Zena's CS232 Project 4

For this project, we built a circuit that had a predefined small set of instructions which is implemented through a state machine. This created a programmable light display, on which the three red LEDs represent input bits, which is read into the state machine, from which it decides what the output should be, which is what is represented with the 8 green LEDs. A 1 in the byte (8 bits) means the light is on, while a 0 means it is off. There are 8 possible instructions, meaning we are using 3 bit input, and the light register (LR) holds 8 bits, which hold the numbers we will manipulate using the instructions that come from the 3 former bits.

Task 1 was to create a new project in Quartus. We refer to it as "lights". Incidentally, this project is the first time we were required to do all of the work in VHDL, without any block diagram files.

Task 2 was to create the initial file with the instructions in it, referred to as "lightrom." The program needs to have two things connected to it: "addr", which counts up from 0000 to 1111, and "data", which is the output of numbers we send to the next file to be interpreted as instructions. All this file is doing is sending certain outputs for "data" based on how far up "addr" has counted. A picture is below.

```
entity lightrom is

   port
   (
      addr     : in std_logic_vector  (3 downto 0);
      data : out std_logic_vector (2 downto 0)
   );

end entity;

architecture rtl of lightrom is
begin

data <=
      "000" when addr = "0000" else -- move 0s to LR  00000000
      "101" when addr = "0001" else -- bit invert LR  11111111
      "101" when addr = "0010" else -- bit invert LR  00000000
      "101" when addr = "0011" else -- bit invert LR  11111111
      "001" when addr = "0100" else -- shift LR right 01111111
      "001" when addr = "0101" else -- shift LR right 00111111
      "111" when addr = "0110" else -- rotate LR left 01111110
      "111" when addr = "0111" else -- rotate LR left 11111100
      "111" when addr = "1000" else -- rotate LR left 11111001
      "111" when addr = "1001" else -- rotate LR left 11110011
      "010" when addr = "1010" else -- shift LR left  11100110
      "010" when addr = "1011" else -- shift LR left  11001100
      "011" when addr = "1100" else -- add 1 to LR    11001101
      "100" when addr = "1101" else -- sub 1 from LR  11001100
      "101" when addr = "1110" else -- bit invert LR  00110011
      "011";                        -- add 1 to LR    00110100
```

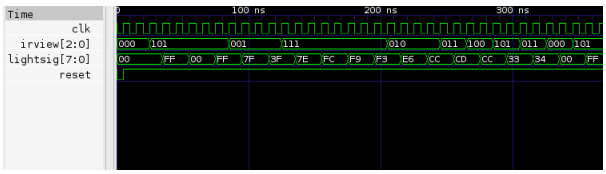
Task 3 was to make "lights", the VHD file that would control what the lights do based on what instructions are received. "IRView" is the vector output that controls the 3 red LEDs, "lights" is the output for the 8 green lights, and there is also an internal clock ("clk") as well as a reset button. PC takes in the "addr" bits, which are counting up, IR is where the instructions are stored, and LR is the output for the green LEDs. For the process, we reset all of these if reset is pressed, and if not, the clock begins ticking. As the time moves up and the cases are cycled through, LR's value is decided by instructions, given through when statements. For example, if IR is 000, then LR has 00000000 loaded into it. At the end of the file, we also output IRview as the red lights and lights as the green lights. My code is shown below:

```
 begin
    if reset = '0' then --reset the simulation if the reset button is pressed
       PC <= "0000";--counting up
       IR <= "000";--instructions here
       LR <= "00000000"; --output into green LEDs here
       state <= sFetch;
    elsif (rising_edge(slowclock)) then
       case state is
          when sFetch=>
             IR <= ROMvalue;
             PC <= PC + 1;
             state <= sExecute;
          when sExecute=>
             case IR is
                when "000"=>
                   LR <= "00000000";
                   state <= sFetch;
                when "001"=>
                   LR <= LR srl 1; --rotate right
                   state <= sFetch;
                when "010"=>
                   LR <= LR sll 1; --rotate left
                   state <= sFetch;
                when "011"=>
                   LR <= LR +"00000001"; --add 1
                   state <= sFetch;
```

Task 4 was to make sure we were importing the lightrom into this file so the corresponding inputs and outputs could connect. We just had to add the lightrom file as a component right after the start of the architecture section, then create an instance of it before the main process begun and after the architecture statements.

Task 5 was to test the circuit in a gtkwave simulation. As we can see, the clock moves forward at certain intervals, and the lights output is changing based on what the irview is. There is a bit of a lag due to the clock timing, but one can see, for example, that lightsig is 00 at first, then

when irview is 101 (the instruction to invert the bits), it becomes FF. In other words, the bits go from being all 0s to being all 1s, represented in hex in this case.



For Task 6, we had to set the code up so that it would work with a physical circuit. This was just a matter of setting up the pins, then changing the clock, for our purposes, to go 32 million times slower than the actual internal clock so we could see our results. A video of me testing my circuit is below. As we can see, the board cycles through all of the instructions we gave it in the lightrom, represented by the red LEDs, and the output is represented by the green LEDs.

http://sendvid.com/wv6yrzb7

In Task 7, we had to create two more VHD files with sets of instructions like in lightrom. The ones I made are shown below.

```
"000" when addr = "0000" else -- move 0s to LR  00000000
"110" when addr = "0001" else -- rotate LR right 00000000
"101" when addr = "0010" else -- bit invert LR 11111111
"100" when addr = "0011" else -- sub 1 from LR 11111110
"010" when addr = "0100" else -- shift LR left 11111100
"101" when addr = "0101" else -- bit invert LR 00000011
"010" when addr = "0110" else -- shift LR left 00000110
"000" when addr = "0111" else -- load 0s to LR 00000000
"111" when addr = "1000" else -- rotate LR left 00000000
"011" when addr = "1001" else -- add 1 to LR 00000001
"101" when addr = "1010" else -- bit invert LR 11111110
"110" when addr = "1011" else -- rotate LR right 01111111
"011" when addr = "1100" else -- add 1 to LR 10000000
"010" when addr = "1101" else -- shift LR left 00000000
"001" when addr = "1110" else -- shift LR right 00000000
"101";                        -- bit invert LR 11111111
```

```
"000" when addr = "0000" else -- move 0s to LR 00000000
"001" when addr = "0001" else -- shift LR right 00000000
"010" when addr = "0010" else -- shift LR left 00000000
"011" when addr = "0011" else -- add 1 to LR 00000001
"100" when addr = "0100" else -- subtract 1 from LR 00000000
"101" when addr = "0101" else -- invert bits 11111111
"110" when addr = "0110" else -- rotate LR right 11111111
"111" when addr = "0111" else -- rotate LR left 11111111
"100" when addr = "1000" else -- subtract 1 from LR 11111110
"001" when addr = "1001" else -- shift LR right 01111111
"001" when addr = "1010" else -- shift LR right 00111111
"011" when addr = "1011" else -- add 1 to LR 01000000
"011" when addr = "1100" else -- add 1 from LR 01000001
"101" when addr = "1101" else -- invert bits 10111110
"110" when addr = "1110" else -- rotate LR right 01011111
"101";                        -- invert bits 10100000
```

I did the first extension, too. This was to make the a longer program, which meant that "addr" would need to have (at least) another bit. So I changed the kind of vector addr was to a 4 down 0 one instead of 3 downto 0 like it was before. Everywhere that needed a change in amount of bits was changed from 4 to 5. Then, the only other adjustment was giving more instructions -- 32 in total -- to the machine to test if it worked. My test program is shown below:

```
"000" when addr = "00000" else -- move 0s to LR 00000000
"001" when addr = "00001" else -- shift LR right 00000000
"010" when addr = "00010" else -- shift LR left 00000000
"011" when addr = "00011" else -- add 1 to LR 00000001
"100" when addr = "00100" else -- subtract 1 from LR 00000000
"101" when addr = "00101" else -- invert bits 11111111
"110" when addr = "00110" else -- rotate LR right 11111111
"111" when addr = "00111" else -- rotate LR left 11111111
"100" when addr = "01000" else -- subtract 1 from LR 11111110
"001" when addr = "01001" else -- shift LR right 01111111
"001" when addr = "01010" else -- shift LR right 00111111
"011" when addr = "01011" else -- add 1 to LR 01000000
"011" when addr = "01100" else -- add 1 from LR 01000001
"101" when addr = "01101" else -- invert bits 10111110
"110" when addr = "01110" else -- rotate LR right 01011111
"101" when addr = "01111" else -- invert bits 10100000
"011" when addr = "10000" else -- add 1 to LR 10100001
"001" when addr = "10001" else -- shift LR right 01010000
"010" when addr = "10010" else -- shift LR left 10100000
"011" when addr = "10011" else -- add 1 to LR 10100001
"100" when addr = "10100" else -- subtract 1 from LR 10100000
"101" when addr = "10101" else -- invert bits 01011111
"110" when addr = "10110" else -- rotate LR right 10101111
"111" when addr = "10111" else -- rotate LR left 01011111
"100" when addr = "11000" else -- subtract 1 from LR 01011110
"001" when addr = "11001" else -- shift LR right 00101111
"001" when addr = "11010" else -- shift LR right 00111111
"011" when addr = "11011" else -- add 1 to LR 01000000
"011" when addr = "11100" else -- add 1 from LR 01000001
"101" when addr = "11101" else -- invert bits 10111110
"110" when addr = "11110" else -- rotate LR right 01011111
```

Next, I did extension 2, which was to add instructions to the system. One could have made more instructions directly by giving the data (red light) 4 bits instead of 3, for example, but I decided to indirectly make branch directions by having certain inputs equal different things if a certain condition was true. If 101 was input, for instance, the branch1 variable would be true, and it would mean that the next time the execute state was entered, it would follow different rules. An input of 111 in the first branch would take us into the second branch, too. Some instructions could be used to exit each branch, as well. Looking back, I could have implemented this using different kinds of Execute states, as well, but this worked as it should have anyways, as shown below:

http://sendvid.com/ibqkzqxu

This project taught me a lot about VHDL, more so than any project so far, because we did everything in it without using any block diagram files like the past times. I also gained a lot of experience using Moore state machines. This project was a very intuitive one that I had little trouble following.