CS1100 - Lecture 14

Instructor : Jasine Babu

Teaching Assistants : Nikhila K N, Veena Prabhakaran

Representation of Integers

There are many possible ways of represnting integer values. The representation we are most familiar with, is known as the decimal number system. But there are many other ways to represent integers. Many of us have also come across number systems like Roman number system as well. A number system is called a positional value system, if the number system has a fixed number `b` as base and the representation of any number requires only `b` distinct symbols and each position in the representation of a number has a power of `b` as its positional value. Roman number system is not a positional value system and that makes it more difficult compared to the decimal system to directly add or multiply two numbers represented using Roman number system. We will now look into some positional value systems.

Decimal Number System

The decimal number system has exactly ten basic symbols, `0` to `9`, with which all integers can be represented. It is a positional value system and it has base value `10`. Each position from right to left has a weight asscociated with it, which is a power of the base value `10`, depending on the position of the digit. For example, in the decimal number `752`, the positional value of 2 is 10^0, the positional value of 5 is 10^1 and the positional value of 7 is 10^2. Note that,

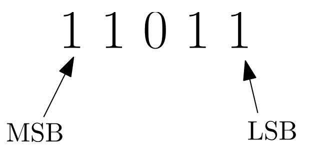
$$752 = 7 \times 10^2 + 5 \times 10^1 + 2 \times 10^0$$

In this representation, each power of `10` is the place value of the corresponding digit.

Binary Number System

The binary number system is the another positional value number system. This number system has only two basic symbols `0` and `1` and the system has base 2. The symbols in this numbers system is called *bits*.

Suppose we have a binary number



MSB: most significant bit

LSB: Least significant bit

In such a representation, the positional value of each bit is a power of 2. From the right to left, value of each position increases as increasing powers of 2. The right most bit is called the least significant bit and it has positional value 2^0. The next bit from the right has positional value 2^1 and so on.

The calculation of decimal equivalent of binary number `11011` is shown below.

$$11011_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 8 + 0 + 2 + 1 = 27_{10}$$

It is not difficult to see that any positive integer can be represnted using binary represenation. To convert a decimal number to binary, we will divide the number repeatedly by 2 and collect the remainders in the reverse order. Please revisit Experiment 4 in Lab Excercise Week 5 for converting a number from its decimal to binary represenation.

Hexa-decimal Number System

This number system has 16 symbols and it has the base value 16. In this number system uses digits from `0` to `9` and alphabets `A` to `F`. In similar way as we did the conversion from binary to decimal and vice versa, it is also possible to calculate the decimal equivalent of hexa-decimal numbers and hexa-decimal equivalent of decimal numbers. Suppose we have a hexa-decimal number `12F`. The calculation of decimal equivalent of hexa-decimal number `12F` is shown as follows.

$$12F_{16} = 1 \times 16^2 + 2 \times 16^1 + 15 \times 16^0 = 256 + 32 + 15 = 303_{10}$$

To convert a decimal number to hexa-decimal, divide the number repeatedly by 16 and collect the remainders in the reverse order (in a similar way as we divided repeatedly by 2 and collected the remainders for converting from decimal to binary).

Octal Number System

The Octal number system has only symbols from `0` to `7`. This is also a positional value number system and it has the base value `8`. As in case of binary and hexa decimal number systems, finding the decimal equivalent to each octal number and vice vera are possible. The calculation of decimal equivalent to the octal number `1357` is shown below.

$$1375_8 = 1 \times 8^3 + 3 \times 8^2 + 7 \times 8^1 + 5 \times 8^0 = 512 + 192 + 56 + 5 = 765_{10}$$

Exercise (Binary to Octal and Hexa-decimal conversions)

1. Compute the binary represenation of 765_{10}. Compare this binary represenation with the octal represenation of 765_{10} given above. Can you make out any relation between the octal and the binary represenations? (Consider three bits at a time from the right end of the binary represenation and compute the octal value represented by each of these three bit blocks).

2. Given the binary represenation of some positive integer `n`, how can we convert it into its corresponding octal represenation?

3. Given the octal represenation of some positive integer `n`, how can we convert it into its corresponding binary represenation?

4. Compute the hexa-decimal represenation of 765_{10}. Compare this hexa-decimal represenation with the binary represenation of 765_{10} given above. Can you make out any relation between the hexa-decimal and the binary represenations?

Representation of Integers in Computers

Since computers are made up of electronic devices like transistors and diodes, which usually operate based on a *high* and a *low* input/output voltage levels, it is natural to interpret a *high* voltage level as `1` and a *low* voltage level as `0`. This makes the binary number system the natural choice for representing numbers in present day computers.

Usually the unit to measure the storage of data in computers is known as *bytes*. A *byte* is a collection 8 *bits*. The binary represenation of 27_{10} is `1 1 0 1 1` which requires only 5 *bits*. If we want to represent it using *1 byte*, it is enough to fill the first three positions with 0s, since it will not affect the value. The following figure shows the *8 bit(1 byte)* binary representation of decimal number 27_{10}, as it is done in a computer.

0	0	0	1	1	0	1	1

If a computer uses *32 bits (4 bytes)* to store an integer, it will store the decimal number 27 as shown in the figure below.

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	0	0	1	1	0	1	1

Signed Numbers

The common represenation used in computers to store signed numbers is the 2's complement represenation. In this represenation, the weight of the left most bit (MSB) is negative. The following figure shows the 2's complement represenation of a binary number using k bits. The weights of each position is shown below the corresponding bit. Each

b_{k-1}	b_{k-2}	b_{k-3}	$\cdots$	b_2	b_1	b_0
$-(2^{k-1})$	2^{k-2}	2^{k-3}	$\cdots$	2^2	2^1	2^0

bit can be 0 or 1. The decimal value of the number represented above will be

$$b_{k-1} \times -(2^{k-1}) + b_{k-2} \times 2^{k-2} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

Note that, negative numbers will have their left most bit 1 and positive numbers will have their left most bit 0.

The following figure shows the signed 2's complement representation of the positive number 27 using 8 bits.

0	0	0	1	1	0	1	1

Consider a signed binary number `1 0 0 1 1 0 1 1` which is assumed to be in a 2's complement represenation using 8 bits. The decimal equivalent of this number is

$$1 \times -(2^7) + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -101_{10}$$

Note that, the difference between the 2's complement represenations of 27_{10} and -101_{10} is only in the left most bit. This is because, $-101_{10} = -128_{10} + 27_{10}$ and $128_{10} = 2^{k-1}$ where k is the number of bits used in the represenation.

In general, if n is a positive number less than 2^{k-1}, we get the 2's complement represenation of $-(2^{k-1} - n)$ by flipping the left most bit in the 2's complement represenation of n from 0 to 1. Or in other words, if p is a negative number, whose magnitude `m` is less than or equal to 2^{k-1}, then the 2's complement represenation of p can be obtained by first taking the *k-bit* binary represenation of $2^{k-1} - m$ and flipping the left most bit from 0 to 1.

Why 2's complement represenation?

As human beings, 2's complement represenation may seem difficult to understand. It would have been easier for us if the 2's complement represenation of -101_{10} can be obtained by taking the 2's complement represenation of $+101_{10}$ and flip the left most bit. Such a represenation is called the *sign-magnitude represenation.*

However, the computer represenation of numbers should be suitable for performing arithmetic operations like addition and subtraction efficiently. When we are doing operations like adding a positive number with a negative number or subtracting one positive number from another positive number, it turns out that 2's complement represenation allows the computations to be done more efficiently.

Example:
Suppose we have two numbers $+27_{10}$ and -27_{10}. The sum of these two numbers is `0`. In sign-magnitude form, the represenation of $+27$ is `0 0 0 1 1 0 1 1` and the represenation of -27 is `1 0 0 1 1 0 1 1`. Suppose we try to do an addition in a similar way as we do addition of decimal numbers. We will first add the right most bits and record the resultant bit and the carry bit (if any). The carry from the right most position will be added with the second bit from the right of both the numbers and so on.

```
carry       1 1   1 1

+27   0 0 0 1 1 0 1 1

-27   1 0 0 1 1 0 1 1
      ---------------
      1 0 1 1 0 1 1 0
```

Since the result `1 0 1 1 0 1 1 0` is in sign-magnitude represenation, it corresponds to the decimal value -54. This result will be wrong, because the expected result is 0. Therefore, if we have to properly do addition in sign-magnitude represenation, the method of addition has to be different. The way to do this turns out to be more complicated than the method discussed above.

Now, consider adding $+27$ and -27 using 2's complement represenation using the method which we used above. The 2's complement represenation of these numbers are respectively `0 0 0 1 1 0 1 1` and `1 1 1 0 0 1 0 1`.

$$\begin{array}{lcccccccc}
\text{carry} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \\
+27 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
\text{-}27 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Note that, the result is correct now.