Unit 2
Arithmetic Processor Design

Designed By : Deepak Bhalla

Number System

- We have already mentioned that computer can handle with two type of signals, therefore, to represent any information in computer, we have to take help of these two signals.
- These two signals correspond to two levels of electrical signals, and symbolically we represent them as 0 and 1.
- In our day to day activities for arithmetic, we use the *Decimal Number System*. The decimal number system is said to be of base, or radix 10, because it uses ten digits and the coefficients are multiplied by power of 10

- A decimal number such as 5273 represents a quantity equal to 5 thousands plus 2 hundred, plus 7 tens, plus 3 units. The thousands, hundreds, etc. are powers of 10 implied by the position of the coefficients. To be more precise, 5273 should be written as

$$5\times10^{3} + 2\times10^{2} + 7\times10^{1} + 3\times10^{0}$$

- In decima eed 10 different symbols. But in computer we have provision to represent only two symbols. So directly we cannot use decimal number system in computer arithmetic.
- For computer arithmetic we use **binary number system**.

- The binary number system uses two symbols to represent the number and these two symbols are 0 and 1.
- The binary number system is said to be of base 2 or radix 2, because it uses two digits and the coefficients are multiplied by power of 2.
- The binary number 110011 represents the quantity equal to:

(in decimal)

$$1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 51$$

- To distinguish between radix numbers, the digits will be enclosed in parenthesis and the radix of the number inserted as a subscript.
- For example, to show the equality between decimal and binary forty-five we will write

$$(101101)_2 = (45)_{10}$$

- Besides decimal and binary number systems, the octal number system are there
- **Octal Number :** The octal number system is said to be of base, or radix 8, because it uses 8 digits and the coefficients are multiplied by power of 8.
  Eight digits used in octal system are: 0, 1, 2, 3, 4, 5, 6 and 7.
- **Hexadecimal number :** The hexadecimal number system is said to be of base, or radix 16, because it uses 16 symbols and the coefficients are multiplied by power of 16. Sixteen digits used in hexadecimal system are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.

For example, octal 736.4 is converted to decimal as follows:

$(736.4)_8 = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1}$

$= 7 \times 64 + 3 \times 8 + 6 \times 1 + 4 / 8 = (478.5)_{10}$

For example, Hex. Decimal F3 is converted to decimal as follows:

$(F3)_{16} = F \times 16 + 3 = 15 \times 16 + 3$

$= (243)_{10}$

Octal and Hexadecimal Numbers

- The conversion from and to binary, octal, and hexadecimal representation plays an important part in digital computers.
- Since 2^3 = 8 and 2^4 = 16, each octal digit represents to 3 binary digits and each hexadecimal digit corresponds to 4 binary digits.
- The conversion of binary to octal is easily accomplished by partitioning the binary number into groups of three bits each.

Binary	Octal	Hexadecimal	Decimal
01101000	150	68	104
00111010	072	3A	58
---------------	------	------	-----
10100010	242	A2	162

Binary representation of 41.6875 is 101001.1011

Therefore any real number can be converted to binary number system

There are **two schemes** to represent real number :

- Fixed-point representation
- Floating-point representation

Fixed Point Representation

- Binary representation of 41.6875 is 101001.1011
- To store this number, we have to store **two information**,

  -- the part before decimal point and

  -- the part after decimal point.
- This is known as fixed-point representation where the position of decimal point is fixed and number of bits before and after decimal point are also predefined.

- If we use 16 bits before decimal point and 7 bits after decimal point, in signed magnitude form
- One bit is required for sign information, so the total size of the number is 24 bits

  (1(sign) + 16(before decimal point) + 7(after decimal point)).

Integer Representation

Representation of Unsigned Integers

- Any integer can be stored in computer in binary form.
- As for example:
  The binary equivalent of integer 107 is 1101011, so 1101011 are stored to represent 107.
- What is the size of Integer that can be stored in a Computer?

- It depends on the word size of the Computer. If we are working with 8-bit computer, then we can use only 8 bits to represent the number. The eight bit computer means the storage organization for data is 8 bits.
- In case of 8-bit numbers, the minimum number that can be stored in computer is 00000000 (0) and maximum number is 11111111 (255) (if we are working with natural numbers).
- In general, for *n-bit* number, the range for natural number is from

$$0 \text{ to } 2^n - 1$$

Designed By : Deepak Bhalla MYcsvtu Notes

- Any arithmetic operation can be performed with the help of binary number system. Consider the following two examples, where decimal and binary additions are shown side by side.

01101000	104
00110001	49
----------------	------
10011001	153

- ➢ In the above example, the result is an 8-bit number, as it can be stored in the 8-bit computer, so we get the correct results.

10000001	**129**
10101010	**178**
-----------------	**------**
100101011	**307**

- In the above example, the result is a 9-bit number, but we can store only 8 bits, and the most significant bit (MSB) cannot be stored.
- The result of this addition will be stored as (00101011) which is 43 and it is not the desired result. Since we cannot store the complete result of an operation, and it is known as the overflow case.

Signed Integer

- When a integer binary number is positive , the sign is represented by 0 and a magnitude by a positive binary numbers.
- When a number is negative , the sign is represented by 1 but the rest of the numbers may be represented by one of the 3 possible ways:
  - **Signed-Magnitude form.**
  - **Signed 1's complement form.**
  - **Signed 2's complement form.**

Signed magnitude form:

- In signed-magnitude form, one particular bit is used to indicate the sign of the number, whether it is a positive number or a negative number.
- Other bits are used to represent the magnitude of the number.
- Generally, Most Significant Bit (MSB) is used to indicate the sign and it is termed as signed bit. 0 in signed bit indicates positive number and 1 in signed bit indicates negative number.
- For example, consider the signed num 14 stored in an 8 bit register
- +14 is represented by a sign bit of 0 in the left most position followed by the binary equivalent

of 14 : 00001110.

- Note that each of the 8 bits of the register must have a value and therefore 0's must be inserted in the most significant positions following the sign bit.
- Although there is one way to represent +14
- But there is 3 ways to represent -14 with eight bits.
  - In signed magnitude representation 1 0001110
  - The signed magnitude representation of -14 is obtained from +14 by complementing only the sign bits.

The Concept of Complement

- The concept of complements is used to represent signed number.
- Consider a number system of *base-r* or *radix-r*. There are two types of complements,
- The radix complement or the *r's* complement.
- The diminished radix complement or the *(r 1)'s* complement.

Diminished Radix Complement :

Given a number *N* in base *r* having *n* digits , the *(r - 1)'s* complement of *N* is defined a$(r^n - 1) - N$

For decimal numbers , *r* = 10 and *r* - 1 =9

, so the 9's complement of *N* is . $(10^n - 1) - N$

e.g., 9's complement of 5642 is 9999 - 5642 = 4357.

Radix Complement :

- The *r's* complement of an *n-digit* number in base *r* is defined as   for *N* != 0   and   0 for *N* = 0.
- *r's* complement is obtained by adding 1 to the (r - 1)'s complement, since

$$(r^n - N) = \left[(r^n - 1) - N\right] + 1$$

- e.g., 10's complement of 5642 is 9's complement of 5642 + 1,   i.e., 4357 + 1 = 4358
- e.g.,   2's complement of 1010 is 1's complement of 1010 + 1,   i.e., 0101 + 1 = 0110.

Representation of Signed integer in 1's complement form:

- Consider the eight bit number 01011100, 1's complements of this number is 10100011. If we perform the following addition:
- If we perform the following addition:

```
0 1 0 1 1 1 0 0
1 0 1 0 0 0 1 1
---------------------------
1 1 1 1 1 1 1 1
```

- If we add 1 to the number, the result is 100000000

- Since we are considering an eight bit number, so the 9th bit (MSB) of the result can not be stored. Therefore, the final result is 00000000.
- Since the addition of two number is 0, so one can be treated as the negative of the other number. So, 1's complement can be used to represent negative number.
- The signed -1’s complement representation of -14 is obtained by complementing all the bits of +14 including sign bit.

Representation of Signed integer in 2's complement form:

- Consider the eight bit number 01011100
- 2's complements of this number is 10100100
- The signed-2’s of -14 is obtained by taking the 2’s complement of +14 including its sign bit

Representation of -14 is

- In signed magnitude representation

  1 0001110

- In signed-1’s complement representation

  1 1110001

- In signed-2’s complement representation

  1 1110010

Designed By : Deepak Bhalla MYcsvtu Notes

Introduction to Arithmetic Processor

- Arithmetic instructions in digital computers manipulate data to produce results necessary for the solution of computational problems.
- The 4 basic arithmetic operations are addition, subtraction, multiplication and division.
- An arithmetic processor is the part of a processor unit that executes arithmetic operations.
- An arithmetic instruction may specify binary or decimal data, and in each case the data may be in fixed-point or floating point form.
- Fixed points numbers may represent integers or fractions

- The arithmetic processor is very simple if only a binary fixed point *add* instruction is included.
- It would be more complicated if it includes all four arithmetic operations for binary and decimal data in fixed point and floating point representation.
- Earlier we are taught to perform the basic arithmetic operations in signed magnitude representation.
- We see various arithmetic algorithm and also see the procedure for implementing them with digital hardware

We consider addition, subtractio multiplication and division for the followin types of data

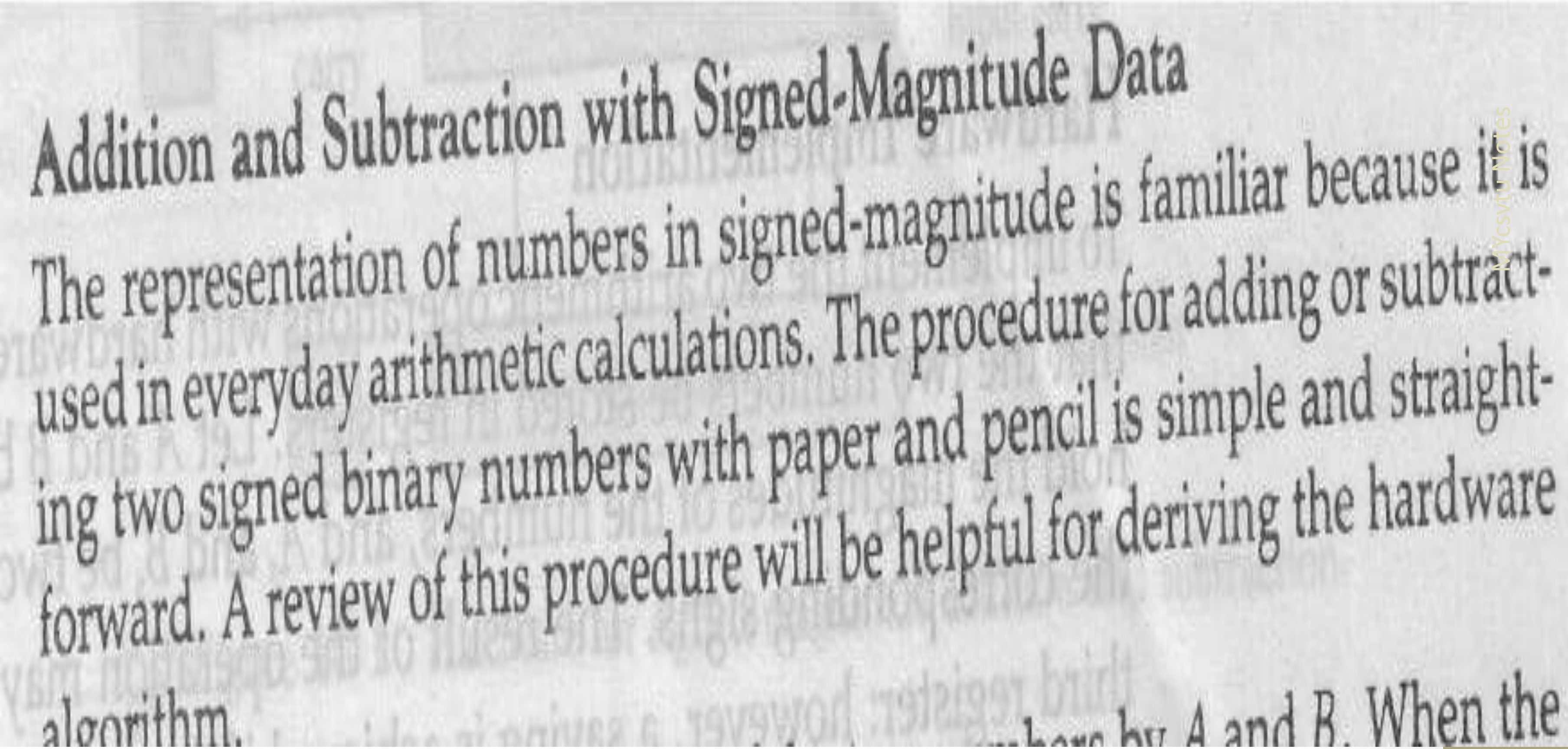
- Fixed point binary data in signed- magnitude representation.
- Fixed point binary data in signed-2’s complement representation
- Floating point binary data
- Binary coded decimal data

Fixed Point Arithmetic Addition & Subtraction

- As stated before there are 3 ways of representing negative fixed point binary numbers.
  - **Signed-Magnitude form.**
  - **Signed 1’s complement form.**
  - **Signed 2’s complement form.**
- Most computers use the signed 2’s complement representation when performing arithmetic operations with integer.

- For floating point operations, most computers use the sign magnitude representation.

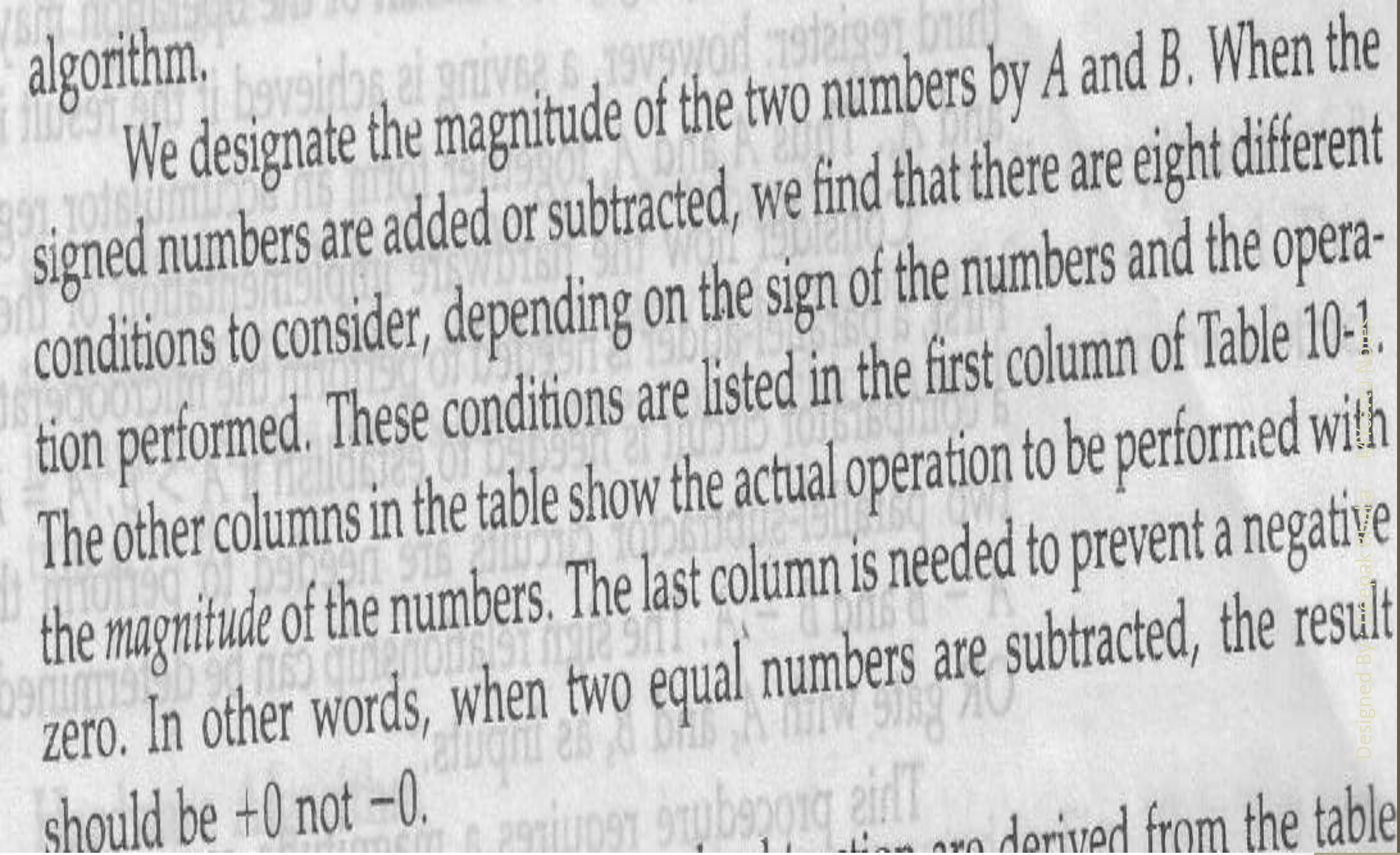
Addition & Subtraction with signed Magnitude Data



Addition and Subtraction with Signed-Magnitude Data

The representation of numbers in signed-magnitude is familiar because it is used in everyday arithmetic calculations. The procedure for adding or subtracting two signed binary numbers with paper and pencil is simple and straightforward. A review of this procedure will be helpful for deriving the hardware algorithm.

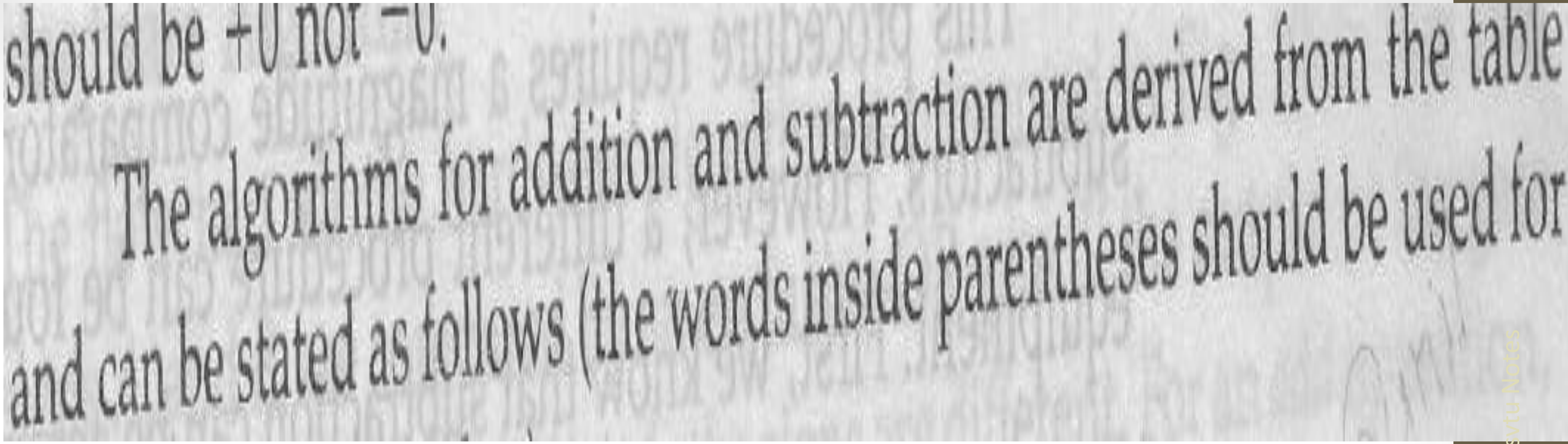
by A and B. When the



algorithm.

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 conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of Table 10-1. The other columns in the table show the actual operation to be performed with the *magnitude* of the numbers. The last column is needed to prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not −0.

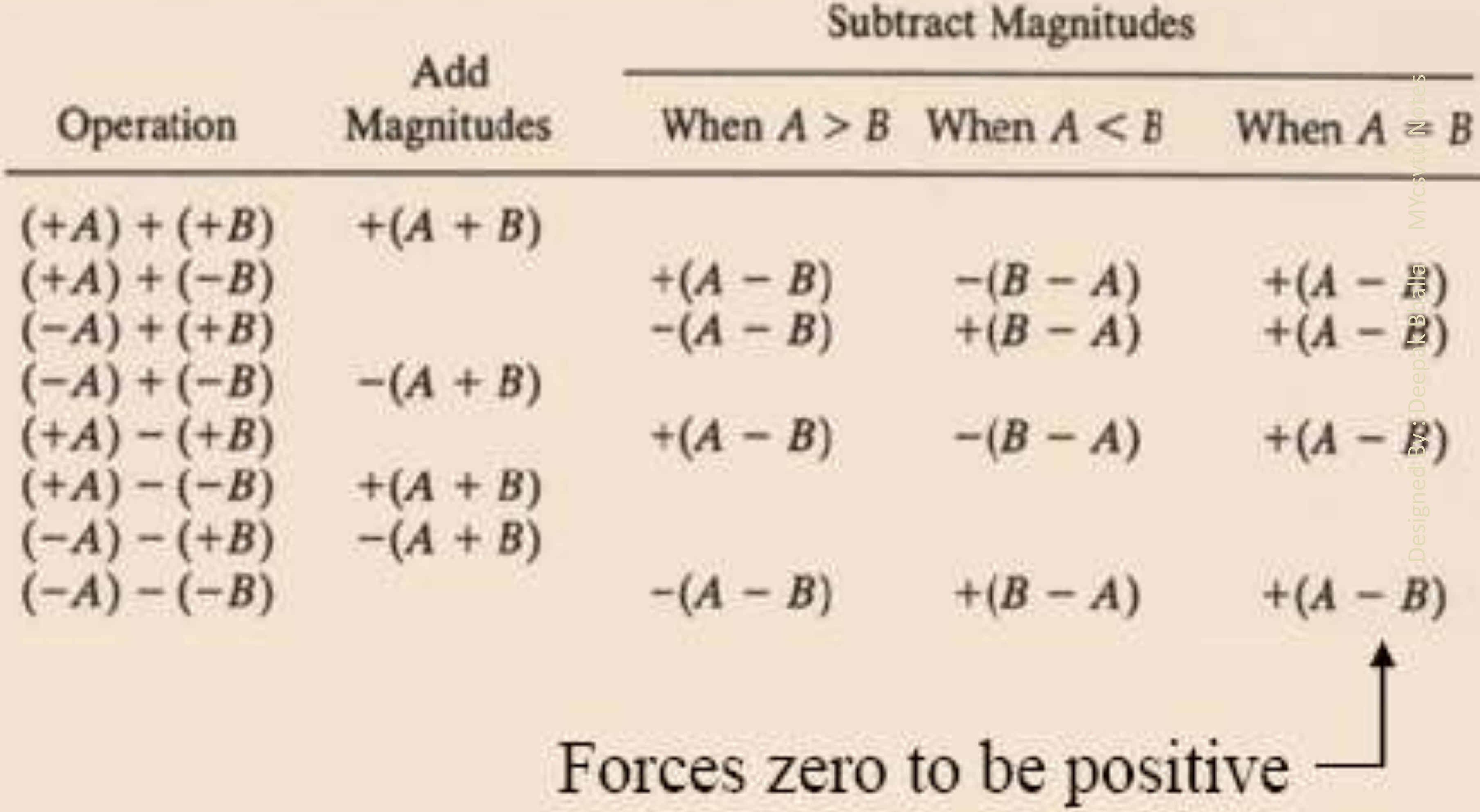
are derived from the table



should be +0 nor −0.

The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words inside parentheses should be used for

Add / Subtract Signed-Magnitude



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

Forces zero to be positive

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 signs 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 sign of A if A<B
- If the two magnitudes are equal, subtract B from A and make the sign of the result positive

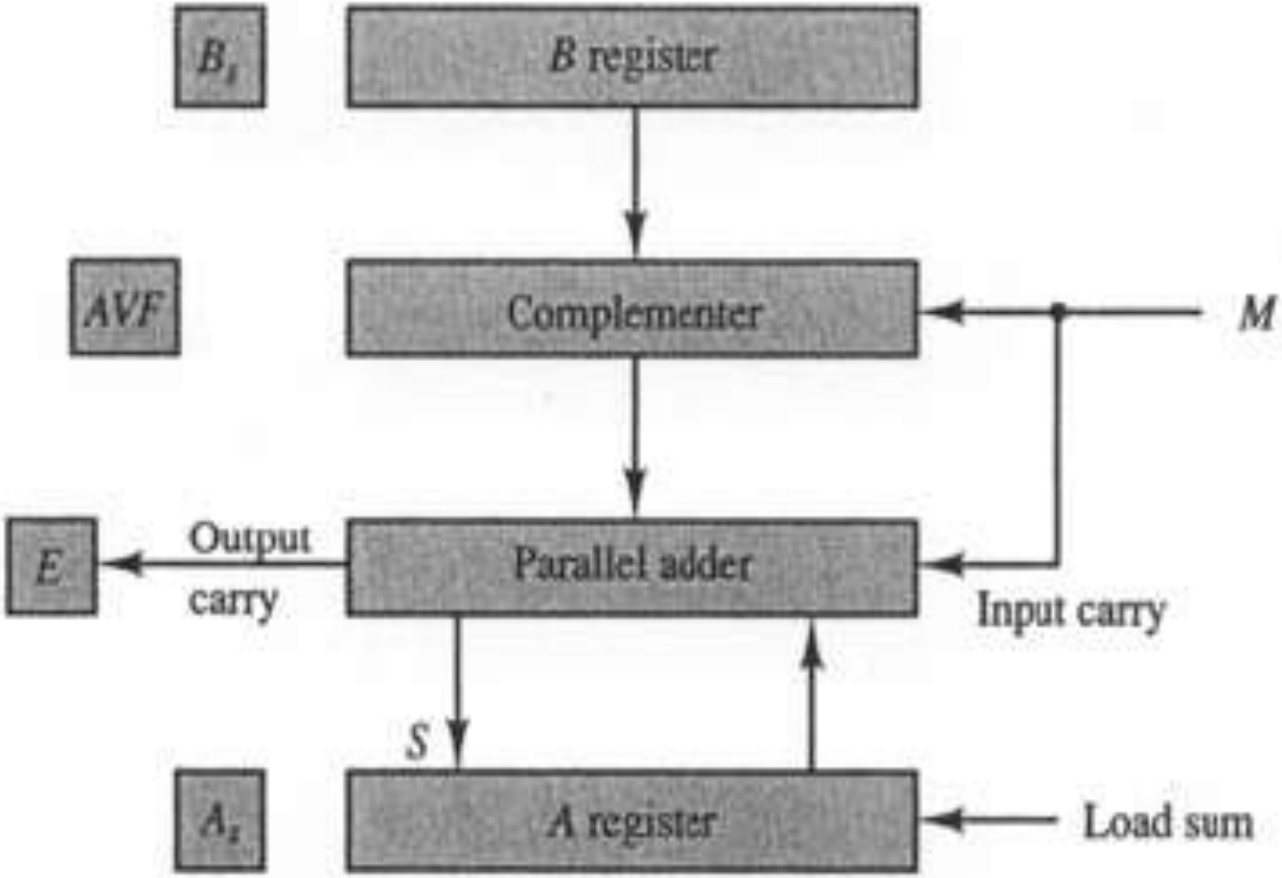
Hardware Implementation For Signed Magnitude Add & Sub

- To implement the two arithmetic operations with hardware, it is first necessary that the two numbers be stored in registers.
- Let A and B be 2 registers that holds the magnitudes of the number, and A_s and B_s be 2 flip flops that hold the corresponding signs.
- The result of the operation may be transferred to the third register
- However, a saving is achieved if the result is transferred into A & $A_{s.}$

- Thus A and A_s together form an accumulator register.
- Consider now the hardware implementation of the algorithm above.
- Firstly, a parallel adder is needed to perform the micro operation A + B
- Second, a comparator circuit is needed to establish if A > B, A = B, or A < B
- Third, Two parallel subtractor circuits are needed to perform the microoperations A – B and B – A
- The sign relation ship can be determined from an exclusive OR gate with A_s and B_s as inputs.

- This procedure requites a magnitude comparator, an adder, & two subtractors.
- However, a different procedure can be found that requires less equipments.
- First, we know that subtraction can be accomplish by means of complement and add.
- Second, the result of a comparison can be determined from the end carry after the subtraction
- **Careful investigation of the alternatives reveals that the use of 2’s complement for subtraction and comparison is an efficient procedure that requires only an adder and a complementer**

Hardware


B_s
B register
AVF
Complementer
M (Mode control)
E
Output carry
Parallel adder
Input carry
S
A_s
A register
Load sum

- Figure 10.1 shows a block diagram of the hardware for implementing the addition and subtraction operation.
- It consists of register A & B and sign flip flop's A_s and B_s.
- Subtraction is done by adding A to the 2's complement of B.
- The output carry is transferred to flip flop E, where it can be checked to determine the relative magnitude's of the two numbers.
- The add-overflow flip flop AVF holds the overflow bit when A and B are added.

- The addition of A plus B is done through the parallel adder.
- The S (sum) output of the adder is applied to the input of the A register
- The complementer provides and output of B or the complement of B depending on the state of the mode control M.
- The complementer consists of exclusive OR gates and the parallel adder consists of a full adder circuit as shown in fig 4.7.
- The M signal is also applied to the input carry of the adder.
- When M = 0, the output of B is transferred to the adder, the input carry is 0, and the output of adder is equal to the sum A + B

- When $M = 1$, the 1's complement of B is applied to the adder, the input carry is 1 and the output $S = A + B' + 1$.
- This equal to A plus 2's complement of B, which is equivalent of $A – B$.

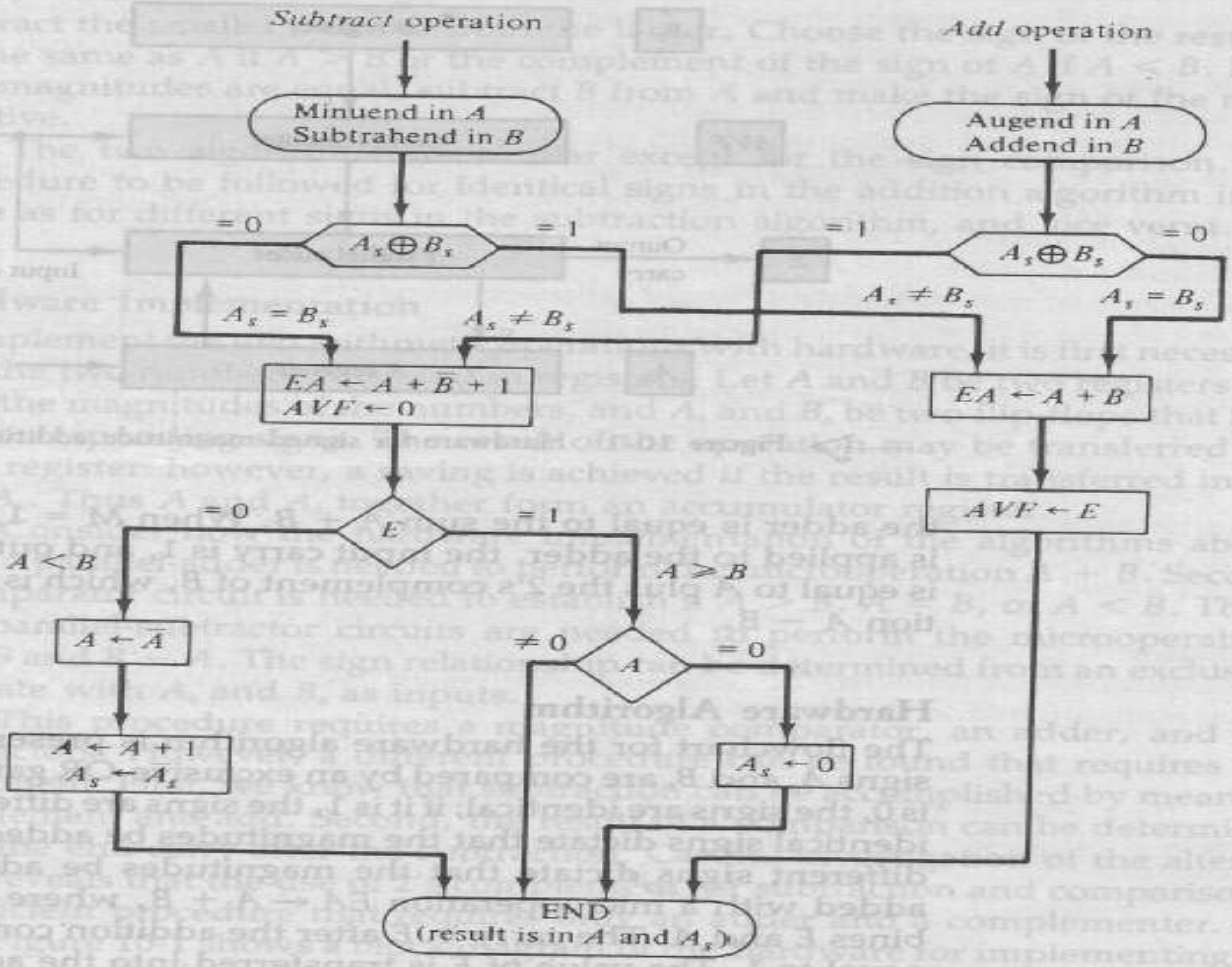


Figure 10-2 Flowchart for add and subtract operations.

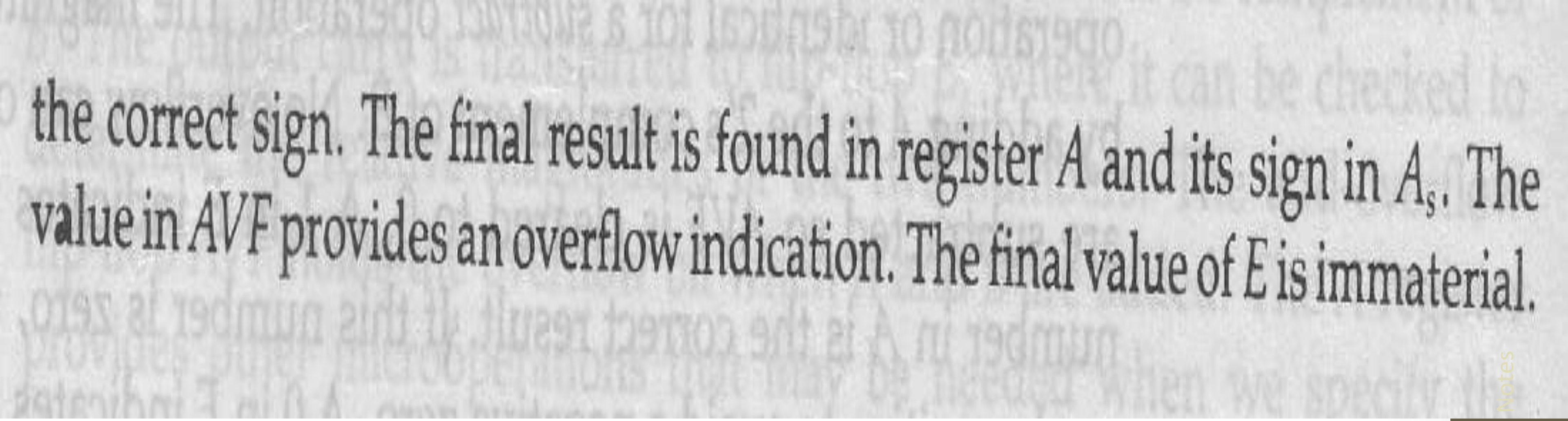
Hardware Algorithm

The flowchart for the hardware algorithm is presented in Fig. 10-2. 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 combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.

different for an *add*

equal to 1. The

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 1 in E indicates that $A \geq B$ and the number in A is the correct result. If this number is zero, the sign A_s 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 $A \leftarrow \overline{A} + 1$. 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_s 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_s to obtain

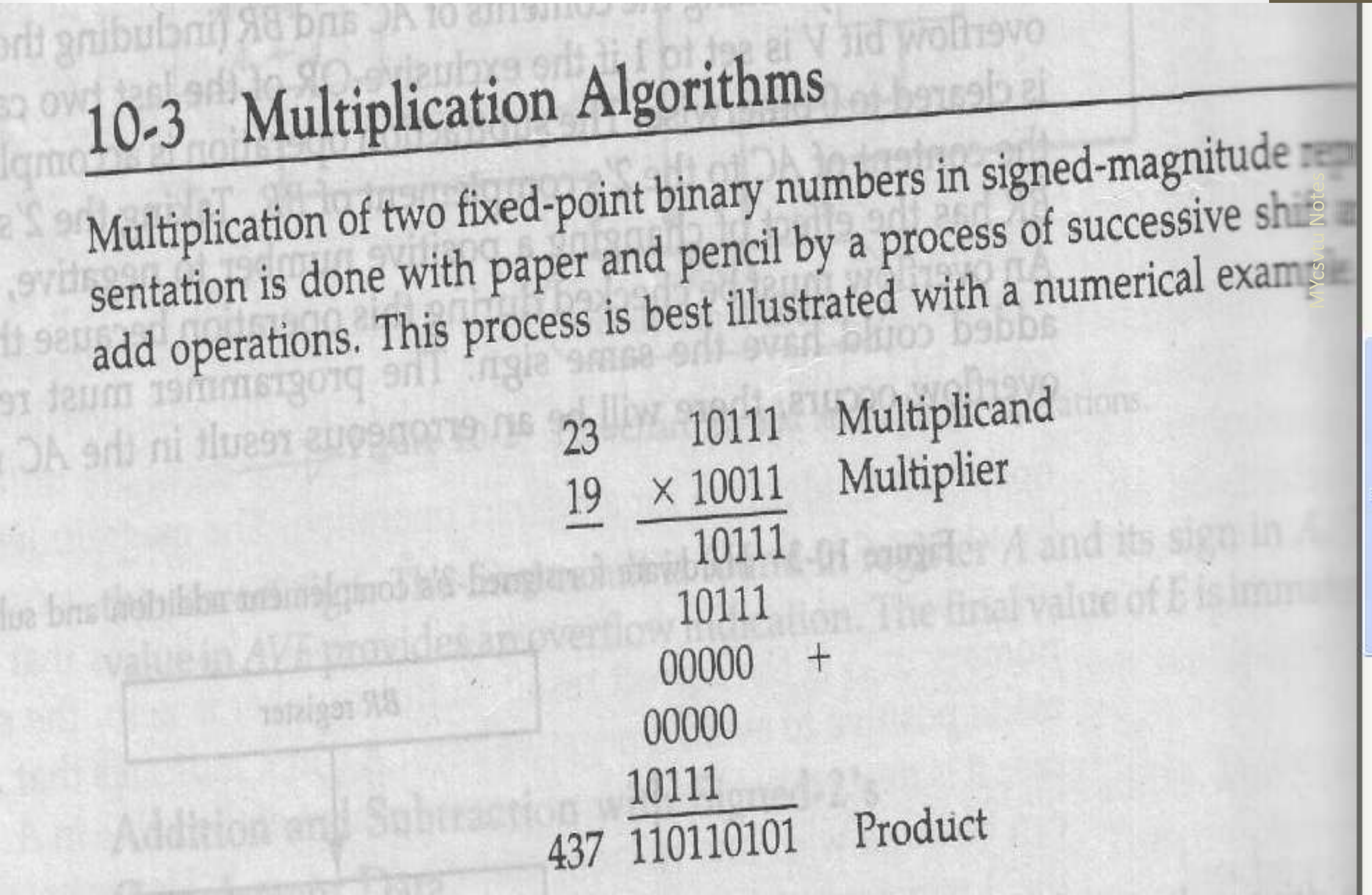


the correct sign. The final result is found in register *A* and its sign in A_s. The value in *AVF* provides an overflow indication. The final value of *E* is immaterial.

Assignment

Write an algorithm for adding and subtracting numbers unsigned 2’s complement representation

Multiplication Algorithm



10-3 Multiplication Algorithms

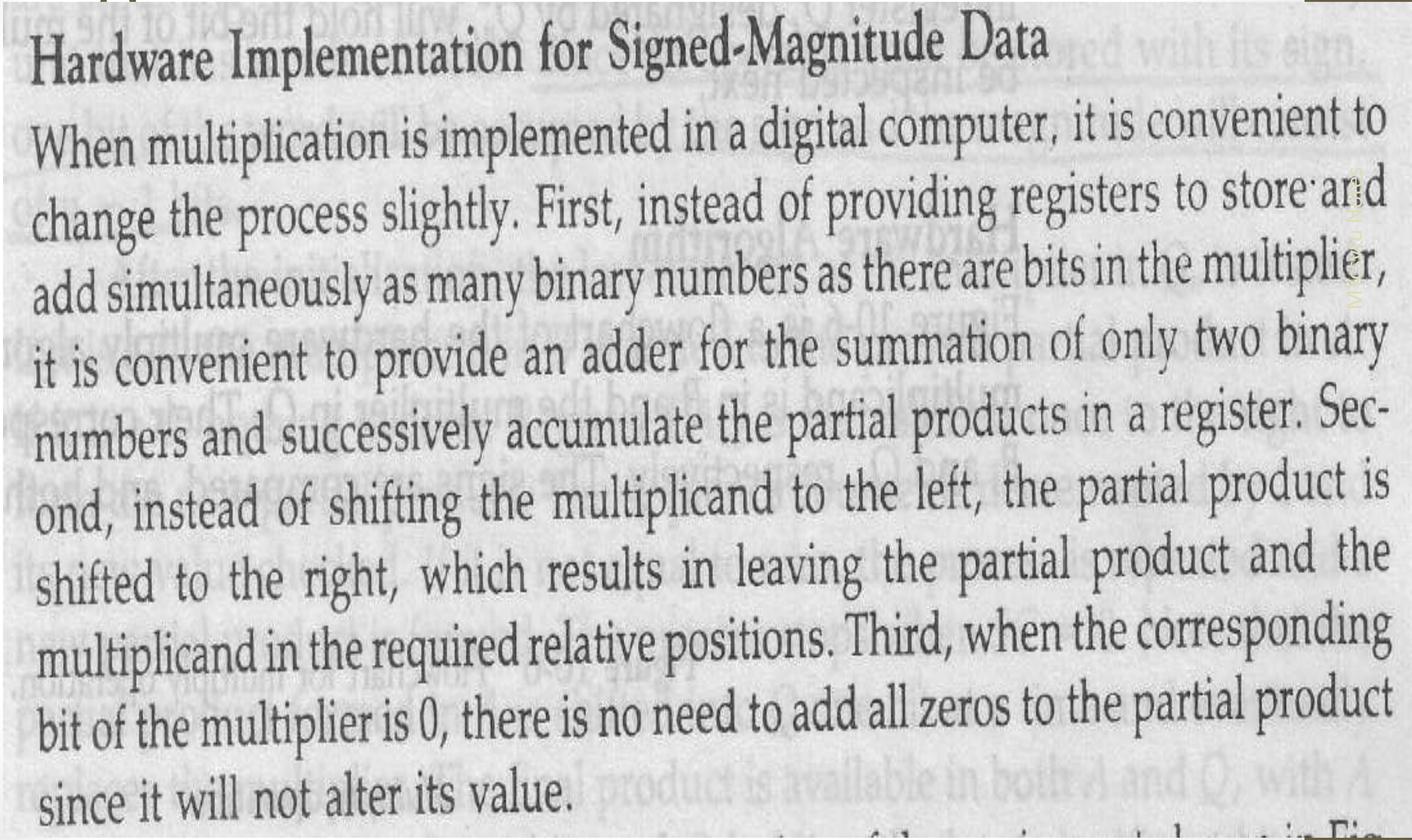
Multiplication of two fixed-point binary numbers in signed-magnitude re sentation is done with paper and pencil by a process of successive shi add operations. This process is best illustrated with a numerical exam

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

Process for multiplicatio

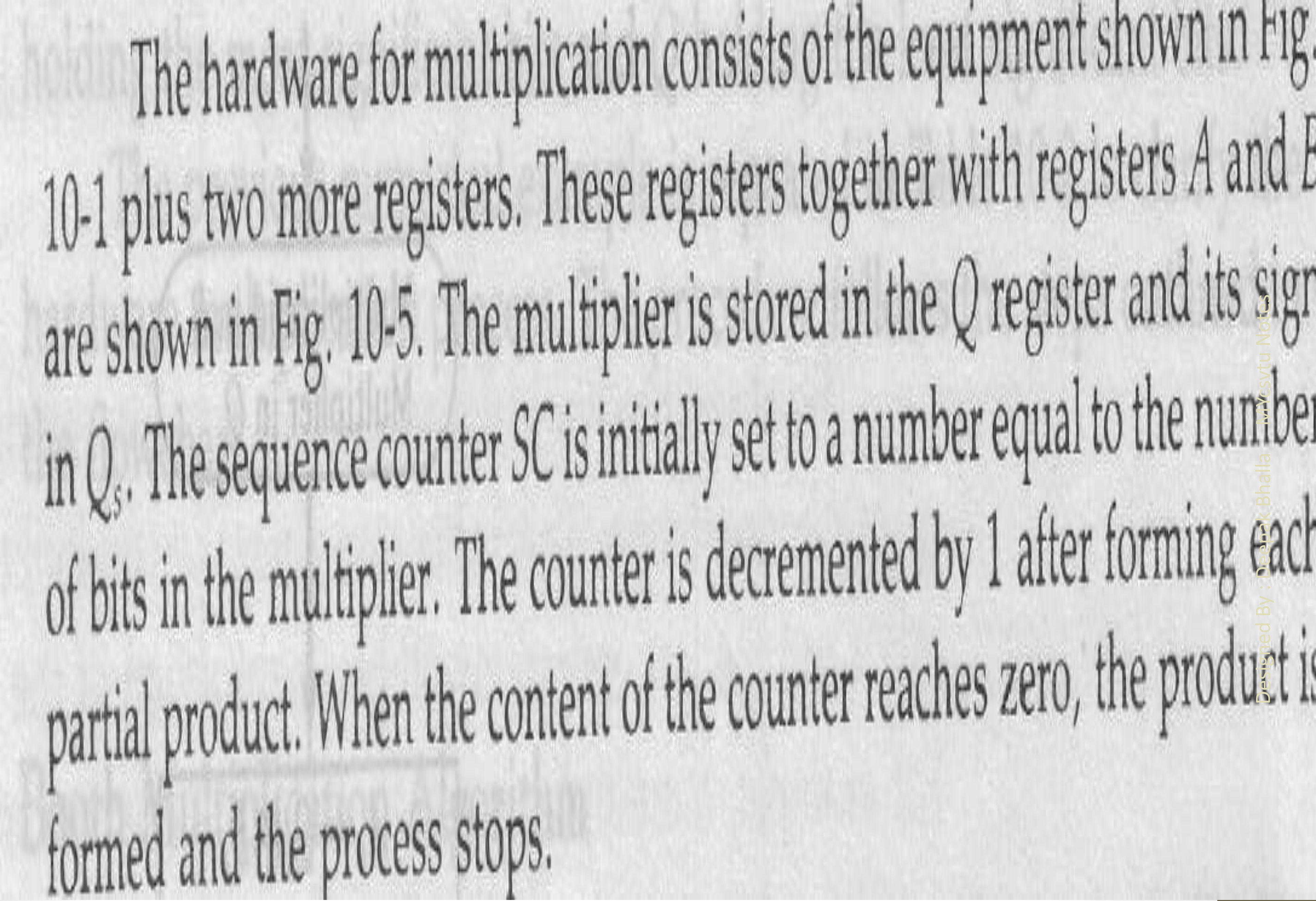
- The process consists of looking at successive bits of the multiplier, least significant bit first.
- If the multiplier bit is a 1, the multiplicand is copied down;otherwise, zeroes are copied down.
- The numbers copied down in successive lines are shifted one position to the ;left from the previous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

Hardware Implementation for signed magnitude data



Hardware Implementation for Signed-Magnitude Data

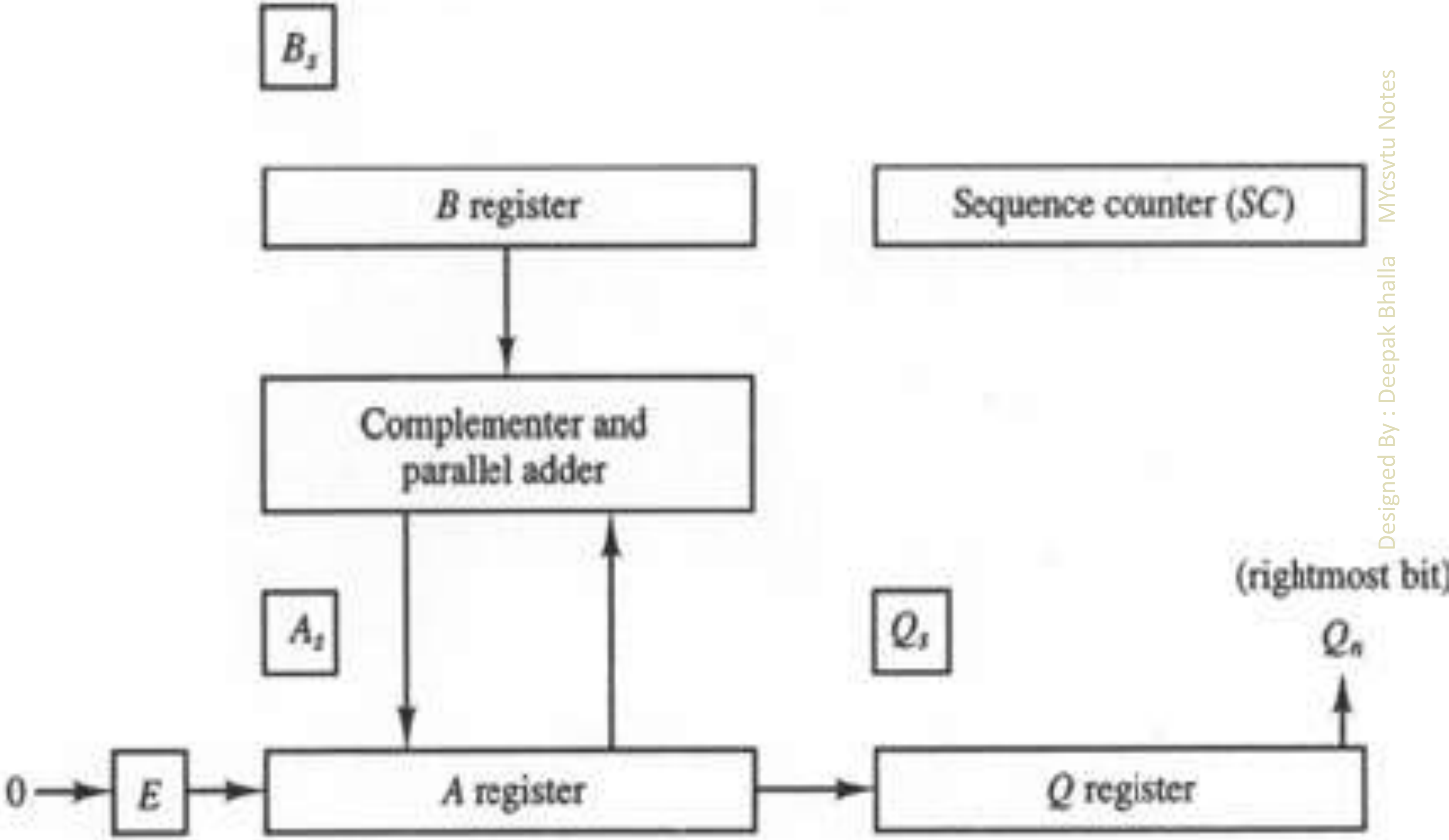
When multiplication is implemented in a digital computer, it is convenient to change the process slightly. First, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in a register. Second, instead of shifting the multiplicand to the left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required relative positions. 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.



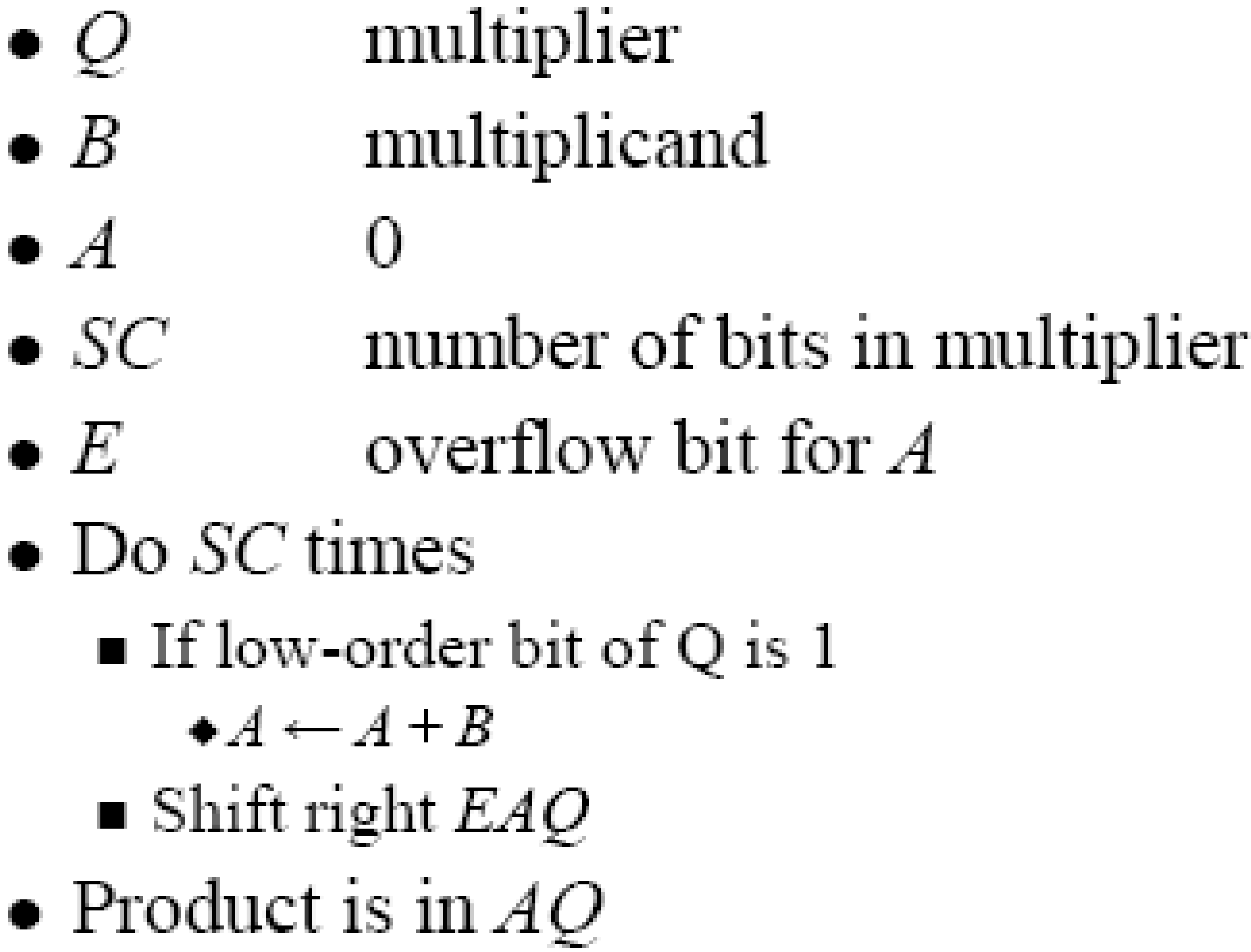
The hardware for multiplication consists of the equipment shown in Fig. 10-1 plus two more registers. These registers together with registers *A* and B are shown in Fig. 10-5. The multiplier is stored in the *Q* register and its sign in Q_s. The sequence counter *SC* is initially set to a number equal 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.

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 partial product and multiplier are shifted to the right. This shift will be denoted by the statement shr EAQ to designate the right shift depicted in Fig. 10-5. The 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 is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip-flop in register Q, designated by Q_n, will hold the bit of the multiplier, which must be inspected next.

Hardware


B_s
B register
Sequence counter (SC)
Complementer and parallel adder
(rightmost bit)
A_s
Q_s
Q_n
0
E
A register
Q register

Description



- Q multiplier
- B multiplicand
- A 0
- SC number of bits in multiplier
- E overflow bit for A
- Do SC times
    - If low-order bit of Q is 1
        - $A \leftarrow A + B$
    - Shift right EAQ
- Product is in AQ

Hardware Algorithm

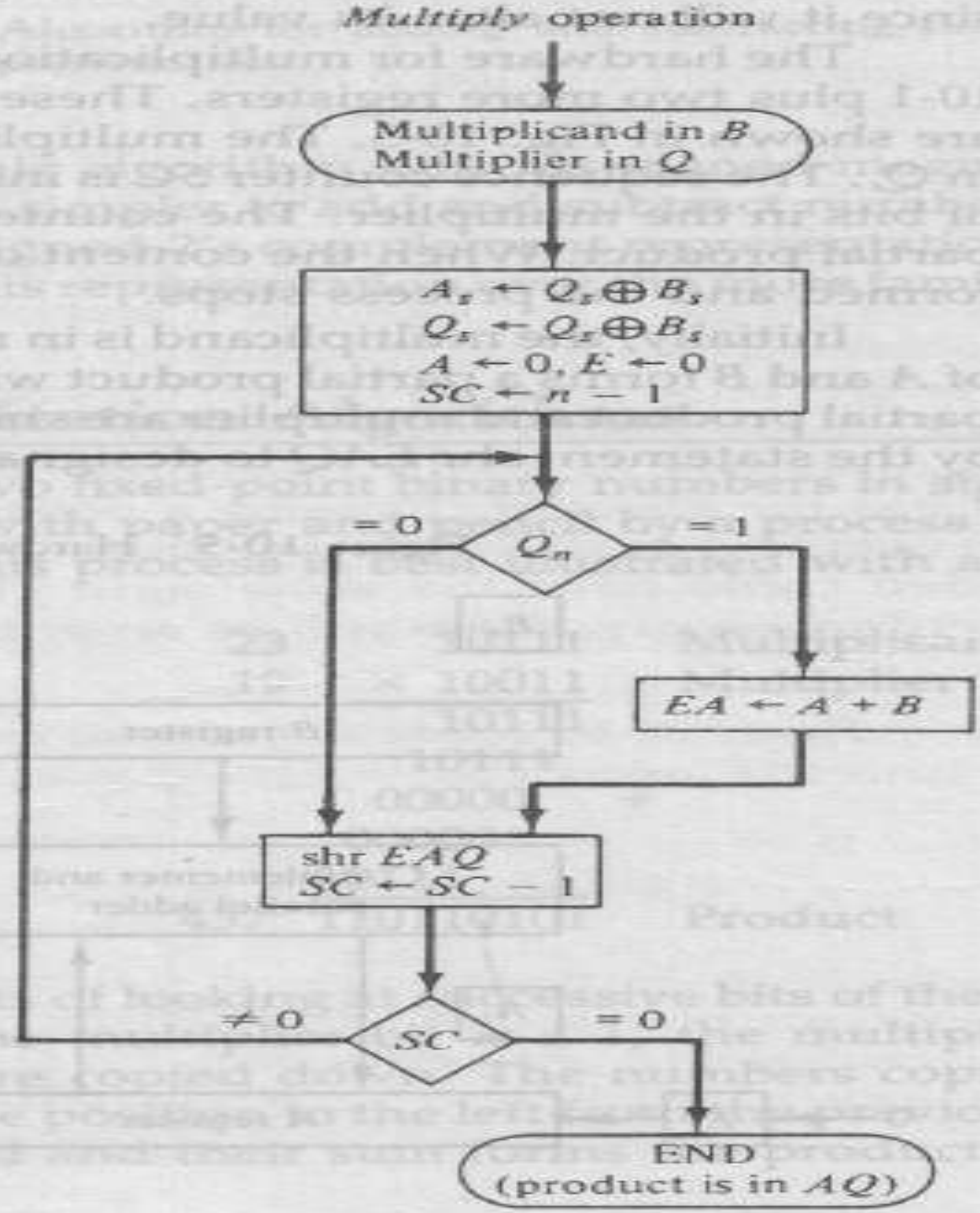
Figure 10-6 is a flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in B_s and Q_s, respectively. The signs are compared, and both A and Q are set

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.

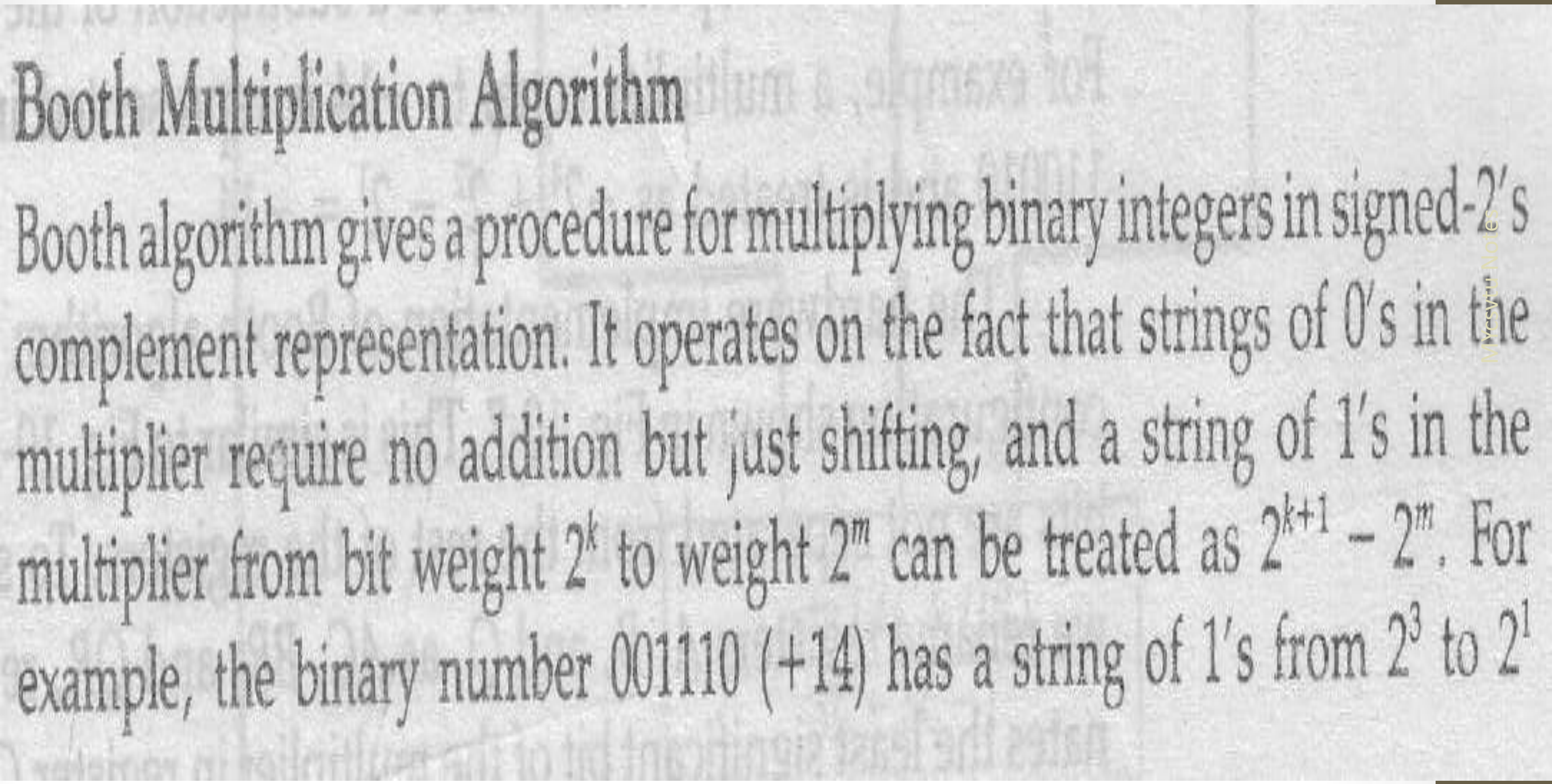
After the initialization, the low-order bit of the multiplier in Q_n is tested. If it is a 1, the multiplicand in *B* is added to the present partial product in *A*. If it is a 0, nothing is done. Register *EAQ* is then shifted once to the right to form the new partial product. The sequence counter is 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.

The previous numerical example is repeated in Table 10-2 to clarify the hardware multiplication process. The procedure follows the steps outlined in the flowchart.

Figure 10-6 Flowchart for multiply operation.



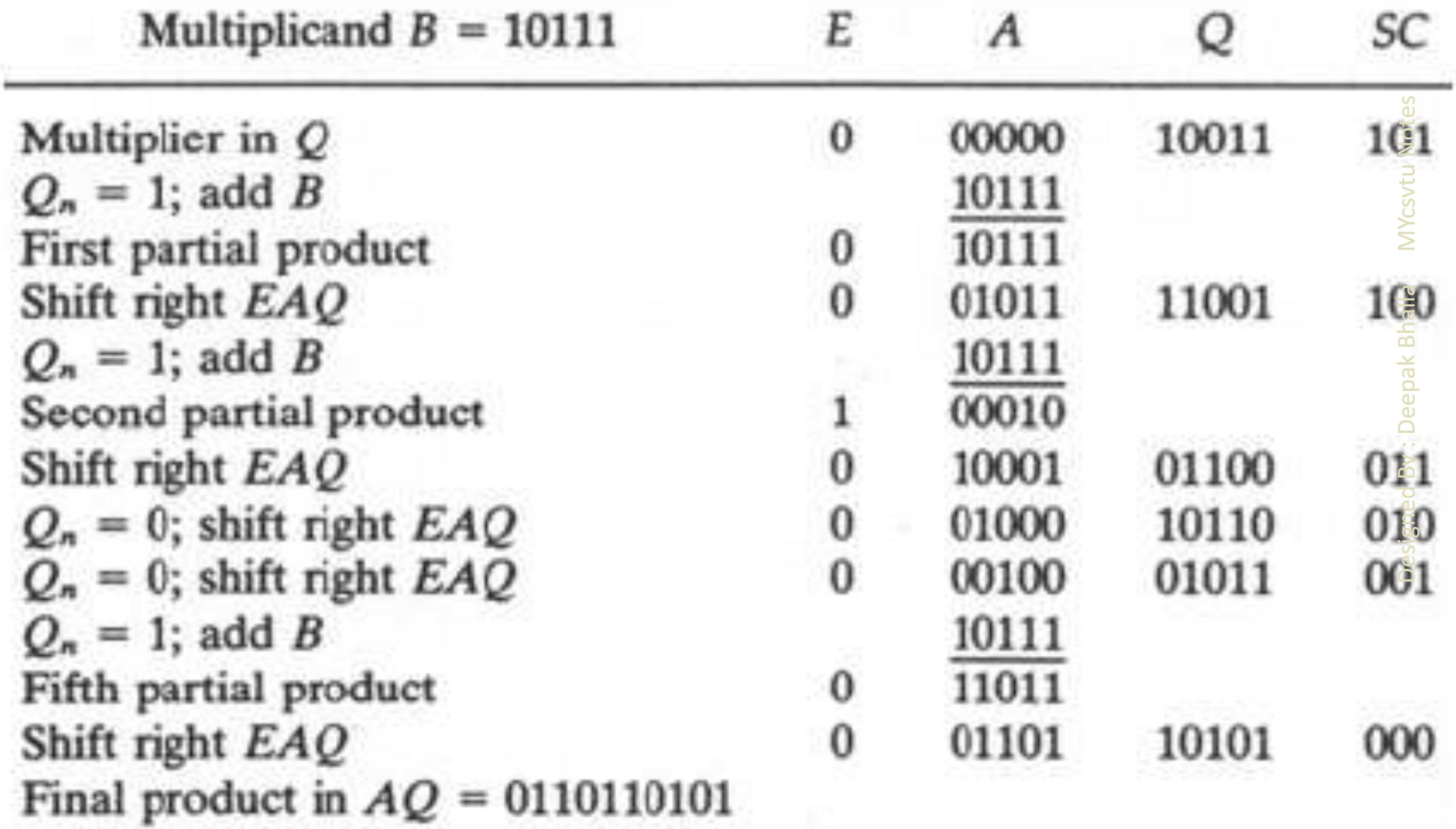
Booth Multiplication Algorithm



Booth Multiplication Algorithm

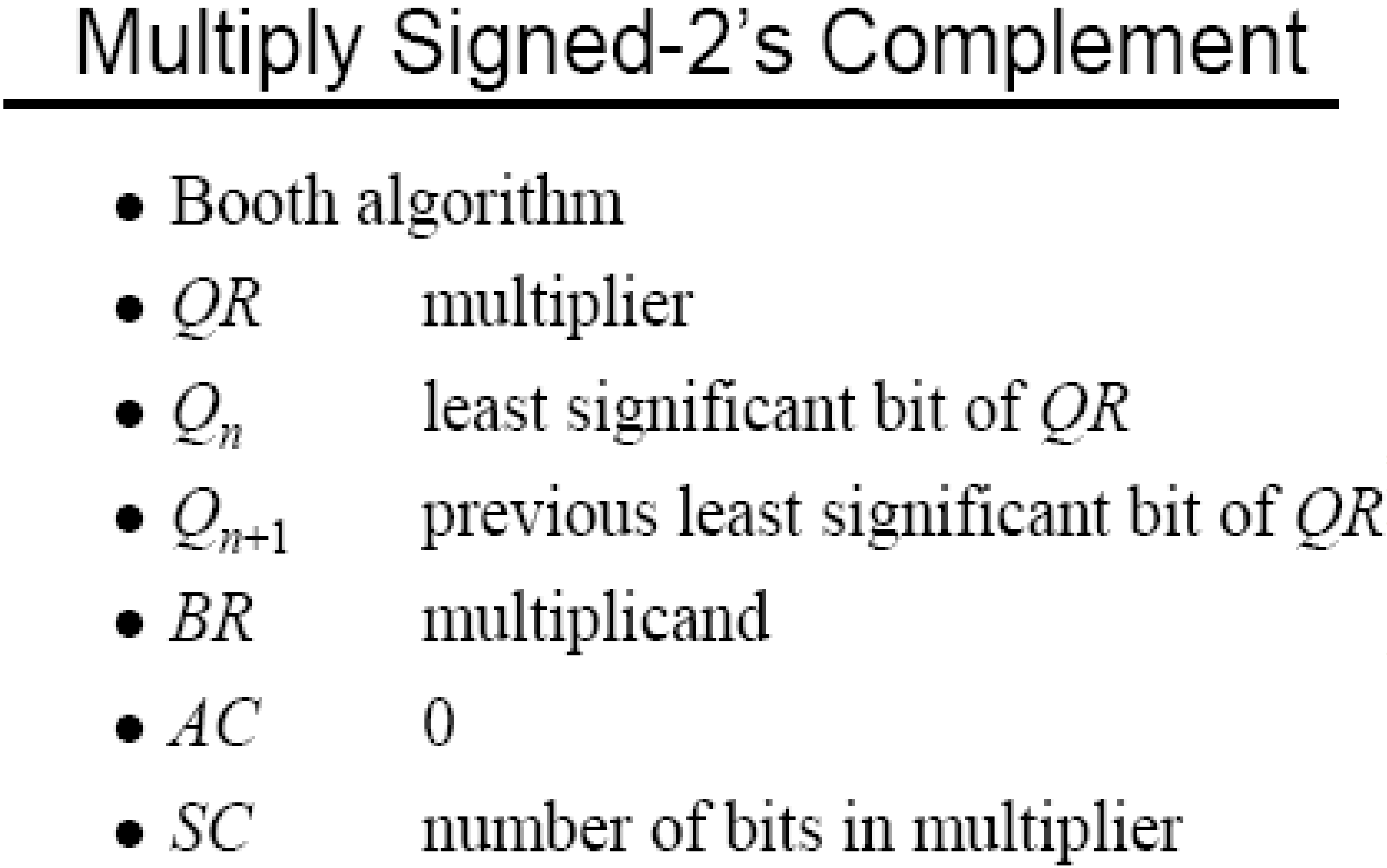
Booth algorithm gives a procedure for multiplying binary integers in signed-2′s complement representation. It operates on the fact that strings of 0′s in the multiplier require no addition but just shifting, and a string of 1′s in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$. For example, the binary number 001110 (+14) has a string of 1′s from 2^3 to 2^1

Example: 23 x 19 = 437



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 $AQ = 0110110101$				

Numerical Example for binary multiplier



Multiply Signed-2's Complement

- Booth algorithm
- QR multiplier
- Q_n least significant bit of QR
- Q_{n+1} previous least significant bit of QR
- BR multiplicand
- AC 0
- SC number of bits in multiplier

Division Algorithm

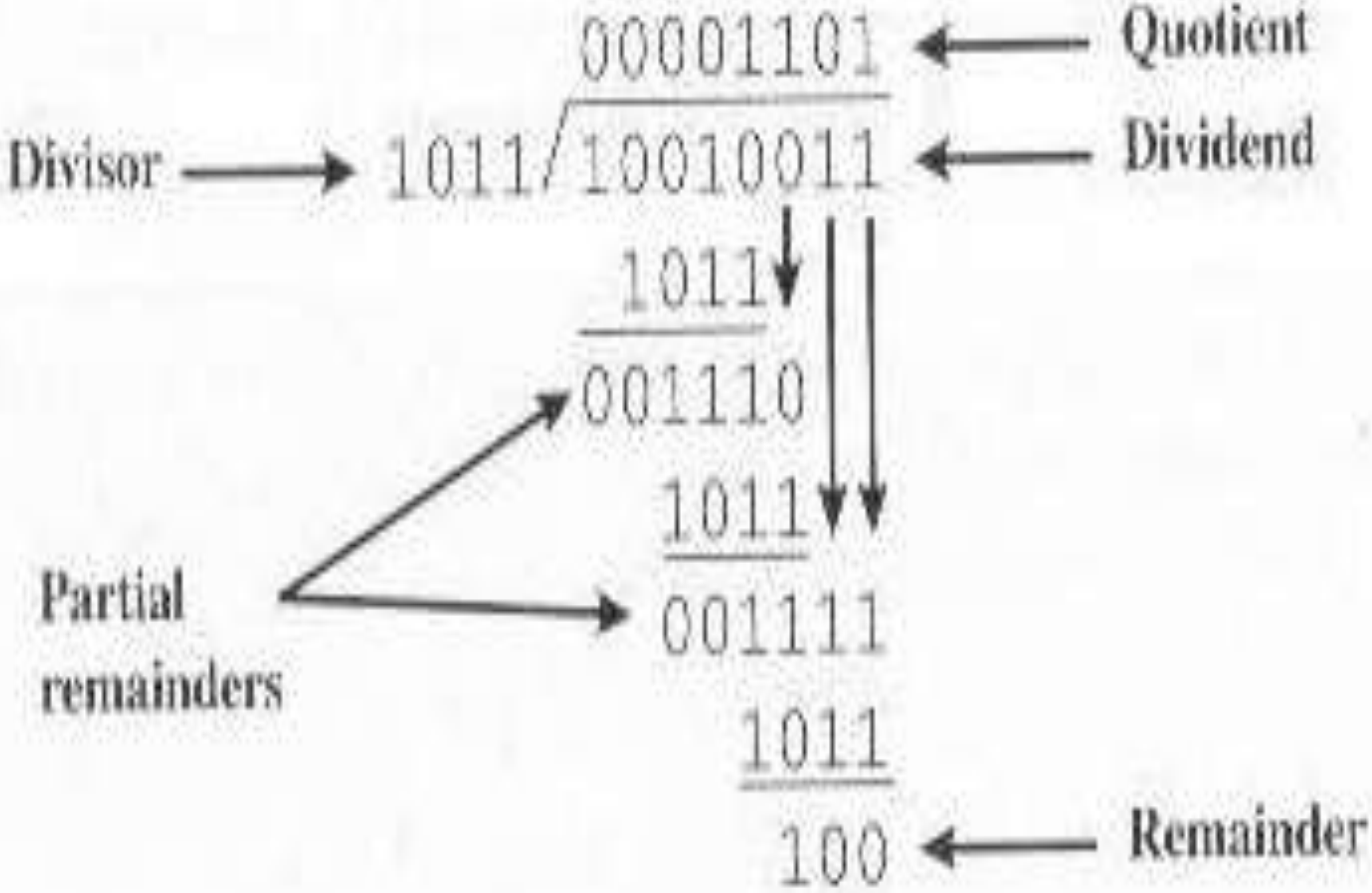


Figure 9.15 Example of Division of Unsigned Binary Integers

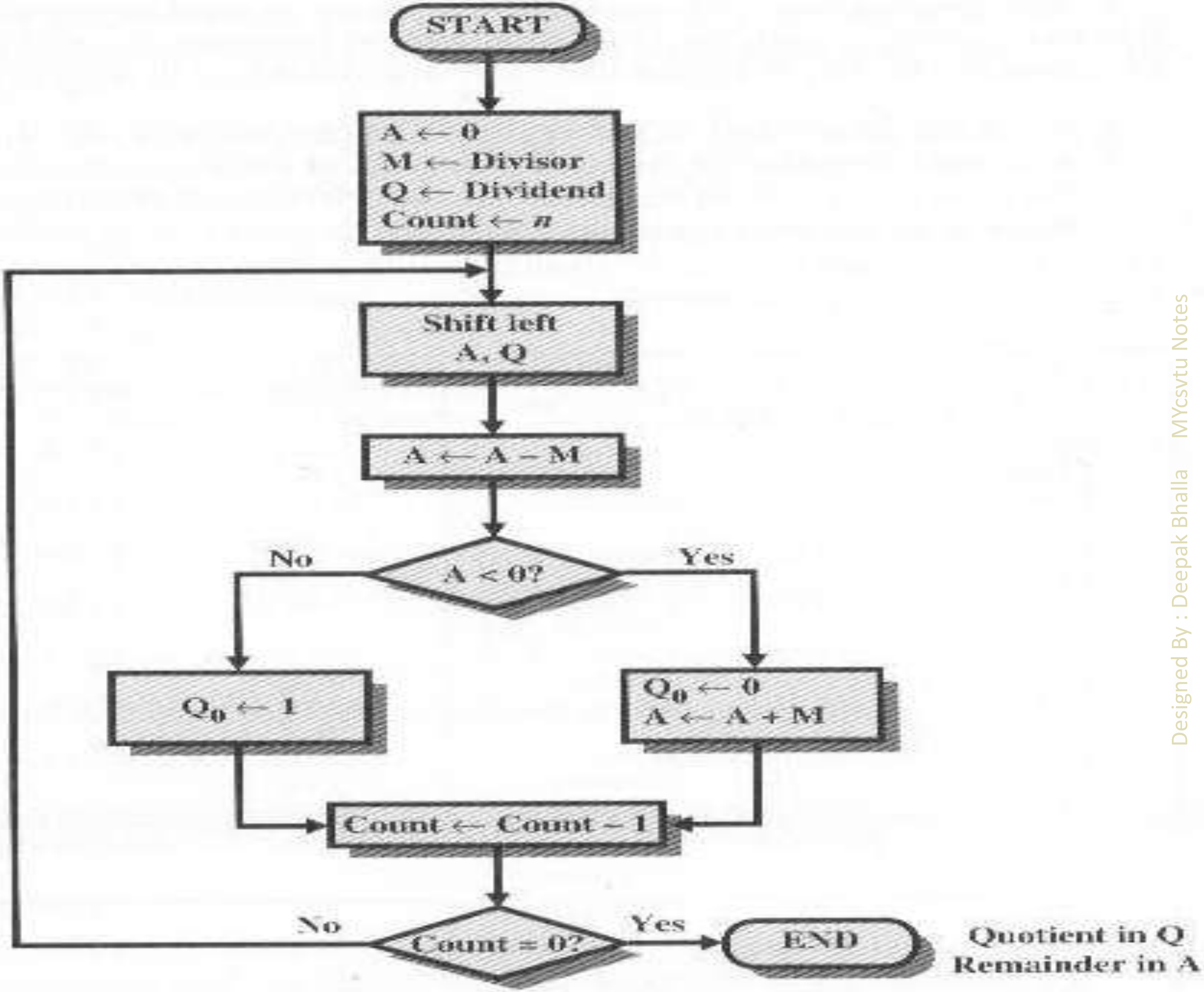


Figure 9.16 Flowchart for Unsigned Binary Division

- Figure shows a machine algorithm that corresponds to the long division process.
- The divisor is placed in the M register, the dividend in the Q register.
- At each step, the A and Q registers together are shifted to the left 1 bit.
- M is subtracted