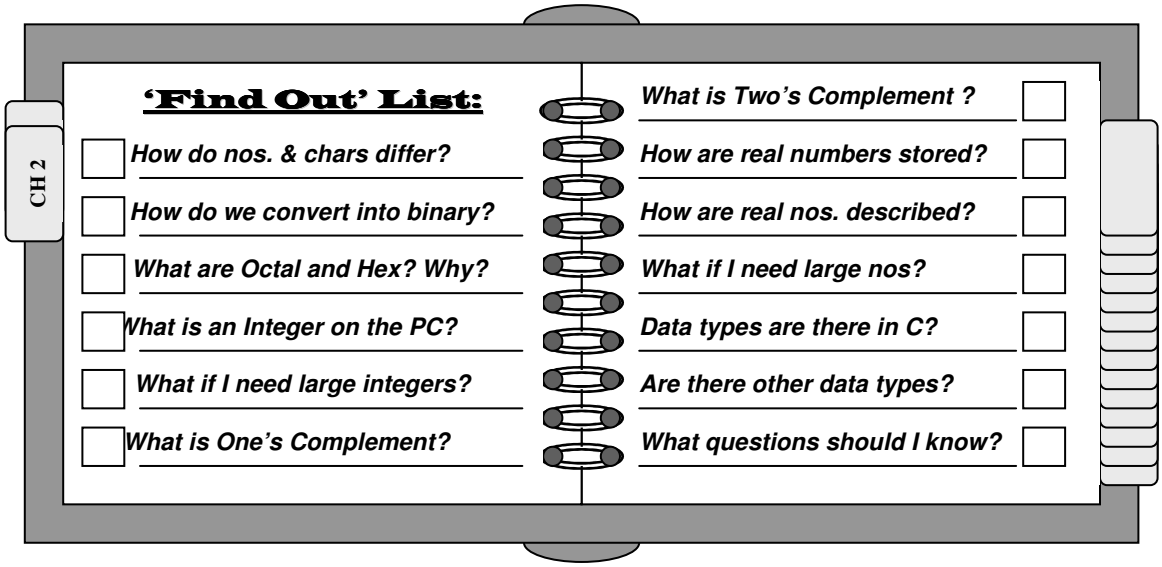
# CHAPTER 2:
# BASIC DATA TYPES

*"Simplicity of life, even the barest, is not a misery,*
*but the very foundation of refinement"*
*William Morris (1834–96)*




## Introduction

In this chapter, we consider two new basic data types: ***integers***, and ***reals*** (or floating point numbers). We also continue our discussion of the data type which we introduced in the previous chapter: ***characters***. All of these data types are, in turn, composed of bits.

> ***We Already Know*:**
> **'Donuts' are arbitrarily grouped as bytes.**

Although we refer to these data types as *basic*, they are, as we saw before, also abstract data types. Keep in mind, that as far as the computer is concerned, they are stored as high and low voltages. How we group them together, and how we interpret the groupings, is totally up to us (or at least the language designers). *Must the sequence of bits 01000001 be interpreted as the character 'A'?* Of course not. It is just a convention that we understand (unless you are using EBCDIC). The same holds true for our interpretation of basic data types. For example, we later talk about the sign-bit being the leftmost bit (if that doesn't make sense right now, it will). *Must it be that way?* No, we could have made it the rightmost bit (although it might change the way we manipulate the bits)

This is an extremely important chapter. As we noted in the Section overview, these basic data types are the building blocks for all of the following abstract data types to come. At

times, we may seem overly repetitious, but that is only because we wish to emphasize the importance of these data types.

## Characters vs. Numbers

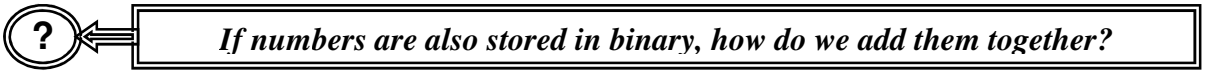
It is important to note that in the above sections, we were talking about character representation. ASCII, as we noted, is a scheme for representing characters or symbols, and nothing else. Humans typically view the set of digits {0..9} as either numbers OR characters/symbols.


Usually, we do NOT perform mathematical operations with characters or symbols (although, in fact, we *can*). For example, certainly, we would not think of adding the strings to the right[1]:

"**Jumping"**
**+ "Jack"**
**"Flash"**

As we saw in the previous chapter, characters are agreed upon interpretations for our sequence of bits. For our purposes, we will be discussing characters as they correspond to the ASCII. Coding scheme.

✓ ***Do nos. & chars differ?***




It is actually a very simple procedure, but we do need to go over some basic rules first.

## Adding Numbers in Binary

Before we begin our discussion, note that we are loosely using the term 'numbers'. In this text, we will discuss two types of numbers: ***Integers*** and ***Real Numbers***. Integers, by definition, are whole numbers (such as 0, 1, 567, –23). Real Numbers are rational (or irrational) numbers. We will discuss real numbers as ***floating point*** numbers, or such numbers as 123.456, 0.00043, -2.987. For now, we will illustrate how integers can be represented in binary.

Adding binary numbers is much simpler than adding numbers in decimal, but it might not seem that way initially. If you think back to when you first started to added, you will remember that you needed first learn all of the different combinations in an addition table: 1+1=2, 1+2=3, ... 2+1=3, 2+2=4, ... 9+7=16, 9+8=17, 9+9=18. There are 45 unique combinations (assuming you understood transitivity, i.e., 2+3 =3+2) which you had to learn, *not* counting the rule that 0 added to any number is that number.

[1] Strings actually can be added together in a process known as *concatenation*. "Jumping" + "Jack", however, would yield the string "JumpingJack".

The rules for adding binary numbers are considerably simpler. There are only 3 (or 4) unique combinations, *including* the rule that 0 added to any number is that number. The *entire* set of combinations (including transitivity) are:

Calculation 2.1.

```
  0      1      0      1
+ 0    + 0    + 1    + 1
---    ---    ---    ---
  0      1      1     10
```

Of the examples above, the last one seems to give students difficulties. It is really no different than, for example, adding 6 and 8 to get 14. In both cases, we add single digit numbers to get a two digit number. Carrying over numbers is the same in binary as it is in decimal:

```
Carry-Over:   1111111
               11011010
            + 10111111
              ---------
              110011011
```

Calculation 2.2.

Now let's add 3+4. We know that the answer is 7, but if we add the numbers in binary (as given in the ASCII table, Addendum 1.1.), we come up with:

```
  0110011   (ASCII Code for '3')
+ 0110100   (ASCII Code for '4')
-------
  1100111   (ASCII Code for 'g')
```

Calculation 2.3.


Nothing, really. We have repeatedly stated that ASCII is a coding scheme for representing *characters and symbols only*. Just because we see a symbol which could also be interpreted as a digit (e.g., 8) displayed, that does *not* mean that it is necessarily a number, any more than a license plate *number(?)* QV-123X is a number.

We have already seen the approach necessary to represent numeric values when we began our discussion of bits. The progression of numbers in binary is very simple: 0 1 10 11 100 101 110 111 ... Table 2.1 shows this progression for the integers 0 through 49.

Table 2.1.

| Decimal | Binary | Decimal | Binary | Decimal | Binary | Decimal | Binary | Decimal | Binary |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 10 | 1010 | 20 | 10100 | 30 | 11110 | 40 | 101000 |
| 1 | 1 | 11 | 1011 | 21 | 10101 | 31 | 11111 | 41 | 101001 |
| 2 | 10 | 12 | 1100 | 22 | 10110 | 32 | 100000 | 42 | 101010 |
| 3 | 11 | 13 | 1101 | 23 | 10111 | 33 | 100001 | 43 | 101011 |
| 4 | 100 | 14 | 1110 | 24 | 11000 | 34 | 100010 | 44 | 101100 |
| 5 | 101 | 15 | 1111 | 25 | 11001 | 35 | 100011 | 45 | 101101 |
| 6 | 110 | 16 | 10000 | 26 | 11010 | 36 | 100100 | 46 | 101110 |
| 7 | 111 | 17 | 10001 | 27 | 11011 | 37 | 100101 | 47 | 101111 |
| 8 | 1000 | 18 | 10010 | 28 | 11100 | 38 | 100110 | 48 | 110000 |
| 9 | 1001 | 19 | 10011 | 29 | 11101 | 39 | 100111 | 49 | 110001 |

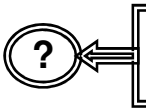
*Now*, going back to our previous example, if we add 3 + 4 (using the numbers from the table), we find that the calculations are correct (since the result is 7 (or $111_2$).

| | | | |
|---|---|---|---|
| **3** | = | **11** | (binary) |
| **+ 4** | = | **100** | (binary) |
| **7** | = | **111** | (binary) |

Calculation 2.4

***How are characters actually stored?***

As we have alluded, there are actually no such things as characters. Characters are actually stored as numbers (integers) and displayed as characters (according to the ASCII coding scheme). For example, the character '0' (zero) is stored as the numeric value 48, or in binary as 0110000 (check table 2.1). *Why 48?* If you check the ASCII table from the appendix in chapter 1, you will find the character '0' associated with the 48th character in the sequence (the 49th, actually, since the sequence starts off with zero). Hence, you will sometimes hear a reference made to ASCII 48 (the character '0') or ASCII 27 (the Escape character), or ASCII 32 (blank space) or ASCII 97 (lower case 'a').



***Why don't the characters '0' .. '9' have the same sequence of bits in ASCII as as the digits 0 .. 9 ??***

No real reason, although there are some efficiency considerations. One reason is because they wanted the characters '0' .. '9' to be viewed uniquely from their numeric counterparts.

**CAVEAT**

From this point on, we will begin including examples in both C and C++. The functionality of both languages tend to be somewhat compiler specific, as well as dependent upon error and warning level settings. While we will try to be generic in our code, the student is advised to check their compiler settings.

In the C/C++ programming language, you can refer to characters either by their numeric value or by the character associated with a value. Consider the following snippets of code:

***Note*:**
**Since C is a subset of C++, and we will be using examples from both, we will refer only to C++**

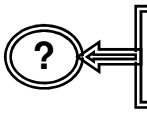
C Code 2.1

```
char ch;                    // ch is the variable we will use to hold the ASCII character
ch = 'T';                   // #1: Assign the ASCII Character T to the variable ch
printf("The character is    : %c\n",ch);        // Output: The character is: T
printf("The decimal value is: %d\n",ch);        // Output: The decimal value is 84
ch = 50;                    // #2: Assign the numeric value 50 to the variable ch
printf("The character is    : %c\n",ch);        // Output: The character is: 2
printf("The decimal value is: %d\n",ch);        // Output: The decimal value is 50
```

C++ Code 2.1

```
char ch;                    // ch is the variable we will use to hold the ASCII character
ch = 'T';                   // #1: Assign the ASCII Character T to the variable ch
cout <<"The Character is " << ch << endl;          // Output: The character is: T
cout <<"The Decimal Value is " << (int) ch << endl;  // Output: The decimal value is 84
ch = (char) 50;             // #2: Assign the numeric value 50 to the variable ch
cout <<"The Character is " << ch << endl;          // Output: The character is: 2
cout <<"The Decimal Value is " << (int) ch << endl;  // Output: The decimal value is 50
```

Notice that BOTH numeric data types and characters can be assigned to the variable *ch, although it is not considered good programming style* (which is why in the C++ code first cast the numeric value as a character). In case #1, we assign the ASCII character 'T', which is really the value 84. In Case #2, we assign the numeric value 50, which corresponds to the ASCII character '2' (NOT the numeric value 2).



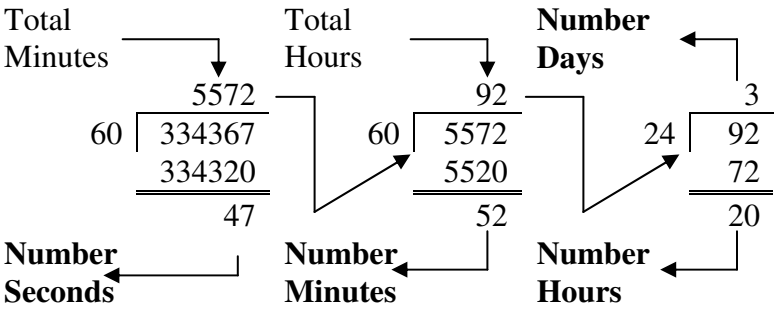

***The Table (2.1) only goes up to 49. What If I need to know the binary representations for larger numbers ?***

Any integer can be converted to binary. We need to learn the procedure for doing so, however.

## Converting from Decimal to Binary and Binary to Decimal

The process of converting from decimal to binary is very easy, and is similar to ones which are used every day. For example, if I were to ask you how many days, hours, minutes and seconds are in 334,367 seconds you would probably go through the following process shown in Figure 2.1.

Figure 2.1



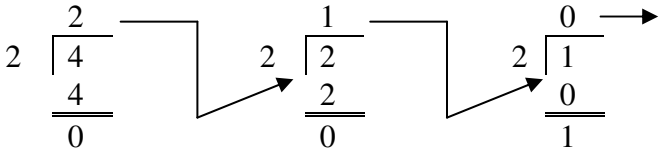

Thus, there are 3 days, 20 hours, 52 minutes, and 47 seconds in 334,367 seconds. The method we employed is also known as modulus arithmetic, which yields a quotient (e.g., the quotient of 334367/60 is 5572) and a remainder (e.g., the remainder of 334367/60 is 47). In some computer languages (e.g., Pascal), we would use the keywords **DIV** to get the quotients (e.g., 334367 **DIV** 60 = 5572) and **MOD** to get the remainder (e.g., 33467 **MOD** 60 = 47). In C, we use the division ( **/** ) operator to get the quotient (in modulus arithmetic, the quotient of two integers *must* always be an integer), and the symbol **%** (percent sign) to get the remainder.

Notice that we first obtained the number of seconds (47), then the number of minutes (52), then the number of hours (20), and finally the number of days (3). However, because we wanted Days, Hours, Minutes, and Seconds, we collected from last to first (we mention that because it will come into play a little later).


We know that there can never be more than 60 seconds in a minute, so we divided by 60. We know that there can never be more than 60 minutes in an hour, so we divided by 60. We know that there can never be more than 24 hours in a day, so we divided by 24.


The procedure is the same. We know that there can never be more than two values (0 and 1), so all we have to do is divide by 2. For example, we all ready know (from the table), that $4_{10} = 100_{2.}$ Following the same procedures:

Figure 2.2.




**Note**: There is no reason to continue since 0 divided by 2 will always yield a remainder of 0.

As we did before, we gather the remainders from right to left, and hence we know $4_{10} = 100_{2.}$ Using the **MOD** (in Pascal) or **%** (in C) and **DIV** (in Pascal) or **/** (in C) notation we introduced earlier, the procedure could be given as:

Figure 2.3.

| Quotient | Remainder | | Quotient | Remainder |
|---|---|---|---|---|
| 4 **DIV** 2 = 2 | 4 **MOD** 2 = 0 | **OR:** | 4 / 2 => 2 | 4 % 2 = 0 |
| 2 **DIV** 2 = 1 | 2 **MOD** 2 = 0 | | 2 / 2 => 1 | 2 % 2 = 0 |
| 1 **DIV** 2= 0 | 1 **MOD** 2 = 1 | | 1 / 2 => 0 | 1 % 2 = 1 |

***And collecting from last to first, or bottom to top,*** we, find that **$4_{10} = \underline{100_{2.}}$**

The procedure is quite simple:

**Conversion to Binary Procedures**

1. Get the quotient AND the remainder of the decimal and the base-to-be-converted-to (in this case, base 2)
2. Store the remainder
3. IF the quotient is 0, the result is the string of remainders from latest to first (Reverse of order stored).
4. IF the quotient is greater than 0, set the new decimal equal to the quotient and go to step 1.

Notice that in the algorithm, we stated 'base-to-be-converted-to'. The algorithm will work for any base (greater than 1). We'll have examples of converting to Octal (base 8) and Hex (base 16) a little later.

Let's try a longer one. Let's Convert the decimal $208_{10}$ to binary.

Figure 2.3.

| Quotient | Remainder | | Quotient | Remainder |
|---|---|---|---|---|
| 208 **DIV** 2 = 104 | 208 **MOD** 2 = 0 | **OR:** | 208 / 2 = 104 | 208 **%** 2 = 0 |
| 104 **DIV** 2 = 52 | 104 **MOD** 2 = 0 | | 104 **/** 2 = 52 | 104 **%** 2 = 0 |
| 52 **DIV** 2 = 26 | 52 **MOD** 2 = 0 | | 52 **/** 2 = 26 | 52 **%** 2 = 0 |
| 26 **DIV** 2 = 13 | 26 **MOD** 2 = 0 | | 26 **/** 2 = 13 | 26 **%** 2 = 0 |
| 13 **DIV** 2 = 6 | 13 **MOD** 2 = 1 | | 13 **/** 2 = 6 | 13 **%** 2 = 1 |
| 6 **DIV** 2 = 3 | 6 **MOD** 2 = 0 | | 6 **/** 2 = 3 | 6 **%** 2 = 0 |
| 3 **DIV** 2 = 1 | 3 **MOD** 2 = 1 | | 3 / 2 = 1 | 3 **%** 2 = 1 |
| 1 **DIV** 2 = 0 | 1 **MOD** 2 = 1 | | 1 **/** 2 = 0 | 1 **%** 2 = 1 |

***Collecting from last to first, or bottom to top,*** we find that **$208_{10}$ = $\underline{11010000}_2$.**

The C/C++ code follows the procedures given above exactly. However, there is one little 'twist', which need not be noted at this time. We will return to conversion from decimal to binary in Chapter 5 (Strings).


Calculation 2.5.

We already have some clues that the number is probably correct. For instance, we know how many bits are needed to represent the number $208_{10}$:

$$\mathbf{n = \lceil log(I)/.30103 \rceil = \lceil log(208)/.30103 \rceil}$$
$$\mathbf{= \lceil 2.319/.30103 \rceil = \lceil 7.700 \rceil = 8}$$

So, we know that we have the correct number of bits. *But* we still can't be sure unless we can convert from binary to decimal to check our work. Converting from binary to decimal may appear more confusing at first, but in fact it is also very easy. Keep in mind that we have a number of different ways for representing the same number (in the same base). For example:

Calculation 2.6.

$$\begin{aligned}
\mathbf{12{,}056_{10}} &= \mathbf{10{,}000 + 2{,}000 + 0 + 50 + 6} \\
&= \mathbf{1 * 1{,}000 + 2 * 1000 + 0 * 100 + 5 * 10 + 6 * 1} \\
&= \mathbf{1 * 10^4 + 2 * 10^3 + 0 * 10^2 + 1 * 10^1 + 6 * 10} \\
&= \mathbf{12{,}056}
\end{aligned}$$

The same approach can be applied to other conversions, regardless of the base used. For example, we could represent the binary equivalent of $208_{10}$ ($11010000_2$) as:

Calculation 2.7.

$$\begin{aligned}
11010000_2 &= 1000000 + 1000000 + 000000 + 10000 + 0000 + 000 + 00 + 0 \\
&= 1 * 2^7 + 1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^1 + 0 * 2^1 + 0 * 2^0 \\
&= 1 * 128 + 1 * 64 + 0 * 32 + 1 * 16 + 0 * 8 + 0 * 4 + 0 * 2 + 0 * 1 \\
&= 128 + 64 + 0 + 16 + 0 + 0 + 0 + 0 \\
&= 208_{10}
\end{aligned}$$

Notice that we have gone about it the hard way. Whenever a digit is 0 (zero), it need not be considered in our calculations since 0 times any number is 0. We could have rewritten the example as:

Calculation 2.8.

$$\begin{aligned}
11010000_2 &= 1000000 + 1000000 + 10000 \\
&= 1 * 2^7 + 1 * 2^6 + 1 * 2^4 \\
&= 1 * 128 + 1 * 64 + 1 * 16 \\
&= 128 + 64 + 16 \\
&= \underline{208}_{10} +
\end{aligned}$$

Notice also that even our simplified method of calculation can be further simplified, since 1 (one) times any number is that number:

Calculation 2.9.

$$\begin{aligned}
11010000_2 &= 1000000 + 1000000 + 10000 \\
&= 2^7 + 2^6 + 2^4 \\
&= 128 + 64 + 16 \\
&= \underline{208}_{10} +
\end{aligned}$$

Let's take one more, slightly longer, example[2]. Let's convert the integer $479_{10}$ to binary:

| Quotient | Remainder |
|---|---|
| 479 / 2 = 104 | 479 % 2 = 1 |
| 239 / 2 = 52 | 239 % 2 = 1 |
| 119 / 2 = 26 | 119 % 2 = 1 |
| 59 / 2 = 13 | 59 % 2 = 1 |
| 29 / 2 = 6 | 29 % 2 = 1 |
| 14 / 2 = 3 | 14 % 2 = 0 |
| 7 / 2 = 3 | 7 % 2 = 1 |
| 3 / 2 = 1 | 3 % 2 = 1 |
| 1 / 2 = 0 | 1 % 2 = 1 |

Figure 2.3.

$111011111_2$

Again, we could have predicted (approximately) that the binary equivalent would be

$$\lceil \log(479)/\log(2) \rceil = \lceil 2.68/0.301 \rceil = \lceil 8.9 \rceil = 9$$

So we need 9 bits, and we als know that the binary representation would end in a '1', since the number is odd

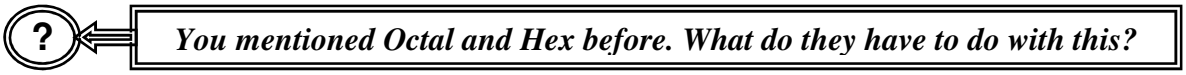
Checking the binary equivalent, we would find:

Calculation 2.10.

$$\begin{aligned}
111011111_2 &= 1 * 2^8 + 1 * 2^7 + 1 * 2^6 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 \\
&= 1 * 256 + 1 * 128 + 1 * 64 + 1 * 16 + 1 * 8 + 1 * 4 + 1 * 2 + 1 * 1 \\
&= 256 + 128 + 64 + 16 + 8 + 4 + 2 + 1 \\
&= \underline{479}_{10}
\end{aligned}$$

[2] From this point on, we will use only C/C++ notation

The basic concepts (conversion from decimal to binary or binary to decimal) are the same ***regardless*** of the base.




We often see Octal (base 8) and Hexadecimal (base 16) used in computer literature (if you check, for example, any ASCII or EBCDIC table, in addition to indicating the decimal value of the coding scheme, it will also list Octal and Hex Values). *Why?* Basically because it is convenient to convert from Binary to Octal, or from Binary to Hex (and vice-versa). Much easier than it is from Binary to Decimal (or vice versa).

## Octal

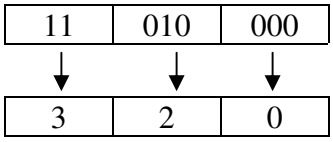

As the term implies, there are only 8 digits (0 .. 7) in the ***Octal*** numbering system (just as there are 10 (0 .. 9) in decimal, and 2 (0, 1) in binary). The advantage, when it comes to binary, is the correspondence. For example, a byte (8 bits) can be readily, and quickly, represented by three octal digits, since $2^3 = 8$:

Table 2.2.

| **Octal Number** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **Binary Number** | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

A nice match. But it gets even better. Take, for example, the number $208_{10}$. We already know that the binary equivalent is $11010000_2$ (from Figure 2.3. and calculation 2.10.). In Octal it is:

Figure 2.4.



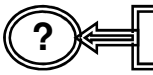

Which is a Direct transfer from Table 2.2


Figure 2.5.

| **Quotient** | **Remainder** |
|---|---|
| 208 / 8 = 26 | 208 % 8 = 0 |
| 26 / 8 = 3 | 26 % 8 = 2 |
| 3 / 8 = 0 | 3 % 2 = 3 |

Remember, we previously said that the procedures we used are the same for *any* base. Using the same procedure we used to convert from decimal to binary, we can convert from decimal to octal.

→ **$320_8$**




***How do we convert from Octal to Decimal ??***

Calculation 2.11.

| | | | | | | |
|---|---|---|---|---|---|---|
| $320_8$ | = | $3 * 8^2$ | + | $2 * 8^1$ | + | $0 * 8^0$ |
| | = | 3 * 64 | + | 2 * 8 | + | 0 * 1 |
| | = | 192 | + | 16 | + | 0 |
| | = | $\underline{208}_{10}$ | | | | |

We use essentially the same method we used we converted from binary to decimal: We keep track of the exponent position and the value of the digit at that position. The only difference is that we can have the values 0..7. To convert the octal number $320_8$, we follow the procedure given in calculation 2.11.


***Does it work the same for Hex (Hexadecimal) ??***

***Absolutely***. Nothing has changed. As we said earlier, it works the same for *ALL* bases.

## Hexadecimal (Hex)


The rational for using ***Hexadecimal*** (base 16) is the same as that for using Octal. Given 4-bits, we can represent up to 16 pieces of information, which is convenient since we need only 2 Hexadecimal digits to represent a byte (since 1 byte = 8 bits = $2^8$ = 256 = $16^2$ = 2 Hex digits). The *disadvantage* is that instead of memorizing 8 conversion tables (as in octal), one needs to memorize 16 conversion tables (Actually 14 since $0_2 = 0_{16}$ and $1_2 = 1_{16}$, or 7 if you already know the Octal conversions). Additionally, while all 8 digits are found in the decimal digit set (i.e., 0 = 0, 1 = 1, ...., 6 = 6, 7 = 7), Hex contains six symbols which are not part of the decimal digit set. The correspondence between Decimal and Hex is as follows:

Table 2.3.

| Decimal | Hex | Decimal | Hex | Decimal | Hex |
|---|---|---|---|---|---|
| 0 | 0 | 6 | 6 | 12 | C |
| 1 | 1 | 7 | 7 | 13 | D |
| 2 | 2 | 8 | 8 | 14 | E |
| 3 | 3 | 9 | 9 | 15 | F |
| 4 | 4 | 10 | A | | |
| 5 | 5 | 11 | B | | |


***Why do we need 'A' through 'F'? Why can't we use 10 through 15 ??***

Because the symbols 10 (through 15) actually consist of 2 symbols each. For example the number 13 consists of the symbols '1' and '3'. For any base, the elements of that base are represented by a *unique* symbol. In decimal, the number 10 is represented by two digits: '1' and '0'

Nonetheless, even though the Hex numbering system contains more digits than the decimal number system, none of the rules of conversion have changed. Consider the (decimal) number $197_{10}$, which in Binary would be:

. Figure 2.6

| Quotient | Remainder |
|---|---|
| 197 / 2 = 98 | 197 % 2 = 1 |
| 98 / 2 = 49 | 98 % 2 = 0 |
| 49 / 2 = 24 | 49 % 2 = 1 |
| 24 / 2 = 24 | 24 % 2 = 0 |
| 12 / 2 = 6 | 12 % 2 = 0 |
| 6 / 2 = 3 | 6 % 2 = 0 |
| 3 / 2 = 1 | 3 % 2 = 1 |
| 1 / 2 = 0 | 1 % 2 = 1 |

→ **$11000101_2$** Which we know is **$197_{10}$** since

**$2^7 + 2^6 + 2^2 + 2^0 = 128 + 64 + 4 + 1 = 197$**

We also know that in Octal, the number is:

**111 000 101**

**7 0 5**

We further know (from the rules stated above), that in Hexadecimal the decimal number $197_{10}$ would be:

Figure 2.7.

| Quotient | Remainder |
|---|---|
| 197 / 16 = 12 | 197 % 16 = 5 |
| 12 / 16 = 0 | 12 % 16 = 12 = C |

→ **$C5_{16}$** Which we know is **$197_{10}$** since

**$C(=12)*16^1 + 5*16^0 = 12*16 + 5*1 = 192 + 5 = \underline{197}_{10}$**

As far as converting from Hex to decimal (or vice versa), it is merely a matter of using conversion Tables (or memorizing the conversion) just as we did with Octal:

Table 2.4.

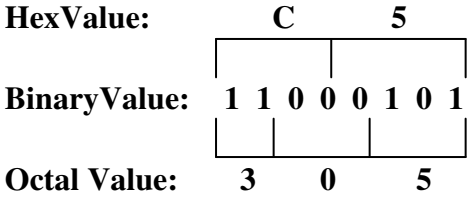
| Binary | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hex | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

Therefore (according to the Tables): **C** **5** = Hexadecimal Equivalent

**$197_{10} = 11000101_2 =$** **1100** **0101**

Figure 2.8.

To convert between Hex and Octal, or Octal and Hex, The easiest method is to use binary, as in Figure 2.8.




✓ ***What are Octal and Hex? Why?***

Note from Addendum 1.1 (and Addendum 1.2.), that any character can be represented using combinations of 2 Hex Digits.

# Characters

We have seen that there is a difference between characters and numbers, in terms of how the computer interprets them. We have seen that characters are stored according to ASCII (or EBCDIC) coding schemes, and the digits (0 .. 9) may also be considered characters and stored in ASCII format. We have also seen that characters do *NOT* exist, except in the abstract. Characters are stored as numeric values on 8-bits or 1-byte.

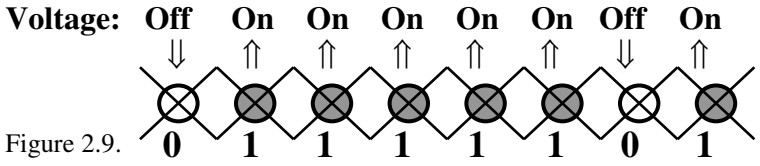
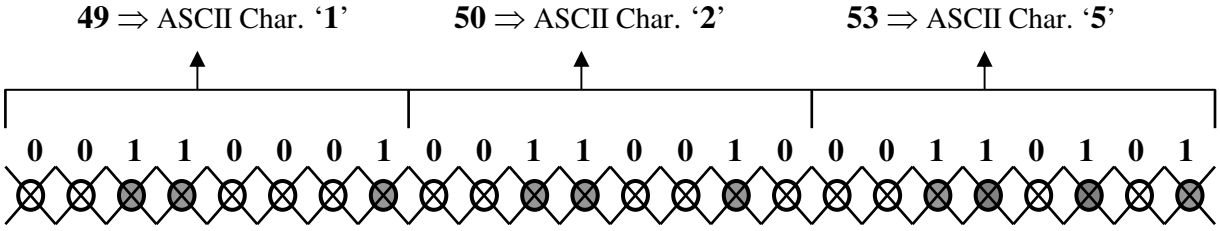



Figure 2.9.

Take, for example, the character **{** (left brace). From the ASCII table, we know that this is actually stored as the numeric value number $125_{10}$ (or $01111101_2$ on 8-bits). If we could look inside of RAM, we might see the value stored as it is in Figure 2.9.

Keep in mind that this is a ***numeric*** value. It is the numeric value 125, or ***ASCII 125*** (actually, the 126th sequence of bits in the ASCII table). It is *NOT* the character string "125". *If* we were to store the character string "125" in ASCII format, we would have to save it using three bytes (24 bits), as:

Figure 2.10.




If this was the only manner in which we could store numeric values, we would have a severe problem.


Remember, a byte contains only 8-bits. That means that we can only represent 256 pieces of information or the integers from 0 through 255.


Obviously we need more bits. Rather than trying to determine how many more bits we need, since we have been working with bytes, and 8 bits is basic unit which is addressed[3]

[3] There is such a thing as a 'nibble' which is 4-bits.

by the computer, it makes sense to see what numbers we can obtain by dealing with multiples of 8 bits. *Why not double the number of bits?*

# Integers

**CAVEAT**

***The Times, they are a changin'***. Just a few years ago, integers on PCs required 16-bits. Now, we are finding that 32-bit integers are more common, and perhaps will become the rule (certainly, Visual C++ compilers use 32-bit integers). We are introducing integers as 16-bit data types for two reasons: (1) they are initially easier to grasp, since they are intuitive transitions from characters, and (2) there are still some compilers that use 16-bit integers. We do discuss 32-bit integers a little later, but students should check their

We have already defined ***integers*** as the set of whole numbers. In fact, it is the set of all whole numbers ranging from minus infinity to positive infinity. Obviously, we would have a problem implementing that definition, since we can not have an infinite amount of memory. RAM may be getting larger with each passing day, but it will never reach infinity.

Def

If we use 16-bits (2 bytes) we know that we can represent $2^{16}$ = 65,536 numbers (or the integers from 0 to 65,535). Certainly, this is a considerably larger range than that obtained with only 8-bits. On the PC, this (16-bits) is the number allocated for an integer. However, there is one apparent restriction: we have limited ourselves to working with only non-negative numbers.

? ***How do we allow for both positive and non-positive number?***

The fact that a number is either positive or non-positive (we use this terminology because of zero; *is zero positive or negative*?), works to our advantage. Positive or non-positive is a binary condition. To represent it, we need only one bit (e.g., 0 if positive, 1 if non-positive). Therefore. if we have 16 bits to begin with, we can sacrifice 1 bit for the sign, leaving us with 15 bits ($2^{15}$ = 32,768) to use for the number.

PC Computer designers decided to allocate 16 bits for an integer, at least originally. That means that integers were stored with a 1-bit sign (the left-most bit), with the remaining 15-bits used to represent the value.

Table 2.5.

| 1-bit Sign | 15-bit Value |
|---|---|

This is the definition of the data type integer on a PC: A 16-bit signed integer, with 1-bit used for the sign, and the remaining 15-bits used to represent the value of the integer. The range for an integer is therefore $-2^{15}$ (-32,768) to $+2^{15}-1$ (+32,767).

✓ ***What is an Integer on the PC?***

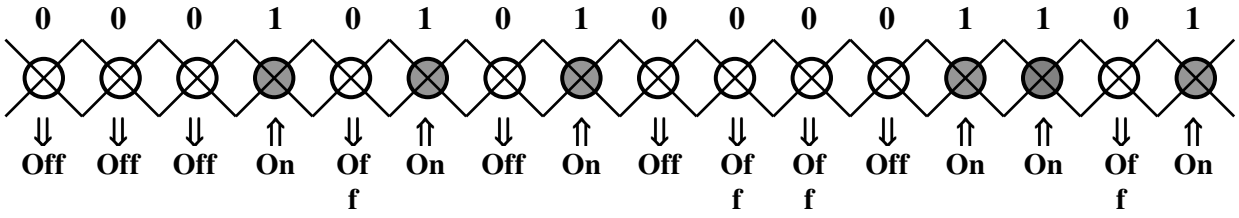
### *Why $+2^{15} - 1$ ??*

We want to represent 0 (which is neither positive nor negative). Of course, we could have a +0 and a -0, which some computer architectures did allow for (more on that later).

### *What does this mean in practice?*

Let's take the value +5,389. The binary equivalent is $1010100001101_2$ (we leave it to you to check the conversion). On 15-bits (the number of bits we have available to represent an integer value) the binary equivalent would be $001010100001101_2$. If we assign a 0 (zero) to the left-most bit to mean positive, the number becomes $0001010100001101_2$ (on 16-bits). In RAM, we would store the number as:

Figure 2.11.

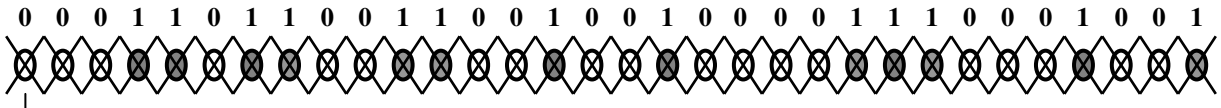



W here the left-most bit is the **Sign-Bit**, and 0 ⇒ Positive

### *Are 16-bits enough?*

Depends[4]. The IBM-PC was initially designed as a *16-bit machine* (i.e., it has a two byte **word size***)*, and uses 16-bits for an integer (see earlier footnote). Mainframes and mini-computers, which were designed as 32-bit machines use 4-bytes for an integer. Super-computers may use 62-bit integers. In the case of mainframes and minicomputers, the machine is capable of storing whole numbers between the range $-2^{31}$ to $+2^{31} - 1$ (we still need one bit for the sign and we still lose one for the value because we have to represent 0), or all the whole numbers between -2,147,483,,648 and +2,147,483,647 (inclusive). The number $456,278,921_{10} = 11011001100100100001110001001_2$ would be stored as:

Figure 12.12.

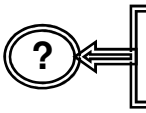



Where the left-most bit is still the sign bit, and 0 (zero) ⇒ positive

Check:

$11011001100100100001110001001_2 = 2^{28} + 2^{27} + 2^{25} + 2^{24} + 2^{21} + 2^{20} + 2^{17} + 2^{14} + 2^{9} + 2^{8} + 2^{7} + 2^{3} + 2^{0}$
= 268,435,456 + 134,217,728 + 33,554,432 + 16,777,216 + 2,097,152 + 1,048,576 + 131,072
+ 16,384 + 512 + 256 + 128 + 8 + 1
= **456,278,921**

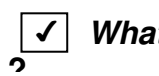
[4] Keep our Caveat in mind



***Does that mean that we can never have integers larger than 32,767 (or smaller than -32,767) on the PC?***

*Yes and no.* On some older machines, and older software packages, maybe. As mentioned earlier, some modern PC applications, such as Windows 95 and 98, allow for 32-bit integers[5]. The standard data type **int** (integer) can still be16-bits, however. There is also a variable type called **short**[6] (or **unsigned short**), which uses the 16-bits as an *signed* (by default, unless the *unsigned* prefix is used) integer (much like the numeric byte we talked about earlier). In other words (since we do not need to devote 1-bit to the sign) we can represent the non-negative whole number 0 through 65,535 (still a total of $2^{16}$ = 65,536 pieces of information).

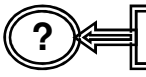
The C++ programming language also allows for a data type called **long**[7] (or **unsigned long**). Intuitively, we could have predicted that this data type uses 32-bits without the sign-bit, meaning that we can represent the whole number from 0 through 4,294,967,295 ($2^{32}$ = 4,292,967,296), inclusive. On mainframes, however, the data type **long** uses 64-bits, meaning that (signed) longs take on (approximately) the range –9,223,372,036,854,780,000 to 9,223,372,036,854,780,000, and unsigned longs cover the range 0 through 18,446,744,073,709,600,000.



***What if I need larger integers?***

Now, if you really need large integers, get a supercomputer, such as a Cray. A word size on a Cray is 128-bits. That means a signed integer can take on the values from $-2^{127}$ to $+2^{127}$ - 1. That means you can represent the whole numbers from, well, ... it's a large number:

-1.701411834605e+38 to +1.701411834605e+38 - 1. A number to 38 places. *Very* big.



***How does the computer actually do calculations with integers?***

## Manipulating Integers

Adding positive integers together is not a big deal, and follows the procedures we previously described (see Calculation 2.1.). For example, if we wished to add the integers $312_{10}$ (=$100111000_2$, or $0000000100111000_2$ using 16-bits) and $312_{10}$, we would add them as:

[5] Even though Windows 95 and 98 allow for 32-bit integers, as we noted earlier, many of the C compilers in use (at the time of this writing) still assume 16-bit integers.
[6] Even if ALL PCs use 32-bit integers, shorts will still require 16-bits.
[7] As of this writing, the data type in Visual C++ long still uses 32-bits.

Calculation 2.11.

| Carry-Over | | | | | | | 1 | | 1 | 1 | 1 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **312:** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** |
| **+ 312:** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** |
| **624:** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **0** |

Which is correct since:

$2^9 + 2^6 + 2^5 + 2^4 =$
$512 + 64 + 32 + 16 =$ **624**

Using our scheme for signed integers (i.e., allocating 1-bit for the sign), we (sort-of) implied that a negative integer (or non-positive) had the same value as a positive integer, and that only the sign bit would vary (from 0 for positive, to 1 for negative).

Well, *not quite*.

Calculation 2.12.

This time, let's add the values $-312_{10}$ and $+312_{10}$. We know that if we add $-312_{10}$ and $+312_{10}$ the result is 0. Since we know the binary equivalent of $+312_{10}$, the binary equivalent of $-312_{10}$ should be the same, except that the sign bit (the left-most bit) will have a '1' instead of a '0'. But when we add the two values together we find that the resultant value is actually $-624_{10}$, or $1000001001110000_2$.

| Carry-Over | | | | | | | | 1 | | | 1 | 1 | 1 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **312:** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** |
| **+ -312:** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** |
| **-624:** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** | **0** |


Yes, something is wrong. As it turns out, one of the advantages of using binary (as opposed to decimal), is that manipulation of bits is very easy, although slightly different than what we are accustomed to.


In binary, the negative of a positive number is the *opposite* of that number. For example, the negative of the positive number $0000000100111000_2$ would be $1111111011000111_2$ (the '0' bits have been changed to '1' and the '1' bits have been changed to '0'). This procedure is known as **complementing**[8], or simply changing the value of the bit (i.e., if the value is 0, it becomes 1; if the value is 1, it becomes 0)

Calculation 2.13.

It may look and sound strange, but let's try the same addition using this scheme.

| **312:** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **1** | **0** | **0** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **312 Complemented** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **0** | **0** | **0** | **1** | **1** | **1** |
| **???:** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |

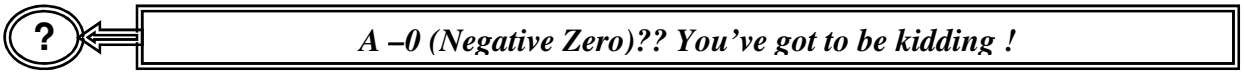

[8] Note the spelling. While this is a nice number, we are NOT compl**i**menting it

Keep in mind our earlier statement: The left-most bit is the sign-bit, and if it is '0' it is positive, and if it is '1' it is negative. Also, remember our later statement: If the number stored is negative, then it is stored as a *complemented* number.


Figure 2.13.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Complement:** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| | | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ |
| | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

To interpret a complemented value, we must again complement it (or uncomplement, if you will). Since, for our example, the left-most bit is a '1', it means it is negative. To determine the value, we must complement the other 15-bits. When we do, we find the value is actually **–0**.



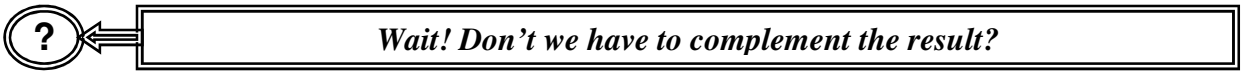

In some (older) machines, this is a legal value. After all, -0 = +0, since 0 is neither positive nor negative. In these machines, which rely on **One's Complement**, assuming a 16-bit integer, the range of integers is $-2^{15} - 1$ (-32,767) to $+2^{15} - 1$ (+32,767). We give-up one integer (not such a big loss), and must understand that -0 = +0, but it works just fine.

One's complement is not always confusing. Let's take another example, using one's complement. Let's add $-1{,}248_{10}$ (where $1{,}248_{10} = 10011100000_2$, and the one's complemented value is $1111101100011111_2$), and the number $2{,}456_{10}$ ($100110011001_2$) which should adds to 1,208 ($10010111000_2$).

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Carry-Over | 1 | 1 | 1 | 1 | 1 | | 1 | 1 | | | | 1 | 1 | 1 | 1 | 1 |
| **-1,248:** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **1** |
| **+2,456:** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **1** | **0** | **0** | **1** | **1** | **0** | **0** | **1** |
| **1,208:** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **1** | **0** | **1** | **1** | **1** | **0** | **0** | **0** |

Carry-over Dropped

Calculation 2.14.




No. As we mentioned above, we have to complement *only* non-positive results (i.e., if the sign bit is 1). The result of our addition here is positive (i.e., the sign bit is 0).

Figure 2.14.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Carry-Over | | | | | | | | | | | | | 1 | | | |
| **4:** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** |
| **+ -8:** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **1** |
| **Sum:** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** |
| **Complement** | | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ |
| **Result:** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** |

Let's take a simple example where the result is negative. Let's add +4 ($100_2$) and –8. Since $8_{10} = 1000_2$, then when we complement, the value -8 is $1111111111110111_2$.

The result, of course, is $-4_{10}$ ($111111111111011_2$), but in order to find this out, we must again complement the sum of our binary addition ($0000000000000100_2 = 4_{10}$).

For one's complement, the rules are relatively simple:

**One's Complement**

1. If a number is negative, complement all the bits (including the sign-bit)
2. If the resultant number is negative, complement all the value bits (except the sign-bit).
3. If there is a carry-over from the sign-bit, disregard it.

✓ ***What is One's Complement ?***

It may initially seem complicated, but in fact it is very simple. Don't forget that that the only real attribute the computer has is that it is fast. *How long does it take to complement a number?* How many MHz is your computer running at?

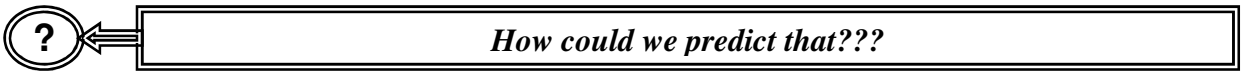
It's a very fast methodology. The only draw-back is that we can obtain negative values for 0 (and we lose 1 integer value, since we represent both +0 and -0).


Yes, but we need to modify our procedure slightly.

## Two's Complement

We can eliminate a negative zero (-0) using a procedure called **two's complement**, which is almost universally used to represent negative numbers. The principles are the same, as are the rules, except that it requires 1 additional step. If, *and only if*, the resultant of the additional is negative, we must **add** 1 bit **after** we perform a one's complement.

Let's take a relatively simple example: **128 + (- 128).** We could predict, without any calculations that $128_{10} = 10000000_2$ on 8-bits Or $0000000010000000_2$ on 16-bits




Some integers are readily converted into binary based on what we already know. Obviously, we know how the decimal values 0 and 1 would appear in binary (the same). Some can be readily determined based on what we know about how many bits it takes to represent an integer. For example, we know that given 7-bits, we can represent 128 pieces of information, or the integers from 0 through 127. That means that $1111111_2$ *must* represent the value $127_{10}$. If that is true, then 1 more than that ($1111111_2 + 1 = 10000000_2$) must represent the value $128_{10}$. Consider the following representations:

Table 2.6.

| No. Bits | Amount of Information | Range | Known Conversion #1 | Known Conversion #1 |
|---|---|---|---|---|
| 3 | 8 | 0 – 7 | $7_{10} = 111_2$ | $8_{10} = 1000_2$ |
| 4 | 16 | 0 – 15 | $15_{10} = 1111_2$ | $16_{10} = 10000_2$ |
| 5 | 32 | 0 – 31 | $31_{10} = 11111_2$ | $32_{10} = 100000_2$ |
| 6 | 64 | 0 – 63 | $63_{10} = 111111_2$ | $64_{10} = 1000000_2$ |
| 7 | 128 | 0 – 127 | $127_{10} = 1111111_2$ | $128_{10} = 10000000_2$ |
| 8 | 256 | 0 – 255 | $255_{10} = 11111111_2$ | $256_{10} = 100000000_2$ |
| 9 | 512 | 0 – 511 | $511_{10} = 111111111_2$ | $512_{10} = 1000000000_2$ |
| 10 | 1,024 | 0 – 1,023 | $1{,}023_{10} = 1111111111_2$ | $1{,}024_{10} = 10000000000_2$ |

Now, returning to the issue at hand, let's convert the integer $-128_{10}$ into binary using two's complement.

Figure 2.15.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| First convert $128_{10}$ into binary: | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| Then complement: | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ |
| Carry Overs | | | | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| $-128_{10}$ in one's Complement: | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| Add 1: | **+** **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** |
| $-128_{10}$ in two's Complement: | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

Calculation 2.15:

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Carry-Over | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | |
| **128:** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **+ -128:** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **0:** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

The Carry-over is again dropped

*Now* when we add the numbers 128 + -128, we get the numeric value zero (0), not –0 (negative zero).

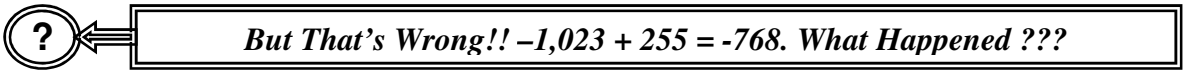
Let's take another example: Let's add $255_{10}$ and $-1{,}023_{10}$. Obviously, the result is –768. We already know the binary equivalents of $255_{10}$ ($11111111_2$ on 8-bits or $0000000011111111_2$ on 16-bits) and $1{,}023_{10}$ ($1111111111_2$ on 8-bits or $0000001111111111_2$ on 16-bits) because they are readily convertable (see Table 2.6.). Now, following our procedure:

Figure 2.16.

First convert $1{,}023_{10}$ into binary:
Then complement:
$-1{,}023_{10}$ in one's Complement:
Add 1:
Carry-Overs
$-1{,}023_{10}$ in two's Complement:
Now add $255_{10}$:
Since the Resultant is negative:
Then complement:
Yielding a value we can evaluate:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ |
| | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **+** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** |
| | | | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** |
| **+** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| | | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ |
| | | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |

$$-(1011111111_2) = -(2^9 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^1)$$
$$= (512 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1) = \mathbf{-767}$$




Calc. 2.16.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Carry-Over: | | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| **Resultant:** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** |
| **Add 1:** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** |
| **New Result:** | **0** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

$$= -(2^9 + 2^8) = -(512 + 256) = \mathbf{-768}$$

We omitted one step. Remember, when we performed two's complement, we ***added*** one (1) to the one's complemented number. When we complement a negative number, we must once again ***add*** 1 (one) to the one's complemented result.


Figure 2.17.

If we add ***after*** we complement, or subtract ***before*** we complement, we come out with the same result. For example, let's take the integer $17_{10}$ ($= 10001_2$). The two's complement of $10001_2$ is $1111111111101111_2$ (on 16-bits).

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** | **0** | **0** | **0** | **1** |
| | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ | ⇓ |
| | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **1** | **0** |
| **+** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **1** |
| | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **1** | **0** | **1** | **1** | **1** | **1** |

To evaluate the number $1111111111101111_2$ we can either:

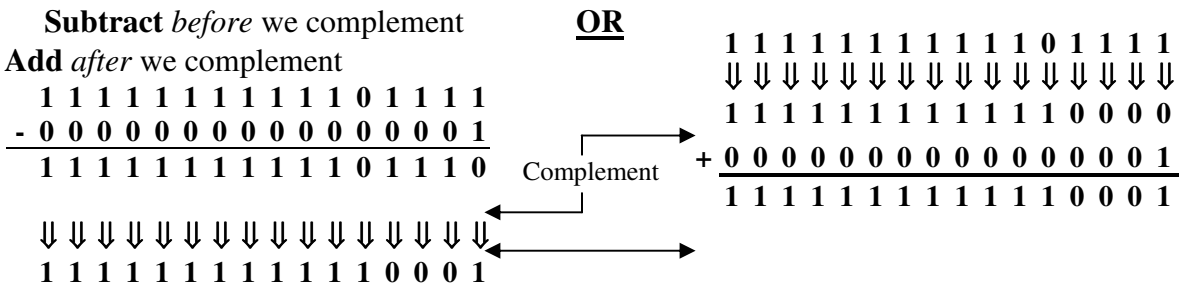
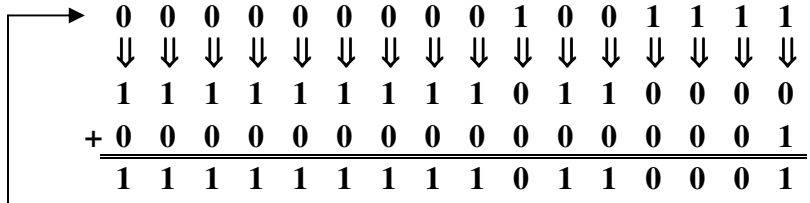



Figure 2.18.

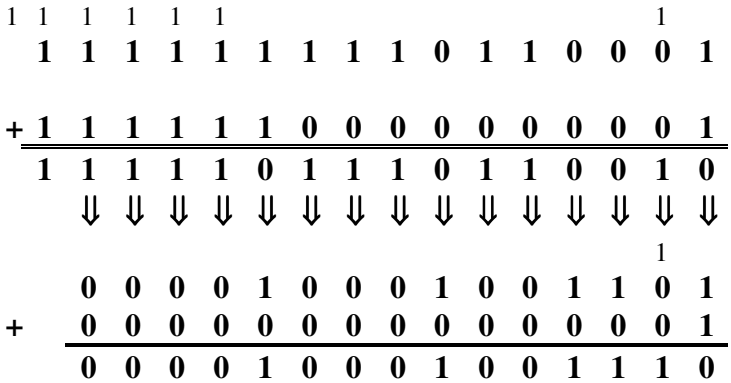
One final example: Let's add $-79_{10}$ and $-1{,}023_{10}$ (which we know from Figure 2.16. is $1111110000000001_2$ (on 16-bits) in two's complement. $-79_{10}$ would be:

Figure 2.19.

| Quotient | Remainder |
|---|---|
| 79 / 2 = 39 | 79 % 2 = 1 |
| 39 / 2 = 19 | 39 % 2 = 1 |
| 19 / 2 = 9 | 19 % 2 = 1 |
| 9 / 2 = 4 | 9 % 2 = 1 |
| 4 / 2 = 2 | 4 % 2 = 0 |
| 2 / 2 = 1 | 2 % 2 = 0 |
| 1 / 2 = 0 | 1 % 2 = 1 |




Adding the two integers together:




- 79
+ -1,023
Since the left-most bit is '1':
Complement

To get One's Complement
And add 1
Then Evaluate:

$-(2^{10} + 2^6 + 2^3 + 2^2 + 2^1) = -(1{,}024 + 64 + 8 + 4 + 2) = -1{,}102$

which is correct

Notice that if we add two complemented integers together, the values will be correct. As always, however, if the resultant value contains a '1' in the left-most digit, we will again have to complement to find out the true value.

Perhaps slightly, but at 450 MHz it really doesn't make much difference. There is another advantage to complementing: ***The computer never has to subtract.*** We may enter an equation as x = 5 - 3, but the computer will respond with x = 5 + (-3) (complementing, as we have seen, is no big deal). In fact, the computer can deal with multiplication and division by adding, although the procedures involved will not be dealt with in this text.

***What is Two's Complement?*** ☑


## Real/Floating-Point Numbers[9]

Just as it was obvious that we can't do too much if we limit ourselves to 8-bit integers, it should be equally as obvious that we can't limit ourselves to whole numbers (or even 16-bit or 32-bits, for that matter). Real Numbers, or floating-point numbers, are not limited to whole numbers; theoretically, they include all numbers from infinitely large to infinitely small, with an infinite number of values between any whole number. Practically speaking, we know that we have to do the best we can given the limitations (storage space) we have to deal with.

Real numbers pose another question for us. We saw that integers can be either negative or positive (as can real numbers), and do account for that, we assigned 1-bit for the sign. Reals also have a decimal component, which we must keep track of.


It is not as bad as it might seem. Take, for example, the two real numbers 12.345 and 123.45. *What is the major difference between the two?* Where the decimal point is.

Consider the integer 12,056. We could represent in various formats, each of them yielding the same value

Calc. 2.17.

| | | | |
|---|---|---|---|
| **12,506** | **= 1,250.6 * 10** | **= 1,250.6 * $10^1$** | **= 1,250.6 E1** |
| | **= 125.06 * 100** | **= 125.06 * $10^2$** | **= 125.06 E2** |
| | **= 12.506 * 1,000** | **= 12.506 * $10^3$** | **= 12.506 E3** |
| | **= 1.2506 * 10,000** | **= 1.2506 * $10^4$** | **= 1.2567 E4** |
| | **= 0.12506 * 100,000** | **= 0.12506 * $10^5$** | **=** |

[9] The terms real and floating-point are frequently used inter-changably. Any number can be represented as a floating point number; real numbers include the numbers between integers.

The same concepts can be applied to real numbers. For example, the real number 123.45.

Calc. 2.18.

$$\begin{aligned} 123.45 &= 12.345 * 100 = 12.345 * 10 = 12.345\ E1 \\ &= 1.2345 * 100 = 1.2345 * 10^2 = 1.2345\ E2 \\ &= 0.123 * 1{,}000 = 0.12345 * 10^3 = 0.12345\ E3 \end{aligned}$$

Any number, whether integer or floating-point, can be represented in the same fashion:

Calc. 2.19.

| | | |
|---|---|---|
| **256 = 0.256 E+3** | **-734,56682 = -0.73456482E+3** | **0.03 = 0.3 E-1** |
| **-65,536 = -0.65536 E+5** | **22.4321 = 0.224321E+2** | **-0.00567 = -0.567 E-2** |
| **94,967,291 = 0.94967291 E+8** | **-0.8 = -0.8E+0** | **0.0000034 = 0.34 E-5** |

Using this approach, it is apparent that we must keep track of three components: the ***sign*** (as before: + or -) of the number, the ***mantissa*** (which by definition is the decimal component of a number) or value (in the case of 123.45, the value .12345), and the ***characteristic*** of the exponent (in the case of 123.45 = .12345E+3, the exponent +3). What we need to consider is how many bits we wish to devote to each of the components.

***How are real numbers stor*** ✓

The sign is easy: we still need only 1 bit. If we, again, decide to use only 16 bits, that leaves us with 15 bits to distribute between the exponent and the mantissa. *How much information can we store?* There is a trade-off: allowing more bits to the mantissa (or value), means that we can represent numbers more *precisely*. Giving more bits to the characteristic (exponent) means that we can have larger and smaller number, or we change the *magnitude* of the value.

For example, if we (arbitrarily) decide to assign 4-bits to the characteristic (exponent value), and the remaining 11 bits to the mantissa (numeric value), then (keeping in mind that the characteristic of the exponent can positive or non-positive), the possible range of numbers is:

Calculation 2.20.

Given an 11-bit mantissa: $I = 2^{11} = 2{,}048$ => the values 0 through 2,047 (inclusive)
Given a 4-bit exponent: $I = 2^4 = 16$ but since exponents take on pos. and neg. values:
Exponent Range $= -2^3$ to $+2^3$ -1 or -8 to +7

That means that the range of numbers we can represent is -.2047E-8 to +.2047E+7, or more simply -.000000002047 to +2,047,000. However, there is a caution here. The value 123.45 is clearly in the range of values given, BUT we can not represent it.


Under our present scheme we are only allowing 11-bits to the mantissa. Eleven bits will allow us to represent numbers up to 2047 to four decimal points of precision. The value

12345 (overlooking the decimal point) requires 5 decimals of precision and therefore we would 14-bits (since $2^{13} = 8192$ and $2^{14} = 16384$).

It does not make sense to talk about a *range* of real numbers. We need to be concerned about how ***precisely*** we can represent values. Consider the following values and how many bits are necessary to represent them as decimals:

Calculation 2.21.

**20.49 = 0.2049 E+2 ⇒ log(2049)/.30103 = 3.312/.30103 = 11.002 ⇒ 12-bits needed**

**-798.76 = -0.79876 E+3 ⇒ log(79876)/.30103 = 4.9024/.30101 = 16.29 ⇒ 20-bits needed**

**0.0876542 = 0.876542 E-1 ⇒ log(876542)/.30103 = 5.9408/.30101 = 19.74 ⇒ 20-bits needed**

**CAVEAT**

We are being slightly misleading here. Converting the decimal component of a real number to binary is somewhat different than converting an integer to binary. The basic concept (how many bits are needed) that we are addressing, however, is basically the same. For those of you wishing to see how the decimal component of real number is actually calculated, and how real numbers are actually stored, we include the process in Addendum 2.1.

Floating-point numbers are generally described in terms of ***level of precision*** (how precisely, or to how many decimal points, we can represent a number). With 11-bits devoted to the mantissa, we have only three decimal points of precision.

Def

***How are real nos. described?*** ✓

***Why three? Can't we represent, for example, the number 2,012, and isn't that four decimal points of precision ??***

Yes, but, as we saw above we can not represent the number 2,049 (or any four digit number larger than 2,047 for that matter). The level of precision stated implies that *all* numbers can be displayed at that level of precision[10].

***What level of precision should we have??***

That depends on what you want. However, this is really a zero-sum game: If we add more digits to the mantissa to increase the level of precision, we must reduce the number of bits we allocate to the characteristic, meaning we decrease the magnitude of the numbers we can represent.

***What does magnitude have to do with it??***

[10] Some texts and manuals would say, in this case, three *or* four decimals of precision.

Consider the numbers **4.5**, **450000**, and **0.000000000045**. If we were to put them in exponential format, they would be represented as **0.45E+1**, **0.45E+6**, **0.45E-10**. The precision needed (2 decimal points), and the number of bits required for the mantissa (6) are the same. However, we must allocate 2-bits to the characteristic to represent 0.45E+1, since $2^2 = 4 \Rightarrow$ the characteristic values –2 to +1. To represent 0.45E+6 we need 4-bits since $2^4 = 16 \Rightarrow$ the characteristic values –8 to +7. To represent the value 0.45E-10 we need 5-bits since $2^5 = 32 \Rightarrow$ the characteristic values –16 to +15. Remember, however: For every bit we add to the characteristic, we take one away from the mantissa.


If we stick to using 16-bits, and knowing that we must allocate 1-bit for the sign, the combinations using the remaining 15-bits are as follows (Table 2.7.):

Table 2.7.

| Characteristic Bits | Mantissa Bits | Precision (decimal points) | Magnitude (Exponent Ranges[11]) |
|---|---|---|---|
| 1 | 14 | $2^{14} = 16{,}384 \Rightarrow$ **4** | $2^1 = 2 \Rightarrow -2^0$ to $+2^0-1 \Rightarrow$ -1 to 0 |
| 2 | 13 | $2^{13} = 8{,}192 \Rightarrow$ **3** | $2^2 = 4 \Rightarrow -2^1$ to $+2^1-1 \Rightarrow$ -2 to 1 |
| 3 | 12 | $2^{12} = 4{,}048 \Rightarrow$ **3** | $2^3 = 8 \Rightarrow -2^2$ to $+2^2-1 \Rightarrow$ -4 to 3 |
| 4 | 11 | $2^{11} = 2{,}048 \Rightarrow$ **3** | $2^4 = 16 \Rightarrow -2^3$ to $+2^3-1 \Rightarrow$ -8 to 7 |
| 5 | 10 | $2^{11} = 2{,}048 \Rightarrow$ **3** | $2^5 = 32 \Rightarrow -2^4$ to $+2^4-1 \Rightarrow$ -16 to 15 |
| 6 | 9 | $2^9 = 512 \Rightarrow$ **2** | $2^6 = 64 \Rightarrow -2^5$ to $+2^5-1 \Rightarrow$ -32 to 31 |
| 7 | 8 | $2^8 = 256 \Rightarrow$ **2** | $2^7 = 128 \Rightarrow -2^6$ to $+2^6-1 \Rightarrow$ -64 to 63 |
| 8 | 7 | $2^7 = 128 \Rightarrow$ **1** | $2^8 = 256 \Rightarrow -2^7$ to $+2^7-1 \Rightarrow$ -128 to 127 |
| 9 | 6 | $2^6 = 64 \Rightarrow$ **1** | $2^9 = 512 \Rightarrow -2^8$ to $+2^8-1 \Rightarrow$ -256 to 255 |
| 10 | 5 | $2^5 = 32 \Rightarrow$ **1** | $2^{10} = 1{,}024 \Rightarrow -2^9$ to $+2^9-1 \Rightarrow$ -512 to 511 |
| 11 | 4 | $2^4 = 16 \Rightarrow$ **1** | $2^{11} = 2{,}048 \Rightarrow -2^{10}$ to $+2^{10}-1 \Rightarrow$ -1,024 to 1,023 |
| 12 | 3 | $2^3 = 8 \Rightarrow$ **1** | $2^{12} = 4{,}096 \Rightarrow -2^{11}$ to $+2^{11}-1 \Rightarrow$ -2,048 to 2,047 |
| 13 | 2 | $2^2 = 4 \Rightarrow$ **1** | $2^{13} = 8{,}192 \Rightarrow -2^{12}$ to $+2^{12}-1 \Rightarrow$ -4,096 to 4,095 |
| 14 | 1 | $2^1 = 2 \Rightarrow$ **1** | $2^{14} = 16{,}384 \Rightarrow -2^{13}$ to $+2^{13}-1 \Rightarrow$ -8,192 to 8,191 |

Looking at the table, it becomes obvious that none of the combinations are really adequate. The mantissa is too small, and, in most cases, there is really no need to have an exponent as large as $2^{13}$ (the number 0.4E+8192 is 4 followed by 8,191 zeros).

Just as we saw that 8-bits was insufficient to represent integers, we need to increase the number of bits used to represent real numbers.


[11] This is not quite correct. Exponent values *Biased*, but we will not go into a discussion of how this works here.

What level of precision do you need, and what type of machine do you have? We have already seen that different machines store the same numeric value types differently. The most extreme example is the difference between the manner in which the PC (especially the older machines) store integers versus how a supercomputer stores integers.

In the PC, floating-point numbers are generally stored using 32-bits; double the number for an **int** or **short;** the same as a **long** integer. *How are the bits distributed?* Remember, we still need 1-bit for the sign, leaving us 31-bits to allocate between the mantissa and the characteristic. Let's look at some different combinations, disregarding the allocation schemes we know are inadequate (Table 2.8.).

Table 2.8.

| Characteristic Bits | Mantissa Bits | Precision (decimal points) | Magnitude (Exponent Ranges) |
|---|---|---|---|
| 3 | 28 | $2^{28}$ = 268,435,456 ⇒ **8** | $2^3$ = 8 ⇒ $-2^2$ to $+2^2-1$ ⇒ -4 to 3 |
| 4 | 27 | $2^{27}$ = 134,217,728 ⇒ **8** | $2^4$ = 16 ⇒ $-2^3$ to $+2^3-1$ ⇒ -8 to 7 |
| 5 | 26 | $2^{26}$ = 67,108,864 ⇒ **7** | $2^5$ = 32 ⇒ $-2^4$ to $+2^4-1$ ⇒ -16 to 15 |
| 6 | 25 | $2^{25}$ = 33,554,432 ⇒ **7** | $2^6$ = 64 ⇒ $-2^5$ to $+2^5-1$ ⇒ -32 to 31 |
| 7 | 24 | $2^{24}$ = 16,777,216 ⇒ **7** | $2^7$ = 128 ⇒ $-2^6$ to $+2^6-1$ ⇒ -64 to 63 |
| 8 | 23 | $2^{23}$ = 8,388,608 ⇒ **6** | $2^8$ = 256 ⇒ $-2^7$ to $+2^7-1$ ⇒ -128 to 127 |
| 9 | 22 | $2^{22}$ = 4,194,304 ⇒ **6** | $2^9$ = 512 ⇒ $-2^8$ to $+2^8-1$ ⇒ -256 to 255 |
| 10 | 21 | $2^{21}$ = 2,097,152 ⇒ **6** | $2^{10}$ = 1,024 ⇒ $-2^9$ to $+2^9-1$ ⇒ -512 to 511 |
| 11 | 20 | $2^{20}$ = 1,048,576 ⇒ **6** | $2^{11}$ = 2,048 ⇒ $-2^{10}$ to $+2^{10}-1$ ⇒ -1,024 to 1,023 |
| 12 | 19 | $2^{19}$ = 524,288⇒ **5** | $2^{12}$ = 4,096 ⇒ $-2^{11}$ to $+2^{11}-1$ ⇒ -2,048 to 2,047 |
| 13 | 18 | $2^{18}$ = 262,144 ⇒ **5** | $2^{13}$ = 8,192 ⇒ $-2^{12}$ to $+2^{12}-1$ ⇒ -4,096 to 4,095 |
| 14 | 17 | $2^{17}$ = 132,768 ⇒ **5** | $2^{14}$ = 16,384⇒ $-2^{13}$ to $+2^{13}-1$ ⇒ -8,192 to 8,191 |


***So which one do we choose??***

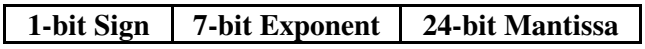
Table 2.9.

| Level of Precision | Mantissa Bits | Characteristic Bits |
|---|---|---|
| 5 | 17 | 14 |
| 6 | 20 | 11 |
| 7 | 24 | 7 |
| 8 | 27 | 4 |

Still not the easiest choice. However, looking at the table, since we know that we are concerned with level of precision, we know, for each level of precision, what the optimum number of bits is. In other words, if we need 6-decimals of precision, we should allocate 20-bits to the mantissa and 11-bits to the characteristic. Allocating 21-bits to the mantissa and 10-bits to the characteristic would still leave us with 6-decimals of precision, but would decrease the magnitude of the number by 50% (in this case, reducing the exponent value from -1,024 to 1,023 to –512 to –511).

The question, then, becomes: *Which of the 4 combinations is most appealing?* While there is no universal agreement, and various software compilers still interpret the combinations differently, the generally preferred combination is to assign 24-bits to the mantissa (yielding a 7 decimals of precision) and 7 bits to the characteristic (allowing for a magnitude from E-64 to E+63), shown in Figure 2.20. While this not a universal layout, it is quite typical.

Figure 2.20.




Although, as we already know, it doesn't make sense to talk about the range of real numbers, but for illustration purposes, note that we could represent such numbers as:

0.1E-64 = 0.00000000000000000000000000000000000000000000000000000000000000000000000000000001

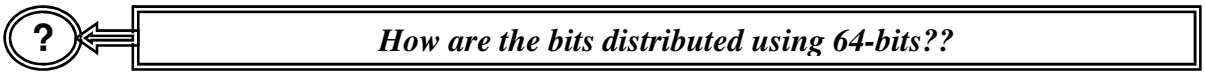
through

+.16777215E+63 = +167,772,150,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000

and ***all*** numbers in between, BUT **only** to 7 decimals of precision.


## Double-Precision Real Numbers

The data type **float** that we have been discussing is also frequently referred to as a *single precision* real number. As with integers (e.g., **int** vs. **long**), most application packages allow you to increase the number of bits used to store a number. C++ also allows for a data type called **double** (sometimes called a *double precision* real number), which, as could have been predicted, uses 64-bits.




That becomes tricky. The number is stored using 64-bits, but how many bits are allocated to each of the components is a function of the system and the software package being used. C++ allows for bit-manipulation, meaning you can address, and manipulate, each of the individual bits (many packages do not allow bit-manipulation, or *bit-fiddling*, as it is referred to). Some systems use all the 32 additional bits for the mantissa (=> $2^{56}$ = 72,057,594,037,927,900, implying the values from 0 to 72,057,594,037,927,900, or *all* numbers up to 16 decimals of precision), others allocate less. ANSI (The American National Standards Institute) C++ requires that the **double** type represent at least 10 digits of precision (meaning that a minimum of 34-bits (total) must be assigned to the mantissa since $2^{34}$ = 17,179,869,184 (10 digits of precision) and $2^{33}$ = 8,589,934,592 (9 digits of precision)).

If even larger numbers are needed, C++ also allows for a **long double** data type (sometimes referred to as a *quad* word), which uses 128-bits. However, ANSI C++ only requires that **long double** floating point data types provide 10 digits of precision (the same

as double). In this case, the numbers are of increased magnitude, but not necessarily of increased precision. If, for example, there are 128-bits, 1-bit is allocated for the sign, only 34-bits to the mantissa, and the remaining 93-bits to the exponent, the largest value that could be represented is

+.17179869184E93 =
17,179,869,184,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,00,000,000,000

Now, if you really want some computing power, some of the supercomputers, which have 128-bit word sizes and allow for quad (4-times) floating-point numbers, using 512-bits, then

••••

***What if I need large numbsers?*** ✓ Well, you get the idea.

## Numeric Data Types in C

Before we summarize the numeric data types available in C, let's not forget that we have also discussed the data type **char**. A character *is* a numeric data type also, but we generally associate it with the ASCII (or EBCDIC) coding scheme. From that perspective, we should keep in mind that.

**The Data Type Character**

1. All characters are stored as 8-bits as a numeric value, but according to a prescribed coding scheme (ASCII or EBCDIC).
2. Mathematical operations CAN be performed on characters, but the results are not what we might expect (i.e., '3' + '4' = 'g', or 51 + 52 = 103, which is the ASCII character 'g').

Numeric values are stored using various numbers of bits (always in multiples of 8, however), and, as we have seen, can be manipulated to perform numeric operations. We have already discussed most the numeric data types. The ones that were not previously mentioned can be readily understood from their descriptions and an understanding of the data types that were discussed[12]. A summary of the different data types we have discussed so far is outlined in Table 2.10.


***Why there so many duplicate data types in C?***

[12] Remember, we are talking about C data types storage on the PC. The same data types are available on different computers also, but the storage requirements differ.

We have talking about how C++ uses variables/data types on the PC, but as we have previously noted, different computers (mainframes, minicomputers, and supercomputers) store things differently. Nonetheless, C++ is intended to be a transportable language. That means that we can use the same C++ code on a variety of machines, even though they might process it a little differently. For example, while a **short** and an **int** are the same data types on the PC, a short is 16-bits on a mainframe, while an int is stored on 32-bits (remember, we also noted that things will be changing on PCs as well).

Table 2.10.

| Type | Variable | Bits | Range | Precision | Comments |
|---|---|---|---|---|---|
| Character | **char** | 8 | | | Stored via Coding |
| | **unsigned** | | 0 to 255 | | Scheme |
| | **signed** | | -128 to 127 | | |
| Integer | **short** | 16 | | | Same as **signed short** |
| | **unsigned** | | 0 to 65,535 | | Same as **unsigned int** |
| | **signed** | | -32,768 to 32,767 | | Same as **signed int** |
| | **int**[13] | 16 | | | Same as **signed int** |
| | **unsigned** | | 0 to 65,535 | | Same as **unsigned short** |
| | **signed** | | -32,768 to 32,767 | | Same as **signed short** |
| | **long** | 32 | | | Same as **signed long** |
| | **unsigned** | | 0 to 4,294,967,295 | | |
| | **signed** | | -2,147,483,648 to 2,147,483,648 | | Same as **long** |
| Real | **float** | 32 | | 7 | |
| | **double** | 64 | | 10 (min) | |
| | **long double** | 128 | | 10 (min) | |

There is one other note to be made here. ***By default***, ALL numeric data types are signed (meaning they can take on positive as well as non-positive values). That means that an **int** is the same as a **signed int**. Also note there is no such thing as an **unsigned float** (or **double** or **long double**).


Don't forget, characters are really numeric values. Therefore, by default characters are signed. We will see more about characters in Chapter 5, when we discuss the abstract data type string.

***What data types are there in C+*** ✓

C/C++ may be relatively primitive languages, but they are also very powerful ones. At this point in time, some of our students frequently make the statement "COBOL is much better than C, because you don't have to know all of this stuff". That is not quite true, but there is some truth to it. In COBOL, you might declare a variable as '99' (an integer) without

[13] Remember our Caveat

having to know how it is truly stored, but you are declaring it nonetheless. On the other hand, there commands which can be issued in C++ and not in COBOL, and abstract data types created using C, which can't be constructed in COBOL. No pain, no gain.

Knowing about basic data types, how they are stored, and how they are manipulated has a number of important implications for programmers:

**Programming Implications**

1. Since the amount of space which each data type can vary (from 8-bits to 128-bits), the programmer should be careful when assigning data types. If, for example, one knows that the values assigned to a variable would never be outside the range 0 to 100, it certainly wouldn't make much sense to store them as long doubles (128-bits) when they could be stored as a numeric byte (8-bits). The 120-bit difference (15 bytes) could be considerable if, for example, each datum is stored in a 1,000 x 1,000 array (1,000,000 elements). There could be a savings of 15,000,000 bytes (15MB) if the data were stored in the appropriate format.

2. While we have frequently made reference to how fast the computer is, it is obvious that the more bits we use, and the more complex the data type (i.e., integers vs. floating-point), the longer the operations to be performed on them will take. This is especially true when going from unsigned integers to signed integers to floating-point numbers.

## Other Data Types (Sort-Of)

There is one additional data type which is available in C++ (but not in C): a **bool**. These are logical data types which are stored on 8-bits. They are equivalent to **Boolean** variables (in FORTRAN) and take on either the value 'true' or 'false'. They are extremely useful to evaluate logical operations or as programs flags. Since true and false are binary conditions, the logical assumption would be that they require only 1-bit (not byte) of storage. In fact, since the byte is the basic unit of data retrieval, they are stored on 8-bits (notice that if they were truly stored on 1-bit, they would be a unique data type, and not a data structure). In effect, they are numeric bytes which can take on the values $0_{10}$ = $00000000_2$ if 'false' or some other value (e.g., $1_{10} = 00000001_2$) if 'true'.

In some other computer languages, there are a few additional data types available. There really aren't (as the addendum '(sort-of)' in the section heading implies), but it might appear as if there are. Some of these might deserve some attention, although we will not describe them in great detail. In fact, these are really basic abstract concepts, or logical structures used for the organization of data. We include them here because often, they are treated as basic data types.

A **Byte** (in Pascal) is really the data type **unsigned char** in C. Consequently, it can take on the integer values from 0 through 255.

**Strings** are not really a unique data type either but rather a chain of character bytes. From our perspective, they are words, sentences, expressions, etc. In some programming languages each string takes up a fixed number of bytes in memory, in others the amount of storage taken-up depends on the number of characters in the string (1-byte per character). Each character in the string is stored in ASCII (or EBCDIC) format. We devote an entire chapter (Chapter 5) to a discussion of strings, primarily because of some of the manipulations which are performed on them.

***Are there other data types?*** ✓

## Summary

Some of you may think that material contained in this chapter is dull and boring. It is. But then again, so are arithmetic tables. However, just as we must suffer through memorizing arithmetic tables, we must put to memory many of the concepts covered in this chapter if we are to understand how the computer functions and deals with basic data.

The basic data types covered in this chapter are all there are. Period. In the section 'Other data types (sort-of)' we mentioned some structures which are frequently referred to as data types. They aren't. There are only three (four, maybe) data types: Characters, integers and floating-point numbers (the fourth might be unsigned numbers, whether numeric bytes and unsigned integers (short, int, or long), but these are merely variations on a theme). If you understand these basic data types, you understand (essentially) how the computer functions.

## Chapter Terminology: Be able to fully describe these terms

%
/
carry-over
char
characteristic of the exponent
characters
complementing
DIV
double
float
int
integer
long
long double
magnitude
mantissa
MOD
modulus arithmetic
numbers
octal
precision
quotient
real numbers
remainder
signed char
signed int
signed long
unsigned char
unsigned int
unsigned long

## Suggested Assignments

The assignments given below, along with the following review questions, are suggestions as to how you might further, or strengthen, your understanding of the material covered in this chapter. The review questions have definite answers; these assignments are intended as exercises.

1. Memorize the Octal to Binary Conversion Tables. For convenience sake, it is reproduced below:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Binary Number: | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Octal Number: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

2. Memorize the Hexadecimal to Binary Conversion Tables. For convenience sake, it is reproduced below:

| Hex: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Binary: | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

## Review Questions

1. Convert the following decimal numbers to binary (assume a 16 bit Integer: 1 bit for the sign, 15 bits for the value). If necessary, assume two's complement.

   a. 100
   b. 978
   c. 32,767
   d. -87
   e. -5,345
   f. -32,789

2. Convert the following Binary numbers (1 bit sign, 15 bit value) to Decimal, Octal and Hex

   a. 0,110010111011111
   b. 0,001101110111011
   c. 1,010100111001100 (One's Complement)
   d. 1,111111111111010 (Two's Complement)

3. Add the following numbers in binary:

   a. $78_{10} + 78_{10}$ using one's complement
   b. $-AB_{16} + 70_{8}$ using one's complement
   c. $439_{5} + 64_{7}$ using two's complement

4. You are building a database of addresses. The question was raised as to whether zip codes should be stored as numbers or strings. Explain what the differences are, and what the advantages/disadvantages are for each.

5. A new machine has been developed which will use a 20-bit register. Reals will be stored with a 1-bit sign and a 6-bit exponent. What will be the range for reals? What will be the range for Integers?

## Review Question Answers
**(NOTE: CHECKING THE ANSWERS BEFORE YOU HAVE TRIED TO ANSWER THE QUESTIONS DOESN'T HELP YOU AT ALL)**

**1. Convert the following decimal numbers to binary (assume a 16 bit Integer: 1 bit for the sign, 15 bits for the value). If necessary, assume two's complement.**

1.a.

| | | |
|---|---|---|
| 100 DIV 2 = 50 | 100 MOD 2 = 0 | ↑ $100_{10} = 1100100_2$ |
| 50 DIV 2 = 25 | 50 MOD 2 = 0 | |
| 25 DIV 2 = 12 | 25 MOD 2 = 1 | Chk: $2^6 + 2^5 + 2^2 = 64 + 32 + 4$ |
| 12 DIV 2 = 6 | 12 MOD 2 = 0 | $= 100$ |
| 6 DIV 2 = 3 | 6 MOD 2 = 0 | |
| 3 DIV 2 = 1 | 3 MOD 2 = 1 | Ans: 0,000000001100100 |
| 1 DIV 2 = 0 | 1 MOD 2 = 1 | |

b.

| | | |
|---|---|---|
| 978 DIV 2 = 489 | 978 MOD 2 = 0 | ↑ $978_{10} = 1111010010_2$ |
| 489 DIV 2 = 244 | 489 MOD 2 = 1 | |
| 244 DIV 2 = 122 | 244 MOD 2 = 0 | Chk: $2^9 + 2^8 + 2^7 + 2^6 + 2^4 + 2^1$ |
| 122 DIV 2 = 61 | 122 MOD 2 = 0 | $= 512+256+128+64+16+2$ |
| 61 DIV 2 = 30 | 61 MOD 2 = 1 | $= 978$ |
| 30 DIV 2 = 15 | 30 MOD 2 = 0 | |
| 15 DIV 2 = 7 | 15 MOD 2 = 1 | |
| 7 DIV 2 = 3 | 7 MOD 2 = 1 | Ans: 0,000001111010010 |
| 3 DIV 2 = 1 | 3 MOD 2 = 1 | |
| 1 DIV 2 = 0 | 1 MOD 2 = 1 | |

c.

| | | |
|---|---|---|
| 32,767 DIV 2 = 16,383 | 32,767 MOD 2 = 1 | $32{,}767_{10} = 111111111111111_2$ |
| 16,383 DIV 2 = 8,191 | 16,383 MOD 2 = 1 | |
| 8,191 DIV 2 = 4,095 | 8,191 MOD 2 = 1 | Chk: $2^{14}+2^{13}+2^{12}+2^{11}+2^{10}+2^9+2^8$ |
| 4,095 DIV 2 = 2,047 | 4,095 MOD 2 = 1 | $+2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0$ |
| 2,047 DIV 2 = 1,023 | 2,047 MOD 2 = 1 | = 16,384 + 8,192 + 4,096 + |
| 1,023 DIV 2 = 511 | 1,023 MOD 2 = 1 | 2,048 + 1,024 + 512 + 256 + |
| 511 DIV 2 = 255 | 511 MOD 2 = 1 | 128 + 64 + 32 + 16 + 8 + 4 + |
| 255 DIV 2 = 127 | 255 MOD 2 = 1 | 2 + 1 |
| 127 DIV 2 = 63 | 127 MOD 2 = 1 | = 32,767 |
| 63 DIV 2 = 31 | 63 MOD 2 = 1 | |
| 31 DIV 2 = 15 | 31 MOD 2 = 1 | Which we could have predicted |
| 15 DIV 2 = 7 | 15 MOD 2 = 1 | |
| 7 DIV 2 = 3 | 7 MOD 2 = 1 | |
| 3 DIV 2 = 1 | 3 MOD 2 = 1 | Ans: 0,111111111111111 |
| 1 DIV 2 = 0 | 1 MOD 2 = 1 | |

d.

| | | |
|---|---|---|
| 87 DIV 2 = 43 | 87 MOD 2 = 1 | $87_{10} = 1010111_2$ |
| 43 DIV 2 = 21 | 43 MOD 2 = 1 | |
| 21 DIV 2 = 10 | 21 MOD 2 = 1 | Chk: $2^6 + 2^4 + 2^2 + 2^1 + 2^0$ |
| 10 DIV 2 = 5 | 10 MOD 2 = 0 | = 64 + 16 + 4 + 2 + 1 |
| 5 DIV 2 = 2 | 5 MOD 2 = 1 | = 87 |
| 2 DIV 2 = 1 | 2 MOD 2 = 0 | |
| 1 DIV 2 = 0 | 1 MOD 2 = 1 | |

BUT we need to complement:

0,000000001010111 ⇒ 1,111111110101000 (One's Complement)

\+ 1

1,111111110101001 (Two's Complement) → Ans.

e.

| | | |
|---|---|---|
| 5,344 DIV 2 = 2,672 | 5,345 MOD 2 = 0 | $5{,}345_{10} = 1010011100000_2$ |
| 2,672 DIV 2 = 1,336 | 2,672 MOD 2 = 0 | |
| 1,336 DIV 2 = 668 | 1,336 MOD 2 = 0 | Chk: $2^{12}+2^{10}+2^7+2^6+2^5 2^0$ |
| 668 DIV 2 = 334 | 668 MOD 2 = 0 | = 4,096 + 1,024 + 128 + 64 + 32 |
| 334 DIV 2 = 167 | 334 MOD 2 = 0 | = 5,344 |
| 167 DIV 2 = 83 | 167 MOD 2 = 1 | |
| 83 DIV 2 = 41 | 83 MOD 2 = 1 | |
| 41 DIV 2 = 20 | 41 MOD 2 = 1 | |
| 20 DIV 2 = 10 | 20 MOD 2 = 0 | BUT (again) we need to |
| 10 DIV 2 = 5 | 10 MOD 2 = 0 | complement |
| 5 DIV 2 = 2 | 5 MOD 2 = 1 | |
| 2 DIV 2 = 1 | 2 MOD 2 = 0 | |
| 1 DIV 2 = 0 | 1 MOD 2 = 1 | |

$$\begin{array}{rl} & \quad 11111 \\ 0{,}001010011100000 \Rightarrow & 1{,}110101100011111 \text{ (One's Complement)} \\ + & \qquad\qquad\quad 1 \\ \hline & 1{,}111010110100000 \text{ (Two's Complement)} \end{array} \rightarrow \underline{\textbf{Ans}}.$$

f. We Can't because $\lceil \log(32{,}789)/\log(3) \rceil = \lceil 4.516/0.30103 \rceil = \lceil 15.002 \rceil = \underline{\textbf{16}}$ and we only have 15-bits on which to store the integer value. However, we could store the value as a **signed long** (if it were positive, we could also have stored it as an **unsigned int**) using 32-bits (1-bit for the sign and 31-bits for the value):

| | | |
|---|---|---|
| 32,789 DIV 2 = 16,394 | 32,789 MOD 2 = 1 | $32{,}767_{10} = 100000000010101_2$ |
| 16,394 DIV 2 = 8,197 | 16,394 MOD 2 = 0 | |
| 8,197 DIV 2 = 4,098 | 8,197 MOD 2 = 1 | Chk: $2^{15} + 2^4 + 2^2 + 2^0$ |
| 4,098 DIV 2 = 2,049 | 4,098 MOD 2 = 0 | $= 32{,}768 + 16 + 4 + 1$ |
| 2,049 DIV 2 = 1,024 | 2,049 MOD 2 = 1 | $= 32{,}789$ |
| 1,024 DIV 2 = 512 | 1,024 MOD 2 = 0 | |
| 512 DIV 2 = 256 | 512 MOD 2 = 0 | |
| 256 DIV 2 = 128 | 256 MOD 2 = 0 | |
| 128DIV 2 = 64 | 128 MOD 2 = 0 | ***BUT*** (of course) we still must |
| 64 DIV 2 = 32 | 64 MOD 2 = 0 | complement |
| 32 DIV 2 = 16 | 32 MOD 2 = 0 | |
| 16 DIV 2 = 8 | 16 MOD 2 = 0 | |
| 8 DIV 2 = 4 | 8 MOD 2 = 0 | |
| 4 DIV 2 = 2 | 4 MOD 2 = 0 | |
| 2 DIV 2 = 1 | 2 MOD 2 = 0 | |
| 1 DIV 2 = 0 | 1 MOD 2 = 1 | |

$$\begin{array}{rl} 0,\ 00000000000000010000000010101 \Rightarrow & 1{,}11111111111111101111111101010 \quad \text{(1's Complement)} \\ + & \qquad\qquad\qquad\qquad\qquad\qquad 1 \\ \hline & 1{,}11111111111111101111111101011 \quad \text{(2's Complement)} \end{array}$$

**2. Convert the following Binary numbers (1 bit sign, 15 bit value) to Decimal, Octal and Hex**

2.a. Decimal:

$$\begin{aligned} 0{,}110010111011111_2 &= 2^{14} + 2^{13} + 2^{10} + 2^8 + 2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 \\ &= 16{,}384 + 8{,}192 + 1{,}024 + 256 + 128 + 64 + 16 + 8 + 4 + 2 + 1 \\ &= +\ 26{,}079_{10} \end{aligned}$$

Octal:

**+ 6 2 7 3 7**

0, 110 | 010 | 111 | 011 | $111_2$

Chk: $6 * 8^4 + 2 * 8^3 + 7 * 8^2 + 3 * 8^1 + 7 * 8^0$
$= 6 * 4,096 + 2 * 512 + 7 * 64 + 3 * 8 + 7 * 1$
$= 24,576 + 1,024 + 448 + 24 + 7$
$= 26,079_{10}$

Hexadecimal:

**+ 6 5 D F**

0, 110 | 0101 | 1101 | $1111_2$

Chk: $6 * 16^3 + 5 * 16^2 + D = 13 * 16^1 + F = 15 * 16^0$
$= 6 * 4,096 + 5 * 256 + 13 * 16 + 15 * 1$
$= 24,576 + 1,280 + 208 + 15$
$= 26,079_{10}$

2.b. Decimal:

$0,001101110111011 = + 2^{12} + 2^{11} + 2^9 + 2^8 + 2^7 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0$
$= + 4,096 + 2,048 + 512 + 256 + 128 + 32 + 16 + 8 + 2 + 1$
$= + 7,099_{10}$

Octal:

**+ 1 5 6 7 3**

0, 001 | 101 | 110 | 111 | 011

Chk: $1 * 8^4 + 5 * 8^3 + 6 * 8^2 + 7 * 8^1 + 3 * 8^0$
$= 1 * 4,096 + 5 * 512 + 6 * 64 + 7 * 8 + 3 * 1$
$= 4,096 + 2,560 + 384 + 56 + 3$
$= + 7,099_{10}$

Hexadecimal:

**+ 1 B B B**

0, 001 | 1011 | 1011 | 1011

Chk: $= 1 * 16^3 + B=11 * 16^2 + D=13 * 16^1 + B=11 * 16^0$
$= 1 * 4,096 + 11 * 256 + 11 * 16 + 11 * 1$
$= 4,096 + 2,816 + 176 + 11$
$= +7,099$

2.c. Decimal:

Since negative, we must first complement: $1,010100111001100 \rightarrow 0,101011000110011$

$101011000110011 = 2^{14} + 2^{12} + 2^{10} + 2^9 + 2^5 + 2^4 + 2^1 + 2^0$
$= 16,384 + 4,096 + 1,024 + 512 + 32 + 16 + 2 + 1$
$= -22,067$

Octal:

**5 3 0 6 3**

101 | 011 | 000 | 110 | 011

Chk: $= 5 * 8^4 + 3 * 8^3 + 6 * 8^1 + 3$
$= 5 * 4,096 + 3 * 512 + 6 * 8 + 3$
$= 20,480 + 1,536 + 48 + 3$
$= -22,067$

Hexadecimal:

| **5** | **6** | **3** | **3** |
|---|---|---|---|
| 101 | 0110 | 0011 | 0011 |

Chk: $= 1 * 16^3 + 6 * 16^2 + 3 * 16^1 + 3$
$= 5 * 4{,}096 + 6 * 256 + 3 * 16 + 3$
$= 20{,}480 + 1{,}536 + 48 + 3$
$= -22{,}067$

2.d. Decimal:

Since the binary number: 1,111111111111010 has the value '1' in the left-most position, the number is negative and we must complement the right 15-bits AND add '1' to the complemented result:

$$\begin{array}{l} 000000000000101 \\ +\quad\quad\quad\quad\quad 1 \\ \hline 000000000000110_2 = -6_{10} = -6_8 = -6_{16} \end{array}$$

Are we sure?? As a check, let's store the value $-6_{10}$.

$6 / 2 = 3$ $\quad 6 \% 2 = 0$
$3 / 2 = 1$ $\quad 3 \% 2 = 1$
$1 / 2 = 0$ $\quad 1 \% 2 = 1$ ↑ $= 110_2 = 0000000000000110_2$ on 16-bits

Since the value is negative (and stored as a two's complement), we must complement and add 1 to the complement:

| | |
|---|---|
| 0000000000000110 | The Value +6 |
| ↓ | |
| 1111111111111001 | One's Complement |
| + 1 | Add 1 |
| 1111111111111010 | Two's Complement |

**3. Add the following numbers in binary:**

3.a. $78_{10} + 78_{10}$ using one's complement:

| | | |
|---|---|---|
| 78 DIV 2 = 39 | 78 MOD 2 = 0 | ↑ |
| 39 DIV 2 = 19 | 39 MOD 2 = 1 | |
| 19 DIV 2 = 9 | 19 MOD 2 = 1 | |
| 9 DIV 2 = 4 | 9 MOD 2 = 1 | |
| 4 DIV 2 = 2 | 4 MOD 2 = 0 | |
| 2 DIV 2 = 1 | 2 MOD 2 = 0 | |
| 1 DIV 1 = 0 | 1 MOD 2 = 1 | |

$78_{10} = 1001110_2$

Chk: $= 2^6 + 2^3 + 2^2 + 2^1$
$= 64 + 8 + 4 + 2$
$= 78$

Carry-over: 1 111 Chk: $= 2^7 + 2^4 + 2^3 + 2^2$

0,000000001001110 $= 128 + 16 + 8 + 4$

+ 0,000000001001110 $= 156 = 78 + 78$

0,000000010011100

3.b. $-AB_{16} + 70_8$ using one's complement

$AB_{16}$ = **A** **B**

1010 1011 = 0,000000010101011

Complementing: 1,111111101010100

$70_8$ = **7** **0**

111 000 = 0,000000000111000

Carry-Over: 111

Adding the two we get: 1,111111101010100

+ 0,000000000111000

1,111111110001100

But Since the result is negative:

111111110001100 Complemented = $000000001110011_2 = 115_{10}$

Which we know to be true since $AB_{16} = 171_{10}$ and $70_8 = 56_{10}$ and

$-171_{10} + 56_{10} = 115_{10}$

3.c. $439_5 + 64_7$ using two's complement:

$439_5 = 4 * 5^2 + 3 * 5^1 + 9 * 5^0$

$= 4 * 25 + 3 * 5 + 9 * 1$

$= 100 + 15 + 9$

$= 124_{10}$

$64_7 = 6 * 7^1 + 4 * 7^0$

$= 6 * 7 + 4 * 1$

$= 42 + 4$

$= 46_{10}$

1111100 101110

| | | | |
|---|---|---|---|
| 124 DIV 2 = 62 | 124 MOD 2 = 0 | 46 DIV 2 = 23 | 46 MOD 2 = 0 |
| 62 DIV 2 = 31 | 62 MOD 2 = 0 | 23 DIV 2 = 11 | 23 MOD 2 = 1 |
| 31 DIV 2 = 15 | 31 MOD 2 = 1 | 11 DIV 2 = 5 | 11 MOD 2 = 1 |
| 15 DIV 2 = 7 | 15 MOD 2 = 1 | 5 DIV 2 = 2 | 5 MOD 2 = 1 |
| 7 DIV 2 = 3 | 7 MOD 2 = 1 | 2 DIV 2 = 1 | 2 MOD 2 = 0 |
| 3 DIV 2 = 1 | 3 MOD 2 = 1 | 1 DIV 2 = 0 | 1 MOD 2 = 1 |
| 1 DIV 2 = 0 | 1 MOD 2 = 1 | | |

Adding: 1111100

+ 0101110

$10101010_2 = 170_{10}$

Since we are not dealing with negative numbers, we do not need to complement

**4. You are building a database of addresses. The question was raised as to whether zip codes should be stored as numbers or strings. Explain what the differences are, and what the advantages/disadvantages are for each.**

If Stored as a number (assuming, e.g., the zip code 12345):
It can not be stored as a numeric byte (Max value = 255)
It can not be stored as a signed integer
(Max positive value = 32767; The zip code 32768 not stored)
It can not be stored as an unsigned integer
(Max value = 65536; The zip code 65537 not stored)
It can be stored as a signed long (Max pos. value = 2147483647)
or Unsigned long (Max Value = 4294967296)

Therefore it will take 32-bits to store it as an integer and can perform mathematical operations on it (do we ever really need to?). Operations are more complex than with ASCII characters.

If Stored as an ASCII Character:

It will require 5-bytes (1-byte more than if stored as an integer). On the other hand, operations (especially printing) will be much easier. Assuming there are 99999 Zip codes (there are not), that means that it would require 399,996 bytes (3,199,968 bits) to store them all as an integer. It would require 499,995 bytes (3,999,960 bits) to store them all as ASCII characters (20% more storage).

**5. A new machine has been developed which will use a 20 bit register. Reals will be stored with a 1-bit sign and a 6-bit exponent. What will be the range/precision level for reals? What will be the range for Integers?**

Integers: Given 20-bits: 1-bit for the sign, 19-bits for the value = $-2^{19}$ to + $2^{19}$ - 1
=> -524,288 to +524,287

Reals: Given 1-bit sign, 6-bit exponent, 13-bit values:

6-bit sign: Exponent Value = $-2^5$ to $+2^5-1$
13-bit value: Range = $2^{13}$ = 0 to 8,192
⇨ 3 decimals of precision

## C/C++ Programming Assignments

1. This program is a slight variation on the programming assignment given in Chapter 1. The difference is that we will print out the characters after we have stored them as different basic data types. To print them out, however, we will need to cast them.

```
#include <stdio.h>                          // Include the Standard Input-Ouput (IO) Headers File
int main(void)                              // MAIN is a function name which returns an integer
{                                           // This is similar to a BEGIN statement
char ch;                                    // ch is the variable where we will store an ASCII character
int inumber;
long lnumber;
float fnumber;                              // BUT it is really a signed numeric byte (on 8-bits)

ch = 'T';                                   // Assign the ASCII Character T to the variable ch
printf("The values of character %c are %d decimal, %o octal and %x Hexadecimal\n",ch,ch,ch,ch);
// The output will appear as:   The values of character T are 84 decimal, 124 octal and 54 Hexadecimal

ch = 38;                                    // Assign ASCII 38 (decimal) to the variable
printf("The values of character %c are %d decimal, %o octal and %x Hexadecimal\n",ch,ch,ch,ch);
// The output will appear as:     The values of character & are 38 decimal, 56 octal and 26 Hexadecimal

ch = 0135;                                  // putting 0 (zero) in front assigns the Octal value
printf("The values of character %c are %d decimal, %o octal and %x Hexadecimal\n",ch,ch,ch,ch);
// The output will appear as:     The values of character ] are 93 decimal, 135 octal and 5d Hexadecimal

ch = 0X6b;                                  // putting 0X (Zero X) in front assigns the hexadecimal val
printf("The values of character %c are %d decimal, %o octal and %x Hexadecimal\n",ch,ch,ch,ch);
// The output will appear as:      The values of character k are 107 decimal, 153 octal and 6b Hexadecimal

ch  = '3';                                  // Assign the Character 3
ch1 = 52;                                   // Assign the Character 4
ch = ch + ch1;                              // Add the two together - What is the outcome?
printf("The outcome is %c, decimal %d\n",ch,ch1);

ch = 50 + 7;
printf("The outcome is %c, decimal %d\n",ch,ch);     // output: The outcome is 9, decimal 57

ch = 211;
printf("The outcome is %c, decimal %d\n",ch,ch);     // output: The outcome is ╙, decimal –45: See 2.a. Below

ch2 = 211;
printf("The outcome is %c, decimal %d\n",ch);        // output: The outcome is ╙, decimal 211: See 2.b. Below

return(0);                                  // Return a 0 Value to the function
}                                           //  End of function Main
```

2.a. In the above program (refer to the comment made). Why does *ch1* print out as the character ╚ ??

b. In the above program (refer to the comment made). Why does *ch2* print out as the number 211 when in the line above ch1 prints out as the number -45 ??

3. Modifying the above program, print out your name BUT using the following pattern:

letters: 1, 5, 9, 13, …. Enter the input as characters
letters: 2, 6, 10, 14, …. Enter the input as integers
letters: 3, 7, 11, 15, …. Enter the input as octal numbers
letters: 4, 8, 12, 16, …. Enter the input as hexadecimal numbers

For example, for my name (Peeter Kirs), I would enter (and print) as:

```
ch = 'P';
printf ("%c");
ch = 101;
printf ("%c");
ch = 0146;
printf ("%c");
ch = 0X74;
printf ("%c");
ch = 'e';
printf ("%c");
ch = 114;
printf ("%c");
ch = 040;            // NOTE: This is the space which separates first & last name
printf ("%c");
ch = 0X4B;
printf ("%c");
ch = 'i';
printf ("%c");
ch = 0163;
printf ("%c");
ch = 0X73;
printf ("%c");
```

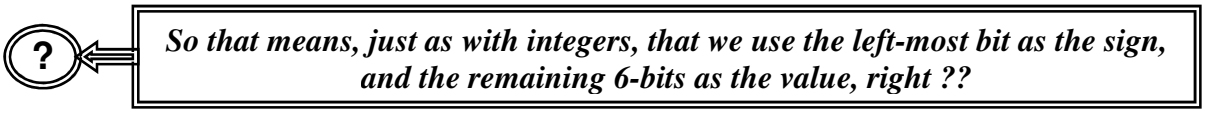
# Addendum 2.1: Storing Floating-Point Numbers

We already know the basic layout for the data type **float** (from figure 2.20):

| 1-bit Sign | 7-bit Exponent | 24-bit Mantissa |
|---|---|---|

We also know that (generally), the left-most bit is the sign, and can take-on either '0' or '1' values.

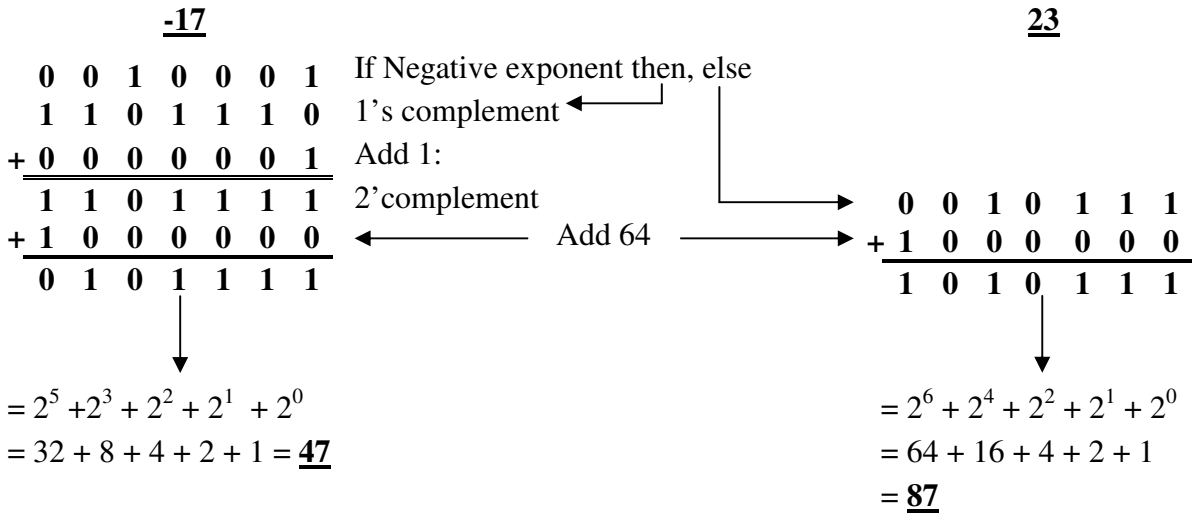
The **characteristic of the exponent** is not that difficult to understand either. Since we have 7-bits, we have a total of $2^7 = 128$ combinations, but since we know that the exponent can be either negative or non-negative, we really only have ½ that number.




*Not quite*. The characteristic of the exponent is stored as a ***biased*** exponent. That means that rather than storing the sign and the value separately, we add (bias) a constant term to the true value. In our case, we would add the value 64 (which is ½ of 128) to the true value. The exponent value $-17_{10}$ (= $10001_2$, or $0010001_2$ on 7-bits) would actually be stored as the value $-17 + 64 = 47_{10}$ (= $101111_2$, or $0101111_2$ on 7-bits); the exponent value $23_{10}$ (=$10111_2$ or $0010111_2$ on 7-bits) would actually be stored as the value $23 + 64 = 87_{10}$ (=$1010111_2$).


There are some technical reasons, which we need not go into, but of course, it does circumvent the step of having to store the sign and the value separately.

To convert the decimal exponents to binary:



There is one additional consideration: the range of exponent values is actually $-(2^6 - 1)$ through $+(2^6 - 1)$, or –63 through +63. The binary representation $0000000_2$ (the decimal value 0) is reserved for other uses (We need not go into that here).

To convert a binary exponent value back to decimal, of course, we have to first subtract the value $64_{10}$ (= $1000000_2$), which we already know is best accomplished by adding the two's complemented value of $64_{10}$ (= $0111111_2 + 1 = 1000000_2$) to the characteristic. For example, if we find the bit sequences used above:

| **Decimal 47:** | **0** | **1** | **0** | **1** | **1** | **1** | **1** |
|---|---|---|---|---|---|---|---|
| **Add –64:** | **1** | **0** | **0** | **0** | **0** | **0** | **0** |
| | **1** | **1** | **0** | **1** | **1** | **1** | **1** |

If negative, then (2's) complement and evaluate

**-** ($\mathbf{0010000}_2 + 1_2$)= - ($10001_2$)= - (24 + 1) = - (16 + 1) = -**<u>17</u>**$_{10}$

| **Decimal 87:** | **1** | **0** | **1** | **0** | **1** | **1** | **1** |
|---|---|---|---|---|---|---|---|
| **Add –64:** | **1** | **0** | **0** | **0** | **0** | **0** | **0** |
| | **0** | **0** | **1** | **0** | **1** | **1** | **1** |

$= 2^4 + 2^2 + 2^1 + 2^0 = 16 + 4 + 2 + 1$
$=$ **<u>23</u>**

One final quick note on characteristics: the range of exponent values is actually $-(2^6 - 1)$ through $+(2^6 - 1)$, or –63 through +63. The binary representation $0000000_2$ (the decimal value 0) is reserved for other uses.

Converting the mantissa to binary requires a somewhat different algorithm than we used to convert integers to binary, but it still has to do with exponent position. For example, the integer 456 would have the exponent positions shown at the left (in other words, $456 = 4*10^2 + 5*10^1 + 6*10^0$). If we were to consider the real number 456.789, however, the exponent positions would appear as the do on the right (in other words, $456.789 = 4*10^2 + 5*10^1 + 6*10^0 + 7*10^{-1} + 8*10^{-2} + 9*10^{-3}$).

| **Exponent Position:** | **2** | **1** | **0** |
|---|---|---|---|
| **Digit:** | 4 | 5 | 6 |

| **Exponent Position:** | **2** | **1** | **0** | | **-1** | **-2** | **-3** |
|---|---|---|---|---|---|---|---|
| **Digit:** | 4 | 5 | 6 | . | 7 | 8 | 9 |