Contents

Computers use *binary* (base 2) number system, as they are made from binary digital components (known as transistors) operating in two states - on and off. In computing, we also use *hexadecimal* (base 16) or *octal* (base 8) number systems, as a *compact* form for represent binary numbers.

Computers uses binary system in their internal operations, as they are built from binary digital electronic components. However, writing or reading a long sequence of binary bits is cumbersome and error-prone. Hexadecimal system is used as a compact form or shorthand for binary bits. Each hex digit is equivalent to 4 binary bits, i.e., shorthand for 4 bits, as follows:

0H (0000B) (0D)	1H (0001B) (1D)	2H (0010B) (2D)	3H (0011B) (3D)
4H (0100B) (4D)	5H (0101B) (5D)	6H (0110B) (6D)	7H (0111B) (7D)
8H (1000B) (8D)	9H (1001B) (9D)	AH (1010B) (10D)	BH (1011B) (11D)
CH (1100B) (12D)	DH (1101B) (13D)	EH (1110B) (14D)	FH (1111B) (15D)

Integers, for example, can be represented in 8-bit, 16-bit, 32-bit or 64-bit. You, as the programmer, choose an appropriate bit-length for your integers. Your choice will impose constraint on the range of integers that can be represented. Besides the bit-length, an integer can be represented in various *representation* schemes, e.g., unsigned vs. signed integers. An 8-bit unsigned integer has a range of 0 to 255, while an 8-bit signed integer has a range of -128 to -127: both representing 256 distinct numbers.

Integer Representation

Integers are *whole numbers*

Computers use *a fixed number of bits* to represent an integer. The commonly-used bit-lengths for integers are 8-bit, 16-bit, 32-bit or 64-bit. Besides bit-lengths, there are two representation schemes for integers:

1. *Unsigned Integers*: can represent zero and positive integers.
2. *Signed Integers*: can represent zero, positive and negative integers. Three representation schemes had been proposed for signed integers:
    1. Sign-Magnitude representation
    2. 1's Complement representation
    3. 2's Complement representation

n-bit Unsigned Integers

Unsigned integers can represent zero and positive integers, but not negative integers. The value of an unsigned integer is interpreted as "*the magnitude of its underlying binary pattern*".

E.g. Example 1:

Suppose that n=8 and the binary pattern is $0100\ 0001_2$, the value of this unsigned integer is $1\times2^0 + 1\times2^6$

$$= 65_{10}$$

Signed Integers

Signed integers can represent zero, positive integers, as well as negative integers.

Three representation schemes are available for signed integers:

1. Sign-Magnitude representation
2. 1's Complement representation
3. 2's Complement representation

In all the above three schemes, the *most-significant bit* (msb) is called the *sign bit*. The sign bit is used to represent the *sign* of the integer - with 0 for positive integers and 1 for negative integers. The *magnitude* of the integer, however, is interpreted differently in different schemes.

1. Signed Magnitude Representation

So far, we've been working with all positive numbers

Now we need a way to represent signed numbers like-18_{10} or -34_{10}

One solution is to add an extra digit to the front of our binary number to indicate whether the number is positive or negative. In computer terminology, this digit is called a **sign bit**.

Remember that a "bit" is simply another name for a binary digit

When our number is positive, we make our sign bit zero, and

When our number is negative, we make our sign bit one.

This approach is called the **Signed Magnitude Representation**

Exercise: convert the decimal numbers -5_{10} and -1_{10} to binary using 4-bit Signed Magnitude Representation

1. First, we convert 5 and 1 to binary.

```
101   (5)
  1   (1)
```

2. Now we add a sign bit to each one. Notice that we have padded '1' with zeros so it will have four bits.

```
0101  (5)
0001  (1)
```

3. To make our binary numbers negative, we simply change our sign bit from '0' to '1'.

```
1101 (-5)
1001 (-1)
```

Since we are using 4-bit signed magnitude representation, we know the first bit is our sign and the remaining three bits are our number (Be sure that you do not mistake the number 1101_2 for 13_{10})

Example 1: Suppose that no. of bits (n) $n=8$ and the binary representation is $0\ 100\ 0001_2$.
Sign bit is 0 ⇒ positive
Absolute value is $100\ 0001_2 = 65_{10}$
Hence, the integer is $+65_{10}$

Example 2: Suppose that $n=8$ and the binary representation is $1\ 000\ 0001_2$.
Sign bit is 1 ⇒ negative
Absolute value is $000\ 0001_2 = 1_{10}$
Hence, the integer is -1_{10}

Example 3: Suppose that $n=8$ and the binary representation is $0\ 000\ 0000_2$.
Sign bit is 0 ⇒ positive
Absolute value is $000\ 0000_2 = 0_{10}$
Hence, the integer is $+0_{10}$

Example 4: Suppose that $n=8$ and the binary representation is $1\ 000\ 0000_2$.
Sign bit is 1 ⇒ negative

Absolute value is $000\ 0000_2 = 0_{10}$
Hence, the integer is -0_{10}

The drawbacks of sign-magnitude representation are:

1. There are two representations ($0000\ 0000_2$ and $1000\ 0000_2$) for the number zero, which could lead to inefficiency and confusion.
2. Positive and negative integers need to be processed separately.

While representing negative numbers with signed magnitude is simple, performing arithmetic with signed magnitude is difficult.

What we would really like to have is a way of representing signed numbers that simplifies our binary arithmetic.

Suppose we could represent signed numbers in such a way that all our computations could be completed using only addition. (This would allow our computer processor to perform all arithmetic operations with only addition circuitry.)

We already know that multiplication is really repeated addition, so this is not a problem.

This leaves division and subtraction. But division is just repeated subtraction.

Now we are left with only subtraction.

How can we convert our subtraction problems to addition?

Consider the subtraction problem 30_{10} - 6_{10}.

- We can convert this problem to an equivalent addition problem by changing 6_{10} to -6_{10}.
- Now we can state the problem as 30_{10} + (-6_{10}) = 24_{10}.

We can do something similar to this in binary by representing negative numbers as complements. We will look at two ways of representing signed numbers using complements: **1's complement** and **2's complement**.

2. 1's complement

Representing a signed number with 1's complement is done by changing all the bits that are 1 to 0 and all the bits that are 0 to 1. Reversing the digits in this way is also called complementing a number.

Let's look at an example in 4-bit arithmetic. How can we represent the number -5_{10} in 1's complement?

First, we write the positive value of the number in binary.

0101 (+5)

Next, we reverse each bit of the number so 1's become 0's and 0's become 1's

1010 (-5)

This table is 4-bit binary numbers in **1's complement notation**

Binary	Decimal
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0
1111	-0
1110	-1
1101	-2
1100	-3
1011	-4
1010	-5
1001	-6
1000	-7

NOTE:

- All of the negative values begin with a 1
- The most significant bit always tells us the sign of the number

- The only exception to this rule is -0

  In 1's complement, we have two ways of representing the number zero.

- The values +0 to +7 are the same as the normal binary representation. Only the negative values must be complemented.

Example 1: Suppose that *n*=8 and the binary representation 0 100 0001_2.
Sign bit is 0 ⇒ positive
Absolute value is 100 $0001_2 = 65_{10}$
Hence, the integer is $+65_{10}$

Example 2: Suppose that *n*=8 and the binary representation 1 000 0001_2.
Sign bit is 1 ⇒ negative
Absolute value is the complement of 000 0001_2, i.e., 111 $1110_2 = 126_{10}$
Hence, the integer is -126_{10}

Example 3: Suppose that *n*=8 and the binary representation 0 000 0000B.
Sign bit is 0 ⇒ positive
Absolute value is 000 0000B = 0D
Hence, the integer is +0D

Example 4: Suppose that *n*=8 and the binary representation 1 111 1111_2.
Sign bit is 1 ⇒ negative
Absolute value is the complement of 111 1111_2, i.e., 000 $0000_2 = 0_{10}$
Hence, the integer is -0_{10}

Converting into 1's Complement

Here is a quick summary of how to find the 1's complement representation of any decimal number *x*.

1. If *x* is positive, simply convert *x* to binary.
2. If *x* is negative, write the positive value of *x* in binary
3. Reverse each bit.

Again, the drawbacks are:

1. There are two representations (`0000 0000B` and `1111 1111B`) for zero.
2. The positive integers and negative integers need to be processed separately.

3. 2's complement

Representing a signed number with 2's complement is done by adding 1 to the 1's complement representation of the number. To illustrate, let's look at the same example we used for 1's complement. How can we represent the number -5_{10} in 2's complement using 4-bits?

First, we write the positive value of the number in binary.	**0101 (+5)**
Next, we reverse each bit to get the 1's complement.	**1010**
Last, we add 1 to the number.	1011 (-5)

Four basic rules in order to perform **binary addition**

Rule 1	Rule 2	Rule 3	Rule 4
0 + 0 0	0 + 1 1	1 + 0 1	1 + 1 10

two 1s always generate a carry to the next column

Exers: Add

```
  1
  1+
____
```

```
1 +
1 +
1
-------
```

```
 10
  1+
____
```

Adding Two Binary Numbers Tutorial:

http://courses.cs.vt.edu/~csonline/NumberSystems/Lessons/AddingTwoBinaryNumbers/index.html

Exers: Use binary arithmetic to add : $3_{10}+5_{10}$

$7_{10} + 4_{10}$

$26_{10}+12_{10}$

0011010 + 001100 = 00100110

1 1	carry
0 0 1 1 0 1 0	$= 26_{10}$
+0 0 0 1 1 0 0	$= 12_{10}$
0 1 0 0 1 1 0	$= 38_{10}$