Carry look Ahead Adder (CLA Adder)

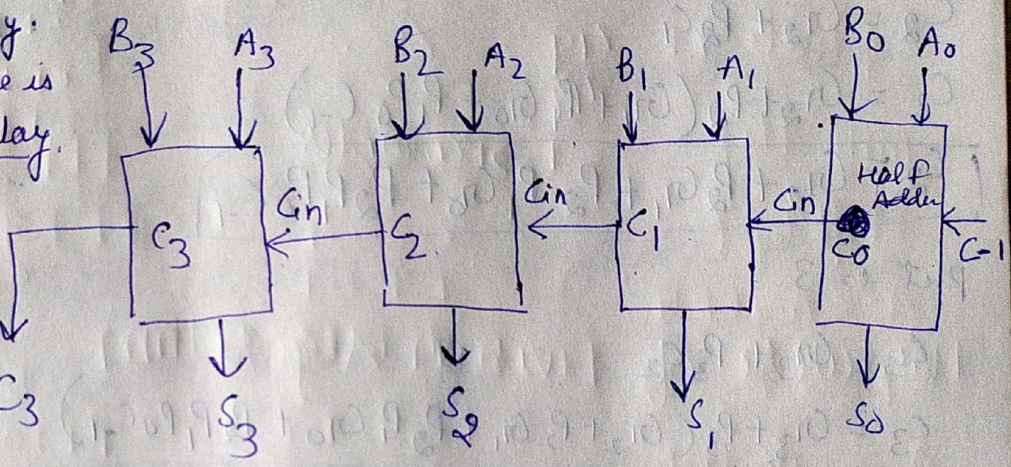
⇒ Predict the carry before it is actually produced.

⇒ It is superior than full adder in term of speed and speed is more important feature of any digital circuit.

⇒ In full adder when we add two number A and B and let suppose both A and B are 4-bit numbers. The sum will take some time because it deepend on carry and this time is called propagation delay.

⇒ The propagation delay of sum and carry inside adder is different.

⇒ But we are more concerned about the propegation delay of carry.



$C_i = G_i + P C_{i-1}$

A	B	Cin	C_0	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

A⊕B=1 & Cin=1 output=1

A·B

$C_0 = A \cdot B + (A \oplus B) \cdot C_{in}$

G_i → Carry generator

P → carry propagator

$C_0 = G_i + P C_{in}$

$C_i = G_i + P_i C_{i-1}$

put $i=0$

$$C_0 = G_0 + P_0 C_{-1} \quad \text{(I)}$$

put $i=1$

$$C_1 = G_1 + P_1 C_0 \quad \text{(II)}$$

$$C_1 = G_1 + P_1 (G_0 + P_0 C_{-1})$$

$$C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{-1}$$

put $i=2$

$$C_2 = G_2 + P_2 C_1$$

$$C_2 = G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_{-1})$$

$$C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{-1}$$

put $i=3$

$$C_3 = G_3 + P_3 C_2$$

$$C_3 = G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{-1})$$

$$C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{-1}$$

$G_1 = A_1 \cdot B_1$ $\quad G_0 = A_0 B_0$

$G_2 = A_2 \cdot B_2$

$G_3 = A_3 \cdot B_3$

$P_3 = A_3 \oplus B_3$

$P_2 = A_2 \oplus B_2$

$P_1 = A_1 \oplus B_1$

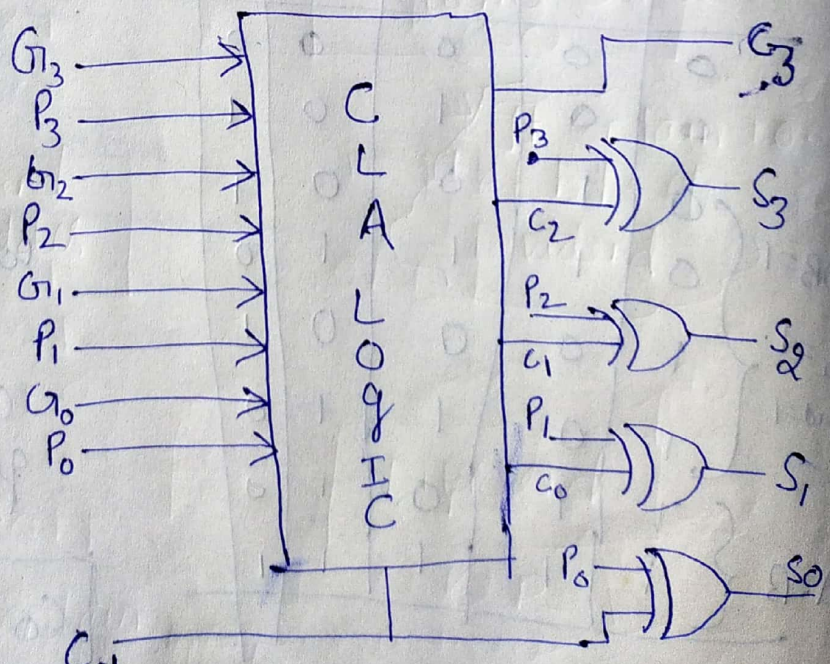
$P_0 = A_0 \oplus B_0$

$S_3 = A_3 \oplus B_3 \oplus C_2$

$S_2 = A_2 \oplus B_2 \oplus C_1$

$S_1 = A_1 \oplus B_1 \oplus C_0$

$S_0 = A_0 \oplus B_0 \oplus C_{-1}$

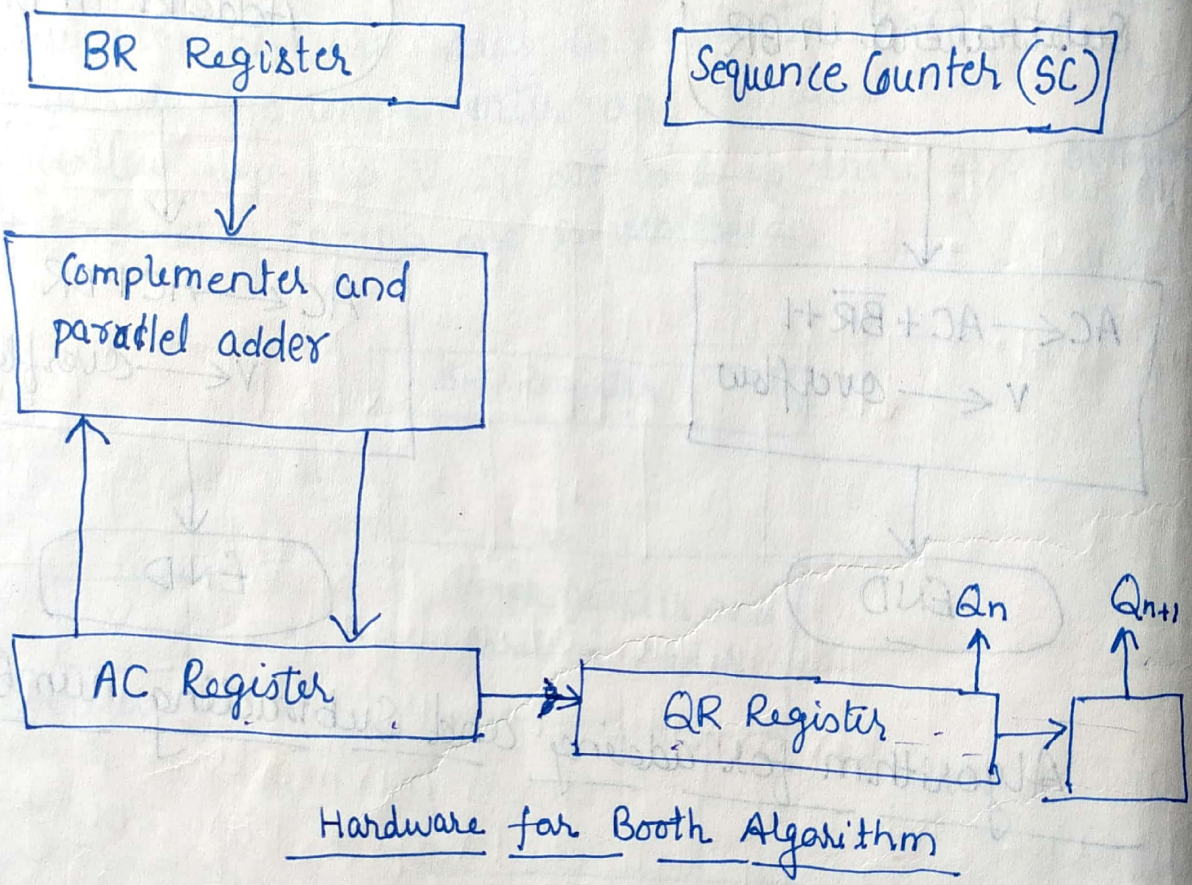


Booth Multiplication Algorithm

⇒ The hardware implementation of Booth algorithm requires the register configuration as shown below.

⇒ We use register AC, BR and QR. Qn designates the least significant bit of the multiplier in register QR.

⇒ An extra flip-flop Qn+1 is appended to QR for a double bit inspection of the multiplier.



Hardware for Booth Algorithm

Flowchart for Booth Algorithm

⇒ The flowchart for Booth algorithm is shown in fig next page. AC and appended bit Qn+1 are initially cleared 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.

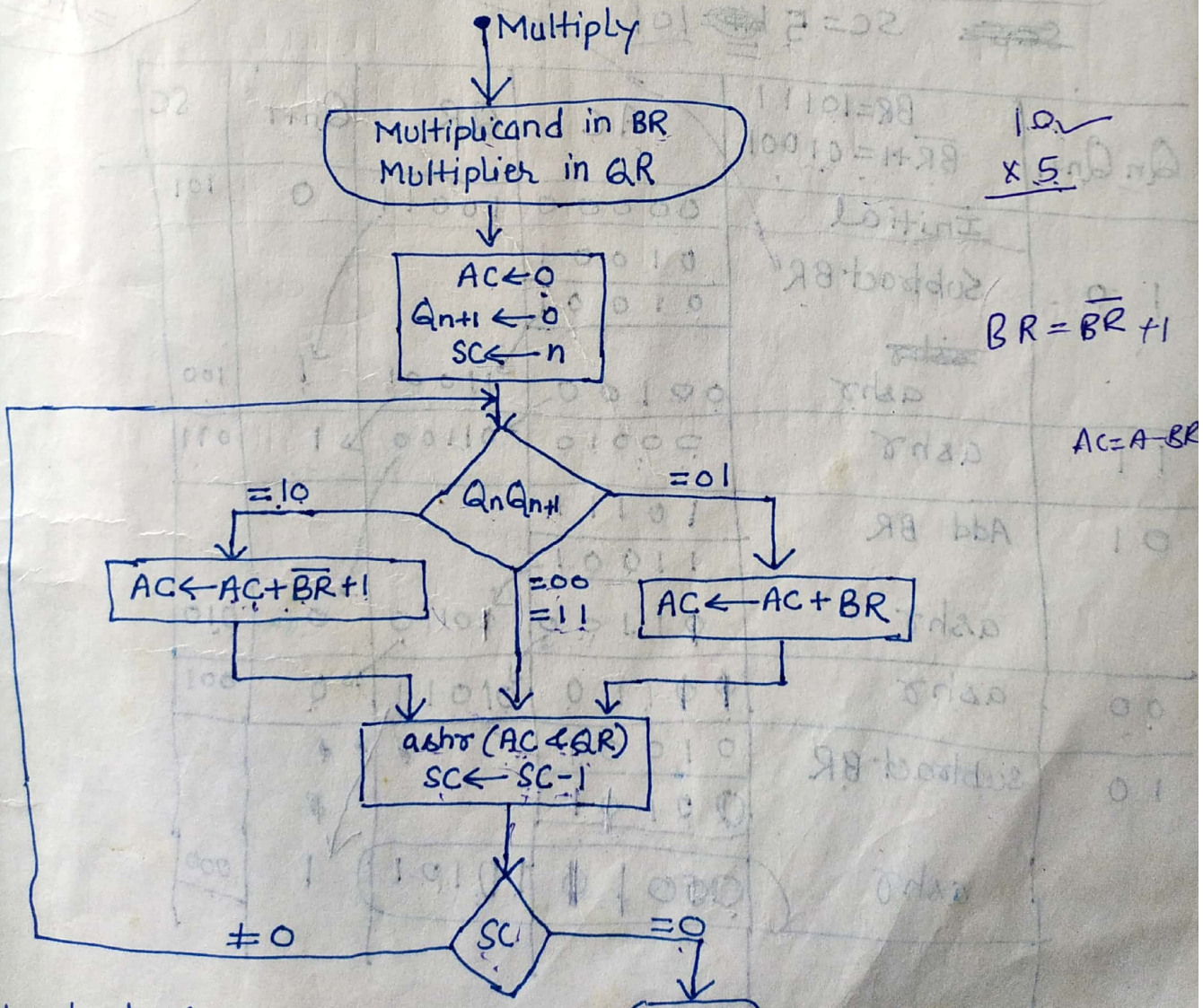
⇒ If the two bits are equal to 10, it means that the first 1 in a string of 1's has been encountered. This requires a subtraction of multiplicand from the partial product in AC.

⇒ If the two bits are equal to 01, it means that the first 0 in 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 i.e 00 or 11 the partial product does not change.

⇒ The next step is to shift right the partial product and multiplier (including bit Q_{n+1}). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC unchanged.

⇒ The next step is the sequence counter (SC) is decremented and the computational loop is repeated n times.



Flowchart of Booth algorithm

10
× 5

$BR = \overline{BR} + 1$

AC = A − BR

ex ⇒ Show the step-by-step multiplication process using Booth algorithm of (-9) × (-13).

-9
×-13

Soln ∴ BR = -9, QR = -13, n = 5 bit

∴ 9 = 01001

one's complement of 01001 is 10110

two's complement

```
 01110     10110
   +1        +1
-9 01111 = 10111
```

-9 = 10111

Now, BR = 10111

SC = 5 = 101

13 = 01101

one's complement of 01101 is 10010

two's complement of 10010 is

```
 10010
   +1
-13 = 10011
```

-13 = 10011

QR = 10011

$A \leftarrow A + \overline{BR} + 1$

$= A - BR$

BR = 10111

$\overline{BR} + 1 = 01001$

$Q_n Q_{n+1}$		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	00110	0	010
0 0	ashr	11110	00011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10001	1	000

-9 × -13 = +117 = 01110101 Ans

(2) Show the step-by-step multiplication process using Booth algorithm when the following binary numbers are multiplied. Assume 5-bit registers that hold signed numbers.

(a) (+15) X (+13) (b) (+15) X (-13)

Soln ⇒

(2) Multiply (+4) and (-7) using booth multiplication algorithm and show the step by step procedure of multiplication.

Soln Here Multiplicand = +4

BR = 4

Multiplier (QR) = -7

BR = +4 = +100 (in Binary)

= 0100

$\overline{BR}+1$ = 1011 (2's complement)

+1

1100 (-4)

QR = -7

+7 = +111

7 = 0111

2's complement of 7 i.e 0111

i.e ⇒ 0111

= 1000 (one's compl)

+1

-7 = 1001 = QR

Add 1 because last bit is 1

Qn Qn+1	BR = 0100 $\overline{BR}+1$ = 1100	AC	QR (Qn)	Q_{n+1}	SC
	initial	0000	1001	0	100
1 0	Subtract BR i.e ADD $\overline{BR}+1$	+1100 1100			
	ashr	1110	0100	1	011
0 1	Add BR	+0100 0010 (bit error)			
	ashr	0001	0010	0	010
0 0	ashr	0000	1001	0	001
1 0	Subtract BR i.e Add $\overline{BR}+1$	+1100 1100			
	ashr	1110	0100	1	000 → End

So the Multiplication of (+4)×(-7) = -28 = 11100100 Ans

NOTE ⇒ How to check -28 = 11100100 for that we do 2's complement of +28 if it would be +28 then given Ans will be -28. i.e +28 = 00011100 = 11100011

+1

11100100 correct

Array Multiplier

⇒ The multiplication of two binary numbers can be done with one microoperation by means of a combinational circuit that forms the product bits all at once. This fast way of multiplying two numbers is known as Array Multiplier

⇒ Consider the multiplication of two 2-bit numbers as shown in fig below. The multiplicand bits are b_1 and b_0, the multiplier bits are a_1 and a_0 and the product is $C_3 C_2 C_1 C_0$.

⇒ The first partial product is formed by multiplying a_0 by $b_1 b_0$.

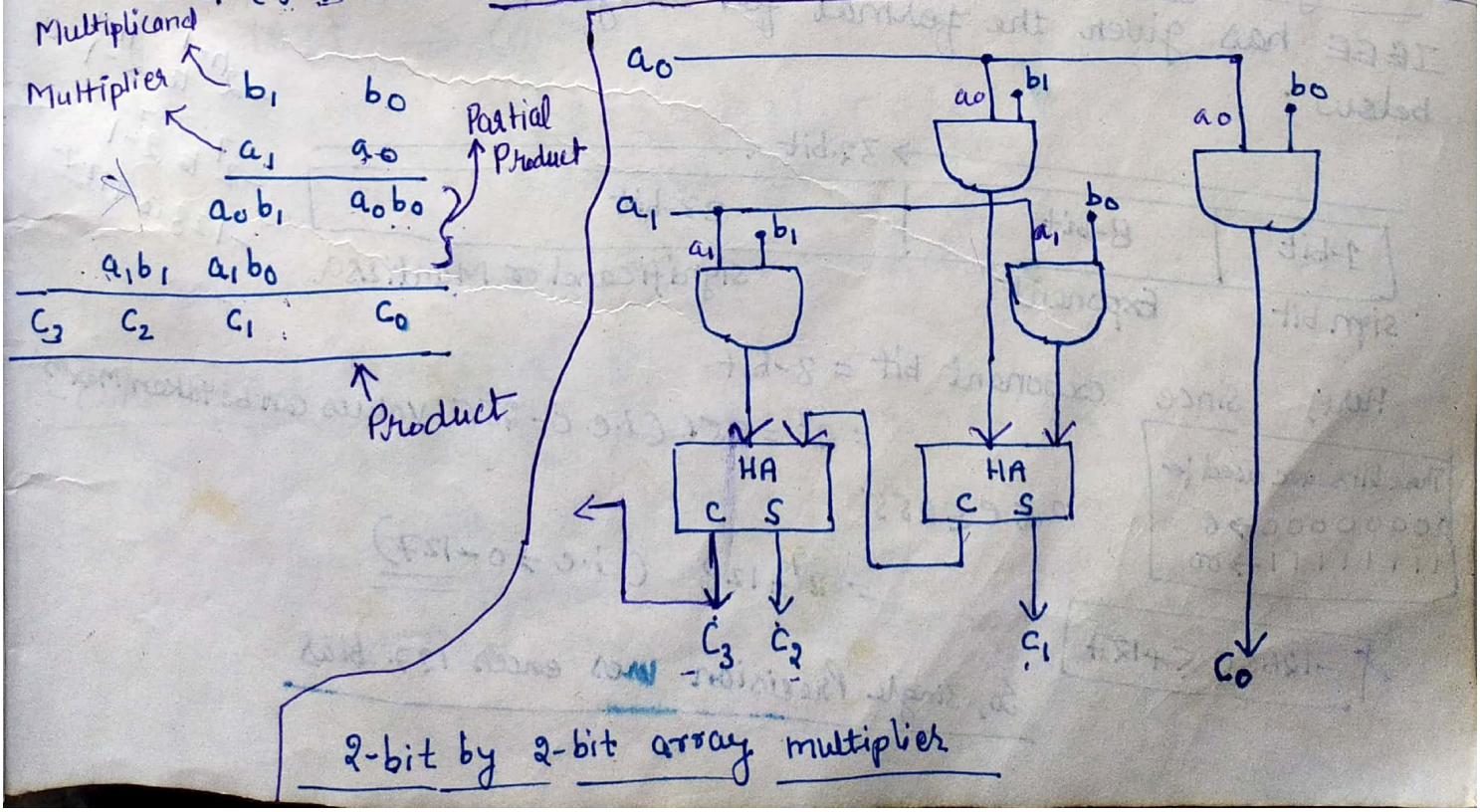
⇒ The multiplication of two bits such as a_0 and b_0 produces a 1 if both bits are 1. otherwise it produces a 0.

⇒ This is identical to an AND operation and can be implemented with an AND gate.

⇒ As shown in fig, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a_1 by $b_1 b_0$ and is shifted one position to the left.

⇒ The two partial product is added with two half-adder (HA).

NOTE → 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.



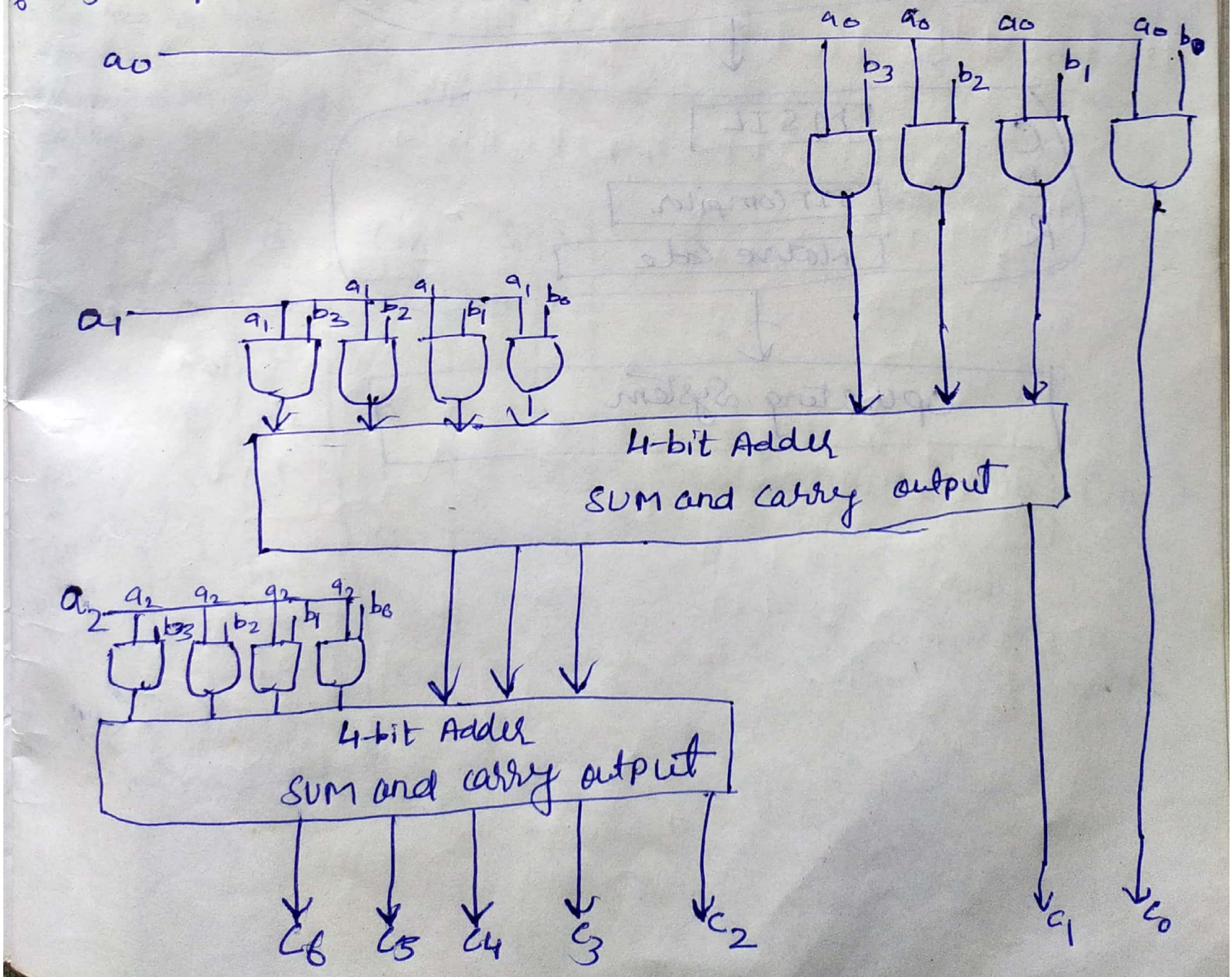
2-bit by 2-bit array multiplier

(2) Draw a 4-bit by 3-bit array multiplier.

Soln ⇒ Let 4-bit of multiplicand are b_0, b_1, b_2, b_3 and 3-bit of multiplier are a_0, a_1, a_2

So, when we multiply

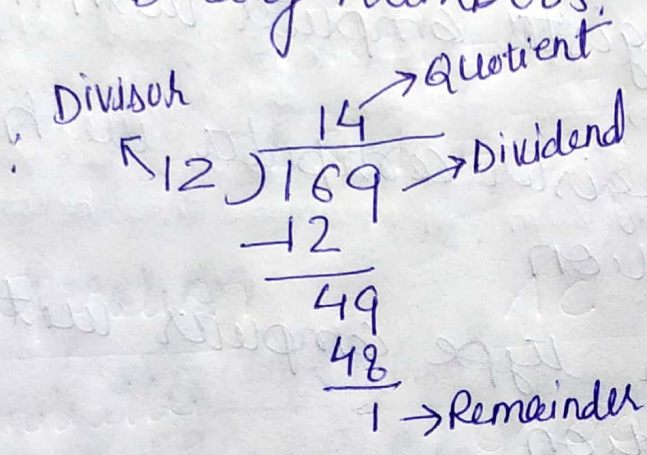
$$\begin{array}{r} b_3\ b_2\ b_1\ b_0 \\ \times\ a_2\ a_1\ a_0 \\ \hline a_0b_3\ \ a_0b_2\ \ a_0b_1\ \ a_0b_0 \\ a_1b_3\ \ a_1b_2\ \ a_1b_1\ \ a_1b_0 \\ a_2b_3\ \ a_2b_2\ \ a_2b_1\ \ a_2b_0 \\ \hline c_6\ c_5\ \ c_4\ \ c_3\ \ c_2\ \ c_1\ \ c_0 \end{array}$$

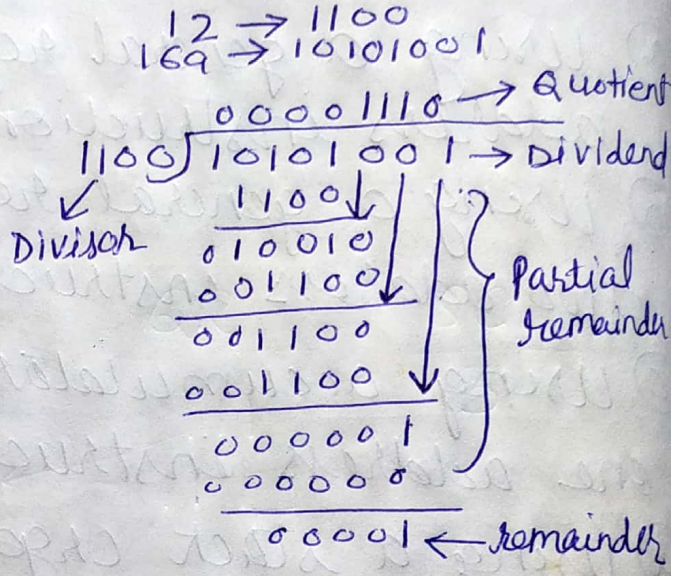


Division Algorithm

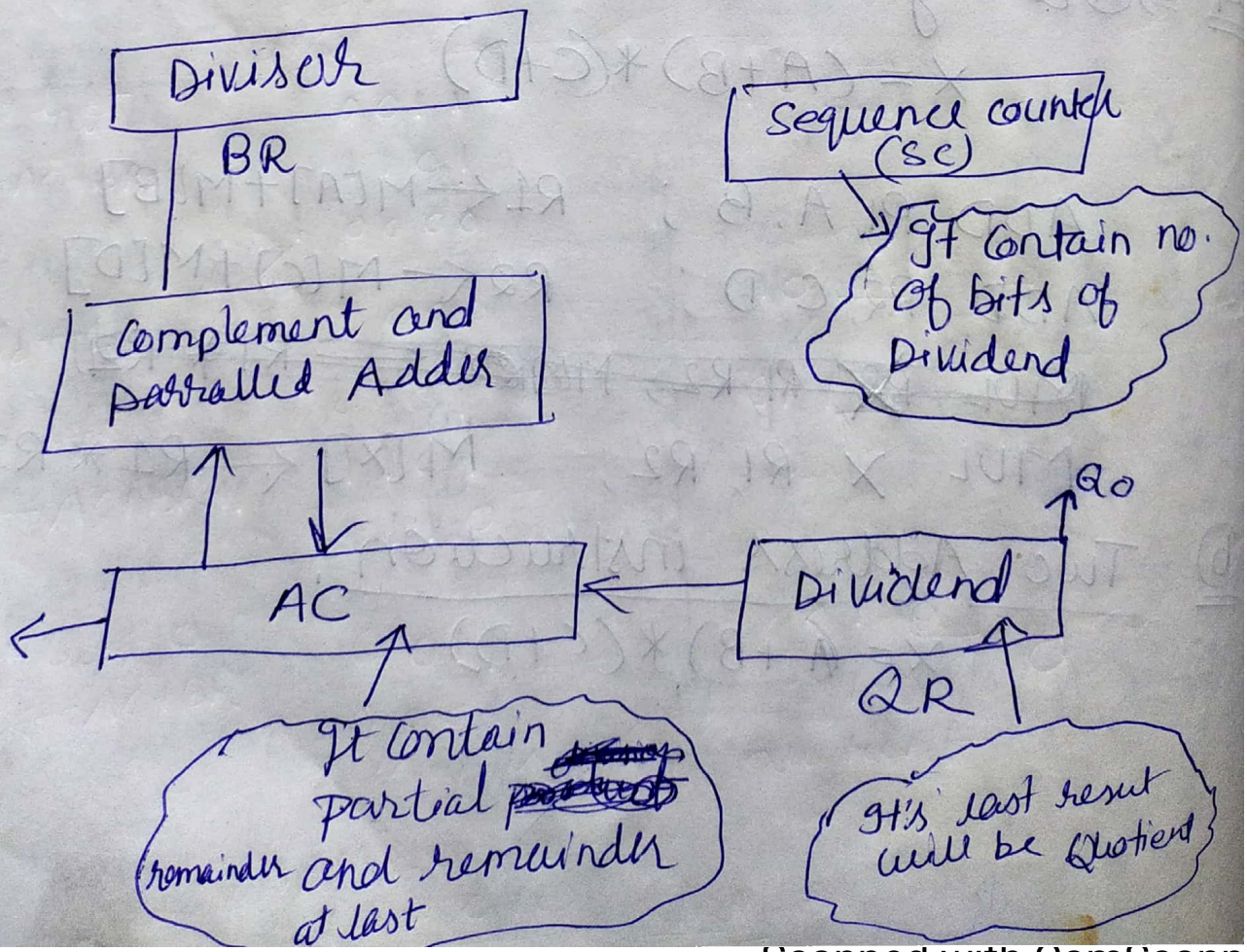
⇒ Division algorithm is used to divide the
⇒ two binary numbers.

eg:

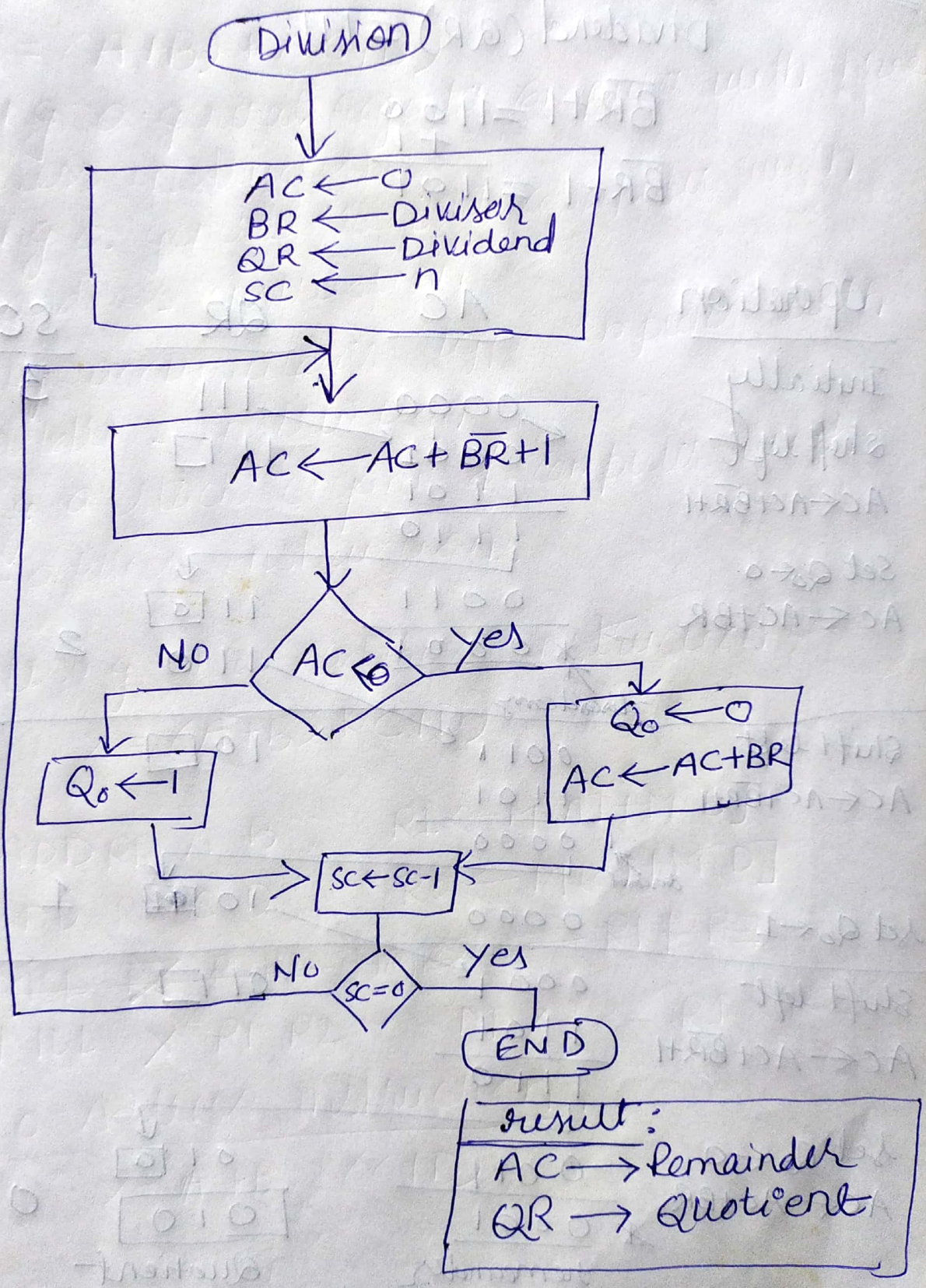




Hardware of Division Algorithm



Flowchart of Division Algorithm



ex ⇒ Divide 111 by 011 using division algorithm

Soln ⇒

Divisor (BR) = 011 = 0011

Dividend (QR) = 111

$\overline{BR}+1 = 1100 + 1$

$\overline{BR}+1 = 1101$

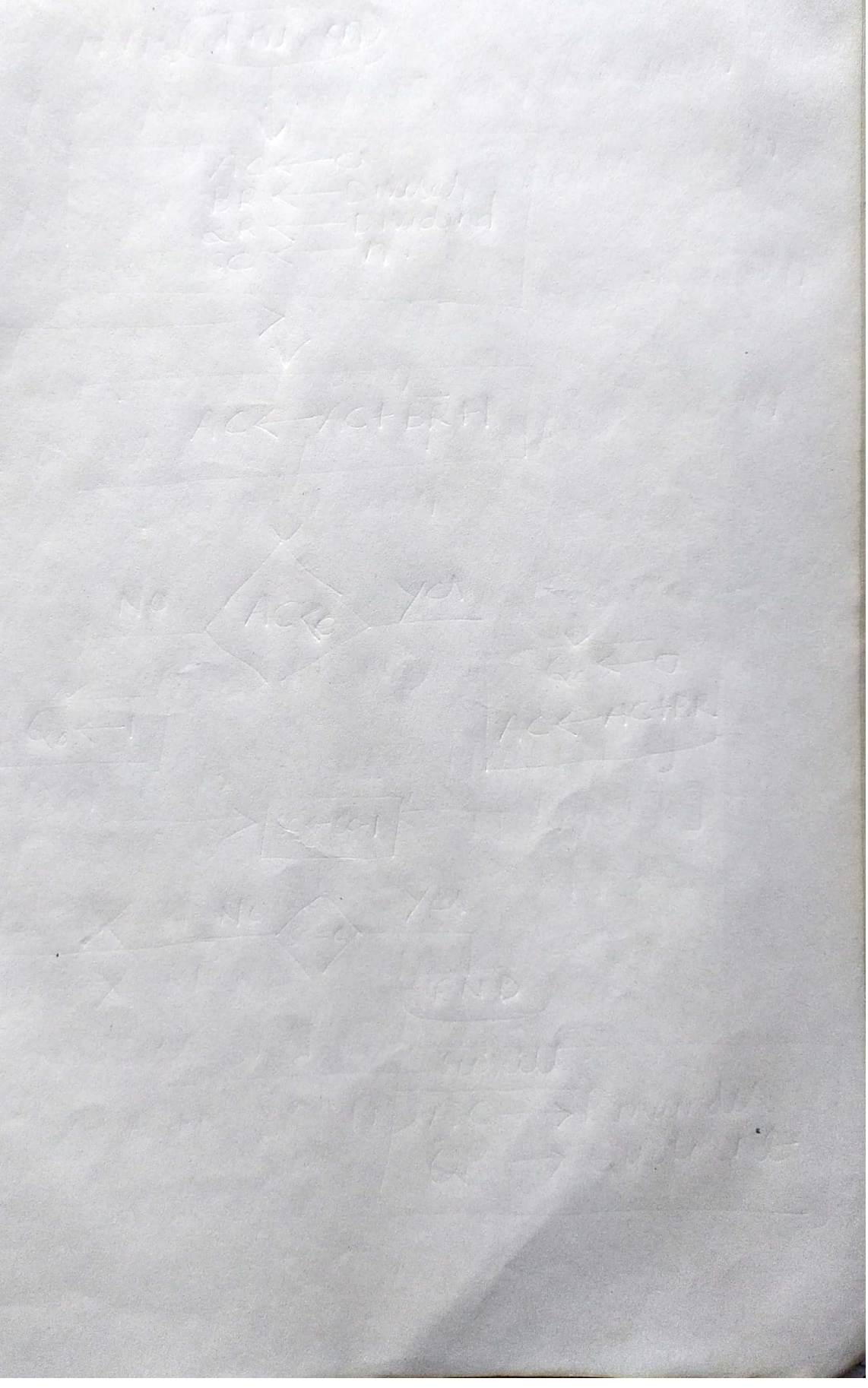
Operation	AC	QR	SC
Initially	0000	111	3
shift left	0001	11 □	
$AC \leftarrow AC+\overline{BR}+1$	1101		
	1110		
Set $Q_0 \leftarrow 0$		11 0	
$AC \leftarrow AC+BR$	0011		
	* 0001	110	2
	(discard carry)		
Shift left	0011	10 □	
$AC \leftarrow AC+\overline{BR}+1$	1101		
	10000		
	(discard)		
		10 1	1
Set $Q_0 \leftarrow 1$	0000		
shift left	0001	01 □	
$AC \leftarrow AC+\overline{BR}+1$	1101		
	1110		
		01 0	
Set $Q_0 \leftarrow 0$	0011		
$AC \leftarrow AC+BR$	* 0001	010	0
	remainder	Quotient	

remainder = 0001
Quotient = 010

Ans

arithm

② Divide 1010 by 0011 using division algorithm.



IEEE standard for Floating Point Numbers

⇒ In 1985, IEEE (Institute of Electrical and electronic Engineers) gave a floating-point numbers standard for both single precision and double precision floating point numbers

⇒ In digital computers floating-point numbers consists of three part: ~~called significand or Man~~ a sign bit, an exponent part (representing the exponent on a power of 2) and Fractional part called significand or Mantissa

sign bit	Exponent	Mantissa or significand

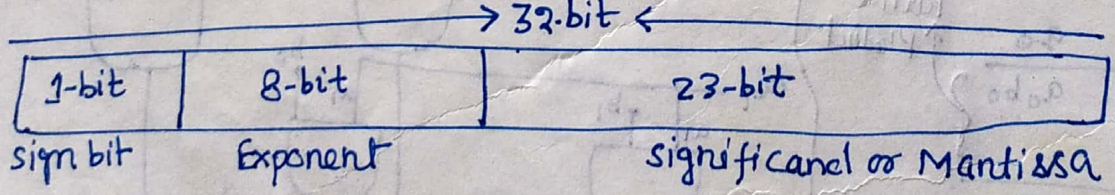
Floating-Point Representation

⇒ Here, sign bit will be always one either 0 for +ve or 1 for -ve

⇒ Exponent and Mantissa bits depends on whether it is ~~sigle~~ single precision or double precision

① Single Precision or 32-bit representation

IEEE has given the format for single precision as shown below:



-2^{n-1} to $2^{n-1}-1$

-2^7 to 2^7-1

-128 to +127

Here, Since exponent bit = 8-bit

$= 2^8 = 256$ (i.e 0-255 values can be taken Maximum)

$0 \le e \le 255$

$= 2^7 = 128$ (i.e → 0-127)

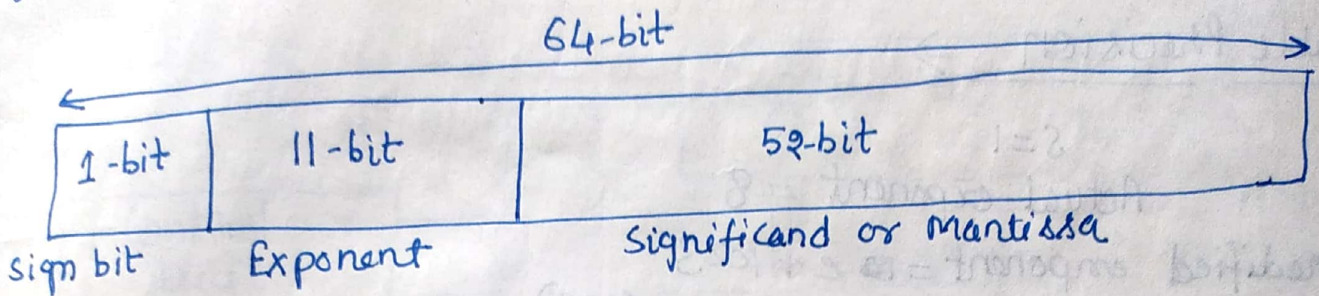
These bits are used for
00000000 → 0
11111111 → 00

$-126 \le e \le +127$

So, single Precision uses excess 127 bias

Double-Precision or 64-bit representation

IEEE has given the format for double-precision as shown below



Here, exponent = 11-bit

$2^{11} = 2048$ (maxm values can be represent)

$2^{10} = 1024$

$\therefore 0 \leq e \leq 1023$

i.e $-1022 \leq e \leq 1023$

So, in double precision we uses excess-1023 bias for exponent

ex ⇒ Represent -307.1875_{10} in single precision and double precision formats.

soln $(307)_{10} \rightarrow (100110011)_2$

$0.1875 \rightarrow (0.0011)_2$

$-307.1875 \rightarrow -100110011.0011_2$

taking Normalized form

$-307.1875_{10} = -1.0011001100110011 \times 2^{+8}$

For single Precision ⇒

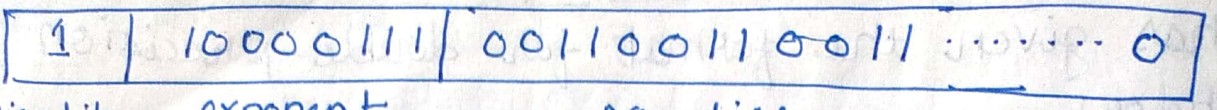
$-307.1875_{10} = -1.00110011001100110011 \times 2^{+8}$

Here sign bit = 1

Actual Exponent (e) = 8

Modified Exponent = e + 127 = 8 + 127 = $(135)_{10}$
= $(10000111)_2$

Mantissa = 00110011001100110011 → 0



Double Precision ⇒

S = 1

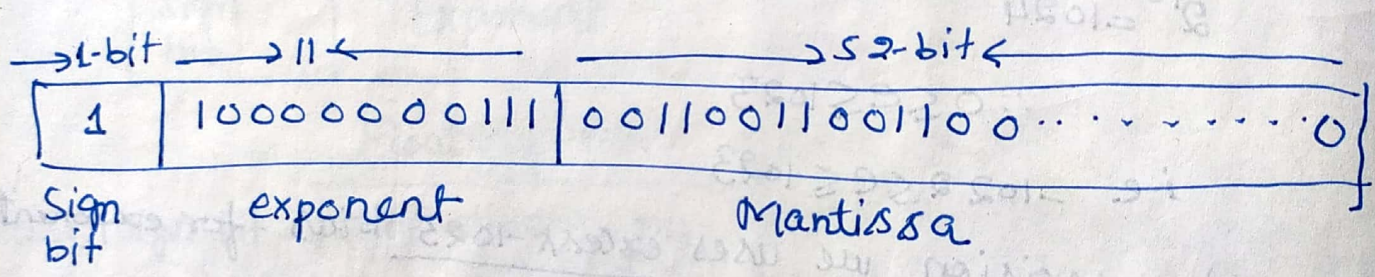
Actual exponent = 8

Modified exponent = e + 1023

= 8 + 1023 = $(1031)_{10}$

= 10000000111_2

Mantissa = 00110011001100 0



Logic Microoperations ⇒

⇒ Logic microoperations specify binary operations for strings of bits stored in registers.

Ex ⇒ P: R1 ← R1 ⊕ R2

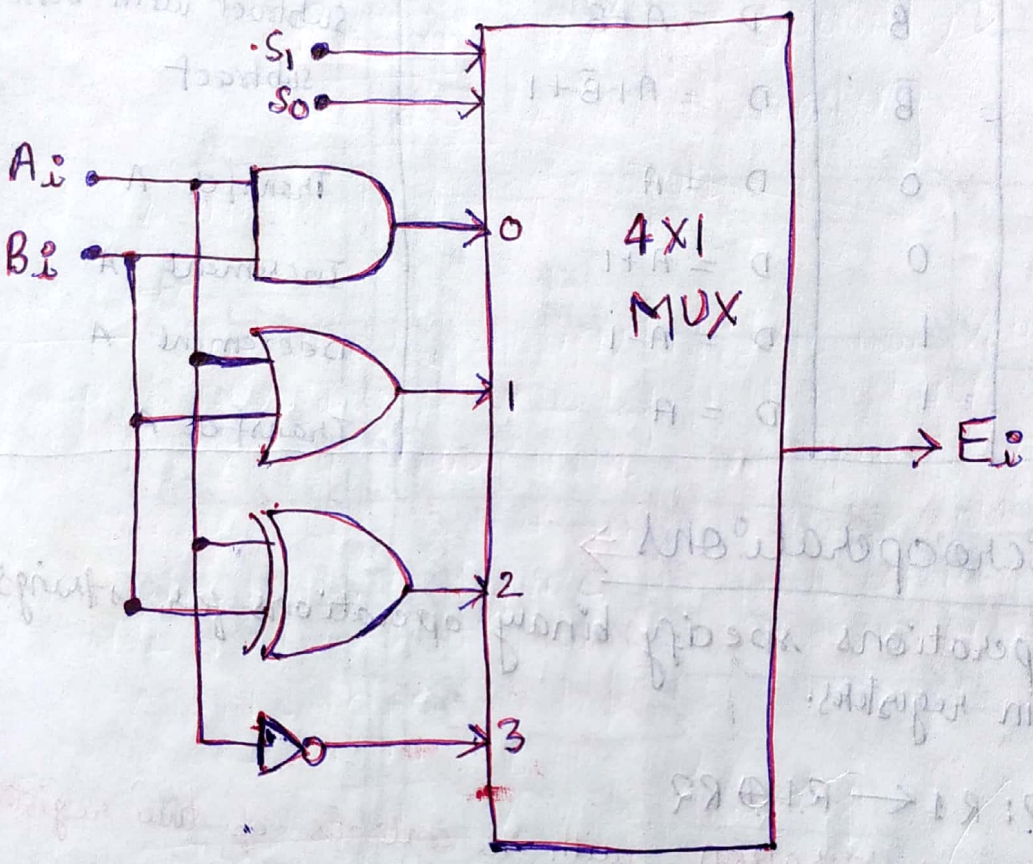
It is a XOR microoperation with the contents of two registers R1 and R2.

⇒ There are 16 different logic operations that can be performed with two binary variables, most computers use only four - AND, OR, XOR and complement. from which all other can be derived.

⇒ fig in next page shows one stage of circuit that generates the four basic logic microoperation. It consist of four gates and a multiplexer.

⇒ Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer.

⇒ The two selection inputs S_1 and S_0 choose one of the data inputs, of the multiplexer and direct its value to the output.



One stage of Logic circuit

S_1	S_0	output	operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

Function Table

(3) Design a digital circuits that perform four logic operations exclusive-OR, exclusive-NOR, NOR and NAND. Use two selection variable. Show Logic diagram of one typical stage?

Applications of Logic Microoperations

Selective-Set ⇒ The selective-set operation sets to 1 bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B.

ex ⇒

```
1 0 1 0    A before
1 1 0 0    B (logic operand)
-------
1 1 1 0    A After
```

⇒ It is is OR operation, so OR microoperation can be used to selective-set.

Selective-Complement ⇒ The selective complement operation complements bits in A where there are corresponding 1's in B. it does not affect bit positions that have 0's in B,

ex ⇒

```
1 0 1 0    A before
1 1 0 0    B (logic operand)
-------
0 1 1 0    A after
```

⇒ It is is XOR microoperation

Selective-clear ⇒ The selective-clear operation clears to 0 the bits in A only where there are corresponding 1's in B,

ex ⇒

```
1 0 1 0    A before
1 1 0 0    B (logic operand)
-------
0 0 1 0    A after
```

⇒ It is a $A \leftarrow A \wedge \bar{B}$ microoperation.

Mask operation ⇒ It is similar to selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B.

ex ⇒

```
1010    A before
1100    B (logic operand)
----
1000    A after masking
```

⇒ it is AND microoperation.

Insert ⇒ The insert operation inserts a new value into group of bits. This is done by first masking the bits and then ORing them with the required value. for example, suppose that an A register contains eight bits 01101010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits;

```
01101010    A before
00001111    B (mask)
--------
00001010    A After masking
```

and then insert the new value:

```
00001010    A before
10010000    B (insert)
--------
10011010    A after insertion
```

insert is an OR microoperation.

Clear ⇒ The clear operation compares the words in A and B and produces an all 0's result if the two numbers are equal. this is achieved by an XOR microoperation

ex ⇒

```
1010    A
1010    B
----
0000    A ← A⊕B
```

Shift Microoperations

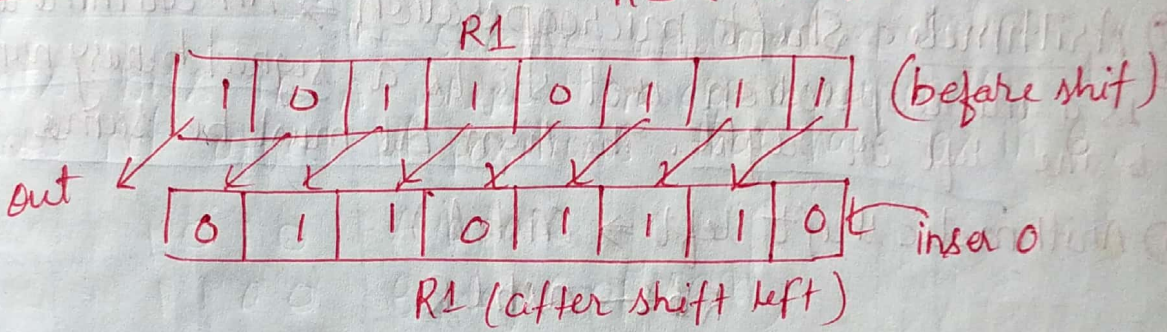
Shift microoperations are used for serial transfer of data. There are three types of shifts microoperations

1. logical shift microoperation
2. Circular shift microoperation
3. Arithmetic shift microoperation

① **Logical shift microoperation ⇒** A logical shift is one that transfer 0 through the serial input. there are two types of logical shift microoperation

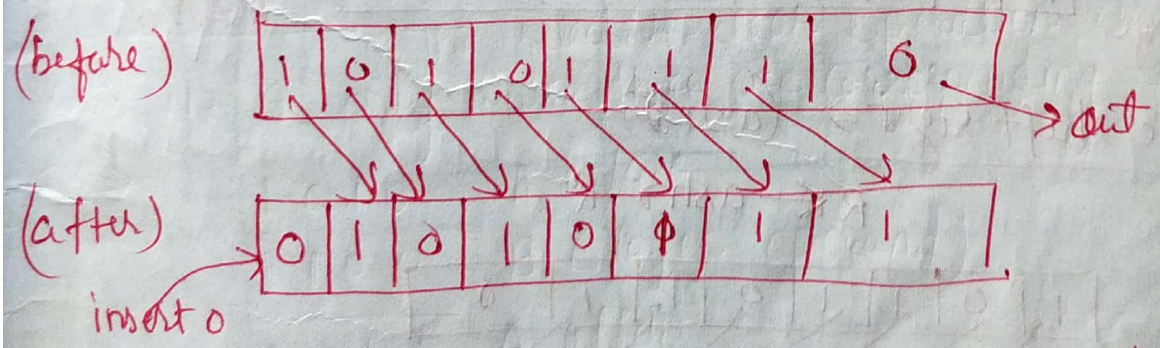
ⓐ **logical shift left (shl) ⇒**

$R1 \leftarrow shl\ R1$



ⓑ **logical shift right (shr) ⇒**

$R1 \leftarrow shr\ R1$

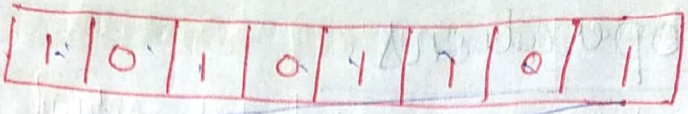


② **Circular Shift Microoperation ⇒** The circular shift circulates the bits of the register around the two ends without loss of information. There are two types of circular shift.

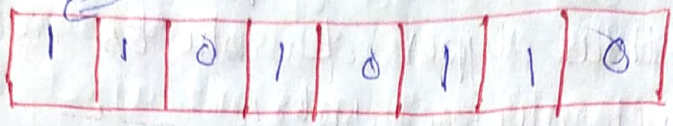
ⓐ. circular shift left ⇒ (cil)

$R \leftarrow cil\ R$

(before)

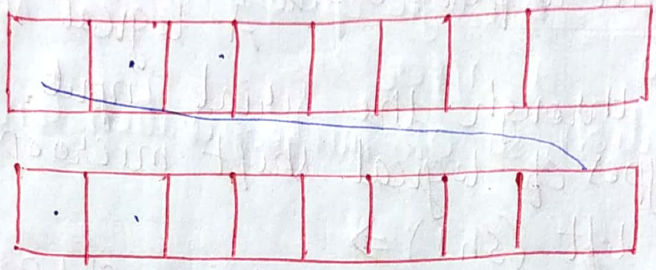


(after)



(b) Circular shift right ⇒ (Cir)

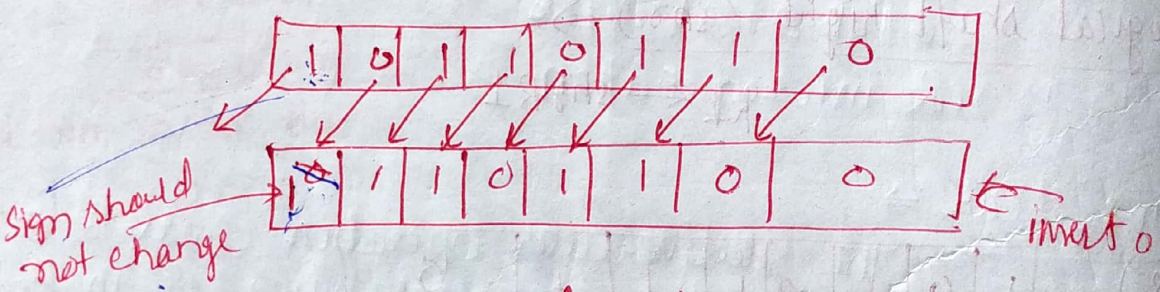
R ← Cir R



(3) Arithmetic Shift microoperation ⇒ An arithmetic shift is a microoperation that shifts a signed binary number to the left or right. The sign bit must be same.

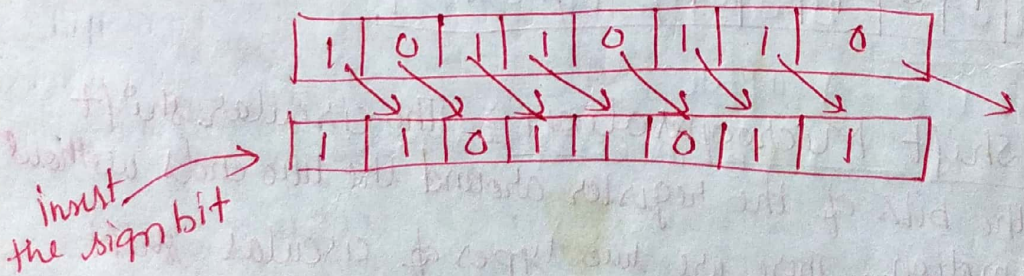
(a) Arithmetic shift left ⇒ (Ashl)

R ← ashl R



(b) Arithmetic shift right ⇒ (ashr)

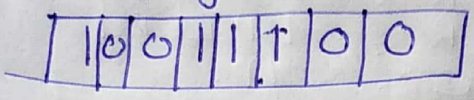
R ← ashr R



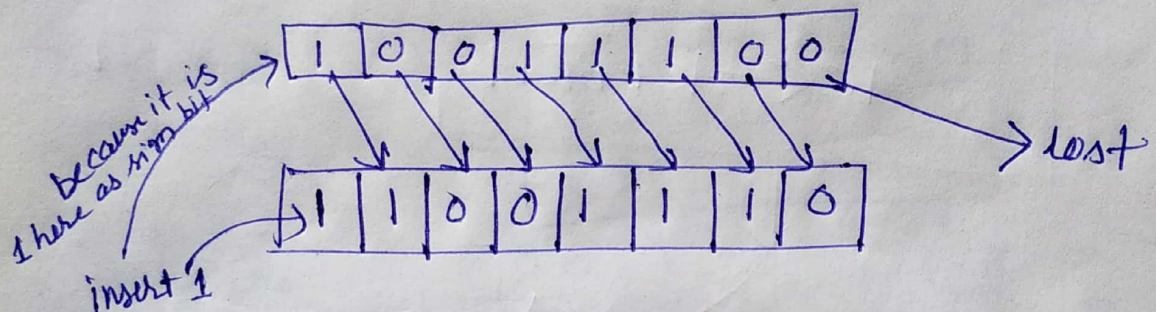
NOTE ⇒ In Arithmetic shif left if leftmost two bits are unequal then overflow condition occurs.

(a) An 8-bit register contains the binary value 10011100. What is the register value after an arithmetic shift right? Starting from the initial number 10011100, determine the register value after an arithmetic shift left, and state whether there is an overflow.

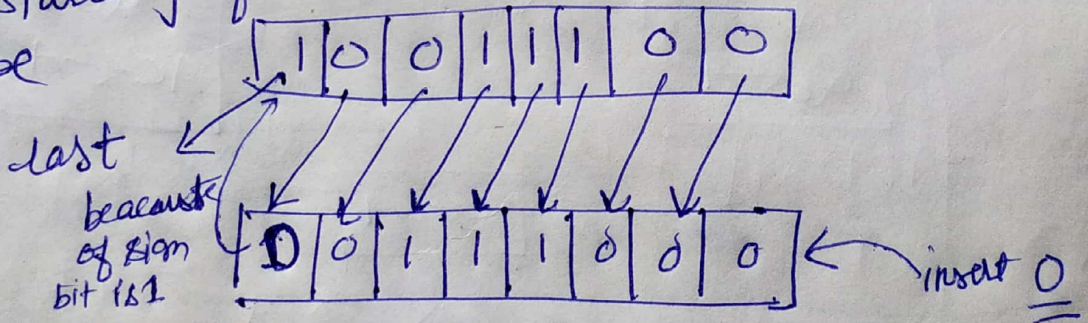
Soln Given 8-bit register with binary value



Register value after arithmetic shift right

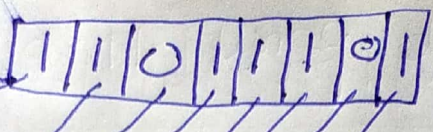


Now, starting from initial number, arithmetic shift left will be



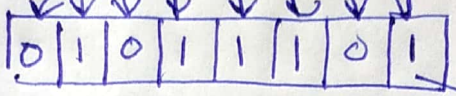
Here, overflow because a negative number changed to positive number.

(a) Starting from initial value R=11011101, determine the sequence of binary value in R after logical shift left, followed by circular shift right, followed by logical shift right and circular shift left.

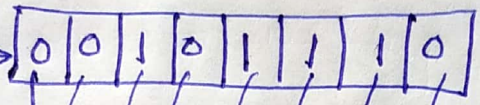
Sol^{n} ⇒ given R = 

logical shift left:

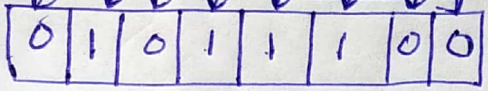
Circular shift right:



logical shift right:
0 insert



Circular shift left:



(d) Register A holds the 8-bit binary 11011001. Determine the B operand and the logic microoperation to be performed in order to change the value in A to:

(a). 01101101

(b). 11111101

Sol^{n} ⇒ given Register A binary value = 11011001

B = 10110100

(a).

Change value of A ← A ⊕ B = 01101101

So, B = 10110100 Ans

(b). A value = 11011001

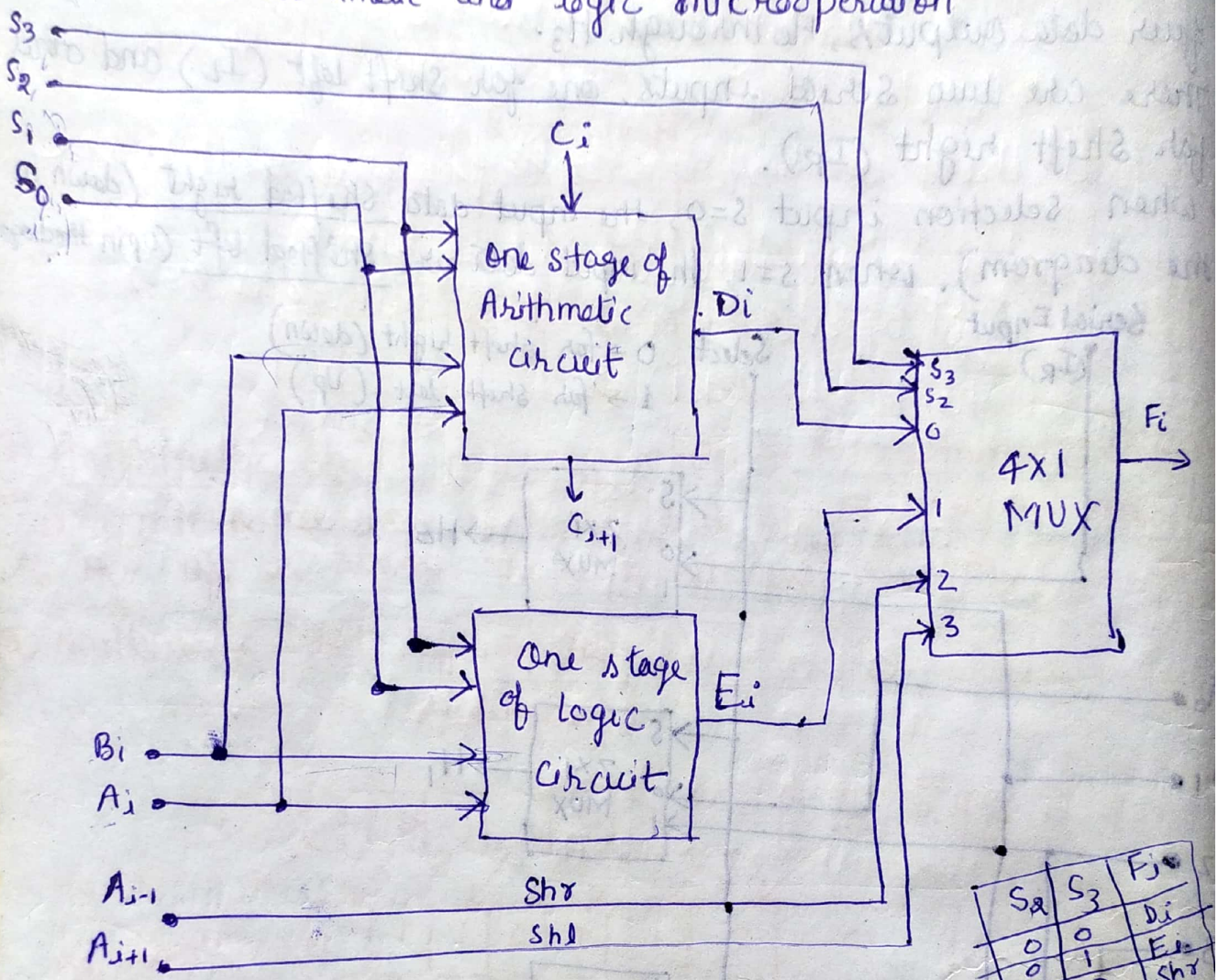
B 11111101 (OR)

Change value of A ← A ∨ B = 11111101

So, B = ~~00~~ 11111101 Ans

Arithmetic Logic Shift Unit (ALU)

⇒ The arithmetic, Logic and shift unit circuit can be combined into one ALU with common selection variables. Input A_i and B_i are applied to arithmetic and Logic unit. selection variable S_1 and S_0 select the arithmetic and logic microoperation



S_2	S_3	F_i
0	0	D_i
0	1	E_i
1	0	shr
1	1	shl

One Stage of ALU.

⇒ A 4x1 Multiplexer at the output chooses between an arithmetic output D_i and a logic output in E_i. and shift left and shift right.

⇒ The other two data inputs to the multiplexer receive inputs. A_{i-1} for the shift right and A_{i+1} for shift left operation

⇒ The input carry C_i must be connected to output carry C_{i+1} in sequen

⇒ The above circuits provides eight arithmetic operation, four Logic operation and two shift operations.

⇒ Each operation is selected with the five variables S_3, S_2, S_1, S_0 and C_{in}. The input carry C_{in} is used for selecting an arithmetic operation only.

⇒ when $S_3S_2 = 00$, First eight arithmetic operation are selected.

⇒ when $S_3S_2 = 01$, next four logic operation are selected. the input carry C_{in} has no effect during the logic operation and is shown by dont care X's.

⇒ The last two operations are shift operations when $S_3S_2=10$ shift right operation is selected.
when $S_3S_2=11$ shift left operation is selected.

Operation Select					operation	Fuction	
S_3	S_2	S_1	S_0	Cin			
0	0	0	0	0	F=A	Transfer A	
0	0	0	0	1	F=A+1	Increment A	
0	0	0	1	0	F=A+B	Addition	
0	0	0	1	1	F=A+B+1	Add with carry	
0	0	1	0	0	$F=A+\bar{B}$	Subtract with borrow	
0	0	1	0	1	$F=A+\bar{B}+1=A-B$	Subtraction	
0	0	1	1	0	F=A-1	Decrement A	
0	0	1	1	1	F=A	Transfer A	
0	1	0	0	X	$F=A \wedge B$	AND	Logic
0	1	0	1	X	$F=A \vee B$	OR	
0	1	1	0	X	$F=A \oplus B$	XOR	
0	1	1	1	X	$F=\bar{A}$	Complement A	
1	0	X	X	X	F=shrA	Shift right A	
1	1	X	X	X	F=shlA	Shift left A	

Function Table for Arithmetic shift unit

4-to-16 to