Problem 1: Logic

Consider the circuit below.

1. What function $F$ of the provided inputs $A$, $B$, and $C$ does the circuit compute?
   \[ F = (A + B)\overline{A}C(B + C) \]

2. Which input, if any, can be tied to a 0 or 1 to produce a two-input XOR gate? How should it be tied?
   Input $C$ can be tied to 0 in order to implement $A \oplus B$.

3. What should inputs $B$ and $C$ be tied to in order to turn this circuit into a NOT gate (i.e. $F = \overline{A}$)?
   Input $B$ should be tied to 1 and input $C$ should be tied to 0.

4. If this circuit were to be fabricated, what is the minimum number of test vectors that would be needed to ensure that there are no stuck-at-0 faults at the outputs of the 5 logic gates? A stuck-at-0 fault is a defect where once the output of a gate goes to 0, it can never transition to a 1 regardless of its inputs.
We essentially need to make each gate go to 0 at some point followed by a 1 at some point. To drive all of the gates to 0, we can set all \( A, B, C \) to 0. Unfortunately, it is not possible to drive all gates to 1 with a single vector as having the output of the final gate be 1 requires the output of its input AND gate to be 0. So, we need one vector \( (A, B = 1, C = 0) \) to drive all of the gates except the final gate to 1 and one vector \( A, B, C = 1 \) to drive the final gate to 1. This translates to a total of 3 test vectors.

**Problem 2: Logic Simplification**

Refer to this truth table containing 4 inputs \( (A, B, C, D) \) and 1 output \( (Q) \).

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1. Write an unsimplified sum-of-products expression for \( Q \) directly from the truth table.
\[
\overline{A}\overline{B}\overline{C}D + \overline{A}B\overline{C}D + \overline{A}BCD
\]

2. Use a Karnaugh map to write the simplified sum-of-products representation of \( Q \).
\[
\overline{A}\overline{C}D + \overline{A}BD
\]

3. Use a Karnaugh map to write the simplified product-of-sums representation of \( Q \).
\[
D(B + \overline{C})\overline{A}
\]

4. Using the product-of-sums representation, draw the circuit that implements this function using only 2-input AND and OR gates with the minimum number of total gates. How many AND and OR gates are there? You may use inputs and their complements (e.g. \( A \) and \( \overline{A} \)) as inputs to your logic gates.

2 AND gates and 1 OR gate are required.

5. Transform this circuit such that it is made up of only inverters and 2-input NAND gates using the minimum number of total gates. How many inverters and NAND gates are there? Once again, you may use inputs and their complements as inputs to your logic gates.
Problem 3: FSMs

In this problem, you will design an FSM for determining whether a number is divisible by 3. The number will be fed in one bit per clock cycle, from most significant to least significant. The output of the FSM should indicate whether the provided bits currently correspond to a number that is divisible by 3.

For example, for input 0, the output should be 1. For the input sequence 1, 1, 0, the output should be 0, 1, 1.

1. Fill in the transitions and associated outputs of the state diagram of this circuit. Populate the transition trigger and output with X if the transition cannot occur. As a hint, the only state you need to store is the remainder of the number when divided by 3.

(1) 0/1
(2) 1/0
(3) 1/1
(4) X/X
(5) 0/0
(6) 0/0
(7) 1/0

2. What type of state machine is this?

It is a Mealy machine since the transitions have associated outputs.

3. Fill in the blanks in the provided code to implement this state machine. You may assume that the output should be updated at the rising clock edge at which the input is processed (in this case, this allows us to make the output value dependent only the current state, which is also updated at the rising clock edge).

```
module div3 ( input wire in, input wire clk, output wire out );
    reg [1:0] state = 0;

    localparam S1 = 0;
```
localparam S2 = 1;
localparam S3 = 2;

assign out = (___[A]___) ? 1'b1 : 1'b0;

always @(posedge clk) begin
  case (state)
    S1:
      begin
        if (in == 0) ___[B]___;
        else ___[C]___;
      end
    S2:
      begin
        if (in == 0) ___[D]___;
        else ___[E]___;
      end
    S3:
      begin
        if (in == 0) ___[F]___;
        else ___[G]___;
      end
  endcase
end
endmodule;

(a) state == S1
(b) state <= S1
(c) state <= S2
(d) state <= S1
(e) state <= S3
(f) state <= S2
(g) state <= S3

4. Suppose we wanted to verify the functionality of our FSM by adding the following assertion to the above div3 module:

assert property (  
  @(posedge clk)
  ($past(in) === 1'dx) [state == ___[YOUR CODE HERE]___]
);  

Complete this assertion by expressing state as an expression of $past(state) and $past(in) given that the value of state is essentially the remainder of the current value when divided by 3.  

(2*$past(state) + $past(in)) % 3
5. Suppose we wanted to modify the state machine to determine whether a number is divisible by 6 (instead of just divisible by 3). What is the minimum number of total states needed to implement this new FSM as a Mealy machine? Do not include the "Start" state.

No new states are needed since we only need to change transition C to output 0. As such, we only need 3 states.

6. Implement the state machine from Part 5 in Verilog and simulate it with the input below.

Input: 1101100010101110100101010101010101010101000101110101

(a) What is the output sequence for this input? Provide your answer as a 64-bit bit string, in the same format as the input. Your output should have 64 digits representing the output after each of the 64 digits of the input are passed to the FSM. As a sanity check, the first 7 digits of your output should be 0010011.

An example output sequence corresponding to an input that is never divisible by 6 is provided below:

0000000000000000000000000000000000000000000000000000000000000000

The desired output is:

0010011100000001000010000010000010000010000010000010000000001000

(b) Attach your Verilog module and testbench. Make sure your module contains an assertion similar to that of Part 4.

The Verilog module and testbench are included below:

```verilog
module div3 ( input wire in, input wire clk, output reg out );

reg [1:0] state = 0;

localparam S1 = 0;
localparam S2 = 1;
localparam S3 = 2;

assert property ( @ (posedge clk) ($past(in) === 1'dx) (state == (2*$past(state) + $past(in)) % 3) );

always @(posedge clk) begin
  out <= 1'b0;
  case (state)
    S1: begin
      if (in == 0) begin
        state <= S1;
        out <= 1'b1;
      end
      else state <= S2;
    end
    S2: begin
```

Note: The rest of the code is not included due to the page limit.
if (in == 0) state <= S3;
else state <= S1;
end
S3:
begin
if (in == 0) state <= S2;
else state <= S3;
end
endcase
end
endmodule;

// Testbench
module test;

wire in;
reg clk;
wire out;
reg [63:0] number;
reg [63:0] outseq;
reg [6:0] ctr;

always #10 clk = ~clk;

assign in = number[63];
// Instantiate design under test
div3 DUT(.in(in), .clk(clk), .out(out));

initial begin
    clk = 0;
    number =
        64'b1101100101011011010101010101010101010101010101010101000101110101;
    outseq = 64'b0;
    ctr = 0;

    repeat(64) @(negedge clk) begin
        number = number << 1;
        outseq = outseq << 1 || out;
        ctr = ctr + 1;
    end

    $display("%b", outseq);
    $finish;
end
endmodule