CMU 18-447: Introduction to Computer Architecture

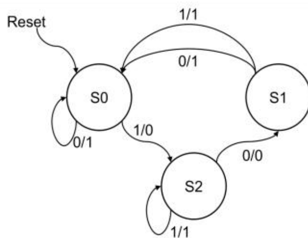
Lab 1.5 (Warmup): Synthesis Workflow and SystemVerilog Register File

Not Due

In this tutorial, you will take a quick tour of the tools we will use in this class, and what it takes to run them by hand. You will then construct a simple register file in SystemVerilog, which should be a review of material from 18-240. If you have not touched Verilog or SystemVerilog in a while, this lab should allow you to recall the basics. In later labs (beginning with Lab 2), you will be developing a more significant codebase in SystemVerilog and simulating it with these tools.

Part 1: Synthesis Tutorial

We will walk through the SystemVerilog synthesis workflow for this class with a simple state machine. The state transition diagram and SystemVerilog description of this Mealy finite state machine with 1-bit input and 1-bit output is given below. (This SystemVerilog file, FSM.sv, can be found in the tarball to save you the trouble of typing it by hand.)




```
module FSM(
    output reg Out,
    input reset_b,
    input clock,
    input In);

     parameter [1:0] S0 = 2'd0, S1 = 2'd1, S2 = 2'd2;
    reg [1:0] state;
     reg [1:0] nextState;

    // Combinatorial
     always_comb begin
          case (state)
          S0: begin
               nextState = In ? S2 : S0;
               Out = In ? 0 : 1;
          end
          S1: begin
               nextState = S0;
               Out = 1;
          end
          S2: begin
               nextState = In ? S2 : S1;
               Out = In ? 1 : 0;
          end
          default: begin
               nextState = 2'bxx;
               Out = 1'bx;
          end
          endcase
     end

    // Synchronous
     always_ff @ (posedge clock, negedge reset_b)
begin
          if (~reset_b)
               state <= S0;
          else
               state <= nextState;
     end
endmodule
```

Preliminaries

Begin by making sure that your environment has the 18-447 settings loaded into it. If your shell is bash (find out by running the command echo $SHELL), to load the 18-447 environment, run the command: source /afs/ece/class/ece447/bin/setup447. If your shell is not bash, change your shell to bash, and see the previous sentence. (You can temporary start bash by entering bash at the command prompt or use chsh to make a long term change.). If you prefer to use other shells, we will assume that you know enough about UNIX to modify the scripts to your liking.

You can run this tutorial from any of the ECE Linux machines. As a rule of thumb, unless you are sitting in front of a terminal in WEH-3716, you should remote log-in to one of the headless ece000~ece031 servers; you will find they are much faster. Please consult https://wiki.ece.cmu.edu/index.php/Clusters for a detailed guide to the ECE computing resources.

To work on this tutorial remotely, you will need a machine with X-windows. (If you have a Windows machine, you can install Xwin-32 downloadable from My Andrew.)

Simulation

This semester, we will be using NC-Verilog. Its flow is very similar to ModelSim, which you may have used before. Create a directory that you will be working in, and copy the above FSM into a SystemVerilog file in that directory -- name it, for instance, FSM.sv. To test the FSM, you should also write a testbench; put it in FSM_tb.sv (or something like it), and name the top-level module FSM_tb. Make sure that your testbench traverses each state in the state-transition diagram at least once. (When you write your testbench, make sure to remember that reset_b is active-low, and make sure to have a $finish! A good starting point for the testbench can be found in /afs/ece/class/ece447/labs/lab1.5/FSM_tb.v.)

Once you have both a testbench and the FSM, compile them. To do so, run:

```
ncvlog -sv -linedebug -messages FSM.sv FSM_tb.sv
```

Hopefully, the output for this said "errors: 0, warnings: 0" for both source files. If not, go back and fix the warnings (or errors!) now. The ncvlog command is similar to ModelSim's vlog; the -linedebug option enables the simulator to give you line-level debugging capabilities, and the -messages option causes ncvlog to print each module it compiles. (N.B.: do not confuse ncvlog with ncverilog! The programs do very different things, and calling the latter when you meant the former will probably result in a lot of confusion.)

After the testbench and FSM have been compiled, they still need to be transformed into a "simulation snapshot" before they can be simulated; this process is called "elaboration". In ModelSim, this happened as part of simulation, but in NC-Verilog, this is done by a separate program called ncelab. To elaborate a snapshot, run:

```
ncelab +access+rwc -messages worklib.FSM_tb
```

If everything was successful, there will be no warnings or errors (which you can see by looking for "*W" or "*E"), and at the end ncelab will say "Writing initial simulation snapshot: worklib.FSM_tb:module". The +access+rwc option tells ncelab to include debugging access in the elaborated snapshot; -messages works as in ncvlog.

Finally, to simulate the snapshot that ncelab produced, use ncsim. ncsim is analogous to vsim from ModelSim, but as usual with NC-Verilog, it is a lot more powerful. ncsim can run in GUI mode, but in this example, we'll run it at the command line and tell it to output debugging information, so we can use the GUI debugger later. To tell ncsim that we want it to run in command line mode, and wait for commands for us first, we use the switch -tcl, and we give it the module that ncelab elaborated for us:

```
ncsim -tcl worklib.FSM_tb:module
```

ncsim tells us it's ready for commands with the "ncsim>" prompt. Run the following commands:

```
ncsim> database -open -shm -default waves
```

This opens up a waveform database in SimVision format, and calls it 'waves'.

```
ncsim> probe -shm -create -depth all -all
```

This makes sure that all of the signals inside your design are logged to the waveform database.

```
ncsim> run
```

This causes ncsim to run your simulation until either 1) everything interesting stops, or 2) it hits a $finish (or $stop, or ...). If you get the warning "*W,RNQUIE: Simulation is complete.", it means that you probably forgot a $finish, but NC-Verilog detected that nothing interesting will ever happen again (i.e., the clock has stopped ticking, or all the initial blocks have run out), and ended the simulation for you. Now that the simulation is over, exit ncsim.

```
ncsim> exit
```

and ncsim will return you to your shell. It's worth taking a look around now at what files have been created by the tools; you should see INCA_libs, which contains all the compiled files (it's the equivalent of the work directory from ModelSim), ncelab.log (as it sounds), ncsim.key (a log of commands you ran), ncsim.log (as it sounds), ncvlog.log (as it sounds), and waves.shm (a directory containing waveforms for SimVision to use).

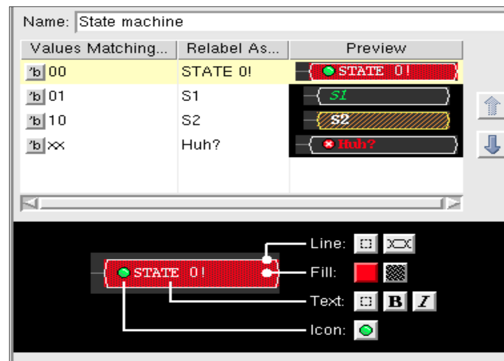
If everything went well, this is great; we wouldn't need a debugger. Sadly, things don't always go well, so we have a wonderful waveform viewer / integrated debugger called SimVision. To start SimVision, run:

```
simvision waves.shm
```

SimVision will pop up a main window called the Design Browser. On the left is a tree of modules in your design; on the right, the signals within any given selected module. Click the + symbol next to the top-level module, and then click on your instantiation of the FSM module. Observe that the signals within it appear on the right side. We want to look at all of them, so right-click on the FSM module instantiation and click "Send to Waveform View". (Side note: SimVision has a lot of useful features other than those that we'll cover in this tutorial... play with them while you go! You might discover things that can save you hours later.)

SimVision should now have popped up a box called the Waveform Viewer. Play with it some; things you might find useful are the zoom controls on the right, and knowing that you can control-click and drag to zoom. There are too many features to cover here, but one that you might find immensely useful later is the

Mnemonic Map. The Mnemonic Map lets you view arbitrary signals as descriptions that make sense to humans, and colorize signals depending on their contents. To use it, click on state[1:0], go to the Edit menu, select Create, and select Mnemonic Map.... Take a look at the example maps present, and then click the new map button ( ). Name the new map "State machine", and click on the first line in the list. Use the new mapping button ( ), and the settings boxes to make your settings look like this (you can be more creative, or lazier, if you feel get the gist of it):



Select the state[1:0] signal in the Waveform Viewer, then press Apply To Selected Signals in the Mnemonic Maps box. (Do this for nextstate, too.)

Finally, you might want to save your SimVision settings if you want to analyze a new version of the signals later. You can save your current SimVision state by going to the File menu, and pressing Save Command Script.... In that box, SimVision gives you a command to use to load this command file on startup; next time you want to restore your settings, use that command when you start SimVision.

Part 2: Register File

In the second half of this practice lab, you will construct a register file in SystemVerilog. Although this lab is for practice, and we are not requiring you to hand in your code, the module that you write in this section will be useful to you in **Lab 2** if you write it properly.

Requirements

The register file is a unit in the processor which supplies the functional units (for example, the ALU) with operand values and stores the results of computation for subsequent use. Modern instruction sets typically supply the programmer with 8~32 architecturally-visible registers for integer computation. In this lab, you will implement a register file that is suitable for executing one MIPS instruction per cycle. To support one instruction per cycle, the register file must allow two concurrent reads and one write per cycle because a MIPS instruction can require up to two input operands and can produce one result value.

Your register file should contain **32 registers**, each of which holds **32 bits**. It should have have the following input and output ports:

- Inputs: Two 5-bit source register numbers (one for each read port), one 5-bit destination register number (for the write port), one 32-bit wide data port for writes, one write enable signal, clock, and reset.
- Outputs: Two 32-bit register values, one for each of the read ports.

The register file should behave as follows:

- Writes should take effect synchronously on the rising edge of the clock and only when write enable is also asserted (active high).

- The register file read port should output the value of the selected register combinatorially.
- The output of the register file read port should change after a rising clock edge if the selected register was modified by a write on the clock edge.
- Reading register zero ($0) should always return (combinatorially) the value zero.
- Writing register zero has no effect.
- If reset is asserted on a rising clock edge, the contents of the register file should be zero'ed.

Implementation and Synthesis

Go ahead and write a module RF with the specification given above, and build a testbench for your register file. Ensure that you test all reasonable cases (e.g., read a register while it is being updated; read the same register with both read ports at the same time; reset the register file and ensure that all registers read zero).

We do not require any deliverables from this lab. However, the TA's will be very willing to assist you with any difficulties that you have. Make sure that you can implement the register file correctly before moving onto the more complex challenges in Lab 2!

Optional Tool: VCS

Aside from NC-Verilog, vcs is another tool by Synopsys. Its flow is very similar to ModelSim, which you may have used before. Once you have both a testbench and the FSM, compile them. To do so, run:

```
vcs -sverilog +lint=all +v2k FSM.sv FSM_tb.sv
```

Hopefully, the output contains no errors or warnings. If not, go back and fix the warnings (or errors!) now. After the testbench and FSM have been compiled, simply run to begin the simulation:

```
./simv
```