ECSE 325

Lab 4 Report

Lucas Bluethner, 260664905

Yutian Jing, 260680087

March 30, 2018

Introduction

The purpose of this lab was to learn how to specify timing constraints and perform static timing analysis of a synthesized circuit using TimeQuest timing analyzer. In the previous lab, we implemented a 25-tap FIR filter in direct form (*Fig. 1*). In this lab, we implemented the 25-tap FIR filter in broadcasting (direct-transpose) form (*Fig. 2*), using the same input files from the previous labs to verify our design. To verify the functionality of our design, we wrote testbench code to read a given test vector and obtain the output test vector. We then compared these results with the previous labs output results (the expected output signal).

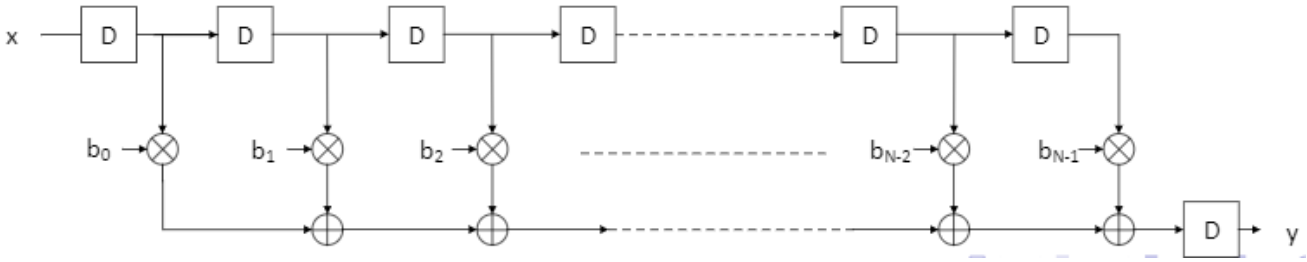


Fig. 1 Direct form of an N-tap FIR filter

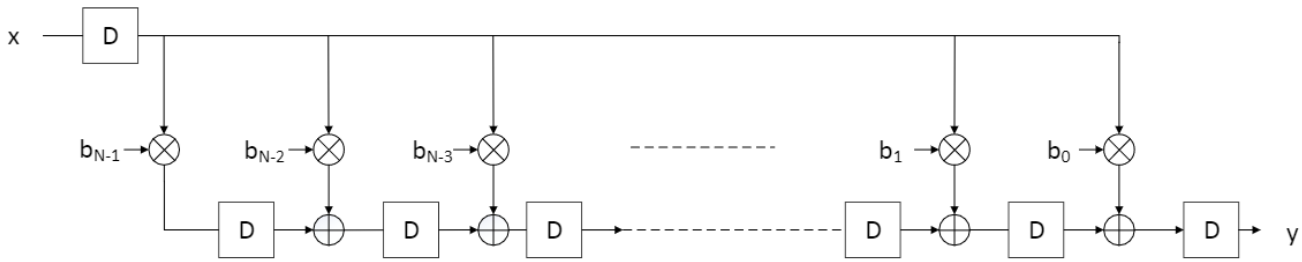


Fig. 2 Broadcasting (direct-transpose) form of an N-tap FIR filter

Code Implementation

The difference between the direct form of the FIR filter and the broadcasting form is not a huge difference. The direct form gets its output buy multiplying each of the 25 most recent input signals by the corresponding weight, and then adds them all together. Whereas the broadcasting form has registers between the adders to perform many small additions separated by delay elements.

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

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

entity g64_lab4 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_lab4;

architecture filter of g64_lab4 is

type COEFS is array(0 to 24) of signed(15 downto 0); -- array for coefficients
signal coef : COEFS;
type OUTPUTS is array(0 to 24) of signed(16 downto 0); -- array for intermediate outputs
signal x_value : OUTPUTS := (others => "00000000000000000");

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 the multiplication of x and a weight
        variable mult  : signed(31 downto 0) := (others => '0');
        begin
            if (rst = '1') then -- if reset is high, reset everything to zero
                y <= (others => '0');
                mult := (others => '0');
                x_value <= (others => "00000000000000000");
            elsif (rising_edge(clk)) then
                 -- loop through and multiply the input by each coeffiecient
                for i in 0 to 23 loop
                    -- the new value is the multiplication,
                    -- plus the previous input signal multiplied by the coefficient
                    mult := signed(x)*coef(i);
                    x_value(i) <= mult(31 downto 15) + x_value(i+1);
                end loop;
                -- last value doesn't need the addition b/c there is no 26th element
                mult := signed(x)*coef(24);
                x_value(24) <= mult(31 downto 15);
                -- the value at index zero is the filtered output
                y <= std_logic_vector(x_value(0));
            end if;
    end process;
end filter;
```

Results

Testbench Code & Simulation

Below is the testbench code we wrote to verify the functionality of the broadcasting 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_lab4_tb is
end g64_lab4_tb;

architecture test of g64_lab4_tb is

-- Declare the Component Under Test
component g64_lab4 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_lab4;

-- 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_lab4_INST   : g64_lab4
        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_Iline1   : line; -- will hold one line read from the input file
        variable v_Oline    : line; -- will hold one line of the output file
        variable v_in       : std_logic_vector(15 downto 0); -- variable for the input value
    begin
        -- reset the circuit
        rst_in  <= '1';
        wait until rising_edge(clk_in);
        wait until rising_edge(clk_in);
        rst_in <= '0';
        -- open the input to be read from, and the output file to be written to
        file_open(file_VECTORS_in, "lab3-in-fixed-point.txt", read_mode);
        file_open(file_RESULTS, "lab3-out.txt", write_mode);

        -- read line-by-line (one input value at a time) until EOF
        while not endfile(file_VECTORS_in) loop
            -- read a line from the input file and store the value in v_in
            readline(file_VECTORS_in, v_Iline1);
            read(v_Iline1, v_in);
            -- map the input value to x_in
            x_in <= v_in;

            wait until rising_edge(clk_in);
        end loop;
        -- map the output value to y_out and write it on one line of the output file
        write(v_Oline, y_out);
        writeline(file_RESULTS, v_Oline);
        wait;

    end process;
end architecture test;
```

Running the simulation, it was verified that the broadcasting FIR filter functioned as expected, as seen in *Fig. 3* below. This matches exactly the output values we obtained from the previous lab.

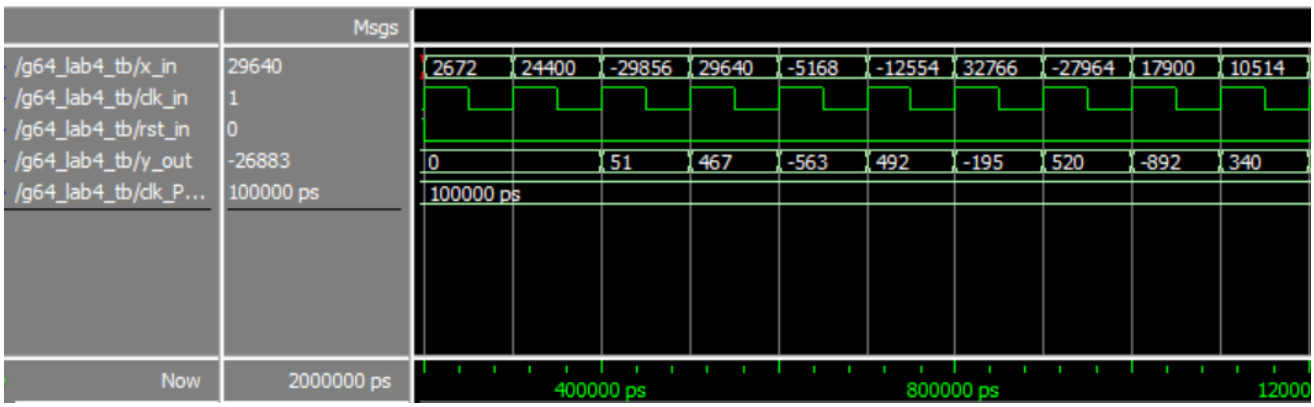


Fig. 3 Testbench Simulation of the 25-tap FIR filter in broadcasting form

Resource Utilization

Compilation Report & Chip Planner

Flow Summary	
<<Filter>>	
Flow Status	Successful - Thu Mar 29 17:11:10 2018
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	g64_lab4
Top-level Entity Name	g64_lab4
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	205 / 32,070 (< 1 %)
Total registers	441
Total pins	35 / 457 (8 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	13 / 87 (15 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Fig. 4 Compilation Report: Flow Summary

From the flow summary we can see that 441 registers were used in the implementation. As the order of the FIR filter increases (the number of taps increases) more resources will be required to realize the design because more registers, adder and multipliers will be required. The Chip Planner below (*Fig. 5*) shows the used resources in dark blue. Much of the resources are still unused and therefore the 25-tap broadcasting FIR filter could have been implemented on a much smaller chip.

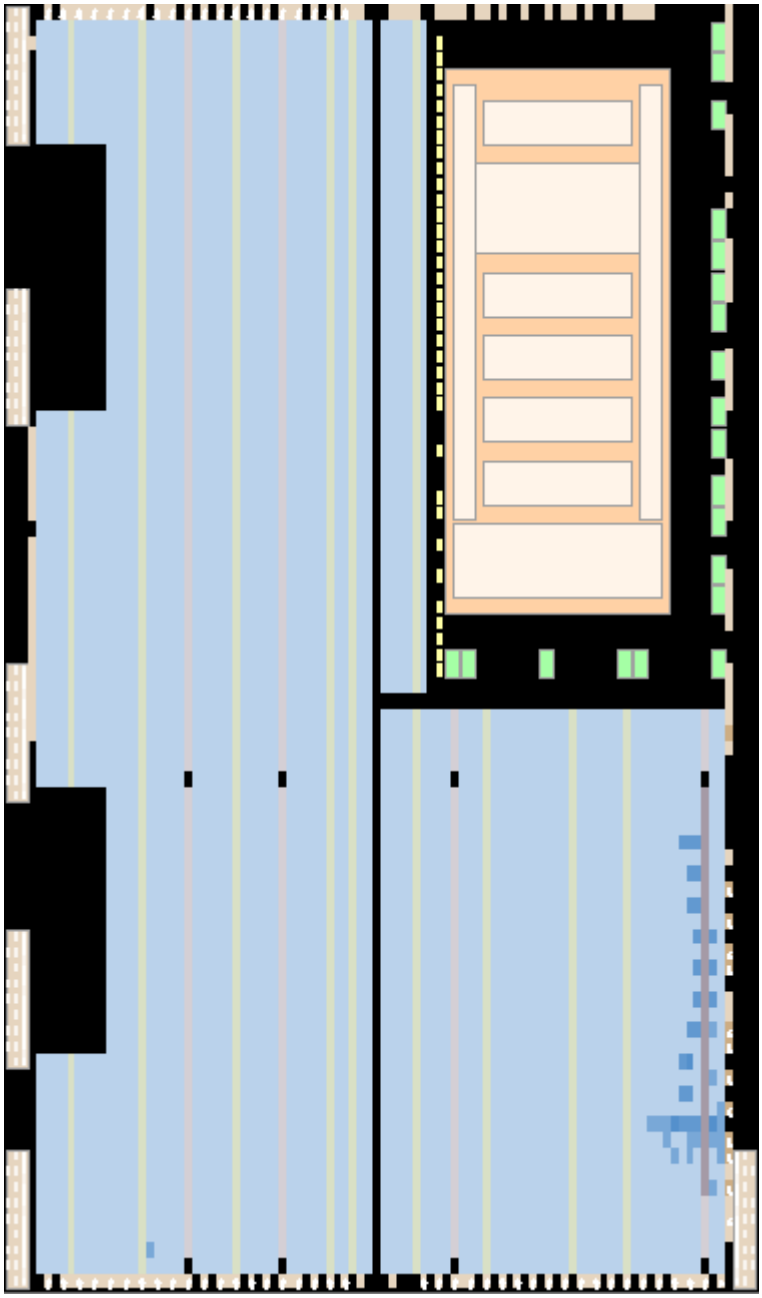


Fig. 5 Chip Planner

RTL View

The RTL view of the circuit is a bit messy (*Fig. 6*), but if you take a closer look (*Fig. 7*) you can begin to compare it with the diagram in *Fig. 2*.

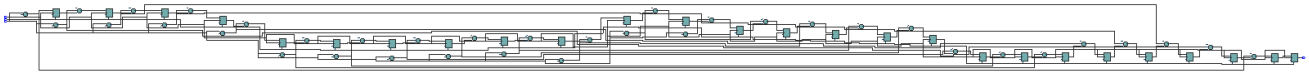


Fig. 6 RTL View of the circuit

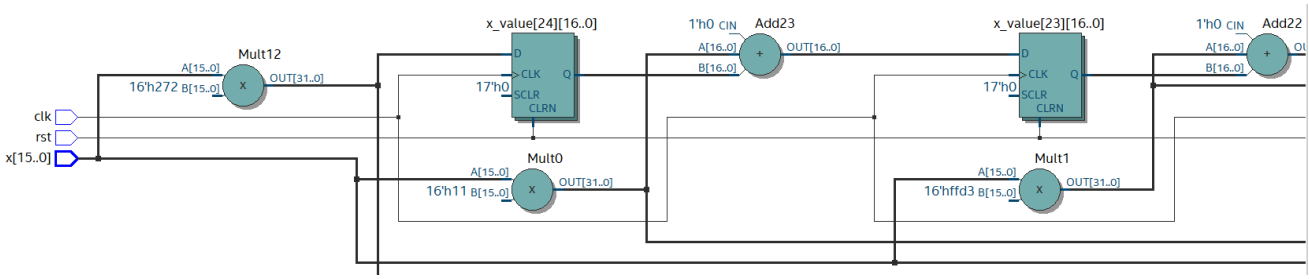


Fig. 7 Closer look at the RTL View of the circuit

Just like the diagram shown in *Fig. 2*, the RTL View shows the the input signal is "broadcasted" to each multiplier, and each adder has a register in between it, as we expected.

TimeQuest Timing Analysis

To ensure a properly working circuit, we also took into consideration timing constraints. A Synopsys Design Constraints (.sdc) file is what is used to specify timing constraints. Thus, we specified the timing constraints did by adding a file to the project called ***g_64_testbed.sdc***. The file contained one line of code:

```
create_clock -period 20 [get_ports clk]
```

This constrains the clock port with a 20 ns clock period requirement.

After recompiling our design with the timing constraint applied, we looked at the timing summaries for the Slow Model and Fast Model. Specifically, the Fmax Summary (*Fig. 8*), which gives the maximum clock speed of 325.52 MHz, and the Setup (*Fig. 9*) and Hold (*Fig. 10*) summaries, which give the slack amounts for the setup and hold constraints. Slack is the margin by which the required timing requirement is met or not. It is the difference between the required time and the arrival time. A positive slack value indicates the margin by which a requirement was met, whereas a negative slack value indicates the margin by which a requirement was not met. Thus, since both the Setup and Hold summaries show a positive slack value, our design met the requirement of a 20 ns clock period.

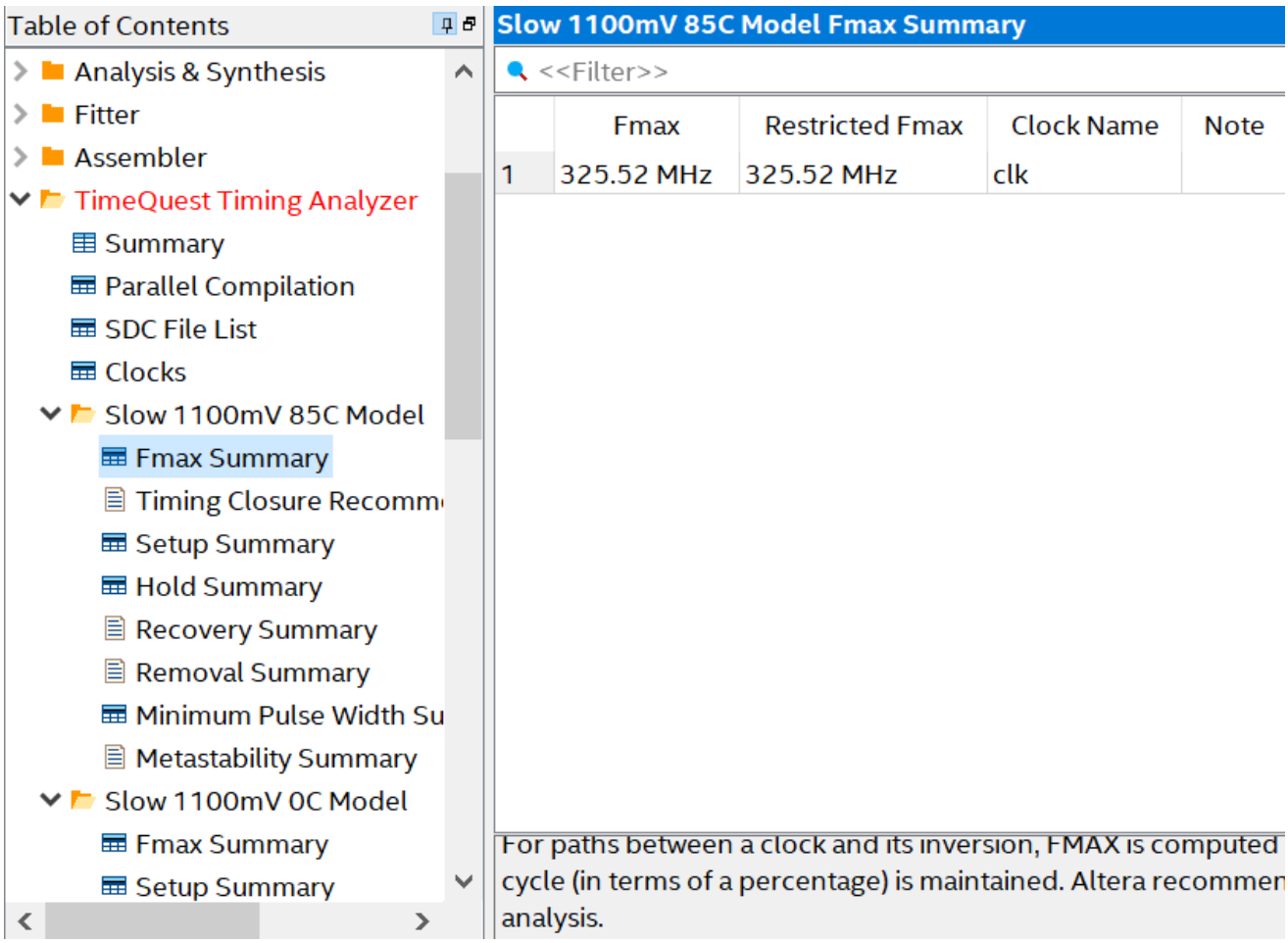


Fig. 8 Fmax Summary

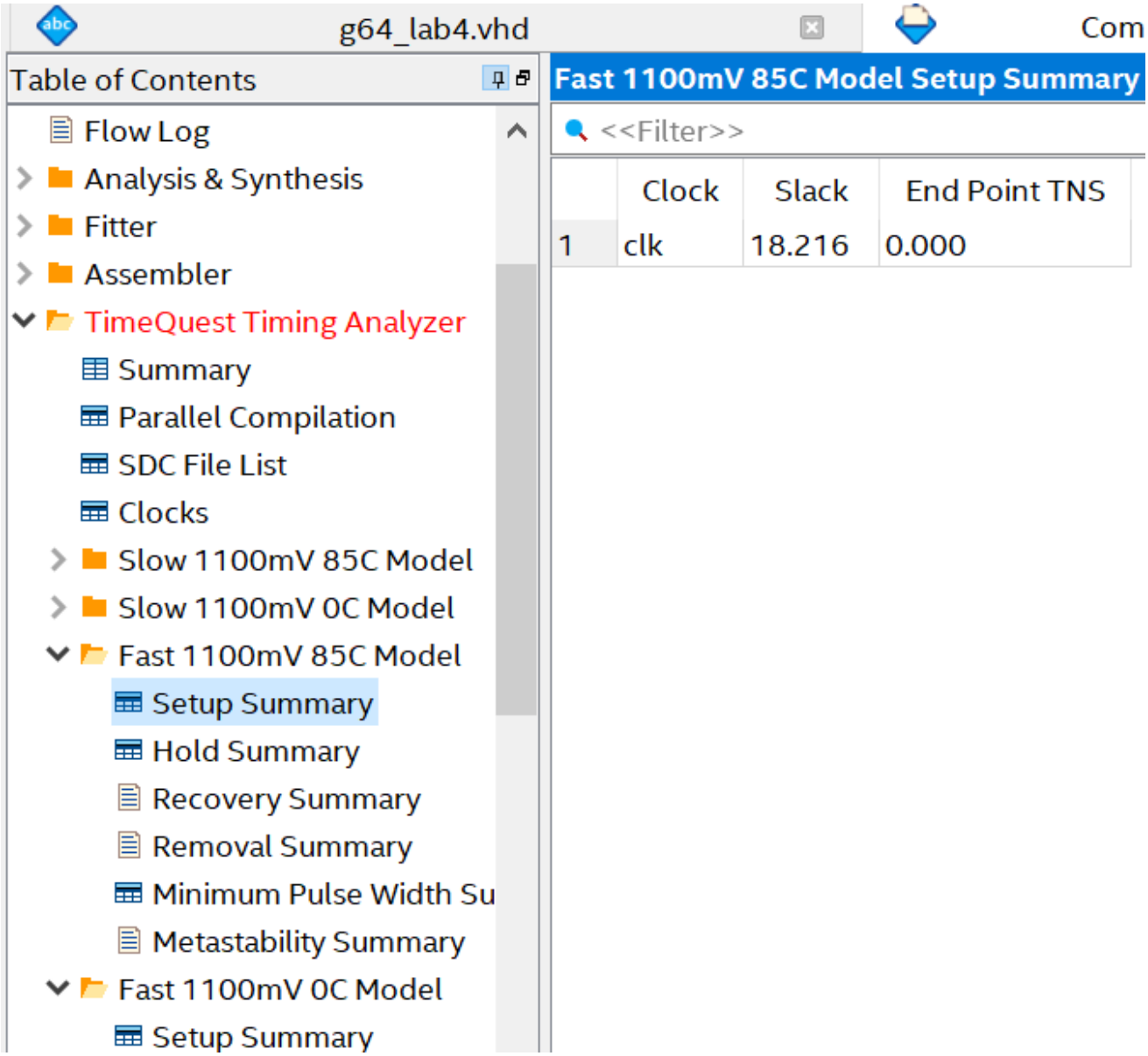


Fig. 9 Setup Summary

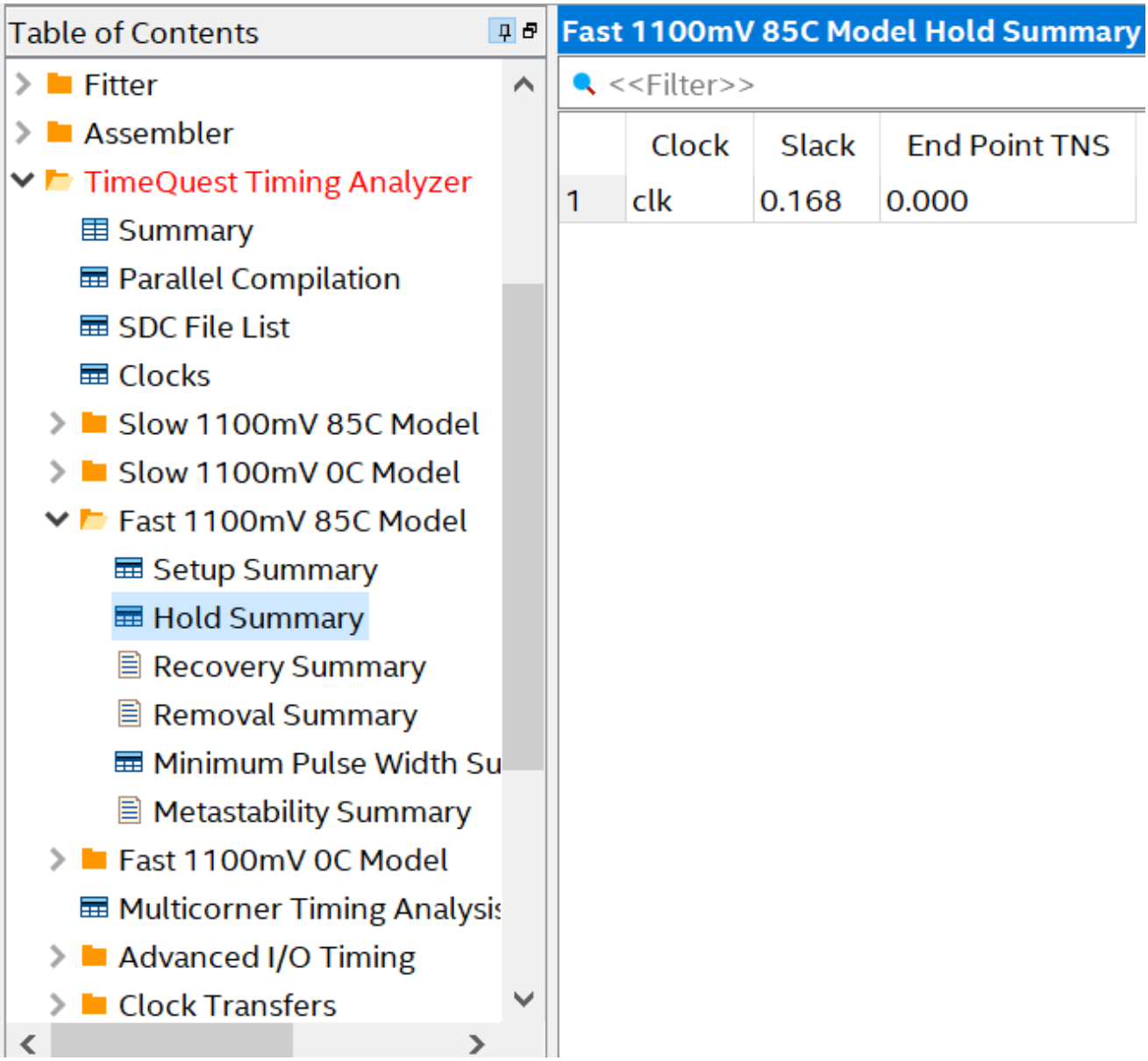
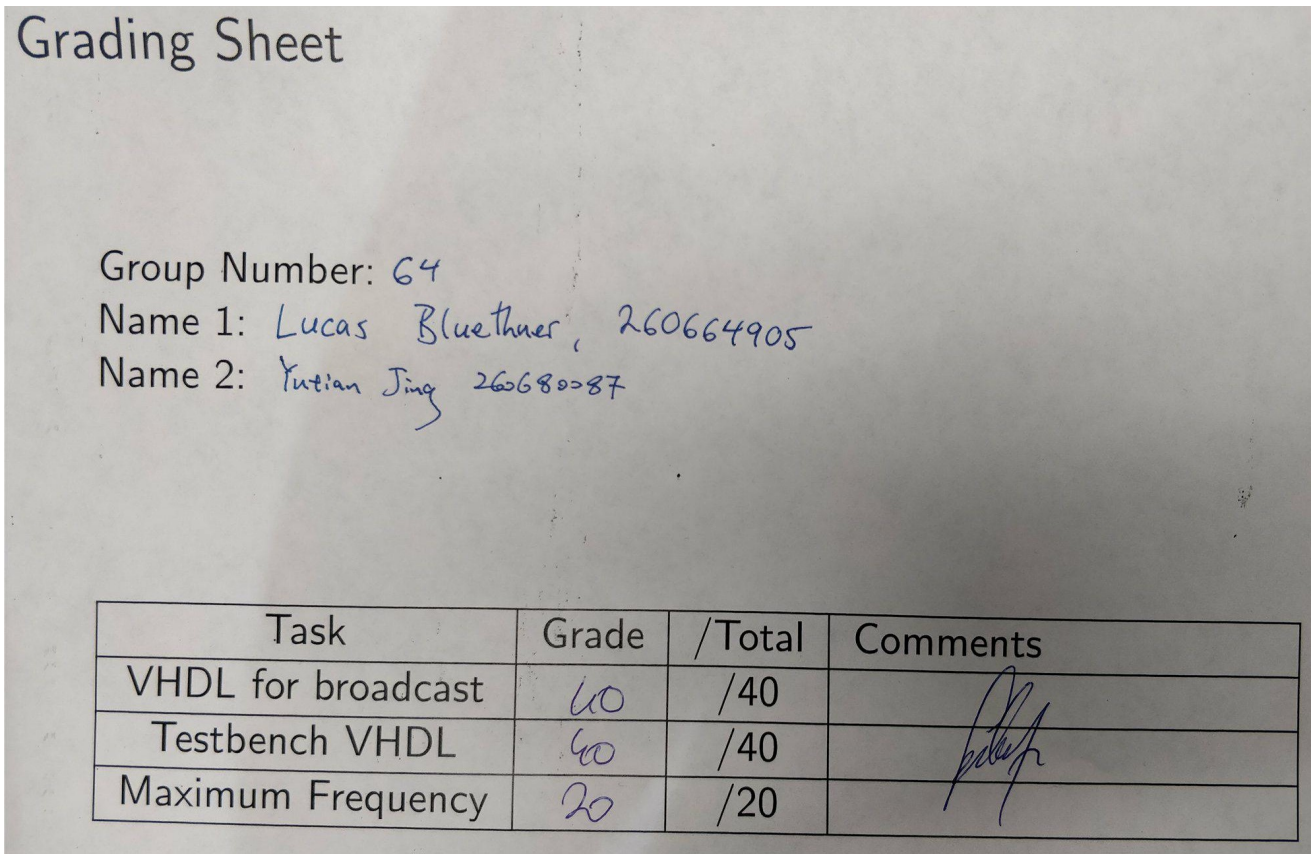


Fig. 10 Hold Summary

Conclusion

The purpose of this lab was to learn how to specify timing constraints and perform static timing analysis of a synthesized circuit using TimeQuest timing analyzer. We did this by implementing a 25-tap FIR filter in broadcasting (direct-transpose) form in VDHL. We then wrote testbench code to verify the functionality of our design which was successful because our output matched the results obtained from the previous lab, the expected outputs. Next, we constrained the clock period of our design to 20 ns and recompiled our design. Analyzing the Setup and Hold Summaries, we found that our design successfully met the timing constraints because all of the slack values were positive values.

Grading Sheet



Grading Sheet

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

Task	Grade	/Total	Comments
VHDL for broadcast	40	/40	
Testbench VHDL	40	/40	
Maximum Frequency	20	/20	

Fig. 8 Grading Sheet