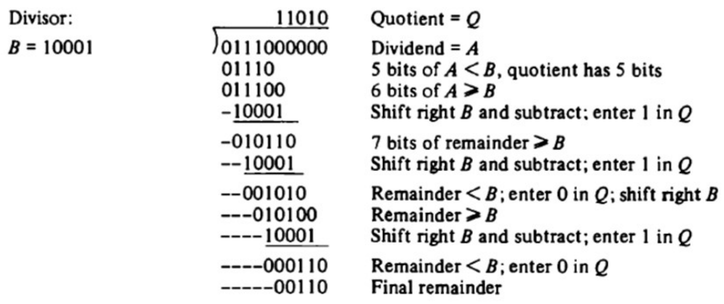
Final result appears in the A and Q registers:

- $A = 0001$
- $Q = 0101$
- $AQ = 00010101_2 = 21_{10}$

Division of two signed bit numbers

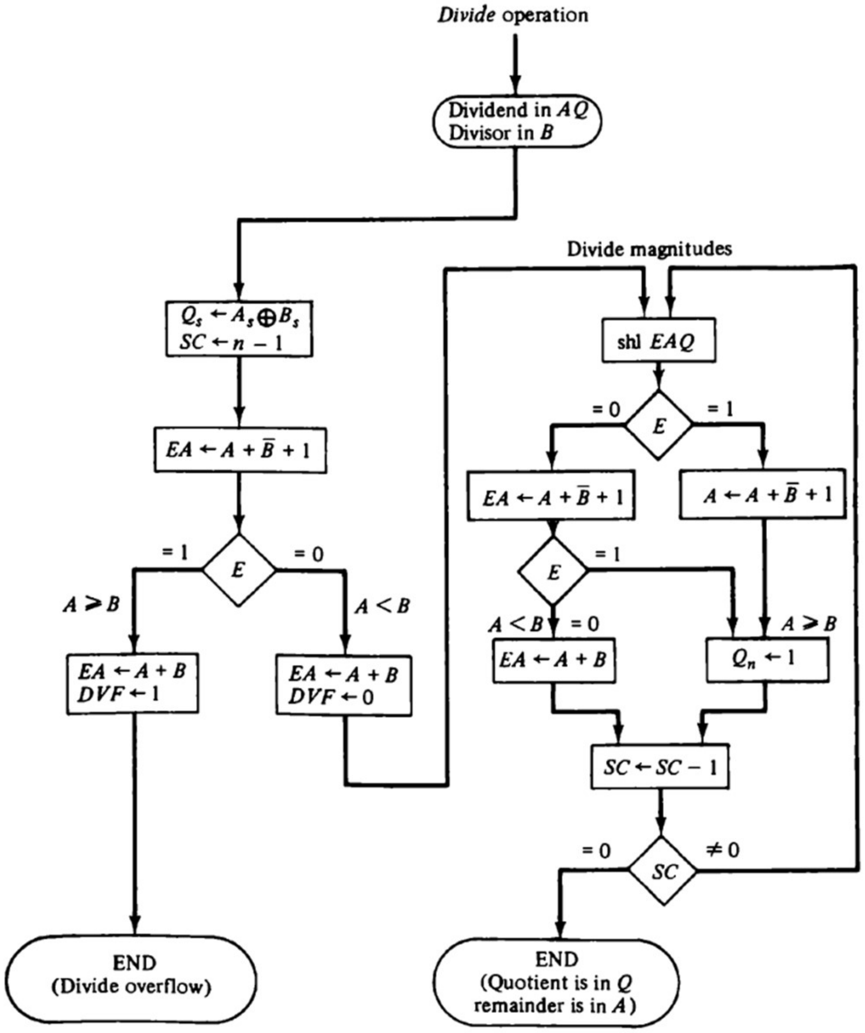
- Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations.
- Binary division is much simpler than decimal division because here the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor.

Figure 10-11 Example of binary division.



Divisor:	11010	Quotient = Q
B = 10001	0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geqslant B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geqslant B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geqslant B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

Figure 10-13 Flowchart for divide operation.



- Divisor in B
- Dividend in AQ

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

Perform Division Restoring Algorithm Dividend = 11 Divisor = 3

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

Divisor $B = 10001$, $\overline{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		$\underline{01111}$		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		$\underline{01111}$		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		$\underline{01111}$		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		$\underline{10001}$		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		$\underline{01111}$		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		$\underline{01111}$		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		$\underline{10001}$		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Figure 10-12 Example of binary division with digital hardware.