CpE 487 Digital Design Lab

Lab 4: Hex Calculator

1. Introduction

In this lab, we will program the FPGA on the *Nexys2* board to function as a simple hexadecimal calculator capable of adding and subtracting four-digit hexadecimal numbers. We will use a 16-buttom keypad to input hex numbers and the push-buttons on the *Nexys2* board to act as operation buttons (+, –, = and clear). The results will be displayed on the four 7-segment displays on the Nexys2 board. The reason we build a hex calculator (rather than a decimal one) is that it makes the 7-segment display operation much simpler. Converting the output to decimal format would require a large amount of additional VHDL code.

2. Keypad Interface

We will be using a 16-button keypad module *PmodKYPD* which plugs into one of the PMOD connectors on the *Nexys2* board. The module and its connections are shown in Figure 1. The 16-buttons are labeled "0 1 2 3 4 5 6 7 8 9 A B C D E F", so we can use this keyboard to enter numbers in hexadecimal format. Documentation on the module can be found on the course website.

Connector J1 – Column/Row Indicators		
Pin	**Signal**	**Description**
1	COL4	Column 4
2	COL3	Column 3
3	COL2	Column 2
4	COL1	Column 1
5	GND	Power Supply Ground
6	VCC	Power Supply (3.3V)
7	ROW4	Row 4
8	ROW3	Row 3
9	ROW2	Row 2
10	ROW1	Row 1
11	GND	Power Supply Ground
12	VCC	Power Supply (3.3V)

Figure 1 PmodKYPD 16-button Keypad

The keyboard is set up as a matrix in which each row of buttons from left to right are tied to a row pin, and each column from top to bottom is tied to a column pin. There are four row pins and four column pins. When a button is pushed, it connects its row pin to its column pin.

Each row is normally held high ('1') by a pull-up resistor connected to VCC. To test the buttons in a particular column, that column should be driven low ('0'), while all other columns are held high. If a button in that row is depressed while its column pin is low, it will pull the corresponding row pin low. The FPGA can therefore read the state of the buttons by driving the column lines and reading the row lines. The FPGA "walks" a logic '0' through each column line

(in turn, while keeping the other column lines high) and reads the row pins to determine if any buttons have been pushed.

3. Hardware Setup

Plug the *PmodKYPD* module into PMOD jack *JA1* on the *Nexys2* boards as shown in Figure 2. Note that the *PmodKYPD* uses all 12 pins of jack JA1 and must be connected with a 12 pin cable. Note the white alignment marks on the connectors as shown in Figure 2. The alignment mark should be on the top-right for the connector attaching to the *Nexys2* board and the top-left for the connector attaching to the *PmodKYPD* module. Please be careful not to bend the pins on the *PmodKYPD* module.

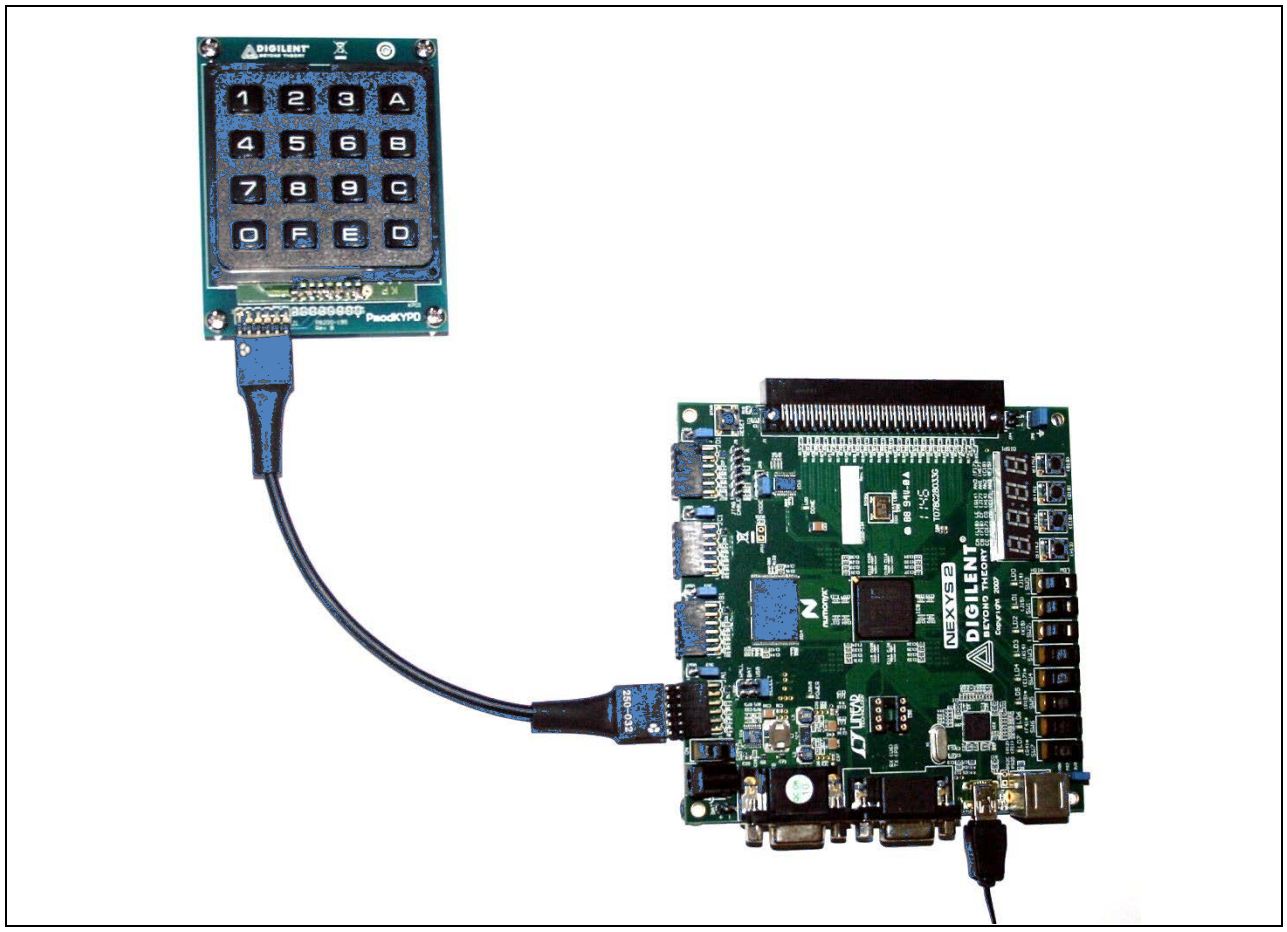

Figure 2 PmodKYPD Module connected to PMOD jack JA1

4. Configuring the FPGA

4.1 Create a New Project

Use the Xilinx ISE software to create a new project named *Hexcalc* using the same *project settings* as in Labs 1 and 2.

4.2 Add Source for "keypad"

Create a new VHDL source module called *keypad* and load the following source code into the edit window. Expand the **Synthesize** command in the *Process* window and run **Check Syntax** to verify that you have entered the code correctly.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity keypad is
   Port ( samp_ck : in  STD_LOGIC;                          -- clock to strobe columns
          col : out  STD_LOGIC_VECTOR (4 downto 1);         -- output column lines
          row : in  STD_LOGIC_VECTOR (4 downto 1);          -- input row lines
          value : out  STD_LOGIC_VECTOR (3 downto 0);       -- hex value of key depressed
          hit : out  STD_LOGIC);                            -- indicates when a key has been pressed
end keypad;

architecture Behavioral of keypad is
signal CV1, CV2, CV3, CV4: std_logic_vector (4 downto 1) := "1111";  -- column vector of each row
signal curr_col: std_logic_vector (4 downto 1) := "1110";            -- current column code
begin

-- This process synchronously tests the state of the keypad buttons. On each edge of samp_ck,
-- this module outputs a column code to the keypad in which one column line is held low while the
-- other three column lines are held high. The row outputs of that column are then read
-- into the corresponding column vector. The current column is then updated ready for the next
-- clock edge. Remember that curr_col is not updated until the process suspends.

strobe_proc: process
        begin
                wait until rising_edge(samp_ck);
                case curr_col is
                when "1110" =>
                        CV1 <= row; curr_col <= "1101";
                when "1101" =>
                        CV2 <= row; curr_col <= "1011";
                when "1011" =>
                        CV3 <= row; curr_col <= "0111";
                when "0111" =>
                        CV4 <= row; curr_col <= "1110";
                when others =>
                        curr_col <= "1110";
                end case;
        end process;

-- This process runs whenever any of the column vectors change. Each vector is tested to see
-- if there are any '0's in the vector. This would indicate that a button had been pushed in
-- that column. If so, the value of the button is output and the hit signal is assereted. If
-- not button is pushed, the hit signal is cleared
```

```
out_proc: process (CV1, CV2, CV3, CV4)
        begin
                hit <= '1';
                if      CV1(1) = '0' then value <= X"1";
                elsif CV1(2) = '0' then value <= X"4";
                elsif CV1(3) = '0' then value <= X"7";
                elsif CV1(4) = '0' then value <= X"0";
                elsif CV2(1) = '0' then value <= X"2";
                elsif CV2(2) = '0' then value <= X"5";
                elsif CV2(3) = '0' then value <= X"8";
                elsif CV2(4) = '0' then value <= X"F";
                elsif CV3(1) = '0' then value <= X"3";
                elsif CV3(2) = '0' then value <= X"6";
                elsif CV3(3) = '0' then value <= X"9";
                elsif CV3(4) = '0' then value <= X"E";
                elsif CV4(1) = '0' then value <= X"A";
                elsif CV4(2) = '0' then value <= X"B";
                elsif CV4(3) = '0' then value <= X"C";
                elsif CV4(4) = '0' then value <= X"D";
                else hit <= '0'; value <= X"0";
                end if;
        end process;

        col <= curr_col;

end Behavioral;
```

This module synchronously tests the state of the keypad buttons. On each rising edge of the clock *samp_ck* (approx. every 1.5 ms) the module outputs a code to the 4 column lines of the switch matrix, a code which will enable exactly one column. The first time it outputs 1110, setting column 1 low and all others high. On the second edge, it outputs 1101, setting column 2 low and all others high, continuing to columns 3 and 4 and then returning to column 1 to repeat the process. Each time it enables one of the columns, it reads the row lines into a column vector to determine if any button in that column has been pushed. A separate process runs whenever any of the column vectors change. When that happens, the process searches for a ‘0’ in any of the column vectors. If it finds a ‘0’, it outputs the appropriate hex value and asserts the *hit* signal. If there are no ‘0’s in the column vectors (meaning no buttons are currently being pushed), it clears the *hit* signal. The code assumes that only one button is pushed at a time.

4.2 Add Source for “leddec16”

Create a new VHDL source module called *leddec16* and load the following source code into the edit window. Expand the **Synthesize** command in the *Process* window and run **Check Syntax** to verify that you have entered the code correctly.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity leddec16 is
    Port (dig : in  STD_LOGIC_VECTOR (1 downto 0);          -- which digit to currently display
          data : in STD_LOGIC_VECTOR (15 downto 0);         -- 16-bit (4-digit) data
          anode: out STD_LOGIC_VECTOR (3 downto 0);         -- which anode to turn on
          seg : out  STD_LOGIC_VECTOR (6 downto 0));        -- segment code for current digit
end leddec16;

architecture Behavioral of leddec16 is
signal data4: STD_LOGIC_VECTOR (3 downto 0);      -- binary value of current digit
begin

-- Select digit data to be displayed in this mpx period

          data4 <= data(3 downto 0) when dig="00" else          --digit 0
                   data(7 downto 4) when dig="01" else          --digit 1
                   data(11 downto 8) when dig="10" else         --digit 2
                   data(15 downto 12);                          --digit 3

-- Turn on segments corresponding to 4-bit data word
          seg <=  "0000001" when data4="0000" else      --0
                  "1001111" when data4="0001" else      --1
                  "0010010" when data4="0010" else      --2
                  "0000110" when data4="0011" else      --3
                  "1001100" when data4="0100" else      --4
                  "0100100" when data4="0101" else      --5
                  "0100000" when data4="0110" else      --6
                  "0001111" when data4="0111" else      --7
                  "0000000" when data4="1000" else      --8
                  "0000100" when data4="1001" else      --9
                  "0001000" when data4="1010" else      --A
                  "1100000" when data4="1011" else      --B
                  "0110001" when data4="1100" else      --C
                  "1000010" when data4="1101" else      --D
                  "0110000" when data4="1110" else      --E
                  "0111000" when data4="1111" else      --F
                  "1111111";

-- Turn on anode of 7-segment display addressed by 2-bit digit selector dig
          anode <=      "1110" when dig="00" else       -- digit 0
                        "1101" when dig="01" else       -- digit 1
                        "1011" when dig="10" else       -- digit 2
                        "0111" when dig="11" else       -- digit 3
                        "1111";
end Behavioral;
```

This module is similar to the code developed in Lab 2 to display a 16-bit hex counter. The four 7-segment displays are time multiplexed using an input signal *dig* which counts from 0 to 3 at a rate of approx. 380 Hz. The four digits are thus refreshed approx. 95 times per second. The segment code determines which anode to take low ('0'). It also determines which 4-bits of the 16-bit data to display. These four bits are translated into a 7-segment code and output to the common display segment lines.

4.3 Add Source for top level "hexcalc"

Create a new VHDL source module called *hexcalc* and load the following source code into the edit window. Expand the **Synthesize** command in the *Process* window and run **Check Syntax** to verify that you have entered the code correctly.

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

entity hexcalc is
Port (    clk_50MHz : in  STD_LOGIC;                                  -- system clock (50 MHz)
          SEG7_anode : out STD_LOGIC_VECTOR (3 downto 0);  -- anodes of four 7-seg displays
          SEG7_seg : out STD_LOGIC_VECTOR (6 downto 0);     -- common segments of 7-seg displays
                    bt_clr: in STD_LOGIC;                              -- calculator "clear" button
                    bt_plus: in STD_LOGIC;                           -- calculator "+" button
                    bt_eq: in STD_LOGIC;                              -- calculator "-" button
                    KB_col: out STD_LOGIC_VECTOR (4 downto 1);       -- keypad column pins
                    KB_row: in STD_LOGIC_VECTOR (4 downto 1 ));      -- keypad row pins
end hexcalc;

architecture Behavioral of hexcalc is
component keypad is
   Port ( samp_ck : in  STD_LOGIC;
          col : out  STD_LOGIC_VECTOR (4 downto 1);
          row : in  STD_LOGIC_VECTOR (4 downto 1);
          value : out  STD_LOGIC_VECTOR (3 downto 0);
          hit : out  STD_LOGIC);
end component;
component leddec16 is
   Port ( dig : in  STD_LOGIC_VECTOR (1 downto 0);
                       data : in STD_LOGIC_VECTOR (15 downto 0);
                              anode: out STD_LOGIC_VECTOR (3 downto 0);
          seg : out  STD_LOGIC_VECTOR (6 downto 0));
end component;

signal cnt: std_logic_vector(20 downto 0);          -- counter to generate timing signals
signal kp_clk, kp_hit, sm_clk: std_logic;
signal kp_value: std_logic_vector (3 downto 0);
signal nx_acc, acc: std_logic_vector (15 downto 0);          -- accumulated sum
```

```
signal nx_operand, operand: std_logic_vector (15 downto 0);      -- operand
signal display: std_logic_vector (15 downto 0);                  -- value to be displayed
signal led_mpx: STD_LOGIC_VECTOR (1 downto 0);                   -- 7-seg multiplexing clock


type state is (ENTER_ACC, ACC_RELEASE, START_OP, OP_RELEASE,
                    ENTER_OP, SHOW_RESULT);          -- state machine states
signal pr_state, nx_state: state;                    -- present and next states

begin

ck_proc: process(clk_50MHz)
        begin
                if rising_edge( clk_50MHz) then          -- on rising edge of clock
                        cnt <= cnt+1;                    -- increment counter
                end if;
        end process;
        kp_clk <= cnt(15);                               -- keypad interrogation clock
        sm_clk <= cnt(20);                               -- state machine clock
        led_mpx <= cnt(18 downto 17);                    -- 7-seg multiplexing clock

kp1:    keypad port map (samp_ck => kp_clk, col => KB_col,
                        row => KB_row, value => kp_value, hit => kp_hit);

led1:   leddec16 port map (dig => led_mpx, data => display,
                        anode => SEG7_anode, seg => SEG7_seg);

sm_ck_pr: process (bt_clr, sm_clk)       -- state machine clock process
        begin
                if bt_clr = '1' then                             -- reset to known state
                        acc <= X"0000";
                        operand <= X"0000";
                        pr_state <= ENTER_ACC;
                elsif rising_edge (sm_clk) then                  -- on rising clock edge
                        pr_state <= nx_state;                    -- update present state
                        acc <= nx_acc;                           -- update accumulator
                        operand <= nx_operand;                   -- update operand
                end if;
        end process;

-- state maching combinatorial process
-- determines output of state machine and next state
sm_comb_pr: process (kp_hit, kp_value, bt_plus, bt_eq, acc, operand, pr_state)
        begin
                nx_acc <= acc;                   -- default values of nx_acc, nx_operand & display
                nx_operand <= operand;
                display <= acc;
                case pr_state is                 -- depending on present state...
                when ENTER_ACC =>                -- waiting for next digit in 1st operand entry
                        if kp_hit = '1' then
                                nx_acc <= acc(11 downto 0) & kp_value;
```

```
                        nx_state <= ACC_RELEASE;
                elsif bt_plus = '1' then
                        nx_state <= START_OP;
                else nx_state <= ENTER_ACC;
                end if;
        when ACC_RELEASE =>             -- waiting for button to be released
                if kp_hit = '0' then
                        nx_state <= ENTER_ACC;
                else nx_state <= ACC_RELEASE;
                end if;
        when START_OP =>            -- ready to start entering 2nd operand
                if kp_hit = '1' then
                        nx_operand <= X"000" & kp_value;
                        nx_state <= OP_RELEASE;
                        display <= operand;
                else nx_state <= START_OP;
                end if;
        when OP_RELEASE =>              -- waiting for button ot be released
                display <= operand;
                if kp_hit = '0' then
                        nx_state <= ENTER_OP;
                else nx_state <= OP_RELEASE;
                end if;
        when ENTER_OP =>            -- waiting for next digit in 2nd operand
                display <= operand;
                if bt_eq = '1' then
                        nx_acc <= acc + operand;
                        nx_state <=SHOW_RESULT;
                elsif kp_hit = '1' then
                        nx_operand <= operand(11 downto 0) & kp_value;
                        nx_state <= OP_RELEASE;
                else nx_state <= ENTER_OP;
                end if;
        when SHOW_RESULT =>             -- display result of addition
                if kp_hit = '1' then
                        nx_acc <= X"000" & kp_value;
                        nx_state <= ACC_RELEASE;
                else nx_state <= SHOW_RESULT;
                end if;
        end case;
    end process;

end Behavioral;
```

This is the top level that hooks it all together. This module:

a) Creates an instance of the keypad interface and 7-segment decoder interface modules
b) Make connection to the external keypad, display and buttons
c) Has a timing process to generate clocks for the keypad, display multiplexer and state machine

d) Implements a finite state machine to implement the operations of the calculator in response to button pushes.

The finite state machine uses a number of variables to keep track of the addition operation. The variable *acc* is an accumulator that holds the current summation result. The variable *operand* holds the value of the next operand that will be added to the accumulator. The variable *display* holds the value currently being displayed on the 7-segment displays. The variable *pr_state* is the current state of the state machine. Depending on the current state, the machine will react to pushed keypad buttons or operation buttons to update variables, change the output and select the next state. Operation of the state machine is summarized in Figure 3.

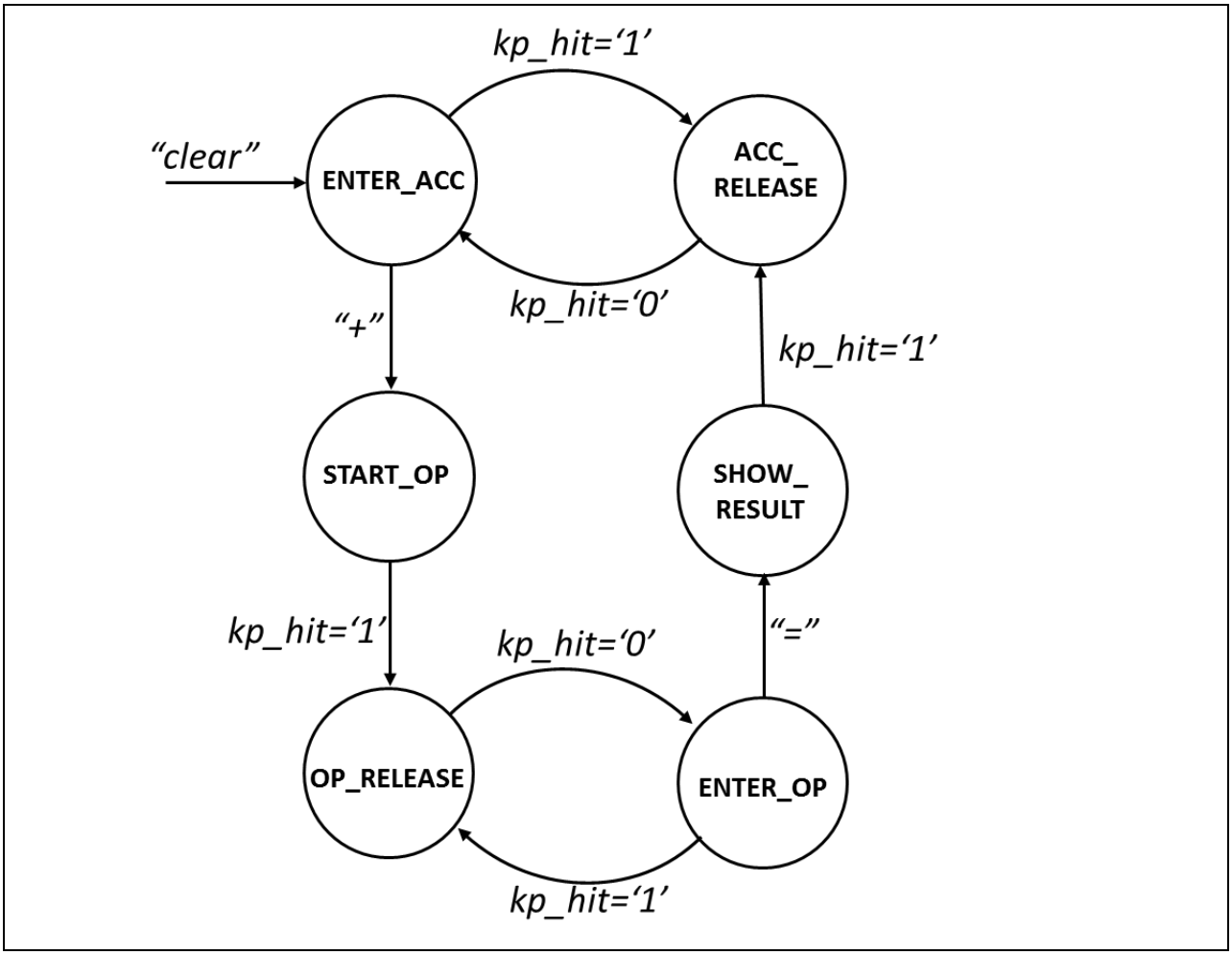

Figure 3 Calculator State Transition Diagram

When the *clear* button is pushed, the machine enters the ENTER_ACC state. In this state the machine waits for a keypad button to be pushed. When a keypad button is pushed, the code adds the new digit to the 16-bit word in the accumulator and then waits in the ACC_RELEASE state for the button to be released. It then returns to the ENTER_ACC state to wait for the next digit. This process continues until the "+" button is pushed. The machine then enters the START_OP state where it waits for the first digit of the second operand. Once a keypad button has been pushed, it records the hex digit and then enters the OP_RELEASE state waiting for the keypad

button to be released. Once released, the machine enters the ENTER_OP state where it continues to received operand digits each time a keypad button is pushed. This continues until the user presses the "=' button at which point it performs the addition and goes to the SHOW_RESULT state. Once in the SHOW_RESULT state, it shows the result of the addition and waits for a keypad button push to start a new calculation.

4.4 Synthesis and Implementation

Highlight the *hexcalc* module in the *Hierarchy* window and execute the **Synthesis** command in the *Process* window.

Add an Implementation Constraint source file *hexcalc.ucf* and enter the following data into the edit window:

```
NET     "clk_50MHz"     LOC = B8;

NET     "KB_col(1)"     LOC = M15;
NET     "KB_col(2)"     LOC = L17;
NET     "KB_col(3)"     LOC = K12;
NET     "KB_col(4)"     LOC = L15;

NET     "KB_row(1)"     LOC = M16;
NET     "KB_row(2)"     LOC = M14;
NET     "KB_row(3)"     LOC = L16;
NET     "KB_row(4)"     LOC = K13;

NET     "bt_clr"        LOC = H13;     # BTN3
NET     "bt_plus"       LOC = B18;     # BTN0
NET     "bt_eq"         LOC = D18;     # BTN1

NET "SEG7_seg[0]" LOC = H14;
NET "SEG7_seg[1]" LOC = J17;
NET "SEG7_seg[2]" LOC = G14;
NET "SEG7_seg[3]" LOC = D16;
NET "SEG7_seg[4]" LOC = D17;
NET "SEG7_seg[5]" LOC = F18;
NET "SEG7_seg[6]" LOC = L18;

NET "SEG7_anode[0]" LOC = F17;
NET "SEG7_anode[1]" LOC = H17;
NET "SEG7_anode[2]" LOC = C18;
NET "SEG7_anode[3]" LOC = F15;
```

The configuration file sets up push-button BTN0 as the "+" key, BTN1 as the "=" key and BTN3 as the "clear" key.

Now highlight the *hexcalc* module in the Hierarchy window and run **Implement Design** followed by **Generate Programming File** (don't forget to change the *FPGA Start-up Clock* to be the *JTAG Clock*).

4.5 Download and Run

Use the *Adept* software to download your configuration file *hexcalc.bit*. Once loaded, you should see "0000" appear on the seven segment displays. Enter a multi-digit hex number using the keypad. It should appear, once character at a time on the 7-segments displays. Once you have entered the first operand, press the "+" key (BTN0). Now enter the second operand and press the "=" key (BTN1). The value of the sum of the operands should now appear on the display. You can now start a second calculation by once again entering the first operand using the keypad. The "clear" key (BTN3) should always set the result on the display to zero.

4.6 Now let's make some changes …

Modify your VHDL code to do the following:

(a) Modify the *leddec16* module to that it performs leading zero suppression (i.e. it does not display zeros ahead of the first non-zero digit). For example, the number "0023" should appear as "23" with the leading zeros suppressed. *Hint: You can turn off any digit in the display by never taking its anode to '0'. Modify the "anode <=" conditional assignment statement in leddec16 so that it only turns on a particular digit if it is non-zero or if there is non-zero information in the higher order digits of the data word.*

(b) Expand the calculator to also do subtraction operations. Use the button BTN2 (pin E18 on the *Nexys2* board) as the "–" key. *Hint: Modify the ENTER_ACC state to also test for the "–" key being depressed. Create a new signal which will record whether the "+" key or the "–" key was pushed. Then, when in state ENTER_OP and the "=" key is pressed, test your new signal to determine whether you should do an addition or a subtraction.*