Computer Architecture Fall 2018
Assignment 2: ISA emulation

Deadline: Friday, November 9, 2018

In this second assignment you will develop an emulator that is capable of executing simple RISC-V programs. The RISC-V architecture has been chosen because of its simplicity. The main learning objective of this assignment is to acquire a deep understanding of how processors execute instructions and how the different categories of instructions are implemented.

We will limit ourselves to the execution of instructions. We will not concern ourselves with simulating the detailed characteristics of DDR memory nor will we model and simulate a cache subsystem. Within this assignment we will assume that a memory load/store operation can be performed in a single clock cycle. Of course, these things can be simulated in detail if the emulator were to be further extended.

You do not have to start from scratch. From BlackBoard a skeleton, or rather a starting point, can be downloaded. A number of things have already been implemented within this starting point: loading programs in ELF format, a simplified emulation of a memory bus and memory, register file and simple “devices” to output characters and to the halt the system.

During the development of the emulator it is very important to test your code. You will have to write test programs (unit tests) to test individual instructions. We will refer to these as micro programs. To help you get started, the starting point contains a test harness that will be elaborated on below.

To test the emulator in its entirety, a collection of more extensive test programs can be obtained from BlackBoard. These programs increase in complexity: `basic.bin`, `hello.bin`, `hellof.bin`, `comp.bin`, `matvec.bin`, `matvecu.bin`, `brainfuck.bin`, `mandel.bin`. These test programs will also be used to determine the correctness score of your program when grading the assignment.

Background Information

For general theory on Instruction Set Architectures we refer to Appendix A of the textbook (Hennessy and Patterson) and the lecture on Appendix A. In Appendix C the “classic RISC” pipeline is described, which consists of five steps to execute an instruction: Instruction Fetch, Instruction Decode, Execute, Memory, Write Back. In this assignment, we will process instructions according to these five steps. Appendix C.3 describes the implementation of a pipeline for a simple MIPS processor.

Detailed information on the RISC-V instruction set can be found in the architecture reference manual: `https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`.

Working on the Assignment

Important: to be able to compile the starting point a modern C++-compiler is required (at least g++ 5.x or Clang 3.5). The compiler that is available on the University workstations (Ubuntu 16) is recent enough. To be able to compile RISC-V programs (to create micro programs) the `ca2018` environment is required:

```
source /vol/share/groups/liacs/scratch/ca2018/ca2018.bashrc
```

As soon as you manage to compile the starting point: it is expected that the emulator without modifications gets stuck in an infinite loop. The given starting point is not functional and many essential parts are missing. Time to get started!

The Code Base in a Nutshell

The class `Processor` forms the core of the code base. Within this class the five steps of instruction execution are implemented. To implement these steps different components are put to use such as an instruction decoder, register file and ALU.

We recommend to start this assignment by working on the instruction decoder for RISC-V instructions. The task of an instruction decoder is easily described: given an *instruction word*, determine the *control signals* that are necessary to control the components in the *data path* (register, ALU, and so on). You will have to isolate different fields from the instructions. To do so, add fields to `struct DecodedInstruction` in `inst-decoder.h` in which you can store the results of a decoded instruction. This way `DecodedInstruction` is a representation of a decoded instruction.

The actual code for the instruction decoder must be placed in `inst-decoder.cc`. The method `decodeInstruction` must decode the given instruction, which is always 32 bits in size in our case. Properly structure your code and do not create a single, very large function! Use the fact that different instruction formats are distinguished: write subroutines to decode the different instruction formats. Avoid the use of "magic numbers" in the code and instead use enums and constants where desirable.

Ensure that when a malformed or illegal instruction is given (for instance a non-existent or non-implemented opcode) the exception `IllegalInstruction` is thrown.

You will notice that it will be useful to print the decoded instructions to the terminal, especially when debugging the emulator. To accomplish this, you will have to write a method that can print a `DecodedInstruction` to the terminal in a useful way. In fact, this boils down to writing a "mini disassembler". A start can be found in `inst-formatter.cc`. To enable the printing of instructions to the terminal the command-line option `-d` must be supplied.

After an instruction has been decoded, it can be executed. Most instructions need to perform an ALU operation. ALU operations are implemented in `alu.cc` and `alu.h`. Methods are present in the `ALU` class to configure the operands and to read the result. Left to be added are methods to configure the operation to be performed (for example by passing opcode and/or function code). After all values have been set, one can call the method `execute()` to perform the operation according to the set control signals. Once completed, the result can be read using the method `getResult()`.

Finally, all components must be interconnected with one another. This is done in the class `Processor` (in `processor.cc`, `processor.h`). In the header file can be seen that `Processor` contains a member variable for each component in the system. The methods according to the five steps of instruction execution must be implemented in `processor.cc`: `instructionFetch`, `instructionDecode`, and so on. For example, for "instruction decode" the current instruction must be decoded and subsequently the correct operands and codes must be set on the ALU. In the "execute" step the method `execute()` is called on the ALU to perform the configured operation.

Within this assignment we will assume that each of the five steps requires one clock cycle. To execute a reg-reg ALU instruction all five steps must be performed, therefore this takes five clock cycles.

Also note that the emulator only works on a single instruction at a time, so it is a non-pipelined execution! For this assignment, we restrict ourselves to non-pipelined execution of instructions.

Testing

We strongly recommend to implement the instructions one by one, or in small groups of similar instructions. For a given instruction, first implement decoding, followed by formatting, ALU and whatever is needed in `class Processor`. After that, this individual instruction can be tested. To do so, we recommend to write a micro program. It is possible to write micro programs that consist of just a single instruction. Look at `tests/add.s` and `tests/add.conf` as an example. In the `.conf`-file the pre- and post-conditions of the test are specified: so how the registers must be initialized and what values the registers must contain after completion of the program. Construct the pre- and post-conditions in such a way that you can ensure that the tested instruction functions correctly. When you want to add tests for a new instruction, say `nop`, you need to add files `nop.s` and `nop.conf`. If the RISC-V toolchain is installed on your machine, the command `make nop.bin` will generate the binary program. Subsequently, `make runtests` will execute all micro programs in the `tests` subdirectory.

It is also possible to manually execute a micro program using the special unit test mode of the emulator:

```
./rv64-emu -t tests/add.conf
```

A “normal” program can be executed as follows:

```
./rv64-emu ../rv64-examples/hello.bin
```

When the emulator exits, all current values of all registers are printed to the terminal. Using the option `-d` the instruction dump functionality may be enabled.

The First Steps

To help you get started, we suggest the following as the first steps to take towards a completed implementation:

- Start with making the necessary modifications such that the emulator is able to execute the `add` instruction (non-pipelined). This can be verified by running the provided example micro program in the `tests` subdirectory. Implement instruction fetch in the class `Processor` and increment `PC` each time with 4 bytes. Uncomment the code that checks for the “test end marker”. This marker is a special codeword, with the value `0xddffccff`, that will stop the emulator once this codeword is encountered (when in “unit test mode”). Note that if this marker were not present, the emulator would at some point read beyond the end of the memory (text segment in our case) which would lead to an abnormal program termination.
- As the next step, determine which instructions need to be implemented to be able to execute `basic.bin`.
- Implement all of these instructions one by one (non-pipelined). Write new and separate micro programs to test these instructions. Note that `basic.bin` is still a simple test that is terminated using the test end marker as described above.
- After `basic.bin`, continue with `hello.bin`. Note that the last instruction that is executed by `hello.bin` is a store instruction to a specific memory address. At this particular memory address a system control module is connected. This module ensures that the emulator will halt and is cleanly shutdown. So, a correct execution of `hello.bin` will not lead to an “abnormal program termination”, instead “System halt requested” should appear in the terminal.
- Continue in the same way for the more complicated test programs.

Floating-point Instructions

For the base assignment the maximum grade that can be obtained is 9. In order to obtain a 10, you will have to show us that you are capable of adding new features to the emulator independently, without extensive guidance from the instructors.

This year, the feature that needs to be implemented in order to obtain the grade 10 is floating-point support, such that the emulator becomes capable of running the Mandelbrot example when compiled using floating-point instructions.

Note that in the collection of test programs `mandel.bin` has been compiled using the soft-float ABI. This means that `mandel.bin` does not contain any floating-point instruction. Instead, these instructions are emulated in software using solely integer instructions. So, start with recompiling `mandel.c` using the toolchain that is installed on the University workstations. Ensure that the resulting executable contains floating-point instructions. Subsequently, implement these instructions and make sure that the emulator is capable of executing this test program without problems.

Submission and Grading

Teams may be formed that consist of at most *two* persons. The following needs to be submitted:

- Source code of the emulator. Please do *not* submit binaries or object files (with the exception of micro programs). Use `make clean`! Make sure any comments in your code are written in English.
- A concise report in text format, `report.txt`, written *in English.* The report must describe:
  - How the different components are controlled from `class Processor` and which choices you have made for your implementation.
  - Which parts of your emulator do and do not work.
  - Which test programs the emulator can execute without problems.
  - In case you have worked in a team of two: each student should write a paragraph *individually* which describes your contribution to the project (what parts did you work on?) and how you experienced the collaboration with your lab partner (was the collaboration efficient? Did your lab partner put in enough effort in your opinion? Etc.).
  - In case you have implemented floating-point instructions: describe which instructions have been implemented and what modifications were necessary to achieve this.

Ensure that all files that are submitted (source code, report, etc.) contain your names and student IDs! Please create a "gzipped tar" archive of your source code:

```
tar -czvf opdracht2-sXXXXXXX-sYYYYYYY.tar.gz opdracht2/
```

Substitute `XXXXXXX` and `YYYYYYY` with your student IDs. Submit the tar-archive and your report through the BlackBoard submission site. Also note your names and student IDs in the text box in the submission website.

Deadline: Friday, November 9, 2018.

The maximum grade that can be obtained is 10. The grade is the sum of the scores for the following components, maximum score between square brackets:

- [**2 out of 10**] Quality of the submission: report, layout and modularity of the code, quality of the instruction formatter, quality of the micro programs that have been written as unit tests.
- [**5.5 out of 10**] Correctness. This will be judged by having your emulator execute the different test programs in increasing complexity. We will stop as soon as the emulator does not execute a program correctly. This can be seen as a "level" that has been attained. The attained level determines the score for this component:
  - Level 0: `basic.bin`: 0.75 points
  - Level 1: `hello.bin`: 1.5 points
  - Level 2: `hellof.bin`: 2.0 points
  - Level 3: `comp.bin`: 3.0 points

  - Level 4: `matvec.bin` and `matvecu.bin`: 3.5 points
  - Level 5: `brainfuck.bin`: 4.5 points
  - Level 6: `mandel.bin`: 5.5 points

  Note: we do not consider scores between levels. A score corresponding to a level can only be attained if the program for that level can be executed correctly.
- [**1.5 out of 10**] Robustness of the implemented instructions. This will be tested using our own, private, unit test suite.
- [**1 out of 10**] Implementation of floating-point instructions, such that the emulator can execute a "hard-float" version of `mandel.bin`.

Note that in order to receive a sufficient grade ($\geq$ 5.5), at least the level `hellof.bin` must be attained, together with perfect scores for quality and robustness.

The programs that are submitted will be graded automatically. All source code that is submitted will be subjected to plagiarism checks.