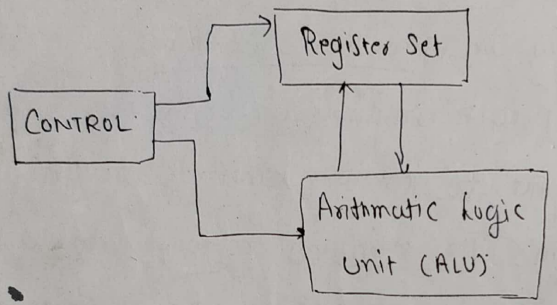
MODULE-3 Central Processing Unit"

Central Processing Unit

The part of the computers that performs the bulk of data processing operations is called the central processing unit and referred to as CPU. The CPU is made of 3- major parts shown as below:-



[Major Component of CPU].

1. Register set stores the intermediate data used during the execution of instructions.
2. The arithmetic logic unit (ALU) perform the require microoperation for executing the instructions.
3. The control unit supervise the transfer of information among the registers and instruct the ALU as to which operation to perform.

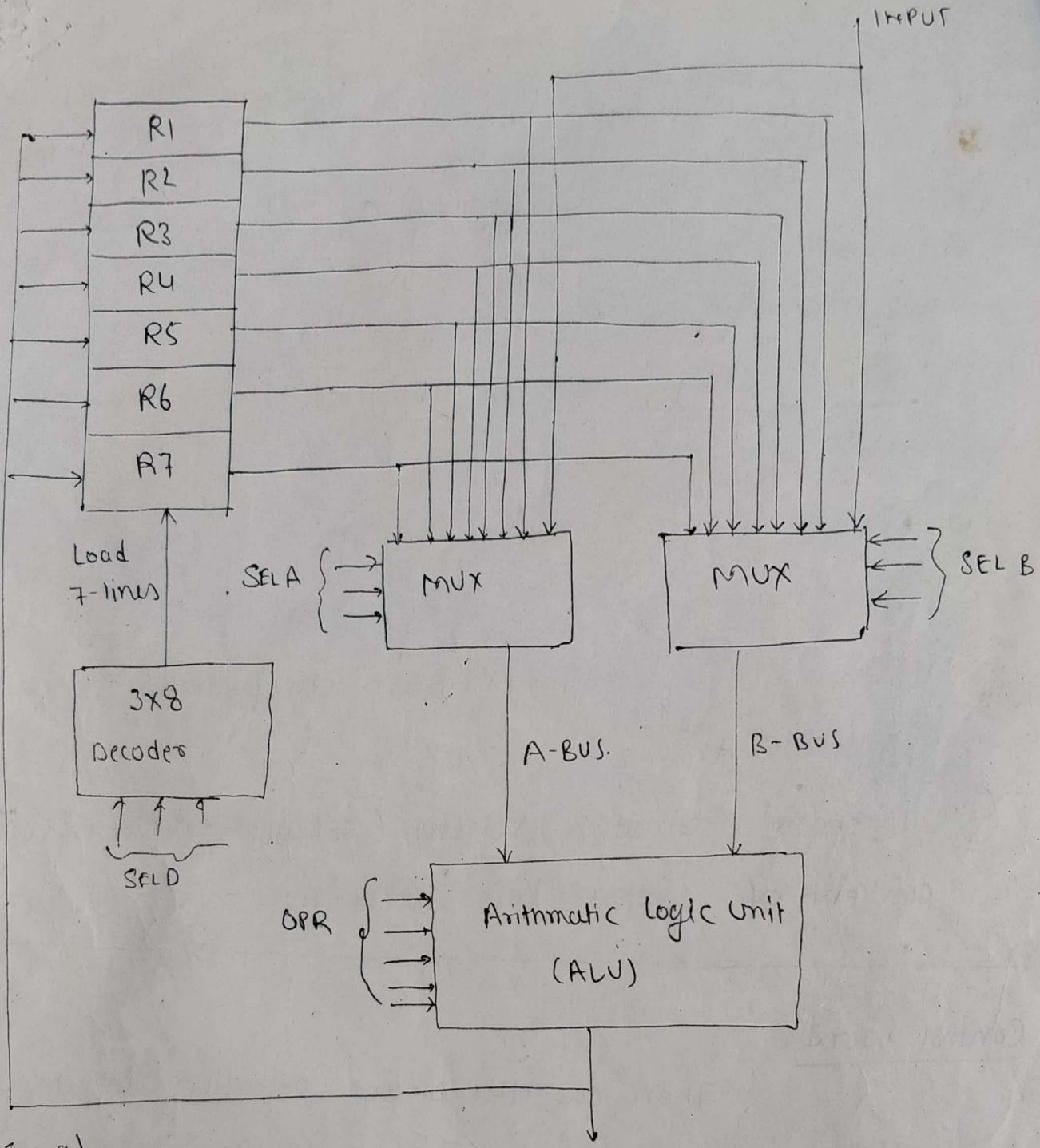
CPU performs a variety of functions dictated by the type of instructions that user are machine incorporated in computer.

General Register Organization

In the programming environment, the memory locations are needed for storing pointers, counters, return addresses, temporary results and partial products during various microoperations. But memory access is most time consuming process in computer, so it is more convenient and more efficient to store these intermediate values in processor registers. When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfer, but also while performing various microoperations.

A bus organization for 7-CPU registers is shown in fig (a).

The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection line in each multiplexer select one register or the input data for the particular bus. The A and B buses form the input to a common arithmetic logic unit (ALU). Operation selected in the ALU determines the arithmetic or logic microoperation that is to be performed.



(Fig-a) [BLOCK DIAGRAM]

The result of the microoperation is available for o/p data and also goes into the input of all registers. The register that receives the information from the output bus is selected by decoder. The decoder activates one of the register load input, thus providing a transfer path between the data in the

output bus and the inputs of selected destination register.

e.g. For example if we have to perform the following operation:-

$R1 \leftarrow R2 + R3$

then control must provide binary selection variables to following selector inputs:-

(1) MUX A Selector (SELA): to place the content of R2 into bus A.

(2) MUX B Selector (SELB): to place the content of R3 into bus B.

(3) ALU operation selector (OPR): to provide the arithmetic addition A+B.

(4) Decoder Destination selector (SELD): to transfer the content of output bus into R1.

Control word:-

There are 14- binary selection inputs in the unit, and their combined value specifies a control word. The 14-bit control word is as follows:-

3	3	3	5
SELA	SELB	SELD	OPR

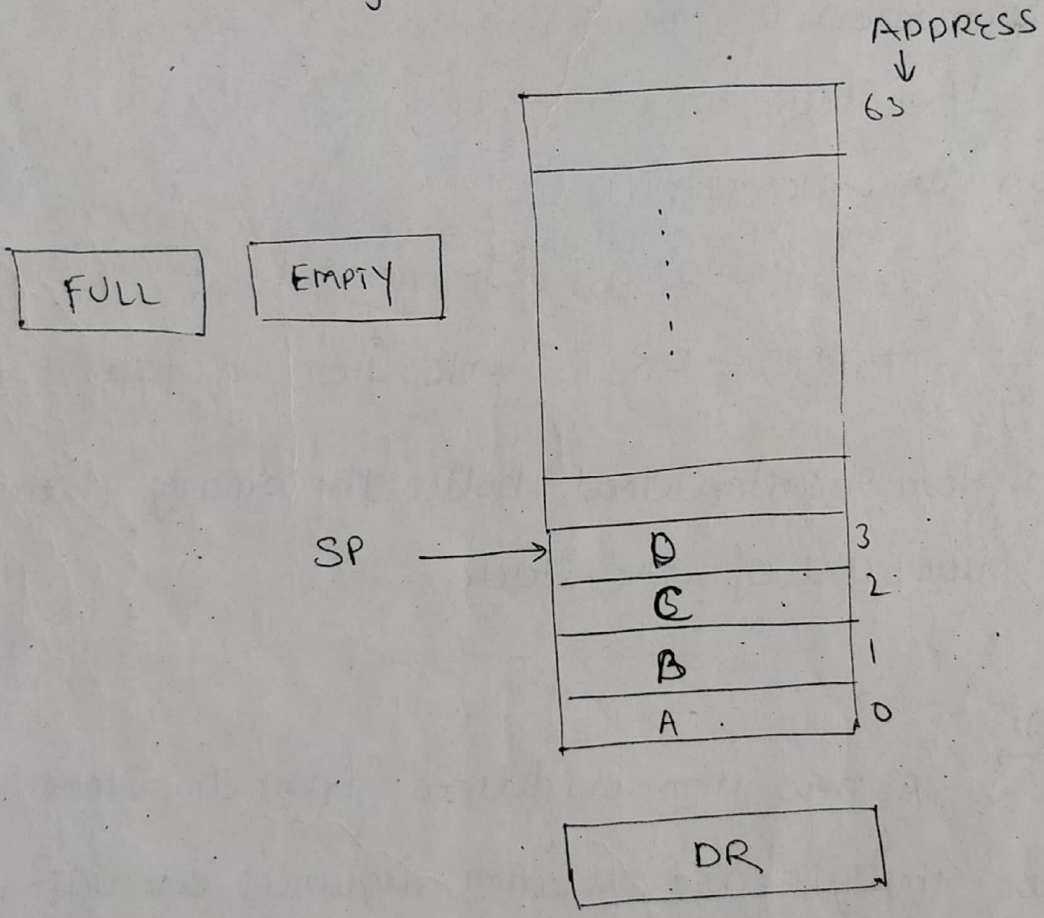
[control word].

It consists of 4- fields. Three fields contain three bits each and one field has 5- bits.

Stack Organization →:

A stack is a storage device that stores information in such a manner that the item stored last is the first item retrived. So, Stack is last-in first-out (LIFO) device. The register that holds the address for the stack is called a stack pointer (SP), because its value always points to the top item of the stack.

Block diagram for 64-word stack is as:-



Basic operation of a Stack →

There are basic 2-operation of Stack:-

① Push or insertion operation:- The operation of insertion of new item in ~~thought~~ called Push operation, because it can be thought of the result of pushing a

new item on top.

(2) Pop or Deletion :- The operation of deletion is called pop-up because it can be thought of as the result of removing one item so that stack pops-up.

PUSH OPERATION :-

Initially SP is ~~cleared~~ cleared to '0', EMPTY is set to 1, and FULL is cleared to '0'. So that SP points to the word at address '0' and Stack is marked empty and not full. If Stack is not full i.e. FULL=0, a new item can be inserted as:-

SP ← SP+1	Increment stack pointer.
M[SP] ← DR	write item on top of stack.

DR is data register that holds the binary data to be written or read out of the stack.

Pop operation :-

A new item is deleted from the stack if stack is not empty, and operation sequences are as:-

DR ← M[SP]	Read item from top of stack.
SP ← SP-1	Decrement stack pointer
If (SP=0) then EMPTY ← 1	Check if stack is empty.
FULL ← 0	Mark the stack not full.

Reverse Polish Notation →

A stack organization is very effective for evaluating arithmetic expressions. Common arithmetic operations are written in infix notation. in which operator written between operands. There are three type of notations for arithmetic operation as follows:-

① Infix notation:-
In infix notation operator is placed between the operands. e.g. (A*B)+(C*D) A+B

② Prefix notation:-
In prefix notation, operator is placed before the operands. eg.

A+B infix notation
+AB Prefix notation.

③ Postfix notation:- In postfix notation, operator is placed after the operands.

e.g. A+B infix notation
AB+ postfix notation.

Evaluation of arithmetic operation by using stack →

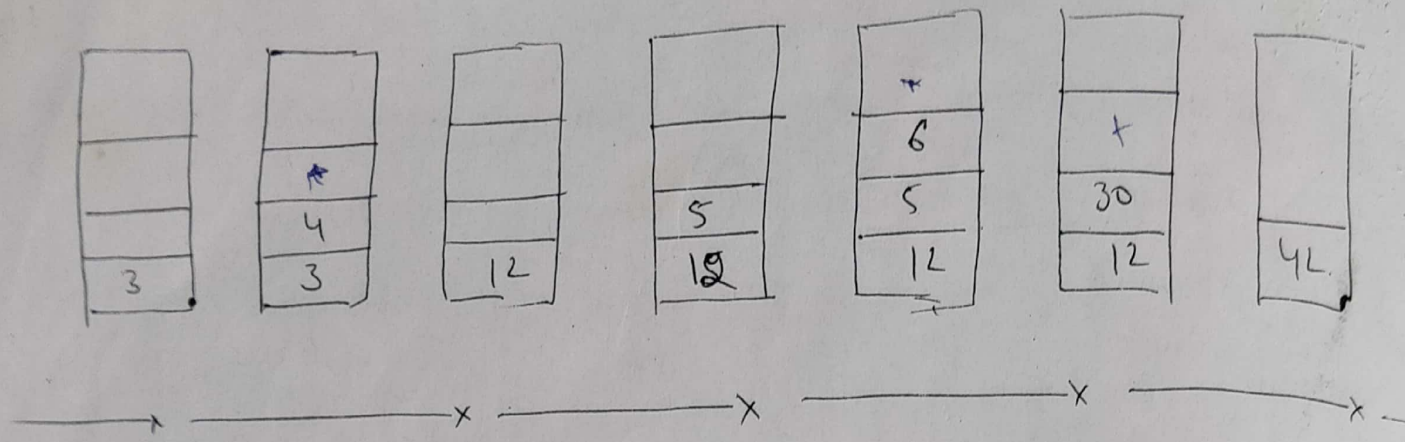
Suppose we have the arithmetic expression:-

(3*4)+(5*6).

2*3+ 6/2*12

calculate postfix notation i.e. 34*56*+

New Stack operations are as follows:-

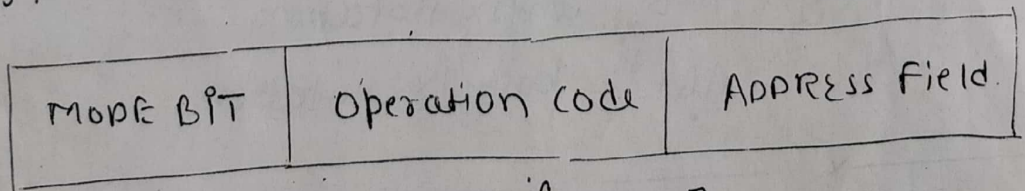


Instruction Formats

An instruction format has the following fields:-

1. An operation code field that specifies the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

and diagram shown as:-



[Instruction format].

Type of instructions

Following are the type of instructions:-

① Three- Address Instructions →

Computer with three-address instruction formats can use each address field to specify either a processor register or a memory operand. Following program evaluates $X = (A*B)*(C+D)$ as:-

```
ADD   R1, A, B,      R1 ← M[A] + M[B],
ADD   R2, C, D       R2 ← M[C] + M[D],
MUL   X, R1, R2      M[X] ← R1 * R2;
```

The advantage of three address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that binary-coded instruction requires too many bits to specify three addresses.

② Two-Address Instructions:-

Two-address instructions are the most common in commercial computers. Each address field can specify either a processor register or a memory word. Program to generate $X = (A+B)*(C+D)$ is as:-

```
MOV   R1, A      R1 ← M[A];
ADD   R1, B      R1 ← R1 + M[B];
MOV   R2, C      R2 ← M[C];
ADD   R2, D      R2 ← R2 + M[D];
MUL   R1, R2     R1 ← R1 * R2
MOV   X, R1      M[X] ← R1
```

③ One-Address Instructions →: One address instructions use an accumulator (AC) register for all data manipulation. For multiplication and division there is need for a second register.

Program to evaluate X = (A+B) * (C+D) is as:-

LOAD	A	AC ← M[A]
ADD	B	AC ← AC + M[B]
STORE	T	M[T] ← AC
LOAD	C	AC ← M[C]
ADD	D	AC ← AC + M[D]
MUL	T	AC ← AC * M[T]
STORE	X	M[X] ← AC.

④ Zero-address Instructions →: A stack organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicate with Stack. Program to evaluate X = (A+B) * (C+D) is as follows:-

PUSH	A	TOS ← A
PUSH	B	TOS ← B
ADD		TOS ← A+B
PUSH	C	TOS ← C
PUSH	D	TOS ← D
ADD		TOS ← C+D
MUL		TOS ← (C+D) * (A+B)
POP	X	M[X] ← TOS.

TOS = Top of Stack

ADDRESSING MODES →

The addressing mode specifies a rule for interpeting or modifying the address field of the instruction. The way the operands are chosen during program execution is dependent on the addressing mode of instruction. Computer uses addressing mode due to following main reasons:-

(1) To give programming versatility (easibility) to the user by providing such facilities as pointer to memory, counter for loop control, indexing of data and program relocation.

(2) To reduce the number of bits in the addressing field of the instruction.

Different Type of addressing modes :-→

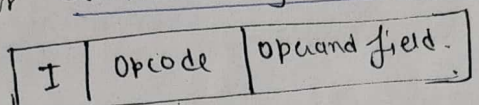
(1) Implied mode → / Direct addren mode

In implied mode, the operands are specified implicitly in the definition of the instruction. e.g. the instruction "complement accumulator" is an implied mode. because operand in accumulator register is implied in definition of instruction. Another example in zero-addren instructions in stack-organized computer are implied-mode instruction since operands are implied to be on top of stack.

(2) Immediate mode :→ / Indirect addren mode.

In this mode the operand is

specified in the instruction itself. In other words, immediate mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjuction with the operation specified in instruction. Immediate mode instructions are useful for initializing registers to a constant value.



(3) Register Indirect mode: In this mode, ~~operands are in register~~ instruction specify a register in the CPU whose contents give the address of operand in memory. In other words selected register contain the address of operand rather than operand itself. e.g. LXI H, 2500H - Load H-L Pair with 2500H
MOV A, M. → move content of memory location whose address in H-L pair (2500) to AC.

(4) Auto incremented/or autodecremented mode:

(4) Register mode: In this mode, operands are in registers that reside within CPU. Particular register is selected from a register field in the instruction. e.g. MOV A B. 78, ADD B.

(5) Auto increment or autodecrement mode: This is similar to register indirect mode except that the register in incremented or decremented after its value is used to access memory. When the address fired

the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.

(6) Direct address mode → In direct address mode, the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by address field of instruction.

(7) Indirect address mode → In this mode, address field of instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

In a few address modes requires that the address field of instruction be added to content of a specific register in CPU. The effective address in these mode is calculated as:-

Effective Address = address part of instruction + content of PC + address part of instruction

(8) Relative Address mode: In this mode content of Program counter is added to address part of instruction in order to obtain the effective address. The address part (is usually) of instruction is usually a signed

number which can be either positive or negative. When this number is added to content of program counter, the result produces an effective address whose position in memory is relative to address of the next instruction.

e.g. Program counter = 825

Address part of instruction = 25

Effective address computation for relative address mode is = 825 + 25 = 850.

So, effective address is 24 memory location forward from address of the next instruction.

(9) Index Addressing mode →: In index addressing mode content of index register is added to address part of instruction to obtain the effective address. The index register is a special CPU register that contains an index value. Address field of instruction defines the begining address of data array in memory.

(10) Base Register Addressing mode →: In base register addressing mode content of base register is added to the address part of instruction to obtain effective address. This is similar to index addressing mode except that the register is now called base register instead of index register.

Program Control - PC

Instructions are always stored in successive memory locations. When processed in the CPU, the instructions are fetched from consecutive memory locations and executed. Each time an instruction is fetch from memory, the program counter is incremented so that it contains the address of next instruction in sequence. After the execution of data transfer or data manipulation instruction, control returns to fetch cycle with the program counter containing the address of the instruction next in sequence. A program control instructions specify condition for altering the content of program counter, while data transfer and data manipulation instruction specify conditions for data processing operations.

Following are some typical instructions ~~are~~ Program Control are listed as below:-

Branch	BR
Jump	Jmp
Skip	SKP
Call	CALL
Return	RET
Compare	CMP.
Test	TST.

Reduced Instruction set computer (RISC) & CISC

CISC → In the early days, computer had small and simple instruction sets, forced mainly by the need to minimized the hardware used to implement them. Many computers have instruction sets that include more than 100 and sometimes even more than 250 instructions. A computer with a large number of instructions is classified as a complex Instruction set computer.

RISC → It is recommended that computers use fewer instructions with simple construct so they can be executed much faster within the CPU without having to use memory as often. This type of computers is classified as a reduced instruction set computer or RISC.

CISC characteristics

1. A large number of instructions - typically from 100 to 250 instructions.
2. Some instructions that perform specialized tasks and are used infrequently.
3. A large variety of addressing modes - typically from 5 to 20 different mode.

(4) Variable length instruction formats.

(5) Instruction that manipulate operands in memory.

RISC characteristics

(1) Relatively few instructions.

(2) Relatively few addressing modes.

(3) Memory access limited to load and store instructions.

(4) All operations done within the registers of the CPU.

(5) ~~Fixed~~ Fixed - length, easily decoded instruction format.

(6) Single-cycle instruction execution.

(7) ~~Hard~~

(8) RISC is ~~hardwired~~ Hardwired rather than ~~microprogrammed~~ microprogrammed control.

MULTIPLICATION ALGORITHMS

Multiplication of two fixed-point numbers in signed-magnitude representation is done with following example:-

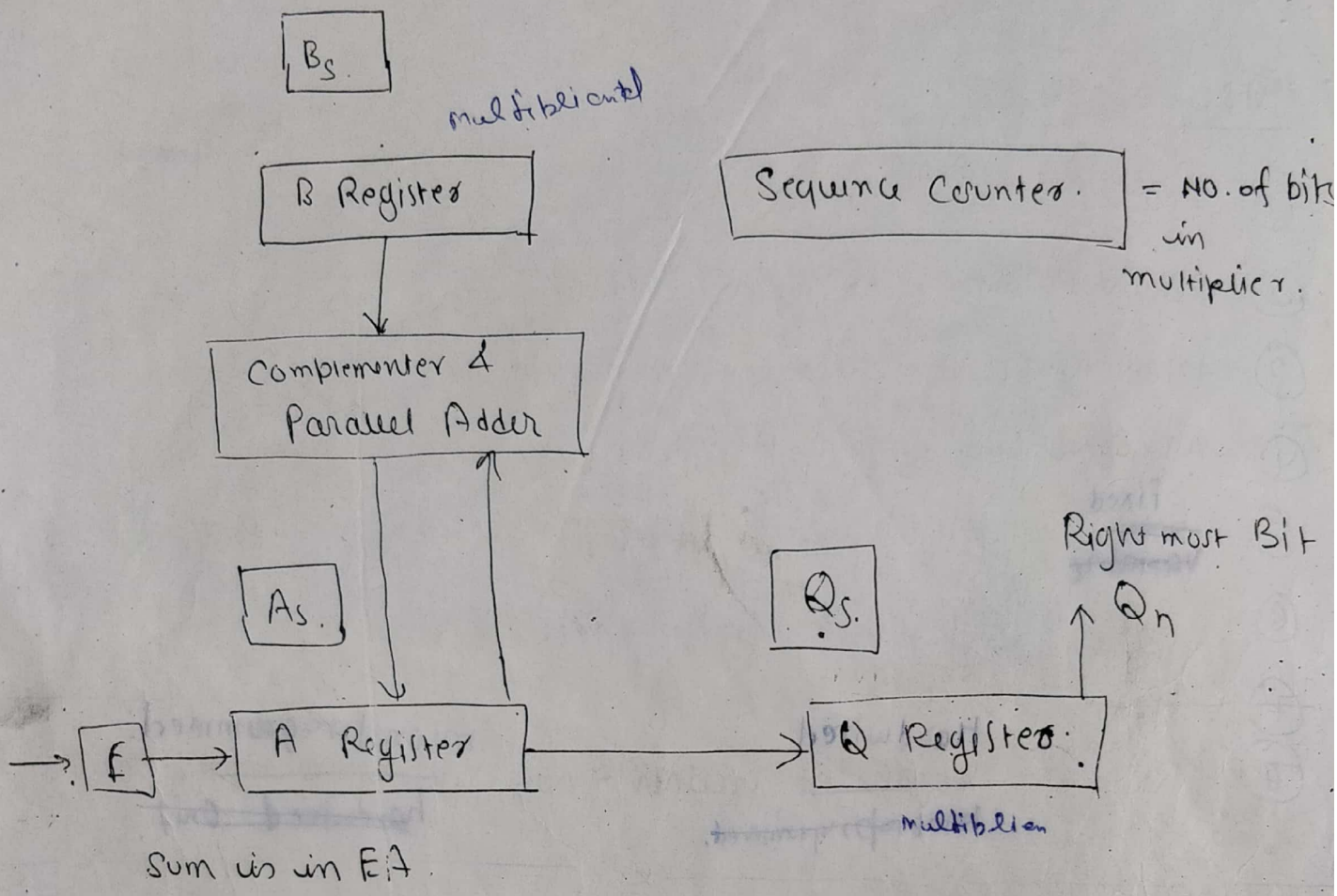
23 → 10111 → Multiplicand

19 → × 10011 → Multiplier

```
  10111
  10011 x
 00000xx
000000xxx
10111xxxx
```

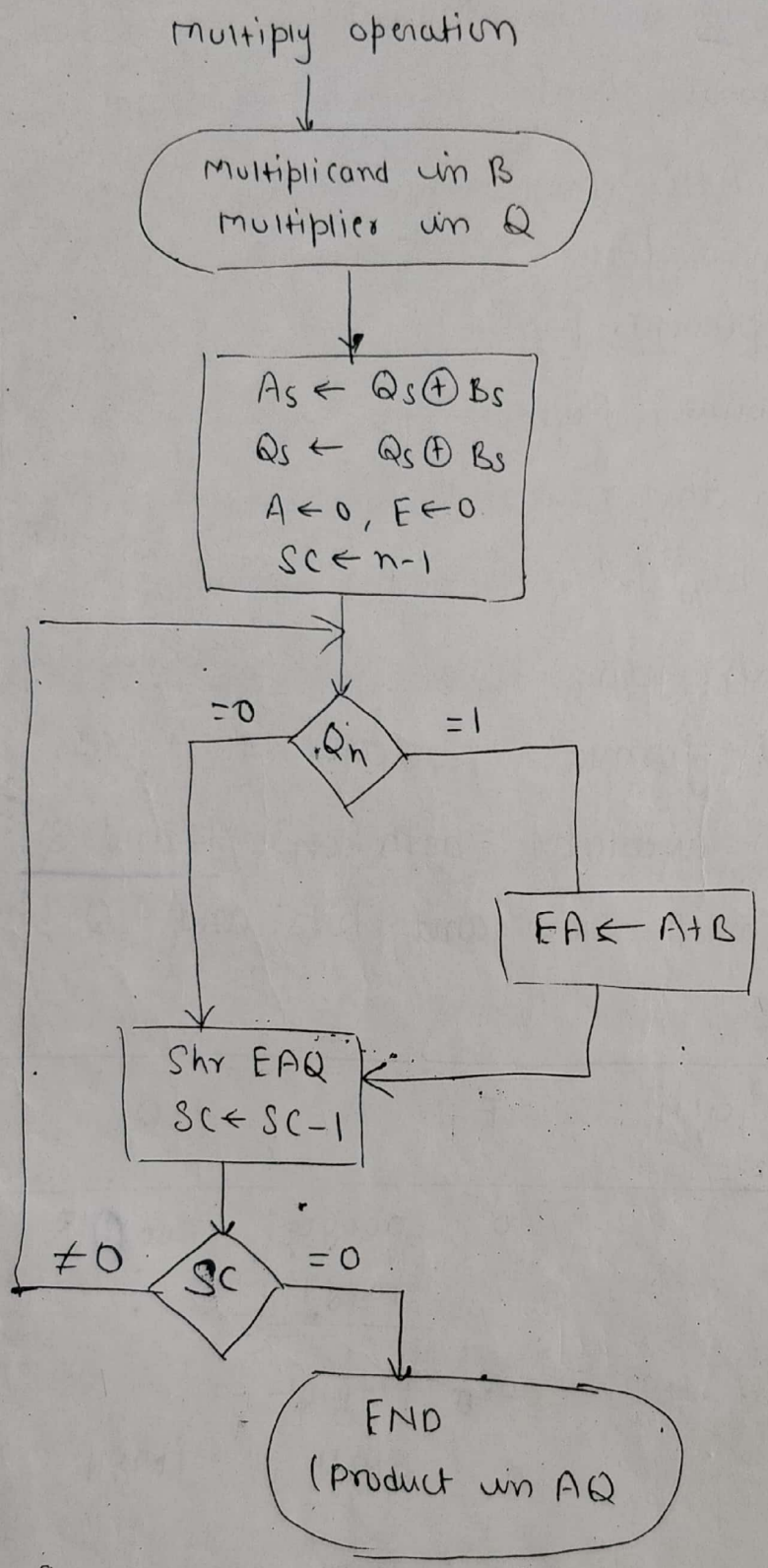
437 → 110110101 → Result.

Hardware Implementation for signed-magnitude Data.



[Hardware for multiply operation]

Hardware for multiplication consists of various registers, these registers together with registers A and B as shown in figure. The multiplier is shown in Q register and its sign in Qs. Sequence counter is initially set to a number equal to number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When content of counter reaches to zero, product is formed and procen stops.



[Flow chart for multiply algorithm].

Initially the multiplicand is in B and multiplier in Q. Their corresponding signs are stored in Bs and Qs respectively. The signs are compared and both A and Q are set to corresponding to sign of 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 number of bits of multiplier.

After the initialization, low-order bit of multiplier in Qn is tested. If it is a 1, multiplicand in B is added to present partitial product in A. If it is '0', nothing is done. Register EAQ is then shifted once to right to form the new partial product. Sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, procin is repeated and new partial product formed. Process stops when SC=0. The final product is available both in A and Q, with A holding the most significant bits and Q holds. least significant bits.

Multiplicand B = 10111

	E	A	Q	SC
Multiplier in Q	0	00000	10011	101 (5)
① Qn=1; add B		10111		
First partial Product	0	10111		
Shift Right EAQ	0	01011	11001	100 (4)
Qn=1, add B	0	10111		
Second partial Product	1	00010	11001	
Shift Right EAQ	0	10001	01100	011 (3)
Qn=0, shift Right EAQ	0	01000	10110	010 (2)
Qn=0, shift Right EAQ.	0	00100	01011	001 (1)

	E	A	Q
(5) Qn=1; add B		00100	
		10111	
Fift Partial Product	0	11011	
Shift Right EAQ	0	01101	10101
			000

Final Product in AQ= 0110110101

Qn= Stores the lower-order bit of the multiplier.

A = Register

E = carry holder register.

Booth Multiplication Algorithm

Booth algorithm works for both positive or negative multipliers in 2's complement representation. For example, a multiplier equal to -14 is represented in 2's complement as 110010 and is treated as $-2^4 + 2^2 - 2^1 = -14$.

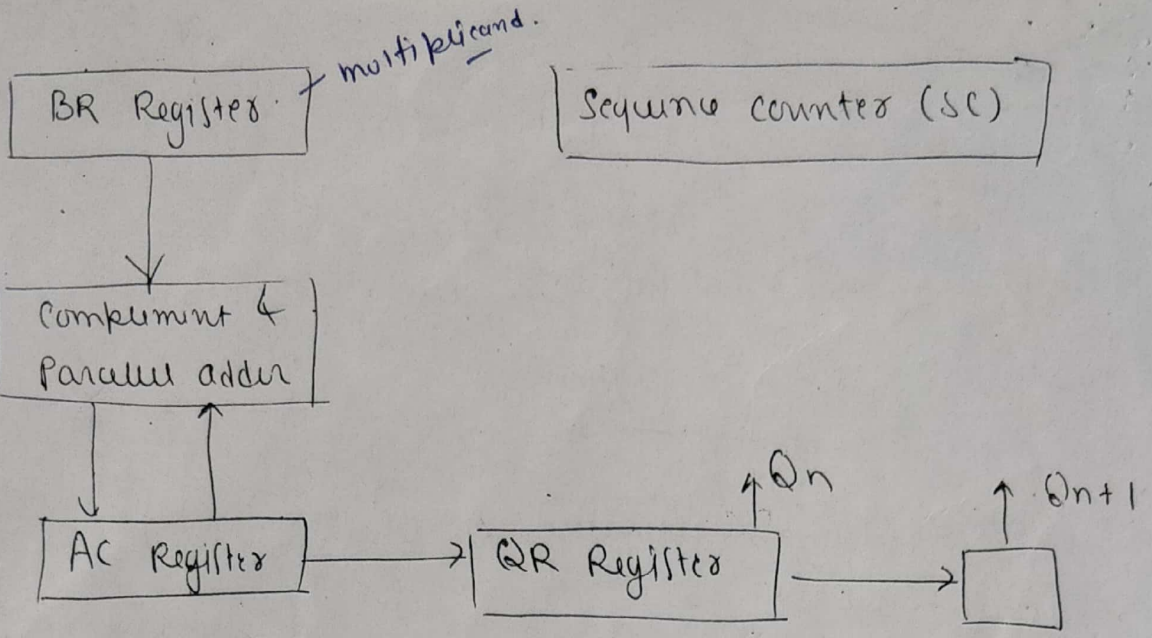
Binary Number of 14 = 01110 (+14).

1's Complement = 10001

2's " = 110010 (+14) 2's complement.

Hardware implementation of Booth algorithm require following register configuration.

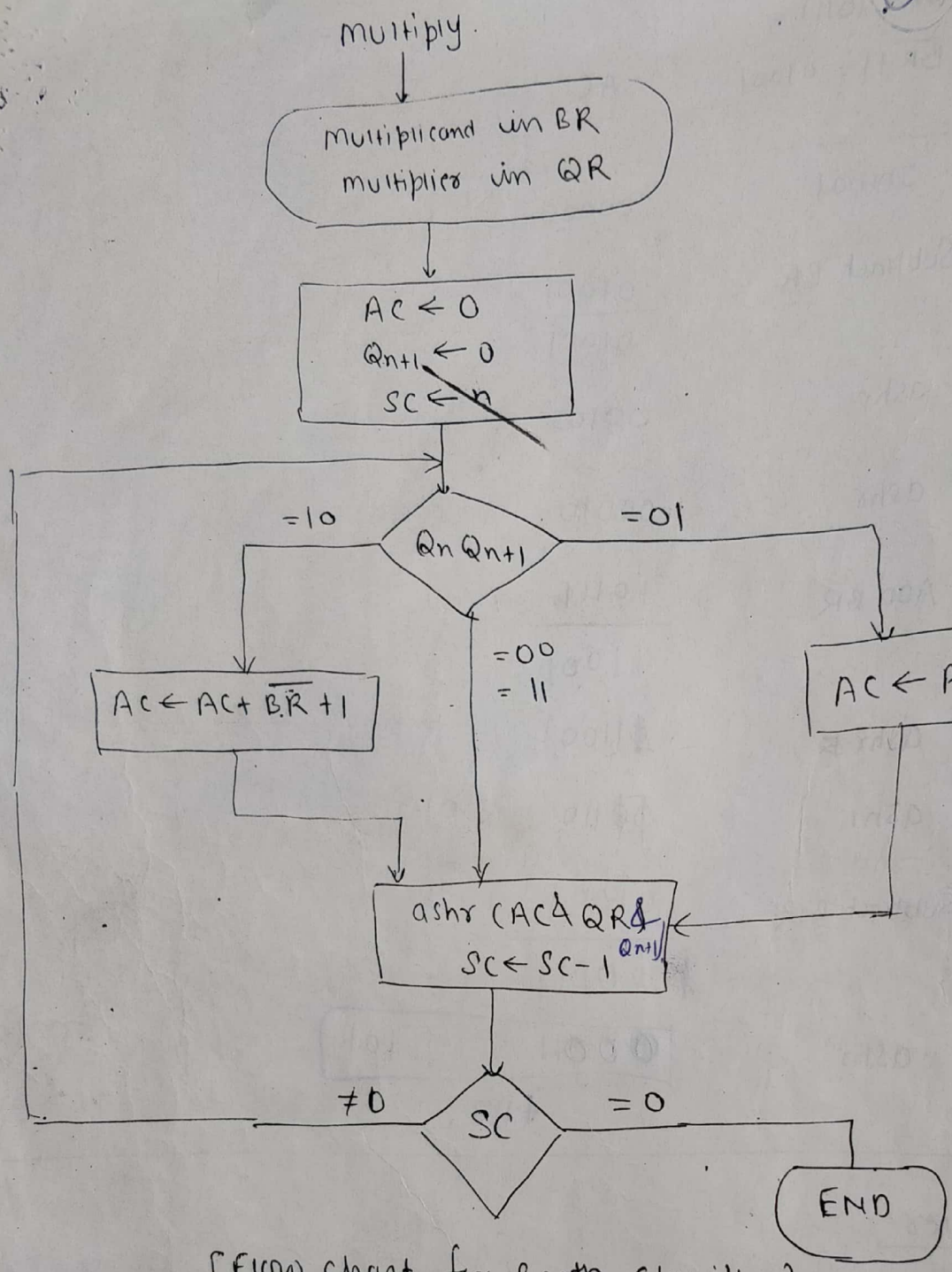
Hardware requirement is same as with previous example except here we rename registers. A, B and Q, as AC, BR and QR. Qn designates the least significant bit of multiplier. An extra flip flop Qn+1 is appended to QR.



[Hardware for Booth algorithm]

Flowchart for Booth algorithm:

AC and appended bit Q_{n+1} are initially cleared to 0 and sequence counter SC is set to number equal to bits of the multiplier. The two bits of the multipliers in Q_n and Q_{n+1} are tested, if two bits are equal to 10, it means that the first '1' in a string of 1's has been encountered. This require subtraction of multiplicand from partial product in AC. If two bits are equal to 01, it means that first '0' in string of 0's has been encountered. This requires addition of multiplicand to partial product of AC. When two bits are equal, partial product does not change. Next step is to shift Right partial product and multiplier. Sequence counter is decremented and computational loop repeated n-times.



[Flow chart for Booth algorithm]

e.g. $(-9) \times (-13)$

Example of Booth algorithm with the help of

Multiplicand = 10111 (-9) and

Multiplier = 10011 (-13) is shown in table as below :→

+9 = 01001	+13 = 01101
(-9) = 10110 1's complement	(-13) = 10010 1's complement of T. O.
= 10111 2's complement	= 10011 2's complement

$(-9 \times -13) = +117$

BR = 10111

$\overline{BR}+1$ = 01001

Q_n Q_{n+1}		AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101 (5)
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100 (4)
1 1	ashr	00010	01100	1	011 (3)
0 1	ADD BR	10111			
		11001			
	ashr	11100	10110	0	010 (2)
	ashr	11110	01011	0	001 (1)
0 0	ashr	11110			
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000 (0)

+117.

Array multiplier:

Multiplication of two binary numbers can be done with one micro operation by means of combinational circuit that forms the product of bits all at once. This is the fast way of multiplying two numbers since all it takes is the time for the signals to propagate through gates that form the multiplication array. However, an array multiplier requires a large number of gates.

MODULE 3

Booth Algorithm

(1) || (1)

(a) $(+15) \times (+13) = +195 = (0\,011\,000011)_2$

BR = 0 1111 (+15);
$\overline{BR}+1$ = 1 0001 (−15);
QR = 0 1101 (+13)

15 × 13

	Qn	Qn+1	AC	QR	Qn+1	SC
		Initial	00000	01101	0	101
①	1	0	Sub BR	10001		
				10001		
		ashr	11000	10110	1	100
②	0	1	~~Sub BR~~ Add BR	00111		
				00111	10110	1
		ashr.	00011	11011	0	011
				10001		
③	1	0	Sub BR	10100		
				10100	11011	0
		ashr.	11010	01101	1	001
④	1	1	ashr.	11101	00110	1
⑤	0	1	Add BR	01111		
				01100	00110	1
		ashr.	00110	00011	0	000

0 - 01100 00110 1
00110 00011 0

(b) $(+15) \times (-13) = -195$.

$195 = (0\,11000011)_2$.

1's comp of 195 = 10011100

2's " " 195 = 10011101

$-195 =$ ~~(10100~~ (1 10011101) 2's complement

BR = 0 1111 (+15)

$\overline{BR}+1 = (10000+1) = (10001)$ (-15).

QR = (-13) => (+13) = 01101.

= (-13) = 10010

= (10011) (-13)

= (10011) 2's complement

BR = −01111
BR+1 = 10001

11(2)

Qn	Qn+1	AC	QR	Qn+1	SC
		00000	10011	0	101 (5)
1	0	Sub BR	10001		
		10001	10011	0	
		ashr 11000	11001	1	100 (4)
1	1	ashr 11100	01100	1	011 (3)
0	1	Add BR	01111		
		01011	01100	1	
		ashr 00101	10110	0	010 (2)
0	0	~~Sub BR~~	10001		
		ashr 11001	01011	0	001 (1)
1	0	Sub BR	10001		
		10011	11011	0	
		ashr 11001	11101	1	000 (0)

→ In 2's complement form

−195 ← Ans.

How to check → 11001 11101

↳ 1's complement → 00110 00010

2's " → 00110 00011

+195. ← 00110 00011

Q $(-9) \times (-13) = +117$

BR = (-9) = 01001 = 10110+1

= (10111) 2's complement.

$\overline{BR}+1$ = 01001,

QR = 01101 = 10010+1 = (10011) 2's of (-13).

Ans = (+117) = 01110101

$\therefore (-9) \times (-13) = +117$ (2) 11(3)

Qn	Qn+1		AC	QR	Qn+1	SC
		initial	00000	10011	0	101
			01001			
1	0	SubBR	01001	10011	0	
		ashr	00100	11001	1	100
1	1	ashr	00010	01100	1	011
			10111			
0	1	addBR	11001	01100	1	
		ashr	11100	10110	0	010
0	0	ashr	11110	01011	0	001
			01001			
1	0	SubBR	00111	01011	0	
		ashr	00011	10101	1	000

(AC & QR)

Result.

1+4+16+32+64

64
32
16
5

117.

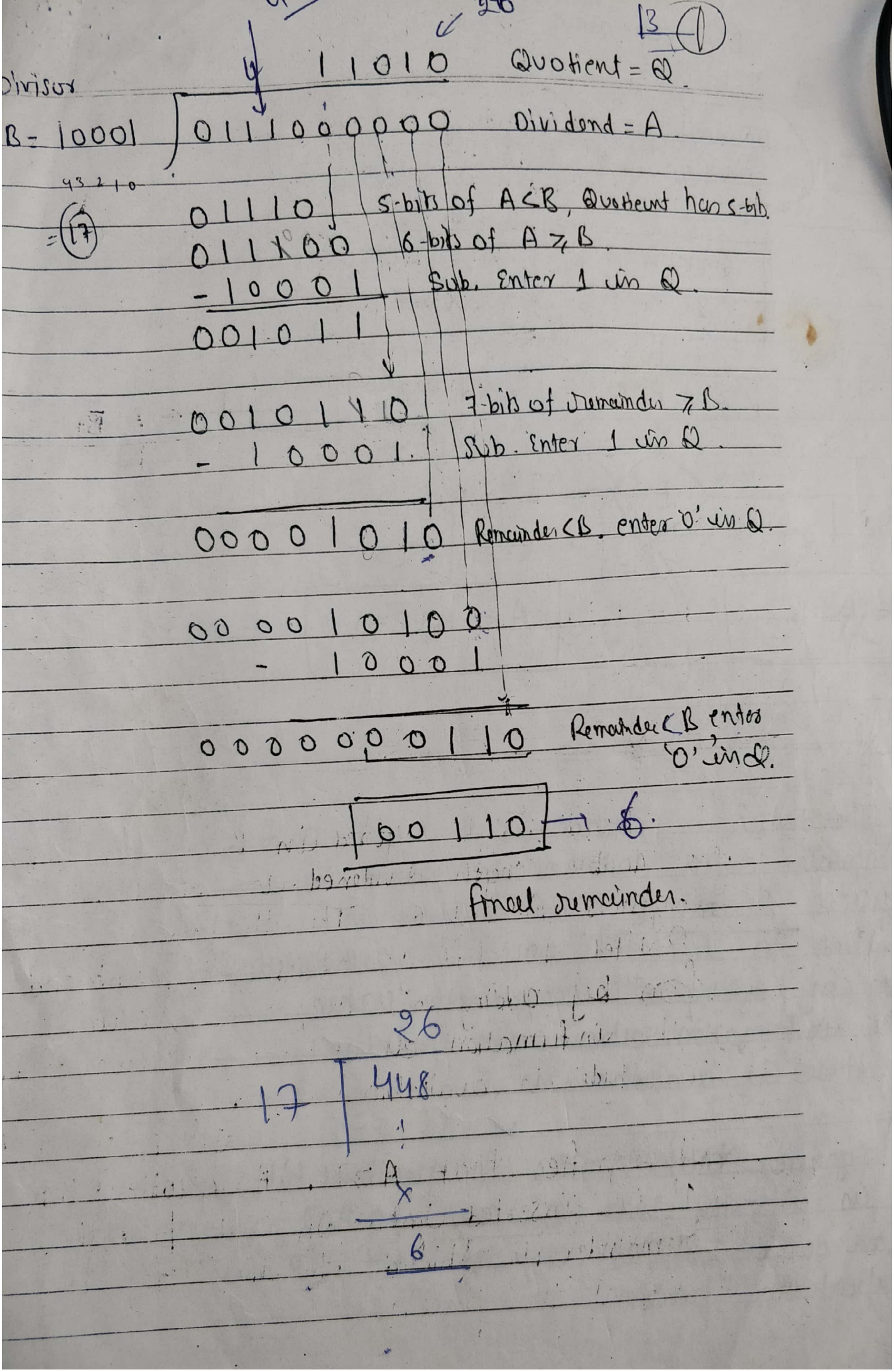
Qn 7 6 5 4 3 2 1 0
1 1 0 0 1 1 1 0

* Booth algorithm requires examination of the multiplier bits and shifting of the partial product. Prior to shifting, the multiplicand may be added to partial product, subtracted from the partial product or left unchanged according to following rules:-

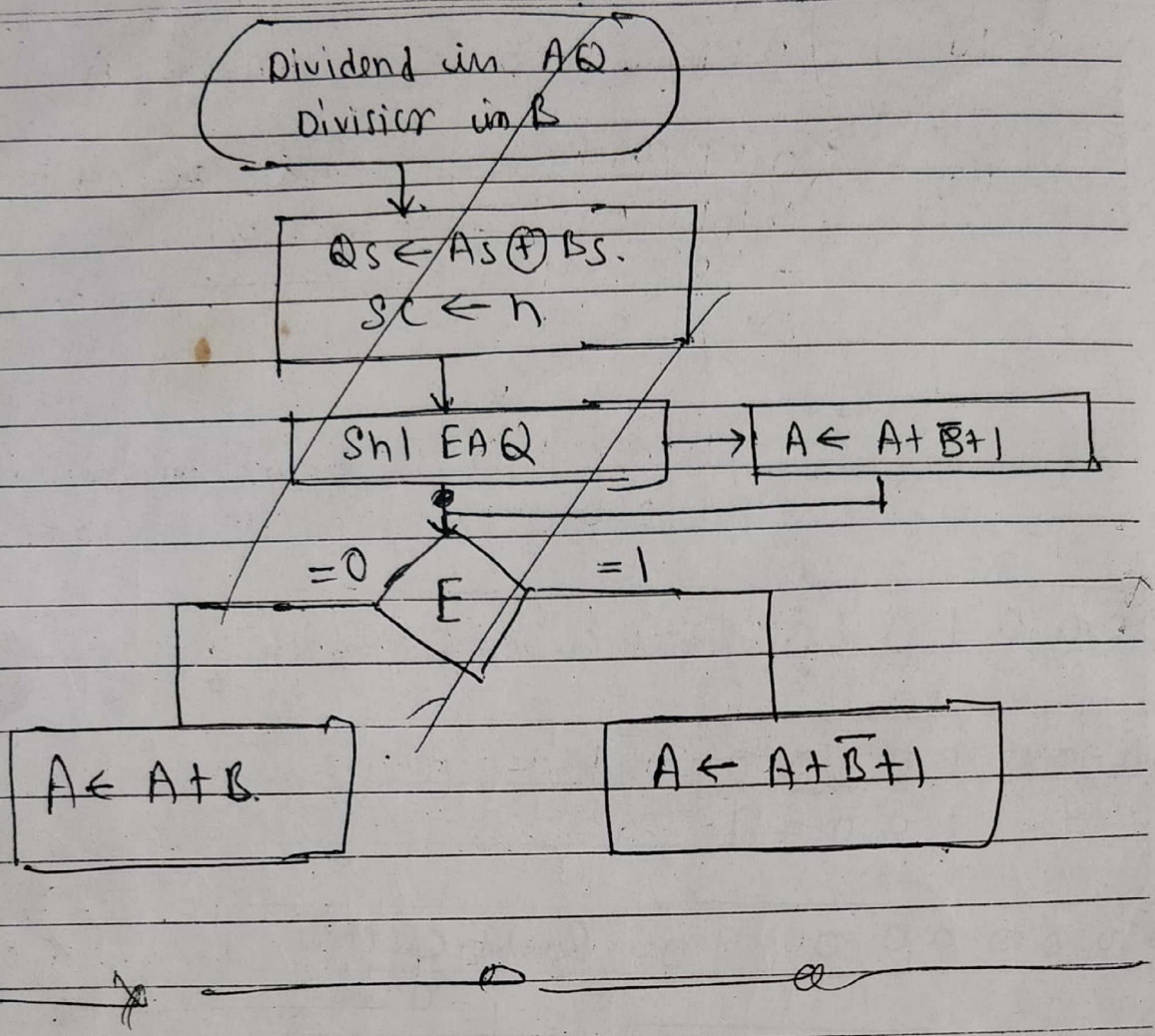
1) multiplicand is subtracted from partial product upon encountering the first (from least significant) 1 in string of 1's in multiplier.

2) multiplicand is added to partial product upon encountering the first '0' in string of 0's in multiplier.

3) Partial product does not change when the multiplier bits are identical to previous multiplier bit.



Divide operation



The divisor is stored in B register and the double-length dividened is stored in registers A and Q. The dividend is shifted to left and divisor is subtracted by adding 2's complement value. The information about the relative magnitude is available in E.

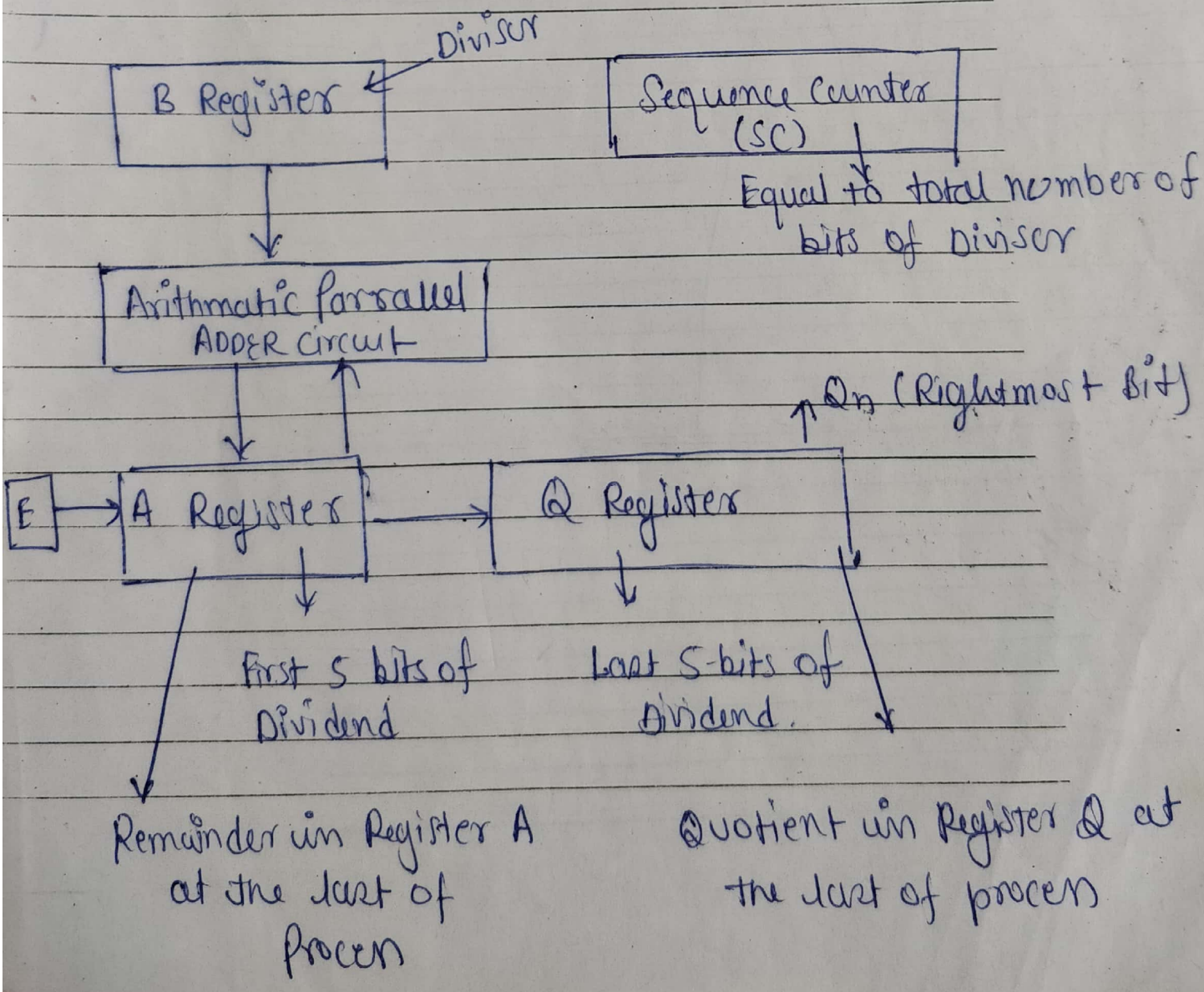
If E=1.

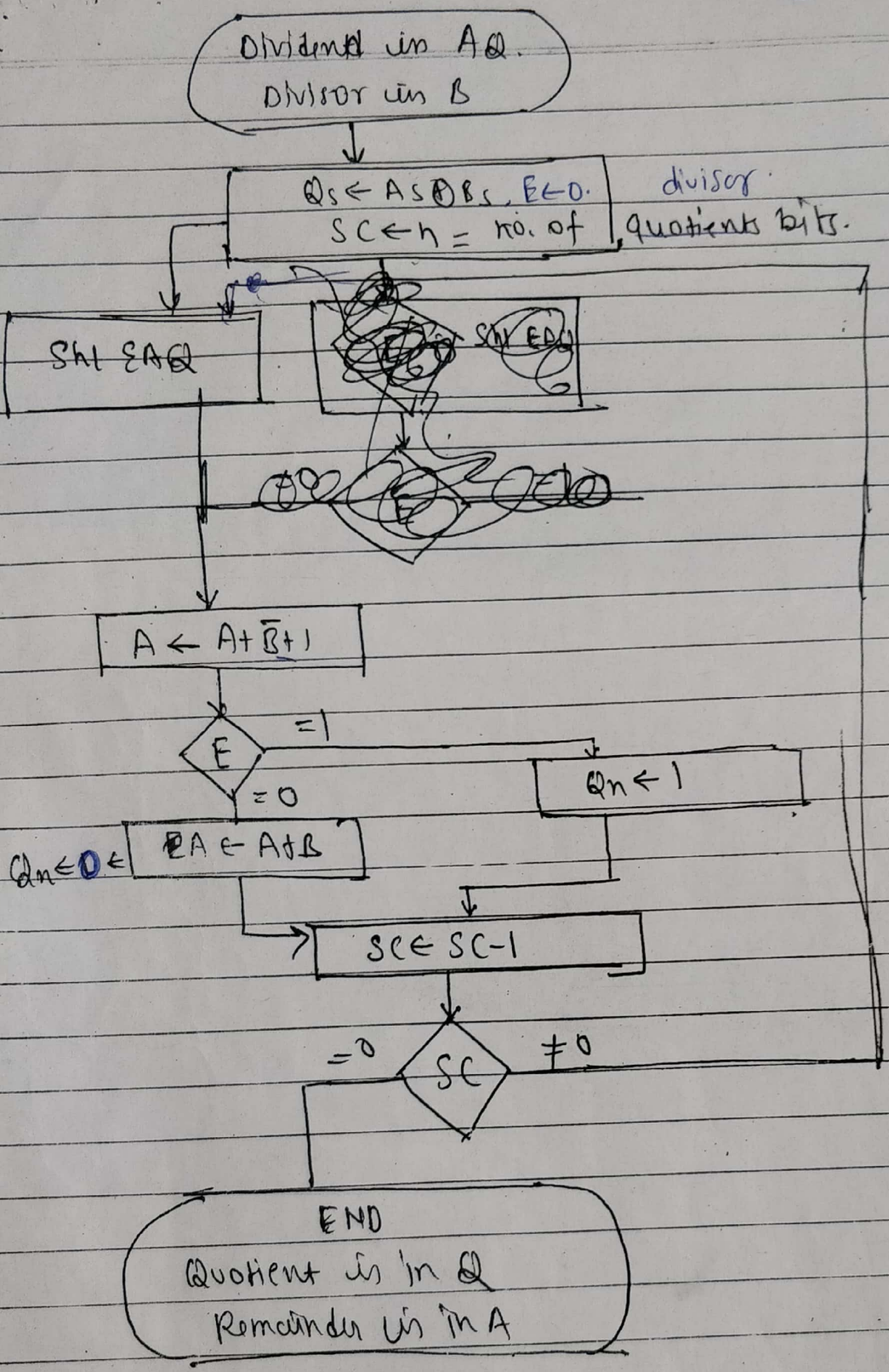
it signifies that A ≥ B. A quotient bit '1' is inserted into Qn and partial remainder is shifted to the left to repeat the procen.

≤ If E=0, it signifies that A<B so the quotient in Q_n remains a '0'. 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 & the process is repeated again until all five quotient bits are formed.

The final remainder stored in register A and the quotient in register Q.

Hardware Configuration →





Division Algo. flow diagram

divide-overflow flip flop DVF is set and operation is terminated. If A < B, no divide overflow occurs, so value of 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 higher-order bit shifted into E. If bit shifted into E is 1, then 'B' must be subtracted from A and 1 inserted into Qn for quotient bit.

If the shift-left operation inserts a '0' into E, the divisor is subtracted by adding its 2's complement value and carry is transferred into E. If E=1, it signifies that A ≥ B, therefore Qn is set to 1. If E=0, it signifies that A < B and original number is restored by adding B to A. And value of sequence counter is decremented by 1. This process is repeated again with register A holding partial remainder until value of sequence counter (SC) is not equal to '0'.

Explanation of division algorithm with help of example.

Explanation of division algorithm is shown as;- →

P.T.O.

Divisor B = 10001 $\quad\quad$ $\bar{B}+1 = 01111$

	E	A	Q	SC
Dividend		01110	00000	5 (101)
Shl EAQ	0	11100	00000	
add $\bar{B}+1$		01111		
	1	01011		
E=1				
Set Qn←1	1	01011	00001	4 (100)
Shl EAQ	0	10110	00010	
Add $\bar{B}+1$		01111		
	1	00101		
E=1				
Set Qn←1	1	00101	00011	3 (011)
Shl EAQ	0	01010	00110	
Add $\bar{B}+1$		01111		
	0	01001	00110	2 (010)
E=0, Qn←0				
ADD B		10001		
	1	01010	00110	
Restore Remainder				
Shl EAQ	0	10100	01100	
Add $\bar{B}+1$		01111		
	0	00011	01100	1 (001)
E=1,				
Set Qn←1	1	00011	01101	
Shl EAQ	0	00110	11010	0
Add $\bar{B}+1$		01111		
	0	10101	11010	Continue...

0 10101 11010

E=0, Set Qn←0

Add B.

10001

01110 11010

Remainder. Quotient

IS(1)

000(0).

An array multiplier can be implemented, with a combinational circuit, Consider multiplication of following 2-bit numbers:-

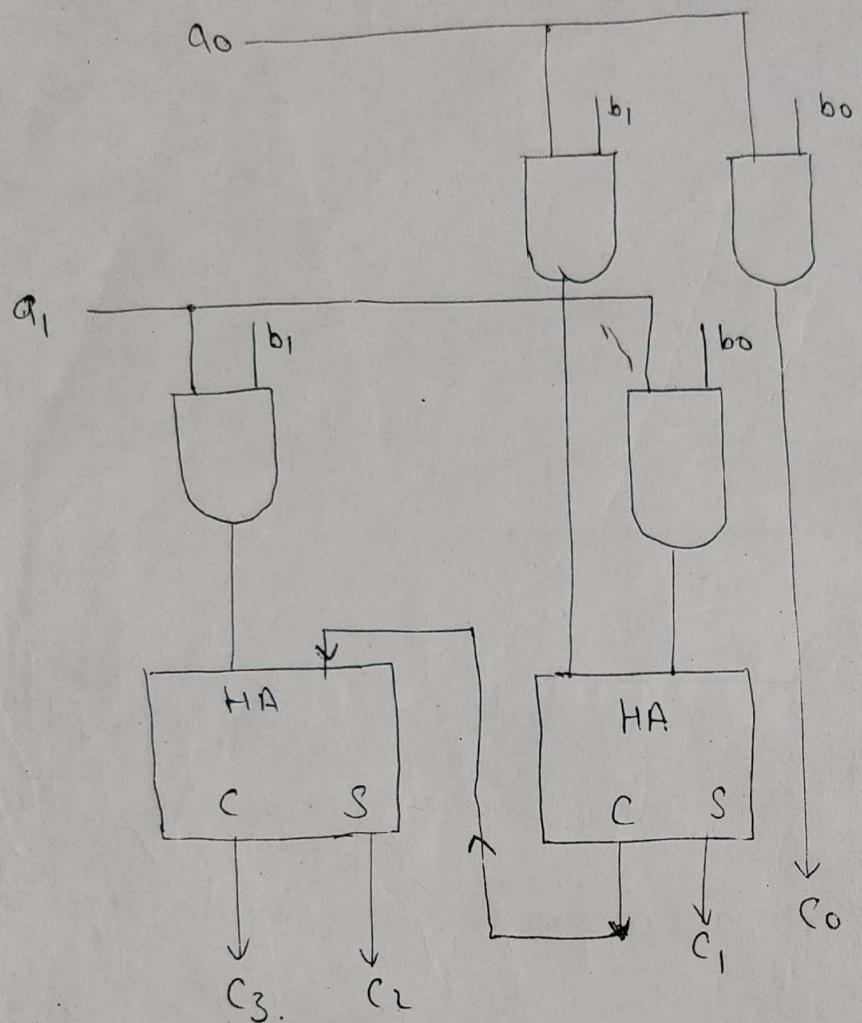
$$\begin{array}{cccc} & & b_1 & b_0 \\ & & a_1 & a_0 \\ \hline & & a_0b_1 & a_0b_0 \\ & a_1b_1 & a_1b_0 & \\ \hline c_3 & c_2 & c_1 & c_0 \\ \hline \end{array}$$

The first partial product is formed by multiplying a_0 by b_0b_1. Multiplication of two bits produces a 1, if both bits are 1, otherwise it produces '0'. This is identical to AND operation and can be implemented with AND Gate.

Second partial product is formed by multiplying a_i by b_ib_0 and is shifted one position to the left. These two partial products are added with the two-half adder circuits and accordingly sum and carry will be generated to find out the required product bits.

And 2-bit by 2-bit array multiplier is shown as below:→

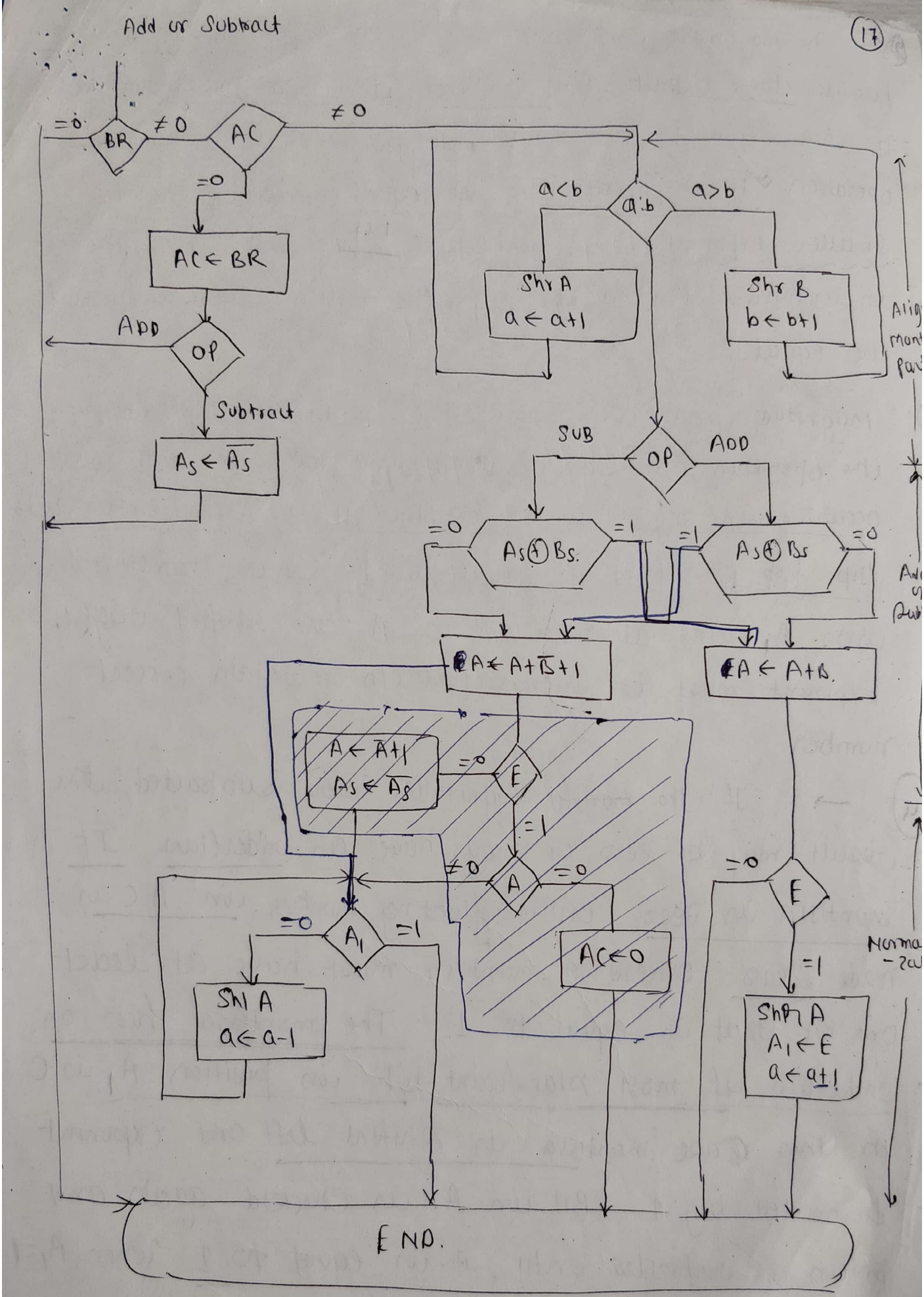
P.T.O.



[2-bit by 2-bit array multiplier]

Division Algorithm :

The division procedure is illustrated in the following example. The divisor B consists of 5-bits and the dividend A, of ten bits. The 5-most significant bit of dividened is compare with the divisor. Since 5-bit number is smaller than B, we will try by taking the sixth most significant bit of A and compare this number with B 6-bit number is greater than B so we palace a '1' for Quotient bit



(Addition OR Subtraction of floating point numbers.)

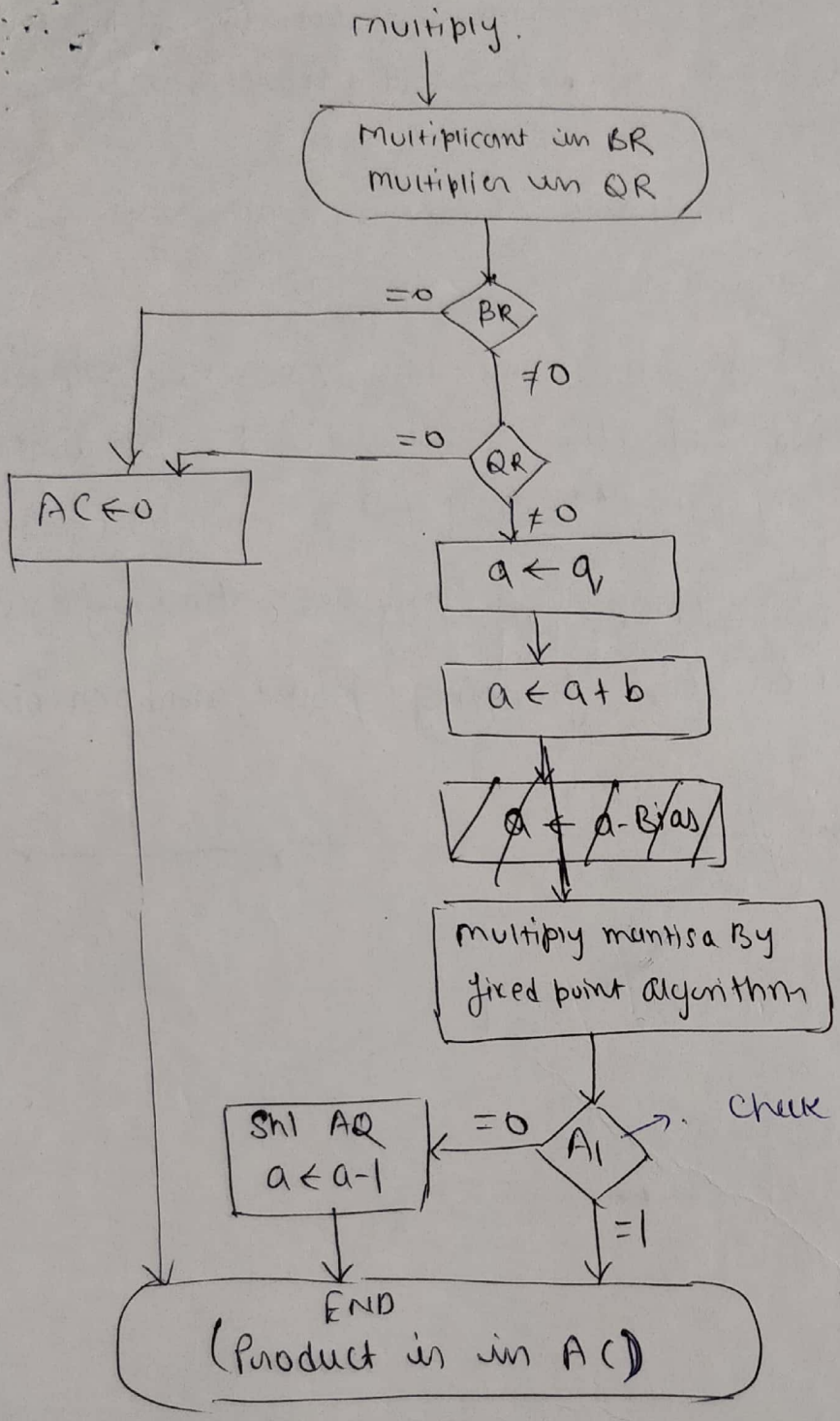
A_1 = most Significant of mantissa,

(2) The magnitude comparator attached to exponents a and b provide three outputs that indicate their relative magnitude. If two exponents are equal, we perform the arithmatic operation. If exponents are not equal, mantissa having smaller exponent is shifted to ~~right~~ left and its exponent incremented. Procan is repeated until two exponents are equal.

(3) Magnitude part is now added or subtracted depending upon the operation and sign of two mantissas. If an overflow occurs when magnitudes are added, it is transferred into flip flop E. If E is equal to 1, bit is transferred into A_1 and all other bits of A are shifted right. Exponent must be incremented to maintain correct number.

(4) → If the ~~manifu~~ magnitudes were subtracted, the result may be zero or may have an underflow. If mantissa is zero, entire floating number in AC is made zero. Otherwise mantissa must have at least one bit that is equal to 1. The mantissa has an Underflow if most significant bit in position A_1 is 0. In this case mantissa is shifted left and exponent decremented by 1. Bit in A_1 is checked again and procan is repeated until it is equal to 1. When A mantissa is normalized and operation is complete.

Multiplication of floating point numbers →



A = mantissa part

[Multiplication of floating-point number]

① The Two operands are checked to determine if they contain a Zero. If either operand is equal to zero, product in AC is set to 0 and operation is termina If neither of operands is equal to zero, procen continue with exponent addition.

The exponent of multiplier is in 'q' and adder is between exponents a and b. It is necessary to transfer exponents from q to a, add two exponents and transfer sum into 'a'.

② Multiplication of mantissa is done as same with fixed point multiplication algorithm.

③ The product may have an underflow, so most significant bit in A is checked. If it is equal to 1, product is already normalized. If it is equal to '0', mantisa in A is shifted left and exponent decremented. And finally multiplication operation for floating point number is completed.