Programmazione e Calcolo Scientifico

Matteo Cicuttin

Politecnico di Torino

March 9, 2024

What is a computer?

- What is a computer in your opinion?
- What does a computer do?

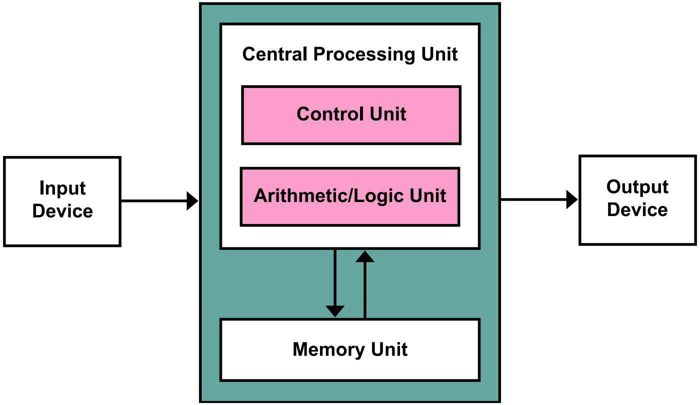
What is a computer?

A computer is a machine capable of executing very simple instructions. Typically they do not do much more than simple arithmetic, comparisons, data movement and jumps.

Factorial in x86 assembly language:

```
fact:                       ;
    mov     eax, 1          ; eax = 1
    cmp     edi, 0          ; if (edi == 0)
    je      fact_end        ;   go to fact_end
fact_loop:                  ;
    imul    eax, edi        ; eax = eax * edi
    sub     edi, 1          ; edi = edi - 1
    jnz     fact_loop       ; if (edi != 0)
                            ;   go to fact_loop
fact_end:                   ;
    ret                     ; return to caller
```

Von Neumann architecture



In this course we will study:

- Processor
- Memory
- Their interactions

Image from `https://en.wikipedia.org/wiki/Von_Neumann_architecture`

Processor: The data path

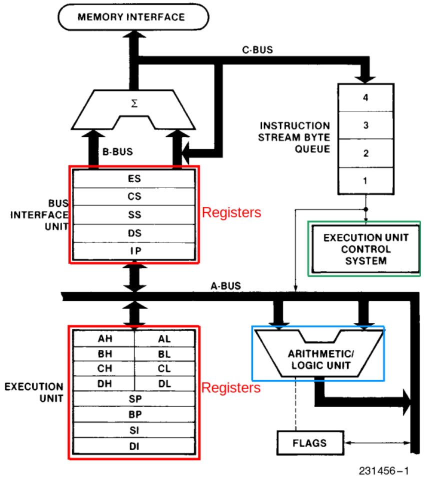


Figure 1. 8088 CPU Functional Block Diagram

- Registers: basic memory units in a computer system. Some are user-visible, some are not
- Arithmetic logic unit: does the (integer) math
- Control unit: makes everything work together
- Flags: Status bits about the result of the last operation (overflow, carry, zero, ...)

But who drives the flow of data in the data path?

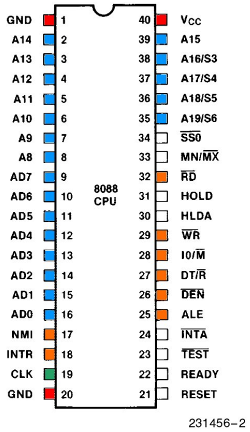
Image from the Intel 8088 datasheet, document No. 231456.

Fetch-decode-execute

CPU runs instructions from a program stored in memory. A register usually called `PC` or `IP` points to the next instruction to execute. From startup to shutdown CPU does what is called **instruction cycle**:

1. **Fetch**: retrieve the next instruction from memory
2. **Decode**: figure out the type of instruction (memory access and if is direct/indirect, arithmetic operation, jump, ...)
3. **Execute**: Activate the appropriate units of the CPU, for example the ALU for an arithmetic instruction. Possibly modify `IP` if it was a jump
4. **Repeat**

Interface to the outside world



GND 1
A14 2
A13 3
A12 4
A11 5
A10 6
A9 7
A8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GND 20
8088 CPU
40 Vcc
39 A15
38 A16/S3
37 A17/S4
36 A18/S5
35 A19/S6
34 $\overline{\text{SS0}}$
33 MN/$\overline{\text{MX}}$
32 $\overline{\text{RD}}$
31 HOLD
30 HLDA
29 $\overline{\text{WR}}$
28 IO/$\overline{\text{M}}$
27 DT/$\overline{\text{R}}$
26 $\overline{\text{DEN}}$
25 ALE
24 $\overline{\text{INTA}}$
23 $\overline{\text{TEST}}$
22 READY
21 RESET
231456–2

Figure 2. 8088 Pin Configuration

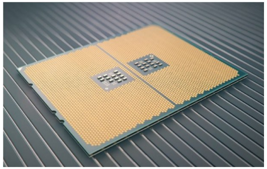
- Power: without power the CPU won't work :)
- Data and address signals: `D0...D7` are the data I/O pins, `A0...A19` are the address pins $\Longrightarrow$ where I want to read data from
- Control signals: ask the memory to read or write, interrupts...
- Clock: drives all the operations inside the CPU

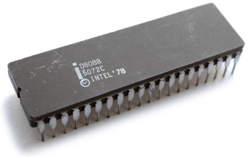
The 8088 is a CPU from 1979. At power-on, modern CPUs appear **exactly** like an 8086/8088: in order to use all their features, the operating system has to switch them into **protected mode** (32 bit) or **long mode** (64 bit).

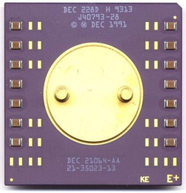
Image from the Intel 8088 datasheet, document No. 231456.

Some CPUs

intel
XEON
4th Gen
Intel Xeon Scalable Processor



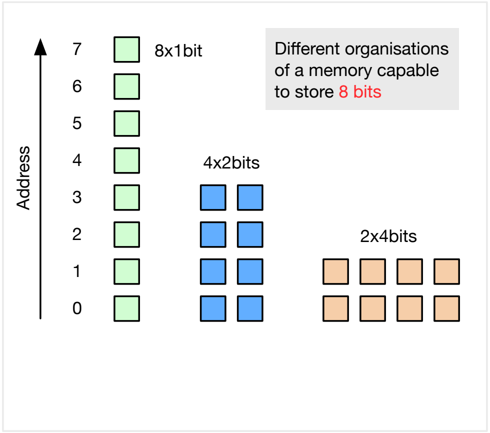

D8088
5072C
© INTEL '78


DEC 228D H 9313
J40793-28
© Ⓜ DEC 1991
DEC 21064-AA
21-35023-13
KE E+

Sun
© Ⓜ 1999, SMI
Ultra SPARC III
D1219879 0206
PG2.2 980 USA
SME 1052B LGA
7154
-1050
EEC

The memory

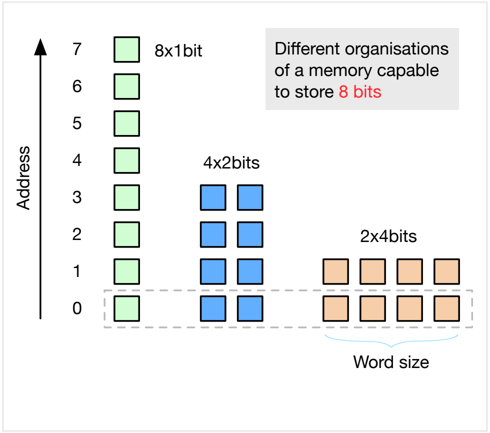
The memory holds 0s and 1s, which we call bits (binary digits). Nothing more.



- You can arrange your bits in various ways however...

The memory

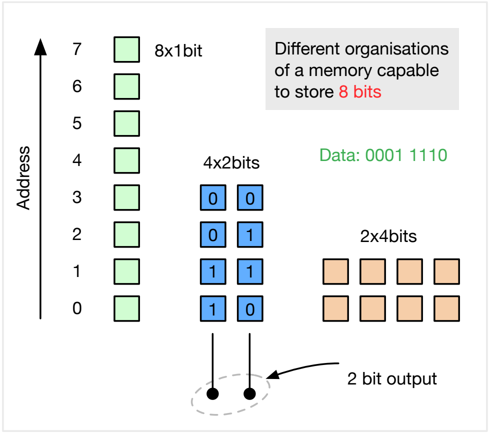
The memory holds 0s and 1s, which we call bits (binary digits). Nothing more.



- You can arrange your bits in various ways however...
- And this determines the **word size** of your memory

The memory

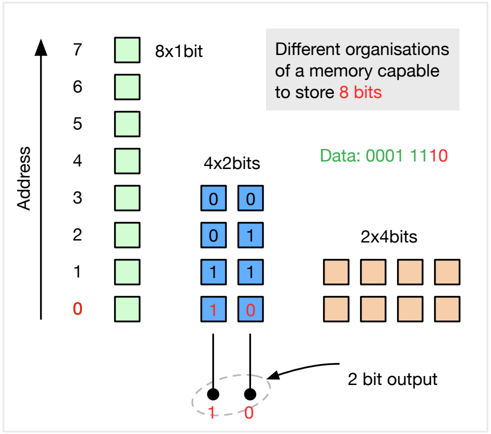
The memory holds 0s and 1s, which we call bits (binary digits). Nothing more.



- We consider for example the 4x2bit layout: memory has 2 byte output.

The memory

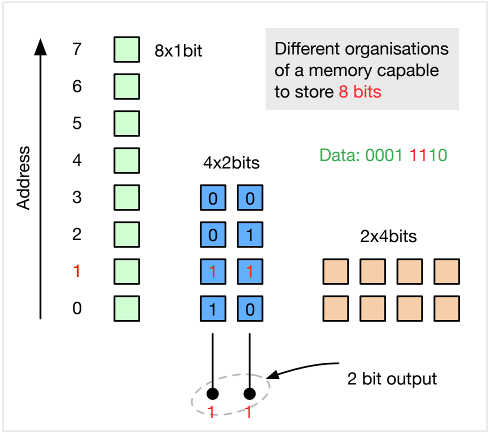
The memory holds 0s and 1s, which we call bits (binary digits). Nothing more.



- We consider for example the 4x2bit layout: memory has 2 byte output.
- The word we're interested in is selected by specifying an address.

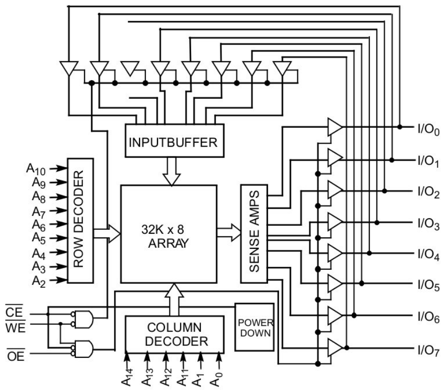
The memory

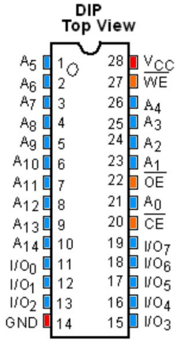
The memory holds 0s and 1s, which we call bits (binary digits). Nothing more.



- We consider for example the 4x2bit layout: memory has 2 byte output.
- The word we're interested in is selected by specifying an address.

A memory from the '80s: the 62256


INPUTBUFFER
ROW DECODER
32K x 8 ARRAY
SENSE AMPS
COLUMN DECODER
POWER DOWN
A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2
$\overline{CE}$
$\overline{WE}$
$\overline{OE}$
A_{14} A_{13} A_{12} A_{11} A_1 A_0
I/O_0 I/O_1 I/O_2 I/O_3 I/O_4 I/O_5 I/O_6 I/O_7



256 kbits, layout is 32k by 8bits: notice that we have exactly 15 address pins (`A0...A14`)

Connecting the memory and the CPU

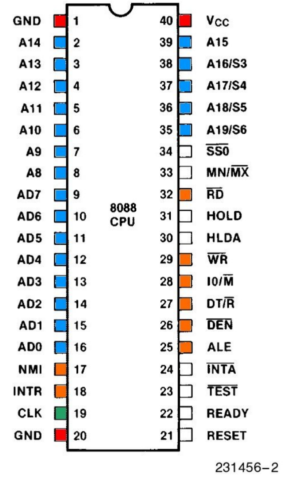
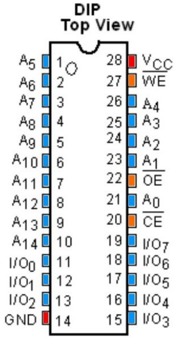


Figure 2. 8088 Pin Configuration



The system bus of the 8088

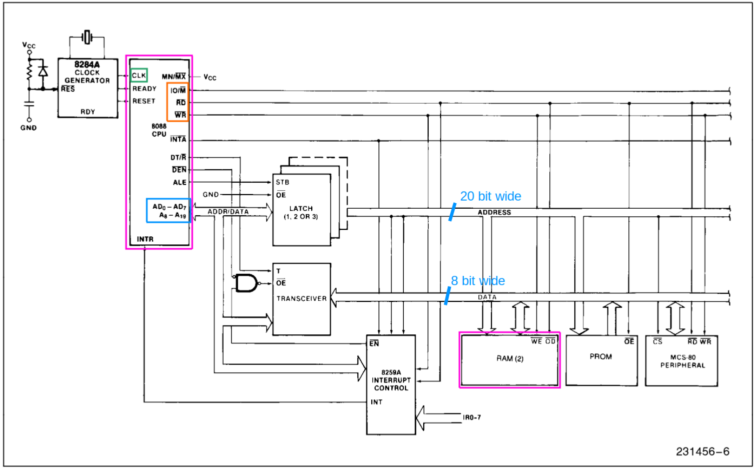
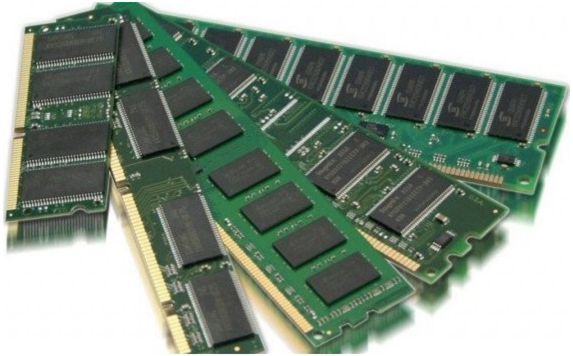


Figure 6. Demultiplexed Bus Configuration

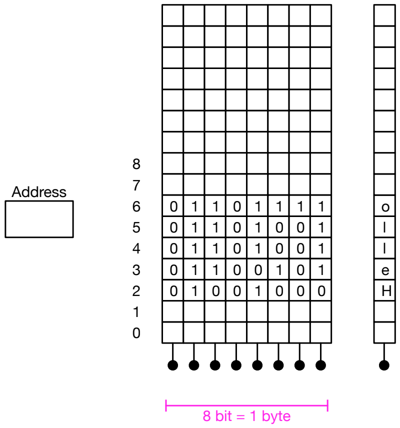
- For each bus operation, one byte gets transferred

Memory in a modern PC



- From the original Pentium onwards, data bus width in PCs is 64 bit (yes, even in 32 bit architectures)
- On the 32 bit machines, address bus is 32 bits
- On 64 bit machines, even if you have 64 bit pointers, physical address bus width is anywhere between 39 and 48

Memory from the software point of view

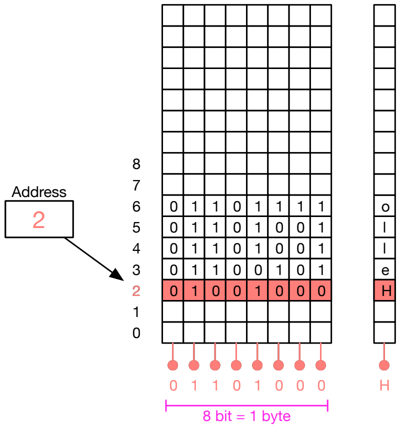


Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

Memory from the software point of view

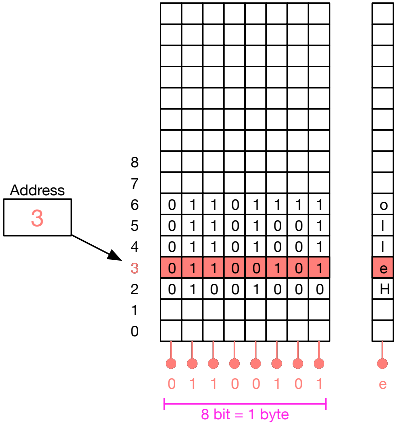


Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

Memory from the software point of view

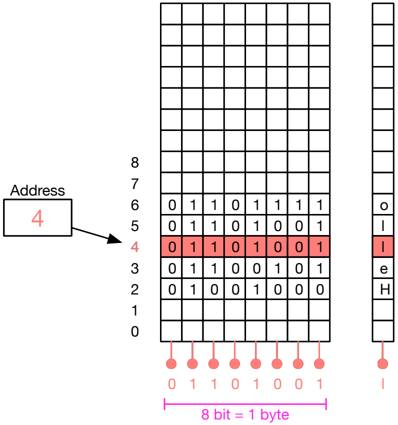


Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

Memory from the software point of view

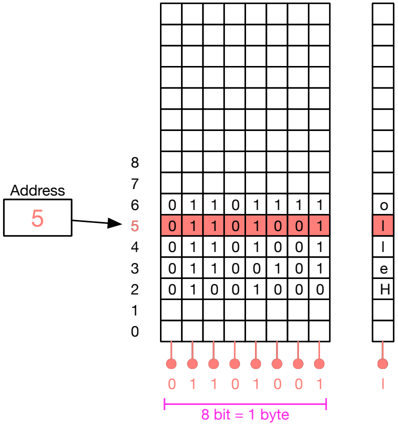


Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

Memory from the software point of view

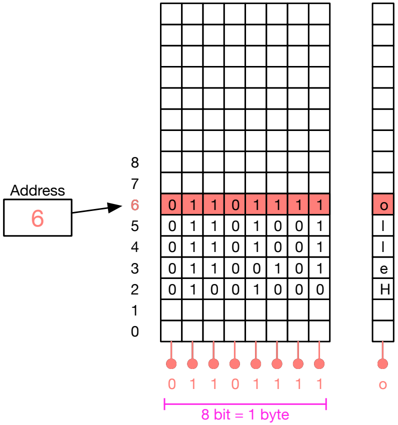


Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

Memory from the software point of view



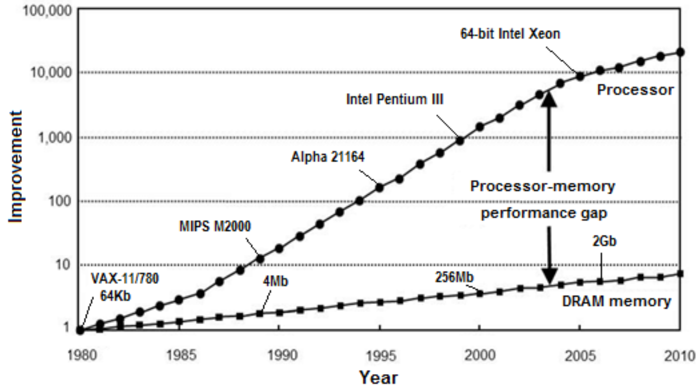
Memory is addressable with byte granularity. In other words, at the lowest level is a huge array of bytes.

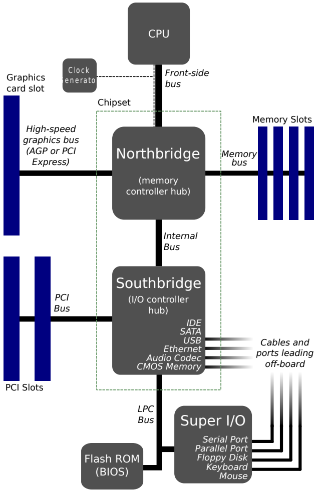
- The address is the index in this big array
- String `Hello` starts at address 2

This concept must be **crystal clear**. Otherwise you are guaranteed to have huge difficulties with C++.

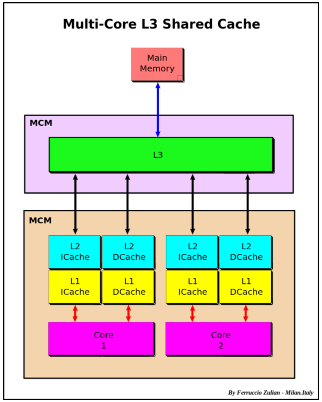
The different evolution of CPUs, memories and other peripheals

CPUs and memories evolved at different speeds. This gave rise to layered architectures.





The need for cache memory



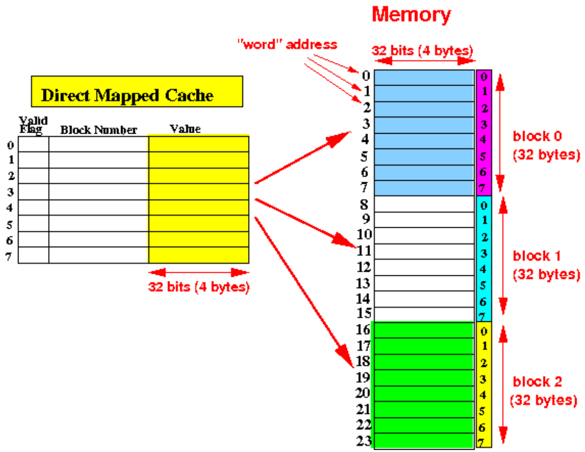
How to handle speed difference between CPU and memory?

- Cache: small but fast memory
- Locality principle suggests to introduce caches
- Algorithms must be cache-aware, otherwise you are going to waste resources
- Knowledge of CPU-memory interactions and caching mechanisms is required to develop efficient algorithms

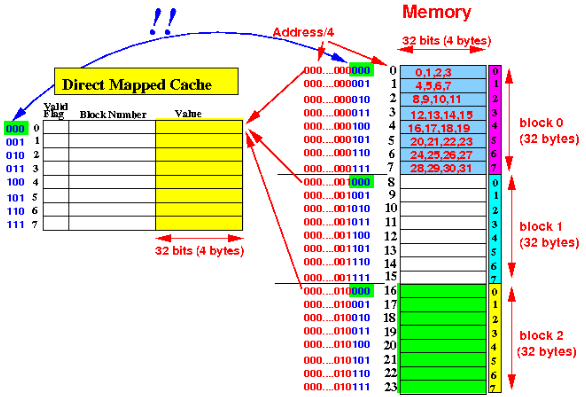
On modern machines:

- L1: 32-128 kB, $\mathcal{O}(1\ \text{TB/s})$
- L2: 512 kB, $\mathcal{O}(1\ \text{TB/s})$
- L3: 6-128 MB, $\mathcal{O}(400\ \text{GB/s})$
- Main memory: 8-128 GB, $\mathcal{O}(100\ \text{GB/s})$

A simple, direct-mapped cache



A simple, direct-mapped cache



Understanding AXPY, XNPY and matrix multiplication performance

We now have all the knowledge to understand AXPY and XNPY performance.

AXPY:

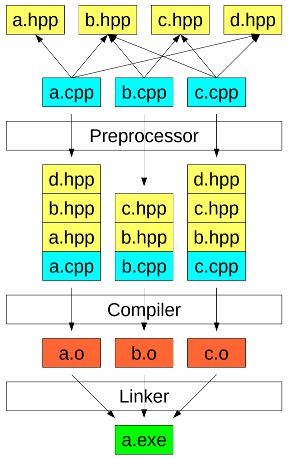
```
for (size_t i = 0; i < N; i++)
    y[i] = alpha*x[i] + y[i];
```

XNPY:

```
for (size_t i = 0; i < N; i++) {
    double xpow = 1.0;
    for (size_t p = 0; p < pow; p++)
        xpow *= x[i];
    y[i] = xpow + y[i];
}
```

- The CPU can process much more data than memory is able to deliver
- The `x` in XNPY is in fast memory after first read, therefore no need to re-read from main memory
- Matrix multiplication speed depends on the cache access: if you don't use cache correctly, your code will be slow

C++ is a compiled language



C++ is a **compiled** language. This means that before executing a program, you have to translate it to machine code. This involves different steps:

- Preprocessor
- Compiler
- Linker

C++ is statically typed language

In C++, the **type** of every entity must be known **to the compiler** at its point of use.

Entities are

- values: 42 is an integer, 42.123 is a floating point number
- names: a variable `x` has a type, a function `f` has a type
- expressions: `x+3*y` has a type, `f(1,2,3)` has a type

The type of an entity determines the set of operations applicable to it.

The minimal C++ program

The minimal C++ program is the following:

```
int main()
{}
```

- `main` is a **function** which takes no parameters and does nothing
- in every program the execution starts at `main`, therefore every C++ program must have a unique `main` function
- Curly braces { and } denote respectively the beginning and the end of a **block**
- `main` returns an integer to the system. 0 means “everything ok”, nonzero means “there was a problem”

A C++ program producing some output

Useful programs typically produce some kind of output

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world\n";
    return 0;
}
```

Functions

Mathematically, a function takes some parameters and produces some output. For example

$$\sqrt{\cdot} : \mathbb{R} \to \mathbb{R}, \qquad ||\cdot|| : \mathbb{R}^N \to \mathbb{R}, \qquad \lfloor\cdot\rfloor : \mathbb{R} \to \mathbb{Z}$$

C++ functions mostly overlap the concept of mathematical functions...

```
double square(double x)
{
    return x*x;
}
```

...however functions may not return a value. Historycally, they were called *procedures*.

```
void print_square(double x)
{
    std::cout << square(x) << "\n";
}
```

Declarations

In a program, we introduce entities by **declaring** them.

```
char c;    int n;    double d;
```

With a declaration we introduce an entity of a certain type:

- Type: set of possible values and possible operations on an object
- Object: some memory that holds a value of a certain type
- Value: set of bits interpreted according to a type
- Variable: a named object

Arithmetic and comparison

Arithmetic operators

- `x+y`: plus
- `+x`: unary plus
- `x-y`: minus
- `-x`: unary minus
- `x*y`: multiplication
- `x/y`: division
- `x%y`: remainder (integer)

Comparison operators

- `x == y`: equal
- `x != y`: not equal
- `x < y`: less than
- `x > y`: greater than
- `x <= y`: less or equal than
- `x >= y`: greater or equal than

Logical operators

- `x & y`: bitwise and
- `x | y`: bitwise or
- `x ^ y`: bitwise xor
- `~ x`: bitwise negation
- `x && y`: logical and
- `x || y`: logical or
- `!x`: logical negation