

How Programming Languages Work (Basics)

Kenjiro Taura

2024/05/19

Contents

Introduction	2
CPU and machine code : an overview	9
A glance at ARM64 machine (assembly) code	18

Introduction



Why you want to make a language, today?

- new hardware
 - GPUs, AI chips, Quantum, ...
 - new instructions (e.g., SIMD, matrix, ...)
- new general purpose languages
 - Scala, Julia, Go, Rust, etc.

Why you want to make a language, today?

- 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.)

Taxonomy : interaction mode

- **interactive / read-eval-print-loop (REPL)**
 - type code directly or load source code in a file interactively
 - Julia
- **batch compile**
 - convert source into an executable file
 - and run it (typically the “main” function)
 - Go, Rust
- some language implementations provide both
 - OCaml

Taxonomy : execution strategy

- **interpreter** executes source code directly with its input
 - interpreter (*source-code, input*) \rightarrow *output*
- **compiler** first converts source code into **a machine (assembly) code** that is directly executed by the CPU
 - compiler (*source-code*) \rightarrow *machine-code*;
 - *machine-code (input)* \rightarrow *output*
- **translator** or **transpiler** are like compiler, but convert into another language, not machine (assembly) code

A (minor) note: machine code vs. assembly code

- in many contexts, they are used almost interchangeably
- machine (assembly) *languages* are almost interchangeable, too
- if asked a difference,
 - **machine** code is *the* real encoding of instructions interpretable by a CPU
 - **assembly** code refers to a *textual (human-readable) representation* of machine code

Taxonomy : compiler/translator

- **ahead-of-time (AOT)** compiler converts all the program parts into assembly before execution
- **just-in-time (JIT)** compiler converts program parts incrementally as they get executed (e.g., a function at a time)

CPU and machine code : an overview

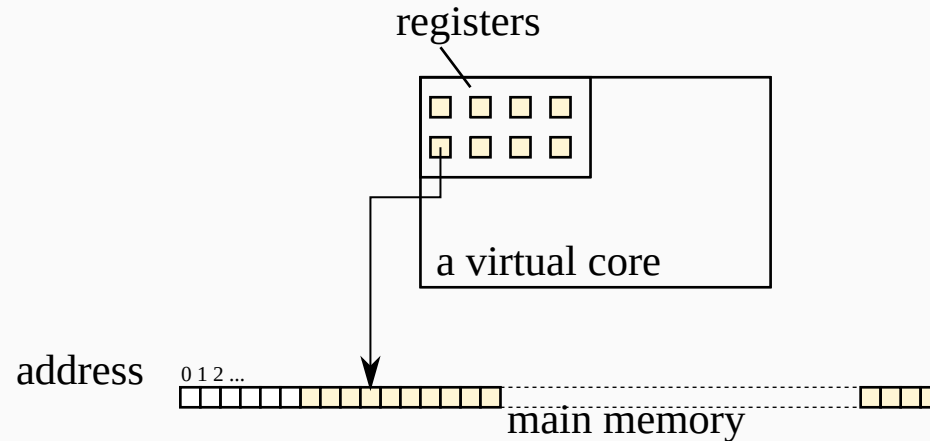
High-level (programming) languages vs. assembly languages

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

high-level language	assembly language
variables	<i>registers and memory</i>
structs and arrays	memory and load/store instructions
expressions	arithmetic instructions
if / loop	compare, conditional branch instructions
functions	branch and link instructions

What a CPU looks like

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

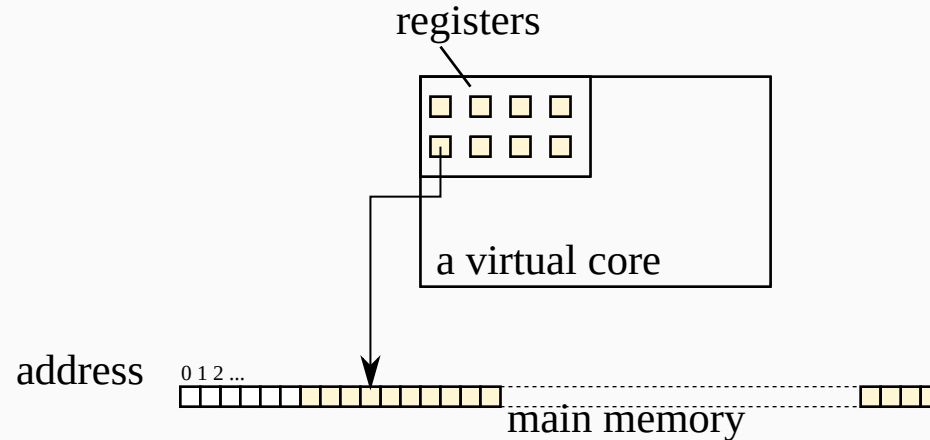


Terminology note : CPU \ni core \ni virtual core

- a *CPU* has multiple (typically, 2 to >100) *cores*
- a core has multiple (typically, 1 to a few) *virtual cores* or *hardware threads*
- each virtual core has its own registers and is capable of fetching and executing instructions
- all virtual cores of a CPU share the main memory
- they are often used interchangeably when the distinction is not important
- we are only concerned about a single virtual core

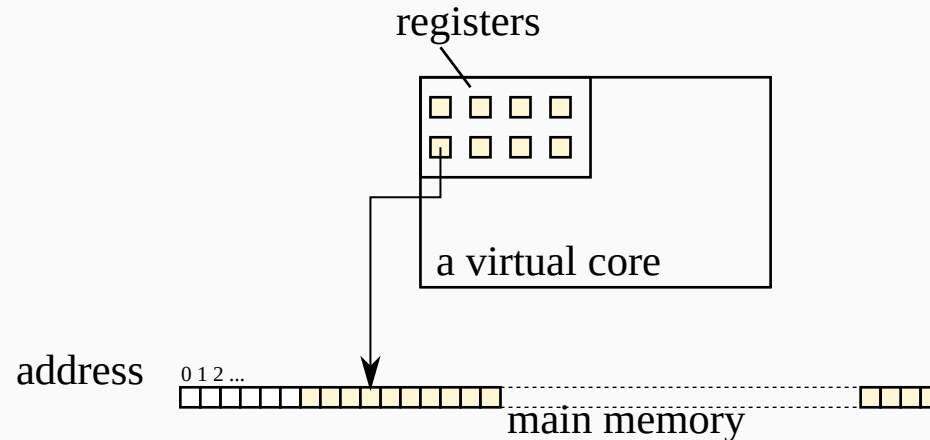
Main memory

- \approx a large array indexed by integers, called *addresses*
- in machine code level, an address is just an integer



Main memory

- each address typically stores 8 bits (a *byte*) of data
- a larger word is stored in consecutive addresses. e.g.,
 - 32 bit (4 byte) word occupies 4 consecutive addresses
 - 64 bit (8 byte) word occupies 8 consecutive addresses



What a virtual core does

- a core is a machine that does the following:

repeat:

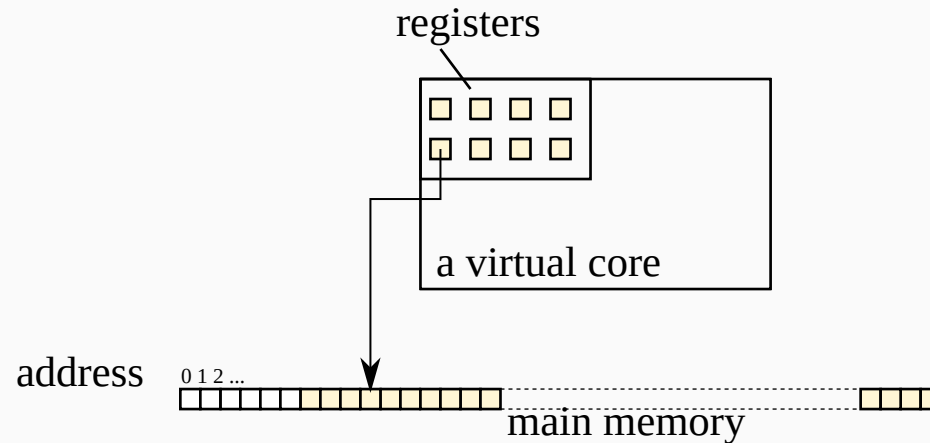
```
instruction = memory[program counter]
```

```
execute instruction
```

- *program counter* (or *instruction pointer*) is the register that specifies the address to fetch the next instruction from

What an instruction does

1. perform some computation on specified register(s) or a memory address
2. change the program counter
 - typically to the next address of the instruction just executed



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

A glance at ARM64 machine (assembly) code

A glance at ARM64 machine (assembly) code

Rust

```
#[no_mangle]
pub fn add123(x:i64, y:i64) -> i64 {
    y + 123
}
```

assembly

```
.text
.file    "pl06.lebfa1..."
.section .text.add123,...
.globl   add123
.p2align        2
.type     add123,@function

add123:
.cfi_startproc
add      x0, x1, #123
ret

.Lfunc_end0:
.size    add123, .Lfunc_...
.cfi_endproc
```

Unimportant lines

- indented lines starting with a dot (e.g., `.file`, `.section`, `.text`, etc.) are *directives* (not instructions) and largely not important
- unintended lines ending in a colon (e.g., `add123:`) are *labels* used to human-readably specify jump targets

```
.text
.file    "pl06.lebfa1..."
.section .text.add123,...
.globl   add123
.p2align          2
.type     add123,@function

add123:
    .cfi_startproc
    add     x0, x1, #123
    ret
.Lfunc_end0:
    .size
add123, .Lfunc_end0-add123
    .cfi_endproc
```

Unimportant lines

- indented lines starting with a dot (e.g., `.file`, `.section`, `.text`, etc.) are *directives* (not instructions) and largely not important
- unintended lines ending in a colon (e.g., `add123:`) are *labels* used to human-readably specify jump targets

```
add123:
```

```
    add    x0, x1, #123
```

```
    ret
```

```
.Lfunc_end0:
```

How to look at assembly

- focus on lines that are *instructions*
- look for a label *similar to* the function name, which is where its instructions start
 - the label may not be exactly the same as the function name (*name mangling*)

add123:

add x0, x1, #123

ret

.Lfunc_end0:

How to look at instructions

- ex.

```
add    x0, x1, #123
```

performs $x0 = x1 + 123$

- `add` is an *instruction name* or *mnemonic*
- takes three *operands* (`x0`, `x1`, and `#123`)
 - `x0`, `x1` : register
 - `#123` : constant (*immediate value* or *literal*)

ARM64 registers

- integer registers $\times 32$
 - 64 bit : `x0`, `x1`, ..., `x31`
 - 32 bit : `w0`, `w1`, ..., `w31`
 - uses low 32 bits of `x0`, `x1`, ..., `x31`
- floating point registers $\times 32$
 - 64 bit (double precision) : `d0`, `d1`, ..., `d31`
 - 32 bit (single precision): `s0`, `s1`, ..., `s31`
 - uses low 32 bits of `d0`, `d1`, ..., `d31`

ARM64 registers

- implicit registers (state)
 - conditional code register
 - program counter

ARM64 instructions

- arithmetic
- move
- load / store
- compare
- conditional / unconditional branch
- branch and link
- return

Arithmetic

- ex.

```
sub x0, x1, x2
```

performs

$$x0 = x1 - x2$$

- typically takes three operands

Return

Things to learn in the exercise

1. Calling convention / ABI : How parameters and return values are passed (typically via registers)
2. Data representation : Learn how data types (ints, floats, structs, pointers, arrays) are represented

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

3. Control flow : How conditionals and loops are implemented
4. Function calls : How function call/return is implemented