UNIT – II

ARITHMETIC FOR COMPUTERS

Addition and subtraction - Multiplication - Division - Floating Point operations - Subword parallelism.

1. ARITHMETIC AND LOGIC UNIT (ALU)

The Arithmetic and Logic Unit (ALU) is responsible for performing arithmetic operations such as add, subtract, division and multiplication and logical operations such as ANDing, ORing, Inverting etc.

The arithmetic operation to be performed is based on the data type.

Two basic data types are implemented in the computer system:

- Fixed point numbers
- Floating point numbers.

Representing numbers in such data types is commonly known as **fixed point representation** and **floating point representation**.

Based on the complexity of operation done by ALU its design has classified into two types:

1. **Combinational logic circuits based ALU**
2. **Sequential logic circuits based ALU**

Simple ALU perform fixed point addition and subtraction as well as word logical operations, can be realized by COMBINATIONAL CIRCUITS.

Complex ALU perform multiplication and division as well as floating point operation. It can be performed using SEQUENTIAL LOGIC CIRCUITS based ALU.

To construct a ALU we need four basic hardware components such as

1. **AND gates**
2. **OR gates**
3. **Inverters**
4. **Multiplexers**

1.1. ALU CONTROL LINES

ALU is capable of performing both arithmetic and logic operations based control lines. Control line specifies which operation has to be executed by the ALU. It is shown below:

ALU control lines	Function
0000	AND
0001	OR
0010.	Add

0110	Subtract
0111	Set on less than
1100	NOR

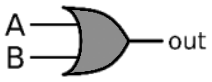
The MIPS word is 32 bits wide so we need a 32 bit wide ALU. It is constructed by connecting 32 1 bit ALU's in series.

1.2. 1-Bit ALU

The logical operations are simpler when compared to arithmetic operations because they map directly onto the hardware components.

1.2.1. OR Operation

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

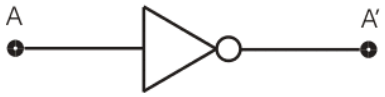


1.2.2. AND Operation

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

1.2.3. NOT Operation

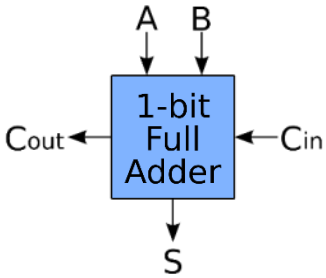
A	$\bar{A}$
0	1
1	0



1.3. Adder Circuit in 1 Bit ALU

Adder is a circuit used to perform addition operation. Based on the input and output adder can be classified into two types.

- Half adder
- Full adder



Generally adder must have two inputs for operands arid a single output for the sum. In some cases we have second output called as **Carry Out**. The carry out from the neighbour adder must be included as an input, so we need a third input. This input is called **Carry In**.

1-bit adder is also called as **full adder** because it has 3 inputs and 2 outputs.

An adder with only the 2 inputs is called a **half adder**.

1.4. SUBTRACTION

The subtraction consists of four possible elementary operations,

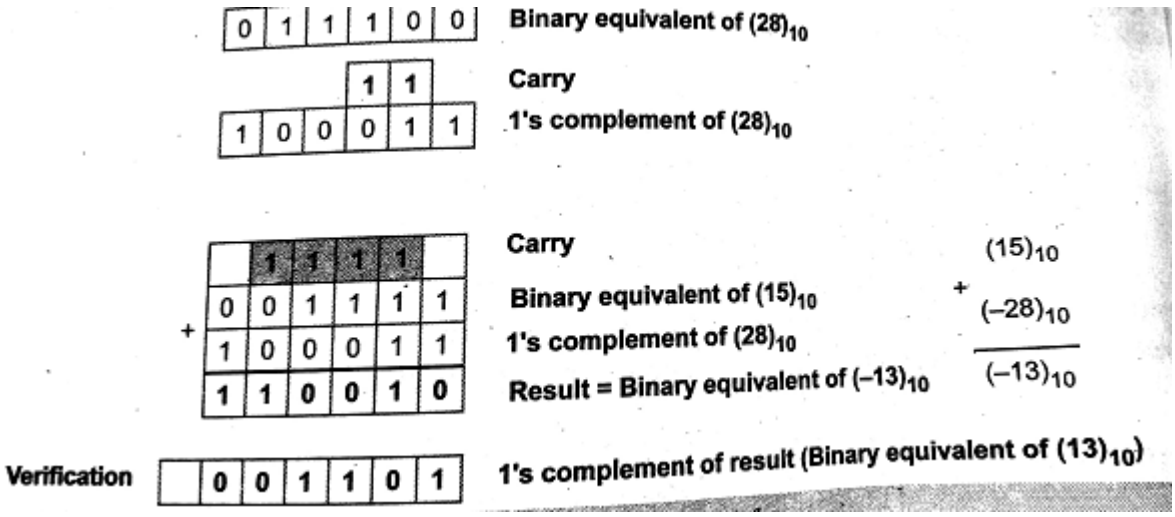
S.NO	OPERATIONS
1.	0-0=0
2.	0-1=1 with 1 borrow
3.	1-0=1
4.	1-1=0

1.4.1. BINARY SUBTRACTION USING 1'S COMPLEMENT METHOD

In a 1's complement subtraction, negative number is represented in the 1's complement form and actual addition is performed to get the desired result.

Eg: operation A-B

1. Take 1's complement of B
2. 2. Result← a+ 1's complement of B
3. If carry is generated then the result is positive and in the true form. Add carry to the result to get the final result.
4. If carry is not generated then the result is negative and in the 1's complement form.
   Example:
   1. Perform $(15)_{10}$ –$(28)_{10}$ using 6-bit 1's complement representation.



1.4.2. BINARY SUBTRACTION USING 2'S COMPLEMENT METHOD

In a 2's complement subtraction, negative number is represented in the complement form and actual addition is performed to get the desired result.

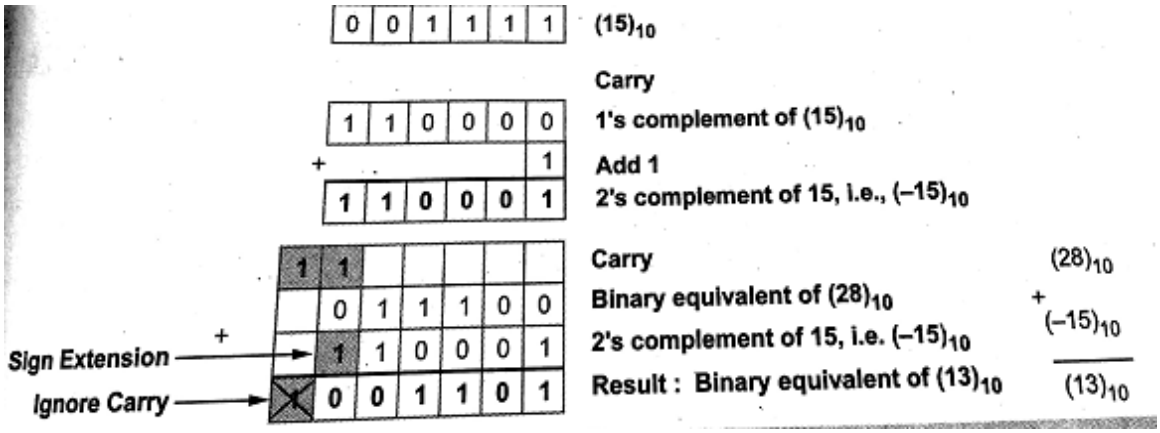
Eg: operation A-B

1. Take 2's complement of B
2. 2. Result← A+ 2's complement of B
3. If carry is generated then the result is positive and in the true form. In this carry is ignored.

4. If carry is not generated then the result is negative and in the 2's complement form.

Example:

1. Perform (28)- (15) using 6 bit 2's complement representation.



Truth table for full adder is shown below:

Inputs			Outputs		Comments
A	B	Carry in	Carry out	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
o	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	o	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

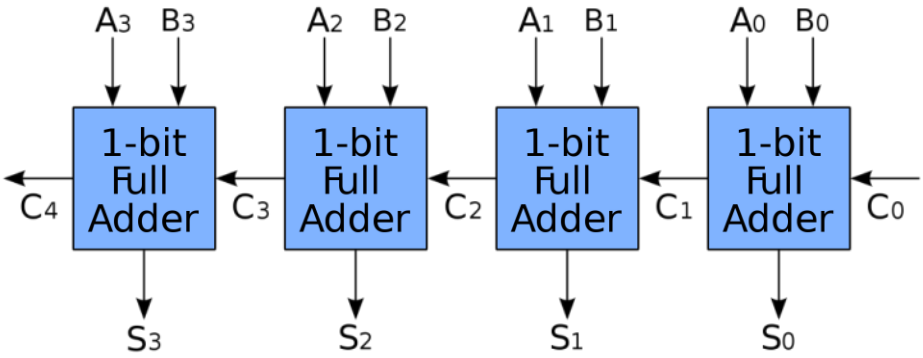
2. DESIGN OF RIPPLE CARRY ADDER

Half Adders can be used to add two one bit binary numbers. It is also possible to create a logical circuit using multiple full adders to add N-bit binary numbers.

Each full adder inputs a $\mathbf{C_{in}}$, which is the $\mathbf{C_{out}}$ of the previous adder.

This kind of adder is a Ripple Carry Adder, since each carry bit "ripples" to the next full adder. The first full adder may be replaced by a half adder.

The block diagram of 4-bit Ripple Carry Adder is shown here below



The layout of ripple carry adder is simple, which allows for fast design time; however, the ripple carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder.

The gate delay can easily be calculated by inspection of the full adder circuit. Each full adder requires three levels of logic.

In a 32-bit ripple carry adder, there are 32 full adders, so the critical path delay is 31 * 2(for carry propagation) + 3(for sum) = 65 gate delays.

$$\boldsymbol{Procpagational\ Delay = nd}$$

Where,

n is the number of full adders used

d is the delay in sec

While using parallel connected full adders, unlike serial adders the amount of hardware needed gets increasing as the number of bits becomes higher.

3. Design of Fast Adder

The ***n-bit*** adder ripple carry adder is implemented using full-adder stages. In which the carry output of each full-adder stage is connected to the carry input of the next higher-order stage.

The sum and carry outputs of any stage cannot be produced until the input carry occurs; this leads to a time delay in the addition process. This delay is known as carry propagation delay, consider the following addition.

```
  0101
+ 0011
------
 1000
```

Addition of the LSB position produces a carry into the second position. This carry, when added to the bits of the second position (stage), produces a carry into the third position. The latter carry, when added to the bits of the third position, produces a carry into the last position.

The key thing to notice in this example is that the sum bit generated in the last position **(MSB)** depends on the carry that was generated by the addition in the previous positions.

This means that, adder will not produce correct result until LSB carry has propagated through the intermediate ***full-adders***. This represents a **time delay** that depends on the **PROPAGATION DELAY** produced in each ***full-adder***.

3.1. PROPAGATION DELAY

If each full-adder has a propagation delay of 30 ns, then propagation delay 90 ms after LSB carry is generated. Therefore, total time required to perform addition is 90 + 30= 120 ns. This situation becomes much worse for more number of bits. If the adder were handling 16-bit numbers, the carry propagation delay could be 480 ns.

Full-adder requires two gate delays and sum requires only one gate delay. When we connect full adder circuits in cascade to generate ***n-bit*** ripple adder. C_{n-1} is available in **2(n-1)** gate delays. The final carry-out, C_n is available after **2n** gate delays.

Thus for 4-bit ripple adder C4 is available after 8(2x4) gate delays, C3 is available in 6[2(4—1)] gate delays and S3 is available in 7 gate delays.

One method of speeding up this process by eliminating inter stage carry delay is called look ahead-carry addition. This method utilizes logic gates to look at the lower-order bits of the augend and addend to see if a higher-order carry is to be generated. It uses two functions: carry generate and carry propagate.

3.2. CARRY LOOK AHEAD CIRCUIT

The full adder defines two functions:

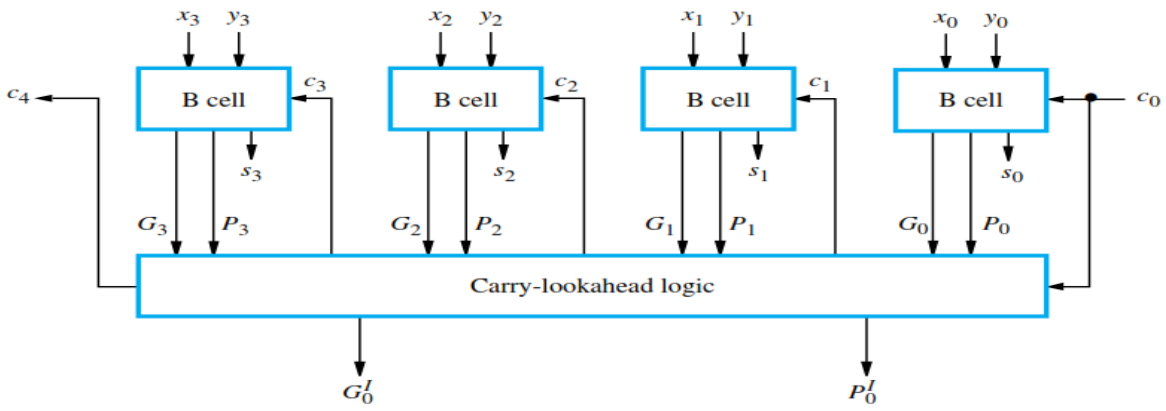
- Carry generate
- Carry propagate.

The output sum and carry can be expressed as

$$P_i = X_i \oplus Y_i$$

$$G_i = X_i Y_i$$

G_i is called a carry generate and it produces on carry when both X_i and Y_i are one, regardless of the input carry.



P_i is called a carry propagate because it is term associated with the propagation of the carry from G_i to C_{i+1}. Now C_{i+1} can be expressed as a sum of products function of the **P** and **G** outputs of all the preceding stages.

For example, the carriers in a four stage carry-look ahead adder are defined as follows:

$$C_1 = G_0 + P_0 C_{in}$$

The Carry for the bit position i can be calculated by using the following formula,

$$C_i = g_i + p_i C_{i-1} \quad (1)$$
$$C_{i-1} = g_{i-1} + p_{i-1} C_{i-2} \quad (2)$$

Substituting equation (2) in (1)

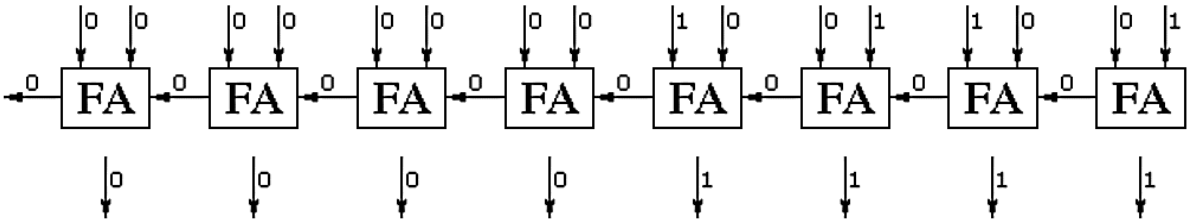
$$C_i = g_i + p_i(g_{i-1} + p_{i-1} C_{i-2}) \quad (3)$$
$$C_i = g_i + p_i g_{i-1} + p_i p_{i-1} C_{i-2} \quad (4)$$

Sum of the ith stage can be computed as follows,

$$S_i = p_i \oplus g_i \oplus C_{i-1}$$

4. Ripple Carry Adder

A	0 0 0 0 1 0 1 0	: 10
B	+ 0 0 0 0 0 1 0 1	: 5
Sum	0 0 0 0 1 1 1 1	: 15

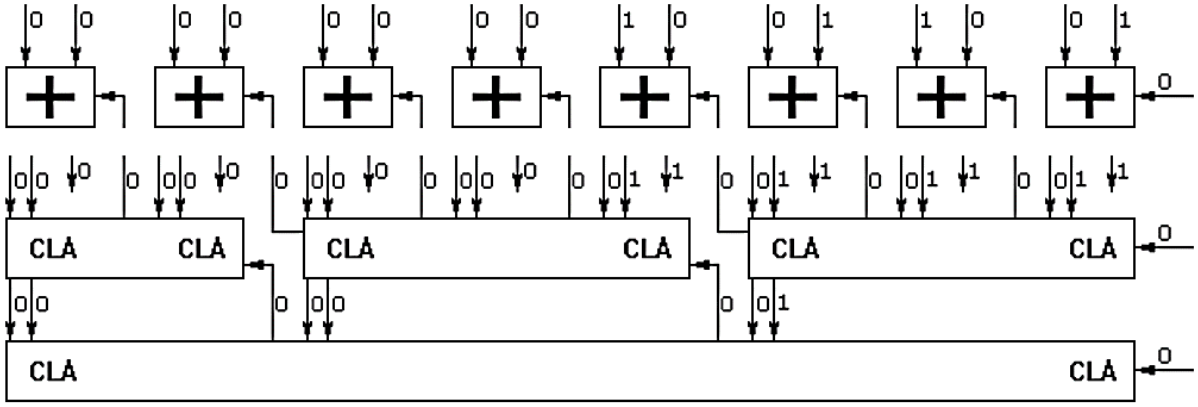


5. Carry Look Ahead Adder

$$P_i = X_i \oplus Y_i$$

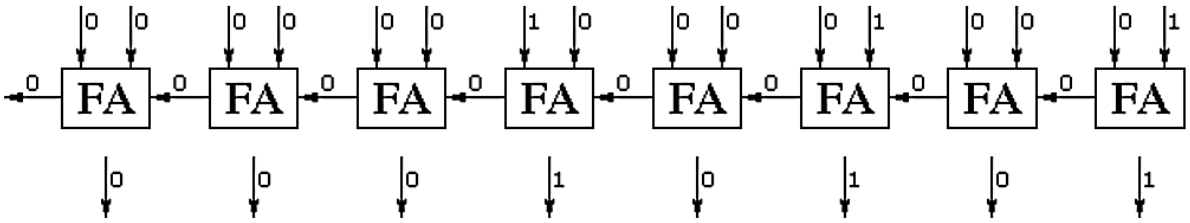
$$G_i = X_i Y_i$$

$$S_i = p_i \oplus g_i \oplus C_{i-1}$$



6. Ripple Carry Adder

A	0 0 0 1 0 0 0 0 : 16
B	+ 0 0 0 0 0 1 0 1 : 5
Sum	0 0 0 1 0 1 0 1 : 21

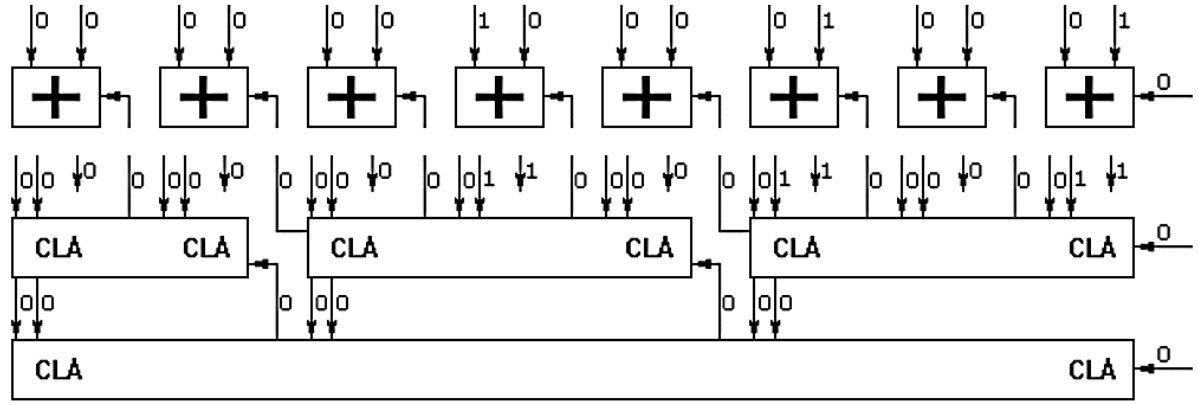


7. Carry Look Ahead Adder

$$P_i = X_i \oplus Y_i$$

$$G_i = X_i Y_i$$

$$S_i = p_i \oplus g_i \oplus C_{i-1}$$



8. SEQUENTIAL MULTIPLICATION

The multiplication of binary number is done in the same way as decimal numbers are multiplied. Consider the example, Multiplying **1010** by **1000**

Multiplicand				1	0	1	0
Multiplier			X	1	0	0	0
				0	0	0	0
			0	0	0	0	
		0	0	0	0		
	1	0	1	0			
Product	1	0	1	0	0	0	0

The first operand is called the **MULTIPLICAND** and second operand is called the **MULTIPLIER**, the final result is called the **PRODUCT**

Take the digits of the multiplier one at the time from right to left, multiplying the multiplicand by the single digit of the multiplier.

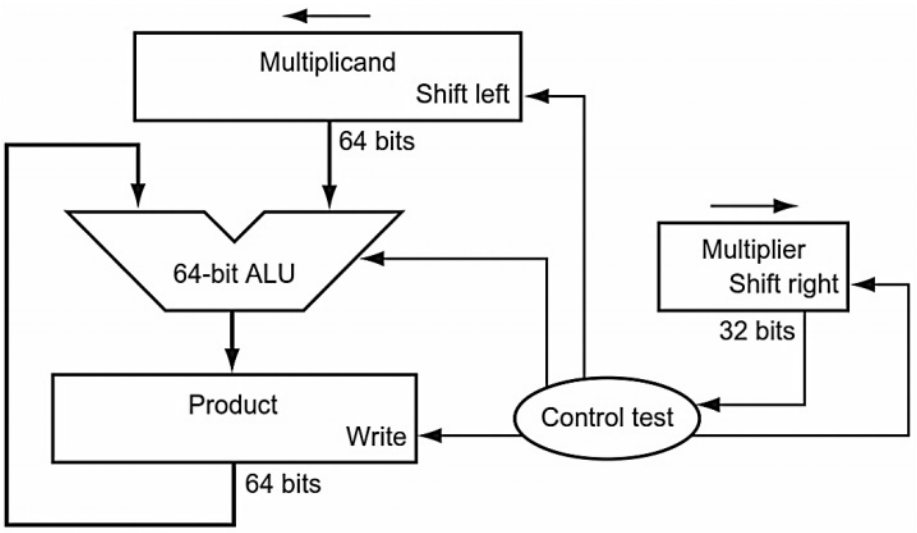
Shifting the intermediate product one digit to the left of the earlier intermediate products.

The numbers of digits in the product is larger than the number of digits in either the multiplicand or the multiplier.

Multiplication is usually implemented by some form of ***repeated addition***.

To compute X *x* Y is to add the multiplicand Y to itself X times, where X is the multiplier

8.1. MULTIPLICATION HARDWARE



The multiplicand register, ALU and product register are all 64 bit and multiplier is 32 bits. The 32 bit multiplicand starts in the right half of the **multiplicand** register and is **shifted left 1 bit** on each step.

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

The multiplication algorithm starts with the product initialized to 0. ***Control decides when to shift*** the multiplicand and multiplier registers and when to write new values into the **product register**.

8.2. MULTIPLICATION ALGORITHM

STEP 1: The least significant bit of the multiplier (multiplier 0) determines whether the multiplicand is added to the product register.

STEP 2: The left shift has the effect of moving the intermediate operands to the left.

STEP 3: The shift right gives the next bit of the multiplier to examine in the following iteration.

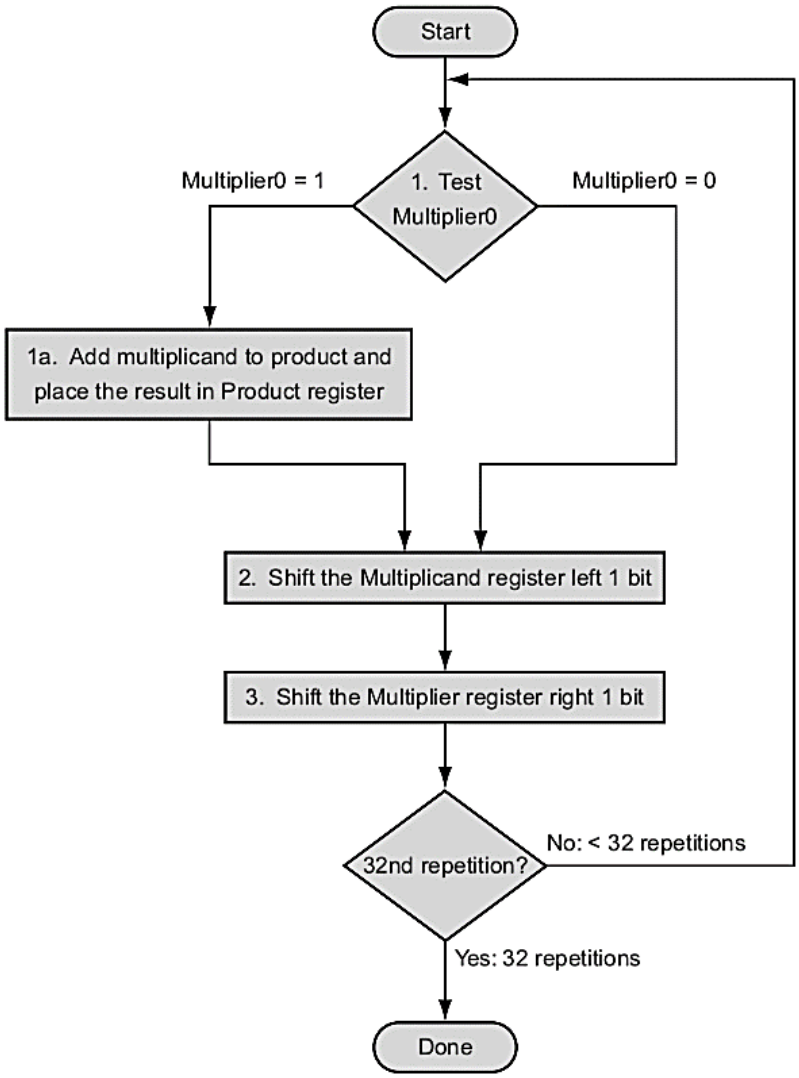
STEP 4: These three steps are repeated 32 times to obtain the product.

8.2.1. CLOCK CYCLES FOR MULTIPLICATION ALGORITHM

This algorithm requires almost 100 clock cycles to multiply two 32 bit numbers.

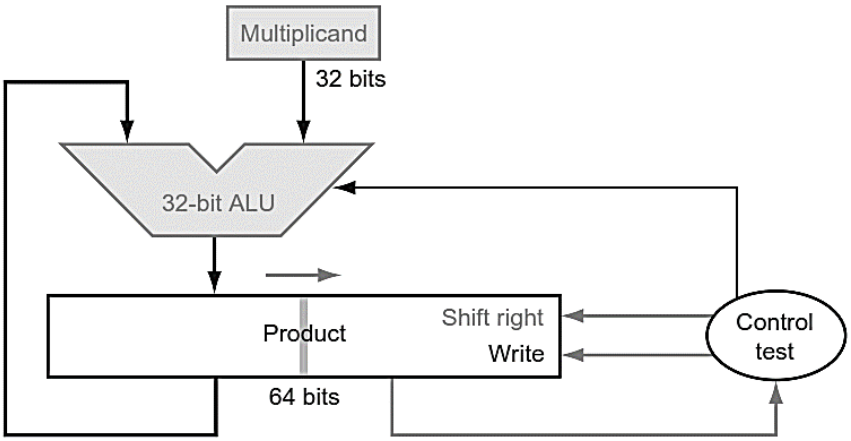
Multiply take multiple clock cycles without affecting performance.

We can increase the speed of process by performing the operations in parallel manner. That means, the multiplier and multiplicand are shifted while the multiplicand is added to the product ***if the multiplier bit is a 1***



8.3. REFINED MULTIPLICATION HARDWARE

The hardware is used to ensure that it is the right bit of the multiplier and gets the pre-shifted version of the multiplicand.



Hardware will take 1 clock cycle per step.

Compare with the first version of hardware, the multiplicand register, ALU and multiplier register are all 32 bits.

The product register only has 64 bits. The product is shifted right. The separate multiplier register also disappeared and the multiplier is placed instead in the right half of the product register.

8.4. EXAMPLE

Multiply 2_{10} by 3_{10}

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001①	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	001①	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000⓪	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000⓪	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

A	0 0 0 0 1 1 0 1	13
X	x 0 0 0 0 0 1 1 0	6
Y	0 0 0 0 1 0 -1 0	recoded multiplier
Shift Only	0 0 0 0 0 0 0 0	
Add -A	+ 1 1 1 1 0 0 1 1	
	1 1 1 1 0 0 1 1 0	
Shift	1 1 1 1 1 0 0 1 1 0	

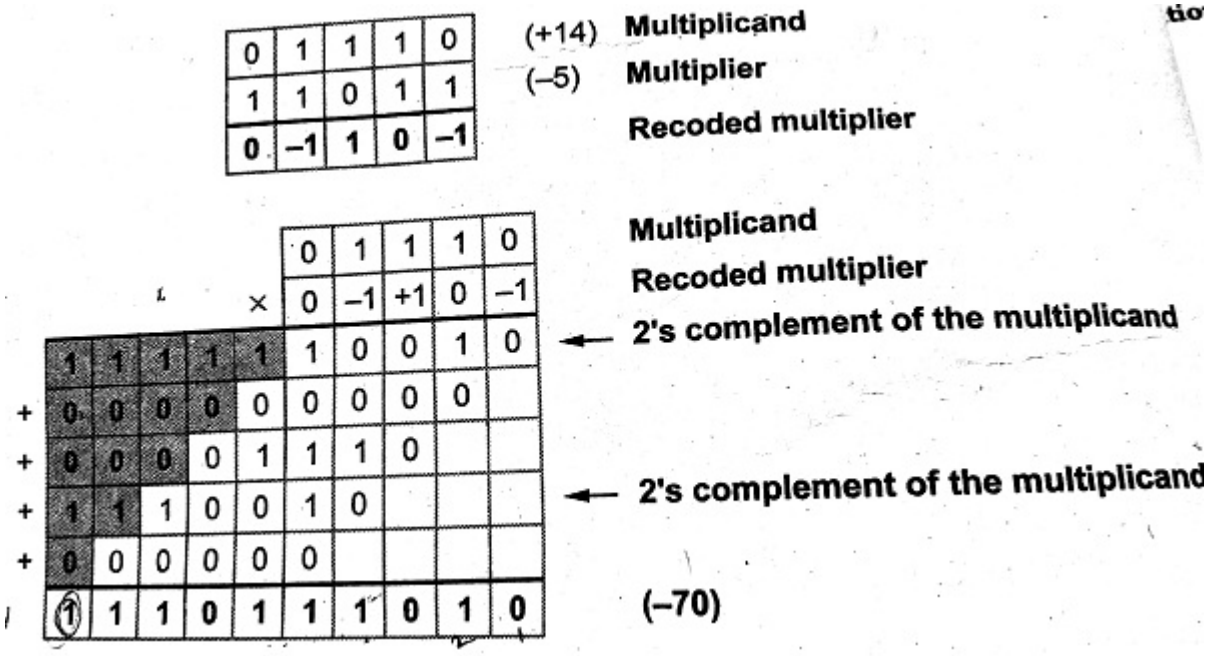
Shift Only	1 1 1 1 1 1 0 0 1 1 0
Add A	+ 0 0 0 0 1 1 0 1
	0 0 0 0 1 0 0 1 1 1 0
Shift	0 0 0 0 0 1 0 0 1 1 1 0
Shift Only	0 0 0 0 0 0 1 0 0 1 1 1 0
Shift Only	0 0 0 0 0 0 0 1 0 0 1 1 1 0
Shift Only	0 0 0 0 0 0 0 0 1 0 0 1 1 1 0
Shift Only	0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 0 78

8.5. SIGN MULTIPLICATION-BOOTH'S ALGORITHM

A powerful algorithm for signed number multiplication is a booth's algorithm, which generates a 2n-bit product and treats both positive and negative numbers uniformly.
In general, for booth's algorithm recoding scheme can be given as:

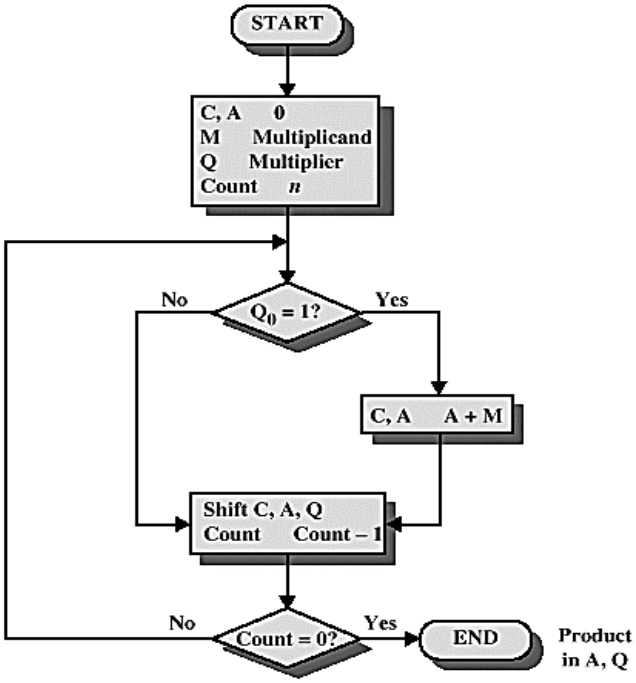
Multiplier		Version of multiplier selected by bit i
Bit i	Bit i-1	
0	0	0
0	1	1
1	0	-1
1	1	0

Example:
Multiply 01110(+14) and 11011(-5)



9. BOOTH`S ALGORITHM FOR MULTIPLICATION

Step 1: Load A = 0, Q_{-1} =0
B = Multiplicand
Q = Multiplier
SC = n
Step 2: Check the status of Q_0Q_{-1}
if Q_0Q_{-1} = 10 Perform $A \leftarrow A - B$
if Q_0Q_{-1} = 01 perform $A \leftarrow A + B$
Step 3: Arithmetic shift right: A, Q, Q_{-1}
Step 4: Decrement sequence counter
if not zero, repeat step 2 through 4
Step 5: Stop



Example: Multiply (-7) and (3) by using booth's multiplication. Give the flow table of multiplication.

Multiplicand (B) = (–7) = 1 0 0 1 , Multiplier (Q)= (3) = 0 0 1 1

SC		A				Q					Operation
		A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
1 0 0		0	0	0	0	0	0	1	1	0	Initial
		0	0	0	0	A					$Q_0 Q_{-1}$ = 1 0
	+	0	1	1	1	2's complement of B					∴ A ← A – B
		0	1	1	1	0	0	1	1	0	
0 1 1		0	0	1	1	1	0	0	1	1	Arithmetic shift right
0 1 0		0	0	0	1	1	1	0	0	1	Arithmetic shift right
		0	0	0	1	A					$Q_0 Q_{-1}$ = 0 1
	+	1	0	0	1	B					
		1	0	1	0	1	1	0	0	1	∴ A ← A + B
0 0 1		1	1	0	1	0	1	1	0	0	Arithmetic shift right
0 0 0		1	1	1	0	1	0	1	1	0	Arithmetic shift right
Result : =		1	1	1	0	1	0	1	1		= –21

9.1. BIT PAIR RECODING OF MULTIPLIERS

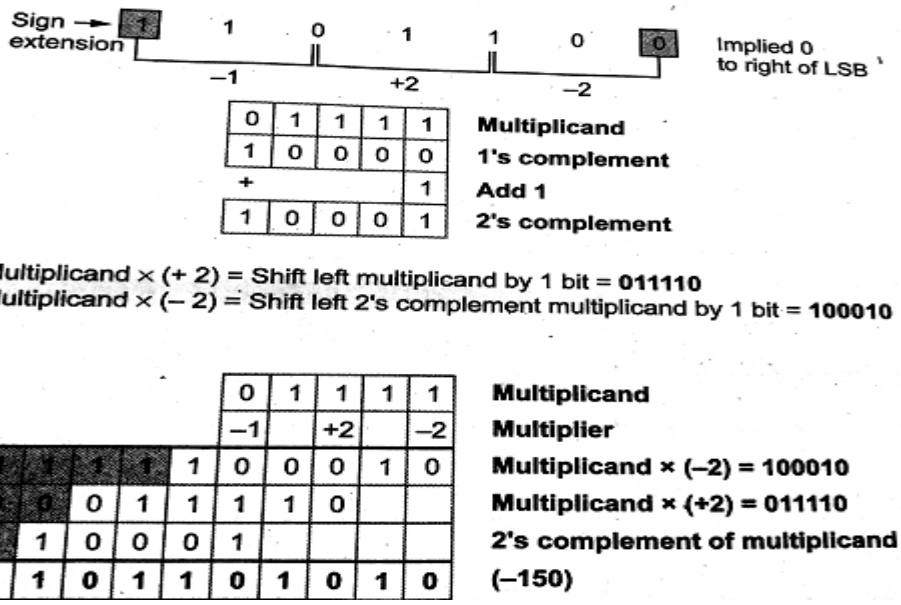
To speed up the multiplication process in the booth's algorithm a technique called bit pair recoding is used. It is also called modified booth's algorithm.

Multiplier bit Pair		Multiplier bit on the right	Bit-Pair recoded multiplier bit at position i
i+1	i	i-1	
0	0	0	0
0	0	1	+1
0	1	0	+1
0	1	1	+2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Eg:Solve the following using bit-pair recoding method.

Multiplicand=01111(15)

Multiplier=10110 (-10)



10. DIVISION

The division process for binary numbers is similar to the decimal numbers. In the division process, first the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor.

Until this condition occurs, 0's are placed in the quotient from left to right.

When the condition is satisfied, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a **PARTIAL REMAINDER**.

From this point onwards, the division process is required. In each repetition cycle, additional bits from the dividend are brought down to the partial remainder until the result is greater than or equal to the divisor and the divisor is subtracted from the result to produce a new partial remainder.

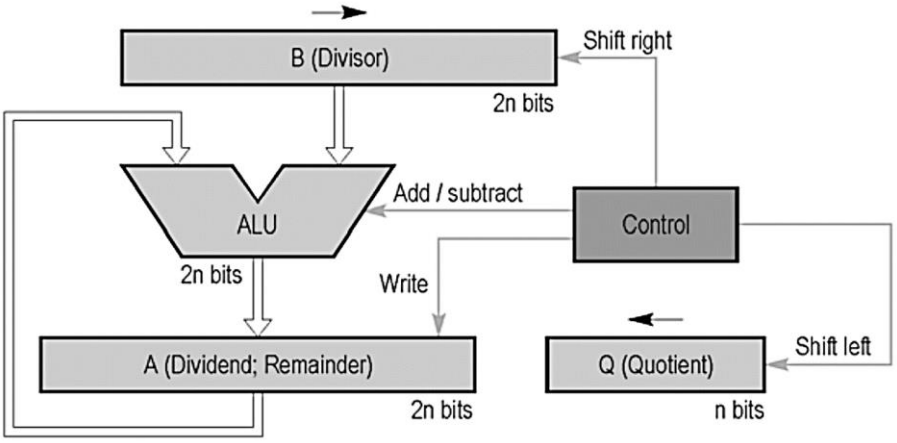
The process continues until all the bits of the dividend are brought down and result is still less than the divisor.

10.1. RESTORING DIVISION ALGORITHM

It consists of **n+1 bit** binary adder, shift, add and subtract control logic and registers A, B, and Q.

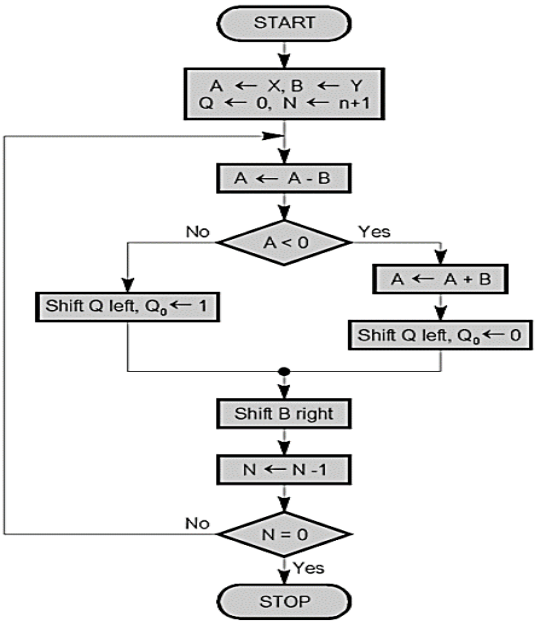
Divisor and **dividend** are loaded into ***register B and register Q***, respectively. Register A is initially set to zero. The division operation is then carried out.

After the division is complete, the n-bit quotient is in register Q and the remainder is in register A.



10.1.1. OPERATION STEPS

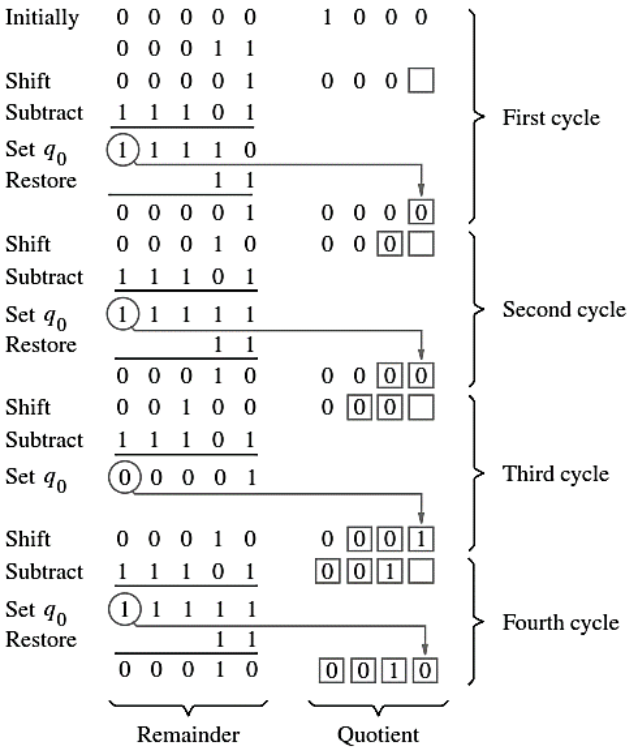
1. Shift A and Q left one binary position.
2. Subtract divisor (i.e. add 2's complement of divisor (B)) from A and place answer back in A $(A \leftarrow A - B)$.
3. If the sign bit of A is 1, set Q0 to 0 and add divisor back to A (that is, restore A); Otherwise, set Q0 to 1.
4. Repeat steps 1, 2, and 3 **n** times.



10.1.2. EXAMPLE

Dividend: 1000

Divisor: 0011

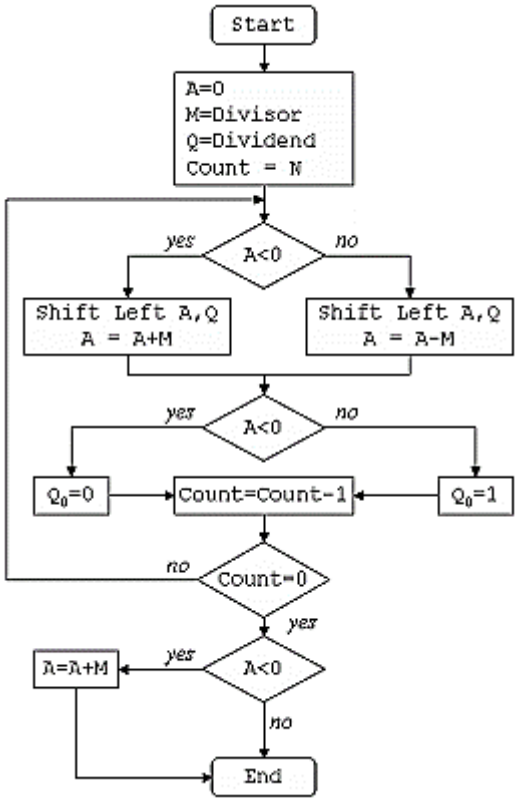


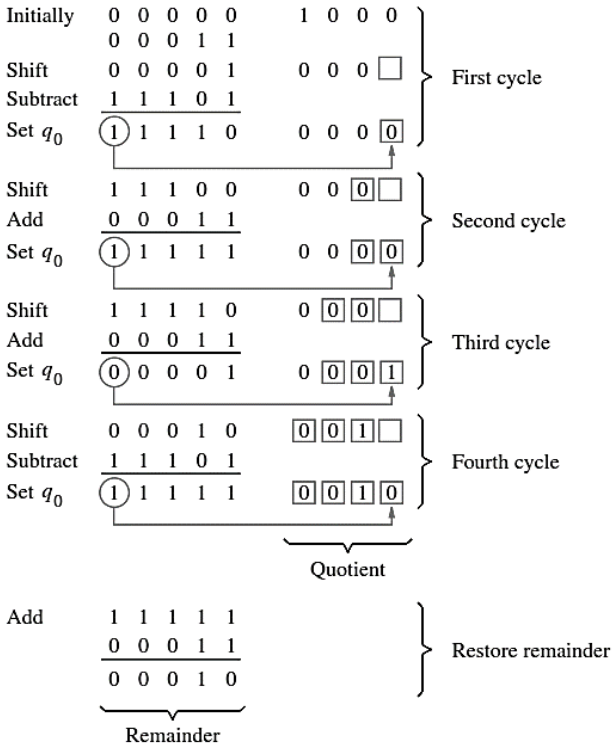
10.2. NON-RESTORING DIVISION

STEP 1: If the sign of A is 0, shift A and Q left one bit position and subtract divisor from A; otherwise, shift A and Q left and add divisor to A. If the sign of A is 0, set Q_0 to 1; otherwise, set Q_0 to 0.

STEP 2: Repeat steps 1 and 2 for n times.

STEP 3: If the sign of A is 1, add divisor to A.





11. FLOATING POINT REPRESENTATION

In addition to signed and unsigned numbers, programming languages support numbers with fractions called reals in mathematics. Let's consider some examples for real numbers.

3.14159265 (Pi)

2.71828(e)

0.00000000.1 or 1.0 x 10-9 (seconds in nanoseconds)

3,155,760,000 or 3.15576 x 109 (seconds in a century)

The last two numbers are bigger than 32 bit signed integer. These numbers is called scientific notation.

11.1. SCIENTIFIC NOTATION

It is a notation that renders numbers with a single digit to the left of the decimal point.

A number in scientific notation has no leading 0's is called a **NORMALIZED NUMBER**.

Binary number also can be represented in the normalized form. To keep a binary number in normalized form, we need a base value that can increase or decrease by exactly the .number of bits the number must be shifted to have one non zero digit to the left of the decimal point Computer arithmetic supports such number is called **FLOATING POINT**

11.1.1. ADVANTAGES OF SCIENTIFIC NOTATION

- It simplifies exchange of data that includes floating point numbers.
- It simplifies the floating point arithmetic algorithm to know that numbers will always be in this form.
- It increases the accuracy of the numbers that can be stored in word.

11.2. FLOATING POINT REPRESENTATION IN MIPS

A designer of a floating point representation must find a compromise the size of the fraction and the size of the exponent. Because a fixed word size means we must take a bit from one to add a bit to the other.

Fraction is the value generally between 0 and 1 it is called the **mantissa**. **Exponent** is the numerical representation system of floating point arithmetic.

Increasing the size of the fraction enhances the precision of the fraction. Increasing the size of the exponent increases the range numbers that can be represented.

Floating point numbers are usually a multiple of the size of a word. The representation of a MIPS floating point number is shown below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

S	Exponent	Fraction
1 bit	8 bits	23 bits

where S is the sign of the floating point number (1 – Negative & 0 – Positive)

Exponent is the value of the 8-bit exponent field. Fraction is the 23bit number. In general floating point numbers are of the form

$(1)^{s}F2^{E}$

F - Involves the value in the fraction field

E-involves the value in the exponent field

Thus cause the overflow interrupts in floating point arithmetic as well as in integer arithmetic.

11.2.1. OVERFLOW (FLOATING POINT)

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

Overflow means the exponent is too large to be represented in the exponent field.

11.2.2. UNDERFLOW (FLOATING POINT)

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

Underflow occurs when the negative exponent is too large to fit in the exponent field.

Both underflow and overflow condition will occur in floating point arithmetic. One way to reduce chances of underflow or overflow is to offer another format that has a larger exponent

11.2.3. DOUBLE PRECISION

It is a floating point value represented in two 32-bit words.

The representation of a double precision floating point number take two MIPS words as shown below:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

S	Exponent	Fraction
1 bit	11 bits	20 bits

where

S is the sign of the number

Exponent is the value of the 11 bit exponent field

12. FLOATING POINT OPERATION

12.1. FLOATINH POINT ADDIDTION

To illustrate the floating point addition, let us consider the following example $\mathbf{9.999_{ten} \times 10^{1} + 1.610_{ten} \times 10^{-1}}$

Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Step 1:

To add these numbers we must align the decimal point of the number that has the smaller exponent. We need to convert the smaller number, $\mathbf{1.610_{ten} \times 10^{-1}}$ to match the larger exponent.

$$\mathbf{1.610_{ten} \times 10^{-1} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}}$$

Step 2:

Next step is the addition of the significands

$$\begin{array}{rr} & 9.999_{ten} \\ + & 0.016_{ten} \\ \hline & 10.015_{ten} \end{array}$$

The sum is $\mathbf{10.015_{ten} \times 10^{1}}$

Step 3:

This sum is not in normalized scientific notation, so we need to adjust it:

$$\mathbf{10.015_{ten} \times 10^{1} = 1.0015_{ten} \times 10^{2}}$$

Whenever the exponent is increased or decreased, we must check for overflow or underflow.

Step 4:

Since we assumed that the significand can be only four digits long (excluding the sign), we must round the number. If the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number

$$\mathbf{1.0015_{ten} \times 10^{2}}$$

is rounded to four digits in the significand to

$$\mathbf{1.002_{ten} \times 10^{2}}$$

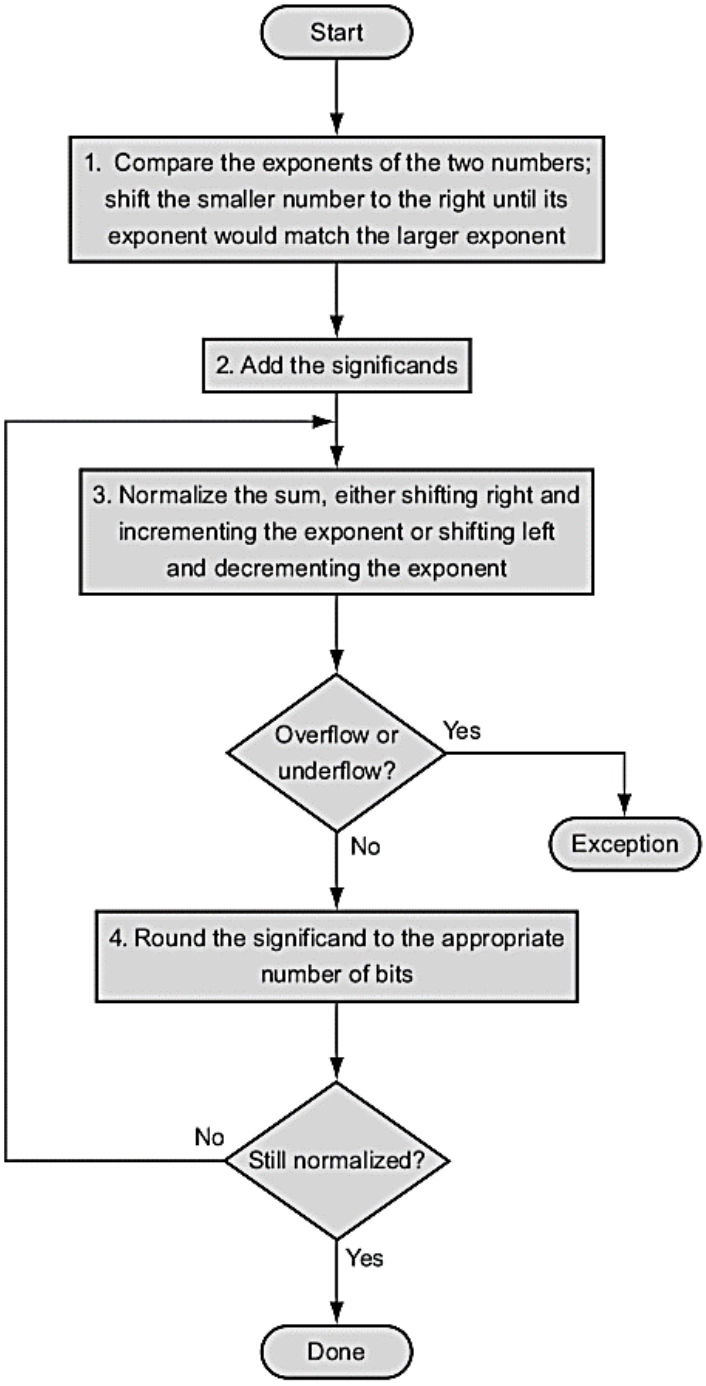
12.1.1. FLOATING POINT ADDITION ALGORITHM

Steps 1 and 2 are similar to the example, adjust the significand of the number with the smaller exponent and then add the two significands.

Step 3 normalizes the results, forcing a check for overflow or underflow. The test for overflow and underflow in step 3 depends on the precision of the operands.

The pattern of all 0 bits in the exponent is reserved and used for the floating-point representation of zero.

The pattern of all 1 bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers



12.2. FLOATING POINT MULTIPLICATION

To illustrate the floating point addition, let us consider the following example

$$(1.110_{ten} \times 10^{10}) \times (9.200_{ten} \times 10^{-5})$$

Assume that we can store only four digits of the significand and two digits of the exponent

Step 1

The exponent of the product is calculated by simply adding the exponents of the operands together

$$New\ exponent = 10 + (-5) = 5$$

Step 2

Next comes the multiplication of the significands

$$\begin{array}{r} 1.110_{ten} \\ \times\ 9.200_{ten} \\ \hline 0000 \\ 0000 \\ 2220 \\ 9990 \\ \hline 10212000_{ten} \end{array}$$

There are three digits to the right of the decimal point for each operand, so the decimal point is placed six digits from the right in the product significand

$$\mathbf{10.21200}_{ten}$$

Assuming that we can keep only three digits to the right of the decimal point, the product is

$$\mathbf{10.212 \times 10^5}$$

Step 3

This product is unnormalized, so we need to normalize it:

$$\mathbf{10.212 \times 10^5 = 1.0212 \times 10^6}$$

After the multiplication, the product can be shifted right one digit to put it in normalized form by adding 1 to the exponent.

At this point, we can check for overflow and underflow. Underflow may occur if both operands are small that is, if both have large negative exponents.

Step 4

We assumed that the significand is only four digits long (excluding the sign), so we must round the number. The number

$$\mathbf{1.0212 \times 10^6}$$

is rounded to four digits in the significand to

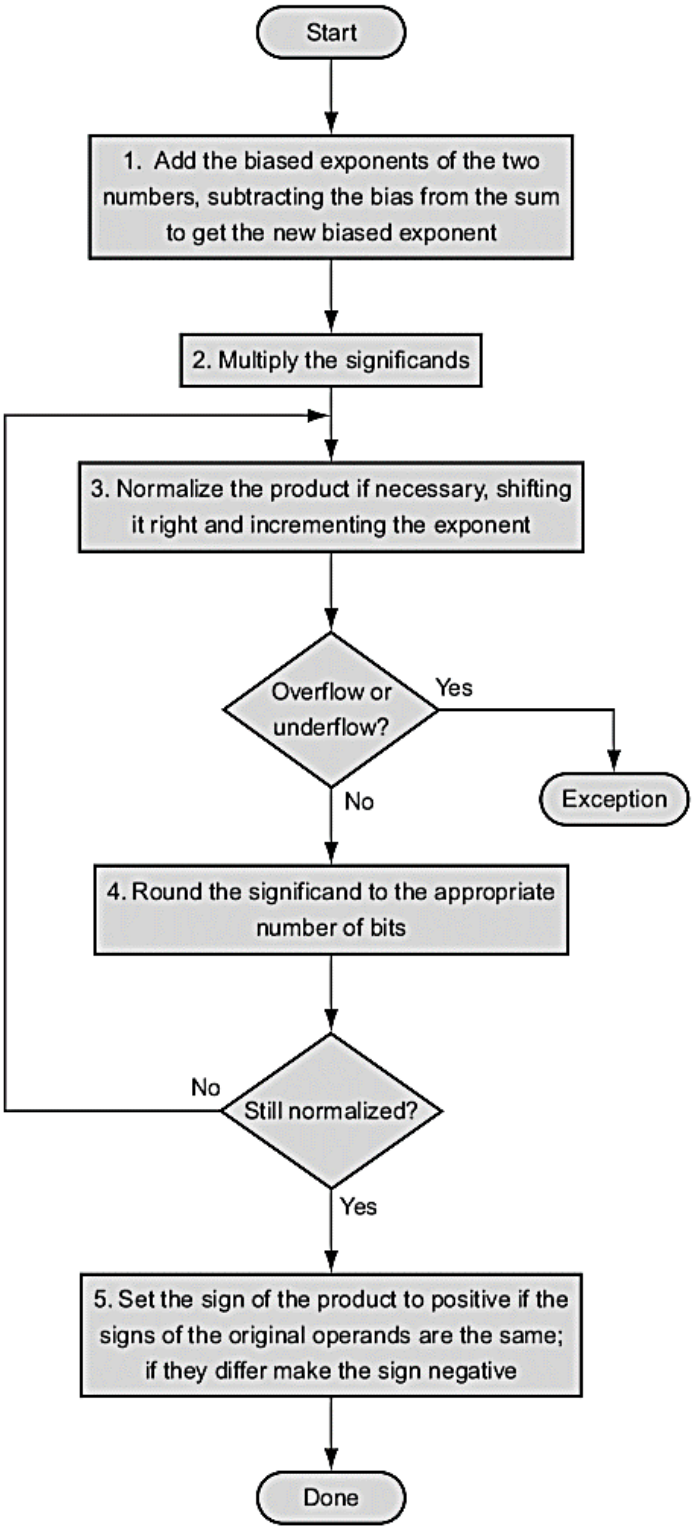
$$\mathbf{1.021}_{ten} \mathbf{\times 10^6}$$

Step 5

The sign of the product depends on the signs of the original operands. If they are both the same, the sign is positive; otherwise, it's negative.

Hence, the product is

$$+\mathbf{1.021}_{ten} \mathbf{\times 10^6}$$



13. SUBWORD PARALLELISM

- Every desktop microprocessor by definition has its own graphical displays, as transistor budgets increased it was inevitable that support would be added for graphics operations.
- Many graphics systems originally used 8 bits to represent each of the three primary colors plus 8 bits for a location of a pixel.

- The addition of speakers and microphones for teleconferencing and video games suggested support of sound as well.
- Audio samples need more than 8 bits of precision, but 16 bits are sufficient.
- Every microprocessor has special support so that bytes and halfwords take up less space when stored in memory (see Section 2.9), but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there was little support beyond data transfers.
- Architects recognized that many graphics and audio applications would perform the same operation on vectors of this data.
- By partitioning the carry chains within a 128-bit adder, a processor could use parallelism to perform simultaneous operations on short vectors of sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands.
- The cost of such partitioned adders was small. Given that the parallelism occurs within a wide word, the extensions are classified as subword parallelism.
- It is also classified under the more general name of data level parallelism. They have been also called vector or SIMD, for single instruction, multiple data (see Section 6.6). The rising popularity of multimedia applications led to arithmetic instructions that support narrower operations thcan easily operate in parallel.
- For example, ARM added more than 100 instructions in the NEON multimedia instruction extension to support subword parallelism, which can be used eithwith ARMv7 or ARMv8.
- It added 256 bytes of new registers for NEON that can bviewed as 32 registers 8 bytes wide or 16 registers 16 bytes wide. NEON supporall the subword data types you can imagine except 64-bit fl oating point numbers:
    - 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers.
    - 32-bit fl oating point numbers.

PART – A

ARITHMETIC LOGIC UNIT

1. What is ALU?[May/june-2016]

An arithmetic logic unit (ALU) is a digital electronic circuit that performs arithmetic and bitwise logical operations on integer binary numbers. It is a fundamental building block of the central processing unit (CPU)

2. State two basic data types implemented in the computer system.

Based on the number system two basic data types are implemented in the computer system:

- Fixed point numbers
- Floating point numbers.

Representing numbers in such data types is commonly known as ***fixed point representation*** and ***floating point representation***.

3. Write a note on scalar data types.

All of the commonly used data types such as, char, int, short long, float, and double are called scalar or base data types because they hold a single data item.

4. What is signed magnitude representation?

Signed magnitude representation is a scheme for representing negative integers. It uses one bit to indicate the sign. "0" indicates a positive integer, and "1" indicates a negative integer. The rest of the bits are used for the magnitude of the number. So $\mathbf{-24_{10}}$ is represented as:

1001 1000

The sign "1" means negative

The magnitude is 24 (in 7-bit binary)

5. What is one's complement of numbers? Find l's complement of $(10101100)_2$

The l's complement of a binary number is the number that results when we change all 1/s to zeros and the zeros to ones.

6. What is 2's complement of numbers? Find 2's complement of $(01011011)_2$

The 2's complement is the binary number that results when we add 1 to the l's complement. It is given as 2's complement = l's complement +1.

ADDITION AND SUBTRACTION

7. How do you relate addition and subtraction?

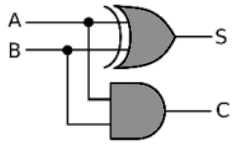
We can relate addition and subtraction operations of numbers by the following relationship:

$$(\pm A) - (+B) = (\pm A) + (-B)$$

$$(\pm A) - (-B) = (\pm A) + (+B)$$

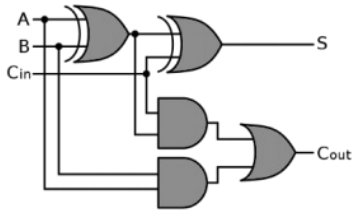
8. What is half adder? Draw the half adder circuit

The logic circuit which performs addition of two binary bits is called a half-adder.



9. What is full adder? Draw the full-adder circuit.

The circuit which performs addition of three bits (two significant bits and a previous carry) is a full-adder.



10. What is a ripple carry adder

The n-bit parallel adder using ***n*** number of full-adder circuits connected in cascade, i.e. the carry output of each adder is connected to the carry input of the next higher-order adder is called ripple carry adder.

11. Define overflow rule in addition.[Nov/Dec-2015][Nov/Dec-2016]

When both operands **a** and **b** have the same sign, an overflow occurs when the sign of result does not agree with the signs of **a** and **b**.

(+A)+(+B)= -C

(-A)+(-B)= +C

12. How overflow occur in subtraction? [May/June 2015]

Overflow in subtraction occurs when the subtrahend is larger than the minuend.

(+A)-(-B)= -C

(-A)-(+B)= +C

Example:

0010 0011
(-) 0110 1110
Will cause overflow

13. Subtract $(11011)_2$ - $(100011)_2$ using 2's complement.[Nov/Dec-2017].

2's complement=1's complement +1

2's complement of10011 is

```
  1  0  0  1  1
  0  1  1  0  0   1's complement
              1
  0  1  1  0  1   2's complement

  1  1  0  1  1
  0  1  1  0  1   2's complement of 10011
 [1] 0  1  0  0  0
```

Discard the carry.

Result is 01000

14. What is guard bit? What are the ways to truncate guard bit?[Nov/Dec-2016].

The first two extra bits kept on the right during intermediate calculations of floating point format.

Several ways,

1. Chopping
2. Rounding.
3. Von-Neumann rounding.

15. Subtract $(11010)_2 - (10000)_2$ using 1's complement and 2's complement method.[May/June-2017]

MULTIPLICATION

16. Discuss the principle behind the Booth's multiplier.

Booth's algorithm generates a 2n-bit product and treats both positive and negative numbers uniformly. This algorithm suggest that we can reduce the number of actions required for multiplication by representing multiplier as a difference between two numbers.

17. Discuss the role of Booth's algorithm in the design of fast multipliers.

To speed-up the multiplication process in the Booth's algorithm a technique called bit-pair recoding is used. It is also called modified Booth's algorithm. It halves the maximum number of summands. In this technique, the Booth-recoded multiplier bits are grouped in pairs. Then each pair is represented by its equivalent single bit multiplier reducing total number of multiplier bits to half.

18. What is combinational multiplier or array multiplier?

The multiplier which uses **n** shifts and adds operations to multiply **n-bit** binary number is called combinational multiplier or array multiplier.

19. How bit pair recoding of multiplier speeds up the multiplication process?

It guarantees that the maximum number of summands that must be added is **n/2** for **n-bit** operands.

20. How CSA speeds up multiplication?

It reduces the time needed to add the summands. Instead of letting the carries ripple along the rows, they can be saved and introduced into the next row, at the correct waited position.

DIVISION

21. Write down the steps for restoring division.

The following are the steps for restoring division

1. Shift **A** and **Q** left one binary position.
2. Subtract divisor from **A** and place answer back in A ($\boldsymbol{A \leftarrow A - B}$)
3. If the sign bit of A is 1, set Q_0 to 0 and add divisor back to A, Otherwise, set Q_0 to 1.
4. Repeat steps 1, 2, and 3 n times.

22. Write down the steps for non-restoring division.

The following are the steps for non-restoring division

1. If the sign of A is 0, shift A and Q left one bit position and subtract divisor from A; otherwise, shift A and Q left and add divisor to A.
2. If the sign of A is 0, set Q_0 to 1; otherwise, set Q_0 to 0.
3. Repeat steps 1 and 2 for n times.
4. If the sign of A is 1, add divisor to A.

23. What is the advantage of non-restoring over restoring division?

Non restoring division avoids the need for restoring the contents of register after successful subtraction.

24. Divide $(1001010)_2$ / $(1000)_2$. [Nov/Dec-2017].

```
                1001          Quotient
Divisor 101 | 1001010        → Dividend
              1000
              ----
                10
                101
                1010
                1000
              ------
                0010          Remainder
              ------
```

FLOATING POINT REPRESENTATION

25. Define IEEE floating point single and double precision standard.

The 32-bit standard representation (single-precision representation) occupies a single 32-bit word. The 32-bits are divided into three fields as shown below:

- (field 1) Sign - 1 - bit
- (field 2) Exponent - 8 - bits
- (field 3) Mantissa - 23 - bits

The 64-bit standard representation (double-precision representation) occupies two 32-bit words. The 64-bits are divided into three fields as shown below:

- (field 1) Sign - 1 - bit
- (field 2) Exponent - 1 - bit
- (field 3) Mantissa - 52 – bits

26. Mention the situations under which a processor sets exception flag.

- Underflow
- Overflow
- Divide by zero
- Inexact Invalid

27. State the representation of double Precision floating Point number.[Nov/Dec-2015]

28. Define underflow and overflow

UNDERFLOW : In a single precision, if the number requires an exponent less than -126 or in a double precision, if the number requires an exponent less than - 1022 to represent its normalized form the underflow occurs.

OVERFLOW: In a single precision, if the number requires an exponent greater than + 127; or in a double precision, if the number requires an exponent greater than + 1023 to represent its normalized form the overflow occurs.

FLOATING POINT OPERATIONS

29. State the rules of floating point multiplication.

1. Add the exponents and subtract bias. (127 in case of single precision numbers and 1023 in case of double precision numbers).
2. Multiply the mantissas and determine the sign of the result. I .
3. Normalize the result.

30. State the rules of floating point division.

1. Subtract exponents and add bias (127 in case of single precision numbers and 1023 in case of double precision numbers).

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

31. What do you mean by guard bits?

The mantissas of initial operands and final results are limited to 24-bits, including the implicit leading 1. But if we provide extra bits in the intermediate steps if calculations we can get maximum accuracy in the final result. These extra bits used in the intermediate calculations are called guard bits.

32. State the commonly used methods of truncation.

There are three commonly used methods of truncation

1. Chopping
2. Von Neumann rounding
3. Rounding

SUBWORD PARALLELISM

33. What do you mean by subword parallelism? [May/June 2015] ?[May/June-2016]

Subword Parallelism is a technique that enables the full use of word-oriented datapaths when dealing with lower-precision data. It is a form of low-cost, small-scale SIMD parallelism.

PART – B

ARITHMETIC LOGIC UNIT

1. Explain in detail how a simple Arithmetic Logic Unit can be constructed.

ADDITION AND SUBTRACTION

2. Explain the design and working of ripple carry adder in detail.

3. Explain in detail the principle of carry-look-ahead adder. *[Refer Page No:2.5]*

[OR]

With a neat diagram explain in detail about the logical design of fast adder. *[Refer Page No:2.5]*

[OR]

Briefly explain Carry Look Ahead adder [Nov/Dec 2014 - 6M] *[Refer Page No:2.5]*

4. Construct a Ripple Carry Adder Circuit and a Carry Look Ahead Adder Circuit to find the sum of two number A=10 and B=5 (Use 8 bit binary).

5. Construct a Ripple Carry Adder Circuit and a Carry Look Ahead Adder Circuit to find the sum of two number A=16 and B=5 (Use 8 bit binary).

MULTIPLICATION

6. **Explain the sequential version of multiplication and its hardware. [May/June 2015 – 16M].** *[Refer Page No:2.9]*
7. **Explain in detail about multiplication algorithm with suitable example and diagram.[Nov/Dec-2015] [May-17]** *[Refer Page No:2.9]*
8. **Define booth multiplication algorithm with suitable example.[May-16]**
9. **Explain Booth's algorithm for the multiplication of signed two's complement numbers. [Dec-16]**
10. **Multiply the following pair of signed numbers using Booth`s bit-pair recording of the multiplier. A=+13(Multiplicand) and B=-6 (Multiplier) [Nov/Dec 2014 - 10M]**
11. **Explain Booth`s Algorithm for multiplication and multiply the following pair using the same A=10, B=4. ?[May/june-2016]**
12. **Explain Booth`s Algorithm for multiplication and multiply the following pair using the same A=16, B=4.**
13. **Explain Booth`s Algorithm for multiplication and multiply the following pair using the same A=20, B=16.**

DIVISION

14. **Explain in detail the process of restoring and non-restoring division.** *[Refer Page No:2.15]*
15. **Explain in detail about division algorithm with suitable example and diagram. [Nov/Dec-2015] [Nov/Dec-2016]** *[Refer Page No:2.15]*

FLOATING POINT OPERATIONS

16. **Write a short note on how floating point numbers are represented in MIPS.** *[Refer Page No:2.17]*
17. **Explain briefly about floating point addition and subtraction algorithm. [May/june-2016]** *[Refer Page No:2.20]*
18. **Explain how floating point addition is carried out in a computer system. Give an example for a binary floating point addition. [May/June 2015 – 16M].**
19. **Explain with an example how Floating Point Multiplication is carried out by a computer system.** *[Refer Page No:2.21]*