**CST 133**

**Digital Electronics Design with Verilog**

**Lab 2 - Structural Design of a 4-byte - 2:1 Multiplexer**

**Objective:**

**The objective of this lab is to design a simple structured circuit that can multiplex four - hex value using a 4-bit wide 2:1 Multiplexer by means of Verilog’s structural design method. The circuit to implement the multiplexer module is comprised of four – four to one multiplexers in parallel.**

**Challenge 1 - To Do:**

**You are to create all of the structural sub-modules for the four-bit 2:1 multiplexer. The block diagram of the top level module for this multiplexer is shown below:**

A

0

A

1

A

2

A

3

B

0

B

1

B

2

B

3

SEL

4

–

WIDE

2

:

1

MUX

M

0

M

1

M

2

M

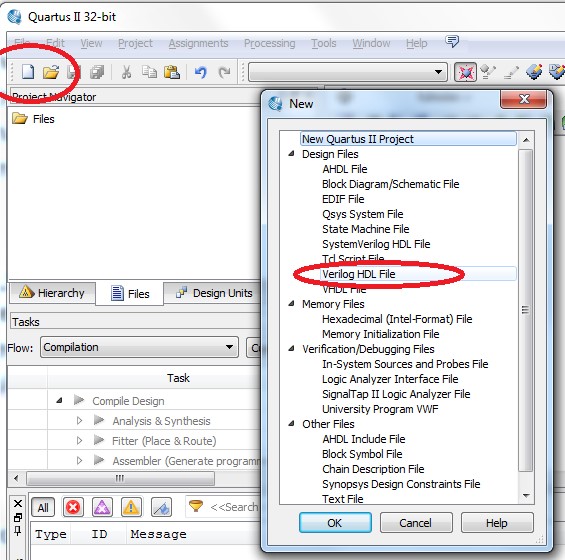
3

**The four-bit 2:1 multiplexer will have two - four bit inputs hex numbers and will produce a four bit multiplexed output. The four input bits are A[3:0], B[3:0], and the output is M[3:0]. The Select line is the SEL signal.**

**Procedures:**

**Step 1.**

Create a new project called lab2 using the template project provided on the course website. Create a Verilog structural model for a 2:1 multiplexer within this project. Call it mux2\_1b.



This module receives three input signals ( a, b, sel ) and generates an output signal (x). Use Quartus II to compile your source code and correct any syntax errors. Simulate the design using Modelsim.

Consider that your file will probably start off like this:

module mux2\_1b(

input a,

input b,

input sel,

output reg x

);

//add code here

endmodule

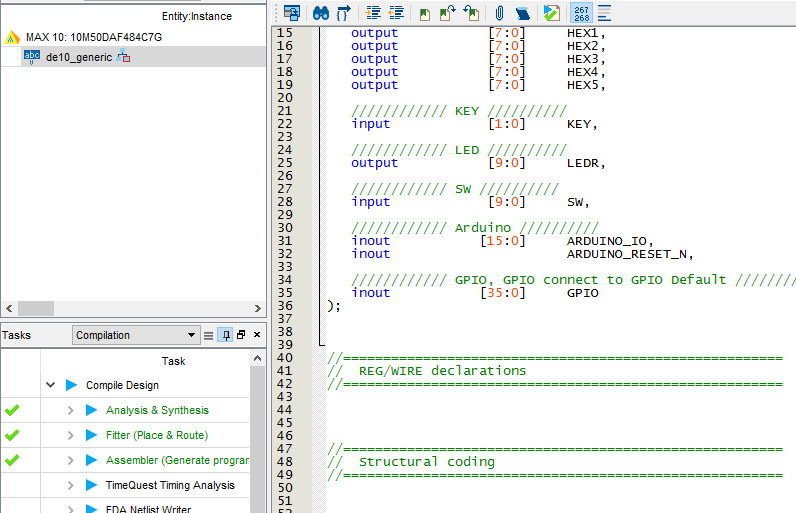
**Step 2.**

1. Use the 2:1 multiplexer you just developed to develop a structural model for a 4-bit wide 2:1 multiplexer.
2. Create another Verilog file and call it mux2\_4b.
3. What you should do within the mux2\_4b file is instantiate four copies of the mux2\_1b. Connect the four muxes with wires to produce 4-bit wide 2:1 multiplexer. **Refer to lab 1 if you need a refresher on how to do this.** This is an example of hierarchical design with reusable modules.

The mux2\_4b module receives input signals a[3:0], b[3:0], sel, and generates an output signals y[3:0]. Use the Quartus compiler to compile your source code and correct any syntax errors. Simulate the design using Modelsim.

**Step 3.**

1. Now, we must instantiate a copy of the mux2\_4b in your top level file (de10\_generic.v) and hook up the mux2\_4b inputs and outputs to switches and LEDs on the DE10-Lite board.
2. In the de10\_generic.v file, find the section that looks like the section circled in red. This is where we will code.



1. Instantiate a copy of mux2\_4b. Refer to lab1 for how to instantiate a copy of a module in another file.
2. Wire the copy of mux2\_4b to the switches as indicated below. Use the signal list below to determine pins for connection to your input and output signals (Only a suggestion)

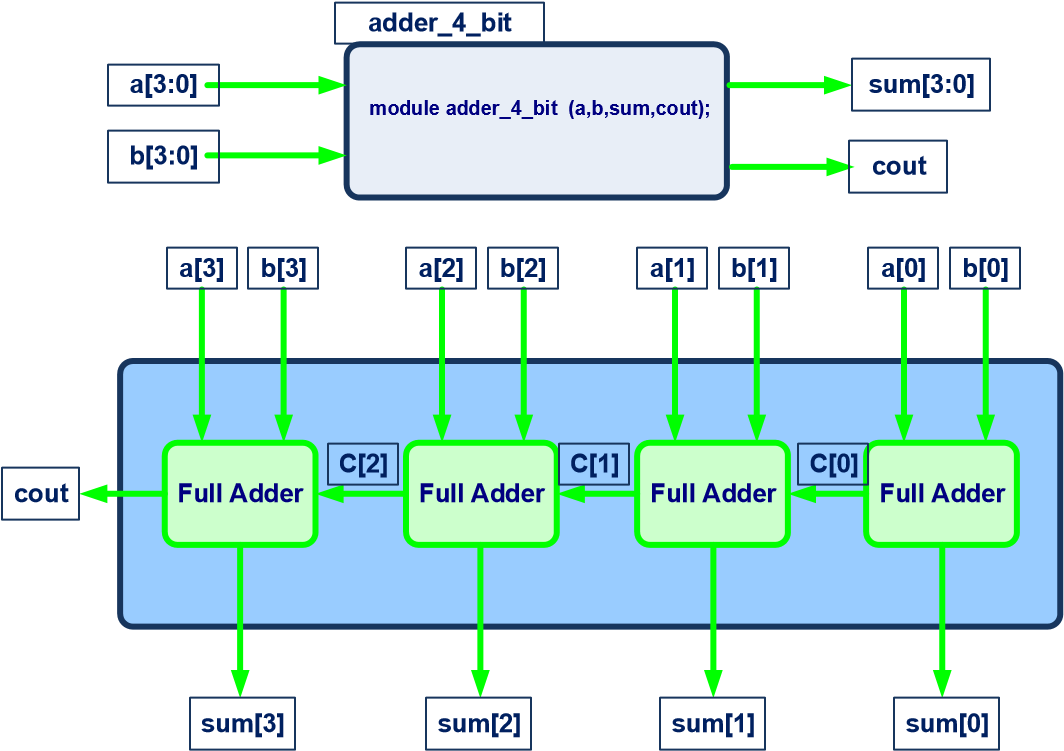
|  |  |  |  |
| --- | --- | --- | --- |
| Signal | Switch | Signal | LED’s |
| A[0] | **SW[0]** | Y[3] | LEDR[3] |
| A[1] | **SW[1]** | Y[2] | LEDR[2] |
| A[2] | **SW[2]** | Y[1] | LEDR[1] |
| A[3] | **SW[3]** | Y[0] | LEDR[0] |
| B[0] | **SW[4]** |  |  |
| B[1] | **SW[5]** |  |  |
| B[2] | **SW[6]** |  |  |
| B[3] | **SW[7]** |  |  |
| SEL | **SW[8]** |  |  |
|  | **SW[9]** |  |  |

1. Configure your design and download it to the DE10-Lite board. Perform a physical test on your design. Have your instructor check off this part of the lab before moving on.

**Challenge 2 -To Do:**

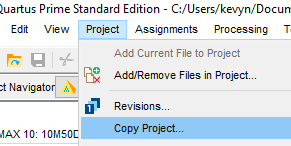
**You are to create all of the structural sub-modules for the four-bit adder. The block diagram of the top-level and hierarchy are shown in the diagram below:**

# – Bit Structured Adder



**Procedures:**

1. Take your lab1 project, copy it and rename it to lab2\_adder by going to Project > Copy Project. Make sure you select another directory.



Note that lab1 already has your 1-bit full adder design. For the first full adder in your chain, note that there is no carry-in. Just set the carry-in to 0. We need to instantiate multiple copies of the 1-bit full adder, we will be able to create the 4-bit full adder design shown above.

1. Create a new Verilog file called adder\_4b and instantiate four copies of the 1-bit adder. Wire them up as shown above. The module I/O should look similar to this:

module adder\_4b( input [3:0]a, input [3:0] b, output [3:0] sum, cout);

//instantiate modules and wires here.

endmodule

1. Simulate the design using Modelsim. Show that the design can add two numbers (Non-zero) and handle carry-out successfully in simulation.

1. Instantiate a copy of adder\_4b in your de10\_generic.v file (This is your top level file).

Configure your design and download it to the DE10-Lite development board and then perform a physical test on your design. Use the signal list below to determine pins for connection to your input and output signals (Only a suggestion.)

|  |  |  |  |
| --- | --- | --- | --- |
| **Switch 1** | **Signal** | **LED’s** | **Signal** |
| **SW0** | A[0] | LEDR[3] | sum[3] |
| **SW1** | A[1] | LEDR[2] | sum[2] |
| **SW2** | A[2] | LEDR[1] | sum[1] |
| **SW3** | A[3] | LEDR[0] | sum[0] |
| **SW4** | B[0] | LEDR[4] | cout |
| **SW5** | B[1] |  |  |
| **SW6** | B[2] |  |  |
| **SW7** | B[3] |  |  |
|  |  |  |  |

1. **Go to Tools -> Netlist Viewers -> RTL Viewer. Take a screenshot of the schematic.**

**Challenge 3- Coding it another way:**

1. Comment out your adder\_4b.

2. Create another Verilog module called adder\_4b\_v2 with the same inputs and outputs.

3. In this module, use an assign statement and the ‘+’ operator to add numbers. You’ll still need carry in and carry out. Figure out how to do that as well.

4. Once you’re done, test it on the board.

5. Go to Tools -> Netlist Viewers -> RTL Viewer. Take a screenshot of the schematic**.**

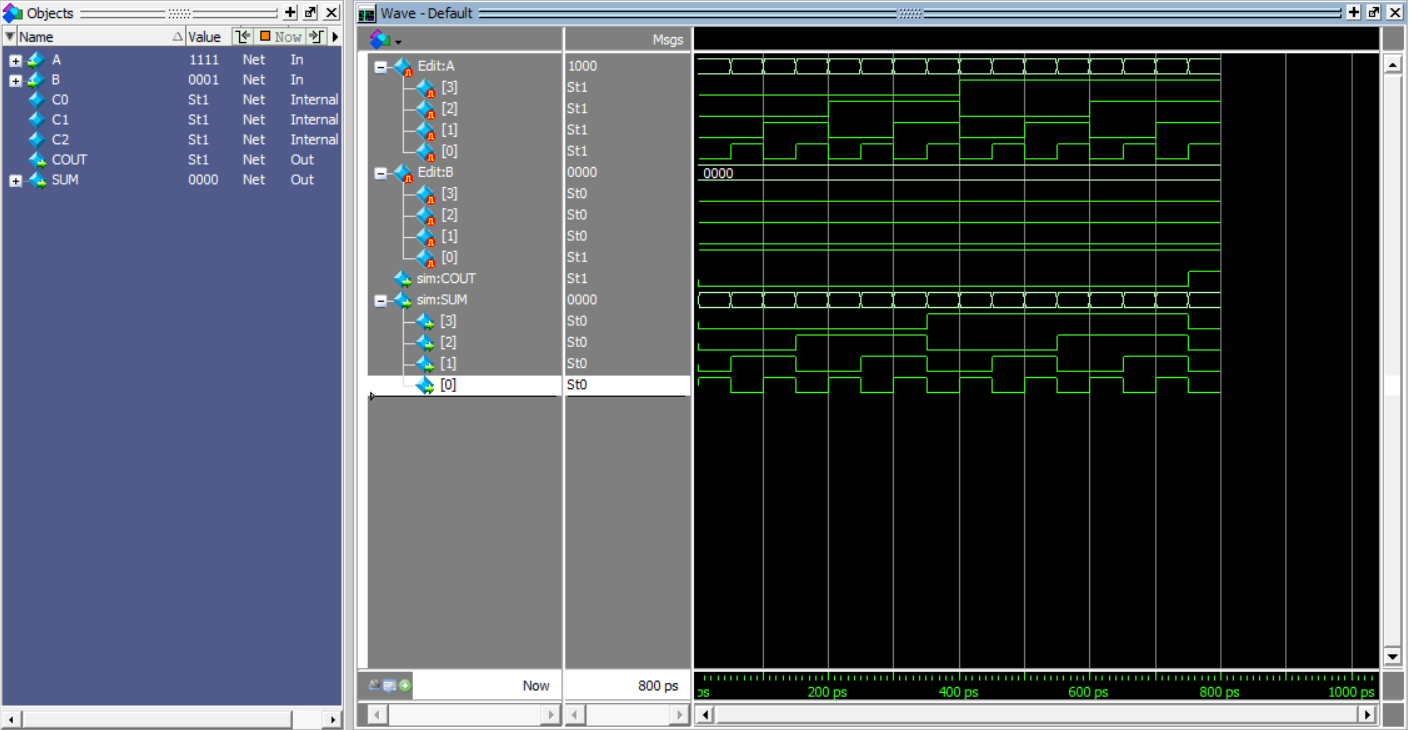
**Submission**

Submit the following:

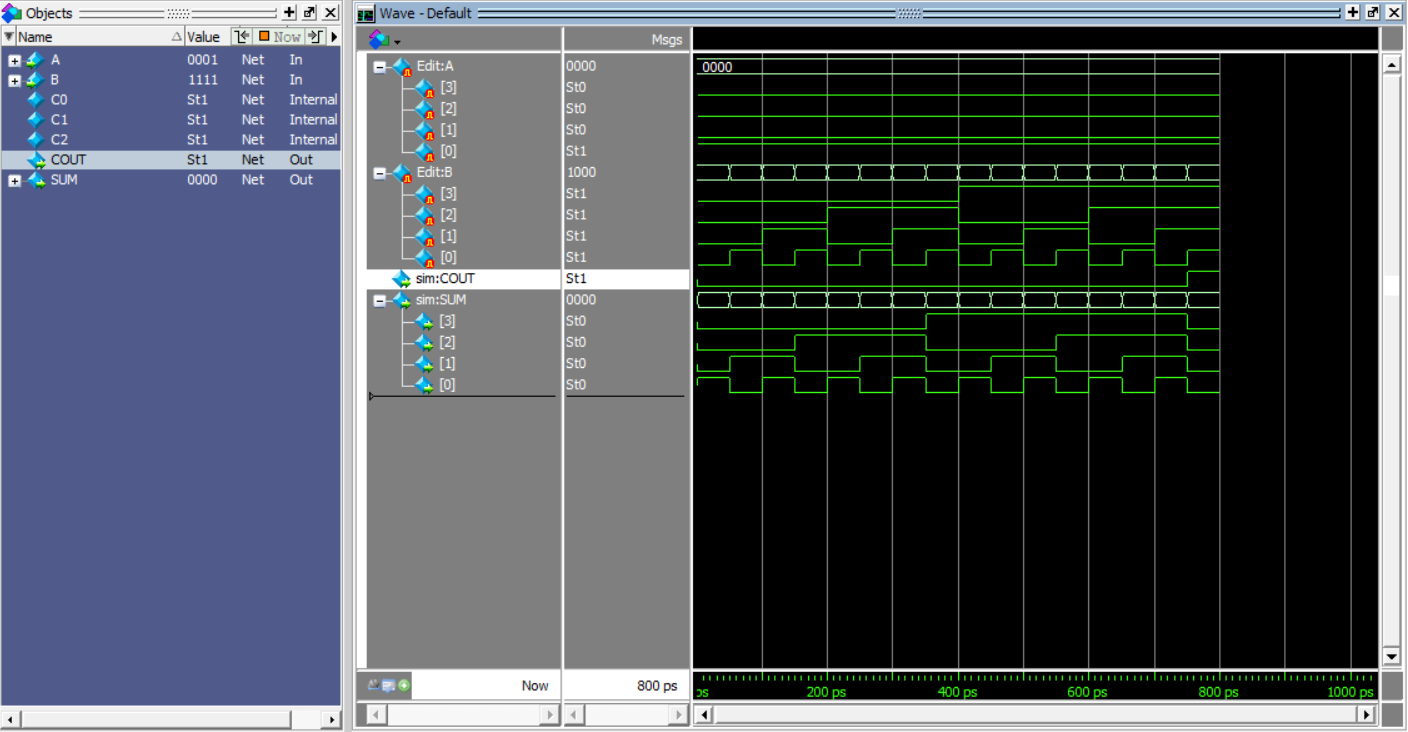
1. Zipped project file. Remember to perform a clean project before zipping up the project.

1. Simulation screenshot showing carry-out and addition functioning correctly.

**A counts from 0000 to 1111, B stays at 0001**

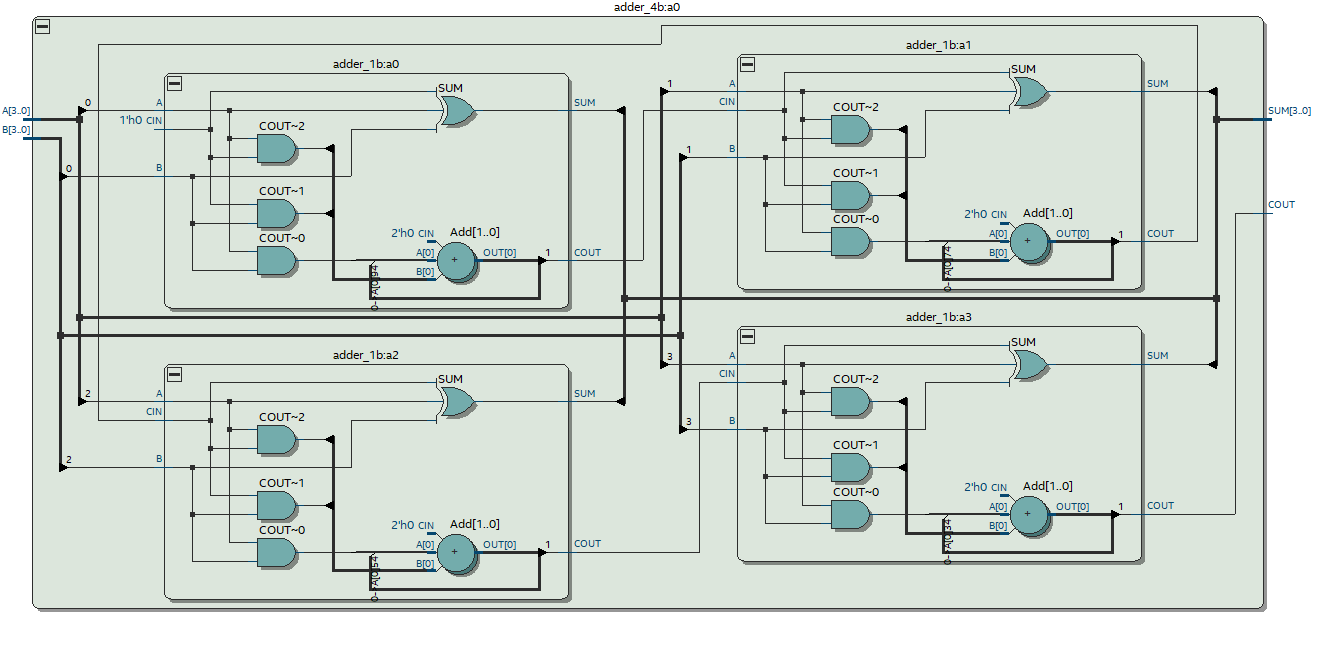


**Below: now *B* counts from 0000 to 1111, and *A* stays at 0001; same SUM/COUT values**

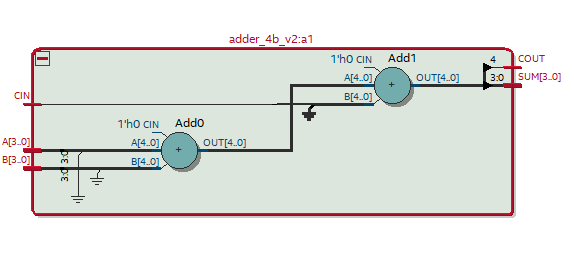


1. RTL diagrams of the 4-bit structured adder and the adder you formed the second way.

**4-BIT STRUCTURED ADDER**



**ADDER A SECOND WAY**



1. Are there any differences between the two RTL diagrams? Which would you assume is the better method?

**Method 2 appears to be significantly simpler (maybe my own fault) as it has far fewer gates involved, which might lead me to use it more often. The second method didn’t require the creation of a 1-bit full adder submodule, as Verilog simply takes care of it for the user.**