Embedded Systems Spring 2019
Lab 2

Jonathan Colella

March 7, 2019

Purpose

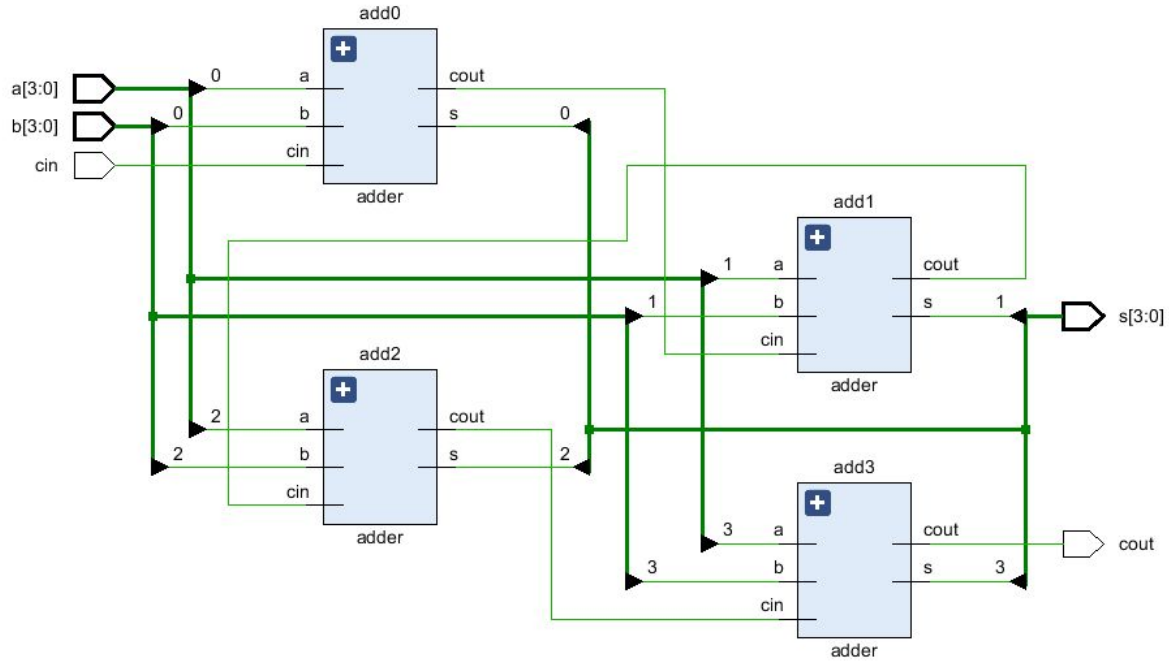
This lab centers around building an ALU with 16 functions. It has a 4 bit wide datapath, which can easily be expanded to an arbitrary number of bits through modification of the VHDL code. The alu was written into a test environment that interfaces it with the ZyBo board. The board is able to accept 4 bit wide data inputs and store them in 3 different registers, and trigger saving of data from the switches when debounced inputs from the ZyBo's buttons are pressed.

1) Ripple Carry Adder

Theory of Operation

This device is an adder with carry in and out functionality. It is made up of four individual one bit adders that are cascaded with their carries chained together. It can add together 2 four bit numbers and provide an output.

Elaboration Schematic



Design (Ripple)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ripple_adder is
    Port ( cin : in STD_LOGIC;
           a : in STD_LOGIC_VECTOR (3 downto 0);
           b : in STD_LOGIC_VECTOR (3 downto 0);
           s : out STD_LOGIC_VECTOR (3 downto 0);
           cout : out STD_LOGIC);
end ripple_adder;

architecture Behavioral of ripple_adder is

--components
component adder is
    Port (a, b, cin : in std_logic;
          s, cout : out std_logic);
end component;

--signals
signal cout_from0 : std_logic;
signal cout_from1 : std_logic;
signal cout_from2 : std_logic;
signal cout_from3 : std_logic;

begin
 add0: adder
 port map (a => a(0), b => b(0), cin => cin, s => s(0), cout => cout_from0);

 add1: adder
 port map (a => a(1), b => b(1), cin => cout_from0, s => s(1), cout =>
cout_from1);

 add2: adder
 port map (a => a(2), b => b(2), cin => cout_from1, s => s(2), cout =>
cout_from2);

 add3: adder
 port map (a => a(3), b => b(3), cin => cout_from2, s => s(3), cout =>
cout_from3);

 cout <= cout_from3;
```

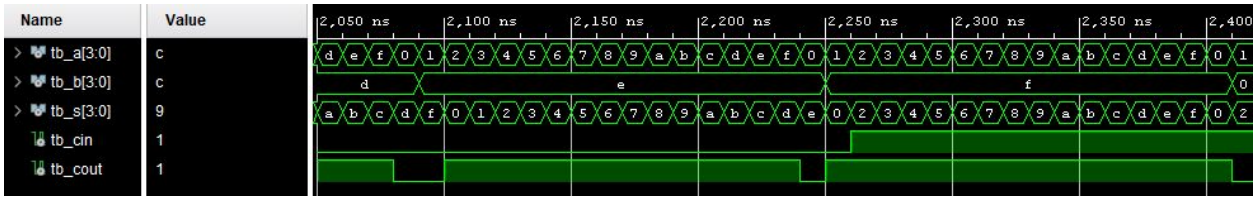
Design (Single)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity adder is
    Port ( a : in STD_LOGIC;
           b : in STD_LOGIC;
           cin : in STD_LOGIC;
           s : out STD_LOGIC;
           cout : out STD_LOGIC);
end adder;

architecture Behavioral of adder is
signal sub_abxor : std_logic;
signal sub_addand : std_logic;
signal sub_xorand : std_logic;
begin

sub_abxor <= a XOR b;
sub_addand <= a AND b;
sub_xorand <= sub_abxor AND cin;
s <= sub_abxor XOR cin;
cout <= sub_addand OR sub_xorand;

end Behavioral;
```

Test Bench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ripple_adder_tb is
end ripple_adder_tb;

architecture testbench of ripple_adder_tb is

--components
component ripple_adder
    Port ( cin : in STD_LOGIC;
           a : in STD_LOGIC_VECTOR (3 downto 0);
           b : in STD_LOGIC_VECTOR (3 downto 0);
           s : out STD_LOGIC_VECTOR (3 downto 0);
           cout : out STD_LOGIC);
end component;

--signals
signal tb_a : std_logic_vector (3 downto 0) := "0000";
signal tb_b : std_logic_vector (3 downto 0) := "0000";
signal tb_s : std_logic_vector (3 downto 0) := "0000";
signal tb_cin : std_logic := '0';
signal tb_cout : std_logic;

begin

dut: ripple_adder port map (cin => tb_cin, a => tb_a, b => tb_b, s => tb_s,
cout => tb_cout);

process begin
    wait for 10ns;
    tb_a <= tb_a +1;
    if (tb_a = "0000") then
    tb_b <= tb_b +1;
    end if;
    if (tb_b = "1111") then
    tb_cin <= '1';
    end if;
end process;
end testbench;
```

Simulation



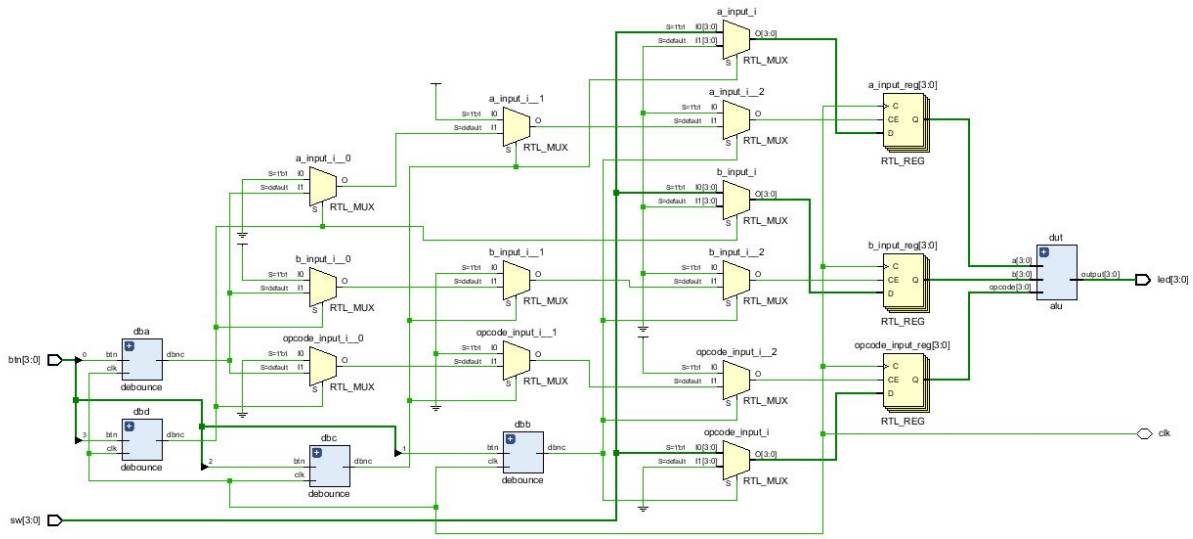
The output verifies the adder's operation.

2) ALU

Theory of Operation

This circuit is a full 4 bit alu with 16 functions. It can take in two four bit numbers and perform operations on them, which are stored in a 4 bit wide register. The alu has been loaded into a test environment that includes component level design and an interface for the ZyBo board.

Elaboration Schematic



Design (Tester)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity alu_tester is
    Port ( btn : in STD_LOGIC_VECTOR (3 downto 0);
           sw : in STD_LOGIC_VECTOR (3 downto 0);
           clk : inout std_logic;
           led : out STD_LOGIC_VECTOR (3 downto 0));
end alu_tester;

architecture rtl_ckt of alu_tester is

--components
component debounce is
Port ( clk : in STD_LOGIC;
           btn : in STD_LOGIC;
           dbnc : out STD_LOGIC);
end component;

component alu is
Port ( opcode : in STD_LOGIC_VECTOR (3 downto 0);
           a : in STD_LOGIC_VECTOR (3 downto 0);
           b : in STD_LOGIC_VECTOR (3 downto 0);
           output : out STD_LOGIC_VECTOR (3 downto 0));
end component;

--signals
signal dbnc_0 : std_logic;
signal dbnc_1 : std_logic;
signal dbnc_2 : std_logic;
signal dbnc_3 : std_logic;
signal debounced : std_logic_vector (3 downto 0);
signal opcode_input : std_logic_Vector (3 downto 0);
signal a_input : std_logic_vector (3 downto 0);
signal b_input : std_logic_vector (3 downto 0);
signal alu_output : std_logic_vector (3 downto 0);
begin

--debounce the buttons
dba: debounce
port map (clk => clk, btn => btn(0), dbnc => dbnc_0);

dbb: debounce
port map (clk => clk, btn => btn(1), dbnc => dbnc_1);

dbc: debounce
```

```
port map (clk => clk, btn => btn(2), dbnc => dbnc_2);

dbd: debounce
port map (clk => clk, btn => btn(3), dbnc => dbnc_3);

debounced <= dbnc_0 & dbnc_1 & dbnc_2 & dbnc_3;

process(clk) --gathering input
begin
    if(rising_edge(clk)) then
        if (debounced(2) = '1') then
            opcode_input <= sw;
        elsif (debounced(1) = '1') then
            a_input <= sw;
        elsif (debounced(0) = '1') then
            b_input <= sw;
        elsif (debounced(3) = '1') then
            opcode_input <= "0000";
            a_input <= "0000";
            b_input <= "0000";
        end if;
    end if;

led <= alu_output; --write the output to the LEDs
end process;

dut: alu
port map (opcode => opcode_input, a => a_input, b => b_input, output => alu_output);
end rtl_ckt;
```

Design (ALU)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity alu is
    Port ( opcode : in STD_LOGIC_VECTOR (3 downto 0);
           a : in STD_LOGIC_VECTOR (3 downto 0);
           b : in STD_LOGIC_VECTOR (3 downto 0);
           output : out STD_LOGIC_VECTOR (3 downto 0));
end alu;

architecture Behavioral of alu is
--components

begin
    main: process(opcode)
    begin
        case (opcode) is --main case statement for ALU operations
        when "0000" => output <= A + B;
        when "0001" => output <= A - B;
        when "0010" => output <= A + 1;
        when "0011" => output <= A - 1;
        when "0100" => output <= 0 - A;
        when "0101" => if (A > B) then output <= "0001"; else output <= "0000"; end
if;
        when "0110" => output <= std_logic_vector(shift_left(unsigned(A), 1));
        when "0111" => output <= std_logic_vector(shift_right(unsigned(A), 1));
        when "1000" => output <= std_logic_vector(shift_right(signed(A), 1));
        when "1001" => output <= NOT A;
        when "1010" => output <= A AND B;
        when "1011" => output <= A OR B;
        when "1100" => output <= A XOR B;
        when "1101" => output <= A XNOR B;
        when "1110" => output <= A NAND B;
        when "1111" => output <= A NOR B;
        end case;
    end process;

end Behavioral;
```

Discussion

Observations

This lab was definitely very interesting to write. I found the logical progression of difficulty from the single adder to the ripple carry adder to the alu to be very smooth, and completing each part prepared me well to complete the next. I didn't struggle too much to finish the content in a timely manner, and was able to enjoy a very short debugging process as the VHDL code was nearly correct on the first try.

Follow-Up

One question that I do have about this lab involves the ripple carry adder. The lab instructions do not mention including the custom adder from part A in the alu, but it is entirely possible to do so. I did not include this design in the final submission, and instead opted for the built in std_numeric function instead for simplicity's sake. The shift operations were also a little strange to implement, and I had to do a lot of reading of the supplied materials to understand how to implement them in the ALU.