CME341 Assignment 4

This assignment consists of 10 questions, of which the first 8 MAY be considered for marking purposes. Questions 9 and 10 are optional. However, it is recommended that students try working through the process described in Questions 9 and 10 for at least one of the circuits from Question 8.

1. The verilog description below is an example of how code can get butchered by an engineer with lazy or careless debugging habits. The lazy debugger wanted to try something and yet be able to easily reverse the change without thinking. To do this the engineer added a begin-end wrapper and then a second if-else statement. When the change worked the lazy debugger didn't delete the extra if-else statement.

   A U of S grad would have changed "a==4'b1111" to a==4'b0111" and not added the second if-else statement or the begin-end wrapper.

```
module if_else_combinational_logic(
    input [3:0] a, b,
    output reg [3:0] y );
      always @ *
      begin
         if (a==4'b1111)
             y=b;
         else
             y=~b;
         if (a==4'b0111)
             y=b;
         else
             y=~b;
      end
endmodule
```

   (a) Enter the Verilog HDL above and compile it. Does the compiler issue a warning about ignoring the first if-else statement?

   (b) Simulate the circuit and verify that the first if-else statement is ignored.

2. Modify the "if_else_combinational_logic" prototype given in question 1 by deleting the "begin" and "end" statements.

   (a) The modified prototype violates the rules for a procedure (i.e. always ⋯). What is the problem?

   (b) Compile the modified prototype and carefully read the error. Does the error message explain the problem?

   (c) Over the course of the year most students will encounter this error message several times, and not because they are lazy debuggers. Is the most likely reason:
   Reason 1: The student forgot to include the begin-end wrapper.
   Reason 2: The student wrote the body of procedure without putting the "always ⋯ " line at the top of it.

   (d) Is it good practice to routinely place begin-end wrappers around an always procedure? Could this mask some typos that the complier would otherwise flag (for example a sequence of nested if-else statements where an else has inadvertently been omitted)?

3. Generate a table that shows the output versus the 16 possible values of "select" for the module "case_statement_example" given below. For example the entry in your table for select=4'b1111 should be data_0.

```
module case_statement_example (select,
           data_0, data_1, data_2, data_3,
           data_out);

input [3:0] select;
input [1:0] data_0, data_1, data_2, data_3;
output [1:0] data_out;

reg [1:0]  data_out;

always @ (select or data_0 or
             data_1 or data_2 or data_3)
begin
   casex (select)  // case x is used when don't cares are
```

```
// used in the case item
   4'b1xxx : data_out = data_0; // if (select[3]==1)
   4'bx1xx : data_out = data_1; // else if (select[2]==1
   4'bxx1x : data_out = data_2; // else if (select[1]==1)
   4'bxxx1 : data_out = data_3; // else if (select[0]==1)
   default : data_out = 2'b00; // else
   endcase
end
endmodule
```

4. Compile the Verilog HDL in Question 3 and verify that the Quartus compiler handles the "don't cares" properly.

5. Compose a Verilog HDL module for a combinational logic circuit that increments an 8 bit input variable called "low_number" by either 1, 7, 255 or 249 when the value of a two bit variable, "select", is 0, 1, 2 and 3 respectively. The output variable, which is called "high_number", is an 8 bit vector. Compile and test the resulting circuit.

6. Compose a Verilog HDL module representing a combinational logic circuit that interleaves two 16 bit input vectors, A and B, to make one 32 bit output vector, C. That is, make C[0]=A[0], C[1]=B[0], C[2]=A[1], . . ., C[31]=B[15]. Please use a for loop in your design. Compile and test the resulting circuit.

7. (a) There is a typo in the program below. The "3'b1001" should be "4'b1001". (This happens from time to time.). The vector "a" does not respond to vector "b" as intended. With the typo, the value of "a" does not depend on b. Why?

```
module literal_ex (b, a);
input [3:0] b;
output [3:0] a;
reg [3:0] a;

always @ *
if ( (&b)==1'b1 ) a=3'b1001;
else a=4'b0001;
```

```
endmodule
```

(b) Compile the program. In the testbench make the signal `b` count from 0 to 4'b1111 in increments of 1 at intervals of 1 μs. This can be done with the verilog commands given below:

```
initial
   b=4'H0;
always
  #1000 b = b + 4'H1; // the delay of 1000 assumes
                      // a timescale of 1 ns
```

Simulate the program and verify that x=4'b0001 for all values of b.

(c) Determine how the compiler handles such typos?

8. Draw schematic diagrams for the sequential circuits described by the Verilog HDL segments given below. (Note that these descriptions are intended to demonstrate different constructs and are not necessarily good descriptions or useful circuits.)

(a)
```
input clk, toggle; output q;
always @ (posedge clk)
     if (toggle == 1'b1) q = ~q;
     else q = q;
```

(b)
```
input d, clk_enable, clk; output q;
always @ (posedge clk)
     if (clk_enable == 1'b1) q = d;
     else q = q;
```

(c)
```
input d, set, clear, clk; output q;
always @ (posedge clk)
     if (set == 1'b1) q = 1'b1;
```

```
          else if (clear == 1'b1) q = 1'b0;
          else q = d;
```

(d)
```
input d, set, clear, clk; output q;
always @ (posedge clk or posedge clear or posedge set)
     if (set == 1'b1) q = 1'b1;
     else if (clear == 1'b1) q = 1'b0;
     else q = d;
```

(e)
```
input clk, input [3:0] d; output [3:0] q; reg [3:0] q;
always @ (posedge clk)
   casex({d[3],d[1]})
   2'b00 : begin q[3]=1'b0; q[2:0] = q[2:0]; end
   2'b1x : begin q[3] = ~q[3]; q[2:0] = 3'b000; end
   default: q = d;
   endcase
```

(f)
```
input clk, input d; output [3:0] q; reg [3:0] q;
integer i;
always @ (posedge clk)
  begin
   for(i = 2; i >= 0; i = i-1)  q[i+1] = q[i];
   q[0] = d;
  end
```

9. (a) Generate structural descriptions, in Verilog, of your hardware schematics (i.e. use primitives for each hardware element in the schematic diagrams). Use DFFE (which is not a true Verilog primitive but is built into most compilers) to instantiate your flip/flops. This is a prototype for a d-flip/flop with enable and asynchronous clear and set. An example instantiation of DFFE is given below. Inputs `clrn` and `prn` are active low, input `ena` is active high. You

do not need to provide connections to inputs `clrn`, `prn` or `ena` if they are not used.

```
DFFE flop_1(
   .d(d),        // the d input
   .clk(clk),    // the clock, positive edge trigger
   .ena(enable), // if used, d goes to q on clock edge when ena is
                 // high  otherwise q does not change
   .clrn(clear_bar), // if used, clears q when low
   .prn(set_bar),    // if used, sets q when low
   .q(q)    // output
   );

// NOTE if clrn and prn are both active (i.e. both low) the
// output, q, is undefined.
```

(b) Compile your circuits and test them using the simulator. DFFE is a special Quartus primitive. For the simulator to properly simulate the .vo file a pre-compiled library named `alter_ver` must be include before the simulation is loaded. I.e. this library is included at the same time as the `cycloneive_ver` pre-compiled library is included.

Compile your circuits and test them using the simulator. Before compiling a design that uses flip/flops, you should enter some information on the clock used. If this is not done Quartus issues a warning:

"Critical warning (332148): Timing requirements not met"

This warning is issued for one of two reasons; (a) The propagation delay through a logic circuit that connects the output of one flip/flop to the input of another is too large to meet the set-up time for the D input. In essence this means the propagation delay exceeds one clock period so a change in a Q output does not reach the D input of the following flip/flop before the next positive clock edge. (b) The TimeQuest Timing analyser has not been told the period of a clock so it can not determine if timing has been met or not. If this is the cause of the warning there will be a companion warning: Critical Warning (332012): Synopsis Design Constraints File not found.

For the project at hand the companion warning is also issued. This means the second reason listed above is the cause of Critical warning (332148). In this case the way to remedy Critical warning (332148) is to generate a Synopsis Design Constraints (.sdc) file that contains information on the period of the clock. The .sdc file is generated by selecting **Tools** $\longrightarrow$ **TimeQuest Timing Analyser**. Once the Timing analyser is open select **Netlist** $\longrightarrow$ **Creat Timing Netlist**. Ensure"Post-fit" and"slow-corner" options are selected, then click "OK". Select **Constraints** $\longrightarrow$ **Create Clocks** to open the clock setting window. In this window there are 5 fields labelled: `Clock Name; Input Pin; Period; Rising; and Falling`. Only the first three fields need to be completed.

Clock Name: Enter another name for the clock, e.g. system_clock.

Target: This field is used to set the port pin that has the clock connected to it in the design. To find the port pin, in this case "clk", click the "..." button to the right of the `Target` field. A new dialog box should appear. To search for the "clk" signal in the design, ensure that the filter field contains a "*". Set the collection drop down box to "get_ports" to search the port list of your design's top level module. Click the "List" button. Once the"List" button has been clicked Time Quest should present a list of signals in the lower left hand box in the window. This box will list all possible port pins available in the design. Find and select the "clk" port pin and click the ">" button to set it as the target pin for the clock. Click "ok".

Period: Enter the period of the clock. The TimeQuest timing analyser uses this information to determine if the circuit the synthesizer produced will be able to run at the specified clock frequency.

Rising: Leave this field blank. It is used to specify a phase offset. To be specific, it specifies the time of the rising edge as a fraction of the period. The default setting is 0.0.

Falling: Leave this field blank. It is used to specify the duty cycle. To be specific, it specifies the time of the falling edge as a fraction of the period. The default setting is 0.5.

Once all the setting have been entered in the "create clock" window click "run". These settings will be put into a Synopsys Design Constraints (.sdc) file. The commands generated from the entries will be shown in the "SDC command" segment of the window, which is just beneath the "base clock setting" segment of the window. Observe and remember these commands (perhaps cut and paste this into a temporary file to avoid remembering the command).

To create the SDC file select **Constraints** $\longrightarrow$ **Write SDC file** and click "OK".

Close the TimeQuest window.

After finishing verify that the SDC script file was properly created. The is done by selecting **file** $\longrightarrow$ **open** and then in the "type of fil" box select "Script files (*.tcl *.sdc *.qip *.sip)". The file`counter.sdc` should appear in the directory window. Double click on `counter.sdc` to see its contents. This file can be created and edited directly without going through the TimeQuest Timing Analyser Wizard.

Add this file to your project.

Recompile the project.

10. (a) Take the behavioral descriptions from which you derived the schematics and flesh them out to complete modules. Things like module declarations have to be added and there may be syntax errors that need to be fixed.

    (b) Compile your descriptions and test the resulting circuits using the simulator.

    (c) Compare the results of the simulations of the behavioral descriptions to the results of the simulations of the structural descriptions to verify that your schematic diagrams are correct.