Lab 6
Adder/Subtractor in Verilog

1. Objectives

- Familiarize students with Verilog language and FPGA CAD tools
- Design logic circuits using combinational building blocks
- Design and implement different types of adders
- Design a subtractor using an adder
- Simulate a Verilog design using a self-checking testbench
- Introduce timing analysis using CAD tools
- Compare the time delay and area consumption of different types of adders

2. Material

- Basys 3 board
- Xilinx Vivado software

3. Verilog Concepts

a. Generate block

Verilog provides **`generate`** statements to **produce a variable amount of hardware** depending on the value of a parameter. `generate` supports for loops and if statements to determine how many of what types of hardware to produce.

Suppose you need to implement an 8-input AND gate using 2-input AND gates. A possible design is shown in Figure 1 with the 8 inputs applied as an 8-bit vector `a` and with an internal 8-bit wire vector `x`. The module implementation is shown in Code Example 1.

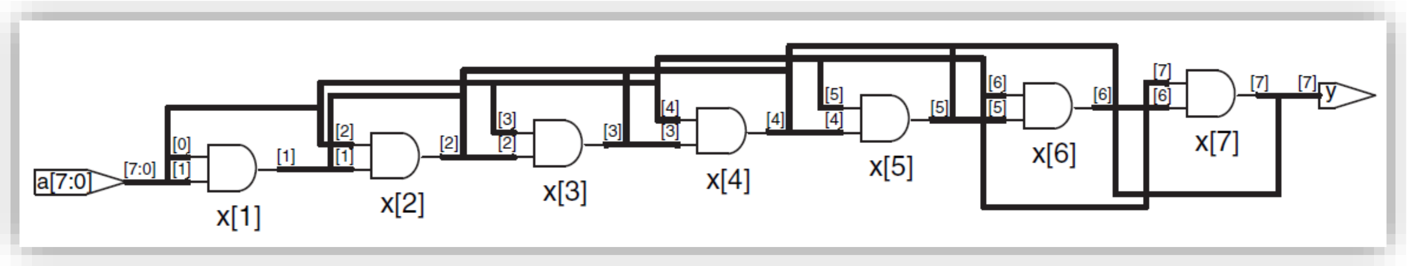

Figure 1. and8 synthesized circuit

The code of Example 1 is correct from a syntax point of view, however, it is not very clear. It can also be more tedious with an increasing number of instances (imagine a 32-input AND gate). This is why Verilog provides `generate` statements. Code Example 2 shows how to generate an 8-input AND gate using a `generate` statement.

Code Example 1: 8-input AND gate

```
module AND(input a, b, output c);
      assign c = a&b;
endmodule

module AND8 (input[7:0] a, output y);
      wire [7:0] x;

      assign x[0] = a[0];
      AND And1(x[1],a[1],x[0]);
      AND And2(x[2],a[2],x[1]);
      AND And3(x[3],a[3],x[2]);
      AND And4(x[4],a[4],x[3]);
      AND And5(x[5],a[5],x[4]);
      AND And6(x[6],a[6],x[5]);
      AND And7(x[7],a[7],x[6]);
      assign y = x[7];

endmodule
```

Code Example 2: 8-input AND gate using generate statement

```
module AND_usingGenerate(input[7:0] a, output y);
      wire[7:0] x;

      genvar i;
      generate
            assign x[0] = a[0];
            for (i=1; i<=7; i=i+1) begin: mygenloop
            AND myAND(x[i],a[i],x[i-1]);
            // or: assign x[i] = a[i] & x[i-1];
            end
      endgenerate

      assign y = x[7];
endmodule
```

Example 2 demonstrates how to use `generate` statements to produce an N-input AND function from a cascade of two-input AND gates. The for statement loops through i = 1, 2, … , 7 to produce many consecutive AND gates. The `begin` in a generate for loop must be followed by a : and an arbitrary label (`mygenloop`, in this case).

Of course, a reduction operator would be cleaner and simpler for this application, as seen below:

```
assign y = &a;
// &a is much easier to write than
// assign y = a[7] & a[6] & a[5] & a[4] & a[3] & a[2] & a[1] & a[0];
```

Yet the example is given to illustrate the general principle of hardware generators.

- Generate regions can only occur directly within a module, and they cannot nest.
- Whilst `if/else`, `for` and `case` statements cannot be normally used in Verilog, they can be used inside the generate block.
- In general, `generate` blocks are used for either:
  - Allowing the instantiation of Verilog modules multiple times (using for loop).
    - The index of the for loop must be defined as a `genvar` variable (`genvar i` in the previous example)
    - The for loop must have `begin/end` keywords.
  - Selecting one Verilog module from multiple modules (using `if/else` or `case` statements)
- A generate block is declared by `generate` and `endgenerate` keywords. These two keywords are optional like `begin/end`. However, it is a very good practice to include them for readability.
- A generate block should have a name (e.g. `mygenloop` in the previous example).

b. Self-checking testbenches

Verification consumes more than 50% of the development time of any digital system. Verification activities are performed at different steps during the design process. When you develop a Verilog module, you verify its functionality through simulation. For that, you develop a testbench and use it to apply different test cases.

A testbench is an HDL module that is used to test another module, called the device under test (DUT). Some tools also call the module to be tested the unit under test (UUT). The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output patterns are called **test vectors**.

There are several **types of testbenches** which include:

- **Simple testbench** (similar to the ones developed in the previous labs)
- **Self-checking testbench**
- **Self-checking testbench with test vectors**

In simple testbenches, the user must view the results of the simulation and verify by inspection that the correct outputs are produced. Testbenches are simulated the same as other HDL modules. However, they are not synthesizable. Checking for correct outputs is tedious and error-prone. Moreover, determining the correct outputs is much easier when the design is fresh in your mind; if you make minor changes and need to retest weeks later, determining the correct outputs becomes a hassle. A much better approach is to write a self-checking testbench.

Self-checking testbenches are implemented by **having a series of expected vectors**. These vectors are **compared at defined run-time intervals to actual simulation results**. If actual results match expected results, the simulation succeeds. If results do not match expectations, the testbench reports the discrepancies (Figure 2).

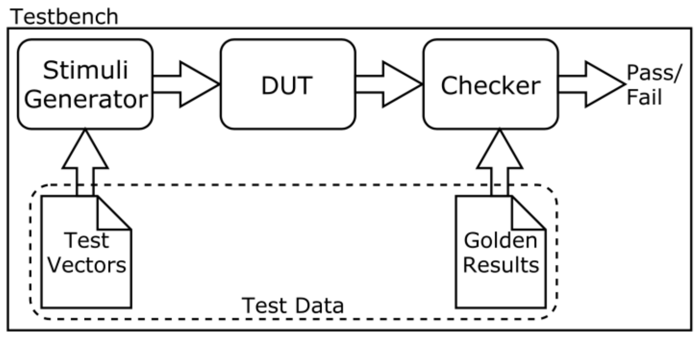

Figure 2. Self-Checking Testbench using Golden Vectors

Code Example 3: A self-checking testbench for the full adder

```
module full_adder_1_testbench();
reg x,y,cin;
wire s,cout;

// instantiate device under test
full_adder_1 dut(.a(x),.b(y),.cin(cin),.sum(s),.cout(cout));
// apply inputs one at a time
// checking results
initial begin
{x, y, cin} = 3'b000;
#100
if(s == 0 && cout == 0)  $display("Test passed for 000");

{x, y, cin} = 3'b001;
#100
if(s == 1 && cout == 0)  $display("Test passed for 001");

{x, y, cin} = 3'b010;
#100
if(s == 1 && cout == 0)  $display("Test passed for 010");
…………

{x, y, cin} = 3'b111;
#100
if(s == 1 && cout == 1)  $display("Test passed for 111");

end
endmodule
```

Example 3 above creates a self-checking testbench that contains all the input combinations and checks on the results. The `if` statement checks if a specified condition is true. The `$display` system task prints a message on the Tcl Console.

An alternative method to **generate the truth table** is to use `for` loops as shown below for the previous example.

```
// generate truth table
integer i;
for ( i = 0; i < 8; i = i + 1 )
 {x, y, cin} = i;           #100
 // every 100ns set x, y, cin to the binary representation of i
end
```

Another way to implement the self-checking testbench is through the usage of a golden reference model. Golden reference models are high-level descriptions of a design and are used to compare to the results of the model under test during simulation. The usage of golden model is illustrated in Figure 3. An example of a self-checking test using a golden model is given in Example 4 for a full adder module.

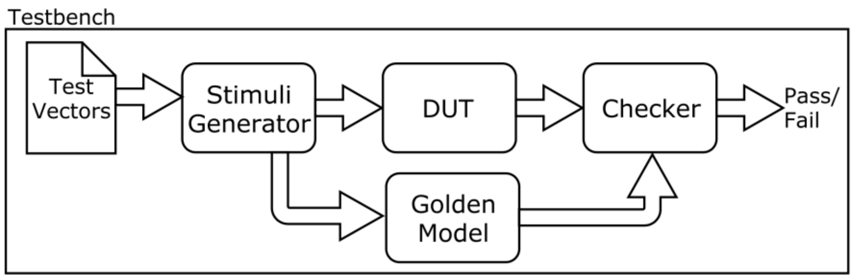

Figure 3. Self-checking Testbench using Golden Model

Code Example 4: A self-checking testbench for the full adder

```
module full_adder_1_testbench();
reg x,y,cin;
wire s,cout;
wire ref_s, ref_cout, err;
// instantiate device under test
full_adder_1 dut(.a(x),.b(y),.cin(cin),.sum(s),.cout(cout));

// Golden Model
assign {ref_cout, ref_s} = x + y + cin;

// The Checker
assign err = (ref_s != s) || (ref_cout != cout);
```

Code Example 4: (continued)

```
// Stimuli Generator
integer i;

initial begin
for ( i = 0; i < 8; i = i + 1 ) begin
     {x, y, cin} = i;         #10
     // every 10 ns set x, y, cin to the binary rep. of i
     if (err == 1)
         $display("Input combination %d failed.", i);
end

end
endmodule
```

4. Pre-lab Questions

1. Design a 1-bit full adder (FA) circuit
   a. Write the truth table
   b. Get the simplified function of the outputs using k-maps.
   c. Draw the circuit
2. Draw the block diagram of a 4-bit ripple carry adder using full adders. Show full adders in your diagram as black boxes only showing their inputs and outputs.
3. Draw the block diagram of an 8-bit ripple carry adder using full adders.
4. Draw the block diagram of an 8-bit carry select adder using 4-bit ripple carry adders.
5. Draw the block diagram of a 4-bit adder/subtractor circuit. The circuit should have an input (add/sub) that determines the operation (0: add, 1: subtract)

5. Experiments

Exp 1. [2 pts] Full Adder

Develop a Verilog module that implements the full adder using the Boolean expressions of the outputs (pre-lab Q1) (name it `full_adder_1`).

Exp 2. [2 pts] 4-bit ripple carry adder (4-RCA)

1. Develop a Verilog module that implements a 4-bit ripple carry adder (pre-lab Q2) (name it `rca_4`) using the `full_adder_1` full adder module developed in Exp 1.

2. Develop a self-checking testbench module to test your module and simulate it. The testbench should check on all input combinations and displays an error message on the Tcl console if the output of any combination is different than the expected output.

   Hint: describe your golden model in a functional description as follows
   ```
   assign {ref_cout, ref_s} = x + y + cin
   ```
   Hint: you have two 4-bit inputs and 1-bit `cin`, so you should have 512 input combinations.

Exp 3. [2 pts] 8-bit ripple carry adder (8-RCA)

1. Develop a Verilog module that implements 8-bit ripple carry adder (pre-lab Q3) using the `full_adder_1` module and `generate` blocks (name it `rca_8`).
2. Develop a self-checking testbench to test your module and simulate it.
3. Record for the 8-bit RCA the critical path delay and the area in terms of LUTs (check the Project Summary tab and observe the utilization and timing results).

Exp 4. [2 pts] 8-bit carry select adder (8-CSA)

1. Develop a Verilog module that implements an 8-bit carry select adder (name it `csa_8`) using three `rca_4` modules (pre-lab Q4).
2. Develop a self-checking testbench to test your module and simulate it.
3. Record for the 8-bit CSA the critical path delay and the area in terms of LUTs (check the Project Summary tab and observe the utilization and timing results).

Exp 5. [2 pts] 4-bit adder/subtractor

1. Develop a Verilog module that implements a 4-bit adder/subtractor (pre-lab Q5) (name it `addsub_4`), using the `rca_4` module developed in Exp 2.
2. Synthesize, implement, generate bitstream and program your design on Basys 3 board. Connect the two 4-bit module inputs to 8 sliding switches and the output result and carryout to LEDs.

Lab Report [10 Points]

1. [1.5 Pts] Provide all your Verilog design and testbench files and Vivado constraint files of Experiments 1 to 5.
2. [1.5 Pts] Report the values that you got in experiments 3 and 4 in the following table and comment on the results.

	8-bit RCA	8-bit CSA
Delay (ns)		
Area (LUTs)		

3. [2 Pts] Design a 12-person YES only voting system (i.e., counting the number of persons voting yes) using multiple 1-bit full adders, and 4-bit full adders. The system has binary inputs, such that each person can vote using a switch, where "1" implies voting "YES", and "0" implies that the person will not participate in the voting. Draw a neat diagram for the circuit.

   Hint: An efficient design of the system would contain 3 stages. The first stage has multiple 1-bit full adders, the second stage contains multiple 4-bit full adders, and the third stage contains a single 4-bit full adder (unused MSB inputs of 4-bit full adders may be connected to "0" or ground).

4. [2.5 Pts] Write a Verilog module for the 12-person voting system using generate blocks, utilizing the "full_adder_1", and "rca_4" modules implemented in the lab.

5. [2.5 Pts] Develop a self-checking testbench to test your 12-person voting system module and simulate it. Provide a screenshot of the expected output.