Introduction to Verilog and Verilog Design

During this course you will learn to construct digital logic using the Verilog Hardware Description Language (HDL). While Verilog has some resemblance to the C language, the underlying concepts are very different.

If there is one piece of advice I can pass along prior to beginning the lab projects, it is this:

Sketch out a block diagram of the circuitry you are trying to construct, complete with your best guess as to define blocks that perform simple functions, and how the blocks are connected, including interconnection signal names. Use the diagram while you code. Understand what your code is doing as you construct it.

1. How to construct a Verilog Design

A Verilog design (or project) is constructed using one or more Verilog modules. A complete design can be created using a single module, or with multiple modules using a hierarchical approach.

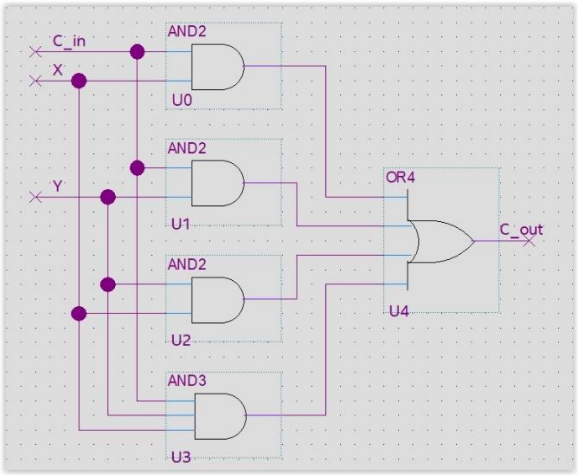
Best practice is to have a single Verilog module in a file. The name of the module and the name of the file should match. The file name should have a .v extension.

Verilog is case sensitive.

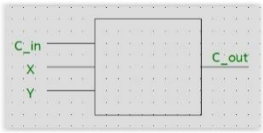
A module contains a list of the input signals to the module, and the output signals generated by the module. It is easy to conceptualize the module as a block in a diagram, where the inputs and outputs to the block are the inputs and outputs defined in the module.

Your designs will have one top level module, and may contain multiple lower level modules. In this class we will construct designs that will be compiled and programmed into an Intel Max10 FPGA on your DE10-Lite board. The inputs and outputs of the top level module will connect to pins on the FPGA. No other lower level module ports make direct connections to the FPGA.

Let's try an example. Someone hands you a schematic that looks like this:



You can see that the inputs to the module are C_in, X, and Y, and the output is C_out. A simplified block diagram of this schematic looks like this:



You can begin to code this specific module as shown below. Note that the input and output signal names can be anything you choose, as long as the names are valid Verilog syntax (at this point we are not interested in targeting this code to our boards). In the example below the same signal names are used as are shown on the schematic. Note that // denotes a line comment (block comments can use /* */ like C). Any of the 3 examples will compile without error (other than we are missing the actual code).

```
// Example1:
module my_design (X, Y, C_in, C_out);
input X;
input Y;
input C_in;
output C_out;
…
…
endmodule

// Example 2:
module my_design (X, Y, C_in, C_out);
input X, Y, C_in;
output C_out;
…
…
```

```
endmodule

// Example 3:
module my_design (input X, input Y, input C_in, output C_out);
…
…
endmodule
```

You can see that a Verilog module starts with the keywork **module**, and a list of signals entering and leaving the module. The direction of the signals must be defined.

2. Completing the design

You now have the framework for this design module. To complete the module you must add the code that will function like the design in the schematic.

Verilog provides a rich syntax that gives you options for how you can describe a design. In general you should try to provide a description of the circuit behavior, and not provide an implementation. What do I mean by this? Keep reading, it will be explained.

Appendix A in the course text has a lot of good Verilog reference information. Plus, there are many websites which explain and demonstrate Verilog. We will discuss Verilog during class lectures, and work through examples. At this time it is not important that you know enough Verilog to code a design.

Below is valid Verilog that describes the exact implementation as shown in the schematic. Note that the Verilog language contains gate level primitives (**and, or**).

Approach 1:

```
module my_design (input X, input Y, input C_in, output C_out);

wire out1, out2, out3, out4;

and U0 (out1,C_in, X);
and U1 (out2,C_in, Y);
and U2 (out3, X, Y);
and U3 (out4, C_in, X, Y);
or  U4  (C_out, out1, out2, out3, out4);

endmodule
```

We must create 4 unique instances of AND gates, and one unique instance of an OR gate. The module or component name (or in this case, the built in Verilog component) is followed by a unique instance name (again, anything that is legal Verilog), but always unique). The

list of ports in each instantiation is using positional association, where the first port is the output, followed by the input signals. You would need to read the Verilog documentation to know that the output is listed first, but trust me, it is.

Since the design needed some additional signals to connect the AND gate outputs to the OR gate inputs, the **`wire`** data type was included. You use **`wire`** whenever you need to connect something that is driving a signal (the AND gates in this case) to some logic inputs (the OR gate inputs in this case). The names of the signals are up to you (again, the names must be valid Verilog syntax). Note that all inputs and outputs of a module are type **`wire`** by default, unless explicitly redefined (to be discussed later).

The Verilog above is correct, **but not recommended**. The code above is the exact implementation of the schematic. The purpose of code is not immediately obvious, and would require someone to draw out the resulting logic to understand it. A better approach would be to create a higher level or more abstract description of circuit behavior.

Approach 2:

```
module my_design (input X, input Y, input C_in, output C_out);

assign C_out = C_in & X |
               C_in & Y |
               X & Y    |
               C_in & X & Y;
endmodule
```

The keyword **`assign`** is required anytime you make an assignment to a signal that is outside any block of procedural code (to be discussed later). A common error is forgetting the **`assign`** keyword. The & is the Verilog AND operator, and | is the Verilog OR operator.

Is the above code easier to understand than Approach 1? Yes, but we simply replaced the textual version of the schematic gate symbols with a Boolean equation. This is a higher level of abstraction, but still more descriptive of implementation than of behavior.

By writing the equation across four lines, it is clear that the AND function (&) in each line is performed first, and then the four results are OR'd (|) together. If the equation was written on a single line, use of parentheses is recommended to eliminate any ambiguity of which Boolean function is performed first.

```
assign C_out = (C_in & X) | (C_in & Y) | (X & Y) | (C_in & X & Y);
```

Approach 3:

```
module my_design (input X, input Y, input C_in,
                  output reg C_out);
```

```
always @ (X, Y, C_in)
   begin
     if (X + Y + C_in > 1)
        C_out = 1;
     else
         C_out = 0;
    end
endmodule
```

Approach 3 uses an **if / else** construct and arithmetic operators to describe the desired behavior. Each input is a one bit wide entity, which can be considered as a numeric value just as it can be considered a Boolean value. The three inputs are added together, and if the sum is greater than 1 (Verilog defaults to decimal number interpretations unless other instructed), C_out is assigned the value 1, otherwise assigned 0.

Whether you know it or not, the design we are using describes the generation of a carry out bit whenever two 1 bit inputs are added together along with a carry bit from a previous adder stage. If two or more of the inputs are one, a carry bit must be generated. Therefore, it makes perfect sense to add the inputs together. Notice that we are now describing behavior of the circuit, but not exact implementation. This more abstract approach to design is the reason industry moved away from schematics and moved to HDL.

Several things about Approach 3. Anytime **if / else** logic is used in Verilog, it must be inside a procedural block (the **always** block). Any code construct that must be interpreted in a specific order is considered to be procedural code, and must be inside a procedural block. For code we intend to compile and synthesize into logic, **always** blocks are used.

Notice that the a**lways** block contains a list of inputs. This list is referred to as a sensitivity list, and this list instructs a simulator that the output of the procedural block must be reevaluated any time one of the signals in the sensitivity list changes. Notice also the **begin / end** construct. You need to surround code blocks by **begin / end** in the same manner you would use { } in C – whenever there is more than one executable statement.

Finally, notice that the keyword **assign** is not used when assigning to C_out, since the assignment is made inside a procedural block. Next notice that the keyword **reg** was added to the list of ports in the first line of code. Verilog requires that an assignment in a procedural block must be to a signal of type **reg**. Understanding when to use **wire** and when to use **reg** in Verilog is one of the most misunderstood aspects of the language. Any time you make an assignment inside a procedural block, the signal on the left side of the = must be of type **reg**, and the keyword **assign** is not used. Anytime you make an assignment outside a procedural block, the signal of the left side of the equal sign must be of type **wire**, and the keyword **assign** must be used.

3. Use of multiple modules in a design

When writing software in a language such as C, you often create functions for code that you will execute multiple times. You can call the function as many times as needed, and will get different results based on the inputs you provide. Functions simplify a complex design, and make the resulting code easier to read. A function in C is compiled one time, resulting in a block of object code that is reused as many times as the function is called.

In Verilog, you may also create a module that you will instantiate (create an instance of) multiple times. An example might be a 7 segment display decoder. You write the decoder module to display correct values on a single 7 segment display. Then, if you need to use multiple displays, you instantiate the decoder module multiple times. The key difference is that each time you instantiate the decoder module, additional unique decoder block logic is created.

Let's go back to the original schematic design. Because I am stubborn, I refuse to use the AND and OR primitives provided in the Verilog language, and insist on making my own (my_and2, my_and3, and my_or4). My design will now consist of four separate files (my_design.v, my_and2.v, my_and3.v, and my_or4.v), because I am following the best practices that dictate one module per file, and that the module name and the filename match.

```
// File my_design.v
module my_design (input X, input Y, input C_in, output C_out);

wire out1, out2, out3, out4;

my_and2 U0 (.x(out1), .a(C_in), .b(X));
my_and2 U1 (.x(out2), .a(C_in), .b(Y));
my_and2 U2 (.x(out3), .a(X),    .b(Y));
my_and3 U3 (.x(out4), .a(C_in), .b(X), .c(Y));
my_or4  U4 (.x(C_out), .a(out1), .b(out2), .c(out3),
            .d(out4));
endmodule

// File my_and2.v
module my_and2 (input a, input b, output x);
assign x = a & b;
endmodule

// File my_and3.v
module my_and3 (input a, input b, input c, output x);
```

```
assign x = a & b & c;
endmodule

// File my_or4.v
module my_or4 (input a, input b, input c, input d, output x);
assign x = a | b | c | d;
endmodule
```

Notice that the instantiations of the lower level modules in my_design are not using a positional association, but instead use named association. Named association is recommended. The port names in the instantiation must match the names given in the individual modules. The port name is preceded by a period, then the signal that will be attached to the port name is inside parenthesis.

4. More on `wire and reg`

Verilog uses different data types, depending on the needs of the user. The two types we will use in this class are **`wire and reg`**.

A **`wire`** is simply a connection between an output driver and an input. Think of a wire as any line on a schematic that connects an input and output together. A wire completes a connection, but has no other properties such as memory, or remembering the value it holds.

All inputs and outputs of a module are **`wire`** s by default. **`Wire`** s can take on the values of logic high (1), logic low (0), high-impedance (Z), or unknown (X). If you are assigning a value to a signal outside of a procedural block, the assignment must be to a signal of type **`wire`**, and must include the **`assign`** keyword.

`wire` bob;

`assign` bob = something_or_other;

A **`reg`** in Verilog is defined as a variable data type. A **`reg`** has the characteristic that it can maintain its assigned value until the value is changed, in a similar manner that a variable in C will hold its value until changed. A value of type **`reg`** can only be updated within a procedural block. Or, put another way, any assignment done inside a procedural block must be to a signal of type **`reg`**. The assign keyword is not used in this case.

`reg` frank;

`always` @ (blah blah)

frank = something_else;

If code in an always block is assigning a value to a module output, you will need to modify the output to be type **`reg`**.

```
module test (X, Y);
input X;
output reg Y;
```
or

```
module test (input X, output reg Y);
```

5. Vectors in Verilog

It is common to have groups of signals (a bus or a vector or a one dimensional array) that are referred to using a single name. Verilog supports vector notation through the use of indices that are indicated through the use of square brackets **[ ]**. For example, a design may have an address bus that is 16 bits wide. To define this bus, the Verilog syntax looks like this:

```
wire [15:0] address_bus;
```

or

```
reg [0:15] address_bus;
```

The first index refers to the leftmost, or most significant bit in the address bus. Whether you use a high to low or low to high index scheme is up to you. If you are dealing with an industry standard you should follow the standard, otherwise just be consistent throughout your design.

The indices must be to the left of the vector name when defining the vector, but then go to the right of the vector name when used.

```
assign address_bus [15:0] = new_address [15:0];
```

Note that if you are assigning a value to every signal in the vector, you can omit the indices. However, code is more readable when indices are included.

To select a single signal from a bus:

```
address_bus[7] or address_bus[7:7]
```

To select a range of signals from a bus:

```
address_bus[7:0]
```

For module inputs or outputs, the indices must come before the input or output name.

```
module bubba (catfish, flyrod);

input [7:0] catfish;

output [1:0] flyrod;
```

or

```
module bubba (input [7:0] catfish, output [1:0] flyrod);
```

It is very important in Verilog you do not assign a value of one width to a vector of another width. Verilog will assign something to the other bits that is guaranteed to break your design.