Chapter 2

COMPUTER ARITHMETIC

Overview

- Arithmetic and Logic Unit
- Number System
- Complements
- Integer Representation
- Fixed and Floating Point Representation
- Binary Arithmetic with Signed Number
- Serial and Parallel Adder
- Carry-Lookahead Adder
- Multiplication and Division Algorithms
- Booth's Algorithm
- Division Algorithms

Different Data Types

The different data types found in the registers of digital computers are categorized as follows:

(a) Numbers that are used in arithmetic computations.
(b) Letters of the alphabet used in data processing.
(c) Other discrete symbols used for specific purposes.

Number System

A number system, whose base or radix is r, is a system that uses distinct symbols for r digits. The base or radix of the system is the number of different symbols that the system contains.

Example

A binary number system has ***base 2*** (0 and 1 are the two digit symbols used in this system), a decimal number system has ***base 10***, an octal number system has ***base 8***, whereas a hexadecimal number system has ***base 16*** etc.

Alphanumeric Character Set and ASCII

Alphanumeric character set is a set of elements including 10 decimal digits, 26 letters of the alphabet and a number of special characters, like +, -, =, $, * etc. If only the uppercase letters are included then the set contains between 32 and 64 elements (in this case the binary code requires 6 bits) but if both uppercase as well as lowercase letters are included then it contains between 64 and 128 elements (in this case the binary code requires 7 bits).

Example

American Standard Code for Information Interchange (**ASCII**) is the standard alphanumeric binary code that requires 7 bits to code 128 characters.

2.2 Complements

In digital computers, complements are generally used to simplify subtraction operation and also for logical manipulations.

Types of Complements

For each base r system, there are two types of complements, which are the r's complement and the (r-1)'s complement.

1's Complement

The ***1's complement*** is useful when there are some logical operations. This is because the change of 1 to 0 or 0 to 1 is equivalent to any logical complement operation. However the 1's complement is seldom used for arithmetic operations in recent day computers.

2's Complement of a Binary Number

The 2's complement of a binary number can be formed by leaving all least significant 0's and the first 1 unchanged, and then replacing all 0's by 1's and vice versa.

2's complement is useful in case of arithmetic operations as there is no need for overflow corrections.

2.3 Different Ways of Representing an Integer

Positive integers (including 0) can be represented as unsigned numbers but representing negative integers need a notation for negative values. The sign of a number is represented with a bit placed in the leftmost position of the number that is equal to 0 for positive numbers, whereas for negative numbers, it is equal to 1.

For a positive integer binary number, though, there is only a single way of representation where the sign is represented by 0 and the magnitude by a positive binary number, in case of negative numbers (the sign is represented by 1) there are three possible ways of representing a number:

(a) Signed-magnitude representation

Such representation consists of the magnitude along with a negative sign.

(b) Signed-1's complement representation

Here the negative number is presented in 1's complement of its actual positive value.

(c) Signed 2's complement representation

Here the negative number is represented in 2's complement of its actual positive value.

2.4 Fixed-Point and Floating-Point Representations

In a number there may be a binary (or decimal) point, the position of which is needed to represent fractions, integers, or mixed integer-fraction numbers. In a register, there are two ways of specifying the position of (i.e. representation of) the binary point: either by giving it a fixed position or by employing a floating-point representation. In general, it can be stated that there is no method that is capable of representing all real numbers using finite register lengths and so approximations should be used to represent values. Apart from the two forms, fixed-point and floating-point representations stated above, other forms are: Rational Number Systems using ratios of integers and Logarithmic Number Systems using signs and logarithms of values.

Fixed-Point Method

This assumes that the binary point is always fixed in one specific point in the register i.e. either the binary point lies in the ***extreme left*** of the register thus making the stored

number a ***fraction***, or it lies in the *extreme* ***right*** of the register making the stored number an ***integer.***

Though in either of the above cases, the binary point is not present physically but its presence is assumed to treat a stored number as fraction or as integer.

Floating-Point Method

Such a representation uses a second register to store the number that designates the position of the decimal point stored in the first register.

Example of floating-point numbers

3.14159, 0.000000000567 etc.

Explanation of Floating-Point Representation of Numbers

Consider the number 703000000000000000000000. Eliminating the zeros it can be noted that this number is just

$7.03 \times 10000000000000000000000 = 7.03 \times 10^{23}$

This is known as ***scientific notation.***

There are two parts in the floating-point representation of a number. They are:

Mantissa: This is the first part. This part is represented by a signed, fixed-point number. Mantissa may be a fraction or it may be an integer.

Exponent: This is the second part, which indicates the position of the decimal point in the number i.e. where the decimal point is located.

Example

To represent the number +7154.232. The representation of this number in floating-point is as follows:

+0.7154232 (Fraction part or the mantissa part → m)

+04 (exponent part → e)

The exponent part indicates the actual position of the decimal. Here +04 indicates that the actual position of the decimal point in the number is four positions to the right of the decimal point as indicated in the fraction.

So this number can be represented as $m \times r^e$ $(+0.7154232 \times 10^{+4})$, where r is the radix. The m and e values along with their sign-bits are physically represented in registers.

Floating-point binary numbers can also be represented similarly.

Example

Binary number +1101.01 can be represented in floating-point as follows:

01101010 (8-bit fractional part or mantissa part)

000100 (6-bit exponent part)

Decimal point in the number is four places to the right of the indicated decimal point in the fraction. Hence, in the exponent part it is indicated by binary 4. As the fraction is positive, hence it is denoted by the leftmost 0-bit.

Normalization of Floating-Point Numbers

If the most-significant bit of the mantissa in a floating-point number is nonzero, the number is said to be ***normalized***. So 1500 is normalized but 0150 is not. This is also true for binary numbers. Like, 00010010 is not normalized. To normalize it, just left shifting of three positions along with the discarding of the leading zeros is needed. This will result in10010000, which is a normalized representation. While adding two normalized mantissas, it may result to an ***overflow*** condition (known as significant or mantissa overflow), which may be corrected by shifting the sum to the right and then by incrementing the exponent. A denormalized floating-point number, or the floating-point number that has a 0 bit in the most significant position is said to have an ***underflow*** (significant underflow). To correct this condition, it is needed to shift the mantissa to the left decrementing the exponent value until a nonzero digit appears in the most significant bit position. Apart from mantissa overflow and underflow conditions, exponent overflow and underflow conditions may also occur, which can be similarly solved by rounding of the values (i.e. by shifting mechanism).

IEEE Representation of Floating-Point Numbers

All modern computers use the floating-point representation that was specified in IEEE standard 754. Here as was discussed before numbers are represented by a mantissa and an exponent.

Out of the multiple number of bit widths specified by IEEE standard 754 for floating-point numbers, single-precision and double-precision widths are the most commonly used widths. Figure 3 shows the two formats. Single-precision numbers are 32-bits long with 8-bits of exponent, 23-bits of fraction and 1 sign-bit, whereas double-precision numbers are 64-bits long with 11-bits of exponent, 52-bits of fraction and 1 sign-bit.

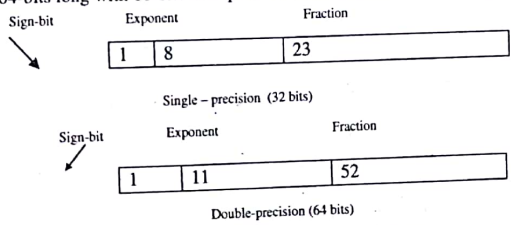


Fig: 3 IEEE 754 Floating-point formats

2.5 Addition and Subtraction of Signed-Magnitude Numbers

Addition - Subtraction Algorithm:

The addition - subtraction algorithm for addition and subtraction of signed-magnitude numbers states that:

(a) If the signs of the two numbers are identical (different), the two magnitudes are to be added and the result will have the sign of the 1st number.

(b) If the signs of the two numbers are different (identical), then the magnitudes of the numbers are compared and the smaller number is subtracted from the larger.

(i) The sign of the result will be that of the 1st number provided that the 1st number is larger than the 2nd one.

(ii) The sign of the result will be the complement of the 1st number provided that the 1st number is smaller than the 2nd one.

(iii) If the two magnitudes are equal, then the 2nd number is subtracted from the 1st one and the sign of the result is made positive.

Hardware implementation for signed-2's complement addition and subtraction

The addition of two numbers in signed-2's complement form consists of adding the numbers with the sign bits treated just like the other bits in the number. However, any carry-out of the sign-bit position is discarded. Similarly, in case of subtraction, the 2's complement of the subtrahend is added to the minuend.

Figure 4, shows the register configuration for the hardware implementation.

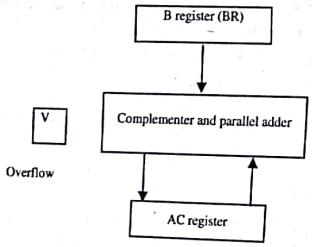


Fig: 4 Hardware for addition and subtraction with signed 2's complement data

As shown in the figure 4, the two numbers to be added or subtracted are stored in the B register (storing the addend) and in the AC (Accumulator) register (storing the augend). The leftmost bits in B and AC represent the sign bits of the numbers. The entire addition and subtraction of the two numbers (along with their sign-bits) are performed in the

parallel adder and complementer block. If there is an overflow then the overflow flip-flop V will be set to 1 and in that case the output carry will be discarded.

Steps for adding and subtracting numbers in signed 2's complement representation with the help of flowcharts

Steps for addition

Step 1

The augend is put in AC and the addend is put in BR.

Step 2

If any of the number is negative, then its 2's complement representation must be considered.

Step 3

The sum is obtained by adding the contents of AC and BR (including their sign-bits) in the parallel adder and complementer block.

Step 4

The result is stored in AC. If the XOR of the last two carries is 1, then there is a carryout and the overflow bit V is set to 1 (else it is cleared to 0).

Step 5

If there is an overflow then the carryout bit must be discarded. If the result is negative then its 2's complement must be considered to get the actual positive number and then its negative integer representation is considered as the ultimate result.

Steps for subtraction

Step 1

The minuend is put in AC and the subtrahend is put in BR.

Step 2

If any of the number is negative, then its 2's complement representation must be considered. Also the subtraction operation is accomplished by adding the content of AC to the 2's complement of B. This is because, taking the 2's complement of B, has the effect of changing a negative number to positive and a positive number to negative.

Step 3

The result is obtained by performing the computation in the parallel adder and complementer block.

Step 4

The result is stored in AC. An overflow is checked similarly.

Step 5

The carryout bit must be discarded provided there is an overflow.

Flowcharts

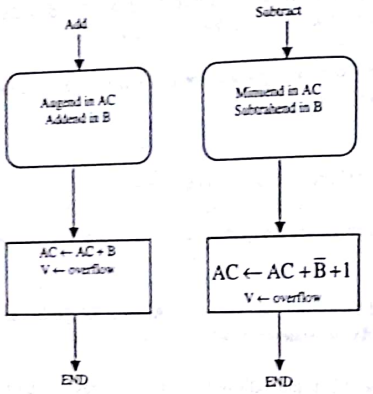


Fig: 5 For Addition Fig: 6 For Subtraction

2.6 Serial and Parallel Adders

A ***serial adder*** is a binary adder that adds the two numbers bit-pair wise. Each bit-pairs are added in a single clock pulse. The carry of each pair is propagated to the next pair.

A ***parallel adder*** is a binary adder that generates the arithmetic sum of two binary numbers of any lengths using multiple cascaded full-adder circuits. Here the output carry from one full-adder is connected to the input carry of the next high order full-adder. Here the entire addition is done in a single pulse.

Though parallel adders are very fast than serial adders but they need a very complex circuitry.

Working Principle of a Parallel Adder

Parallel adder is the digital circuit that generates the arithmetic sum of two binary numbers of any lengths. Multiple cascaded full-adder circuits constitute a parallel adder. The output carry from one full-adder is connected to the input carry of the next high order full-adder and so on. So an n-bit parallel (binary) adder requires n full-adders. The augend bits of A (say, n data bits) come from one register (say, Register R1) while the addend bits of B (n data bits) come from another register (say, register R2). Subscript numbers from right to left designates the augend bits and the addend bits. Subscript 0 denotes the lower-order bit while the subscript n denotes the higher-order bit. The carries are connected in a chain through the full-adders. C_0 is the input carry to the multiple

cascaded binary adders and C_4 is the output carry. The required sum bits are generated by the S outputs of the full-adders. The sum can be stored either to a third register (say, Register R3) or may be in any one of the source registers (replacing its previous content). Figure 7, shows the interconnection of four full-adders to provide a 4-bit binary adder.

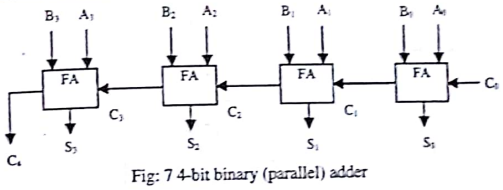


Fig: 7 4-bit binary (parallel) adder

Here suppose that '10' i.e. 1010 is to be added to '5' i.e. 0101. So the augend bits of A_0 through A_3 will be 0, 1, 0 and 1 (i.e. A_0 will input 0, A_1 will input 1, A_2 will input 0 and A_3 will input 1). Similarly, for B also (the addend bits). Adding the two numbers will give the output as 1111 (i.e. 15). So the output line S_0 will output 1, S_1 will output 1, S_2 will output 1 and S_3 will output 1. In the example there are no carry-bits.

2.7 Carry –Lookahead Adder

This is a very fast adder circuit which speeds up the generation of carry signals by using the same (almost) logic as that of parallel adder.

Explanation of Carry-Lookahead Addition

This circuit basically speeds up the generation of carry signals. The logic expressions for sum (s_i) and carry-out (c_i) of stage i are:

$s_i = A_i \oplus B_i \oplus c_i$ and $c_i + 1 = A_iB_i + A_ic_i + B_ic_i$

$= A_iB_i + (A_i + B_i)c_i = G_i + P_ic_i$

where $G_i = A_iB_i$ and $P_i = A_i + B_i$.

The expressions G_i and P_i are called the generate and propagate functions for stage i respectively. So if the generate function for stage i is equal to 1 i.e. if $G_i = 1$, then $c_i + 1 = 1$ (when both A_i and B_i are equal to 1).

The propagate function (P) means that an input carry will produce an output carry when either of A or B is 1. So all G and P functions can be formed independently and in parallel in only one logic gate delay.

Now, if the propagate function can be realized as $P_i = A_i B_i$, then a simple circuit can be derived using a cascade of two 2-input XOR gates (to realize 3-input XOR function). The figure 8, shows the basic cell that can be used in each bit stage.

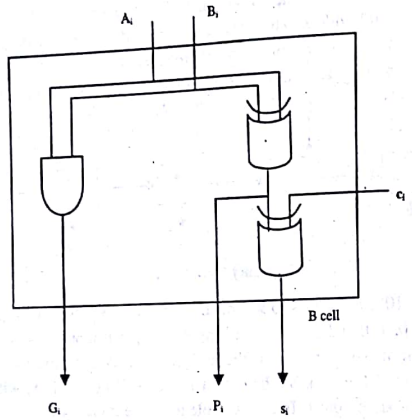


Fig: 8 Bit-stage cell

Hence, expanding c_i in terms of i – 1 subscripted variables and then substituting the value into the $c_i + 1$ expression, the following expression is obtained:

$$c_i + 1 = G_i + P_iG_{i-1} + P_iP_{i-1}c_{i-1}$$

So, expansion of this type of expression will give the final expression for any carry variable as:

$$c_i + 1 = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots + P_iP_{i-1}\ldots P_1G_0 + P_iP_{i-1}\ldots P_0c_0$$

This means that all carries can be obtained three gate delays after the input signals A, B, and c are applied as only one gate delay is required to develop all G_i and P_i signals, and then followed by two gate delays in the AND-OR circuit for $c_i + 1$. After a further XOR gate delay, ultimately, all sum bits are available. Hence, irrespective of n, the n-bit addition process thus requires overall four gate delays.

Considering the design of a 4-bit adder, the carries can be implemented as:

$$c_1 = G_0 + P_0c_0$$
$$c_2 = G_1 + P_1G_0 + P_1P_0c_0$$
$$c_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$$
$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

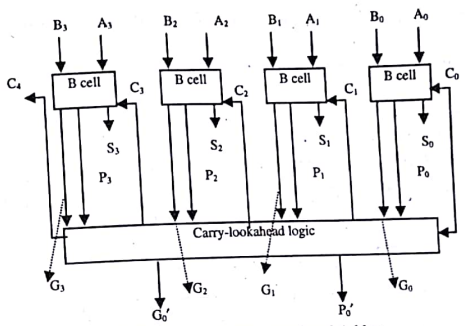


FIG: 9 4-Bit Carry-Lookahead Adder

Figure 9, shows the block diagram of the 4-bit adder, where the carries are generated in the block labeled carry-look ahead logic. Such types of adders are called ***carry-lookahead adder***. So, for all carry bits the total delay is 3-gate delays and for all sum bits the total delay is 4-gate delays.

2.8 Multiplication of Signed-Magnitude Data

Hardware implementation for multiplication of signed-magnitude data

Two fixed-point binary numbers represented in signed-magnitude can be multiplied by successive shift and add operations.

To multiply signed-magnitude data, register A (A holds the partial product values after each bit of the multiplier is multiplied to the multiplicand) is initially set to 0, the ***multiplicand***, initially, is placed in register ***B***, ***multiplier***, initially, is placed in register ***Q***, A_s and B_s are the two flip-flops that hold the sign-bits of the values in A and B registers. Q_s register holds the sign-bit of the multiplier. Flip-flop ***E*** holds the output carry. Sequence counter ***SC*** keeps track of the number of bits in the multiplier and counts down by 1 (i.e. decremented by 1) after each bit in the multiplier is multiplied to the multiplicand bits. Initially SC content is set to the number of bits in the multiplier. As soon as the content of SC counts down to zero (i.e. all the multiplier bits are multiplied with the multiplicand bits), the entire product is then obtained and the process stops.

Sum of A and B forms a partial product that is transferred to the EA register and both the partial product as well as the multiplier are shifter to the right as denoted by 'shr EAQ' which means that EA and Q registers are shifted to the right. Now, the following shifts occurs:

(a) The least significant bit of A register is shifted to the most significant position of Q.

(b) The output carry bit from E is shifted to the most significant position of A register.

(c) 0 is shifted to E.

After the completion of the shifts, one partial product bit is shifted into Q. This push the multiplier bits one position to the right. Thus the Q_n flip-flop, which is the rightmost flip-flop in register Q, holds the multiplier bit to be inspected next.

Figure 10, shows the hardware implementation for the above discussed multiply operation.

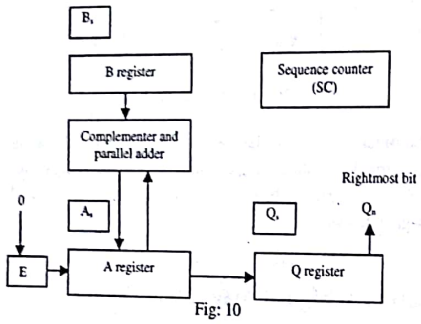


Fig: 10

Explanation of the hardware multiply algorithm with a flowchart

The hardware multiply operation for signed magnitude data proceeds as follows:

Steps

(a) The multiplicand and the multiplier are, initially, stored in register B and in register Q.

(b) Contents of registers A and E are cleared (A = 0 and E = 0).

(c) Sequence counter SC, initially, contains the number of multiplier bits i.e. if the multiplier is, say, 1011, so SC = 4, initially (Here the sign-bit of the number, stored in register Q as multiplier, is not considered. If the sign-bit is considered then SC = n – 1 i.e. = 3, where n is the total number of multiplier bits). SC content gets decremented by 1 as soon as one bit in the multiplier is multiplied to the multiplicand.

(d) Comparing the sign-bits, both A and Q are set to correspond sign of the product i.e. they are set as per the product sign.

(e) The low-order bit of the multiplier i.e. the bit in Q_n is tested to see if it is 1 or 0.

(f) If it is 1, then just the multiplicand in B is added to the present partial product value in A. This is because anything multiplied by 1 is that number itself and so

multiplying by 1 gives back the multiplicand itself which is just added to the existing partial product value

(g) If the low-order bit of Q = 0, then nothing is to be done as multiplying by 0, gives 0 itself.

(h) After each multiplier bit is multiplied, register EAQ is just shifted once to the right and this forms the new partial product.

(i) On each step, the SC count is checked to see whether 0 or not. SC = 0, means all the bits in the multiplier is multiplied and final product is obtained. This marks the end of the process.

If SC = 1, it means that all the multiplier bits are not multiplied yet and the process needs to be repeated till SC = 0.

(j) The final product is available both in A and Q registers. Register A holds the most significant bits while Q holds the least significant bits.

Flowchart

The flowchart below shows the steps of the multiplication procedure.

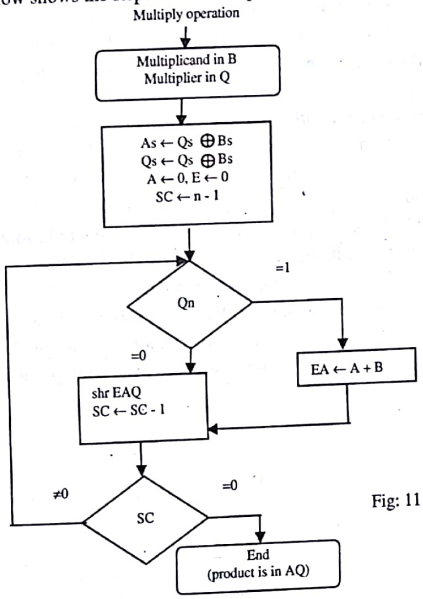


Fig: 11

2.9 Booth's Multiplication Algorithm

Aim of Booth's multiplication algorithm

Booth's algorithm provides a procedure by which binary integers in signed-2's complement representation (i.e. multipliers can be positive or negative) can be multiplied.

Concept of the algorithm

The basic concept of this algorithm is that, as multiplication of multiplicand by strings of zeros in the multiplier, requires no addition but just shifting and multiplication by strings of 1s require addition, hence in this algorithm string of 1's in the multiplier from bit weight 2^k to weight 2^m are treated as $2^{k+1} - 2^m$. Say, the binary number 011110 (+14) has a string of 1's from 2^4 to 2^1 (where k = 4, m = 1) and hence the number can be represented as $2^{k+1} - 2^m = 2^5 - 2^1 = 32 - 2 = 30$. So, if we are multiplying MD by 14 i.e. MD × 14 (MD is the multiplicand and 14 is the multiplier), then it can be done as MD x 24 – MD x 21. Hence, by shifting the binary multiplicand four times to the left and then subtracting MD shifted once to left, the result of multiplication can be obtained.

Hardware configuration for Booth's multiplication algorithm

The figure 12, shows the hardware implementation for Booth's algorithm.

Register BR: It holds the multiplicand along with its sign-bit.
Register AC: It holds the partial product after each stage of multiplication.
Register QR: It holds the multiplier along with its sign-bit.
Q_n: It designates the least significant bit of the multiplier.
Q_{n+1}: This is a flip-flop with the purpose of double-bit inspection of the multiplier.
Sequence Counter (SC): Keeps track of the number of bits in the multiplier and decrements by 1 after each multiplier bit is multiplied to the multiplicand.

Diagram

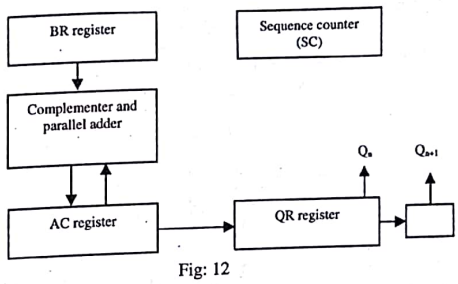


Fig: 12

Explanation of Booth's algorithm with the help of a flowchart

Steps

(a) Initially there is no partial product and hence AC is cleared to 0.
(b) Q_{n+1} is initially cleared to 0.
(c) SC initially holds the number of bits 'n' in the multiplier.
(d) If the two bits Q_n and $Q_{n+1} = 10 \Rightarrow$ first 1 in a string of 1's has been encountered and hence the multiplicand is to be subtracted from the partial product value in AC register.
(e) If the two bits Q_n and $Q_{n+1} = 01 \Rightarrow$ first 0 in a string of 0's has been encountered and hence the multiplicand is to be added to the partial product value in AC register.
(f) If the two bits Q_n and $Q_{n+1} = 00$ or 11 (i.e. they are equal) $\Rightarrow$ the partial product value remains unchanged. In this technique, overflow cannot occur as the two numbers that are added always have opposite signs.
(g) The partial product (i.e. AC) and the multiplier (i.e. QR) are shifted right. However, the sign-bit in AC remains unchanged.
(h) If SC = 0, the process stops else it continues.

Flowchart

The flowchart for Booth's algorithm is shown below.

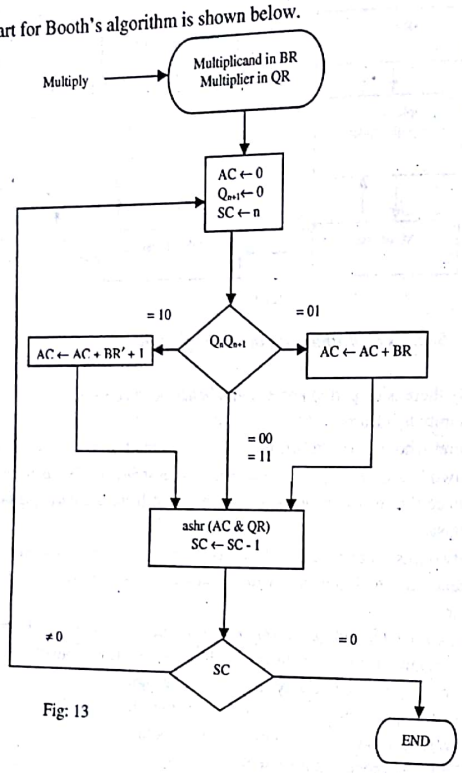


Fig: 13

2.10 Division Algorithms

Division of two fixed-point binary numbers represented in signed-magnitude form is done by successive compare, shift and subtract technique. However in division, it may give rise to an overflow result i.e. if the expected quotient is of n-bits but the actual quotient comes as n+1 bits then that condition is an overflow condition, which must be taken care of.

Hardware Implementation of Division Algorithms

The hardware implementation is as shown below:

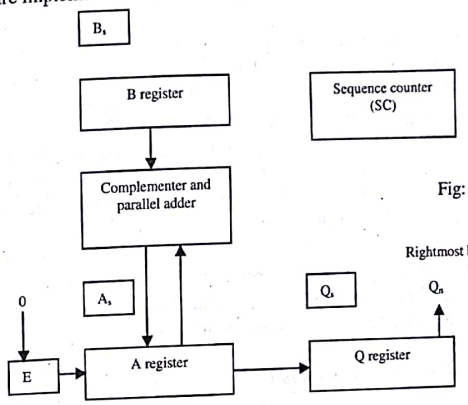


Fig: 14

(The hardware implementation is identical to that explained under the heading 'hardware implementation for multiplication of signed magnitude data'). In division techniques, register EAQ is shifted to the left, 0 is inserted into Q_n, previous E value is lost.

Partial Remainder

After every compare, shift and subtract step in the division process ***partial remainder*** is obtained, which is actually the difference between the dividend and the divisor. This is so called as after the first partial remainder is found out, the division process could have stopped here thus obtaining the 1st 1-bit of the quotient and the remainder equal to the partial remainder.

Division Overflow Condition

Such condition occur when a division results in a quotient with an overflow i.e. if the result is expected to consist of n number of bits, but the actual result consists of n+1 number of bits, then an overflow condition has occurred.

Need of handling Division Overflow Condition

This condition is troublesome when performing division operation in computer i.e. when division operation is implemented with hardware. This is because after the division operation, the result is to be stored in some register with a finite or fixed length (storage capacity). So if the number of bits to be stored actually, exceeds to that expected then it cannot be stored in the register provided. And if the result is stored in a memory register

of fixed length then it is a problem to add any extra flip-flop for the overflow bit. Hence overflow conditions need to be handled always.

Different Techniques to Handle Division Overflow Condition

The different techniques taken to handle overflow condition are as follows:

(a) Programmer must check whether the DVF flip-flop (this is a special flip-flop that detects the overflow condition and gets enabled if there is an overflow) is set after each division operation and if set then control branches to a subroutine that takes corrective measures (like rescaling the data) to handle the overflow condition.

(b) Operation of a computer may be stopped if there is an overflow. This condition is called divide stop. After the overflow is handled, operation resumes.

(c) To provide an interrupt request if the DVF is set. This will suspend the currently running program and will give an error message to the user to correct the program. Once corrected again the program will run accordingly.

(d) Overflows may be avoided by using floating-point data.

Explanation for division of two binary numbers

Flowchart

The flowchart (figure 15) for division of two unsigned binary numbers are as shown below:

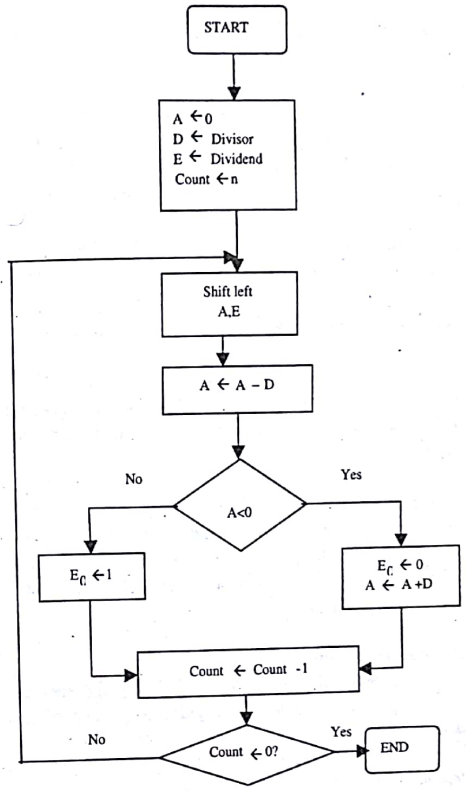


Fig: 15

Explanation of the above flowchart

The division mechanism involves repetitive shifting followed by addition or subtraction. The steps are as follows:

Step 1

Register D contains the divisor while registers A and E contain the dividend. The dividend in register E is expressed as a 2n-bit two's complement number.

Step 2

Contents of registers A, E are shifted by 1-bit position to the left.

Step 3

Content of D (i.e. the divisor) is subtracted from that of A and the result is stored in A itself (i.e. A ←A-D) provided that D and A have the same signs else the contents are added and stored in A (i.e. A ←A+D).

Step 4

The above operation is only said to be successful provided the sign of A remains the same both before and after the operation.

(i) If the operation is successful or (A=0 AND Q=0), hence E_0 can be set to 1 (i.e. E_0 ← 1).

(ii) If the operation is unsuccessful and (A≠0 OR E≠0), hence E_0 can be set to 0 (i.e. E_0 ← 0) and the previous value of A can be restored.

Step 5

Depending on as many numbers of times as there are bit positions in E, steps 2 to 4 may be repeated.

Step 6

Register A stores the remainder. Register E stores the quotient provided that the signs of the divisor and the dividend are the same else two's complement of the content of register E is the correct quotient.

Restoring Division Technique

Restoring division technique is the hardware method of performing division operations. Here after each division step, the partial remainder obtained, restored by adding the divisor to the negative difference. This is done to get back the original AC value or to restore the value after every division step.

Nonrestoring Division Technique

In ***nonrestoring division technique***, if the difference is negative then the divisor is not added directly to the partial remainder. It is added only after shifting the negative difference to the left.

Comparison Method of Division

In ***comparison method*** of division, before the subtraction operation is done, A and B values are compared. If A >=B then B value is subtracted from A value, else if A < B then nothing needed to be done. Shifting the partial remainder to the left, the numbers are compared again. Before the subtraction, however, the values can be compared by seeing the parallel-adder's end carry bit.

Related Questions & Answers

Question 1

Represent the signed number 12 stored in an 8- bit register in different integer representations.

Answer:

The signed number 12 is stored in an 8-bit register. So it is represented in the following ways:

(A) Positive number representation: Here '+12' is represented as 00001100, where the sign bit 0 is in the leftmost position of the representation followed by the binary equivalent of 12.

(B) Negative number representation: Here there are three different ways of representation, which are as follows:

(a) Signed-magnitude representation of '-12': Here only the sign-bit is changed and rest remains the same as in (A) i.e. the result will be 10001100.

(b) Signed-1's complement representation of '-12': Here along with the change in sign-bit, the remaining binary digits also get complemented i.e. the result will be 11110011.

(c) Signed-2's complement representation of '-12': Here the sign-bit is changed as well as the remaining bits are represented in their 2's complement form i.e. the result is 11110100.

Question 2

Explain the relative advantages & disadvantages of parallel adder over serial adder. **[WBUT 2003, 2006]**

Answer: Refer to section 2.6.

Question 3

Between 1's and 2's complement which one is more advantageous and why? What are the disadvantages of 2's complement in comparison to 1's complement?

Answer:

2's complement is more advantageous. This is because:

(a) In 2's complement while addition, the end carry can be ignored (which is not the case with 1's complement).

(b) In 2's complement, 0 has got only one representation (in 1's complement there are two representations of 0 i.e. '+0' and '-0').

Disadvantages

From a signed magnitude representation, it is much easier to compute 1's complement than to compute 2's complement.

Question 4

Give the flowchart for division of two binary numbers and explain. **[WBUT 2004]**

Answer:

The flowchart (figure 16) for division of two unsigned binary numbers is shown below:

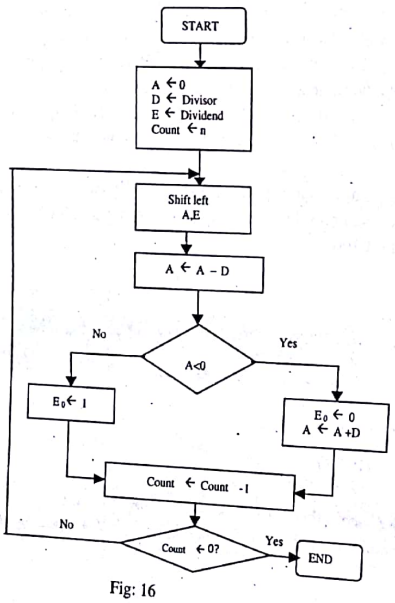


Fig: 16

Explanation of the above flowchart

The division mechanism involves repetitive shifting followed by addition or subtraction. The steps are as follows:

Step 1

Register D contains the divisor while registers A and E contain the dividend. The dividend in register E is expressed as a 2n-bit two's complement number.

Step 2

Contents of registers A, E are shifted by 1-bit position to the left.

Step 3

Content of D (i.e. the divisor) is subtracted from that of A and the result is stored in A itself (i.e. A ← A-D) provided that D and A have the same signs else the contents are added and stored in A (i.e. A ← A+D).

Step 4

The above operation is only said to be successful provided the sign of A remains the same both before and after the operation.

(i) If the operation is successful or (A=0 AND Q=0), hence E_0 can be set to 1 (i.e. $E_0 \leftarrow 1$).

(ii) If the operation is unsuccessful and (A≠0 OR E≠0), hence E_0 can be set to 0 (i.e. $E_0 \leftarrow 0$) and the previous value of A can be restored.

Step 5

Depending on as many numbers of times as there are bit positions in E, steps 2 to 4 may be repeated.

Step 6

Register A stores the remainder. Register E stores the quotient provided that the signs of the divisor and the dividend are the same else two's complement of the content of register E is the correct quotient.

Question 5

Explain how the real numbers are represented in Computer memory. Briefly explain the standard formats. **[WBUT 2003]**

Briefly explain the IEEE 754 standard format for floating point representation. **[WBUT 2003, 2006]**

Answer: Refer to section 2.4.

Question 6

Explain restoring division technique with a diagram.

Answer:

Consider the diagram (figure 17) below. Register D stores the n-bit positive divisor and register E loads the n-bit positive dividend. The content of register A initially is set to 0. In the first step, the contents of registers A and E are shifted left by one binary position. Then A – D is computed and the result is again placed in A i.e. A ← A – D. Provided the sign bit of register A is 1, the least significant bit (LSB) of the dividend in E (i.e. E_0) is set to 0 and contents of D is added back to that of A (i.e. $E_0 \leftarrow 0$, and A ← A + D) which means the value of A is again restored else $E_0 \leftarrow 1$ (if sign of A is 0). All these steps are performed n number of times.

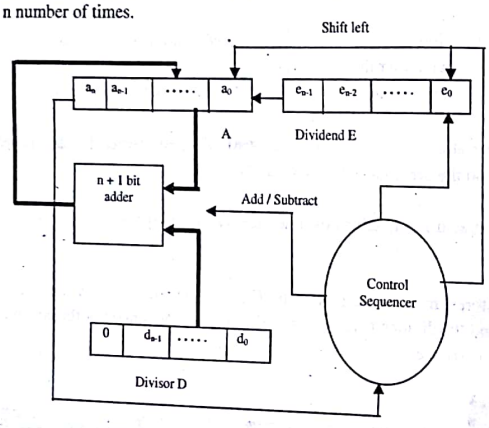


Fig: 17 Circuits for restoring division technique

Question 7

Explain non-restoring division technique.

Answer:

Suppose while performing division by restoring division technique, subtraction of the divisor content in D from that of A leads to a negative result (i.e. an unsuccessful subtraction), still the value of A is to be restored. This is however time consuming and can be seen as an unnecessary overhead. This drawback of the restoring division

technique can b avoided in the non-restoring division technique that works in the following manner:

After the subtraction is done in the restoring division technique, provided that sign of A positive (i.e. if A ← 0), then both the contents in A and E are shifted one bit to the left and D is subtracted from A (i.e. 2A – D) else if A is negative (i.e. sign of A ← 1) again both the contents of A and E are shifted left by 1 bit but D is added to A (i.e. effectively 2A + D is performed). Based on the result of the correct operation the E_0 bit is set to either 0 or 1.

Question 8

Compare Restoring & Non-Restoring Division algorithms.

[WBUT 2003, 2005, 2006, 2007]

Answer: Refer to question 6 and 7.

Question 9

What is truncation?

Answer:

In floating-point operations (addition, subtraction, division or multiplication), the length of the mantissas of the initial operands as well as of final results must be kept limited to 24-bits (including the extra guard bits). So in order to obtain the actual final result by removing the guard bits, the mantissa (which is extended owing to the presence of guard bits) length should be ***truncated*** (i.e. reduced) to create a 24-bit number that is approximately equivalent to the non-truncated version of mantissa (i.e. mantissa with guard bits).

So (mantissa + guard bits) – guard bits =

Non-truncated or extended mantissa of 24 bits

truncated or actual result in mantissa (approx. equivalent to extended version of mantissa).

Chopping technique (biased technique in nature) and rounding procedure (an unbiased technique) can do truncation.

Question 10

What are biased exponents?

Answer:

Biased Exponent:

Exponents are commonly stored in different IEEE standard formats as unsigned integers. However, an exponent can be negative as well as positive, and so there must be some technique for representing negative exponents using unsigned integers. This technique is called "biasing". A positive number is added to the exponent before it is stored in to the floating point number. The stored exponent is then called a "***biased exponent***". If the exponent contains 8 bits, the bias number 127 is added to the exponent before it is stored so that, for example, an exponent of 1 is stored as 128. Since the unsigned exponent can represent numbers between 0 and 255, it should be possible to store exponents whose values range from -127 to +128 i.e. -127 would be stored as the biased exponent value 0, and +128 would be stored as the biased value 255.

Question 11

Give the merits and demerits of the floating point and fixed point representations for storing real numbers. **[WBUT 2004]**

Answer: Refer to section 2.4.

Question 12

What are guard bits? **[WBUT 2004]**

OR,

What is the necessity of Guard bits? **[WBUT 2008, 2009]**

Answer:

Guard bits are additional or extra bits present in the ALU registers that load the exponent and significant of each operand prior to a floating point operation. Guard bits are basically used to pad out (i.e. to add or place) the right end of the significant with extra 0's to keep the length of the numbers fixed.

Question 13

In the context of floating-point arithmetic, explain precision. What are the different modes of precision?

Answer:

Floating-point representations have a mantissa or fraction part and an exponent or characteristic part. In this context, the number of digits represent able in the fraction or mantissa part indicates the precision of a floating-point number system, which is generally measured in digits of radix r. For example, if the fraction part of a certain floating-point number is n bits long, then the number has a n bit precision. In other words, the number is represented with n bits of precision.

In a floating-point number system, two modes of precision may occur: single and double. The different precision modes are generally used to overcome overflow or underflow related problems in floating-point systems. While single precision indicates the different floating-point operations defined with standard operands, like a floating-point value represented with a single 32 bit word, in case of double precisions, the word-format is simply double the length having 64 bits (i.e. two 32-bit words) with a larger exponent.

Refer to figure 3 in section 2.4.

Question 14

Write short notes on 'Carry save.Adder'. **[WBUT 2004]**

Answer: Refer to section 2.7.

Question 15

Consider a large floating-point representation using 16-bits for the exponent. If a bias is used to represent the exponent, what would the bias is expected to be?

Answer:

Just as the exponent bias on the IEEE floating point number with 8 exponent bits is 28/2 - 1 = 127, the exponent bias on a number with 16 exponent bits is expected to be 216/2 - 1 = 32767. This divides the range of possible exponent values almost equally between negative and positive exponents. The "- 1" in "216/2 - 1" in arises from the fact that you need to reserve one exponent value, namely 0x0000, to be used in the special value representing the number 0.

Question 16

Differentiate between real and floating-point numbers.

Answer:

Parameters	Real	Floating-Point
Number of values	Infinite	Finite
Range of numbers	- Infinity ..... + Infinity	Finite
Spacing	Can be both constant and infinite	The inter-number gap varies
Errors	????????????????	Possibilities of incorrect results are there.

Question 17

Design a 4-bit Arithmetic unit using multiplexers and full-adders.

Answer:

The arithmetic micro operations like addition, subtraction, increment and decrement can be implemented in a composite arithmetic circuit. The diagram below shows the four-bit composite arithmetic circuit.

The circuit constitutes of four 4 x 1 multiplexers and four 4-bit full-adder circuits. C_{in} indicates the carry-in, C_{out} indicates the carry-out, S_1 and S_0 are the selection lines controlling the multiplexers, A_i and B_i are the 4-bit inputs whereas D_i is the 4-bit output. The C_i's of each full adder receives the carry-in input whereas the C_{out} or carry-out propagates from full-adder 1 to full-adder 4. The A_i inputs go to the X_i inputs of the adders. The B_i inputs and the complements of B_i inputs go to the multiplexers. The remaining two data inputs of the multiplexers, i.e. 2 and 3, are connected to the logic-0 and logic-1 respectively. So, the output of the binary adders i.e., the D_is can b calculated as: $D_i = X_i + Y_i + C_{in} = A_i + Y_i + C_{in}$.

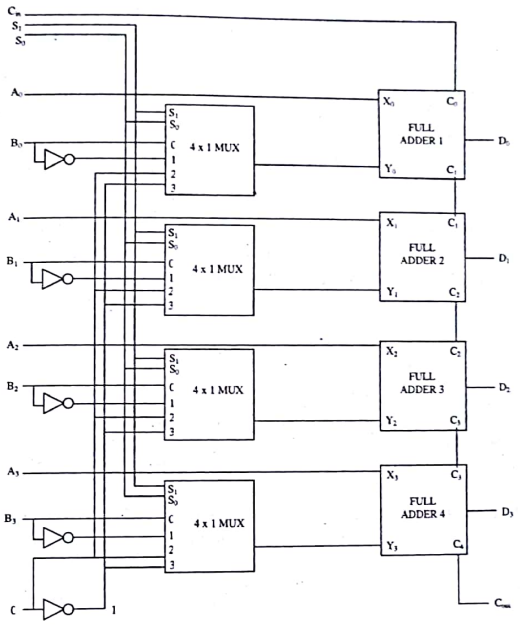


Fig: Four – Bit Arithmetic Circuit

Now the following eight microoperations can be generated by making C_{in} either 0 or 1 and by controlling the value of Y with the help of the selection lines.

S_1	S_0	C_{in}	Y_i	$D = A_i + Y_i + C_{in}$	Microoperation
0	0	0	B_i	$D = A_i + B_i$	Add
0	0	1	B_i	$D = A_i + B_i + 1$	Add with carry
0	1	0	B_i'	$D = A_i + B_i'$	Subtract with borrow
0	1	1	B_i'	$D = A_i + B_i' + 1$	Subtract
1	0	0	0	$D = A_i$	Transfer A
1	0	1	0	$D = A_i + 1$	Increment A
1	1	0	1	$D = A_i - 1$	Decrement A
1	1	1	1	$D = A_i$	Transfer A

(a) Addition: The addition microoperation is performed with or without carry depending on whether C_{in} = 0 or 1 with S_1 and S_0 remaining 0 in both the cases.

(i) Addition without carry: In this case with C_{in} = 0, the value of B_i is applied to the input Y_i, which gives output

$D = A_i + B_i + 0 = A_i + B_i$.

(ii) Addition with carry: In this case with C_{in} = 1, the value of B_i is applied to the input Y_i, which gives output

$D = A_i + B_i + 1$.

(b) Subtraction: The subtraction microoperation is performed with or without borrow depending on whether C_{in} = 0 or 1 with S_1 = 0 and S_0 = 1 in both the cases.

(i) Subtraction with borrow: Here with C_{in} = 0, the complemented value of B_i is applied to the input Y_i giving output $D = A_i + B_i'$.

(ii) Subtraction without borrow: Here with C_{in} = 1, the complemented value of B_i is applied to the input Y_i giving output $D = A_i + B_i' + 1$.

(c) Transfer: The value of A_i is directly transferred to the output D with S_1 = 1, S_0 = 0, C_{in} = 0 and value at Y_i also 0.

(d) Increment: The value of A_i is incremented by 1 i.e. unlike the previous microoperation where $D = A_i$, here $D = A_i + 1$. In this case both S_1 and C_{in} = 1 and S_0 = 0. Value at Y_i = 0.

(e) Decrement: The value of A is decremented by 1 giving the output $D = A_i - 1$ when both S_1 and S_0 = 0 and C_{in} = 0. The input at Y_i in this case is equal to 1.

(f) Transfer: Here also the value of A_i is directly transferred to the output D but unlike the transfer microoperation discussed in (c), in this case, the values of S_1, S_0, C_{in} and that at Y_i are equal to 1.

Question 18

Give the non-restoring division algorithm for division of two binary numbers. **[WBUT 2005]**

Answer: Refer to section 2.10 and question number 5.

Question 19

Write short notes on 'Parallel adder'. **[WBUT 2005]**

Answer: Refer to section 2.6.

Question 20

In floating-point number representation system, if 24 bits are reserved for mantissa and 8 bits are reserved for signed exponent, determine the values of maximum & minimum positive & negative numbers in this scheme.

Answer:

In the 32-bit floating-point number system, negative numbers are represented in 2's complement form, where the leading bit is the sign bit. As stated, 24 bits are reserved for mantissa and 8 bits are reserved for signed exponent.

Thus, the maximum positive number is $+[2^{23} - 1] * (2^{127})$ and the minimum positive number is $+(1) * (2^{-128})$.

The maximum negative number is $-[2^{23} - 1] * (2^{127})$ and the minimum negative number is $-(1) * (2^{-128})$.

Note that, $+0 < [+1 * 2^{-128}]$ and $-0 < [-1 * 2^{-128}]$.

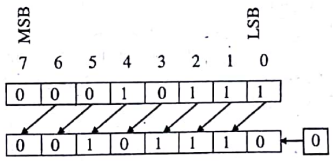
Question 21

Explain the arithmetic operations requiring right shift and left shift. Cite suitable examples where possible.

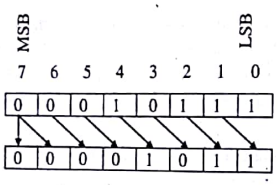
Answer:

Arithmetic shift operations, such as the left and the right shifts, are sometimes also known as signed shift operations. In case of such shift operations involving binary values, every bit of the operand get shifted by a given number of bit positions as required. The adjacent bit fills in the empty bit positions.

Arithmetic Left Shift: In the arithmetic left shift operations, the bits are shifted left depending on the number of shifts required. For example, for a single left shift operation, the bits are shifted to the immediate left bit position, whereas for a double left shift operation, the bits are shifted by two places to the left. Consider the single left shift operation for the binary number 10111 (i.e. 23) represented by a eight bit storage (i.e. 00010111). The resultant shifted number will be 00101110, as shown in the figure:



Arithmetic Right Shift: In the arithmetic right shift operations, the bits are shifted right depending on the number of shifts required. For example, for a single right shift operation, the bits are shifted to the immediate right bit position, whereas for a double right shift operation, the bits are shifted by two places to the right. Consider the single right shift operation for the binary number 10111 (i.e. 23) represented by a eight bit storage (i.e. 00010111). The resultant shifted number will be 00001011, as shown in the figure:



Question 22

IEEE format for floating point representation. **[WBUT 2006]**

Answer: Refer to section 2.4.

Question 23

Draw the logic diagram and discuss the advantages of a carry look ahead adder over conventional parallel adder. **[WBUT 2006, 2010]**

Answer:

Refer to section 2.7 for the first part of the question.

Second part: In case of parallel adders, cascading of n-full adders are required to add two n-bit numbers together. The carry signals thus 'ripple' through the adders from right to left and all the related logic gates take a non-zero time delay (propagation delay) to respond to a change in the input. This is because in case of conventional parallel adders, the result of an addition of two bits depends on the carry generated by the addition of the previous two bits. So, the sum of the most significant bit is only available after the carry signal has rippled through the adder from the least significant stage to the most significant stage. But, in case of a carry-look ahead adder, the carry does not have to depend explicitly on the preceding one and can be expressed as functions of relevant addend and augend bits. So, the overall delay is much lesser than the conventional parallel adder.

Question 24

Two 4 bit Arithmetic numbers are to be multiplied using the principle of carry save adders. Assume the numbers to be $A_3\ A_2\ A_1\ A_0$ and $B_3\ B_2\ B_1\ B_0$. Show the arrangement and interconnection of the adders and the input signals so as to generate an eight bit product $P_7\ P_6\ P_5\ P_4\ P_3\ P_2\ P_1\ P_0$. **[WBUT 2008]**

Answer:

'n' number of disjoint full-adders make an n-bit carry-save adder. Using this concept, the input to a carry-save adder consists of 3 n-bit numbers to be added and the output consists of n-sum bits (indicated as S) and n carry bits (indicated as C).

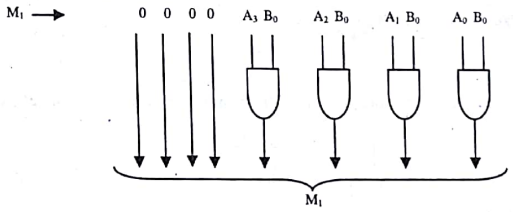
Going by this concept, multiplication of 2 4-bit numbers $A_3\ A_2\ A_1\ A_0$ and $B_3\ B_2\ B_1\ B_0$ are to be done with the help of carry-save adders.

Step 1: The two numbers if multiplied gives the product as:

				A_3	A_2	A_1	A_0	
				B_3	B_2	B_1	B_0	
0	0	0	0	A_3B_0	A_2B_0	A_1B_0	A_0B_0	$\rightarrow M_1$
0	0	0	A_3B_1	A_2B_1	A_1B_1	A_0B_1	0	$\rightarrow M_2$
0	0	A_3B_2	A_2B_2	A_1B_2	A_0B_2	0	0	$\rightarrow M_3$
0	A_3B_3	A_2B_3	A_1B_3	A_0B_3	0	0	0	$\rightarrow M_4$

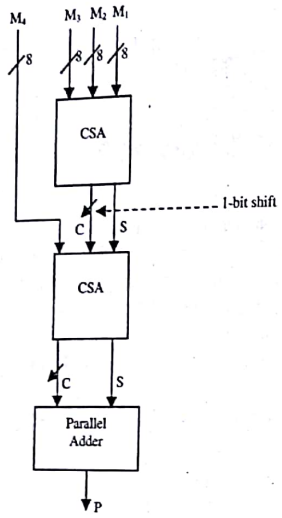
So, we indicate the inputs to the carry-save adders (CSA) as M_1, M_2, M_3 and M_4.

Step 2: How to generate M_1 - M_4?



Similarly M_2, M_3 and M_4 can also be obtained. So, each of M_1 - M_4 consists of 8 bits and hence 8 wires (1 wire per bit) are needed (per case) to specify the values.

Step 3: Block diagram showing the arrangement and interconnection of the CSAs and the input signals:

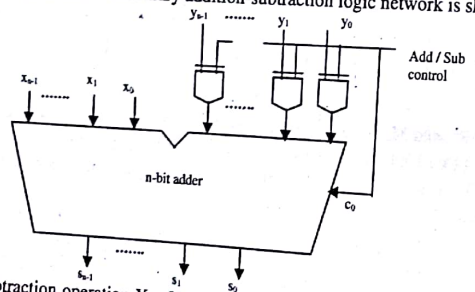


Question 25

Explain with a diagram, a 4-bit adder-subtractor circuit.

Answer:

The circuit diagram of a 4-bit binary addition-subtraction logic network is shown below:



Here the subtraction operation X – Y on 2's complement numbers X and Y is performed by adding the 2's complement of Y to X. The circuit is able to perform either addition or

subtraction depending on the value supplied to the Add/Sub control line. c_0 is a carry-in signal. For subtraction when both the Add/Sub control line and c_0 is 1, the value of the Y is 2's complemented. For addition both the Add/Sub control line and c_0 is 0 and Y vector is applied unchanged to one of the adder inputs.

Question 26

Briefly explain the IEEE 754 standard format for floating point representation.
[WBUT 2007, 2009]

Answer: Refer to section 2.4.

Question 27

If two n-bit numbers are added, then what would be the maximum length of the result?
[WBUT 2008]

Answer:

For unsigned numbers:

When two unsigned numbers are added, the sum can be a n+1 bit number or an n- bit number depending on whether there is a carry out of the MSB or not.
(Ex. Assuming n=3, 3+4=7 i.e., 011+100=111; 3+7=10, i.e., 011+111=1010).

For signed numbers:

When two n-bit signed numbers (MSB is the sign bit n-1 least significant bits are the magnitude bits) are added, the sum is a n-bit number with the MSB representing the sign of the sum. It should be noted that the position of the sign bit is fixed (it is the MSB i.e., the n^{th} bit), the magnitude of the sum should be a n-1 bit number. That is, the augend and the addend should be such that there should not be an overflow (carry into the MSB which is the sign bit). In case there is an overflow, there must be a correction, if required. No correction is needed in case of the two's complement addition.
Example: 3+(+3)=(1)00+(0)11=(1)11 which is minus zero in one's complement. This one's complement addition is correct because there is no end-around carry. However, if -2 and +3 are added in one's complement, there will be an end-around carry that will need a correction by adding the end-around carry to the LSB. On the other hand, no correction is needed in case of two's complement addition. i.e., -3+(+3)=(1)01+(0)11=1000. The result is now correct (i.e., zero) and needs no correction but just ignoring the end-around carry out of the sign bit.

Question 28

If two n-bit numbers are multiplied, then what would be the maximum length of the result? [WBUT 2008]

Answer:

For unsigned numbers:

When two n-bit unsigned numbers are multiplied, the product is a 2n-bit unsigned number. (Ex. 7*6=42, i.e., 111*110=101010).

For signed numbers:

When two n-bit signed numbers are multiplied (MSB is the sign bit and the least significant n-1 bits are the magnitude bits), the product is a 2n-1 bit signed number where the MSB represents the sign bit and the least significant 2(n-1)=2n-2 bits represent the magnitude. Example, (+7)(-7)=-49, i..e., (0,111)(1,111)=1,110001 in sign magnitude representation. In practice, 4-bit and 8-bit operand registers are used so that the operands are 0,111 and 1,111 but the product is 1,0110001.

Question 29

How NaN (Not a Number) and Infinity are represented in this standard. [WBUT 2007]

Answer:

NaN: NaN or Not a Number is the symbol for any invalid operation result. For example, dividing 0 by 0 or subtracting an infinite value from another would produce invalid results, which would be represented by NaN. A NaN result would allow an user to re-check a decision and figure out the problem.

Infinity: An exponent of all 1s and a fraction of all 0s are used to denote the values of +infinity and –infinity. The sign bit is used to distinguish between negative and positive infinity.

Question 30

Write short notes on Adder-subtractor circuit. [WBUT 2007, 2009]

Answer: Refer to question number 25.

Question 31

Represent the decimal value – 7.5 in IEEE – 754 single precision floating-point format [WBUT 2008]

Answer:

The IEEE-754 format is as follows.

Bit 31: sign bit

Bit 30-23: 8-bit biased exponent represented in excess-127 form

Bit 22-0: 23-bit normalized mantissa (magnitude), where the decimal point is assumed to lie just on the right of the most significant 1 bit in the real number (integer + fraction).

Now, -7.5 = -111.1 = -1.111 * 2 to the power +2

So, bit 31 = 1

30-23 = 127 + 2 = 129D = 10000001B

22-0 = 1110....0 (3 1's followed by 20 0's)

Hence the representation is C0F00000H.

Question 32

Write short notes on-

a) Non-restoring division method [WBUT 2008]

b) Booth's algorithm. [WBUT 2008]

Answer:

a) Refer to section 2.10 and question number 7.

b) Refer to section 2.8.

Question 33

Write $+7_{10}$ in IEEE 754 floating point representation in double precision. [WBUT 2009]

Answer:

Bit 31 (sign bit) for the given number is 0 (positive sign).

Bits 23 to 30 (exponent field): 1000000.

Bits 0 to 22 (significant): 1.11000000000000000000.

Hence, converting to hex, the result becomes 40E00000.

Question 34

Give the merits and demerits of floating-point and fixed-point representations for storing real numbers.

Answer:

Merits of **fixed-point** representation are as follows:

1) Representation of numbers is simple.
   Assumption of a fixed binary or radix point allows easy and simple representation of fractional numbers as well.
2) Arithmetic operations performed are simple and easy.
3) As numbers represented in fixed-point notation are spaced very evenly along the number line hence chances of getting exact results in calculations are more.
4) Hardware circuitry needed to implement such arithmetic operations are simple.
5) Calculations are fast.

Demerits of **fixed-point** representation are as follows:

1) Range of numbers that can be represented is very limited. Representations of very large numbers or very small fractions are not possible.

Merits of **floating point** representation are as follows:

1) Range of numbers (very large and very small) that can be represented are wide.
2) Such wide range of numbers can be represented with only a few digits.

Demerits of **floating point** representation are as follows:

1) Calculations need more time.
2) Complex circuitry is needed for computation.
3) As the numbers are non-evenly spaced along the number line hence calculations produce non-exact solutions that need to get rounded of to the nearest value.
4) Acute trade-off exists between range of numbers that can be represented and the precision.

Question 35

Explain Booth's algorithm for multiplication of signed-2's Complement numbers using a flowchart & show how the multiplication is accomplished using a suitable example. **[WBUT 2003, 2004, 2005, 2007, 2009, 2010]**

OR,

Explain Booth's Algorithm with flow-chart and suitable example. Illustrate this with an example by multiplying (– 9) × (– 13). **[WBUT 2006] [WBUT 2010]**

Answer: Refer to section 2.9 and problem number 5, 6 and 7 in Work out Example.

Question 36

Multiply –7 by –3 using Booth's algorithm. **[WBUT 2004]**

Answer: Refer to similar problem 5, 6 and 7 in Work out Example.

Question 37

Apply Booth's algorithm to multiply the two numbers $(+14)_{10}$ and $(-11)_{10}$. Assume the multiplier and multiplicand to be 5 bits each. **[WBUT 2005]**

Answer: Refer to section 2.9 for the first part. Refer to problems 5, 6 and 7 in Work out Example for the second part.

Question 38

a) A 32-bit floating-point binary number has a bit plus a sign for the exponent. Negative Numbers in the mantissa and exponent are in signed-magnitude representation. What are the longest and smallest positive qualities that can be represented excluding zero? Explain with example.
b) Explain with diagrams, Serial & Parallel Adders.
c) ADD A+B, where $A = 63.11236589 \times 10^{15}$ & $B = 0.002365991 \times 10^{-29}$. **[WBUT 2009]**

Answer:

a) In the 32-bit floating-point number system, negative numbers are represented in 2's complement form, where the leading bit is the sign bit. If, 24 bits are reserved for mantissa and 8 bits are reserved for signed exponent, then the maximum (longest) positive number is $+[2^{23} - 1] * (2^{127})$ and the minimum (smallest) positive number is $+(1) * (2^{-128})$.

b) Refer to section 2.6.

c) $A = 63.11236589 \times 10^{15} = 63112365890000000$
$B = 0.002365991 \times 10^{-29} = 2.365991E\text{-}32$
Result of addition: 126224731780000000

Question 39

Explain non-restoring division algorithm and explain the hardwire diagram. Perform the Restoring division operation with 19 divided by 8. [WBUT 2010]

Answer:

1st Part :

Non-restoring division algorithm:

Refer to section 2.10 and question number 7.

2nd Part:

Refer to similar Example number 15 of Worked-Out Example.

Question 40

What is the difference between carry-look ahead adder and carry ripple adder?

Answer:

In case of parallel / ripple-carry adders, cascading of n-full adders are required to add two n-bit numbers together. The carry signals thus 'ripple' through the adders from right to left and all the related logic gates take a non-zero time delay (propagation delay) to respond to a change in the input. This is because in case of conventional parallel adders, the result of an addition of two bits depends on the carry generated by the addition of the previous two bits. So, the sum of the most significant bit is only available after the carry signal has rippled through the adder from the least significant stage to the most significant stage. But, in case of a carry-look ahead adder, the carry does not have to depend explicitly on the preceding one and can be expressed as functions of relevant addend and augend bits. So, the overall delay is much lesser than the conventional parallel adder.

Question 41

What is the difference between carry-look ahead adder and carry ripple adder?

Answer:

Let X and Y are two numbers to be added and S the sum of the two. So if sign-bit of S is not the same as the sign-bits of X and Y (i.e. if X and Y are positive numbers with sign-bits = 0 and the sign-bit of S = 1) then it can be said that an overflow has occurred. Consider an example: X = +1001 and Y = +0111 i.e. S = (X = 01001) + (Y = 00111) = 10000. So, the leftmost 1 is an overflow.

Hence it can be concluded that if both X and Y have same sign then there is a possibility of overflow as their summation may not be represent able using the given number of bits

(in the above example, say four bits are allocated to represent the sum but actually five bits are needed). However if the two numbers to be added (i.e. X and Y) have opposite sign then usually overflow never occurs.

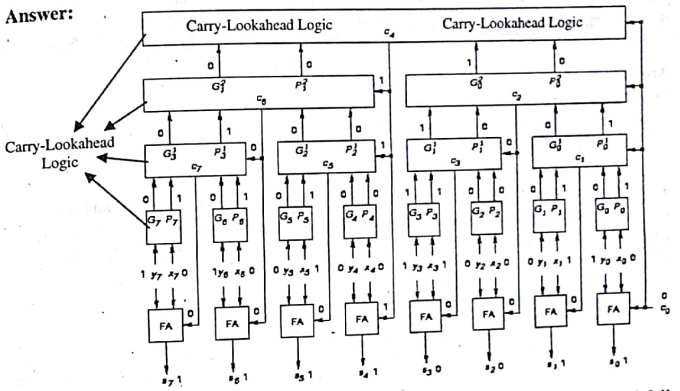
Overflow expression:

Overflow occurs if the expression $X_{n-1} Y_{n-1} \overline{S_{n-1}} + \overline{X_{n-1}} \overline{Y_{n-1}} S_{n-1}$ is true.

Question 42

What is the difference between carry-look ahead adder and carry ripple adder?

Answer:



The above picture shows an 8-bit carry-lookahead adder circuit implemented using 8 full adders (FA). Refer to Section 2.7 in Chapter 2 for a similar explanation of its working principles.

Question 43

What is the difference between carry-look ahead adder and carry ripple adder?

Answer:

In case of parallel / ripple-carry adders, cascading of n-full adders are required to add two n-bit numbers together. The carry signals thus 'ripple' through the adders from right to left and all the related logic gates take a non-zero time delay (propagation delay) to

respond to a change in the input. This is because in case of conventional parallel adders, the result of an addition of two bits depends on the carry generated by the addition of the previous two bits. So, the sum of the most significant bit is only available after the carry signal has rippled through the adder from the least significant stage to the most significant stage. But, in case of a carry-look ahead adder, the carry does not have to depend explicitly on the preceding one and can be expressed as functions of relevant addend and augend bits. So, the overall delay is much lesser than the conventional parallel adder.

Question 44

In a carry-look adder, compute the total time needed to perform one addition using gate delay of each gate δ μs and with no delay involved in the connecting wires.

Answer:

Operations of carry look ahead adder for addition is

i) generation of generate & propagate function

ii) generation of carry

iii) generation of summation.

The carry look ahead adder requires AND OR gates with as many as n +1 inputs (for C_n), which is impractical in hardware realization. To comprise, we pack n = 4 bits as a block with carry look ahead, still use ripple carry between the blocks (C_4, C_8, C_{12}, & C_{16}).

There are n/4 blocks in a n-bit adder and the total gate delays can be found as:

Operations	No. of gate delays
generation of generate & propagate function	1
Generation of carry Ci (i=1,2,3,4) in blocks 1	2
Generation of carry Ci (i=5,6,7,8) in block 2	2
. . .	. . .
Generation of sum Si	3
Total	1+2(n/4)+3

$\therefore$ total gate delay for 8 bit

Adder $= 1+2\times\left(\frac{8}{4}\right)+3$

$= 1+4+3=8$

$\therefore$ Total time required for an addition $= (8\times\delta)\,\mu s = 8\delta\,\mu s$

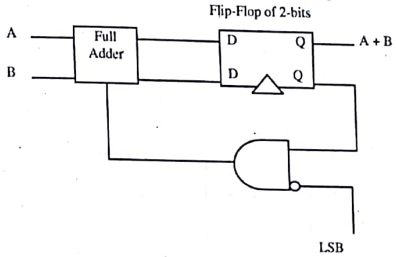
Question 45

What is serial adder? Discuss it briefly with diagram.

Answer:

The serial adder accepts two arbitrary length serial strings of digits as inputs, which may be provided by two simultaneously clocked shift registers. The output is the sum of the two input bi streams.

In the diagram shown, A and B are the arbitrary length input streams produced, which are fed to the full adder circuit. The output is the sum of the strings of data.



Question 46

Give reason(s) for converting a non-normalized floating-point number in normalized one.

Answer:

Converting a de-normalized number into the normalized form helps to gain one extra (i.e. more) significant binary digit in the obtained fraction. There is no need to store the leftmost digit if its known to be one as it can be always assumed to be present. In the IEEE 754 representation, to the left of the fraction, there is always an implied 1 bit and a binary point. The fractional part of a floating-point number is thus referred to as a significand.

Question 47

Describe the working principle of binary incrementer.

Answer:

The increment micro-operation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. This micro-operation is easily implemented with a binary counter. Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment micro-operation must be done with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders connected in cascade. The diagram, below, shows a 4-bit combinational circuit incrementer. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from A_0 through A_3, adds one to it, and generates the incremented output in S_0 through S_3. The output carry C_4 will be 1 only after incrementing binary 1111. This also causes outputs S_0 through S_3 to go to 0.

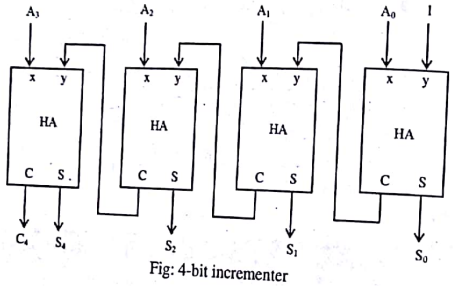


Fig: 4-bit incrementer

Worked-Out Examples

Example 1

Convert: (i) $(736.4)_8$ to decimal
(ii) $(101101)_2$ to decimal
(iii) $(F3)_{16}$ to decimal
(iv) $(1010111101110010)_2$ to octal and hexadecimal.

Solution:

(i) $(736.4)_8$ to decimal

$(736.4)_8 = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1} = (478.5)_{10}$.

(ii) $(101101)_2$ to decimal

$(101101)_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (45)_{10}$.

(iii) $(F3)_{16}$ to decimal

$(F3)_{16} = F \times 16 + 3 = 15 \times 16 + 3 = (243)_{10}$.

(iv) $(1010111101110010)_2$ to octal

1 010 111 101 110 010 = 127562 (grouping three binary digits from right to left and finding the integer the value of each group).

$(1010111101110010)_2$ to hexadecimal

1010 1111 0111 0010 = AF72 (grouping four binary digits from right to left and finding the integer value of each group).

Example 2

Compute 9's and 10's complements of 667899 and 546700.

Solution:

9's complement of 667899 and 54670 are:

999999 – 667899 = 332100
and 99999 – 54670 = 45329.

10's complement of 667899 and 54670 are:

999999 – 667899 + 1 = 332100 + 1 = 332101
and 99999 – 54670 + 1 = 45329 + 1 = 45330.

Example 3

Compute 1's and 2's complements of 1011110 and 1110011.

Solution:

1's complement of 1011110 and 1110011 are 0100001 and 0001100 respectively.

2's complement of 1011110 and 1110011 are 0100010 and 0001101 respectively.

Example 4

Add: (i) + 6 and +13 (ii) + 6 and – 13
(iii) – 6 and –13 (iv) Sub: - 6 and –13.

Solution:

(i) To add +6 and +13 i.e. to add 00000110 and 00001101. Hence result is *'+19' or 00010011.*

(ii) To add +6 and –13 i.e. to add 00000110 and 11110011 (2's complement of +13). Hence result is '-7' or 11111001 (2's complement of the result is 00000111, which is binary equivalent of +7. So the original negative number is equal to –7).

(iii) To add –6 and –13 i.e. to add 11111010 (2's complement of +6) and 1110011 (2's complement of +13). This gives 111101101. But the leftmost '1' is the carry and is discarded giving the actual result as *'-19' or 11101101* (2's complement of the result is 00010011 which is binary equivalent of +19. So the original negative number is equal to –19).

(iv) To subtract –6 and –13 i.e. (-6) – (-13) = +7. Taking the 2's complement of both the numbers, it gives 11111010 – 11110011. If again the 2's complement of the subtrahend (i.e. 2's complement of –13) is taken, then it gives +13. Hence to add –6 and +13 i.e. 11111010 + 00001101 = 100000111.

Discarding the end carry, it gives 00000111 (or +7) as the final result.

[*Note:* In case of 2's complement arithmetic operations like addition and subtraction, if the sign bit of a number is 1 i.e. if it's a negative number, then the entire number must be represented in 2's complement form]

Example 5

Perform (-9) x (-13) using Booth's algorithm.

Solution:

Both the numbers are converted in their 2's complement form as they are (-ve).

Hence, BR = 10111 and QR = 10011.

SC = 5

Q_n	Q_{n+1}		AC	QR	Q_{n+1}	SC
		Initial	00000	10011	0	101
1	0	Subtract BR	+01001			
			01001			
		ashr	00100	11001	1	100
1	1	ashr	00010	01100	1	011
0	1	Add BR	+10111			
			11001			
		ashr	11100	10110	0	010
0	0	ashr	11110	01011	0	001
1	0	Subtract BR	01001			
			00111			
		ashr	00011	10101	1	000

Hence, the 10-bit final product appears in AC and QR and is = 0001110101 = +117.

Example 6

Perform A = 0100110 x B = 1011011 using Booth's algorithm.

Solution:

SC = 7-bits; BR = 1011011; $\overline{BR} + 1 = 0100101$

Q_n	Q_{n+1}		AC	QR	Q_{n+1}	SC
0	0	Initial	0000000	0100110	0	7
1	0	ashr	0000000	0010011	0	6
		sub BR	+0100101			
			0100101	0010011		
		Ashr	0010010	1001001	1	5
		Ashr	0001001	0100100	1	4
1	1	Add BR	+1011011			
0	1		1100100	0100100		
		ashr	1110010	0010010	0	3
		Ashr	1111001	0001001	0	2
0	0	Sub BR	+0100101			
1	0	Ashr	00011110	0001001		
			0001111	0000100	1	1
0	1	Add BR	+1011011			
			1101010	0000100		
		Ashr	1110101	0000010	0	0

So, the result = 00010101111110 (in 2's complement form) = -1406

Example 7

Apply Booth's algorithm to multiply the two numbers $(+14)_{10}$ and $(-12)_{10}$.

Solution:

Let multiplicand +14 be in register BR and multiplier –12 in register QR.

i.e. BR = 01110 and QR = 01100.

Now the two's complement of the multiplier will be 10100 and will be stored in QR.

To convert this multiplier (10100) into Booth's multiplier, it should be converted as per the following rules:

(a) If $bit_m = 0$ and $bit_{m-1} = 0$ then result is 0.

(b) If $bit_m = 1$ and $bit_{m-1} = 0$ then result is –1.

(c) If $bit_m = 1$ and $bit_{m-1} = 1$ then result is 0.

(d) If $bit_m = 0$ and $bit_{m-1} = 1$ then result is 1.

(here consider bit_{m-1} the lower bit to the right).

So computing as per the above rules, starting from the least significant bit, 10100 gets converted into –11-100 (which is the Booth's multiplier).

Now perform the operation i.e. (01110) X (–11-100) by ordinary simple multiplication to get the result as 1101011000 which is a negative number. So it is needed to get converted into its two's complement form i.e. -0010101000 (or –168), which is the ultimate result.

Example 8

Compute the product of following pairs of unsigned integers. Generate the full 8-bit result.

(i) Ob1001 X Ob0110

(ii) Ob1111 X Ob1111

Solution:

(i) Ob1001 X Ob0110 i.e. 1001 X 0110

(i.e. 9 X 6) = 00110110 (+54) [the result can be obtained simply by multiplying the two numbers by performing simply binary multiplication].

(ii) Ob1111 X Ob1111 i.e. 1111 X 1111

(i.e. 15 X 15) = 11100001 (+225).

Example 9

Use 8-bit two's complement integers, perform the following computations:

(i) -34 + (-12) (ii) 17 – 35

(iii) -22 – 7 (iv) 18 – (-5)

(v) 26 – (–4) (vi) 1–7

Solution:

(i) -34 + (-12)

-34 in its two's complement form will be represented as 11011110 while –12 in its two's complement form will be represented as 11110100. On addition of these two two's complemented numbers, the result obtained is 111010010. On discarding the carry-bit from the left we get the result as ***11010010*** or - 46 (the two's complement form of the result is 00101110 which is + 46. Hence – 46 is the required result).

(ii) 17 – 35 i.e. 17 + (-35).

+17 is given by 00010001 while –35 in its two's complement representation is given as 11011101. On addition of these two numbers the result obtained is ***11101110*** or –18 (the two's complement form of the result is 00010010 which is binary equivalent of + 18. hence –18 is the result).

(iii) -22 – 7 i.e. (-22) + (-7)

Two's complement form of –22 is 11101010 whereas that of –7 is 11111001. So addition of these two numbers gives the result as 111100011. Discarding the end carry-bit we get

11100011 or –29 as the result (as two's complement of the result obtained is 00011101 which is the binary equivalent of +29, hence it can be said that –29 is the result).

(iv) 18 – (-5) i.e. 18 + 5

Here, +18 is given as 00010010 whereas +5 is given as 00000101. On addition of these two numbers, the result obtained is *00010111* or +23.

(v) 26 – (–4) i.e. 26 + 4 = 00011010 + 00000100 = 00011110 = +30.

(vi) 1 – 7 i.e. 1 + (-7) = 00000001 + 11111001 = 11111010 = -6.

Example 10

A floating-point number system uses 16 bits for representing a number. The most significant bit is the sign bit. The least significant nine bits represent the mantissa and remaining 6 bits represent the exponent. Assume the numbers are stored in the normalized format with one hidden bit.

i) Give the representation of $-1.6 * 10^3$ in this number system.

ii) What is the value represented by 0000100110000000?

Solution:

(i) Since the Mantissa is normalized, its value must lie between 0.5 and $1-2^{-10}$.

Now, $-1.6 \times 10^3 = -1600 = -1600/2048 \times 2^{11}$

$= -0.78125 = (0.1100100000)_2$.

The mantissa is represented as a 10-bit number with its leading bit (i.e. 1) always hidden or suppressed.

Hence the Mantissa (bits 8 to 0) is 100100000.

Now Exponent is +11 i.e. 1011. Since it is biased with 100000, the actual Exponent (Bits 14 to 9) is 100000 + 1011 = 101011.

The sign of the number is negative.

Hence the floating point representation of -1.6×10^3 is 1101011100100000.

(ii) 0000100110000000 i.e. the sign of the number is positive.

The Mantissa part is (actually a 10-bit number) 0.1110000000.

The Exponent part is 000100. Since the Exponent is biased, its actual value is negative and can be obtained by taking the 2's complement of 000100, which is –11100, i.e. –28.

Hence the number is $-1110000000 \times 2^{-28} = 11.1 \times 2^{-30} = 3.5 \times 10^{-9}$ (approx.).

Example 11

What is the result of the following addition $(7176)_8$ and $(1653)_8$ without changing them to any other base?

Solution:

Result of addition of $(7176)_8$ and $(1653)_8$ is $(11051)_8$.

Example 12

The base-ten number 203 can be expressed as a sum of powers of two as $2^7 + 2^6 + 2^3 + 2^1 + 2^0$

What is the corresponding base-two (binary) number?

Solution:

11001011

Explanation

Every power of two corresponds to a position in a binary number. The positions are numbered from right to left, starting from zero. Every power of two that appears in the sum (0, 1, 3, 6, and 7) corresponds to a one in the binary number. So, the answer has 1's in positions 0, 1, 3, 6, and 7.

Example 13

Data is represented in computers in the form of binary numbers. Suppose that a binary number such as 10111001001001000111001011001100 is stored in a computer. What this data represents? What does your answer illustrate about data representation?

Solution:

There is no way to tell what this data represents, without knowing more about the context. Binary numbers do not have a fixed meaning. The same binary number can be used to represent many different kinds of data, depending on how that data is encoded. This 32-bit number might represent four ASCII characters, two Unicode characters, a single real number, the colors of a few pixels in some image, a date, a bit of music, and so on. This illustrates that binary numbers are *symbols,* which do not have a fixed, intrinsic

meaning. They get their meaning from context and from the particular encoding that is used.

Example 14

What is the base-10 integer that is represented in base-2 as 1100101_2?

Solution:
Each bit in a binary number corresponds to a power of two. The rightmost bit corresponds to 2^0, the next bit to the left corresponds to 2^1, the next bit to 2^2, and so on.
So,
$1100101_2 = 2^6 + 2^5 + 2^2 + 2^0 = 64 + 32 + 4 + 1 = 101$

Example 15

Divide 0111 by 0011 using restoring division technique.

Solution:
The dividend is initially stored in Q register (4-bits) and the divisor in M register.
Now, as the rule of the division goes, 5-bit registers are used for the divisor M and the remainder A. Also, the high-order bit of M and all the bits in A are initially cleared (or made 0).
Therefore, initial values in Q = 0111, in M = 00011 (5-bit register) and that in A = 00000 (5-bit register). 2's compliment of M is considered while subtracting.
$a_4 \rightarrow$ sign-bit of A; $q_0 \rightarrow$ cleared if a_4 is –ve and set if +ve.

The division steps are as explained below:
1st Cycle:

A					Q				Explanation
a_4	a_3	a_2	a_1	a_0	q_3	q_2	q_1	q_0	
0	0	0	0	0	0	1	1	1	Initial Values

0	0	0	0	0	1	1	1	0	Shifted Left as a Pair (A and Q)
1	1	1	0	1	1	1	1	0	M is subtracted from A
0	0	0	0	0	1	1	1	0	Since result is –ve, A is restored (Add M to A)
0	0	0	0	0	1	1	1	0	q_0 is cleared

2nd Cycle:

0	0	0	0	1	1	1	0	0	Shifted Left as a Pair (A and Q)
1	1	1	1	0	1	1	0	0	M is subtracted from A
0	0	0	0	1	1	1	0	0	Since result is –ve, A is restored (Add M to A)
0	0	0	0	1	1	1	0	0	q_0 is cleared

3rd Cycle:

0	0	0	1	1	1	0	0	0	Shift Left as a Pair (A and Q)
0	0	0	0	0	1	0	0	0	M is subtracted from A
0	0	0	0	1	1	0	0	1	q_0 is set as the result is +ve.

4th Cycle:

0	0	0	0	1	0	0	1	0	Shift Left as a Pair (A and Q)
1	1	1	1	0	0	0	1	0	M is subtracted from A
0	0	0	0	1	0	0	1	0	Since result is –ve, A is restored (Add M to A)
0	0	0	0	1	0	0	1	0	q_0 is cleared

Therefore, the final value in A = 0001 is the remainder (the extra sign-bit is not considered) and that in Q = 0010 is the quotient.

Example 16

Divide 1000 by 0011 using restoring division technique.

Solution:
The dividend is initially stored in Q register (4-bits) and the divisor in M register.
Now, as the rule of the division goes, 5-bit registers are used for the divisor M and the remainder A. Also, the high-order bit of M and all the bits in A are initially cleared (or made 0).
Therefore, initial values in Q = 1000, in M = 00011 (5-bit register) and that in A = 00000 (5-bit register). 2's complement of M is considered while subtracting.
$a_4 \rightarrow$ sign-bit of A; $q_0 \rightarrow$ cleared if a_4 is –ve and set if +ve.

The division steps are as explained below:

1st Cycle:

A					Q				Explanation
a_4	a_3	a_2	a_1	a_0	q_3	q_2	q_1	q_0	
0	0	0	0	0	1	0	0	0	Initial Values
0	0	0	0	1	0	0	0	0	Shifted Left as a Pair (A and Q)
1	1	1	1	0	0	0	0	0	M is subtracted from A
0	0	0	0	1	0	0	0	0	Since result is –ve, A is restored (Add M to A)
0	0	0	0	1	0	0	0	0	q_0 is cleared

2nd Cycle:

0	0	0	1	0	0	0	0	0	Shifted Left as a Pair (A and Q)
1	1	1	1	1	0	0	0	0	M is subtracted from A
0	0	0	1	0	0	0	0	0	Since result is –ve, A is restored (Add M to A)
0	0	0	1	0	0	0	0	0	q_0 is cleared

3rd Cycle:

0	0	1	0	0	0	0	0	0	Shift Left as a Pair (A and Q)
0	0	0	0	1	0	0	0	0	M is subtracted from A
0	0	0	0	1	0	0	0	1	q_0 is set as the result is +ve.

4th Cycle:

0	0	0	1	0	0	0	1	0	Shift Left as a Pair (A and Q)
1	1	1	1	1	0	0	1	0	M is subtracted from A
0	0	0	1	0	0	0	1	0	Since result is –ve, A is restored (Add M to A)
0	0	0	1	0	0	0	1	0	q_0 is cleared

Therefore, the final value in A = 0010 is the remainder (the extra sign-bit is not considered) and that in Q = 0010 is the quotient.

Example 17

Divide 1000 by 0011 using non-restoring division technique.

Solution:

The non-restoring division technique is almost the same as the restoring one. However, unlike the restoring technique, in this case, the restore operations are not used and the contents of M register is added to that of A register whenever the sign of A is 1.

The process proceeds as follows:

The dividend is initially stored in Q register (4-bits) and the divisor in M register.

Now, as the rule of the division goes, 5-bit registers are used for the divisor M and the remainder A. Also, the high-order bit of M and all the bits in A are initially cleared (or made 0).

Therefore, initial values in Q = 1000, in M = 00011 (5-bit register) and that in A = 00000 (5-bit register). 2's complement of M is considered while subtracting.

$a_4 \rightarrow$ sign-bit of A; $q_0 \rightarrow$ cleared if a_4 is –ve and set if +ve.

The division steps are as explained below:

1st Cycle:

A					Q				Explanation
a_4	a_3	a_2	a_1	a_0	q_3	q_2	q_1	q_0	
0	0	0	0	0	1	0	0	0	Initial Values
0	0	0	0	1	0	0	0	0	Shifted Left as a Pair (A and Q)
1	1	1	1	0	0	0	0	0	M is subtracted from A
1	1	1	1	0	0	0	0	0	q_0 is cleared as sign of A is -ve

2nd Cycle:

1	1	1	0	0	0	0	0	0	Shifted Left as a Pair (A and Q)
1	1	1	1	1	0	0	0	0	A + M (i.e. 11100 + 00011)
1	1	1	1	1	0	0	0	0	q_0 is cleared as sign of A is -ve

3rd Cycle:

1	1	1	1	0	0	0	0	0	Shift Left as a Pair (A and Q)
0	0	0	0	1	0	0	0	0	A + M (i.e. 11110 + 00011)
0	0	0	0	1	0	0	0	1	q_0 is set as sign of A is +ve.

4th Cycle:

0	0	0	1	0	0	0	1	0	Shift Left as a Pair (A and Q)
1	1	1	1	1	0	0	1	0	M is subtracted from A
1	1	1	1	1	0	0	1	0	q_0 is cleared
0	0	0	1	0					A + M (i.e. A + 00011) to restore the remainder

Therefore, the final value in A = 0010 is the remainder (the extra sign-bit is not considered) and that in Q = 0010 is the quotient.

Example 18

Convert IEEE 64-bit format ABCD000000000000$_{16}$ in decimal value.

Solution:

(abcd0000000000000)$_{16}$ = (1.010 1011 1100 1101 0000 ....0000)

Biased exponent = (010 1011 1100)b = (700)d

Hence, actual exponent = 700 − 1023 = −323d

Mantissa (considering the "implied 1") = (1.1101)b = (1+.5+.25+.0625) d = 1.8125d

Therefore, the actual number is (−1.8125 * 2 to the power −323).

Example 19

Convert IEEE 32-bit format 40400000_{16} in decimal value.

Solution:

$(40400000)_{16}$ = (0.100 0000 0.100 0000 0000 0000 0000 0000)b

= +1.1 * 2 to the power (128-127) = +1.1 * 2 = +11b = (+3)d

PS: "1" bit just before the decimal point is the "implied 1".

Example 20

Write $+710_{10}$ in IEEE 64-bit format.

Solution:

+710d = +(1011000110)b = +(1.011000110 * 2 to the power 9)b.

So, the sign bit is 0; the normalized mantissa part is the 52-bit number 0110001100.....0 and the 11-bit exponent part is (9+1023)d = (1032)d = (10000001000)b.

Hence, +710d is represented as 0,10000001000.0110001100..........0

(the ',' and the '.' signs have been added just for understanding).

[Note: d stands for decimal and b stands for binary]

Example 21

Multiply +12 and −11 using modified Booth's sequential method of multiplication. Draw the corresponding circuit block diagram. **[WBUT 2008]**

Solution:

Modified Booth's algorithm replaces the immediately adjacent addition and subtraction pairs by a single operation.


```
A      →        00 00 00 00 00 00 11 00  → 12
B      →        1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1   → -11
Y      →        0 0 0 0 0 0 0 0 0 0 0 -1 0 1 0 1   → recoded multiplier
------------------------------------------------------------
Add A  →        + 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
2-Bit Shift →     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
Add A  →        + 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
------------------------------------------------------------
                0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0
2-Bit Shift →   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0
Add -A →        + 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0
------------------------------------------------------------
                1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0
2-Bit Shift →   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0
2-Bit Shift Only →  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0
2-Bit Shift Only →  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0
2-Bit Shift Only →  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0
2-Bit Shift Only → 1 11 11 11 11 11 11 11 11 11 11 11 01 11 11 00
2-Bit Shift Only → 1 11 11 11 11 11 11 11 11 11 11 11 11 01 11 11
Result         → -132
```

Example 22

Multiply (+15) and (−11) using Booth's algorithm.

Solution:

A	0000 11 11	15
X	1111 01 01	-11
Y	000-11-11-1	recoded multiplier

Add -A	+1111 00 01
Shift	1111 10 00 1
Add A	+0000 11 11
	0000 01 11 1
Shift	0000 00 11 11
Add -A	+1111 00 01
	1111 01 00 11
Shift	1111 10 10 011
Add A	+0000 11 11
	0000 10 01 011
Shift	0000 01 00 1011
Add -A	+1111 00 01
	1111 01 01 1011
Shift	1111 10 10 11011
Shift Only	1111 11 01 011011
Shift Only	1111 11 10 1011011
Shift Only	1111 11 11 01011011 -165 (Final Answer)

Example 23

Multiply (-12) and (+6) using Booth's algorithm. **[WBUT 2011]**

Solution:

A	1111010 0	-12
X	x 0000011 0	6
Y	000010-10	recoded multiplier

Shift Only	0000000 00
Add -A	+0000110 0
	0000110 00
Shift	0000011 000
Shift Only	0000001 1000
Add A	+1111010 0
	1111011 1000
Shift	1111101 11000
Shift Only	1111110 111000
Shift Only	1111111 0111000
Shift Only	1111111 10111000
Shift Only	1111111 11011100 0 -72 (answer)

Example 24

Using IEEE single-precision floating point numbers to compute 13.25+4.5. **[WBUT 2009]**

Solution:

In order to add floating-point numbers, it is required to first represent them with the same exponent.

i.e. 13.25 = 1.325 * 10^1

4.5 = 0.45 * 10^1

Therefore, (1.325 + 0.45) * 10^1 = 1.775 * 10^1

Now, e = 1 and s = 1.775. Hence, true sum = 17.75

Example 25

Convert –32.75 to IEEE 754 single-precision floating point.

Solution:
Conversion to 32-bit single-precision floating point:
Bit 31 is the –ve sign-bit = 1;
Bits 23 to 30 (exponent field) = 10000100 (for exponent 132-127 = 5);
Bits 0 to 22 (significand) = 1 .00000110000000000000000 (decimal value of the significand = 1.0234375);
Hence, in hexadecimal, the converted number is: C2030000

Example 26

Apply restoring division procedure to divide decimal number 20 by decimal 3.

Solution:
An overview of solving the problem is given here.

00000	10100 (20)	
Shift	00001	0100__
Sub	11101	
Set q_0=0	11110	01000 (on setting q_0=0)
Restore 00011		
i.e.	00001	01000
Marks the end of first cycle.		
Shift	00010	1000__
Sub	11101	
Set q_0=0	11111	10000
Restore 00011		
i.e.	00010	10000
Marks the end of second cycle.		
Shift	00101	0000__
Sub	11101	
Set q_0=1	00010	00001
Marks the end of third cycle.		

..... This will continue in the fourth cycle.....

Fifth cycle:		
Shift	00010	0011__
Sub	11101	
Set q_0=0	11111	00110

So, at the end of the fifth cycle, 11111 is the remainder and 00110 is the quotient.
As the remainder is negative, hence we need to add the divisor to get its original value.
i.e. 11111 + 00011 = 00010.
Hence, quotient = 00110 and remainder = 00010.]

Example 27

Multiply -5 by -3 using Booth's algorithm.

Solution:

A	1 1 1 1 1 0 1 1	-5
X	x 1 1 1 1 1 1 0 1	-3
Y	0 0 0 0 0 -1 1 -1	recoded multiplier
Add -A	+ 0 0 0 0 0 1 0 1	
Shift	0 0 0 0 0 0 1 0 1	
Add A	+ 1 1 1 1 1 0 1 1	
	1 1 1 1 1 1 0 1 1	
Shift	1 1 1 1 1 1 1 0 1 1	
Add -A	+ 0 0 0 0 0 1 0 1	
	0 0 0 0 0 0 1 1 1 1	
Shift	0 0 0 0 0 0 0 1 1 1	
Shift Only	0 0 0 0 0 0 0 0 1 1 1 1	
Shift Only	0 0 0 0 0 0 0 0 0 1 1 1 1	
Shift Only	0 0 0 0 0 0 0 0 0 0 1 1 1 1	
Shift Only	0 0 0 0 0 0 0 0 0 0 0 1 1 1 1	
Shift Only	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1	15

Example 28

Write down the steps to subtract 110·101101 from 10110·1110

Solution:

10110.1110 – 110.101101

Step 1:

A + $1.00111011111100001110001 * 2^{13}$ = 10110.1103515625

B + $1.10111000011001111000011 * 2^{6}$ = 110.1010971069336

Step 2: Alignment

A + $1.00111011111100001110001|000 * 2^{13}$

B + $0.00000011011100001100111|101 * 2^{13}$

Step 3:

A-B + $1.00111000100000000001001|011 * 2^{13}$

Step 4: After normalization

A-B + $1.00111000100000000001001|01 * 2^{13}$

Step 5: Rounding to zero

A-B + $1.00111000100000000001001 * 2^{13}$ = 10000.0087890625

Step 6: Rounding to nearest even number

A-B + $1.00111000100000000001001 * 2^{13}$ = 10000.0087890625

Step 7: Rounding to plus infinity

A-B + $1.00111000100000000001010 * 2^{13}$ = 10000.009765625

Example 29

Perform the subtraction with the following unsigned binary number by taking the 2's complement of the subtrahend: 11010 – 1101.

Solution:

2's complement of the subtrahend i.e. 2's complement of 1101 = 11. Therefore, 11010 - 11 = 10111.

Short Questions & Answers

1. What is NaN or Not a Number in IEEE 754?

Answer:

NaN or Not a Number is the symbol for any invalid operation result. For example, dividing 0 by 0 or subtracting an infinite value from another, would produce invalid results, which would be represented by NaN. A NaN result would allow an user to re-check a decision and figure out the problem.

2. Why do computers use binary numbers, rather than ordinary, base-10 numbers?

Answer:

Values in a computer are represented by wires, which can be on or off. It is convenient to use these two values to represent the binary digits 0 and 1.

3. In a computer system, why can't only floating-point numbers be used?

Answer:

A computer system has integers and floating-point numbers. Now, in case of application requiring floating-point representations, extensive manipulation of the operations are required. Also, most of the floating-point applications require only integer values. So, it is best to have both integers and floating-point numbers.

4. Why is Carry Look-Ahead Adder (CLA) called a fast parallel adder?

Answer:

CLA is a very fast adder circuit which speeds up the generation of carry signals by using the same (almost) logic as that of parallel adders. That is why it is also called a fast parallel adder.

Multiple Choice Type Questions

1. With 2's Complement representation, the range of values that can be represented on the data bus of an 8 bit micro-processor is given by: [WBUT 2003]

(i) -128 to + 127 (ii) - 128 to + 128
(iii) - 127 to 128 (iv) - 256 to + 256

Answer: (i)

2. When signed numbers are used in binary arithmetic, then which one of the following notations would have unique representation for zero? [WBUT 2003, 2007, 2008, 2009]

(i) Sign Magnitude (ii) 1's complement
(iii) 2's complement (iv) None

Answer: (iii)

3. What is the 2's complement representation of - 24 in a 16 bit micro-computer? [WBUT 2003]

(i) 0000 0000 0001 1000 (ii) 1111 1111 11100111
(iii) 1111 1111 11101000 (iv) 001 0001 11110011

Answer: (iii)

4. Subtractor can be implemented using [WBUT 2006]

a) adder b) complementer
c) both (a) & (b) d) none of these

Answer: c)

5. Maximum n bit 2's complement number is [WBUT 2007, 2009]

a) $2n$ b) $2^n - 1$
c) $2^{n-1} - 1$ d) Cannot be said

Answer: (c)

6. State true or false:
Adding 01101101_2 to 10100010_2 in 8-bit 2's complement binary will cause an overflow: [WBUT 2007, 2009]

a) True b) False

Answer: (b)

7. The conversion of $(FAFAFA)_{16}$ into octal form is [WBUT 2008]

a) 76767676 b) 76575372

c) 76737672 d) 76727672

Answer: (b)

8. Which logic gate has the highest speed? [WBUT 2009]

a) ECL b) TTL c) RTL d) DTL

Answer: a)

9. Booth's algorithm for computer arithmetic is used for [WBUT 2009]

a) multiplication of numbers in sign magnitude form
b) multiplication of numbers in 2's complement form
c) division of numbers in sign magnitude form
d) division of numbers in 2's complement form

Answer: b)

10. The conversion (FAFAFB) 16 into octal form is [WBUT 2009]

a) 76767676 b) 76575372
c) 76737672 d) None of these

Answer: d)

11. A decimal no. has 30 digits. Approximately, how many would the binary representation have? [WBUT 2009]

a) 30 b) 32 c) 60 d) 90

Answer: d)

12. The logic circuit in ALU is [WBUT 2009]

a) Entirely combinational b) Entirely sequential
c) Combinational cum sequential d) None of these

Answer: c)