Lab Three: Introduction to Verilog

Ben Smith

***Abstract*—Basic properties of the Verilog and System Verilog hardware descriptive languages are described in this document. The focus is on Behavioral modeling, automated verification and modular design.**

I. Introduction

THIS lab will introduce text based design entry in particular, Verilog's behavioral modeling ability. It is a powerful tool that allows the digital designer to abstract themselves from the burdens of structural modeling. Text based design entry is the Industry standard for digital design. The two most prevalent languages are Verilog and VHDL are both IEEE standards. In the previous lab we used schematic representation of gates,we created individual gate "structures" and wired them together to implement the design. This provides a great intuitive explanation of what was built on the FPGA but can become very cumbersome. Imagine creating Karnaugh maps for all 72GPIO pins, or better yet the 548 user configurable pins on the Altera's larger FPGA The Stratix [1]; this is simply unreasonable! Verilog provides a very concise way to describe the behavior of very large systems. Behavioral modeling allows the use of higher level statements like If and Case statements that can implement complex functionality in only a few lines of HDL. If you don't know what these programming constructs are, don't worry, we will explore them in this lab. The purpose of this lab is to introduce the following concepts:

- Verilog behavioral modeling
- Verilog's constant syntax
- Verilog behavioral blocks
- Testbench assertions
- Instantiate a System Verilog module
- Use a System Verilog Testbench
- Synthesize Verilog code for an FPGA

A. Included Screencasts

1) TIME - Some video
2) TIME - Some video
3) TIME - Some video
4) TIME - Some video

B. Verilog Design Entry

Verilog is a powerful way to describe circuits. Logic diagrams like those being used in lecture can become cumbersome in large designs. "Text based design entry" can be less prone to error because it is easier to track differences in large designs with a diff tool than a sprawling schematic. Verilog is a text based hardware descriptive language the begun being used in ASIC(Application Specific Integrated Circuit). It is now the language of choice for FPGAs(Field Programmable Gate Array) and CPLDs(Computer Programmable Logic Devices). It is important to note that if you have some programming experience Verilog is superficially similar to many programming languages. Quartus provides a comprehensive solution for testing Verilog and synthesizing it for use on their FPGAs and CPLDs. Altera offers a tutorial very similar to the one used in Lab Two for Verilog design entry instead of the graphical method. The same tutorial that was used in lab one is also written for Verilog design entry, it can be found here for the interested reader.

C. Anatomy of a Verilog Module

The "module" is at the heart of Verilog. Clever design will allow you to create a module that you can reuse many times. These labs will stress design for re-usability as it is the core of efficiency in the workplace.

```
module <ModuleName>(
  input  wire        <PortName>,
  input  wire  [7:0] <PortName>,
  output reg         <PortName>,
  output reg   [7:0] <PortName>
);

always @(<SensativityList>)
  begin
    //logic goes here
  end
endmodule
```

Listing 1: Template for System Verilog Modules

Notice the [7:0] next to the wire declarations. This is a way to declare a "parallel" bus. We are just hooking up 7 wires at once. Compare this template to the example constant adder module from the Laboratory Procedure section.

```
//|       Title: Example adder module for CSUS CPE/EEE64
//|      Author: Ben Smith
//| Description: This module will add a specified number
   to a paramterized constant.
module Adder(
input     wire  [3:0]    UserNumber,
output    reg   [7:0]    sum = 0
);

parameter constant = 4'b0000;

//| This is a verilog behavioral block that executes
   whenever UserNumber changes value
always @(UserNumber)
  begin
    sum = constant + UserNumber;
  end
endmodule
```

Listing 2: Adder from example code

D. Verilog Modularity

One of the most important features of Verilog is it's ability to reuse a design. Reusing code allows you to rapidly assemble and test new designs. The configurability of the FPGA allows a designer to rapidly prototype a design. Reusing these modules is very similar to how you would reuse code in the workplace to be more productive. You could think of this as the source libraries that would be available at the company that you might work for.

E. Instantiation of a Verilog module

At the core of modular design is the module instantiation. Think of it of plopping a piece of hardware down on a breadboard. You could make a LS7400 Verilog module and every instantiation would be another discrete device just like using a real LS7400 on your breadboard. In Verilog the module name, is the name of the module you are instantiating.

```
<Module><InstanceName>(
  .<PortName>(<Wire>),
  .<PortName>(<Reg>)
);
```

Listing 3: Template for System Verilog module instantiation

F. Parameterization of Verilog modules

A Verilog module's parameters allow a module to be reused in a number of different situations. An example would be a variable length shift register. In one application you might need a 32-bit version in another a 64-bit. Building the module in a particular manner will allow a parameter to control the length with the parameter. The parameter and its default value is specified in the Adder Module on line

```
parameter constant = 4'b0000;

```

This is the Value that will be used if the parameter is not specified in the module instantiation. An example of parameter usage when instantiating a new module is below. Anything in angle brackets is something that you will need to replace with the information form your design.

```
<Module>#(
  .<ParameterName>()
  )<InstanceName>(
  .<PortName>(<Wire>),
  .<PortName>(<Reg>)
);
```

Listing 4: Paramaterized instantiation example

Notice the addition of the #() before the instance name in the last design. We can see the adder module instantiation in the test bench follows this syntax.

```
Adder #(
  .constant(SpecifiedConstant)
)AdderDUT(
  .UserNumber(Number),
  .sum(Sum)
);
```

Listing 5: Instantiation example form the testbench

G. Test Bench for automated debugging

Verilog roughly breaks into two halves synthesizable and non-synthesizable. FPGAs synthesis cantake a very long time, using a simulator to verify individual modules can be much faster than resynthesizing the entire design. The Testbench also offers a unique ability to check expected outputs and generate test stimulus. We will use a test bench to check the provided Verilog modules are providing the desired operation in part C of the procedure. This simulation should be verified against the known truth table for the logic gate to ensure the module is accurate. Verification is a very important topic in logic design. [1]

[1]If you are particularly driven to be an expert in programmable logic I highly recommend a series of MOOC courses on debugging taught by Andreas Zeller on debugging. Both classes deal only with Python but the way of thinking is important topic. The classes are Software Testing and Software Debugging the first few videos in each series cover the important topics. Keep in mind these classes are outside the scope of this class and I only offer them because how much they helped me be a better designer.

```
localparam SpecifiedConstant = 4'b0001;

module adderTestbench();
  //|
  //| Local reg/wire declarations
  //|--------------------------------------------
  reg             SimClk = 0;
  reg      [3:0]  Number = 0;
  wire     [7:0]  Sum;

  //|
  //| DUT instantiation
  //|--------------------------------------------
  Adder #(
      .constant(SpecifiedConstant)
    )AdderDUT(
      .UserNumber(Number),
      .sum(Sum)
    );

  //|
  //| Generate simulation clock
  //|--------------------------------------------
  always
    #1 SimClk = !SimClk;

  //|
  //| Main testbench logic
  //|--------------------------------------------
  always @(posedge SimClk)
    begin
      //| This assertion will list an error if not met
      assert (Sum == SpecifiedConstant + Number)
        $display("Case %d: Pass", Number);
      else
        $error("Case %d: FAIL:%d + %d /= %d", Number,
     Number, SpecifiedConstant, Sum);

      Number++;
    end
endmodule
```

Listing 6: Example testbench

This section will explore the basics of Modelsim and using a Verilog Testbench in Modelsim. The schematic representation of the first lab's logic blocks have been replaced with Verilog behavioral code. Using a simulator for single logic gates is a bit asinine but the experience gained with the example test bench will help you greatly in the future, particularly when you write your own testbench in the next lab. Start a new simulation and add the waveforms as shown in screencast 2. Figure 1 shows an example of what the wave section should look like.

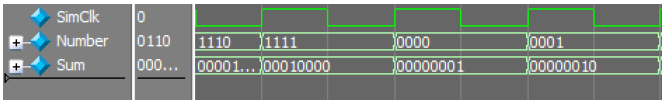

Fig. 1: Example output of testbench

Take a moment to look at the simulation transcript, it provides the states of the logic elements being tested. I prefer having the simulator give a test listing instead of reading the waveforms. This is from the $display() lines in the testbench. listing the outputs can be a very powerful debugging tool. I typically use the $assert() statement, which will be explored below, which can execute two different blocks of code based on a logical test and alert you when an unexpected result is produced.

```
# Case  0: Pass
# Case  1: Pass
# Case  2: Pass
# Case  3: Pass
# Case  4: Pass
# Case  5: Pass
# Case  6: Pass
# Case  7: Pass
```

```
# Case  8: Pass
# Case  9: Pass
# Case 10: Pass
# Case 11: Pass
# Case 12: Pass
# Case 13: Pass
# Case 14: Pass
# Case 15: Pass
```

Listing 7: Example output of Testbench

Verification is more than half the battle when working with Verilog. Many times a design will work Mentor Graphics HDL simulator Modelsim is installed with Quartus. Modelsim is used extensively in logic design with Verilog. Fortunately Verilog offers a number of tools to make checking your code easier. The first of which is the assertion; it will run two different blocks of code depending on if a logical condition is met, It works much like an if statement that might be more familiar.

```
//| This assertion will list an error if not met
assert (Logical statement)
    begin
    //<code-for-true-case>
    end
else
    begin
    //<code-for-false-case>
    end
```

Listing 8: Template for System Verilog Modules

The same code is used to test the adder from this lab's example code. The true case is used to display the valid output of the module. The false case throws a simulation error and shows the user the case.

```
//| This assertion will list an error if not met
  assert (Sum == SpecifiedConstant + Number)
  $display("Case %d: Pass", Number);
  else
  $error("Case %d: FAIL:%d + %d /= %d", Number, Number,
     SpecifiedConstant, Sum);
```

Listing 9: Template for System Verilog Modules

The logical statement in this code block checks to see if the output of the module is equal to the sum of specified constant and Number. Many designers write the test bench from specification in advance of the Verilog module. Testing should be an integral part of Verilog development from the beginning.

All of the code from this first section is provided in source.zip. There is quite a learning curve to this part, be sure to watch the screen cast which describes the included modules. You will be assigned a constant by the lab instructor that the input number will be added to. This number should be entered as a parameter in the adder module's instantiation as is done in the test bench.

II. Laboratory Procedure

A. Signal Tap Embedded Logic analyzer

The single most valuable debugging tool in Quartus is the Signaltap Logic analyzer. The device is implemented by the on chip logic and transmits information to Quartus over the USB link. Signaltap provides the state of a signal with repsect to time. This is identical to the Simulator waveform but is recorded from the running logic on the FPGA. Once again Altera provides a comprehensive tutorial on the use of signal tap.

B. 4-bit and Constant Adder Synthesis

The second section's adder will be synthesized and loaded onto the FPGA for this section. Use your dip switches and the LED circuits from the previous labs to test the adder for expected operation. This section is included with the example code, all you need to do is change the constant and test its operation.

Laboratory Demo: Simple adder

Specification: Change the parameters of the provided module and attach the 4-bit input to an external switch. Configure signaltap to trigger on any change of the input switches, Display the input and output waveforms.

Deliverable: DE0-Nano configured with adder and Signaltap module

Process : The live signaltap window will be inspected as the student cycles through the inputs.

C. Design of Comparator

This section requires the previous sections code to be modified to add two 4-bit inputs. This will require the removal of the previous modules' parametrization and the addition of an additional input port.

Fig. 2: Example output of testbench

Laboratory Demo: Comparitor

Specification: Assign comparitor output to external LED module and use dipswitches for two four-bit inputs

Deliverable: DE0-Nano configured with Comparitor

Process : Output LEDs will be inspected for proper operation as the student cycles through the inputs

III. Lab Report

DOCUMENTATION is the most important part of an engineer's job. The sharper your writing skills the more employable you will be. Keep that in mind as you have to churn out documentation throughout college. This lab manual is written in almost textbook This particular lab report will require:

A. figures to include

- Code listing of Full Adder
- Code listing for comparator
- It's important to remember Verilog is a Hardware Descriptive Language. The synthesis actually implements the Verilog in code. Quartus contains what are called "Netlist Viewers" that show the actual implementation. take a look at how Quartus synthesized your design as is shown in screencastX. Include screen captures of the combinatorial blocks listed in your design.

B. Questions to Answer

1) Do you think you would prefer the schematic design entry method of Lab two or the text based representation of this lab? In what situations do you think Schematic entry would be better, in which situations would Text base be advantageous.

IV. Conclusion

VERILOG is an IEEE standard(1364) [2], it is pervasive in industry and can be used to develop specialized hardware in the form of ASICs or reconfigurable FPGAs. It is important to underscore the differences between Verilog and a programming language like C, Java, even Assembly. Verilog offers the ability to take parallel action. Two numbers can be multiplied at once, multiple registers can be set and cleared. Entire microprocessors can be implemented on the Nano, one of the later labs will explore Altera's softprocessor the NIOS II. The TerASIC documentation included with the DE0-Nano kit is pretty good and worth the read. It will help you get the most out of the FPGA. The CD included with the Nano will also include circuit schematics that can provide a great reference when it comes time to make one of your own.

References

[1] A. D. Overview. (2013) Stratix v device overview. [Online]. Available: http://www.altera.com/literature/hb/stratix-v/stx5_51001.pdf

[2] W. Foundation. (2013) Verilog. [Online]. Available: https://en.wikipedia.org/wiki/Verilog