Designing a 4-bit binary Adder-Subtractor using Xilinx Vivado IP Integrator

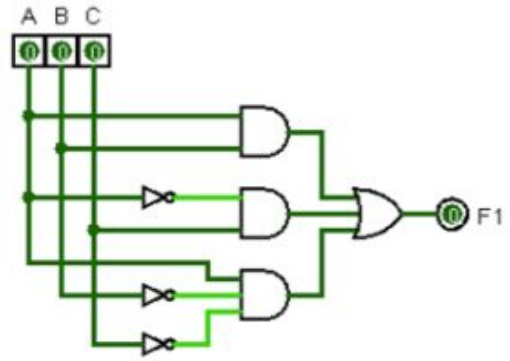
Aaron Mai, Dustin Nguyen
CECS 440
February 18, 2021

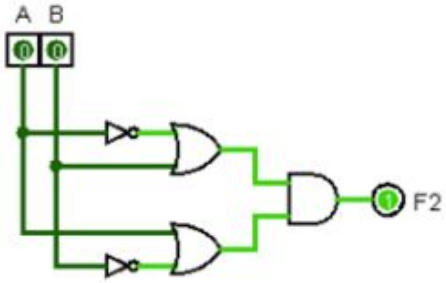
College of Computer Engineering and Science
California State University, Long Beach

1. Introduction

In this lab we further practiced our familiarization with the Vivado IP generator by designing a 4-bit Adder-Subtractor. We had 2 parts to the lab: Part 1 which consisted of building 2 logic circuits with a testbench for each to verify the functionality and part 2 where we created and tested the 4-bit Adder/Subtractor. For part 1 we first created a VHDL code for both circuits and defined our in ports and out ports. We then coded the gates for each of the circuits. We then worked on the testbench creating specific inputs and cases to test our circuits. Before we ran the simulation we created a truth table to lay out all the input and output combinations so that we could verify our simulation results. We approached part 2 similarly to part 1. We first wrote out a basic diagram of the 4-bit adder to get an overview of the modules we would need. The first module we worked on was the full adder. The full adder was built similar to the design we used for a verilog module converted to VHDL. We then moved on to creating the block diagram. Using 4 full adders, 4 xor gates, 8 splicers, and a concatenation block we assembled our diagram. Our diagram had 3 main ports. We had the 3:0 A, 3:0 B, 1 bit carry, and 3:0 Sum. Each of the A inputs connected to a splicer and then into each of the full adders. The B inputs are also connected to a splice but are then connected to a xor gate. This xor gate has its other input connected to the 1 bit carry. This allows the gate to output a signal that changes the adder to a subtractor. The xor gate outputs are then connected to the full adder. The full adder is then connected to the concatenation block which is then connected to our sum output. When designing our testbench we first defined the ports we would be using. Once we define the ports we are able to make test cases to see if our adder was functioning correctly. To verify our results we just did the math of the A and B inputs to confirm the output Sum.

2. Part 1: Pre-Lab Block Diagram

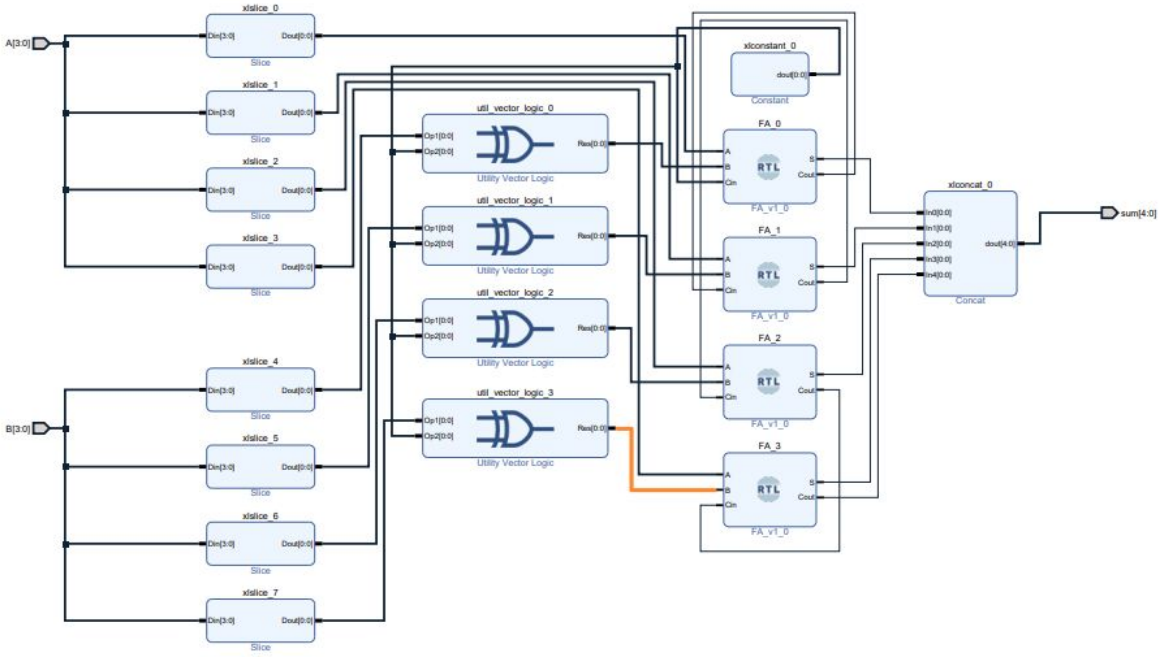




$F_1(A,B,C) = A.B + A'.C + A.B'.C'$

$F_2(A,B,C) = (A'+B).(A+B')$

Part 2: Full Adder/Subtractor Block Diagram



We First create a block diagram to give our design the correct specifications. Here we can define our ports as well as inputs and outputs. This lets us then be able to simulate the design to confirm that it is working as intended.

3. Generate HDL Wrapper code for Adder/Subtractor

```
--Copyright 1986-2020 Xilinx, Inc. All Rights Reserved.
----------------------------------------------------------------------------------
--Tool Version: Vivado v.2020.2 (win64) Build 3064766 Wed Nov 18 09:12:45 MST 2020
--Date        : Thu Feb 18 12:38:07 2021
--Host        : Aaron-PC running 64-bit major release  (build 9200)
--Command     : generate_target design_1_wrapper.bd
--Design      : design_1_wrapper
--Purpose     : IP block netlist
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
entity design_1_wrapper is
  port (
    A : in STD_LOGIC_VECTOR ( 3 downto 0 );
    B : in STD_LOGIC_VECTOR ( 3 downto 0 );
    sum : out STD_LOGIC_VECTOR ( 4 downto 0 )
  );
end design_1_wrapper;

architecture STRUCTURE of design_1_wrapper is
  component design_1 is
  port (
    sum : out STD_LOGIC_VECTOR ( 4 downto 0 );
    B : in STD_LOGIC_VECTOR ( 3 downto 0 );
    A : in STD_LOGIC_VECTOR ( 3 downto 0 )
  );
  end component design_1;
begin
design_1_i: component design_1
     port map (
      A(3 downto 0) => A(3 downto 0),
      B(3 downto 0) => B(3 downto 0),
      sum(4 downto 0) => sum(4 downto 0)
    );
end STRUCTURE;
```

4. TestBenches

Testbench code for Full Adder

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

entity fa4_tb is
--  Port ( );
end fa4_tb;

architecture Behavioral of fa4_tb is
signal A,B: std_logic_vector (3 downto 0);
signal sum: std_logic_vector(4 downto 0);

component design_1_wrapper is
port(
    A: in std_logic_vector (3 downto 0);
    B: in std_logic_vector (3 downto 0);

    sum: out std_logic_vector(4 downto 0)
    );
    end component design_1_wrapper;
begin
design_1_i: design_1_wrapper
    port map(
    A => A,
    B => B,
    sum => sum
    );
    process
        constant period: time := 10 ns;

        begin
        A <= "1110";
        B <= "0001";
        wait for period;

        A <= "1111";
        B <= "0001";
        wait for period;

        A <= "1010";
        B <= "0101";
        wait for period;

        A <= "1010";
        B <= "0101";
        wait for period;
```

```
        A <= "1111";
        B <= "1111";
        wait for period;
        end process;
 end Behavioral;
```

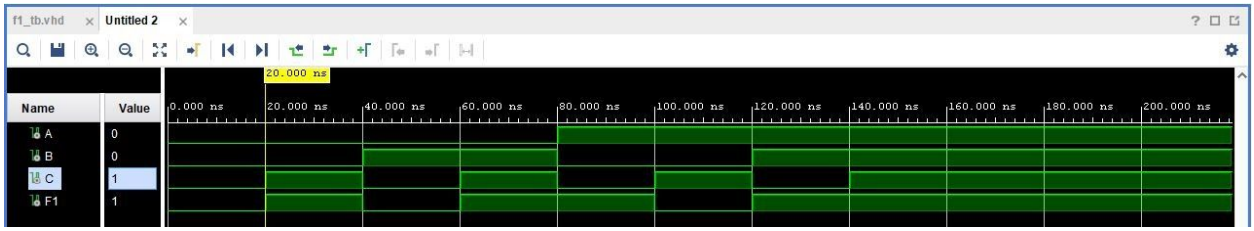
Testbench code for logic circuit 1 + 2

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

entity f2_tb is
--  Port ( );
end f2_tb;

architecture tb of f2_tb is
signal A, B : std_logic;
signal F : std_logic;

begin

    UUT : entity work.logic_circuit_F2 port map (A => A, B => B, F => F);
     A <= '0', '1' after 20 ns, '0' after 40 ns, '1' after 60 ns;
     B <= '0', '1' after 40 ns;

end tb;
```

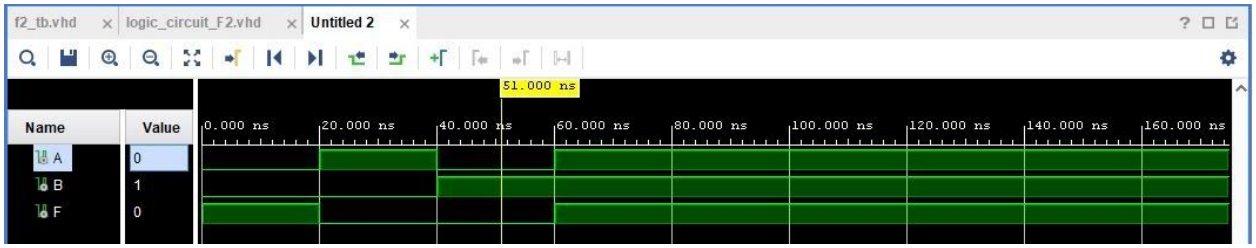
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity f1_tb is
--  Port ( );
end f1_tb;

architecture tb of f1_tb is
signal A, B, C : std_logic;
signal F1 : std_logic;

begin

      UUT : entity work.logic_circuit_f1 port map (A => A, B=> B, C=> C, F1 => F1);
      tb1 : process
        constant period: time := 20 ns;

        begin
            A <= '0';
```

```
        B <= '0';
        C <= '0';
        wait for period;

        A <= '0';
        B <= '0';
        C <= '1';
        wait for period;

        A <= '0';
        B <= '1';
        C <= '0';
        wait for period;

        A <= '0';
        B <= '1';
        C <= '1';
        wait for period;

        A <= '1';
        B <= '0';
        C <= '0';
        wait for period;

        A <= '1';
        B <= '0';
        C <= '1';
        wait for period;

        A <= '1';
        B <= '1';
        C <= '0';
        wait for period;

        A <= '1';
        B <= '1';
        C <= '1';
        wait for period;


        wait;
    end process;
```

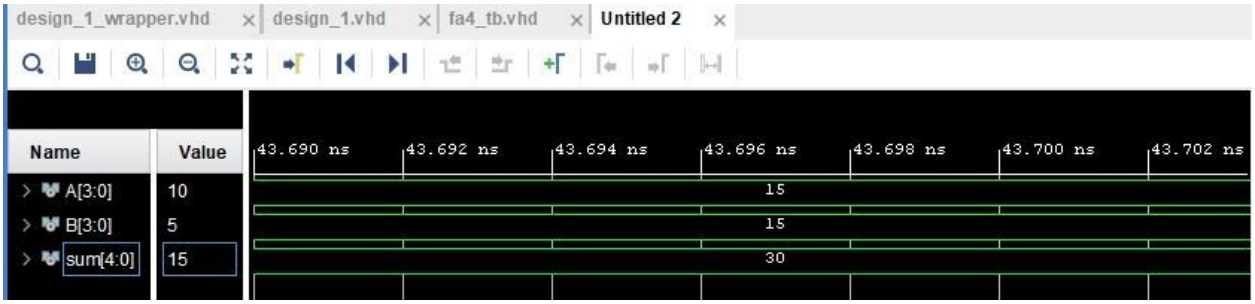
5. Waveforms



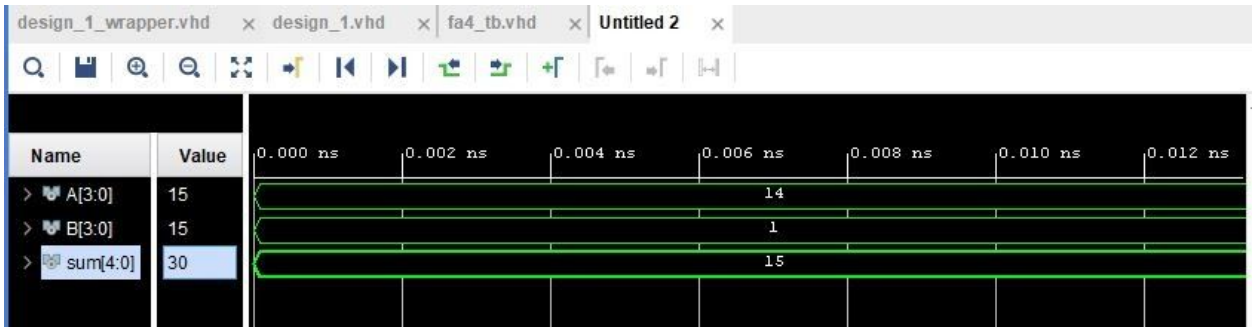
Waveforms for logic circuit F1



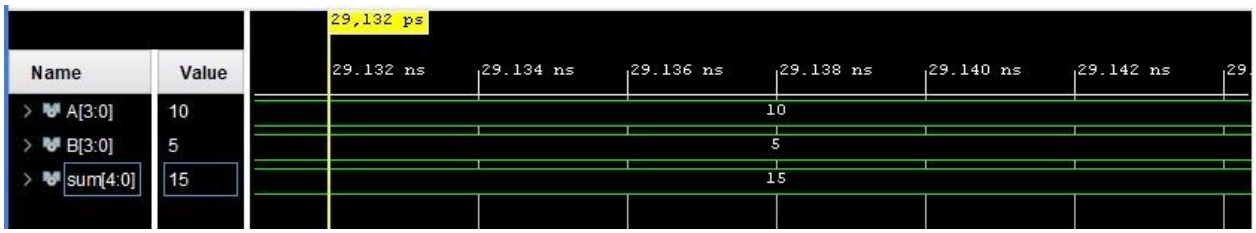
Waveforms for logic circuit F2



Waveforms for 4-bit adder 15 +15 = 30



Waveforms for 4-bit adder 14 + 1 = 15



Waveforms for 4-bit adder 10 + 5 = 15

6. Conclusion

In conclusion we were able to create, build, and verify the circuits given to us using the VHDL and the IP integrator. Learning how to use the code from the full adder and implementing it into a block diagram will help use later on when designing more complicated code. Being able to create the block diagrams also helped us with our understanding of the circuit and gave us a visual of the top level of our design. In the future we hope to continue to develop our knowledge in VHDL.