Zena's CS232 Project 5

For this project, we made a more complex version of the light display from Project 4. While Project 4 took only 3 bits of instructions, this takes 10. The first two bits specify the kind of instruction: 00 for move, 01 for operation, 10 for branch, and 11 for conditional branch. Within each kind of instruction, the rest of the bits hold different values that also will depend on what numbers they are assigned in the instruction register. The provided chart is shown below:

Opcode	Format	Description
00	[C1 C0] [Dest1 Dest0] [Src1 Src0] [Val3 Val2 Val1 Val0]	Move from SRC to DEST
	Dest10: 00 = ACC, 01 = LR, 10 = ACC low 4 bits, 11 = ACC high 4 bits	
	Src10: 00 = ACC, 01 = LR, 10 = IR low 4 bits sign extended, 11 = all 1s	
01	[C1 C0] [Op2 Op1 Op0] [Src1 Src0] [Dest0] [Val1 Val0]	Binary operator DEST = DEST op SRC
	Op210: 000 = add, 001 = sub, 010 = shift left, 011 = shift right maintain sign bit	
	Op210: 100 = xor, 101 = and, 110 = rotate left, 111 = rotate right	
	Src10: 00 = ACC, 01 = LR, 10 = IR low 2 bits sign extended, 11 = all 1s	
	Dest0: 0 = ACC, 1 = LR	
10	[C1 C0] [U3 U2 U1 U0] [Addr3 Addr2 Addr1 Addr0]	Branch to ADDR
11	[C1 C0] [Src0] [U2 U1 U0] [Addr3 Addr2 Addr1 Addr0]	Branch to ADDR if SRC is 0
	Src0: 0 = ACC, 1 = LR	

With these instructions, we were able to implement loops, as well.

The first task was to create a new vhdl project in quarts called pld2--mine, however, is called Project5. After that, task 2 called for a ROM (read-only memory) called pldrom, which is where I stored the instructions for Project5 to interpret. Pldrom has two signals: addr (address; the position we are at when counting up) and data, which is where the instructions are stored. I mapped those into Project5 afterwards.

Task 3 was to create the skeleton of the top level (Project5/pld2) file. Important to note is that the file now has not one, but rather two, execute states, so that the first execute state can be used to assign sources and the second execute state can assign the destinations. The first is more about storing values while the second execute state is about manipulating those stored values. Notice that since the branch instructions don't need destinations, they also don't need to do anything in sExecute2. The flow of the execution is always going from sFetch to sExecute1 to sExecute2, as one can see in the picture below.

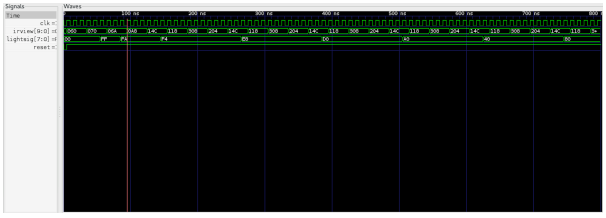
```
            when sFetch=>
               IR <= ROMvalue;
               PC <= PC + 1;
               state <= sExecute1;
            when sExecute1=>
               state<=sExecute2;
               case IR(9 downto 8) is
                  when "00"=> --when MOVE
                     case IR(5 downto 4) is
                  when "01"=> --when OP
                     case IR(4 downto 3) is
                  when "10"=> --when Branch
                     PC<= unsigned(IR(3 downto 0));
                  when "11"=> --when Branch if 0
                     case IR(7) is
                     if SRC="00000000"
                        then PC<=unsigned(IR(3 downto 0));
                  when others=>
                     null;
                  end case;
            when sExecute2=>
               state<=sFetch;
               case IR(9 downto 8) is
                  when "00"=> --Move
                     case IR(7 downto 6) is --when destination is...
                  when "01"=> --Binary OP
                     case IR(2) is --check DEST
                  when "10"=> --branch(nothing)
                     null;
                  when "11"=> --branch(nothing)
                     null;
                  when others=>
                     null;
               end case;
```

The black lines are the places that correspond to checking and carrying out the corresponding instructions based on the additional info rows in the table. Under the first line, for example, the source is checked, and if it is 00, the source will be assigned ACC, and so on.

Task 4 was to format the first execute state. This meant that I had to assign the corresponding bits of IR that would indicate where to look for what kind of info (source, destination, etc.) to the things the instructions dictated for them. For instance, when the case of IR (9 down 8) is 00, we are in the move case, and within the move case, the source is found in IR(5 down 4), so we look within that to decide what to assign to where.

In task 5, I had to do the same thing with the second execute state--tell it where to look for what and what to do with what it found. In the move and binary operation cases, it checked the destinations and assigned them things based on the source values. The operations were also listed here, such as xor, adding, shifting, etc. The branching instructions don't need any more information so they don't do anything here, which is why I used "null" as a way to pass through those instructions and do nothing.

For task 6, we just needed to test if our file worked in gtkwave. The picture of my simulation is below. The instruction register's hex value equivalents of their 10-digit binary is shown in the second row, and the 8-bit converted to hex is under it, representing which lights are turned on. The running of programs on the circuit board after was much easier to interpret, however.



Task 7 was to write such a program and run it on the circuit board. The program was supposed to load 16 into the light register, make it count down to 0, then make it flash from all 1's to all 0's 8 times. A picture of my instructions is below.

```
data <=
    "0001100000" when addr = "0000" else -- move 0000 from IR to ACC low 4 bits 00000000/00000000
    "0011100001" when addr = "0001" else -- move 0001 from IR to ACC high 4 bits 00000000/00010000
    "0001000000" when addr = "0010" else -- move ACC into LR 00010000/00010000
    "0100110101" when addr = "0011" else -- subtract 1 from LR (00001111 first time)/00010000
    "1110000110" when addr = "0100" else -- branch to addr 6 if LR is 0
    "1000000011" when addr = "0101" else -- branch to addr 3
    "0010101000" when addr = "0110" else -- load 1000 into ACC low 4 bits
    "0011100000" when addr = "0111" else -- load 0000 into ACC high 4 bits
    "0001110000" when addr = "1000" else -- load into LR all 1's 11111111/00001001
    "0110011100" when addr = "1001" else -- xor LR with all 1's
    "0100011000" when addr = "1010" else -- add -1 to ACC
    "1100000000" when addr = "1011" else -- branch to addr 0 if ACC=0
    "1000000111" when addr = "1100" else -- branch to addr=7
    "1100110011" when addr = "1101" else -- garbage
    "1111111111" when addr = "1110" else -- garbage
    "1111111111";                        -- garbage
```

Task 8 was to make two more programs. For my first, I made a program that displays all 1's, then all 0's, then changes between 10101010 and 01010101 7 times. My instructions and a video of the program are shown below:

```
data <=
    "0000110000" when addr = "0000" else -- load all 1's into the ACC 00000000/11111111
    "0001000000" when addr = "0001" else -- load ACC into LR 11111111/11111111
    "0110011100" when addr = "0010" else -- xor LR with all 1's 00000000/11111111
    "0010101010" when addr = "0011" else -- load 1010 into ACC low 4 bits 00000000/11111010
    "0011101010" when addr = "0100" else -- load 1010 into ACC high 4 bits 00000000/10101010
    "0001000000" when addr = "0101" else -- load ACC into LR 10101010/10101010
    "0011101111" when addr = "0110" else -- load 1111 into ACC high 4 10101010/11111010
    "0110011100" when addr = "0111" else -- xor LR with all 1's 01010101/11111010
    "0100010001" when addr = "1000" else -- add 1 to ACC 01010101/11111011
    "1100000000" when addr = "1001" else -- if ACC=0, branch to 0000
    "1000000111" when addr = "1010" else -- branch to 0111
    "1000000111" when addr = "1011" else -- garbage
    "1100110011" when addr = "1100" else -- garbage
    "1111111111" when addr = "1101" else -- garbage
    "1111111111" when addr = "1110" else -- garbage
    "1111111111";                        -- garbage
```

21653483.480p.mp4

My other program counts up to 4, shifts left 5 times, and leaves that bit sitting there until after the program returns to start and starts shifting after reaching 4 again. My code for the program is shown below.

```
  data <=
   "0000100100" when addr = "0000" else -- load 4 into ACC 00000000/00000011
   "0100010101" when addr = "0001" else -- add 1 to LR 00000001/00000011
   "0100110001" when addr = "0010" else -- subtract 1 from ACC 00000001/00000010
   "1100000101" when addr = "0011" else -- branch to 0101 if ACC=0
   "1000000001" when addr = "0100" else -- branch to 0001
   "0010100101" when addr = "0101" else -- load 5 into ACC low 4 bits 00000001/00000101
   "0101001100" when addr = "0110" else -- shift LR left 1
   "0100110001" when addr = "0111" else -- subtract 1 from ACC
   "1100000000" when addr = "1000" else -- branch to 0 if AC=0
   "1000000110" when addr = "1001" else -- branch to 0110
   "1111111111" when addr = "1010" else -- garbage
   "1111111111" when addr = "1011" else -- garbage
   "1111111111" when addr = "1100" else -- garbage
   "1111111111" when addr = "1101" else -- garbage
   "0000000000" when addr = "1110" else -- garbage
   "1111111111";                        -- garbage
```

I did extension 4, which was to make a freeze/pause button. Upon being pressed and released, the program will stop moving forward. After being pressed again, it will resume as usual from where it left off. If reset is hit, it will be un-paused, as well. To add the ability to pause, I made an input at the beginning of Project5 for the pause button, called "pause", which keeps track of the state of the button, as well as an internal value called "paused," which keeps track of whether or not the counter is paused. I added the pause button into the clock process after the main one, then added in various lines of code to account for the pause button. On reset, for example, paused equals 1, which represents not being paused, while 0 represents being paused. The elif case checks what the state of paused is, and always returns the opposite, meaning that pressing it will either pause or unpause it depending on the current state. When the clock counts up, we also have to make sure that when the paused case is true (so 0), the counter will not progress any further, while if it is off (so 1), it will proceed as usual. My code and video are below.

```
process(clk, reset,pause)
begin
   if reset = '0' then
      counter <= "00000000000000000000000";
      paused<='1'; --not paused
   elsif pause = '0' then --if pause is pressed
      case paused is
         when '0'=> --paused to not paused
            paused<='1';
         when others=> --not paused to paused
            paused<='0';
      end case;
elsif (rising_edge(clk)) then
   case paused is
      when '0'=> --when it is paused
         counter<= counter + 0;
      when others => --when it is not paused
         counter <= counter + 1;
   end case;
end if;
```

pausevid.mp4

This project taught me a lot about how to check and categorize bits from the instruction register and how to control the data flow through the states. I also learned about the null function and was able to solve problems involving making the LR follow certain patterns while before this was very difficult for me to comprehend.

Thanks to: Professor Maxwell, Professor Taylor, Melody, Arthur