Group 11, L02
Anish Paramsothy, paramsa, 400300097
Maryem Abdullatif, abdulm41, 400297192

McMaster University

Lab 3: Digital Signal Processing on FPGAs

MECHTRON 3TB4 POSTLAB, McMaster University

Date Submitted: March 9th, 2025

Due Date: March 10th, 2025

Contents

1 Introduction

The 3rd lab of the course utilized signal processing, in which we connected to the DE1-SoC's `MIC IN/LINE IN` and `LINE OUT` in order to take in an audio and perform the following tasks:

- Utilize an FIR filter to filter out a specific frequecy from a secret code audio file. This would allow us to hear the code better.
- Create an Echo by taking in audio, delay it, and then summing it with the original to create the echo effect.
- Create modes by using `SW[0]` and `SW[1]` to change between them. The modes are outlined below:
    - `SW[1]` ON and `SW[0]` ON: *Audio will be normal.*
    - `SW[1]` OFF and `SW[0]` OFF: *Audio will be normal.*
    - `SW[1]` ON and `SW[0]` OFF: *Audio will echo.*
    - `SW[1]` OFF and `SW[0]` ON: *Audio will go through an FIR filter.*

2 Tutorial

2.1 MATLAB Filtering

During Tutorial 3, a MATLAB file that was given was used to generate a list of coefficents, with the number of coefficients equaling the number of taps we plan to have in our FIR filter.

We had to change the number of taps during the lab to actually perform better filtering. The assumption was since the signal fed into the video was not perfectly 2000 Hz, we needed to broaden the scope of the filter be reducing the number of taps, so that it could include more extra noise in the filtering. The entire, finished Matlab code can be seen below:

```
% Read the .wav file (replace file_name by your group s file
% Variable x stores the wave and fs stores the sampling rate

[x, fs]=audioread('C:\Users\Anish\OneDrive\Documents\FIFTH YEAR\WINTER 2025\MECHTRON 3TB4\Labs\Lab3\Tutorial3\L02_Group11.wav\L02_Group11.wav');

% Perform FFT on the original signal to determine the frequency of the "noise"

L=length(x);
NFFT=2^nextpow2(L);
X=fft(x,NFFT)/fs;

% Show the sampling rate
fs

% We know the sampling rate is 8000
% We need now to plot our FFT to find the source of the noise.
% Plot single-sided amplitude spectrum

f=fs/2*linspace(0,1,NFFT/2+1);
plot(f,2*abs(X(1:NFFT/2+1)));

% Reading the FFT we realize that the frequency we want to remove is 2000HZ
% (Hint: our noise is a pure sine wave)
% Now specify the frequency you want to eliminate by setting:

fkill=2000/4000;

% Hint: fkill is always in the range of 0 to 1, and is normalized to frequency of fs/2.
% Determine the coefficients of the FIR filter that will remove that frequency.
% Start off the following blank with the value 4, to numbers larger than 160.
% Note: the following filter only works with EVEN numbers.
%the sharper the cliff, the more precise it is at removing the frequency.

coeff=firgr(12,[0,fkill-0.48, fkill, fkill+0.48, 1], [1,1,0,1,1],{ 'n' , 'n', 's' , 'n' , 'n' });
% Plot the filter
% Plot the frequency response of the designed filter to verify that it satisfies the
% Requirements
freqz(coeff,1);
% You should try different filter lengths in the firgr command and find out which one is
% the best. Filter length of 4 is terrible. Ideally, your filter should only filter out
% the noise while passing all other signals. Try increasing your filter length until
% you can achieve an adequate result.
% Be sure to plot (with freqz()) each time you create a new filter. If you pick a filter
% length too large, the filter will "blow up". If you are unsure whether your filter has
% blown up or not, seek help from a TA.
% Save these coefficients in a text file, You will need them when coding the FIR
```

```
    filter.
fid=fopen( 'C:\Users\Anish\OneDrive\Documents\FIFTH YEAR\WINTER 2025\MECHTRON 3TB4\
    Labs\Lab3\Tutorial3\L02_Group11.wav\Tutorial3.txt', 'w' );
% If you make a typing error with the following for-end block, you need to start from
    the "for" line again.

for i=1:length(coeff)
fprintf(fid, 'coeff[%3.0f]=%10.0f;\n',i-1,32768*coeff(i));
end
fclose(fid);

% Filter the input signal x(t) using the designed FIR filter to get y(t).

y=filtfilt(coeff,1,x);

% Perform FFT on the filtered signal to observe the absence of frequency of the "noise
    ".

Y=fft(y,NFFT)/L;

% Play the unfiltered sound (your system must have a
% working speaker or headphone)
% Multiply by 3 to make the volume 3 times louder.

sound(3*x,fs);

% Pause 5 seconds. (this is only necessary if you run these commands as a script)

pause(5);

% Play the filtered sound

sound(3*y,fs);

% The secret code for your group is JTAC15 (or GTAC15)
% Create two plots to compare

subplot(2,1,1);

% The first plot shows the FFT of the original signal.

plot(f,2*abs(X(1:NFFT/2+1)));
xlabel('frequency (Hz)');
ylabel('|X(f)|');

% The second plot shows the FFT of the filtered signal.

subplot(2,1,2);
plot(f, 2*abs(Y(1:NFFT/2+1)));
xlabel('frequency(Hz)');

%Write the filtered audio file to disk.
audiowrite('C:\Users\Anish\OneDrive\Documents\FIFTH YEAR\WINTER 2025\MECHTRON 3TB4\
    Labs\Lab3\Tutorial3\L02_Group11.wav\filtered2.wav',y,fs);
```

The first step in formulating the Matlab file was to determine the sampling rate (which was 8000 Hz) and then use that to plot a FFT to figure out which frequency we wanted to remove.

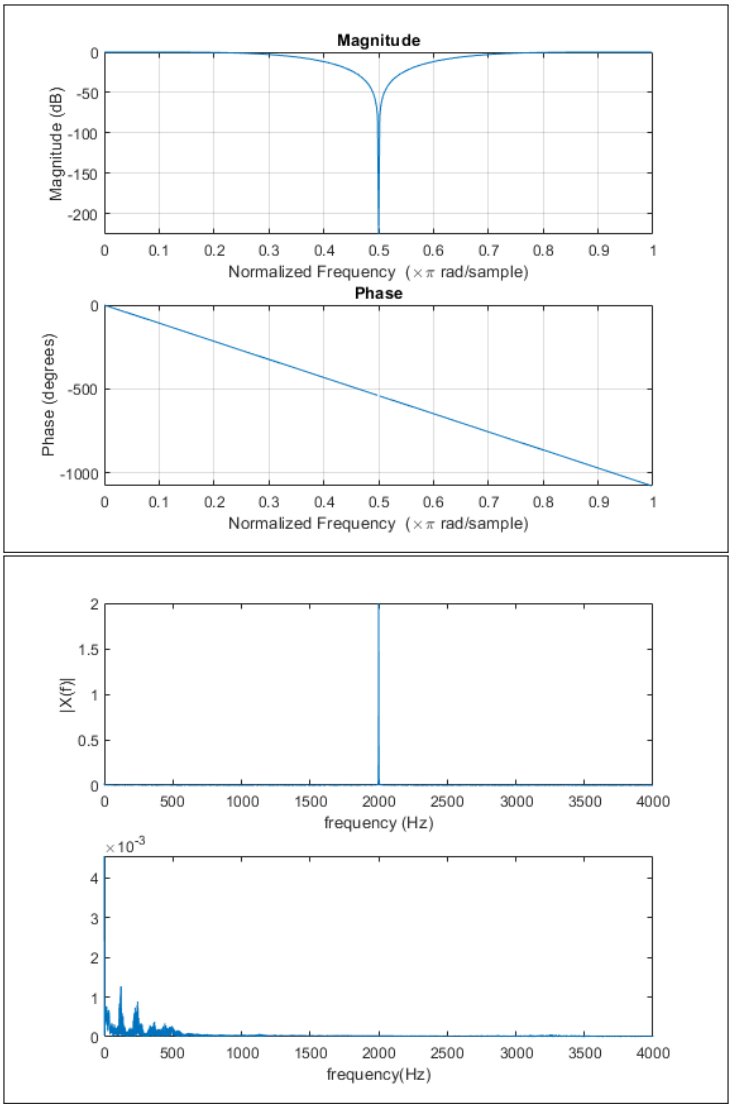

Figure 1: Frequency graphs

From the plots generated, we can tell that the frequency that needs to be removed is the 2000 Hz frequency.

We need to calculate the normalized frequency, which is shown in `fkill`, where we take the frequency we want to remove (cutoff frequency) and divide it by the Nyquist frequency (which is half the sampling frequency).

$$f_{\text{kill}} = \frac{f_{\text{cutoff}}}{f_{\text{Nyquist}}} \tag{0.1}$$

Using the `firgr` function, the coefficients for the FIR filter can be generated. Note that we wanted 65 coefficients since that is the default parameter we set in the FIR module later on. However, as mentioned above, we needed to tweak it down to 13 in order to actually get a better filtering effect. The Matlab file also generated an unfiltered file that removed the noise.

3 Modules

3.1 Number of Samples

The first part of the pre-lab was to determine the number of samples in a sound that lasts X milliseconds, where X is 97 or 92.

$$\text{Number of Samples}_{(\text{Anish})} = f_s \times 97\text{ms} \times \frac{1\text{s}}{1000\text{ms}} = 776$$
$$\text{Number of Samples}_{(\text{Maryem})} = f_s \times 92\text{ms} \times \frac{1\text{s}}{1000\text{ms}} = 736$$

3.2 DSP Subsystem

The DSP System acts as a 3-to-1 multiplexer that flips with the three modes: normal, FIR, and echo. The following code was created:

```
module dsp_subsystem (input sample_clock,  input reset, input [1:0] selector, input
    [15:0] input_sample, output [15:0] output_sample);

  //assign output_sample = input_sample;

  wire [15:0] echo_output;
  wire [15:0] FIR_output;
  reg [15:0] output_MUX;

  echo (.sample_clock(sample_clock), .input_sample(input_sample), .output_sample(
    echo_output));
  FIR_FILTER(.clk(sample_clock), .reset(reset),.input_sample(input_sample), .
    output_sample(FIR_output));

  assign output_sample = output_MUX;

  always @(*) begin

    case(selector)
      2'b00: output_MUX = input_sample;
      2'b01: output_MUX = FIR_output;
      2'b10: output_MUX = echo_output;
      default: output_MUX = input_sample;
    endcase

  end

endmodule
```

The RTL view of the DSP circuit can be seen below:

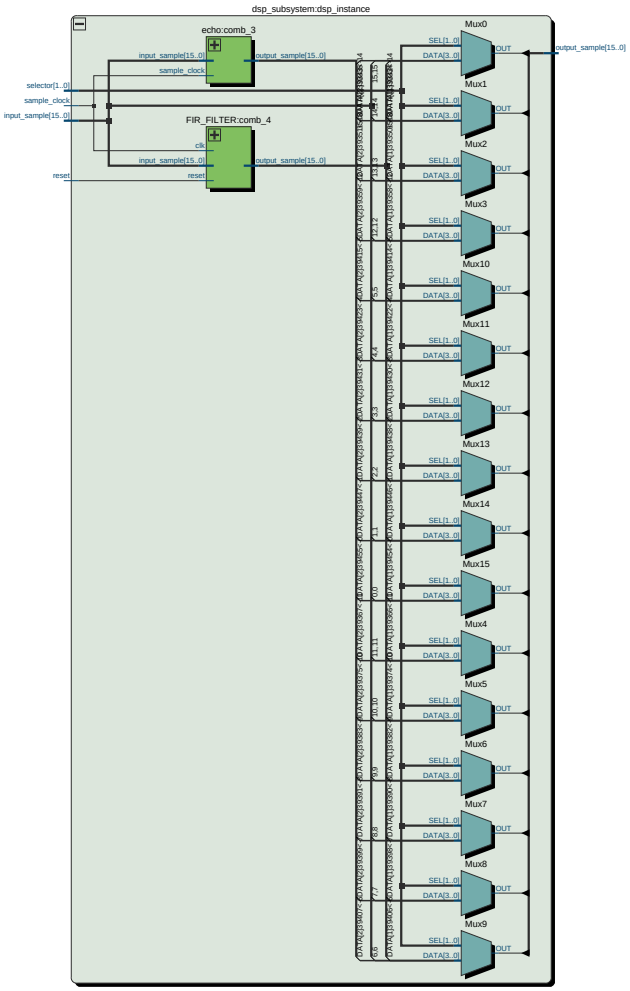

Figure 2: DSP Subsystem

3.3 FIR Filter

The FIR Filter follows the following equation:

$$y[n] = \sum_{k=0}^{N-1} b_k x[n-k] \tag{0.2}$$

where $x[n]$ is the filter input, b_k are the filter coefficients, $y[n]$ is the filter output, and N is the number of filter coefficients (or order of the filter).

Implementing an FIR filter in Verilog requires the use of a multipler. An instance of the `LPM_MULT` library component can be called to perform this. The goal is to multiply the coefficient to a specific input in the input sample, and summing those together. The input is created from taking the input

sample from the `LINE IN` and setting it equal to index 0 of the input. It is then shifted, while index 0 is replaced with the next sample.

The Verilog implementation can be seen below. The use of fewer coefficients resulted in better filtering when the mode was in `LINE IN`, but more distortion when the mode was `MIC IN`.

A couple changes were implemented in the debugging process during the lab period. The first change was separating the summation and input updating into different `always` blocks. This is because the FIR filter only updates inputs during the positive clock edge, while the summation is constantly occuring. The second change was reducing the number of coefficients from 65 to 13, which resulted in far better filtering. We assume this is due to the imperfect nature of the frequency in the provided audio, so broadening the range of the filter would improve its ability to distinguish the noise from the secret code.

```
module FIR_FILTER(input clk, input reset, input [15:0] input_sample, output reg signed
     [15:0] output_sample);

  parameter N = 13;
  wire signed [15:0] coeff [0:N-1];

  integer j;

  // filter coefficients. used to determine how much weight each past input sample
    gets in the output calculation
  // 12 taps (fkill +- 0.48)
  assign coeff[0] = 515;
  assign coeff[1] = -0;
  assign coeff[2] = -3078;
  assign coeff[3] = 0;
  assign coeff[4] = 7677;
  assign coeff[5] = -0;
  assign coeff[6] = 22540;
  assign coeff[7] = -0;
  assign coeff[8] = 7677;
  assign coeff[9] = 0;
  assign coeff[10] = -3078;
  assign coeff[11] = -0;
  assign coeff[12] = 515;

  reg signed [15:0] uk [0:N-1];
  // to prevent errors in next steps from truncation, set the output of multiplier to
    be the maximum amount of bits it can be
  wire signed [31:0] multiplier_output [0:N-1];
  reg signed [31:0] summation;

  // perform signed multiplication
  generate
    genvar i;
    for (i = 0; i < N; i = i + 1) begin: multiplication
      multiplier (.dataa(uk[i]), .datab(coeff[i]), .result(multiplier_output[i]));
    end
  endgenerate

  always @(posedge clk or posedge reset) begin

    if (reset) begin
      //output_sample <= 0;
      for (j = 0; j < N; j = j + 1) begin
        uk[j] <= 0;
      end
    end

    else begin

```

```
      // constantly updating inputs with the newest sample
      for (j = N-1; j > 0; j = j - 1) begin
        uk[j] <= uk[j-1];
      end
      uk[0] <= input_sample;

    end

  end

  always @(*) begin

    // FIR filter summation (performing uk0*b0+uk-1*b1+...+uk-N*bN)
    summation = 0;
    for (j = 0; j < N; j = j + 1) begin
      summation = summation + multiplier_output[j];
    end

    // outputting FIR calculation
    output_sample <= summation[15:0];

  end

endmodule
```

The multiplier can be seen below as well:

```
// megafunction wizard: %LPM_MULT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: lpm_mult

// ============================================================
// File Name: multiplier.v
// Megafunction Name(s):
//      lpm_mult
//
// Simulation Library Files(s):
//      lpm
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 17.1.0 Build 590 10/25/2017 SJ Lite Edition
// ************************************************************


//Copyright (C) 2017  Intel Corporation. All rights reserved.
//Your use of Intel Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Intel Program License
//Subscription Agreement, the Intel Quartus Prime License Agreement,
//the Intel FPGA IP License Agreement, or other applicable license
//agreement, including, without limitation, that your use is for
//the sole purpose of programming logic devices manufactured by
//Intel and sold by Intel or its authorized distributors.  Please
//refer to the applicable agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module multiplier (
```

```
    dataa,
    datab,
    result);

    input [15:0]  dataa;
    input [15:0]  datab;
    output  [15:0]  result;

    wire [15:0] sub_wire0;
    wire [15:0] result = sub_wire0[15:0];

    lpm_mult  lpm_mult_component (
                .dataa (dataa),
                .datab (datab),
                .result (sub_wire0),
                .aclr (1'b0),
                .clken (1'b1),
                .clock (1'b0),
                .sclr (1'b0),
                .sum (1'b0));
    defparam
        lpm_mult_component.lpm_hint = "MAXIMIZE_SPEED=5",
        lpm_mult_component.lpm_representation = "SIGNED",
        lpm_mult_component.lpm_type = "LPM_MULT",
        lpm_mult_component.lpm_widtha = 16,
        lpm_mult_component.lpm_widthb = 16,
        lpm_mult_component.lpm_widthp = 16;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: AutoSizeResult NUMERIC "0"
// Retrieval info: PRIVATE: B_isConstant NUMERIC "0"
// Retrieval info: PRIVATE: ConstantB NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: PRIVATE: LPM_PIPELINE NUMERIC "0"
// Retrieval info: PRIVATE: Latency NUMERIC "0"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: SignedMult NUMERIC "1"
// Retrieval info: PRIVATE: USE_MULT NUMERIC "1"
// Retrieval info: PRIVATE: ValidConstant NUMERIC "0"
// Retrieval info: PRIVATE: WidthA NUMERIC "16"
// Retrieval info: PRIVATE: WidthB NUMERIC "16"
// Retrieval info: PRIVATE: WidthP NUMERIC "16"
// Retrieval info: PRIVATE: aclr NUMERIC "0"
// Retrieval info: PRIVATE: clken NUMERIC "0"
// Retrieval info: PRIVATE: new_diagram STRING "1"
// Retrieval info: PRIVATE: optimize NUMERIC "0"
// Retrieval info: LIBRARY: lpm lpm.lpm_components.all
// Retrieval info: CONSTANT: LPM_HINT STRING "MAXIMIZE_SPEED=5"
// Retrieval info: CONSTANT: LPM_REPRESENTATION STRING "SIGNED"
// Retrieval info: CONSTANT: LPM_TYPE STRING "LPM_MULT"
// Retrieval info: CONSTANT: LPM_WIDTHA NUMERIC "16"
// Retrieval info: CONSTANT: LPM_WIDTHB NUMERIC "16"
// Retrieval info: CONSTANT: LPM_WIDTHP NUMERIC "16"
// Retrieval info: USED_PORT: dataa 0 0 16 0 INPUT NODEFVAL "dataa[15..0]"
// Retrieval info: USED_PORT: datab 0 0 16 0 INPUT NODEFVAL "datab[15..0]"
// Retrieval info: USED_PORT: result 0 0 16 0 OUTPUT NODEFVAL "result[15..0]"
// Retrieval info: CONNECT: @dataa 0 0 16 0 dataa 0 0 16 0
// Retrieval info: CONNECT: @datab 0 0 16 0 datab 0 0 16 0
// Retrieval info: CONNECT: result 0 0 16 0 @result 0 0 16 0
// Retrieval info: GEN_FILE: TYPE_NORMAL multiplier.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL multiplier.inc FALSE
```

```
// Retrieval info: GEN_FILE: TYPE_NORMAL multiplier.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL multiplier.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL multiplier_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL multiplier_bb.v FALSE
// Retrieval info: LIB_FILE: lpm
```

The FIR filter module created the following RTL circuit below. note that the entire circuit is provided, but screen shots of various sections were taken since it is difficult to see.

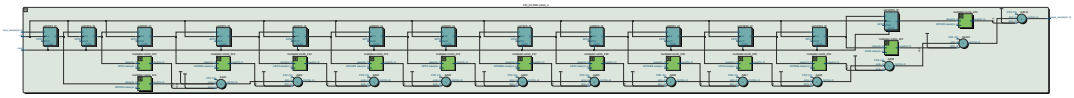

Figure 3: RTL of FIR

Since this does not tell you much from how many components there are, there are a few screenshots below of snippets of the RTL circuit:

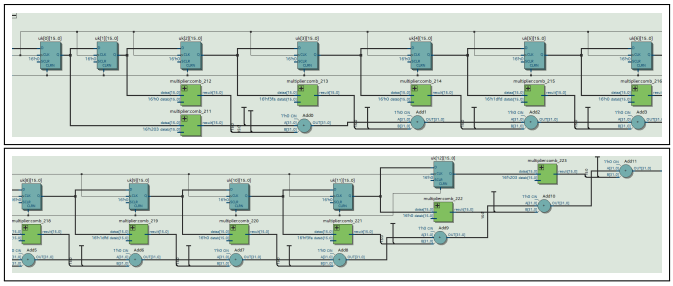

Figure 4: RTL of FIR

Note that these snippets show that the circuit follows the general trend of the FIR filter, and when running the code, it seems to perform some functionality of filtering out the noise.

3.4 Echo Machine

The other mode inside the DSP was the echo machine, which would take any input audio, and delay it by an amount.

The echo is meant to take the input and divide it using a shift register to create the delayed sample. This is then added with the original to create the echo effect.

The current problem is that since we need to use many registers it can eat away at the total resources that the FPGA has. In order to deal with this, using the shift register IP Catalog component, there is a way to implement a memory based register system.

With all of this in mind, the echo machine was formed:

```
module echo(input sample_clock, input [15:0] input_sample, output reg [15:0]
    output_sample);

  wire [15:0] echo_in;
  wire [15:0] echo_out;

  assign echo_in = input_sample;

  shift_register (.clock(sample_clock), .shiftin(echo_in), .shiftout(echo_out));
```

```
9
10   always @(posedge sample_clock) begin
11     output_sample <= input_sample + (echo_out >> 2);
12   end
13
14 endmodule
```

The shift register is also noted below:

```
1  // megafunction wizard: %Shift register (RAM-based)%
2  // GENERATION: STANDARD
3  // VERSION: WM1.0
4  // MODULE: ALTSHIFT_TAPS
5
6  // ============================================================
7  // File Name: shift_register.v
8  // Megafunction Name(s):
9  //      ALTSHIFT_TAPS
10 //
11 // Simulation Library Files(s):
12 //      altera_mf
13 // ============================================================
14 // ************************************************************
15 // THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
16 //
17 // 17.1.0 Build 590 10/25/2017 SJ Lite Edition
18 // ************************************************************
19
20
21 //Copyright (C) 2017  Intel Corporation. All rights reserved.
22 //Your use of Intel Corporation's design tools, logic functions
23 //and other software and tools, and its AMPP partner logic
24 //functions, and any output files from any of the foregoing
25 //(including device programming or simulation files), and any
26 //associated documentation or information are expressly subject
27 //to the terms and conditions of the Intel Program License
28 //Subscription Agreement, the Intel Quartus Prime License Agreement,
29 //the Intel FPGA IP License Agreement, or other applicable license
30 //agreement, including, without limitation, that your use is for
31 //the sole purpose of programming logic devices manufactured by
32 //Intel and sold by Intel or its authorized distributors.  Please
33 //refer to the applicable agreement for further details.
34
35
36 // synopsys translate_off
37 `timescale 1 ps / 1 ps
38 // synopsys translate_on
39 module shift_register (
40     clock,
41     shiftin,
42     shiftout,
43     taps);
44
45     input   clock;
46     input [15:0]  shiftin;
47     output  [15:0]  shiftout;
48     output  [767:0]  taps;
49
50     wire [15:0] sub_wire0;
51     wire [767:0] sub_wire1;
52     wire [15:0] shiftout = sub_wire0[15:0];
53     wire [767:0] taps = sub_wire1[767:0];
54
55     altshift_taps ALTSHIFT_TAPS_component (
56                 .clock (clock),
57                 .shiftin (shiftin),
```

```
                .shiftout (sub_wire0),
                .taps (sub_wire1)
                // synopsys translate_off
                ,
                .aclr (),
                .clken (),
                .sclr ()
                // synopsys translate_on
                );
    defparam
        ALTSHIFT_TAPS_component.intended_device_family = "Cyclone V",
        ALTSHIFT_TAPS_component.lpm_hint = "RAM_BLOCK_TYPE=MLAB",
        ALTSHIFT_TAPS_component.lpm_type = "altshift_taps",
        ALTSHIFT_TAPS_component.number_of_taps = 48,
        ALTSHIFT_TAPS_component.tap_distance = 32,
        ALTSHIFT_TAPS_component.width = 16;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ACLR NUMERIC "0"
// Retrieval info: PRIVATE: CLKEN NUMERIC "0"
// Retrieval info: PRIVATE: GROUP_TAPS NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: PRIVATE: NUMBER_OF_TAPS NUMERIC "48"
// Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
// Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
// Retrieval info: PRIVATE: TAP_DISTANCE NUMERIC "8"
// Retrieval info: PRIVATE: WIDTH NUMERIC "16"
// Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
// Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: CONSTANT: LPM_HINT STRING "RAM_BLOCK_TYPE=MLAB"
// Retrieval info: CONSTANT: LPM_TYPE STRING "altshift_taps"
// Retrieval info: CONSTANT: NUMBER_OF_TAPS NUMERIC "48"
// Retrieval info: CONSTANT: TAP_DISTANCE NUMERIC "8"
// Retrieval info: CONSTANT: WIDTH NUMERIC "16"
// Retrieval info: USED_PORT: clock 0 0 0 0 INPUT NODEFVAL "clock"
// Retrieval info: USED_PORT: shiftin 0 0 16 0 INPUT NODEFVAL "shiftin[15..0]"
// Retrieval info: USED_PORT: shiftout 0 0 16 0 OUTPUT NODEFVAL "shiftout[15..0]"
// Retrieval info: USED_PORT: taps 0 0 768 0 OUTPUT NODEFVAL "taps[767..0]"
// Retrieval info: CONNECT: @clock 0 0 0 0 clock 0 0 0 0
// Retrieval info: CONNECT: @shiftin 0 0 16 0 shiftin 0 0 16 0
// Retrieval info: CONNECT: shiftout 0 0 16 0 @shiftout 0 0 16 0
// Retrieval info: CONNECT: taps 0 0 768 0 @taps 0 0 768 0
// Retrieval info: GEN_FILE: TYPE_NORMAL shift_register.v TRUE
// Retrieval info: GEN_FILE: TYPE_NORMAL shift_register.inc FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL shift_register.cmp FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL shift_register.bsf FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL shift_register_inst.v FALSE
// Retrieval info: GEN_FILE: TYPE_NORMAL shift_register_bb.v FALSE
// Retrieval info: LIB_FILE: altera_mf
```

The tap distance was what created the delay, so a noticeable echo was present around 32 tap distance and above.

The RTL of the echo can be seen below:

This also checks out with the standard filter device, and it was verified through actual testing using the FPGA.

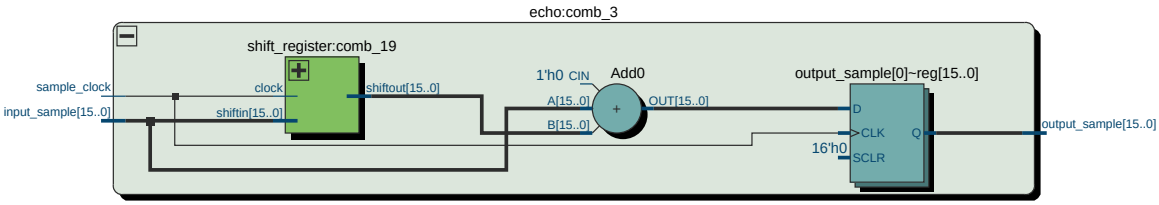

Figure 5: Echo machine

4 Conclusion

4.1 Compilation Report

Running the code gives us the following compilation report:

Flow Summary

<<Filter>>

Flow Status	Successful - Mon Mar 03 21:38:18 2025
Quartus Prime Version	17.1.0 Build 590 10/25/2017 SJ Lite Edition
Revision Name	lab3
Top-level Entity Name	lab3
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	535 / 32,070 (2 %)
Total registers	152
Total pins	13 / 457 (3 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total DSP Blocks	8 / 87 (9 %)
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 %)

Figure 6: Compilation Report

4.2 Post-lab Questions

Q1: What values did you use for the echo delay and the division factor?

- **Number of Taps:** 48
- **Tap Distance:** 32
- **Bit Width:** 16
- **Division Factor:** 2

Q2: There are a limited number of multipliers on the Cyclone V FPGA. How can you rediesign the filter to use less multiplier units?

In order to reduce the number of multiplier units, we need to reduce the number of coefficients used. This occured during the debugging process in the lab, where we reduced the amount of multipliers from 65 to 13. This also helped filter a broader range of signals.

Q3: Open the compilation report in Quartus, and report the following numbers:

- **Total Number of logic elements used by your circuit:** 535/32,070
- **Total number of registers:** 152
- **Total number of pins:** 13/457
- **The maximum number of logic elements that can fit on the FPGA used:** 32,070

Q4: What is a logic element, and what are the componenets that make up a logic element on the Cylone V FPGA?

Logic elements are used for performing logic operations (smallest unit of logic), and in the Cyclone series they are made up of a four-input LUTs, a programmable register, and a carry chain [Intel.com].

Q5: What is the purpose of the multiplexers in the lab design?

Multiplexers act as a tool to combine different sets of logic, and separating them through various cases. In the scope of this lab they are used to switch between the regular, FIR, and echo modes of output.

Q6: How many memory bits does it take to store the samples in the delay circuit (shift register) in your echo machine?

Memory Bits can be calculated using the following equation:

$$\begin{aligned}\text{Total Memory Bits} &= \text{Number of Taps} \times \text{Bit Width} \times \text{Tap Distance} \\ \text{Total Memory Bits} &= 48 \text{ Taps} \times 16 \text{ Bit Width} \times 32 \text{ Tap Distance} \\ \text{Total Memory Bits} &= \boxed{24,576}\end{aligned}$$

Therefore the number of memory bits it takes to store the samples in the delay circuit is 24,576.

Q7: Considering that each logic block contains only one register, would your design have fit onto the FPGA had you not used the memory-based shift register?

The design would have fit on the board. The idea is the number of memory bits in this model (without a RAM-based shift register) would essentially be equivalent to the number of shift registers used, because each logic block only contains one register. With this in mind, we would still have less than the total of 32,070 logic units available. It should also be noted that a memory-based shift register has less design complexity.

4.3 Wrap-Up

Overall, the third lab was a fascinating use of the FPGA, as we were able to input sound from our devices and modify them in the two different ways. We were able to apply new methods, such as using the built in shift register and multiplier components, while also applying new logic into hardware. The output was satisfying and was very exciting!