RISC-V Single-Cycle and Multi-Cycle Processors

You have been promoted to senior engineer at GATE Co. You are in charge of improving the old microprocessors used in the elevators. To get ready for your new job, you may want to check out the appendix in Section 4.

The old processors in the elevator are so old, they were designed by drawing gates and connecting them with wires. In order to modernize the processor design methodology and to make it easier to modify, you have decided to implement the new processor in Bluespec System Verilog, and you have chosen the RISC-V ISA.

1 Decode

While looking for a starting point for the new processor, you came across some slides online for MIT’s 6.084 course. The slides had all the necessary code required for a single cycle processor except for the decoder. Fortunately 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.

Luckily you also found the git repository for lab 5 which contains different sets of tests located in sw/:

- sw/microtests
- sw/fullasmtests

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 at the end looks like what it should look like (the expected state of the processor (register file and pc) for each test can be found in the folder expectedmicrotests).

But how do we make the processor stop? The SingleCycle processor (and every processor you will write in this lab), should stop when they encounter an invalid (or unimplemented) instruction, and dump the state of the processor (you can take a look at SingleCycle.bsv to see how it is done). Every microtest ends with an unimp instruction (unimp is a pseudoinstruction that describe some unimplemented instruction), that will make the processor stop and dump its state.

If you fail a micro test, you should take a look at the source code, the expected state (in the expectedmicrotests folder), and your final state (in the test_out folder) and try to understand what went wrong.

The fullasmtests are more complicated self-checking tests. If you fail one of these tests, first make sure you are passing all the microtests, then look at how you are decoding the instruction mentioned in the name of the test.

Exercise 1 (50 Points - only passing microtests is 35 points): Fill in the decoder function in Decoder.bsv. Note that you will have to implement the AUIPC instruction in order to pass all the tests. The encoding for this instruction and its behavior can be found on the last two pages of the RISC-V ISA Reference from the course website (updated when this lab was posted). Test your decoder by building the single cycle processor with make SingleCycle. You can run a suite of tests on the processor using the test.sh script (run this script by typing ./test.sh or bash test.sh in the same directory as test.sh). You should pass the microtests and the full asm tests.

Warning: If your repository has a test_decoder.sh script, you should not use it. This script was used when developing the lab to produce the expected results found in the expectedmicrotests folder. If you used it,
use the following commands to revert expectedmicrotests back to the original state:
git checkout 9bcfa4 expectedmicrotests/
git commit -m "fixed expectedmicrotests"
git push

Note: You should ignore any warnings about mem.vmh emitted by the simulator. Any errors signal that there is something wrong with the test setup.

2 Princeton Multicycle

By filling in the decode function, you made the single-cycle processor work, but since that processor uses magic memory, it is not realistic. If you want a real 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 then in a later cycle, the response method will be ready, and firing that method will give you the read data.

Exercise 2 (20 Points - only passing microtests is 15 Points): Complete the MulticyclePrinceton.bsv file which already instantiates a realistic memories. Build your processor with make PrincetonMultiCycle. You can run a suite of tests on the processor using test.sh (you should select the multicycle option, when asked). You should pass the microtests and the full asm tests.

Note To be able to run the microtests, remember to stop on unsupported instruction, using the same code than the SingleCycle does:

```
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
```

3 Princeton Multicycle with GCD accelerator

... six months later ...

3.1 Profiling Software

The software engineers at GATE Co. have been using your multicycle processor for months now, but recently you learned that the code they write spends most of its time doing GCDs

As most of the time is spent doing GCD, performance engineers wrote the GCD code in asm. Here is a snippet you were given where the gcd is defined. 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
```

1It is frankly quite a mystery why would an elevator ever need to compute GCDs, but the software engineers probably know what they are doing.
RISC-V Single-Cycle and Multi-Cycle Processors

6.084 Computation Structures – Spring 2018

You secretly think that you could do the GCD in hardware instead of in software, but before investing the time in doing so, let’s profile the existing code. We want to get an idea of how many cycles are spent doing this gcd compared to how much time is spent in the entire program.

To profile the code, you are going to add some clock cycle counters to your design. These counters are registers of some Bit#(n) type that are incremented autonomously in their own rule. The first counter should always increment to tell you how long the current program has been running. The second counter should start incrementing when the first instruction in gcd is executed, and it should stop incrementing when the ret is executed. You should display the counters when you execute the first instruction in the exit function. To get the address of the instructions in the test program, look at the gcd_sw.dump file.

Discussion Question 1 (5 Points): In the test sw/gcdtests/gcd_sw.S, how many cycles does the multicycle processor spend in the gcd routine, and how many cycles does the processor take in total to run the program.

Note: This is a discussion question, so this will not be autograded, you should add your two numbers in a discussion.txt file. During checkoff you will show the TA the code that got your numbers. You can comment out the display statements when you submit your code if they get in the way for other portions of the lab.

Note 2: To run just the gcd_sw.S test, copy the corresponding gcd_sw.vmh file to the top directory and rename it mem.vmh. Then start the simulator by calling the executable simulating your processor: ./PrincetonMulticycle. When the simulator starts, it will look for a file called mem.vmh to get the initial state of memory.

Note 3: You do not need to complete this problem before continuing in the lab. If you get stuck, you can continue with the later parts and ask for help at office hours or over piazza.

3.2 A Hardware GCD Accelerator

Looking at the code of GCD in RISC-V assembly, it seems that there is a fair amount of time spent in branches and other administrative tasks. A hardware implementation of GCD may be faster since the control flow is hard-wired. Therefore let’s consider the following GCD module:

```
interface GCD;
    method Action start(Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
endinterface

module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0);
    Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy <- mkReg(False);
    rule gcd;
        if (x >= y) begin
            // subtract
            x <= x - y;
        end else if (x != 0) begin
            // swap
            x <= y;
            y <= x;
        end
    endrule
```
method Action start(Bit#(32) a, Bit#(32) b) if (!busy);
x <= a;
y <= b;
busy <= True;
endmethod

method ActionValue#(Bit#(32)) getResult if (busy && (x==0));
busy <= False;
return y;
endmethod
endmodule

Discussion Question 2 (5 Points): How many cycles would it take for this module to compute the three following gcd:

- 25 and 17
- 779 and 1025
- 12920 and 43605
- 1000000 and 1007001

To get this number, you should write a small testbench that instantiates the gcd module (defined in GCD.bsv) and feed it the numbers. The technique on how to do that has been documented and explained in the optional lab. You will also probably use a similar technique as the previous question to add a counter counting the cycles, and printing it.

Note: It is also a discussion question, so you will have to show the code that allowed you to get those numbers during checkoff.

3.3 Connecting the GCD Module

This hardware module is nice, but the software engineers have no idea how to use it, and more precisely how to use it in their existing programs.

Here is one way to hook up an accelerator to a processor:

- Pick an opcode that is free (for example 7'b0001011)
- Extend the decoded instruction type to allow another iType: the GCD type.
- Figure out what you need in this decoded instructions, in this case, two source registers, and a destination register.
- For GCD instructions in the doExecute rule, start the GCD and go to the GCDWait state.
- When the GCD is done, write back the result produced by GCD in the destination register specified by the instruction.

This way you can just change what is at the label gcd, replacing the function with a single instruction:

```
#define GCD(rd, rs1, rs2) .word 0x0000000B | ((rd) << 7) | ((rs1) << 15) | ((rs2) << 20)
gcd:        GCD(10, 10, 11)
ret
```

Note: The #define statement defines a pre-processor macro that the assembly can use to easily write the constant for a GCD instruction.
Exercise 3 (5 Points): Modify Decode.bsv to support the decoding of a new gcd instruction on opcode 7’b0001011.

Exercise 4 (10 Points - only passing gcd_hw_simple is 5 Points): Complete MulticyclePrincetonGCD.bsv to handle the new GCD instruction in hardware by using the GCD module in GCD.bsv. You can test this module by running test.sh and choosing to run the GCD tests.

Discussion Question 3 (5 Points): What is the speedup (the run time for gcd_sw.S, divided by the run time for gcd_hw.S)?
4 Appendix

4.1 Additional BSV

4.1.1 Question Mark

*TL;DR: Question marks in BSV can mess things up. Only use them if you know their value (or data that depends on their value) will never be used.*

In BSV, a question mark (?) is used to denote a “Don’t Care” value, but the exact semantics of the question mark go against most people’s intuition. Due to how they are handled inside the compiler, *they can result in optimizations that do not make sense from the code.* For example, consider the following code where a and b are registers.

```bsv
rule doStuff;
    Bool x = ?;
    case (x)
        True: a <= 0;
        False: b <= 0;
    endcase
endrule
```

If x was not a question mark, then this rule either writes 0 to a or it writes 0 to b. Since x is a question mark, *this rule may do nothing or write 0 to both a and b!* Many times when ? gets printed, it shows up as the bit pattern `0b10101010...` or the hex pattern `0xAAAA...`.

4.1.2 Maybe Types

*Maybe*() is a type constructor that takes in a type and returns a new type with an extra bit to specify if the associated data is valid or not. In the processor *Maybe*(Bit#(5)) is used to represent the destination register. Instructions that write to the register file have a Valid destination register. Instructions that do not write to the register file have an Invalid destination register.

```bsv
Maybe#(Bit#(5)) rd_a = Valid(3); // for an instruction that writes to x3
Maybe#(Bit#(5)) rd_b = Invalid; // for an instruction that does not write to a register
```

To figure out if a value of a Maybe type is valid or invalid, use the `isValid()` function.

```bsv
function Bool isValid(Maybe#(t) x);
    // returns True if x is Valid, otherwise returns False
endfunction
isValid(rd_a) // returns True
isValid(rd_b) // returns False
```

To get the data from a Valid value of a Maybe type, use the `fromMaybe()` function.

```bsv
function t fromMaybe(t default_value, Maybe#(t) x);
    // returns the Valid value of x if x is Valid, otherwise returns default_value
endfunction
fromMaybe(? , rd_a) // returns 3 (valid value of rd_a)
fromMaybe(? , rd_b) // returns ? (read note above about ?)
```

4.1.3 Struct Types

A *struct*, or structure, in BSV is a way to collect values of multiple types into a value of a single Struct type. In the processor, Struct types are used to hold the decoded instruction (DecodedInst), the executed
instruction (ExecInst), and the memory requests (MemReq). Struct types are defined using a typedef similar to Enum types.

```markdown
typedef struct {
    MemOp op;
    Word addr;
    Word data;
} MemReq deriving (Bits, Eq, FShow);
```

The above typedef defines a struct type MemReq with the fields named op, addr, and data. These fields are of type MemOp, Word, and Word respectively. The fields can be accessed and assigned using the syntax below.

```markdown
MemReq r = ?;
r.op = St;
r.addr = 0x400;
r.data = 1234
```

The above code creates a store request to write data 1234 to address 0x400. The short-hand notation for creating the same store request is shown below.

```markdown
MemReq r = MemReq{ op: St, addr: 0x400, data: 1234 };
```

### 4.1.4 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.

```markdown
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.

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