EEL 4712C - Digital Design: Lab Report

Cole Rottenberg
11062528

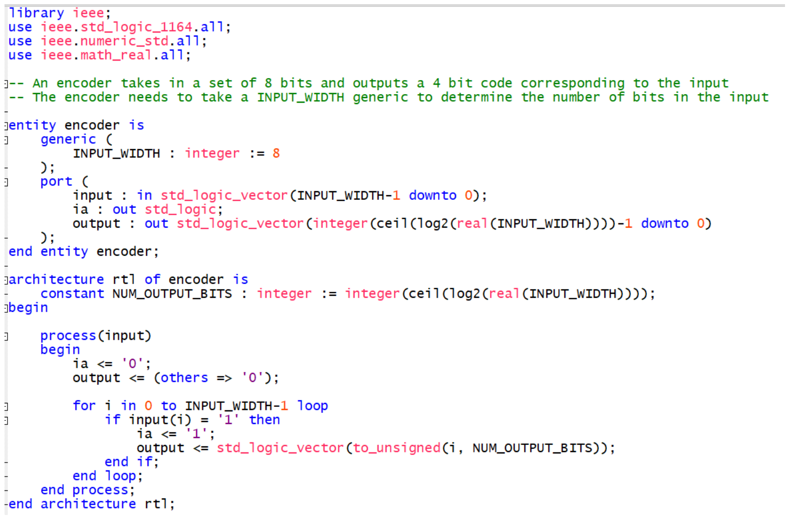
Due Date

Prelab Report

Prelab Design and Implementation

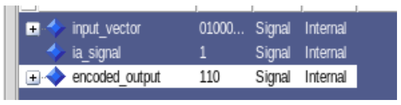
Part 1: RTL Compenent Design and Simulation

1. 8-bit Encoder



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

-- An encoder takes in a set of 8 bits and outputs a 4 bit code corresponding to the input
-- The encoder needs to take a INPUT_WIDTH generic to determine the number of bits in the input

entity encoder is
    generic (
        INPUT_WIDTH : integer := 8
    );
    port (
        input : in std_logic_vector(INPUT_WIDTH-1 downto 0);
        ia : out std_logic;
        output : out std_logic_vector(integer(ceil(log2(real(INPUT_WIDTH))))-1 downto 0)
    );
end entity encoder;

architecture rtl of encoder is
    constant NUM_OUTPUT_BITS : integer := integer(ceil(log2(real(INPUT_WIDTH))));
begin

    process(input)
    begin
        ia <= '0';
        output <= (others => '0');

        for i in 0 to INPUT_WIDTH-1 loop
            if input(i) = '1' then
                ia <= '1';
                output <= std_logic_vector(to_unsigned(i, NUM_OUTPUT_BITS));
            end if;
        end loop;
    end process;
end architecture rtl;
```

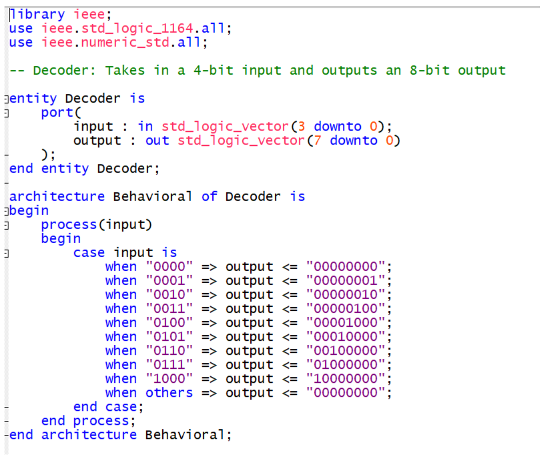
Figure 1: 8-bit Encoder Implementation



```
input_vector    01000...  Signal  Internal
ia_signal       1         Signal  Internal
encoded_output  110       Signal  Internal
```

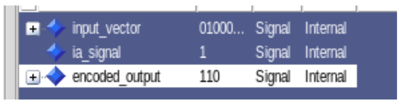
Figure 2: 8-bit Encoder Simulation

2. 8-bit Decoder



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

-- Decoder: Takes in a 4-bit input and outputs an 8-bit output

entity Decoder is
    port(
        input : in std_logic_vector(3 downto 0);
        output : out std_logic_vector(7 downto 0)
    );
end entity Decoder;

architecture Behavioral of Decoder is
begin
    process(input)
    begin
        case input is
            when "0000" => output <= "00000000";
            when "0001" => output <= "00000001";
            when "0010" => output <= "00000010";
            when "0011" => output <= "00000100";
            when "0100" => output <= "00001000";
            when "0101" => output <= "00010000";
            when "0110" => output <= "00100000";
            when "0111" => output <= "01000000";
            when "1000" => output <= "10000000";
            when others => output <= "00000000";
        end case;
    end process;
end architecture Behavioral;
```

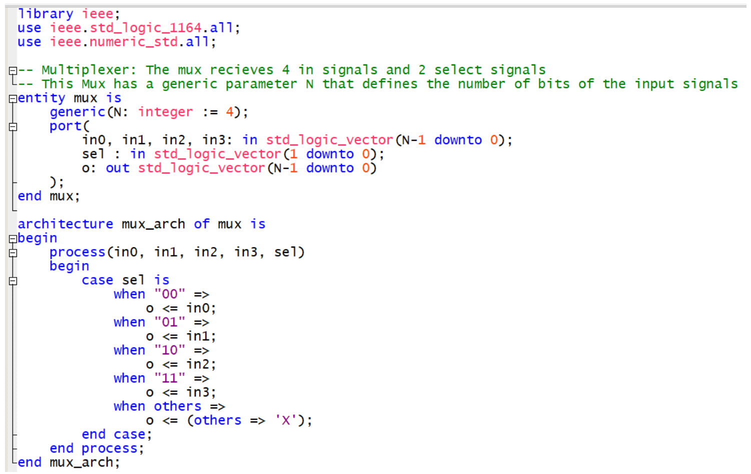
Figure 3: 8-bit Decoder Implementation



input_vector	01000...	Signal	Internal
ia_signal	1	Signal	Internal
encoded_output	110	Signal	Internal

Figure 4: 8-bit Decoder Simulation

3. 8-bit Multiplexer



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

-- Multiplexer: The mux recieves 4 in signals and 2 select signals
-- This Mux has a generic parameter N that defines the number of bits of the input signals
entity mux is
    generic(N: integer := 4);
    port(
        in0, in1, in2, in3: in std_logic_vector(N-1 downto 0);
        sel : in std_logic_vector(1 downto 0);
        o: out std_logic_vector(N-1 downto 0)
    );
end mux;

architecture mux_arch of mux is
begin
    process(in0, in1, in2, in3, sel)
    begin
        case sel is
            when "00" =>
                o <= in0;
            when "01" =>
                o <= in1;
            when "10" =>
                o <= in2;
            when "11" =>
                o <= in3;
            when others =>
                o <= (others => 'X');
        end case;
    end process;
end mux_arch;
```

Figure 5: 8-bit Multiplexer Implementation

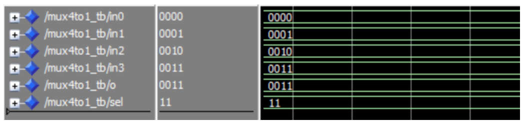
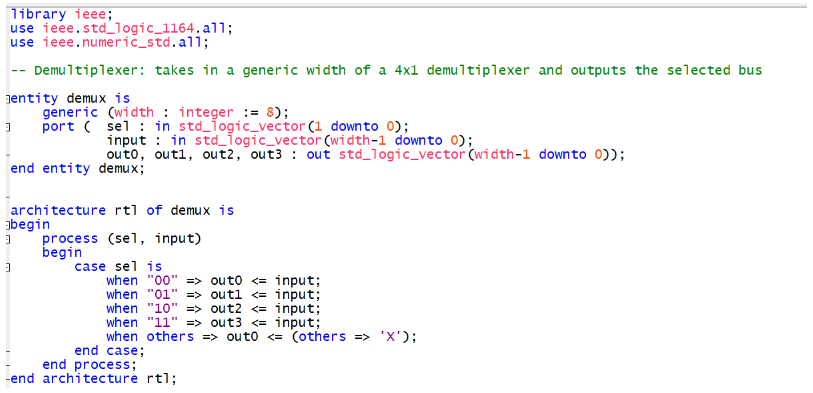

/mux4to1_tb/in0 0000 0000
/mux4to1_tb/in1 0001 0001
/mux4to1_tb/in2 0010 0010
/mux4to1_tb/in3 0011 0011
/mux4to1_tb/o 0011 0011
/mux4to1_tb/sel 11 11

Figure 6: 8-bit Multiplexer Simulation

4. 8-bit Demultiplexer



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- Demultiplexer: takes in a generic width of a 4x1 demultiplexer and outputs the selected bus

entity demux is
    generic (width : integer := 8);
    port (  sel : in std_logic_vector(1 downto 0);
            input : in std_logic_vector(width-1 downto 0);
            out0, out1, out2, out3 : out std_logic_vector(width-1 downto 0));
end entity demux;


architecture rtl of demux is
begin
    process (sel, input)
    begin
        case sel is
            when "00" => out0 <= input;
            when "01" => out1 <= input;
            when "10" => out2 <= input;
            when "11" => out3 <= input;
            when others => out0 <= (others => 'X');
        end case;
    end process;
end architecture rtl;
```

Figure 7: 8-bit Demultiplexer Implementation

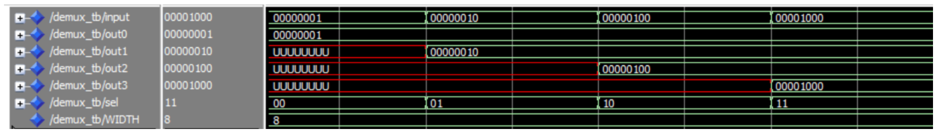
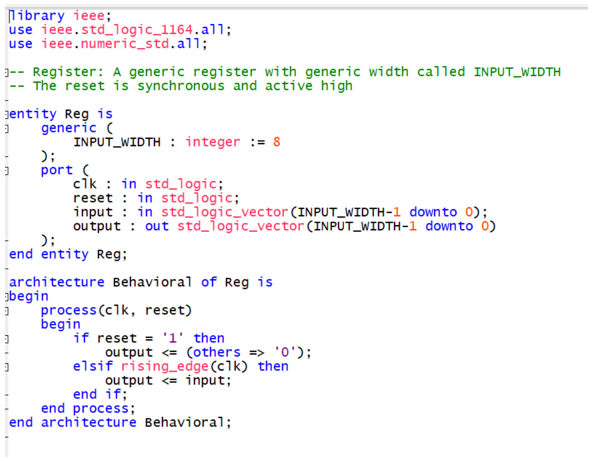

/demux_tb/input 00001000 00000001 00000010 00000100 00001000
/demux_tb/out0 00000001 00000001
/demux_tb/out1 00000010 UUUUUUUU 00000010
/demux_tb/out2 00000100 UUUUUUUU 00000100
/demux_tb/out3 00001000 UUUUUUUU 00001000
/demux_tb/sel 11 00 01 10 11
/demux_tb/WIDTH 8 8

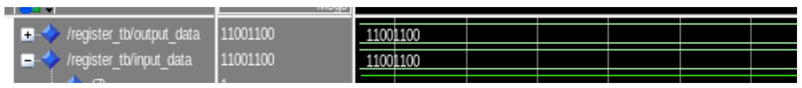
Figure 8: 8-bit Demultiplexer Simulation

5. Register



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- Register: A generic register with generic width called INPUT_WIDTH
-- The reset is synchronous and active high

entity Reg is
    generic (
        INPUT_WIDTH : integer := 8
    );
    port (
        clk : in std_logic;
        reset : in std_logic;
        input : in std_logic_vector(INPUT_WIDTH-1 downto 0);
        output : out std_logic_vector(INPUT_WIDTH-1 downto 0)
    );
end entity Reg;

architecture Behavioral of Reg is
begin
    process(clk, reset)
    begin
        if reset = '1' then
            output <= (others => '0');
        elsif rising_edge(clk) then
            output <= input;
        end if;
    end process;
end architecture Behavioral;
```

Figure 9: Register Implementation

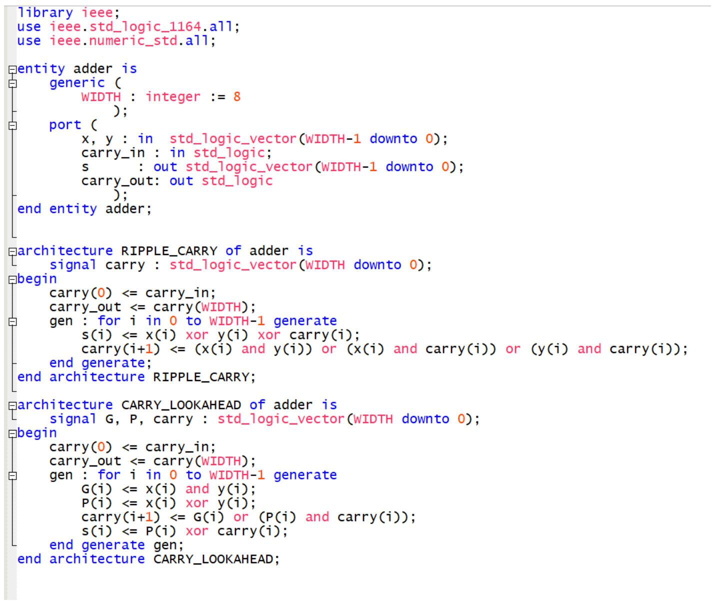


```
/register_tb/output_data   11001100   11001100
/register_tb/input_data    11001100   11001100
```

Figure 10: Register Simulation

Part 2: Ripple Carry Adder

1. Implementation of RC Adder

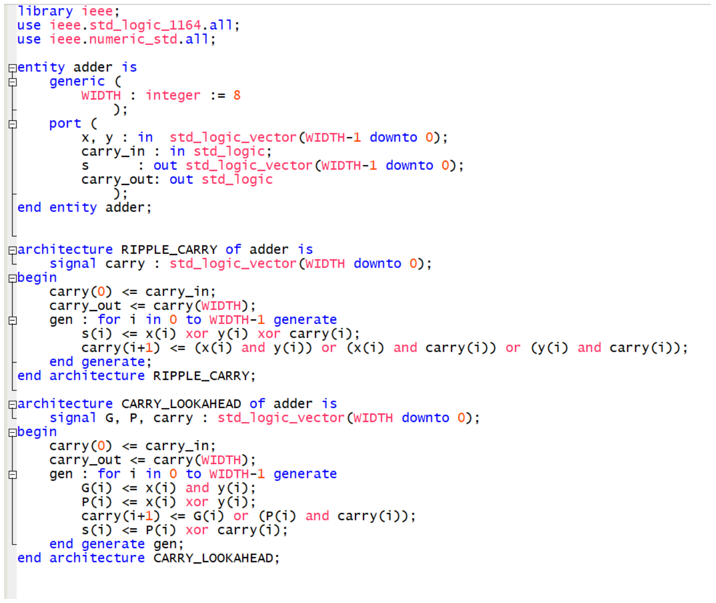


```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder is
    generic (
        WIDTH : integer := 8
            );
    port (
        x, y : in  std_logic_vector(WIDTH-1 downto 0);
        carry_in : in std_logic;
        s       : out std_logic_vector(WIDTH-1 downto 0);
        carry_out: out std_logic
            );
end entity adder;

architecture RIPPLE_CARRY of adder is
    signal carry : std_logic_vector(WIDTH downto 0);
begin
    carry(0) <= carry_in;
    carry_out <= carry(WIDTH);
    gen : for i in 0 to WIDTH-1 generate
        s(i) <= x(i) xor y(i) xor carry(i);
        carry(i+1) <= (x(i) and y(i)) or (x(i) and carry(i)) or (y(i) and carry(i));
    end generate;
end architecture RIPPLE_CARRY;

architecture CARRY_LOOKAHEAD of adder is
    signal G, P, carry : std_logic_vector(WIDTH downto 0);
begin
    carry(0) <= carry_in;
    carry_out <= carry(WIDTH);
    gen : for i in 0 to WIDTH-1 generate
        G(i) <= x(i) and y(i);
        P(i) <= x(i) xor y(i);
        carry(i+1) <= G(i) or (P(i) and carry(i));
        s(i) <= P(i) xor carry(i);
    end generate gen;
end architecture CARRY_LOOKAHEAD;
```

Figure 11: Ripple Carry Adder Implementation

Part 3: Carry Lookahead Adder

1. Implementation of CL Adder



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

entity adder is
    generic (
        WIDTH : integer := 8
            );
    port (
        x, y : in  std_logic_vector(WIDTH-1 downto 0);
        carry_in : in std_logic;
        s       : out std_logic_vector(WIDTH-1 downto 0);
        carry_out: out std_logic
            );
end entity adder;

architecture RIPPLE_CARRY of adder is
    signal carry : std_logic_vector(WIDTH downto 0);
begin
    carry(0) <= carry_in;
    carry_out <= carry(WIDTH);
    gen : for i in 0 to WIDTH-1 generate
        s(i) <= x(i) xor y(i) xor carry(i);
        carry(i+1) <= (x(i) and y(i)) or (x(i) and carry(i)) or (y(i) and carry(i));
    end generate;
end architecture RIPPLE_CARRY;

architecture CARRY_LOOKAHEAD of adder is
    signal G, P, carry : std_logic_vector(WIDTH downto 0);
begin
    carry(0) <= carry_in;
    carry_out <= carry(WIDTH);
    gen : for i in 0 to WIDTH-1 generate
        G(i) <= x(i) and y(i);
        P(i) <= x(i) xor y(i);
        carry(i+1) <= G(i) or (P(i) and carry(i));
        s(i) <= P(i) xor carry(i);
    end generate gen;
end architecture CARRY_LOOKAHEAD;
```

Figure 12: Carry Lookahead Adder Implementation

Reflection

I struggled to use the Questa Sim simulator and ended up deciding to use ModelSim instead. After I made the switch, I was able to complete the prelab without any issues. I learned a lot about the different components and how they can be used to build more complex systems. I also learned about the different types of signals and how they can be used to control the flow of data in a system.

Prelab Homework

Postlab Report

Problem Statement

The goal of Lab 1 focused on the design and implementation of various digital components. The lab required the design and simulation of an 8-bit encoder, 8-bit decoder, 8-bit multiplexer, 8-bit demultiplexer, and an 8-bit register. The lab also required the design and simulation of a ripple carry adder and a carry lookahead adder. The lab was designed to help students understand the basics of digital design and how to use various components to build more complex systems.

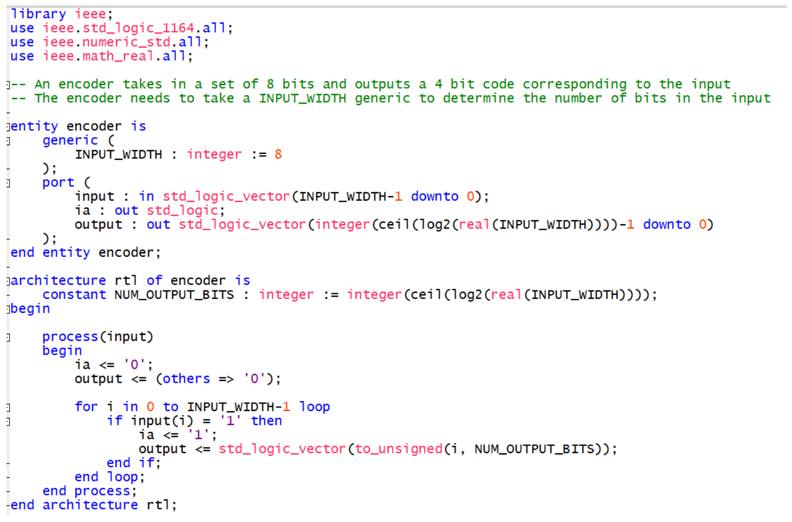
Design

The overall design of the RTL components was relatively straightforward. The 8-bit encoder and decoder were designed using a case statement to map the input to the output. The multiplexer and demultiplexer were designed using a case statement to select the input or output based on the control signal. The register was designed using a process statement to control the flow of data.

The ripple carry adder was more challenging than the previous components designed. The adder was designed to first allow for a generic input size. The logic used a for generate statement to create the full adders and connect them together. The carry lookahead adder was designed using a generate statement to create the carry lookahead logic and then connect the full adders together. The individual logic differed with in each for generate statement, but the overall structure was the same. The pros of using this design method allows for the components to be easily reused and modified.

Implementation

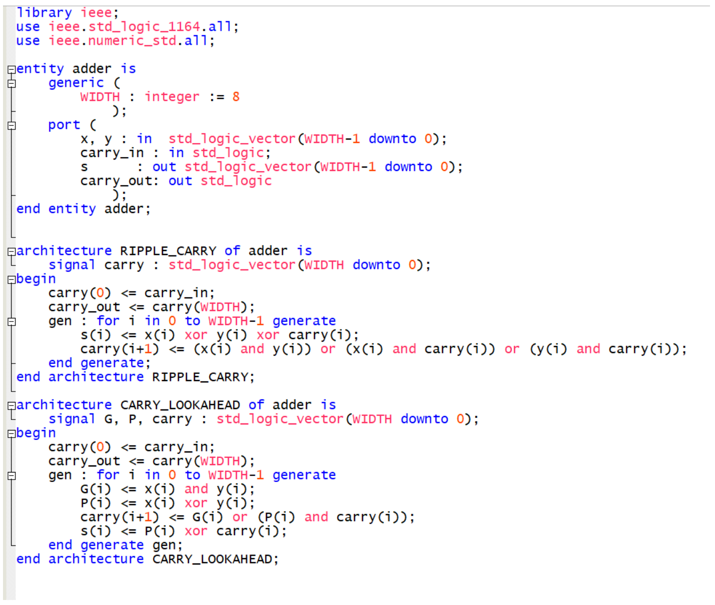
As described before the implementation followed a similar pattern for each component. The 8-bit encoder and decoder were designed using a case statement to map the input to the output. The multiplexer and demultiplexer were designed using a case statement to select the input or output based on the control signal. The register was designed using a process statement to control the flow of data. The figure below shows the implementation of the 8-bit encoder as a base example.



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

-- An encoder takes in a set of 8 bits and outputs a 4 bit code corresponding to the input
-- The encoder needs to take a INPUT_WIDTH generic to determine the number of bits in the input

entity encoder is
    generic (
        INPUT_WIDTH : integer := 8
    );
    port (
        input : in std_logic_vector(INPUT_WIDTH-1 downto 0);
        ia : out std_logic;
        output : out std_logic_vector(integer(ceil(log2(real(INPUT_WIDTH))))-1 downto 0)
    );
end entity encoder;

architecture rtl of encoder is
    constant NUM_OUTPUT_BITS : integer := integer(ceil(log2(real(INPUT_WIDTH))));
begin

    process(input)
    begin
        ia <= '0';
        output <= (others => '0');

        for i in 0 to INPUT_WIDTH-1 loop
            if input(i) = '1' then
                ia <= '1';
                output <= std_logic_vector(to_unsigned(i, NUM_OUTPUT_BITS));
            end if;
        end loop;
    end process;
end architecture rtl;
```

Figure 13: 8-bit Encoder Implementation

The implementation of the ripple carry adder was more complex than the previous components. The adder was designed to first allow for a generic input size. The logic used a for generate statement to create the full adders and connect them together. The carry lookahead adder was designed using a generate statement to create the carry lookahead logic and then connect the full adders together. The individual logic differed with in each for generate statement, but the overall structure was the same. The figure below shows the implementation of the ripple carry adder and carry lookahead adder.



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder is
    generic (
        WIDTH : integer := 8
            );
    port (
        x, y : in  std_logic_vector(WIDTH-1 downto 0);
        carry_in : in std_logic;
        s       : out std_logic_vector(WIDTH-1 downto 0);
        carry_out: out std_logic
            );
end entity adder;

architecture RIPPLE_CARRY of adder is
    signal carry : std_logic_vector(WIDTH downto 0);
begin
    carry(0) <= carry_in;
    carry_out <= carry(WIDTH);
    gen : for i in 0 to WIDTH-1 generate
        s(i) <= x(i) xor y(i) xor carry(i);
        carry(i+1) <= (x(i) and y(i)) or (x(i) and carry(i)) or (y(i) and carry(i));
    end generate;
end architecture RIPPLE_CARRY;

architecture CARRY_LOOKAHEAD of adder is
    signal G, P, carry : std_logic_vector(WIDTH downto 0);
begin
    carry(0) <= carry_in;
    carry_out <= carry(WIDTH);
    gen : for i in 0 to WIDTH-1 generate
        G(i) <= x(i) and y(i);
        P(i) <= x(i) xor y(i);
        carry(i+1) <= G(i) or (P(i) and carry(i));
        s(i) <= P(i) xor carry(i);
    end generate gen;
end architecture CARRY_LOOKAHEAD;
```

Figure 14: Ripple Carry Adder Implementation

Testing

The design was tested using ModelSim. The testbench for each component was designed to test the functionality of the component. The testbench for the 8-bit demultiplexer is shown below.

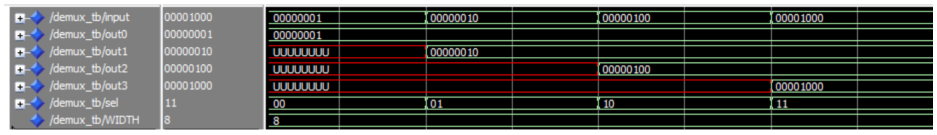
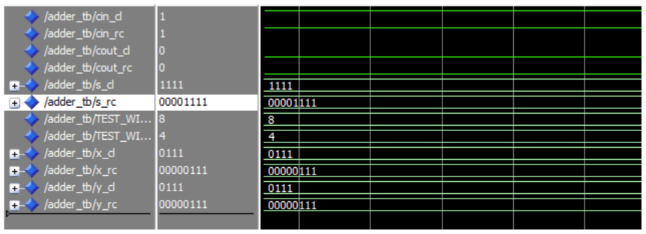

/demux_tb/input 00001000 00000001 00000010 00000100 00001000
/demux_tb/out0 00000001 00000001
/demux_tb/out1 00000010 UUUUUUUU 00000010
/demux_tb/out2 00000100 UUUUUUUU 00000100
/demux_tb/out3 00001000 UUUUUUUU 00001000
/demux_tb/sel 11 00 01 10 11
/demux_tb/WIDTH 8 8

Figure 15: 8-bit Encoder Simulation

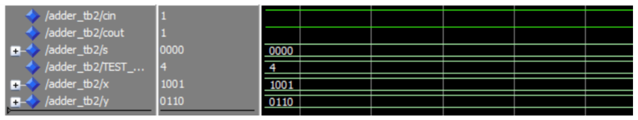
The testbench for the ripple carry adder is shown below. It also includes the carry lookahead adder. The signals of the ripple carry adder are denoted with "rc" and the signals of the carry lookahead adder are denoted with "cl".



```
/adder_tb/cin_cl        1
/adder_tb/cin_rc        1
/adder_tb/cout_cl       0
/adder_tb/cout_rc       0
/adder_tb/s_cl          1111        1111
/adder_tb/s_rc          00001111    00001111
/adder_tb/TEST_WI...    8           8
/adder_tb/TEST_WI...    4           4
/adder_tb/x_cl          0111        0111
/adder_tb/x_rc          00000111    00000111
/adder_tb/y_cl          0111        0111
/adder_tb/y_rc          00000111    00000111
```

Figure 16: Adder Simulation

The second testbench for the adder is shown below. It is more simplistic.



```
/adder_tb2/cin          1
/adder_tb2/cout         1
/adder_tb2/s            0000        0000
/adder_tb2/TEST_...     4           4
/adder_tb2/x            1001        1001
/adder_tb2/y            0110        0110
```

Figure 17: Adder Simulation 2

Conclusions

The work described in the report was more time intensive which reflects the learning process of the suite of tools at my disposal. The design and implementation of the components was relatively straightforward. The testing of the components was also straightforward. The only issue I encountered was using the Questa Sim simulator. I was able to resolve this issue by using ModelSim. The future improvements for the lab would be to use the Questa Sim simulator and to improve the testbenches for the components. Moving forward, challenges like the one I faced will be easier to overcome considering the tooling will remain the same for future labs.

Appendix