Introduction to Scientific Computing

Dr Hanno Rein

Last updated: October 12, 2018

1 Computers

A computer is a machine which can perform a set of calculations. The purpose of this course is to give you the ability to tell the computer which calculation to perform to solve a given problem of scientific nature.

This section of the course can only give you an extremely limited overview. I encourage you to read up a bit more on computer architecture, especially if you plan to dive deeper into the subject.

1.1 How computers work

A computer consists of two main components, the memory and the processor. The memory stores commands and all sorts of data. Memory comes in the form of hard drives, RAM, a L1 caches or USB sticks. Some of these are fast, others are portable or very cheap. For us, the distinction between them is not very important. The processor, also called CPU, reads commands from the memory and executes them. Here are some examples of possible instructions:

- beep
- add register a and register b and store the value in register c
- move memory content at address 123 to register a
- skip the next instruction
- go back 10 instructions

We could just write these instructions by hand and enter them one by one into the memory. This would involve a lot of repetitive work, reading lots of manuals and would take a long time. And indeed, just 40 years ago, this was the only way to get any *programming* done. Today, the computers are fast enough so that we can add a layer (or two) of abstraction between this very basic level of commands (called machine code) and what we actually have to write in our programs.

We are interested in assembler because it is extremely close to what actually happens on a computer. When we start using python next, you will see many high level instructions. These get translated into the most basic instructions for you, by something called a compiler, runtime or interpreter. Think of it as a translator between different languages.

The following figure illustrates how a program entered by a programmer is translated into machine code.

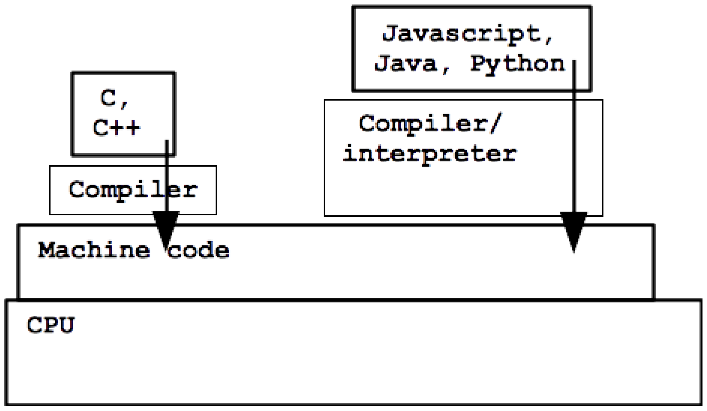

Figure 1: Software language stack. Source: `http://web.stanford.edu/class/cs101/`.

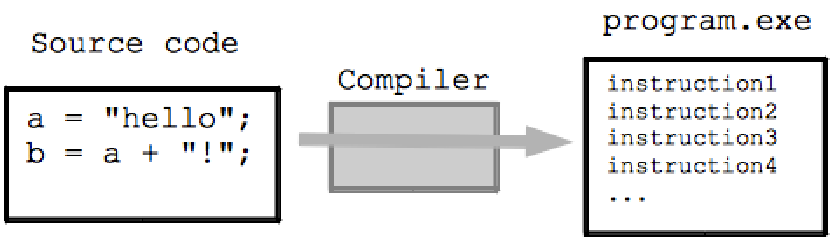

Figure 2: Compiler. Source: `http://web.stanford.edu/class/cs101/`.

1.2 Assembler

Let's construct a computer. This is a very simple example, but it illustrates the ideas that are used in more complex systems.

We'll construct a computer with 16 memory locations. We call those registers and give them a unique number, from 0 to 15. Each register can store exactly one number. The storage for the program itself is separate and not limited. It could be a punch card. There is also a so called instruction pointer that points to the current instruction in the program. The instruction counter starts at the beginning of the program and increases by one after every command. For now, our computer has only six commands. They all have a very similar form and are composed of at most 4 parts:

```
COMMAND PARAMETER PARAMETER PARAMETER
```

Some of the commands only have one or two parameter, in which case the others are simply ignored

Every command together with any possible combination of parameters gives something called a machine code, a single number that tells the computer what to do in one step.

Here are all the commands that we'll really need in our assembler language.

Command	Argument 1	Argument 2	Argument 3	Result
`SET`	s0	r1		r1 := s0 (The first argument is interpreted as a number, rather than a register address)
`COPY`	r0	r1		r1 := r0
`PRINT`	r0			Print value of r0
`INPUT`	r0			Input value from user and store it in r0
`ADD`	r0	r1	r2	r2 := r1 + r2
`SUB`	r0	r1	r2	r2 := r0 - r1
`IF`	r0	r1		Execute next command only if r0 > r1
`JUMP`	r0			Jump r0 commands forwards (backwards if negative)

To write and run an assembler program you have to write down the commands in the right order. If you had taken this course 40 years ago, you'd be taking our your punch cards and start punching. We will write them in a text file. You should by now know how to edit a text file on the linux system.

Every time you want to run your program, you need to save it first. Then, you run the simulator (or interpreter) and tell it to read in your text file. It will go through the text file, line by line, and execute exactly the command that you gave it.

Let's have a demo of the simplest possible example program. A program that outputs '1'.

```
SET 1 r0
PRINT r0
```

Code 1: Assembler program that outputs '1'.

Let's add two numbers.

```
SET 1 r0
SET 2 r1
ADD r0 r1 r2
PRINT r2
```

Code 2: Assembler program that adds two numbers and outputs the result.

Let's output all numbers from 0 to 9.

```
SET 0 r0
PRINT r0
SET 1 r0
PRINT r0
SET 2 r0
PRINT r0
SET 3 r0
PRINT r0
SET 4 r0
PRINT r0
SET 5 r0
PRINT r0
SET 6 r0
PRINT r0
SET 7 r0
PRINT r0
SET 8 r0
PRINT r0
SET 9 r0
PRINT r0
```

Code 3: Assembler program that outputs numbers from '0' to '9'.

This was rather dull. Let's find a smarter way.

```
SET 0 r0
SET 1 r2
SET -2 r3
PRINT r0
ADD r0 r2 r0
JUMP r3
```

Code 4: Assembler program that outputs all numbers.

The above program runs for ever. To make it stop we need a conditional statement. And now the correct way of only outputting the first 10 numbers

```
SET 0 r0
SET 10 r1
SET 1 r2
SET -3 r3
PRINT r0
ADD r0 r2 r0
IF r1 r0
JUMP r3
```

Code 5: Assembler program that outputs the first 10 numbers.

Our assembler language is very basic. For example, it has no commands for multiplication. How can we solve this?

I'll give you ten minutes. Come up with a assembler program that multiplies the two numbers in register 0 and 1 and store the result in register 2. You can assume that the two numbers are integers.

```
SET 3 r0
SET 4 r1
SET 0 r2
SET 0 r3
SET 1 r4
SET −4 r5
SET 3 r6
ADD r4 r3 r3
IF r3 r1
JUMP r6
ADD r0 r2 r2
JUMP r5
PRINT r2
```

Code 6: Assembler program multiplying two numbers.

Last but not least, let's write an algorithm for dividing two numbers, a and b. Let's assume that a and b are positive integers and that the result $c = a/b$ is also an integer.

```
SET 42  r0
SET 7   r1
SET −1  r2
SET 1   r3
SET 3   r7
SET −9  r8
SET −4  r9
 ADD r3 r2 r2
 SET 0 r4
 SET 0 r5
  ADD r3 r5 r5
  IF r5 r2
  JUMP r7
  ADD r1 r4 r4
  JUMP r9
 IF r0 r4
 JUMP r8
PRINT r2
```

Code 7: Assembler program dividing two integers.

Some other possible programs, I could ask you for:

- Square a given number
- Print out first N squares (1,2,4,8,16,...)
- Subtract a number from another number, only using additions.

We will now use this set of commands to calculate the Fibonacci numbers. This is a series of number that occurs frequently in nature. The first few numbers in the sequence are:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

They can be calculated by the recursion formula

$$F_n = F_{n-1} + F_{n-2}$$

i.e. the next number is the sum of the previous two numbers. The first two numbers are set two one. Let's now implement this in our Assembler language

```
SET 1 r0
PRINT r0
SET 1 r1
PRINT r1
ADD r0 r1 r2
PRINT r2
COPY r1 r0
COPY r2 r1
SET -5 r3
JUMP r3
```

Code 8: Simple assembler program to calculate Fibonacci numbers.

As a comparison, here is the same program in python

```
a=1
print a
b=1
print b
while 1:
        c = a+b
        print c
        a=b
        b=c
```

Code 9: Python program to calculate the Fibonacci numbers.