# CS 207 Digital Logic - Spring 2019
# Lab 2 - Verilog for Boolean Algebra

Monday, Feb. 25, 2019

## 1 Experiment A

In the lecture we learnt the basics of Boolean algebra. Boolean operations have some properties, e.g., commutative, associative, distributive, DeMorgan, and absorption.

### 1.1 Testing Commutative Property

In the last lab session, we came across two different kinds of HDL design: gate-level and data-flow modeling. Either one can be adopted to test the commutative property of binary OR and AND: $x + y = y + x$ and $xy = yx$:

commut.v

```
module commut(input A, input B, output C, output D);
wire A, B, C, D;
and and_1(C, A, B);
or  or_1 (D, A, B);
endmodule
```

commut_tb.v

```
module commut_tb();
reg A, B;
wire C_1, C_2, D_1, D_2;
commut commut_1(.A(A), .B(B), .C(C_1), .D(D_1));
commut commut_2(.A(B), .B(A), .C(C_2), .D(D_2));

//-- Here we use a new type of (co)-routine called task.
task check_equal;
begin
    if (C_1 != C_2)
        $display("---->ERROR! C_1 != C_2");
    if (D_1 != D_2)
        $display("---->ERROR! D_1 != D_2");
end
```

```
endtask

initial begin
    $dumpfile("commut_tb.vcd");
    $dumpvars(0, commut_tb);
    //-- $monitor prints the value of these variables upon
        ↪ changes
    $monitor("A is %b, B is %b, C_1 is %b, C_2 is %b, D_1 is
        ↪  %b, D_2 is %b.", A, B, C_1, C_2, D_1, D_2);
    # 5 A = 0; B = 0;
    # 5 check_equal;
    # 5 A = 0; B = 1;
    # 5 check_equal;
    # 5 A = 1; B = 1;
    # 5 check_equal;
    # 5 A = 1; B = 0;
    # 5 check_equal;
    # 5 $display("Simulation finished!");
    # 10 $finish;
end
endmodule
```

In this test, we use a new set of keywords related to task. Tasks are used in all programming languages, generally known as procedures or subroutines. Data is passed to the task, the processing done, and the result returned. They have to be specifically called, with data ins and outs, rather than just wired in to the general netlist. Included in the main body of code, they can be called many times, reducing code repetition.

Tasks are defined in the module in which they are used. It is possible to define a task in a separate file and use the compile directive `'include "task_file.v"` to include the task in the file which instantiates the task. Tasks can have any number of inputs and outputs. The variables declared within the task are local to that task. The order of declaration within the task defines how the variables passed to the task by the caller are used. Tasks can take, drive and source global variables, when no local variables are used. When local variables are used, basically output is assigned only at the end of task execution.

Besides the adopted usage, task can also have the following definition with local inputs and outputs:

```
module simple_task();
task convert;
input [7:0] temp_in;
output [7:0] temp_out;
begin
    temp_out = (9 / 5) * ( temp_in + 32);
end
endtask
endmodule
```

And should be invoked by

```
convert (temp_a, temp_b);
```

The expected output of the commutative test is as follows:

```
$ vvp commut_tb.out
VCD info: dumpfile commut_tb.vcd opened for output.
A is x, B is x, C_1 is x, C_2 is x, D_1 is x, D_2 is x
   ↪ .
A is 0, B is 0, C_1 is 0, C_2 is 0, D_1 is 0, D_2 is
   ↪ 0.
A is 0, B is 1, C_1 is 0, C_2 is 0, D_1 is 1, D_2 is
   ↪ 1.
A is 1, B is 1, C_1 is 1, C_2 is 1, D_1 is 1, D_2 is
   ↪ 1.
A is 1, B is 0, C_1 is 0, C_2 is 0, D_1 is 1, D_2 is
   ↪ 1.
Simulation finished!
```

The results accord with the rule. Test passed!

### 1.2 Have a Try

Use gate-level model to verify the following binary operation properties:

- Associative: $x + (y + z) = (x + y) + z$ and $x(yz) = (xy)z$.
- Distributive: $x(y + z) = xy + xz$ and $x + yz = (x + y)(x + z)$.
- DeMorgan: $(x + y)' = x'y'$ and $(xy)' = x' + y'$
- Absorption: $x + xy = x$ and $x(x + y) = x$

## 2 Experiment B

### 2.1 Have a Try

Use gate-level model Verilog to verify the following sum-of-products and product-of-sums transformation:

- $(b+d)(a'+b'+c)$ and its sum-of-products form (Derive the sum-of-products form with a pen and paper first).
- $a'b + a'c' + abc$ and its product-of-sums form.

**Assignment**
Save the source code in `sop.v` (for sum-of-products) and `pos.v` (for product-of-sums), and the testbench in `sop_tb.v` and `pos_tb.v`, respectively. Save the

command line output in `pos_tb.log` and `sop_tb.log`, respectively. Assignment 1 requires the source and output files.

# 3 Experiment C

## 3.1 Have a Try

Implement a module that is equivalent to a two-input XOR using only AND, OR, and NOT gates with gate-level modeling. Test the module with all possible combinations of inputs.

**Assignment**
Save the source code in `xor.v` and the testbench in `xor_tb.v`. Save the command line output in `xor_tb.log`. Assignment 1 requires the source and output files.