Lab 6: RISC-V Single-Cycle and Multi-Cycle Processors

Due date: Thursday April 4th 11:59:59pm EST

Getting started: To create your initial Lab 6 repository, please visit the repository creation page at https://6004.mit.edu/web/spring19/user/labs/lab6. Once your repository is created, you can clone it into your VM by running:

```
git clone git@github.mit.edu:6004-spring19/labs-lab6-{YourMITUsername}.git lab6
```

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 by April 12th. 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.

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.

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 6 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. Try to determine what went wrong by examining the differences in state.

### 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 processor.

### 2 Decode

The lecture 13 slides 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. You can find the code from lecture in Singlecycle.bsv, but the function definition inside decode is empty. The types for the decoder are provided so all you have to do is fill in the decoder function. When this is done you will have a working single-cycle processor design.

**Exercise 1 (50 points):** Fill in the decode function in Decode.bsv.

All the processor related-types are defined in ProcTypes.bsv. Please refer to slides 11–20 of Lecture 13 for a list of the instructions to decode and how to decode them.

To get you started, we have implemented the AUIPC instruction in the decoder function for you. AUIPC stands for Add Upper Immediate Program Counter. The RISC-V ISA syntax is AUIPC rd, offset and the functional description is reg[rd] <- pc + offset. It’s similar to LUI because it takes a 20-bit immediate that is loaded into the upper bits. However, it then adds in the current value of the program counter (the address of the AUIPC instruction itself).

The line of code provided to you looks like this:
opAuipc: dInst = DecodedInst { dst: dstValid, src1: dSrc, src2: dSrc,
    imm: immU, brFunc: Dbr, aluFunc: Dalu, iType: AUIPC };

AUIPC has its own dedicated instruction type (iType). We don’t care about the sources (src1 and src2),
branch function (brFunc), or the ALU function (aluFunc); therefore, we use the default values for each. We
only need to provide the destination register and the immediate (which is the 20-bit immU version for this
instruction). The destination field of a DecodedInst is different from the source fields. Instead of just a
register index, it is a structure defined by:

```plaintext
typedef struct {
    Bool valid;
    RIndx index;
} RDst deriving (Bits, Eq);
```

A struct defines a new type with multiple fields inside it. Structs are useful because they allow a variable
to hold multiple types of information in a structured way, similar to objects in object oriented programming.
The definition of RDst shown above uses struct to create a new type. We can then create a new variable
using this type like this:

```plaintext
// Introduction of a RDst:
// Note the sneaky "," here versus the ";" in the typedef 4 lines before
RDst example1 = RDst{ valid: False, index: dst };
```

In this example, example1 is initialized with the valid field set to False and the index field set to the
value of variable dst. You can then access the values stored in these fields using example1.valid and
example1.index.

Build your Single-Cycle processor by running `make Singlecycle`.

You can run a suite of tests on the processor using the `test.sh` script by running `./test.sh` or `bash
test.sh`. You should pass the microtests (option 5) and the fullasmtests (option 6).

Note: You can ignore any warnings about `mem.vmh` emitted by the simulator.

## 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 will be ready and can be read by firing another method.

**Exercise 2 (40 points):** Complete the Multicycle.bsv file which already instantiates a realistic memory
module.

All the processor related-types are defined in `ProcTypes.bsv`.

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 we did in `Singlecycle.bsv`:

```plaintext
if(eInst.iType == Unsupported) begin
```
You must complete Exercises 1 and 2 to pass this lab. 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 quicksort you implemented in Lab 2.

Exercise 3 (10 points): Compile your quicksort from Lab 2 and run the program on your Multi-cycle processor.

To compile your code, copy quicksort.S into the my_software directory and run make lab2.

To run your code on your Multi-cycle processor, run ./run_lab2.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.

```
gcd:        beqz a0, a0_is_zero
gcd_loop:   beqz a1, gcd_end
            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.

```
interface PerfCounter;
    method Action startRoutine();
    method Action stopRoutine();
    method Bit#(64) peek();
endinterface
```
• **startRoutine** should be called when the program enters the routine, and will 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. You can instantiate a new `PerfCounter` using something like: `let perfCount <- mkPerfCounter;`.

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:** 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?

The memory the Multi-cycle processor used so far has a latency of 5 cycles between each request and response. Suppose we are considering switching to a different type of memory with a latency of 50 cycles. Let’s see what impact this will have on our software performance.

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:** Now with the slower memory, 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? Would you say that memory latency is an important factor to consider in this processor design? Given that there are no loads or stores in the GCD code above, is this result surprising?
6 Appendix

6.1 Additional materials on Bluespec Constructs

6.1.1 FShow

Deriving FShow when defining an Enum or a Struct type asks the compiler to generate an implementation of the fshow() function to pretty-print values of the newly-defined type. FShow is derived for all the Enum and Struct types in the lab to help with debugging. You can use fshow() to print the DecodedInst from decode in an easy-to-read format.

```
DecodedInst dInst = decode(inst);
$display("[PC = 0x%x] ", pc, fshow(dInst));
```

The above code produces a message like the one below.

```
[PC = 0x00000000] DecodedInst { iType: OPIMM, aluFunc: Add, brFunc: Lt, rd: tagged Valid 'h01, rs1: 'h00, rs2: 'h0a, imm: 'h00000001 }
```

Note that fshow() does not need a format character in the format string of the display statement, it is just concatenated to the end of the format string. If you want more complicated formatting, you can add another format string after it like below.

```
DecodedInst dInst = decode(inst);
$display("[PC = 0x%x] [iType = " , pc, fshow(dInst.iType), "] [imm = 0x%zx]", dInst.imm);
// Displays the string "[PC = 0x00000000] [iType = OPIMM] [imm = 0x00000001]"
```