Problem 1: Given \( F = AB + CD' \)

a. Implement \( F \) using as few 2 input NAND gates as possible. Assume that only the true literals (A, B, C, D) are available, not their complements (A', B', C', D').

b. Write \( F' \) in product of sums notation

Solution 1:

a) First draw the 2-level logic implementation directly from the Boolean equation. Since we only have the true literals available, we can create inverters by connecting the inputs on a NAND gate. Next, change the AND gates to NAND, and add the inverter bubbles at the input of the OR gate to maintain the same functionality. Finally, use DeMorgan’s law to convert the OR gate with inverted inputs to a NAND gate (\( X + Y = XY \)).

b) Use DeMorgan’s law to find the PoS notation:

\[
\overline{F} = AB + CD = AB \cdot CD
\]

\[
\overline{F} = (A + B) \cdot (C + D)
\]
Problem 2: Given $G = (A+B)(C'+D)$

c. Implement $G$ using as few 2 input NOR gates as possible. Assume that only the true literals are available, not their complements.
d. Write $G'$ in sum of products notation.

Solution 2:

a) Follow the same steps as in problem 1 above: draw the two level logic implementation, switch the first level of logic to NOR gates and add inverter bubbles to the input of the AND gate, then apply DeMorgan’s law ($\overline{XY} = \overline{X} + \overline{Y}$).

b) Use DeMorgan’s law to find the SoP notation of $G'$:

$$
\overline{G} = (A + B)(C + D) = (A + B) + (C + D) = \overline{AB} + \overline{CD}
$$
Problem 3: Answer the following questions for the FSM below:
   a. Is this a Mealy or a Moore machine?
   b. Briefly describe the function of this sequence detector. When is the output 1?
   c. Write a Verilog module which would implement this FSM for input variable "In" and output variable "Out." Use the same standard format as was presented in the Lab 3 lecture and used in Lab 3. (Define your states; use one always block for next state and output; use one always block for state transition)

Solution 3:

a) Since the output depends ONLY on the current state, this is a Moore machine. This is evident from the bubbles – the value in the lower half of the bubble is the output for that state, and the arcs are only labeled with inputs. (If this was a Mealy machine, there would be no outputs in the state bubbles, and instead the outputs would be on the arcs with the inputs.)

b) The output is 1 when we detect the first ‘1’ in a sequence of inputs. The output is high when the first one is detected, then immediately set low again when the next output comes in. In other words, this FSM detects the rising edge of the input. A 0 on the input sets the FSM back to the original state (Sue), and we wait for the first ‘1’ again…

c) Verilog implementation: (next page; code in red marks a common mistake and Verilog pitfall – if you don’t know why each piece of code is marked in red, go study FSM.pdf! 😊)
module FSM (input wire Reset, input wire In, output wire Out);

localparam Sue = 2'b00, Ken = 2'b01, Bob = 2'b10;

reg [1:0] CurrentState;
reg [1:0] NextState;

always @ (posedge Clock) begin
    if (Reset) begin
        CurrentState <= Sue;
    end
    else begin
        CurrentState <= NextState;
    end
end

assign Out = (CurrentState == Ken);

always @ (*) begin
    NextState = CurrentState;
    case (CurrentState)
        Sue : begin
            NextState = (In) ? Ken : Sue;
        end
        Ken : begin
            NextState = (In) ? Bob : Sue;
        end
        Bob : begin
            NextState = (In) ? Bob : Sue;
        end
        default : begin
            NextState = Sue; /* Ensure self-starting */
        end
    endcase
endmodule

/* This looks to be a positive-edge detector for a synchronous signal */
Problem 4: For the following questions, assume that only the true variables (A,B,C) are available, and not their complements. Try to use the fewest gates and the fewest inputs possible.

a. Implement the function \( F(A,B,C) = \sum m(0,1,4,7) \) using a 4:1 mux and at most 1 inverter.

b. Use a 3:8 decoder and OR and AND gates to implement \( F(A,B,C) = \sum m(0,4,7) \)

c. Use a 3:8 decoder and OR and AND gates to implement \( G(A,B,C) = \Pi M(1,6) \)

Solution 4:

a) Draw the truth table, shown below. The 4:1 mux has two select bits, 4 inputs, and 1 output. We use the mux select bits A and B to choose which section of the truth table we want to look at, and we see that f really only takes 4 values: 1, 0, C’ and C. These become the 4 inputs to the mux (where the inverter is used to create C’). The truth table and the circuit are shown below.

<table>
<thead>
<tr>
<th>ABC</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>0</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>

4-a: Transition table

b) A 3:8 decoder takes three inputs ABC, and asserts its 8 output lines based on the values of ABC. Basically, the decoder selects the minterms of F, which can be fed into an OR gate to get the output.
c) Given that $G(A,B,C) = \Pi M(1,6)$, we know that $G(A,B,C) = \Sigma m(0,2,3,4,5,7)$. Connect each of these minterms from the decoder to a 6-input OR gate to get $G$.

Note: if you were not restricted to using OR and AND gates only for this problem, you could significantly reduce your input count by implementing $G = M_1 \cdot M_6 \rightarrow \overline{G} = \overline{M_1} + \overline{M_6}$. Using DeMorgan’s law on the complemented maxterms gives you minterms, so you only need a 2-input NOR gate to implement $G$. 
Problem 5: Design a counter with one control input. When the input is high, the counter should sequence through three states: 10, 01, 11 and repeat. When the input is low the counter should sequence through the same states in the opposite order 11, 01, 10 and repeat.

a. Draw the state diagram and state transition table
b. Implement the counter using D flip flops and whatever gates you like.
c. Is your counter self-starting with the input either high or low?

Solution 5:
a) The state diagram and state transition table are shown below. Since the state 00 is not used in this counter, its next state values are Don’t Cares.

b) We’ll need at least two flip-flops for this, since we have two state bits. First use K-maps to find the next-state logic for Q1+ and Q0+:

\[ Q1^+ = \overline{In} Q0 + In Q0 + Q1 \]
\[ Q0^+ = Q0 + \overline{In} Q1 + In Q1 \]

We’ll implement this using 2 D flip-flops as shown in the circuit below:
c) To find out if the counter is self-starting, we need to know how the “don’t cares” were implemented in our logic. Remember that Don’t Cares WILL take on a real logic value, which will be determined by your implementation. In this case, we can see from the K-maps that all the X’s are included in the boxed implicants. That means we are treating them like 1’s. Go back to the state transition table and replace the XX states with the actual values from our implementation:

![State Transition Diagram]

<table>
<thead>
<tr>
<th>In</th>
<th>Q1 Q0</th>
<th>Q1+ Q0+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>XX --&gt; 11</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

State Transition Table

We see that our counter is self-starting, because the unused 00 state will still lead us to our counter sequence, regardless of the value of the input.
Problem 6: A finite state machine has one input and one output. The output becomes 1 and remains 1 thereafter when at least two 0s and at least two 1s have occurred as inputs, in any order after reset. Draw a state diagram of this FSM as a Moore machine. Try to minimize the number of states.

Solution 6:
Although we have not yet learned the method to minimize state machines, we can still make wise choices about our state definition to prevent excess states. For example, note that the inputs can occur “in any order”, so we don’t need to be concerned about the particular sequence of the inputs. Since this problem asks for a Moore machine, we need to define states that have an output associated with that state.

We’ll define our states by using three numbers ABC, where:
A = the number of 0’s that have occurred since reset
B = the number of 1’s that have occurred since reset
C = output
**Problem 7:** A Moore machine has one input and one output. The output should be 1 if the total number of 0s at the input is odd, and the total number of 1s at the input is an even number greater than 0. Draw a state diagram. Try to minimize the number of states.

**Solution 7:**
Again, without going through the formal state minimization process, we can still try to avoid redundant states. It is useful to think of states as unique situations or conditions, rather than a sequence of inputs. We’ll define our states as follows:

- A = ODD or EVEN number of zeros received since reset
- B = 0 if there has NOT been a one at the input, 1 if there HAS
- C = output

![State Diagram]

The state diagram is shown above with 4 states. We’ll have two states where B = 0, meaning we have not yet seen a 1 on the input. Each time we receive a 0, we’ll toggle between ODD and EVEN, with no change to B. Once we receive a 1 on the input, we go to a parallel set of states where B = 1 and again toggle between ODD and EVEN when we receive zeros. Since we don’t care *how many* 1’s we have received, a 1 on the input does not change the state after the initial jump to the B = 1 states. Also note that you cannot get from a B = 1 state back to a B = 0 state unless a Reset occurs.
Problem 8: Design a 3 FlipFlop counter which transitions through states \( Q_2Q_1Q_0 = 000, 100, 110, 111, 011, 001 \) and then repeats.
   a. Draw the state diagram and state transition table
   b. Draw the Karnaugh maps, clearly indicating the implicants that you use in your covers of the next-state functions.
   c. Implement the counter using D flip flops and whatever gates you like.
   d. Is your counter self starting? If yes, show the transitions of the unused states. If no, change it to make it self starting, and show the transitions of the unused states.

Solution 8:

a) 010 and 101 are not included in the defined sequence, so their next states are Don’t Cares. They are currently not connected to the rest of the state diagram. The state diagram and state transition table are:

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Q_2Q_1Q_0 )</td>
<td>( Q_2^+ Q_1^+ Q_0^+ )</td>
</tr>
<tr>
<td>000</td>
<td>100</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
</tr>
<tr>
<td>010</td>
<td>XXX</td>
</tr>
<tr>
<td>011</td>
<td>001</td>
</tr>
<tr>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>101</td>
<td>XXX</td>
</tr>
<tr>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>111</td>
<td>011</td>
</tr>
</tbody>
</table>

b) The K-maps come from the transition table:
From the K-maps, we obtain the next-state functions:

\[ Q_2^+ = \overline{Q_0} \]
\[ Q_1^+ = Q_2 \]
\[ Q_0^+ = Q_1 \]

**c)** Implementing the counter using D flip-flops is very simple since our next-state functions don’t contain any combinational logic.
d) To find out if the counter is self-starting, we go back to find out how the “don’t cares” were implemented. Find Q2 Q1 Q0 = 010 in the K-map and see if the X’s in those squares are included in implicants. We find that the next state of 010 has gone from XXX → 101, and the next state of 101 has gone from XXX → 010. The new transition table looks like this:

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2 Q1 Q0</td>
<td>Q2+ Q1+ Q0+</td>
</tr>
<tr>
<td>000</td>
<td>100</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
</tr>
<tr>
<td>010</td>
<td>XXX --&gt; 101</td>
</tr>
<tr>
<td>011</td>
<td>001</td>
</tr>
<tr>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>101</td>
<td>XXX --&gt; 010</td>
</tr>
<tr>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>111</td>
<td>011</td>
</tr>
</tbody>
</table>

8-a: Transition table

With this implementation, states 010 and 101 toggle back and forth and are not connected to the rest of the counter states, so this is NOT a self-starting counter.

To make this counter self-starting, we need to change one of the don’t care values to so that we alter the next state logic and prevent state 010 and 101 from transitioning to each other. If we change the Next State of 101 from 010 to 011, then both of the unused states (010 and 101) will eventually lead back to the counter states. To make this change, notice that we need to change the Q0+ bit of the next state for 101. Make the change in the K-map and determine the new next-state logic:
This change is reflected in the transition table and the transition diagram as well:

\[ Q_0^+ = Q_1 + Q_2Q_0 \]

Now we can clearly see that any state will eventually lead to the counter sequence, so our counter is self-starting. The new implementation looks like this: