2 Verilog

Please answer the following four questions about Verilog.

(a) [6 points] Does the following code result in a sequential circuit or a combinational circuit? Explain why.

```
module concat (input clk, input data_in1, input data_in2,
                                 output reg [1:0] data_out);
 always @ (posedge clk, data_in1)
   if (data_in1)
      data_out = {data_in1, data_in2};
   else if (data_in2)
      data_out = {data_in2, data_in1};
endmodule
```

Answer and concise explanation:

Sequential circuit.

Explanation.
This code results in a sequential circuit because `data_in2` is *not* in the sensitivity list, and thus a latch is inferred for `data_out`.

(b) [6 points] In the following code, the input `clk` is a clock signal. What is the hexadecimal value of the output `c` right after the third positive edge of `clk` if initially `c = 8'hE3` and `a = 4'd8` and `b = 4'o2` during the entire time?

```
1 module mod1 (input clk, input [3:0] a, input [3:0] b, output reg [7:0] c);
2 always @ (posedge clk)
3   begin
4     c <= {c, &a, |b};
5     c[0] <= ^c[7:6];
6   end
7 endmodule
```

Please answer below. Show your work.

8'hC4.

Explanation.
Cycle 1: c <= {c, &a, |b} → c <= {1110_0011, 0, 1} → c <= {1000_1101}
c[0] <= ^c[7:6] → c[0] <= ^{11} → c[0] <= 0
At the first positive edge of *clk*, $c = 8'b1000\_1100$
Cycle 2: c <= {c, &a, |b} → c <= {1000_1100, 0, 1} → c <= {0011_0001}
c[0] <= ^c[7:6] → c[0] <= ^{10} → c[0] <= 1
At the second positive edge of *clk*, $c = 8'b0011\_0001$
Cycle 3: c <= {c, &a, |b} → c <= {0011_0001, 0, 1} → c <= {1100_0101}
c[0] <= ^c[7:6] → c[0] <= ^{00} → c[0] <= 0
At the third positive edge of *clk*, $c = 8'b1100\_0100 \rightarrow c = 8'hC4$

Note that since the assignments to *c* are non-blocking, $c[7:6]$ in line 5 is not affected by the assignment to *c* in line 4 in the same cycle.

(c) [6 points] Is the following code syntactically correct? If not, please explain the mistake(s) and how to fix it/them.

```
module 1nn3r ( input [3:0] d, input op, output[1:0] s);
  assign s = op ? (d[1:0] - d[3:2]) :
                        (d[3:2] + d[1:0]);
endmodule

module top ( input wire [6:0] instr, input wire op, output reg z);

  reg[1:0] r1, r2;

  1nn3r i0 (.instr(instr[1:0]), .op(instr[7]), .z(r1) );
  1nn3r i1 (.instr(instr[3:2]), .op(instr[0]), .z(r2) );
  assign z = r1 | r2;

endmodule
```

Answer and concise explanation:

The code is not syntactically correct.

Explanation.

- Module names cannot start with a number → '1nn3r' is not a legal module name.
- The output signal 'z' has to be declared as a 'wire' but not 'reg'.
- 'r1' and 'r2' has to be declared as 'wire's.
- The module '1nn3r' does not have ports named 'instr' and 'z'. Those need to be changed to 'd' and 's', respectively.

(d) [6 points] Does the following code correctly implement a counter that counts from 1 to 11 by increments of 2 (e.g., 1, 3, 5, 7, 9, 11, 1, 3 ...)? If so, say "Correct". If not, correct the code with minimal modification.

```
module odd_counter (clk, count);
  wire clk;
  reg[2:0] count;
  reg[2:0] count_next;

  always@*
  begin
    count_next = count;
    if(count != 11)
      count_next = count_next + 2;
    else
      count_next <= 1;
  end

  always@(posedge clk)
    count <= count_next;
endmodule
```

Answer and concise explanation:

No, the implementation is not correct.

Explanation.
The correct implementation:

```
module odd_counter (input clk, output count);
  wire clk;
  reg[3:0] count;
  reg[3:0] count_next;

  always@*
  begin
    count_next = count;
    if(count != 11)
      count_next = count_next + 2;
    else
      count_next = 1;
  end

  always@(posedge clk)
    count <= count_next;
endmodule
```

(e) [6 points] Does the following code correctly instantiate a 4-bit adder? If so, say "Correct". If not, correct the code with minimal modification.

```
module adder(input a, input b, input c, output sum, output carry);
assign sum = a ^ b ^ c;
assign carry = (a&b) | (b&c) | (c&a);
endmodule


module adder_4bits(input [3:0] a, input [3:0] b, output [3:0] sum, carry);
wire [2:0]s;

adder u0 (a[0],b[0],1'b0,sum[0],s[0]);
adder u1 (a[1],s[0],b[1],sum[1],s[1]);
adder u2 (a[2],s[1],b[2],sum[2],s[2]);
adder u3 (a[3],s[2],b[3],sum[3],carry);
endmodule
```

Yes.

Explanation: Even though the wire s is swapped with the input b, the final computation produced by the module *adder* is still going to be correct since the *or* and *and* operations are commutative.