Numbering Systems

By Majid Haghoo

Integers

Introduction

Consider a number such as 888. The rightmost 8 is read and understood as eight, the middle one as eighty, and the leftmost one as eight hundred. Why the same symbol (8) is considered so differently when it is used in different positions within the same number?

The current system uses Arabic numerals: 0, 1, 2, ... (vs. Roman numerals: I, II, III, IV, ...). Each symbol, called a **digit**, has two values:

- Digit value
- Position value

The position value for the rightmost digit is 1, for the next digit (to the left) is 10, then 100, and so on.
For example consider 31685: (read from right)

$$31685 = 3 \text{ x } 10000 + 1 \text{ x } 1000 + 6 \text{ x } 100 + 8 \text{ x } 10 + 5 \text{ x } 1 = 3 \text{ x } 10^4 + 1 \text{ x } 10^3 + 6 \text{ x } 10^2 + 8 \text{ x } 10^1 + 5 \text{ x } 10^0$$

The 10-symbol Arabic numeral system that we are using is called **decimal**. The ten symbols are chosen probably because we have ten fingers. The number of symbols also determines the **base** of the system. The decimal system has base 10. There are many other systems. We can count in any systems. For example, when we count in decimal, we count as follow:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, **10**, 11, 12, 13, 14, 15, 16, 17, 18, 19, **20**, 21, ..., 98, 99, **100**

If we want to count in base 6, then we have:

Base 6	0	1	2	3	4	5	**10**	11	12	13	14	15	**20**	21	22	23	...	53	54	55	**100**	101	102
Decimal Value	0	1	2	3	4	5	**6**	7	8	9	10	11	**12**	13	14	15	...	33	34	35	**36**	37	38

In the decimal system, when we pass 9, then we use 10 to count the next number. Since in this system (base 6), we have only 6 symbols (0-5), hence we cannot use 6 and above. So when we pass 5, we have to use two symbols (10) to represent 6. So in this system, 10 is equivalent to 6 in decimal, 11 is equivalent to 7 in decimal and so on.

Exercise: Count in base 2 up to 15 numbers: (Hint: the last binary number will be 1110)

Base 2															
Decimal Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

In addition to the decimal system, we can define other systems - infinitely many. But we are interested in four systems as shown in the following table. **Decimal** is our natural number system. The **binary** system is used because computer hardware uses a two-state technology, each binary digit id called **bit**, which can be **0** or **1**. The **hexadecimal** (hex for short) system has a particular relation with binary: every hex digit is exactly 4 binary digits. And finally **octal** is an alternative to hexadecimal system: every octal digit is exactly three binary digits.

Popular Number Systems			
Name	**Base**	**Symbols (Digits)**	**Purpose**
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	Natural
Binary	2	0, 1	Hardware
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	Software, shorthand for binary
Octal	8	0, 1, 2, 3, 4, 5, 6, 7	Software, shorthand for binary

Many high-level programming languages support hexadecimal and octal system. Binary number are not supported but some operations are performed in binary, such as C/C++ bit-wise & (And), || (Or), and ^ (Xor, exclusive Or) operations. The hex digits A-F are not case sensitive; however the uppercase is most preferred.

Understanding these number systems is essential to understanding programming and other computer concepts in-depth. They are also very useful in understanding, among other things, Internet IP address, networks and sub-networks. The number systems are hardware and software independent: regardless of what hardware platform or what software is used these number systems will not be affected. We might have a new computer architecture, a new operating system, a new programming language, or even new techniques for passing parameters. They all still will use these number systems as described here.

We will study decimal, binary, and hexadecimal systems and their inter-relations. Octal is very similar to hexadecimal. We will describe converting between decimal, binary, and hexadecimal; binary arithmetic, and two's complement system.

Conversions

We will study conversions from decimal, binary and hexadecimal.

Converting from Decimal to Binary

Given a decimal number, we want to convert it to binary. We divide the number repeatedly by 2 until we reach zero. During the process, we record quotient and remainder of divisions. For example, convert decimal number 11 to binary.

1. Divide 11 by 2. We get a quotient of 5 and remainder 1.
2. Then divide 5 by 2. We get 2 as quotient and 1 as remainder.
3. Then we divide 2 by 2. We get 1 as quotient and 0 as remainder.
4. Finally we divide 1 by 2. We get quotient 0 and remainder 1.

When the quotient reaches zero, we stop. Our number is the inverse series of remainders that were produced: 1, 1, 0, 1. Therefore, 11 decimal is equivalent to 1011 binary.

Example 1: Convert decimal **108** to binary.

Stop when reach 0

0	1	3	6	13	27	54	108	Quotient
	1	1	0	1	1	0	0	Remainder

Therefore, decimal **108** is binary **1101100** or **110_1100** (hyphen is for readability purpose).

Example 2: Convert **2233** from decimal to binary.

0	1	2	4	8	17	34	69	139	279	558	1116	2233	Quotient
	1	0	0	0	1	0	1	1	1	0	0	1	Remainder

Therefore, decimal **2233** is binary **100010111001** or **1000_1011_1001** (hyphen is for readability purpose).

Exercise: Convert **1234** from decimal to binary.

1234	Quotient
	Remainder

Converting from Decimal to Hexadecimal

The technique for converting decimal to hexadecimal is identical to that of decimal to binary, except that we divide by 16 rather than by 2. Remember that 16 is the base for hexadecimal system. In this system we A-F represent 10-15, respectively and so on.

Example 2: Convert **23579** from decimal to binary.

0	5	92	1473	23579	Quotient
	5	12 = C	1	11 = B	Remainder

Therefore, decimal **23579** is hex **5C1B**.

Exercise: Convert **7890** from decimal to binary.

7890	Quotient
	Remainder

Converting from Binary to Decimal

As we described before that a decimal number such as 31685 can be written as:

31685 = **3** x **10000** + **1** x **1000** + **6** x **100** + **8** x **10** + **5** x **1**

Or,

31685 = **3** x $\mathbf{10^4}$ + 1 x $\mathbf{10^3}$ + 6 x $\mathbf{10^2}$ + 8 x $\mathbf{10^1}$ + 5 x $\mathbf{10^0}$ (Note: $10^0 = 1$)

Similarly a binary number such as **11_0111_1001** can be written as:

$1 \times 2^9 + 1 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 =$

1 x 512 + 1 x 256 + 0 x 128 + 1 x 64 + 1 x 32 + 1 x 16 + 1 x 8 + 0 x 4 + 0 x 2 + 1 x 1 =

512 + 256 + 0 + 64 + 32 + 16 + 8 + 0 + 0 + 1 = **889**

We can simplify this conversion technique:

1. Write the binary number in an expanded form (more space between individual digits).
2. Below the binary number, from the rightmost digit, write 1 then 2 then 4, 8, 16, and so on. Let us call these numbers "**multiplicands**."

3. Multiply each binary digit by corresponding multiplicand.
4. Add the all products

1	1	0	1	1	1	1	0	0	1	Binary Number
512	256	128	64	32	16	8	4	2	1	Multiplicands
512	256	0	64	32	16	8	0	0	1	Products

Decimal equivalent = 512 + 256 + 0 + 64 + 32 + 16 + 8 + 0 + 0 + 1 = **889**

We can simply the process further by noting that (1) multiplying a number by zero yields zero and (2) multiplying a number by 1 yields the number itself. Thus, we can add together those numbers at the second row where there is 1 in the first row above them, ignoring the other columns. In the following table "zero" columns are grayed out.

1	1	0	1	1	1	1	0	0	1	Binary Number
512	256	128	64	32	16	8	4	2	1	Multiplicands

Exercise: Convert binary **1010_1010_1010** to decimal.

	Binary Number
	Multiplicands

Converting from Hexadecimal to Decimal

This conversion is very similar to conversion from binary to decimal; however, we cannot simplify as much as we did for binary to decimal conversion.

1. Write the hexadecimal number in an expanded form (more space between individual digits).
2. Below the hex number, from the rightmost digit, write 1 then 16 then 256 (16^2), 4096 (16^3), and so on. Let us call these numbers "**multiplicands**."
3. Multiply each binary digit by corresponding multiplicand.
4. Add the all products

Example: Convert **6C3F** hex to decimal:

6	C=12	3	F=15	Hex Number
4096	256	16	1	Multiplicands
24576	3072	48	15	Products

The equivalent decimal number is: 15 + 48 + 3072 + 24576 = 27711

Exercise: Convert **A2B** hex to decimal.

	Hex Number
	Multiplicands
	Products

Converting Between Binary and Hexadecimal

The hexadecimal and binary systems have a close relationship between them. Every hexadecimal digit is exactly four binary digits. Therefore, you can convert the hex number digit-by-digit; there is no need for addition, multiplication, or division. Before we can proceed, however, we need to construct a table showing the binary values for all 16 hexadecimal digits.

Hexadecimal-Binary Conversion		
Decimal	**Hexadecimal**	**Binary**
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Converting from Hex to Binary

Using the table, replace hex digits, one-by-one, by its binary equivalent. For example, convert, hex number **4FED2** to binary. Replacing 0100 for 4, 1111 for F, 1110 for E, 1101 for D, and 0010 for 2, we have:

0100_1111_1110_1101_0010

For readability, we leave underscores between binary digits. Should we keep the leading zero(s)? Mathematically, the leading zeros are not significant; you might drop them. But when we dealing with computer memory word or registers, the leading zeros are as significant as others are.

Converting from Binary to Hex

Beginning from the right, from groups of 4-digit binaries. If needed, you may add leading zero(s) to the leftmost group to make it 4 digits. Then using the table, replace each group with its equivalent hexadecimal value. Example,

10_1110_0001_0110_0111_0001_1111

Each group must be 4 digits, therefore add two leading zeros to the left most group:

0010_1110_0001_0110_0111_0001_1111

Replace each group from the table: **2 E 1 6 7 1 F** = **2E1_671F**

Convert `1A2B6F` to binary

Convert `1_1111_1100_0011_1011_0101` to hex.

Binary Arithmetic

In the decimal system, when we two numbers, we begin from the rightmost digit and add digit by digit until the additions is complete. When adding two single digits whose sum exceeds 9 (highest digit in decimal), we deduct 10 from the sum and carry it (10) as 1 to next higher digit. For example, consider adding 27 and 46. We begin by adding 6 and 7, which is 13. Since 13 is more than 9, we deduct 10 from 13, reducing to 3, and 10 is carried as 1 to next higher digit. So we add 1 to sum of 2 and 4, making it 7, thus 27 + 46 = 73.

Binary Addition

When in adding in binary, we will have **carry** when sum of two single digits exceeds 1 (the highest digit in binary). In this case, we add the carry to the sum of next two digits (to the left). We continue this process until the addition is complete. Examples:

First Number		1	0	1
Second Number		1	0	0
Carry	1			
Result	**1**	**0**	**0**	**1**

First Number		1	0	1	1	0	1	1
Second Number		1	1	1	1	0	1	0
Carry	1	1	1	1		1		
Result	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**

First Number		1	1	1	1	1	1	1
Second Number								1
Carry	1	1	1	1	1	1	1	
Result	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

(Can you think of a similar addition in decimal?)

Creating Hex-Binary table by counting

Now that we have learned binary addition, we can build the hexadecimal binary table presented before.

Begin from 0000

Add 1 to 0000, obtaining 0001

Add 1 to 0001, obtaining 0010

And continue...

Exercise: Compute the remaining elements of the table.

Exercise: Perform each of the following binary additions:

1 + 1_0000,
1 + 1_1111
111+ 1111
1000_0000 + 1111_1111
1_0011 + 1_0001,
110_1001_0001 + 1110
11_1111_1111 + 11_1111

Now Convert each number to decimal including added numbers and verify your calculations.

Representing Integers in computer

A **bit** (stands for **binary digit**) is the smallest unit of storage in computers. A bit can either be 0 or 1 and **never anything else**. Different devices handle bits in different manners. A memory bit can either contain a high-voltage, or a low-voltage and nothing in between. The high voltage corresponds to 1 and low voltage corresponds to 0. A bit in magnetic devices such as hard disk is a magnetized spot. The two opposite magnetic orientations determine value of bit. In optical devices, each spot can be a reflective or non-reflective, giving us two possibilities.

In practice, the bits are grouped together, forming a **word** or **computer word**. A word is typically consists of 8, 16, 32, 64 bits, or more. In this lecture, we will assume a word can be any number of bits. All bits in a word must be either 1 or 0. They can not have anything else or they cannot be left empty. The following shows a few examples of 9-bit words:

Word in textbook is defined as 16-bits only

Do not get confused

1	1	0	1	0	0	0	0	1

1	1	1	1	1	1	1	1	1

0	0	0	0	1	0	0	0	1

The following show some impossible values in words: (what is the problem?)

W	e	d	n	e	s	d	a	y

1	2	3	4	5	6	7	8	9

1	1		0	0	0	0	0	1

Given a number such as 435, it is converted to binary 1_1011_0011 and stored in those nine bits. If a number when converted to binary is less than the word size, we fill bits from the right side; all unused bits on the left side are filled with zeros.

Examples: Show each of the following decimal numbers in 9-bit words:

20, 207, 333

After converting 20 to binary, we have 1_0100. These five bits occupy the five bits on the right side, leaving 4 bits, which are filled with zeros (shaded).

0	0	0	0	1	0	1	0	0

Decimal 207 is equivalent to 8-digit binary 1100_1111, which fills the right 8 bits; the remaining leftmost bit is filled with zero.

0	1	1	0	0	1	1	1	1

Decimal 333 is equivalent to 9-digit binary 1_0100_1101, which fills all nine available bits.

1	0	1	0	0	1	1	0	1

Exercise: Show each of the following decimal numbers in 9-bit words:

0, 1, 10, 123, 505

If the decimal number when converted to binary exceeds the word size, then we say we have **overflow** and that number cannot be represented with given word size. In practice, programming languages support 8-bit, 16-bit, and 32-bit words for integers. The last one allows us to represent integers as large as two billion. It is the programmer's responsibility to choose a number type (within given possibilities) so that minimizes the overflow potential.

There is a formula between the word size and range of numbers that can be represented using that word. First we work with unsigned (no negative) numbers. Later we will show how to deal with negative numbers. Let **n** be the word size. The range of the unsigned numbers that can be represented with n bits is:

$0 : 2^n - 1$

Examples: Find range for unsigned numbers for each on the following word size:

4, 5, 8, 16

4	$0:2^4-1$, yielding 0:15
5	$0:2^5-1$, yielding 0:31
8	$0:2^8-1$, yielding 0:255
16	$0:2^{16}-1$, yielding 0:65535

Exercises: Find range for unsigned numbers for each on the following word size:

1, 3, 6, 10

Two's Complement

We learned how to represent non-negative (zero and positive) integers in computers. How we represent negative numbers as well. We need to put side a single bit for the sign of the number. Of course, we cannot use + or – symbols in binary. We use the leftmost bit for this purpose and we use 0 as sign of zero and positive integers and 1 for negatives. One technique is use the remaining bits to store the number itself. This is called **sign and magnitude** technique. There is a better technique, which is called **Two's complement**. We will study this technique in details. In particular we will study how to convert from decimal to two's complement and back to decimal.

Let us assume again that n is the word size for our computer. Since one bit is put aside for the sign of the number, hence, we have only n-1 bits for the magnitude of the number. The range of the signed numbers that can be represented with n bits is:

$-2^{n-1} : 2^{n-1} - 1$

Examples: Find range for unsigned numbers for each on the following word size:

4, 5, 8, 16

4	$-2^{4-1}:2^{4-1}-1$, yielding –8 : 7
5	$-2^{5-1}:2^{5-1}-1$, yielding –16 : 15
8	$-2^{8-1}:2^{8-1}-1$, yielding –128 : 127
16	$-2^{16-1}:2^{16-1}-1$, yielding –32,768 : 32,767

Note: If you look at programming books, you might notice that the range of integer type is –32,768:32,767. This is because many programming languages use 16 bits to represent integers.

Exercises: Find range for unsigned numbers for each on the following word size:

1, 3, 6, 10

Converting from Decimal to Two's Complement

Let us use a **10-bit word size** for our examples in this part. The sign bit is bordered bold. This is for the sake of discussion only. Hardware treats the sign bit like other bits. It even does not know it is the sign bit or what is the sign. It does not what is negative (or positive number). Software does.

Converting **positive integers** to two's complement representation is like a regular conversion from decimal to binary as we studied before. The only difference is to watch for possible overflow and fill the remaining bits, if any, with zeros. Filling the remaining bits with zeros automatically takes care of assigning the sign bit. Remember that we have only n-1 bits (and not n bits) for the number itself. The converted binary must not exceed n-1 digits.

Examples: Show each of the following decimal numbers in 10-bit two's complement system:

10, 100, 1000

Before proceed, establish the range for this 10-bit two's complement system. Using the formula that we learned for this:

$-2^{10-1} : 2^{10-1} - 1$

Or, -512:511.

Decimal 10 is 1010 in binary; we have:

0	0	0	0	0	0	1	0	1	0

Decimal 100 is 1100100; so we have:

0	0	0	1	1	0	0	1	0	0

Decimal 1000 is 11_1110_1000. This is exceeds 9 digits, so we have **overflow**.

Exercises: Show each of the following decimal numbers in 10-bit two's complement system: 0, 6, 77, 888

Converting negative numbers to two's complement system is a three-step process as follow:

1. Ignoring the sign, convert the number to binary.
2. After conversion, change every 1 to 0 and every 0 to 1.
3. Add 1 to the number above.

Example: Convert decimal –88 to 10-bit two's complement system.

1. Convert positive 88 to binary 101_1000:

0	**0**	**0**	1	0	1	1	0	0	0

2. Change every 1 to 0 and every 0 to1:

1	1	1	0	1	0	0	1	1	1

3. Add 1 to the above number yields the final result:

1	1	1	0	1	0	1	0	0	0

Example: Convert decimal –512 to 10-bit two's complement system.

What is the range of 10-bit two's complement system? It is –512:511. So it is within the range.

1. Convert positive 512 to binary 10_0000_0000. Doesn't this exceeds the 9-digit limit. Yes it does. However this is a special case and we can proceed.

1	0	0	0	0	0	0	0	0	0

2. Change every 1 to 0 and every 0 to1:

0	1	1	1	1	1	1	1	1	1

3. Add 1 to the above number yields the final result:

1	0	0	0	0	0	0	0	0	0

This yields the same thing we obtained in step 1, above. Again this is the special case, so we had the number in the step one. So for a given two's complement system the smallest negative number of the range is always has 1 as its sign bit and all the remaining bits are zeros.

Example: Convert decimal –600 to 10-bit two's complement system.

What is the range of 10-bit two's complement system? It is –512:511. So –600 is out of range and would produce **overflow**.

Exercises: Show each of the following decimal numbers in 10-bit two's complement system:

-1, -6, -77, -888

Converting from Two's Complement to Decimal

First check the sign bit. If it is zero, then do a simple binary-decimal conversion as we have learned before.

Example: convert the following 10-bit two's complement number to decimal:

0	0	1	1	1	1	1	0	0	1

The sign bit is zero. So we do a simple binary-decimal conversion.

0	0	1	1	1	1	1	0	0	1
	512	128	64	32	16	8	4	2	1

So the decimal number is 128 + 64 + 32 + 16 + 8 + 1 = 249

For negative numbers, we can do the three-step process mentioned above in reverse. However, there is a simple method. Like the binary-decimal conversion, put the powers of two below the number. This time, however, **add the numbers that have zero above them**. And add an extra one to the result. Change the sign of the number to negative.

Example: convert the following 10-bit two's complement number to decimal:

1	1	0	1	1	1	0	0	0	1

The sign bit is 1 so the number must be negative.

1	1	0	1	1	1	0	0	0	1
	256	128	64	32	16	8	4	2	1

So consider the numbers that have zero above them (un-shaded ones). So the number is: 128 + 8 + 4 + 2 = 142. Adding an extra one yields 143. So the decimal equivalent is –143.

Example: convert the following 10-bit two's complement number to decimal:

1	0	0	0	0	0	0	0	0	0

The sign bit is 1 so the number must be negative.

1	1	0	1	1	1	0	0	0	1
	256	128	64	32	16	8	4	2	1

So consider the numbers that have zero above them (un-shaded ones). So the number is: 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 511. Adding an extra one yields 512. So the decimal equivalent is –512.

Exercises: Convert each of the following 10-bit two's complement number to decimal.

1	0	1	0	1	0	1	0	1	0

1	0	1	1	0	0	1	1	1	1

0	1	1	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

Review Exercises

1. Convert binary 1110_0101 to decimal.
2. Convert 55 decimal to binary.
3. Convert 55 hex to decimal.
4. Add 11_1110_0011 + 1_0101_0001.
5. Find the range for unsigned 12-bit system.
6. Find the range for signed 12-bit two's complement system.
7. Convert 55 to 7-bit two's complement system.
8. Convert -3 to 7-bit two's complement system.
9. Convert the following number from two's complement to decimal.

1	1	1	1	1	1	1

10. Convert the following number from two's complement to decimal.

0	1	1	0	1	1	0

Numbers in Mathematics vs. Numbers in Computers

There are two differences:

- In mathematics, we don’t have limit. In computers we do: number of bit allocated for each integer
- In mathematics, leading zeros are not important, in Computers all bit are important.

Therefore converting from mathematics to computer, we have to check for overflow, and sign bit and all of that.

Advantages of Two's Complement System

They are two methods to represent all integers

1. Sign & magnitude (also called one's complement)
2. Two's complement

Sign & magnitude, seem easier but has one problem. There are two representations for zero:

1 000000000000000 and **0** 000000000000000

This might seem a minor issue, but remember for many zero-involved operations, you have check two possibilities.

Two's complement does not have this problem. However, there is another big advantage of two's complement is for addition (and subtractions). Here is the reasons:

Consider adding two numbers, where either of them could be positive/negative:

1. Positive + Positive
2. Negative + negative
3. Larger positive + smaller negative (in magnitude)
4. Larger negative (in magnitude) + smaller positive.

Therefore it must be checked for four possibilities before adding.

In two's complement just add them. Do not worry about sign. It be will automatically computed. See the examples below.

This is actually a computer architecture issue, but it closely relates to assembly as well. That is why we are discussing here.

To avoid unnecessary details, we use 8-bit signed two's complement examples. In this system, the range of numbers will be `-128:127`.

Examples:

	Example	Total	Sign of numbers	Result	Note
1	50 + 62	112	Positive + Positive	Within range	112 is within range
2	85 + 92	177	Positive + Positive	Overflow	177 is outside of range
3	(-73) + (-19)	-92	Negative + Negative	Within range	-92 is within range
4	(-88) + (-77)	-165	Negative + Negative	Overflow	-165 is outside of range
5	88 + (-77)	11	Positive + Negative	Within range	11 is within range *
6	(-125) + 91	-34	Negative + Positive	Within range	-34 is within range *

Parentheses are for clarity.

*** Adding positive and negative numbers always is within their limit.**

Example 1: **50 + 62, positive + positive**

Carry into sign bit

Carry out of sign bit

	S								Decimal	
	0	0	1	1	0	0	1	0	50	Number 1
	0	0	1	1	1	1	1	0	62	Number 2
0	0	**1**	**1**	**1**	**1**	**1**	0			Carry
	0	1	1	1	0	0	0	0	**112**	Result

Analysis: **No overflow**, when there are no carries into and out of sign bit.

Example 2: **85 + 92, positive + positive**

	S								Decimal	
	0	1	0	1	0	1	0	1	85	Number 1
	0	1	0	1	1	1	0	0	92	Number 2
0	1	0	**1**	**1**	**1**	0	0			Carry
	1	0	1	1	0	0	0	1	**177**	Result

Analysis: **Overflow**, there is a carry into sign bit but not out of sign bit.

Example 3: **(-73) + (-19), negative + negative**

	S								Decimal	
	1	0	1	1	0	1	1	1	-73	Number 1
	1	1	1	0	1	1	0	1	-19	Number 2
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**			Carry
	1	0	1	0	0	1	0	0	**-92**	Result

Analysis: **No overflow**, there is a carry into sign bit and a carry out of sign bit.

Example 4: **(-88) + (-77), negative + negative**

	S								Decimal	
	1	0	1	1	0	0	0	0	-88	Number 1
	1	0	1	1	0	0	1	1	-77	Number 2
1	**0**	**1**	**1**	**0**	**0**	**0**	**0**			Carry
1	0	1	1	0	0	0	1	1	**-165**	Result

Analysis: **Overflow**, there is no carry into sign bit but there is a carry out of sign bit.

Example 5: **88 + (-77), positive + negative**										
	S								Decimal	
	0	1	0	1	1	0	0	0	88	Number 1
	1	0	1	1	0	0	1	1	-77	Number 2
1	**1**	**1**	**1**							Carry
1	0	0	0	0	1	0	1	1	**11**	Result

Analysis: **No overflow**, there is a carry into sign bit and a carry out of sign bit.

Example 6: **(-125) + 91, negative + positive**										
	S								Decimal	
	1	0	0	0	0	0	1	1	-125	Number 1
	0	1	0	1	1	0	1	1	91	Number 2
0	**0**					**1**	**1**			Carry
	1	1	0	1	1	1	1	0	**-34**	Result

Analysis: **No overflow**, when there are no carries into and out of sign bit.

Carry into sign bit	Carry out of sign bit	Overflow	$C_{in} \oplus C_{out}$
0	0	No	0
1	0	Yes	1
0	1	Yes	1
1	1	No	0

Overflow = $C_{in} \oplus C_{out}$

So overflow is exclusive or of two carries.

Real Numbers

Float-Point Numbers

Two's complement is not used hare! Put it aside for now!

Real numbers are represented by floating-point format.

Floating-point bits are divided into three parts: sign, exponent, and mantissa.

1. **Sign**: represents positive or native numbers. 0 for positives, 1 for negatives. **Two's complement format is not used in floating point numbers**. Sign always takes one bit.
2. **Exponent**: represent magnitude of the number, both large numbers (10^{38} or more) and very small numbers (10^{-38} or less). Note that negative exponent does not mean numbers is negative. The **Sign** bit determines if the number is negative or positive.
3. **Mantissa**: represents how many digits are accurate. For example a number such as 0.1234567890 is accurate approximately 6 or 7 digits: it is really 0.1234567 or 0.1234568 (rounded).

They are three sizes to represent floating point numbers:

1. Single-precision, 32 bits
2. Double-precision, 64 bits
3. Extended-precision, 80 bits

The table shows how these bits are divided, accuracy, and magnitude.

	# of bits	Sign	Exponent	Mantissa	Precision (decimal digits)	Magnitude
Single-precision	32	1	8	23 *	6-7	$10^{\pm 38}$
Double-precision	64	1	11	52 **	14-15	$10^{\pm 308}$
Extended-precision	80	1	15	64 ***	17-18	$10^{\pm 4932}$

* Note: In the textbook, it talks about **24-bit mantissa**. That is the number of mantissa bits can be represented. **But only 23 bits allocated for mantissa**. During tests, read questions carefully.

** In double precision, 52 bits allocated, but 53 bits can be represented.

*** In the extended-precision, 64 bit allocated and 64 bit represented.

In the following while we explain floating-points representations, we also explain why with 23 allocated bits, we can get 24 bits of information.

Converting real numbers to floating-point format.

We use 32-bit format to avoid lengthy computations.

Example: Convert decimal 87.65 to 32-bit floating point representation.

Before describing the algorithm, we will covert integer part and fraction part separately.

Computing Mantissa

1. Covert **87** to binary: **101_0111**
2. Now covert **0.65** to binary
3. Multiply the fraction part by 2 (0.65 by 2, you get 1.30)
4. Record the whole part (which is either 0 or 1)
5. Continue multiplying the fraction by 2, record the whole parts
6. Stop when you reach zero, or you have enough digits in binary
7. 0.65 x 2 = **1**.30 **1** record this column, digits are produced from **left to right**.
8. 0.3 x 2 = **0**.6 **0**
9. 0.6 x 2 = **1**.2 **1**
10. 0.2 x 2 = **0**.4 **0**
11. 0.4 x 2 = **0**.8 **0**
12. 0.8 x 2 =**1**.6 **1**

The fraction part so far is **101001** and continues. See complete 24 bits below.

So the number is **1010_111.1_0100_1100_1100_1100** (24 bits)

Or, the number is: **1010_111.1_0100_1100_1100_1100** x 2^0

Move the decimal point to the **left** until only a single **1** left on the left side. And add 1 to exponent of 2 for each move.

For some numbers, you might need to move the decimal point to right to normalize the number.

1010_11.11_0100_1100_1100_1100 x 2^1

1010_1.111_0100_1100_1100_1100 x 2^2

1010_.1111_0100_1100_1100_1100 x 2^3

1.010_1111_0100_1100_1100_1100 x 2^6

This is called **normalized**: Namely leaving just 1 to the left of decimal point.

Since, always there would be a single **1** to the left of the decimal point, there is no need to keep it, but we can take this bit into account for computations.

That is why the textbook, talks about 24 bits (rather than 23).

So the mantissa is **010_1111_0100_1100_1100_1100**

Computing Exponent

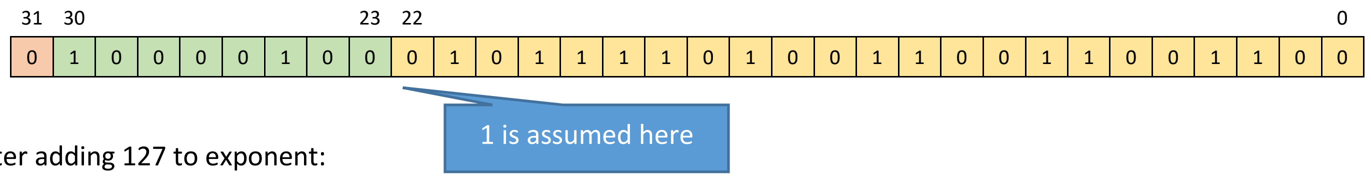
Exponent represents magnitude of the number. It can be negative, zero, or positive. Negative exponent does not mean the number is negative. It implies number's magnitude is small.

The exponent computations is performed by so called **excel 127** for single precision numbers. It simply means add 127 to exponent: 127 + 6 = 133. Convert 133 to 8-bit unsigned format. It is 1000_0101.

Thus:

1. Sign = 0
2. Exponent = 1000_0101
3. Mantissa = **010_1111_0100_1100_1100_1100**

So combining all together we have:



After adding 127 to exponent:

1. Exponent 1-126 implies real exponent is negative.
2. Exponent 127 means real exponent is 0
3. Exponent 128-255 means real exponent is positive.

The table shows the exponents for single-precision, double-precision, and extended-precision.

Exponent	# of bits	Excess-n	Magnitude
32 bits	8	127	$10^{\pm 38}$
64 bits	11	1023	$10^{\pm 308}$
80 bits	15	15383	$10^{\pm 4932}$

How exponent in binary and magnitude in decimal are related? (Optional)

Consider the equation:

$$10^x = 2^n$$

$x = n * \log10(2) = n * 0.30103$

For 32-bit number, 8 bit exponent (-127, 127) $x = \pm 127 * 0.30103 = \pm 38$

For 64-bit number, 11 bit exponent (-1023, 1023) $x = \pm 1023 * 0.30103 = \pm 308$

For 80-bit number, 15 bit exponent (-16383m 16383) $x = \pm 16383 * 0.30103 = \pm 4932$

How mantissa in binary and precision in decimal are related? (Optional)

Works in similar manner.

Final Notes:

Difference between Computer and Math

Mathematics is the crux of the computing or programming. After all, ALU (Arithmetic Logic Unit) is the one who performs all operations. And computer originally was invented to do mathematical computations. However, there are differences between computer and math. The differences regarding numbers include:

1. Computer deals with finite arithmetic, but there is no limit in math.
2. In math, leading zeros are not important, but in computer leading zeros are important as the other parts.
3. In math, trailing zero after the decimal point is mostly not important, in computer there are important.

Therefore, when doing assignments, read carefully to see what and how the question has been asked:

- Do the problem mathematically. These questions are asking for conversion, and there is no mention of word-side or bits.
- Do the problem with computer in mind. In these questions, computer-related terms are mentioned, such word-side, signed, unsigned, 2's complement…

Converting between Bases

Only useful number system bases are decimal, binary, hexadecimal, and octal. However, **academically**, you must learn how convert from any base, say **N**, to any other base, say **M**.

First convert from base **N** to decimal, then convert from decimal to base M.

Example: convert **1234** from base **5** to base **6**.

First convert to decimal:

1 x 5^3 + 2 x 5^2 + 3 x 5^1 + 4 x 5^0 = 1 x 125 + 2 x 25 + 3 x 5 + 4 x 1 = **194** (in decimal)

Now convert from decimal to base 6:

0	5	32	194	Quotient
	5	2	2	Remainder

So this number in base **6** is **522**.

Therefore, in assignments you might be asked to manipulate numbers in any base!

==

This question is not part of required materials:

Can a number base be negative? (I read questions like these were asked in Microsoft interviews)