

# Compilation and Program Analysis (#1) : Intro

Laure Gonnord & Matthieu Moy & Gabriel Radanne & other

<https://compil-lyon.gitlabpages.inria.fr/>

Master 1, ENS de Lyon et Dpt Info, Lyon1

2022-2023



# Your teachers



**Figure:** Gabriel Radanne and Ludovic Henrio (CAP)

- Photo © Inria / G .Scagnelli

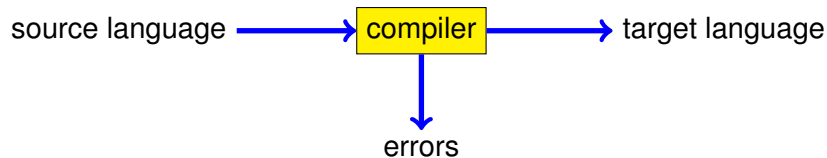
- 1 Intro, what's compilation & what's cool about it
- 2 Compiler phases
- 3 The RISC-V architecture in a nutshell
- 4 One example

# Credits

A large part of the compilation part of this intro course is inspired by the Compilation Course of JC Filliâtre at ENS Ulm who kindly offered the source code of his slides.

Most of the material is shared between the UCBL course “MIF08” (Compilation et Traduction de Programmes), the ENS course “CAP” (Compilation et Analyse de Programme) and ESISAR (Valence).

# What's compilation?



## Compilation toward the machine language

We immediately think of the translation of a high-level language (C,Java,OCaml) into the machine language of a processor (x86, PowerPC...)

```
% gcc -o sum sum.c
```

```
int main(int argc, char **argv) {
    int i, s = 0;
    for (i = 0; i <= 100; i++) s += i*i;
    printf("0*0+...+100*100 = %d\n", s);}

```

→

```
0010011110111101111111111110000010101111101111110000000000010100
10101111101001000000000000010000010101111101001010000000000100100
10101111101000000000000000001100010101111101000000000000000011100
100011111010111000000000000011100

```

## Target Language

This aspect (compilation into assembly) will be presented in this course, but we will do more:

### Compilation is not (only) code generation

A large number of compilation techniques are not linked to assembly code production.

Moreover, languages can be

- interpreted (Basic, COBOL, Ruby, Python, etc.)
- compiled into an intermediate language that will be interpreted (Java, OCaml, Scala, etc.)
- compiled into another high level language (or the same !)
- compiled “on the fly” (or just on time)

# Compiler/ Interpreter

- A compiler translates a program  $P$  into a program  $Q$  such that for all entry  $x$ , the output  $Q(x)$  is the same as  $P(x)$ .

$$\forall P \exists Q \forall x \dots$$

- An interpreter is a program that, given a program  $P$  and an entry  $x$ , computes the output of  $P(x)$ :

$$\forall P \forall x \exists s \dots$$



# Compiler vs Interpreter

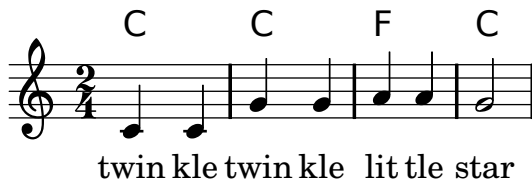
Or :

- The compiler makes a complex work once, to produce a code for whatever entry.
  - An interpreter makes a simpler job, but on every entry.
- In general the code after compilation is more efficient.

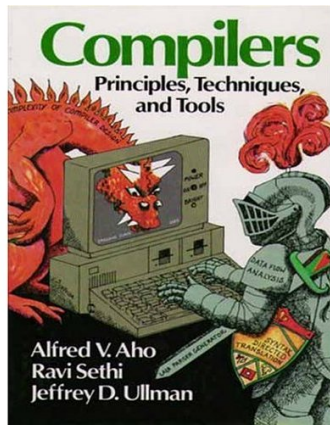
## Example



```
\chords { c2 c f2 c }
\new Staff \relative c' { \time 2/4 c4 c g'4 g a4 a g2 }
\new Lyrics \lyricmode { twin4 kle twin kle lit tle star2 }
```



# Compiler Quality



Quality criteria ?

- correctness
- efficiency of the generated code
- its own efficiency

"Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct."

(Dragon Book, 2006)

# Program Analysis

To prove:

- Correctness of compilers/optimisations phases.
- Correctness of programs: invariants

... also in this course!



# The course

- Syntax Analysis : lexing, parsing, AST, types.
- Evaluators.
- Code generation and optimisations.
- Formal Semantics (many versions!)
- Language extensions.

Goal:

Become familiar with the mechanisms inside a compiler

Understand how to reason about programming languages

# The Lab

A complete compiler for the RISC-V architecture!

⇒ Big guided programming project all along the semester

Support language: Python 3

Goal:

Become able to navigate a medium-sized programming project

Understand the link between theory and implementation

# Grading

- Three graded labs
- One homework
- One final exam

$$Note = \frac{exam + average(Lab3, Lab4, Lab5, homework)}{2}$$

# Course Organization

Everything is on the webpage:

<https://compil-lyon.gitlabpages.inria.fr/>

Read your emails !



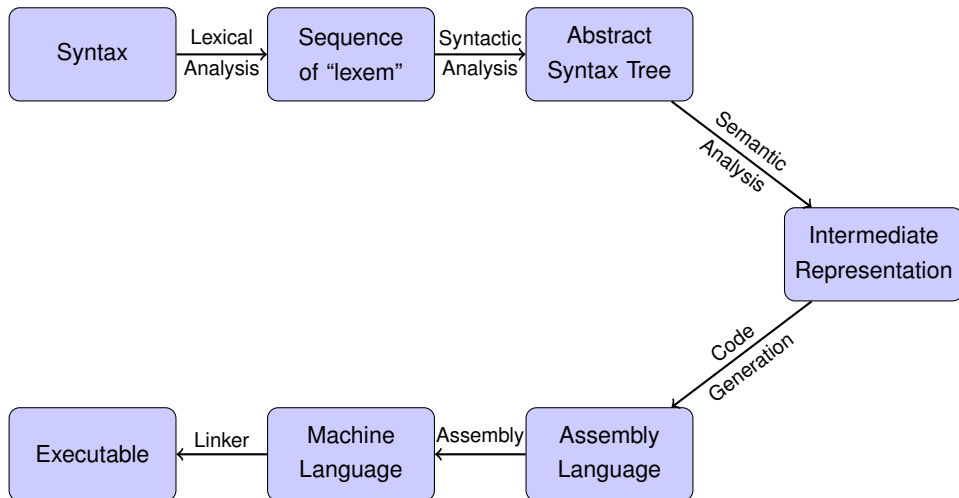
- 1 Intro, what's compilation & what's cool about it
- 2 **Compiler phases**
- 3 The RISC-V architecture in a nutshell
- 4 One example

# Compiler phases

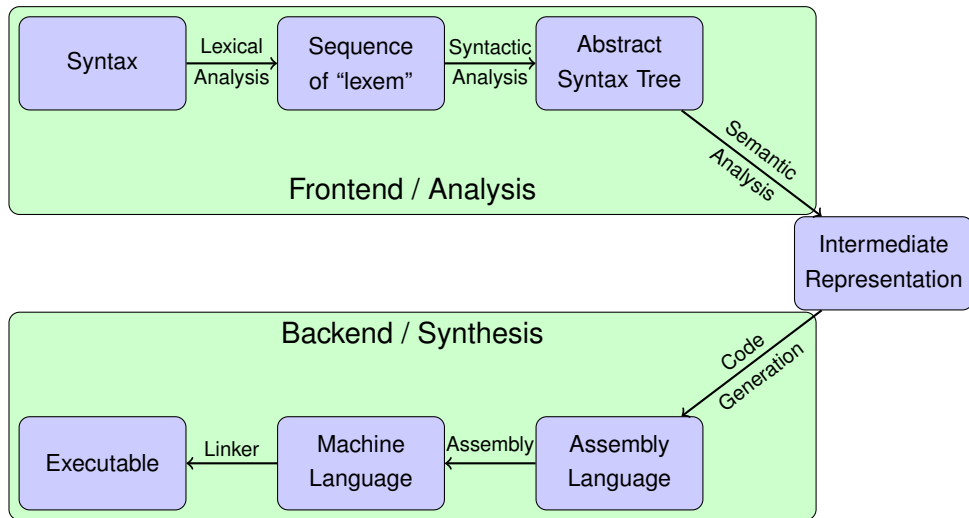
Usually, we distinguish two parts in the design of a compiler:

- an analysis phase:
  - recognizes the program to translate and its meaning.
  - raises errors (syntax, scope, types . . .)
- Then a synthesis phase:
  - produces a target file.
  - sometimes optimises.

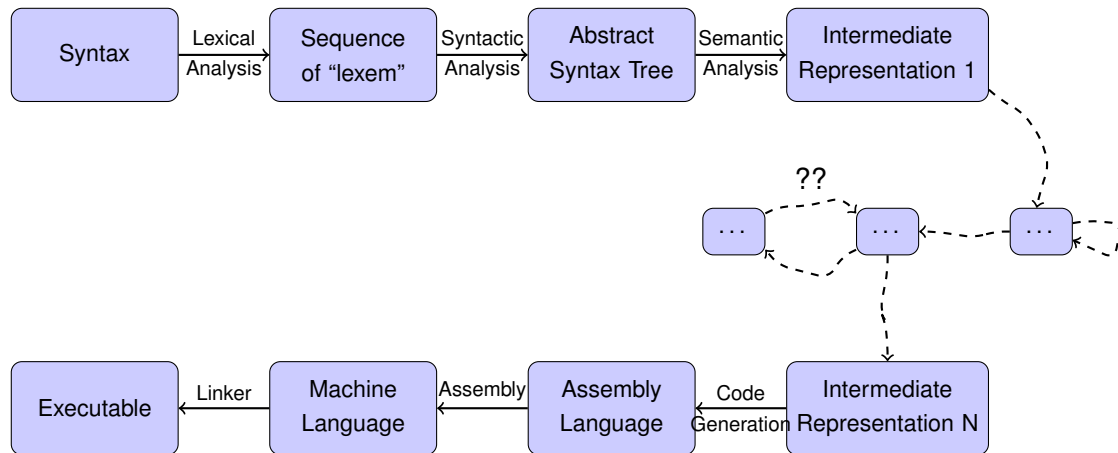
# A Standard™ Compiler Pipeline



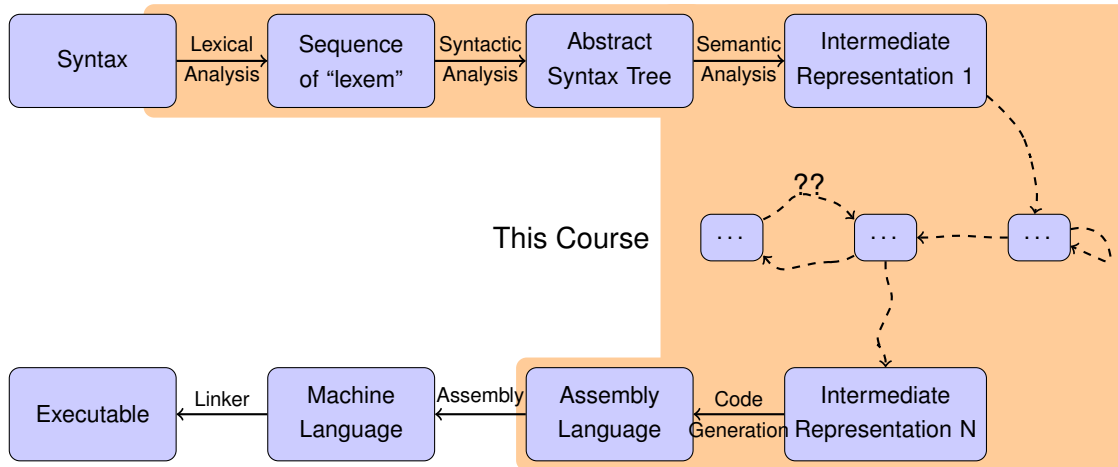
# A Standard™ Compiler Pipeline



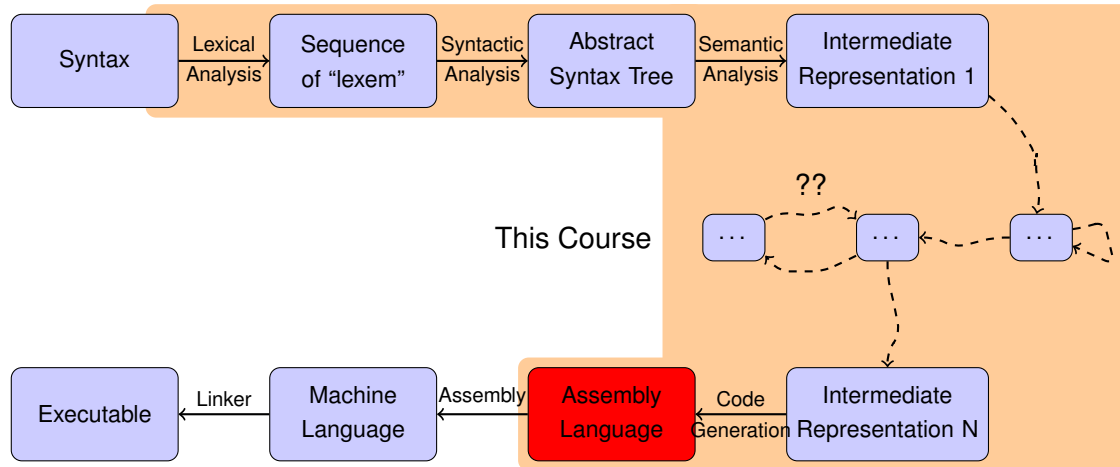
# A Standard™ Compiler Pipeline



# A Standard™ Compiler Pipeline



# A Standard™ Compiler Pipeline



NEXT:

assembly



# Compilation and Program Analysis (#1) : Intro

Laure Gonnord & Matthieu Moy & Gabriel Radanne & other

<https://compil-lyon.gitlabpages.inria.fr/>

Master 1, ENS de Lyon et Dpt Info, Lyon1

2022-2023



- 1 Intro, what's compilation & what's cool about it
- 2 Compiler phases
- 3 The RISC-V architecture in a nutshell**
- 4 One example

## Our target machine : RISC-V

Excerpts from <https://en.wikipedia.org/wiki/RISC-V>

*RISC-V (pronounced "risk-five") is an open-source hardware instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. [...] RISC-V has a modular design, consisting of alternative base parts, with added optional extensions.*

- ▶ We will use a subset of the RV64I Base Integer Instruction Set, 64-bit + some shortcuts.

# RISC-V ecosystem

- Versions of state-of-the-art compilers are available:  
`riscv64-unknown-elf-gcc` for us.
- ISA simulators are available: `spike` for us.

# RISC-V registers

- Memory is addressed as 8-bit bytes
- 64-bit words can be accessed with the load (ld) and store (sd) instructions.
- In the RV64I version, instructions are encoded on 32 (32 as well in the RV32I)
- 32 (64-bit) registers (x0, x1, ...), the first integer register is a zero register, and the rest is general purpose. Registers have symbolic names (sp, fp, ...) to implement standard conventions. **Use only symbolic names when you write code**
- In the base RV64I ISA, there are four core instruction formats (R/I/S/U).

# RISC-V ISA

We provide you an external document with a summary of the ISA.

## Example : ADD instructions

- `add rd, rs1, rs2`, does `rd <- rs1 + rs2`.
  - ↪ All operands are registers.
  - ↪ Example : “`add t1, t2, t3`” executes `t1 <- t2 + t3`.
- `addi rd, rs1, imm`, does `rd <- rs + imm`.
  - ↪ The last operand is an immediate value (on xx bits) encoded in the instruction.
  - ↪ Example : “`addi t1, t2, 5`” executes `t1 <- t2 + 5`.

# RISC-V ADD/ADDi : encoding

R or I-typed instructions

| class                | action                     | encoding   |
|----------------------|----------------------------|--|
| add rd, ri, rj (R)   | $r_d \leftarrow r_i + r_j$ | 00000000 <rj(5bits)> <ri(5bits)> 000 <rd(5bits)> 0110011 |
| addi rd, ri, cte (I) | $r_d \leftarrow r_i + cte$ | <cte(12bits)> <ri(5bits)> 000 <rd(5bits)> 0010011        |

Example: assemble `addi t1, t2, 5`



# RISC-V ADD/ADDi : encoding

R or I-typed instructions

| class                | action                     | encoding   |
|----------------------|----------------------------|--|
| add rd, ri, rj (R)   | $r_d \leftarrow r_i + r_j$ | 00000000 <rj(5bits)> <ri(5bits)> 000 <rd(5bits)> 0110011 |
| addi rd, ri, cte (I) | $r_d \leftarrow r_i + cte$ | <cte(12bits)> <ri(5bits)> 000 <rd(5bits)> 0010011        |

Example: assemble addi t1, t2, 5

cte=5, ri=t2=x7, 000, rd=t1=x6, 0010011

# RISC-V ADD/ADDi : encoding

R or I-typed instructions

| class                | action                     | encoding  |
|----------------------|----------------------------|---|
| add rd, ri, rj (R)   | $r_d \leftarrow r_i + r_j$ | 00000000 <rj (5bits)> <ri (5bits)> 000 <rd (5bits)> 0110011 |
| addi rd, ri, cte (I) | $r_d \leftarrow r_i + cte$ | <cte (12bits)> <ri (5bits)> 000 <rd (5bits)> 0010011        |

Example: assemble addi t1, t2, 5

cte=5, ri=t2=x7, 000, rd=t1=x6, 0010011

0000|0000|0101| 0011|1 000 |0011|0 001|0011

# RISC-V ADD/ADDi : encoding

R or I-typed instructions

| class                | action                     | encoding  |
|----------------------|----------------------------|---|
| add rd, ri, rj (R)   | $r_d \leftarrow r_i + r_j$ | 00000000 <rj (5bits)> <ri (5bits)> 000 <rd (5bits)> 0110011 |
| addi rd, ri, cte (I) | $r_d \leftarrow r_i + cte$ | <cte (12bits)> <ri (5bits)> 000 <rd (5bits)> 0010011        |

Example: assemble addi t1, t2, 5

cte=5, ri=t2=x7, 000, rd=t1=x6, 0010011

0000|0000|0101| 0011|1 000 |0011|0 001|0011

0x00538313

# RISC-V: branching

## Unconditional branching (jump and link):

- `jal rd, c`, does  $rd = PC + 4$ ;  $PC += c$  (focus on PC for the moment)

## Test and branch (branch if lower than, etc.):

- `blt rs1, rs2, c`, does  $PC += c$  if  $rs1 < rs2$
- ▶ Shortcuts: “`j label`” and “`blt rs1, rs2, label`”
- ▶ The label is assembled into the adequate offset of the jump.
- ▶ See the list of operators in `riscv_isa.pdf`

## RISC-V Memory accesses instructions 1/2

- Load from memory (64-bit **double word**)  $r_d \leftarrow Mem[r_s + off]$ :

1 **ld** rd, off(rs)

- Store to memory:

1 **sd** rs, off(rd)

- Load effective **address** (shortcut)

1 **la** rd, **label**

See the ISA for more info.

- 1 Intro, what's compilation & what's cool about it
- 2 Compiler phases
- 3 The RISC-V architecture in a nutshell
- 4 One example

# Ex : Assembly code - demo

---

```

1 #simple RISCv assembly demo
2 #riscv64-unknown-elf-gcc demo20.s ../../TP/TP01/riscv/libprint.s -o demo20
3 #spike pk demo20
4     .text
5     .globl main
6 main:
7     addi    sp,sp,-16
8     sd      ra,8(sp)
9     # your assembly code here
10    addi    t1, zero, 5           # first op : cte
11    la      t3, mydata           # second, from memory
12    ld      t4, 0(t3)
13    add     a0, t1, t4           # add --> a0 = result
14    call    print_int
15    call    newline
16    ## /end of user assembly code
17    ld      ra,8(sp)
18    addi    sp,sp,16
19    ret
20    .section .rodata
21 mydata:
22    .dword 37

```

---

## Exercise: RISC-V Assembly

Assemble the following instruction:

```
mv t1, s1
```



## Exercise: RISC-V Assembly

Assemble the following instruction:

```
mv t1, s1
```

Shortcut for: `addi t1, s1, 0`

## Exercise: RISC-V Assembly

Assemble the following instruction:

```
mv t1, s1
```

Shortcut for: `addi t1, s1, 0`

Format I

## Exercise: RISC-V Assembly

Assemble the following instruction:

```
mv t1, s1
```

Shortcut for: `addi t1, s1, 0`

Format I

| imm[11:0] | rs1 | funct3 | rd | opcode |
|-----------|-----|--------|----|--------|
|           |     |        |    |        |

## Exercise: RISC-V Assembly

Assemble the following instruction:

```
mv t1, s1
```

Shortcut for: `addi t1, s1, 0`

Format I

| imm[11:0]      | rs1    | funct3 | rd     | opcode   |
|----------------|--------|--------|--------|----------|
| 0              | 9      | 0      | 6      | 0010011  |
| 0000 0000 0000 | 0100 1 | 000    | 0011 0 | 001 0011 |

## Exercise: RISC-V Assembly

Assemble the following instruction:

```
mv t1, s1
```

Shortcut for: `addi t1, s1, 0`

Format I

| imm[11:0]      | rs1    | funct3 | rd     | opcode   |
|----------------|--------|--------|--------|----------|
| 0              | 9      | 0      | 6      | 0010011  |
| 0000 0000 0000 | 0100 1 | 000    | 0011 0 | 001 0011 |
| 0x00048313     |        |        |        |          |

## Exercise: RISC-V Disassembly

Disassemble the following instruction: 0x00810383

## Exercise: RISC-V Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

## Exercise: RISC-V Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011



## Exercise: RISC-V Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011

Type I, lb, lw or lbu

## Exercise: RISC-V Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011

Type I, lb, lw or lbu

| imm[11:0]      | rs1    | funct3 | rd     | opcode   |
|----------------|--------|--------|--------|----------|
| 0000 0000 1000 | 0001 0 | 000    | 0011 1 | 000 0011 |

## Exercise: RISC-V Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011

Type I, lb, lw or lbu

| imm[11:0]      | rs1    | funct3 | rd     | opcode   |
|----------------|--------|--------|--------|----------|
| 0000 0000 1000 | 0001 0 | 000    | 0011 1 | 000 0011 |

Func3 = 0  $\Rightarrow$  lb; rs = x2 = sp; rd = x7 = t2

## Exercise: RISC-V Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011

Type I, lb, lw or lbu

| imm[11:0]      | rs1    | funct3 | rd     | opcode   |
|----------------|--------|--------|--------|----------|
| 0000 0000 1000 | 0001 0 | 000    | 0011 1 | 000 0011 |

Func3 = 0  $\Rightarrow$  lb; rs = x2 = sp; rd = x7 = t2

lb t2, 8(sp)

## Exercise: RISC-V Program

Write a program that counts from 0 to infinity.

We provide `call print_int` (to display `a0` as an integer) and `call newline`.

## Exercise: RISC-V Program

Write a program that counts from 0 to infinity.

We provide call `print_int` (to display `a0` as an integer) and call `newline`.

```
.globl main
main:
    li a0, 0
lbl:
    call print_int
    call newline
    addi a0, a0, 1
    j lbl
```