Course – Computer Organization and Architecture

Course Instructor

Dr. Umadevi V

Department of CSE, BMSCE

Unit-4

Arithmetic: Addition and Subtraction of Signed Numbers, Design of Fast Adders, Multiplication of Unsigned Numbers, Multiplication of Signed Numbers

Fast Multiplication: Bit-Pair Recoding of Multipliers, Integer Division, Floating-Point Numbers and Operations : Arithmetic Operations on Floating-Point Numbers, Guard Bits and Truncation , Implementing Floating-Point Operations

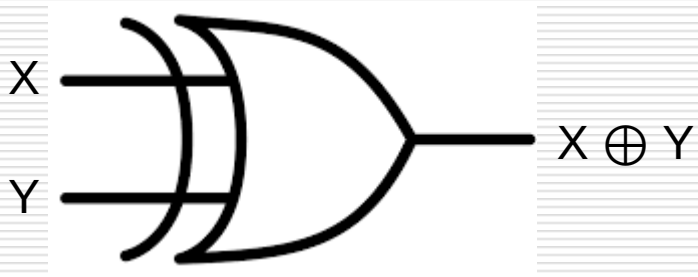
Addition and Subtraction of Signed Numbers

Rough Slide to explain
Designing Logic Circuit for Addition of Two numbers

X	**7**		**0**		**1**		**1**		**1**	
Y	**+6**		**0**	1	**1**	1	**1**	0	**0**	0 ← Carry In
	13		**1**		**1**		**0**		**1**	

Rough Slide to explain
Designing Logic Circuit for Addition of Two numbers

X	Y	Ex-OR $X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0



$$X \oplus Y = \bar{X}Y + X\bar{Y}$$

Rough Slide to explain
Designing Logic Circuit for Addition of Two numbers

X	Y	Logical OR X+Y	$\overline{X+Y}$	$\bar{X}$	$\bar{Y}$	$\bar{X}\,\bar{Y}$	Logical AND XY	$\overline{XY}$	$\bar{X}+\bar{Y}$
0	0	0	1	1	1	1	0	1	1
0	1	1	0	1	0	0	0	1	1
1	0	1	0	0	1	0	0	1	1
1	1	1	0	0	0	0	1	0	0

$$\overline{X+Y} = \bar{X}\,\bar{Y}$$

$$\overline{XY} = \bar{X} + \bar{Y}$$

X	$\bar{X}$	$\bar{\bar{X}}$
0	1	0
1	0	1

$$\bar{X} = \bar{\bar{X}}$$

Rough Slide to explain
Designing Logic Circuit for Addition of Two numbers

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

Rough Slide to explain
Designing Logic Circuit for Addition of Two numbers

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

$$S_i = \bar{X}_i \bar{Y}_i C_i + \bar{X}_i Y_i \bar{C}_i + X_i \bar{Y}_i \bar{C}_i + X_i Y_i C_i$$

$$= \bar{X}_i \bar{Y}_i C_i + X_i Y_i C_i + \bar{X}_i Y_i \bar{C}_i + X_i \bar{Y}_i \bar{C}_i$$

$$= C_i (\bar{X}_i \bar{Y}_i + X_i Yi) + \bar{C}_i (\bar{X}_i Yi + X_i \bar{Y}_i)$$

$$= C_i (\overline{X_i \oplus Y_i}) + \bar{C}_i (X_i \oplus Y_i)$$

$$S_i = C_i \oplus X_i \oplus Y_i$$

$$\overline{X_i \oplus Y_i} = \overline{\bar{X}_i Yi + Xi\bar{Y}_i}$$
$$= \overline{\bar{X}_i Yi}\ \overline{Xi\bar{Y}_i}$$
$$= (\bar{\bar{X}}_i + \bar{Y}_i)(\bar{X}_i + \bar{\bar{Y}}_i)$$
$$= X_i\bar{X}_i + X_iY_i + \bar{Y}_i\bar{X}_i + \bar{Y}_iY_i$$
$$= X_iY_i + \bar{Y}_i\bar{X}_i$$

Rough Slide to explain
Designing Logic Circuit for Addition of Two numbers

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

$C_i = \bar{X}_i Y_i Ci + X_i \bar{Y}_i C_i + X_i Y_i \bar{C}_i + X_i Yi C_i$

C_{i+1}

$C_{i+1} = X_iY_i + X_iC_i + Y_iC_i$

Karnaugh map

	00	01	11	10
O			1	
1		1	1	1

Addition of signed numbers

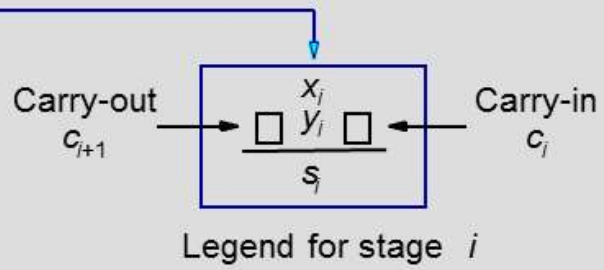
x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

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

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

Example:

$$\begin{array}{rcl} X & & 7 \\ +\ Y & = & +\ 6 \\ \hline Z & & 13 \end{array} = \begin{array}{r} 0\ 1\ 1\ 1 \\ +\ 0\ 0\ 1\ 1\ 0\ 0 \\ \hline 1\ 1\ 0\ 1 \end{array}$$



At the i^{th} stage:

Input:

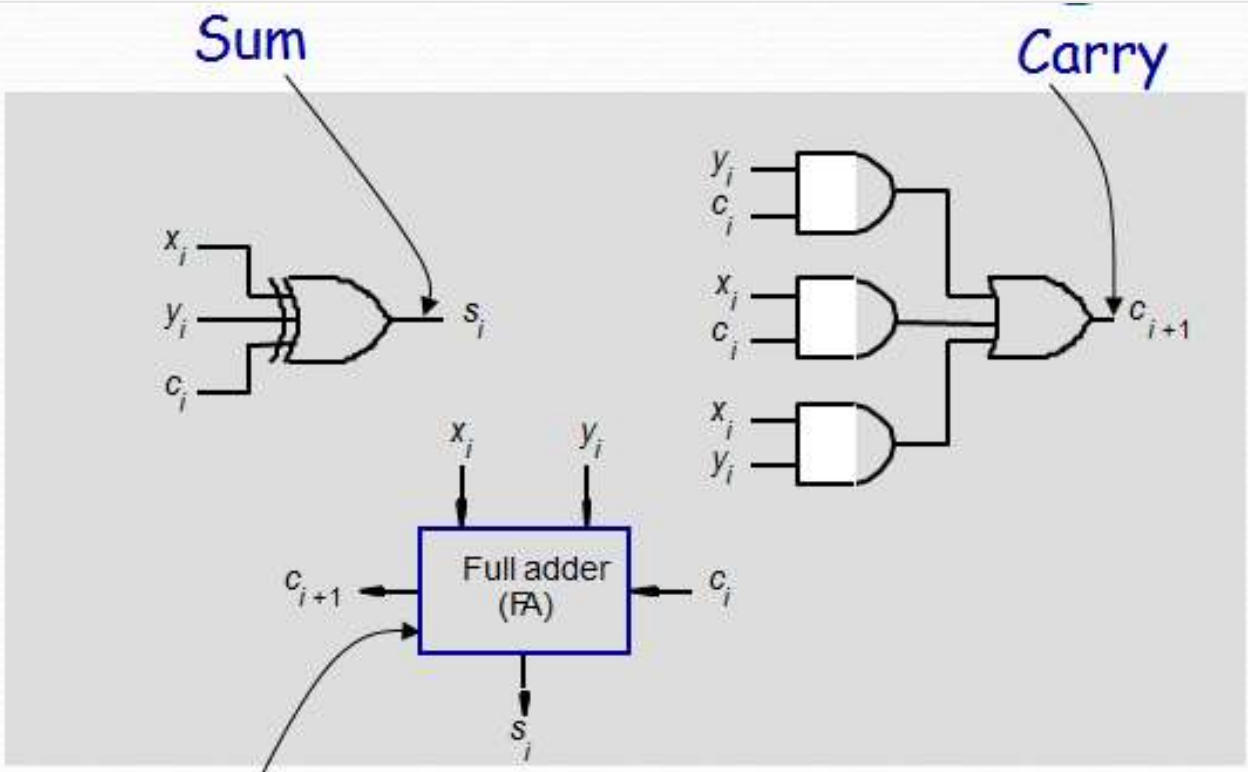
c_i is the carry-in

Output:

s_i is the sum

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

Addition logic for a single stage

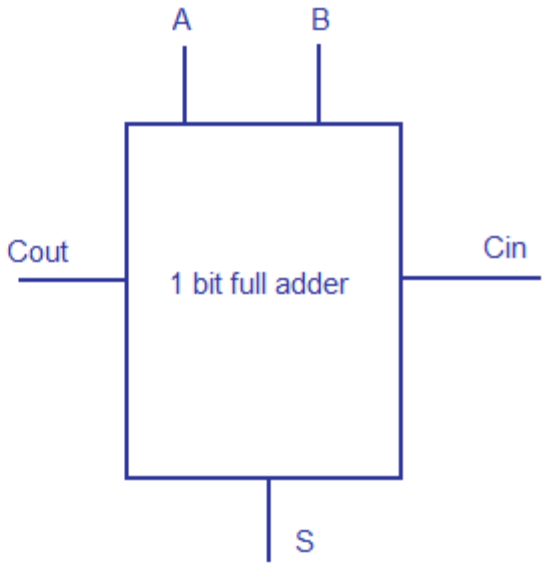


Full Adder (FA): Symbol for the complete circuit for a single stage of addition.

Full Adder

- Full adder is a logic circuit that adds two input operand bits plus a Carry in bit and outputs a Carry out bit and a sum bit.. The Sum out (Sout) of a full adder is the XOR of input operand bits A, B and the Carry in (Cin) bit.

Inputs			Outputs	
A	B	Cin	Cout	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

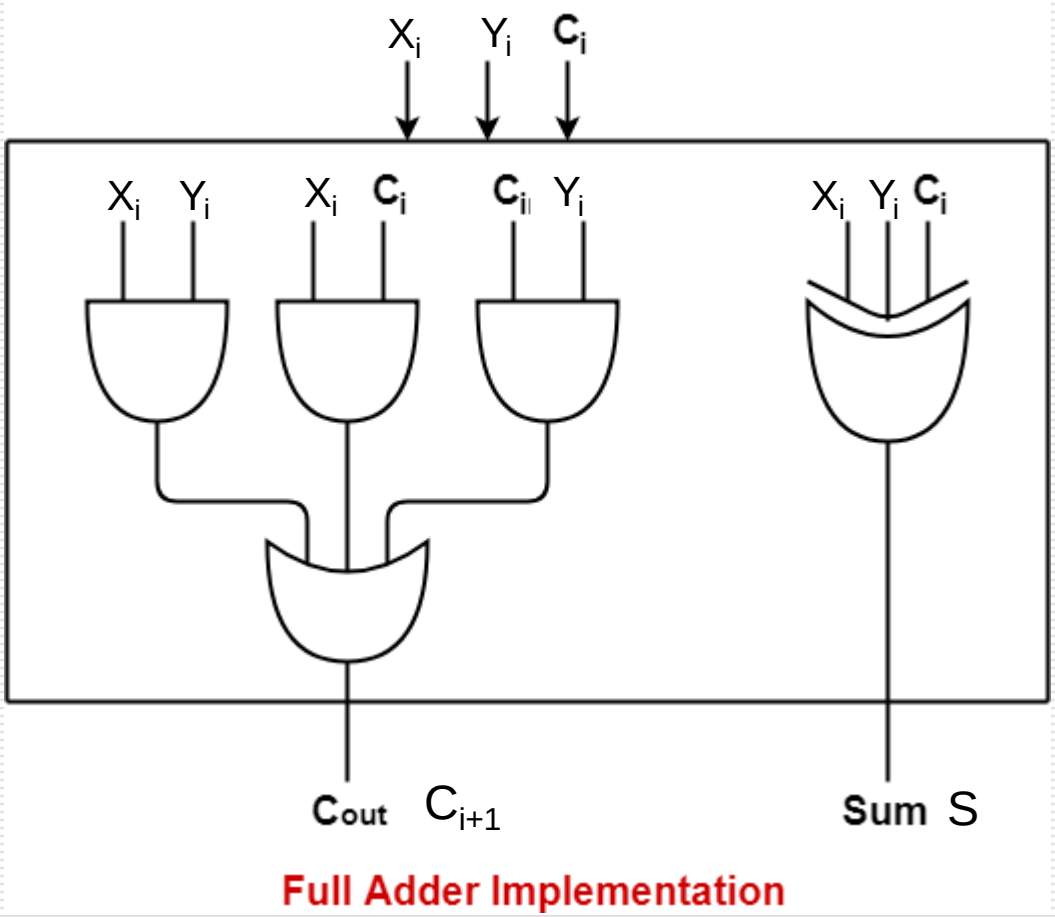


1 bit full adder truth table & schematic

Full Adder

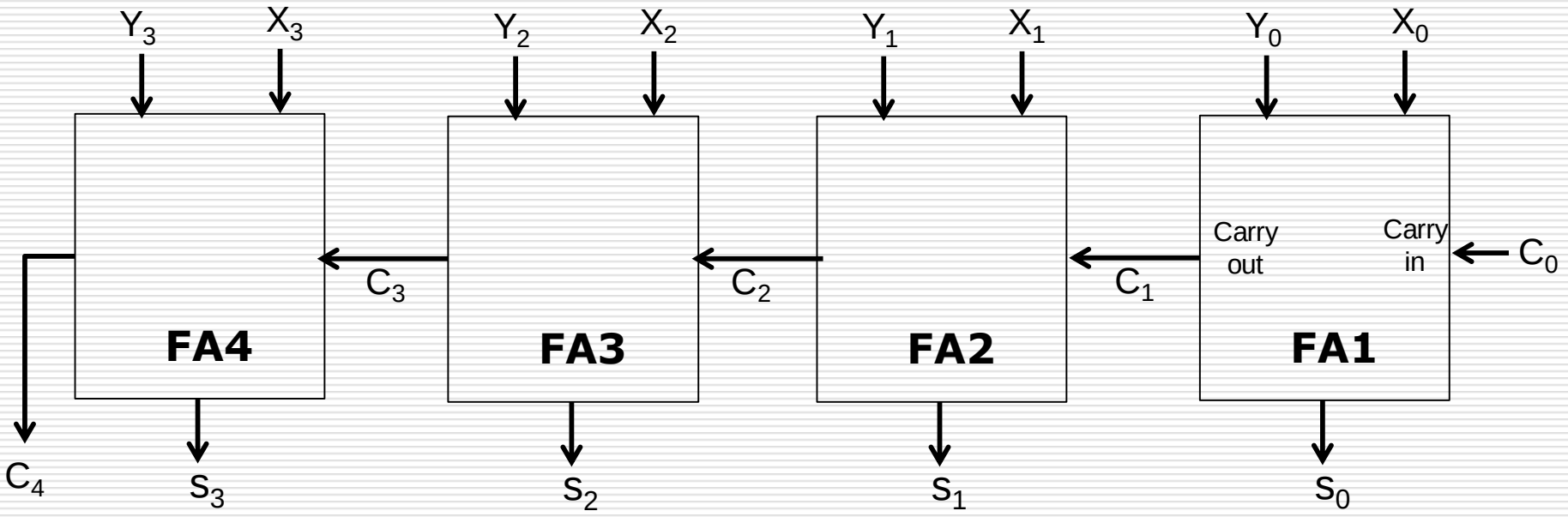
$$S = X_i \oplus Y_i \oplus C_i$$

$$C_{i+1} = X_iY_i + X_iC_i + Y_iC_i$$



Full Adder Implementation

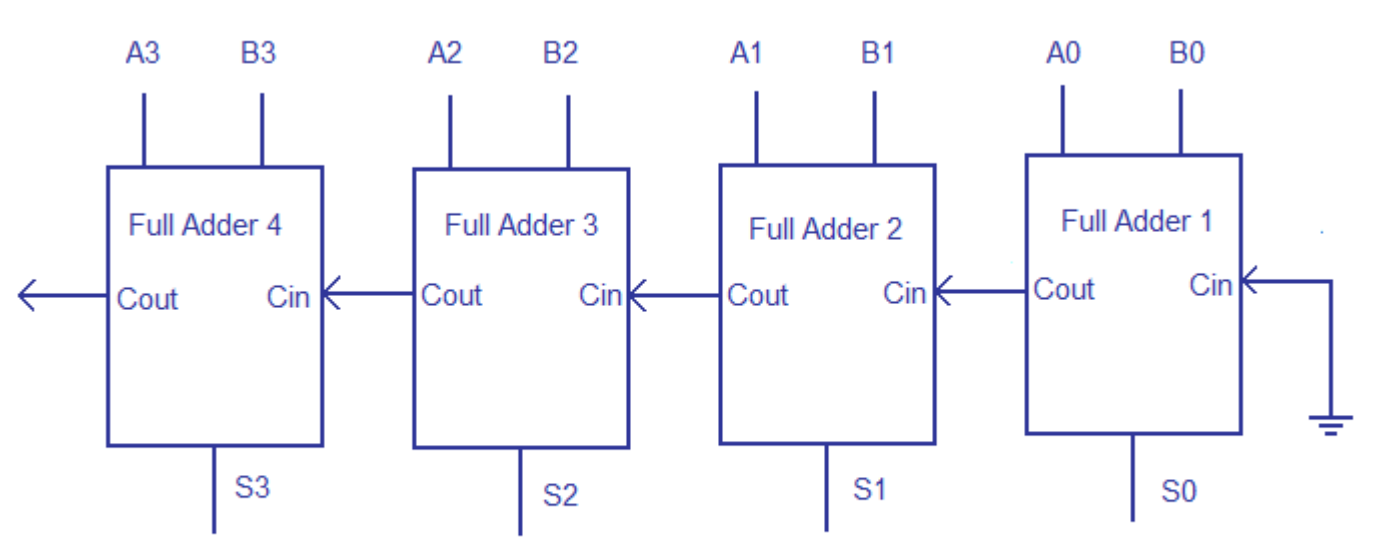
Two 4-bit Adder (Two 4-Bit Ripple Carry Adder) using four Full Adder (FA)



X	**7**		**0**		**1**		**1**		**1**	
Y	**+6**		**0**	1	**1**	1	**1**	0	**0**	0 ← Carry In
	13		**1**		**1**		**0**		**1**	

Ripple carry adder

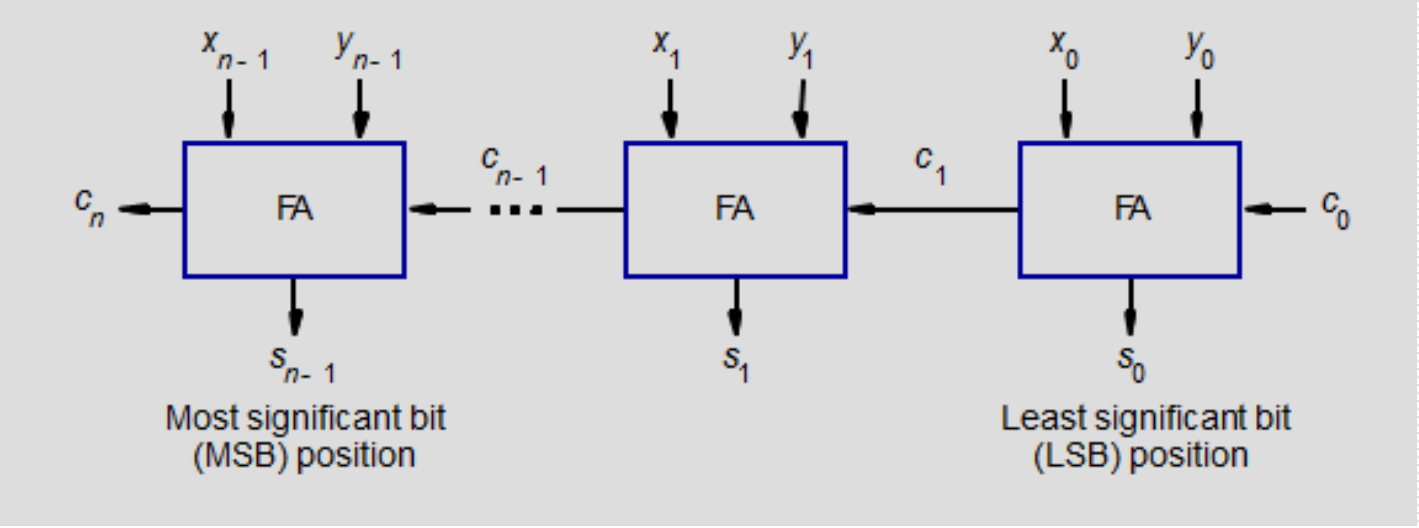
A ripple carry adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascaded, with the carry output from each full adder connected to the carry input of the next full adder in the chain. Figure below, shows the interconnection of four full adder (FA) circuits to provide a 4-bit ripple carry adder. Notice from the Figure that the input is from the right side because the first cell traditionally represents the least significant bit (LSB). Bits A0 and B0 in the figure represent the least significant bits of the numbers to be added. The sum output is represented by the bits S0 to S3 .



4 bit ripple carry adder

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.

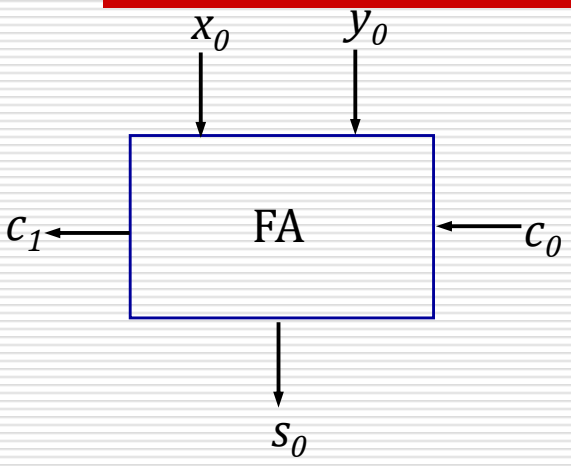


- Carry-in c_0 into the LSB position provides a convenient way to perform subtraction.

Question

Using Full adders, how many full adders are required to construct 16-bit ripple carry adder ?

Computing the Delay

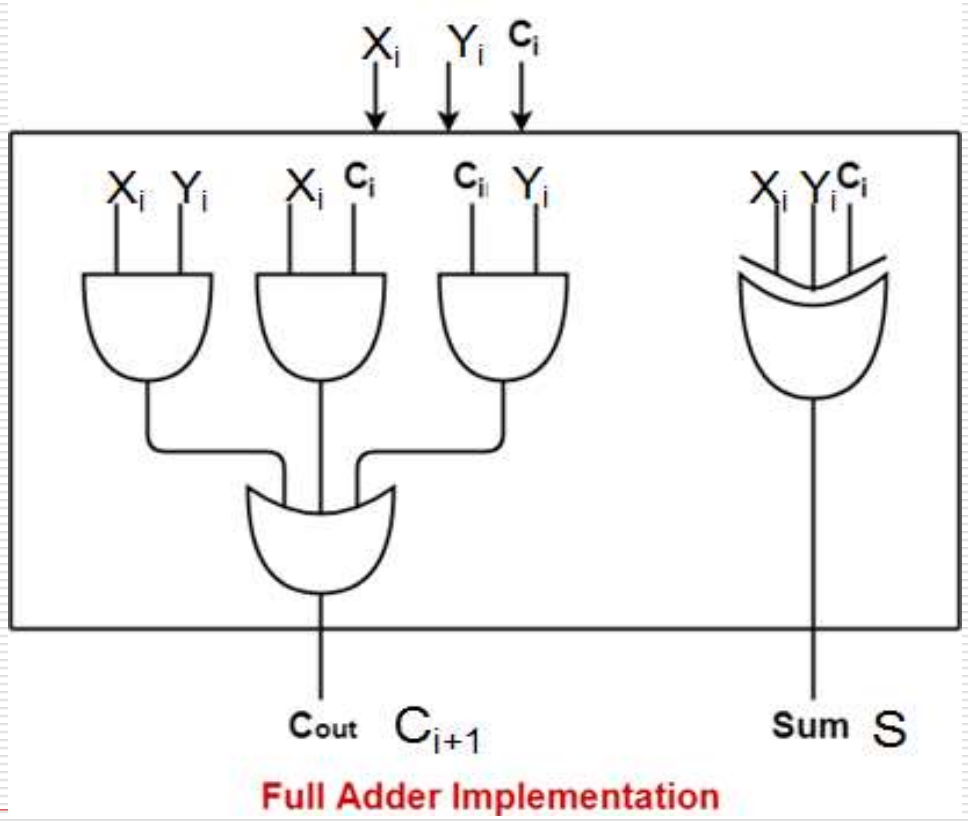


Consider 0^{th} stage:

- c_1 is available after 2 gate delays.
- s_0 is available after 1 gate delay.

$$S = X_i \oplus Y_i \oplus C_i$$

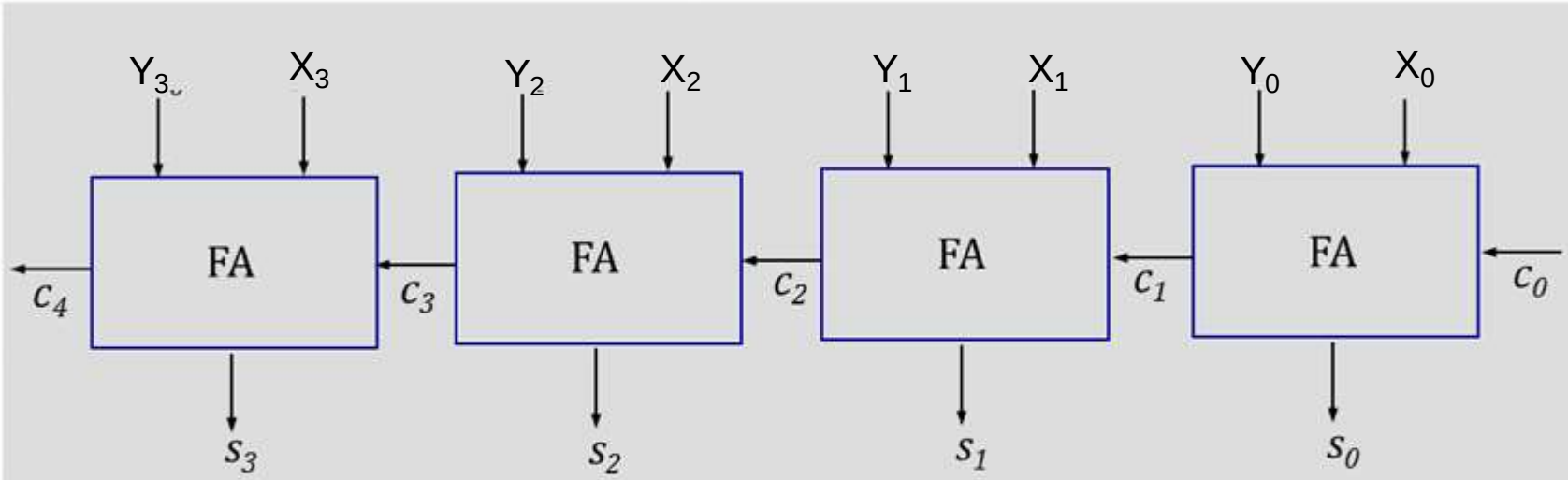
$$C_{i+1} = X_iY_i + X_iC_i + Y_iC_i$$



Full Adder Implementation

Computing the 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,

$s_{n-1} = 2(n-1) + 1 = 2n - 1$ Gate delays

c_n is available after 2n Gate delays

Delay in Ripple carry adder

- Carry propagation delay of a full adder is the time taken by it to produce the output carry bit.
- Sum propagation delay of a full adder is the time taken by it to produce the output sum bit.
- Worst case delay of a ripple carry adder is the time after which the output sum bit and carry bit becomes available from the last full adder.
- We know, In ripple carry adder, a full adder becomes active only when its carry in is made available by its adjacent less significant full adder.
- When carry in becomes available to the full adder, it starts its operation and produces the corresponding output sum bit and carry bit.

Question

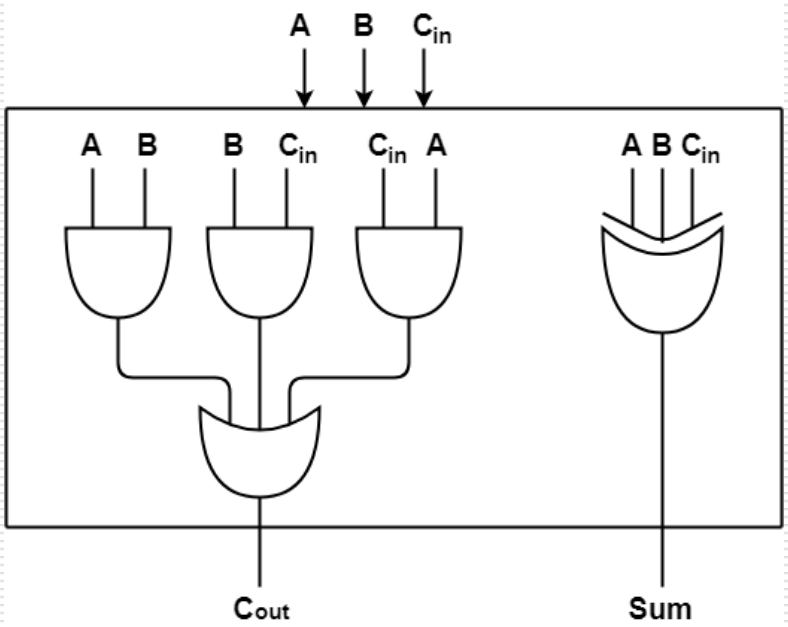
Following figure shows the implementation of full adders in a 16-bit ripple carry adder realized using 16 identical full adders. The propagation delay of the XOR, AND and OR gates are 20 ns, 15 ns and 10 ns respectively. The worst case delay of this 16 bit adder will be _____?

A) 395 ns

B) 220 ns

C) 400 ns

D) 300 ns



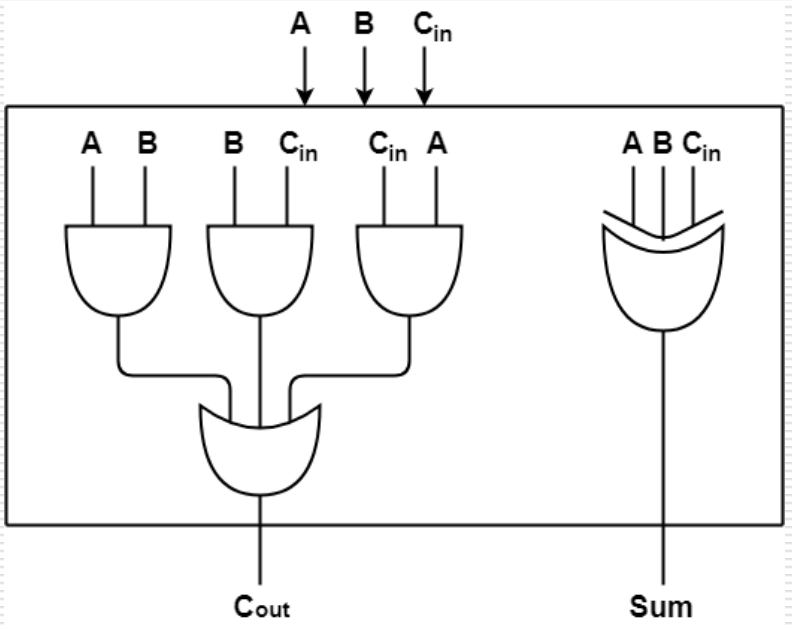
Full Adder Implementation

Answer

We will consider the last full adder for worst case delay.

Time after which output carry bit becomes available from the last full adder

= Total number of full adders x Carry propagation delay of a full adder



Full Adder Implementation

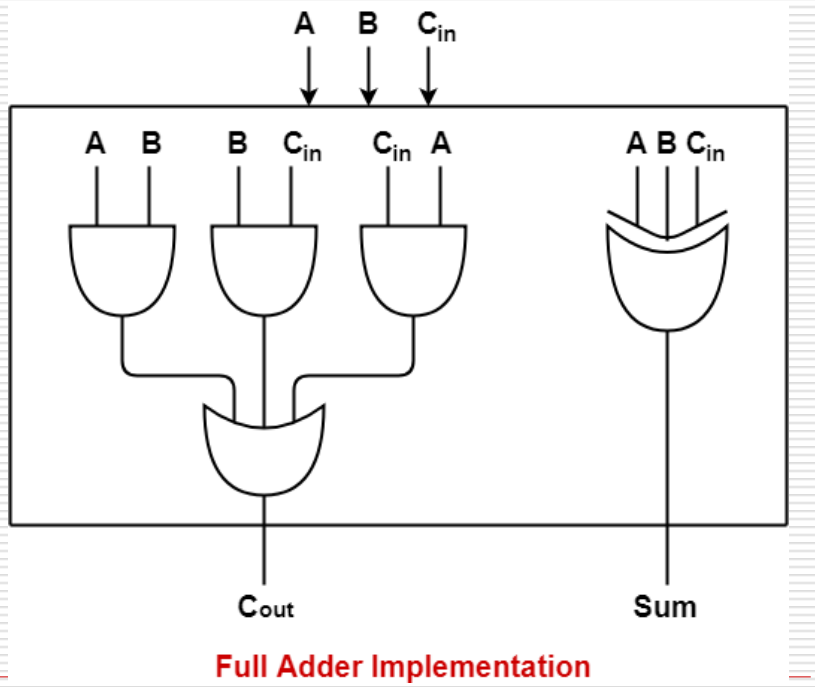
Answer

We will consider the last full adder for worst case delay.

Time after which output carry bit becomes available from the last full adder

= Total number of full adders x Carry propagation delay of a full adder

= Total number of full adders x { Propagation delay of AND gate +
Propagation delay of OR gate }



Full Adder Implementation

Answer

We will consider the last full adder for worst case delay.

Time after which output carry bit becomes available from the last full adder

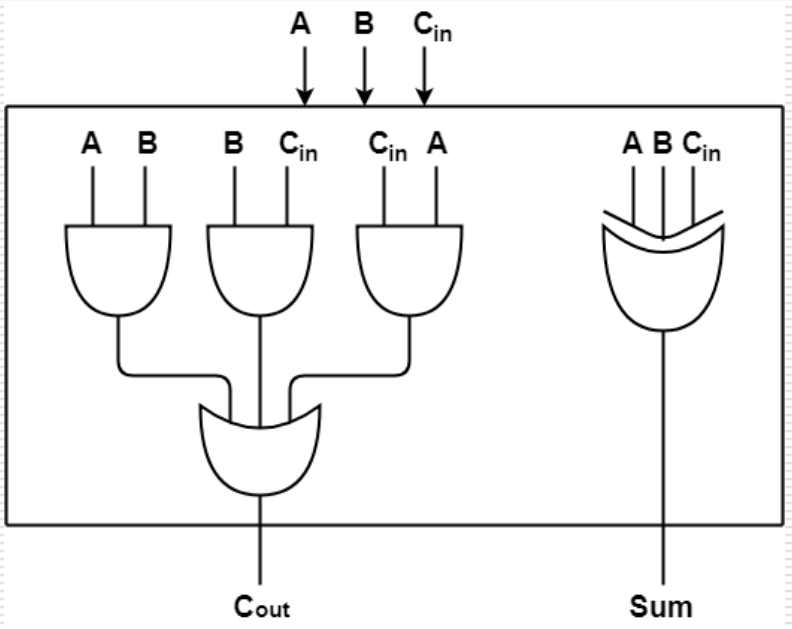
= Total number of full adders x Carry propagation delay of a full adder

= Total number of full adders x { Propagation delay of AND gate + Propagation delay of OR gate }

= 16 x { 15 ns + 10 ns }

= 16 x 25 ns

= 400 ns



Full Adder Implementation

Answer (Contd…)

We will consider the last full adder for worst case delay.

Time after which output carry bit becomes available from the last full adder

= Total number of full adders x Carry propagation delay of a full adder

= Total number of full adders x { Propagation delay of AND gate +
Propagation delay of OR gate }

= 16 x { 15 ns + 10 ns }

= 16 x 25 ns

= 400 ns

Time after which output sum bit becomes available from the last full adder

= Time taken for its carry in to become available +
Sum propagation delay of a full adder

Answer (Contd…)

We will consider the last full adder for worst case delay.

Time after which output carry bit becomes available from the last full adder

= Total number of full adders x Carry propagation delay of a full adder

= Total number of full adders x { Propagation delay of AND gate + Propagation delay of OR gate }

= 16 x { 15 ns + 10 ns }

= 16 x 25 ns

= 400 ns

Time after which output sum bit becomes available from the last full adder

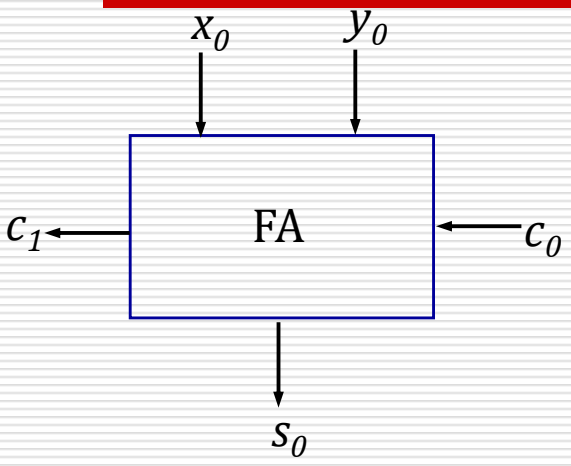
= Time taken for its carry in to become available + Sum propagation delay of a full adder

= { Total number of full adders before last full adder X Carry propagation delay of a full adder} + Propagation delay of XOR gate

= { 15 x (15 ns + 10 ns) } + 20 ns

= 395 ns

Computing the Delay

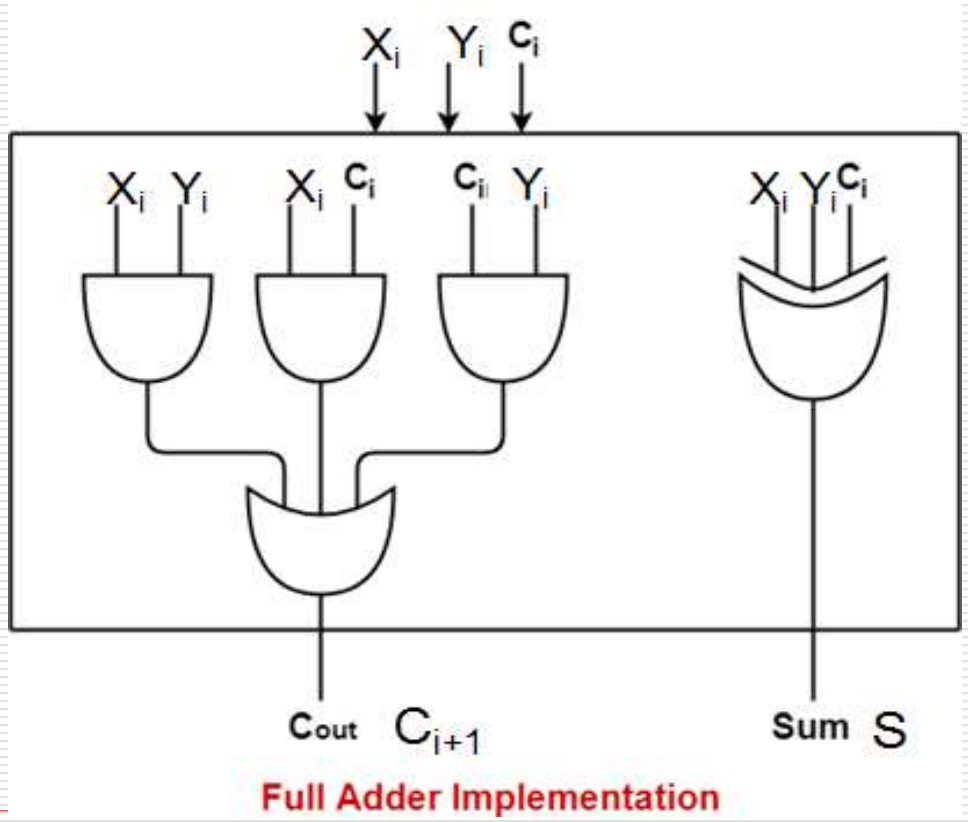


Consider 0^{th} stage:

- c_1 is available after 2 gate delays.
- s_0 is available after 1 gate delay.

$$S = X_i \oplus Y_i \oplus C_i$$

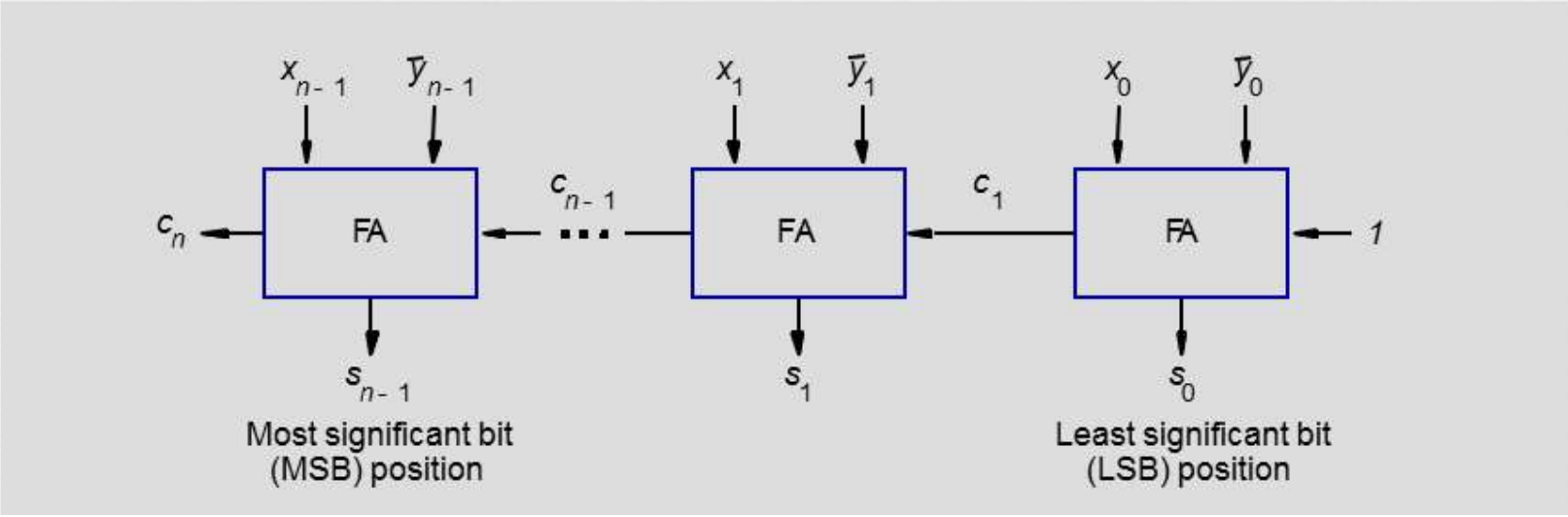
$$C_{i+1} = X_iY_i + X_iC_i + Y_iC_i$$



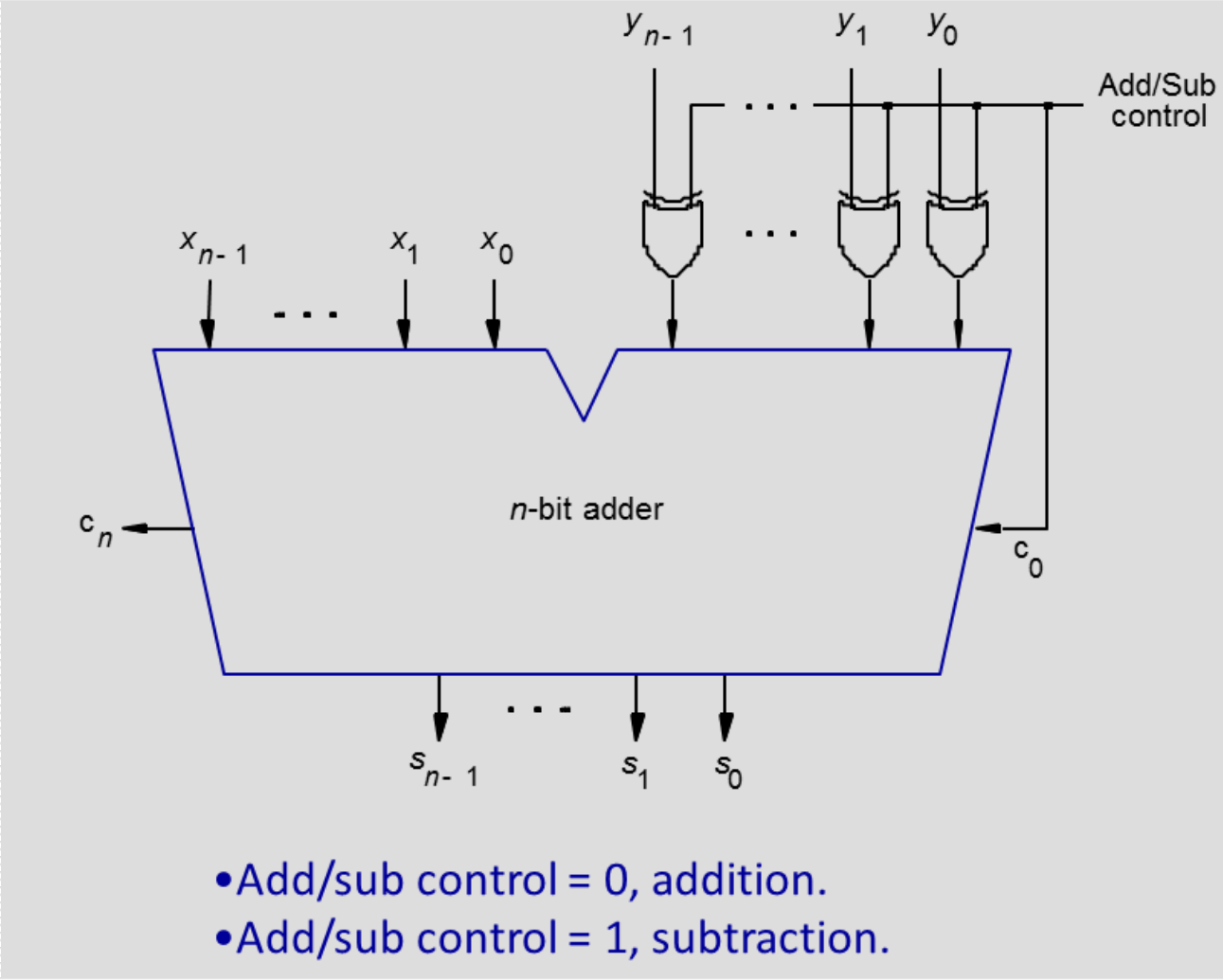
Full Adder Implementation

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.
- $X - Y = X + \overline{Y} + 1$
- 2's complement of positive and negative numbers is computed similarly.

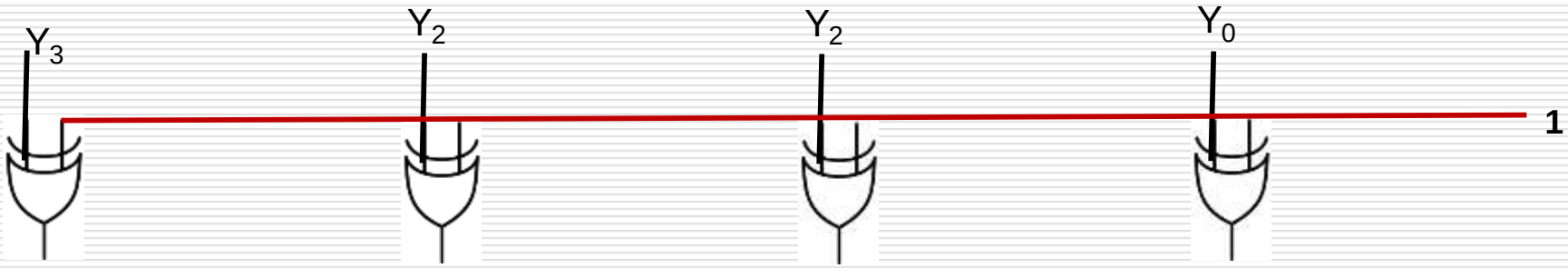


n-bit adder/subtractor (contd..)



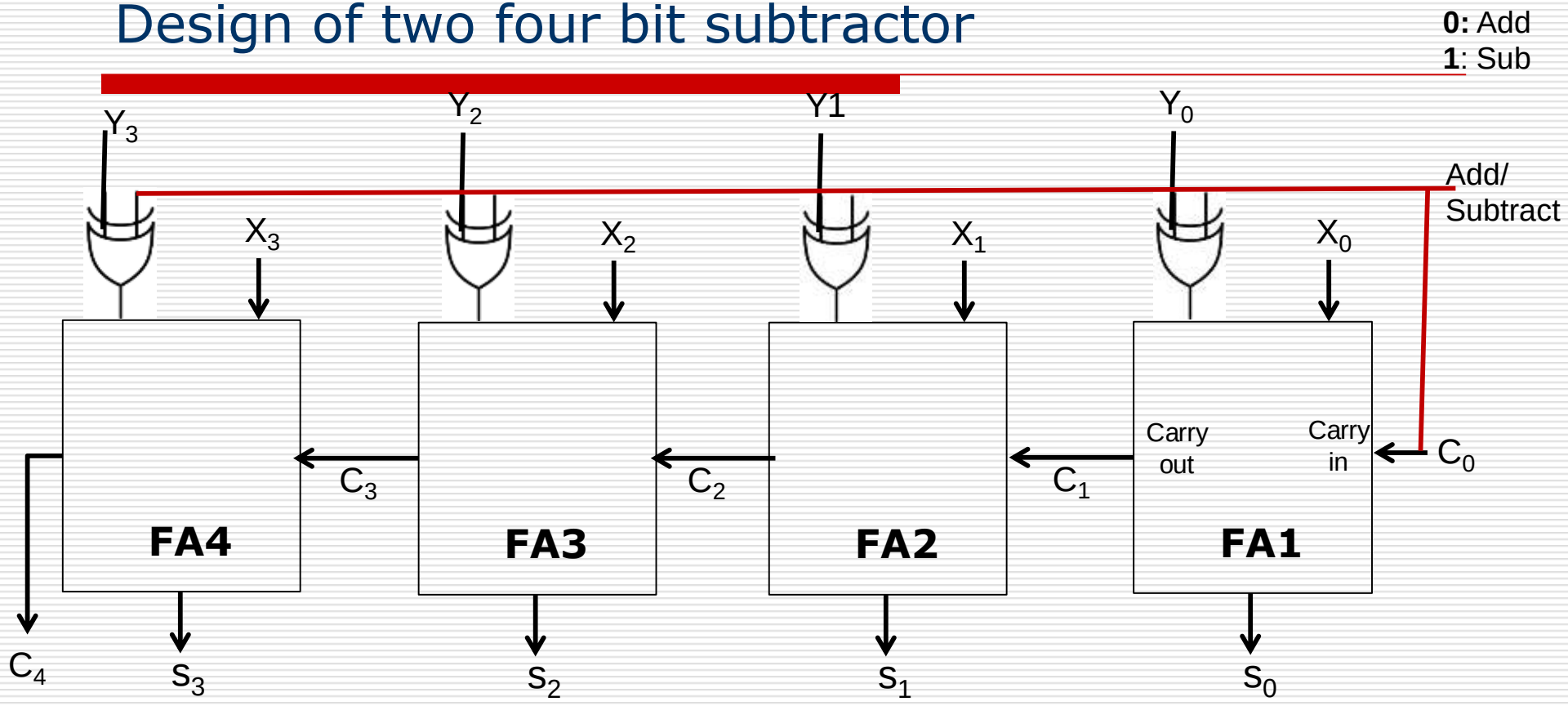
- Add/sub control = 0, addition.
- Add/sub control = 1, subtraction.

Rough Slide to explain
Design of two four bit subtractor



X	Y	Ex-OR $X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

Design of two four bit subtractor



X	**7**		**0**		**1**		**1**	
-Y	**-6**	**+**	**1**		**0**		**1**	**0** ← 2's Complement of 6
	1	**1**	**0**		**0**		**0**	**1**
		Ignore Carryout						

Detecting Overflows

It occurs when the signs of the two operands are the same, but the sign of the result is different.

Overflow can be added to the *n*-bit adder by implementing the logic expression

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:

$$Overflow = x_{n-1} y_{n-1} \bar{s}_{n-1} + \bar{x}_{n-1} \bar{y}_{n-1} s_{n-1}$$

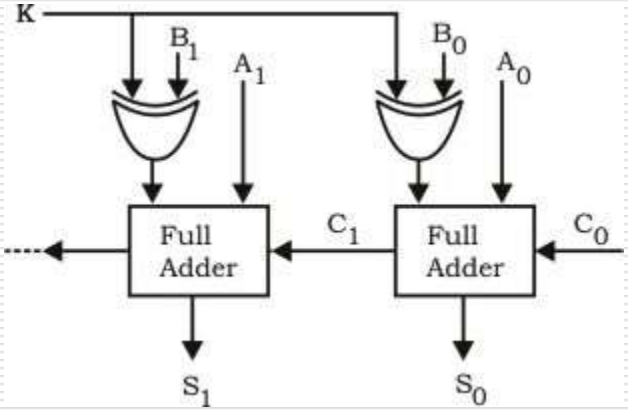
OR

$$Overflow = c_n \oplus c_{n-1}$$

X	**7**		**0**		**1**		**1**		**1**	
Y	**+6**		**0**	1	**1**	1	**1**	0	**0**	0 ← Carry In
	13		**1**		**1**		**0**		**1**	

Question

Consider the ALU shown below. If the operands are in 2's complement representation, which of the following operations can be performed by suitably setting the control lines K and C0 only (+ and - denote addition and subtraction respectively) ?



a. A + B, and A - B, but not A + 1

b. A + B, and A + 1, but not A – B

c. A + B, but not A - B, or A + 1

d. A + B, and A - B, and A + 1

Answer: (a).A + B, and A - B, but not A + 1

Answer

Explanation: We can set value of k and c as 0 or 1

Two things we need to know—

If we take xor of any number with 1 we get it in its complement form.

If we take xor of any number with 0 we get that number itself.

So on setting k=1 we can get –B and c will work like select signal

Like c=0 means add

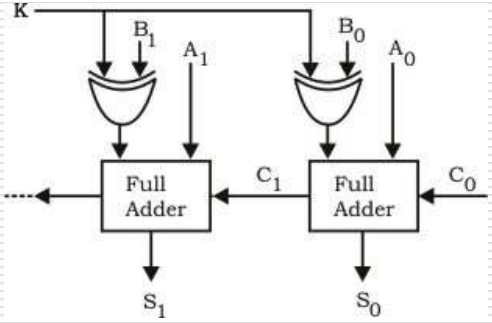
C=1 means subtract

Hence with k=1 c=1 we get A-B

With K=0 c=0 we get A+B

We need b=1,c=0 or b=0,c=1 to get A+1
since b isn't predefined we can't get A+1

So Ans is (a) part A + B, and A - B, but not A + 1



Unit-4

Design of Fast Adders

Design of Fast Adders

- Source of bottleneck with Ripple Carry Adder is Carry Signal Generation
- Propogation of Carry Signals should be Faster

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 Addition

$S = X_i \oplus Y_i \oplus C_i$

$C_{i+1} = X_iY_i + X_iC_i + Y_iC_i$

$C_{i+1} = X_iY_i + C_i(X_i + Y_i)$

$= G_i + C_iP_i$

Where $G_i = X_i\ Y_i$ Generate Function

$P_i = X_i + Y_i$ Propogate Function

Note: Propogate function means whether an input carry produces and output carryout

Note: Irrespective of P_i and C_i, if G_i is 1 then carry out is 1

Carry Lookahead Addition (Contd…)

$S = X_i \oplus Y_i \oplus C_i$

$C_{i+1} = X_iY_i + C_i(X_i + Y_i)$

$= G_i + C_iP_i$

Where $G_i = X_i Y_i$ Generate Function

$P_i = X_i + Y_i$ Propogate Function

X_i	Y_i	$G_i = X_iY_i$
0	0	0
0	**X**	0
X	0	0
1	1	1

Note: **X** any thing 0 or 1

Carry Lookahead Addition (Contd…)

$S = X_i \oplus Y_i \oplus C_i$

$C_{i+1} = X_iY_i + C_i(X_i + Y_i)$

$= G_i + C_iP_i$

Where $G_i = X_i Y_i$ Generate Function

$P_i = X_i + Y_i$ Propogate Function

X_i	Y_i	$G_i = X_iY_i$
0	0	0
0	**X**	0
X	0	0
1	1	1

X_i	Y_i	C_i	$P_i = X_i + Y_i$	P_iC_i
X	**X**	**0**	**X**	**0**
0	**0**	1	0	**0**
X	1	1	1	1
1	**X**	1	1	1

Note: **X** any thing 0 or 1

Carry Lookahead Addition (Contd…)

$S = X_i \oplus Y_i \oplus C_i$

$C_{i+1} = X_iY_i + C_i(X_i + Y_i)$
$= G_i + C_iP_i$
Where $G_i = X_i Y_i$ Generate Function
$P_i = X_i + Y_i$ Propogate Function

X_i	Y_i	$G_i=X_iY_i$
0	0	0
0	**X**	0
X	0	0
1	1	1

X_i	Y_i	C_i	$P_i=X_i+Y_i$	P_iC_i
X	**X**	**0**	**X**	**0**
0	**0**	1	0	**0**
X	1	1	1	1
1	**X**	1	1	1

Note: **X** any thing 0 or 1

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

We can re-write
$P_i = X_i + Y_i$ as $P_i = X_i \oplus Y_i$

Carry Lookahead Addition (Contd…)

$S = X_i \oplus Y_i \oplus C_i$

$C_{i+1} = X_iY_i + C_i(X_i + Y_i)$
$= G_i + C_iP_i$
Where $G_i = X_i Y_i$ Generate Function
$P_i = X_i + Y_i$ Propogate Function

X_i	Y_i	$G_i=X_iY_i$
0	0	0
0	X	0
X	0	0
1	1	1

X_i	Y_i	C_i	$P_i=X_i+Y_i$	P_iC_i
X	X	0	X	0
0	0	1	0	0
X	1	1	1	1
1	X	1	1	1

Note: X any thing 0 or 1

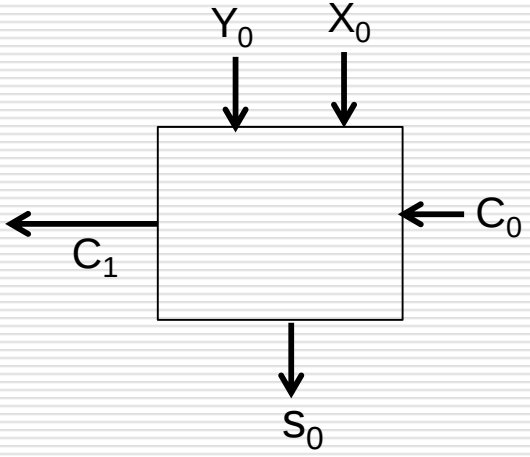
X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

We can re-write
$P_i = X_i + Y_i$ as $P_i = X_i \oplus Y_i$

$P_i = X_i \oplus Y_i$ differs from
$P_i = X_i + Y_i$ only when $X_i=Y_i=1$
But in this case $G_i=1$,so it doe not matter
Whether P_i is 0 or 1

Designing 4-bit Carry Lookahead Adder (CLA)

		c3	c2	c1	c_0
X		x3	x2	x1	x0
Y		y3	y2	y1	y0
S	c_4	S_3	S_2	S_1	S_0

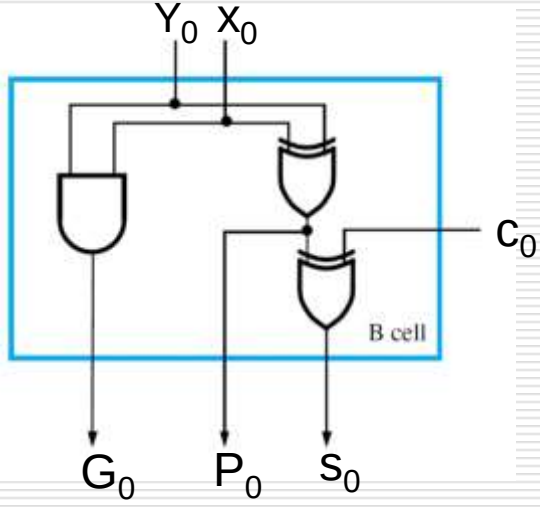


$S_0 = X_0 \oplus Y_0 \oplus C_0$

$C_1 = G_0 + C_0P_0$

Where $G_0 = X_0Y_0$

$P_0 = X_0 \oplus Y_0$



Bit Stage Cell or B-Cell

Designing 4-bit Carry Lookahead Adder (CLA)

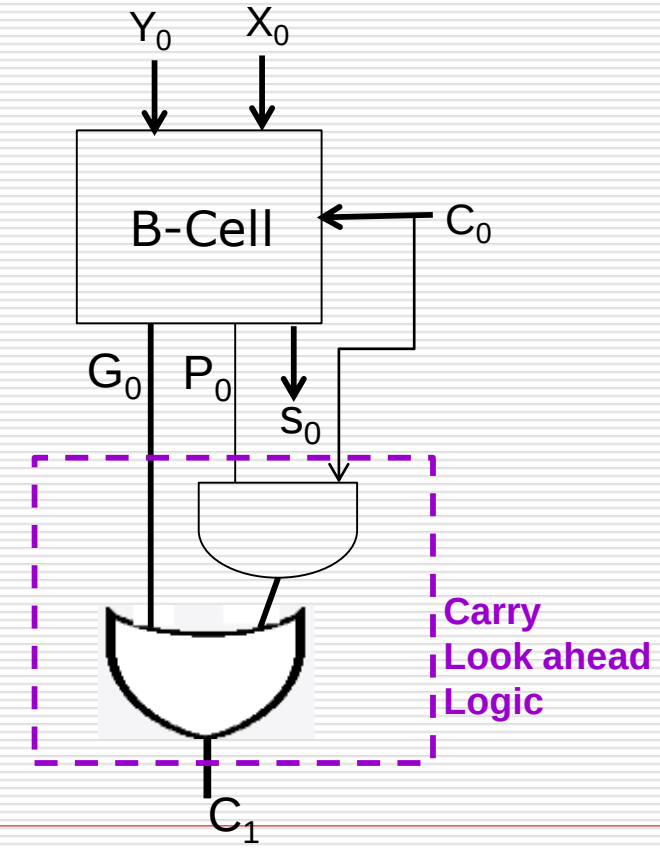
		c3	c2	c1	c_0
X		x3	x2	x1	x0
Y		y3	y2	y1	y0
S	c_4	S_3	S_2	S_1	S_0

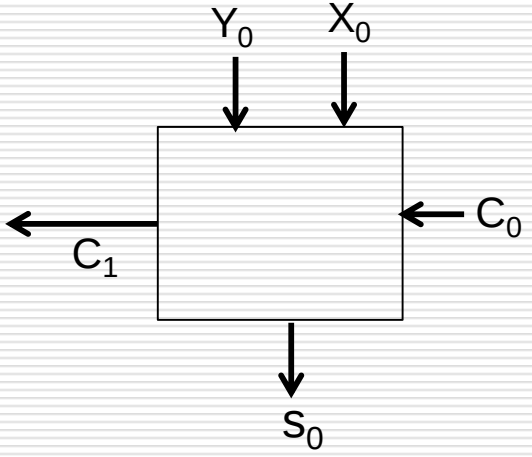
$S_0 = X_0 \oplus Y_0 \oplus C_0$

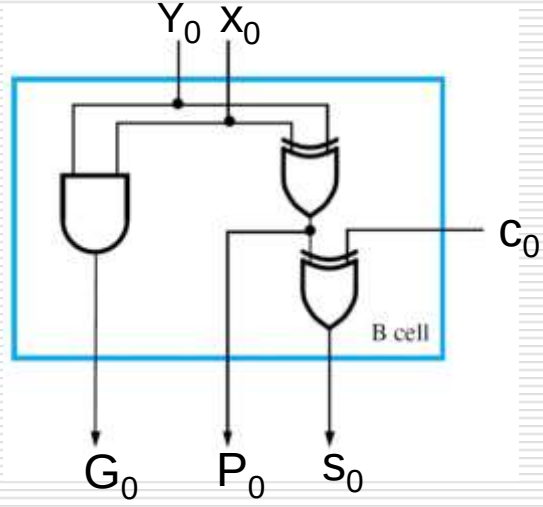
$C_1 = G_0 + C_0P_0$

Where $G_0 = X_0Y_0$

$P_0 = X_0 \oplus Y_0$







Bit Stage Cell or B-Cell

Designing 4-bit Carry Look Ahead (CLA) Adder using four Bit Stage cell or B-Cell

$S_0 = x_0 \oplus Y_0 \oplus C_0 \quad c_1 = G_0 + P_0c_0 \qquad G_0 = X_0Y_0 \quad P_0=X_0 \oplus Y_0$

Designing 4-bit Carry Look Ahead (CLA) Adder using four Bit Stage cell or B-Cell

$S_0 = x_0 \oplus Y_0 \oplus C_0 \quad c_1 = G_0 + P_0c_0 \quad G_0 = X_0Y_0 \quad P_0=X_0 \oplus Y_0$

$S_1 = x_1 \oplus Y_1 \oplus C_1 \quad c_2 = G_1 + P_1C_1 \quad G_1= X_1Y_1 \quad P_1=X_1 \oplus Y_1$

$$c_2 = G_1 + P_1G_0 + P_1P_0c_0$$

Designing 4-bit Carry Look Ahead (CLA) Adder using four Bit Stage cell or B-Cell

$S_0 = x_0 \oplus Y_0 \oplus C_0 \quad c_1 = G_0 + P_0c_0 \quad G_0 = X_0Y_0 \quad P_0=X_0 \oplus Y_0$

$S_1 = x_1 \oplus Y_1 \oplus C_1 \quad c_2 = G_1 + P_1C_1 \quad G_1= X_1Y_1 \quad P_1=X_1 \oplus Y_1$

$$c_2 = G_1 + P_1G_0 + P_1P_0c_0$$

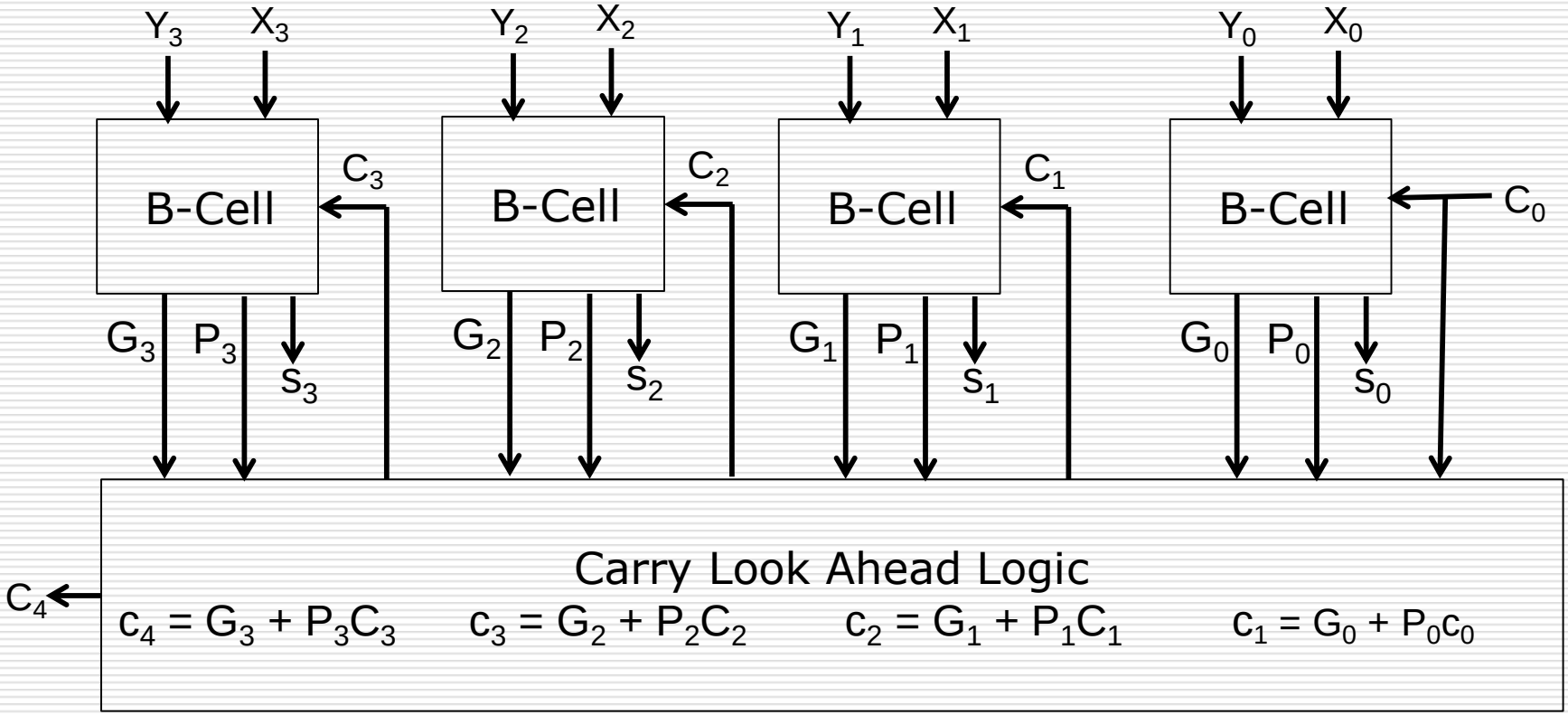
$S_2 = x_2 \oplus Y_2 \oplus C_2 \quad c_3 = G_2 + P_2C_2 \quad G_2= X_2Y_2 \quad P_2=X_2 \oplus Y_2$

$$c_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$$

$S_3 = x_3 \oplus Y_3 \oplus C_3 \quad c_4 = G_3 + P_3C_3 \quad G_3= X_3Y_3 \quad P_3=X_3 \oplus Y_3$

$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

Designing 4-bit Carry Look Ahead (CLA) Adder using four Bit Stage cell or B-Cell (Contd…)



Performance Analysis of CLA

Number of gate Delays for **Carry** Signal Generation

1 + 2 = **3** Gate Delays

G_i P_i Generation

AND , OR of $G_i + P_iC_i$

$c_1 = G_0 + P_0c_0$

$G_0 = X_0Y_0$

$P_0 = X_0 \oplus Y_0$

Number of gate Delays for **Sum** Signal Generation

3 + 1 = **4** Gate Delays

For Carry $S_i = X_i \oplus Y_i \oplus C_i$

P_i 1

Comparison between RCA and CLA

	Number of Gate Delays					
	For 4-bit		For 8-bit		For n-bit	
	S	C	S	C	S	C
RCA	7	8	15	16	2n-1	2n
CLA	4	3	4	3	4	3

S: Sum Signal
C: Carry Signal

⟵ Remains Same

Disadvantages of CLA

1. **For 4-bit Adder**

$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$ We need **Five** Input AND Gate

2. **For 8-bit Adder**

$c_8 = G_7 + P_7G_6 + P_7P_6\,G_5 + P_7P_6\,P_5G_4 + P_7P_6\,P_5P_4G_3 + P_7P_6\,P_5P_4P_3\,G_2$

$+ P_7P_6\,P_5P_4P_3\,P_2G_1 + P_7P_6\,P_5P_4P_3\,P_2P_1G_0 + P_7P_6\,P_5P_4P_3\,P_2P_1P_0c_0$

We need **Nine** Input AND Gate

Performance Analysis of CLA

CLA	Fan-In Required
4-Bit	5
8-bit	9
n-bit	(n+1)

Fan-In: Number of Input's to the Gate

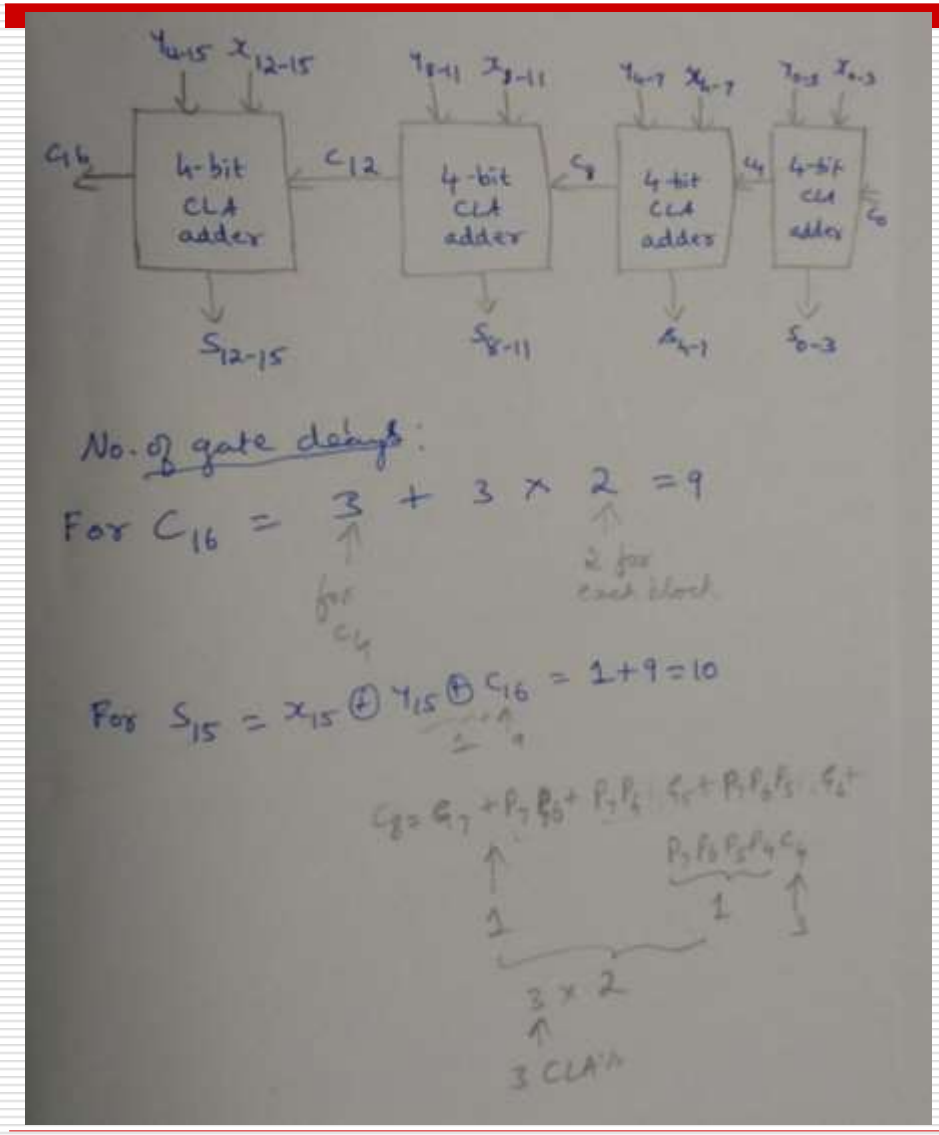
Advantages of CLA

Number of Gate Delays are Less

Disadvantages of CLA

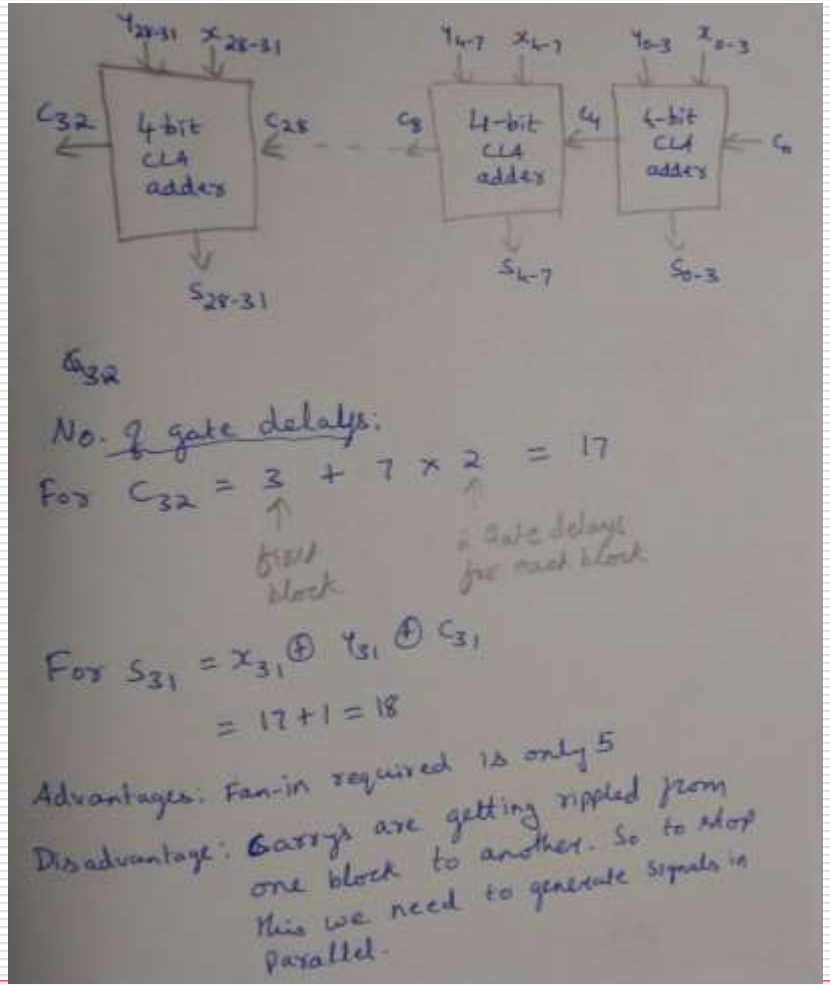
Fan-In required is Very high i.e., (n+1)

Designing 16-bit adder using 4-bit CLA adders



Designing 32-bit adder using 4-bit CLA adders

Cascading eight 4-bit CLA adder to obtain 32-bit adder



Unit-4

Multiplication of Unsigned Numbers, Multiplication of Signed Numbers,

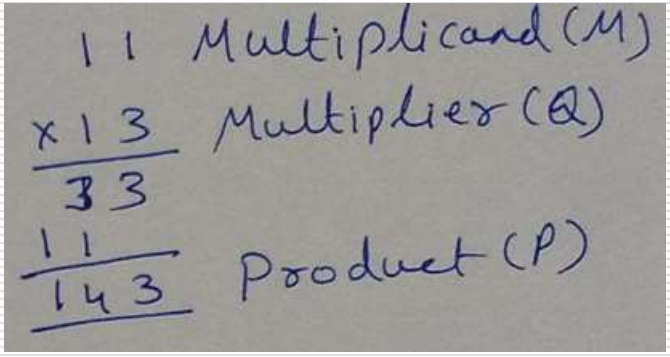
Fast Multiplication, Floating-Point Numbers and Operations

Unit-4

Array Multiplication: Using Full Adders

Multiplication of Unsigned Numbers: Array Multiplier

Example



Multiplication of Unsigned Numbers: Array Multiplier

Example

		1	0	1	1 (11)
	x	1	1	0	1 (13)

Multiplication of Unsigned Numbers: Array Multiplier

Example

	1	0	1	1	(11)
x	1	1	0	1	(13)
	1	0	1	1	

Multiplication of Unsigned Numbers: Array Multiplier

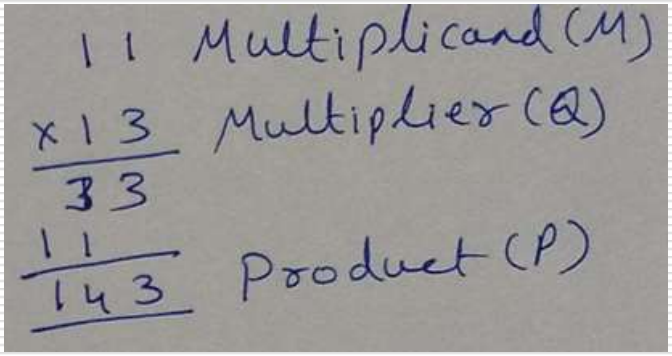
Example

		1	0	1	1 (11)
	x	1	1	0	1 (13)
		1	0	1	1
	0	0	0	0	

Multiplication of Unsigned Numbers: Array Multiplier

Example

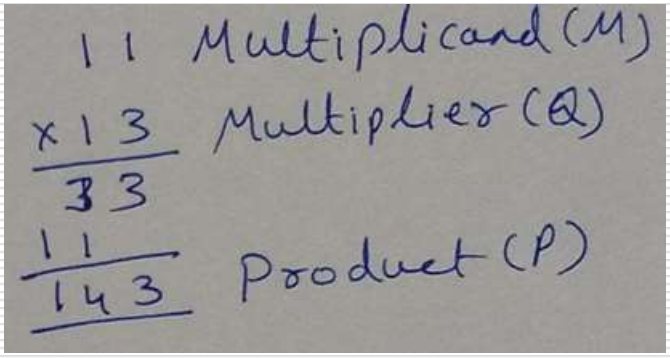
				1	0	1	1	(11)
			x	1	1	0	1	(13)
				1	0	1	1	
			0	0	0	0		
		1	0	1	1			
	1	0	1	1				
1	0	0	0	1	1	1	1	



Multiplication of Unsigned Numbers: Array Multiplier

Example

				1	0	1	1	(11)
			x	1	1	0	1	(13)
				1	0	1	1	
			0	0	0	0		
		1	0	1	1			
	1	0	1	1				
1	0	0	0	1	1	1	1	

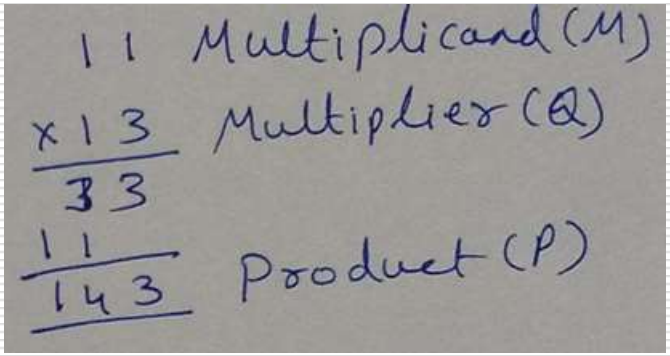


				1	0	1	1	(11)
			x	1	1	0	1	(13)
				1	0	1	1	
			0	0	0	0		
		0	1	0	1	1	⟸	Partial Sum or Partial Product

Multiplication of Unsigned Numbers: Array Multiplier

Example

				1	0	1	1	(11)
			x	1	1	0	1	(13)
				1	0	1	1	
			0	0	0	0		
		1	0	1	1			
	1	0	1	1				
1	0	0	0	1	1	1	1	



				1	0	1	1	(11)
			x	1	1	0	1	(13)
				1	0	1	1	
			0	0	0	0		
			0	1	0	1	1	⟸ Partial Sum or Partial Product
		1	0	1	1			
		1	1	0	1	1	1	
	1	0	1	1				
1	0	0	0	1	1	1	1	

Multiplication of Unsigned Numbers: Array Multiplier

				1	0	1	1	(11)
			x	1	1	0	1	(13)
				1	0	1	1	
			0	0	0	0		
		1	0	1	1			
	1	0	1	1				
1	0	0	0	1	1	1	1	

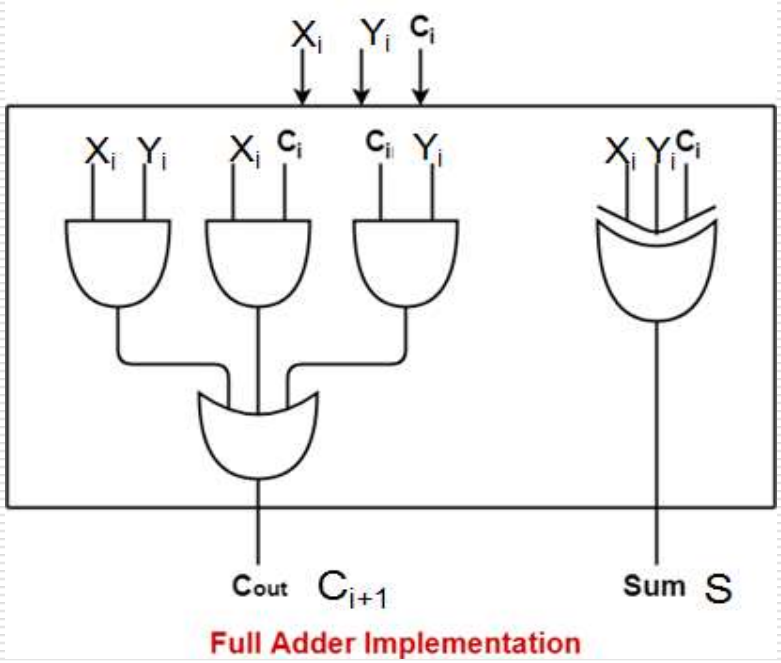
				m_3	m_2	m_1	m_0	Multiplicand (M)
			x	q_3	q_2	q_1	q_0	Multiplier (Q)
				m_3q_0	m_2q_0	m_1q_0	m_0q_0	
			m_3q_1	m_2q_1	m_1q_1	m_0q_1		
		m_3q_2	m_2q_2	m_1q_2	m_0q_2			
	m_3q_3	m_2q_3	m_1q_3	m_0q_3				
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	

Array Multiplication of Unsigned Numbers using Full Adders

				m_3	m_2	m_1	m_0	Multiplicand (M)
			x	q_3	q_2	q_1	q_0	Multiplier (Q)
				m_3q_0	m_2q_0	m_1q_0	m_0q_0	
			m_3q_1	m_2q_1	m_1q_1	m_0q_1		
		m_3q_2	m_2q_2	m_1q_2	m_0q_2			
	m_3q_3	m_2q_3	m_1q_3	m_0q_3				
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	

$$S = X_i \oplus Y_i \oplus C_i$$

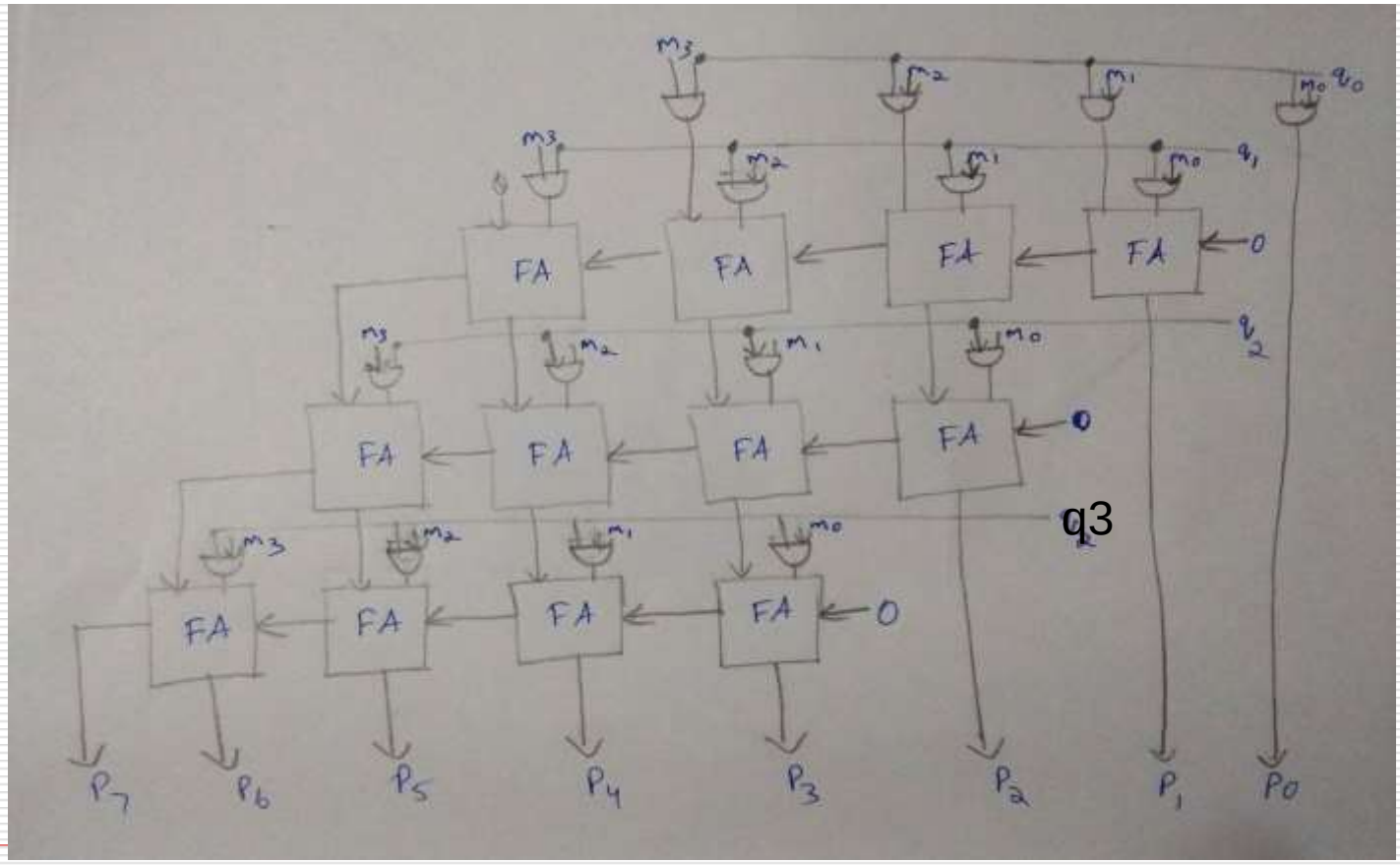
$$C_{i+1} = X_iY_i + X_iC_i + Y_iC_i$$



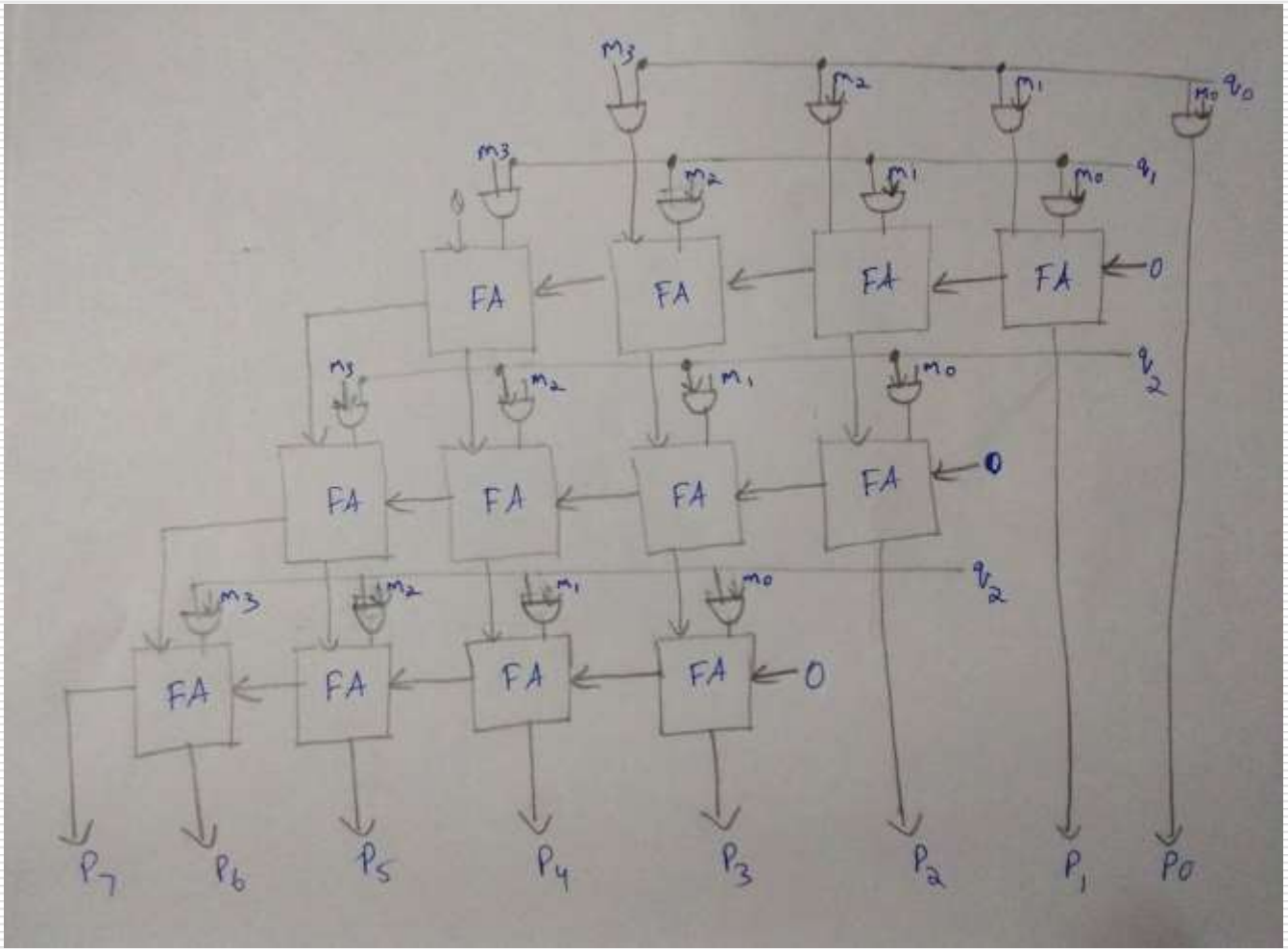
Full Adder Implementation

Array Multiplication of Unsigned Numbers using Full Adders

				m_3	m_2	m_1	m_0	Multiplicand (M)
			x	q_3	q_2	q_1	q_0	Multiplier (Q)
				m_3q_0	m_2q_0	m_1q_0	m_0q_0	
			m_3q_1	m_2q_1	m_1q_1	m_0q_1		
		m_3q_2	m_2q_2	m_1q_2	m_0q_2			
	m_3q_3	m_2q_3	m_1q_3	m_0q_3				
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	

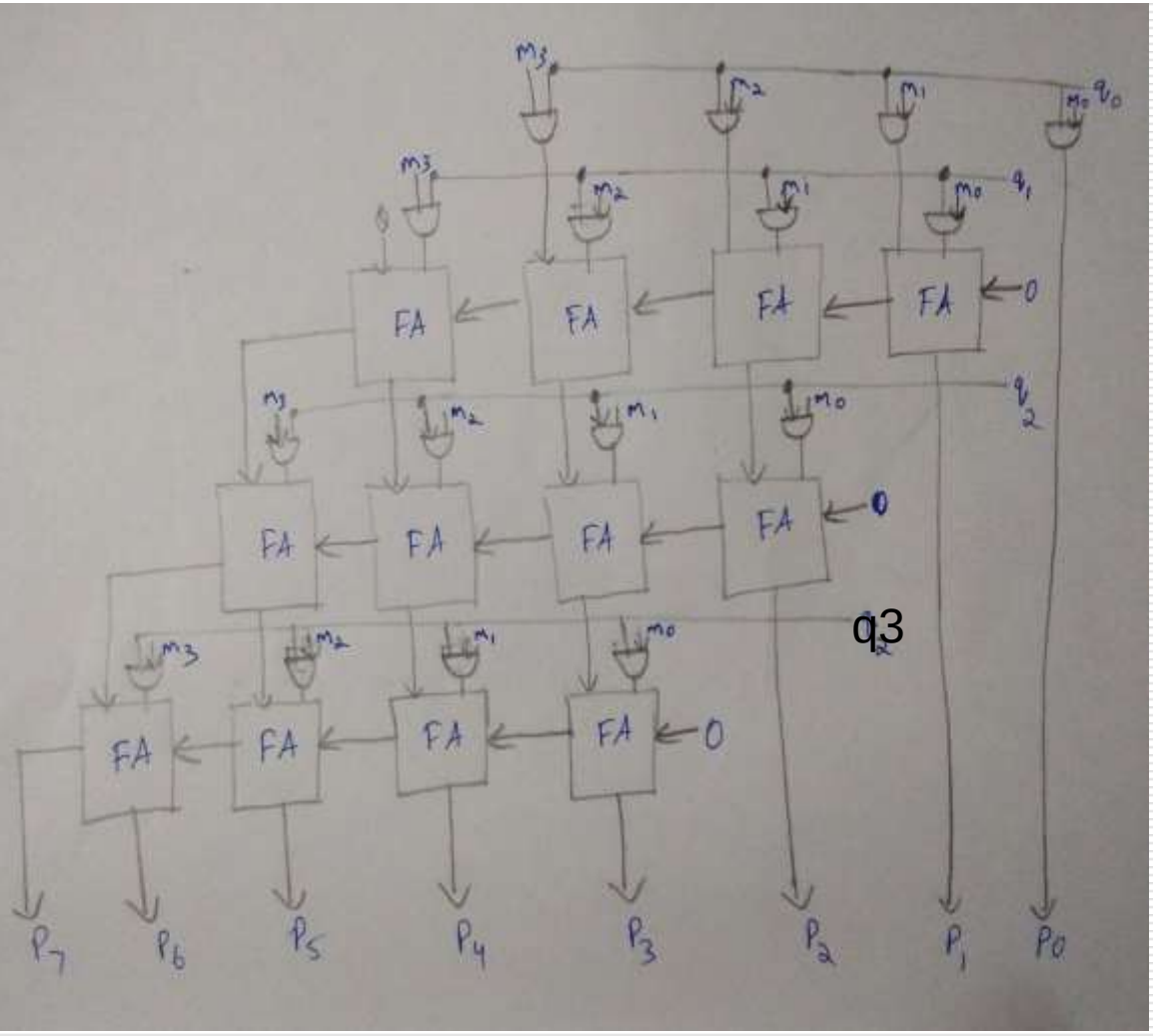


Array Multiplication of Unsigned Numbers using Full Adders

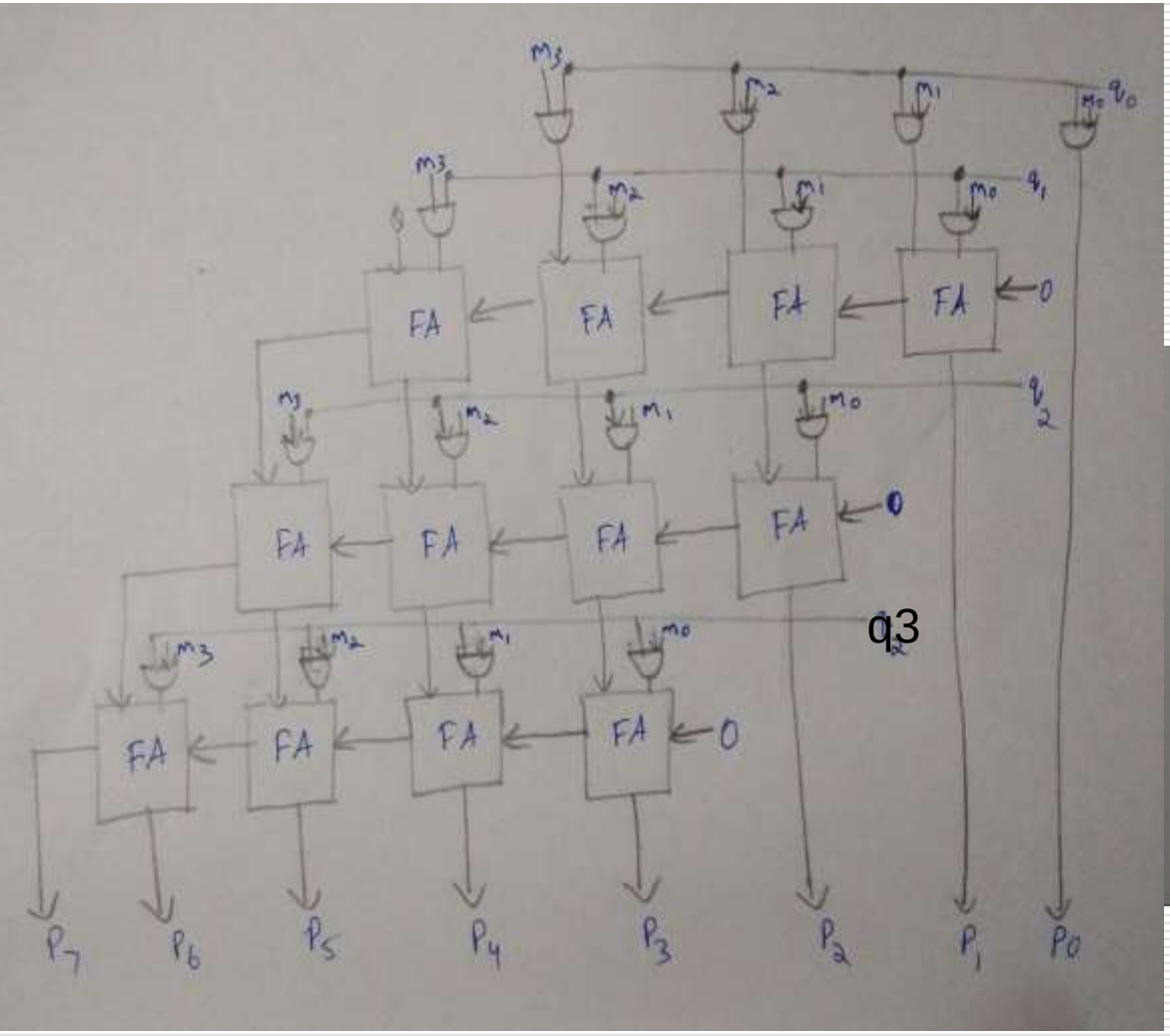


Number of full Adders required is (n-1)n. Hence we can use sequential Multiplication which uses one n-bit adder

Performance of Array Multiplication using Full Adders



Performance of Array Multiplication using Full Adders

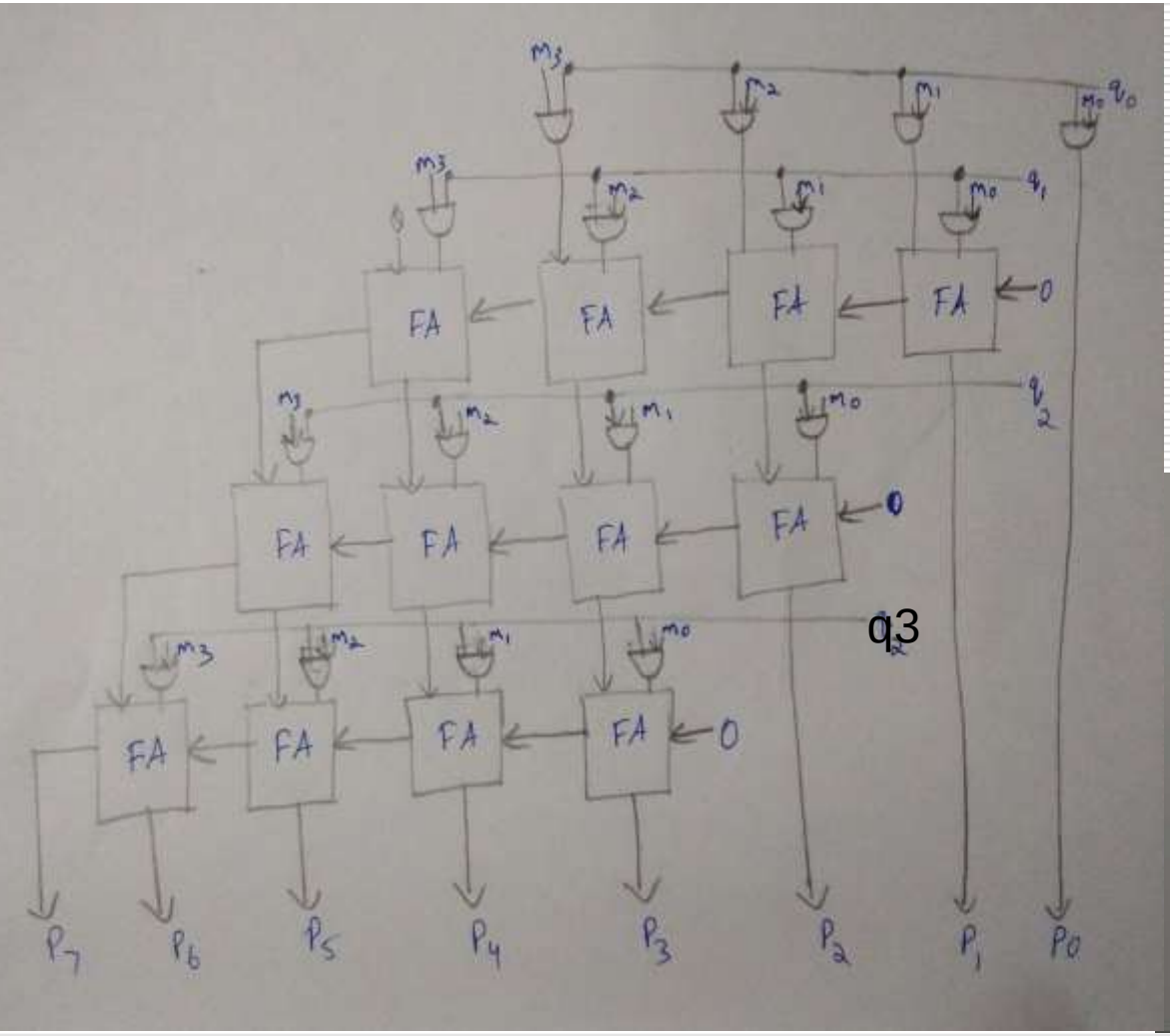


q3

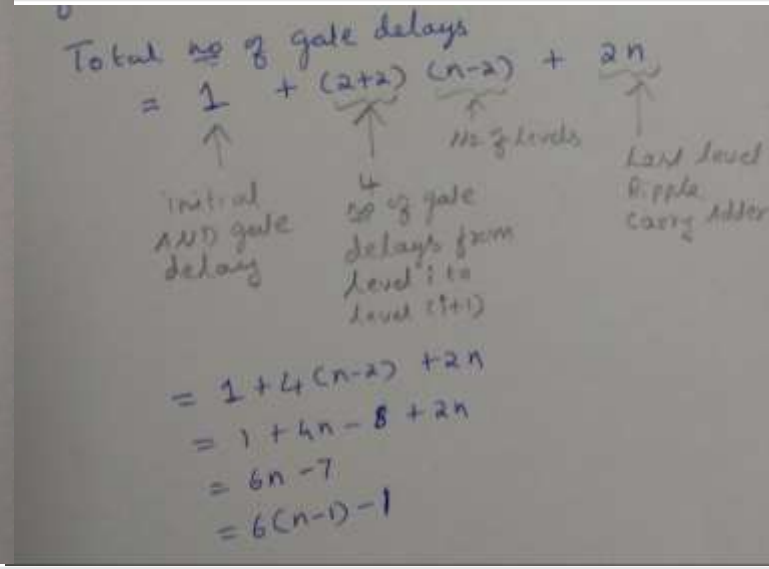
Total no of gate delays
= 1 + (2+2)(n-2) + 2n
1 → Initial AND gate delay
(2+2) → no of gate delays from level i to level (i+1)
(n-2) → no of levels
2n → Last level Ripple Carry Adder

$= 1 + 4(n-2) + 2n$
$= 1 + 4n - 8 + 2n$
$= 6n - 7$
$= 6(n-1) - 1$

Performance of Array Multiplication using Full Adders

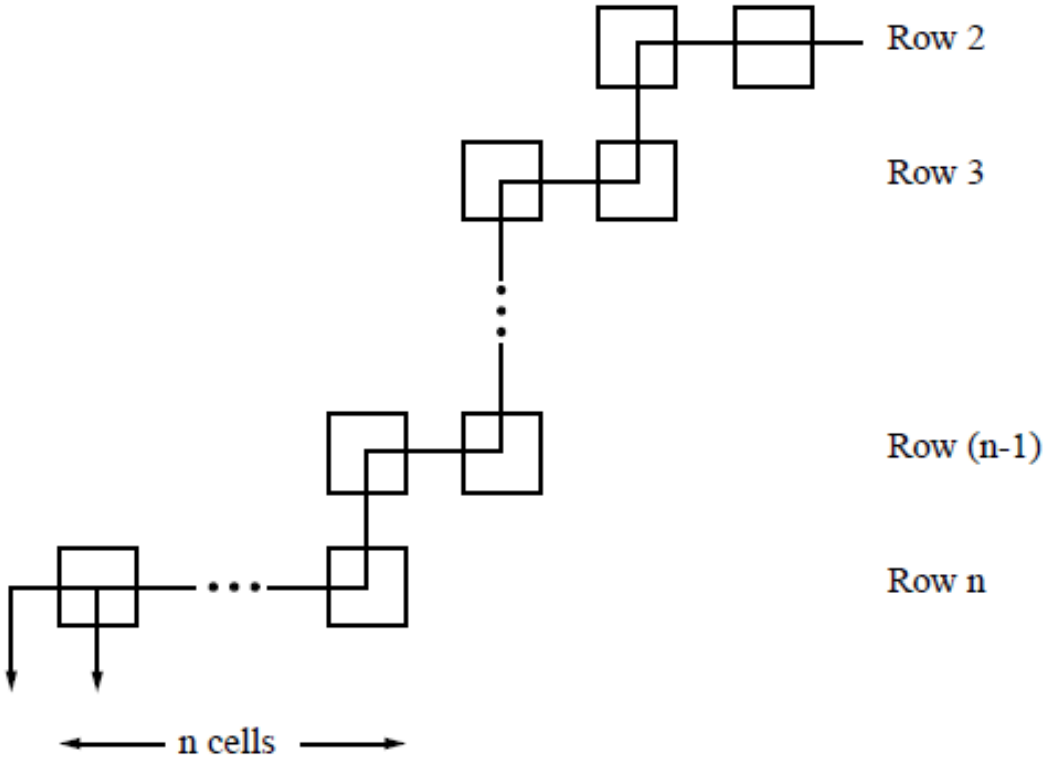


The worst-case signal propagation delay path is from the upper right corner of the array to the high-order product bit output at the bottom left corner of the array. This critical path consists of the staircase pattern that includes the two cells at the right end of each row, followed by all the cells in the bottom row. Assuming that there are two gate delays from the inputs to the outputs of a full-adder block, FA, the critical path has a total of $6(n-1)-1$ gate delays, including the initial AND gate delay in all cells, for an $n \times n$ array.



Performance of Array Multiplication using Full Adders

The worst case delay path is shown in the following figure:



Each of the two FA blocks in rows 2 through $n-1$ introduces 2 gate delays, for a total of $4(n-2)$ gate delays. Row n introduces $2n$ gate delays. Adding in the initial AND gate delay for row 1 and all other cells, total delay is:

$$4(n-2)+2n+1 = 6n-8+1 = 6(n-1)-1$$

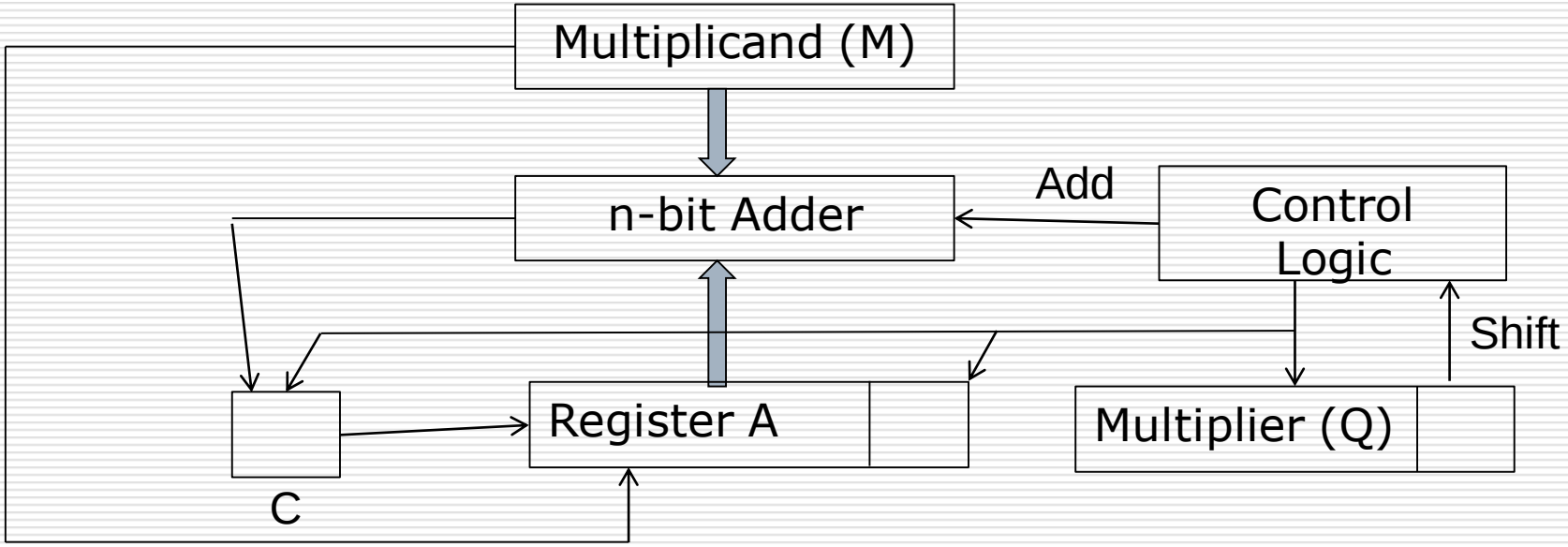
Multiplication of Unsigned Numbers: Sequential Multiplication

Sequential Multiplication

- This is an Add Shift method
- Using One n-bit Adder

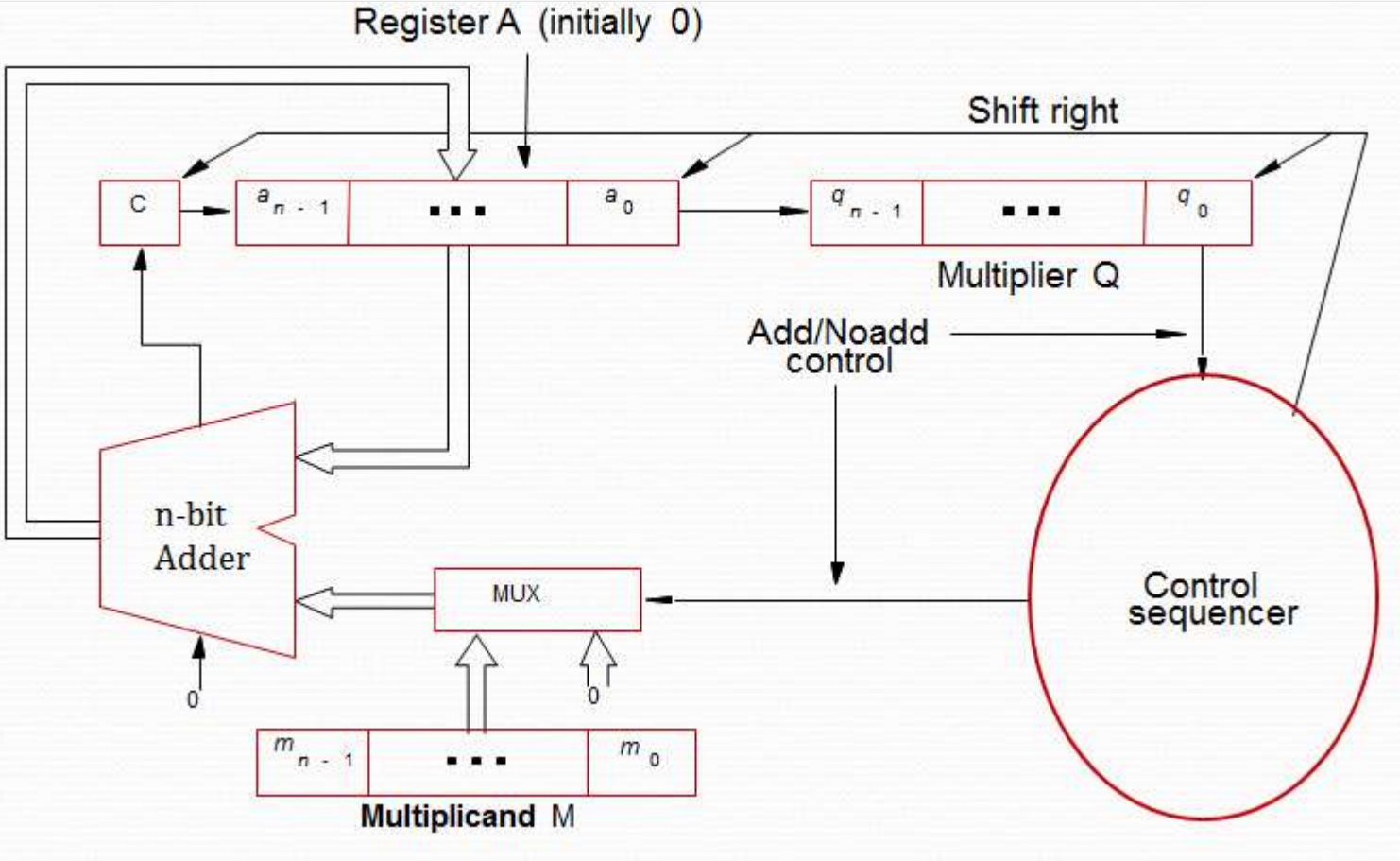
Multiplication of Unsigned Numbers: Sequential Multiplication

Hardware Structure of Add-Shift Method of Multiplication



				1	0	1	1	(11)
			x	1	1	0	1	(13)
				1	0	1	1	
			0	0	0	0		
		1	0	1	1			
	1	0	1	1				
1	0	0	0	1	1	1	1	

Sequential Circuit Multiplier



Sequential Multiplication or Add-Shift Method of Multiplication

If q_0 =1 then {

Add A=A+M

Shift Right 1-bit C → A → Q

}

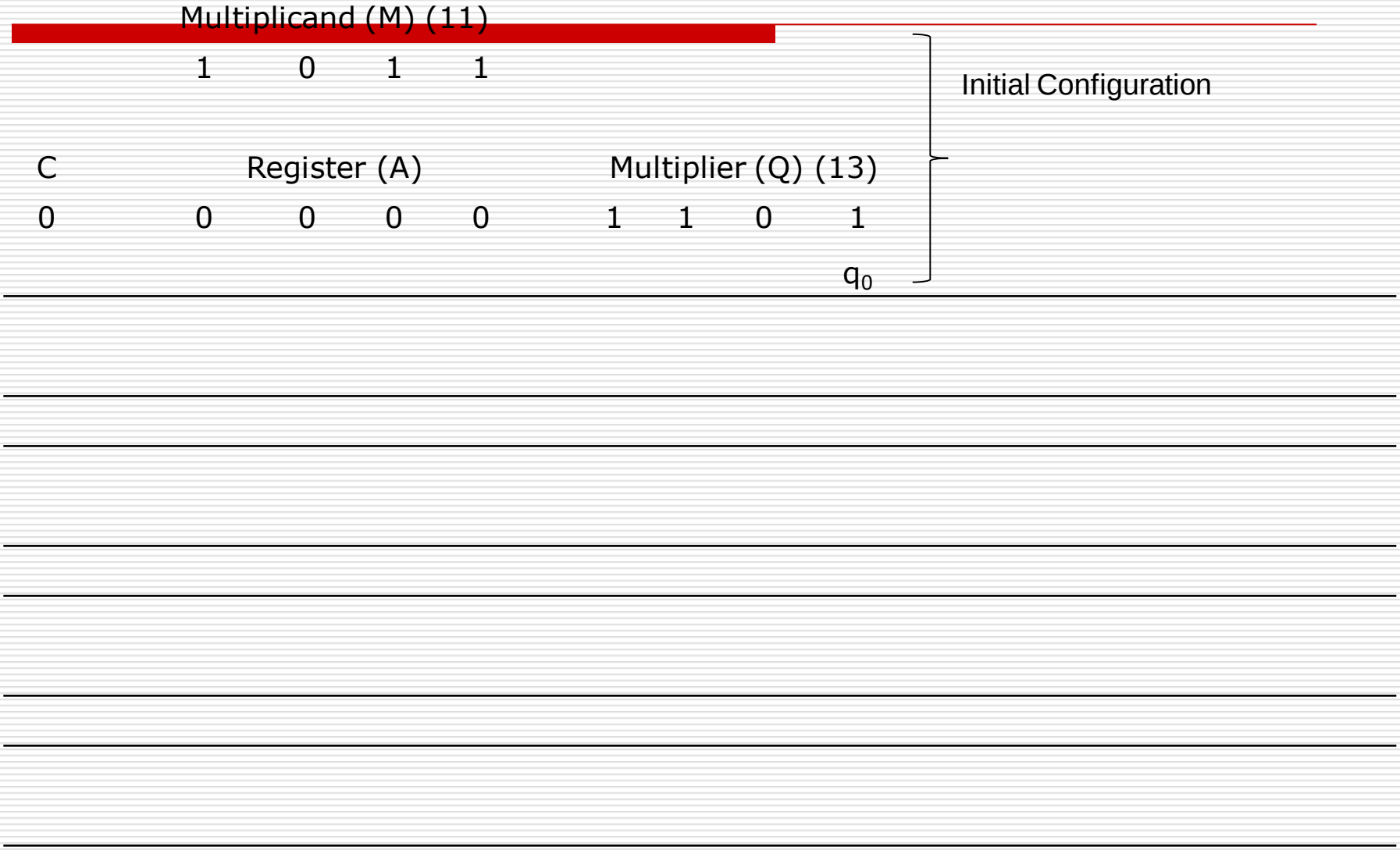
Else {

No Add

Shift Right 1-bit C → A → Q

}

Sequential or Add-Shift method of Multiplication



Sequential or Add-Shift method of Multiplication

Multiplicand (M) (11)

1 0 1 1

C	Register (A)				Multiplier (Q) (13)			
0	0	0	0	0	1	1	0	1
								q_0

Initial Configuration

If q_0 =1 then Add M and A

Sequential or Add-Shift method of Multiplication

Multiplicand (M) (11)

1 0 1 1

Initial Configuration

C	Register (A)				Multiplier (Q) (13)				
0	0	0	0	0	1	1	0	1	If q_0 =1 then Add M and A
								q_0	
0	1	0	1	1	1	1	0	1	Add A=A+M ∵ q_0=1

Sequential or Add-Shift method of Multiplication

Multiplicand (M) (11)

	1	0	1	1

C	Register (A)				Multiplier (Q) (13)				
0	0	0	0	0	1	1	0	1	Initial Configuration
								q_0	If q_0 =1 then Add M and A
0	1	0	1	1	1	1	0	1	Add A=A+M ∵ q_0=1
0	0	1	0	1	1	1	1	0	Shift Right C → A → Q

Sequential or Add-Shift method of Multiplication

	Multiplicand (M) (11)								
	1	0	1	1					Initial Configuration
C	Register (A)				Multiplier (Q) (13)				
0	0	0	0	0	1	1	0	1	If q_0 =1 then Add M and A
								q_0	
0	1	0	1	1	1	1	0	1	Add A=A+M ∵ q_0=1
0	0	1	0	1	1	1	1	0	Shift Right C → A → Q
0	0	1	0	1	1	1	1	0	No Add , Because q0=0
0	0	0	1	0	1	1	1	1	Shift

Sequential or Add-Shift method of Multiplication

	Multiplicand (M) (11)								
	1	0	1	1					
C	Register (A)				Multiplier (Q) (13)				
0	0	0	0	0	1	1	0	1	Initial Configuration
								q_0	If q_0 =1 then Add M and A
0	1	0	1	1	1	1	0	1	Add A=A+M ∵ q_0=1
0	0	1	0	1	1	1	1	0	Shift Right C → A → Q
0	0	1	0	1	1	1	1	0	No Add , Because q0=0
0	0	0	1	0	1	1	1	1	Shift
0	1	1	0	1	1	1	1	1	Add A=A+M ∵ q_0=1
0	0	1	1	0	1	1	1	1	Shift Right C → A → Q

Sequential or Add-Shift method of Multiplication

Multiplicand (M) (11)

		1	0	1	1				
C		Register (A)				Multiplier (Q) (13)			
0		0	0	0	0	1	1	0	1
									q_0

Initial Configuration

If q_0 =1 then Add M and A

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

Add A=A+M ∵ q_0=1

Shift Right C → A → Q

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

No Add , Because q0=0

Shift

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

Add A=A+M ∵ q_0=1

Shift Right C → A → Q

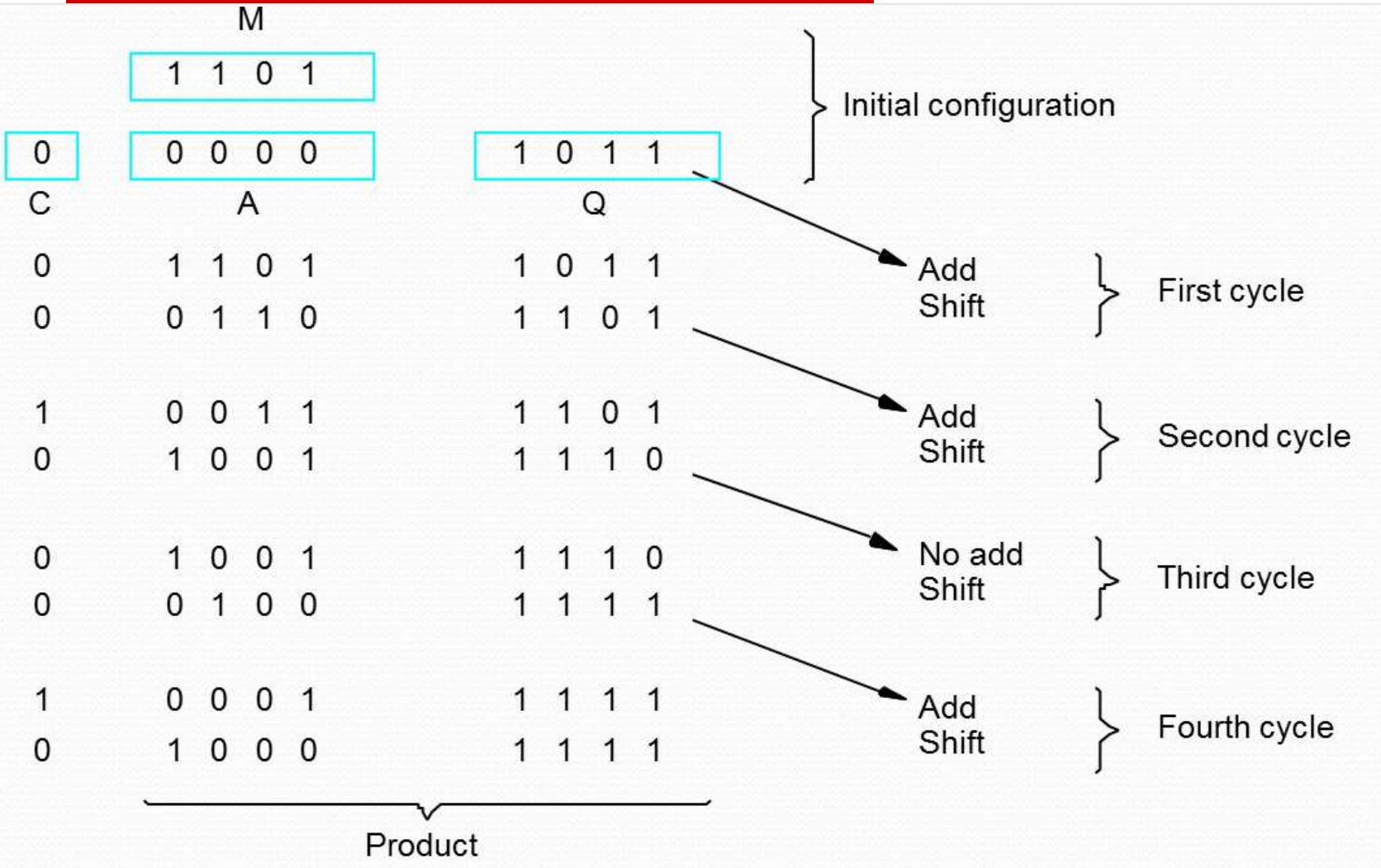
1		0	0	0	1	1	1	1	1
0		1	0	0	0	1	1	1	1

Add A=A+M ∵ q_0=1

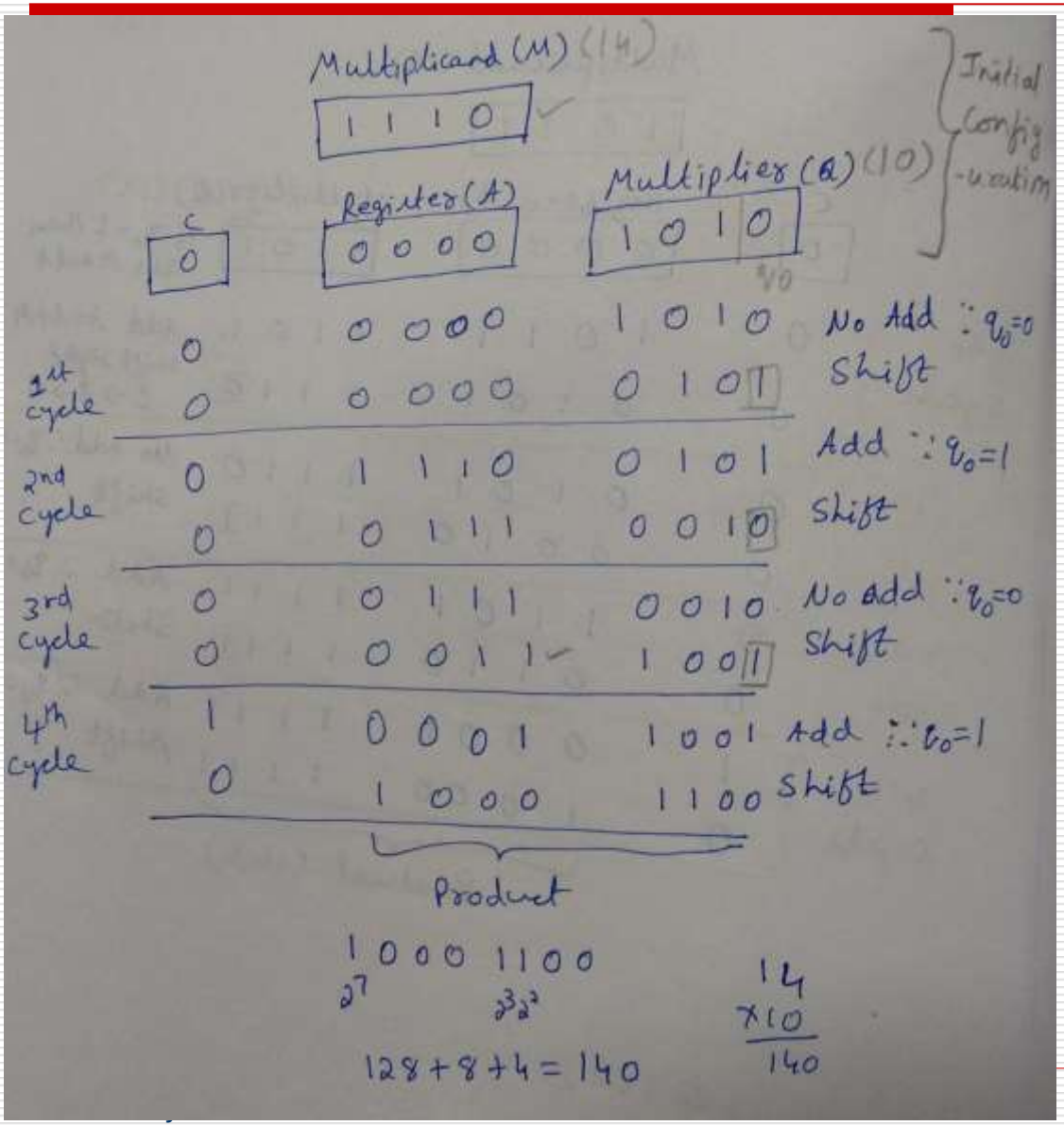
Shift Right C → A → Q

Product

Sequential Multiplication : Add-Shift Method of Multiplication



Sequential or Add-Shift method of Multiplication

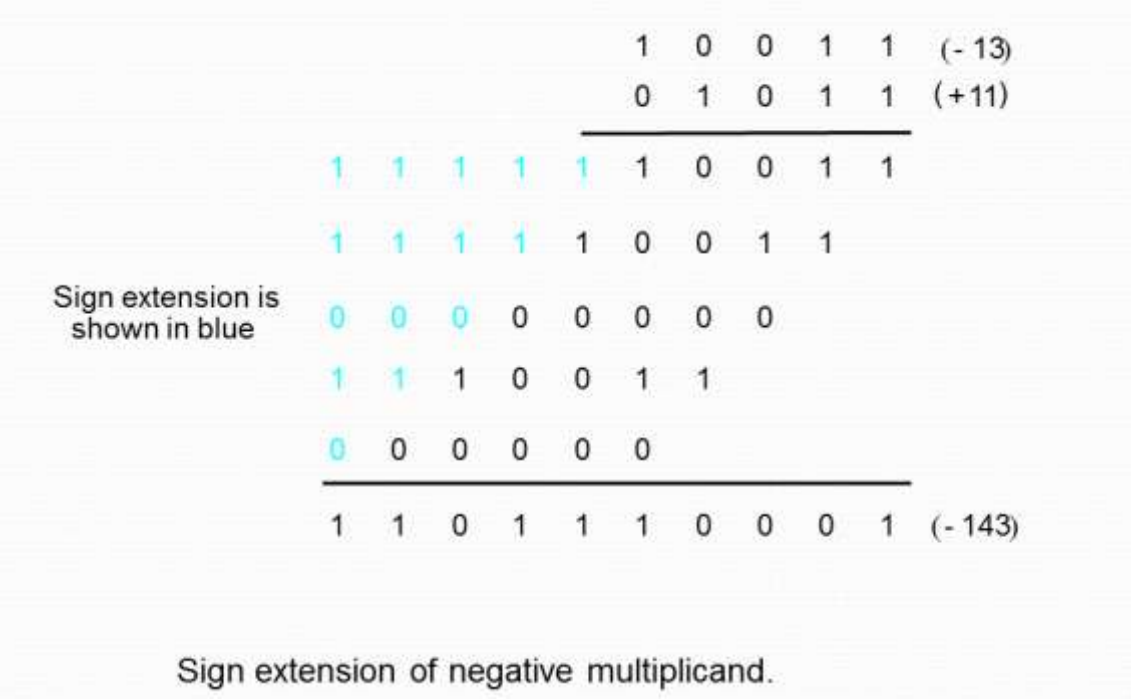


Unit-4

Signed Multiplication

Signed Multiplication

First, consider the case of a positive multiplier and a negative multiplicand. When we add a negative multiplicand to a partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend. Figure shows an example in which a 5-bit signed operand, −13, is the multiplicand. It is multiplied by +11 to get the 10-bit product, −143. The sign extension of the multiplicand is shown in blue. The hardware discussed earlier can be used for negative multiplicands if it is augmented to provide for sign extension of the partial products.



Sign extension of negative multiplicand.

Signed Operands Multiplication: Different cases

Case-1

+ve Multiplicand
+ve Multiplier

```
   +2
x  +1
  ----
   +2
```

```
                0   1   0    +2
            x   0   0   1    +1
          -------------------
                0   1   0
            0   0   0
        0   0   0
    ---------------------------
0   0   0   0   1   0    +2
```

Signed Operands Multiplication: Different cases

Case-1

+ve Multiplicand
+ve Multiplier

```
   +2
x  +1
 ----
   +2
```

```
                0   1   0   +2
            x   0   0   1   +1
          -------------------
                0   1   0
            0   0   0
        0   0   0
    ---------------------------
    0   0   0   0   1   0   +2
```

Case-2

-ve Multiplicand
+ve Multiplier

```
   -2
x  +1
 ----
   -2
```

2's complement of +2

```
   +2   010
        101
          1
        -----
        110
```

```
                1   1   0   -2
            x   0   0   1   +1
          -------------------
    1   1   1   1   1   0
    0   0   0   0   0
    0   0   0   0
    ---------------------------
    1   1   1   1   1   0   -2
```

Signed Operands Multiplication: Different cases

Case-1	Case-2	Case-3	
+ve Multiplicand +ve Multiplier	-ve Multiplicand +ve Multiplier	+ve Mud -ve Mur	

Case-1

```
   +2
x  +1
  ----
   +2
```

```
                0   1   0   +2
            x   0   0   1   +1
          ---------------
                0   1   0
            0   0   0
        0   0   0
    ---------------------
    0   0   0   0   1   0   +2
```

Case-2

```
   -2
x  +1
  ----
   -2
```

2's complement of +2

```
   +2   010
        101
          1
       -----
        110
```

```
                1   1   0   -2
            x   0   0   1   +1
          ---------------
    1   1   1   1   1   0
    0   0   0   0   0
    0   0   0   0
    ---------------------
    1   1   1   1   1   0   -2
```

Case-3

```
   +2
x  -1
  ----
   -2
```

Use Case-2

```
   -2
x  +1
  ----
   -2
```

Signed Operands Multiplication: Different cases

Case-1	Case-2	Case-3	Case-4
+ve Multiplicand +ve Multiplier	-ve Multiplicand +ve Multiplier	+ve Mud -ve Mur	-ve Mud -ve Mur
+2 x +1 ---- +2 0 1 0 +2 x 0 0 1 +1 --------- 0 1 0 0 0 0 0 0 0 --------- 0 0 0 0 1 0 +2	-2 x +1 ---- -2 2's complement of +2 +2 010 101 1 ----- 110 1 1 0 -2 x 0 0 1 +1 --------- 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 --------- 1 1 1 1 1 0 -2	+2 x -1 ---- -2 **Use Case-2** -2 x +1 ---- -2	-2 x -1 ---- +2 **Use Case-1** 2 x 1 --- 2

Booths Algorithm for Signed Operand Multiplication

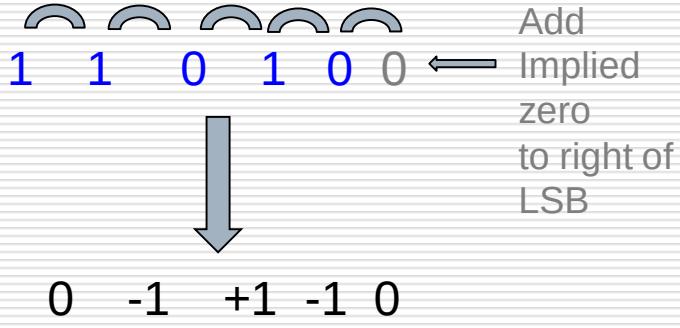
1. It reduces the number of additions
2. Negative Multiplier and Positive multiplier are treated as same

Booth Multiplier Recoding Table

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

Example of
Booth Multiplier Recording



Booths Algorithm: Example

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

+2

X +3

+6

Multiplicand	0	0	1	0
Multiplier	0	0	1	1

Booths Algorithm: Example

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

+2	Multiplicand	0	0	1	0	
X +3	Multiplier	0	0	1	1	0 ⟸ Add Implied zero to right of LSB
+6	Booth Recoded Multiplier	0	+1	0	-1	

Booths Algorithm: Example

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

+2	Multiplicand	0	0	1	0		
X +3	Multiplier	0	0	1	1	0	⇐ Add Implied zero to right of LSB
+6	Booth Recoded Multiplier	0	+1	0	-1		

	Multiplicand	0	0	1	0		
	Recoded Multiplier	0	+1	0	-1		
1	1	1	1	1	1	1	0 ⇐ 2's Complement Of Multiplicand

2's Complement of +2

+2 0010

1101

1

1110

Booths Algorithm: Example

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

+2	Multiplicand	0	0	1	0		
X +3	Multiplier	0	0	1	1	0	⇐ Add Implied zero to right of LSB
+6	Booth Recoded Multiplier	0	+1	0	-1		

	Multiplicand	0	0	1	0			
	Recoded Multiplier	0	+1	0	-1			
2's Complement of +2		1	1	1	1	1	1	0 ⇐ 2's Complement Of Multiplicand

+2 0010
1101
1

1110

0	0	0	0	0	0	0

Booths Algorithm: Example

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

+2	Multiplicand	0	0	1	0		
X +3	Multiplier	0	0	1	1	0	⇐ Add Implied zero to right of LSB
+6	Booth Recoded Multiplier	0	+1	0	-1		

	Multiplicand	0	0	1	0			
	Recoded Multiplier	0	+1	0	-1			
2's Complement of +2	1	1	1	1	1	1	1	0 ⇐ 2's Complement Of Multiplicand
+2 0010	0	0	0	0	0	0	0	
1101								
1	0	0	0	0	1	0		

1110								

Booths Algorithm: Example

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

+2	Multiplicand	0	0	1	0		
X +3	Multiplier	0	0	1	1	0	⇐ Add Implied zero to right of LSB
+6	Booth Recoded Multiplier	0	+1	0	-1		

	Multiplicand	0	0	1	0			
	Recoded Multiplier	0	+1	0	-1			
1	1	1	1	1	1	1	0	⇐ 2's Complement Of Multiplicand
0	0	0	0	0	0	0		
0	0	0	0	1	0			
0	0	0	0	0				
0	0	0	0	0	1	1	0	+6

2's Complement of +2

+2 0010

1101

1

1110

Booths Algorithm: Example

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

-2
X +3

-6

Multiplicand	1	1	1	0
Multiplier	0	0	1	1

Booths Algorithm: Example

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

-2 Multiplicand 1 1 1 0

X +3 Multiplier 0 0 1 1 0 ⇐ Add Implied zero to right of LSB

-6 Booth Recoded Multiplier 0 +1 0 -1

Multiplicand 1 1 1 0

Recoded Multiplier 0 +1 0 -1

2's Complement of +2

+2 0010

1101

1

1110

2's Complement of +6

+6 0110

1001

1

1010

0	0	0	0	0	0	1	0	⇐ 2's Complement Of Multiplicand
0	0	0	0	0	0	0		
1	1	1	1	1	0			
0	0	0	0	0				
1	1	1	1	1	0	1	0	-6

Booths Algorithm: Example

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

+2 Multiplicand 0 0 1 0

X -3 Multiplier 1 1 0 1 0 ⇐ Add Implied zero to right of LSB

-6 Booth Recoded Multiplier 0 -1 +1 -1

Multiplicand 0 0 1 0

Recoded Multiplier 0 -1 +1 -1

2's Complement of +3

+3 0011

1100

1

1101

2's Complement of +6

+6 0110

1001

1

1010

1	1	1	1	1	1	1	0	⇐ 2's Complement Of Multiplicand
0	0	0	0	0	1	0		
1	1	1	1	1	0			
0	0	0	0	0				
1	1	1	1	1	0	1	0	-6

Booths Algorithm: Example

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

-2 Multiplicand 1 1 1 0

X -3 Multiplier 1 1 0 1 0 ⟸ Add Implied zero to right of LSB

+6 Booth Recoded Multiplier 0 -1 +1 -1

Multiplicand 1 1 1 0

Recoded Multiplier 0 -1 +1 -1

0	0	0	0	0	0	1	0	⟸ 2's Complement Of Multiplicand
1	1	1	1	1	1	0		
0	0	0	0	1	0			
0	0	0	0	0				
0	0	0	0	0	1	1	0	+6

2's Complement of +2

+2 0010
1101
1

1110

2's Complement of +3

+3 0011
1100
1

1101

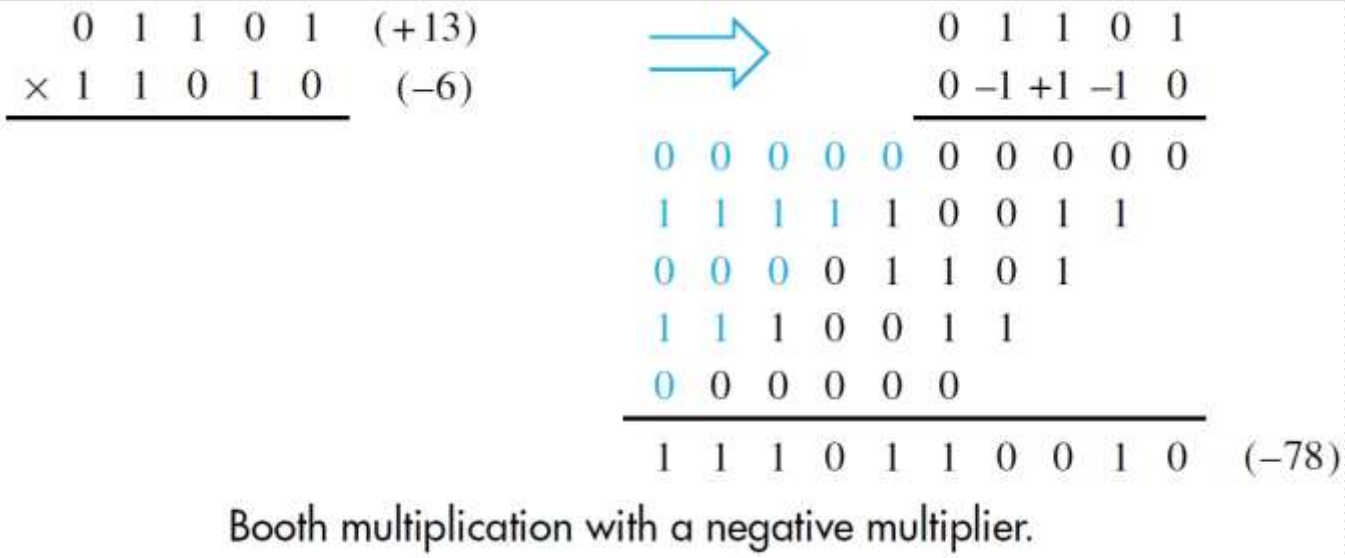
Question

Assuming 5-bit 2's-complement number representation, multiply the multiplicand A = +13 by the multiplier B = -6 using the normal Booth algorithm

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

Answer



Booth multiplication with a negative multiplier.

Booth Multiplier Recoding Table

Multiplier		
Bit i	Bit i-1	
0	0	0
0	1	+1
1	0	-1
1	1	0

Question

Multiply each of the following pairs of signed 2's-complement numbers using the Booth algorithm. In each case, assume that A is the multiplicand and B is the multiplier.

(a) A = 010111 and B = 110110

(b) A = 110011 and B = 101100

(c) A = 001111 and B = 001111

Answer

Multiply each of the following pairs of signed 2's-complement numbers using the Booth algorithm. In each case, assume that A is the multiplicand and B is the multiplier.

(a) A = 010111 and B = 110110

(b) A = 110011 and B = 101100

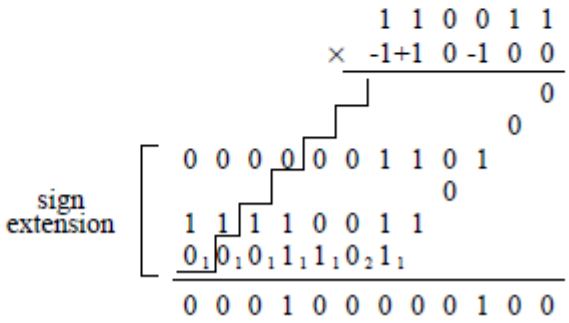
(c) A = 001111 and B = 001111

(a)

010111 × 110110 = -230 (+23 × -10)

(b)

110011 × 101100 = 260 (-13 × -20)



Answer

Multiply each of the following pairs of signed 2's-complement numbers using the Booth algorithm. In each case, assume that A is the multiplicand and B is the multiplier.

(a) A = 010111 and B = 110110

(b) A = 110011 and B = 101100

(c) A = 001111 and B = 001111

(c)

```
  110101          -11
 ×011011         × 27
 ------          ----
                 -297
```

```
                                          1  1  0  1  0  1
                                       ×  +1 0 -1+1 0 -1
                                       ------------------
 0  0  0  0  0  0 | 0  0  0  1  0  1  1
                                                   0
 1  1  1  1 | 1  1  0  1  0  1
 0  0  0 | 0  0  1  0  1  1
                                                0
 1  1  1  1  0  1  1  0  1  1     1
 ---------------------------------------------------
 1  1  1  0  1  1  0  1  0  1  1  1
```

sign extension

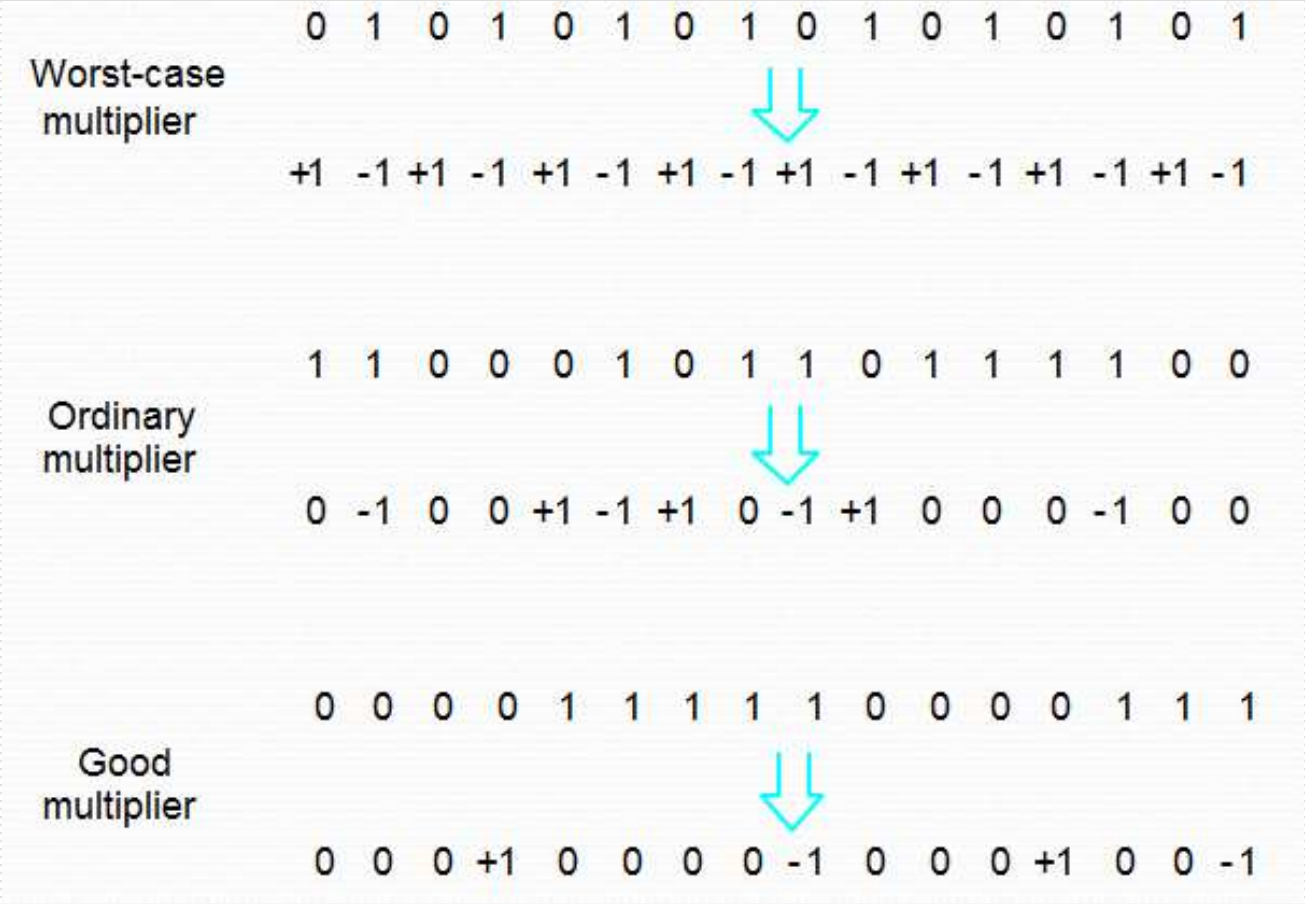
(d)

```
  001111          15
 ×001111         × 15
 ------          ----
                 225
```

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

Booth Algorithm

- Best case: a long string of 1's (skipping over 1s)
- Worst case: 0's and 1's are alternating



Fast Multiplication

- Bit-Pair Recoding
- Carry-Save Addition
- Summand Addition tree using 3-2 Reducers

Bit-Pair Recoding of Multipliers

Multiplier Bit Pair		Multiplier Bit on the right	Multiplicand selected at position i	Remarks
(i-1)	i	(i-1)		
0	0	0	**0**	No Addition required only shift Partial Product (PP)
0	0	1	**+1**	Add Multiplicand to PP
0	1	0	**+1**	Add Multiplicand to PP
0	1	1	**+2**	Add two times Multiplicand and then add to PP
1	0	0	**-2**	Add two times 2's complement of Multiplicand and then add to PP
1	0	1	**-1**	Add 2's Complement of Multiplicand to PP
1	1	0	**-1**	Add 2's Complement of Multiplicand to PP
1	1	1	**0**	No Addition required only shift PP

Bit-Pair Recoding of Multipliers

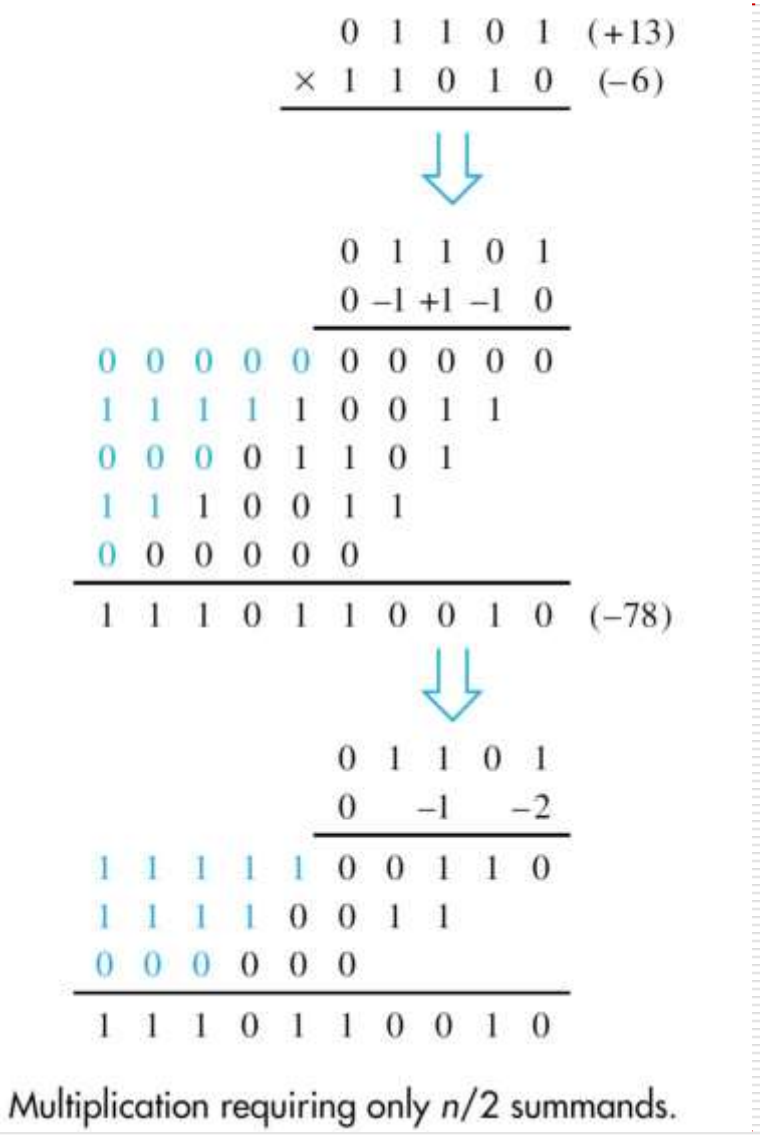
Bit-pair recoding is the product of the multiplier results in using at most one summand for each pair of bits in the multiplier. It is derived directly from the Booth algorithm. Grouping the Booth-recoded multiplier bits in pairs will decrease the multiplication only by $n/2$ summands.

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

Multiplier Bit Pair		Multiplier Bit on the right	Multiplicand selected at position i	Remarks
(i-1)	i	(i-1)		
0	0	0	**0**	No Addition required only shift Partial Product (PP)
0	0	1	**+1**	Add Multiplicand to PP
0	1	0	**+1**	Add Multiplicand to PP
0	1	1	**+2**	Add two times Multiplicand and then add to PP
1	0	0	**-2**	Add two times 2's complement of Multiplicand and then add to PP
1	0	1	**-1**	Add 2's Complement of Multiplicand to PP
1	1	0	**-1**	Add 2's Complement of Multiplicand to PP
1	1	1	**0**	No Addition required only shift PP



Multiplication requiring only $n/2$ summands.

Rough Slide to explain Bit-Pair Recoding of Multipliers

Multiplier Bit Pair		Multiplier Bit on the right	Multiplicand selected at position i	Remarks
(i-1)	i	(i-1)		
0	0	0	0	No Addition required only shift Partial Product (PP)
0	0	1	+1	Add Multiplicand to PP
0	1	0	+1	Add Multiplicand to PP
0	1	1	+2	Add two times Multiplicand and then add to PP
1	0	0	-2	Add two times 2's complement of Multiplicand and then add to PP
1	0	1	-1	Add 2's Complement of Multiplicand to PP
1	1	0	-1	Add 2's Complement of Multiplicand to PP
1	1	1	0	No Addition required only shift PP

+13 Multiplicand 0 1 1 0 1

X -6 Multiplier 1 1 0 1 0

-78

2's Complement of +13

+13 01101

10010

1

10011

2's Complement of +6

+6 00110

11001

1

11010

Rough Slide to explain Bit-Pair Recoding of Multipliers

Multiplier Bit Pair (i-1)	i	Multiplier Bit on the right (i-1)	Multiplicand selected at position i	Remarks
0	0	0	0	No Addition required only shift Partial Product (PP)
0	0	1	+1	Add Multiplicand to PP
0	1	0	+1	Add Multiplicand to PP
0	1	1	+2	Add two times Multiplicand and then add to PP
1	0	0	-2	Add two times 2's complement of Multiplicand and then add to PP
1	0	1	-1	Add 2's Complement of Multiplicand to PP
1	1	0	-1	Add 2's Complement of Multiplicand to PP
1	1	1	0	No Addition required only shift PP

+13	Multiplicand		0	1	1	0	1	
X -6	Multiplier	1	1	1	0	1	0	0
-78	Booth Recoded Multiplier		0		-1		-2	

2's Complement of +13

```
+13  01101
     10010
           1
      --------
     10011
```

2's Complement of +6

```
+6   00110
     11001
           1
     ---------
     11010
```

Rough Slide to explain Bit-Pair Recoding of Multipliers

Multiplier Bit Pair (i-1)	Multiplier Bit Pair i	Multiplier Bit on the right (i-1)	Multiplicand selected at position i	Remarks
0	0	0	0	No Addition required only shift Partial Product (PP)
0	0	1	+1	Add Multiplicand to PP
0	1	0	+1	Add Multiplicand to PP
0	1	1	+2	Add two times Multiplicand and then add to PP
1	0	0	-2	Add two times 2's complement of Multiplicand and then add to PP
1	0	1	-1	Add 2's Complement of Multiplicand to PP
1	1	0	-1	Add 2's Complement of Multiplicand to PP
1	1	1	0	No Addition required only shift PP

	Multiplicand		0	1	1	0	1		
+13									
X -6	Multiplier	1	1	1	0	1	0	0	
-78	Booth Recoded Multiplier		0		-1		-2		
	Multiplicand		0	1	1	0	1		
	Recoded Multiplier		0		-1		-2		
1	1	1	1	1	0	0	1	1	0

2's Complement of +13

+13 01101

10010

1

10011

2's Complement of +6

+6 00110

11001

1

11010

Rough Slide to explain Bit-Pair Recoding of Multipliers

Multiplier Bit Pair (i-1)	i	Multiplier Bit on the right (i-1)	Multiplicand selected at position i	Remarks
0	0	0	0	No Addition required only shift Partial Product (PP)
0	0	1	+1	Add Multiplicand to PP
0	1	0	+1	Add Multiplicand to PP
0	1	1	+2	Add two times Multiplicand and then add to PP
1	0	0	-2	Add two times 2's complement of Multiplicand and then add to PP
1	0	1	-1	Add 2's Complement of Multiplicand to PP
1	1	0	-1	Add 2's Complement of Multiplicand to PP
1	1	1	0	No Addition required only shift PP

2's Complement of +13
+13 01101
10010
1

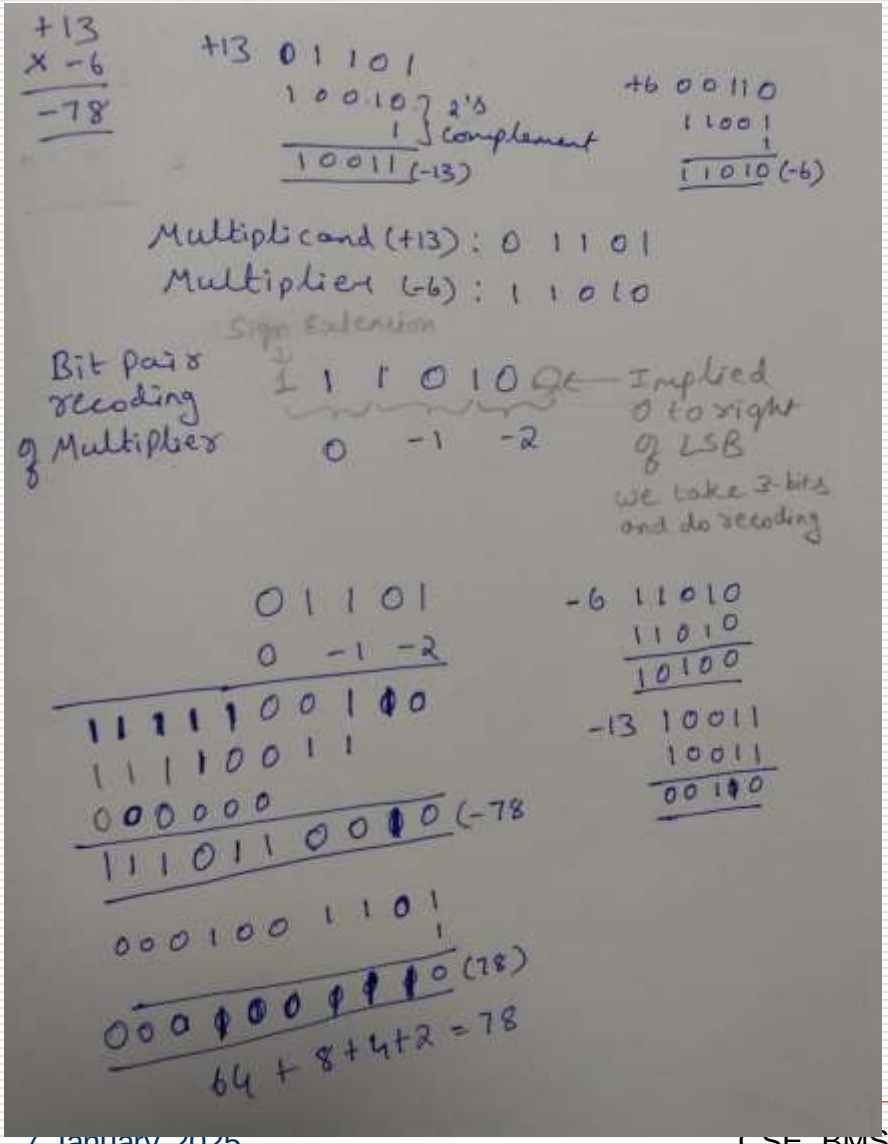
10011

2's Complement of +6
+6 00110
11001
1

11010

				Multiplicand		0	1	1	0	1	
			+13								
			X -6	Multiplier	1	1	1	0	1	0	0
			-78	Booth Recoded Multiplier		0		-1		-2	
				Multiplicand		0	1	1	0	1	
				Recoded Multiplier		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		-78

Example: Bit-Pair Recoding of Multipliers



Question

Assuming 6-bit 2's-complement number representation, multiply the multiplicand A = -13 by the multiplier B = -20 using the Booth Bit-Pair Recoding algorithm

Booth Bit Pair Recoding Table

Multiplier Bit Pair		Multiplier Bit on the right	Multiplicand selected at position i	Remarks
(i-1)	i	(i-1)		
0	0	0	0	No Addition required only shift Partial Product (PP)
0	0	1	+1	Add Multiplicand to PP
0	1	0	+1	Add Multiplicand to PP
0	1	1	+2	Add two times Multiplicand and then add to PP
1	0	0	-2	Add two times 2's complement of Multiplicand and then add to PP
1	0	1	-1	Add 2's Complement of Multiplicand to PP
1	1	0	-1	Add 2's Complement of Multiplicand to PP
1	1	1	0	No Addition required only shift PP

Booth Bit-Pair Recoding Multiplication : Example

$A = 110011 \ (-13 \rightarrow \text{2's compliment of } 110011)$

$B = 101100 \ (-20 \rightarrow \text{2's compliment of } 101100)$

Multiply the signed 2's complement numbers using the bit-pair recoding of the multiplier.

$$\begin{array}{rl}
1\ 1\ 0\ 0\ 1\ 1 & -13 \\
\times\quad -1\ \ -1 & \times -20 \ (\text{Booth recoding } -1\ +1\ 0\ -1\ 0\ 0) \\
\hline
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 & \\
0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1 & \text{-2's compliment} \\
0\ 0\ 0\ 0\ 1\ 1\ 0\ 1 & \\
\hline
0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0 & = (+260)
\end{array}$$

Thus, the resultant value is $\boxed{+(260)}$.

Booth Bit-Pair Recoding Multiplication : Example

$A = 010111\ (+23)$

$B = 110110\ (-10 \rightarrow \text{2's compliment of } 110110)$

Multiply the signed 2's complement numbers using the bit-pair recoding of the multiplier.

$$
\begin{array}{rl}
0\ 1\ 0\ 1\ 1\ 1 & 23 \\
\times\ \ -1\ \ +2\ \ -2 & \times -10\ (\text{Booth recoding } 0\ -1\ +1\ \ 0\ \ -1\ \ 0) \\
\hline
1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ \ 0\ \ 0\ \ 1\ \ 0 & \\
0\ 0\ 0\ 0\ 1\ \ 0\ 1\ \ 1\ \ 1\ \ 0 & \\
1\ 1\ 1\ 0\ 1\ \ 0\ 0\ 1 & \\
\hline
1\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ \ 0 & = -230
\end{array}
$$

Thus, the resultant value is $\boxed{-(230)}$.

Carry-Save Addition of Summands

Multiplication requires the addition of several summands. A technique called *carry-save addition* (CSA) can be used to speed up the process. Consider the 4 × 4 multiplication array shown in Figure 9.16*a*. This structure is in the form of the array shown in Figure 9.6, in which the first row consists of just the AND gates that produce the four inputs *m*3*q*0, *m*2*q*0, *m*1*q*0, and *m*0*q*0.

Instead of letting the carries ripple along the rows, they can be "saved" and introduced into the next row, at the correct weighted positions, as shown in Figure 9.16*b*. This frees up an input to each of three full adders in the first row. These inputs can be used to introduce

Carry-Save Addition of Summands

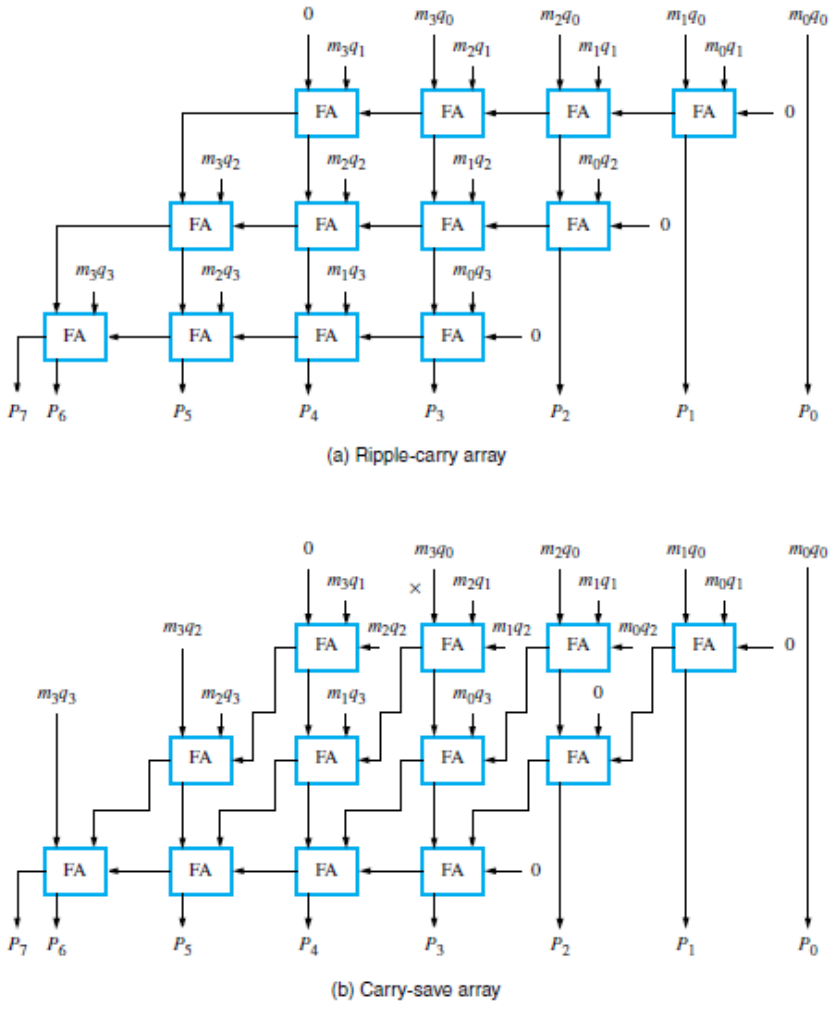
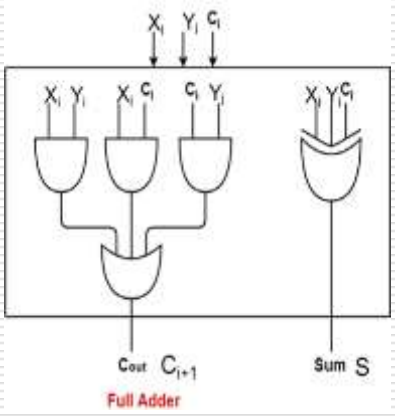


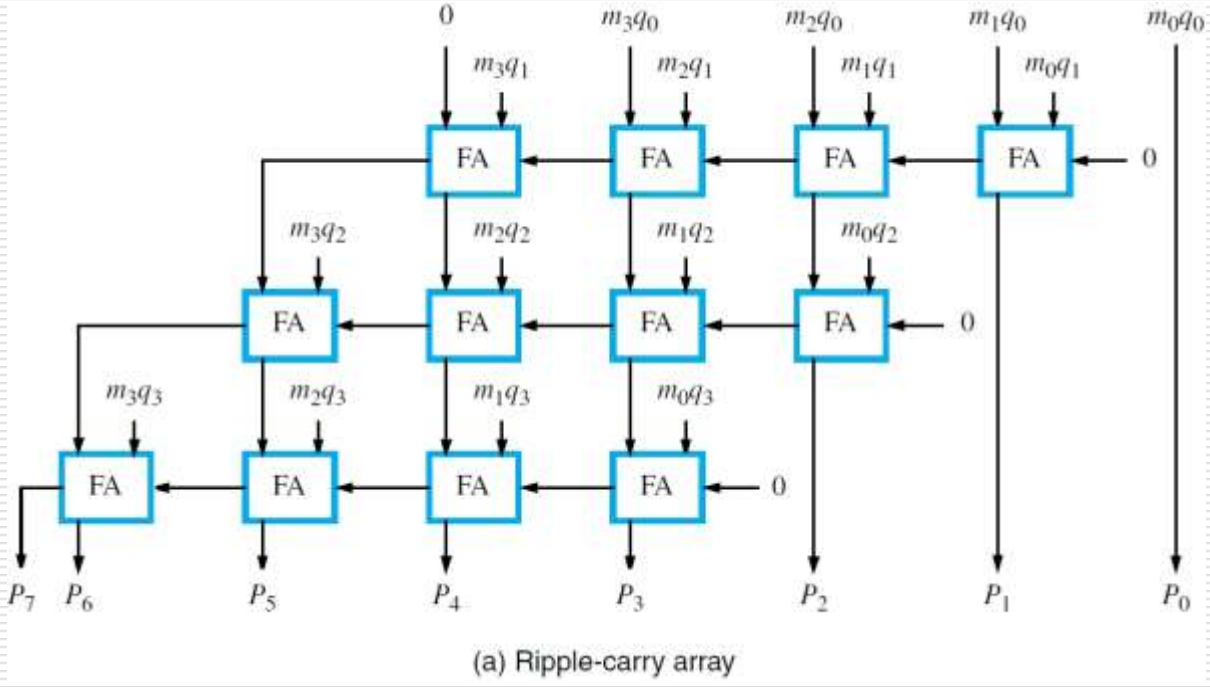
Figure 9.16 Ripple-carry and carry-save arrays for a 4 × 4 multiplier.

Rough Slide to explain Carry-Save Addition of Summands

				1	0	1	1	(11)
			x	1	1	0	1	(13)
				1	0	1	1	
			0	0	0	0		
		1	0	1	1			
	1	0	1	1				
1	0	0	0	1	1	1	1	

				m_3	m_2	m_1	m_0	Multiplicand (M)
			x	q_3	q_2	q_1	q_0	Multiplier (Q)
				m_3q_0	m_2q_0	m_1q_0	m_0q_0	
			m_3q_1	m_2q_1	m_1q_1	m_0q_1		
		m_3q_2	m_2q_2	m_1q_2	m_0q_2			
	m_3q_3	m_2q_3	m_1q_3	m_0q_3				
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	

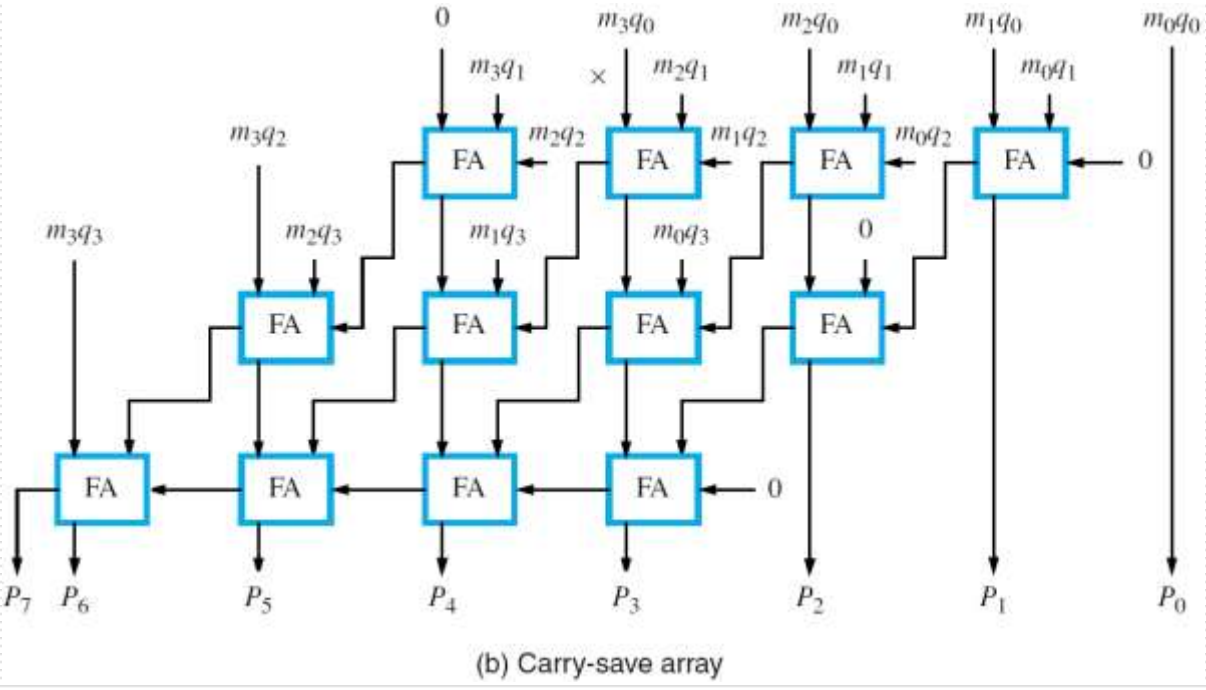




(a) Ripple-carry array

Rough Slide to explain Carry-Save Addition of Summands

				m_3	m_2	m_1	m_0	Multiplicand (M)
			x	q_3	q_2	q_1	q_0	Multiplier (Q)
				m_3q_0	m_2q_0	m_1q_0	m_0q_0	
			m_3q_1	m_2q_1	m_1q_1	m_0q_1		
		m_3q_2	m_2q_2	m_1q_2	m_0q_2			
	m_3q_3	m_2q_3	m_1q_3	m_0q_3				
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	



(b) Carry-save array

Summand Addition Tree using 3-2 Reducers

A more significant reduction in delay can be achieved when dealing with longer operands than those considered in Figure 9.16. We can group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of *S* and *C* vectors in one full-adder delay. Here, we will refer to a full-adder circuit as simply an adder. Next, we group all the *S* and *C* vectors into threes, and perform carry-save addition on them, generating a further set of *S* and *C* vectors in one more adder delay. We continue with this process until there are only two vectors remaining. The adder at each bit position of the three summands is called a *3-2 reducer*, and the logic circuit structure that reduces a number of summands to two is called a *CSA tree*. The final two *S* and *C* vectors can be added in a carry-lookahead adder to produce the desired product. Consider the example shown in Figure 9.17. It involves adding the six shifted versions of the multiplicand for the case of multiplying two, 6-bit, unsigned numbers, where all six bits of the multiplier are equal to 1. The six summands, A, B, . . . , F are added by carry-save addition in Figure 9.18. The blue boxes in these two figures indicate the same operand bits, and show how they are reduced to sum and carry bits in Figure 9.18 by carry-save addition. Three levels of carry-save addition are performed, as shown schematically in Figure 9.19. This figure shows that the final two vectors S4 and C4 are available in three adder delays after the six input summands are applied to level 1. The final regular addition operation on S4 and C4, which produces the product, can be done with a carry-lookahead adder.

Summand Addition Tree using 3-2 Reducers (Contd..)

					1	0	1	1	0	1	(45)	M
				×	1	1	1	1	1	1	(63)	Q
					1	0	1	1	0	1		A
				1	0	1	1	0	1			B
			1	0	1	1	0	1				C
		1	0	1	1	0	1					D
		1	0	1	1	0	1					E
	1	0	1	1	0	1						F
1	0	1	1	0	0	0	1	0	0	1	1	(2,835)	Product

Figure 9.17 A multiplication example used to illustrate carry-save addition as shown in Figure 9.18.

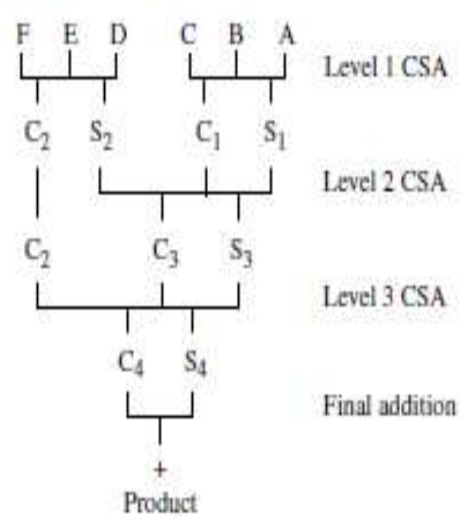


Figure 9.19 Schematic representation of the carry-save addition operations in Figure 9.18.

Summand Addition Tree using 3-2 Reducers (Contd..)

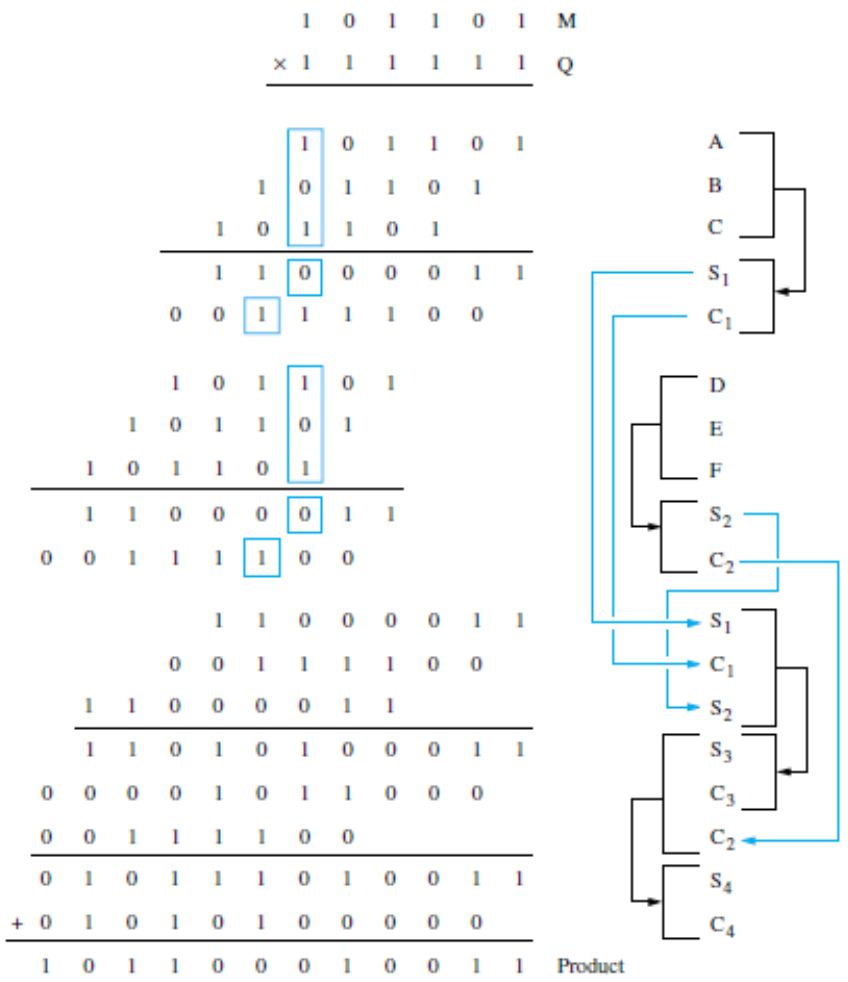


Figure 9.18 The multiplication example from Figure 9.17 performed using carry-save addition.

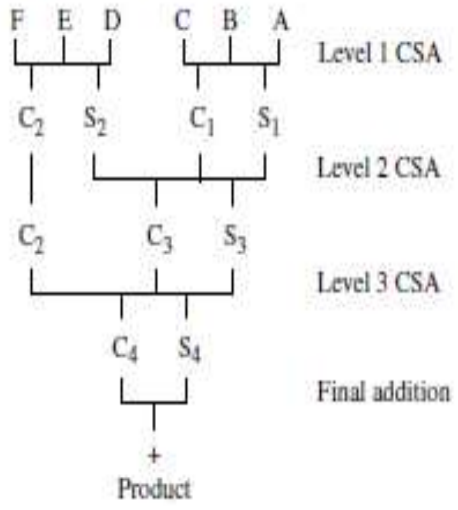


Figure 9.19 Schematic representation of the carry-save addition operations in Figure 9.18.

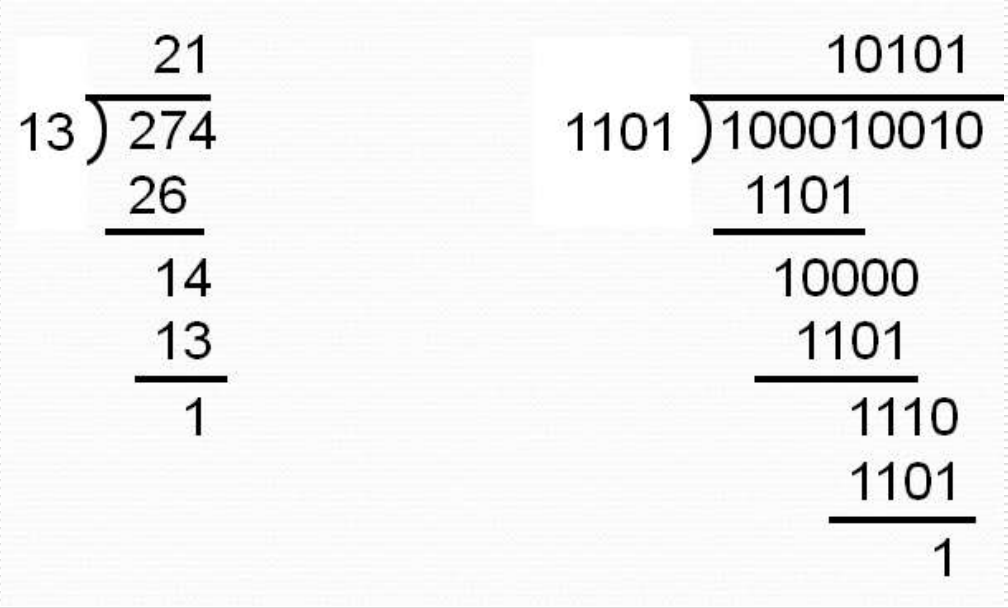
Unit-4

Integer Division

Two Methods: Restoring and Non-restoring algorithms

Manual Division

- Longhand division examples.

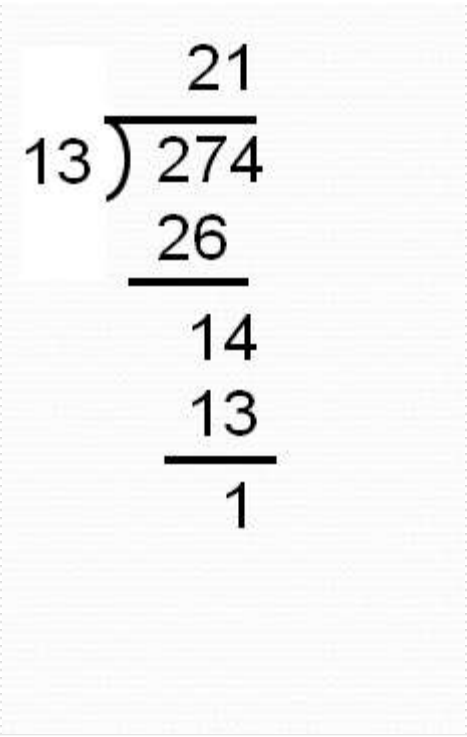


Longhand Division Steps

- Position the divisor appropriately with respect to the dividend and performs a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction.

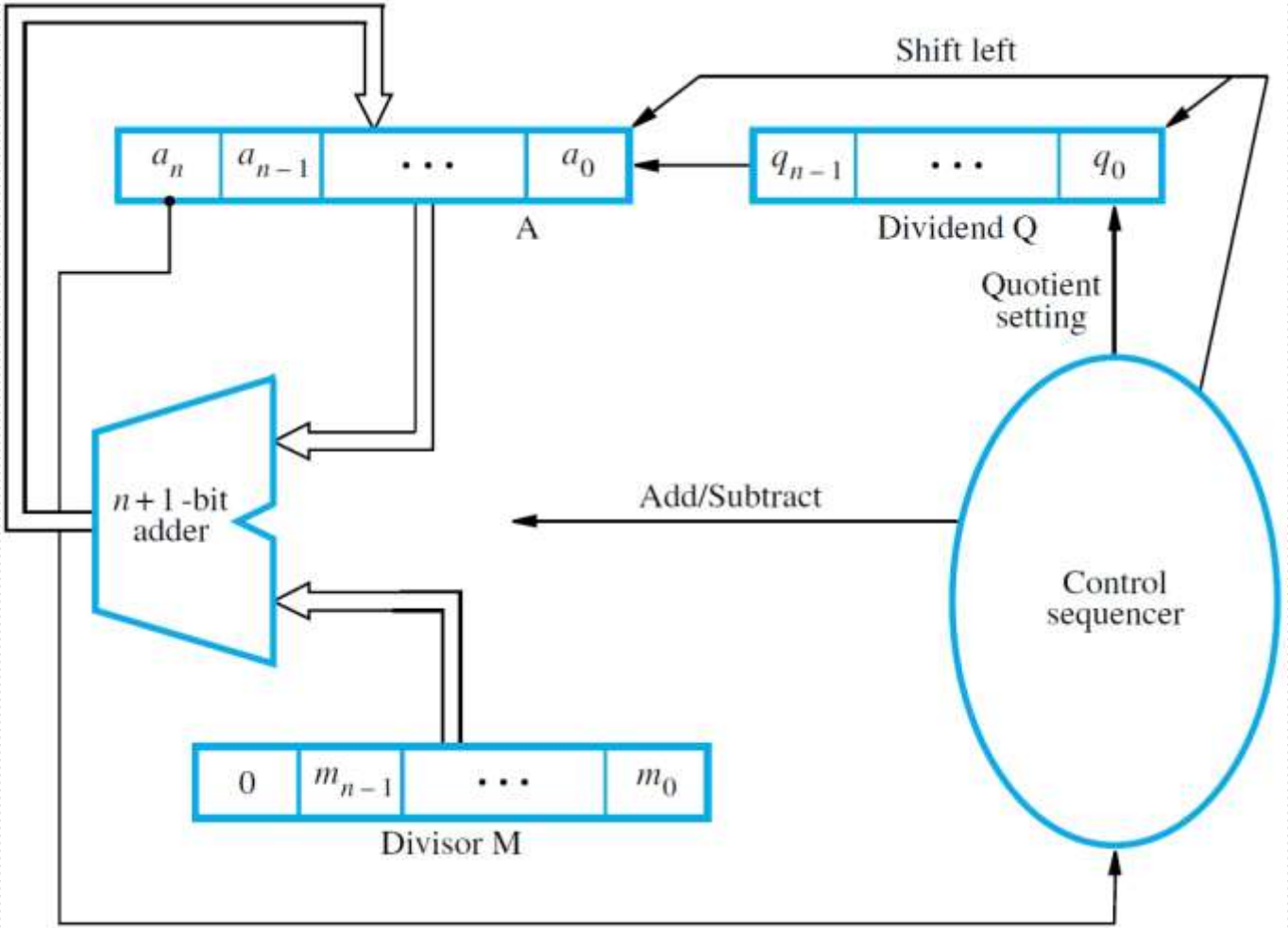
Rough slide to explain Manual Division

- Longhand division examples.

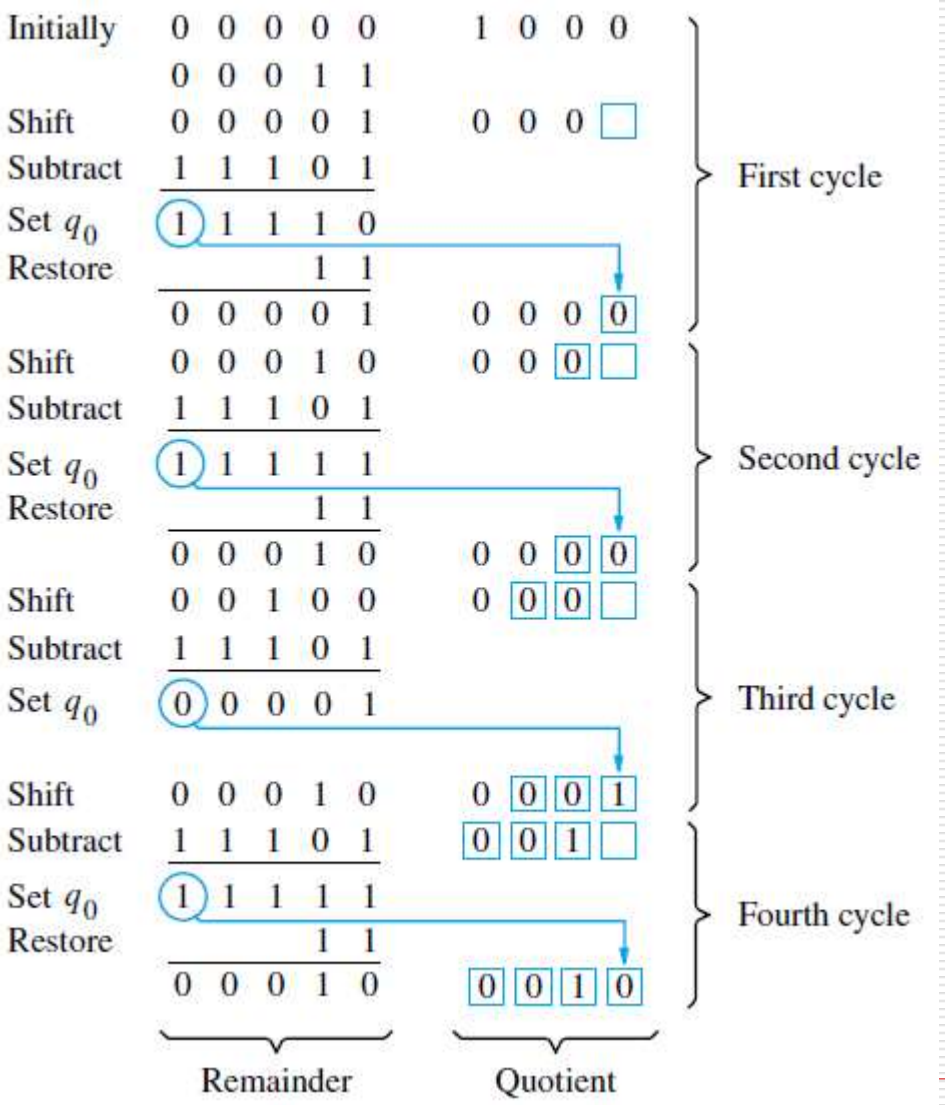


13 is Divisor
274 is Dividend
21 is Quotient
1 is Remainder

Circuit Arrangement for Binary Division



Restoring Division Example



Restoring division algorithm

Initially Register M=Divisor (n+1 bits)

A=0 (n+1 bits)

Q=Dividend (n bits)

After completion of Division:

Register Q=Quotient

Register A: Remainder

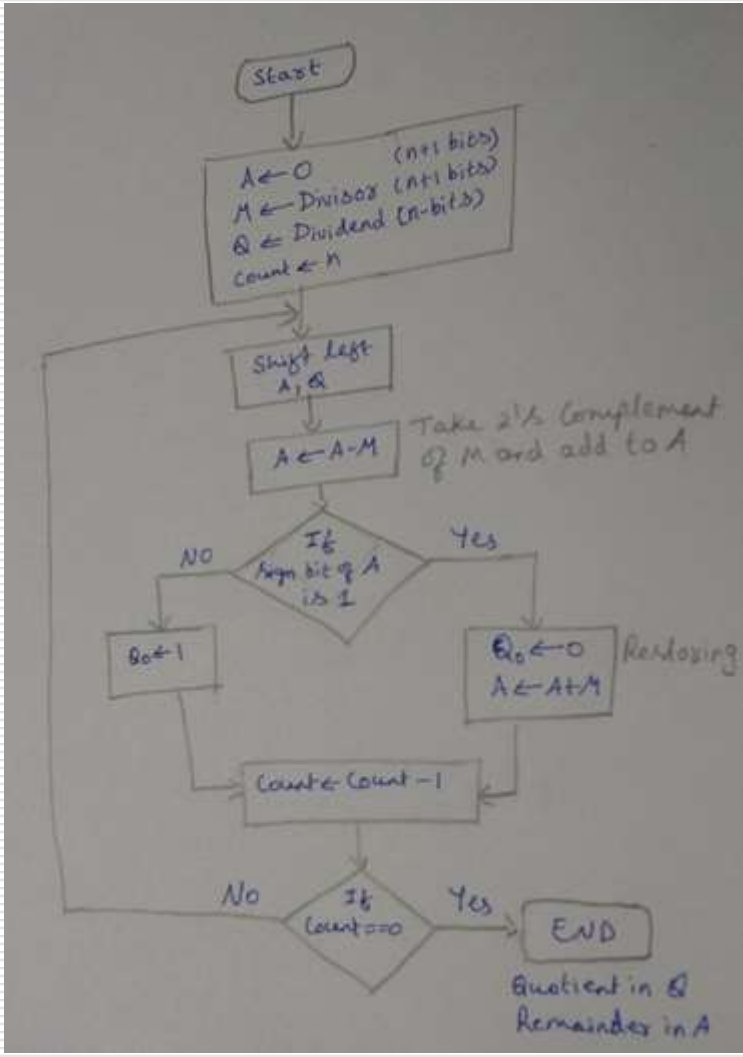
Algorithm:

Do the following for n-times

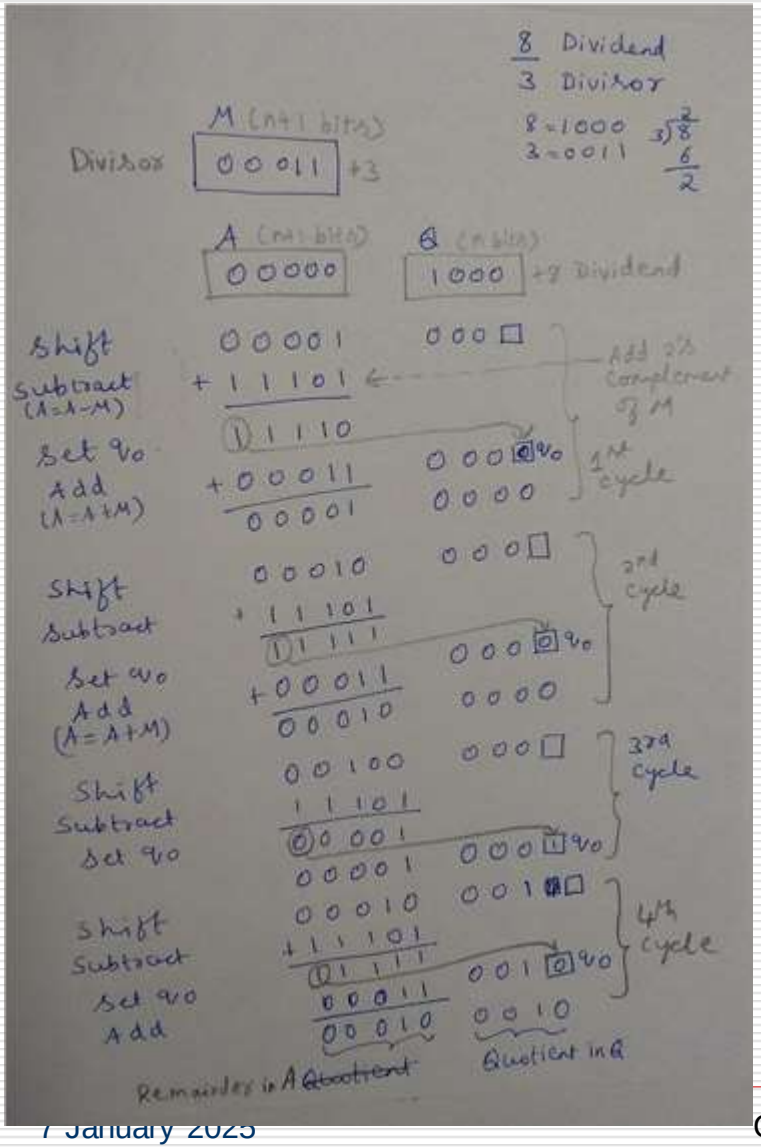
a. Shift A and Q left one binary position

b. Subtract M from A and Place the answer back in A

c. If the Sign of A is 1, set q_0 to 0 and add M back to A (i.e., Restore A) Otherwise Set q_0 to 1



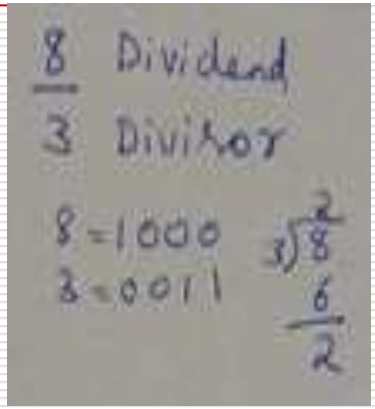
Restoring division algorithm example



Rough Slide to explain Restoring division

	M (n+1) Bits					
Divisor	0	0	0	1	1	+3

	A(n+1 Bits)					Q(n Bits)				
	0	0	0	0	0	1	0	0	0	+8



Initially Register
M=Divisor (n+1 bits)
A=0 (n+1 bits)
Q=Dividend (n bits)

Rough Slide to explain Restoring division

Algorithm:
Do the following for n-times
a. Shift A and Q left one binary position
b. Subtract M from A and Place the answer back in A
c. If the Sign of A is 1, set q_0 to 0 and add M back to A (i.e., Restore A) Otherwise Set q_0 to 1

	M (n+1) Bits					
Divisor	0	0	0	1	1	+3

	A(n+1 Bits)					Q(n Bits)				
	0	0	0	0	0	1	0	0	0	+8
Shift	0	0	0	0	1	0	0	0		

Rough Slide to explain Restoring division

Algorithm:
Do the following for n-times
a. Shift A and Q left one binary position
b. Subtract M from A and Place the answer back in A
c. If the Sign of A is 1, set q_0 to 0 and add M back to A (i.e., Restore A) Otherwise Set q_0 to 1

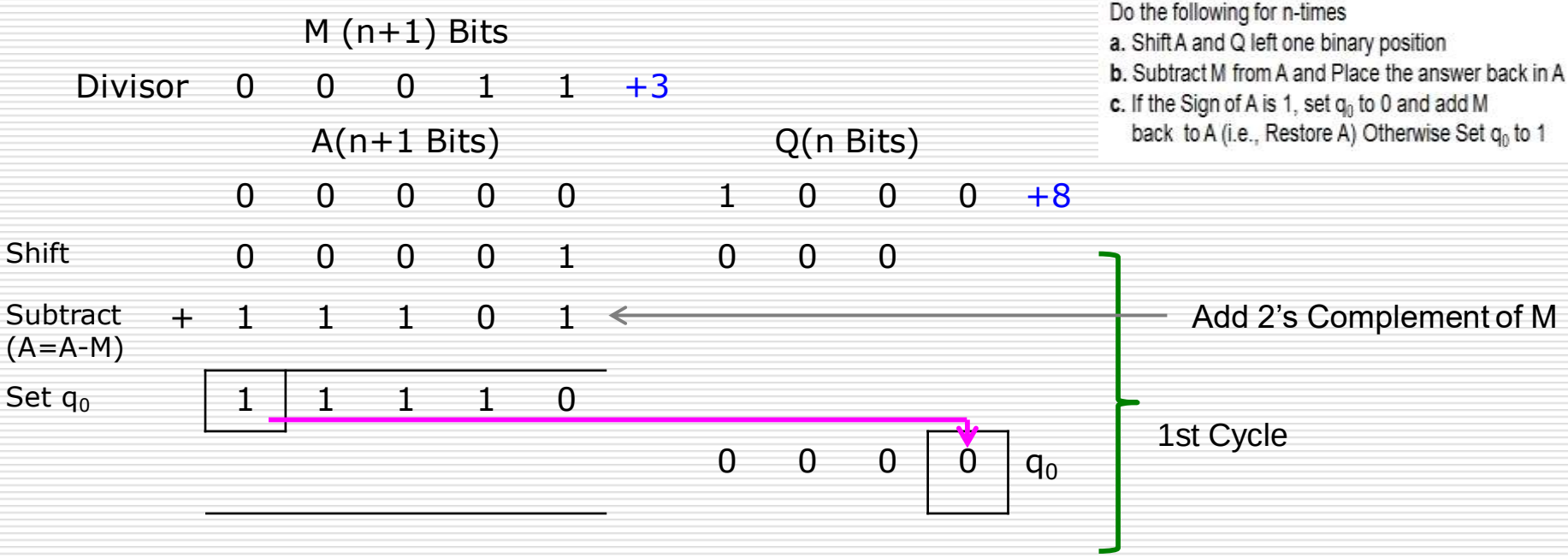
		M (n+1) Bits										
Divisor		0	0	0	1	1	+3					
		A(n+1 Bits)						Q(n Bits)				
		0	0	0	0	0		1	0	0	0	+8
Shift		0	0	0	0	1		0	0	0		
Subtract (A=A-M)	+	1	1	1	0	1	←	Add 2's Complement of M				

Rough Slide to explain Restoring division

Algorithm:
Do the following for n-times
a. Shift A and Q left one binary position
b. Subtract M from A and Place the answer back in A
c. If the Sign of A is 1, set q_0 to 0 and add M back to A (i.e., Restore A) Otherwise Set q_0 to 1

		M (n+1) Bits										
Divisor		0	0	0	1	1	+3					
		A(n+1 Bits)						Q(n Bits)				
		0	0	0	0	0		1	0	0	0	+8
Shift		0	0	0	0	1		0	0	0		
Subtract (A=A-M)	+	1	1	1	0	1	← Add 2's Complement of M					
Set q_0		1	1	1	1	0						

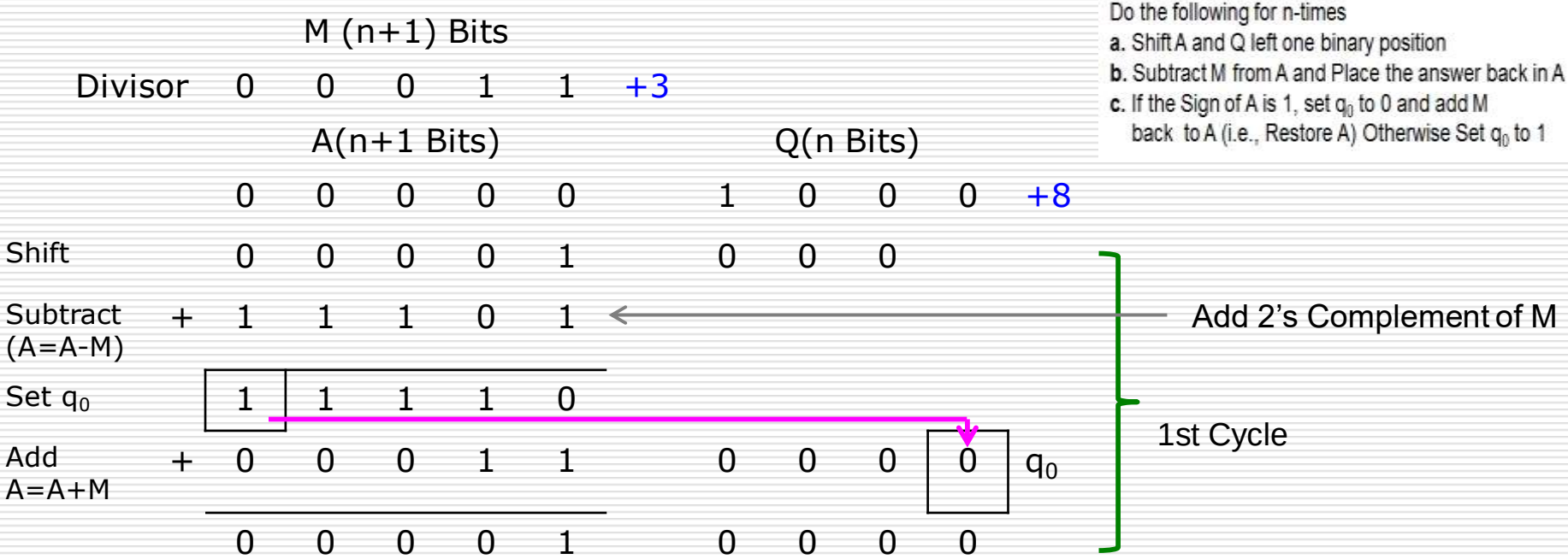
Rough Slide to explain Restoring division



Algorithm:
Do the following for n-times
a. Shift A and Q left one binary position
b. Subtract M from A and Place the answer back in A
c. If the Sign of A is 1, set q_0 to 0 and add M back to A (i.e., Restore A) Otherwise Set q_0 to 1

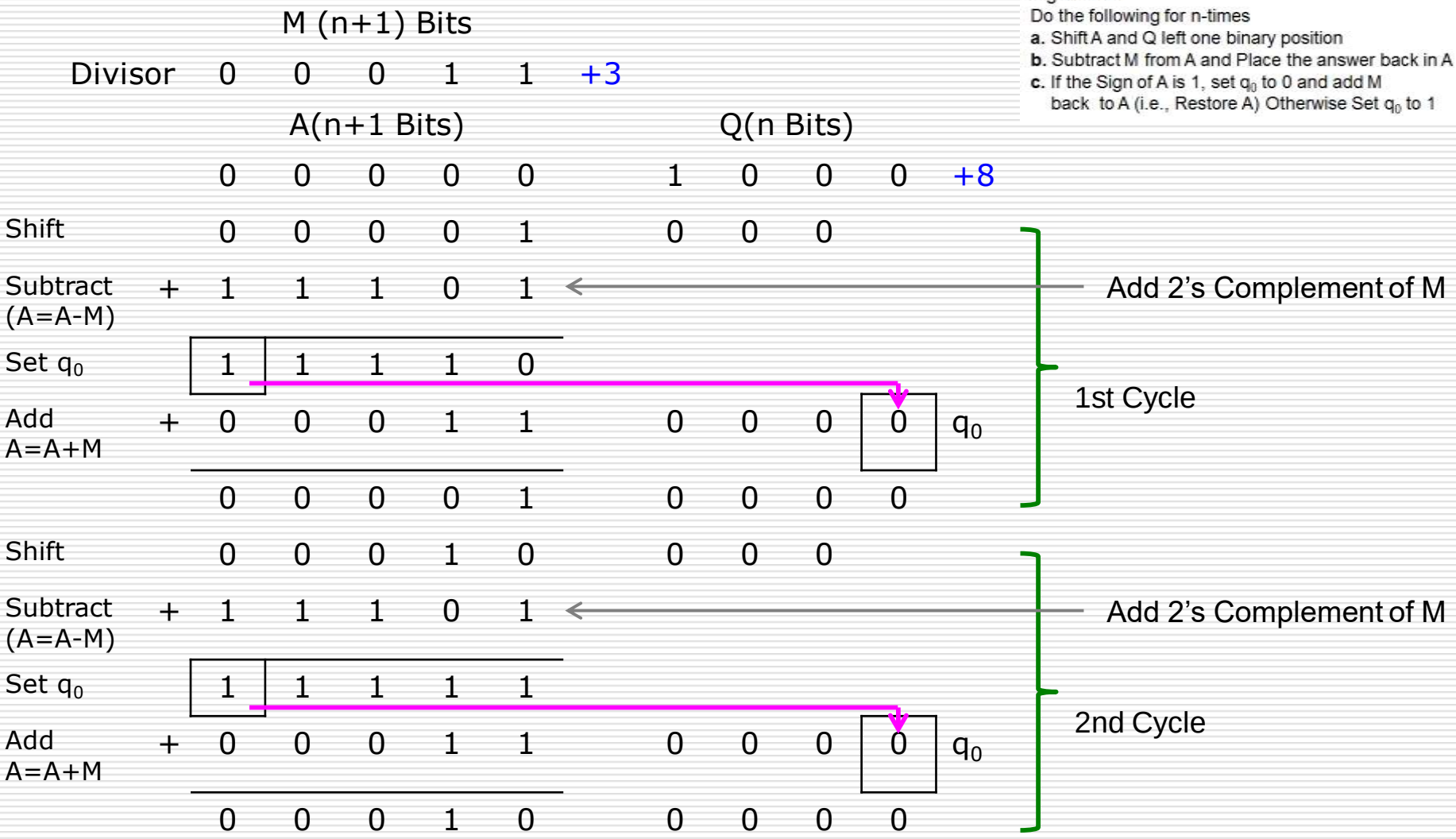
Rough Slide to explain Restoring division

Algorithm:
Do the following for n-times
a. Shift A and Q left one binary position
b. Subtract M from A and Place the answer back in A
c. If the Sign of A is 1, set q_0 to 0 and add M back to A (i.e., Restore A) Otherwise Set q_0 to 1



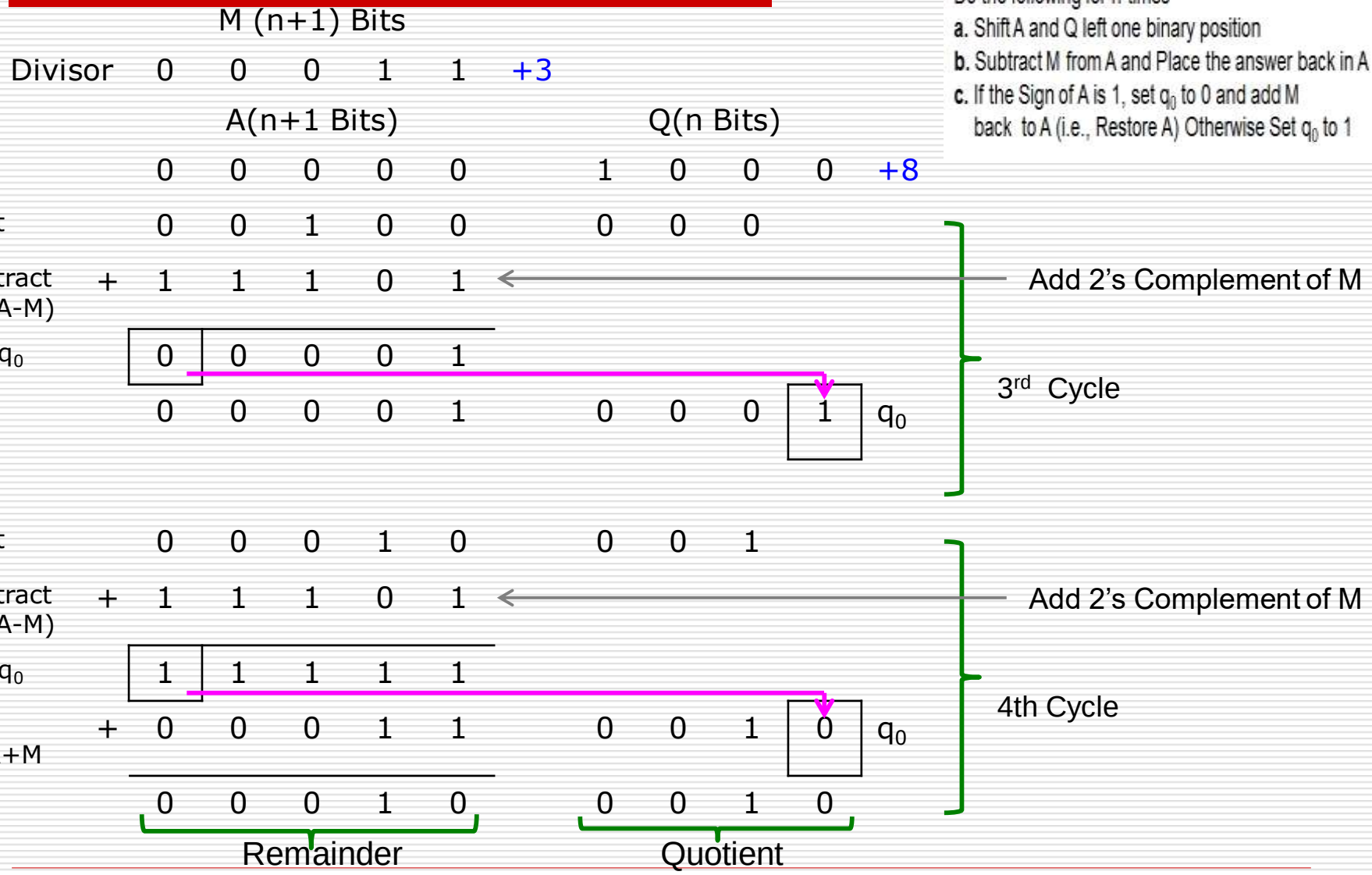
Rough Slide to explain Restoring division

Algorithm:
Do the following for n-times
a. Shift A and Q left one binary position
b. Subtract M from A and Place the answer back in A
c. If the Sign of A is 1, set q_0 to 0 and add M back to A (i.e., Restore A) Otherwise Set q_0 to 1



Rough Slide to explain Restoring division

Algorithm:
Do the following for n-times
a. Shift A and Q left one binary position
b. Subtract M from A and Place the answer back in A
c. If the Sign of A is 1, set q_0 to 0 and add M back to A (i.e., Restore A) Otherwise Set q_0 to 1



Question

Using restoring division algorithm, divided 25 by 4

Restoring Division Algorithm

Initially Register M=Divisor (n+1 bits)

A=0 (n+1 bits)

Q=Dividend (n bits)

After completion of Division:

Register Q=Quotient

Register A: Remainder

Algorithm:

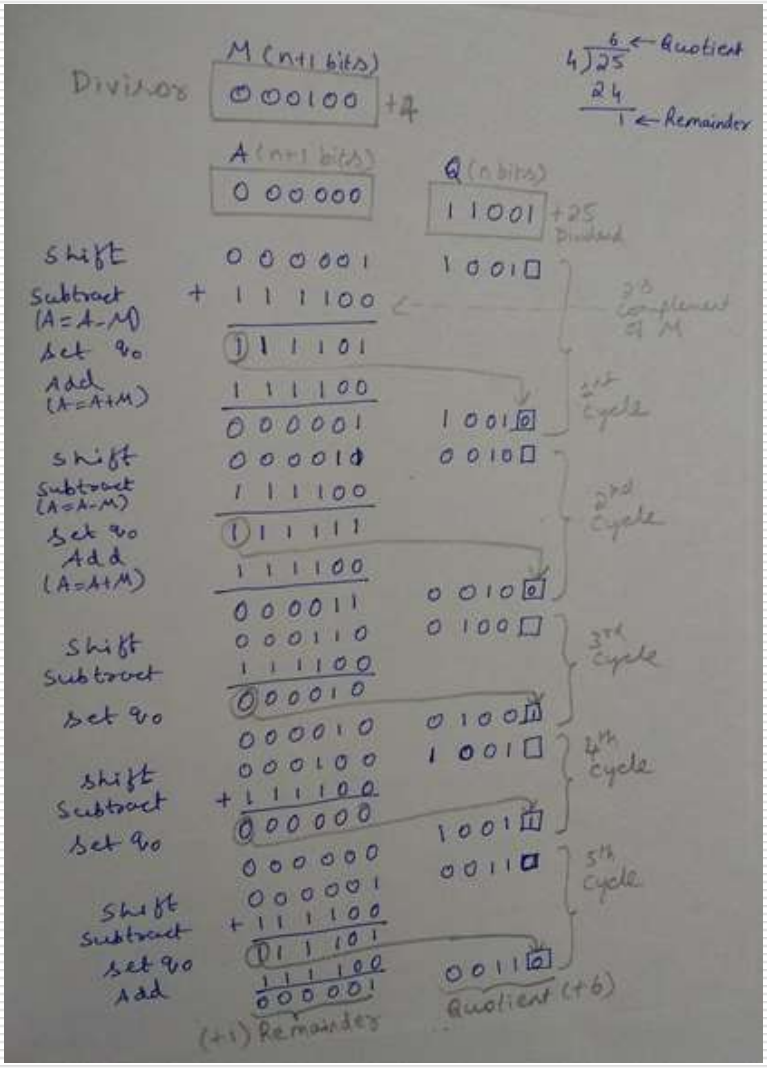
Do the following for n-times

a. Shift A and Q left one binary position

b. Subtract M from A and Place the answer back in A

c. If the Sign of A is 1, set q_0 to 0 and add M back to A (i.e., Restore A) Otherwise Set q_0 to 1

Restoring division algorithm example



Non-Restoring division algorithm

Non restoring Division Algorithm

Initially Register M = Divisor (n+1 bits)
Register Q = Dividend (n bits)
Register A = 0 (n+1 bits)

After completion of Division
Register A = Remainder
Register Q = Quotient

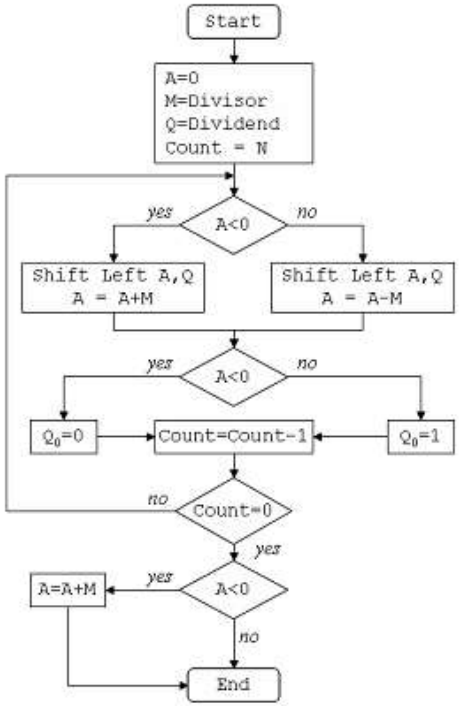
Algorithm:

Step1: Do the following 'n' times

(a) If (sign of A is 0) then {
Shift A and Q left one bit position
Subtract A = A - M
}
else {
Shift A and Q left
Add A = A + M
}

(b) Now, If (sign of A is 0) then
{ Set q_0 to 1 }
else
{ set q_0 to 0 }

Step2: After 'n' iterations are over, if sign of A is 1, add M to A



Non-Restoring division algorithm

Initially Register M=Divisor (n+1 bits)

A=0 (n+1 bits)

Q=Dividend (n bits)

After completion of Division:

Register Q: Quotient

Register A: Remainder

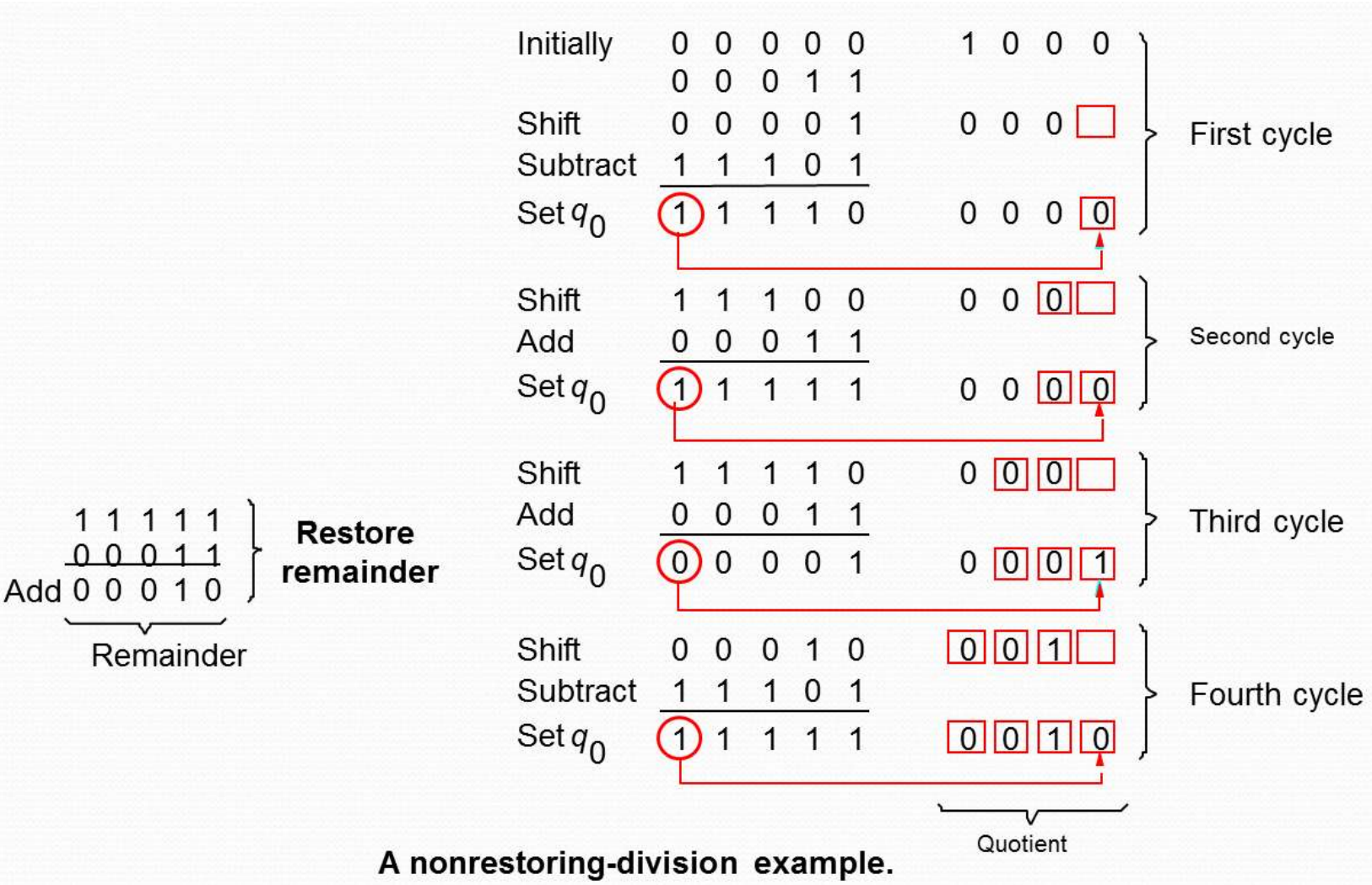
Algorithm:

Step 1: (Repeat *n* times)

- If the sign of A is 0shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
- Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.

Step2: If the sign of A is 1, add M to A

Non-Restoring division algorithm: Example

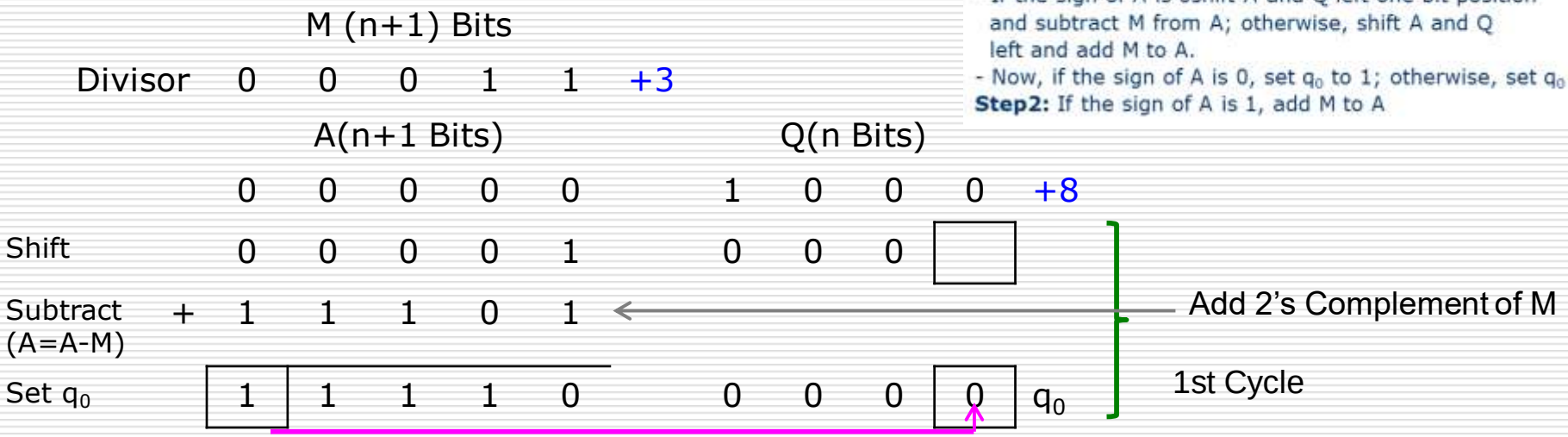


A nonrestoring-division example.

Rough Slide to explain Non-Restoring division algorithm: Example

Algorithm:
Step 1: (Repeat *n* times)
- If the sign of A is 0shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
- Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.

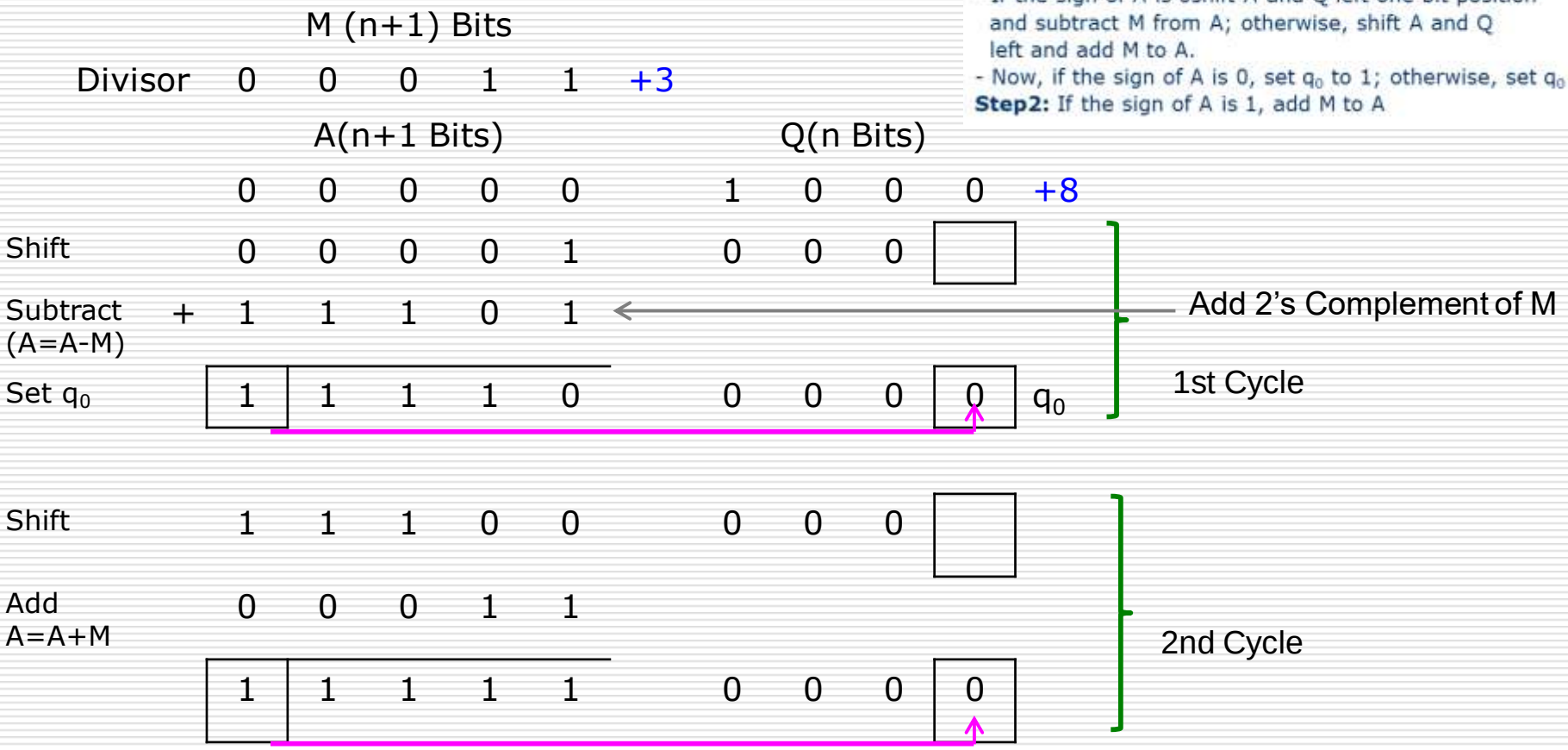
Step2: If the sign of A is 1, add M to A



Rough Slide to explain Non-Restoring division algorithm: Example

Algorithm:
Step 1: (Repeat *n* times)
- If the sign of A is 0shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
- Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.

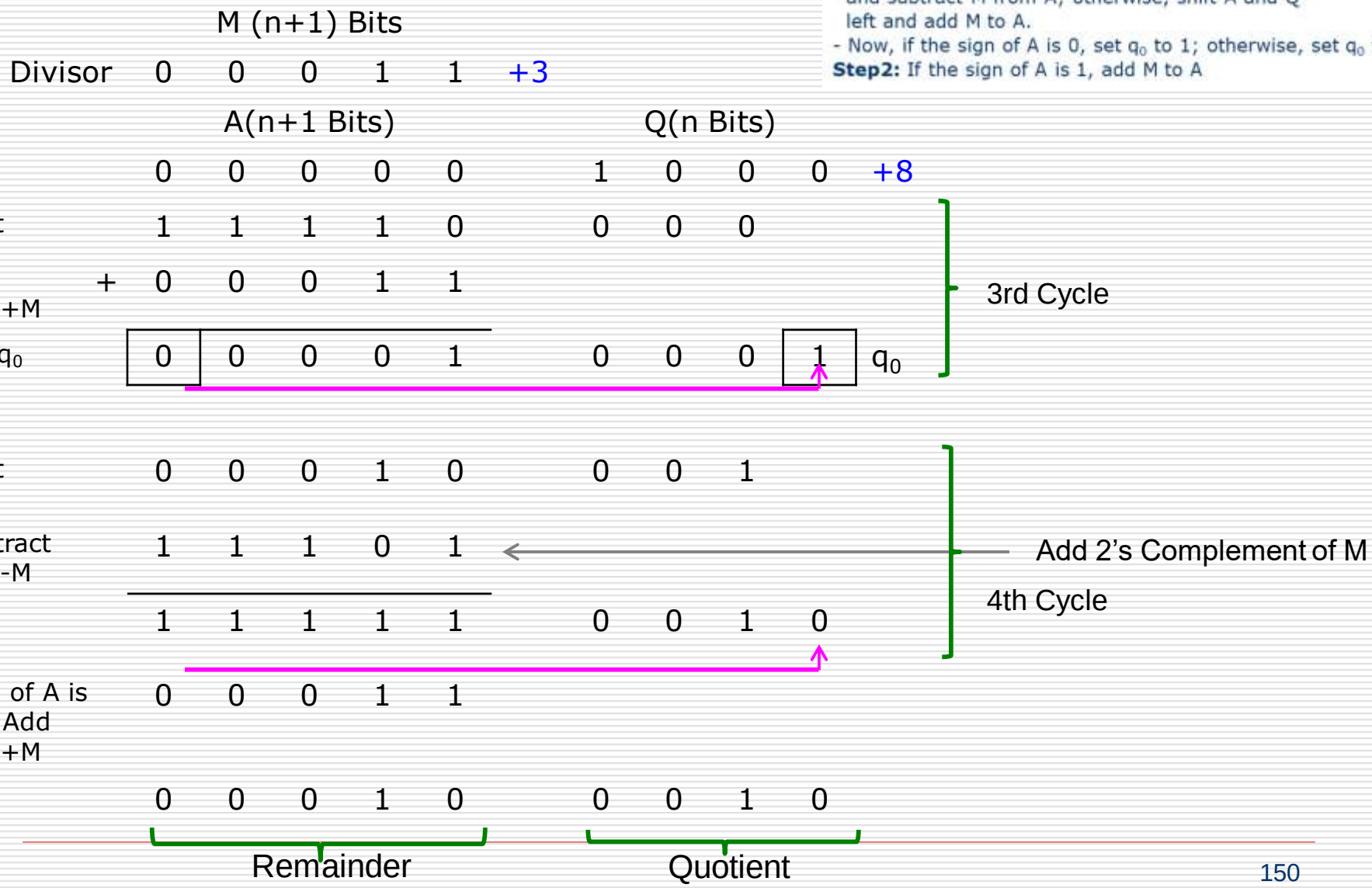
Step2: If the sign of A is 1, add M to A



Rough Slide to explain Non-Restoring division algorithm: Example

Algorithm:
Step 1: (Repeat *n* times)
- If the sign of A is 0shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
- Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.

Step2: If the sign of A is 1, add M to A



Question

Using non-restoring division algorithm, divided 25 by 4

Restoring Division Algorithm

Initially Register M=Divisor (n+1 bits)

A=0 (n+1 bits)

Q=Dividend (n bits)

After completion of Division:

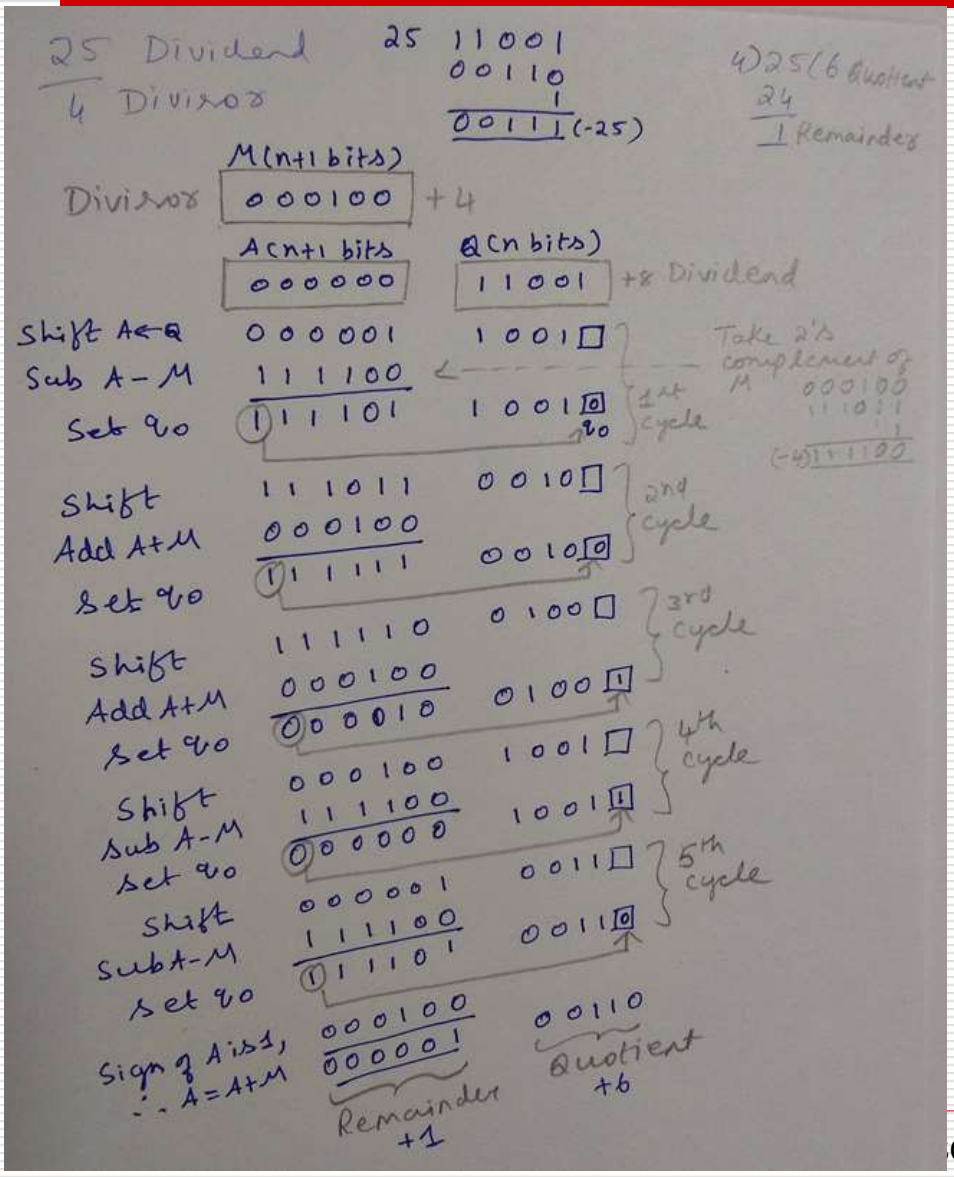
Register Q=Quotient

Register A: Remainder

Algorithm:

- Step 1: (Repeat *n* times)
  - If the sign of A is 0shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
  - Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.
- Step2: If the sign of A is 1, add M to A

Non-Restoring division algorithm: Example



Unit-4

Floating-point Numbers and Operations

Floating-Point Representation

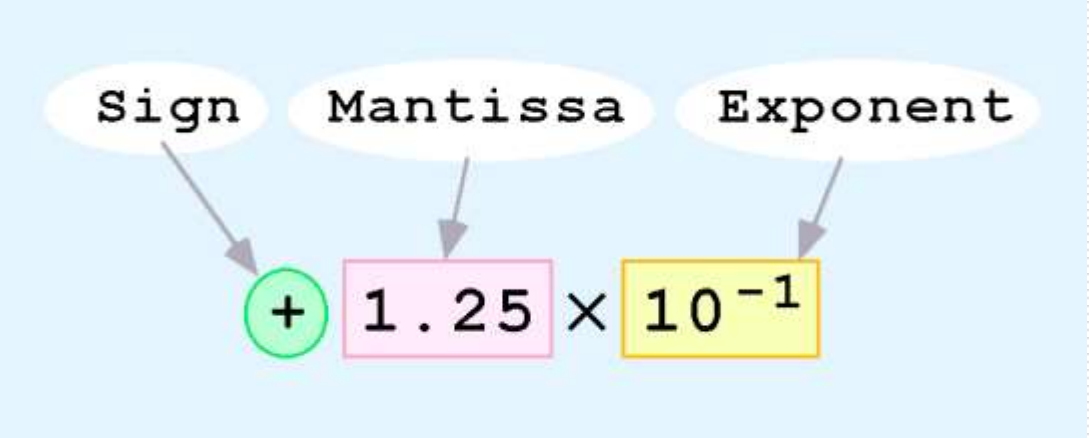
- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

    $0.125 = 1.25 \times 10^{-1}$

    $5{,}000{,}000 = 5.0 \times 10^{6}$

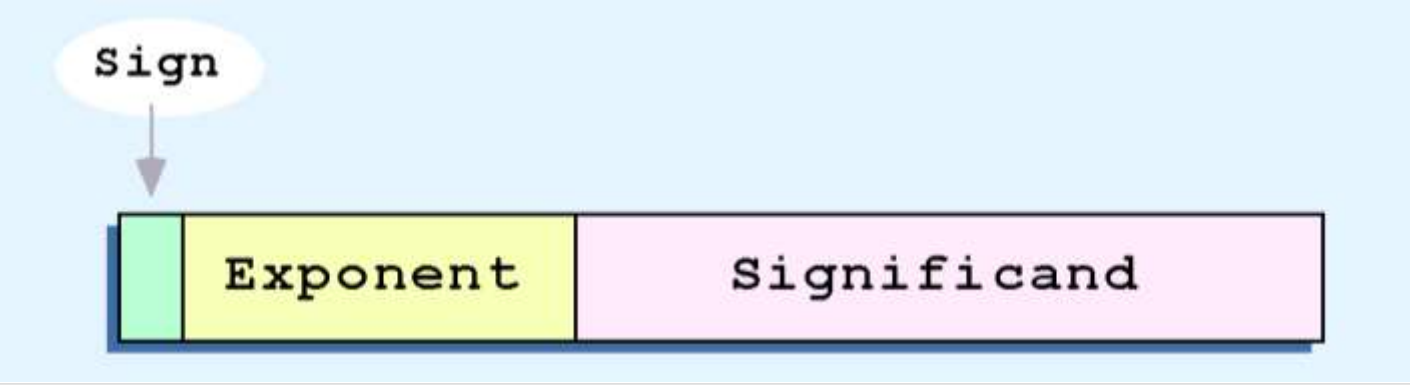
Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



Floating-Point Representation

- Computer representation of a floating-point number consists of three fixed-size fields:



- This is the standard arrangement of these fields.

Note: Although "significand" and "mantissa" do not technically mean the same thing, many people use these terms interchangeably.

Rough Slide to explain Excess Notation

1. Add the excess value (2^{N-1}, where N is the number of bits used to represent the number) to the number.

2. Convert the resulting number into binary format.

The 2^{N-1} is often referred to as the *Magic Number* for computing the excess representation of the number (except that there is no magic in it). Table 7 presents all the numbers that can be represented using the excess-8 notation.

Number	Excess Number	Bit Pattern
7	15	1111
6	14	1110
5	13	1101
4	12	1100
3	11	1011
2	10	1010
1	9	1001
0	8	1000
-1	7	0111
-2	6	0110
-3	5	0101
-4	4	0100
-5	3	0011
-6	2	0010
-7	1	0001
-8	0	0000

Table 7. Numbers using the Excess-8 representation

Excess Notation

The excess notation is a means of representing both negative and positive numbers in a manner in which the order of the bit patterns is maintained. The algorithm for computing the excess notation bit pattern is as follows:

1. Add the excess value (2^{N-1}, where N is the number of bits used to represent the number) to the number.
2. Convert the resulting number into binary format.

The 2^{N-1} is often referred to as the *Magic Number* for computing the excess representation of the number (except that there is no magic in it). Table 7 presents all the numbers that can be represented using the excess-8 notation.

Number	Excess Number	Bit Pattern
7	15	1111
6	14	1110
5	13	1101
4	12	1100
3	11	1011
2	10	1010
1	9	1001
0	8	1000
-1	7	0111
-2	6	0110
-3	5	0101
-4	4	0100
-5	3	0011
-6	2	0010
-7	1	0001
-8	0	0000

Table 7. Numbers using the Excess-8 representation

The number of bits used to represent a code in excess-8 is 4 bits ($2^{4-1} = 8$). Also, the bit patterns are in sequence (the largest number that can be represented has the bit pattern 1111).

Example 11. Consider the following operation 7 – 2. Substituting the bit patterns from the table:

$$\begin{array}{r} 1111 \\ +\ 0110 \\ \hline 1\ 0101 \end{array}$$

The result of the addition operation is the bit-pattern used for 5 in binary. The excess notation representation however takes longer to compute than the 2's complement notation. The excess notation will however play an important part in computing floating-point representations.

IEEE floating point number representation

- The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the **Institute of Electrical and Electronics Engineers (IEEE)**. The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability. IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macs, and most Unix platforms.

There are several ways to represent floating point number but IEEE 754 is the most efficient in most cases. IEEE 754 has 3 basic components:

The Sign of Mantissa –
This is as simple as the name. 0 represents a positive number while 1 represents a negative number.

The Biased exponent –
The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent.

The Normalised Mantissa –
The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.

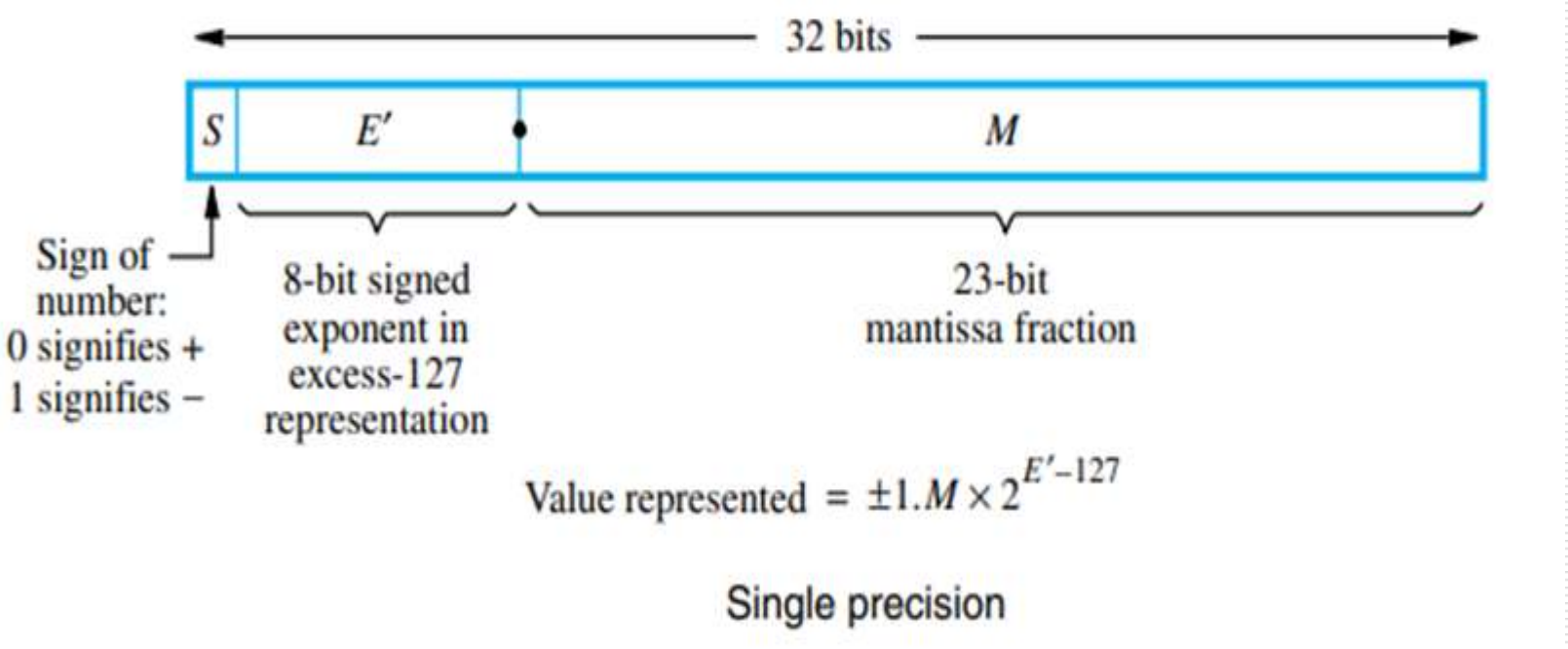
IEEE floating point number representation

IEEE 754 numbers are divided into two based on the three components (Sign, Exponent, Mantissa) :

1. Single Precision
2. Double Precision

IEEE standard floating-point formats: Single Precision

The basic IEEE format is a 32-bit representation, shown in Figure below. The leftmost bit represents the sign, S, for the number. The next 8 bits, E , represent the signed exponent of the scale factor (with an implied base of 2), and the remaining 23 bits, M , are the fractional part of the significant bits.



IEEE standard floating-point formats: Single Precision (Contd….)

The full 24-bit string, B, of significant bits, called the mantissa, always has a leading 1, with the binary point immediately to its right. Therefore, the mantissa

$$B = 1.M = 1.b_{-1}b_{-2}\ldots b_{-23}$$

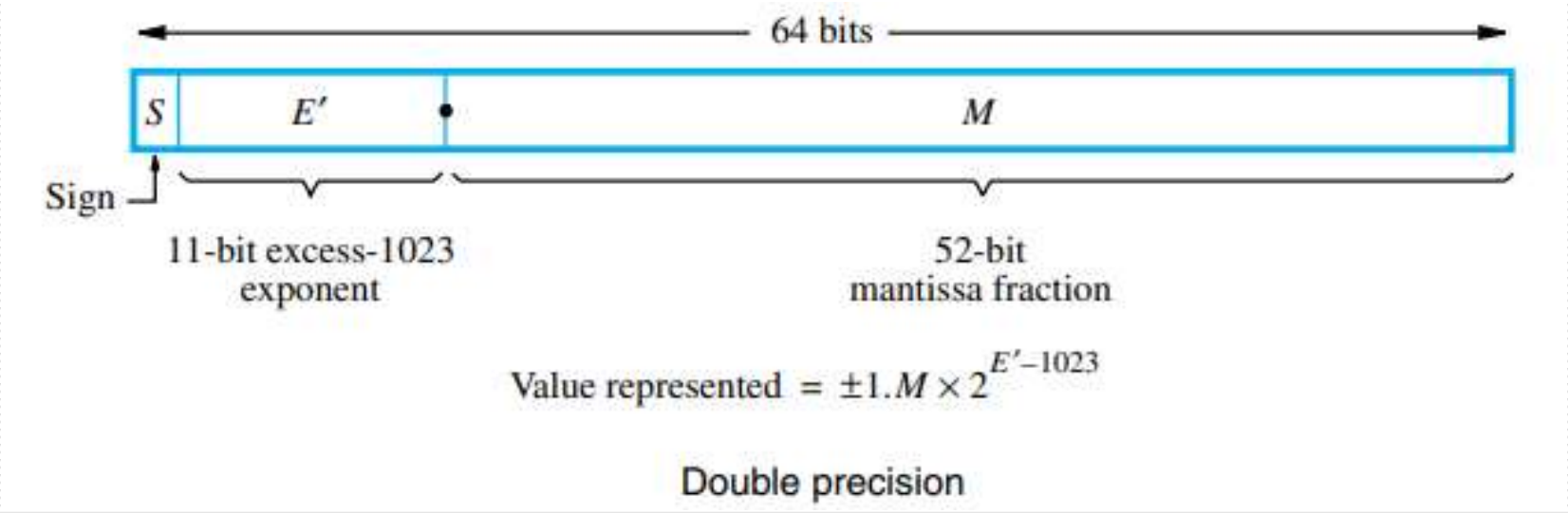
has a value

$$V(B) = 1 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-23} \times 2^{-23}$$

Instead of the actual signed exponent, E, the value stored in the exponent field is an unsigned integer E = E + 127. This is called the excess-127 format. Thus, E is in the range $0 \le E \le 255$. The end values of this range, 0 and 255, are used to represent special values, as described later. Therefore, the range of E for normal values is $1 \le E \le 254$. This means that the actual exponent, E, is in the range $-126 \le E \le 127$. The use of the excess-127 representation for exponents simplifies comparison of the relative sizes of two floating-point numbers. The 32-bit standard representation in Figure (in previous slide) is called a single-precision representation because it occupies a single 32-bit word. The scale factor has a range of 2^{-126} to 2^{+127}, which is approximately equal to $10^{\pm 38}$. The 24-bit mantissa provides approximately the same precision as a 7-digit decimal value.

IEEE standard floating-point formats: Double Precision

To provide more precision and range for floating-point numbers, the IEEE standard also specifies a double-precision format, as shown in Figure. The double-precision format has increased exponent and mantissa ranges. The 11-bit excess-1023 exponent E has the range $1 \leq E \leq 2046$ for normal values, with 0 and 2047 used to indicate special values, as before. Thus, the actual exponent E is in the range $-1022 \leq E \leq 1023$, providing scale factors of 2^{-1022} to 2^{1023} (approximately $10^{\pm 308}$). The 53-bit mantissa provides a precision equivalent to about 16 decimal digits.



Example: Single and Double Precision

Represent the number **-307.1875** in IEEE single and double precision floating point number formats

Step 1: Convert Decimal number to binary

Integer

2	307		
2	153	-	1
2	76	-	1
2	38	-	0
2	19	-	0
2	9	-	1
2	4	-	1
2	2	-	0
	1	-	0

Therefore 307 is 100110011

Fraction

0.1875 x 2	= 0.3750	0
0.375 x 2	= 0.750	0
0.75 x 2	= 1.50	1
0.5 x 2	= 1.0	1

Therefore 0.1875 is 0.0011

307.1875 =100110011.0011

Example: Single and Double Precision

Represent the number -307.1875 in IEEE single and double precision floating point number formats

Step 2: Normalize the Number

$$-307.1875 = -100110011.0011$$

$$-100110011.0011 = -1.001100110011 \times 2^{8}$$

Step 3: Representation

a. Single Precision

S = 1 ; E = 8 E' = E+127 = 8+127 = 135 = 10000111 ; M=001100110011

S	E'	M
1	10000111	00110011001100…0
1-Bit	8-Bits	23-Bits

Example: Single and Double Precision

Represent the number -307.1875 in IEEE single and double precision floating point number formats

Step 2: Normalize the Number

$$-307.1875 = -100110011.0011$$

$$-100110011.0011 = -1.001100110011 \times 2^{8}$$

Step 3: Representation

b. Double Precision

S = 1 ; E = 8 E' = E+1023 = 8+1023 = 1031 = 10000000111 ; M=0011001100111

S	E'	M
1	10000000111	001100110011 00...0
1-Bit	11-Bits	52-Bits

Example: Single and Double Precision

Represent −307.1875 in single-precision and double precision floating point representation format

Solution:

Step 1: Convert decimal number to binary

Integer:

2 | 307	
2 | 153	−1
2 | 76	−1
2 | 38	−0
2 | 19	−0
2 | 9	−1
2 | 4	−1
2 | 2	−0
1	−0

$\therefore 307 = 100110011$

Fraction:

$0.1875 \times 2 = 0.3750$	0
$0.3750 \times 2 = 0.750$	0
$0.750 \times 2 = 1.50$	1
$0.5 \times 2 = 1.0$	1

$\therefore 0.1875 = 0011$

$\therefore$ Binary number for $307.1875 = 100110011.0011$

Step 2: Normalize the number

$-100110011.0011 = -1.00110011\,0011 \times 2^{8}$

Step 3: Representation

(a) Single precision

$S = 1$, $E = 8$, $M = 00110011\,0011$

$E' = 8 + 127 = 135 = 10000111_2$

$E = E' - 127$

$\therefore E' = E + 127$

$\therefore$ number is

S	E'	M
1	10000111	00110011001 10-0
1 bit	8-bits	23 bits

(b) Double precision

$S = 1$ $\quad E' = E + 1023 = 8 + 1023 = 1031_{10} = 10000000111_2$

$\therefore$ number is

S	E'	M
1	10000000111	00110011001 10-00
1 bit	11-bits	52 bits

2 | 135	
2 | 67	−1
2 | 33	−1
2 | 16	−1
2 | 8	−0
2 | 4	−0
2 | 2	−0
1	−0

2 | 1031	
2 | 515	−1
2 | 257	−1
2 | 128	−1
2 | 64	−0
2 | 32	−0
2 | 16	−0
2 | 8	−0
2 | 4	−0
2 | 2	−0
1	−0

Example: Single and Double Precision

Represent the number 0.0625 in IEEE single and double precision floating point number formats

Step 1: Convert Decimal number to binary

Integer Part = 0

Fraction Part

0.0625 x 2	= 0.1250	0
0.1250 x 2	= 0.250	0
0.25 x 2	= 0.50	0
0.5 x 2	= 1.0	1

Therefore 0.0625 is 0.0001

0.0625 =0.0001

Example: Single and Double Precision

Represent the number 0.0625 in IEEE single and double precision floating point number formats

Step 2: Normalize the Number

$$0.0625 = 0.0001 = 1.0 \times 2^{-4}$$

Step 3: Representation

a. Single Precision

S = 0 ; E = -4 E' = E+127 = -4+127 = 123 = 01111011 ; M=0

S	E'	M
0	01111011	000000000…0
1-Bit	8-Bits	23-Bits

Example: Single and Double Precision

Represent the number 0.0625 in IEEE single and double precision floating point number formats

Step 2: Normalize the Number

$$0.0625 = 0.0001 = 1.0 \times 2^{-4}$$

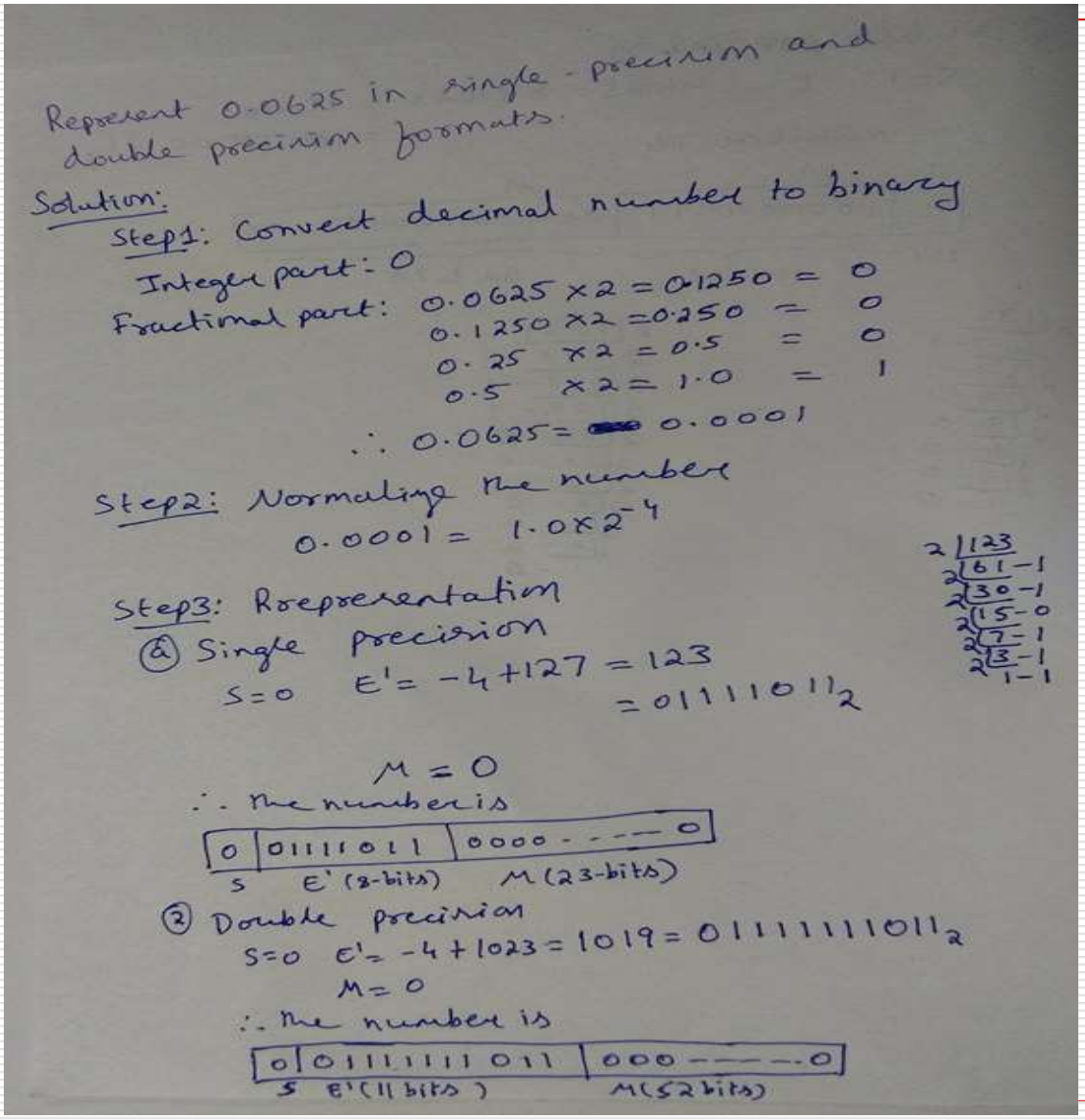
Step 3: Representation

a. Double Precision

S = 0 ; E = -4 E' = E+1023 = -4+1023 = 1019 = 01111111011 ; M=0

S	E'	M
0	01111111011	000000000…0
1-Bit	11-Bits	52-Bits

Example: Single and Double Precision



Represent 0.0625 in single-precision and double precision formats.

Solution:

Step1: Convert decimal number to binary

Integer part: 0

Fractional part:

$0.0625 \times 2 = 0.1250 = 0$

$0.1250 \times 2 = 0.250 = 0$

$0.25 \times 2 = 0.5 = 0$

$0.5 \times 2 = 1.0 = 1$

$\therefore 0.0625 = 0.0001$

Step2: Normalize the number

$0.0001 = 1.0 \times 2^{-4}$

Step3: Representation

(a) Single precision

$S = 0 \quad E' = -4 + 127 = 123$

$= 01111011_2$

$M = 0$

$\therefore$ The number is

0	01111011	0000 ---- 0
S	E' (8-bits)	M (23-bits)

(b) Double precision

$S = 0 \quad E' = -4 + 1023 = 1019 = 01111111011_2$

$M = 0$

$\therefore$ The number is

0	01111111011	000 ---- 0
S	E' (11 bits)	M (52 bits)

$2 \mid 123$

$2 \mid 61 - 1$

$2 \mid 30 - 1$

$2 \mid 15 - 0$

$2 \mid 7 - 1$

$2 \mid 3 - 1$

$1 - 1$

precision format

Solution:

Step 1: Convert decimal number to binary

Integer part: $5 = 101_2$

Fractional part:

$0.2 \times 2 = 0.4 \quad 0$

$0.4 \times 2 = 0.8 \quad 0$

$0.8 \times 2 = 1.6 \quad 1$

$0.6 \times 2 = 1.2 \quad 1$

$0.2 \times 2 = 0.4$

$\therefore 0.2 = 0.0011001100110011\ldots$

$\therefore$ Binary number is $101.0011001100110011\ldots$

Step 2: Normalize the number

$101.0011001100110011\ldots = 1.0100110011001100\ldots \times 2^2$

Step 3: Representation

(a) Single Precision

$S = 0 \quad E = 2$

$\therefore E' = 2 + 127 = 129 = 10000001_2$

$M = 01001100110011\ldots$

0	10000001	01001100110011...
S	E' (8-bits)	M (23-bits)

(b) Double precision

$S = 0 \quad E = 2$

$\therefore E' = 2 + 1023 = 1025 = 10000000001$

0	10000000001	0100110011001100...
S	E' (11-bits)	M (52 bits)

Unit-4

Arithmetic Operations on Floating-Point Numbers, Guard Bits and Truncation , Implementing Floating-Point Operations

Arithmetic Operations on Floating-Point Numbers

Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.

2. Set the exponent of the result equal to the larger exponent.

3. Perform addition/subtraction on the mantissas and determine the sign of the result.

4. Normalize the resulting value, if necessary

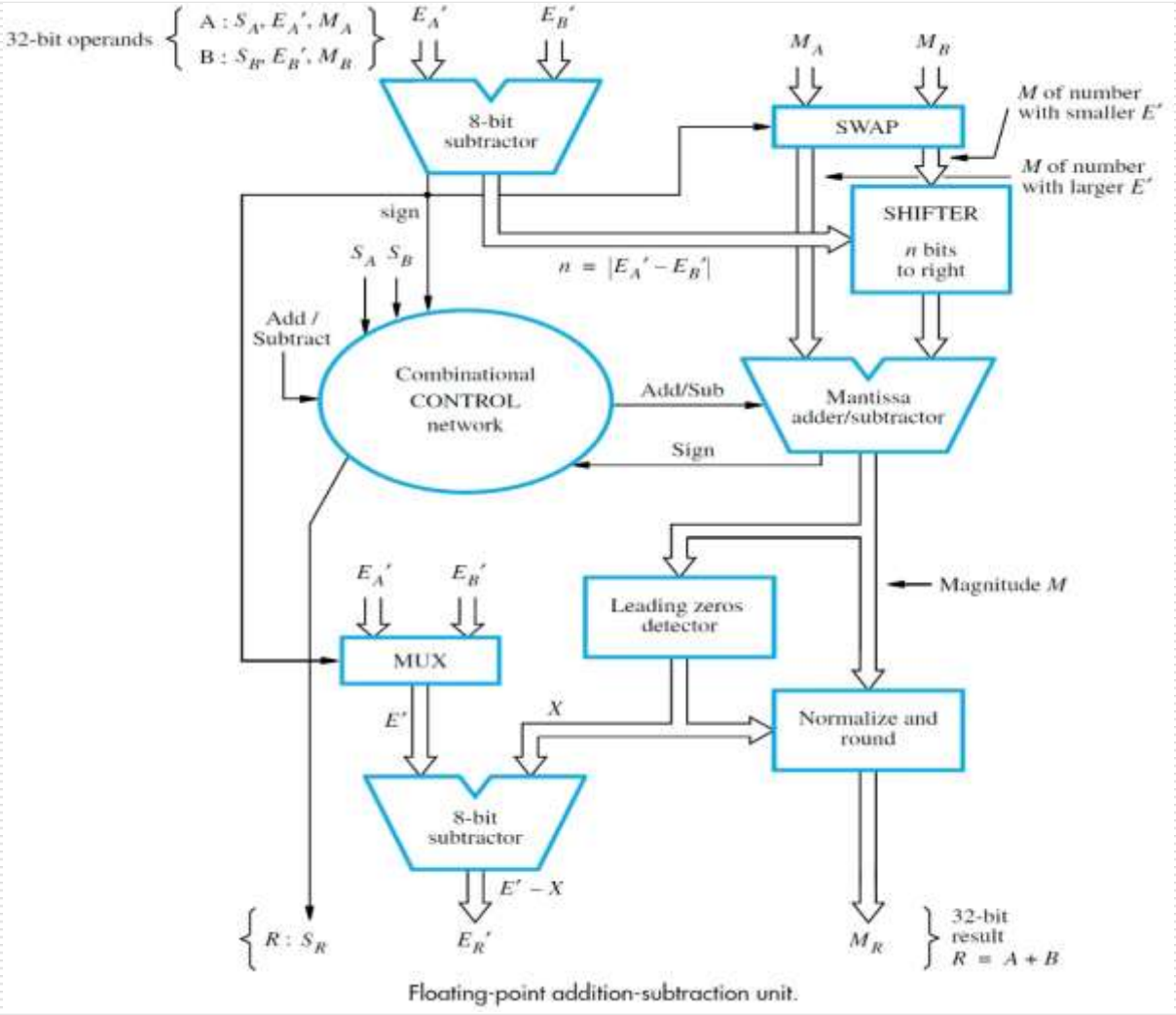
Example: Add 1.110×2^4 and 1.100×2^2

Alignment: 1.100×2^2 has to be aligned to 0.01100×2^4

Addition: Add two numbers to get 10.0010×2^4

Normalize: 1.00010×2^5

Implementing Floating-Point Operations



Floating-point addition-subtraction unit.

Arithmetic Operations on Floating-Point Numbers

Multiply Rule

1. Add the exponents and subtract 127 to maintain the excess-127 representation.

2. Multiply the mantissas and determine the sign of the result.

3. Normalize the resulting value, if necessary

Multiplication of a pair of Floating Point numbers

$X = m_x \times 2^a \qquad Y = m_y \times 2^b$

$X * Y = (m_x + m_y) \times 2^{(a+b)}$

Example:

Multiply $X = 1.00 \times 2^{-2}$ and $Y = -1.01 \times 2^{-1}$

Add Exponents: (-2) + (-1) = (-3) Subtract 127: (-3)-127 =-130

Multiply mantissa: 1.00 * -1.01 = -1.010000

Normalize: -1.0100×2^{-130}

Arithmetic Operations on Floating-Point Numbers

Divide Rule

1. Subtract the exponents and add 127 to maintain the excess-127 representation.

2. Divide the mantissas and determine the sign of the result.

3. Normalize the resulting value, if necessary

Divide of a pair of Floating Point numbers

$X = m_x \times 2^a \qquad Y = m_y \times 2^b$

$X / Y = (m_x / m_y) \times 2^{(a-b)}$

Example:

Divide $X = 1.0000 \times 2^{-2}$ and $Y = -1.0100 \times 2^{-1}$

Subtract Exponents: $(-2) - (-1) = (-1)$ Subtract 127: $(-1)+127 = 126$

Divide mantissa: $1.0000 / -1.0100 = -0.1101$

Result: -0.1101×2^{126}

Normalize: -1.101×2^{125}

Rough Slide to explain: Guard bits

Example of Subtracting two FP numbers

2.95×10^2

-2.39×10^0

We will allow two Guard digits

	2	.	9	5			x	10^2
-	0	.	0	2	3	9	x	10^2
	2	.	9	2	6	1	x	10^2

Rough Slide to explain: Truncation

Removing guard bits in generating a final result requires that the extended mantissa be truncated to create a 24-bit number that approximates the longer version.

Three different Ways of Truncations:

1. Chopping

0.000000 0.000111

2. Von Neumann rounding

0.000000 0.000001 0.000010 0.000100 0.000111

3. Rounding

0.000000 0.000001 0.000010 0.000100 0.000111

Guard Bits and Truncation

- Let us consider some important aspects of implementing the steps in the preceding algorithms. Although the mantissas of initial operands and final results are limited to 24 bits, including the implicit leading 1, it is important to retain extra bits, often called guard bits, during the intermediate steps. This yields maximum accuracy in the final results.
- Removing guard bits in generating a final result requires that the extended mantissa be truncated to create a 24-bit number that approximates the longer version. This operation also arises in other situations, for instance, in converting from decimal to binary numbers. We should mention that the general term rounding is also used for the truncation operation, but a more restrictive definition of rounding is used here as one of the forms of truncation. There are several ways to truncate. The simplest way is to remove the guard bits and make no changes in the retained bits. This is called **chopping**. Suppose we want to truncate a fraction from six to three bits by this method. All fractions in the range $0.b_{-1}b_{-2}b_{-3}000$ to $0.b_{-1}b_{-2}b_{-3}111$ are truncated to $0.b_{-1}b_{-2}b_{-3}$. The error in the 3-bit result ranges from 0 to 0.000111. In other words, the error in chopping ranges from 0 to almost 1 in the least significant position of the retained bits. In our example, this is the b_{-3} position. The result of chopping is a biased approximation because the error range is not symmetrical about 0.

Guard Bits and Truncation (Contd…)

The next simplest method of truncation is **von Neumann rounding**. If the bits to be removed are all 0s, they are simply dropped, with no changes to the retained bits. However, if any of the bits to be removed are 1, the least significant bit of the retained bits is set to 1. In our 6-bit to 3-bit truncation example, all 6-bit fractions with $b_{-4}b_{-5}b_{-6}$ not equal to 000 are truncated to $0.b_{-1}b_{-2}1$. The error in this truncation method ranges between −1 and +1 in the LSB position of the retained bits. Although the range of error is larger with this technique than it is with chopping, the maximum magnitude is the same, and the approximation is unbiased because the error range is symmetrical about 0.

Unbiased approximations are advantageous if many operands and operations are involved in generating a result, because positive errors tend to offset negative errors as the computation proceeds. Statistically, we can expect the results of a complex computation to be more accurate.

The third truncation method is a **rounding** procedure. Rounding achieves the closest approximation to the number being truncated and is an unbiased technique. The procedure is as follows: A 1 is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed. Thus, $0.b_{-1}b_{-2}b_{-3}1$ ... is rounded to $0.b_{-1}b_{-2}b_{-3}$ + 0.001, and $0.b_{-1}b_{-2}b_{-3}0$ ... is rounded to $0.b_{-1}b_{-2}b_{-3}$.

Thanks for Listening