CPE/EEE 64

Lab Three: Verilog and Behavioral Modeling

Instructor:
Dennis DHALQUIST

Abstract

Two input logic gates are synthesized for the Altera Cyclone IV FPGA using the Quartus IDE. The logic gates are verified using a System Verilog testbench and Mentor's Modelsim HDL simulator

COLLEGE OF ENGINEERING AND COMPUTER SCIENCE

CONTENTS

LIST OF FIGURES

LISTINGS

I. Introduction

THIS lab will introduce Verilog's behavioral modeling ability. It is a powerful tool that allows the programmer to abstract themselves from the burdons of stuctural modeling. In the previous lab we used whats called Structual modeling, we created indivual gate "structures" and wired them together to implement the design. This is the most basic use of Verilog but bmagine creating Karnaugh maps for all 72GPIO pins, or better yet the 548 user configurable pins on the Stratix [1]. This is simply unreasonable, behavioral modeling allows the use of higher level statements like If's and Cases. if you don't know what these are, don't worry, we will explore them throughly in this lab. The purpose of this lab is to introduce the following concepts:

- Verilog behavioral modeling
- Construction of adders and Comparitors
- Verilog's constant syntax
- Verilog behavioral blocks
- Testbench assertions
- Instantiate a System Verilog module
- Use a System Verilog Testbench
- Synthesize Verilog code for a FPGA

A. *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. Verilog is a text based hardware descriptive language the begun being used in ASIC(Application Specific Integrated Circuit) and now is the language of choice for FPGAs(Field Programmable Gate Array). Quartus provides a comprehensive solution for testing Verilog and synthesyzing it for use on a FPGA.

B. *Anatomy of a Verilog Module*

C. *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 ability to rapidly prototype a design is one the biggest advantages of the FPGA. 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. A Verilog module

D. *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, ¡Module¿ in the listing below, is the name of the module you are instantiating.

```
<Module><InstanceName>(
    .Port1(<Wire>),
    .Port2(<Reg>)
  );

```

Listing 1. Module instantiation template

E. *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 14.s

```
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(ParameterValue)
  )<InstanceName>(
    .Port1(<Wire>),
    .Port2(<Reg>)
  );

```

Listing 2. Module instantiation template with parameterization

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 3. Module instantiation from Adder test bench

F. 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.

G. Included Screencasts

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

II. Laboratory Procedure

A. Verilog 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 II-A shows an example of what the wave section should look like.

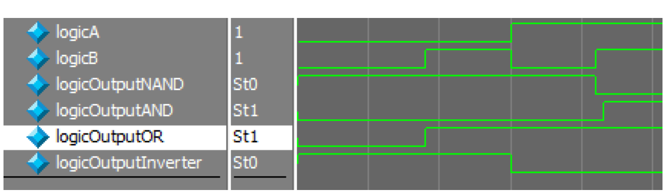

Fig. 1. Example output of testbench

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.

```
A:0 B:0 - Inverter:1 AND:0 OR:0 NAND:1
A:0 B:1 - Inverter:1 AND:0 OR:1 NAND:1
A:1 B:0 - Inverter:0 AND:0 OR:1 NAND:1
A:1 B:1 - Inverter:0 AND:1 OR:1 NAND:0
```

B. Verilog Testbench II

Verification is more than half the battle when working with Verilog. 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)
  <code-for-true-case>
else
  <code-for-false-case>
```

Listing 4. Template for System Verilog assertion

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 5. Assertion Example from test bench

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.

C. 8-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.

D. 8-bit Full Adder Synthesis

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.

E. 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.

References

[1] Altera. (2013) Stratix family overview. [Online]. Available: http://www.altera.com/devices/fpga/stratix-fpgas/stratix/stratix-gx/overview/sgx-overview.html