Advanced VLSI Verification Exercise #1

Exercise goals:

1. Get used to the course's tools
2. Demonstrate a simple design and testbench

Setting up a Project

1. Set up a new project, using the instructions from **Exercise Setup**

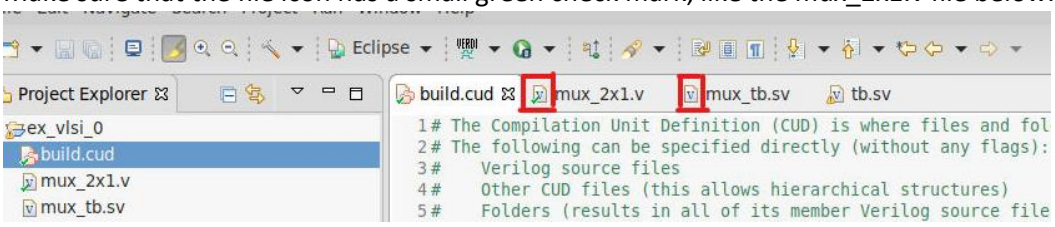
First Project – A Mux

You are now going to write your first design module, with Verilog language, and add it to the CUD. It's going to be a simple 2x1 Mux.

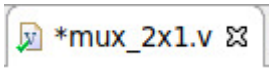
1. Select **File → New → File**
   Select your project folder, to mark it as the parent folder for the new file. Name the file **mux_2x1.v**. Select Finish.

2. In the main editor area, a new tab with your new file should pop up. Get back to build.cud, and under all the comments, add a line containing the name of your module **mux_2x1.v**. Save (Ctrl + S).

   Make sure that the file icon has a small green check mark, like the mux_2x1.v file below.



   This mark means that this file is being compiled live on every change. Note that we have another file, mux_tb.sv (you will write it soon), which doesn't have this check mark (yet) – it means that it's not part of the build.cud, and it will not show many warnings/errors.

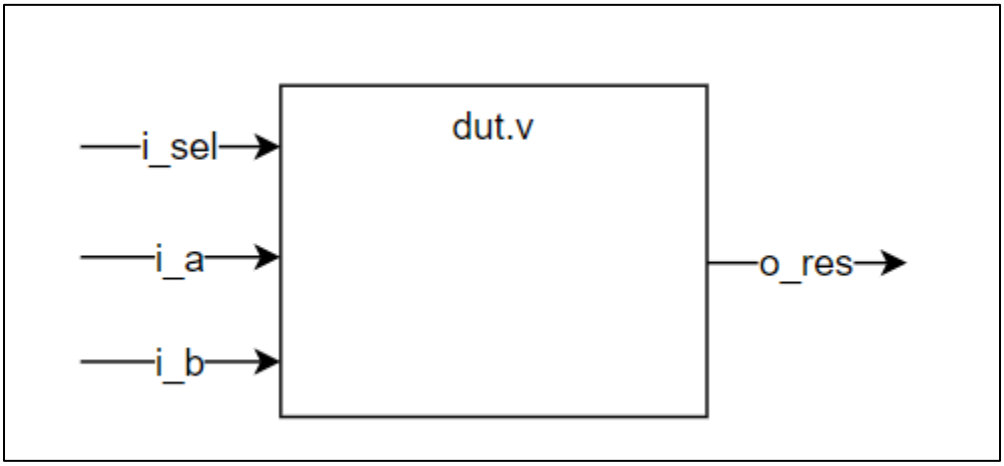
   In addition, make sure to save your file often while you edit it, since un-saved files sometimes do not show all the warnings/errors. You can see that the file was not saved since the last change if there's an asterisk next to it, like so: 

3. Open your **mux_2x1.v** file, and start writing the following code:

```
module mux_2x1 (
      input i_sel,
      input i_a,
      input i_b,
      output o_res);
```

Press Enter, and see that Euclide automatically appends `endmodule` to the end of your code. This is mandatory Verilog syntax. Every module must be declared with the keywords module and endmodule. Luckily, Euclide helps us with some syntax shortcuts like this one.

We have created a module with the module name **mux_2x1** and declared its ports. A block diagram of it would look like so:



For now, the module is empty, and it doesn't have any logic. The requirement is to have `o_res` equal to `i_a` if `i_sel` is 0, and `i_b` if `i_sel` is 1.

4. To implement the Mux logic, add the following line of code, above `endmodule`:

```
assign o_res = (i_sel) ? i_b : i_a;
```

Please write this line as it is, even if you think it's wrong.
This line implements the combinatorial logic of the Mux, by assigning the o_res output a value; either i_a or i_b, depending on the value of the i_sel. It's OK if you don't understand the syntax completely yet.

Congratulations, you just wrote your first Verilog module 😊 and thanks to Euclide, it's probably syntax-error free.

In order to verify that this Mux works properly, we have to create a testbench file, in which we will instantiate it, drive its inputs and observe its outputs.

5. Repeat steps 1 - 2 for a new file with the name **mux_tb.sv**. Note the .sv extension, denoting a SystemVerilog module (SystemVerilog will be our HVL). **Make sure to add mux_tb.sv to the build.cud file!** Verify it with the green check mark.

6. Similar to step 8, start writing `module mux_tb;` and press enter to let Euclide finish the module declaration. Now, let's declare the testbench variables, which we will later connect to

the Mux design, and drive with actual values (0s and 1s) to test the Mux behavior under different inputs.

7. Inside the mux_tb module, declare 4 variables like so:

```
logic i_a;
logic i_b;
logic i_sel;
logic o_res;
```

   Don't worry about the 'logic' keyword. You will learn about it next recitation. For now, think of this as just declaring the testbench variables. Also don't worry about the warnings – we will drive the variables with values soon.

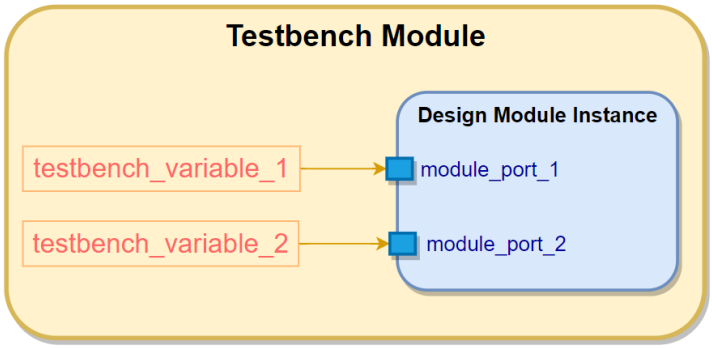
8. When we instantiate design modules inside testbench modules, we need to connect all the design module's ports to testbench variables. We will first describe the required syntax and show a block diagram, and then you will implement it for the Mux.

   The syntax for instantiating a module is:

```
module-name module-instance-name (.module_port_1 (testbench_variable_1),
                                  .module_port_2 (testbench_variable_2)
                                 );
```

   It can be a bit confusing, but essentially what happens is this:

   Take the module that has the name `module-name` (the name is defined inside the module file, see what you did in step 3), and create an instance of it with the name `module-instance-name`. Now, since the module usually has ports, we need to connect them to variables from our testbench (where we instantiate this module). This is done inside the parenthesis – each input/output (port) from the module is written, and next to it – the variable that it should connect to.



9. Follow the diagram and the syntax example above and write the instantiation for the mux_2x1 module.

   If it's getting difficult, press Ctrl + Space for Euclide 'auto-complete' syntax suggestions.

   After you're done, there should be no errors. It's possible that there are warnings.

The testbench code we wrote until now 'builds' the testing environment. But we didn't actually test anything yet. To do that we must actually define a sequence of operations over time. This can be done in Verilog/SystemVerilog using an 'initial' block. 'Initial' is like main() function in C – it's the entry point of the program, and it's the point from which simulation begins.

We will write the code for a sequence of input cases, and then run the simulation and observe the waveform.

10. Below your instantiation, start writing:
    `initial begin`
    Press Enter, and see Euclide add the mandatory 'end'.

    Also, above the 'end', write `$finish();` This will end the simulation. Between the begin and $finish(), we want to actually test some inputs to the Mux.
    Start by initializing the variables with 0, like so:

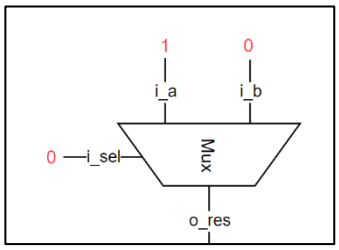
    ```
    i_a = 0;
    i_b = 0;
    i_sel = 0;
    ```

    All of the initializations above occur at the same simulation time 0. Make sure they are written below the 'initial begin' and above the '$finish();'.

    Now we want to advance the simulation time, and as time progresses, to test different input cases. We can advance simulation by using #n, where n is the number of simulation time units to advance.

11. Under the variable initialization, write `#10` to advance simulation by 10 time units.

12. Now, let's test the following case:



To do that, change the value of i_a from 0 to 1 and advance the simulation time again:

```
i_a = 1;
#10;
```

Note that we didn't change i_sel and i_b, they keep their values from before.

13. Repeat steps 11 - 12 with all the different truth table input cases for a 2x1 Mux. For every case, change only the required inputs. We wrote the code for two of them ([0, 0, 0], [0, 1, 0]), there are 6 remaining. In addition, print the o_res variable after each case using the $display function.

    Remember that we want to check that the design behaves like so:

    If i_sel = 0, the output o_res should be equal to i_a.

    If i_sel = 1, the output o_res should be equal to i_b.

Simulating the Design

1. It's time to actually simulate the design and testbench using our simulator VCS.
2. Select **Run → External Tools → Test**. See that there aren't any syntax errors, and if there are, fix them (consult the forum, google or ChatGPT).

   Once again, some information is printed to the console. You should see the following:

```
$finish called from file "mux_tb.sv", line 24.
$finish at simulation time                   30
           V C S   S i m u l a t i o n   R e p o r t
```

3. Now, let's see the actual waveform of our simulation. Select **Run → External Tools → Waves**

QUESTION

If the design behaves as expected, show screenshots of all the input cases on waveform.
If the design does not behave correctly, show a screenshot proving that.
