Chapter 1
Integer Representations

Latest update: 27/09/2016

Table of Contents

I. Number Bases

1. Definitions

1.1. Base-10 or Decimal Numeral System

Since human beings have ten fingers, they use the **'base-10 numeral system'** (also called the **'decimal numeral system'**) in order to represent numbers.

This base-10 system uses **ten symbols (digits)**: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

These symbols should be written down side by side and may be separated by a point. For example, a decimal number can be represented as follows:

$$1\ 5\ 7\ .\ 6\ 2\ 5$$

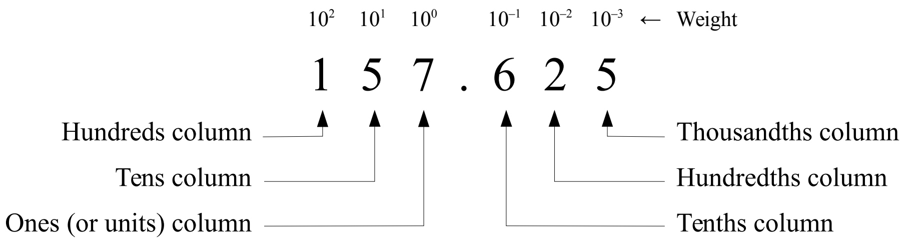
The point is called the **'radix point'** (or the **'decimal point'** in the base-10 system) and is used to separate the integer part of a number (157 for the example above) from its fractional part (0.625).

Each symbol has a numbered position (or column).

$$\begin{array}{ccccccc} \scriptstyle 2 & \scriptstyle 1 & \scriptstyle 0 & & \scriptstyle -1 & \scriptstyle -2 & \scriptstyle -3 \\ 1 & 5 & 7 & . & 6 & 2 & 5 \end{array}$$

The position of a digit is positive when the digit is to the left of the radix point and negative when it is to the right. The first position to the left of the point is the position 0 (or the column 0).

In this **'positional notation'**, the value of a symbol depends on its position. Each digit should be multiplied by a **'weight'**. The weight of a symbol is 10 raised to the position of the symbol.



This number can be expressed as a sum of products:

$$\mathbf{157.625} = \mathbf{1}\times10^2 + \mathbf{5}\times10^1 + \mathbf{7}\times10^0 + \mathbf{6}\times10^{-1} + \mathbf{2}\times10^{-2} + \mathbf{5}\times10^{-3}$$

1.2. Base-2 or Binary Numeral System

A computer stores and manipulates only 0s and 1s. Therefore, when it comes to studying the representation of numbers in a computer, the **‘base-2 numeral system’** (also called the **‘binary numeral system’**) proves to be more appropriate than the decimal numeral system.

The base-2 system uses **two symbols**: 0 and 1.

In the same way as the decimal system, these symbols should be written down side by side and may be separated by a radix point. Each symbol has a numbered position (or column). For example, a binary number can be represented as follows:

7	6	5	4	3	2	1	0		–1	–2	–3
1	0	0	1	1	1	0	1	.	1	0	1

In the binary system, the radix point can be called the **‘binary point’** and the weight of a symbol is 2 raised to the position of the symbol:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
1	0	0	1	1	1	0	1	.	1	0	1

or

128	64	32	16	8	4	2	1		.5	.25	.125
1	0	0	1	1	1	0	1	.	1	0	1

This number can be expressed as a sum of products:
$10011101.101_2 = \mathbf{1}\times2^7 + \mathbf{0}\times2^6 + \mathbf{0}\times2^5 + \mathbf{1}\times2^4 + \mathbf{1}\times2^3 + \mathbf{1}\times2^2 + \mathbf{0}\times2^1 + \mathbf{1}\times2^0 + \mathbf{1}\times2^{-1} + \mathbf{0}\times2^{-2} + \mathbf{1}\times2^{-3}$
$10011101.101_2 = 128 + 16 + 8 + 4 + 1 + 0.5 + 0.125$
$10011101.101_2 = 157.625_{10}$

Notes:

- When we manipulate several different bases at a time, the bases should be added in subscript to the right of the numbers. The bases are then explicitly specified.
- If a base is not specified in subscript, the context should be clear.

In order to manipulate binary numbers easily, it is advisable to know the first positive and negative powers of two by heart.

Positive Power of Two	
2^{0}	1
2^{1}	2
2^{2}	4
2^{3}	8
2^{4}	16
2^{5}	32
2^{6}	64
2^{7}	128
2^{8}	256
2^{9}	512
2^{10}	1,024
2^{11}	2,048
2^{12}	4,096
2^{13}	8,192
2^{14}	16,384
2^{15}	32,768
2^{16}	65,536

Negative Power of Two	
2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625
2^{-5}	0.03125

1.3. Base-16 or Hexadecimal Numeral System

The binary numeral system is very useful to represent numbers inside a computer, but unfortunately, this binary representation is not very convenient for human beings; even small numbers require many digits, which is very confusing. That is the reason why **'the base-16 numeral system'** (also called the **'hexadecimal numeral system'**) is commonly used. What is more, any binary number can easily be converted into its hexadecimal equivalent (this point will be described in detail later on).

The base-16 system uses **sixteen symbols**: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

The letters A, B, C, D, E and F are used to represent the base-10 values 10, 11, 12, 13, 14 and 15 respectively.

In the same way as previously, these symbols should be written down side by side and may be separated by a radix point. Each symbol has a numbered position (or column). For example, a hexadecimal number can be represented as follows:

$$\overset{1}{9}\;\overset{0}{\mathrm{D}}\;.\;\overset{-1}{\mathrm{A}}$$

In the hexadecimal system, the radix point can be called the **‘hexadecimal point’** and the weight of a symbol is 16 raised to the position of the symbol:

$$\overset{16^1}{9}\quad \overset{16^0}{\mathrm{D}}\ .\ \overset{16^{-1}}{\mathrm{A}}$$

This number can be expressed as a sum of products:
$9D.A_{16} = \mathbf{9}\times16^1 + \mathbf{13}\times16^0 + \mathbf{10}\times16^{-1}$
$9D.A_{16} = 144 + 13 + 0.625$
$9D.A_{16} = 157.625_{10}$

In order to manipulate hexadecimal numbers easily, it is advisable to know the first positive and negative powers of sixteen by heart.

Positive Power of Sixteen	
16^0	1
16^1	16
16^2	256
16^3	4,096
16^4	65,536

Negative Power of Sixteen	
16^{-1}	0.0625

1.4. Any Base Numeral System

More generally, a numeral system can be founded on many bases: if we consider a base-*b* system, *b* can be any whole number greater than or equal to two.

A base-*b* system uses *b* symbols. If the ten digits are not enough, letters of the alphabet can be used (starting with ‘A’).

These symbols should be written down side by side and may be separated by a radix point. Assuming that a_i represents a symbol of a base *b*, any number (*N*) of this base can be represented as follows:

$$\underbrace{\overset{b^n}{a_n}\quad \overset{b^{n-1}}{a_{n-1}}\quad \overset{b^{n-2}}{a_{n-2}}\quad \dots\quad \overset{b^2}{a_2}\quad \overset{b^1}{a_1}\quad \overset{b^0}{a_0}}_{\text{Integer Part}}\ \underset{\overset{\uparrow}{\text{Radix Point}}}{\bullet}\ \underbrace{\overset{b^{-1}}{a_{-1}}\quad \overset{b^{-2}}{a_{-2}}\quad \dots\quad \overset{b^{-m+1}}{a_{-m+1}}\quad \overset{b^{-m}}{a_{-m}}}_{\text{Fractional Part}}\quad \overset{\longleftarrow \text{ Weights}}{}$$

Lastly, this number can be expressed as a sum of products: $N = \sum_{i=-m}^{i=n} a_i b^i$

2. Conversions

2.1. Converting Integers Between 0 and 15

In order to perform decimal, binary and hexadecimal conversions effectively, you should be able to convert any integer between 0 and 15 instantly from memory. Therefore, the table below should be memorized:

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

It is noteworthy that, at the very most, only four digits are needed to represent any binary number between 0 and 15. The weights can only be 1, 2, 4 and 8, which makes the conversion easy.

For instance: $1101_2 = 8 + 4 + 1 = 13_{10}$

2.2. Converting Numbers from Any Base to Decimal

2.2.1. Method

To convert a base-*b* number into a decimal number:

- Write down the base-*b* number as a sum of products: $\sum_{i=-m}^{i=n} a_i b^i$ (see I.1.4. Any Base Numeral System)
- Work out the result.

2.2.2. Examples

- **4213.04_5 → base 10**
  $= 4\times5^3 + 2\times5^2 + 1\times5^1 + 3\times5^0 + 0\times5^{-1} + 4\times5^{-2}$
  $= 4\times125 + 2\times25 + 1\times5 + 3\times1 + 4\times0.04$
  $= 500 + 50 + 5 + 3 + 0.16$
  $= 558.16_{10}$

- **$1\ 0110\ 1001.0101_2$ → base 10**
  $= 1\times2^8 + 0\times2^7 + 1\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0 + 0\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 1\times2^{-4}$
  $= 256 + 64 + 32 + 8 + 1 + 0.25 + 0.0625$
  $= 361.3125_{10}$

- **$B3C.A_{16}$ → base 10**
  $= 11\times16^2 + 3\times16^1 + 12\times16^0 + 10\times16^{-1}$
  $= 11\times256 + 3\times16 + 12 + 10\times0.0625$
  $= 2{,}816 + 48 + 12 + 0.625$
  $= 2{,}876.625_{10}$

2.3. Converting Numbers from Decimal to Any Base

The method of converting the integer part of a number is different from the method of converting its fractional part. So, we have to convert these two parts separately.

2.3.1. Method for the Integer Part (Successive Division Method)

To convert the integer part of a decimal number into a base-*b* number:

- Divide the integer part of the decimal number successively by *b* until the quotient is 0.
- **The remainders make up the integer part of the base-*b* number.** Starting with the last remainder, write down all the remainders side by side from left to right (the last remainder should be the leftmost digit of the base-*b* number). If *b* is greater than ten, the values of some remainders may be represented by a letter.

2.3.2. Method for the Fractional Part (Successive Multiplication Method)

To convert the fractional part of a decimal number into a base-*b* number:

- Multiply the fractional part of the decimal number by *b*.
- Multiply the fractional part of the product thus obtained by *b*.
- Repeat the operation until the precision is high enough or until the fractional part of the product is 0.
- **The integer parts of the products make up the fractional part of the base-*b* number.** Starting with the first product, write down all the integer parts of the products side by side from left to right (the last integer part of the product should be the rightmost digit of the base-*b* number). If *b* is greater than ten, the values of some integer parts may be represented by a letter.

2.3.3. Examples

- **715.40625_{10} → base 2**

715 / 2 = 357	Remainder = **1**
357 / 2 = 178	Remainder = **1**
178 / 2 = 89	Remainder = **0**
89 / 2 = 44	Remainder = **1**
44 / 2 = 22	Remainder = **0**
22 / 2 = 11	Remainder = **0**
11 / 2 = 5	Remainder = **1**
5 / 2 = 2	Remainder = **1**
2 / 2 = 1	Remainder = **0**
1 / 2 = 0	Remainder = **1**

715_{10} = 10 1100 1011_2

0.40625 × 2 = **0**.8125
0.8125 × 2 = **1**.625
0.625 × 2 = **1**.25
0.25 × 2 = **0**.5
0.5 × 2 = **1**

0.4025_{10} = 0.01101_2

Therefore, **715.40625_{10} = 10 1100 1011.01101_2**

- **41,695.78125_{10} → base 16**

41,695 / 16 = 2,605	Remainder = **15**
2,605 / 16 = 162	Remainder = **13**
162 / 16 = 10	Remainder = **2**
10 / 16 = 0	Remainder = **10**

41,695_{10} = A2DF_{16}

0.78125 × 16 = **12**.5
0.5 × 16 = **8**

0.78125_{10} = $0.\text{C8}_{16}$

Therefore, **41,695.78125_{10} = A2DF.C8_{16}**

2.4. Converting Numbers Between Binary and Power-of-Two Bases

2.4.1. Method

Each digit of a base-2^n number is actually made up of *n* binary digits. Consequently, each digit of a base-2^n number can be converted independently into its binary equivalent. Similarly, a binary number can be split up into groups of *n* digits and each group can be converted into its base-2^n equivalent.

2.4.2. Examples

- **$10010110110011111.11100101001_2$ → base 16 (2^4)**
  = 0001 0010 1101 1001 1111.1110 0101 0010_2 ← *Groups of 4 digits*
  = $12D9F.E52_{16}$

- **$10010110110011111.11100101001_2$ → base 8 (2^3)**
  = 010 010 110 110 011 111.111 001 010 010_2 ← *Groups of 3 digits*
  = 226637.7122_8

- **$3A271E65.7B3C_{16}$ → base 2**
  = 0011 1010 0010 0111 0001 1110 0110 0101.0111 1011 0011 1100_2 ← *Groups of 4 digits*
  = $111010001001110001111001100101.01111011001111_2$

- **4270.634_8 → base 2**
  = 100 010 111 000.110 011 100_2 ← *Groups of 3 digits*
  = 100010111000.1100111_2

2.5. Converting Numbers from Any Base to Any Base

2.5.1. Method

If the two bases are powers of the same integer (n), the numbers should be converted via base n. Otherwise, the numbers should be converted via base 10.

2.5.2. Examples

- **$3D7.5_{16}$ → base 8** ← *$16 = 2^4$ and $8 = 2^3$ (via base 2)*
  = 0011 1101 0111.0101_2
  = 001 111 010 111.010 100_2
  = 1727.24_8

- **23_5 → base 8**

  $23_5 = 2×5 + 3$
  $23_5 = 13_{10}$

  13 / 8 = 1 Remainder = **5**
  1 / 8 = 0 Remainder = **1**

  $23_5 = 15_8$

II. Basic Binary Operations

1. Additions

Binary additions follow the same rules as decimal additions. For instance, let us perform the following addition: 1010110 + 1011101

Step 1

$$\begin{array}{ccccccccc}
 & & 1 & 0 & 1 & 0 & 1 & 1 & \mathbf{0} \\
+ & & 1 & 0 & 1 & 1 & 1 & 0 & \mathbf{1} \\ \hline
 & & & & & & & & \mathbf{1}
\end{array}$$

Step 2

$$\begin{array}{ccccccccc}
 & & 1 & 0 & 1 & 0 & 1 & \mathbf{1} & 0 \\
+ & & 1 & 0 & 1 & 1 & 1 & \mathbf{0} & 1 \\ \hline
 & & & & & & & \mathbf{1} & 1
\end{array}$$

Step 3

$$\begin{array}{ccccccccc}
 & & & & & & \mathbf{1} & & \\
 & & 1 & 0 & 1 & 0 & \mathbf{1} & 1 & 0 \\
+ & & 1 & 0 & 1 & 1 & \mathbf{1} & 0 & 1 \\ \hline
 & & & & & & \mathbf{0} & 1 & 1
\end{array}$$

Step 4

$$\begin{array}{ccccccccc}
 & & & & \mathbf{1} & \mathbf{1} & & & \\
 & & 1 & 0 & 1 & \mathbf{0} & 1 & 1 & 0 \\
+ & & 1 & 0 & 1 & \mathbf{1} & 1 & 0 & 1 \\ \hline
 & & & & & \mathbf{0} & 0 & 1 & 1
\end{array}$$

Step 5

$$\begin{array}{ccccccccc}
 & & & \mathbf{1} & \mathbf{1} & 1 & & & \\
 & & 1 & 0 & \mathbf{1} & 0 & 1 & 1 & 0 \\
+ & & 1 & 0 & \mathbf{1} & 1 & 1 & 0 & 1 \\ \hline
 & & & & \mathbf{1} & 0 & 0 & 1 & 1
\end{array}$$

Step 6

$$\begin{array}{ccccccccc}
 & & & \mathbf{1} & 1 & 1 & & & \\
 & & 1 & \mathbf{0} & 1 & 0 & 1 & 1 & 0 \\
+ & & 1 & \mathbf{0} & 1 & 1 & 1 & 0 & 1 \\ \hline
 & & & \mathbf{1} & 1 & 0 & 0 & 1 & 1
\end{array}$$

Step 7

$$\begin{array}{ccccccccc}
 & \mathbf{1} & & 1 & 1 & 1 & & & \\
 & & \mathbf{1} & 0 & 1 & 0 & 1 & 1 & 0 \\
+ & & \mathbf{1} & 0 & 1 & 1 & 1 & 0 & 1 \\ \hline
 & 1 & \mathbf{0} & 1 & 1 & 0 & 0 & 1 & 1
\end{array}$$

Step 8

$$\begin{array}{ccccccccc}
 & \mathbf{1} & & 1 & 1 & 1 & & & \\
 & & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
+ & & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ \hline
 & \mathbf{1} & 0 & 1 & 1 & 0 & 0 & 1 & 1
\end{array}$$

- **Step 1:** 0 + 1 = 1
- **Step 2:** 1 + 0 = 1
- **Step 3:** 1 + 1 = 10 (0 carry 1)
- **Step 4:** 1 + 0 + 1 = 10 (0 carry 1)
- **Step 5:** 1 + 1 + 1 = 11 (1 carry 1)
- **Step 6:** 1 + 0 + 0 = 1
- **Step 7:** 1 + 1 = 10 (0 carry 1)
- **Step 8:** 1 + 0 + 0 = 1

Therefore, 1010110 + 1011101 = 10110011

2. Subtractions

Binary subtractions follow the same rules as decimal subtractions. For instance, let us perform the following subtraction: 1101001 – 110110

Step 1

$$\begin{array}{cccccccc} & 1 & 1 & 0 & 1 & 0 & 0 & \mathbf{1} \\ - & 0 & 1 & 1 & 0 & 1 & 1 & \mathbf{0} \\ \hline & & & & & & & \mathbf{1} \end{array}$$

Step 2

$$\begin{array}{cccccccc} & 1 & 1 & 0 & 1 & 0 & {}_{1}\mathbf{0} & 1 \\ - & 0 & 1 & 1 & 0 & {}_{1}1 & \mathbf{1} & 0 \\ \hline & & & & & & \mathbf{1} & 1 \end{array}$$

Step 3

$$\begin{array}{cccccccc} & 1 & 1 & 0 & 1 & {}_{1}\mathbf{0} & {}_{1}0 & 1 \\ - & 0 & 1 & 1 & {}_{1}0 & {}_{1}\mathbf{1} & 1 & 0 \\ \hline & & & & & \mathbf{0} & 1 & 1 \end{array}$$

Step 4

$$\begin{array}{cccccccc} & 1 & 1 & 0 & \mathbf{1} & {}_{1}0 & {}_{1}0 & 1 \\ - & 0 & 1 & 1 & {}_{1}\mathbf{0} & {}_{1}1 & 1 & 0 \\ \hline & & & & \mathbf{0} & 0 & 1 & 1 \end{array}$$

Step 5

$$\begin{array}{cccccccc} & 1 & 1 & {}_{1}\mathbf{0} & 1 & {}_{1}0 & {}_{1}0 & 1 \\ - & 0 & {}_{1}1 & \mathbf{1} & {}_{1}0 & {}_{1}1 & 1 & 0 \\ \hline & & & \mathbf{1} & 0 & 0 & 1 & 1 \end{array}$$

Step 6

$$\begin{array}{cccccccc} & 1 & {}_{1}\mathbf{1} & {}_{1}0 & 1 & {}_{1}0 & {}_{1}0 & 1 \\ - & {}_{1}0 & {}_{1}\mathbf{1} & 1 & {}_{1}0 & {}_{1}1 & 1 & 0 \\ \hline & & \mathbf{1} & 1 & 0 & 0 & 1 & 1 \end{array}$$

Step 7

$$\begin{array}{cccccccc} & \mathbf{1} & {}_{1}1 & {}_{1}0 & 1 & {}_{1}0 & {}_{1}0 & 1 \\ - & {}_{1}\mathbf{0} & {}_{1}1 & 1 & {}_{1}0 & {}_{1}1 & 1 & 0 \\ \hline & \mathbf{0} & 1 & 1 & 0 & 0 & 1 & 1 \end{array}$$

- **Step 1:** 1 – 0 = 1
- **Step 2:** 0 – 1 → borrow 1 → 10 – 1 = 1
- **Step 3:** 0 – (1 + 1) = 0 – 10 → borrow 1 → 10 – 10 = 0
- **Step 4:** 1 – (0 + 1) = 1 – 1 = 0
- **Step 5:** 0 – 1 = 1 → borrow 1 → 10 – 1 = 1
- **Step 6:** 1 – (1 + 1) = 1 – 10 → borrow 1 → 11 – 10 = 1
- **Step 7:** 1 – (0 + 1) = 1 – 1 = 0

Therefore, 1101001 – 110110 = 110011

3. Multiplications

Binary multiplications follow the same rules as decimal multiplications. For instance, let us perform the following multiplication: 10110 × 101

$$\begin{array}{cccccccc|l} & & & 1 & 0 & 1 & 1 & 0 & \leftarrow \text{Multiplicand} \\ \times & & & & & 1 & 0 & 1 & \leftarrow \text{Multiplier} \\ \hline & & & 1 & 0 & 1 & 1 & 0 & \leftarrow \textbf{Step 1:} \text{ The rightmost multiplier digit is 1; write down the multiplicand} \\ + & & 0 & 0 & 0 & 0 & 0 & & \leftarrow \textbf{Step 2:} \text{ The next multiplier digit is 0; write down zeros shifted left} \\ + & 1 & 0 & 1 & 1 & 0 & & & \leftarrow \textbf{Step 3:} \text{ The next multiplier digit is 1; write down the multiplicand shifted left} \\ \hline & 1 & 1 & 0 & 1 & 1 & 1 & 0 & \leftarrow \textbf{Step 4:} \text{ Add the partial products} \end{array}$$

Therefore, 10110 × 101 = 1101110

4. Divisions

Binary divisions follow the same rules as decimal divisions. For instance, let us perform the following division: 11111001 / 110

Step 1:

$$\begin{array}{cccccccccc|l}
 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 1 & 0 & 0 & 1 & & 1\,1\,0 \\ \cline{11-11}
- & \mathbf{1} & \mathbf{1} & \mathbf{0} & & & & & & & \mathbf{1} \\ \cline{2-4}
 & & & \mathbf{1} & & & & & & &
\end{array}$$

- The divisor (110) goes into the first three digits of the dividend (111) one time.
- Record a ‘1’ in the quotient.
- Subtract the divisor (110) from the first three digits of the dividend (111).

Step 2:

$$\begin{array}{cccccccccc|l}
 & 1 & 1 & 1 & \mathbf{1} & 1 & 0 & 0 & 1 & & 1\,1\,0 \\ \cline{11-11}
- & 1 & 1 & 0 & \downarrow & & & & & & 1\,\mathbf{0} \\ \cline{2-4}
 & & & 1 & \mathbf{1} & & & & & &
\end{array}$$

- Bring down the next digit from the dividend (1) to the remainder.
- The new remainder (11) is smaller than the divisor (110).
- Record a ‘0’ in the quotient.

Step 3:

$$\begin{array}{cccccccccc|l}
 & 1 & 1 & 1 & 1 & \mathbf{1} & 0 & 0 & 1 & & 1\,1\,0 \\ \cline{11-11}
- & 1 & 1 & 0 & & \downarrow & & & & & 1\,0\,\mathbf{1} \\ \cline{2-4}
 & & & \mathbf{1} & \mathbf{1} & \mathbf{1} & & & & & \\
 & & - & \mathbf{1} & \mathbf{1} & \mathbf{0} & & & & & \\ \cline{4-6}
 & & & & & \mathbf{1} & & & & &
\end{array}$$

- Bring down the next digit from the dividend (1).
- The divisor goes into the new remainder (111) one time.
- Record a ‘1’ in the quotient.
- Subtract the divisor.

Step 4:

```
  1 1 1 1 1 0 0 1  | 1 1 0
– 1 1 0            |--------
  -----            | 1 0 1 0
      1 1 1        |
    – 1 1 0        |
      -----        |
          1 0      |
```

- Bring down the next digit from the dividend (0).
- The new remainder (10) is smaller than the divisor.
- Record a ‘0’ in the quotient.

Step 5:

```
  1 1 1 1 1 0 0 1  | 1 1 0
– 1 1 0            |--------
  -----            | 1 0 1 0 0
      1 1 1        |
    – 1 1 0        |
      -----        |
          1 0 0    |
```

- Bring down the next digit from the dividend (0).
- The new remainder (100) is still smaller than the divisor.
- Record a ‘0’ in the quotient.

Step 6:

```
  1 1 1 1 1 0 0 1  | 1 1 0
– 1 1 0            |--------
  -----            | 1 0 1 0 0 1
      1 1 1        |
    – 1 1 0        |
      -----        |
          1 0 0 1  |
        –   1 1 0  |
          -------  |
              1 1  |
```

- Bring down the next digit from the dividend (1).
- The divisor goes into the new remainder (1001) one time.
- Record a ‘1’ in the quotient.
- Subtract the divisor.

Step 7:

$$
\begin{array}{cccccccccc|l}
 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & & 1\ 1\ 0 \\ \cline{11-11}
- & 1 & 1 & 0 & & & & & & & 1\ 0\ 1\ 0\ 0\ 1\ \mathbf{.} \\ \cline{2-4}
 & & & 1 & 1 & 1 & & & & & \\
 & & - & 1 & 1 & 0 & & & & & \\ \cline{4-6}
 & & & & & 1 & 0 & 0 & 1 & & \\
 & & & & - & & 1 & 1 & 0 & & \\ \cline{6-9}
 & & & & & & & 1 & 1 & \mathbf{0} &
\end{array}
$$

- There is no more digit to bring down from the dividend.
- Record a binary point in the quotient.
- Bring down an extra 0 to the remainder.

Step 8:

$$
\begin{array}{cccccccccc|l}
 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & & 1\ 1\ 0 \\ \cline{11-11}
- & 1 & 1 & 0 & & & & & & & 1\ 0\ 1\ 0\ 0\ 1\ .\ \mathbf{1} \\ \cline{2-4}
 & & & 1 & 1 & 1 & & & & & \\
 & & - & 1 & 1 & 0 & & & & & \\ \cline{4-6}
 & & & & & 1 & 0 & 0 & 1 & & \\
 & & & & - & & 1 & 1 & 0 & & \\ \cline{6-9}
 & & & & & & & \mathbf{1} & \mathbf{1} & \mathbf{0} & \\
 & & & & & & \mathbf{-} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \\ \cline{8-10}
 & & & & & & & & & \mathbf{0} &
\end{array}
$$

- The divisor goes into the new remainder (110) one time.
- Record a ‘1’ in the quotient.
- Subtract the divisor.
- The final remainder is 0.

Therefore, 11111001 / 110 = 101001.1

III. Bits, Words and Bytes

1. Definitions

A '**bit'**, which is an abbreviation of **'binary digit'**, is the smallest unit of information a computer can store. A bit can be either 0 or 1.

A **'word'** consists of one or more bits that can be manipulated as a whole; that is, a fixed-size group of bits. For example:

- A 1-bit word is made up of 1 bit, which can be arranged into two patterns: 0 and 1.
- A 2-bit word is made up of 2 bits, which can be arranged into four patterns: 00, 01, 10 and 11.
- A 3-bit word is made up of 3 bits, which can be arranged into eight patterns: 000, 001, …, 110, 111.
- A 4-bit word is made up of 4 bits, which can be arranged into sixteen patterns: 0000, 0001, …, 1111.
- **An *n*-bit word is made up of *n* bits, which can be arranged into 2^n patterns.**

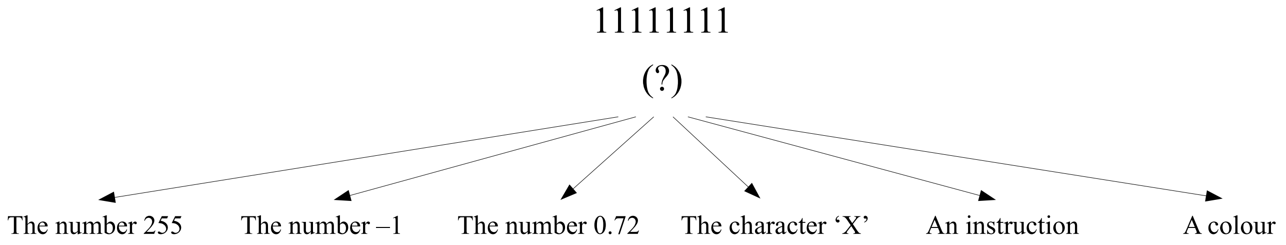
In some contexts, the size of a word can be implicit. For instance, when it comes to programming a 68000 microprocessor, the default size of a word is 16 bits; then the term 'word' means '16-bit word' and the term 'long word' means '32-bit word'. On the other hand, when it comes to programming an ARM microprocessor, the default size of a word is 32 bits; then the term 'word' means '32-bit word' and the term 'half-word' means '16-bit word'. Out of context, the size of a word is undefined and should be specified.

A **'byte'** is an 8-bit word, which can be arranged into 256 patterns. Nowadays, microprocessors are commonly byte oriented; meaning, they can manipulate pieces of data that are multiples of 8 bits.

The leftmost bit of a word is the **'most significant bit'** (usually abbreviated to **'MSB'**) and its rightmost bit is the **'least significant bit'** (usually abbreviated to **'LSB'**). The bits of an *n*-bit word are numbered from 0 to *n – 1*; the bit 0 is the LSB; the bit *n – 1* is the MSB.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	← bit number
Example of a 16-bit word:	1	0	1	1	0	0	1	1	1	0	0	1	0	0	0	1	
	↑															↑	
	MSB															**LSB**	

Now that you know what a word is, an important question must be considered: what does a word represent inside a computer? For instance, what does the following byte represent?



In fact, we cannot answer this question because the context is missing. **A word in itself represents nothing.** The representation of a word depends on the context. Out of context, all we know about a byte is that it can be arranged into 256 patterns of 0s and 1s. Therefore, according to the context, a programmer will be able to associate each pattern with a particular meaning. For instance, the byte 11111111_2 can represent the value 255 if it is used as an unsigned integer, or it can also represent the value –1 if it is used as a signed integer. Therefore, the byte 11111111_2 can represent anything according to the context.

To sum up, a computer can handle only groups of 0s and 1s. If we want to manipulate any other types of data (e.g. integers, characters, images, etc.), we have to encode them with 0s and 1s.

For the sake of convenience, a word is also commonly represented in its hexadecimal form. For instance, the byte 11111111_2 is equivalent to the byte FF_{16}.

2. One's Complement

The one's complement inverts each bit of a word.

Some examples:

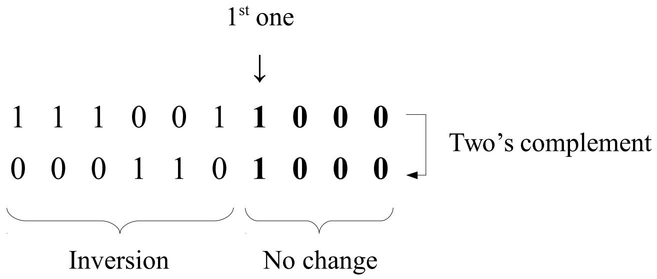
Word		**One's Complement of the Word**	
Binary	**Hexadecimal**	**Binary**	**Hexadecimal**
00000000	00	11111111	FF
00110110	36	11001001	C9
10011000	98	01100111	67
10111100	BC	01000011	43
11111111	FF	00000000	00

3. Two's Complement

The two's complement inverts each bit of a word and adds one. In other words, it is the one's complement plus one.

For instance, the two's complement of the 10-bit word 1110011000_2 is $0001100111_2 + 1_2 = 0001101000_2$

The two's complement of a word can also be obtained by keeping the first bits as they are (from the LSB up to the first '1'), and inverting all the others:



Some examples:

Word		Two's Complement of the Word	
Binary	**Hexadecimal**	**Binary**	**Hexadecimal**
00000000	00	00000000	00
00110110	36	11001010	CA
10011000	98	01101000	68
10111100	BC	01000100	44
11111111	FF	00000001	01

The two's complement is mostly used for encoding signed integers.

IV. Encoding Integers

1. Unsigned Integers

1.1. Introduction

There are many ways to encode unsigned integers. In this part, we will see the most commonly used by computers, which is also the most natural.

Each pattern of a word represents **directly** the value of a positive binary number as if the word itself were this number. That is to say, **the binary representation of the word is identical to the binary representation of the number**. For instance, if we want to encode the binary number 101_2 into an unsigned 8-bit word, this word will be: 00000**101**.

Since an n-bit word can be arranged into 2^n patterns, this word can represent integers from 0 to $2^n - 1$.

1.2. Encoding Unsigned Integers

To convert an unsigned integer into a binary word, use the method of converting **the integer part** of a base-10 number into a base-2 number. See I.2.3. Converting Numbers from Decimal to Any Base.

Example: Let us encode the number 50 into an 8-bit binary word.

50 / 2 = 25 Remainder = **0**
25 / 2 = 12 Remainder = **1**
12 / 2 = 6 Remainder = **0**
6 / 2 = 3 Remainder = **0**
3 / 2 = 1 Remainder = **1**
1 / 2 = 0 Remainder = **1**

$50_{10} = 00110010_2$

1.3. Decoding Unsigned Integers

To convert a binary word into an unsigned integer, use the method of converting a base-2 number into a base-10 number. See I.2.2. Converting Numbers from Any Base to Decimal.

Example: Let us decode the 6-bit word 100110_2.

$100110_2 = 32 + 4 + 2 = 38_{10}$

1.4. Unsigned Overflow

When operations are performed with fixed-size words, some results may be larger than the capacity of the words. Consequently, a result cannot be encoded properly; that is to say, the encoded result does not represent the right or the expected result. It is then said that an **‘unsigned overflow’** occurred.

For instance, let us consider the following 8-bit addition (the two operands and the result are 8 bits wide):

$$
\begin{array}{cccccccccccl}
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & {\scriptstyle 1} & & & & \\
 & & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & \leftarrow & 250_{10} \\
+ & & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \leftarrow & 10_{10} \\
\hline
 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \leftarrow & 4_{10}\ (\cancel{260}_{10}) \\
 & \uparrow & & & & & & & & & & \\
 & \text{Carry} & & & & & & & & & &
\end{array}
$$

The right result is 260 and needs 9 bits to be encoded. Actually, this ninth bit is the **‘carry’**. So, the 8-bit result is 4 and does not represent the expected result (the carry is left out).

A byte can be used to encode values between 0 and 255. Obviously, the expected result 260 is too large and cannot be encoded with 8 bits.

To take an actual example, have a look at the C language program below:

```c
#include <stdio.h>

int main()
{
    // Declare 3 unsigned 8-bit variables.
    unsigned char a, b, c;

    // Initialize the operands.
    a = 250;
    b = 10;

    // Perform the addition.
    c = a + b;

    // Display the 3 variables.
    printf("%u + %u = %u", a, b, c);

    return 0;
}
```

This program displays the following text on the terminal:

```
250 + 10 = 4
```

Actually, the incrementation cycles through the patterns of the word as shown below:

Decimal	Binary		
248	11111000		
249	11111001	+ 1	
250	11111010	+ 1	
251	11111011	+ 1	
252	11111100	+ 1	
253	11111101	+ 1	
254	11111110	+ 1	
255	11111111	+ 1	250 + 10 = 4
0	00000000	+ 1	
1	00000001	+ 1	
2	00000010	+ 1	
3	00000011	+ 1	
4	00000100	+ 1	
5	00000101	+ 1	

The same line of reasoning applies to subtraction:

Decimal	Binary		
248	11111000		
249	11111001	– 1	
250	11111010	– 1	
251	11111011	– 1	
252	11111100	– 1	
253	11111101	– 1	
254	11111110	– 1	
255	11111111	– 1	4 – 10 = 250
0	00000000	– 1	
1	00000001	– 1	
2	00000010	– 1	
3	00000011	– 1	
4	00000100	– 1	
5	00000101	– 1	

It is noteworthy that:

- 255 + 1 = 0
- 0 – 1 = 255

Notes:

- When it comes to adding numbers, the unsigned overflow can be called the **‘carry’**.
- When it comes to subtracting numbers, the unsigned overflow can be called the **‘borrow’**.

2. Signed Integers

2.1. Introduction

There are many ways to encode signed integers. In this part, we will see the most commonly used by computers, which is called the **‘two’s complement system’**.

The patterns of a fixed-size word are divided into halves. The first half is used to represent positive integers and the second half is used to represent negative integers.

Since an *n*-bit word can be arranged into 2^n patterns, this word can represent signed integers from $–2^{n-1}$ to $2^{n-1} – 1$.

For instance, a 4-bit word can represent integers from –8 to 7 as shown below:

Integer	**4-bit Word**	
–8	1000	Negative integers (MSB = 1)
–7	1001	
–6	1010	
–5	1011	
–4	1100	
–3	1101	
–2	1110	
–1	1111	
0	0000	Positive integers (MSB = 0)
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	

It is noteworthy that **the MSB is 0 when the number is positive and 1 when it is negative**. Therefore, the MSB of an encoded signed integer is also called the **'sign bit'**.

In the two's complement system, the two's complement of a word represents the additive inverse (or the opposite) of a number.

For instance, 2'sC (3_{10}) = -3_{10}:

- $3_{10} = 0011_2$
- 2'sC (0011_2) = $1101_2 = -3_{10}$

It is noteworthy that a byte can represent integers from –128 to 127.

2.2. Subtraction Using the Two's Complement

Since the two's complement represents the opposite of a number, subtraction can be replaced by addition as long as we ignore the carry.

For instance, let us consider the following 8-bit subtraction: $115_{10} - 62_{10}$

We know that:

- $115_{10} = 01110011_2$
- $62_{10} = 00111110_2$
- 2'sC (62_{10}) = 11000010_2

Subtraction

$$\begin{array}{cccccccccl} & 0 & 1 & {}_{1}1 & {}_{1}1 & {}_{1}0 & {}_{1}0 & 1 & 1 & \leftarrow 115_{10} \\ - & 0 & {}_{1}0 & {}_{1}1 & {}_{1}1 & {}_{1}1 & 1 & 1 & 0 & \leftarrow 62_{10} \\ \hline & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & \leftarrow 53_{10} \end{array}$$

Addition

$$\begin{array}{ccccccccccl} & 1 & 1 & & & & & 1 & & & \\ & & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & \leftarrow 115_{10} \\ + & & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & \leftarrow 2\text{'sC}\,(62_{10}) = -62 \\ \hline & \cancel{1} & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & \leftarrow 53_{10} \\ & \uparrow & & & & & & & & & \\ & \cancel{\text{Carry}} & & & & & & & & & \end{array}$$

The 8-bit result of the subtraction and that of the addition are the same: $00110101_2 = 53_{10}$

Therefore, $115_{10} - 62_{10} = 115_{10} + 2\text{'sC}\,(62_{10}) = 53_{10}$

2.3. Encoding Signed Integers

The method of converting signed integers depends on the sign of the integer.

If the integer is positive, it can be encoded as an unsigned integer.
See IV.1.2. Encoding Unsigned Integers.

If the integer is negative:

- Convert its absolute value into its binary form.
- Work out the two's complement of the absolute value.

Example: Let us encode the number –17 into an 8-bit binary word:

17 / 2 = 8	Remainder = **1**
8 / 2 = 4	Remainder = **0**
4 / 2 = 2	Remainder = **0**
2 / 2 = 1	Remainder = **0**
1 / 2 = 0	Remainder = **1**

$17_{10} = 00010001_2$
$-17 = 2\text{'sC}\,(00010001_2)$
$-17 = 11101111_2$

2.4. Decoding Signed Integers

The method of converting binary words into signed integers depends on the sign bit of the word (i.e. the MSB).

If the sign bit is 0, the integer is positive and can be decoded as an unsigned integer.
See IV.1.3. Decoding Unsigned Integers.

If the sign bit is 1, the integer is negative:

- Work out the two's complement of the word.
- Convert the two's complement in its decimal form and write down the minus sign in front of it.

Example: Let us decode the 6-bit word 100110_2.

The sign bit is 1:
$2\text{'sC}(100110_2) = 011010_2$
$011010_2 = 16 + 8 + 2 = 26_{10}$
$100110_2 = -26_{10}$

2.5. Signed Overflow

Just like with unsigned integers, some fixed-size operations may generate results that cannot be encoded properly. It is then said that a **‘signed overflow’** occurred.

For instance, let us consider the following 8-bit addition:

$$\begin{array}{cccccccccl}
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & & & \\
 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & \leftarrow 120_{10} \\
+ & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & \leftarrow 10_{10} \\
\hline
 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \leftarrow -126_{10}\ (\cancel{130}_{10}) \\
 & \uparrow & & & & & & & & \\
 & \text{Negative} & & & & & & & &
\end{array}$$

A byte can be used to encode values between –128 and 127. Obviously, the expected result 130 is too large and cannot be encoded. It is noteworthy that the signed result is negative and equal to –126.

To take an actual example, have a look at the C language program below:

```c
#include <stdio.h>

int main()
{
    // Declare 3 signed 8-bit variables.
    signed char a, b, c;

    // Initialize the operands.
    a = 120;
    b = 10;

    // Perform the addition.
    c = a + b;

    // Display the 3 variables.
    printf("%i + %i = %i", a, b, c);

    return 0;
}
```

This program displays the following text on the terminal:

```
120 + 10 = -126
```

Actually, the incrementation cycles through the patterns of the word as shown below:

Decimal	Binary	
118	01110110	
119	01110111	+ 1
120	01111000	+ 1
121	01111001	+ 1
122	01111010	+ 1
123	01111011	+ 1
124	01111100	+ 1
125	01111101	+ 1
126	01111110	+ 1
127	01111111	+ 1
–128	10000000	+ 1
–127	10000001	+ 1
–126	10000010	+ 1
–125	10000011	+ 1

120 + 10 = –126

The same line of reasoning applies to subtraction:

Decimal	Binary	
118	01110110	
119	01110111	– 1
120	01111000	– 1
121	01111001	– 1
122	01111010	– 1
123	01111011	– 1
124	01111100	– 1
125	01111101	– 1
126	01111110	– 1
127	01111111	– 1
–128	10000000	– 1
–127	10000001	– 1
–126	10000010	– 1
–125	10000011	– 1

– 126 – 10 = 120

2.6. Sign Extension

The sign-extension operation increases the number of bits used to encode a signed integer by replicating the sign bit to the left.

For instance, if we want to extend an 8-bit signed integer to a 16-bit signed integer, the sign bit of the 8-bit word (bit 7) is copied to all bits to the left (from bit 8 to bit 15).

Example for a positive number:

Decimal Representation	104
8-bit Binary Word	01101000
16-bit Binary Word	**00000000**01101000

Sign Extension

Example for a negative number:

Decimal Representation	–126
8-bit Binary Word	10001101
16-bit Binary Word	**11111111**10001101

Sign Extension

V. Other codes

There is a multiplicity of codes for various types of applications. We will only see a few of them.

1. BCD Code

BCD stands for **'Binary-Coded Decimal'**. This code is mostly used to display decimal numbers.

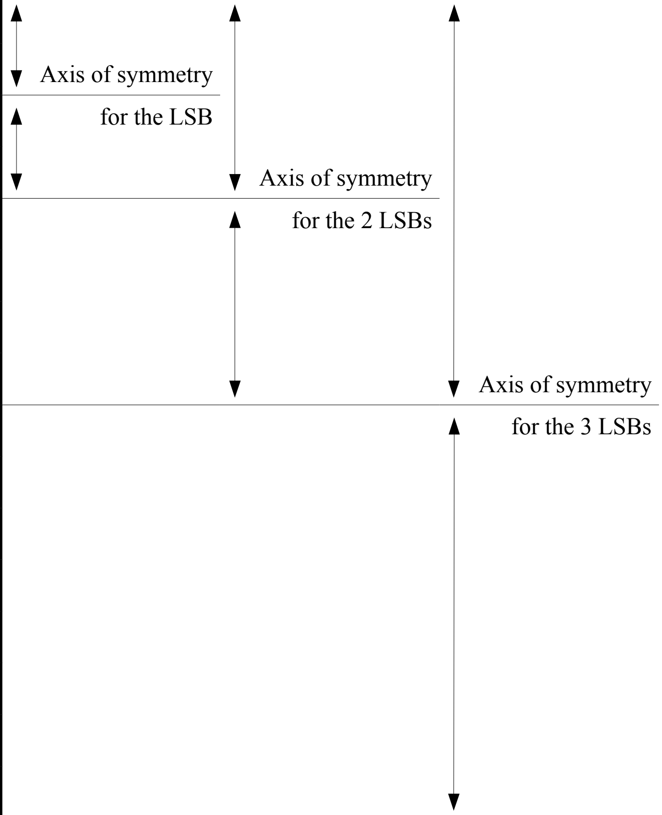
Decimal digits are encoded into groups of four bits. For example: $3{,}768_{10} = 0011\ 0111\ 0110\ 1000_2$

3	7	6	8	← Decimal
0011	0111	0110	1000	← BCD

2. Gray Code

The Gray code is another way to encode unsigned integers. The following table gives the first sixteen patterns of the Gray code:

Decimal	Natural Binary	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000



The Gray code is also called the **'reflected binary code'** because the least significant bits (LSBs) have axes of symmetry. Moreover, it is an **'unweighted code'**; that is to say, the digits do not have any weights.

The main characteristic of the Gray code is that only one bit changes from one pattern to another. That is the reason why this code is widely used in some algorithms, specific applications, error corrections and Karnaugh maps. The latter are used to reduce logical expressions and will be described in detail in a forthcoming chapter.

3. ASCII Code

ASCII stands for 'American Standard Code for Information Interchange'.

The ASCII code is used to encode characters. For instance:

- '0' = $30_{16} = 00110000_2$
- 'A' = $41_{16} = 01000001_2$
- 'a' = $61_{16} = 01100001_2$