CHAPTER 7
COMPUTER ARITHMETIC
BSC.CSIT 3rd semester

LH-6

Er. Rolisha Sthapit

- 7.1. Addition and Subtraction with Signed Magnitude Data, Addition and Subtraction with Signed 2’s Complement Data
- 7.2. Multiplication of Signed Magnitude Data, Booth Multiplication, Division of Signed magnitude Data, Divide Overflow

INTRODUCTION

- Computer Arithmetic includes the arithmetic operation like addition, subtraction, multiplication and division.
- These operations are performed usually in signed 2’s complement.
- However, the processing can be preceded with signed magnitude, signed 1’s complement and signed 2’s complement.
- For every process, we design a hardware and analyze the corresponding algorithm used.

7.1 ADDITION AND SUBTRACTION WITH SIGNED MAGNITUDE DATA

- In this process, we designate the magnitude of two numbers by A and B.
- When two signed numbers A and B are added and subtracted, we find 8 different conditions to consider as described in following table:

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

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

Addition (subtraction) algorithm: when the signs of A and B are identical (different), add magnitudes and attach the sign of A to result. When the signs of A and b are different (identical), compare the magnitudes and subtract the smaller form larger.

- **Hardware Implementation**

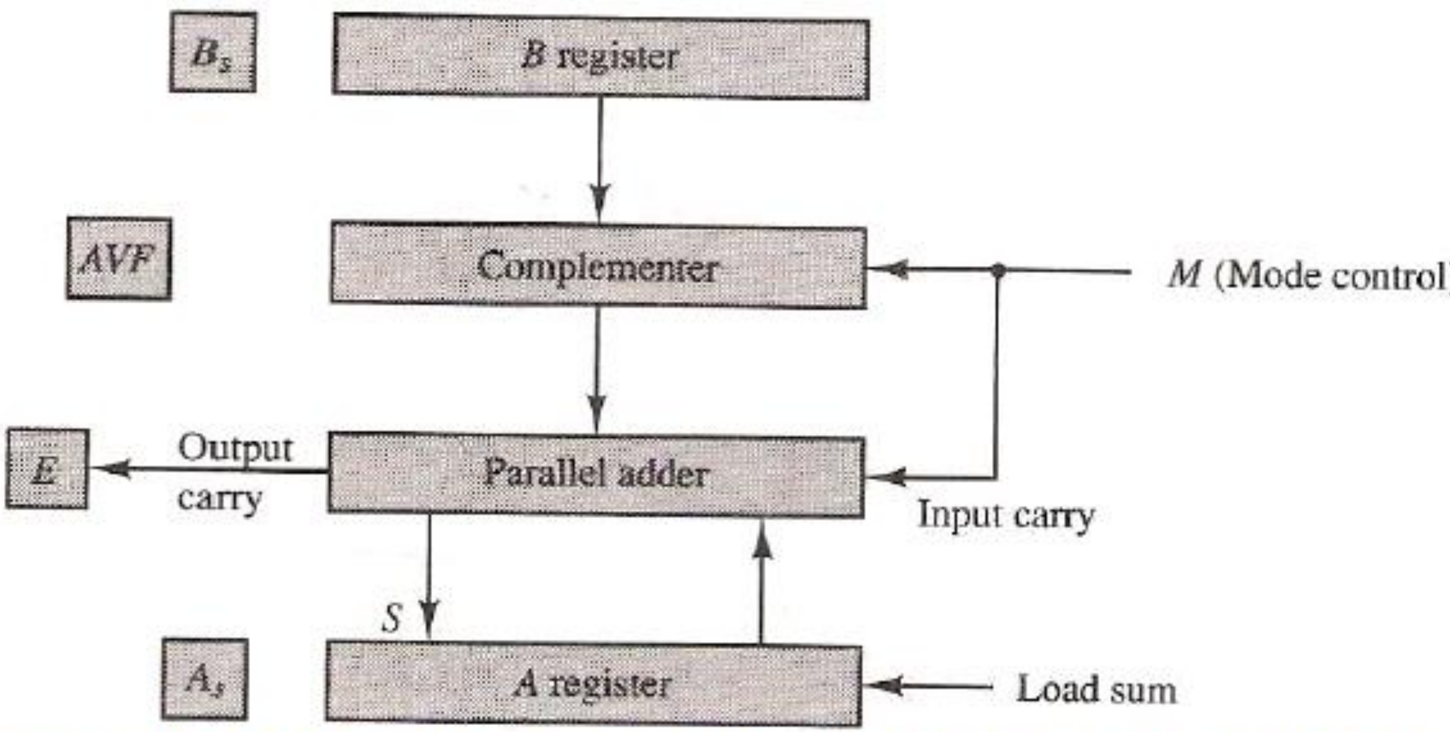


Fig: hardware for signed-magnitude addition and subtraction

- To implement the two arithmetic operations with hardware, we have to store numbers into two register A and B.
- Let A_s and B_s be two flip-flops that holds corresponding signs.
- The result is transferred to A and A_s. A and A_s together form a accumulator.

Block Diagram Description:

- Hardware above consists of registers A and B and sign flip-flops A_s and B_s.
- Subtraction is done by adding A to the 2's complement of B.
- Output carry is transferred to flip-flop E, where it can be checked to determine the relative magnitude of two numbers.
- Add-overflow flip-flop AVF holds overflow bit when A and B are added. Addition of A and B is done through the parallel adder.
- The sum (S) output of adder is applied to A again.
- The complementer provides an output of B or B' depending on mode input M.

- When M = 0, the output of B is transferred to the adder, the input carry is 0 and thus output of adder is A+B.
- When M=1, 1's complement of B is applied to the adder, input carry is 1 and output is S = A+B'+1 (i.e. A-B).

Hardware Algorithm

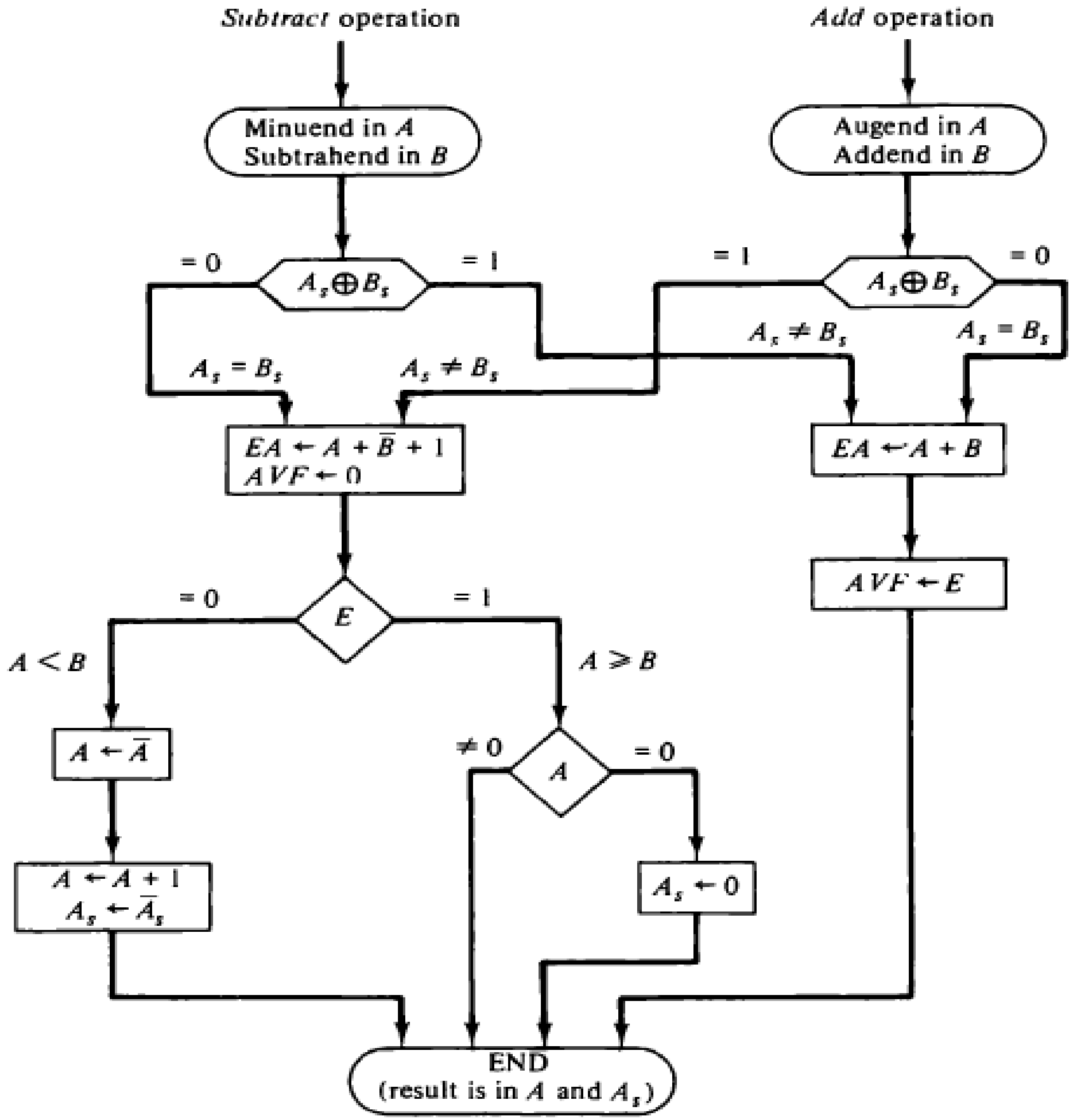


Figure 10-2 Flowchart for add and subtract operations.

- The flowchart for the hardware algorithm is shown above.
- 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 E A $\leftarrow$ A + B, where EA is a register that com-bines 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 complement of B .
- No overflow can occur if the numbers are subtracted so AVF is cleared to 0.
- A 1in E indicates that $A \geq B$ and the number in A is the correct result.
- If this number is zero, the sign A must be made positive to avoid a negative zero. A 0 in E indicates that $A < B$.

- For this case it is necessary to take the 2's complement of the value in A. This operation can be done with one microoperation $A \leftarrow A' + 1$.
- 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 A_s. The value in AVF provides an overflow indication. The final value of E is immaterial.

EXAMPLE

Perform 45 + (-23)

- Operation is add
- 45 = 00101101
- -23 = 10010111
- A_s = 0 A=0101101
- B_s = 1 B=0010111
- As $\oplus$ Bs =1
- EA=A + B' + 1 = 0101101 + 1101000 +1 = 10010110
- AVF=0
- => E=1 A= 0010110
- Result is AsA= 0 0010110

Exercise

Perform

- (-65) + (50)
- (-30) + (-12)
- (20) + (34)
- (40) – (60)
- (-20) – (50)

Addition and Subtraction with Signed 2's Complement Data

- The addition of two numbers in signed 2's complement form consists of adding the numbers with signed bit treated the same as the other bits of numbers.
- A carry out of the sign bits position is discarded. The subtraction consists of the first taking the 2's complement of the subtrahend and then adding it to minuend.

Hardware Implementation

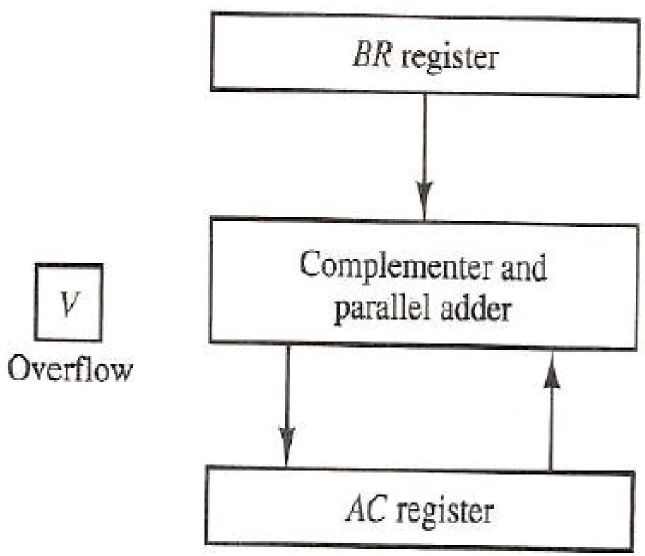


Fig: hardware for signed-2's complement addition and subtraction

→Register configuration is same as signed-magnitude representation except sign bits are not separated. The leftmost bits in AC and BR represent sign bits.

→Significant difference: sign bits are added are subtracted together with the other bits in complementer and parallel adder. The overflow flip-flop V is set to 1 if there is an overflow. Output carry in this case is discarded.

Hardware Algorithm

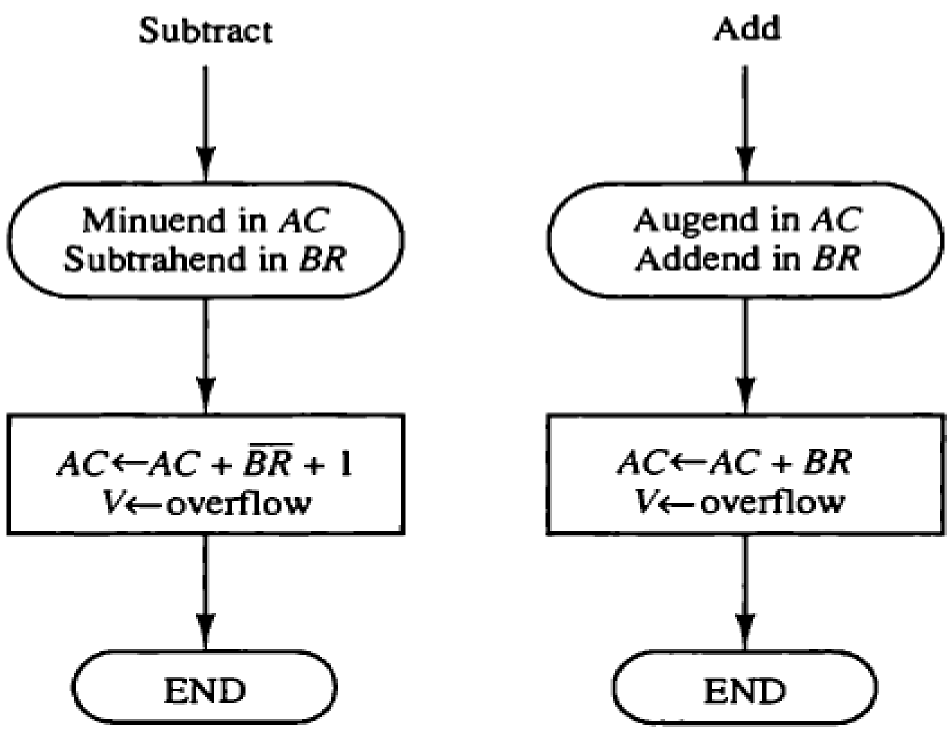


Figure 10-4 Algorithm for adding and subtracting numbers in signed-2's complement representation.

The algorithm for adding and subtracting two binary numbers in signed-2's complement representation is shown in the flowchart of Fig. 10-4. The sum is obtained by adding the contents of *AC* and *BR* (including their sign bits). The overflow bit *V* is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise. The subtraction operation is accomplished by adding the content of *AC* to the 2's complement of *BR*. Taking the 2's complement of *BR* has the effect of changing a positive number to negative, and vice versa.

Example:

- Perform 33 + (-35)
- AC = 33 = 00100001
- BR = -35 = 2's complement of 35 = 11011101
- AC + BR = 11111110 = -2 which is the result

- Comparing this algorithm with its signed magnitude counterpart, it is much easier to add and subtract numbers. For this reason most computers adopt this representation over the more familiar signed-magnitude.

Multiplication Algorithms

- 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.

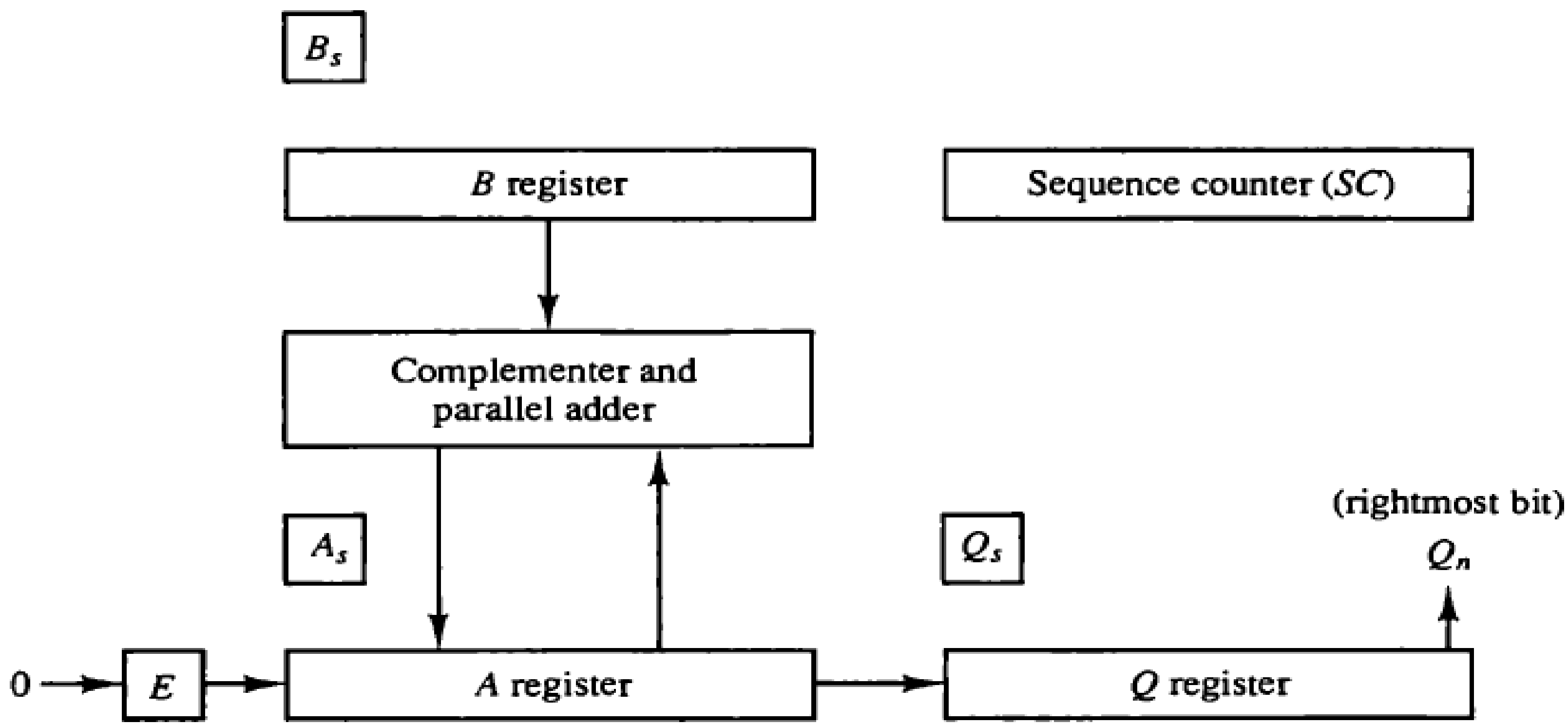
```
 23        10111    Multiplicand
 19      × 10011    Multiplier
         -------
           10111
          10111
         00000     +
        00000
       10111
       ---------
437    110110101    Product
```

- The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down.
- The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.

MULTIPLICATION USING SIGNED MAGNITUDE DATA

- **Hardware Implementation**

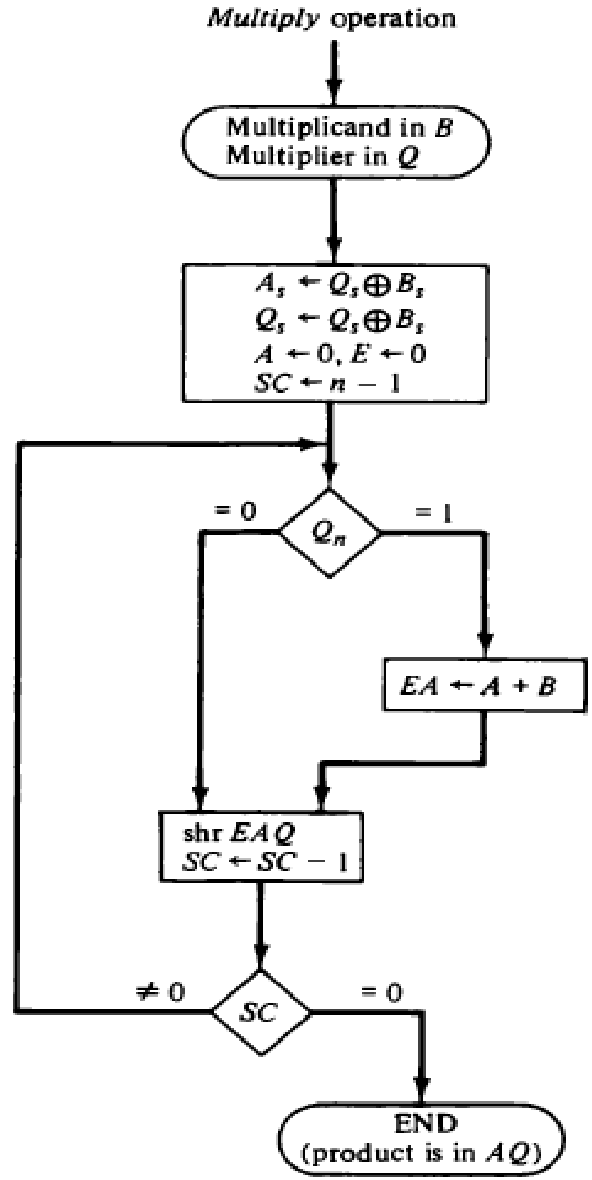
Figure 10-5 Hardware for multiply operation.



- The hardware for multiplication consists of the equipment shown in Fig. A_s and B_s stores sign bit and these registers together with registers A and B are shown in Fig.
- The multiplier is stored in the Q register and its sign in Q_s. The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.

- **Hardware Algorithm**

Figure 10-6 Flowchart for multiply operation.



- Figure above is a flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs, respectively. The signs are compared, and both A and Q are set to correspond to the sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to a number equal to the number of bits of the multiplier. We are assuming here that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n - 1 bits.

- After the initialization, the low-order bit of the multiplier in Q_n is tested. If it is a 1, the multiplicand in B is added to the present partial product in A. If it is a 0, nothing is done. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. The process stops when SC = 0. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

EXAMPLE MULTIPLY 23*19 B=23 Q=19

TABLE 10-2 Numerical Example for Binary Multiplier

Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

ASSIGNMENT

USING SIGNED MAGNITUDE MULTIPLICATION, MULTIPLY THE FOLLOWING

- 17*-13
- -13*10
- 22*25
- 10*-20

MULTIPLICATION USING SIGNED 2'S COMPLEMENT DATA (BOOTH'S ALGORITHM)

- This algorithm gives a method for multiplying binary integers in signed 2's complement representation. As in other algorithm, Booth algorithm requires examination of the multiplier bits and shifting of the partial product. Before shifting, the multiplicand may be added to the partial product, subtracted from the partial product or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant are in a string of 1's in a multiplier.

2. The multiplicand is added to the partial product upon encountering the first zero (provided that there was a previous 1) in a string of 0’s in the multiplier.

3. The partial product doesn’t change when the multiplier bit is identical to the previous multiplier bit.

- Hardware Implementation

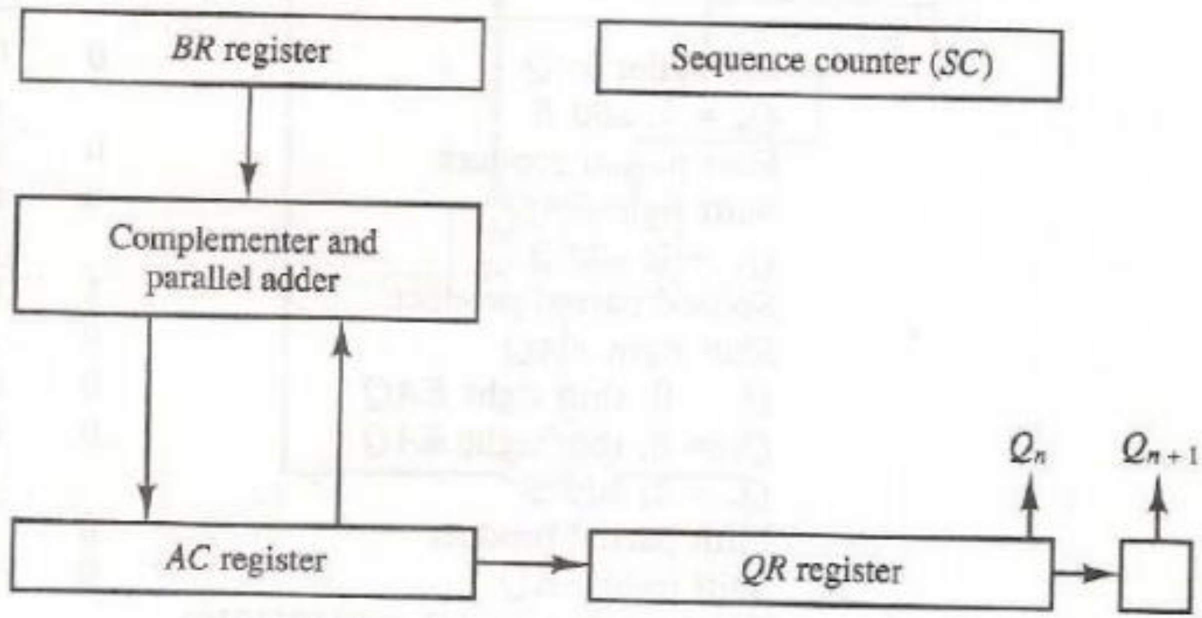
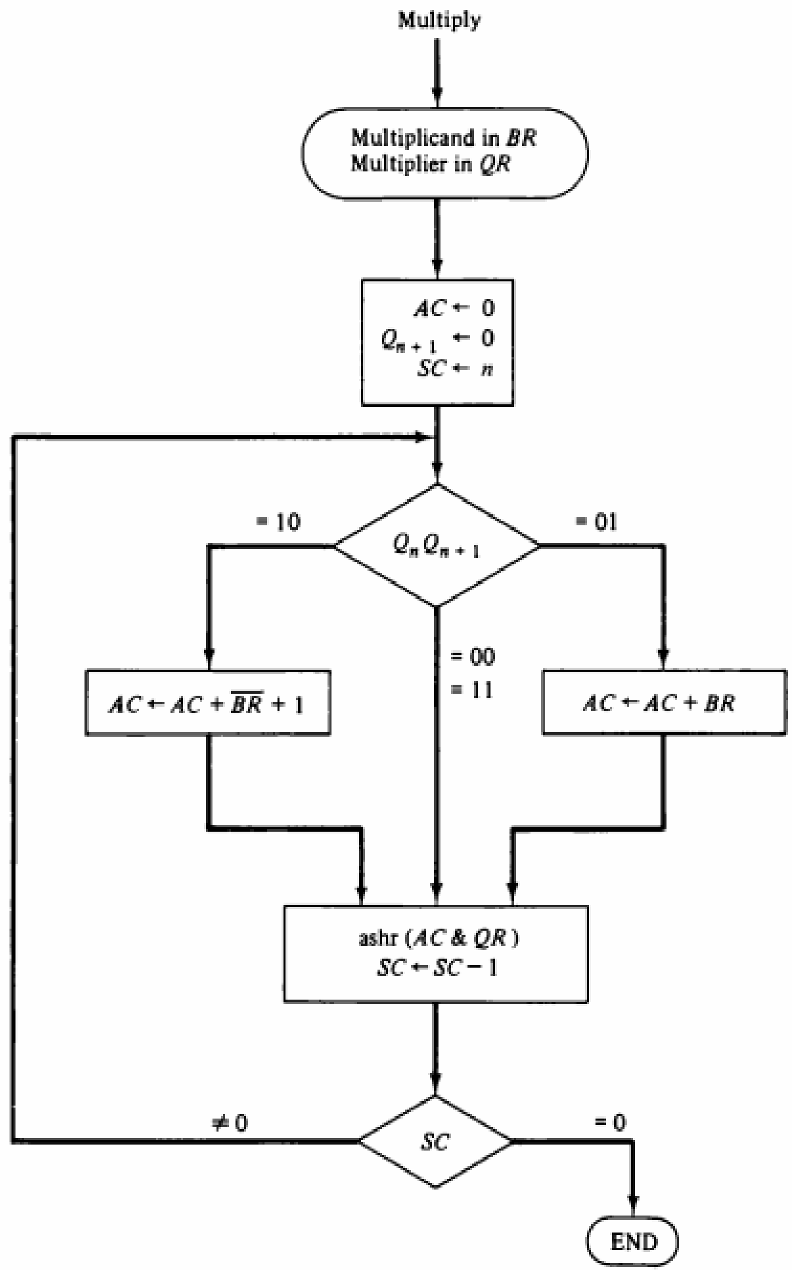


Fig. Hardware for Booth Algorithm

- For hardware implementation it requires the configuration as shown in figure. It consists o AC, BR and QR register to store partial product, multiplicand and multiplier respectively. Q_n designates LSB of multiplier in register QR.
- An extra flipflop Q_{n+1} is appended to QR to facilitate the storage of previous LSB.

- Hardware Algorithm



- Multiplicand is in BR and multiplier is in QR. AC and the appended bit Q_{n+1} are initially cleared to zero and the SC is set to a number equal to a number of bits in the multiplier. The two bits in the multiplier Q_n Q_{n+1} are determined. This is arithmetic shift right which shifts AC and QR to the right and leaves the sign bit in AC unchanged. The final product appears in AC and QR. The final value of Q_{n+1} is the original sign bit of the multiplier and shouldn't be taken as part of the product.

MULTIPLY -9*-13 using BOOTH Algorithm

TABLE 10-3 Example of Multiplication with Booth Algorithm

$Q_n\,Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

EXERCISE

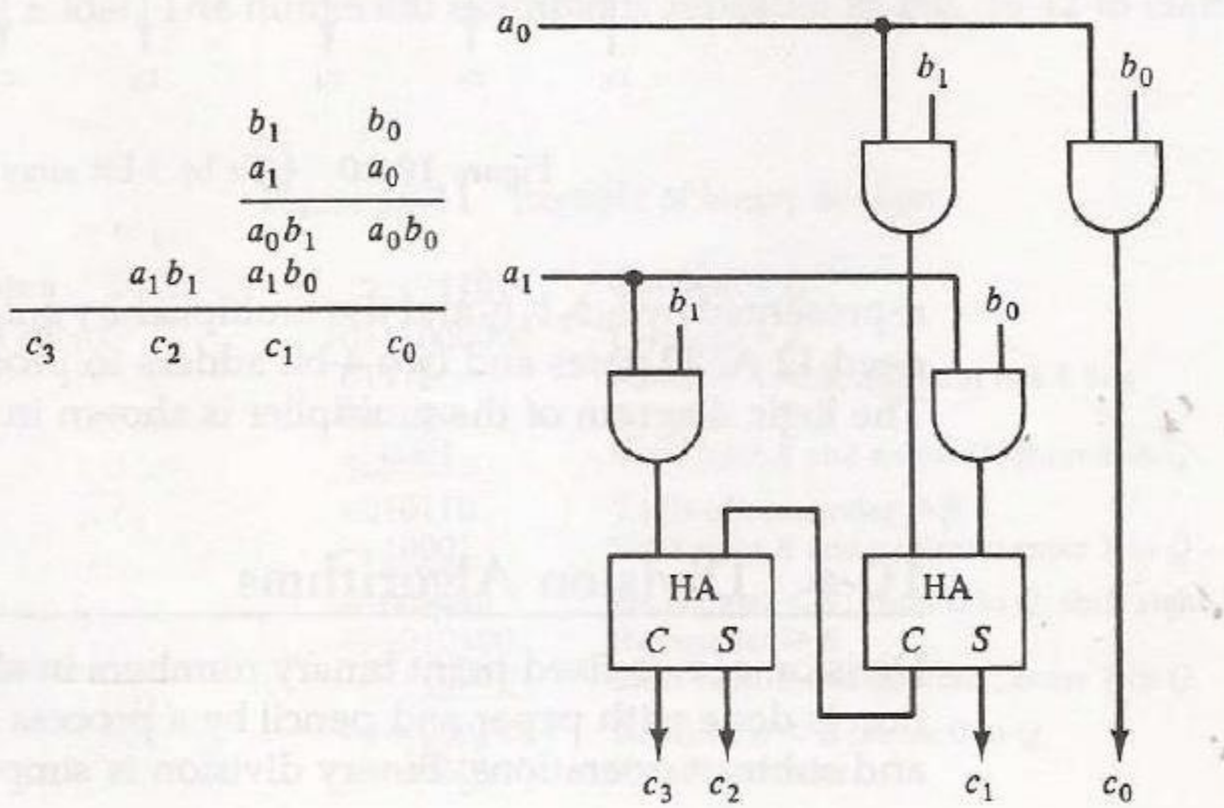
Multiply following using Booth Algorithm:

- -7*19
- -100*200
- -25*-24

ARRAY MULTIPLIER

- Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift microoperations.
- The multiplication of two binary numbers can be done with one micro-operation by means of a combinational circuit that forms the product bits all at once.
- This is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires a large number of gates, and for this reason it was not economical until the development of integrated circuits.

Figure 2-bit by 2-bit array multiplier.



- To see how an array multiplier can be implemented with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in Fig. The multiplicand bits are b_1 and b_0, the multiplier bits are a_l and a_0, and the product is $c_3c_2c_1c_0$. The first partial product is formed by multiplying a_0 by b_1, b_0. The multiplication of two bits such as a_0 and b_0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can be implemented with an AND gate.

- As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a_1 by b_1, b_0 and is shifted one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

- A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier.
- The binary output in each level of AND gatesis added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand bits we need j x k AND gates and (j — 1) k-bit adders to produce a product of j + k bits. As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be represented by b_3 b_2 b_1 b_0 and the multiplier by a_2 a_1 a_0. Since k = 4 and j = 3, we need 12 AND gates and two 4-bit adders to produce a product of seven bits.

- The logic diagram of the multiplier is shown in Figure

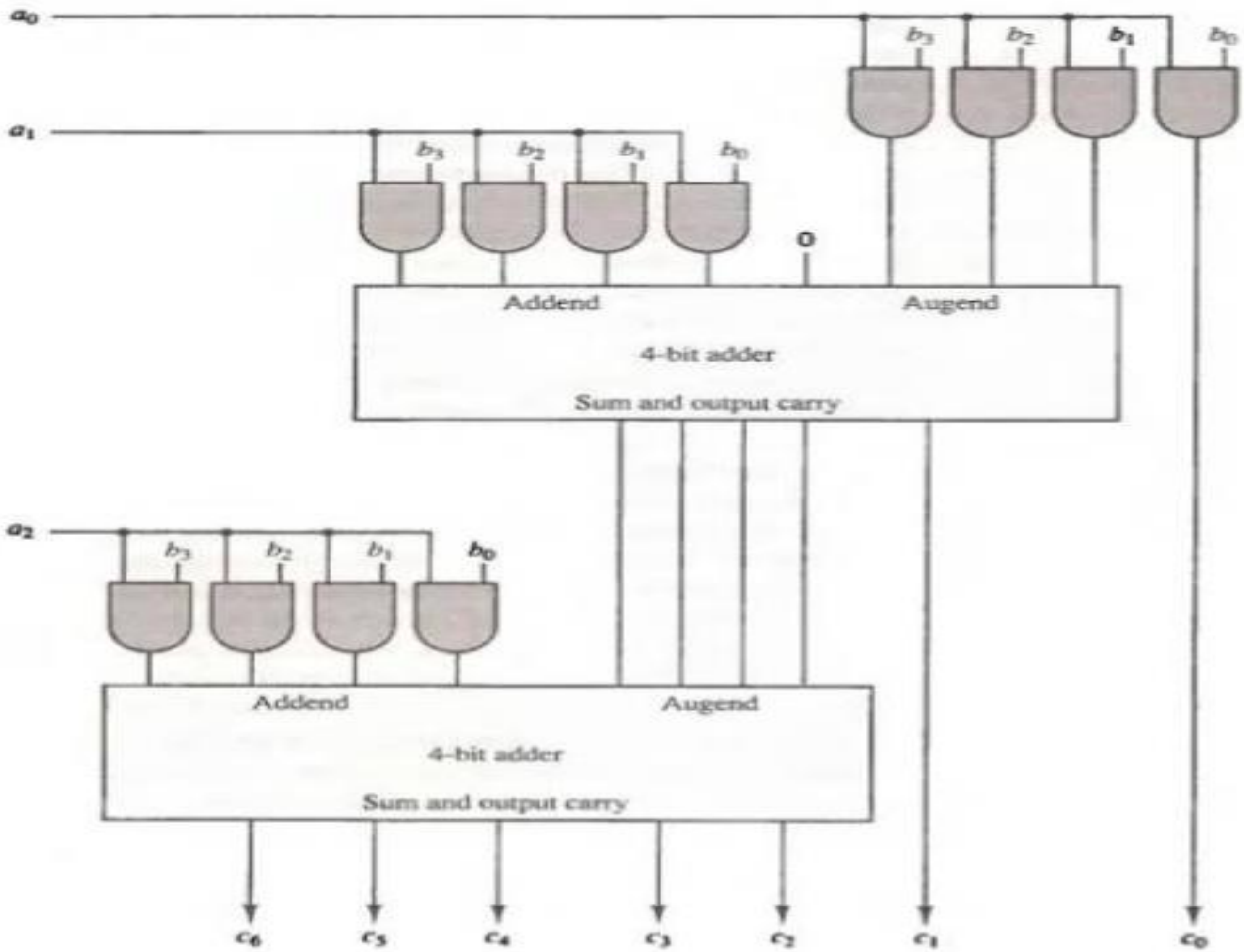


Figure 10-10 4-bit by 3-bit array multiplier.

Division of Signed magnitude Data

- Division of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations.
- Binary division is simpler than decimal division because 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. The division process is illustrated by a numerical example in Fig.

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 $A \geq B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geq B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geq B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

- The divisor B consists of five bits and the dividend A, of ten bits. The five most significant bits of the dividend are compared with the divisor.
- Since the 5-bit number is smaller than B, we try again by taking the six most significant bits of A and compare this number with B. The 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. The divisor is then shifted once to the right and subtracted from the dividend.
- The difference is called a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1.

- The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Note that the result gives both a quotient and a remainder.

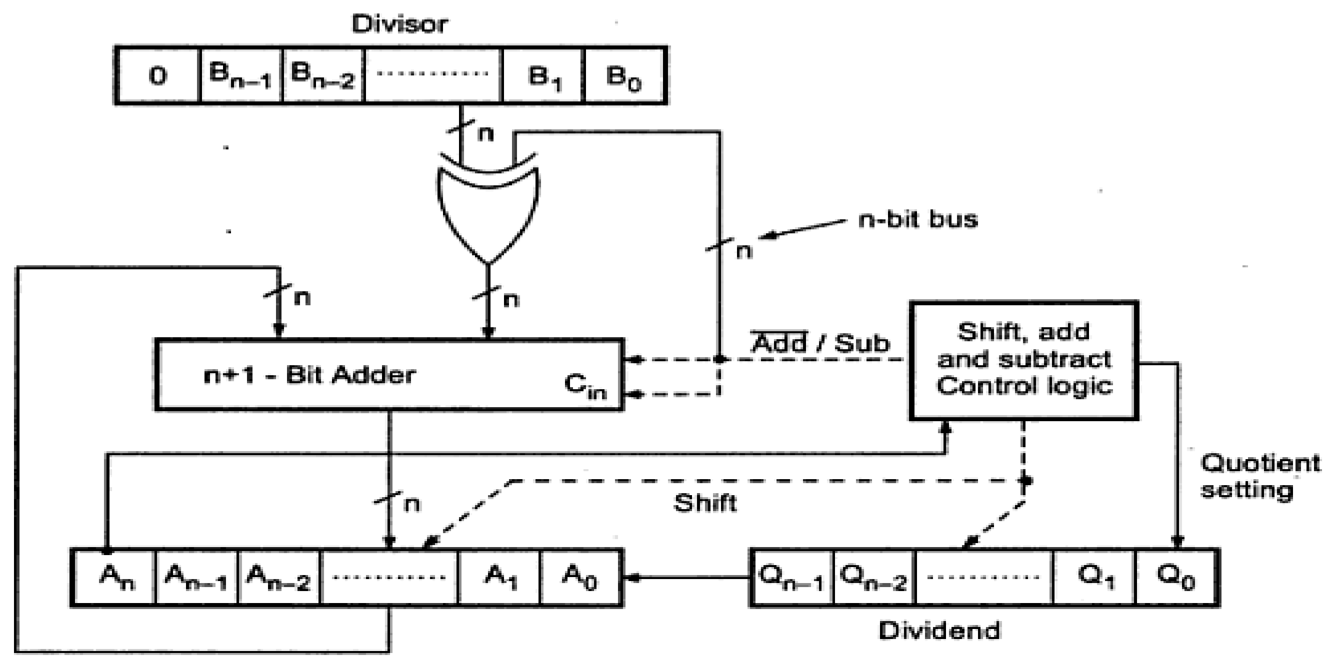


Fig. 4.22 Hardware to implement binary division

Hardware Implementation for Signed-Magnitude Data

- When the division is implemented in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, the dividend, or partial remainder, is shifted to the left, thus leaving the two numbers in the required relative position.
- Subtraction may be achieved by adding A to the 2's complement of B. The information about the relative magnitudes is then available from the end-carry. The hardware for implementing the division operation is identical to that required for multiplication. Register EAQ is now shifted to the left with 0 inserted into Q_n and the previous value of E lost. The numerical example is repeated in Fig. to clarify the proposed division process.

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

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

- The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. The information about the relative magnitude is available in E. If E = 1, it signifies that A B. A quotient bit 1 is inserted into Q, and the partial remainder is shifted to the left to repeat the process. If E = 0, it signifies that A < B so the quotient in Q. remains a 0 (inserted during the shift). The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed.

- Note that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and the final remainder is in A. Before showing the algorithm in flowchart form, we have to consider the sign of the result and a possible overflow condition. The sign of the quotient is determined from the signs of the dividend and the divisor. If the two signs are alike, the sign of the quotient is plus. If they are unlike, the sign is minus. The sign of the remainder is the same as the sign of the dividend.

Divide Overflow

- The division operation may result in a quotient with an overflow. This is not a problem when working with paper and pencil but is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length.
- To see this, consider a system that has 5-bit registers. We use one register to hold the divisor and two registers to hold the dividend. From the example above we note that the quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor. The quotient is to be stored in a standard 5-bit register, so the overflow bit will require one more flip-flop for storing the sixth bit.

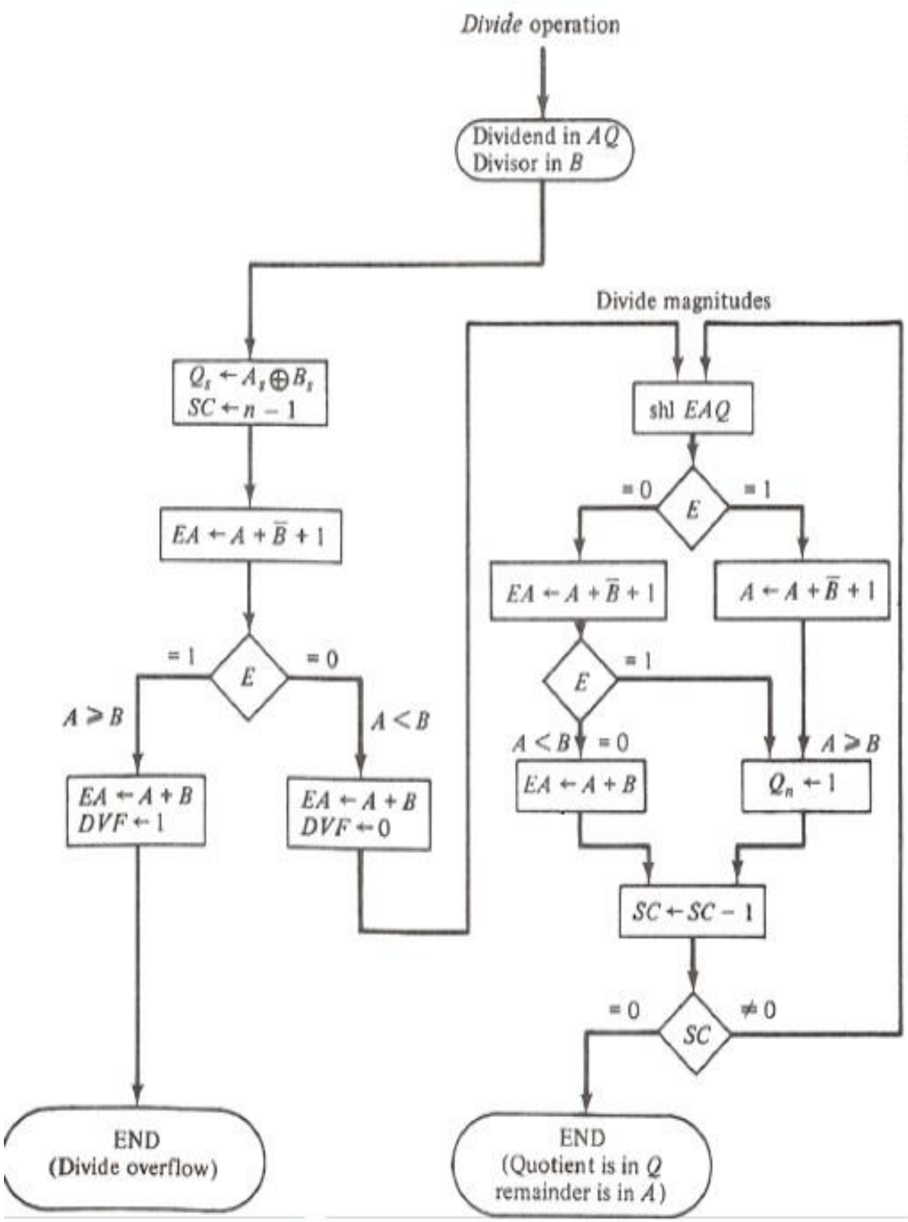
- **This divide-overflow condition must be avoided in normal computer operations because the entire quotient will be too long for transfer into a memory unit** that has words of standard length, that is, the same as the length of registers. Provisions to ensure that this condition is detected must be included in either the hardware or the software of the computer, or in a combination of the two.
- **When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows:** A divide-overflow condition occurs if the high-order half bits of the dividend constitute a number greater than or equal to the divisor.

- Another problem associated with division is the fact that a division by zero must be avoided. The divide-overflow condition takes care of this condition as well. This occurs because any dividend will be greater than or equal to a divisor which is equal to zero. Overflow condition is usually detected when a special flip-flop is set**. We will call it a divide-overflow flip-flop and label it DVF.**
- The occurrence of a divide overflow can be handled in a variety of ways. In some computers it is the responsibility of the programmers to check if DVF is set after each divide instruction. They then can branch to a subroutine that takes a corrective measure such as renting the data to avoid overflow.

- In some older computers, the occurrence of a divide overflow stopped the computer and this condition was referred to as a **divide stop.**
- Stopping the operation of the computer is not recommended because it is time consuming. The procedure in most computers is **to provide an interrupt request when DVF is set**. The interrupt causes the computer to suspend the current program and branch to a service routine to take a corrective measure. The most common corrective measure is to remove the program and type an error message explaining the reason why the program could not be completed. It is then the responsibility of the user who wrote the program to rescale the data or take any other corrective measure. The best way to avoid a divide overflow is to use floating-point data.

Hardware Algorithm

- The hardware divide algorithm is shown in the flowchart below. The dividend is in A and Q and the divisor in B. The sign of the result is transferred into Qs to be part of the quotient.
- A constant is set into the sequence counter SC to specify the number of bits in the quotient. As in multiplication, we assume that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.



- B: Divisor, AQ: Dividend
- If A>=B (oh yes, magnitudes are compared subtracting one from another and testing E flip-flop), DVF is set and operation is terminated prematurely. If A<B, no overflow and dividend is restored by adding B to A (since B was subtracted previously to compare magnitudes).
- Division starts by left shifting AQ (dividend) with high order bit shifted to E. Then E=1, EA>B so B is subtracted from EA and Q_n is set to 1. If E=0, result of subtraction is stored in EA, again E is tested. E=1 signifies A>=B, thus Q_n is set to 1 and E=0 denotes A<B, so original number is **restored** by adding B to A and we leave 0 in Q_n.
- Process is repeated again with register A holding partial remainder. After n-1 times Q contains magnitude of Quotient and A contains remainder. Quotient sign in Q_s and remainder sign in A_s.

- A divide-overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A. If A $\geq$B, the divide-overflow flip-flop DVF is set and the operation is terminated prematurely. If A <B, no divide overflow occurs so the value of the dividend is restored by adding B to A.

The division of the magnitudes starts by shifting the dividend in *AQ* to the left with the high-order bit shifted into *E*. If the bit shifted into E is 1, we know that EA > *B* because *EA* consists of a 1 followed by *n* -1 bits while *B* consists of only *n* - 1 bits. In this case, *B* must be subtracted from *EA* and 1 inserted into Q_n for the quotient bit. Since register *A* is missing the high-order bit of the dividend (which is in E), its value is $EA - 2^{n-1}$. Adding to this value the 2's complement of *B* results in

$$(EA - 2^{n-1}) + (2^{n-1} - B) = EA - B$$

The carry from this addition is not transferred to E if we want E to remain a 1.

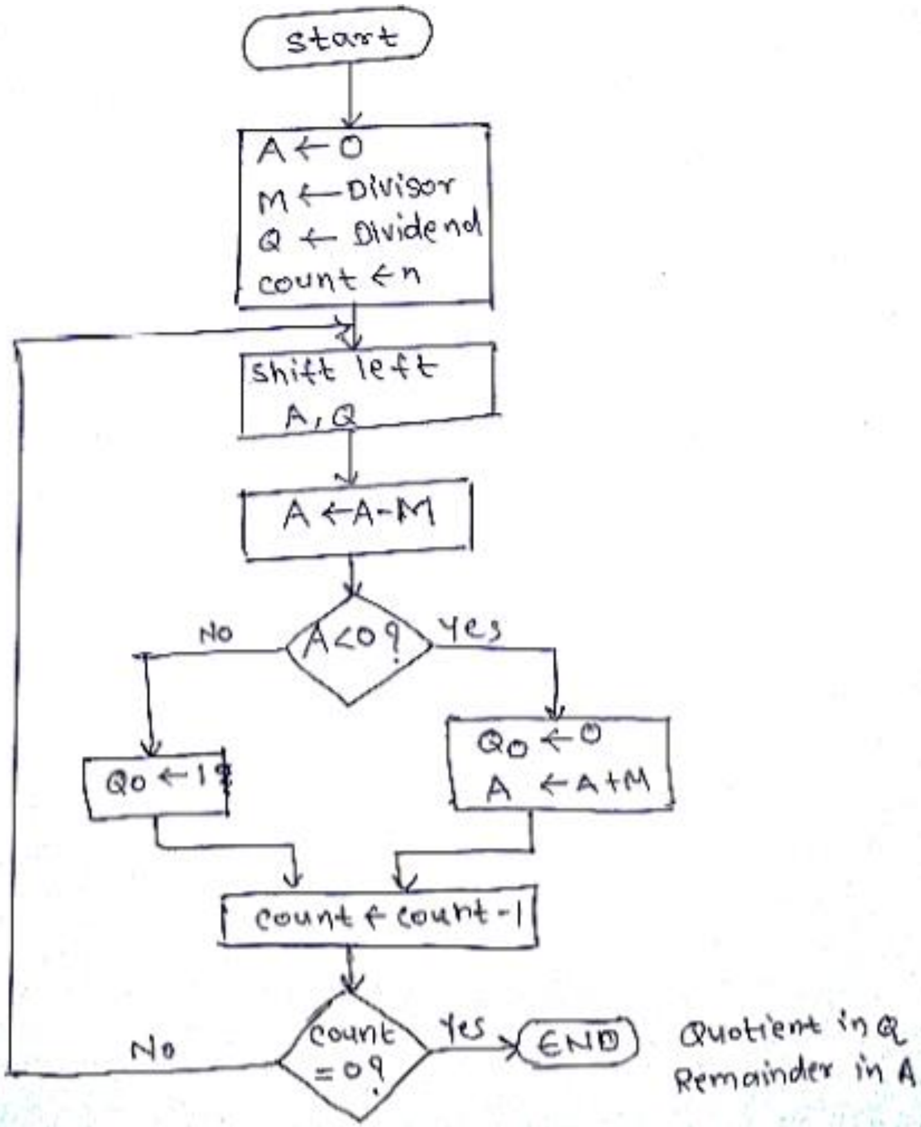
If the shift-left operation inserts a 0 into E, the divisor is subtracted by adding its 2's complement value and the carry is transferred into E. If E = 1, it signifies that $A \geq B$; therefore, Q_n is set to 1. If E = 0, it signifies that $A < B$ and the original number is restored by adding B to A. In the latter case we leave a 0 in Q_n (0 was inserted during the shift). This process is repeated again with register A holding the partial remain-der. After n - 1 times, the quotient magnitude is formed in register Q_s and the remainder is found in register A. The quotient sign is in Q, and the sign of the remainder in A_s is the same as the original sign of the dividend.

Restoring Method

- The hardware method just described is called the restoring method. The reason for this name is that the partial remainder is restored by adding the divisor to the negative difference.

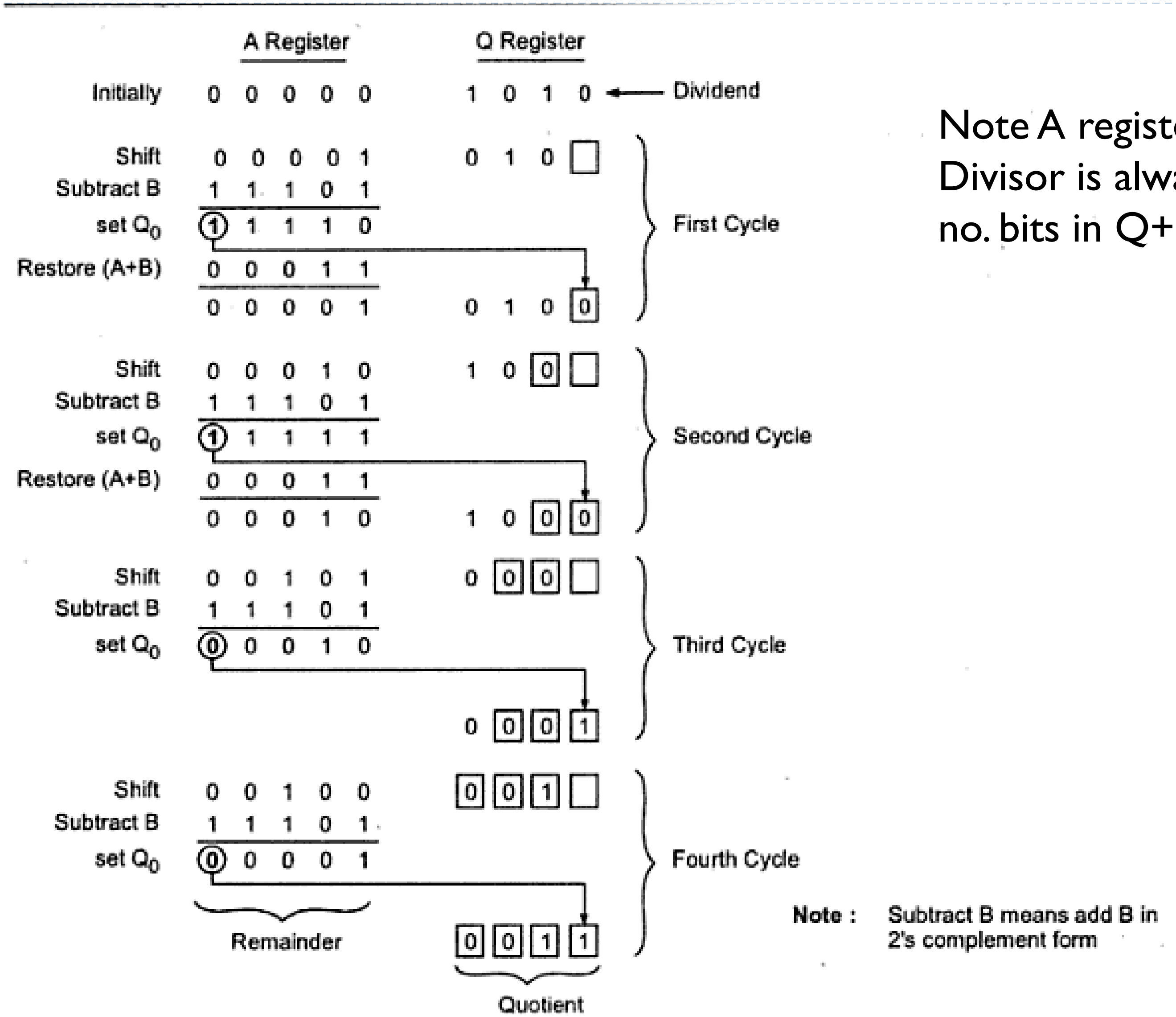
Division Operation Steps :

1. Shift A and Q left one binary position.
2. Subtract divisor from A and place answer back in A ($A \leftarrow A - B$).
3. If the sign bit of A is 1, set Q_0 to 0 and add divisor back to A (that is, restore A); Otherwise, set Q_0 to 1.
4. Repeat steps 1, 2, and 3 n times.



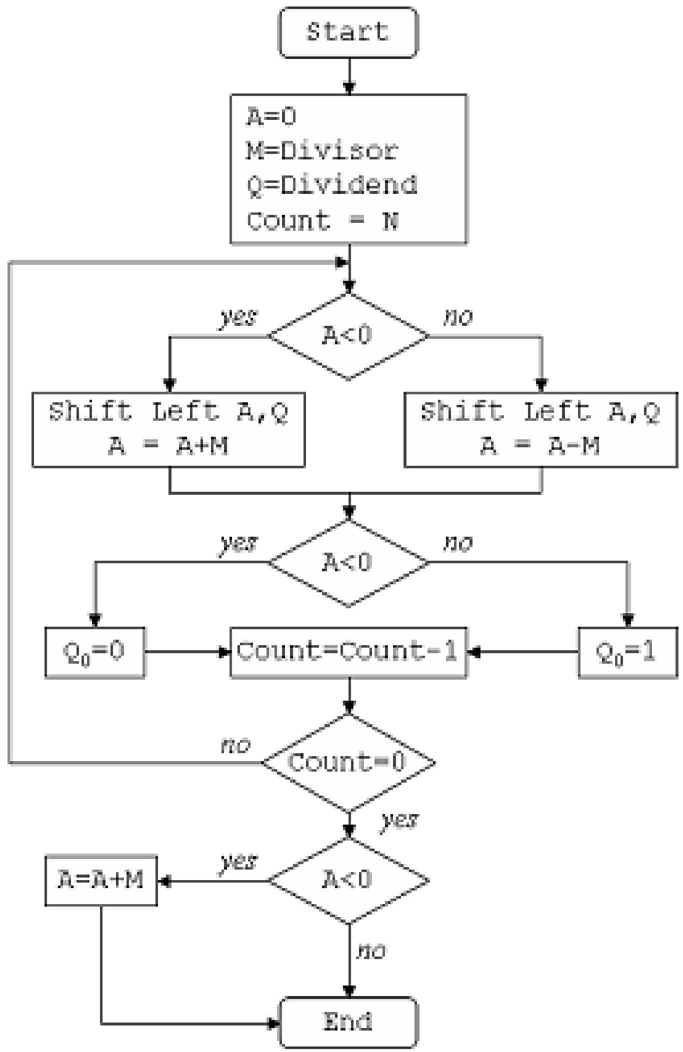
Note:
A<0 means to check the MSB 1 or 0.
MSB 1 represents negative number.
Ie. If MSB of A is 1 then yes condition
If MSB is 0 then No condition

Divide 10 by 3 i.e. Dividend(Q)=10 and Divisor (B)=3



Note A register and Divisor is always n+1 ie no. bits in Q+1

Non- Restoring Method



Divide 10 by 3 i.e. Dividend(Q)=11 and Divisor (B/M)=3

```
Dividend =11
Divisor  =3
-M =11101
```

N	M	A	Q	ACTION
4	00011	00000	1011	Start
		00001	011_	Left shift AQ
		11110	011_	A=A-M
3		11110	0110	Q[0]=0
		11100	110_	Left shift AQ
		11111	110_	A=A+M
2		11111	1100	Q[0]=0
		11111	100_	Left Shift AQ
		00010	100_	A=A+M
1		00010	1001	Q[0]=1
		00101	001_	Left Shift AQ
		00010	001_	A=A-M
0		00010	0011	Q[0]=1

```
Quotient  = 3 (Q)
Remainder = 2 (A)
```

Comparison and Non-Restoring Method

- Two other methods are available for dividing numbers, the comparison method (restoring method) and the non-restoring method. In the comparison method A and B are compared prior to the subtraction operation.
- Then if $A \geq B$, B is subtracted from A. If $A < B$ nothing is done. The partial remainder is shifted left and the numbers are compared again.
- The comparison can be determined prior to the subtraction by inspecting the end-carry out of the parallel-adder prior to its transfer to register E. In the non-restoring method, B is not added if the difference is negative but instead, the negative difference is shifted left and then B is added.

- In restoring the operations performed are A - B + B; that is, B is subtracted and then added to restore A. The next time around the loop, this number is shifted left (or multiplied by 2) and B subtracted again. This gives 2(A - B + B) - B = 2A - B.
- This result is obtained in the non-restoring method by leaving A - B as is. The next time around the loop, the number is shifted left and B added to give 2(A - B) + B = 2A - B, which is the same as before.

- Thus, in the non-restoring method, B is subtracted if the previous value of Q_n was a 1, but B is added if the previous value of Q_n was a 0 and no restoring of the partial remainder is required.
- This process saves the step of adding the divisor if A is less than B, but it requires special control logic to remember the previous result. The first time the dividend is shifted, B must be subtracted. Also, if the last bit of the quotient is 0, the partial remainder must be restored to obtain the correct final remainder.