Programming Language (5)
Basics of Programming Language Implementation

田浦

Contents

Contents

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

What a machine (assembly) code looks like

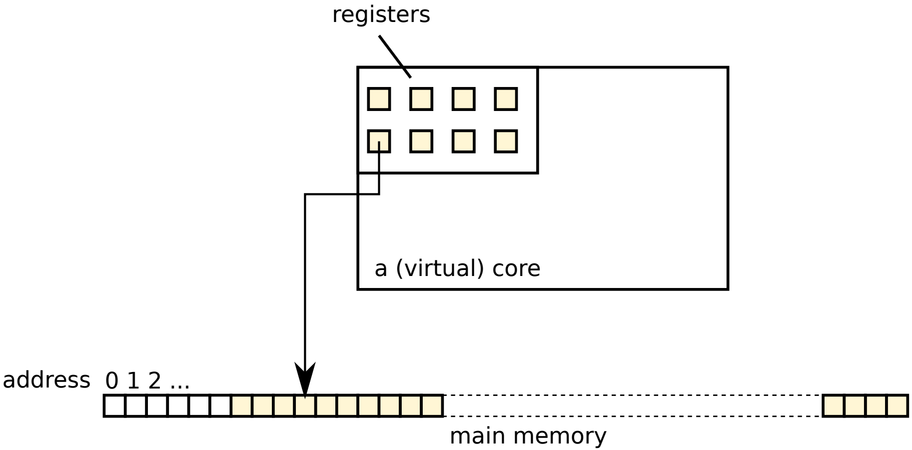
- it *is* just another programming language
- it has many features present in programming languages

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

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

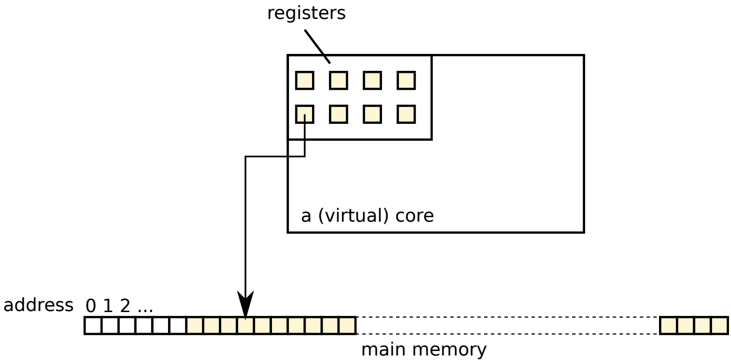
What a CPU (core) looks like

- 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

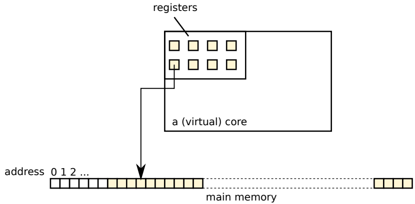


What a CPU (core) does

- 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

```
repeat:
  inst = memory[program counter]
  execute inst
```

- an instrcution
  - 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

The first glance

```
    .file   "add123.go"
    .section    .go_export,"",@progbits

      ...

    .text
    .globl  go_0pl06.Add123
    .type   go_0pl06.Add123, @function
go_0pl06.Add123:
.LFB0:
    .cfi_startproc
    cmpq    %fs:112, %rsp
    jb      .L3
.L2:
    leaq    123(%rdi), %rax
    ret
.L3:
    movl    $0, %r10d
    movl    $0, %r11d
    call    __morestack
    ret
    jmp     .L2
    .cfi_endproc
.LFE0:
    .size   go_0pl06.Add123, .-go_0pl06.Add123
    .globl  go.pl06..types

      ...
```

looks scary?

Unimportant lines

```
	.file "add123.go"
	.section .go_export,"",@progbits

	  ...

	.text
	.globl go_0pl06.Add123
	.type go_0pl06.Add123, @function
go_0pl06.Add123:
.LFB0:
	.cfi_startproc
	cmpq	%fs:112, %rsp
	jb	.L3
.L2:
	leaq	123(%rdi), %rax
	ret
.L3:
	movl	$0, %r10d
	movl	$0, %r11d
	call	__morestack
	ret
	jmp	.L2
	.cfi_endproc
.LFE0:
	.size go_0pl06.Add123, ...
	.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_0pl06.Many__args:`, ...) are *labels* and used for the target of jump instructions or call instructions

Where to look

```
go_0pl06.Add123:
.LFB0:

    cmpq    %fs:112, %rsp
    jb      .L3
.L2:
    leaq    123(%rdi), %rax
    ret
.L3:
    movl    $0, %r10d
    movl    $0, %r11d
    call    __morestack
    ret
    jmp     .L2

.LFE0:




       ...
```

- focus on lines having *instructins*
- 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.cdot.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 (of GNU assembler)

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

```
addq x,y
```

  and its effect is

```
y += x
```

- many two operand instructions behave similarly

```
opq x,y ≡ y = y op x
```

- especially confusing is `subq`

```
subq x,y ≡ y = y - x
```

Syntax of operands

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

where

- n : a constant (`4`, `8`, etc.)
- R : regiser 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,8(%rdi)` : writes the value of `%rax` to the address specified by `%rdi+8`
  - `leaq 16(%rdi,8,%rsi),%rax` : `%rax = %rdi + 8 * %rsi + 16`; this instruction looks like reading/writing memory, but it is actually just a peculiar arithmetic (common in address calculation but also used for integer addition)

Julia assembly syntax

- *syntax and operand order actually differ between assemblers*
- they are of course identical in the binary level
- in particular, output from Julia (`code_native`) is different
  - *destination-first syntax*

$$\texttt{addq}\ x, y \quad \equiv \quad x \mathrel{\texttt{+=}} y$$

  - address operands are more intuitive. ex.

GNU	Julia
`mulq (%rdi),%rax`	`mulq %rax,[%rdi]`
`movq %rax,8(%rdi)`	`movq [%rdi+8],%rax`
`leaq 16(%rdi,8,%rsi),%rax`	`leaq %rax,[%rdi+8*%rsi+16]`

Things to learn in the exercise

1. **calling convention or ABI :** function's incoming parameters and the return value are put in places (typically registers) predetermined by convention
2. **data representation :** once you know where incoming parameters and return values are, understand how data (integers, floating point numbers, structs, pointers to something, arrays, etc.) are represented, by compiling simple functions that work on them. e.g.,

```
f(a, i) = a[i]
```

3. **control flow :** how various control flows (conditionals and loops) are implemented
4. **function calls :** how function calls are implemented