MEGHE GROUP
VALUES REDEFINED

DTEL(Department for Technology Enhanced Learning)

The Centre for Technology enabled Teaching & Learning , N Y S S, India

Teaching Innovation - Entrepreneurial - Global

MEGHE GROUP
VALUES REDEFINED

DEPARTMENT OF COMPUTER TECHNOLOGY

IV-SEMESTER

COMPUTER ARCHITECTURE AND ORGANIZATION

UNIT NO.4
ARITHMETICS

UNIT 4:- SYLLABUS

1. Number Representation , Addition of Positive numbers
2. Logic Design for fast adders
3. Addition and Subtraction , Arithmetic and Branching conditions
4. Multiplications of positive numbers, Signed-Operand multiplication
5. Fast Multiplication, Booth's Algorithm
6. Integer Division, Floating point numbers and operations

UNIT-4 SPECIFIC OBJECTIVE / COURSE OUTCOME

The student will be able to:

1 **Design** of Arithmetic unit to perform fixed point and floating point arithmetic operations.

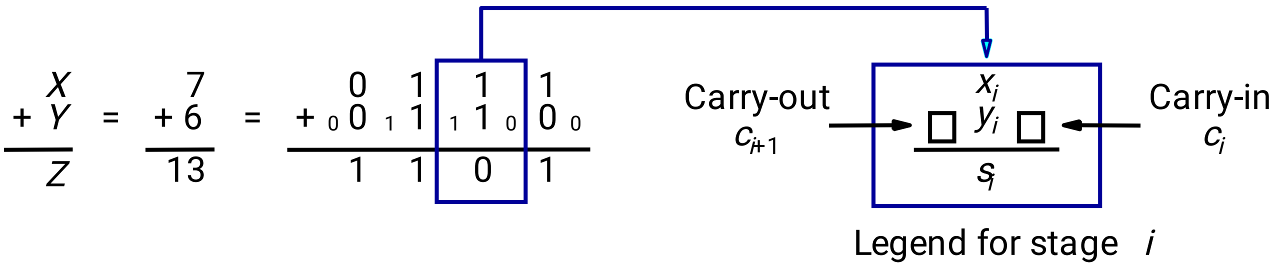
Add of signed& unsigned nos

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
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

$$s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:



Legend for stage *i*

At the i^{th} stage:

Input:

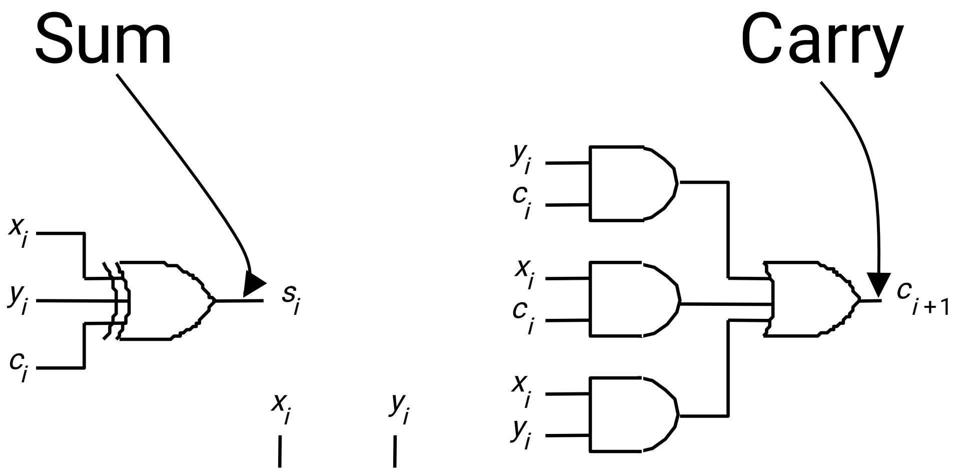
c_i is the carry-in

Output:

s_i is the sum

c_{i+1} carry-out to $(i+1)^{st}$ state

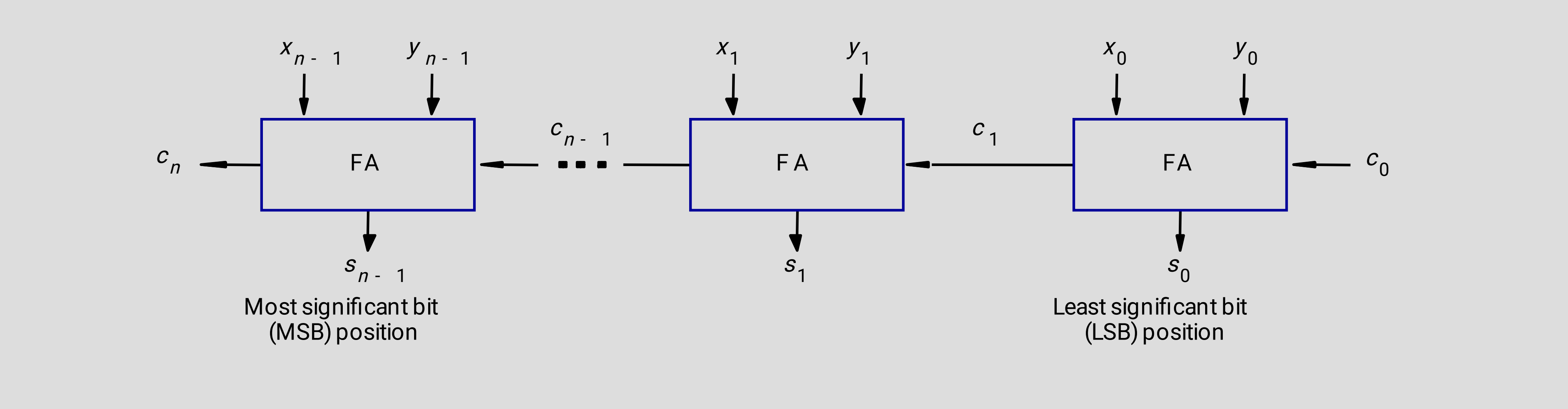
Add of logic for a single stage



Full Adder (FA): Symbol for the complete circuit for a single stage of addition.

n- bit Adder

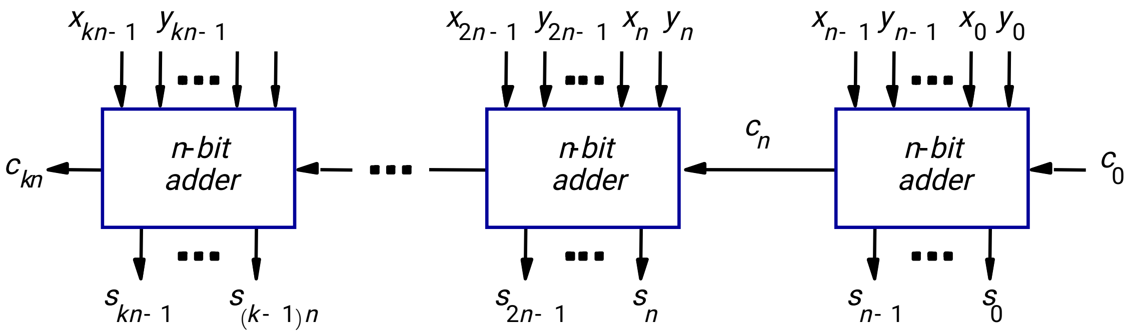
- Cascade *n* full adder (FA) blocks to form a *n*-bit adder.
- Carries propagate or ripple through this cascade, *n*-bit ripple carry adder.



Carry-in c_0 into the LSB position provides a convenient way to perform subtraction.

K n-bit Adder

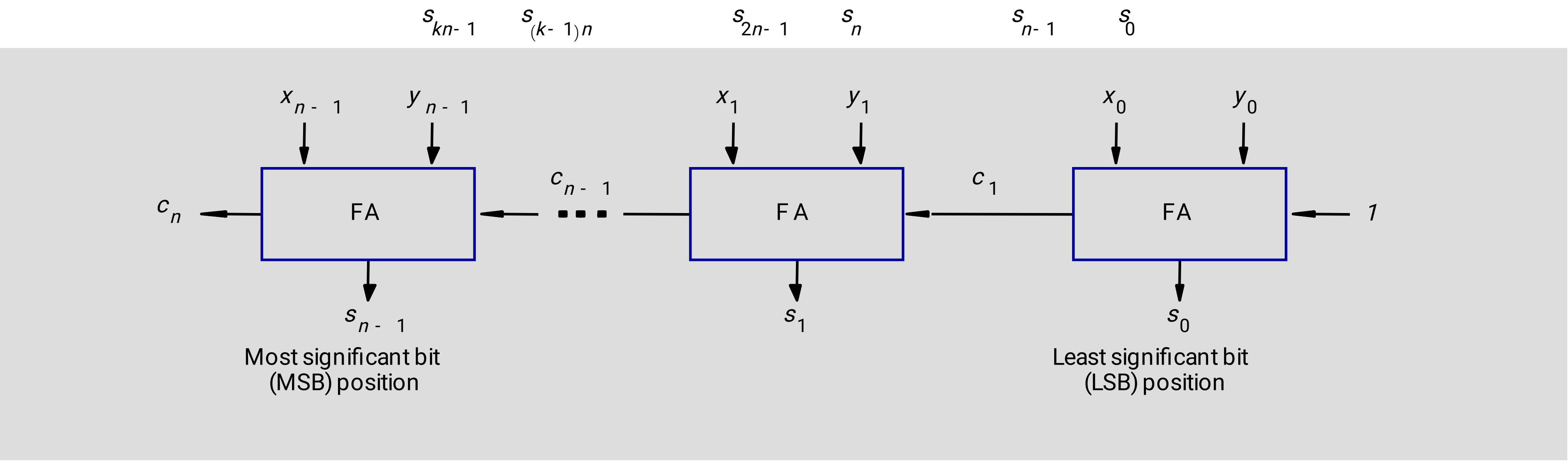
K n-bit numbers can be added by cascading *k n*-bit adders.



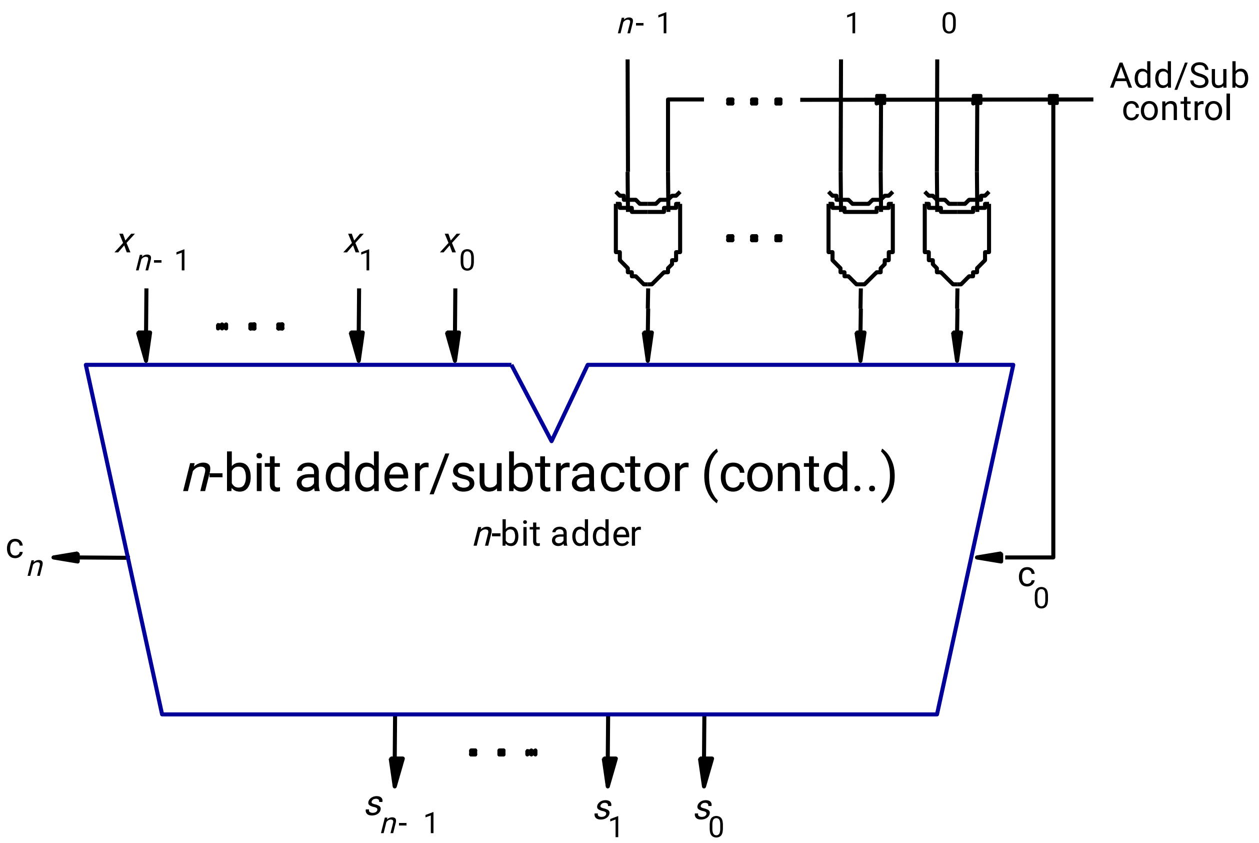
Each *n*-bit adder forms a block, so this is cascading of blocks Carries ripple or propagate through blocks, Blocked Ripple Carry Adder

n-bit Subtractor

- Recall $X - Y$ is equivalent to adding 2's complement of Y to X.
- 2's complement is equivalent to 1's complement + 1.
- $X - Y = X + Y + 1$
- 2's complement of positive and negative numbers is computed similarly.



n-bit adder/subtractor (contd)



- Add/sub control = 0, addition.
- Add/sub control = 1, subtraction.

Detecting Overflows

- Overflows can only occur when the sign of the two operands is the same.

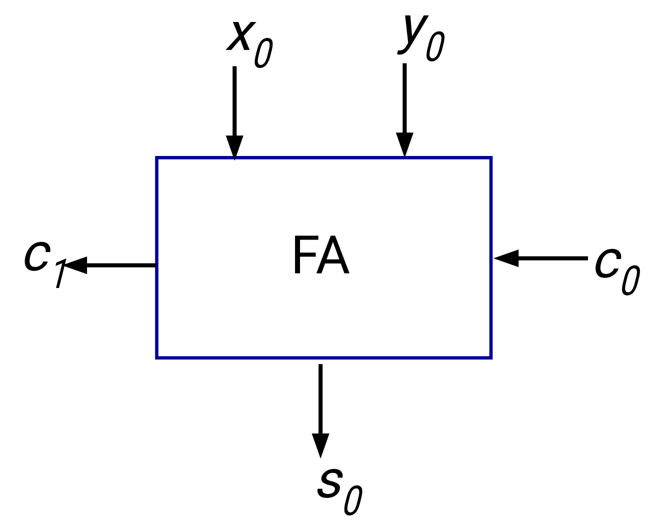
- Overflow occurs if the sign of the result is different from the sign of the operands.

- Recall that the MSB represents the sign.
  - x_{n-1}, y_{n-1}, s_{n-1} represent the sign of operand *x*, operand *y* and result *s* respectively

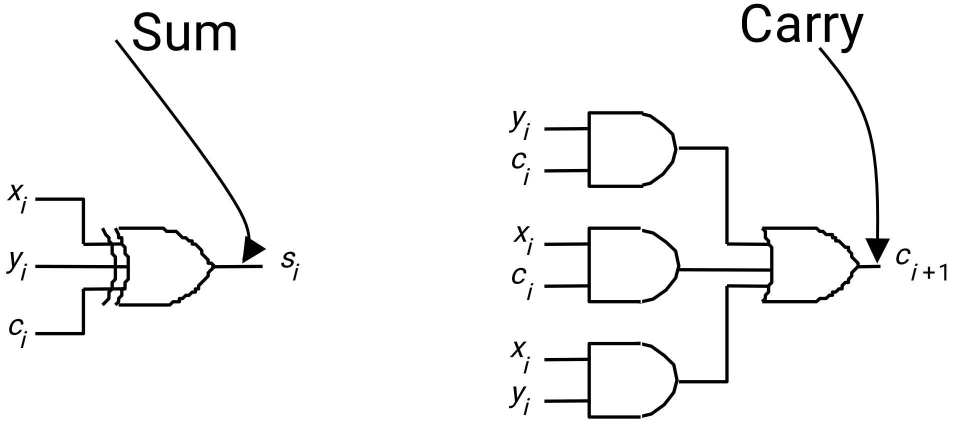
- Circuit to detect overflow. can be implemented by the following logic expressions:

Computing the Add time

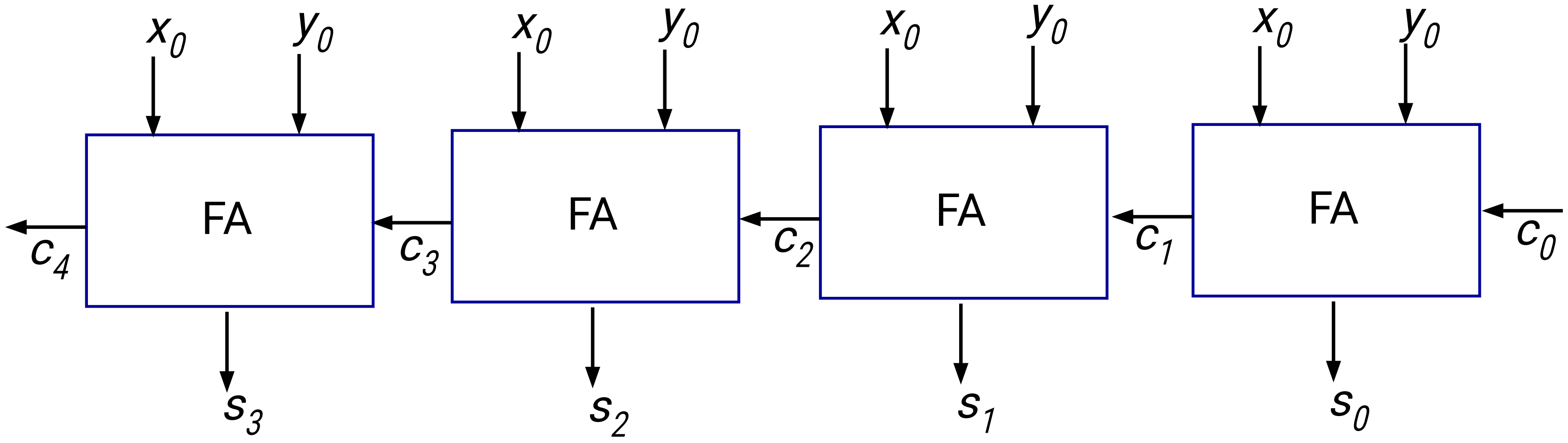
Consider *0th* stage:



- c_1 is available after 2 gate delays.
- s_1 is available after 1 gate delay.



Cascade of 4 Full Adders, or a 4-bit adder



- s_0 available after 1 gate delays, c_1 available after 2 gate delays.
- s_1 available after 3 gate delays, c_2 available after 4 gate delays.
- s_2 available after 5 gate delays, c_3 available after 6 gate delays.
- s_3 available after 7 gate delays, c_4 available after 8 gate delays.

For an *n*-bit adder, s_{n-1} is available after *2n-1* gate delays c_n is available after *2n* gate delays

LECTURE 1:- ARITHMETICS Fast Addition

Recall the equations:

$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Second equation can be written as: $c_{i+1} = x_i y_i + (x_i + y_i) c_i$

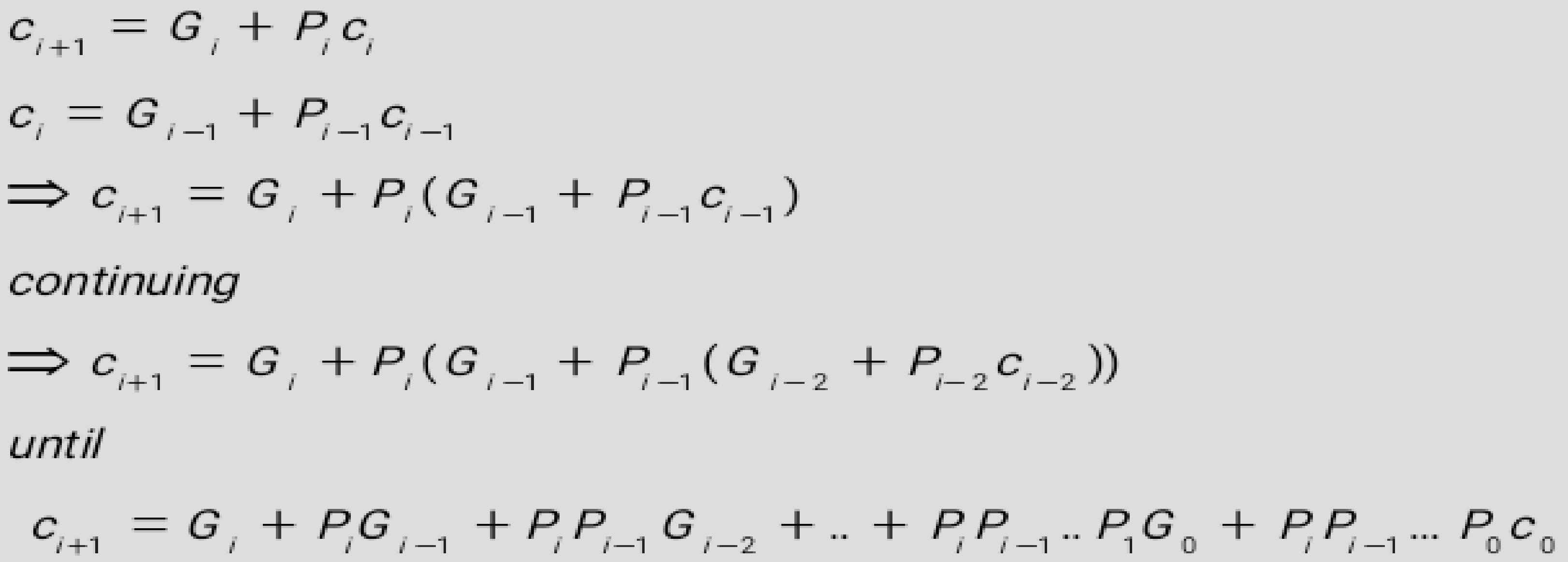
We can write:

$$c_{i+1} = G_i + P_i c_i$$

$$where \;\; G_i = x_i y_i \;\; and \;\; P_i = x_i + y_i$$

- G_i is called generate function and P_i is called propagate function
- G_i and P_i are computed only from x_i and y_i and not c_i, thus they can be computed in one gate delay after X and Y are applied to the inputs of an n-bit adder.

Carry Lookahead Adder



$$c_{i+1} = G_i + P_i c_i$$

$$c_i = G_{i-1} + P_{i-1} c_{i-1}$$

$$\Rightarrow c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1} c_{i-1})$$

continuing

$$\Rightarrow c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1}(G_{i-2} + P_{i-2} c_{i-2}))$$

until

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + .. + P_i P_{i-1} .. P_1 G_0 + P_i P_{i-1} ... P_0 c_0$$

•All carries can be obtained 3 gate delays after *X, Y* and c_0 are applied.

-One gate delay for P_i and G_i

-Two gate delays in the AND-OR circuit for c_{i+1}.

•This is called Carry Lookahead adder.

Carry Lookahead Adder

- Performing *n*-bit addition in 4 gate delays independent of *n* is good only theoretically because of fan-in constraints.

- Last AND gate and OR gate require a fan-in of (n+1) for a n-bit adder.

- In order to add operands longer than 4 bits, we can cascade 4-bit Carry-Lookahead adders. Cascade of Carry-Lookahead adders is called Blocked Carry-Lookahead adder.

Blocked carry lookahead Adder

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

Rewrite this as:

$$P_0^{I} = P_3 P_2 P_1 P_0$$

$$G_0^{I} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

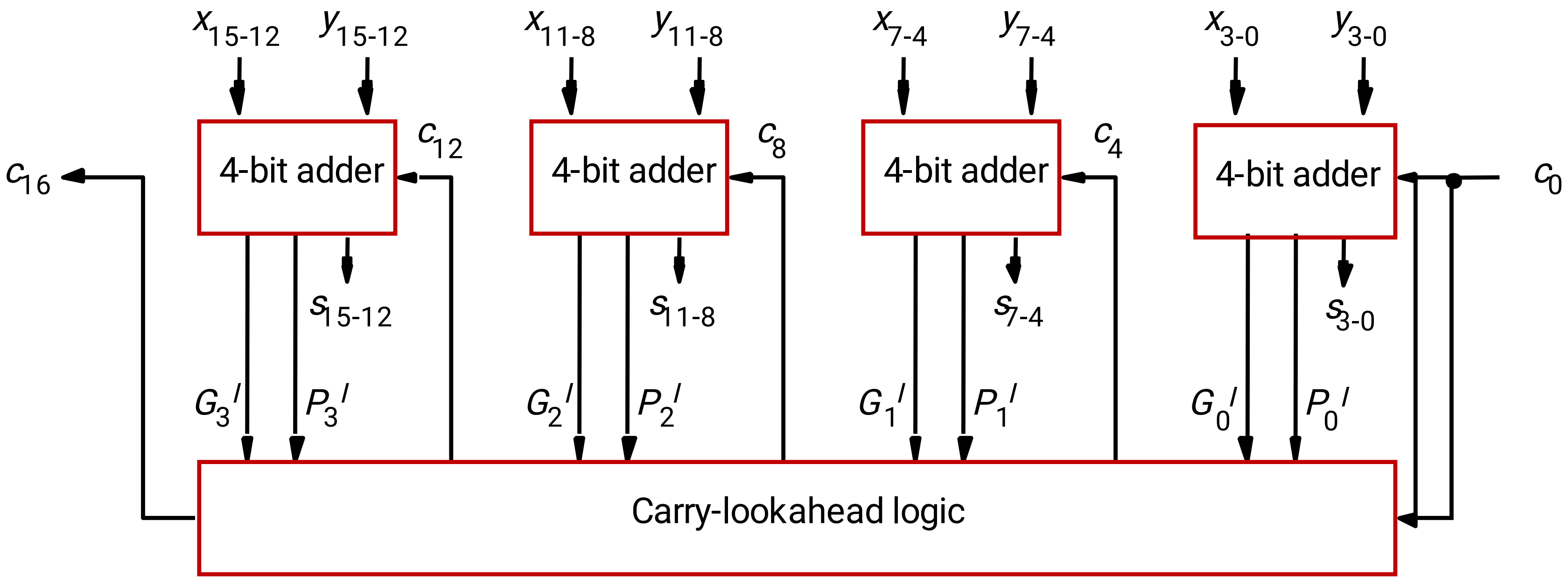
Subscript I denotes the blocked carry lookahead and identifies the block.

Cascade 4 *4*-bit adders, c_{16} can be expressed as:

$$c_{16} = G_3^{I} + P_3^{I} G_2^{I} + P_3^{I} P_2^{I} G_1^{I} + P_3^{I} P_2^{I} P_1^{0} G_0^{I} + P_3^{I} P_2^{I} P_1^{0} P_0^{0} c_0$$

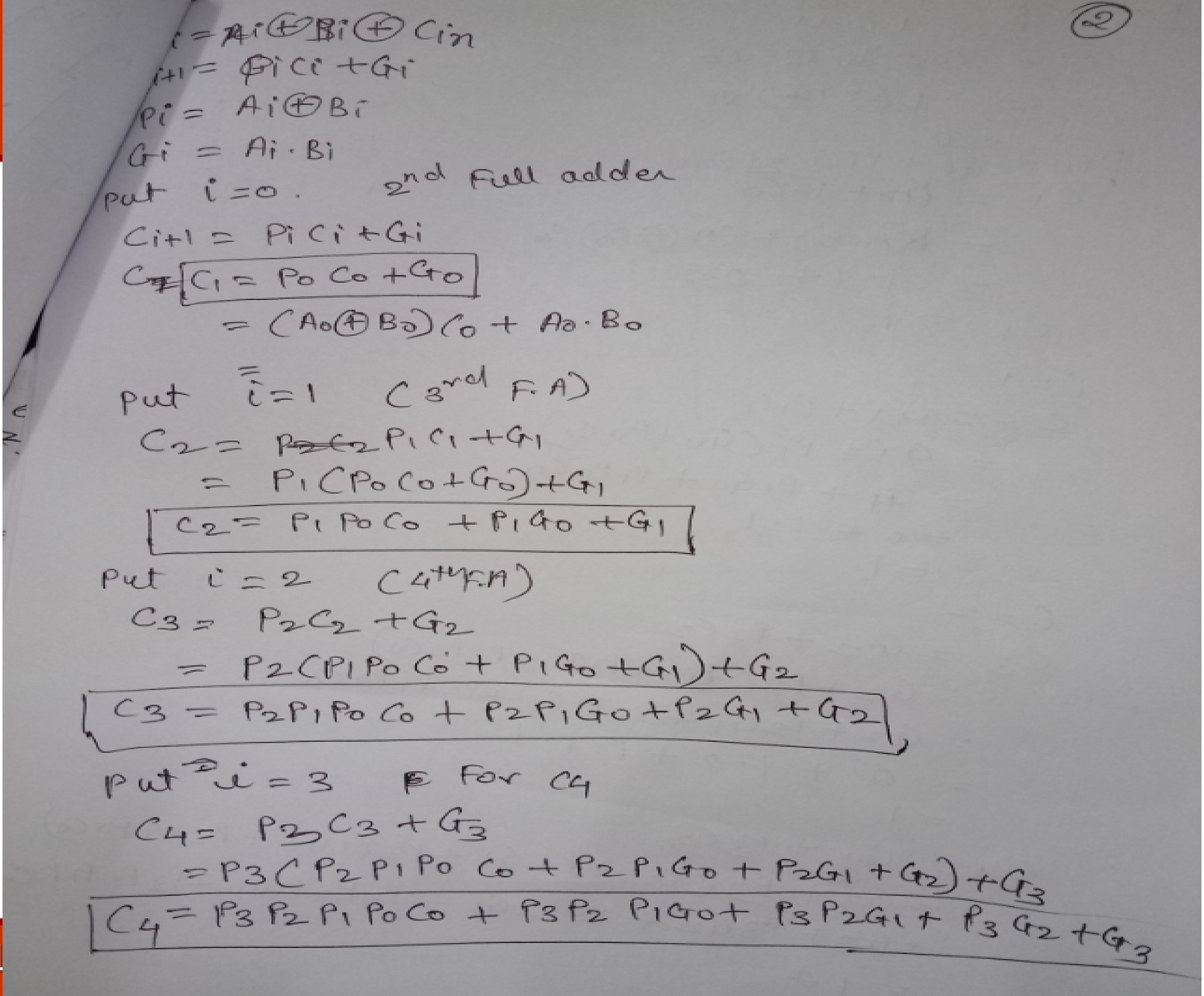
LECTURE 1:- ARITHMETICS

Blocked carry lookahead Adder



After x_i, y_i and c_0 are applied as inputs:

- G_i and P_i for each stage are available after 1 gate delay.
  - All carries are available after 5 gate delays.
- c_{16} is available after 5 gate delays.



2

$i = A_i \oplus B_i \oplus C_{in}$

$_{i+1} = P_i C_i + G_i$

$P_i = A_i \oplus B_i$

$G_i = A_i \cdot B_i$

put $i = 0$. 2nd Full adder

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

$\boxed{C_1 = P_0 C_0 + G_0}$

$= (A_0 \oplus B_0) C_0 + A_0 \cdot B_0$

put $i = 1$ (3rd F.A)

$C_2 = \cancel{P_2 C_2}\, P_1 C_1 + G_1$

$= P_1 (P_0 C_0 + G_0) + G_1$

$\boxed{C_2 = P_1 P_0 C_0 + P_1 G_0 + G_1}$

Put $i = 2$ (4th F.A)

$C_3 = P_2 C_2 + G_2$

$= P_2 (P_1 P_0 C_0 + P_1 G_0 + G_1) + G_2$

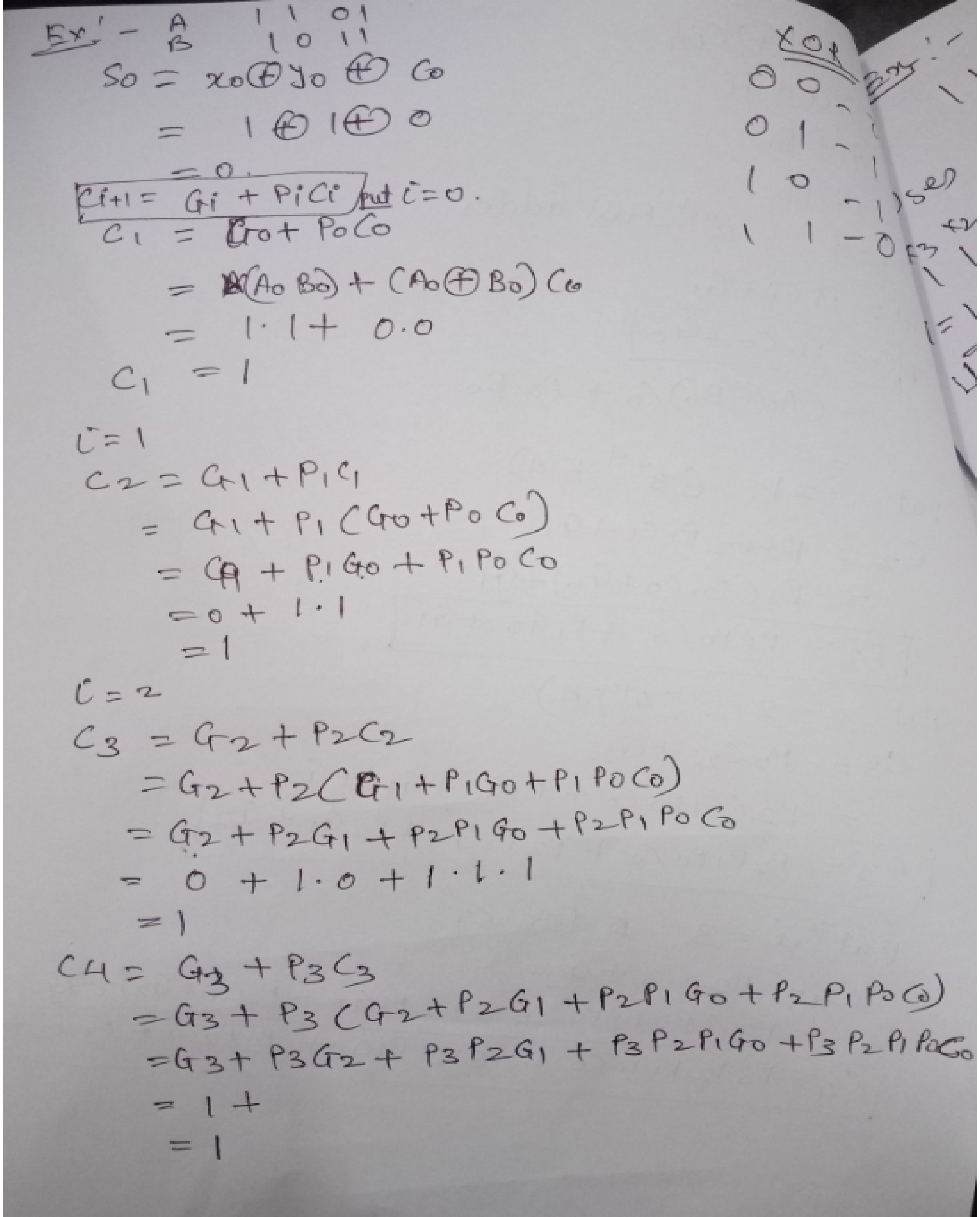
$\boxed{C_3 = P_2 P_1 P_0 C_0 + P_2 P_1 G_0 + P_2 G_1 + G_2}$

put $i = 3$ For C_4

$C_4 = P_3 C_3 + G_3$

$= P_3 (P_2 P_1 P_0 C_0 + P_2 P_1 G_0 + P_2 G_1 + G_2) + G_3$

$\boxed{C_4 = P_3 P_2 P_1 P_0 C_0 + P_3 P_2 P_1 G_0 + P_3 P_2 G_1 + P_3 G_2 + G_3}$



Ex! - A 1 1 0 1
B 1 0 1 1

$S_0 = x_0 \oplus y_0 \oplus C_0$

$= 1 \oplus 1 \oplus 0$

$= 0.$

$\boxed{C_{i+1} = G_i + P_iC_i}$ put $i = 0.$

$C_1 = G_0 + P_0C_0$

$= (A_0 B_0) + (A_0 \oplus B_0) C_0$

$= 1 \cdot 1 + 0 \cdot 0$

$C_1 = 1$

$i = 1$

$C_2 = G_1 + P_1C_1$

$= G_1 + P_1 (G_0 + P_0 C_0)$

$= G_1 + P_1 G_0 + P_1 P_0 C_0$

$= 0 + 1 \cdot 1$

$= 1$

$i = 2$

$C_3 = G_2 + P_2C_2$

$= G_2 + P_2 (G_1 + P_1G_0 + P_1 P_0 C_0)$

$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

$= 0 + 1 \cdot 0 + 1 \cdot 1 \cdot 1$

$= 1$

$C_4 = G_3 + P_3C_3$

$= G_3 + P_3 (G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0)$

$= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$

$= 1 +$

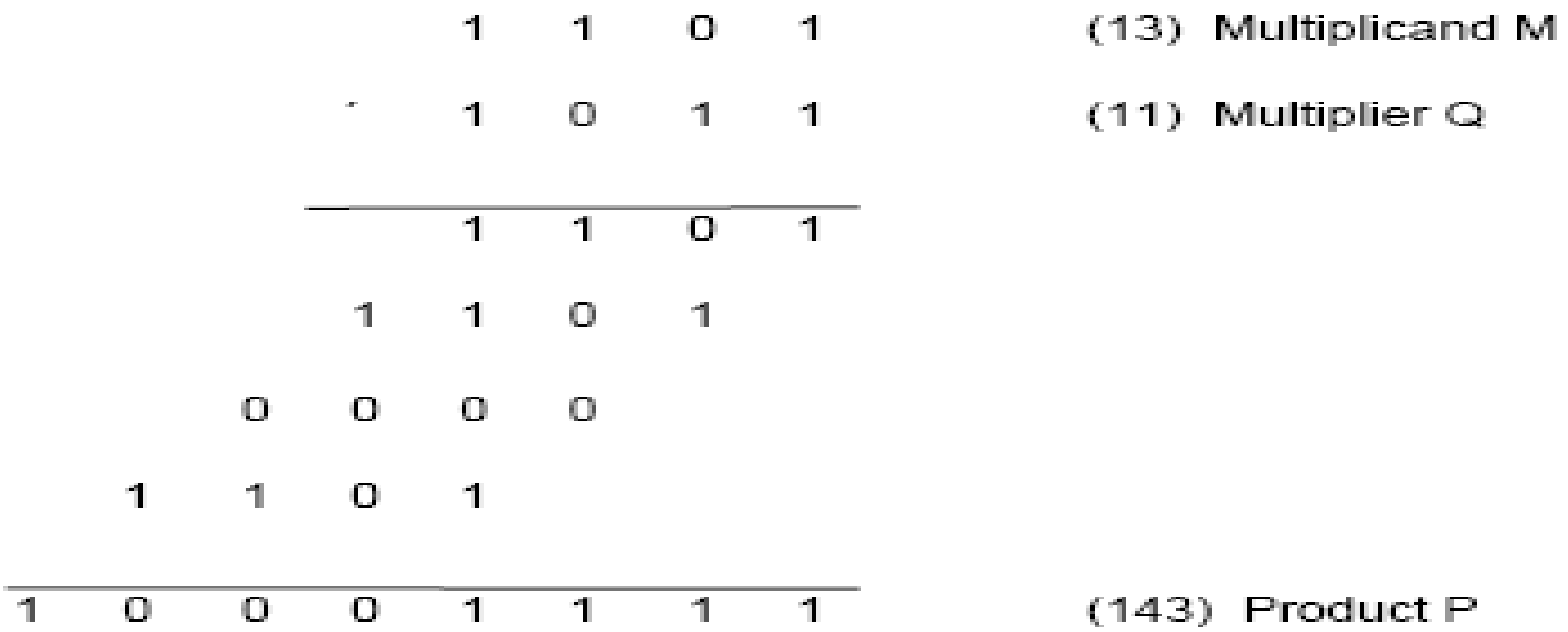
$= 1$

XOR

0	0	-	0
0	1	-	1
1	0	-	1
1	1	-	0

THANK YOU

Multiplication of unsigned numbers



```
                1   1   0   1        (13)  Multiplicand M
                1   0   1   1        (11)  Multiplier Q
            -----------------
                1   1   0   1
            1   1   0   1
        0   0   0   0
    1   1   0   1
-----------------------------
1   0   0   0   1   1   1   1        (143)  Product P
```

Product of 2 *n*-bit numbers is at most a *2n*-bit no.

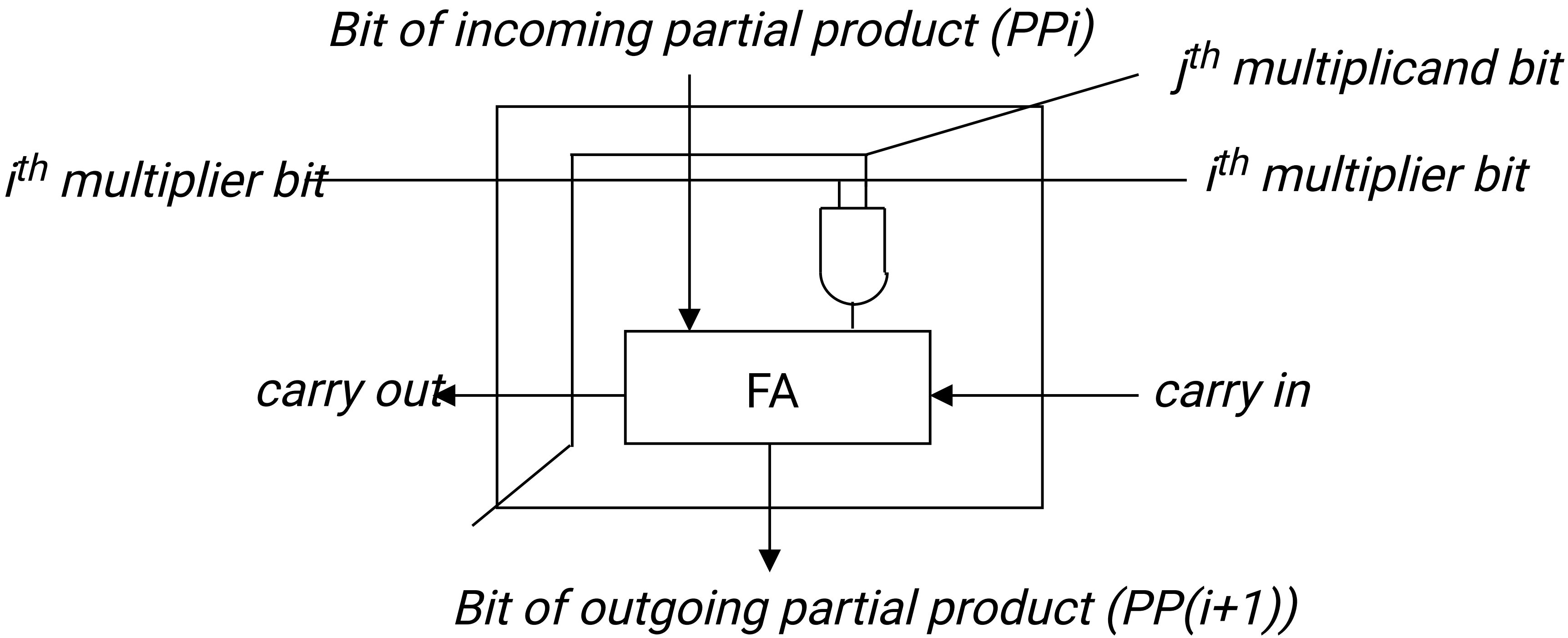
Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand.

Multiplication

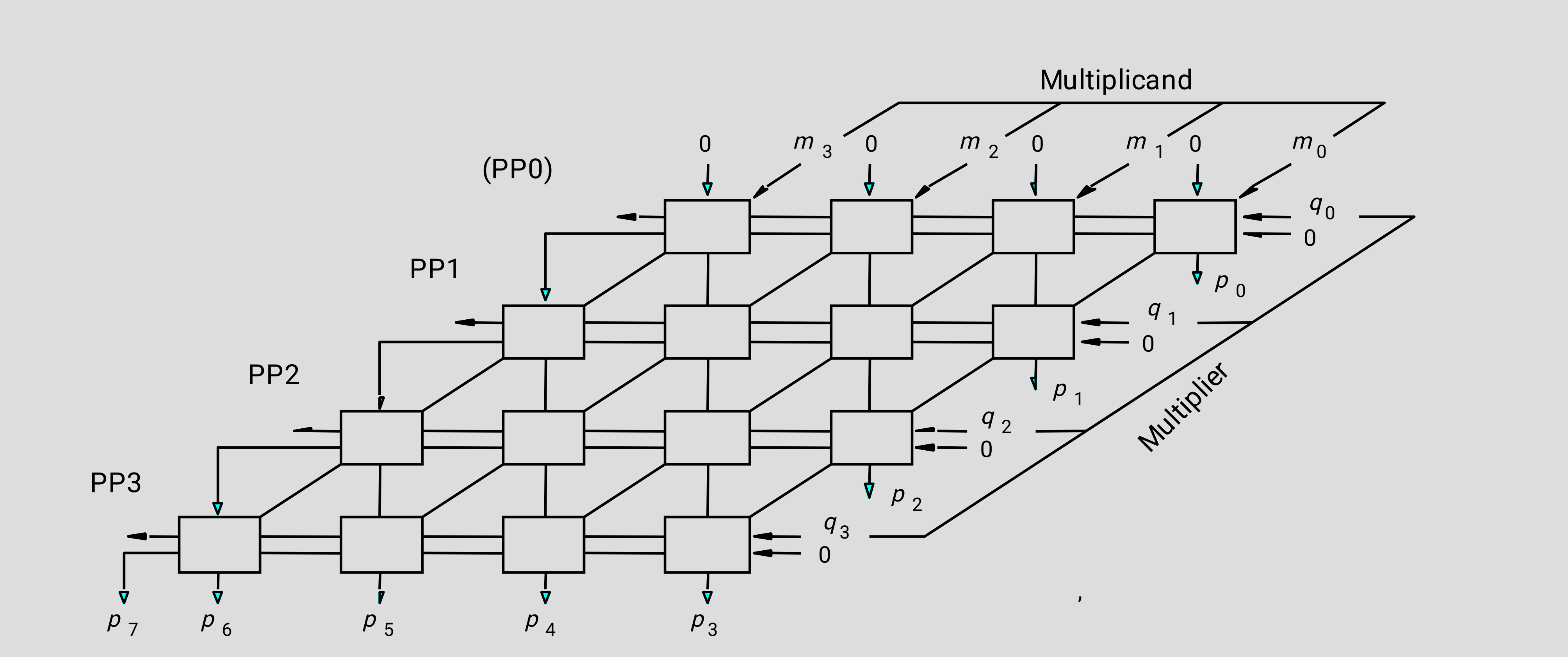
Multiplication of unsigned numbers

- We added the partial products at end.
  - Alternative would be to add the partial products at each stage.
- Rules to implement multiplication are:
  - If the *i*th bit of the multiplier is 1, shift the multiplicand and add the shifted multiplicand to the current value of the partial product.
  - Hand over the partial product to the next stage
  - Value of the partial product at the start stage is 0.

Typical multiplication cell


Bit of incoming partial product (PPi)
jth multiplicand bit
ith multiplier bit
ith multiplier bit
FA
carry out
carry in
Bit of outgoing partial product (PP(i+1))

Combinatorial array multiplier



Multiplicand is shifted by displacing it through an array of adders.

Multiplication

Combinatorial array multiplier

- Combinatorial array multipliers are:
  1. Extremely inefficient.
  2. Have a high gate count for multiplying numbers of practical size such as 32-bit or 64-bit numbers.
  3. Perform only one function, namely, unsigned integer product.
- Improve gate efficiency by using a mixture of combinatorial array techniques and sequential techniques requiring less combinational logic.

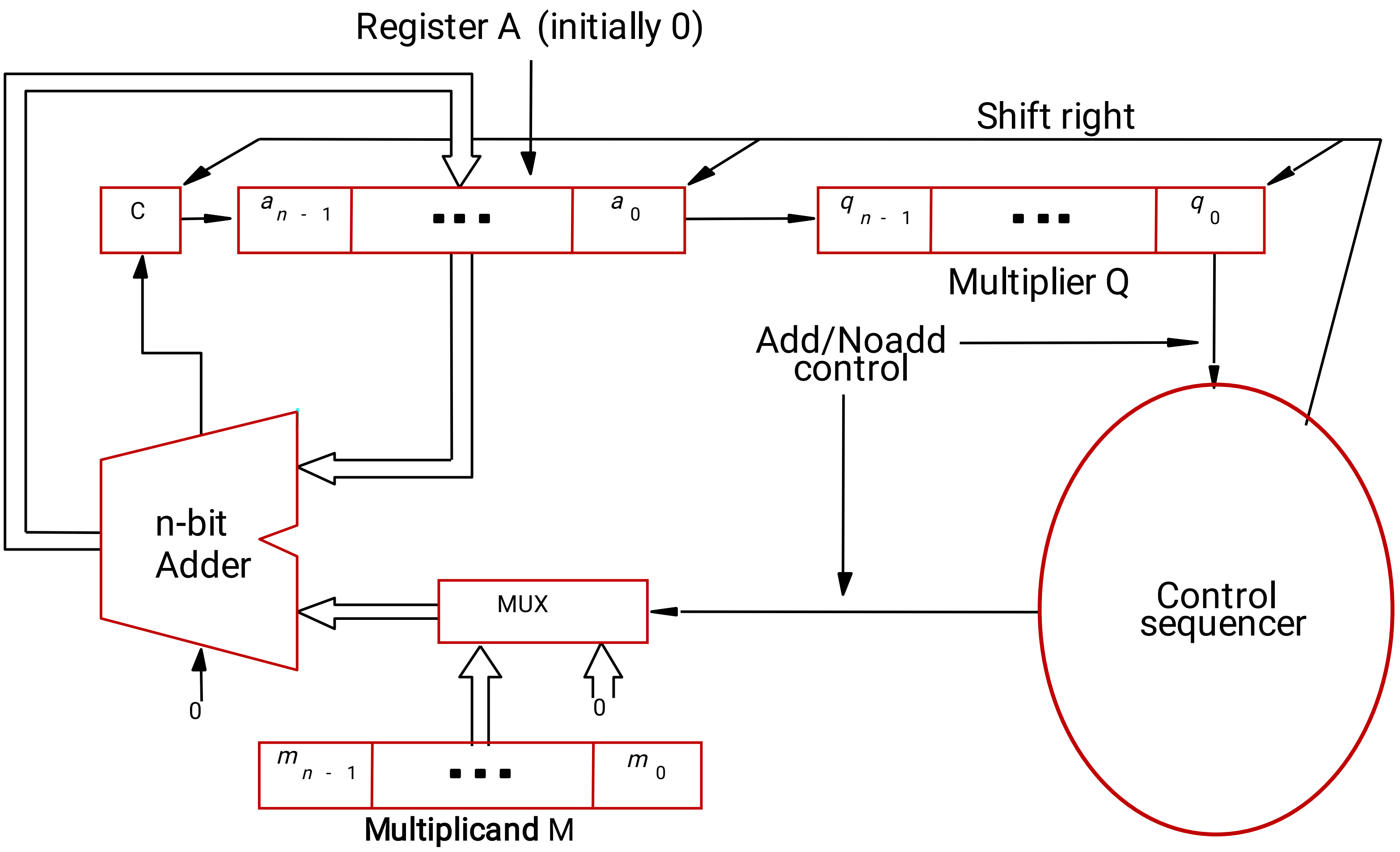
Multiplication

Sequential multiplication

- Recall the rule for generating partial products:
    1. If the ith bit of the multiplier is 1, add the appropriately shifted multiplicand to the current partial product.
    2. Multiplicand has been shifted left when added to the partial product.
- However, adding a left-shifted multiplicand to an unshifted partial product is equivalent to adding an unshifted multiplicand to a right-shifted partial product

Multiplication

Sequential Circuit Multiplier



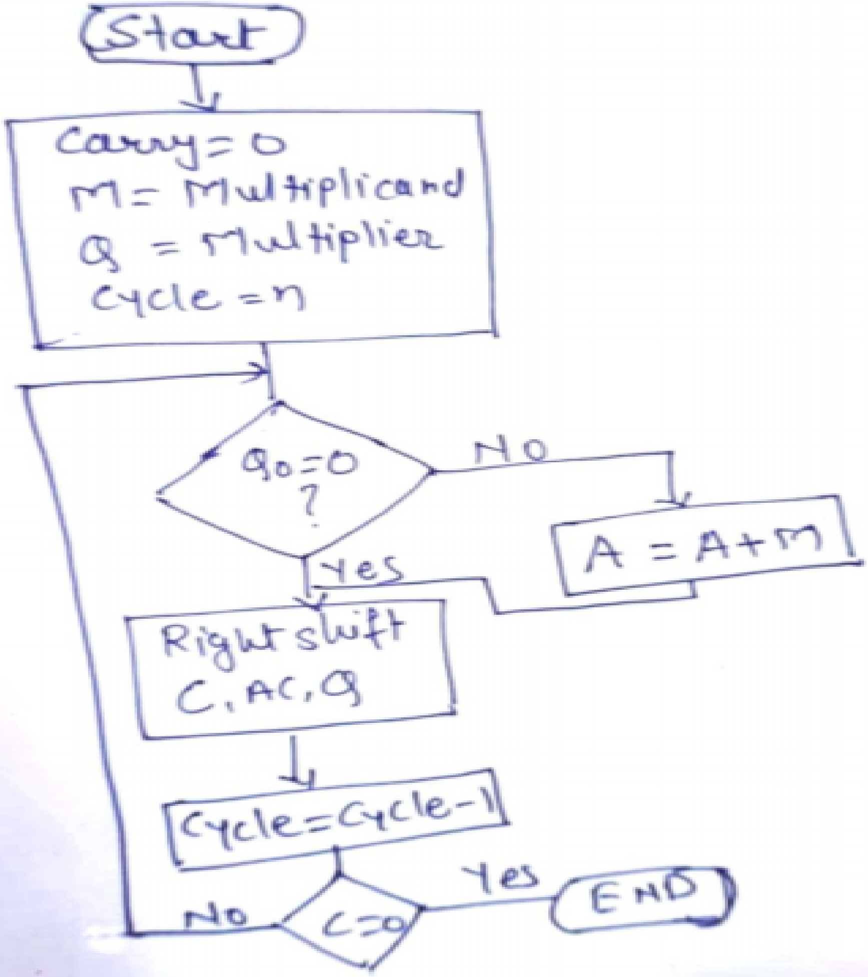
LECTURE 2: ARITHMETICS

Multiplication

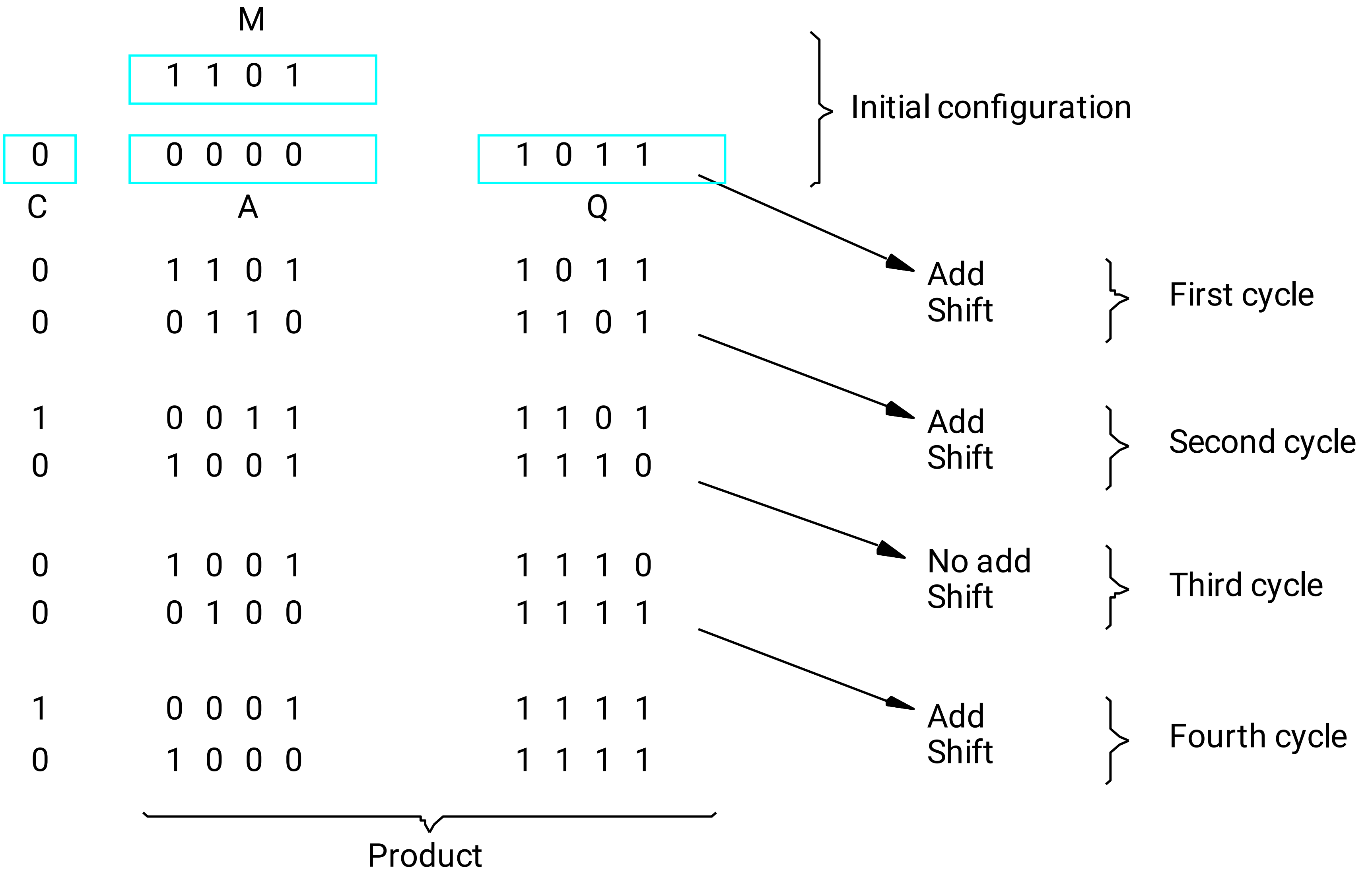
UNIT-4

CAO

- Algorithm for multiplying fixed point unsigned number



Sequential multiplication



M

1 1 0 1

C	A	Q		
0	0 0 0 0	1 0 1 1		Initial configuration
0	1 1 0 1	1 0 1 1	Add	First cycle
0	0 1 1 0	1 1 0 1	Shift	
1	0 0 1 1	1 1 0 1	Add	Second cycle
0	1 0 0 1	1 1 1 0	Shift	
0	1 0 0 1	1 1 1 0	No add	Third cycle
0	0 1 0 0	1 1 1 1	Shift	
1	0 0 0 1	1 1 1 1	Add	Fourth cycle
0	1 0 0 0	1 1 1 1	Shift	

Product

THANK YOU

Booth's Algorithm

Multiplier		Version of multiplicand selected by bit *i*
Bit *i*	Bit *i*-1	
0	0	0XM
0	1	+1XM
1	0	_1XM
1	1	0XM

Booth multiplier recoding table.

Signed Multiplication

- Considering 2’s-complement signed operands, what will happen to (-13)×(+11) if Sequential multiplication following the same method of unsigned multiplication?

```
                    1   0   0   1   1   (-13)
                    0   1   0   1   1   (+11)
                  ---------------------
1   1   1   1   1   1   0   0   1   1

1   1   1   1   1   0   0   1   1
                                          Sign extension is
0   0   0   0   0   0   0   0             shown in blue

1   1   1   0   0   1   1

0   0   0   0   0   0
---------------------------------------
1   1   0   1   1   1   0   0   0   1   (-143)
```

Sign extension of negative multiplicand.

Booth’s Algorithm

- Consider in a multiplication, the multiplier is positive 0011110, how many appropriately shifted versions of the multiplicand are added in a standard procedure?

```
              0 1 0 1 1 0 1
              0 0+1+1+1+1 0
          ---------------
              0 0 0 0 0 0 0
            0 1 0 1 1 0 1
          0 1 0 1 1 0 1
        0 1 0 1 1 0 1
      0 1 0 1 1 0 1
    0 0 0 0 0 0 0
  0 0 0 0 0 0 0
---------------------------
0 0 0 1 0 1 0 1 0 0 0 1 1 0
```

Booth’s Algorithm

- Since 0011110 = 0100000 – 0000010, if we use the expression to the right, what will happen?

```
                       0  1  0  1  1  0  1
                       0 +1  0  0  0 -1  0
                     ---------------------
  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  1  1  1  1  1  1  1  0  1  0  0  1  1
  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0
  0  0  0  1  0  1  1  0  1
  0  0  0  0  0  0  0  0
------------------------------------------
  0  0  0  1  0  1  0  1  0  0  0  1  1  0
```

2’s complements of multiplicand

Booth’s Algorithm

- In general, in the Booth scheme, -1 times the shifted multiplicand is selected when moving from 0 to 1, and +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left.

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

⇩

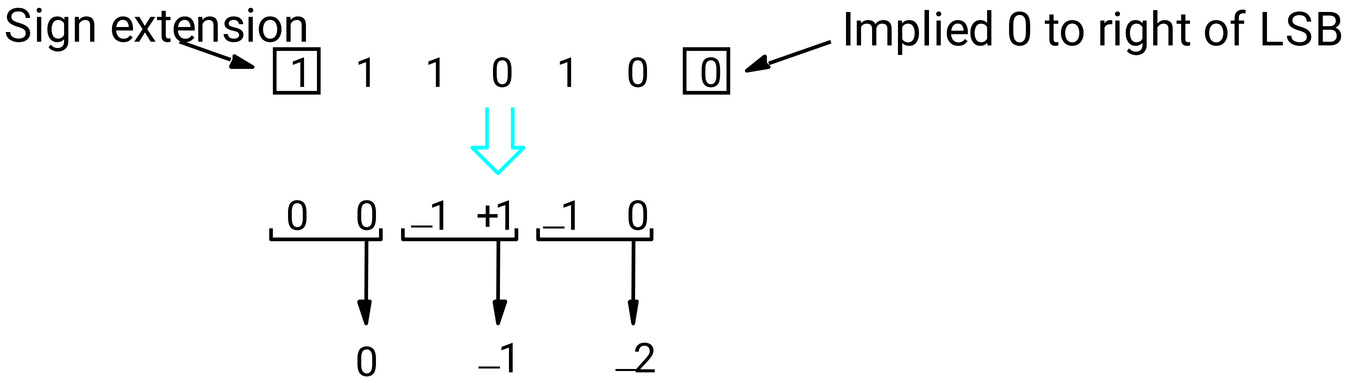
0+1-1+1 0-1 0+1 0 0-1+1-1+1 0-1 0 0

Booth recoding of a multiplier.

THANK YOU

Bit-Pair Recoding of Multipliers:

- Bit-pair recoding halves the maximum number of summands (versions of the multiplicand).



(a) Example of bit-pair recoding derived from Booth recoding

Bit-Pair Recoding of Multipliers:

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position *i*
i+1	*i*	*i*−1	
0	0	0	0 X M
0	0	1	+1 X M
0	1	0	+1 X M
0	1	1	+2 X M
1	0	0	−2 X M
1	0	1	−1 X M
1	1	0	−1 X M
1	1	1	0 X M

(b) Table of multiplicand selection decisions

Bit-Pair Recoding of Multipliers:

```
      0  1  1  0  1   (+13)
   ×  1  1  0  1  0   (- 6)
   ________________
```

```
                 0  1  1  0  1
                 0 -1 +1 -1  0
                 _____________
  0  0  0  0  0  0  0  0  0  0
  1  1  1  1  1  0  0  1  1
  0  0  0  0  1  1  0  1
  1  1  1  0  0  1  1
  0  0  0  0  0  0
  ____________________________
  1  1  1  0  1  1  0  0  1  0   (- 78)
```

```
                 0  1  1  0  1
                 0    -1    -2
                 _____________
  1  1  1  1  1  0  0  1  1  0
  1  1  1  1  0  0  1  1
  0  0  0  0  0  0
  ____________________________
  1  1  1  0  1  1  0  0  1  0
```

THANK YOU