ECE463/563 – Microprocessor Architecture
Project #1

Due date: February 28, 2019 – March 9, 2019 (see below)

Objective

The goal of this project is to design and implement a C/C++ cycle-accurate simulator of a 5-stage MIPS-like pipelined processor. *Students taking the course at the undergraduate level (ECE463) will implement only an integer pipeline simulator. Students taking the course at the graduate level (ECE563) will also implement a floating-point pipeline simulator.*

Due date

Simulator of integer pipeline: February 28, 2019
Simulator of floating-point pipeline: March 9, 2019

Code organization

The `project1_code.tar.gz` archive contains C and C++ code templates for the simulator, as well as test cases to validate its operation. *ECE563 students are required to use C++ code templates*.

In order to extract the code in a Linux environment, you can invoke the following command:

```
tar -xzvf project1_code.tar.gz
```

This will create a `project1_code` folder containing two subfolders: `c` and `c++`, the former containing the C templates and the latter containing the C++ templates. The content of these two subfolders is very similar; the `c++` subfolder, however, contains also the templates for the simulator of the floating-point pipeline.

The `c` and `c++` folders have the following content:

- `sim_pipe.h/sim_pipe.cc` (or `sim_pipe.c`): code templates for the integer pipeline simulator. These are the only files that you need to modify to implement the simulator.
- `sim_pipe_fp.h /sim_pipe_fp.cc`: code templates for the floating-point pipeline simulator (these files are present only in the `c++` folder).
- `testcases`: test cases to validate the operation of the integer and floating-point pipeline simulators. This folder contains twelve test cases (six for each simulator). For each of them, you will find two files: `testcaseN.cc` and `testcaseN.out`. The former contains the test case implementation, and the latter the expected output of the test case. You should not modify any of the test case files.
- `Makefile`: `Makefile` to be used to compile the code. The use of this `Makefile` will cause an object file (`.o`) to be created for each C or C++ file that is part of the project. If the compilation succeeds, the binaries corresponding to the test cases will be generated in the `bin` folder. You don't need to modify the `Makefile`

unless you are working on the floating-point pipeline simulator or you are using a library that is not included by default.

- `asm`: assembly files used by the test cases.
- `bin`: once you compile the code, the test cases binaries will be saved into this folder.

Important

The `sim_pipe.h` and `sim_pipe_fp.h` header files are commented and contain details on the functions/methods that you need to implement. Be sure to read the comments in these header files carefully before you start coding.

Assumptions & Requirements

1. The integer simulator operates on 32-bit integer numbers stored in data memory; the floating-point simulator operates on 32-bit integer and 32-bit floating-point numbers stored in data memory.
2. The integer simulator has 32 integer registers (R0-R31). In addition to these, the floating-point simulator uses 32 single-precision floating-point registers (F0-F31)
3. The instruction and data memories are separated.
4. The instruction memory returns the instruction fetched within the clock cycle, while the data memory has a configurable latency. If the data memory has a latency of *L* clock cycles, the MEM stage will take *L*+1 clock cycles.
5. All the stages of the integer pipeline except the MEM stage take 1 clock cycle (each) to complete.
6. All the stages of the floating-point pipeline except for the EX and MEM stages take 1 clock cycle (each) to complete.
7. The integer simulator needs to support the following instructions (which are listed in the `sim_pipe.h` header file).
   - `LW` – *Load word*: Loads a 32-bit integer into a register from the specified address.
   - `SW` – *Store word*: Stores a 32-bit integer into data memory at a specified address.
   - `ADD/SUB/XOR` – *Add/Sub/Xor*: Computes the addition/subtraction/exclusive XOR of the values of two integer registers and stores the result into an integer register.
   - `ADDI/SUBI` – *Add/Sub Immediate*: Computes the addition/subtraction of the value of an integer register and a sign-extended immediate value and stores the result into an integer register.
   - `BEQZ, BNEZ, BLTZ, BGTZ, BLEZ, BGEZ` – *Branch if the value of the register is* $=, \neq, <, >, \leq, \geq$ zero.
   - `JUMP` – *Unconditional branch*.
   - `EOP` – *End of program*: Special instruction indicating the end of the program.
   - `NOP` – Special instruction *inserted automatically by the processor* to implement stalls.

   In addition to these, the floating-point simulator supports the following instructions:

   - `LWS` – *Load word*: Loads a 32-bit floating-point into a register from the specified address.
   - `SWS` – *Store word*: Stores a 32-bit floating-point value into data memory at a specified address.

- `ADDS/SUBS/MULTS/DIVS` – *Add/Sub/Mult/Div*: Computes the addition/subtraction/ /multiplication/division of the values of two floating-point registers and stores the result into a floating-point register.

8. *Data hazards*: The considered pipelined processors don't support forwarding.
9. *Control hazards*: The logic to compute the target of conditional and unconditional branches is in the EX stage; control hazards should require 2 stalls to be resolved. In the presence of a control hazard, the processor keeps fetching the same instruction until the hazard is resolved.
10. *Pipeline registers*: If the value of a special-purpose register written by a stage is irrelevant to the instruction currently processed by that stage, the value of this register should be set to `UNDEFINED`. For example, when decoding an instruction, the ID stage writes special-purpose registers `A`, `B` and `IMM`, and it propagates the value of special-purpose registers `IR` and `NPC`. Instruction `ADD R1 R2 R3`, however, does not use register `IMM`. As a consequence, when processing this instruction, the simulator should set the value of register `ID_EX_IMM` to `UNDEFINED`. Similarly, when processing a `NOP`, the simulator should set the values of registers `ID_EX_A`, `ID_EX_B` and `ID_EX_IMM` to `UNDEFINED`. Note that this is not necessarily the way the hardware would work (the pipeline registers could also retain the last value written to them). But, since the `print_registers` function prints only the registers that have a value ≠ `UNDEFINED`, this assumption will make debugging a bit easier.
11. *Execution units in floating-point pipeline simulator*: The floating-point pipeline simulator should be configurable in the number of floating-point units and their latency. The function `init_exec_unit` can be invoked at the beginning to configure the execution units used. Execution units can be of four kinds: `INTEGER` unit, floating-point `ADDER`, floating-point `MULTIPLIER` and floating-point `DIVIDER` (see `exe_unit_t` data type). For simplicity, *assume that all execution units are unpipelined*. Note that, if an execution unit has a latency of *L* clock cycles, an instruction using that unit will stay in the EX stage for *L+1* clock cycles.
12. *CPI computation*: The `EOP` instruction should be excluded from the computation of the CPI.

Suggestion

When implementing the pipeline simulator, proceed and test your code incrementally. The test cases have been designed to test incrementally different aspects of the code. In particular, for the integer simulator we recommend following these steps:

1. Implement the pipeline processing without hazards detection and handling => `testcase1` uses an assembly code that is free from hazards. In other words, `testcase1` is designed to simply test the operation of the data path and the pipeline registers.
2. Introduce data hazards handling in your simulator => `testcase2` and `testcase3` are meant to verify the correctness of the data hazards handling logic.
3. Introduce control hazards handling in your simulator => `testcase4` is meant to verify the correctness of the control hazards handling logic.
4. Introduce structural hazards handling => `testcase5` is meant to verify the correctness of the structural hazards handling logic.
5. Finally, `testcase6` allows you to test the correct operation of your simulator on a full program (it uses the same assembly file provided for project #0).

Testing

As mentioned above, the compilation process generates a separate binary for each test case in the `testcases` folder. To execute `testcaseX`, you can go in the `c/c++` folder and invoke:

```
./bin/testcaseX
```

To check if your output is correct, you can compare it with file `testcaseX.out` in the `testcases` folder. On Linux, you can use the `diff` utility to do so.

For example, you can invoke

```
./bin/testcaseX > my_output
```

```
diff my_output testcases/testcaseX.out
```

The first command will run the test case and save its output into `my_output`. The second command will compare your output with the reference output line-by-line.

Grading guide

ECE463 students

- 10% report
- If you have written a substantial amount of code but your code does not compile or does not execute (that is, none of the test cases execute), you will receive 40% credit for your code,
- Otherwise:
    - 45% for correct execution of test case #1;
    - 10% for correct execution of test case #2;
    - 5% for correct execution of test case #3;
    - 10% for correct execution of test case #4;
    - 10% for correct execution of test case #5;
    - 10% for correct execution of test case #6;

ECE563 students

- 10% report
- 60% score from integer pipeline code (using the grading guidelines above)
- 30% score from floating-point pipeline code, as follows:
    - 5% if you have written a substantial amount of code but your code does not compile or does not execute (that is, none of the test cases execute)
    - otherwise: 10% for test case #0 and 4% for each other test case

Resources

C/C++ programming

If you need a C/C++ refresher, there are many resources online that you can consider. The following C++ tutorial covers fundamental concepts and is fairly compact:

http://www.cplusplus.com/doc/tutorial/

For this project, you don't need to use and know advanced C++ features. In addition to concepts related to C programming, you need to look into the following C++ features: (i) classes (you don't need to use inheritance or polymorphism in your code), (ii) input/output with files using the C++ standard library.

In addition, while not required, you might find classes of the Standard C++ Library useful. You can have a look here:
http://www.cplusplus.com/reference/

GNU make

You will use GNU make to compile and link your code for all the class projects. If you have never used GNU make before, you can find plenty of resources online. All the required Makefiles are provided.

A complete manual is available here:

https://www.gnu.org/software/make/manual/make.html

However, for this course you need to make only basic usage of GNU make, and so you can simply have a look at brief tutorials such as:

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor

Debugging

Recommendation: use `gdb` and `valgrind` for debugging.

Submission instructions

Report: Your report should be no longer than 4 pages (11 pt. Times New Roman font), including figures. The report should be typed. It should include the following information:

- Description of data structures used to model instruction memory, general purpose registers and pipeline registers;
- Description of your approach to handle data, control and structural hazards;
- Brief explanation of what works and what not in your implementation;
- If you modified the `Makefile`, indicate it in the report.

Save your report in *pdf* format, with file name `project1_report.pdf`. Include the report in the `project1_code` folder.

If you are taking the ECE563 version of the class, along with the floating-point simulator you should submit a `project1_report_fp.pdf`. This report should be no longer than 2 pages (11 pt. Times New Roman font), and should include only the following information:

- Description of your approach to handle the additional hazards (i.e., the ones not present in an integer pipeline)
- Brief explanation of what works and what not in your implementation;
- If you modified the `Makefile`, indicate it in the report.

1. **Test cases:** You should **not** modify any of the test cases. All the functionality should be included in the `sim_pipe.h`, `sim_pipe.c/cc`, `sim_pipe_fp.h`, and `sim_pipe_fp.cc` files.

2. **Code:** Independently of the development environment and operating system you used to develop your code, your code should compile and run on the `grendel.ece.ncsu.edu` Linux machine, and it should compile using the provided `Makefile` (you can modify the `Makefile` if you are using libraries which are not included or if you have implemented the floating-point pipeline simulator).
3. You should invoke "`make clean`" before submitting your code. That is, your submission should not contain any object or binary files.
4. Remove the folder containing the templates that you did not use. In other words, if you used the C templates, delete the `c++` folder and its content; if you used the C++ templates, delete the `c` folder and its content. Your `project1_code` folder should contain the report(s) and one of the `c` or `c++` folders.
5. For each submission: go to the parent folder of the `project1_code` folder. Compress the whole `project1_code` directory into a *tar.gz* file.

```
tar -zcvf project1_code.tar.gz project1_code
```

6. Submit your project through Moodle on the due date (no need to print the report and take it to class).