Lab 1

Objective: To practice and revisit program compilation and disassembly.

1 Compilation

Compiler is essential to bridge the gap between programs written in high-level languages such as C and the processor. In this question, you will experiment with the basics of the GNU C compiler (`gcc`) that is ported to compile for the RISCV processor.

1.1 Preliminaries You need to perform this exercise using the `riscv-gnu-toolchain`, which includes the GNU compilation toolchain with tools such as `gcc`, `objdump`, `gas`, etc. The tools are designed to produce RISCV instructions even when you run them on an Intel based machine – a process that is called **cross compilation**.

To conform to the GNU naming convention for cross compiler, you will find that the tools we are using has a rather long prefix in the name, such as:

- `riscv32-unknown-elf-gcc`
- `riscv32-unknown-elf-objdump`
- `riscv32-unknown-elf-gas`

In Linux systems running a modern shell such as `bash`, you can simply press `<TAB>` to complete the file name. That will save you a lot of typing.

The simplest way to access these tools is to log in to our own server for this course, which has all the tools already installed. To do so, follow instructions in 1.2 and 1.3.

Alternatively, you may install the RISCV toolchain yourself if you are confident in Linux. You will need to clone the tools from `https://github.com/riscv/riscv-tools`. Ask questions on Piazza for help.

1.2 Log in to the server You can access the server `tux-1.eee.hku.hk` via `ssh`:

```
yourmachine$ ssh your_eee_login@tux-1.eee.hku.hk
```

This will give you a text-only command line interface. If you need access to a GUI-based environment (e.g. to use the Logisim installed on tux-1), you will have to use the remote desktop software x2go. The instructions for setting up x2go are posted on Piazza.

1.3 Setup Environment Before you can start working on the homework, you have to set up your UNIX environment to include the RISC-V tool chain in your PATH. Source the class bashrc for that purpose:

```
tux-1$ source ~elec3441/elec3441.bashrc
```

You can save that line in your own `.bashrc` if you want to have the environment set up automatically every time you log in.

If you see errors such as `riscv32-unknown-elf-gcc: command not found`, it is generally because you have not executed this line.

1.4 Get the File Copy and untar the source file to your own account.

```
tux-1$ tar xzvf ~elec3441/elec3441lab1.tar.gz
tux-1$ export LAB1ROOT=~/elec3441lab1
```

If you want to work on your own machine, you can also download the files from the following URL:

`http://www.eee.hku.hk/~elec3441/sp24/handout/elec3441lab1.tar.gz`

From now on, the directory you have expanded will be referred to as `${LAB1ROOT}`. The files for this part of the homework are located in `${LAB1ROOT}/compilation`.

1.5 Compile the file `ArrayAccu.c` to RISC-V assembly code using:

```
tux-1$ riscv32-unknown-elf-gcc -O1 -S ArrayAccu.c -o ArrayAccu.s
```

The switch `-S` tells `gcc` to produce human readable assembly code instead of machine readable binaries. The switch `-o` tells `gcc` to store the output in the file `ArrayAccu.s`. Finally, the `-O1` (capital letter Oh) tells the compiler to perform optimization. Different level of optimization can be specified with `-O1`, `-O2`, or no optimization by omitting the switch.

Look at the content of `ArrayAccu.s` and answer the following questions:

1. The program starts running with the C function `main`. In `ArrayAccu.s`, where is `main` is defined?
2. Search through `ArrayAccu.s`, where is `main` being called?
3. The `ArrayAccu()` function is called within `main`. Look at the code before the function is called, and the code **inside** `ArrayAccu()`, where are the input arguments and output argument stored?
4. Can you see how the return address `ra` being used?

1.6 Check Yourself

Make sure have completed the steps before coming to the lab. Be prepared to show your work during the lab.

1.7 Branches Compile the 2 files `branch1.c` and `branch2.c` as follows: Save the following 2 code segment as 2 different files. Now, compile them using different optimization level:

```
tux-1$ riscv32-unknown-elf-gcc -O0 -S branch1.c -o branch1_0.s
tux-1$ riscv32-unknown-elf-gcc -O0 -S branch2.c -o branch2_0.s
tux-1$ riscv32-unknown-elf-gcc -O1 -S branch1.c -o branch1_1.s
tux-1$ riscv32-unknown-elf-gcc -O1 -S branch2.c -o branch2_1.s
tux-1$ riscv32-unknown-elf-gcc -O2 -S branch1.c -o branch1_2.s
tux-1$ riscv32-unknown-elf-gcc -O2 -S branch2.c -o branch2_2.s
```

Read through the compiled code and answer the following questions:

1. Does the compiler implement the two branches differently? How are they different?
2. How many instructions are needed for the two branches?
3. Does the generated code differ with -O1 -O2 and no optimization?

1.8 Loops Compile the 2 files `loop1.c` and `loop2.c` using different optimization level similar to the above questions. Then answer the following questions:

1. How does the compiler implement the two different loops?
2. How many instructions are needed for the two loops?
3. How does the generated code differ with -O1 -O2 and no optimization?

1.9 If-then-else vs. Switch Compile the files `branch.c` and `case.c` as before and answer the following questions:

1. How does the compiler implement the following two code segment differently?
2. How does the generated code differ with -O1 -O2 and no optimization?

1.10 Function Calls Compile and compare the two files `funcall1.c` and `funcall2.c`, then answer the following questions:

1. How does the compiler implement the following two code segment differently?
2. How does the generated code differ with -O1 -O2 and no optimization?

1.11 Submission Submit a short answer to the answer from 1.8 to 1.10 on Moodle.