Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CS Basics
4) Development Process Codification of numbers

E. Benoist & C. Grothoff
Fall Term 2018-19

Computer Science Division

Development Process

- Binary vs text files
- "Endianness"
- Negative Numbers
- Introduction to Floating Point
  - Floating point binary number
  - Fixed point numbers
  - Floating points
- Compilation
  - Assembly Language
- Development process
- Using Make

Binary vs text files

Binary vs. text files

- **Different file formats**
  - Proprietary file formats: doc, ppt, xls
  - Images: jpeg, gif, png
  - Documents: pdf
  - Executable: exe, dll, so, class
  - Texts: txt, xml, html, tex
  - Program source files: java, asm, cpp, c
- **Two big families**
  - Binary files (office, images, executables, ...)
  - Text files (txt, tex, html, source files, ...)

Text files

- **Text files**
  - ASCII files
  - Each letter is encoded on 1 byte
  - Standardized on 7 bits (for English)
  - 94 visible characters
  - Plus other invisibles like: cariage return, tab, space, bell, ...
- **Text files encoding**
  - Different encoding formats for accents
  - Depends on the language: western Europeans, eastern Europeans, ...
  - ISO latin1, UTF-8, ...
- **Letters encoded on more than one byte**
  - Unicode permits to encode any language
  - Characters can be coded on more than one byte
  - Arabic, chinese, herbrew, ...
- **How to see text data?**
  - Using any text editor or IDE: notepad (windows), gEdit, Emacs, Kate, Eclipse, Net Beans ...

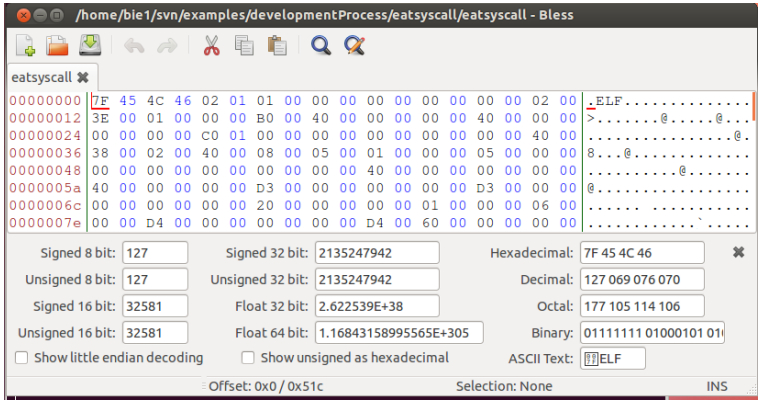
Binary files

- **Proprietary file formats**
  - Office files (Word, Excel, Powerpoint, ...)
  - Many other applications
- **Executable files**
  - Windows: exe,
  - Linux: elf (32 bits), elf64 (64-bit)
  - Contain machine instructions encoded in binary
  - Can be analyzed with a **decompiler**
- **Images**
  - tiff: bitmap of an image (high-quality photos)
  - jpeg: format family, lossy compression (for online photos)
  - gif: 8-bit color, animations possible (legacy)
  - png: 32-bit color, no animations, modern lossless compression (screenshots)
  - svg: vector graphics
  - File formats are known, libraries manipulate those binary files
- **How to see any type of binary data?**
  - using an hex editor

Example of Binary File

- **Using the editor** `Bless`
  - Can open a file
- **Two versions**
  - See the text version on the left (bytes are interpreted as chars)
  - See the binary version on the right
  - Can read and edit any binary
  - You can edit executable files: BUT DO NOT DO IT!

The binary editor Bless



Interpreting Raw Data

- **Data is ultimately always encoded in binary**
  - Common format for text is ASCII
  - Capital letter "S" in encoded with 53H
  - Corresponds also to the decimal number 83
  - In the computer it is a set of 8 bits 01010011B
  - This pattern can be anything else in a binary computer program
  - Can be part of an instruction
  - Can be part of a 16-bit number
  - Can be part of a 32-bit integer
  - Can be any data (floating point numbers, objects, address, . . .)
- **Example**
  - 53H may be interpreted as value 83
  - 53 61H may be interpreted as the decimal 21'345
  - 53 61 6D 0A H may be interpreted as the decimal 1'398'893'834
  - 53 61 6D 0A 77 61 73 0AH may be interpreted as the floating point $4.54365038640977.10^{93}$

"Endianness"

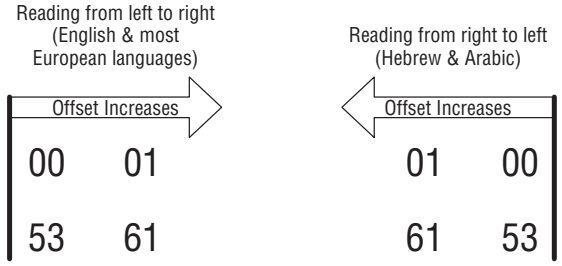
"Endianness"

- **Raw data are stored in an array**
  - It can be displayed from right to left (like in French and German)

    0 1 2 3 4 5 6 7
    53H 61H 6DH 0AH 77H 61H 73H 0AH
  - It can be displayed from left to right (like in Arabic or Hebrew)

    7 6 5 4 3 2 1 0
    0AH 73H 61H 77H 0AH 6DH 61H 53H
  - It remains the same array, and the same number
- **Which number does this array represents?**
  - 53616D0A7761730AH
  - or 0A7361770A6D6153H

Difference of order



So is it "53 61H" or "61 53H" ?

Figure 5-4: Differences in display order vs. differences in evaluation order

Convention

- **Big Endian**
  - numbers are written with the most significant bytes first
  - The first end contains the "big" bytes
  - Number 20A1H is written in the array

    ```
    0    1
    20H A1H
    ```
- **Little Endian**
  - numbers are written with the least significant bytes first
  - The first end contains the "little" bytes
  - Number 20A1H is written in the array

    ```
    0    1
    A1H 20H
    ```
- **Convention**
  - Intel x86 architecture uses little endian
  - Other processors may use big endian
  - Some processors can even be switched!

Big Endian vs Little Endian

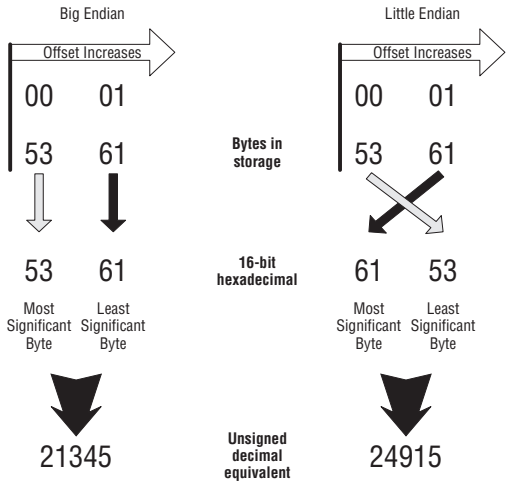


Figure 5-5: Big endian vs. little endian for a 16-bit value

Negative Numbers

How to Represent Negative Numbers?

- **Numbers can be positive**
  - Decimal: 15
  - Stored in memory in hexadecimal: 0FH (on 8 bits)
  - or 0000000FH in 32 bit
- **And can also be negative**
  - Decimal: -15
  - How should we represent it?

Different possible representations

- **One bit for the sign**
  - The first bit is used for the sign
  - 15 is respresented by 000FH on a 16-bit architecture
  - -15 by 800FH
- **Problems:**
  - Two representations of 0 exist: 0000H and 8000H
  - Standard (bit-level) addition does not work:
    15 + (-15) = 000FH + 800FH = 801EH = -30
- **One's Complement**
  - The negative number is obtained by inverting all the bits
  - 15 = 000FH
  - 15 = 0000 0000 0000 1111B
  - -15 is represented by 1111 1111 1111 0000B
  - -15 is represented by FFF0H
- **Problems**
  - We have two different representations of 0 0000H and FFFFH
  - Addition does not work:
    15 = 000FH and -5 = FFFAH. The sum is 10009H which makes 9 if we ignore the overflow.

Two's Complement

- **Negative numbers are designed to respect addition**
  - 15 + (-10) = 5
  - Idea: -X is represented by $2^n - X$ if n is the number of bits
  - X + (-X) = X $+2^n - X = 0$ + overflow
- **Example 1**
  - Notation for -1 on 16 bits
  - $2^{16} - 1 = 2^{15} + 2^{14} + 2^{13} + 2^{12} + 2^{11} + \cdots + 2^1 + 2^0$
  - -1 is written FFFFH
  - 1+(-1) = 10000H (overflow is out of the 16 bits)
- **Example 2**
  - Notation for -20 on 16 bits
  - $2^{16} - 20 = 65516$ = hard to compute

Two's Complement

- **Method for more easily computing Two's complement**
  - Computing the representation of -X
  - Take the binary representation of X
  - Invert all the bits of X
  - Add one
- **Example 2 (Cont): -20**
  - 20= 0000 0000 0001 0100B
  - We invert all the bits
  - 1111 1111 1110 1011B
  - We add one
  - 1111 1111 1110 1100B
  - Which can be noted in hexadecimal as: FFECH
- **Example 3: -32**
  - 32 = 0000 0000 0010 0000B
  - it makes 1111 1111 1101 1111B
  - We add one: 1111 1111 1110 0000B
  - Representation = FFE0H

Introduction to Floating Point

Floating points

At the beginning of computer science, CPUs only provided integer arithmetic. But, for scienfical uses, it is necessary to use "real numbers". The data types used to *model* $\mathbb{R}$, the real numbers, have special properties that every programmer must know. Most of programming languages (and CPUs and modern GPUs) now support the IEEE 754 norm.

Floating point binary number

Convert decimal into binary

- **For integers**
  Divide by 2, and each time remember the remainder
- **Example: 543**

```
543  |  1
271  |  1
135  |  1
67   |  1
33   |  1
16   |  0
8    |  0
4    |  0
2    |  0
1    |  1
```

543 = 1000011111B

Convert decimal into binary (Cont.)

- **For floating points numbers**
  For the decimal part: multiply each time by two.
- **Example: write 0.25 in binary**

```
0.25 | 0.
0.5  | 0
1    | 1
```

0.25 = 0.01 B

Convert decimal into binary (Cont.)

- **Example 2: write 0.1 in binary**

```
0.1 | 0.
0.2 | 0
0.4 | 0
0.8 | 0
1.6 | 1 -> 0.6
1.2 | 1 -> 0.2
0.4 | 0
0.8 | 0
1.6 | 1 -> 0.6
1.2 | 1 -> 0.2
...
```

$$0.1 = 0.0001100110011\overline{00}B$$

Fixed point numbers

Fixed point numbers

How to represent the decimal number 41.13 with fixed point arithmetic ? Remember that every number in base b can be written as:

$$x = \sum_{i=-\infty}^{\infty} d_i b^i$$

where the d_i are the digits of the numbers and b the base. So the number $(41.13)_{10}$ may be written as the sum:

$$41.13 = 4 * 10^1 + 1 * 10^0 + 1 * 10^{-1} + 3 * 10^{-2}$$

Binary fixed point numbers

As the number 41.13 exist independantly of its representation, it should be possible to write this number using the binary notation.

1. First code the integer part of the number in binary. The decimal value 41 is written $(101001)_2$.
2. Compute the decimal part of the number. The value 0.13 is written as $(0.001000010100011110\ldots)_2$

Finally, we arrive at:

$$(41.13)_{10} = (101001.001000010100011110\ldots)_2$$

Some mathematical considerations

- A rational number ($\mathbb{Q}$) may be written written either with a finite decimal form or with a periodic form. The property "to be rational" is independant of the base used to write the number.
- A rational number may have a finite representation in one base and a periodic infinite representation in another base. For example, the number (1/3) has an infinite periodic represention in base 10, but may be written as $(0.1)_3$ in base 3. The decimal number (0.1) has an infinite periodic representation in base 2. $(0.1)_{10} = (0.0\overline{0011})_2$
- Irrational numbers ($\mathbb{R} \setminus \mathbb{Q}$) have infinite and non-periodic form. Theses numbers does not have a finite representation in any base.

Floating points

Floating points

- The main drawback of fixed point numbers is that their size depends on their magnitude and their precision. But, in engineering, most of the time, one can reduce the precision as the magnitude augment. Engineers use mostly the concept of *significant digits*.
- This notation is usually known as "engineering notation" in pocket calculators. For example one can write:
  - $1.344 \cdot 10^4$ for the number 13440
  - $2.342 \cdot 10^{-5}$ for the number 0.00002342
  - $4.430 \cdot 10^0$ for the number 4.430

  All theses number have 4 significant digits (the mantissa). The "scaling" is done by the exponent of 10. The mantissa is always a number between 1.000 and 9.999.

IEEE 754 representation

The norm IEEE 754 used for binary representation of floating point numbers is based on the same idea.

- The mantissa is a number between 1.0 and 2.0 (smaller than 2.0). The number of significant digits is given by the type of floating point numbers used (`float`, `double`)
- The exponent is now an exponent of 2. It may take positive or negative values.
- The norm introduces special values (negative infinity, positive infinity, not a number, zero) that would be studied later in this chapter.
- The IEEE 754 norm also introduces rounding rules.

Floating numbers: a first example

For this example, the number is coded on 32 bits, which correspond to the `float` format of Java for example.
How would be the decimal number 0.5 represented?
$(0.5)_{10} =$ 0 0111 1110 0000 0000 0000 0000 0000 000

- The first bit (red) is the sign bit. Its value is 0 for positive number and 1 for negative numbers.
- The second group (blue) is the exponent. Here one have the value -1. The exponent is coded as an unsigned integer with a bias. The value of the exponent is computed by the formula valueOfTheField - 127.
- The third field contains the mantissa which here is 0 (there is a hidden bit!)

The exponent

- As explained above, the exponent is coded as an unsigned integer (no two-complement) with a bias which corresponds to the half of the magnitude of the exponent. For example if the exponent is coded on 8 bits, its magnitude is $2^8 = 256$ and the bias would be $2^{8-1} - 1 = 127$.
- Some values of the exponent are reserved to represent special values of numbers. These values are $(00)_{16}$ and $(FF)_{16}$.

The mantissa

- As explained before the matissa contains a value in the interval $1 \leq \text{mantissa} < 2$.
- On this interval, one may see that the first bit is always 1 and therefore is not represented explicitly (it is called the hidden bit). The norm IEEE 754 defines a special format (called denormalized numbers) to change this behavior.
  This hidden bit causes the value of the mantissa to be 0 in the example above.
- The bits of the mantissa represent the sum of negative powers of 2.

Special values

As it was mentioned above, the norm IEEE 754 introduces some special values.

Exponent	Mantissa	Value	Description
00_{16}	$= 0$	0	Zero
00_{16}	$\neq 0$	$\pm 0.m \cdot 2^{-126}$	Denormalized
01_{16} to FE_{16}	any	$\pm 1.m \cdot 2^{e-127}$	Normalized
FF_{16}	$= 0$	$\pm\infty$	$\pm$ Infinity
FF_{16}	$\neq 0$	NaN	Not a Number

Remark: There are two possible representations for the value 0 (+0 and -0)

Comments on special values

- The value zero needs a special representation as it is not possible to represent it using the standard model (see example above).
- Denormalized values are not provided by standard programming languages. Be extremly careful if you need them (performance issues).
- Infinity represents numbers whose magnitude may not be represented in the model. Arithmetical operations where one operand is infinity are well defined by IEEE norm (see next slide).
- Operations where an operand has the value NaN cause an error.

Operations with infinity

Operation	Result
$x/\pm\infty$	0
$\pm\infty \cdot \pm\infty$	$\pm\infty$
$\pm$ non zero / 0	$\pm\infty$
$\infty + \infty$	∞
$\pm 0/\pm 0$	NaN
$\infty - \infty$	NaN
$\pm\infty/\pm\infty$	NaN
$\pm\infty \cdot 0$	NaN

Different types of floating points

The norm IEEE 754 defines two types of floating points: single precision and double precision. They differ only by the number of bits used to store them

	Sign	Exponent	Mantissa	Bias
Single precision	1	8	23	127
Double precision	1	11	52	1023

Rounding

IEEE standard has four different rounding modes. The first is the default, the others are called *directed rounding*.

- **Round to nearest**: rounds to the nearest value. If the number fall in the midway it is rounded to the nearest value with an even (zero) least significant bit.
- **Round toward 0**
- **Round toward** $+\infty$
- **Round toward** $-\infty$

Example of rounding

Rounding mode	+11.5	+12.5	-11.5	-12.5
to nearest, ties to even	+12.0	+12.0	-12.0	-12.0
to nearest, ties away from zero	+12.0	+13.0	-12.0	-13.0
toward 0	+11.0	+12.0	-11.0	-12.0
toward $+\infty$	+12.0	+13.0	-11.0	-12.0
toward $-\infty$	+11.0	+12.0	-12.0	-13.0

Compilation

Compilation

- **Text in, machine code out**
- **Compiler**
  - A program translator that reads source code (C, C++, Pascal, Java, ...)
  - writes out object code files
- **Assembler**
  - A special type of translator
  - Designed for "Assembly language"
  - Characteristic: *Total control over the object code*

Assembly Language

Assembly Language

- **Machine instructions are written in a text file**
  - Readable by humans
- **Mnemonic**
  - Every machine instruction has a "mnemonic"
- **Example**
  - Machine instruction `9CH`
  - Pushes the flags registers onto the stack
  - Mnemonic: `PUSHF`
  - Easier to remember than `9CH`

Assembly source code

- **Arrangement of mnemonics**

```
mov eax,4            ; 04H specifies the sys_write kernel call
mov ebx,1            ; 01H specifies stdout
mov ecx,Message      ; Load starting address of display string into EC
mov edx,MessageLength ; Load the number of chars to display into EDX
int 80H              ; Make the kernel call
```

- **Mnemonic + operands = instruction**

Comments

- **Comments start with ";"**
  - You can write anything after the ";"
  - Each line should have comments
- **Assembly is not Java**
  - Need to comment any action
  - Very difficult to read
- **Beware "write-only" source code**
  - You write code in October
  - You start to learn C in November
  - You need to understand your code in January
  - Impossible to remember the meaning of your program
- **Solution: comment each and every line in assembly**

Compilation

- **Assembler**
  - Transforms assembly language source into an *object module*
- **Object code files can not be run**
  - They need to be linked to other object code
- **One file containing all functionalities would be to large**
  - Source is split in many files
  - Each is responsible for some functionaly

The assembler and the linker

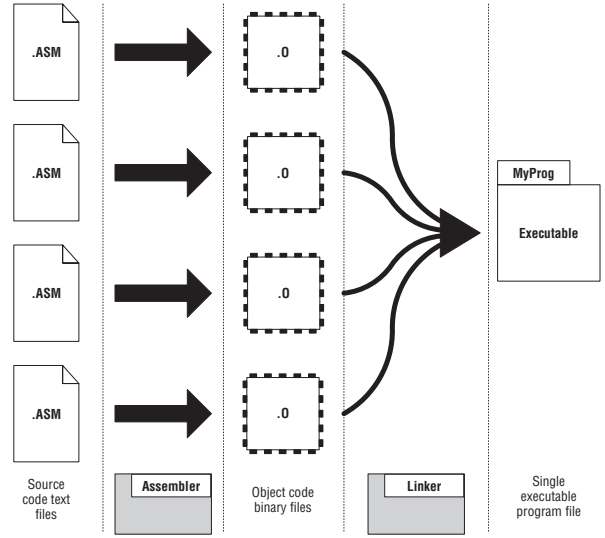


Figure 5-8: The assembler and linker

Object modules

- **The linker seams superfluous**
  - At the begining: only one single file
- **Some parts of programs may be reused**
  - Routines
  - Libraries of functions
  - Tested once, reused many times
- **Object module contains**
  - Program code, including named procedures
  - References to named procedures lying outside the module
  - Named data objects such as numbers and strings with predifined values
  - Named data objects that are just empty space "place holders"
  - Reference to data objects lying outside the module
  - Debugging information
  - Other, less common odds and ends that help the linker create the executable file

Development process

Development process

- **Create your assembly language source**
- **Use your assembler to create an object module from your source file**
- **Use your linker to create a program file**
- **Test the program by running it inside a debugger**
- **Update your code (and repeat the process)**

Development process in Assembler

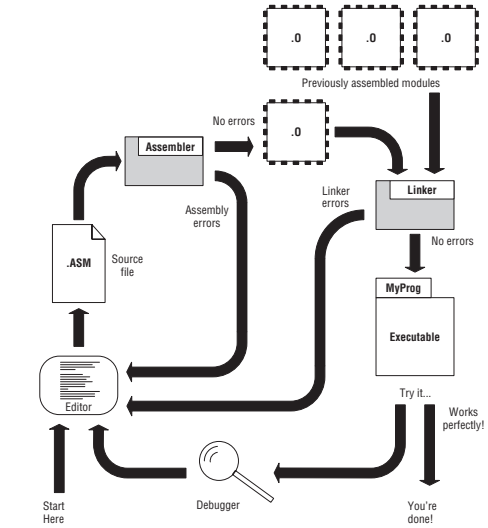


Figure 5-9: The assembly language development process

Development process I

- **Edit the source code file**
  - Using an editor: gEdit for GNU/Linux for instance
  - Save file as a `.asm` file
  - Do not forget comments!
- **Assemble your file**
  - Transform your `.asm` into a `.o`
  - Generates compilation errors
- **Back to the editor**
  - Use error messages to find your syntax errors
  - and try to re-compile
- **Compilation warning messages**
  - Never ignore any warning messages!
  - Even if your code is correct, there us usually a way to improve it

Development process II

- **Link your files**
  - Input: some `.o` files
  - Output: an executable program
  - Linker: `ld` (like load)
- **Linker Errors**
  - Reference to external functions are not right
  - Mostly it is YOUR fault
  - Try to read and understand messages
- **Testing the executable file**
  - Start the program
  - Produces an error (crash of the program)
  - Or a bug (do not do what you want to)

Development process III

- **Debugging using a debugger**
  - Designed to help locate and identify bugs
- **Execute machine instructions**
  - One at a time
  - See the different registers or memory after each step

Compilation instruction

- **Compile the file** `eatsyscall.asm`

```
$ cd BTI7061/asm-4-codification/examples/↘
→helloWorld
$ nasm -f ELF64 -g -F dwarf eatsyscall.asm -↘
→o eatsyscall.o
$ ls
eatsyscall.asm       eatsyscall.o
```

  - `nasm` invokes the assembler
  - `-f ELF64` the .o file will be generated in the "elf" format (64-bit)
  - `-g` debug information is to be included in the .o file
  - `-F dwarf` debug information is to be generated in the "dwarf" format (for use in DDD)
  - `eatsyscall.asm` is the source file
  - `-o eatsyscall.o` output is eatsyscall.o

Link your program with LD

- **Link the file** `eatsyscall.o`

```
$ ld -o eatsyscall eatsyscall.o
$ ls
eatsyscall.asm       eatsyscall.o       ↘
→eatsyscall
```

  - `ld` is the linker
  - `-o eatsyscall` the output file is the executable file eatsyscall if you do not provide an output file, the file a.out is generated
  - `eatsyscall.o` the input file

- **Test the file**

```
$ ./eatsyscall
Eat at Joe's!
```


Debugger

- **Invoke the gdb debugger**
  - Run `$ gdb BINARY` in a shell
  - Use `(gdb) break LABEL` to set a breakpoint
  - Use `(gdb) run` to start the execution
  - Use CTRL-x a to view the source while debugging
  - Use `(gdb) n` to run the next instruction
  - Use `(gdb) print $REG` to inspect a register
- **Invoke the kdgb debugger**
  - KDE App starting in a window

    `$ kdbg eatsyscall`

  - Application more complex to deal with

    `$ ddd eatsyscall`

- **Watch the program execute step by step**
  - See the changes in each register
  - See the changes in memory

Using Make

Using Make

- **Make is used for organizing the compilation of files**
  - In the C world (C, asm, C++)
- **Developer writes a Makefile**
  - Contain the dependencies of files to be compiled, and linked
  - Recompile files only if needed
- **Similar to ANT for Java**

Dependences

- **Object files depend on source files**
  - If the source file (assembler, C, C++) is newer, the `.o` must be recompiled
  - Compare date of `.o` with date of `.asm`, `.c` or `.cpp`
- **Executable depends on object files**
  - The linker will have to relink the files if one has changed.
- **Programmer must define rules**
  - For each file generated, list the file it depends on
  - If not clear, give the command line for its generation

Rules

- **One executable depends on one object (linking is implicit)**

```
eatsyscall:  eatsyscall.o
```

- **One executable depends on many objects (linking is implicit)**

```
linkbase:   linkbase.o   linkparse.o   ↘
→linkfile.o
```

- **Linking can also be explicit**

```
eatsyscall:  eatsyscall.o
     ld -o eatsyscall eatsyscall.o
```

  Second line must be indented by a single tab character at the begining of the line

Compiling a trivial asm file

- **Need two steps**
  - First compile the `.asm` file into a `.o` file
  - Then link the `.o` file into an executable binary
  - The order of the rules is not the order of the execution of the steps.

```
eatsyscall: eatsyscall.o
    ld -o eatsyscall eatsyscall.o
eatsyscall.o: eatsyscall.asm
   nasm -f ELF64 -g -F dwarf eatsyscall.asm
```

Conclusion

- **Encodage**
  - Little Endian: the first "End" contains the "little" bits
  - The light-weight bits are first
- **Real numbers**
  - Are composed of a mantissa and an exponent
- **Compiling**
  - First compile `.asm` into `.o`
  - Link all `.o` files into one executable
  - Process managed typically by `make`

Bibliography

- This course corresponds to chapter *5* and *6* of the course book: **Assembly Language Step by Step (3rd Edition)**
- Course of CPVR speciality: Introduction to Computer perception and virtual reality, Claude Fuhrer BFH-TI (FRC1)