1. Create a J-K flip-flop from a single D-type flip flop and any boolean logic gates needed. Approach this as a state machine design: draw a state diagram, write down the truth table, and implement the combinatoric logic using gates.

2. Design a complex 3-bit counter. If input $M = 0$, three bits $Q_2Q_1Q_0$ count in a normal binary sequence: 000 001 010 011 100 101 110 111. If input $M = 1$, these bits count in a Grey code sequence: 000 001 011 010 110 111 101 100. The input $M$ can be changed in mid-sequence, in which case the count should simply continue to the next state in the selected representation. Draw a full state diagram for this counter specifying all transitions, and solve the combinatoric logic to set the next state bits as a function of $(M, Q_2, Q_1, Q_0)$.

3. Demonstrate a solution for the previous problem using a read-only memory and a 3-bit register. Assume you have a ROM with 6 address lines (64 memory locations) and 8 output data lines (8 bits per address). Draw a block diagram showing how you would wire up the ROM and register (be specific, showing the connection for each pin) and show the contents of each ROM memory address (in table form) that implements the counter logic.

4. Design a synchronous bi-polar stepper motor controller with a full-step drive sequence. The controller needs four output lines ($ABC\!D$) to drive four leads on the stepper motor. To drive the motor forward, the count sequence should be as shown in the table below, repeating such that each lead is HI for 2 ticks and LO for 2. To drive the motor backwards, this count sequence simply needs to be reversed. Your controller should have one input bit to indicate the direction for the motor to turn, and also an input clock to control the motor speed. You may solve this problem in any manner you wish, but you need to justify your solution. A state machine is the obvious solution, but not the only possibility.

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

5. The verilog module below (next page) shows a block that implements a finite state machine. Draw a schematic showing the input and outputs of this module. Draw a state diagram that represents the logic of this state machine and try to describe in words what the state machine does. Note this FSM uses “one hot” encoding (one bit at a time HI) to define the states. This is often a more efficient way to define states on an FPGA than normal binary encoding.
module myStateMachine (input clock, input d, output q);

reg [3:0] theState;

initial
  theState <= 4'h1;

always @(posedge clock)
begin
  if (d && theState == 4'h1)
    theState <= 4'h2;
  else if (d && theState == 4'h2)
    theState <= 4'h4;
  else if (d && theState == 4'h4)
    theState <= 4'h8;
  else if (d && theState == 4'h8)
    theState <= 4'h2;
  else
    theState <= 4'h1;
end

assign q = (theState == 4'h8);
endmodule