Zack Fravel

010646947

L001 - Mon. 4:10-5:55 PM

Lab 3 - Register

2/22/16

Introduction

The objective of lab 3 was to design and implement a register file with 16 registers, each being 16 bits wide. This register file can basically be thought of as an array of 16 different std logic vectors. The register is going to allow us to store and use multiple 16 bit numbers for use with our ALU. We were given a test bench to test our register file, it was the task of the lab to write the VHDL and parse through the simulation and analyze what is actually going on.

Approach

The register has a lot of different pieces to it. The way to think about it an paint a clear picture is imagining an array with 16 different spaces each allowing a 16 bit number. The inputs and outputs, a, b and c, are split into data and address inputs. The data inputs are 16 bits wide and represent the actual numbers being used by the machine; the address inputs are 4 bits wide and are used by the machine to differentiate between the different values stored in the 16 spaces in the register. The outputs b and c are then passed through to the next component. We also have load, clear, and clk inputs. The load input is an enabler, when asserted the load signal tells the register to pass through the a_data to the a_addr on the rising edge of the clock. The address inputs for b_addr and c_addr however are not associated with the clock and therefore the updated output is always being put through regardless of the clock. The clear and clk work as they have on previous projects, the clk is positive edge triggered and the clear is asynchronous with negative logic. In the end what we have is a register whose outputs are always available to the ALU, however the write-in values are determined by the rising edge of the clock.

Experimentation

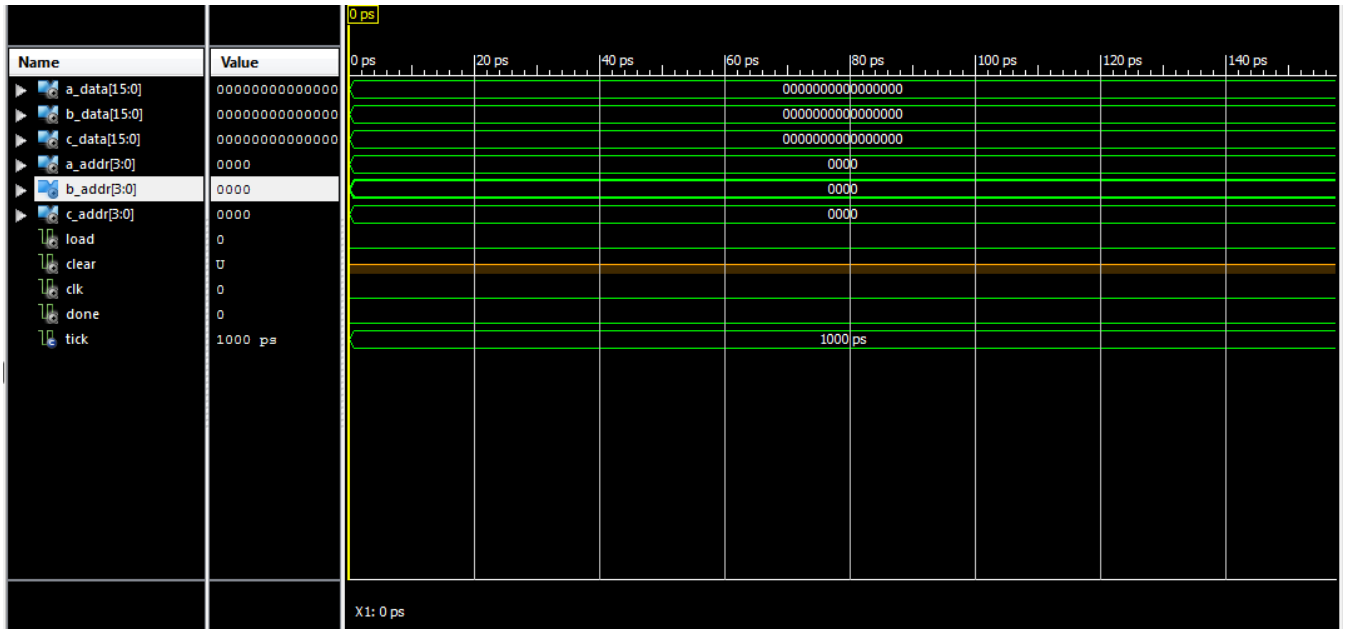
The main work that was needed to be done in the lab was the actual VHDL code of the

register file, since our test bench was given to us. We were given an entity outline and the high level logical description of what the register was supposed to do. In order to create the "array of std logic vectors" in our design, we want to create an actual array type called regfile in our VHDL that is comprised of 16 bit std logic vectors. This is going to allow us to differentiate between the different spaces in our register by converting the address input signals into integers and passing that integer into the "registers" regfile signal. Since a lot of the write-in functionality is determined by multiple inputs, the implementation began with writing a process involving the clk, load, and clear signals. We also want to declare an integer, reg, that allows us to easily clear the whole register using a for loop. We want to run this loop always when clear = '0.' The other part of the write-in process relies on the rising edge of the clock and load = '1,' with those two conditions met the register converts the a_addr into an integer and passes a_data into the register associated with that address. Registers 0 and 1 are always assigned the values x"0000" and x"0001" respectively (these are hex representations of 16 bit numbers). Our outputs, b_data and c_data are always being asserted, regardless of the clock, with the same integer address conversion.

```
34  entity reg_file is
35     port (
36          a_data : in std_logic_vector(15 downto 0); -- input data port
37          a_addr : in std_logic_vector(3 downto 0); -- register select for input a
38          load : in std_logic; -- load enable
39
40          b_data : out std_logic_vector(15 downto 0); -- first output data port
41          b_addr : in std_logic_vector(3 downto 0); -- register select for output b
42
43          c_data : out std_logic_vector(15 downto 0); -- second output data port
44          c_addr : in std_logic_vector(3 downto 0); -- register select for output c
45
46          clear : in std_logic; -- asynchronous reset, negative logic
47          clk : in std_logic -- positive edge triggered clock
48      );
49  end entity reg_file;
50
51
52
53  architecture Behavioral of reg_file is
54
55     type regfile is array(0 to 15) of std_logic_vector(15 downto 0);
56     signal registers : regfile;
57
58  begin
59     write : process(load, clear, clk) is
60     variable reg : integer;
61     begin
62        if(clear = '0') then
63           for reg in 0 to 15 loop
64              registers(reg) <= x"0000";
65           end loop;
66        elsif(rising_edge(clk) and load = '1') then
67           registers(conv_integer(a_addr)) <= a_data;
68        end if;
69
70        registers(0) <= x"0000";
71        registers(1) <= x"0001";
72
73     end process write;
74
75     b_data <= registers(conv_integer(b_addr));
76     c_data <= registers(conv_integer(c_addr));
77
78  end Behavioral;
79
```

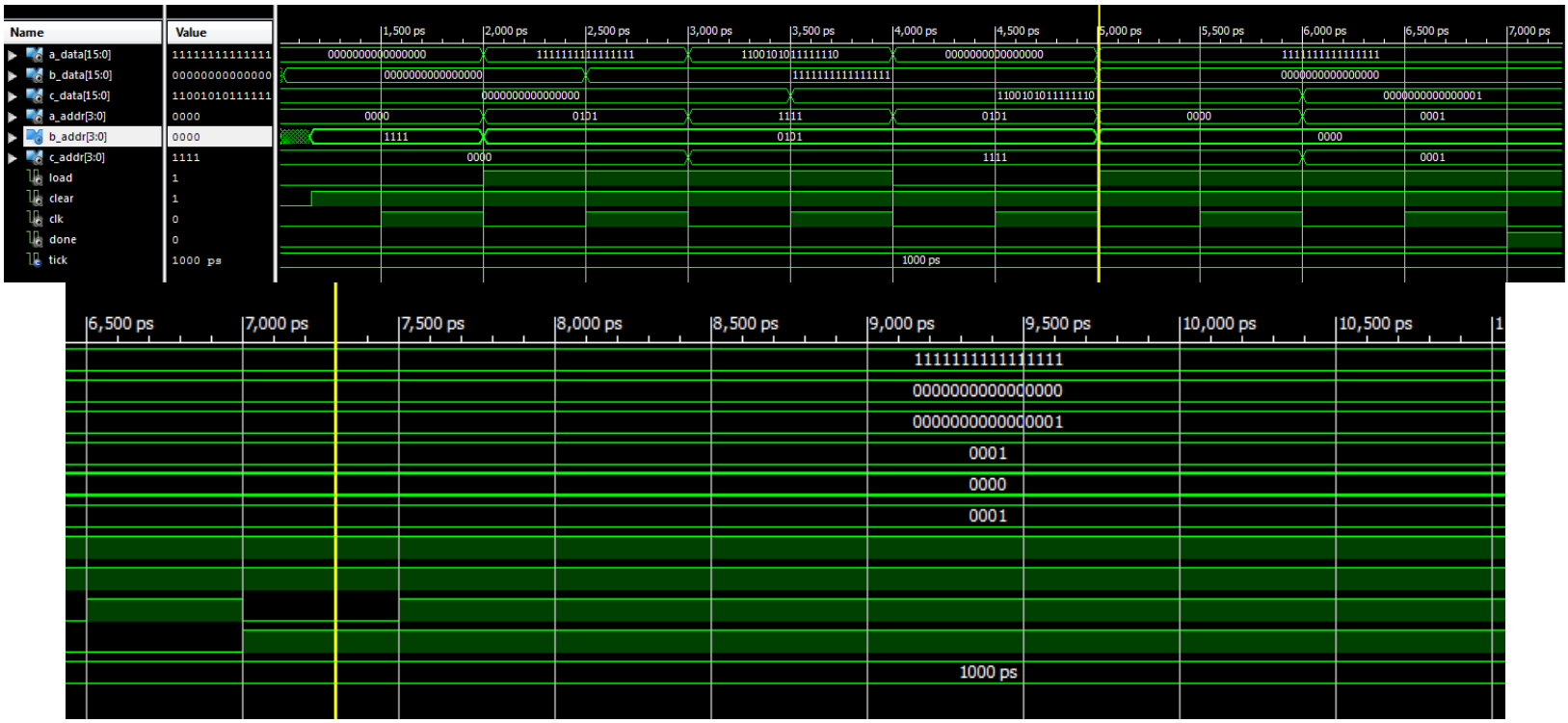
Results

Since we were given a test bench for our register file, its the assignment of the lab to analyze whats going on with the vector waveform file. The test bench would output a “PASS” or “FAIL” signal on all its tests through the Xilinx program during a simulation, so you know if the test bench is fully being implemented and functioning correctly. The test bench begins with declaring the clock tick signal to be 1 ns. The simulation begins with load being asserted to ‘0’ so no data is allowed to be passed through. Below is the beginning of the waveform simulation.



Here it can be seen that we have an undefined clear signal, this is because in the code for the test bench there just actually hasn't been anything assigned to the clear signal. It isn't until the first process where the clock cycle begins and values begin to change. The next step in the simulation the clock is asserted, however nothing happens because the load and clear signal are both set at ‘0.’ After that, we run through every possible value for b_addr, which every value has already been set to x“0000” except of course for register 1, which is always set to x”0001.” It can be seen that the only time the value of b_data change is the when b_addr = “0001.”

The next part of the test bench is when things really start happening as data is being written in. The test bench waits until the end of the clear phase and asserts the signal of clear and load to ‘1,’ allowing new data to write into the register file. We then assign a_addr to x”5” or “0101” and the a_data value to x”FFFF” which would just be “1111111111111111” in binary and set b_addr equal to x”5” as well. a_data also writes in x”CAFE” into the “F” address, which would be address 16 or “1111.” We then make load = ‘0’ and make a change to register 5 back to x”0000”however this change isn't asserted. After changing load back to ‘1,’ we attempt to write-in x”FFFF” into registers 0 and 1 with a_data. However, both of those registers are hard coded and it can be shown that b_data and c_data don’t change from x”0000” to x”0001” respectively.



Conclusion

As illustrated by the explanation of the design and simulations, the register file works exactly how it was outlined in the design of the lab. There are no known errors. The only issue I had in getting my code to work was making sure I was using the correct syntax in the ieee.stdlogic.arith.all with the conv_integer function. Other than that, I had an error that where my registers 0 and 1 weren't assigned values correctly, however it was a simple typo. After fixing those issues, the register file ran just as it should. This lab did a really good job at showing the levels of abstraction that are in place in Xilinx to help implement more complex ideas into a circuit fairly simply. For example using for loops and different signals to make differentiating different pieces of data more readable to a person.