Zack Fravel

010646947

L001 4:10 - 5:55 PM

Lab 2 - ALU

2/15/16

Introduction

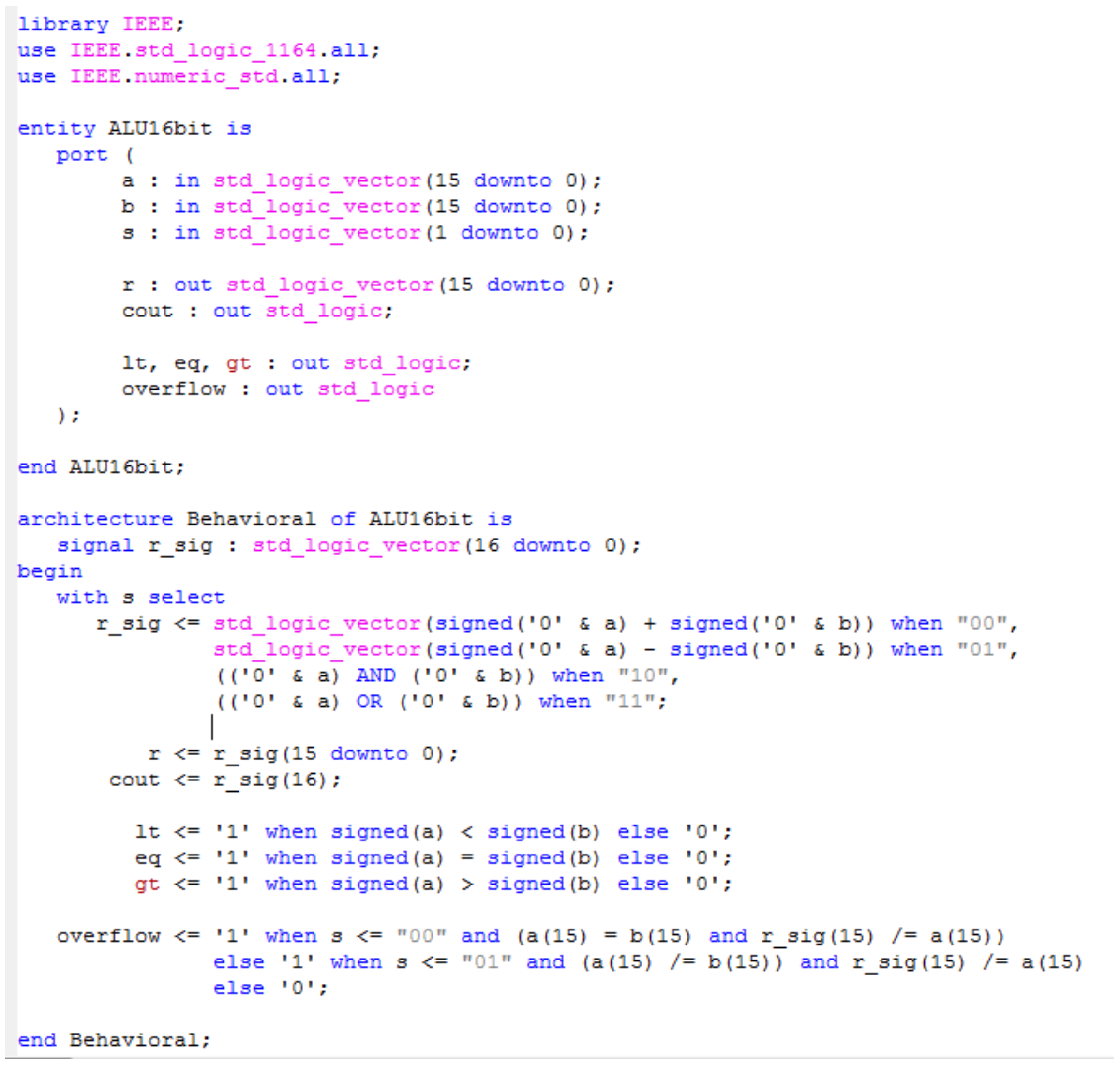
The objective of this lab assignment was to write VHDL code to complete an already laid out entity for a 16 bit ALU, or Arithmetic Logic Unit. The ALU is a unit that takes in two 16 bit pieces of data and has the ability to add, subtract, and perform AND and OR logic based on a selector input, much like a mux. The ALU also is supposed to be able to account for overflow with signed addition, as well as put through a carry-out with addition. Along with that, the ALU can tell whether the two inputs are less than, equal to, or greater than each other. Along with the VHDL, we were also assigned to create a test bench for simulation testing.

Approach

As previously stated, the ALU is designed for 16 bit use. Therefore, the inputs a, b and the output, r, are all 16 bit std logic vectors. The other input we have to worry about is "s," for selector, which is a 2 bit std logic vector. Each of s's inputs: "00," "01," "10," and "11" represent the 4 different operations the ALU is designed to perform. The other five outputs are all std logic, single bits. These outputs are cout (carry out), lt (less than), eq (equal to), gt (greater than), and overflow.

Since addition and subtraction are two operations our ALU is designed to perform, we have to take into account a few things. First of all, we need to decide what kind of arithmetic we want to use. For our purposes with the ALU, we want to use the ieee.numeric.all library to allow for signed addition and subtraction using (+) and (-) respectively. The numeric library also enables us to use ">," "<," and "=" logic for comparing the two inputs. We always want to take into account for the carry out value and extend the result by a single bit to account for addition of larger numbers. Another thing to take into account with signed addition of numbers is overflow, which is when the addition of two numbers of the same sign or subtraction of two numbers with different signs yields an unexpected result. For example, two negative numbers being added together and netting a positive result through the carry values.

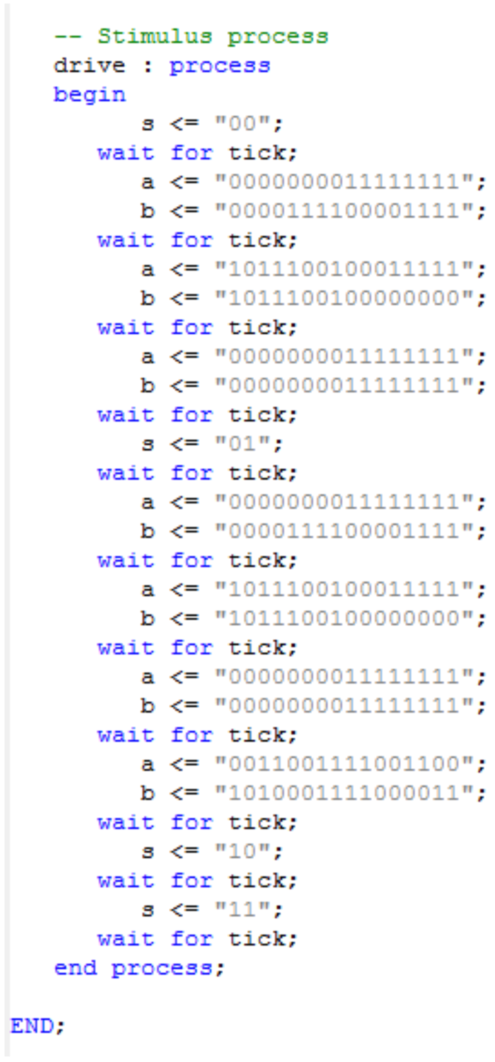
That takes care of the first two operations. The final two were much easier to implement, just by using standard AND and OR logic along with the "with-s-select" in the code. Below is the VHDL code for the 16 bit ALU. The 17 bit signal r_sig was used as a temporary output value so I was able to split the carry out and from the rest of the result as well as use it for the overflow logic.



```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity ALU16bit is
   port (
        a : in std_logic_vector(15 downto 0);
        b : in std_logic_vector(15 downto 0);
        s : in std_logic_vector(1 downto 0);

        r : out std_logic_vector(15 downto 0);
        cout : out std_logic;

        lt, eq, gt : out std_logic;
        overflow : out std_logic
   );

end ALU16bit;

architecture Behavioral of ALU16bit is
   signal r_sig : std_logic_vector(16 downto 0);
begin
   with s select
      r_sig <= std_logic_vector(signed('0' & a) + signed('0' & b)) when "00",
               std_logic_vector(signed('0' & a) - signed('0' & b)) when "01",
               (('0' & a) AND ('0' & b)) when "10",
               (('0' & a) OR ('0' & b)) when "11";

          r <= r_sig(15 downto 0);
       cout <= r_sig(16);

         lt <= '1' when signed(a) < signed(b) else '0';
         eq <= '1' when signed(a) = signed(b) else '0';
         gt <= '1' when signed(a) > signed(b) else '0';

   overflow <= '1' when s <= "00" and (a(15) = b(15) and r_sig(15) /= a(15))
               else '1' when s <= "01" and (a(15) /= b(15)) and r_sig(15) /= a(15)
               else '0';

end Behavioral;
```

Experimentation

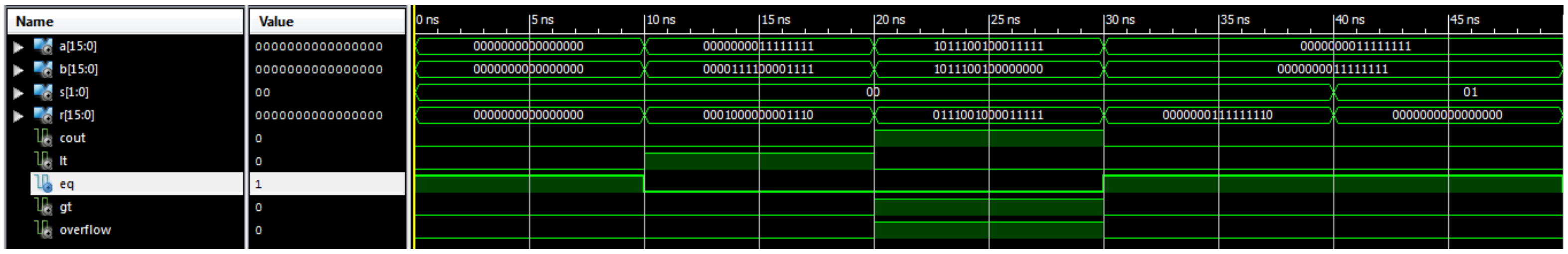
Once I had the VHDL written and compiling correctly, it was time to move on to the test bench to start simulating. I wanted to create a test bench to run a simulation that would show all the functionality of the ALU in a short amount of time. I initialized the selector at “00” to start with addition. I did this before assigning values to a and b to show that the computer is correctly adding whenever I introduce the inputs. From here I show two more adds, then move the selector to “01” and show 4 examples of subtraction. With both of these I included one example of overflow to show its functionality. After that, I showed the AND and OR logic for a set of inputs. Below is the test bench for the ALU.



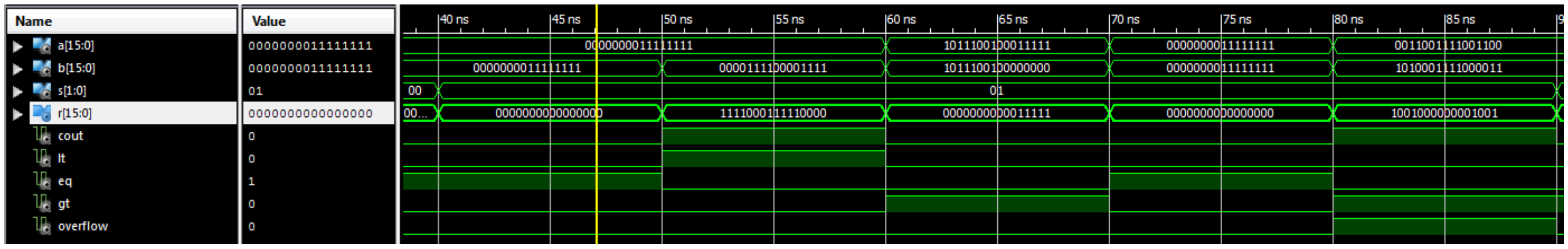
```
  -- Stimulus process
  drive : process
  begin
        s <= "00";
     wait for tick;
        a <= "0000000011111111";
        b <= "0000111100001111";
     wait for tick;
        a <= "1011100100011111";
        b <= "1011100100000000";
     wait for tick;
        a <= "0000000011111111";
        b <= "0000000011111111";
     wait for tick;
        s <= "01";
     wait for tick;
        a <= "0000000011111111";
        b <= "0000111100001111";
     wait for tick;
        a <= "1011100100011111";
        b <= "1011100100000000";
     wait for tick;
        a <= "0000000011111111";
        b <= "0000000011111111";
     wait for tick;
        a <= "0011001111001100";
        b <= "1010001111000011";
     wait for tick;
        s <= "10";
     wait for tick;
        s <= "11";
     wait for tick;
  end process;

END;
```

Results

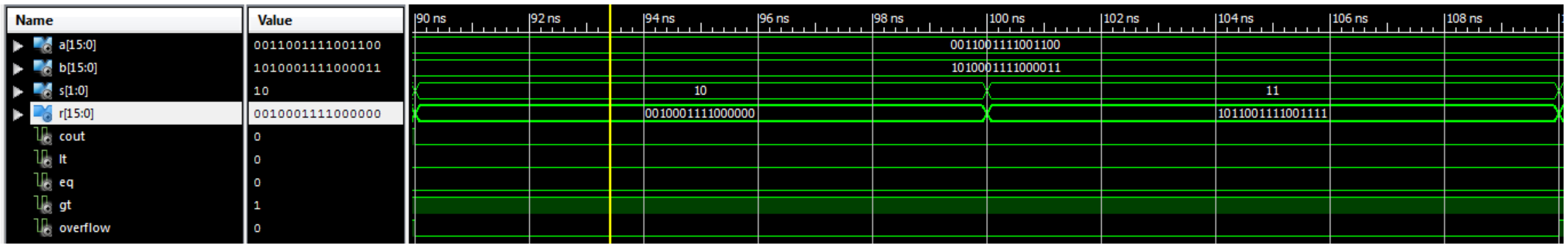
Below are the first 50 ns of results from the test bench.



Both a and b are initially set to “0000000000000000” with s set to “00” to represent addition. Its fairly easy to follow the diagram and see the resulting output waveforms change each time the inputs change. In this section, there are 4 examples of addition showing an example for lt, gt, and eq as well as one with overflow. Below are the results for subtraction.



Here I show another 4 examples, one for each of the different cases the ALU can handle. The final example shows overflow with subtraction, which occurs when two numbers of different sign are subtracted and the result is not correct. The final waveform shows the AND and OR logic, represented by “10” and “11” on the selector, s, respectively. The output here is just the combinational logic output of the two inputs.



Conclusion

As far as I know, the 16 bit ALU I designed works for all cases needed for our purposes. There are no known issues with my design, the most difficult part was figuring out the correct logic for calculating overflow with signed addition. This lab was also good in helping me figure out better ways of organizing my test bench files to generate waveform diagrams that are easier to follow and show more functionality in less space. This also gave me more experience with working with signed addition/subtraction and the things you have to take into account (e.g. concatenating a ‘0’ to each input when performing operations to account for the cout).