Computer Fundamentals: Lecture 2 Reading

Before the second lecture, please make sure you understand the material summarised here. Some of you will be very familiar with it and will not need to spend much time on it; others may not have covered it before and will need to spend an hour or two in the library to get up to speed. There are also many resources online that will help—Google is your friend here.

You should check your understanding by attempting the first section on the examples sheet before the lecture.

Positive Integers and Bases

What is 7+3? In most contexts you would answer "10", but actually the questions is underspecified since the *base* of the numbers was not given. By base we mean the number of digits in the number system. We're all most comfortable with base-10 (*decimal*), which uses the ten digits 0–9.[1] In general, for any k-digit decimal number $N_{10} = a_{k-1}a_{k-2}...a_1a_0$, we have:

$$N_{10} = \sum_{0}^{k-1} a_i 10^i = a_o + a_1.10^1 + a_2.10^2 + ... \quad (1)$$

So 123=3+2.10^1+1.10^2. For a general base, b, this trivially extends:

$$N_b = \sum_{0}^{k-1} a_i b^i = a_0 + a_1.b + a_2.b^2 + ... \quad (2)$$

Base-2 (*binary*, b=2)—with the binary digits (bits) 0 and 1 only—is also common. In fact, there is something 'special' about binary since you cannot communicate information with anything less than two symbols. So, binary is the smallest number of useful symbols for encoding information, and hence the bit is used as the unit of information.

Binary is also the easiest to implement in hardware (since it's just one switch per bit) and so the vast majority of computers are built on it. Usually we like our numbers to be displayed in decimal, however, which is mildly irritating because ten can't be represented efficiently with an integer number of bits (see examples sheet).

One of the downsides to binary is that relatively small numbers require lots of digits that are harder for the human brain to recall and handle. e.g. 1234_{10}=10011010010_2. Instead, we prefer to use higher bases.

In computing, we prefer power-of-two bases, since these correspond to straightforward partitioning of a binary number. Base-8 (*octal*) is easy to deal with because it's the closest power-of-two to 10. It uses the symbols 0–7 inclusive and effectively groups a binary number into 3-bit chunks:
1234_{10}=10011010010_2=$010\ 011\ 010\ 010_2$=2322_8

Even more common is to group them into chunks of four to get a base-16 representation (*hexadecimal* or just "hex"). Here we have a problem since we need 16 symbols, but only have 10 in our vocabulary (0–9). We use the alphabet to extend our symbols, which range from 0–9 followed by A–F (i.e. 10_{10}=A_{16}, 15_{10}=F_{16}). So now:
1234_{10}=10011010010_2=$0100\ 1101\ 0010_2$=$4D2_{16}$

Because hex is so commonly used and subscripts aren't easy to add in code, we preface a hex number with "0x", so $4D2_{16}$ is 0x4D2. For some reason there isn't a uniformly-adopted prefix for octal—you may see 123_8 written as 0o123, 0123, o123, q123, or something else entirely! Thankfully, hex is the preferred choice these days.

Bits, Bytes, Words, etc

When working with computers, you come across a variety of units for data. The *bit* (BInary digiT) is probably the only one that is totally unambiguous—it is either 0 or 1, and we know that it represents a fundamental unit of data.

A *byte* was historically unstandardised, and was the number of bits used to represent a single character on the system in use. However, there is now a de-facto standard of 8 bits in a byte and it is common for a byte to be assumed to be 8 bits. As such, it can take values between 0 and 255 or 0x00 to 0xFF in hex.

A *nibble* or *nybble* is usually defined as half a byte, which is de-facto 4-bits or one hex digit.

[1]It's an interesting question why we favour base-10 in our society. The most oft-quoted answer is that we have ten digits on two hands, but there isn't any particular evidence for this. All we know is that many early cultures seem to have used decimal systems.

A *word* is platform-specific and is the number of bits used by the processor to represent a single instruction. Most desktop machines use either 32-bit or 64-bit words; some embedded/mobile devices will use 8-bit or 16-bit words.

We often deal with large values and need prefix-multipliers. This should be straightforward except that SI multipliers are designed for base-10, not base-2. This means that each new prefix ends up being a multiple of 1,024 (2^{10}) and not 1,000. e.g. a kilobyte is usually 1,024 bytes and not 1,000 bytes. Usually the context is obvious so the difference is irrelevant. However, you will encounter people who insist on using the IEEE standard (introduced 10 years or so ago). This defines kibi, mebi- and gibi- as follows:

Name	Suffix	Size (bytes)
kilobyte	KB	1,000
kibibyte	KiB	1,024
megabyte	MB	1,000,000
mebibyte	MiB	1,048,576
gigabyte	GB	1,000,000,000
gibibyte	GiB	1,073,741,824

Most people just stick with kilo-, mega-, giga-, etc. Try not to get upset either way.

Binary Addition and Subtraction

The binary addition algorithm is an easier version of the standard decimal addition algorithm you all did at achool. Add the bits from right-to-left, carrying the most significant bit each time. E.g.

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

To do subtraction, you were probably taught to 'borrow' from the left where necessary. We do exactly the same in binary subtraction. When you came across $(a - b)$ and $a < b$ (i.e. a negative result) you were most likely taught to compute $(b - a)$ and add a negative sign. This works in binary too, although representing negative numbers in a machine ends up being more complicated—don't worry too much about that yet, since it will be covered in lecture 2. For now, just make sure you can perform addition and subtraction in binary on paper.