How Programming Languages Work (Basics)

Kenjiro Taura

2024/05/19

Contents

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

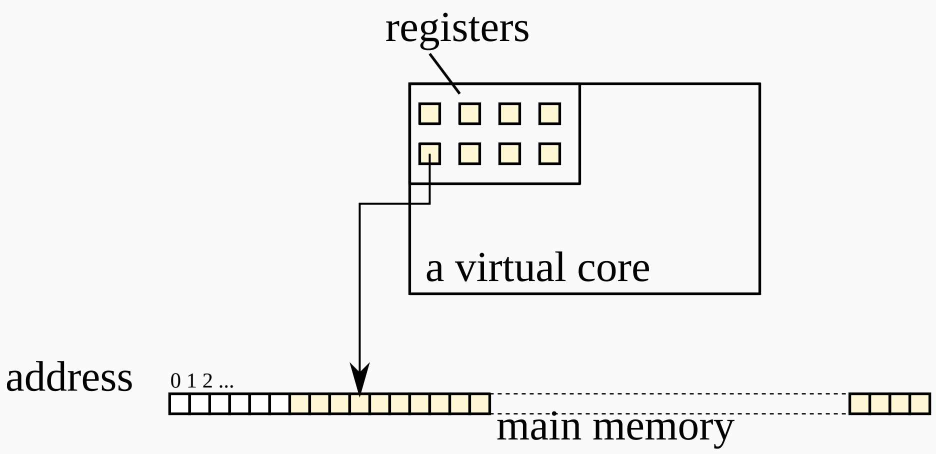
High-level (programming) languages vs. assembly languages

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

high-level language	**assembly language**
variables	*registers* and *memory*
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

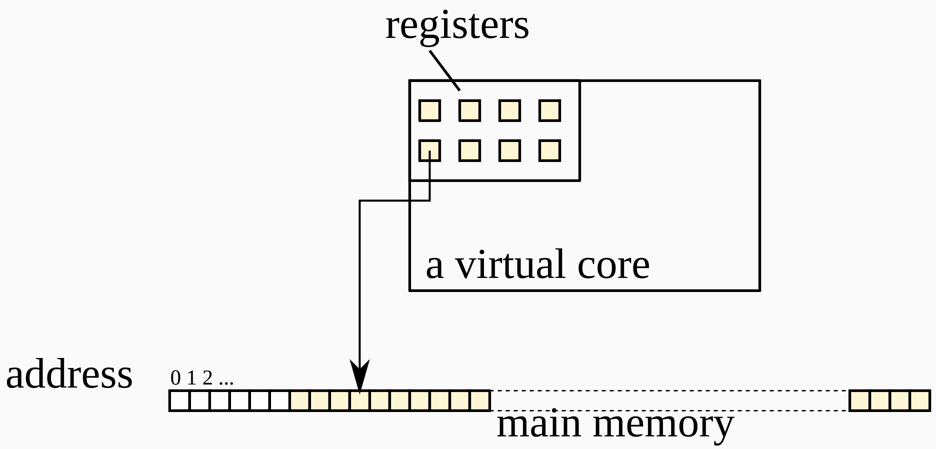


Terminology note : CPU $\ni$ core $\ni$ virtual core

- 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
- this course is only concerned about a single virtual core

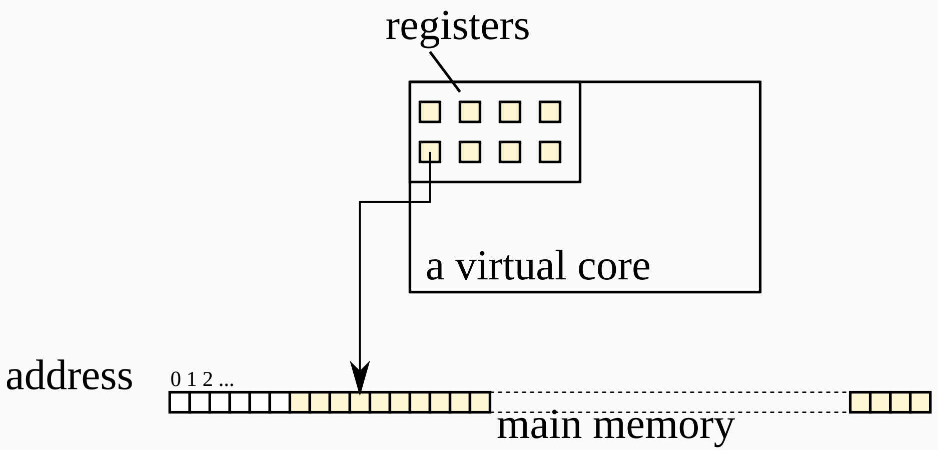
Main memory

- ≈ 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

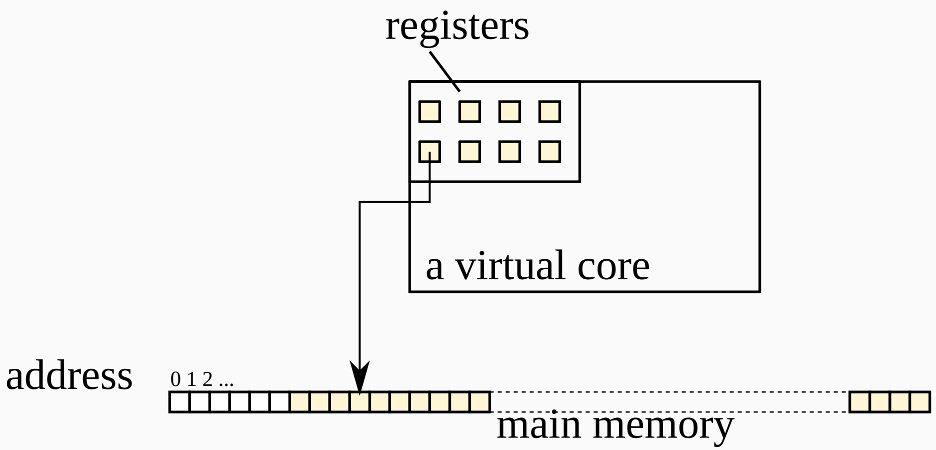
- a core is a machine that does the following:

```
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 istruction 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   "pl06.1ebfa1..."
        .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    "pl06.1ebfa1..."
        .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)
  - condition code register — holds the result of compare instruction
  - program counter — holds the address of the next instruction

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
- the result is written to the first operand

Move

- ex.

```
mov x0, x1
```

performs

$$\texttt{x0} = \texttt{x1}$$

Load/store

- ex.

```
ldr x0, [x1]
```

fetches the 64-bit value from the address in `x1` register and puts it in `x0`

- in a pseudo C notation

```
x0 = *(long*)x1
```

- ex.

Load/store

```
str x0, [x1]
```

writes the value in `x0` to the address in `x1` register

- in a pseudo C notation

```
*(long*)x1 = x0
```

Load/store variations

- constant offset

```
ldr x0, [x1,#8]
```

≈ `x0 = *(long*)(x1+8)`

- scaled variable offset

```
ldr x0, [x1, x2, lsl #3]
```

≈ `x0 = *(long*)(x1 + (x2 << 3))`

- pre-increment

Load/store variations

```
ldr x0, [x1,#8]!
```

≈ `x1 += 8; x0 = *(long*)x1`

- post-increment

```
ldr x0, [x1],#8
```

≈ `x0 = *(long*)x1; x1 += 8`

- signed offset

```
ldur x0, [x1,#-8]
```

Load/store variations

$\approx$ `x0 = *(long*)(x1 - 8)`

Compare

- ex.

```
cmp x0, x1
```

$\approx$

condition code register $=$ `c x0 - x1`

- note:
  - *condition code register* does not hold the value of `c x0 - x1` itself
  - it stores whether it is > 0, $= 0$, < 0, etc. as an array of bits

Conditional / unconditional branch

- ex.

$$\texttt{b.eq}\ \textit{label}$$

$\approx$ branch to *label* if the last comparison was equal (the '$=0$' bit is set in condition code register)

- ex.

$$\texttt{b}\ \textit{label}$$

unconditionally branches to *label*

Branch and link

- ex.

`bl` *label*

≈ jump to *label*; `x30 =` the next address of the `bl` instruction

- used to jump to a function, remembering where to return after the function

Return

- ex.

```
ret
```

≈ jump to the address `x30` (presumably set by `bl` instruction)

How a function (call) works — Applicatoin Binary Interface (ABI)

- ABI specifies assumptions upon function entry and requirements upon function return
- upon function entry
  - arguments are on specific registers defined by convention
  - `sp` (= `x31`) register points to the end of *stack*
  - the function must not use region at and above `sp` (can use area below `sp`)
-

  upon function return

How a function (call) works — Applicatoin Binary Interface (ABI)

- `sp`, `x30`, and a few other registers determined by convention (*callee save* registers) must have the same value as function entry
- return value must be on a specific register defined by convention

Illustrating function call

- `bl` *foo* instruction jumps to label (address) *foo* and sets `x30` register to the address immediately following the `bl` instruction

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