Digital Logic Design

* Boolean Algebra / Switching Algebra

basic operations — + – Disjunction (OR)
. – Conjunction (AND)
' – negation

Logic Gates:

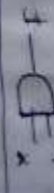
3 basic gates ← { AND (·), OR (+), NOT (')
XOR (⊕)

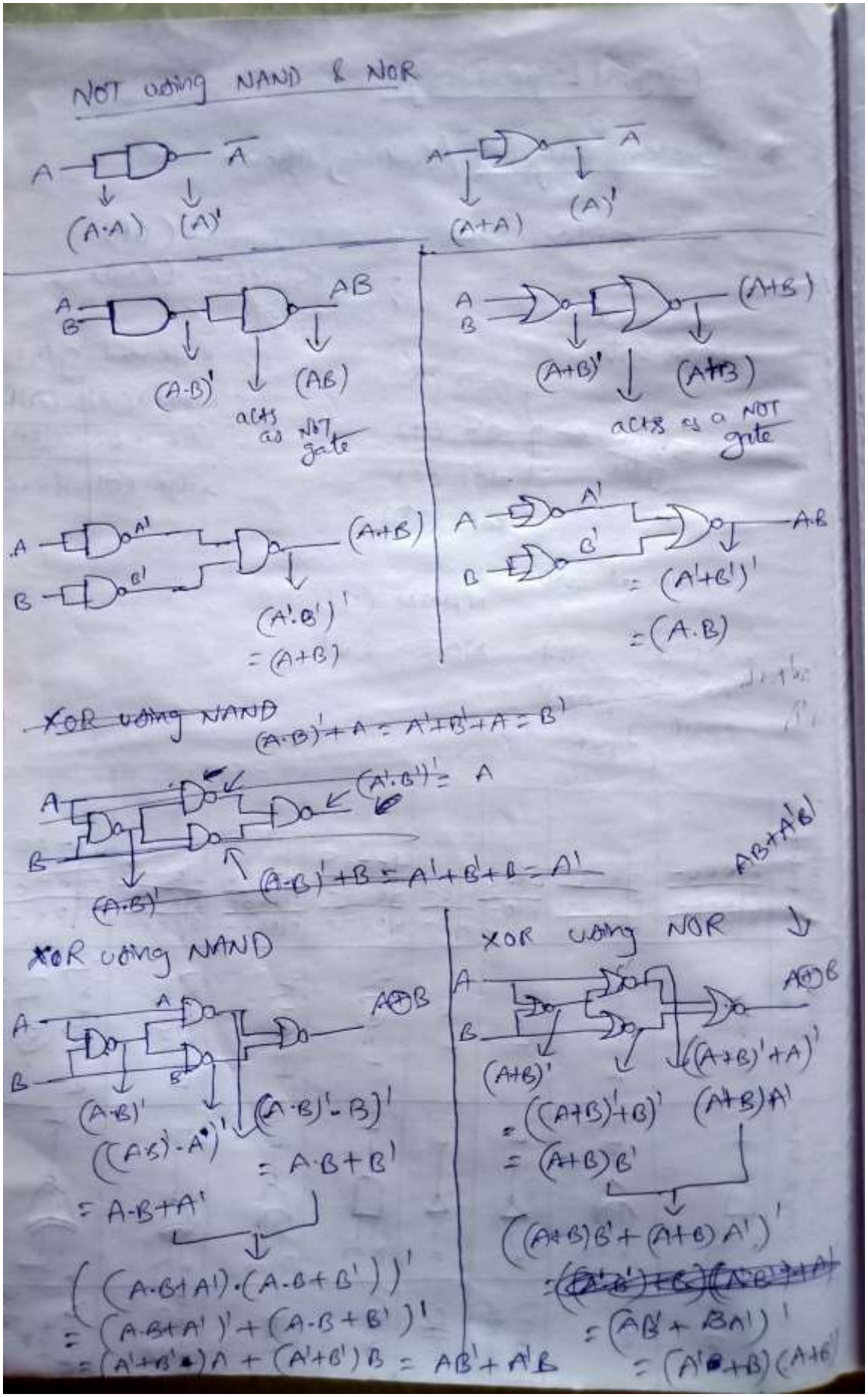
extended gates:

NAND (↑)	~(AND)
NOR (↓)	~(OR)
XNOR (⊙)	~(XOR)

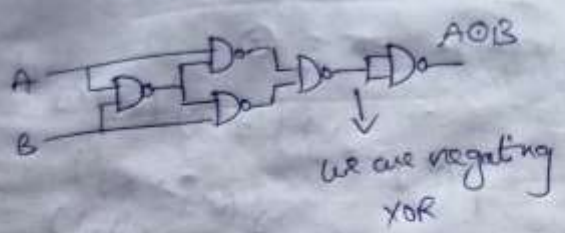
Universal Gates:
NAND (↑)
NOR (↓)

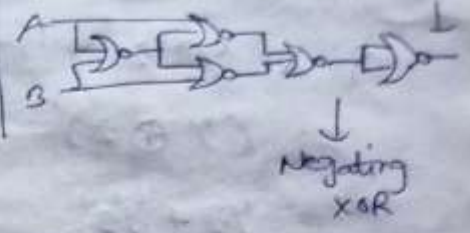
every gate can be created using NAND & NOR

Name	Graphical representation	Algebraic function	Truth table
AND		F = x·y	x y | F: 1 0 0 0, 1 0 – 0, 0 – 0 –, 0 0 – –
OR		F = x+y	x y | F: 0 – – –, 1 0 – 0, 0 0 – –
NOT / Inverter		F = x'	x | F: 1 0, 0 –
Buffer		F = x	x | F: 0 –, 1 0
NAND		F = (x·y)'	x y | F: – – – 0, 0 – 1 –, 1 0 – 1
NOR		F = (x+y)'	x y | F: 1 0 0 0, 0 – 1 –, 1 0 – 1
Exclusive-OR (XOR)		F = x'y + xy' = x⊕y	x y | F: 0 – – 0, 0 – 1 –, 1 0 – 1
Exclusive-NOR or equivalence		F = xy + x'y' = (x⊕y)' = x⊙y	x y | F: 1 0 0 1, 0 – 1 –, 1 0 – 1


NOT using NAND & NOR
A
$\bar{A}$
(A·A)
(A)'
A
$\bar{A}$
(A+A)
(A)'
A
B
AB
(A·B)'
(AB)
acts as NOT gate
A
B
(A+B)
(A+B)'
(A+B)
acts as a NOT gate
A
A'
B
B'
(A+B)
(A'.B')'
=(A+B)
A
A'
B
B'
A·B
=(A'+B')'
=(A.B)
XOR using NAND
(A·B)'+A = A'+B'+A = B'
A
B
(A'.B')' = A
(A·B)'
(A·B)'+B = A'+B'+B = A'
AB+A'B'
XOR using NAND
A
B
A⊕B
(A·B)'
((A·B)'·B)'
= A·B+B'
= A·B+A'
((A·B+A')·(A·B+B'))'
=(A·B+A')'+(A·B+B')'
=(A'+B')A + (A'+B')B = AB'+A'B
XOR using NOR
A
B
A⊕B
(A+B)'
((A+B)'+A)'
=((A+B)'+B)'
=(A+B)B'
(A+B)A'
((A+B)B'+(A+B)A')'
=(AB'+BA')'
=(A'B+B)(A+B')

XNOR gate:





1) which of the following doesnot represent exclusive NOR of x and y'

A) $xy + x'y'$ (True)

B) $x \oplus y'$ (True) $xy + x'y'$

C) $x' \oplus y$ (True) $x'y' + xy$

(D) $x' \oplus y'$ (False) ✓

2) The truth table

x	y	f(x,y)
0	0	0
0	1	0
1	0	1
1	1	1

represents

A) x ✓

B) x+y

C) $x \oplus y$

D) y

3) Define Connective * for the Boolean variables x and y as $x * y = xy + x'y'$ $*=\odot$ assume

let $z = x * y$

P: $x = y * z$

Q: $y = x * z$

R: $(x * y) * z = 1$ = $z * z = 1$ ✓ (True)

$y = x \odot z$, $(x+x') + x' \cdot x'$

$= (x \oplus z)'$ $= ((y \odot z) \oplus (x \odot z))'$

$= (x'z + xz')'$

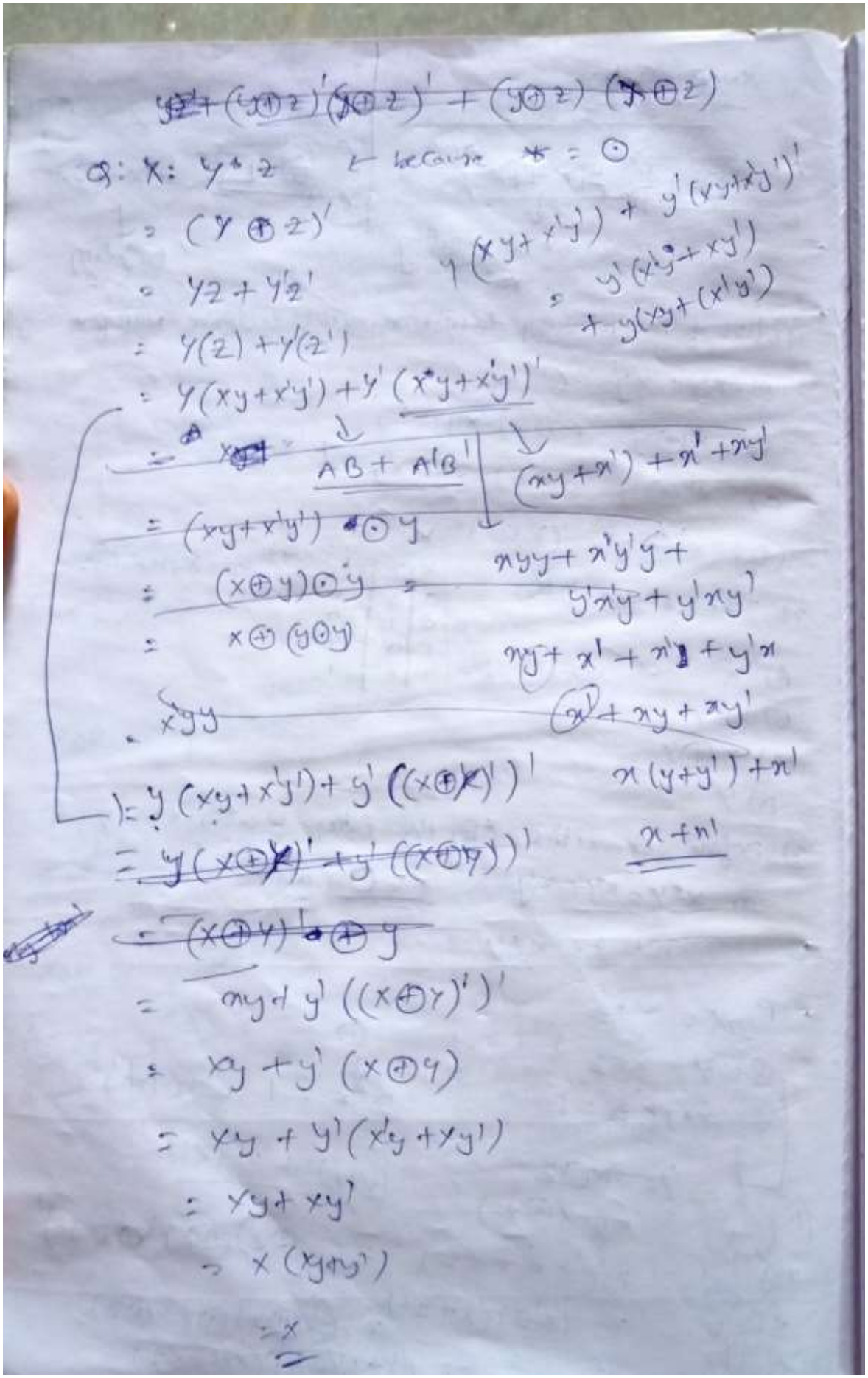
$= (x+z)(x'+z')$ $= (y \oplus z) \odot (x \oplus z)$

$= (yz' + y'z) \odot (xz' + x'z)$

(1) $xy * y'$

$x'y * y'$

(y)


Q: X: y * z ← because * = ⊙
= (y ⊙ z)'
= yz + y'z'
= y(z) + y'(z')
= y(xy + x'y') + y'(x'y + x'y')'
= y(xy + x'y') + y'(x'y + xy')
= y'(x'y + xy') + y(xy + (x'y')
AB + A'B'
(xy + x') + x' + xy'
= (xy + x'y') ⊙ y
= (x ⊙ y) ⊙ y
= x ⊙ (y ⊙ y)
xyy + x'y'y + y'xy + y'xy'
xy + x' + x' + y'x
x' + xy + xy'
x(y + y') + x'
x + x'
= y(xy + x'y') + y'((x ⊙ x)')'
= y(x ⊙ x)' + y'((x ⊙ y))'
= (x ⊙ y)' ⊙ y
= xy + y'((x ⊙ y)')'
= xy + y'(x ⊙ y)
= xy + y'(x'y + xy')
= xy + xy'
= x(y + y')
= x

$y = x * z$

$= x \odot (x \odot y)$

$= x(xy + x'y') + (x'y' + x'y')x'$

$= xy + x'y$

$= y(1) = y$

$x * y * z$

$y * z = x$

So $x * x$

$x * x =$

$x'x' + xx$

$= 1$

Properties of Boolean Algebra

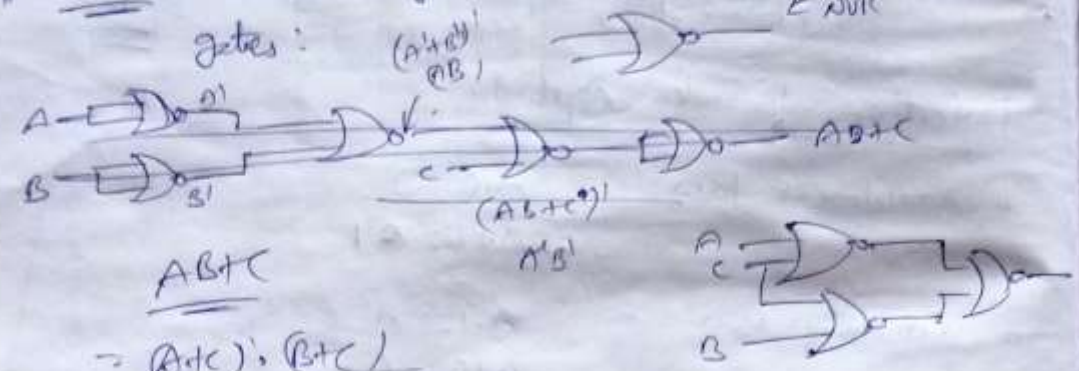
* Annulment law:- $A \cdot 0 = 0$, $A + 1 = 1$
* Identity law :- $A \cdot 1 = A$, $A + 0 = A$
* Idempotent law :- $A + A = A$, $A \cdot A = A$
* Complement law = $A + A' = 1$, $A \cdot A' = 0$
* Double negation law = $(A')' = A$
* Commutative law = $A + B = B + A$, $A \cdot B = B \cdot A$
* Associative law = $A + (B + C) = (A + B) + C$, $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
* Distributive law = $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$, $A + (B \cdot C) = (A + B) \cdot (A + C)$
* Absorption law = $A \cdot (A + B) = A$, $A + AB = A$ $= A(1 + B) = A(1) = A$

* Demorgan's law :

$(A.B)' = A'+B'$

$(A+B)' = A'.B'$

Q: AB+C minimum gates if we use only 2-input NOR gates:



AB+C

$= (A+C)'.(B+C)'$

$= ((A+C)'+(B+C)')'$ - 3 gates

Sum of minterms or Product of maxterms

Minimization:

↳ Reduces Cost Complexity

Minimization using Algebraic Manipulation:

upto 4 or 5 variables (better)

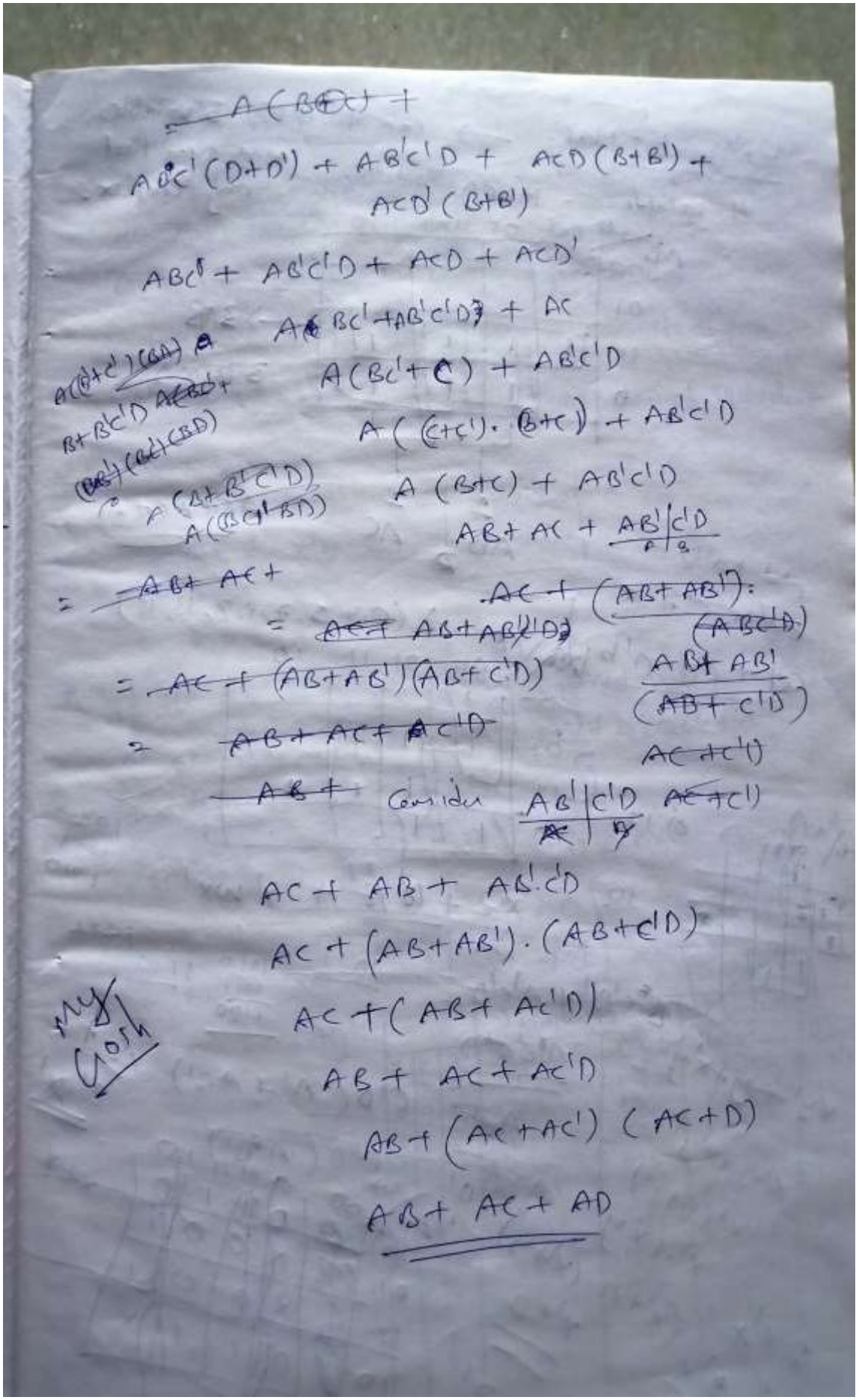
laws:

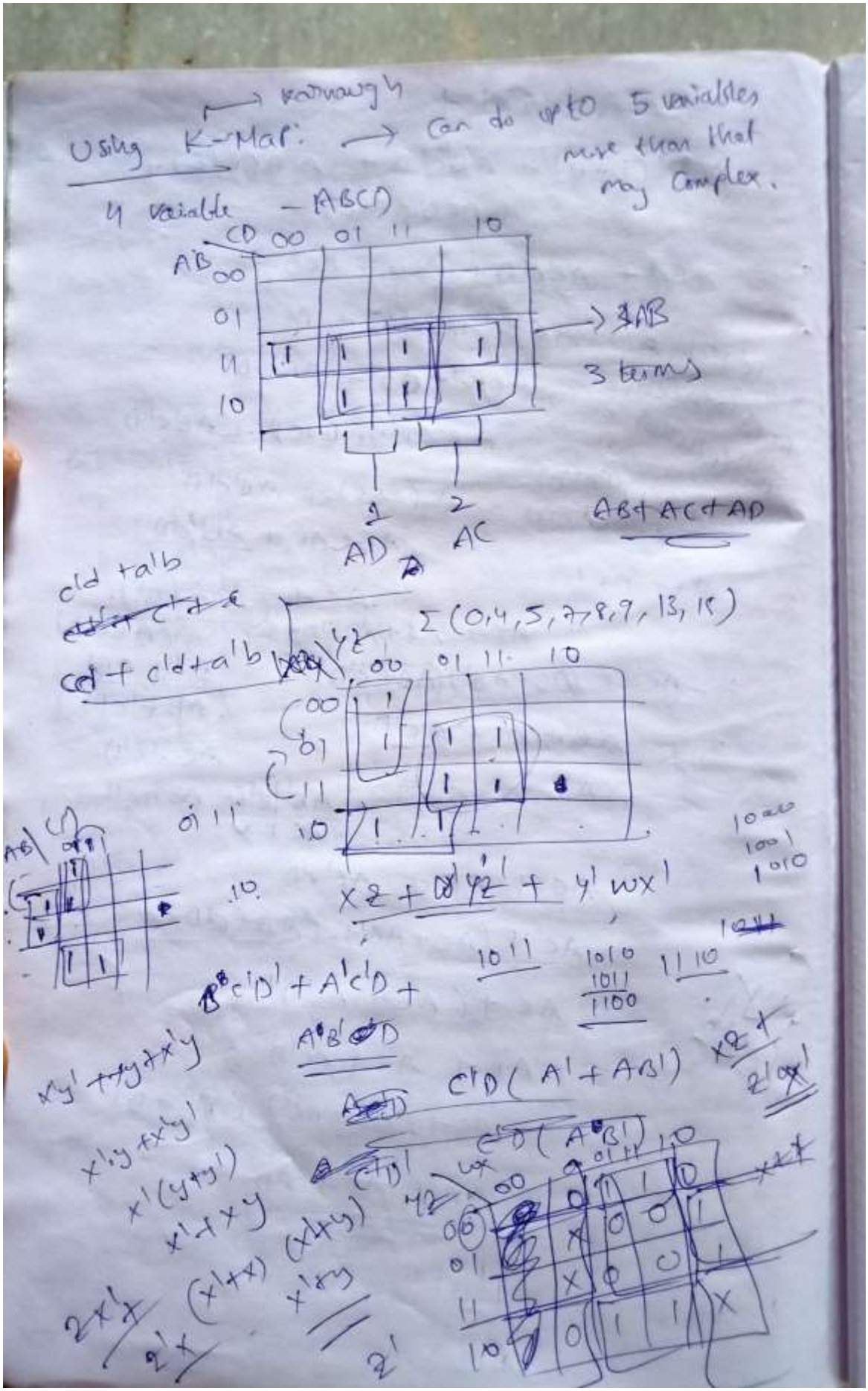
1. $A+A' = 1$
2. $A+A'B = (A+A')(A+B) = A+B$
3. $A+AB = A(1+B) = A$

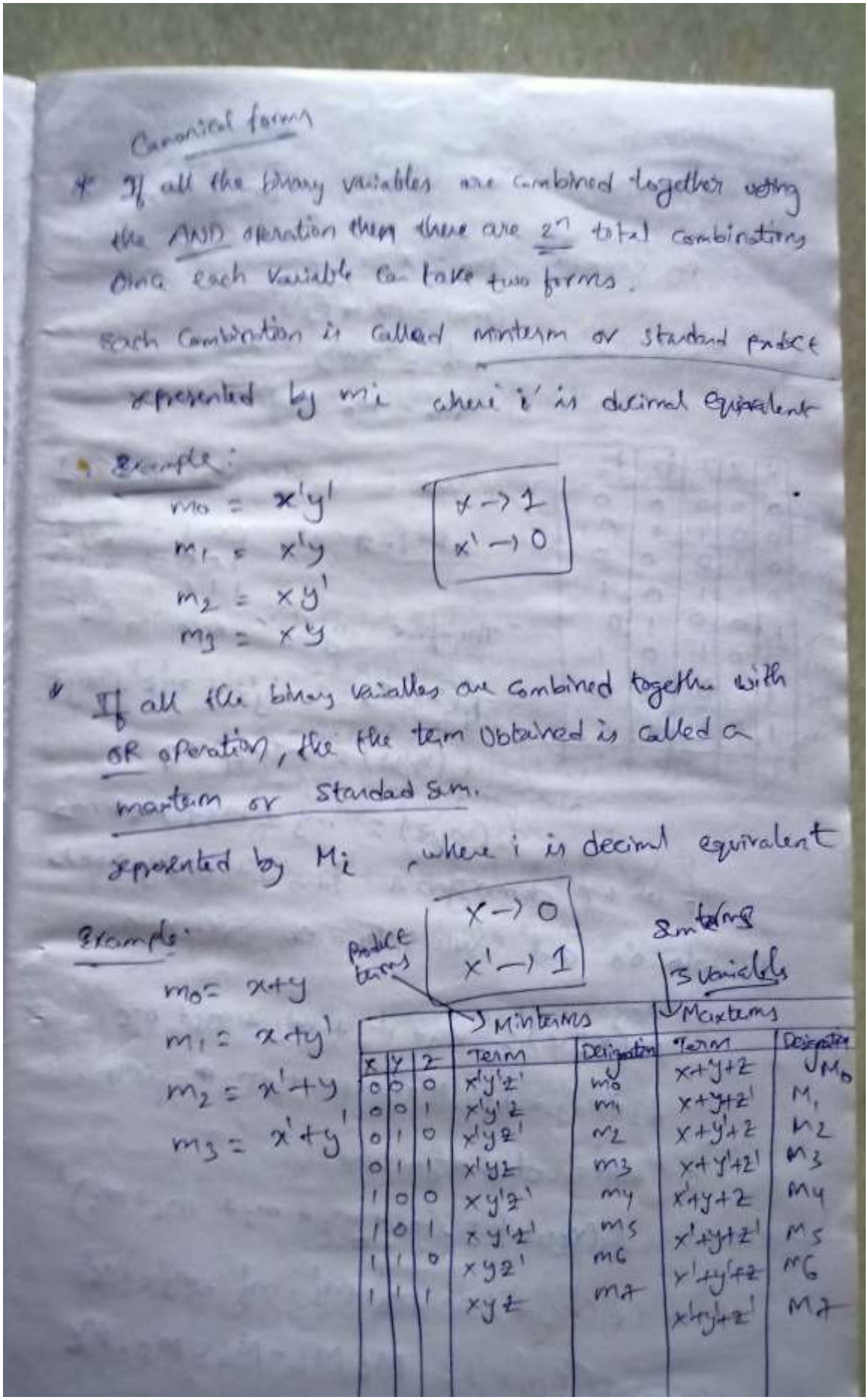
$F = ABC'D' + ABC'D + AB'C'D + ABCD + AB'CD + ABCD' + AB'CD'$

$= ABC(D+D') + AB'C(D+D') + ABC'(D'+D) + AB'C'D$

$= ABC + AB'C + ABC' + AB'C'D$


A(BC')+
ABC'(D+D')+AB'C'D+ACD(B+B')+
ACD'(B+B')
ABC'+AB'C'D+ACD+ACD'
ABC'+AB'C'D+AC
A(BC'+AB'C'D)+AC
A(BC'+C)+AB'C'D
A((C+C').(B+C))+AB'C'D
A(B+C)+AB'C'D
AB+AC+AB'C'D
AB+AC+(AB+AB').(AB+C'D)
AC+(AB+AB').(AB+C'D)
AB+AC+AB.C'D
Consider AB'|C'D
AC+AB+AB'.C'D
AC+(AB+AB').(AB+C'D)
AC+(AB+AC'D)
AB+AC+AC'D
AB+(AC+AC').(AC+D)
AB+AC+AD
My Work


Using K-Map → thorough
→ Can do upto 5 variables more than that may complex.
4 variable – ABCD
CD 00 01 11 10
AB 00 01 11 10
→ AB
3 terms
2 AD
2 AC
AB + AC + AD
cd + a'b
cd + c'd + a'b
Σ(0,4,5,7,8,9,13,15)
xz + w'y'z' + y'wx'
A'c'D' + A'c'D +
A'B'D
C'D(A' + AB')
xy' + xy + x'y
x'(y+y')
x'+xy
(x'+x)
(x'+y)
x'+y
2x'
2'x
2'
xz +
z'x'



Canonical forms

* If all the binary variables are combined together using the AND operation then there are 2^n total combinations since each variable can take two forms.

Each combination is called minterm or standard product represented by m_i where 'i' is decimal equivalent

Example:

$m_0 = x'y'$
$m_1 = x'y$
$m_2 = xy'$
$m_3 = xy$

$x \to 1$
$x' \to 0$

* If all the binary variables are combined together with OR operation, the the term obtained is called a maxterm or standard sum.

represented by M_i where i is decimal equivalent

Example:

$M_0 = x+y$
$M_1 = x+y'$
$M_2 = x'+y$
$M_3 = x'+y'$

$x \to 0$
$x' \to 1$

Product terms → Minterms
Sum terms → Maxterms
3 variables

x	y	z	Minterm	Designation	Maxterm	Designation
0	0	0	$x'y'z'$	m_0	$x+y+z$	M_0
0	0	1	$x'y'z$	m_1	$x+y+z'$	M_1
0	1	0	$x'yz'$	m_2	$x+y'+z$	M_2
0	1	1	$x'yz$	m_3	$x+y'+z'$	M_3
1	0	0	$xy'z'$	m_4	$x'+y+z$	M_4
1	0	1	$xy'z$	m_5	$x'+y+z'$	M_5
1	1	0	xyz'	m_6	$x'+y'+z$	M_6
1	1	1	xyz	m_7	$x'+y'+z'$	M_7

Relation b/w minterm & maxterm

In general $m_i = (M_i)'$ (or) $M_i = (m_i)'$

for example $m_0 = x'y'$

$(m_0)' = (x'y')'$

$m_0' = x+y$

$m_0' = M_0$

	x	y	z	f_1	f_2
m_0	0	0	0	0	0
m_1	0	0	1	1	0
m_2	0	1	0	0	0
m_3	0	1	1	0	1
m_4	1	0	0	1	0
m_5	1	0	1	0	1
m_6	1	1	0	0	1
m_7	1	1	1	1	1

Take function value which produces 1

f_1 is 1 at 001 → $x'y'z$ (minterms)

100 → $xy'z'$

111 → xyz

Therefore algebraic expression

$f_1(x,y,z) = x'y'z + xy'z' + xyz$

$f_1(x,y,z) = m_1 + m_2 + m_7$

$f_2(x,y,z) = m_3 + m_5 + m_6 + m_7$

If we use demorgan's law it simply inverts 1's into 0's and AND into OR vice versa.

$f_1(x,y,z)' = m_0 + m_2 + m_3 + m_5 + m_6$

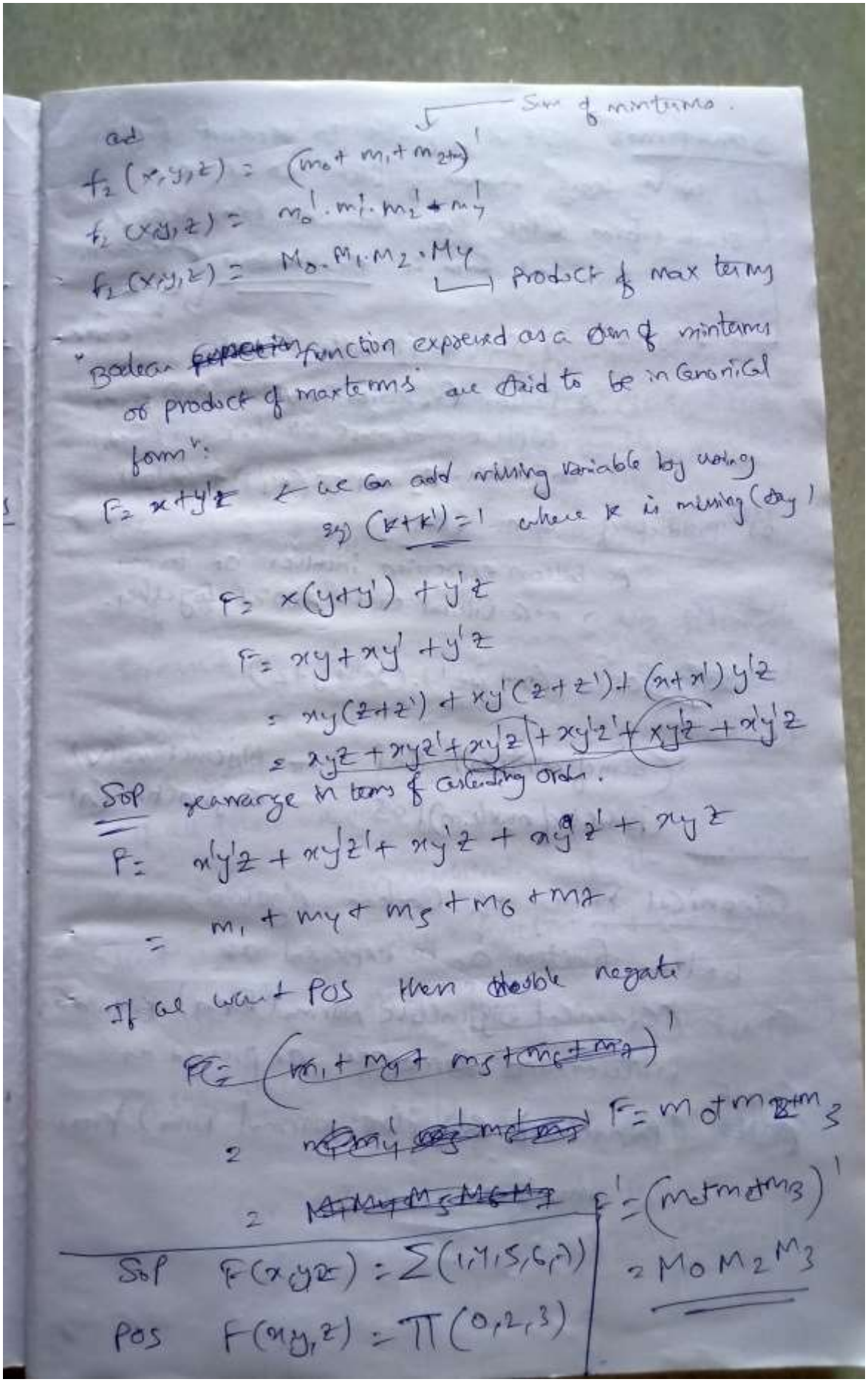
$f_2(x,y,z)' = m_0 + m_1 + m_2 + m_4$

on using demorgan law:-

$f_1(x,y,z) = (m_0 + m_2 + m_3 + m_5 + m_6)'$

$f_1(x,y,z) = m_0' \cdot m_2' \cdot m_3' \cdot m_5' \cdot m_6'$

$f_1(x,y,z) = M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6$



Sum of minterms.

ad

$f_2(x,y,z) = (m_0 + m_1 + m_2 + m_4)'$

$f_2(x,y,z) = m_0' \cdot m_1' \cdot m_2' \cdot m_4'$

$f_2(x,y,z) = M_0 \cdot M_1 \cdot M_2 \cdot M_4$ — Product of max terms

"Boolean ~~expression~~ function expressed as a Sum of minterms or product of maxterms are said to be in Canonical form":

$F = x + y'z$ ← we can add missing variable by using eg) $(k + k') = 1$ where k is missing (say)

$F = x(y + y') + y'z$

$F = xy + xy' + y'z$

$= xy(z + z') + xy'(z + z') + (x + x')y'z$

$= xyz + xyz' + xy'z + xy'z' + xy'z + x'y'z$

SOP rearrange M terms of ascending order.

$F = x'y'z + xy'z' + xy'z + xyz' + xyz$

$= m_1 + m_4 + m_5 + m_6 + m_7$

If we want POS then double negate

$F = (M_0 + M_2 + M_3)'$

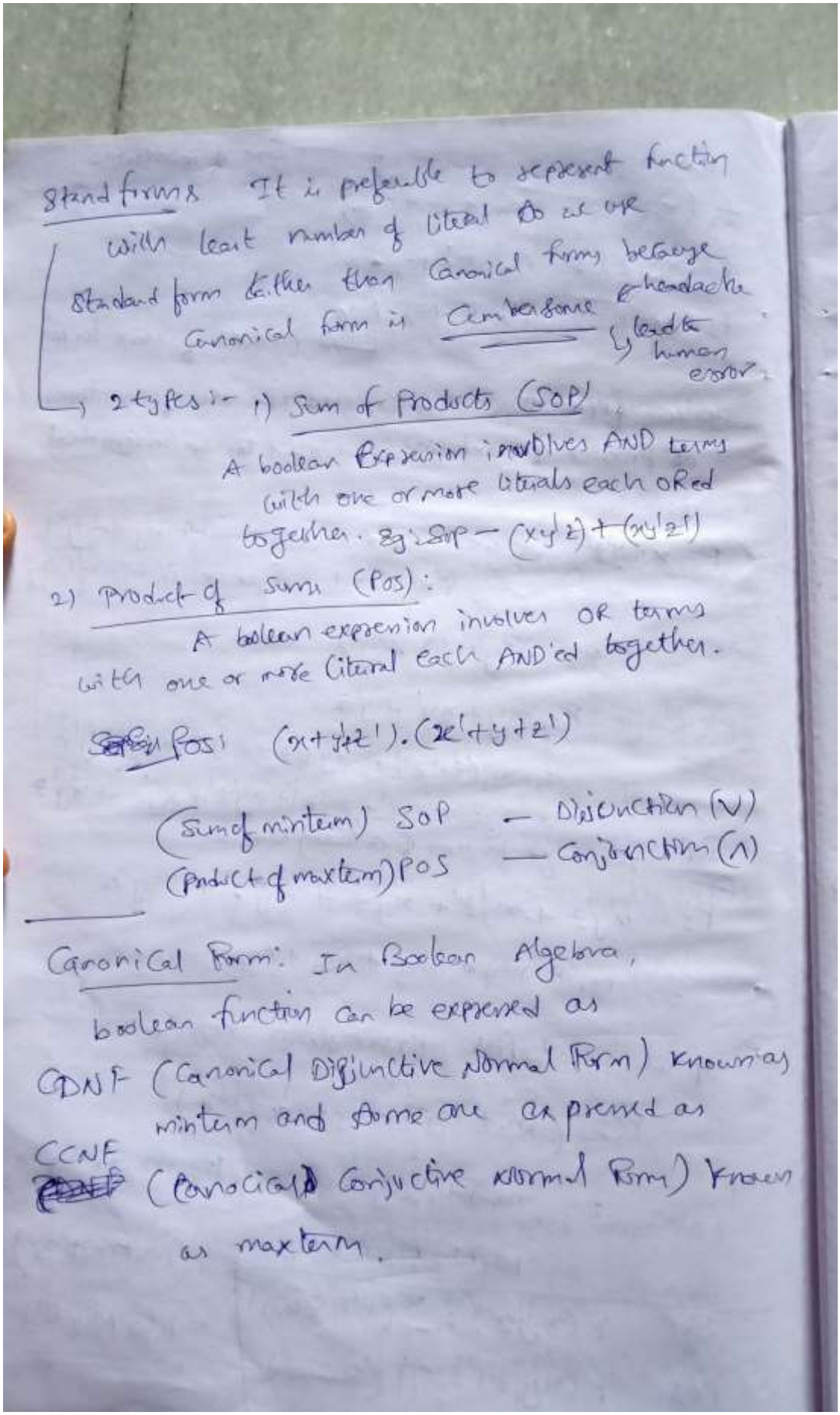
$F = m_0 m_2 m_3$

$F' = (m_0 m_2 m_3)'$

$= M_0 M_2 M_3$

SOP $F(x,y,z) = \sum(1,4,5,6,7)$

POS $F(x,y,z) = \Pi(0,2,3)$



Standard forms: It is preferable to represent function with least number of literal so we use standard form rather than Canonical form because Canonical form is cumbersome & headache (leads to human error).

2 types :- 1) Sum of Products (SOP)

A boolean Expression involves AND terms with one or more literals each ORed together. eg: SOP — $(xy'z) + (xy'z')$

2) Product of Sums (POS):

A boolean expression involves OR terms with one or more literal each AND'ed together.

POS: $(x+y+z').(x'+y+z')$

(Sum of minterm) SOP — Disjunction ($\vee$)
(Product of maxterm) POS — Conjunction ($\wedge$)

Canonical Form: In Boolean Algebra, boolean function can be expressed as CDNF (Canonical Disjunctive Normal Form) known as minterm and some are expressed as CCNF (Canonical Conjuctive Normal Form) known as maxterm.

In minterm we look for function where the output results in "1".
while in maxterm we look for function where the output is "0"

Standard forms

A Boolean variable can be expressed in either true form or (Complemented) form. In Standard form Boolean functions will contain all the variables in either true form or Complimented form. while in Canonical number of variables depends on the output

of Pos or Sop.

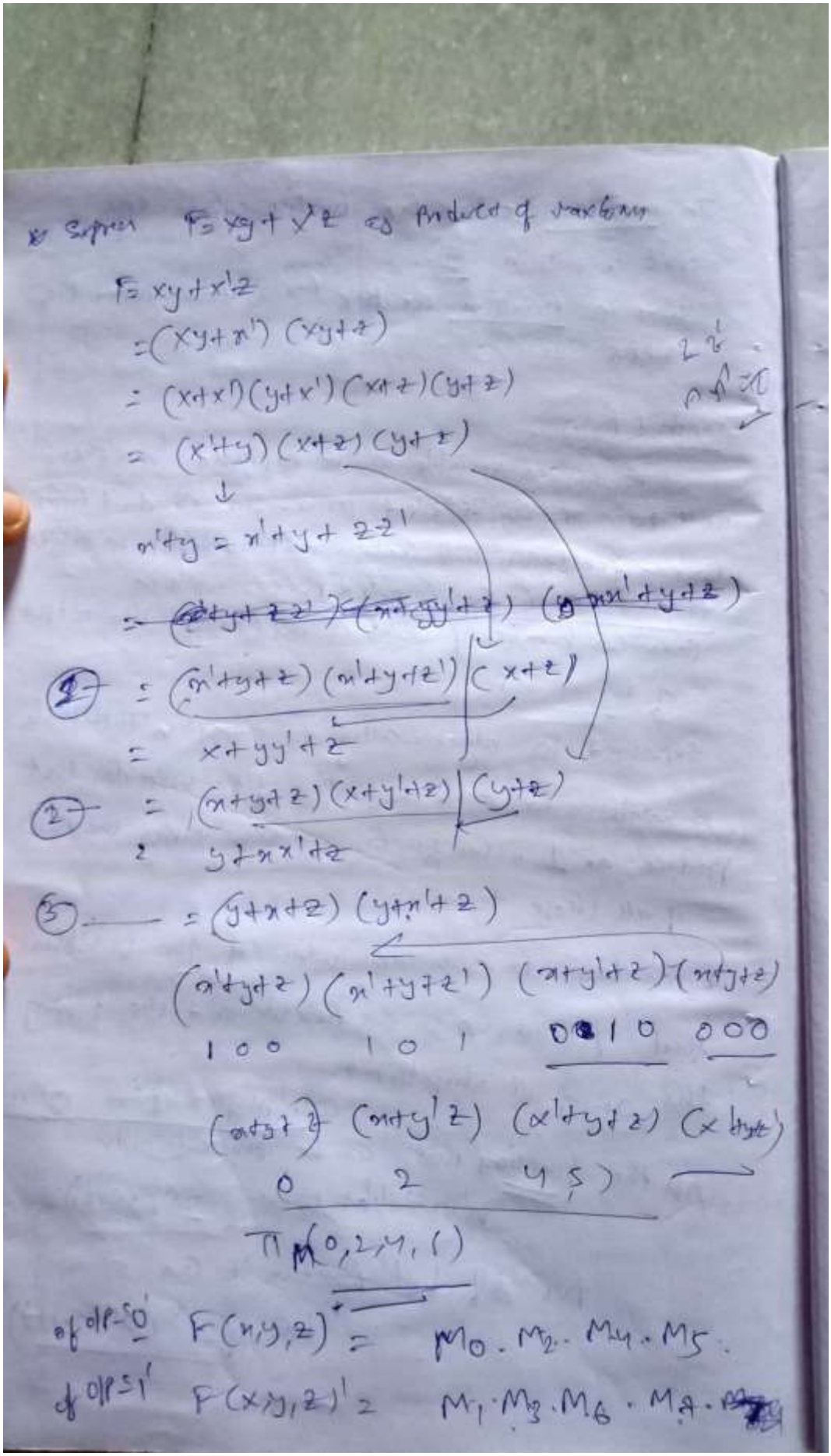
Expressed in algebraically form a given TruthTable

* minterm for each combination of the variables that produce as 1 in the function and then taking the AND of all those terms.

* Maxterms to each combination of the variables that produces 0 in the function and the taking the OR of all those terms.

All the functions that can be formed with n variables to be $(2^{(2^n)})$ in single function (2^n)

for Self-dual functions it can be $(2^{(2^{(n-1)})})$


Express F = xy + x'z as Product of maxterms
F = xy + x'z
= (xy + x')(xy + z)
= (x + x')(y + x')(x + z)(y + z)
= (x' + y)(x + z)(y + z)
x' + y = x' + y + zz'
= (x' + y + z)(x + y + z')(x' + y + z)
= (x' + y + z)(x' + y + z')(x + z)
= x + yy' + z
= (x + y + z)(x + y' + z)(y + z)
= y + zx'
= (y + x + z)(y + x' + z)
(x' + y + z)(x' + y + z')(x + y' + z)(x + y + z)
100 101 010 000
(x + y + z)(x + y' + z)(x' + y + z)(x + y + z')
0 2 4 5
ΠM(0, 2, 4, 5)
of o/p = 0' F(x, y, z) = M0 . M2 . M4 . M5
of o/p = 1' F(x, y, z)' = M1 . M3 . M6 . M7

$F(x,y,z) = (M_1, M_3, M_6, M_7)$

SOP $\sum m_1 + m_3 + m_6 + m_7$

Eg: $F = y' + xz' + xyz$

$= (x+x')y' + x(y+y')z' + xyz$

$= xy' + x'y' + xyz' + xy'z' + xyz$

$= xy'z + xy'z' + x'y'z' + x'y'z + xyz' + xy'z' + xyz$

Dual of a Boolean expression means interchanging 0's and 1's means changing + into . and . into + and 0's and 1's

Not as Demorgans

Principal of Duality: Any theorem or identity in Boolean algebra remains true if 0 and 1 are swapped and . and + are swapped throughout.

2 properties for self-dual functions:-

1) It is neutral (no. of minterms = no. of maxterms)

2) A single function does not contain two mutually exclusive terms.

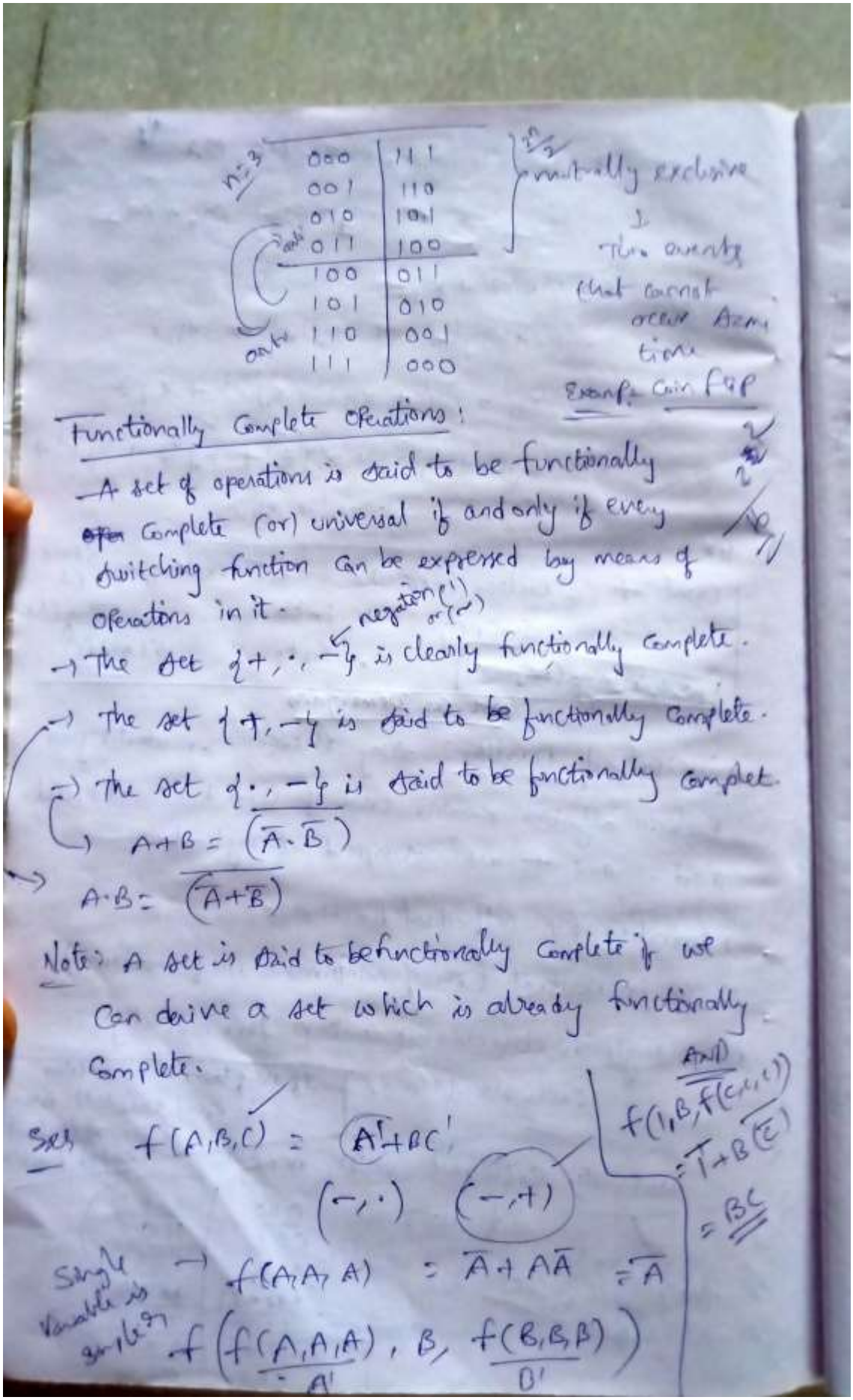
If no. of terms in a function is equal to (2^{n-1}) then it is a neutral function

Mutually exclusive pairs means.

if $n=3$

they cannot occur at same time

000	111
001	110
010	101
011	100
100	011
101	010
110	001
111	000



n=3	
000	111
001	110
010	101
011	100
100	011
101	010
110	001
111	000

mutually exclusive

Two events that cannot occur At a time

Example: Coin Flip

Functionally Complete Operations:

A set of operations is said to be functionally Complete (or) universal if and only if every switching function can be expressed by means of operations in it.

→ The set {+, ·, −} (negation (') or (−)) is clearly functionally complete.

→ The set {+, −} is said to be functionally Complete.

→ The set {·, −} is said to be functionally complet.

↳ $A+B = (\overline{\bar{A}\cdot\bar{B}})$

→ $A\cdot B = \overline{(\bar{A}+\bar{B})}$

Note: A set is said to be functionally Complete if we can derive a set which is already functionally Complete.

Sol: $f(A,B,C) = A'+BC'$

$(-,\cdot)$ $(-,+)$

AND

$f(1,B,f(C,C)) = \bar{1}+B(\bar{C}) = BC$

Single variable is similar → $f(A,A,A) = \bar{A}+A\bar{A} = \bar{A}$

$f\left(f(A,A,A), B, f(B,B,B)\right)$

A' $0'$

$= (A')' + B(B')' = A + BB = A + B$

Eg: $f(A,B) = \overline{A} + B$

$f(A,A) = \overline{A} + A = 1$

$f(B,B) = \overline{B} + B = 1$

Let's take support of '0'

$f(A,0) = \overline{A} + 0 = \overline{A}$

to prove $A \cdot B$

$f(f(A,0), B) = \overline{(\overline{A})} + B = A \cdot B$

without taking (f,0) additional zero negation/complement cannot be achieved so it is partially functional complete

(1+, ·, −)

Function can be

Functional Complete (fun)

Partial (1+, ·, −) functional Complete

Not Functional Complete.

Eg: $f(A,B) = \overline{A}B$

$f(A,A) = \overline{A}A = 0$

$f(B,B) = \overline{B}B = 0$

f is Partially Functionally Complete

(f,1) is functionally Complete

$f(A,1) = \overline{A}(1) = \overline{A}$

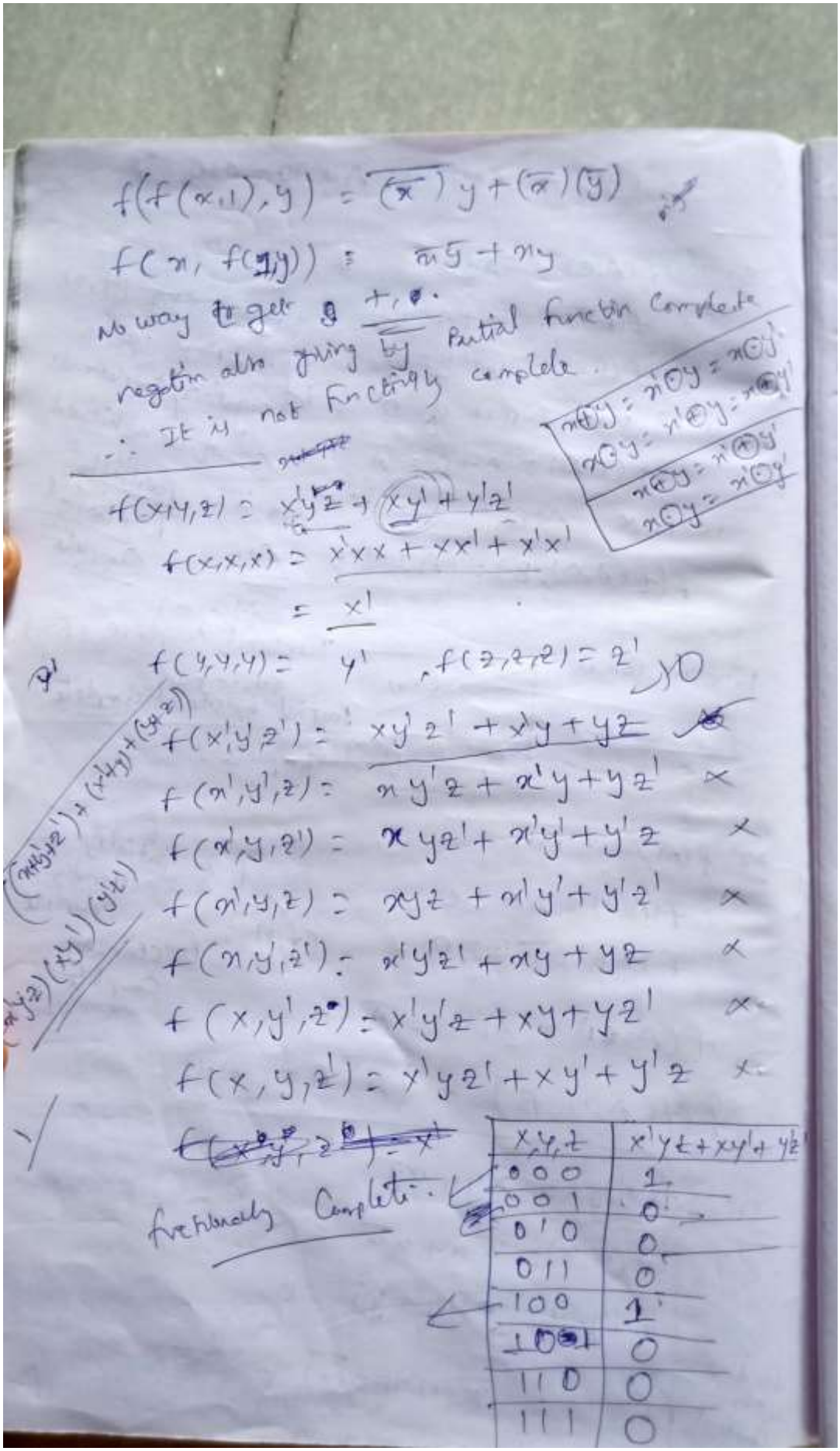
$f(f(A,1), B) = \overline{(\overline{A})}B = A \cdot B$

Eg: $f(x,y) = \overline{x}y + x\overline{y}$

$f(x,x) = \overline{x}x + x\overline{x} = 0$

$f(y,y) = \overline{y}y + y\overline{y} = 0$

feeding with Constant → $f(x,1) = \overline{x}(1) + x(\overline{1}) = \overline{x}$



$f(f(x,1),y) = \overline{(x)}\,y + (\overline{x})(\overline{y})$

$f(x, f(y,y)) = \overline{x}\,\overline{y} + xy$

No way to get 0 & 1.

negation also going by Partial function Complete

∴ It is not Functionally complete.

$x \oplus y = x' \oplus y = x \oplus y'$

$x \odot y = x' \odot y = x \odot y'$

$x \oplus y = x' \odot y'$

$x \odot y = x' \odot y'$

$f(x,y,z) = x'y'z + xy' + y'z'$

$f(x,x,x) = x'xx + xx' + x'x'$

$= x'$

$f(y,y,y) = y' \quad , f(z,z,z) = z'$

$f(x',y',z') = xy'z' + x'y + yz$

$f(x',y',z) = xy'z + x'y + yz'$ ×

$f(x',y,z') = xyz' + x'y' + y'z$ ×

$f(x',y,z) = xyz + x'y' + y'z'$ ×

$f(x,y',z') = x'y'z' + xy + yz$ ×

$f(x,y',z) = x'y'z + xy + yz'$ ×

$f(x,y,z') = x'yz' + xy' + y'z$ ×

x,y,z	$x'yz + xy'z + yz'$
000	1
001	0
010	0
011	0
100	1
101	0
110	0
111	0

Functionally Complete.

Properties for post's Functional Completeness theorem:

1. T_0 - class of all 0-preserving functions, such as
($f \in T_0$) $f(0,0\ldots,0) = 0$.

2. T_1 = class of all 1-preserving functions, such as
($f \in T_1$) $f(1,1,\ldots,1) = 1$.

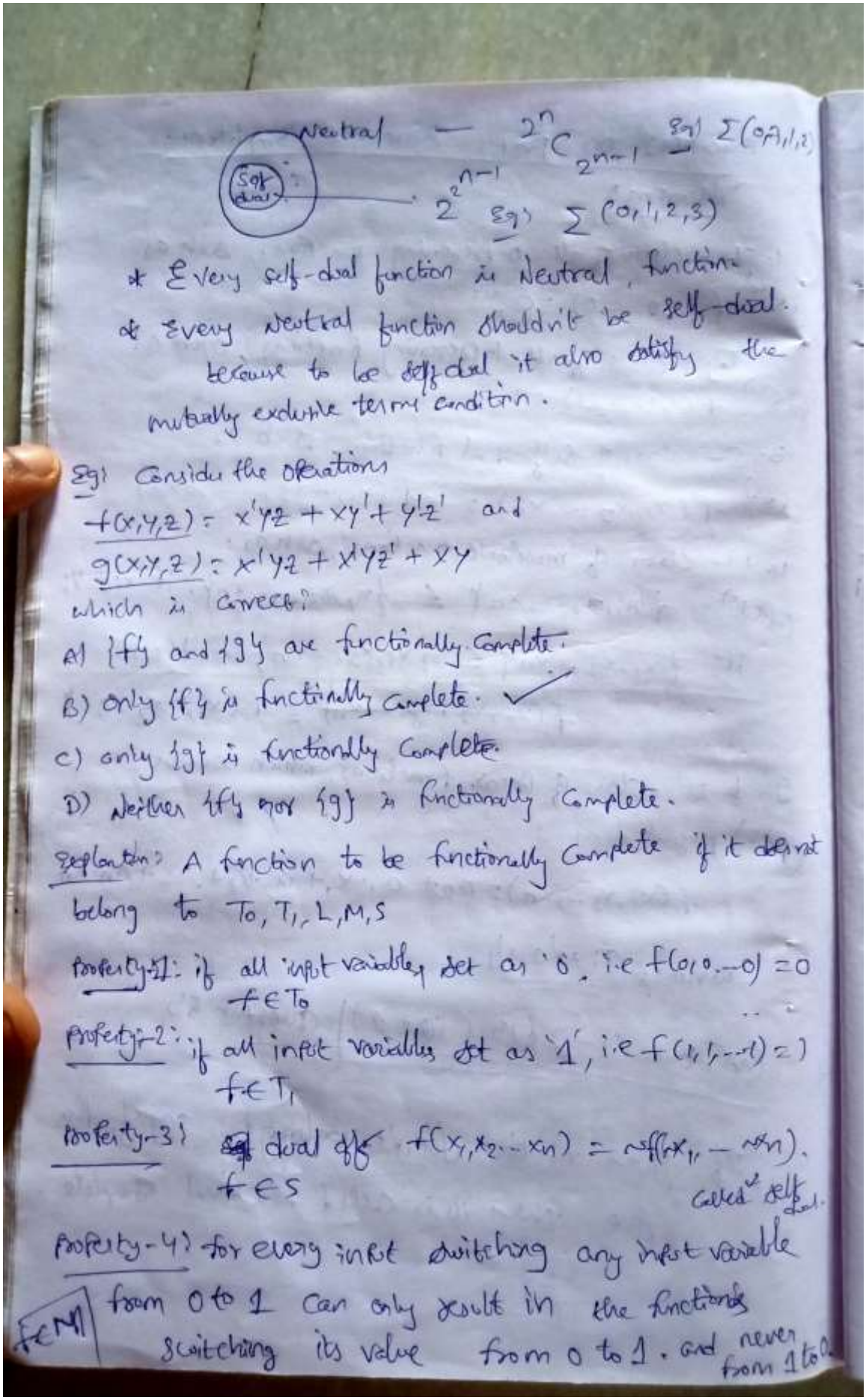
3. S - class of self-dual functions, such as
($f \in S$) $f(x_1, x_2 \ldots x_n) = \neg f(\neg x_1, \neg x_2, \ldots \neg x_n)$

4. M - class of monotonic functions, such as:
($f \in M$) $\{x_1, x_2 \ldots, x_n\} \leq \{x_1, x_2, \ldots, x_n\}$, if $x_i \leq y_i$
if $\{x_1, x_2 \ldots x_n\} \leq \{x_1, x_2, \ldots x_n\}$ then
$f(x_1, x_2, x_3 \ldots x_n) \leq f(x_1, x_2 \ldots x_n)$

5. L - class of linear functions, which can be
($f \in L$) presented as:
$f(x_1, x_2 \ldots, x_n) = a_0 + a_1 \cdot x_1 + a_2 \cdot x_2 + \ldots + a_n \cdot x_n$
where $a_i = \{0, 1\}$

GATE - CS - 2015 (Set 1) | question 65

If given function doesn't belong to any of the five then it is called functional Complete.



* Every self-dual function is Neutral function.
* Every Neutral function should'nt be self-dual.
because to be self dual it also satisfy the mutually exclusive terms condition.

Eg: Consider the operations
$f(x,y,z) = x'yz + xy' + y'z'$ and
$g(x,y,z) = x'yz + x'yz' + xy$
which is correct?

A) {f} and {g} are functionally Complete.
B) only {f} is functionally complete. ✓
C) only {g} is functionally Complete.
D) Neither {f} nor {g} is functionally Complete.

Explanation: A function to be functionally Complete if it doesn't belong to T_0, T_1, L, M, S

Property-1: if all input variables set as '0', i.e $f(0,0,\ldots 0) = 0$
$f \in T_0$

Property-2: if all input variables set as '1', i.e $f(1,1,\ldots 1) = 1$
$f \in T_1$

Property-3: dual of $f(x_1, x_2, \ldots x_n) = \overline{f(\overline{x_1}, \ldots \overline{x_n})}$.
$f \in S$ called self dual.

Property-4: for every input switching any input variable from 0 to 1 can only result in the functions switching its value from 0 to 1. and never from 1 to 0.

FCM

Property-5: Boolean function is linear if one of the below is true:

1) For every 1-value of 'f', the number of 1's in the corresponding input is odd,

01	1

and for every 0-value of 'f', the number of 1's in the corresponding input is even

11	0

2) For every 1-value of 'f', the number of 1's in the corresponding input is even,

11	1

and for every 0-value of 'f', the number of 1's in the corresponding input is odd.

	f
01	0

g(x,y,z) is clearly not functionally complete because

g(0,0,0) = (1)(0)(0) + (1)(0)(1) + (0)(0) = 0 $F \in T_0$

f(x,y,z) is neither preserving 0 nor 1.

* F is not linear, not monotone, not self dual.

- $f(x,y,z) \neq \sim f(\sim x, \sim y, \sim z)$

- ~~$\sim f((x'y'z) + (x+y') + (y'+z'))$~~

~~$fd(x,y,z) = \sim$~~ ~~$(x+y).(y+z) \in 2$~~

$fd(x,y,z) = (x'+y+z).(x+y').(y'z')$

$= (x'y' + xy + xy' + xz + y'z')(y'z')$

$= x'y'z' + xy'z' + y'z'(x'+x)$

$= y'z'$

$f(A,B,C) = AB+BC+CA$

$f_d(A,B,C) = (A+B)(B+C)(C+A)$

$= (AB+AC+B+BC)(C+A)$

$= ABC+AC+BC+BC+AB+AC+AB+ABC$

$= ABC+AC+BC+AB$

$= AB(C+1)+AC+BC$

$= AB+AC+BC$

K-Map: Note: Always remember POS $\neq$ (SOP)'

the correct form is (POS of F) = (SOP of F')'

steps for solving expression using kmap:

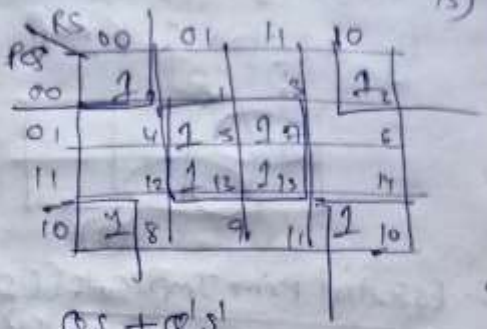
1. Select K-Map according to the number of variables.
2. Identify minterms or maxterms as given in problems.
3. For SOP put 1's in blocks ~~for~~ of K-Map respective to the minterms (0's elsewhere).
4. For POS put 0's in blocks of K-Map respective to the maxterms (1's elsewhere).
5. Make rectangular groups containing total items in powers of two like 2, 4, 8 .. (except 1) and try to cover as many elements as you can in one group.
6. From the groups made in step 5 find the product terms and sum them up for SOP form.

SOP Form:

$\Sigma A,B,C\ (1,3,6,7)$

$= A'C + AB$

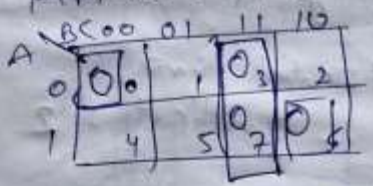
$F(P,Q,R,S) = \Sigma(0,1,5,7,8,10,13,15)$



$QS + Q'S'$

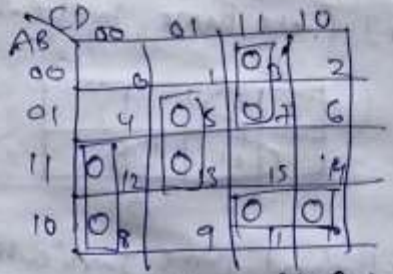
POS Form:

$F(A,B,C) = \Pi(0,3,6,7)$



$(A+B+C)(B+C')(A+B+C)$

$F(A,B,C,D) = \Pi(3,5,7,8,10,11,12,13)$



$(A'+C'+D')(B'+C'+D)(A'+C+D)$

$(A'+B+C')$

Implicants:-

- Implicant is a product / minterm term in sum of products (SOP) or sum / maxterm term in product of sums (POS) of a Boolean function. Eg: $F = AB + ABC + BC$.

Implicants are AB, ABC and BC.

Minimum no. of Essential Prime Implicants = 0

Maximum no. of Essential Prime Implicants = 2^{n-1}.

No. of Prime implicants need not be same as the no. of terms obtained in minimized function of K-Map.

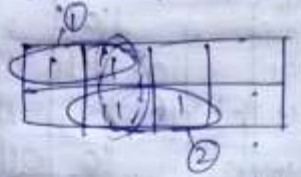
1. Prime Implicants: (largest possible group of 1's)

Select group of adjacent minterms which is allowed by definition of K map are called prime Implicants (PI).

No. of Prime Implicants = 3

2. Essential Prime Implicants (EPI) (at least there is single which can't be combined many other way)

These are those groups (subcubes) which cover atleast one minterm that can't be covered by any other prime implicant. And EPI are those implicants which always appear in solution.



No. of Essential Prime Implicants = 2

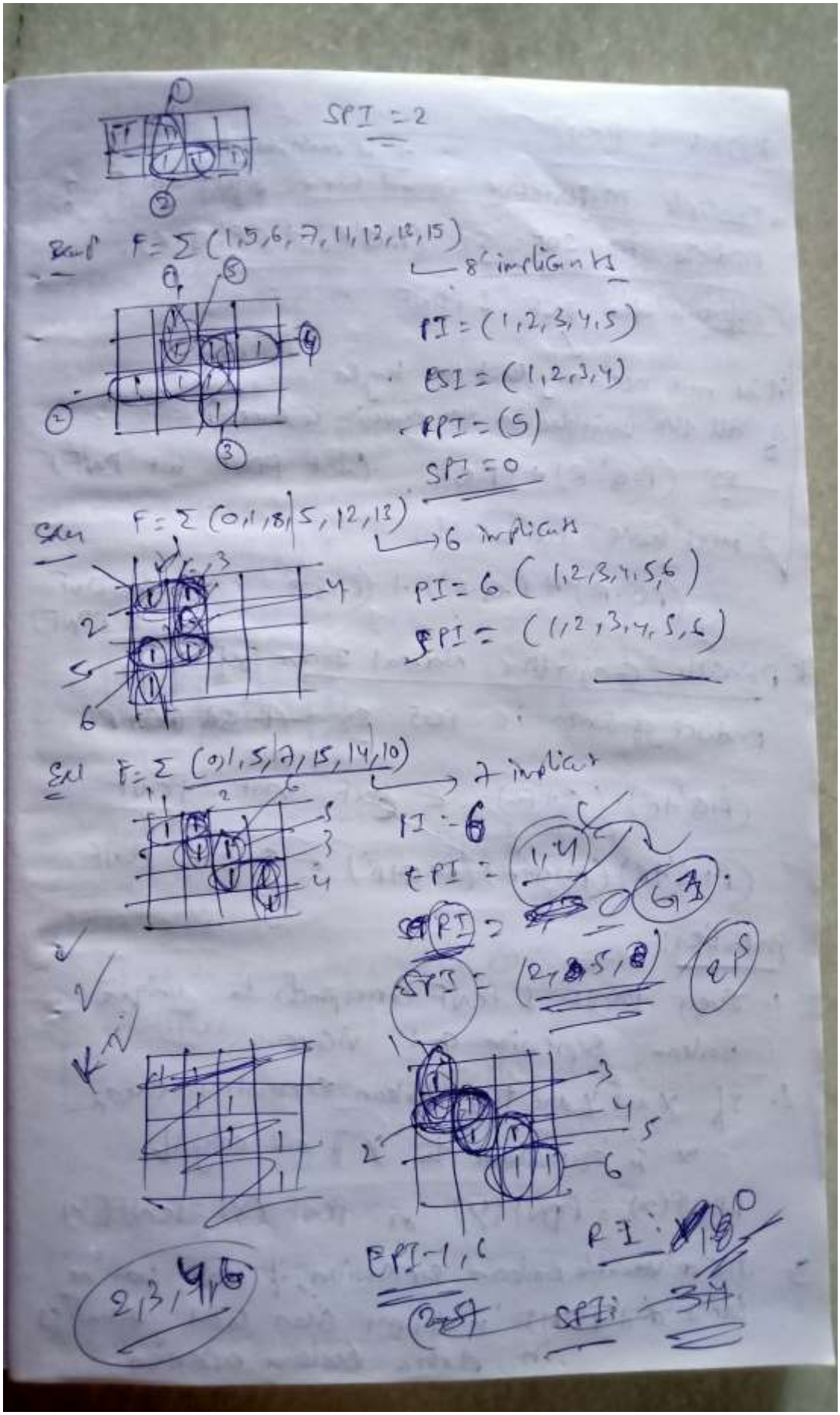
3. Redundant prime Implicants:

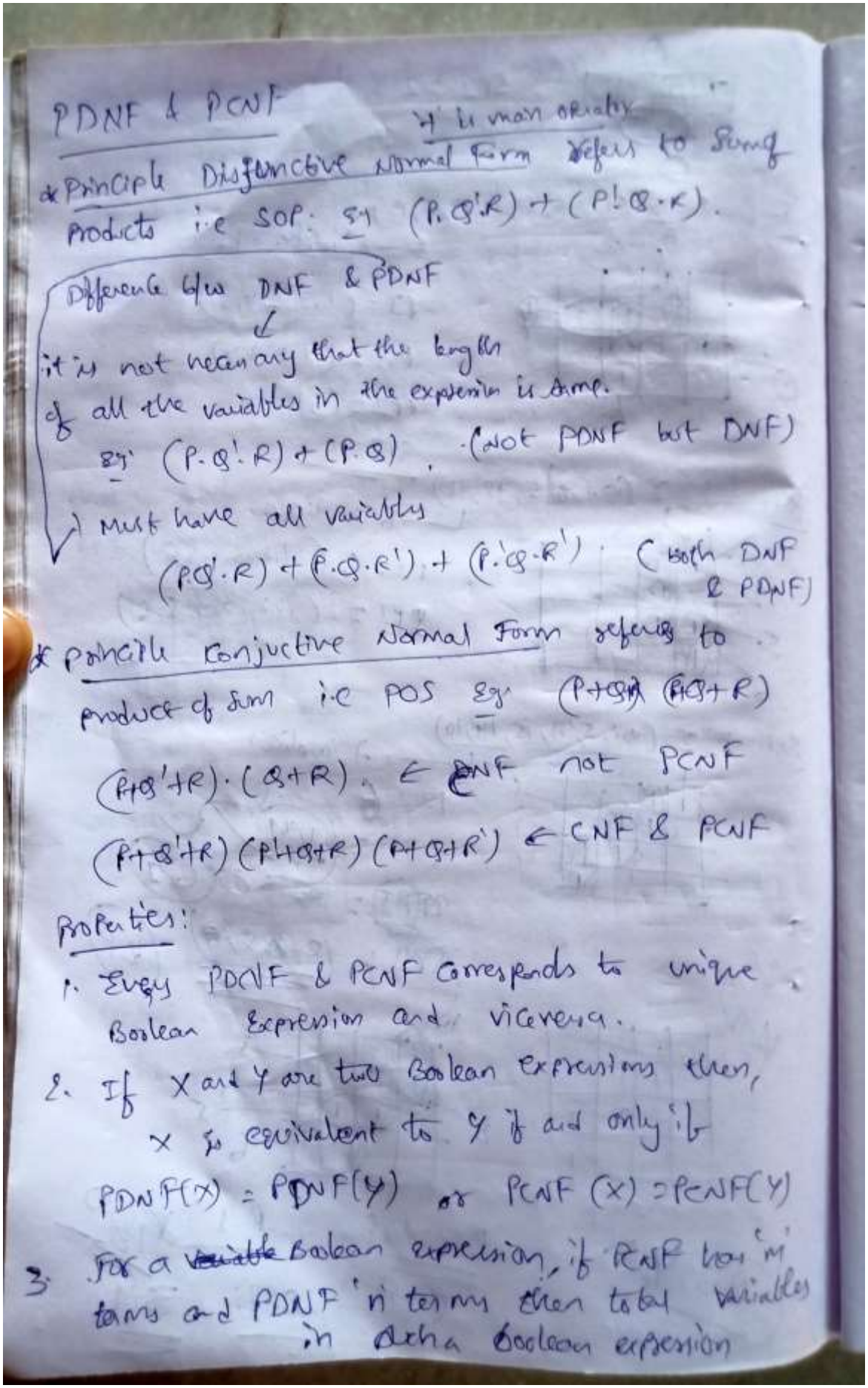
The prime implicants for which each of its minterms is covered by some essential prime implicant are redundant prime implicants (RPI).

No. of Redundant prime Implicants = 1

4. Selective prime Implicants:

The prime implicants which are neither essential nor redundant prime implicants are called SPI. these are known as non-essential prime implicants. They may appear in some solution or may not appear in some solution.


SPI = 2
Exp F = Σ (1,5,6,7,11,12,13,15)
8 implicants
PI = (1,2,3,4,5)
ESI = (1,2,3,4)
RPI = (5)
SPI = 0
Exp F = Σ (0,1,8,5,12,13)
6 implicants
PI = 6 (1,2,3,4,5,6)
SPI = (1,2,3,4,5,6)
Exp F = Σ (0,1,5,7,15,14,10)
7 implicants
PI = 6
EPI = (1,4)
SPI = (2,3,5,6)
EPI = 1,6
2,3,4,6



PDNF & PCNF

* Principle Disjunctive Normal Form refers to sum of products i.e SOP. Eg: $(P.Q'.R) + (P.Q.R)$.

Difference b/w DNF & PDNF

it is not necessary that the length of all the variables in the expression is same.

Eg: $(P.Q'.R) + (P.Q)$ (not PDNF but DNF)

Must have all variables

$(P.Q'.R) + (P.Q.R') + (P.Q.R')$ (both DNF & PDNF)

* Principle Conjuctive Normal Form refers to product of sum i.e POS Eg: $(P+Q+R)(P+Q'+R)$

$(P+Q'+R).(Q+R)$ ← DNF not PCNF

$(P+Q'+R)(P+Q+R)(P+Q+R')$ ← CNF & PCNF

Properties:

1. Every PDNF & PCNF corresponds to unique Boolean Expression and vice versa.
2. If X and Y are two Boolean expressions then, X is equivalent to Y if and only if PDNF(X) = PDNF(Y) or PCNF(X) = PCNF(Y)
3. For a ~~variable~~ Boolean expression, if PCNF has 'm' terms and PDNF 'n' terms then total variables in such a Boolean expression

iii) $\log_2(m+n)$

Variable Entrant Map (VEM)

- K-map becomes a difficult technique to manage when number of variables exceed 5 or 6. So VEM is used to increase the effective size of k-map. It allows a smaller map to handle large numbers of variables.
- this is done by writing output in terms of input.

Example: $F(A,B,C) = \Sigma(0,1,2,5)$ 'F' in terms of 'C'.

A,B,C	F
0 0 0	1
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	0

A B	F
0 0	1
0 1	C'
1 0	C
1 1	0

And the VEM is

A \ B	$\bar{B}$	B
$\bar{A}$	1	$\bar{C}$
A	C	0

~~Advantages~~:

- It is commonly used to solve problems involving multiplexers.
- A(VEM) can be used to plot more than 'n' variable using an 'n' variable K-map.

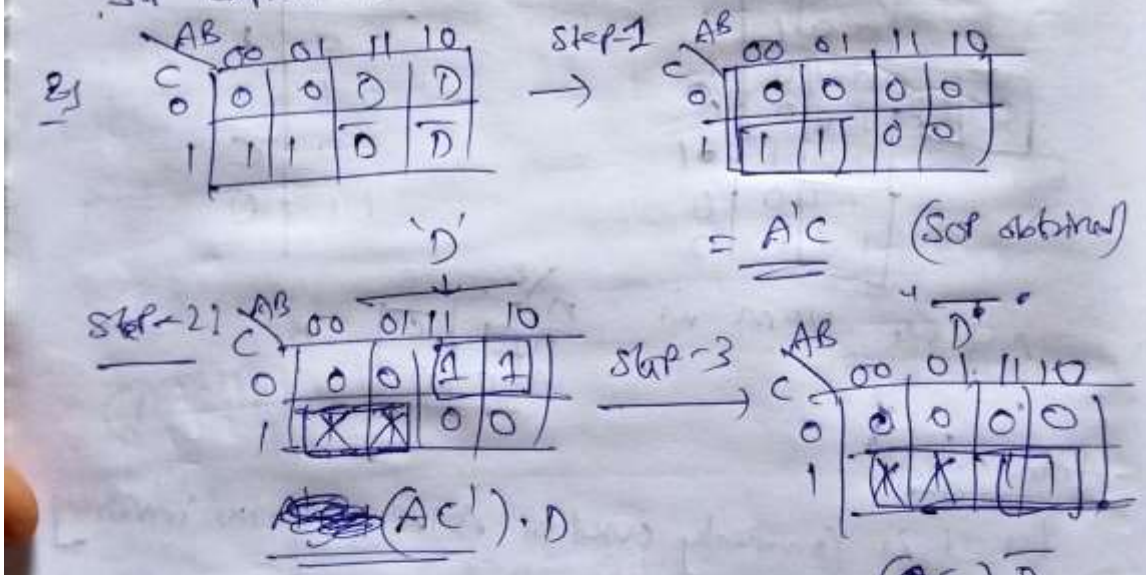
Steps to solve Variable Entrant Map:

Step-1: Keep all the variables as zero and get the expression. (Complement & not Complement take as 2 variables in K-Map)

Step-2: Select any variable and keep 1 in it, other variable should be kept as '0', given 1's should be replaced by don't care. Multiply obtained SOP with concerned variable.

Step-3: Repeat the above process for other variables too.

Step-4: SOP of VEM is obtained by OR-ing the previous SOP expression.



Step-4: ORing

$$\bar{A}C + AC\bar{D} + C\bar{D}$$

Consensus theorem / Redundancy theorem:

$Y = AB + A'C + BC = AB + A'C$

The Conjuctive dual of this equation is

$(A+B).(A'+C).(B+C) = (A+B).(A'+C)$

$(B+C)$ / (BC) is redundent term.

Conditions for applying Redundancy theorem are:-

1. Three variables must present in the expression.
Here, A, B and C are used variables in above exerple.
2. Each variables is repeated twice.
3. One variable must present in Complemented form.

After applying this theorem we can only take those terms which contains the Complented variable.

Proof: $Y = AB + A'C + BC$

$= AB + A'C + BC(A+A')$

$= AB + A'C + ABC + A'BC$

$= AB(1+C) + A'C(1+B)$

$= AB + A'C$

Exerwl $F = AB + BC' + AC$ Conditions satisfied.

(Complement exists & repeated 2 times.)

$= AB + BC' + ABC + AB'C$

~~$= AB(1+C) + B'C(1+A)$~~

$= AB + B'C + AC(B+B') + ABC' +$

$= AB(+ABC' + BC' + AC$

$AC + BC' =$

$P = (A+B)\cdot(A'+C)\cdot(B+C)$

$= (A+B)\cdot(A'+C)\cdot(B+C+AA')$

$= (A+B)(A'+C)(B+C+A)(A'+C+B)$

$= (A+B+C)(A+B+C')(A'+C)(B+C+A)$

$\cdot(A+C+B)$

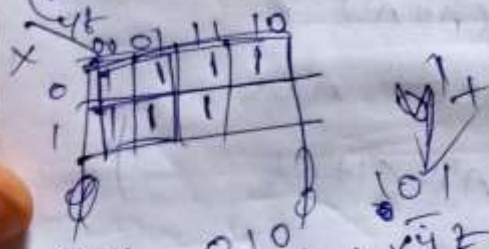
$= (A+B+C')(A'+C)(A'+C+B)$

$= A\,A'(A+B)(A'+C).$

remaining

Logic redundancy is necessary because it causes unnecessary network complexity and raises the cost of implementation.

$\overline{(ABC)\cdot(DE)'}$



$y'+$

$\bar{x}yz + \bar{x}y\bar{z}' + x\bar{y}z + xyz$

$\bar{x}y'z + \bar{x}y'z' + x\bar{y}\bar{z} + xyz$

011 010 101 111

001 000 100 101

* Combinational Circuits:

Half Adder

A ⊕ B

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

A (AND) B

Full Adder

A	B	C_{in}	S	Cout
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

Logical expression for Sum:

$= A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in}$

$= C_{in}(A'B' + AB) + C_{in}'(A'B + AB')$

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

Logical expression for Cout:

$= A'BC_{in} + AB'C_{in} + ABC_{in}' + ABC_{in}$

~~$= BC + AB'C' + ABC'$~~ = ~~BC + A~~

~~$= BC + AC'$~~ $= AB +$

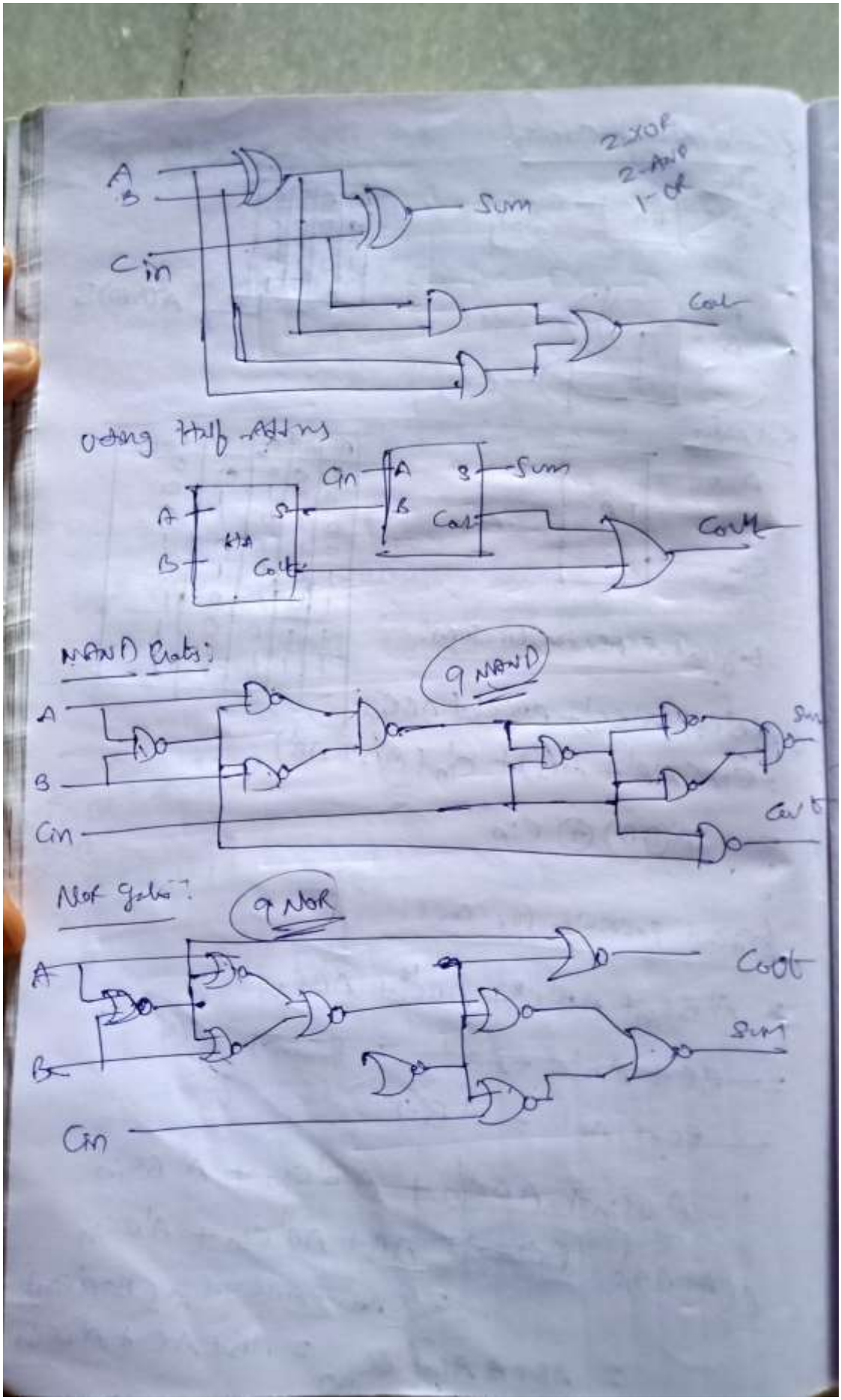
$= ABC_{in} + ABC_{in}' + AB'C_{in} + A'BC_{in}$

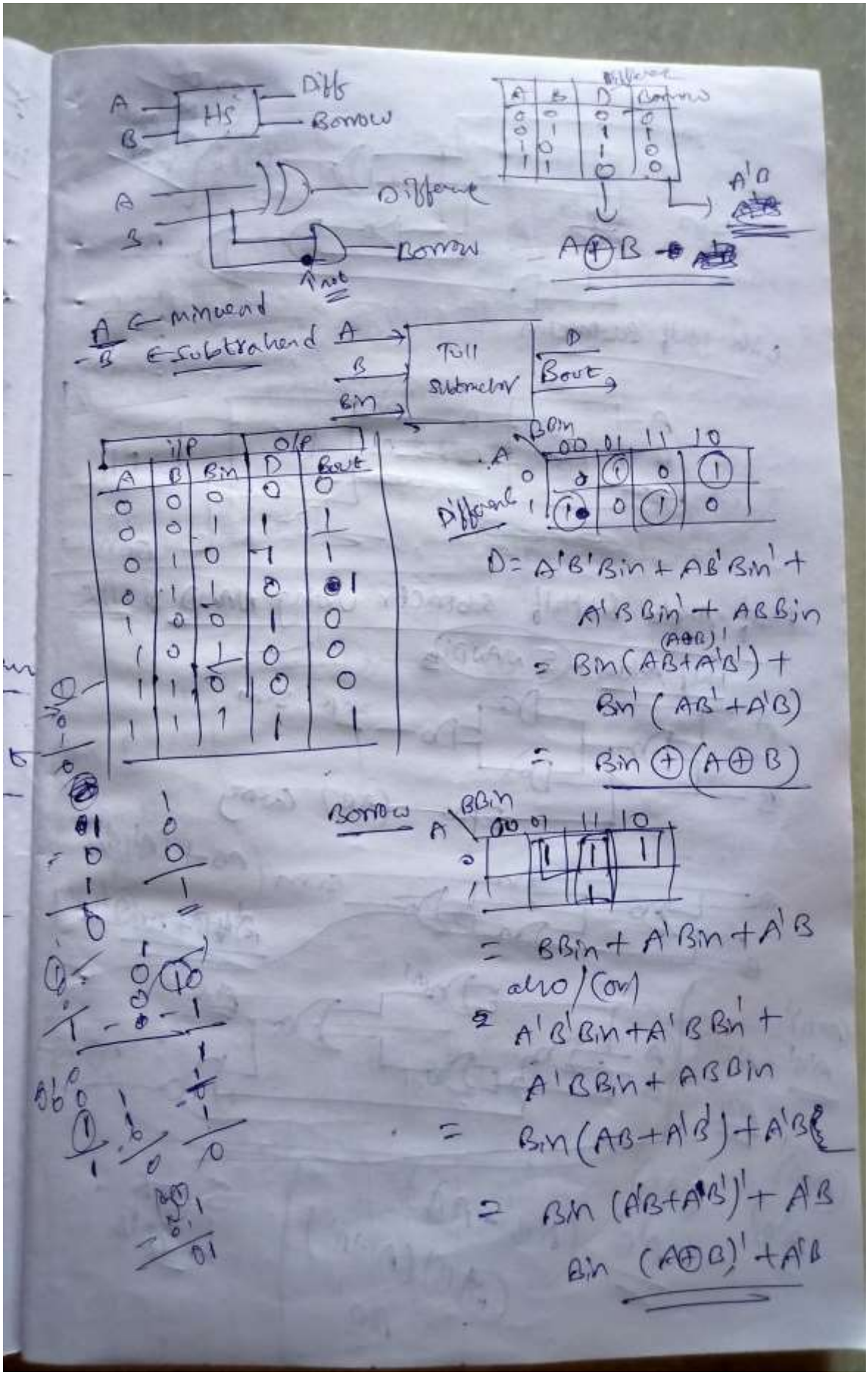
$C(A + A'B) =$ ~~BC + A~~ $AB + AB'C_{in} + A'BC_{in}$

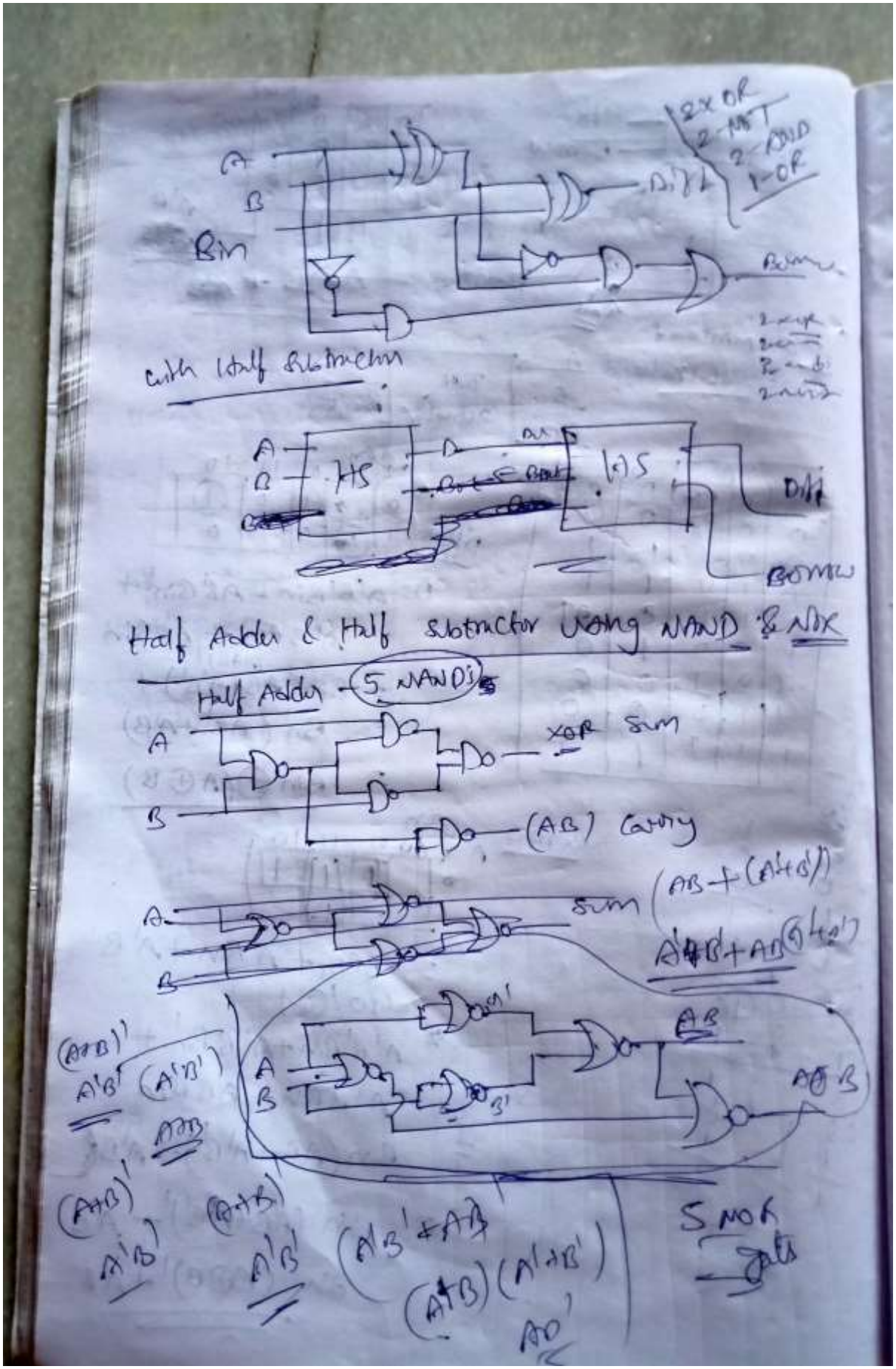
$=$ ~~AB + ABC + A~~ $A(B + B'C_{in})$

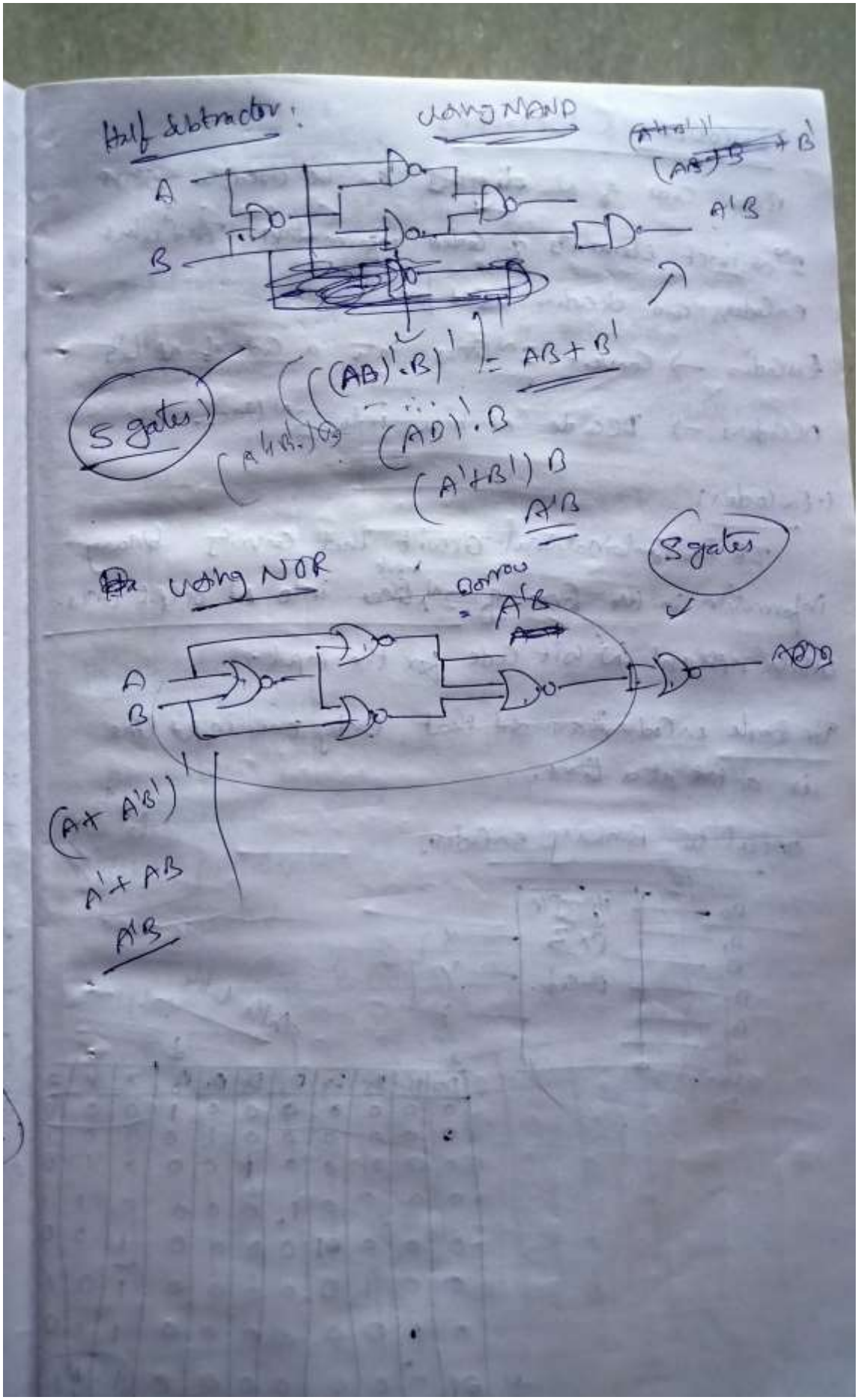
$= AB + AC + A'BC_{in}$

$= AB +$ ~~A~~ $AC + BC_{in}$


A
B
Cin
Sum
Cout
2-XOR
2-AND
1-OR
using Half Adders
HA
NAND Gates:
9 NAND
NOR Gates:
9 NOR


A
B
HS
Diff
Borrow
Different
Borrow
Not
A'B
A⊕B
A ← minuend
B ← subtrahend
Full Subtractor
Bin
Bout
D
i/p
o/p
Bin
Different
D = A'B'Bin + AB'Bin' + A'BBin' + ABBin
= Bin(AB+A'B') + Bin'(AB'+A'B)
= Bin ⊕ (A⊕B)
Borrow
= BBin + A'Bin + A'B
= A'B'Bin + A'BBin' + A'BBin + ABBin
= Bin(AB+A'B') + A'B
= Bin (AB+A'B')' + A'B
Bin (A⊕B)' + A'B


2 XOR
2 NOT
2 AND
1 OR
A
B
Bin
with Half Subtractor
HS
HS
Diff
Borrow
Half Adder & Half Subtractor using NAND & NOR
Half Adder - 5 NAND's
XOR Sum
(AB) Carry
Sum
AB+(A'+B')
A'B+AB(A+B')
(AB)'
A'B'
(A'B')
AB
(A+B)'
(A+B)
A'B'
A'B'
(A'B'+AB)
(A+B)(A'+B')
AB
A⊙B
5 NOR gates


Half Subtractor:
using NAND
A
B
A'B
5 gates
(AB)'.B)' = AB+B'
(AB)'.B
(A'+B').B
A'B
using NOR
Borrow
= A'B
5 gates
A
B
(A+A'B')'
A'+AB
A'B

ENCODERS & DECODERS

Binary code of N digits can be used to store 2^N distinct elements of coded information. This what encoders and decoders used for.

Encoder → Convert 2^N lines into a code of N bits.

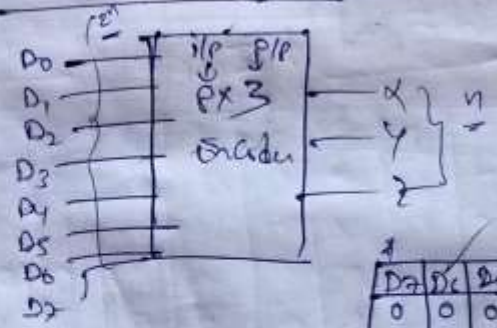
Decoders → Decode N bits into 2^N lines.

1. Encoder:

It is a combinational circuit that converts binary information in the form of 2^N max lines into N output lines which represent N bit code for the input.

For simple encoder, it assumed that only one input line is active at a time.

Octal to Binary encoder:



Truth table

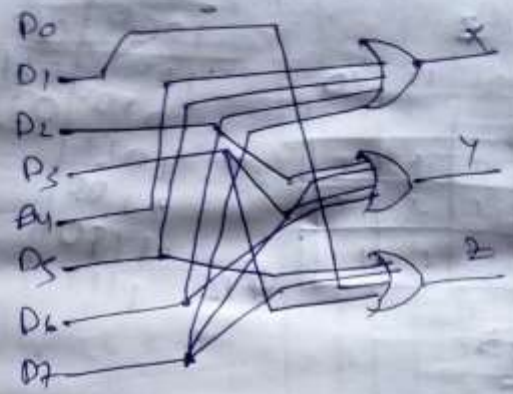
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	x	y	z
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
→ 1	0	0	0	0	0	0	0	1	1	1

$X = D_4 + D_5 + D_6 + D_7$

$Y = D_2 + D_3 + D_6 + D_7$

$Z = D_1 + D_3 + D_5 + D_7$

realized with OR gates



Priority Encoder

A priority encoder is an encoder circuit in which inputs are given priorities. When more than one inputs are activated the same, the input with higher priority takes precedence and the output corresponding to that is generated.

D3 has highest priority
D0 has lowest

4 to 2 priority encoder - 4×2

with table

D_3	D_2	D_1	D_0	X	Y	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

If more than one input is high priority encoder gives binary code corresponding to the input which is high priority

Implementation:

It can be clearly seen that the condition for valid bit to be '1' is that at least any one of the inputs should be high.

$V = D_0 + D_1 + D_2 + D_3$

For X:

D_1D_0 / D_3D_2	00	01	11	10
00	X			
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

D_3	D_2
1000	0100
1001	0101
1010	0110
1011	0111
1100	
1101	
1110	
1111	

$X = D_2 + D_3$

~~$Y = D_1 D_2' + D_3$~~

Decoder: A decoder does the opposite job of an encoder. It is a combinational circuit that converts 'n' lines of input into 2^n lines (maximum) of output.

Truth table — 3-8 line decoder — n-m decoder

$n \leq 2^m$

X	Y	Z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

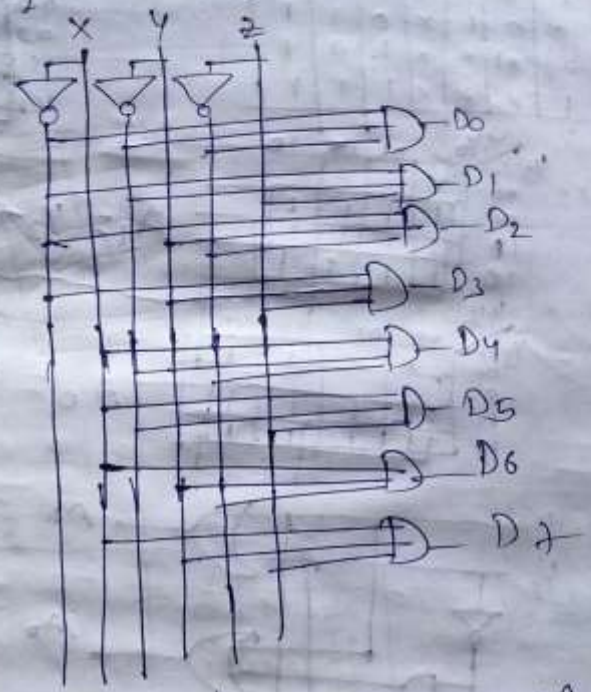
Implementation

$D_0 = x'y'z'$ ← x=0, y=0, z=0, D_0 is high..

similarly

$D_1 = x'y'z$, $D_2 = x'yz'$, $D_3 = x'yz$, ...

Logic circuit



Decimal to BCD ENCODER:

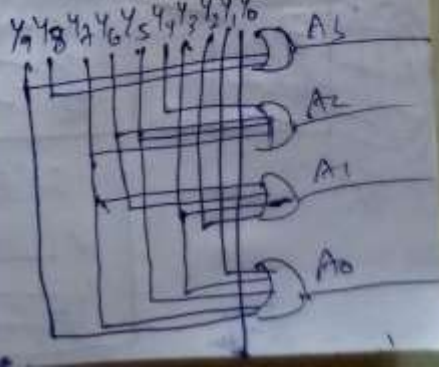
Y_9	Y_8	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A_3	A_2	A_1	A_0
0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	1
0	0	0	1	0	0	0	0	0	0	0	1	1	0
0	0	1	0	0	0	0	0	0	0	0	1	1	1
0	1	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0	0	1

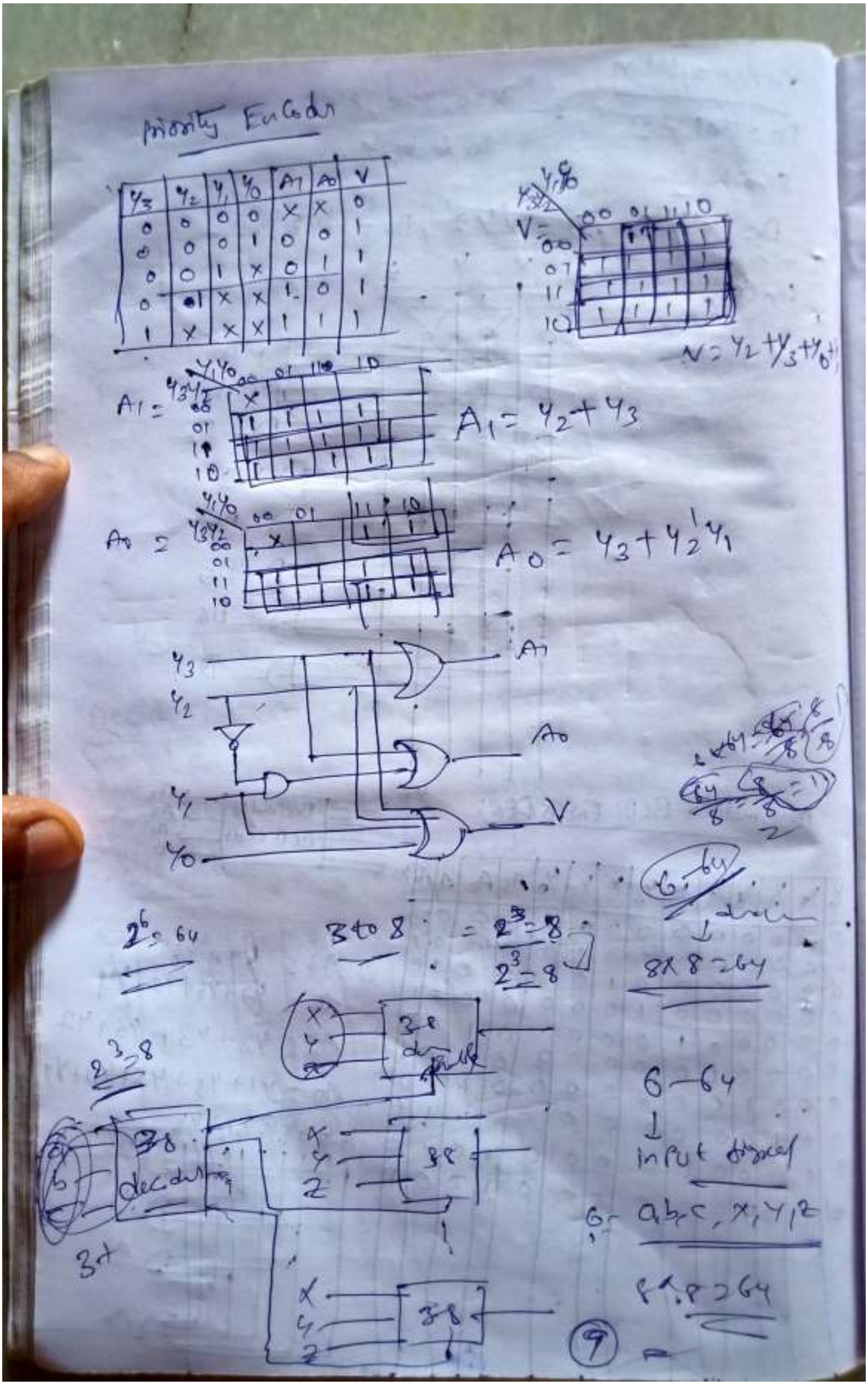
$A_3 = Y_9 + Y_8$

$A_2 = Y_7 + Y_6 + Y_5 + Y_4$

$A_1 = Y_2 + Y_3 + Y_6 + Y_7$

$A_0 = Y_1 + Y_3 + Y_5 + Y_7 + Y_9$





Priority Encoder
A1 = Y2 + Y3
A0 = Y3 + Y2'Y1
A1
A0
V
3 to 8
8x8 = 64
6 - 64
input
decoder

Multiplexer:-

It is a combinational circuit which have many data inputs and single output depending control or select inputs.

For N input lines, $\log_2 N$ (base 2) selection lines, or we can say that 2^n input lines, n selection lines are required.

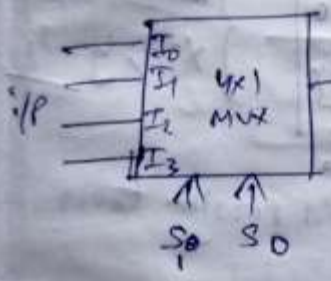
Multiplexer are also known as "Data noselector"

Parallel to serial Convertor,

many-to one-circuit,

universal-logic circuit.

Application: mainly used to increase amount of data that can be sent over the network with certain amount of time & bandwidth.



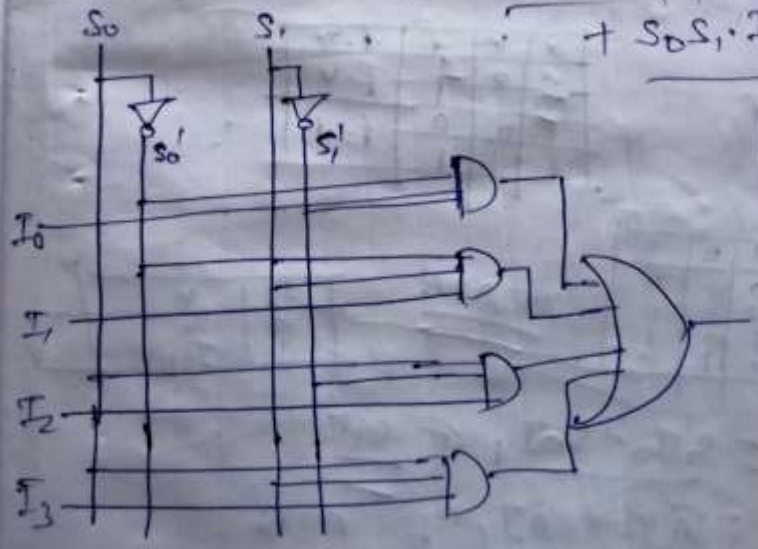
4:1 MUX

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

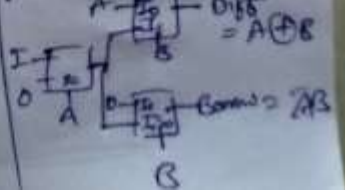
S_0	Y
0	I_0
1	I_1

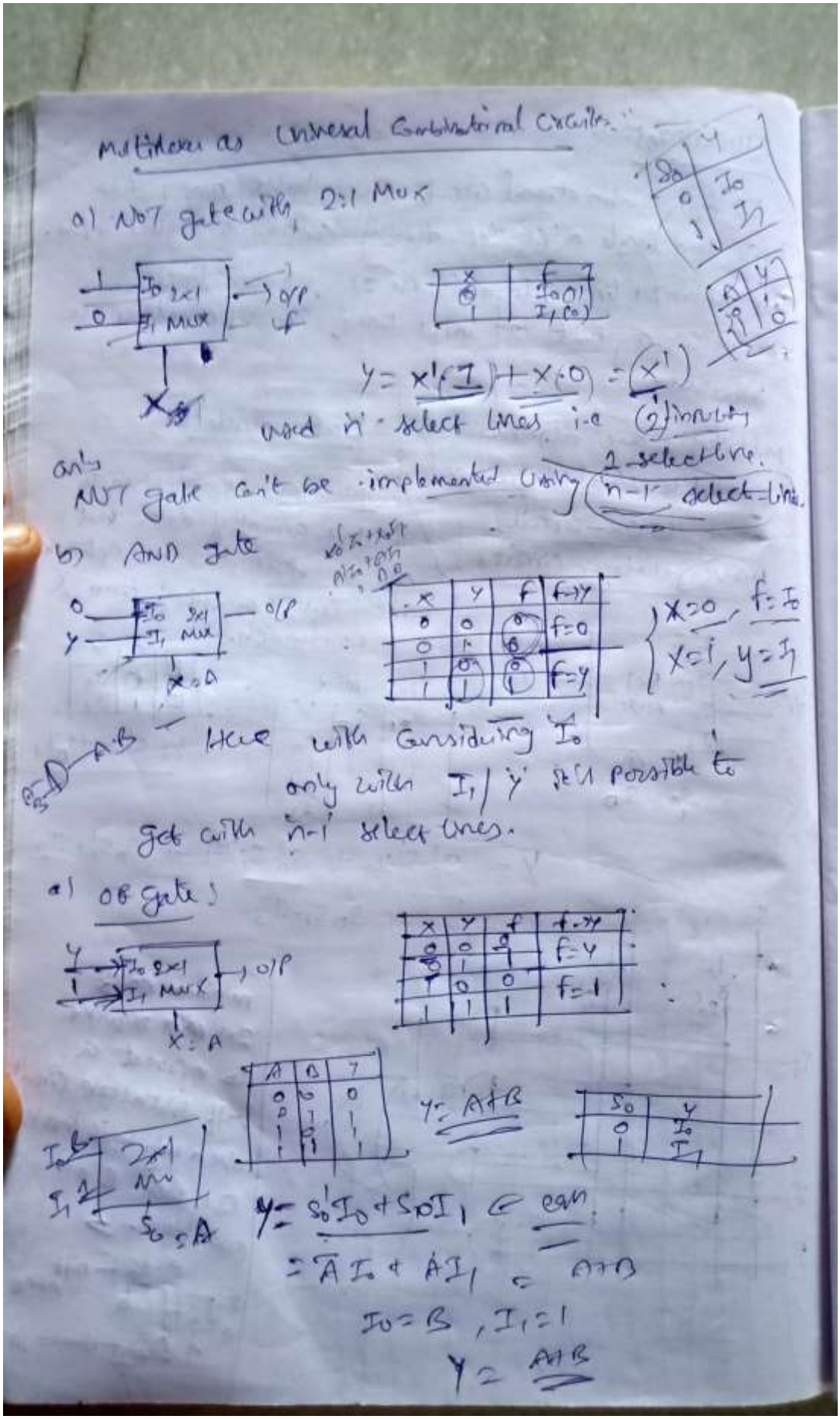
$Y = S_0' I_0 + S_0 I_1$

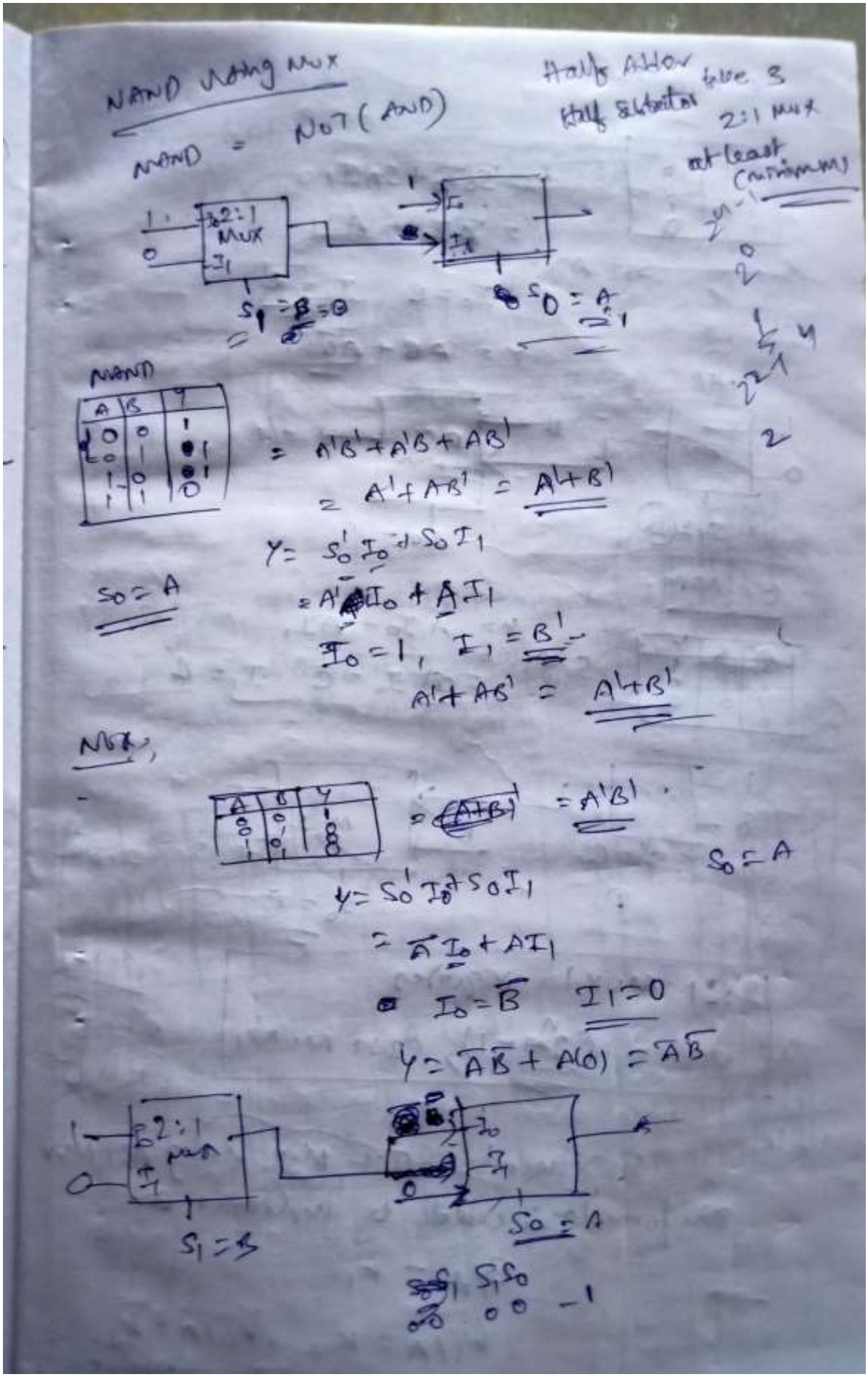
$$Y = S_0' S_1' I_0 + S_0' S_1 I_1 + S_0 S_1' I_2 + S_0 S_1 I_3$$

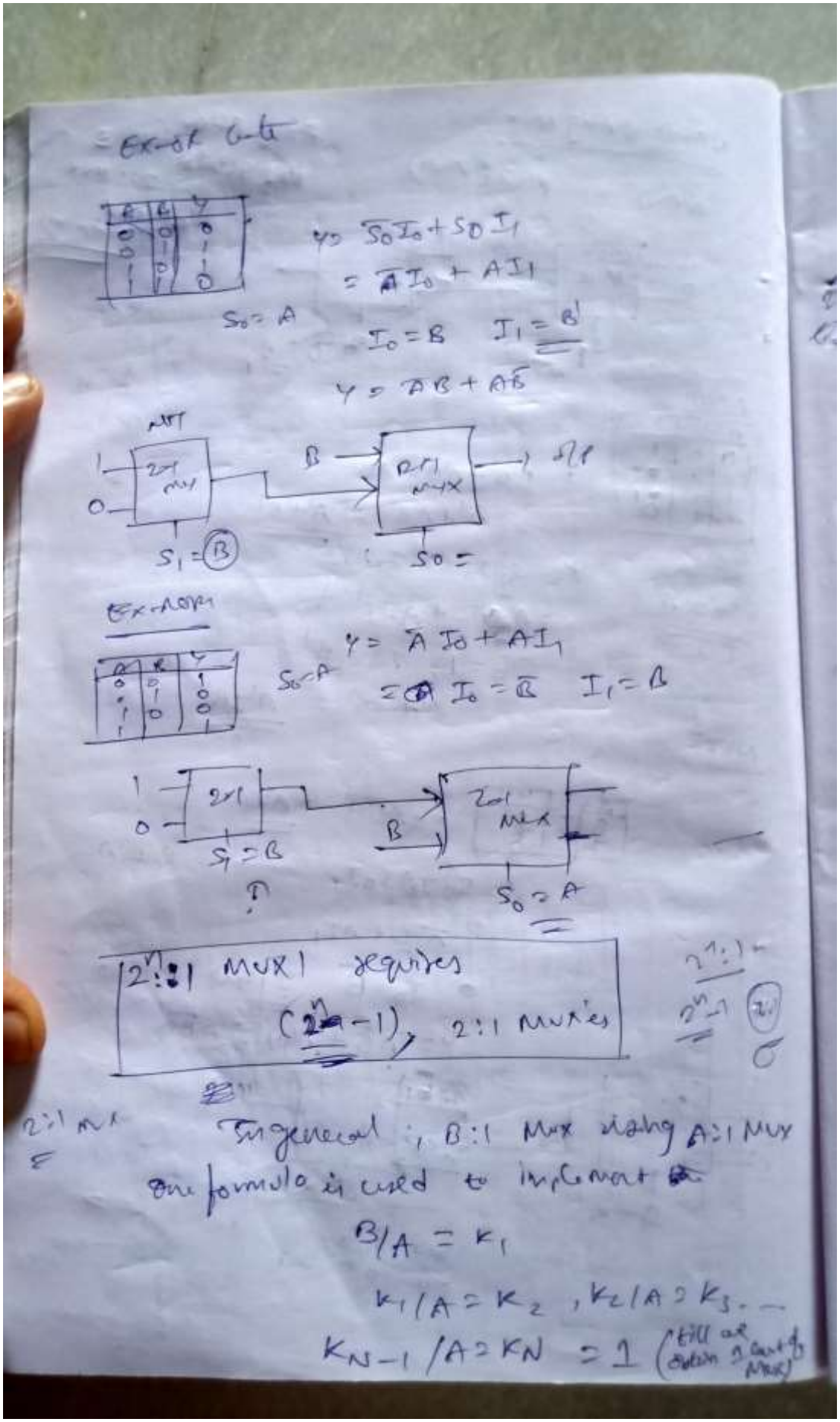


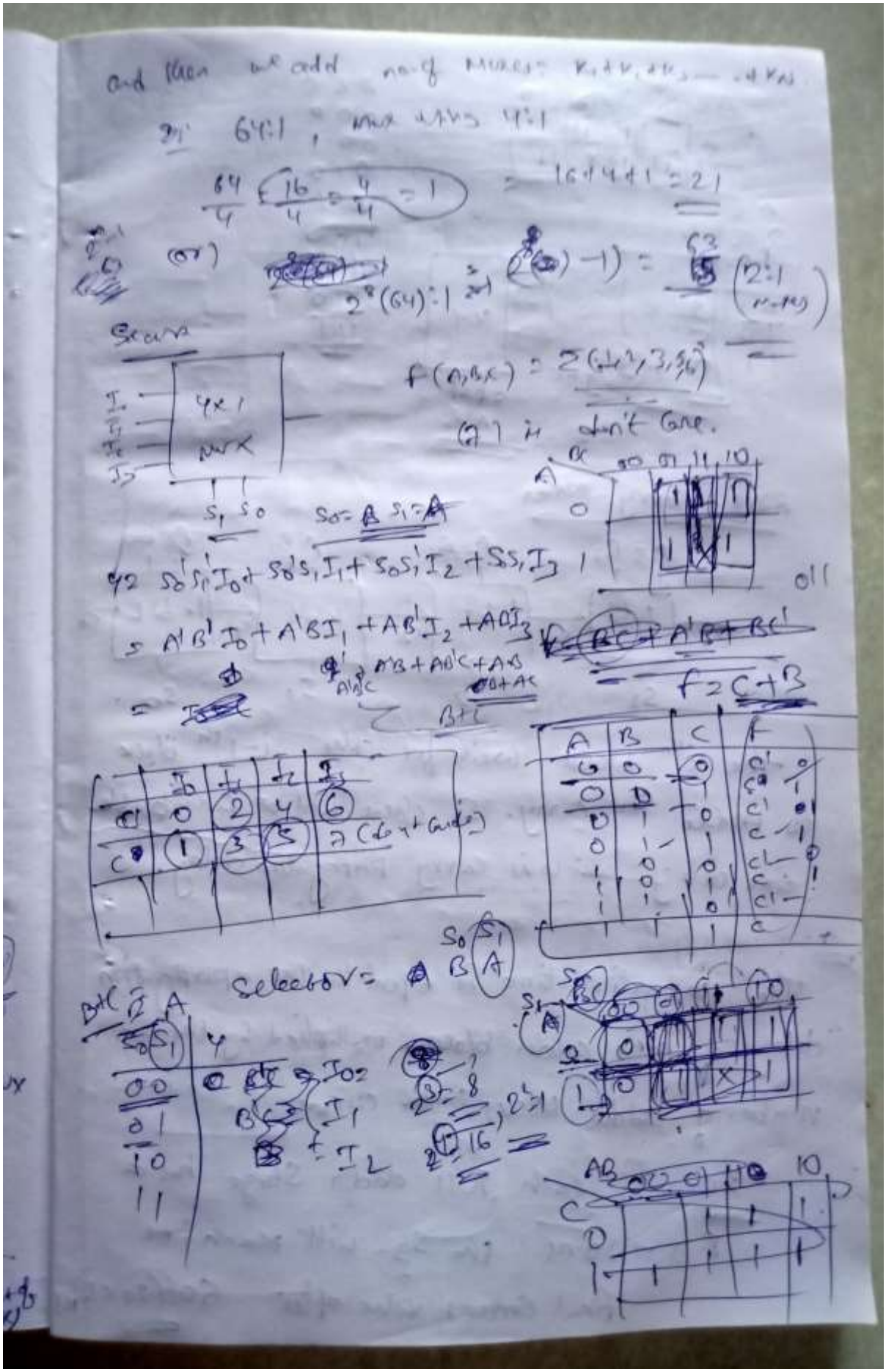
minimum no. of 2x1 MUX required to implement a half-subtraction circuit when only basic inputs 0, 1, A and B are available = 3

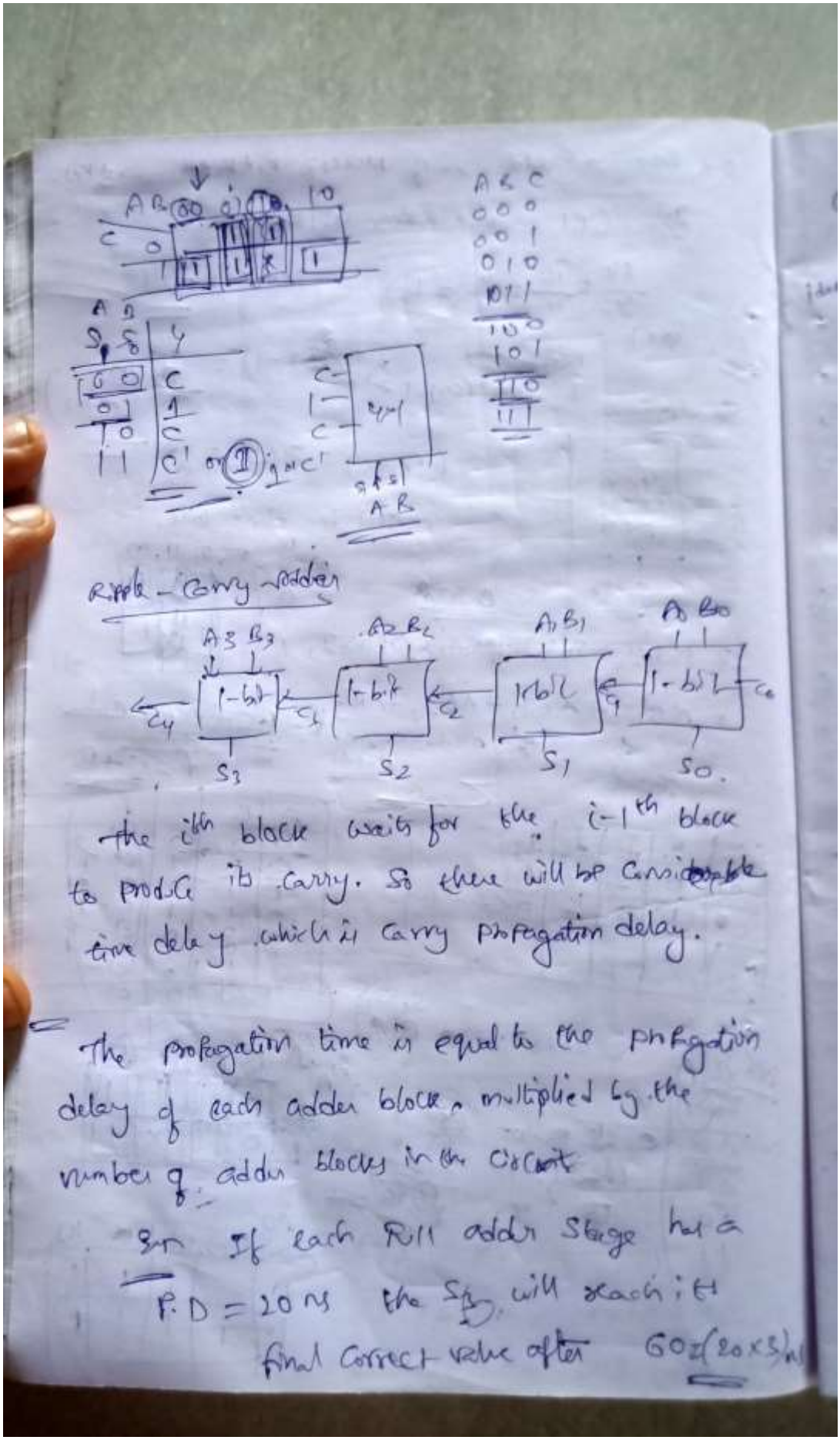



Multiplexer as Universal Combinational Circuits
a) NOT gate with 2:1 Mux
y = x'·I + x·0 = x'
used n-1 select lines i.e (2) inverter
1 select line.
and NOT gate can't be implemented using n-1 select lines.
b) AND gate
AB
Here with considering I0 only with I1 it's not possible to get with n-1 select lines.
c) OR Gate
y = A + B
y = S0'I0 + S0I1 = eqn
= Ā I0 + A I1 = A + B
I0 = B, I1 = 1
Y = A + B


NAND using MUX
NAND = NOT (AND)
Half Adder
Half Subtractor
2:1 MUX
at least
(minimum)
NAND
A B Y
0 0 1
0 1 1
1 0 1
1 1 0
= A'B' + A'B + AB'
= A' + AB' = A' + B'
Y = S0' I0 + S0 I1
= A' I0 + A I1
I0 = 1, I1 = B'
A' + AB' = A' + B'
S0 = A
NOT
A B Y
0 0 1
0 1 0
1 0 0
1 1 0
= (A+B)' = A'B'
Y = S0' I0 + S0 I1
= Ā I0 + A I1
I0 = B̄ I1 = 0
Y = ĀB̄ + A(0) = ĀB̄
S1 = B
S0 = A
S1 S0
0 0 - 1


Ex-or Gate
A B Y
0 0 0
0 1 1
1 0 1
1 1 0
$S_0 = A$
$Y = S_0 I_0 + S_0 I_1$
$= \bar{A} I_0 + A I_1$
$I_0 = B \quad I_1 = B'$
$Y = A\bar{B} + A\bar{B}$
NOT
2:1 MUX
B
2:1 MUX
$S_1 = B$
$S_0 =$
EX-NOR
A B Y
0 0 1
0 1 0
1 0 0
1 1 1
$S_0 = A$
$Y = \bar{A} I_0 + A I_1$
$I_0 = \bar{B} \quad I_1 = B$
2:1
B
2:1 MUX
$S_1 = B$
$S_0 = A$
2^n:1 MUX1 requires
$(2^n - 1)$, 2:1 MUXes
2^n:1
$2^n - 1$
2:1
2:1 MUX
In general, B:1 Mux using A:1 Mux
one formula is used to implement
$B/A = K_1$
$K_1/A = K_2$, $K_2/A = K_3$ ...
$K_{N-1}/A = K_N = 1$ (till we obtain 1 output MUX)

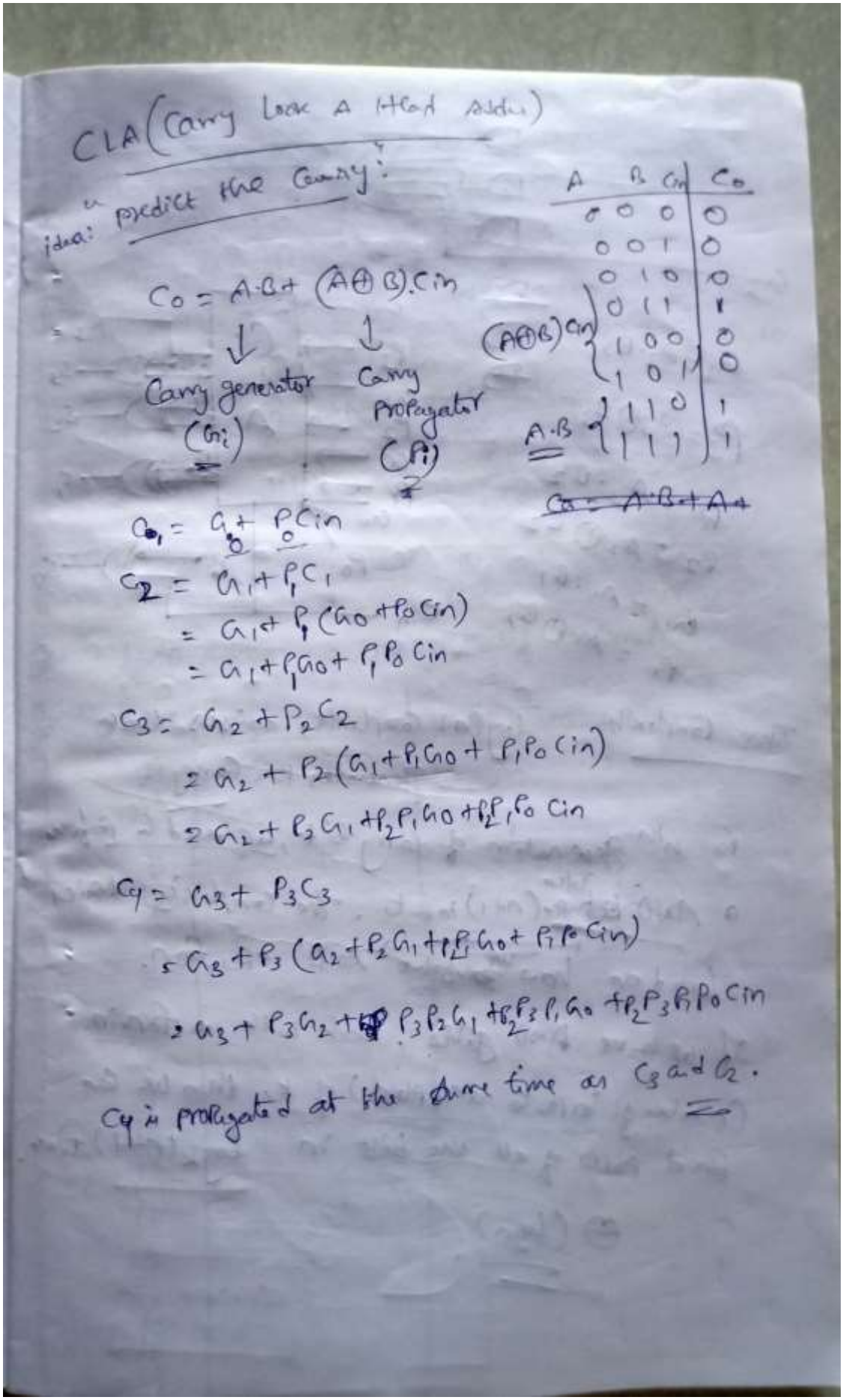




the i^{th} block waits for the $(i-1)^{th}$ block to produce its carry. So there will be considerable time delay which is carry propagation delay.

The propagation time is equal to the propagation delay of each adder block, multiplied by the number of adder blocks in the circuit

Ex: If each full adder stage has a P.D = 20 ns the S_3 will reach its final correct value after 60 (20 × 3) ns



CLA (Carry Look Ahead Adder)

idea: predict the Carry:

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

↓ Carry generator (G_i) — ↕ Carry Propagator (P_i)

A	B	C_{in}	C_0
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$(A \oplus B) C_{in}$: rows 0 1 2 3 4 5 — $A \cdot B$: rows 6 7

~~$C_0 = A \cdot B + A \cdot$~~

$C_1 = G_0 + P_0 C_{in}$

$C_2 = G_1 + P_1 C_1$

$= G_1 + P_1 (G_0 + P_0 C_{in})$

$= G_1 + P_1 G_0 + P_1 P_0 C_{in}$

$C_3 = G_2 + P_2 C_2$

$= G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_{in})$

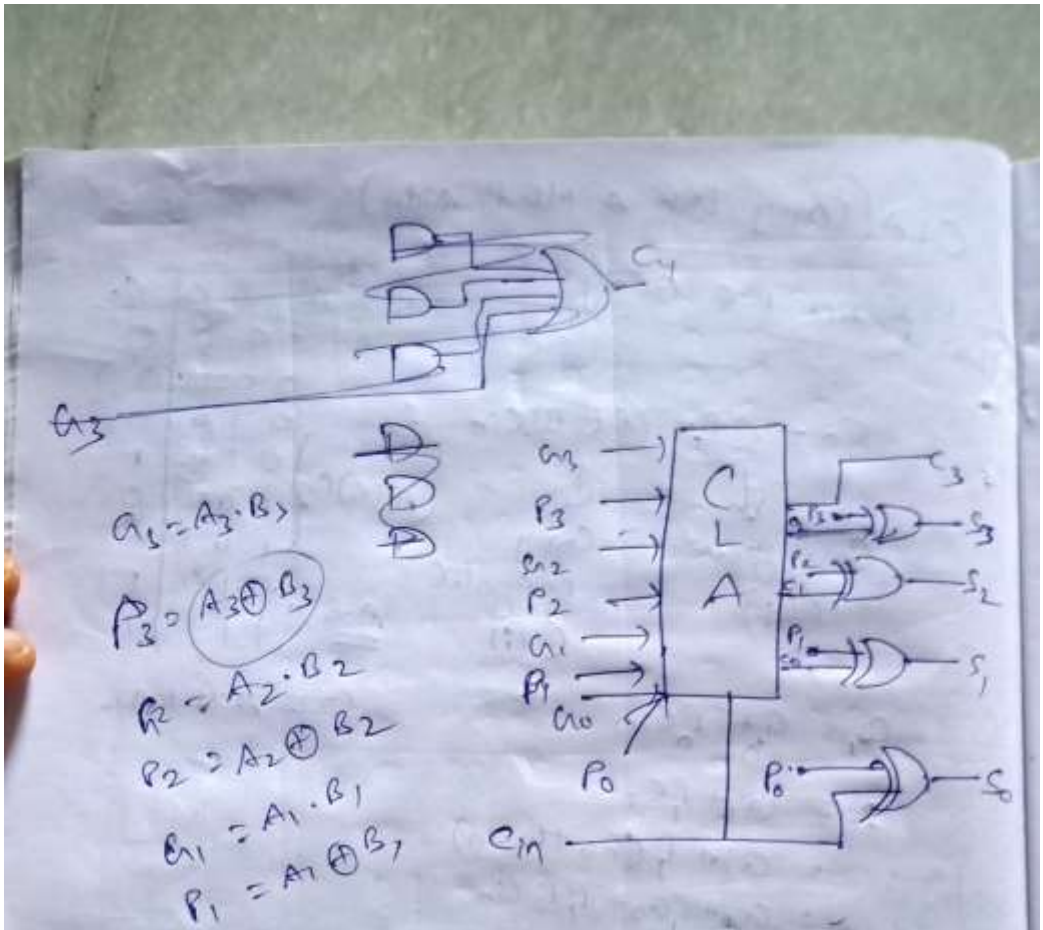
$= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$

$C_4 = G_3 + P_3 C_3$

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

$= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$

C_4 is propagated at the same time as C_3 and C_2.



Time Complexity — $\log(n)$ Complexity arises that generates the Carry.

for n^{th} generation of carry bit, we need to perform a AND ~~between~~ $(n+1)$ inputs. The complexity basically based on how we ~~solve~~ AND operation.

If we have AND gates, each with a fan-in (number of inputs accepted) of k, then we can find AND of all the bits in $\log_k(n+1)$ time.

$\Theta(\log n)$

Consider a two level logic implementation of the look-ahead carry generator. Assume that all P_i and G_i are available for the carry generator circuit and that the AND and OR gates can have any number of inputs.

The number of AND gates needed to implement the look-ahead carry generator for a 4-bit with S_3, S_2, S_1, S_0 and C_4 as its outputs respectively.

Answer : 10, 4

AND Gates $\frac{n(n+1)}{2}$, OR "n"

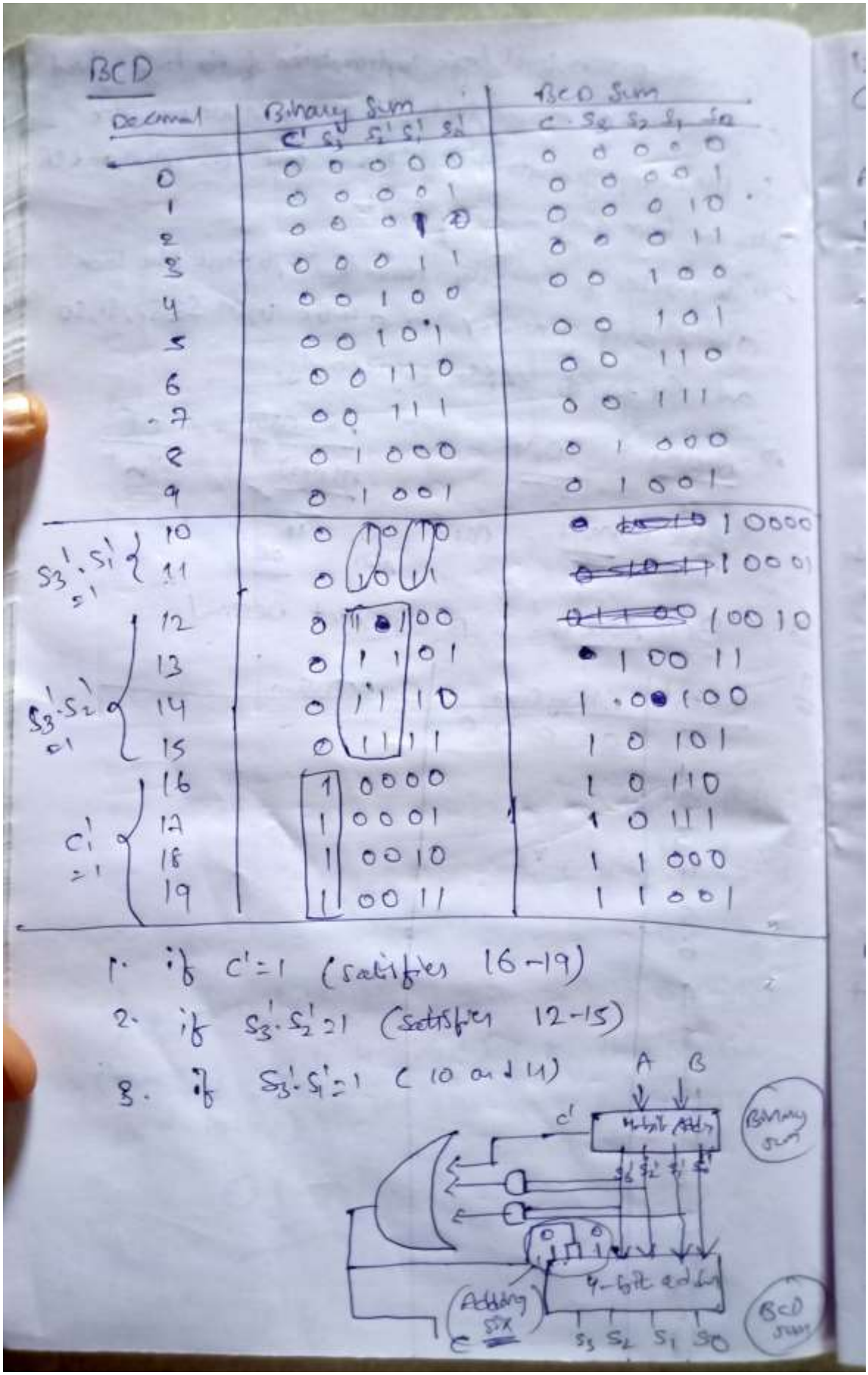
Here $n=4$ then 10, 4

AND OR

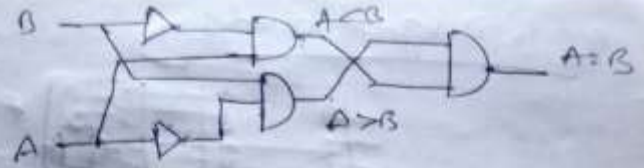
23 BCD Adder: Binary Coded Decimal.

19

Decimal	Binary Sum	BCD Sum
0	0 0 0 0 0	
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		


BCD
Decimal
Binary Sum
C' S4' S3' S2' S1'
BCD Sum
C S8 S4 S2 S1
0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 1 0 0 0 0 1
2 0 0 0 1 0 0 0 0 1 0
3 0 0 0 1 1 0 0 0 1 1
4 0 0 1 0 0 0 0 1 0 0
5 0 0 1 0 1 0 0 1 0 1
6 0 0 1 1 0 0 0 1 1 0
7 0 0 1 1 1 0 0 1 1 1
8 0 1 0 0 0 0 1 0 0 0
9 0 1 0 0 1 0 1 0 0 1
10
11
12
13
14
15
16
17
18
19
S3'.S1' = 1
S3'.S2' = 1
C' = 1
1. if C' = 1 (satisfies 16-19)
2. if S3'.S2' = 1 (satisfies 12-15)
3. if S3'.S1' = 1 (10 and 11)
A B
C'
4-bit adder
Adding 6
S3 S2 S1 S0
Binary sum
BCD sum

1-bit Comparator



$A>B: AB'$

$A<B: A'B$

$A=B: AB + A'B' \quad (A \odot B)$

$(A<B) + (A>B) = AB' + A'B$

Complement on both sides

$((A<B) + (A>B))' = (AB')' (A'B)'$

$= (A'+B)(A+B')$

$= (AA' + AB + A'B' + BB')$

$((A<B) + (A>B))' = AB + A'B'$

$= (A=B)$

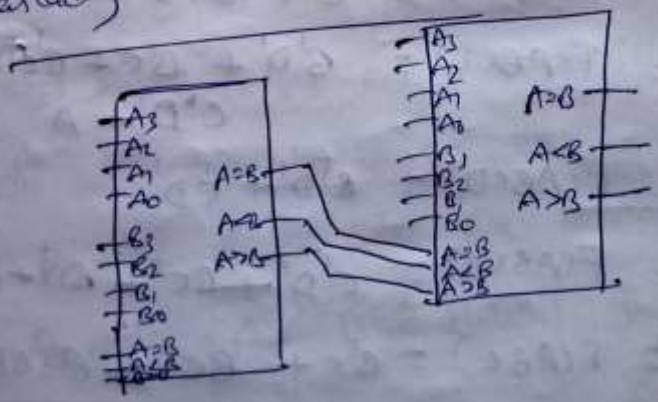
2-bit

$A>B: A_1B_1' + A_0B_1'B_0' + A_1A_0B_0'$

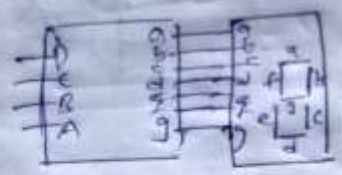
$A=B: (A_0 \odot B_0)(A_1 \odot B_1)$

$A<B: A_1'B_1 + A_0'B_1B_0 + A_1'A_0'B_0$

Cascading comparators of 4-bit:



BCD to 7-Segment decoder:



← 7 Segment display

BCD to 7 Segment Decoder

Cathode – Anode
Anode – Cathode

Common Cathode type BCD to 7 Segment

A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

For common Anode interchange 0's with 1's in outputs 1's with 0's

K-map functions

for a = F(ABCD) = $B'D' + C + BD + A$

b = F(ABCD) = $B' + C'D' + CD$

c = F(ABCD) = $C' + D + B$

d = F(ABCD) = $B'D' + BC + B'CD + C'D' + A$

e = F(ABCD) = $B'D' + CD'$

f = F(ABCD) = $C'D' + BC' + BD' + A$

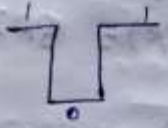
g = F(ABC) = $B'C + BC' + A + GD'$

Static hazards - A temporary fluctuation in output of the circuit is called hazards.

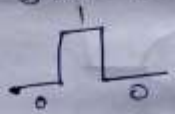
1) Static
2) Dynamic
3) Functional

Static hazard: It takes place when change in an input causes the o/p to change momentarily before stabilizing to its correct value. (momentarily)

Static-1 hazard: current o/p is '1' and after the (SOP) input changes it state changes to '0' before setting on to 1.

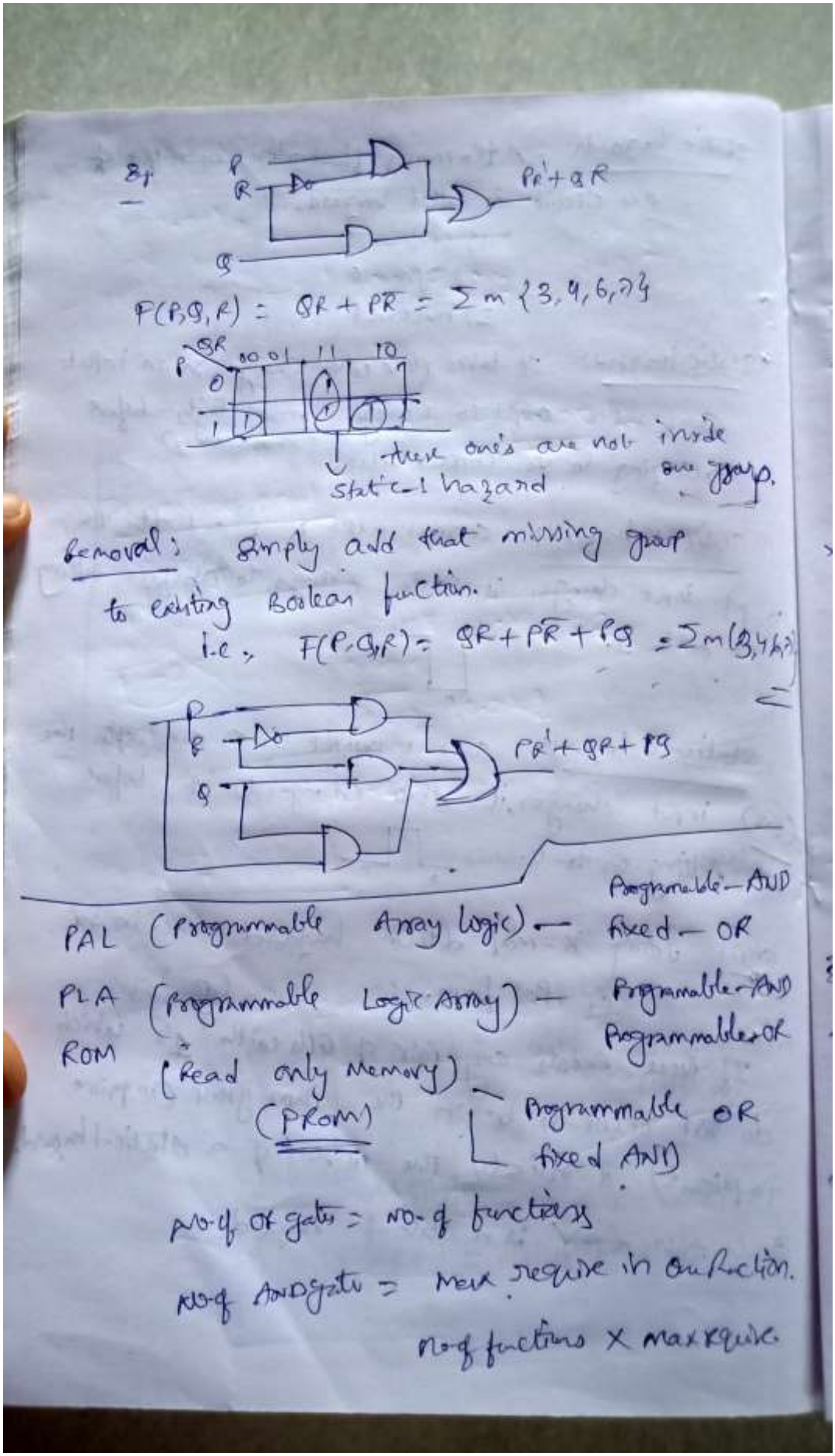


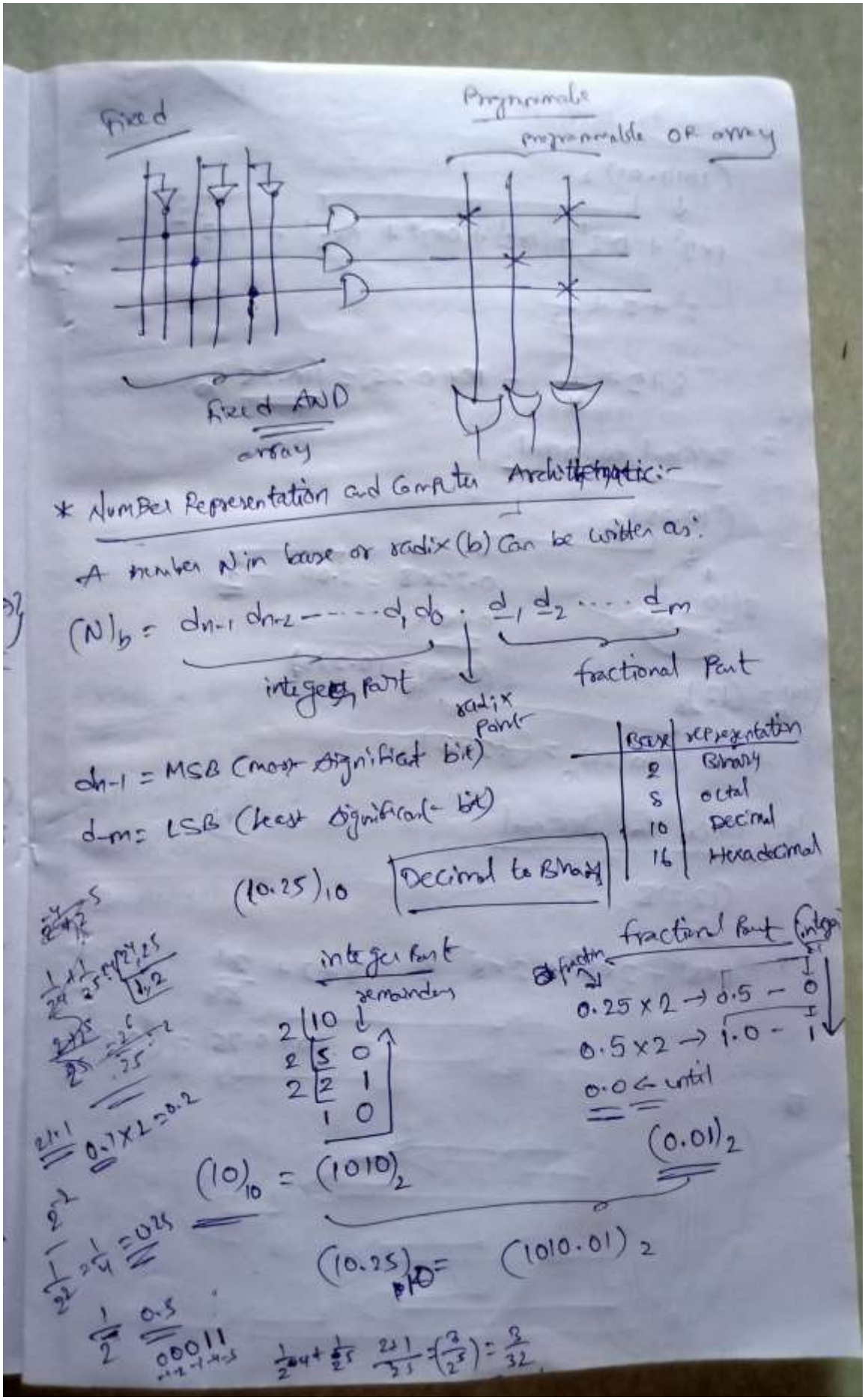
Static-0 hazard: current output is 0 and after the (POS) input changes, it state changes to '1' before setting on to 0.



steps using K-map detect hazard either in SOP or POS form.

If there exists any pair of cells with 1's which do not occur to be in the same group (i.e. prime implicant), it indicates the presence of a static-1 hazard. Each such a pair is a static-1 hazard


P
R
Q
PR'+QR
F(P,Q,R) = QR + PR = Σm{3,4,6,7}
QR
00 01 11 10
P
0
1
these one's are not inside any group.
Static-1 hazard
Removal: simply add that missing group to existing Boolean function.
i.e; F(P,Q,R) = QR + PR' + PQ = Σm(3,4,6,7)
PR'+QR+PQ
PAL (Programmable Array Logic) — Programable-AND fixed-OR
PLA (Programmable Logic Array) — Programable-AND Programable-OR
ROM (Read only Memory) — Programable OR fixed AND
(PROM)
No of OR gates = no. of functions
No of AND gate = Max require in a function.
no of functions × max require



* Number Representation and Computer Arithmetic:-

A number N in base or radix (b) can be written as:

$(N)_b = d_{n-1} d_{n-2} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-m}$

integer part | radix point | fractional Part

d_{n-1} = MSB (most significant bit)

d_{-m} = LSB (least significant bit)

Base	representation
2	Binary
8	octal
10	Decimal
16	Hexadecimal

$(10.25)_{10}$ Decimal to Binary

integer Part → remainders

2 | 10 ↓
2 | 5 — 0
2 | 2 — 1
1 — 0 ↑

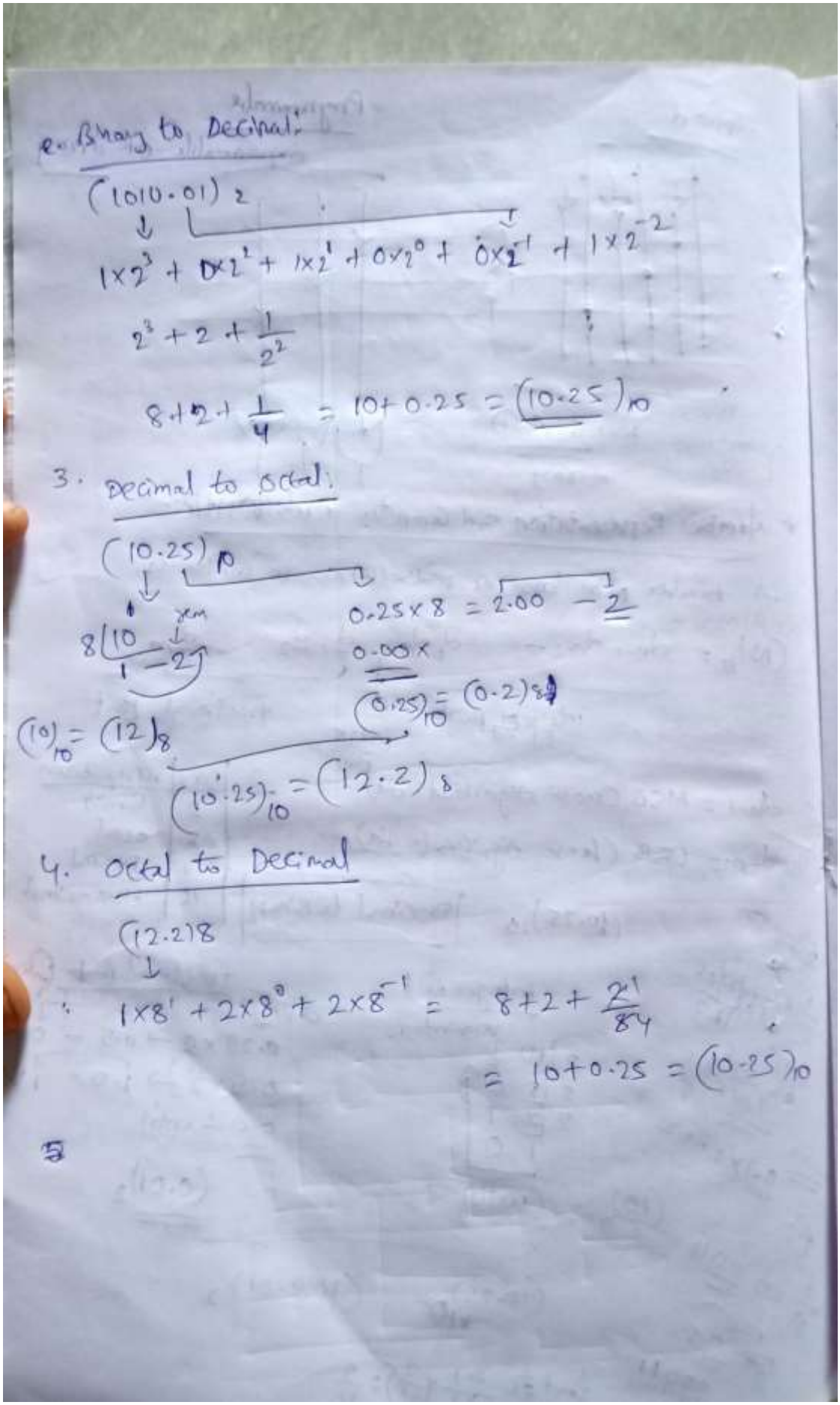
$(10)_{10} = (1010)_2$

fractional Part → integer

0.25 × 2 → 0.5 — 0 ↓
0.5 × 2 → 1.0 — 1 ↓
0.0 ← until

$(0.01)_2$

$(10.25)_{10} = (1010.01)_2$



e. Binary to Decimal:

$(1010.01)_2$

$1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 + 0\times2^{-1} + 1\times2^{-2}$

$2^3 + 2 + \frac{1}{2^2}$

$8 + 2 + \frac{1}{4} = 10 + 0.25 = (10.25)_{10}$

3. Decimal to Octal:

$(10.25)_{10}$

$8\overline{)10}$ rem

$1 - 2$

$(10)_{10} = (12)_8$

$0.25 \times 8 = 2.00 - 2$

$0.00 \times$

$(0.25)_{10} = (0.2)_8$

$(10.25)_{10} = (12.2)_8$

4. Octal to Decimal

$(12.2)_8$

$1\times8^1 + 2\times8^0 + 2\times8^{-1} = 8 + 2 + \frac{2^1}{8^1}$

$= 10 + 0.25 = (10.25)_{10}$

5. Hexadecimal and Binary

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Hexa to Binary

Eg. $(3A)_{16} = (0011\ 1010)_2$

Binary to Hexa

00 11 | 110 | 1011 (groups of 4 from right)

↓ add zeroes = | ↓ | ↓ ← left out

3 D B $= (3DB)_{16}$

Excess-3 binary code is unweighted - self complementary BCD code.

Self Complementary property means that the 1's complement of an excess-3 number is the excess-3 code of the 9's complement of corresponding decimal number.

Excess-3 means adding 3 to it. 1's Complement

← Binary

A B C D

↳ LSB

↓

MSB

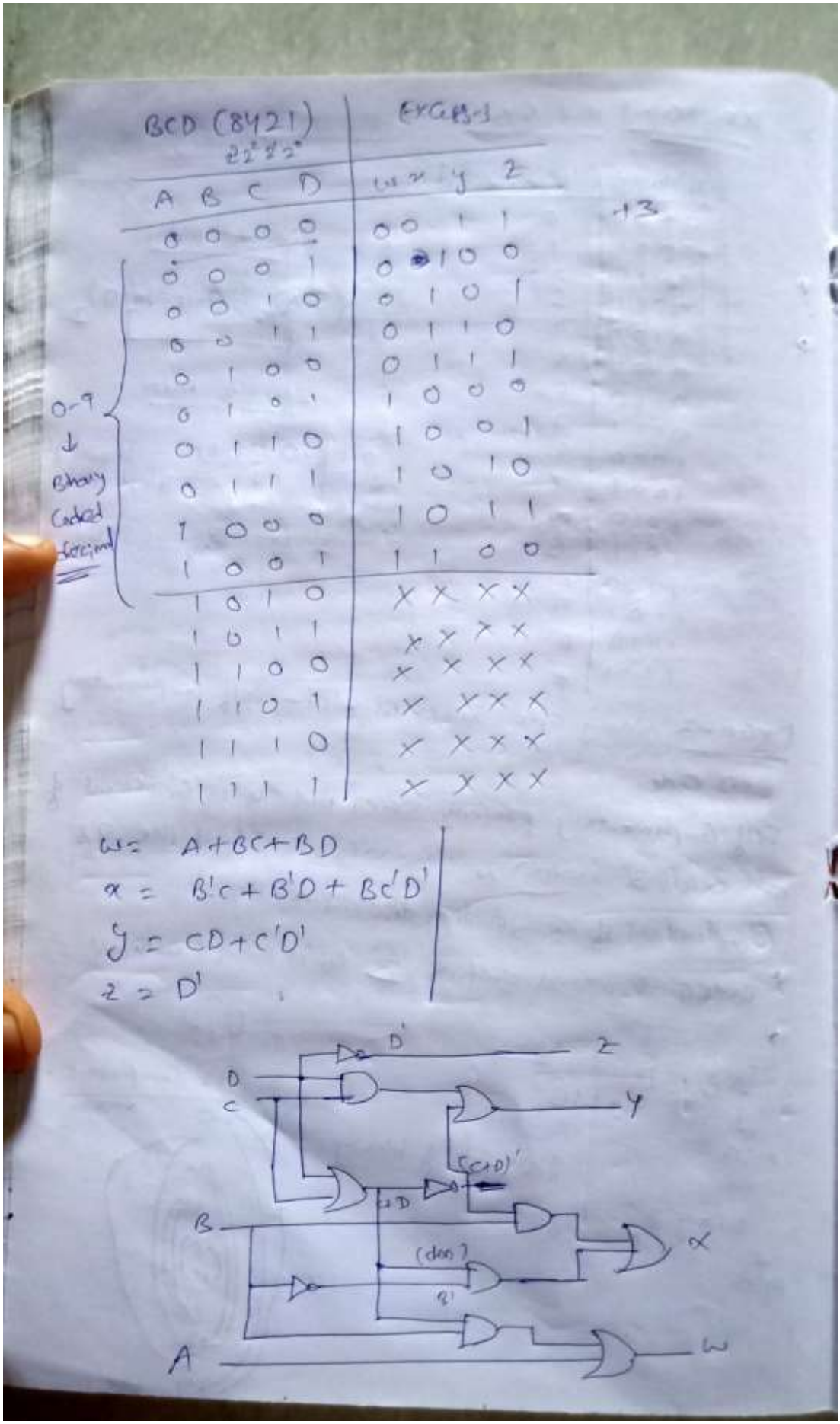
gray code of binary

w x y z

↳ LSB

↓

MSB


BCD (8421)
$2^3 2^2 2^1 2^0$
A B C D
EXCESS-3
w x y z
+3
0-9
↓
Binary
Coded
Decimal
X X X X
W = A + BC + BD
x = B'C + B'D + BC'D'
y = CD + C'D'
z = D'
D'
z
D
C
y
(C+D)'
B
(don't)
x
B'
A
w

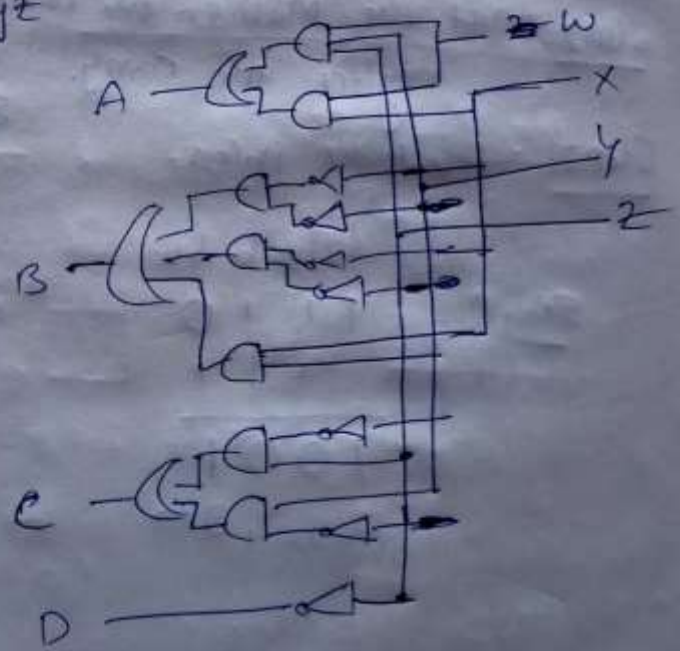
Excess-3	BCD (8421)
w x y z	A B C D
0 0 0 0	X X X X
0 0 0 1	X X X X
0 0 1 0	X X X X
0 0 1 1	0 0 0 0
0 1 0 0	0 0 0 1
0 1 0 1	0 0 1 0
0 1 1 0	0 0 1 1
0 1 1 1	0 1 0 0
1 0 0 0	0 1 0 1
1 0 0 1	0 1 1 0
1 0 1 0	0 1 1 1
1 0 1 1	1 0 0 0
1 1 0 0	1 0 0 1
1 1 0 1	X X X X
1 1 1 0	X X X X
1 1 1 1	X X X X

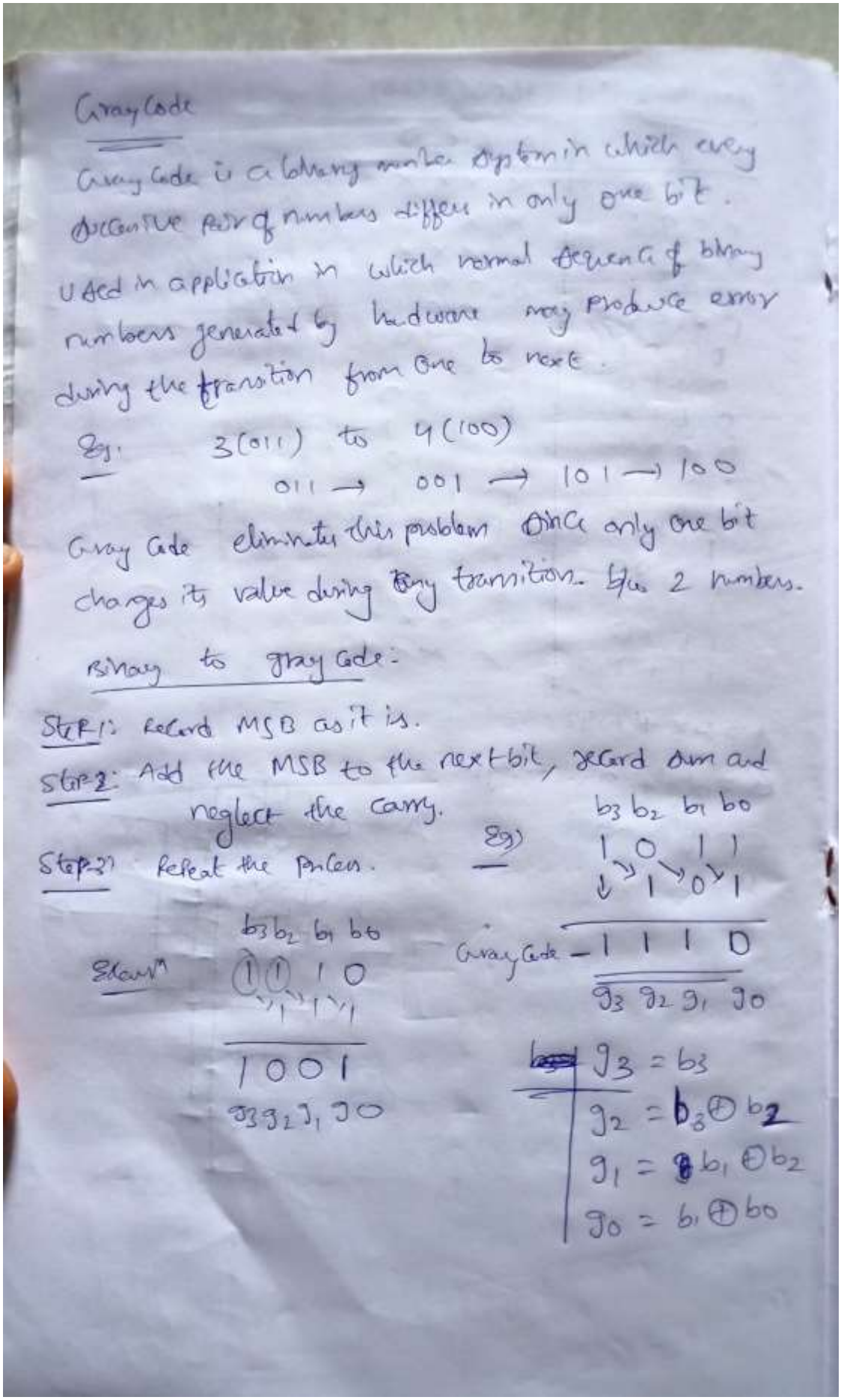
$A = wx + wyz$

$B = xy' + x'z' + xyz$

$C = y'z + yz'$

$D = z'$





Gray Code

Gray code is a binary number system in which every successive pair of numbers differs in only one bit.

Used in application in which normal sequence of binary numbers generated by hardware may produce error during the transition from one to next.

Eg: 3(011) to 4(100)

011 → 001 → 101 → 100

Gray code eliminates this problem since only one bit changes its value during any transition between 2 numbers.

Binary to gray code:

Step 1: Record MSB as it is.

Step 2: Add the MSB to the next bit, record sum and neglect the carry.

Step 3: Repeat the process.

Eg)

$b_3\ b_2\ b_1\ b_0$

1 0 1 1

1 1 0 1

Gray code – 1 1 1 0

$g_3\ g_2\ g_1\ g_0$

Eg)

$b_3\ b_2\ b_1\ b_0$

1 0 1 0

1 1 1 1

1 0 0 1

$g_3\ g_2\ g_1\ g_0$

$g_3 = b_3$

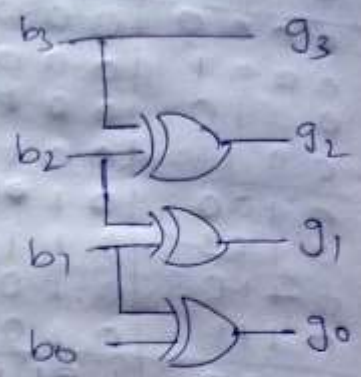
$g_2 = b_3 \oplus b_2$

$g_1 = b_1 \oplus b_2$

$g_0 = b_1 \oplus b_0$

Binary	Gray
$b_3\ b_2\ b_1\ b_0$	$g_3\ g_2\ g_1\ g_0$
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 1
0 0 1 0	0 0 1 1
0 0 1 1	0 0 1 0
0 1 0 0	0 1 1 0
0 1 0 1	0 1 1 1
0 1 1 0	0 1 0 1
0 1 1 1	0 1 0 0
1 0 0 0	1 1 0 0
1 0 0 1	1 1 0 1
1 0 1 0	1 1 1 1
1 0 1 1	1 1 1 0
1 1 0 0	1 0 1 0
1 1 0 1	1 0 1 1
1 1 1 0	1 0 0 1
1 1 1 1	1 0 0 0

B-G



Gray to Binary:

Step-1 Record the MSB bit

Step-2 Add MSB to the next bit of Gray code, record the sum and neglect the carry.

Step-3 Repeat the Process.

$b_3 = g_3$

$b_2 = b_3 \oplus g_2$

$b_1 = b_2 \oplus g_1$

$b_0 = b_1 \oplus g_0$

Example

~~$g_3\ g_2\ g_1\ g_0$~~

~~1 1 1 0~~

~~1 0 0 1~~

Binary

$g_3\ g_2\ g_1\ g_0$

1 1 1 0

1 0 1 1

Binary: $b_3\ b_2\ b_1\ b_0$

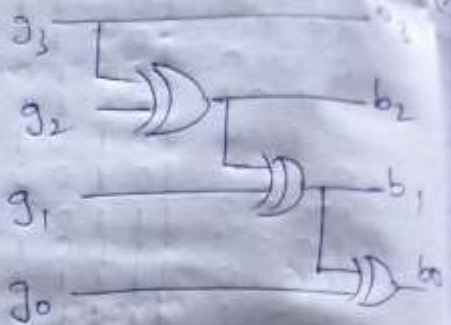
$g_3\ g_2\ g_1\ g_0$

1 0 0 1

1 ① 1 0

$b_3\ b_2\ b_1\ b_0$

Gray	Binary
$g_3\ g_2\ g_1\ g_0$	$b_3\ b_2\ b_1\ b_0$
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 1
0 0 1 0	0 0 1 1
0 0 1 1	0 0 1 0
0 1 0 0	0 1 1 1
0 1 0 1	0 1 1 0
0 1 1 0	0 1 0 0
0 1 1 1	0 1 0 1
1 0 0 0	1 1 1 1
1 0 0 1	1 1 1 0
1 0 1 0	1 1 0 0
1 0 1 1	1 1 0 1
1 1 0 0	1 0 0 0
1 1 0 1	1 0 0 1
1 1 1 0	1 0 1 1
1 1 1 1	1 0 1 0



Negative Number Representation:

* **Sign Magnitude:**

first bit dedicated to represent the sign and hence it is called sign bit.

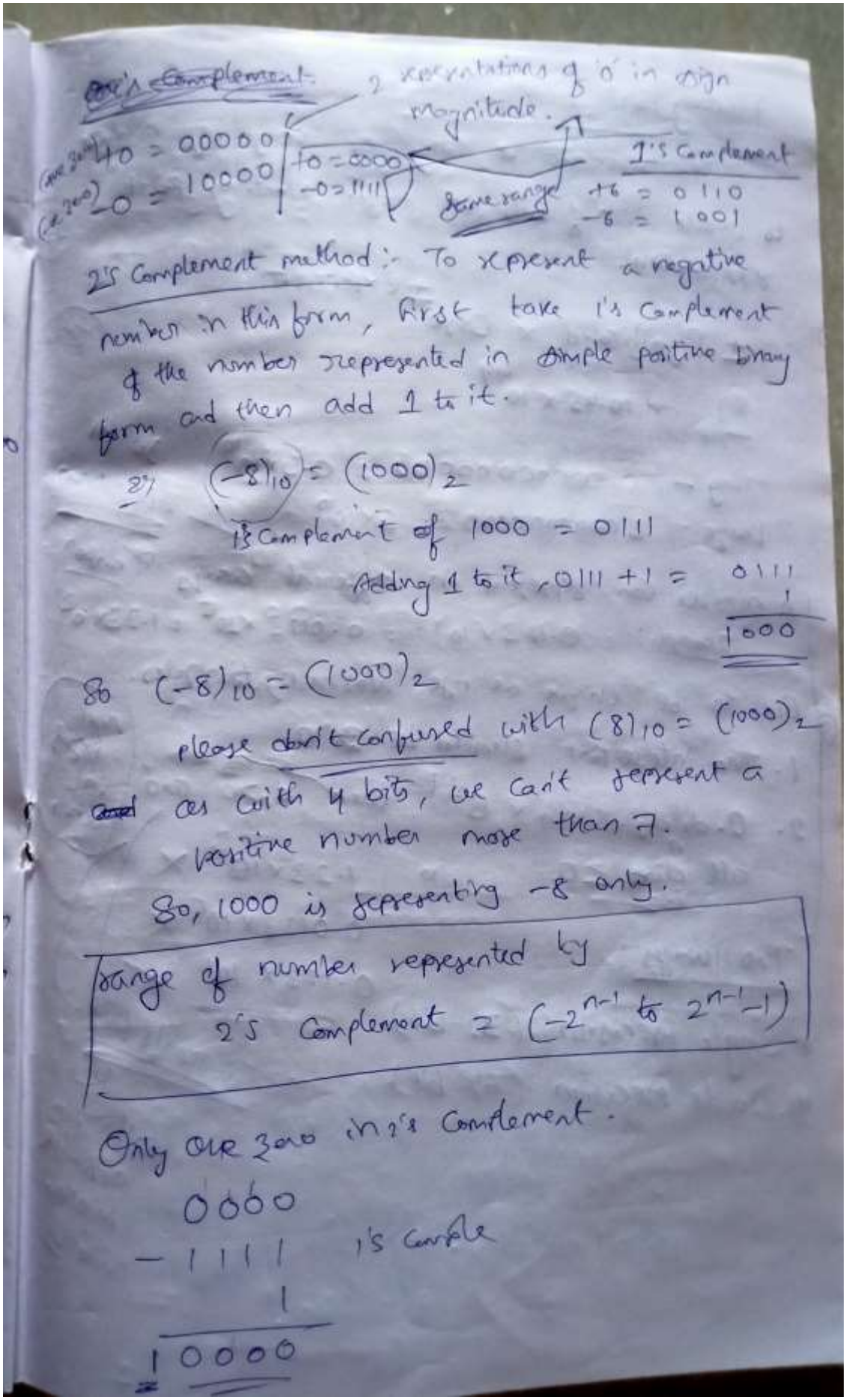
- '1' represents negative sign
- '0' represents positive sign

in sign magnitude of a n-bit number
- first bit is sign
- (n-1) bits for magnitude of a number.

Eg: $+25 = 0\ 1\ 1\ 0\ 0\ 1$ (1100 1 = 25)

$-25 = 1\ 1\ 1\ 0\ 0\ 1$ (1100 1 = 25)

range of numbers in sign magnitude = $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$



2's Complement method :- To represent a negative number in this form, first take 1's complement of the number represented in simple positive binary form and then add 1 to it.

2) $(-8)_{10} = (1000)_2$

1's Complement of 1000 = 0111

Adding 1 to it, 0111 + 1 = 1000

So $(-8)_{10} = (1000)_2$

please don't confused with $(8)_{10} = (1000)_2$

as with 4 bits, we can't represent a positive number more than 7.

So, 1000 is representing -8 only.

Range of number represented by 2's Complement = $(-2^{n-1}$ to $2^{n-1}-1)$

Only one zero in 2's Complement.

0000

-1111 1's Comple

1

10000

Floating Point Representation:

To represent very tiny or very big numbers to reduce no. of bits we prefer Floating Point representation.

$\pm$ Significand $\times$ Base $^{\pm \text{Exponent}}$

Eg $+10.45 \times 10^{44}$

Tiny – $0.0000000000005 = 0.5 \times 10^{-10}$

Large – $5000000000 = 5 \times 10^{10}$

Just for representation don't confuse

A number can represent in different ways

Example: $0.123 \times 10^4 = 0.0123 \times 10^5 = 1.23 \times 10^3$

But to be unique we use Normalization rules.

1. The integer part should be zero.
2. $0.d_1 d_2 \ldots d_n \times B^{\pm E}$ then $d_1 > 0$ and all $d_i \geq 0$ $\quad i=2$

1.23×10^3 ✗

0.0123×10^5 ✗

0.123×10^4 ✓

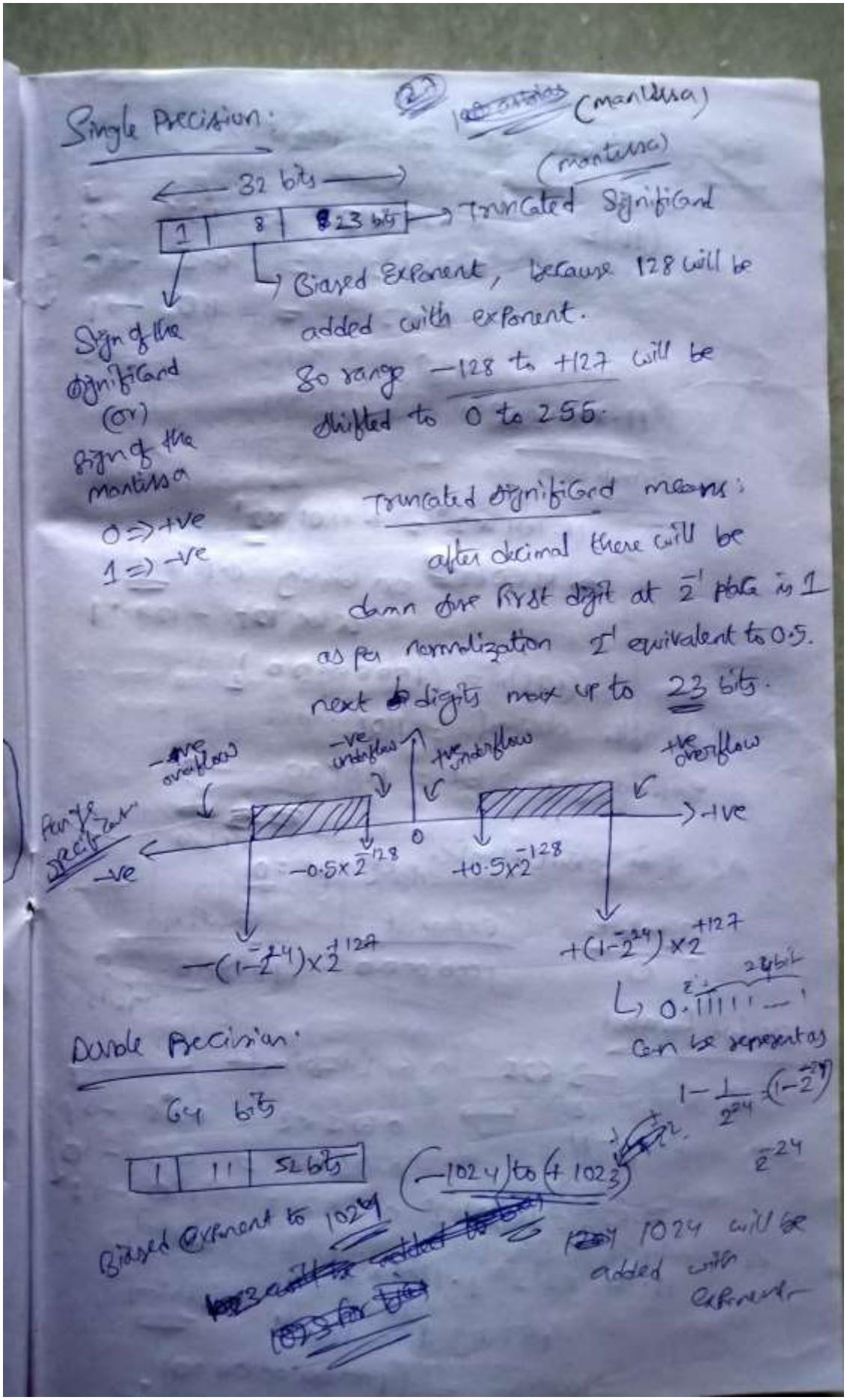
Tutorials point

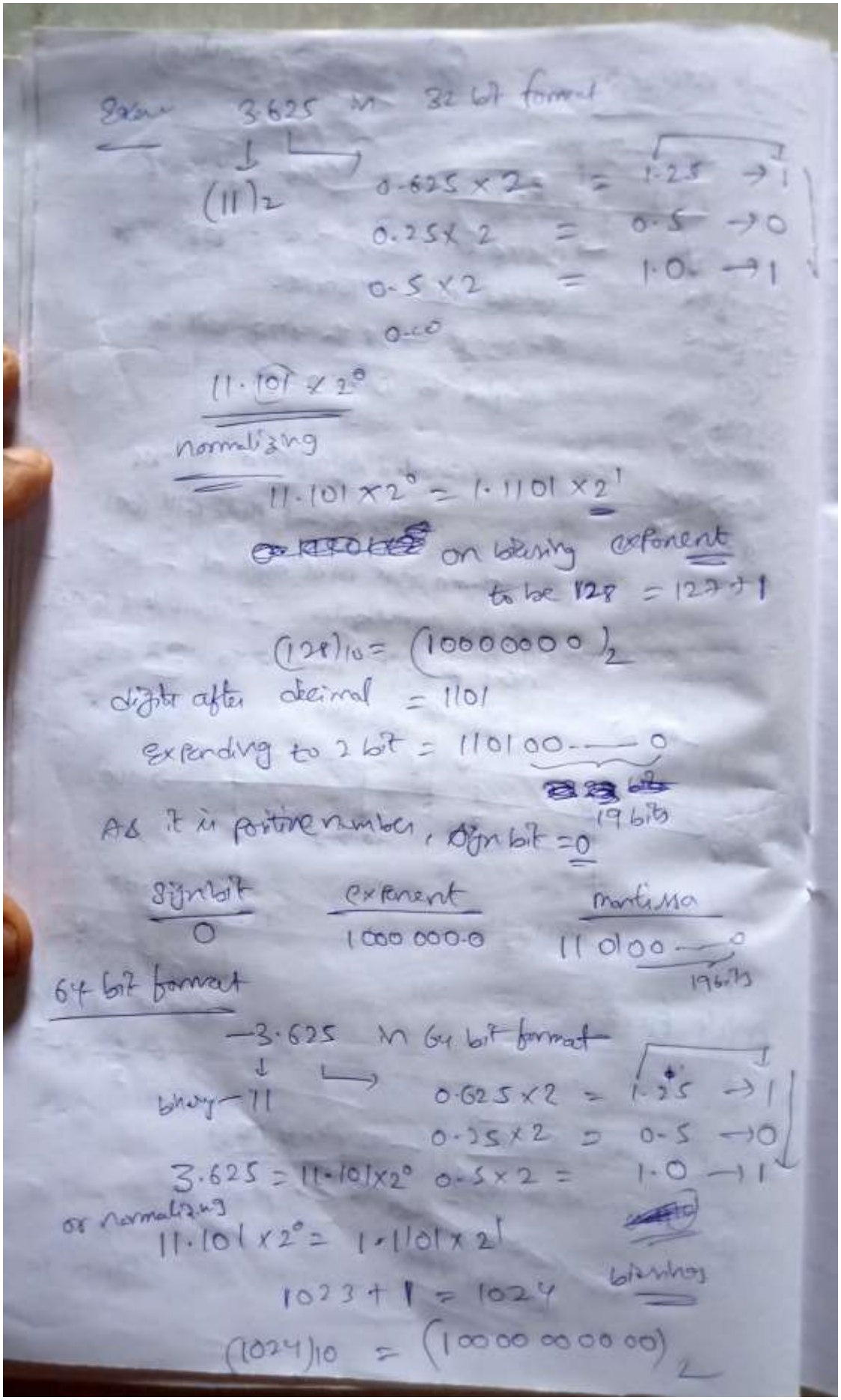
Two ways:

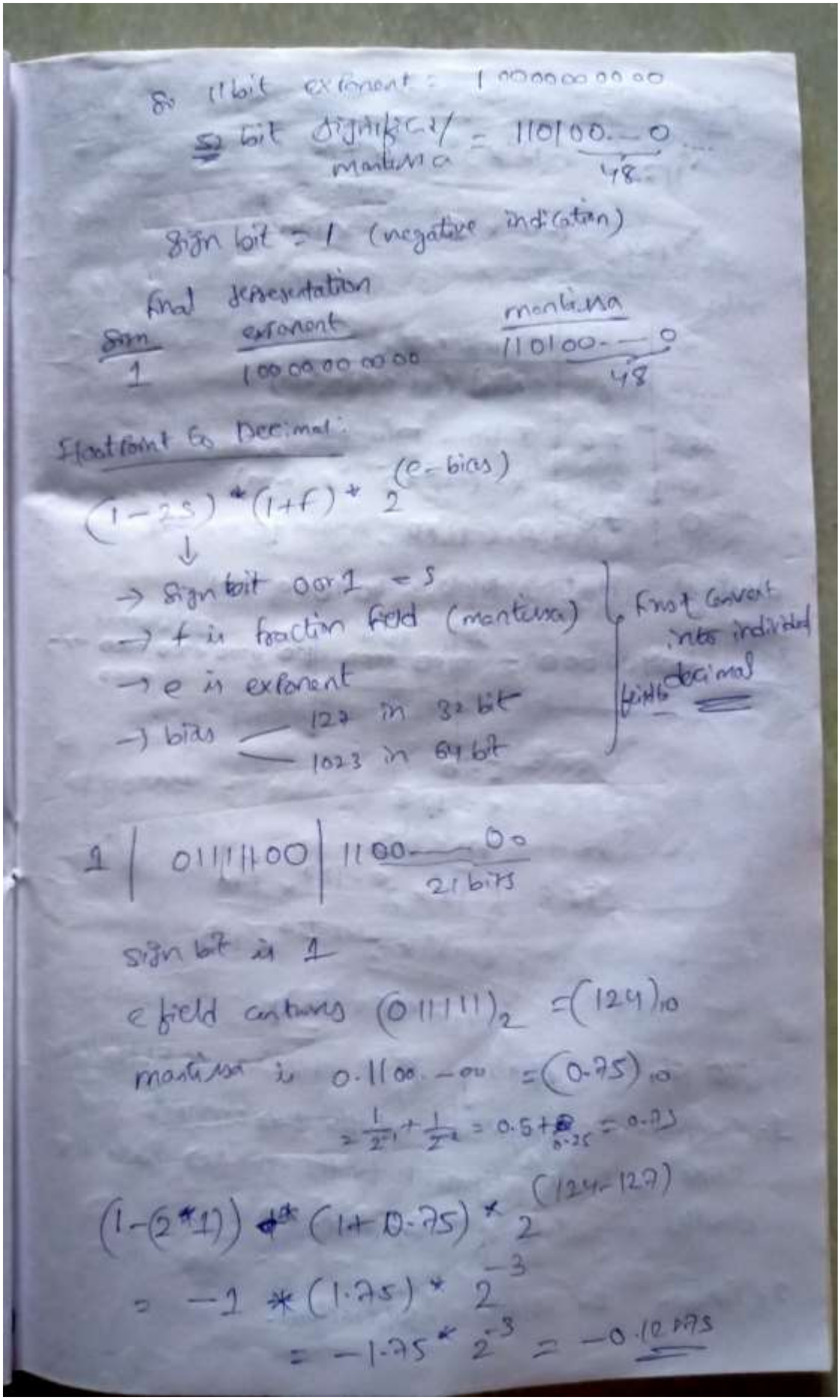
* Single Precision (32-bit)
* Dable Precision (64-bit)

next digit is zero

follows opposite motion anticlockwise


Single Precision:
(mantissa)
32 bits
1
8
23 bits
Truncated Significand
(mantissa)
Biased Exponent, because 128 will be added with exponent.
So range −128 to +127 will be shifted to 0 to 256.
Sign of the significand (or) Sign of the mantissa
0 => +ve
1 => −ve
Truncated significand means:
after decimal there will be
down the first digit at 2^{-1} place is 1
as per normalization 1 equivalent to 0.5.
next 23 digits max up to 23 bits.
−ve overflow
−ve underflow
+ve underflow
+ve overflow
+ve
−ve
0
-0.5×2^{-128}
$+0.5\times2^{-128}$
$-(1-2^{-24})\times2^{+127}$
$+(1-2^{-24})\times2^{+127}$
$0.\overbrace{111\cdots1}^{23\text{bit}}$
Can be represented as
$1-\frac{1}{2^{24}}$
$(1-2^{-24})$
2^{-24}
Double Precision:
64 bits
1
11
52 bits
Biased Exponent to 1024
(−1024) to (+1023)
1024 will be added with exponent


Ex: 3.625 in 32 bit format
(11)2
0.625 x 2 = 1.25 → 1
0.25 x 2 = 0.5 → 0
0.5 x 2 = 1.0 → 1
0.0
11.101 x 2^0
normalizing
11.101 x 2^0 = 1.1101 x 2^1
on biasing exponent
to be 128 = 127 + 1
(128)10 = (10000000)2
digits after decimal = 1101
expanding to 23 bit = 11010000...0
19 bits
As it is positive number, sign bit = 0
sign bit | exponent | mantissa
0 | 1000000 | 11010000...0
19 bits
64 bit format
-3.625 in 64 bit format
binary - 11
0.625 x 2 = 1.25 → 1
0.25 x 2 = 0.5 → 0
0.5 x 2 = 1.0 → 1
3.625 = 11.101 x 2^0
or normalizing
11.101 x 2^0 = 1.1101 x 2^1
1023 + 1 = 1024
biasing
(1024)10 = (10000000000)2



8b 11bit exponent : 10000000000
52 bit significand / mantissa = 11010000...0 (48 zeros)

Sign bit = 1 (negative indication)

final representation

Sign	exponent	mantissa
1	10000000000	11010000...0 (48 zeros)

Floatpoint to Decimal:

$(1-2S) * (1+f) * 2^{(e-bias)}$

→ Sign bit 0 or 1 = S
→ f is fraction field (mantissa)
→ e is exponent
→ bias — 127 in 32 bit
— 1023 in 64 bit

First Convert into individual fields decimal

1 | 01111100 | 1100...00 (21 bits)

sign bit is 1

e field contains $(01111100)_2 = (124)_{10}$

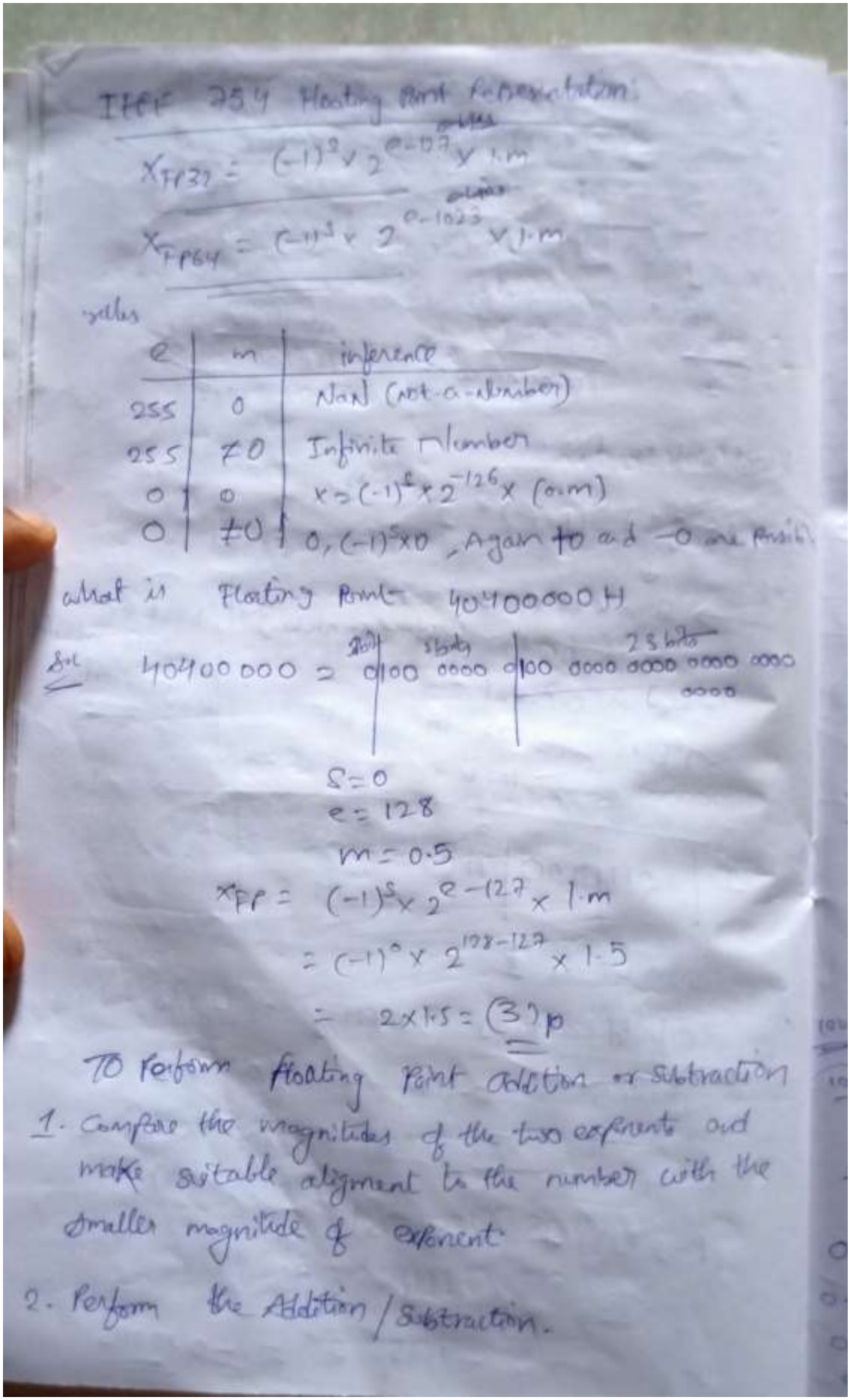
mantissa is $0.1100...00 = (0.75)_{10}$

$= \frac{1}{2^1} + \frac{1}{2^2} = 0.5 + 0.25 = 0.75$

$(1-(2*1)) * (1+0.75) * 2^{(124-127)}$

$= -1 * (1.75) * 2^{-3}$

$= -1.75 * 2^{-3} = -0.21875$



IEEE 754 Floating point representation:

$X_{FP32} = (-1)^s \times 2^{e-127} \times 1.m$

$X_{FP64} = (-1)^s \times 2^{e-1023} \times 1.m$

values

e	m	inference
255	0	NaN (not a number)
255	≠0	Infinite number
0	0	$x = (-1)^s \times 2^{-126} \times (0.m)$
0	≠0	0, $(-1)^s \times 0$, Again to add -0 are possible

what is Floating Point 40400000 H

Sol 40400000 = 0100 0000 0100 0000 0000 0000 0000 0000

s = 0

e = 128

m = 0.5

$X_{FP} = (-1)^s \times 2^{e-127} \times 1.m$

$= (-1)^0 \times 2^{128-127} \times 1.5$

$= 2 \times 1.5 = (3)_D$

TO Perform floating Point Addition or Subtraction

1. Compare the magnitudes of the two exponents and make suitable alignment to the number with the smaller magnitude of exponent

2. Perform the Addition / Subtraction.

3. Perform normalization by shifting the resulting mantissa and adjusting the resulting exponent.

Example: Add 1.1100×2^4 and 1.1000×2^2

1. **Alignment:** 1.1000×2^2 has to be aligned as 0.0110×2^4

2. **Addition:** Add two numbers to get

~~1.1100~~
~~1.1000~~
~~10 100~~

$$\begin{array}{r} 1.1100 \\ 0.0110 \\ \hline 10.0010 \end{array} \times 2^4$$

3. **Normalization:** Final normalized result is 0.1000×2^6 (Assuming 4 bits are followed after radix point).

Eg: $x = 9.75$, $y = 0.5625$

9.75
↓ → $0.75 \times 2 \rightarrow 1.50 \times 1$
1001. $0.5 \times 2 \Rightarrow 1.00 \times 1$

$1001.11 \times 2^0 = 10.0111 \times 2^2$

~~0.10011 × 2^4~~ $= 1.00111 \times 2^3$

1001.1100
100.11101
1.00111×2^3

exponent 261 0111 0 00... 19 bits

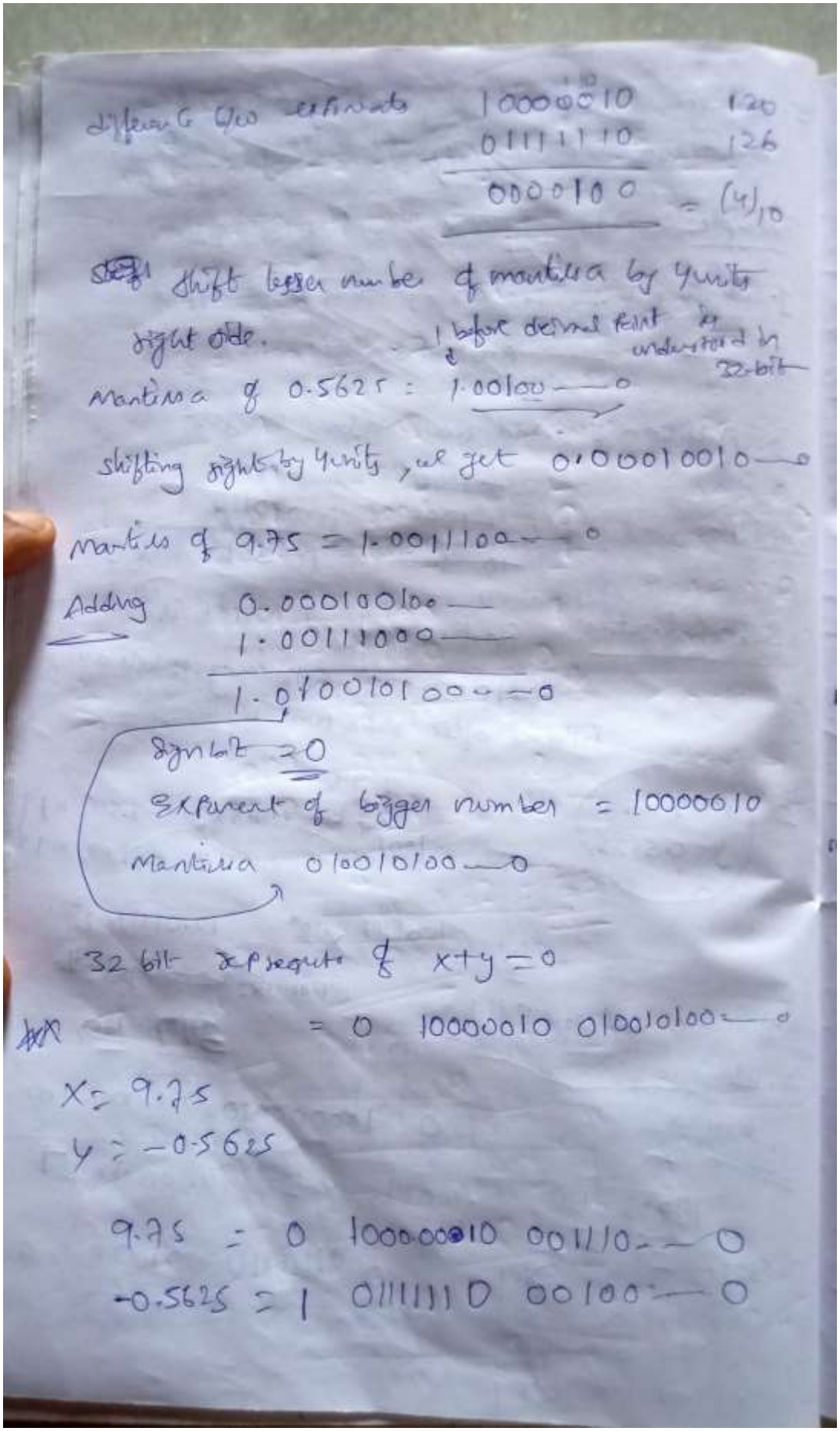
= ~~3+127~~ = 130

0 10000010 0011000— 00

127+3 = 130 $(0.1001)_2$ $(1.001) \times 2^{-1}$

$0.5625 \times 2 \rightarrow 1.125 \rightarrow 1$
$0.125 \times 2 \rightarrow 0.25 \rightarrow 0$
$0.25 \times 2 \rightarrow 0.5 \rightarrow 0$
$0.5 \times 2 \rightarrow 1.0 \rightarrow 1$

0 01111110 00100 0...— 0



Difference between exponents 10000010 130
01111110 126
0000100 = $(4)_{10}$

Step 1 Shift lesser number of mantissa by 4 units right side.

Mantissa of 0.5625 : 1.00100——0 (1 before decimal point is understood in 32-bit)

shifting right by 4 units, we get 0.000100100——0

Mantissa of 9.75 = 1.0011100——0

Adding 0.000100100——
1.00111000——
1.0100101000——0

Sign bit = 0

Exponent of bigger number = 10000010

Mantissa 00100100——0

32 bit representation of x+y = 0

= 0 10000010 00100100——0

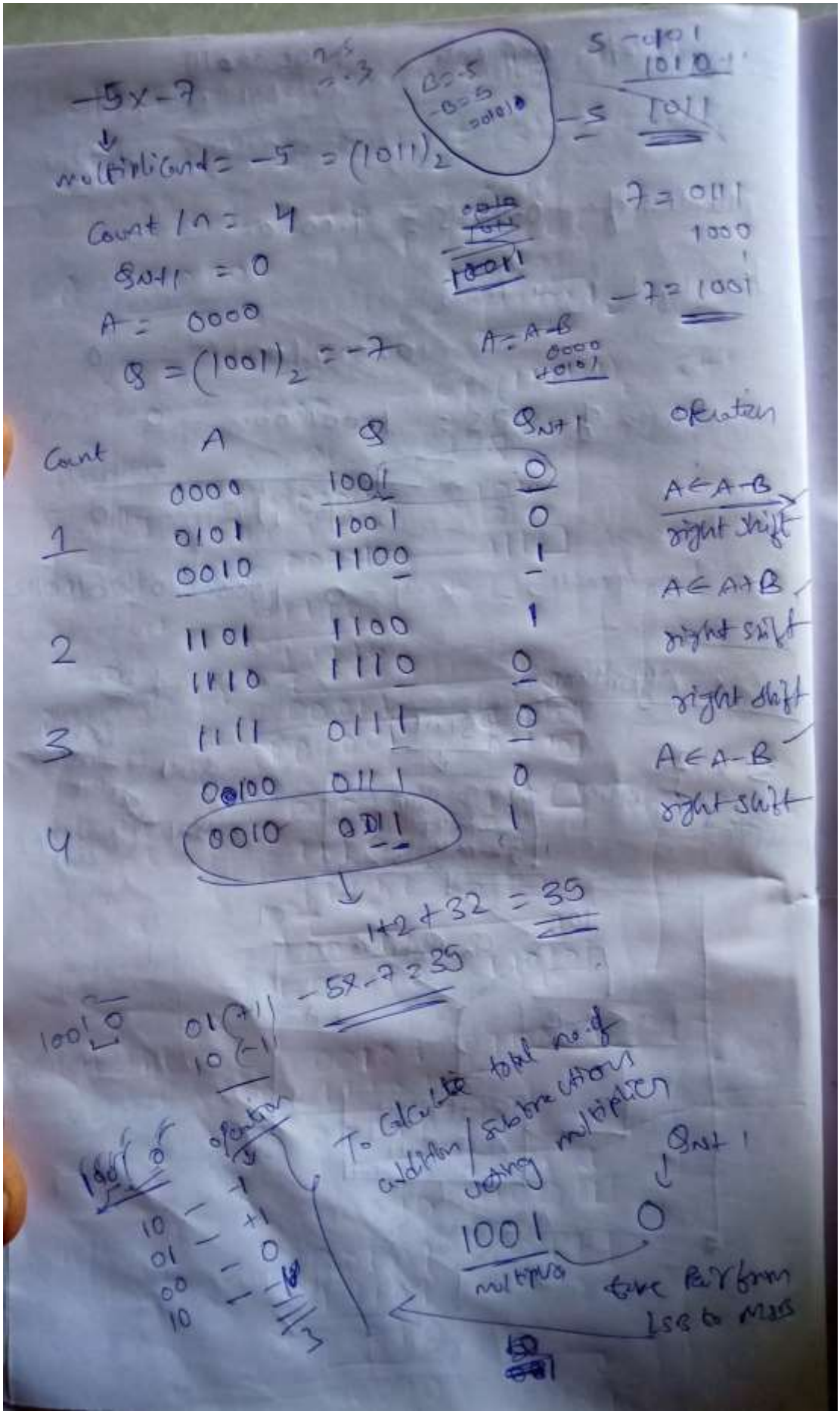
X = 9.75

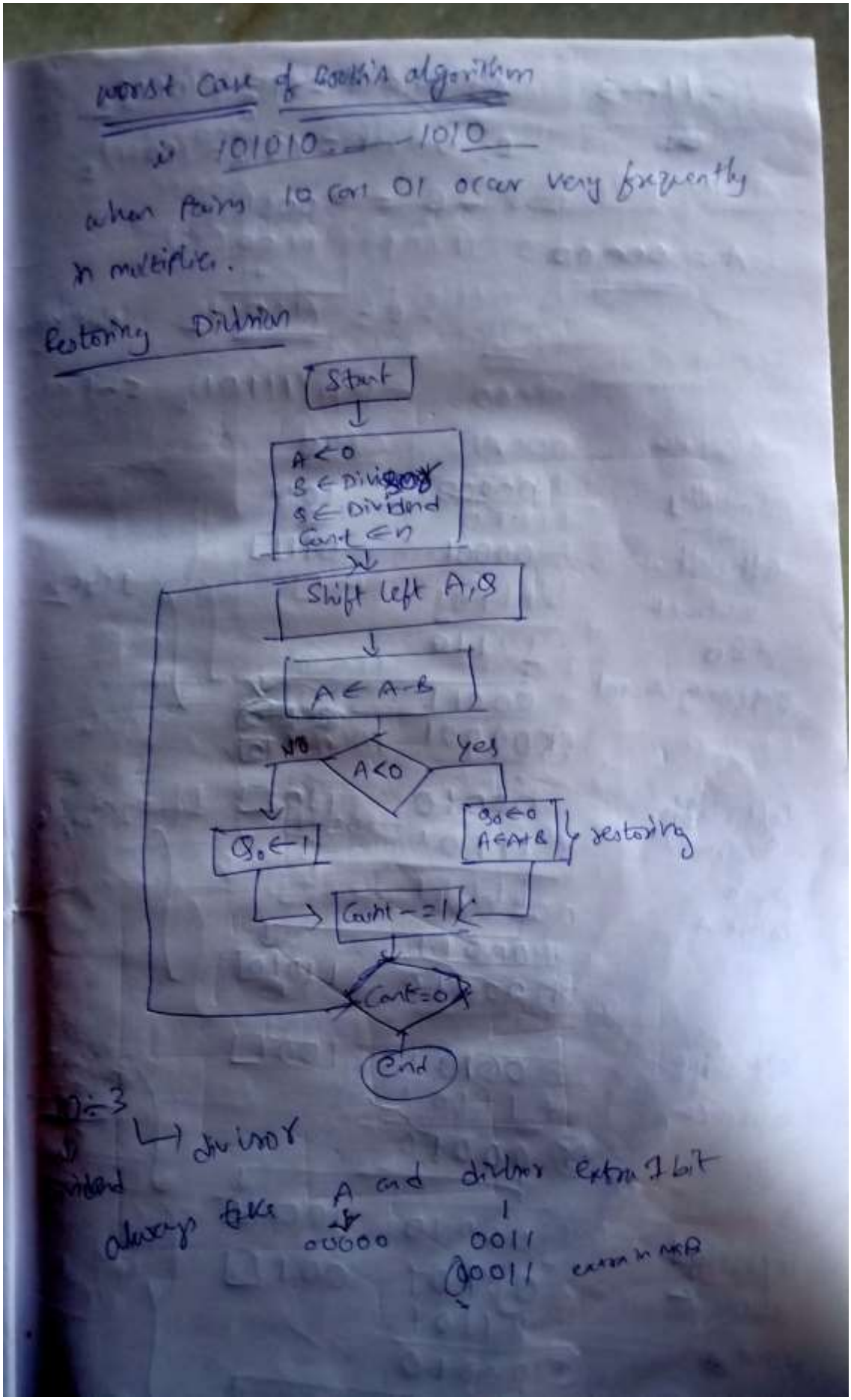
Y = −0.5625

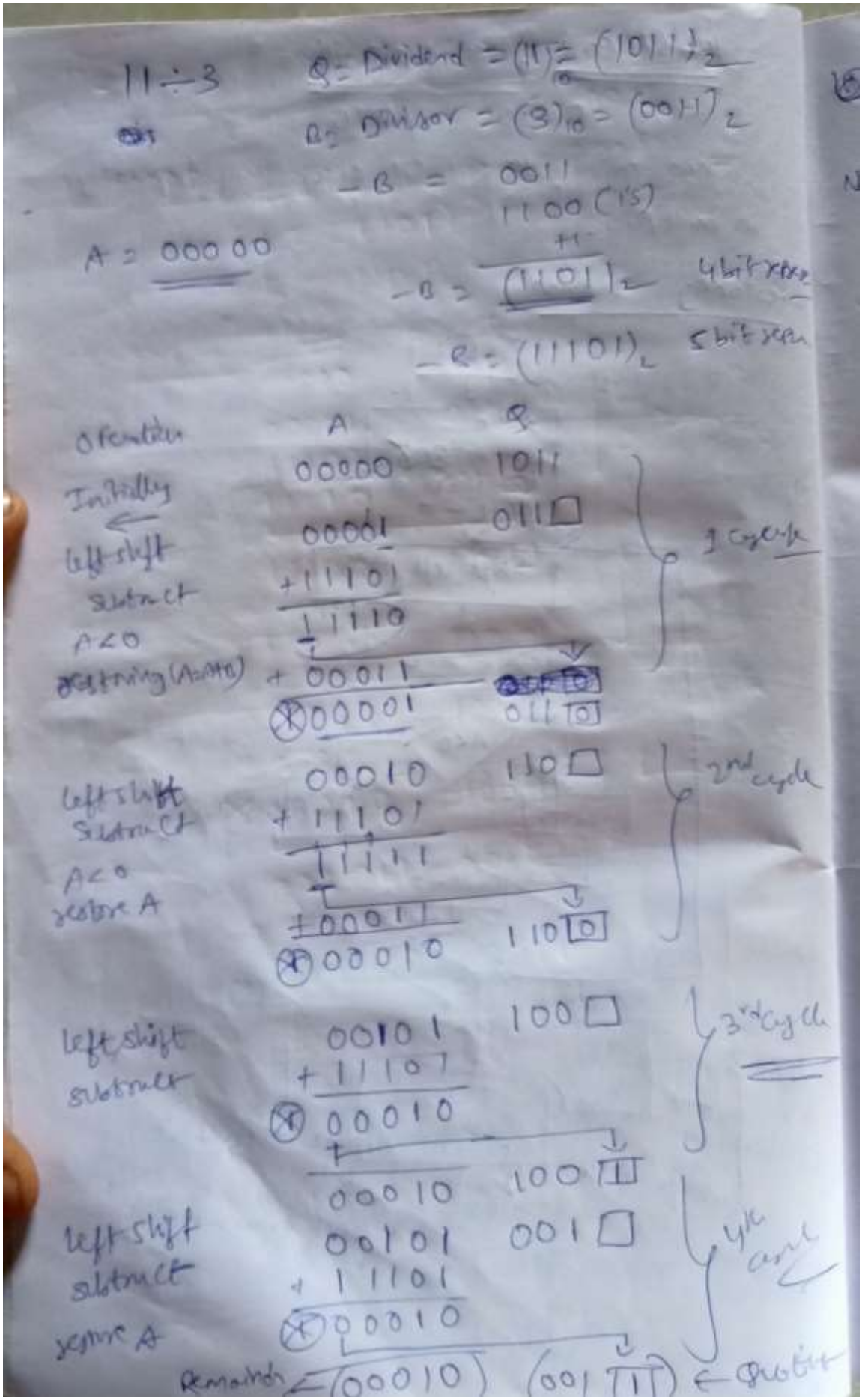
9.75 = 0 10000010 00111——0

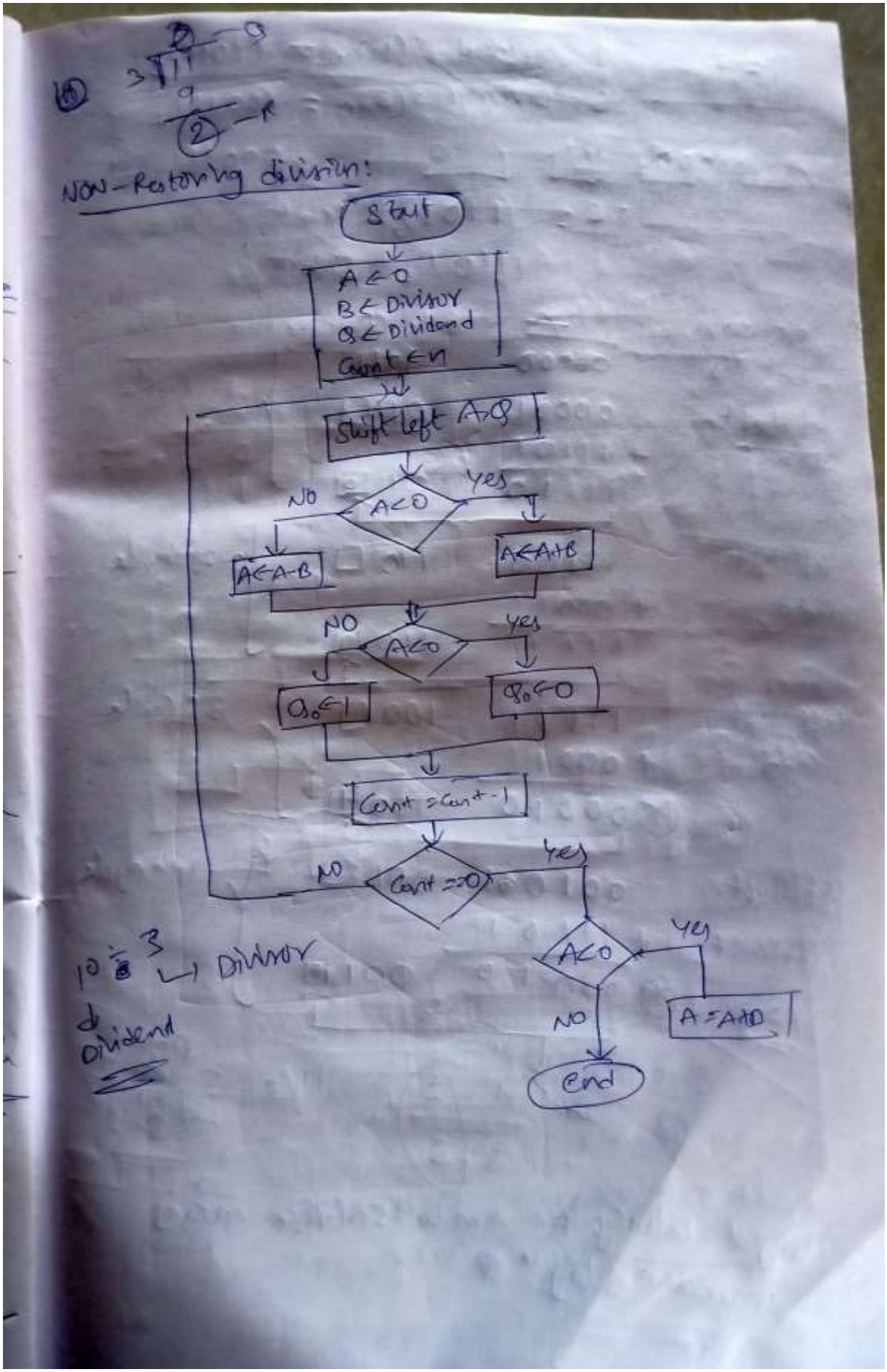
−0.5625 = 1 01111110 00100——0

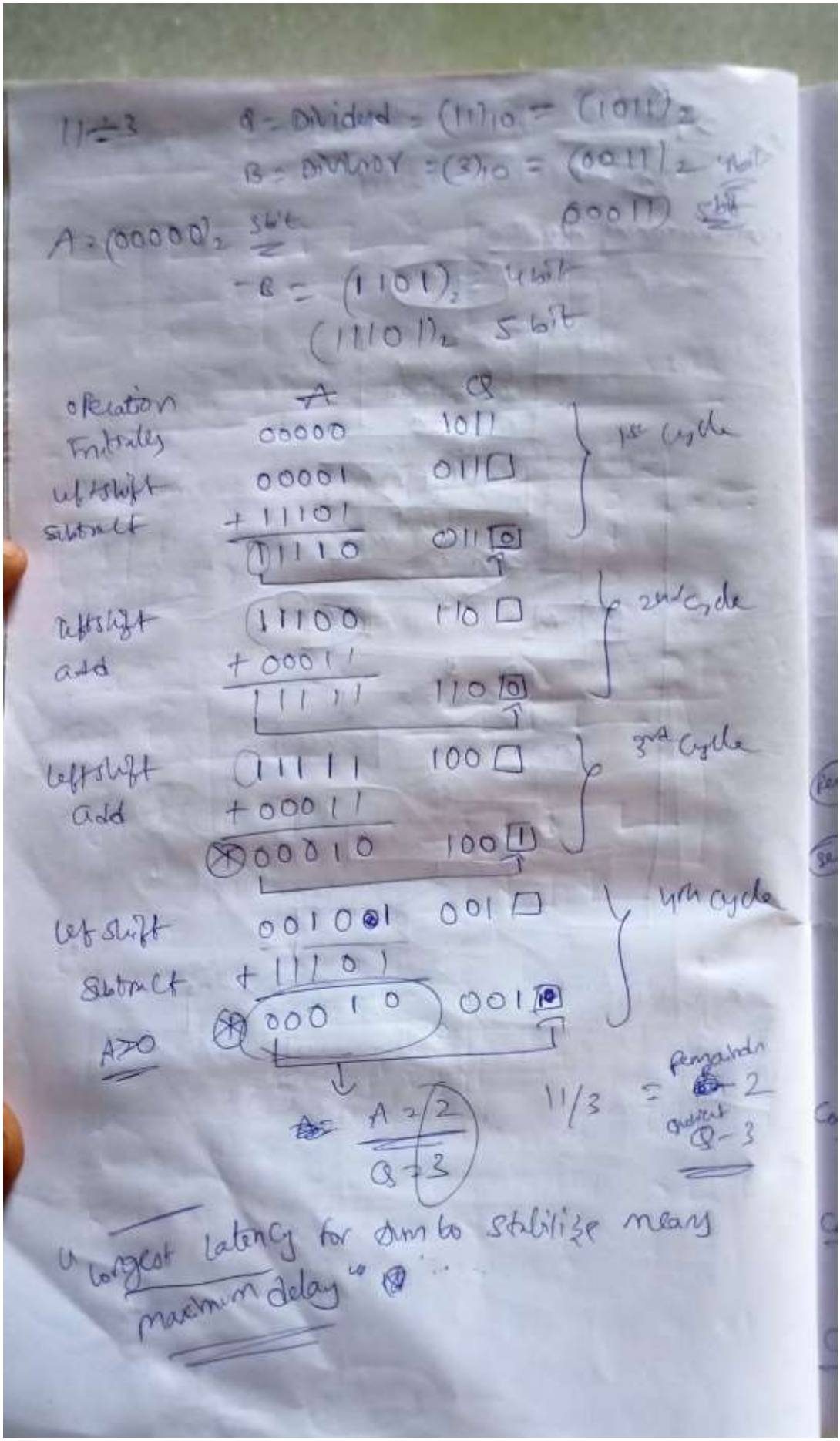

difference b/w exponents
1000 0010
011 111 10
0000 0100 (4)10
mantissa of −0.5625 = 1.0010 0 — 0
shift by 4 units
0.0001 0010 — 0
mantissa of 0.75 = 1.0011 00 — 0
1.0010 110 — 0
So finally = x − y = 0 1000 0010 0010 0110
Booth's Algorithm:
Count ← no. of bits
3 × 2
multiplied
multiplier
Start
A ← 0, Q−1 ← 0
B ← multiplicand
Q ← multiplier
count ← n
Previous bit
(same may use)
10
01
Q0, Q−1
A ← A − B
A ← A + B
Shift Right
A, Q, Q−1
Count = Count − 1
NO
Count = 0?
YES
END


-5 x -7
multiplicand = -5 = (1011)2
Count / n = 4
Q n+1 = 0
A = 0000
Q = (1001)2 = -7
5 = 0101
1010
-5 = 1011
7 = 0111
1000
-7 = 1001
A = A-B
0000
+0101
Count
A
Q
Q n+1
operation
0000
1001
0
A ← A-B
1
0101
1001
0
right shift
0010
1100
1
A ← A+B
2
1101
1100
1
right shift
1110
1110
0
right shift
3
1111
0111
0
A ← A-B
0100
0111
0
right shift
4
0010
0011
1
1+2+32 = 35
-5x-7 = 35
1001 0
01 (+1)
10 (-1)
operation
1001 0
10 - +1
01 - 0
00 - 0
10 - -1
3
To Calculate total no of addition / subtraction during multiplication
1001
0
Q n+1
multiplier
Perform
LSB to MSB


Worst Case of Booth's algorithm
i) 101010 ---- 1010
when pairs 10 or 01 occur very frequently
in multiplier.
Restoring Division
Start
A<0
S ← Divisor
S ← Dividend
Count ← n
Shift left A,Q
A ← A-B
NO
A<0
yes
Q0 ← 1
Q0 ← 0
A ← A+B
restoring
Count - = 1
Count=0
End
Q÷3
divisor
always take
A
00000
and divisor extra 1 bit
1
0011
00011


11 ÷ 3
Q = Dividend = (11)10 = (1011)2
B = Divisor = (3)10 = (0011)2
-B = 0011
1100 (1's)
+1
-B = (1101)2
4bit 2's
-B = (11101)2
5bit 2's
A = 00000
Q
Initially
00000 1011
Left shift 00001 011□
Subtract +11101
A<0 11110
restoring (Add B) + 00011
00001 011 0
1 cycle
Left shift 00010 110□
Subtract +11101
A<0 11111
restore A +00011
00010 110 0
2nd cycle
Left shift 00101 100□
Subtract +11101
00010 100 1
3rd cycle
Left shift 00101 001□
Subtract +11101
restore A 00010
4th cycle
Remainder (00010) (001 11) ← Quotient


Start
A←0
B←Divisor
Q←Dividend
Count←n
Shift left A,Q
A<0
NO
yes
A←A-B
A←A+B
A<0
NO
yes
Q0←1
Q0←0
Count = Count-1
Count = 0
NO
yes
A<0
yes
A=A+B
NO
End
10 ÷ 3
Divisor
Dividend


11÷3
Q = Dividend = (11)10 = (1011)2
B = Divisor = (3)10 = (0011)2
A = (00000)2 5bit
-B = (1101)2 4bit
(11101)2 5bit
Operation
Initially
Left shift
Subtract
Left shift
add
Left shift
Add
Left Shift
Subtract
A>0
1st cycle
2nd cycle
3rd cycle
4th cycle
A = 2
Q = 3
11/3 = Remainder 2
quotient Q = 3
longest latency for sum to stabilize means maximum delay

Sequential circuits

Combination circuit produces an output based on input variable only. Sequential circuit produces an output based on current input and previous input variables.

That means it include memory elements.

Latch :- Capable of storing 1-bit information.

2 types (Synchronous) (different) (less number of states)
- Asynchronous (faster) ← effected changes when change in input digital.
- Synchronous (slower) ← level output change state at the start of the output pulse and remains in that until next input or clock pulse.

↓ Flipflops, Counters, Registers, RAM, MOSFET/MEALY

SR- Latch

NOR

A	B	y
0	0	1
0	1	0
1	0	0
1	1	0

NOR

A	B	y
0	0	1
0	1	0
1	0	0
1	1	0

S	R	Q	$\bar{Q}$
0	0	memory (as before)	
0	1	0	1
1	0	1	0
1	1	not used	

Case-1: S=0, R=1, Q=0 & Q'=1

S=0, R=0, Q=0 Q'=1 (memory)

Case-2: S=1, R=0, Q=1 & Q'=0

S=0, R=0, Q=1, & Q'=1 (memory)

Case-3: S=1, R=1, Q=0 & Q'=0

S=0, R=0, Q=0 & $\bar{Q}$=1

Q=1 & $\bar{Q}$=0

S	R	Q	$\bar{Q}$
0	0	Not used	
0	1	1	0
1	0	0	1
1	1	memory	

Case-1: S=0, R=1, Q=1, Q'=0

S=1, R=1, Q=1 Q'=0

Case-2: S=1, R=0, Q=0, Q'=1

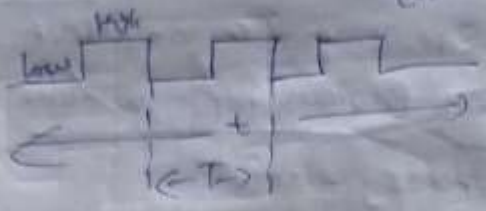
S=1, R=1, Q=0, Q'=1

Case-3: S=0, R=0, Q=1, Q'=1

S=1, R=1, Q=1 & Q'=0

Q=0 & Q'=1

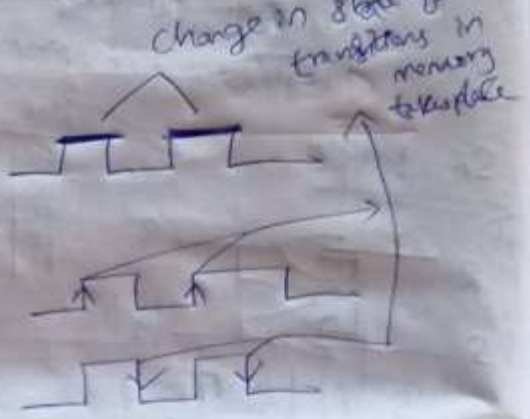
Duty Cycle = Ratio of time when signal is High / total time



Duty cycle = $t/T = 1/2 = 50\%$

Clock: It decides time of the input pulses.
It's a signal varies low to high as high to low.
(Control input)

Triggering methods:



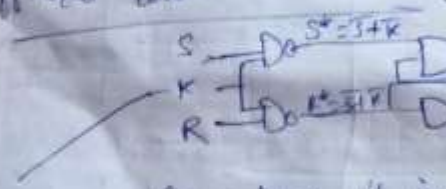
Level triggering

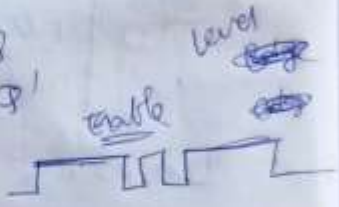
Edge triggering

+ve edge

-ve edge

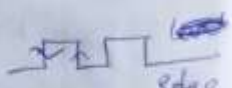
Diff b/w Latch & Flipflop:





K acts Enable when it is Latch

K acts as Clock when it is Flipflop

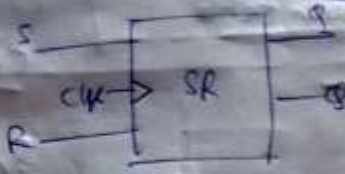


SR FLIPFLOP :

S^*	R^*	Q	Q'
0	0	not used	
0	1	1	0
1	0	0	1
1	1	Memory	

$S^* = \overline{S + CLK}$

$R^* = \overline{R + CLK}$



Q_n ← present state

next state

Truth Table

CLK	S	R	Q	Q'
0	x	x	memory	
1	0	0	memory	
1	0	1	1	0
1	1	0	0	1
1	1	1	not used	

⇒

CLK	S	R	Q_{n+1}
0	x	x	Q_n
1	0	0	Q_n
1	0	1	0 (reset)
1	1	0	1 (set)
1	1	1	Invalid

Characteristic table:

Present state + inputs → next state

CLK = 1

Q_n	S	R	Q_{n+1}
0	0	0	0 (Q_n)
0	0	1	0
0	1	0	1
0	1	1	X
1	0	0	1 (Q_n)
1	0	1	0
1	1	0	1
1	1	1	X

$Q_{n+1} = S + Q_n\overline{R}$ (using K-map)

Excitation Table:

Q_n	Q_{n+1}	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

present + next state

↓

inputs

D-Flip Flop (Delay)

Truth table

clk	D	Q_{n+1}
0	X	Q_n
1	0	0
1	1	1

Characteristic Table

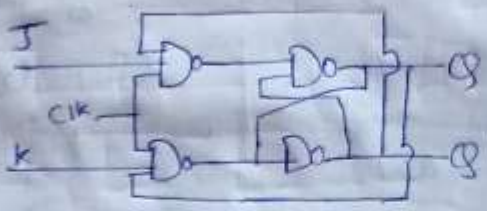
Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1

$Q_{n+1} = D$

Excitation Table

Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

JK Flip flop



clk = 0, memory

clk = 1, J = 1, K = 0, Q = 1, $\bar{Q}$ = 0

clk = 1, J = 0, K = 1, Q = 0, $\bar{Q}$ = 1

clk = 1, J = 1, K = 1, assume Q = 0, Q' = 1

Racing around

Q = 0, 1, 0, 1 ... Q_{n+1}

$\bar{Q}$ = 1, 0, 1, 0 ... $\bar{Q}_{n+1}$

Truth table

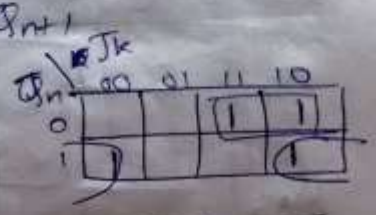
Clk	J	K	Q_{n+1}
0	x	x	Q_n (memory)
1	0	0	Q_n
1	0	1	0
1	1	0	1
1	1	1	$\overline{Q_n}$ (toggle)

Characteristic Table

Q_n	J	K	Q_{n+1}
0	0	0	$Q_n = 0$
0	0	1	0
0	1	0	1
0	1	1	$\overline{Q_n} = 1$
1	0	0	$1 = Q_n$
1	0	1	0
1	1	0	1
1	1	1	$0 = \overline{Q_n}$

Excitation Table

Q_n	Q_{n+1}	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0



$= \overline{Q_n} J + Q_n \overline{K}$

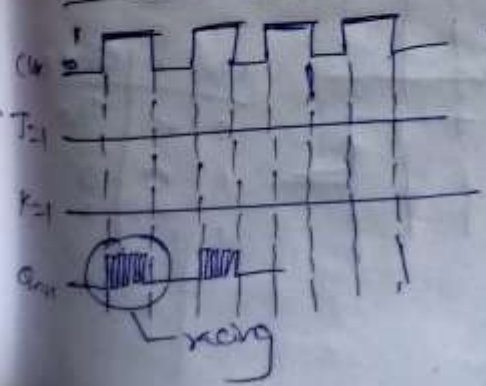
$= Q_{n+1}$

$= \overline{Q_{n+1}}$

Toggling can be controlled (Race around / Const. ← Uncontrolled)

Race around can be Convert into toggle and try to Control.

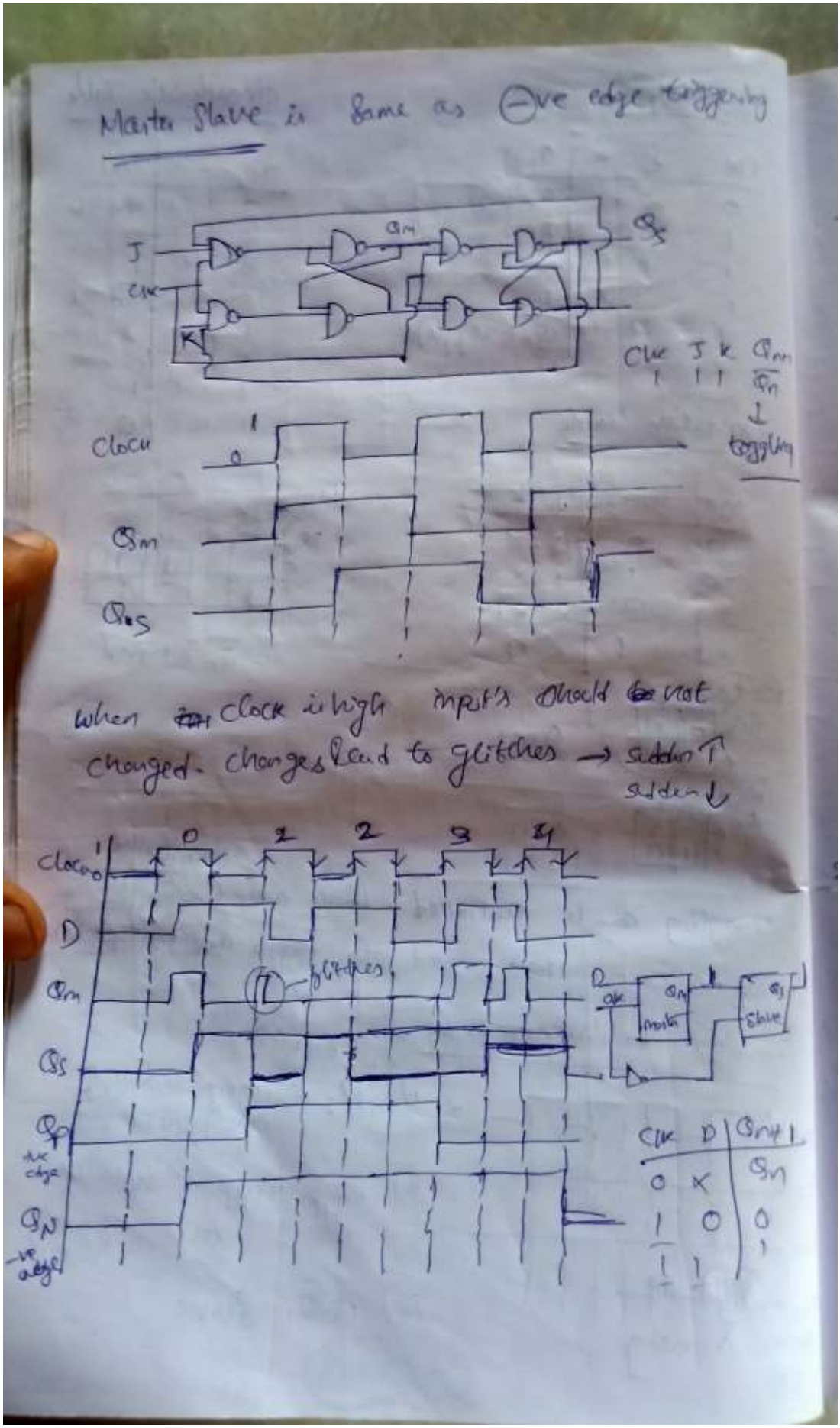
Conditions to eliminate Racing:



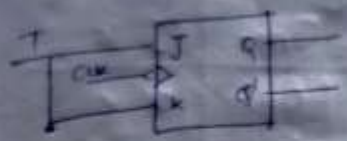
× i) $T/2 <$ Propagation delay of Flipflop (not practical)

ii) Edge triggering instead of level triggering

iii) Master - Slave


Master Slave is same as -ve edge triggering
J
CLK
K
Clock
Qm
Qs
CLK J K Qm
1 1 1 Qm
toggling
when clock is high inputs should not changed. changes lead to glitches → sudden ↑ sudden ↓
D
glitches
master
slave
CLK D Qn+1
0 X Qn
1 0 0
1 1 1

T- Flipflop (Toggle flipflop) only Toggle action



Truth Table

Clk	T	Q_{n+1}
0	x	Q_n (memory)
1	0	Q_n (memory)
1	1	$\bar{Q}_n$ (Toggle)

Characteristic Table:

Q_n	T	Q_{n+1}
0	0	$Q_n = 0$
0	1	$\bar{Q}_n = 1$
1	0	1
1	1	0

$Q_{n+1} = Q_n \oplus T$

Excitation Table

Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Flip Flop Conversion:

1. Identify available and required Flipflop.
2. Make characteristic table for required flip flop.
3. Make excitation table for available flipflop.
4. Write boolean expression for available ff.
5. Draw the circuit.

Example Jk to D

avail ff = Jk

req. ff = D

~~Characteristic of D~~

~~Q_n~~	~~D~~	~~Q_{n+1}~~
0	x	memory
+	0	0
+	1	1

Characteristic of D

Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1

excitation of JK

Q_n	Q_{n+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Q_n	D	Q_{n+1}	J	K
0	0	0	0	X
0	1	1	1	X
1	0	0	X	1
1	1	1	X	0

for J

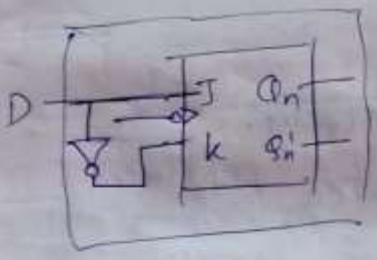
Q_n \ D	0	1
0	0	1
1	X	X

$J = D$

for K

Q_n \ D	0	1
0	X	X
1	1	0

$K = D'$



SR to JK

avail ff = SR

req ff = JK

excitation of SR

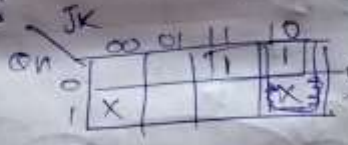
Q_n	Q_{n+1}	SR
0	0	0X
0	1	10
1	0	01
1	1	X0

Characteristic of JK

Q_n	J	K	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Q_n	J	K	Q_{n+1}	S	R
0	0	0	0	0	X
0	0	1	0	0	X
0	1	0	1	1	0
0	1	1	1	1	0
1	0	0	1	X	0
1	0	1	0	0	1
1	1	0	1	X	0
1	1	1	0	0	1

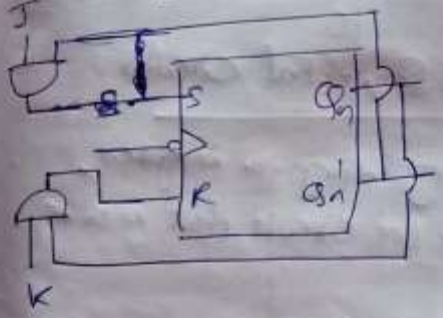
S



$S = \bar{Q}_n J$

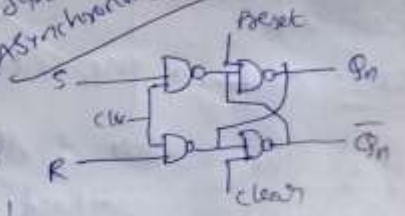
R

$R = Q_n K$



Synchronous inputs = S, R, J, K, D, T
Asynchronous inputs = preset & clear

Preset & clear inputs

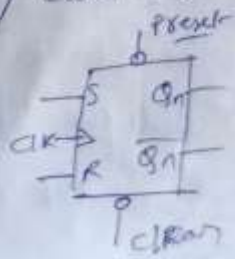


(Asynchronous or overriding)

preset = 0 $\Rightarrow$ $Q_n = 1$ and $\bar{Q}_n = 0$

clear = 0 $\Rightarrow$ $Q_n = 0$ and $\bar{Q}_n = 1$

What ever the value of clock and synchronous inputs the value overrided by using preset/clear (asynchronous) inputs.



preset	clear	Q_n
0	0	Not used
0	1	1 (preset)
1	0	0
1	1	ff will perform normally

State Diagrams

For JK Flipflop:

Q_n	J	K	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

$Q_{n+1} = \bar{Q}_n J + Q_n \bar{K}$

State eqn =

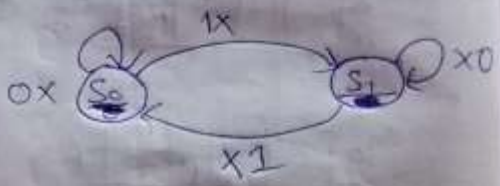
Present state + I/P

1 flipflop = 2 states

$2^1 = 2$ States

$S_0 = 0$

$S_1 = 1$



Design Procedure for clocked Sequential circuits

Step-1: A state diagram or timing diagram is given, which describes the behaviour of the circuit that is to be designed.

Step-2: Obtain the state table.

Step-3: The number of states can be reduced by state reduction method.

Step-4: Do state assignment (If required) Alphabets into binary.

Step-5: Determine the no. of flipflops required and assign letter symbols.

Step-6: Decide the type of flipflop used

Step-7: Derive the circuit excitation table from state table

Step-8: Obtain the expression for circuit output and flipflop input.

Step 4: Implement the circuit.

State reduction: If Next state and output are same of two present states then 1 will be considered other are reduced.

Example: AB|S, 00|0, 01|1, 10|1 — when Carry 0; AB|S, 11|0 — when Carry 1; AB|S, 01|0, 10|0, 11|1; AB|S, 00|1

Adder

AB	S	C
00	0	0
01	1	0
10	1	0
11	0	1

State assignment

Carry 0 — A

Carry 1 — B

State table

Present State	Next State, output (Sum)					
	$x_1=0$	$x_2=0$	$x_1=0$ $x_2=1$	$x_1=1$ $x_2=0$	$x_1=1$ $x_2=1$	
A — 0	A, 0		A, 1	A, 1	B, 0	
B — 1	A, 1		B, 0	B, 0	B, 1	

Replace assignments with numbers

Present State	Next State, output (Sum)			
	00	01	10	11
0	0, 0	0, 1	0, 1	1, 0
1	0, 1	1, 0	1, 0	1, 1

Separate outputs only in K map

Y — separate x_1x_2

(P) Present State	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$= P'x_1'x_2 + Px_1x_2' + Px_1'x_2' + \overline{Px_1x_2}$

$= P(x_1 \oplus x_2) + P'(x_1 \oplus x_2)$

$= P(x_1 \oplus x_2)' + P'(x_1 \oplus x_2)$

$= P \oplus (x_1 \oplus x_2)$

Ex: Circuit excitation table:

Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

D Flip Flop

Present State & Input					
P	x_1	x_2	P^+	ff i/p D	y
0	0	0	0	0	0
0	1	0	0	0	1
0	0	1	1	1	1
1	0	1	0	0	0
0	1	0	1	1	1
1	1	0	1	1	0
0	1	1	1	1	0
1	1	1	1	1	1

Future output of Ckt

$y = Px_1'x_2' + P'x_1'x_2 + P'x_1x_2' + Px_1x_2$

$= P \oplus x_1 \oplus x_2$

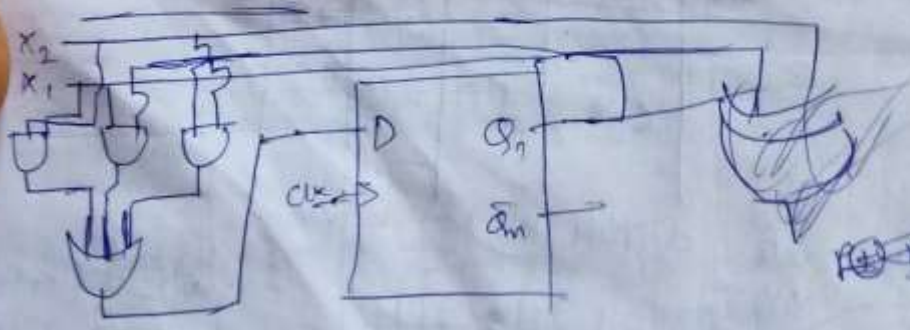
$D = P'x_1'x_2 + Px_1x_2' + P'x_1x_2 + Px_1x_2$

$= x_1x_2 + Px_1'x_2 + Px_1x_2'$

$= x_2x_1 + x_2P + Px_1x_2'$

$= x_1x_2 + x_1P + x_2P$

Circuit diagram



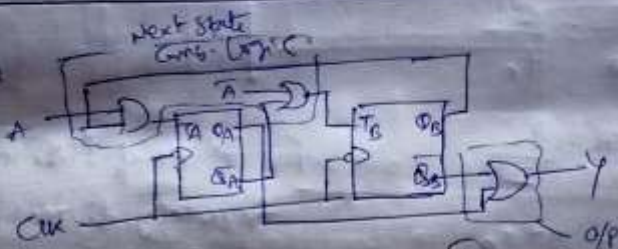
$P \oplus x_1 \oplus x_2$

Two models representing Synch. sequential circuits.

i) Moore circuit/moore state machine :-

o/p is function of present only

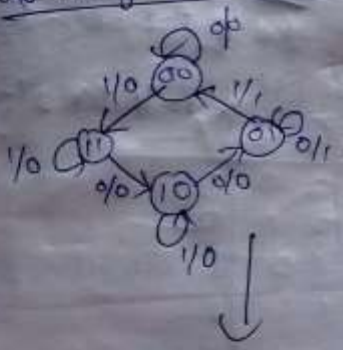
Example

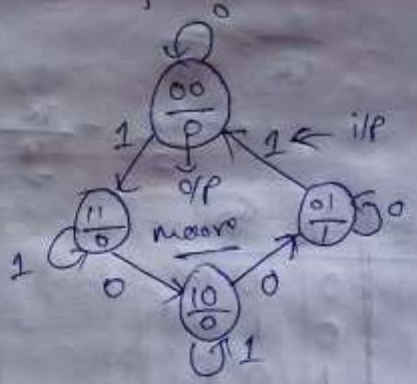


Clearly independent of input

$Y = \overline{Q_B} + Q_A$

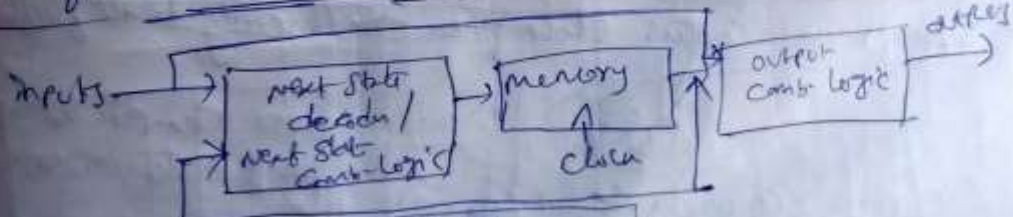
State diagram Conversion:



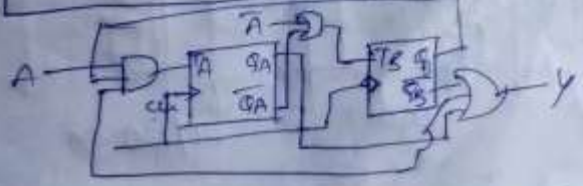


ii) Mealy circuit / Mealy State machine: the output is function of present state as well as inputs

No. of States are less for same logic Compared to moore

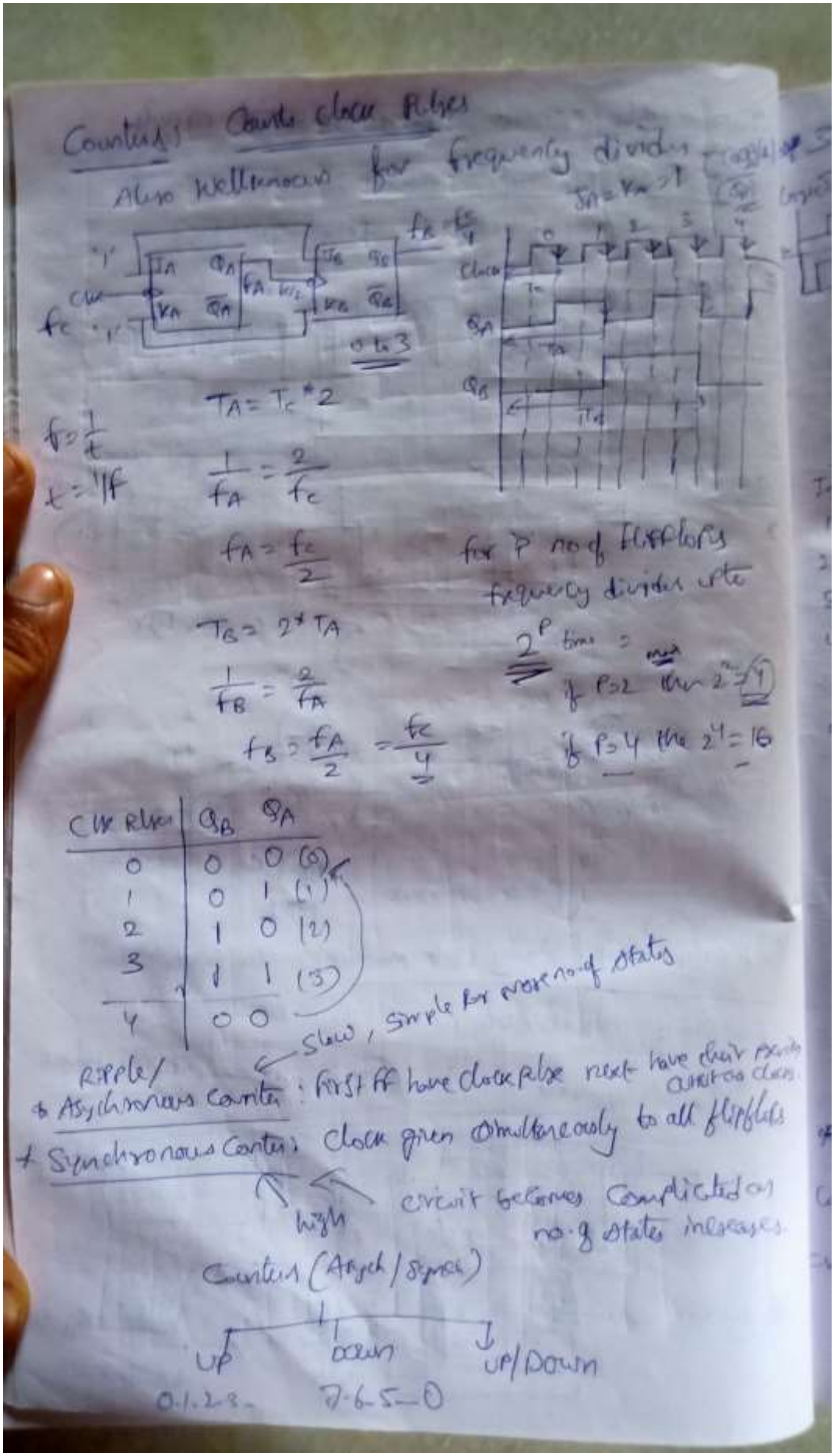


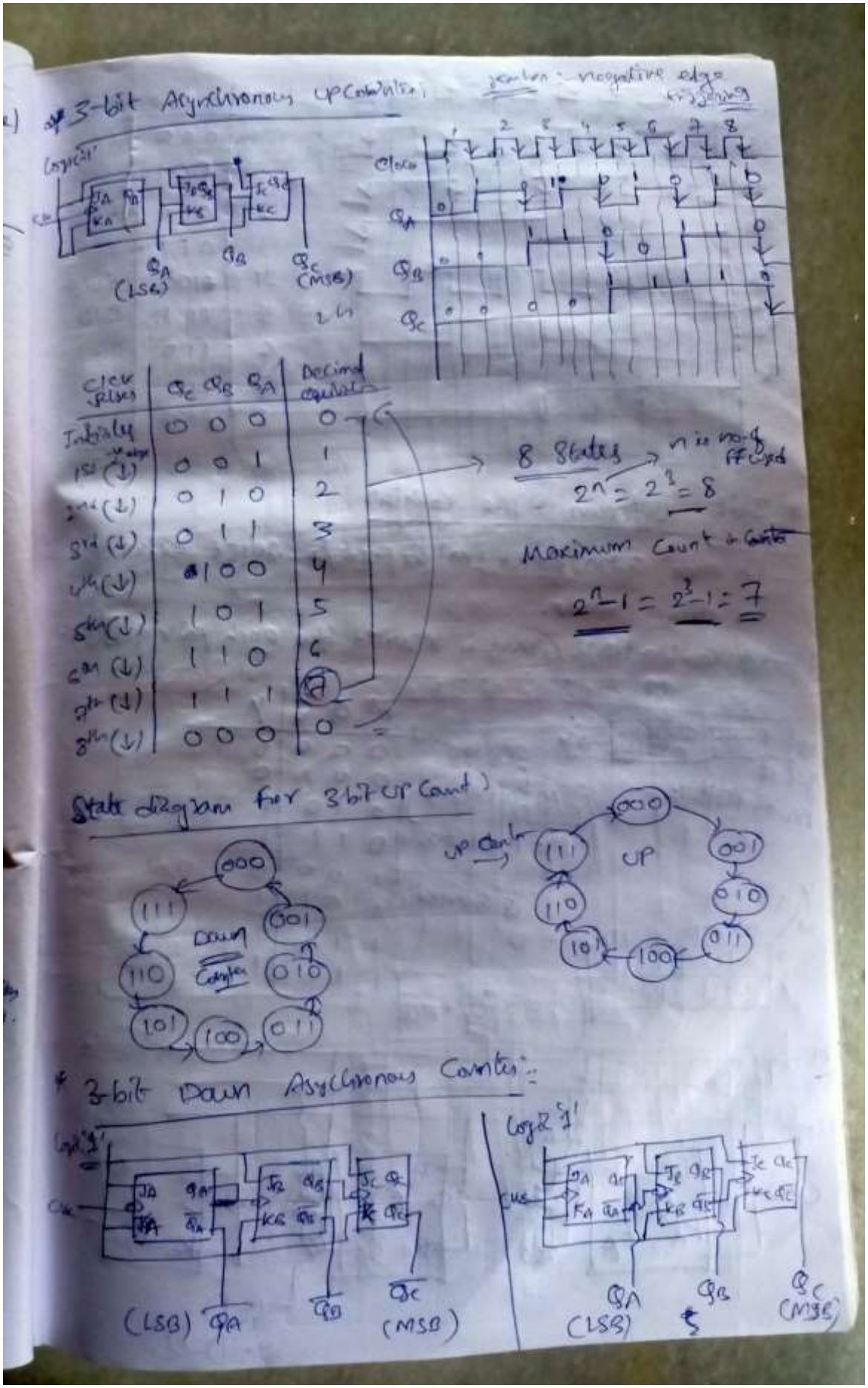
Example

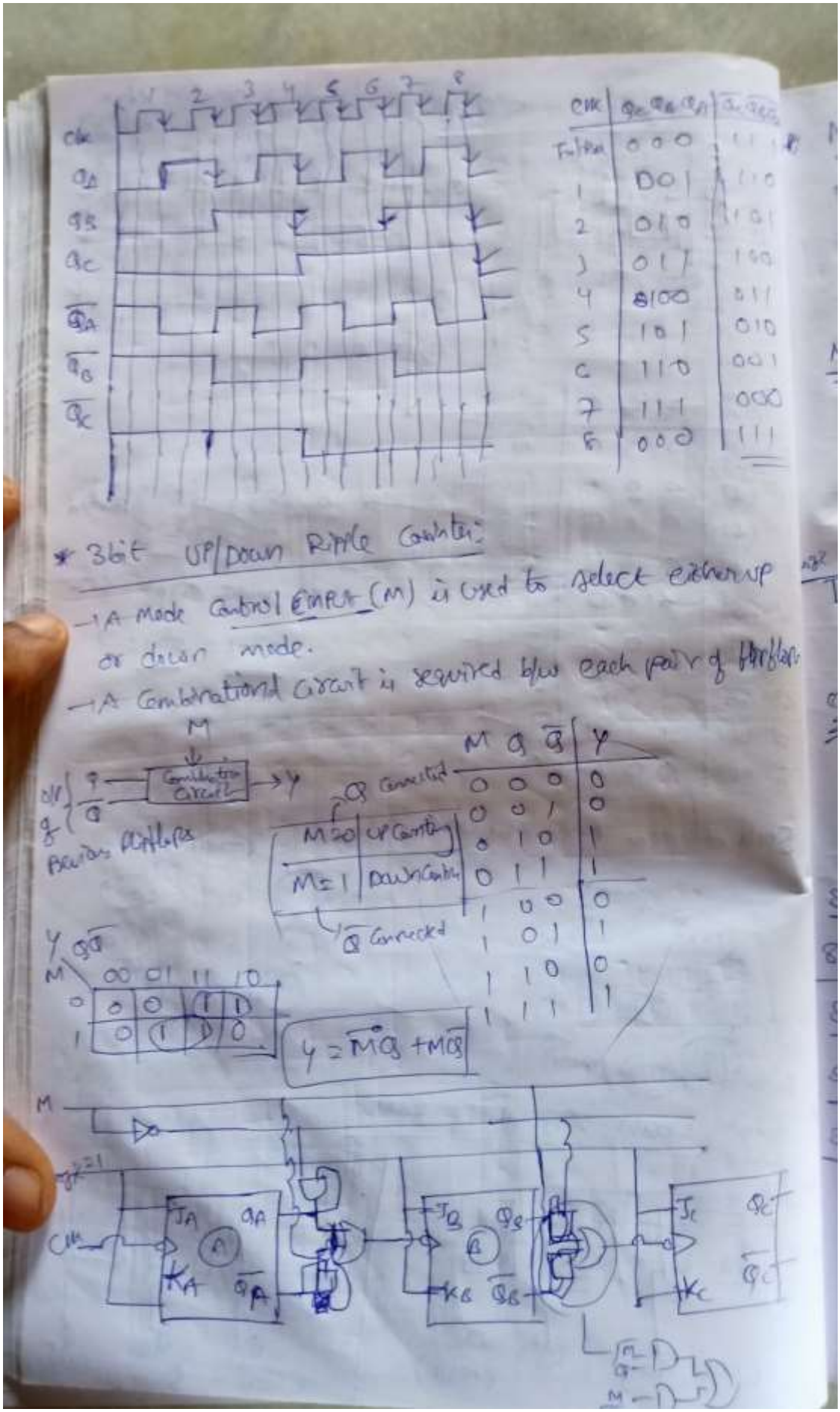


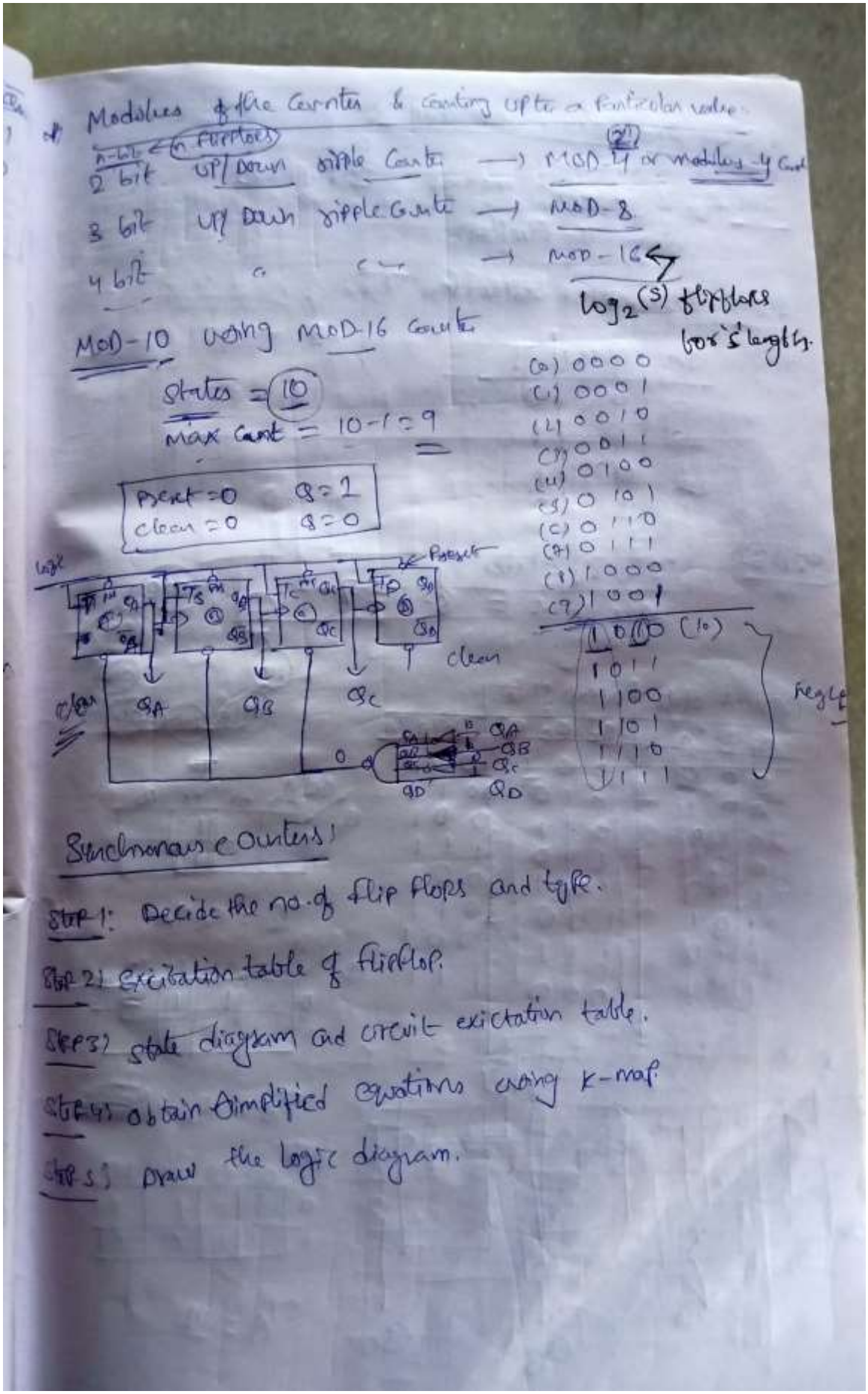
$Y = \overline{Q_B} + Q_A + A$

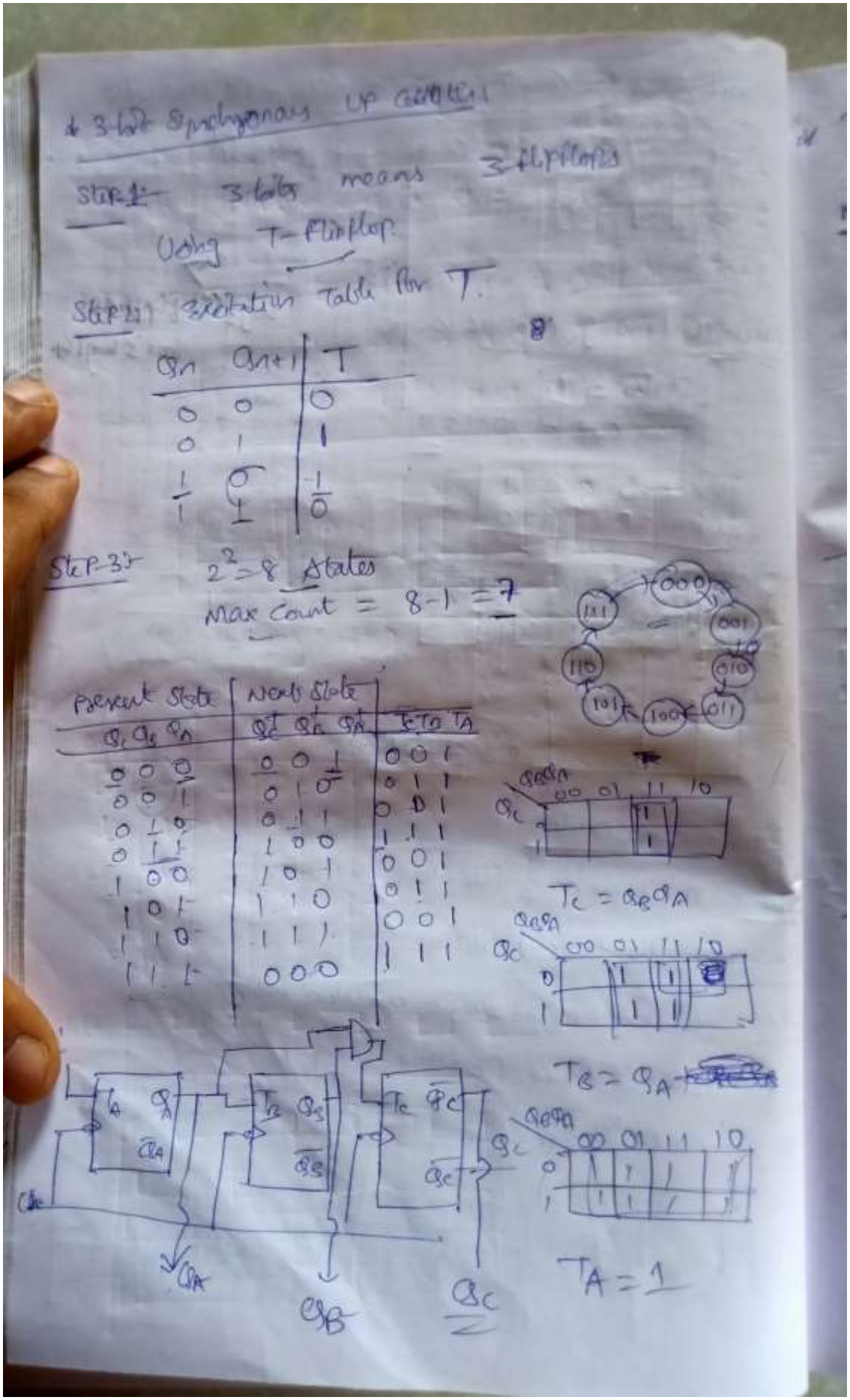
dependent on i/p

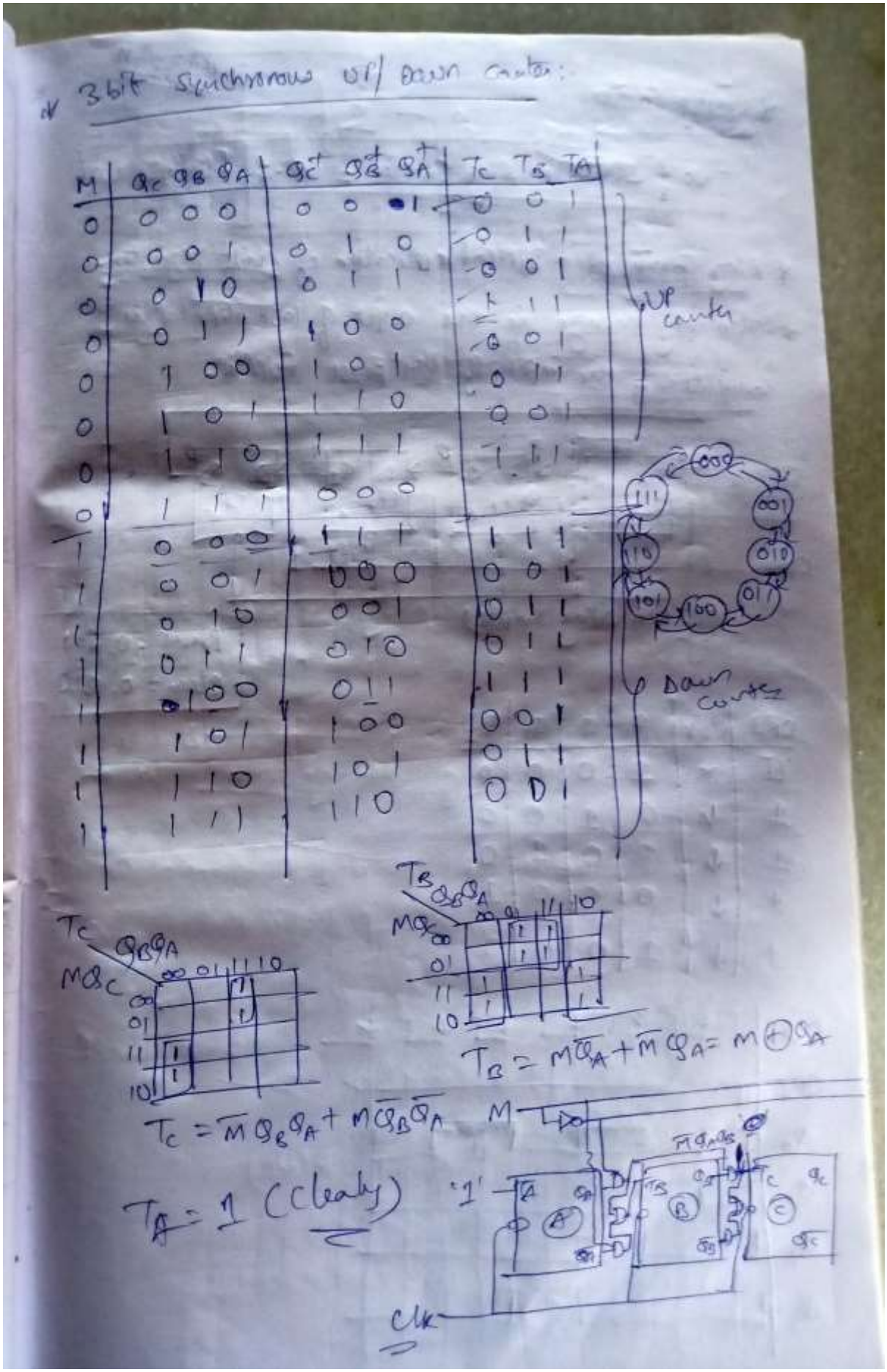

Counters: Counts clock Pulses
Also Wellknown for frequency divider
0 to 3
$T_A = T_c * 2$
$f = \frac{1}{t}$
$t = \frac{1}{f}$
$\frac{1}{f_A} = \frac{2}{f_c}$
$f_A = \frac{f_c}{2}$
$T_B = 2 * T_A$
$\frac{1}{f_B} = \frac{2}{f_A}$
$f_B = \frac{f_A}{2} = \frac{f_c}{4}$
for P no of flipflops
frequency divider rate
2^P times
if P=2 then $2^2 = 4$
if P=4 then $2^4 = 16$
CLK Pulse | Q_B Q_A
0 | 0 0 (0)
1 | 0 1 (1)
2 | 1 0 (2)
3 | 1 1 (3)
4 | 0 0
Slow, simple for more no of states
Ripple/ & Asynchronous Counter: First FF have clock pulse next have their Async control clock
Synchronous Counter: Clock given simultaneously to all flipflops
high
circuit becomes Complicated as no of states increases
Counters (Async / Sync)
UP
Down
UP/Down
0-1-2-3-
7-6-5-0

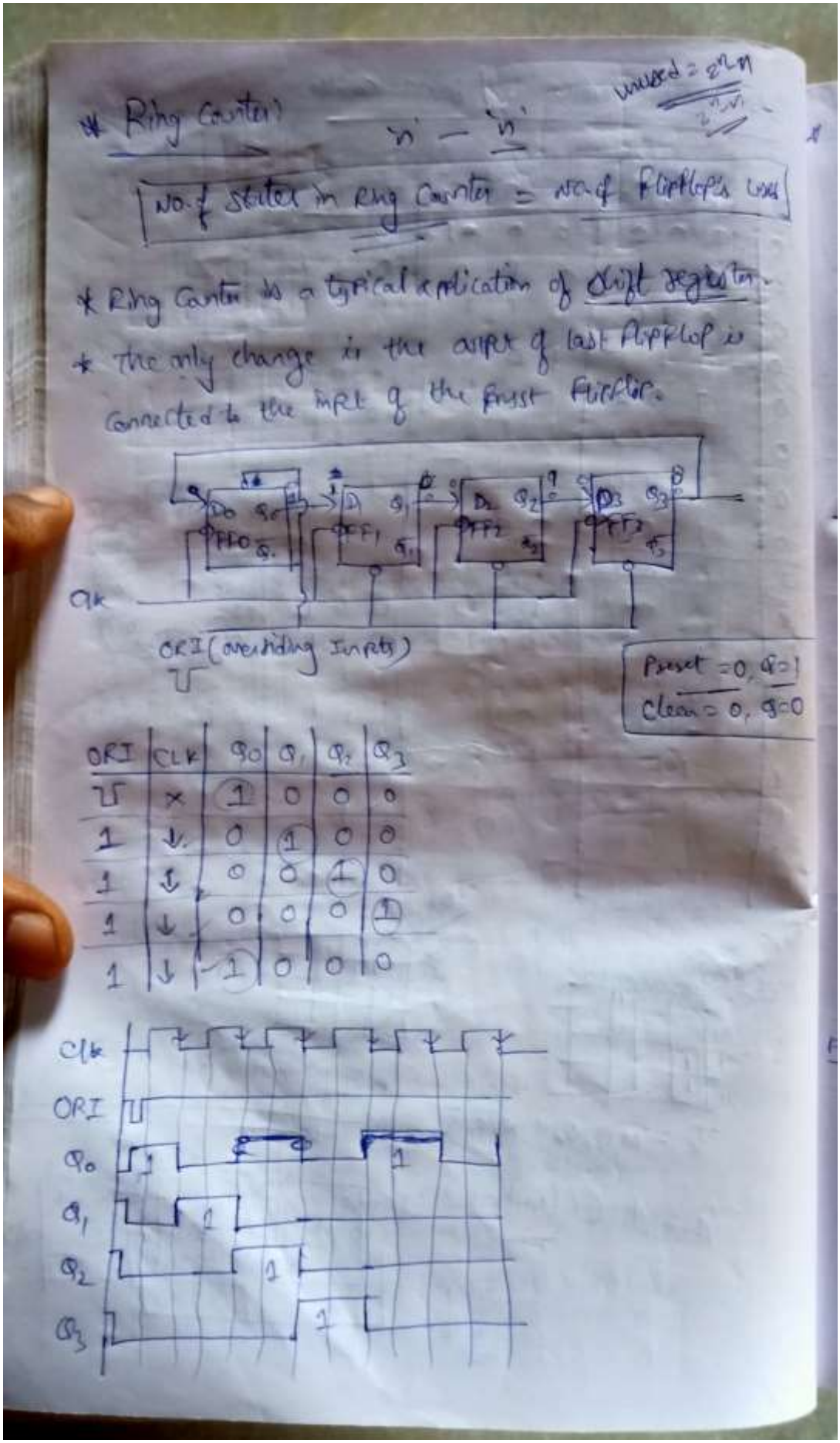

3-bit Asynchronous UP Counter
Logic '1'
Clock
Q_A (LSB)
Q_B
Q_C (MSB)
Clock
Q_A
Q_B
Q_C
Clock Pulses
Decimal equivalent
Initially 0 0 0 0
1st (↓) 0 0 1 1
2nd (↓) 0 1 0 2
3rd (↓) 0 1 1 3
4th (↓) 1 0 0 4
5th (↓) 1 0 1 5
6th (↓) 1 1 0 6
7th (↓) 1 1 1 7
8th (↓) 0 0 0 0
8 States
$2^n = 2^3 = 8$
Maximum Count = Counts
$2^n - 1 = 2^3 - 1 = 7$
State diagram for 3 bit UP Count
UP Count
UP
000
001
010
011
100
101
110
111
DOWN Counter
3-bit Down Asynchronous Counter
(LSB) Q_A
Q_B
(MSB) Q_C
Logic '1'
Q_A (LSB)
Q_B
Q_C (MSB)

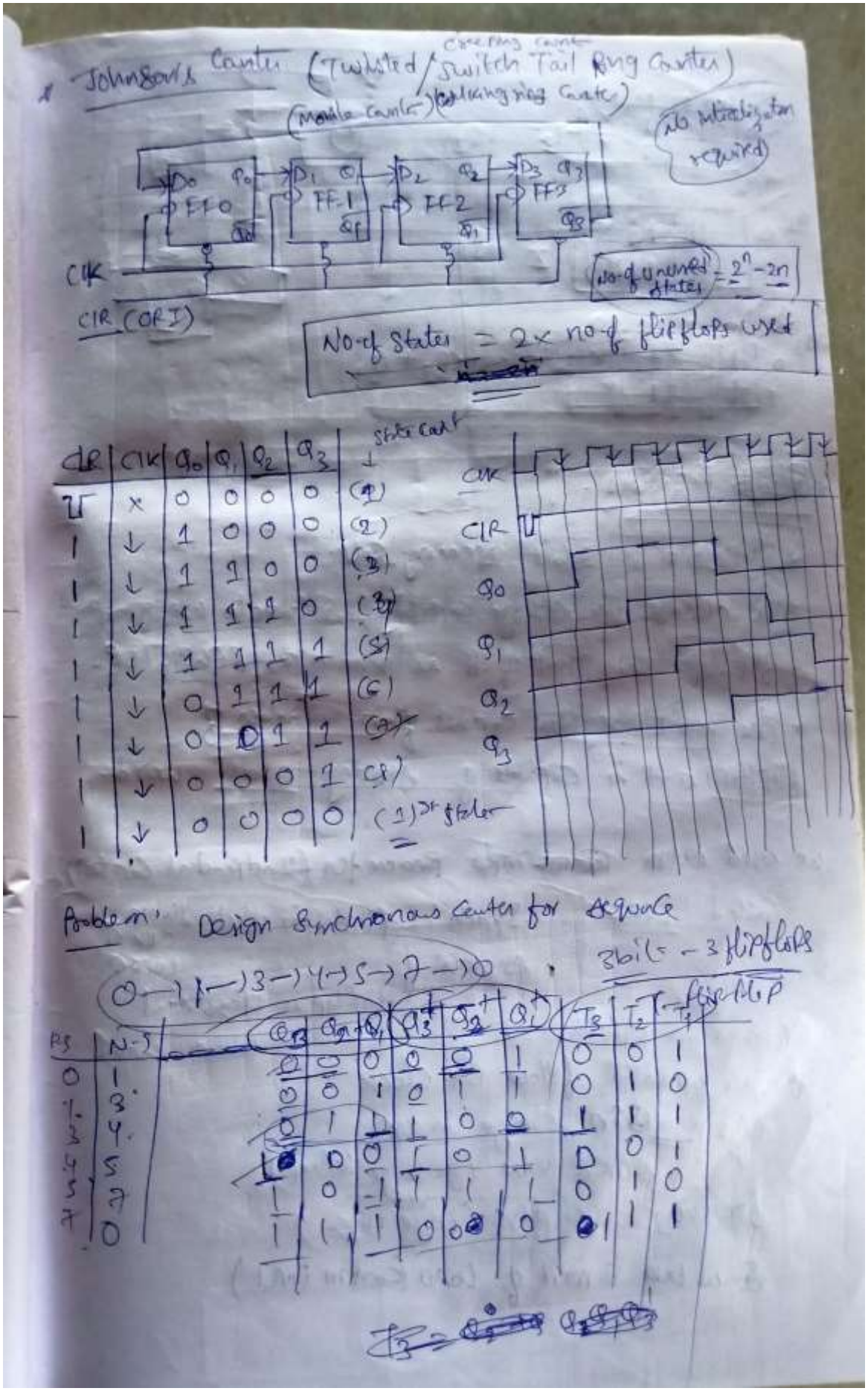

1 2 3 4 5 6 7 8
clk
Q_A
Q_B
Q_C
$\overline{Q_A}$
$\overline{Q_B}$
$\overline{Q_C}$
clk | Q_C Q_B Q_A | $\overline{Q_C}$ $\overline{Q_B}$ $\overline{Q_A}$
Initial 0 0 0 | 1 1 1
1 0 0 1 | 1 1 0
2 0 1 0 | 1 0 1
3 0 1 1 | 1 0 0
4 1 0 0 | 0 1 1
5 1 0 1 | 0 1 0
6 1 1 0 | 0 0 1
7 1 1 1 | 0 0 0
8 0 0 0 | 1 1 1
* 3bit UP/Down Ripple Counter:
→A Mode Control input (M) is used to select either up or down mode.
→A Combinational circuit is required b/w each pair of flip flops
M
Combinational Circuit
Y
Q
$\overline{Q}$
Required flip flops
M Q $\overline{Q}$ | Y
0 0 0 | 0
0 0 1 | 0
0 1 0 | 1
0 1 1 | 1
1 0 0 | 0
1 0 1 | 1
1 1 0 | 0
1 1 1 | 1
M=0 | up Counter
M=1 | Down Counter
Q Connected
$\overline{Q}$ Connected
Y = $\overline{M}Q + M\overline{Q}$
M
clk
J_A K_A Q_A $\overline{Q_A}$
J_B K_B Q_B $\overline{Q_B}$
J_C K_C Q_C $\overline{Q_C}$

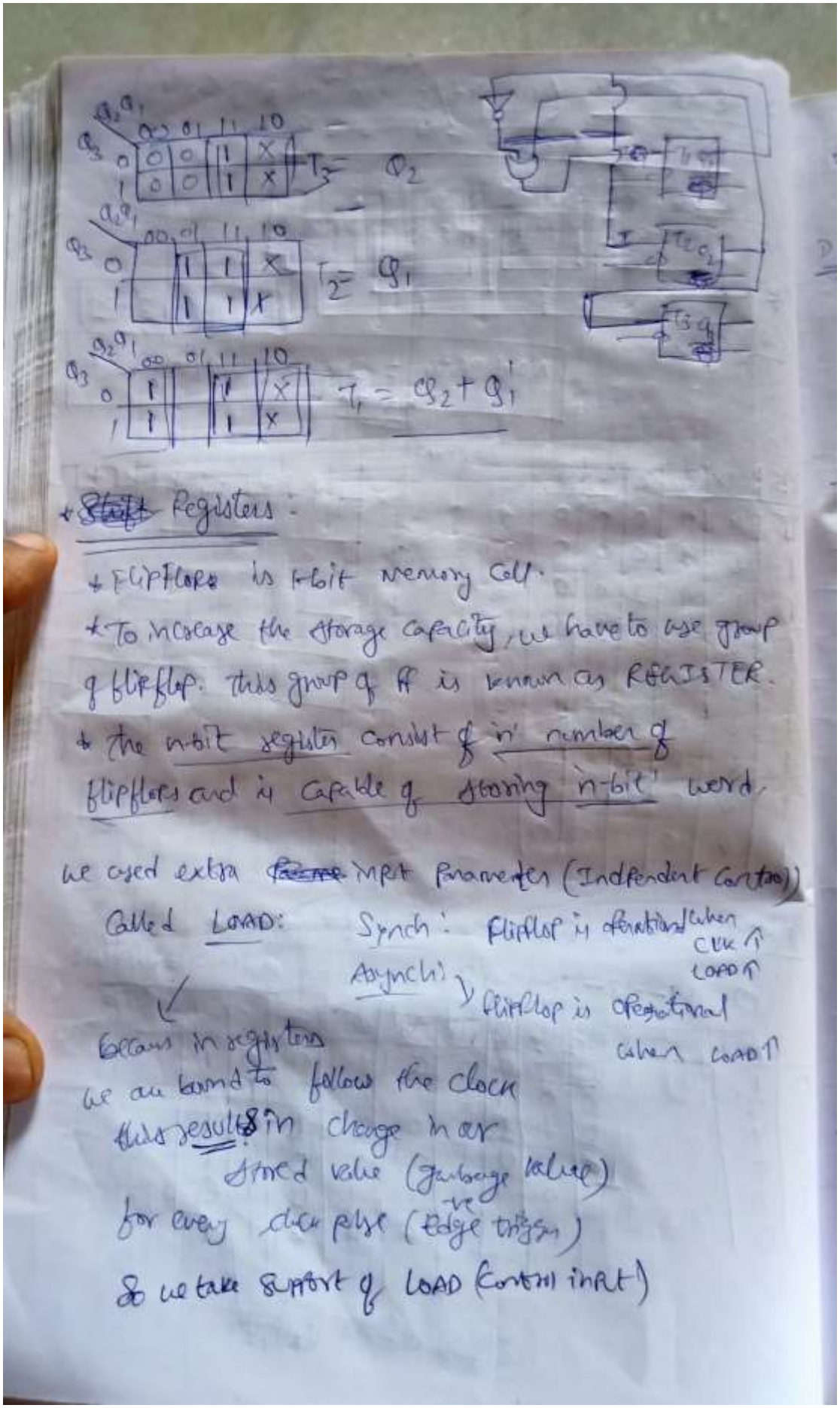

Modulus of the Counter is counting up to a particular value.
n-bit ← n flipflops
2 bit UP/Down ripple Counter → MOD-4 or modulus-4 Counter
3 bit UP/Down ripple Counter → MOD-8
4 bit " " → MOD-16
log2(S) flipflops for 'S' lengths.
MOD-10 using MOD-16 Counter
States = 10
Max Count = 10-1=9
Preset =0 Q=1
clear =0 Q=0
Preset
clear
QA
QB
QC
QD
(0) 0000
(1) 0001
(2) 0010
(3) 0011
(4) 0100
(5) 0101
(6) 0110
(7) 0111
(8) 1000
(9) 1001
1010 (10)
1011
1100
1101
1110
1111
Synchronous Counters:
STEP 1: Decide the no. of flip flops and type.
STEP 2: Excitation table of flipflop.
STEP 3: state diagram and circuit excitation table.
STEP 4: obtain simplified equations using K-map
STEP 5: Draw the logic diagram.


3-bit Synchronous UP Counter
Step-1: 3 bits means 3 Flipflops
Using T-Flipflop
Step-2: Excitation Table for T.
Qn Qn+1 T
0 0 0
0 1 1
1 0 1
1 1 0
Step-3: 2^3 = 8 States
Max Count = 8-1 = 7
000
001
010
011
100
101
110
111
Present State
Next State
Q2 Q1 Q0
T2 T1 T0
TC = QB QA
TB = QA
TA = 1
QA
QB
QC
CLK


3bit Synchronous up/down counter:
M | Qc QB QA | Qc+ QB+ QA+ | Tc TB TA
Up counter
Down counter
000
001
010
011
100
101
110
111
Tc QB QA
MQc
Tc = M̄ QB QA + M Q̄B Q̄A
TB QB QA
MQA
TB = M Q̄A + M̄ QA = M ⊕ QA
TA = 1 (clock)
M
'1'
A
B
C
Clk


* Ring Counter
n - n
unused = $2^n - n$
No. of states in Ring Counter = No. of Flipflops used
* Ring Counter is a typical application of shift register.
* The only change is the output of last flipflop is connected to the input of the first flipflop.
D0 Q0 FF0
D1 Q1 FF1
D2 Q2 FF2
D3 Q3 FF3
Clk
ORI (overriding Inputs)
Preset = 0, Q = 1
Clear = 0, Q = 0
ORI | CLK | Q0 | Q1 | Q2 | Q3
1 | x | 1 | 0 | 0 | 0
1 | ↓ | 0 | 1 | 0 | 0
1 | ↓ | 0 | 0 | 1 | 0
1 | ↓ | 0 | 0 | 0 | 1
1 | ↓ | 1 | 0 | 0 | 0
Clk
ORI
Q0
Q1
Q2
Q3


Johnson's Counter (Twisted / Switch Tail Ring Counter)
(Moebius Counter) (Creeping Ring Counter)
No initialization required
FF-0
FF-1
FF-2
FF-3
CLK
CLR (OR I)
No-of unused states = $2^n - 2n$
No-of States = 2 × no-of flipflops used
state count
CLK
CLR
Q_0
Q_1
Q_2
Q_3
(1) Ist state
Problem: Design Synchronous Counter for sequence
0 → 1 → 3 → 4 → 5 → 7 → 0
3bits - 3 flipflops
flip flop
T_3 T_2 T_1
Q_3^+ Q_2^+ Q_1^+
PS
N-S



* Registers :

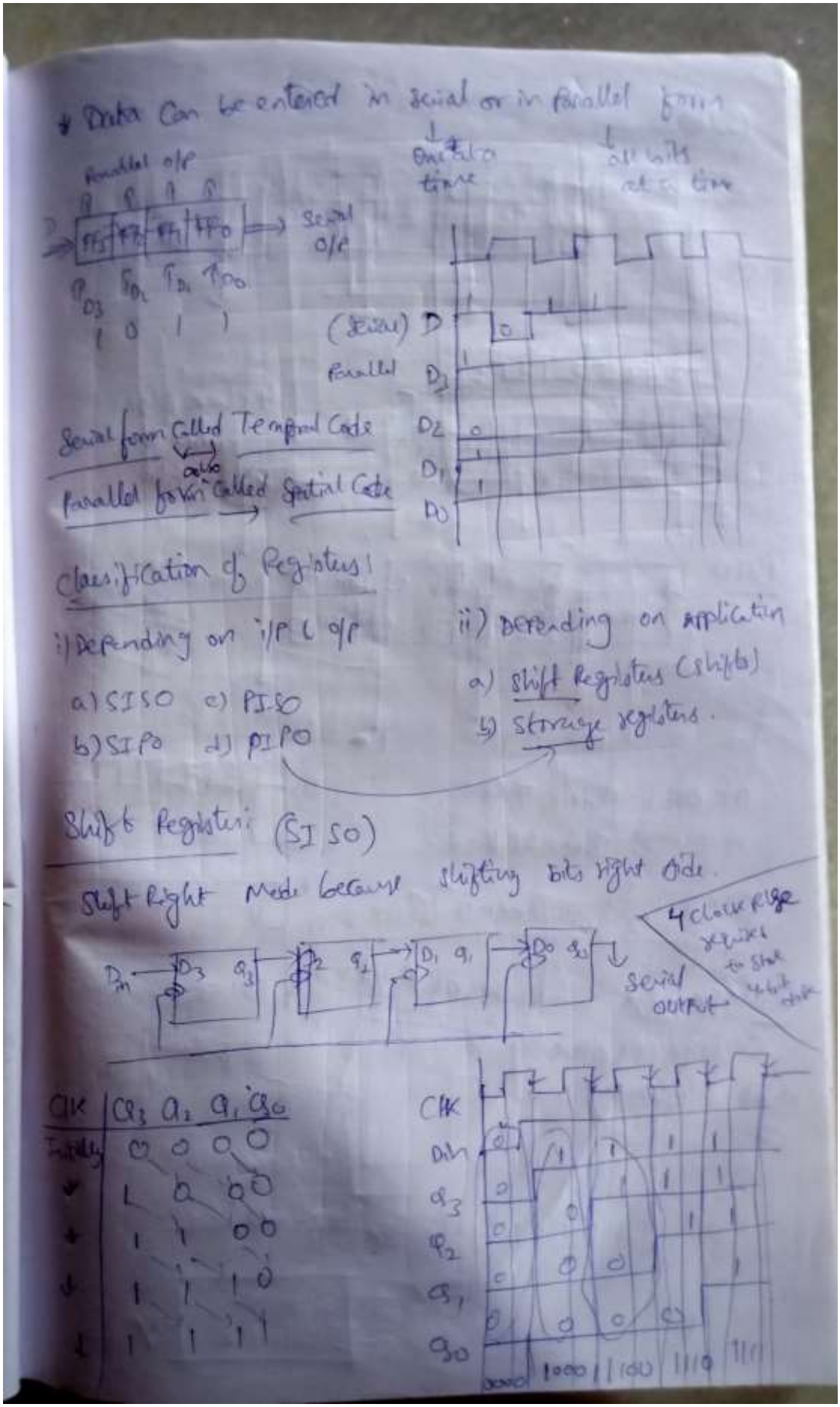
* FLIPFLOPS is 1-bit memory cell.
* To increase the storage capacity, we have to use group of flipflop. This group of ff is known as REGISTER.
* the n-bit register consist of n number of flipflops and it capable of storing n-bit word.

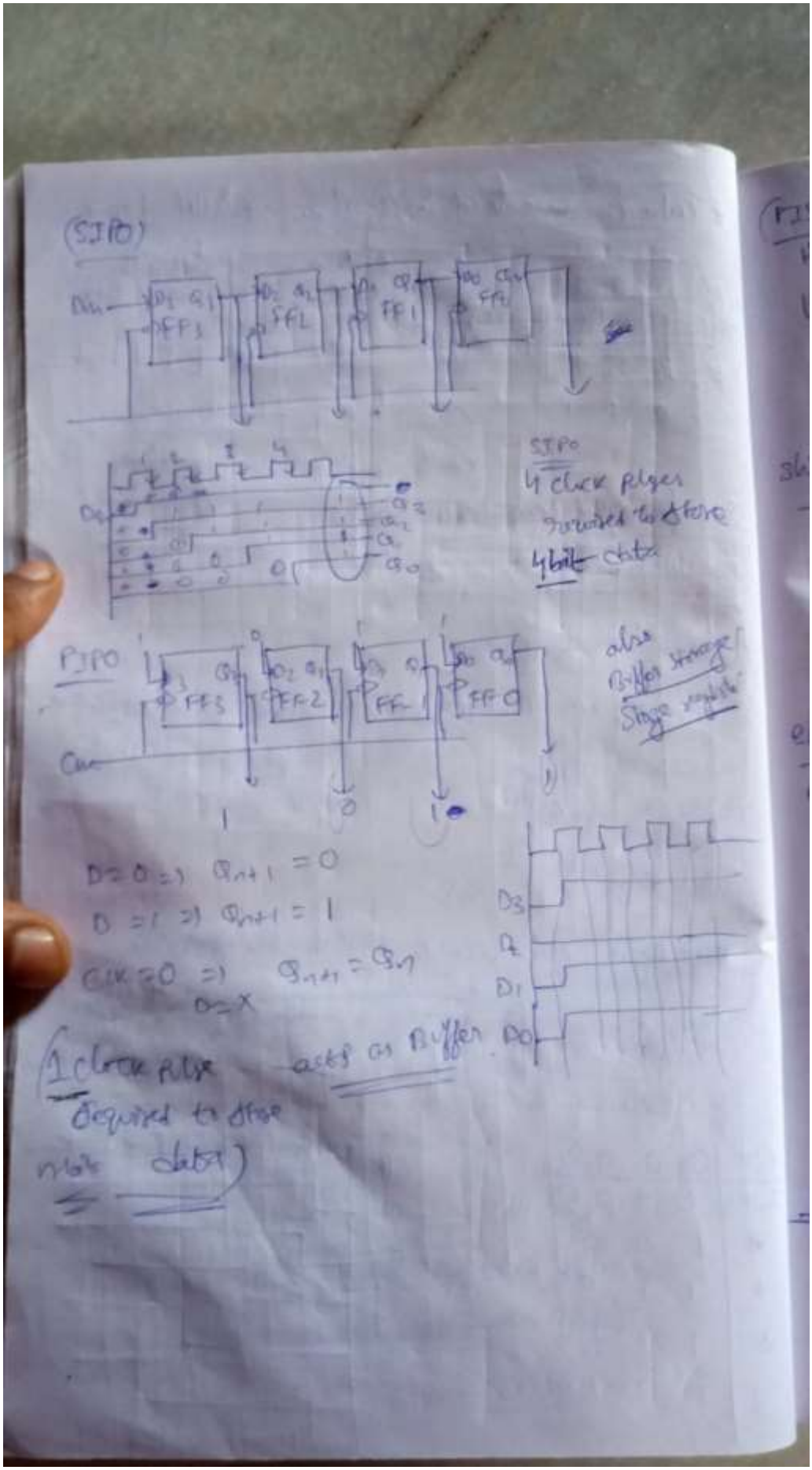
we used extra INPUT parameter (Independent control) called LOAD:

Synch: Flipflop is operational when CLK ↑ LOAD ↑

Asynch: Flipflop is operational when LOAD ↑

Because in registers we are bound to follow the clock this results in change in our stored value (garbage value) for every clock pulse (-ve edge trigger) so we take support of LOAD (control input)


* Data can be entered in serial or in parallel form
Parallel o/p
Serial o/p
Serial form called Temporal Code
Parallel form called Spatial Code
Classification of Registers:
i) Depending on i/p & o/p
a) SISO c) PISO
b) SIPO d) PIPO
ii) Depending on application
a) Shift Registers (shifts)
b) Storage registers.
Shift Registers (SISO)
Shift Right Mode because shifting bits right side.
Serial output
4 clock rise required to store 4 bit data
CLK | Q3 Q2 Q1 Q0


(SIPO)
SIPO
4 clock pulses required to store 4 bit data
PIPO
also Buffer storage Stage register
D=0 => Qn+1 = 0
D = 1 => Qn+1 = 1
CLK=0 => Qn+1 = Qn
D=X
1 clock pulse acts as Buffer required to store 4 bit data

(PISO)

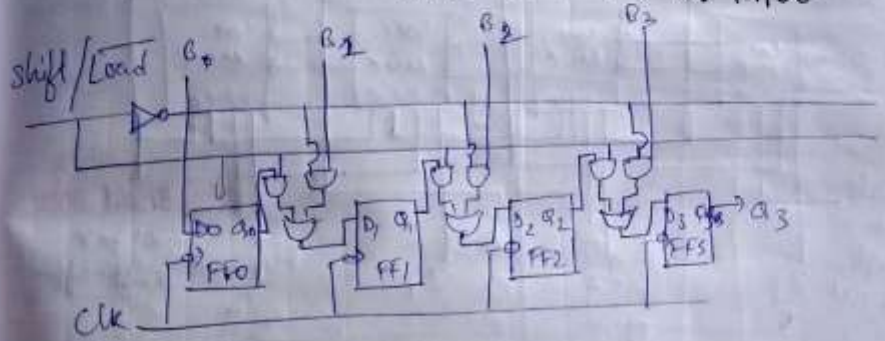
Here we use both shift and Load Control unit

Load, to load inputs into flipflops Parallel

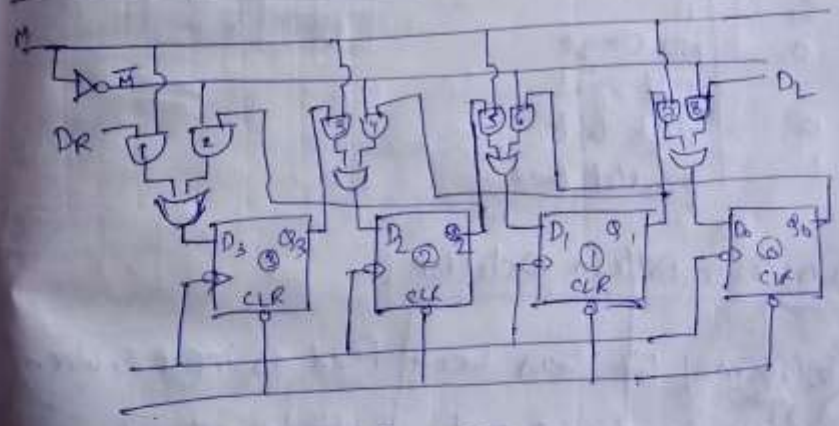
Shift, to shift right data through flipflops.

n-flipflops n+1 Clock cycles required

n-bit ⟵ ⟶ for input



Bidirectional shift Register:



M=1 shift right $(a \div 2^n)$

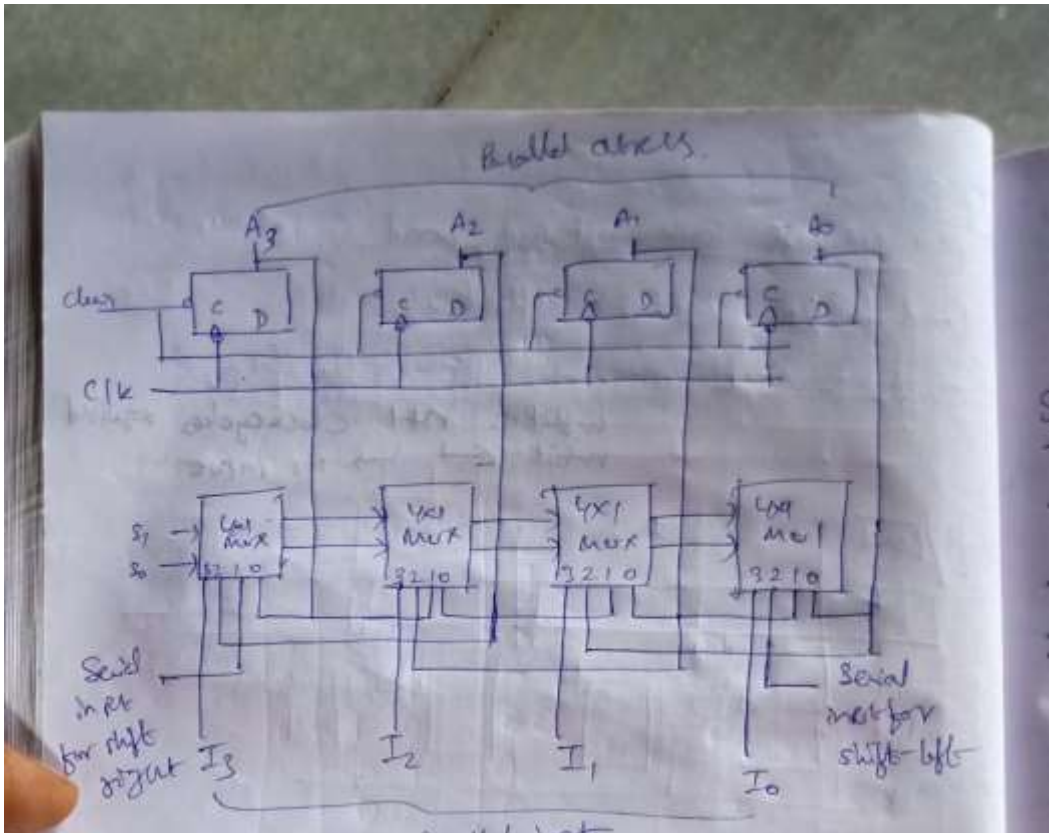
M=0 shift left $(a \times 2^n)$

Universal shift Register

Bidirectional shift Register + Parallel loading

(SISO SIPO
SIPO PISO)

4 Registers

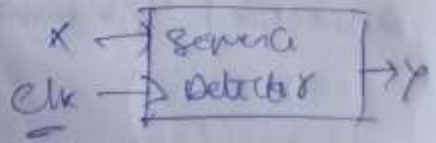


Mode Control		Operation
S_1	S_0	
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

Sequence or Pattern Detector

→ The bitstream of bit has been feed as input, when the clock is high and particular pattern/sequence is detected.

→ As soon as sequence is detected the output becomes high and then again becomes low.



x = 0 1 0 0 1 0 1 0 0 ← non-overlapping (for 010)

x = 0 1 0 0 1 0 1 0 0 ← over-lapping

Step-1: Draw state diagram (Mealy machine)

Step-2: State assignment

Step-3) Circuit state table. (State reduction possible)

Step-4) Implement circuit.

Design 101 sequence Detector

~~Input: 0 1 0 0 1 0 1 0 0 1~~

~~Output: 0 0 0 0 0 0 0 0 1 0 0 0 0~~ (non overlap)

~~overlapping: 0 0 0 0 0 1 0 1 0 0 0~~

Input: 0 1 0 0 1 0 1 0 0 1

overlapping: 0 0 0 0 1 0 1 0 1 0 0 0 0

non-overlapping: 0 0 0 0 1 0 0 0 1 0 0 0 0

101 → 3bit

S_0 2^3 = states

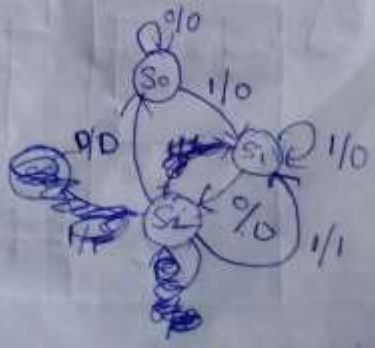
Step-1 state diagram:

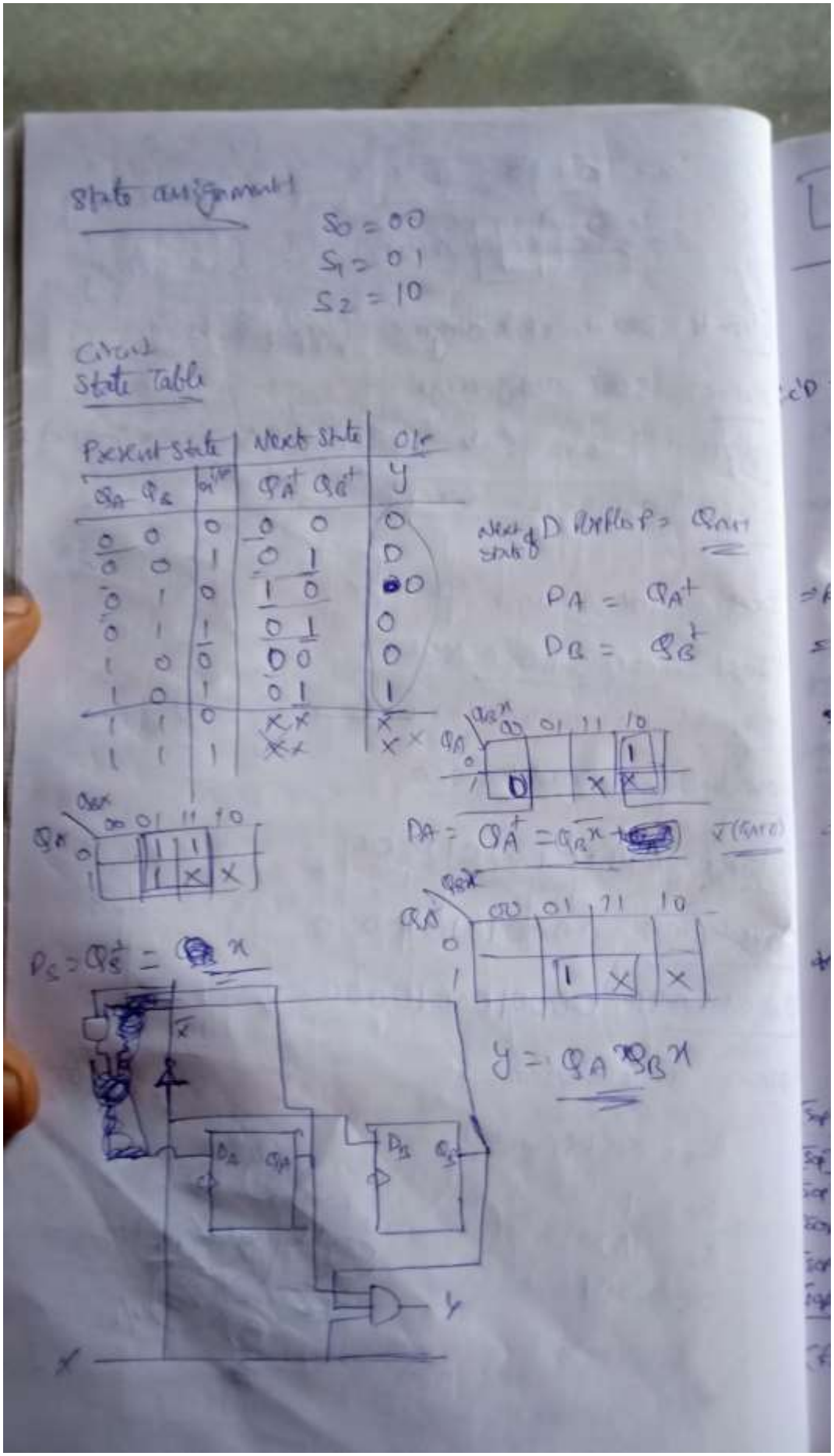
S_0 = reset (power up)

S_1 = 1

S_2 = 10

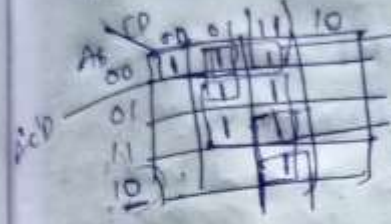
S_3 = 101




State assignment
S0 = 00
S1 = 01
S2 = 10
Circuit
State Table
Present state
Next State
O/P
Next state of D flip flop = Q(next)
DA = QA+
DB = QB+
DA = QA+ = QBx
DB = QB+ = x
y = QA QB x
x
y

refresh rate of RAM : 2 milliseconds

$f(A,B,C,D) = BD + A'B'C' + ACD + BCD$



$A'B'D = ABD + A'BD + A'B'C' + ACD + B'CD$

$= ABCD + ABC'D + A'CBD + A'C'BD + A'B'CD + A'B'C'D + ABCD + ABC'D + AB'CD + A'BCD$

$= A'B'C'D' + A'B'C'D + AB'CD + ABCD + ABC'D + A'BCD + A'BC'D$

$= \Sigma(0, 1, 3, 7, 11, 13, 15)$

Static Hazard resolution by adding.

$= BD + A'B'C' + ACD + B'CD + A'C'D + A'B'D$

* Addition of positive and negative number cannot give overflow.

* The maximum no. of levels that are present b/w inputs and outputs is two in two-level logic 'n' for n-level logic

Degenerative

[SOP] AND-AND
[SOP] AND-NAND
[SOP] OR-OR [SOP]
[POS] OR-NOR
[POS] NAND-NOR
[POS] NOR-NAND

[Can be obtained using single logic gate]

(fan-in) of logic gate increases.

Non-Degenerative

AND-OR [SOP] NOR-NAND
AND-NOR [POS] NOR-OR
OR-AND [POS] NOR-NOR [POS]
OR-NAND
NAND-AND
NAND-OR
NAND-NAND [SOP]

[Cannot obtained using single logic gate)