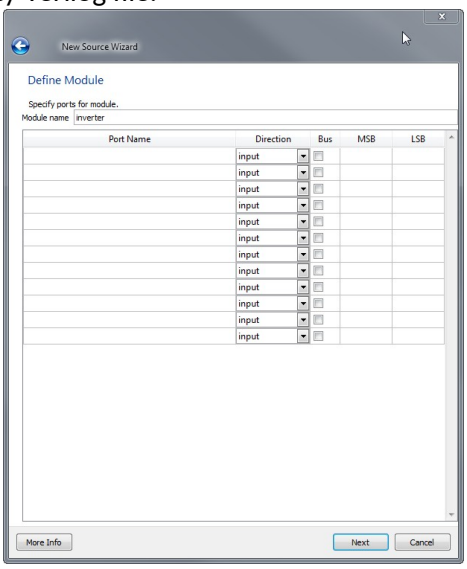
We are going to create a behavioral model of an inverter.

1. Create a new project.
2. Create a new source file. Type: Verilog Module. Name it ‘inverter’.
3. The Add Source wizard will ask you to define the module. Here is where we can create our input and output definitions. Leave it all blank for now and click ‘Next’ and then ‘Finish’
4. It will create an empty Verilog file.



5. The generated code declares and names the component.

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date:    10:45:26 01/23/2013
7  // Design Name:
8  // Module Name:    inverter
9  // Project Name:
10 // Target Devices:
11 // Tool versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////////////////////////////////////
21 module inverter(
22     );
23
24
25 endmodule
26
```

   a. ‘module’ indicates that it is a Verilog component.
   b. Input and output name declarations will go within the parentheses.
   c. ‘endmodule’ designates… the end of the module.

6. An inverter has two ports: one input and one output.
    a. Let's name the input 'inp' and the output 'op'
    b. Input/Output ports are declared like variables in C. <type> <name>;
    c. I/O port names are also listed in the module declaration.
    d. Modify your code as shown below.

```
21  module inverter(inp, op );
22
23  input inp;
24  output op;
25
26
27  endmodule
28
```

    e. 'module inverter(inp, op);' indicates that we have a module named inverter with two external connections: inp and op.

7. An inverter has one function that is always constant – it inputs a and outputs $\acute{a}$.
    a. We need to define this behavior in Verilog.
    b. We can define a static, concurrent function by using the 'assign' keyword.
    c. The unary operator for 'not' is a tilde '~'
    d. So, to define that op is always the inverse of inp, we use the line: 'assign op =~inp'
    e. Modify your code as shown.
    f. Create a testbench and ensure that it functions properly.

```
21  module inverter(inp, op );
22
23  //Declare inputs/outputs
24  input inp;
25  output op;
26
27  //Set op as the inverse of inp
28  assign op = ~inp;
29
30  endmodule
```

8. Another way to create the inverter is to use Verilog Primitives.
    a. Primitives are basic gates built into Verilog and the FPGA.
    b. For an inverter, the primitive is instantiated like this: 'not(<output>, <input);'
    c. Create a new source, call it 'inv_primitive'. Give it an input 'inp' and output 'op'
    d. Create the functionality using a 'not' primitive.
    e. Your code should look like this:
9. Test it.

```
21  module inv_primitive(inp, op );
22
23  //Declare inputs/outputs
24  input inp;
25  output op;
26
27  //Set op as the inverse of inp
28  not(op,inp);
29
30  endmodule
```

10. Now, let's look at a couple ways to make a multiplexer.
11. Let's do a simple 2-to-1 multiplexer
    a. Create a new source. Call it 'mux_21_simple'
    b. Give it the inputs 'mux_in' and 'sel', and an output 'mux_out'
    c. 'mux_in' is a 2-bit input. Declare it as so: 'input [1:0] mux_in;'
    d. Your code should look like this:

```
21  module mux_21_simple(mux_in, sel, mux_out);
22
23  //Declare inputs/outputs
24  input sel;
25  input [1:0] mux_in;
26  output mux_out;
27
28  endmodule
```

    e. Now we need to model the functionality. Since the function is constant we can use 'assign' again.
    f. For this we can use a ternary operator. It's essentially a short 'if' statement.
        i. The syntax is '(<control>) ? <Value_if_true> : <value_if_false>
    g. So, for the multiplexer, we can use: 'assign mux_out = (sel) ? mux_in[1] : mux_in[0];'
        i. When select is '1', the output is set to the value of 'mux_in[1]'
        ii. When select is '0' the output is set to the value of 'mux_in[0]'
    h. Your code should now look like this:

```
21  module mux_21_simple(mux_in, sel, mux_out);
22
23  //Declare inputs/outputs
24  input sel;
25  input [1:0] mux_in;
26  output mux_out;
27
28  assign mux_out = (sel) ? mux_in[1] : mux_in[0];
29
30  endmodule
```

12. Test and make sure it works correctly.

13. Now, let's look at some behavioral designs.
    a. In the behavioral design style, the functionality of the behavior of the design is described.
14. We will construct a behavioral multiplexer.
    a. Create a new source, and call it 'mux_21_beh', with the same inputs and outputs as the previous mux.
    b. In a behavioral model, wires, signals and in/out ports are not statically assigned. They do not constantly hold the current value. Instead, they are periodically set. So, we need to declare that the output will hold its value after it has been set.
    c. To do this, we use the 'reg' keyword. This designates the signal/wire as a register, meaning that any value assigned to it will be held until it is set to something else.
    d. Declare a reg named 'mux_out', the same as your output name.
    e. Your code should look like this:

```
21  module mux_21_beh(mux_in, sel, mux_out);
22
23  //Declare inputs/outputs
24  input sel;
25  input [1:0] mux_in;
26  output mux_out;
27
28  reg mux_out;
29
30  endmodule
```

    f. Behavioral models are typically sequential designs. Sequential functions are defined within an 'always' block of code.
        i. The syntax is 'always @(<triggers>) begin'
        ii. The 'triggers' should be replaced with the *sensitivity list*.
            1. This is the list of signals that trigger the sequential function.
            2. When the value of any of the triggers changes, the code is executed.
        iii. The block is ended by the 'end' keyword.
        iv. We will create an 'always' block to define the multiplexer.
            1. The design should update the output whenever the input changes.
            2. We want it to run whenever 'sel' or 'mux_in' changes.
            3. Therefore, the triggers are 'sel or mux_in'
            4. Add this code under your 'reg' declaration:

```
26  output mux_out;
27
28  reg mux_out;
29
30  always @(sel or mux_in)
31
32
33  end
34
35  endmodule
```

        v. We will put the behavioral definition inside this always block.
            1. When either input changes, the output should be updated.
            2. We can use if-else statements for this model.

    a. Syntax: ‘if (conditional) begin … end
3. If sel is ‘0’, we set the output to mux_in[0].If it is ‘1’, set it to mux_in[1];
4. Comparison operator is ‘==’
5. To specify that a value you enter is in binary, it is proceeded by <length>’b<number>.
    a. Example, 13 is a 4-bit number. So it is written as 4’b1101
    b. 3 is a 2-bit number, written as 2’b11
    c. Remember the apostrophe!
6. So, we need to compare ‘sel’ against 1’b0 and 1’b1.
7. Modify your code to add this functionality:

```
30  always @(sel or mux_in) begin
31  
32      if (sel == 1'b0) begin
33          mux_out = mux_in[0];
34      end
35      else if (sel == 1'b1) begin
36          mux_out = mux_in[1];
37      end
38  
39  end
```

g. Create a testbench and verify the functionality.
h. Now, we will do the same thing with case statements. Create a new source, called ‘mux_21_case’. Give it the same inputs, outputs, and registers. Also create the ‘always’ block, with the same triggers.
    i. Case statements start with ‘case(value)’ and end with ‘endcase’
    ii. Switches use the syntax: <targetValue> : <operation>;
    iii. The default case (the function that runs if none of the other cases are true) is declared by ‘default : <function>;’
    iv. The binary value ‘z’ is used to designate ‘nothing’. It’s not a 1 or 0, just low impedance nothing. We want this default to output ‘nothing’ (it means that the sel input is invalid)
    v. The mux code is as follows:

```
30  always @(sel or mux_in) begin
31  
32      case(sel)
33          1'b0: mux_out = mux_in[0];
34          1'b1: mux_out = mux_in[1];
35          default: mux_out = 1'bz;
36      endcase
37  
38  end
```

15. Test and verify.
16. Next, we will create a basic positive-edge-triggered D flip-flop.
    a. Create a new source called ‘dff’
    b. Give it the inputs ‘clk’ and ‘d’ (each one bit), and the one-bit output ‘q’
    c. Also make ‘q’ a register, since it is the output from a sequential module.
    d. We want the DFF to update on the positive edge of each clock cycle.

i. The trigger/sensitivity list is: ‘posedge clk’
ii. This means that the ‘always’ block will run each time the clock goes from ‘0’ to ‘1’ – the POSitive EDGE
iii. Each time, it should set the output to the value of the input. So, ‘q=d;’

e. Your code should look like this:

```
21  module dff (clk, d, q);
22
23  //Multiple inputs or outputs can
24  //be declared on a single line
25  input d, clk;
26  output q;
27
28  reg q; //to hold the output value
29
30  //run each time the clock goes
31  //from 0->1
32  always @(posedge clk) begin
33     q = d;
34  end
35
36  endmodule
```

17. Create a testbench for it.
    a. It may create a clock function for you. If not, you will need to create it in your testbench
        i. If it adds 'clk = 0' in the 'initial begin' block, remove it.
        ii. It should look like this:

```
25  module dff_testbench;
26
27      // Inputs
28      reg clk;
29      reg d;
30
31      // Outputs
32      wire q;
33
34      // Instantiate the Unit Under Test (UUT)
35      dff uut (
36          .clk(clk),
37          .d(d),
38          .q(q)
39      );
40
41      //Creates a clock with 100ns cycle time.
42      //50ns low and 50ns high
43      always begin //Basically, while(true)
44          clk = 1'b0; //Clock = 0
45          #50;        //Wait 50ns
46          clk = 1'b1; //clock = 1
47          #50         //wait 50 ns and repeat
48      end
49
50      initial begin
51          // Initialize Inputs
52          d = 0;
53          // Wait 100 ns for global reset to finish
54          #100;
55          //Start test values
56          d = 1;
57          #130;
58          d = 0;
59      end
60
61  endmodule
```

18. Simulate it and observe its behavior. Modify the timings and see how it changes the simulation.
    a. Initially, the value of 'q' will be invalid. This is because it is not set until the clock goes from 0->1

19. You can use '$finish' to designate the end of the simulation. So, the line '#50 $finish' just before 'end' tells the simulator to halt 50ns after the last operation.

```
50      initial begin
51          // Initialize Inputs
52          d = 0;
53          // Wait 100 ns for global reset to finish
54          #100;
55          //Start test values
56          d = 1;
57          #130;
58          d = 0;
59          #50 $finish;
60      end
```

    a. When ISim reaches the 'finish' it will open a new tab with the testbench source code in it. Just switch back to the first tab to view the waveforms. Then you can zoom to fit, etc.

20. Structural Design
    a. Let's make a structural NAND gate. This is essentially an inverter hanging off the end of an 'and' gate.
    b. We need to create a simple AND gate first. Follow the same steps as for creating the inverter. The file and module name should be 'and_21'.
    c. The inputs are 'in1' 'in2' and 'out' .
    d. The function is 'assign out = in1 and in2'
    e. The code should look as follows:

```
21  module and_21(in1, in2, out);
22
23  input in1, in2;
24  output out;
25
26  assign out = in1 and in2;
27
28  endmodule
```

i. Pro Tip: You can actually put both modules in the same file. In that case the single file would look like this:

```
21  module inverter(inp, op);
22
23  //Multiple inputs or outputs can
24  //be declared on a single line
25  input inp;
26  output op;
27
28  assign op = !inp;
29
30  endmodule
31
32  //Begin And21 module
33  module and_21(in1, in2, out);
34
35  input in1, in2;
36  output out;
37
38  assign out = in1 and in2;
39
40  endmodule
```

ii. This style is completely voluntary. If you experience issues, do not use this format, or ask for help.

(Next Page)

f. Now we create the NAND gate. Create a new module, and name it ‘nand_21’ (You may do this in a new source file, or at the bottom of your combined file as described above.)
    i. Two single bit inputs: inp_a and inp_b
    ii. One single bit output: nand_out
g. To instantiate a component in a structural design, it is declared as follows: <module_name> <unique_identifier> (.<in/out_name_1(local_connection), …)
h. Use a ‘wire’ to connect two components. The wire must have a unique name
    i. A convenient naming convention is to use the names of the components it is connecting. For example ‘wire and1_inv1’ connects the and1 components with the inv1 component. This is not required.
i. Refer to the following code:

```
21  module nand_21(inp_a, inp_b, nand_out);
22
23  //Inputs to the design
24  input inp_a, inp_b;
25  //Output
26  output nand_out;
27
28  //This wire will connect the and
29  // output with the inverter input
30  wire and1_inv1;
31
32  //Declare the 'and' instance.
33  // Name it and1.
34  // You can split it across multiple
35  // lines to make it easy to read.
36  and_21 and1(
37     .in1(inp_a),
38     .in2(inp_b),
39     .out(and1_inv1) );
40
41  //Declare the 'inverter' instance
42  // Name it inv1
43  inverter inv1(
44     .inp(and1_inv1),
45     .op(nand_out) );
46
47  endmodule
```

21. Create a testbench for your ‘nand_21’ module and verify the functionality.
22. Misc. Hints
    a. You can combine values into a single bus with the following format
        i. Say you have an output ‘out[3:0] and an inputs ‘in_a[1:0]’ and ‘in_b[1:0]’.
        ii. You want to set bits 3 and 2 of the output to the value of ‘in_a’, bit 1 to ‘0’ and bits 1 and 0 to the value of in_b[1].
        iii. Use this statement: ‘assign out = {in_a, 1’b0, in_b[1]}’
        iv. Out will then be a 4-bit number with the following values (in_a[1], in_a[0], 0, b_in[1])

v. You can also use this syntax to shift an input when assigning it to an output, such as a shift left or shift right. For an 8-bit input/output, with a shift-left of one, the statement is: 'out = {in[2:0],1'b0};'

b. Basic mathematical functions can be done natively in Verilog.
   i. Y = A + B sets Y to the value of the sum of A and B. It will discard overflow if not handled properly.
   ii. If A and B are 4-bit numbers and Y is also 4-bit, then any carry bit will be discarded.
   iii. If A and B are 4-bit numbers and Y is a 5-bit number, MSB of Y will hold the carry.
   iv. Y=A-B works exactly as you would expect it to.
c. You can define parameters, which work similar to constants or #defines in C.
   i. 'parameter <name> = <value>;'
   ii. You can use this as arguments.
      1. parameter size = 4;
      2. wire [size-1:0] inp; //Creates a wire with <size> bits in it.
d. Primitives do not need to have instance names.
e. Wires CANNOT be connected to different outputs.
f. Wires CAN be connected to multiple inputs, but only one output. For example, the output of an adder can be connected both to a multiplexer and an inverter, or another adder. Or an input can be connected to multiple components.

A Verilog reference PDF has been posted to Canvas.

I wish you luck with this tutorial and the lab assignment