Dataflow Modeling with Verilog

Ben Heard

February 9, 2021

Abstract

For this lab you are going to create a couple of designs using the dataflow style of modeling in Verilog. You'll then run a predetermined testbench to demonstrate that your design works.

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. Specifically, that last updates you made for lab 2.

```
$ 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_003
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_003
   (a) lab_003_parta.v
   (b) lab_003_partb.v
   (c) lab_003_partc.v
   (d) lab_003_tb.v

2 Dataflow in 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.

2.1 Dataflow Modeling

In lab 2 you implemented a design using a gate level structure style of modeling. This should have looked a lot like working with a breadboard placing and connecting the physical gates. While this is familiar it's really not any more efficient than building the gates themselves. What we would really like to leverage Verilog for is the ability to more efficiently implement designs.

Dataflow modeling is the next level of abstraction. In dataflow modeling you essentially create a complex gate that implements functionality described by an expression. I use this long way of saying it to remind you that this is still a description of hardware and, while it may look like there are equations that read from the top to the bottom of a file, the expressions are not evaluated in order. They are, in fact, representing real hardware that is always operational.

Specifically, in dataflow modeling the expression is evaluated any time that one of its operands changes. This is different than reading expressions top down; the expressions only update what's on the left hand side (LHS) when one or more of the operands on the right hand side (RHS) changes.

There are a number of operators that will greatly simplify implementations later. For now we're just going to use the ones that represent the same functions that we have in primitives. The following operators implement gate functionality.

- & - logical AND

- | - logical OR
- ~ - bit-wise negation
- ^ - logical XOR

In combination, and with parentheses for clarity, you can create NAND, NOR, and XNOR by first performing the logical operation above and then inverting. For example, the AND of A and B would look like $A\&B$ while the NAND would look like $\sim (A\&B)$.

2.1.1 Assign Statement

In a structural implementation you instantiate existing modules to bring functionality into your design. In dataflow modeling you are specifying the flow of data through an expression to a variable. This is done with the **assign** statement. For example, the following are equal.

```
and U1 ( f, a, b, c );
```

and

```
assign f = a & b & c;
```

Instead of instantiating a gate and referencing as U1, you are able to implement the functionality directly using the assign statement. Here is a more complex example. In the example above, you may assume that a, b, c, and f are all inputs or outputs to the module. In the example below we have an intermediate wire that needs to be declared; think about the schematics that we've been drawing in class.

```
wire w1;

and U1 ( w1, a, b );
or U2 ( f, w1, c );
```

and compare that to the dataflow implementation as

```
assign f = ( a & b ) | c;
```

2.2 A Simple Design

This is the same design as provided as an example in lab 2. Instead of the structural implementation this is in a dataflow style. Note that I did retain some of the intermediate wires to make the implementation more clear.

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

  wire and0, and1;

  assign f = ~( and0 | and1 );
  assign and0 = a & b;
  assign and1 = c & d;

endmodule
```

One very important thing to notice here is that it looks like the assignment to f comes before the assignments to and0 and and1. In a programming language this is a problem because expressions are evaluated in order. In a hardware description language this is not an issue because it represents hardware that will update if any of the operands on the RHS change.

3 Lab 3 Exercises

3.1 Part A Implementation

Implement a Verilog description of the following truth table. Start by working out the minimized equations with Kmaps. Develop the minimized equations for the truth table and implement it in a dataflow style. The file lab_003_parta.v has already been started for you and contains the module header, port declarations and the terminator. Add any wire declarations that you need.

A	B	C	G
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Table 1: Part A Truth Table

3.1.1 Simulating Part A

The file lab_003_tb.v contains three independent testbenches. Open it and take a look. It shows that you may declare more than one module in a file. There is a testbench for each of part A, B, and C of the lab. Follow the directions shown to simulate the testbench. You may also need to create and map the work library if you haven't done that already. See the section at the end of this manual for a refresher on using ModelSim.

```
$ vlog lab_003_parta.v
$ vlog lab_003_tb.v
$ vsim -c lab_003_parta_tb

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

There is no self checking portion of this testbench. You will need to inspect the output to determine if the output matches the expected output from the truth table above.

3.2 Part B Implementation

Implement a Verilog description of the following truth table. Start by working out the minimized equations with Kmaps. Develop the minimized equations for the truth table and implement it in a dataflow style. The file lab_003_partb.v has already been started for you and contains the module header, port declarations and the terminator. Add any wire declarations that you need.

G	D	F
0	0	1
0	1	1
1	0	0
1	1	1

Table 2: Part B Truth Table

3.2.1 Simulating Part B

The file lab_003_tb.v contains three independent testbenches. Open it and take a look. It shows that you may declare more than one module in a file. There is a testbench for each of part A, B, and C of the lab. Follow the directions shown to simulate the testbench. You may also need to create and map the work library if you haven't done that already. See the section at the end of this manual for a refresher on using ModelSim.

```
$ vlog lab_003_partb.v
$ vlog lab_003_tb.v
$ vsim -c lab_003_partb_tb

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

Again, there is no self checking portion of this testbench. You will need to inspect the output to determine if the output matches the expected output from the truth table above.

3.3 Part C Implementation

In parts A and B you implemented small designs using a dataflow style. In part C you are going to instantiate each of them and connect them together. Notice in the truth tables that I used A, B, and C as inputs that produce G and then, in part B, I used G and D as inputs to produce F as an output. You are to use the skeleton provided for part C to instantiate both of the part A and part B modules and connect them together. The resulting schematic will look like this.

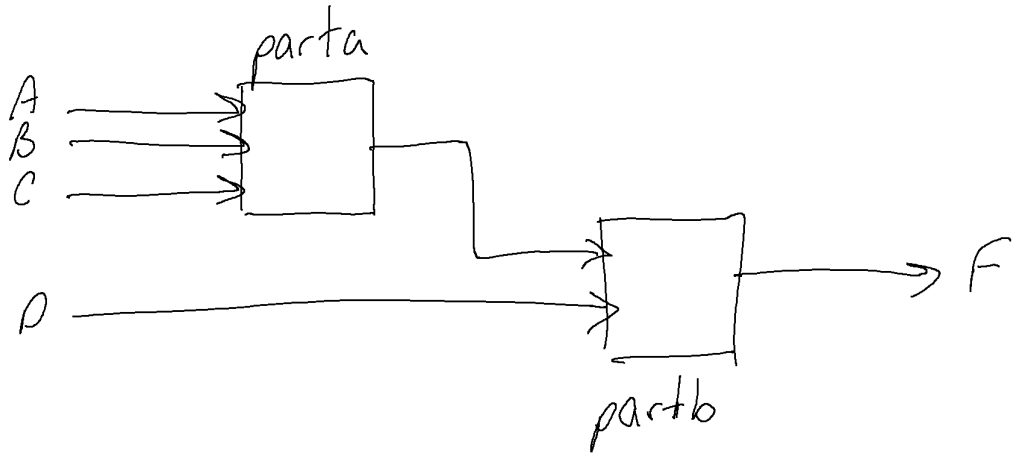

Figure 1: PartA and PartB Connected

3.3.1 Instantiating a Module

Very similar to what was done with a primitive gate, a general module instantiation is given by $< modulename >< reference > (< portconnections >)$;. While the module name and reference make sense, the port connections are little different. Recall that in the gate primitives, the output always come first followed by the inputs. In a custom made module, the order is created by the author and established in the module definition.

You may use that order or you may use a formal actual notation to explicitly connect to a specific port without knowing the port order. Take the following which is the module definition for part B.

```
module lab_002_partb (
  input G, D,
  output F
);
   ...
endmodule
```

We can instantiate this two ways. The first relies on knowing the order of the ports in the module definition. While the second only requires knowledge of the port names.

Ports in Order

```
wire connect_to_D;
wire connect_to_G;

lab_002_parta U1 (
  connect_to_G,
  connect_to_D,
  F
);
```

Ports Explicitly Connected

```
wire connect_to_D;
wire connect_to_G;

lab_002_parta U1 (
  .G( connect_to_G ),
  .F( F )
  .D( connect_to_D ),
);
```

Notice that the order is no longer important when connecting explicitly. NOTE: you may not mix the two methods, i .e. you may not connect some as named ports while trying to connect others in port order.

In the partc Verilog file, instantiate one instance each of parts A and B. Create an internal wire that will convey the output of part A to the input of part B. Connect the top level inputs and outputs to the remaining ports and simulate with the lab_003_partc_tb testbench.

Again, there is no checking of the output. You will need to verify that your design behaves as expected. This may first require that you work out what the answer should be.

3.4 What to Submit ... and How

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

```
$ git status
```

You'll see that the files you edited are now **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 files to the staging area to prepare the commit.

```
$ git add <files here>
$ git status
```

Your status should now show it as being modified but also ready to commit. Commit your changes with a relevant 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 3 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
```

4 Refresher about using ModelSim

4.1 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.

```
$ add modelsim10.7c

MentorGraphics Modelsim 10.7c
------------------------------
Available for 64 bit Linux


Use the following command(s) to start

        vsim &                  Visual Simulator


Application(s) Maintained by ECE,
send problem reports to ecehelp@ncsu.edu

add.sh: Type exit to return to previous shell instance.
```

There are newer versions available but this version will work for our purposes. 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 provided for you.

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.

4.1.1 TCSH

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

4.1.2 BASH

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

4.2 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 where you want to create the library.

```
$ 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.

4.3 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 <dsn_file>.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 <design>

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

$ vlog <tb_file>.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 <tb_file>

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

Note that the <dsn_file> is just a reference. Whichever design you compile in these steps would appear.

4.4 Simulating a Design from the Command Line

One the modules of interest are compiled you may load them into the simulator and simulate. Start by loading the design into the simulator.

```
$ vsim -c <tb_module>
```

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.

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

# 10.7c

# vsim -c <tb_module>
# Start time: 21:51:03 on Feb 01,2021
# ** Note: (vsim-3812) Design is being optimized...
# //  ModelSim SE-64 10.7c Aug 17 2018Linux 3.10.0-1127.18. ...
# //
# //  Copyright 1991-2018 Mentor Graphics Corporation
# //  All Rights Reserved.
# //
# //  ModelSim SE-64 and its associated documentation conta ...
# //  secrets and commercial or financial information that  ...
# //  Mentor Graphics Corporation and are privileged, confi ...
# //  and exempt from disclosure under the Freedom of Infor ...
# //  5 U.S.C. Section 552. Furthermore, this information
# //  is prohibited from disclosure under the Trade Secrets ...
# //  18 U.S.C. Section 1905.
# //
# Loading work.<tb_module>(fast)
VSIM 1>
```

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 quit the simulator with **quit**.

4.5 Verilog Module Components Refresher

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**