assign1.md

Assignment #1: Playing with RISC-V Compiler and Emulator

- SNU ECE-430.322
- Computer Organization, 2020 Spring
- **Due date: 11:59pm, May 13, 2020**

In this assignment, you will be playing with RISC-V compiler and emulator.

Notes on Running Environment for Assignment #1

This assignment assumes that you are running an x86-based Linux machine. As you have already installed the vagrant running Ubuntu (for lab projects), you may use it for this assignment. You can use your own Linux box as well if you have one.

Install: RISC-V Compiler

- Prepare GNU Toolchain for RISC-V. Check The xPack GNU RISC-V Embedded GCC for more details. Here `BASE` denotes the base directory that you will be using throughout this assignment, and you can pick any directory for this `BASE`.

Since my machine is Intel x86 (assume all of yours are the same as well), what we do here is so-called cross-complilation. That is, the host platform is Intel x86 (which runs the compilation process) while the target platform is RISC-V (for which the compilation generates an execuable). It is called cross-compilation because the host and target platforms are different.

```
$ cd BASE
BASE $ wget http://compsec.snu.ac.kr:40404/downloads/xpack-riscv-none-embed-gcc-8.2.0-3.1-linux-x64.tgz
BASE $ tar zxvf xpack-riscv-none-embed-gcc-8.2.0-3.1-linux-x64.tgz
```

Now all the binaries for RISC-V compliation are placed in `BASE/xPacks/riscv-none-embed-gcc/8.2.0-3.1/bin`. For easy access to these binaries (i.e., execute the binary anywhere without typing this long path), let's setup the PATH environment variable. In short, this PATH environment variable setup allows you to run executables in `BASE/xPacks/riscv-none-embed-gcc/8.2.0-3.1/bin` without specifying the long directory path. You will need to setup this PATH environment everytime you spawn a new shell prompt.

```
BASE $ export PATH=`pwd`/xPacks/riscv-none-embed-gcc/8.2.0-3.1/bin:$PATH
```

If you can see the following message, your RISC-V compiler installation should be correct.

```
$ riscv-none-embed-gcc -v 2>&1 | tail -n1
gcc version 8.2.0 (xPack GNU RISC-V Embedded GCC, 64-bit)
```

Install: RISC-V Emulator

- Prepare RISC-V RV32I emulator, rv32emu. `rv32emu` takes an RISC-V executable as input, and emulates the execution as if the underlying CPU is RISC-V.

```
$ cd BASE
BASE $ wget http://compsec.snu.ac.kr:40404/downloads/rv32emu.tar.gz
BASE $ tar zxvf ./rv32emu.tar.gz
BASE/rv32emu $ cd rv32emu
```

```
BASE/rv32emu $ sudo apt-get install libelf-dev
BASE/rv32emu $ make CC=gcc-7
```

If your build is right, you should be able to see the `rv32emu` executable, `emu-rv32i`. If you can see the message below, your `rv32emu` should be good to go.

```
BASE/rv32emu $ file ./emu-rv32i
./emu-rv32i: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/l, for GNU/Linux
3.2.0, BuildID[sha1]=4a955df470c34a2d739ba342335696f1c9e04b20, not stripped
```

Compiling Your First RISC-V Program

Now we will play with our first RISC-V program, `first.c`.

```
$ cd BASE/rv32emu
BASE/rv32emu $ cat ./first.c
void _start()
{
    volatile char* tx = (volatile char*) 0x40002000;
    const char* hello = "Hello RISC-V!\n";
    while (*hello) {
        *tx = *hello;
        hello++;
    }
}
```

Two interesting things can be seen in this program: 1) No `main()`, but `_start()` and 2) No `printf()`, but writing to the address `0x40002000`.

First, `first.c` only has `_start()` and it does not have `main()` that you are (probably) familiar with. Since `first.c` would not be running on top of the underlying operating system, we don't want any code which may rely on the operating system. Since there are many of such code between `_start` (which is the true entry point of an executable) and `main` (which is the pseudo entry point of the executable), `first.c` takes the complete control over the execution by declaring `_start()` such that the very first instruction to be executed (by `rv32emu`) is the first instruction in `_start()`.

Second, `first.c` does not have `printf()` function. Instead, it relies on writing a char value to the address `0x40002000`. That is, as you write any byte value to this address, it will be printed back to your terminal running `rv32emu`. Since `rv32emu` does emulate full-fledged I/O devices, it simply emulates an UART interface at the address `0x40002000` to support print-like features.

To complete this assignment, you may not need to understand all the details. You will learn more when taking the operating systems class.

The command below compiles `first.c`.

```
BASE/rv32emu $ riscv-none-embed-gcc -march=rv32i -mabi=ilp32 -nostdlib ./first.c -o first
```

The command above uses the compiler, `riscv-none-embed-gcc` to build `first.c`. In this command, `-march=rv32i` specifies which ISA to be used, `-mabi=ilp32` specifies which ABIs to be used (see RISC-V ABIs for more details), `-nostdlib` specifies that this complilation won't link the standard startup and libraries (which implies that the entry point would be `_start()` not `main()`).

Emulating Your First RISC-V Program

Once you have the executable `first`, you can execute it (more precisely, emulate it) using `rv32emu`.

```
BASE/rv32emu $ ./emu-rv32i first
Hello RISC-V!
```

```
>>> Execution time: 29144 ns
>>> Instruction count: 62 (IPS=2127367)
>>> Jumps: 14 (22.58%) - 0 forwards, 14 backwards
>>> Branching T=13 (92.86%) F=1 (7.14%)
```

The first line of the output shows the message (printed through the emulated UART interface). Output lines starting with `>>>` are log messages generated by `rv32emu`, which summarizes the execution results.

Task1. Implement Iterative Bubble

Implement your iterative bubble sort algorithm by filling up `bubble_iter.c`. In this task, you should implement your bubble in an iterative way.

If you think your implementation is done, you can build your bubble like below. (Or you may run `make bubble_iter` that we prepared for you. Have a look at `Makefile`).

```
BASE/rv32emu $ riscv-none-embed-gcc -march=rv32i -mabi=ilp32 -O0 -nostdlib -o bubble_iter bubble_iter.c
```

If your bubble implementation is right, you will see the output like below. No need to have the same lines starting with `>>`. We intentionally blinded these lines with `----`.

```
BASE/rv32emu $ ./emu-rv32i ./bubble_iter
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
88 89 90 91 92 93 94 95 96 97 98 99
>>> Execution time: ----
>>> Instruction count: ----
>>> Jumps: ----
>>> Branching ----
```

Note: You won't be able to run instructions other than RV32I. This suggests that you won't be able to use multiply (i.e., MUL in RISC-V) or divide instructions. As such, your bubble should do all the things with RV32I.

Note: You are not allowed to change `Makefile` (even if you do so, this assignment won't get easier).

Task2. Implement Recursive Bubble

Implement your recursive bubble sort algorithm by filling up `bubble_recur.c`. In this task, your bubble implementation should be recursive.

Build and running this recursive bubble is similar to the iterative one.

```
BASE/rv32emu $ riscv-none-embed-gcc -march=rv32i -mabi=ilp32 -O0 -nostdlib -o bubble_recur bubble_recur.c
BASE/rv32emu $ ./emu-rv32i ./bubble_recur
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
88 89 90 91 92 93 94 95 96 97 98 99
>>> Execution time: ----
>>> Instruction count: ----
>>> Jumps: ----
>>> Branching ----
```

Task3. A Static Look with Disassembly

This task asks you to write `report.pdf` after looking at the RISC-V assembly of `bubble_iter` and `bubble_recur`. You can get the assembly of these using objdump.

```
BASE/rv32emu $ riscv-none-embed-objdump -d ./bubble_iter

./bubble_iter:     file format elf32-littleriscv

Disassembly of section .text:

00010074 <bubble_sort_iter>:
   10074:       fd010113                addi    sp,sp,-48
   10078:       02812623                sw      s0,44(sp)
   ...
   ...
   ...
```

When writing the report, we need you to answer following questions (you don't need to write any other stuffs):

- Q1. How are arguments of `bubble_sort_iter()` and `bubble_sort_recur()` maintained in the stack?
- Q2. Does `bubble_sort_iter()` and `bubble_sort_recur()` use JAL, JALR, or both?
- Q3. How does `bubble_sort_iter()` and `bubble_sort_recur()` restore the stack before returning to a caller function?
- Q4. What is `ret` instruction shown in `objdump`? (RISC-V ISA does not have `ret` instruction

Task4. A Dynamic Look with Emulator Statistics

This task asks you to write `report.pdf` after looking at the emulator statistics. In particular, you will have a look at instruction counts when emulating `bubble_iter` (that you implemented in Task1) and `bubble_recur` (that you implemented in Task2).

To get the instruction counts from `rv32emu`, you will need to modify `emu-rv32i.c` ---i.e., you will need to declare `DEBUG_EXTRA` in `emu-rv32i.c`. After this modification, you will need to rebuild `rv32emu` by running the `make` command like below.

```
BASE/rv32emu $ make CC=gcc-7
```

Using this modified version of `rv32emu`, now you can get the instruction count if you run `emu-rv32i`.

```
BASE/rv32emu $ ./emu-rv32i bubble_iter
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
88 89 90 91 92 93 94 95 96 97 98 99

Registers:
x0 zero: 00000000
x1 ra:   00000000
x2 sp:   00020074
...

Instructions Stat:
LUI     = 3
JAL     = 462
JALR    = 3
...

Five Most Frequent:
1) LW   = 78413 (46.31%)
2) ADDI = 21641 (12.78%)
3) SLLI = 19505 (11.52%)
...
```

- Q5. Shortly explain how `emu-r32i.c` implements the RISC-V CPU emulation (i.e., how it emulates many different RISC-V instructions without running on a real RISC-V CPU). Your answer should be less than 40 words.
- Q6. Compare the instructions counts between `bubble_iter` and `bubble_recur`. What's the notable differences between these two?
- Q7. Why do you think the differences in Q6 is observed? Relate your answer with the differences between iteration and resursion.

Submission

Prepare your submission file, `assign1.zip`, which zips **only** following three files: `bubble_iter.c` (Task1), `bubble_recur.c` (Task2), `report.pdf` (Task3 and Task4). Once your `assign1.zip` is ready, submit it through our submission website.