assign2.md

Assignment #2: Extending RISC-V ISA for Hash Computation

- SNU ECE-430.322
- Computer Organization, 2020 Spring
- **Due date: 11:59pm, May 23, 2020**

In this assignment, you will be extending RISC-V ISA, particularly the RISC-V emulator, which computes a simple hash function.

Notes on Running Environment for Assignment #2

For this Assignment #2, the compiler setup is the same as Assignment #1 but the emulator is not. You should download `rv32emu-assign2.tar.gz` and unzip it to the directory `BASE/rv32emu-assign2`.

Install: RISC-V Compiler

Follow the instruction that we provided in Assignment #1. Don't forget to setup the `PATH` environment variable.

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

As you have done in Assignment #1, 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

You will need to do the following to prepare the emulator. Different from Assignment #1, notice that we are going to use `rv32emu-assign2.tar.gz` for Assignment #2.

```
$ cd BASE
BASE $ wget http://compsec.snu.ac.kr:40404/downloads/rv32emu-assign2.tar.gz
BASE $ tar zxvf ./rv32emu-assign2.tar.gz
BASE/rv32emu-assign2 $ cd rv32emu-assign2
BASE/rv32emu-assign2 $ make CC=gcc-7
```

If you see the message like below, your `rv32emu` should be ready.

```
BASE/rv32emu-assign2 $ 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]=d0ac207601f702e9ad4f556f73d1818221190c75, not stripped
```

Unleased RV32M Instructions

In Assignment #1, you were not allowed to use multiply/divide since we were following the strict RV32I model. For Assignment #2, we will be enabling RV32M instrutcions. This is done by following two changes: 1) Changing compiler option: use `-march=rv32im` and 2) Enabling emulator's internal feature by dropping `STRICT_RV32I`.

The first one is related to how compiler's generating the code, and `-march=rv32im` instructs the compiler to use both `RV32I and RV32M` instruction. Note that in Assignment #1, you were asked to use `-march=rv32i`. So when you invoke the compiler, you will need to be careful to specifically provide `-march=rv32im`. In fact, it would be better to use `make` command (see `Makefile` that we provided, which is updated to have `-march=rv32im` for you).

The second one is related to `rv32emu`'s optional features. By default, `rv32emu` only supports RV32I instructions. If `STRICT_RV32I` is not defined, then `rv32emu` enables mlutiply/divide like instructions. You can check more details in `emu-rv32i.c`, searching for `STRICT_RV32I`. We already made the change in `emu-rv32i.c` for you.

Since RV32M is also enabled, we can run the multiply program, `multiply.c`.

```
BASE/rv32emu-assign2 $ cat ./multiply.c
void _start()
{
    volatile char* tx = (volatile char*) 0x40002000;
    char x = 2;
    char y = 3;
    char z = x * y;
    *tx = z + 0x30;
}
```

From Assignment #2, we will be compiling to-be-emulated program with following three stages, which will help you to clearly see the generated assembly in the middle for your executable.

- C code (`multiply.c`) will be first compiled into an assembly (`multiply.s`). The provided compiler option is `-S`.
- The assembly will be compiled into an object file (`multiply.o`). The provided compiler option is `-c`.
- Finally the object file will be linked into an executable (`multiply`).

```
BASE/rv32emu-assign2 $ riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -O0 -nostdlib -S -o multiply.s multiply.c
BASE/rv32emu-assign2 $ riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -O0 -nostdlib -c -o multiply.o multiply.s
BASE/rv32emu-assign2 $ riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -O0 -nostdlib -o multiply multiply.o
```

Once build is done, you can again run `emu-rv32i`, confirming that `emu-rv32i` now supports multiply instrucitons.

```
BASE/rv32emu-assign2 $ ./emu-rv32i multiply
6
>>> Execution time: 30664 ns
>>> Instruction count: 23 (IPS=750065)
>>> Jumps: 1 (4.35%) - 0 forwards, 1 backwards
>>> Branching T=0 (-nan%) F=0 (-nan%)
```

Hash Function Implementation with RV32IM

A hash function is any function that can be used to map data of arbitrary size to fixed-size values. This assignment will implement the very simple hash function, `hash_rv32m()` as shown below (this is a variant of MurmurHash).

```
// Implemented in hash.c

unsigned hash_rv32m(const char *key)
{
    unsigned hash_value = 0x23198485;
    for (;*key;++key) {
      hash_value^= *key;
      hash_value*= 0x5bd1e995;
      hash_value^= hash_value>> 15;
    }
    return hash_value;
}
```

This hash function takes a string `key` as input, and produces 32-bit `hash_value` as output. In particular, based on the initial `hash_value`, it keeps updating `hash_value` while iterating over the `key` string. All these are implemented within `hash.c` that we provided. We are not going to discuss details of the mathematical or compuational aspects of hash, but if you are interested, you may have a look at the Wikipedia article.

We implemented the RV32IM version in `hash.c`, which can be emulated with following commands.

```
BASE/rv32emu-assign2 $ make hash
riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -O0 -nostdlib -S -o hash.s hash.c
```

```
riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -O0 -nostdlib -c -o hash.o hash.s
riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -O0 -nostdlib -o hash hash.o

BASE/rv32emu-assign2 $ ./emu-rv32i ./hash
9988e12d
00fd5bcb

>>> Execution time: 687487 ns
>>> Instruction count: 2979 (IPS=4333172)
>>> Jumps: 162 (5.44%) - 26 forwards, 136 backwards
>>> Branching T=136 (91.28%) F=13 (8.72%)
```

The first output `9988e12d` is the hash of `Hello x86 Hello x86 Hello x86 Hello x86 Hello x86` and the second `00fd5bcb` is the hash of `Hello risc-v Hello risc-v Hello risc-v Hello risc-v Hello risc-v`.

Introducing New RISC-V ISA extension, RV32Z

This assignment asks you to extend RISC-V ISA, which we will call `RV32Z`. `RV32Z` is a custom ISA that we create for this assignment, particularly designed in mind to accelerate the hash computation. That is, `RV32Z` provides dedicated instruction sets to compute the hash such that the hash computation can be done much more efficiently. You will see that to compute the same hash function, the number of executed instructions with `RV32Z` will be much less than that of `RV32IM`.

Note: This custom ISA design is in fact quite popular and commonly used engineering tactics. To accelerate image processing, many mobile processors are extended with dedicated instructions sets. Particularly focusing on hash, you can also think about why Bitcoin miners use dedicated FPGA/ASIC devices to accelerate their mining process (Bitcoin minining is nothing but the hash computation---check more here).

`RV32Z` introduces three new instructions `hash_init`, `hash_update`, and `hash_digest`, all of which are `R-Type` instructions.

- `hash_init rd, rs1, rs2`: `hash_init` instructs to initialize the internal structure to compute hash (i.e., initializing the hash vlaue). `rd`, `rs1`, and `rs2` are ignored. `funct` is ignored as well.
- `hash_update rd, rs1, rs2`: `hash_update` updates. `REG[rs1]` should provide a char value (a single byte) to update the hash value. `rd` and `rs2` are ignored. `funct` is ignored as well.
- `hash_digest rd, rs1, rs2`: `hash_digest` finally returns the computed hash value through `rd`. `rs1` and `rs2` are ignored. `funct` is ignored as well.

Opcodes for RV32Z instructions are as follows.

RV32Z instruction	opcode
`hash_init`	1111100
`hash_update`	1111101
`hash_digest`	1111110

You must have a close look at `hash_rv32z.c` which implements all these `RV32Z` support to compute hash in `hash_rv32z()`. Note that `hash_rv32z.c` mixes up the assembly within the C code, which is called inline assembly.

To build `hash_rv32z`, you can do the followings.

```
BASE/rv32emu-assign2 $ make hash_rv32z
riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -O0 -nostdlib -S -o hash_rv32z.s hash_rv32z.c
riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -O0 -nostdlib -c -o hash_rv32z.o hash_rv32z.s
riscv-none-embed-gcc -march=rv32im -mabi=ilp32 -O0 -nostdlib -o hash_rv32z hash_rv32z.o
```

- If you try to look at the assembly of `hash_rv32z` using `objdump`, you won't be able to find `RV32Z` instructions. This is because `objdump` does not understand `RV32Z` so it doesn't know how to decode `RV32Z` instructions.
- Looking at the assembly compiled in `hash_rv32z.s` would be better, but you will see custom-defined macros (that we implemented in `hash_rv32z.c` to invoke RV32Z instructions.

Task1: Emulating RV32Z for Hash Computation

Your task is to modify `emu-rv32i.c` to support `RV32Z`. You should not modify any other files. More specifically, your modified emulator should be able to emulate `hash_rv32z` as follows. If your `emu-rv32i`'s `RV32Z` support is right, you should be able to do as follows.

```
BASE/rv32emu-assign2 $ ./emu-rv32i hash_rv32z
9988e12d
00fd5bcb

>>> Execution time: 410402 ns
>>> Instruction count: 1627 (IPS=3964405)
>>> Jumps: 162 (9.96%) - 26 forwards, 136 backwards
>>> Branching T=136 (91.28%) F=13 (8.72%)
```

Notice the first two lines in the output, which generated the same hash value as we have done with the `RV32IM` version (i.e., `hash.c`)

Tips: While playing with the emulator, you may enable debug macros like `DEBUG_OUTPUT` or `DEBUG_EXTRA`, which will provide much better information while developing your RV32Z support.

Task2: Report Write-up on RV32Z

This task asks you to write `report.pdf`, which answers following questions.

- Q1. Count the number of executed instructions in `hash` and `hash_rv32z`. Which one executes more instructions?
- Q2. We designed the custom `RV32Z` in hopes it would accelerate the performance of hash computation. Can you relate your answer in Q1 to justify why `RV32Z`-based hash computation would be faster than `RV32IM`-based hash computation?

Submission

Prepare your submission file, `assign2.zip`, which **only** zips following two files: `emu-rv32i.c` and `report.pdf`. Once your `assign2.zip` is ready, submit it through our submission website.