ECSE 325

Lab 2 Report

Lucas Bluethner, 260664905

Yutian Jing, 260680087

March 23, 2018

Introduction

The purpose of this lab was to learn how to implement digital filters using VHDL, and to verify the design by performing testbench simulations using ModelSim. In specific, a 25-tap Finite Impulse Response (FIR) filter was required to be implemented in this lab. An FIR filter is a filter whose response to any finite length input is of finite period. This filter is implemented by convolving the input signal with the digital filter's impulse response. For a causal filter of order N, each value of the output sequence is a weighted sum of the most recent input values:

$$y(n) = \sum_{i=0}^{N} b_i \times x(n-i),$$

where $x(n)$, $y(n)$ and b_i are the input signal, output signal, and weights, respectively. *Fig. 1* shows the direct form of an N-tap FIR filter.

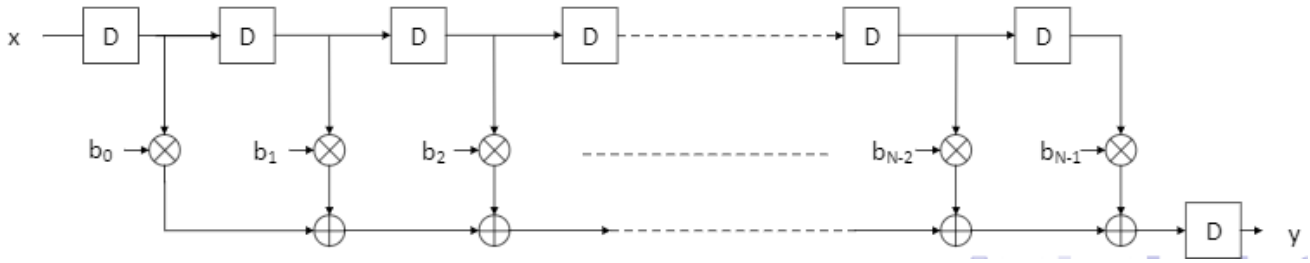


Fig. 1 Direct form of an N-tap FIR filter

Code Implementation

We were given two files, ***lab3-In.txt*** and ***lab3-coef.txt***. ***lab3-In.txt*** contains 1000 floating-point numbers, which will be the inputs to the filter implemented in VHDL, and ***lab3-coef.txt*** contains the weights (or coefficients) in floating point representation. The first task was to write a program to convert these floating-point numbers to fixed-point and write them to two new files. Below is the the 'C' code we modified from lab 2 to implement this conversion.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

char *decimal_to_binary(int);

/*
 * This method reads the floating point values from to separate files,
 * convert them to fixed point, and stores them in two new files
 */
void main()
{
    // Open the files
    FILE *lab3_in = fopen("lab3-In.txt", "r");
    FILE *lab3_coef= fopen("lab3-coef.txt", "r");

    // Create arrarys to hold the x values and y values
    float inputs[1000];
    float coef[25];
    int i;

    // Read each floating point number from the files and store them in their corresponding array
    for (i = 0; i < 1000; i++)
    {
        fscanf(lab3_in, "%f", &inputs[i]);
    }
    for (i = 0; i < 25; i++)
    {
        fscanf(lab3_coef, "%f", &coef[i]);
    }

    // Done with these files, close them
    fclose(lab3_in);
    fclose(lab3_coef);

    // Go through th arrary multiplying each number by 2^15 because
    // b = a x 2^F where a = floating point number, F = fractional length
    for (i = 0; i < 1000; i++)
    {
        inputs[i] = inputs[i] * 32768;
    }

    for (i = 0; i < 25; i++)
```

```
    {
        coef[i] = coef[i] * 32768;
    }

    // Open the files where the fixed point values will be written to
    FILE *lab3_in_fixed_point = fopen("lab3-in-fixed-point.txt", "w");
    FILE *lab3_coef_fixed_point = fopen("lab3-coef-fixed-point.txt", "w");

    int n;
    char *pointer;

    // iterate through each number, convert it to binary (2's complement) then print it to
it's corresponding file
    for (i = 0; i < 1000; i++)
    {
        n = (int)inputs[i];
        pointer = decimal_to_binary(n);
        fprintf(lab3_in_fixed_point, "%s\n", pointer);
    }
    for (i = 0; i < 25; i++)
    {
        n = (int)coef[i];
        pointer = decimal_to_binary(n);
        fprintf(lab3_coef_fixed_point, "%s\n", pointer);
    }

    // Free the pointer from memory and close the files because we are done with them
    free(pointer);
    fclose(lab3_in_fixed_point);
    fclose(lab3_coef_fixed_point);
}

/*
 * This method take an integer as an argument an converts it to a 16-bit binary number
 * If the integer is negative, the number gets convert to its positive binary value,
 * then gets converted again to 2's complement
 * returns a pointer to the MSB of the binary number in memory
 */
char *decimal_to_binary(int n)
{
    int c, d, count, i, invert_spot;
    char *pointer;
    int neg_number = 0;

    // intialize the counter
    count = 0;

    // will point to the MSB (sign bit) of the binary number in memory
    pointer = (char *)malloc(16 + 1);

    // Array to hold the binary digits
    int binary_digits[16];


```

```
95      if (pointer == NULL)
96          exit(EXIT_FAILURE);
97
98      // If the number is negative, make it positive and set the neg_number flag high
99      if (n < 0)
100     {
101         n = n * (-1);
102         neg_number = 1;
103     }
104
105     // Converts the decimal number to its unsigned binary value storing it bit by bit in
    the array
106     for (c = 15; c >= 0; c--)
107     {
108         d = n >> c;
109
110         if (d & 1)
111             binary_digits[count] = 1;
112         else
113             binary_digits[count] = 0;
114         count++;
115     }
116
117     // if the argument n was postive, simply copy the values from the array to memory
    starting at the pointer
118     if(!neg_number)
119     {
120         for(i = 0; i < 16; i++)
121         {
122             if(binary_digits[i] == 0)
123             {
124                 *(pointer + i) = 0 + '0';
125             }
126             else
127             {
128                 *(pointer + i) = 1 + '0';
129             }
130         }
131     }
132     else
133     // the argument n was positive, convert to 2's complement
134     {
135         // Starting from the LSB, iterate through the array backwards until the first '1'
    bit is reached
136         // All the numbers before (and not including) this 1 value need to be inverted
137         for (i = 15; i >= 0; i--)
138         {
139             if (binary_digits[i] == 1)
140             {
141                 invert_spot = i;
142                 break;
143             }

144         }
```

```
145          // Starting at the MSB, iterate throught the arrary storing the inverted value in
     memory starting at the pointer
146         for (i = 0; i < 16; i++)
147         {
148             if (i < invert_spot)
149             {
150                 if (binary_digits[i] == 0)
151                 {
152                     *(pointer + i) = 1 + '0';
153                 }
154                 else
155                 {
156                     *(pointer + i) = 0 + '0';
157                 }
158             }
159             else
160             // Once the '1' found above is reached, simply copy the values over from the
     array to the next spot in memory
161             {
162                 if (binary_digits[i] == 0)
163                 {
164                     *(pointer + i) = 0 + '0';
165                 }
166                 else
167                 {
168                     *(pointer + i) = 1 + '0';
169                 }
170             }
171         }
172     }
173 
174     // Add the null terminator to the end of the string
175     *(pointer + count) = '\0';
176 
177     // return a pointer the the MSB in memory
178     return pointer;
179 }
```

After converting the files to fixed point representation, the next task was implement the 25-tap FIR filter in VDHL, using the values in ***lab3-in-fixed-point.txt*** as the inputs to the filter. The values from ***lab3-coef-fixed-point.txt*** were not read from the file like the input value, but instead kept in array. We also used an array to store the 25 most recent input values. Next, all we had to do was implement the formula from earlier. Given an new input x, the entire input array gets shifted by one index and the new input gets inserted at the zero index. Each of the 25 most recent input values is then multiplied by the corresponding weight and the product is accumulated to the total output, y. Upon iterating through the entire array, y is the filtered value of x. An important step is to reset y back to zero (line 59) after an input gets filtered (once per clock cycle), otherwise the output will get accumulated with the previous filtered values as well.

Below is the VHDL code that implements the 25-tap FIR filter.

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

entity g64_lab3 is
    port (  x       : in std_logic_vector (15 downto 0);   -- input signal
            clk     : in std_logic;                        -- clock
            rst     : in std_logic;             -- asynchronous active-high reset
            y       : out std_logic_vector (16 downto 0));  -- output signal
end g64_lab3;

architecture filter of g64_lab3 is

type COEFS is array(0 to 24) of signed(15 downto 0); -- arrary for coefficients
signal coef : COEFS;
type INPUTS is array(0 to 24) of signed(15 downto 0); -- array to store 25 most recent
inputs
signal input : INPUTS := (others => "0000000000000000");

begin

-- coefficient initializations
coef <= ("0000001001110010",
            "0000000000010001",
            "1111111111010011",
            "1111111011011110",
            "0000001100011001",
            "1111110110100111",
            "1111110000001110",
            "0000110110111100",
            "1110110001110011",
            "0000110111110111",
            "0000001100000111",
            "1110101000001010",
            "0001111000110011",
            "1110101000001010",
            "0000001100000111",
            "0000110111110111",
            "1110110001110011",
            "0000110110111100",
            "1111110000001110",
            "1111110110100111",
            "0000001100011001",
            "1111111011011110",
            "1111111111010011",
            "0000000000010001",
            "0000001001110010");

    process(clk, rst)
        variable mult  : signed(31 downto 0) := (others => '0');
        variable y_out  : signed(16 downto 0) := (others => '0');
```

```
        begin
            if (rst = '1') then
                -- asynchronous active-high reset: set everything back to zero
                mult := (others => '0');
                y_out := (others => '0');
                y <= (others => '0');
            elsif (rising_edge(clk)) then
                -- reset output zero so previous output value isn't accumulated
                y_out := (others => '0');
                for i in 0 to 23 loop
                    input(i+1) <= input(i); -- shift each input by one in the array
                end loop;                  -- (overwriting the oldest value)
                -- newest value is put into the input array at index 0
                input(0) <= signed(x);
                for n in 0 to 24 loop(
                     -- multiply each input by corresponding coefficient
                    mult := (input(n)*coef(24-n));
                    -- and accumulate to output
                    y_out := y_out + mult(31 downto 15);
                end loop;
                y <= std_logic_vector(y_out);
            end if;
    end process;
end filter;
```

Results

Testbench Code & Simulation

Below is the testbench code we wrote to verify the functionality of the 25-tap FIR filter.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use STD.textio.all;
use ieee.std_logic_textio.all;

entity g64_lab3_tb is
end g64_lab3_tb;

architecture test of g64_lab3_tb is

-- Declare the Component Under Test
component g64_lab3 is
    port(x  : in std_logic_vector(15 downto 0);
        clk : in std_logic;
        rst : in std_logic;
        y   : out std_logic_vector(16 downto 0));
end component g64_lab3;

-- Testbench Internal Signals
file file_VECTORS_in    : text;
file file_RESULTS       : text;

constant clk_PERIOD : time := 100 ns;

signal x_in         : std_logic_vector(15 downto 0);
signal clk_in       : std_logic;
signal rst_in       : std_logic;
signal y_out        : std_logic_vector(16 downto 0);

begin
    -- Instantiate MAC
    g64_lab3_INST   : g64_lab3
        port map (
            x => x_in,
            clk => clk_in,
            rst => rst_in,
            y => y_out
        );

    -- Clock generation
    clk_generation  : process
    begin
        clk_in <= '1';
        wait for clk_PERIOD/2;
```

```
        clk_in <= '0';
        wait for clk_PERIOD/2;
    end process clk_generation;

    -- Feeding Inputs
    feeding_instr   : process is
        variable v_Iline : line; -- will hold the input line read from the input file
        variable v_Oline  : line; -- will hold the output line to be written to output file
        variable v_in     : std_logic_vector(15 downto 0); -- input value
    begin
        -- reset the circuit
        rst_in  <= '1';
        wait until rising_edge(clk_in);
        wait until rising_edge(clk_in);
        rst_in <= '0';
        file_open(file_VECTORS_in, "lab3-in-fixed-point.txt", read_mode);
        file_open(file_RESULTS, "lab3-out.txt", write_mode);

        while not endfile(file_VECTORS_in) loop
            -- read from the file, line-by-line until EOF,
            -- storing the line in v_Iline, then reading the value into v_in
            readline(file_VECTORS_in, v_Iline);
            read(v_Iline, v_in);

            x_in <= v_in; -- map v_in to x

            wait until rising_edge(clk_in);
        end loop;

        -- write the filtered value to the output file and map it to y
        write(v_Oline, y_out);
        writeline(file_RESULTS, v_Oline);
        wait;

    end process;
end architecture test;
```

Running the simulation, it was verified that the 25-tap FIR filter functioned as expected, as seen in Fig. 2 below.

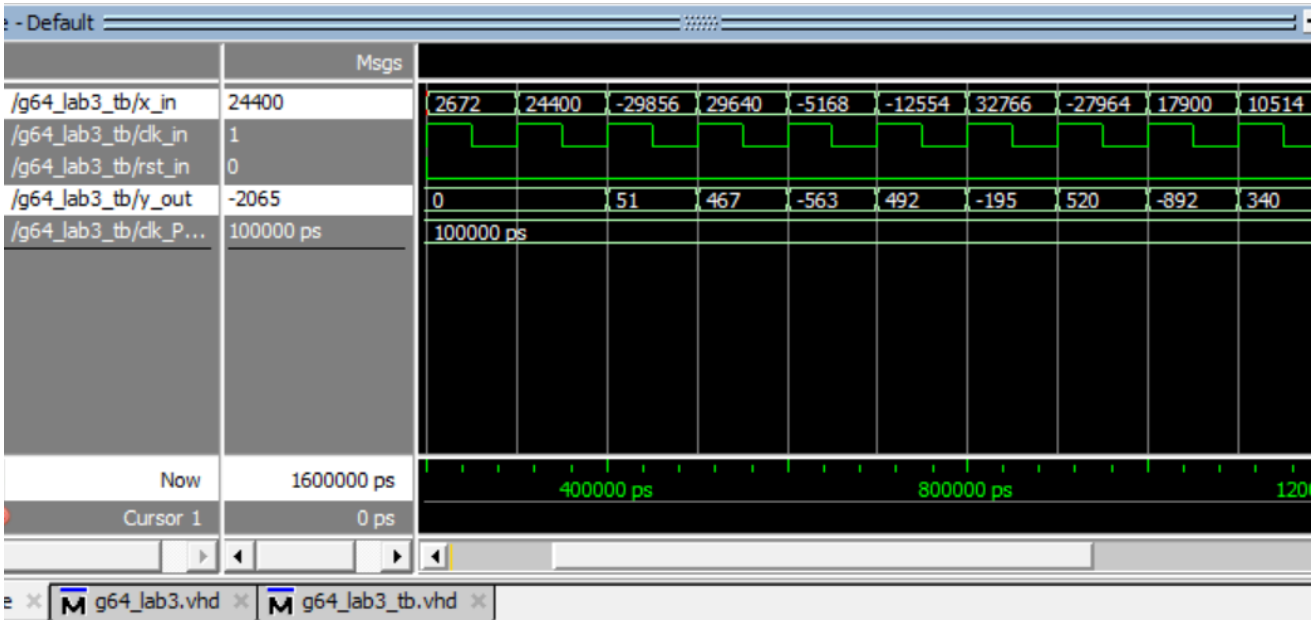


Fig. 2 Testbench Simulation of the 25-tap FIR filter using ModelSim

Resource Utilization

Compilation Report & Chip Planner

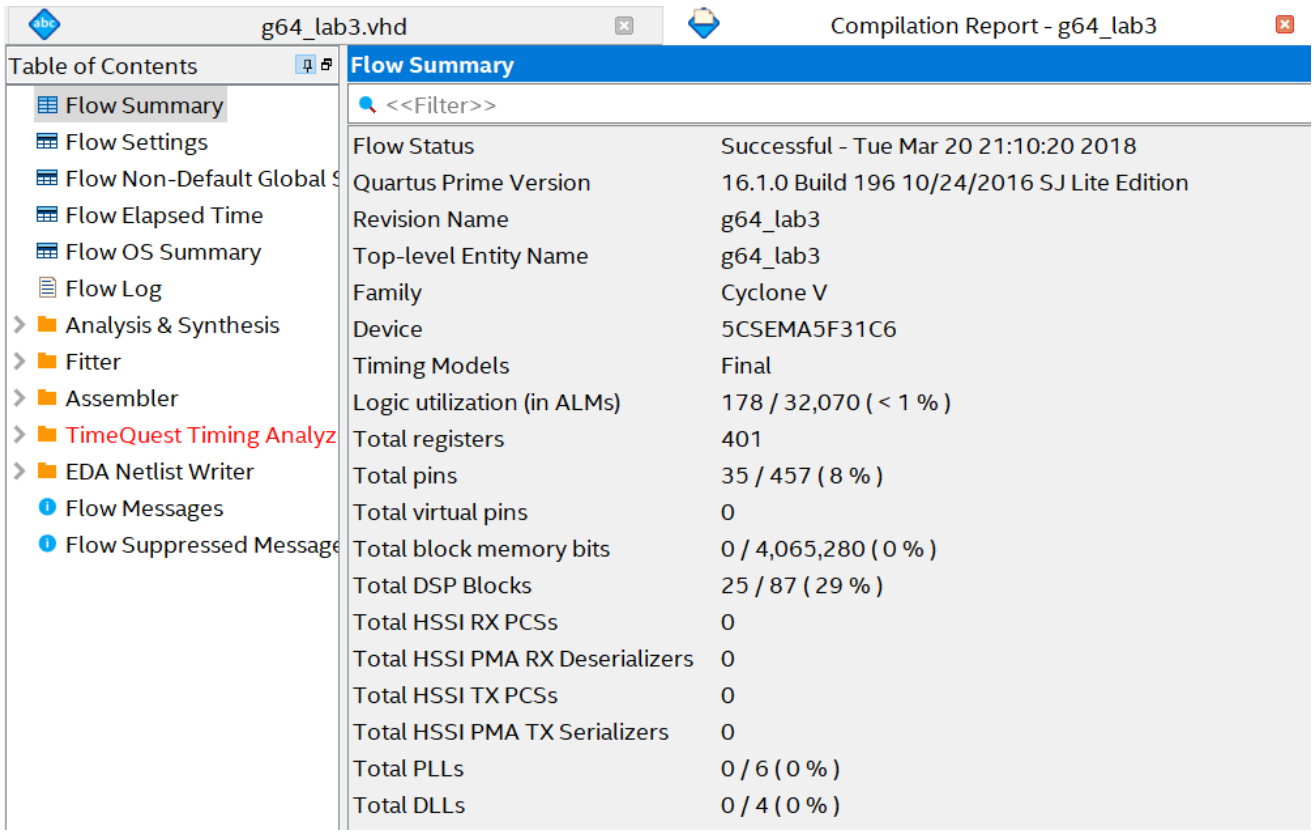


Fig. 3 Compilation Report: Flow Summary

From the flow summary we can see that 401 registers are used in the implementation. The FIR filter is more resource-inefficient than the MAC unit and the bit counter implemented in the previous labs. As the order of the FIR filter increases, i.e. number of taps increases, more resources will be required to realize the design as arrays are used in the VHDL code to temporarily store the convolution from the taps. The Chip Planner below (*Fig. 4*) shows the used resources in dark blue. Much of the resources are still unused and therefore this design could have been implemented on a smaller chip.

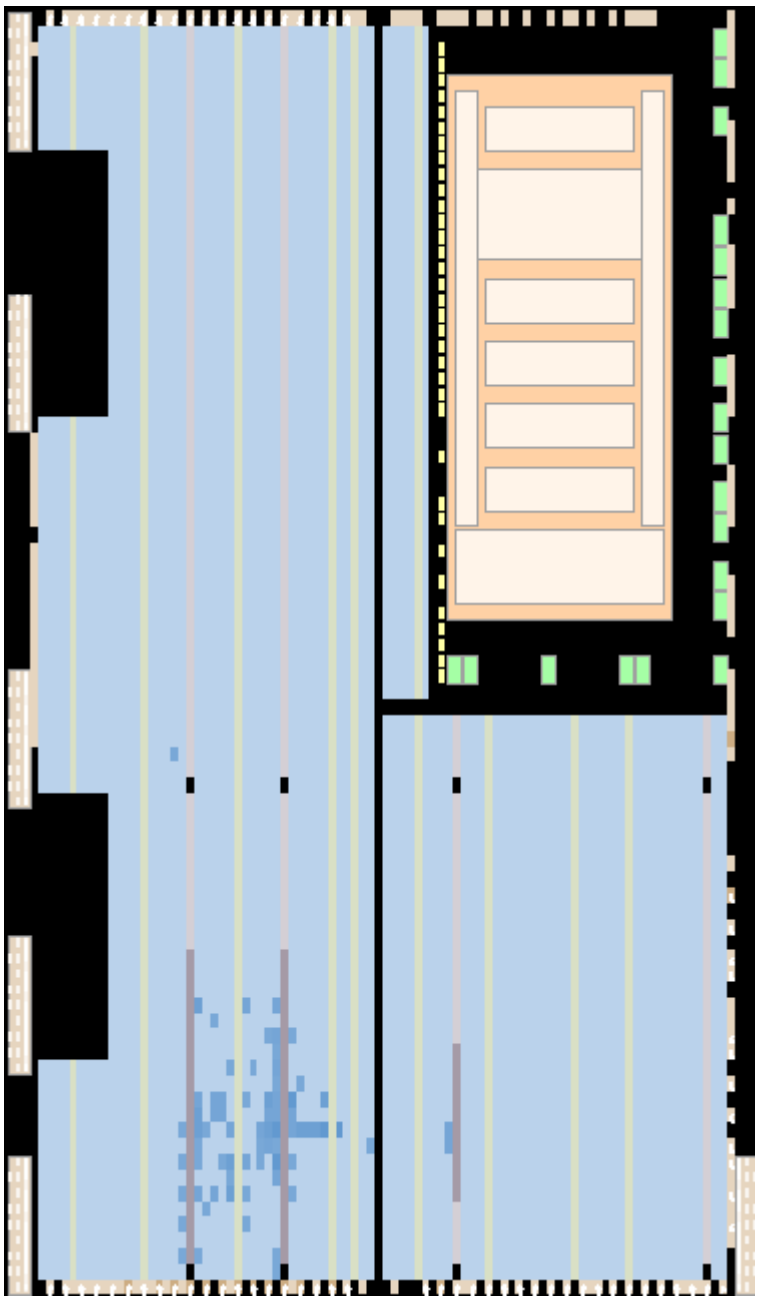


Fig. 4 Chip Planner

RTL View

Fig. 5 below is the Register Transfer Level (RTL) view of the 25-tap FIR filter.

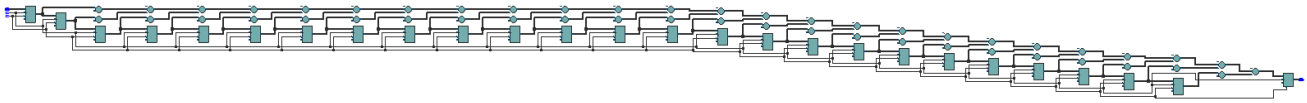


Fig. 5 RTL View of the Circuit

Fig. 6 and Fig. 7 below take a closer look at the RTL view.

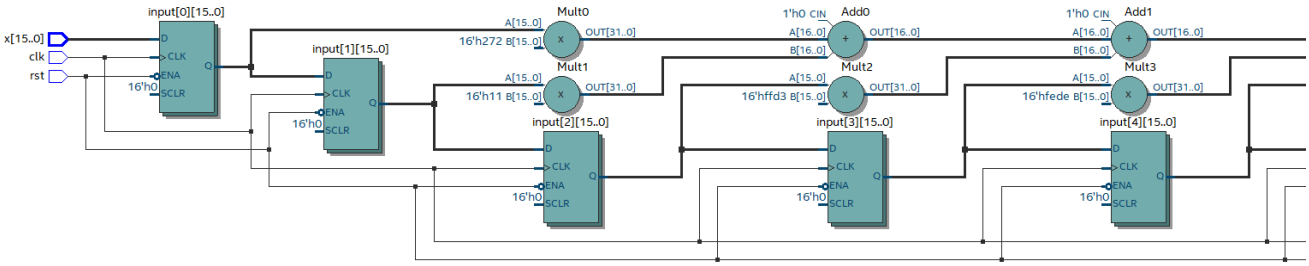


Fig. 6 Zoomed in RTL View of the Beginning of the Circuit

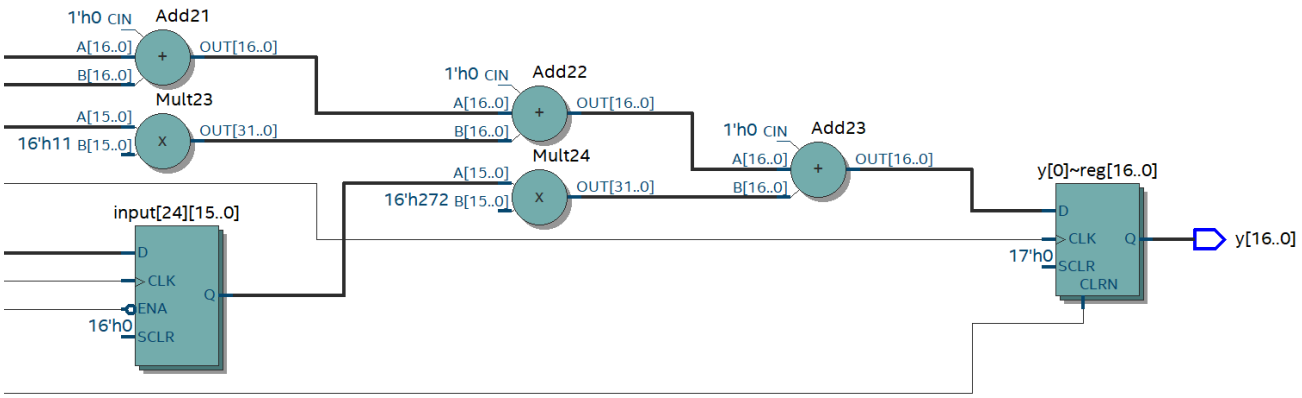


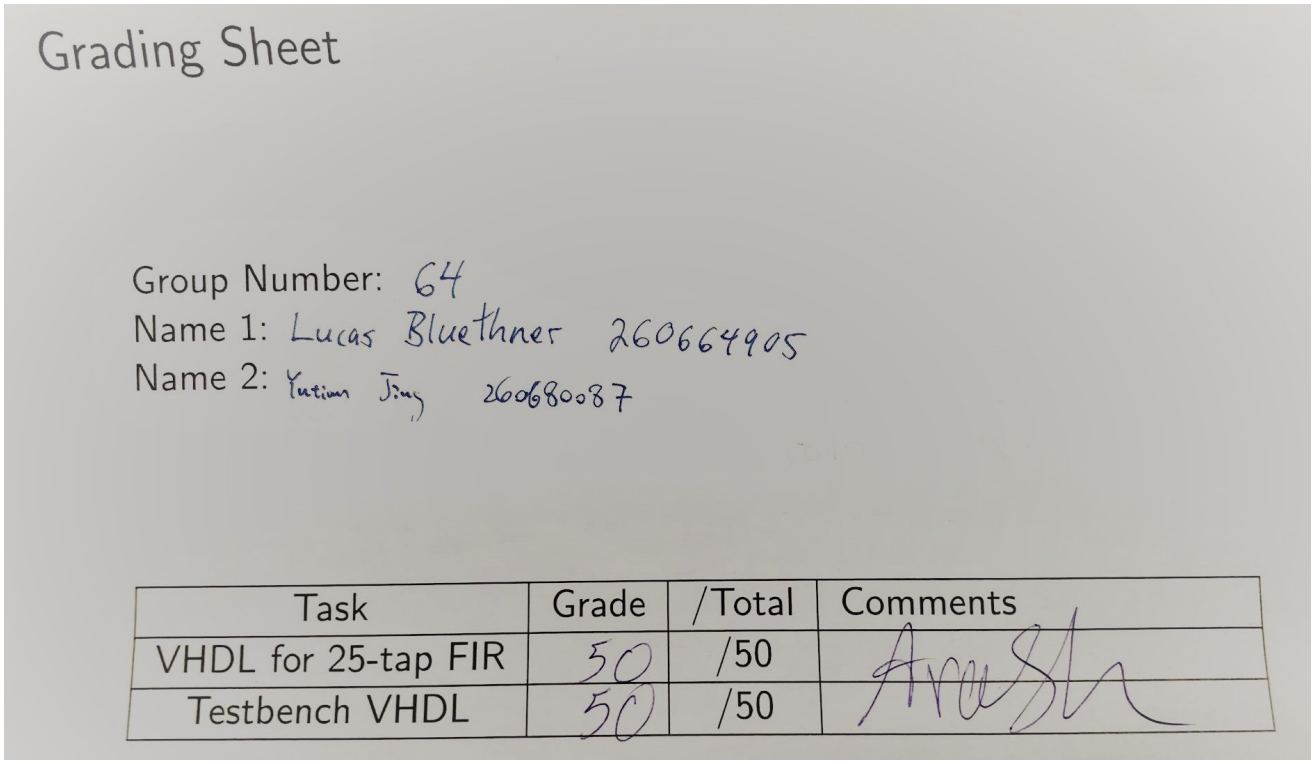
Fig. 7 Zoomed in RTL View of the End of the Circuit

Comparing the RTL view to the direct form of the FIR filter (Fig. 1) verifies that our VHDL design is indeed implementing the filter in the direct form of the filter. Registers, adders and multipliers are the only resources used to implement the filter.

Conclusion

As stated in the introduction, the purpose of this lab was to learn how to implement digital filters using VHDL, and to verify the design by performing testbench simulations using ModelSim. More specifically, we implemented a 25-tap Finite Impulse Response (FIR) and verified its behaviour by running a testbench simulation in ModelSim. Fortunately, we did not run into any problems during this lab and our FIR filter designed in VHDL output the expected filtered values.

Grading Sheet



Grading Sheet

Group Number: 64
Name 1: Lucas Bluethner 260664905
Name 2: Yutian Jing 260680087

Task	Grade	/Total	Comments
VHDL for 25-tap FIR	50	/50	
Testbench VHDL	50	/50	

Fig. 8 Grading Sheet