Addition and Subtraction:→

* Most computers use the signed 2's complement representation when performing arithmetic operations with integers.
* For floating-point operations, most computers use the signed-magnitude representation for mantissa.

Addition and Subtraction with Signed-Magnitude Data:→

* We designate the magnitude of the two numbers by A and B. When the signed numbers are added or subtracted, we find that there are eight different condition to consider, depending on the sign of the numbers and operation performed. These condition are showed in the below (fig) Table.

Table: 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)	-(B-A)	+(A-B)
(+A)-(+B)				
(+A)-(-B)	+(A+B)			
(-A)-(+B)	-(A+B)			
(-A)-(-B)		-(A-B)	+(B-A)	+(A-B)

* The algorithm for addition and subtraction are derived from the table and can be stated as follows (The word inside parenthesis should be used for the subtraction algorithm)

Addition (Subtraction) Algorithm:→

* When the signal of A and B are identical (different), add the two magnitude and attach the sign of A to the result.
* When the sign of A and B are different (identical), compare the magnitudes and subtract the smaller number from the larger.
* Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B.
* If the two magnitudes are equal, subtract B from A and make the sign of the result positive.
* The two algorithms are similar except for the sign comparison. The procedure to be followed for identical signs in the addition algorithm is the same as for the different signs in the subtraction algorithm, and vice versa.

Hardware Implementation:→

* Let A and B be two registers that holds the two numbers, and As, and Bs be two F-F that holds the corresponding signs.

* The result of the operation may be transferred to a third register; however, a saving is achieved if the result is transferred into A and Ap. Then A and Ap together form an accumulator register.
* First, a parallel-adder is needed to perform the micro operation A+B.
* Second the comparator circuit is needed to establish if A > B, A = B, or A < B.
* Third two parallel-subtractor circuits are needed to perform the micro-operation A-B and B-A.
* The sign relationship can be determined from an Ex-OR gate with As and Bs as i/ps.
* Careful investigation of the alternatives reveals that the use of 2's complement for subtraction and comparision is an efficient procedure that requires only one adder and a complementor.
* Figure on the next page shows the HW for implementing the addition and subtraction operation.
* It consists of register A and B and sign F-F As and Bs. Subtraction is done by adding A to the 2's complement of B. The O/P carry is transferred to F-F E, where it can be checked to determine the relative magnitude of the

* The result of the operation may be transferred to a third register; however, a saving is achieved if the result is transferred into A and Ap. Then A and Ap together form a accumulator register.
* First, a parallel-adder is needed to perform the micro operation A+B.
* Second the comparator circuit is needed to establish if A > B, A = B, OR A < B.
* Third two parallel-subtractor circuits are needed to perform the micro-operation A-B and B-A.
* The sign relationship can be determined from an Ex-OR gate with As and Bs as i/ps.
* Careful investigation of the alternatives reveals that the use of 2's complement of for subtraction and comparision is an efficient procedure that requires only one adder and a complementor.
* Figure on the next page shows the HW for implementing the addition and subtraction operation.
* It consists of register A and B and sign F-F As and Bs. Subtraction is done by adding A to the 2's complement of B. The O/P carry is transferred to F-F E, where it can be checked to determine the relative magnitude of the

of the two numbers is. The add-overflow F-F AVF holds the overflow bit when A and B are added.

* The A Register provides other Micro operations that may be needed when we specify the sequence of steps in the algorithm.

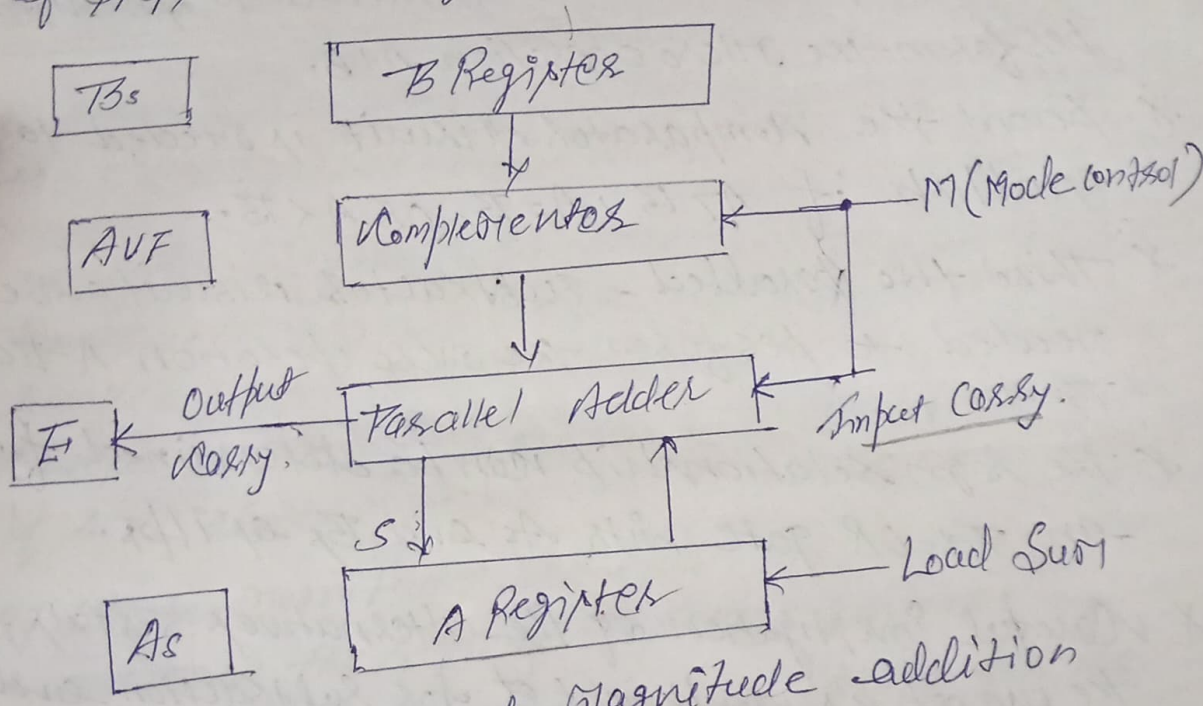


Fig: HW for signed-Magnitude addition and Subtraction.

* The addition of A plus B is done through the parallel adder. The S (sum) o/p of the adder is applied to the i/p of the A Register.
* The Complementer consists of Ex-OR gates and the parallel adder consists of full-adder circuit.
* The M signal is also applied to the i/p carry of the adder. When M=0 the o/p of B is transferred to the adder, the i/p carry is 0, and the o/p of the adder is equal to the sum A+B.

* When M=1, the 1's complement of B is applied to the adder, the i/p carry is 1, and output S = A+B̄+1. This is equal to A + 2's complement of B, which is equal to the subtraction A-B.

Hardware Algorithm:→

* The flowchart for the HW algorithm is presented in the below fig.
* The two signs As and Bs are compared by an Ex-OR gate. If the o/p is of the gate is 0, the signs are identical; if it is 1, the signs are different.
* The magnitudes are added with a microoperation EA ← A+B, where EA is a register that combines E and A.
* The carry in E is transferred into the add-overflow ~~with~~ A F-F AVF.
* The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the number are subtracted so AVF is cleared to 0.
* A 1 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 micro-operation A ← Ā+1.
* However, we assumes that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two micro-operations.

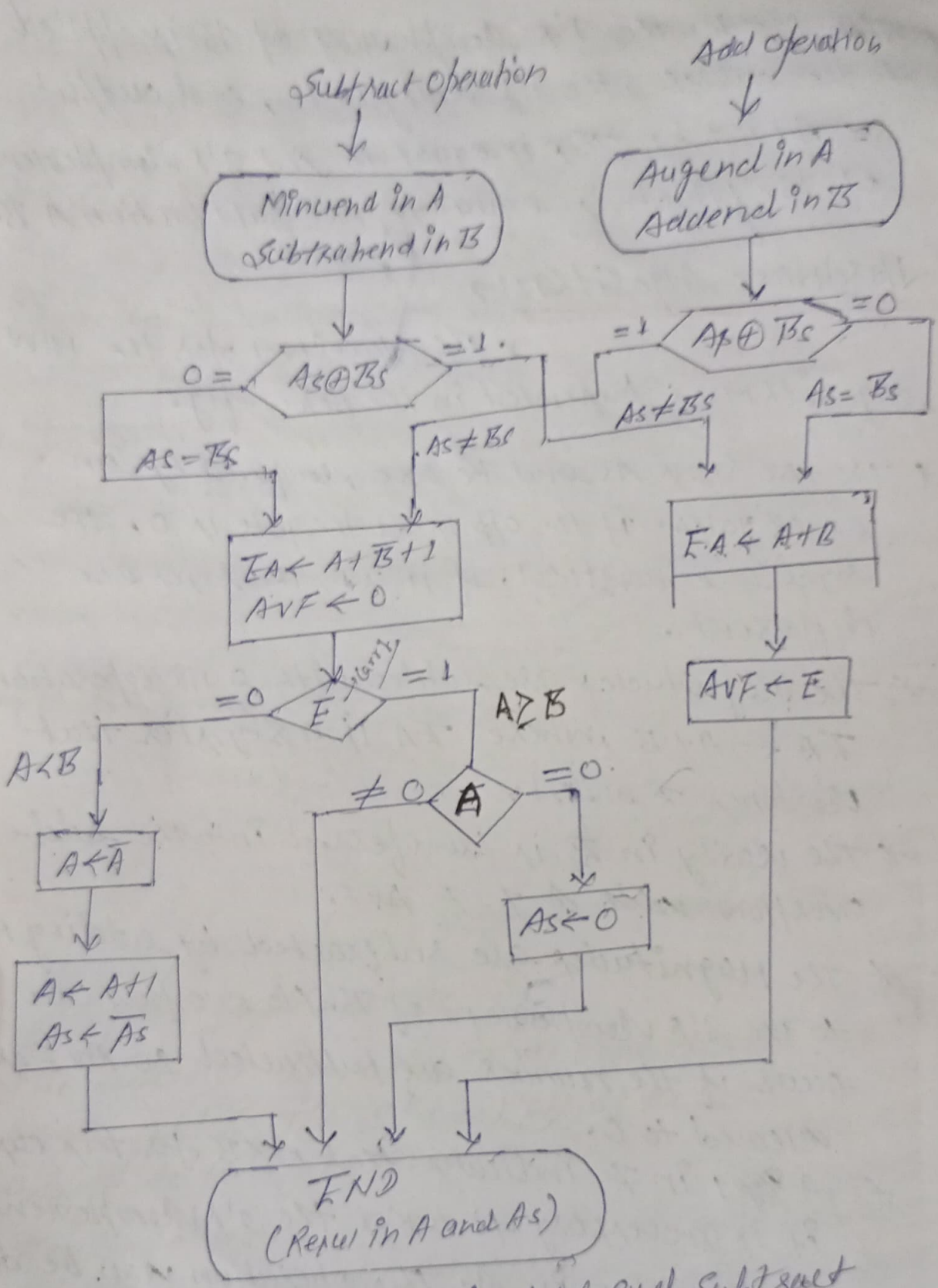


Fig: Flow chart for Add and Subtract

* In other parts of the flowchart, the sign of the result is the same as the sign of A, so no change in As is required.

* However, when A < B, the sign of the result is the complement of the original sign of A. It is necessary to to complement As to obtain the correct sign.
* The Final Result is found in the Register A and its sign in As. The value of AVF provides an Overflow Indication. The final value of E is immaterial

Addition and Subtraction with Signed 2's Complement Data:→

* The addition of two numbers in signed-2's complement form consists of adding the number with the sign bits treated the same as the other bits of the number.
* A carry-out of the sign-bit position is discarded. The subtraction consists of first taking the 2's complement of the subtrahend and then adding it to the Minuend.
* When two numbers of n digits each are added and the sum occupies n+1 digits, we say that an overflow occured. An overflow can be detected by inspecting the last two carries out of the addition. When the two carries are applied to an Ex-OR gate, the overflow is detected when the o/p of the gate is equal to 1.
* The Register configuration for the HW HW Implementation is shown in fig in the next page.

* We name the A register AC and the B register BR. The leftmost bit in AC and BR represent the sign bits of the number.
* The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder
* The overflow flip-flop V is set to 1 if there is an overflow. The output carry in this case is discarded.

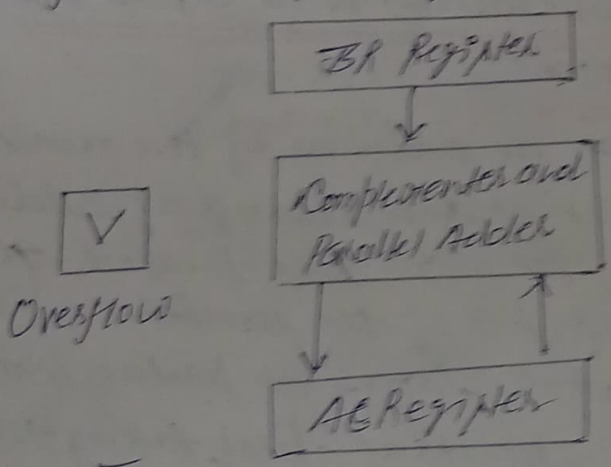
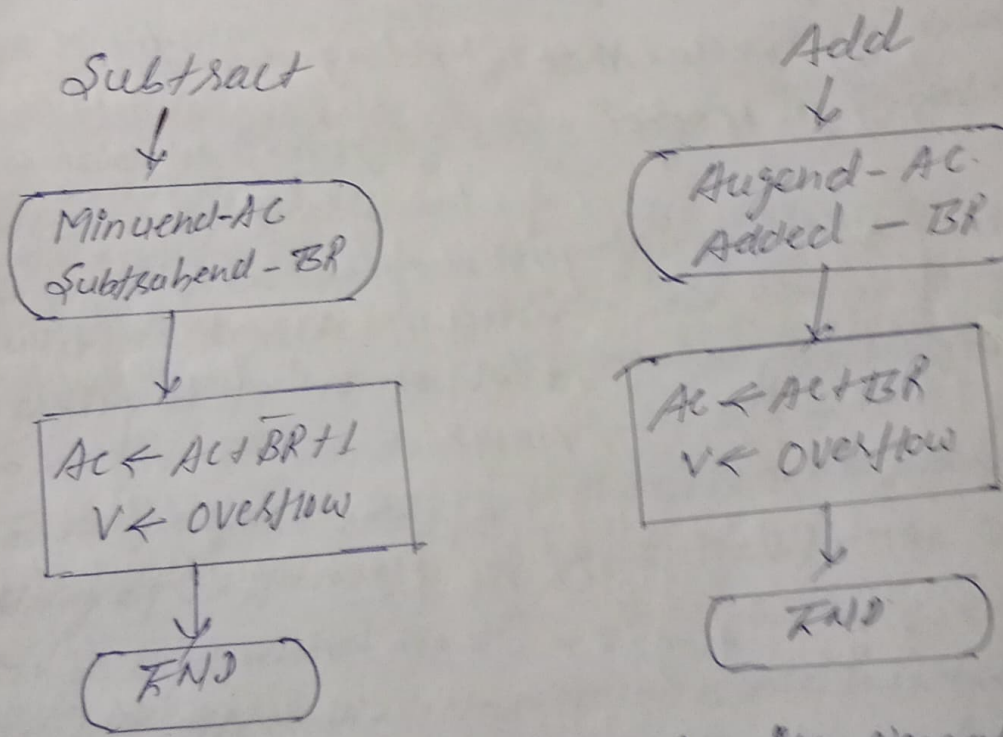


Fig: HW for signed 2's complement addition and subtraction.

* The algorithm for adding and subtracting two binary number in signed 2's complement representation is shown in the flowchart in the next page.

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 EX-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.
* An overflow must be checked during this operation because the two numbers added could have the same sign.
* The programmer must realize that if an overflow occurs, there will be an erroneous result in the AC register.



* Comparing this algorithm with the signed-magnitude counterpart, we noted that it is much simpler to add and subtract numbers if negative numbers are maintained in signed-2's complement representation. That's why most of computers adopt this representation.

Multiplication Algorithm:→

* Multiplication of two fixed-point binary numbers in signed-Magnitude representation is done with paper and pencil by a process of successive shift and add operation.

```
    23                10111   Multiplicand
  * 19                10011   Multiplier
  ----              ---------
                      10111
                     10111
                    00000
                   00000
                  10111
  ----           ----------
  437             111010101   Product
```

Booth Multiplication Algorithm:→

* Booth algorithm gives a procedure for multiplying binary integers in signed-2's complement representation.
* 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 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$.
* For example, the binary number 001110 (+14) has a string of 1's from 2^3 to 2^1 ($k=3, m=1$). The number can be represented as $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$. Therefore, the multiplication $M \times 14$, where M is the multiplicand and 14 the multiplier, can be done as $M \times 2^4 - M \times 2^1$.
* Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

* Booth algorithm requires examination of the multiplier bits and shifting of the partial product.
* Prior to the 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 1 in a string of 1's in the multiplier.
2) The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
3) The partial product does not change when the multiplier bit is identical to the previous multiplier bit.
* The algorithm works for positive or negative multipliers in 2's complement representation. This is because a negative multiplier ends with a string of 1's and the last operation will be a subtraction of the appropriate weight.
* The hardware implementation of Booth algorithm requires the register configuration shown in fig on the next page.
* Initially, the multiplicand is in register BR and the multiplier in QR, partial result in AC register. Qn designates the least significant bit of the multipliers in register QR. An extra F-F Qn is appended to QR to facilitate a double bit inspection of the multipliers.

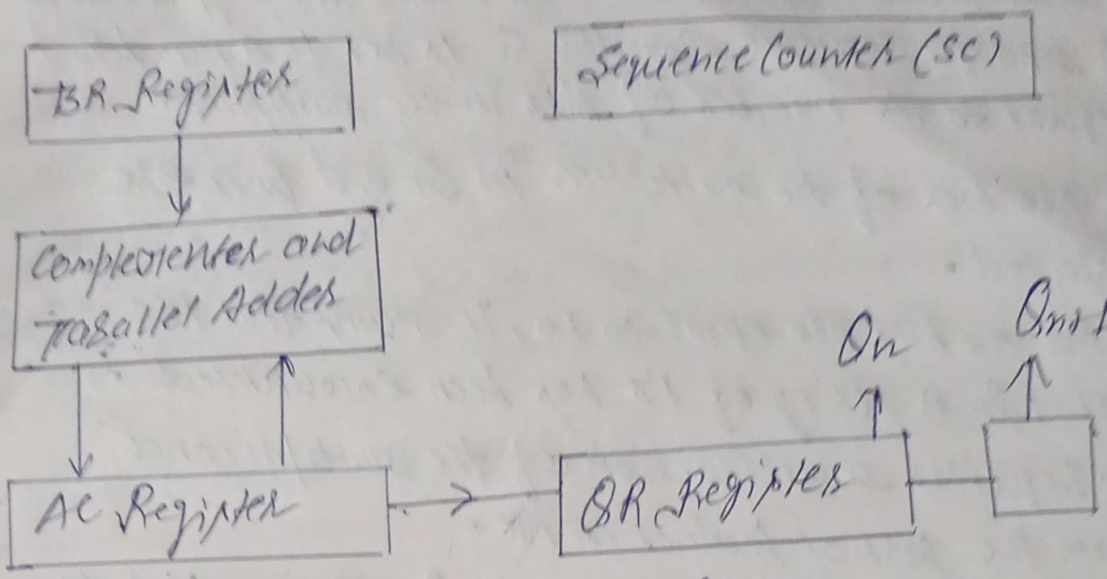


Fig: HW for Booth Algorithm

* Below fig. shows the flowchart for Booth Algorithm.

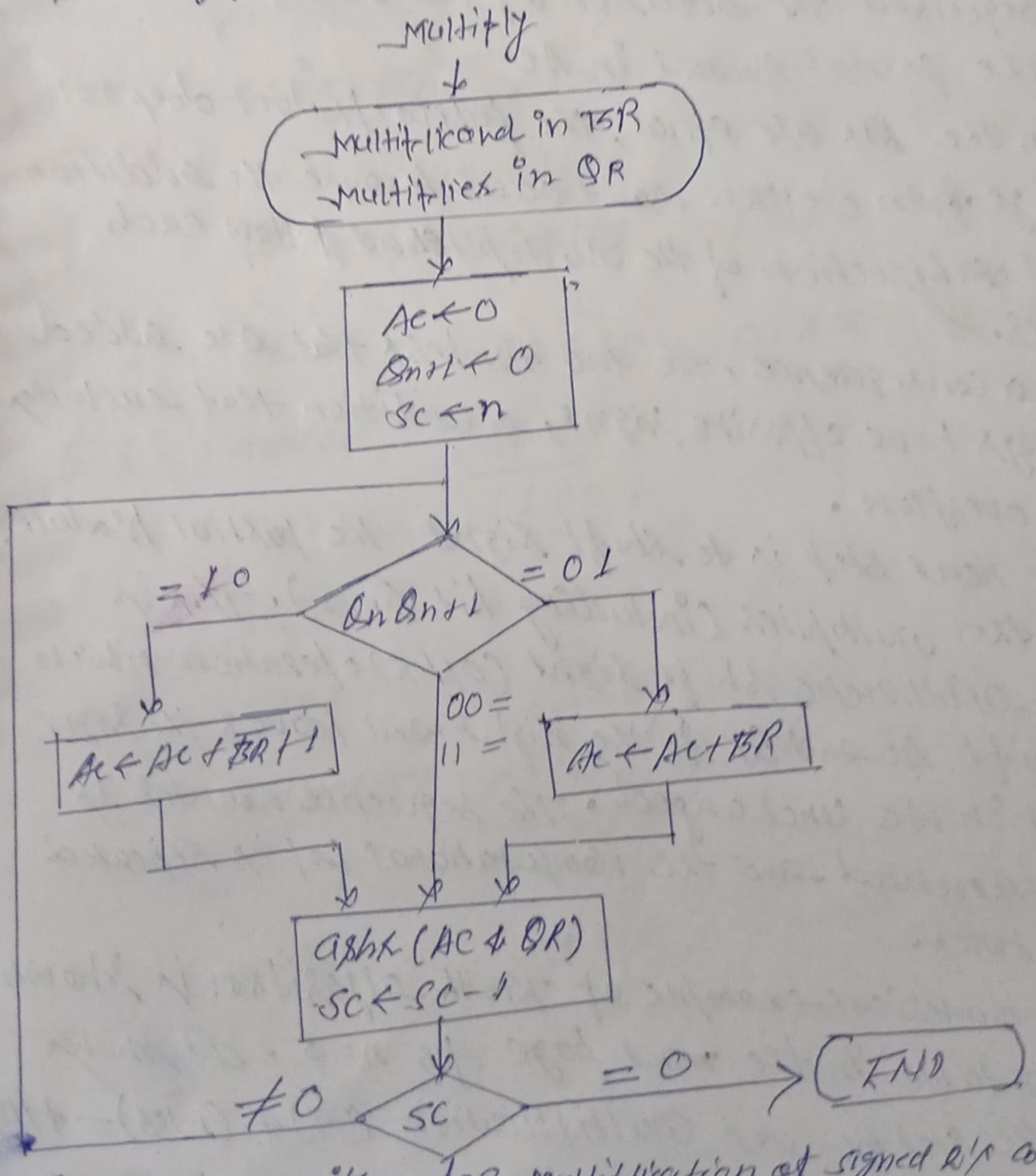


Fig: Booth algorithm for multiplication of signed 2's complement numbers.

* AC and the appended bit Qn+1 are initially released to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier.
* The two bits of the multiplier in Qn and Qn+1 are inspected.
- If the two bits are equal to 10, it means that the first 1 in a string of 1's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.
- If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC
* When the bits are equal, the partial product does not change. * An overflow can't occur because the addition and subtraction of the multiplicand follow each other.
* As a consequence, the two numbers that are added always have opposite signs, a condition that excludes an overflow.
* The next step is to shift right the partial product and the multiplier (including bit Qn+1). This is an arithmetic shift right (ashr) operation which shift AC and QR to the right and leaves the sign bit in AC unchanged. The sequence counter is decremented and the computational loop is repeated n times.
* A numerical example of Booth algorithm is shown in table in the next page for n=5. It shows the step-by-step multiplication (-9) * (-13) = +117.

* The Multiplier in QR is negative and that the Multiplicand in BR is also negative.
* The 10-bit product appears in AC and QR and is positive. The final value of Q_{n+1} is the original signal also sign bit of the Multiplier and should not be taken as part of the product

Table: Example of Multiplication with Booth Algo.

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			
		11100			
	ashr	00011	10101	1	000

Array Multipliers:→

* 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 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 IC.
* Array multipliers can be implemented with a combinational circuit, the multiplication of two 2-bit numbers is shown in below fig.

$$\begin{array}{cccc} & & b_1 & b_0 \\ & & a_1 & a_0 \\ \hline & & a_0b_1 & a_0b_0 \\ & a_1b_1 & a_1b_0 & \\ \hline C_3 & C_2 & C_1 & C_0 \end{array}$$

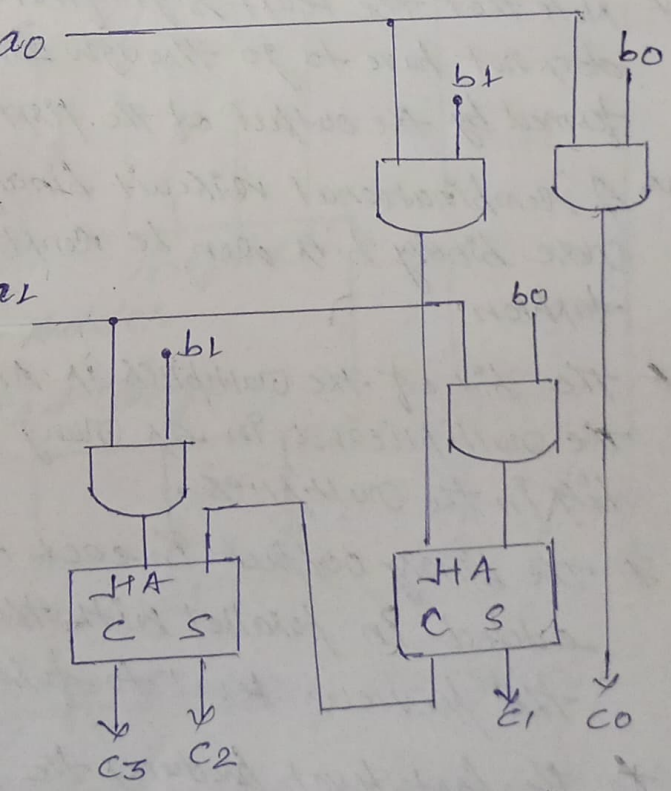


Fig: 2-bit by 2-bit Array Multiplier.

* 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 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.
* The second partial product is formed by multiplying a_1 by $b_1 b_0$ and 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 binary bits can be constructed in a similar fashion.
* The 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 gates is 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*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.

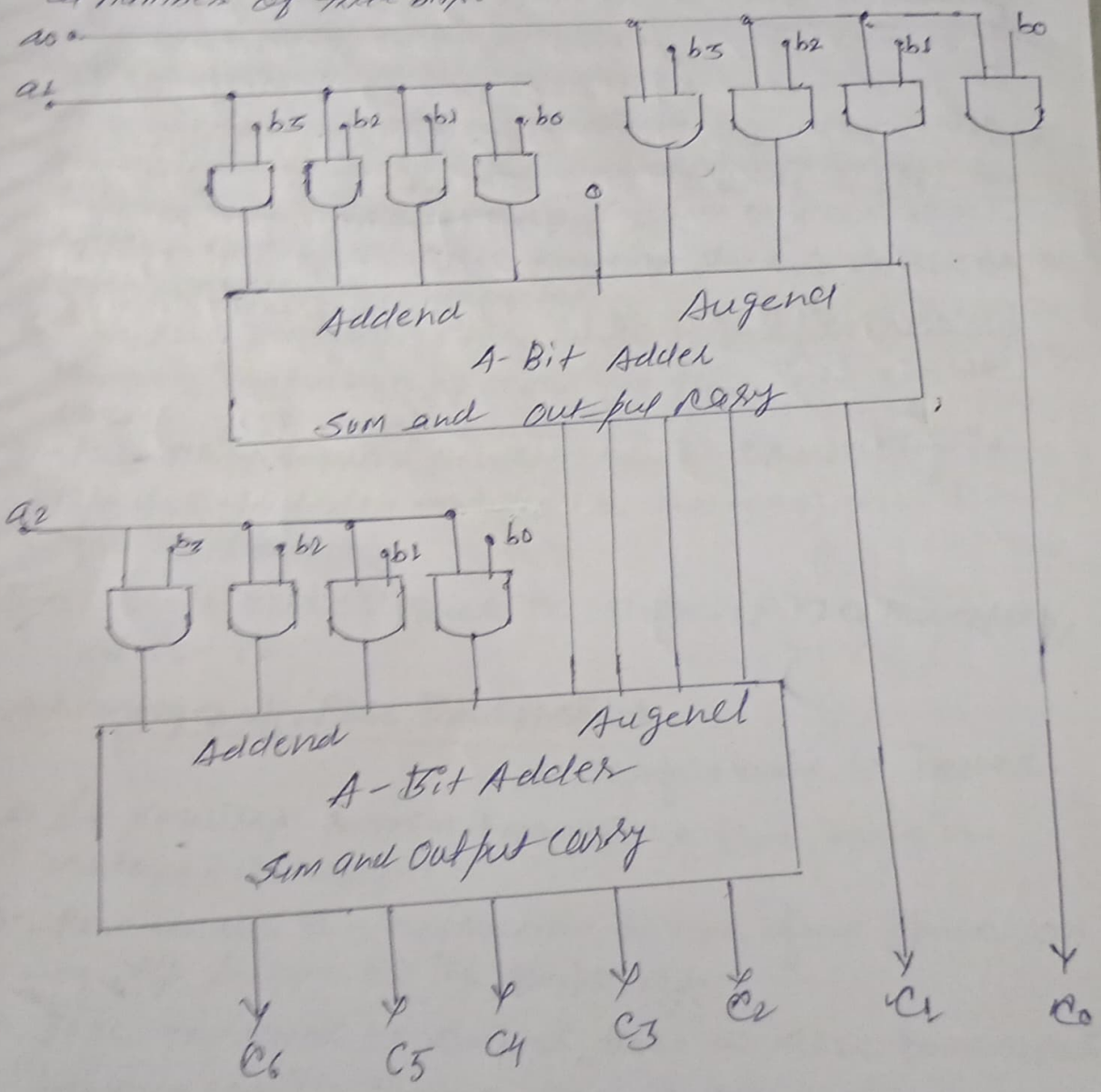


Fig: 4-Bit by 3-Bit Array Multiplier.

* 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.