Lab 2: Combinational Logic Design in Verilog

EE 316: Digital Logic Design

Overview

This lab is intended to take you through your first full-length Vivado project. By the end of this lab, you should be able to:

- Create a project from scratch in Vivado.
- Write design, simulation, and constraints files.
- Design decoders and multiplexers using structural, dataflow, and behavioral modeling.
- Describe basic syntactical features of Verilog.

Be sure to attend your assigned lab section each week. There is a fair amount of background information for this lab, and we will discuss the key concepts/ideas you need to know in-person. We will also field questions about the lab exercises.

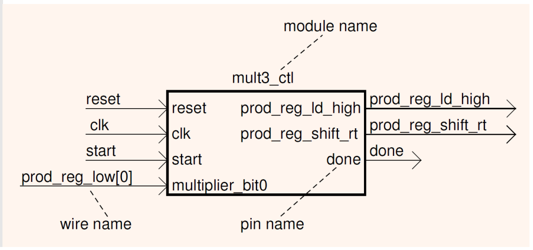
Background

The Structure of Verilog

The basic building block of Verilog is the *module*. A module describes the boundaries of a piece of Verilog code (with the keywords `module` and `endmodule`), the inputs and outputs of the code (with ports), what other modules the module needs (lower-level modules), and what the module does (the actual code). You can think about modules as "black boxes" that may contain a variety of circuits, other modules, and other pieces of code floating around but accomplishes some goal.

To create a module, you must specify its name, inputs and outputs. This process/piece of code is called the *module declaration.* The module declaration also specifies the data type and width of the inputs and outputs (more on that later). The inputs and outputs are collectively called *ports*: going back to the "black box" analogy, these are the input and output terminals of the black box. When you add a design file and specify its name, inputs and outputs, Vivado automatically creates the module declaration for you, so you will often not have to write it yourself. An example of a module declaration and its corresponding block diagram component is given below:

```
module mult3_ctl(
  input   reset,
  input   clk,
  input   start,
  input   multiplier_bit0,
  output logic prod_reg_ld_high,
  output logic prod_reg_shift_rt,
  output logic done);
```



The module declared above has four input ports and three output ports and is named `mult3_ctl`. Each port has a name, which is given in the declaration. The `logic` keyword is used in SystemVerilog and is beyond the scope of this class.

When you use a module in another module, you must *instantiate* the module. This means that you create an instance of the module inside the other module and specify which ports in the top module are connected to those in the lower module. In the black-box analogy, you are creating another black-box inside the larger black-box. The piece of Verilog below shows an example of instantiating the `mult3_ctl` module that was shown above in the `mult3` module. In this case, the `mult3_ctl` module is the *lower-level module,* and the `mult3` module is the *higher-level module.*

```
module mult3(
  input   reset,
  input   clk,
  input   [31:0]  a_in,
  input   [31:0]  b_in,
  input   start,
  output logic [63:0] product,
  output logic done);
//
//  Other code goes in here
//
//instantiate the control module
mult3_ctl mult3_ctl_0(
  .reset              (reset             ),
  .clk                (clk               ),
  .start              (start             ),
  .multiplier_bit0    (prod_reg_low[0]   ),
  .prod_reg_ld_high   (prod_reg_ld_high  ),
  .prod_reg_shift_rt  (prod_reg_shift_rt ),
  .done               (done              ));
endmodule
```

This specific instance of `mult3_ctl` is named `mult3_ctl_0`; you can instantiate the same module multiple times so long as you have different names for each module. In the above instantiation, the port `reset` in `mult3` is associated with the port `reset` in `mult3_ctl`, the port `clk` in `mult3` with the port `clk` in `mult3_ctl`, and so on. A key point to remember (unrelated to the instantiation) is that a module must end with the `endmodule` keyword; otherwise, Verilog will throw a syntax error!

There are several ways you can associate (or *map*) ports between a higher-level module and a lower-level module in an instantiation, but we will focus on two: by order and by name. Suppose you have two modules as declared below, and you want to instantiate `bar` in `foo`.

```
module foo(

input a,
input b,
output c);

// Ports by order
bar bar0(a, b, c);

// Ports by name
bar bar1(.x(a), .y(b), .z(c));

endmodule
```

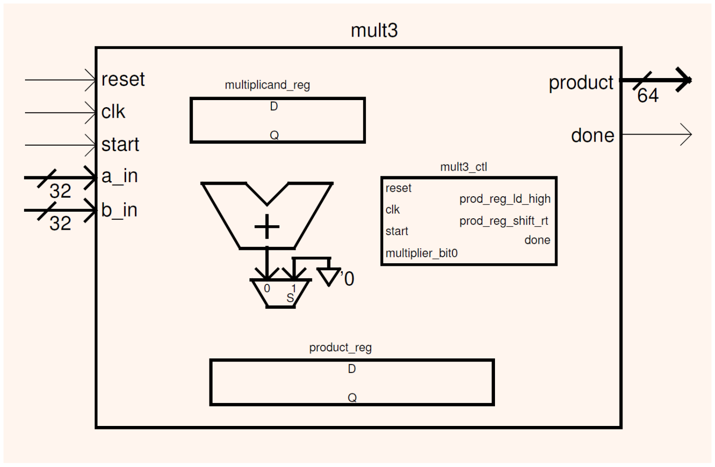
```
module bar(
input x,
input y,
output z);

// Dataflow OR gate
assign z = x | y;

endmodule
```

Associating ports *by order* in your instantiation means that you insert the ports of `foo` directly into the instantiation, assuming that the first instantiated port (in this case, `a`) is mapped to `x` in `bar`. This is often a bad idea, as you may switch the ports around or add and remove ports later in your design process. Associating ports *by name* in your instantiation means that you explicitly specify the port connections; i.e. `a` in `foo` to `x` in `bar`, etc. In general, we encourage you to always associate your ports by name, as it is easier to debug and makes your code easier to understand.

Instantiating modules is extremely important; in fact, many large, high-level designs often use a *top module* that controls data flow between lower-level modules. Top modules can include instantiations of lower-level modules as well as its own combinational and sequential logic blocks. This *hierarchical design* is an extremely useful tool to manage large projects with many modules, allowing designers to abstract some lower-level functionality and focus on the system design. An example of such a hierarchical design, using the `mult` code shown earlier, is shown below:



In the figure above, `mult3` is the top module and contains various other components and modules inside its body, including `mult3_ctl`, `multiplicand_reg`, and `product_reg`. As we go through the semester, we will build increasingly complex systems that will require multiple levels of modules.

Testbenches

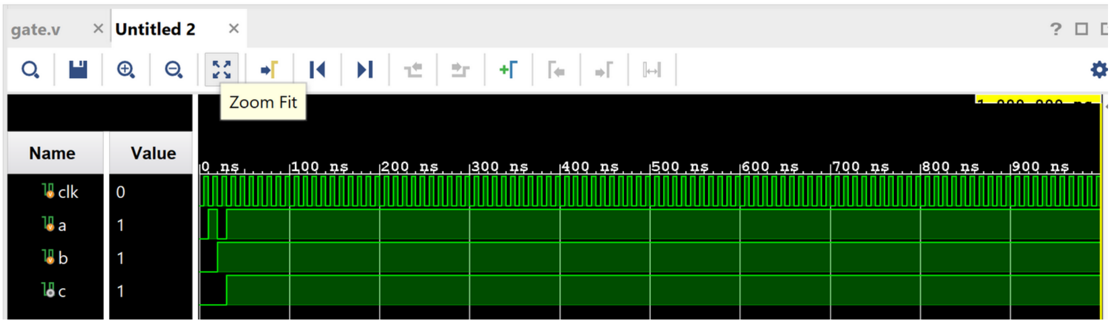
Simulation is a powerful way to verify that your circuit design works without having to test it on an actual device. To simulate your design, you will need a *testbench*. A testbench is a virtual structure that a simulation package (in this case, Vivado) will use as input triggers and output observers to your top module. In Vivado, the output of a simulation is a *waveform.* An example testbench and waveform was given to you in the tutorial for Lab 1 (reproduced here).

```
module tb_gate;
    reg a;      //Define the inputs of our device-under-test (DUT) as registers
    reg b;
    wire c;     //Define the output of our device-under-test (DUT) as a wire

    gate g(.a(a), .b(b), .c(c)); // This line instantiates the gate module. In other words, one instance of the gate module
                                 // is created and acts as a "black-box". We call this the "device/unit-under-test" (DUT).
                                 // This "black-box" functions like the gate module. We have named it g, and connected
                                 // the variable a in the testbench to the input a in gate, b to b, and c to c.

    initial begin                // This initial block toggles the inputs. In simulation, Vivado observes the outputs.
    a=0; b=0;                    // Unlike the always block, the initial block runs only once.
    #10                          // A delay of 10ns is used to ensure the circuit output is displayed.
    a=1; b=0;
    #10
    a=0; b=1;
    #10
    a=1; b=1;
    #10;
    end

endmodule
```



Name	Value
clk	0
a	1
b	1
c	1

In the testbench, the ports of the module under test (usually the top module) are declared, and the module under test is instantiated. Then, an *initial block* lists the order of triggers for the module under test: both inputs start at 0 for 10ns, then `a=1` for 10ns, and so on. The `#10` represents the delay of the simulation in nanoseconds; this is so that the module can actually respond to the trigger and that the output of the module can be displayed on the simulation waveform. As opposed to an *always block* (characteristic of behavioral modeling, see the tutorial for an example), an initial block only runs once. Also note that the testbench is declared as a module that has no input or output ports. This is because the testbench does not require inputs to run, and the output of the simulation is the waveform, not a specific signal.

It is also important to remember that because simulation runs on a computer and not on a real FPGA, it often takes much longer for a simulation to run than it takes the actual board to run. For combinational circuits, this is not an issue, but for sequential circuits such as state machines, several changes will need to be made to adjust for this limitation. We will discuss this in future labs.

Constraints Files

Constraints files are used in the synthesis and implementation steps of the FPGA design process to create the netlist and place your components (more on this later). Most importantly, it tells Vivado which on-board (hardware) components correlate to the inputs and outputs of your (software) top module. These specifications drive the placement and routing of your design.

Constraints files have a specific syntax that Vivado uses to generate the settings for each pin on the Basys3 board. Consider a snippet of a constraints file below:

```
set_property PACKAGE_PIN V17 [get_ports {a}]

set_property IOSTANDARD LVCMOS33 [get_ports {a}]
```

In the snippet above, pin V17 refers to switch SW0 on the board. During the synthesis stage, Vivado will connect/map pin V17 to input `a` in the top module in the netlist, and during the implementation stage, pin V17 will be routed to `a`. Also, LVCMOS33 is a keyword that sets pin V17 to use 3.3V logic and defines a “1” as 3.3V. The board can be set to other logic standards such as TTL (5V logic).

Constraints files often come pre-packaged with the specific board you use. You can find the template for the constraints file for the Basys3 board under Files > Labs > Reference > Basys3_Master.xdc. To use, add the file as a constraints file via the ``Add Sources..." dialog (the same as adding a design file, except choose “constraints file”) and uncomment the necessary lines by removing the preceding hashtag on each line. Then, edit the lines with the port names in your top module.

Verilog: Types of Modeling

There are three types of modeling in Verilog: structural, dataflow, and behavioral modeling. Structural (also known as gate-level) modeling uses gates to model combinational circuits. Verilog supports all of the main gates: AND, OR, NAND, NOR, NOT, XOR, and XNOR. The following syntax is used to create an instance of a gate:

```
[gate] [gatename] ([output], [input1], [input2], …);
```

The module below describes the structural implementation of an XOR gate using ANDs, ORs and NOTs. Additional wires are declared to hold the output of each gate, and the final output is the output of the intermediate wires.

```
module xor_gate(input a, input b, output c);

// Declare wires for intermediate values
wire nota;
wire notb;
wire anotb;
wire bnota;

// Gates
not not1(nota, a); // nota = a'
not not2(notb, b); // notb = b'

and g1(anotb, a, notb); // anotb = ab'
and g2(bnota, b, nota); // bnota = a'b
or g3(c, anotb, bnota); // c = ab' + a'b

endmodule
```

Structural modeling is usually used to describe extremely low-level designs. Most higher-level designs, including combinational logic circuits, are described using dataflow modeling and/or behavioral modeling.

Dataflow modeling is used to describe most combinational logic circuits. It is characterized by *continuous assignment* with the `assign` keyword. Continuous assignment means that the expression is evaluated continuously: the value of the statement is immediately updated if there is a change in the inputs. An example of a dataflow model for the XOR gate is shown below.

```
module xor_gate(input a, input b, output c);

assign c = (~a & b) | (a & ~b);

endmodule
```

The advantage of dataflow modeling is that logical functions can be directly translated into Verilog using *bitwise operators.* Bitwise operators (as opposed to logical or arithmetic or relational operators) are used in combinational logic circuits to represent AND (&), OR (|), NOT (~), and XOR (^) gates.

Behavioral modeling strives to describe the *functionality* of a design as opposed to its gate-level or circuit implementation. It is primarily used to model sequential circuits but can be used to represent combinational circuits (as seen in Lab 1). The basic building blocks of behavioral modeling are `initial` and `always` blocks. `initial` blocks only execute once, while `always` blocks execute whenever a port in its *sensitivity list* updates. The sensitivity list is the list of ports that are in parentheses after the @ in its syntax, which may contain an asterisk. Testbenches have initial blocks to specify the combination of inputs to test a module with. Always blocks are widely used in sequential design, which we will see later. Below is an example of a behavioral implementation of the aforementioned XOR gate.

```
module xor_gate(input a, input b, output c);

reg c_buf = 0; // Declare buffer register for output
assign c = c_buf; // Set output to buffer register
always @(*) begin // Combinational always block
 case({a,b})
  2'b00: c_buf = 0;
  2'b01: c_buf = 1;
  2'b10: c_buf = 1;
  2'b11: c_buf = 0;
  default: c_buf = 0;
end

endmodule
```

Note that the always block's code begins and ends with the keywords `begin` and `end`. These keywords are the equivalent of braces in C: they define the boundaries for a block of code. Also note the structure of the `case` statement. The `{a,b}` is called the *concatenation operator*: it concatenates the signals a and b and treats it as a single unit, which is checked with each of the two-bit options, 00, 01, 10, and 11. The `default` case can be specified in case you have an input combination missing. Furthermore, note that the sensitivity list contains an asterisk (*). This indicates that the always block should continuously be updated, as our design is combinational and not sequential. If, say, we only wanted the block to update on a clock signal `clk`, we can replace the asterisk with `clk` (more on this later).

Procedure

In this lab, you will design and implement a 3x8 decoder and a 4x1 multiplexer.

Part A: Decoder

Super Smash Bros is a popular cross-platform video game in which characters from various Nintendo franchises fight each other on a platform, seeking to knock the other person off the platform. Depending on their chosen characters, players of the game have access to a variety of special moves and combos that deal a large amount of damage to their opponents. These special moves are accessed by pressing specific combinations of buttons on their video game controllers.

To model the behavior of the video game controller, we will use a 3x8 decoder. For inputs, we will use the board buttons btnU (up), btnL (left), and btnR (right) as the inputs and LED0 through LED7 to signify which combo (combo0 through combo7) is currently selected. When LED0 is on, combo0 is selected; when LED1 is on, combo1 is selected, and so on through LED7. Only one combo should be enabled at one time. SW0 is used as an enable switch: if the SW0 is high, the combos are enabled, and if SW0 is low, the combos are disabled.

1. Complete the truth table below for a 3x8 decoder, assuming that the switch SW0 is high.

up	left	right	combo0	combo1	combo2	combo3	combo4	combo5	combo6	combo7
0	0	0								
0	0	1								
0	1	0								
0	1	1								
1	0	0								

1	0	1								
1	1	0								
1	1	1								

2. Write the 8 output logic equations for the 3x8 decoder.
3. Create a new design file in Vivado, and name it `decoder.v`. When prompted to add inputs and outputs, click "OK" and skip adding inputs and outputs. Vivado will declare an empty module for you.
4. Edit the module declaration to contain the following ports:

```
module decoder(

// Creates four input ports, shorthand for spelling all the inputs out

input up, left, right, enable,

output combo0, combo1, combo2, combo3, combo4, combo5, combo6, combo7

);


// Structural


// Dataflow


// Behavioral


endmodule
```

5. Using the logic equations you wrote and the tutorial code from Lab 1 as a guide, model the decoder behavior using structural, dataflow, and behavioral modeling.
6. Using the code below and the tutorial code from Lab 1 as a guide, write a testbench (simulation source) for the decoder. Name it `tb_decoder.v`. You should have one testbench for all three types of modeling. Run simulations verifying the functionality of the decoder.

```
module tb_decoder;


reg up, left, right, enable;

wire combo0, combo1, combo2, combo3, combo4, combo5, combo6, combo7;


// Instantiate module under test here
```

```
decoder uut( // Fill in the port assignment here
       );

initial begin

// Insert testcases here


end
endmodule
```

7. Add the constraints file from Canvas, name it `decoder_constrs.xdc` and edit it to the module specifications. Use the tutorial code from Lab 1 as a guide.
8. Program the board with your decoder and verify that all three types of modeling work.

Part B: Multiplexer

Dance Dance Revolution is another popular video game series that requires players to "dance" on a platform, pressing four pads with colored arrows pointing left, right, up, and down with their feet. To read in player inputs, the game designer must rapidly switch between the four pads, checking to see if it is pressed. (This behavior is actually quite common and will be the basis for a future lab on sequential logic design.) A 4x1 multiplexer can be used to toggle between the four different arrow pads.

For this lab, we will model the multiplexer as follows: the select lines will be switches SW1 and SW0, and the four input lines will be the buttons btnU (up), btnD (down), btnL (left), and btnR (right). The output will be tied to LED0: when the selected button is on, LED0 is on, and when the selected button is off, LED0 is off. Let the buttons btnU (up), btnD (down), btnL (left), and btnR (right) correspond to the select line values {SW1,SW0} = 00, 01, 10, and 11, respectively.

1. Write the truth table for the 4x1 multiplexer, using at most four rows.
2. Write the output logic equation for the multiplexer.
3. Create a new design file in Vivado and name it `mux.v`. In the project source hierarchy, right-click on the file and set it as top.
4. Edit the module declaration to include the input and output ports:

```
module mux(
// Fill in ports here


);
```

```
// Structural


// Dataflow


// Behavioral


endmodule
```

5. Using the logic equation you wrote and the tutorial code from Lab 1 as a guide, model the multiplexer using structural, dataflow, and behavioral modeling.
6. Using the tutorial code from Lab 1 as a guide, write a testbench for the multiplexer. Name it `tb_mux.v`. You should have one testbench for all three types of modeling. Run simulations verifying the functionality of the multiplexer.

   Because the multiplexer has 64 different input combinations, you should not exhaustively test every input combination. Choose your testcases judiciously so that you have a subset of testcases that can verify the functionality.
7. Add the constraints file from Canvas, name it `mux_constrs.xdc` and edit it to the module specifications.*
8. Program the board with your mux and verify that it works.

*If you need help with this step, please read the "EE 316 Vivado Multiple Lab Parts in One Project" document in the labs' reference folder on Canvas.

Submission

When you submit labs, you will need to submit two things: your zipped source code and a PDF answering questions given in the lab document. Below are the instructions for creating your zip file. Your PDF should be submitted separately, NOT included inside your zip file.

1. Create the zip file with your design files, testbenches, constraints files, and bitstreams. You should have a total of 2 design files (`decoder.v` and `mux.v`), 2 testbench files (`tb_decoder.v` and `tb_mux.v`), 2 constraints files (`decoder_constrs.xdc` and `mux_constrs.xdc`), and 2 bitstreams (`decoder.bit` and `mux.bit`), for a total of eight files. You can choose the type of modeling for your bitstreams.
2. Create your PDF. Include the following, in order:
   a. A cover page with your name, EID, section unique number, and professor.

   b. Completed truth tables for the decoder and multiplexer.
   c. Boolean logic equations you wrote for the decoder and multiplexer.
   d. Two screenshots of the circuit schematics (found under “Elaborated Design”) generated by Vivado, one for the decoder and one for the multiplexer (you can choose the type of modeling).
   e. Two screenshots of the simulation waveforms, clearly labeled with the module and type of modeling under test (e.g. “Decoder – structural” or “Multiplexer – dataflow”).
3. Submit your zip and PDF as one submission on Canvas.

<u>Checkout Process/Grading</u>

The checkout process involves a 10min one-on-one meeting with a TA the week after you submit your lab. You will schedule your checkout the week the lab is due via the Canvas appointment scheduler. The checkouts will be held in EER 0.716, the lab room. During the checkout, the TA will ask you to demonstrate the functionality of your code on the FPGA board, look at your source files, and ask you questions regarding the lab. The questions may range from syntax questions (e.g. "what does .a(a) mean?") to high-level design questions, such as "why did you choose these testcases?" You should be able to answer the questions in enough detail to show the TA that you understand the main concepts in the lab. Some sample checkout questions are provided below.

When it is time for you to check out, have your lab report (PDF file) and Vivado project open, and flash the board before the TA comes to meet with you. **<u>You MUST check out a Basys3 board from the lab checkout desk (EER 1.834) BEFORE you come to the lab.</u>** If you aren't prepared at your scheduled time, the TA will move on to the next person. If you miss your checkout time, you will need to re-schedule with the head TA.

Your grade will be determined during your checkout by the TA who is checking you out. The rubric is given below and posted on Canvas. Your TA will use the Canvas rubric to assess you based on a combination of your code’s functionality, your documentation (PDF file), and your oral responses to the TA during checkout. You should know your grade when you walk out of your checkout. Late submissions will incur a 5% penalty per day up to a week, after which your grade will be a zero. In addition, failure to check out within a week of submitting your lab will also result in a zero.

<u>Sample Checkout Questions</u>

Checkout questions may include, but are not limited to (in no particular order):

1. What does Vivado do in the synthesis and implementation phases of FPGA programming?
2. What is the role of the constraints file in the FPGA programming process?
3. What does structural Verilog mean? Why is it useful?
4. Why does the synthesis fail if the constraints file variables do not match the variables you declare in your top module?
5. Aside from what connections are made between the board and the modules, what other information does the constraints file tell you?
6. Explain how your code works.
7. What does .a(a) mean?
8. What is the difference between port matching by name and port matching by order?
9. What is the difference between module instantiation and module declaration?
10. Why is behavioral Verilog useful?
11. What is one advantage of dataflow Verilog over structural Verilog?
12. What is a port?
13. What is the difference between port association by order and by name? Which is better?
14. Why are there no input or output ports in the testbench module declaration?
15. What is the difference between structural, dataflow, and behavioral modeling?
16. What is a sensitivity list?
17. How many times does an initial block run?

Rubric

Below is the grading rubric for this lab. Please review it before submitting your lab to ensure you meet all the requirements.

	Missing	Attempted	Half-correct	Almost correct	Fully correct
Assessed from zip file submitted on Canvas					
Decoder: Models	0	5	6	7	8
Decoder: Testbench	0	5	6	7	8

Mux: Models	0	5	6	7	8
Mux: Testbench	0	5	6	7	8
Assessed from deliverables PDF					
Truth tables	0	5	6	7	8
Logic equations	0	5	6	7	8
Schematics	0	5	6	7	8
Waveforms	0	5	6	7	8
Assessed from in-person checkout					
	No-show	Needs work	Fair	Very good	Excellent
Board functionality: Decoder	0	5	6	7	8
Board functionality: Mux	0	5	6	7	8
Checkout questions	0	2.5	5	7.5	10
On-time submission and checkout	0	2.5	5	7.5	10