CHAPTER 7
COMPUTER ARITHMETIC

Outline

- 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 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

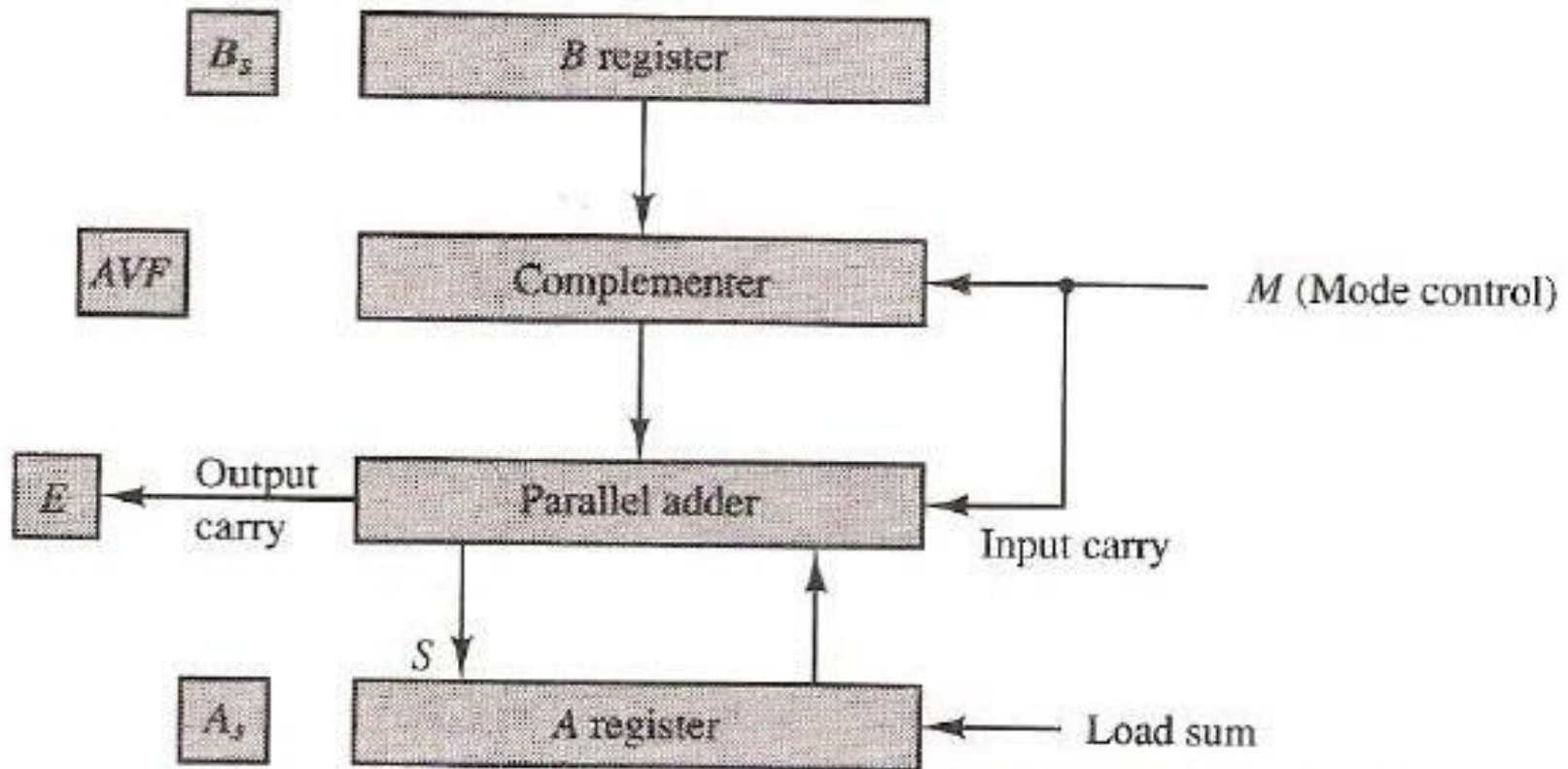| Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$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

Hardware Implementation

Fig: hardware for signed-magnitude addition and subtraction

Explanation

- 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.

Cont…

- Hardware above 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.

- 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.

cont..

- 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).

©सरोज थापा

Hardware Algorithm
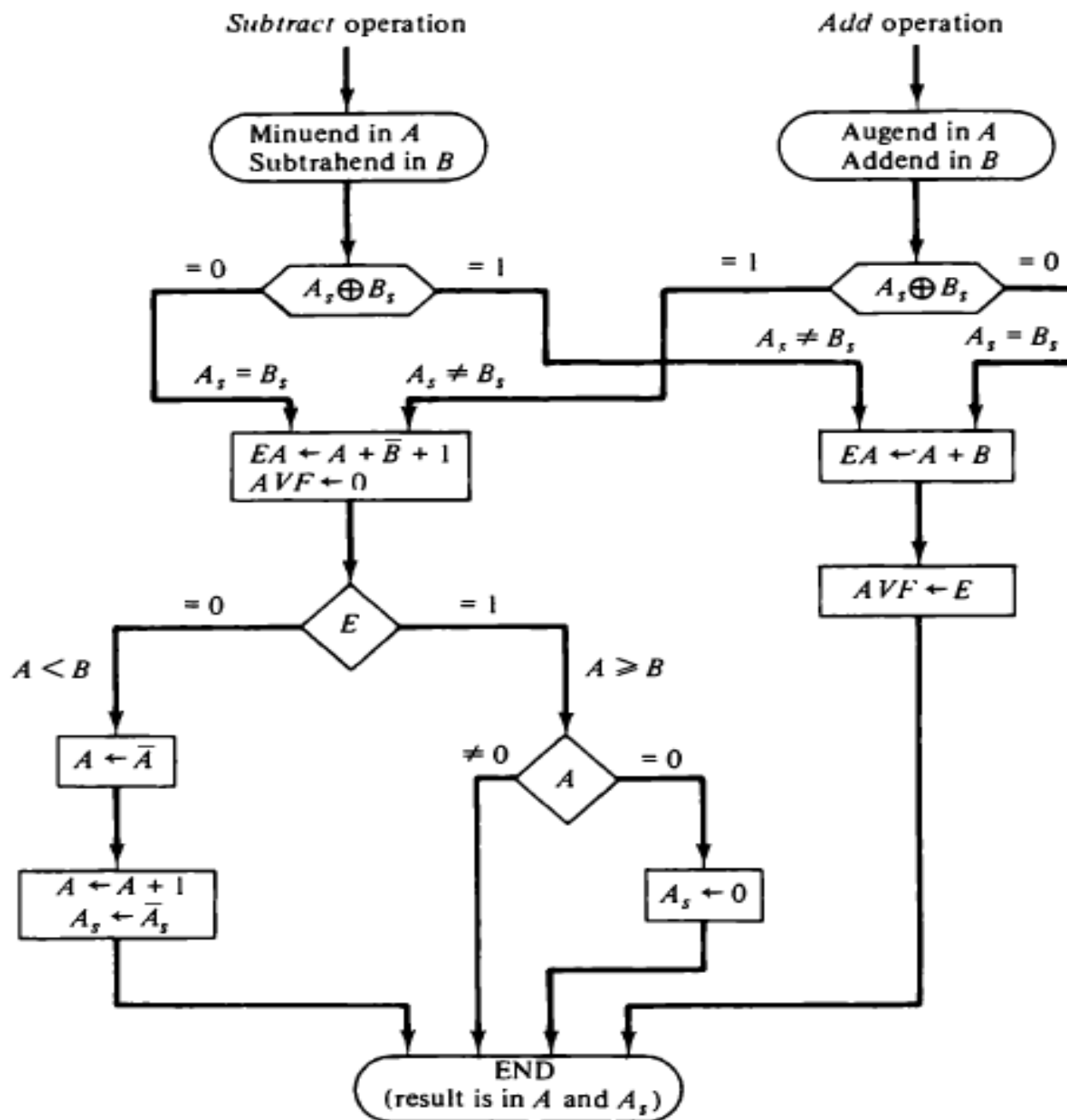

Figure 10-2 Flowchart for add and subtract operations.

Description

- The flowchart for the hardware algorithm is shown above.

- The two signs A_s and B_s 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 $EA \leftarrow A + B$, 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.

©सरोज थापा

Cont..

- 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.
- E=1 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.

Cont..

- 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 = 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

As = 0   A=0101101

Bs = 1   B=0010111

As xor Bs =1

EA=A + B' + 1 = 0101101 + 1101000 +1 = 10010110

AVF=0

=> E=1   A= 0010110

Result : As A= 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.
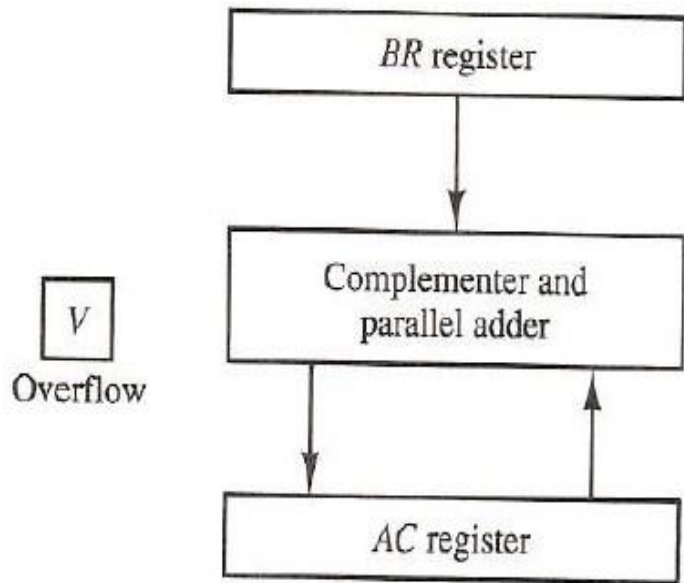
©सरोज थापा

Hardware

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.

©सरोज थापा

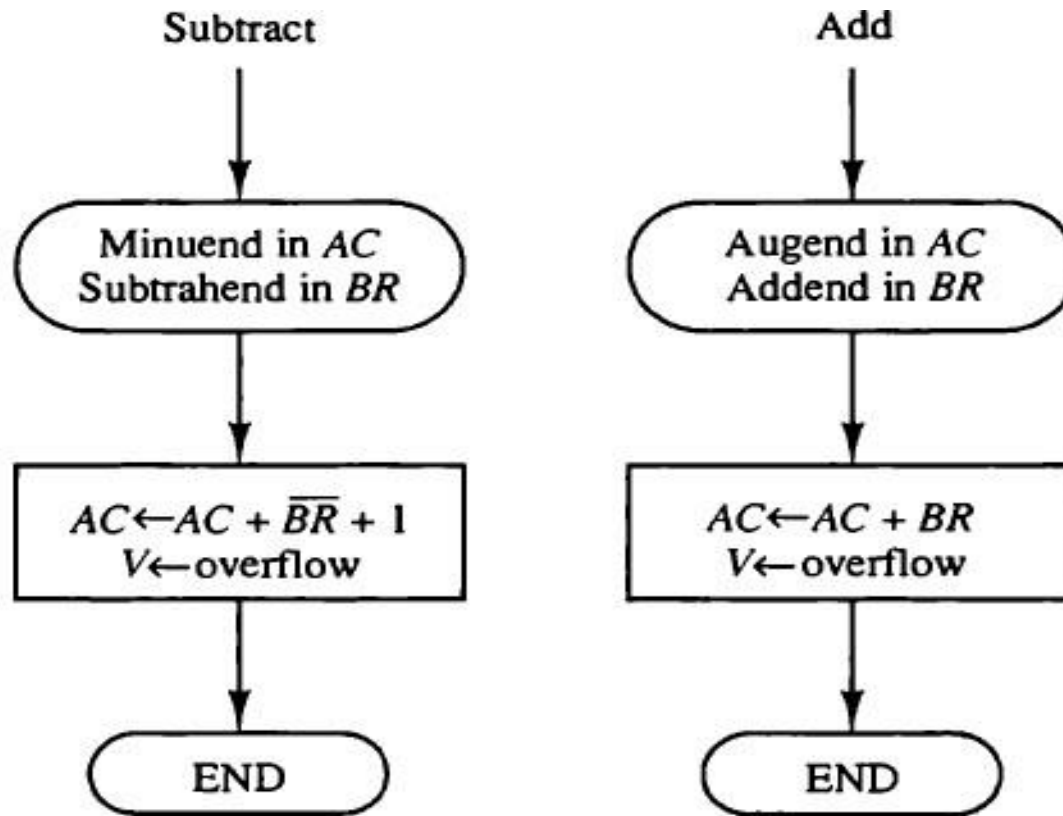# Hardware Algorithm

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

©सरोज थापा

. Explanation

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 XOR 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.

©सरोज थापा

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
```

©सरोज थापा

Cont…

- 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.

Hardware implementation of SIGNED MAGNITUDE MULTIPLICATION

Figure 10-5   Hardware for multiply operation.
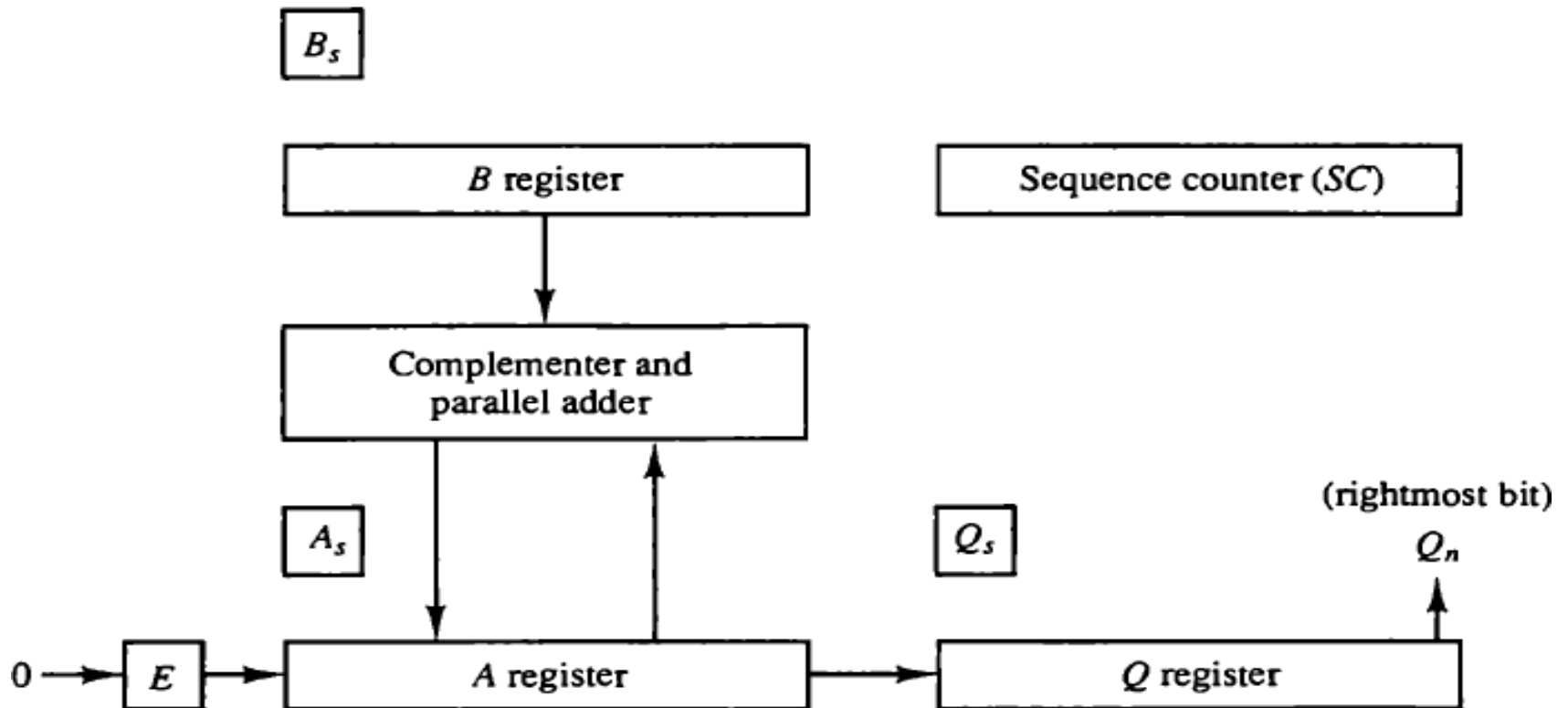
Cont..

- 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.

Flowchart of Multiplication:

Multiply Operation

Multiplicand in B
Multiplier in Q

As (sign of A) = Qs (sign of Q) **XOR** Bs (sign of B)
$A = 0$
$E = 0$
Sequence Counter (SC) = n (number of bits in Q)

Qn

$= 1$

$= 0$

$E A = A + B$

SHIFT RIGHT
E A Q
$SC = SC - 1$

END

$0 \#$

SC

$= 0$

©सराज थापा
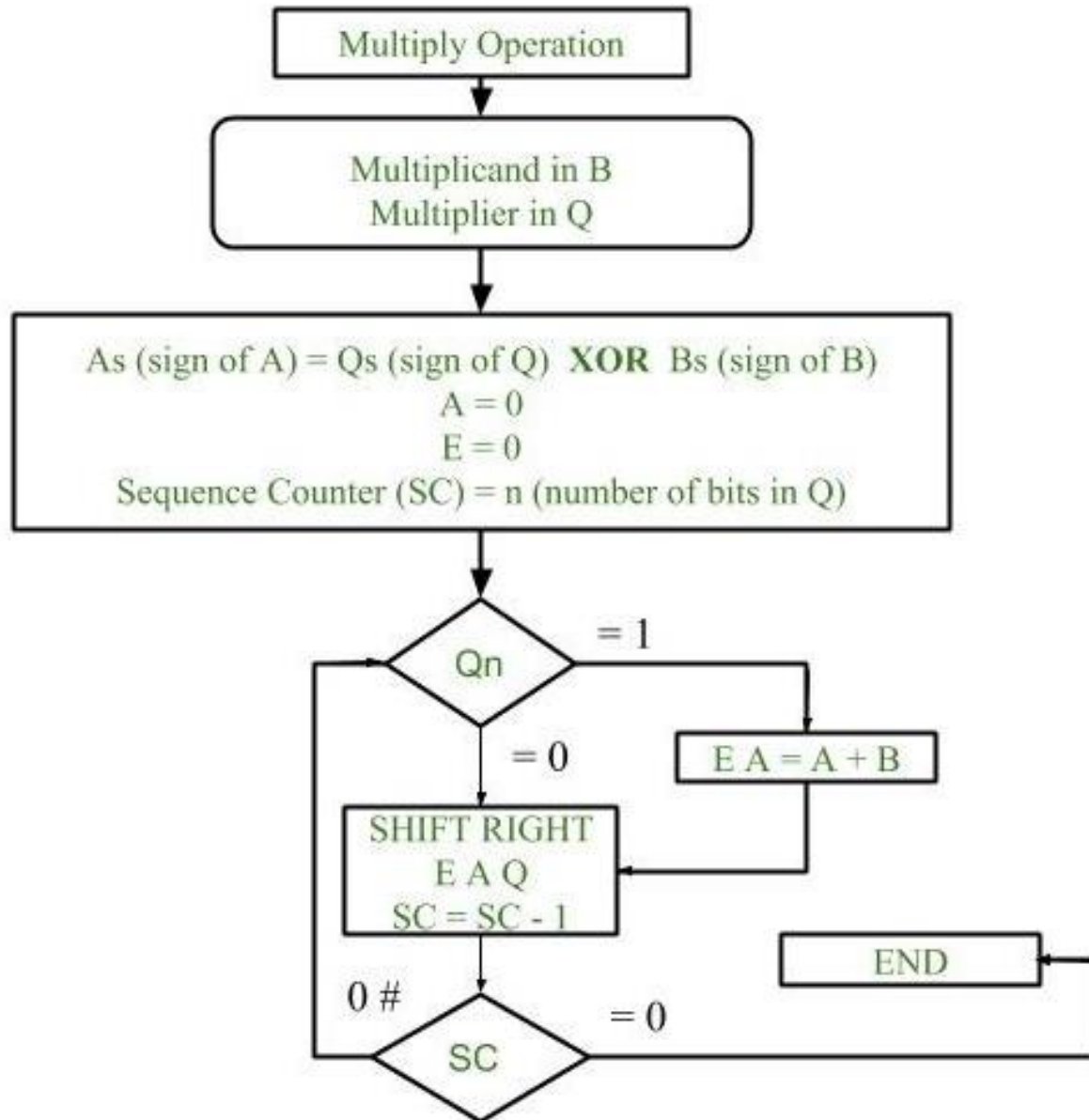
Cont…

- **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.

Cont..

- After the initialization, the low-order bit of the multiplier in Qn 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.

©सरोज थापा

EXAMPLE MULTIPLY 23*19 B=23 Q=19

Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q Qn = 1; add B	0	00000 10111	10011	101
First partial product Shift right EAQ	0 0	10111 01011	11001	100
Qn = 1; add B Second partial product	1	10111 00010		
Shift right EAQ	0	10001	01100	011
Qn = 0; shift right EAQ	0	01000	10110	010
Qn = 0; shift right EAQ	0	00100	01011	001
Qn = 1; add B Fifth partial product	0	10111 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

©सरोज थापा

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.

Hardware Implementation

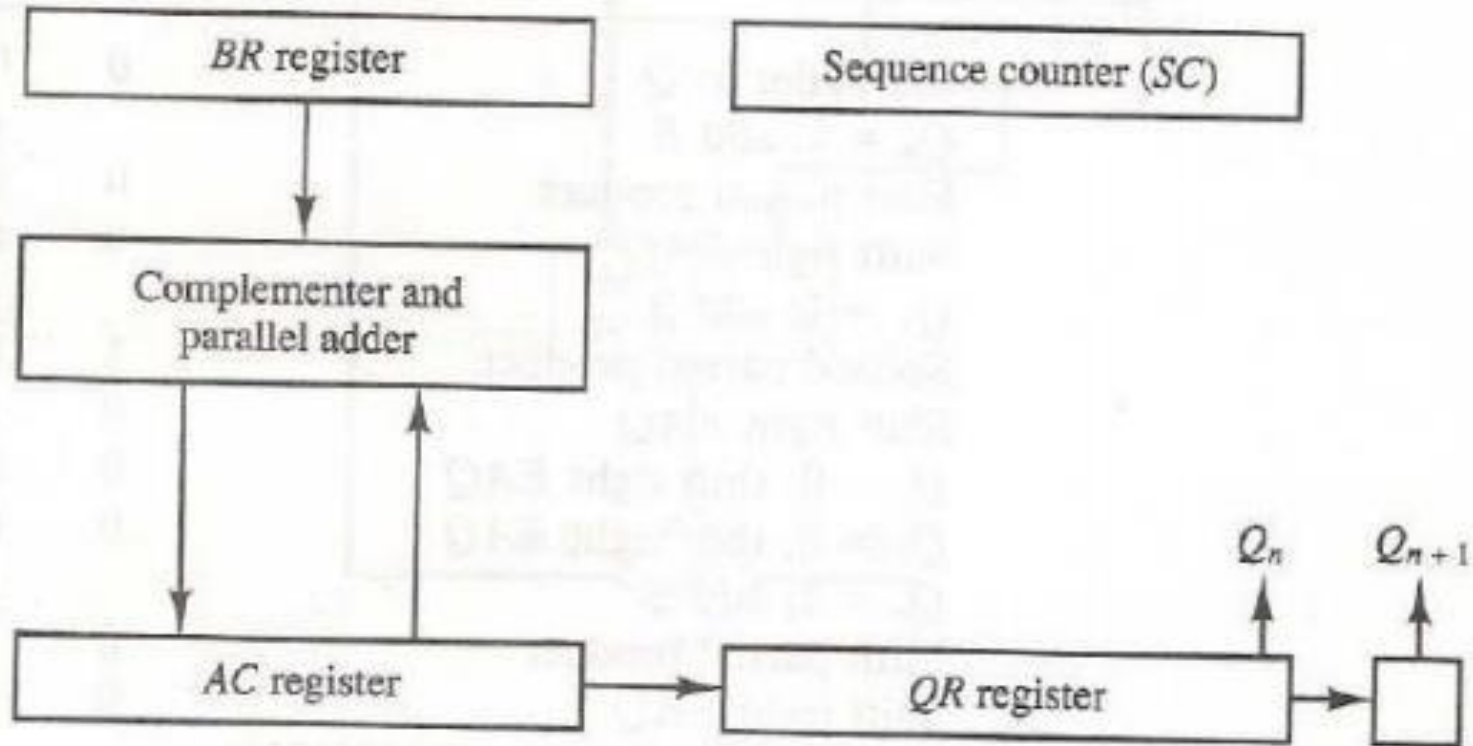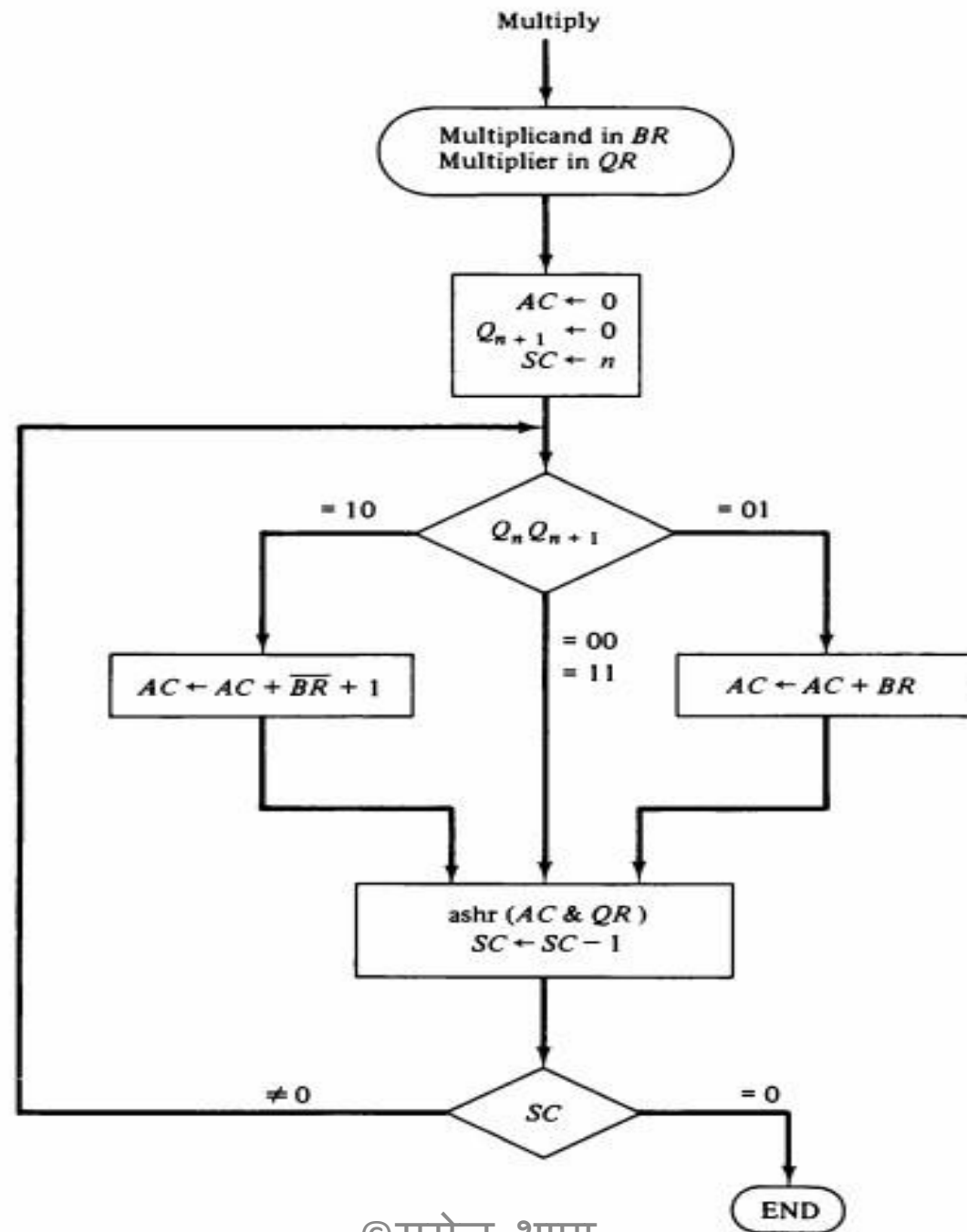Fig. Hardware for Booth Algorithm

©सरोज थापा

Cont..

- For hardware implementation it requires the configuration as shown in figure. It consists of 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.

©सरोज थापा

Hardware Algorithm

Multiply

Multiplicand in BR
Multiplier in QR

$AC \leftarrow 0$
$Q_{n+1} \leftarrow 0$
$SC \leftarrow n$

$Q_n Q_{n+1}$

= 10 → $AC \leftarrow AC + \overline{BR} + 1$

= 01 → $AC \leftarrow AC + BR$

= 00
= 11

ashr $(AC \& QR)$
$SC \leftarrow SC - 1$

SC

$\neq 0$

$= 0$ → END

©सरोज थापा

Description

- 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.

©सरोज थापा

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
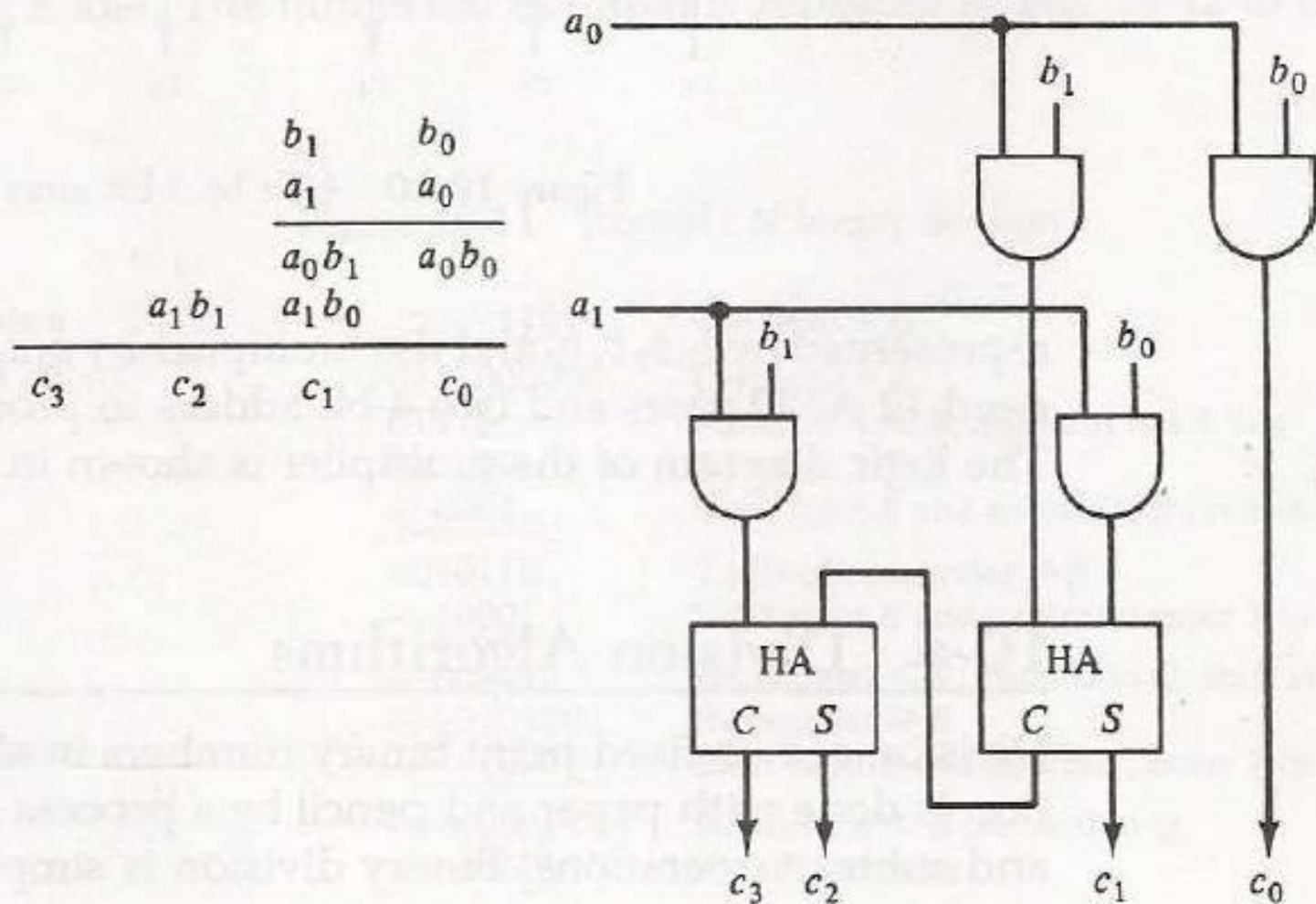
Multiply following using Booth Algorithm:

- -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.

©सरोज थापा

Figure 2-bit by 2-bit array multiplier.

$$
\begin{array}{cccc}
 & & b_1 & b_0 \\
 & & a_1 & a_0 \\
\hline
 & & a_0 b_1 & a_0 b_0 \\
 & a_1 b_1 & a_1 b_0 & \\
\hline
c_3 & c_2 & c_1 & c_0 \\
\end{array}
$$

Explanation

- 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_1 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.

©सरोज थापा

Cont..

- 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.

Division of Signed magnitude Data
[Restoring Method]

- 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.

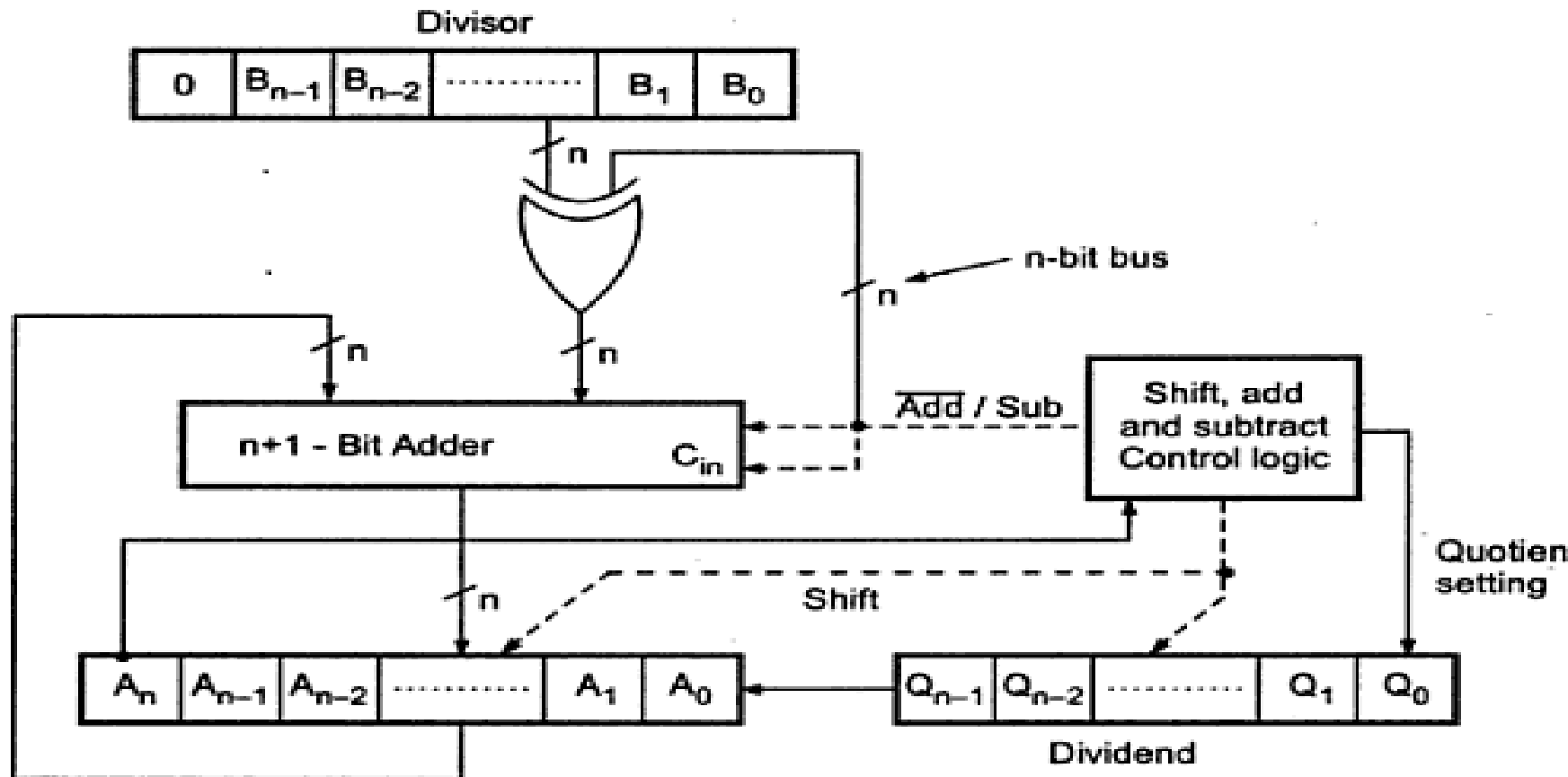# Hardware Implementation for Signed-Magnitude Data

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.
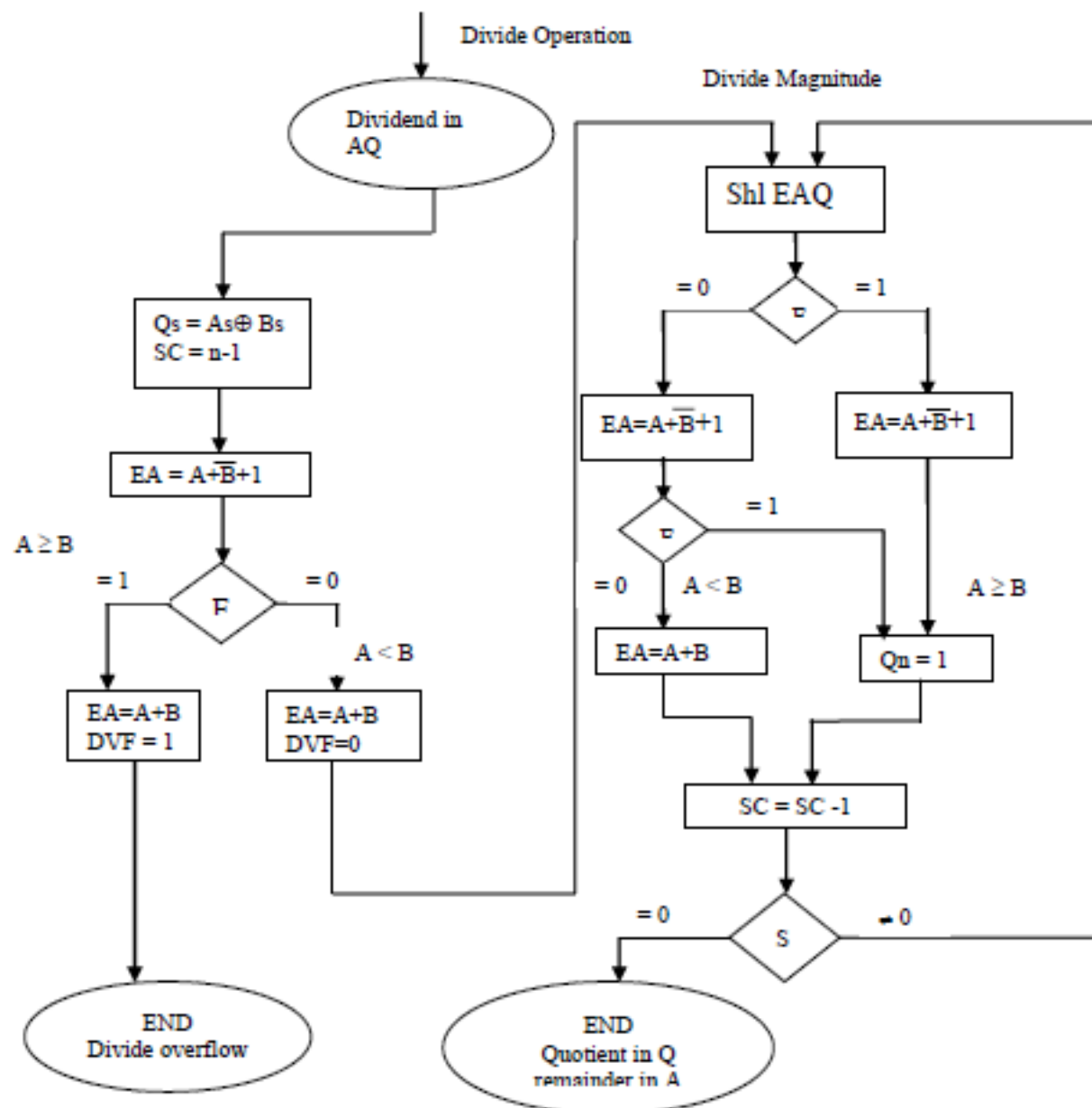
©सरोज थापा

Cont..

- Note that **while the partial remainder is shifted left, the quotient bits are shifted also and after n 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.

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.

©सरोज थापा

Division of Signed Magnitude Data

Divide Operation

Dividend in AQ

$Q_s = A_s \oplus B_s$
$SC = n-1$

$EA = A + \overline{B} + 1$

$A \geq B$

F
= 1
= 0

$A < B$

EA = A + B
DVF = 1

EA = A + B
DVF = 0

END
Divide overflow

Divide Magnitude

Shl EAQ

E
= 0
= 1

$EA = A + \overline{B} + 1$

$EA = A + \overline{B} + 1$

E
= 1

= 0 $A < B$

$A \geq B$

EA = A + B

$Q_n = 1$

SC = SC - 1

S
= 0
≠ 0

END
Quotient in Q
remainder in A

Example: Divide 15/3 using Restoring Division Method
Dividend   (Q):1111
AQ=00001111
B=0011
B'+1=1100+1=1101

Operation	E	A	Q	SC
	0	0000	1111	4
Shl EAQ A+B'+1 E=0,Qn=0 So A+B	0 0 1	0001 +1101 1110 +0011 0001	1110	
Shl EAQ A+B'+1 E=1,SET Qn=1	0 1	0011 +1101 0000	1100 1101	3
Shl EAQ A+B'+1 E=0,Qn=0 So A+B	0 0 1	0001 +1101 1110 +0011 0001	1010	2
Shl EAQ A+B'+1 E=1,SET Qn=1	0 1	0011 +1101 0000 Remainder	0100 0101 Quotient	1

What about Non restoring Division method???

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.

Cont..

- 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.

Cont..

- 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.

Divide Overflow

- The division operation may result in a quotient with an overflow.

- 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.

Explanation

- **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.

©सरोज थापा

Cont..

- 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.

Cont..

- 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.

END OF CHAPTER 7

END OF THIS COURSE

THANK YOU

©सरोज थापा