



COA UNIT-2 Notes

Computer Organization & Architecture (Dr. A.P.J. Abdul Kalam Technical University)

Adders:

- Half Adders
- Full Adders
- n-bit Binary Parallel Adder
- Carry Look-ahead Adder (or Fast adder or speed adder).

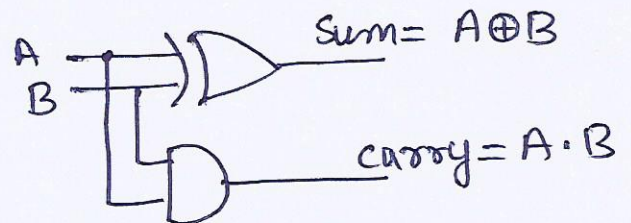
Half Adder:

Half adder is a combinational circuit that adds two bits.

Truth Table of Half Adder:

input		output	
A	B	Sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

circuit of Half adder



Full Adder:

Full adder is a combinational circuit that adds 3 bits.

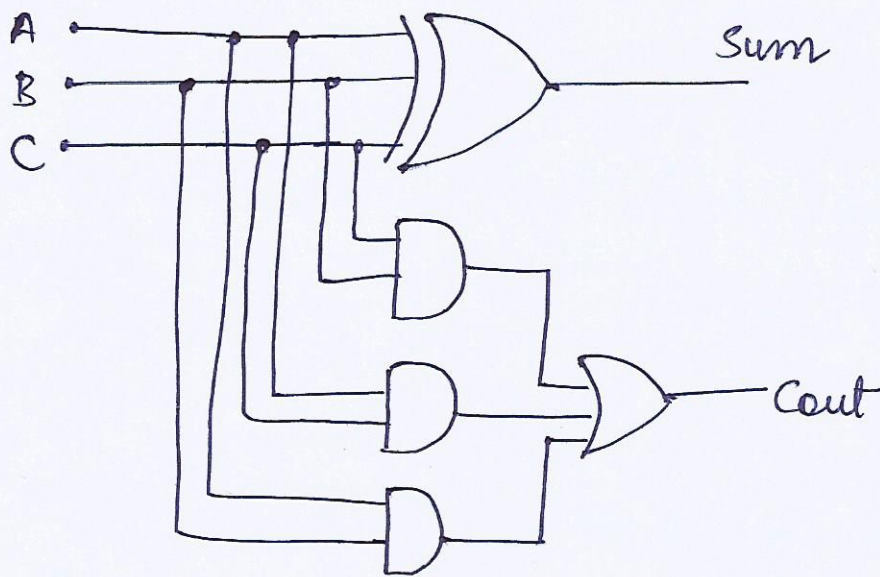
Truth Table:

A	B	C	Sum	Count
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\text{Sum} = A \oplus B \oplus C$$

$$\text{Count} = AB + BC + AC$$

Logic diagram of Full adder:



$$\text{cout} = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

$$= \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC + ABC + ABC$$

$$= BC[\bar{A} + A] + AC[\bar{B} + B] + AB[\bar{C} + C]$$

$$= BC \cdot 1 + AC \cdot 1 + AB \cdot 1$$

$$\boxed{\text{cout} = BC + AC + AB}$$

Properties.

$$\{ \because x + x = x \}$$

$$\{ x + \bar{x} = 1 \}$$

$$\{ x \cdot 1 = x \}$$

n-bit Binary Parallel Adder:

→ An n-bit binary parallel adder adds 2 n-bit numbers.

→ An n-bit binary adder uses n full adders.

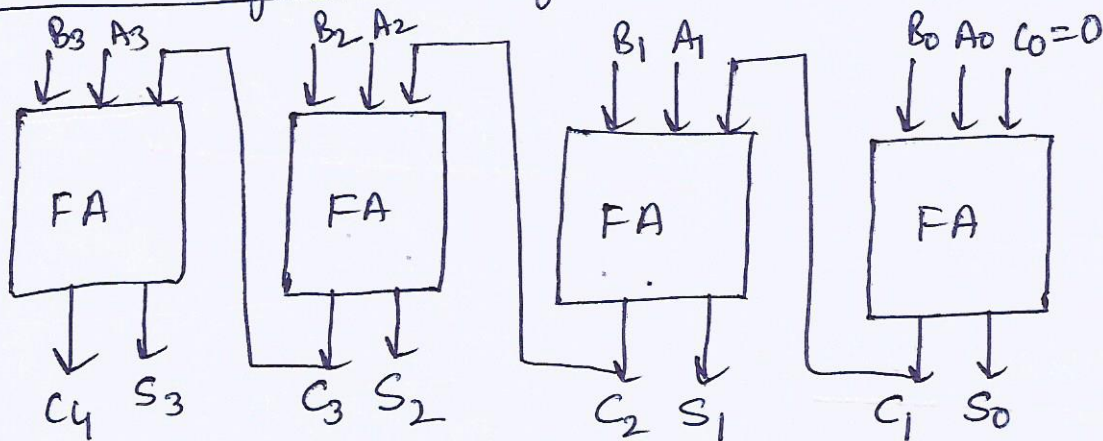
Example An 4-bit binary adder adds two 4-bit binary numbers

$$A = A_3 A_2 A_1 A_0$$

$$B = B_3 B_2 B_1 B_0$$

Addition

$$\begin{array}{r}
 C_3 \ C_2 \ C_1 \ C_0=0 \Rightarrow \text{carry input} \\
 A_3 \ A_2 \ A_1 \ A_0 \\
 + \ B_3 \ B_2 \ B_1 \ B_0 \\
 \hline
 S_3 \ S_2 \ S_1 \ S_0 \Rightarrow \text{sum} \\
 C_4 \ C_3 \ C_2 \ C_1 \Rightarrow \text{carry output}
 \end{array}$$

4-bit binary adder diagram:

A 4-bit binary parallel adder consists of 4 full adders in cascade, with output carry from one full adder, connected to the carry input of the next full adder.

Carry lookahead Adder:

Drawback of n-bit parallel adder -

- The carry output of one stage (or full adder) is connected to the carry input of the next higher stage (or full adder).

This carry is called ripple carry.

- Therefore, the sum and carry can not be produced until the input carry occurs.
- This leads to a time delay in addition process. This delay is known as carry propagation delay.

Carry-Lookahead Adder:

This method utilizes logic gates to look at the lower order bits of augend and addend to see if a higher order carry is to be generated or not

Carry look-ahead concept uses two functions

① Carry Propagate (P_i)

② Carry Generate (G_i)

Full adder using two half adders:



Carry Propagation $P_i = A_i \oplus B_i$
 Carry Generation $G_i = A_i \cdot B_i$

Both RHS does not contain any C_i

The output $S_i = P_i \oplus C_i$
 $C_{i+1} = G_i + P_i C_i$

we can create n-bit ~~look~~ carry look-ahead adder using these two equations.

Example

4 bit carry look-ahead adder:

For 4-bit, i is 0 to 3.

To remove the dependency of C_{i+1} on C_i , we need to express C_{i+1} into A_i, B_i and C_0 .

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

Put

$i=0$ $C_{0+1} = G_0 + P_0 C_0$

$$C_1 = G_0 + P_0 C_0$$

$i=1$

$C_{1+1} = G_1 + P_1 C_1$

$$C_2 = G_1 + P_1 [G_0 + P_0 C_0]$$

$i=2$

$C_{2+1} = G_2 + P_2 C_2$

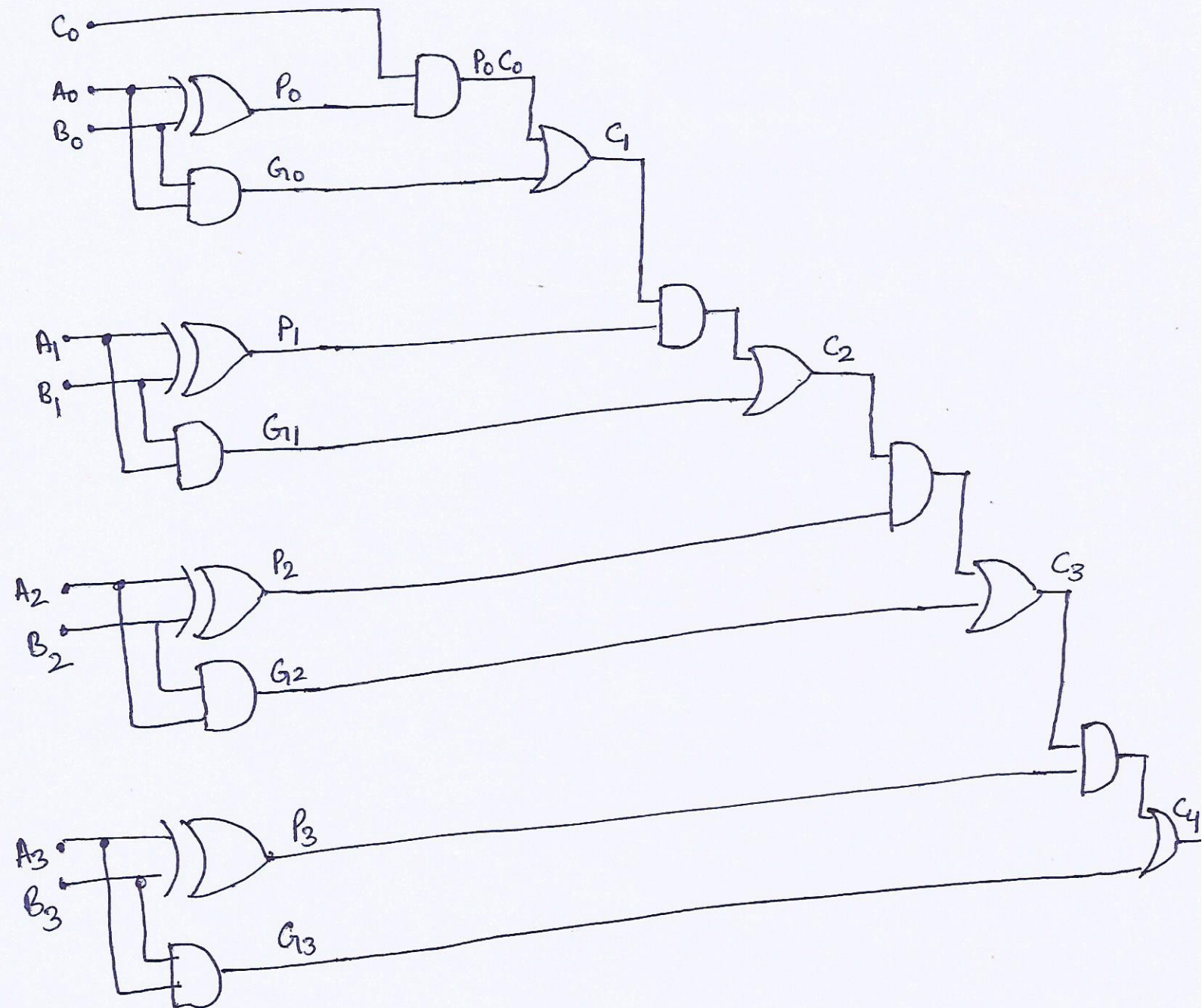
$$C_3 = G_2 + P_2 [G_1 + P_1 [G_0 + P_0 C_0]]$$

$i=3$

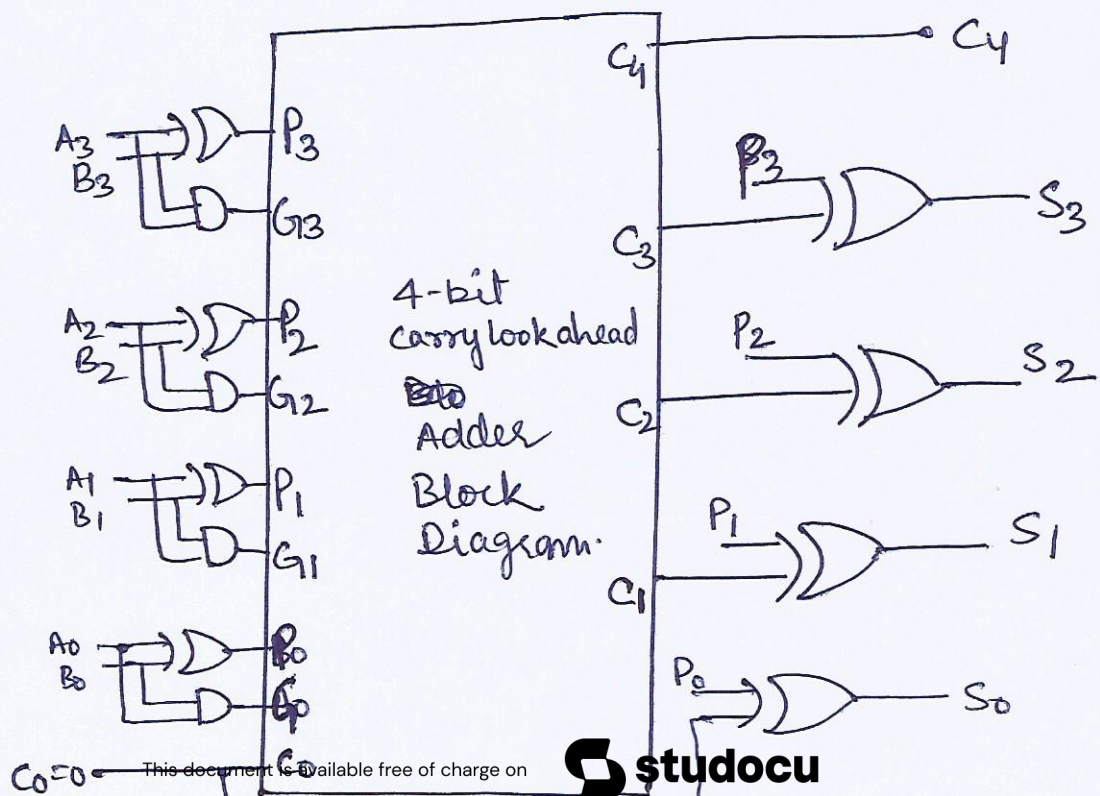
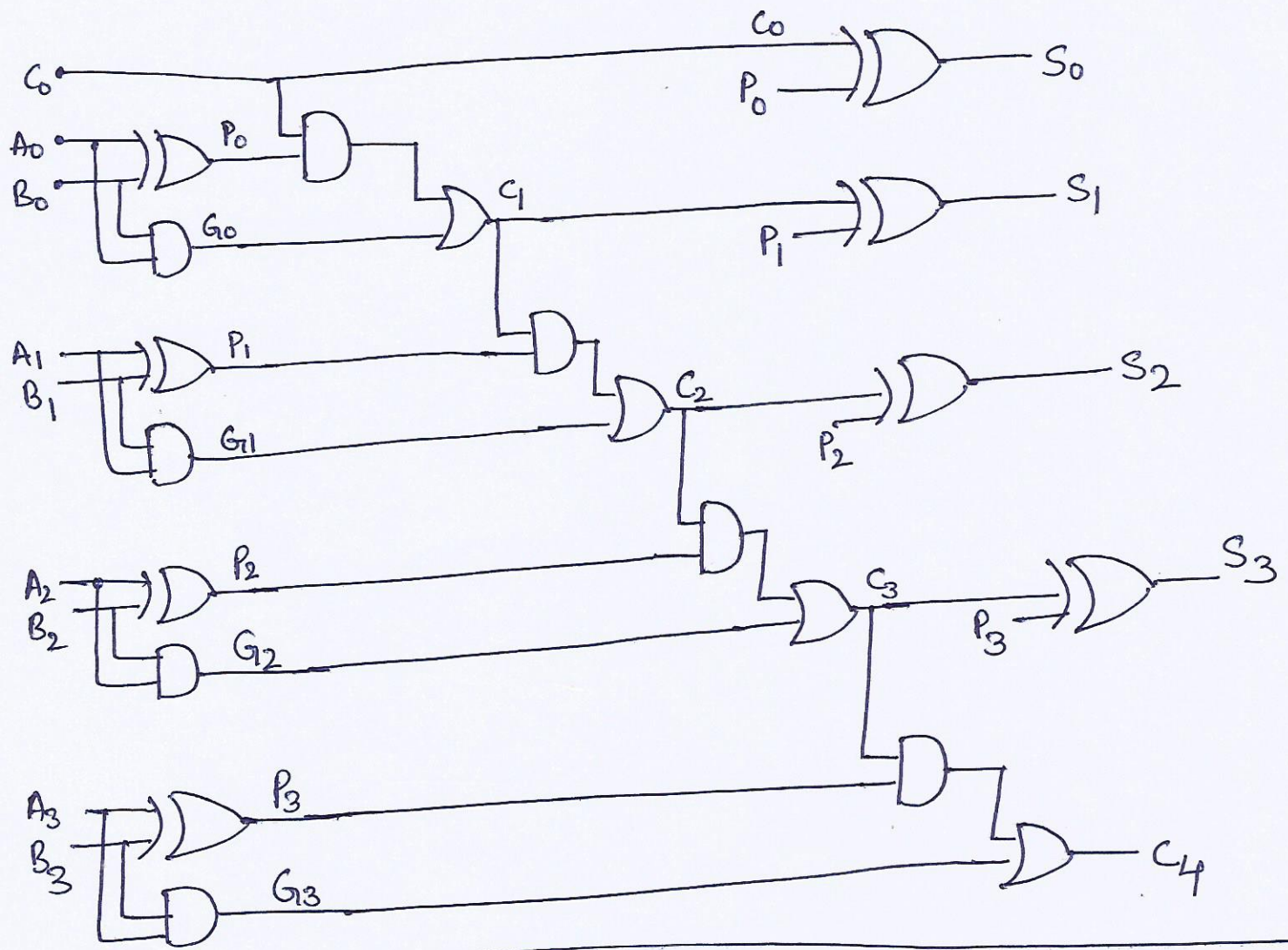
$$C_{3+1} = G_3 + P_3 C_3$$

$$C_4 = G_3 + P_3 [G_2 + P_2 [G_1 + P_1 [G_0 + P_0 C_0]]]$$

4-bit carry look-ahead ~~Adder~~ Carry Generation circuit

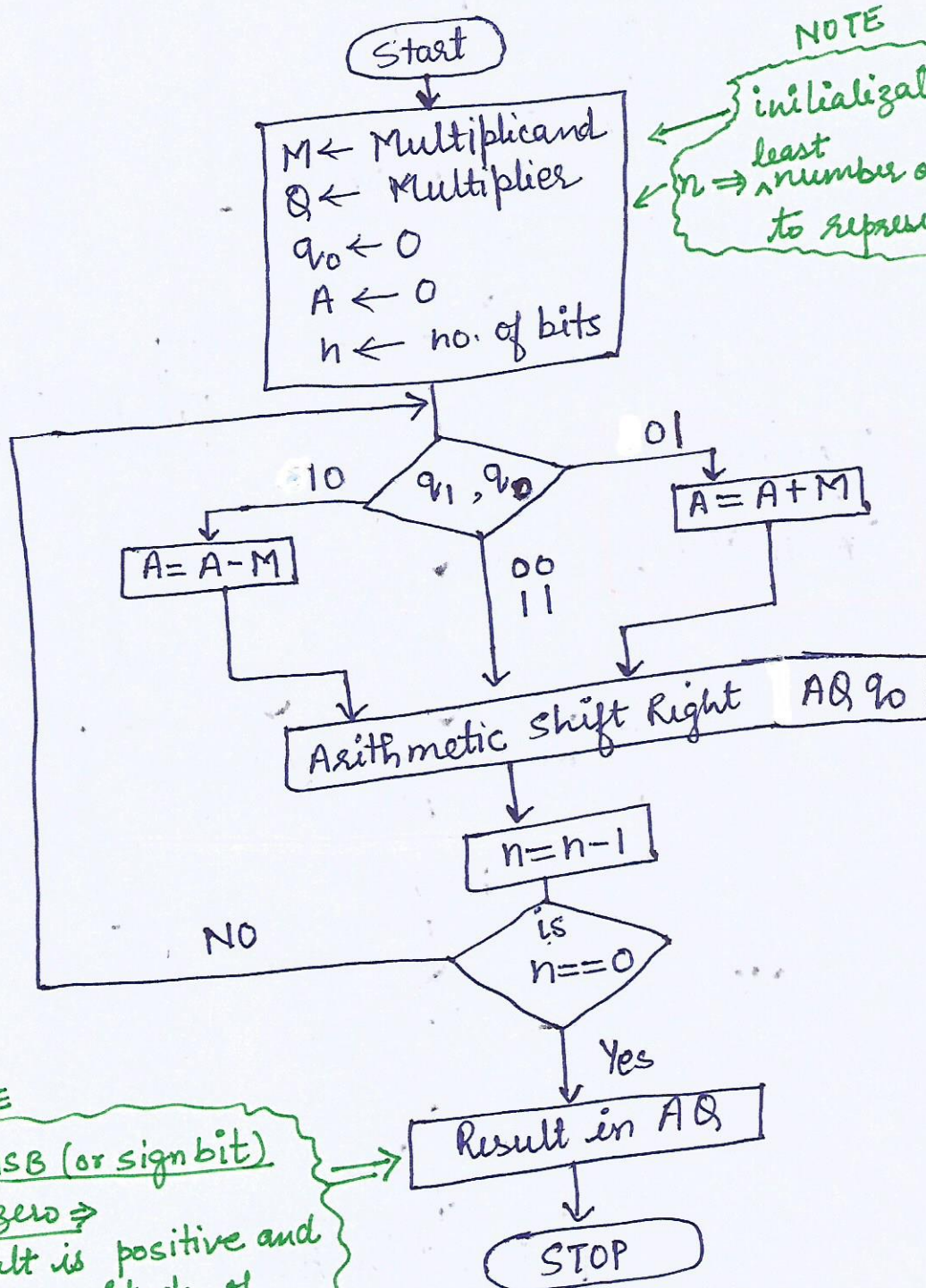


4 bit carry lookahead adder circuit:



Signed Multiplication (Booths Algorithm)

Flow chart:



NOTE

if MSB (or sign bit) is zero $\Rightarrow$

Result is positive and the magnitude of the product.

if MSB (or sign bit) 1

Result is negative take 2's complement to get the magnitude and place negative sign

Signed Booth Multiplication Example

Example $(-7) \times (+3) = (-21)$

$$M = (-7)_{10} = 2^3(0111) = (1001)_2$$

$$-M = (0111)_2$$

Tracing Table

	n	A	Q_{n-1}	Q_n	Action/Comment
	4	0000	0011	0	initialization
STEP ①		0111	0011	0	$A = A - M \Rightarrow A = A + (-M)$
		0011	1001	1	ASR A Q_{n-1}
					$n = n - 1$ $n = 3$
STEP ②	3	0011	1001	1	ASR A Q_{n-1}
		0001	1100	1	$n = n - 1$ $n = 2$
STEP ③	2	0001	1100	1	$A = A + M$
		1010	1100	1	ASR A Q_{n-1}
		1101	0110	0	$n = n - 1$ $n = 1$
STEP ④	1	1101	0110	0	ASR A Q_{n-1}
		1110	1011	0	$n = n - 1$ $n = 0$
	0	1110	1011	0	

Result $\Rightarrow A \ Q$

11101011

↑ sign bit is one. so the result is negative, take 2's complement to get the magnitude and place negative sign.

Final result: $(-21)_{10}$

11101011

↓ 1's complement

00010100

↓ +1

00010101 $\Rightarrow (21)_{10}$

Array Multiplier:

Mindakshi

Binary Multiplication is done by doing additions. Partial Products are calculated by multiplying the multiplicand by each bit of the multiplier and then summing the partial products.

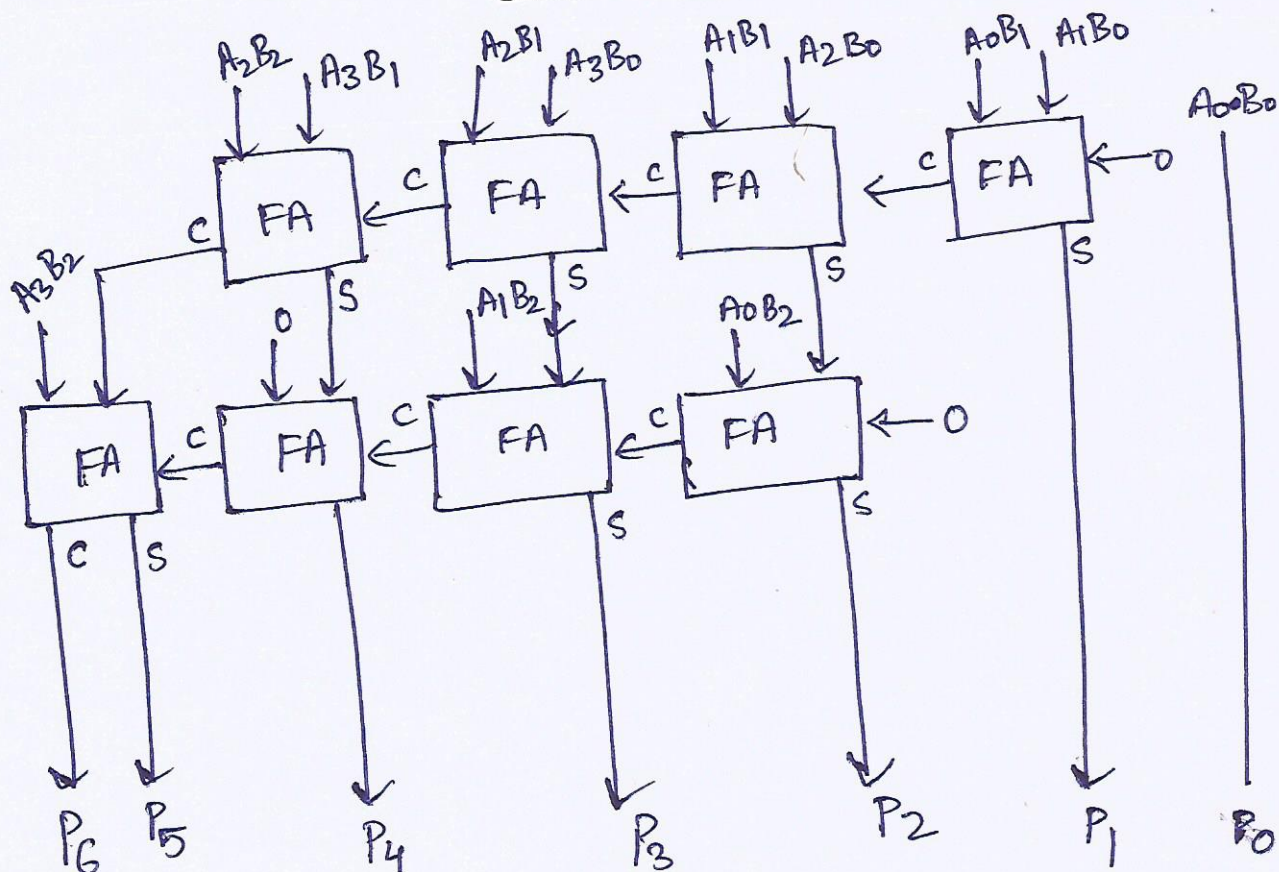
Ques: Design 4x3 binary or Array multiplier.

Given $A \Rightarrow A_3 A_2 A_1 A_0$

$B \Rightarrow B_2 B_1 B_0$

Number of bits in product $\Rightarrow 4 + 3 = 7$

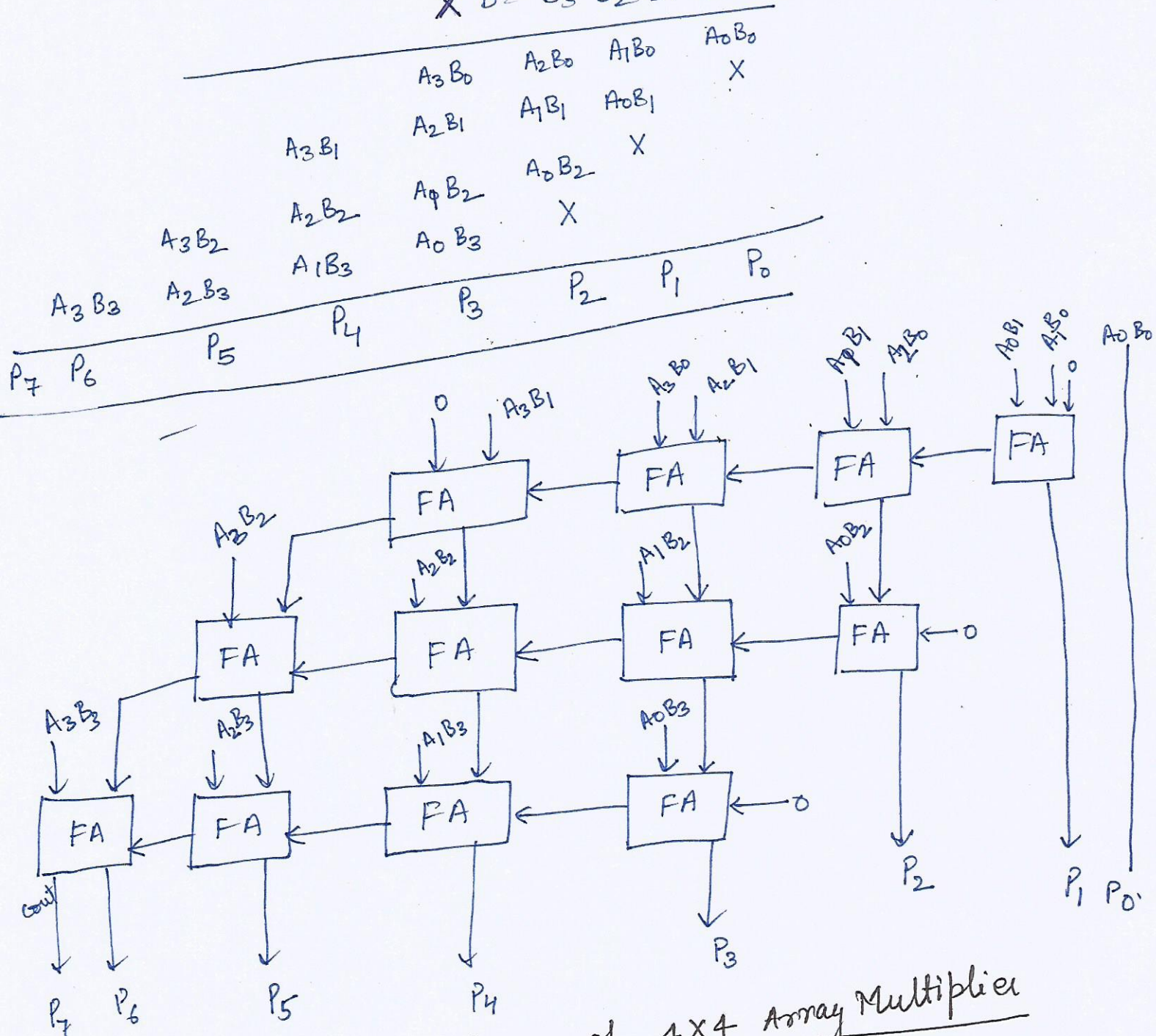
A_3	A_2	A_1	A_0
$\times$	B_2	B_1	B_0
$A_3 B_0$	$A_2 B_0$	$A_1 B_0$	$A_0 B_0$
			$\times$
$A_3 B_1$	$A_2 B_1$	$A_1 B_1$	$A_0 B_1$
			$\times$
$A_3 B_2$	$A_2 B_2$	$A_1 B_2$	$A_0 B_2$
			$\times$
P_6	P_5	P_4	P_3
			P_2
			P_1
			P_0



4X4 Binary Array Multiplier:

$$A = A_3 A_2 A_1 A_0$$

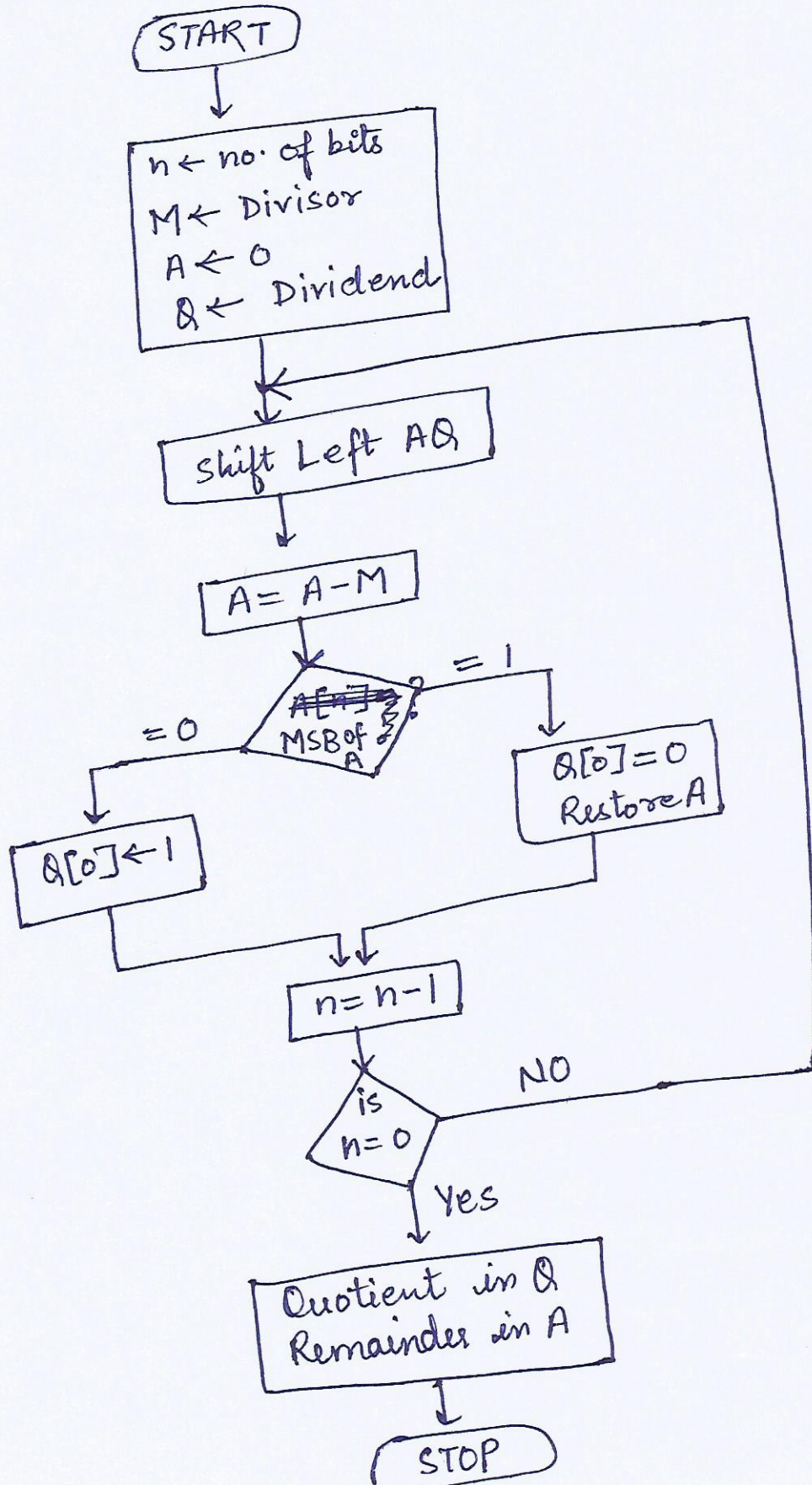
$$\times B = B_3 B_2 B_1 B_0$$



circuit Diagram of 4X4 Array Multiplier

Division of unsigned Integer: Restoring Division Algorithm:

Flow chart:



Example: Divide 11 by 3.

(Q) Dividend = 11

(M) Divisor = 3

$M = 3 \Rightarrow (00011)_2 \Rightarrow$ Number of bits in $M = n+1$

$Q = 11 \Rightarrow (1011)_2$

$-M = (11101)_2$

Tracing Table

n	M	A	Q	Action/comment
				Initialization
4	00011	00000	1011	
		00001	011?	Shift left AQ
		11110	011?	$A = A - M \Rightarrow 00001 + 11101 = 11110$
				$A[n] = 1$
		00001	0110	$Q[0] \leftarrow 0$ Restore A
3		00010	110?	$n = n - 1$
		11111	110?	Shift left AQ
		00010	1100	$A = A - M$
		00010	1100	$Q[0] \leftarrow 0$, Restore A
2		00101	100?	$n = n - 1$
		00010	100?	SL AQ
		00010	100?	$A = A - M$
		00010	1001	$Q[0] \leftarrow 1$
1		00010	1001	$n = n - 1$
		00101	001?	SL AQ
		00010	001?	$A = A - M$
		00010	0011	$Q[0] \leftarrow 1$

n	M	A	Q	Action/comment
0	00011	00010	0011	$n = n - 1$

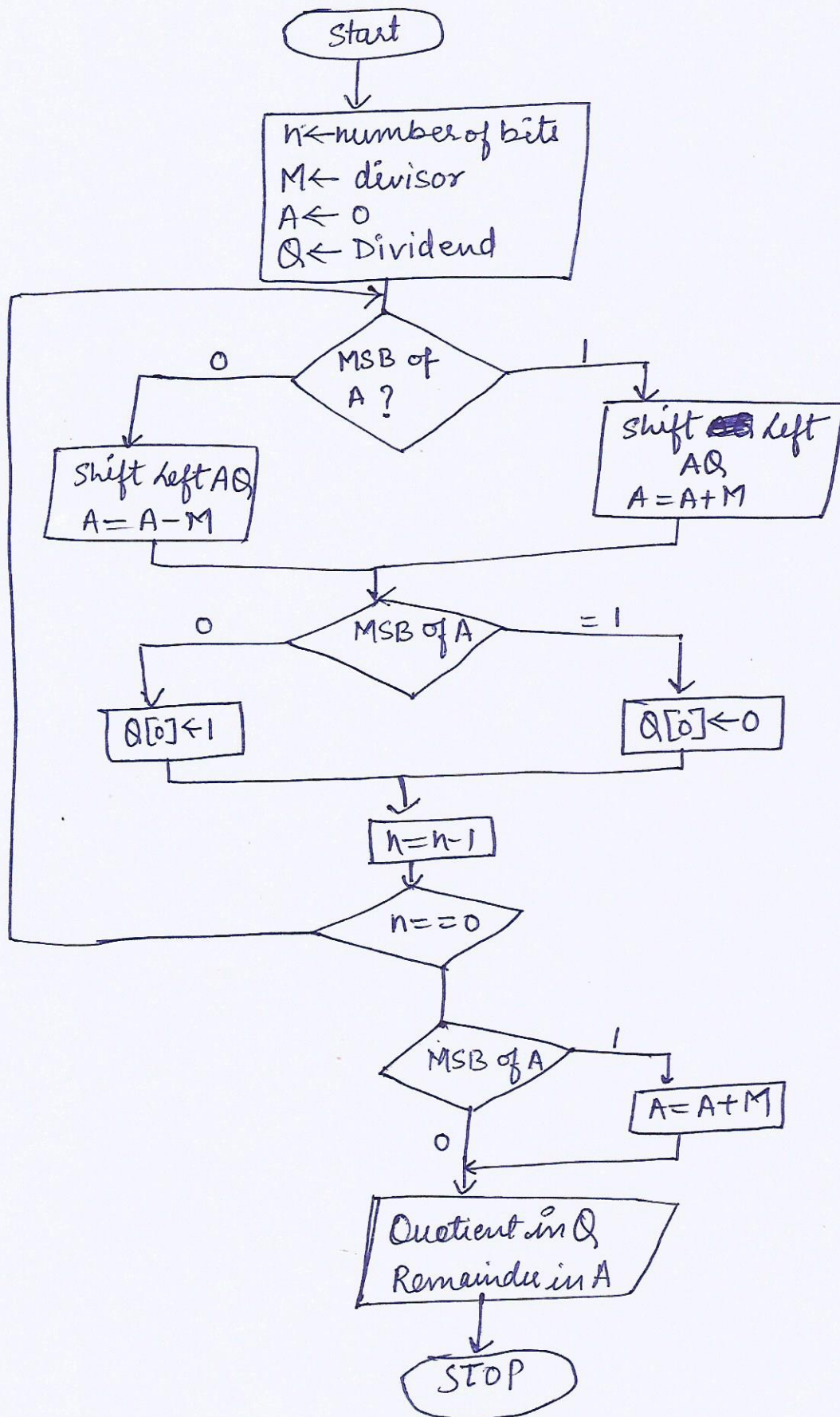
Now Quotient in Q = 0011 $\Rightarrow$ 3

Remainder in A = 00010 $\Rightarrow$ 2

11 (Dividend) / 3 (Divisor) $\Rightarrow$ 3 Quotient
2 Remainder

Flow chart of Unsigned Integer Division

Non-Restoring Method



Example Divide 11 by 3.

(Q) Dividend = 11

(M) Divisor = 3

Number of bits $\Rightarrow n = 4$

Number of bits $M = n + 1 = 5$

$-M = 11101$

Tracing Table:

n	M	A	Q	Action / comment
				Initiatiziation
4	00011	00000	1011	SL AQ
		00001	011?	$A = A - M$
		11110	011?	$Q[0] \leftarrow 0$
		11110	0110	$n = n - 1$
3		11110	0110	
		11100	110?	SL AQ
		11111	110?	$A = A + M$
		11111	1100	$Q[0] \leftarrow 0$
		11111	1100	$n = n - 1$
2		11111	1100	
		11111	1100?	SL AQ
		00010	100?	$A = A + M$
				$\begin{array}{r} 11111 \\ + 00011 \\ \hline 100010 \\ \uparrow \text{discard it} \end{array}$
		00010	1001	$Q[0] \leftarrow 1, n = n - 1$
1		00101	001?	SL AQ
		00010	001?	$A = A - M$
		00010	0011	$Q[0] \leftarrow 1$
		00010	0011	$n = n - 1$
0 (Yes)				

MSB of $A = 0 \Rightarrow$ then Result:

Quotient_{in} $Q = (0011)_2 = 3$

Remainder in $A = (00010)_2 = 2$

Ques Divide 7 by 3 using Non-Restoring method.

$$\begin{aligned} Q = \text{Dividend} = 7 &\Rightarrow 111 \\ M = \text{Divisor} = 3 &\Rightarrow 0011 \end{aligned} \quad \left. \vphantom{\begin{aligned} Q = \text{Dividend} = 7 &\Rightarrow 111 \\ M = \text{Divisor} = 3 &\Rightarrow 0011 \end{aligned}} \right\} n=3$$

$$-M = 1101$$

0 $\rightarrow$ Subt
1 $\rightarrow$ Add.

n	A	Q	Action
3	0000 $\uparrow$ 0001 +1101	111 11? 11?	Initialization SL AQ, A = A - M
	1110 $\uparrow$ 1110	11? 110	Q[0] $\leftarrow$ 0
2	1101 0011	10? 10?	Shift Left A = A + M
carry disc	0000 0001	10? 01?	Q[0] $\leftarrow$ 1
1	1101		A = A - M
	1110	010	Q[0] $\leftarrow$ 0
0			

Remainder Quotient

$$A \Rightarrow 1110$$

$$A \Rightarrow 1110$$

$$\begin{array}{r} +M \Rightarrow +0011 \\ \hline 0001 \end{array}$$

Remainder = 1 Quotient = 2

Floating Point Representation :

→ we can represent a floating point number in the form

$$\pm S \times B^{\pm E}$$

$B \Rightarrow$ Base, For binary $B=2$.

$\Rightarrow$ This number can be stored in a binary word with three fields

- Sign (plus or minus)
- significand or Mantissa (S)
- Exponent E.

Sign	Biased Exponent	Mantissa or significand
------	-----------------	-------------------------

IEEE 754 Format :

- ① Single Precision (32 bit)
- ② Double Precision (64 bit)

	Sign bit	Biased Exponent bits	Bias	Fraction bits
Single Precision	1	8	127	23 bits
Double Precision	1	11	1023	52 bits

Two functions for representing floating point numbers

→ Normalization

→ Biasing

Sign $\Rightarrow$ 0 for positive numbers
1 for negative numbers.

A normalized number is one in which the most significant digit of the significand is one.

For base 2 representation

$$1. bbb \dots b \times 2^{\pm E}$$

$$1. M \times 2^{\pm \text{Exponent}}$$

where $M \Rightarrow$ Mantissa / fraction / significand

→ Because the most significant bit is always one, it is unnecessary to store.

→ Given a number that is not normalized, the number may be normalized by shifting the radix point to the right of left most 1 bit and adjusting the exponent accordingly.

→ Biasing: Exponent is stored in biased representation.

→ In single precision case, 8 bits field yields the numbers 0 through 255.
with a bias of 127 ($2^7 - 1$), the true exponent values are in range -127 to +128

→ In double precision case, 11 bits field yields the numbers 0 through 2047
with a bias of 1023 ($2^{10} - 1$), the true exponent values are in range. -1023 to +1024

⇒ Why Biasing? So that the bits in exponents can be treated as unsigned or nonnegative integers.

Instead of storing exponent in 2's complement, we can store it as unsigned number with the help of biasing

Ques: Represent $(-1234.125)_{10}$ into single Precision and double precision representation.

Ans:- ① First convert 1234.125 into binary number.

$$(1234.125)_{10} = 10011010010.001$$

② Now Normalize this number $\Rightarrow$ move the decimal point right to the leftmost 1.

$$1.0011010010001 \times 2^{+10}$$

Single Precision: 32 bit

(i) calculate biased exponent.

$$\begin{aligned} \text{Biased Exponent } E' &= E + \text{Bias} \\ &= +10 + 127 \\ &= 137 \\ &= 10001001 \end{aligned}$$

sign = negative = 1

sign	Biased Exponent	Mantissa (23)
1	10001001	001101001000100000000000

Double Precision: 64 bits:

(i) calculate biased exponent $E' = E + \text{Bias}$

$$= 10 + 1023$$

$$= 1033$$

$$= 10000001001$$

sign	Exponent (E')	Mantissa (52)
1	10000001001	00110100100010000000000000000000000000000000

Ques Represent 000100110101.001101 using single precision and double precision.

Given number 000100110101.001101

Normalized Number $\Rightarrow 1.00110101001101 \times 2^{+9}$

Biased Exponent (SP): $E' = E + 127$
 $= 9 + 127$
 $\Rightarrow 136$
 $\Rightarrow 10001000$

Biased Exponent (DP): $E' = E + 1023$
 $= 9 + 1023 \Rightarrow 1032$
 $\Rightarrow 100000001000$

single Precision

0	10001000	001101010011010000000000
---	----------	--------------------------

Double Precision

0	100000001000	00110101001101000000000000000000000000000000
---	--------------	--

Ques: Represent $-(0.00000100011001101)_2$ using single precision and double precision.

Normalize the number.

$$\Rightarrow 1.00011001101 \times 2^{-6}$$

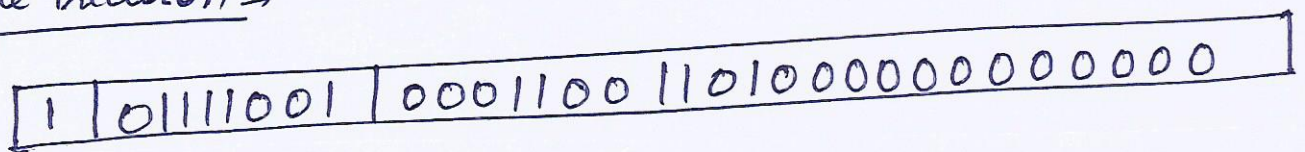
Biased Exponent in SP:

$$E' = E + \text{Bias} \Rightarrow -6 + 127 \\ \Rightarrow 121 \\ \Rightarrow 01111001$$

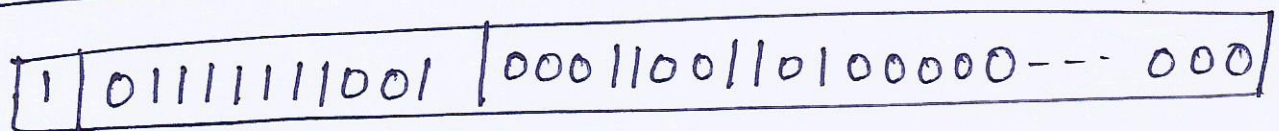
Biased Exponent in DP:

$$E' = E + \text{Bias} \Rightarrow -6 + 1023 \\ \Rightarrow 1017 \\ \Rightarrow 01111111001$$

single precision $\Rightarrow$



Double Precision:-



Floating-Point Addition and Subtraction:

- Four basic phases of the algorithm for addition and subtraction.

1. Check for zeros.
2. Align the significands
3. Add or subtract the significands
4. Normalize the result.

Floating Point Numbers	Addition and subtraction
$X = X_s \times B^{X_E}$ $Y = Y_s \times B^{Y_E}$	$X+Y = (X_s \times B^{X_E - Y_E} + Y_s) \times B^{Y_E}$ $X-Y = (X_s \times B^{X_E - Y_E} - Y_s) \times B^{Y_E}$ } $X_E \leq Y_E$

Example:

$$X = 0.3 \times 10^2 = 30$$

$$Y = 0.2 \times 10^3 = 200$$

$$\begin{aligned} X+Y &= (0.3 \times 10^{2-3} + 0.2) \times 10^3 = (0.03 + 0.2) \times 10^3 \\ &= 0.23 \times 10^3 \\ &= 230 \end{aligned}$$

$$\begin{aligned} X-Y &= (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (0.03 - 0.2) \times 10^3 \\ &= (-0.17) \times 10^3 \\ &= -170 \end{aligned}$$

Phase ①: Zero check: If either operand is 0, the other is reported as the result.

Phase ②: Significand alignment: to manipulate the numbers so that the two exponents are equal.

Alignment may be achieved by shifting either the smaller number to right (increasing its exponent) or shifting the larger number to left.

- often small number is picked for shifting

Phase ③ Addition: Next, two significands are added together, taking ^{into} account their signs.

Phase ④ Normalization: This phase normalize the result. Normalization consists of shifting significand digits left until the most significant digit is nonzero. Each shift causes a decrement of the exponent and thus could cause an exponent underflow. Finally, the result must be rounded off and the reported.

For floating-point addition and subtraction, it is necessary to ensure both the operands have the same exponent value. This may require shifting the radix point on one of the operands to achieve alignment.

A floating-point may produce one of these conditions:

- Exponent overflow
- Exponent Underflow
- Significant underflow
- Significant overflow

Exponent Overflow: A positive exponent exceeds the maximum possible exponent value. In some systems, this may be designated as $+\infty$ or $-\infty$.

Exponent Underflow: A negative exponent is less than the minimum possible exponent value. (e.g. -200 is less than -127) This means the number is too small to be represented and it may be reported as zero.

Significant underflow: In the process of aligning significands digits may flow off the rightend of the significand. So, some form of rounding is required.

Significant overflow: The addition of two significands of the same sign may result in a carry out of the most significant bit. This can be fixed by realignment.

Flow chart for Floating Point Addition and Subtraction

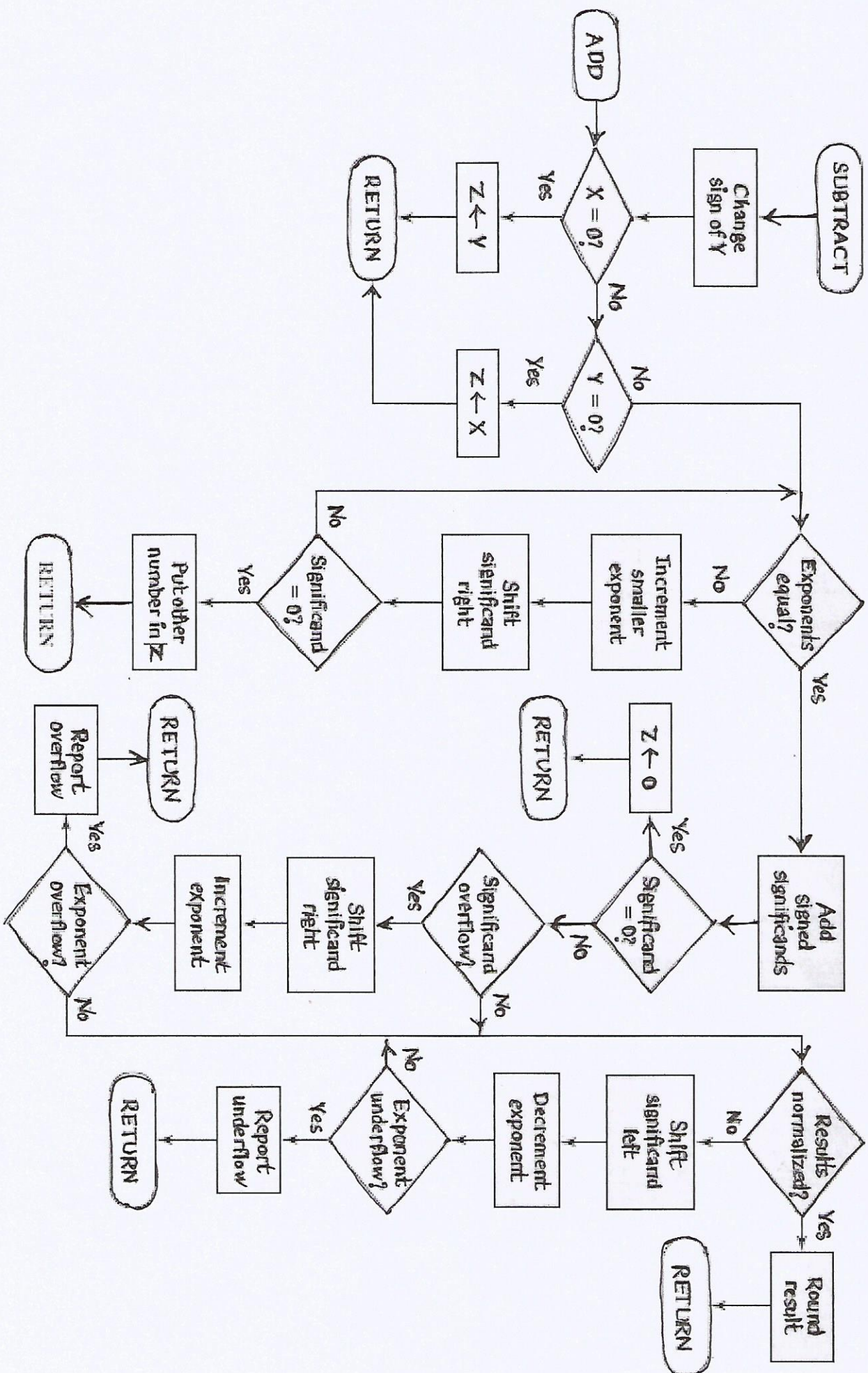


Figure 9.22 Floating-Point Addition and Subtraction ($Z \leftarrow Z \pm Y$)

Flow chart for Floating Multiplication

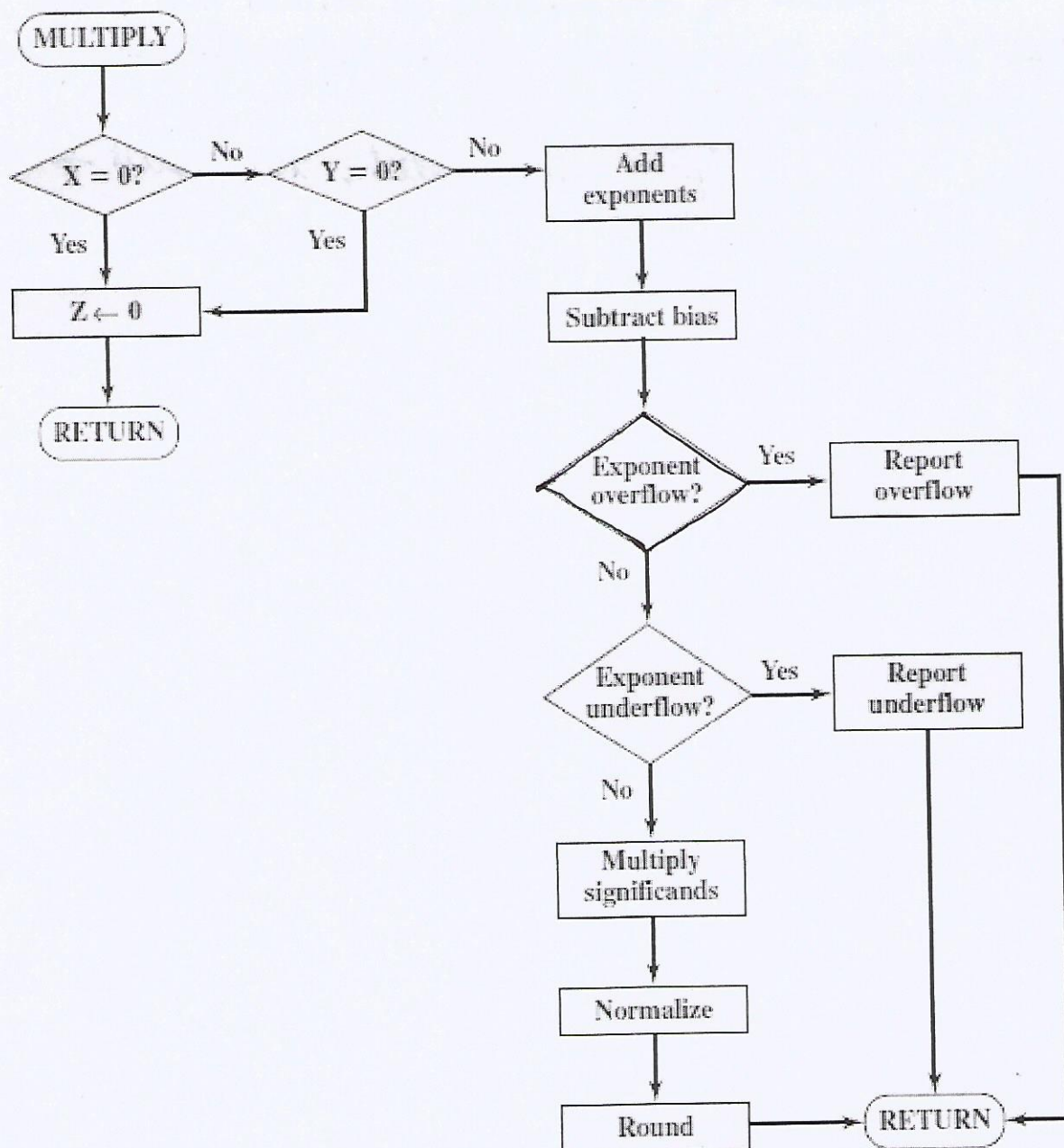


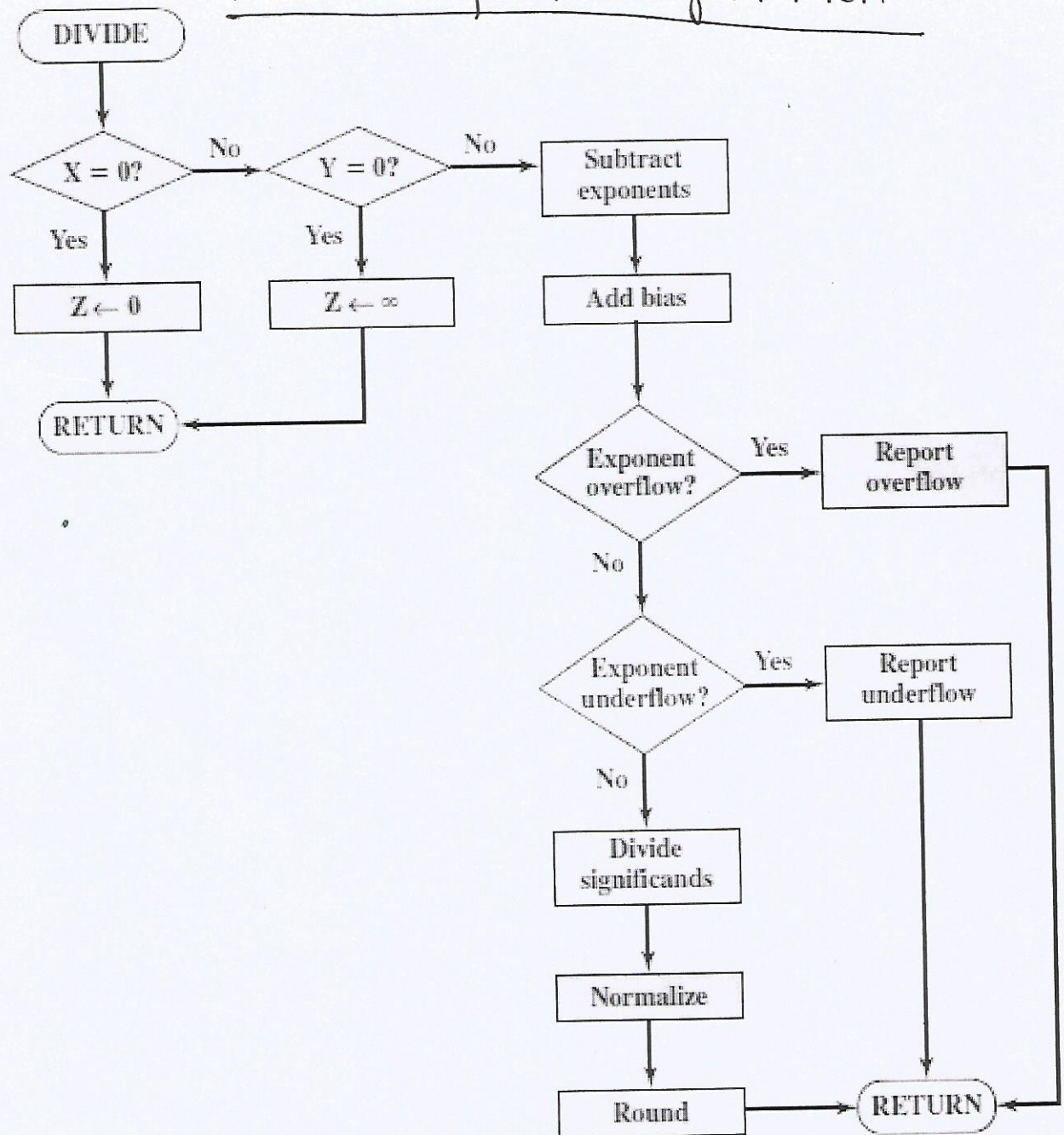
Figure 9.23 Floating-Point Multiplication ($Z \leftarrow X \times Y$)

Floating-Point Multiplication:

- if either operand is 0 (zero), zero is reported as result.
- The next, ~~step~~ step is to add the exponents. if the exponents are stored in biased form, the exponent sum would have doubled the bias. Thus, the bias must be subtracted from the sum. The result could be either an exponent underflow or exponent overflow, which would be reported.

- if the exponent of the product is within the range, the next step is to multiply the significand, taking into account their signs.
- After the product is calculated, the result is normalized and rounded.

Flow chart for Floating Division



Floating Point division:-

- Test for zero → if divisor is zero, an error is issued or result is set to infinity.
→ if dividend is zero, the result is zero.
- Next, the divisor exponent is subtracted from the dividend exponent. This removes bias, which must be added back in.
- Test are made for exponent underflow or overflow.
- The next step is to divide the significands.
- Followed by normalization and rounding.

Logic Microoperations :

→ Logic microoperations specify binary operations ~~for~~ for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables

For Example

- AND ($\wedge$)
- OR ($\vee$)
- NOT ($\neg$)
- NOR
- NAND
- X-OR ($\oplus$)
- X-NOR ($\odot$)

$$R_1 = 1010$$

$$R_2 = 1100$$

$$\begin{array}{r} \text{AND} \Rightarrow \\ \hline 1010 \\ 1100 \\ \hline 1000 \end{array}$$

$$\begin{array}{r} \text{OR} \Rightarrow \\ 1010 \\ 1100 \\ \hline 1110 \end{array}$$

$$\begin{array}{r} \text{NOT} \\ \hline 1010 \\ 0101 \end{array}$$

$$\begin{array}{r} \text{NOR} : \\ \hline 1010 \\ 1100 \\ \hline 0001 \end{array}$$

$$\begin{array}{r} \text{NAND} :- \\ 1010 \\ 1100 \\ \hline 0111 \end{array}$$

$$\begin{array}{r} \text{X-OR} \\ \hline 1010 \\ 1100 \\ \hline 0110 \end{array}$$

$$\begin{array}{r} \text{X-NOR} \\ \hline 1010 \\ 1100 \\ \hline 1001 \end{array}$$

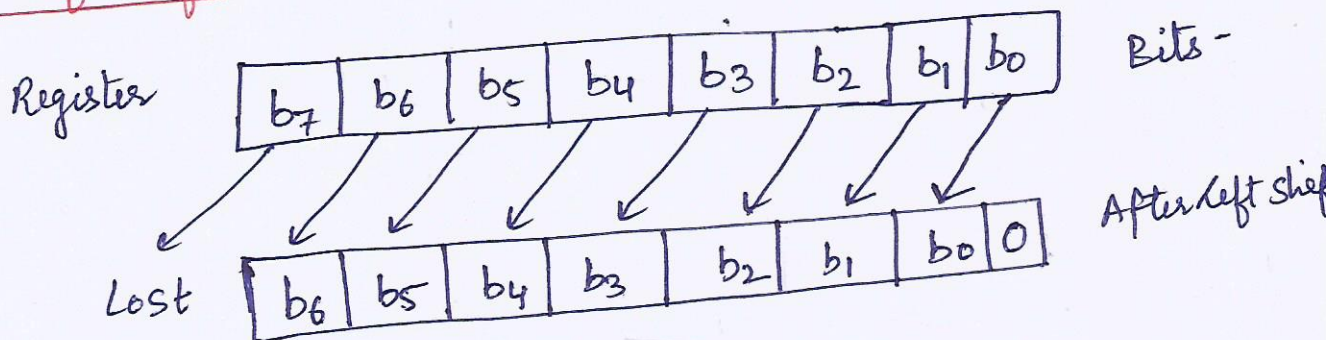
Shift Microoperations

→ Shift Micro operations are used for serial transfer of data.

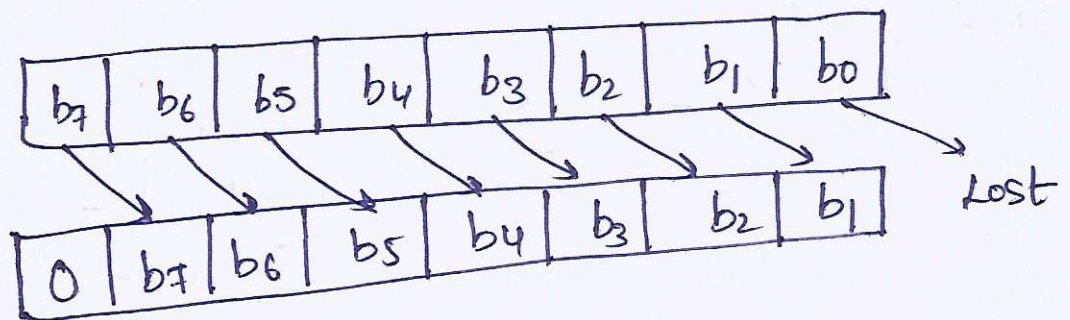
→ Types of Shift operations

- Logical shift
 - shift left
 - shift right
- Circular shift or Rotate
 - circular Right shift
 - circular left shift
- Arithmetic
 - Arithmetic Right shift
 - Arithmetic left shift

Logical shift left:



Logical shift Right:



Example

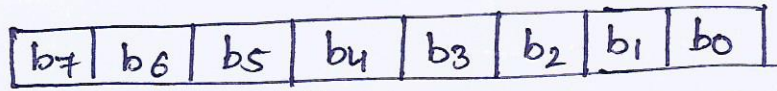
1 0 0 1 0 1 1 1
0 0 1 0 1 1 1 0 After shift Left

1 0 0 1 0 1 1 1
0 1 0 0 1 0 1 1 After shift Right

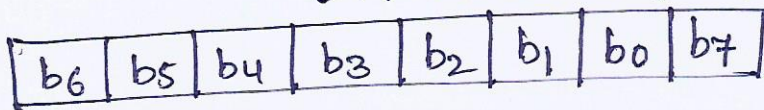
Circular shift: (Also known as Rotate operation)

→ circulates the bits of the register around the two ends without loss of information.

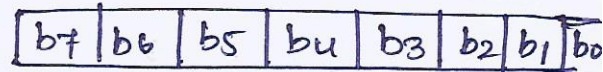
circular left shift



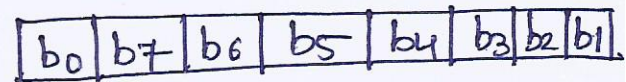
⇓ After circular left shift



Circular Right shift



After ⇓ circular Right shift



Example

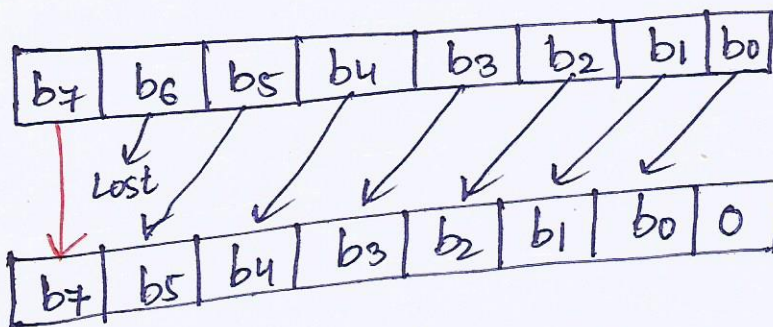
10010111
// // // //
00101111
Circular Left shift

10010111
\\ \\ \\ \\ \\ \\ \\ \\
11001011 ⇒ circular Right shift

Arithmetic shift:

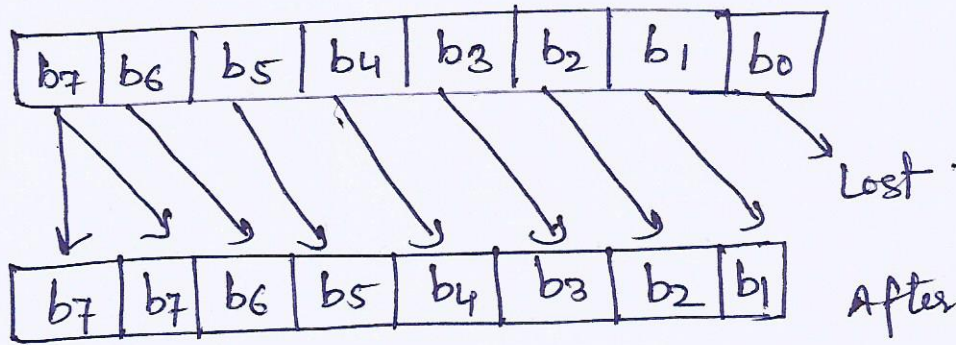
- An Arithmetic shift operation shift a signed binary number to the left or right
- Arithmetic shift leaves the sign bit unchanged.

Arithmetic Left Shift

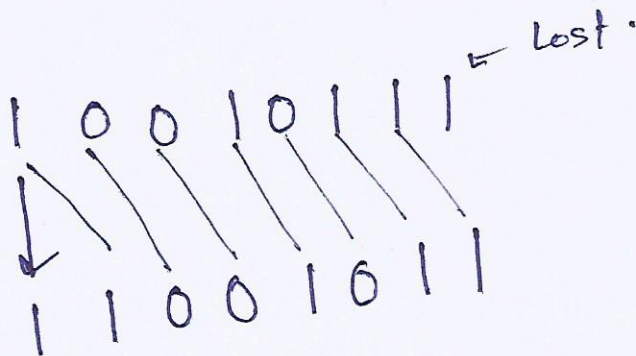


10010111
↓ ↓ ↓ ↓ ↓ ↓
10101110
lost

Arithmetic Right Shift :



After Arithmetic Right shift

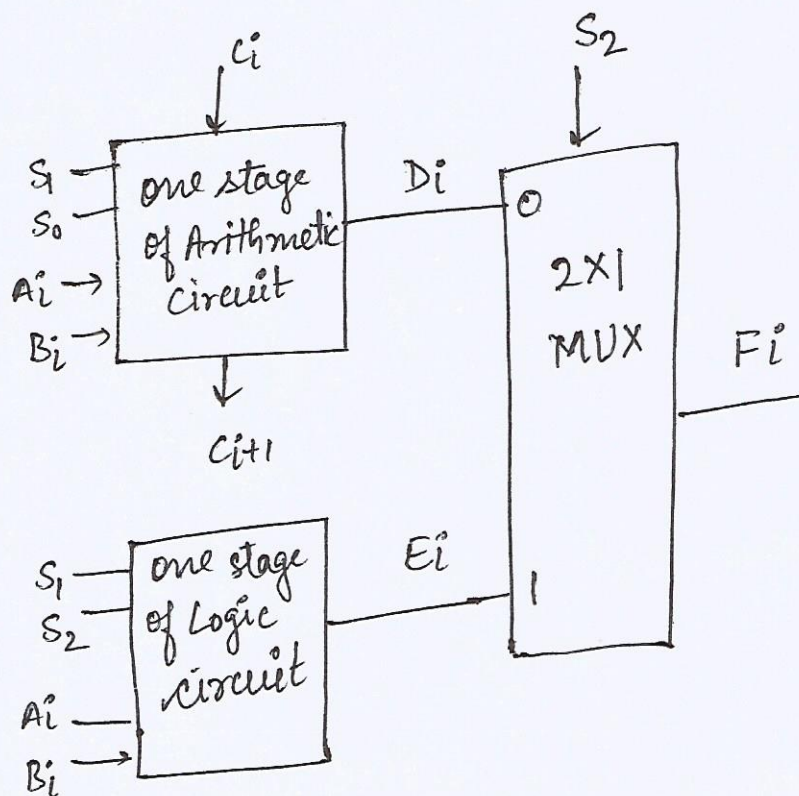


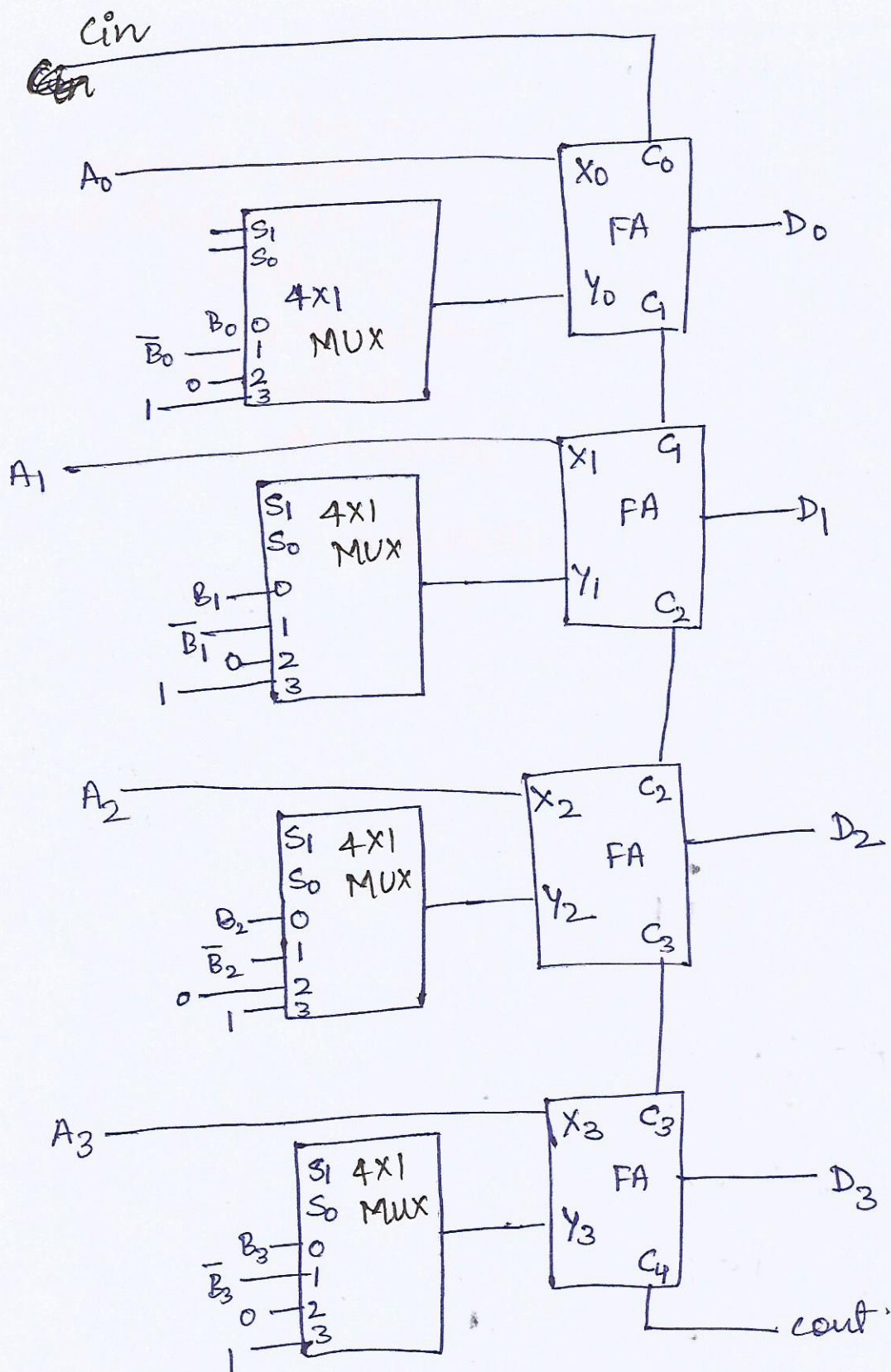
Arithmetic Logic Unit :

- CPU contains ALU, CU and Registers
- ALU is responsible for arithmetic and logical operations
- Basically ALU is a digital circuit that performs arithmetic operations like addition, subtraction, division and multiplication, and logical operations like AND, OR, XOR, NOT, etc.

- Types of ALU ⇒ ① Combinational ALU
② Sequential ALU.

combinational ALU :





4 bit Arithmetic Circuit performing operations like addition, subtraction, increment, decrement and transfer etc.

S_1	S_0	C_{in}	Input	Output $D = A + Y + C_{in}$
0	0	0	B	$D = A + B \rightarrow \text{Add}$
0	0	1	B	$D = A + B + 1 \rightarrow \text{Add with carry}$
0	1	0	$\bar{B}$	$D = A + \bar{B} \rightarrow \text{Subtract with Borrow}$
0	1	1	$\bar{B}$	$D = A + \bar{B} + 1 \rightarrow \text{Subtract}$
1	0	0	0	$D = A \rightarrow \text{Transfer A}$
1	0	1	0	$D = A + 1 \rightarrow \text{Increment A}$
1	1	0	1	$D = A - 1 \rightarrow \text{Decrement A}$
1	1	1	1	$D = A \rightarrow \text{Transfer A}$

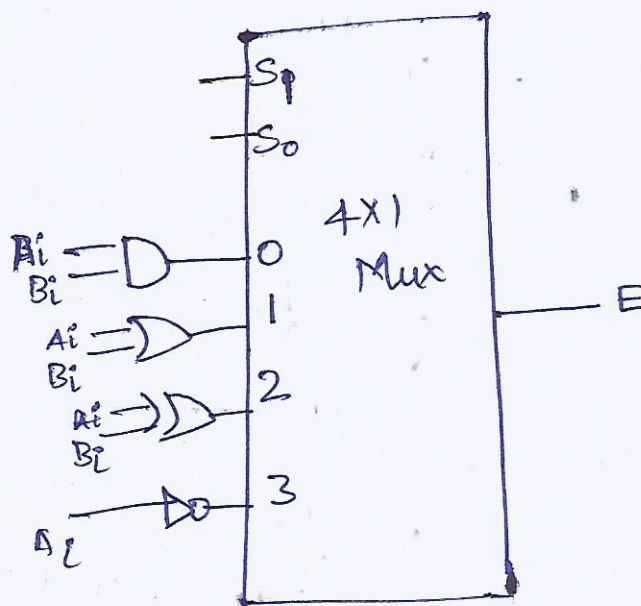
when $S_1=1$ and $S_0=1$ then
 Y input is $(B_3 B_2 B_1 B_0)' 1111$
 which is 2's complement
 of 0001 $\Rightarrow$ It makes

$$\begin{array}{r} A_3 A_2 A_1 A_0 \\ + 1 \ 1 \ 1 \ 1 \end{array} \Rightarrow \begin{array}{r} A \\ + 2's \text{ compl} \end{array}$$

$$\Rightarrow A + (-1)$$

$$\Rightarrow A - 1$$

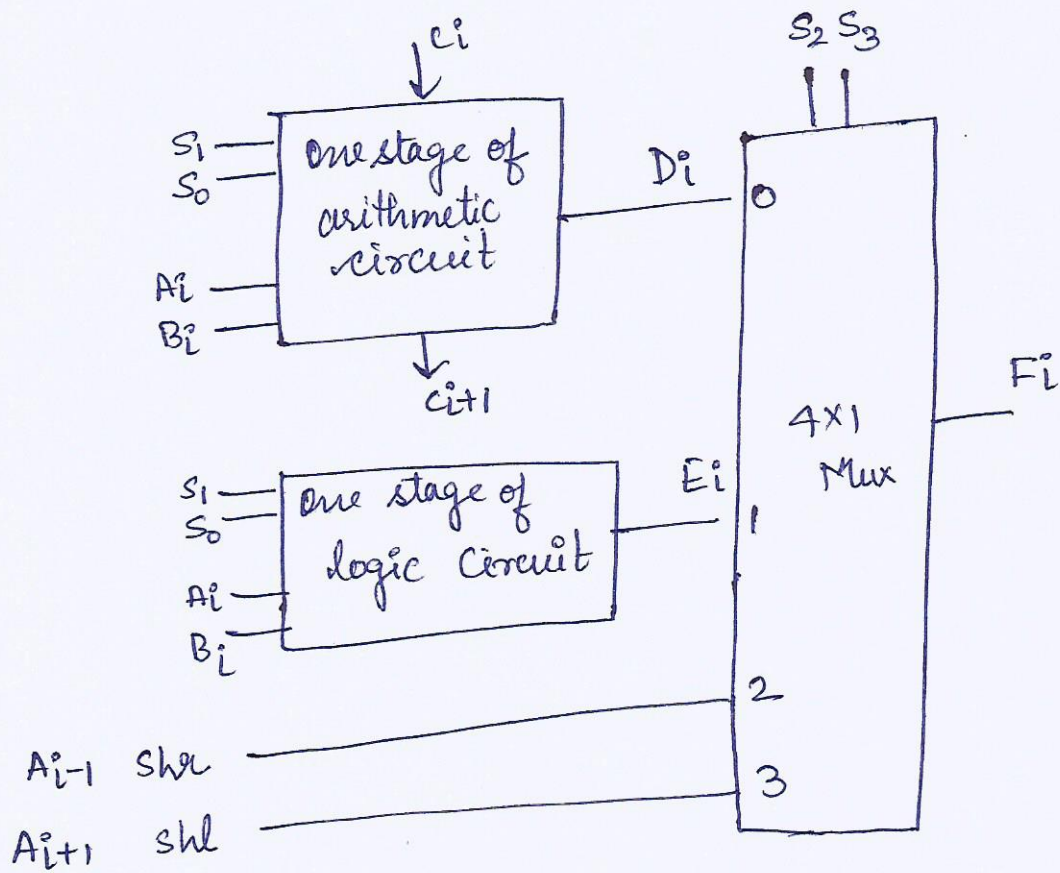
one stage of logic circuit:-



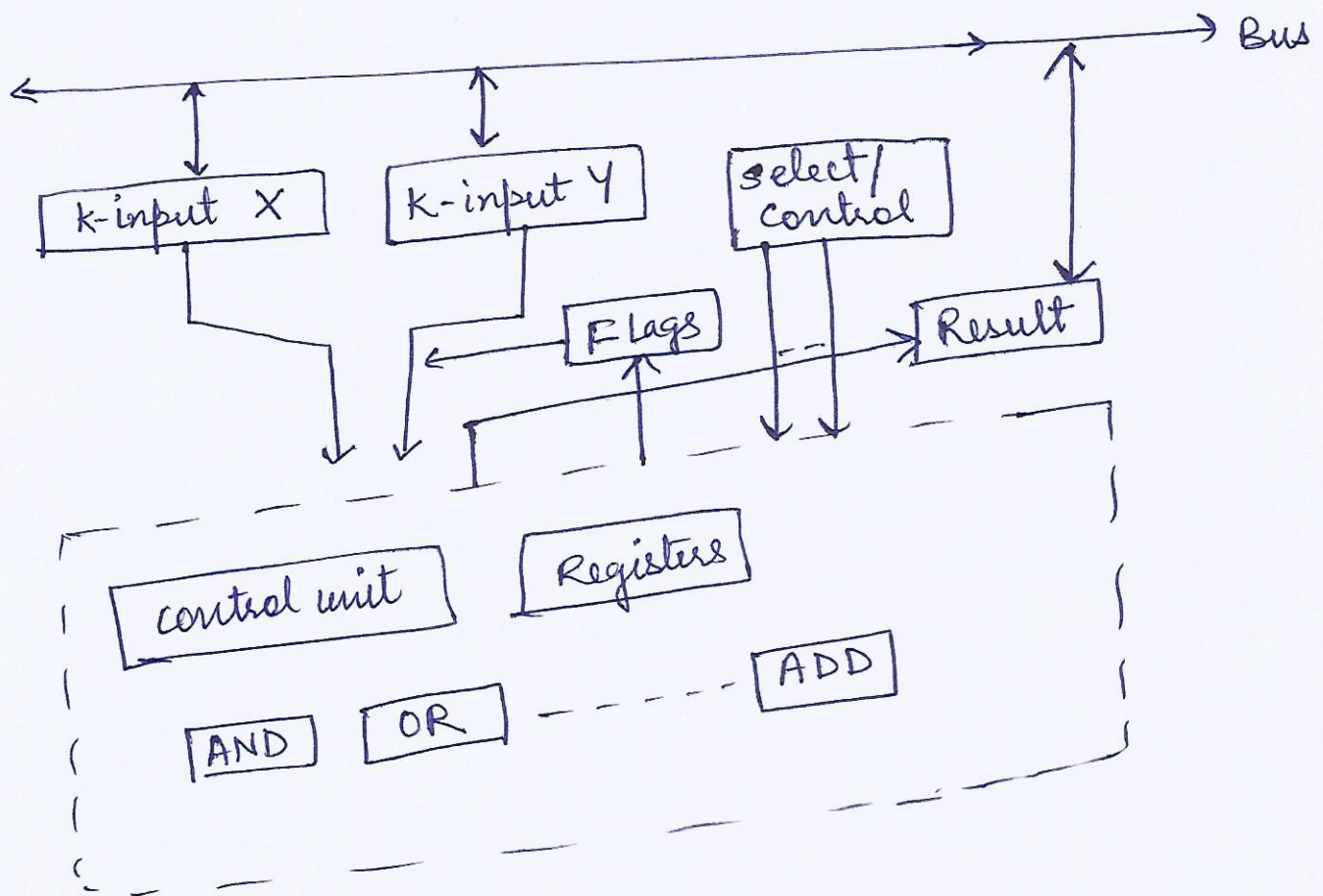
Function Table

S_1	S_0	Logic operation (E)
0	0	$A_i \text{ AND } B_i$
0	1	$A_i \text{ OR } B_i$
1	0	$A_i \text{ XOR } B_i$
1	1	$\text{NOT } A_i$

one stage of arithmetic logic shift unit



Sequential logic circuit based ALU



- Two Registers X and Y stores data or operands.
- select/control selects the appropriate arithmetic or logic operation which is performed over the data stored on register X and Y.
- After the execution of operation the result will be stored in Result register.
- After the execution of operation flags may be set such as carry, zero result, positive or negative result, overflow, division by zero etc.