EECS 150
Lab 4: Simulation and Testing

Prof. John Wawrzynek
TAs: Simon Scott, Ian Juch
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

1 Introduction

In this lab, you will learn how to simulate your modules and test them in software before pushing them to the board. In the previous labs, you had to push your code through the entire tool chain and impact the bit stream onto the FPGA before you could verify that your design worked. This is feasible for simple designs that can quickly be synthesized and quickly verified on the board, but this approach does not scale.

1.1 RISC-V

The final project for this class will be an FPGA implementation of a RISC-V (pronounced *risk-five*) CPU. RISC-V is a new instruction set architecture (ISA) developed here at UC Berkeley. It was originally developed for computer architecture research and education purposes, but recently there has been a push towards commercialization and industry adoption. For the purposes of this lab, you don't need to delve too deeply into the details of RISC-V. However, it may be good to familiarize yourself with it, as this will be at the core of your final project. Check out the official Instruction Set Manual and explore `riscv.org` for more information.

2 Prelab

1. Read through sections 2.2 and 2.3 starting on page 11 in the RISC-V Instruction Set Manual to understand how the different types of instructions are encoded. Most of this should be familiar as this is similar to MIPS.

2. Read through sections 2.4, 2.5, and 2.6 starting on page 13 in the Instruction Set Manual and think about how each of the instructions will use the ALU.

You do not need to read 2.7 or 2.8, as you will not be implementing those instructions in the project.

3 Lab Procedure

In this lab, you will be writing the ALU that you will be using later on for the next lab and your project. You will also be learning techniques for simulating and verifying your design which are critical aspects of the development flow and for your project.

3.1 Relevance to Final Project

The ALU that we will implement in this lab is for a RISC-V instruction set architecture. Your project may or may not directly make use of the ALU that we will implement in this lab but you may need a modified or entirely new ALU depending on your project. Pay close attention to the design patterns and how the ALU is intended to function in the context of the RISC-V processor. In particular it is important to note the separation of the datapath and control used in this system which we will explore more in the next lab.

The specific instructions that your ALU will need to support are shown in the tables below that describe the ISA. Before you get overwhelmed by the tables, remember that you will only be implementing the ALU and the ALU decoder, not the entire processor. These tables are here for your reference. In the event that we were to implement a processor, the tables contain the ISAs you would be using. The branch condition should **not** be calculated in the ALU. Depending on your CPU implementation, your ALU may or may not need to do anything for branch, jump, load, and store instructions (i.e., it can just output 0). However, for the purposes of this lab, we will assume that all of those instructions will use the ALU to calculate the target addresses.

3.2 Functional Specification

The encoding of each instruction is shown in the table below. There is a detailed functional description of each of the instructions in Section 2.4 (starting on page 13) of the Instruction Set Manual. Pay close attention to the functional description of each instruction as there are some subtleties. Also, note that the `LUI` instruction is somewhat different from the MIPS version of `LUI` which some of you may be used to.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	SB-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	UJ-type

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI rd,imm
imm[31:12]				rd	0010111	AUIPC rd,imm
imm[20|10:1|11|19:12]				rd	1101111	JAL rd,imm
imm[11:0]		rs1	000	rd	1100111	JALR rd,rs1,imm
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ rs1,rs2,imm
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE rs1,rs2,imm
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT rs1,rs2,imm
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE rs1,rs2,imm
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU rs1,rs2,imm
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU rs1,rs2,imm
imm[11:0]		rs1	000	rd	0000011	LB rd,rs1,imm
imm[11:0]		rs1	001	rd	0000011	LH rd,rs1,imm
imm[11:0]		rs1	010	rd	0000011	LW rd,rs1,imm
imm[11:0]		rs1	100	rd	0000011	LBU rd,rs1,imm
imm[11:0]		rs1	101	rd	0000011	LHU rd,rs1,imm
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB rs1,rs2,imm
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH rs1,rs2,imm
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW rs1,rs2,imm
imm[11:0]		rs1	000	rd	0010011	ADDI rd,rs1,imm
imm[11:0]		rs1	010	rd	0010011	SLTI rd,rs1,imm
imm[11:0]		rs1	011	rd	0010011	SLTIU rd,rs1,imm
imm[11:0]		rs1	100	rd	0010011	XORI rd,rs1,imm
imm[11:0]		rs1	110	rd	0010011	ORI rd,rs1,imm
imm[11:0]		rs1	111	rd	0010011	ANDI rd,rs1,imm
0000000	shamt	rs1	001	rd	0010011	SLLI rd,rs1,shamt
0000000	shamt	rs1	101	rd	0010011	SRLI rd,rs1,shamt
0100000	shamt	rs1	101	rd	0010011	SRAI rd,rs1,shamt
0000000	rs2	rs1	000	rd	0110011	ADD rd,rs1,rs2
0100000	rs2	rs1	000	rd	0110011	SUB rd,rs1,rs2
0000000	rs2	rs1	001	rd	0110011	SLL rd,rs1,rs2
0000000	rs2	rs1	010	rd	0110011	SLT rd,rs1,rs2
0000000	rs2	rs1	011	rd	0110011	SLTU rd,rs1,rs2
0000000	rs2	rs1	100	rd	0110011	XOR rd,rs1,rs2
0000000	rs2	rs1	101	rd	0110011	SRL rd,rs1,rs2
0100000	rs2	rs1	101	rd	0110011	SRA rd,rs1,rs2
0000000	rs2	rs1	110	rd	0110011	OR rd,rs1,rs2
0000000	rs2	rs1	111	rd	0110011	AND rd,rs1,rs2

3.3 Lab Files

As before, to retrieve the lab files you should navigate to your `labs` git repo and do a `git pull`. Make sure that you have both a `/src` and `/sim` folder in the `/lab4` directory. Notice that there is no `Makefile` at the top level as we will not be synthesizing our design to the board, only verifying it works in simulation.

3.4 Testing the Design

Before writing any of our modules, we will first write the tests so that once you've written the modules you'll be able to test them immediately. Another reason why you should write your tests first is that if you need to change your module design, you can always run it against your test to see if it still works. You should also understand the expected functionality of these modules before writing any code or tests.

There are a few approaches you can use to test your design. For this lab, you will only be testing two modules, so you will resort to unit testing. For your project, you will be expected to write unit tests for any modules that you design and implement and write integration tests.

3.4.1 Verilog Testbench

One way of testing Verilog code is with test bench files. The skeleton of a test bench file has been provided for you in `ALUTestbench.v`. There are several important parts of this file to note:

- `` `timescale 1ns / 1ps `` - This specifies the reference time unit and the time precision. This means that every delay in the test bench is 1ns long and the simulation should be accurate up to 1ps.
- Clock generation is done with the code below. Since the ALU is actually only combinational logic, this portion is not necessary. You may treat it as a reference for when you need to write a test bench for a sequential circuit.
  - The `initial` block to set the clock to 0 at the beginning of the simulation. You **must** start the clock at 0, otherwise you will be trying to change inputs at the same time the clocks changes and it will cause strange behavior.
  - You must use an always block without a trigger list to cause the Clock to change by itself

```
parameter Halfcycle = 5; //half period is 5ns
localparam Cycle = 2*Halfcycle;
reg Clock;
// Clock Signal generation:
initial Clock = 0;
always #(Halfcycle) Clock = ~Clock;
```

- `task checkOutput;` - this task encapsulates some Verilog that you would otherwise have to copy paste over and over. Note that it is **not** the same thing as a function (as Verilog also has functions).

- `{$random} & 31'h7FFFFFFF` - `$random` generates a pseudorandom 32-bit integer. We mask the result to get it into the appropriate range.

For these two modules, the inputs and outputs that you care about are `opcode`, `funct`, `add_rshift_type`, `A`, `B` and `Out`. To test your design thoroughly, you should work through every possible `opcode`, `funct`, and `add_rshift_type` that you care about, and verify that the correct `Out` is generated from the `A` and `B` that you pass in.

The test bench generates random values for `A` and `B` and computes `REFout = A + B`. It also contains calls to `checkOutput` for load and store instructions, for which the ALU should perform addition. It will be up to you to write tests for the remaining combinations of `opcode`, `funct`, and `add_rshift_type` to test all your other instructions.

Remember to restrict `A` and `B` to reasonable values (e.g. masking them, or making sure that they are not zero) if necessary to guarantee that a function is successfully tested. Please also write tests where the inputs and the output are hardcoded. These should be corner cases that you want to be certain are stressed during testing.

3.4.2 Test Vector Testbench

An alternative way of testing is to use a test vector, which is a series of bit arrays that map to the inputs and outputs of your module. The inputs can be all applied at once if you are testing a combinational logic block, such as in this lab, or applied over time for a sequential logic block (e.g. an FSM).

You will write a Verilog test bench that takes the parts of the bit array that correspond to the inputs of the module, feeds those to the module, and compares the output of the module with the output bits of the bit array. The bit vector should be formatted as follows:

```
[106:100] = opcode
[99:97] = funct
[96] = add_rshift_type
[95:64] = A
[63:32] = B
[31:0] = REFout
```

Open up the skeleton provided to you in `ALUTestVectorTestbench.v`. You need to complete the module by making use of `$readmemb` to read in the test vector file (named `testvectors.input`), writing some assign statements to assign the parts of the test vectors to registers, and writing a for loop to iterate over the test vectors.

The syntax for a for loop can be found in `ALUTestbench.v`. `$readmemb` takes as its arguments a filename and a reg vector, e.g.:

```
reg [5:0] bar [0:20];
$readmemb(foo.input, bar);
```

3.4.3 Writing Test Vectors

Additionally, you will also have to generate actual test vectors to use in your test bench. A test vector can either be generated in Verilog (like how we generated A, B using the random number generator and iterated over the possible opcodes and functs), or using a scripting language like

python. Since we have already written a Verilog test bench for our ALU and decoder, we will tackle writing a few test vectors by hand, then use a script to generate test vectors more quickly.

Test vectors are of the format specified above, with the 7 `opcode` bits occupying the left-most bits. Open up the file `sim/tests/testvectors.input` and add test vectors for the following instructions to the end (i.e. manually type the 107 zeros and ones required for each test vector): `SLT`, `SLTU`, `SRA`, and `SRL`.

In the same directory, we've also provided a test vector generator written in Python, which is a popular language used for scripting. We used this generator to generate the test vectors provided to you. If you're curious, you can read the next paragraph and poke around in the file. If not, feel free to skip ahead to the next section.

The script `ALUTestGen.py` is located in `sim/tests`. Run it so that it generates a test vector file in the `/sim/tests/` folder. Keep in mind that this script makes a couple assumptions that aren't necessary and may differ from your implementation:

- Jump, branch, load and store instructions will use the ALU to compute the target address.
- For all shift instructions, `A` gets shifted by `B`. In other words, `B` is the shift amount.
- For the `LUI` instruction, the value to load into the register is fed in through the `B` input.

You can either match these assumptions or modify the script to fit with your implementation. All the methods to generate test vectors are located in the two Python dictionaries `opcodes` and `functs`. The lambda methods contained (separated by commas) are respectively: the function that the operation should perform, a function to restrict the `A` input to a particular range, and a function to restrict the `B` input to a particular range.

If you modify the Python script, run the generator to make new test vectors. This will overwrite the file, so if you want to save your handwritten test vectors, rename the file before running the script, then append them once the file has been generated.

```
% python ALUTestGen.py
```

This will write the test vector into the file `testvectors.input`. Use this file as the target test vector file when loading the test vectors with `$readmemb`.

3.5 Writing Verilog Modules

For this lab, we've provided the module interfaces for you. They are logically divided into a control (`ALUdec.v`) and a datapath (`ALU.v`). The datapath contains the functional units while control contains the necessary logic to drive the datapath. You will be responsible for implementing these two modules. Descriptions of what each of the inputs and outputs of the module mean can be found in the top few lines of the files. The ALU should take an `ALUop` and its two inputs `A` and `B`, and provide an output dependent on the `ALUop`. The operations that it needs to support are outlined in the Functional Specification. Don't worry about sign extensions, they should take place outside of the ALU. The ALU decoder uses the `opcode`, `funct`, and `add_rshift_type` to determine the `ALUop` that the ALU should carry out. The `funct` input corresponds to the `funct3` field from the ISA encoding table. The `add_rshift_type` input is used to distinguish between `ADD/SUB`, `SRA/SRL`, and `SRAI/SRLI`; you will notice that each of these pairs have the same `opcode` and `funct3`, but differ in the `funct7` field.

You will find the case statement useful, which has the following syntax:

```
always@(*) begin
    case(foo)
        3'b000: // something happens here
        3'b001: // something else happens here
        3'b010, 3'b011: // you can have more than
                        // one case do the same thing
        default: // everything else
    endcase
end
```

To make your job easier, the lab comes with two Verilog header files (`Opcode.vh` and `ALUop.vh`). They provide, respectively, macros for the opcodes and functs in the ISA, and macros for the different ALU operations. You can feel free to change `ALUop.vh` to optimize the `ALUop` encoding, but if you change `Opcode.vh`, you will break the test bench skeleton provided to you. You can use these macros in by placing a backtick in front of the macro name, e.g.:

```
case(opcode)
`OPC_STORE:
```

is the equivalent of:

```
case(opcode)
7'b0100011:
```

3.6 Using Modelsim

Once you've written your test benches as well as implemented the Verilog modules, you can now simulate your design. In this class, you will be using ModelSim, a popular hardware simulation and debugging environment. The staff has wrapped up the functionality that you will need from ModelSim in a `Makefile`. To simulate your design, you must first compile it and fix any syntax errors that arise:

```
% cd ~/lab4/sim
% make compile
```

Once you have your design compiling, you need to run some test cases. The build system looks inside the tests directory for test cases to run. Each test case is a `.do` file, which is a script in Tcl, a scripting language used by a variety of CAD tools. For the most part you don't need to worry about the details of Tcl; you will just be using it to issue commands directly to ModelSim. The following is the Tcl script that runs `ALUTestbench`.

```
set MODULE ALUTestbench
start $MODULE
add wave $MODULE/*
add wave $MODULE/DUT1/*
add wave $MODULE/DUT2/*
run 100us
```

The first line sets the value of the variable `MODULE` to `ALUTestbench`. Its value is referenced through the rest of the script as `$MODULE`. The `start` command tells ModelSim which Verilog module it should simulate.

The `add` command is interesting. By default, ModelSim doesn't collect any waveform information from the simulation. '*' is a shortcut for "anything/all", so these commands tell ModelSim to record the signals for all the signals in the test bench as well as the signals in `DUT1` and `DUT2`. Once you start building designs with more complexity, you may want to look at the signals inside a given submodule. To add these signals, simply edit the `.do` file by adding a new `add wave <target>` command. For example, if `DUT1` and `DUT2` contain a module called `my_submodule`:

```
add wave $MODULE/DUT1/my_submodule/*
add wave $MODULE/DUT2/my_submodule/*
```

Finally, the `run` command actually runs the simulation. It takes an amount of time as an argument, in this case 100us (100 microseconds). Other units (ns, ms, s) are possible. The simulation will run for this amount of time. In most cases this will serve as a timeout because your test benches should cause the simulation to exit (using the `$finish()` system call) when they are done.

Let's try running the simulation. To run all of the cases in the tests directory:

```
% make
```

This will first recompile your design if needed, then run the simulation. Other commands that may be useful are:

- `make clean`: Sometimes you can accidentally cancel a simulation or otherwise cause make to believe that your simulation results are up to date even if they aren't. If you're in doubt, run this command before running make.
- `make results/<testcasename>.transcript`: When you have multiple test benches in your project and you only want to run one of them.

You should see the output of simulation printed to your terminal. It will also be written to `results/<testcasename>.transcript`. You should see the one of the following lines in the output:

```
# FAIL: Incorrect result for opcode 000000, funct: 100011:
# A: 0xdbfa08fd, B: 0x318c32a8, DUTout: 0xaa6dd655, REFout: 0x559229ab
```

or

```
# ALL TESTS PASSED!
```

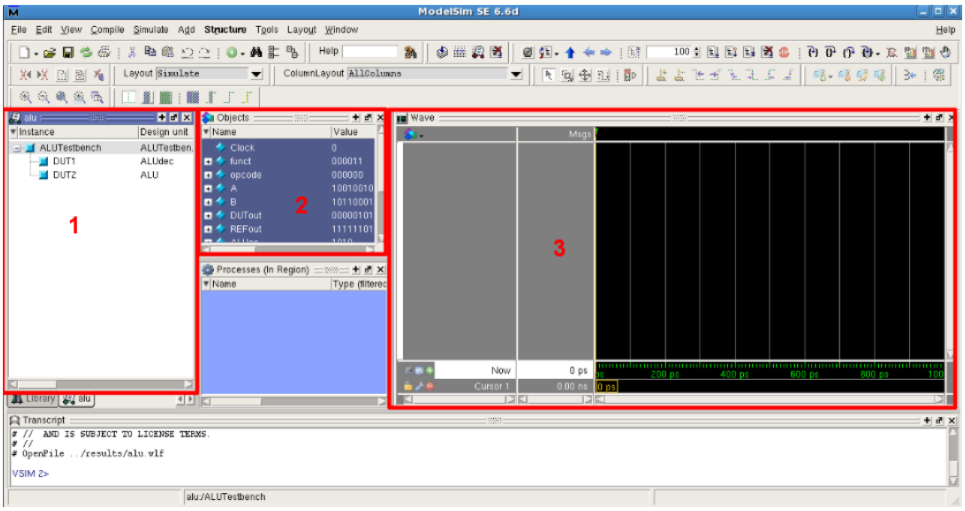
3.7 Viewing Waveforms

After simulation completes you can view the waveforms for signals that you added in your test case script. The waveform database is stored in `.wlf` files inside the results directory. To view them use the `viewwave` script included in the `sim` directory.

```
% ./viewwave results/alu.wlf
```

This will pop open a ModelSim window that shows you a hierarchical view of the signals that your simulation captured.

Note: ModelSim is your **FRIEND**! Throughout the course of the project, ModelSim will be your primary tool for debugging your designs. It is very important that you spend the time to understand how to set up and run tests and use ModelSim to view the results.



The above is a screenshot of ModelSim when you first open it. The boxed screens are:

1. List of the modules involved in the test bench. You can select one of these to have its signals show up in the object window.

2. Object window - this lists all the wires and regs in your module. You can add signals to the waveform view by selecting them, right-clicking, and doing Add ¿ To Wave ¿ Selected Signals.

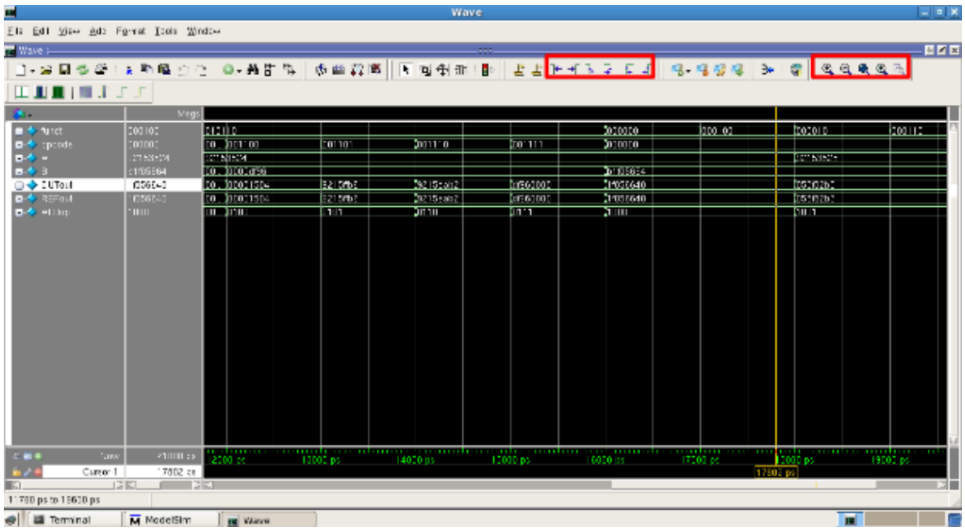
3. Waveform viewer - The signals that you add from the object window show up here. You can navigate the waves by searching for specific values, or going forward or backward one transition at a time.

As an example of how to use the waveform viewer, suppose you get the following output when you run `ALUTestbench`:

```
# FAIL: Incorrect result for opcode 0110011, funct: 101:, add_rshift_type: 1
#       A: 0x92153524, B: 0xffffde81, DUTout: 0x490a9a92, REFout: 0xc90a9a92
```

The `$display()` statement actually already tells you everything you need to know to fix your bug, but you'll find that this is not always the case. For example, if you have an FSM and you need to look at multiple time steps, the waveform viewer presents the data in a much neater format. If your design had more than one clock domain, it would also be nearly impossible to tell what was going on with only `$display()` statements. Besides, you want to get some practice using ModelSim anyhow.

Add all the signals from `ALUTestbench` to the waveform viewer and you see the following window:



The two highlighted boxes contain the tools for navigation and zoom. You can hover over the icons to find out more about what each of them do. You can find the location (time) in the waveform viewer where the test bench failed by searching for the value of `DUTout` output by the `$display()` statement above (in this case, `0x490a9a92`:

1. Selecting `DUTout`
2. Clicking `Edit > Wave Signal Search > Search for Signal Value > 0x490a9a92`

Now you can examine all the other signal values at this time. Compare the `DUTout` and `REFout` values at this time, and you should see that they are similar but not quite the same. From the `opcode`, `funct`, and `add_rshift_type`, you know that this is supposed to be an SRA instruction, but it looks like your ALU performed a SRL instead. However, you wrote

```
Out = A >>> B[4:0];
```

That looks like it should work, but it doesn't! It turns out you need to tell Verilog to treat `B` as a signed number for SRA to work as you wish. You change the line to say:

```
Out = $signed(A) >>> B[4:0];
```

After making this change, you run the tests again and cross your fingers. Hopefully, you will see the line:

```
# ALL TESTS PASSED!
```

If not, you will need to debug your module until all test from the test vector file and the hardcoded test cases pass.

ModelSim is professional grade software and one of the most widely used simulators in industry. As such it has quite a few features that may be useful in certain instances; far too many to detail here. If you need to do something with it, and you feel like that functionality should exist already, Google it. Or ask a TA. But try Google first. If you discover something useful, share your findings!

4 Checkoff

Congratulations! You've written and thoroughly tested a key component in your processor and should now be well-versed in testing Verilog modules. Please answer the following questions to be checked off by a TA.

1. In ALUTestbench, the inputs to the ALU were generated randomly. When would it be preferable to perform an exhaustive test rather than a random test?
2. What bugs, if any, did your test bench help you catch?
3. For one of your bugs, come up with a short assembly program that would have failed had you not caught the bug. In the event that you had no bugs and wrote perfect code the first time, come up with an assembly program to stress the SRA bug mentioned in the above section.

Be prepared to show your working ALU test bench files to your TA and explain your hardcoded cases. You should also be able to show that the tests for the test vectors generated by the python script and your hardcoded test vectors both work.