INF108: Compilation

Louis Jachiet

Introduction

Introduction

Who am I?

Who am I?

Louis Jachiet

Associate Professor

Teaching:

- Databases
- Competitive programming
- Big data framework (Hadoop, Spark, etc.)
- And now Compilation!

Research:

- Fine grained complexity of algorithms in the RAM model
- Query evaluation on big data platforms
- Efficient evaluation of queries on words and graphs

Who are you?

Please answer honestly, the form won't be used for grading!

Introduction

Overview

The main questions this course tackles

How is it possible to "reason" starting from simple transistors?

The main questions this course tackles

CPU can only run very simple programs, how does one makes functions, objects, etc.?

The main questions this course tackles

Computer programs can be written using high-level syntax, how can we easily write programs that "understand" code?

The main questions this course tackles

Chapter II - From logical gates to a CPU

How is it possible to “reason” starting from simple transistors?

Chapter I - Running programs on a CPU

CPU can only run very simple programs, how does one makes functions, objects, etc.?

Chapter III - Making sense of computer programs

Computer programs can be written using high-level syntax, how can we easily write programs that “understand” code?

The main questions this course tackles

Chapter II - From logical gates to a CPU

How is it possible to “reason” starting from simple transistors?

Chapter I - Running programs on a CPU

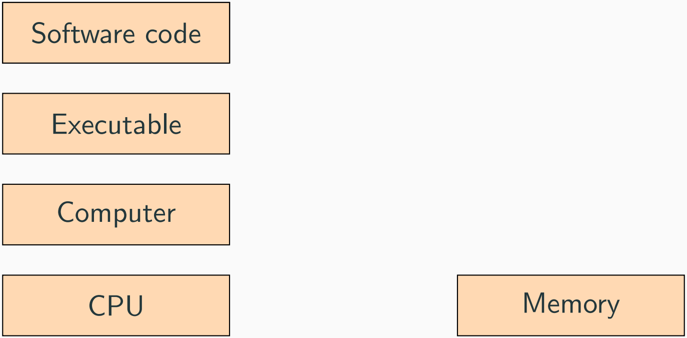
CPU can only run very simple programs, how does one makes functions, objects, etc.?

Chapter III - Making sense of computer programs

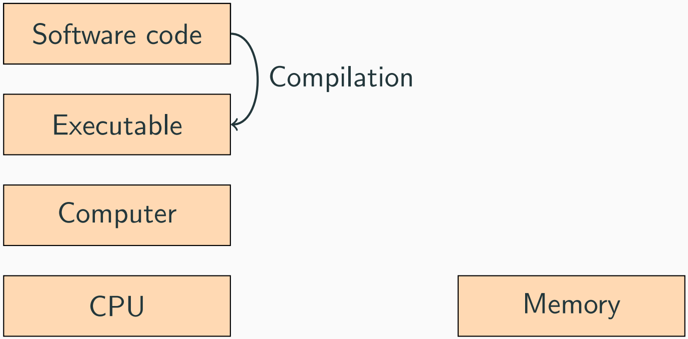
Computer programs can be written using high-level syntax, how can we easily write programs that “understand” code?

Chapter IV - Making a compiler!

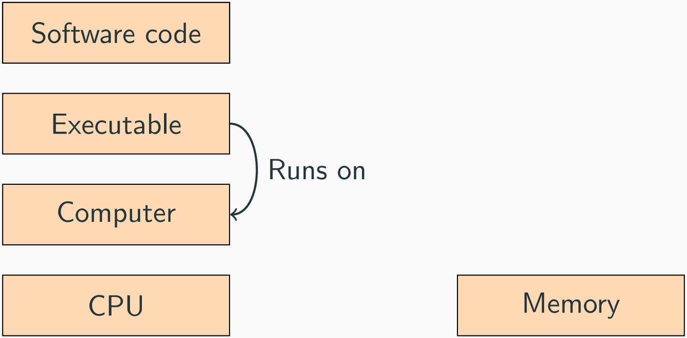
Abstraction



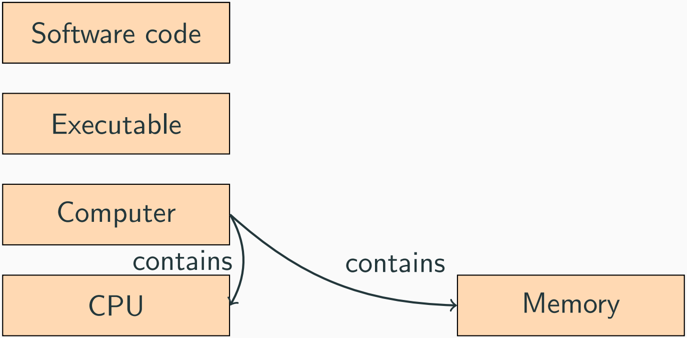
Abstraction



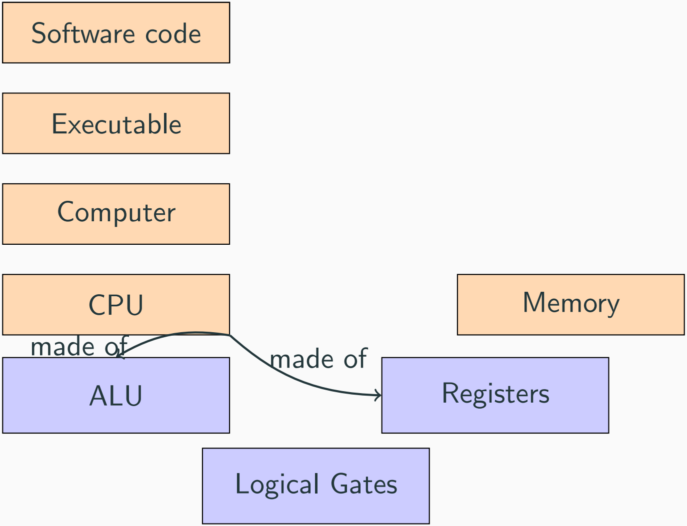
Abstraction



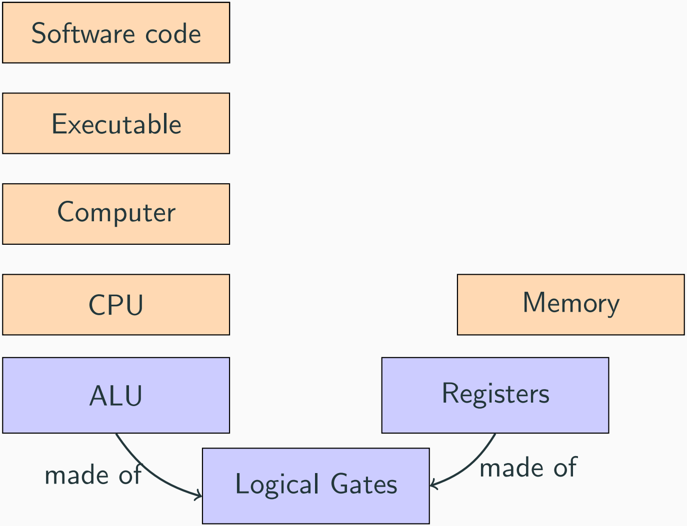
Abstraction



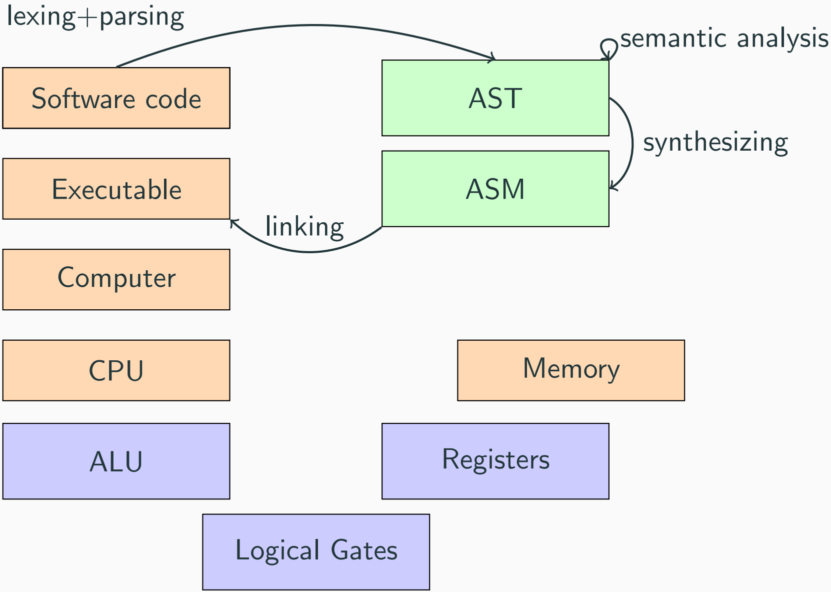
Abstraction



Abstraction



Abstraction



C code

```c
#include <stdio.h>

int fibo(int n) {
  if(n<=1) return 1;
  return fibo(n-1)+fibo(n-2);
}

int main () {
  printf("%d\n", fibo(12));
  return 0;
}
```

Assembly code (x86)

```
	.file	"a.c"
	.text
	.globl	fibo
	.type	fibo, @function
fibo:
.LFB0:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	pushq	%rbx
	subq	$24, %rsp
	.cfi_offset 3, -24
	movl	%edi, -20(%rbp)
	cmpl	$1, -20(%rbp)
	jg	.L2
	movl	$1, %eax
	jmp	.L3
.L2:
	movl	-20(%rbp), %eax
	subl	$1, %eax
	movl	%eax, %edi
	call	fibo
	movl	%eax, %ebx
	movl	-20(%rbp), %eax
	subl	$2, %eax
	movl	%eax, %edi
	call	fibo
	addl	%ebx, %eax
.L3:
	movq	-8(%rbp), %rbx
	leave
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE0:
	.size	fibo, .-fibo
	.section	.rodata
.LC0:
	.string	"%d\n"
	.text
	.globl	main
	.type	main, @function
main:
.LFB1:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movl	$12, %edi
	call	fibo
	movl	%eax, %esi
	leaq	.LC0(%rip), %rdi
	movl	$0, %eax
	call	printf@PLT
	movl	$0, %eax
	popq	%rbp
	.cfi_def_cfa 7, 8
	ret
	.cfi_endproc
.LFE1:
	.size	main, .-main
	.ident	"GCC: (Debian 10.2.1-6) 1
	.section	.note.GNU-stack,"",@pro
```

gcc -s fibo.c

Binary (ELF)

```
7f45 4c46 0201 0100 0000 0000 0000 0000
0300 3e00 0100 0000 5010 0000 0000 0000
4000 0000 0000 0000 7839 0000 0000 0000
0000 0000 4000 3800 0b00 4000 1e00 1d00
0600 0000 0400 0000 4000 0000 0000 0000
4000 0000 0000 0000 4000 0000 0000 0000
6802 0000 0000 0000 6802 0000 0000 0000
0800 0000 0000 0000 0300 0000 0400 0000
a802 0000 0000 0000 a802 0000 0000 0000
a802 0000 0000 0000 1c00 0000 0000 0000
1c00 0000 0000 0000 0100 0000 0000 0000
0100 0000 0400 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
6805 0000 0000 0000 6805 0000 0000 0000
```

Why this course?

- Improve programming skills (practice & understanding)
- Learn project management (on a small project)
- Get notions of how a simple CPU works
- Understand compilers where they shine and where you can help them
- Being able to write (simple) compilers

Introduction

Terminology

Compiler

A *compiler* is any program that transforms "programs" from a *source* language to a *target* language:

- usually from high-level (C/C++/Scala/etc.) to machine code;

Compiler

A *compiler* is any program that transforms “programs” from a *source* language to a *target* language:

- usually from high-level (C/C++/Scala/etc.) to machine code;
- but it can target intermediate language (e.g. java to JVM or ocaml bytecode);

Compiler

A *compiler* is any program that transforms “programs” from a *source* language to a *target* language:

- usually from high-level (C/C++/Scala/etc.) to machine code;
- but it can target intermediate language (e.g. java to JVM or ocaml bytecode);
- or can be interpreted in a broad sense (e.g. LaTeX to PDF);

Compiler

A *compiler* is any program that transforms “programs” from a *source* language to a *target* language:

- usually from high-level (C/C++/Scala/etc.) to machine code;
- but it can target intermediate language (e.g. java to JVM or ocaml bytecode);
- or can be interpreted in a broad sense (e.g. LaTeX to PDF);
- and can sometimes work in a JIT fashion.

Compilers: important properties

Compilers should be correct

The semantics of programs must be preserved.

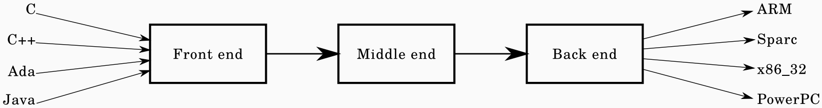
Compilers should produce fast code

Any gain in efficiency of the produced translates to efficiency gains for all programs.

Compilers should be fast

Source codes can be very large!

Compiler schema



Schematic 3 phases representation of compiler as depicted on Wikipedia

First phase: lexical Analysis

- First step of a compiler
- Splits the input text into individual a list of tokens (usually with regexps)

```
int fibo(int n)
{
  if(n<=1)
    return 1;
  return
     fibo(n-1)
   + fibo(n-2);
}
```

```
INT ID(``fibo'') LPAR INT ID(``n'') RPAR
LBRACKET
   IF LPAR ID(``n'') LEQ CST(1) RPAR
     RETURN CST(1) SEMICOL
  RETURN
   ID(``fibo'') LPAR ID(``n'') MINUS CST(1) RPAR
  PLUS
   ID(``fibo'') LPAR ID(``n'') MINUS CST(2) RPAR
  SEMICOL
RBRACKET
```

First phase: Parsing / Syntactic Analysis

- Second step of a compiler
- Organize tokens into a Abstract Syntax Tree (usually based on grammars)

```
int fibo(int n)
{
  if(n<=1)
    return 1;
  return
     fibo(n-1)
   + fibo(n-2);
}
```

```
Function
(
   "fibo",
   [("n",Tint)],
   Sequence(
     If(
        Less(Var "n", Cst 1),
        Return (Cst 1)),
     Return
       Plus(
        Call("fibo",[Minus(Var "n", Cst 1)]),
        Call("fibo",[Minus(Var "n", Cst 2)])
       )
   )
)
```

First phase: Semantical Analysis

- Annotate the AST
- Check variables/functions definitions
- Check type information

```
int fibo(int n)
{
  if(n<=1)
    return 1;
  return
     fibo(n-1)
   + fibo(n-2);
}
```

```
Function
(
   "fibo",
   [("n", Tint)],
   Sequence(
     If(
        Less(Var "n", Cst 1),
        Return (Cst 1)),
     Return
       Plus(
        Call("fibo",[Minus(Var "n", Cst 1)]),
        Call("fibo",[Minus(Var "n", Cst 2)])
       )
   )
)
```

Second phase: Dead code elimination

```
int fibo(int n)
{
  if(n<=1)
    return 1;
  if(false)
    printf("Computing fib %d\n",n);
  return
     fibo(n-1)
   + fibo(n-2);
}
```

Second phase: Constant folding

```
const int days_in_year = 365 ;
const int hours_in_day = 24 ;
const int min_in_hours = 60 ;
const int seconds_in_hours = 60 ;

int main() {
  print("There are %d seconds in an ordinary year!",
   days_in_year * hours_in_day *
   min_in_hours * seconds_in_hours );
}
```

Second phase: Aliasing Analysis

```c
int f(int * a, int * b) {
  *a = 42;
  *b = 12;
  return *a;
}
```

Second phase: Liveliness Analysis

```
int f(int a) {
  int x = f(a) ;
  int y = f(x) ;
  int z = f(y) ;
  return *a;
}
```

Second phase: many others analysis

- Available expression
- Common subexpression elimination
- Dead store elimination
- Induction variable
- Data Flow Analysis
- ...

Tentative organization of the course

Tools used in the class

Programming in OCaml

- ocaml
- dune
- menhir
- ocamllex

Programming in MIPS

- spim, xspim

Installing on Debian-based linux:

```
apt-get install menhir menhir-doc libmenhir-ocaml-dev \
        ocaml-dune spim
```

Homework

Projects

P1	⋆⋆	Simple AST to MIPS	26/09
P2	⋆⋆⋆	CPU	03/10
P3	⋆	PtiPython interpreter	10/10
P4	⋆⋆⋆⋆⋆	Compiler (ptitC to MIPS)	18/10 & 31/10

Organization

- For the harder projects, you can work alone or in a team of two
- The less competent should be the one coding
- Copy of code between groups is forbidden
- A large part of the projects can be done in class
- Test & version your code!

Submitting projects

Deadline

All projects need to be submitted before 18:00 the day of the deadline. The only valid excuse for a late project is a medical certificate.

Content

You will generally have to submit a zip file with a folder containing your data. The zip file should not contain `.DS_Store` or compiler generated files (`.o`, `.exe`, the `_build` folder, etc.). The program should compile without manual intervention.

Test

You are invited to include your own tests in the submission. Good tests will be rewarded.

Grading

Final exam

$F =$ grade of the 1.5h exam with “high level” questions.

Three small projects

$S =$ average of **P1** AST to MIPS, **P2** CPU and **P3** PtiPython.

One big project

$C =$ grade for the compiler

Final grade:

$$max\left(\frac{S+F+C}{3}, \frac{2}{5}\times(S+F+C-min(S,F,C))\right)$$

Course organization

Date	Homework	Content
13/09		MIPS
19/09	P1	From a simple AST to MIPS
20/09		Logic gates
26/09	P2	CPU
27/09		Parsing / Lexing / Typing
03/10	P3	Semantics and Interpreter
04/10		Advanced topics in compilation part 1
10/10		Compiler Project part 1 (lexer/parser)
11/10	P4-1	Advanced topics in compilation part 2
17/10		Compiler Project part 2
18/10		Filler (OS?)
24/10		Advanced topics in compilation part 3
25/10	P4-2	Compiler Project part 3