I.C.E.™

Inspiring Creativity & Endeavour

IES | GATE | PSU

COMPUTER ORGANIZATION AND ARCHITECTURE

For

COMPUTER SCIENCE

COMPUTER ORGANIZATION AND ARCHITECTURE

SYLLABUS

Machine instructions and addressing modes, ALU and data-path, CPU control design, Memory interface, I/O interface (Interrupt and DMA mode), Instruction pipelining, Cache and main memory, Secondary storage.

ANALYSIS OF GATE PAPERS

Exam Year	1 Mark Ques.	2 Mark Ques.	Total
2003	2	3	8
2004	1	7	15
2005	4	8	20
2006	1	7	15
2007	2	6	14
2008	-	12	24
2009	2	4	10
2010	1	4	9
2011	1	4	9
2012	2	2	6
2013	1	4	9
2014 Set-1	1	3	7
2014 Set-2	1	3	7
2014 Set-3	1	2	5
2015 Set-1	1	1	3
2015 Set-2	1	2	5
2015 Set-3	1	2	5
2016 Set-1	1	2	5
2016 Set-2	1	5	11
2017 Set-1	4	5	14
2017 Set-2	4	3	10
2018	3	4	11

CONTENTS

Topics **Page No**

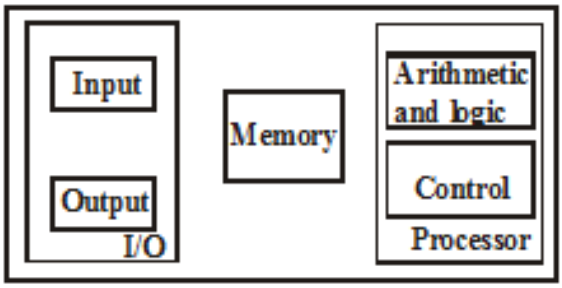
1 OVERVIEW OF COMPUTER SYSTEM

1.1 INTRODUCTION

Digital computer or simply computer is fast electronic calculating machine that accepts digitized input information processes it according to a list of internally stored instructions, and produces the resulting output information. The list instruction is called a computer program, and the internal storage is called computer memory. Many types of computers exist that differ in many factors lime size, cost, computational power and intended use.

1.2 FUNCTIONAL UNITS

A computer consists of five functionally independent main parts:



Basic functional units of a computer

1.2.1 Input unit

- Accepts coded information from human operators, from electromechanical devices such as keyboards, or from other computers over digital communication lines.
- Many other kinds of input devices are available, including Joysticks, Trackballs, and mousses. These are often used as graphic input devices in conjunction with displays.

1.2.2 Memory Unit

- The function of the memory unit is to store program and data. There are two classes of storage, called primary and secondary.
- Primary storage is a fast memory that operates at electronic speeds. Programs must be stored in the memory while they are being executed.
- The memory contains a large number of semiconductor storage cells, each capable of storing one bit of information.
- Programs must reside in the memory during execution. Instructions and data can be written into the memory or read out under the control of the processor.
  Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called random-access-memory (RAM).
- The time required to access one word is called the memory access time. This time is fixed, independent of the location of the word being accessed.
- The small, fast RAM units are called caches, they are tightly coupled with the processor and are often contained on the same integrated circuit to achieve high performance.
- The main memory is largest and slowest unit. Although primary storage is essential, it tends to be expensive. Thus, additional cheaper, secondary storage is used when large amounts of data any many programs which are infrequently used have to be stored.

1.2.3 Arithmetic and Logic Unit

- Most of the operations are executed in the arithmetic and logic unit (ALU)of the processor.
  For example: for addition of two numbers, they are brought into the processor, and the actual addition is carried out of the ALU.

- Any other arithmetic or logic operation, like multiplication, division is initiated by bringing the required operands into the processor, where the operation is performed by the ALU.
- The control and the arithmetic and logic units are many times faster than the other devices connected to a computer system. This enables a single processor to control a number of external devices such as keyboards, displays, magnetic and optical disks.

1.2.4 Output Unit

The output unit is the counterpart of the input unit. Its function is to send processed results to the outside world. For example: printers.

1.2.5 Control Unit

- The memory, arithmetic and logic, and input and output units store and process information and perform input and output operations. The control unit is effectively the centre that sends control signals to other units and senses their states.

The operation of a computer:

- The computer accepts information in the form of programs and data through an input unit and stores it in the memory.
- Information stored in the memory is fetched, under program control, into an arithmetic and logic unit, where it is processed.
- Processed information is output through a output unit and all activities inside the machine is directed by the control unit.

1.3 NUMBERS & ARITHMETIC OPERATIONS

Computers are built using logic circuits that operate on information represented by two values as 0 and 1 and we define the amount of information as a bit information.

1.3.1 Number Representation

Consider an n-bit vector
$C = C_{n-1}\ldots\ldots\ldots\ldots C_1C_0$
Where C_i = 0 or 1 for $0 \le i \le n-1$. This vector can represent unsigned integer values V in the range 0 to 2^n-1, where
$V(C) = C_{n-1} \times 2^{n-1} + \ldots\ldots + c_1 \times 2^1 + c_0 \times 2^0$
Three systems are used for representing the positive and negative numbers:

1. Sign and Magnitude

- The leftmost bit is 0 for positive numbers and 1 for negative numbers.
- In this, negative values are represented by changing the most significant bit from 0 to 1 in the vector C of the corresponding positive value.
- For example:
  $+5 \Rightarrow 0101$
  $-5 \Rightarrow 1101$

2. 1's Complement

- The leftmost bit is 0 for positive numbers and 1 for negative numbers
- Negative values are obtained by complementing each bit of the corresponding positive number.

For example:
For -3 we can find by complementing each bit in the vector 0011 to yield 1100.

- Same operation is used for converting a negative number to the corresponding positive value. The operation of forming the 1's complement of a given number is equivalent to subtracting that number from 2^n-1.

3. 2's Complement

- The leftmost bit is 0 for positive numbers and 1 for negative numbers.
- In this, forming the 2's complement of a number is done by subtracting that number from 2^n.
  Hence, the 2's complement of a number is obtained by adding 1 to the 1's complement of that number.

1.3.2 Arithmetic Addition

1. In Signed-Magnitude form

- Follows the rules of ordinary arithmetic
  If the signs are same ⇒ add two magnitudes and give the sum common sign.
  If the signs are different ⇒ subtract smaller magnitude from the larger and give the result, the sign of the larger magnitude.
  For Example:
  (+35) + (-37) = -(37-25) = -2

2. In 2's complement form

- The system does not require a comparison or subtraction only addition and complementation is necessary.
- The procedure is as follows: Add the two numbers including their sign bits and discard any carry out of the sign (left most) bit position.

Note: The negative number must initially be in 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

For Example:

6	00000110
+13	00001101
+19	00010011
−6	11111010
+13	00001101
+17	00000111

1.3.3 Arithmetic Subtraction

- Subtraction of two signed numbers, when negative numbers are in 2's complement form, subtraction is very simple and can be done as follows:
  → Take the 2's complement of the subtrahend (including the sign bit)
  → Add it to the minuend (including the sign bit)
  → A carryout of the sign bit position is discarded.
- Changing a positive number to a negative number is easily done by taking its 2's complement and vice-versa is also true.
  For example : (-6) – (-13) = +7
  In binary format, it is written as 11111010 – 11110011

The subtraction is changed to addition by taking the 2's complement of the subtrahend (-13) to give (+13).
In binary format this is
11111010 + 00001101 = 100000111
and removing the end carry, we obtain the answer as 00000111 ⇒ (+7).

1.3.4 Overflow in Integer Arithmetic

- In the 2's complement number representation system, n-bits can represent values in the range -2^{n-1} to $+2^{n-1}-1$.
  When the result of an arithmetic operation is outside the represent able range, an arithmetic overflow has occurred.
- While adding unsigned numbers, the carry out from the most significant bit position serves as the overflow indicator. This is not applicable for adding signed numbers.

For example:
By using 4-bit signed numbers, if we try to add the numbers +7 and +4, the output is 1011 ⇒ -5 ⇒ Incorrect result (with carry-out =0)

Note: Overflow may occur if both summands have the same sign. The addition of numbers with different signs cannot cause overflow.

❖ A single method to detect overflow is to examine the signs of two summands X and Y and the sign of the result. When both operands X and Y have the same sign, an overflow occurs when the sign of S is not the same as the signs of X & Y.

1.4 DECIMALFIXED-POINT REPRESENT-ATION

- The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit. A 4-bit decimal code requires four flip flops for each decimal digit.

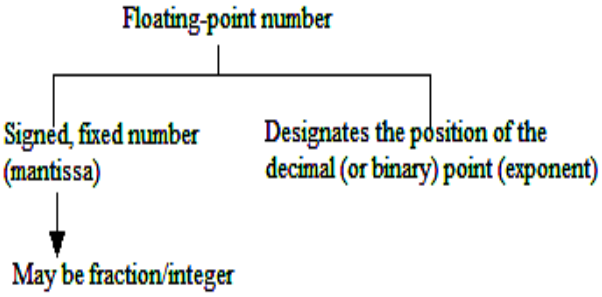
Disadvantages

- By representing numbers in decimal we are wasting amount of storage space since the number of bits needed to store a decimal number in a binary code is greater than the number of bits needed of its equivalent binary representation.
- The circuits required to perform decimal arithmetic are more complex.

Advantage

- In applications like business data processing we require small amounts of arithmetic computations (in decimal format).
- The representation of signed numbers in binary is similar to the representation of signed decimal numbers in BCD. The sign of a decimal number is usually represented with four bits to confirm with the 4-bit code of the decimal digits.
- The signed-magnitude system is difficult to use with computers. The signed complement system can be either the 9's or the 10's complement is the one most commonly used. To obtain the 10's complement of a BCD number, we first take the 9's complement and then add one to the least significant digit. The 9's complement is calculated from the subtraction of each digit from 9.
- The subtraction of decimal numbers is either unsigned or in the signed-10's complement system. Take the 10's complement of the subtrahend and add it to the minuend.

1.5 FLOATING POINT REPRESENTATION



Example :
+ 6132.789
Fraction: +0.6132789
Exponents: +0.4

- Floating point is always interpreted to represent a number in the following form $m \times r^e$ m and e are physically represented in the register (including the signs). The radix r and the radix-point position of the mantissa are always assumed.
- A floating point binary number is represented in a similar manner except that it uses base-2 for exponent.

For example: The binary number + 1001.11 is represented with 8 bit fraction and 6 bit exponent as follows.

Fraction	Exponent
01001110	000100

- A floating point number is said to be normalized if the most significant digit of the mantissa is nonzero.
  For example:
  The decimal number 250 is normalized but 00035 is not.
  Regardless of where the position of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero.
  The number can be normalized by shifting three positions to the left and discarding the leading 0's to obtain 11010000. Normalized numbers provide the maximum possible precision for the floating point number. A zero cannot be normalized in floating point by all 0's in the mantissa and exponent.

1.6 SIGNED-OPERAND MULTIPLICATION

The multiplication of signed operands generates a double-length product in the 2's complement number system. In general, accumulate partial products by adding versions of the multiplicand as selected by the multiplier bits.

Case (i):

Positive multiplier and negative multiplicand.

- When we add a negative multiplicand to a partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend.

For example:

The 5 bit signed operand, -13 is the multiplicand and it is multiplied by +11, to get the product as -143.

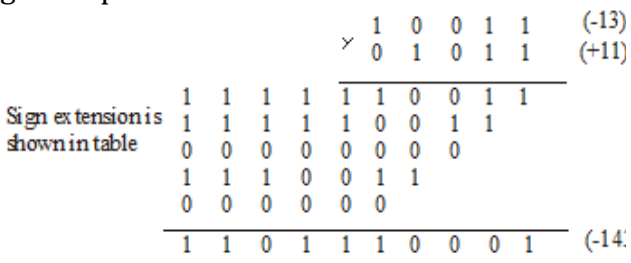


Fig. :Sign extension of negative multiplicand

Note : For a negative multiplier, a solution is to form the 2's complement of both the multiplier and multiplicand and proceed as in the case of a positive multiplier. A algorithm called the Booth's algorithm works equally well for both negative and positive multipliers.

1.7 BOOTH'S ALGORITHM

The Booth's algorithm generates a 2n-bit product and treats both positive and negative numbers uniformly. A powerful algorithm for signed number multiplication, the Booth's algorithm generates a 2n-bit product and treats both positive and negative numbers uniformly. Consider a multiplication operation in which the multiplier is positive and has a single block of 1s, for example, 0011110. To derive the product, we could add four appropriately shifted versions of the multiplicand, as in the standard procedure. However, we can reduce the number of required operations by regarding this multiplier as the difference between the two numbers:

```
      0100000   (32)
-     0000010   (2)
      -------
      0011110   (30)
```

This suggests that the product can be generated by adding 2^5 times the multiplicand to the 2's complement of 2^1 times the multiplicand. For convenience, we can describe the sequence or required operations by recoding the preceding multiplier as 0+1000-10.

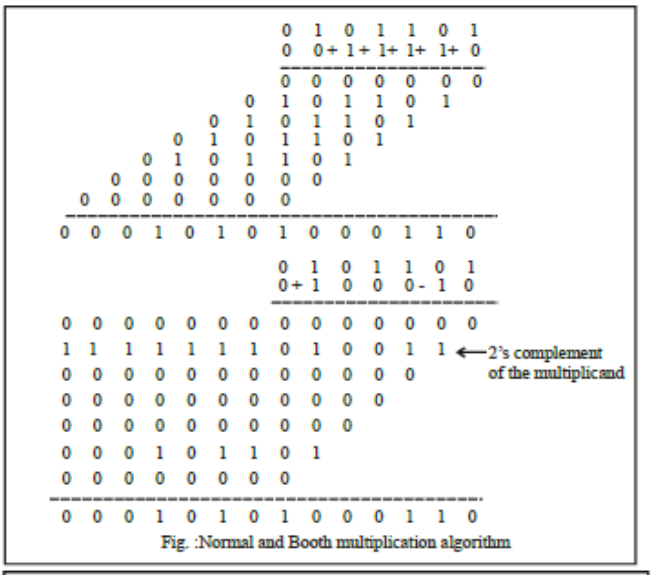


Fig. :Normal and Booth multiplication algorithm

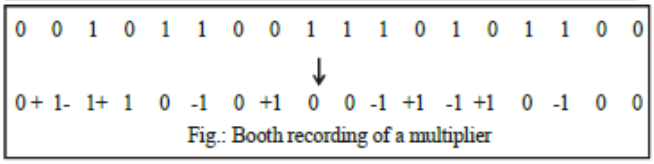


Fig.: Booth recording of a multiplier

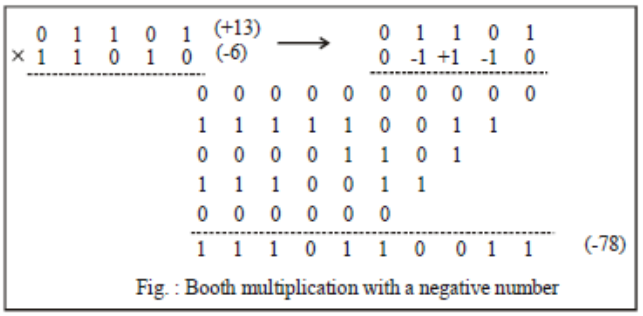


Fig. : Booth multiplication with a negative number

- In general, in Booth's scheme, -1 time the shifted multiplicand is selected when moving from 0 to 1, and +1 time the shifted multiplicand is selected when moving from 1 to 0, as the

multiplier is scanned from right to left. Figure 9 illustrates the normal and the Booth's algorithms for the example just discussed. The Booth's algorithm clearly extends to any number of blocks of 1s in a multiplier, including the situation in which a single 1 is consider a block see figure 10 for another example of recoding the multiplier. In this example, the least significant bit is 1. This situation is uniformly handled by assuming that an implied 0 lies to its right.

- The Booth's algorithm can also be used for negative multiplier, as figure shows. To see the correctness of this technique in general, we use a property of negative number representations in the 2's complement system. Let the leftmost zero of a negative number, X, be at a bit position k, that is

  $X = 11.....10x_{k-1}....x_0$

  The value of X is given by

  $V(X) = -2^{k+1} + x_{k-1}*2^{k-1} + x_0 *2^0$

  This is supported by observing that

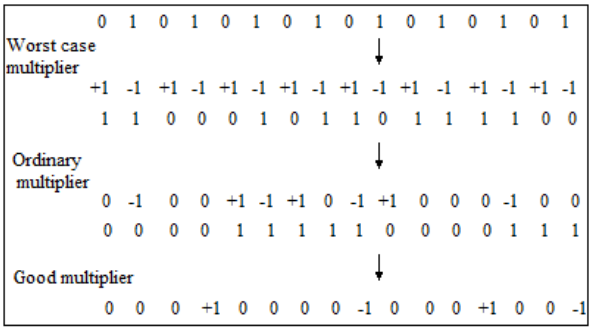
  $11.....100 \quad ....0$

  $+ 00.....00x_{k-1} \quad ....x_0$

  $X = 11.....10x_{k-1} \quad .....x_0$

Table: Booth multiplier recording table

Multiplier		Version of multiplicand by bit I
Bit I	Bit -1	
0	0	0 × M
0	1	+1 × M
1	0	-1 × M
1	1	0 × M



- The top number is the 2's complement representation of -2^{k+1}. The recoded multiplier now consists of the part corresponding to the second number, with -1 added in position k+1. For example, the multiplier 110110 becomes 0-1 +10-10.
- The Booth's technique for recoding multipliers is summarized in above table. The transformation 011...110 $\Rightarrow$ +100.....0 -10 is called skipping over 1s. This term is derived from the case in which the multiplier has its 1s grouped into a few contiguous block; only a few versions of the multiplicand, that is, the summands, must be added to generate the product, thus speeding up the multiplication operation. However, in the worst case that of alternating 1s and 0s in the multiplier-each bit of the multiplier selects a summand. In fact, this results in more summands than if the Booth algorithm were not used. A 16-bit, worst-case multiplier, an ordinary multiplier, and a good multiplier are shown in figure 12.
- The Booth's algorithm has three attractive features
  1. It handles both positive and negative multipliers uniformly.
  2. Second, it achieves some efficiency in the number of additions required when the multiplier has a few large blocks of 1s.
  3. The speed gained by skipping over 1s depends on the data. On average, the speed of doing multiplication with the Booth's algorithm is the same as with the normal algorithm.

1.8 INTEGER DIVISION

Decimal division and the binary-coded division of the same value

A circuit that implements division by longhand method operates as follows:

→ It positions the divisor appropriately with respect to the dividend and performs a subtraction.

→ If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.

→ If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, is repositioned for another subtraction.

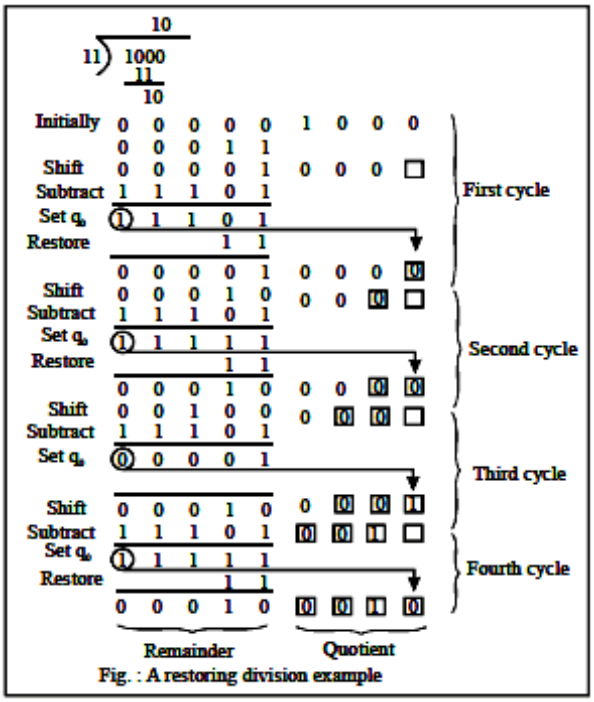


Fig. : A restoring division example

Do the following n times:

→ Shift A and Q left one binary position

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

→ If the sign of A is 1, set q_0 to 0 & add M back to A (i.e. restore A) otherwise, set q_0 to 1.

- This algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction.
- Consider the sequence of operations that takes place after the subtraction operation in the preceding algorithm.
- If A is positive ⇒Shift left and subtract M i.e. we performs 2A-M
- If A is negative⇒We restore it by performing A+M and then we shift it left and subtract M.
- This is equivalent to performing 2A+ M
- The q_0 bit is appropriately set to 0 or 1 after the correct operation has been performed.

1.9 NON-RESTORING-DIVISION ALGORITHM

Step1 :

Do the following n times.

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

Step 2 :

If the sign of A is 1,add M to A, step 2 is needed to leave the proper positive remainder is A at the end of n cycles.

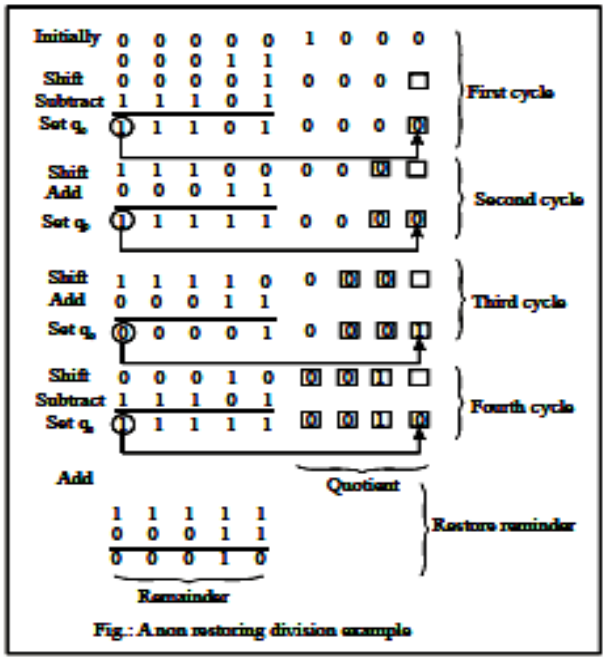


Fig. : A non restoring division example

Note : The restore operations are no longer needed and that exactly one add or subtract operation is performed per cycle.

1.10 FLOATING-POINT NUMBERS AND OPERATIONS

- Up to this, we have deal only with fixed-point numbers and have considered them as integers, that is, as having an implied binary point at the right end of the number. By assuming the binary fraction point is just to the right of the sign bit, thus representing a fraction. In

the 2's complement system, the signed value F, represented by the n-bit binary fraction.
$C = C_0C_{-1}C_{-2}\ldots\ldots b_{-(n-1)}$ is given by
$F(c) = -C_0 \times 2^0 + C_{-1}\times 2^{-1}+C_{-2}\times 2^{-2}+\ldots..+C_{-(n-1)} \times 2^{-(n-1)}$
Where the range of F is,
$-1 \le F \le 1-2^{-(n-1)}$

- Consider the range of values represent able in a 32-bit, signed, fixed-point format. Interpreted as integers, the value range is approximately 0 to $\pm 2.15 \times 10^{-9}$. If we consider them to be fraction, the range is approximately $\pm 4.55\times 10^{-10}$ to ± 1.
- Hence, we need to accommodate both very large integers and very small fractions. To do this, a computer must be able to represent numbers and operate on then in such a way that the position of the binary point is variable and is automatically adjusted as computation proceeds.
  Such a representation is called as floating point representation.
- Due to the position of binary or floating point in a number is variable and it must be given strictly in the floating point representation.

❖ By convention, when the decimal point is placed to the right of the first (nonzero) significant digit, the number is said to be normalized.
Thus, floating point number representation is number representation in which a number is represented by its sign, a string of significant digits, known as mantissa and an exponent to an implied base for the scale factor.

1.10.1 IEEE STANDARD FOR FLOATING-POINT NUMBERS

- A general form is
  $\pm X_1.X_2X_3X_4X_5X_6X_7 \times 10^{\pm Y1Y2}$
  Where X_i and Y_i are decimal digits.

(i) The number of significant digits (7)

(ii) The exponent range (± 99) are sufficient for a wide range of calculations. It is possible to approximate this mantissa precision and scale factor range in a binary representation that occupies 32 bits. A 24-bit mantissa can approximately represent a 7-digit decimal number, and an 8-bit exponent to an implied base of 2 provides a scale factor with a reasonable range. One bit is needed for the sign of the number. Because the leading nonzero bit of a normalized binary mantissa must be a 1, it does not have to be included explicitly in the representation. Thus, total of 32-bits is needed.

- The standard explained above for representing floating-point numbers in 32-bits has been developed and specified in detail by the Institute of Electrical and Electronics Engineers (IEEE). This standard describes both the representation and the way in which the four basic arithmetic operations are to be preformed.
- The 32-bit representation is given in figure below

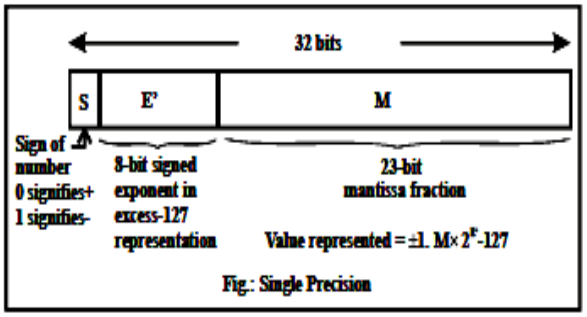


Fig.: Single Precision

→ The sign of the number is given in the first bit
→ Followed by a representation for the exponent (to the base 2) of the scale factor.

- Instead of the signed exponent, E, the value actually stored in the exponent field is an unsigned integer E' = E + 127. This is called the excess -127 format. Thus E' is in the range $0 \le E' \le 225$. The end values of the range, 0 and 255 are used to represent special values. Therefore, the range of E' for

normal values is $1 \le E' \le 254$, that means the actual exponent, E, is in the range $-126 \le E \le 127$.

- The last 23 bits represent the mantissa. The most significant bit of the mantissa is always equal to 1 because binary normalization is used. This bit is not explicitly represented; it is assumed to be the immediate left of the binary point.
  Hence, the 23-bits stored in M field actually represent the fractional part of the mantissa, i.e. the bits to the right of the binary point. The following figure shows an example of a single precision floating-point.

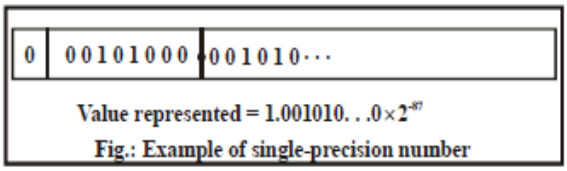


Fig.: Example of single-precision number

- The 32-bit standard representation in figure is called a single-precision representation because it occupies a single 32-bit word.

→ Scale factor has a range or 2^{-126} to $2^{+127} \cong 10^{\pm 38}$

→ The 24-bit mantissa provides ≅ same precision as a 7-digit decimal value.

- To provide more precision and range for floating-point numbers, the IEEE standard.
  Also specifies a double-precision format as shown in figure below.

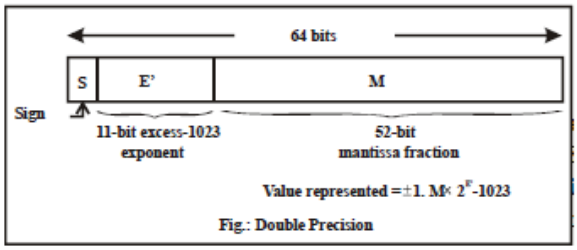


Fig.: Double Precision

→ The double precision format has increased exponent and mantissa range.

→ The 11-bit excess-1023exponent E' has the range $1 \le E' \le 2046$ for normal values, with special values.

→ Actual exponent E is the range $-1022 \le E \le 1023$, providing scale factors of 2^{-1022} to 2^{1023}. The 53-bit mantissa provides a precision equivalent to about 16 decimal digits.

→ The double precision format hasincreased exponent and mantissa range.

→ The 11-bit excess-1023exponent E' has the range $1 \le E' \le 2046$ for normal values, with 0 and 2047 used to indicate special values.

→ Actual exponent E is the range $-1022 \le E \le 1023$, providing scale factors of 2^{-1022} to 2^{1023}. The 53-bit mantissa provides a precision equivalent to about 16 decimal digits.

1.10.2 Basic aspects of operating with floating-point numbers:

- If a number is not normalized, then put in normalized form by shifting the fraction and adjusting the exponent.
  The following figure shows an normalized value $0.0010110\ldots \times 2^9$ and its normalized version, $1.0110\ldots.. \times 2^6$.

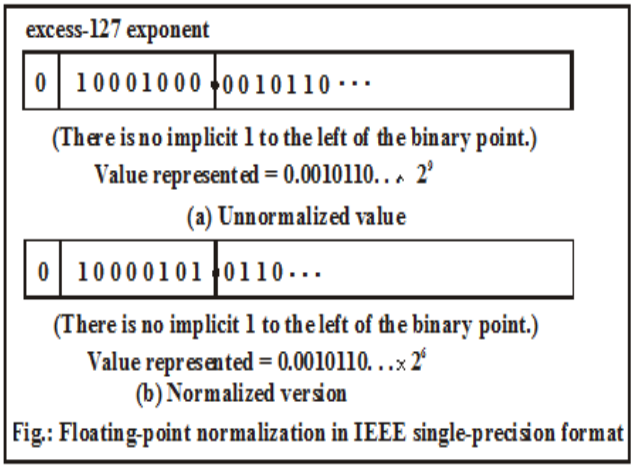


Fig.: Floating-point normalization in IEEE single-precision format

Since, the scale factor is in the form 2^i, shifting the mantissa right or left by one bit position is adjusted by an increase or a decrease of 1 in the exponent.

- As computations proceed, a number that does not fall in the representable range of normal numbers might be generated. In single precision, this means that its normalized representation requires an exponent less than-126 or greater than +127.

1.10.3 Arithmetic operations on floating-point numbers:

The rules for addition and subtraction can be stated as follows:
Add/Subtract Rule

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2. Set the exponent of the result equal to the larger exponent.
3. Perform addition/subtraction on the mantissas and determine the sign of the result.
4. Normalize the resulting value, if necessary. Multiplication and division are somewhat easier than addition and subtraction, in that no alignment of mantissas is needed. Multiply Rule
   1. Add the exponents and subtract 127.
   2. Multiply the mantissas and determine the sign of the result.
   3. Normalize the resulting value, if necessary.

Parameter	Format			
	Single	Single Extended	Double	Double Extended
Word width (bits)	32	≥43	64	≥79
Exponent width (bits)	8	≥11	11	≥15
Exponent bias	127	Unspecified	1023	Unspecified
Maximum exponent	127	≥1023	1023	≥16383
Minimum exponent	-126	≤-1022	-1022	≤-16382
Number range (base 10)	10^{38},10^{+38}	Unspecified	10^{308},10^{+308}	Unspecified
Significant width (bits)	23	≥31	52	≥63
Number of exponents	254	Unspecified	2046	Unspecified
Number of fractions	223	Unspecified	252	Unspecified
Number of values	1.98×2^{31}	Unspecified	1.99×2^{63}	Unspecified

1.10.4 Divide Rule

1. Subtract the exponents and add 127
2. Divide the mantissas and determine the sign of the result.
3. Normalize the resulting value, if necessary.
   The addition or subtraction of 127 in the multiply and divide rules results from using the excess-127 notation for exponents.

1.10.5 IEEE 754 Format Parameters

1.10.6 Precision Consideration

Prior to a floating point operation, the exponent and significant of each operand are loaded into the ALU registers.
In case of significant $\Rightarrow$ the length of the register is almost always greater than the length of the significant plus and implied bit. The register contains additional bits, called guard bits, which are used to pad out the right end of the significant with 0's

Example

$X = 1.000.....00 \times 2^{1}$ $-\ Y = 0.111.....11 \times 2^{1}$ $Z = 0.000.....01 \times 2^{1}$ $= 1.000.....00 \times 2^{-22}$	$X = 1.000.....00\ \ 0000 \times 2^{1}$ $-\ Y = 0.111.....11\ \ 1111 \times 2^{1}$ $Z = 0.000.....01\ \ 1000 \times 2^{1}$ $= 1.000.....00\ \ 0000 \times 2^{-23}$
Fig.: Binary example, without guard bits	Fig.: Binary example, with guard bits

1.10.7 Rounding

A number of techniques have been explored for performing rounding.

1	Round to nearest	The result is rounded to the nearest represent able number
2	Round toward + ∞	The result is rounded up toward plus infinity
3	Round toward - ∞	The result is rounded up down toward negative infinity
4	Round toward 0	The result is rounded toward zero

Note: Round to nearest is the default rounding mode listed in the standard.

1.10.8 IEEE STANDARD FOR BINARY FLOATING-POINT ARITHMETIC

- Infinity:
  Infinity arithmetic is treated as the limiting case of real arithmetic, with the infinity values given the following interpretation.
  $-\infty <$ (every finite number) $< +\infty$
  For example
  $6 + (+\infty) = +\infty$
  $6 / (+\infty) = +0$
  $6 \times (+\infty) = +\infty$

$(+\infty)-(-\infty)=+\infty$

- Quiet and Signaling NaNs
  A Nan is a symbolic entity encoded in floating-point format, of which there are two types:
  (i) Signaling
  (ii) Quiet

  (i) Signaling
  A signaling Nan signals an invalid operation exception whenever it appears as an operand.
  Signaling Nan's affords values for uninitialized variables and arithmetic like enhancement that are not the subject of the standard.
  (ii) Quiet
  A quiet Nan's propagates through almost every arithmetic operation without signaling an exception

  Note : Both types of Nan's have the same general format: an exponent of all ones and nonzero fraction. The actual bit pattern of the nonzero fraction is implementations dependent; the fraction values can be used distinguish quiet Nan's from signaling Nan's and to specify particular exception conditions.

1.10.9 Table: Operations that Produce a Quiet NaN

Operation	Quiet NaN Produced by
Any	Any operation on a signaling NaN
Add or subtract	Magnitude subtraction of infinities: $(-\infty)+(-\infty)$ $(-\infty)+(+\infty)$ $(+\infty)-(+\infty)$ $(-\infty)-(-\infty)$
Multiply	$0\times\infty$
Division	$\frac{0}{0}$ or $\frac{\infty}{\infty}$
Remainder	x REM 0 or ∞ REM y
Square root	$\sqrt{x}$ where x < 0

1.10.10 Demoralized Numbers

Renormalized numbers are useful for exponent under flow, therefore they are included in IEEE 754.

- ❖ When the exponent of the result becomes too small (a negative exponent with too large a magnitude), the result is demoralized by right shifting the fraction and incrementing the exponent for each shift, until the exponent is within a represent able range.
- ❖ The above figure explains the effect of the addition of renormalized numbers. The represent able numbers can be grouped into intervals of the form $[2^n, 2^{n+1}]$
- ❖ Within each such interval, the exponent portion of the number remains constant while the fraction varies, producing a uniform spacing of represent able numbers within the interval.
- ❖ As approaches towards zero, each successive interval is half number of representable numbers. Hence, the density of representable numbers increases as we approach zero.
- ❖ If only normalized numbers are used, there is a gap between the smallest normalized number and 0. In case of 32-bit IEEE 754 format, there are 2^{23} represent able numbers in each interval, and the smallest represent able positive number is 2^{-126}. With the addition of demoralized numbers, an additional 2^{23} number are uniformly added between o and 2^{-126}
- ❖ Without demoralized numbers, the gap between the smallest representable nonzero number and zero is much wider than the gap between the smallest representable nonzero number and the next larger number.
- ❖ In case of demoralized numbers is referred to as gradual underflow. Gradual underflow fills in the gap and reduces the impact of exponent underflow to a level comparable with round off among the normalized numbers.

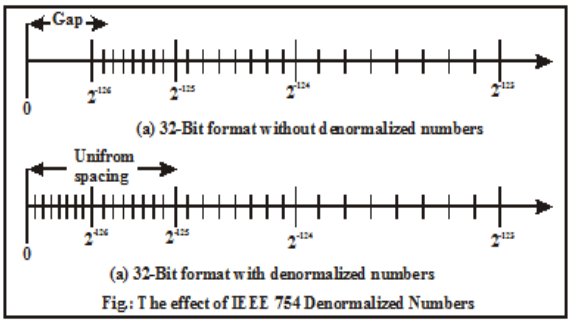


Fig.: The effect of IEEE 754 Denormalized Numbers

					1	0	0	1	1	(–13)
					0	1	0	1	1	(+11)
1	1	1	1	1	1	0	0	1	1	
1	1	1	1	1	0	0	1	1		
0	0	0	0	0	0	0	0			
1	1	1	0	0	1	1				
0	0	0	0	0	0					
1	1	0	1	1	1					(–143)