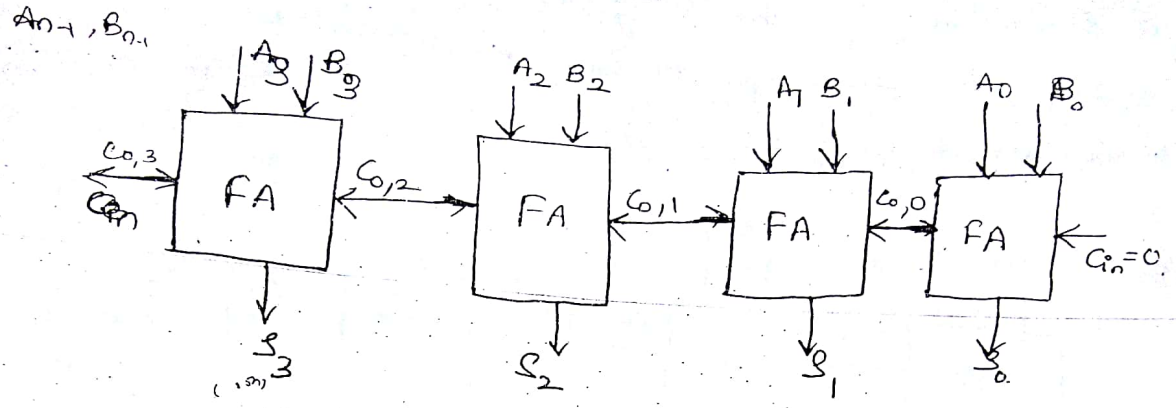
* Ripple Carry Adder (or) Parallel Adder.

4-bit Ripple Carry Adder.



> N-bit Adder can be constructed by cascading N full-Adders circuits in series, such that first

> full adders 'Co' is connected to next fulladder carry-in

and so on.

This arrangement is known as ripple carry Adder, since carry bit ripples from one stage to another.

> The delay through the circuit depends upon the number of stages that must be traversed & is a function of the applied i/p signals.

> for some input combinations no rippling occurs at all, while for other, the carry has to ripple from LSB to the MSB. posn. The propogation delay of such structure is defined as the worst case delay over all possible patterns.

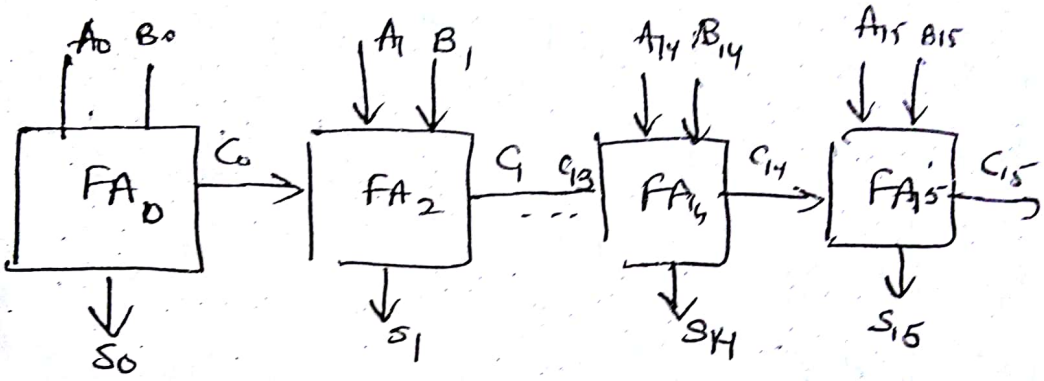
> The delay is then proportional to the no. of bits in the i/p words N' & is approximated by

$$t_{adder} \cong (N-1)t_{carry} + t_{sum}$$

16-bit
$t_{PC} = 12$
$t_S = 15$
$12(15) + t_{sum}$
127

> The propogation delay of the ripple carry adder is linearly proportional to N. This property becomes very important when designing adders for wide datapaths.

* A 16-bit RCA is realized using 16 identical full adders (FA) as shown in fig. The carry-propagation delay of each FA is 12ns & the sum-propagation delay of each FA is 15ns. The worst case delay of this 16-bit Adder?



$$T_{add} = (N-1)t_{pc} + t_{sum}$$

$$= (16-1)t_{pc} + t_{sum}$$

$$= 15 \times 12n + 15ns.$$

$$= 195ns$$

bit adders.

Hence, to increase the speed of the adder, faster carry generation techniques are made use of. These techniques improve the speed of operation of the adder, but on another hand increase the floor area as well.

(i) Carry Look Ahead Adder.

In RCA, the final carry-o/p cannot be obtained without the propogatn of carry-in through the previous adder cells. Using the generate & propogate functn, this carry propogatn can be avoided & the carry o/p's can be computed directly.

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

Inputs A, B, C_{-1} (C_{-1} = C_{in})

$i = 0$

$C_0 = G_0 + P_0 C_{-1}$ — (1)

$i = 1$

$C_1 = G_1 + P_1 C_0$ — (2)

$i = 2$

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

$C_2 = G_2 + P_2 C_1$

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

$i = 3$

$C_3 = G_3 + P_3 C_2$

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

A	B	C_{in}	S	Cout	
0	0	0	0	0	No carry generation i.e Cout = 0
0	0	1	1	0	
0	1	0	1	0	Carry Propogate Cout = C_{in}
0	1	1	0	1	
1	0	0	1	0	
1	0	1	0	1	
1	1	0	0	1	Cout = 1
1	1	1	1	1	

G_i P_i

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

Carry generatn — carry propogatn

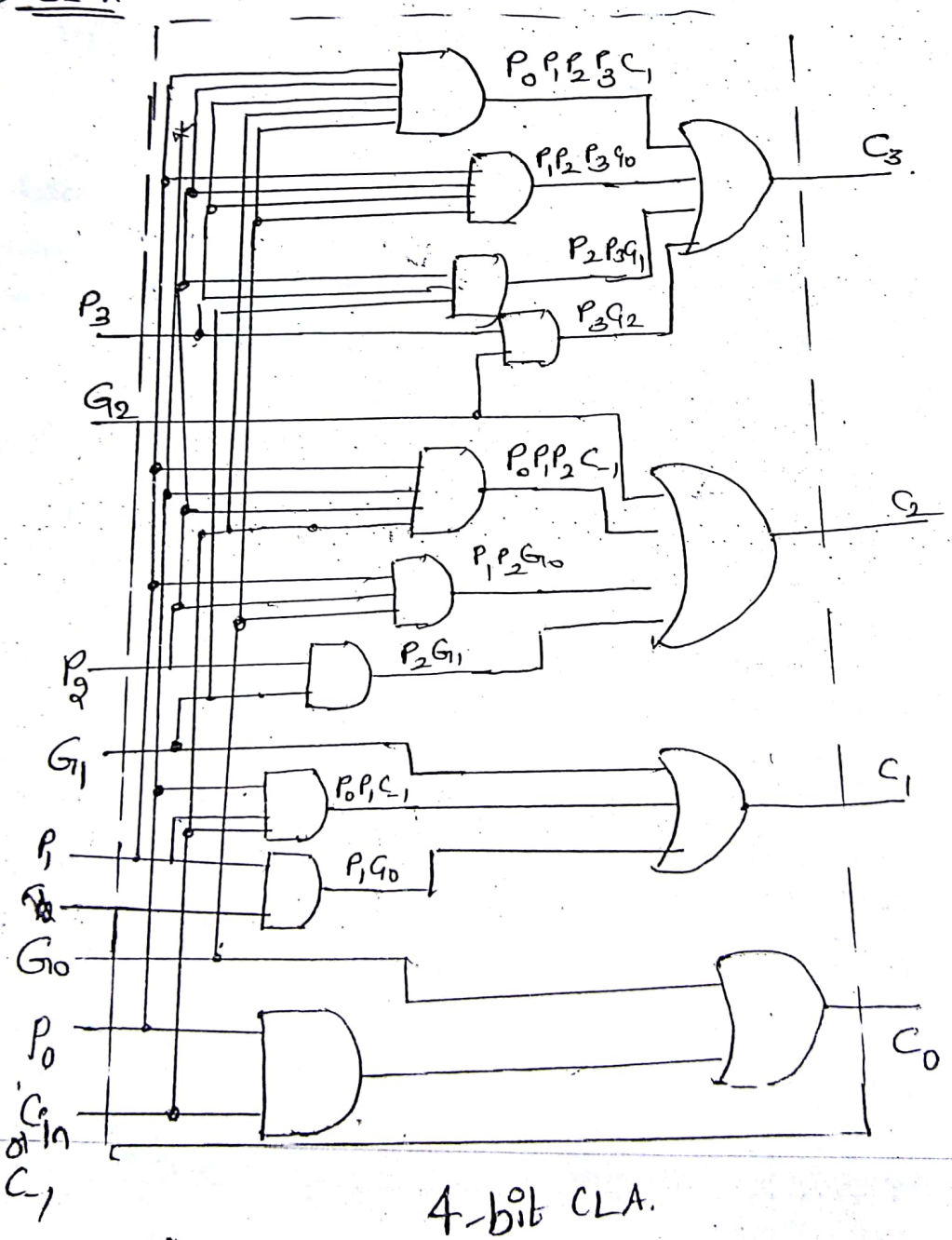
$C_0 = G_i + P C_{in}$

> These equations indicate that carry outputs can be computed directly, without the need for propogating through the adder cells

> Hence, as there is no ripple effect, the speed of the adder is increased.

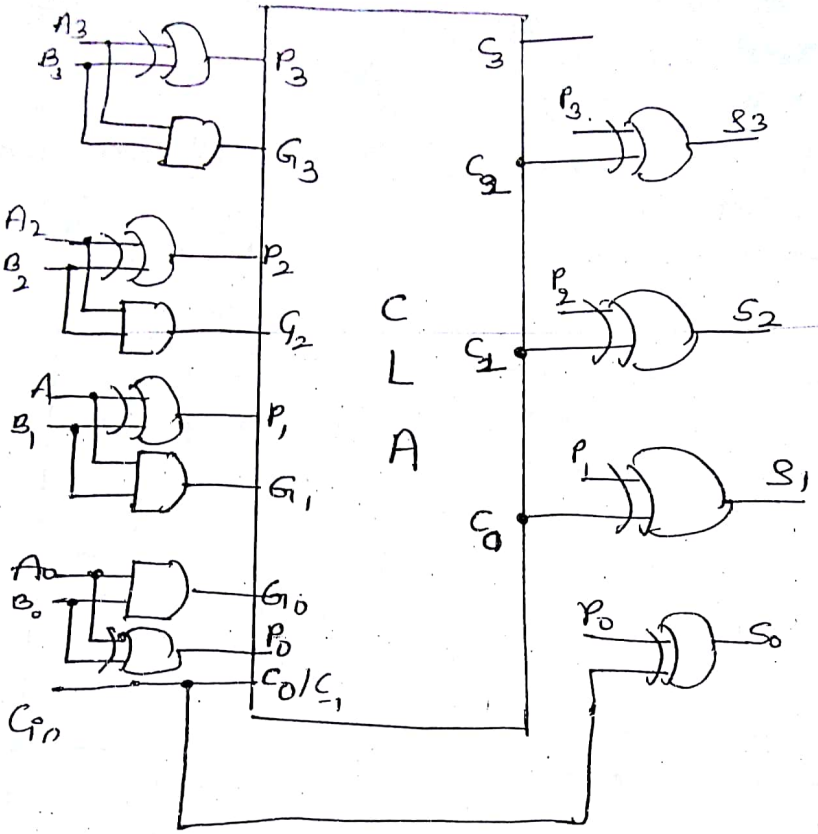
> But the limitation is that with increased no. of carry outputs, the expression becomes much larger & logic becomes quite complexity.

> CLA:



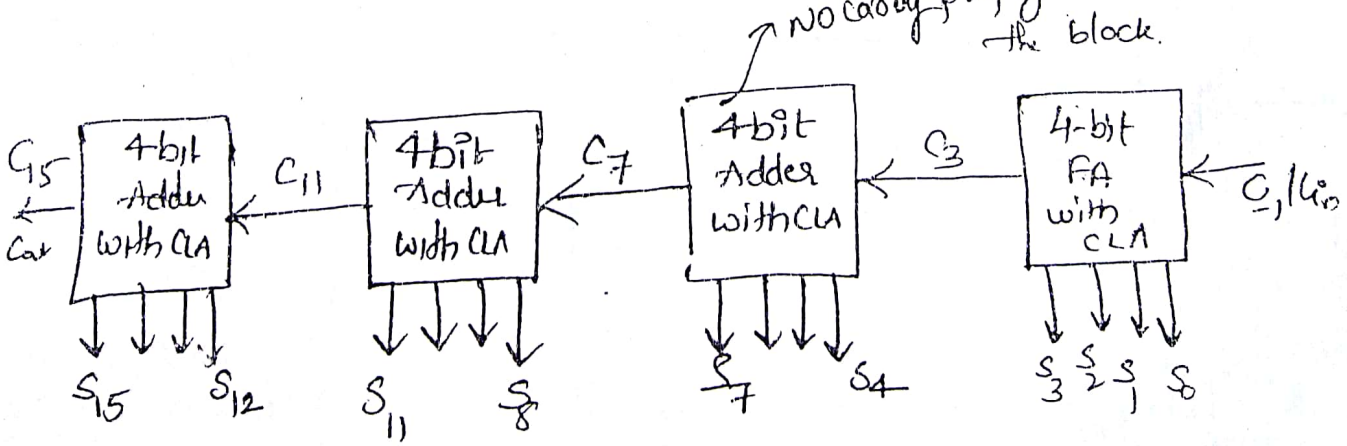
4-bit CLA.

4 bit full Adder Using CLA.



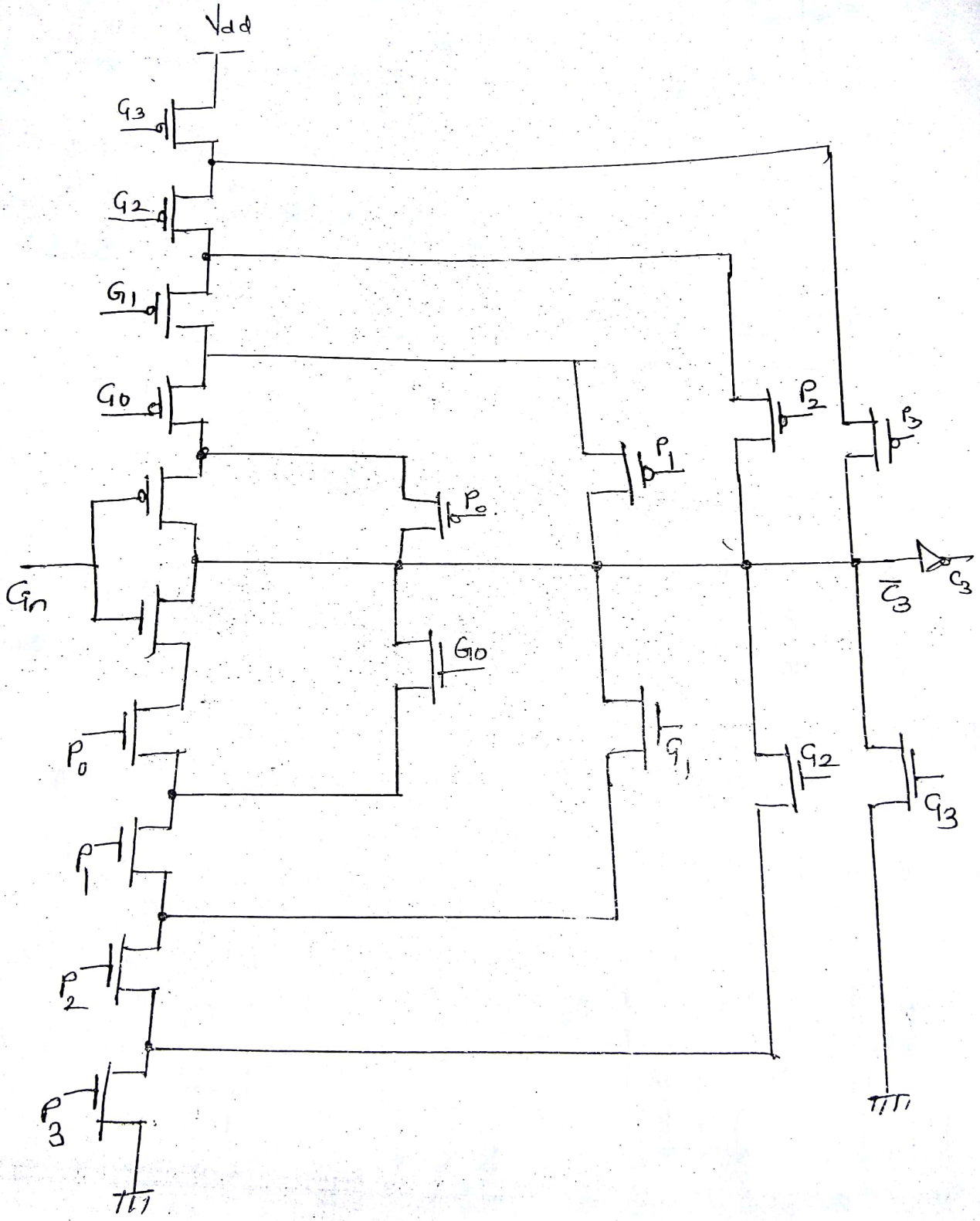
This unit produces the four carry outputs simultaneously However, ever C_0, C_1, C_2 are required only for producing the sum bits. These carry o/ps will not ripple through.

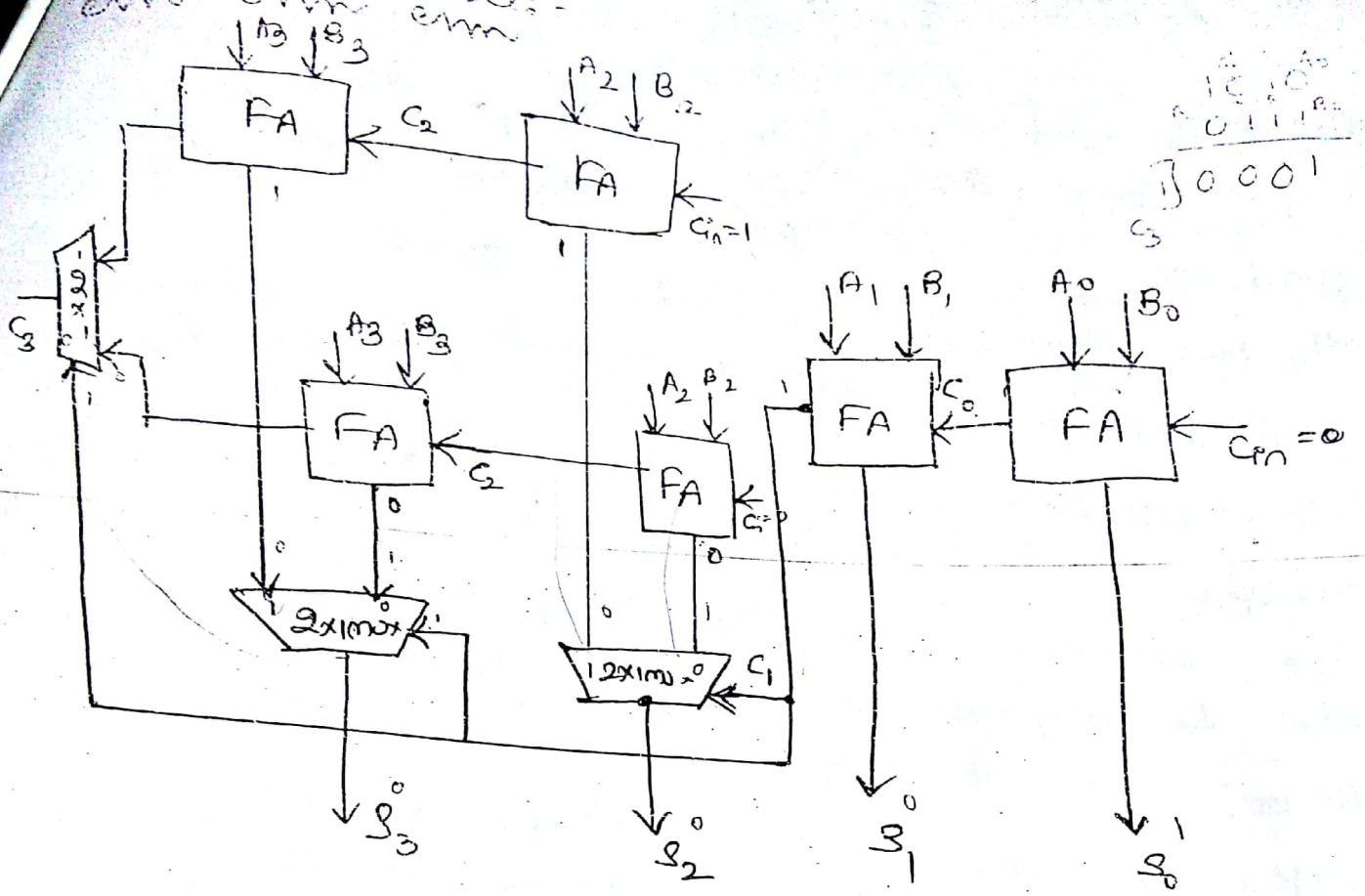
* 16-bit full-Adder using 4-bit FullAdder.



for $\overline{C_3}$ expression – CMOS implementation

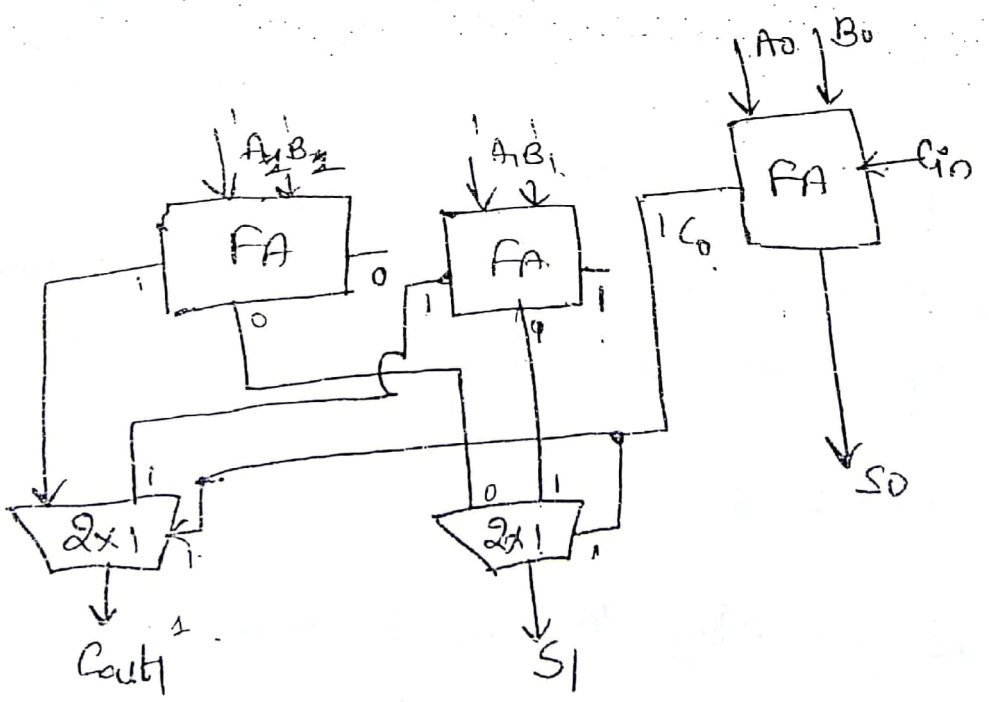
$$\overline{C_3} = G_3 + P_3G_2 + P_2P_3G_1 + P_1P_2P_3G_0 + P_0P_1P_2P_3C_{-1}$$





4-bit CSA.

2-bit CSA:



$C_{in}=0$

A_1 A_0
1 1
B_1 B_0
1 1
1 0

*Carry Select Adder

- A standard logic design technique to accelerate the critical path is to pre-compute the o/ps for both possible i/ps, & then use a multiplexer to select btw the two o/p choices.
- CSA consists of two ripple carry adders & a multiplexer.
- The calculation is done twice with an assumption that the carry i/p is '0' & another adder with an assumption that the carry i/p is 1.
- After the two results are calculated, the correct sum, as well as the correct carry-out, is then selected with the multiplexer once the carry-in is known.

*Improvement in Speed.

Let us consider the RCA, for an n-bit adder, the computation time is given by $T = nk_1$, where k_1 is the delay through one adder cell.

→ for, Carry Select Adder, if the adder is divided into blocks, with each block containing two adder cells, then $T = 2k_1 + (n-2)k_2$

where k_2 is the delay through the mux.

→ Hence it is observed that, if not planned properly then 'k_2' may ~~practically~~ there may not be any great advantage.

→ T_2 - Ans.

Let the n-bit adder be divided into 'M' blocks & each block contain 'p' adder cells in series i.e. $n = MP$ (Ex: 16-bit → 4 blocks → each block has 4 adder cells). The completion time (T) is the sum of the propogation delay through the first block & propogatn delay through the mures.

i.e. $T = Pk_1 + (M-1)k_2 \quad -(1)$

To obtain minimum T eq(1) has to be differentiated w.r.t 'M' & the result to be equated to zero.

$n = MP$

$P = n/M$

i.e. $\frac{dT}{dM} = \frac{d}{dM}\left\{\frac{n}{M}k_1 + (M-1)k_2\right\} = 0$

$nk_1\left(-1/M^2\right) + k_2 - 0 = 0$

(or) $\frac{nk_1}{M^2} = k_2 \Rightarrow \boxed{M = \sqrt{\frac{nk_1}{k_2}}}$

Ex:- for a 64-bit CSA, given $k_1 = 4ns$ & $k_2 = 1ns$. find the no. of blocks & no. of adder cells in each block for achieving min. T.

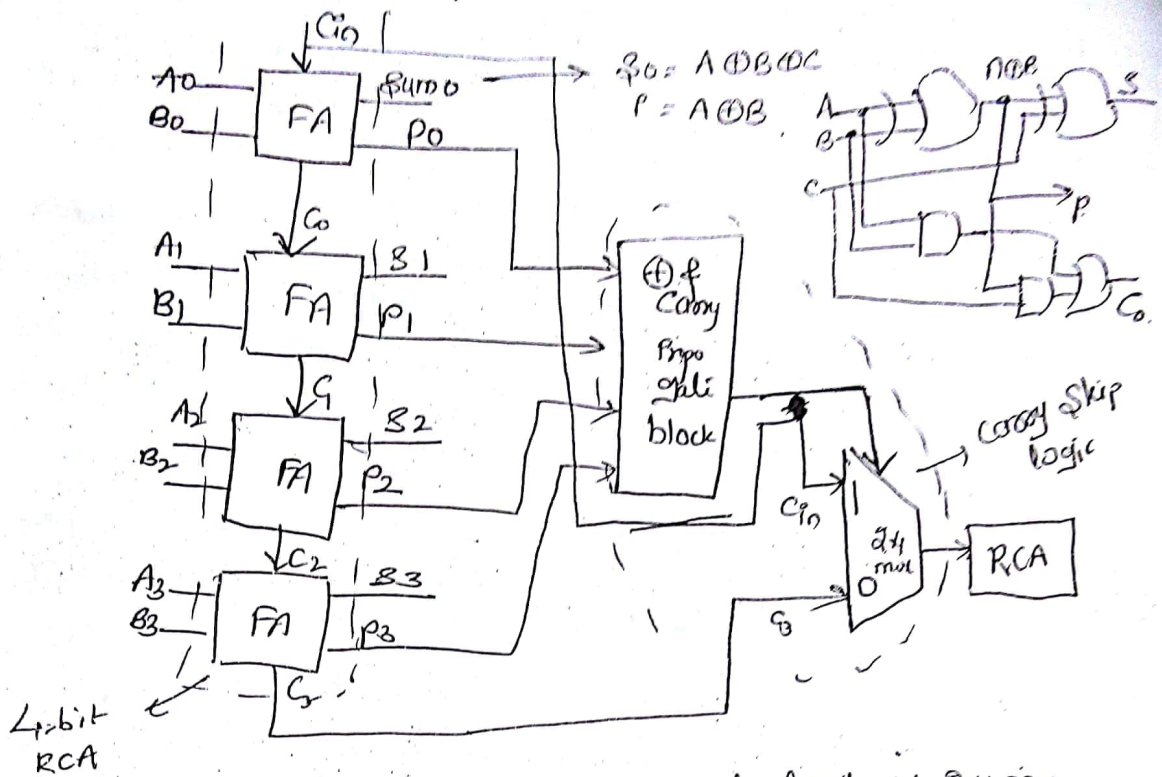
Given $n = 64$, $k_1 = 4ns$, $k_2 = 1ns$

$M = \sqrt{\frac{64 \times 4}{1}} = \sqrt{256} = 16$

$P = \frac{64}{16} = 4$, $T = 16 + 15 = \underline{31ns}$

* Carry Skip Adder: / carry-by-pass Adder.

4-bit carry skip Adder.



Carry skip logic:- Using carray propogate logic + 2x1 mux

Carry is skipped of Carry is propogated to next RCA blk.

-> carry propogate:- $BP = (A_0 \oplus B_0)(A_1 \oplus B_1)(A_2 \oplus B_2)(A_3 \oplus B_3)$

block --> if $P = 1$, then no need to wait for C_3 generation directly passed C_{in} to next block which eliminates carry generation time. i.e. C_3

Eg:-

$A = 1010$

$B = 0110$

$C_2 \; S \; C_0 = 0000$

$P = A \oplus B \Rightarrow P_0 - 0$

$P_1 - 0$

$P_2 - 1$

$P_3 - 1$

$BP = P_0 \cdot P_1 \cdot P_2 \cdot P_3$

$= 0$.

$\therefore$ Wait for C_3 generation

->gn

> Thus, to save this propogation time a special signal called as "block propogation" is used

> This is defined as ($BP = \pi P_i$) & if this signal is '1', then carry-in need not be propogated through the block

> Instead it can be directly transmitted through a mux, to the next block

Thus, the propogation delay gets minimized & speed of the adder gets increased.

* Case if BP = 0, then it indicates that there can be generation of carry & hence the i/p carry needs to be propogated through the block. Now to reduce, this delay, the choice of the block size becomes necessary, for which the compilation is -

Let 'k_1' be the time needed by the carry signal to propogate thro' the adder cell & 'k_2' - propogation delay of mux. Then compilation time is given by,

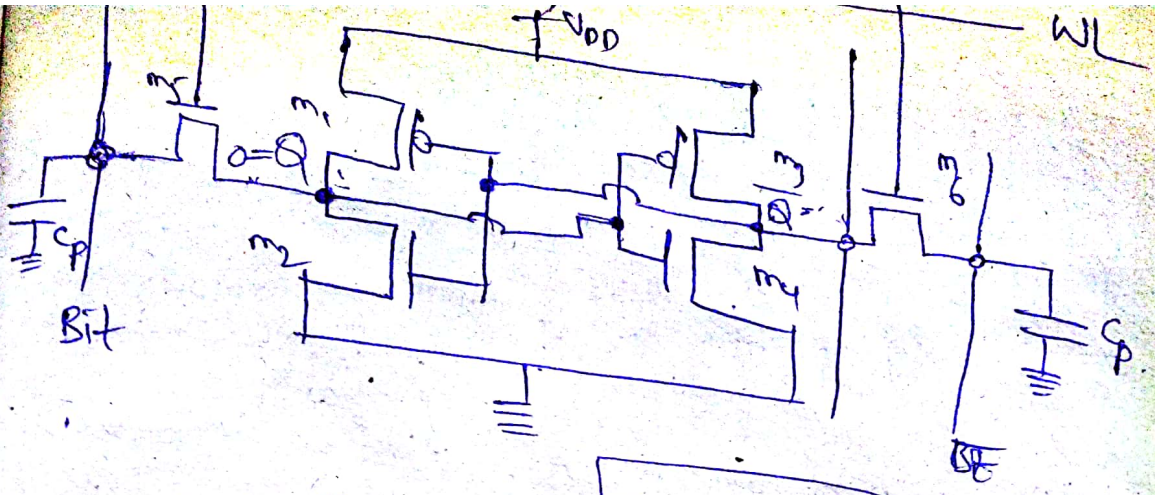
$$T = 2(P-1)k_1 + (M-2)k_2 \quad \text{---(1)} \qquad \boxed{n = MP}$$

$$\therefore T_{min} \Rightarrow \frac{dT}{dM} = \frac{d}{dM}\left[2\left\{\frac{n}{M} - 1\right\}k_1 + (M-2)k_2\right] = 0$$

$$\Rightarrow 2n\left\{-1/M^2\right\}k_1 + k_2 = 0 = 0$$

$$\frac{2nk_1}{M^2} = k_2$$

$$\boxed{M = \sqrt{\frac{2nk_1}{k_2}}}$$



Read

V_{th} of M_4 = 0V & M_3 = off

$\bar{Q} = 0$

2x-5

~~5x2~~

00 → No opn
01 → Add
10 → Sub
11 → No opn
} ARS ⊕

5 → 00101 → -5 → 2's → 11011

2 → 00010

↳ 5 no. of passes

00000 11011 0

Pass(1): 1 1 1 1 0 0

0 0 0 1 0

1 1 1 1 0

1 1 1 1 0 11011 0

1 1 1 1 1 0 1 1 0 1 1

Pass 2 :→

1 1 1 1 1 . 0 1 1 0

1 1 1 . 1 1 1 . 0 1 1 0

Pass 3 :→

1 1 1 1 1

(1) 0 0 0 1 0
1 1 1
0 0 0 0 1

0 0 0 0 1 1 0 1 1 0

0 0 0 0 0 0 1 1 0 1 1 0

Pass 4 :→

0 0 0 0 0
0 0 0 1 0
1 1 1 1 0

1 1 1 1 0 1 1 0 1 1 0
↓
1 1 1 1 1 0 1 1 0 1 1

Pass 5:-

1 1 1 1 1 1 0 1 1 0 1

1 1 1 1 1 1 1 0 1 1 0 1

neglect

2's Complement.

0 0 0 0 0 0 1 0 0 1

+ 1

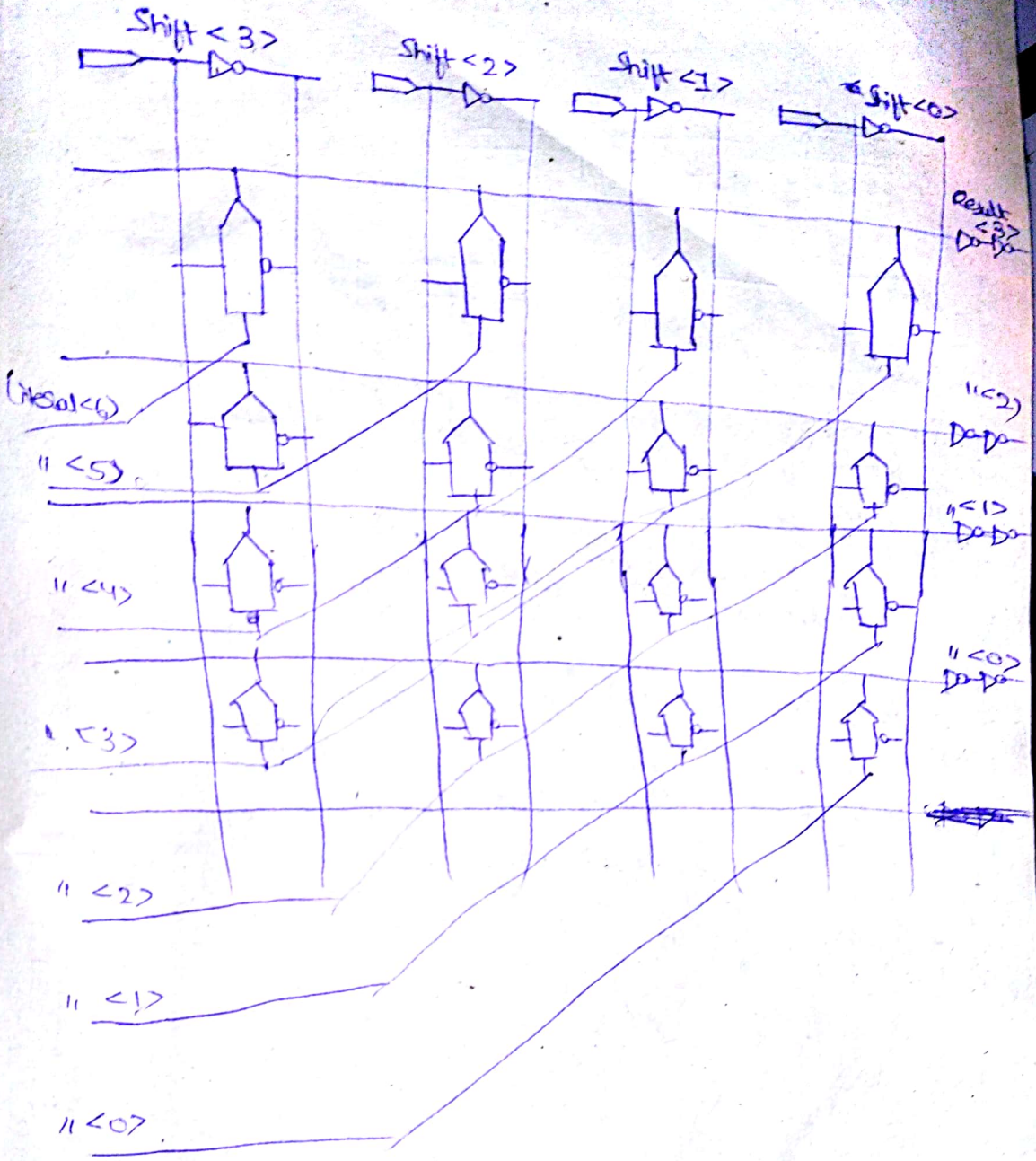
0 0 0 0 0 0 1 0 1 0

10.

Carry generated in Pass3

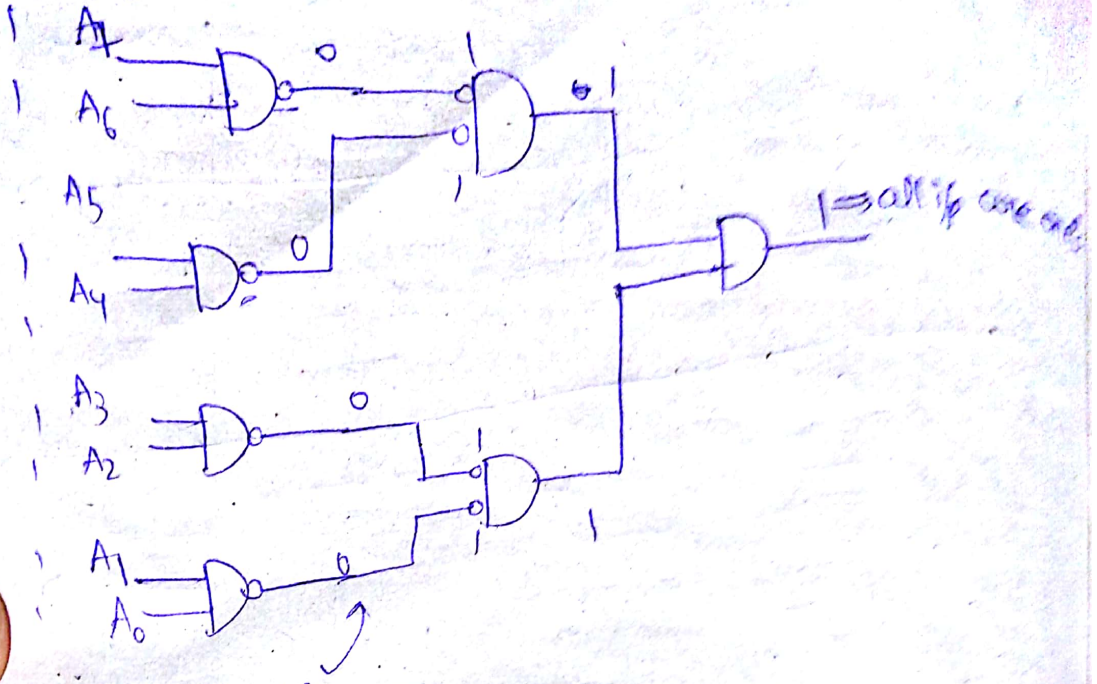
So it is -10

*Shift Registers :-

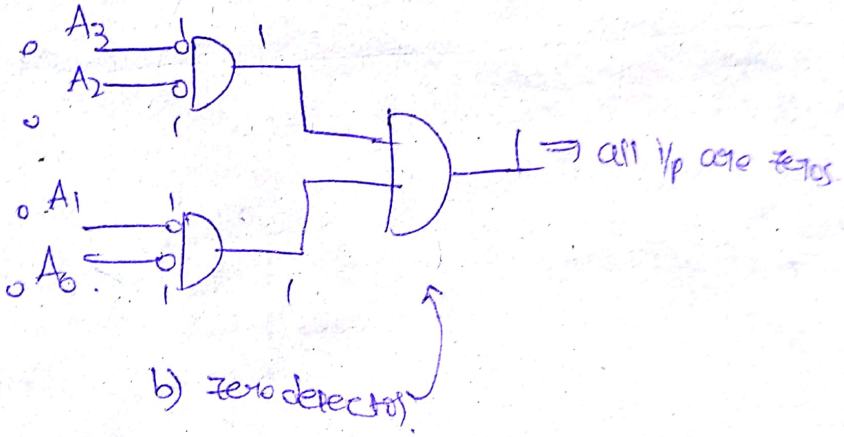


Shift	~~literal~~ Result
0	literal <3:0>
1	literal <4:1>
2	literal <5:2>
3	literal <6:3>

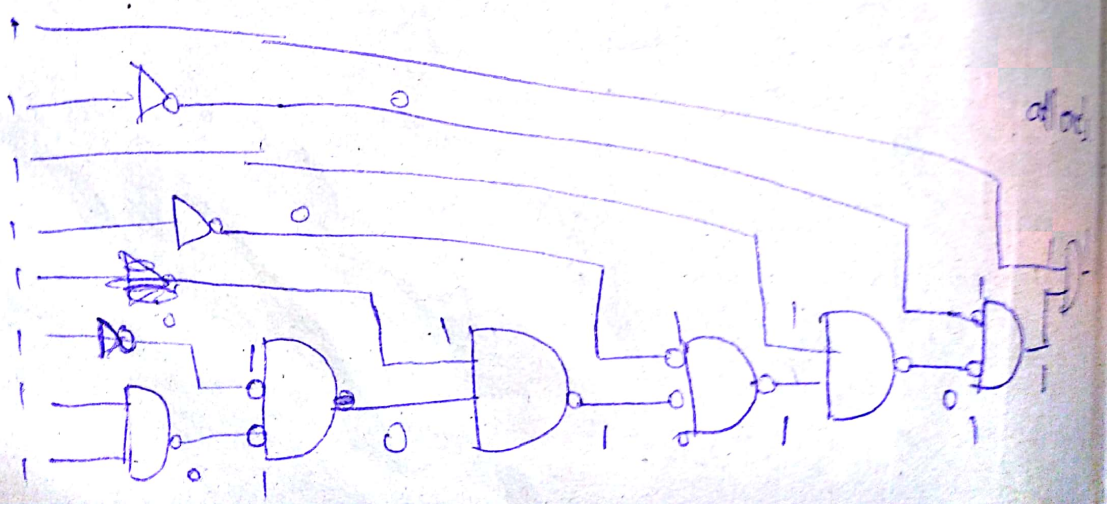
*Zero / One detector

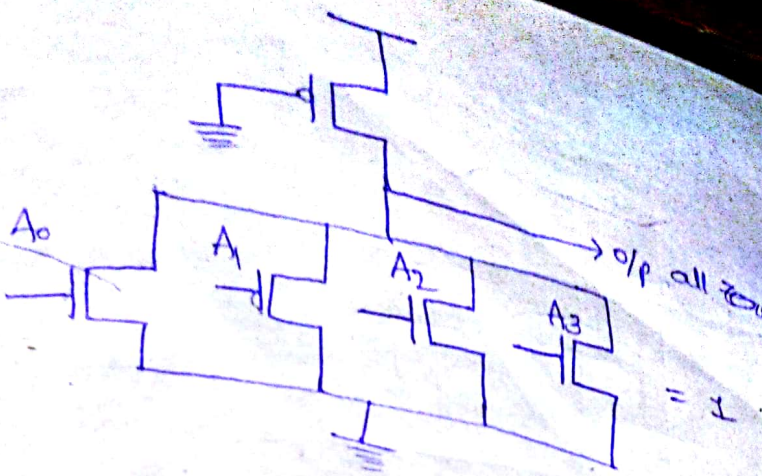


a) one detector

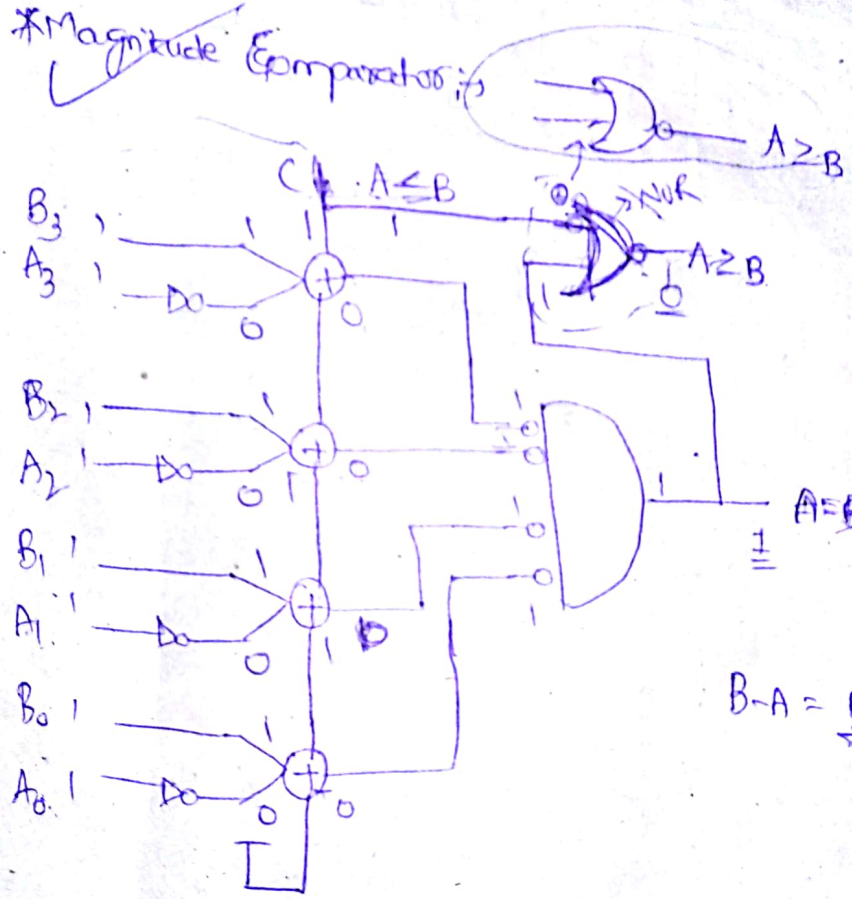


b) zero detector



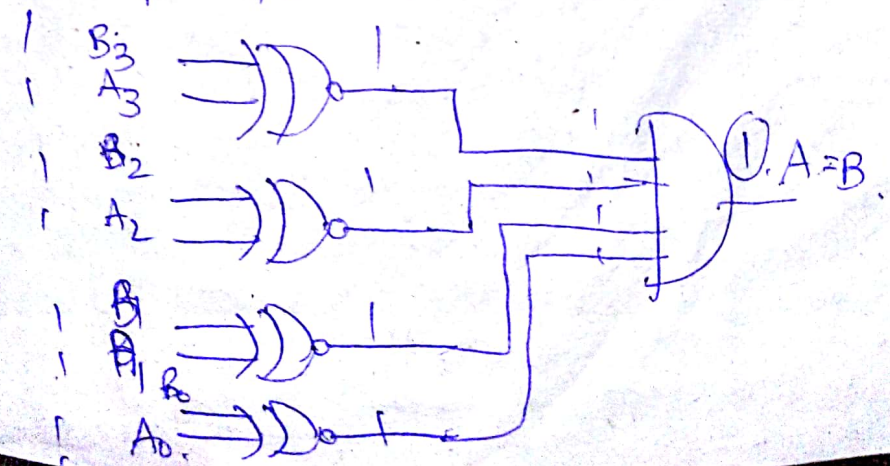


*Magnitude Comparator:-

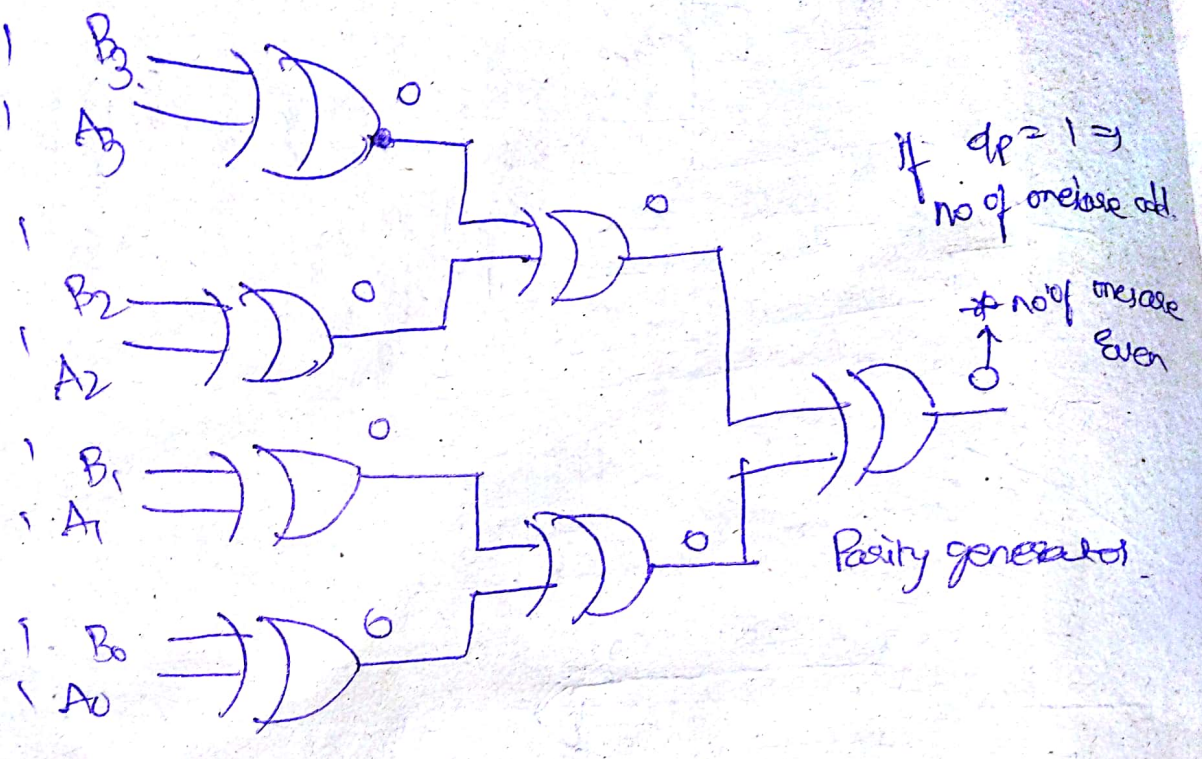


B-A = B + $\bar{A}$ + 1

*Equality Comparator:-



* Parity generator :-



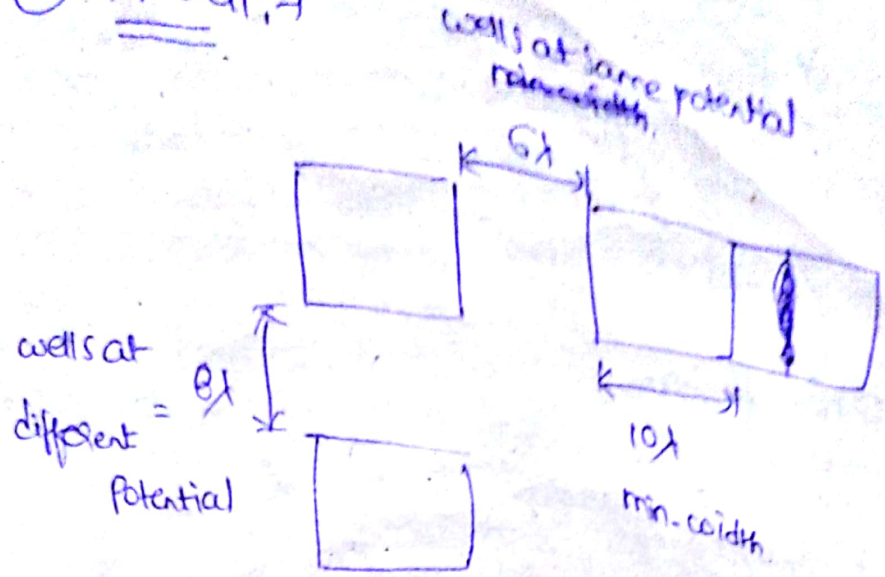
if op = 1 ⇒ no of one's odd

* no of one's even

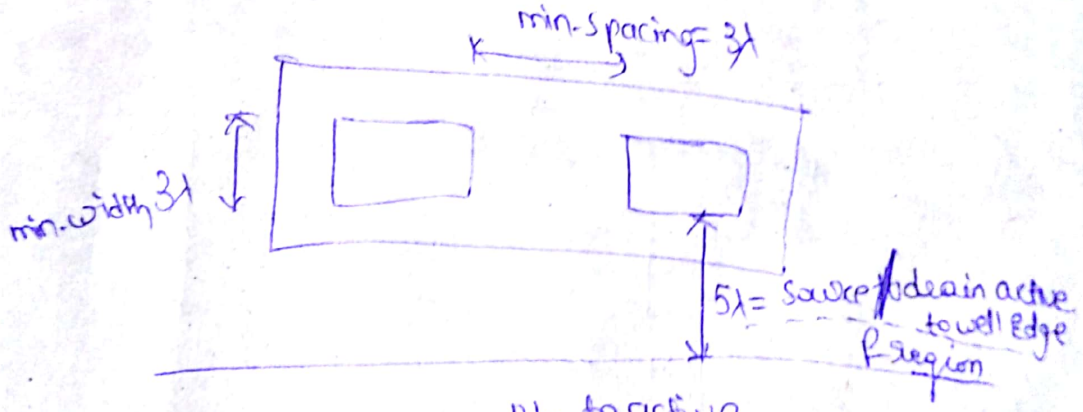
28T full adder

LAMDA BASED DESIGN RULES

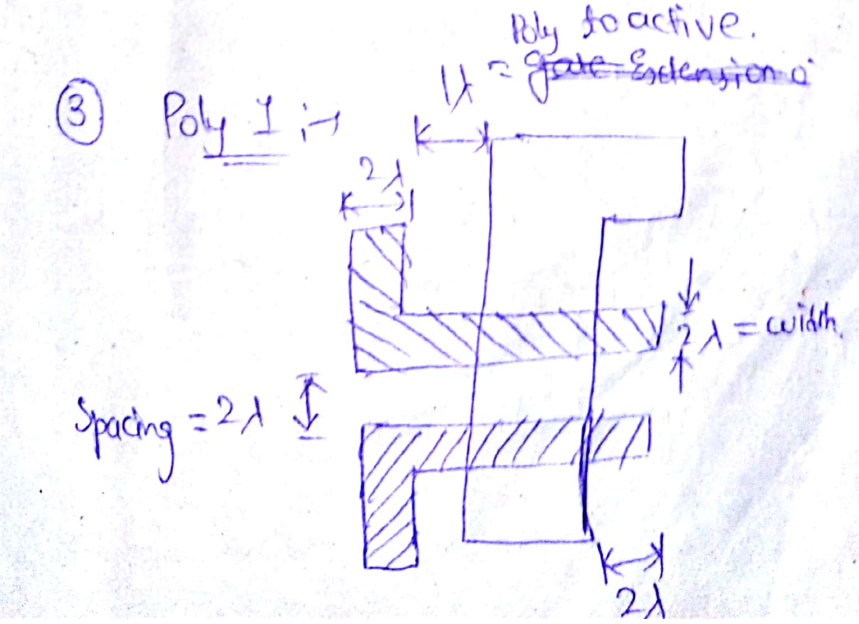
(1) N-well :-



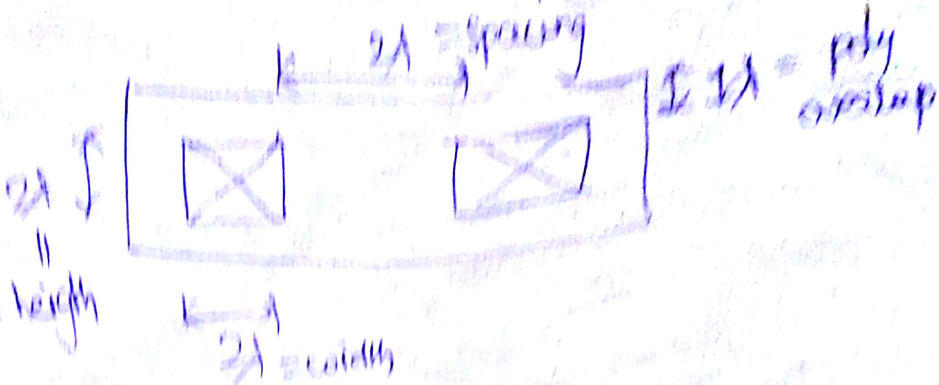
(2) Active area :-



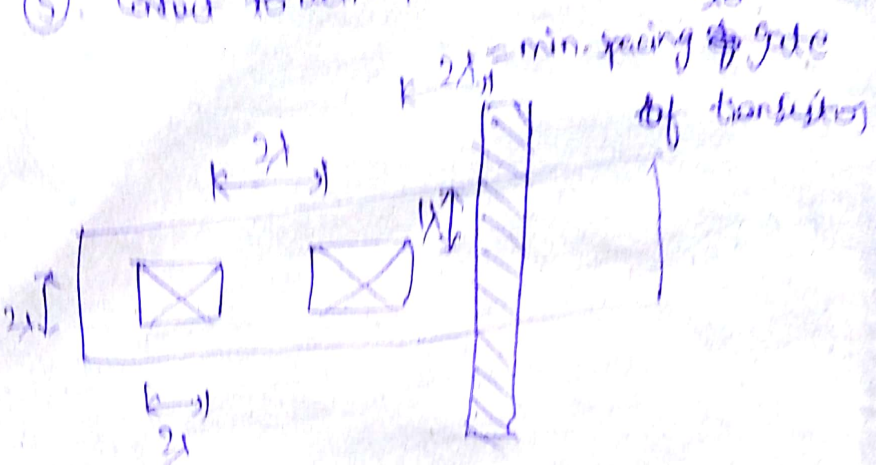
(3) Poly I :-



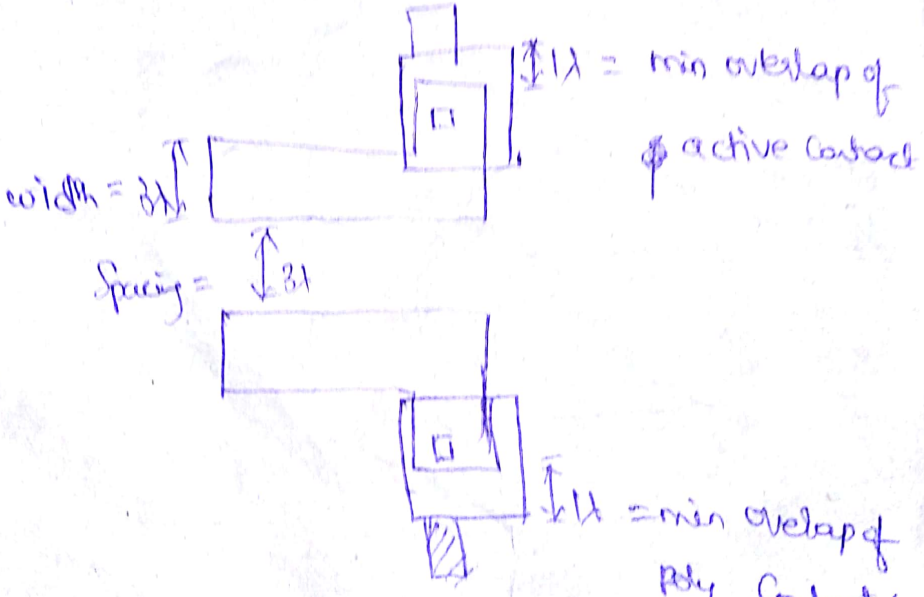
(4) Contact to poly :-

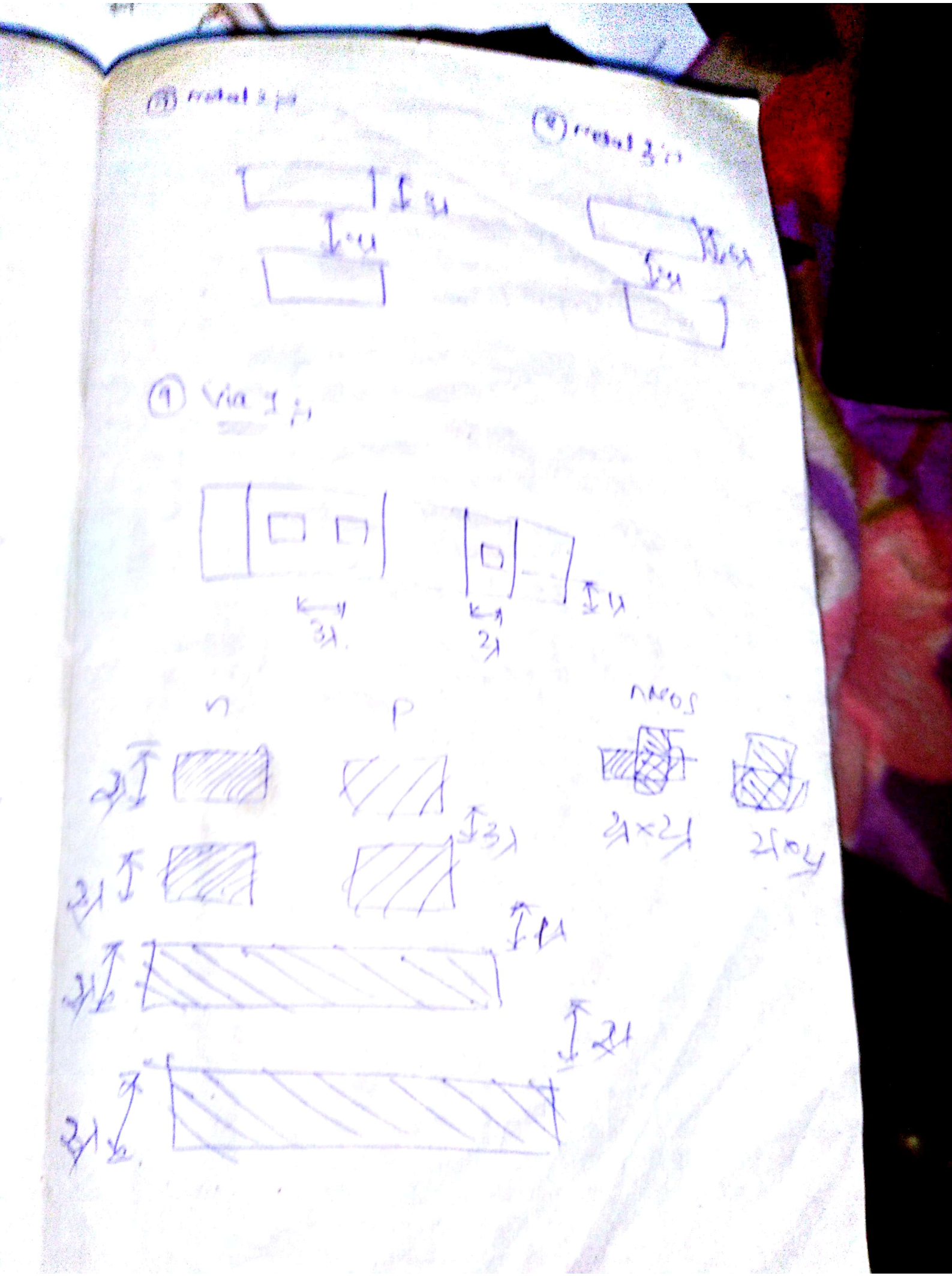


(5) Contact to active :-



(6) metal 1 :-





Unit-IV

Data path Sub systems

*Adders

→ Ripple Carry adder.

→ Carry look ahead adder.

→ Carry skip adder. / Carry bi-pass adder.

→ Carry select adder.

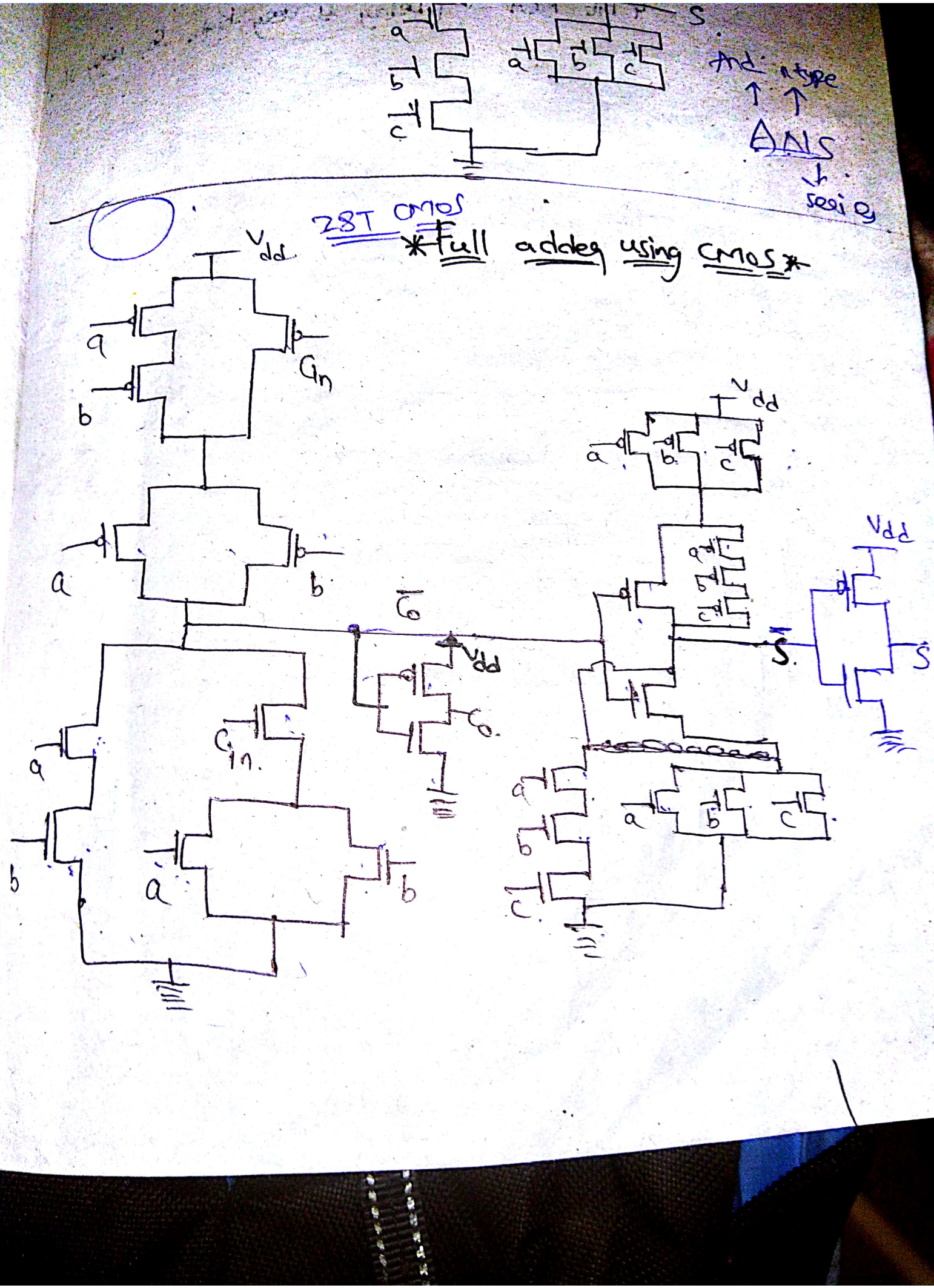
a	b	C_{in}	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$S = a \oplus b \oplus c$

$C_o = ab + bc + ca$

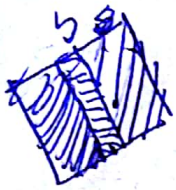
$S = abc + \bar{C}_o(a+b+c)$

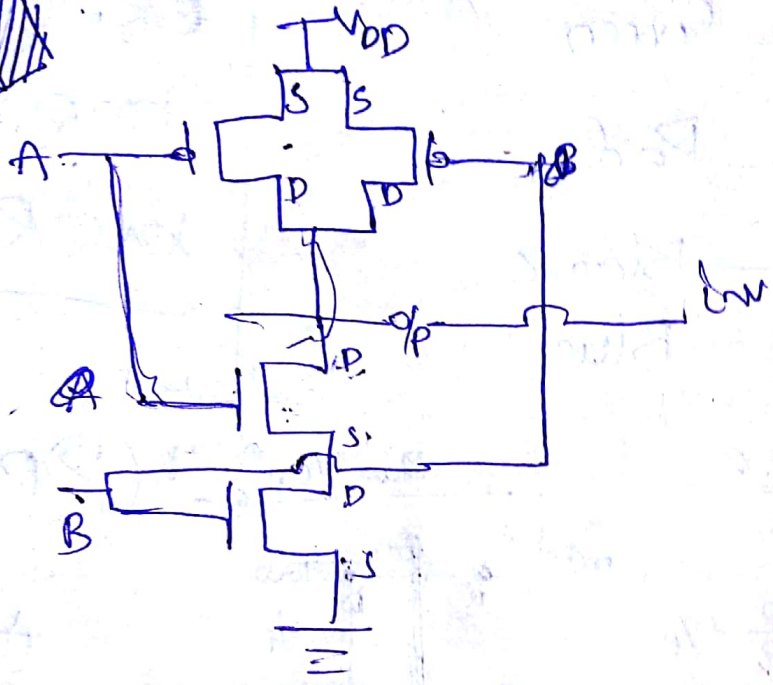
$C_o = ab + C_{in}(a+b)$

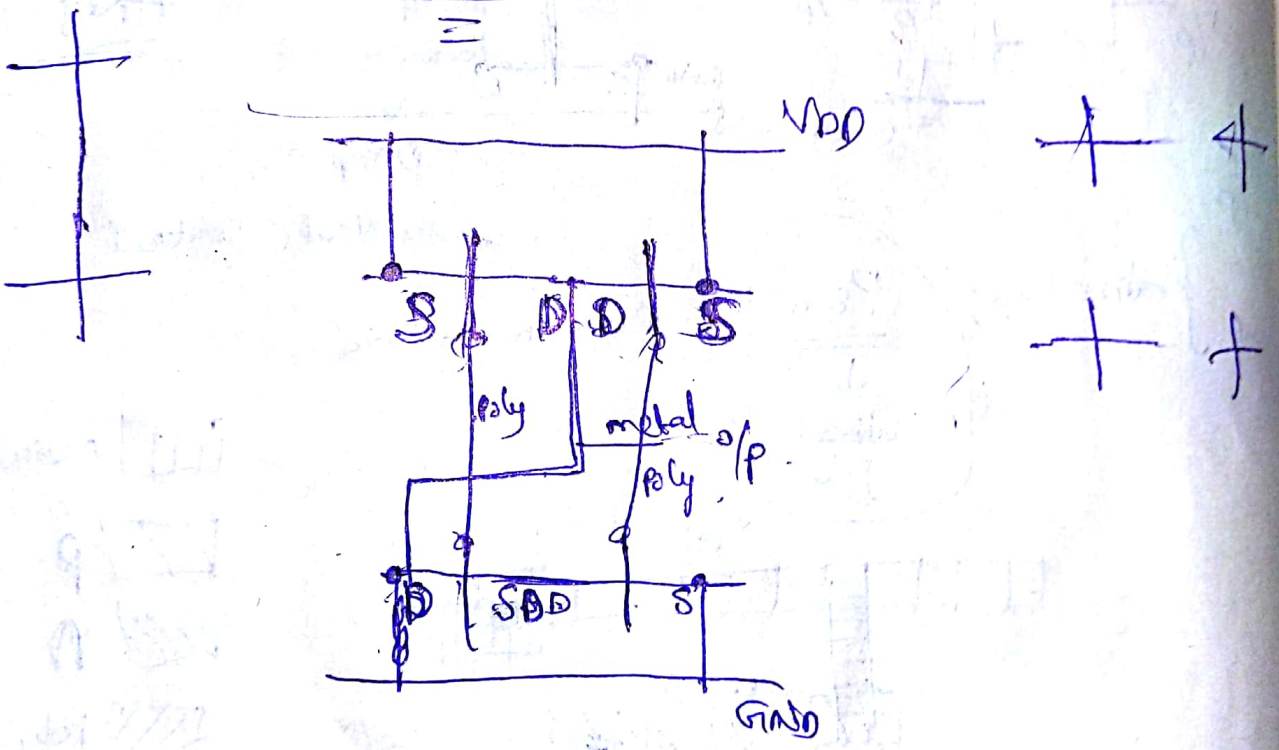


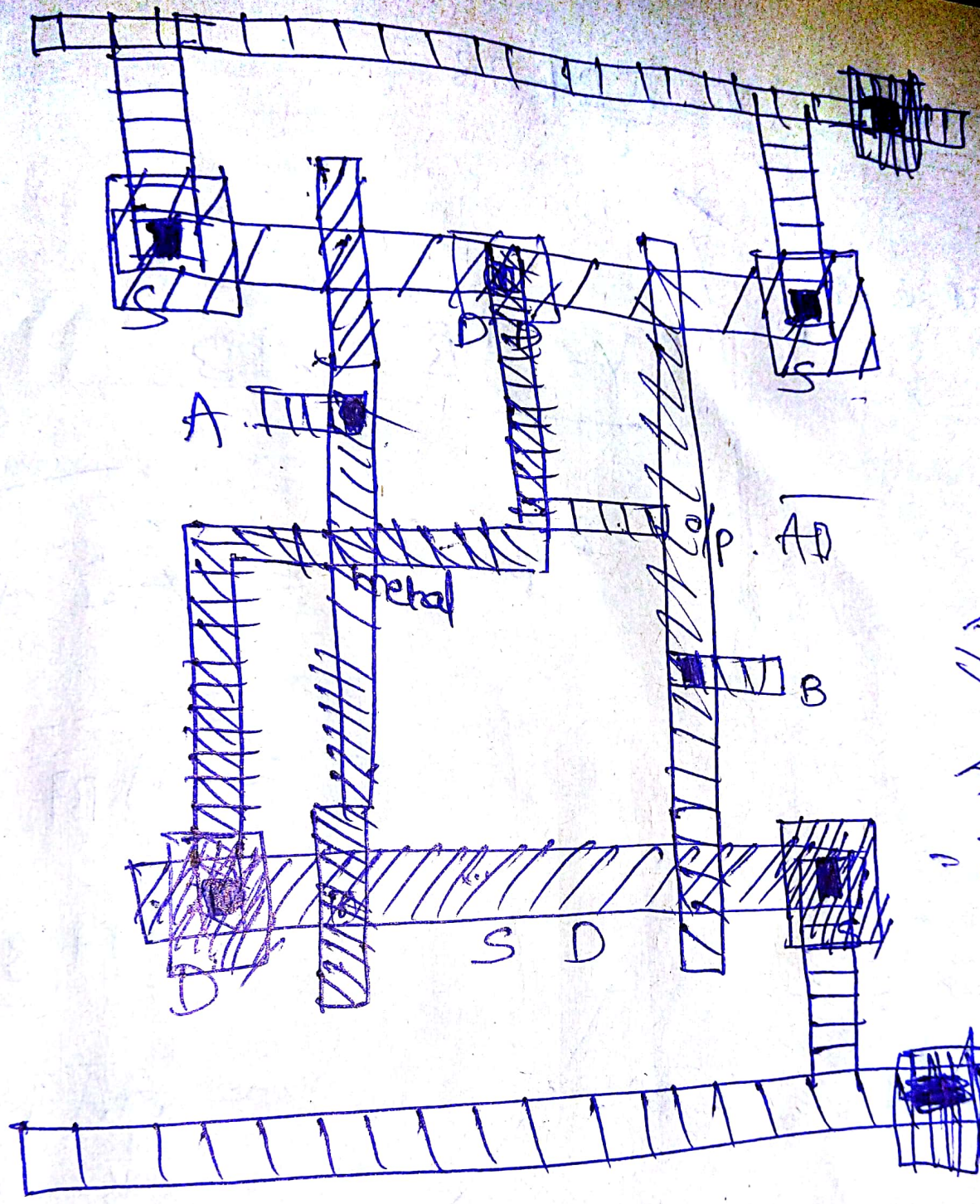
NAND = $\overline{AB}$

A N S
And n-type ↓ series

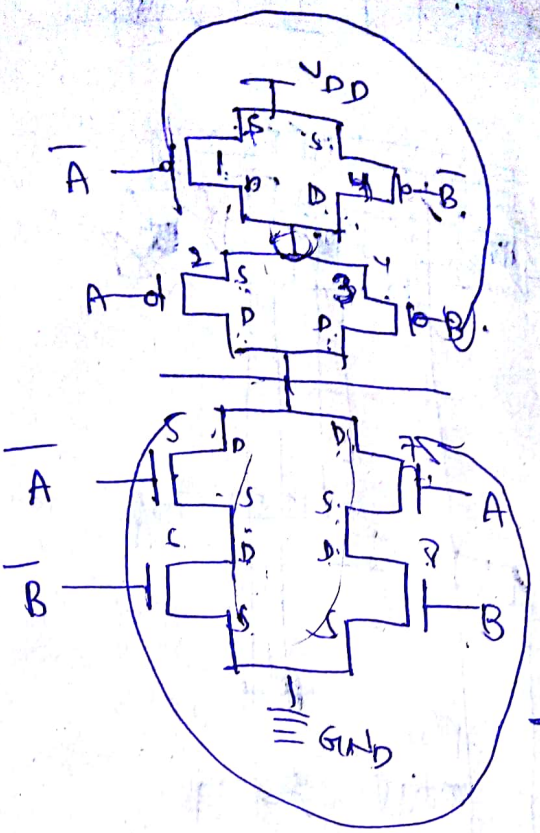


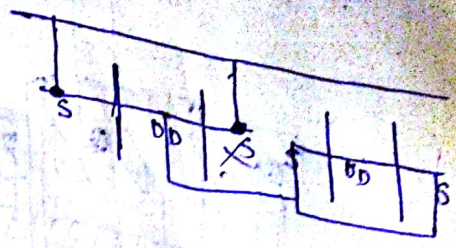


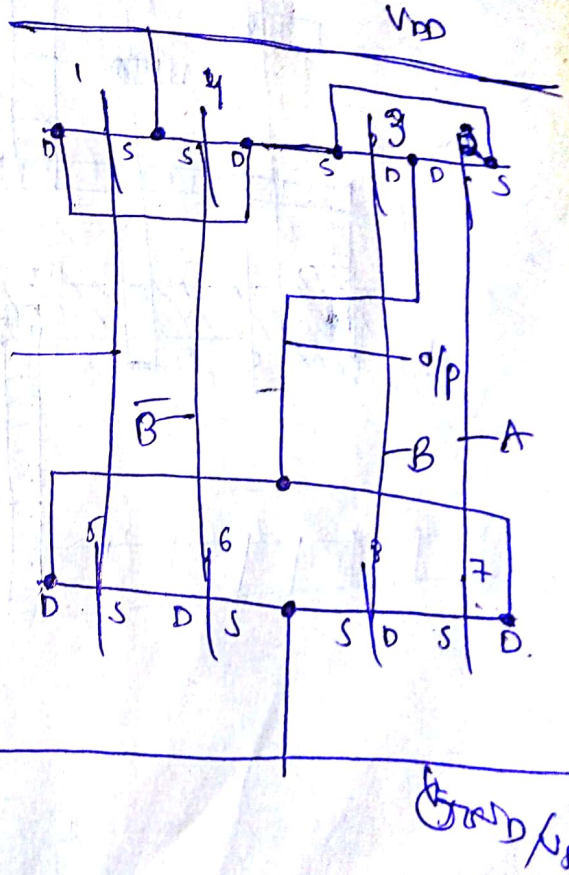




$Y = \bar{A}\bar{B} + AB$



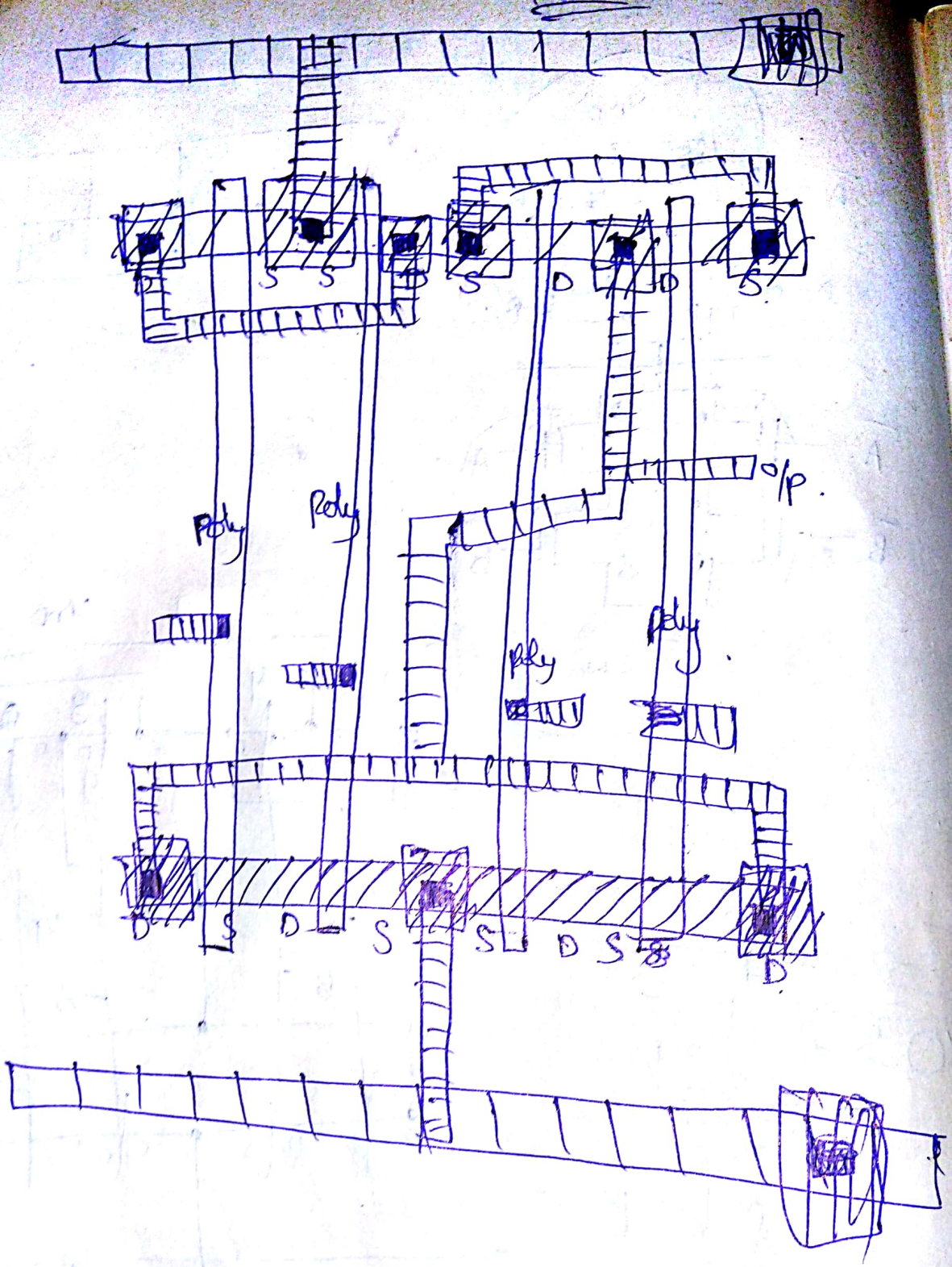




PMOS { + S
. P

NMOS { + P
. S

XOR



XNOR = XOR

Now $\rightarrow Y = A\bar{B} + \bar{B}A$ — 1/2

Y will be from $Y = \overline{A\bar{B}} \cdot \overline{\bar{B}A}$

$= (\overline{A\bar{B}})\ (\overline{\bar{B}A})$

$= (\bar{A} + B)\ (B + A)$

$= \bar{A}B + \bar{A}A + B\bar{B} + BA$

$= AB + \bar{A}\bar{B}$

OR $A + B$

$\overline{A + B} = \bar{A} \cdot \bar{B}$

I/P $\overline{A \cdot B} = \bar{A} + \bar{B} = A + B$

O/P = $A \cdot B$

And output high

Ans

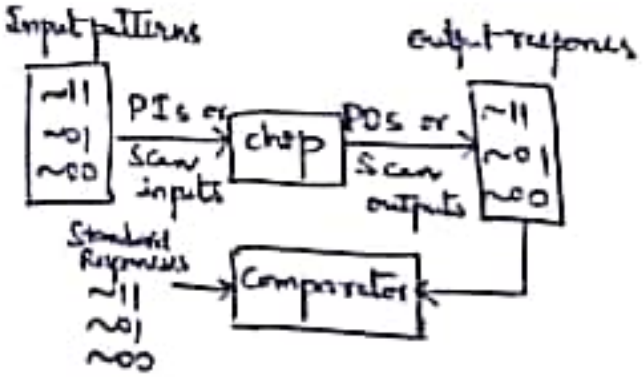
Q. Why testing is needed in VLSI design. Explain the principle of testing.

Ans: Need for testing:

1) During fabrication process several types of defects may exists such as catastrophic, crystalline. Catastrophic defect is due to contamination, resulting in destruction of all the transistors on the chip and crystalline defect is because of destruction of single transistor on chip.

2) It is necessary to test the chip from the flaws. Hence it is mandatory to check the chip regarding its performance and functionality. Identifying the fault chips is a complex job and also time consuming. The faulty chip creates huge difficulty in system debugging. It also increases the debugging cost. Therefore the Design For Testability (DFT) is necessary.

Testing principle:



- When the chip is digital, the stimuli are called Test patterns or test vectors.
- Automatic test equipment (ATE) carries out this process. A powerful computer operating under the control of a Test program, a program written in a high level language. Digital Signal Processor (DSP) used for analog Testing.
- chips are automatically fed to the tester through the wafer handler system. A probe card or membrane probe contacts pads of bare or packaged chip.

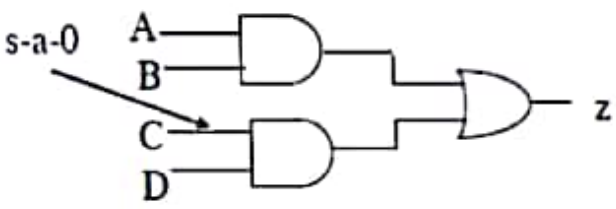
Scanned by CamScanner

Stuck-At Faults

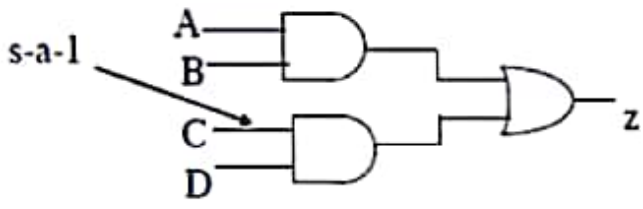
- How does a chip fail?
  - Usually failures are shorts between two conductors or opens in a conductor
  - This can cause very complicated behavior
- A simpler model: *Stuck-At*
  - Assume all failures cause nodes to be "stuck-at" 0 or 1, i.e. shorted to GND or V_{DD}
  - Not quite true, but works well in practice

Single Stuck-Line (SSL) Model

- A single node in the circuit is stuck-at 1 (s-a-1) or 0 (s-a-0).



Fault-free function z = AB+CD
Faulty function z^f = AB



Fault-free function z = AB+CD
Faulty function z^f = AB+D

Number of possible stuck-at faults in a circuit with *n* lines?

Number of faults reduced by finding equivalent classes

Bridging Faults

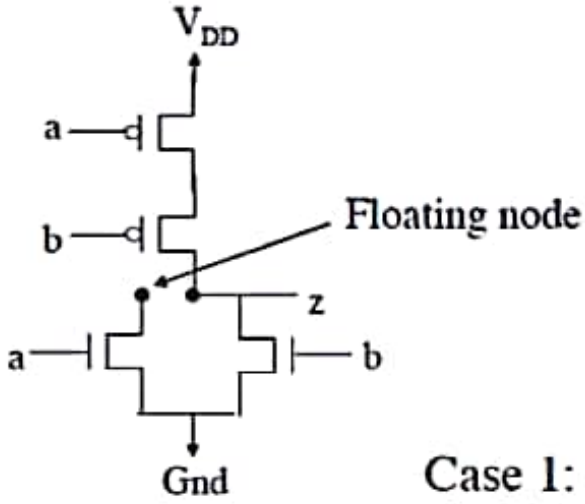
- Models short circuits, pairs of nodes considered
- Number of bridging faults?
- Feedback vs non-feedback bridging faults

A	B	z	z^f	Wired-AND	Wired-OR
0	0	0	0	0	0
0	1	0	?	0	1
1	0	1	?	0	1
1	1	1	1	1	1

z^f = ?

What are the test patterns in this example?

Stuck-Open Faults



Fault-free circuit: $z = \overline{a+b}$

Faulty circuit: $z^f = \overline{a+b} + \bar{a}b\tilde{z}$

$\tilde{z}$: Previous value of z

Case 1: a = b = 1, z pulled down to 0

Case 2: a = 1, b =0, z retains previous state

A test for a stuck-open fault requires two patterns

{ab = 00, ab = 10}

15.6 Design for Testability

The keys to designing circuits that are testable are controllability and observability. Restated, controllability is the ability to set (to 1) and reset (to 0) every node internal to the circuit. Observability is the ability to observe, either directly or indirectly, the state of any node in the circuit. Good observability and controllability reduce the cost of manufacturing testing because they allow high fault coverage with relatively few test vectors. Moreover, they can be essential to silicon debug because physically probing internal signals has become so difficult.

We will first cover three main approaches to what is commonly called *Design for Testability* (DFT). These may be categorized as follows:

- *Ad hoc* testing
- Scan-based approaches
- Built-in self-test (BIST)

15.6.1 *Ad Hoc* Testing

Ad hoc test techniques, as their name suggests, are collections of ideas aimed at reducing the combinational explosion of testing. They are summarized here for historical reasons. They are only useful for small designs where scan, ATPG, and BIST are not available. A complete scan-based testing methodology is recommended for all digital circuits. Having said that, the following are common techniques for ad hoc testing:

- Partitioning large sequential circuits
- Adding test points
- Adding multiplexers
- Providing for easy state reset

A technique classified in this category is the use of the bus in a bus-oriented system for test purposes. Each register has been made loadable from the bus and capable of being driven onto the bus. Here, the internal logic values that exist on a data bus are enabled onto the bus for testing purposes.

Frequently, multiplexers can be used to provide alternative signal paths during testing. In CMOS, transmission gate multiplexers provide low area and delay overhead.

Any design should always have a method of resetting the internal state of the chip within a single cycle or at most a few cycles. Apart from making testing easier, this also makes simulation faster as a few cycles are required to initialize the chip.

15.5.6 Fault Coverage

A measure of goodness of a set of test vectors is the amount of *fault coverage* it achieves. That is, for the vectors applied, what percentage of the chip's internal nodes were checked? Conceptually, the way in which the fault coverage is calculated is as follows. Each circuit node is taken in sequence and held to 0 (S-A-0), and the circuit is simulated with the test vectors comparing the chip outputs with a *known good machine*—a circuit with no nodes artificially set to 0 (or 1). When a discrepancy is detected between the *faulty machine* and the good machine, the fault is marked as detected and the simulation is stopped. This is repeated for setting the node to 1 (S-A-1). In turn, every node is stuck (artificially) at 1 and 0 sequentially. The fault coverage of a set of test vectors is the percentage of the total nodes that can be detected as faulty when the vectors are applied. To achieve world-class quality levels, circuits are required to have in excess of 98.5% fault coverage. The *Verification Methodology Manual* [Bergeron05] is the bible for fault coverage techniques.

15.5.7 Automatic Test Pattern Generation (ATPG)

Historically, in the IC industry, logic and circuit designers implemented the functions at the RTL or schematic level, mask designers completed the layout, and test engineers wrote the tests. In many ways, the test engineers were the Sherlock Holmes of the industry, reverse engineering circuits and devising tests that would test the circuits in an adequate manner. For the longest time, test engineers implored circuit designers to include extra circuitry to ease the burden of test generation. Happily, as processes have increased in density and chips have increased in complexity, the inclusion of test circuitry has become less of an overhead for both the designer and the manager worried about the cost of the die. In addition, as tools have improved, more of the burden for generating tests has fallen on the designer. To deal with this burden, *Automatic Test Pattern Generation* (ATPG) methods have been invented. The use of some form of ATPG is standard for most digital designs.

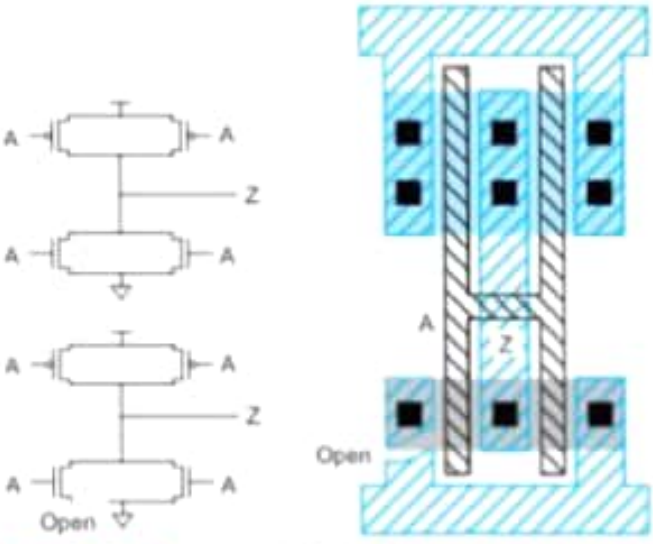


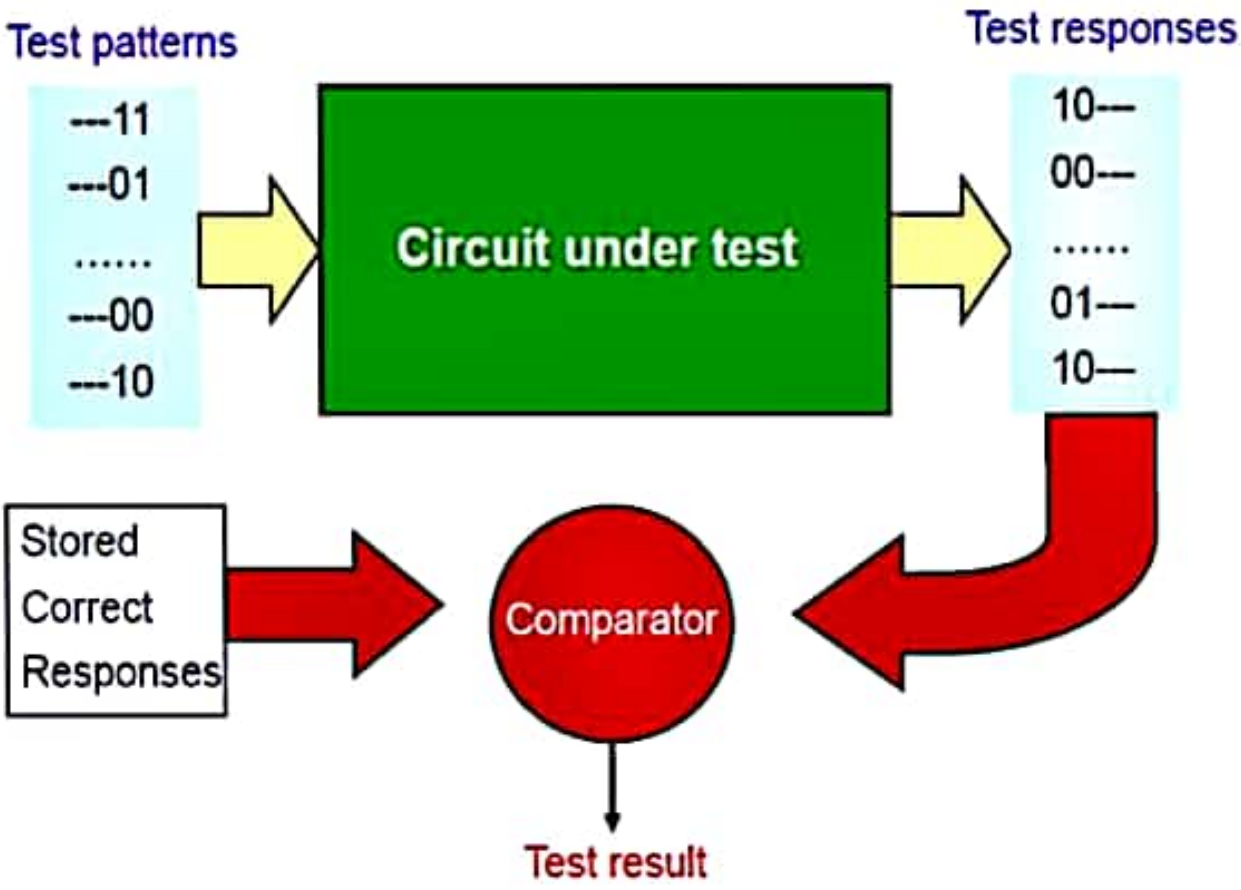
FIGURE 15.15 An example of a delay fault

Commercial ATPG tools can achieve excellent fault coverage. However, they are computation-intensive and often must be run on servers or compute farms with many parallel processors. Some tools use statistical algorithms to predict the fault coverage of a set of vectors without performing as much simulation. Adding scan and built-in self-test, as described in Section 15.6, improves the observability of a system and can reduce the number of test vectors required to achieve a desired fault coverage.

15.5.8 Delay Fault Testing

The fault models dealt with until this point have neglected timing. Failures that occur in CMOS could leave the functionality of the circuit untouched, but affect the timing. For

How to Test Chips?

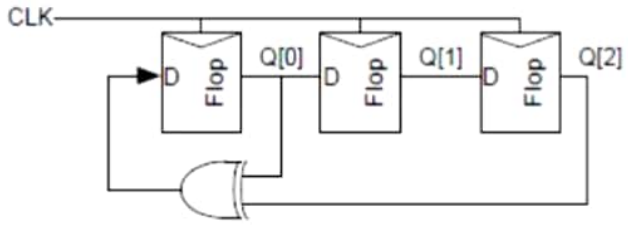


Built-in Self-test

- Built-in self-test lets blocks test themselves
  - Generate pseudo-random inputs to comb. logic
  - Combine outputs into a *syndrome*
  - With high probability, block is fault-free if it produces the expected syndrome

Linear Feedback Shift Register

- Shift register with input taken from XOR of state
- *Pseudo-Random Sequence Generator*



Step	Q
0	111
1	110
2	101
3	010
4	100
5	001
6	011
7	111 (repeats)

4.4.1 Architecture of Braun multiplier

An $n \times n$-bit Braun multiplier requires $n(n-1)$ adders and n^2 AND gates [1, 5]. One efficient implementation of the Braun multiplier is the regular layout of the adder array as shown in Fig. 4.2. The internal structure of the full adder used in the Braun multiplier is depicted in Fig. 4.3 [6]. This makes Braun multipliers ideal for Very Large Scale Integration (VLSI) and Application Specific Integrated Circuit (ASIC) realization.

Each of the $X_i Y_j$ product bits is generated in parallel with the AND gates [1]. Each partial product can be added to the previous sum of partial products by using a row of adders. The carry-out signals are shifted one bit to the left and are then added to the sums of the first adder and the new partial product. The shifting of the carry-out bits to the left is done by a Carry Save Adder (CSA). As the carry bits are passed diagonally downward to the next adder stage, there is no horizontal carry propagation for the first four rows. Instead, the respective carry bit is "saved" for the subsequent adder stage. Ripple Carry Adders (RCA) are used at the final stage of the array to output the final result.

4.4.2 Performance of Braun multiplier

The Braun multiplier performs well for unsigned operands that are less than 16 bits, in terms of speed, power and area. Besides, it has a simple and regular structure as compared to other multiplier schemes. However, the number of components required in building the Braun multiplier increases quadratically with the number of bits. This makes the Braun multiplier inefficient and so it is rarely employed while handling large operands. Another pitfall of the Braun multiplier is its potential susceptibility to glitching problems at the last stage of the full adders due to the exploitation of the Ripple Carry Adders (RCA).

4.4.3 Speed consideration

The delay of the Braun multiplier is dependent on the delay of the full adder cell and also on the final adder in the last row. In the multiplier array, a full adder with balanced carry and sum delays is desirable because the sum and carry signals are both in the critical path. The speed and power of the full adder are very important for large arrays.

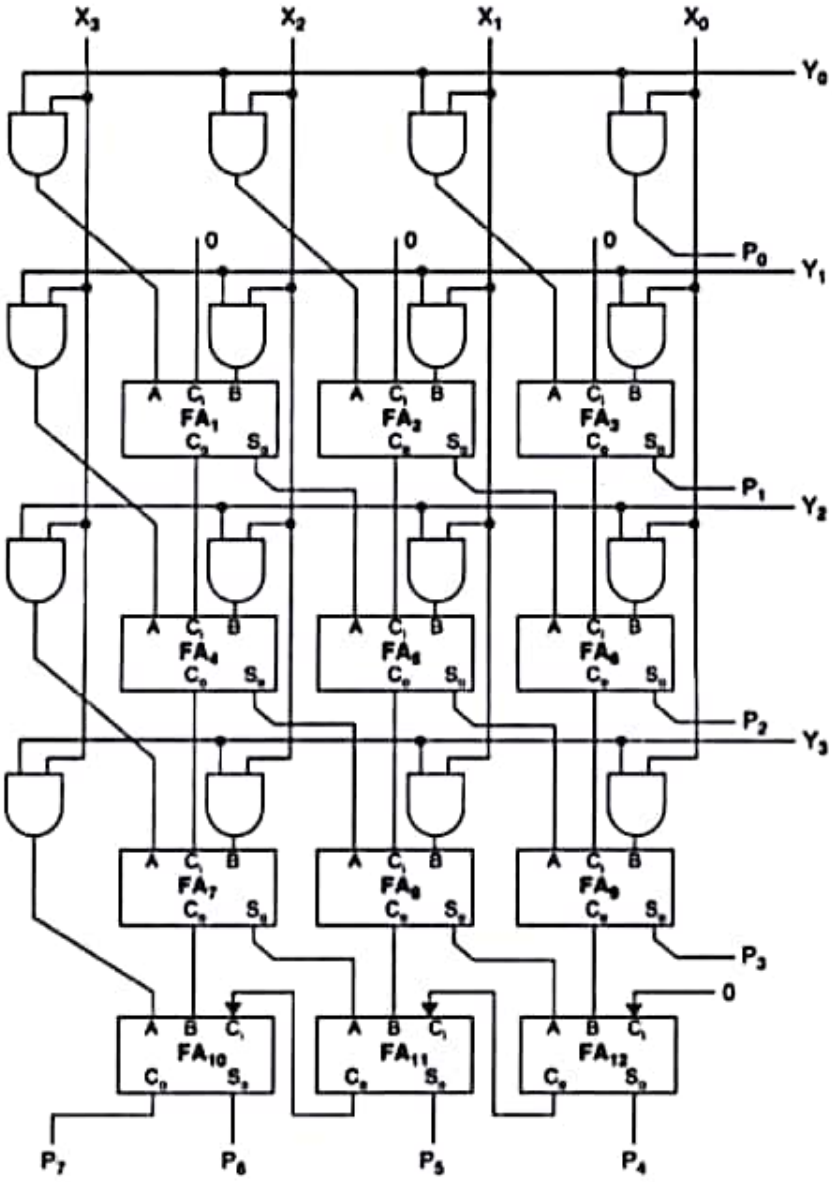
The worst-case multiplication time of a Braun multiplier can be expressed as [5]

$$t_{\text{Braun}} = (n-1)t_{\text{carry-save}} + t_{\text{AND}} + (n-1)t_{\text{Ripple-Carry}} \tag{4.12}$$

where $t_{\text{carry-save}}$ = time required to generate Carry-out (C_{out}) or Sum (S_{out}) at the output after the inputs are supplied to a CSA

$t_{\text{Ripple-Carry}}$ = time taken for the Carry-out (C_{out}) or Sum (S_{out}) to be generated at the output after the inputs are supplied to a RCA

t_{AND} = delay of an AND gate



X: 4-bit multiplicand
Y: 4-bit multiplier
P: 8-bit product of X and Y
$P_n = X_i Y_j$ is a product bit

Figure 4.2 Schematic diagram of a 4 × 4-bit Braun multiplier.

27.1 Array Multiplication

parallel multiplier is based on the observation that partial products in the ultiplication process may be independently computed in parallel. For ample, consider the unsigned binary integers X and Y.

$$X = \sum_{i=0}^{m-1} X_i 2^i$$

$$Y = \sum_{j=0}^{n-1} Y_j 2^j$$

The product is found by

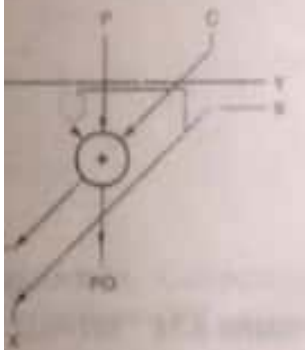
$$P = X \times Y = \sum_{i=0}^{m-1} X_i 2^i \cdot \sum_{j=0}^{n-1} Y_j 2^j$$

$$= \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (X_i Y_j) 2^{i+j}$$

$$= \sum_{k=0}^{m+n-1} P_k 2^k.$$

Thus P_k are the partial product terms called summands. There are mn summands, which are produced in parallel by a set of mn AND gates. For 4-bit numbers, the expression above may be expanded as in Table 8.2.

An $n \times n$ multiplier requires $n(n-2)$ full adders, n half adders, and n^2 AND gates. The worst-case delay associated with such a multiplier is $(2n+1)\tau_g$, where τ_g is the worst-case adder delay. Figure 8.35 shows a cell that may be used to construct a parallel multiplier. The X_i term is propagated diagonally from top right to bottom left, while the Y_j term is propagated horizontally. Incoming partial products enter at the top. Incoming *CARRY IN* values enter at the top right of the cell. The bit-wise AND is performed in the cell, and the *SUM* is passed to the next cell below. The *CARRY OUT* is passed to the bottom left of the cell. Figure 8.36 shows the multiplier array with the partial products enumerated. This arrangement may be drawn as a square array, as shown in Fig. 8.37, which is the most convenient for implementation. In this version the degeneracy of the first two rows of the multiplier are shown. The first row of the multiplier adders has been replaced with AND gates while the second row employs half-adders rather than full adders. This optimization might not be done if a completely regular multiplier were required (i.e., one

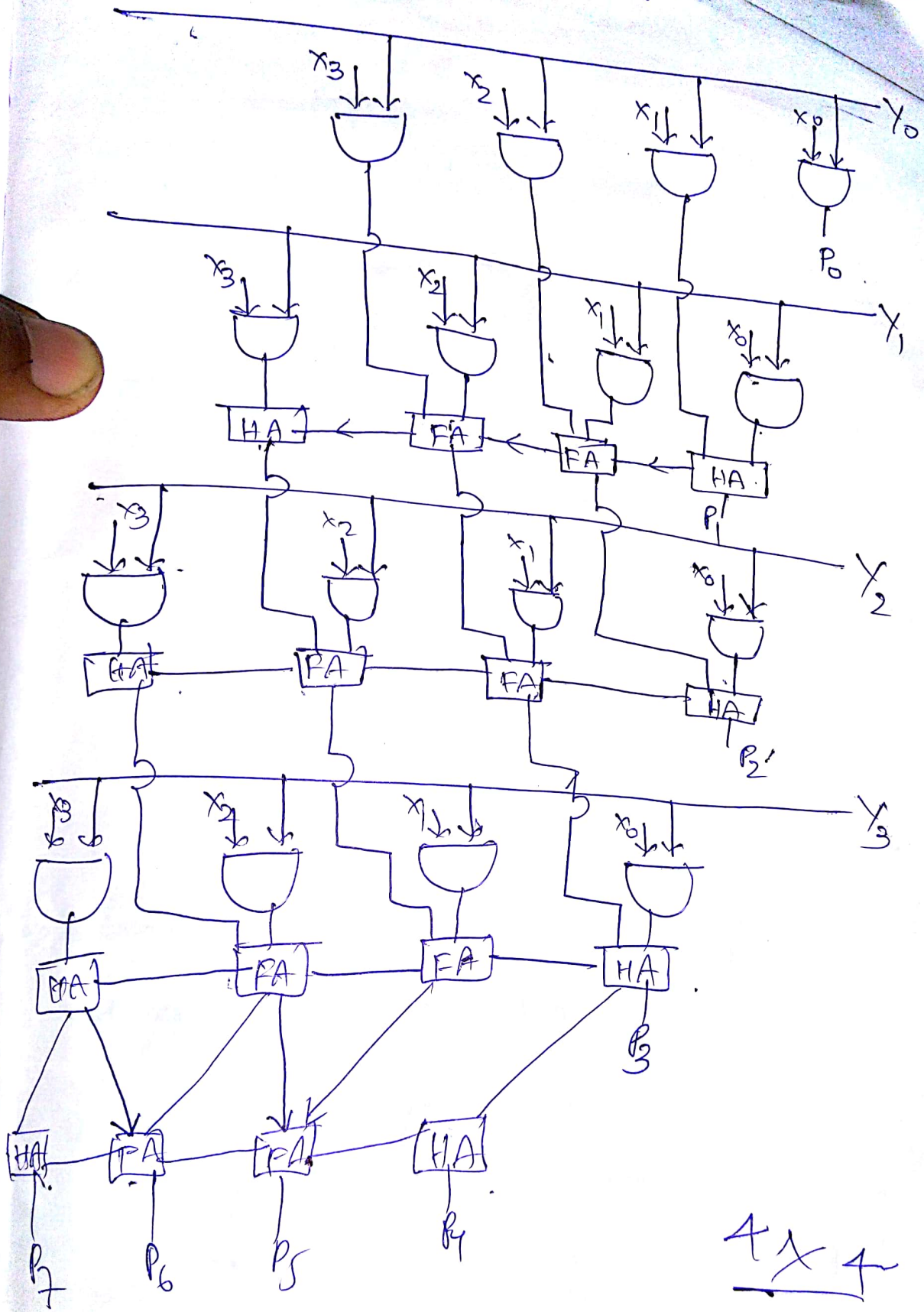


URE 8.35 Array
tiplier cell

TABLE 8.2 4-bit Multiplier Partial Products

				X3	X2	X1	X0	Multiplicand
				Y3	Y2	Y1	Y0	Multiplier
				X3Y0	X2Y0	X1Y0	X0Y0	
			X3Y1	X2Y1	X1Y1	X0Y1		
		X3Y2	X2Y2	X1Y2	X0Y2			
	X3Y3	X2Y3	X1Y3	X0Y3				
P7	P6	P5	P4	P3	P2	P1	P0	Product

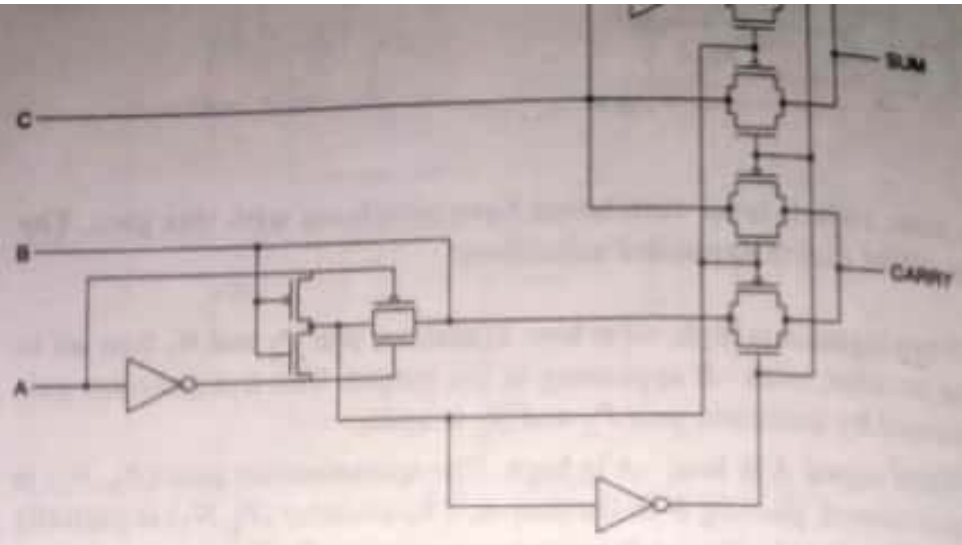
ARRAY Multiplier



4 × 4

n(n−2) FA.

n



gure 8.13 Optimized-
ea TG adder

mate goal. Two transistors may be eliminated by using an inverter on the output of the XOR gate. In addition with some optimization, the output buffers may be eliminated, as shown in Fig. 8.13.[4]

8.2.1.5 Carry-Lookahead Adders

The linear growth of adder carry-delay with the size of the input word for an n-bit adder may be improved by calculating the carries to each stage in parallel. The carry of the ith stage, C_i, may be expressed as

$$C_i = G_i + P_i \cdot C_{i-1}, \tag{8.6}$$

where

$$G_i = A_i \cdot B_i \qquad \textit{generate signal}$$
$$P_i = A_i + B_i \qquad \textit{propagate signal.}$$

Expanding this yields

$$C_i = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \ldots + P_i \ldots P_1 C_0. \tag{8.7}$$

The sum S_i is generated by

$$S_i = C_{i-1} \oplus A_i \oplus B_i$$
$$\text{or } C_{i-1} \oplus P_i \text{ (if } P_i = A_i \oplus B_i). \tag{8.8}$$

The size and fan-in of the gates needed to implement this carry-lookahead scheme can clearly get out of hand. As a result, the number of stages of look-

ead is usually limited to about four. For four stages of lookahead, the propriate terms are

$$C_0 = G_0 + P_0 CI$$
$$C_1 = G_1 + P_1 G_0 + P_1 P_0 CI$$
$$C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 CI$$
$$C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 CI. \quad (8.9)$$

Figure 8.14 shows a generic carry-lookahead adder. The *PG* generation d *SUM* generation circuits surround a carry-generate block. A possible plementation of the carry gate for this kind of carry-lookahead adder for 4 s is shown in Fig. 8.15. Note that the gates have been partitioned to keep e number of inputs less than or equal to four. This is typical of the type of rry lookahead that would be used in a gate-array or standard-cell design. e circuit and layout are quite irregular. Taking the term of C_3, we note that may be expressed as

$$C_3 = G_3 + P_3 \cdot (G_2 + P_2 \cdot (G_1 + P_1 \cdot (G_0 + P_0 \cdot CI))). \quad (8.10)$$

his function may be implemented as a domino CMOS (nMOS) gate, as own in Fig. 8.16(d). Carry $C_0 - C_2$ are generated similarly. Note that the rst-case delay path in this circuit has six n-transistors in series. A high- ed static version of the carry-lookahead gate for C_3 is shown in Fig. 8.17.[5]

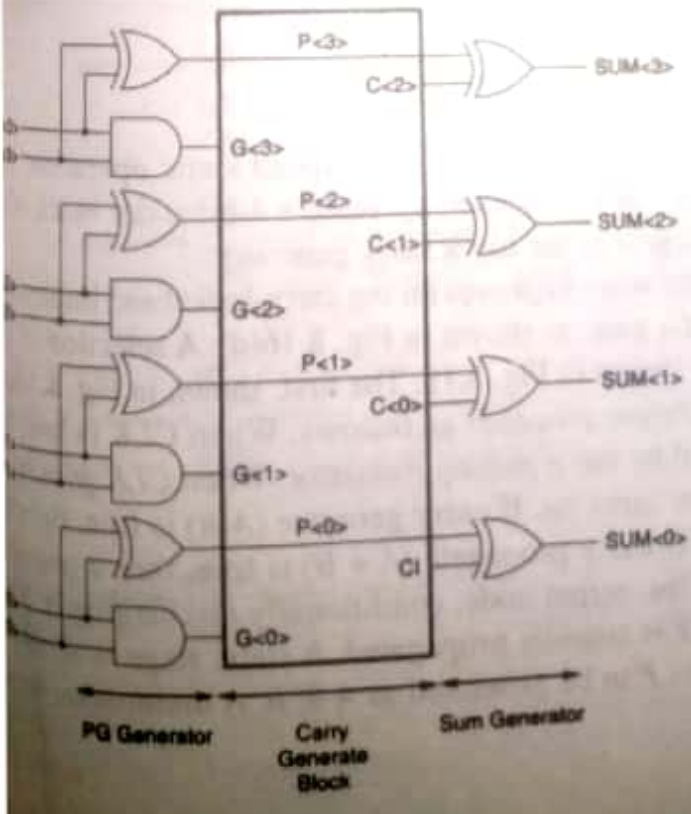


FIGURE 8.14 G
carr d

- What is the next step in the evolution of programmable logic?

  –More gates!

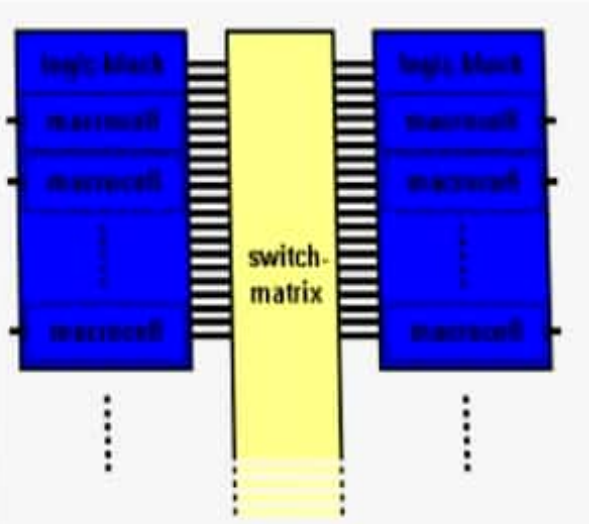
- How do we get more gates?
- We could put several PALs on one chip and put an interconnection matrix between them!!
  - This is called a ***Complex PLD (CPLD).***

Complex PLDs

•To reach more complexity, the logical consequence of the SPLD-architecture was the CPLD-architecture.

•A "complex programmable logic device" (CPLD) contains many SPLD-like (PAL-like) devices interconnected via a programmable switch matrix.

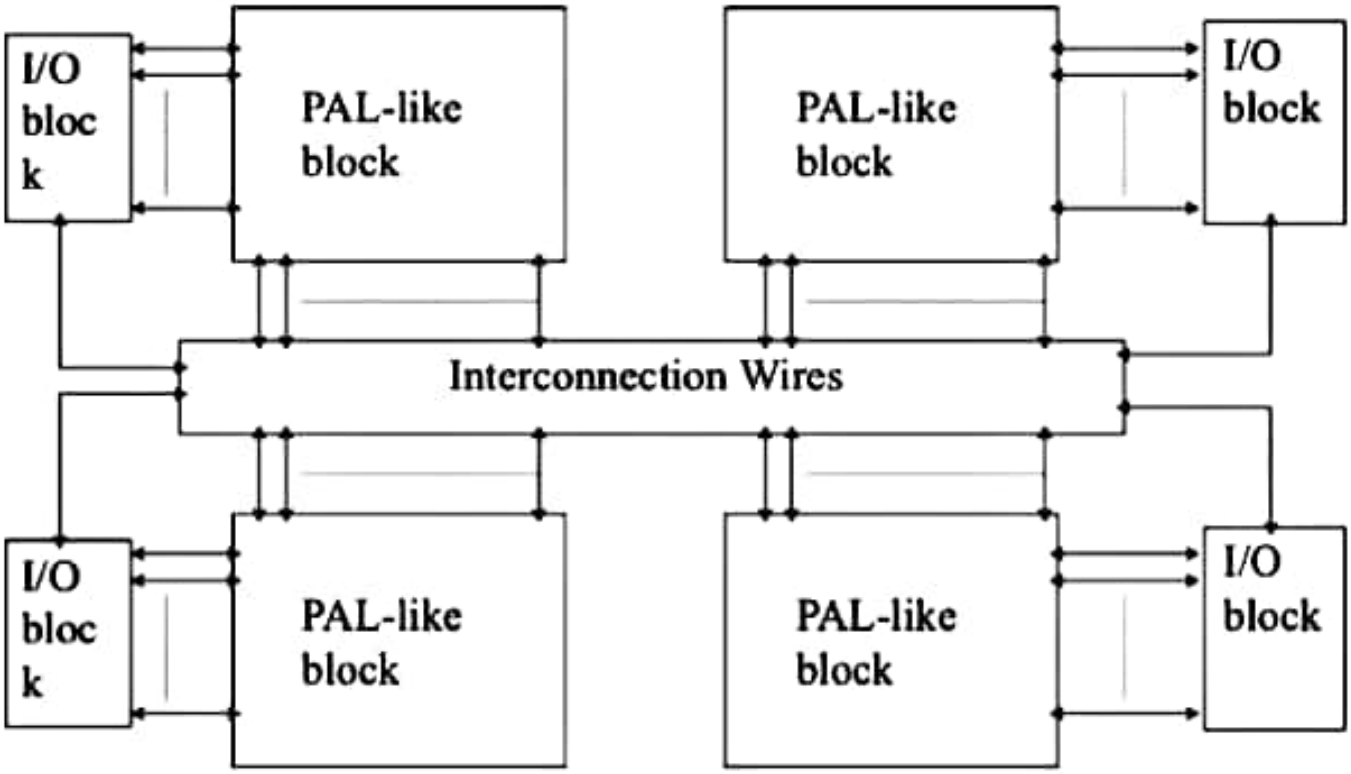
•The SPLD-like devices were called logic-blocks, which contain many SPLD-like macrocells.



CPLD-architecture

Architecture of CPLD

Architecture of CPLD



Complex PLDs

- Significant characteristics for the CPLD-architecture :
  - » product terms generated in programmable macrocells
  - » typically one dedicated flip-flop per macrocell
  - » many macrocells per logic-block
  - » typically all logic-blocks identical
  - » minimum two logic-blocks per device
  - » routing between logic-blocks via global switch matrix

Complex PLDs

- Main-advantages :
  - » predictable timing
  - » fast pin-to-pin delay
  - » efficient resource utilization by switch-matrix
  - » medium design complexities possible

Main-disadvantages :

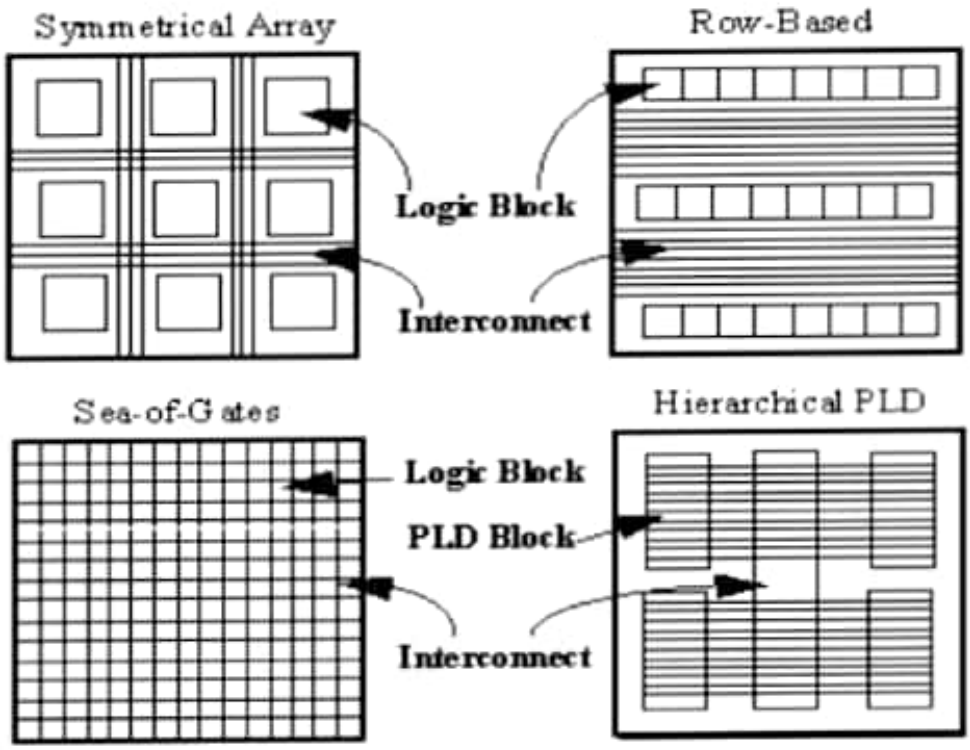
» higher complexities need a very complex (expensive) switch-matrix

What is an FPGA?

- Field Programmable Gate Array
- Fully programmable alternative to a customized chip
- Used to implement functions in hardware
- Also called a Reconfigurable Processing Unit (RPU)

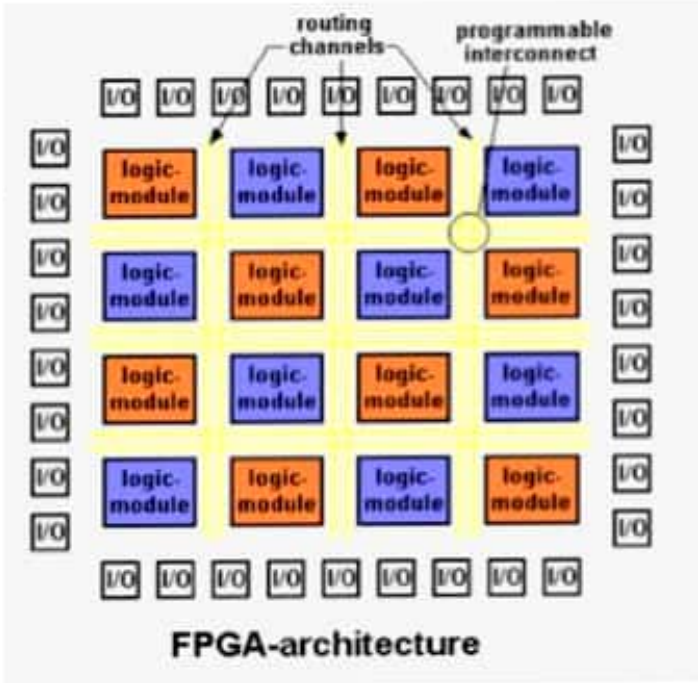
The FPGA approach to arrange primitive logic elements (logic cells) arrange in rows/columns with programmable routing between them.

FPGA Architectures



FPGA

- The FPGA-architecture consists of many logic-modules, which are placed in an array-structure. The channels between the logic-modules are used for routing.
- The array of logic-modules is surrounded by programmable I/O-modules and connected via programmable interconnects.
- This freedom of routing allows every logic-module to reach every other logic-module or I/O-module.
- The worldwide first PLD with FPGA-architecture was developed by Xilinx in 1984.

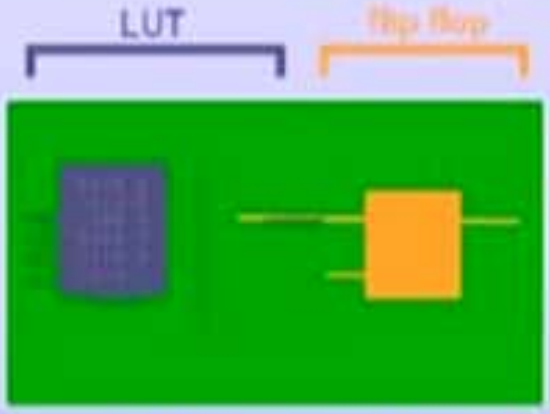


FPGA-architecture

- Applications of FPGAs include digital signal processing, software-defined radio, aerospace and defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy, metal detection and a growing range of other areas.

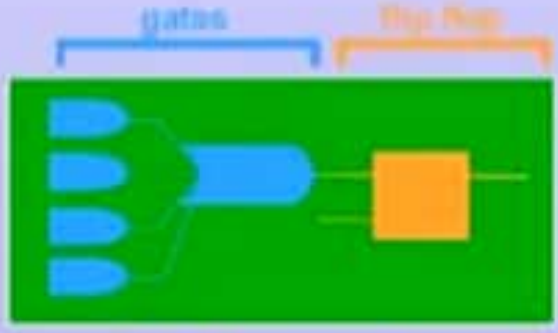
FPGA Types

- ***2 types of FPGAs***
- Reprogrammable (SRAM-based)
  - Xilinx, Altera, Lattice, Atmel



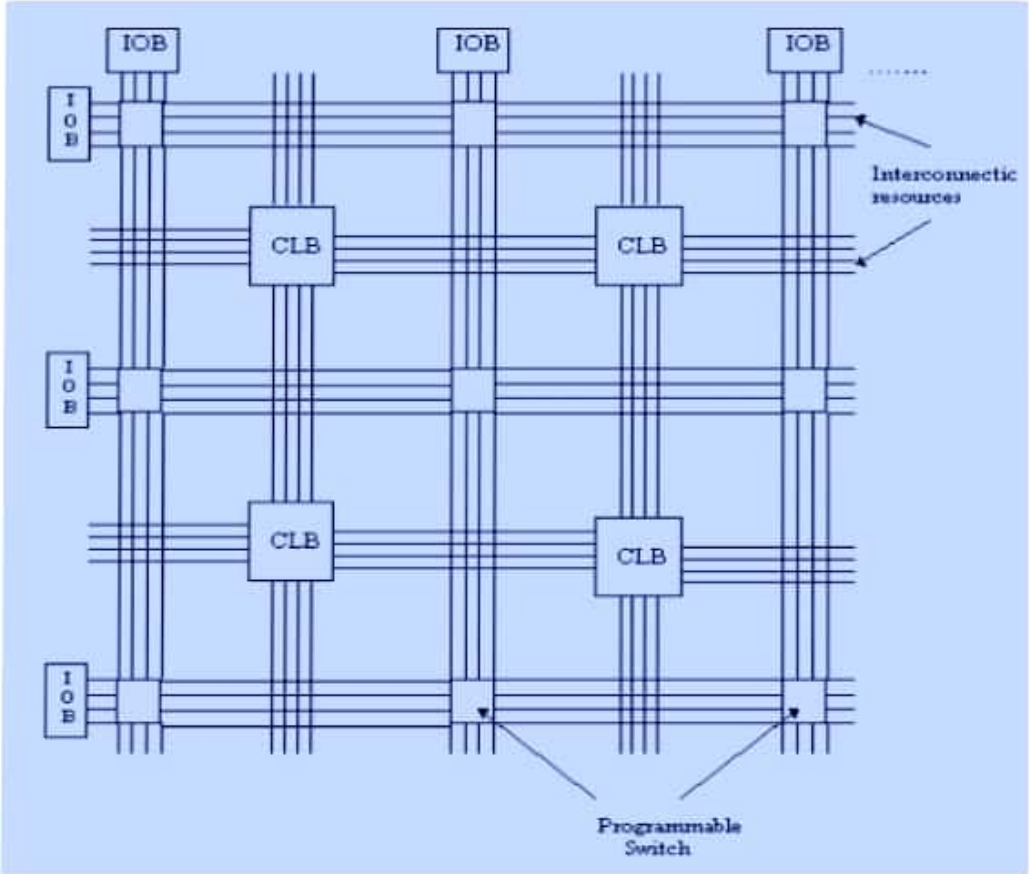
SRAM logic cell

- One-time Programmable (OTP)**(Anti-fuse technology)**
  - Actel, Quicklogic



OTP logic cell

Xilinx FPGA



FPGA	CPLD
1) FPGA architecture contains gate array like structures.	1) CPLD architecture contains PAL like structures.
2) Its density range is from medium to high.	2) Its density range is from low to medium.
3) FPGA speed depends on the application.	3) CPLD speed is very high.
4) For Interconnection purpose routing channels are used.	4) For Interconnection purpose crossbar connections are used.
5) In FPGA power consumption is medium.	5) In CPLD power consumption is high.
6) Once supply is removed, data is lossed in FPGA, i.e. they are volatile.	6) CPLD contains on chip non-volatile memory.
7) FPGAs are internally based on look-up tables (LUTs).	7) CPLDs form the logic functions with sea-of-gates.

D S I P

B

D S P I