Lab 1

The target machine : RISCV

Objective

- Be familiar with the RISCV instruction set.
- Install the RISCV toolchain and simulator.
- Understand how it executes on the RISCV processor with the help of a simulator.
- Write simple programs, assemble, execute.

1.1 The RISCV processor, instruction set, simulator

EXERCISE #1 ► **Lab preparation**
If you haven't done so already, follow the instructions to install `riscv-xxx-gcc` and `spike` on your machine (see `INSTALL.md` file).

EXERCISE #2 ► RISCV **C-compiler and simulator, first test**
In the directory `TP01/riscv/`:

- Compile the provided file `ex1.c` with:
  `riscv64-unknown-elf-gcc ex1.c -o ex1.riscv`
  It produces a RISCV binary named `ex1.riscv`.
- Execute the binary with the RISCV simulator:
  `spike pk ex1.riscv`
  This should print:
  `bbl loader`
  `42`
  If you get a runtime exception, try running `spike -m100 pk ex1.riscv` instead: this limits the RAM usage of spike to 100 MB (the default is 2 GB).
- The corresponding RISCV code can be obtained in a more readable format by:
  `riscv64-unknown-elf-gcc ex1.c -S -o ex1.s -fverbose-asm`
  (have a look at the generated .s file!)

The objective of this sequence of labs is to design **our own (subset of) C compiler for** RISCV.

EXERCISE #3 ► **Documents**
Some documentation can be found in the RISCV ISA (`riscv_isa.pdf` on the course webpage.

`https://compil-lyon.gitlabpages.inria.fr/`

In the architecture course, you already saw a version of the target machine RISCV. The instruction set is depicted in Appendix **??**.

1.1.1 Hand exercises

EXERCISE #4 ► **TD**
On paper, write (in RISCV assembly language) a program which initializes the t_0 register to 1 and increments it until it becomes equal to 8.

EXERCISE #5 ► **TD : sum**
Write a program in RISCV assembly that computes the sum of the 10 first positive integers (excluded 10).

1.1.2 Assembling, disassembling

EXERCISE #6 ► **Hand assembling, simulation of the hex code**
Assemble by hand (on paper) the instructions:

```
        .globl main
main:
         addi a0, a0, 1
         bne a0, a0, main
end:
         ret
```

You will need the set of instructions of the RISCV machine and their associated opcode. All the info is in the ISA documentation.

To check your solution (**after** you did the job manually), you can redo the assembly using the toolchain:

```
riscv64-unknown-elf-as -march=rv64g asshand.s -o asshand.o
```

`asshand.o` is an ELF file which contains both the compiled code and some metadata (you can try `hexdump asshand.o` to view its content, but it's rather large and unreadable). The tool `objdump` allows extracting the code section from the executable, and show the binary code next to its disassembled version:

```
riscv64-unknown-elf-objdump -d asshand.o
```

Check that the output is consistent with what you found manually.

EXERCISE #7 ► **Hand disassembling**
Guess a RISCV program that assembles itself into :

Listing 1.1: disass.lst

```
disass.o: format de fichier elf64-littleriscv


Déassemblage de la section .text:

0000000000000000 <main>:
   0: 00128313  xx
   4: ffdff06f  yy
   8: 00008067  zz
```

From now on, we are going to write programs using an easier approach. We are going to write instructions using the RISCV assembly.

1.2 RISCV **Simulator**

EXERCISE #8 ► **Execution and debugging**
See `https://www.lowrisc.org/docs/tagged-memory-v0.1/spike/` for details on the Spike simulator.

`test_print.s` is a small but complete example using Risc-V assembly. It uses the `println_string`, `print_int`, `print_char` and `newline` functions provided to you in `libprint.s`. Each function can be called with `call print_...` and prints the content of register `a0` (`call newline` takes no input and prints a new-line character).

1. First test assembling and simulation on the file `test_print.s`:
   `riscv64-unknown-elf-as -march=rv64g test_print.s -o test_print.o`
2. Optionally, run `riscv64-unknown-elf-objdump -D` as in previous exercise. The `-D` option shows all sections, including `.rodata`.
3. The `libprint.s` library must be assembled too:

```
riscv64-unknown-elf-as -march=rv64g libprint.s -o libprint.o
```

4. We now link these files together to get an executable:
   ```
   riscv64-unknown-elf-gcc test_print.o libprint.o -o test_print
   ```
   The generated `test_print` file should be executable, but since it uses the Risc-V ISA, we can't execute it natively (try `./test_print`, you'll get an error like `Exec format error`).
5. Run the simulator:
   ```
   spike pk ./test_print
   ```
   The output should look like:
   ```
   bbl loader
   HI MIF08!
   42
   a
   ```
   The first line comes from the simulator itself, the next two come from the `println_string`, `print_int` and `print_char` calls in the assembly code.
6. We can also view the instructions while they are executed:
   ```
   spike -l pk ./test_print
   ```
   Unfortunately, this shows all the instructions in pk (Proxy Kernel, a kind of mini operating system), and is mostly unusable. Alternatively, we can run a step-by-step simulation starting from a given symbol. To run the instructions in `main`, we first get the address of `main` in the executable:
   ```
   $ riscv64-unknown-elf-nm test_print | grep main
   000000000001014c T main
   ```
   This means: `main` is a symbol defined in the `.text` section (T in the middle column), it is global (capital T), and its address is 1014c (you may not have the same address). Now, run spike in debug mode (`-d`) and execute code up to this address (`until pc 0 1014c`, i.e. "Until the program counter of core 0 reaches 1014c"). Press Return to move to the next instruction and `q` to quit:
   ```
   $ spike -d pk ./test_print
   : until pc 0 1014c
   bbl loader
   :
   core   0: 0x000000000001014c (0xff010113) addi    sp, sp, -16
   :
   core   0: 0x0000000000010150 (0x00113423) sd      ra, 8(sp)
   :
   core   0: 0x0000000000010154 (0x0000e517) auipc   a0, 0xe
   :
   core   0: 0x0000000000010158 (0x41450513) addi    a0, a0, 1044
   : q
   $
   ```

Remark: For your labs, you may want to assemble and link with a single command (which can also do the compilation if you provide `.c` files on the command-line):

```
riscv64-unknown-elf-gcc -march=rv64g libprint.s test_print.s -o main
```

In real-life, people run compilation+assembly and link as two different commands, but use a build system like a `Makefile` to re-run only the right commands.

EXERCISE #9 ► **Algo in** RISCV **assembly**

Write (in `minmax.s`) a program in RISCV assembly that computes the min of two integers, and stores the result in a precise location of the memory that has the label `min`. Try with different values. We use 64 bits of memory to store ints, i.e. use `.dword` directive and `ld` and `sd` instructions.

EXERCISE #10 ► **(Advanced) Algo in** RISCV **assembly**

Write and execute the following programs in assembly:

- Count the number of non-nul bits of a given integer, print the result.

- Draw squares and triangles of stars (character '*') of size n, n being stored somewhere in memory.
  Examples:
  n=3 square:
  ```
  ***
  ***
  ***
  ```
  n=3 triangle:
  ```
    *
   * *
  * * *
  ```
  The function `print_char` expects the ASCII encoding of the character you want to print.