**UNIT - II ARITHMETIC OPERATIONS**

**2.1 ALU**

An **arithmetic logic unit** (**ALU**) is a digital circuit that performs arithmetic and logical operations.

The ALU is a fundamental building block of the central processing unit (CPU)/

processor of a computer.

 The processors are composed of very powerful and very complex ALUs.

 The ALU was proposed by the famous mathematician, John von Neumann in 1945.

 The ALU works on two types of numbers

1. Fixed point numbers

2. Floating point numbers

 The basic operations are implemented in hardware level. ALU is having collection of two types of operations, namely -

1. Arithmetic operations

2. Logical operations

Consider an ALU having 4 arithmetic operations and 4 logical operations.

 To identify any one of these four logical operations or four arithmetic operations, two control lines are needed.

 Also to identify the any one of these two groups- arithmetic or logical, another control line is needed.

 So, with the help of three control lines, any one of these eight operations can be identified.

 Arithmetic operations include addition, subtraction, multiplication and division.

 The four logical operations include OR, AND, NOT & EX-OR.

 We need three control lines to identify any one of these operations.

 The input combination of these control lines are shown below.

 Control line C

2

is used to identify the group: logical or arithmetic,

o C2=0: arithmetic operation

o C2=0: logical operation.

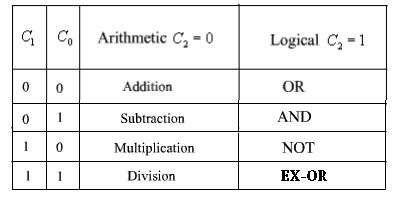
 Control lines C

0

and C are used to identify any one of the four operations in a group.

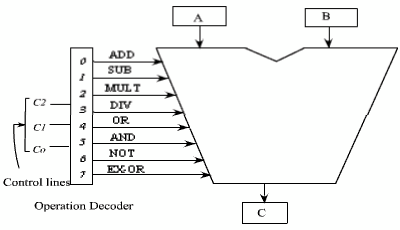
1

One possible combination is given here.



 A 3 x 8 decoder is used to decode the instruction.

 The block diagram of the ALU is shown.



**Figure 2.1:** Block Diagram of the ALU

 The ALU has got two input registers named as A and B and one output storage register, named as C.

 It performs the operation as: C = A operator B

 The input data are stored in A and B, and according to the operation specified in the control lines, the ALU perform the operation and put the result in register C.

**Example**

 If the contents of controls lines are, 000, then the decoder enables the addition operation and it activates the adder circuit and the addition operation is performed on the data that are available in storage register A and B.

 After the completion of the operation, the result is stored in register C.

**2.1.1 Logic Gates**

 There are several logic gates exists in digital logic circuit.

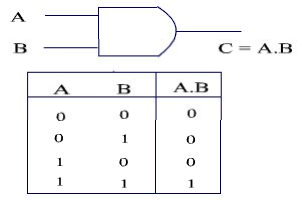
 These logic gates can be used to implement the logical operation.

 Some of the common logic gates are AND, OR, EX-OR etc.

**AND gate**

 The AND gate produces output is high (1) if both the inputs are high (1).

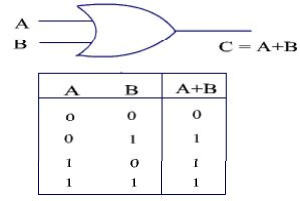
 The AND gate and its truth table are as shown.



**AND gate and its truth table. OR gate**

 The output of the OR gate is high if any one of the input is high.

 The OR gate and its truth table are as shown.

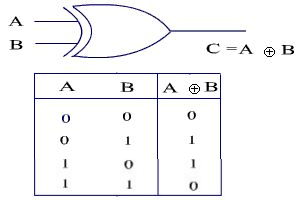


OR gate and its truth table

**EX-OR gate**

 The output of the EX-OR gate is high if either of the input is high.

 The EX-OR gate and its truth table is given as follows.



**EX-OR gate and its truth table**

**2.1.2 Number System**

 Every computer stores numbers, letters, and other special characters in coded form.

 The number system is classified based on the number of characters/symbols/

numbers that it supports, which is called as the base or radix of a number system.

 The most commonly used number systems are

o Decimal number system

o Binary Number System

**Decimal Number System**

 The base of the decimal number system is 10.

 The digits start from 0 to 9.

 The successive positions, read from left to right represent units, tens, hundreds, thousands, and so on.

 Example: the decimal number 7516 (written as 7516

) consists of the digit 6 in

10

the units position, 1 in the tens position, 5 in the hundreds position, and 7 in the thousands position, and its value can be written as:(7x1000) + (5x100) + (1x10)

+ (6xl)(or)2000 + 500 + 80 + 6(or)2586.

**Binary Number System**

 The base of the binary number system is 2.

 “**Binary digit**” is often referred to by the common abbreviation as ‘**bit**’.

 A “bit” in computer terminology means either a 0 or a 1.

 Each position in a binary number represents a power of the base (2).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1 Bit** | **2 Bit** |  | **3 Bit** |  |  | **4 Bit** |  |  |  | **Decimal Value** |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
|  | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 2 |
|  | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 3 |
|  |  |  | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 4 |
|  |  |  | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 5 |
|  |  |  | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 6 |
|  |  |  | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 7 |
|  | | | | | | 1 | 0 | 0 | 0 | 8 |
| 1 | 0 | 0 | 1 | 9 |
| 1 | 0 | 1 | 0 | 10 |
| 1 | 0 | 1 | 1 | 11 |
| 1 | 1 | 0 | 0 | 12 |
| 1 | 1 | 0 | 1 | 13 |
| 1 | 1 | 1 | 0 | 14 |
| 1 | 1 | 1 | 1 | 15 |

**2.1.3 Fixed Point (Integer) Representation**

 A **fixed-point number** representation is a real data type for a number.

 It has a set of digits after and before the radix point (‘.’).

 Fixed-point numbers are useful for representing fractional values (decimal and binary numbers).

 It is used in low-cost embedded microprocessors since

o the processor has no floating point unit (FPU).

o it provides improved performance or accuracy.

**2.1.4 Signed Numbers Representation**

 This is used to represent zero, positive and negative numbers.

 Three representation schemes had been proposed for signed integers:

1. Sign-Magnitude representation

2. 1’s Complement representation

3. 2’s Complement representation

**1. Sign-Magnitude representation**

 An *n-bit signed binary number* consists of two parts

o Sign bit

 It is the left most bit.

 It is also called as the Most Significant Bit (MSB)

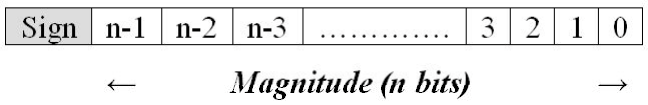
 Sign Bit Value

 0 - Positive Integer or Zero

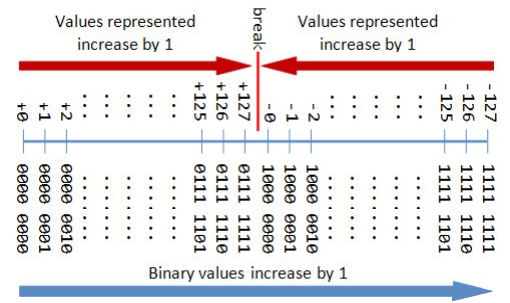
 1 - Negative Integer or Zero

o Magnitude bits

 Remaining n-1 bits - M*agnitude*



 **Sign Magnitude representation from -127 to +127 integer**



 **Example**

o Suppose that *n*=8 and the binary representation is 0 100 00012

 Sign bit

 0  positive

 Absolute value

 100 0001

2

= 65

10

Hence, the integer is +65

10

o Suppose that *n*=8 and the binary representation is 1 100 00012

 Sign bit

 1  Negative

 Absolute value

 100 0001

2

= 65

10

Hence, the integer is -65

10

 **Drawbacks**

o Two representations (0000 00002 and 1000 00002) for the number zero, which could lead to inefficiency and confusion

o Positive and negative integers need to be processed separately

**2. 1’s Complement representation**

 An *n-bit signed binary number* consists of two parts

o Sign bit

 The left most bit, also called the Most Significant Bit (MSB)

 Sign Bit Value

 0 - Positive Integer or Zero

 1 - Negative Integer or Zero

o Magnitude

 Remaining n-1 bits - M*agnitude*

 Positive Integer

o Absolute value of the integer is equal to “the magnitude of the (n-1)-bit binary pattern”

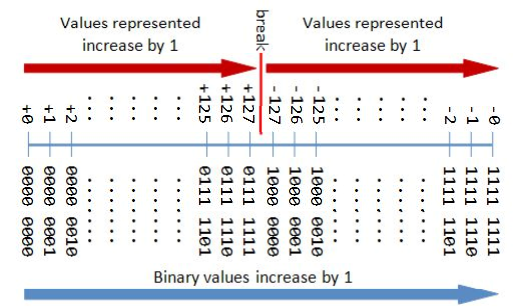
 Negative Integer

o Absolute value of the integer is equal to “the magnitude of the complement

(inverse) of the (n-1)-bit binary pattern”

 Hence called 1’s complement

 **1’s Complement representation from -127 to +127 integer**



 **Example**

o Suppose that *n*=8 and the binary representation is 0 100 00012

 Sign bit

 0  positive

 Absolute value

 100 0001

2

10

= 65

Hence, the integer is +65

10

o Suppose that *n*=8 and the binary representation is 1 100 00012

 Sign bit

 1  Negative

 Absolute value

 100 0001

2

2

= 011 1110

(1’s Complement)

= 62

10

Hence, the integer is -62

10

 **Drawbacks**

o Two representations (0000 00002 and 1111 11112) for the number zero, which could lead to inefficiency and confusion

o Positive and negative integers need to be processed separately

**3. 2’s Complement representation**

 An *n-bit signed binary number* consists of two parts

o Sign bit

 The left most bit, also called the Most Significant Bit (MSB)

 Sign Bit Value

 0 - Positive Integer or Zero

 1 - Negative Integer or Zero

o Magnitude

 Remaining n-1 bits - M*agnitude*

 Positive Integer

o Absolute value of the integer is equal to “the magnitude of the (n-1)-bit binary pattern”

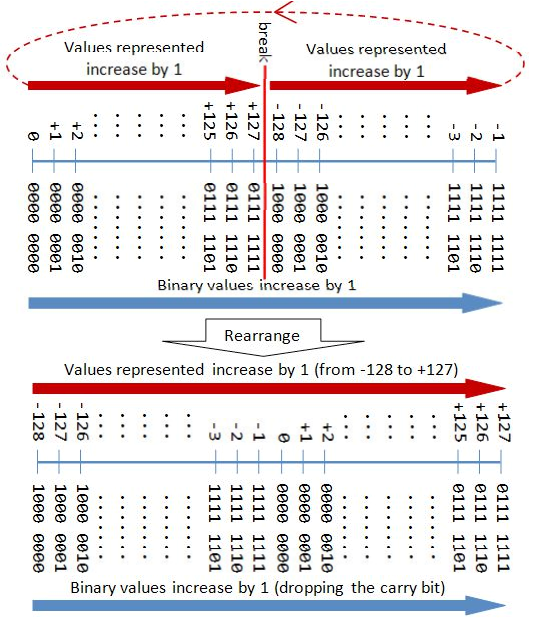
 Negative Integer

o Absolute value of the integer is equal to “the magnitude of the complement (inverse) of the (n-

1)-bit binary pattern plus one”

 Hence called 2’s complement

 **2’s Complement representation from -127 to +127 integer**



 Sign bit

 0 Ò! positive

 Absolute value

 100 0001

2

= 65

10

Hence, the integer is +65

10

o Suppose that *n*=8 and the binary representation is 1 100 00012

 Sign bit

 1  Negative

 Absolute value

 100 0001

2

= 011 1110

2

= 011 1111

(1’s Complement) (2’s Complement)

2

= 63

10

Hence, the integer is -63

10

**2.1.5 Unsigned Numbers**

 An *n-bit unsigned binary number* contains only magnitude part

o All n bits – M*agnitude*

o Represent integers from 0 to 2n-1

|  |  |  |
| --- | --- | --- |
| **n** | **Minimum** | **Maximum** |
| 8 | 0 | 28-1 =255 |
| 16 | 0 | 216-1 =65,535 |
| 32 | 0 | 232-1 =4,294,967,295 (9+ digits) |

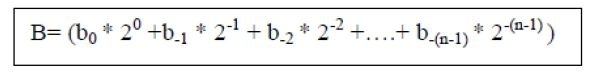
….

 Represent zero and positive integers, but not negative integers

**Floating-point Number**

 The fractional binary numbers are represented by considering the binary point.

 If binary point is assumed to the right of the sign bit ,we represent the fractional binary numbers as,



**2.2 ADDITION AND SUBTRACTION OF SIGNED NUMBERS**

**2.2.1 Addition of numbers**

 The ALU performs addition by adding each bit of the addend with every bit of the augend from right to left.

 At the end of every bit by bit addition, the sum bit is noted and the carry bit is passed to the immediate digit to the left.

**Terminologies**

**Carry**

This represents the overflow result while performing addition of two or more binary numbers.

**Example**

Carry  0 1 1 - Augend 1 0 1 1

Addend  0 0 1 1

Operator  +

——————— Resultant  1 1 1 0

**Least Significant Bit (LSD)**

The leftmost bit of every binary number is the LSD of the binary number.

**Most significant Bit (MSD)**

The rightmost bit of every binary number is the MSD of the binary number.

**Example**

|  |  |  |
| --- | --- | --- |
| 1 1 | 0 | 0 |
|  |  |  |
| **MSD** |  | **LSD** |

 Binary addition always starts from LSD and move towards MSD with or without having carry bit.

 The table below shows the binary addition rules which are followed while performing addition of two binary numbers.

Let the general expression be: **operand1 + operand2**.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operand1** | **Operand2** | **Result** | **Carry** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

For the expression that involves three operands, the general expression shall be:

**operand1 + operand2 + operand3**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operand1** | **Operand2** | **Operand3** | **Result** | **Carry** |
| 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 |

Steps involved in performing addition of two binary numbers are,

Step 1 : Start form the LSD of both the numbers by applying the rules of binary addition.

Step 2 : The result of adding the individual bits of each column is noted in the result row, with the carry bit carried over to the preceding column.

Step 3 : The carry bit is considered while adding the preceding column bits. Step4 : Repeat the above steps unlit MSD is processed.

**Example**

**1. Add (101)**

**2**

**with (1110) .**

Carry  **1 1 0 0 -**

Augend  1 0 1

Addend  

1 1 1 0

**2**

Operator  +

————————— Sum **1 0 0 1 1**

**Result = (10011)**

**2**

**2. Add (100)**

**2**

**2**

**with (11) .**

Carry  **0 0 -**

Augend  1 0 0

Addend  0 1 1

Operator  +

—————————— Sum **1 1 1**

**Result = (111)**

**2**

**3. Add (1110)**

**2**

**with (1101) .**

Carry  **1 1 0 0 -**

Augend  1 1 1 0

Addend  

1 1 0 1

**2**

Operator  +

—————————— Sum **1 1 0 1 1**

**Result = (11011)**

**2**

**Other Examples:**

1. (00011010)

2

2

+ (00001100)

1 1 carry

0 0 0 1 1 0 1 0 = 26

(base 10)

+ 0 0 0 0 1 1 0 0 = 12

(base 10)

0 0 1 0 0 1 1 0 = 38

(base 10)

2. (00010011)

2

2

+ (00111110)

1 1 1 1 1 carry

0 0 0 1 0 0 1 1 = 19

(base 10)

+ 0 0 1 1 1 1 1 0 = 62

(base 10)

0 1 0 1 0 0 0 1 = 81

(base 10)

**2.2.2 Adder Circuits**

**2.2.2.1 Binary Adder circuit**

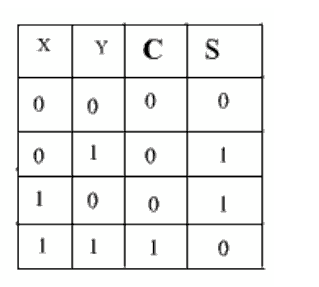
 Binary adder is used to add two binary numbers.

 The adder circuit needs two binary inputs and two binary outputs.

 The input variables elect the augends and addend bits.

 The output variables produce the sum and carry.

**Truth table**

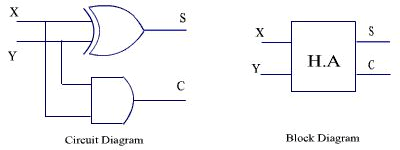


C: Carry Bit S: Sum Bit

The simplified sum of products expressions are

S  x ' y  xy ' C  xy

The circuit is implemented as follows.



**Circuit diagram and Block diagram of Half Adder**

This circuit cannot handle the carry input, so it is termed as half adder.

**2.2.2.2 Full Adder circuit**

 A full adder is a combinational circuit that forms the arithmetic sum of three bits.

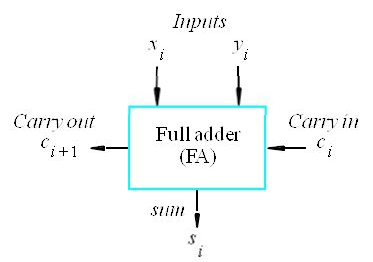
 It consists of three inputs and two outputs.

 Two of the input variables, denoted by x and y, represent the two bits to be added.

 The third input Z, represents the carry from the previous lower position.

**Block Diagram**

The complete circuit for single stage of addition is given



At the ith stage:

 Input:

o xi, the first input

o yi, the second input

o ci, the carry-in

 Output:

o si is the sum

o ci+1 carry-out to (i+1)

st

state

**Truth Table**

**i i i**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **y** | **Carry-inC** | **SumS** | **Carry-outC** |
| 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 |

The simplified expression for Sum, S and C are

S  x ' y ' z  x ' yz ' xy`z  xyz

C  xy  xz  yz

 xy  xy ' z  x ' yz

The above expressions can be written as follows.

S  z  (x  y)

 z '(xy ' x ' y)  z(xy ' x ' y ')

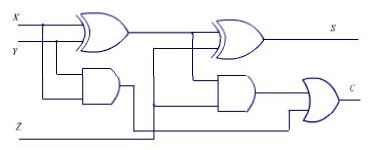
 z '(xy ' x ' y)  z(xy  x ' y ')

 xy ' z ' x ' yz ' xyz  x ' y ' z

C  z(xy ' x ' y)  xy  xy ' z  x ' yz  xy

**Circuit diagram - full adder**

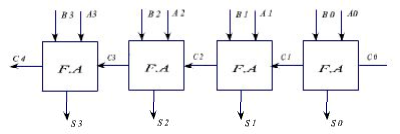
**Full Adder**



**2.2.2.3 4 – bit Adder circuit**

 To get the four bit adder, we have to use 4 full adder blocks.

 The carry output the lower bit is used as a carry input to the next higher bit.



**4-bit adder circuit**

**2.2.2.4 n-bit ripple-carry adder**

 Cascade n full adder (FA) block to form a n-bit adder.

 Each full adder inputs a C

in

, which is the C

out

of the previous adder

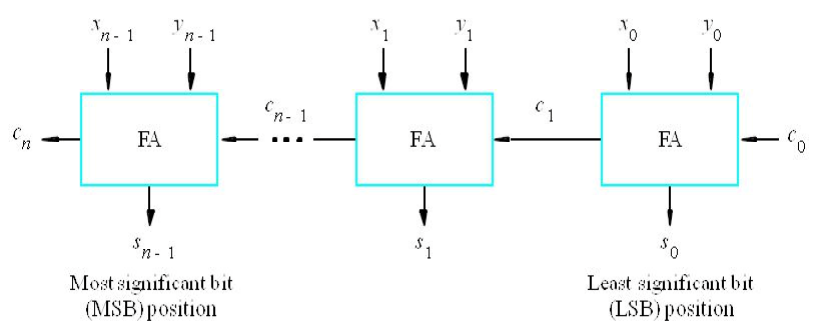
o This adder is called as a n-bit ripple carry adder

 Each carry bit “ripples” or “propagates” to the next full adder

**Block Diagram**

o Used to add two input X and Y

o xn-1 and yn-1 – Sign bits



 Carry-in c

0

**Advantage**

into the LSB position provides a convenient way to perform subtraction

 Allows for fast design time

**Disadvantage**

 Relatively slow

o Each full adder must wait for the carry bit to be calculated from the previous full adder

**2.2.2.5 Cascade of k n-bit Adders**

 Circuit to add K n-bit numbers by cascading k n-bit adders

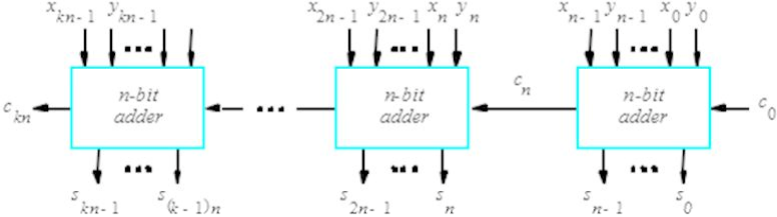
 Each n-bit adder forms a block, so this is cascading of blocks

 Carries ripple or propagate through blocks, Blocked Ripple Carry Adder

**Block Diagram**

 The carry-in, c0, into the least-significant-bit(LSB) position provides a convenient means of adding 1 to a number

 Forming the 2’s-complement of a number involves adding 1 to the 1’s complement of the number



**2.2.3 Subtraction of numbers**

 The ALU performs subtraction by subtracting each bit of the minuend with every bit of the subtrahend from right to left.

 At the end of every bit by bit subtraction, the difference bit is noted and the borrow bit is passed to the immediate digit to the left.

**Terminologies**

**Borrow**

This represents the underflow value while performing subtraction of two or more binary numbers.

Borrowing occurs whenever a smaller bit (0) is subtracted by a larger bit (1).

**Example**

Borrow **- 1 -**

0 10

 

Minuend ~~1~~ 0 1

Subtrahend 0 1 1

Operator  -

————————— Difference 0 1 0

 Binary subtraction starts from LSD and move towards MSD with or without borrowing a bit from the preceding bit.

 The table below shows the binary addition rules which are followed while performing subtraction of two binary numbers.

Let the general expression be: **operand1 - operand2**.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operand1** | **Operand2** | **Result** | **Borrow** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

**Steps involved in performing subtraction of two binary numbers**

Step 1 : Start form the LSD of both the numbers by applying the rules of binary subtraction. Step 2 : The difference of each column is noted, with the borrow bit taken from the preceding

column, if required.

Step 3 : The borrow bit is considered while subtracting the preceding column bits. Step4 : Repeat the above steps unlit MSD is processed.

Step 5 : If the borrow bit of the MSD of the minuend is ’0’, the resultant difference is a positive value. Stop the operation.

Step 6 : If the borrow bit of the MSD of the minuend is ’1’, the resultant difference is a negative value. Hence perform two’s complementation of the difference value and stop the operation.

**Finding Two’s complement**

1. Take one’s complement by inverting 0’s by 1’s and 1’s by 0’s.

2. Add ‘1’ to the one’s complement result.

**Example: 1. Subtract (110)**

**2**

**by (101) .**

Borrow **0 0 1**

0**‘!** 10**‘!**

**2**

Minuend 1 1 0

Subtrahend 1 0 1

Operator  -

—————————- Difference  **0 0 1**

**Result = (001)**

**2**

**Example: 2. Subtract (100) by (11) .**

**2 2**

Borrow  **0 1 1**

1**‘!**10**‘!** 10**‘!**

Minuend ~~1~~ 0 0

Subtrahend 0 1 1

Operator  -

—————————- Difference  **0 0 1**

**Example: 3. Subtract (101)**

**2**

**by (110) .**

Borrow  **1 1 0**

0**‘!** 10**‘!**

**2**

Minuend ~~1~~ 0 1

Subtrahend 1 1 0

Operator  -

—————————- Difference  **1 1 1**

Taking two’s complement of (111)  000 + 1  001.

**Result = - (001)** [‘-‘ is indicated since the borrow bit of the minuend’s MSD is 1]

**2**

**Example: 4. Subtract (11) by (100) .**

**2 2**

Borrow  **1 0 0**

10**‘!**

Minuend 0 1 1

Subtrahend 1 0 0

Operator  -

—————————- Difference  **1 1 1**

Taking two’s complement of (111)  000 + 1  001.

**Result = - (001)** [‘-‘ is indicated since the borrow bit of the minuend’s MSD is 1]

**2**

**Other Examples**

1.00100101 – 00010001 = 00010100

0 1 borrows

(base 10)

(base 10)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | = | 37 |
| - 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | = | 17 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | = | 20 |

2.00110011 – 00010110 = 00011101 1

(base 10)

0 0 1 1 borrows

(base 10)

(base 10)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 1 | 1 | 0 | 0 | 1 1 | = | 51 |
| - 0 | 0 0 | 1 | 0 | 1 | 1 0 | = | 22 |
| 0 | 0 0 | 1 | 1 | 1 | 0 1 | = | 29 |

(base 10)

**2.2.4 Binary Subtractor circuit**

 The subtraction operation can be implemented with the help of binary adder circuit, since (A – B) = A + (-B).

 2’s complement representation of a number is treated as a negative number of the given number.

 2’s complement of a number shall be calculated by complementing each bit and adding 1 to it.

 The circuit for subtracting A-B consist of an added with inverter placed between each data input B and the corresponding input of the full adder.

 The input carry C

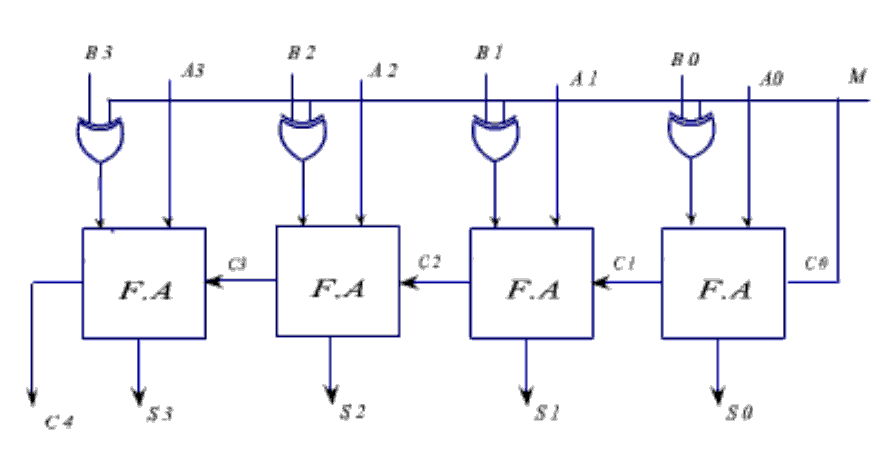
0

must be equal to 1 when performing subtraction.

 The operation thus performed becomes A, plus the 1’s complement of B , plus 1.

 This is equal to A plus 2’s complement of B.

 With this principle, a single circuit can be used for both addition and subtraction.



**4-bit adder subtractor**

 The circuit diagram of a 4-bit adder subtractor is shown above.

 Here, the mode ( M ) is the selection input line, which will determine the operation,

 If M=0, then addition, A+B is performed.

 If M=1, then (A – B) = A + (-B) is performed. The operation of OR gate:

x  0  x

x  1  x '

If M=0,

Bi  0  Bi

Bi  1  Bi '

**2.2.5 Addition / Subtraction Logic Unit**

 Perform subtraction operation, X-Y

o Find 2’s complement of Y

o Add it with X

 Add/Sub control

o Used to decide whether addition or subtraction is performed

o 0 – Addition

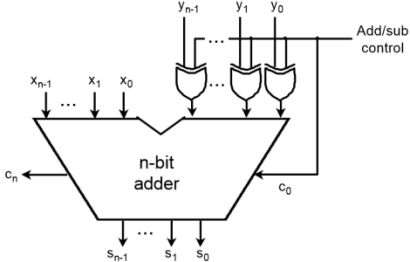
 Supply the Y-vector unchanged to one of the adder input

 Carry-in signal value is 0

o 1 - Subtraction

 Supply the 1’s complement of Y-vector

 Carry-in signal value is 1



**2.2.6 Overflow in Addition and Subtraction**

 An overflow occurs when the result of the operation cannot be represented with the available hardware; here it is 32–bit word. While adding or subtracting two

32-bit numbers may yield a result that needs 33rd bit to be full expressed.

 Overflow cannot occur when adding operands with different signs. The reason is the sum must be no larger than one of the operands.

 For example, -10 + 4 = -6. Since the operands fit in 32 bits and the sum is no

larger than an operand, the sum must fit in 32 bits as well. Therefore, no overflow can occur when adding positive and negative operands.

 Similarly in Subtraction we subtract by negating the second operand and then

add i.e. x - y = x + (-y). Therefore, when we subtract operands of the same sign we are actually adding operands of different signs.

 When can overflow occur then?

o In addition, overflow occurs, when adding two positive numbers ends up in a negative result or vice versa. This means a carry out occurred in the sign bit.

o In Subtraction, overflow occurs when we subtract a negative number from a positive number and get a negative result, or vice versa. This means borrow occurred from the sign bit.

o Unsigned integers are commonly used for memory addresses where overflows are ignored.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Operand A | Operand B | Result indicating overflow |
| A+B | (+) ve | (+) ve | (-) ve |
| A+B | (-) ve | (-) ve | (+) ve |
| A-B | (+) ve | (-) ve | (-) ve |
| A-B | (-) ve | (+) ve | (+) ve |

 The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

o Add (add), add immediate (addi), and subtract (sub) cause exceptions on overflow.

o Add unsigned (addu), add immediate unsigned (addiu), and subtract unsigned

(subu) do not cause exceptions on overflow.

 MIPS detects overflow with an exception, also called an interrupt on many computers.

An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed.

 MIPS includes a register called the exception program counter (EPC) to contain the address of the instruction that caused the exception

**2.3 MULTIPLICATION**

 Multiplication of two numbers is also performed on binary digits.

 The operand to be multiplied is called the multiplicand.

 The second operand that quantifies the multiplicand is called the multiplier.

 The final result is the product value that is obtained after multiplication.

**2.3.1 Manual multiplication algorithm**

 Multiplication process involves generation of partial product, one for each digit in multiplier

o Partial products are added to produce the final product

 In binary system partial products are easily defined

o If multiplier bit is 0, the partial product is 0

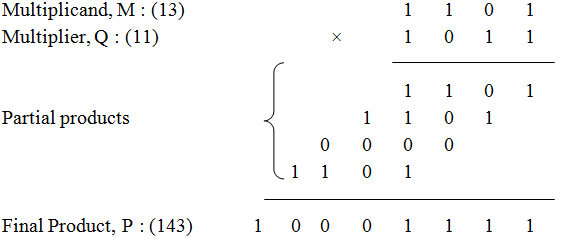
o If multiplier bit is 1,the partial product is multiplicand

 Product of two n-digit numbers can be accommodated in **2n** digits

 So, product of the two 4-bit numbers fits into 8 bits

**Example**

Multiplying 1101 by 1011



 First operand is called Multiplicand, second one is Multiplier and final result is called product

 Each successive partial product is shifted one position to the left relative to the next partial product

 Final product is produced by summing the partial products

**2.3.1.1 Multiplication Hardware – First Version**

 The hardware given below is used to multiply two binary data.

 The Multiplicand register, the ALU and the product register handle 64 bits whereas the multiplier register can handle 32 bits.

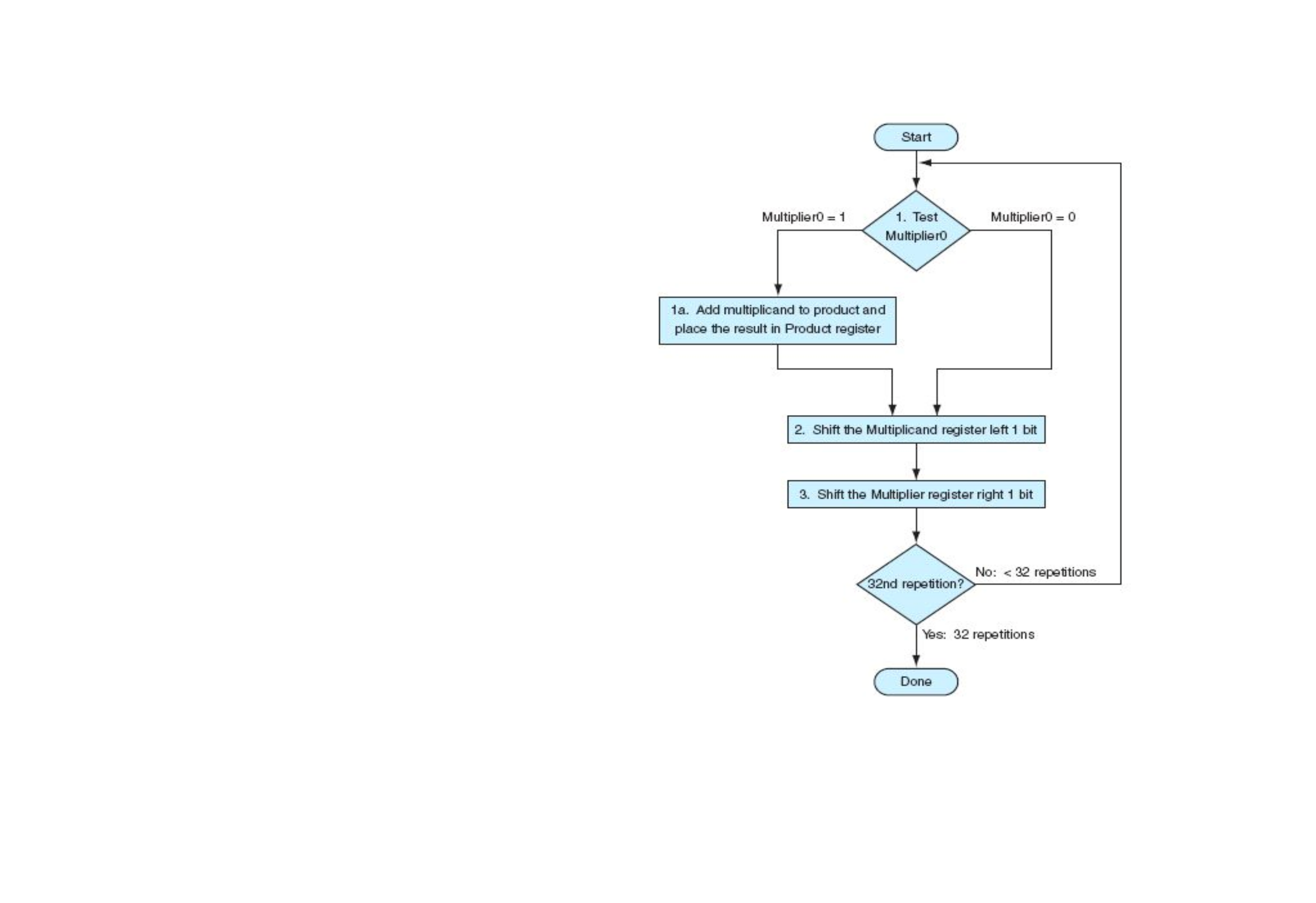
 The multiplicand is processed from right to left, by shifting left one bit on each step.

 The multiplier is shifted in the opposite direction at each step.

 The product register is initialized to 0. The value changes based on the control test decision.

 The control test component selects when to shift the multiplicand and the multiplier and when to write product bits in product register.

**l t**



**Ar thmetic Operat ons** 2 29

**F owchar**

 The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register.

 The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying by hand.

 The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration.

 These three steps are repeated 32 times to obtain the product.

 If each step took a clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers.

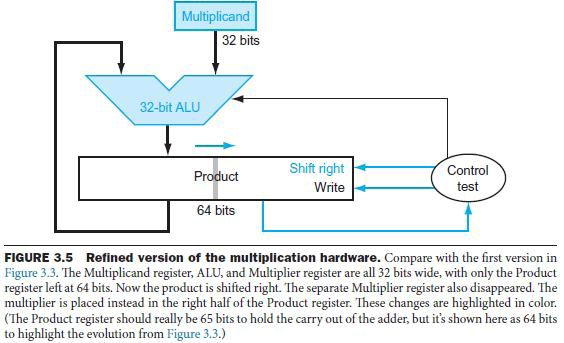
 The relative importance of arithmetic operations like multiply varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply.

**2.3.1.2 Multiplication Hardware - Refined Version**

 The refined version has 32 bit Multiplicand register and ALU, with the product register having 64 bits long.

 The multiplier register is placed instead of the right half of the product register.

 Based on the decision made by the control test, the product register is shifted right one bit at each step.



**2.3.1.3 Array Implementation of positive binary operands**

 Implemented in a combinational two-dimensional logic array

 Full Adder(FA)

o Main component in each cell

 AND gate

o Determines whether a multiplicand bit, mj, is added to the incoming partial- product bit, based on the value of the multiplier bit, q

i.

 If q =0, PPi is passed vertically downward unchanged.

i

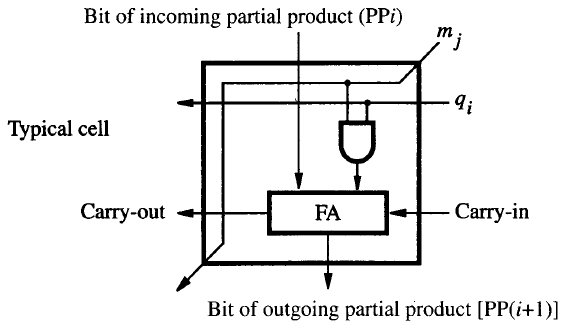
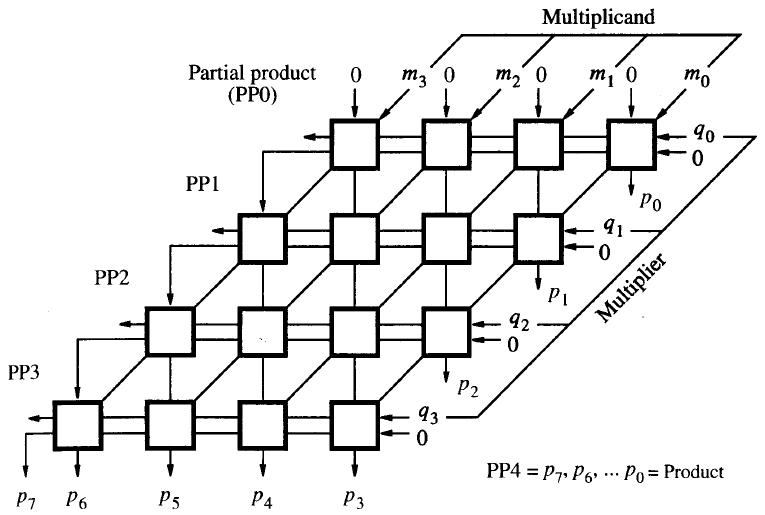
 If q =1, PPi is added with the multiplicand to generate PP(i+1), for each row 0 d”

i

i d” 3.

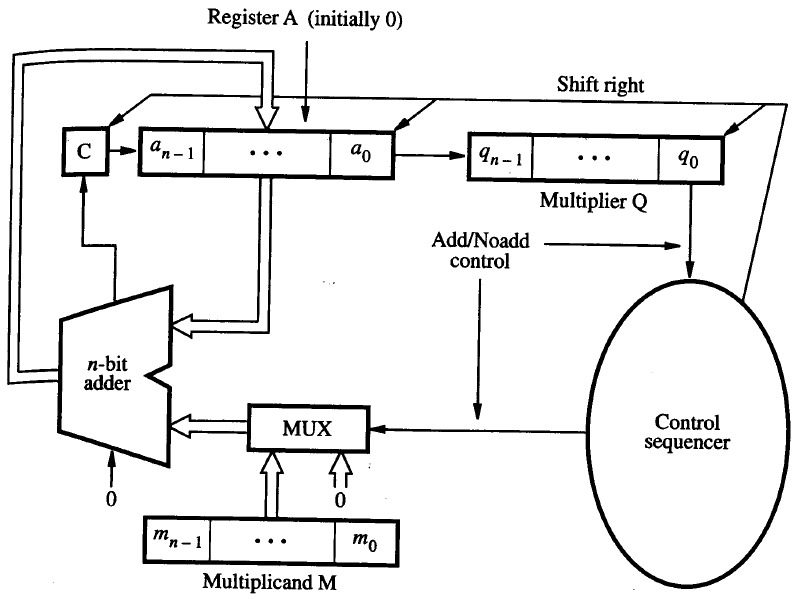
 PP0 is all 0’s.

 PP4 is the desired product.



**2.3.1.4 Sequential Circuit Binary Multiplier**

**Register Configuration**



**Algorithm**

 Multiplier is loaded into the register Q and multiplicand is loaded into the register B

 Register C and A is initially set to 0

 Check the whether q

0

of Q register bit is 0 or 1

 If q

0

bit is 1

o Add multiplicand and partial product

o Shift all bits of C,A and Q registers to the right one bit

 If q

0

bit is 0

 C bit goes to A

n-1, 0

A

goes to Q

n-1 0

, Q

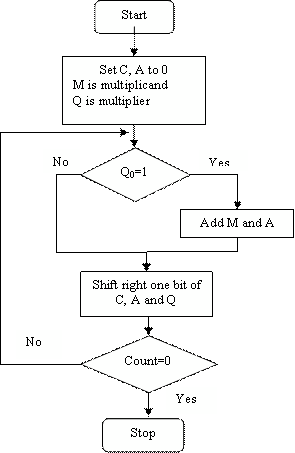
is lost

o No need to perform addition

o Shift all bits of C,A and Q registers to the right one bit

 Repeat the step to get the desired results in A and Q registers

**Flowchart**



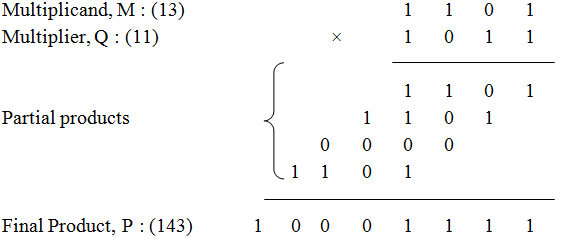
**Example**

 Multiplicand (M) : 1101

 Multiplier (Q) : 1011

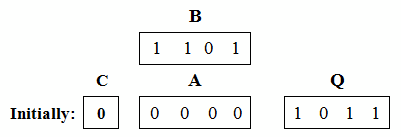
Longhand Multiplication

Multiplying 1101 by 1011

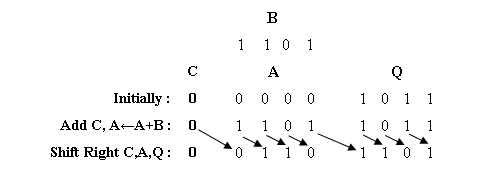


**Multiplication Process**

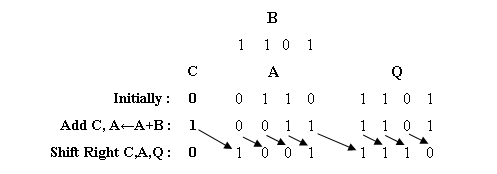
 **Initially**



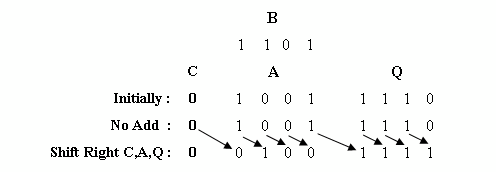
 **First Cycle**



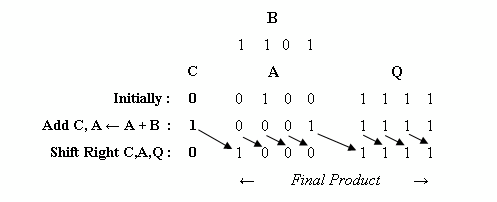
 **Second Cycle**



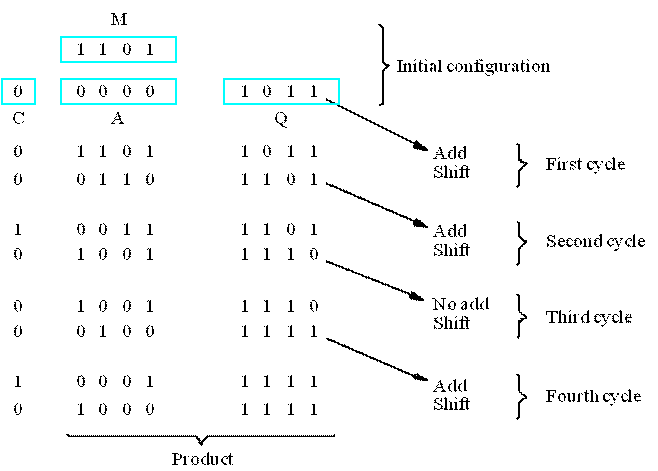
 **Third Cycle**



 **Fourth Cycle**



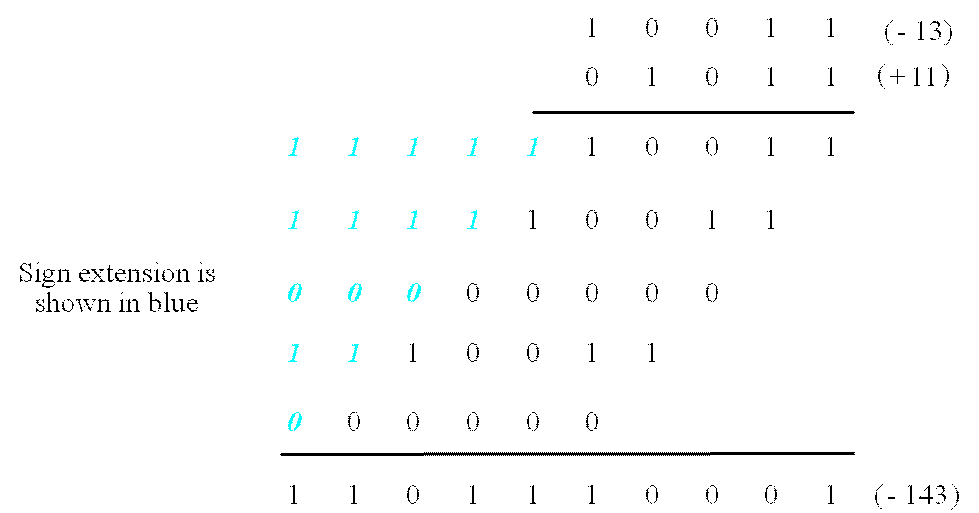
**Illustration**



**2.3.1.5 Signed-Operand Multiplication**

**Example**

Considering 2’s-complement signed operands, perform (-13)  (+11) if the same unsigned multiplication method is followed.



 For a negative multiplier, a straightforward solution is to form the 2’s-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier

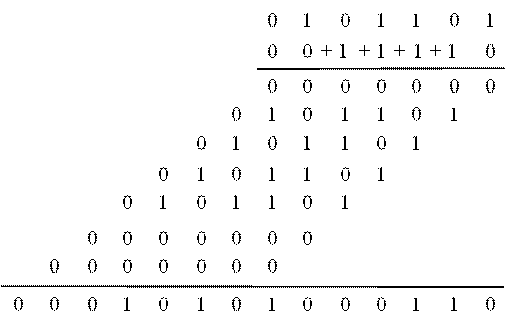
o Possible because complementation of both operands does not change the value or the sign of the product

**2.3.2 Booth’s Algorithm**

 A technique that works equally well for both negative and positive multipliers.

 Generates a 2n-bit product and treats both positive and negative 2’s complement n- bit operands uniformly.

 Consider a multiplication, where the multiplier is positive 0011110 (30)

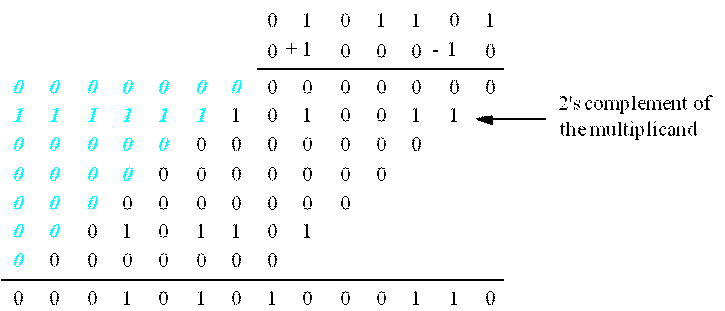


 To reduce the number of required operations

o Represent the multiplier in terms of difference between two numbers(2n- bit product)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 64 32 16 8 4 | 2 1 |  |  |  |  |
| 0 0 1 1 1 | 1 0  |  | 30 |  |
| o Represent 30 as | 32(25) – 2(21) |  |  |  |
|  | **64 32 16 8** | **4** | **2** | **1** |
| 25  32  | 0 1 0 0 | 0 | 0 | 0 |
| 21  2 -  | 0 0 0 0 | 0 | 1 | 0 | - |
| 30 | 0 0 1 1 | 1 | 1 | 0 |  |

o Multiplication



**2.3.2.1 Booth recoding of a multiplier**

 In the Booth scheme,

o -1 times the shifted multiplicand is selected when moving from 0 to 1, and

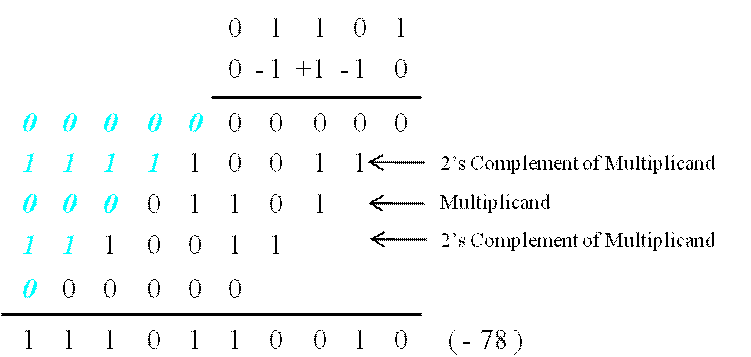
o +1 times the shifted multiplicand is selected when moving from 1 to 0, as the multiplier is scanned from right to left

 **Example**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **0** | **1** | **0 1** | **1** | **0** | **0** | **1** | **1** | **0** |
| 0 | +1 | -1 | +1 0 | -1 | 0 | +1 | 0 | -1 |  |

**2.3.2.2 Booth multiplication with a negative multiplier**

|  |  |  |  |
| --- | --- | --- | --- |
| **Multiplicand: (+13)**  | **0 1 1 0 1**  **0 1** | **1 0** | **1** |
| **Multiplier: (-6)**  | **× 1 1 0 1 0 0 -1** | **+1 -1** | **0** |



**Booth multiplier Recoding Table**

|  |  |  |
| --- | --- | --- |
| **Multiplier** | | **Version of multiplicand** |
| **Bit i** | **Bit i-1** | **selected by bit i** |
| 0 | 0 | 0 × M |
| 0 | 1 | +1 × M |
| 1 | 0 | -1 × M |
| 1 | 1 | 0 × M |

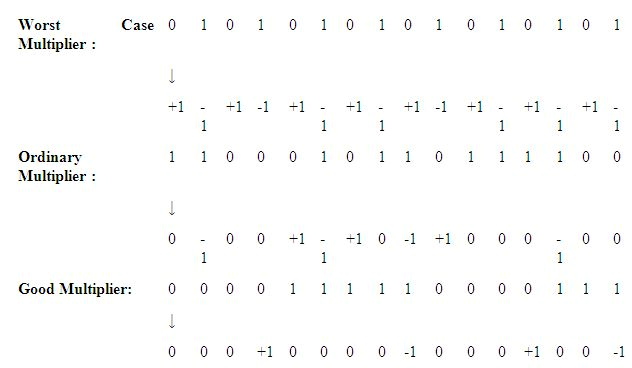
16-bit Multiplier Types

 Best case

o a long string of 1’s (skipping over 1s)

 Worst case

o 0’s and 1’s are alternating



**Advantage**

 Used for both negative and positive integers

 Achieves more efficiency in number of addition

**2.3.3 Fast Multiplication**

The techniques used for speeding up the multiplication operation are

 Bit-Pair Recoding of Multipliers

 Carry-Save Addition of Summands

**2.3.3.1 Bit-Pair Recoding of Multipliers**

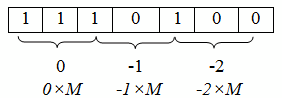
 Halves the maximum number of summands (versions of the multiplicand)

 Derived from the booth algorithm

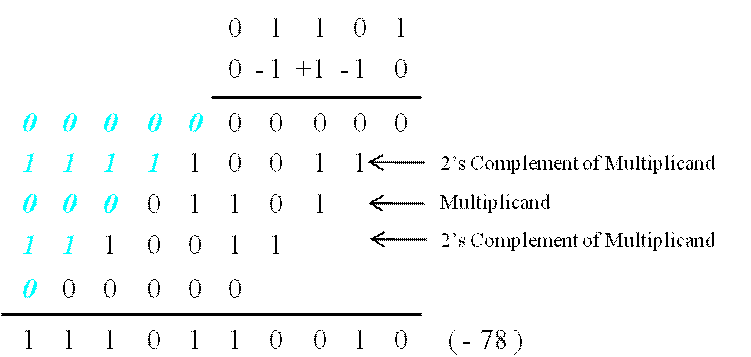
**Multiplicand selection decisions table**

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

**Example**

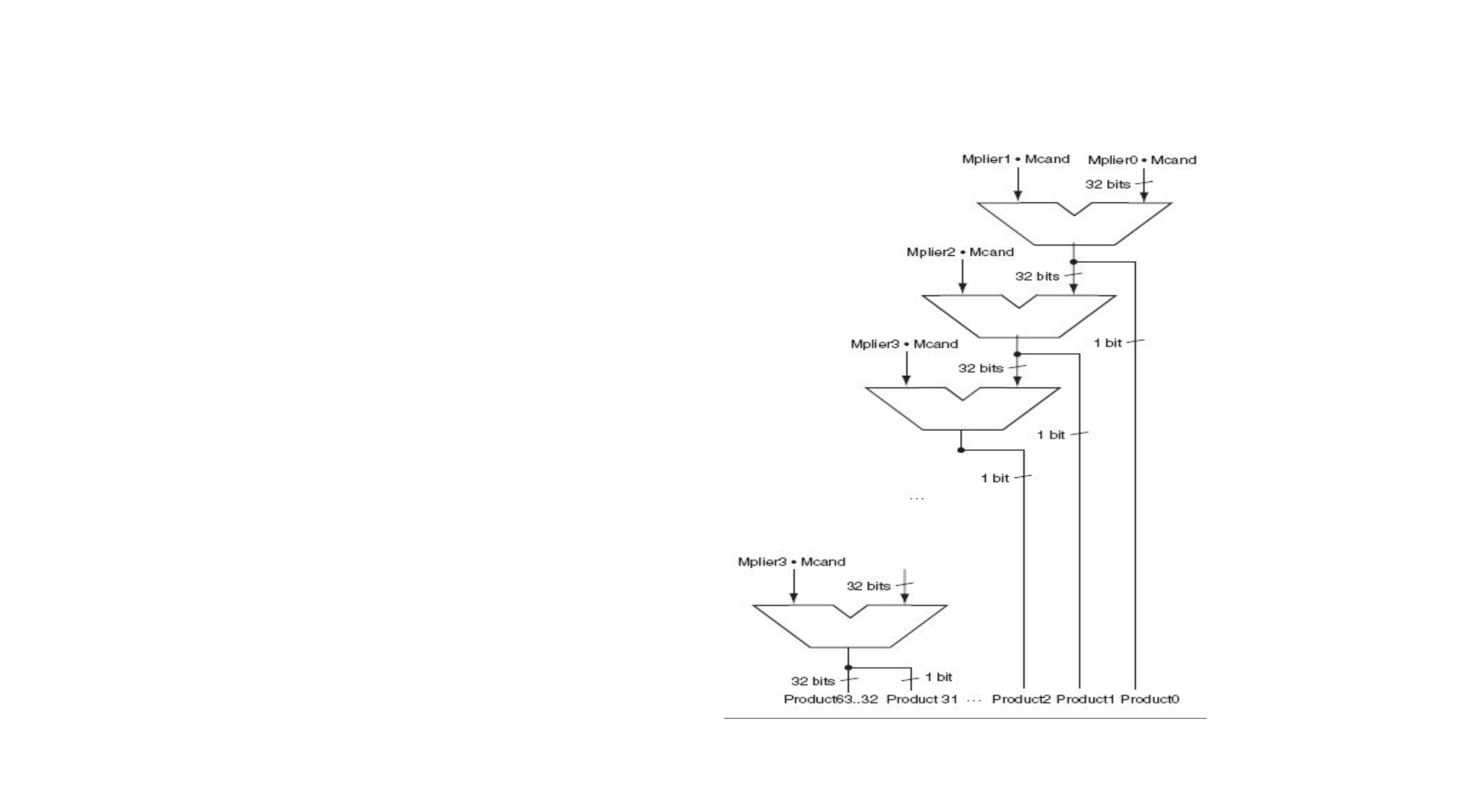
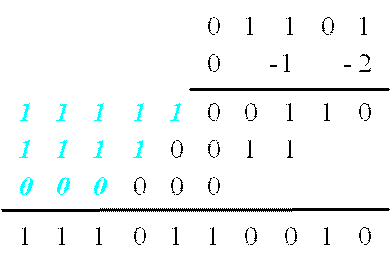


**Using Booth Recoding**



**Fas Mu p cat on hardware**

**t lti li i**



**2.3.3.2 Carry-Save Addition (CSA) of Summands**

 Reduces the time needed to add the summands and Speed up the addition process

**Ripple-carry Array**

 Also called row ripple form

 Each row consists of AND gates that implement the bit products

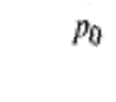
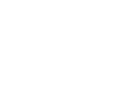
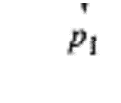
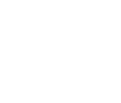
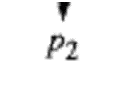
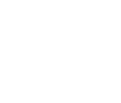
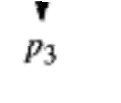
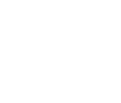
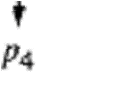
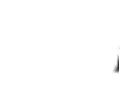
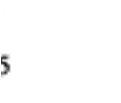
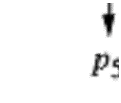
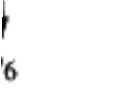
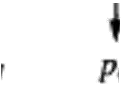
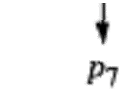
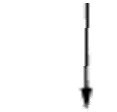
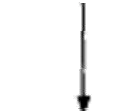
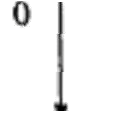
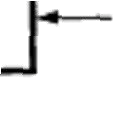
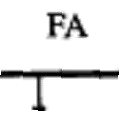
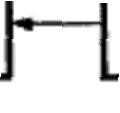
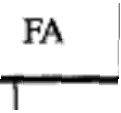
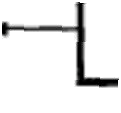
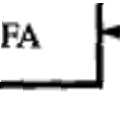
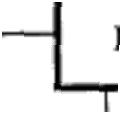
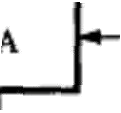
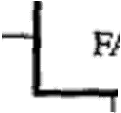
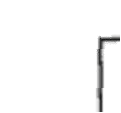
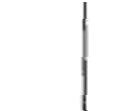
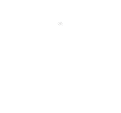
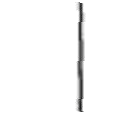
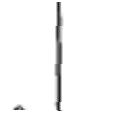
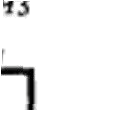
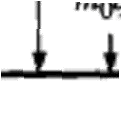
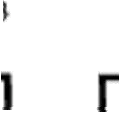
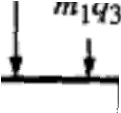
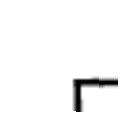
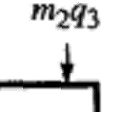
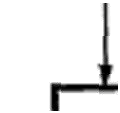
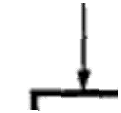
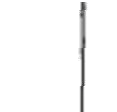
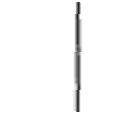
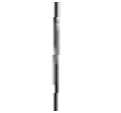
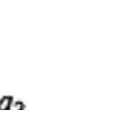
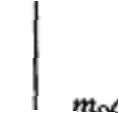
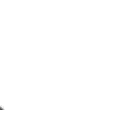
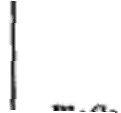
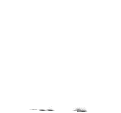
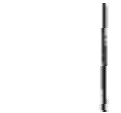
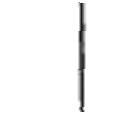
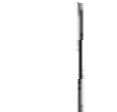
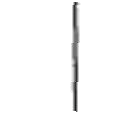
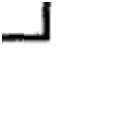
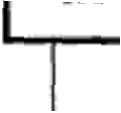
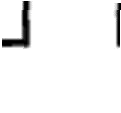
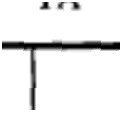
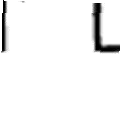
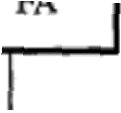
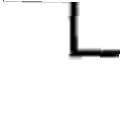
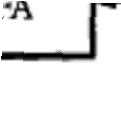
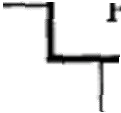
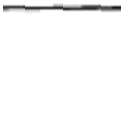
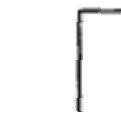
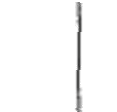
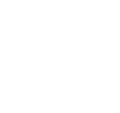
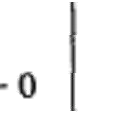
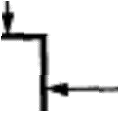
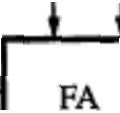
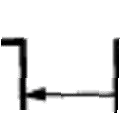
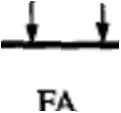
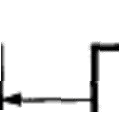
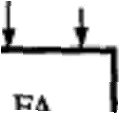
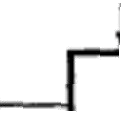
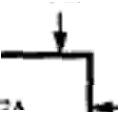
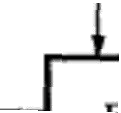
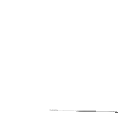
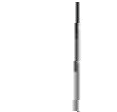
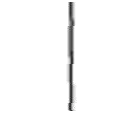
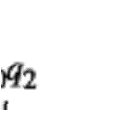
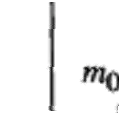
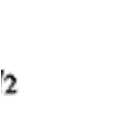
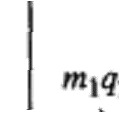
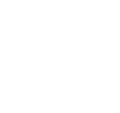
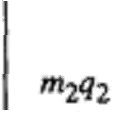
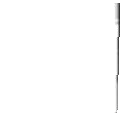
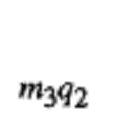
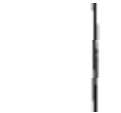
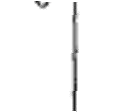
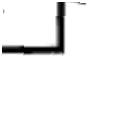
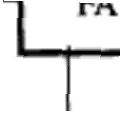
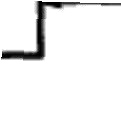
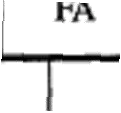
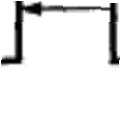
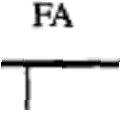
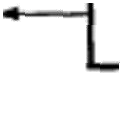
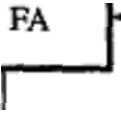
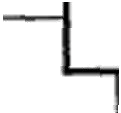
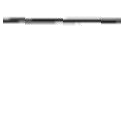
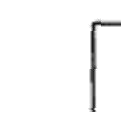
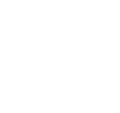
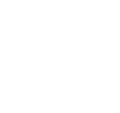
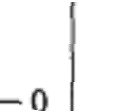
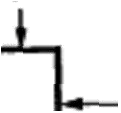
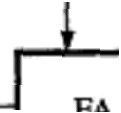
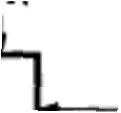
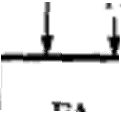
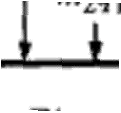
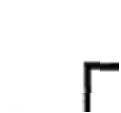
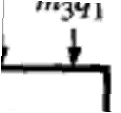
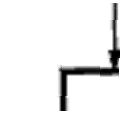
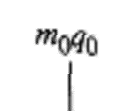
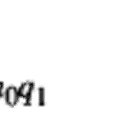
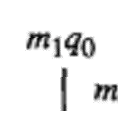
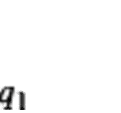
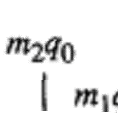
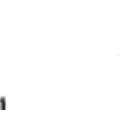
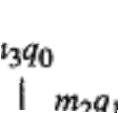
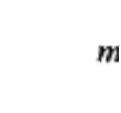
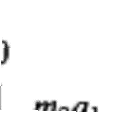
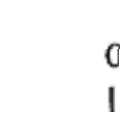
First Row bit products: m q , m q , m q and m q

o

3 0

2 0

1 0



0 0

 Carries are ripple along the rows

**Carry-save Array**

 Carries are saved and introduced into the next row

o Frees up an input to three full adders in the first row

o Inputs are introduced the third summand bit products m2q2, m1q2 and m0q2

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

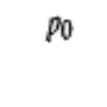
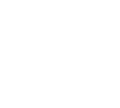
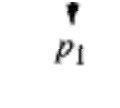
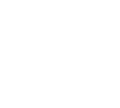
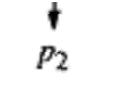
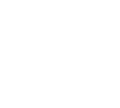
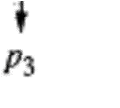
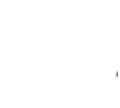
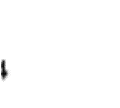
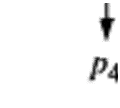
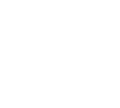
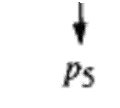
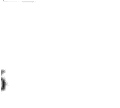
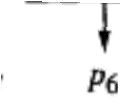
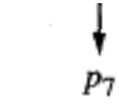
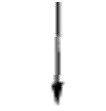
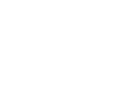
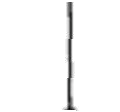
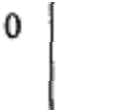
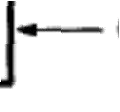
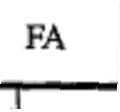
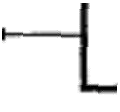
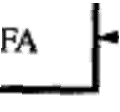
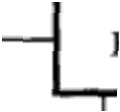
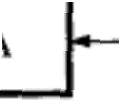
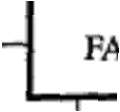
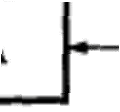
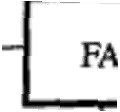
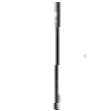
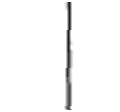
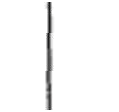
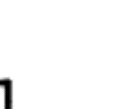
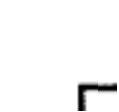
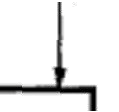
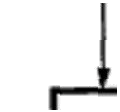
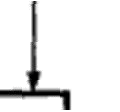
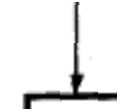
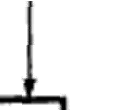
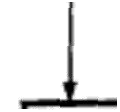
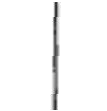
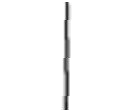
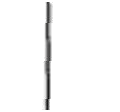
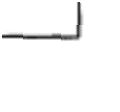
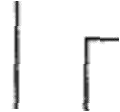
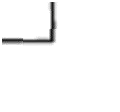
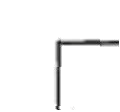
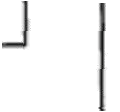
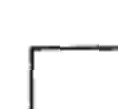
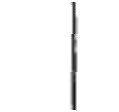
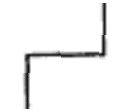
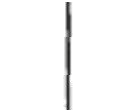
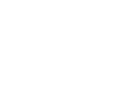
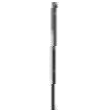
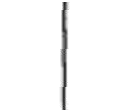
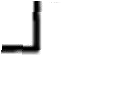
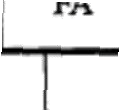
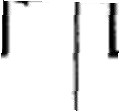
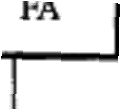
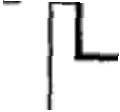
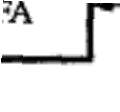
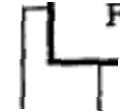
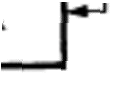
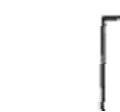
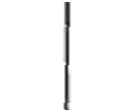
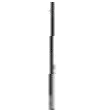
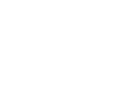
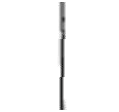
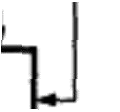
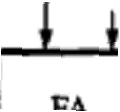
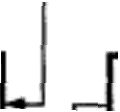
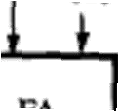
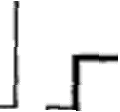
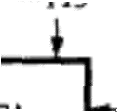
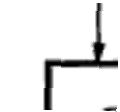
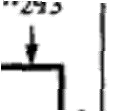
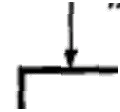
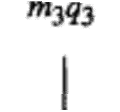
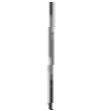
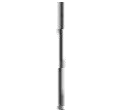
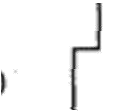
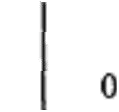
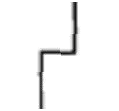
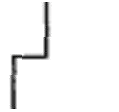
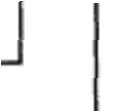
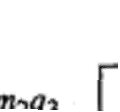
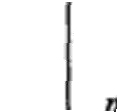
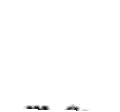
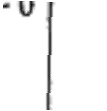
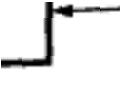
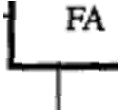
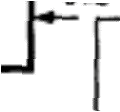
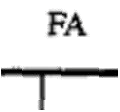
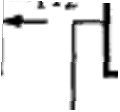
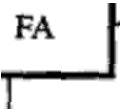
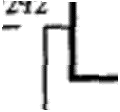
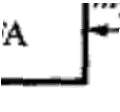
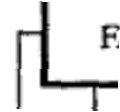
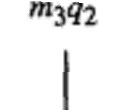
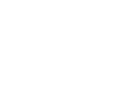
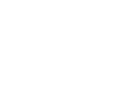
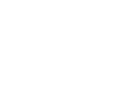
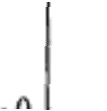
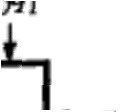
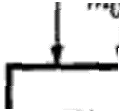
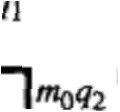
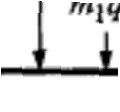
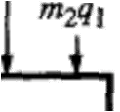
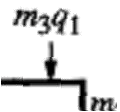
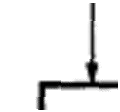
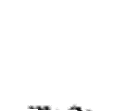
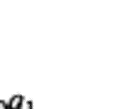
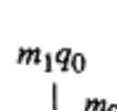
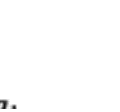
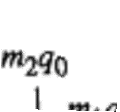
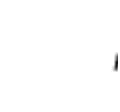
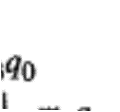
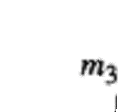
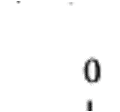
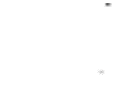
o Third input is used to introduce the bit products m2q3, m1q3 and m0q3 of the fourth summand

o High-order bit products m3q2 and m3q3 of the third and fourth summands are introduced into the remaining free inputs at the left end in the second and third rows

o Saved carry bits and sum bits from the second row are added in the third row to produce final product bits

 The delay through the carry-save array is somewhat less than delay through the ripple- carry array. This is because the S and C vector outputs from each row are produced in parallel in one full-adder delay.

 Consider the addition of many summands,



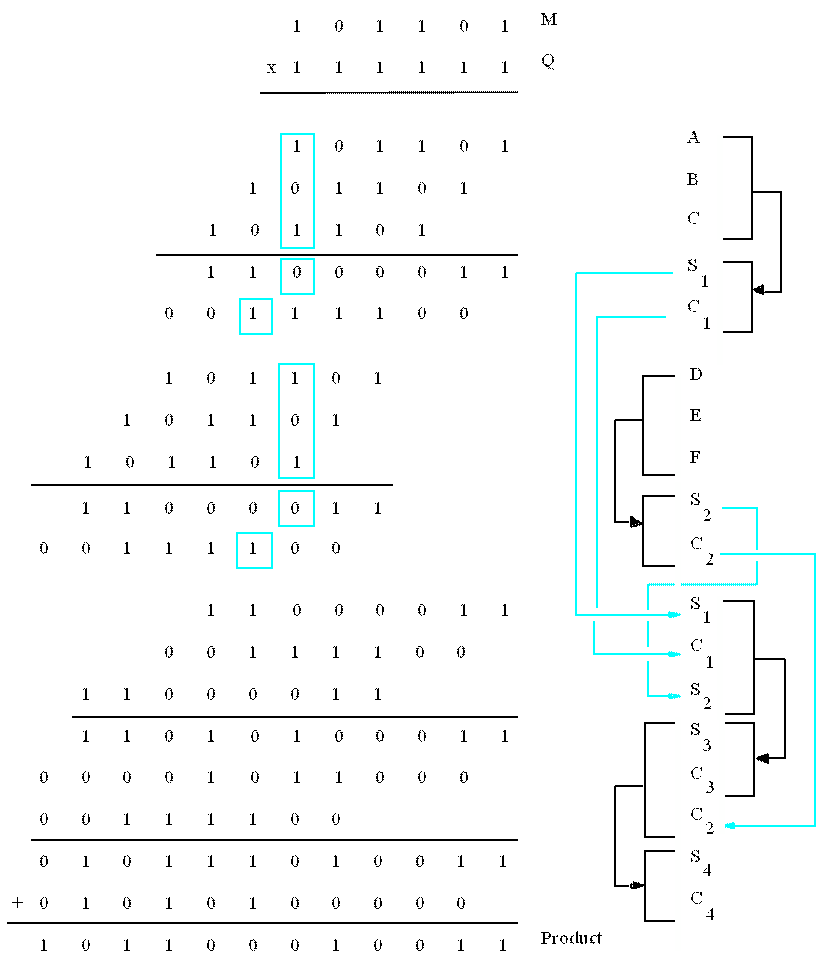
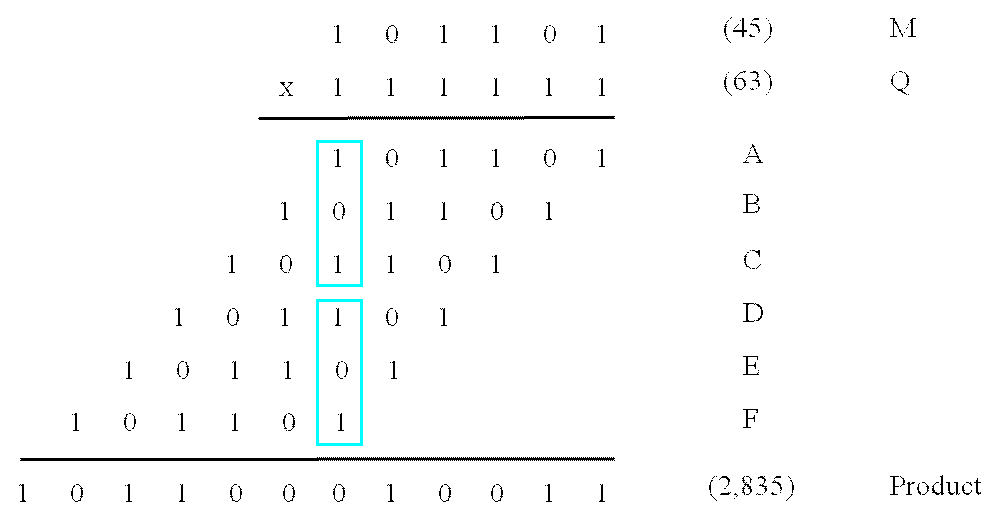
o Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay

o 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

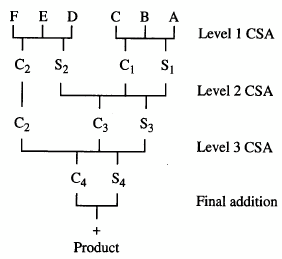
o Continue with this process until there are only two vectors remaining

o It can be added in a RCA or CLA to produce the desired product

**Example**



**Schematic representation of the carry-save addition operations**



 When the number of summands is large, the time saved is proportionally much greater.

 Some omitted issues

o Sign-extension

o Computation width of the final CLA/RCA

o Bit-pair recoding

**2.4 DIVISION**

**2.4.1 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

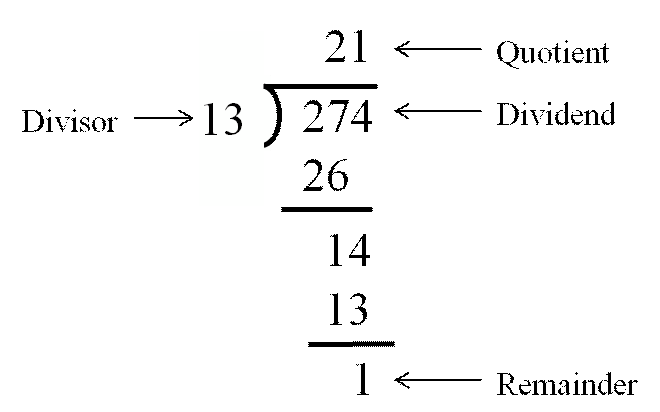
 a quotient bit of 0 is determined

 the dividend is restored by adding back the divisor, and

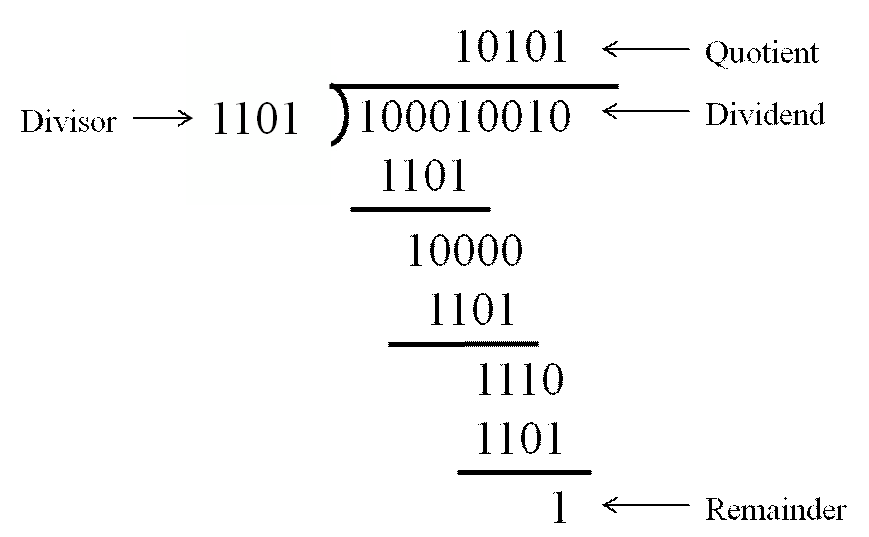
 the divisor is repositioned for another subtraction.

**Example**

 **Decimal**



 **Binary**



**2.4.2 Restoring Division**

**Basic Operation**

 Initially

o An n-bit positive-divisor is loaded into register M

o An n-bit positive-dividend is loaded into register Q at the start of the operation

o Register A is set to 0

 After division operation

o An n-bit quotient is in register Q

o Remainder is in register A

**Steps in division operation**

1. Shift A and Q left one binary position

2. Subtract M from A, and place the answer back in A

 For subtraction, find 2’s complement of M and add with A

o A- B = A + 2’s complement(B)

3. If the sign of A is 1,

  Set q

0

to 0 and add M back to A (restore A)

Otherwise,

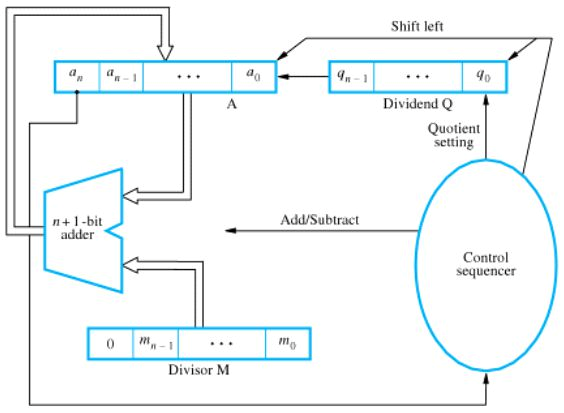
  Set q

0

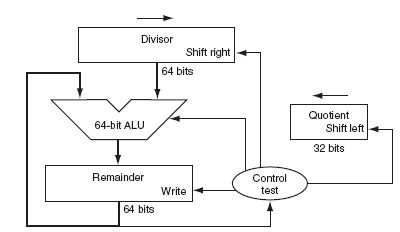
to 1

4. Repeat these steps n times

**Logic Diagram**



**Implementation hardware**



 The 32-bit Quotient register set to 0.

 Every iteration of the algorithm needs to move the divisor to the right one digit.

 We start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend.

 The Remainder register is initialized with the dividend.

 The system first subtract the divisor in step 1

 If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a).

 If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b).

 The divisor is shifted right and then we iterate again.

 The remainder and quotient will be found in their namesake registers after the iterations are complete.

**Revised hardware**

 The following figure shows the revised hardware for multiplication.

 The speedup comes from shifting the operands and the quotient simultaneously with the subtraction.

 This refinement halves the width of the adder and registers by noticing where there are unused portions of registers and adders.

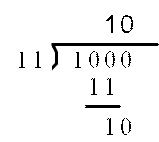
**Example**

 Dividend (Q) : 1000

 Divisor (M) : 11

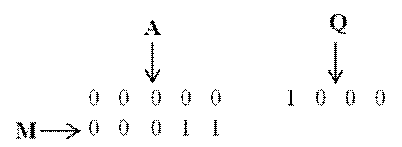
2’s Complement of M

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| M: | **0** | **0** | **0** | **1** | **1** |  |
| 1’s Complement: | 1 | 1 | 1 | 0 | 0 |
| +1 : |  |  |  |  | 1 | + |
| 2’s Complement: | 1 | 1 | 1 | 0 | 1 |  |
| Longhand Division |  |  |  |  |  |  |

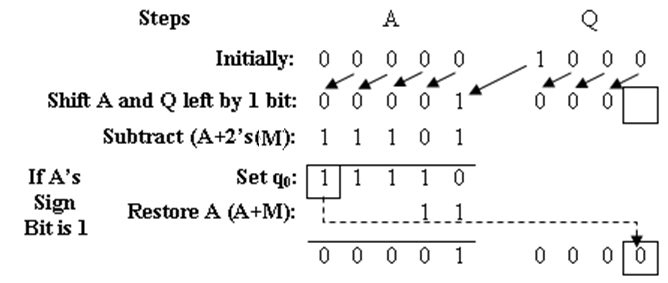


Restoring Division

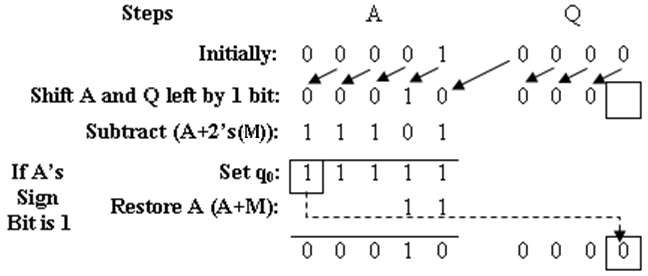
 **Initially**



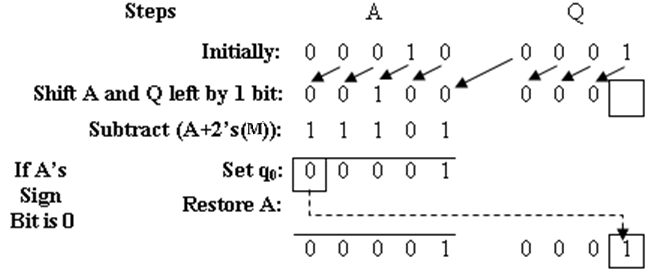
 **First Cycle**



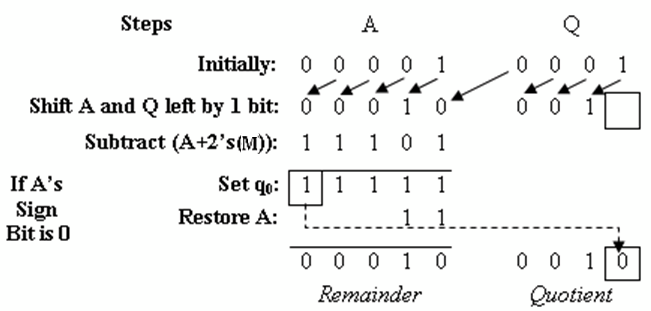
 **Second Cycle**



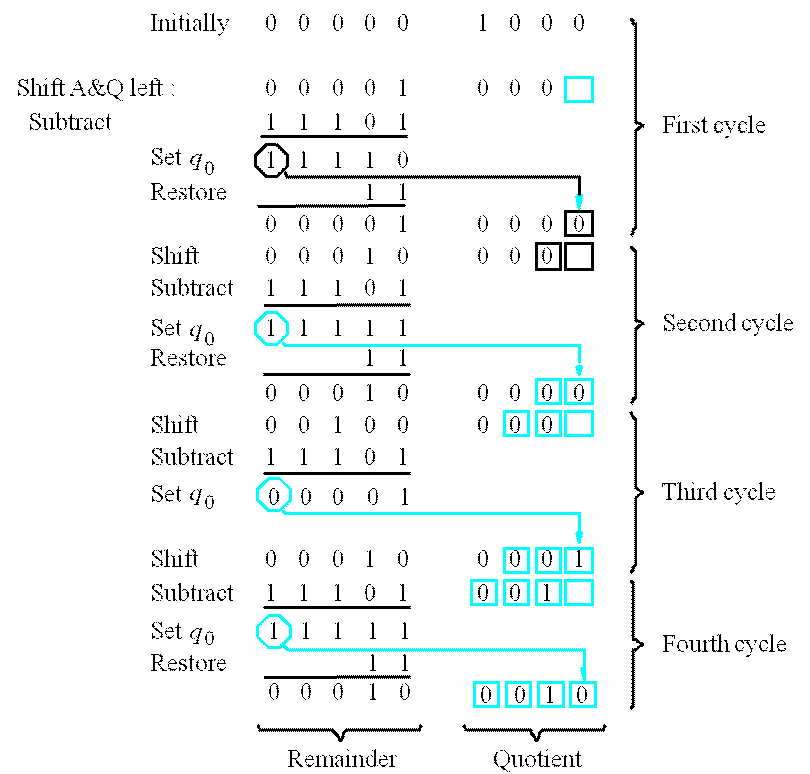
 **Third Cycle**



 **Fourth Cycle**

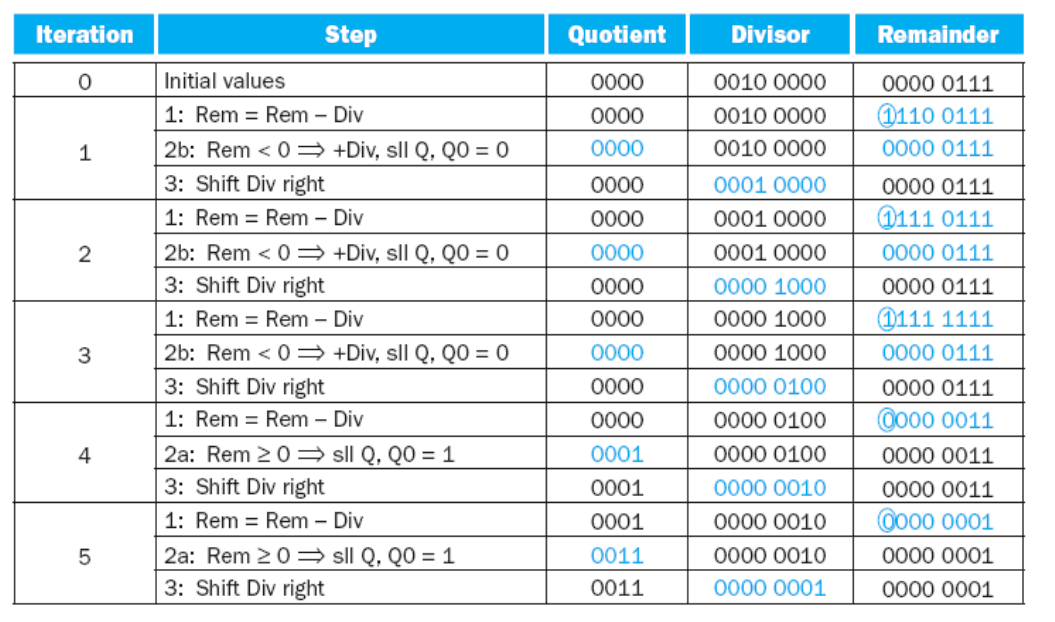


**Illustration**



**Another Example**

Divide: 7



2

10

10

2

by 2

, or 0000 0111

by 0010 .

**2.4.3 Non-restoring Division**

 Avoid the need for restoring A after an unsuccessful subtraction

**Steps**

1. Repeat n times

 If the sign of A is 0,

o Shift A and Q left one bit position and subtract M from A Otherwise

o Shift A and Q left and add M to A

 Now, if the sign of A is 0, set q

0

2. If the sign of A is 1, add M to A

**Example**

 Dividend (Q) : 1000

 Divisor (M) : 11

2’s Complement of M

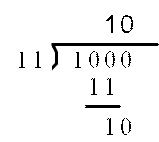
to 1; otherwise, set q

0

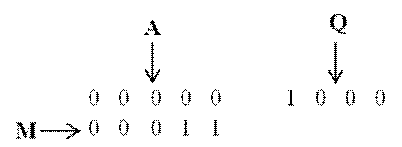
to 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| M: | **0** | **0** | **0** | **1** | **1** |
| 1’s Complement: | 1 | 1 | 1 | 0 | 0 |
| +1 : |  |  |  |  | 1 + |
| 2’s Complement: | 1 | 1 | 1 | 0 | 1 |
| Longhand Division |  |  |  |  |  |

Restoring Division

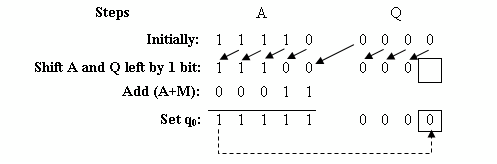
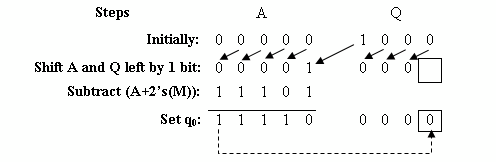


 **Initially**



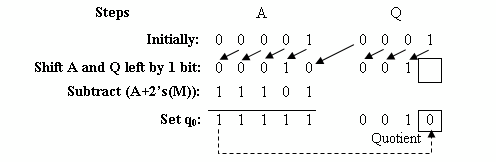
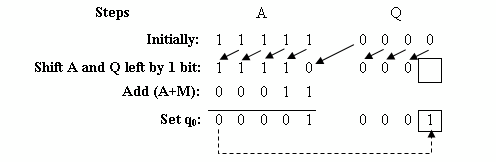
 **First Cycle**

 **Second Cycle**



 **Third Cycle**

 **Fourth Cycle**

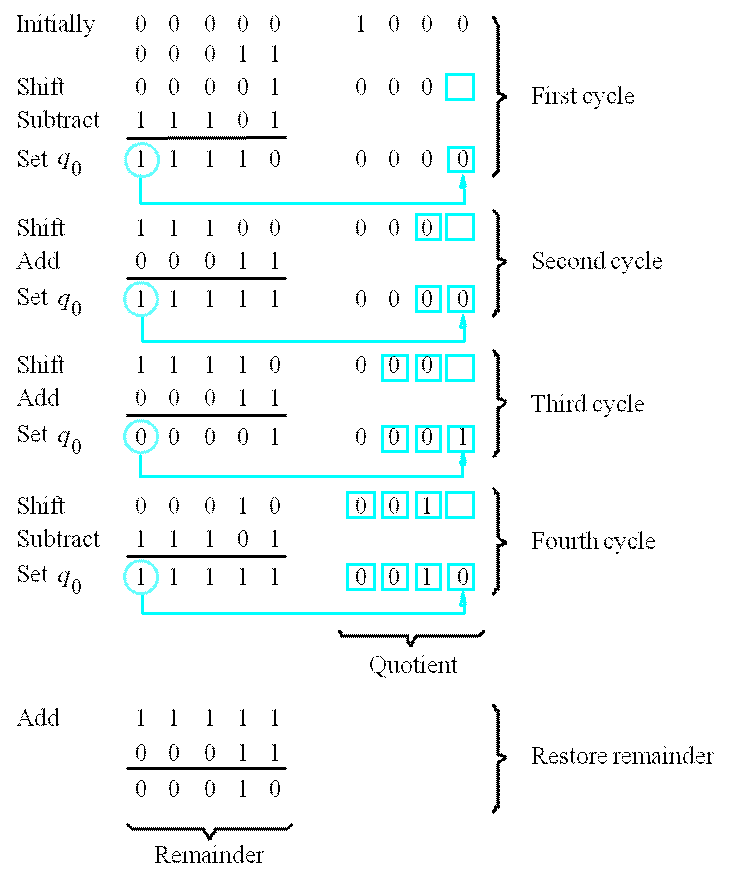


**Restore Remainder**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Add | **1** | **1** | **1** | **1** | **1** |  |
| M: | 0 | 0 | 0 | 1 | 1 | + |
|  | 0 | 0 | 0 | 1 | 0 |  |

Remainder

Illustration



**2.5 FLOATING POINT REPRESENTATION**

**2.5.1 Floating Point Number**

A floating point is a computer arithmetic that represents numbers in which the binary point is not fixed.

The numerical value of a finite number can be represented by four integer components.

 Sign (s)

 Base (b)

 Significant or Fraction (m)

 Exponent (e)

**Representation: (-1)s**  **m**  **be**

**Example:**

6.345×1023

-7.525×10-12

6.642×10-32

**Terminologies**

**Scale factor**

Scale factor indicates the position of the decimal point with respect to the significant

digits.

For the above example, the scale factor is 1023,10-12,10-32

**Fraction (m)** is represented in the following format

Where

m m-1

i

i

…. i

2

i

1

i . F

0

1

F

2

… F

n-1 n

F

i - integer parts

F - fraction parts

**Overflow (floating-point)**

Overflow is a situation in which a positive exponent becomes too large to fit in the exponent field.

**Underflow (floating-point)**

Underflow is a situation in which a negative exponent becomes too large to fit in the exponent field.

One way to reduce chances of underflow or overflow is to choose another format that has a larger exponent – double precision format.

**Double precision**

Double precision is a floating point value that is represented in two 32-bit words.

**Single precision**

Single precision is a floating point value which is represented in a single 32-bit word.

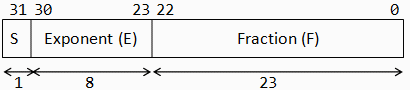
**IEEE format for Single-Precision Floating-Point Numbers**

In 32-bit single-precision floating-point representation: Most significant bit is the sign bit (S),

0 - Positive numbers

1 - Negative numbers

Next 8 bits represent exponent (E) Remaining 23 bits represents fraction (F)



Value represented =  1.F  2E-127

**Example**

0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 …………… 0

Value represented =  1.001010 ..... 0  2-87

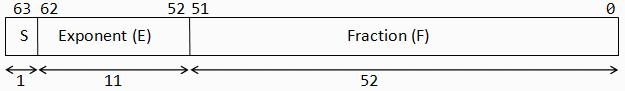
**2.5.3 IEEE format for Double-Precision Floating-Point Numbers**

In 64-bit double-precision floating-point representation: Most significant bit is the sign bit (S),

0 - Positive numbers

1 - Negative numbers

Next 11 bits represent exponent (E) Remaining 52 bits represents fraction (F)



Value represented =  1.F  2E-1023

**2.5.4 Special Values**

 End values in E : 0 and 255

o Used to represent special values

 Different Special Values



|  |  |  |  |
| --- | --- | --- | --- |
| **E Value** | **F Value** | **S Value** | **Represented Value** |
| 0 | 0 | 0 | +0 |
| 1 | -0 |
| 255 | 0 | 0 | + |
| 1 | - |
| 0 |  0 |  | Denormal |
| 255 |  0 |  | NaN (Not a Number) |

**2.5.5 Exceptions**

 In IEEE standard, processor set exception flags when the exception occurs

 Types of Exception

o Underflow

 Occurs when an number requires an exponent less than -126 (for single precision) or -1022(for double precision) to represent it in normalized form

o Overflow

 Occurs when an number requires an exponent greater than

+127 (for single precision) or +1023(for double precision) to represent it in normalized form

o Divide by Zero

 Occurs when any number is divided by zero

o Inexact

 Occurs when any result requires rounding in order to be represented in one of the normal formats

o Invalid

 Occurs when the operations such as

0

0 or

1 are attempted

**2.6 FLOATING POINT OPERATIONS**

**2.6.1 Floating-Point Addition**

 First, convert the two representations to scientific notation. Thus, we explicitly represent the hidden 1.

 In order to add, we need the exponents of the two numbers to be the same.

 This is done by rewriting Y. This will result in Y being not normalized, but value is equivalent to the normalized Y.

 Add x - y to Y’s exponent. Shift the radix point of the mantissa (significant) Y left by x - y to compensate for the change in exponent.

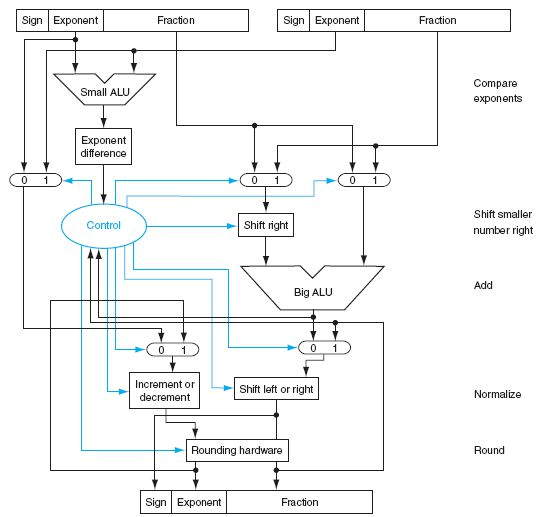
 Add the two mantissas of X and the adjusted Y together.

 If the sum in the previous step does not have a single bit of value 1, left of the radix point, and then adjust the radix point and exponent until it does.

 Convert back to the one byte floating point representation.

**Flowchart**

**Hardware Implementation**



**2.6.2 Floating point Multiplication**

 First, convert the two representations to scientific notation. Thus, we explicitly represent the hidden 1.

 Let **x** be the exponent of **X**. Let **y** be the exponent of **Y**. The resulting exponent (call it **z**) is the sum of the two exponents. **z** may need to be adjusted after the next step.

 Multiply the mantissa of **X** to the mantissa of **Y**. Call this result **m**.

 If **m** is does not have a single 1 left of the radix point, and then adjust the radix point so it does, and adjust the exponent **z** to compensate.

 Add the sign bits, mod 2, to get the sign of the resulting multiplication.

 Convert back to the one byte floating point representation, truncating bits if needed.

**Flowchart**

**2.7 ACCURATE ARITHMETIC**

 Unlike integers, which can represent exactly every number between the smallest and largest number, floating-point numbers are normally approximations for a number they can’t really represent.

 IEEE 754 has two extra bits on the right during intermediate additions, called guard and round, respectively

 **Guard** - The first of two extra bits kept on the right during intermediate calculations of floating point numbers; used to improve rounding accuracy.

 **Round** - Method to make the intermediate floating point result fit the floating point format; the goal is typically to find the nearest number that can be represented in the format.

 IEEE 754 has four rounding modes: always round up (toward +”), always round down (toward – “), truncate, and round to nearest even

 **Sticky bit** - A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit

**2.8 SUBWORD PARALLELISM**

 A Subword is a lower precision unit of data contained within a word.

 In subword parallelism, multiple subwords are packed into a word and then process whole words.

 With the appropriate subword boundaries, this technique results in parallel processing of subwords.

 Since the same instruction is applied to all subwords within the word - form of

SIMD(Single Instruction Multiple Data) processing.

 Possible to apply subword parallelism to noncontiguous subwords of different sizes within a word

 Practical implementation is simple if subwords are same size and they are contiguous within a word

 The data parallel programs that benefit from subword parallelism tend to process data that are of the same size

 Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple

Data (SIMD)

 Subword parallelism is an efficient and flexible solution for media processing because algorithm exhibit a great deal of data parallelism on lower precision data.

 Useful for computations unrelated to multimedia that exhibit data parallelism on lower precision data

**Example**

**Basic idea**

Treat a 64-bit register as a vector of 2 32-bit or 4 16-bit or 8 8-bit values (short vectors)

Partition 64-bit datapaths to handle multiple narrow operations in parallel

 Graphics and audio applications can take advantage of performing simultaneous operations on short vectors

o Example: 128-bit adder:

 Sixteen 8-bit adds

 Eight 16-bit adds

 Four 32-bit adds

These are also called as data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD).