This paper contains 4 questions

Answer THREE questions.

An outline marking scheme is shown in brackets to the right of each question.

University approved calculators MAY be used.

A foreign language dictionary is permitted ONLY IF it is a paper version of a direct Word to Word translation dictionary AND it contains no notes, additions or annotations.

10 page examination paper.
Question 1.

(a) Explain the difference between “one-hot” state assignment and binary state assignment. Why is one-hot assignment preferred in FPGA designs? 

[8 marks]

(b) A state machine has two inputs, A, B, and one output, Z. If the sequence of input pairs: A=B, A=1 B=0, A=B is detected, Z becomes 1 during the final cycle of the sequence, otherwise the output remains at 0. (Note that one sequence can overlap with the next sequence.) Draw an ASM diagram to describe this system.

[10 marks]

(c) Write a SystemVerilog model of this ASM diagram. Your model should include the clock and an active low asynchronous reset.

[15 marks]
Indicative Solution for Question 1.

a) “One-hot” means one flip-flop per state. Exactly one flip-flop output is asserted in each clock cycle. Binary encoding means that the states are mapped to a minimum number of bits. The one-hot style is preferred in FPGAs because FPGAs are rich in flip-flops, but relatively poor in combinational logic. Binary encoding can require a lot of combinational logic. Using the combinational logic in a CLB means that the flip-flop cannot be used for anything else. At best, therefore, binary encoding uses the same number of CLBs as one-hot, but the flip-flops are wasted. In the worse case, more CLBs are used.

b)
c)

module asm (input logic n_reset, clk, a, b,
    output logic z);

enum {s0, s1, s2} present_state, next_state;

always_ff @(posedge clk, negedge n_reset)
    if (!n_reset)
        present_state <= s0;
    else
        present_state <= next_state;

always_comb
    begin
        z = '0;
        next_state = present_state;
        case (present_state)
            s0:
                if (a == b)
                    next_state = s1;
            s1:
                if (¬b)
                    if (a)
                        next_state = s2;
                    else
                        next_state = s1;
                else
                    if (a)
                        next_state = s1;
                    else
                        next_state = s0;
            s2:
                if (a == b)
                    begin
                        z = '1;
                        next_state = s1;
                    end
                else
                    next_state = s0;
        endcase
    end
endmodule
Question 2.

(a) A Binary Coded Decimal (BCD) counter counts from 0 to 9 (decimal) and then returns to 0. How many flip-flops are needed to implement this counter? How many unused values are there? Is it possible to predict what would happen if the counter entered an unused state (for example, as a result of a timing error) if the exact implementation is not known?

[8 marks]

(b) Write a SystemVerilog model of a BCD counter. Comment on whether your design is self-correcting.

[14 marks]

(c) The output from a BCD counter may be displayed on a seven-segment LCD. Show how a SystemVerilog model of a seven-segment decoder can be written. The decoder should display ‘E’ (for Error) if a value greater than 9 is applied to the input. What will be displayed if ‘X’ or ‘Z’ inputs are applied? Why is it necessary to decode ‘X’ and ‘Z’ inputs?

[11 marks]
Indicative Solution for Question 2.

a) 4 flip-flops, 6 unused values. We don’t know exactly what would happen – it depends on how the next state logic is constructed. One possibility is that the counter gets stuck in the unused states.

b)  

```verilog
module BCD (output logic [3:0] q,
            input logic clk, n_reset);

always_ff @(posedge clk, negedge n_reset)
    if (˜n_reset)
        q <= 0;
    else
        if (q < 9)
            q <= q + 1;
        else
            q <= 0;
endmodule
```

This design is self-correcting – if it gets into one of states 1010 to 1111, it goes back to 0000.

c)  

```verilog
module sevenseg(output logic [6:0] data,
                 input logic [3:0] address);

always_comb
    casez (address)
        4'b0000 : data = 7'b1110111;
        4'b0001 : data = 7'b0010010;
        4'b0010 : data = 7'b1011101;
        4'b0011 : data = 7'b1011011;
        4'b0100 : data = 7'b0111010;
        4'b0101 : data = 7'b1101011;
        4'b0110 : data = 7'b1101111;
        4'b0111 : data = 7'b1010010;
        4'b1000 : data = 7'b1111111;
        4'b1001 : data = 7'b1111011;
        4'b101?,
        4'b11?? : data = 7'b1101101;
        default : data = 7'b0000000;
    endcase
endmodule
```

NB The full model is not needed – an outline is sufficient. The display is blanked for ‘X’ and ‘Z’ values. The ‘X’ and ‘Z’ values can only occur in simulation, so the default case wouldn’t be synthesised and is only there to aid debugging.
Question 3.

(a) Explain what is meant by a *vacuous assertion*. How can you know whether an assertion has passed vacuously? [8 marks]

(b) The following code shows the SystemVerilog module header for a synchronous up/down binary counter. If the input ‘up’ is true, the counter increments; when ‘up’ is false, the counter decrements. When the counter reaches the all ones or all zeros states, it returns to the all zeros or all ones states in the next state, respectively.

```verilog
module bincounter #(parameter N)
  (output logic [N-1:0] Count,
   input logic up, clock, n_reset);

Show how a suitable testbench could be written in SystemVerilog (full, syntactically-correct code is not required), including:

(i) an instance of the counter;

(ii) code to generate the clock and reset; and

(iii) code to generate the ‘up’ input.

[15 marks]

(c) Write a SystemVerilog property to verify that the counter increments from the all ones state to the all zeros state. Show how this property may be included in an assertion that generates a warning message if incorrect behaviour is detected. If no warning is generated, how can you detect that the assertion has not passed vacuously? [10 marks]
Indicative Solution for Question 3.

a) A vacuous assertion is one that can never fail. This may be a liveness property (something good happens eventually). The easiest way to detect a vacuous pass is to count how many times each assertion is tested, using the cover construct. If an assertion is never tested, it’s not very useful.

b)

module testbench;
parameter N = 8;
logic [N-1:0] Count;
logic clock, n_reset, up;

bincounter b0 (.*);

initial
begin
reset = '1;
clock = '0;
#1ns reset = '0;
#1ns reset = '1;
forever #5ns clock = ~clock;
end

initial
begin
//some sequence of values for up. These can be timed,
// or synchronised with the clock.
end
endmodule

c)

property Wrap;
@ (posedge clock) (up && Count == {N{1'b1}} |=> (Count == {N{1'b0}}));
endproperty

assert property (Wrap)
else $display("Warning Count did not wrap");

cover property (Wrap);
Question 4.

(a) Explain the sensitive path algorithm. How is it used in testing combinational circuits?

[8 marks]

(b) Derive the logical function of the circuit in Figure 2. Hence suggest which stuck fault is undetectable. Show that your reasoning is correct by using the sensitive path algorithm to try to find a test for that fault.

[8 marks]

(c) Write down the stuck-at-fault list for the circuit shown in Figure 2. Using the sensitive path algorithm derive tests for all potential faults except the undetectable fault.

[11 marks]

(d) What would be the effect of removing the gate where the undetectable fault is situated?

[6 marks]
Indicative Solution for Question 4.

a) The sensitive path algorithm a node is set to a value and that value is propagated to an output by sensitising paths. Only primary inputs are controlled.

b) 

\[ Y = A \cdot B + A \cdot \overline{C} + B \cdot C \]

A.B is a redundant term. Therefore one fault at G is undetectable. Because it's AND-OR logic, we would expect the undetectable fault to be D/0.

To test for D/0 \( \Rightarrow \) D = 1 \( \Rightarrow \) A = B = 1. To propagate E = F = 0. E = 0 \( \Rightarrow \) A = 0 and/or C = 1. F = 0 \( \Rightarrow \) B = 0 and/or C = 0. Contradiction!

b) Fault list: A/0, A/1, B/0, B/1, C/0, C/1, D/0, D/1, E/0, E/1, F/0, F/1, Y/0, Y/1 (Can also include \!C/0, \!C/1.)

To test for D/1, set D = 0, \( \Rightarrow \) A = 0 or B = 0. To propagate to Y, E = F = 0. E = 0 \( \Rightarrow \) A = 0 and/or C = 1. F = 0 \( \Rightarrow \) B = 0 and/or C = 0. Therefore a test is 000/0. Also covers Y/1, D/1, E/1, F/1, A/1.

We can observe that we need to set E, F to 1 in turn to cover E/0, F/0. Therefore the tests are:

100/1 covers Y/0, E/0, C/1, A/0 011/1 covers Y/0, F/0, B/0, C/0

B/1 not covered. 101/0 covers Y/1, E/1, D/1, F/1, C/0, B/1 (Note dual path sensitisation which may cause confusion.)

d) Remove the gate at D. The gate is (presumably) there to protect against a static hazard. If the gate is removed the hazard can still exist, but in a synchronous circuit, this doesn’t matter as any glitch will finish before the next clock edge. And the circuit is simpler. No effect on testing.