Computers and Programs

You have seen some C-code

- Computer hardware does not directly understand this.

This week we check what a computer does at the lowest level.

- How is data represented (numbers, ...)
- The internal workings of a machine
- The language that a computer speaks at the lowest level
- Translators, Interpreters

Data Representation: The Problem

The data required for a program to execute is stored in the computer's memory

- Memories store only 1s and 0s

$\Rightarrow$ We need a way of *encoding* all the data we need in strings of bits

$\Rightarrow$ Conversely, we need a way of *interpreting* strings of '0's and '1's

Bit pattern **`01001001`** could mean:

- Character 'H' (ASCII encoding)
- Whole Number 73 (binary encoding)
- Whole Number 3 (there are 3 ones!)
- Ratio 0100 / 1001 = 4/9 = 0.444444444...
- A command to calculate π
- Phrase "Richard D. James for President"

We need some conventions!

Character Encoding: ASCII

ASCII (American Standard Code for Information Interchange) is the most widely used character encoding

- Encoding
  - 7 bits per character (128 possible characters)
  - **`'A'`** = 01000001, **`'B'`** = 01000010,
  - **`'a'`** = 01100001, **`'#'`** = 00100011, ... (see any text book)

Character Encoding: Unicode

- ASCII has been replaced by Unicode
- Unicode defines codes for characters used in every major written language
  - Latin (used in English), Cyrillic (Russian), Greek, Hebrew, and Arabic, plus other alphabets used in India, Asia and across Europe, Chinese, Japanese and Korean ideographs.
  - Includes common symbols, punctuation, etc, etc, and has lots of spare room for expansion
- Encoding:
  - 16 bits per character ($>$ 1000000 characters with expanding codes)
  - ASCII is compatible subset: 1st 128 characters are the same
- Characters are known as a **`char`** in C.
- Literal characters in C are enclosed between primes `'`-s.

Positive Whole Numbers

Use n bits to represent any number between 0 and $2^n - 1$.

- Use a binary number system (as opposed to a decimal system)

$$\begin{array}{rr} 00000_2 & 0_{10} \\ 00001_2 & 1_{10} \\ 00010_2 & 2_{10} \\ \ldots & \ldots \\ 11110_2 & 30_{10} \\ 11111_2 & 31_{10} \end{array}$$

- $\ldots dcba_2 = 2^0 a + 2^1 b + 2^2 c + 2^3 d + \ldots = a + 2b + 4c + 8d + \ldots$
- 8 bits can represent numbers between 0 and 255
- 32 bits can represent numbers between 0 and 4294967295
- 64 bits can represent numbers between 0 and 18446744073709551615
- Or in hexadecimal

Positive Whole Numbers

Use n bits to represent any number between 0 and $2^n - 1$.

- Use a binary number system (as opposed to a decimal system)

00000_2	0_{10}	0_{16}	(0x0)
00001_2	1_{10}	1_{16}	(0x1)
00010_2	2_{10}	2_{16}	(0x2)
...	...		
11110_2	30_{10}	$1E_{16}$	(0x1E)
11111_2	31_{10}	$1F_{16}$	(0x1F)

- $\ldots dcba_2 = 2^0a + 2^1b + 2^2c + 2^3d + \ldots = a + 2b + 4c + 8d + \ldots$
- 8 bits can represent numbers between 0 and $0xFF$
- 32 bits can represent numbers between 0 and $0xFFFFFFFF$
- 64 bits can represent numbers between 0 and $0xFFFFFFFFFFFFFFFF$
- Or in hexadecimal

Negative Whole Numbers

Two common schemes exist:

- Sign and magnitude:

Sign	Magnitude

  - 1 bit to denote sign
  - The scheme is symmetric, so for every n we can represent $-n$
  - Unfortunately there are two zeros (+0 and -0)
- 2's complement
  - Negative numbers count from the largest unsigned number down; **`11..11`** = -1
  - Signed addition remains the same as unsigned addition
  - Only one zero (**`00...00`**)
  - There is one negative number (**`100..00`**) that has no positive counterpart

2's complement most widely used

2’s Complement (4 bits)

2’s Complement

- For an n bit value,
  - value of bit i is $\begin{cases} -2^i & \text{if } i = n-1 \\ 2^i & \text{otherwise} \end{cases}$
  - or value of bit i is 2^i and if bit $n-1$ is set, subtract 2^n
- Non-negative values can be thought of as having an infinite number of leading zeroes
- Negative values can be thought of as having an infinite number of leading ones
- There is a quick way to find the two’s complement of a number: invert every bit and add one
- Known as **`int`**, **`short`**, **`long`**, **`unsigned int`**, **`unsigned short`**, **`unsigned long`** (sometimes even **`long long`**)

Chars are integers

In C **`char`** is also an integer number

- ASCII: **`'A'`** = 01000001 = 65.

Character arithmetic

- **`'D'`** (68) - **`'A'`** (65) = 3 (not **`'3'`**)
- **`'7'`** (55) - **`'3'`** (51) = 4 (not **`'4'`**)
- Lower and upper case: **`'A'`** + 32 = **`'a'`**
  - This arithmetic is only true in ASCII.

Overflow

- A fundamental property of computer arithmetic is that numbers are stored in a fixed number of bits
- It is possible that the result of a computation involving two n bit numbers requires more than n bits; we call this *overflow*
  - Example: **`0111 + 0010 = 1001 ( 7 + 2 = -7) !!!`**
  - Example: **`1011 - 0100 = 0111 (-5 - 4 = +7) !!!`**
- Overflow can be detected: occurs when adding numbers of the same sign, or subtracting numbers of different signs. We can infer the correct sign, so we can decide if it is wrong

$\Rightarrow$ Ignore it? Stop the program? Saturating arithmetic?

Some properties of numbers

- A left-shift performs multiplication by a power of two
  - Example: **`00101 << 1 = 01010 ( 5*2 =  10)`**
- A right-shift performs division by a power of two — but only on positive numbers
  - Example: **`0110 >> 1 = 0011 ( 6/2 = 3)`**
  - Example: **`0101 >> 1 = 0010 ( 5/2 = 2)`**
- To multiply and divide by arbitrary numbers, we can use the standard pencil and paper algorithm — but it's much easier in binary!
- These don't account for signed numbers (more in computer architecture)

Real Numbers

- The last major thing we need is a way to represent real numbers
- In many calculations the range of numbers is very large
  - For example, a calculation in astronomy might involve the mass of a proton 1.6×10^{-27}kg and the mass of the sun 2×10^{30}kg
- It would be wasteful (of storage and computation time) to store these both in the same form
- Instead, we use a system based on scientific notation

Normalisation

- We are used to writing decimal numbers in scientific notation

$$\begin{gathered}6.022 \times 10^{23} \\ 3.142 \times 10^{0} \\ 2.998 \times 10^{8} \\ 1.602 \times 10^{-19} \\ 2.718 \times 10^{0}\end{gathered}$$

- A *normalised* number has a single non-zero digit to the left of the decimal point: 1.0×10^2 is normalised; 10.0×10^1 and 0.01×10^4 are not
- In binary, a normalised number is of the form: $\pm 1.zzzz \times 2^{yyyy}$

IEEE 754 Floating Point

- Sign and Magnitude.
- Mantissa has an implied leading 1 (one)
- The exponent uses *biased notation*
- Number = $(-1)^S \times (1 + \text{mantissa} \cdot 2^{-53}) \times (2^{\text{exponent}-1023})$
- Size is 64 bits
- Range: 2.22507e-308 to 1.79769e+308 (+/-)

$\Rightarrow$ **`double`** in C, (**`float`**) uses less bits, (1.17549e-38 to 3.40282e+38 (+/-)

More Floating Point

- The biased exponent allows positive numbers to be compared using sign-magnitude integer hardware

IEEE 754 special forms:

Sign	Exponent	Mantissa	
S	0 < Exp < Max	Any bit pattern	Normalized
S	0	0	Zero
S	0	Any non-zero bit pattern	Denormal
S	11....11	0	Infinity
S	11....11	Any non-zero bit pattern	Not a Number (NaN)

Floating Point Gotchas

- Floating point numbers are not real numbers (they are an improper subset, and operators neither associate nor commute)
- There are *four* rounding modes, towards $+\infty$, towards $-\infty$, towards 0 and towards nearest even. Which is best depends on your algorithm
- Floating point arithmetic can *underflow* as well as overflow
- There are *two* zeros ($+0$ and -0), with some funny rules — $\sqrt{-0} = -0$ on round to $-\infty$
- Floating point numbers increase in finite, *variable sized*, steps
- Decimal floating point numbers cannot always be represented exactly. i.e., 0.1 is approximated as 0.10000000149011611938476562500

Why is it impossible to represent 0.1 precisely ???

Concluding:

Everything is represented by strings of 0s and 1s.

- By interpreting the 0s and 1s the right way we can store:
  - Characters (ASCII, Unicode)
  - Whole numbers (**`int`**, usually 2’s complement)
  - “Real” numbers (**`double`**, kind-of sign magnitude)

Next lecture:

- Inner workings of the machine..., (Computer Instructions, Computer Systems, Computer Architecture, Storage Systems, Programming in High level languages, Language Processing)

Inner workings

The essential parts of a computer are a *processor*, a *memory*, and *I/O devices*

- Memory:
  - Stores data.
- Processor:
  - Performs calculations on the data, compares data, moves data, makes decisions
- I/O devices:
  - Interact with the user (screen, keyboard)
  - Permanent and mass storage

Memory

Stores bit strings; usually groups of 8 bits (known as a byte)

- Memory bytes have *addresses*, like pigeon holes
  - The processor can store and load data from specific addresses

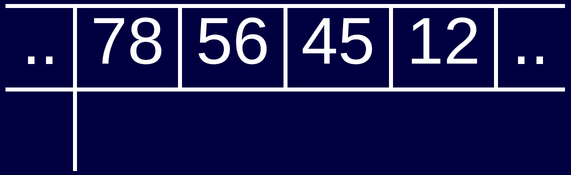
Two types of memory:

1. Main memory (also known as core memory):
   - Earliest memory (1960), magnet core memory, coils magnetised N-S or S-N. 1000 bits for 10000 pounds, 1 kilo.
   - Current memory stores electric charge, 64,000,000,000 bits for 4 pounds, couple of grammes
2. Backing store (also known as mass storage):
   - Early: magnetic drum, heavy, very slow, as much as 100,000 bits.
   - Current: solid state or magnetic plates, 5,000,000,000,000 bits, 2 ounces, 100 pounds.

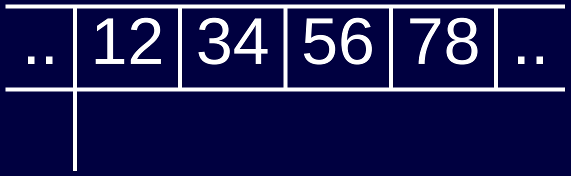
Integer Storage I

- One byte integers give us a range of -128 to +127
- For larger integers we need multiple bytes (typically 4 or 8)
- Considering the four-byte number **`0x12345678`**, we have two layout choices (only for integers, floats can be laid out in many different ways!):

Little Endian: the least significant byte is stored in the lowest memory address



Big Endian: the least significant byte is stored in the highest memory address

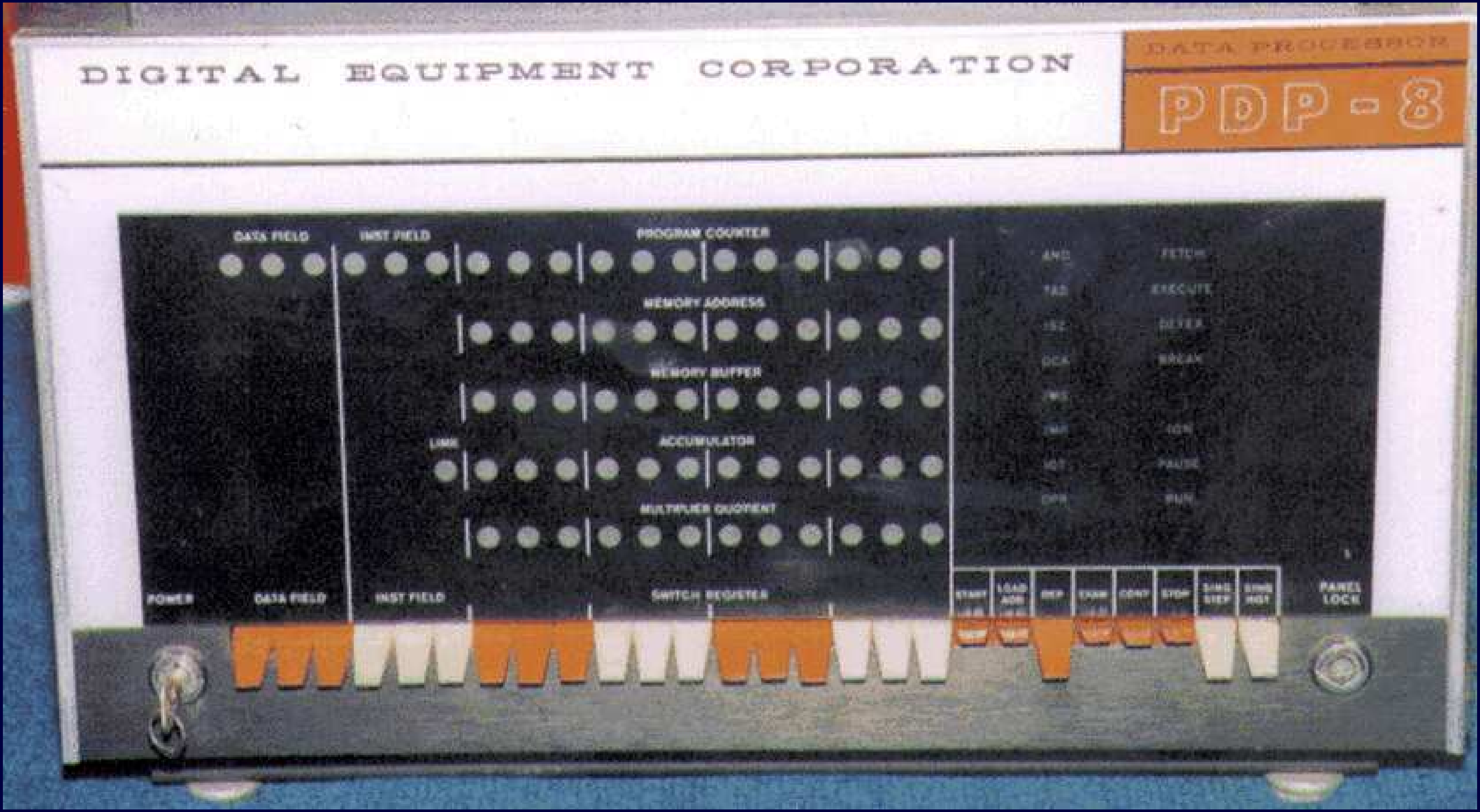


Processor

- A small electronic units that interprets a basic computer language:
    1. reads an machine code instruction from memory (say 0100 0010)
    2. Executes that machine code instruction
    3. Goes on to the next machine code instruction.
- Machine code instructions pointed to by a program counter
- Only limited machine code instructions in a processor. e.g.,
    - Load a constant (eg, 0100 0010)
    - Load a value from memory (eg, 1100 1110)
    - Add two integers (eg, 0011 0010)

$\Rightarrow$ Machine code is the *only* language is executed *directly* by the hardware.

Machine code



PDP-8 (1965): The buttons are for the programming.

Machine code vs Assembly

Machine code is unreadable

- What does **`01000100 01000101 11111001`** do?

⇒ A readable notation was invented, called assembly language

Assembly language uses *mnemonics*, short abbreviations, for instructions.

For example:

```
LDC   4
LDC   5
ADD
```

This will load the constants 4 and 5 and add them, giving 9.

⇒ Easier to read then **`01000100 01000101 11111001`**

Assembly language - II

- The hardware does not interpret assembly language.
- Assembly is *translated* into machine code by a program known as the assembler.
- Machine code is executed by loading it into the computer’s memory, and by jumping to the first instruction

Assembly language vs HLL

Problems with assembly language:

1. It is still unreadable
2. It is not portable; each processor has its own assembly language.

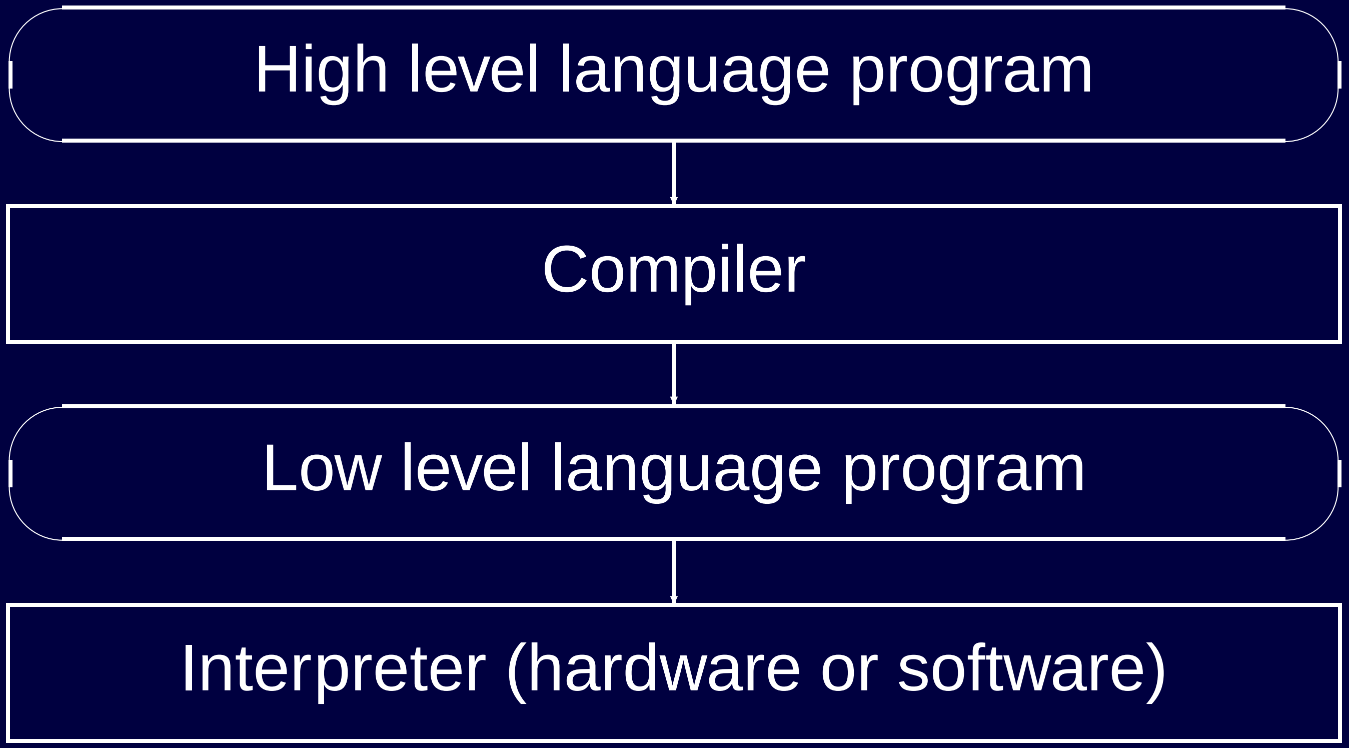
This lead to the invention of High Level Languages (HLL)

- High level language is machine independent
- High level languages support expressions, data types, functions, ...

High level language is translated into assembly language, which is in turn translated into machine code which is in turn executed by the machine.

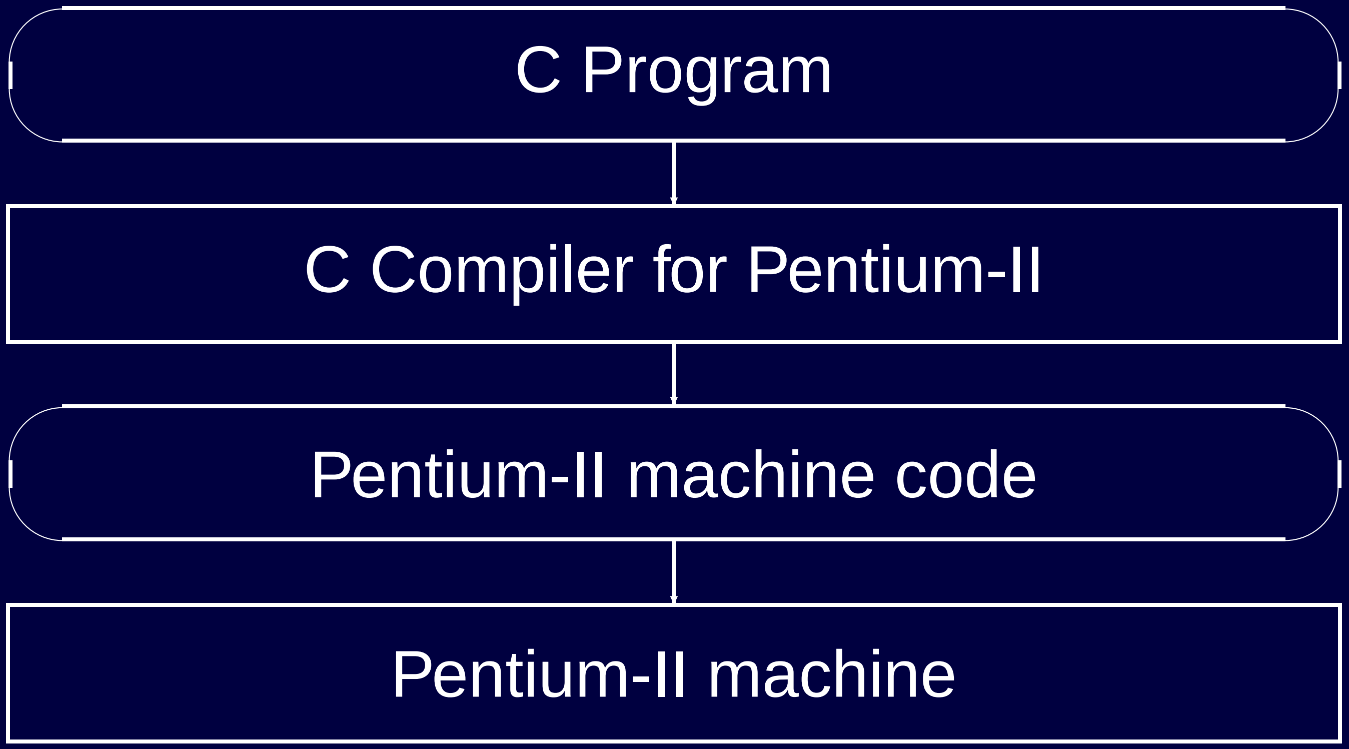
- So who cares? My HLL program is executed by the machine?

Compilation process


High level language program
Compiler
Low level language program
Interpreter (hardware or software)

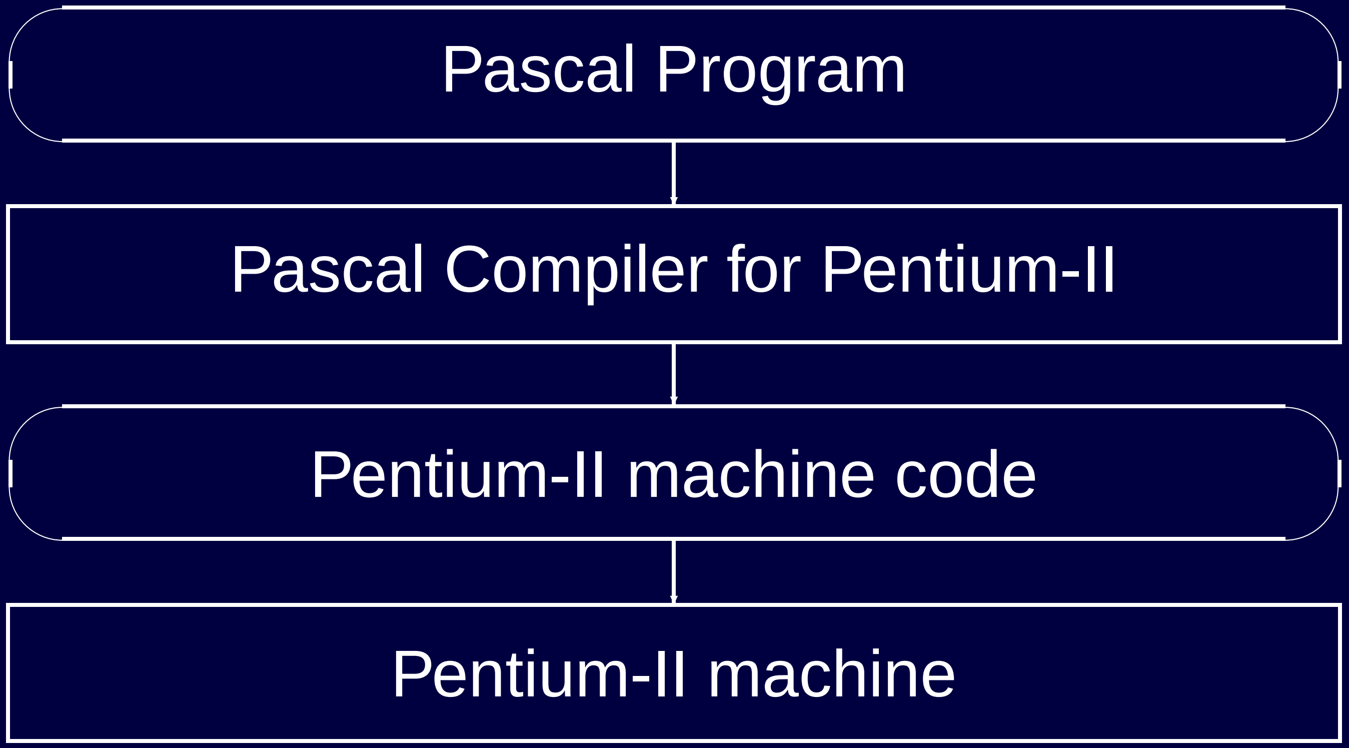
- General structure

Compilation process



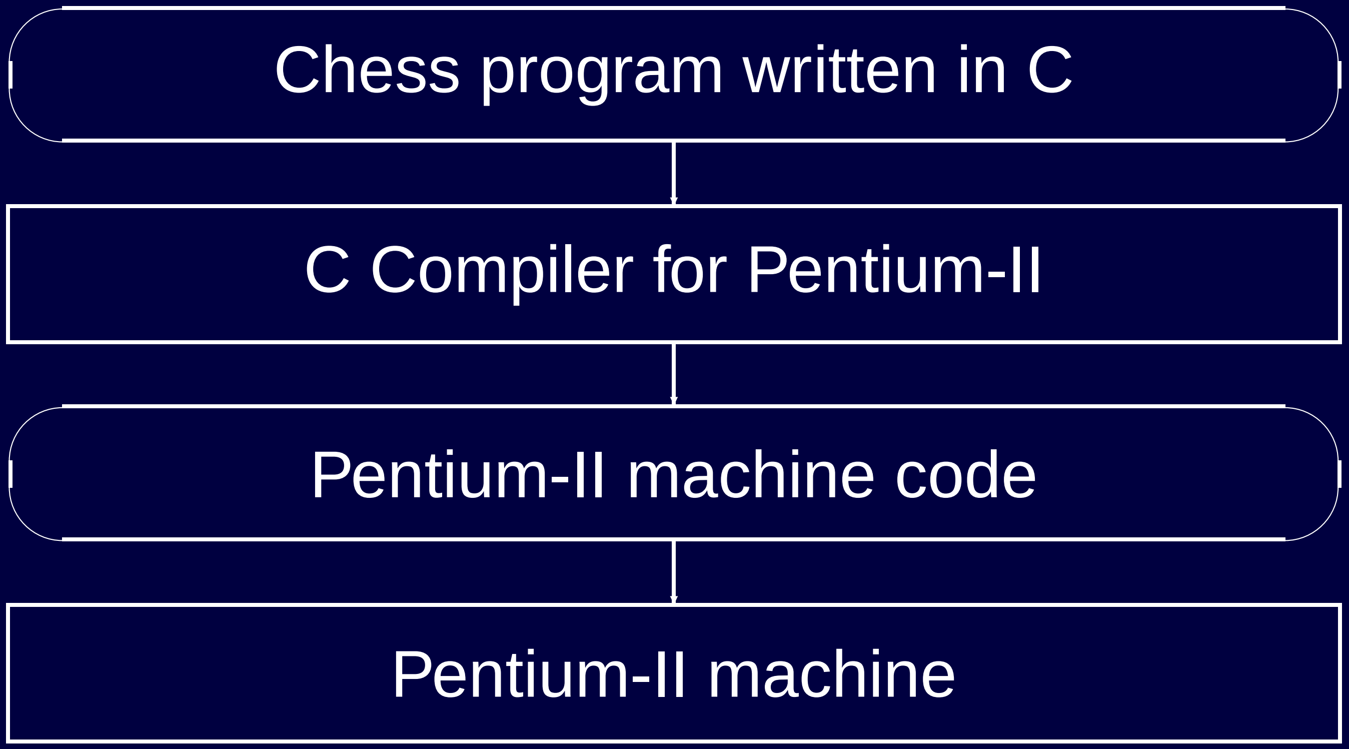
- Example: Pentium II with two languages

Compilation process



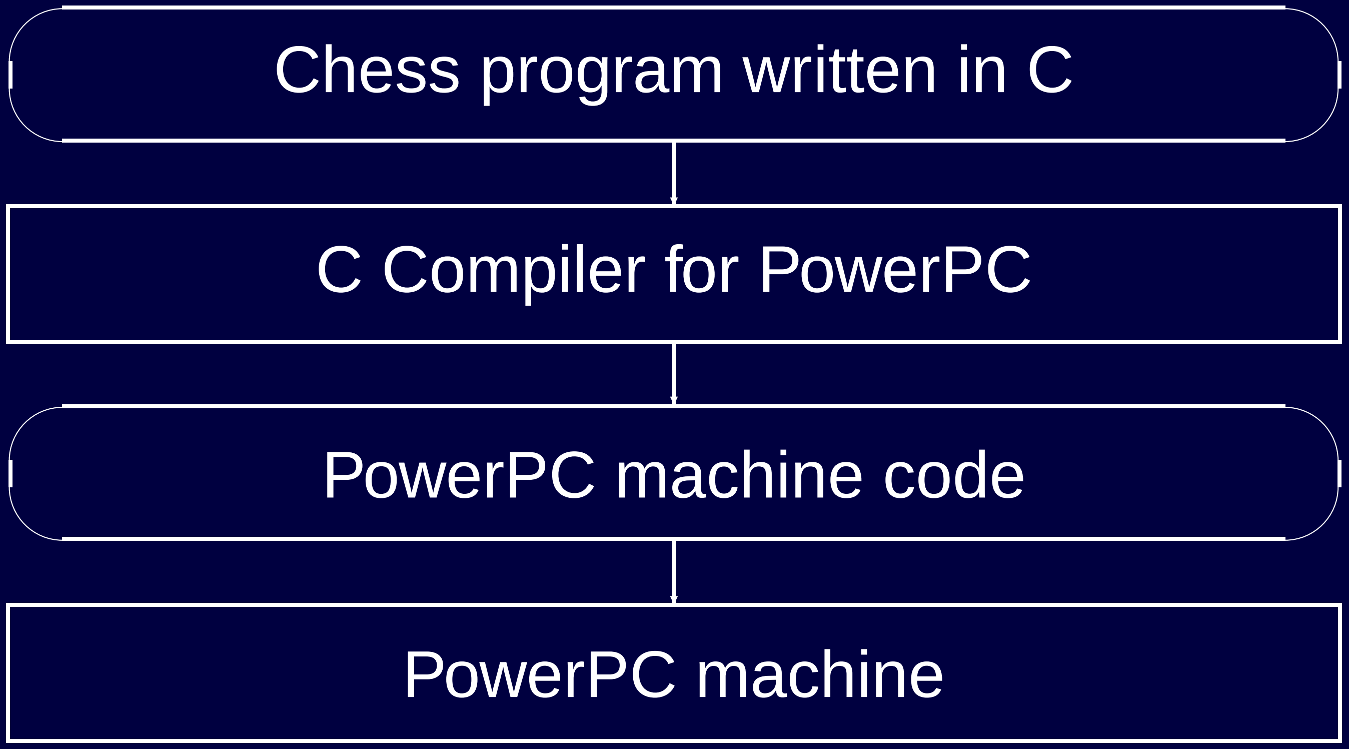
- Choice of programming language thanks to compiler

Compilation process



- Example: Chess in C on two machines

Compilation process



- Portability of code thanks to high level language

Compilation process

- Compiler:
  - Takes a program written in X, produces a program written in Y.
  - Y has exactly the same meaning as X
  - Y is typically closer to the machine than X
- Possibly multiple compilation steps:
  - C++ is translated into C by a C++-compiler
  - C is translated into assembly by a C-compiler
  - Assembly is translated into machine-code by an assembler (a simple compiler)

Phases of a compiler

1. Lexical analysis
   - Translates stream of characters into words ‘m’, ‘a’, ‘i’, ‘n’ becomes “main”
   - Tells user about any illegal characters, for example `@`
2. Parsing (grammar analysis)
   - Places brackets in an expression “3+2*4” becomes “3+(2*4)”
   - Tells user about any incorrect grammar, eg, unbalanced **`( )`**
3. Semantic checking
   - Verifies types (`+int+`, `double`, ...)
   - Verifies existence of all identifiers.
4. Code generation
   - Generates low level program. No errors.

Example,Lexical analysis

The input program

```
double Sine( double r ) {
  return r - r*r*r/6.0 + r*r*r*r/120 ;
}
```

The tokenised program, after lexical analysis

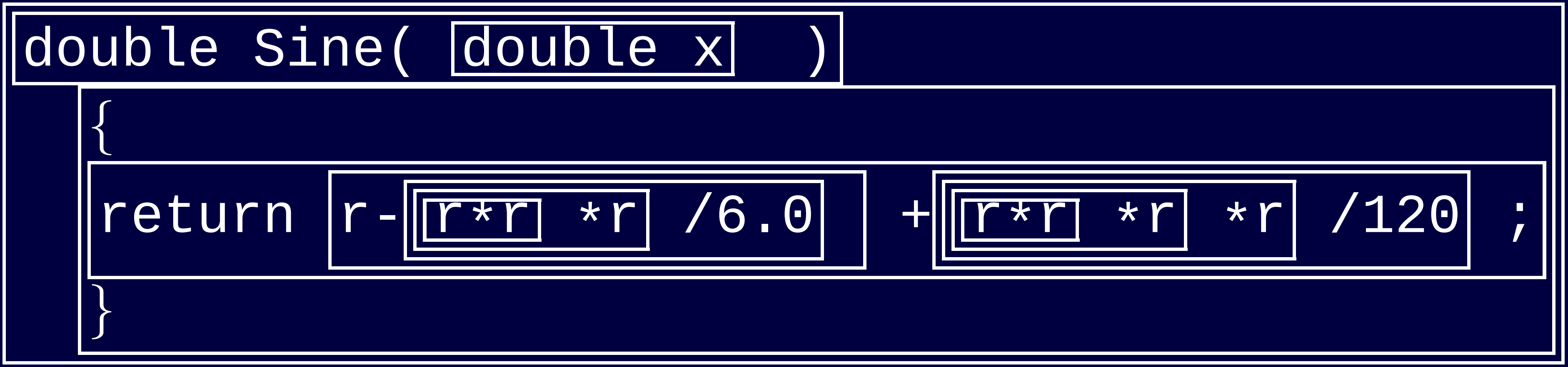
$\boxed{\texttt{double}}\ \boxed{\texttt{Sine}}\ \boxed{\texttt{(}}\ \boxed{\texttt{double}}\ \boxed{\texttt{r}}\ \boxed{\texttt{)}}\ \boxed{\texttt{\{}}\ \boxed{\texttt{return}}\ \boxed{\texttt{r}}\ \boxed{\texttt{-}}\ \boxed{\texttt{r}}\ \boxed{\texttt{*}}\ \boxed{\texttt{r}}\ \boxed{\texttt{*}}\ \boxed{\texttt{r}}$
$\boxed{\texttt{/}}\ \boxed{\texttt{6.0}}\ \boxed{\texttt{+}}\ \boxed{\texttt{r}}\ \boxed{\texttt{*}}\ \boxed{\texttt{r}}\ \boxed{\texttt{*}}\ \boxed{\texttt{r}}\ \boxed{\texttt{*}}\ \boxed{\texttt{r}}\ \boxed{\texttt{/}}\ \boxed{\texttt{120}}\ \boxed{\texttt{;}}\ \boxed{\texttt{\}}}$

Example,Grammar analysis

The input program

```
double Sine( double r ) {
  return r - r*r*r/6.0 + r*r*r*r/120 ;
}
```

The parsed program, after grammatical analysis



```
double Sine( double x  )
    {
    return r- r*r  *r  /6.0  + r*r  *r  *r  /120 ;
    }
```

Example,Semantic analysis

The input program

```
double Sine( double r ) {
  return r - r*r*r/6.0 + r*r*r*r/120 ;
}
```

Semantic check:

- `r` is defined.
- `+` is adding floating point numbers
- `/` is dividing floating point numbers
  $\Rightarrow$ 120 should be 120.0 (*coercion*)
- `*` is multiplying floating point numbers... and so on.

Example,Code Generation

The input program

```
double Sine( double r ) {
  return r - r*r*r/6.0 + r*r*r*r/120 ;
}
```

Code generation:

The output program

```
  LD  r          DIV            MUL
  LD  r          SUB            MUL
  LD  r          LD  r          LD  r
  MUL            LD  r          MUL
  LD  r          MUL            LDC 120.0
  MUL            LD  r          DIV
  LDC 6.0        LD  r          ADD
```

Concluding

Types:

- Booleans, Integers, Floating point numbers, Characters.
- All represented by bit patterns.
- How we interpret bit strings implies what they mean

Machine:

- Executes machine code.
- Compiler translates High Level Language program to machine code.