Introduction - Compilation Courses - 2020

Laure Gonnord & Matthieu Moy & other

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

Master 1, ENS de Lyon et Dpt Info, Lyon1

2021-2022

Your teachers

Figure: Laure Gonnord (Esisar) - Matthieu Moy (MIF08) - Gabriel Radanne (CAP)

- Photo © Inria / G .Scagnelli

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.

What's compilation?

$$\begin{tikzcd}
\text{source language} \arrow[r] & \boxed{\text{compiler}} \arrow[r] \arrow[d] & \text{target language} \\
& \text{errors} &
\end{tikzcd}$$

Compilation toward the machine language

We immediatly think of the translation of a high-level language (C,Java,OCaml) into the machine language of a processor (Pentium, 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);}
```

$\longrightarrow$

```
0010011110111101111111111110000010101111101111110000000000010100
1010111110100100000000000010000010101111101001010000000000100100
1010111110100000000000000001100010101111101000000000000000011100
10001111101011110000000000011100
```

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 intro a program Q such that for all entry x, the output $Q(x)$ is the same as $P(x)$.

$$\forall P \; \exists Q \; \forall x ...$$

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

$$\forall P \; \forall x \; \exists s ...$$

Compiler vs Interpreter

Or :

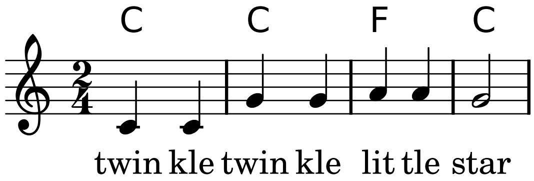
- The compiler makes a complex work once, to produce a code for whatever entry.
- An interpreter makes a simpler job, but on every entry.

► In general the code after compilation is more efficient.

Example

$$\begin{tikzcd}\text{source} \arrow[r] & \boxed{\text{lilypond}} \arrow[r] & \text{PostScript file} \arrow[r] & \boxed{\text{gs}} \arrow[r] & \text{image}\end{tikzcd}$$

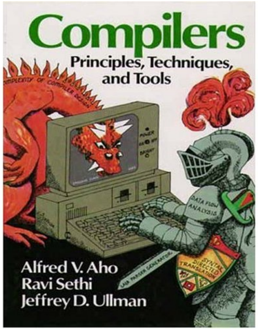
```
\chords { c2 c f2 c }
\new Staff \relative c' { \time 2/4 c4 c g'4 g a4 a g2 }
\new Lyrics \lyricmode { twin4 kle twin kle lit tle star2
```



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.”

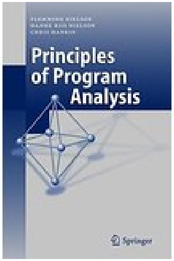
(Dragon Book, 2006)

Program Analysis ENSL only

To prove:

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

. . . the second part of the course.



Course Objective

Be familiar with the mechanisms inside a (simple) compiler.

ENSL only Be familiar with basis of program analysis.

- And understand the links between them!

Course Content - Compilation Part

- Syntax Analysis : lexing, parsing, AST, types.
- Evaluators.
- Code generation.
- Code Optimisation.

Lab : a complete compiler for the RISCV architecture !

Support language: Python 3

Frontend infrastructure : ANTLR 4.

Course Content - Analysis Part ENSL Only!

- Concrete semantics (many versions!)
- Analysis on SSA
- Language extensions

Labs : Language extensions.

Support language: Python 3.

Course Organization

Everything is on the webpage:

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

Read your emails !

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.

Analysis Phase

$$\begin{tikzcd}
\text{source code} \arrow[d] \\
\boxed{\text{lexical analysis}} \arrow[d] \\
\text{sequence of “lexems” (\textit{tokens})} \arrow[d] \\
\boxed{\text{syntactic analysis (Parsing)}} \arrow[d] \\
\text{abstract syntax tree (\textit{AST})} \arrow[d] \\
\boxed{\text{semantic analysis}} \arrow[d] \\
\text{abstract syntax (+ symbol table)}
\end{tikzcd}$$

Synthesis Phase

$$\begin{tikzcd}
\text{abstract syntax} \arrow[d] \\
\boxed{\text{code production (numerous phases)}} \arrow[d] \\
\text{assembly language} \arrow[d] \\
\boxed{\text{assembly } (\texttt{as})} \arrow[d] \\
\text{machine language} \arrow[d] \\
\boxed{\text{linker } (\texttt{ld})} \arrow[d] \\
\text{executable code}
\end{tikzcd}$$

NEXT:

<u>assembly</u>

Introduction - Compilation Courses - 2020

Laure Gonnord & Matthieu Moy & other

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

Master 1, ENS de Lyon et Dpt Info, Lyon1

2021-2022

Our target machine : RISCV

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.

RISCV ecosystem

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

RISCV 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, the first integer register is a zero register, and the rest is general purpose (but have symbolic names 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).

RISCV 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`.

RISCV ADD/ADDi : encoding

R or I-typed instructions

class	action	encoding
add rd, ri, rj (R)	$r_d \leftarrow r_i + r_j$	`0000000|<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`

RISCV: branching

Unconditional branching:

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

Test and branch :

- `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 the companion sheet.

RISCV Memory accesses instructions 1/2

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

```
ld rd, off(rs)
```

- Store to memory:

```
sd rs, off(rd)
```

- Load effective address (shortcut)

```
la rd, label
```

See the ISA for more info.

Ex : Assembly code - demo

```
#simple RISCV assembly demo
#riscv64-unknown-elf-gcc  demo20.s ../../TP/TP01/code/libprint.s -o demo20
#spike pk demo20
        .text
        .globl main
main:
        addi    sp,sp,-16
        sd      ra,8(sp)
# your assembly code here
        addi t1, zero, 5            # first op : cte
        la t3, mydata               # second, from memory
        ld t4, 0(t3)
        add a0, t1, t4              # add --> a0 = result
        call print_int
        call newline
## /end of user assembly code
        ld      ra,8(sp)
        addi    sp,sp,16
        ret
        .section .rodata
mydata:
        .dword 37
```

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
addi t1, s1, 0
```

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
addi t1, s1, 0
```

Format I

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
addi t1, s1, 0
```

Format I

imm[11:0]	rs1	funct3	rd	opcode

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
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: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
addi t1, s1, 0
```

Format I

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

Exercise: RISCV Disassembly

Disassemble the following instruction: 0x00810383

Exercise: RISCV Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Exercise: RISCV Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011

Exercise: RISCV Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011

Type I, lb, lw or lbu

Exercise: RISCV 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: RISCV 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: RISCV 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 ⇒ lb; rs = `x2` = `sp`; rd = `x7` = `t2`

```
lb t2, 8(sp)
```

Exercise: RISCV 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: RISCV 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
```