EC605: Computer Engineering Fundamentals Fall 2016

Vivado Verilog and Simulation Tutorial

Goals

- Introduction to Verilog design using the Vivado design environment.
- Recreate the mux from the first Lab in Vivado structural design.

Overview

This tutorial demonstrate how to use Vivado to test a Verilog design via simulation. Using the knowledge learned from the first Lab, an example 4x1 multiplexer Verilog project is created and tested.

Task 1: Verilog gate level implementation

1. Open Vivado, and create a new project. This can also be done by selecting "*New Project*" from the *File* drop down menu on the top left corner. Refer to the *Vivado Block Design Tutorial* for details.

2. Let's start by adding a simple 2x1 mux coded in gate level Verilog:

   Select *Project Manager* in the *Flow Navigator*, click on *Add Sources.*
   Select "*Add or create design sources*", and click *Next*.
   Click on "*Add Files...".*
   Navigate and select the provided source file "*Tutorial_Mux.v*".
   Click *Finish* once added.

3. Double click on the newly added file under the *Design Sources* folder. This will open the file for editing. Now take a look at what the file contains.

   If you remember from the previous lab, we can create a 2x1 mux using 2 AND gates, 1 OR gate, and 1 INV. In class you were introduced to gate primitives and how to use them in Verilog.

Task 2: Verilog structural design

1. Next add the second file *Tutorial_Mux4.v* to the current design.

2. Open this file and look at the contents.

   In class you learned about the Hierarchy of Modules. This file creates a 4x1 multiplexer by using 3 copies of the original gate level implementation of a 2x1 mux found in *Tutorial_Mux.v*. If the files are added to the same project, all modules created in these files are available to use. A common practice is to have each module in a separate file, with the file name the same as the module name.

   As a reminder, each instance of a module lives permanently in hardware and needs to be declared with a unique identifier for the compile to function correctly. This is why each declaration of "Gate_Level_MUX" is called with a unique name (mux1, mux2, mux3).

   Note that in the provided design, each module instance contains a list of the entities (wires, regs or ports), which connect to each of the ports of the module instance. Verilog provides the ability to

specify both the instance port name and the outside entity (e.g. wire) name when making connections. This makes the order in which the connections are listed irrelevant, and helps to visually represent the connection. For example:

```
Gate_Level_MUX mux2(.A(C), .B(D), .sel(sel[0]), .Y(w2));
```

The module *Gate_Level_MUX* has input entities *A*, *B*, *sel*, and an output *Y*. We can see with this declaration that our input *C* is connected to entity *A*, input D is connected to entity *B*, and the output of the 2x1 mux, *Y,* is connected to our wire *w2*.

```
.internal_parameter(outside_connection)
```

Task 3: Verilog Testbench

The last file in our tutorial contains our Verilog test bench to verify functionality of the 4x1 multiplexer from the previous task.

1. Lets add the test bench file to the project:

   Under *Project Manager* in the *Flow Navigator*, click on *Add Sources.*
   Select "*Add or create simulation sources"*.
   Select "*Add Files..."*.
   Navigate and select the provided source file "*Tutorial_TestBench.v*".
   Click *Finish* once added.

2. Open the test bench file by double clicking on it.

   Just like in the 4x1 implementation we first start with declaring a copy of the Mux4 to test. Outputs of this module are connected to the wire data type, and inputs are connected with the reg data type. This is important because our test bench will change the inputs of the system and monitor the outputs on a waveform.

   The *initial* block is the main method, which is used to perform sequential statements in a test bench. Note: initial blocks are not synthesizable, and can only be used when running a test bench. The statements in an initial block are run in order, and can be delayed using the *#Number* notation. These delays are important to show how the system will function over multiple input changes.

```
A = 0;
B = 1;
C = 1;
D = 0;

#5 sel = 0;
#5 sel = sel + 1;
#5 sel = sel + 1;
#5 sel = sel + 1;
```

   The first four lines are run in order, and since there is no added delay, they occur right away in one

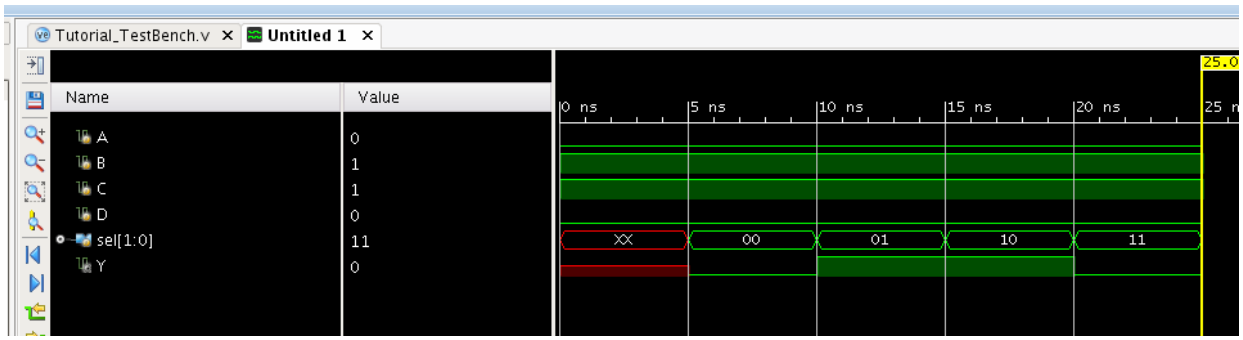
time step. On the waveform, all the four inputs, A, B, C, D will start with an initial value.

The following four lines change the "*sel*" bus after 5 time units each. On the waveform, you will see that every 5 nanoseconds step, the "*sel*" reg is changed and the output "*Y*" signal changes accordingly. However, since the initial condition of the "*sel*" was not declared, its starting value will be set to "*xx*", which means that the signal has an unknown condition. The use of the X and Z output is part of how Verilog test benches distinguish between unknown and unconnected wire states, respectively. In this class we will mostly focus on known 0 or 1 states, and try to ignore/remove any instances of the X and Z output.

3. Run the test bench:

   First verify that the test bench module is currently selected as the top module to run.
   Right Click on "*Structural_testbench*" module under *Simulation Sources* → sim_1.
   Choose "*Set as Top*".
   In the *Flow Navigator* menu, under *Simulation*, select *Run Simulation* → *Run Behavioral Simulation.*

   This will open up the test bench file and a waveform. If it does not open, you may need to click on the "*Untitled 1*" tab to view the waveform.



   You may use the icons on the left side bar to zoom in and out in the waveform. You may also scroll right and left, and you can open the simulation waveform in a separate window.

   Make sure that you understand how the waveform reflects the testbench, and that it clearly shows that the provided design functions correctly.

Task 3: Verilog Procedural Design

1. Add the last tutorial file into our project "*Tutorial_Mux4_Alt.v*" as a design source.

2. Open this file and look at the contents.

   The function of this module is to create a 4x1 mux using procedural Verilog code. This means that instead of specifying the individual gates, we are specifying how the inputs should react to produce a specific output.

3. Lets check to see if both versions of our 4x1 mux pass the provided test bench.

Add a new instance of the new module into our test bench as follows:

```
wire Y2;
Tutorial_Mux4_Alt uut2(.A(A), .B(B), .C(C), .D(D), .sel(sel), .Y(Y2));
```

Note: Make sure to specify the name of the new module "*Tutorial_Mux4_Alt*" and a new object name "*uut2*". Notice that the inputs *A*, *B*, *C*, *D*, and *sel* are the same for the new module, but our output has changed to *Y2*. This way both modules receive the same test bench produced inputs, and we can compare the outputs separately.

4. Re-run the simulation with the newly included module and check the waveform output.

   The new waveform should have an output signal for both *Y* and *Y2*. Both of these wires should produce identical results meaning both versions have an identical behavior with the provided inputs.