Chapter 4 – 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 hexadecimal to decimal, are often best done via binary.

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, 32, and 64 bits.

The next topic for this chapter is the storage of real (floating point) numbers. This discussion will mention the standard put forward by the Institute of Electrical and Electronic Engineers, the IEEE Standard 754 for floating point numbers, but will focus on the base–16 format used by IBM Mainframes. The chapter closes with a discussion of codes for storing characters, focusing on the EBCDIC system used on IBM mainframes.

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. Note that the hexadecimal numbering system calls for more than ten digits, so we use the first six letters of the alphabet.

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

The fact that the bases for octal and hexadecimal are powers of the basis for binary facilitates the conversion between these bases. 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 octal and hexadecimal is best done by first converting to binary.

Except for an occasional reference, we shall not use the octal system much, but focus on the decimal, binary, and hexadecimal numbering systems.

The figure below shows the numeric equivalents in binary and decimal of the first 16 hexadecimal numbers. The following is taken from that figure.

Binary (base 2)	Decimal (base 10)	Hexadecimal (base 16)
0000	00	0
0001	01	1
0010	02	2
0011	03	3
0100	04	4
0101	05	5
0110	06	6
0111	07	7
1000	08	8
1001	09	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Note that conversions from hexadecimal to binary can be done one digit at a time, thus DE = 11011110, as D = 1101 and E = 1110. We shall normally denote this as DE = 1101 1110 with a space to facilitate reading the binary.

Conversion from binary to hexadecimal is also quite easy. Group the bits four at a time and convert each set of four. Thus 10111101, written 1011 1101 for clarity is BD because 1011 = B and 1101 = D.

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, thus obtaining 0011 1010, which is converted to 3A as 0011 = 3 and 1010 = A.

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 an arbitrary member of that infinite set. Computer storage formats allow only for the representation of a large, but finite, subset of the integers.

It is easy to show that an N–bit binary integer can represent one of 2^N 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 $B_N B_{N-1} \ldots B_3 B_2 B_1$. By the inductive hypothesis, this can represent one of 2^N possible values.
3. We now consider an (N+1)–bit integer, written as $B_{N+1} B_N B_{N-1} \ldots B_3 B_2 B_1$. By the inductive hypothesis, there are 2^N values of the form $0 B_N B_{N-1} \ldots B_3 B_2 B_1$, and 2^N values of the form $1 B_N B_{N-1} \ldots B_3 B_2 B_1$.
4. The total number of (N+1)–bit values is $2^N + 2^N = 2^{N+1}$. 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 2^N integer values in the range 0 through $2^N - 1$, inclusive. Sample ranges include:

N =	4	0 through	$2^4 - 1$	0 through	15
N =	8	0 through	$2^8 - 1$	0 through	255
N =	12	0 through	$2^{12} - 1$	0 through	4095
N =	16	0 through	$2^{16} - 1$	0 through	65535
N =	20	0 through	$2^{20} - 1$	0 through	1,048,575
N =	32	0 through	$2^{32} - 1$	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.

We begin this discussion by describing how to convert unsigned decimal integers to unsigned binary numbers. Remember the requirement that the decimal number fit within the range supported by the representation to be used; the number 967 cannot be converted to 8–bit binary as it lies outside the range 0 – 255. As $2^{10} = 1024$, this number requires 10 bits.

In this discussion, we shall use as many bits as are needed to represent the number. After we do this, we shall then consider signed representations that allow the use of negative integers.

Both whole numbers (integers) and fractional parts of decimal numbers can be converted to binary and represented in that notation. Obviously, the integer formats discussed above do not support fractional parts; they are needed to support the floating point representations.

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 in binary. 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 $2^4 < 23 \leq 2^5$. 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 top to bottom as least significant bit to most significant bit. 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	This last step must be done to get the first 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. 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

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.

We now develop a “power of 2” notation that will be required when we study the IEEE floating point standard. We have just shown that decimal 23.375 = 10111.011 binary. Recall that in the scientific “power of 10” notation, when we move the decimal to the left one place we have to multiply by 10. Thus, $1234 = 123.4 \bullet 10^1 = 12.34 \bullet 10^2 = 1.234 \bullet 10^3$.

We apply the same logic to the binary number. In the IEEE standard we need to form the number as a **normalized number**, which is of the form $1.xxx \bullet 2^p$. In changing 10111 to 1.0111 we have moved the decimal point (O.K. – it should be called binary point) 4 places to the left, so $10111.011 = 1.0111011 \bullet 2^4$. Recalling that $2^4 = 16$ and $2^5 = 32$, and noting that $16.0 < 23.375 \le 32.0$ we see that the result is as expected.

Conversion from binary to decimal is quite easy. One just remembers the decimal representations of the powers of 2. We convert 10111.011 binary to decimal. Recalling the positional notation used in all number systems:

$$\begin{aligned} 10111.011 &= 1\bullet2^4 + 0\bullet2^3 + 1\bullet2^2 + 1\bullet2^1 + 1\bullet2^0 + 0\bullet2^{-1} + 1\bullet2^{-2} + 1\bullet2^{-3} \\ &= 1\bullet16 + 0\bullet8 + 1\bullet4 + 1\bullet2 + 1\bullet1 + 0\bullet0.5 + 1\bullet0.25 + 1\bullet0.125 \\ &= 23.375 \end{aligned}$$

Conversions between Decimal and Hexadecimal

The conversion is best done 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 (More Confusing 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.

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
```

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 = 0.33333333333333333333333333333333…, 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.

Consider the following sequence of multiplications

1/4 • 10 = 2½

½ • 10 = 5. Thus 1/4 = 25/100 = 0.25.

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.

In decimal numbering, a fraction is terminating if and only if it can be represented in the form $J / 10^K$ for some integers J and K. We have seen that $1/4 = 25/100 = 25/10^2$, thus the fraction 1/4 is a terminating fraction because we have shown the integers J = 25 and K = 2. Note that this representation is never unique if it exists; 1/4 = 250/1000 (J = 250 and K = 3), and 1/4 = 2500/10000 (J = 2500 and K = 4). By convention, we use the smallest values.

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 $10^K = 2^K•5^K$. To motivate the first proof, note that $10^0 = 1 = 9•0 + 1$, $10 = 9•1 + 1$, $100 = 9•11 + 1$, $1000 = 9•111 + 1$, etc. If 1/3 were a terminating decimal, we could solve the following equations for integers J and M.

$\frac{1}{3} = \frac{J}{10^K} = \frac{J}{9 \bullet M + 1}$, 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.

$\frac{1}{3} = \frac{J}{10^K} = \frac{J}{2^K \bullet 5^K}$, which becomes $3•J = 2^K•5^K$. This has an integer solution J only if the right hand side of the equation can be factored by 3. But neither 2^K nor 5^K can be factored by 3, so the right hand side cannot be factored by 3 and hence the equation is not solvable.

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.

The proof that no terminating representation exists depends on the fact that any terminating fraction in binary can be represented in the form $\frac{J}{2^K}$ for some integers J and K. Thus we solve $\frac{1}{5} = \frac{J}{2^K}$ or $5 \bullet J = 2^K$. 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 representation of an integer requires 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.

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

Immediately, we have the Boolean implementation of a Full Adder; how to make such a circuit using only AND, OR, and NOT gates.

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
0001      0001      0101
0111      1100      0010
```

In the first sum, we add 1 to an even number. This is quite easy to do. Just change the last 0 to a 1. 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 give 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 numbers.

Signed Integers

Fixed point numbers include real numbers with a fixed number of decimals, such as those commonly used to denote money amounts in the United States. We shall focus only on integers and relegate the study of real numbers to the floating point discussion.

Integers are stored in a number of formats. The most common formats today include 16 and 32 bits. The new edition of Visual Basic will include a 64-bit standard integer format.

Bits in the storage of an integer are usually 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. As we shall see, the IBM Mainframe documentation numbers the bits left to right, with bit 0 being the leftmost.

Although 32-bit integers are probably the most common, we shall focus on eight-bit integers because they are easy to illustrate. In these discussions, the student should recall the powers of 2: $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, $2^5 = 32$, $2^6 = 64$, $2^7 = 128$, and $2^8 = 256$.

The simplest topic to cover is the storage of **unsigned integers**. As there are 2^N possible combinations of N binary bits, there are 2^N unsigned integers ranging from 0 to $2^N - 1$. For eight-bit unsigned integers, this range is 0 though 255, as $2^8 = 256$. 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.

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. Note the leading 0; we must have an 8-bit representation. As another example, consider the decimal number 12. It can be represented in binary as 1100, but is 0000 1100 as an 8–bit binary number. Recall that the space in the binary is for readability only.

Decimal 100 = 0110 0100
One's complement 1001 1011; in one's complement, decimal –100 = 1001 1011 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 $-0 \neq 0$, a violation of some of the basic principles of mathematics.

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.

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, and we avoid the messy problem of having $-0 \neq 0$.

In N- bit two's complement arithmetic, the range of integers that can be represented is -2^{N-1} through $2^{N-1} - 1$ inclusive, thus the range for eight-bit two's complement integers is –128 through 127 inclusive, as $2^7 = 128$. The table at the right shows a number of binary representations for this example.

+ 127	0111 1111	
+ 10	0000 1010	
+1	0000 0001	
0	0000 0000	
– 1	1111 1111	The number is
– 10	1111 0110	negative if and
– 127	1000 0001	only if the high
– 128	1000 0000	order 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 -2^{63} to $2^{63} - 1$. As an exercise in math, I propose to do a rough calculation of 2^{63}. 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 $2^{63} = (10^{0.30103})^{63} = 10^{18.9649} = 10^{0.9649} \bullet 10^{18} = 9.224 \bullet 10^{18}$, so a 64-bit integer allows the representation of 18 digit numbers and most 19 digit numbers.

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 $A_{15}A_{14}A_{13}A_{12}A_{11}A_{10}A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0$. Consider placing this number into a 32-bit register with bits numbered R_{31} through R_0, with R_{31} being the sign bit. Part of the solution is obvious: make $R_k = A_k$ for $0 \le k \le 15$. 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, which evaluates to 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, evaluating as 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 1000 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 $0 \ldots 2^N - 1$, inclusive. For N-bit two's-complement integers the range of non-negative integers representable is $0 \ldots 2^{N-1} - 1$, 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 $0 \le 123 \le 2^7 - 1$, 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.

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	-2^{15} to $2^{15}-1$	or – 32768	to 32767
32-bit	-2^{31} to $2^{31}-1$	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 $24576 = 16384 + 8192 = 2^{14} + 2^{13}$.

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 $0 \le M \le 32767$ and $-32768 \le N \le 0$. If $|M| \ge |N|$, we have $0 \le (M + N) \le 32767$ and the sum is valid. Otherwise, we have $-32768 \le (M + N) \le 0$, 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–64 and Excess–127

We now cover **excess–M** representation for both M = 64 and M = 127. This method is of limited utility in storing integers, but is used to store the exponent in several common floating point representations, including the IEEE floating point standard (with M = 127) and the IBM Mainframe format (M = 64). 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 N–bit unsigned binary number.

The observant reader will note an obvious relationship between the values of the integers N and M. Recall that an N–bit number can represent 2^N values, as unsigned integers this represents the range 0 through $2^N - 1$. One would expect that $M \approx 2^N / 2 = 2^{N-1}$. The following table gives some values for typical floating–point standards.

Format	N	2^N	M
IEEE Single Precision	8	256	127
IEEE Double Precision	11	2048	1023
IBM Single Precision	7	128	64
IBM Double Precision	7	128	64

As an example consider 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 $0 \le (N + 127) \le 255$, or $-127 \le N \le 128$.

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 and IBM single precision standards for storing floating point numbers in a computer.

Normalized Numbers

The last topic to be discussed prior to defining the standards 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 defined in terms of a positive base B, with $B \geq 2$ being required. Such a normalized number is one with a representation of the form $X \bullet B^P$, where we require that $1.0 \leq X < B$. The two most common representations use B = 2 and B = 16. Thus we have the following requirements for the representation.

B = 2 $N = X \bullet 2^P$, with $1.0 \leq X < 2.0$

B = 16 $N = X \bullet 16^P$, with $1/16 = 0.625 \leq X < 1.0$

Before considering how to store floating–point numbers is the standard format, we ask a very simple question, which holds for all representations.

"What common number cannot be represented as a normalized number?"

The answer is **zero**. Consider the binary floating point formats. There is no power of 2 such that $0.0 = X \bullet 2^P$, where $1.0 \leq X < 2.0$. 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 $= 1.0 \bullet 2^0$, thus X = 1.0 and P = 0.
1.5 $= 1.5 \bullet 2^0$, thus X = 1.5 and P = 0.
2.0 $= 1.0 \bullet 2^1$, thus X = 1.0 and P = 1
0.25 $= 1.0 \bullet 2^{-2}$, thus X = 1.0 and P = -2
7.0 $= 1.75 \bullet 2^2$, thus X = 1.75 and P = 2
0.75 $= 1.5 \bullet 2^{-1}$, thus X = 1.5 and P = -1.

Normalizing a Number (Style of the IEEE Format)

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: $9.375 \bullet 10^{-2} = 0.09375$. 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 $1.1 \bullet 2^{-4}$ in the normalized notation.
0.75	• 2 = 1.5	1	
0.5	• 2 = 1.0	1	

Please note that these representations take the form $X \bullet 2^P$, 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 = $2^6 \bullet (1 + ¼)$.

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 $1.01000000011 \bullet 2^6$, as expected. For convenience, we write this as $1.0100\ 0000\ 0110 \bullet 2^6$.

Normalizing a Number (Style of IBM Mainframe)
Consider again 80.09375 = 1010000.00011 binary, rewritten as 0101 0000.0001 1000, or 0x50.18. In hexadecimal, this should be seen as $0.5018 \bullet 16^2$, where the number 5018 is really a hexadecimal representation of the decimal number $(5/16 + 0/16^2 + 1/16^3 + 8/16^4)$ that might better be written as 0x0.5018.

Infinity and NAN (Not–A–Number)
Some formats for storing floating–point numbers, such as the IEEE 754 standard, allow for representing items that are not really numbers. Among these values are the pseudo–numbers called Infinity and NAN. We explain the two with a thought experiment.

Consider the quotient 1/0. The equation 1 / 0 = X is equivalent to solving for a number X such that $0 \bullet X = 1$. There is no such number. Loosely speaking, we say $1 / 0 = \infty$.

More precisely, consider the value of (1.0 / *w*) for *w* > 0. As the real number *w* becomes very small, the value (1.0 / *w*) grows very large. As the value *w* approaches 0, the value (1.0 / *w*) grows without bound. In terms of calculus, we say that $\lim_{w \to 0} \left(\frac{1}{w} \right) = \infty$. In the precise reasoning associated with calculus, we never consider the pseudo–number 1/0, just the limit.

Now consider the quotient 0/0. Again we are asking for the number X such that $0 \bullet X = 0$. The difference here is that this equation is true for every number X. In terms of the standards, 0 / 0 is called “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, or finding the angle whose trigonometric sine has value 2.

There is no evidence that the IBM Mainframe assembler supports either Infinity or NaN.

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 $0 \le N \le 127$, we have $0 \le (E + 64) \le 127$, or $-64 \le E \le 63$.

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 16^{63}. Note that $16^{63} = (2^4)^{63} = 2^{252}$, and $16^{-64} = (2^4)^{-64} = 2^{-256} = 1.0 / (2^{256})$ and recall that $\log_{10}(2) = 0.30103$. Using this, we compute the maximum number storable at about $(10^{0.30103})^{252} = 10^{75.86} \approx 9 \bullet 10^{75}$. We may approximate the smallest positive number at $1.0 / (36 \bullet 10^{75})$ or about $3.0 \bullet 10^{-77}$. In summary, the following real numbers can be represented in this standard: $X = 0.0$ and $3.0 \bullet 10^{-77} < X < 9 \bullet 10^{75}$.

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 2^F.

We can summarize the precision for each format as follows.

Single precision	F = 24	1 part in 2^{24}.
Double precision	F = 56	1 part in 2^{56}.
Extended precision	F = 120	1 part in 2^{120}.

The first power of 2 is easily computed; we use logarithms to approximate the others.

2^{24}		= 16,777,216
2^{56}	$\approx (10^{0.30103})^{56} = 10^{16.85}$	$\approx 9 \bullet 10^{16}$.
2^{120}	$\approx (10^{0.30103})^{120} = 10^{36.12}$	$\approx 1.2 \bullet 10^{36}$.

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 10^7, but not as good as 1 in 10^8; 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	$3.0 \bullet 10^{-77} < X < 9 \bullet 10^{75}$	7 digits
Double Precision	$3.0 \bullet 10^{-77} < X < 9 \bullet 10^{75}$	16 digits
Extended Precision	$3.0 \bullet 10^{-77} < X < 9 \bullet 10^{75}$	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 $X \bullet 16^E$, 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 $X \bullet 16^E$, with $0.625 \le X < 1.0$. As $128 \le X < 256$, the number is converted to the form $X \bullet 16^2$.
Note that $128 = (1/2) \bullet 16^2 = (8/16) \bullet 16^2$, and $0.5 = (1/512) \bullet 16^2 = (8/4096) \bullet 16^2$.
Hence, the value is $128.50 = (8/16 + 0/256 + 8/4096) \bullet 16^2$; it is $16^2 \bullet 0x0.808$.

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 $16^2 \bullet 0x0.808$.

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 $16^0 \bullet 0.375 = 16^0 \bullet (6/16)$. This is in the required format $X \bullet 16^E$, with $0.625 \le X < 1.0$.

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 $16^2 \bullet 0x0.7B33333333\ldots..$

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 $X \bullet 16^1$. The fraction field is `100 000`, so the value is $16^1 \bullet (1/16) = 1.0$.

Example 6: Another in "Reverse"
We are given the single precision representation of the number. It is `BEC8 0000`.
What is the value of the number stored? We begin by examination of the first two hex digits.

Field	Sign	Characteristic						
Value	1	0	1	1	1	1	1	0
Hex value	B				E			

The characteristic has value 0x3E or decimal 3•16 + 14 = 62. The exponent has value 62 – 64 = –2. The number is then 16^{-2}•0x0.C8 = 16^{-2}•(12/16 + 8/256), which can be converted to decimal in any number of ways. I prefer the following conversion.
16^{-2}•(12/16 + 8/256) = 16^{-2}•(3/4 + 1/32) = 16^{-2}•(24/32 + 1/32) = 16^{-2}•(25/32)
= 25 / (32•256) = 25 / 8192 ≈ 3.0517578•10^{-3}.
The answer is approximately the negative number **–3.0517578•10^{-3}**.

Why Excess–64 Notation for the Exponent?
We have introduced two methods to be used for storing signed integers: two's–complement notation and excess–64 notation. One might well ask why two's-complement notation is not used to store the exponent in the characteristic field.

The answer for integer notation is simple. Consider some of examples.
`128.50` is represented as `4280 8000`. Viewed as a 32–bit integer, this is positive.
`-128.50` is represented as `C280 8000`. Viewed as a 32–bit integer, this is negative.
`1.00` is represented as `4110 0000`. Viewed as a 32–bit integer, this is positive.
`-3.05•10^-3` is represented as `BEC8 0000`. Viewed as a 32–bit integer, this is negative

It turns out that the excess–64 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. For example, the above examples would yield the following order.

`4280 8000` for `128.50`. This is the largest.
`4110 0000` for `1.00`.
`BEC8 0000` for `-3.05•10^-3`.
`C280 8000` for `-128.50`. This is the most smallest (most negative).

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 shall 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

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 EBCDIC.

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

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.

EBCDIC

The EBCDIC (Extended Binary Coded Decimal Interchange Code) was developed in 1963 and 1964 by IBM for use on its System/360 line of computers. It was created as an extension of the BCD (Binary Coded Decimal) encoding that existed at the time.

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			

Some Final Comments

We end this chapter with a few comments on the value of numbers and their representation. This set of comments addresses an issue that is a problem for only a few students.

While these comments apply to all numeric forms, they are restricted to a discussion of positive integer forms. This restriction is only for the purpose of simplicity in the discussion; having grasped these arguments, the reader may extend them to other numeric formats.

Consider the number 6143. This is obviously a positive integer. What is to be said about it?

The first thing to say is that it is not possible to write an integer. The set of integers can be viewed as an abstract idea; what we write is best called the **"print representation"** of a member of the set of integers. Thus, when we write "6143", this set of symbols represents, and is interpreted as, denoting the abstract concept associated with the integer.

In considering assembler language, it is important to note that integers are what they are. We may speak of a number of representations of an integer; but the representations do not define that integer, they only show how the integer might be represented. As an example, let us consider the integer denoted by the print representation 6143.

If this integer is to be printed by an IBM assembler language program, it must be explicitly be converted to its EBCDIC representation: F6 F1 F4 F3. This sequence of four bytes will be printed on the output device as "6143".

One may ask if this number is a binary, decimal, or hexadecimal number. This question is not valid; the integer is what it is. One may ask about its representation in different bases.

As a binary number, decimal 6143 is represented as 1 0111 1111 1111.

As a hexadecimal number, decimal 6143 is represented as 0x17FF.

Put another way, we can ask the invalid question "Is 0x17FF a binary number?". The answer is simple: the number is what it is and is not to be confused with the mode of representation. This number can be represented in many other forms, as a binary number it is 1 0111 1111 1111 or 0001 0111 1111 1111, depending on one's preferences.

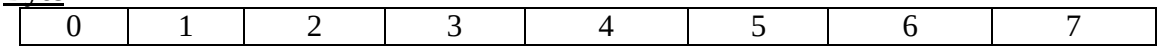
The "bottom line" is quite simple. No integer is to be viewed as essentially binary, octal, decimal, or hexadecimal; integers are what they are and nothing more. Rather we should ask about various representations of the number.

A Note on Bit Numbering.

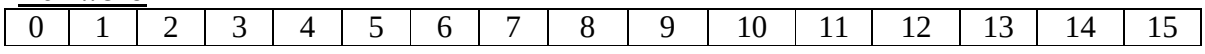
Any N–bit number has bit numbers ranging from 0 through (N – 1). The common notation assigns bit 0 as the rightmost bit and bit (N – 1) as the leftmost bit, usually the sign bit.

In IBM terminology, the leftmost bit is bit zero, so we have the following.

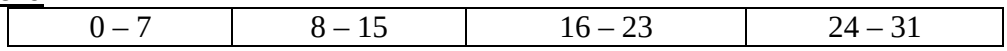
Byte



Halfword



Fullword



Some Solved Problems

Here is a collection of solved problems, taken from the answer keys to a few homework assignments and several quizzes. These have been only minimally edited from the original.

In order to save time, this section retains the font styles used in the original answer keys. The questions are in black font, the answers are in blue font, and comments are in red font.

1. Write the EBCDIC representation of the 9 character sequence "CPSC 3121".

Answer: Look up the EBCDIC codes. The relevant codes are:

'C'	C3	'P'	D7	'S'	E2	Blank	40
'3'	F3	'1'	F1	'2'	F2		

The codes: C3 D7 E2 C3 40 F3 F1 F2 F1, or C3D7 E2C3 40F3 F1F2 F1

2. 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 2^N distinct representations.
For unsigned integers, this is the set of integers from 0 through $2^N - 1$.
For 2's–complement, this is the set from $-(2^{N-1})$ through $2^{N-1} - 1$.

a) For 10–bit unsigned the range is 0 though $2^{10} - 1$, or 0 through 1023.

b) For 10–bit 2's–complement, this is $-(2^9)$ through $2^9 - 1$, 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.

Comment: This question always fools some students, who feel compelled to take the two's–complement and thus produce the representation of –366.

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.`**

Comment: The conversion to binary just gives the value **`1010 1100`**.
As this is to be a 10–bit number, it must be expanded to 10 bits
before the two's–complement is taken. Convert **`00 1010 1100`**.

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

3. This question concerns the range of integers that can be represented in different formats. The answers can be either decimal or powers of 2 (-2^{15}).

   a) What range can be represented by 12–bit unsigned integers?

   b) What range can be represented by 12–bit two's–complement signed integers?

   c) What range can be represented by 16–bit two's–complement signed integers?

Answer: In general, the ranges for N–bit integer representations are
0 through 2^N-1 for N–bit unsigned integers, and
$-(2^{N-1})$ through $(2^{N-1}) - 1$ for N–bit two's–complement integers.

a) For 12–bit unsigned integers, the range is 0 through $2^{12} - 1$ or 0 through 4,096.

b) For 12–bit two's–complement, the range is $-(2^{11})$ through $(2^{11}) - 1$,
or –2,048 through 2,047.

c) For 16–bit two's–complement, the range is $-(2^{15})$ through $(2^{15}) - 1$,
or –32,768 through 32,767.

4. a) How many hexadecimal digits are required to represent an 8–bit integer?

   b) How many hexadecimal digits are required to represent a 32–bit integer?

ANSWER: Recall that each hexadecimal digit represents four bits.

a) It takes **two hexadecimal digits** to represent an 8–bit byte, or 8–bit integer.

b) It takes **eight hexadecimal digits** to represent a 32–bit integer.

5. The System/360 uses 32–bit two's–complement form for binary numbers.
   Show the following answers as 16–bit binary numbers or four–digit hexadecimal.

   a) Show the binary and hexadecimal representation of the positive integer +349.

   b) Show the binary and hexadecimal representation for the negative integer –589.

Answer: a) Find 349 by repeated division.

349 / 2	= 174	remainder = 1	
174 / 2	= 87	remainder = 0	
87 / 2	= 43	remainder = 1	1 0101 1101, or 0001 0101 1101
43 / 2	= 21	remainder = 1	0000 0001 0101 1101
21 / 2	= 10	remainder = 1	
10 / 2	= 5	remainder = 0	0x01 5D
5 / 2	= 2	remainder = 1	
2 / 2	= 1	remainder = 0	1•256 + 5•16 + 13 =
1 / 2	= 0	remainder = 1	256 + 80 + 13 = 349

Answer: b) Find 589 by repeated division.

589 / 2	= 294	remainder = 1	
294 / 2	= 147	remainder = 0	
147 / 2	= 73	remainder = 1	10 0100 1101 or 0010 0100 1101
73 / 2	= 36	remainder = 1	0000 0010 0100 1101
36 / 2	= 18	remainder = 0	
18 / 2	= 9	remainder = 0	0x02 4D
9 / 2	= 4	remainder = 1	
4 / 2	= 2	remainder = 0	2•256 + 4•16 + 13 =
2 / 2	= 1	remainder = 0	
1 / 2	= 0	remainder = 1	512 + 64 + 13 = 589

Take the two's complement of 0000 0010 0100 1101

The positive binary number	**0000 0010 0100 1101**
Its one's complement	**1111 1101 1011 0010**
Add one to get	**1111 1101 1011 0011 or 0xFD B3**

Comment: **The correct answer to the first problem is shown above.**
I accepted 0001 0101 1101 or 0x15D, which is a 12–bit number.

The 12–bit answer to part b) is just wrong. Deduct 4 points.

Also: **A correct answer without showing the method is only partly correct.**
I deducted 4 points for failure to show the method.

Also: **At this point, this is not Packed Decimal Notation.**
The leading 1111 is 0xF, the hexadecimal digit.

6. Use the System/360 standard for Packed Decimal representation.
   a) Represent the positive integer +19372 in the packed decimal format.
   b) Represent the negative integer –8191 in the packed decimal format.
   c) Represent the positive number 4,321.54 in the packed decimal format.
   d) Represent the negative number –3.14159 in the packed decimal format.

Answer: a) This has five digits, so 19 37 2C

b) This has four digits, so 08 19 1D

c) This has six digits, so 04 32 15 4C Decimal point is not stored.

d) This has six digits, so 03 14 15 9D Decimal point is not stored.

Comment: There is not much method to show for this one, so I did not deduct for just showing the answer.

Also: There must be an odd number of decimal digits. Writing c) as 432154C is incorrect; this is 6 decimal digits.

7. 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`**

8. 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`**.

9. The following numbers are given in System/360 Packed Decimal format. Perform the indicated additions and show the sum as a packed decimal.
   HINT: I would first convert to standard decimal, add them, and convert to packed format

   a) The sum of 012C and 47344D.

   b) The sum of 019D and 019C.

   c) The sum of 32256C and 128D.

   d) The sum of 044D and 122D.

Answer: a) This is 12 – 47344 = – 47332. 47 33 2D

b) This is 19 – 19 = 0 0C. (0D is wrong)

c) This is 32256 – 128 = 32128 32 12 8C

d) This is –44 – 122 = –166 16 6D

Comment: **The answer to c) is `0C`. The standard defines `0D` as incorrect.**

We must have a sign half–byte in this notation.
`0000` is not a correct answer. It must have a sign indicator.
`0` is also not a correct answer. It must have a sign indicator.

Also: **Some students want to show the binary equivalent of each answer, such as `0100 0111 0011 0011 0010 1101` for a).**
This is not necessary and tends to be a bit confusing.

10. 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.

Comment: **The purpose of this problem is to remind one that the decimal point is not stored. Consider the answer `0102345C`. It could represent any of the following: 1.02345, 10.2345, 102.345, 1023.45, etc.**

11. 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**

b) **032C** + **027D**

c) **10003C** + **09989D**

d) **666D** + **444D**

e) **091D** + **0C**

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** + **09989D** represents 10003 –9989 = 14. This is represented as **01 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**.