Lab 1-3: And, Half-Adder and Full-Adder

Alexander Gaskins and Nikola Ciric

Purpose

The three labs being discussed in this report provide concern towards the functional performance of logic gates in providing desired outputs based on varying inputs. It is instantiating a representation of how these various operations work in conjunction with one another to provide different signal results. We begin by exploring AND gates in Lab 1, followed by exploring two corresponding examples of how these logic handlers can be combined to perform real-life functions. In Lab 2, this was done by setting up a schematic that adds two bits together. Lab 3 builds upon this process, incorporating the half-adder design into a more complex schematic that takes in three inputs and returns two outputs. This is called a full adder, and is the backbone behind simple addition performed in many digital systems, beginning with calculators.

Data Collected

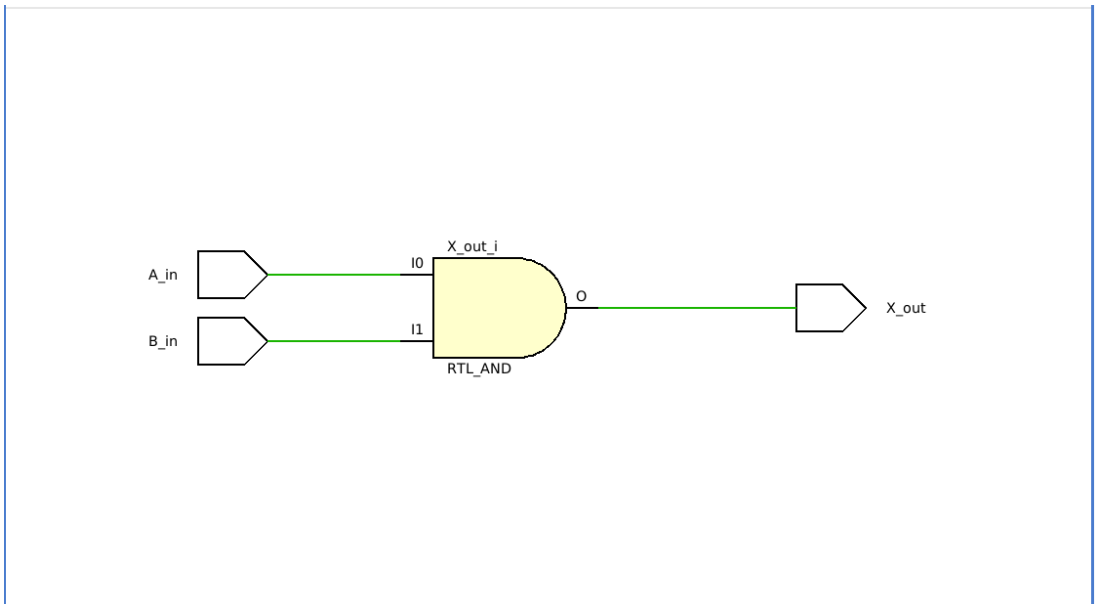
As seen in the code snippet below defining a simple two input AND gate is fairly easy only requiring one line in the architecture definition section. The output labeled as X_out is created by taking two input signals A_in and B_in while they undergo the logical AND operator. Once the synthesis is run this code can be easily viewed in the schematic as seen below which is indeed an AND gate.

Lab 1 VHDL code

/home/nikola/Downloads/VivadoProjects/Lab1/Lab1.srcs/sources_1/new/AND_gate.vhd

```
1
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4
5   entity AND_gate is
6       Port ( A_in : in STD_LOGIC;
7              B_in : in STD_LOGIC;
8              X_out : out STD_LOGIC);
9   end AND_gate;
10
11  architecture ANDFunction of AND_gate is
12  begin
13
14  X_out <= (A_in and B_in);
15
16  end ANDFunction;
17
```

Lab 1 Schematic (AND gate)

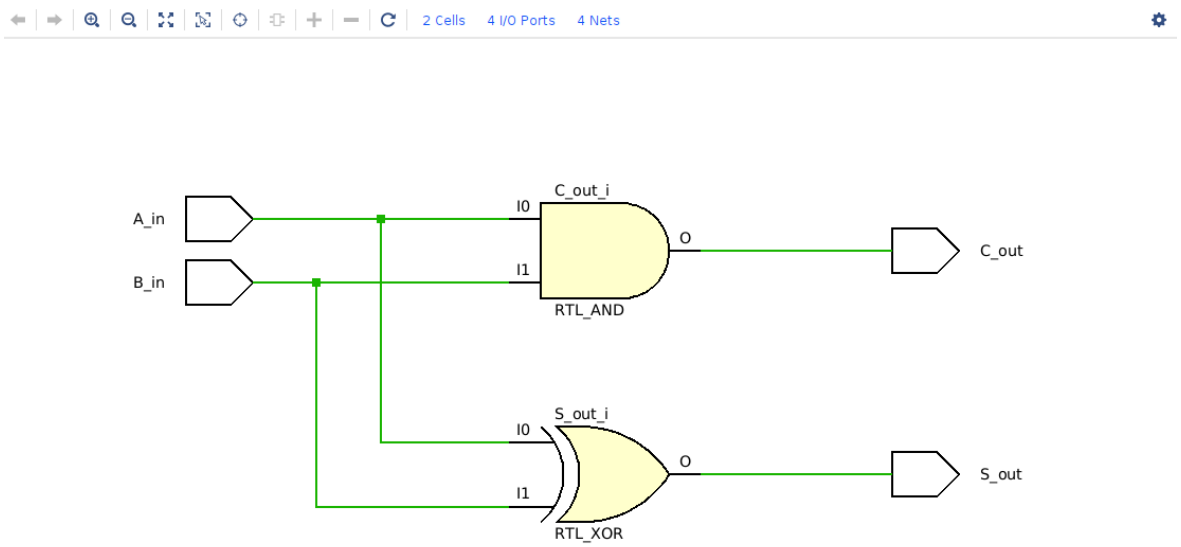


For lab 2 the VHDL code is a little more complex because now the two inputs are not only going through an AND gate but also through a XOR gate. This combination of gates is called a half adder and still only requires two input signals, however outputs two signals called carry and sum. Once again the schematic drawing shows this logic in an easier, more readable way.

Lab 2 VHDL code

```
1
2   library IEEE;
3   use IEEE.STD_LOGIC_1164.ALL;
4
5   entity HalfAdder is
6       Port ( A_in : in STD_LOGIC;
7              B_in : in STD_LOGIC;
8              S_out : out STD_LOGIC;
9              C_out : out STD_LOGIC);
10  end HalfAdder;
11
12  architecture Behavioral of HalfAdder is
13  begin
14  S_out <= (A_in xor B_in);
15  C_out <= (A_in and B_in);
16
17  end Behavioral;
18
```

Lab 2 Schematic (Half Adder)

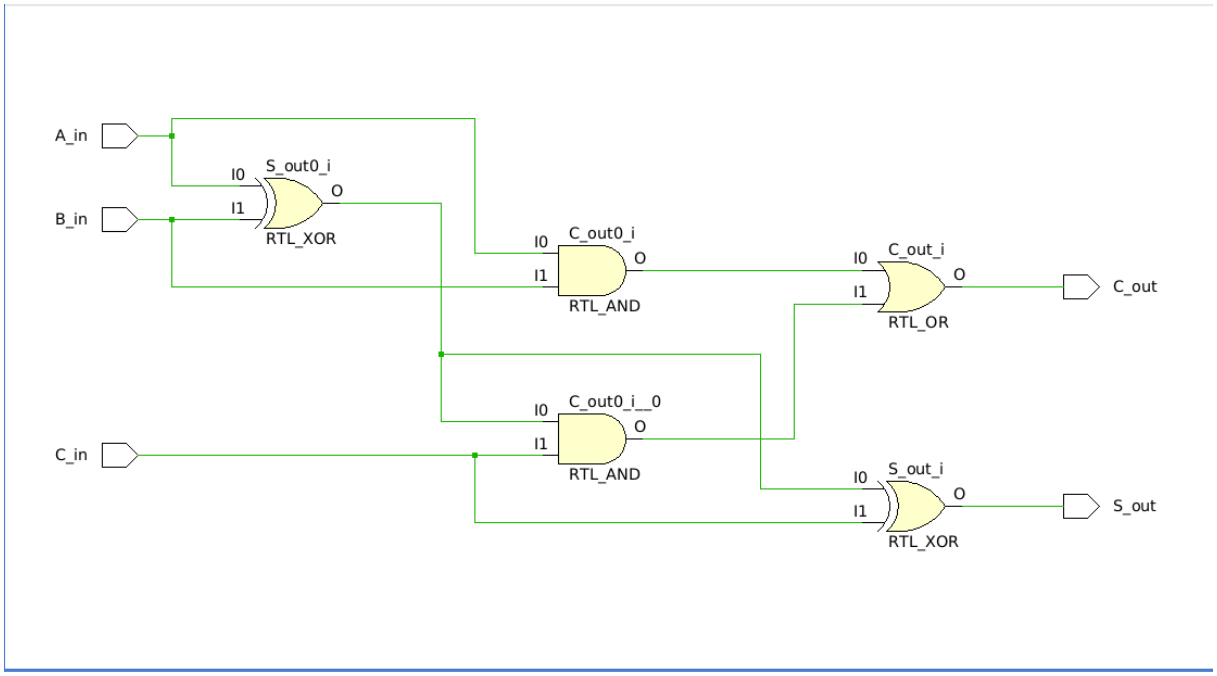


Finally our task for lab 3 was to use what we know from the previous two labs and make a full adder digital circuit. A full adder circuit is basically two half adders that are logically OR together. From the code below and the schematic shows that this type of digital circuit requires three inputs two being signal inputs and the third a carry from the previous operation. This kind of circuit still outputs two signals: a sum and a carry.

Lab 3 VHDL code

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  entity FullAdder is
6      Port ( A_in : in STD_LOGIC;
7             B_in : in STD_LOGIC;
8             C_in : in STD_LOGIC;
9             S_out : out STD_LOGIC;
10            C_out : out STD_LOGIC);
11 end FullAdder;
12
13 architecture Behavioral of FullAdder is
14
15 begin
16 S_out <= ((A_in xor B_in) xor C_in);
17 C_out <= ((A_in and B_in) or ((A_in xor B_in) and C_in));
18 end Behavioral;
19
```

Lab 3 Schematic (Half Adder)



Calculations

AND Truth Table

A_in	B_in	X_out
1	1	1
1	0	0
0	1	0
0	0	0

Half Adder Truth Table

A_in	B_in	S_out	C_out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder Truth Table

A_in	B_in	C_in	S_out	C_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

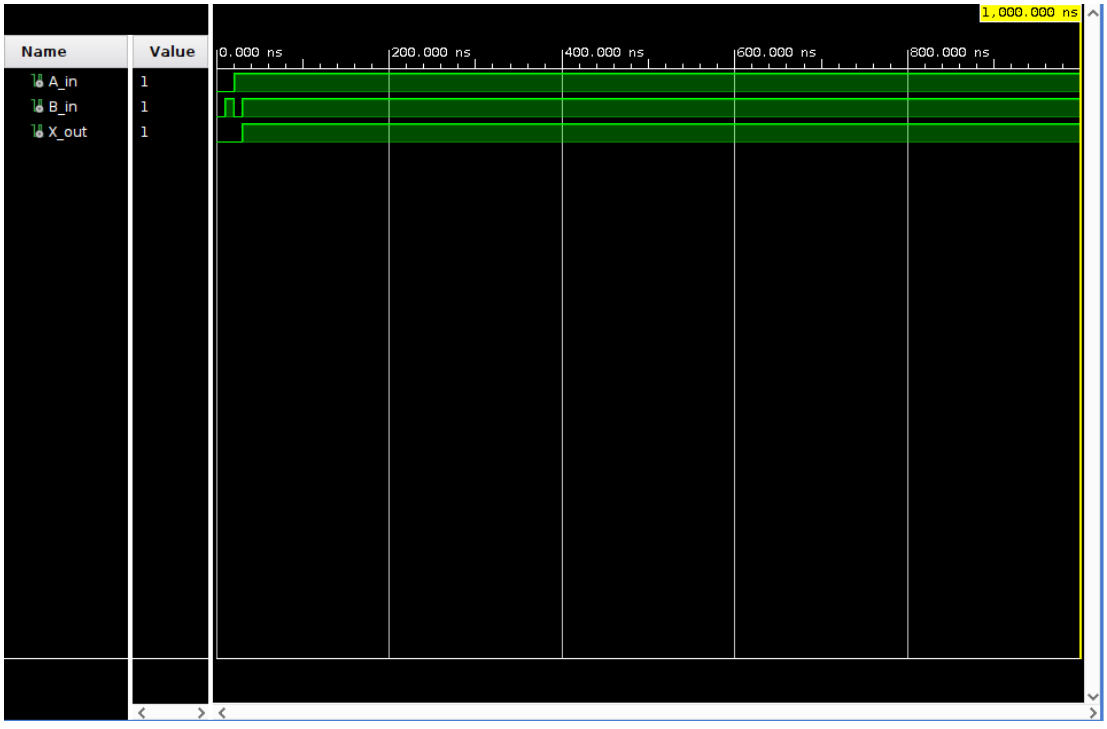
Circuit Simulation/Results

From our original VHDL code we were able to create a new file called the test bench that would basically be the driver code to run the simulation and output in a timestamp graph style. As seen in the graph output for the lab 1 circuit, the output X_out values match up with what should be expected from a two input AND gate.

Lab 1 VHDL Test Bench

```vhdl
library IEEE;
use IEEE.std_logic_1164.ALL;
entity testbench_AND is
end testbench_AND;
architecture Behavioral of testbench_AND is
 component AND_gate
 PORT(  A_in: in STD_LOGIC;
        B_in: in STD_LOGIC;
        X_out: out STD_LOGIC

        );
 end component;

 signal A_in, B_in, X_out : STD_LOGIC;

 begin

 uut: AND_gate
    PORT MAP(   A_in => A_in,
                B_in => B_in,
                X_out => X_out
                );

 process
 begin

    A_in <= '0';
    B_in <= '0';
    wait for 10 ns;
    A_in <= '0';
    B_in <= '1';
    wait for 10 ns;
    A_in <= '1';
    B_in <= '0';
    wait for 10 ns;
    A_in <= '1';
    B_in <= '1';
    wait for 10 ns;

    assert false report "end of test";

    wait;
  end process;
  end Behavioral;
```

Lab 1 Output

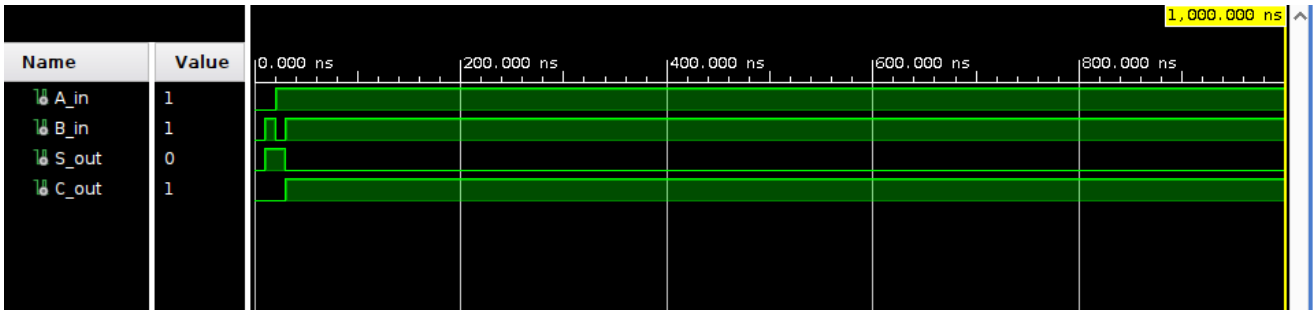


For lab 2 the test bench code is relatively the same as lab1 just with more outputs declared. Once again the output graph represents how the sum and carry output signals will change based on the varying values of the A_in and B_in signals. When comparing our graph results to the truth table of a half adder the results match up perfectly. For example when both A and B have a value of 1 the carry is a 1 and the sum equal to 0.

Lab 2 VHDL Test Bench

```
library IEEE;
use IEEE.std_logic_1164.ALL;
entity testbench_HalfAdder is
end testbench_HalfAdder;
architecture Behavioral of testbench_HalfAdder is
 component HalfAdder
 PORT(  A_in: in STD_LOGIC;
        B_in: in STD_LOGIC;
        S_out: out STD_LOGIC;
        C_out: out std_logic

        );
 end component;

 signal A_in, B_in, S_out,C_out : STD_LOGIC;

 begin

 uut: HalfAdder
    PORT MAP(   A_in => A_in,
                B_in => B_in,
                S_out => S_out,
                C_out => C_out
                );
 process
 begin

    A_in <= '0';
    B_in <= '0';
    wait for 10 ns;
    A_in <= '0';
    B_in <= '1';
    wait for 10 ns;
    A_in <= '1';
    B_in <= '0';
    wait for 10 ns;
    A_in <= '1';
    B_in <= '1';
    wait for 10 ns;

    assert false report "end of test";

    wait;
  end process;
  end Behavioral;
```

Lab 2 Output



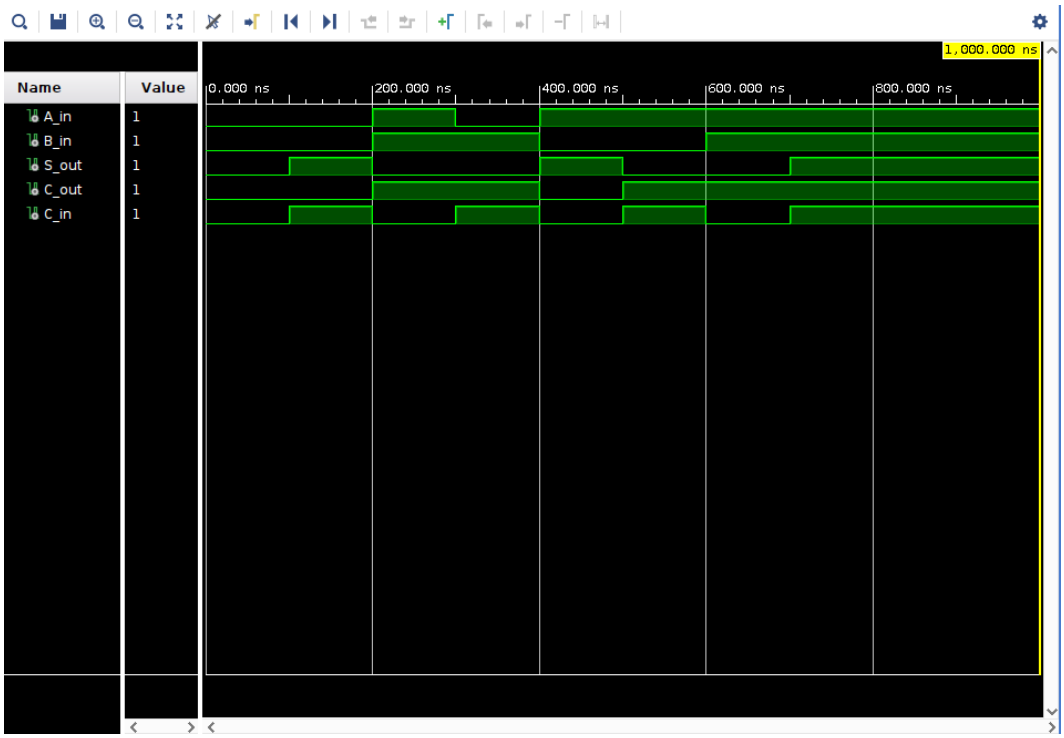
Finally for lab 3 once again the test bench code is very similar but now has one more input declared being C_in, the carry for the previous operation. As seen with the previous two labs the time stamp graph shows how the sum and carry will change as the two input signals and the previous carry are processed through a full adder digital circuit. When compared with the truth table of a full adder

circuit our results match up perfectly. For example When all the input signals are 1 the sum and carry output are both 1.

Lab 3 VHDL Test Bench

```
use IEEE.std_logic_1164.ALL;
entity testbench_FullAdder is

end testbench_FullAdder;
architecture Behavioral of testbench_FullAdder is

 component FullAdder
 PORT(  A_in: in STD_LOGIC;
        B_in: in STD_LOGIC;
        C_in: in std_logic;
        S_out: out STD_LOGIC;
        C_out: out std_logic
        );
 end component;
 signal A_in, B_in, S_out,C_out , C_in: STD_LOGIC;
 begin
 uut: FullAdder
    PORT MAP(   A_in => A_in,
                B_in => B_in,
                C_in => C_in,
                S_out => S_out,
                C_out => C_out
                );
 process
 begin
    A_in <= '0';
    B_in <= '0';
    C_in<='0';
    wait for 100 ns;
    A_in <= '0';
    B_in <= '0';
    C_in<='1';
    wait for 100 ns;
    A_in <= '1';
    B_in <= '1';
    C_in<='0';
    wait for 100 ns;
    A_in <= '0';
    B_in <= '1';
    C_in<='1';
    wait for 100 ns;
    A_in <= '1';
    B_in <= '0';
    C_in<='0';
    wait for 100 ns;
    A_in <= '1';
    B_in <= '0';
    C_in<='1';
    wait for 100 ns;
    A_in <= '1';
    B_in <= '1';
    C_in<='0';
    wait for 100 ns;
    A_in <= '1';
    B_in <= '1';
    C_in<='1';
    wait for 100 ns;

    assert false report "end of test";

    wait;
  end process;
```

Lab 3 output


Name
Value
A_in
1
B_in
1
S_out
1
C_out
1
C_in
1
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns
1,000.000 ns

Conclusion

From the acquired data, it is evident that the results are ideal with regards to the desired functionality corresponding with various input-output operations. We see that the individual gates merely compared and responded to input values. For example, using an AND gate like in Lab 1, a specific condition is implemented upon the input signals, acting similar to a filter, and that it regulates inputs to create a specific result based on what it is given. This particular logic gate is often related to a binary multiplier, as it only provides a true output when both binary input values are true (1). In this regard, an OR gate is seen as a binary adder, with its result depending on at least one true input being supplied. The methodology behind creating more complex schematics all boils down to these two gates, and how bits are handled and recombined. Rather than decimal addition, we essentially just designed a schematic that shifts output values depending on the input. This is why we had the carry value in both Lab 2 and Lab 3, as this allows the schematic to determine whether or not a value should be changed. Altogether, the labs discussed in this report outline how such a simple concept can be easily turned into something much more powerful with a few enhancements.