**Laboratory 6: Multiplexers**

**6.1 Objectives**

The objectives of this laboratory are:

* To become familiar with continuous assignments and procedural programming in Verilog.
* To design various multiplexer configurations in Verilog.
* To implement a 4-bit wide 4:1multiplexer in the Nexys2 FPGA prototyping board.

A multiplexer is a device that selects one of several input signals and forwards the selected input to the output. Typical multiplexers come in 2:1, 4:1, 8:1, and 16:1 forms. A multiplexer of 2n inputs has n select lines. A TTL series 8:1 MUX is 74151. It has three select lines S2, S1, S0. Each of the 8 possible combinations of S2, S1, S0 selects In this lab, you will design a half adder and a full adder using Verilog, on a Nexys2 board (from Digilent), which contains a Spartan 3E FPGA (from Xilinx). You will also use the I/Os on the Nexys2 board (Figure 5.2) to read in input bits and display the output.

**5.2 Verilog**

**5.2.1 Continuous Assignment**

In Lab 5, for designing the half adder and full adder, you may have written structural Verilog only in the gate primitive form. In this lab, you will also be using another way of writing structural code: *Continuous Assignment*. Here is an example of a 2:1 MUX description in primitive gates representation as well as continuous assignment statement form (also called a net-assignment statement).

**Primitive Gate Form:**

and(sa, ~s, a);

and(sb, s, b);

or(z, sa, sb);

**Continuous Assignment:**

assign z = (~s & a) | (s & b);

Here, ~ is the not operator and & is the and operator. This is very similar to the SOP or POS equations written for each output, and it is simpler than the primitive gate representation. From now on, we will use one of both methods as needed. Below is a list of Verilog operators to be used for the net-assignment statements.

~ not

| or

& and

^ xor

~^ xnor

Some characteristics of continuous assignments are:

 They can be used for modeling combinational logic.

 They are outside the procedural blocks (**always** statement and **initial** blocks).

 The left-hand side of a continuous assignment must be net (wire) data type.

 They may have a delay specified.

**5.2.2 Behavioral (Procedural) Verilog Basics**

For this lab, you will also use procedural Verilog, which allows if statements, for loops, and case statements. All procedural Verilog programs begin with the **always** statement:

always @(argument)

The argument can be **posedge** (for positive edge) or **negedge** (for negative edge) of a signal, or just the signal name:

always @(posedge clock)

always @(S)

In the first statement, the procedure is executed every positive edge of the clock and in the second case whenever S changes. The **or** and **and** operators can be used to combine signals together, as shown below:

always @(S or Q)

always @(S and Q)

Verilog uses the **begin** and **end** statement unlike the C programming language which uses open and closed braces to define a function. The following Verilog code gives a description of a 2:1 MUX in procedural form using the **if** statement.

always @ (\*)

begin

if (s == 0)

z = a;

else

z = b;

end

The **if** statement must be contained in an **always** block. Note also that y must be declared to be of type **reg**, as it is assigned a value within the **always** block. The notation \* in the always statement refers to the sensitivity list, which in this case is a, b, c. Thus, the notation @(\*) in this statement is equivalent to @(a, b, s). Any time any of these input values change, the **if** statement within the **always** block is executed. The use of the \* notation is a convenience that prevents you from omitting any of the signals or inputs used in the **always** block. A Verilog program can contain multiple **always** blocks, and these blocks are executed concurrently.

**Parametric Statements**

We can use the Verilog parameter statement to design a generic 2:1 MUX with input and output bus widths of arbitrary size, as shown below: In general, the syntax is:

*// A Generic N-bit wide 2:1 MUX using a parameter*

**module** mux2g

#(**parameter** N = 4)

(**input wire** [N-1:0] a,

**input wire** [N-1:0] b,

**input wire** s,

**output reg** [N-1:0] z

);

**always** @(\*)

**if**(s == 0)

z = a;

**else**

z = b;

**endmodule**

Note the use of the **parameter**statement that defines the bus width Nto have a default value of 4. This value can be overridden when the multiplexer is instantiated as shown below for an 8-bit wide 2:1 multiplexer called M8.

// An *8-bit wide 2:1 MUX using a parameter*

**module** mux28(

**input wire** [7:0] a,

**input wire** [7:0] b,

**input wire** s,

**output wire** [7:0] z

);

mux2g #(

.N(8))

M8 (.a(a),

.b(b),

.s(s),

.z(z)

);

**endmodule**

**Case Statement**

The Verilog **case** statement can also be used to implement a multiplexer. The **case**statement is an example of a *procedural statement* that must be within an **always**block. A typical line in the **case** statement, such as

2: z = c[2];

will assign the value of c[2] to the output zwhen the input value s[1:0] is equal to 2 (binary 10). Note that the output zmust be of type **reg**because its value is assigned within an **always**clause. In the **case**statement the alternative value preceding the colon in each line represents the value of the **case**parameter, in this case the 2-bit input s. These values are decimal values by default. If you want to write a hex value you precede the number with ’has in ’hAwhich is a hex value A. Similarly, a binary number is preceded with a ’b as in ’b1010 which has the same value (10) as ’hA. Normally, binary numbers are preceded with the number of bits in the number such as 4’b107. Using this notation, the number 8’b110011 will be the binary number 00110011.

We can build a 4:1 MUX using a **case** statement as follows:

// A *4-to-1 MUX using case statement*

**module** mux41c (

**input wire** [3:0] c ,

**input wire** [1:0] s ,

**output reg** z

);

**always** @(\*)

**case**(s)

0: z = c[0];

1: z = c[1];

2: z = c[2];

3: z = c[3];

**default**: z = c[0];

**endcase**

**endmodule**

**6.3 Pre-Lab**

Please make sure to complete the prelab before you attend your lab section. The lab will be long and frustrating if you do not do the prelab ahead of time. In this lab, you will use Verilog to implement several multiplexer configurations on the Xilinx Spartan 3E FPGA. The Verilog used to describe the multiplexers will be in structural as well as procedural.

**6.3.1 Pre-Lab Procedure**

**Part 1: 2:1 Multiplexer**

1. Create a top-level design called mux21\_topthat connects inputs *a* and *b* to the rightmost two slide switches of Nexys2, connects input *s* to *btn*[0] of Nexys2, and connects output *y* to *ld*[0].
2. Perform a functional simulation of your design.
3. Create a symbol for the multiplexer.

**Part 2: 4-Bit Wide 2:1 MUX Using if Statement**

1. Create an N-bit wide 2:1 MUX using the parameter statement. Create a symbol for it.
2. Create a 4-bit wide 2:1 MUX by instantiating the N-bit wide MUX designed above.
3. Functional Simulation. Perform a functional simulation of the circuit to verify that it is working correctly.
4. Create Symbol. Create a symbol for the multiplexer to use in the graphical editor. This creates a symbol file that is a Graphic File and can be viewed and edited by opening it.

**Part 3: 4-Bit Wide 4:1 MUX**

1. Design a 4-bit wide 4:1 multiplexer from three 4-bit wide 2:1 multiplexers.
2. Perform a functional simulation of the circuit. Paste the results in your prelab report.
3. Create a symbol for the 4-bit wide 4:1 MUX to use in the graphical editor.
4. Design a 4:1 multiplexer using the Verilog **case** statement.
5. Simulate the design. Paste the results in your prelab report.
6. Bring your Verilog codes in a flash drive.

**6.3.2 Pre-Lab Report**

In your prelab report, include the following:

1. Circuit schematics, Verilog programs, and simulation resultsfor all multiplexers discussed above.

Incorrect or incomplete designs and Verilog programs will not receive full credit. **If you have any problems with Verilog syntax and other pre-lab related issues, please resolve them before coming to the lab. Your TA may not be able to help you with these issues during the lab session.**

**6.4 Lab Procedure**

Set your 4-bit 4:1 MUX Verilog file (the hierarchical one built from three 2:1 multiplexers) as the top module of your design and implement the complete design (*synthesize, map,* and *Place & Route*) using the Xilinx ISE tools. Program the FPGA using the bit-stream file which is generated in the process. For checkoff, you will show the TA the following:

1. Show the multiplexer working on the board.
2. Demonstrate at least one selection operation from the patterns you have used in the test bench. Show that your simulation results above match the observed waveforms on the DLA.
   1. **Post-Lab Report**

Along with your post-lab report, submit the observed waveforms (in separate sheets) for the 4-bit wide 4:1 multiplexer.