1. Write a VHDL module that defines a sequential circuit implementing each of the state
diagrams in problem 1 in problem set 7. Use the state diagrams that appear in the posted
solutions. Write these VHDL modules in the style shown on page 4-30 of the lecture notes.
That is, use explicit state variables.

library IEEE;
use IEEE.std_logic_1164.all;

entity partA is
  port (
    A, B: in STD_LOGIC;
    X, Y: out STD_LOGIC
  );
end partA;

architecture explicit of partA is
  signal s: STD_LOGIC;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if s = '0' and A = '0' and B = '0' then
        s <= '1';
      elsif s = '1' and A = '1' then
        s <= '0';
      end if;
    end if;
  end process;

  X <= '1' when (s= '0' and ((A = '0' and B = '0')
    or (A = '0' and B = '1')
    or (A = '1' and B = '1'))
  or (s= '1' and ((A = '0' and B = '0'))) else
    '0';

  Y <= '1' when (s= '0' and ((A = '0' and B = '0')
    or (A = '1' and B = '0')
    or (A = '1' and B = '1'))
  or (s= '1' and ((A = '1' and B = '0')
    or (A = '1' and B = '1'))) else
    '0';

  -- alternate form
  -- X <= ((not s) and B) or ((not A) and (not B));
  -- Y <= A or ((not s) and (not B));
end explicit;

-----------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity partB is
  port (    
    A, B: in STD_LOGIC;
    X, Y: out STD_LOGIC
  );
end partB;

architecture explicit of partB is
signal s0, s1: STD_LOGIC;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if s1 = '0' and s0 = '0' and A = '0' and B = '0' then
        s1 <= '0'; s0 <= '1';
      elsif s1 = '0' and s0 = '0' and A = '1' and B = '0' then
        s1 <= '1'; s0 <= '0';
      elsif s1 = '0' and s0 = '0' and A = '0' and B = '1' then
        s1 <= '1'; s0 <= '0';
      elsif s1 = '0' and s0 = '1' and A = '0' and B = '1' then
        s1 <= '1'; s0 <= '0';
      elsif s1 = '0' and s0 = '1' and A = '1' and B = '1' then
        s1 <= '1'; s0 <= '0';
      elsif s1 = '1' and s0 = '1' and A = '0' and B = '1' then
        s1 <= '0'; s0 <= '1';
      elsif s1 = '1' and s0 = '1' and A = '1' and B = '0' then
        s1 <= '0'; s0 <= '1';
      elsif s1 = '1' and s0 = '1' and A = '1' and B = '1' then
        s1 <= '0'; s0 <= '1';
      end if;
    end if;
  end process;
  X <= s1;
  Y <= s0;
end explicit;

--------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity partC is
  port (    
    A, B: in STD_LOGIC;
    X: out STD_LOGIC
  );
end partC;

architecture explicit of partC is
signal s0, s1: STD_LOGIC;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if s1 = '0' and s0 = '0' and A = '0' and B = '0' then
        s1 <= '0'; s0 <= '1';
      elsif s1 = '0' and s0 = '0' and A = '1' and B = '1' then
        s1 <= '1'; s0 <= '0';
      elsif s1 = '0' and s0 = '1' and A = '0' and B = '1' then
        s1 <= '1'; s0 <= '0';
      elsif s1 = '1' and s0 = '1' and A = '0' and B = '1' then
        s1 <= '0'; s0 <= '1';
      elsif s1 = '1' and s0 = '1' and A = '1' and B = '0' then
        s1 <= '0'; s0 <= '1';
      elsif s1 = '1' and s0 = '1' and A = '1' and B = '1' then
        s1 <= '0'; s0 <= '1';
      end if;
    end if;
  end process;
  X <= s1;
  Y <= s0;
end explicit;
elsif s1 = '0' and s0 = '1' and A = '0' and B = '1' then
  s1 <= '0'; s0 <= '0';
elsif s1 = '0' and s0 = '1' and A = '1' and B = '0' then
  s1 <= '0'; s0 <= '0';
elsif s1 = '0' and s0 = '1' and A = '1' and B = '1' then
  s1 <= '1'; s0 <= '0';
elsif s1 = '1' and s0 = '0' and A = '0' and B = '0' then
  s1 <= '1'; s0 <= '1';
elsif s1 = '1' and s0 = '0' and A = '1' and B = '0' then
  s1 <= '1'; s0 <= '1';
elsif s1 = '1' and s0 = '0' and A = '0' and B = '1' then
  s1 <= '0'; s0 <= '0';
elsif s1 = '1' and s0 = '0' and A = '1' and B = '1' then
  s1 <= '0'; s0 <= '0';
elsif s1 = '1' and s0 = '1' and A = '0' and B = '0' then
  s1 <= '0'; s0 <= '0';
elsif s1 = '1' and s0 = '1' and A = '0' and B = '1' then
  s1 <= '0'; s0 <= '0';
elsif s1 = '1' and s0 = '1' and A = '1' and B = '0' then
  s1 <= '0'; s0 <= '0';
elsif s1 = '1' and s0 = '1' and A = '1' and B = '1' then
  s1 <= '0'; s0 <= '0';
end if;
end if;
end process;
X <= '1' when (s1 = '0' and s0 = '0')
  or (s1 = '0' and s0 = '1' and A = '0' and B='0')
  or (s1 = '0' and s0 = '1' and A = '1' and B='0')
  or (s1 = '1' and s0 = '0') else
  '0';
end explicit;

library IEEE;
use IEEE.std_logic_1164.all;

entity partD is
  port (A, B: in STD_LOGIC;
        X: out STD_LOGIC);
end partD;

architecture explicit of partD is
  signal s2, s0, s1: STD_LOGIC;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if s2 = '0' and s1 = '0' and s0 = '0' and A = '0' then
        s2 <= '0'; s1 <= '0'; s0 <= '1';
      elsif s2 = '0' and s1 = '0' and s0 = '0' and A = '1' then
        s2 <= '1'; s1 <= '0'; s0 <= '1';
      elsif s2 = '0' and s1 = '0' and s0 = '1' and A = '1' then
        s2 <= '0'; s1 <= '1'; s0 <= '0';
      elsif s2 = '0' and s1 = '0' and s0 = '1' and A = '1' then
        s2 <= '1'; s1 <= '0'; s0 <= '1';
      elsif s2 = '0' and s1 = '1' and s0 = '0' and A = '1' then
        s2 <= '1'; s1 <= '1'; s0 <= '0';
      elsif s2 = '0' and s1 = '1' and s0 = '0' and A = '1' then
        s2 <= '0'; s1 <= '0'; s0 <= '1';
elsif \( s_2 = '1' \) and \( s_1 = '0' \) and \( s_0 = '1' \) and \( A = '1' \) then
\[
\begin{align*}
  s_2 &\leq '0'; \\
  s_1 &\leq '1'; \\
  s_0 &\leq '0'; \\
\end{align*}
\]
elsif \( s_2 = '1' \) and \( s_1 = '1' \) and \( s_0 = '0' \) and \( A = '1' \) then
\[
\begin{align*}
  s_2 &\leq '0'; \\
  s_1 &\leq '0'; \\
  s_0 &\leq '1'; \\
\end{align*}
\]
elsif \( s_2 = '1' \) and \( s_1 = '1' \) and \( s_0 = '0' \) and \( A = '1' \) then
\[
\begin{align*}
  s_2 &\leq '0'; \\
  s_1 &\leq '0'; \\
  s_0 &\leq '0'; \\
\end{align*}
\]
end if;
end if;
end process;
X <= \( s_2 \);
Y <= \( s_0 \);
end explicit;

2. Write a VHDL module that defines a sequential circuit implementing each of the state diagrams in problem 2, of problem set 7. Write these VHDL modules in the style shown on page 4-31 of the lecture notes. That is, use symbolic state variables, with state names red, green, blue and yellow in place of \( s_1 s_0 = 00, 01, 10 \) or 11.

library IEEE;
use IEEE.std_logic_1164.all;

entity partA is
  port (
    A, B: in STD_LOGIC;
    X, Y: out STD_LOGIC
  );
end partA;

architecture explicit of partA is
  type state_type is (red, green);
  signal state: state_type;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if state = red and B = '1' then
        state <= green;
      elsif state = green and A = B then
        state <= red;
      end if;
    end if;
  end process;
  X <= '1' when (state = red and (A = '0' and B = '0')
               or (A = '0' and B = '1')
               or (A = '1' and B = '1'))
               or (state = green and B = '0')
               else '0';
  Y <= '1' when (state = red and A = '1')
               or (state = green and (A = '0' and B = '0')
               or (A = '0' and B = '1')
               or (A = '1' and B = '1'))
               else '0';
end explicit;

---------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity partB is
  port (  
    A, B: in STD_LOGIC;  
    X, Y: out STD_LOGIC  
  );
end partB;

architecture explicit of partB is
  type state_type is (red, blue, yellow);
  signal state: state_type;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if state = red and B = '0' then
        state <= blue;
      elsif state = red and A = '1' and B = '1' then
        state <= yellow;
      elsif state = blue and A = '0' then
        state <= yellow;
      elsif state = yellow and A = '0' and B = '0' then
        state <= red;
      elsif state = yellow and A = '1' and B = '1' then
        state <= blue;
    end if;
  end if;
end process;
  X <= '1' when state = red else '0';
  Y <= '1' when state = yellow else '0';
end explicit;

-----------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity partC is
  port (  
    A, B: in STD_LOGIC;  
    X, Y: out STD_LOGIC  
  );
end partC;

architecture explicit of partC is
  type state_type is (red, blue, green, yellow);
  signal state: state_type;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if state = red and A = '0' and B = '1' then
        state <= green;
      elsif state = red and A = '1' then
        state <= blue;
      elsif state = red and A = '0' and B = '0' then
        state <= yellow;
      elsif state = green and A = '0' and B = '1' then
        state <= red;
      elsif state = green and A = '1' then
        state <= blue;
      end if;
    end if;
  end process;
end explicit;
state <= yellow;

elsif state = green and B = '0' then
state <= yellow;
elsif state = green and B = '1' then
state <= red;
elsif state = yellow and A = '0' and B = '0' then
state <= red;
elsif state = yellow and A = '0' and B = '1' then
state <= blue;
end if;
end process;
X <= '1' when (state = red and A = '0' and B = '1')
or state = green
or (state = blue and A = '0' and B = '1')
or (state = yellow and A = '0') else
'0';
end explicit;
3. In each of the block diagrams shown below, the blocks are Moore-model sequential circuits. The numeric intervals adjacent to the signal names specify the timing characteristics of each circuit. So for example, the interval [-5,+.5] at input A of the right hand block in the first diagram means that input A must be stable starting 5 ns before a rising clock transition and continuing until .5 ns after the rising clock transition. Similarly, the interval [+1.5,+8] at output X of the left hand block means that this output may be changing at any time during the interval from 1.5 ns after a rising clock transition to 8 ns after a rising clock transition.

For each diagram, determine if there are situations that can cause timing violations no matter how large the clock period is (this may happen if an output of one block can start changing shortly after the clock changes, but the corresponding input on the other block requires that the output remains stable following the clock transition). If the circuit can have such a timing violation, explain how it can be eliminated. Then, for each circuit, determine the shortest clock period for which we can guarantee that the circuit operates correctly (after any needed modifications are made). What is the corresponding clock frequency? Assume that there is no clock skew between the different circuit components and that a single inverter has a minimum delay of .5 ns and a maximum delay of 2 ns.

In this circuit, output C of the right hand block can change .5 ns after the clock transition, while the input Z of the left hand block requires that it be stable until 1 ns after the clock changes. This problem can be corrected by adding a pair of inverters on the interconnecting signal to add some delay. Once this change is made, the timing specifications on signal X→A require a minimum period of 13 ns, the timing specifications on signal Y→B also require a minimum period of 13 ns and the timing specifications on signal C→Z require a minimum period of 8 ns (accounting for the maximum of 4 ns of delay that was added). So the minimum clock period is 13 ns, which corresponds to a frequency of 77 MHz.

In this circuit, there are no timing violations that occur for all clock periods, so we don’t need to modify the circuit. Signal W→A requires a minimum period of 15 ns, X→B requires a minimum period of 13 ns, C→Y requires a minimum period of 12 ns and D→Z requires a minimum period of 15 ns. So, the minimum clock period is 15 ns, which corresponds to a frequency of 67 MHz.
In this circuit, there are no timing violations that occur for all clock periods, so we don’t need to modify the circuit. Signal $W \rightarrow A$ requires a minimum period of 13 ns, $B \rightarrow X$ requires a minimum period of 13 ns, $C \rightarrow G$ requires a minimum period of 13 ns and $F \rightarrow Y$ requires a minimum period of 14 ns. So, the minimum clock period is 14 ns, which corresponds to a frequency of 71 MHz.

4. Repeat problem 3, but this time assume that there can be as much as 1 ns of clock skew between different blocks. That is, assume that the clock can arrive up to 1 ns earlier (or later) at one block relative to another.

In the first circuit, the $C \rightarrow Z$ signal is still the only one that causes a problem at all clock periods. Because of the clock skew, we need additional delay on this signal. With two inverter pairs on this signal, we get a minimum of 2 ns of delay which enough to eliminate the timing violation. This adds up to 4 ns of delay to this signal which means that it requires a clock period of at least 11 ns if there is no clock skew and 12 ns if there is 1 ns of clock skew. The other two signals both require a clock period of at least 14 ns with the clock skew included, so the minimum clock period is 14 ns and the corresponding frequency is 71 MHz.

In the second circuit, signal $C \rightarrow Y$ can change 1 ns after the clock changes and is required to be stable until .4 ns after the clock changes. If the clock arrives at the left-hand block 1 ns later than at the right-hand block, we can get a timing violation. This can be corrected by adding an inverter pair to this signal. This is the only modification we need to make. With the inverter pair on $C \rightarrow Y$, the minimum clock period required for this signal is 17 ns, assuming 1 ns of clock skew. This is the worst-case requirement, so the minimum clock period is 17 ns. The corresponding clock frequency is 59 MHz.

In the third circuit, the signal $C \rightarrow G$ can change 1 ns after the clock changes and is required to be stable until .2 ns after the clock changes. If the clock arrives at the bottom block 1 ns later than at the right-hand block, we can get a timing violation. This can be corrected by adding an inverter pair to this signal. This is the only modification we need to make. With the inverter pair on $C \rightarrow G$, the minimum clock period required for this signal is 18 ns, assuming 1 ns of clock skew. This is the worst-case requirement, so the minimum clock period is 18 ns. The corresponding clock frequency is 56 MHz.
5. In this problem, you are to design a serial multiply-by-5 circuit. Your circuit will have a single data input \( D \), a synchronous reset input \( R \) and a single output \( Q \). Whenever \( R \) is high, your circuit should output 0, but when \( R \) drops low, your circuit should treat input \( D \) as a binary value, received with the least significant bit first. Your circuit should output a value which is equal to the input value, multiplied by 5. So if you receive the values 0001101 (where the rightmost bit is received on the first clock tick following reset, then the second bit from the right, etc.) your circuit will output 1000001. Note that for any value \( x \), \( 5x = 4x + x \), and \( 4x \) is the same as \( x \) with two bits additional bits at the right. So you can perform the multiplication by adding \( x \) to a delayed version of itself.

Create a block diagram for a circuit that implements the serial multiply-by-5 function, assuming that you have two sub-circuits, one of which implements a delay of two clock ticks and another which implements a serial adder (such a circuit takes two binary values, starting with their least significant bit and outputs the sum).

![Delay and Adder Diagram](image)

Make logic diagrams of each of the two sub-circuits. For the serial adder, also show the state transition diagram for the circuit, the state table, the next-state equations and the output equations.

The state diagram for the serial adder is shown below. This is for a Mealy model circuit. The actual implementation adds a flip flop on the output, converting it to a Moore model circuit and delaying the output sum bit by one clock tick relative to the input bits.

![State Transition Diagram](image)

<table>
<thead>
<tr>
<th>( S_0 )</th>
<th>( A )</th>
<th>( B )</th>
<th>( D_0 )</th>
<th>( S_0 )</th>
<th>( \text{Sum} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00/0</td>
<td>00/0</td>
<td>00/0</td>
<td>00/0</td>
<td>00/0</td>
<td>00/0</td>
</tr>
<tr>
<td>01/1</td>
<td>01/1</td>
<td>01/1</td>
<td>01/1</td>
<td>01/1</td>
<td>01/1</td>
</tr>
<tr>
<td>10/0</td>
<td>10/0</td>
<td>10/0</td>
<td>10/0</td>
<td>10/0</td>
<td>10/0</td>
</tr>
<tr>
<td>11/1</td>
<td>11/1</td>
<td>11/1</td>
<td>11/1</td>
<td>11/1</td>
<td>11/1</td>
</tr>
</tbody>
</table>

\[ D_0 = AB + AS_0 + BS_0 \]
\[ \text{Sum} = A \oplus B \oplus S_0 \]

Schematics for both circuits appear below.
Show how you could use the same idea to produce a serial multiply-by-13 circuit. Show the block diagram of the resulting circuit.

Since 13 is 1101 in binary, a multiply by 13 will require two adders and two delays. The block diagram that illustrates how this is done is shown below. The reason for using a second two stage delay is to compensate for the one clock tick of delay that occurs in the first adder.
Show how to generalize this idea to produce a general 4 bit serial multiplier. This circuit has a reset input and two serial data inputs, $A$ and $B$ and has a single output, which is equal to the product of $A$ and $B$. Show a block diagram for such a circuit, using serial adders as building blocks.

The block diagram appears below. When the reset signal drops, the Mreg block at the bottom stores the bits of the multiplier, starting with the least significant bits. These bits are used to control the addition of partial products in the adders at right. The delay blocks act to delay the input value by appropriate amounts. The initial delay is needed because the Mreg block also has a delay of one clock tick. Two stage delays are used in most of the delay blocks to compensate for the one tick delay in the serial adder blocks.
6. A 4 bit twisted ring counter is a sequential circuit which produces the following sequence of output values: 0000, 1000, 1100, 1110, 0111, 0011, 0001 and then repeats. Design a circuit for a 4 bit twisted ring counter that uses four D flip flops. Draw a state transition diagram, a state table and a schematic for your circuit. Design an alternate implementation using just three flip flops and draw a state transition diagram, state table and a schematic for your circuit. If your designs are extended to implement an \( n \) bit twisted ring counter, how many flip flops are required using each of the two approaches. In what situations would you prefer the first method? In what situations would you prefer the second?
If the designs are extended to \( n \) bits, the first will need \( n \) flip flops, while the second will need \( 1 + \log_2 n \) (assuming \( n \) is a power of 2). So for example, if \( n=64 \), this would be 64 vs. 7. On the other hand, the second will need more gates to generate the output logic, which could more than compensate for the savings in the number of flip flops. The first is preferable if speed is critical, but the second might be better for large values of \( n \) if it is important to minimize the number of flip flops.