Digital Logic

1 Data Representations

1.1 The Binary System

The Binary System is a way of writing numbers using only the digits 0 and 1. This is the method used by the (digital) computer.

The system we use to write numbers is called decimal (or denary).

In the DECIMAL system "fifty two" is written as 52.
In the BINARY system "fifty two" is written as 110100.

1.2 Converting from Binary to Decimal

How can we convert numbers from the binary system to the decimal system? Follow this example. We have 10110011. Note that this number is a sequence of 8 bits, so it is a byte. Add the weights as seen in the diagram. Add all weights that are associated with the bits that are equal to 1.

Weights	128	64	32	16	8	4	2	1
Number	1	0	0	1	0	1	1	0

Add 128 + 16 + 4 + 2. This amounts to 150.

Sometimes we use subscripts to indicate whether a number is binary of decimal as shown here:

A binary number: 10010110_2
A decimal number: 150_{10}

The method we have seen here is called the positional notation method.

1.2.1 Exercise

Convert the following binary numbers to decimal.

a. 11001
b. 11011101
c. 1110010

1.3 Converting from Decimal to Binary

Suppose we want to convert the decimal number 149 to binary. One way we can do this is by repeatedly dividing the number by 2 and then taking the remainder digits to get the solution. An example is shown here:

2	149		
2	74	remainder	1
2	37	remainder	0
2	18	remainder	1
2	9	remainder	0
2	4	remainder	1
2	2	remainder	0
2	1	remainder	0
	0 Quotient after dividing by 2	remainder	1 Remainder after dividing by 2

After performing this process you have to read the remainders FROM BOTTOM TO TOP. This gives us that the decimal number 149 is equal to the binary number 10010101.

1.3.1 Exercise

Convert the following decimal numbers to binary.

a. 108
b. 53
c. 74

1.4 Number Bases

Positional number systems depend on a special number called a base. A positional number system (like the decimal and binary systems) is such that a digit represents a number according to the position where it is e.g. in the number 2725 the first '2' (from the left) represents 2000 while the second '2' represents 20.

Note that:
$2725 = 2x1000 + 7x100 + 2x10 + 5 = 2x\mathbf{10^3} + 7x\mathbf{10^2} + 2x\mathbf{10^1} + 5x\mathbf{10^0}$.

Therefore we say that the base number of the decimal system is 10.

Working on the same principles we can show that the base number of the binary system is 2.

Let us investigate the number 10010110_2.
$10010110 = 1x128 + 0x64 + 0x32 + 1x16 + 0x8 + 1x4 + 1x2 + 0x1.$
$10010110 = 1x2^7 + 0x2^6 + 0x2^5 + 1x2^4 + 0x2^3 + 1x2^2 + 1x2^1 + 0x2^0.$

Note also that the base number shows us how many different symbols are required to express a number i.e. in the decimal system we need 10 symbols (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9) while in the binary system we need 2 (0 and 1).

1.5 Adding and Subtracting Binary Numbers

The following example shows two numbers being added:

Carry							
First number	0	0	1	1	1	1	0
Second number	1	0	0	1	0	1	1
Result of addition							1

Step 1:
0 + 1 gives 1

Carry					1		
First number	0	0	1	1	1	1	0
Second number	1	0	0	1	0	1	1
Result of addition						0	1

Step 2:
1 + 1 gives 0 carry 1

Carry				1	1		
First number	0	0	1	1	1	1	0
Second number	1	0	0	1	0	1	1
Result of addition					0	0	1

Step 3:
1 + 0 + 1 gives 0
carry 1

Carry			1	1	1		
First number	0	0	1	1	1	1	0
Second number	1	0	0	1	0	1	1
Result of addition				1	0	0	1

Step 4:
1 + 1 + 1 gives 1
carry 1

Proceeding in this way we get:

30

75

105

Carry		1	1	1	1		
First number	0	0	1	1	1	1	0
Second number	1	0	0	1	0	1	1
Result of addition	1	1	0	1	0	0	1

1.5.1 Exercise in Addition

Add the following binary numbers. Then check your result by using decimal numbers.

a. 1010001 and 0101110
b. 1011011 and 1011100

The following example shows subtraction:

Borrow							
First number	1	0	1	1	0	1	1
Second number	0	1	0	1	1	1	0
Result of subtraction							1

Step 1:
1 – 0 gives 1

Borrow							
First number	1	0	1	1	0	1	1
Second number	0	1	0	1	1	1	0
Result of subtraction						0	1

Step 2:
1 – 1 gives 0

Borrow							
First number	1	0	1	1	0	1	1
Second number	0	1	0	1	1	1	0
Result of subtraction						0	1

Step 2:
1 – 1 gives 0

Borrow					2		
First number	1	0	1	~~1~~	0	1	1
Second number	0	1	0	1	1	1	0
Result of subtraction					1	0	1

Step 3:
0 - 1 means that we have to borrow

Proceeding in this way we get

Borrow		2		2	2		
First number	~~1~~	0	~~1~~	~~1~~	0	1	1
Second number	0	1	0	1	1	1	0
Result of subtraction	0	1	0	1	1	0	1

91

46

45

1.5.2 Exercise in Subtraction

Perform the following binary subtractions. Then check your results by using decimal numbers:

a. 10001101 – 00101100
b. 10000101 – 01010111

1.6 Hexadecimal Numbers

Hexadecimal numbers are based on the number 16. A hexadecimal number uses 16 symbols which are 0, 1, 2 … 8, 9, A, B, C, D, E and F. A stands for ten, B stands for eleven, C stands for twelve etc. The term 'hexadecimal' is shortened to 'hex'.

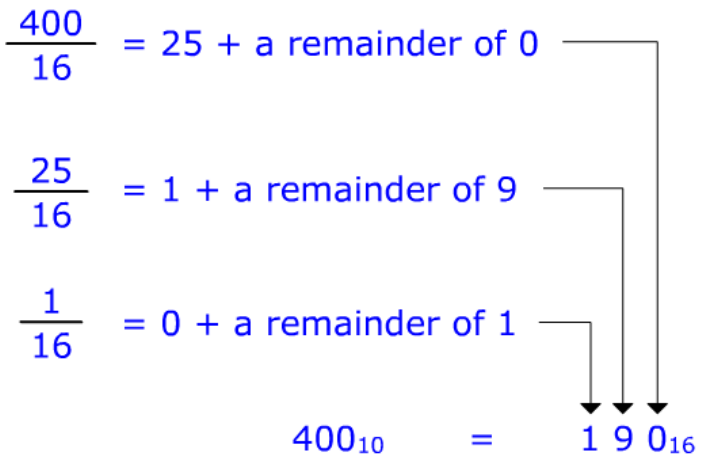
Why are hexadecimal numbers used? Since large binary numbers use a long sequence of ones and zeros, hexadecimal numbers can be used as substitutes. They are much shorter than binary numbers.

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

An example of a hex (hexadecimal) number: 3C90B.

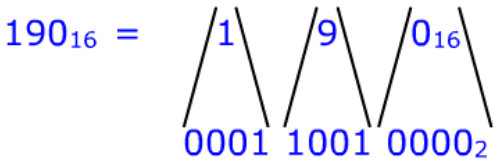
1.7 Conversion from decimal to hexadecimal

The following example shows how the decimal number 400 is converted to hex.



1.8 Conversion from Hex to Binary

In the following example the hex number 190 is converted to binary.



Leading zeros, in binary numbers, may be omitted:

$$000110010000_2 = 110010000_2$$

1.9 Conversion from binary to hex

It is easy to convert from an integer binary number to hex. This is accomplished by the following two steps:

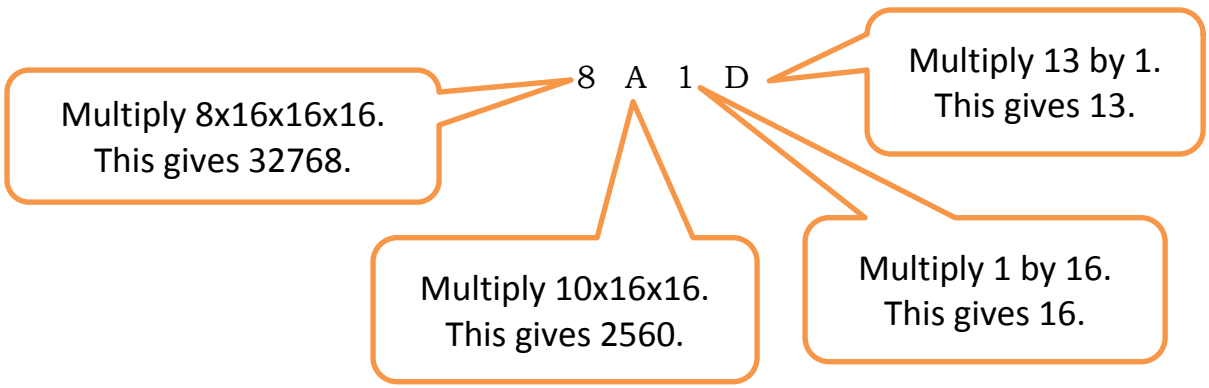
1. Break the binary number into 4-bit sections from the LSB (least significant bit) to the MSB (most significant bit).
2. Convert the 4-bit binary number to its Hex equivalent.

The following example shows how the binary value 1010111110110010 is converted to hex.

1010	1111	1011	0010
A	F	B	2

1.10 Conversion from hex to decimal

Let us convert 8A1D to decimal.



Finally add 32768 + 2560 + 16 + 13. This gives 35357.

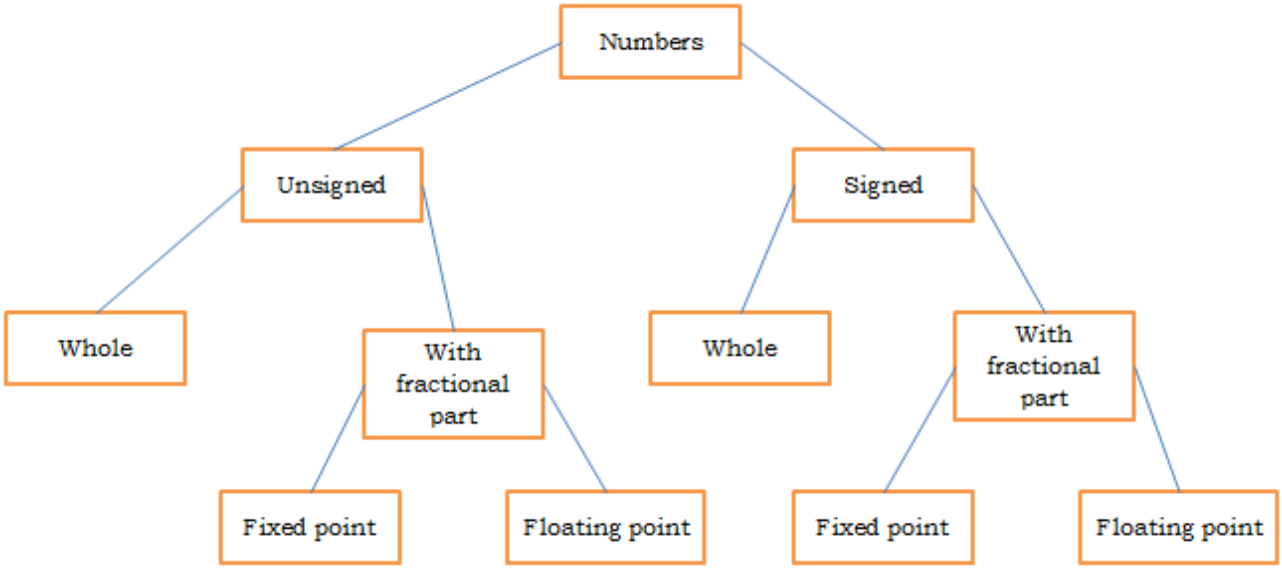
1.10.1 Exercise

Perform the following conversions:

a. Convert 10001110101_2 to hex.
b. Convert 20154_{10} to hex.
c. Convert $3F7A_{16}$ to decimal.
d. Convert $5BA_{16}$ to binary.

1.11 Classification of numbers

Numbers can be classified as shown in the following tree structure. Unsigned numbers refer to a range of numbers where all numbers are positive while signed numbers refer to a range of numbers with numbers that are both positive and negative.



Consider three bits. There are eight different combinations of ones and zeros (these are 000, 001, 010, 011, 100, 101, 110 and 111). These eight combinations can be made to represent the range of numbers from 0 to 7 or the numbers from -4 to 3. We say that the range "from 0 to 7" is unsigned and the range "from -4 to 3" is signed.

1.12 Sign and Magnitude

In the Sign and Magnitude representation the leftmost bit is reserved to represent the sign. 1 indicates negative and 0 indicates positive. Let us take an example with 8 bits.

sign	64	32	16	8	4	2	1	
0	1	0	0	1	1	0	1	77_{10}
1	1	0	0	1	1	0	1	-77_{10}

Now let us consider 4 bits. All the numbers are represented in the following table.

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

Note that the value zero is repeated twice. The range of values is between -7 and 7. If we had one byte the minimum value would be 11111111 (-127_{10}) and the maximum would be 01111111 (127_{10}).

Apart from the repeated zero another problem with sign and magnitude representation is that it is not good for computation e.g. making an addition of two numbers etc.

1.13 Two's Complement

Another representation scheme to represent both positive and negative numbers is called two's complement. It is now the nearly universal way of doing this. Integers are represented in a fixed number of bits. Both positive and negative integers can be represented.

How to Construct the Negative of an Integer in Two's Complement:

Start with an N-bit representation of an integer.

To calculate the N-bit representation of the negative integer:

1. Change every 0 to 1 and every 1 to 0).
2. Add one.

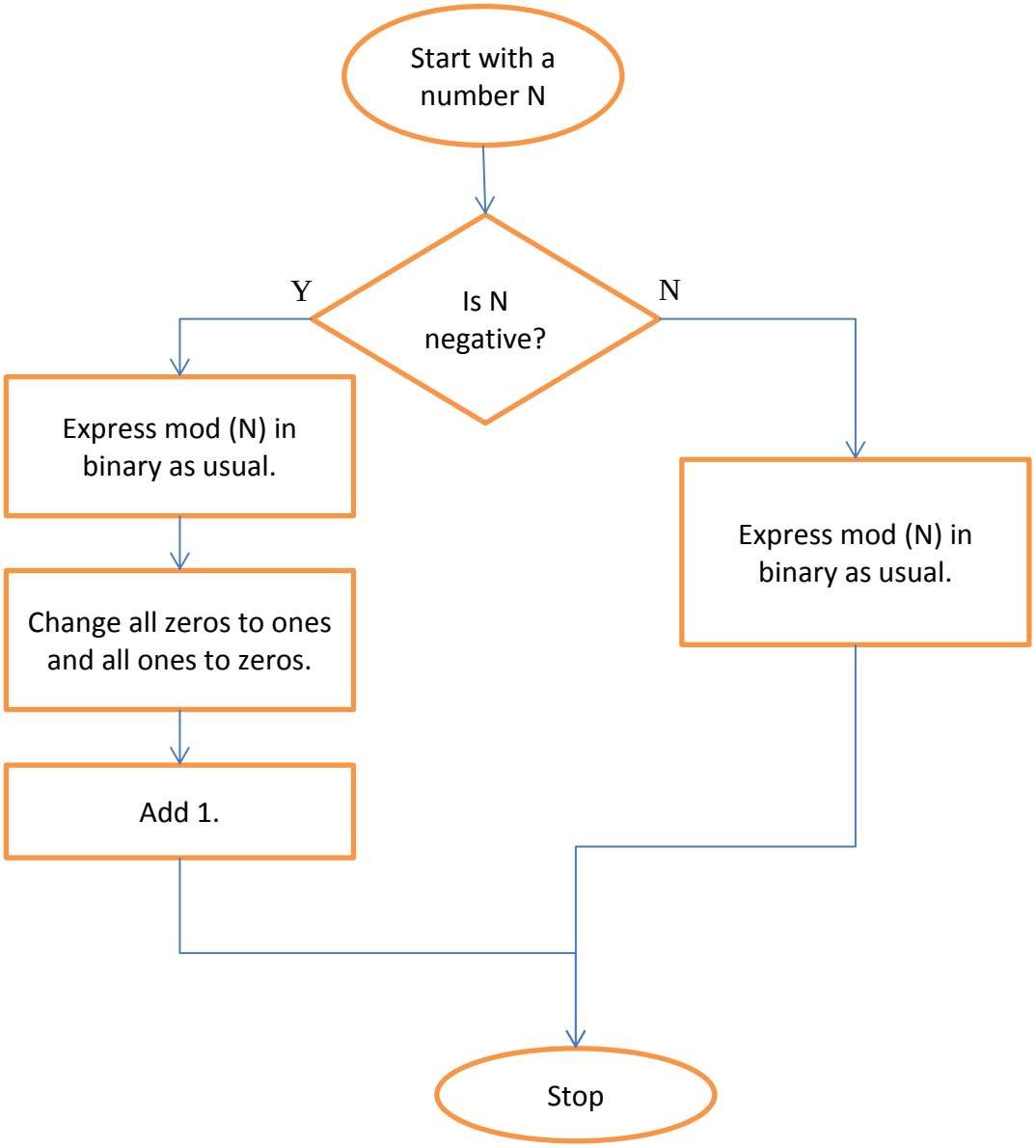
Suppose we have one byte to represent a number, say -87_{10}, in two's complement. This would be done in the following steps:

(1) Write 87_{10} in binary. This gives 01010111.

(2) Change all ones to zeros and all zeros to ones. This gives 10101000 (this number is called one's complement)
(3) Add 1. 10101000 + 1 gives 10101001.

Therefore $-87_{10} = 10101001_2$ (in two's complement).

To express a decimal number is two's complement follow the following algorithm:



1.13.1 Exercise

Express the following decimal numbers both in sign-and-magnitude and in two's complement (assume one byte of space): (a) 43 (b) -99 (c) -106

1.13.2 Faster Computation of Two's Complement

To work out the two's complement of a number do the following:

1. Start from the right hand side and move to the left hand side by considering all the bits.
2. If a bit is equal to zero write a zero, move to the next bit on the left and repeat step 2. If it is equal to one write a 1 and go to step 3.
3. From the next digit on the left to the leftmost digit convert all bits from 0 to 1 and from 1 to 0.

Example 1

Express 57_{10} in two's complement.

$57_{10} = 00111001_2$

Example 2

Express -104_{10} in two's complement.

$104_{10} = 01101000_2$
$-104_{10} = 10011000_2$

1.13.3 Maximum and Minimum Numbers

If we consider one byte:

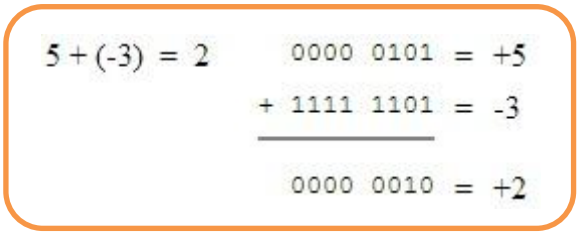
- The maximum number we can represent (in two's complement) is 01111111
- The minimum number is 10000000

These are equivalent to 127 and -128. So the range of values is [-128, 127]

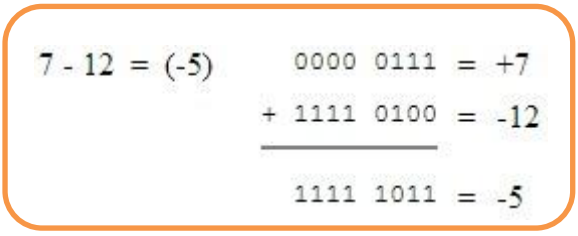
1.13.4 Subtracting using two's complement and additional

Suppose we want to calculate 5-3. We follow these steps:

1. 5 is converted to binary. Call this f.
2. 3 is converted to binary (two's complement). Call this t.
3. Add f and t but ignore the overflow bit.
4. You have the result.



Another example:



1.13.5 Exercise

Perform the following calculations using two's complement arithmetic. All numbers are memorized in one byte.

(a) 99 – 56, (b) 85 – 106, (c) -25 – 30

1.14 Fractions

Suppose we know that the number 1011.0110_2 is unsigned, what is its conversion in decimal?

$$1011.0110_2 = 1x2^3 + 0x2^2 + 1x2^1 + 0x2^0 + 0x2^{-1} + 1x2^{-2} + 1x2^{-3} + 0x2^{-4}$$
$$= 8 + 0 + 2 + 1 + 0 + 0.25 + 0.125 + 0$$
$$= 11.375_{10}$$

In this representation:

- the maximum number is 1111.1111 (15.9375), and
- the minimum number is 0000.0000

This representation is called fixed-point representation

1.15 Maximum and minimum values

This table shows some maximum and minimum values (on one byte).

	Minimum		Maximum	
	Decimal	Binary	Decimal	Binary
Unsigned integer (i.e. whole number)	0	00000000	255	11111111
Signed integer (sign and magnitude)	-127	11111111	127	01111111
Signed integer (two's complement)	-128	10000000	127	01111111
Unsigned fractional number with fixed point representation (three decimal places)	0	00000.000	31.875	11111.111
Signed fractional number with fixed point representation (three decimal places, sign and magnitude)	-15.875	11111.111	15.875	01111.111
Unsigned fractional number with fixed point representation (three decimal places, two's complement)	-16	10000.000	15.875	01111.111

1.16 Floating-point representation

A number can be represented in two parts: mantissa and exponent. This is what we do in numbers in standard form e.g. instead of writing 436000000000000 we write 4.36×10^{14}. Likewise instead of writing 0.00000029 we write 2.9×10^{-7}. In the first example 4.36 is called the **mantissa** and 14 is called the **exponent**.

In binary numbers we can use the same principle. However the base 10 is replaced by 2. e.g. 110100000000000_2 is expressed as $1.101_2 \times 2^{14}$. In practice if we had 8 bits to represent the mantissa and 8 bits to represent the exponent the number would be represented as 11010000 (mantissa) and 00001110 (exponent) with the added assumption that there is a binary point in the mantissa after the leftmost bit.

Example 1

Find the decimal value of the number with a mantissa equal to 01011000 and an exponent of 0010011. Both the mantissa and the exponent are expressed in two's complement where the mantissa has a fractional part (there is an assumed binary point between the first and second bit from the left) while the exponent is an integer.

$0.1011000_2 = 0.5 + 0.125 + 0.0625 = 0.6875_{10}$
$0010011_2 = 19_{10}$
Therefore number = $0.6575 \times 2^{19} = 360448$

Example 2

Conditions same as in example 1 but mantissa is equal to 01010100 and exponent equal to 11110100.

$0.1010100_2 = 0.65625_{10}$
$11110100_2 = -12_{10}$
Therefore number = $0.65625 \times 10^{-12} = 0.000000000000065625$

1.16.1 Exercise

Given the conditions in examples 1 and 2 find the decimal value of the following binary floating-point numbers:

(a) Mantissa = 01110000, exponent = 00010110
(b) Mantissa = 01011000, exponent = 11010000
(c) Mantissa = 10001100, exponent = 00110101
(d) Mantissa = 10110010, exponent = 11100100

1.16.2 Normalisation

Let us consider an example where a floating point number has a mantissa of 5 bits and an exponent of 3 bits. Let us assume that both numbers have a two's

complement representation. Also let us assume that between the two leftmost bits of the mantissa there is a fractional point.

These are some examples of numbers:

Mantissa						Exponent			Mantissa	Exponent	Value
0	.	1	0	1	0	0	1	1	0.625	3	5
0	.	0	0	1	1	0	0	1	0.1875	1	0.375
0	.	0	1	1	0	1	0	1	0.375	-3	0.046875
1	.	0	0	1	1	0	1	0	-0.8125	2	-3.25
1	.	1	1	1	0	1	0	1	-0.125	-3	-0.01563

Out of these numbers the first and the fifth are in a form called Normalised. In these the mantissa starts with 0.1 or 1.0.

Note that:

- If the mantissa starts with 0.1 then the mantissa is a value greater than or equal to 0.5 but less than 1.
- If the mantissa starts with 1.0 then the mantissa is a value greater than or equal to -1 but less than 0.

The following table shows the largest and smallest numbers (not necessarily normalised) of the scheme described above:

Mantissa					Exponent			Mantissa	Exponent	Value	
0	.	1	1	1	1	0	1	1	0.9375	3	7.5	Largest value
0	.	0	0	0	0	-	-	-	0		0	Smallest non-negative value
0	.	0	0	0	1	1	0	0	0.0625	-4	0.00390625	Smallest positive (not zero) value
1	.	0	0	0	0	0	1	1	-1	3	-8	Smallest value
1	.	1	1	1	1	1	0	0	-0.0625	-4	-0.00390625	Largest negative value

The following table shows the largest and smallest numbers in the normalised form:

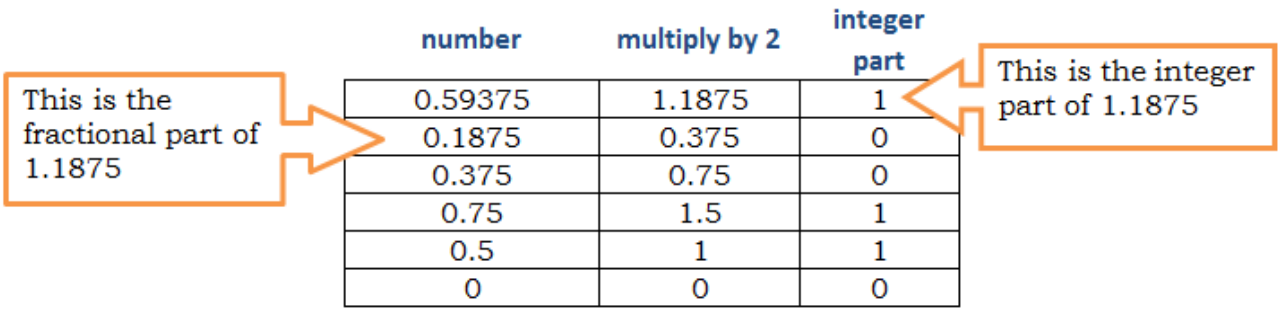
Mantissa					Exponent			Mantissa	Exponent	Value	
0	.	1	1	1	1	0	1	1	0.9375	3	7.5	Largest value
0	.	1	0	0	0	1	0	0	0.5	-4	0.03125	Smallest positive value
1	.	0	1	1	1	1	0	0	-0.5625	-4	-0.03515625	Largest negative value
1	.	0	0	0	0	0	1	1	-1	3	-8	Smallest value

1.16.3 Converting a decimal number into a normalised floating-point binary number

1.16.3.1 Example 1

Convert 55.59375_{10} in floating-point binary such that (i) both mantissa and exponent are in two's complement and (ii) mantissa has 16 bits and exponent has 8 bits.

Convert 55.59375_{10} to binary.
$55_{10} = 110111_2$
$0.59375_{10} = 0.10011_2$ (this is calculated by the method shown in the following table)



number	multiply by 2	integer part
0.59375	1.1875	1
0.1875	0.375	0
0.375	0.75	0
0.75	1.5	1
0.5	1	1
0	0	0

Therefore $55.625_{10} = 110111.0000.1011_2$
Normalise this number
$110111.10011_2 = 0.11011110011_2 \times 2^6$
Solution:
mantissa = 0.11011110011 (16 bits)
exponent = 00000110 (8 bits)

1.16.3.2 Example 2

Convert 0.046875_{10} in floating-point binary such that (i) both mantissa and exponent are in two's complement and (ii) mantissa has 16 bits and exponent has 8 bits.

Convert 0.046875_{10} to binary.
$0.046875_{10} = 0.000011_2$
Normalise this number
$0.000011_2 = 0.11_2 \times 2^{-4}$
Solution:
mantissa = 0.110000000000000 (16 bits)
$4_{10} = 00000100_2$
$-4_{10} = 11111100_2$
exponent = 11111100 (8 bits)

1.16.3.3 Example 3

Convert -21. 0.09375_{10} in floating-point binary such that (i) both mantissa and exponent are in two's complement and (ii) mantissa has 16 bits and exponent has 8 bits.

Convert 21.09375_{10} to binary.
$21.09375_{10} = 10101.00011_2$
Normalise this number
$10101.00011_2 = 0.1010100011_2 \times 2^5$
mantissa = - 0.10101000110000 (16 bits)
find the two's complement of the mantissa
mantissa = 1.01010111010000 (16 bits)
exponent = 00000101 (8 bits)

1.16.3.4 Example 4

Convert -0.15625_{10} in floating-point binary such that (i) both mantissa and exponent are in two's complement and (ii) mantissa has 16 bits and exponent has 8 bits.

Convert 0.15625_{10} to binary.
$-0.15625_{10} = -0.00101_2$
Normalise this number $-0.00101_2 = -0.101_2 \times 2^{-3}$
mantissa = - 0.1010000000000000 (16 bits)
exponent = - 00000011 (8 bits)
Find two's complement of both mantissa and exponent
mantissa = 1.0110000000000000 (16 bits)
exponent = 11111101 (8 bits)

1.16.3.5 Exercise

Convert the following decimal numbers in floating-point binary such that (i) both mantissa and exponent are in two's complement and (ii) mantissa has 16 bits and exponent has 8 bits.

a) 49.1875
b) 0.046875
c) – 37.0625
d) – 0.15625

1.17 Overflow and Underflow

Overflow is said to occur when the number that one gets as a result of some arithmetic operation is larger in magnitude than the largest number one can represent in the given location. What if the operation of two numbers yields a number which is smaller than the smallest representable number? Is this underflow? Note that this may still be called an overflow since, we are only concerned with the magnitude.

If a number's magnitude is too small to represent in the given location we talk of underflow.

1.18 Codes

1.18.1 ASCII Code

ASCII stands for American Standard Code for Information Interchange. ASCII is a code for representing English characters as numbers, with each letter assigned a number from 0 to 127. For example, the ASCII code for uppercase M is 77. It also represents control codes (like 'backspace' and 'line feed' which causes a printer to advance its paper) as numbers. Most computers use ASCII codes to represent text, which makes it possible to transfer data from one computer to another.

Dec	Hex	Char	Dec	Hex	Char
32	20	Space	64	40	@
33	21	!	65	41	A
34	22	"	66	42	B
35	23	#	67	43	C
36	24	$	68	44	D
37	25	%	69	45	E

Text files stored in ASCII format are also called ASCII files. Text editors and word processors are usually capable of storing data in ASCII format, although ASCII format is not always the default storage format. Most data files, particularly if they contain numeric data, are not stored in ASCII format. Executable programs are never stored in ASCII format.

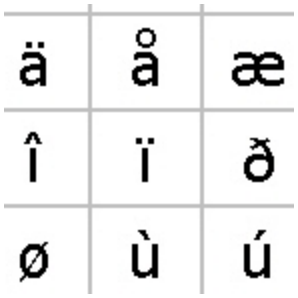
The standard ASCII character set uses just 7 bits for each character. There are several larger character sets that use 8 bits, which gives them 128 additional characters. The extra characters are used to represent non-English characters, graphics symbols, and mathematical symbols. Several companies and organizations have proposed extensions for these 128 characters. The DOS operating system uses a superset of ASCII called extended ASCII or high ASCII. A more universal standard is the ISO Latin 1 set of characters, which is used by many operating systems, as well as Web browsers.

Another set of codes that is used on large IBM computers is EBCDIC.

1.18.2 EBCDIC

EBCDIC stands for Extended Binary-Coded Decimal Interchange Code. EBCDIC is an IBM code for representing characters as numbers. Although it is widely used on large IBM computers, most other computers, including PCs and Macintoshes, use ASCII codes.

1.18.3 Unicode



Unicode is a standard for representing characters as integers. Unlike ASCII, which uses 7 bits for each character, Unicode uses 16 bits, which means that it can represent more than 65,000 unique characters. This is a bit of overkill for English and Western-European languages, but it is necessary for some other languages, such as Greek, Chinese and Japanese. Many analysts believe that as the software industry becomes increasingly global, Unicode will eventually supplant ASCII as the standard character coding format.

1.18.4 Gray Code

The reflected binary code, also known as Gray code after Frank Gray, is a binary numeral system where two successive values differ in only one bit. The reflected binary code was originally designed to prevent spurious output from electromechanical switches.

Today, Gray codes are widely used to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems. Bell Labs researcher Frank Gray introduced the term “Reflected Binary Code” in his 1947 patent application.

Decimal number	Gray number	Binary number
0	000	000
1	001	001
2	011	010
3	010	011
4	110	100
5	111	101
6	101	110
7	100	111

Notice that state 7 can roll over to state 0 with only one switch change. This is called the "cyclic" property of a Gray code.

Gray was most interested to minimize the effect of error in the conversion of analogue signals to digital; his codes are still used today for this purpose, and others. Gray codes are also used in labelling the axes of Karnaugh maps.

Applications of Gray code:

- error correction
- particular logic circuits to prevent transient states.

1.18.5 Binary Coded Decimal

Abbreviated as BCD, binary-coded decimal is a format for representing decimal numbers (integers) in which each digit is represented by four bits (a nibble). For example, the number 375 would be represented as:

0011 0111 0101

One advantage of BCD over binary representations is that there is no limit to the size of a number. To add another digit, you just need to add a new 4-bit sequence. In contrast, numbers represented in binary format are generally limited to the largest number that can be represented by 8, 16, 32 or 64 bits.