Allan Reuben (260783324)
Noah Zwack (260769910)
Group 10

ECSE 325 Lab 3 Report:
Timing Constraint Specification and Timing Analysis using TimeQuest

In this lab, we were asked to implement a bandpass Finite Impulse Response (FIR) filter. We were given a block diagram on how the device should operate, and it was left up to us how to implement it. In this report, we shall discuss our implementation of the FIR filter and its associated testbench. We shall also discuss the timing analysis that was done on this circuit to ensure it was operating as expected, as well as an alternate implementation of the FIR filter. Throughout this lab, the content of the sdc file was “create_clock -period 20 [get_ports clk]” , just as specified in the lab document.

FIR Filter (1st implementation)

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

-- Entity Declaration
entity g10_FIR 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 g10_FIR;

architecture a0 of g10_FIR is

-- CONSTANTS
constant taps : integer := 25; -- Number of weights/taps

-- TYPES OF ARRAYS
type t_weights is array (0 to taps - 1) of signed(15 downto 0); -- For each tap
type t_pipeline is array (0 to taps - 1) of std_logic_vector(15 downto 0); -- For
the pipeline
type t_products is array (0 to taps - 1) of signed(31 downto 0); -- For storing the
multiplication of the pipeline value and the weight

-- FUNCTIONS
```

```
-- Initializes the weights array to the appropriate values
function init_weights
	return t_weights is
variable temp_weights : t_weights;
begin
	temp_weights(0) := "0000001001110010";
	temp_weights(1) := "0000000000010001";
	temp_weights(2) := "1111111111010011";
	temp_weights(3) := "1111111011011110";
	temp_weights(4) := "0000001100011001";
	temp_weights(5) := "1111110110100111";
	temp_weights(6) := "1111110000001110";
	temp_weights(7) := "0000110110111100";
	temp_weights(8) := "1110110001110011";
	temp_weights(9) := "0000110111110111";
	temp_weights(10) := "0000001100000111";
	temp_weights(11) := "1110101000001010";
	temp_weights(12) := "0001111000110011";
	temp_weights(13) := "1110101000001010";
	temp_weights(14) := "0000001100000111";
	temp_weights(15) := "0000110111110111";
	temp_weights(16) := "1110110001110011";
	temp_weights(17) := "0000110110111100";
	temp_weights(18) := "1111110000001110";
	temp_weights(19) := "1111110110100111";
	temp_weights(20) := "0000001100011001";
	temp_weights(21) := "1111111011011110";
	temp_weights(22) := "1111111111010011";
	temp_weights(23) := "0000000000010001";
	temp_weights(24) := "0000001001110010";
	return temp_weights;
end function init_weights;

-- Rounds a 32-bit input to a 17-bit output value
function round (input : signed(31 downto 0))
	return signed is
variable rounded : signed(16 downto 0); -- The rounded value
begin
	-- Copy the 17 most significant bits from the input to the rounded value
	rounded := input(31 downto 15);
	-- If the next most significant bit from the input is '1', round up.
Otherwise, round down.
	if (input(14) = '1') then
```

```
            if (input(31) = '0') then -- If number is positive, add 1. If
negative, subtract 1.
                  rounded := rounded + 1;
            else
                  rounded := rounded - 1;
            end if;
      end if;
      return rounded;
end function round;

-- ARRAYS
signal r_weights : t_weights;
signal r_pipeline : t_pipeline := (others => (others => '0'));
signal r_products : t_products := (others => (others => '0'));

begin
      -- Map the empty array of weights to their actual values using the function
init_weights
      r_weights <= init_weights;

      -- Main process
      process (rst, clk)
      variable sum_result : signed(31 downto 0) := (others => '0'); -- Temporarily
stores the result of the sum
      begin
            -- Asynchronous reset: setting the pipeline and the output to all '0's
            if (rst = '1') then
                  r_pipeline <= (others => (others => '0'));
                  y <= (others => '0');
            elsif (rising_edge(clk)) then
                  r_pipeline <= (x & r_pipeline(0 to taps - 2)); -- Take the new
input and shift it into the pipeline
                  sum_result := (others => '0'); -- Reset the sum variable
                  for i in 0 to taps - 1 loop
                        sum_result := sum_result + r_products(i); -- Sum the
products produced by each tap
                  end loop;
                  y <= std_logic_vector(round(sum_result)); -- Round the sum
before setting it to the input
            end if;
      end process;

      -- Generate statement for the products
      products: for i in 0 to taps - 1 generate
```

```
            -- Multiply each pipeline value with its corresponding weight
            r_products(i) <= (signed(r_pipeline(i)) * r_weights(i));
        end generate;

end a0;
```

Figure 1: Code for g10_FIR.vhd

Our implementation for the first description of the FIR filter is shown in the figure above. In our program, we have three arrays: one that holds the values of the weights, one that holds the values of the pipeline, and one that holds the products of the two previous arrays. A function is used to populate the array of weights, and a generate statement continuously populates the array of products when a new value is shifted into the pipeline. In the process block, there's an asynchronous reset portion that is checked first. If reset is not high, it will shift the next input into the pipeline and then add up everything in the array of products and store it in a temporary variable. It then rounds the final answer to the appropriate number of bits before feeding it to the output. The rounding function simply takes the first 17 bits, then adds or subtracts '1', for a positive or negative number respectively, if the next significant bit is 1.

The resource utilization of this implementation of the FIR filter can be seen below.

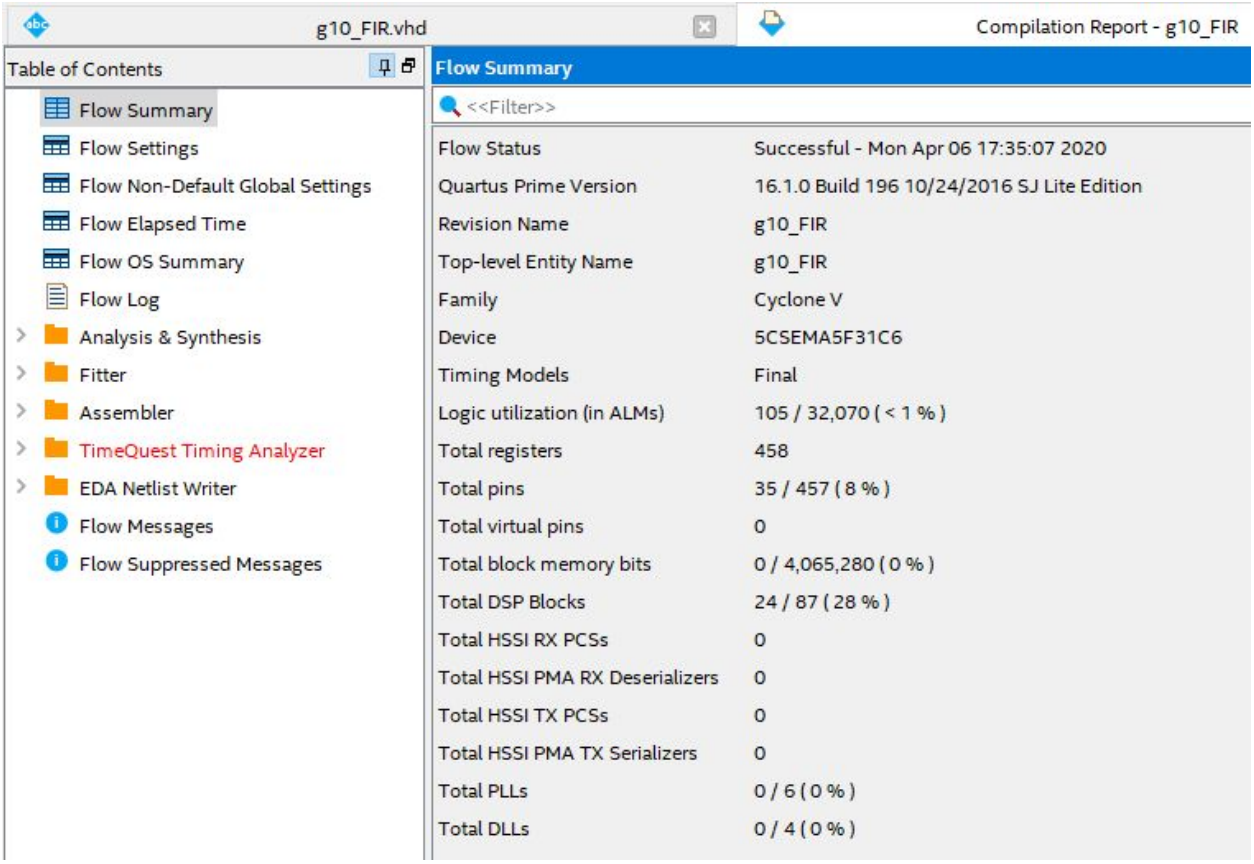


Figure 2: Resource utilization of the first FIR implementation

FIR Filter Testbench

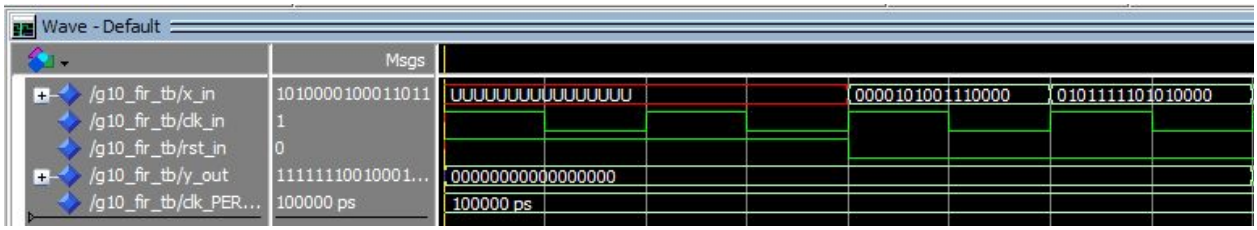
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_textio.all;
use STD.textio.all;

entity g10_FIR_tb is
end g10_FIR_tb;

architecture test of g10_FIR_tb is
--------------------------------------------------------------------------------
-- Declare the Component Under Test
--------------------------------------------------------------------------------
component g10_FIR 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 g10_FIR;

--------------------------------------------------------------------------------
-- Testbench Internal Signals
--------------------------------------------------------------------------------
file file_VECTORS_X : 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 FIR

      g10_FIR_INST : g10_FIR
      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;

-------------------------------------------------------------------------------
	-- Providing Inputs
-------------------------------------------------------------------------------
	feeding_instr : process

	variable v_Iline : line;
	variable v_Oline : line;
	variable v_x_in : std_logic_vector(15 downto 0);

	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 file and create an output file
		file_open(file_VECTORS_X, "lab3-in-fixed-point.txt", read_mode);
		file_open(file_RESULTS, "lab3-out.txt", write_mode);
		-- Feed data into the design
		while not endfile(file_VECTORS_X) loop
			readline(file_VECTORS_X, v_Iline);
			read(v_Iline, v_x_in);
			x_in <= v_x_in;
			write(v_Oline, y_out);
			writeline(file_RESULTS, v_Oline);
			wait until rising_edge(clk_in);
		end loop;
		wait;
	end process;
end test;
```

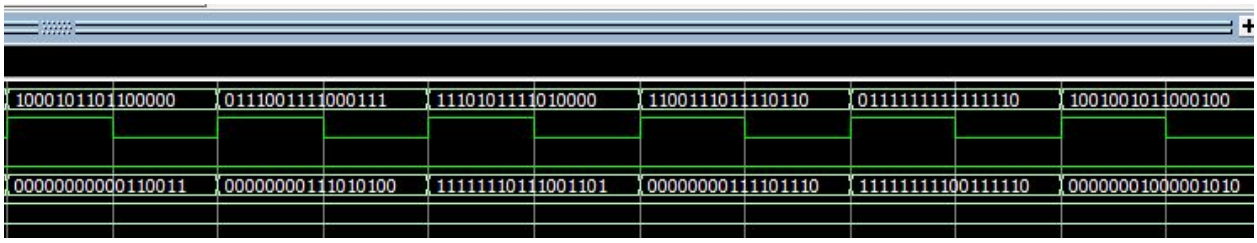
Figure 3: Code for g10_FIR_tb.vhd

Our testbench for testing our code is shown in the figure above. We begin by declaring our component under test, creating internal signals, then mapping the internal signals to an instance of our component. We have a process for generating the clock, followed by the process for feeding in the inputs. In this process, we read the pre-processed input data from a text file, feed it to the component, and then write each output value to a file.

A snippet of the simulation output is shown in the figure below.

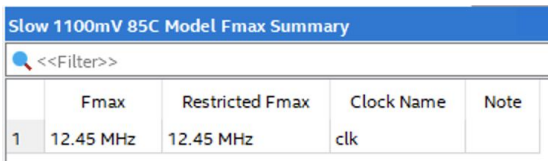


(a)



(b)

Figure 4: A sample output when simulating the testbench. The rightmost edge of (a) and the leftmost edge of (b) should be connected, but were separated for easier visibility



Aggregate Results [hide details]

	Issue	Category	Paths Affected
1	DSP Register Packing	HDL	20
2	Long Combinational Path	HDL	20

Top Recommendations [hide details]

★★★★★ DSP block Add0~8 is not fully utilizing internal DSP register banks. Design performance may be limited. for the path from **r_pipeline[1][0]** to **y[0]~reg0** [show details]

★★★★★ Reduce the levels of combinational logic for the path from **r_pipeline[1][0]** to **y[0]~reg0** [show details]

★★★★★ DSP block Add0~8 is not fully utilizing internal DSP register banks. Design performance may be limited. for the path from **r_pipeline[1][0]** to **y[0]~reg0** [show details]

★★★★★ Reduce the levels of combinational logic for the path from **r_pipeline[1][0]** to **y[0]~reg0** [show details]

★★★★★ DSP block Add0~8 is not fully utilizing internal DSP register banks. Design performance may be limited. for the path from **r_pipeline[1][0]** to **y[0]~reg0** [show details]

★★★★★ Reduce the levels of combinational logic for the path from **r_pipeline[1][0]** to **y[0]~reg0** [show details]

★★★★★ DSP block Add0~8 is not fully utilizing internal DSP register banks. Design performance may be limited. for the path from **r_pipeline[1][0]** to **y[0]~reg0** [show details]

★★★★★ Reduce the levels of combinational logic for the path from **r_pipeline[1][0]** to **y[0]~reg0** [show details]

★★★★★ DSP block Add0~8 is not fully utilizing internal DSP register banks. Design performance may be limited. for the path from **r_pipeline[1][0]** to **y[0]~reg0** [show details]

★★★★★ Reduce the levels of combinational logic for the path from **r_pipeline[1][0]** to **y[0]~reg0** [show details]

Fast 1100mV 0C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	-41.259	-740.888

Fast 1100mV 85C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	-44.539	-830.555

Timing Closure Recommendations

Summary [hide details]

This design contains failing setup paths with a worst-case slack of -79.346 ns. Run Rep
For recommendations for any particular path, click the appropriate link in the table bel

Top Failing Paths [hide details]

	Slack	From	To	Recommendations
1	-79.346	r_pipeline[1][0]	y[0]~reg0	Report recommendations for this path
2	-79.346	r_pipeline[1][0]	y[0]~reg0	Report recommendations for this path
3	-79.346	r_pipeline[1][0]	y[0]~reg0	Report recommendations for this path
4	-79.346	r_pipeline[1][0]	y[0]~reg0	Report recommendations for this path
5	-79.346	r_pipeline[0][0]	y[0]~reg0	Report recommendations for this path

Slow 1100mV 85C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	-79.346	-1668.876

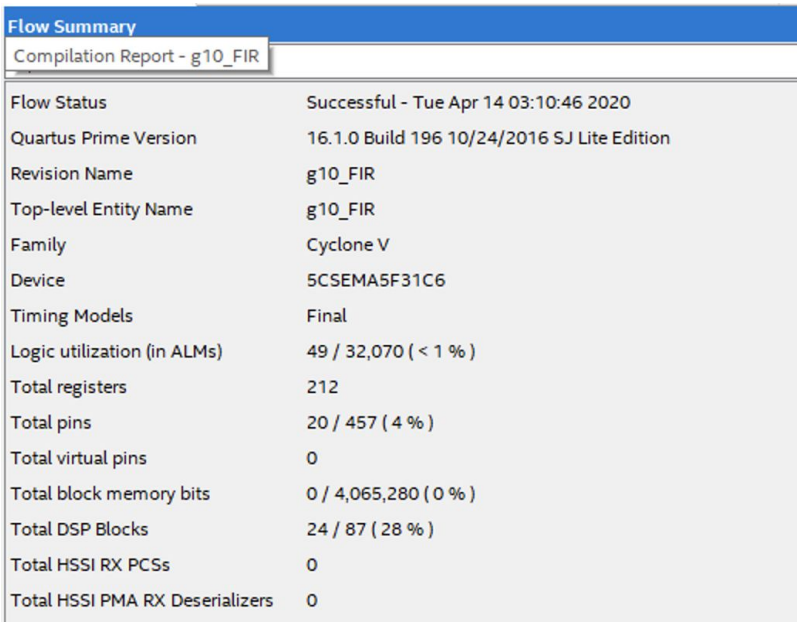
Figures 5-10: Max frequency and timing violations of the Direct FIR Implementation

Allan Reuben (260783324)

10

Noah Zwack (260769910)

Group 10

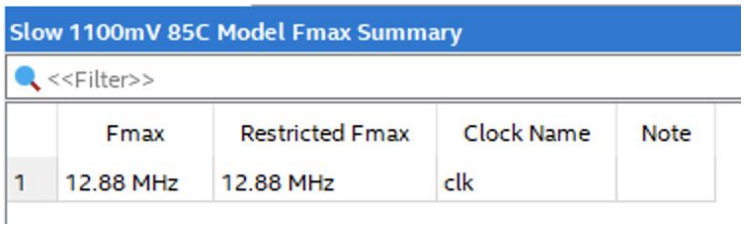


Flow Summary

Compilation Report - g10_FIR

Flow Status	Successful - Tue Apr 14 03:10:46 2020
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
Revision Name	g10_FIR
Top-level Entity Name	g10_FIR
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	49 / 32,070 (< 1 %)
Total registers	212
Total pins	20 / 457 (4 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	24 / 87 (28 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0

Figure 11: Resource utilization of the reduced bitwidth program



Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	12.88 MHz	12.88 MHz	clk	

Figure 12: Fmax of the reduced bitwidth program

In this implementation, we also noted a maximum clock frequency of 12.45MHz, or a period of ≈85ns. This period is in violation of the 20ns target.

One method we attempted to resolve this problem is to reduce the amount of combinational logic needed in the circuit. As part of this, we successively reduced the bitwidth of the given data (and of the outputted data) as a means to reduce the amount of logic needed in the circuit. By reducing the number of input bits to just 8, we could reduce the amount of registers needed significantly from 458 to just 212 (shown in figure 11 vs figure 2) without impacting the RMSE of the output (was 0.049 with a 10 bit output). However, figure 13 shows that despite the reduced logic use, the maximum frequency of the circuit is not raised significantly (only 0.43 MHz faster), and still violated the 20ns constraint, so we moved on to implementing the broadcast method.

Broadcasting Form of the FIR Filter (2nd Implementation)

The broadcasting (or transpose) form of the FIR filter works by reversing the direction of the

branches, and reversing the roles of the input and outputs. As the broadcast form is just a different implementation of the FIR filter, it is governed by the same equation (Fig 5) as the direct implementation.

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

Figure 13: Equation for the FIR Filter

Despite following the same mathematical rule, the broadcasting form supports a higher system frequency because the input signal reaches all of the multipliers at the same time, and does not need extra pipelining infrastructure, reducing the amount of required registers.

Below is our implementation of the broadcast-form of the FIR filter.

Allan Reuben (260783324)

12

Noah Zwack (260769910)

Group 10

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

entity g10_FIR_B 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) -- Outpput signal
);
end g10_FIR_B;

architecture a0 of g10_FIR_B is
type array_1 is array(24 downto 0) of signed(15 downto 0);
signal temp_weights: array_1;
type array_2 is array(24 downto 0) of signed(31 downto 0);
signal output_array: array_2;

begin
-- Input weights, truncated, and reversed in order
temp_weights(25) <= "0000001001110010";
temp_weights(24) <= "0000000000010001";
temp_weights(23) <= "1111111111010011";
temp_weights(22) <= "1111111011011110";
temp_weights(21) <= "0000001100011001";
temp_weights(20) <= "1111110110100111";
temp_weights(19) <= "1111110000001110";
temp_weights(18) <= "0000110110111100";
temp_weights(17) <= "1110110001110011";
temp_weights(16) <= "0000110111110111";
temp_weights(15) <= "0000001100000111";
temp_weights(14) <= "1110101000001010";
temp_weights(13) <= "0001111000110011";
temp_weights(12) <= "1110101000001010";
temp_weights(11) <= "0000001100000111";
temp_weights(10) <= "0000110111110111";
temp_weights(9) <= "1110110001110011";
temp_weights(8) <= "0000110110111100";
temp_weights(7) <= "1111110000001110";
temp_weights(6) <= "1111110110100111";
temp_weights(5) <= "0000001100011001";
temp_weights(4) <= "1111111011011110";
temp_weights(3) <= "1111111111010011";
temp_weights(2) <= "0000000000010001";
temp_weights(1) <= "0000001001110010";
```

Allan Reuben (260783324)

13

Noah Zwack (260769910)

Group 10

```
process(clk, rst)
    variable temp: signed(31 downto 0) := (others => '0');
    variable output: signed(16 downto 0) := (others => '0');
begin
    -- Reset all values on reset
    if rst = '1' then
        temp := (others => '0');
        output := (others => '0');
        y <= (others => '0');
    -- On rising edge of the clock
    elsif rising_edge(clk) then
        output := (others => '0');
        -- Do the multiplication and store in array
        for i in 0 to 24 loop
            output_array(i) <= temp_weights(i)*signed(x);
        end loop;

        -- Sum the results of the multipliation between the weight and input
        for i in 0 to 24 loop
            temp := output_array(i);
            output := output + temp(31 downto 15);
        end loop;
        -- Set output
        y <= std_logic_vector(output);
    end if;
end process;
end a0;
```

Figure 14: Our implementation of the broadcast-form FIR filter.

In this implementation, we declare 2 arrays, the first being used to store the weights of the filter to be used in multiplication, and the second used to store the output of the multiplication. The output array is longer, as to store a n bit by n bit multiplication you need 2n bits. We then declare 2 variables, temp and output. Temp is used to temporarily store the output of the multiplication, before it is truncated to only the most important 17 bits. Our program first checks if there is a reset (and if there is, reset the device), then on every clock edge, perform the multiplication for each tap, and then truncate and sum the multiplications to generate the output. As shown by figures 15-19 below, the broadcast FIR performed much faster and did not violate the 20ns clock constraint.

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	126.73 MHz	126.73 MHz	clk	

Fast 1100mV 85C Model Hold Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	0.239	0.000

Slow 1100mV 85C Model Hold Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	0.374	0.000

Fast 1100mV 85C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	15.226	0.000

Slow 1100mV 85C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	clk	12.109	0.000

Figures 15-19: Timing report of the broadcast FIR. No violations