2 Verilog [40 points]

Please answer the following four questions about Verilog.

(a) [10 points] Does the following code result in a sequential circuit or a combinational circuit? Please explain your answer.

```
1
2  module sevensegment (input [3:0] data, output reg [6:0] segments);
3      always @ ( * )
4          case (data)
5              4'd0: segments = 7'b111_1110;
6              4'd1: segments = 7'b011_0000;
7              4'd2: segments = 7'b110_1101;
8              4'd3: segments = 7'b111_1001;
9              4'd4: segments = 7'b011_0011;
10         endcase
11 endmodule
```

Sequential circuit.

Explanation:
This code results in a sequential circuit, as all the left-hand side signals are not assigned in every possible condition. For example, for values of `data` that are more than 4, segment is not assigned to a specific value.

(b) [10 points] Does the following code result in an output signal which is zero except for one clock cycle in every three clock cycles (0-0-1-0-0-1...)? If not, please enable this functionality by adding minimal changes. Explain your answer.

```
1  module divideby3 (input clk, input reset, output q);
2      reg [1:0] curVal, nextVal;
3      parameter S0 = 2'b00; parameter S1 = 2'b01; parameter S2 = 2'b10;
4      always @ (*)
5          case (curVal)
6              S0: nextVal = S1;
7              S1: nextVal = S2;
8              S2: nextVal = S0;
9              default: nextVal = S0;
10         endcase
11     assign q = (curVal == S0);
12 endmodule
```

No.

Explanation:
The FSM misses state register. This leads to curVal not changing to nextVal. To fix the problem, the state register should be added:

always @ (posedge clk, posedge reset)
if (reset) curVal <= S0;
else curVal <= nextVal;

(c) [10 points] The following code implements a circuit and we initialize all inputs and registers of the circuit to zero. We apply the following changes to the input signals in two subsequent steps. What are the values of `out` and `tmp` after each step? Please show your work.

- **Step 1:** `sel` changes to 1.
- **Step 2:** While `sel` is still 1, `b` changes to 1.

```
module mod1 (input sel, input a, input b, input c, output out);
    reg tmp = 1'b0;
    always @ (sel)
        if (sel)
            tmp <= ~(a & b);
            out <= tmp ^ c;
        else
            tmp <= 0;
            out <= 0;
endmodule
```

After step 1, tmp and out are respectively 1 and 0 . After step 2, tmp and out are respectively 1 and 0

Explanation:
After step 1: sel is in the sensitivity list of the always block. Therefore, the if statement executes. Since tmp and out are in the left hand side of non-blocking assignments, they execute concurrentlyi, and the second assignment in the if block does not see the new value of tmp when executing.
After step 2: The values of tmp and out do not change because b is not in the sensitivity list of the always block.

(d) [10 points] Is the following code syntactically correct and result in deterministic values for all signals? If not, please explain the mistake(s).

```
module top (input [1:0] in1, in2 , input op, output reg [1:0] z, output reg s);

wire tmp;
always@(*) begin
    tmp = in1[0] & in2[0];
    z[0] = tmp & op;
end
always@(*) begin
    tmp = in1[1] | in2[1];
    z[1] = tmp & (~p)
end
assign s = (z[1] > z[0])
endmodule
```

The code is *not* syntactically correct and does not result in deterministic values.

Explanation:

- 'tmp' has to be declared as a *reg* since it is used in the always blocks.
- 's' should not to be declared as a *reg* since it is driven by the *assign* statement.
- 'tmp' signal is connected to multiple drivers and leads to non-deterministic values.