1. Use VHDL to design the 4 bit arithmetic unit specified in problem 4 of homework 5 (you may borrow from the posted solution, if you wish). Use a dataflow description to define basic building blocks for each component (e.g. half-adders and full-adders), then use structural VHDL to assemble these into larger components, and to assemble the components into the complete circuit. Simulate the circuit on the same inputs as in homework 5.

library IEEE;
use IEEE.std_logic_1164.all;

entity flip is
  port (
    en: in STD_LOGIC;
    a: in STD_LOGIC_VECTOR (3 downto 0);
    x: out STD_LOGIC_VECTOR (3 downto 0)
  );
end flip;

architecture flip_arch of flip is
begin
  x <= (not a) when en = '1' else a;
end flip_arch;

library IEEE;
use IEEE.std_logic_1164.all;

entity inc is
  port (
    en: in STD_LOGIC;
    a: in STD_LOGIC_VECTOR (3 downto 0);
    x: out STD_LOGIC_VECTOR (3 downto 0)
  );
end inc;

architecture inc_arch of inc is
begin
  signal c: STD_LOGIC_VECTOR (2 downto 0);
  c(0) <= a(0) and en;
  c(1) <= a(1) and c(0);
  c(2) <= a(2) and c(1);
  x(0) <= a(0) xor en;
  x(1) <= a(1) xor c(0);
  x(2) <= a(2) xor c(1);
end inc_arch;
library IEEE;
use IEEE.std_logic_1164.all;

entity pass is
  port (
      en: in STD_LOGIC;
      a: in STD_LOGIC_VECTOR (3 downto 0);
      x: out STD_LOGIC_VECTOR (3 downto 0)
  );
end pass;

architecture pass_arch of pass is
begin
  x <= a when en = '1' else "0000";
end pass_arch;

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

library IEEE;
use IEEE.std_logic_1164.all;

entity adder is
  port (
      cin: in STD_LOGIC;
      a, b: in STD_LOGIC_VECTOR (3 downto 0);
      x: out STD_LOGIC_VECTOR (3 downto 0);
      cout: out STD_LOGIC
  );
end adder;

architecture adder_arch of adder is
begin
  c(0) <= (a(0) and b(0)) or (a(0) and cin) or (b(0) and cin);
  c(1) <= (a(1) and b(1)) or (a(1) and c(0)) or (b(1) and c(0));
  c(2) <= (a(2) and b(2)) or (a(2) and c(1)) or (b(2) and c(1));
  c(3) <= (a(3) and b(3)) or (a(3) and c(2)) or (b(3) and c(2));
  cout <= c(3);
  x(0) <= a(0) xor (b(0) xor cin);
  x(1) <= a(1) xor (b(1) xor c(0));
  x(2) <= a(2) xor (b(2) xor c(1));
  x(3) <= a(3) xor (b(3) xor c(2));
end adder_arch;

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

library IEEE;
use IEEE.std_logic_1164.all;

entity control is
  port (
      c: in STD_LOGIC_VECTOR (2 downto 0);
  );
architecture control_arch of control is
begin
  na <= '1' when (c = "010") or (c = "111") else '0';
  nb <= '1' when (c = "011") or (c = "110") else '0';
  pa <= '1' when (c /= "001") and (c /= "011") else '0';
  pb <= '1' when (c /= "000") and (c /= "010") else '0';
  v <= '1' when ((c = "010") and sa = '1' and sx = '1') or
    ((c = "011") and sb = '1' and sx = '1') or
    ((c = "100") and adder_cout = '1') or
    ((c = "101") and (sa = sb) and (sa /= sx)) or
    ((c = "110") and (sa /= sb) and (sa /= sx)) or
    ((c = "111") and (sa /= sb) and (sb /= sx))
else '0';
end control_arch;

library IEEE;
use IEEE.std_logic_1164.all;
entity arithuvs is
  port (a, b: in STD_LOGIC_VECTOR (3 downto 0);
        c: in STD_LOGIC_VECTOR (2 downto 0);
        x: out STD_LOGIC_VECTOR (3 downto 0);
        v: out STD_LOGIC);
end arithuvs;
architecture arithuvs_arch of arithuvs is
  component flip
    port (en: in STD_LOGIC;
          a: in STD_LOGIC_VECTOR (3 downto 0);
          x: out STD_LOGIC_VECTOR (3 downto 0))
  end component;
  component inc
    port (en: in STD_LOGIC;
          a: in STD_LOGIC_VECTOR (3 downto 0);
          x: out STD_LOGIC_VECTOR (3 downto 0))
  end component;
  component pass
    port (en: in STD_LOGIC;
          a: in STD_LOGIC_VECTOR (3 downto 0);
          x: out STD_LOGIC_VECTOR (3 downto 0))
  end component;
component adder
    port (  
        cin: in STD_LOGIC;  
        a, b: in STD_LOGIC_VECTOR (3 downto 0);  
        x: out STD_LOGIC_VECTOR (3 downto 0);  
        cout: out STD_LOGIC  
    );
end component;

component control
    port (  
        c: in STD_LOGIC_VECTOR (2 downto 0);  
        sa, sb, sx, adder_cout: in STD_LOGIC;  
        na, nb, pa, pb, v: out STD_LOGIC  
    );
end component;

signal a1, a2, a3, b1, b2, b3, xb: STD_LOGIC_VECTOR (3 downto 0);
signal na, nb, pa, pb, adder_cout: STD_LOGIC;
signal zero: STD_LOGIC;
begin
    zero <= '0';
    flip_a: flip port map(na, a, a1);
    flip_b: flip port map(nb, b, b1);
    inc_a: inc port map(na, a1, a2);
    inc_b: inc port map(nb, b1, b2);
    pass_a: pass port map(pa, a2, a3);
    pass_b: pass port map(pb, b2, b3);
    addem: adder port map(zero, a3, b3, xb, adder_cout);
    ctl: control port map(c, a(3), b(3), xb(3), adder_cout,  
                           na, nb, pa, pb, v);

    x <= xb;
end arithuvs_arch;
2. Write a VHDL specification for a 3 bit multiplier with 6 output bits. Your specification should include a 3 bit adder module. Use structural VHDL to combine the adder modules to form the multiplier. Perform a unit delay simulation of your circuit for all 64 possible pairs of input values and display the results so that inputs and outputs are represented numerically (that is, combine the signals to form a bus) so that the results are easy to check. In your simulation, set the simulation precision to 1 ns and have the inputs change once every 20 ns.

library IEEE;
use IEEE.std_logic_1164.all;

entity adder is
  port (cin: in STD_LOGIC;
         a, b: in STD_LOGIC_VECTOR (2 downto 0);
         x: out STD_LOGIC_VECTOR (2 downto 0);
         cout: out STD_LOGIC);
end adder;

architecture adder_arch of adder is
  signal c: STD_LOGIC_VECTOR (2 downto 0);
begin
  c(0) <= (a(0) and b(0)) or (a(0) and cin) or (b(0) and cin);
  c(1) <= (a(1) and b(1)) or (a(1) and c(0)) or (b(1) and c(0));
  c(2) <= (a(2) and b(2)) or (a(2) and c(1)) or (b(2) and c(1));
  cout <= c(2);
  x(0) <= a(0) xor (b(0) xor cin);
  x(1) <= a(1) xor (b(1) xor c(0));
  x(2) <= a(2) xor (b(2) xor c(1));
end adder_arch;

library IEEE;
use IEEE.std_logic_1164.all;

entity mult is
  port (a, b: in STD_LOGIC_VECTOR (2 downto 0);
        x: out STD_LOGIC_VECTOR (5 downto 0));
end mult;

architecture mult_arch of mult is
  component adder
    port (cin: in STD_LOGIC;
          a, b: in STD_LOGIC_VECTOR (2 downto 0);
          x: out STD_LOGIC_VECTOR (2 downto 0);
          cout: out STD_LOGIC);
  end component;
  signal pp0, pp1, pp2, spp0: STD_LOGIC_VECTOR (2 downto 0);
  signal isum: STD_LOGIC_VECTOR (3 downto 0);
  signal zero: STD_LOGIC;
begin
  zero <= '0';
pp0 <= a when b(0) = '1' else "000";
pp1 <= a when b(1) = '1' else "000";
pp2 <= a when b(2) = '1' else "000";
spp0 <= '0' & pp0(2 downto 1);
add0: adder port map(zero, pp1, spp0, isum(2 downto 0), isum(3));
add1: adder port map(zero, pp2, isum(3 downto 1), x(4 downto 2),
x(5));
x(1) <= isum(0); x(0) <= pp0(0);
end mult_arch;
3. Consider the flip flop in Figure 4-35 of Mano & Kime. Construct (by hand) a timing diagram for this circuit to determine how it behaves. Show the \( C \) and \( D \) inputs, the intermediate \( S \) and \( R \) signals and the \( Q \) and \( Q' \) outputs.

To understand how the circuit works, consider the diagram shown below.

The first thing to do is note that when \( C = 0 \), the \( S \) and \( R \) signals are both high. This means that the output latch just retains its old value. So, whenever \( C = 0 \), there is no change to \( Q \) and \( Q' \). Also, when \( C = 0 \), signal \( B \) equals \( D' \) (the complement of the \( D \) input). Because of this, signal \( A \) equals \( D \).

Now, consider what happens when \( C \) changes from 0 to 1, while \( D \) stays stable. The two inner NANDs on the left hand side are now enabled. So in particular, the second one from the top propagates the value on the other input to its output (complementing it, as it propagates). This makes the \( S \) signal equal to \( D' \). Now, the second NAND from the bottom has \( C = 1 \) at one input and has \( D' \) at both of its other inputs. So its output will be \( D \). That is \( R \) will be \( D \). So now the output latch has complementary values on its inputs and will propagate these to the outputs, so that we will have \( Q = D \) and \( Q' = D' \).

Now, the last thing to note is that after \( C \) goes high, subsequent changes to the \( D \) input don't affect the stored value, because after the transition, the feedback input of the bottom NAND is equal to \( D \). If the \( D \) input now changes to \( D' \), this NAND will have different values on its two inputs, so signal \( B \) will be high. But this high output just reinforces the feedback through the top NAND gate, without allowing any other signal values to change. So all changes to \( D \) that follow the rising clock edge have no effect on the output value.
So the circuit operates as a positive edge-triggered $D$ flip flop. Note that it is important that the $D$ input stays stable when the clock is rising. If this condition is not met, the operation of the circuit is unpredictable and can include oscillations on the outputs.
4. Derive a state transition diagram ("bubble" diagram) for the sequential circuit shown below. Give a set of test inputs for this circuit that will verify every transition in the state diagram (including "self-loops").

The following sequence exercises all the transitions.

\[
D = 0010 \ 1011 \ 10 \\
R = 1110 \ 0001 \ 01
\]