Lab 0

Warm-up : the target machine : RISCV

Objective

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

0.1 The RISCV processor, instruction set, simulator

EXERCISE #1 ► **Lab preparation**
Save your modifications from the last lab, then pull the lab repository.

EXERCISE #2 ► **Installations - Linux machines at Esisar**
On the Esisar machines, all installations have been done for you. However, you might have to add some stuff in your `PATH`.

EXERCISE #3 ► **Installations - on your machines**
See the `README.md` in the repository.

EXERCISE #4 ► RISCV **C-compiler and simulator, first test**
In the directory `TP_RISCV/startup/`:

- Compile the provided file `ex1.c` with:
  `riscv64-unknown-elf-gcc ex1.c -o ex1.riscv`
  It produces a RISCV binary.
- Execute the binary with the RISCV simulator:
  `spike pk ex1.riscv`
  This should print 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 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 the CS444 sequence of labs will be to design **our own (subset of) C compiler for** RISCV.

EXERCISE #5 ► **Documents**
Some documentation about the RISCV ISA can be found on the Chamilo webpage.

EXERCISE #6 ► **A first** RISCV **program**
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.

0.1.1 Assembling, disassembling

EXERCISE #7 ► **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.

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

0.2 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 newline 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. The `libprint.s` library must be assembled too:
   `riscv64-unknown-elf-as -march=rv64g libprint.s -o libprint.o`
3. 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`).
4. Run the simulator:
   `spike pk ./test_print`
   The output should look like:
   ```
   bbl loader
   HI CS444!
   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.
5. 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. 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 fact, complete `minmax.s` between TODO and END TODO) 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 ▸ Routines in assembly (read-only exercise)

In `len.s` we give you an exemple of a routine and a call to this routine (using a stack). Read and explain why we need to save the ra register on the stack. Illustrate for instance by removing the routine prelude and postlude and making a call to an external printing function.

EXERCISE #11 ▸ Caesar code

(Only) This exercise will be evaluated. Instructions on Chamilo.
Create a file named `str_codecesar.s`, starting with:

```
 # Code de César en RISCV
 # CS444, binôme : NOM1, NOM2
.section .text
.globl main
main:
        addi    sp,sp,-16
        sd      ra,8(sp)

        ...
```

A chain s being stored in memory, as well as a dec number, compute the Caesar code of the chain: shift every letter value by dec.

Your code should print the input string and the encoded one (thanks to a call to `print_string`. For instance, with the Hello World chain and a dec equal to 4:

```
Hello world!
Lipps${svph%
```