CSE4010 Lab01
Verilog HDL
Adders and Subtractors

Objectives

After this lab experiment you should be able to:

1. Build an adder-subtractor using a ripple-carry adder and XOR gates
2. Build hierarchical designs using Verilog
3. Perform addition of unsigned and signed 2's complement numbers

Preparation

1. Review binary signed and unsigned addition and subtraction: Then answer the question blow:
    - https://www.electronics-tutorials.ws/binary/signed-binary-numbers.html
    - **Question**: Add and subtract the following numbers using 2's complement if signed . Determine when Overflow and Carry out should occur. Show all steps you performed, do not just give the answer, show your steps. **Find A+ B, and A – B for all values .**
      Example: A = 6, B =1 , then Sum = A+B = 7 , and Diff = A-B = 6-1= 5 . Convert these numbers to 5- bit binary numbers and then perform addition and subtraction.

Table 1 Addition and subtraction of 5-bit numbers A, B

A	6	2	-3	-5
B	1	15	4	-12

2. Verilog HDL review: watch the following two videos
    - *Verilog HDL Basics.mp4*
    - *Verilog Test Bench.mp4*

3. Review Appendix A, B and C for more details

Assignment

1. Design and compile a full adder circuit in Verilog. Save the file as **fulladd.v**
2. Simulate **fulladd** using Xilinx ISE project Navigator or any similar simulator you are comfortable with (Icarus: http://iverilog.icarus.com/ , ModelSim: https://www.mentor.com/company/higher_ed/modelsim-student-edition).
3. Design a testbench to test the functionality of your fulladd, save the file as **tb_fulladd.v**
4. Design and compile a 5-bit adder using your full adder circuit (**fulladd)** in Verilog – Use the **Structural model** (review class notes on Verilog for example). Save the Verilog file named **adder5.v** in the same folder.
    - Make sure to add outputs for **Overflow**, and **CarryOut**
5. Simulate the design using tools in in 2 above and take screenshots of the

simulation. Use the values from table 1 above.

6. Design an **adder-subtractor** in a new Verilog file name **addsub.vhd**. Use Structural Model to connect to **adder5** created in Step 4. Add additional code to provide the subtraction; use an input AS to indicate add/subtract. (Remember we can use the XOR gate to take one's complement of the number B)
7. Compile and simulate **addsub** using the sets of values from Table1.
8. Determine when overflow should occur and verify this in your observations.
9. Include all the results and the design in your lab report
10. For each design simulation, you have to show your simulation results and your code and/or schematic in the lab report. (take a screenshot)

Deliverables:

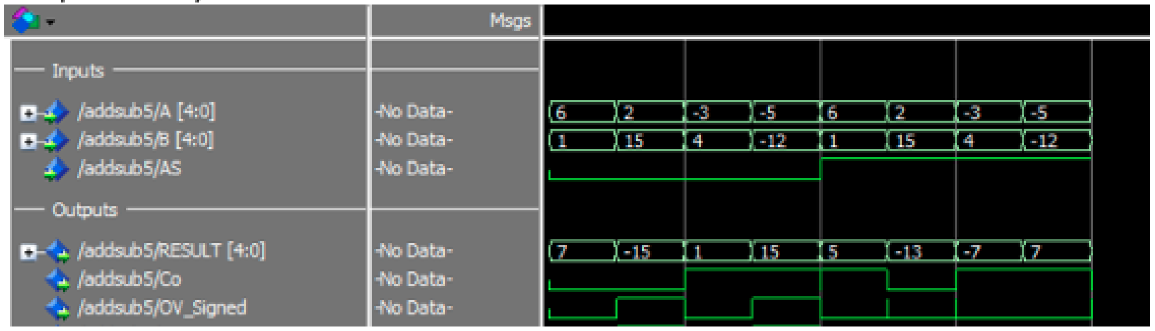
1. Create a report document , new file ***CSE4010Lab01Report.docx***
2. Complete the lab according to the instructions. Save the work to the document
3. Save the **Verilog** code you developed in the report document
4. Take screenshots of the simulation waveforms and insert them into the report document
5. Complete this report and upload it to Blackboard.
6. Submit all Verilog files you developed.

Grading

This lab is worth 100 points.

Appendix-A:

Sample run of your function **addsub.v** (this simulation was done in ModelSim)



Appendix-B:

- **Ripple Carry Adder**

To build a 4-bit ripple carry adder, four 1-bit full adders can be cascaded as shown in Figure 1.

- Each 1-bit full-adder adds one bit of X[3...0] to the corresponding bit of Y[3...0].
- The Sum bits S forms a part of the output and the generated carry bit, Cout, is fed into the next stage as shown in the circuit below.
- The full adder calculating the sum of the first bit (Least Significant Bit) gets a Cin of zero.

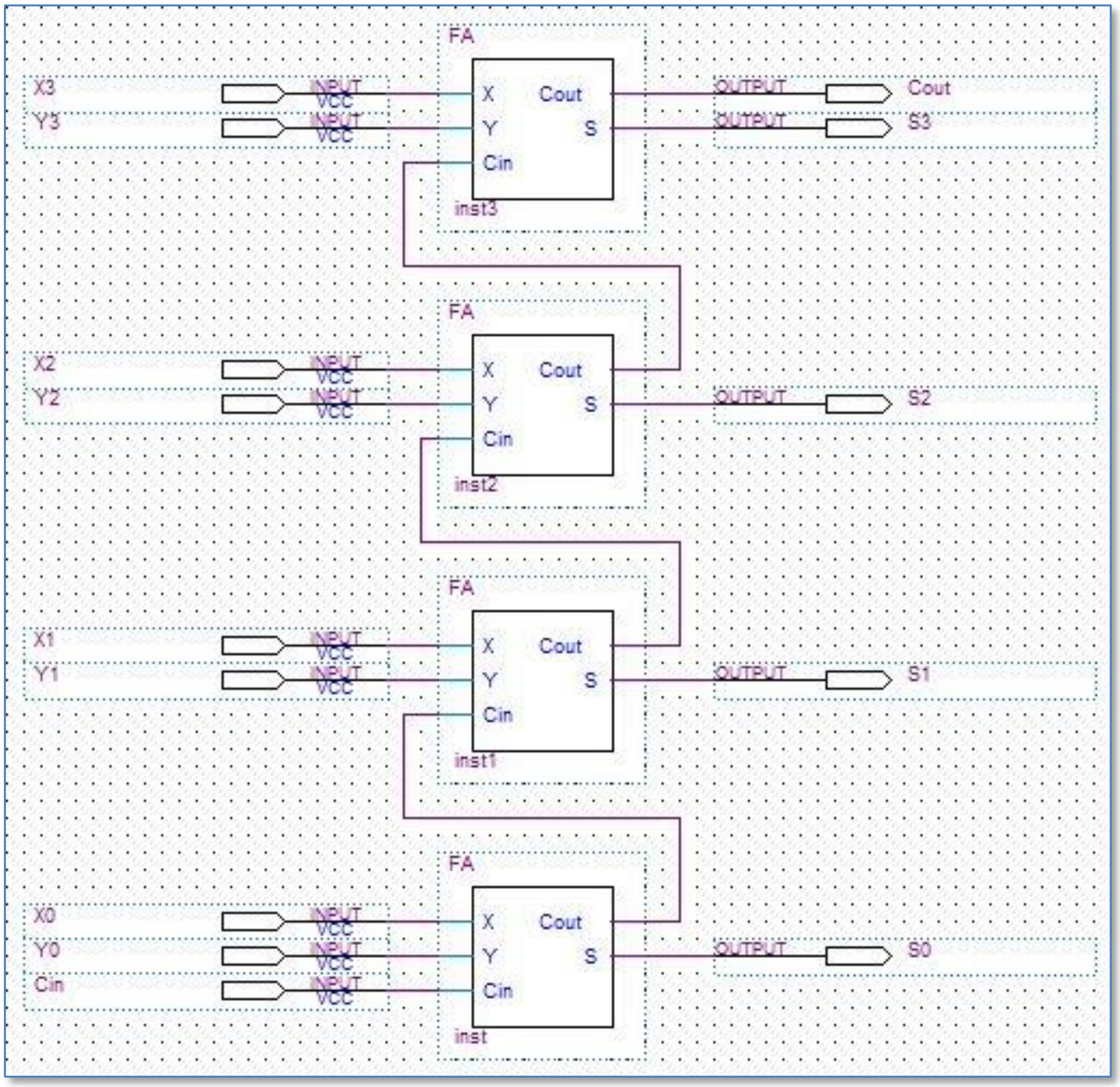


Figure 1 4-bit fulladder

- **Subtraction**

Subtraction is the same as 2's complementary addition. To compute X – Y, all we need to do is add the 2's complement of Y to X. The Ripple Carry Adder codified to accomplish this by adding 1's complement of Y input with a carry in bit = 1. A control input is used to specify the subtract operation. 1's complement of Y for subtraction is achieved by XORing each bit of Y with the control input (=1). The Control input is also connected to Cin of the adder. This is shown in Figure 2. When Control (Cin) is 0 the circuit adds X[3..0] to Y[3..0] and when Control is 1 it subtracts Y[3..0] from X[3..0].

- **Overflow:**

The result of arithmetic operations should fit in the bits available for output. If the result of an arithmetic operation doesn't fit into the bits provided, then an arithmetic overflow occurs. Overflow can be detected by comparing the carry-in and carry-out of the sign bit in the numbers.

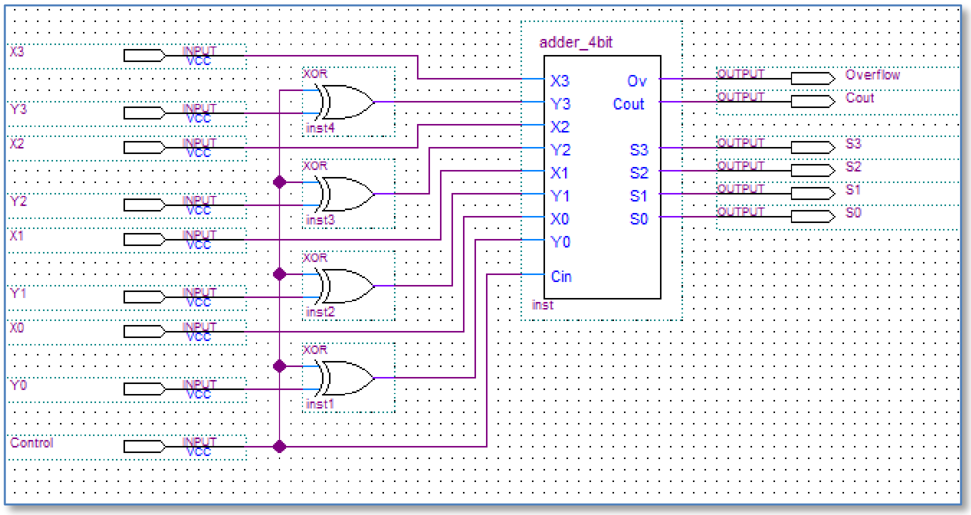


Figure 2 Adder/Subtractor

Appendix-C:
Verilog Testbench

A test bench is an HDL program used for describing and applying a stimulus to an HDL model of a circuit in order to test it and observe its response during simulation. The results of a test are only as good as the test bench that is used to test a circuit. Care must be taken to write stimuli that will test a circuit thoroughly, exercising all of the operating features that are specified. However, the test benches considered here are relatively simple, since the circuits we want to test implement only combinational logic. The examples are presented to demonstrate some basic features of HDL stimulus modules.
Test benches use the initial statement to provide a stimulus to the circuit being tested. The initial statement executes only once, starting from simulation time 0, and may continue with any operations that are delayed by a given number of time units, as specified by the symbol #. For example, consider the **initial** block

```
initial
        begin
                A _ 0; B _ 0;
            #10 A _ 1;
            #20 A _ 0; B _ 1;
        end
```

The block is enclosed between the keywords **begin** and **end** . At time 0, A and B are set to 0. Ten time units later, A is changed to 1. Twenty time units after that (at t = 30), A is changed to 0 and B to 1.

Inputs specified by a three-bit truth table can be generated with
the initial block:

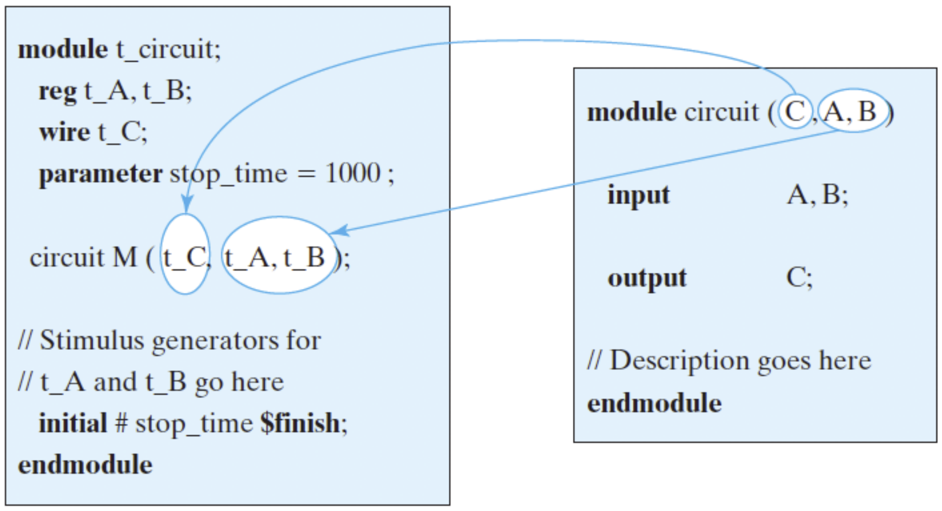
```
initial
        begin
            D = 3'b000;
            repeat (7)
            #10 D + D + 3'b001;
        end
```

When the simulator runs, the three-bit vector D is initialized to 000 at time = 0. The keyword repeat specifies a looping statement: D is incremented by 1 seven times, once every 10 time units. The result is a sequence of binary numbers from 000 to 111.
A stimulus module has the following form:

```
        module test_module_name;
        // Declare local reg and wire identifiers.
        // Instantiate the design module under test.
        // Specify a stopwatch, using $finish to terminate the simulation.
        // Generate stimulus, using initial and always statements.
        // Display the output response (text or graphics (or both)).
        endmodule
```

test module is written like any other module, but it typically has no inputs or outputs. The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local reg data type. The outputs of the design module that are displayed for testing are declared in the stimulus module as local wire data type. The module under test is then instantiated, using the local identifiers in its port list.

Figure 3 below clarifies this relationship. The stimulus module generates inputs for the design module by declaring local identifiers t_A and t_B as reg type and checks the output of the design unit with the wire identifier t_C . The local identifiers are then used to instantiate the design module being tested. The simulator associates the (actual) local identifiers within the test bench, t_A, t_B , and t_C, with the formal identifiers of the module (A, B, C). The association shown here is based on position in the port list, which is adequate for the examples that we will consider. The reader should note, however, that Verilog provides a more flexible name association mechanism for connecting ports in larger circuits.



```
module t_circuit;
  reg t_A, t_B;
  wire t_C;
  parameter stop_time = 1000 ;

 circuit M ( t_C, t_A, t_B );

// Stimulus generators for
// t_A and t_B go here
  initial # stop_time $finish;
endmodule
```

```
module circuit ( C, A, B )

  input        A, B;

  output       C;

// Description goes here
endmodule
```

Figure 3 Testbench and UUT

The response to the stimulus generated by the initial and al ways blocks will appear in text format as standard output and as waveforms (timing diagrams) in simulators having graphical output capability. Numerical outputs are displayed by using Verilog system tasks. These are built-in system functions that are recognized by keywords that begin with the symbol $. Some of the system tasks that are useful for display are:

$display —display a one-time value of variables or strings with an end-of-line return,
$write —same as $display , but without going to next line,
$monitor —display variables whenever a value changes during a simulation run,
$time —display the simulation time,

$finish —terminate the simulation.
The syntax for $display, $write, and $monitor is of the form

Task-name (format specification, argumentlist);

The format specification uses the symbol % to specify the radix of the numbers that are displayed and may have a string enclosed in quotes (“). The base may be binary, decimal, hexadecimal, or octal, identified with the symbols %b, %d, %h, and %o, respectively (%B, %D, %H, and %O are valid too). For example, the statement

```
$display ("%d %b %b", C, A, B);
```

specifies the display of C in decimal and of A and B in binary. Note that there are no commas in the format specification, that the format specification and argument list are separated by a comma, and that the argument list has commas between the variables. An example that specifies a string enclosed in quotes may look like the statement

```
$display ("time = %0d A =%b", $time, A, B);
```

and will produce the display

```
time = 3 A =  10 B= 1
```

where (time =), (A =), and (B =) are part of the string to be displayed. The format specifiers %0d, %b, and %b specify the base for $time , A and B , respectively.
In displaying time values, it is better to use the format %0d instead of %d. This provides a display of the significant digits without the leading spaces that %d will include. (%d will display about 10 leading spaces because time is calculated as a 32-bit number.)