

Programming Language (5)

Basics of Programming Language Implementation

田浦

Contents

- 1 Introduction
- 2 CPU and machine code : An overview
- 3 A glance at x86 machine (assembly) code

Contents

- 1 Introduction
- 2 CPU and machine code : An overview
- 3 A glance at x86 machine (assembly) code

Two basic forms of language implementation

- **interpreter**: interprets and executes programs (takes a program and an input; and computes the output)
- **compiler**: translates programs into **a machine (assembly) code**, that can directly execute by the processor
 - ▶ **ahead-of-time (AOT)**: the entire program is compiled before execution
 - ▶ **just-in-time (JIT)**: programs are incrementally compiled as they get executed (e.g., a function at a time)

*regardless of details, the central issue is how to translate **a source program** → **machine code***

Why do you want to build a language, today?

- new hardware
 - ▶ GPUs (CUDA, OpenACC, OpenMP), AI chips, Quantum, ...
 - ▶ new instruction set (e.g., SIMD, matrix, ...) of the processor
- new general purpose languages
 - ▶ Scala, Julia, Go, Rust, etc.
- special purpose (domain specific) languages
 - ▶ statistics (R, MatLab, etc.)
 - ▶ data processing (SQL, NoSQL, SPARQL, etc.)
 - ▶ deep learning
 - ▶ constraint solving, proof assistance (Coq, Isabelle, etc.)
 - ▶ macro (Visual Basic (MS Office), Emacs Lisp (Emacs), Javascript (web browser), etc.)

Contents

- 1 Introduction
- 2 CPU and machine code : An overview
- 3 A glance at x86 machine (assembly) code

What a machine (assembly) code looks like

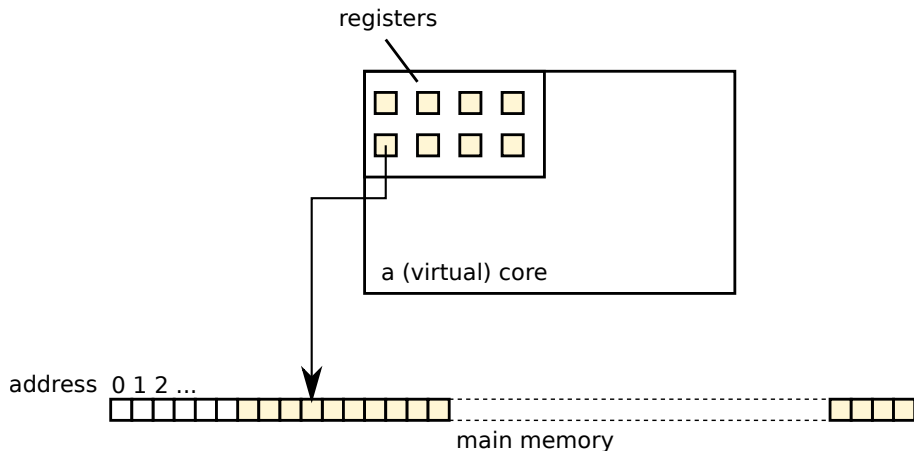
- it *is* just another programming language
- it has many features present in programming languages

source	machine code
expressions	arithmetic instructions
if statement	compare / conditional jump instructions
variables	<i>registers</i> and <i>memory</i>
structs and arrays	memory and load/store instructions

compilation is nothing like a magic; it's more like translating English to French

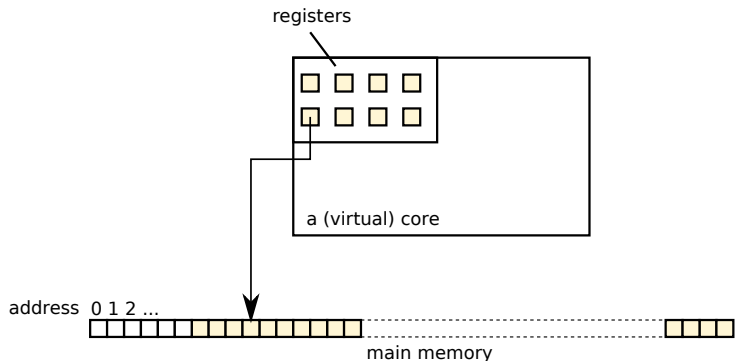
What a CPU (core) looks like

- a small number (typically < 100) of *registers*
 - ▶ each register can hold a small amount of (e.g., 64 bit) data
- majority of data are stored in *memory* (a few to ~ 1000 GB)



Memory

- where majority of data your program processes are stored
- memory is essentially a large flat array indexed by integers, often called *addresses*
- an address is just an integer

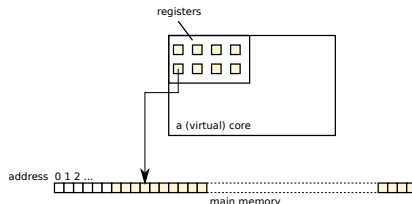


What a CPU (core) does

- a special register, called *program counter* or *instruction pointer* specifies the address to fetch the next instruction at
- a CPU core is essentially a machine that does the following

```
1 repeat:  
2   inst = memory[program counter]  
3   execute inst
```

- an instruction
 - ▶ performs some computation of values on a few registers or a memory location, and
 - ▶ changes the program counter (typically to the next instruction on memory)



Exercise objectives

- `pl06_how_it_gets_compiled`
- learn how a **compiler** does the job,
- by inspecting assembly code generated from functions of the source language

Contents

- 1 Introduction
- 2 CPU and machine code : An overview
- 3 A glance at x86 machine (assembly) code

The first glance

```
1      .globl  go_Op106.Many__args
2      .type   go_Op106.Many__args, @function
3 go_Op106.Many__args:
4      .LFB0:
5          .cfi_startproc
6          cmpq    %fs:112, %rsp
7          jb      .L3
8      .L2:
9          movq    48(%rsp), %rax
10         addq    40(%rsp), %rax
11         addq    32(%rsp), %rax
12         addq    24(%rsp), %rax
13         addq    16(%rsp), %rax
14         addq    8(%rsp), %rax
15         addq    %r9, %rax
16         addq    %r8, %rax
17         addq    %rcx, %rax
18         addq    %rdx, %rax
19         addq    %rsi, %rax
20         addq    %rdi, %rax
21         ret
22      .L3:
23         movl    $0, %r10d
24         movl    $48, %r11d
25         call    __morestack
26         ret
27         jmp     .L2
28         .cfi_endproc
29      .LFE0:
30         .size   go_Op106.Many__args, .-go_Op106.Many__args
31         .globl  go.pl06..types
```

Unimportant lines

```
1      .globl go_Op106.Many__args
2      .type go_Op106.Many__args, ...
3      go_Op106.Many__args:
4      .LFB0:
5      .cfi_startproc
6      cmpq    %fs:112, %rsp
7      jb     .L3
8      .L2:
9      movq    48(%rsp), %rax
10     addq    40(%rsp), %rax
11     addq    32(%rsp), %rax
12     addq    24(%rsp), %rax
13     addq    16(%rsp), %rax
14     addq    8(%rsp), %rax
15     addq    %r9, %rax
16     addq    %r8, %rax
17     addq    %rcx, %rax
18     addq    %rdx, %rax
19     addq    %rsi, %rax
20     addq    %rdi, %rax
21     ret
22     .L3:
23     movl    $0, %r10d
24     movl    $48, %r11d
25     call    __morestack
26     ret
27     jmp     .L2
28     .cfi_endproc
29     .LFE0:
30     .size go_Op106.Many__args, ...
31     .globl go.pl06..types
```

- indented lines beginning with a dot (e.g., `.file`, `.section`, `.ascii`, `.text`, `.globl`, ...) are not instructions and *largely not interesting or import*
- lines with a symbol followed by a colon (e.g., `.L2:`, `.LFE0:`, `go_Op106.Many__args:`, ...) are *labels* and used for the target of jump instructions or call instructions

Where to look

```
1      .globl go_Op106.Many__args
2      .type go_Op106.Many__args, ...
3  go_Op106.Many__args:
4  .LFB0:
5      .cfi_startproc
6      cmpq    %fs:112, %rsp
7      jb     .L3
8  .L2:
9      movq    48(%rsp), %rax
10     addq    40(%rsp), %rax
11     addq    32(%rsp), %rax
12     addq    24(%rsp), %rax
13     addq    16(%rsp), %rax
14     addq    8(%rsp), %rax
15     addq    %r9, %rax
16     addq    %r8, %rax
17     addq    %rcx, %rax
18     addq    %rdx, %rax
19     addq    %rsi, %rax
20     addq    %rdi, %rax
21     ret
22 .L3:
23     movl    $0, %r10d
24     movl    $48, %r11d
25     call    __morestack
26     ret
27     jmp     .L2
28     .cfi_endproc
29 .LFE0:
30     .size go_Op106.Many__args, ...
31     .globl go.p106..types
```

- focus on lines having *instructions*
- instructions for a function start with a label *similar to* the function name, but it may not be exactly the same (name mangling)

Registers

- general-purpose 64 bit integer registers: `r{a,b,c,d}x`, `rdi`, `rsi`, `r[8-15]`, `rbp`
- general-purpose floating point number registers: `xmm[0-15]`
- stack pointer register: `rsp`
- a compare flag register: `eflags`, not directly used by instructions
 - ▶ implicitly set by compare instructions
 - ▶ implicitly used by conditional jump instructions
- an instruction pointer register: `rip`, not directly used by instructions
 - ▶ set by every instruction
- https://wiki.cdote.senecapolytechnic.ca/wiki/X86_64_Register_and_Instruction_Quick_Start

Frequently used instructions

learn details and other instructions from the exercise

- `addq` (+), `leaq` (+), `subq` (-), `imulq` (\times), `idivq` ($/$)
- `movq` : move values between registers or between register and memory (load/store)
- `cmpq` : compare two values and set the result into the `eflags` register
- `jl` ($<$), `jle` (\leq), `jg` ($>$), `jge` (\geq), `je` ($=$), `jne` (\neq) : jump if a condition (indicated by `eflags`) is met
- `call`, `ret` : call or return from a function

How to read instructions and operands (GNU assembler)

- e.g., `addq` instructions takes two operands

```
1 addq x,y
```

and its effect is

```
1 y += x
```

- many two operand instructions behave similarly

```
1 opq x,y  $\equiv$  y = y op x
```

- especially confusing is `subq`

```
1 subq x,y  $\equiv$  y = y - x
```

- *syntax and operand order actually depend on the assembler*

Syntax of operands

- $\$n$: immediate value of n
- $\%R$: register named R
- (\dots) : address operand (details in the next slide)

where

- n : a constant (4, 8, etc.)
- R : register name (`rax`, `rbx`, `rdi`, etc.)

ex.

- `addq $1,%rax` : add 1 to `%rax` register
- `subq $1,%rax` : subtract 1 from `%rax` register

Address operands

- an address operand (...) specifies an address, and can be
 - ▶ $(\%R) : R$
 - ▶ $n(\%R) : R + n$
 - ▶ $n(\%R, s, R') : R + sR' + n$
- where
 - ▶ n, s : integer constants
 - ▶ R, R' : register names
- ex.
 - ▶ `mulq (%rdi), %rax` : reads address specified by `%rdi` and multiply `%rax` by it
 - ▶ `movq %rax, (%rdi)` : writes the value of `%rax` to the address specified by `%rdi`
 - ▶ `leaq 4(%rdi), %rax` : `%rax = %rdi + 4`; this instruction looks like reading/writing memory, but it is actually just a peculiar arithmetic (common in address calculation)

Things to watch in the exercise

- calling convention or ABI : function's incoming parameters and the return value are put in places (typically registers) predetermined by convention
- observe where incoming parameters are by compiling simple functions like

```
1 f(int x, int y) = x + y
```

- change parameter types and function body to know
 - ▶ how data are represented (integers, floating point numbers, arrays, structs, etc.)
 - ▶ how various expressions are implemented (arithmetic, array access, structure access, etc.)