Interpretation and Compilation of Languages

Master Programme in Computer Science

Mário Pereira   mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

April 22, 2025

Lecture 7

based on lectures by Jean-Christophe Filliâtre and Léon Gondelman
previous editions by João Costa Seco, Luís Caires, and Bernardo Toninho

The Road so Far

We have reached the mid-pint of our course.

We have studied the frontend part of a compiler.

- operational semantics
- lexical analysis
- syntactic analysis
- static typing

We are now going to focus on the backend part of a compiler.

1. Presentation of the practical project.

Today: `x86-64` Assembly

2. `x86-64` Architecture
3. `x86-64` Instruction Set
4. The Challenge of Compilation

A Little Bit of Computer Arithmetic (recap)

An integer is represented using n bits,
written from right (least significant) to left (most significant)

b_{n-1}	b_{n-2}	...	b_1	b_0

typically, n is 8, 16, 32, or 64.

$$\begin{aligned} \text{bits} &= b_{n-1} b_{n-2} \dots b_1 b_0 \\ \text{value} &= \sum_{i=0}^{n-1} b_i 2^i \end{aligned}$$

bits	value
$000 \dots 000$	0
$000 \dots 001$	1
$000 \dots 010$	2
$\vdots$	$\vdots$
$111 \dots 110$	$2^n - 2$
$111 \dots 111$	$2^n - 1$

example: $00101010_2 = 42$

The most significant bit b_{n-1} is the sign bit

$$\begin{aligned}\text{bits} &= b_{n-1} b_{n-2} \dots b_1 b_0 \\ \text{value} &= -b_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i\end{aligned}$$

example:

$$\begin{aligned}11010110_2 &= -128 + 86 \\ &= -42\end{aligned}$$

bits	value
$100\dots000$	-2^{n-1}
$100\dots001$	$-2^{n-1}+1$
$\vdots$	$\vdots$
$111\dots110$	-2
$111\dots111$	-1
$000\dots000$	0
$000\dots001$	1
$000\dots010$	2
$\vdots$	$\vdots$
$011\dots110$	$2^{n-1}-2$
$011\dots111$	$2^{n-1}-1$

Beware!

According to the context, the same bits are interpreted either as a signed or unsigned integer.

example:

- $11010110_2 = -42$ (signed 8-bit integer)
- $11010110_2 = 214$ (unsigned 8-bit integer)

The machine provide operations such as

- logical (aka bitwise) operations: and, or, xor, not
- shift operations
- arithmetic operations: addition, subtraction, multiplication, etc.

operation		example
negation	x	00101001
	not x	11010110
and	x	00101001
	y	01101100
	x and y	00101000
or	x	00101001
	y	01101100
	x or y	01101101
xor	x	00101001
	y	01101100
	x xor y	01000101

Shift Operation

- logical shift left (inserts least significant zeros)

$$\leftarrow \begin{array}{|c|c|c|c|c|c|}\hline b_{n-3} & \ldots & b_1 & b_0 & 0 & 0 \\ \hline\end{array} \leftarrow$$

(« in Java, `lsl` in OCaml)

- logical shift right (inserts most significant zeros)

$$\rightarrow \begin{array}{|c|c|c|c|c|c|}\hline 0 & 0 & b_{n-1} & \ldots & b_3 & b_2 \\ \hline\end{array} \rightarrow$$

(»> in Java, `lsr` in OCaml)

- arithmetic shift right (duplicates the sign bit)

$$\rightarrow \begin{array}{|c|c|c|c|c|c|}\hline b_{n-1} & b_{n-1} & b_{n-1} & \ldots & b_3 & b_2 \\ \hline\end{array} \rightarrow$$

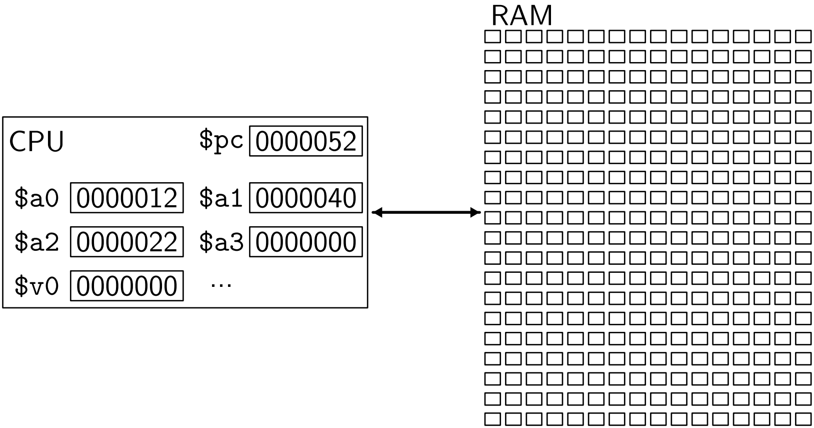
(»> in Java, `lsr` in OCaml)

A Little Bit of Architecture

Roughly speaking, a computer is composed

- of a CPU, containing
  - few integer and floating-point registers
  - some computation power

- memory (RAM)
  - composed of a large number of bytes (8 bits)
    for instance, 1 GiB $= 2^{30}$ bytes $= 2^{33}$ bits, that is $2^{2^{33}}$ possible states
  - contains data and instructions



accessing memory is costly (at one billion instructions per second, light only traverses 30 centimeters!)

reality is more complex:

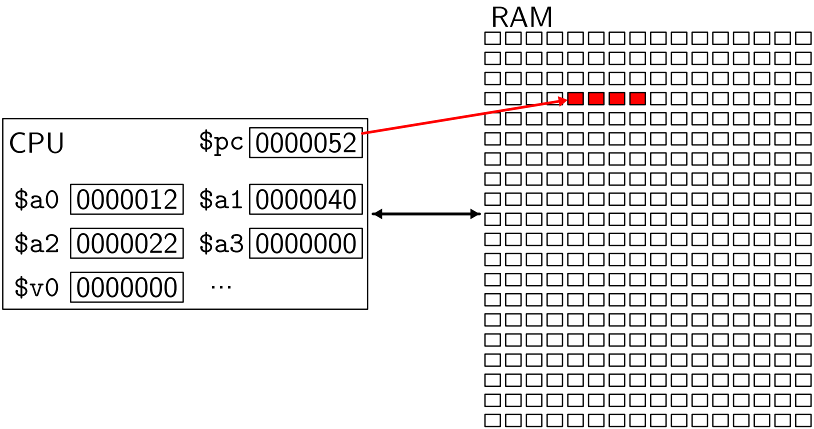
- several (co)processors, some dedicated to floating-point
- one or several memory caches
- virtual memory (MMU)
- etc.

Execution Principle

Execution proceeds according to the following:

- a register (%pc) contains the address of the next instruction to execute
- we read one or several bytes at this address (*fetch*)
- we interpret these bytes as an instruction (*decode*)
- we execute the instruction (*execute*)
- we modify the register %pc to move to the next instruction (typically the one immediately after, unless we jump)

Execution Principle



instruction :	001000	00110	00101	0000000000001010
decoding :	addi	%a2	%a1	10

i.e. add 10 to register %a2 and store the result in the register %a1

Again, reality is more complex:

- pipelines
  - several instructions are executed in parallel
- branch prediction
  - to optimize the pipeline, we attempt at predicting conditional branches

Which Architecture for this Course?

two main families of microprocessors

- CISC (*Complex Instruction Set*)
  - many d'instructions
  - many addressing modes
  - many instructions read / write memory
  - few registers
  - examples: VAX, PDP-11, Motorola 68xxx, Intel x86
- RISC (*Reduced Instruction Set*)
  - few instructions
  - few instructions read / write memory
  - many registers
  - examples: Alpha, Sparc, MIPS, ARM

We choose `x86-64` for this course.

x86-64 Architecture

- 64 bits
  - arithmetic, logical, and transfer operations over 64 bits
- 16 registers
  - `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rbp`, `%rsp`, `%rsi`, `%rdi`, `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14`, `%r15`
- addresses memory over at least 48 bits ($\geq$ 256 TB)
- many addressing modes

We do not code in machine language, but using the assembly language.

The assembly language provides several facilities:

- symbolic names
- allocation of global data

Assembly language is turned into machine code by a program called an assembler (a compiler).

We are using Linux and GNU tools.
In particular, GNU assembly, with AT&T syntax.

x86-64 Program

The assembly directive

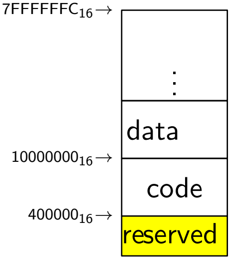
```
        .text
```

indicates that the instructions will follow and the directive

```
        .data
```

indicates that the data will follow

The code will be loaded starting from the address `0x400000` and the data from the address `0x10000000`.



```
        .text                           # instructions follow
        .globl main                     # make main visible for ld
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    $message, %rdi          # argument of puts
        call    puts
        movq    $0, %rax                # return code 0
        popq    %rbp
        ret

        .data                           # data follow
message:
        .string "hello, world!" # 0-terminated string
```

Demo!

A step-by-step execution is possible using gdb (*the GNU debugger*)

```
> gcc -g -no-pie hello.s -o hello
> gdb hello
GNU gdb (GDB) 7.1-ubuntu
...
(gdb) break main
Breakpoint 1 at 0x401126: file hello.s, line 4.
(gdb) run
Starting program: .../hello

Breakpoint 1, main () at hello.s:4
4               pushq   %rbp
(gdb) step
5               movq    %rsp, %rbp
(gdb) info registers
...
```

An alternative is Nemiver

```
> nemiver hello
```

Instruction Set

63	31	15	8 7	0
%rax	%eax	%ax	%ah	%al
%rbx	%ebx	%bx	%bh	%bl
%rcx	%ecx	%cx	%ch	%cl
%rdx	%edx	%dx	%dh	%dl
%rsi	%esi	%si		%sil
%rdi	%edi	%di		%dil
%rbp	%ebp	%bp		%bpl
%rsp	%esp	%sp		%spl

63	31	15	8 7	0
%r8	%r8d	%r8w		%r8b
%r9	%r9d	%r9w		%r9b
%r10	%r10d	%r10w		%r10b
%r11	%r11d	%r11w		%r11b
%r12	%r12d	%r12w		%r12b
%r13	%r13d	%r13w		%r13b
%r14	%r14d	%r14w		%r14b
%r15	%r15d	%r15w		%r15b

- loading a constant into a register

```
        movq    $0x2a, %rax   # rax <- 42
        movq    $-12,  %rdi
```

- loading the address of a label into a register

```
        movq    $label, %rdi
```

- copying a register into another register

```
        movq    %rax, %rbx   # rbx <- rax
```

- addition of two registers

```
addq    %rax, %rbx      # rbx <- rbx + rax
```

similarly, `subq` (subtraction), `imulq` (signed multiplication)

- addition of a register and a constant

```
addq    $2, %rcx        # rcx <- rcx + 2
```

- particular case

```
incq    %rbx            # rbx <- rbx+1
```

(similarly, `decq`)

- negation

```
negq    %rbx            # rbx <- -rbx
```

- logical not

```
        notq    %rax                # rax <- not(rax)
```

- and, or, exclusive or

```
        orq     %rbx, %rcx          # rcx <- or(rcx, rbx)
        andq    $0xff, %rcx         # erases bits >= 8
        xorq    %rax, %rax          # zeroes %rax
```

- shift left (inserting zeros)

```
salq    $3, %rax     # 3 times
salq    %cl, %rbx    # cl times
```

- arithmetic shift right (duplicating the sign bit)

```
sarq    $2, %rcx
```

- logical shift right (inserting zeros)

```
shrq    $4, %rdx
```

- rotation

```
rolq    $2, %rdi
rorq    $3, %rsi
```

The suffix `q` means a 64-bit operand (*quad words*).

Other suffixes are allowed

suffix	#bytes	
`b`	1	(*byte*)
`w`	2	(*word*)
`l`	4	(*long*)
`q`	8	(*quad*)

```
        movb    $42, %ah
```

(when the suffix is omitted, the assembler tries to infer)

When operand sizes differ, one must indicate the extension mode

```
movzbq  %al, %rdi        # with zeros extension
movswl  %ax, %edi        # with sign  extension
```

An operand between parentheses means an indirect addressing,
i.e., the data in memory at this address.

```
        movq    $42, (%rax)      # mem[rax] <- 42
        incq    (%rbx)           # mem[rbx] <- mem[rbx] + 1
```

Note: the address may be a label

```
        movq    %rbx, x
```

Limitation

Operations do not allow several memory accesses

```
        addq    (%rax), (%rbx)
        Error: too many memory references for ‘add’
```

One has to use a temporary register.

```
        movq    (%rax), %rcx
        addq    %rcx, (%rbx)
```

The general form of the operand is

$$A(B, I, S)$$

and it stands for address $A + B + I \times S$ where

- A is a 32-bit signed constant
- I is 0 when omitted
- $S \in \{1, 2, 4, 8\}$ (is 1 when omitted)

Example:

```
movq    -8(%rax,%rdi,4), %rbx  # rbx <- mem[-8+rax+4*rdi]
```

Operation `lea` computes the effective address of the operand

$$A(B, I, S)$$

```
leaq    -8(%rax,%rdi,4), %rbx   # rbx <- -8+rax+4*rdi
```

Note: we can make use of it to perform arithmetic

```
leaq  (%rax,%rax,2), %rbx  # rbx <- 3*%rax
```

Most operations set the processor flags, according to their outcome.

flag	meaning
ZF	the result is 0
CF	a carry was propagated beyond the most significant bit
SF	the result is negative
OF	arithmetic overflow (signed arith.)
etc.	

(notable exception: `lea`)

Three instructions can test the flags

- conditional jump
  (j*cc*)

```
jne    label
```

- computes 1 (true) or 0 (false)
  (set*cc*)

```
setge %bl
```

- conditional mov
  (cmov*cc*)

```
cmovl %rax, %rbx
```

suffix		meaning	flags
e	z	$= 0$	ZF
ne	nz	$\neq 0$	~ZF
s		< 0	SF
ns		≥ 0	~SF
g		$>$ signed	~(SF^OF)&~ZF
ge		$\geq$ signed	~(SF^OF)
l		$<$ signed	SF^OF
le		$\leq$ signed	(SF^OF)|ZF
a		$>$ unsigned	~CF&~ZF
ae		$\geq$ unsigned	~CF
b		$<$ unsigned	CF
be		$\leq$ unsigned	CF|ZF

One can set the flags without storing the result anywhere, as if doing a subtraction or a logical and.

```
        cmpq    %rbx, %rax        # flags of rax - rbx
```

(beware of the direction!)

```
        testq   %rbx, %rax        # flags of rax & rbx
```

Unconditional Jump

- to a label

```
        jmp     label
```

- to a computed address

```
        jmp     *%rax
```

Many, many other instructions
[*Enumerating x86-64 — It's Not as Easy as Counting*]

Including SSE instructions operating on large registers containing several integers or floating-point numbers.

The Challenge of Compilation

Why to Keep Learning Assembly?

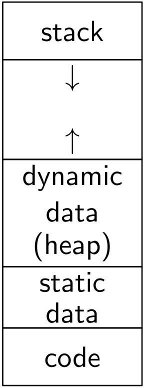
The challenge of compilation: to translate a high-level program into this instruction set.

In particular, we have to

- translate control structures (tests, loops, exceptions, etc.)
- translate function calls
- translate complex data structures (arrays, structures, objects, closures, etc.)
- allocate dynamic memory

Observation: function calls can be arbitrarily nested
$\Rightarrow$ registers cannot hold all the local variables
$\Rightarrow$ we need to allocate memory.

Yet function calls obey a *last-in first-out* mode, so we can use a stack.



The stack is allocated at the top of the memory, and increases downwards; `%rsp` points to the top of the stack.

Dynamic data (which needs to survive function calls) is allocated on the heap (possibly by a GC), above static data, and increases upwards.

This way, no collision between the stack and the heap (unless we run out of memory).

Note: each program has the illusion of using the whole memory; the OS creates this illusion, using the MMU.

- pushing

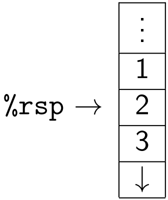
```
pushq    $42
pushq    %rax
```

- popping

```
popq     %rdi
popq     (%rbx)
```

example:

```
pushq    $1
pushq    $2
pushq    $3
popq     %rax
```



When a function f (the caller)
needs to call a function g (the callee),
it cannot simply do

```
    jmp    g
```

since we need to come back to the code of f when g terminates.

The solution is to make use of the stack.

Two instructions for this purpose:

instruction

```
        call    g
```

1. pushes the address of the next instruction on the stack
2. transfers control to address g

and instruction

```
        ret
```

1. pops an address from the stack
2. transfers control to that address

Problem: any register used by g is lost for f.

There are many solutions, but we typically resort to calling conventions.

- up to six arguments are passed via registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- other arguments are passed on the stack, if any
- the returned value is put in `%rax`

- registers `%rbx`, `%rbp`, `%r12`, `%r13`, `%14` and `%r15` are callee-saved, i.e., the callee must save them if needed; typically used for long-term data, which must survive function calls
- the other registers are caller-saved, i.e., the caller must save them if needed; typically used for short-term data, with no need to survive calls

- `%rsp` is the stack pointer, `%rbp` the frame pointer

Alignment

On function entry, `%rsp` $+ 8$ must be a multiple of 16.

Library functions (such as `scanf` for instance) may fail if this is not ensured.

Stack alignment may be performed explicitly

```
f:    subq $8, %rsp  # align the stack
      ...
      ...    # since we make calls to extern functions
      ...
      addq $8, %rsp
      ret
```

or indirectly

```
f:    pushq %rbx  # we save %rbx
      ...
      ...         # because we use it here
      ...
      popq  %rbx  # and we restore it
      ret
```

... are nothing more than conventions.

In particular, we are free not to use them as long we stay within the perimeter of our own code.

When linking to external code (*e.g.* `puts` earlier), however, we must obey the calling conventions.

Function Calls, in Four Steps

There are four steps in a function call

1. for the caller, before the call
2. for the callee, at the beginning of the call
3. for the callee, at the end of the call
4. for the caller, after the call

They interact using the top of the stack, called the stack frame and located between `%rsp` and `%rbp`.

1. passes arguments in `%rdi`,...,`%r9`, and others on the stack, if more than 6
2. saves caller-saved registers, in its own stack frame, if they are needed after the call
3. executes

```
        call  callee
```

The Callee, at the Beginning of the Call

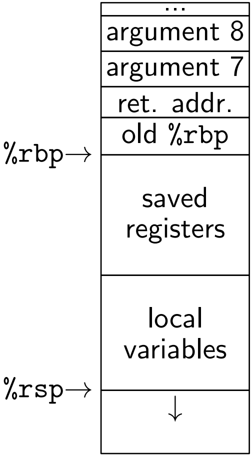
1. saves %rbp and set it, for instance with

```
pushq   %rbp
movq    %rsp, %rbp
```

2. allocates its stack frame, for instance with

```
subq    $48, %rsp
```

3. saves callee-saved registers that it intends to use



%rbp eases access to arguments and local variables, with a fixed offset (whatever the top of the stack)

The Callee, at the End of the Call

1. stores the result into `%rax`
2. restores the callee-saved registers, if needed
3. destroys its stack frame and restores `%rbp` with

   ```
           leave
   ```

   that is equivalent to

   ```
           movq    %rbp, %rsp
           popq    %rbp
   ```

4. executes

   ```
           ret
   ```

1. pops arguments 7, 8, ..., if any
2. restores the caller-saved registers, if needed

- a machine provides
  - a limited instruction set
  - efficient registers, costly access to the memory
- the memory is split into
  - code / static data / dynamic data (heap) / stack
- function calls make use of
  - a notion of stack frame
  - calling conventions

Exercise : let's program factorial

- with a loop
- with a recursive function

Bibliography

- *Computer Systems: A Programmer's Perspective*
  (R. E. Bryant, D. R. O'Hallaron)
  its PDF appendix *x86-64 Machine-Level Programming*

- *Notes on x86-64 programming* by Andrew Tolmach
  (available on this week's lab page)