Verilog and ModelSim Introduction

Ben Heard

January 30, 2022

Abstract

For this lab you're going to walk through some steps to use Modelsim to simulate a simple design for a 2:1 multiplexer (mux). You'll also create a small design of your own and simulate it. Along the way you'll learn a little about implementing designs structurally using Verilog and more about Git for how we'll share content this semester.

1 Getting the Lab Content

There is a bit of content that I've placed in your repositories. Note, however, that "in your repository" means that I've put it in GitHub. You won't automatically see the new content in your directory (where you cloned the repository for lab 1). You'll need to tell GitHub that you want the new materials.

In the first lab you cloned the repository. I recommended placing it in an ece310 folder and you may or may not have done that. Change into the directory where you cloned your repository and then change into the repository itself. For me that looks like the following. (Note that the $ represents the prompt and is not part of the command that I typed in).

```
$ cd ece310/my_unity_id
```

Now that you are in your repository you can look at the status and the log. It should reflect what you saw that last time you worked with your repository and, in this case, should show that there is nothing to commit and that your last commit had a comment related to the README.md file from lab 1.

```
$ git status
$ git log
```

To get the new material from GitHub you need to interact with GitHub and tell it that you want to **pull** any updates that have been made to the repository. This is done with the following pull command.

```
$ git pull
```

1.1 What should have shown up

1. A directory hierarchy containing labs/lab_002
2. A README.md file in that directory (inspect its contents)
3. This PDF file (in a GUI session you could launch acroread)
4. A file called modelsim.ini
5. Some Verilog files down in labs/lab_002
   (a) lab_002_parta.v
   (b) lab_002_parta_tb.v
   (c) lab_002_partb_tb.v

2 Using ModelSim to simulate a Design

With the materials for the lab we're going to first just use ModelSim to simulate a design. For this part we'll be using the lab_002_parta files. The first step is making ModelSim available in your environment.

2.1 Adding ModelSim

Modelsim is installed on the ECE Linux Lab machines and will run both from the command line and as a GUI application. For this lab we're just going to run it in the command line mode so there is no need to get a full featured GUI connection. A simpler SSH session, e. g. with PuTTY, will work fine. Issue the following command.

```
$ module load module load modelsim/2021.2
```

In addition adding ModelSim to your environment you'll need to set an environment variable so that ModelSim knows where to look for you configuration file. It is possible to have a single configuration file somewhere but, since the file is so small, it's much easier to have a file per project that you're working on. The configuration file is called modelsim.ini and one has already been placed in the labs/lab_002 folder.

Setting and environment variable is different depending on the shell that you are using. Generally, everyone should be using the BASH shell in their accounts but I'll include how to do this for both BASH and TCSH shells. You may find which of the shells you are using by entering the following command.

```
$ echo $0
```

It should be one of /bin/bash or /bin/tcsh. Depending on which shell you're using execute one of the following.

2.1.1 TCSH

```
$ setenv MODELSIM modelsim.ini
```

2.1.2 BASH

```
$ export MODELSIM=modelsim.ini
```

2.2 Creating a Verilog library

ModelSim simulations require a two step process. The first is to compile your design to an intermediate format in a Verilog library. Then, you can run the simulator picking up design content from the library. Libraries can be called anything but a default and defacto standard library to use is one called **work**. Change into the directory for lab 2 and create the work library.

```
$ cd labs/lab_002
$ vlib work
$ vmap work work
```

```
Model Technology ModelSim SE-64 vmap 10.7c Lib Mapping Util ...
vmap work work
Modifying modelsim.ini
```

That last command updates the contents of the modelsim.ini file to tell ModelSim that your work library (the one where it should look for compiled designs) is called work. If the vmap command fails it's most likely because you haven't set the MODELSIM environment variable correctly. After that, look for whether the modelsim.ini file is in your directory and whether you have write access to it.

2.3 Compiling a Design

Once you have created the work library you can compile your Verilog files into it. A recommended practice is to have a single Verilog module per file. So, it's very common to have many files. It's also a recommended practice to keep you modules and the testbench modules separate as well. Notice that there are three Verilog files provided for this lab. We'll compile the two that we need now, one at a time.

```
$ vlog lab_002_parta.v

Model Technology ModelSim SE-64 vlog 10.7c Compiler 2018.08 ...
Start time: 21:52:29 on Feb 01,2021
vlog lab_002_parta.v
-- Compiling module mux21

Top level modules:
        mux21
End time: 21:52:29 on Feb 01,2021, Elapsed time: 0:00:00
Errors: 0, Warnings: 0

$ vlog lab_002_parta_tb.v

Model Technology ModelSim SE-64 vlog 10.7c Compiler 2018.08 ...
Start time: 21:52:51 on Feb 01,2021
vlog lab_002_parta_tb.v
-- Compiling module mux21_tb
```

```
Top level modules:
        mux21_tb
End time: 21:52:51 on Feb 01,2021, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
```

This has compiled the description and a testbench for a 2:1 multiplexer into the work library. You may inspect the contents of the work/ directory but don't manually change anything there.

2.4 Simulating a Design from the Command Line

One the modules of interest are compiled you may load them into the simulator and simulate. In this first example we'll load the 2:1 mux testbench into the simulator and run it. Start by loading it into the simulator.

```
$ vsim -c mux21_tb
```

Wait, what's happening here. Why are we loading some mux21_tb into the simulator? Turns out that you don't load a file into the simulator, you load the module that you want to simulate. In this case the name of the module is mux21_tb; it has the structure for stimulating out mux21 implementation.

If you see something about failing to open a display it's because you neglected to put -c on the command line. The -c tells ModelSim to run in the terminal instead of trying to open a GUI. If you've forgotten the -c option but are running in a terminal with a GUI environment then you may see the graphical version of ModelSim open. It is possible to run simulations in the graphical tool but I recommend closing it and running from the command line for this first assignment.

Once you've loaded the module into the simulator you'll be presented with some feedback about loading the simulator and a VSIM 1> prompt like below.

```
Reading pref.tcl

# 2021.2

# vsim -c mux21_tb
```

```
# Start time: 00:01:07 on Jan 30,2022
# ** Note: (vsim-3812) Design is being optimized...
# //  ModelSim SE 2021.2 Apr 14 2021 Linux 3.10.0-1160.53.1.el7.x86_64
# //
# //  Copyright 1991-2021 Mentor Graphics Corporation
# //  All Rights Reserved.
# //
# //  ModelSim SE and its associated documentation contain trade
# //  secrets and commercial or financial information that are the property of
# //  Mentor Graphics Corporation and are privileged, confidential,
# //  and exempt from disclosure under the Freedom of Information Act,
# //  5 U.S.C. Section 552. Furthermore, this information
# //  is prohibited from disclosure under the Trade Secrets Act,
# //  18 U.S.C. Section 1905.
# //
# Loading work.mux21_tb(fast)
```

At the simulator prompt you have a lot of control over how the simulator progresses. It's possible to advance the simulation for a small amount of time, inspect the values of signals at the current time, select values to store to a log file for waveform inspection later, etc. For now we'll just tell it to run the simulation to completion and see what happens.

```
VSIM 1> run -all
```

You should see the following output. I've formatted it to look like a truth table when it prints to the display. A key point of note is the time column on the left. This is the time unit when the input stimulus and output on the right are active in the circuit.

```
#                    0: S A B | F
#                    0: ------+--
#                    0: 0 0 0 | 0
#                   10: 0 0 1 | 0
#                   20: 0 1 0 | 1
#                   30: 0 1 1 | 1
#                   40: 1 0 0 | 0
#                   50: 1 0 1 | 1
```

```
#                    60: 1 1 0 | 0
#                    70: 1 1 1 | 1
# ** Note: $stop    : lab_002_parta_tb.v(32)
#    Time: 80 ns  Iteration: 0  Instance: /mux21_tb
# Break at lab_002_parta_tb.v line 32
# Stopped at lab_002_parta_tb.v line 32
```

You quit the simulator with **quit**.

3 Structural Verilog

Verilog is a hardware description language (HDL). While it may appear similar to a number of programming languages that you have seen over the years it is not a sequential programming language. Specifically, there is no concept of program flow like reading instructions from the top of a file to the bottom of a file. The file that you are writing is creating a description of hardware and you should always think about it that way.

There are a number of different ways to write hardware descriptions in Verilog and they are broken in the behavioral descriptions and procedural descriptions. In a behavioral description you are explicitly describing the behavior of the design while in a procedural description you are describing effects that should occur in response to specific events. Even within behavioral descriptions you may describe behavior structurally or with a dataflow model.

The styles of implementation, then fall into the following.

- Behavioral
  - Structural
    - Gate-Level Structural
  - Dataflow
- Procedural

Notice that I've added a further decomposition adding that gate-level structural is a specific form of structural implementation. This is the closest style to building a circuit up on a breadboard. In a gate-level structural implementation you use the Verilog built-in primitives of and, or, not, nand, nor, etc. and wire them together to describe behavior.

3.1 Verilog Module Components

There are a few things that make up a Verilog module description. I've listed them below with whether or not they're required.

1. **Module Header (required)** - This is, at it's most basic the string **module** followed by the name of your module.
2. **Port Declarations** - Generally, a function is only useful if it is able to connect to another module. Think of the port declarations and the interface through which signals flow. Notice, however, that port declarations are not required. There are cases where the module won't interface to any other. Typically we see that when a module is a testbench; it encapsulates the stimulus for the device under test (DUT) and the instantiation of the DUT within the module.
3. **Variable Declarations** - These are additional variables that are necessary for the description. Think of them as any other wires you might need that aren't given in the port declarations.
4. **Instantiations** (required or optional if other description) - Verilog modules are able to build on each other so that you can take an existing design, encapsulate it, and add more functionality to it. To do so you instantiate the module within your design.
5. **Functional Description** (required or optional if instantiations) - Alternatively to instantiating logic you may write your own functional descriptions in any of the styles above.
6. **Terminator** - This is simply the string **endmodule**

Open the lab_002_parta.v file. You'll see each of these components. There is a module header that decribes the module as **mux21**; there are port declarations that show inputs for A, B, and S and an output F; there are variable declarations for the wires between the gates; there is a functional description using gate primitives; and there is there is the terminator.

Open the lab_002_parta_tb.v file. You'll see all but the port declarations. This is a testbench so there are no ports through which it will interface to any other module. The variable declarations are there to be able to provide stimulus to the DUT; there is an instantiation of the DUT; and there are some functional descriptions in the form of initial blocks (these happen to be procedural descriptions).

3.2 Built-in Primitives

Verilog offers some built-in primitves and that's what we'll use for this lab. The following are all of the built-in gate primitves in Verilog. Note that these are just the gates; there are other primitives that aren't relevent to this lab.

- not
- and
- nand
- or
- nor
- xor
- xnor

In order to build, put down, implement, etc. one of these gates you instantiate it. That's the Verilog term. To instantiate any module you use the following template.

<module_name> <designator> (<port_connections>);

The module name is the name of the module you want to instantiate. Many times this will be the name of your module that provides some custom functionality. In the case of the gate primitives it's the name of the gate. The designator is required to uniquely identify each module/gate that you've instanced. The port connections are which wires that you would like to connect.

When connecting wires there are two styles. The first is with positional mapping where the order of the wires is important; think order of arguments to a function in programming languages. The other is named mapping. This allows you to include both the formal name (the name of the port on the module) as well as the actual wire that you want to connect to it.

Gate level primitives only support positional mapping. And, they're always ordered such that the output bit is first followed by all of the input bits. In this way you can use the same gate name for multiple input versions. For example.

```
and U1 (f, a, b, c, d);
and U2 (g, f, e);
```

In this example there is both a 4 input AND gate that has inputs a, b, c, and d and a 2 input AND gate with inputs f and e and output g. In this example you can see the output from the first (as f) becomes an input to the second (the same f).

Note, the not primitive may only ever have a single input and single output; i.e. you'll need one for each inverter in your design.

3.3 A Simple Design

In addition to the design in the lab materials, here is another simple design of an AND-OR-INVERT (aoi) circuit.

```
module aoi22 (
  input a, b, c, d,
  output f
);

  wire and0, and1;
  wire or_out;

  and U1 ( and0, a, b );
  and U2 ( and1, c, d );
  or U3 (or_out, and0, and1);
  not U4 ( f, or_out);

  // the or and not could also
  // have been the nor gate
  // nor U3 ( f, and0, and1 )

endmodule
```

This should look familiar. However, I do want to point out a couple of things.

1. **Port Declarations** - In a port declaration list if none of input or output are provided, the last known will be used. E.g. in the module above, each of a, b, c, and d are inputs.

2. **Comments** - Comments follow the C syntax and may be one line as // or multi-line with /* */.

4 Lab 2 Exercises

4.1 Simulating the existing design

While there is nothing to submit relative to simulating the canned design provided, I highly recommend it. This is the simulatio of part A above.

4.2 Design Implementation

Implement a Verilog description of the following truth table. Start by working out the minimized equations with Kmaps. Complete a drawn out schematic with instance names and wire names. This will allow you to directly create the gate level structural design. The file lab_002_partb.v has already been started for you and contains the module header, port declarations and the terminator. Open that file and put in variable declarations for each wire and instances for each gate.

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 1: Truth Table

4.3 Simulating your design

The file lab_002_partb_tb.v contains a simple testbench that will both take your design through all input states (exhaustive) and compare the output with the expected output. If there is an error in any of the input vectors,

an ERROR message will appear in the output of the simulator. Follow the directions above to simulate the testbench. I.e.

```
$ vlog lab_002_partb.v
$ vlog lab_002_partb_tb.v
$ vsim -c lab_002_partb_tb

...
VSIM 1> run -all
```

You shouldn't need to create and map a new work library as it's already present from your earlier simulations.

4.4 What to Submit ... and How

You only need to submit your design. Take a look at your git status.

```
$ git status
```

You'll see that the file lab_002_partb.v is **modified**. This is because you've made changes and it's a file that git is aware of as being part of your repository. You'll also see a list of other files and directories that are untracked. These will include work/ and transcript and may include other files. These are transient files and don't need to be submitted.

Add your modified file to the staging area to prepare the commit.

```
$ git add lab_002_partb.v
$ git status
```

Your status should now show it as being modified but also ready to commit. Commit your changes with a relevent comment.

```
$ git commit -m "Your comment here"
```

Now that you've committed your changes you can push those changes to GitHub.

```
$ git push
```

Now that it's been committed and pushed to GitHub you may record the commit SHA in the lab 2 quiz. Remember that you may get the SHA value from either the log or the rev-parse command.

```
$ git log -1
$ git rev-parse HEAD
```

As a last step, I recommend that you remove any files that are untracked. This just keeps your working area clean and reduces clutter that you'll see in any git status. I find that the easiest way to remove untracked files is to issue the command git clean. If you do this with the recursive options then it will remove untracked files from where you are and down the directory hierarchy. You can be at the top of the repository and issue it and it will remove all untracked files.

```
$ git clean -fd
```