Design of Computational Elements

In this lecture, you will learn to

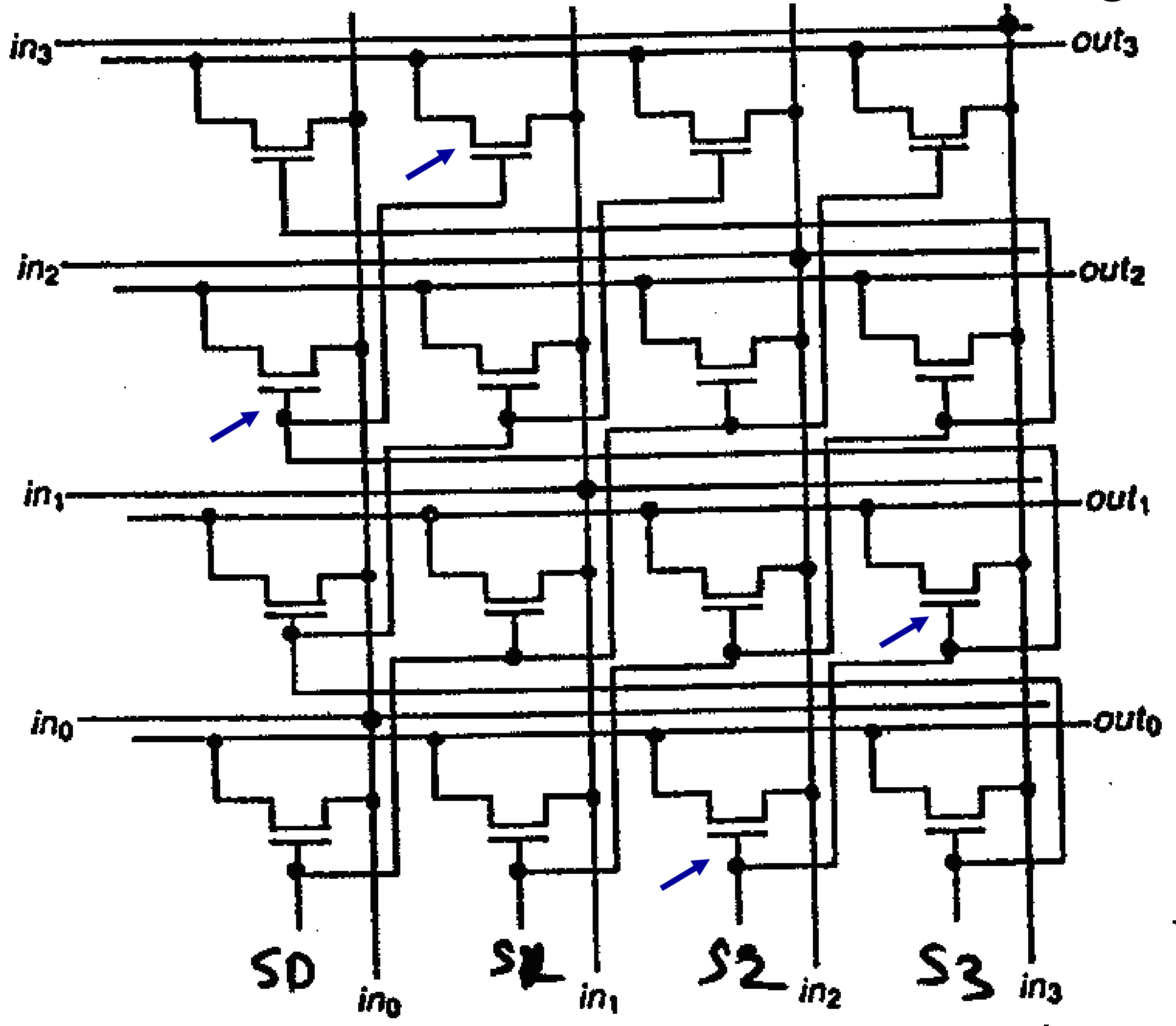
- Design Computational Elements.
- The focus is on Barrel shifter, Adder and simple Multiplier design.

Barrel Shifter Design

- As division and multiplication of integers represented as binary numbers by powers of 2 leads to left or right shift operations, barrel shifter forms one important computational element.
- Example: 14/2=7. 14=1110,7=0111 and operation is equivalent to cyclic right shift.
- Uses transmission gates.
- Barrel Shifter should perform any shift

For example, in 4 bit case, (i) 4 input lines, (ii) 4 lines for 4 output bits, (iii) Gate inputs for selecting any of 0, 1, 2, 3 bit shift operations are needed. In the barrel shifter arrangement, a single high poly line can achieve the shifting operation.

Barrel Shifter Design



ure 8-7 4 X 4 barrel shifter

- Setting S1, S2, S3 achieves 1, 2, 3 bit shift, respectively.
- Set S2 to logic 1. This turns on the MOSFETs marked by arrows and shiftsin2->out0, in3->out1, in0->out2, in1->out3 giving cyclic shift.

Adder Design

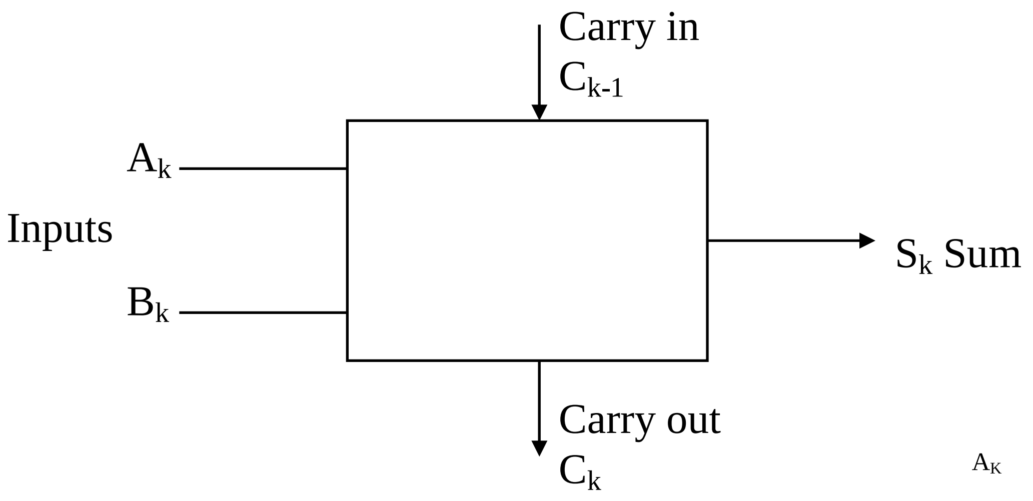
● An adder forms a vital part any computer system. The truth table for a 1 bit full adder is as shown.

● Both input bits of 1 and no input carry give 0 sum and 1 carry outputs as binary number can only be o or 1 and sum 2 overflows to next bit position.

Inputs			Outputs	
A_K	B_K	Previous carry C_{k-1}	S_K	C_K
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Adder Design

● Schematically, the adder element is as shown as a block diagram.



Inputs			Outputs	
A_K	B_K	Previous carry C_{k-1}	S_K	C_K
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

The full adder functionality can be logically described as

If $A_k = B_k$ then $S_k = C_{k-1}$ Else $S_k = \overline{C_{k-1}}$.

If $A_k = B_k$ then $C_k = A_k = B_k$ Else $C_k = C_{k-1}$.

Adder Design

- How can we implement this function?

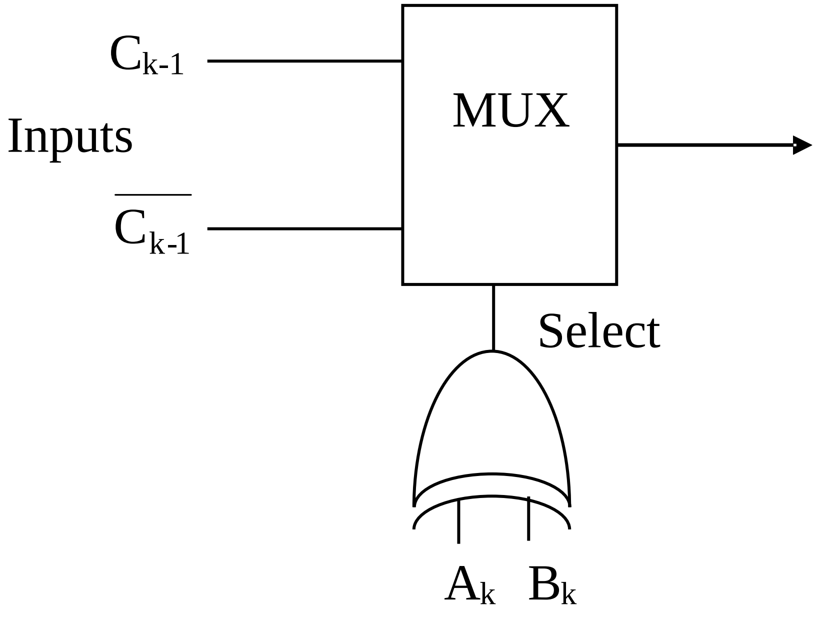
We have to make a decision if $A_k = B_k$. This can be done using a 2-input XOR gate where the output is zero when two inputs are equal.

Using this as control signal for a 2-input multiplexer, correctly selected logic variable is placed at the output.

This leads to the following implementations for the SUM and CARRY outputs.

Adder Design

- SUM Implementation:

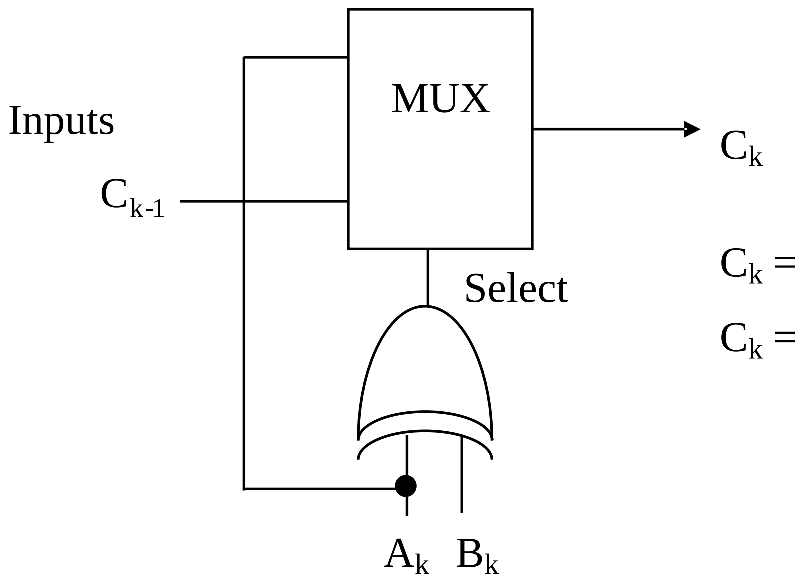


$S_k = C_{k-1}$ if select = 0

else $S_k = \overline{C_{k-1}}$

Adder Design

- CARRY OUT implementation:



$C_k = A_k$ if select = 0

$C_k = C_{k-1}$ if select = 1

- Both these implementations need an inverter, an XOR which possibly can be shared and one independent Multiplexer cells.

Adder Design

● The 1-bit full adder cells can be cascaded to implement adders involving more bits.

● If each adder cell has a delay of τ, then this type of cascaded ripple carry n-bit adder has a total summation time of $n\tau$ if τ is carry bit delay. For the adder cell implementation above, τ will be a sum of the XOR and 2 to 1 multiplexer delay. Hence more efficient cell implementations are sought.

Adder Design

A set of equivalent Boolean Expressions for S_k and C_k of one bit adder, which can be more efficiently implemented, are

$C_k = (A_k + B_k)C_{k-1} + A_kB_k$

$S_k = \overline{C_k}(A_k + B_k + C_{k-1}) + A_kB_kC_{k-1}$

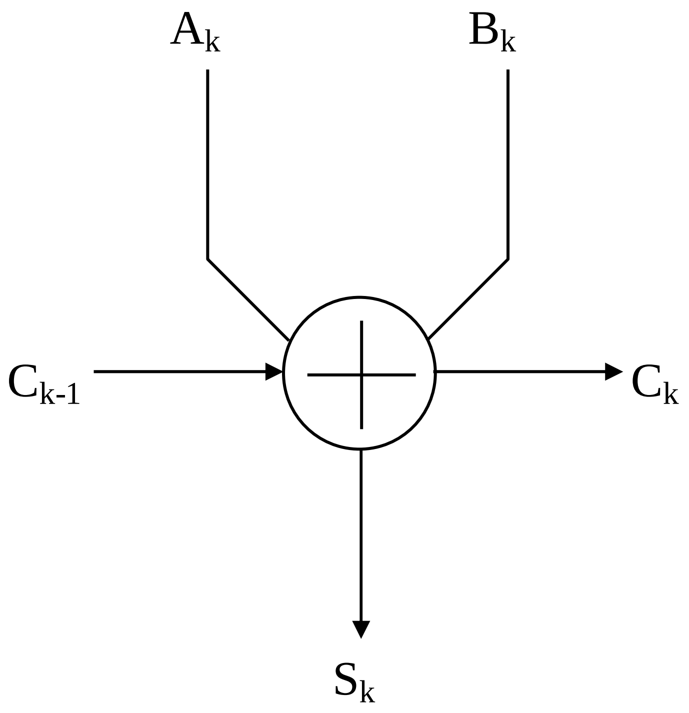
$= A_k \oplus B_k \oplus C_{k-1}.$

Obviously, it is easy to implement $\overline{C_k}$ and $\overline{S_k}$.

● As carry propagation delay determines overall delay of the adder, this implementation will be faster. In the previous circuit, XOR was used to implement CARRY OUT. Here just a 3 input NOR is sufficient for output CARRY implementation.

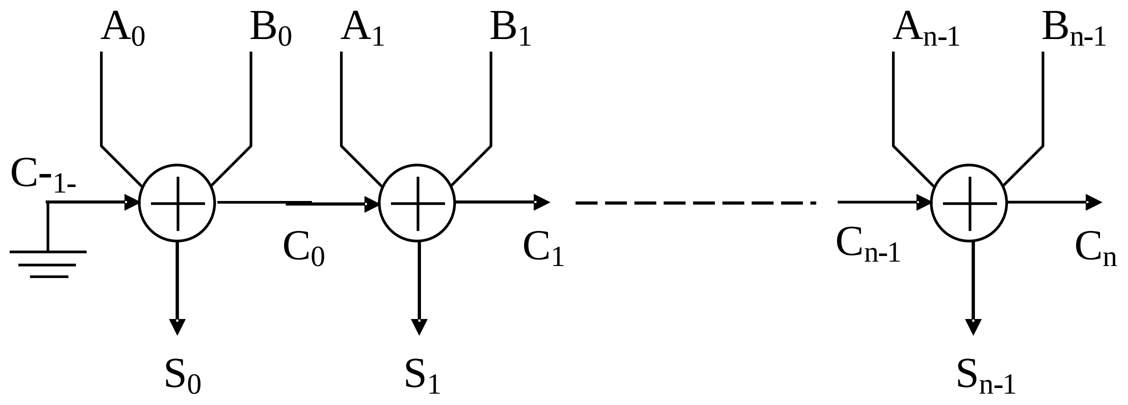
Adder Design

● A simple and compact adder cell representation without reference to details is often used to explain architecture and is given below.



Adder Design

● Hence cascaded ripple carry implementation of n-bit adder is as shown. Carry bits propagate sideways. Sum bits are available once carry is known.

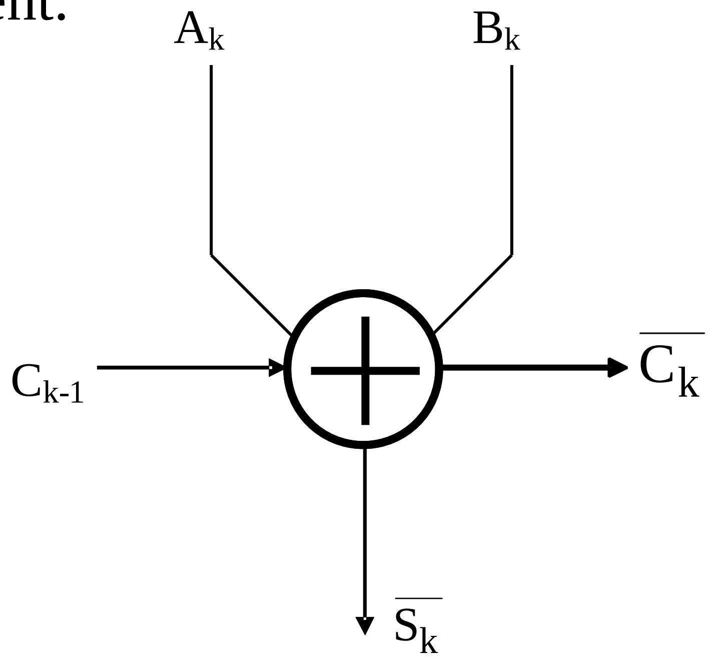


● This implementation is still not that efficient as it does not use an inverted carry and requires an inverter for implementing the OR gate for carry.

Adder Design

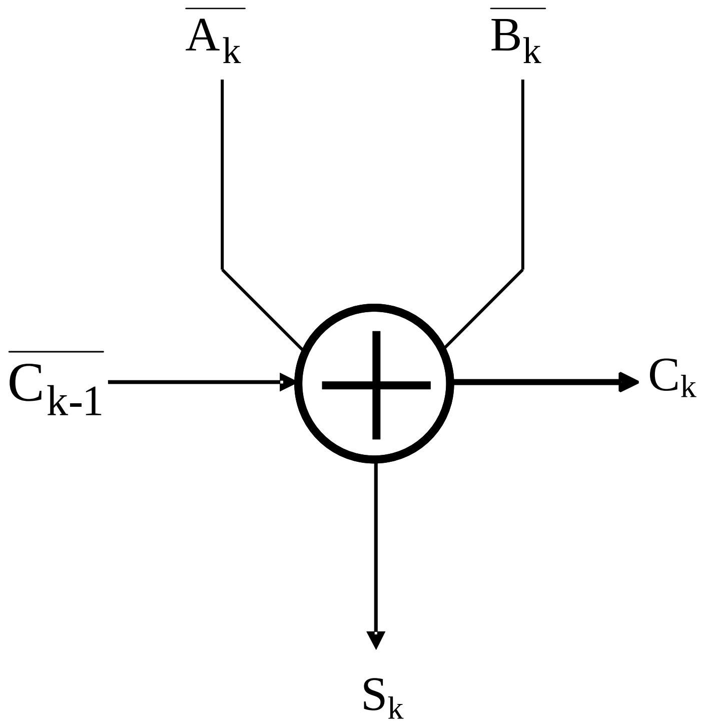
- To remove this inverter delay, a new “bold” adder cell below actually outputs inverted carry out and inverted if normal input bits and carry appear at its input.

Clearly, bold adder cell below has $\overline{C_k}$ and $\overline{S_k}$ as its output when its input has actual bits to be added and carry as these are much faster to implement.



Adder Design

● Obviously, this new cell actually outputs normal carry out and sum if inverted input bits and inverted carry appear at its input as shown.



Adder Design

● An efficient adder implementation using the two versions of bold cell is possible as shown below.

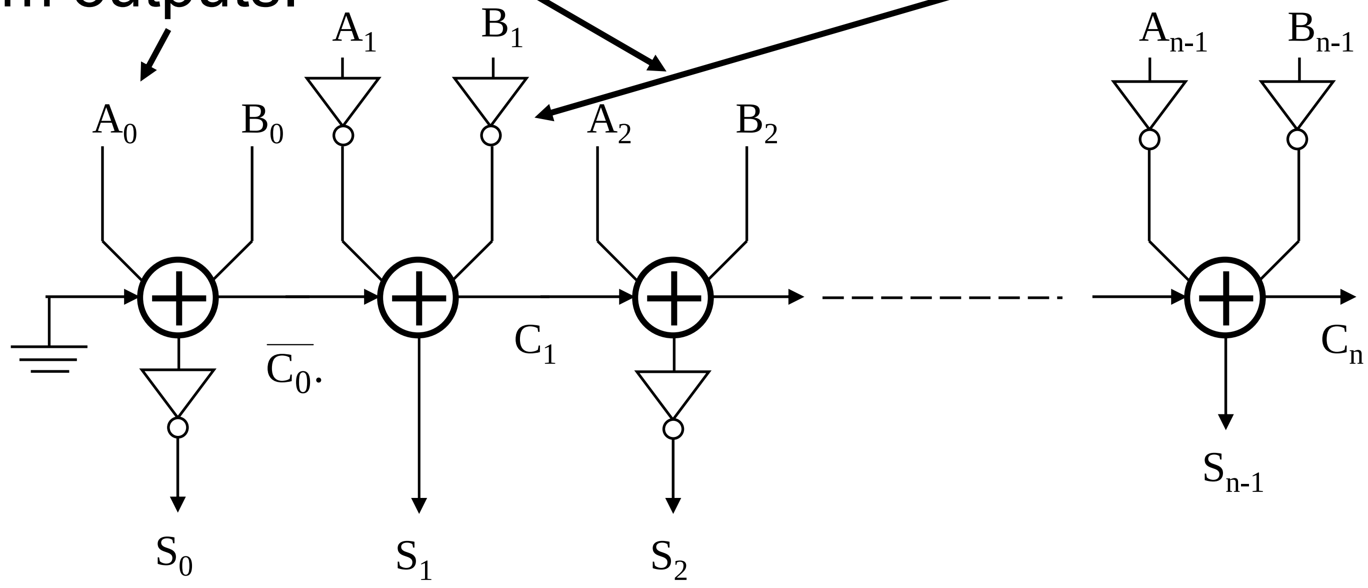
For this efficient implementation, carry propagation delay must be reduced. Hence every even cell must be able to directly use $\overline{C_k}$ as input and there must be no need to invert $\overline{C_k}$.

Clearly, every odd cell must be able to directly use C_k .

Adder Design

● Odd cells-Normal input-inverted carry and sum outputs.

● Even cells-Inverted input-normal carry/sum outputs.



● As input bits to be added are available at the same time, they are inverted for all even cells.

● For second and other even cells, S_1, S_3, S_5, S_7, ….are directly available as all inputs are inverted.

● The output with such inverted bits as input will directly yield S_k and C_k (needed for odd cells) in place of $\overline{C_k}$ and $\overline{S_k}$. You can verify this Bullion equivalence.

Adder Design

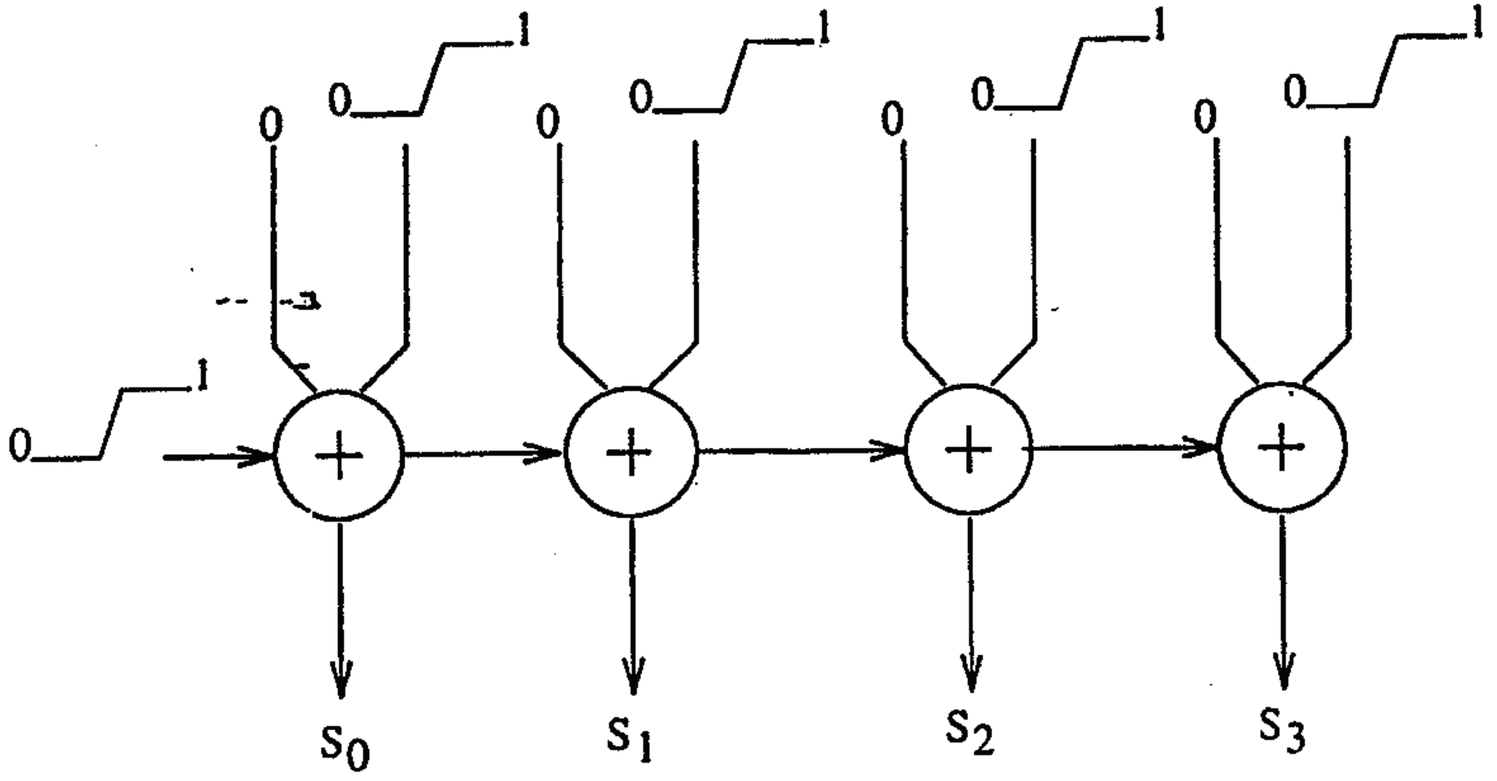
● Since, for the first and every odd cell original bits and carry are the inputs, the output of these cells is $\overline{C_k}$ and $\overline{S_k}$. Hence the sum output of the cell needs to be inverted to get S_k. Thus this is done for every odd cell.

● This cascading balances delays and avoids using inverter for each carry. $\overline{C_0}$ delay allows time for inversion of A_1, B_1, … etc.

● Such ripple carry adders may suffer from glitch phenomenon depicted in the following figure. Hence careful optimization is needed so that this does not pose a serious problem.

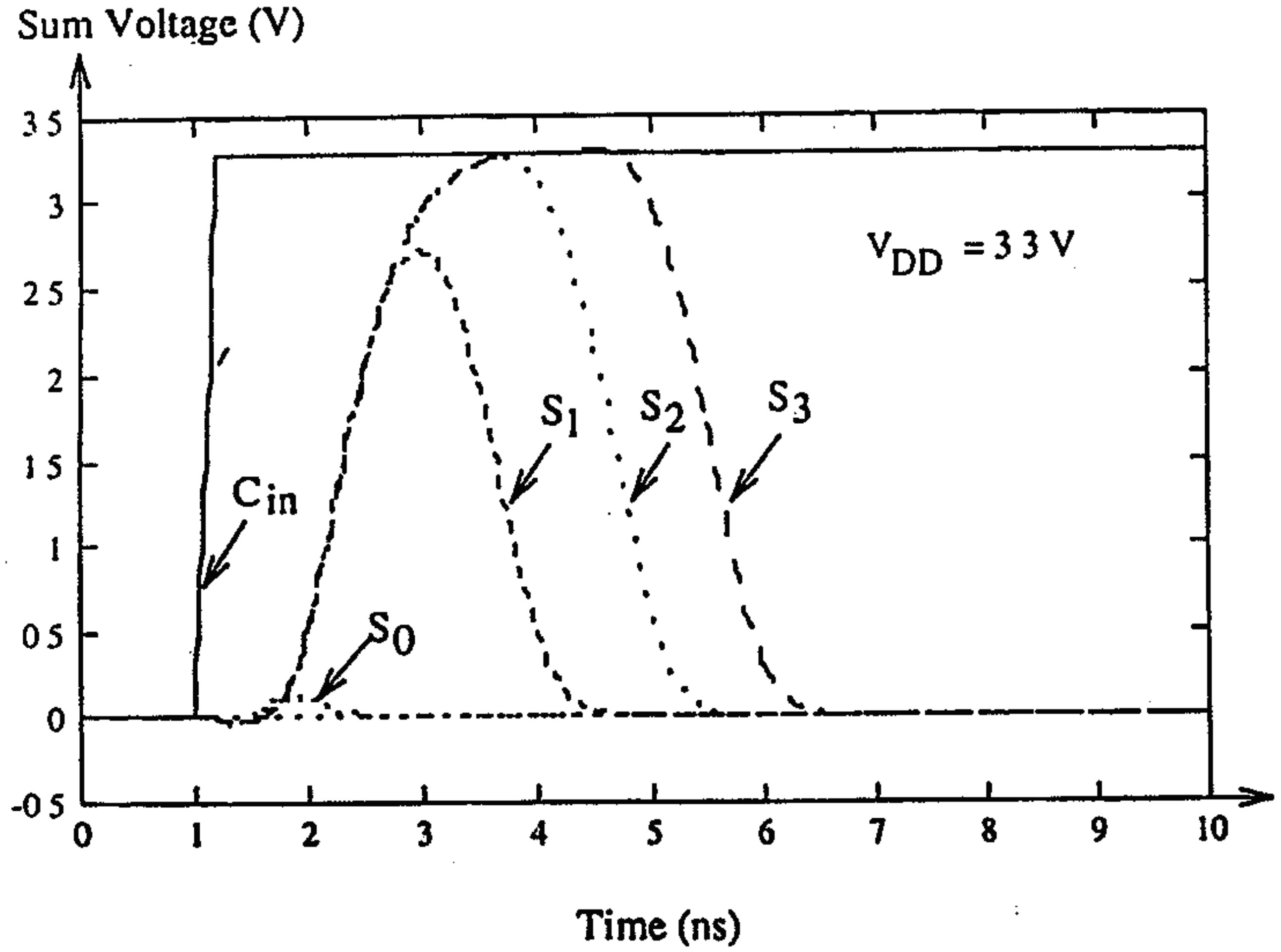
Adder Design

● Let us assume all logic states are initially 0. The sum of this adder is expected be zero and carry 1 in the steady state.



Adder Design

● However, due to carry propagation delay, sum bits may momentarily rise. This gives spurious glitches and unnecessary power consumption.



Adder Design

- Even more efficient implementations of adders are needed for the speed requirements today. Some of these implementations now will be discussed.

CLA Adder Implementation

● In carry look ahead (CLA) adder, carry for all the bit positions is predicted in advance so that the sum of each bit position can be carried out independently.

● The adder actually uses intermediate or auxiliary inputs known as propagate (P) and generate (G) signals defined by the logic functions below which do not need carry bits in their implementation.

$\overline{P_i} = \overline{A_i \oplus B_i}$ $\quad \oplus$ indicates XOR

$\overline{G_i} = \overline{A_i\ B_i}$

CLA Adder Implementation

● the carry and sum is predicted for any bit position with the help of P_i's and G_i's. The carry expressions are:

$$C_1 = G_1 + P_1\,C_0$$

$$= \overline{\overline{G_1} \bullet (\overline{C_0} + \overline{P_1})}$$

$$C_2 = \overline{\overline{G_2} \bullet (\overline{P_2} + \overline{G_1} \bullet (\overline{C_0} + \overline{P_1}))} = \overline{\overline{G_2} \bullet (\overline{P_2} + \overline{C_1})}$$

$$C_3 = \overline{\overline{G_3} \bullet (\overline{P_3} + \overline{G_2} \bullet (\overline{P_2} + \overline{G_1} \bullet (\overline{C_0} + \overline{P_1})))}$$

$$C_4 = \overline{\overline{G_4} \bullet (\overline{P_4} + \overline{G_3} \bullet (\overline{P_3} + \overline{G_2} \bullet (\overline{P_2} + \overline{G_1} \bullet (\overline{C_0} + \overline{P_1}))))}$$

and so on.

And S_i = Sum at bit position i = $\overline{\overline{P_i} \oplus C_{i\text{-}1}}$

CLA Adder Implementation

● As is clear, to predict carry for more than 4 bit positions needs a very large overhead and 5 series connected devices. Hence, the CLA unit is limited to 4 bits and 2 or more of these are used to do the addition.

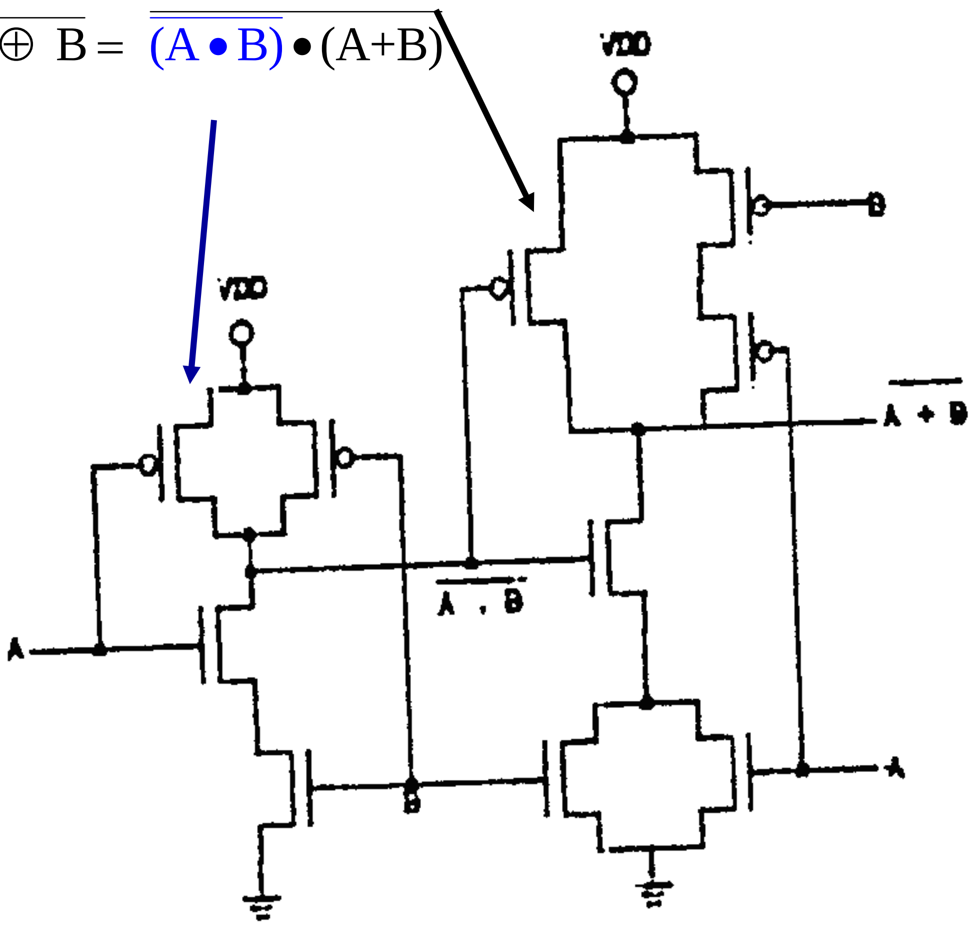
● Since C_4 is predicted very fast, next carry bits are also available.

● The schematics for the carry positions and XNOR are shown next along with the layout of XNOR and CLA unit.

CLA Adder Implementation

- XNOR schematic:

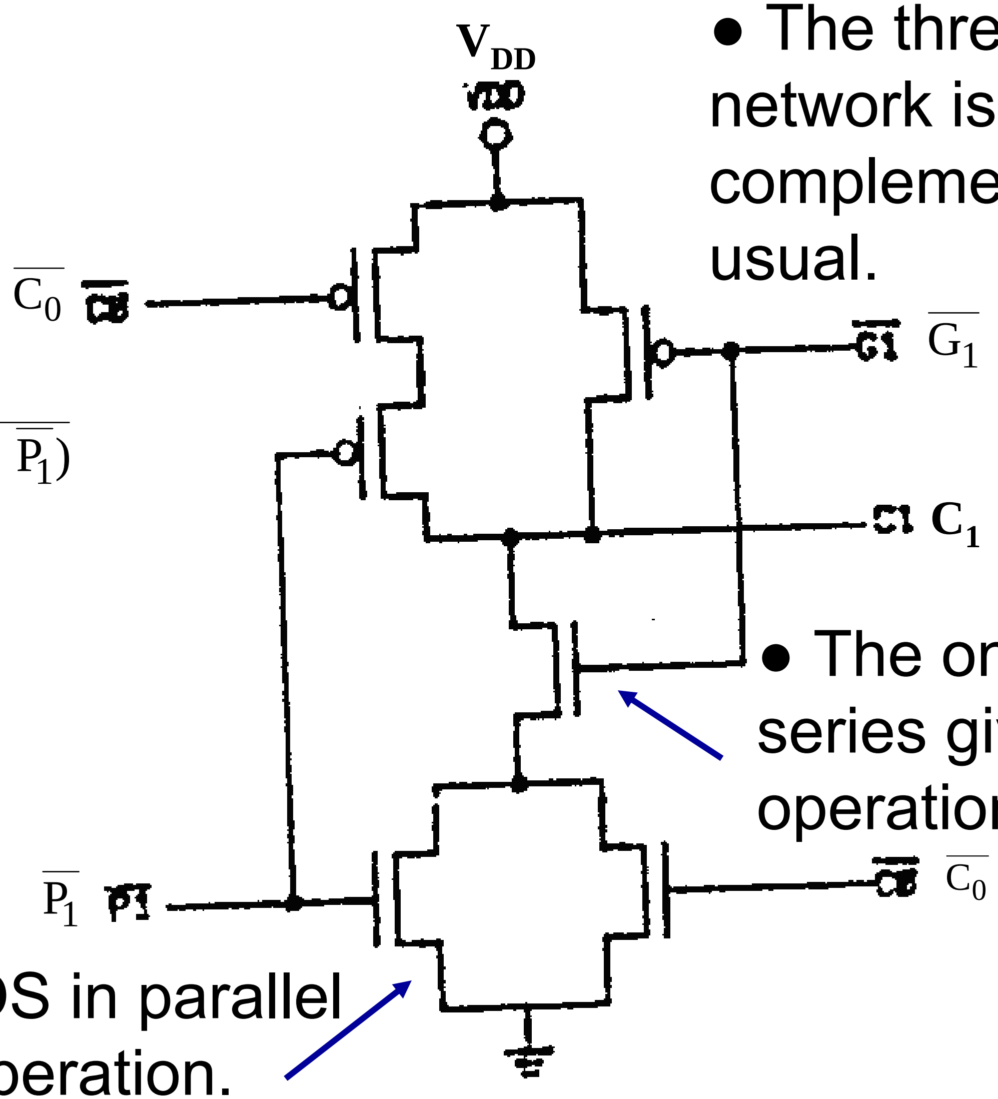
$$\overline{A \oplus B} = \overline{\overline{(A \bullet B)} \bullet (A+B)}$$



CLA Adder Implementation

- C_1 schematic:

$C_1 = \overline{\overline{G_1} \bullet (\overline{C_0} + \overline{P_1})}$

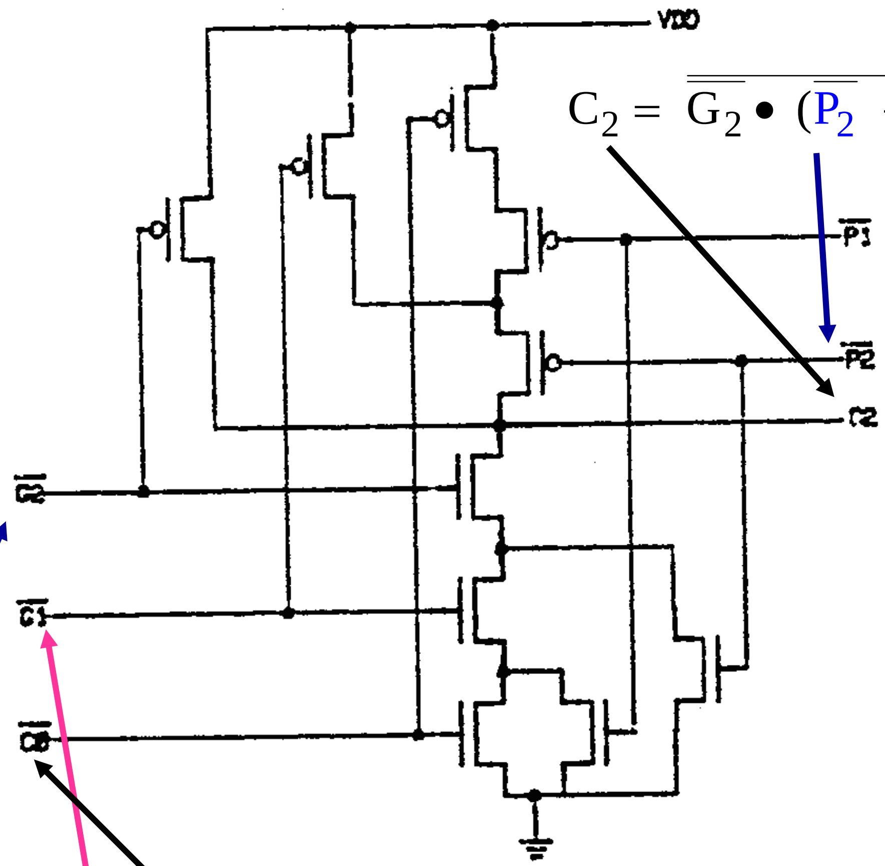


- The three p-MOS network is complementary as usual.
- The one n-MOS in series gives NAND operation with the NOR.
- Two n-MOS in parallel give NOR operation.

CLA Adder Implementation

- C_2 schematic:

$$C_2 = \overline{\overline{G_2} \bullet (\overline{P_2} + \overline{G_1} \bullet (\overline{C_0} + \overline{P_1}))}$$

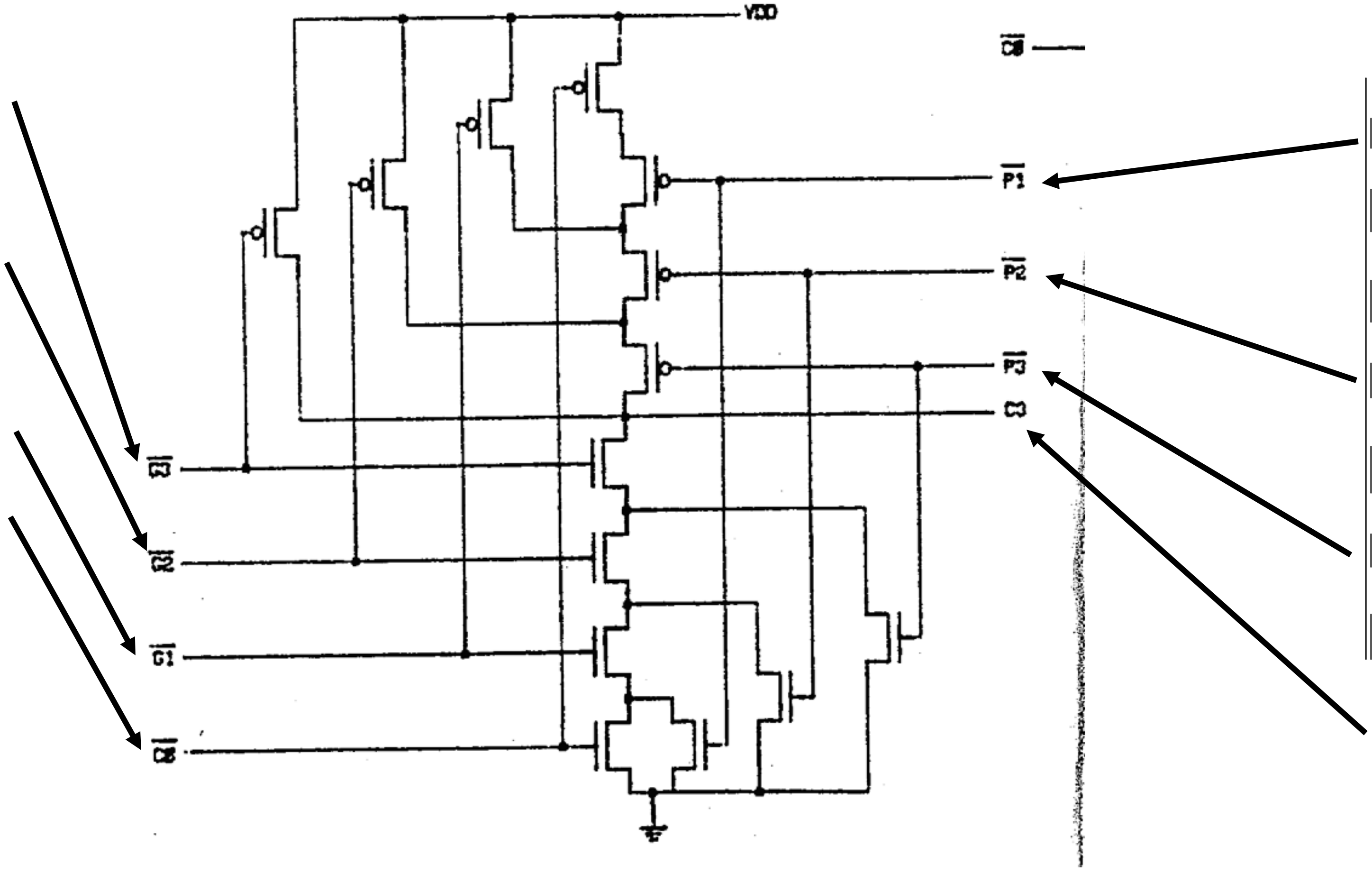


$$C_2 = \overline{\overline{G_2} \bullet (\overline{P_2} + \overline{G_1} \bullet (\overline{C_0} + \overline{P_1}))}$$

CLA Adder Implementation

- C_3 schematic: $C_3 = \overline{\overline{G_3} \bullet (\overline{P_3} + \overline{G_2} \bullet (\overline{P_2} + \overline{G_1} \bullet (\overline{C_0} + \overline{P_1})))}$

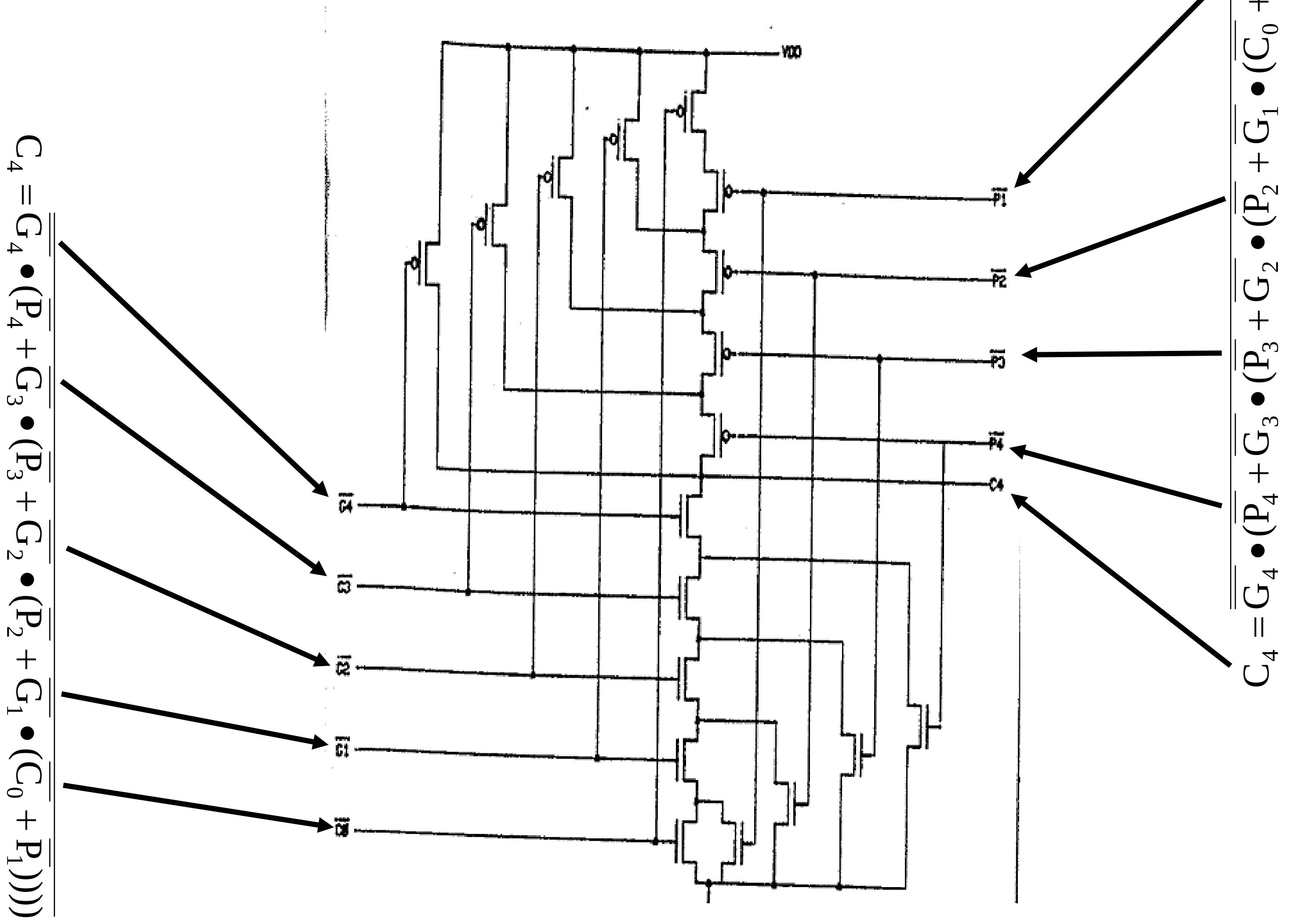
$$C_3 = \overline{\overline{G_3} \bullet (\overline{P_3} + \overline{G_2} \bullet (\overline{P_2} + \overline{G_1} \bullet (\overline{C_0} + \overline{P_1})))}$$



$$C_3 = \overline{\overline{G_3} \bullet (\overline{P_3} + \overline{G_2} \bullet (\overline{P_2} + \overline{G_1} \bullet (\overline{C_0} + \overline{P_1})))}$$

CLA Adder Implementation

- C_4 schematic: $C_4 = \overline{\overline{G_4} \bullet (\overline{P_4} + \overline{G_3} \bullet (\overline{P_3} + \overline{G_2} \bullet (\overline{P_2} + \overline{G_1} \bullet (\overline{C_0} + \overline{P_1}))))}$



$C_4 = \overline{\overline{G_4} \bullet (\overline{P_4} + \overline{G_3} \bullet (\overline{P_3} + \overline{G_2} \bullet (\overline{P_2} + \overline{G_1} \bullet (\overline{C_0} + \overline{P_1}))))}$
VDD
$\overline{P1}$
$\overline{P2}$
$\overline{P3}$
$\overline{P4}$
C4
$\overline{G4}$
$\overline{G3}$
$\overline{G2}$
$\overline{G1}$
$\overline{C0}$
$C_4 = \overline{\overline{G_4} \bullet (\overline{P_4} + \overline{G_3} \bullet (\overline{P_3} + \overline{G_2} \bullet (\overline{P_2} + \overline{G_1} \bullet (\overline{C_0} + \overline{P_1}))))}$

CLA Adder Implementation

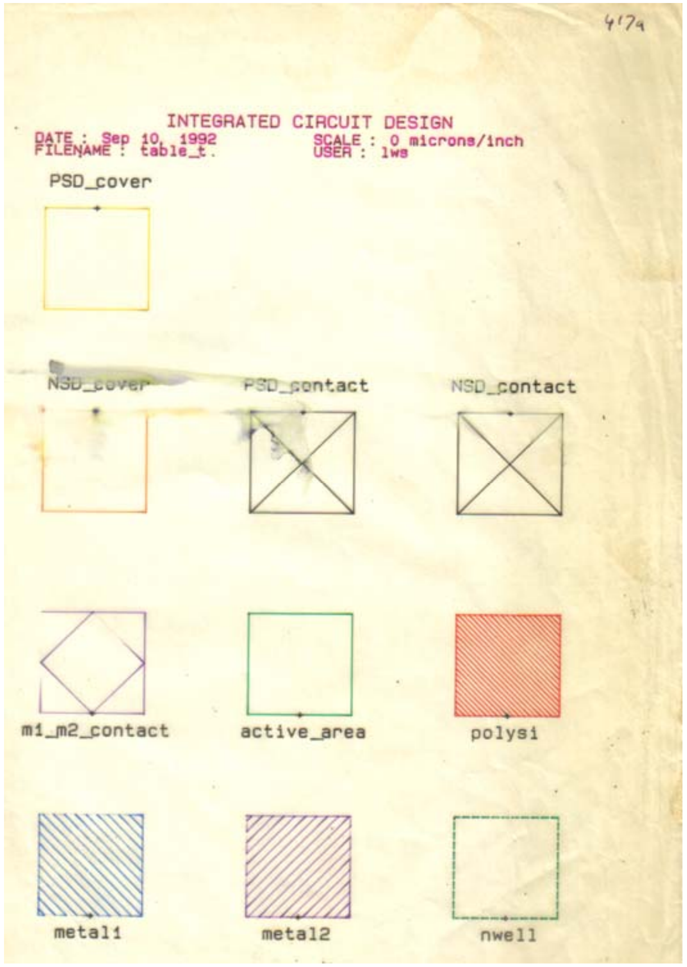
● As is clear to predict carry for more than 4 bit positions needs a very large overhead and 5 series connected devices. Hence, the CLA unit is limited to 4 bits and 2 or more of these are used to do the addition.

● Since C_4 is predicted very fast, next carry bits are also available.

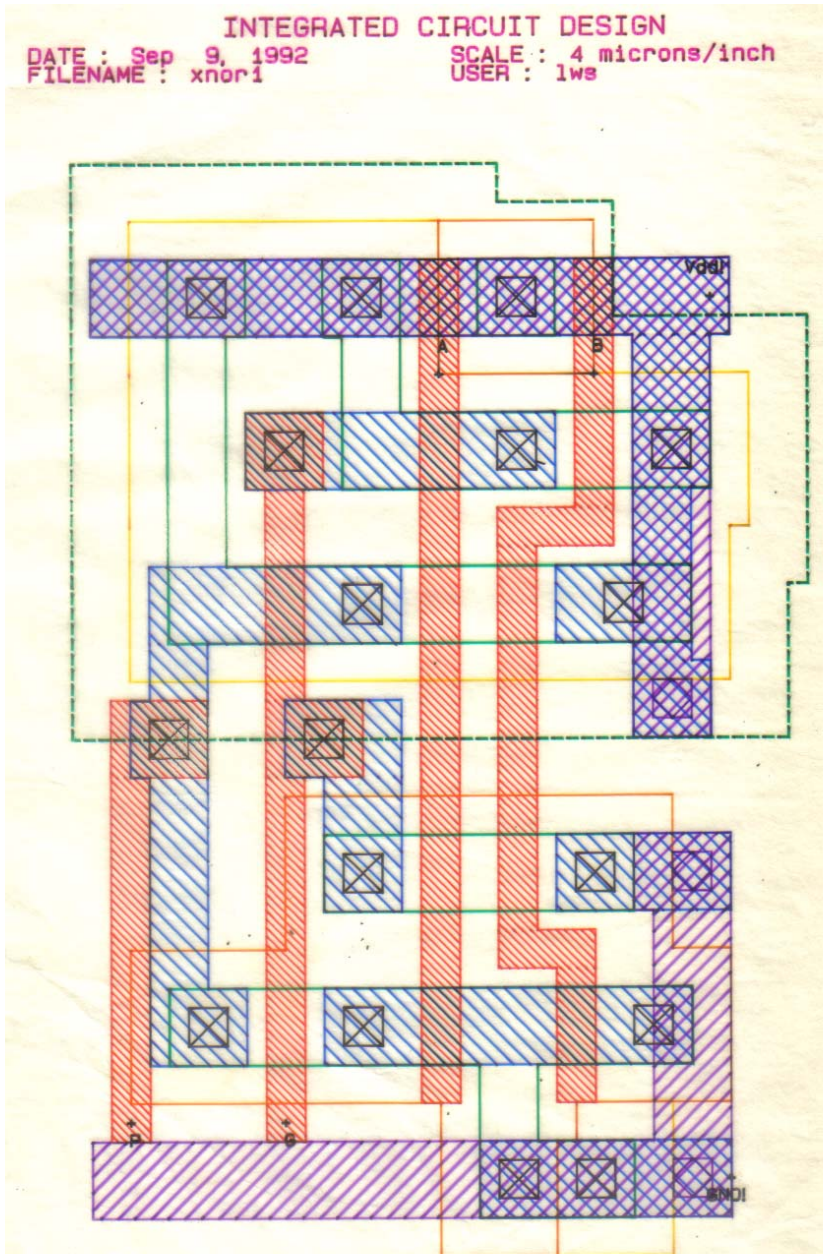
● The schematics for the carry positions and XNOR are shown next along with the layout of XNOR and CLA unit.

CLA Adder Implementation

- Layer convention in this is some what different and is given. Basic colors are same except p-MOS are covered by PSD cover and n-MOS by NSD cover over common green area.

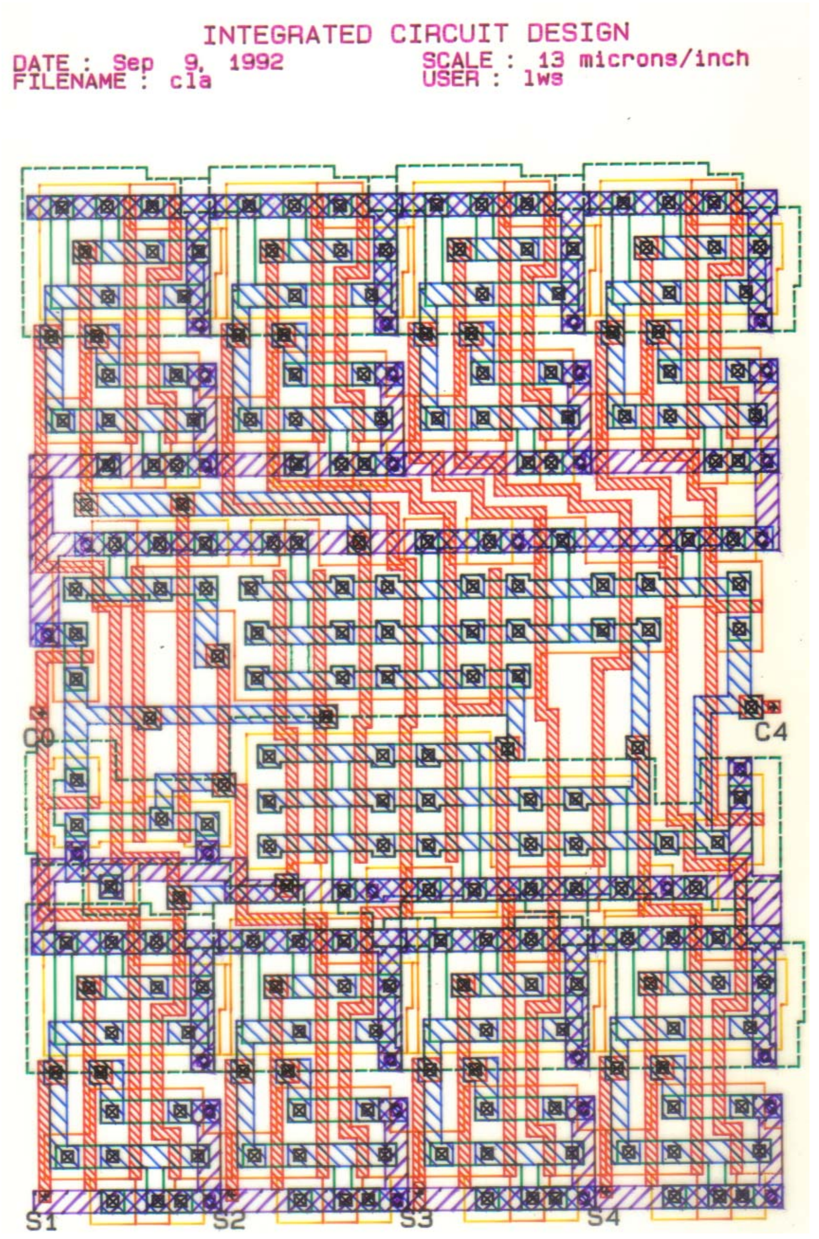


CLA Adder Implementation



- XNOR layout-You can easily figure out how it implements P and G.

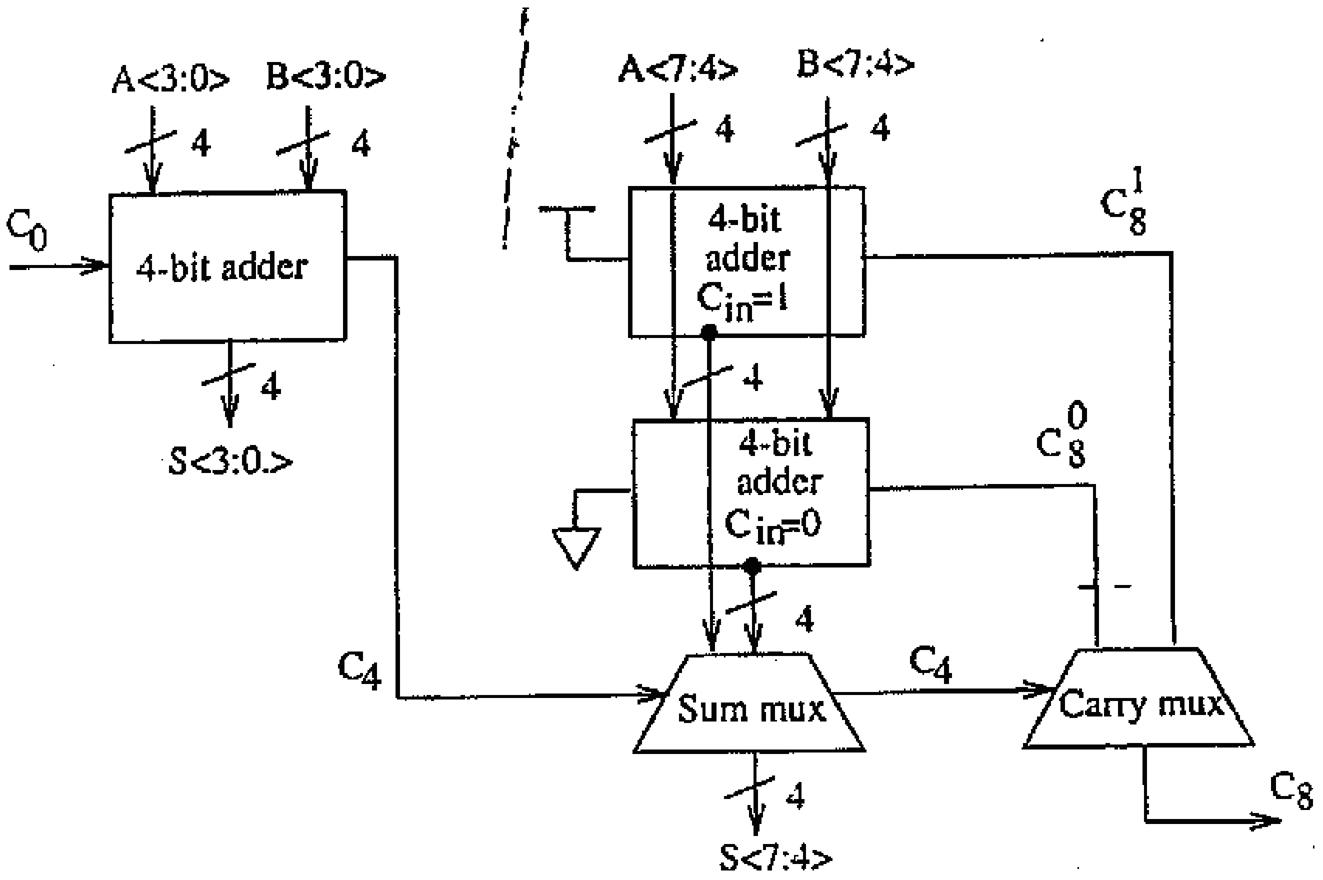
CLA Adder Implementation



- 4-bit CLA cell layout.
- 4 XNOR gates that give needed P and G logic.
- Fast C_4 OUTPUT that can connect to next 4bit stage.
- 4 XNOR gates that give needed sum bits.

CSA Adder Implementation

- Carry Select Adder (CSA) block diagram and operation.



- Sums are calculated assuming C_{in} = 0 and C_{in} = 1. The sum bits are transferred to the output in one shot once carry is known using the multiplexer.

CSA Adder Implementation

● Carry Select Adder (CSA) block size and timing

● If k_1 is one bit adder delay then for n bits, simple RCA delay = $k_1 n$. If m parallel paths (blocks) are used and k_2 is carry multiplexer delay, the total CSA delay is given by

$$\text{CSA delay} = k_1 \frac{n}{m} + k_2 (m - 1)$$

● Optimization needed to get number of blocks m that achieves the best timing.

● CLA is the fastest at high power and area. RCA is slow but is very regular and has low power. CSA gives compromise.

Simple unsigned Braun multiplier

● Consider two n-bit unsigned binary integers and their product as shown below.

$$X = \sum_{i=0}^{n-1} X_i \, 2^i \qquad\qquad Y = \sum_{j=0}^{n-1} Y_j \, 2^j$$

$$\therefore XY = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} X_i Y_j \, 2^{i+j}$$

● Product binary number has 2n bits.

● $X_i\, Y_i$ terms can be calculated using AND gates and hence these gates will not be explicitly shown.

Simple unsigned Braun multiplier

$$XY = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} X_i Y_j \, 2^{i+j}$$

● For a given value of (i + j), there are many i, j combinations, which have the same (i + j) value, that contribute to the (i + j)th i.e. $2^{(i+j)}$ bit. For example, for 2^4 bit position, X_0Y_4, X_1Y_3, X_2Y_2, X_3Y_1, X_4Y_0 contribute and have to be added.

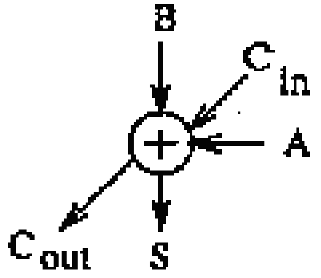
● Adder cells are required to do these additions.

Simple unsigned Braun multiplier

● Let us now look at binary number multiplication process.

$$
\begin{array}{cccccccccl}
 & & & & X_3 & X_2 & X_1 & X_0 & & = X \\
 & & & & Y_3 & Y_2 & Y_1 & Y_0 & & = Y \\
\hline
 & & & & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 & & \\
 & & & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 & & & \\
 & & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 & & & & \\
 & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 & & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 & & = P = X*Y
\end{array}
$$

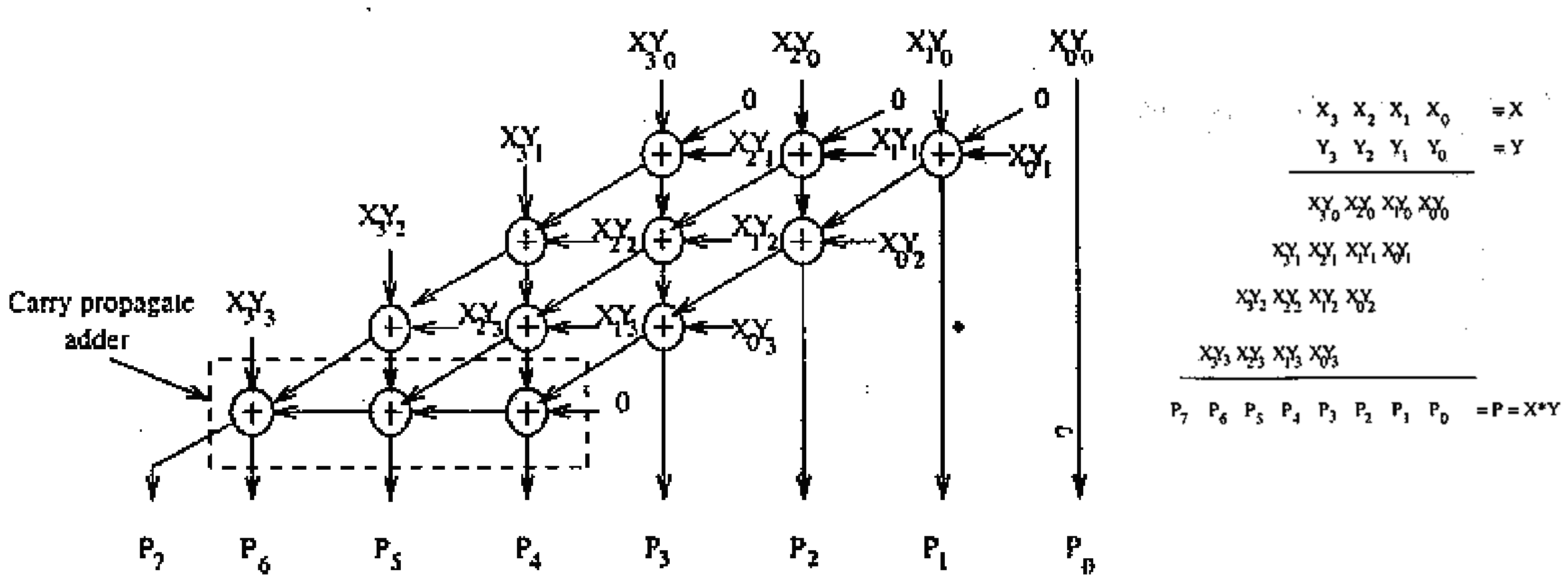
● In P_1 calculation if there is carry while adding X_0Y_1 and X_1Y_0, it has to propagate sideways to P_2. Hence all carries need to propagate sideways as additions for product terms are performed.



● This leads to an adder cell representation with an ease to propagate carry sideways as shown.

Simple unsigned Braun multiplier

● Using this cell, multiplier is implemented as shown.


X_3Y_0 X_2Y_0 X_1Y_0 X_0Y_0
X_3Y_1 X_2Y_1 X_1Y_1 X_0Y_1
X_3Y_2 X_2Y_2 X_1Y_2 X_0Y_2
X_3Y_3 X_2Y_3 X_1Y_3 X_0Y_3
0
Carry propagate adder
P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0
X_3 X_2 X_1 X_0 $=X$
Y_3 Y_2 Y_1 Y_0 $=Y$
P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0 $=P=X*Y$

● For n bits, n(n-1) adders and n^2 AND gates are required in this implementation. For 4 bits, 12 adders and 16 ANDs are needed.

● Completes out discussion on computational elements.