APPENDIX A

Some Number Systems

A.1 Binary, Decimal, and Hexadecimal

The number system we are most familiar with, decimal, makes use of 10 different symbols to represent numbers, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Each of these symbols represents a number, and we make larger numbers by using groups of symbols. In this case the digit most to the right represents units, the next represents tens, the next hundreds, and so on. For example, the number 249, shown in Fig. A.1, is evaluated by adding the values in each position:

2 hundreds + 4 tens + 9 units = 249

OR

$2 \times 10^2 + 4 \times 10^1 + 9 \times 10^0 = 249$

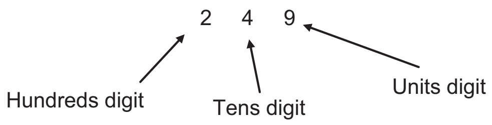

Figure A.1
The decimal number 249.

The *base* or *radix* of the decimal system, just described, is ten. We almost certainly count in the decimal system due to the accident of having ten fingers and thumbs on our hands. There is nothing intrinsically correct or superior about it. It is quite possible to count in other bases, and the world of digital computing almost forces us to do this.

The binary counting system has a base or radix of 2. It therefore uses just two symbols, normally 0 and 1. These are called binary digits or bits. Numbers are made up of groups of digits. The value each digit represents again depends upon its position in the number. Therefore the 4-bit number 1101, shown in Fig. A.2, is interpreted as:

$(1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 8 + 4 + 1 = 13$

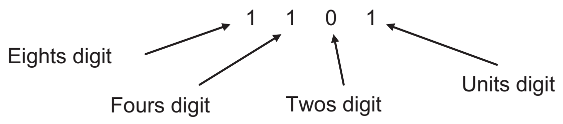

Figure A.2
The binary number 1101.

Similarly, the value 0110 binary = 6 decimal. Note that we refer to the units digit as "bit 0" or the least significant bit (LSB). The two's digit is called "bit 1" and so on, up to the most significant bit (MSB).

We are obviously interested in binary representation of numbers, because that is how digital machines perform mathematical operations. But sometimes there are just too many 0s and 1s to keep track of. We often group bits in 4s, so we often look at 4-bit numbers, 8-bit numbers, 12, 16, 20, 24, 28, and 32-bit numbers.

A single *byte* is made up of 8 bits. With 1 byte we can count up to 255 in decimal, for example:

$$0000\ 0000 \text{ binary} = 0 \text{ decimal}$$

$$1111\ 1111 \text{ binary} = 255 \text{ decimal}$$

$$1001\ 1100 \text{ binary} = 128 + 16 + 8 + 4 = 156$$

The range of 0 to 255, offered by a single byte, is very great, however. To perform mathematical calculations to a high accuracy we need to work with 16, 24, or 32-bit systems. With 16 bits we can count or resolve up to 65,535, for example:

$$1111\ 1111\ 1111\ 1111 \text{ binary} = 65535 \text{ decimal}$$

$$0111\ 0111\ 0111\ 0110 \text{ binary} = 30582 \text{ decimal}$$

Working with large binary numbers is something that is not easy for a human; we just cannot absorb all those 1s and 0s. A very convenient alternative is to use hexadecimal, which works to base 16. This means that we can now count up to 15 (decimal) before there is an overflow, and each new overflow column is an increased power of 16. Consider the hexadecimal number 371 as shown in Fig. A.3.

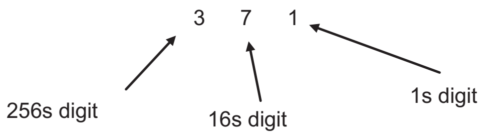

Figure A.3
Hexadecimal number system.

Here we see that 371 in hexadecimal is $(3 \times 256) + (7 \times 16) + 1 = 881$ in decimal.

Table A.1: 4-bit values in binary, hexadecimal, and decimal.

4-Bit Binary Number				Hexadecimal Equivalent	Decimal Equivalent
0	0	0	0	0x0	0
0	0	0	1	0x1	1
0	0	1	0	0x2	2
0	0	1	1	0x3	3
0	1	0	0	0x4	4
0	1	0	1	0x5	5
0	1	1	0	0x6	6
0	1	1	1	0x7	7
1	0	0	0	0x8	8
1	0	0	1	0x9	9
1	0	1	0	0xA	10
1	0	1	1	0xB	11
1	1	0	0	0xC	12
1	1	0	1	0xD	13
1	1	1	0	0xE	14
1	1	1	1	0xF	15

Now we need to represent 16 numbers with just a single digit. We use the decimal digits for numbers 0 to 9, but to represent numbers 10 to 15 we use letters: "A" or "a" is adopted to represent decimal 10. Similarly "B" or "b" = 11, "C" = 12, "D" = 13, "E" = 14, "F" = 15. As convention, we put "0x" before a hexadecimal number; this means that all the bits before the number are zero or clear. For example, 0xE5 = $(14 \times 16) + 5 = 229$. Table A.1 shows the equivalent 4-bit values in both binary, hexadecimal, and decimal.

The great advantage of hexadecimal numbers is that we can represent 4-bit numbers with a single digit. An 8-bit number is represented with two hexadecimal digits, and 16-bit numbers with four digits. You can then see that by individually looking at groups of 4 bits, we can easily generate the hexadecimal equivalent, as with the following examples:

255 decimal	=	1111 1111	=	0xFF
156 decimal	=	1001 1100	=	0x9C
65535 decimal	=	1111 1111 1111 1111	=	0xFFFF
30582 decimal	=	0111 0111 0111 0110	=	0x7776

While we correctly use the two values of 0 and 1 in all binary numbers above, it's worth noting that different terminology is sometimes used when we apply electronic circuits to

Table A.2: Some terminology for logic levels.

Logic 0	Logic 1
0	1
Off	On
Low	High
Clear	Set
Open	Closed

represent these numbers. This is done particularly by those who are thinking more in terms of the circuit, than of the numbers. This terminology is shown in Table A.2.

A.2 Representation of Negative Numbers—Two's Complement

Simple binary numbers allow only the representation of unsigned numbers, which under normal circumstances are considered to be positive. Yet we must have a way of representing negative numbers as well. A simple way of doing this is by offsetting the available range of numbers. We do this by coding the largest anticipated negative number as zero and counting up from there. In the 8-bit range, with symmetrical offset, we can represent −128 as 00000000, 1000000 then represents zero, and 11111111 represents +127. This method of coding is called *offset binary* and is illustrated in the Table A.3. It is used on occasions (for example, in analog-to-digital converter outputs), but its usefulness is limited as it is not easy to do arithmetic with.

Table A.3: Two's complement and offset binary.

Two's Complement	Decimal	Offset Binary
0111 1111	+127	1111 1111
0111 1110	+126	1111 1110
	:	
	:	
0000 0001	+1	1000 0001
0000 0000	0	1000 0000
1111 1111	−1	0111 1111
1111 1110	−2	0111 1110
	:	
	:	
1000 0010	−126	0000 0010
1000 0001	−127	0000 0001
1000 0000	−128	0000 0000

Let us consider an alternative approach. Suppose we took an 8-bit binary down counter, and clocked it from any value down to, and then below, zero. We would get this sequence of numbers:

Binary	Decimal
0000 0101	5
0000 0100	4
0000 0011	3
0000 0010	2
0000 0001	1
0000 0000	0
1111 1111	−1?
1111 1110	−2?
1111 1101	−3?
1111 1100	−4?
1111 1011	−5?

This gives a possible means of representing negative numbers—effectively we subtract the magnitude of the negative number from zero, within the limits of the 8-bit number, or whatever other size is in use. This representation is called *two's complement*. It can be shown that using two's complement leads to correct results when simple binary addition and subtraction is applied. Two's complement notation can be applied to binary words of any size.

The two's complement of an n-bit number is found by subtracting it from 2^n, that's where the terminology comes from. Rather than doing this error-prone subtraction, an easier way of reaching the same result is to complement (i.e., change 1 to 0, and 0 to 1) all the bits of the positive number, and then add 1. Hence to find −5 we follow the procedure:

original number		**complement all**		**add one**
0000 0101 (+5)	->	1111 1010	->	1111 1011 (-5 in 2's comp.)

To convert back, simply subtract one and complement again. Note that the MSB of a two's complement number acts as a "sign bit," 1 for negative, 0 for positive. The 8-bit binary range, shown both for two's complement and offset binary, appears in Table A.3.

A.2.1 Range of Two's Complement

In general, the range of an n-bit two's complement number is from $-2^{(n-1)}$ to $+\{2^{(n-1)} - 1\}$. Table A.4 summarizes the ranges available for some commonly used values of n.

Table A.4: Number ranges for differing word sizes.

Number of Bits	Unsigned Binary	Two's Complement
8	0 to 255	−128 to +127
12	0 to 4095	−2048 to +2047
16	0 to 65,535	−32,768 to +32,767
24	0 to 16,777,215	−8,388,608 to +8,388,607
32	0 to 4,294,967,295	−2,147,483,648 to +2,147,483,647

A.3 Floating Point Number Representation

Numbers described so far appear to be all integers, we haven't been able to represent any sort of fractional number, and their range is limited by the size of the binary word representing them. Suppose we need to represent really large or small numbers? Another way of expressing a number, which greatly widens the range, is called *floating point* representation.

In general a number can be represented as $a \times r^e$, where a is the *mantissa*, r is the radix, and e is the exponent. This is sometimes called scientific notation. For example the decimal number 12.3 can be represented as

$$\begin{gathered} 1.23 \times 10^1 \text{ or} \\ 0.123 \times 10^2 \text{ or} \\ 12.3 \times 10^0 \text{ or} \\ 123 \times 10^{-1} \text{ or} \\ 1230 \times 10^{-2}. \end{gathered}$$

Floating point notation adapts and applies scientific notation to the computer world. The name is derived from the way the binary point can be allowed to float, by adjusting the value of the exponent, to make best use of the bits available in the mantissa. Standard formats exist for representing numbers by their sign, mantissa and exponent, and a host of hardware and software techniques exist to process numbers represented in this way. Their disadvantage lies in their greater complexity, and hence usually slower processing speed and higher cost. For flexible use of numbers in the computing world they are, however, essential.

The most widely recognized and used format is the *IEEE Standard for Floating-Point Arithmetic* (known as IEEE 754). In single precision form, this makes use of 32-bit representation for a number, with 23 bits for the mantissa, 8 bits for the exponent, and a sign bit, as seen in Fig. A.4. The binary point is assumed to be just to the left of the MSB of the mantissa. A further bit, always 1 for a nonzero number, is added to the mantissa,

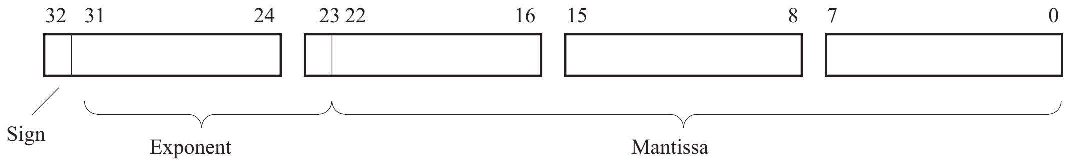

Figure A.4
IEEE 754 32-bit floating point format.

making it effectively a 24-bit number. Zero is represented by 4 zero bytes. The number 127 is subtracted from the exponent, leading to an effective range of exponents from -126 to $+127$. Exponent 255 (leading to 128, when 127 is subtracted) is reserved to represent infinity. The value of a number represented in this format is then

$$(-1)^{\text{sign}} \times 2^{(\text{exponent}-127)} \times 1.\text{mantissa}$$

This allows number representation in the range:

$$\pm 1.175494 \times 10^{-38} \text{ to } \pm 3.402823 \times 10^{+38}.$$