CDA 3201L Lab #4

Combinational Logic Design and Verilog HDL

Welcome to CDA3201L Lab #4! The purpose of this lab is to introduce you to modeling basic combinational circuits using a hardware description language (HDL). Most real-world digital circuits are designed using HDLs, with the two most popular languages being Verilog and VHDL. Verilog is a lot like C in syntax, but there are some hardware-specific features in the language. VHDL is a little more verbose. Electronic design automation (EDA) tools are able to read in a description of a circuit written in Verilog or VHDL and process it in some way. For now, we will focus on modeling a circuit using structural / gate-level Verilog statements and simulating it.

<u>To prepare for Lab #4, carefully read and work with your lab partner to complete the prelab BEFORE you come to lab.</u>

Prelab

The basic building block for a Verilog design is a *module*. The module describes an interface using a port list and includes information about what the input and output pins are, and the width (number of bits) of each. The module encapsulates the logic of the design. You can use two input gates like **nand**, **nor**, **xor**, etc. to build the functionality of the module. The general syntax is: **gate_type (out, in_1, in_2)**, where **out** is the output wire, and **in_1** and **in_2** are the two input wires. For example, the function f = ab + cd can be modeled as follows:

```
module my_function(a, b, c, d, f);

      input a, b, c, d;
      output f;

      wire out0, out1;

      and(out0, a, b);
      and(out1, c, d);
      or(f, out0, out1);

endmodule
```

Gates with 3 or more inputs are also available, e.g. **and (out, in_1, in_2, ..., in_n)**, as are inverters, e.g. **not (out, in)**. The design can be verified with a Verilog simulator using a file called a *testbench*. A testbench is a special module that has no inputs or outputs, but instantiates the circuit under test (CUT). This is similar to instantiating a new object in an object-oriented programming paradigm. It also sets up initial conditions, sets up the simulation parameters, and defines input patterns. Here is an example testbench for module my_function:

```
module my_function_tb();

      reg a, b, c, d;
      wire f;

      my_function U0(a, b, c, d, f);

      initial begin

            $dumpfile("test.vcd");
            $dumpvars;
            $display("Starting simulation...\n");
            $display("Time\ta\tb\tc\td\tf\n");
            $monitor("%2d\t%d\t%d\t%d\t%d\t%d",$time,a, b, c, d, f);

            a = 0;
            b = 0;
            c = 0;
            d = 0;

            #10 a = 1; b = 1;
            #10 a = 0;
            #10 c = 1; d = 1;
            #10 c = 0; a = 1;
            #10 $finish;

      end

endmodule
```

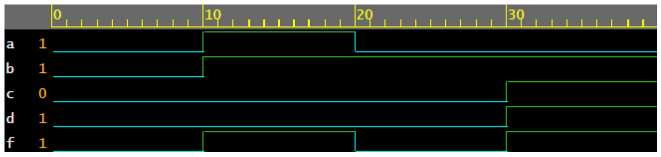
Let's break down the code step by step. The first line (reg a, b, c, d) defines four *registers* that will hold a single bit. These will serve as inputs to our circuit. We then declare a *wire*, f, which will connect to the output of our module. Then, we instantiate our module and "connect" the registers a, b, c, d and wire f to the corresponding input and output ports on our module. The name "U0" is just an identifier; in theory we could instantiate the module my_function multiple times, as long as they have unique names (e.g. U1, U2, U3, etc.)

Next, we have an "initial" block which sets up some parameters for simulation. We define an output file "test.vcd" which will hold simulation information, then instruct the simulator to save all variables (a, b, c, d, f) as their values change into the test.vcd file. This is used for displaying a timing diagram / waveform later. The "display" command outputs a string to the console. The "monitor" line tells the simulator to output the specified information any time any of the listed variables change. The first argument for the $monitor function is the format string (just like in C/C++/Python) that defines how we want the variables to be displayed in the console.

Now, we start changing our inputs. First, they are all set to 0 (a=0, b=0, etc.). Then, we wait 10 time steps (#10) before changing a to 1 and b to 1. Since our function is true when a=b=1, we would expect f to be 1 at this point. We once again wait 10 time steps and change a back to 0 (and of course, we would expect f to change to 0 as well). We then set c and d to 1 (expecting f to be 1), and wait another 10 time steps before changing c to 0 and a to 1. Finally, we finish the simulation, #10 time steps later at t = 50.

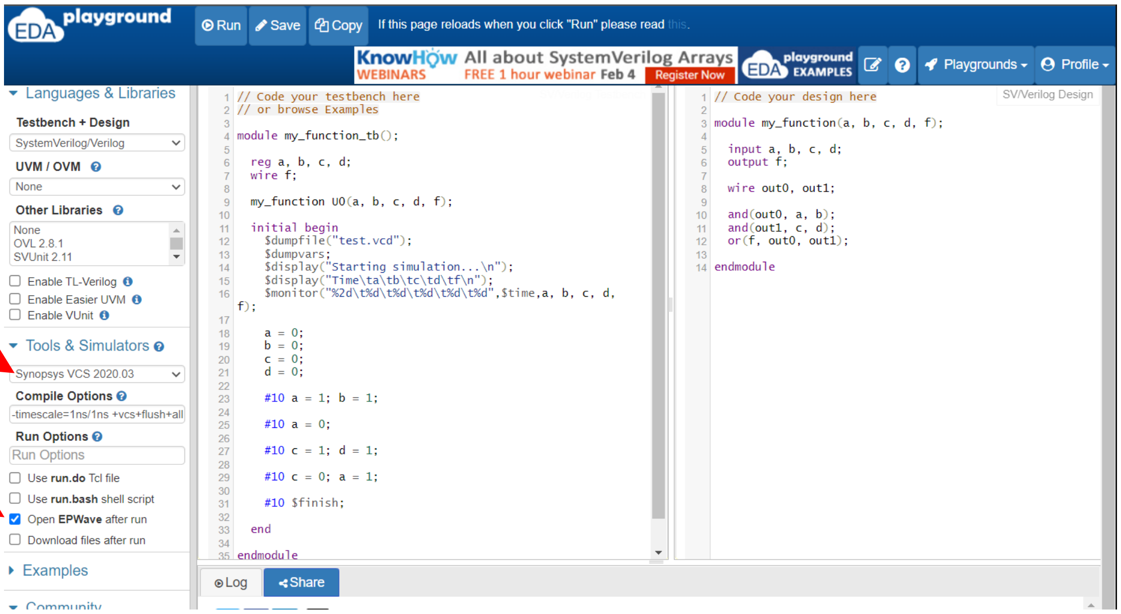
Here is a look at the output in the console and the waveform:

```
Chronologic VCS simulator copyright 1991-2020
Contains Synopsys proprietary information.
Compiler version Q-2020.03-SP1-1; Runtime version Q-2020.03-SP1-1;
Starting simulation...

Time    a       b       c       d       f

 0      0       0       0       0       0
10      1       1       0       0       1
20      0       1       0       0       0
30      0       1       1       1       1
40      1       1       0       1       1
$finish called from file "testbench.sv", line 31.
$finish at simulation time                   50
           V C S   S i m u l a t i o n   R e p o r t
Time: 50 ns
CPU Time:      0.420 seconds;       Data structure size:   0.0Mb
```



We can confirm that when a= b = c = d = 0, f = 0. When a = b = 1, f = 1, and when a changed to 0, f also changed to 0, as expected. When c = d = 1, the output was 1, and when c = 0 and a = b = 1, the output remained 1. Notice the time in the left column increments by 10 – this is because we delayed #10 time units between changing inputs. The simulation finished after 50 time units. We can also observe this behavior in the timing diagram / waveform:

Test this circuit yourself – create an account on edaplayground.com (it is free, but you must use a .edu email to access the commercial simulators like Synopsys VCS). Copy the module code into the right pane, and the testbench code into the left pane. Make sure to select Synopsys VCS from the Tools & Simulators menu, and check the “Open EPWave after run” option.



Testbench + Design: SystemVerilog/Verilog
UVM / OVM: None
Tools & Simulators: Synopsys VCS 2020.03
Compile Options: -timescale=1ns/1ns +vcs+flush+all
[✓] Open EPWave after run

```
// Code your testbench here
// or browse Examples

module my_function_tb();

  reg a, b, c, d;
  wire f;

  my_function U0(a, b, c, d, f);

  initial begin
    $dumpfile("test.vcd");
    $dumpvars;
    $display("Starting simulation...\n");
    $display("Time\ta\tb\tc\td\tf\n");
    $monitor("%2d\t%d\t%d\t%d\t%d\t%d",$time,a, b, c, d,
f);

    a = 0;
    b = 0;
    c = 0;
    d = 0;

    #10 a = 1; b = 1;

    #10 a = 0;

    #10 c = 1; d = 1;

    #10 c = 0; a = 1;

    #10 $finish;

  end

endmodule
```

```
// Code your design here

module my_function(a, b, c, d, f);

  input a, b, c, d;
  output f;

  wire out0, out1;

  and(out0, a, b);
  and(out1, c, d);
  or(f, out0, out1);

endmodule
```

Click Run at the top (it may ask you to save the file first), and you should see the output in the console. A new window should also open showing the waveform (if it does not, it might have been blocked by your browser). <u>Once you’ve confirmed this is working for you, you are ready to start the lab. You may complete steps 1-3 ahead of time.</u>

Lab Assignment Questions

In this lab you will design and implement a full adder circuit both in physical hardware, as well as Verilog, and confirm they are equivalent by checking outputs in both versions.

The full adder has three inputs, a, b, and c_{in}, and two outputs, sum and c_{out}.

1. (4 pts) Derive the truth table and obtain the MSOP expression for sum and c_{out} using a k-map for each output variable. You should have two expressions, one for sum and one for c_{out}.
2. (4 pts) Show how sum can be implemented using XOR gates, and c_{out} can be implemented as a nand-nand network.
3. (4 pts) This design can be optimized using Boolean algebra, but the derivation is somewhat complex (it requires adding in terms, distributing, and factoring multiple times). Instead, examine your truth table for c_{out}. What are the values of a, b, and c_{in} when $c_{out} = 1$?
    a. Hint #1: Recall that XOR can be written as A'B + AB'.
    b. Hint #2: You already implemented the sum function using XOR gates. Is there anything there you can reuse?
    c. Hint #3: You should only need a total of 2 XOR gates (74LS86) and 3 2-input NAND gates (74LS00) to implement your design.

Experimental Portion of the Lab

1. Confirm your circuit design from the part above with your lab TA, then construct the circuit on your breadboard. Place LEDs at the circuit outputs to confirm functionality. Be sure to connect the LED properly.
2. Demonstrate the circuit to your lab TA by testing different combinations of a, b, and c_{in}.
3. Implement the same optimized circuit using Verilog HDL, as discussed in the prelab. Implement a testbench that exercises all inputs. Confirm functionality by running the simulation and observing the output in the console as well as the waveform.

Please follow the guidelines under the Canvas module "Report Guidelines" for preparing a report. In your report, include/discuss the following:

1. Answers to the lab assignment questions.
2. Code for the module/testbench, as well as console output and waveform for the experimental portion of the lab.