3 Hex numbers, machine code, assemblers, languages

3.1 Octal and hexadecimal numbers

Our decimal number system is derived from the human hand.

All computer data is stored as patterns of 0s and 1s. A 'bit' is a binary digit, i.e., 0 or 1, or an object which can take these values. There is a multiplicative effect, so that 8 bits combined together can take $2^8 = 256$ different values.

A **byte** is a group of 8 bits.

The binary string `01001000` represents

$$0 + 0 \times 2 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 + 0 \times 2^7 = 8 + 64 = 72$$

(that is, 72 in decimal, of course).

It is easy to list the binary strings of length 3 in ascending order:

```
000, 001, 010, 011,
100, 101, 110, 111
```

The rightmost bit is called the 'low-order bit.' The 'low order bit' changes most often; the next bit changes half as often; the high-order bit changes only once.

It is easy to convert a bitstring into an *octal string.* Simply put it in groups of 3, starting from the right. Thus

```
01001000
01 001 000
 1   1   0
```

On the other hand, interpreting 110 as an octal string we get

$$0 + 1 \times 8 + 1 \times 8^2 = 72$$

(again, 72 in decimal).

It is a coincidence that all the octal digits in `01 001 000` are 0 or 1, so it 'looks' like a binary number. To correct the ambiguity, one can use $(\ldots)_b$ to indicate 'to base b'.

Then without ambiguity

$$(01001000)_2 = (110)_8 = (72)_{10}$$

Octal numbers give a compact way to represent bitstrings. So do **hexadecimal numbers**, which are numbers to base 16. We need 16 **hex digits** to form hexadecimal numbers. One uses `a,b,c,d,e,f` (or `A,B,C,D,E,F`) for the digits ≥ 10. Every hex digit equals four binary digits. The hex digits convert to octal, binary, and decimal as follows

hex	octal	binary	decimal
0	0	0000	0
1	1	0001	1
2	2	0010	2
3	3	0011	3
4	4	0100	4
5	5	0101	5
6	6	0110	6
7	7	0111	7
8	10	1000	8
9	11	1001	9
a	12	1010	10
b	13	1011	11
c	14	1100	12
d	15	1101	13
e	16	1110	14
f	17	1111	15

There are procedures for addition, subtraction, multiplication, and division, in binary, octal, and hex. Addition is easy. For example (each calculation is 'staggered' from right to left to show the 'carries').

```
    binary    octal   hex    Decimal

  10100011     243     a3      163
 +11010101    +325    +d5     +213
----------    ----    ---     ----
        10      10      8      376
       10       7     17
      10       5      ---
     1         ----   i.e
    1          i.e.    c3
   1            243    +d5
 10            +325    ---
-----------    ----    178
That is         570
  10100011
 +11010101
-----------
 101111000
```

Multiplication and division in binary involve a fairly large number of very simple steps. Except for trivial cases, they are always 'long multiplication' and 'long division.' We shall not attempt them by hand.

3.2 Features of a computer

A computer has several components, including **Central memory, central processor, hard disc, and terminal (or monitor).**

Long-term data is on the hard disc; the central processor works on short-term data in the central memory.

Here is a C program

```
#include <stdio.h>

main()
{
  printf("Hello\n");
  printf("there\n");
}
```

Create a file `hello.c` containing the above lines, then run

```
gcc hello.c
```

This will create a file `a.out` which the computer can run as a program:

```
aturing% a.out
```

(The '`aturing% `' is a 'command-line prompt.')
will cause the message

```
Hello
there
```

to be written to the terminal.

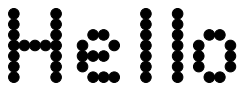
Question: what's the '`\n`' for?

`a.out` is in *machine code*. A computer accepts instructions in a very compact form called its *machine code.* A *machine program* (also called a 'binary' or 'executable') is a list of instructions in machine code. In the 1970s, with small microprocessors, it was common to write programs directly in machine code. Here is an example of machine code. Nowadays, most machine-code programs have thousands of lines like these. For example (I think that this tabulates the `a.out` file compiled from the above program, but it may be something different) on an Intel computer. The instructions are given in hex.

```
Memory    Machine
Address   instructions-----------------------------------
```

```
00000210  69 6e 5f 75 73 65 64 00  5f 5f 6c 69 62 63 5f 73
00000220  74 61 72 74 5f 6d 61 69  6e 00 47 4c 49 42 43 5f
00000230  32 2e 30 00 00 00 02 00  02 00 01 00 00 00 00 00
00000240  01 00 01 00 24 00 00 00  10 00 00 00 00 00 00 00
00000250  10 69 69 0d 00 00 02 00  56 00 00 00 00 00 00 00
00000260  d8 95 04 08 06 05 00 00  d0 95 04 08 07 01 00 00
00000270  d4 95 04 08 07 02 00 00  55 89 e5 83 ec 08 e8 61
00000280  00 00 00 e8 c8 00 00 00  e8 f3 01 00 00 c9 c3 00
00000290  ff 35 c8 95 04 08 ff 25  cc 95 04 08 00 00 00 00
000002a0  ff 25 d0 95 04 08 68 00  00 00 00 e9 e0 ff ff ff
000002b0  ff 25 d4 95 04 08 68 08  00 00 00 e9 d0 ff ff ff
000002c0  31 ed 5e 89 e1 83 e4 f0  50 54 52 68 20 84 04 08
000002d0  68 c0 83 04 08 51 56 68  84 83 04 08 e8 bf ff ff
```

(3.1) Although letters on the terminal look like ordinary newsprint, say, under close inspection the letters spelling `Hello` are just patterns of dots, something like



How are these letters stored on a computer? they could be stored as 7×5 patterns of zeroes and 1s, where 0 means 'no dot' and 1 means 'dot.' This would require 35 bits per letter. Instead, all characters are stored as 8-bit patterns under an internationally agreed code, the ASCII code. To learn more, type

```
man ascii
```

ASCII code for H is 01001000, for e is 01100101, and so on.

Question. The ASCII code for H has octal value 110. The ASCII code for e is 01100101 as a bitstring. What is it in octal? in decimal?

Figure 1 shows the basic computer components.

Conclusions. The computer stores all data as patterns of 0s and 1s, called **bitstrings.** All characters appear on the screen as patterns of dots.

Central memory, processor, hard disc. When you have edited and saved your program `hello.c`, it is now stored on the **hard disc.** (in ASCII, of course). It is *data.*

It is the **processor** which does the work of the computer. Its job is to read **instructions** from central memory and execute them. The instructions are contained in **executable programs.**

When you type

```
gcc hello.c
```

the computer copies an executable program called **gcc** into central memory, then executes that program on the data contained in `hello.c`. It produces a new executable program which is usually called **a.out** and stores it on disc.

When you type (on 'jbell', say)

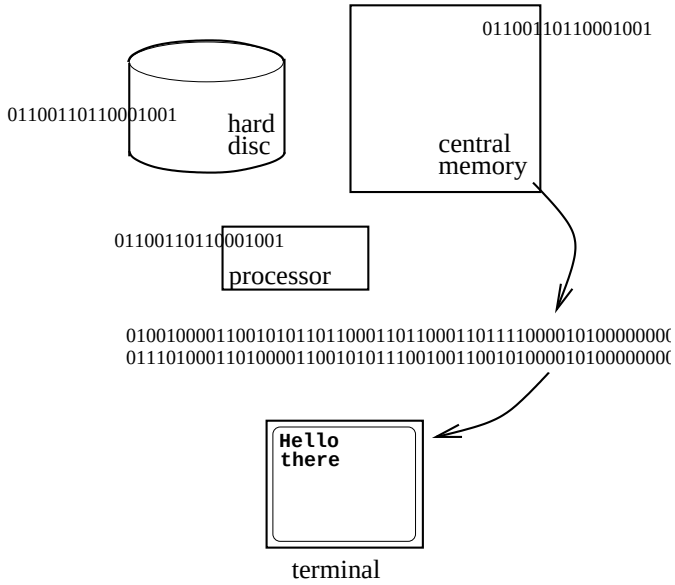


Figure 1: Parts of a computer

```
%jbell a.out
```

the computer copies `a.out` into central memory and executes it, with the results as described.