Lab 1: Introduction to Verilog HDL and the Vivado ISE

Introduction

In this lab simple circuits will be designed by programming a field-programmable gate array (FPGA). At the end of the lab an understanding of the process of programming a programmable logic device (PLD) should be attained, as well as an understanding of the advantages of such an approach over using discrete components.

1 Prelab

1.1. Read the material provided in the supplements (Sections 3, 4, 5.)

1.2. What is a half adder? Explain in terms of input and output signals.

1.3. Using the schematics of the board (Figure 1), if the FPGA sends a logic 1 to one of the LEDs (1-4) would it turn on or off? What about one of the RGB LEDs? (LED0_R, LED0_G, LED0_B)

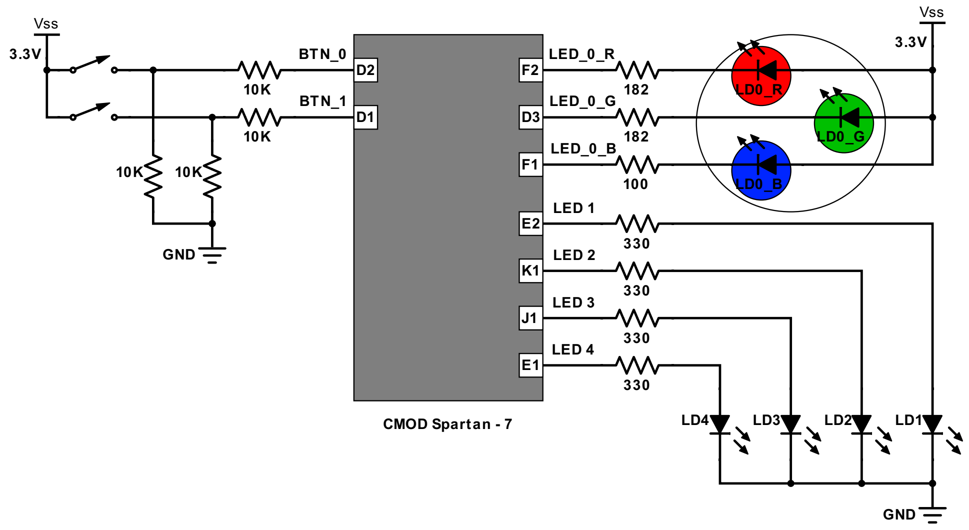

Figure 1: CMOD Spartan 7 LEDs and Buttons

2 Lab

2.1. Write a gate level Verilog program to implement the half adder circuit in Figure 2.

2.2. Assign the two inputs (X and Y) to be the two push buttons on the board and the two outputs (S and C) to be two of the LEDs on the board. **Before you continue, ask the TA to check your assignment.**

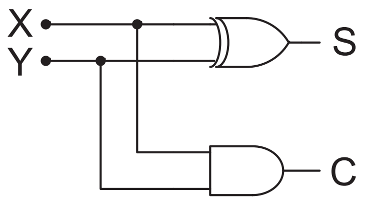

Figure 2: Half Adder Circuit

2.3. Perform both functional (Behavioral) and timing (Post-Implementation) simulations. *The simulations can be validated later with the Logic Analyzer, or in this case with a simple push-button test* **Print** your waveforms. **Are the two waveforms the same? Discuss.**

2.4. Write a Verilog program to implement the same half adder circuit using dataflow modeling and simulate your circuit again. **Note: you don't have to specify your waveforms as you have already saved them in a file.**

2.5. Program your PLD (be sure to have the correct device selected) to implement your circuit and verify that it works. **Ask the TA to initial your lab book once you have it working.**

2.6. Use vectors to describe the inputs and outputs to a half-adder. Again, use either gate level or dataflow programming (user choice) to describe the circuit. Compile code, verify simulations, and upload the FPGA board.

3 Supplement: Verilog

An introduction to Verilog HDL is discussed in the sections to follow.

3.1. In order to write a Verilog HDL description of any circuit you will need to write a module, which is the fundamental descriptive unit in Verilog. A module is a set of text describing your circuit and is enclosed by the key words `module` and `endmodule`.

3.2. As the program describes a physical circuit, you will need to specify the inputs, outputs, behavior of the circuit (gates, etc.), wires, and how each of the gates are wired. To accomplish this you need the keywords input, output, and wire to define the inputs, outputs, and the wiring between the gates, respectively.

3.3. There are multiple ways to model a circuit:

- Gate level modeling
- Dataflow modeling
- Behavioral modeling
- A combination of the above

3.4. A simple program modeling a circuit (Figure 3) at the gate level is described below as Listing 1. The same program is then modeled again in Listing 2, using dataflow.

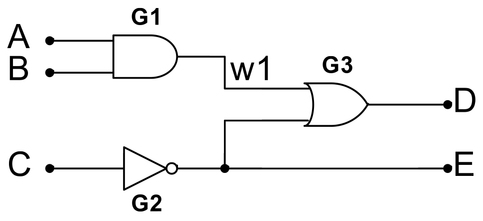

Figure 3: Simple Modeled Circuit

Listing 1: Simple Program in Verilog Modeling a Circuit at the Gate Level

```
module simple_circuit(output D,E, input A,B,C);
        wire w1;          // Creating a virtual wire
        and  G1(w1,A,B);  // Creates AND gate with output w1 and inputs A and B
        not  G2(E,C);     // Creates NOT gate output E and input C
        or   G3(D,w1,E);  // Creates OR gate with output D and inputs w1 and E
endmodule
```

Listing 2: Simple Program in Verilog Modeling a Circuit using Dataflow

```
module simple_circuit(output D,E, input A,B,C);
        wire   w1;          // Creating a virtual wire
        assign w1 = A & B;  // Creates AND gate with output w1 and inputs A and B
        assign E = ~C;      // Creates NOT gate output E and input C
        assign D = w1 | E;  // Creates OR gate with output D and inputs w1 and E
endmodule
```

3.5. As seen above, the outputs come first in the port list followed by the inputs.

- Single line comments begin with //
- Multi-line comments are enclosed by /*...*/
- Verilog is case sensitive

Code Commenting and Formatting: Clear commenting and formatting within your code is very important! Well written code is easy to look at and even easier to debug/fix later. **Hint:** The TAs will help you more if your code is not well written!! The following are tips for well written, user friendly code:

- Single line comments should be clear and concise, describing each line's function/use.
- Multi-line comment blocks should be used for describing complex sections of code, possibly separating blocks of code, describing the ".v" file with useful information (Author, Function, School, Class, etc.).
- Use formatting to your advantage! Space variables and comments over so they are in even columns. (see example code (Listing 3) lines 6-11 & 17-22)
- Use tabbing to your advantage! Tab over when in blocks of code, such as modules, if/else/case statements, etc.

Listing 3: Both bad & good code formatting and commenting

```
 1 /* This type of comment block is quick to make, but isn't very pretty...
 2    Things to notice about the bad_format_and_comment module:
 3    -no comment for module line                          -each term within the module is not tabbed over
 4    -single space after each type, before name           -single space name, before comment
 5    -general comments for each line, do not comment on what purpose the line serves                  */
 6 module bad_format_and_comments(output D,E, input A,B,C);
 7 wire w1; // Create wire
 8 and G1(w1,A,B); // AND gate
 9 not G2(E,C); // NOT gate
10 or G3(D,w1,E); // OR gate
11 endmodule
12 
13 //////////////////////////////////////////////////////////////////////////////////////////////////////
14 // This type of large comment block looks pretty, but takes more typing and time...
15 // Notice the same code from above formatted and commented well.
16 //////////////////////////////////////////////////////////////////////////////////////////////////////
17 module good_fmt_and_commts(output D,E, input A,B,C); // This comment outlines the purpose of module
18         wire w1;          // Creating a virtual wire
19         and  G1(w1,A,B);  // Creates AND gate with output w1 and inputs A and B
20         not  G2(E,C);     // Creates NOT gate output E and input C
21         or   G3(D,w1,E);  // Creates OR gate with output D and inputs w1 and E
22 endmodule
```

3.1. You can use identities describing multiple bits known as vectors. Given an identifier `[7:0] Input_X` you can assign values by:

```
assign [7:0] Input_X = 8'b00001111;
```

where the `8'b` specifies that we are defining an 8-bit binary number and `00001111` is that 8-bit binary number. You can also assign parts of the number as:

```
assign [2:0] Input_X = 3'b101;
```

Which assigns only to bits `2-0` of Input_X.

4 Supplement: Writing Bad Code

A discussion of poor and incorrect coding practices will follow, with emphasis on how to correct the mistakes. For purposes of discussion the ill-written code in Listing 4 will be discussed.

Listing 4: Example of Bad Code

```
1 module Bad code(ABCD,EF);
2         wire1; Define Wire
3         assign w1=C // Assign W1 to input C;
4         asign D = A & B; // Assign D as OR gate with C, A
5         aign E = B | C; /XOR gate
6         assign F = A ^ 1w; // AND gate
7 end modules
```

4.1. The first line of Bad Code (Listing 4) can either be revised to Listing 5 or Listing 6.

Listing 5: Revision I of Bad Code (Line 1)

```
module BadCode(input A,B,C, output D,E,F);
```

Listing 6: Revision II of Bad Code (Line 1)

```
module BadCode(A,B,C,D,E,F);
        input A,B,C;
        output D,E,F;
```

4.2. The first line of Bad Code (Listing 4) is necessary when using input and output variables inside a module. Wires (which are neither inputs, nor outputs) should be defined as seen in Listing 7:

Listing 7: Definition of Wires

```
wire w1;
```

4.3. Whenever a variable (wire) is being given a value, The `assign` keyword must precede the variable being set and the associated logic. Revision of Bad Code (Listing 4) lines 3-6 can be seen in Listing 8.

Listing 8: Proper Usage of Assign Operator

```
assign w1 = C;      //Assign w1 as C;
assign D  = A & B;  // Assign D as AND of A, B
assign E  = C | B;  // Assign E as OR of B, C
assign F  = A ^ w1; // Assign F as XOR of A, w1
```

4.4. The module should be closed with the `endmodule` keyword. A proper revision of the code from Listing 4 is as follows in Listing 9:

Listing 9: Corrected Implementation of Bad Code

```
module BadCode(input A, B, C, output D, E, F);
        wire   w1;          // Define Wire
        assign w1 = C;      // Assign W1 to input C
        assign D  = A & B;  // Assign D as AND of A, B
        assign E  = B | C;  // Assign E as OR of B, C
        assign F  = A ^ w1; // Assign F as XOR of A, w1
endmodule
```

5 Supplement: Vivado

In order to implement the circuits that you will design on the FPGA, there are a few key steps.

5.1 Open Vivado and Start a new Project

5.1.1. Open Vivado FPGA Programming software.

5.1.2. Select "Quick Start" then choose "Create Project" , then "Next".

5.1.3. Set the name of the project. Make the name descriptive, not just lab name (e.g. *Lab_1*) but *Lab_1_HalfAdder*.

5.1.4. Set the "Working Directory." This should be a folder to contain all of your projects. **Make sure to place this folder in an area that you own.** (USB flash drive, NMT geek drive, etc.) A sub-folder of the same name as the project will be created for the project to work within. Select "Next" for the next menu.

5.1.5. Select "RTL Project" then choose "Next".

5.1.6. Add a source file. Select "Create File" then ensure that Verilog code is specified, and name your module file. Once created, ensure that "Target language" and "Simulator language" are set to Verilog. Select "Next" to continue.

5.1.7. Select "Next" to continue to board selection after the constraints page.

5.1.8. Specify the device that you are using. The FPGA we are using is as follows:

- Category: **All**
- Family: **Spartan-7**
- Package: **csga225**
- Speed: **-1**
- Part: **xc7s25csga225-1** (Should be 3rd choice)

5.1.9. Select the board, then Select "Next" to see a project summary, followed by "Finish"

5.2 Writing the Code

5.2.1. Once open, Vivado will open a "Define Module" prompt. Feel free to add any inputs/outputs now, or close to add them later. This Prompt will just pre-fill the new module with inputs/outputs in advance.

5.2.2. Select the newly created module in Project Manager → Sources → Design Sources → "Module_name.v". This opens your file and this is where you will edit the file to create the desired circuit.

Now you are ready to type in your program.

5.3 Simulating the Designed Circuit

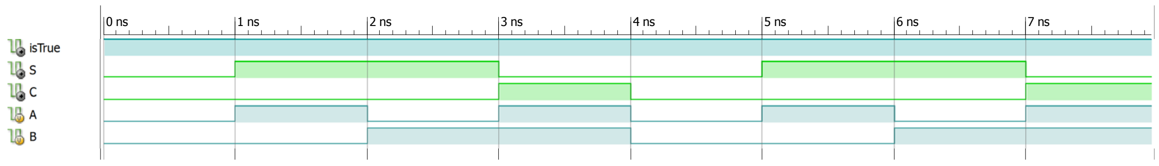

Figure 4: Functional Simulation

Simulation is a synthetic test of functionality such as seen in Figure 4. It is used to validate portions of designs prior to moving them to hardware, which can be a costly and time consuming process. Instead, first a design is validated using models and simulations, then final development is carried out on discrete hardware.

There are two types of modes that we are concerned with:

1) Behavioral (Functional): We are not worried about any delays and we are just looking to make sure that the circuit is working correctly. (correct input/output logic)

2) Timing (Post-*): Simulating the circuit and how it would possibly run on actual hardware (FPGAs). This includes any delays from gates in the circuit.

First perform the functional simulation to verify correct logic implementation, and then perform the timing simulation to look at the timing of the circuits' signals.

When simulating the circuit you need to figure out what waveforms are needed for the inputs that will ensure that your circuit works properly. If you have a simple circuit, you can easily test all the possibilities. As the circuits get more and more complicated you will need to figure out a scheme to verify their operation. In simulating your circuit there are four main steps, and they are as follows:

- Create a "Verilog Test Fixture/file"
- Specify a waveform for each input.
- Run the simulation to generate the output for verification.
- Verify results are consistent with what is expected.

These steps can be achieved according to the following steps:

5.3.1. Right-Click in the "Sources", and choose "Add Source" (the "+" icon).

5.3.2. Choose "Add or Create simulation sources", then "Next".

5.3.3. Choose "Create File", ensure Verilog is selected, and specify a file name (ex. *Test_Half_Adder*) Select "Finish" to continue.

5.3.4. Once created, Vivado will open a "Define Module" prompt. Feel free to add any inputs/outputs now, or close to add them later. This Prompt will just pre−fill the new module with inputs/outputs in advance.

5.3.5. Open the test file, note the first line is a time-scale shown as *length of time unit/ simulator precision*. Modify the first line of code to reflect a more accurate operational frequency. Such an example is seen in Listing 10.

Listing 10: Verilog Time Scale

```
‘timescale 10ns / 1ps
```

5.3.6. The "Test Fixture/file" is a full Verilog environment, but the simplest way to use it is to specify the input values, and how long to wait before changing them (in integer multiples of the major time unit) inside an always block, seen below in Listing 11.

Listing 11: Brute-Force Assignment within Verilog Test Fixture

```
module Test_Simulation_half_adder();
    // Inputs //////////////////////////////////////////////////////////////////
    reg A;              // Input Bit A
    reg B;              // Input Bit B

    // Outputs /////////////////////////////////////////////////////////////////
    wire S;             // Reported Input  = Sum of (A + B)
    wire C;             // Reported Output = Carry of (A + B)

    // Instantiate the Unit(Module) Under Test (UUT) ///////////////////////////
    Half_Adder uut (    // This is in the form: ModuleToBeTested uut(
        .A(A),          // These lines are in the form: .X(Y),
        .B(B),          //   where the first part [.X] is the input variable of
        .S(S),          //   the uut, and the second part [(Y)] is the input
        .C(C)           //   variable for the TestModule. (in this case the
    );                  //   same variables were used in each.

    // Initial Conditions //////////////////////////////////////////////////////
    initial begin       // Set initial conditions for input variables
        A = 0;          // Set input variable A to 0
        B = 0;          // Set input variable B to 0
    end

    // Generation of Truth-Table ///////////////////////////////////////////////
    always
        #1 A = ~A;      // Toggle A - Every 1 Time-Unit (10nS * 1 = 10nS)

    always
        #2 B = ~B;      // Toggle B - Every 2 Time-Units (10nS * 2 = 20nS)
endmodule
```

The code sets A and B to 0, waits 1 time-unit, sets A to 1, waits another 1 time-unit, sets A to 0 and B to 1, then waits another 1 time-unit before setting A and B to 1.

5.3.7. Of additional note is a simple way to generate a clock signal. You can append the code shown in Listing 12 to your “Test Fixture/file” to generate a clock with period of two time-units.

Listing 12: Simple Clock Source in a Verilog Test Fixture

```
always
        #1 clk = ~clk;
```

5.3.8. Save the test file and select “Simulation” → “Run Simulation” from the far left “Flow Navigator”. **Note:** The Drop-Down menu is where you will select either “Behavioral” or “Post-*”, specifying the type of simulation to be run.

5.3.9. Vivado will then display your simulation results. It is likely the simulation will need to be zoomed in to fully analyze the results. To display the simulation results, click the left-pointing arrow in order to see the beginning of your simulation, then click the magnifying glass with the “–” sign, followed by your waveforms window to zoom out.

5.4 Pin Assignment

You need to specify which inputs and outputs from your FPGA chip connect to the inputs and outputs of the physical FPGA board. Some pins have already been internally wired to the LEDs and push buttons on the board. A list of those pins is provided in Table 1 (at the end of the document).

5.4.1. Select “RTL Analysis” from the far left “Flow Navigator”, then “Open Elaborated Design”.

5.4.2. Select the “I/O ports ” tab at the bottom of the console, or if it is not there then from “Window” → “I/O Ports”.

5.4.3. Expand the Ports Sections to view your inputs/outputs

5.4.4. Set desired pins according to Tables 1 and 2 in the “Package Pin” column, then set the “I/O Std” to **LVCMOS18** instead of default (LVCMOS18). **Warning, if this step is skipped your code will not compile later!**

5.4.5. Repeat previous step until all input/output pins are assigned.

5.4.6. Save Constraints (CTRL + S)

5.5 Compiling

5.5.1. Select “Synthesis” from the “Flow Navigator”, then “Run Synthesis”. Once this is done, the top right prompt will show what process is in progress.

5.5.2. After completion you will have the option to view reports or continue to Run Implementation.

5.5.3. If there are no errors, then your program’s syntax is correct. **This does not mean that your program will do what you want; you may have some logic errors.**

5.5.4. Select “Implementation”, from the “Flow Navigator”, then “Run Implementation”. Again, the top right prompt will show what is currently running.

5.5.5. Again, if there are no errors, then your program’s syntax is correct.

5.5.6. Later if you wish, you can skip to selecting “Run Implementation” and Vivado will also run the Synthesis process beforehand.

5.6 Programming the FPGA

The final step is to program the FPGA with your designed circuit. Be sure the correct device is selected.

5.6.1. Select "Program and Debug" from the "Flow Navigator', then "Generate Bitstream". This will create a file that is readable by the FPGA, and can be uploaded and run to implement your design.

5.6.2. Connect your Spartan 7 FPGA board to the computer via USB.

5.6.3. Next, select "Open Hardware Manager", then "Open Target", then "auto-connect" to have Vivado connect to your Spartan 7 FPGA board.

5.6.4. Lastly, select "Program Device", then "**xc7s25_0**", followed by "Next" to upload your bitstream file to the FPGA.

5.6.5. After the bitstream file has been programmed to your board, the FPGA will immediately begin running the implemented code. If successful, the half adder should be running on the board and the buttons should react accordingly. Congratulations!

5.6.6. **Note:** The uploaded code is stored in volatile memory, and will be erased upon board reset or power loss. Later, we will investigate programming the flash chip on-board the Spartan 7 board, allowing programmed code to remain persistent after board reset/power loss.

Table 1: CMOD Spartan 7 — Feature Assignments

Special	Wire	**FPGA Pin**
Button 0	BTN0	**D2**
Button 1	BTN1	**D1**
12 MHz Clock	FPGA-CLK	**M9**
RGB LED Red	LED0	**F2**
RGB LED Green	LED0	**D3**
RGB LED Blue	LED0	**F1**
LED 1	LED1	**E2**
LED 2	LED2	**K1**
LED 3	LED3	**J1**
LED 4	LED4	**E1**

Table 2: CMOD Spartan 7 — DIP Pin Assignments

DIP Pin	**FPGA Pin**	Wire	Wire	**FPGA Pin**	DIP Pin
1	**L1**	PIO01	PIO48	**A4**	48
2	**M4**	PIO02	PIO47	**A3**	47
3	**M3**	PIO03	PIO46	**B4**	46
4	**N2**	PIO04	PIO45	**B3**	45
5	**M2**	PIO05	PIO44	**C1**	44
6	**P3**	PIO06	PIO43	**B1**	43
7	**N3**	PIO07	PIO42	**B2**	42
8	**P1**	PIO08	PIO41	**A2**	41
9	**N1**	PIO09	PIO40	**C5**	40
-	**-**	-	-	**-**	-
-	**-**	-	-	**-**	-
-	**-**	-	-	**-**	-
-	**-**	-	-	**-**	-
-	**-**	-	-	**-**	-
-	**-**	-	-	**-**	-
16	**P14**	PIO16	AIN33P	**A11**	33
			AIN33N	**A12**	
17	**P15**	PIO17	AIN32P	**A13**	32
			AIN32N	**A14**	
18	**N13**	PIO18	PIO31	**J11**	31
19	**N15**	PIO19	PIO30	**M13**	30
20	**N14**	PIO20	PIO29	**L13**	29
21	**M15**	PIO21	PIO28	**J15**	28
22	**M14**	PIO22	PIO27	**K14**	27
23	**L15**	PIO23	PIO26	**L14**	26
24	**VU**	VU	GND	**GND**	25