Computer architecture

What is a computer?

Computer

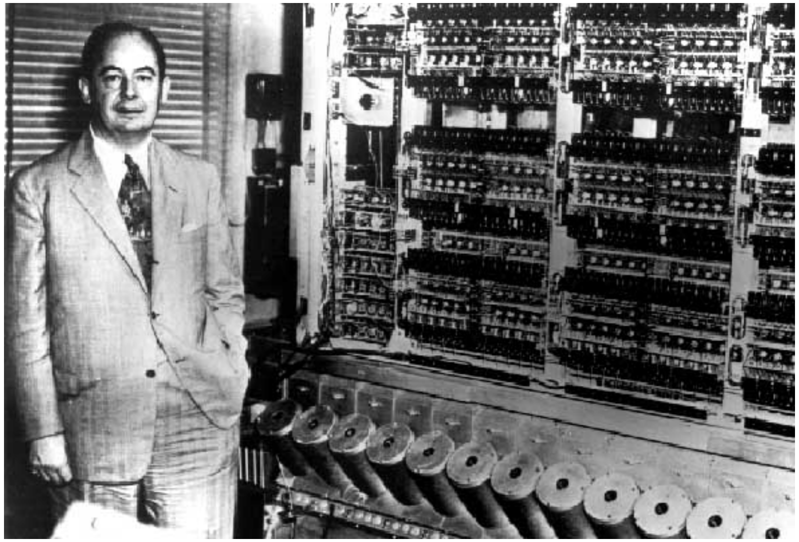
A computer is:

- an electronic device (mechanical, biological?)
- programmable (it runs *programs*, ie. user-defined sequences of instructions)
- processes data (programs read, treat and write information from/to memory and peripherals)

Von Neumann architecture

- processor (CPU: central processing unit)
- memory (RAM: random access memory)
- peripherals

Program: sequence of machine code instructions encoded with binary numbers



Von Neumann architecture (2)

To execute ("run") a program:

- copy entire program somewhere in memory
- point processor at first instruction ("start here")
- processor reads instruction
- processor executes instruction
- processor goes to next instruction
- go on forever

Harvard architecture

- processor
- program memory
- data memory
- peripherals

Advantages:

- can access both memories independently, at the same time: more efficient
- both memories can have different technical characteristics: optimized electronics

In practice, "Modified Harvard architecture":

- common memory for both program and data (allows moving programs around as if they were data)
- different buses and caches for program and data (allows accessing both at the same time)

Harvard architecture (2)

cache:

> “ (computing) A fast temporary storage where recently or frequently used information is stored to avoid having to reload it from a slower storage medium.

bus:

> ‘ (computing) A distinct set of conductors carrying data and control signals within a computer system, to which pieces of equipment may be connected in parallel.

Operating system

Operating system (itself a program!) takes care of:

- loading programs from hard drive to memory
- cleaning up memory when program is over
- running several programs at the same time
- providing libraries for easier access to hardware

Program and data

Instructions in a program are mostly about:

- manipulating data
  - reading data from memory/peripherals
  - writing data to memory/peripherals
  - performing simple operations on data
    - arithmetics
    - comparison
- jumping to other instructions depending on some data ("if this, then jump"). Otherwise a program would always do the exact same thing, boring.
- that's pretty much it!

Program and data

You can now write Mass Effect 3!

Mozilla Firefox:

- ~15 million LOC
- ~60 MB of machine code
- ~10 million instructions

What's data anyway?

Memory is just a big bunch of boxes. Each box:

- can contain a number from 0 to 255. That's a *byte*.
- has an *address*, from 0 to however many bytes of RAM your computer has.

When the processor deals with memory, it looks like this:

- read number from some address, and use it for operation
- write result of operation to some address

What's data anyway? (2)

Anything we want a computer to deal with has to be encoded somehow with bytes.

In practice, the laws of electronics are such that it's easier to have boxes of zeros and ones than boxes of anything from 0 to 255. That's *bits*. A byte is made of 8 bits. But since the processor cannot address individual bits (only bytes), it doesn't really matter.

Other kinds ("types") of data

So, integers from 0 to 255 are not enough?

- take two bytes at a time for integers from 0 to 65535
- 4 bytes: 0 to 4294967295
- 8 bytes: 0 to 18446744073709551615

Fortunately, the processor can also read, write and operate directly on integers of these sizes.

- Negative integers? Just interpret the first bit as the sign, and the rest as a positive integer (that's a lie).
- Non integer numbers? Use part of the bits for the integer part, and the rest for the fractional part (that's also a lie).

Other kinds ("types") of data (2)

- Text? Use a one to one match between characters and numbers (an *encoding*) and interpret numbers as characters. In ASCII, 'a' is 97, 'W' is 87 and '$' is 36.
- How about colors for the pixels on the screen? One number from 0 to 255 for the red component, another for the blue component, another for the green. Three bytes make a pixel.
- Sound? The CD format specifies 2 byte integers for each samples, and 44100 samples per second, each sample representing the intensity of the pressure wave at a given moment.

> It's all numbers!

Appendix

Binary numbers

Base 2 positional numeral system. Just like "ordinary" numbers are base 10.

```
25486 = 2 * 10^4  + 5 * 10^3 + 4 * 10^2 + 8 * 10^1 + 6 * 10^0
      = 2 * 10000 + 5 * 1000 + 4 * 100  + 8 * 10   + 6 * 1


10110 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0
      = 1 * 16  + 0 * 8   + 1 * 4   + 1 * 2   + 0 * 1
      = 22
```

Arithmetics work the same:

```
   1101 (13)                1101 (13)
+  1001 (9)              x    11 (3)
-------                  --------
= 10110 (22)                1101 (13)
                           1101  (26)
                         --------
                         = 100111 (39)
```

Binary numbers (2)

- bit: **b**inary dig**it** (0 to 1)
- octet: 8 bits (0 to 255)
- byte: smallest addressable amount of memory (= 1 octet on most modern CPU architectures)
- nybble: half a byte, 4 bits (0 to 15)
- word: the "normal" integer size for the architecture (was 16 bits, now 32 or 64)

SI prefixes confusing. Prefer binary prefixes:

```
k: kilo = 1000          Ki: kibi = 1024   = 1.024 k
M: mega = 1000^2        Mi: mibi = 1024^2 = 1.049 M
G: giga = 1000^3        Gi: gibi = 1024^3 = 1.074 G
T: tera = 1000^4        Ti: tebi = 1024^4 = 1.099 T
```

Hexadecimal numbers

Same positional numeral system, but base 16.

- hex digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
- 1 hex digit = 1 nybble (0 to f)
- 2 hex digits = 1 byte (0 to ff)

Some uses:

- constants in code: `0xdeadbeef` (= 3735928559)
- colors in HTML/CSS: `#c0c0ff = rgb(192, 192, 255)`
- MAC addresses: `01:23:45:67:89:ab`
- ...

Negative numbers

2's complement: invert all bits, add one (ignoring overflow)

```
5 = 0000 0101  ==>  -5 = 1111 1010 + 1 = 1111 1011
```

Why? Does not change the addition operation:

```
  0000 0101 (5)            0001 0101 (21)
+ 1111 1011 (-5)         + 1111 0111 (-9)
-----------              -----------
  0000 0000 (0)            0000 1100 (12)


  0001 1001 (25)
+ 1001 1100 (-100)
-----------
  1011 0101 (-75)
```

Floating point numbers

Check Wikipedia if interested.

It's... complicated.