EE 1193

INTRODUCTION TO HDL

MODULE-2: DATA REPRESENTATION

Dr. Amit Acharyya, IITH

Overview

- In the early days of computing, there were common misconceptions about computers.
- One misconception was that the computer was only a giant adding machine performing arithmetic operations. Computers could do much more than that, even in the early days.
- The other common misconception, in contradiction to the first, was that the computer could do "anything." We now know that there are indeed classes of problems that even the most powerful imaginable computer finds intractable with the von Neumann model.
- The correct perception, of course, is somewhere between the two.
- We are familiar with computer operations that are non-arithmetic: computer graphics, digital audio, even the manipulation of the computer mouse. Regardless of what kind of information is being manipulated by the computer, the information must be represented by patterns of 1's and 0's (also known as "on-off " codes).
- This immediately raises the question of how that information should be described or represented in the machine—this is the **data representation,** or **data encoding. Graphical images, digital audio, or mouse clicks must all** be encoded in a systematic, agreed-upon manner.

- We might think of the decimal representation of information as the most natural when we know it the best, but the use of on-off codes to represent information predated the computer by many years, in the form of Morse code.

- This module introduces several of the simplest and most important encodings:

- the encoding of signed and unsigned fixed point numbers, real numbers (referred to as **floating point numbers in computer jargon), and the printing characters.**

- We shall see that in all cases there are multiple ways of encoding a given kind of data, some useful in one context, some in another.

- We will also take an early look at computer arithmetic for the purpose of understanding some of the encoding schemes, though we will defer details of computer arithmetic until Module 3.

- In the process of developing a data representation for computing, a crucial issue is deciding how much storage should be devoted to each data value.
- For example, a computer architect may decide to treat integers as being 32 bits in size, and to implement an ALU that supports arithmetic operations on those 32-bit values that return 32 bit results.
- Some numbers can be too large to represent using 32 bits, however, and in other cases, the operands may fit into 32 bits, but the result of a computation will not, creating an **overflow condition, which is described in Module** 3.
- Thus we need to understand the limits imposed on the accuracy and range of numeric calculations by the finite nature of the data representations.
- We will investigate these limits in the next few sections.

Fixed Point Numbers

- In a fixed point number system, each number has exactly the same number of digits, and the "point" is always in the same place.

- Examples from the decimal number system would be 0.23, 5.12, and 9.11. In these examples each number has 3 digits, and the decimal point is located two places from the right. Examples from the **binary number system (in which each digit can take on only one of the** values: 0 or 1) would be 11.10, 01.10, and 00.11, where there are 4 binary digits and the binary point is in the middle.

- An important difference between the way that we represent fixed point numbers on paper and the way that we represent them in the computer is that when fixed point numbers are represented in the computer *the binary point is not stored anywhere, but only assumed to be in a certain* position. One could say that the binary point exists only in the mind of the programmer.

- We start fixed point numbers by investigating the range and precision of fixed point numbers, using the decimal number system. We then take a look at the nature of number bases, such as decimal and binary, and how to convert between the bases.

- With this foundation, we then investigate several ways of representing negative fixed point numbers, and take a look at simple arithmetic operations that can be performed on them.

RANGE AND PRECISION IN FIXED POINT NUMBERS

- A fixed point representation can be characterized by the **range of expressible** numbers (that is, the distance between the largest and smallest numbers) and the **precision (the distance between two adjacent numbers on a number line.)**

- **For** the fixed-point decimal example above, using three digits and the decimal point placed two digits from the right, the range is from 0.00 to 9.99 inclusive of the endpoints, denoted as [0.00, 9.99], the precision is .01, and the **error is 1/2 of** the difference between two "adjoining" numbers, such as 5.01 and 5.02, which have a difference of .01. The error is thus .01/2 = .005.

- That is, we can represent any number within the range 0.00 to 9.99 to within .005 of its true or precise value.

- Notice how range and precision trade off: with the decimal point on the far right, the range is [000, 999] and the precision is 1.0.
- With the decimal point at the far left, the range is [.000, .999] and the precision is .001.
- In either case, there are only 10^3 different decimal "objects," ranging from 000 to 999 or from .000 to .999, and thus it is possible to represent only 1,000 different items, regardless of how we assign range and precision.
- There is no reason why the range must begin with 0. A 2-digit decimal number can have a range of [00,99] or a range of [-50, +49], or even a range of [-99, +0].
- The representation of negative numbers is covered more fully in later section in this module.
- Range and precision are important issues in computer architecture because both are finite in the implementation of the architecture, but are infinite in the real world, and so the user must be aware of the limitations of trying to represent external information in internal form.

THE ASSOCIATIVE LAW OF ALGEBRA DOES NOT ALWAYS HOLD IN COMPUTERS

- In early mathematics, we learned the associative law of algebra:

$$a + (b + c) = (a + b) + c$$

- As we will see, the associative law of algebra does not hold for fixed point numbers having a finite representation.

- Consider a 1-digit decimal fixed point representation with the decimal point on the right, and a range of [-9, 9], with a = 7, b=4, and c=–3.

- Now $a + (b + c) = 7 + (4 + -3) = 7 + 1 = 8$. But $(a + b) + c = (7 + 4) + -3 = 11 + -3$, but 11 is outside the range of our number system!

- We have overflow in an intermediate calculation, but the final result is within the number system.

- This is bad because the final result will be wrong if an inter mediate result is wrong.

- Thus we can see by example that the associative law of algebra does not hold for finite-length fixed point numbers.

- This is an unavoidable consequence of this form of representation, and there is nothing practical to be done except to detect overflow wherever it occurs, and either terminate the computation immediately and notify the user of the condition, or, having detected the overflow, repeat the computation with numbers of greater range. (The latter technique is seldom used except in critical applications.)

: IMPORTANT FOR EMBEDDED SYSTEMS AND DIGITAL IC DESIGN

RADIX NUMBER SYSTEMS

- Here, we will see how to work with numbers having arbitrary bases, although we will focus on the bases most used in digital computers, such as base 2 (binary), and its close cousins base 8 (octal), and base 16 (hexadecimal.)

- The **base, or radix of a number system defines the range of possible values that a** digit may have.

- In the base 10 (decimal) number system, one of the 10 values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 is used for each digit of a number.

- The most natural system for representing numbers in a computer is base 2, in which data is represented as a collection of 1's and 0's.

The general form for determining the decimal value of a number in a radix k fixed point number system is shown below:

$$Value = \sum_{i=-m}^{n-1} b_i \cdot k^i$$

The value of the digit in position i is given by b_i. There are n digits to the left of the radix point and there are m digits to the right of the radix point. This form of a number, in which each position has an assigned weight, is referred to as a **weighted position code**. Consider evaluating $(541.25)_{10}$, in which the subscript 10 represents the base. We have $n = 3$, $m = 2$, and $k = 10$:

$$5 \times 10^2 + 4 \times 10^1 + 1 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2} =$$

$$(500)_{10} + (40)_{10} + (1)_{10} + (2/10)_{10} + (5/100)_{10} \quad = (541.25)_{10}$$

Now consider the base 2 number $(1010.01)_2$ in which $n = 4$, $m = 2$, and $k = 2$:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} \quad =$$

$$(8)_{10} + (0)_{10} + (2)_{10} + (0)_{10} + (0/2)_{10} + (1/4)_{10} \quad = (10.25)_{10}$$

- This suggests how to convert a number from an arbitrary base into a base 10 number using the **polynomial method.**

- **The idea is to multiply each digit by the** weight assigned to its position (powers of two in this example) and then sum up the terms to obtain the converted number.

- Although conversions can be made among all of the bases in this way, some bases pose special problems, as we will see in the next section.

Note: in these weighted number systems we define the bit that carries the most weight as the **most significant bit (MSB), and the bit that carries the least** weight as the **least significant bit (LSB). Conventionally the MSB is the leftmost** bit and the LSB the rightmost bit.

CONVERSIONS AMONG RADICES

- In the previous section, we saw an example of how a base 2 number can be converted into a base 10 number.

- A conversion in the reverse direction is more involved. The easiest way to convert fixed point numbers containing both integer and fractional parts is to convert each part separately. Consider converting $(23.375)_{10}$ to base 2.

- We begin by separating the number into its integer and fractional parts:

$$(23.375)_{10} = (23)_{10} + (.375)_{10}.$$

Converting the Integer Part of a Fixed Point Number—The Remainder Method

- As suggested in the previous section, the general polynomial form for representing a binary integer is:

$$b_i \times 2^i + b_{i-1} \times 2^{i-1} + \ldots + b_1 \times 2^1 + b_0 \times 2^0$$

If we divide the integer by 2, then we will obtain:

$$b_i \times 2^{i-1} + b_{i-1} \times 2^{i-2} + \ldots + b_1 \times 2^0$$

with a remainder of *b0*.

- *As a result of dividing the original integer by 2, we discover* the value of the first binary coefficient *b0*.
- *We can repeat this process on the* remaining polynomial and determine the value of b_1.
- *We can continue iterating* the process on the remaining polynomial and thus obtain all of the b_i.
- *This process* forms the basis of the **remainder method of converting integers between** bases.

- We now apply the remainder method to convert $(23)_{10}$ to base 2. As shown in Figure beside, the integer is initially divided by 2, which leaves a remainder of 0 or 1.
- For this case, 23/2 produces a quotient of 11 and a remainder of 1. The first remainder is the least significant **binary digit (bit) of the converted number (**the rightmost bit).
- In the next step 11 is divided by 2, which creates a quotient of 5 and a remainder of 1.
- Next, 5 is divided by 2, which creates a quotient of 2 and a remainder of 1.
- The process continues until we are left with a quotient of 0. If we continue the process after obtaining a quotient of 0, we will only obtain 0's for the quotient and remainder, which will not change the value of the converted number.
- The remainders are collected into a base 2 number in the order shown in the Figure beside to produce the result $(23)_{10} = (10111)_2$.

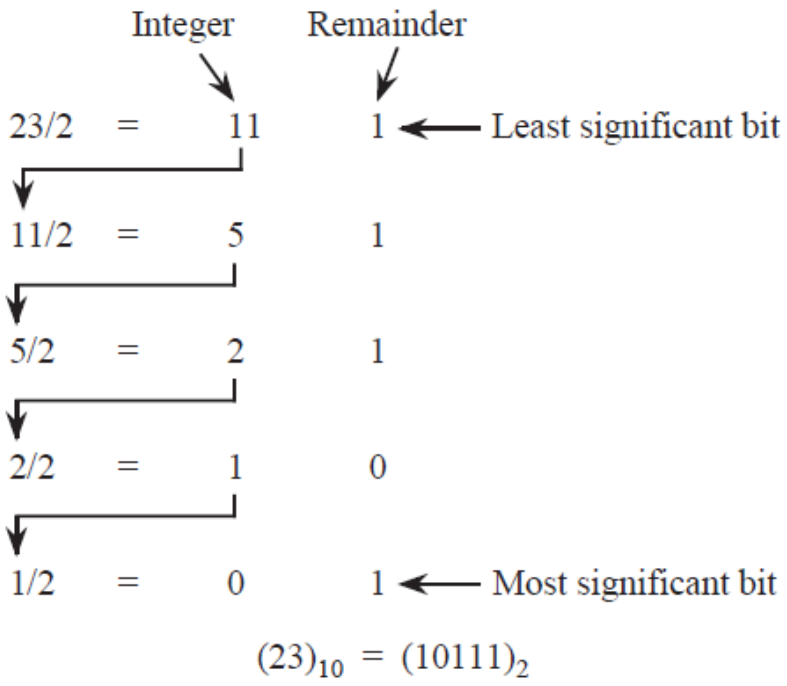


Fig: A conversion from a base 10 integer to a base 2 integer using the remainder method.

- In general, we can convert any base 10 integer to any other base k by simply dividing the integer by the base k to which we are converting.

- We can check the result by converting it from base 2 back to base 10 using the polynomial method:

$$(10111)_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

$$= 16 + 0 + 4 + 2 + 1$$

$$= (23)_{10}$$

At this point, we have converted the integer portion of $(23.375)_{10}$ into base 2.

Converting the Fractional Part of a Fixed Point Number—The Multiplication Method

- The conversion of the fractional portion can be accomplished by successively multiplying the fraction by 2 as described below.
- A binary fraction is represented in the general form: $b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + b_{-3} \times 2^{-3} + \ldots$
  - If we multiply the fraction by 2, then we will obtain: $b_{-1} + b_{-2} \times 2^{-1} + b_{-3} \times 2^{-2} + \ldots$

• We thus discover the coefficient b_{-1}. *If we iterate this process on the remaining* fraction, then we will obtain successive b_i. *This process forms the basis of the* **multiplication method of converting fractions between bases.**

• For the example used here (Figure), the initial fraction $(.375)_{10}$ is less than 1. If we multiply it by 2, then the resulting number will be less than 2. The digit to the left of the radix point will then be 0 or 1. This is the first digit to the right of the radix point in the converted base 2 number, as shown in the figure. We repeat the process on the fractional portion until we are either left with a fraction of 0, at which point only trailing 0's are created by additional iterations, or we have reached the limit of precision used in our representation. The digits are collected and the result is obtained: $(.375)_{10} = (.011)_2$.

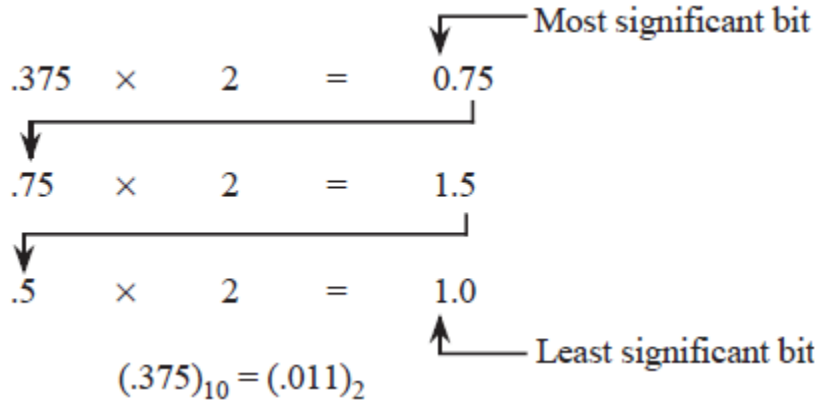


Fig: A conversion from a base 10 fraction to a base 2 fraction using the multiplication method.

- For this process, the multiplier is the same as the target base. The multiplier is 2 here, but if we wanted to make a conversion to another base, such as 3, then we would use a multiplier of 3.
- We again check the result of the conversion by converting from base 2 back to base 10 using the polynomial method as shown below:

$$(.011)_2 = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 0 + 1/4 + 1/8 = (.375)_{10}.$$

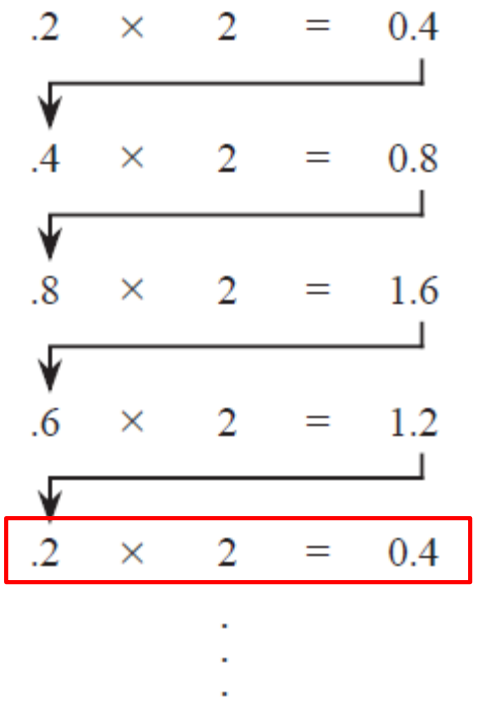
We now combine the integer and fractional portions of the number and obtain the final result:

$$(23.375)_{10} = (10111.011)_2.$$

Non Terminating Fractions

- Although this method of conversion will work among all bases, some precision can be lost in the process. For example, not all terminating base 10 fractions have a terminating base 2 form.

- Consider converting $(.2)_{10}$ to base 2 as shown in Figure. In the last row of the conversion, the fraction .2 reappears, and the process repeats. *As to why this can happen, consider that any* non-repeating base 2 fraction can be represented as *i/2^k for some integers i and k.* (Repeating fractions in base 2 cannot be so represented.)



Binary to other bases

Binary to Octal

- In order to convert a base 2 number into a base 8 number, we partition the base 2 number into groups of three starting from the radix point, and pad the outermost groups with 0's as needed to form triples.
- Then, we convert each triple to the octal equivalent. For conversion from base 2 to base 16, we use groups of four. Consider converting $(10110)_2$ to base 8:

$$(10110)_2 = (010)_2\ (110)_2 = (2)_8\ (6)_8 = (26)_8$$

Notice that the leftmost two bits are padded with a 0 on the left in order to create a full triplet.

Binary to Hexadecimal

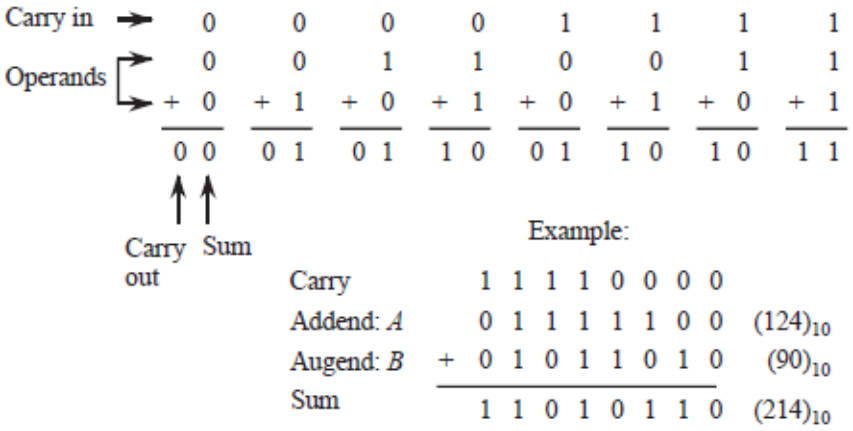
- Now consider converting $(10110110)_2$ to base 16:

$$(10110110)_2 = (1011)_2\ (0110)_2 = (B)_{16}\ (6)_{16} = (B6)_{16}$$

(Note that 'B' is a base 16 digit corresponding to 1110. B is not a variable.)

AN EARLY LOOK AT COMPUTER ARITHMETIC

- We will explore computer arithmetic in detail in Module- 3, but for the moment, we need to learn how to perform simple binary addition because it is used in representing signed binary numbers.
- Binary addition is performed similar to the way we perform decimal addition by hand, as illustrated in Figure beside.
- Two binary numbers *A and B are added from right to left, creating a sum and a carry* in each bit position.
- Since the rightmost bits of *A and B can each assume one of* two values, four cases must be considered: 0 + 0, 0 + 1, 1 + 0, and 1 + 1, with a carry of 0, as shown in the figure.
- The carry into the rightmost bit position defaults to 0.
- For the remaining bit positions, the carry into the position can be 0 or 1, so that a total of eight input combinations must be considered as shown in the figure.



Notice that the largest number we can represent using the eight-bit format shown in Figure is $(11111111)_2 = (255)_{10}$ and that the smallest number that can be represented is $(00000000)_2 = (0)_{10}$. The bit patterns 11111111 and 00000000 and all of the intermediate bit patterns represent numbers on the closed interval from 0 to 255, which are all positive numbers. Up to this point we have considered only unsigned numbers, but we need to represent signed numbers as well, in which (approximately) one half of the bit patterns is assigned to positive numbers and the other half is assigned to negative numbers. Four common representations for base 2 signed numbers are discussed in the next section.

SIGNED FIXED POINT NUMBERS

- Up to this point we have considered only the representation of unsigned fixed point numbers.
- The situation is quite different in representing *signed fixed point* numbers.
- There are four different ways of representing signed numbers that are commonly used: sign-magnitude, one's complement, two's complement, and excess notation.
- We will cover each in turn, using integers for our examples.
- Throughout the discussion, please refer to this Table beside which shows for a 3-bit number how the various representations appear.

Decimal	Unsigned	Sign–Mag.	1's Comp.	2's Comp.	Excess 4
7	111	–	–	–	–
6	110	–	–	–	–
5	101	–	–	–	–
4	100	–	–	–	–
3	011	011	011	011	111
2	010	010	010	010	110
1	001	001	001	001	101
+0	000	000	000	000	100
-0	–	100	111	000	100
-1	–	101	110	111	011
-2	–	110	101	110	010
-3	–	111	100	101	001
-4	–	–	–	100	000

Table: 3-bit Integer Representations

Signed Magnitude

- The **signed magnitude (also referred to as sign and magnitude) representation** is the most familiar to us as the base 10 number system. A plus or minus sign to the left of a number indicates whether the number is positive or negative as in $+12_{10}$ or -12_{10}.
- In the binary signed magnitude representation, the leftmost bit is used for the sign, which takes on a value of 0 or 1 for '+' or '-', respectively.
- The remaining bits contain the absolute magnitude. Consider representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format:

$$(+12)_{10} = (00001100)_2$$

$$(-12)_{10} = (10001100)_2$$

The negative number is formed by simply changing the sign bit in the positive number from 0 to 1. Notice that there are both positive and negative representations for zero: 00000000 and 10000000.

There are eight bits in this example format, and all bit patterns represent valid numbers, so there are $2^8 = 256$ possible patterns. Only $2^8 - 1 = 255$ different numbers can be represented, however, since +0 and -0 represent the same number. We will make use of the signed magnitude representation when we look at floating point numbers

One's Complement

- The **one's complement operation is trivial to perform: convert all of the 1's in** the number to 0's, and all of the 0's to 1's. See the fourth column in the Table before for examples.
- We can observe from the table that in the **one's complement representation** the leftmost bit is 0 for positive numbers and 1 for negative numbers, as it is for the signed magnitude representation. This negation, changing 1's to 0's and changing 0's to 1's, is known as **complementing the bits.**
- **Consider again representing** $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format, now using the one's complement
- representation:

$$(+12)_{10} = (00001100)_2$$

$$(-12)_{10} = (11110011)_2$$

• Note again that there are representations for both +0 and -0, which are 00000000 and 11111111, respectively.

• As a result, there are only 2^8 - 1 = 255 different numbers that can be represented even though there are 2^8 different bit patterns.

The one's complement representation is not commonly used. This is at least partly due to the difficulty in making comparisons when there are two representations for 0. There is also additional complexity involved in adding numbers, which is discussed further in Module 3.

Two's Complement

- The two's complement is formed in a way similar to forming the one's complement: complement all of the bits in the number, but then add 1, and if that addition results in a carry-out from the most significant bit of the number, discard the carry-out.
- Examination of the fifth column of the Table shows that in the **two's complement representation, the leftmost bit is again 0 for positive numbers** and is 1 for negative numbers. However, this number format does not have the unfortunate characteristic of signed-magnitude and one's complement representations: it has only one representation for zero.
- To see that this is true, consider forming the negative of $(+0)_{10,}$ which has the bit pattern:

$$(+0)_{10} = (00000000)_2$$

- Forming the one's complement of $(00000000)_2$ produces $(11111111)_2$ and adding 1 to it yields $(00000000)_2$, thus $(-0)_{10} = (00000000)_2$. The carry out of the leftmost position is discarded in two's complement addition (except when detecting an overflow condition).
- Since there is only one representation for 0, and since all bit patterns are valid, there are $2^8 = 256$ different numbers that can be represented.
- Consider again representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format, this time using the two's complement representation. Starting with $(+12)_{10} = (00001100)_{2,}$ complement, or negate the number, producing $(11110011)_2$.
- Now add 1, producing $(11110100)_2$, and thus $(-12)_{10} = (11110100)_{2:}$

$$(+12)_{10} = (00001100)_2$$

$$(-12)_{10} = (11110100)_2$$

- There is an equal number of positive and negative numbers provided zero is considered to be a positive number, which is reasonable because its sign bit is 0. The positive numbers start at 0, but the negative numbers start at -1, and so the magnitude of the most negative number is one greater than the magnitude of the most positive number. The positive number with the largest magnitude is +127, and the negative number with the largest magnitude is -128.
- There is thus no positive number that can be represented that corresponds to the negative of -128.
- If we try to form the two's complement negative of -128, then we will arrive at a negative number, as shown below:

$$
\begin{aligned}
(-128)_{10} &= (10000000)_2 \\
&\Downarrow \\
&\quad 01111111 \\
&+ \qquad\quad 1 \\
&\text{---------} \\
&\quad (10000000)_2
\end{aligned}
$$

The two's complement representation is the representation most commonly used in conventional computers

Excess Representation

- In the **excess or biased representation, the number is treated as unsigned, but is** "shifted" in value by subtracting the bias from it.
- The concept is to assign the smallest numerical bit pattern, all zeros, to the negative of the bias, and assign the remaining numbers in sequence as the bit patterns increase in magnitude.
- A convenient way to think of an excess representation is that a number is represented as the sum of its two's complement form and another number, which is known as the "excess," or "bias." Once again, refer to the Table, the rightmost column, for examples.
- Consider again representing $(+12)_{10}$ and $(-12)_{10}$ in an eight-bit format but now using an excess 128 representation.
- An excess 128 number is formed by adding 128 to the original number, and then creating the unsigned binary version. For $(+12)_{10,}$ we compute $(128 + 12 = 140)_{10}$ and produce the bit pattern $(10001100)_2$.
- For $(-12)_{10}$, we compute $(128 + -12 = 116)_{10}$ and produce the bit pattern $(01110100)_2$:

$$(+12)_{10} = (10001100)_2$$

$$(-12)_{10} = (01110100)_2$$

Note that there is no numerical significance to the excess value: it simply has the effect of shifting the representation of the two's complement numbers.

- There is only one excess representation for 0, since the excess representation is simply a shifted version of the two's complement representation.
- For the previous case, the excess value is chosen to have the same bit pattern as the largest negative number, which has the effect of making the numbers appear in numerically sorted order if the numbers are viewed in an unsigned binary representation.
- Thus, the most negative number is $(-128)_{10} = (00000000)_2$ and the most positive number is $(+127)_{10} = (11111111)_2$.
- This representation simplifies making comparisons between numbers, since the bit patterns for negative numbers have numerically smaller values than the bit patterns for positive numbers.
- This is important for representing the exponents of floating point numbers, in which exponents of two numbers are compared in order to make them equal for addition and subtraction.

Floating Point Numbers

- The fixed point number representation, which we explored before, has a fixed position for the radix point, and a fixed number of digits to the left and right of the radix point.
- A fixed point representation may need a great many digits in order to represent a practical range of numbers.
- For example, a computer that can represent a number as large as a trillion maintains at least 40 bits to the left of the radix point since $2^{40} \sim 10^{12}$. If the same computer needs to represent one trillionth, then 40 bits must also be maintained to the right of the radix point, which results in a total of 80 bits per number.
- In practice, much larger numbers and much smaller numbers appear during the course of computation, which places even greater demands on a computer.
- A great deal of hardware is required in order to store and manipulate numbers with 80 or more bits of precision, and computation proceeds more slowly for a large number of digits than for a small number of digits.
- Fine precision, however, is generally not needed when large numbers are used, and conversely, large numbers do not generally need to be represented when calculations are made with small numbers.
- A more efficient computer can be realized when only as much precision is retained as is needed.

RANGE AND PRECISION IN FLOATING POINT NUMBERS

- A **floating point representation allows a large range of expressible numbers to be** represented in a small number of digits by separating the digits used for *precision* from the digits used for *range*.
- *The base 10 floating point number representing* Avogadro's number is shown below: $+6.023 \times 10^{23}$ Here, the range is represented by a power of 10, 10^{23} in this case, and the precision is represented by the digits in the fixed point number, 6.023 in this case.
- In discussing floating point numbers, the fixed point part is often referred to as the **mantissa, or significand of the number.**
- **Thus a floating point number can be** characterized by a triple of numbers: sign, exponent, and significand.

- The range is determined primarily by the number of digits in the exponent (two digits are used here) and the base to which it is raised (base 10 is used here) and the precision is determined primarily by the number of digits in the significand (four digits are used here). Thus the entire number can be represented by a sign and 6 digits, two for the exponent and four for the significand.
- Figure below shows how the triple of sign, exponent, significand, might be formatted in a computer.

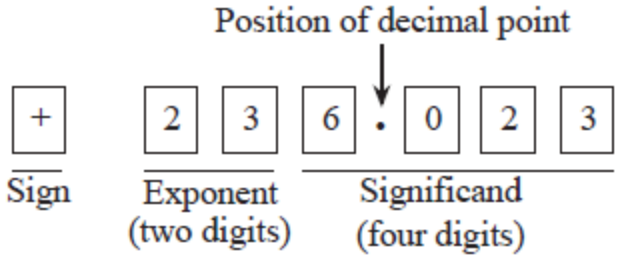


Figure: Representation of a base 10 floating point number.

Notice how the digits are packed together with the sign first, followed by the exponent, followed by the significand. This ordering will turn out to be helpful in comparing two floating point numbers. It has been discussed before that the decimal point does not need to be stored with the number as long as the decimal point is always in the same position in the significand.

- If we need a greater range, and if we are willing to sacrifice precision, then we can use just three digits in the fraction and have three digits left for the exponent without increasing the number of digits used in the representation.
- An alternative method of increasing the range is to increase the base, which has the effect of increasing the precision of the smallest numbers but decreasing the precision of the largest numbers.
- The range/precision trade-off is a major advantage of using a floating point representation, but the reduced precision can cause problems, sometimes leading to disaster.

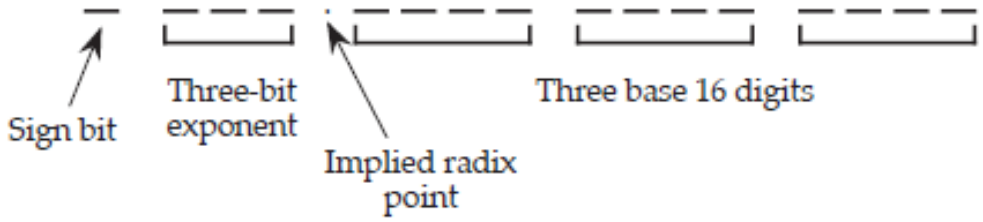
NORMALIZATION

- A potential problem with representing floating point numbers is that the same number can be represented in different ways, which makes comparisons and arithmetic operations difficult.
- For example, consider the numerically equivalent forms shown below: $3584.1 \times 10^0 = 3.5841 \times 10^3 = .35841 \times 10^4.$

In order to avoid multiple representations for the same number, floating point numbers are maintained in **normalized form. That is, the radix point is shifted** to the left or to the right and the exponent is adjusted accordingly until the radix point is to the left of the leftmost nonzero digit. So the rightmost number above is the normalized one. Unfortunately, the number zero cannot be represented in this scheme, so to represent zero an exception is made. The exception to this rule is that zero is represented as all 0's in the mantissa.

REPRESENTING FLOATING POINT NUMBERS IN THE COMPUTER— PRELIMINARIES

- Let us design a simple floating point format to illustrate the important factors in representing floating point numbers on the computer. Our format may at first seem to be unnecessarily complex.
- We will represent the significand in signed magnitude format, with a single bit for the sign bit, and three hexadecimal digits for the magnitude. The exponent will be a 3-bit excess-4 number, with a radix of 16.
- The normalized form of the number has the hexadecimal point to the left of the three hexadecimal digits.
- The bits will be packed together as follows: The sign bit is on the left, followed by the 3-bit exponent, followed by the three hexadecimal digits of the significand.
- Neither the radix nor the hexadecimal point will be stored in the packed form.
- The reason for these rather odd-seeming choices is that numbers in this format can be compared for EQUAL , UNEQUAL, LESS-EQUAL and GREATER-EQUAL in their "packed" format, which is shown in the illustration below:



- Consider representing $(358)_{10}$ in this format.
- The first step is to convert the fixed point number from its original base into a fixed point number in the target base. Using the method described before, we convert the base 10 number into a base 16 number as shown below:

Thus $(358)_{10} = (166)_{16}$. The next step is to convert the fixed point number into a floating point number:

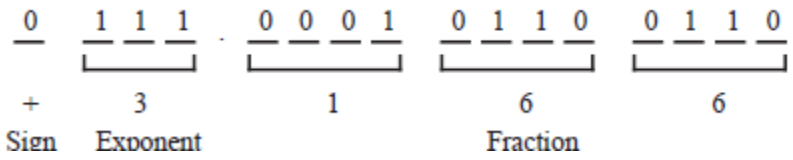
$$(166)_{16} = (166.)_{16} \times 16^0$$

		Integer	Remainder
358/16	=	22	6
22/16	=	1	6
1/16	=	0	1

- The next step is to normalize the number: $(166.)_{16} \times 16^0 = (.166)_{16} \times 16^3$
- Finally, we fill in the bit fields of the number. The number is positive, and so we place a 0 in the sign bit position. The exponent is 3, but we represent it in excess 4, so the bit pattern for the exponent is computed as shown below:

	0 1 1	$(+3)_{10}$
Excess 4	+ 1 0 0	$(+4)_{10}$
Excess 4 exponent	1 1 1	

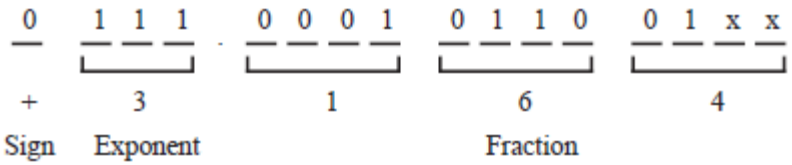
Alternatively, we could have simply computed 3 + 4 = 7 in base 10, and then made the equivalent conversion $(7)_{10} = (111)_2$.
Finally, each of the base 16 digits is represented in binary as 1 = 0001, 6 = 0110, and 6 = 0110. The final bit pattern is shown below:



- Notice again that the radix point is not explicitly represented in the bit pattern, but its presence is implied. The spaces between digits are for clarity only, and do not suggest that the bits are stored with spaces between them. The bit pattern as stored in a computer's memory would look like this: 0111000101100110

The use of an excess 4 exponent instead of a two's complement or a signed magnitude exponent simplifies addition and subtraction of floating point numbers (detailed in Module 3). In order to add or subtract two normalized floating point numbers, the smaller exponent (smaller in degree, not magnitude) must first be increased to the larger exponent (this retains the range), which also has the effect of unnormalizing the smaller number. In order to determine which exponent is larger, we only need to treat the bit patterns as unsigned numbers and then make our comparison. That is, using an excess 4 representation, the smallest exponent is -4, which is represented as 000. The largest exponent is +3, which is represented as 111. The remaining bit patterns for -3, -2, -1, 0, +1, and +2 fall in their respective order as 001, 010, 011, 100, 101, and 110.

- Now if we are given the bit pattern shown above for $(358)_{10}$ along with a description of the floating point representation, then we can easily determine the number. The sign bit is a 0, which means that the number is positive. The exponent in unsigned form is the number $(+7)_{10}$, but since we are using excess 4, we must subtract 4 from it, which results in an actual exponent of $(+7 - 4 = +3)_{10}$.
- The fraction is grouped in four-bit hexadecimal digits, which gives a fraction of $(.166)_{16}$. Putting it all together results in $(+.166 \times 163)_{16} = (358)_{10}$.
- Now suppose that only 10 bits are allowed for the fraction in the above example, instead of the 12 bits that group evenly into fours for hexadecimal digits. How does the representation change? One approach might be to round the fraction and adjust the exponent as necessary. Another approach, which we use here, is to simply truncate the least significant bits by **chopping and avoid making adjustments** to the exponent, so that the number we actually represent is:



This method of truncation produces a biased error, since values of 00, 01, 10, and 11 in the missing bits are all treated as 0, and so the error is in the range from 0 to $(.003)_{16}$. The bias comes about because the error is not symmetric about 0.

We again stress that whatever the floating point format is, that it be known to all parties that intend to store or retrieve numbers in that format. The Institute of Electrical and Electronics Engineers (IEEE), has taken the lead in standardizing floating point formats. The IEEE 754 floating point format, which is in nearly universal usage, is discussed later.

THE IEEE 754 FLOATING POINT STANDARD

- There are many ways to represent floating point numbers, a few of which we have already explored.
- Each representation has its own characteristics in terms of range, precision, and the number of representable numbers.
- In an effort to improve software portability and ensure uniform accuracy of floating point calculations, the IEEE 754 floating point standard for binary numbers was developed (IEEE, 1985).
- There are a few entrenched product lines that predate the standard that do not use it, such as the IBM/370, the DEC VAX, and the Cray line, but virtually all new architectures generally provide some level of IEEE 754 support.
- The IEEE 754 standard as described next must be supported by a computer *system,* and not necessarily by the hardware entirely.
- That is, a mixture of hardware and software can be used while still conforming to the standard.

- The sign bit is in the leftmost position and indicates a positive or negative number for a 0 or a 1, respectively. The 8-bit excess 127 (*not 128) exponent follows,* in which the bit patterns 00000000 and 11111111 are reserved for special cases, as described below.
- For double precision, the 11-bit exponent is represented in excess 1023, with 00000000000 and 11111111111 reserved. The 23-bit base 2 fraction follows.
- There is a hidden bit to the *left of the binary point, which when* taken together with the single-precision fraction form a 23 + 1 = 24-bit significand of the form 1.fff...f where the fff...f pattern represents the 23-bit fractional part that is stored.
- The double-precision format also uses a hidden bit to the left of the binary point, which supports a 52 + 1 = 53 bit significand.
- For both formats, the number is normalized unless **denormalized numbers are supported, as** described later.
- There are five basic types of numbers that can be represented. Nonzero normalized numbers take the form described above.
- A so-called "clean zero" is represented by the reserved bit pattern 00000000 in the exponent and all 0's in the fraction.
- The sign bit can be 0 or 1, and so there are two representations for zero: +0 and -0.

- Infinity has a representation in which the exponent contains the reserved bit pattern 11111111, the fraction contains all 0's, and the sign bit is 0 or 1. Infinity is useful in handling overflow situations or in giving a valid representation to a number (other than zero) divided by zero. If zero is divided by zero or infinity is divided by infinity, then the result is undefined. This is represented by the **NaN** (not a number) format in which the exponent contains the reserved bit pattern 11111111, the fraction is nonzero and the sign bit is 0 or 1.
- A NaN can also be produced by attempting to take the square root of -1.
- As with all normalized representations, there is a large gap between zero and the first representable number.
- The denormalized, "dirty zero" representation allows numbers in this gap to be represented. The sign bit can be 0 or 1, the exponent contains the reserved bit pattern 00000000 which represents -126 for single precision (-1022 for double precision), and the fraction contains the actual bit pattern for the magnitude of the number.
- Thus, there is no hidden 1 for this format.
- Note that the *denormalized representation is not an unnormalized representation.* The key difference is that there is only one representation for each denormalized number, whereas there are infinitely many unnormalized representations.
- Next Figure illustrates some examples of IEEE 754 floating point numbers.

	Value	Bit Pattern		
		Sign	Exponent	Fraction
(a)	$+1.101 \times 2^5$	0	1000 0100	101 0000 0000 0000 0000 0000
(b)	-1.01011×2^{-126}	1	0000 0001	010 1100 0000 0000 0000 0000
(c)	$+1.0 \times 2^{127}$	0	1111 1110	000 0000 0000 0000 0000 0000
(d)	+0	0	0000 0000	000 0000 0000 0000 0000 0000
(e)	–0	1	0000 0000	000 0000 0000 0000 0000 0000
(f)	+∞	0	1111 1111	000 0000 0000 0000 0000 0000
(g)	$+2^{-128}$	0	0000 0000	010 0000 0000 0000 0000 0000
(h)	+NaN	0	1111 1111	011 0111 0000 0000 0000 0000
(i)	$+2^{-128}$	0	011 0111 1111	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Figure: Examples of IEEE 754 floating point numbers in single precision format (a – h) and double precision format (i). Spaces are shown for clarity only: they are not part of the representation.

- Examples (a) through (h) are in single precision format and example (i) is in double precision format.
- Example (a) shows an ordinary single precision number. Notice that the significand is 1.101, but that only the fraction (101) is explicitly represented.
- Example (b) uses the smallest single precision exponent (–126) and
- example (c) uses the largest single precision exponent (127).
- Examples (d) and (e) illustrate the two representations for zero.
- Example (f) illustrates the bit pattern for +infinity. There is also a corresponding bit pattern for –infinity.
- Example (g) shows a denormalized number. Notice that although the number itself is 2-128, the smallest representable exponent is still -126. The exponent for single precision denormalized numbers is always -126, which is represented by the bit pattern 00000000 and a nonzero fraction. The fraction represents the magnitude of the number, rather than a significand. Thus we have $+2^{-128} = +.01 \times 2^{-126}$, which is represented by the bit pattern shown in above Figure.

- Example (h) shows a single precision NaN. A NaN can be positive or negative.
- Finally, example (i) revisits the representation of 2^{-128} but now using double precision. The representation is for an ordinary double precision number and so there are no special considerations here.
- Notice that 2^{-128} has a significand of 1.0, which is why the fraction field is all 0's.

- In addition to the single precision and double precision formats, there are also **single extended and double extended formats. The extended formats are not** visible to the user, but they are used to retain a greater amount of internal precision during calculations to reduce the effects of roundoff errors.
- The extended formats increase the widths of the exponents and fractions by a number of bits that can vary depending on the implementation.
- For instance, the single extended format adds at least three bits to the exponent and eight bits to the fraction.
- The double extended format is typically 80 bits wide, with a 15-bit exponent and a 64-bit fraction.

Rounding

- An implementation of IEEE 754 must provide at least single precision, whereas the remaining formats are optional.
- Further, the result of any single operation on floating point numbers must be accurate to within half a bit in the least significant bit of the fraction.
- This means that some additional bits of precision may need to be retained during computation (referred to as **guard bits), and there** must be an appropriate method of rounding the intermediate result to the number of bits in the fraction.
- There are four rounding modes in the IEEE 754 standard. One mode rounds to 0, another rounds toward +infinity, and another rounds toward -infinity.
- The default mode rounds to the nearest representable number. Halfway cases round to the number whose low order digit is even.
- For example, 1.01101 rounds to 1.0110 whereas 1.01111 rounds to 1.1000.

Character Codes

- Unlike real numbers, which have an infinite range, there is only a finite number of characters.
- An entire character set can be represented with a small number of bits per character.
- Three of the most common character representations, ASCII, EBCDIC, and Unicode.
- Here we will see ASCII

THE ASCII CHARACTER SET

- The American Standard Code for Information Interchange (**ASCII) is summarized** in next Figure, using hexadecimal indices. The representation for each character consists of 7 bits, and all 27 possible bit patterns represent valid characters.
- The characters in positions 00 – 1F and position 7F are special control characters that are used for transmission, printing control, and other non-textual purposes.
- The remaining characters are all printable, and include letters, numbers, punctuation, and a space. The digits 0-9 appear in sequence, as do the upper and lower case letters I [As an aside, the character 'a' and the character 'A' are different, and have different codes in the ASCII table. The small letters like 'a' are called **lower case, and the capital letters like 'A' are** called **upper case. The naming comes from the positions of the characters in a printer's typecase.** The capital letters appear above the small letters, which resulted in the upper case / lower case naming. These days, typesetting is almost always performed electronically, but the traditional naming is still used.]
- This organization simplifies character manipulation.
- In order to change the character representation of a digit into its numerical value, we can subtract $(30)_{16}$ from it. In order to convert the ASCII character '5,' which is in position $(35)_{16}$, into the number 5, we compute $(35 - 30 = 5)_{16}$.
- In order to convert an upper case letter into a lower case letter, we add $(20)_{16}$. For example, to convert the letter 'H,' which is at location $(48)_{16}$ in the ASCII table, into the letter 'h,' which is at position $(68)_{16}$, we compute $(48 + 20 = 68)_{16}$.

00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	´	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

NUL	Null	FF	Form feed	CAN	Cancel
SOH	Start of heading	CR	Carriage return	EM	End of medium
STX	Start of text	SO	Shift out	SUB	Substitute
ETX	End of text	SI	Shift in	ESC	Escape
EOT	End of transmission	DLE	Data link escape	FS	File separator
ENQ	Enquiry	DC1	Device control 1	GS	Group separator
ACK	Acknowledge	DC2	Device control 2	RS	Record separator
BEL	Bell	DC3	Device control 3	US	Unit separator
BS	Backspace	DC4	Device control 4	SP	Space
HT	Horizontal tab	NAK	Negative acknowledge	DEL	Delete
LF	Line feed	SYN	Synchronous idle		
VT	Vertical tab	ETB	End of transmission block		

Figure :The ASCII character code, shown with hexadecimal indices.

Summary of Module-2

- *All data in a computer is represented in terms of bits, which can be organized and interpreted as integers, fixed point numbers, floating point numbers, or characters.*
- *Character codes, such as ASCII, EBCDIC, and Unicode, have finite sizes and can thus be completely represented in a finite number of bits.*
- *The number of bits used for representing numbers is also finite, and as a result only a subset of the real numbers can be represented.This leads to the notions of range, precision, and error.*
- *The range for a number representation defines the largest and smallest magnitudes that can be represented, and is almost entirely determined by the base and the number of bits in the exponent for a floating point representation.The precision is determined by the number of bits used in representing the magnitude (excluding the exponent bits in a floating point representation).*
- *Error arises in floating point representations because there are real numbers that fall within the gaps between adjacent representable numbers.*