1. (40 points) Implement and simulate a single digit BCD adder that uses the excess-3 representation. Your circuit will have two sets of four inputs $a=a_3,a_2,a_1,a_0$ and $b=b_3,b_2,b_1,b_0$. It will also have four outputs $x=x_3,x_2,x_1,x_0$. The output $x$ of your circuit should be the excess-3 sum of the input values $a$ and $b$. So for example, if $a=0100$ (representing the value 1) and $b=1000$ (representing the value 5), the output $x=1001$ (representing the value 6). Your circuit should correctly wrap-around if the two input values are too large. So for example, if $a=0111$ (representing the value 4) and $b=1011$ (representing the value 8), the output $x=0101$ (representing the value 2). Your circuit should also have a carry input $C_{in}$ and a carry output $C_{out}$. A block diagram of a circuit that implements the single digit BCD adder is shown below. The two large blocks are ordinary 4 bit binary adders. In your design notes, include an explanation for why this design produces the correct excess-3 sum and the correct value for $C_{out}$.

**Design notes.** The block diagram for the excess-3 adder appears above. The first thing to notice about this circuit is that the first adder produces an “excess-6” sum, since each of its inputs is excess-3. This means that when the two decimal digits sum to 9, the output of the first adder will be 15 and when the two decimal digits sum to 10, the output of the first adder will be 0 with $C_{out}=1$. More generally, $C_{out}$ will be high if and only if the two decimal digits sum to something larger than 9 and these are exactly the cases for which a carry should be generated. To get an excess-3 sum, we need to add a correction to the output of the first adder. This is what the second adder is for. When there is no carry generated in the first adder, we correct by subtracting 3 (converting an excess-6 value to an excess-3 value).
excess-3 value). When there is a carry, we correct by adding 3. To see this, recall that when the two input digits sum to 10, the coded value from the first adder is 0000. By adding 3 to this, we get 0011, which is the excess-3 representation for 0. It’s easy to see that the circuit in the block diagram adds 0011 when $C_{out} = 1$. When $C_{out} = 0$ it effectively subtracts 3 by adding the 2s-complement of 3. It does this in two steps. First, it adds 1100 (the bit-wise complement of 0011), then it adds 0001 to this, by asserting the carry input of the second adder. The schematic appears below. This corresponds directly to the block diagram.

Use the schematic editor to create a schematic for this circuit and simulate it. You may use the four bit adder component in the schematic editor’s symbol library (you’ll find it in the arithmetic section of the library). Ignore the OFL output (this is used when doing signed arithmetic). Since the circuit is too large for exhaustive testing, select test cases that demonstrate that the circuit works correctly and include an explanation for why these test cases are sufficient. Include “boundary cases” such as adding 0 or 1 and input combinations that are just large enough to generate a carry. Turn in a copy of your design notes, the schematic and the simulation results.
To test the circuit, we start with 19 input combinations that produce all the distinct sum values. The input combinations are 0+0, 0+1, 0+2, 0+3, 0+4, 0+5, 0+6, 0+7, 0+8, 0+9, 1+9, 2+9, 3+9, 4+9, 5+9, 6+9, 7+9, 8+9, 9+9. Note that these input values also produce all possible output values from the first adder. We then repeat all of these with Cin high. Next, we have a set of cases that check the boundary conditions for the Cout signal. We use the input combinations 0+9, 1+8, 2+7, 3+6, 4+5, 5+4, 6+3, 7+2, 8+1, 9+0 with Cin=0 and with Cin=1. This results in sums that are at the boundary of the condition that generates the carry.

While these tests are not an exhaustive check, they provide some confidence that the circuit works correctly, since they produce all possible outputs from the first adder and test the key critical cases where logical errors are most likely to lead to discrepancies between the circuit output and the correct output. The simulation results appears below.

Note that these tests are arranged to make them easy to check. In the first two sections, the output values increase sequentially (as they should) and wrap around at the appropriate place. Also, note the use of combined values in unsigned format in the simulation output. This makes the output relatively easy to check.
2. (60 points) In this problem, you will use VHDL to implement a 3 digit BCD adder that uses the excess-3 format. Your circuit will have two data inputs \( A = A_2 A_1 A_0 \) (where each \( A_i \) is a four bit signal representing a BCD digit in excess-3 notation) and \( B = B_2 B_1 B_0 \) and an output \( X = X_2 X_1 X_0 \) which will equal the sum of the inputs. So if \( A \) is an encoded representation of 237 and \( B \) is an encoded representation of 359, then \( X \) will be an encoded representation of 596. Your circuit should also have a single carry input bit \( C_{in} \) and generate a single carry output bit \( C_{out} \). Start by first creating an entity that implements a single digit BCD adder, based on the design from the previous problem. Combine these using structural VHDL to create the 3 digit adder.

Design notes. The 3 digit adder can be constructed from multiple copies of a single digit adder. The single digit adder works the same way as the design in the previous problem. We first add the two input values together, producing a five bit intermediate value. The extra bit represents the carry out of the four bit sum. The value of this bit becomes the carry-out of the excess-3 adder. If the carry-out is high, we then obtain the desired result by adding three to the intermediate value. If the carry-out is low, we subtract three from the intermediate value. The 3 digit adder is implemented by instantiating this component three times and connecting the carry-in and carry-out signals together, just as we would do for a ripple-carry binary adder.

VHDL code. The VHDL implementing this approach appears below.

```vhdl
-- Design problem 2.2 - 3 digit excess 3 adder
-- Jon Turner (jon.turner@wustl.edu)
--
-- Uses ripple-carry architecture, using single-digit excess-3 adder components.
--------------------------------------------------------------
-- Single digit excess-3 adder component
-- Operates in two stages. First, adds the two input digits producing
-- an excess-6 intermediate result, and generating the base-10 carry.
-- The excess-6 result is then corrected by subtracting 3 when there
-- is no carry and by adding 3 when there is a carry.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ex3add1 is Port (
  A : in std_logic_vector(3 downto 0);
  B : in std_logic_vector(3 downto 0);
  X : out std_logic_vector(3 downto 0);
  Cin : in std_logic;
  Cout : out std_logic);
end ex3add1;
architecture a1 of ex3add1 is
signal ax, bx, xx : std_logic_vector(4 downto 0);
begin
  ax <= "0" & A; bx <= "0" & B; -- extended versions of inputs
  xx <= ax + bx + Cin; -- excess-6 intermediate result
  Cout <= xx(4); -- carry-out if carry from xx
  X <= xx(3 downto 0) + "0011" when xx(4) = '1' else
      xx(3 downto 0) - "0011";
end a1;
```
-- 3 digit excess-3 adder.
-- Ripple-carry design, using structural VHDL to
-- combine 3 single digit adder components.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ex3add3 is Port (
    A2, A1, A0 : in std_logic_vector(3 downto 0);
    B2, B1, B0 : in std_logic_vector(3 downto 0);
    X2, X1, X0 : out std_logic_vector(3 downto 0);
    Cin : in std_logic;
    Cout : out std_logic);
end ex3add3;

architecture a1 of ex3add3 is
component ex3add1 Port ( 
    A : in std_logic_vector(3 downto 0);
    B : in std_logic_vector(3 downto 0);
    X : out std_logic_vector(3 downto 0);
    Cin : in std_logic;
    Cout : out std_logic);
end component;
signal carry: std_logic_vector(3 downto 0);
begin
    carry(0) <= Cin;
    dig0:ex3add1 port map(A0,B0,X0,carry(0),carry(1));
    dig1:ex3add1 port map(A1,B1,X1,carry(1),carry(2));
    dig2:ex3add1 port map(A2,B2,X2,carry(2),carry(3));
    Cout <= carry(3);
end a1;
Do a functional simulation of the circuit to verify its logical correctness. Select test cases that demonstrate that the circuit works correctly and explain why these are sufficient to give you confidence that it will work correctly under all conditions, not just the specific cases simulated.

The simulation results are shown below. The simulation starts with a repeat of the test sequence used in problem 1, applied to the low order digits alone. The inputs to the other digits are left at 0 (3 in encoded form) for these tests. This allows us to verify that the VHDL implementation of the single digit adder is correct. The last section of the simulation is a series tests selected to show that the carry propagates between stages of the adder as it should. In these tests, the input values sum to 999 (CCC in encoded form) and the carry input alternates between 0 and 1. This results in the sum going from 999 to 000 (CCC to 333 in encoded form). Because these tests also result in a carry rippling from the low-order digit to the high-order digit, they produce the worst-case delay for the circuit.
Do a timing simulation (post place & route) to evaluate the delay through the circuit. Select test inputs that can be expected to cause the largest possible delay and justify your choices. Add extra cursors to the simulation display, so you can measure the delay from the time the inputs change to the time the outputs change.

The timing simulation results are shown below. Specifically, we show representative samples from the end of the simulation run described previously. In this section, the input values sum to 999, causing the intermediate values generated within the single digit adders to be \text{FFF}. The carry input alternates between 0 and 1, causing changes to carry values to propagate through all three digits. The first result below shows a case where the carry input goes from 1 to 0. We see that the delay for the carry is 10.7 ns, or just under 3.6 ns per digit (900 ps for each bit in the underlying binary adders). The final output value stabilizes about 1.8 ns later, giving an overall delay of 12.5 ns. The second result below shows the case where the carry input changes from 0 to 1. The delay for the carry propagation is virtually identical to the first case. There is a slightly smaller delay before the data outputs stabilize, but the difference is not very significant. Based on these results, I would expect a 16 digit adder to have a delay of about $16 \times 3.6 + 1.8$ which is about 60 ns.
Examine the synthesis report produced by the CAD tools and turn in a printout of the synthesis report. Highlight the part of the synthesis report that describes the number of circuit elements used to implement the circuit. Because the CAD tools are synthesizing a circuit for an FPGA, the primitive element used to construct the circuit is a lookup table or LUT. There are LUTs with 1, 2, 3 and 4 inputs. How many total LUTs are used for your design? How does this compare to the number of binary adder bits required to implement the circuit? Based on this, how many would you expect to be used in a 16 digit excess-3 adder using the same ripple-carry architecture?

The relevant portion of the synthesis report appears below. A total of 27 LUTs are used in the design. This is exactly the same as the number of binary adder bits (each single digit adder contains a 5 bit adder and a 4 bit adder). Based on this, a 16 digit excess 3 adder can be expected to require $16 \times 9 = 144$ LUTs.

```plaintext
* Final Report

Final Results
RTL Top Level Output File Name : ex3add3.ngr
Top Level Output File Name : ex3add3
Output Format : NGC
Optimization Goal : Speed
Keep Hierarchy : NO

Design Statistics
# IOs : 38

Macro Statistics :
# Adders/Subtractors : 6
# 4-bit addsub : 3
# 5-bit adder carry in : 3

Cell Usage :
# BELS : 72
# LUT1 : 3
# LUT2 : 24
# MUXCY : 21
# XORCY : 24
# IO Buffers : 38
# IBUF : 25
# OBUF : 13

Device utilization summary:

Selected Device : 2v500fg256-6
Number of Slices : 14 out of 3072 0%
Number of 4 input LUTs : 27 out of 6144 0%
Number of bonded IOBs : 38 out of 172 22%
```
Highlight the part of the synthesis report that shows the delay for a worst-case path through the circuit. Compare this delay to what you observed from the simulation. How much of the delay shown in the synthesis report is caused by the input and output buffers for the circuit? Based on the synthesis report and your simulation results, estimate the worst-case delay for a 16 digit excess-3 adder using the same architecture.

The relevant portion of the synthesis report appears below. The worst-case delay cited in the synthesis report is just over 9 ns. This is almost 30% less than the 12.5 ns that we found in simulation. The synthesis report uses an estimated worst-case delay, so it is not that surprising that it does not agree exactly with the simulation output. The report shows that the input buffer accounts for .653+.382 ns or just 1 ns. The output buffer accounts for 3.7 ns, so the total is about 4.7 ns. If we remove this 4.7 ns from the 9 ns reported by the synthesis report, we get 4.3 ns or about 1.4 ns per digit for the internal circuitry. This suggests that for a 16 bit version of the adder, the synthesis report would give a delay of 4.7+16×1.4 or 27 ns. The simulation would probably give an estimate that is 30% larger or just under 36 ns. In fact, this estimate is too high, because the first and last stages contribute more delay than the intermediate stages. By looking more closely at the synthesis report, we see that the “middle” digit (digit 1) contributes a delay of .042×4=.168 ns. If we add .168×13 to the original 12.5 ns estimated by the synthesis report, we get a total of about 11.2 ns. And in fact, if we convert the design to a 16 bit design, this is exactly what we get back from the synthesis report. The simulation can be expected to produce a worst-case delay that is 30% larger, or about 15 ns. The synthesis report for this modified design also reports a total of 144 LUTs, matching the earlier estimate.
<table>
<thead>
<tr>
<th>Component</th>
<th>Type</th>
<th>Delay 1</th>
<th>Delay 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT1: I0-&gt;O</td>
<td></td>
<td>0.347</td>
<td>0.383</td>
<td>dig2_n00011 (dig2_n0001)</td>
</tr>
<tr>
<td>MUXCY: CI-&gt;O</td>
<td></td>
<td>0.042</td>
<td>0.000</td>
<td>dig2_ex3add1 X&lt;0&gt;cy (dig2_ex3add1 X&lt;0&gt;_cyo)</td>
</tr>
<tr>
<td>MUXCY: CI-&gt;O</td>
<td></td>
<td>0.042</td>
<td>0.000</td>
<td>dig2_ex3add1 X&lt;1&gt;cy (dig2_ex3add1 X&lt;1&gt;_cyo)</td>
</tr>
<tr>
<td>MUXCY: CI-&gt;O</td>
<td></td>
<td>0.042</td>
<td>0.000</td>
<td>dig2_ex3add1 X&lt;2&gt;cy (dig2_ex3add1 X&lt;2&gt;_cyo)</td>
</tr>
<tr>
<td>XORCY: CI-&gt;O</td>
<td></td>
<td>0.824</td>
<td>0.383</td>
<td>X2_3_OBUF</td>
</tr>
<tr>
<td>OBUF: I-&gt;O</td>
<td></td>
<td>3.743</td>
<td></td>
<td>X2_3_OBUF (X2&lt;3&gt;)</td>
</tr>
</tbody>
</table>

Total: 9.030ns (7.296ns logic, 1.734ns route)  
(80.8% logic, 19.2% route)