ECE463/563 – Microprocessor Architecture
Project #2

Due date: April 7, 2020

Objective

The goal of this project is to design and implement a C/C++ cycle-accurate simulator of a dynamically scheduled processor implementing the Tomasulo algorithm with Reorder Buffer.

Students taking the course at the undergraduate level (ECE463) do not need to support store instructions (SW and SWS) in their simulator, while students taking the course at the graduate level (ECE563) will need to support store instructions (and, as a consequence, RAW hazards on memory) as well.

Code organization

The `project2_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 project2_code.tar.gz
```

This will create a `project2_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` and `c++` folders have the following content:

- `sim_ooo.h/sim_ooo.cc` (or `sim_ooo.c`): code templates for the dynamically scheduled processor's simulator. These are the only files that you need to modify to implement the simulator.
- `Test cases`: test cases to validate the operation of the dynamically scheduled processor's simulator. This folder contains five test cases. For each of them, you will find two files: `Test caseN.cc` and `Test caseN.out`. The former contains the test case implementation, and the latter the expected output of the test case. <u>You should not modify any of the test case files</u>. *We recommend writing your own test cases to verify the functionality that is not covered by the test cases provided.*
- `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_ooo.h` header file is commented and contains details on the functions/methods that you need to implement. Be sure to read the comments in this header file carefully before you start coding.

Assumptions & Requirements

1. *Data types and registers*: The simulator operates on 32-bit integer numbers and 32-bit floating-point numbers stored in data memory. The simulator has 32 integer registers (`R0-R31`) and 32 single-precision floating-point registers (`F0-F31`).
2. *Memories*: The instruction and data memories are separated. The instruction memory returns the instruction fetched within the clock cycle, while the data memory has a configurable latency.
3. *Execution units and memory*: The number of execution units and their latency must be configurable. The function `init_exec_unit` can be invoked at the beginning to configure the execution units used. Execution units can be of five kinds: `INTEGER` unit, floating-point `ADDER`, `MULTIPLIER`, `DIVIDER` and `MEMORY` (see `exe_unit_t` data type). The integer units are used for integer additions (and branches), subtractions and logic operations; the floating-point adders are for floating-point additions and subtractions; the multipliers are for integer and floating-point multiplications; and the dividers for integer and floating-point divisions. While the simulator can have multiple integer units, adders, multipliers and dividers, it will always have a single data memory unit (used by load and store instructions). While the processor can have multiple execution units of the same type with different latencies, in our test cases all the units of the same type will have the same latency. For simplicity, *assume that all execution units are unpipelined*. You can assume that the latency of the execution stage corresponds to that of the execution unit it uses.
4. *Reservation stations and load buffers:* Besides load buffers, the processor should have three kinds of reservation stations (see `res_station_t` data type). Integer reservation stations (`INTEGER_RS`) feed the integer units, add reservation stations (`ADD_RS`) feed the floating-point adders, and multiplier reservation stations (`MULT_RS`) are shared by multipliers and dividers. Load buffers (`LOAD_B`) are used by memory instructions.
5. *Initialization:* the size of the data memory, the size of the reorder buffer, the number of reservation stations and load buffers, and the issue width can be configured when instantiating the simulator.

```
/* Instantiates the simulator
  Note: registers must be initialized to UNDEFINED value, and data memory to all 0xFF
  values
*/
sim_ooo(unsigned mem_size,           // size of data memory (in byte)
     unsigned rob_size,              // number of ROB entries
     unsigned num_int_res_stations,  // number of integer reservation stations
     unsigned num_add_res_stations,  // number of ADD reservation stations
     unsigned num_mul_res_stations,  // number of MULT/DIV reservation stations
     unsigned num_load_buffers,      // number of LOAD buffers
     unsigned issue_width=1          // issue width
);
```

6. *Reservation stations, ROB, instruction window*: the code template already contains data structures to model reservation stations, ROB and instruction window. You can extend these data structures if you need, but you should not modify existing fields.
7. *Instructions supported*: the processor simulator needs to support the following instructions (which are listed in the `sim_ooo.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/OR/AND/MULT/DIV` – *Add/Sub/Xor/Or/And/Mult/Div*: Computes the addition/subtraction/exclusive XOR/OR/AND/multiplication/division of the values of two integer registers and stores the result into an integer register.
- `ADDI/SUBI/XORI/ORI/ANDI` – *Add/Sub/Xor/Or/And Immediate*: Computes the addition/subtraction/exclusive XOR/OR/AND 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*.
- `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.
- `EOP` – *End of program*: Special instruction indicating the end of the program.

As mentioned above, ECE463 do not need to support the `SW` and `SWS` instructions in their simulator.

8. *Load/store instructions and RAW hazards on memory:* Load and store instructions use the memory unit in different stages. Load instructions load data from memory in the execution stage and send their result to the reservation stations and ROB through the CDB in the write result stage. Store instructions write data to memory at the commit stage and use the execution stage only for the computation of the effective memory address. Load and store instructions from/to different memory addresses do not conflict. However, *your implementation should handle RAW hazards occurring when a load and a store instruction access the same memory address*. Your implementation should assume that, in this situation, the value that will be written to memory by the pending store is bypassed to the load. To meet these requirements, your implementation should process memory instructions as follows:
   - When a load or store instruction is issued, the address field of the corresponding reservation station is filled with the value of the immediate field of the instruction. The address field is updated with the effective memory address when the instruction is moved to the execution stage.
   - When multiple instructions are eligible to move from the issue to the execution stage, they are considered *in program order*. In addition, *the effective memory address is computed in program order*. Load instructions are not allowed to execute if preceded by potentially conflicting store instructions.
   - Store instructions are allowed in the execution stage only if the values of both their input registers are available. Note that, while a less stringent timing is possible, this facilitates the handling of data bypassing between store and dependent load instructions.
   - Store instructions update the destination field of the ROB with the effective memory address when they enter the execution stage, and they update the value field of the ROB with the result that they will later write to memory in the write-result stage.
   - In the presence of a pending store to the same address, data bypassing from the store to the load instruction is done when the load enters the execution stage. In this stage, the load will save the value bypassed in the `value2` field of its reservation station. In the write-result stage, the load will write this value in the reorder buffer.
   - Bypassing happens through the ROB (for the timing, see *timing constraint* on data dependencies below).

- Store instructions spend only one clock cycle in execution stage, and a number of clock cycles equal to the memory latency in the commit stage. In the presence of store-to-load data bypassing, load instructions spend only one clock cycle in the execution stage (since in this case they don't use the memory unit).

Note that data bypassing does not modify the timing of store instructions, but it only speeds up the processing of dependent load instructions.

9. *Other timing constraints:*
   - If an instruction *I* uses a ROB entry, such ROB entry becomes available and can be used in the clock cycle *after I* commits.
   - In the presence of a structural hazard on an execution unit, the execution unit is available to the second instruction in the clock cycle after the first instruction has finished using it. If an instruction *I* uses an execution unit in the execution stage, such execution unit is freed when *I* writes the result and is available to a different instruction starting from the next clock cycle.
   - If an instruction *I* uses a reservation station or a load buffer, such reservation station or load buffer is freed when *I* writes the result, and is available to a different instruction starting from the next clock cycle.
   - If an instruction *J* depends on instruction *I*, the data written by instruction *I* will be available to *J* in the clock cycle when *I* writes the result (say *t*), and (in the absence of data hazard), instruction *J* will be able to execute in the following clock cycle (say *t*+1).
   - In the write result stage branch instructions write their target address in the result field of the corresponding ROB entry. If the branch is mispredicted, the target instruction of the branch is fetched in the clock cycle after the branch instruction *commits*. While a less strict timing is possible, this simplifies the implementation.
   - *CPI computation*: The `EOP` instruction should be excluded from the computation of the CPI.
   - *Memory address calculation*: You can choose between two implementations:
     a. Simplified implementation: Rather than explicitly computing the memory address in the execution stage, assume that address computation is performed before entering that stage. In addition, assume that the address information is used to determine whether the execution stage can be entered by the instruction.
        The outputs in the `testcases` folders follow this implementation.
     b. Accurate implementation: Address calculation is performed in the execution stage. For memory instructions, the execution stage will go through the following steps:
        - address computation by an integer adder (1 clock cycle)
        - access to data memory (number of clock cycles depending on the memory latency).
        At every clock cycle, the address computation unit can compute the memory address of one instruction. The address calculation of one instruction can be done in parallel to the memory access of a previous instruction. Load and store instructions must enter the execution stage in order (since the memory addresses must be computed in order). Other instructions can enter the execution stage out-of-order. The outputs in the `testcases/accurate_address_calculation` folder follow this implementation.

Format of the output

The code template includes functions to print out the content of the data memory, the status of the registers (and, if their value is pending, the ROB entry that will update the register), the content of the reservation stations and load buffers, the content of the reorder-buffer, the status of the pending instructions (that is, the ones still in the

ROB), and the instruction execution history (which shows the cycle-by-cycle execution of all instructions fetched during operation). These functions do not need to be modified, and they are invoked from the test case files.

```
//prints the values of the registers
void print_registers();
```

Example output:

```
GENERAL PURPOSE REGISTERS
Register                 Value  ROB
      R1          10/0x0000000a    -
      R2          20/0x00000014    -
      R3          10/0x0000000a    -
      F0                      -    0
      F1          10/0x41200000    -
      F2                      -    1
      F3                      -    2
      F4          40/0x42200000    -
      F5          50/0x42480000    -
      F6          60/0x42700000    -
      F7          70/0x428c0000    -
      F8          80/0x42a00000    -
      F9          90/0x42b40000    -
      F10        100/0x42c80000     -
```

```
// prints the content of the ROB
void print_rob();
```

Example output:

```
REORDER BUFFER
Entry  Busy  Ready          PC     State  Dest       Value
    1    no     no           -         -     -           -
    2   yes     no  0x00000004       EXE    F2           -
    3   yes     no  0x00000008     ISSUE    F3           -
    4   yes     no  0x0000000c     ISSUE    F4           -
    5    no     no           -         -     -           -
    6    no     no           -         -     -           -
```

```
//prints the content of the reservation stations
void print_reservation_stations();
```

Example output:

```
RESERVATION STATIONS
   Name  Busy          PC          Vj          Vk    Qj    Qk  Dest     Address
   Int1   yes  0x00000010  0x0000000a           -     -     -     4           -
  Load1    no           -           -           -     -     -     -           -
  Load2    no           -           -           -     -     -     -           -
   Add1   yes  0x00000008  0x41200000  0x41f00000     -     -     2           -
   Add2    no           -           -           -     -     -     -           -
  Mult1   yes  0x0000000c  0x41200000  0x42480000     -     -     3           -
  Mult2    no           -           -           -     -     -     -           -
```

```
//print the content of the instruction window
void print_pending_instructions();
```

Example output:

```
PENDING INSTRUCTIONS STATUS
        PC  Issue    Exe     WR Commit
         -      -      -      -      -
0x00000004      1      3      4      -
0x00000008      2      -      -      -
0x0000000c      3      4      -      -
0x00000010      4      -      -      -
         -      -      -      -      -
```

```
//print log
void print_instruction history();
```

Example output:

```
EXECUTION LOG
        PC  Issue    Exe     WR Commit
0x00000000      0      1      2      3
0x00000004      0      3      4      5
0x00000008      0      5      7      8
0x0000000c      0      3     13     14
0x00000010      1      2      4     15
0x00000014      1     14      -      -
0x00000018      4      5      -      -
0x0000001c      8      9     11      -
0x00000020     12      -      -      -
0x00000020     16     17     19     20
0x00000024     16     20     22     23
```

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

- Report = 10 points
- Code = max 50 points

    - 50 points if the code compiles, runs, and you can run successfully at least one test case
    - 0-50 points if the code does not compile/run, but you have written substantial amount of code. The exact score will depend on the status of your code (amount of code written/functionality implemented).
- Test cases = max 40 points
    - Test cases 1-5 = 6 points each
        - 0.5 points if the number of clock cycles and instructions executed match
        - 0.5 point if the final content of registers and memory match
        - 3 points if the execution log fully matches (according to `diff`)
        - 2 points if the cycle-by-cycle execution fully matches (according to `diff`)
    - Test case 6 = 10 points
        - 0.5 points if the number of clock cycles and instructions executed match
        - 1 point if the final content of registers and memory match
        - 5 points if the execution log fully matches (according to `diff`)
        - 3.5 points if the cycle-by-cycle execution fully matches (according to `diff`)

ECE563 students

- Report = 10 points
- Code = max 40 points
    - 40 points if the code compiles, runs, and you can run successfully at least one test case
    - 0-40 points if the code does not compile/run, but you have written substantial amount of code. The exact score will depend on the status of your code (amount of code written/functionality implemented).
- Test cases = max 50 points (10 test cases, 5 points each)
    - Test cases 1-8
        - 0.5 points if the number of clock cycles and instructions executed match
        - 0.5 points if the final content of registers and memory match
        - 2.5 points if the execution log fully matches (according to `diff`)
        - 1.5 points if the cycle-by-cycle execution fully matches (according to `diff`)
    - Test cases 9 & 10
        - 0.5 points if the number of clock cycles and instructions executed match
        - 0.5 points if the final content of registers and memory match
        - 4 points if the execution log fully matches (according to `diff`)

Self-grading

Please test your code on `grendel.ece.ncsu.edu`, fill the following table (according to the grading guide below, and include them in your report).

Test case	**Points**	**Comment (brief explanation of test case evaluation)**
Test case 1		
Test case 2		

Test case 3		
Test case 4		
Test case 5		
Test case 6		
Test case 7		
Test case 8		
Test case 9		
Test case 10		

Comment should be a brief explanation of the test case evaluation. Examples: "*the test case and reference output fully match*", "*everything but the cycle-by-cycle execution matches*", "*the final content of memory and register and the execution log fully match*", etc.

Submission instructions

1. **Report:** Your report should be no longer than 4 pages (11 pt. Times New Roman font), including figures. It should include the following information:
   - Brief description of the salient aspects of your implementation;
   - Brief explanation of what works and what not in your implementation;
   - Table above;
   - If you modified the `Makefile`, indicate it in the report.

   Save your report in *pdf* format, with file name `project2_report.pdf`. Include the report in the `project2_code` folder.

2. **Test cases:** You should **not** modify any of the test cases. All the functionality should be included in the `sim_ooo.h` and `sim_ooo.c/cc`, files. You can add header and C/C++ files, but you don't need to.

3. **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). Have a look at Piazza for information on how to access the `grendel` machine.

4. You should invoke "`make clean`" before submitting your code. That is, your submission should not contain any object or binary files.

5. 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 `project2_code` folder should contain the `project2_report.pdf` and one of the `c` or `c++` folders.

6. Go to the parent folder of the `project2_code` folder. Compress the whole `project2_code` directory into a *tar.gz* file.

```
tar -zcvf project2_code.tar.gz project2_code
```

7. Submit your project through Moodle (no need to print the report and take it to class).