# Module 4. Arithmetic

Dr. Sudhamani M J

CSE Dept.

RNSIT

## Outline

- Numbers - Arithmetic Operations and Characters
- Addition and Subtraction of Signed Numbers
- Design of Fast Adders
- Multiplication of Positive Numbers
- Signed Operand Multiplication
- Fast Multiplication
- Integer Division
- Floating-point Numbers and Operations

## Number, Arithmetic Operations, and Characters

- Signed Integer
- 3 major representations:

    Sign and magnitude

    One’s complement

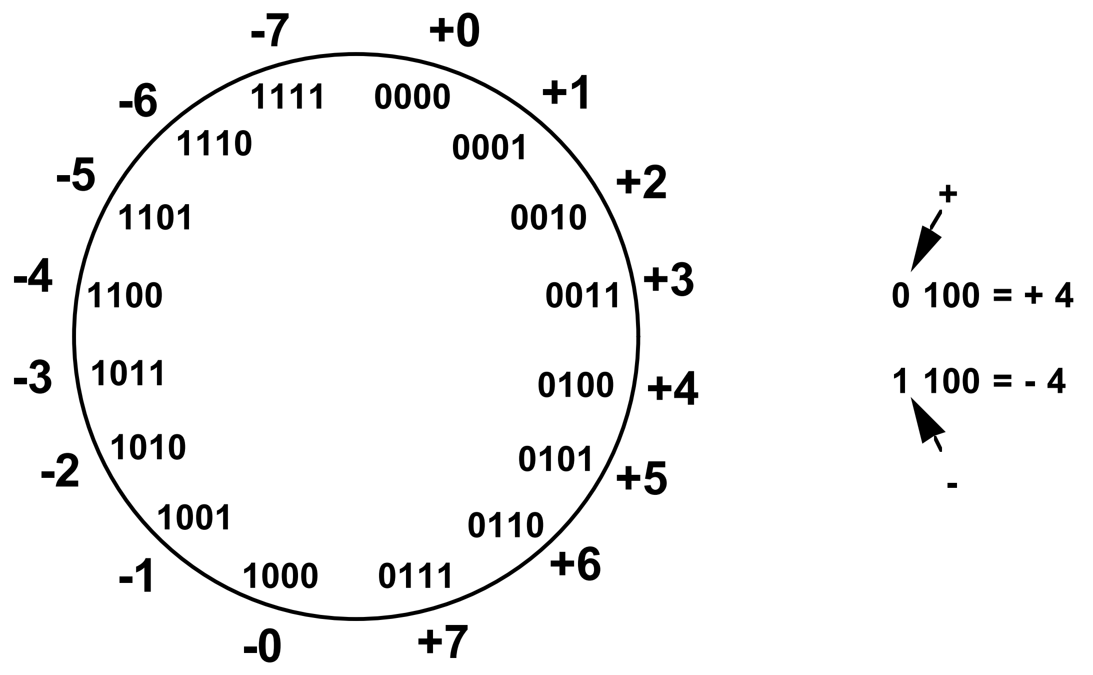
    Two’s complement

- Assumptions:

Using 4-bits16 different values can be represented.

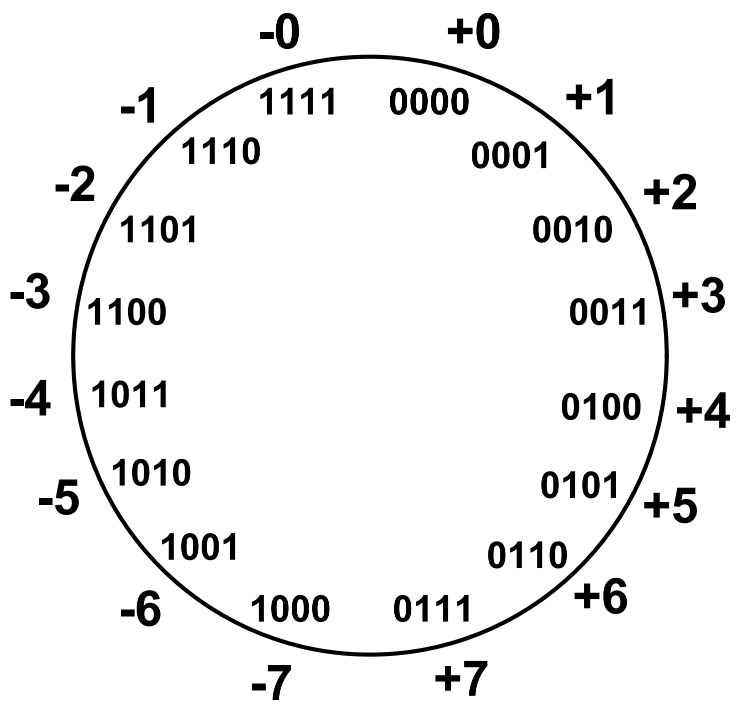
Roughly half are positive, half are negative

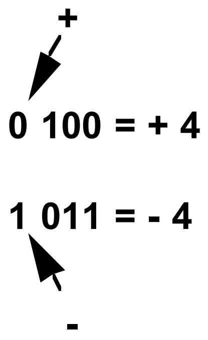
# Sign and Magnitude Representation



**High order bit is sign: 0 = positive (or zero), 1 = negative**
**Three low order bits is the magnitude: 0 (000) thru 7 (111)**
**Number range for n bits = +/-$2^{n-1}$ -1**
**Two representations for 0**

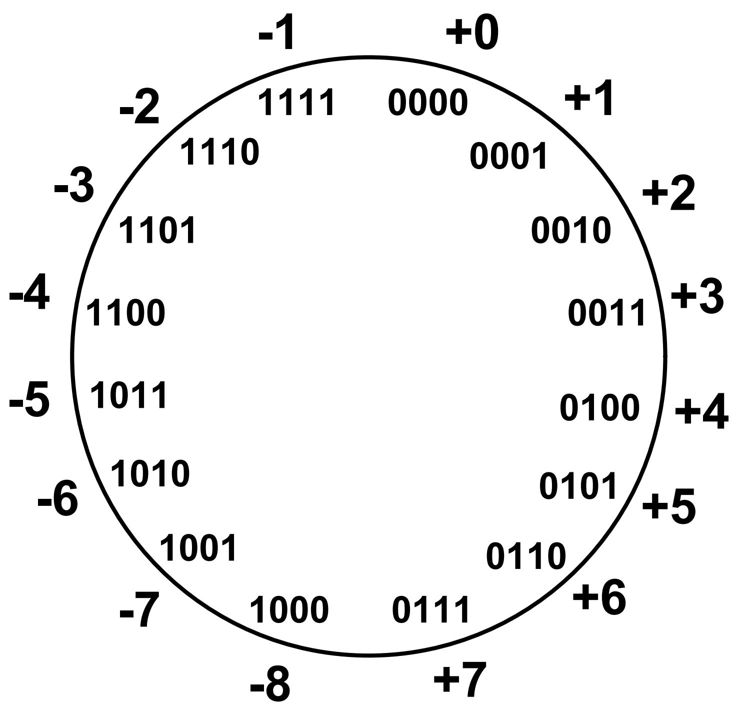
## One’s Complement Representation

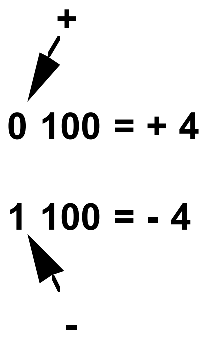




- Subtraction implemented by addition & 1's complement
- Still two representations of 0! This causes some problems
- Some complexities in addition

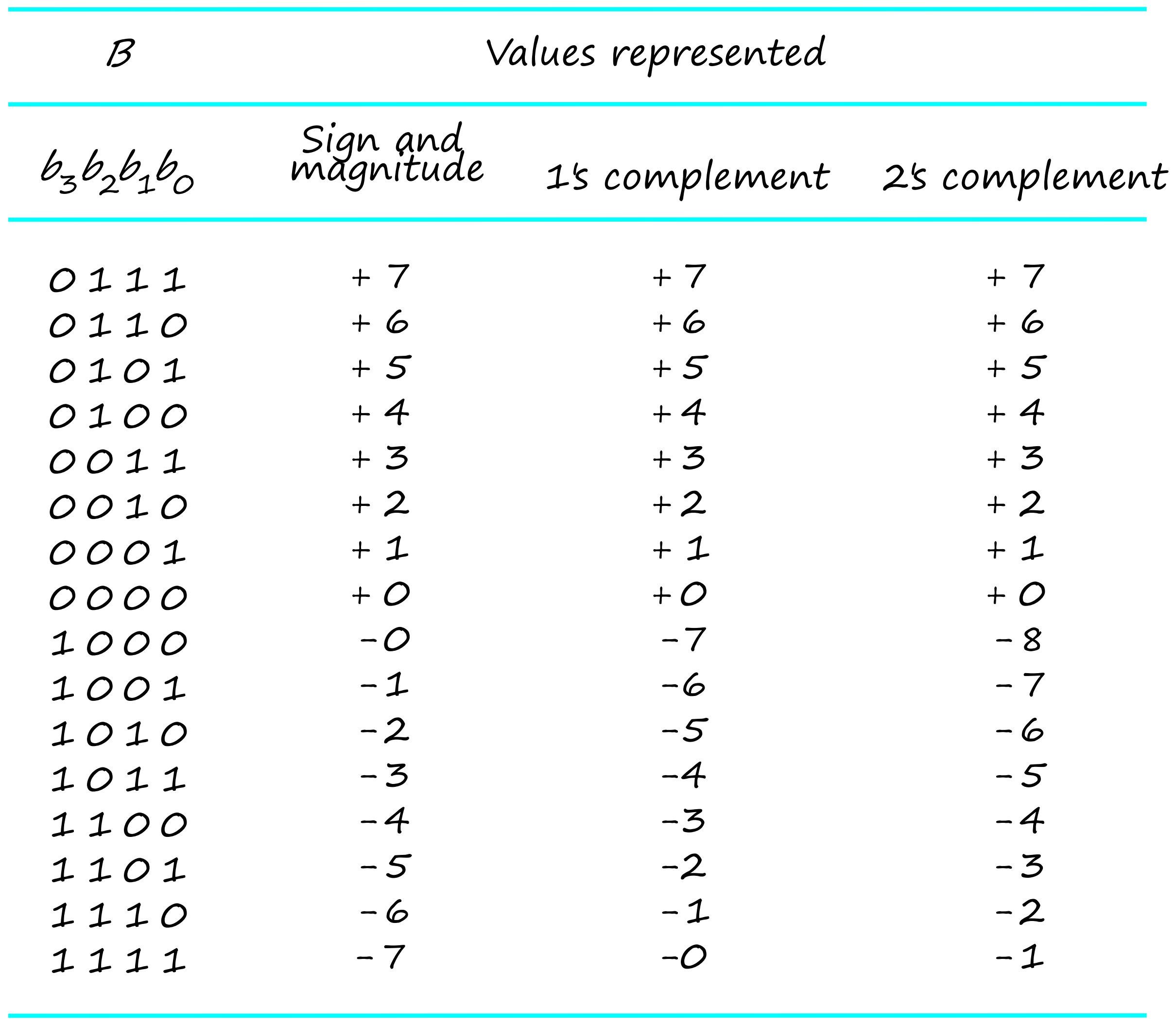
# Two’s Complement Representation





- Only one representation for 0
- One more negative number than positive number

# Binary, Signed-Integer Representations



| $B$ | Values represented | | |
|---|---|---|---|
| $b_3 b_2 b_1 b_0$ | Sign and magnitude | 1's complement | 2's complement |
| 0 1 1 1 | + 7 | + 7 | + 7 |
| 0 1 1 0 | + 6 | + 6 | + 6 |
| 0 1 0 1 | + 5 | + 5 | + 5 |
| 0 1 0 0 | + 4 | + 4 | + 4 |
| 0 0 1 1 | + 3 | + 3 | + 3 |
| 0 0 1 0 | + 2 | + 2 | + 2 |
| 0 0 0 1 | + 1 | + 1 | + 1 |
| 0 0 0 0 | + 0 | + 0 | + 0 |
| 1 0 0 0 | − 0 | − 7 | − 8 |
| 1 0 0 1 | − 1 | − 6 | − 7 |
| 1 0 1 0 | − 2 | − 5 | − 6 |
| 1 0 1 1 | − 3 | − 4 | − 5 |
| 1 1 0 0 | − 4 | − 3 | − 4 |
| 1 1 0 1 | − 5 | − 2 | − 3 |
| 1 1 1 0 | − 6 | − 1 | − 2 |
| 1 1 1 1 | − 7 | − 0 | − 1 |

Figure 2.1. Binary, signed-integer representations.

## Addition and subtraction of binary integers

1. To add two numbers, add their n-bit representations, ignoring the carry-out signal from the MSB position. The sum will be the algebraically correct value in the 2's complement representation as long as the answer is in the range $-2^{n-1}$ through $+2^{n-1}-1$.

2. To subtract two numbers X and Y, ie, to perform X-Y, form the 2's complement of Y and then add it to X, as in rule 1. the result will be algebraically correct value in the 2's complement representation as long as the answer is in the range $-2^{n-1}$ through $+2^{n-1}-1$.

In all these 4-bit examples the answer falls in the range between -8 to +7, if it do not fall within the representable range, **then it is called arithmetic overflow.**

## Examples for addition

```
a) 0010    (+2)        b)  0100  (+4)        c) 1011  (-5)
  +0011    (+3)           +1010  (-6)          +1110  (-2)
---------------           ------------------    ---------------------
   0101    (+5)             1110  (-2)           1001  (-7)
```

## Examples for subtraction

```
a) 1 1 0 1     (-3)                               1 1 0 1
   -1 0 0 1    (-7)   2's complement   ⇒        + 0 1 1 1
---------------                                 ----------
                                                  0 1 0 0  (+4)

b) 0 0 1 0     (+2)                               0 0 1 0
  - 0 1 0 0    (+4)   2's complement   ⇒          1 1 0 0
---------------                                 ----------
                                                  1 1 1 0  (-2)

   c)   1 0 0 1     (-7)                          1 0 0 1
       - 1 0 1 1    (-5)  2's complement   ⇒      0 1 0 1
   -----------------                            ----------
                                                  1 1 1 0  (-2)
```

# Addition/subtraction 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 |

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

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

At the $i^{th}$ stage:

Input:

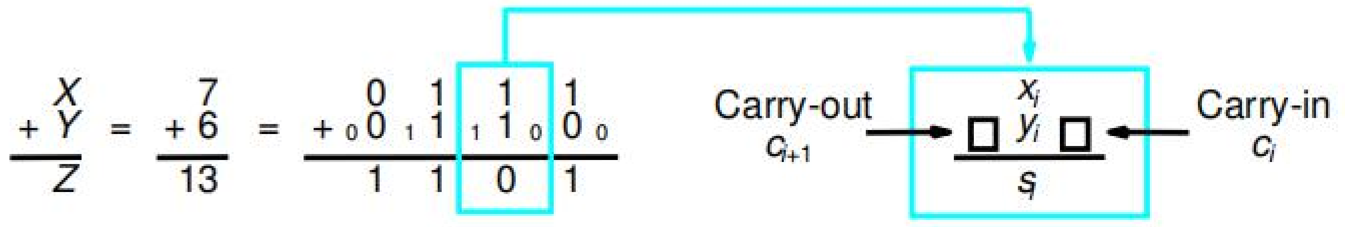
$c_i$ is the carry-in

Output:

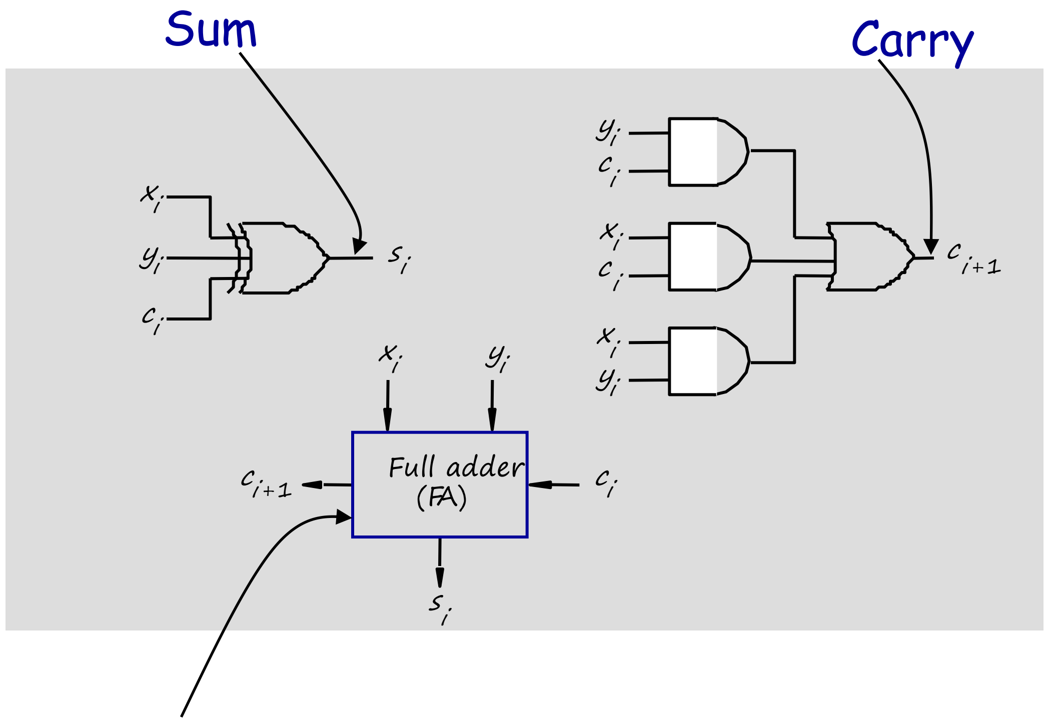
$s_i$ is the sum

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

Example:



# Addition logic for a single stage

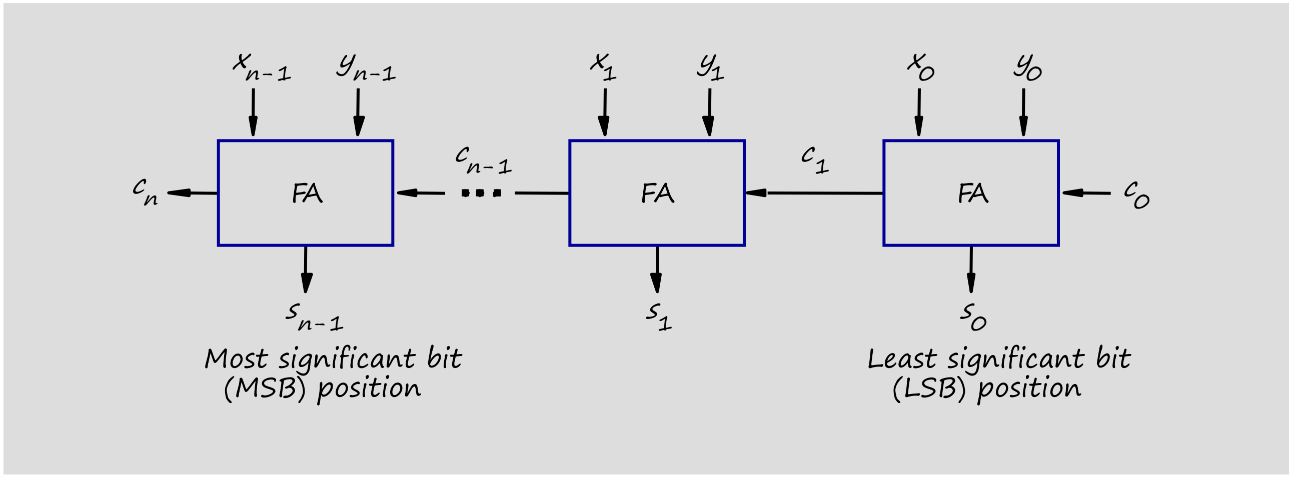


Full Adder (FA): Symbol for the complete circuit for a single stage of addition.

Full adder is combinational logical circuit that performs an addition operation on three one-bit binary numbers.

# *n*-bit adder

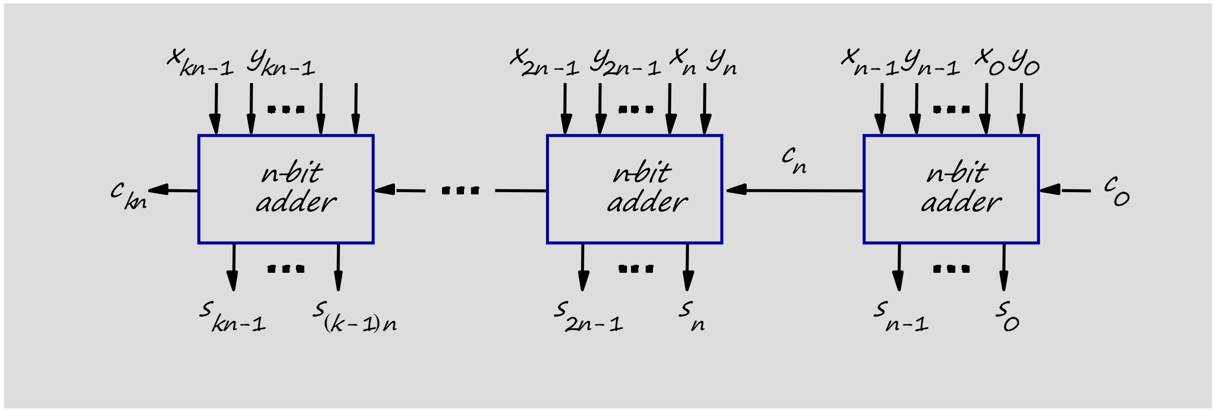
- Cascade *n* full adder (FA) blocks to form a *n*-bit adder.
- Carries propagate or ripple through this cascade, *n*-bit ripple carry adder.



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

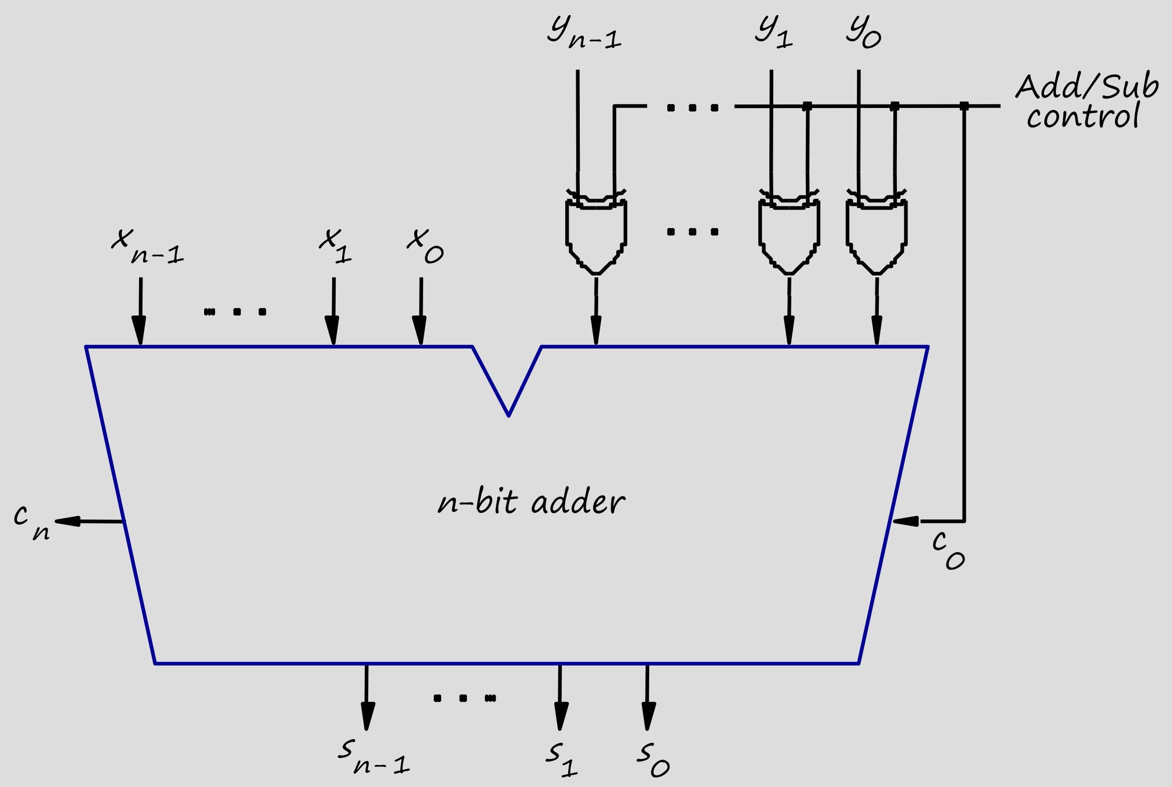
# $K$ $n$-bit adder

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



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

# $n$-bit adder/subtractor (contd..)

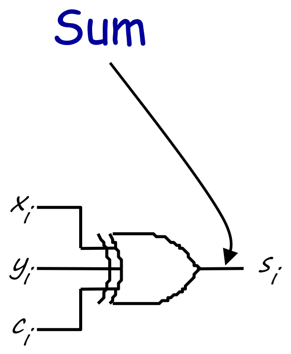


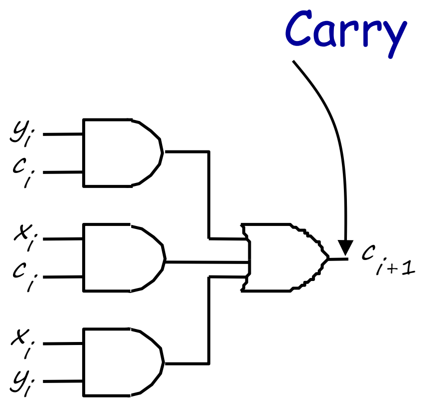
- Add/sub control = 0, addition.
- Add/sub control = 1, subtraction.

note : XOR gate complements the bits

## Finding the gate delays

- The delay through a logic network depends on IC technology used in fabrication and on the number of gates in the paths from input to output
- Sum has 1 gate delay (XOR gate) and Carry has 2 gate delays (AND +OR gates) in FA



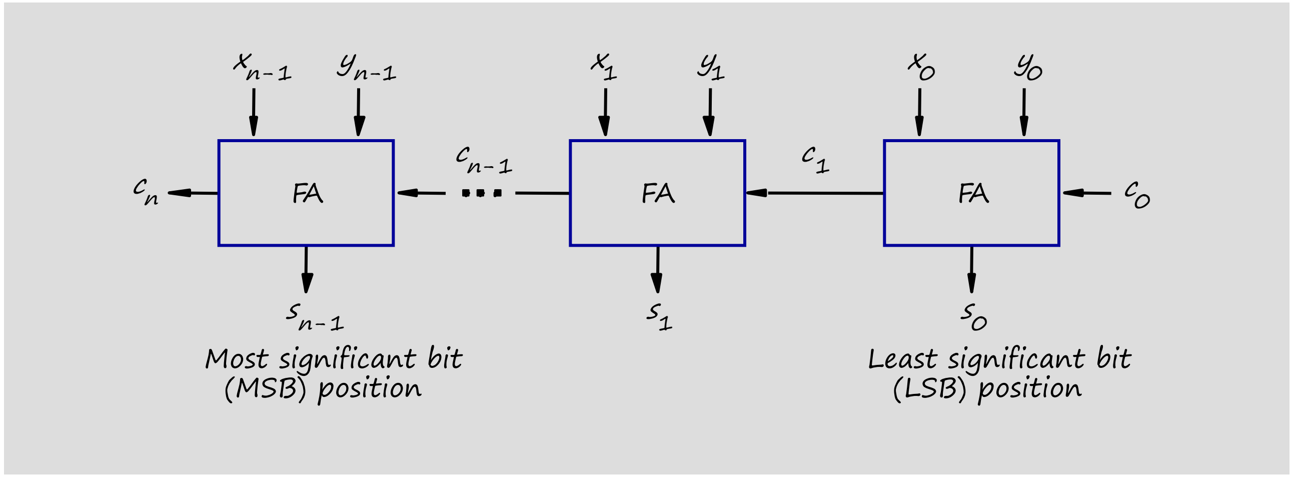


# Gate delay in n-bit ripple carry adder

- Cn-1 is available in 2(n-1) gate delay
  [ 1 for AND gate+1 for OR gate = 2 delay * (n-1) stages =>2(n-1)]

- Cn is available in by adding 2 gate delays to Cn-1 =>2n-2+2= 2n gate delays

- Sn-1 is one XOR gate delay after Cn-1 is obtained
- i.e 2n-2+1=2n-1 gate delays



## 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

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

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

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

*continuing*

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

*until*

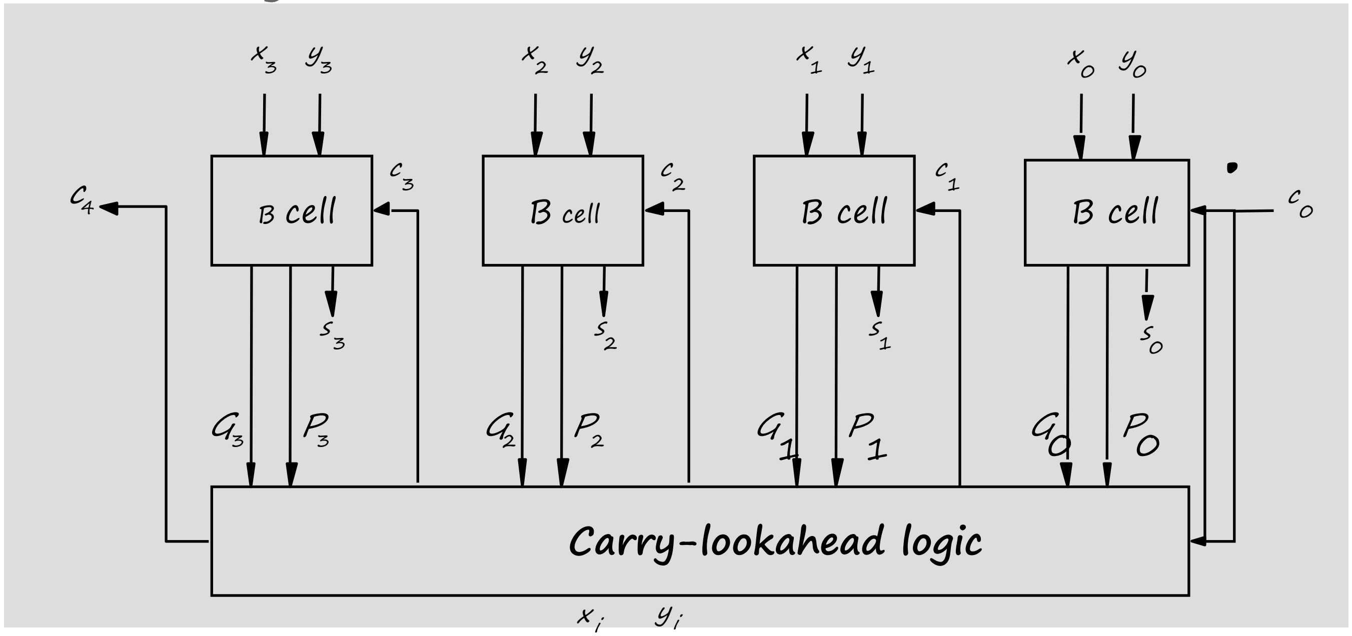
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + .. + P_i P_{i-1} .. P_1 G_0 + P_i P_{i-1} ... P_0 c_0$$

- All carries can be obtained 3 gate delays after $X$, $Y$ and $c_0$ are applied.
  - One gate delay for $P_i$ and $G_i$
  - Two gate delays in the AND-OR circuit for $c_{i+1}$
- All sums can be obtained 1 gate delay after the carries are computed.

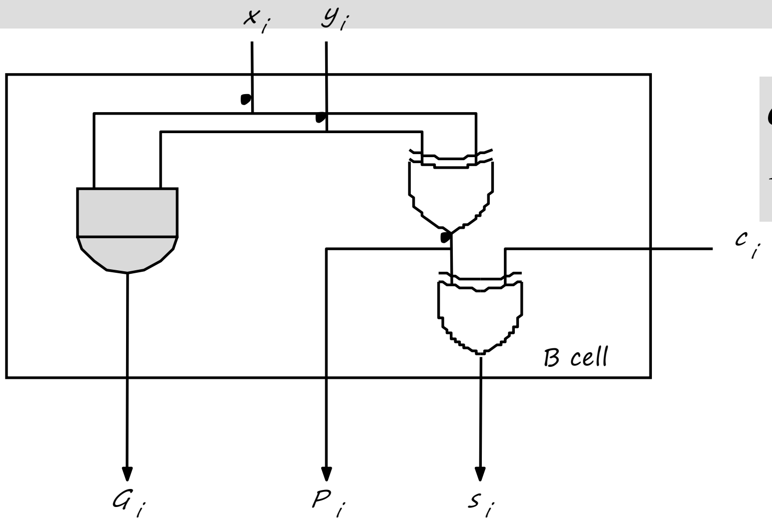
- Independent of $n$, $n$-bit addition requires only 4 gate delays.

- This is called Carry Lookahead adder.

# Carry-lookahead adder



**4-bit carry-lookahead adder**



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

$$\text{where } G_i = x_i y_i \text{ and } P_i = x_i + y_i$$

**B-cell for a single stage**

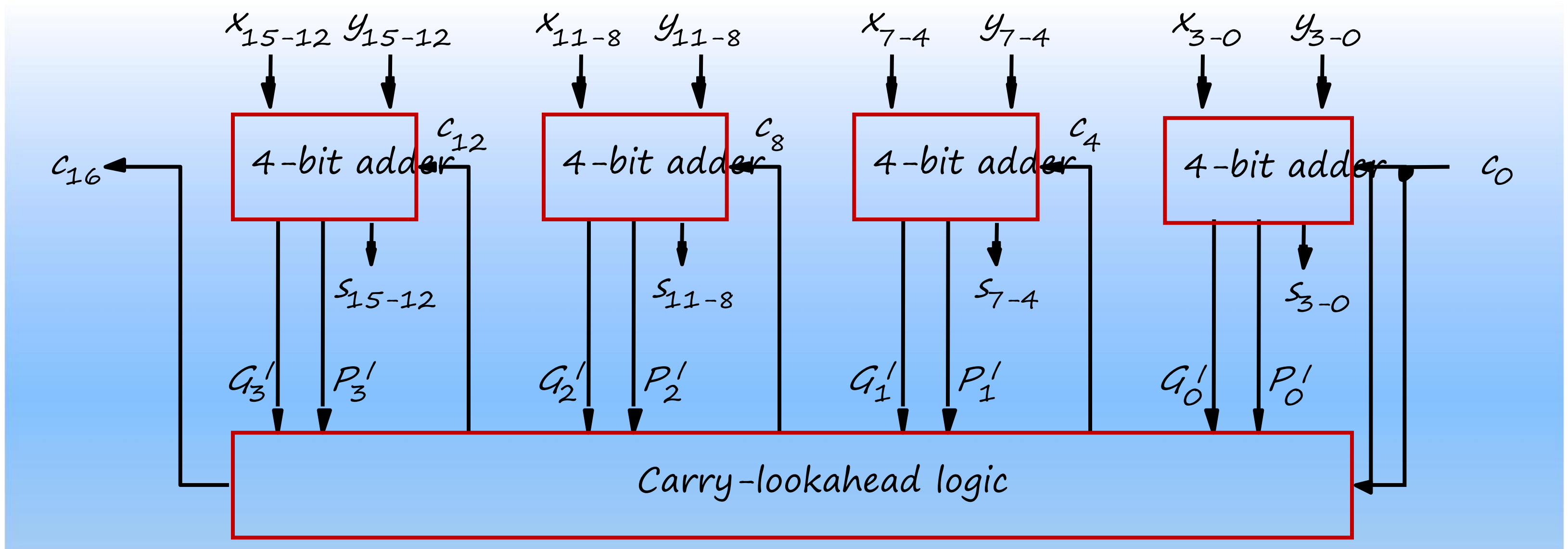
## Carry lookahead adder (contd..)

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

$$c_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + .. + P_iP_{i-1}..P_1G_0 + P_iP_{i-1}...P_0c_0$$

- Last AND gate and OR gate require a fan-in of (n+1) for a n-bit adder.
  - For a 4-bit adder (*n=4*) fan-in of 5 is required.
  - Practical limit for most gates.
- In order to add operands longer than 4 bits, we can cascade 4-bit Carry-Lookahead adders. Cascade of Carry-Lookahead adders is called Blocked Carry-Lookahead adder.

# 16 bit Carry-Lookahead adder

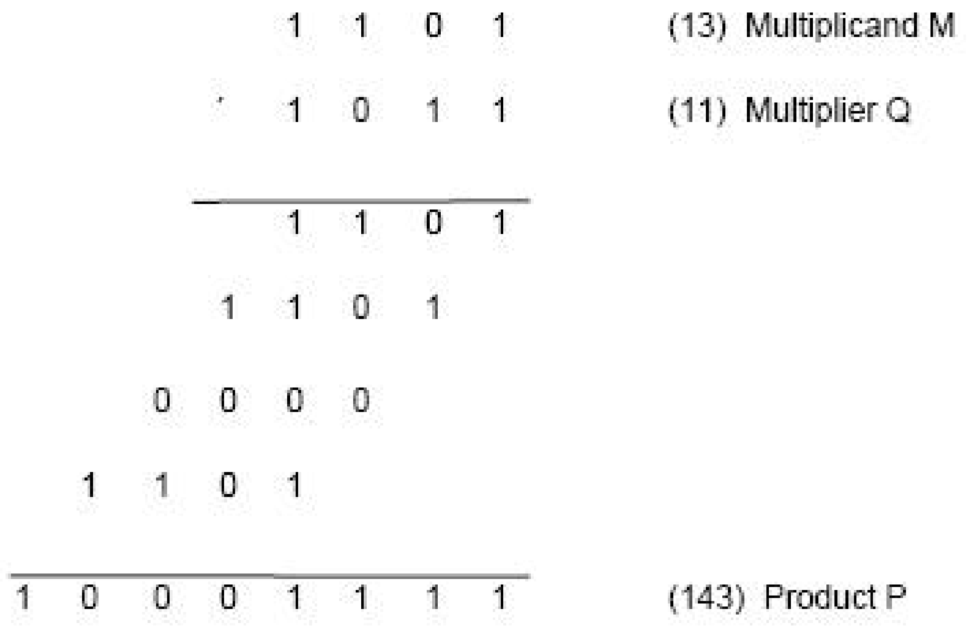


After $x_i$, $y_i$ and $c_0$ are applied as inputs:

- $G_i$ and $P_i$ for each stage are available after 1 gate delay.
- $P^I$ is available after 2 and $G^I$ after 3 gate delays.
- All carries are available after 5 gate delays.
- $c_{16}$ is available after 5 gate delays.
- $s_{15}$ which depends on $c_{12}$ is available after 8 (5+3)gate delays
  (Recall that for a 4-bit carry lookahead adder, the last sum bit is available 3 gate delays after all inputs are available)

## Multiplication

# Multiplication of unsigned numbers



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

Product of 2 *n*-bit numbers is at most a 2*n*-bit number.

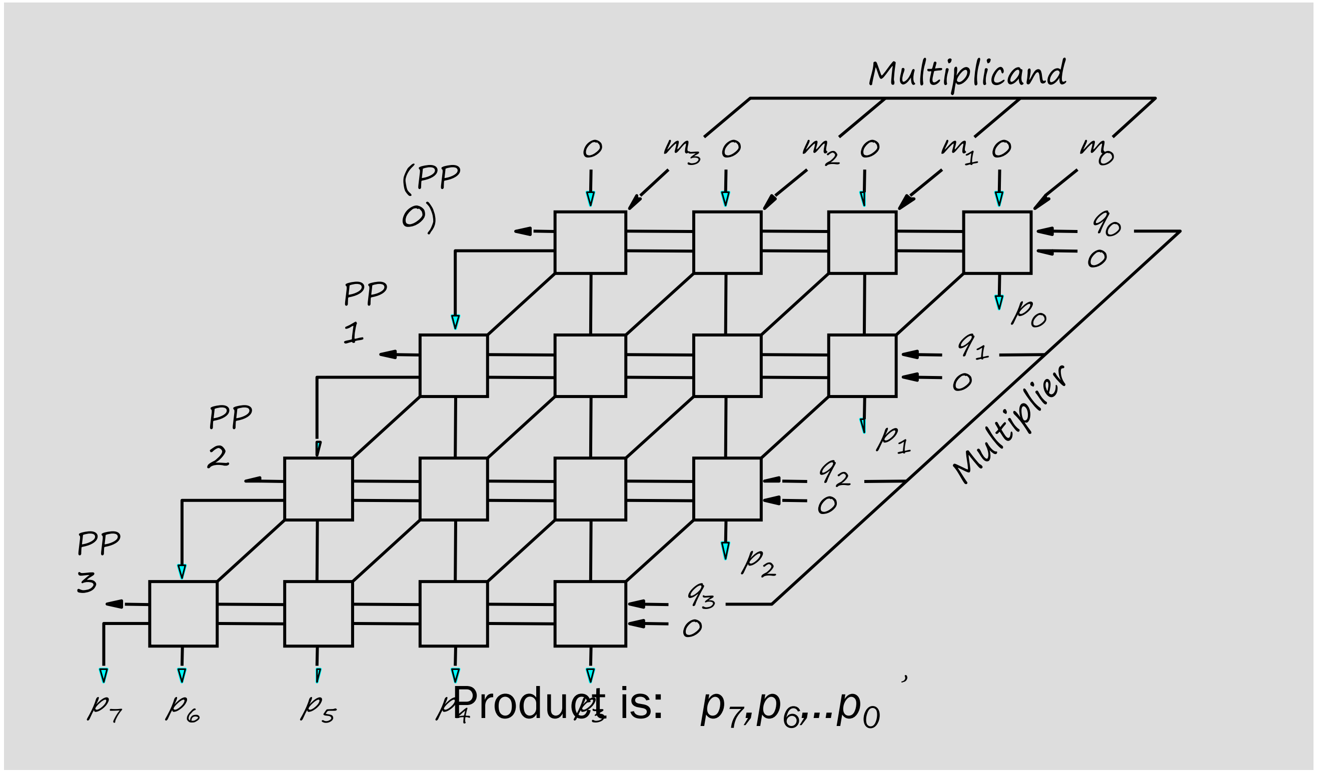
Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand.

## Multiplication of unsigned numbers (contd..)

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

# Combinatorial array multiplier

Combinatorial array multiplier

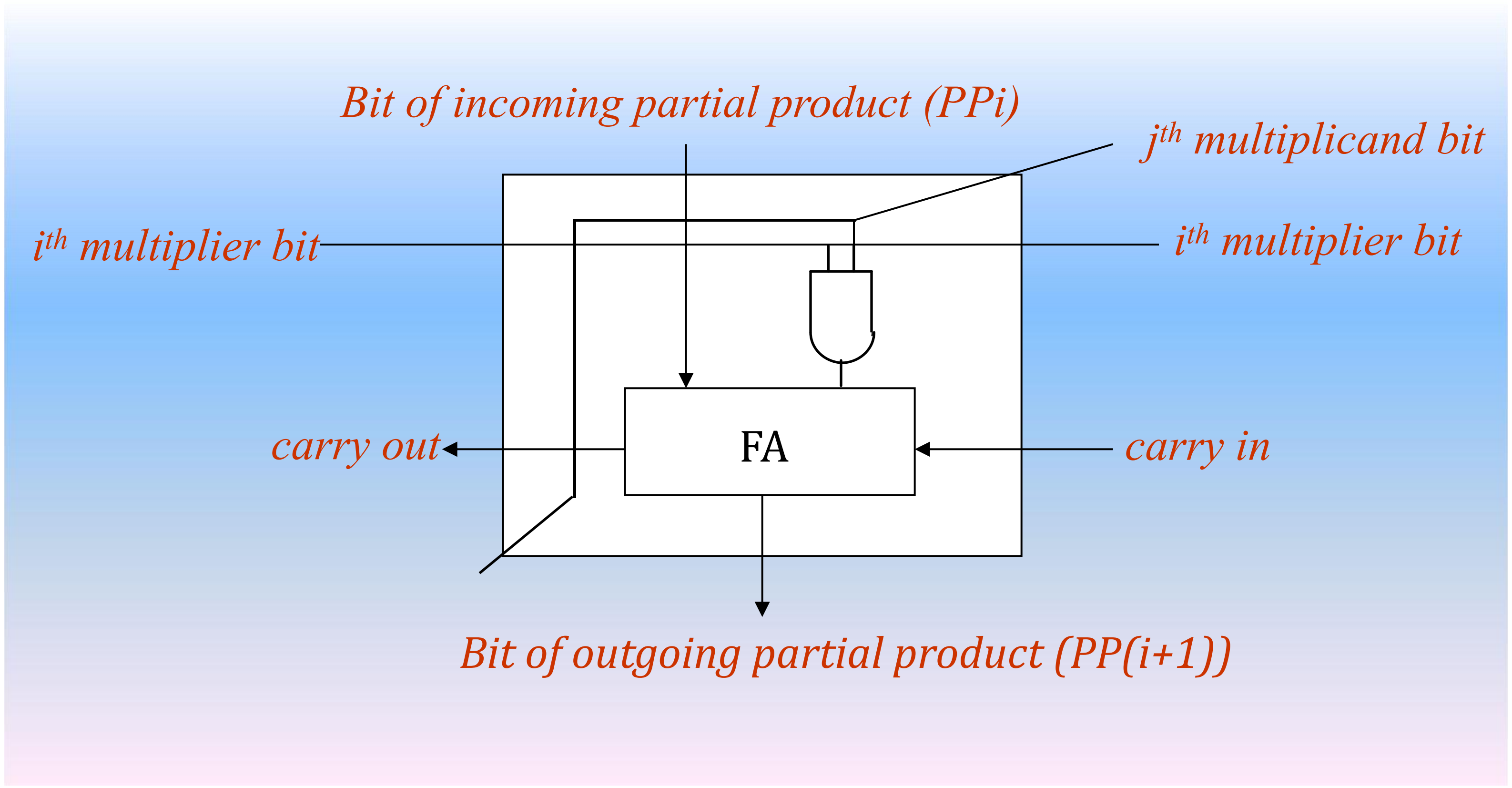


Product is: $p_7, p_6, .. p_0$

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

# Multiplication of unsigned numbers

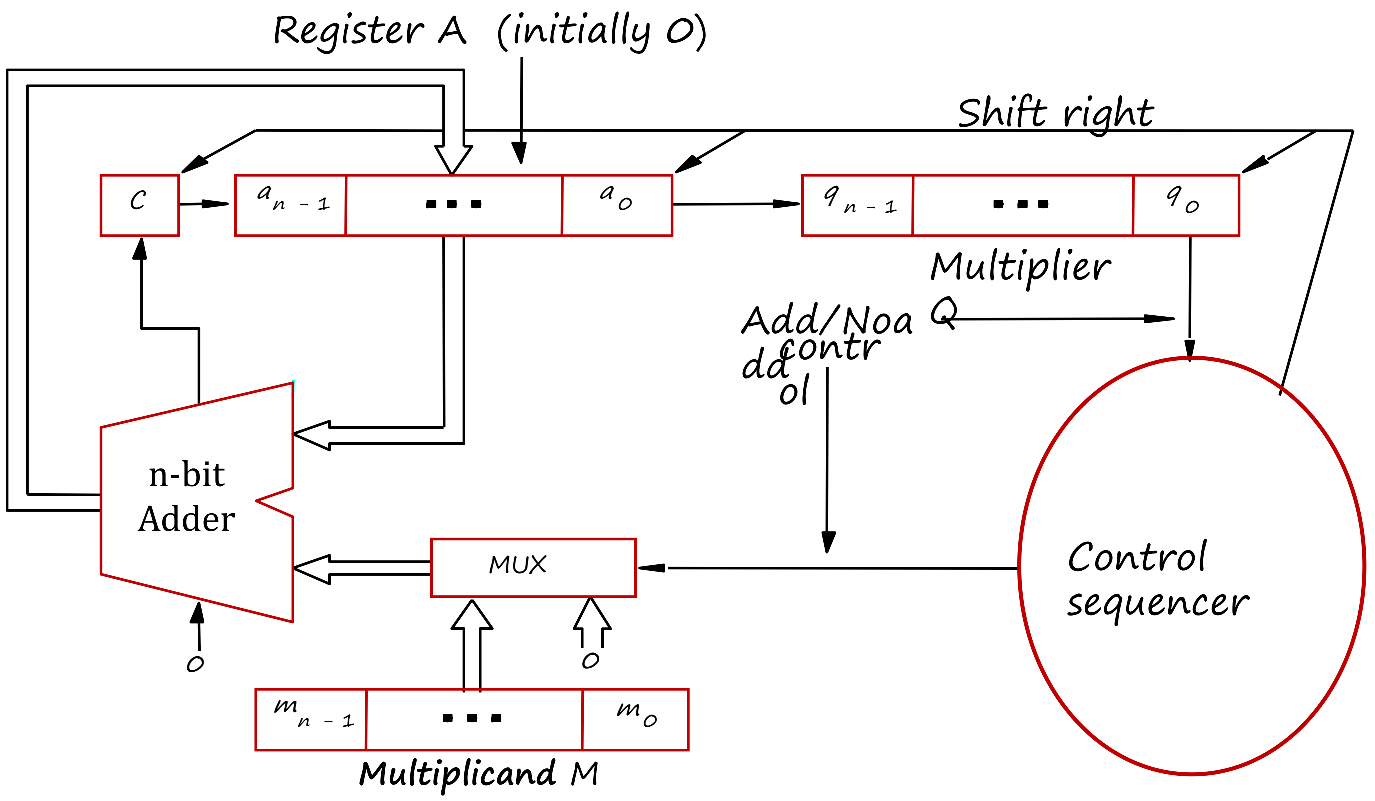
Typical multiplication cell


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

## Combinatorial array multiplier (contd..)

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

# Sequential multiplication



## Algorithm

```
Load
multiplier into Register Q, Multiplicand into register M
Clear C and A

Repeat the following n-times
Step1: if Q0 is 1do the following:
        1) Add M to A and keep the result in A
        2) shift C, A and Q right one bit position
           if Q0 is 1 goto step1; otherwise step 2
step2: if Q0 is 0, do not add M to A , just shift C,A and Q one bit
positon
```

# Sequential multiplication (contd..)

M

$\boxed{1\ 1\ 0\ 1}$

| C | A | Q | | |
|---|---|---|---|---|
| $\boxed{0}$ | $\boxed{0\ 0\ 0\ 0}$ | $\boxed{1\ 0\ 1\ 1}$ | | Initial configuration |
| 0 | 1 1 0 1 | 1 0 1 1 | Add | First cycle |
| 0 | 0 1 1 0 | 1 1 0 1 | Shift | |
| 1 | 0 0 1 1 | 1 1 0 1 | Add | Second cycle |
| 0 | 1 0 0 1 | 1 1 1 0 | Shift | |
| 0 | 1 0 0 1 | 1 1 1 0 | No add | Third cycle |
| 0 | 0 1 0 0 | 1 1 1 1 | Shift | |
| 1 | 0 0 0 1 | 1 1 1 1 | Add | Fourth cycle |
| 0 | 1 0 0 0 | 1 1 1 1 | Shift | |

$\underbrace{1\ 0\ 0\ 0\quad 1\ 1\ 1\ 1}_{\text{Product}}$

## Signed Multiplication

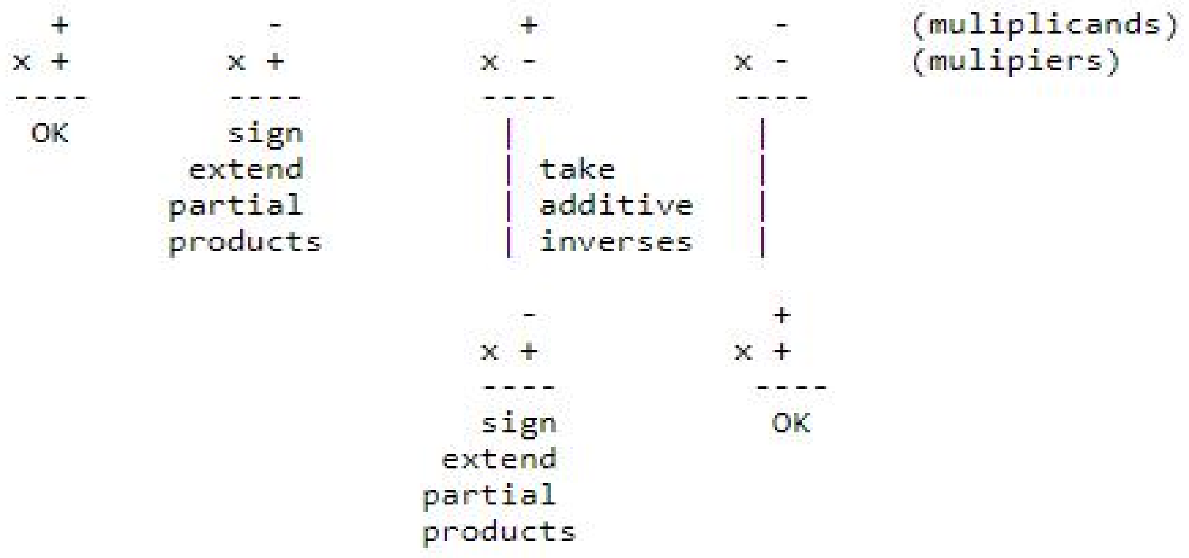
# Signed Multiplication

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

```
                                        1  0  0  1  1   (-13)
                                        0  1  0  1  1   (+11)
                                       ---------------
                         1  1  1  1  1  1  0  0  1  1
                         1  1  1  1  1  0  0  1  1
Sign extension is        0  0  0  0  0  0  0  0
shown in blue            1  1  1  0  0  1  1
                         0  0  0  0  0  0
                        ------------------------------
                         1  1  0  1  1  1  0  0  0  1   (-143)
```

*Sign extension of negative multiplicand.*

# Signed Multiplication



```
   +          -          +          -     (muliplicands)
 x +        x +        x -        x -     (mulipiers)
 ----       ----       ----       ----
  OK         sign        |  take      |
           extend        |  additive  |
          partial        |  inverses  |
          products
                           -          +
                         x +        x +
                         ----        ----
                           sign       OK
                         extend
                        partial
                        products
```

- A technique that works equally well for both negative and positive multipliers – Booth algorithm.

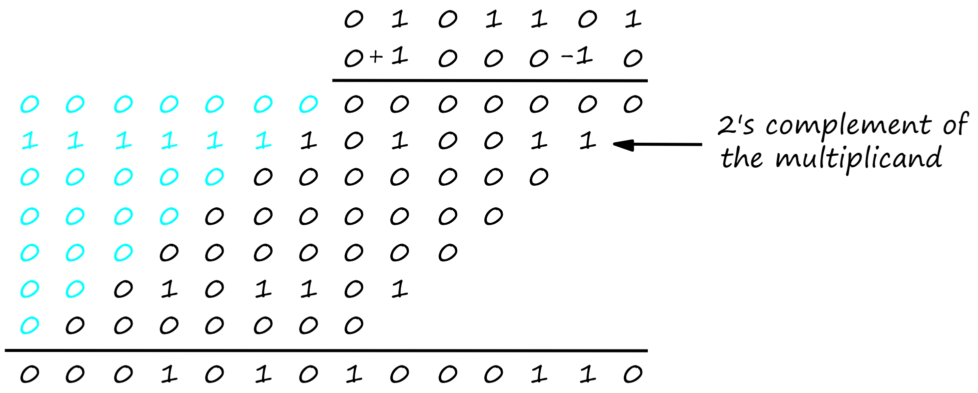
# Booth Algorithm

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

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

# Booth Algorithm

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



```
                            0  1  0  1  1  0  1
                            0 +1  0  0  0 -1  0
                            -------------------
   0  0  0  0  0  0  0  0  0  0  0  0  0  0
   1  1  1  1  1  1  1  0  1  0  0  1  1     <---- 2's complement of
   0  0  0  0  0  0  0  0  0  0  0  0              the multiplicand
   0  0  0  0  0  0  0  0  0  0  0
   0  0  0  0  0  0  0  0  0  0
   0  0  0  1  0  1  1  0  1
   0  0  0  0  0  0  0  0
  -------------------------------------------
   0  0  0  1  0  1  0  1  0  0  0  1  1  0
```

## Booth Algorithm

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

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

Booth recoding of a multiplier.

# Booth Algorithm

$$
\begin{array}{rcccccl}
 & 0 & 1 & 1 & 0 & 1 & (+13) \\
\times & 1 & 1 & 0 & 1 & 0 & (-6) \\
\hline
\end{array}
\quad\Longrightarrow\quad
\begin{array}{cccccccccccl}
 & & & & & 0 & 1 & 1 & 0 & 1 & \\
 & & & & & 0 & -1 & +1 & -1 & 0 & \\
\cline{6-10}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & & \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & & \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & (-78)
\end{array}
$$

*Booth multiplication with a negative multiplier.*

## Booth Algorithm

| Multiplier | | Version of multiplicand selected by bit |
|---|---|---|
| Bit i | Bit i-1 | |
| 0 | 0 | 0 X M |
| 0 | 1 | +1 X M |
| 1 | 0 | −1 X M |
| 1 | 1 | 0 X M |

Booth multiplier recoding table.

# Booth Algorithm

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

*Worst-case multiplier*

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

$\Downarrow$

+1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1 +1 -1

*Ordinary multiplier*

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

$\Downarrow$

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

*Good multiplier*

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

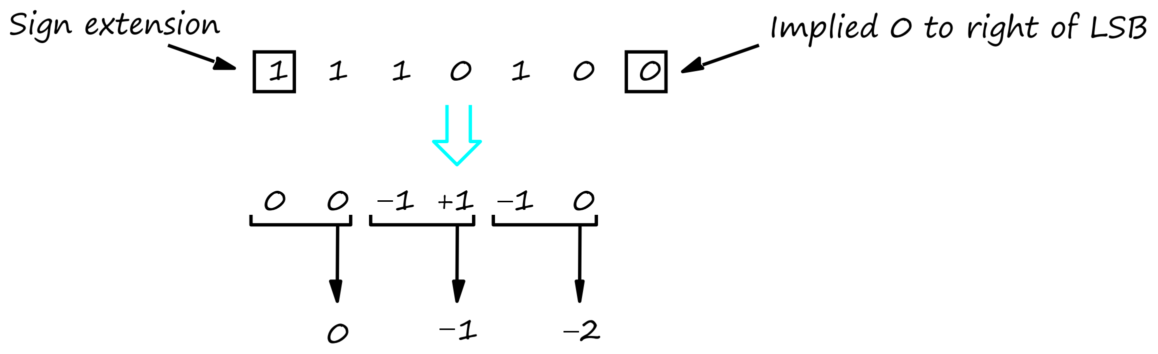
$\Downarrow$

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

## Fast Multiplication

# Bit-Pair Recoding of Multipliers

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



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

## Bit-Pair Recoding of Multipliers

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

(b) Table of multiplicand selection decisions

# Bit-Pair Recoding of Multipliers

01101 (+13) X
11010 (-6)

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

Mul by -2 => add the number twice
-13= 10011
-13= 10011+
---------------------
-26= 1 00110

```
                0  1  1  0  1
                0     -1    -2
               --------------
 1  1  1  1  1  0  0  1  1  0
 1  1  1  1  0  0  1  1
 0  0  0  0  0  0
-----------------------------
 1  1  1  0  1  1  0  0  1  0
```

Figure 6.15. Multiplication requiring only n/2 summa

## solve using bit pair recoding

$A = 010111 \ (+23)$

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

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

```
                0 1  0 1 1  1          23
            ×   −1    +2  −2         ×−10 (Booth recoding 0 −1 +1  0  −1  0)
   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
   1 1 1 1 0 0 0 1 1  0  1  0       =−230
```

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

$$\underbrace{1\,1}\,\underbrace{0\,1}\,\underbrace{1\,0\;0}$$

−1 +2 −2

$$\underbrace{0\;-1}\;\underbrace{+1\;0}\;\underbrace{-1\;0}$$

$-2^0 \qquad +2^1 \qquad -2^1$

$-1 \qquad +2 \qquad -2$

Consider the following binary numbers:

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

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

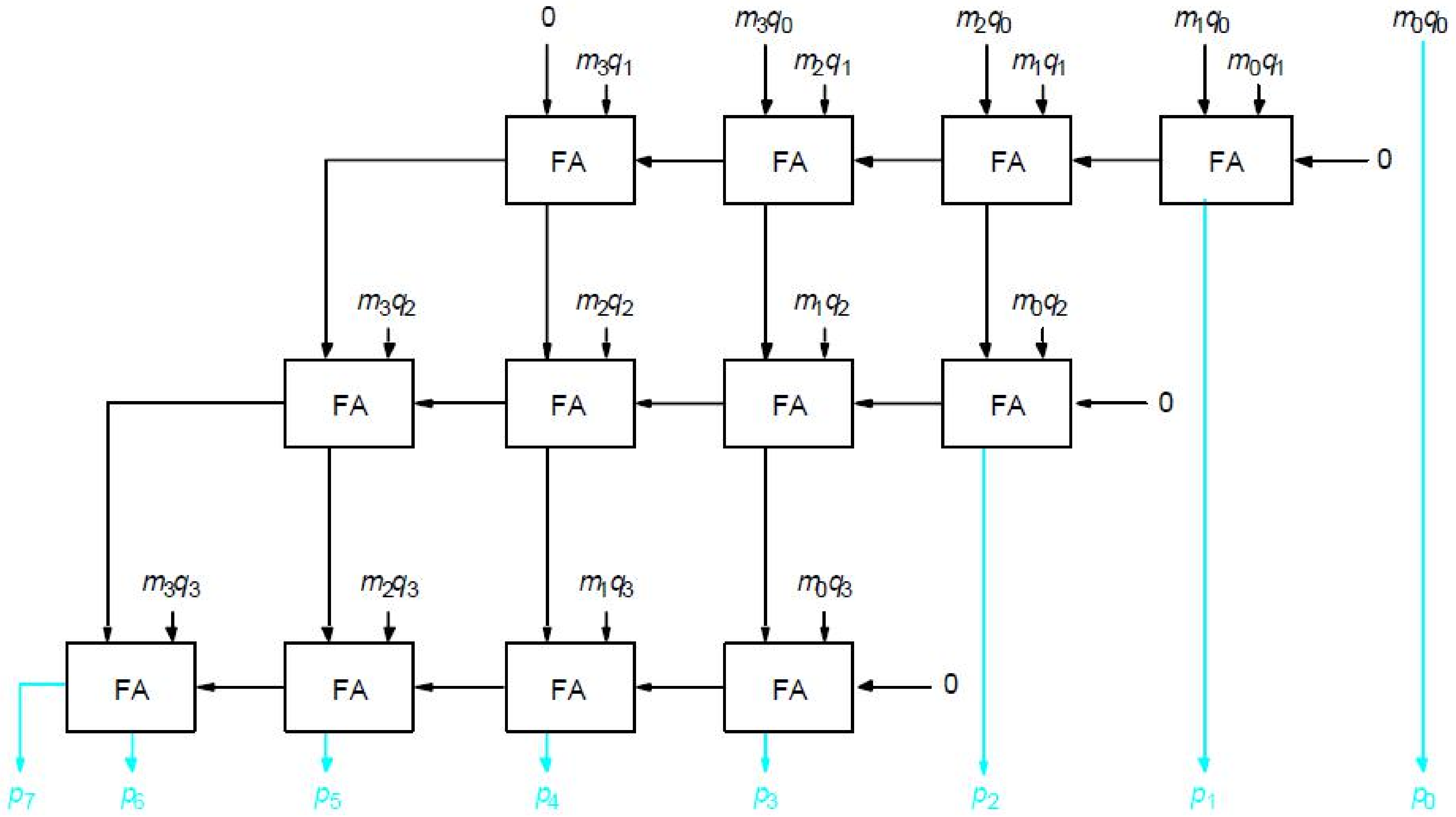
Multiply the signed 2's complement numbers using the bit-pair recoding of the multiplier.

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

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

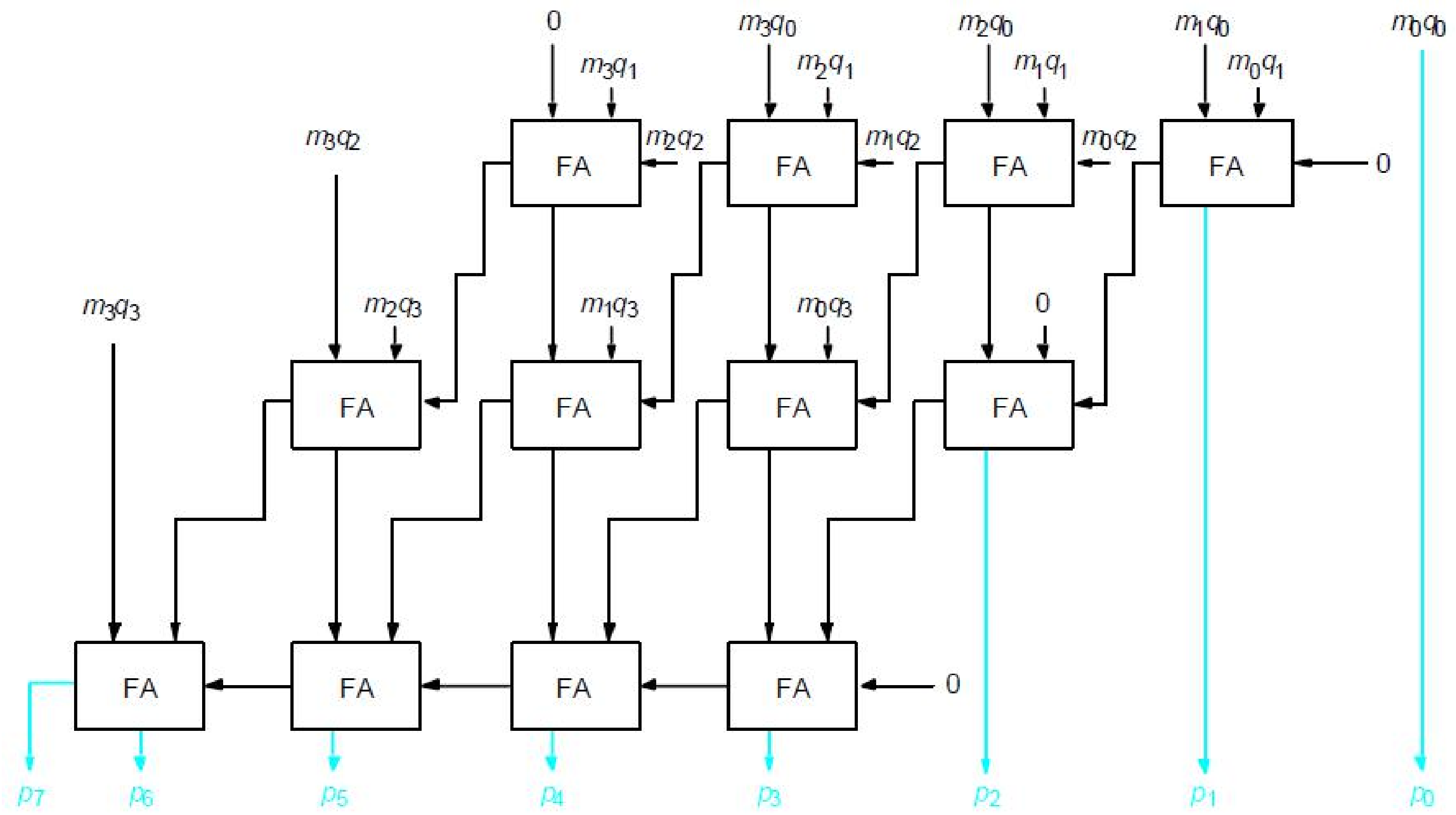
## Carry-Save Addition of Summands

- Carry save addition speeds up the addition process.
- In CSA, instead of letting the carries ripple along the rows, they can be saved and introduced into next row, at correct weighted positions.
- Consider the array for 4x4 multiplication
- First row consisting of just the AND gates that implement the bit products m3q0,m2q0,m1q0 and m0q0 .


0
$m_3q_0$
$m_2q_0$
$m_1q_0$
$m_0q_0$
$m_3q_1$
$m_2q_1$
$m_1q_1$
$m_0q_1$
FA
FA
FA
FA
0
$m_3q_2$
$m_2q_2$
$m_1q_2$
$m_0q_2$
FA
FA
FA
FA
0
$m_3q_3$
$m_2q_3$
$m_1q_3$
$m_0q_3$
FA
FA
FA
FA
0
$p_7$
$p_6$
$p_5$
$p_4$
$p_3$
$p_2$
$p_1$
$p_0$

## Carry-Save Addition of Summands(Cont.,)

- We see that carry from each stage ripples to the next stage delaying the PP terms
- Using CSA method, we can save carry and introduce them in the next row, at the correct weighted position
- This technique reduces the delay and frees up an input to the three full adders in the first row.



Two inputs of each full adder in the second row are the sum and carry outputs of the first row

P0

## Carry-Save Addition of Summands(Cont.,)

- Consider the addition of many summands, 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
  - Group all of 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 full-adder delay
  - Continue with this process until there are only two vectors remaining
  - They can be added in a RCA or CLA to produce the desired product

# Carry-Save Addition of Summands

```
              1 0 1 1 0 1      (45)      M
            x 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 6.17. A multiplication example used to illustrate carry-save addition as shown in Figure 6.18.

## how to speed up?

- Using CSA principle, we can make group of three summands at a time , perform carry save addition in parrallel using one full adder delay.(generates set of S and C)
- Group all generated S and C vectors in threes perform carry save addition on them to generate another set of S and C vector with one more FA delay.
- The process should continue until only two vectors remain to be added, which can be added in a carry lookahead adder to produce the desired product

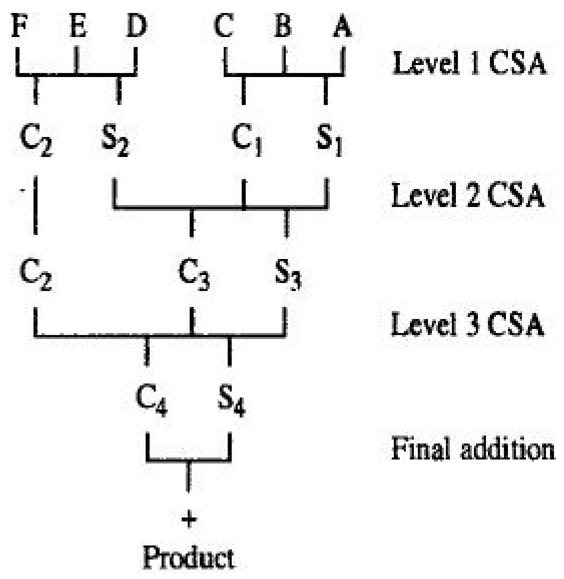


**Figure 6.19** Schematic representation of the carry-save addition

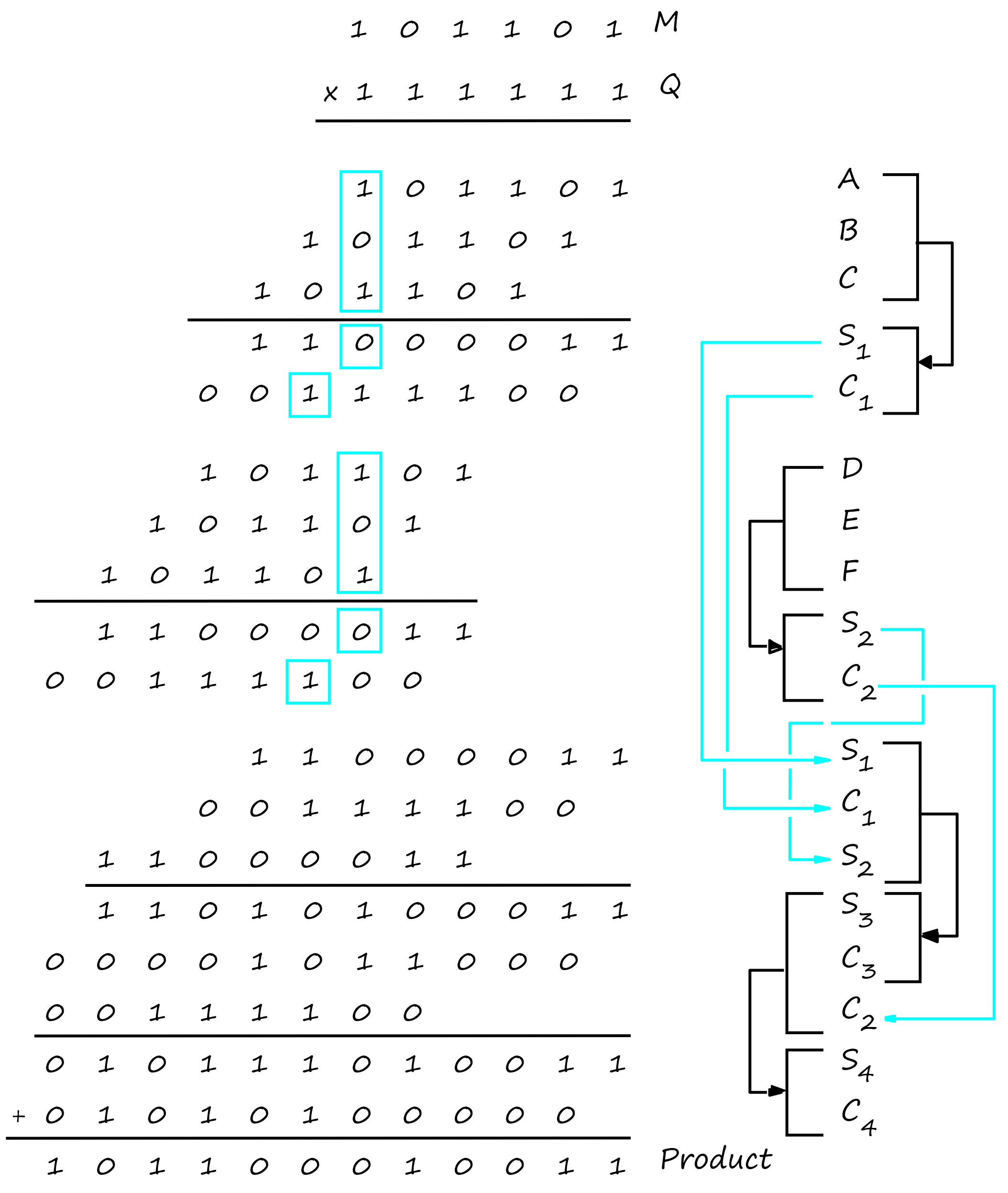


Figure 6.18. The multiplication example from Figure 6.17 performed using carry-save addition.

## Delay calculation

- Delay:
- 1 AND gate elay to generate all six summands at level 1
- 2 gate delays per CSA level till S4 and C4

  =6 gate delays
- CLA gate delay =8
- Total delay for 6 bit number is 1+6+8=15. If each gate delay is 10 ns, then CSA needs 150ns delay.
- array multiplier needs require 6(n-1)-1 =29 gate delay
- So When the number of summands is large, the time saved is proportionally much greaterin CSA.

## Integer Division

- Two methods:

  Restoring &

  Non restoring scheme

## Restoring Division

**I/P:**

Register M is loaded with n-bit divisor (SIZE OF THE REGISTER IS n+1)
Register Q is loaded with n-bit dividend (size is n)
Register A is initially set to 0 (size =n+1)

**O/P**

The final quotient will be stored in n bit Q register.
The final reminder will be stored in n-bit “A” register.

### Algorithm

1. Shift A and Q left one binary position
2. Subtract M from A, and place the answer back in A
3. If the sign of A is 1, set $q_0$ to 0 and add M back to A (restore A); otherwise, set $q_0$ to 1
4. Repeat these steps *n* times

# Circuit Arrangement

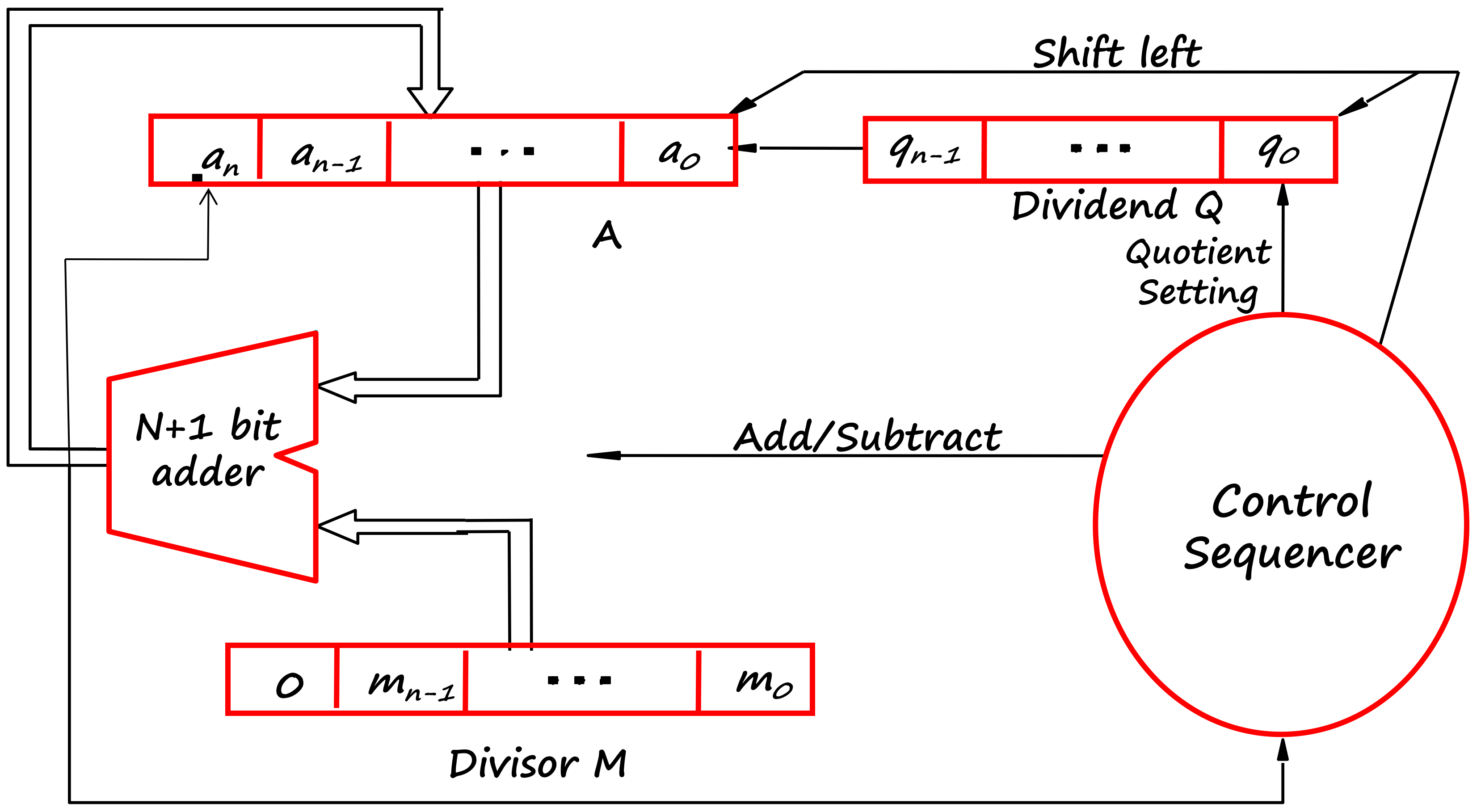
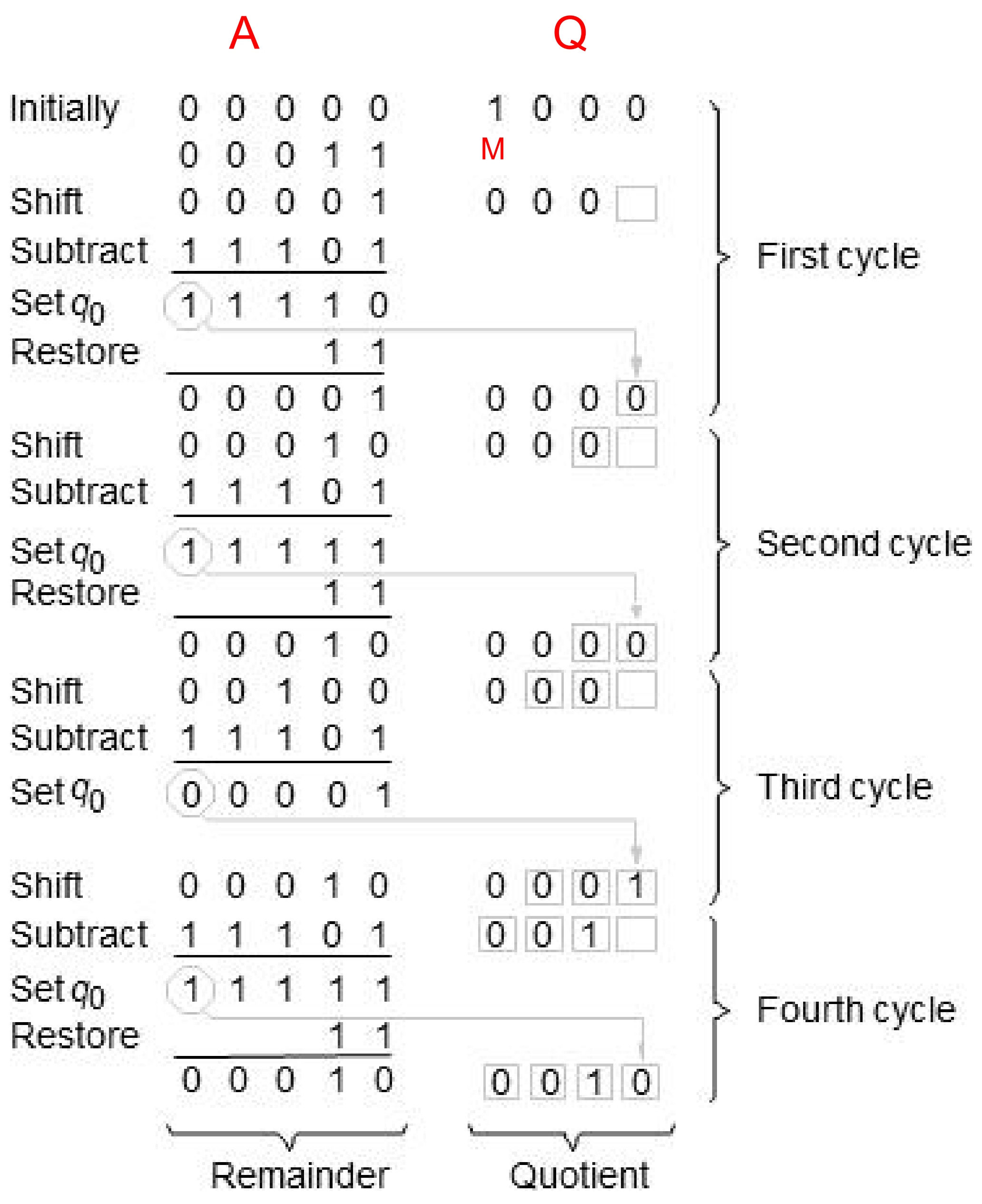
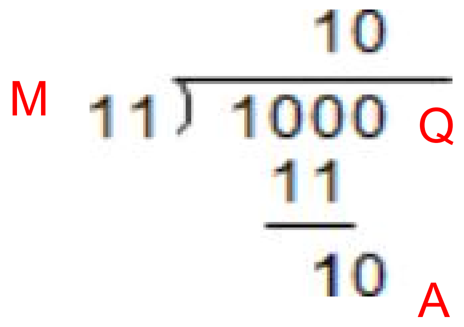


Figure 6.21. Circuit arrangement for binary division.



| | A | Q | |
|---|---|---|---|
| Initially | 0 0 0 0 0 | 1 0 0 0 | First cycle |
| | 0 0 0 1 1 | M | |
| Shift | 0 0 0 0 1 | 0 0 0 □ | |
| Subtract | 1 1 1 0 1 | | |
| Set $q_0$ | 1 1 1 1 0 | | |
| Restore | 1 1 | | |
| | 0 0 0 0 1 | 0 0 0 0 | |
| Shift | 0 0 0 1 0 | 0 0 0 □ | Second cycle |
| Subtract | 1 1 1 0 1 | | |
| Set $q_0$ | 1 1 1 1 1 | | |
| Restore | 1 1 | | |
| | 0 0 0 1 0 | 0 0 0 0 | |
| Shift | 0 0 1 0 0 | 0 0 0 □ | Third cycle |
| Subtract | 1 1 1 0 1 | | |
| Set $q_0$ | 0 0 0 0 1 | | |
| Shift | 0 0 0 1 0 | 0 0 0 1 | |
| Subtract | 1 1 1 0 1 | 0 0 1 □ | Fourth cycle |
| Set $q_0$ | 1 1 1 1 1 | | |
| Restore | 1 1 | | |
| | 0 0 0 1 0 | 0 0 1 0 | |
| | Remainder | Quotient | |



## Nonrestoring Division

- Step 1: (Repeat *n* times)
  - If the sign of A is 0, shift 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

## A nonrestoring-division example.

$$\begin{array}{r} 10 \\ 11\overline{)1000} \\ \underline{11} \\ 10 \end{array}$$

Step 1:

| | | | |
|---|---|---|---|
| Initially | 0 0 0 0 0 | 1 0 0 0 | First cycle |
| | 0 0 0 1 1 | | |
| Shift | 0 0 0 0 1 | 0 0 0 $\boxed{\phantom{0}}$ | |
| Subtract | 1 1 1 0 1 | | |
| Set $q_0$ | 1 1 1 1 0 | 0 0 0 $\boxed{0}$ | |
| Shift | 1 1 1 0 0 | 0 0 $\boxed{0}$ $\boxed{\phantom{0}}$ | Second cycle |
| Add | 0 0 0 1 1 | | |
| Set $q_0$ | 1 1 1 1 1 | 0 0 $\boxed{0}$ $\boxed{0}$ | |
| Shift | 1 1 1 1 0 | 0 $\boxed{0}$ $\boxed{0}$ $\boxed{\phantom{0}}$ | Third cycle |
| Add | 0 0 0 1 1 | | |
| Set $q_0$ | 0 0 0 0 1 | 0 $\boxed{0}$ $\boxed{0}$ $\boxed{1}$ | |
| Shift | 0 0 0 1 0 | $\boxed{0}$ $\boxed{0}$ $\boxed{1}$ $\boxed{\phantom{0}}$ | Fourth cycle |
| Subtract | 1 1 1 0 1 | | |
| Set $q_0$ | 1 1 1 1 1 | $\boxed{0}$ $\boxed{0}$ $\boxed{1}$ $\boxed{0}$ | |
| | | Quotient | |

Step 2:

| | | |
|---|---|---|
| | 1 1 1 1 1 | **Restore remainder** |
| Add | 0 0 0 1 1 | |
| | 0 0 0 1 0 | |
| | Remainder | |

## Non restoring division example 11/5

| | | A | Q | |
|---|---|---|---|---|
| | | 00000 | 1011 | |
| | M | 00101 | | 2's complement of 00101<br>11010 1's complement<br>1+<br>-------------<br>11011 |
| Step 1 | Shift<br>Sub M<br>Set q0 | 00001<br>11011 +<br>------------<br>**1**1100 | 011-<br><br><br>011**0** | 1st cycle |
| | Shift<br>Add<br>Set q0 | 11000<br>00101<br>-----------<br>**1**1101 | 110-<br><br><br>110**0** | 2nd cycle |
| | Shift<br>Add<br>Set q0 | 11011<br>00101<br>------------<br>**0**0000 | 100-<br><br><br>100**1** | 3rd cycle |
| | Shift<br>Sub M<br>Set q0 | 00001<br>11011<br>-----------<br>11100 | 001-<br><br><br>001**0 Quotient** | 4th cycle |
| Step 2 | Restore | 11100<br>00101<br>**---------------**<br>**00001 Remainder** | | |