Due date: Thursday November 7th 11:59:59pm EST

Points: This lab is worth 14 points (out of 200 points in 6.004).

Getting started: To create your initial Lab 6 repository, please visit the repository creation page at https://6004.mit.edu/web/fall19/user/labs/lab6. Once your repository is created, you can clone it into Athena by running:

```
git clone git@github.mit.edu:6004-fall19/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 (https://6004.mit.edu, Labs/Didit tab) to verify that your submission was correctly pushed and passes all the tests.** If you finish the lab in time but forget to push, you will incur the standard penalties for late submissions.

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 (by Wednesday November 13th). The checkoffs for Lab 6 will be held beginning on Friday November 1st. See the course website for lab hours.

Introduction

In this lab you will implement a general-purpose processor, i.e., a circuit capable of performing *arbitrary computations*. In previous labs, you built several combinational and sequential circuits that performed increasingly complex computations. But each of these circuits performed a single function, like adding, multiplying, or sorting numbers.

Specifically, this lab asks you to implement a single-cycle RISC-V processor in Minispec. Your processor should implement RV32I, the base integer 32-bit variant of the RISC-V instruction set architecture (ISA) that we've studied in Lectures 2–4. Your processor should execute one instruction every cycle, following the implementation in Lecture 14. As before, for Minispec-related questions for this lab, you may want to refer to the Minispec combinational and sequential logic tutorials, linked from the class resources page, and the Minispec reference guide.

To pass the lab you must complete and PASS all of the exercises.

Coding guidelines: You are only allowed to change the following files: `ALU.ms`, `Decode.ms`, `Execute.ms`, `Processor.ms`, and `quicksort.S`. Modifications to other files will be overwritten during Didit grading.

1 Processor Organization and Implementation Guidelines

Unlike prior labs, where you had to implement relative small functions and modules, a processor is a large system. Thus, there are many ways to structure the code for your processor. However, *we want you to follow a specific structure*, building the processor using several functions and modules with fixed interfaces. There are three good reasons for imposing this structure:

1. This structure leads to a simple implementation strategy, and we have many tests to help you along the way. Thus, it will help you implement the processor and reduce debugging time.
2. This structure will help us help you. *If you deviate from it, we won't be able to help you in lab or Piazza if you get stuck.*
3. This structure is designed to be *easy to pipeline*. While this is not important now, it will be crucial for the design project.

Figure 1 shows the overall structure of the processor, which mirrors what we learned in Lecture 14.

We already give you implementations for all the state elements: the register file and the instruction and data memories. In this lab, memories have *combinational reads*, which is unrealistic but necessary if we want

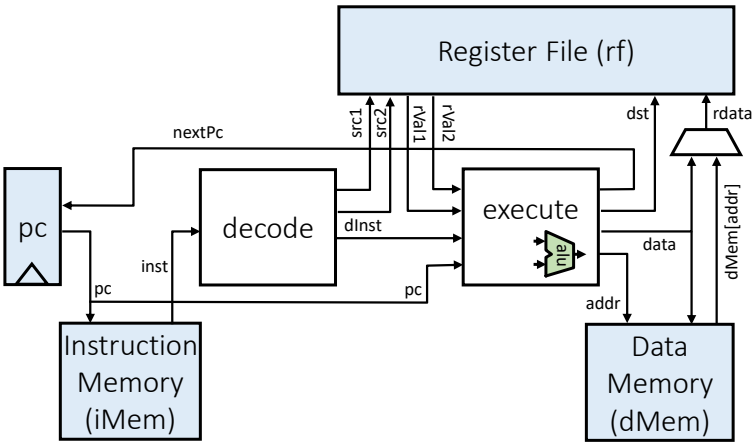

Figure 1: Overall structure of the single-cycle RISC-V processor.

the processor to execute an instruction in a single cycle. Because they are unrealistic, the instruction and data memories are instances of a module called `MagicMemory` (we will see how real memories are implemented in the next lab).

We ask that you structure your code in three main blocks:

1. A `decode` function (in file `Decode.ms`) that implements a RISC-V instruction decoder (Section 4).
2. An `execute` function (in file `Execute.ms`) that computes all outputs of an instruction: the value to write to its destination register (if any), the inputs to the data memory, and the next PC (Section 5). This code will use the ALU you built in Lab 4.
3. A rule within the `Processor` module (in file `Processor.ms`) that uses the `decode` and `execute` functions and all state elements (PC register, instruction memory, register file, and data memory) to execute an instruction.

Due to this structure, the changes to the `Processor` module module will be quite short, and most of your effort will be in implementing the `decode` and `execute` functions.

Implementation guidelines: We recommend that you implement your processor instruction type by instruction type, as described in Lecture 14. To help you along the way, this lab includes a wide range of tests: microtests that test the implementation of a specific instruction, and larger assembly tests. The later sections describe these tests and guide you through the implementation. At a high level, we **heavily recommend** that you follow these steps to implement your processor:

1. Fill in `Processor.ms` to ensure your processor can load instructions from memory, use your decode and execute functions, read from and write to registers, load data from and store data to memory, and update the program counter (`pc`) for the next cycle.
2. For each microtest, *in order*,
   (a) implement the corresponding instruction in `Decode.ms`,
   (b) test with `DecodeTB.ms`,
   (c) implement the instruction in `Execute.ms`, and
   (d) test with `test.py`.
3. For each instruction class (e.g., OP, OP-IMM, BRANCH),
   (a) implement the corresponding instructions in `Decode.ms`,
   (b) test with `DecodeTB.ms`,
   (c) implement the instructions in `Execute.ms`, and
   (d) test with `test.py`.

This is a very methodical approach; it is easier to debug by working in small increments and testing, rather than doing everything at once. Testing frequently makes it easier to isolate the cause of a bug in your processor to a recent change.

2 Processor Test Suite

The git repository for lab 6 contains two sets of tests, located under `sw/`:

- Microtests, in `sw/microtests`
- Full assembly tests, in `sw/fullasmtests`

(There is also a `sw/quicksort` directory, which you will fill in later, in Section 6.)

You can compile the tests into assembly code by running `make -C sw`. 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.

2.1 Microtests

You will test (and debug) your processor by running short test programs. This approach should give you pause: how can you test a processor with the very instructions you're testing? To bring up your processor, you'll add functionality in small increments, and rely on as little of the processor as possible.

To reduce our dependence on a fully correct processor, we want to directly inspect the state of our processor, rather than allow the processor to determine the outcome of its own tests. To do this, the microtests rely on the `unimp` instruction, a pseudoinstruction that represents an unimplemented instruction. This instruction halts the processor and instructs the simulation to dump the processor state (the register file and program counter).

Each microtest is designed to test a specific instruction. Microtests build on each other: a given microtest assumes that instructions tested by prior microtests are correct. For example, the first microtest is a single `unimp`. (Seven additional microtests test for other variants of the `unimp` instruction.) The next microtest tests the `lui` instruction, followed by an `unimp`. Passing subsequent microtests is highly unlikely if any of the preceding microtests fail. The microtests make it reasonably likely that your processor, at a minimum, can correctly run the instructions needed to run the full suite of assembly tests, described later.

If you fail a microtest, the `test.py` script will print out your final state and how it differs from the expected processor state, as well as a short description of the test. You can also take a look at the assembly code in the `sw/build/microtests` folder, the expected processor state in the `sw/microtests/{some_test}.expected` folder, and your final state in the `test_out` folder. Try to determine what went wrong by examining the differences in state.

2.2 Fullasmtests

The `fullasmtests` are more comprehensive 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. In the code that each `fullasmtest` uses to exit, it will write some value at the memory address `0x40001000`. If 0 is written, the test passes. Otherwise, the test fails. 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. As an example, 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`.

2.3 Testing Your Processor with `test.py`

`test.py` is the test script for Lab 6. You can run it by typing `./test.py` or `python3 test.py`. (Note that it does not compile your processor for you, so after you've changed any Minispec, you should run `make Processor` before re-running `test.py`.)

`test.py` will prompt you to specify which tests you want to run on your processor. Alternatively, you can directly specify which tests you want to run as a command-line argument: running `./test.py a` is the same as responding `a` to the prompt, which can be useful if you're repeatedly running the same tests. For example, to test your processor with the first four microtests, you can run `./test.py 1,2,3,4`; to test it on just the `sw` full asm test, you can run `./test.py sw`.

Note that `test.py` processes your processor simulation output, including any `$display` statements you might insert, to check if your test passes, so it doesn't print the output. However, it saves the outputs under the `test_out/` directory, in the subfolders corresponding to the tests' locations in `sw/`. So, after running `test.py` you can view the files in that directory to see the full output, including the results of any display statements you inserted.

You can also directly run your processor without `test.py`. After running `test.py` on some test, the script will leave a `mem.vmh` file in your lab directory with the memory contents for that test. You can then directly run `./Processor` to re-run the simulation and see the full output. In general, you can take any `.vmh` file from those under `sw/build/`, copy it as `mem.vmh` in your lab directory, and run `./Processor`. That will simulate your processor on the program corresponding to that individual test and give you the full output (but without any check as to whether it's correct). For example, to run your processor on the first LUI microtest directly, assuming you've already compiled your processor with `make Processor`, you could use the command:

```
ln -sf sw/build/microtests/01_lui.vmh mem.vmh && ./Processor
```

(Technically this creates a *symbolic link* to the `.vmh` file instead of copying it, but it's faster and for our purposes it has the same effect.)

2.4 Debugging Minispec

Section 7 of the Minispec sequential logic tutorial (linked from the class resources page) covers Minispec functions and system functions that may be useful for debugging your processor. The `$display` system function can be used to print debugging information, and the `fshow` function can be used to convert structs and other complex types to a human-readable string. The tutorial has many more examples.

3 The Processor Module

The `Processor` module in `Processor.ms` should implement the single-cycle processor. We have provided skeleton code for `Processor`, which instantiates all state elements and has a single rule that should execute an instruction. Before you can test your processor, you need to complete the skeleton code for this rule. Fortunately, because we have structured the code to have most of the logic in the `decode` and `execute` functions (Section 1), the `Processor` code you need to write is quite short, less than 20 lines of code.

Exercise 1: Complete the `Processor` module in `Processor.ms`.

You can refer to slide 5 of Lecture 14 for an overview of what your processor needs to do. Overall, your processor needs to do these things every cycle:

1. **Fetch the instruction** your processor should decode and execute from memory, i.e. load it. The program counter `pc` will hold the address of this instruction. For example, at the start of every microtest when `pc` is 0, your processor should load the word at address 0.

   NOTE: The two memory modules, `iMem` and `dMem`, have *combinational reads*: memory reads return data in the same cycle. This is unrealistic, and hence these memories are `MagicMemory` modules. In

future lectures and in the design project, you will learn how to implement a processor with memories where reads return data one or several cycles later.

For this part of the processor, you should use the instruction memory, `iMem`. To load from a memory, call its `read()` method, which accepts a `Word` as the address you want to access. For example, `iMem.read(32'd4)` reads the word at address 4. Addresses must be multiples of 4.

2. **Decode the instruction** to figure out its instruction type, ALU operation, and operands. For example, in microtest 1, the first instruction in raw hexadecimal is `0x000010b7`. To decode it, you must find that it is a LUI instruction with destination register `x1` and immediate `0x00001000`.

   In `Processor.ms`, the `decode` function is already imported from `Decode.ms`. It takes in a single argument, the instruction as a `Word`, and returns a struct of type `DecodedInst`. For now, just call `decode` with the instruction you loaded, and put the result in a variable; you will fill in `decode` in Section 4.

3. **Read from the registers** any values that the instruction might need. We have proivded you with a register file `rf`, of type `RegisterFile`, which contains 32 registers, where register 0 is hardwired to 0. In every cycle, you can read from two of its registers and write to one of its registers. The code to read from `rf` is `rf.rd1(x)` or `rf.rd2(x)` (there are two methods because you should only call each of these methods once per cycle), where `x` is a `Bit#(5)`, the x-number of the register.

   You can get the x-numbers of the registers to read from your `DecodedInst`, which has `src1` and `src2` fields. Note that it is safe and actually simpler than the alternative to always read from two registers every cycle, even for instructions that only need values from zero or one registers, since reading unnecessary values doesn't have side effects and can just be ignored by the next step.

4. **Execute the instruction** to figure out what you need to do. Effects of the instruction include that you might need to write to a register, load data from memory, store data to memory, and update the program counter `pc` to the next value.

   In `Processor.ms`, the `execute` function is already imported from `Execute.ms`. It takes in four arguments:

   (a) the decoded instruction as a `DecodedInst`;

   (b) the value in the first register to be read (`rs1`), as a `Word`, if any;

   (c) the value in the second register to be read (`rs2`), as a `Word`, if any;

   (d) the current program counter (`pc`), as a `Word`.

   It returns a struct of type `ExecInst`. You should call `execute` with the instruction you decoded and the other information required, and put the result in a variable; you will fill in `execute` in Section 5.

5. **Load from or store to** memory, if the instruction requires you to. Use `dMem`. Like `iMem`, you can load from `dMem` with the `read()` method. To *write* to the data memory, use the `write` input to `dMem`, which accepts a `Maybe#(MemWriteReq)`. The `MemWriteReq` struct has the following format:

   ```
   typedef struct { Word addr; Word data; } MemWriteReq;
   ```

   For example, to write `0x1234` to address `0x100`:

   ```
   dMem.write = Valid(MemWriteReq{addr: 32'h100, data: 32'h1234});
   ```

   If you are executing a LW instruction, the data you load from memory needs to get written to a register. You can put the loaded data in the `data` field of your `ExecInst` so that the logic for writing a register (in the next step) can handle it like all other register writes.

6. **Write to a register**, if the instruction requires you to. Writing `rf`, allowed only once each cycle, is done by setting the register file's `wr` input:

   ```
   rf.wr = Valid(RegWriteArgs{index: x, data: data});
   ```

where `x` is the x-number of the register and `data` is a `Word`, the data you are writing into the register.

You can get the register you might need to write from your `ExecInst`, which has a `dst` field.

Note that unlike the reading from registers step, if an instruction isn't supposed to write to a register, then you need to make sure no register is written to. To do this, set `rf.wr` to `Invalid` (or don't set it).

7. **Update the program counter `pc`**. You can again get this from your `ExecInst`, which has a `nextPc` field.

Important: We have provided code that terminates your processor if it encounters an unsupported instruction. **Do not edit this code**, as it is required to be exactly as-is for the microtests to work. You can freely add debugging statements (Section 2.4) **before** this code.

```
// If unsupported instruction, stops simulation and print the state of the processor
// IMPORTANT: Do not modify this code! The microtests check for it.
if (eInst.iType == Unsupported) begin
    $display("Reached unsupported instruction (0x%x)", inst);
    $display("Dumping the state of the processor");
    $display("pc = 0x%x", pc);
    $display(rf.fshow);
    $display("Quitting simulation.");
    $finish;
end
```

4 Decoder

The remainder of the work consists on implementing the `decode` and `execute` functions, found in `Decode.ms` and `Execute.ms`, respectively. Their implementations are mostly empty. However, all of the constants needed are already defined for you in `ProcTypes.ms`, and repeated as comments in the relevant files.

The decoder (i.e., the `decode` function) implements a RISC-V decoder, which generates all the control signals for the instruction. `decode` returns a `DecodedInst` struct that contains all the signals: the instruction type, the source and destination register indices (if any), the 32-bit immediate, and the control inputs to the ALU and branch ALU. For simplicity, these control signals are a bit higher-level than what we saw in Lecture 14. For example, there is no PCSEL signal; instead, selecting the right value for the next PC can be done based on the instruction type.

The 6.004 ISA reference details the encoding of different instruction types, which should suffice to implement `decode`. In addition, slides 12–29 of Lecture 14 show how to decode most instructions, and slide 10 gives a specific example.

Exercise 2: Implement the `decode` function in `Decode.ms`.

You can implement `decode` either all at once, or instruction type by instruction type (in parallel with `execute`, so that you get more and more microtests working).

To get you started, we have implemented the decoding of three instructions for you: `AUIPC`, `ANDI`, and `SLLI`.

`AUIPC` is the instruction mnemonic 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 adds the current value of the program counter (the address of the `auipc` instruction) to the 20-bit upper immediate encoded in the instruction. The part of the decoder implementing `auipc` looks like this:

```
opAuipc: dInst = DecodedInst { dst: validDst, 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 (`RIndx`), it is a Maybe type (`Maybe#(RIndx)`). Your decoder should give a Valid value only when the instruction writes to the register file, and an Invalid value otherwise. In other words, `dst`'s valid bit directly maps to the WERF control signal from Lecture 14; by using a Maybe type, we make it harder to inadvertently write to the register file when we shouldn't.

If you choose to implement your decoder all at once, you can directly run tests on your `decode` function with `make decode_test`, which runs tests from the file `DecodeTB.ms`.

5 Execute

Your processor's `execute` function is mainly providing operands to the ALU. The ALU you built in Lab 4 (`alu` function) has all the necessary functions, so simply copy over your ALU.ms file from Lab 4. If you implemented a fast adder in Lab 4, make sure that your ALU calls it if you would like to use it.

The `execute` function needs to compute three values to pass to the main processor logic:

1. `data`, the value that should be written into a register or stored into memory, if the instruction does either of those things. If not, it can be any value. The value of `data` depends heavily on the instruction; this is where your ALU will be useful.

2. `nextPc`, the value of program counter in the next cycle, i.e., the address of the next instruction that should be executed. For most instructions, this is just `pc + 4` because we want to execute the immediately next instruction in memory, but it will be different for jumps (`JAL`, `JALR`) and branches. We have implemented the default case and the JALR case for you. We have also implemented a simple branch ALU (`aluBr` function), as described in Lecture 14 slide 25, which computes whether a branch should be taken.

3. `addr`, the address that the processor should load from or store to, if the instruction does either of those things. If not, it can be any value.

Exercise 3: Using your ALU from Lab 4, implement in the `execute` function in `Execute.ms`.

As you fill in your processor and `decode` and `execute` functions, you can build your processor by running `make Processor`.

You can run a suite of tests on the processor by running `./test.py`. After filling in your processor and at least parts of your `decode` and `execute` functions, you should be able to pass some early microtests (`./test.py 1`, `./test.py 2`, etc.) To get credit for finishing your processor, you should pass the `microtests` (`./test.py a`) and the `fullasmtests` (`./test.py f`). For more information on using the test script, refer to Section 2.3.

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

6 Run software on your processor

Now that you have a working processor, you can run any software written in its ISA. Let's run the `quicksort` you implemented in Lab 2.

Exercise 4: Compile your `quicksort` from Lab 2 and run the program on your single-cycle processor.

To compile your code, copy `quicksort.S` from Lab 2 into the `sw/quicksort` directory and run `make -C sw quicksort`.
To run your code on your processor, run `./test.py q`.