Lab 4

Modification 1 - Removing the trailing zeros

We add a bit to the **data4** signal in the `leddec16.vhd` file (from 4 to 5):

```
SIGNAL data4 : STD_LOGIC_VECTOR (4 DOWNTO 0); -- binary value of
current digit
```

This bit will be used as a flag to determine if the current digit is a trailing zeros; and thus not to be displayed.

To determine if the current digit is a trailing zero, we add the following code to the `leddec16.vhd` file:

```
data4(0) <=
        '1' WHEN data(15 DOWNTO 12) = "0000" AND dig = "011" ELSE --
If the last digit is 0, turn off the last digit
        '1' WHEN data(15 DOWNTO 8) = "00000000" AND dig = "010" ELSE
-- If the last two digits are 0, turn off the last two digits
        '1' WHEN data(15 DOWNTO 4) = "000000000000" AND dig = "001"
ELSE -- If the last three digits are 0, turn off the last three
digits
        '0  -- Otherwise, turn on the last digit
```

Now because we added another bit we must modify the way we set **data4** to only use the first 4 bits, and then the last bit is set with the above code:

```
-- Select digit data to be displayed in this mpx period
data4(4 DOWNTO 1) <= data(3 DOWNTO 0) WHEN dig = "000" ELSE -- digit
0
            data(7 DOWNTO 4) WHEN dig = "001" ELSE -- digit 1
            data(11 DOWNTO 8) WHEN dig = "010" ELSE -- digit 2
            data(15 DOWNTO 12); -- digit 3
```

Now when we are displaying the digit of the trailing zero we know to set the value to `1111111` indicating that none of the segments should be lit.

```
-- Turn on segments corresponding to 4-bit data word
seg <= "0000001" WHEN data4 = "00000" ELSE -- 0
        --...
        "1111111";
```

Modification 2 - Implementing Subtraction

Now we have a new input port that we can implement in `hexcalc.vhd`:

```
--Type of Operation
TYPE operation IS (ADD, SUB);
SIGNAL op : operation;
```

When we can set the operation to ADD or SUB based on the button pressed:

```
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;
    op <= ADD;
ELSIF bt_sub = '1' THEN
    nx_state <= START_OP;
    op <= SUB;
ELSE nx_state <= ENTER_ACC;
END IF;
```

And when the equal button is pressed we can perform the operation:

```
IF op = ADD THEN
    nx_acc <= acc + operand;
ELSIF op = SUB THEN
    nx_acc <= acc - operand;
END IF;

nx_state <= SHOW_RESULT;
```

Lab 5

Step 1 - Generating a new wave form in the shape of a square

The modulation of the sound was due to the fact that the sound wave was a triangle wave. For this modification, we introduce and output a new wave in the shape of a square wave.

Identifying what to change

The `tone.vhd` file consisted of this line that generated the triangle wave:

```
WITH quad SELECT
data <= index WHEN "00", -- 1st quadrant
        16383 - index WHEN "01", -- 2nd quadrant
        0 - index WHEN "10", -- 3rd quadrant
        index - 16383 WHEN OTHERS; -- 4th quadrant
```

In this

- `quad` is the quadrant of the triangle wave (1st, 2nd, 3rd, or 4th)
- `data` is the output of the triangle wave
- `index` is the index of the triangle wave (the x-axis)

It is obvious that the triangle wave is generated by the `index` variable. The `index` variable is incremented by 1 every clock cycle. The `index` variable is reset to 0 when it reaches 32767. This means that the triangle wave will have a period of 32767 clock cycles.

Changing the Code

We first introduce a new port that will output the square wave in the port list:

```
data_square : OUT SIGNED (15 DOWNTO 0)
```

Now we can output the square wave to the `data_square` port:

```
--Transform into a square wave
WITH quad SELECT
data_square <= -16383 WHEN "00", -- 1st quadrant
        -16383 WHEN "01", -- 2nd quadrant
        16383 WHEN "10", -- 3rd quadrant
        16383 WHEN OTHERS; -- 4th quadrant
```

This will set the pitch to the maximum value when the triangle wave is in the 3rd and 4th quadrants, and set the pitch to the minimum value when the triangle wave is in the 1st and 2nd quadrants.

Now to keep things simple I won't write every single modification here. But the output of tone is passed into the wail entity which then need to output that to `siren.vhd`. Thus the ports are modified in each script to consist of this new `data_square` port.

After all that, in our `siren.vhd` architecture, this data is eventually passed into two signals `data_tri` and `data_square`.

```
w1 : wail
PORT MAP(
    lo_pitch => lo_tone, -- instantiate wailing
siren
    hi_pitch => hi_tone,
    wspeed => wail_speed,
    wclk => slo_clk,
    audio_clk => audio_clk,
    audio_data => data_tri,
    audio_data_square => data_square
);
```

Now we could just output the data_square signal and that would work, but we want to be able to change between the triangle and square wave using a button.

Step 2 - Adding a button to switch between the triangle and square wave

Identifying what to change

In the base script, the output for the left channel (dupilicated to the right) is set with the line:

```
data_L <= data;
```

So we just need to use a process to vary what data to take from (data_tri or data_square) based on the button state.

Changing the Code

First a new pin is registered to a port in the `siren.xdc` file:

```
# Change Shape Button
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 }
[get_ports { bt_change_shape }]; #IO_L4N_T0_D05_14 Sch=btnu
```

I'm not gonna show the port definitions anymore because it's repetitive but just know that the button is registered to the `bt_change_shape` port in the `siren.vhd` file as well.

Now we replace that line with a process:

```
-- this process selects between triangle and square waveforms
select_pr : PROCESS (bt_change_shape)
BEGIN
    IF bt_change_shape = '1' THEN
        data_L <= data_square;
    ELSE
        data_L <= data_tri;
    END IF;
END PROCESS;
```

Step 3 - Changing the wail speed with switches

Identifying what to change

We were easily able to identify what constant affected the wail speed.

Obviously, it was the `wail_speed` constant

Changing the Code

After changing `wail_speed` to a signal, we can then use a port of length 8 that is connected to the switches to set the value of `wail_speed`.

First we register the switches to a port in the `siren.xdc` file:

```
# Wailing Speed Switches
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports
{SW_SPD[0]}]
set_property -dict {PACKAGE_PIN L16 IOSTANDARD LVCMOS33} [get_ports
{SW_SPD[1]}]
set_property -dict {PACKAGE_PIN M13 IOSTANDARD LVCMOS33} [get_ports
{SW_SPD[2]}]
set_property -dict {PACKAGE_PIN R15 IOSTANDARD LVCMOS33} [get_ports
{SW_SPD[3]}]
set_property -dict {PACKAGE_PIN R17 IOSTANDARD LVCMOS33} [get_ports
{SW_SPD[4]}]
set_property -dict {PACKAGE_PIN T18 IOSTANDARD LVCMOS33} [get_ports
{SW_SPD[5]}]
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports
{SW_SPD[6]}]
set_property -dict {PACKAGE_PIN R13 IOSTANDARD LVCMOS33} [get_ports
{SW_SPD[7]}]
```

Now it's as simple as just setting the value of `wail_speed` to the value of the switches:

```
--Change Wail Speed With Switches
wail_speed <= SW_SPD;
```

Step 4 - Changing the shape and speed of the siren in the other channel

Instead of duplicating the output of the left channel to the right channel, we will go through and generate a whole other component for the right with different switches and another button mapped to change the shape of the wave.

Identifying what to change

```
data_R <= data_L; -- duplicate data on right channel
```

Instead of duplicating it like this we will generate a whole other component for the right channel.

Changing the code

Then we went through and basically dupilicated everything done for the left channel to use new vars and signals for the right channel.

```
-- this process selects between triangle and square waveforms for the
right
select_pr_R : PROCESS (bt_change_shape_R)
BEGIN
    IF bt_change_shape_R = '1' THEN
        data_R <= data_square_R;
    ELSE
        data_R <= data_tri_R;
    END IF;
END PROCESS;
```

Moore FSM

Lab 2

Successful Upload

Lab 1 vs 2

After the program was uploaded to the board, like in lab 1 the program starts at 0 and counts up in hexa decimal.

The difference is that the count is now a 16 bit number (opposed to 4 bits) and is displayed on multiple 7 segment displays.

Finite State Machine

We created this Moore Machine that would only output a '1' if the input given is `11100`

In choosing the sequence we could only choose a sequence that switches from '1' to '0' once. This is because as we add to the count, it's binary number will change. We know that binary numbers will cary when it gets big enough. For example, if we had the sequence `100` we know that the third digit will remain a 1 for a total of 4 clock cycles, then it will change to a 0 for > 2 cycles which would cause our fsm to output a **1**.

The following is the output of our test bench where we monitor the state and output based on our input:

Code

The following shows the modification to the **counter.vhd** file:

```
-- counter.vhd --
ARCHITECTURE Behavioral OF counter IS
    SIGNAL cnt : STD_LOGIC_VECTOR (38 DOWNTO 0); -- 39-bit counter
    SIGNAL X : std_logic; -- will store a bit from cnt
    SIGNAL Z : std_logic; -- Monitors the output of the FSM '1' will
reverse the counter
    SIGNAL reverse_clock : std_logic := '0'; -- will be used to
reverse the clock

    SIGNAL clk2 : std_logic; -- clock for the FSM

    type state_type is (A, B, C, D, E, F);
    signal PS, NS : state_type;
BEGIN
    X <= cnt(29);

    PROCESS (clk)
    BEGIN
        IF clk'event and clk = '1' THEN -- on rising edge of clock
            -- Initialize the Z output to '0'
            Z <= '0';

            -- State transition and output logic
            case PS is
                when A =>
                    if (X = '1') then
                        NS <= B;
                    else
                        NS <= A;
                    end if;
                when B =>
                    if (X = '1') then
                        NS <= C;
                    else
                        NS <= A;
                    end if;
                when C =>
                    if (X = '1') then
                        NS <= D;
                    else
                        NS <= A;
                    end if;
                when D =>
                    if (X = '0') then
                        NS <= E;
                    else
                        NS <= D;
                    end if;
                when E =>
                    if (X = '0') then
                        NS <= F;
                    else
                        NS <= B;
                    end if;
                when F =>
                    Z <= '1';
                    if (X = '0') then
                        NS <= A;
                    else
                        NS <= B;
                    end if;
            end case;
            PS <= NS;

            IF Z = '1' THEN -- if the FSM is in state 1
                reverse_clock <= NOT reverse_clock; -- reverse the
clock
            END IF;

            IF reverse_clock = '0' THEN -- if the clock is not
reversed
                cnt <= cnt + 1; -- increment the counter
            ELSE
                cnt <= cnt - 1; -- decrement the counter
            END IF;
        END IF;
    END PROCESS;

    count <= cnt (38 DOWNTO 23); -- 16 bits
    mpx <= cnt (19 DOWNTO 17); -- 3 bits
END Behavioral;
```