Unsigned and Signed Integers

An integer is a number with no fractional part; it can be positive, negative or zero. In ordinary usage, one uses a minus sign to designate a negative integer. However, a computer can only store information in bits, which can only have the values zero or one. We might expect, therefore, that the storage of negative integers in a computer might require some special technique. It is for that reason that we began this section with a discussion of unsigned integers.

As you might imagine, an **unsigned integer** is either positive or zero. Given our discussion in the previous sections about binary numbers, it might seem that there is little more to say about unsigned integers. In fact, there is essentially only one thing, and that is one of the most important things that you will learn in this text. Consider a single digit decimal number: in a single decimal digit, you can write a number between 0 and 9. In two decimal digits, you can write a number between 0 and 99, and so on. Since nine is equivalent to $10^1 - 1$, 99 is equivalent to $10^2 - 1$, etc., in n decimal digits, you can write a number between 0 and $10^n - 1$. Analogously, in the binary number system,

> **an unsigned integer containing n bits can have a value between 0 and $2^n - 1$**
> **(which is 2^n different values**).

This fact is one of the most important and useful things to know about computers. When a computer program is written, the programmer, either explicitly or implicitly, must decide how many bits are used to store any given quantity. Once the decision is made to use n bits to store it, the program has an inherent limitation: that quantity can only have a value between 0 and $2^n - 1$. You will meet these limitations in one form or another in every piece of hardware and software that you will learn about during your career:

- the BIOS (Basic Input Output Software) in older PCs uses 10 bits to store the cylinder number on the hard drive where your operating system begins; therefore those PCs cannot boot an operating system from a cylinder greater than $2^{10} - 1$, or 1023.
- a FAT12 file system (used on Windows diskettes), which allocates file space in units called "clusters", uses 12 bits to store cluster numbers; therefore there can be no more than $2^{12} - 1$ or 4,095 clusters in such a file system.
- a UNIX system keeps track of the processes (programs) it runs using a PID (Process IDentifier); for typical memory sizes, the PID is 16 bits long and so after $2^{16} - 1$ or 65,535 processes, the PIDs must start over at the lowest number not currently in use.

These are just a few examples of this basic principle that you will meet in your future studies.

Most modern computers store memory in units of 8 bits, called a "**byte**" (also called an "**octet**"). Arithmetic in such computers can be done in bytes, but is more often done in larger units called "**(short) integers**" (16 bits), "**long integers**" (32 bits) or "**double integers**" (64 bits). Short integers can be used to store numbers between 0 and $2^{16} - 1$, or 65,535. Long integers can be used to store numbers between 0 and $2^{32} - 1$, or 4,294,967,295. and double integers can be used to store numbers between 0 and $2^{64} - 1$, or

18,446,744,073,709,551,615. (Check these!)

When a computer performs an unsigned integer arithmetic operation, there are three possible problems which can occur:

1. if the result is too large to fit into the number of bits assigned to it, an "**overflow**" is said to have occurred. For example if the result of an operation using 16 bit integers is larger than 65,535, an overflow results.

1. in the division of two integers, if the result is not itself an integer, a "**truncation**" is said to have occurred: 10 divided by 3 is truncated to 3, and the extra 1/3 is lost. This is not a problem, of course, if the programmer's intention was to ignore the remainder!

1. any division by zero is an error, since division by zero is not possible in the context of arithmetic.

Signed Integers

Signed integers are stored in a computer using 2's complement. As you recall, when computing the 2's complement of a number it was necessary to know how many bits were to be used in the final result; leading zeroes were appended to the most significant digit in order to make the number the appropriate length. Since the process of computing the 2's complement involves first computing the 1's complement, these leading zeros become leading ones, and the left most bit of a negative number is therefore always 1. In computers, the left most bit of a signed integer is called the "**sign bit**".

Consider an 8 bit signed integer: let us begin with $0\,0\,0\,0\,0\,0\,0\,0_2$ and start counting by repeatedly adding 1:

- When you get to 127, the integer has a value of $0\,1\,1\,1\,1\,1\,1\,1_2$; this is easy to see because you know now that a 7 bit integer can contain a value between 0 and $2^7 - 1$, or 127. What happens when we add 1?
- If the integer were unsigned, the next value would be $1\,0\,0\,0\,0\,0\,0\,0_2$, or 128 (2^7). But since this is a signed integer, $1\,0\,0\,0\,0\,0\,0\,0_2$ is a negative value: the sign bit is 1!
- Since this is the case, we must ask the question: what is the decimal value corresponding to the signed integer $1\,0\,0\,0\,0\,0\,0\,0_2$? To answer this question, we must take the 2's complement of that value, by first taking the 1's complement and then adding one.
- The 1's complement is $0\,1\,1\,1\,1\,1\,1\,1_2$, or decimal 127. Since we must now add 1 to that, our conclusion is that the signed integer $1\,0\,0\,0\,0\,0\,0\,0_2$ must be equivalent to decimal -128!

Odd as this may seem, it is in fact the only consistent way to interpret 2's complement signed integers. Let us continue now to "count" by adding 1 to $1\,0\,0\,0\,0\,0\,0\,0_2$:

- $1\,0\,0\,0\,0\,0\,0\,0_2$ + $0\,0\,0\,0\,0\,0\,0\,1_2$ is $1\,0\,0\,0\,0\,0\,0\,1_2$.
- To find the decimal equivalent of $1\,0\,0\,0\,0\,0\,0\,1_2$, we again take the 2's complement: the 1's complement is
  $0\,1\,1\,1\,1\,1\,1\,0_2$ and adding 1 we get $0\,1\,1\,1\,1\,1\,1\,1_2$ (127) so $1\,0\,0\,0\,0\,0\,0\,1_2$ is equivalent to -127.
- We see then that once we have accepted the fact that $1\,0\,0\,0\,0\,0\,0\,0_2$ is decimal -128, counting by adding one works as we would expect.

- Note that the most negative number which we can store in an 8 bit signed integer is -128, which is -2^{8-1}, and that the largest positive signed integer we can store in an 8 bit signed integer is 127, which is $2^{8-1} - 1$.
- The number of integers between -128 and + 127 (inclusive) is 256, which is 2^8; this is the same number of values which an unsigned 8 bit integer can contain (from 0 to 255).

- Eventually we will count all the way up to $1\,1\,1\,1\,1\,1\,1\,1_2$. The 1's complement of this number is obviously 0, so
  $1\,1\,1\,1\,1\,1\,1\,1_2$ must be the decimal equivalent of -1.

Using our deliberations on 8 bit signed integers as a guide, we come to the following observations about signed integer arithmetic in general:

- **if a signed integer has n bits, it can contain a number between -2^{n-1} and $+(2^{n-1} - 1)$**.

- **since both signed and unsigned integers of n bits in length can represent 2^n different values, there is no inherent way to distinguish signed integers from unsigned integers simply by looking at them; the software designer is responsible for using them correctly**.

- no matter what the length, if a signed integer has a binary value of all 1's, it is equal to decimal -1.

You should verify that a signed short integer can hold decimal values from -32,768 to +32,767, a signed long integer can contain values from -2,147,483,648 to +2,147,483,647 and a signed double integer can represent decimal values from -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

There is an interesting consequence to the fact that in 2's complement arithmetic, one expects to throw away the final carry: in unsigned arithmetic a carry out of the most significant digit means that there has been an overflow, but in signed arithmetic an overflow is not so easy to detect. In fact, signed arithmetic overflows are detected by checking the consistency of the signs of the operands and the final answer. A signed overflow has occurred in an addition or subtraction if:

- the sum of two positive numbers is negative;
- the sum of two negative numbers is positive;
- subtracting a positive number from a negative one yields a positive result; or
- subtracting a negative number from a positive one yields a negative result.

Integer arithmetic on computers is often called "**fixed point**" arithmetic and the integers themselves are often called fixed point numbers. Real numbers on computers (which may have fractional parts) are often called "floating point" numbers, and they are the subject of the next section.

Go to: Title Page Table of Contents Index
