Solutions to Assignment 1

4.3 and 4.4 Some examples of the code for a four-input XOR function whose input is `a[3:0]` and whose output is y are:

```
module sillyfunction
  ( input logic [3:0] a,
    output logic y ) ;

`define simplest

`ifdef simple

   assign y = a[3] ^ a[2] ^ a[1] ^ a[0] ;

`elsif forloop

   always_comb begin
      y = '0 ;
      for ( int i=0 ; i<$bits(a) ; i++ )
        y ^= a[i] ;
   end

`elsif simplest

   assign y = ^a ;

`endif

endmodule
```

The modified testbench (with original code shown in comments) is:

```
module testbench3 ;
   logic clk, reset;
   // logic a, b, c, y, yexpected;
   logic [3:0] a ;
   logic y, yexpected;
   logic [31:0] vectornum, errors;
   // logic [3:0] testvectors[10000:0];
   logic [4:0] testvectors[10000:0];

   // instantiate device under test
   // sillyfunction dut(a, b, c, y);
   sillyfunction dut(a, y);

   // generate clock
   always
     begin
        clk = 1; #5; clk = 0; #5;
     end

// at start of test, load vectors
// and pulse reset
initial
  begin
     // $readmemb("example.tv",
     // testvectors);
     $readmemb("asg1-example.tv",
               testvectors);
     vectornum = 0;
     errors = 0;
     reset = 1;
     #27;
     reset = 0;

  end
// apply test vectors on rising edge
// of clk
always @(posedge clk)
  begin
     #1; // {a, b, c, yexpected}
         // = testvectors[vectornum];
     #1; {a[3], a[2], a[1], a[0],
          yexpected} = testvectors[vectornum];
  end

// check results on falling edge of clk
always @(negedge clk)
  if (~reset) begin // skip during reset
     if (y !== yexpected) begin // check result
        // $display("Error: inputs
        // = %b", {a, b, c});
        $display("Error: inputs = %b", a);
        $display(" outputs = %b (%b expected)",
                 y, yexpected);
        errors = errors + 1;
     end
     vectornum = vectornum + 1;
     // if (testvectors[vectornum]
     // === 4'bx) begin
     if (testvectors[vectornum] === 5'bx)
       begin
        $display
          ("%d tests completed with %d errors",
           vectornum, errors);
        $finish;
       end
  end
endmodule
```

The test vectors for all input combinations and one error is:

```
0000_0
0001_1
0010_1
0011_0
0100_1
0101_0
0110_0
0111_1
1000_1
1001_0
1010_0
1011_1
1100_0
1101_1
1110_1
1111_0
0001_0
xxxx_x
```

The result of running the testbench (with one error) is:

```
# vsim -batch testbench3 -do "run -all"
...
# Loading work.testbench3
# Loading work.sillyfunction
#
# run -all
# Error: inputs = 0001
#  outputs = 1 (0 expected)
#        17 tests completed with         1 errors
```

4.22 The code and testbench is:

```
module thermometer_code #(N)
  ( input  logic [   N-1:0] x,
    output logic [2**N-2:0] y ) ;
   always_comb begin
      y = '0 ;
      for ( int i=0 ; i<2**N-2 ; i++ )
        y[i] = i < x ;
   end
endmodule

module thermometer_code_tb ;
   logic [2:0] x = 3'b110 ;
   logic [6:0] y ;
   thermometer_code #(3) t0 (.*);
   initial begin
      #1 ;
      $displayb(y) ;
      $finish ;
   end

endmodule
```

The output of the testbench is:

```
# vsim -batch thermometer_code_tb -do "run -all"
...
# Loading work.thermometer_code_tb
# Loading work.thermometer_code
#
# run -all
# 0111111
```

4.48 and 4.49 The two modules are:

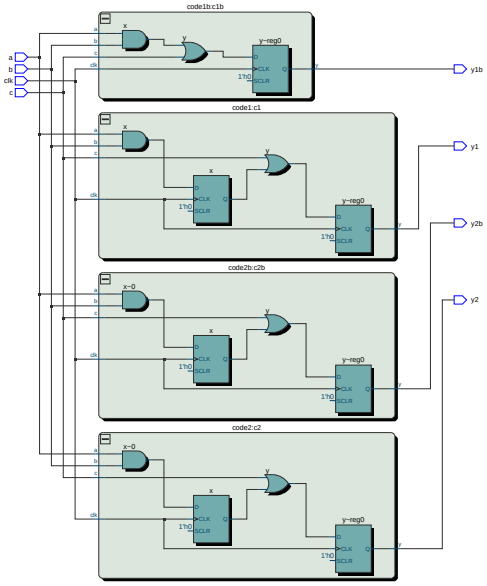
```
module code1(input logic clk, a, b, c,
             output logic y);
   logic x;
   always_ff @(posedge clk) begin
      x <= a & b;
      y <= x | c;
   end
endmodule

module code2 (input logic a, b, c, clk,
              output logic y);
   logic x;
   always_ff @(posedge clk) begin
      y <= x | c;
      x <= a & b;
   end
endmodule
```

The only difference between `code1` and `code2` is the order of the two assignments in the `always_ff` block.

When non-blocking assignments are used, the values of `x` and `y` don't change until the end of the always block. The schematics are the same since the values of `x` and `y` used are the values at the flip-flop outputs.

When blocking assignments are used, the values are updated immediately so in `code1` the value of `x` used in the second assignment is the and result, not the flip-flop output while the behaviour `code2` remains unchanged since the value of `x` is used before it is assignment to.

The generated schematics are as follows (the 'b' suffix indicates blocking assignments):



4.27 An example of a module and testbench for a JK flip-flop is:

```
module jkff ( input logic j, k, clk,
              output logic q ) ;
   always@(posedge clk)
     case({j,k})
       2'b00: q <=  q ;
       2'b01: q <= '0 ;
       2'b10: q <= '1 ;
       2'b11: q <= ~q ;
     endcase
endmodule

module jkff_tb ;

   logic j, k, clk, q;

   jkff j0 (.*) ;

   initial begin
      for ( logic [3:0] i=4'b0100 ;
            i<4'b1101 ; i++ )
        #1us {j,k,clk} = i[2:0] ;
      $stop ;
   end
endmodule
```

For which the simulation output is shown in Figure 1.

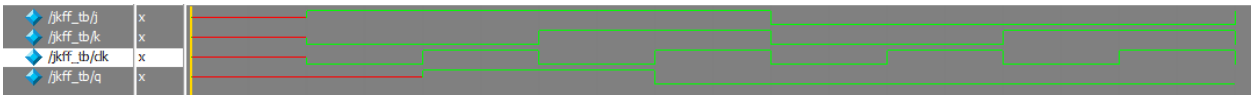


Figure 1: JK flip-flop simulation results.