**Chapter 3 – Data Representation**

The focus of this chapter is the representation of data in a digital computer. We begin with a   
review of several number systems (decimal, binary, octal, and hexadecimal) and a discussion   
of methods for conversion between the systems. The two most important methods are   
conversion from decimal to binary and binary to decimal. The conversions between binary   
and each of octal and hexadecimal are quite simple. Other conversions, such as conversion   
to and from base–5 are of interest only to teachers with a bad sense of humor.

After discussion of conversion between bases, we discuss the methods used to store integers   
in a digital computer: one’s complement and two’s complement arithmetic. This includes a   
characterization of the range of integers that can be stored given the number of bits allocated   
to store an integer. The most common integer storage formats are 16 and 32 bits.

The next topic for this chapter is the storage of real (floating point) numbers. This discussion   
will focus on the standard put forward by the Institute of Electrical and Electronic Engineers,   
the IEEE Standard 754 for floating point numbers. The chapter closes with a discussion of   
codes for storing characters: ASCII, EBCDIC, and Unicode.

**General Remark on Data Storage in a Computer**

Data (numeric, character, and more complex structures) are all stored in the computer and   
on disk in binary form. Each common storage medium supports only two values, either two   
distinct voltages or two distinct magnetic states, or some other two–state system. The most   
common way to denote these two states is based on the binary number system.

Consider what happens when a data value is read from computer memory. Each value is   
encoded as a number of bits (binary values: 0 or 1) and stored in the same number of memory   
cells, which we shall discuss later. When a value is read, each cell returns a voltage to the   
reading circuitry: either a positive voltage (say 1.5 volts), interpreted as a logic 1, or a zero   
voltage, interpreted as a logic 0. It is important to note that the actual value of the positive   
voltage is so dependent on technology that we can say little about its actual value.

A stored–program computer (that is, almost any computer or digital device now in use) stores   
both data and program instructions in memory. Each item stored in memory is best viewed   
as nothing more than a collection of binary bits. What that item signifies depends on how   
that item is used by the computer. An item may be used as an instruction, as an address of   
another item, or as data in one of many formats.

When we study some modern memory architectures, we shall discover several mechanisms   
that make it less likely that an item of one type will be used in an incorrect context. While   
these mechanisms are widespread, they are additions to the basic computer design and not   
fundamental to it. As an example, an unfortunate programming error by the author of these   
notes caused the stored representation of the real number 2.5 to be executed as if it were a   
valid instruction. The problem with that was that it executed properly and redirected the   
program to another area, where the program abruptly ceased execution.

Number Systems

There are four number systems of possible interest to the computer programmer: decimal,   
binary, octal, and hexadecimal. Each system is characterized by its **base** or **radix**, always   
given in decimal, and the set of permissible digits. For small sets, such as each of these,   
we specify the set by listing all of its members; for example the set of ten decimal digits.

Sets specified by listing all items belonging to the set often present the items in some   
sort of order, but that is not necessary, as sets are by definition unordered collections.

Note that the hexadecimal numbering system calls for more than ten digits, so we use the   
first six letters of the alphabet. It is preferable to use upper case letters for these “digits”.

Decimal Base = 10  
 Digit Set = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Binary Base = 2  
 Digit Set = {0, 1}

Octal Base = 8 = 23  
 Digit Set = {0, 1, 2, 3, 4, 5, 6, 7}

Hexadecimal Base = 16 = 24  
 Digit Set = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

While we are more accustomed to decimal notation for writing numeric values, we find that   
they can be written in any base. Here are the binary, octal, and hexadecimal equivalents of   
the first seventeen non–negative decimal integers

Decimal Binary Octal Hexadecimal  
 (base 2) (base 8) (base 16)

0 0000 00 0

1 0001 01 1 Note that conversions from hexadecimal

2 0010 02 2 to binary can be done one digit at a time,

3 0011 03 3 thus DE = 11011110, as D = 1101 and

4 0100 04 4 E = 1110. We shall normally denote

5 0101 05 5 this as DE = 1101 1110 with a space

6 0110 06 6 to facilitate reading the binary.

7 0111 07 7

8 1000 10 8 Conversion from binary to hexadecimal

9 1001 11 9 is also quite easy. Group the bits four at

10 1010 12 A a time and convert each set of four.

11 1011 13 B Thus 10111101, written 1011 1101 for

12 1100 14 C clarity is BD because 1011 = B and

13 1101 15 D 1101 = D.

14 1110 16 E

15 1111 17 F  
 16 10000 20 10

**Notation for Constants**

Given the choice of at least four possible values for the base of the numbering system   
(2, 8, 10, and 16), how do we indicate which base is to be used to interpret a constant   
value, such as **011**? This is a valid number in each of the bases, though its value   
depends on which base is being used.

Pure binary numbers will rarely be used in these notes. When they are, they will be   
almost obvious, having only 0 or 1 as digits. These constants will be noted by a small   
subscript 2, as in **0112**, the binary equivalent of decimal 3.

For values written in octal, decimal, and hexadecimal notation, these notes, along with   
most other textbooks, use the notation of Java and C/C++ to indicate the choice of base   
values. Decimal values are written simply as one would expect. Octal constants are   
prefixed by a leading “**0**” (zero), while hexadecimal constants are prefixed by “**0x**”.

For example:  
 **value = 77** // This is the decimal value 77. No special notation is used.  
 **value = 077** // This is the octal value 077, equal to 7•8 + 7 = 62 decimal.  
 **value = 0x77** // This is hexadecimal 77, equal to 7•16 + 7 = 119 decimal.

**Positional Notation and Decimal Numbers**

Before discussing number bases other than decimal, it would be helpful to use the decimal   
system to review positional notation, as always used to write integer values.

Positional notation is formed using powers of the base. These powers are denoted by   
exponents, written as superscripts. For integer representations, all exponents are taken   
from the set of non–negative integers. We review this notation.

Any non–zero number raised to the zero power has value 1: N0 = 1, for N ≠ 0.  
Here are some powers of the various bases.   
 100 = 1 20 = 1 160 = 1  
 101 = 10 21 = 2 161 = 24 = 16  
 102 = 10•10 = 100 22 = 2•2 = 4 162 = 28 = 256  
 103 = 10•100 = 1000 23 = 2•4 = 8 163 = 212 = 4,096  
 104 = 10•103 = 10,000 24 = 2•23 = 16 164 = 216 = 65,536

Consider the value1453, commonly understood as “one thousand four hundred fifty three”.   
Specifically 1453 = 1000 + 400 + 50 + 3. Written in terms of powers of ten, we see that   
1453 = 1•103 + 4•102 + 5•101 + 3•100. Similarly, the larger number 322,583,006 which   
represents 3•108 + 2•107 + 2•106 + 5•105 + 8•104 + 3•103 + 0•102 + 0•101 + 6•100.

**Positional Notation and Other Base Values**

We are now in a position to generalize the idea of positional notation. Consider 1453.  
We have already seen the decimal evaluation of 1453.   
If read as octal notation 01453 = 1•83 + 4•82 + 5•81 + 3•80 = 811 decimal.  
If read as hexadecimal notation 0x1453 = 1•163 + 4•162 + 5•161 + 3•160 = 5,203 decimal.  
The value 1453 cannot be read as binary, because it has digits other than 0 and 1.

**Which Base to Use?**

The choice of the base to use is made solely for the convenience of the person who must   
examine or otherwise use the information. In many courses, this person is the instructor   
who must grade the homework and who prefers a more compact notation.

Almost always, decimal notation is preferable when writing a computer program in any   
language, such as COBOL, Java, C/C++, or even IBM assembler. The examples below,   
which illustrate proper usage of other bases, are given without explanation.

1. Sometimes it is preferable to use hexadecimal notation when writing software, known as   
 a device driver, used to control external devices, such as printers and disks.  
 For example, one might write: **device\_mask = 0xC040**.

This line of code will set binary bits in a device control register. The register is seen to   
 have sixteen bits, with values set as follows:

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

Of course, one could just as easily write this as device\_mask = –16320 using decimal   
 notation, but most programmers would prefer the hexadecimal. One might note here that,   
 with the exception of classroom exercises, very few programmers write device drivers.

2. Some time it is necessary to examine the raw contents of memory. Thankfully, this is   
 rarely necessary as high–grade debuggers do the job for us. Again, the most common use   
 for this these days is the assignment of vexing problems by overly zealous professors.

As an example (to be explained later), here is the binary representation of the real number   
 –0.75: **1011 1111 0100 0000 0000 0000 0000 0000**

Note that the 32 binary bits are grouped by fours for ease of reading. The hexadecimal   
 notation is even easier to read: **BF40 0000**.

3. The choice of octal or hexadecimal notation usually depends on the value to be   
 represented. Octal digits can be considered as short hand for groups of three binary  
 bits, while hexadecimal digits are short hand for groups of four binary bits.

Most values to be represented in either binary or a variant of it are best considered as   
 values having 8, 16, 32, or 64 bits. As each of these lengths is a multiple of four, the   
 most natural choice would be hexadecimal notation. Consider the example above.

Some older, and now obsolete, computers had word lengths that were multiples of 3;   
 for these octal notation might be preferred. The old PDP–8 used 12–bit words, and   
 the old PDP–9 used 18–bit words, represented as 4 or 6 octal digits respectively.

**Note on History:** This author is quite fond of mentioning obsolete computers, among   
 them the PDP–8, PDP–9, PDP–11, VAX–11/780, CDC–6600,   
 CDC–7600, the Cray–1, and Cray–2. This is done to show that there   
 were actual computers with such–and–such properties.  
 Nothing in this book requires any knowledge of such old favorites.

**Note on Usage:** The remainder of this book will usually ignore the octal base.

The fact that the bases for octal and hexadecimal are powers of the basis for binary   
facilitates the conversion between the bases 2, 8, and 16. The conversion can be done   
one digit at a time, remembering that each octal digit corresponds to three binary bits   
and each hexadecimal digit corresponds to four binary bits. Conversion between decimal   
and one of the other three bases is only a bit more complex.

As noted above, we shall focus on binary, decimal, and hexadecimal numbers. As octal   
notation is rarely used these days, we shall not consider it further.

**How to Group the Bits?**

This is a small problem that occurs in translations between binary and hexadecimal notation.   
As each hexadecimal digit stands for four binary bits, these two translations (binary to hex   
and hex to binary) are just matters of grouping bits. We begin by discussing integers.

Consider the hexadecimal number BAD. We translate digit by digit to obtain the binary.   
Hexadecimal B stands for binary 1011, A for 1010, and D for 1101. The binary value   
represented is **101110101101**, better written as **1011 1010 1101**.

Consider conversion of the binary number **111010** to hexadecimal. If we try to group the   
bits four at a time we get either **11 1010** or **1110 10**. The first option is correct as the   
grouping must be done from the right. We then add leading zeroes to get groups of four   
binary bits, obtaining **0011 1010**, which is converted to **3A** as 0011 = 3 and 1010 = A.  
Note that the padding of the leading **11** to the full four–bit **0011** is not strictly necessary,   
as most readers are quite comfortable with interpreting binary 11 as decimal 3.

We shall have occasion to do conversions involving binary and hexadecimal to the right   
of the decimal point. Consider the hexadecimal value **0.C8**, which is **0.1100 1000** in   
binary. Here the grouping is not a problem; just represent each hex digit as four bits.

Consider the binary value **0.111010**, which represents the decimal value 0.90625   
(think that **0.111010** stands for 1/2 + 1/4 + 1/8 + 0/16 + 1/32 = 29/32). When the digits   
are to the right of the decimal, they must be grouped from the left and padded out with   
trailing zeroes so that each collection has four bits. Thus **0.111010** is converted to  
 **0.1110 1000**, or **0.C8**. Note that leaving the value as **0.1110 10** might lead   
one to consider the last grouping as decimal 2, which is likely to cause confusion.

Unsigned Binary Integers

There are two common methods to store **unsigned** integers in a computer: binary numbers   
(which we discuss now) and Packed Decimal (which we discuss later). From a theoretical   
point of view, it is important to note that no computer really stores the set of integers in   
that it can represent in native form an arbitrary member of that infinite set.

For each of the standard formats (short, integer, long, etc.) there is a largest positive value   
that can be stored and a smallest non–positive value that can be stored. More on this later.  
There is a common data type, usually called “**bignum**”, which allows storage of integers of   
any size. You want a 10,000–digit integer, you got one. However, this data type is mostly   
implemented in software with hardware assists, and is not a native type of a standard CPU.

The standard integer types are stored as collections of a number of binary bits; thus an   
N–bit integer is represented by N binary bits. **1110 1000** would be an 8–bit integer.

It is easy to show that an N–bit binary integer can represent one of 2N possible integer values.   
Here is the proof by induction.

1. A one–bit integer can store 2 values: 0 or 1. This is the base for induction.

2. Suppose an N–bit integer, unconventionally written as BN–1 … B3B2B1B0.  
 By the inductive hypothesis, this can represent one of 2N possible values.

3. We now consider an (N+1)–bit integer, written as BNBN–1 … B3B2B1B0,   
 where either BN = 0 or BN = 1. By the inductive hypothesis, there are   
 2N values of the form 0BN–1 … B3B2B1B0,  
 and 2N values of the form 1BN–1 … B3B2B1B0.

4. The total number of (N+1)–bit values is 2N + 2N = 2N+1. The claim is proved.

By inspection of the above table, we see that there are 16 possible values for a four–bit   
unsigned integer. These range from decimal 0 through decimal 15 and are easily represented   
by a single hexadecimal digit. Each hexadecimal digit is shorthand for four binary bits.

In the standard interpretation, always used in this course, an N–bit **unsigned** integer will   
represent 2N integer values in the range 0 through 2N – 1, inclusive. Sample ranges include:

N = 4 0 through 24 – 1 0 through 15  
 N = 8 0 through 28 – 1 0 through 255  
 N = 12 0 through 212 – 1 0 through 4095  
 N = 16 0 through 216 – 1 0 through 65535  
 N = 20 0 through 220 – 1 0 through 1,048,575  
 N = 32 0 through 232 – 1 0 through 4,294,967,295

For most applications, the most important representations are 8 bit, 16 bit, and 32 bit. To this   
mix, we add 12–bit unsigned integers as they are used in the base register and offset scheme   
of addressing used by the IBM Mainframe computers. Recalling that a hexadecimal digit is   
best seen as a convenient way to write four binary bits, we have the following.

8 bit numbers 2 hexadecimal digits 0 through 255,  
 12 bit numbers 3 hexadecimal digits 0 through 4095,  
 16 bit numbers 4 hexadecimal digits 0 through 65535, and  
 32 bit numbers 8 hexadecimal digits 0 through 4,294,967,295.

Conversions between Decimal and Binary

We now consider methods for conversion from decimal to binary and binary to decimal. We   
consider not only whole numbers (integers), but numbers with decimal fractions. To convert   
such a number, one must convert the integer and fractional parts separately.

At this point in the text, we have yet to consider signed integers. The reason for the focus on   
unsigned values is that all conversions between decimal and the other two formats (binary   
and hexadecimal) are best learned using unsigned values.

Consider the conversion of the number 23.375. The method used to convert the integer part   
(23) is different from the method used to convert the fractional part (.375). We shall discuss   
two distinct methods for conversion of each part and leave the student to choose his/her   
favorite. After this discussion we note some puzzling facts about exact representation of   
decimal fractions in binary; e.g. the fact that 0.20 in decimal cannot be exactly represented.   
As before we present two proofs and let the student choose his/her favorite.

The intuitive way to convert decimal 23 to binary is to note that 23 = 16 + 7 = 16 + 4 + 2 + 1;   
thus decimal 23 = 10111 binary. As an eight bit binary number, this is 0001 0111. Note that   
we needed 5 bits to represent the number; this reflects the fact that 24 < 23 ≤ 25. We expand   
this to an 8-bit representation by adding three leading zeroes.

The intuitive way to convert decimal 0.375 to binary is to note that 0.375 = 1/4 + 1/8 =  
0/2 + 1/4 + 1/8, so decimal .375 = binary .011 and decimal 23.375 = binary 10111.011.

Most students prefer a more mechanical way to do the conversions. Here we present that   
method and encourage the students to learn this method in preference to the previous.

Conversion of integers from decimal to binary is done by repeated integer division with   
keeping of the integer quotient and noting the integer remainder. The remainder numbers are   
then read **bottom to top** as least significant bit to first bit generated. Here is an example.

Quotient Remainder

23/2 = 11 1 Thus decimal 23 = binary 10111

11/2 = 5 1

5/2 = 2 1 Remember to read the binary number

2/2 = 1 0 from bottom to top.

1/2 = 0 1

Conversion of the fractional part is done by repeated multiplication with copying of the   
whole number part of the product and subsequent multiplication of the fractional part. All   
multiplications are by 2. Read the binary results **top to bottom**. Here is an example.

Number Product Binary

0.375 x 2 = 0.75 0

0.75 x 2 = 1.5 1

0.5 x 2 = 1.0 1  
 0.0 x 2 = 0.0 0  
 0.0 x 2 = 0.0 0

The process terminates when the product of the last multiplication is 1.0. At this point we   
copy the last 1 generated and have the result; thus decimal 0.375 = 0.011 binary. Note that   
this conversion was carried two steps beyond the standard termination point to emphasize   
the fact that after a point, only trailing zeroes are generated.

0.375 = 0.3750 = 0.37500, etc. 0.011 = 0.0110 = 0.01100, etc.

**Conversions between Decimal and Hexadecimal**

One way to convert is by first converting to binary. We consider conversion of 23.375   
from decimal to hexadecimal. We have noted that the value is 10111.011 in binary.

To convert this binary number to hexadecimal we must group the binary bits in groups of   
four, adding leading and trailing zeroes as necessary. We introduce spaces in the numbers   
in order to show what is being done.

10111.011 = 1 0111.011.

To the left of the decimal we group from the right and to the right of the decimal we group   
from the left. Thus 1.011101 would be grouped as 1.0111 01.

At this point we must add extra zeroes to form four bit groups. So

10111.011 = 0001 0111.0110.

Conversion to hexadecimal is done four bits at a time. The answer is 17.6 hexadecimal.

Another Way to Convert Decimal to Hexadecimal

Some readers may ask why we avoid the repeated division and multiplication methods in   
conversion from decimal to hexadecimal. Just to show it can be done, here is an example.

Consider the number 7085.791748046875. As an example, we convert this to hexadecimal.

The first step is to use repeated division to produce the whole–number part.

7085 / 16 = 442 with remainder = 13 or hexadecimal D

442 / 16 = 27 with remainder = 10 or hexadecimal A  
 27 / 16 = 1 with remainder = 11 or hexadecimal B  
 1 / 16 = 0 with remainder = 1 or hexadecimal 1.

The whole number is read bottom to top as 1BAD.

Now we use repeated multiplication to obtain the fractional part.

0.791748046875 • 16 = 12.6679875 Remove the 12 or hexadecimal C

0.6679875 • 16 = 10.6875 Remove the 10 or hexadecimal A

0.6875 • 16 = 11.00 Remove the 11 or hexadecimal B

0.00 • 16 = 0.0

The fractional part is read top to bottom as CAB. The hexadecimal value is 1BAD.CAB,   
which is a small joke on the author’s part. The only problem is to remember to write   
results in the decimal range 10 through 15 as hexadecimal A through F.

Long division is of very little use in converting the whole number part. It does correctly   
produce the first quotient and remainder. The intermediate numbers may be confusing.

**442**

**16)7085**

**64**

**68**

**64**

**45**

**32**

**13**

**But Why Do These Methods Work?**

In this optional section, we present some illustrative examples to indicate why each of these   
conversion methods work.

Conversion of Integers: Decimal to Binary

Our sample integer, used above, is 23 = 16 + 4 + 2 + 1 = 1•24 + 0•23 + 1•22 + 1•2 + 1. To   
generalize this to any number representable as 5–bit unsigned binary, we use some algebra.  
Let 23 = A4•24 + A3•23 + A2•22 + A1•2 + A0, where A4 = 1, A3 = 0, A2 = 1, etc. Now begin   
the repeated divisions by 2, noting that we track only whole numbers.

(A4•24 + A3•23 + A2•22 + A1•2 + A0) / 2 = (A4•23 + A3•22 + A2•2 + A1), rem = A0.  
(A4•23 + A3•22 + A2•2 + A1) / 2 = (A4•22 + A3•2 + A2), rem = A1.  
(A4•22 + A3•2 + A2) / 2 = (A4•2 + A3), rem = A2.  
(A4•2 + A3) / 2 = A4 rem = A3.  
A4 / 2 = 0 rem = A4.

Continuation of the process will generate only leading zeroes, reflecting the fact that the   
decimal value 23 is binary 10111 = 010111 = 0010111 = 0001 0111, etc.

Conversion of “Fractional Parts”: Decimal to Binary

Our sample “fractional part” (the author does not know a better term) is 0.375. Since this   
conversion moves a bit quickly, we use here another value: 0.65625 = 1/2 + 1/8 + 1/32.   
Let 0.65625 = B1/2 + B2/22 + B3/23 + B4/24 + B5/25, with B1 = 1, B2 = 0, B3 = 1, etc.  
Now begin the repeated multiplications, tracking only the “fractional part”.

(B1/2 + B2/22 + B3/23 + B4/24 + B5/25) • 2 = B1 + (B2/2 + B3/22 + B4/23 + B5/24)   
(B2/2 + B3/22 + B4/23 + B5/24) • 2 = B2 + (B3/2 + B4/22 + B5/23)   
(B3/2 + B4/22 + B5/23) • 2 = B3 + (B4/2 + B5/22)   
(B4/2 + B5/22) • 2 = B4 + B5/2   
B5/2 • 2 = B5

Continuing this will lead to trailing zeroes, the value in binary being either 0.10101 or   
0.101010, or more conventionally 0.1010 1000.

What about Other Bases?

The methods above apply to conversion between any two radix values, though they are   
probably difficult to use for conversions from any base other than decimal (as we are quite   
familiar with decimal arithmetic). For us, the only other base is hexadecimal.

Consider our sample number 7085 = A3•163 + A2•162 + A1•16 + A0, with A3 = 1,   
A2 = 11 (B in hexadecimal), A1 = 10 (0xA), and A0 = 13 (0xD).

(A3•163 + A2•162 + A1•16 + A0) / 16 = (A3•162 + A2•16 + A1), rem = A0.  
(A3•162 + A2•16 + A1) / 16 = (A3•16 + A2), rem = A1.  
(A3•16 + A2) / 16 = A3 rem = A2.  
A3 / 16 = 0, rem = A3.

# Non-terminating Fractions

We now make a detour to note a surprising fact about binary numbers – that some fractions   
that terminate in decimal are non-terminating in binary. We first consider terminating and   
non-terminating fractions in decimal. All of us know that 1/4 = 0.25, which is a terminating   
fraction, but that 1/3 = 0.333333333333333333333333333333…, a non-terminating fraction.

We offer a demonstration of why 1/4 terminates in decimal notation and 1/3 does not, and   
then we show two proofs that 1/3 cannot be a terminating fraction in either binary or decimal.

Consider the following sequence of multiplications  
 ¼ • 10 = 2½  
 ½ • 10 = 5. Thus 1/4 = 25/100 = 0.25.

Put another way, ¼ = (1/10) • (2 + ½) = (1/10) • (2 + (1/10) • 5).

However, 1/3 • 10 = 10/3 = 3 + 1/3, so repeated multiplication by 10 continues to yield a   
fraction of 1/3 in the product; hence, the decimal representation of 1/3 is non-terminating.

Explicitly, we see that 1/3 = (1/10) • (3 + 1/3) = (1/10) • (3 + (1/10) • (3 + 1/3)), etc.

In decimal numbering, a fraction is terminating if and only if it can be represented in the   
form J / 10K for some integers J and K. We have seen that 1/4 = 25/100 = 25/102, thus the   
fraction 1/4 is a terminating fraction because we have shown the integers J = 25 and K = 2.

Here are two proofs that the fraction 1/3 cannot be represented as a terminating fraction in   
decimal notation. The first proof relies on the fact that every positive power of 10 can be   
written as 9•M + 1 for some integer M. The second relies on the fact that 10 = 2•5, so that   
10K = 2K•5K. To motivate the first proof, note that 100 = 1 = 9•0 + 1, 10 = 9•1 + 1,

100 = 9•11 + 1, 1000 = 9•111 + 1, etc. If 1/3 were a terminating decimal, we could solve   
the following equations for integers J and M.

, which becomes 3•J = 9•M + 1 or 3•(J – 3•M) = 1. But there is no   
integer X such that 3•X = 1 and the equation has no integer solutions.

The other proof also involves solving an equation. If 1/3 were a non-terminating fraction,   
then we could solve the following equation for J and K.

, which becomes 3•J = 2K•5K. This has an integer solution J only if the   
right hand side of the equation can be factored by 3. But neither 2K nor 5K can be factored by   
3, so the right hand side cannot be factored by 3 and hence the equation is not solvable.

From this, it immediately follows that the fraction 1/3, as a decimal number, does not   
terminate. The fact that the base–3 representation (0.13), base–6 representation (0.23), and   
base–9 representation (0.39) do terminate is only of marginal academic interest.

**NOTE:** This discussion of an obscure point in numeric representation might seem to be   
 pointless, but it has direct bearing on the precision with which real numbers may   
 be stored and processed in a computer.

Now consider the innocent looking decimal 0.20. We show that this does not have a   
terminating form in binary. We first demonstrate this by trying to apply the   
multiplication method to obtain the binary representation.

Number Product Binary

0.20 • 2 = 0.40 0  
 0.40 • 2 = 0.80 0

**0.80 • 2 = 1.60 1**

0.60 • 2 = 1.20 1

0.20 • 2 = 0.40 0

0.40 • 2 = 0.80 0

**0.80 • 2 = 1.60 1** but we have seen this – see four lines above.

So decimal 0.20 in binary is 0.00110011001100110011 …, ad infinitum. This might   
be written conventionally as 0.00110 0110 0110 0110 0110, to emphasize the pattern.

The proof that no terminating representation exists depends on the fact that any terminating   
fraction in binary can be represented in the form  for some integers J and K. Thus we   
solve  or 5•J = 2K. This equation has a solution only if the right hand side is divisible   
by 5. But 2 and 5 are relatively prime numbers, so 5 does not divide any power of 2 and the   
equation has no integer solution. Hence 0.20 in decimal has no terminating form in binary.

Binary Addition

The next topic is storage of integers in a computer. We shall be concerned with storage of   
both positive and negative integers. Two’s complement arithmetic is the most common   
method of storing signed integers. Calculation of the two’s complement of a number   
involves binary addition. For that reason, we first discuss binary addition.

To motivate our discussion of binary addition, let us first look at decimal addition. Consider   
the sum 15 + 17 = 32. First, note that 5 + 7 = 12. In order to speak of binary addition, we   
must revert to a more basic way to describe 5 + 7; we say that the sum is 2 with a carry-out of   
1. Consider the sum 1 + 1, which is known to be 2. However, the correct answer to our   
simple problem is 32, not 22, because in computing the sum 1 + 1 we must consider the   
carry-in digit, here a 1. With that in mind, we show two addition tables – for a half-adder   
and a full-adder. The half-adder table is simpler as it does not involve a carry-in. The   
following table considers the sum and carry from A + B.

**Half-Adder A + B**

A B Sum Carry

0 0 0 0 Note the last row where we claim that 1 + 1 yields a

0 1 1 0 sum of zero and a carry of 1. This is similar to the

1 0 1 0 statement in decimal arithmetic that 5 + 5 yields a

1 1 0 1 sum of 0 and carry of 1 when 5 + 5 = 10.

Remember that when the sum of two numbers equals or exceeds the value of the base of the   
numbering system (here 2) that we decrease the sum by the value of the base and generate a   
carry. Here the base of the number system is 2 (decimal), which is 1 + 1, and the sum is 0.  
Say “One plus one equals two plus zero: 1 + 1 = 10”.

For us the half-adder is only a step in the understanding of a full-adder, which implements   
binary addition when a carry-in is allowed. We now view the table for the sum A + B, with   
a carry-in denoted by C. One can consider this A + B + C, if that helps.

**Full-Adder: A + B with Carry**

A B C Sum Carry

0 0 0 0 0

0 0 1 1 0

0 1 0 1 0

0 1 1 0 1

1 0 0 1 0

1 0 1 0 1

1 1 0 0 1

1 1 1 1 1

In the next chapter, we shall investigate the construction of a full adder from digital   
gates used to implement Boolean logic. Just to anticipate the answer, we note that the   
sum and carry table above is in the form of a Boolean truth table, which can be   
immediately converted to a Boolean expression that can be implemented in digital logic.

As an example, we shall consider a number of examples of addition of four-bit binary   
numbers. The problem will first be stated in decimal, then converted to binary, and then   
done. The last problem is introduced for the express purpose of pointing out an error.

We shall see in a minute that four-bit binary numbers can represent decimal numbers in   
the range 0 to 15 inclusive. Here are the problems, first in decimal and then in binary.

1) 6 + 1 0110 + 0001

2) 11 + 1 1011 + 0001

3) 13 + 5 1101 + 0101

**0110 1011 1101** In the first sum, we add 1 to an even number. This

**0001 0001 0101** is quite easy to do. Just change the last 0 to a 1.

**0111 1100 0010** Otherwise, we may need to watch the carry bits.

In the second sum, let us proceed from right to left. 1 + 1 = 0 with carry = 1. The second   
column has 1 + 0 with carry-in of 1 = 0 with carry-out = 1. The third column has 0 + 0   
with a carry-in of 1 = 1 with carry-out = 0. The fourth column is 1 + 0 = 1.

Analysis of the third sum shows that it is correct bit-wise but seems to be indicating that   
13 + 5 = 2. This is an example of “busted arithmetic”, more properly called overflow.   
A given number of bits can represent integers only in a given range; here 13 + 5 is outside   
the range 0 to 15 inclusive that is proper for four-bit unsigned numbers.

Signed and Unsigned Integers

Up to this point, we have discussed only unsigned integers and the conversion of such from   
one base to another. We now consider signed integers and find that consideration of the   
magnitude of such numbers as unsigned integers is an important part of the process.

Integers are stored in a number of formats. The most common formats today include 16 and   
32 bits, though long integers (64–bits) are becoming a standard integer format.   
Although 32-bit integers are probably the most common, our examples focus on eight–bit   
integers because they are easy to illustrate. In these discussions, the student should recall the   
powers of 2: 20 = 1, 21 = 2, 22 = 4, 23 = 8, 24 = 16, 25 = 32, 26 = 64, 27 = 128, and 28 = 256.

Bits in the storage of an integer are numbered right to left, with bit 0 being the right-most or   
least-significant. In eight bit integers, the bits from left to right are numbered 7 to 0. In 32   
bit integers, the bits from left to right are numbered 31 to 0. Note that this is not the notation   
used by IBM for its mainframe and enterprise computers. In the IBM notation, the most   
significant bit (often the sign bit) is bit 0 and the least significant bit has the highest number;   
bit 7 for an 8–bit integer. Here are the bit numberings for a signed 8–bit integer.

Common Notation (8–bit entry) IBM Mainframe Notation

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit # | 7 | 6 | 5 – 1 | 0 |  | Bit # | 0 | 1 | 2 – 6 | 7 |
|  | Sign | MSB |  | LSB |  | Sign | MSB |  | LSB |

The simplest topic to cover is the storage of **unsigned integers**. As there are 2N possible   
combinations of N binary bits, there are 2N unsigned integers ranging from 0 to 2N – 1. For   
eight-bit unsigned integers, this range is 0 though 255, as 28 = 256. Conversion from binary   
to decimal is easy and follows the discussion earlier in this chapter.

Of the various methods for storing **signed integers**, we shall discuss only three  
 Two’s complement  
 One’s complement (but only as a way to compute the two’s complement)  
 Excess 127 (for 8–bit numbers only) as a way to understand the floating point standard.

One’s complement arithmetic is mostly obsolete and interests us only as a stepping-stone to   
two’s complement arithmetic. To compute the one’s complement of a number:

1) Represent the number as an N-bit binary number  
 2) Convert every 0 to a 1 and every 1 to a 0.

Decimal 100 = **0110 0100**

One’s complement **1001 1011**; decimal –100 = **1001 1011** binary.

But consider the value 0 and problems representing it in one’s–complement notation.  
Decimal 0 = **0000 0000**

One’s complement **1111 1111**; decimal –0 = **1111 1111** binary.

There are a number of problems in one’s complement arithmetic, the most noticeable being   
illustrated by the fact that the one’s complement of 0 is 1111 1111. In this system, we have   
–0 ≠ 0, which is a violation of some of the basic principles of mathematics. When coding   
for a computer that used one’s–complement arithmetic, one had to write code such as:  
 **if ( (x == 0) || (x == –0) ) …**

**Notation:**

In discussing the one’s complement notation, we shall borrow some notation from digital   
logic. This notation will be fully discussed in a later chapter, but is quite easy to grasp.

We introduce the idea of logical NOT, as applied to binary values 0 and 1. Simply put  
 NOT(0) = 1  
 NOT(1) = 0.

The logical NOT function is denoted in three distinct ways, but here we focus on only one:   
the over–bar notation. Thus = 1 and = 0.

Let X represent an integer value. Note that the recipe for taking the one’s–complement of a   
number involves taking the logical NOT of each of its binary bits. For this reason, I propose   
to denote the one’s–complement of an integer X by , using the NOT symbol.

Let X = 100. Represented in binary, X = **0110 0100** The one’s complement is = **1001 1011**

Note that for addition X + = **1111 1111**. We shall say more on this later.

**The Two’s Complement**   
For integer arithmetic, all modern computers use two’s–complement notation. In this text,   
we shall reserve the standard symbol “–X” to denote the negative in the two’s–complement   
system. The two’s complement of a number is obtained as follows:

1) First take the one’s complement of the number

2) Add 1 to the one’s complement and discard the carry out of the left-most column.

Decimal 100 = **0110 0100**

One’s complement **1001 1011**

We now do the addition **1001 1011**  
  **1** **1001 1100**

Thus, in eight-bit two’s complement arithmetic

Decimal 100 = 0110 0100 binary  
 Decimal – 100 = 1001 1100 binary

This illustrates one pleasing feature of two’s complement arithmetic: for both positive and   
negative integers the last bit is zero if and only if the number is even. Note that it is essential   
to state how many bits are to be used. Consider the 8-bit two’s complement of 100. Now   
100 = 64 + 32 + 4, so decimal 100 = 0110 0100 binary, and we get the result above.

Consider decimal 12 = 0000 1100 binary. If we took the two’s complement of 1100, we   
might get 0100, giving us no idea how to pad out the high–order four bits.

The real reason for the popularity of two’s complement can be seen by calculating the   
representation of – 0. To do this we take the two’s complement of 0.

In eight bits, 0 is represented as **0000 0000**

Its one’s complement is represented as **1111 1111**.

We now take the two’s complement of 0.

Here is the addition  **1111 1111**  
  **1**  
 **1 0000 0000** – but discard the leading 1.

Thus the two’s complement of 0 is represented as **0000 0000.** As required by algebra,   
this is exactly the same as the representation of 0 and we avoid the messy problem – 0 ≠ 0.

In 8– bit two’s complement arithmetic, + 127 **0111 1111**  
the range of integers that can be + 10 **0000 1010**  
represented is – 2N-1 through 2N-1 – 1 +1 **0000 0001**  
inclusive, thus the range for eight-bit 0 **0000 0000**  
two’s complement integers is –128 – 1 **1111 1111** The number is  
through 127 inclusive, as 27 = 128. The – 10 **1111 0110** negative if and  
table at the right shows a number of – 127 **1000 0001** only if the left–  
binary representations for this example. – 128 **1000 0000** most bit is 1.

We now give the ranges allowed for the most common two’s complement representations.  
 Eight bit – 128 to +127  
 16-bit – 32,768 to +32,767  
 32-bit – 2,147,483,648 to +2,147,483,647

The range for 64-bit two’s complement integers is – 263 to 263 – 1. As an exercise in math,   
I propose to do a rough calculation of 263. This will be done using only logarithms.

There is a small collection of numbers that the serious student of computer science should   
memorize. Two of these numbers are the base-10 logarithms of 2 and 3. To five decimal   
places, log 2 = 0.30103 and log 3 = 0.47712.

Now 263 = (100.30103)63 = 1018.9649 = 100.9649 • 1018. What do we make of 100.9649? Since we   
have log 3 = 0.47712 and 9 = 32, we have log 9 = 2 • log 3 = 2 • 0.47712 = 0.95424. This   
leads to the conclusion that 100.9649 > 100.95424 ≈ 9; hence 9. < 100.95424 < 10.0. We conclude   
that a 64–bit integer allows the representation of 18 digit numbers and most 19 digit values.   
The precise range is –9,223,372,036,854,775,808 through 9,223,372,036,854,775,807.

**Reminder:** For any number of bits, in two’s complement arithmetic the number is negative if   
and only if the high-order bit in the binary representation is a 1.

Sign Extension

This applies to numbers represented in one’s-complement and two’s-complement form. The   
issue arises when we store a number in a form with more bits; for example when we store a   
16-bit integer in a 32-bit register. The question is how to set the high-order bits.

Consider a 16-bit integer stored in two’s-complement form. Bit 15 is the sign bit. We can   
consider bit representation of the number as A15A14A13A12A11A10A9A8A7A6A5A4A3A2A1A0.   
Consider placing this number into a 32-bit register with bits numbered R31 through R0, with   
R31 being the sign bit. Part of the solution is obvious: make Rk = Ak for 0 ≤ k ≤ 15. What is   
not obvious is how to set bits 31 through 16 in R as the 16-bit integer A has no such bits.

For non-negative numbers the solution is obvious and simple, set the extra bits to 0. This is   
like writing the number two hundred (200) as a five digit integer; write it 00200. But   
consider the 16-bit binary number **1111 1111 1000 0101**, decimal –123.   
If we expanded this to **0000 0000 0000 0000 1111 1111 1000 0101** by setting   
the high order bits to 0’s, we would have a positive number, 65413. This is not correct.

The answer to the problem is sign extension, which means filling the higher order bits of the   
bigger representation with the sign bit from the more restricted representation. In our   
example, we set bits 31 through 16 of the register to the sign bit of the 16-bit integer.   
The correct answer is then **1111 1111 1111 1111 1111 1111 1000 0101**.

**Note** – the way I got the value 1111 1111 1000 0101 for the 16-bit representation of – 123   
was to compute the 8-bit representation, which is **1**000 0101. The sign bit in this   
representation is 1, so I extended the number to 16-bits by setting the high order bits to 1.

Nomenclature: “Two’s-Complement Representation” vs. “Taking the Two’s-Complement”

We now address an issue that seems to cause confusion to some students. There is a   
difference between the idea of a complement system and the process of taking the   
complement. Because we are interested only in the two’s-complement system, I restrict my   
discussion to that system.

**Question:** What is the representation of the positive number 123 in 8-bit two’s complement   
arithmetic?

**Answer:** 0111 1011. Note that I did not take the two’s complement of anything to get this.

Two’s-complement arithmetic is a system of representing integers in which the two’s-  
complement is used to compute the negative of an integer. For positive integers, the method   
of conversion to binary differs from unsigned integers only in the representable range.

For N-bit unsigned integers, the range of integers representable is 0 ... 2N – 1, inclusive. For   
N-bit two’s-complement integers the range of non-negative integers representable is

0 ... 2N-1 – 1, inclusive. The rules for converting decimal to binary integers are the same for   
non-negative integers – one only has to watch the range restrictions.

The only time when one must use the fact that the number system is two’s-complement   
(that is – take the two’s-complement) is when one is asked about a negative number. Strictly   
speaking, it is not necessary to take the two’s-complement of anything in order to represent a   
negative number in binary, it is only that most students find this the easiest way.

**Question:** What is the representation of –123 in 8-bit two’s-complement arithmetic?

**Answer:** Perhaps I know that the answer is 1000 0101. As a matter of fact, I can calculate   
this result directly without taking the two’s-complement of anything, but most students find   
the mechanical way the easiest way to the solution. Thus, the preferred solution for most   
students is

1) We note that 0 ≤ 123 ≤ 27 – 1, so both the number and its negative  
 can be represented as an 8-bit two’s-complement integer.

2) We note that the representation of +123 in 8-bit binary is **0111 1011**

3) We take the two’s-complement of this binary result to get the binary  
 representation of –123 as **1000 0101**.

We note in passing a decidedly weird way to calculate the representations of non-negative   
integers in two’s-complement form. Suppose we want the two’s-complement representation   
of +123 as an eight-bit binary number. We could start with the fact that the representation of   
–123 in 8-bit two’s-complement is 1000 0101 and take the two’s complement of 1000 0101   
to obtain the binary representation of 123 = –(–123). This is perfectly valid, but decidedly   
strange. One could work this way, but why bother?

**Summary:** Speaking of the two’s-complement does not mean that one must take the  
 two’s-complement of anything.

**Why Does the Two’s Complement Work?**

We now ask an obvious question. The process of taking the two’s–complement of the binary   
representation of an integer has been described and is well defined. But why does this   
process yield the **negative** of the integer. We begin with a fact noted in passing just above

**Question:** Name the only number with the property that if I add 1 to it, I get 0.

**Answer:** That unique number is negative 1, denoted as –1.

Look now at simple addition for each of 8–bit and 16–bit signed integer values.

In 8–bit **1111 1111**  
 +  **1**  
 **0000 0000** Remember that the carry–out from the left bit is discarded.

In 16–bit **1111 1111 1111 1111**  
 +  **1**  
 **0000 0000 0000 0000** Again, the sum is 0.

From these two examples, it is possible to generalize to a statement that in any system in   
which the number 0 is represented as all bits being 0, a number with all bits 1 is –1.

We now consider the bitwise addition of a binary bit and its one’s–complement, also   
called NOT(a) or . Thus we compute the sum **a** +  for the two possible values of **a**.   
At the bit level, the addition table is simple, as the sum is always 1, without a carry.

**a**  **a +**  0 1 1  
 1 0 1

We now consider the sum of an arbitrary N–bit binary number and its one’s complement,   
recalling that taking the one’s complement is a bit–by–bit operation.

Let **A** be the N–bit binary number represented as **an–1an–2 … a2a1a0**.  
Let be the 1’s–complement of A, represented as .

Positional Notation   
We now review the idea of positional notation for binary representations of integers,   
beginning with N–bit unsigned integers. While the discussions are valid for all N > 0,   
we shall choose to illustrate with N = 8.

But we have shown that for every bit index that 

Thus, we have **A** + = –1. This should be clarified by a few examples. Note that in each   
example we add either 0 + 1 or 1 + 0, so there are no carry bits to consider.

Consider the 8–bit representation of the decimal number 100.  
Let **A** = **0110 0100**  
then = **1001 1011  
A** +  **= 1111 1111,** which represents –1.

Consider the 8–bit representation of the negative decimal number –123.

Let **A** = **1000 0101**  
then = **0111 1010  
A** +  **= 1111 1111,** which represents –1.

If we have **A** + = –1 for any binary value A, then we have **A** + ( **+ 1)** = 0.   
Hence ( **+ 1)** = –**A**, the negative of the value **A**. This is why two’s complement works.

Here is a table showing how to count up from the most negative integer.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Decimal | Positive | One’s Complement | Two’s Complement | Comment |
| 8 |  |  | 1000 | –8 |
| 7 | 0111 | 1000 | 1001 | –8 + 1 |
| 6 | 0110 | 1001 | 1010 | –8 + 2 |
| 5 | 0101 | 1010 | 1011 | –8 + 3 |
| 4 | 0100 | 1011 | 1100 | –8 + 4 |
| 3 | 0011 | 1100 | 1101 | –8 + 5 |
| 2 | 0010 | 1101 | 1110 | –8 + 6 |
| 1 | 0001 | 1110 | 1111 | –8 + 7 |

The above presents an interesting argument and proof, but it overlooks one essential point.   
How does the hardware handle this? For any sort of adder, the bits are just that. There is   
nothing special about the high–order bit. It is just another bit, and not interpreted in any   
special way. To the physical adder, the high–order is just another bit.

We shall discuss this problem when we present the logical design of a binary adder.

Arithmetic Overflow – “Busting the Arithmetic”

We continue our examination of computer arithmetic to consider one more topic – **overflow**.   
**Arithmetic overflow** occurs under a number of cases:

1) when two positive numbers are added and the result is negative  
 2) when two negative numbers are added and the result is positive  
 3) when a shift operation changes the sign bit of the result.

In mathematics, the sum of two negative numbers is always negative and the sum of two   
positive numbers is always positive. The overflow problem is an artifact of the limits on   
the range of integers and real numbers as stored in computers. We shall consider only   
overflows arising from integer addition.

For two’s-complement arithmetic, the range of storable integers is as follows:

16-bit – 215 to 215 – 1 or – 32768 to 32767

32-bit – 231 to 231 – 1 or – 2147483648 to 2147483647

In two’s-complement arithmetic, the most significant (left-most) bit is the sign bit

Overflow in addition occurs when two numbers, each with a sign bit of 0, are added and the   
sum has a sign bit of 1 or when two numbers, each with a sign bit of 1, are added and the sum   
has a sign bit of 0. For simplicity, we consider 16-bit addition. As an example, consider the   
sum 24576 + 24576 in both decimal and binary. Note 24576 = 16384 + 8192 = 214 + 213.

24576 **0110 0000 0000 0000**  
 24576 **0110 0000 0000 0000**  
 – 16384 **1100 0000 0000 0000**

In fact, 24576 + 24576 = 49152 = 32768 + 16384. The overflow is due to the fact that   
49152 is too large to be represented as a 16-bit signed integer.

As another example, consider the sum (–32768) + (–32768). As a 16–bit signed integer,   
the sum is 0!

–32768 **1000 0000 0000 0000**  
 –32768 **1000 0000 0000 0000**  
 0 **0000 0000 0000 0000**

It is easily shown that addition of a validly positive integer to a valid negative integer cannot   
result in an overflow. For example, consider again 16–bit two’s–complement integer   
arithmetic with two integers M and N. We have 0 ≤ M ≤ 32767 and –32768 ≤ N ≤ 0. If  
|M| ≥ |N|, we have 0 ≤ (M + N) ≤ 32767 and the sum is valid. Otherwise, we have   
–32768 ≤ (M + N) ≤ 0, which again is valid.

Integer overflow can also occur with subtraction. In this case, the two values (minuend  
 and subtrahend) must have opposite signs if overflow is to be possible.

Excess–127

We now cover **excess–127** representation. This is mentioned only because it is required   
when discussing the IEEE floating point standard. In general, we can consider an   
excess–M notation for any positive integer M. For an N–bit excess–M representation, the   
rules for conversion from binary to decimal are:

1) Evaluate as an unsigned binary number

2) Subtract M.

To convert from decimal to binary, the rules are

1) Add M

2) Evaluate as an unsigned binary number.

In considering excess notation, we focus on eight–bit excess–127 notation. The range of   
values that can be stored is based on the range that can be stored in the plain eight–bit   
unsigned standard: 0 through 255. Remember that in excess–127 notation, to store an integer   
N we first form the number N + 127. The limits on the unsigned eight-bit storage require   
that 0 ≤ (N + 127) ≤ 255, or – 127 ≤ N ≤ 128.

As an exercise, we note the eight-bit excess-127 representation of – 5, – 1, 0 and 4.

– 5 + 127 = 122. Decimal 122 = 0111 1010 binary, the answer.

– 1 + 127 = 126. Decimal 126 = 0111 1110 binary, the answer.

0 + 127 = 127. Decimal 127 = 0111 1111 binary, the answer.

4 + 127 = 131 Decimal 131 = 1000 0011 binary, the answer.

We have now completed the discussion of common ways to represent unsigned and signed   
integers in a binary computer. We now start our progress towards understanding the storage   
of real numbers in a computer. There are two ways to store real numbers – fixed point and   
floating point. We focus this discussion on floating point, specifically the IEEE standard for   
storing floating point numbers in a computer.

Normalized Numbers

The last topic to be discussed prior to defining the IEEE standard for floating point numbers   
is that of normalized numbers. We must also mention the concept of denormalized numbers,   
though we shall spend much less time on the latter.

A normalized number is one with a representation of the form X • 2P, where 1.0 ≤ X < 2.0.   
At the moment, we use the term denormalized number to mean a number that cannot be so   
represented, although the term has a different precise meaning in the IEEE standard. First,   
we ask a question: **“What common number cannot be represented in this form?”**

The answer is **zero**. There is no power of 2 such that 0.0 = X • 2P, where 1.0 ≤ X < 2.0. We   
shall return to this issue when we discuss the IEEE standard, at which time we shall give a   
more precise definition of the denormalized numbers, and note that they include 0.0. For the   
moment, we focus on obtaining the normalized representation of positive real numbers.

We start with some simple examples.

1.0 = 1.0 • 20, thus X = 1.0 and P = 0.

1.5 = 1.5 • 20, thus X = 1.5 and P = 0.

2.0 = 1.0 • 21, thus X = 1.0 and P = 1

0.25 = 1.0 • 2-2, thus X = 1.0 and P = -2

7.0 = 1.75 • 22, thus X = 1.75 and P = 2

0.75 = 1.5 • 2-1, thus X = 1.5 and P = -1.

To better understand this conversion, we shall do a few more examples using the more   
mechanical approach to conversion of decimal numbers to binary. We start with an example:   
9.375 • 10-2 = 0.09375. We now convert to binary.

0.09375 • 2 = 0.1875 0  
0.1875 • 2 = 0.375 0 Thus decimal 0.09375 = 0.00011 binary  
0.375 • 2 = 0.75 0 or 1.1 • 2-4 in the normalized notation.  
0.75 • 2 = 1.5 1  
0.5 • 2 = 1.0 1

Please note that these representations take the form X • 2P, where X is represented as a   
binary number but P is represented as a decimal number. Later, P will be converted to an   
excess–127 binary representation, but for the present it is easier to keep it in decimal.

We now convert the decimal number 80.09375 to binary notation. I have chosen 0.09375 as   
the fractional part out of laziness as we have already obtained its binary representation. We   
now convert the number 80 from decimal to binary. Note 80 = 64 + 16 = 26 • (1 + ¼).

80 / 2 = 40 remainder 0  
40/2 = 20 remainder 0

20 / 2 = 10 remainder 0

10 / 2 = 5 remainder 0

5 / 2 = 2 remainder 1

2 / 2 = 1 remainder 0

1 / 2 = 1 remainder 1

Thus decimal 80 = 1010000 binary and decimal 80.09375 = 1010000.00011 binary. To get   
the binary point to be after the first 1, we move it six places to the left, so the normalized   
form of the number is 1.01000000011 • 26, as expected. For convenience, we write this as   
1.0100 0000 0110 • 26.

Extended Example: Avogadro’s Number.

Up to this point, we have discussed the normalized representation of positive real numbers   
where the conversion from decimal to binary can be done exactly for both the integer and   
fractional parts. We now consider conversion of very large real numbers in which it is not   
practical to represent the integer part, much less convert it to binary.

We now discuss a rather large floating point number: 6.023 • 1023. This is Avogadro’s   
number. We shall convert this to normalized form and use the opportunity to discuss a   
number of issues associated with floating point numbers in general.

Avogadro’s number arises in the study of chemistry. This number relates the atomic weight   
of an element to the number of atoms in that many grams of the element. The atomic weight   
of oxygen is 16.00, as a result of which there are about 6.023 • 1023 atoms in 16 grams of   
oxygen. For our discussion we use a more accurate value of 6.022142 • 1023 obtained from   
the web sit of the National Institute of Standards ([www.nist.gov](http://www.nist.gov)).

We first remark that the number is determined by experiment, so it is not known exactly. We   
thus see one of the main scientific uses of this notation – to indicate the precision with which   
the number is known. The above should be read as (6.022142 ± 0.0000005) • 1023, that is to   
say that the best estimate of the value is between 6.0221415 • 1023 and 6.0221425 • 1023, or   
between 602, 214, 150, 000, 000, 000, 000, 000 and 602, 214, 250, 000, 000, 000, 000, 000.   
Here we see another use of scientific notation – not having to write all these zeroes.

Again, we use logarithms and anti-logarithms to convert this number to a power of two. The   
first question is how accurately to state the logarithm. The answer comes by observing that   
the number we are converting is known to seven digit’s precision. Thus, the most accuracy   
that makes sense in the logarithm is also seven digits.

In base-10 logarithms log(6.022142 • 1023) = 23.0 + log(6.022142). To seven digits, this last   
number is 0.7797510, so log(6.022142 • 1023) = 23.7797510.

We now use the fact that log(2.0) = 0.3010300 to seven decimal places to solve the equation  
 2X = (100.3010300)X = 10.23.7797520 or 0.30103•X = 23.7797510 for X = 78.9946218.

If we use NA to denote Avogadro’s number, the first thing we have discovered from this   
tedious analysis is that 278 < NA < 279, and that NA ≈ 279. The representation of the number in   
normal form is thus of the form 1.f • 278, where the next step is to determine f. To do this,   
we obtain the decimal representation of 278.

Note that 278 = (100.30103)78 = 1023.48034 = 100.48034 • 1023 = 3.022317 • 1023.  
But 6.022142 / 3.022317 = 1.992558, so NA = 1.992558 • 278, and f = 0.992558.

To complete this problem, we obtain the binary equivalent of 0.992558.

0.992558 • 2 = 1.985116 1  
0.985116 • 2 = 1.970232 1  
0.970232 • 2 = 1.949464 1  
0.949464 • 2 = 1.880928 1  
0.880928 • 2 = 1.761856 1  
0.761856 • 2 = 1.523712 1  
0.523712 • 2 = 1.047424 1

0.047424 • 2 = 0.094848 0  
0.094848 • 2 = 0.189696 0  
0.189696 • 2 = 0.379392 0  
0.379392 • 2 = 0.758784 0

0.758784 • 2 = 1.517568 1

The desired form is 1.1111 1110 0001 • 278.

# IEEE Standard 754 Floating Point Numbers

There are two primary formats in the IEEE 754 standard; **single precision** and **double**   
**precision**. We shall study the single precision format.

The single precision format is a 32–bit format. From left to right, we have  
 1 sign bit; 1 for negative and 0 for non-negative  
 8 exponent bits  
 23 bits for the fractional part of the mantissa.

The eight-bit exponent field stores the exponent of 2 in excess-127 form, with the   
exception of two special bit patterns.

0000 0000 numbers with these exponents are denormalized  
 1111 1111 numbers with these exponents are infinity or Not A Number

Before presenting examples of the IEEE 754 standard, we shall examine the concept of NaN   
or Not a Number. In this discussion, we use some very imprecise terminology.

Consider the quotient 1/0. The equation 1 / 0 = X is equivalent to solving for a number X   
such that 0 • X = 1. There is no such number X such that the result of multiplying it by 0   
yields a 1. Loosely speaking, we say 1 / 0 = ∞; precisely speaking we use the notations of   
the calculus and speak of limits to the fraction 1 / Y as the number Y approaches 0.

Now consider the quotient 0/0. Again we are asking for the number X such that 0 • X = 0.   
The difference here is that this equation is true for every number X. In terms of the IEEE   
standard, 0 / 0 is Not a Number, or NaN.

The number NaN can also be used for arithmetic operations that have no solutions, such as   
taking the square root of –1 while limited to the real number system. While this result   
cannot be represented, it is definitely neither +∞ nor –∞ .

We now illustrate the standard by doing some conversions.

For the first example, consider the number –0.75.

To represent the number in the IEEE standard, first note that it is negative so that the sign bit   
is 1. Having noted this, we convert the number 0.75.

0.75 • 2 = 1.5 1

0.5 • 2 = 1.0 1

Thus, the binary equivalent of decimal 0.75 is 0.11 binary. We must now convert this into   
the normalized form 1.10 • 2–1. Thus we have the key elements required.

The power of 2 is –1, stored in Excess-127 as 126 = **0111 1110** binary.

The fractional part is 10, possibly best written as 10000

Recalling that the sign bit is 1, we form the number as follows:

**1 0111 1110 10000**

We now group the binary bits by fours from the left until we get only 0’s.

**1011 1111 0100 0000**

Since trailing zeroes are not significant in fractions, this is equivalent to

**1011 1111 0100 0000 0000 0000 0000 0000**

or **BF40 0000** in hexadecimal.

As another example, we revisit a number converted earlier. We have shown that

80.09375 = 1.0100 0000 0110 • 26. This is a positive number, so the sign bit is 0. As an   
Excess–127 number, 6 is stored as 6 + 127 = 133 = 1000 0101 binary. The fractional part   
of the number is 0100 0000 0110 0000, so the IEEE representation is

**0 1000 0101 0100 0000 0110 0000**

Regrouping by fours from the left, we get the following

**0100 0010 1010 0000 0011 0000**

In hexadecimal this number is **42A030**, or **42A0 3000** as an eight digit hexadecimal   
number. Note that all single–precision values should be represented with 8 hex digits.

Here is the general rule for floating–point numbers:  
 single precision 32 bit values represent with 8 hexadecimal digits   
 double precision 64 bit values represent with 16 hex digits.

Some Examples “In Reverse”

We now consider another view on the IEEE floating point standard – the “reverse” view.   
We are given a 32-bit number, preferably in hexadecimal form, and asked to produce the   
floating-point number that this hexadecimal string represents. As always, in interpreting any   
string of binary characters, we must be told what standard to apply – here the IEEE-754 single precision standard.

First, convert the following 32-bit word, represented by eight hexadecimal digits, to the   
floating-point number being represented.

**0000 0000** // Eight hexadecimal zeroes representing 32 binary zeroes

The answer is **0.0**. This is a result that should be memorized.

The question in the following paragraph was taken from a mid–term exam for an   
introductory course in computer organization. The paragraphs following were taken   
from the answer key for that exam.

Give the value of the real number (in standard decimal representation) represented by   
the following 32-bit words stored as an IEEE standard single precision.  
 a) **4068 0000**  
 b) **42E8 0000**  
 c) **C2E8 0000**  
 d) **C380 0000**

e) **C5FC 0000**

The first step in solving these problems is to convert the hexadecimal to binary.

a) **4068 0000** = **0100 0000 0110 1000 0000 0000 0000 0000**  
Regroup to get **0 1000 0000 1101 0000** etc. (trailing zeroes can be ignored)

Thus s = 0 (not a negative number)  
 p + 127 = 100000002 = 12810, so p = 1  
and m = 1101, so 1.m = 1.1101 and the number is 1.1101•21 = 11.1012.

But 11.1012 = 2 + 1 + 1/2 + 1/8 = 3 + 5/8 = **3.625**.

b) **42E8 0000** = **0100 0010 1110 1000 0000 0000 0000 0000**  
Regroup to get **0 1000 0101 1101 0000** etc  
Thus s = 0 (not a negative number)  
 p + 127 = 100001012 = 128 + 4 + 1 = 133, hence p = 6  
and m = 1101, so 1.m = 1.1101 and the number is 1.1101•26 = 11101002   
But 11101002 = 64 + 32 + 16 + 4 = 96 + 20 = 116 = **116.0**

c) **C2E80 0000** = **1100 0010 1110 1000 0000 0000 0000 0000**  
Regroup to get **1 1000 0101 1101 0000** etc.  
Thus s = 1 (a negative number) and the rest is the same as b). So **– 116.0**

d) **C380 0000** = **1100 0011 1000 0000 0000 0000 0000 0000**

Regroup to get **1 1000 0111 0000 0000 0000 0000 0000 000**

Thus s = 1 (a negative number)  
 p + 127 = 100001112 = 128 + 7 = 135; hence p = 8.  
 m = 0000, so 1.m = 1.0 and the number is – 1.0 • 28 = **– 256.0**

e) **C5FC 0000** = **1100 0101 1111 1100 0000 0000 0000 0000**  
Regroup to get **1 1000 1011 1111 1000 0000 0000 0000 000**  
Thus s = 1 (a negative number)  
 p + 127 = 1000 10112 = 128 + 8 + 2 + 1 = 139, so p = 12  
 m = 1111 1000, so 1.m = 1.1111 1000

There are three ways to get the magnitude of this number. The magnitude can be written   
in normalized form as 1.1111 1000 • 212 = 1.1111 1000 • 4096, as 212 = 4096.

**Method 1**

If we solve this the way we have, we have to place four extra zeroes after the decimal point   
to get the required 12, so that we can shift the decimal point right 12 places.

1.1111 1000 • 212 = 1.1111 1000 0000 • 212 = 1 1111 1000 00002  
 = 212 + 211+ 210 + 29 + 28 + 27  
 = 4096 + 2048 + 1024 + 512 + 256 + 128 = 8064.

**Method 2**

We shift the decimal place only 5 places to the right (reducing the exponent by 5) to get  
 1.1111 1000 • 212 = 1 1111 1.0 • 27  
 = (25 + 24 + 23 + 22 + 21 + 20) • 27  
 = (32 + 16 + 8 + 4 + 2 + 1) • 128 = 63 • 128 = 8064.

**Method 3**  
This is an offbeat method, not much favored by students.

1.1111 1000 • 212 = (1 + 2–1 + 2–2+ 2–3+ 2–4+ 2–5) • 212  
 = 212 + 211+ 210 + 29 + 28 + 27  
 = 4096 + 2048 + 1024 + 512 + 256 + 128 = 8064.

**Method 4**  
This is another offbeat method, not much favored by students.

1.1111 1000 • 212 = (1 + 2–1 + 2–2+ 2–3+ 2–4+ 2–5) • 212  
 = (1 + 0.5 + 0.25 + 0.125 + 0.0625+ 0.03125) • 4096  
 = 1.96875 • 4096 = 8064.

The answer is **– 8064.0**.

As a final example, we consider the IEEE standard representation of Avogadro’s number.   
We have seen that NA ≈ 1.1111 1110 0001 • 278. This is a positive number; the sign bit is 0.

We now consider the representation of the exponent 78. Now 78 + 127 = 205, so the   
Excess-127 representation of 78 is 205 = 128 + 77 = 128 + 64 + 13 = 128 + 64 + 8 + 4 + 1.   
As an 8-bit binary number this is 1100 1101. We already have the fractional part, so we get

**0 1100 1101 1111 1110 0001 0000**

Grouped by fours from the left we get

**0110 0110 1111 1111 0000 1000 0000 0000**,   
or **66FF 0800** in hexadecimal.

# Range and Precision

We now consider the range and precision associated with the IEEE single precision standard   
using normalized numbers.. The range refers to the smallest and largest positive numbers   
that can be stored. Recalling that zero is not a positive number, we derive the smallest and   
largest representable numbers.

In the binary the smallest normalized number is 1.0 • 2–126 and the largest number is a bit   
less than 2.0 • 2127 = 2128. Again, we use logarithms to evaluate these numbers.  
 – 126 • 0.30103 = – 37.93 = – 38.0 + 0.07, so 2–126 = 1.07 • 10 -38, approximately.

128 • 0.30103 = 38.53, so 2128 = 3.5 • 10 38, as 100.53 is a bit bigger than 3.2.

We now consider the precision associated with the standard. Consider the decimal notation   
1.23. The precision associated with this is ± 0.005 as the number really represents a value   
between 1.225 and 1.235 respectively.

The IEEE standard has a 23-bit fraction. Thus, the precision associated with the standard is   
1 part in 224 or 1 part in 16 • 220 = 16 • 1048576 = 16777216. This accuracy is more precise   
than 1 part in 107, or seven digit precision.

# Denormalized Numbers

We shall see in a bit that the range of normalized numbers is approximately 10 –38 to 10 38.   
We now consider what we might do with a problem such as the quotient 10 –20 / 10 30. In   
plain algebra, the answer is simply 10 –50, a very small positive number. But this number is   
smaller than allowed by the standard. We have two options for representing the quotient,   
either 0.0 or some strange number that clearly indicates the underflow. This is the purpose of   
denormalized numbers – to show that the result of an operation is positive but too small   
to be represented in standard format.

Why Excess–127 Notation for the Exponent?

We have introduced two methods to be used for storing signed integers – two’s-complement   
notation and excess–127 notation. One might well ask why two’s-complement notation is   
used to store signed integers while the excess–127 method is used for exponents in the   
floating point notation.

The answer for integer notation is simple. It is much easier to build an adder for integers   
stored in two’s-complement form than it is to build an adder for integers in the excess   
notation. In the next chapter we shall investigate a two’s-complement adder.

So, why use excess–127 notation for the exponent in the floating point representation? The   
answer is best given by example. Consider some of the numbers we have used as examples.

**0011 1111 0100 0000 0000 0000 0000 0000** for 0.75

**0100 0010 1010 0000 0011 0000 0000 0000** for 80.09375

**0110 0110 1111 1111 0000 1000 0000 0000** for Avagadro’s number.

It turns out that the excess–127 notation allows the use of the integer compare unit to   
compare floating point numbers. Consider two floating point numbers X and Y. Pretend that   
they are integers and compare their bit patterns as integer bit patterns. It viewed as an   
integer, X is less than Y, then the floating point number X is less than the floating point Y.   
Note that we are not converting the numbers to integer form, just looking at the bit patterns   
and pretending that they are integers.

Were the exponents stored in two’s–complement notation, we would have  
**0111 1111 1100 0000 0000 0000 0000 0000** for 0.75

**0000 0011 0010 0000 0011 0000 0000 0000** for 80.09375

Comparing these as if they were integers makes the value 0.75 appear to be larger. Recall   
that it is the leftmost bit that is the sign bit in any standard numeric representation.  
Pretending that each of these values is an integer makes both positive numbers.

Floating Point Equality: X == Y

Due to round off error, it is sometimes not advisable to check directly for equality of floating   
point numbers. A better method would be to use an acceptable relative error. We borrow the   
notation **ε** from calculus to stand for a small number, and use the notation |Z| for the absolute   
value of the number Z.

Here are two valid alternatives to the problematic statement (**X == Y**).

1) Absolute difference |X – Y| ≤ ε

2) Relative difference |X – Y| ≤ ε•(|X| +|Y|)

Note that this form of the second statement is preferable to computing the quotient  
|X – Y| / (|X| +|Y|) which will be NaN (Not A Number) if X = 0.0 and Y = 0.0.

Bottom Line: In your coding with real numbers, decide what it means for two numbers to be   
equal. How close is close enough? There are no general rules here, only cautions. It is   
interesting to note that one language (SPARK, a variant of the Ada programming language)   
does not allow floating point comparison statements such as **X == Y**, but demands an   
evaluation of the absolute value of the difference between X and Y.

The IBM Mainframe Floating–Point Formats

In this discussion, we shall adopt the bit numbering scheme used in the IBM documentation,   
with the leftmost (sign) bit being number 0. The IBM Mainframe supports three formats;   
those representations with more bits can be seen to afford more precision.

Single precision 32 bits numbered 0 through 31,  
 Double precision 64 bits numbered 0 through 63, and  
 Extended precision 128 bits numbered 0 through 127.

As in the IEEE–754 standard, each floating point number in this standard is specified by   
three fields: the sign bit, the exponent, and the fraction. Unlike the IEEE–754 standard, the   
IBM standard allocates the same number of bits for the exponent of each of its formats.   
The bit numbers for each of the fields are shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| Format | Sign bit | Bits for exponent | Bits for fraction |
| Single precision | 0 | 1 – 7 | 8 – 31 |
| Double precision | 0 | 1 – 7 | 8 – 63 |
| Extended precision | 0 | 1 – 7 | 8 – 127 |

Note that each of the three formats uses eight bits to represent the exponent, in what is called   
the **characteristic field**, and the sign bit. These two fields together will be represented by   
two hexadecimal digits in a one–byte field.

The size of the fraction field does depend on the format.  
 Single precision 24 bits 6 hexadecimal digits,  
 Double precision 56 bits 14 hexadecimal digits, and  
 Extended precision 120 bits 30 hexadecimal digits.

The Characteristic Field

In IBM terminology, the field used to store the representation of the exponent is called the   
**“characteristic”**. This is a 7–bit field, used to store the exponent in excess–64 format; if   
the exponent is E, then the value (E + 64) is stored as an unsigned 7–bit number.

Recalling that the range for integers stored in 7–bit unsigned format is 0 ≤ N ≤ 127, we have   
0 ≤ (E + 64) ≤ 127, or –64 ≤ E ≤ 63.

Range for the Standard

We now consider the range and precision associated with the IBM floating point formats.   
The reader should remember that the range is identical for all of the three formats; only the   
precision differs. The range is usually specified as that for positive numbers, from a very   
small positive number to a large positive number. There is an equivalent range for negative   
numbers. Recall that 0 is not a positive number, so that it is not included in either range.

Given that the base of the exponent is 16, the range for these IBM formats is impressive. It   
is from somewhat less than 16–64 to a bit less than 1663. Note that 1663 = (24)63 = 2252, and   
16–64 = (24)–64 = 2–256 = 1.0 / (2256) and recall that log10(2) = 0.30103. Using this, we compute   
the maximum number storable at about (100.30103)252 = 1075.86 ≈ 9•1075. We may approximate   
the smallest positive number at 1.0 / (36•1075) or about 3.0•10–77. In summary, the following   
real numbers can be represented in this standard: X = 0.0 and 3.0•10–77 < X < 9•1075.

One would not expect numbers outside of this range to appear in any realistic calculation.

Precision for the Standard

Unlike the range, which depends weakly on the format, the precision is very dependent on   
the format used. More specifically, the precision is a direct function of the number of bits   
used for the fraction. If the fraction uses F bits, the precision is 1 part in 2F.

We can summarize the precision for each format as follows.  
 Single precision F = 24 1 part in 224.  
 Double precision F = 56 1 part in 256.  
 Extended precision F = 120 1 part in 2120.

The first power of 2 is easily computed; we use logarithms to approximate the others.  
 224 = 16,777,216   
 256 ≈ (100.30103)56 = 1016.85 ≈ 9•1016.  
 2120 ≈ (100.30103)120 = 1036.12 ≈ 1.2•1036.

The argument for precision is quite simple. Consider the single precision format, which is   
more precise than 1 part in 10,000,000 and less precise than 1 part in 100,000,000. In other   
words it is better than 1 part in 107, but not as good as 1 in 108; hence we say 7 digits.

Range and Precision

We now summarize the range and precision for the three IBM Mainframe formats.

|  |  |  |
| --- | --- | --- |
| **Format** | **Positive Range** | **Precision** |
| Single Precision | 3.0•10–77 < X < 9•1075 | 7 digits |
| Double Precision | 3.0•10–77 < X < 9•1075 | 16 digits |
| Extended Precision | 3.0•10–77 < X < 9•1075 | 36 digits |

Representation of Floating Point Numbers

As with the case of integers, we shall most commonly use hexadecimal notation to represent   
the values of floating–point numbers stored in the memory. From this point, we shall focus   
on the two more commonly used formats: Single Precision and Double Precision.

The single precision format uses a 32–bit number, represented by 8 hexadecimal digits.  
The double precision format uses a 64–bit number, represented by 16 hexadecimal digits.

Due to the fact that the two formats use the same field length for the characteristic,   
conversion between the two is quite simple. To convert a single precision value to a double   
precision value, just add eight hexadecimal zeroes.

Consider the positive number 128.0.  
As a single precision number, the value is stored as **4280 0000**.  
As a double precision number, the value is stored as **4280 0000 0000 0000**.

Conversions from double precision to single precision format will involve some rounding.  
For example, consider the representation of the positive decimal number 123.45. In a few   
pages, we shall show that it is represented as follows.

As a double precision number, the value is stored as **427B 7333 3333 3333**.  
As a single precision number, the value is stored as **427B 7333**.

The Sign Bit and Characteristic Field   
We now discuss the first two hexadecimal digits in the representation of a floating–point   
number in these two IBM formats. In IBM nomenclature, the bits are allocated as follows.  
 Bit 0 the sign bit  
 Bits 1 – 7 the seven–bit number storing the characteristic.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Hex digit | 0 | | | | 1 | | | |
| Use | Sign bit | Characteristic (Exponent + 64) | | | | | | |

Consider the four bits that comprise hexadecimal digit 0. The sign bit in the floating–point   
representation is the “8 bit” in that hexadecimal digit. This leads to a simple rule.

If the number is not negative, bit 0 is 0, and hex digit 0 is one of 0, 1, 2, 3, 4, 5, 6, or 7.  
If the number is negative, bit 0 is 1, and hex digit 0 is one of 8, 9, A, B, C, D, E, or F.

Some Single Precision Examples   
We now examine a number of examples, using the IBM single–precision floating–point   
format. The reader will note that the methods for conversion from decimal to hexadecimal   
formats are somewhat informal, and should check previous notes for a more formal method.   
Note that the first step in each conversion is to represent the **magnitude** of the number in the   
required form X•16E, after which we determine the sign and build the first two hex digits.

**Example 1: Positive exponent and positive fraction.**

The decimal number is 128.50. The format demands a representation in the form X•16E,   
with 0.625 ≤ X < 1.0. As 128 ≤ X < 256, the number is converted to the form X•162.  
Note that 128 = (1/2)•162 = (8/16)•162 , and 0.5 = (1/512)•162 = (8/4096)•162.  
Hence, the value is 128.50 = (8/16 + 0/256 + 8/4096)•162; it is 162•0x0.808.

The exponent value is 2, so the characteristic value is either 66 or 0x42 = 100 0010.   
The first two hexadecimal digits in the eight digit representation are formed as follows.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Field | Sign | Characteristic | | | | | | |
| Value | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| Hex value | 4 | | | | 2 | | | |

The fractional part comprises six hexadecimal digits, the first three of which are 808.  
The number 128.50 is represented as **4280 8000**.

**Example 2: Positive exponent and negative fraction.**

The decimal number is the negative number –128.50. At this point, we would normally   
convert the magnitude of the number to hexadecimal representation. This number has the   
same magnitude as the previous example, so we just copy the answer; it is 162•0x0.808.

We now build the first two hexadecimal digits, noting that the sign bit is 1.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Field | Sign | Characteristic | | | | | | |
| Value | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| Hex value | C | | | | 2 | | | |

The number 128.50 is represented as **C280 8000**.  
Note that we could have obtained this value just by adding 8 to the first hex digit.

**Example 3: Negative exponent and positive fraction.**

The decimal number is 0.375. As a fraction, this is 3/8 = 6/16. Put another way, it is   
160•0.375 = 160•(6/16). This is in the required format X•16E, with 0.625 ≤ X < 1.0.

The exponent value is 0, so the characteristic value is either 64 or 0x40 = 100 0000.   
The first two hexadecimal digits in the eight digit representation are formed as follows.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Field | Sign | Characteristic | | | | | | |
| Value | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Hex value | 4 | | | | 0 | | | |

The fractional part comprises six hexadecimal digits, the first of which is a 6.  
The number 0.375 is represented in single precision as **4060 0000**.  
The number 0.375 is represented in double precision as **4060 0000 0000 0000**.

**Example 4: A Full Conversion**  
The number to be converted is 123.45. As we have hinted, this is a non–terminator.

Convert the integer part.  
**123 / 16 = 7** with remainder **11** this is hexadecimal digit B.  
 **7 / 16 = 0** with remainder  **7** this is hexadecimal digit 7.  
Reading bottom to top, the integer part converts as 0x7B.

Convert the fractional part.  
**0.45 • 16 = 7.20** Extract the 7,  
**0.20 • 16 = 3.20** Extract the 3,  
**0.20 • 16 = 3.20** Extract the 3,  
**0.20 • 16 = 3.20** Extract the 3, and so on.

In the standard format, this number is 162•0x0.7B33333333…...

The exponent value is 2, so the characteristic value is either 66 or 0x42 = 100 0010.   
The first two hexadecimal digits in the eight digit representation are formed as follows.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Field | Sign | Characteristic | | | | | | |
| Value | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| Hex value | 4 | | | | 2 | | | |

The number 123.45 is represented in single precision as **427B 3333**.  
The number 0.375 is represented in double precision as **427B 3333 3333 3333**.

**Example 5: One in “Reverse”**We are given the single precision representation of the number. It is **4110 0000**.  
What is the value of the number stored? We begin by examination of the first two hex digits.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Field | Sign | Characteristic | | | | | | |
| Value | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| Hex value | 4 | | | | 1 | | | |

The sign bit is 0, so the number is positive. The characteristic is 0x41, so the exponent is  
1 and the value may be represented by X•161. The fraction field is **100 000**, so the value is  
161•(1/16) = 1.0.

Packed Decimal Formats

While the IBM mainframe provides three floating–point formats, it also provides another   
format for use in what are called “fixed point” calculations. The term “fixed point” refers to   
decimal numbers in which the decimal point takes a predictable place in the number; money   
transactions in dollars and cents are a good and very important example of this.

Consider a ledger such as might be maintained by a commercial firm. This contains credits   
and debits, normally entered as money amounts with dollars and cents. The amount that   
might be printed as “$1234.56” could easily be stored as the integer 123456 if the program   
automatically adjusted to provide the implicit decimal point. This fact is the basis for the   
Packed Decimal Format developed by IBM in response to its business customers.

One may well ask “Why not use floating point formats for financial transactions?”. We   
present a fairly realistic scenario to illustrate the problem with such a choice. This example   
is based on your author’s experience as a consultant to a bank in Rochester, NY.

It is a fact that banks loan each other money on an overnight basis; that is, the bank borrows   
the money at 6:00 PM today and repays it at 6:00 AM tomorrow. While this may seem a bit   
strange to those of us who think in terms of 20–year mortgages, it is an important practice.

Overnight loans in the amount of one hundred million dollars are not uncommon.

Suppose that I am a bank officer, and that another bank wants to borrow $100,000,000   
overnight. I would like to make the loan, but do not have the cash on hand. On the other   
hand, I know a bank that will lend me the money at a smaller interest rate. I can make the   
loan and pocket the profit.

Suppose that the borrowing bank is willing to pay 8% per year on the borrowed amount.   
This corresponds to a payback of (1.08)1/730 = 1.0001054, which is $10,543 in interest.

Suppose that I have to borrow the money at 6% per annum. This corresponds to my paying   
at a rate of (1.06)1/730 = 1.0000798, which is a cost of $7,982 to me. I make $2,561.

Consider these numbers as single–precision floating point format in the IBM Mainframe.

My original money amount is $100,000,000

The interest I make is $10,543

My principal plus interest is $100,010,500 Note the truncation due to precision.  
The interest I pay is $7,982  
What I get back is $100,002,000 Again, note the truncation.

The use of floating–point arithmetic has cost me $561 for an overnight transaction. I do not   
like that. I do not like numbers that are rounded off; I want precise arithmetic.

Almost all banks and financial institutions demanded some sort of precise decimal   
arithmetic; IBM’s answer was the Packed Decimal format.

BCD (**B**inary **C**oded **D**ecimal)

The best way to introduce the Packed Decimal Data format is to first present an earlier   
format for encoding decimal digits. This format is called **BCD**, for “Binary Coded Decimal”.   
As may be inferred from its name, it is a precursor to EBCDIC (Extended BCD Interchange   
Code) in addition to heavily influencing the Packed Decimal Data format.

We shall introduce BCD and compare it to the 8–bit unsigned binary previously discussed for   
storing unsigned integers in the range 0 through 255 inclusive. While BCD doubtless had   
encodings for negative numbers, we shall postpone signed notation to Packed Decimal.

The essential difference between BCD and 8–bit binary is that BCD encodes each decimal in   
a separate 4–bit field (sometimes called “nibble” for half–byte). This contrasts with the usual   
binary notation in which it is the magnitude of the number (and not the number of digits) that   
determines whether or not it can be represented in the format.

We begin with a table of the BCD codes for each of the ten decimal digits. These codes are   
given in both binary and hexadecimal. It will be important for future discussions to note that   
these encodings are actually hexadecimal digits; they just appear to be decimal digits.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Digit | ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ |
| Binary | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 |
| Hexadecimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

To emphasize the difference between 8–bit unsigned binary and BCD, we shall examine a   
selection of two–digit numbers and their encodings in each system.

|  |  |  |  |
| --- | --- | --- | --- |
| Decimal Number | 8–bit binary | BCD (Represented in binary) | BCD (hexadecimal) |
| 5 | 0000 0101 | 0000 0101 | 05 |
| 13 | 0000 1101 | 0001 0011 | 13 |
| 17 | 0001 0001 | 0001 0111 | 17 |
| 23 | 0001 0111 | 0010 0011 | 23 |
| 31 | 0001 1111 | 0011 0001 | 31 |
| 64 | 0100 0000 | 0110 0100 | 64 |
| 89 | 0101 1001 | 1000 1001 | 89 |
| 96 | 0110 0000 | 1001 0110 | 96 |

As a hypothetical aside, consider the storage of BCD numbers on a byte–addressable   
computer. The smallest addressable unit would be an 8–bit byte. As a result of this, all BCD   
numbers would need to have an even number of digits, as to fill up an integral number of   
bytes. Our solution to the storage of integers with an odd number of digits is to recall that a   
leading zero does not change the value of the integer.

In this hypothetical scheme of storage:  
 1 would be stored as 01,  
 22 would be stored as 22,  
 333 would be stored as 0333,  
 4444 would be stored as 4444,  
 55555 would be stored as 055555, and  
 666666 would be stored as 666666.

Packed Decimal Data

The packed decimal format should be viewed as a generalization of the BCD format with the   
specific goal of handling the fixed point arithmetic so common in financial transactions. The   
two extensions of the BCD format are as follows:  
 1. The provision of a sign “half byte” so that negative numbers can be handled.  
 2. The provision for variable length strings.

While the term “fixed point” is rarely used in computer literature these days, the format is   
very common. Consider any transaction denominated in dollars and cents. The amount will   
be represented as a real number with exactly two digits to the right of the decimal point; that   
decimal point has a fixed position in the notation, hence the name “fixed point”.

The packed decimal format provides for a varying number of digits, one per half–byte,   
followed by a half–byte denoting the sign of the number. Because of the standard byte   
addressability issues, the number of half–bytes in the representation must be an even number;   
given the one half–byte reserved for the sign, this implies an odd number of digits.

In the BCD encodings, we use one hexadecimal digit to encode each of the decimal digits.   
This leaves the six higher–valued hexadecimal digits (A, B, C, D, E, and F) with no use; in   
BCD these just do not encode any values. In Packed Decimal, each of these will encode a   
sign. Here are the most common hexadecimal digits used to represent signs.

|  |  |  |  |
| --- | --- | --- | --- |
| Binary | Hexadecimal | Sign | Comment |
| 1100 | C | + | The standard plus sign |
| 1101 | D | – | The standard minus sign |
| 1111 | F | + | A plus sign seen in converted EBCDIC |

We now move to the IBM implementation of the packed decimal format. This section breaks   
with the tone previously established in this chapter – that of discussing a format in general   
terms and only then discussing the IBM implementation. The reason for this change is   
simple; the IBM implementation of the packed decimal format is the only one used.

**The Syntax of Packed Decimal Format**

1. The length of a packed decimal number may be from 1 to 31 digits; the  
 number being stored in memory as 1 to 16 bytes.

2. The rightmost half–byte of the number contains the sign indicator. In constants  
 defined by code, this is 0xC for positive numbers and 0xD for negative.

3. The remaining number of half–bytes (always an odd number) contain the  
 hexadecimal encodings of the decimal digits in the number.

4. The rightmost byte in the memory representation of the number holds one  
 digit and the sign half–byte. All other bytes hold two digits.

5. The number zero is always represented as the two digits 0C, never 0D.

6. Any number with an even number of digits will be converted to an equivalent  
 number with a prepended “0” prior to storage as packed decimal.

7. Although the format allows for storage of numbers with decimal points, neither  
 the decimal point nor any indication of its position is stored. As an example,   
 each of 1234.5, 123.45, 12.345, and 1.2345 is stored as 12345C.

There are two common ways to generate numbers in packed decimal format, and quite a   
variety of instructions to operate on data in this format. We might discuss these in later   
chapters. For the present, we shall just show a few examples.

**1. Store the positive number 144 in packed decimal format.**

Note that the number 144 has an odd number of digits. The format just adds the half–byte   
for non–negative numbers, generating the representation **144C**. This value is often written   
as **14 4C**, with the space used to emphasize the grouping of half–bytes by twos.

**2. Store the negative number –1023 in packed decimal format.**

Note that the magnitude of the number (1023) has an even number of digits, so the format   
will prepend a “0” to produce the equivalent number 01023, which has an odd number of   
digits. The value stored is **01023D**, often written as **01 02 3D**.

**2. Store the negative number –7 in packed decimal format.**

Note that the magnitude of the number (7) has an odd number of digits, so the format   
just adds the sign half–byte to generate the representation **7D**.

**4. Store the positive number 123.456 in packed decimal format.**

Note that the decimal point is not stored. This is the same as the storage of the number  
123456 (which has a decidedly different value). This number has an even number of digits,   
so that it is converted to the equivalent value 0123456 and stored as **01 23 45 6C**.

**5. Store the positive number 1.23456 in packed decimal format.**

Note that the decimal point is not stored. This is the same as the storage of the number  
123456 (which has a decidedly different value). This number has an even number of digits,   
so that it is converted to the equivalent value 0123456 and stored as **01 23 45 6C**.

**6. Store the positive number 12345.6 in packed decimal format.**

Note that the decimal point is not stored. This is the same as the storage of the number  
123456 (which has a decidedly different value). This number has an even number of digits,   
so that it is converted to the equivalent value 0123456 and stored as **01 23 45 6C**.

**7. Store the number 0 in packed decimal form.**

Note that 0 is neither positive nor negative. IBM convention treats the zero as a positive   
number, and always stores it as **0C**.

**8. Store the number 12345678901234567890 in packed decimal form.**

Note that very large numbers are easily stored in this format. The number has 20 digits, an   
even number, so it must first be converted to the equivalent 012345678901234567890. It is   
stored as **01 23 45 67 89 01 23 45 67 89 0C**.

**Comparison: Floating–Point and Packed Decimal**

Here are a few obvious comments on the relative advantages of each format.

1. Packed decimal format can provide great precision and range, more that is required  
 for any conceivable financial transaction. It does not suffer from round–off errors.

2. The packed decimal format requires the code to track the decimal points explicitly.  
 This is easily done for addition and subtraction, but harder for other operations.  
 The floating–point format provides automatic management of the decimal point.

# Character Codes: ASCII

We now consider the methods by which computers store character data. There are three   
character codes of interest: ASCII, EBCDIC, and Unicode. The EBCDIC code is only used   
by IBM in its mainframe computer. The ASCII code is by far more popular, so we consider   
it first and then consider Unicode, which can be viewed as a generalization of ASCII.

The figure below shows the ASCII code. Only the first 128 characters (Codes 00 – 7F in   
hexadecimal) are standard. There are several interesting facts.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Last Digit \ First Digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2 | STX | DC2 | “ | 2 | B | R | b | r |
| 3 | ETX | DC3 | # | 3 | C | S | c | s |
| 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| 5 | ENQ | NAK | % | 5 | E | U | e | u |
| 6 | ACK | SYN | & | 6 | F | V | f | v |
| 7 | BEL | ETB | ` | 7 | G | W | g | w |
| 8 | BS | CAN | ( | 8 | H | X | h | x |
| 9 | HT | EM | ) | 9 | I | Y | i | y |
| A | LF | SUB | \* | : | J | Z | j | z |
| B | VT | ESC | + | ; | K | [ | k | { |
| C | FF | FS | ‘ | < | L | \ | l | | |
| D | CR | GS | - | = | M | ] | m | } |
| E | SO | RS | . | > | N | ^ | n | ~ |
| F | SI | US | / | ? | O | \_ | o | DEL |

As ASCII is designed to be an 8–bit code, several manufacturers have defined extended code   
sets, which make use of the codes 0x80 through 0xFF (128 through 255). One of the more   
popular was defined by IBM. None are standard; most books ignore them. So do we.

Let X be the ASCII code for a digit. Then X – ‘0’ = X – 30 is the value of the digit.  
For example ASCII(‘7’) = 37, with value 37 – 30 = 7.

Let X be an ASCII code. If ASCII(‘A’) ≤ X ≤ ASCII(‘Z’) then X is an upper case letter.  
 If ASCII(‘a’) ≤ X ≤ ASCII(‘z’) then X is an lower case letter.

If ASCII(‘0’) ≤ X ≤ ASCII(‘9’) then X is a decimal digit.

Let X be an upper-case letter. Then ASCII ( lower\_case(X) ) = ASCII ( X ) + 32

Let X be a lower case letter. Then ASCII ( UPPER\_CASE(X) ) = ASCII (X ) – 32.  
The expressions are ASCII ( X ) + 20 and ASCII (X ) – 20 in hexadecimal.

**NOTE:** The first 32 character codes (decimal values 0 through 31) are control  
 characters used to manage a communication process and are not printed. For   
 example the BEL (07) character would ring a bell on old teletype equipment.

Character Codes: EBCDIC

The **E**xtended **B**inary **C**oded **D**ecimal **I**nterchange **C**ode (EBCDIC) was developed by IBM   
in the early 1960’s for use on its System/360 family of computers. It evolved from an older   
character set called BCDIC, hence its name.

EBCDIC code uses eight binary bits to encode a character set; it can encode 256 characters.   
The codes are binary numeric values, traditionally represented as two hexadecimal digits.

Character codes 0x00 through 0x3F and 0xFF represent control characters.  
 0x0D is the code for a carriage return; this moves the cursor back to the left margin.  
 0x20 is used by the ED (Edit) instruction to represent a packed digit to be printed.  
 0x21 is used by the ED (Edit) instruction to force significance.  
 All digits, including leading 0’s, from this position will be printed.  
 0x25 is the code for a line feed; this moves the cursor down but not horizontally.  
 0x2F is the BELL code; it causes the terminal to emit a “beep”.

Character codes 0x40 through 0x7F represent punctuation characters.  
 0x40 is the code for a space character: “ ”.  
 0x4B is the code for a decimal point: “.”.  
 0x4E is the code for a plus sign: “+”.  
 0x50 is the code for an ampersand: “&”.  
 0x5B is the code for a dollar sign: “$”.  
 0x5C is the code for an asterisk: “\*”.  
 0x60 is the code for a minus sign: “–”.  
 0x6B is the code for a comma: “,”.  
 0x6F is the code for a question mark: “?”.  
 0x7C is the code for the commercial at sign: “@”.

Character codes 0x81 through 0xA9 represent the lower case Latin alphabet.  
 0x81 through 0x89 represent the letters “a” through “i”,  
 0x91 through 0x99 represent the letters “j” through “r”, and  
 0xA2 through 0xA9 represent the letters “s” through “z”.

Character codes 0xC1 through 0xE9 represent the upper case Latin alphabet.  
 0xC1 through 0xC9 represent the letters “A” through “I”,  
 0xD1 through 0xD9 represent the letters “J” through “R”, and  
 0xE2 through 0xE9 represent the letters “S” through “Z”.

Character codes 0xF0 through 0xF9 represent the digits “0” through “9”.

**NOTES:**

1. The control characters are mostly used for network data transmissions.  
 The ones listed above appear frequently in user code for terminal I/O.

2. There are gaps in the codes for the alphabetical characters.  
 This is due to the origins of the codes for the upper case alphabetic characters  
 in the card codes used on the IBM–029 card punch.

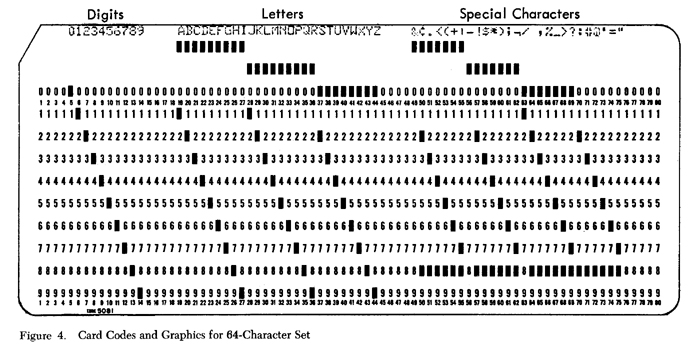
3. One standard way to convert an EBCDIC digit to its numeric value  
 is to subtract the hexadecimal number 0xF0 from the character code.

**An Abbreviated Table: The Common EBCDIC**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Code | Char. | Comment | Code | Char. | Comment | Code | Char. | Comment |
|  |  |  | 80 |  |  | C0 | } | Right brace |
|  |  |  | 81 | a |  | C1 | A |  |
|  |  |  | 82 | b |  | C2 | B |  |
|  |  |  | 83 | c |  | C3 | C |  |
|  |  |  | 84 | d |  | C4 | D |  |
|  |  |  | 85 | e |  | C5 | E |  |
|  |  |  | 86 | f |  | C6 | F |  |
|  |  |  | 87 | g |  | C7 | G |  |
| 0C | FF | Form feed | 88 | h |  | C8 | H |  |
| 0D | CR | Return | 89 | i |  | C9 | I |  |
| 16 | BS | Back space | 90 |  |  | D0 | { | Left brace |
| 25 | LF | Line Feed | 91 | j |  | D1 | J |  |
| 27 | ESC | Escape | 92 | k |  | D2 | K |  |
| 2F | BEL | Bell | 93 | l |  | D3 | L |  |
| 40 | SP | Space | 94 | m |  | D4 | M |  |
| 4B | . | Decimal | 95 | n |  | D5 | N |  |
| 4C | < |  | 96 | o |  | D6 | O |  |
| 4D | ( |  | 97 | p |  | D7 | P |  |
| 4E | + |  | 98 | q |  | D8 | Q |  |
| 4F | | | Single Bar | 99 | r |  | D9 | R |  |
| 50 | & |  | A0 |  |  | E0 | \ | Back slash |
| 5A | ! |  | A1 | ~ | Tilde | E1 |  |  |
| 5B | $ |  | A2 | s |  | E2 | S |  |
| 5C | \* |  | A3 | t |  | E3 | T |  |
| 5D | ) |  | A4 | u |  | E4 | U |  |
| 5E | ; |  | A5 | v |  | E5 | V |  |
| 5F | ­ | Not | A6 | w |  | E6 | W |  |
| 60 | – | Minus | A7 | x |  | E7 | X |  |
| 61 | / | Slash | A8 | y |  | E8 | Y |  |
| 6A | ¦ | Dbl. Bar | A9 | z |  | E9 | Z |  |
| 6B | , | Comma | B0 | ^ | Carat | F0 | 0 |  |
| 6C | % | Percent | B1 |  |  | F1 | 1 |  |
| 6D | \_ | Underscore | B2 |  |  | F2 | 2 |  |
| 6E | > |  | B3 |  |  | F3 | 3 |  |
| 6F | ? |  | B4 |  |  | F4 | 4 |  |
| 79 | ‘ | Apostrophe | B5 |  |  | F5 | 5 |  |
| 7A | : | Colon | B6 |  |  | F6 | 6 |  |
| 7B | # | Sharp | B7 |  |  | F7 | 7 |  |
| 7C | @ | At Sign | B8 |  |  | F8 | 8 |  |
| 7D | ' | Apostrophe | B9 |  |  | F9 | 9 |  |
| 7E | = | Equals | BA | [ | Left Bracket |  |  |  |
| 7F | " | Quote | BB | ] | R. Bracket |  |  |  |

It is worth noting that IBM seriously considered adoption of ASCII as its method for internal   
storage of character data for the System/360. The American Standard Code for Information   
Interchange was approved in 1963 and supported by IBM. However the ASCII code set was   
not compatible with the BCDIC used on a very large installed base of support equipment,   
such as the IBM 026. Transition to an incompatible character set would have required any   
adopter of the new IBM System/360 to also purchase or lease an entirely new set of   
peripheral equipment; this would have been a deterrence to early adoption.

The figure below shows a standard 80–column IBM punch card produced by the IBM 029   
card punch. This shows the card punch codes used to represent some EBCDIC characters.



The structure of the EBCDIC, used for internal character storage on the System/360 and later   
computers, was determined by the requirement for easy translation from punch card codes.   
The table below gives a comparison of the two coding schemes.

|  |  |  |
| --- | --- | --- |
| **Character** | **Punch Code** | **EBCDIC** |
| ‘0’ | 0 | F0 |
| ‘1’ | 1 | F1 |
| ‘9’ | 9 | F9 |
| ‘A’ | 12 – 1 | C1 |
| ‘B’ | 12 – 2 | C2 |
| ‘I’ | 12 – 9 | C9 |
| ‘J’ | 11 – 1 | D1 |
| ‘K’ | 11 – 2 | D2 |
| ‘R’ | 11 – 9 | D9 |
| ‘S’ | 0 – 2 | E2 |
| ‘T’ | 0 – 8 | E3 |
| ‘Z’ | 0 – 9 | E9 |

Remember that the punch card codes   
represent the card rows punched. Each   
digit was represented by a punch in a   
single row; the row number was   
identical to the value of the digit being   
encoded.

The EBCDIC codes are eight–bit binary   
numbers, almost always represented as   
two hexadecimal digits. Some IBM   
documentation refers to these digits as:  
 The first digit is the zone potion,  
 The second digit is the numeric.

A comparison of the older card punch codes with the EBCDIC shows that its design was   
intended to facilitate the translation. For digits, the numeric punch row became the numeric   
part of the EBCDIC representation, and the zone part was set to hexadecimal F. For the   
alphabetical characters, the second numeric row would become the numeric part and the   
first punch row would determine the zone portion of the EBCDIC.

This matching with punched card codes explains the “gaps” found in the EBCDIC set.   
Remember that these codes are given as hexadecimal numbers, so that the code immediately   
following C9 would be CA (as hexadecimal A is decimal 10). But the code for ‘J’ is not   
hexadecimal CA, but hexadecimal D1. Also, note that the EBCDIC representation for the   
letter ‘S’ is not E1 but E2. This is a direct consequence of the design of the punch cards.

Character Codes: UNICODE

The UNICODE character set is a generalization of the ASCII character set to allow for the   
fact that many languages in the world do not use the Latin alphabet. The important thing to   
note here is that UNICODE characters consume 16 bits (two bytes) while ASCII and   
EBCDIC character codes are 8 bits (one byte) long. This has some implications in   
programming with modern languages, such as Visual Basic and Visual C++, especially in   
allocation of memory space to hold strings. This seems to be less of an issue in Java.

An obvious implication of the above is that, while each of ASCII and EBCDIC use two   
hexadecimal digits to encode a character, UNICODE uses four hexadecimal digits. In part,   
UNICODE was designed as a replacement for the ad–hoc “code pages” then in use. These   
pages allowed arbitrary 256–character sets by a complete redefinition of ASCII, but were   
limited to 256 characters. Some languages, such as Chinese, require many more characters.

UNICODE is downward compatible with the ASCII code set; the characters represented by   
the UNICODE codes 0x0000 through 0x007F are exactly those codes represented by the   
standard ASCII codes 0x00 through 0x7F. In other words, to convert standard ASCII to   
correct UNICODE, just add two leading hexadecimal 0’s and make a two–byte code.

The origins of Unicode date back to 1987 when Joe Becker from [Xerox](http://en.wikipedia.org/wiki/Xerox) and Lee Collins and   
[Mark Davis](http://en.wikipedia.org/wiki/Mark_Davis_(Unicode)) from [Apple](http://en.wikipedia.org/wiki/Apple_Inc.) started investigating the practicalities of creating a universal   
character set. In August of the following year Joe Becker published a draft proposal for an   
"international/multilingual text character encoding system, tentatively called Unicode."   
In this document, entitled [Unicode 88](http://www.unicode.org/history/unicode88.pdf), he outlined a 16 bit character model:

“Unicode is intended to address the need for a workable, reliable world text   
encoding. Unicode could be roughly described as "wide-body ASCII" that has   
been stretched to 16 bits to encompass the characters of all the world's living   
languages. In a properly engineered design, 16 bits per character are more than   
sufficient for this purpose.”

In fact the 16–bit (four hexadecimal digit) code scheme has proven not to be adequate to   
encode every possible character set. The original code space (0x0000 – 0xFFFF) was   
defined as the **“Basic Multilingual Plane”**, or BMP. Supplementary planes have been   
added, so that as of September 2008 there were over 1,100,000 “code points” in UNICODE.

Here is a complete listing of the character sets and languages supported by the Basic   
Multilingual Plane. The source is http://www.ssec.wisc.edu/~tomw/java/unicode.html.

|  |  |  |
| --- | --- | --- |
| **Range** | **Decimal** | **Name** |
| 0x0000-0x007F | 0-127 | Basic Latin (what we normally use) |
| 0x0080-0x00FF | 128-255 | Latin-1 Supplement |
| 0x0100-0x017F | 256-383 | Latin Extended-A |
| 0x0180-0x024F | 384-591 | Latin Extended-B |
| 0x0250-0x02AF | 592-687 | IPA Extensions |
| 0x02B0-0x02FF | 688-767 | Spacing Modifier Letters |
| 0x0300-0x036F | 768-879 | Combining Diacritical Marks |
| 0x0370-0x03FF | 880-1023 | Greek |
| 0x0400-0x04FF | 1024-1279 | Cyrillic |
| 0x0530-0x058F | 1328-1423 | Armenian |
| 0x0590-0x05FF | 1424-1535 | Hebrew |
| 0x0600-0x06FF | 1536-1791 | Arabic |
| 0x0700-0x074F | 1792-1871 | Syriac |
| 0x0780-0x07BF | 1920-1983 | Thaana |
| 0x0900-0x097F | 2304-2431 | Devanagari |
| 0x0980-0x09FF | 2432-2559 | Bengali |
| 0x0A00-0x0A7F | 2560-2687 | Gurmukhi |
| 0x0A80-0x0AFF | 2688-2815 | Gujarati |
| 0x0B00-0x0B7F | 2816-2943 | Oriya |
| 0x0B80-0x0BFF | 2944-3071 | Tamil |
| 0x0C00-0x0C7F | 3072-3199 | Telugu |
| 0x0C80-0x0CFF | 3200-3327 | Kannada |
| 0x0D00-0x0D7F | 3328-3455 | Malayalam |
| 0x0D80-0x0DFF | 3456-3583 | Sinhala |
| 0x0E00-0x0E7F | 3584-3711 | Thai |
| 0x0E80-0x0EFF | 3712-3839 | Lao |
| 0x0F00-0x0FFF | 3840-4095 | Tibetan |
| 0x1000-0x109F | 4096-4255 | Myanmar |
| 0x10A0-0x10FF | 4256-4351 | Georgian |
| 0x1100-0x11FF | 4352-4607 | Hangul Jamo |
| 0x1200-0x137F | 4608-4991 | Ethiopic |
| 0x13A0-0x13FF | 5024-5119 | Cherokee |
| 0x1400-0x167F | 5120-5759 | Unified Canadian Aboriginal Syllabics |
| 0x1680-0x169F | 5760-5791 | Ogham |
| 0x16A0-0x16FF | 5792-5887 | Runic |
| 0x1780-0x17FF | 6016-6143 | Khmer |
| 0x1800-0x18AF | 6144-6319 | Mongolian |
| **Range** | **Decimal** | **Name** |
| 0x1E00-0x1EFF | 7680-7935 | Latin Extended Additional |
| 0x1F00-0x1FFF | 7936-8191 | Greek Extended |
| 0x2000-0x206F | 8192-8303 | General Punctuation |
| 0x2070-0x209F | 8304-8351 | Superscripts and Subscripts |
| 0x20A0-0x20CF | 8352-8399 | Currency Symbols |
| 0x20D0-0x20FF | 8400-8447 | Combining Marks for Symbols |
| 0x2100-0x214F | 8448-8527 | Letter–like Symbols |
| 0x2150-0x218F | 8528-8591 | Number Forms |
| 0x2190-0x21FF | 8592-8703 | Arrows |
| 0x2200-0x22FF | 8704-8959 | Mathematical Operators |
| 0x2300-0x23FF | 8960-9215 | Miscellaneous Technical |
| 0x2400-0x243F | 9216-9279 | Control Pictures |
| 0x2440-0x245F | 9280-9311 | Optical Character Recognition |
| 0x2460-0x24FF | 9312-9471 | Enclosed Alphanumerics |
| 0x2500-0x257F | 9472-9599 | Box Drawing |
| 0x2580-0x259F | 9600-9631 | Block Elements |
| 0x25A0-0x25FF | 9632-9727 | Geometric Shapes |
| 0x2600-0x26FF | 9728-9983 | Miscellaneous Symbols |
| 0x2700-0x27BF | 9984-10175 | Dingbats |
| 0x2800-0x28FF | 10240-10495 | Braille Patterns |
| 0x2E80-0x2EFF | 11904-12031 | CJK Radicals Supplement |
| 0x2F00-0x2FDF | 12032-12255 | Kangxi Radicals |
| 0x2FF0-0x2FFF | 12272-12287 | Ideographic Description Characters |
| 0x3000-0x303F | 12288-12351 | CJK Symbols and Punctuation |
| 0x3040-0x309F | 12352-12447 | Hiragana |
| 0x30A0-0x30FF | 12448-12543 | Katakana |
| 0x3100-0x312F | 12544-12591 | Bopomofo |
| 0x3130-0x318F | 12592-12687 | Hangul Compatibility Jamo |
| 0x3190-0x319F | 12688-12703 | Kanbun |
| 0x31A0-0x31BF | 12704-12735 | Bopomofo Extended |
| 0x3200-0x32FF | 12800-13055 | Enclosed CJK Letters and Months |
| 0x3300-0x33FF | 13056-13311 | CJK Compatibility |
| 0x3400-0x4DB5 | 13312-19893 | CJK Unified Ideographs Extension A |
| 0x4E00-0x9FFF | 19968-40959 | CJK Unified Ideographs |
| 0xA000-0xA48F | 40960-42127 | Yi Syllables |
| 0xA490-0xA4CF | 42128-42191 | Yi Radicals |
| 0xAC00-0xD7A3 | 44032-55203 | Hangul Syllables |
| 0xD800-0xDB7F | 55296-56191 | High Surrogates |
| 0xDB80-0xDBFF | 56192-56319 | High Private Use Surrogates |
| **Range** | **Decimal** | **Name** |
| 0xDC00-0xDFFF | 56320-57343 | Low Surrogates |
| 0xE000-0xF8FF | 57344-63743 | Private Use |
| 0xF900-0xFAFF | 63744-64255 | CJK Compatibility Ideographs |
| 0xFB00-0xFB4F | 64256-64335 | Alphabetic Presentation Forms |
| 0xFB50-0xFDFF | 64336-65023 | Arabic Presentation Forms-A |
| 0xFE20-0xFE2F | 65056-65071 | Combining Half Marks |
| 0xFE30-0xFE4F | 65072-65103 | CJK Compatibility Forms |
| 0xFE50-0xFE6F | 65104-65135 | Small Form Variants |
| 0xFE70-0xFEFE | 65136-65278 | Arabic Presentation Forms-B |
| 0xFEFF-0xFEFF | 65279-65279 | Specials |
| 0xFF00-0xFFEF | 65280-65519 | Halfwidth and Fullwidth Forms |
| 0xFFF0-0xFFFD | 65520-65533 | Specials |

Here is a bit of the Greek alphabet as encoded in the BMP.



For those with more esoteric tastes, here is a small sample of Cuneiform in 32–bit Unicode.



Now we see some Egyptian hieroglyphics, also with the 32–bit Unicode encoding.



We close this chapter with a small sample of the 16–bit BMP encoding for the CJK   
(Chinese, Japanese, & Korean) character set. Unlike the above two examples (Cuneiform   
and Egyptian hieroglyphics) this is a living language.



As a final note, we mention the fact that some fans of the Star Trek series have proposed that   
the alphabet for the Klingon language be included in the Unicode 32–bit encodings. So far,   
they have inserted it in the Private Use section (0xE000-0xF8FF). It is not yet recognized as   
an official part of the Unicode standard.

**Solved Problems**As is obvious from the variety of fonts, these problems have been assembled from a variety   
of sources. All of these problems have been used in a teaching context.

1. What range of integers can be stored in an 16–bit word if

a) the number is stored as an unsigned integer?

b) the number is stored in two’s–complement form?

**Answer: a) 0 through 65,535 inclusive, or 0 through 216 – 1.  
 b) –32768 through 32767 inclusive, or –(215) through (215) – 1**

2 You are given the 16–bit value, represented as four hexadecimal digits,  
 and stored in two bytes. The value is 0x812D.

a) What is the decimal value stored here, if interpreted as a packed decimal number?

b) What is the decimal value stored, if interpreted as a 16–bit two’s–complement   
 integer?

c) What is the decimal value stored here, if interpreted as a 16–bit unsigned integer?

**ANSWER:** The answers are found in the lectures for January 13 and January 20.

a) For a packed decimal number, the absolute value is 812 and the value is negative.  
 The answer is **–812**.

b) To render this as a two’s–complement integer, one first has to convert to binary.  
 Hexadecimal **812D** converts to **1000 0001 0010 1101**. This is negative.  
 Take the one’s complement to get **0111 1110 1101 0010**.  
 Add 1 to get the positive value **0111 1110 1101 0011**.  
 In hexadecimal, this is **7ED3**, which converts to 7•163 + 14•162 + 13•16 + 3,   
 or 7•4096 + 14•256 + 13•16 + 3 = 28672 + 3584 + 208 + 3 = 32,467  
 The answer is **–32,467**.

c) As an unsigned binary number the value is obtained by direct conversion from   
 the hexadecimal value. The value is 8•163 + 1•162 + 2•16 + 13,   
 or 8•4096 + 1•256 + 2•16 + 13 = 32768 + 256 + 32 + 13 = **33069**.

3. Give the 8–bit two’s complement representation of the number – 98.

**Answer: 98 = 96 + 2 = 64 + 32 + 2, so its binary representation is 0110 0010.  
 8–bit representation of + 98 0110 0010  
 One’s complement 1001 1101  
 Add 1 to get 1001 1110 9E.**

4. Give the 16–bit two’s complement representation of the number – 98.

**Answer: The 8–bit representation of – 98 is 1001 1110  
 Sign extend to 16 bits 1111 1111 1001 1110**

5 Convert the following decimal numbers to binary.

a) 37.375 b) 93.40625

ANSWER: Recall that the integer part and fractional part are converted separately.

a) 37.375  
 37 / 2 = 18 rem 1  
 18 / 2 = 9 rem 0  
 9 / 2 = 4 rem 1  
 4 / 2 = 2 rem 0  
 2 / 2 = 1 rem 0  
 1 / 2 = 0 rem 1 Answer: 100101.

0.375 • 2 = 0.75  
 0.75 • 2 = 1.50  
 0.50 • 2 = 1.00  
 0.00 • 2 = 0.00 Answer 0.011 **100101.011**

b) 93.40625  
 93 / 2 = 46 rem 1  
 46 / 2 = 23 rem 0  
 23 / 2 = 11 rem 1  
 11 / 2 = 5 rem 1  
 5 / 2 = 2 rem 1  
 2 / 2 = 1 rem 0  
 1 / 2 = 0 rem 1 Answer: 1011101

0.40625 • 2 = 0.8125  
 0.8125 • 2 = 1.6250  
 0.625 • 2 = 1.2500  
 0.25 • 2 = 0.5000  
 0.50 • 2 = 1.0000  
 0.00 • 2 = 0.0000 Answer: 0.01101 **1011101.01101**

6 Convert the following hexadecimal number to decimal numbers.  
 The numbers are unsigned. Use as many digits as necessary  
 a) 0x022 b) 0x0BAD c) 0x0EF

ANSWER: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15  
 160 = 1, 161 = 16, 162 = 256, 163 = 4096, 164 = 65536  
 a) 0x022 = 2•16 + 2 = 32 + 2 = 34 **34**  
 b) 0x0BAD = 11•162 + 10•16 + 13 = 11•256 + 10•16 + 13  
 = 2816 + 160 + 13 = 2989 **2989**  
 c) 0x0EF = 14•16 + 15 = 224 + 15 = 239 **239**

7 Show the IEEE–754 single precision representation of the following real numbers.  
 Show all eight hexadecimal digits associated with each representation.  
 a) 0.0 b) – 1.0 c) 7.625 d) – 8.75

ANSWER:  
a) 0.0 = 0x0000 0000 **0x0000 0000**

b) – 1.0 this is negative, so the sign bit is S = 1  
 1.0 = 1.0•20  
 1.M = 1.0 so M = 0000  
 P = 0 so P + 127 = 127 = 0111 11112

Concatenate S | (P + 127) | M 1 0111 1111 0000  
 Group by 4’s from the left 1011 1111 1000 0  
 Pad out the last to four bits 1011 1111 1000 0000  
 Convert to hex digits BF80  
 Pad out to eight hexadecimal digits **0xBF80 0000**

c) 7.625 this is non-negative, so the sign bit is S = 0  
 Convert 7.625 to binary.  
 7 = 4 + 2 + 1 01112  
 0.625 = 5/8 = 1/2 + 1/8 .1012  
 7.625 111.1012  
 Normalize by moving the binary point  
 two places to the left. 1.11101•22  
 Thus saying that 22 ≤ 7.625 < 23.

1.M = 1.11101 so M = 11101  
 P = 2 so P + 127 = 129 = 1000 0001

Concatenate S | (P + 127) | M 0 1000 0001 11101  
 Group by 4’s from the left 0100 0000 1111 01  
 Pad out the last to four bits 0100 0000 1111 0100  
 Covert to hex digits 40F4 **0x40F4 0000**

d) – 8.75 this is negative, so S = 1  
 8.75 = 8 + 1/2 + 1/4 1000.11  
 1.00011•23

1.M = 1.00011 so M = 00011  
 P = 3 so P + 127 = 130 1000 0010  
 Concatenate S | (P + 127) | M 1 1000 0010 00011  
 Group by 4’s from the left 1100 0001 0000 11  
 Pad out the last to four bits 1100 0001 0000 1100  
 Convert to hex digits C10C **0xC10C 0000**

8 Give the value of the real number (in standard decimal representation) represented by the   
following 32-bit words stored as an IEEE standard single precision.  
 a) 4068 0000 b) 42E8 0000 c) C2E8 0000

ANSWERS:

The first step in solving these problems is to convert the hexadecimal to binary.

a) 4068 0000 = 0100 0000 0110 1000 0000 0000 0000 0000  
Regroup to get 0 1000 0000 1101 0000 etc.

Thus s = 0 (not a negative number)  
 p + 127 = 100000002 = 12810, so p = 1  
and m = 1101, so 1.m = 1.1101 and the number is 1.1101•21 = 11.1012.

But 11.1012 = 2 + 1 + 1/2 + 1/8 = 3 + 5/8 = **3.625**.

b) 42E8 0000 = 0100 0010 1110 1000 0000 0000 0000 0000  
Regroup to get 0 1000 0101 1101 0000 etc  
Thus s = 0 (not a negative number)  
 p + 127 = 100001012 = 128 + 4 + 1 = 133, hence p = 6  
and m = 1101, so 1.m = 1.1101 and the number is 1.1101•26 = 11101002   
But 11101002 = 64 + 32 + 16 + 4 = 96 + 20 = 116 = **116.0**  
  
c) C2E80 0000 = 1100 0010 1110 1000 0000 0000 0000 0000  
Regroup to get 1 1000 0101 1101 0000 etc.  
Thus s = 1 (a negative number) and the rest is the same as b). So **– 116.0**

9 Consider the string of digits “2108”.  
 a) Show the coding of this digit string in EBCDIC.  
 b) How many bytes does this encoding take?

**ANSWER: F2 F1 F0 F8. Four bytes.**

10 Consider the positive number 2108.  
 a) Show the representation of this number in packed decimal format.  
 b) How many bytes does this representation take?

**ANSWER: To get an odd number of decimal digits, this must be represented  
 with a leading 0, as 02108. 02 10 8C. Three bytes.**

11 The following block of bytes contains EBCDIC characters.  
 Give the English sentence represented.

**E3 C8 C5 40 C5 D5 C4 4B**

**Answer:** **E3 C8 C5 40 C5 D5 C4 4B  
 T H E E N D . “THE END.”**

12 Perform the following sums assuming that each is a hexadecimal number.  
 Show the results as 16–bit (four hexadecimal digit) results.

a) 123C + 888C

b) 123C + 99D

**ANSWER: a) In hexadecimal C + C = 18 (12 + 12 = 24 = 16 + 8).  
 3 + 8 + 1 = C (Decimal 12 is 0xC)  
 2 + 8 + 0 = A (Decimal 10 is 0xA)  
 1 + 8 + 0 = 9. 9AC8**

**b) In hexadecimal C + D = 19 (12 + 13 = 25 = 16 + 9)  
 3 + 9 + 1 = D (Decimal 13 is 0xD)  
 2 + 9 + 0 = B (Decimal 11 is 0xB)  
 1 + 0 + 0 = 1 1BD9**

Each of the previous two problems uses numbers written as 123C and 888C.

The numbers in the problem on packed decimal are **not** the same as those in  
the problem on hexadecimal. They just look the same.

Interpreted as packed decimal: 123C is interpreted as the positive number 123, and   
 888C is interpreted as the positive number 888.

If we further specify that each of these is to be read as an integer value, then we have   
the two numbers +123 and +888.

Interpreted as hexadecimal, 123C is interpreted as the decimal number   
1•163 + 2•162 + 3•16 + 12 = 4096 + 512 + 48 + 12 = 4,668

Interpreted as hexadecimal, 888C is interpreted as the decimal number   
8•163 + 8•162 + 8•16 + 12 = 32768 + 2048 + 128 + 12 = 34,968

As for the number 1011, it translates to 0x3F3.

13 The two–byte entry shown below can be interpreted in a number of ways.

**VALUE DC X‘021D’**

a) What is its decimal value if it is interpreted as an unsigned binary integer?  
 b) What is its decimal value if it is interpreted as a packed decimal value?

**ANSWER:** As a binary integer, its value is 2•162 + 1•16 + 13 = 512 + 16 + 13 = 541.

As a packed decimal, this has value – 21.

14 A given computer uses byte addressing with the little-endian structure.  
 The following is a memory map, with all values expressed in hexadecimal.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Address | 104 | 105 | 106 | 107 | 108 | 109 | 10A | 10B | 10C | 10D |
| Value | C2 | 3F | 84 | 00 | 00 | 00 | 9C | C1 | C8 | C0 |

What is the value (as a decimal real number; e.g. 203.75 ) of the floating point   
 number stored at address 108? Assume IEEE–754 single precision format.

**Answer:** The first thing to notice is that the memory is byte-addressable. That means that   
each address takes holds one byte or eight bits. The IEEE format for single precision   
numbers calls for 32 bits to be stored, so the number takes four bytes of memory.

In byte addressable systems, the 32-bit entry at address 108 is stored in the four bytes  
with addresses 108, 109, 10A, and 10B. The contents of these are 00, 00, 9C, and C1.

The next thing to do is to get the four bytes of the 32-bit number in order.

This is a **little-endian** memory organization, which means that the LSB is stored at address   
108 and the MSB is stored as address 10B. In correct order, the four bytes are  
 C1 9C 00 00. In binary, this becomes  
 1100 0001 1001 1100 0000 0000 0000 0000. Breaking into fields we get  
 1**100 0001 1**001 1100 0000 0000 0000 0000, with the exponent field in bold, or

1 1000 0011 0011 1000.

Thus s = 1 a negative number

e + 127 = 1000 00112 = 128 + 2 + 1 = 131, so e = 4  
 m = 00111

So the number’s magnitude is 1.001112 • 24 = 10011.12 = 16 + 2 + 1 + 0.5 = 19.5,

and the answer is **– 19.5.**

15 Consider the 32-bit number represented by the eight hexadecimal digits  
 BEEB 0000. What is the value of the floating point number represented by this  
 bit pattern assuming that the IEEE-754 single-precision standard is used?

ANSWER: First recall the binary equivalents: B = 1011, E = 1110, and 0 = 0000.

Convert the hexadecimal string to binary

Hexadecimal: B E E B 0 0 0 0

Binary: 1011 1110 1110 1011 0000 0000 0000 0000

Regroup the binary according to the 1 | 8 | 23 split required by the format.

Binary: 1011 1110 1110 1011 0000 0000 0000 0000  
Split: 1 011 1110 1 110 1011 0000 0000 0000 0000  
Regrouped: 1 0111 1101 1101 0110 0000 0000 0000 000

The fields in the expression are now analyzed.

Sign bit: S = 1 this will become a negative number

Exponent:

The field contains 0111 1101, or 64 + 32 + 16 + 8 + 4 + 1 = 96 + 24 + 5 = 125. This number   
may be more easily derived by noting that 0111 1111 = 127 and this is 2 less.

The exponent is given by P + 127 = 125, or P = – 2. The absolute value of the number being   
represented should be in the range [0.25, 0.50), or 0.25 ≤ N < 0.50.

Mantissa:

The mantissa field is 1101 0110, so 1.M = 1.1101 0110.

In decimal, this equals 1 + 1/2 + 1/4 + 1/16 + 1/64 + 1/128, also written as  
 (128 + 64 + 32 + 8 + 2 + 1) / 128 = (192 + 40 + 3) / 128 = 235 / 128.

The magnitude of the number equals 235 / 128 • 1/4 = 235 / 512 = 0.458984375.

16 The following are two examples of the hexadecimal representation of   
 floating–point numbers stored in the IBM single–precision format.  
 Give the decimal representation of each. Fractions (e.g., 1/8) are acceptable.

a) C1 64 00 00

b) 3F 50 00 00

**ANSWER: a) First look at the sign and exponent byte. This is 0xC1, or 1100 0001.**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bit** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **Value** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **1** |

**The sign bit is 1, so this is a negative number.  
Stripping the sign bit, the exponent field is 0100 0001 or 0x41 = decimal 65.  
We have (Exponent + 64) = 65, so the exponent is 1.**

**The value is 161•F, where F is 0x64, or 6/16 + 4/256.  
The magnitude of the number is 161•(6/16 + 4/256) = 6 + 4/16 = 6.25. The value is –6.25.**

**b) First look at the sign and exponent byte. This 0x3F, or 0011 1111**

**The sign bit is 0, so this is a non–negative number.  
The exponent field is 0x3F = 3•16 + 15 = decimal 63 = 64 – 1.**

**The value is 16–1•F, where F is 0x50, or 5/16.  
The magnitude of the number is 16–1•(5/16) = 5/256 = 0.01953125.**

17 These questions refer to the IBM Packed Decimal Format.

a) How many bytes are required to represent a 3–digit integer?

b) Give the Packed Decimal representation of the positive integer 123.

c) Give the Packed Decimal representation of the negative integer –107.

**ANSWER:** Recall that each decimal digit is stored as a hexadecimal digit, and   
 that the form calls for one hexadecimal digit to represent the sign.

a) One needs four hexadecimal digits, or two bytes, to represent three decimal digits.

b) **12 3C** c) **10 7D**

18 These questions also refer to the IBM Packed Decimal Format.

a) How many decimal digits can be represented in Packed Decimal form   
 if three bytes (8 bits each) are allocated to store the number?

b) What is the Packed Decimal representation of the largest integer stored in 3 bytes?

**ANSWER:** Recall that N bytes will store 2•N hexadecimal digits. With one of these   
 reserved for the sign, this is (2•N – 1) decimal digits.

a) 3 bytes can store **five decimal digits**.

b) The largest integer is 99,999. It is represented as **99 99 9C**.

19 Convert the following numbers to their representation IBM Single Precision  
 floating point and give the answers as hexadecimal digits.

a) 123.75

b) –123.75

**ANSWER:  
 a)** First convert the number to hexadecimal.

The whole number conversion: 123 / 16 = 7 with remainder = 11 (B)  
 7 / 16 = 0 with remainder 7. 123 = 7B.

The fractional part conversion: .75•16 = 12 (C). The number is 7B.C

The number can be represented as 162 • 0.7BC; the exponent is 2.

The exponent stored with excess 64, thus it is 66 or X‘42’.

Appending the fractional part, we get X‘427BC’.

Add three hexadecimal zeroes to pad out the answer to X‘**427B C000**’

b) The only change here is to add the sign bit as the leftmost bit.

In the positive number, the leftmost byte was X‘**42**’, which in binary would be

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Value | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

Just flip the bit in position 0 to get the answer for the leftmost byte.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Value | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

This is X‘**C2**’. The answer to this part is X‘**C27B C000**’

Convert the following numbers to their representation in packed decimal.  
 Give the hexadecimal representation with the proper number of hexadecimal digits.

a) 123.75

b) –123.7

**ANSWER: a)** 12375 has five digits. It is represented as **12 37 5C**.

b) 1237 has four digits. Expand to 01237 and represent as **01 23 7D**.

20 Give the correct Packed Decimal representation of the following numbers.

a) 31.41 b) –102.345 c) 1.02345

**ANSWER:** Recall that the decimal is not stored, and that we need to have an odd   
 count of decimal digits.

a) This becomes 3141, or 03141. 03141C

b) This becomes 102345, or 0102345 0102345D

c) This also becomes 102345, or 0102345 0102345C.

21 Perform the following sums of numbers in Packed Decimal format. Convert to   
 standard integer and show your math. Use Packed Decimal for the answers.

a) **025C** + **085C** d) **666D** + **444D**

b) **032C** + **027D** e) **091D** + **0C**

c) **10003C** + **09999D**

**ANSWER:** Just do the math required and convert back to standard   
 Packed Decimal format.

a) **025C** + **085C** represents 25 +85 = 110. This is represented as **110C**.

b) **032C** + **027D** represents 32 –27 = 5. This is represented as **5C**.

c) **10003C** + **09999D** represents 10003 –9999 = 4. This is represented as **4C**.

d) **666D** + **444D** represents –666 –444 = –1110. This is represented as **01 11 0D**.

e) **091D** + **0C** represents –91 +0 = –91 This is represented as **091D**.

22 These questions concern 10–bit integers, which are not common.

a) What is the range of integers storable in 10–bit unsigned binary form?

b) What is the range of integers storable in 10–bit two’s–complement form?

c) Represent the positive number 366 in 10–bit two’s–complement binary form.

d) Represent the negative number –172 in 10–bit two’s–complement binary form.

e) Represent the number 0 in 10–bit two’s–complement binary form.

**ANSWER:** Recall that an N–bit scheme can store 2N distinct representations.  
 For unsigned integers, this is the set of integers from 0 through 2N – 1.  
 For 2’s–complement, this is the set from – (2N–1) through 2N–1 – 1.

a) For 10–bit unsigned the range is 0 though 210 – 1, or 0 through 1023.

b) For 10–bit 2’s–complement, this is – (29) through 29 – 1, or – 512 through 511.

c) 366 / 2 = 183 remainder = 0  
 183 / 2 = 91 remainder = 1  
 91 / 2 = 45 remainder = 1  
 45 / 2 = 22 remainder = 1  
 22 / 2 = 11 remainder = 0  
 11 / 2 = 5 remainder = 1  
 5 / 2 = 2 remainder = 1  
 2 / 2 = 1 remainder = 0  
 1 / 2 = 0 remainder = 1. **READ BOTTOM TO TOP!**

The answer is 1 0110 1110, or **01 0110 1110**, which equals **0x16E**.  
**0x16E** = 1•256 + 6•16 + 14 = 256 + 96 + 14 = 256 + 110 = 366.

The number is not negative, so we stop here. Do not take the two’s complement unless the  
number is negative.

d) 172 / 2 = 86 remainder = 0  
 86 / 2 = 43 remainder = 0  
 43 / 2 = 21 remainder = 1  
 21 / 2 = 10 remainder = 1  
 10 / 2 = 5 remainder = 0  
 5 / 2 = 2 remainder = 1  
 2 / 2 = 1 remainder = 0  
 1 / 2 = 0 remainder = 1. **READ BOTTOM TO TOP!**

This number is 1010 1100, or **00 1010 1100**, which equals **0x0AC**.  
**0xAC** = 10•16 + 12 = 160 + 12 = 172.

The absolute value: **00 1010 1100**  
Take the one’s complement: **11 0101 0011**  
Add one:  **1**  
The answer is: **11 0101 0100** or **0x354.**

e) The answer is **00 0000 0000**.  
 You should just know this one.

23 These questions IBM Packed Decimal Form.

a) Represent the positive number 366 as a packed decimal with fewest digits.

d) Represent the negative number –172 as a packed decimal with fewest digits.

e) Represent the number 0 as a packed decimal with fewest digits.

**ANSWER:** a) **366C**  
 b) **172D**  
 c) **0C** (not **0D**, which is incorrect)