GRENOBLE INP UGA Esisar VALENCE

Architecture (CE313)

TP1, 2022-23

Contents

Lab 1
Programming in RISCV assembly 1/2

Objective

Credits: startup lab for compilation courses in Lyon and Valence, in collaboration with Matthieu Moy.

- 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.
- Pair working only. (not more) **Chamilo deposit exercise number XXX : see Chamilo for instructions.**

1.1 Startup: assembling, desassembling, simulating

EXERCISE #1 ▶ **Lab preparation**
Boot on Linux !. Get the archive on chamilo, then extract:

```
tar xvzf archivename.tgz
```

On the Esisar (linux) machines, everything is already installed. If you want to install on your machines, follow the instructions on this webpage (but not during a lab session!)

`https://forge.univ-lyon1.fr/matthieu.moy/mif08-2021/-/blob/main/INSTALL.md`

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

- 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!)

EXERCISE #3 ▶ **Documents**
Some documentation can be found in the RISCV ISA on Chamilo.

1.1.1 Assembling, disassembling

EXERCISE #4 ▶ **Hand assembling, simulation of the hex code**
Assemble by hand (on paper) the instructions:

```
1        .globl main
2 main:
3         addi a0, a0, 1
4         bne a0, a0, main
5 end:
6         ret
```

You will need the set of instructions of the RISCV machine and their associated opcode. All the info is in the (mini) 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.

1.1.2 RISCV **Simulator**

Code source is again in directory `TP01/startup`

EXERCISE #5 ► **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 [1]:
   `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 CE313
   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 (you may not have the same address, so **write somewhere yours**). 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:

[1] you can use any name, and/or add an extension such that .exe or .riscv for your binaries, we do not mind

```
$ 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: 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.

1.2 Let us program!

Source code is now in `TP01/riscv`.

1.2.1 Simple programs to complete

EXERCISE #6 ▶ **MinMax**
During the course, we wrote a program `minmax` that computes the maximum of two 64 bits integers in memory.

- Verify that it actually works (perhaps add some `print_int` calls).
- Extend this program so that it also computes the minimum. Test.

EXERCISE #7 ▶ **Array sum**
In the exercice session we wrote a routine to increment each element of an array by a constant 1. A solution is given in `arrayinc.s`.

- Read, understand, test this program.
- Copy the source into a new file:
  `cp arrayinc.s arraysum.s`
- Edit the new file and modify so that it computes the sum of all elements of the array.

EXERCISE #8 ▶ **Palindromes**
Inspired by `http://home.wlu.edu/~lambertk/classes/210/exercises/hw7.htm`.

A given string is a palindrome if it is equal to its mirror string. For instance *kayak* and *noon* are palindroms.
In pseudo-C, the following pseudo-code actually computes whether a given word is palindromic:

```
bool isPalindrome(char *string){
    int left,right ;
    left = 0 ;
    right = len(string) - 1 ;
    while (left < right){
        if (string[left] != string[right])
            return(false);
        left += 1;
        right -= 1;
    }
    return(true)
}
```

The objective is to write the equivalent in RISCV assembly. We give you a squeleton of code in `TP01/riscv/ispal.s`. A length routine is given.

- Test the `length` routine (call from the main).
- Complete the `ispal` routine, and test.
- Explain why storing `ra` on the stack is mandatory, for which routine ?

1.2.2 Programs "from scratch"

EXERCISE #9 ▶ **Integer sum**
n being stored in memory, write a program that computes the sum $1+2+\ldots+n$, and prints the result.

EXERCISE #10 ▶ **Caesar code**

Create a file named `codecesar.s`, starting with:

```
 # Code de César en RISCV
 # CE313, 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%
```

Please use a routine as shown in the palindrome exercise.