Due date: Thursday October 25th 11:59:59pm EST

Getting started: To create your initial Lab 5 repository, please visit the repository creation page at https://6004.mit.edu/web/fall18/user/labs/lab5. Once your repository is created, you can clone it into your VM by running:

```
git clone git@github.mit.edu:6004-fall18/labs-lab5-{YourMITUsername}.git lab5
```

Turning in the lab: To turn in this lab, commit and push the changes you made to your git repository. After pushing, check the course website to verify that your submission passes all the tests. If you finish the lab in time but forget to push, you will incur the standard late submission penalties.

Check-off meeting: After turning in this lab, you are required to go to the lab for a check-off meeting within 6 days of the lab’s due date. See the course website for lab hours.

RISC-V Single-Cycle and Multi-Cycle Processors

In this lab you will implement RISC-V Single-Cycle and Multi-Cycle Processors in Bluespec. The instruction set architecture (ISA) of your RISC-V processors is RV32I, which is the base integer 32-bit variant. For Bluespec-related questions for this Lab, you may want to check out the Introductory Bluespec User Guide.

Note that you can finish the required part of the lab independently in any order. But we suggest reading Section 1 first which is important for testing your processors.

To pass the lab you must complete and PASS all of the exercises except the ones in the last two sections.

Coding guidelines: You are only allowed to change the following files: Decode.bsv, Multicycle.bsv, bubblesort.S and quicksort.S. Modifications to other files will be overwritten during didit grading. You should provide answers to the discussion questions in discussion.txt.

1 Processor Test Suite

Please read this section first. This section explains how the tests are run for your processor, which is important for debugging this lab.

The git repository for lab 5 contains three sets of tests, located under sw/:

- sw/microtests
- sw/fullasmtests
- sw/gcdtest

You can compile the tests into assembly code by running cd sw && make && cd ... This will compile each test code into two files in the sw/build/ folder:

- `<some_test>.vmh` is the assembly code encoded in 32-bit hexadecimal values, which will be loaded into your processor’s memory before execution.
- `<some_test>.dump` is the annotated assembly code for your reference.

1.1 Microtests

The microtests are numbered, and each microtest may use instructions tested in previous microtests. To make sure that microtests pass, the test script will check that the register file and the PC at the end match the expected state of the processor. The expected processor state (register file and PC) for each test can be found in the folder expectedmicrotests).
To stop your processor after it finishes executing the tests, we rely on the unimp instruction, which is a pseudoinstruction that represents an unimplemented instruction. When a processor encounters an unimp instruction, it should stop and dump its state. To see how it is done, you can take a look at lines 33–40 in Singlecycle.bsv. Each microtest ends with an unimp.

If you fail a microtest, you should take a look at the assembly code in the sw/build/microtests folder, the expected processor state in the expectedmicrotests folder, and your final state in the test_out folder. And try to understand what went wrong.

1.2 Fullasmtests

The fullasmtests are more complicated self-checking tests provided by the RISC-V designers. The fullasmtests don’t exit in the same way as microtests. Instead of hitting an unimp, the processor can also exit using memory-mapped stop. At the exit code, each fullasmtest will write some value at the memory address 0x40001000. If 0 is written, the test is successful. Otherwise, the test failed. You can read the <exit> code section of the annotated *.dump files in the sw/build/fullasmtests directory to understand how to do memory-mapped stop. Below is the <pass> <fail> and <exit> sections of add.dump.

```
000004e4 <fail>:
  4e4: 00c0006f jal x0,4f0 <exit>

000004e8 <pass>:
  4e8: 00000e13 addi x28,x0,0
  4ec: 0040006f jal x0,4f0 <exit>

000004f0 <exit>:
  4f0: 40001537 lui x10,0x40001
  4f4: 01c52023 sw x28,0(x10) # 40001000 <begin_signature+0x40000000>
```

If you fail one of the fullasmtests, first make sure you are passing all the microtests, then look at how your processor executes the instructions for the failed fullasmtest.

1.3 GCDtests

The gcdtests are the test code for your GCD calculation, which will be explained in Section 5. They use memory-mapped stop to halt the processors.

2 Decode

Lecture 11 slides have provided you with a starting point for your RISC-V processor. The slides had all the necessary code required for a Single-Cycle processor except for the decoder. We also provided you with the Single-Cycle processor to you in Singlecycle.bsv, but function definition inside decode is empty. The types for the decoder were provided, so all you have to do is fill in the decoder function, and you will have a working single-cycle processor design.

Exercise 1 (50 points—only passing microtests is 35 points): Fill in the decode function in Decode.bsv.

All the processor related-types are defined in ProcTypes.bsv. Please refer to slides 11–15 of Lecture 11 for what instructions to decode and how to decode them. Note that instruction AUIPC is required for running fullasmtests and we have implemented AUIPC in the decoder function for you.

Build your Single-Cycle processor by running make Singlecycle.
You can run a suite of tests on the processor using the `test.sh` by running `./test.sh` or `bash test.sh`. You should pass the microtests (option 5) and the fullasmtests (option 6).

Note: Ignore any warnings about `mem.vmh` emitted by the simulator. Also note that you can complete the subsequent sections of the lab without a functional decoder.

3 Multi-Cycle Processor

The Single-Cycle processor (`Singlecycle.bsv`) uses a magic memory module which responds in the same clock cycle, which is not realistic. If you want to create a more realistic processor, you will need to use a more realistic memory system that has a request/response interface. That is, to read from the memory, you use the request method and in a later cycle, the response method will be ready and can be read by firing another method.

Exercise 2 (35 points—only passing microtests is 20 points): Complete the `Multicycle.bsv` file which already instantiates a realistic memory module.

All the processor related-types are defined in `ProcTypes.bsv`. If your decoder from Section 2 does not work, you can use the `decode_golden` function packaged as an object file by the course staff instead.

Build your processor by running `make Multicycle`.

You can run a suite of tests on the processor using the `test.sh` by running `./test.sh` or `bash test.sh`. If you haven’t implemented the processor yet, tests will run forever. You can kill the tests with `<Ctrl-C>`. You should pass the microtests (option 5) and the fullasmtests (option 6).

Note: To support the unimp instruction used in microtests, remember to stop the processor by using the same code as the `Singlecycle.bsv`:

```
if(eInst.iType == Unsupported)
    begin
        $display("Reached unsupported instruction (0x%x)", inst);
        $display("Dumping the state of the processor");
        $display("pc = 0x%x", pc);
        rf.displayRFileInSimulation;
        $display("Quitting simulation.");
        $finish;
    end
```

You must complete Exercises 1 and 2. To get full credit, complete the exercises and discussion questions below.

4 Run Software on Multi-cycle processor

Now that you have a working processor, you can run any software supported by our processor’s ISA. Let’s run the bubblesort and quicksort you implemented in Lab 4.

Exercise 3 (5 points): Compile your bubblesort and quicksort from Lab 4 and run the programs on your Multi-cycle processor.

To compile your code, copy files `bubblesort.S` and `quicksort.S` into directory `my_software` and run `make lab4`.

To run your code on Multicycle processor, run `./run_lab4.sh`
5 Profiling GCD Software

For some reason, the software that 6.004 runs spends most of its time doing GCDs. As most of the time is spent doing GCDs, a TA wrote the GCD code in assembly. Here is a snippet of the GCD code. In the lab repo, you can find `sw/gcdtests/gcd_sw.S` which defines this gcd function and uses it in a test.

```assembly
gcd:  beqz a0, a0_is_zero
gcd_loop:  beqz a1, gcd
   bgeu a1, a0, a1_ge_a0
   mv t0, a1
   mv a1, a0
   mv a0, t0
   j gcd_loop
a1_ge_a0:  sub a1, a1, a0
   j gcd_loop
a0_is_zero:  mv a0, a1
gcd_end:  ret
```

We want to get an idea of how many cycles are spent calculating this GCD compared to the time spent running the entire program.

To profile the code, you will use the performance counter library in `PerfCounter.bsv`. The library provides a `PerfCounter` interface with three methods that provide performance profiling capabilities as below.

```assembly
interface PerfCounter;
    method Action startRoutine();
    method Action stopRoutine();
    method Bit#(64) peek();
endinterface
```

- `startRoutine` should be called when the program enters the routine, and should start the clock counter.
- `stopRoutine` should be called when the program leaves the routine. To support recursive routines, `stopRoutine` will stop the clock counter only when we fully finish executing all the steps of the recursions.
- `peek` is unguarded and returns the current clock counter value.

You should include `import PerfCounter::*;` in the beginning of `Multicycle.bsv` to use the PerfCounter module. You should then instantiate two `PerfCounters` in `Multicycle.bsv` to count both how many cycles are spent in the GCD routine as well as how many cycles are spent executing the entire program.

The counter for the entire program should start counting when fetching the first instruction at address 0x0. It should stop counting after executing the last instruction of the program (the instruction before `unimp`).

The counter for the GCD routine should start counting when fetching the first instruction of the `gcd` routine. It should stop counting after executing the last instruction of the `gcd_end` routine. To get addresses of these instructions in the test program, look at the `sw/build/gcdtests/gcd_sw.dump` file.

You should `peek` and display the counters after executing the last instruction in the `exit` routine.

Modify `Multicycle.bsv` to use the two counters to profile `gcd_sw.S`.

To run just the `gcd_sw.S`, run `ln -sf sw/build/gcdtests/gcd_sw.vmh mem.vmh & & ./Multicycle`

Discussion Question 1 (5 points): In the test `sw/gcdtests/gcd_sw.S`, how many cycles does the multi-cycle processor spend in the gcd routine? How many cycles does the processor take in total to run the program?
response. Now suppose memory has a latency of 50 cycles, we want to determine how slower your processor will be?

Build your processor with 50-cycle memory latency by running `make MulticycleSlowMem`.
To run just the `gcd_sw.S` run

```
ln -sf sw/build/gcdtests/gcd_sw.vmh mem.vmh && ./MulticycleSlowMem
```

**Discussion Question 2 (5 points):** Now with the slower memory, in the test `sw/gcdtests/gcd_sw.S`, how many cycles does the multicycle processor spend in the GCD routine? How many cycles does the processor take in total to run the program?