Installing GCC compiler and Assembler for ARM on Intel Architecture and Linux Operating System.

Install `gcc-arm-linux-gnueabi` and `binutils-arm-linux-gnueabi` packages, and then just use `arm-linux-gnueabi-gcc` instead of `gcc` for compilation.

```
sudo apt-get install gcc-arm-linux-gnueabihf gcc-arm-linux-gnueabi binutils-arm-
linux-gnueabi
```

To execute ARM executable file we need ARM executable engine

sudo apt-get install qemu

Lab work

Machine language is built up from discrete statements or instructions implemented by a particular processor. ARM is a family of instruction set architectures for computer processors and is the one used by the processor of the Raspberry Pi. The machine language is interpreted by the computer in term of binary codes. Binary code is what a computer can run. It is composed of instructions, that are encoded in a binary representation (such encodings are documented in the ARM manuals). You could write binary code encoding instructions but that would be painstaking. So instead we will write assembler language. Assembler language is just a thin syntax layer on top of the binary code.

```
.text                 /* -- Code section */
.global main          /* 'main' is our entry point and must be global */
.func main            /* 'main' is a function */

main:                 /* This is main */
       mov r0,#2      /* Put a 2 inside the register r0 */
       bx lr          /* Return from main */
```

Use following commands in command line to assemble the file.

```
$ arm-linux-gnueabi-as -o first.o first.s
```

This will create a first.o. Now link this file to get an executable.

```
$ arm-linux-gnueabi-gcc -o first first.o
```

If everything goes as expected you will get a first file. This is your program. Run it.

```
$ qemu-arm -L /usr/arm-linux-gnueabi/ ./first
```

```
$ qemu-arm -L /usr/arm-linux-gnueabi/ ./first ; echo $?
```

These are comments. Comments are enclosed in /* and /. *Use them to document your assembler as they are ignored. As usually, do not nest* / and / *inside* / because it does not work.

```
.global main /* 'main' is our entry point and must be global */
```

This is a directive for GNU Assembler. A directive tells GNU Assembler to do something special. They start with a dot (.) followed by the name of the directive and some arguments. In this case we are saying that main is a global name. This is needed because the C runtime will call main. If it is not global, it will not be callable by the C runtime and the linking phase will fail.

```
.func main   /* 'main' is a function */
```

Another GNU assembler directive. Here we state that main is a function. This is important because an assembler program usually contains instructions (i.e. code) but may also contain data. We need to explicitly state that main actually refers to a function, because it is code.

```
main:           /* This is main */
```

Every line in GNU Assembler that is not a directive will always be like label: instruction. We can omit label: and instruction (empty and blank lines are ignored). A line with only label:, applies that label to the next line (you can have more than one label referring to the same thing this way). The instruction part is the ARM assembler language itself. In this case we are just defining main as there is no instruction.

```
    mov r0, #2 /* Put a 2 inside the register r0 */
```

Whitespace is ignored at the beginning of the line, but the indentation suggests visually that this instruction belongs to the main function. This is the mov instruction which means move. We move a value 2 to the register r0. In the next chapter we will see more about registers, do not worry now. Yes, the syntax is awkward because the destination is actually at left. In ARM syntax it is always at left so we are saying something like move to register r0 the immediate value 2. We will see what immediate value means in ARM in the next chapter, do not worry again.

In summary, this instruction puts a 2 inside the register r0 (this effectively overwrites whatever register r0 may have at that point).

```
    bx lr       /* Return from main */
```

This instruction bx means branch and exchange. We do not really care at this point about the exchange part. Branching means that we will change the flow of the instruction execution. An ARM processor runs instructions sequentially, one after the other, thus after the mov above, this bx will be run (this sequential execution is not specific to ARM, but what happens in almost all architectures). A branch instruction is used to change this implicit sequential execution. In this case we branch to whatever lr register says. We do not care now what lr contains. It is enough to understand that this instruction just leaves the main function, thus effectively ending our program.

And the error code? Well, the result of main is the error code of the program and when leaving the function such result must be stored in the register r0, so the mov instruction performed by our main is actually setting the error code to 2.

Registers

At its core, a processor in a computer is nothing but a powerful calculator. Calculations can only be carried using values stored in very tiny memories called registers. The ARM processor in a Raspberry Pi has 16 integer registers and 32 floating point registers. A processor uses these registers to perform integer computations and floating point computations, respectively. We will put floating registers aside for now and eventually we will get back to them in a future installment. Let’s focus on the integer registers.

Those 16 integer registers in ARM have names from r0 to r15. They can hold 32 bits. Of course these 32 bits can encode whatever you want. That said, it is convenient to represent integers in two’s complement as there are instructions which perform computations assuming this encoding. So from now, except noted, we will assume our registers contain integer values encoded in two’s complement.

Basic arithmetic

Almost every processor can do some basic arithmetic computations using the integer registers. So do ARM processors. You can ADD two registers. Let’s retake our example from above:

```
/* -- sum01.s */
.global main
.func main

main:
    mov r1, #3      /* r1 ← 3 */
    mov r2, #4      /* r2 ← 4 */
    add r0, r1, r2  /* r0 ← r1 + r2 */
    bx lr
```

If we compile and run this program the error code is, as expected, 7.