Computer Arithmetic

Arithmetic Processor

- Arithmetic instruction in digital computers manipulate data to produce results necessary for the solution of the computational problems.
- An arithmetic processor is the part of a processor unit that execute arithmetic instruction
- An arithmetic instruction may specify binary or decimal data, and it may be represented in, Fixed point (integer or fraction) OR floating point form.
- The designer must be thoroughly familiar with sequence of steps in order to carry out the operation and achieve a correct result.

- Algorithm:
  - The solution to any problem that is stated by a finite number of well defined procedural step is called **algorithm.**
- Flowchart:
  - The convenient method for presenting algorithm is a flowchart.
  - The computational steps are specified in rectangular boxes
  - Decision steps indicated inside diamond-shaped boxes from which two or more alternate path emerge

Addition and Subtraction

- Data types considered for the arithmetic operations are,
  - Fixed-point binary data in signed magnitude representation
  - Fixed-point binary data in signed-2's compliment representation
  - Floating point binary data
  - Binary –coded decimal (BCD) data
- Negative fixed point binary number can be represented in three ways,
  - Signed magnitude (most computers use for floating point operations)
  - Signed 1's compliment
  - Signed 2's compliment(most computer use for integers)

Addition and Subtraction with Signed-Magnitude Data

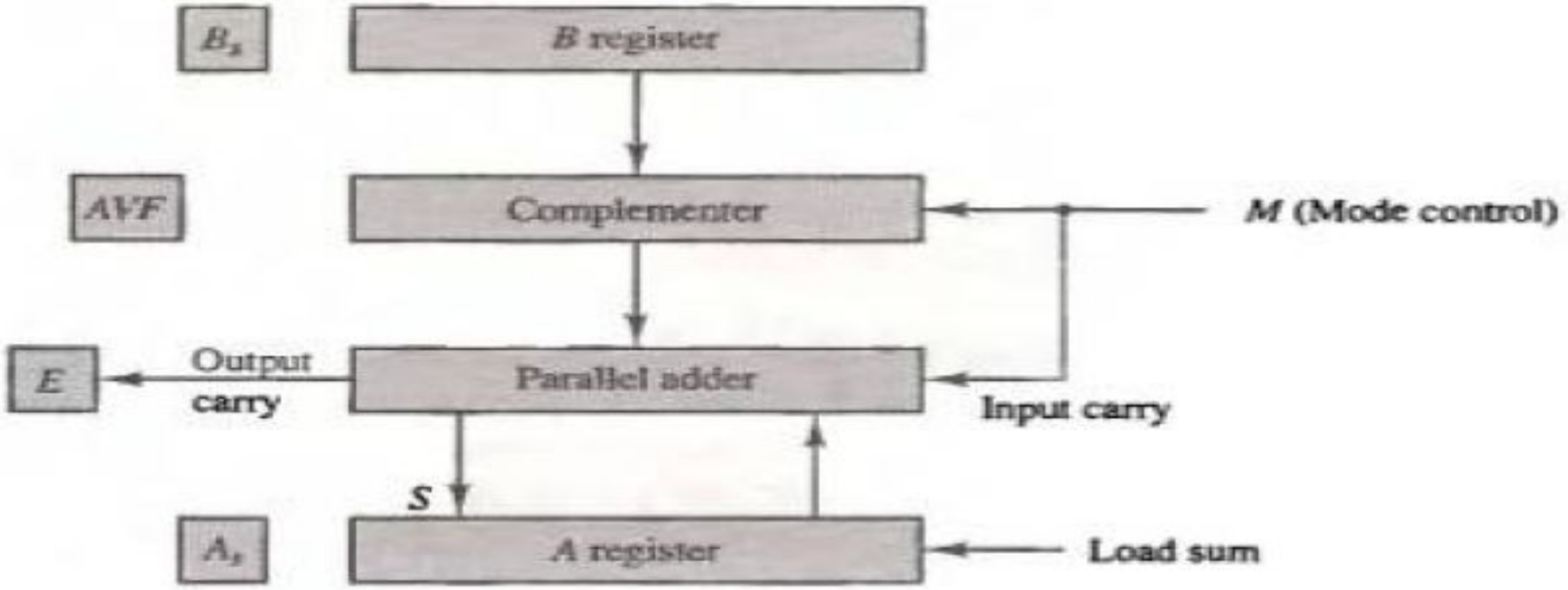
- Eight different conditions to consider for addition and subtraction

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 the two magnitudes 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 same as A if A > B or the compliment of the sign of A if A < B.
  - For equal magnitude subtract B from A and make the sign of the result

Hardware Implementation

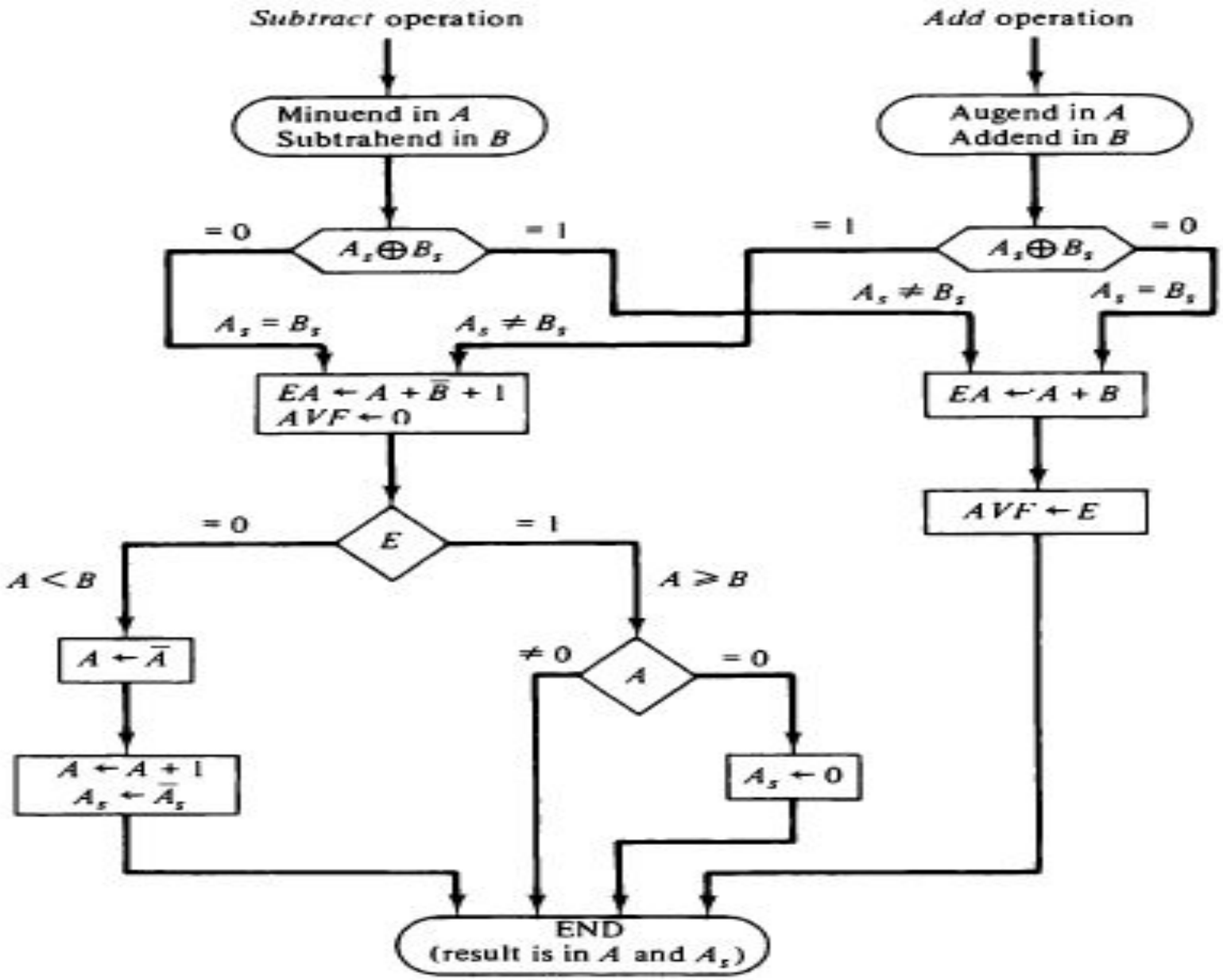
- A and B be two registers that hold the magnitudes of No
- As and Bs be two flip-flops that hold the corresponding signs
- The Result is transferred into A and As.



Cont..

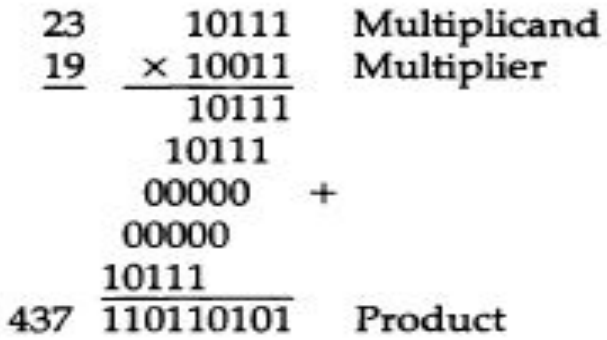
- Parallel adder is needed to perform the micro operation A + B.
- Parallel subtractor are needed to perform A – B or B – A.
  - Can be accomplished by means of compliment and add
- Comparator circuit is needed to establish if A > B, A = B or A < B
  - Comparison can be determine from the end carry after the subtraction
- The sign relationship can be determine from an exclusive-OR gate with As and Bs as inputs
- Output carry are transferred to E flip-flop
  - Where it can be checked to determine the relative magnitude of the Nos.
- Add overflow flip-flop (AVF) holds the overflow bit when A and B are added.

Hardware Algorithm



Multiplication Algorithm

- Multiplication of two fixed point binary numbers in signed magnitude representation is done by the process of successive shift and add operations.
- Numerical Example:



- The process consist nultiplier, least significant bit first.
- If the multiplier bit i nerwise zeros are copied down
- Copied number in successive line shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.

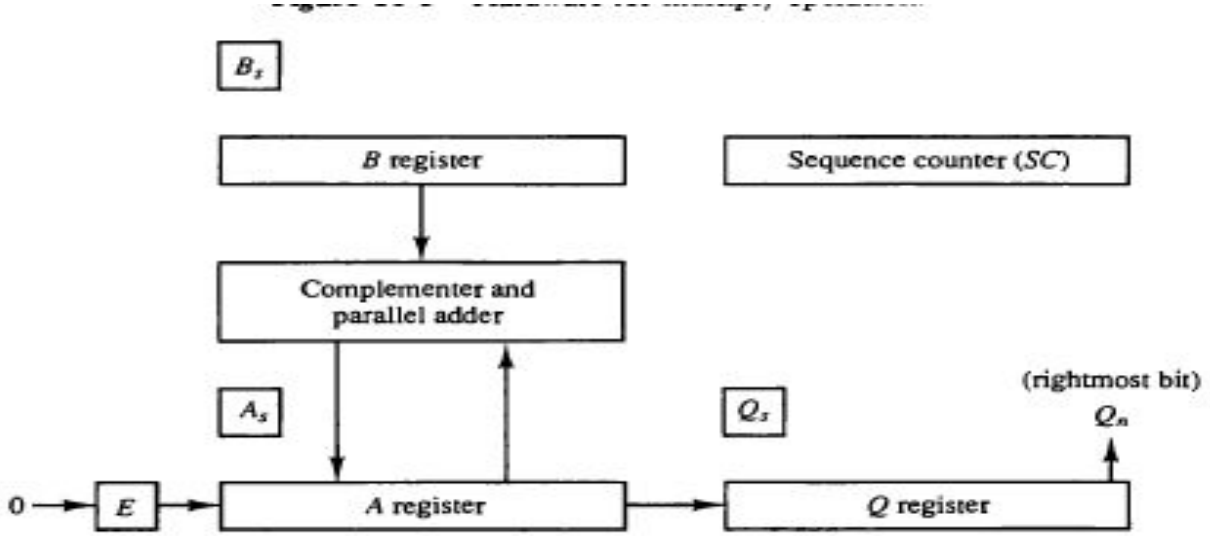
Hardware Implementation

- Multiplication process is change slightly.
- First, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in multiplier,
  - It is convenient to provide an adder for the summation of two number
  - And successively accumulate the partial products in a register
- Second, instead of shifting the multiplicand to the left, the partial product is shifted right
  - Which result in leaving the partial product and the multiplicand in the required relative position
- Third, 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

Cont..

- The hardware for multiplication consist of the equipment shown in figure.
- The multiplier stored in Q register and its sign in Qs.
- 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.

Fig: (Hardware for multiply operation)



Cont..

- Initially the multiplicand is in register B and the multiplier in Q.
- The sum of A and B forms a partial product which is transferred to the EA register.
  - Both the partial product and multiplier are shifted to the right.(shr EAQ)
  - Least significant bit of A is shifted into the most significant position of Q,
  - The bit from E is shifted into the most significant position of A and 0 to E.
- After the shift, one bit of partial product is shifted into Q, pushing the multiplier bit one position to the right.
- In this manner, the rightmost flip-flop in register Q, designated by Qn, will hold the bit of the multiplier, which must be inspected next.

Hardware Algorithm

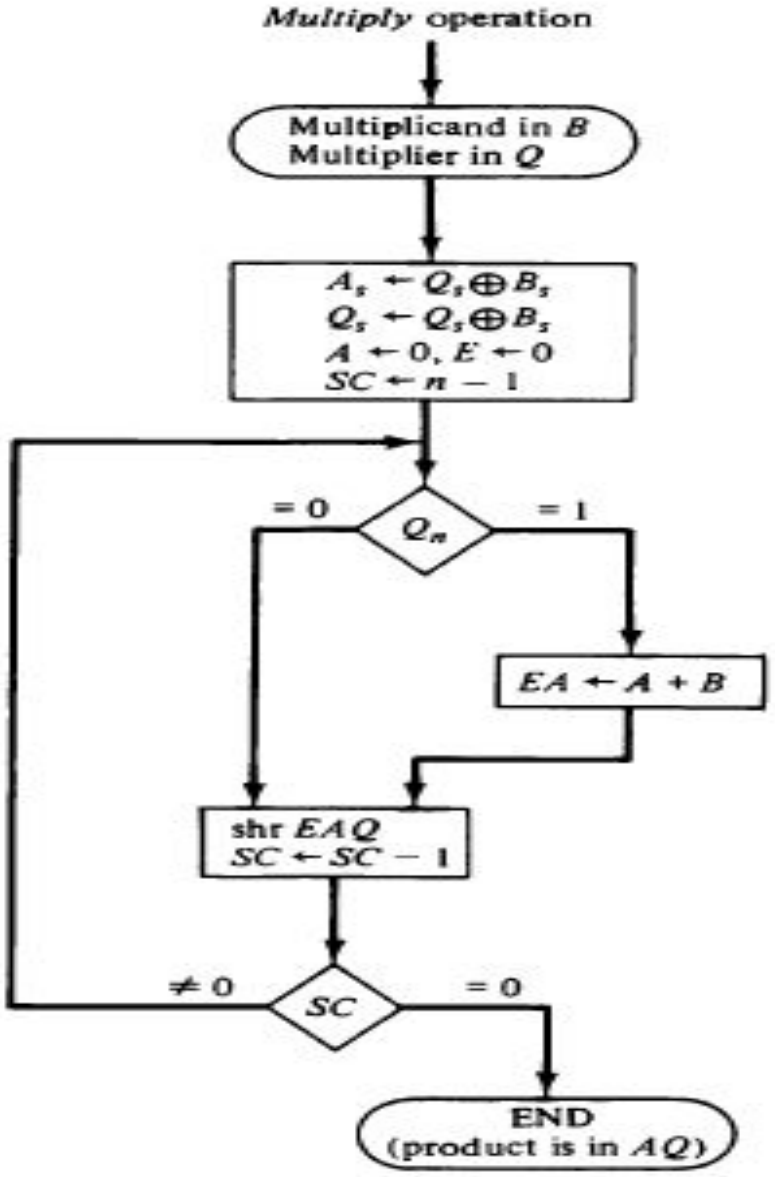
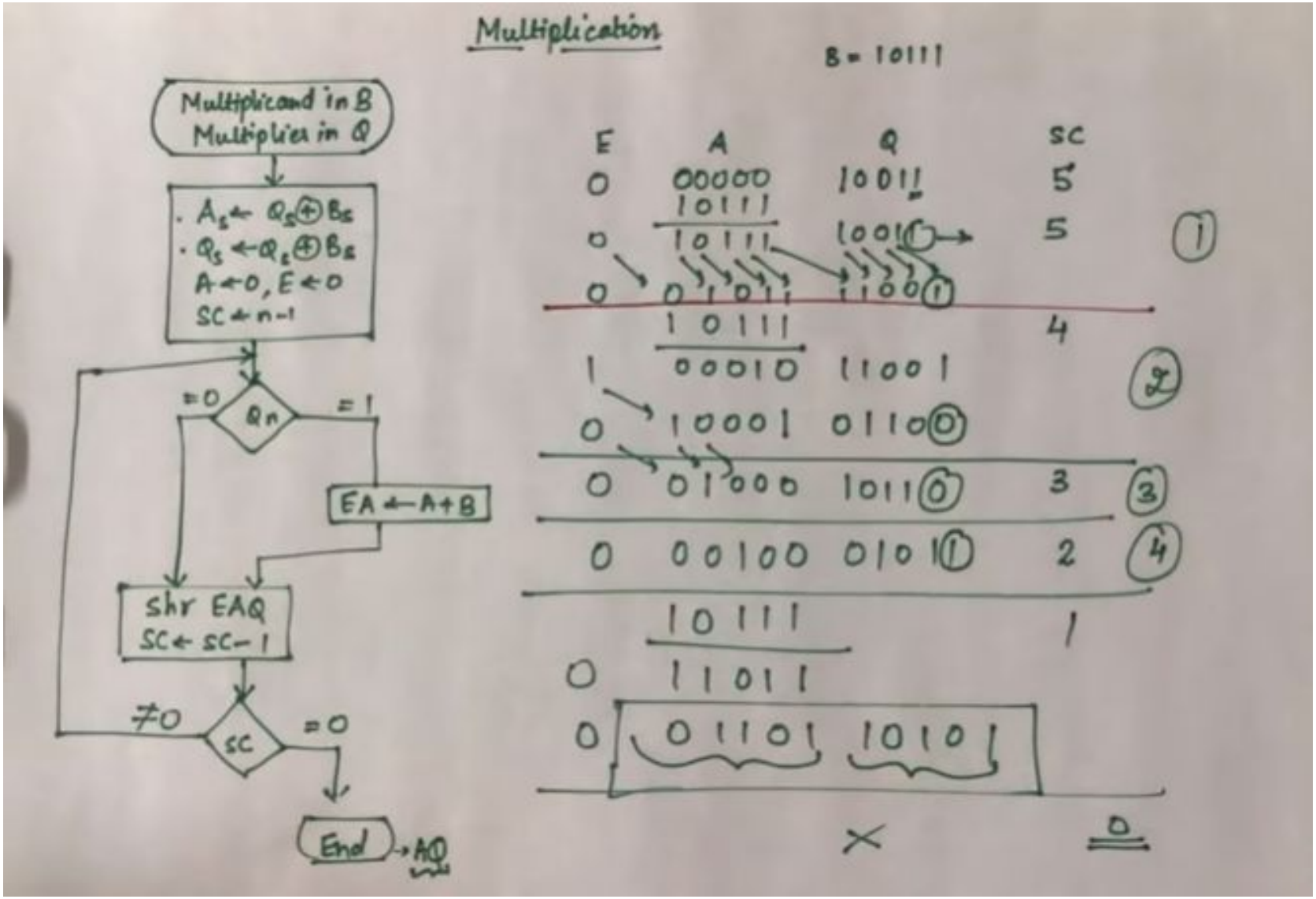


Fig: Numerical Example for binary multiplier

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 $AQ = 0110110101$				

Example of multiplication algorithm

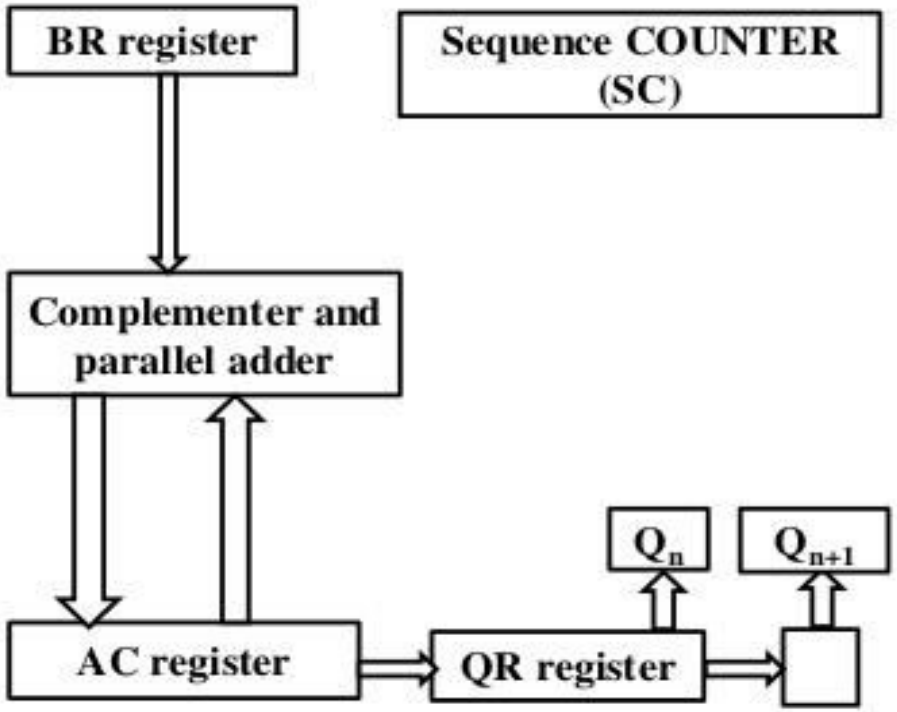


Booth's algorithm

- 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 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.
- As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product.

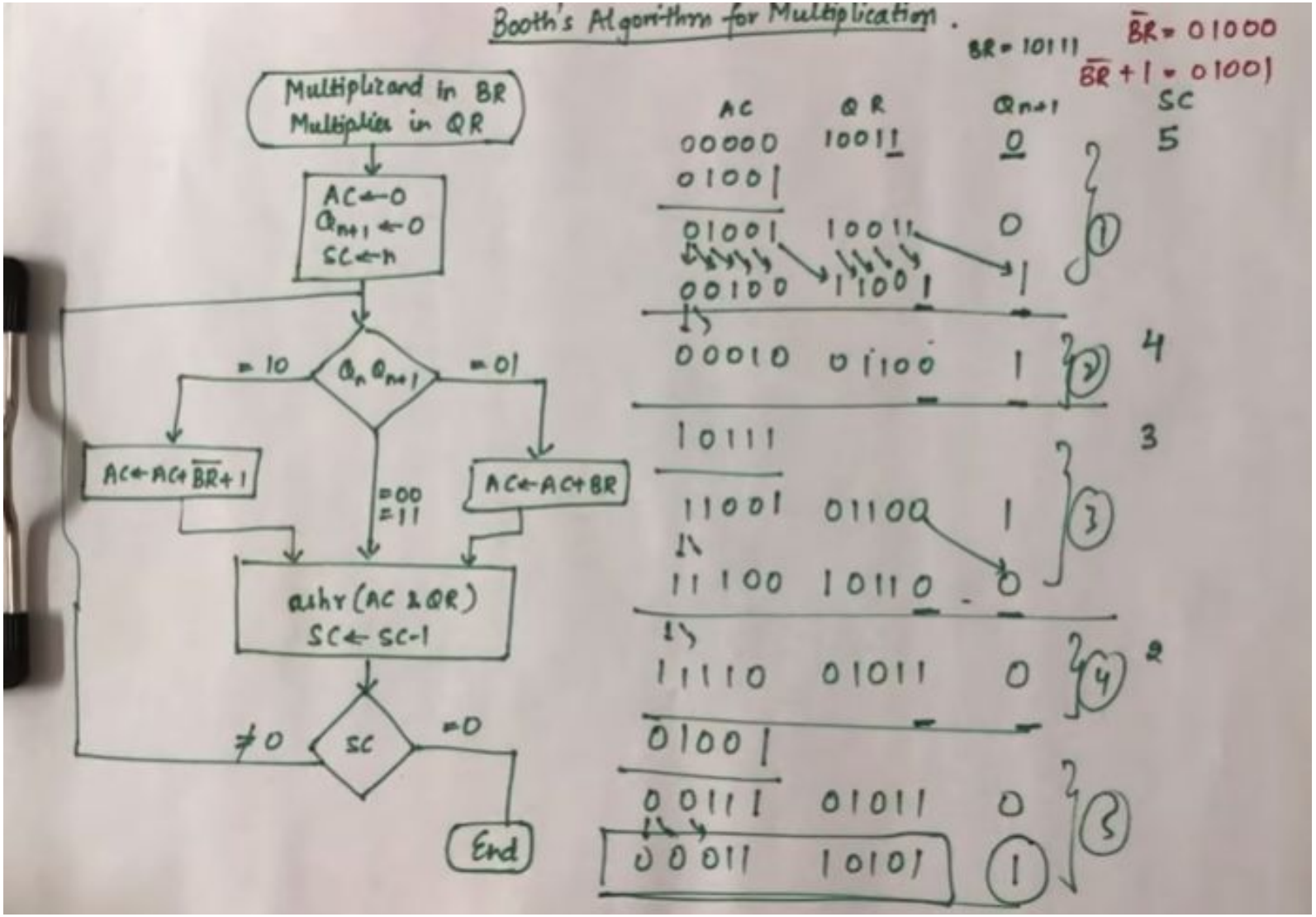
Hardware for Booth Algorithm

- Sign bits are not separated from the rest of the registers
- rename registers A,B, and Q as AC,BR and QR respectively
- Q_n designates the least significant bit of the multiplier in register QR
- Flip-flop Qn+1 is appended to QR to facilitate a double bit inspection of the multiplier



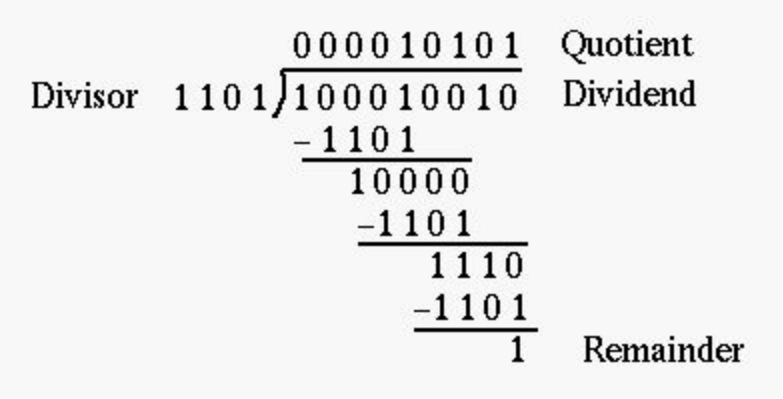
- The algorithm works for positive or negative multipliers in 2’s complement
- representation.
- This is because a negative multiplier ends with a string of 1’s and the last operation will be a subtraction of the appropriate weight.
- The two bits of the multiplier in Qn and Qn+1 are inspected.
- If 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 two bits are equal, the partial product does not change.

Example of Booth multiplication algorithm



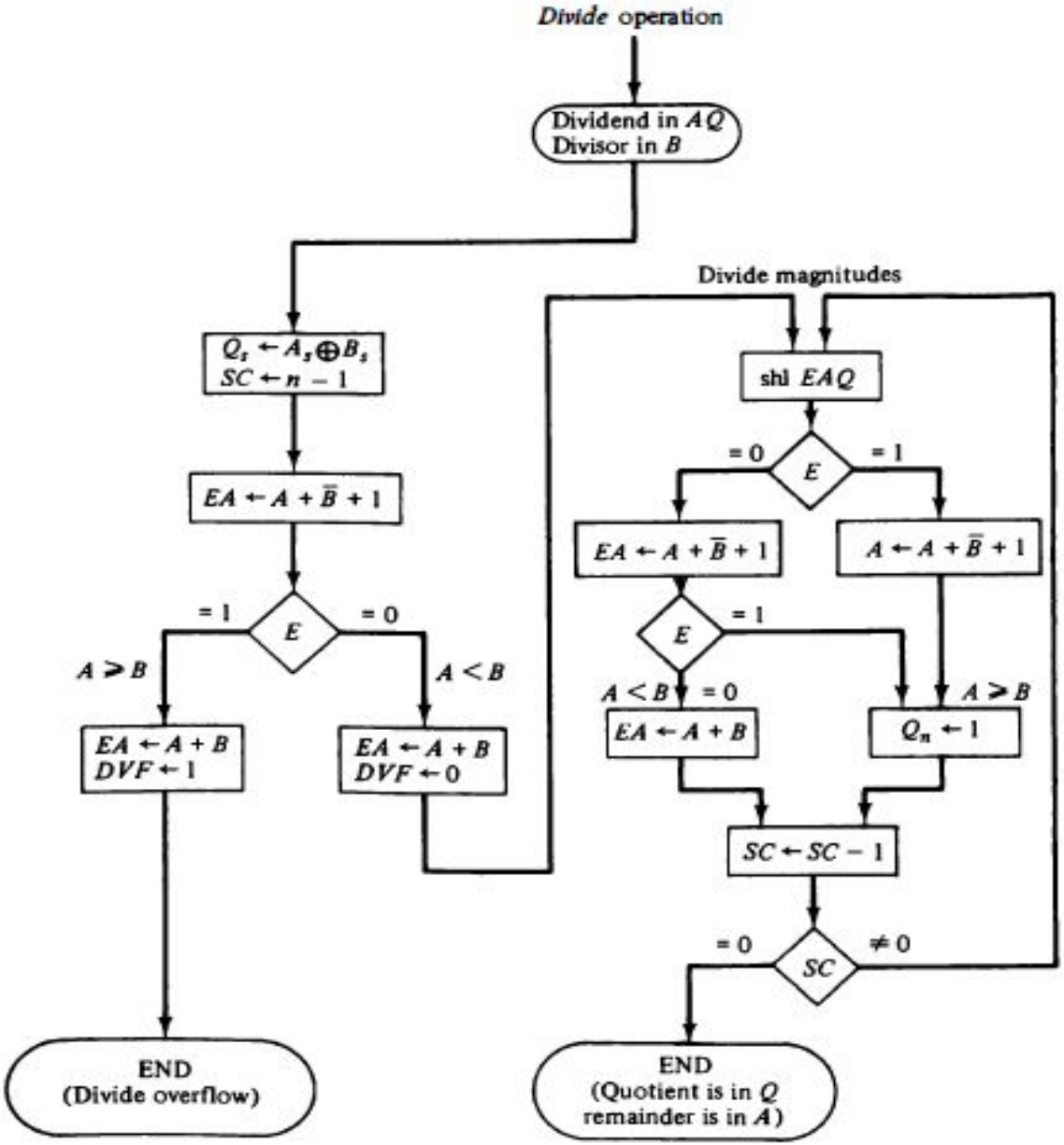
Division Algorithm

- 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. The division process is described in Figure



- The devisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend.
- Now we shift the divisor once to the right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. Comparing a partial remainder with the divisor continues the process. 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. Obviously the result gives both a quotient and a remainder.

- **Hardware Implementation for Signed-Magnitude Data :**
- In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, two dividends, or partial remainders, are shifted to the left, thus leaving the two numbers in the required relative position.
- Subtraction is achieved by adding A to the 2's complement of B. End carry gives the information about the relative magnitudes.
- The hardware required is identical to that of multiplication. Register EAQ is now shifted to the left with 0 inserted into Qn and the previous value of E is lost.


Divide operation
Dividend in AQ
Divisor in B
$Q_s \leftarrow A_s \oplus B_s$
$SC \leftarrow n - 1$
$EA \leftarrow A + \overline{B} + 1$
E
= 1
= 0
$A \geq B$
$A < B$
$EA \leftarrow A + B$
$DVF \leftarrow 1$
$EA \leftarrow A + B$
$DVF \leftarrow 0$
END
(Divide overflow)
Divide magnitudes
shl EAQ
E
= 0
= 1
$EA \leftarrow A + \overline{B} + 1$
$A \leftarrow A + \overline{B} + 1$
E
= 1
$A < B$
= 0
$A \geq B$
$EA \leftarrow A + B$
$Q_n \leftarrow 1$
$SC \leftarrow SC - 1$
SC
= 0
$\neq 0$
END
(Quotient is in Q
remainder is in A)

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

	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		2
Restore remainder	1	01010		
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	

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.

Floating-Point Arithmetic Operation

- Floating point numbers in computer register consist of Mantissa m and an exponent e.

$$m \times r^e$$

- For example,
- Decimal number (537.25) is represented in register with m = 57325 and e = 3.

$$.57325 \times 10^3$$

Cont..

- Floating point representation increases the range of numbers that can be accommodated in given register.
- i.e, A computer with 48-bit words. the range of Fixed point integer is **+/-(2^{47} - 1)** which is approximately **+/-$(10)^{14}$**
- The 48-bit can be used to represent floating point number as
  - **36 bits for the mantissa and 12 bits for the exponent.**
- Computer with shorter word length use two or more words to store floating point number
  - 8-bit microcomputer may use four words to represent 1 floating point no.
  - One word of 8-bit reserved for exponent and 24 bits of three words are used for the mantissa.

Cont..

- Arithmetic operations with floating point no are more complicated and require more complex hardware.
- Addition and subtraction requires
  - Alignment of radix point since the exponents must made equal before add – subtract mantissa
    - E.g $.5372400 \times 10^{2}$ (shift three position to left)

      $+ .1580000 \times 10^{-1}$ (shift three position to right)
  - Second method is preferable because it only reduce accuracy.

$$.5372400 \times 10^{2}$$
$$+ .0001580 \times 10^{2}$$
$$.5373980 \times 10^{2}$$

- When two normalized mantissa are added, sum may contain overflow digit.
  - It can corrected easily by shifting the sum once to the right and incrementing exponent.

Cont..

- Underflow
  - Floating point number that has 0 in the most significant position of the mantissa is said to have an underflow.
  - To normalize, shift the mantissa left and decrement the exponent until nonzero digit appears.

    E.g. $.56780 \times 10^5$

    $.56430 \times 10^5$

    $.00350 \times 10^5$ ⇒ $(.35000 \times 10^3)$
- The Exponent may be represented in any one of the three representations.
  - Signed-magnitude, signed-2's complement, or signed-1's complement.
  - Forth representation employed as a **Biased Exponents.**

Biased Exponents

- Biased is +ve number that is added to each exponent. The sign bit is removed from being a separate entity.
- Consider , exponent that ranges from -50 to 49. internally it is represented by adding to it bias of 50.

$$e + 50$$

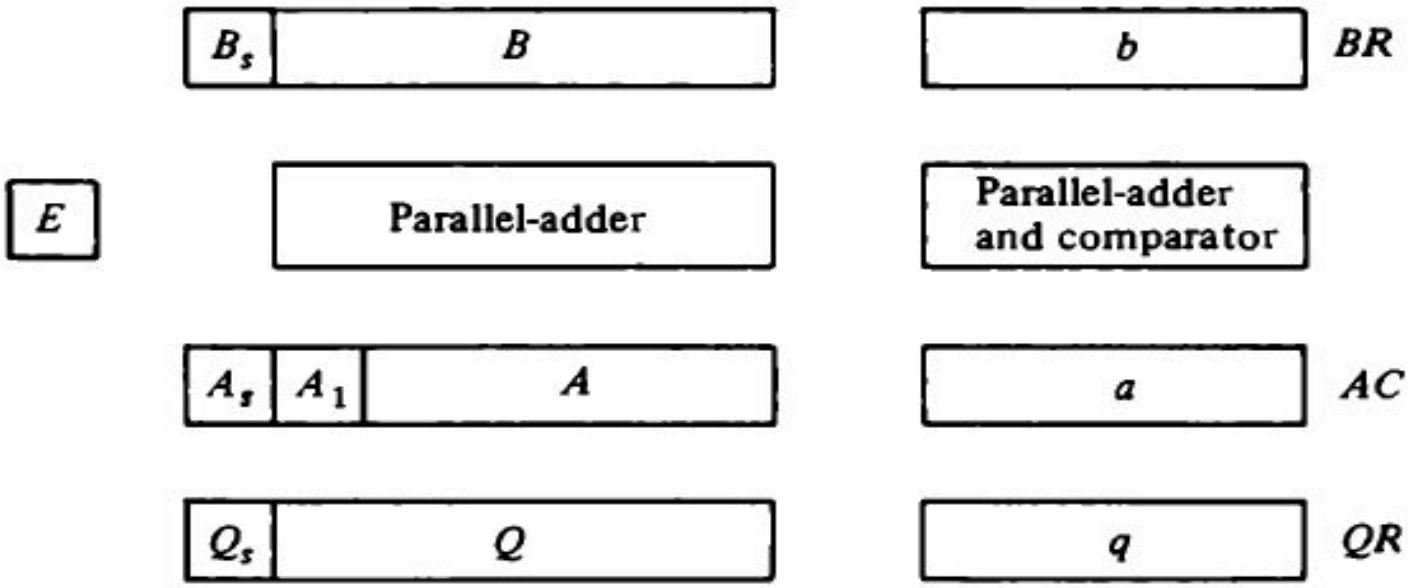
(where, e = actual exponent)

This way the exponent are represented in the range of 00 to 99.

  - Positive number ranges from 99 to 50
  - Negative numbers ranges from 49 to 0.

Register Configuration

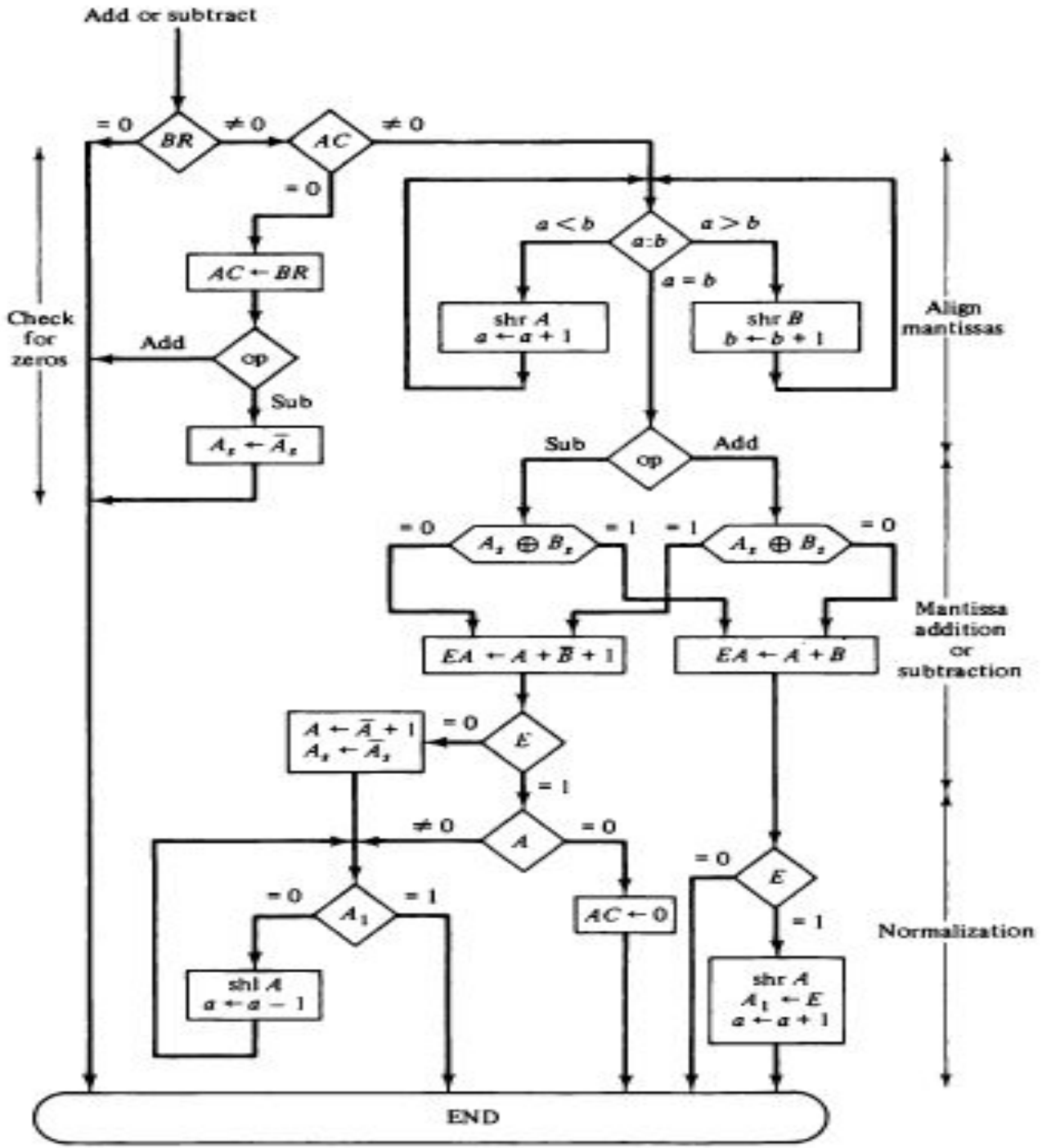
- Same registers as fixed point are used for processing mantissa.
- The exponents are handle in different lies.
- Mantissa in signed magnitude representation and bias exponent
  - Parallel adder adds two mantissa and transfer sum into A and carry in E.
  - Separate parallel adder and magnitude comparator is used for exponent.



Addition And Subtraction

- Two floating point operands are in AC and BR. Sum or difference formed in AC
- Algorithm is divided into four consecutive parts.

1. Check for Zeros
   - Floating point no that is zero cannot be normalized.(we check for zeros in beginning and terminate process if necessary)
2. Align the mantissa
   - The alignment of mantissa must be carried out prior to their operation
3. Add or Subtract the mantissas
4. Normalize the result
   - After add or subtract the result may be unnormalized.
   - Normalization process insure that the result is normalized prior to its transfer to memory.


Add or subtract
= 0
BR
≠ 0
AC
≠ 0
= 0
AC ← BR
Check for zeros
Add
op
Sub
A_s ← Ā_s
a < b
a : b
a > b
a = b
shr A
a ← a + 1
shr B
b ← b + 1
Align mantissas
Sub
op
Add
= 0
A_s ⊕ B_s
= 1
= 1
A_s ⊕ B_s
= 0
EA ← A + B̄ + 1
EA ← A + B
Mantissa addition or subtraction
A ← Ā + 1
A_s ← Ā_s
= 0
E
= 1
≠ 0
A
= 0
= 0
A_1
= 1
AC ← 0
shl A
a ← a − 1
= 0
E
= 1
shr A
A_1 ← E
a ← a + 1
Normalization
END

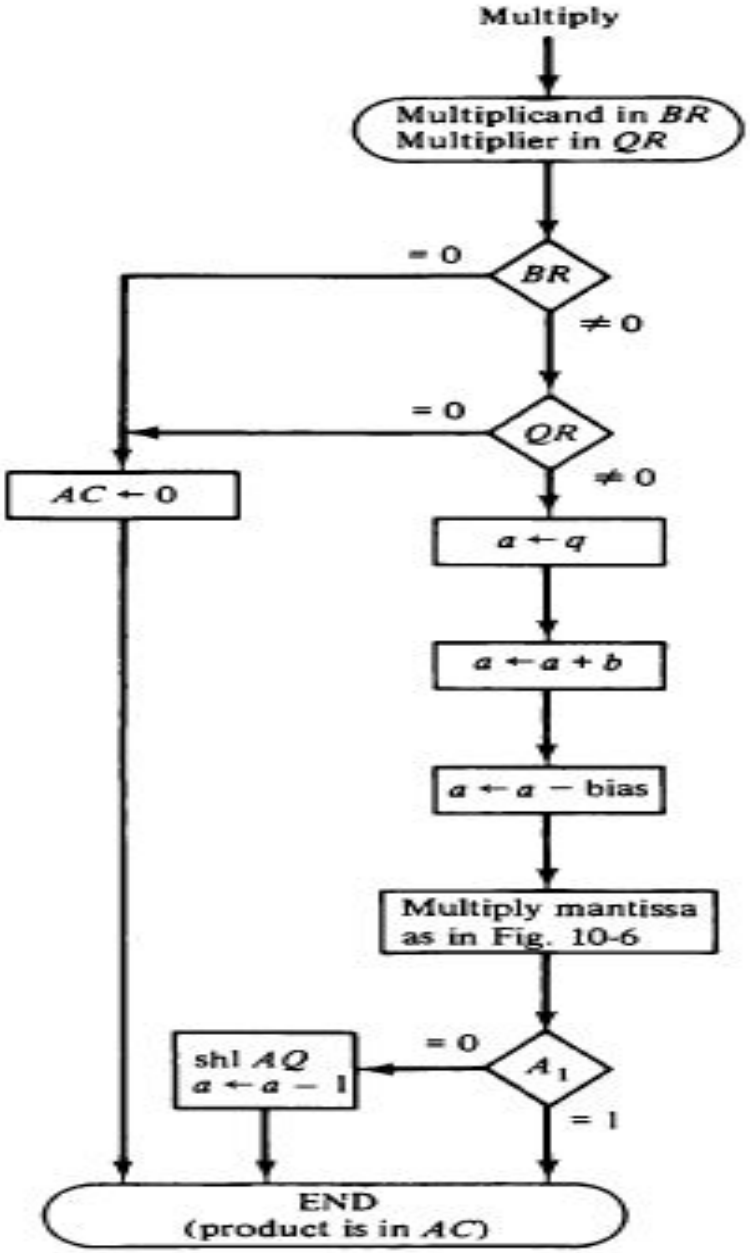
Multiplication

- Require multiplication of mantissa and addition of exponents

  - Multiplication algorithm can be sub-divided into four parts

1. Check for zeros
2. Add the exponents
3. Multiply the mantissas
4. Normalize the product

Algorithm


Multiply
Multiplicand in BR
Multiplier in QR
BR
= 0
≠ 0
QR
= 0
≠ 0
AC ← 0
a ← q
a ← a + b
a ← a − bias
Multiply mantissa
as in Fig. 10-6
A1
= 0
= 1
shl AQ
a ← a − 1
END
(product is in AC)