Supplement(s): Introduction to Xilinx ISE and Verilog

Contents

1 Introduction to the Xilinx ISE

To implement the circuits that you will design on the FPGA there are a few key steps.

1.1 Start a new Project

1.1.1. Select "File," then choose "New Project."

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

1.1.3. Set the "Working Directory." This should be a folder to contain all of your projects. A sub-folder of the same name as the project will be created for the project to work within.

1.1.4. Ensure the "Top-level source type" is set to "HDL," then Click "Next" to specify project settings.

1.1.5. Specify the device that you are using, and VHDL Standard. The device family we are using is the **Spartan6** and the device is an **XC6SLX4** in a **CPG196** package with speed class of **-2**. Additionally, it is recommended that the VHDL Standard be set to **VHDL-200X**.

1.1.6. Click "Next" to see a project summary, then Click "Finish."

1.2 Writing the Code

1.2.1. Right-Click in the "Empty View" and choose "New Source", note the icon.

1.2.2. Choose "Verilog Module" and specify a file name.

1.2.3. Click "Next," then specify inputs and outputs if known.

1.2.4. Click "Next" to see a summary, then choose "Finish."

Now you are ready to type in your program.

1.3 Compiling

1.3.1. Select "Process," then "Implement Top Module." Note the icon. It (the play icon) also shows up on the toolbar.

1.3.2. On the "Design" tab, you can click on "Design Summary/Reports" to see the results of the compilation. Additionally, as part of the "Synthesize - XST" are Schematic Views.

1.3.3. 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.

Table 1: CMOD-S6 Feature Assignments

Special	Wire	**FPGA Pin**
Button 0	BTN0	**P8**
Button 1	BTN1	**P9**
1 Hz Clock	FPGA-LFC	**N7**
8 MHz Clock	FPGA-GCLK	**N8**
LED 0	LD0	**N3**
LED 1	LD1	**P3**
LED 2	LD2	**N4**
LED 3	LD3	**P4**

1.4 Pin Assignment

You need to specify which inputs and outputs. 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.

1.4.1. Select "Tools," "PlanAhead," then "I/O Pin Planning...Pre-Synthesis."

1.4.2. Expand the Ports Sections.

1.4.3. Set Desired pin according to Tables 1,3 in the "Site" form.

1.4.4. Repeat previous step until all pins are assigned.

1.4.5. Save Constraints (CTRL + S)

1.4.6. Xilinx defaults all unused pins to weak pull-down. This is a good choice for pins not connected to anything (it reduces power and noise), but is not good for pins that may be connected to a clock input – you then have both the clock and the chip trying to drive this input. A safer choice is to define all unused pins "As float." Change this setting as follows:

1.4.6.1. Within the ISE Project Navigator window right-click "Generate Programming File" in the process pane and choose "Process Properties..."

1.4.6.2. Browse to "Configuration Options"

1.4.6.3. Specify "Unused IOB Pins" as "float"

1.5 Simulating the Designed Circuit

Simulation is a synthetic test of functionality such as seen in Figure 1. 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 the delays and we are interested to make sure that logically the

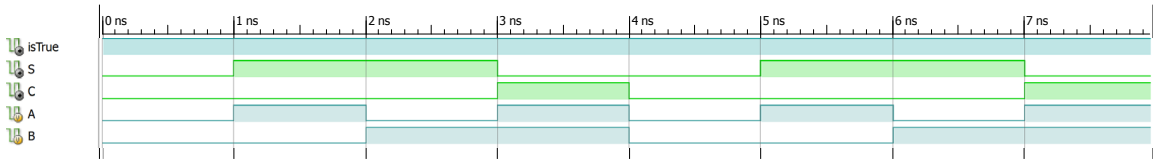

Figure 1: Functional Simulation

circuit is working. 2) Timing (Post-*): we simulated the circuit and include the delays in all the gates. First perform the functional simulation, and then perform the timing.

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

- Create a "Verilog Test Fixture."
- 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:

1.5.1. Right-Click in the "Hierarchy" and choose "New Source", note the icon.

1.5.2. Choose "Verilog Test Fixture" and specify a file name.

1.5.3. Click "Next," then specify the verilog file to be tested.

1.5.4. Click "Next" to see a summary, then choose "Finish."

1.5.5. 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 1.

Listing 1: Verilog Time Scale

```
1 `timescale 10ns / 1ps
```

1.5.6. The "Test Fixture" 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 the always block, seen below in Listing 2.

Listing 2: Brute-Force Assignment within Verilog Test Fixture

```
1 initial begin
2         sw0 = 0;
3         sw1 = 0;
4         #1;
5         sw0 = 1;
```

```
6         #2;
7         sw0 = 0;
8         sw1 = 1;
9         #2;
10        sw0 = 1;
11        sw1 = 1;
12 end
```

The code sets `sw0`, `sw1` to 0, waits 1 time-unit, then sets `sw0` to 1. It waits another 2 time-units, then sets `sw0` to 0 and `sw1` to 1. The program, then, waits another 2 time-units before setting `sw0` and `sw1` to 1.

1.5.7. Of additional note is a simple way to generate a clock signal. Append the code shown in Listing 3 to your "Test Fixture" to generate a clock with period of two time-units.

Listing 3: Simple Clock Source in a Verilog Test Fixture

```
1 always
2         #1 clk = ~clk;
```

1.5.8. Save the "Test Bench" and switch "view:" from "Implementation" to "Simulation". **Note:** The Drop-Down labeled "Behavioral" can be used to specify the type of simulation used.

1.5.9. Click on the "Test Bench" in the "Hierarchy" panel, then expand "ISim Simulator" shown in the "Processes" pane.

1.5.10. Double-Click "...Check Syntax" to validate your code.

1.5.11. Double-Click "Simulate..." to run the simulation.

1.5.12. ISim will launch with your simulation results. It is likely the simulation will need to be zoomed out to appreciate the results.

1.6 Programming the FPGA

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

1.6.1. Ensure you are in the "Implementation" view, then select your desired code.

1.6.2. Under the process pane, double-click "Implement Design."

1.6.3. Then, double-click "Generate Programming File."

1.6.4. Double-click "Configure Target Device" which will Launch iMPACT.

1.6.5. Double-click "Create PROM File."

1.6.6. Choose "Xilinx Flash/PROM", then click the green arrow.

1.6.7. Choose "Auto Select PROM."

1.6.8. Assign a File Name, and Specify the Output File Location to the project directory.

1.6.9. Select the *.bit file generated in the first step as input.

1.6.10. Do NOT add additional devices.

1.6.11. Double-Click "Generate File..."

1.6.12. Go to "Boundary Scan" by clicking on it.

1.6.13. Right-Click and choose "Add Xilinx Device"

1.6.14. Specify the *.bit file created earlier

1.6.15. Double-Click the "SPI/BPI" box and specify the *.mcs file

1.6.16. Specify **SPI PROM S25FL128S** and press "Ok."

1.6.17. Click on the Xilinx FPGA icon to program

Note: Clicking on the FLASH icon will program the device to persist after power cycling, but takes significantly longer.

1.6.18. Double-Click "Program"

2 Introduction to Verilog

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

2.1. 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`.

2.2. As the program describes a physical circuit you will need to specify the inputs, the outputs, the behavior of the circuit and how 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.

2.3. There are multiple ways to model a circuit:

- Gate level modeling
- Dataflow modeling
- Behavioral modeling
- Or a combination of the above

2.4. A simple program modeling a circuit (Figure 2) at the gate level is described below as Listings 4,5.

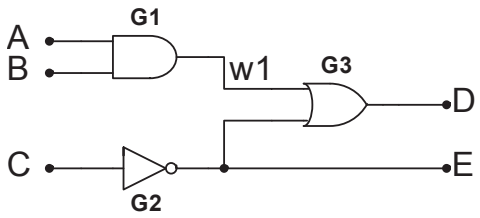

Figure 2: Simple Modeled Circuit

Listing 4: Simple Program in Verilog Modeling a Circuit at the Gate Level

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

Listing 5: Simple Program in Verilog Modeling a Circuit using Data Flow

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

2.5. 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

2.6. A simple program modeling a circuit using dataflow is provided in Listing 5.

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

```
1 assign [7:0] 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:

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

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

3 Registers and Logic

3.1 Verilog Logic Levels

Within Verilog there exist four logic levels, listed in Table 2.

Table 2: Verilog Logic Levels

Logic	Description
0	Logic Zero; False Condition
1	Logic One; True Condition
X	Unknown Logic Value
Z	High Impedance

3.2 Verilog Always and Reg Keywords

3.2.1. Behavioral modeling uses the keywords *always*.

3.2.2. Target output is a type *reg*. Unlike a wire, *reg* is updated only when a new value is assigned. In other words, it is not continuously updated as wire data types.

3.2.3. *[A]lways* may be followed by an event control expression.

3.2.4. *[A]lways* is followed by the symbol '@' which is followed by a list of variables. Each time there is a change in those variables, the *always* block is executed.

3.2.5. There is no semicolon at the end of the *always* block.

3.2.6. The list of variables are separated by logical operator or and not the bitwise OR operator.

3.2.7. Below is an example of an always block:

Listing 6: Example of an Always Block

```
1 always @(A or B)
2         //Do Stuff
```

3.3 Verilog if-else Statements

[I]f-else statements provide a means for conditional outputs based on the arguments of the if statement. An example is offered as Listing 7.

Listing 7: Example of if-else Statement

```
1 output out;
2 input s,A,B;
3 reg out;
4 if(s)
5         out = A; // if s is 1, then out is A
6 else
7         out = B; // else (s =/= 1), then out is B
```

3.4 Verilog case Statements

Case Statements provide an easy way to represent a multi-branch conditional statement.

3.4.1. The first statement that makes a match is executed.

3.4.2. Unspecified bit patterns should be treated using “default” as the keyword.

An example of a case statement is provided in Listing 8.

Listing 8: Four-to-one Line Multiplexer

```
module mux_4x1_example(
        output reg out,
        input [1:0] s,  // Select Represented by 2 bits
        input in_0, in_1, in_2, in_3);
        always @(in_0,in_1,in_2,in_3,s)
                case(s)
                        2'b00: out <= in_0; // if s is 00 then output is in_0
                        2'b01: out <= in_1; // if s is 01 then output is in_1
                        2'b10: out <= in_2; // ...
                        2'b11: out <= in_3;
                endcase
endmodule
```

4 Operators and Parameters

4.1 Parameterization

4.1.1 Macros

Listing 9: Macros in Verilog

```
`define Rst_Addr 8'hFF // Gets Expaned
assign data = `Rst_Addr; // Tic is Necessary
```

4.1.2 Parameters

Listing 10: Parameters in Verilog

```
parameter num = 8;
```

Parameters are constants, not variables.

4.2 Operators

4.2.1 Ternary Operator

Listing 11: Ternary Operator in Verilog

```
assign y = sel ? a : b;
```

If `sel` is true, `y` is assigned to `a`, otherwise it is assigned to `b`.

4.2.2 Concatenation

Listing 12: Concatenation in Verilog

```
{a, b, c}
```

Bits are concatenated using { }.

4.2.3 Comparison

Listing 13: Comparison in Verilog

```
1 if(a > b) y = a;
```

Compare `a` to `b`, if true set `y` equal to `a`. Other comparisons are listed in Listing 14.

Listing 14: Comparison Operators

```
1  >     // Greater than
2  <     // Less than
3  >=    // Greater than or equal to
4  <=    // Less than or equal to
5  ==    // Equality
6  ===   // Equality including X and Z
7  !=    // Inequality
8  !==   // Inequality including X and Z
```

4.2.4 Logical Operators

Listing 15: Logical Operators

```
1  !     // Logical negation
2  &&    // Logical and
3  ||    // Logical or
```

4.2.5 Binary Arithmetic Operators

Listing 16: Binary Arithmetic Operators

```
1  +     // Addition
2  -     // Subtraction
3  *     // Multiplication
4  /     // Division (truncated)
5  %     // Modulus
```

4.2.6 Unary Arithmetic Operators

Listing 17: Unary Arithmetic Operators

```
1  -     // Change the sign of the operand
```

4.2.7 Bitwise Operators

Listing 18: Bitwise Operators

```
1  ~     // Bitwise negation
2  &     // Bitwise AND
3  |     // Bitwise OR
4  ^     // Bitwise XOR
5  ~^    // Bitwise XNOR
6  ^~     // Bitwise XNOR (also)
```

4.2.8 Unary Reduction Operators

- Produce a single bit result by applying the operator to all the bits of the operand.

Listing 19: Unary Reduction Operators

```
1  ~     // Bitwise negation
2  &     // Bitwise AND
3  |     // Bitwise OR
4  ‘&    // Reduction NAND
5  ~|    // Reduction NOR
6  ^     // Bitwise XOR
7  ~^    // Bitwise XNOR
8  ^~     // Bitwise XNOR (also)
```

4.2.9 Shift Operators

- Left operand is shifted by the number of bit positions given by the right operand.
- Zeros are used to fill vacated bit positions.

Listing 20: Shift Operators

```
1  <<    // Logical left shift
2  >>    // Logical right shift
```

4.2.10 Precedence

Listing 21: Precedence

```
1 /* Highest Precedence */
2   !, ~
3   *, /, %
4   +, -
5   <<, >>
6   <, <=, >, >=
7   &
8   ^, ^~
9   |
10  &&
11  ||
12  ?:
13 /* Lowest Precedence */
```

5 Sequential Logic

So far the statements you have been using that are encapsulated by always were blocking type, i.e., they are executed sequentially. For this lab you will need to use a non-blocking procedural assignment. Non-blocking assignments are executed concurrently. To differentiate between the two, let's look at the following two examples:

Listing 22: Blocking Assignments in Verilog

```
1 B = A;
2 C = B + 1;
```

and

Listing 23: Non-blocking Assignments in Verilog

```
1 B <= A;
2 C <= B + 1;
```

The first case (Listing 22) is the blocking one, where the statements are executed sequentially, therefore the first statement stores `A` into `B`, and then 1 is added to `B` and stored into `C`. At the end a value of $A + 1$ is stored into `C`. On the other hand, the second case (Listing 23) is non-blocking. In this case the assignments are made using `<=` instead of a `=` sign. Non-blocking statements are executed concurrently, so values of the right hand side are stored in temporary locations until the entire block is finished and then they will be assigned to the variables on the left. For this case `C` will contain the original value of $B + 1$ rather than $A + 1$ as in the first case.

Use blocking assignments when you must have sequential execution. If you are modelling edge-sensitive behavior use non-blocking assignments. In synchronous sequential circuits, changes will occur based on transitions rather than levels. In this case we need to indicate whether the change should occur based on the rising edge or the falling edge of the signal. This behavior is modeled using the two keywords `posedge` and `negedge` to represent rising and falling edge, respectively, for example:

Listing 24: Edge Sensitive `always` block

```
1 always @(posedge clock or negedge reset)
```

This will start executing on the rising edge of clock or on the rising edge of reset. If your circuit has an asynchronous reset, you may need an `if-else` statement to indicate whether you are resetting or triggering the circuit, in this case the very last statement of the non-blocking assignment needs to be the one related to the clock. For a D flip-flop with asynchronous reset, example code is provided in Listing 25.

Listing 25: An example of a D flip-flop with asynchronous reset

```
1 module D_Flip_Flop(output reg Q, input D, clock, reset);
2         always @ (posedge clock or negedge reset);
3                 if(~reset)
4                         Q <= 1'b0;
5                 else
6                         Q <= D;
7 endmodule
```

6 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 26 will be discussed.

Listing 26: 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
```

6.1. The first line of Bad Code (Listing 27) can either be revised to Listing 27 or Listing 28.

Listing 27: Revision I of Bad Code (Line 1)

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

Listing 28: Revision II of Bad Code (Line 1)

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

6.2. The first line of Bad Code (Listing 27) 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 29:

Listing 29: Definition of Wires

```
1 wire w1;
```

6.3. Whenever a variable (wire) is being given a value, The `assign` keyword must preceede the variable being set and the associated logic. Revision of Bad Code (Listing 27) lines 3-6 can be seen in Listing 30.

Listing 30: Proper Usage of Assign Operator

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

6.4. The module should be closed with the `endmodule` keywork. A proper revision of the code from Listing 26 is as follows in Listing 31:

Listing 31: Corrected Implementation of Bad Code

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

7 CMOD-S6 DIP Assignments

Table 3: CMOD-S6 DIP Assignments

DIP Pin	**FPGA Pin**	Wire	Wire	**FPGA Pin**	DIP Pin
1	**P5**	PIO01	PIO48	**M2**	48
2	**N5**	PIO02	PIO47	**M1**	47
3	**N6**	PIO03	PIO46	**L2**	46
4	**P7**	PIO04	PIO45	**L1**	45
5	**P12**	PIO05	PIO44	**K2**	44
6	**N12**	PIO06	PIO43	**K1**	43
7	**L14**	PIO07	PIO42	**J2**	42
8	**L13**	PIO08	PIO41	**J1**	41
9	**K14**	PIO09	PIO40	**G2**	40
10	**K13**	PIO10	PIO39	**G1**	39
11	**J14**	PIO11	PIO38	**H2**	38
12	**J13**	PIO12	PIO37	**H1**	37
13	**H14**	PIO13	PIO36	**F2**	36
14	**H13**	PIO14	PIO35	**F1**	35
15	**F14**	PIO15	PIO34	**E2**	34
16	**F13**	PIO16	PIO33	**E1**	33
17	**G14**	PIO17	PIO32	**D2**	32
18	**G13**	PIO18	PIO31	**D1**	31
19	**E14**	PIO19	PIO30	**C1**	30
20	**E13**	PIO20	PIO29	**B1**	29
21	**D14**	PIO21	PIO28	**A2**	28
22	**D13**	PIO22	PIO27	**B3**	27
23	**C13**	PIO23	PIO26	**A3**	26
24		VU	GND		25