

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA



BÀI TẬP LỚN MÔN XỬ LÝ TÍN HIỆU SỐ VỚI FPGA

GVHD: Thầy Nguyễn Tuấn Hùng

LAB1: FIR filter on FPGA using systemverilog

STT	Họ và tên	MSSV
1	Lê Duy Thúc	2112416
2	Đoàn Ngọc Minh Hiếu	2113338
3	Nguyễn Đình Đức	2012998
4	Nguyễn Xuân Hoàng	2111253
5	Lê Hoàng Bảo Long	2111663
6	Nguyễn Văn Cường	2112971

BÁO CÁO KẾT QUẢ LÀM VIỆC NHÓM

Lớp: L02

Số thứ tự nhóm: 07

STT	Họ và tên	MSSV	Nhiệm vụ	Hoàn thành
1	Lê Duy Thức	2112416	RTL, fir-filter, fir_tb, debug	120%
2	Đoàn Ngọc Minh Hiếu	2113338	fir_tb, debug	120%
3	Nguyễn Đình Đức	2012998	Parallel to serial, serial to paralle	120%
4	Nguyễn Xuân Hoàng	2111253	Matlab, báo cáo	120%
5	Lê Hoàng Bảo Long	2111663	RTL, báo cáo, hỗ trợ debug	120%
6	Nguyễn Văn Cường	2112971	Rút môn	0%

MỤC LỤC

MỞ ĐẦU	4
I. Yêu cầu đề tài	4
II. Mục tiêu	4
NỘI DUNG	5
I. Bộ lọc FIR truyền thống	5
II. Code bộ lọc FIR	10
III. Testbench mô phỏng	15
1. Testbench	15
2. Kết quả mô phỏng	18
IV. Kết nối phần cứng (Dự định thực hiện trên kit FPGA)	19
1. Serial To Parallel	19
2. Parallel To Serial	20
3. Wrapper	20
V. Kết quả	21
KẾT LUẬN	23
TÀI LIỆU THAM KHẢO	24

MỞ ĐẦU

I. Yêu cầu đề tài

- Chọn bộ lọc FIR và giải thích chức năng và ứng dụng của nó
- Lọc được nhiều âm thanh bằng bộ lọc FIR đã chọn
- Test bộ lọc có hoạt động thực tế bằng Kit FPGA

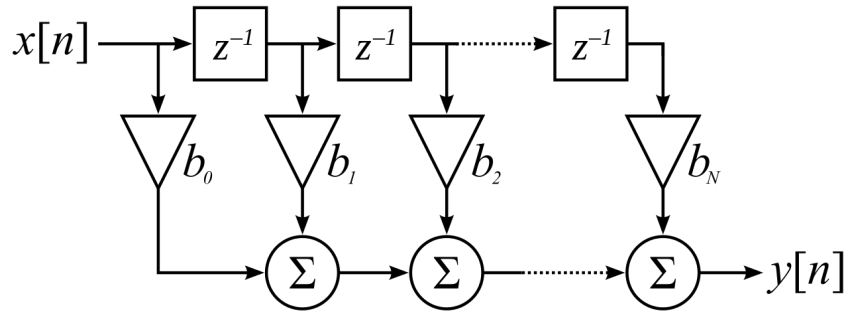
II. Mục tiêu

- Thiết kế và triển khai một hệ thống lọc nhiều âm thanh bằng bộ lọc FIR (Finite Impulse Response) hoạt động trên phần mềm mô phỏng và hoạt động trên kit FPGA DE2.
- Phát triển hệ thống lọc âm thanh giúp cải thiện chất lượng âm thanh đầu ra bằng cách loại bỏ nhiễu. Bộ lọc FIR có thể cho phép nhóm làm được điều này bằng cách áp dụng một bộ lọc có phản hồi hữu hạn tới các tín hiệu âm thanh đầu vào.
- Bộ lọc sẽ được triển khai trên kit FPGA DE2, cùng với đó nhóm sẽ trình bày chi tiết về quá trình thiết kế hệ thống, bao gồm việc lựa chọn các thông số của bộ lọc FIR, mô phỏng và kiểm tra hệ thống.
- Cuối cùng, nhóm đề tài này sẽ mang lại kết quả đúng như mong đợi, cùng với đó là cung cấp thêm những kiến thức hữu ích về xử lý tín hiệu âm thanh trên FPGA và đóng góp vào việc phát triển các ứng dụng âm thanh hiệu quả và linh hoạt hơn trong tương lai.

NỘI DUNG

I. Bộ lọc FIR truyền thống

Bộ lọc FIR là bộ lọc có đáp ứng xung chiều dài hữu hạn, tức là đáp ứng xung chỉ khác 0 trong một khoảng có chiều dài hữu hạn N (từ 0 đến N-1). Bộ lọc FIR với bậc N được biểu diễn như sau:



Hình 1. Sơ đồ biểu diễn của bộ lọc FIR

Trong đó:

- $x[n]$: là tín hiệu vào
- $y[n]$: là tín hiệu ra
- $h[n]$: là các đáp ứng xung
- $y[n]$ và $x[n]$ có mối quan hệ bởi công thức sau:

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n - k]$$

Để tính toán $y[n]$, tín hiệu $x[n]$ sẽ lần lượt đi qua các bộ trễ, bộ nhân và bộ cộng. Đối với bộ lọc FIR bậc N sẽ cần N bộ nhân và N-1 bộ cộng để cho ra giá trị $y[n]$.

Lý do chọn bộ lọc FIR truyền thống trên kit DE2:

- **Tính linh hoạt và dễ triển khai:** Cấu trúc đơn giản của bộ lọc FIR truyền thống giúp dễ dàng triển khai trên kit DE2, từ đó có thể giảm thời gian và công sức cần để phát triển cũng như kiểm tra hệ thống.
- **Khả năng loại bỏ nhiễu tốt cùng với hiệu suất cao:** Bộ lọc FIR có khả năng loại bỏ hiệu quả các nhiễu không mong muốn với hiệu suất cao, đặc biệt phù hợp cho các ứng dụng xử lý tín hiệu âm thanh, nơi yêu cầu hiệu suất lọc cao.
- **Độ ổn định và dễ điều chỉnh:** Bộ lọc FIR có tính ổn định cao và cho phép điều chỉnh chính xác các thông số như bậc lọc, hệ số lọc và tần số cắt, dễ dàng

trong việc thiết kế và tinh chỉnh hiệu suất lọc nhiễu trong mô phỏng và trên kit DE2.

Ứng dụng của bộ lọc FIR truyền thống:

- **Truyền thông:** Được sử dụng để lọc tín hiệu trước khi truyền qua kênh, giúp cải thiện độ rõ nét và giảm nhiễu.
- **Xử lý ảnh và âm thanh:** Nâng cao chất lượng hình ảnh và âm thanh thông qua việc loại bỏ nhiễu.
- **Hệ thống điều khiển:** Lọc tín hiệu đầu vào trước khi đưa vào các bộ điều khiển để tăng độ chính xác của hệ thống.

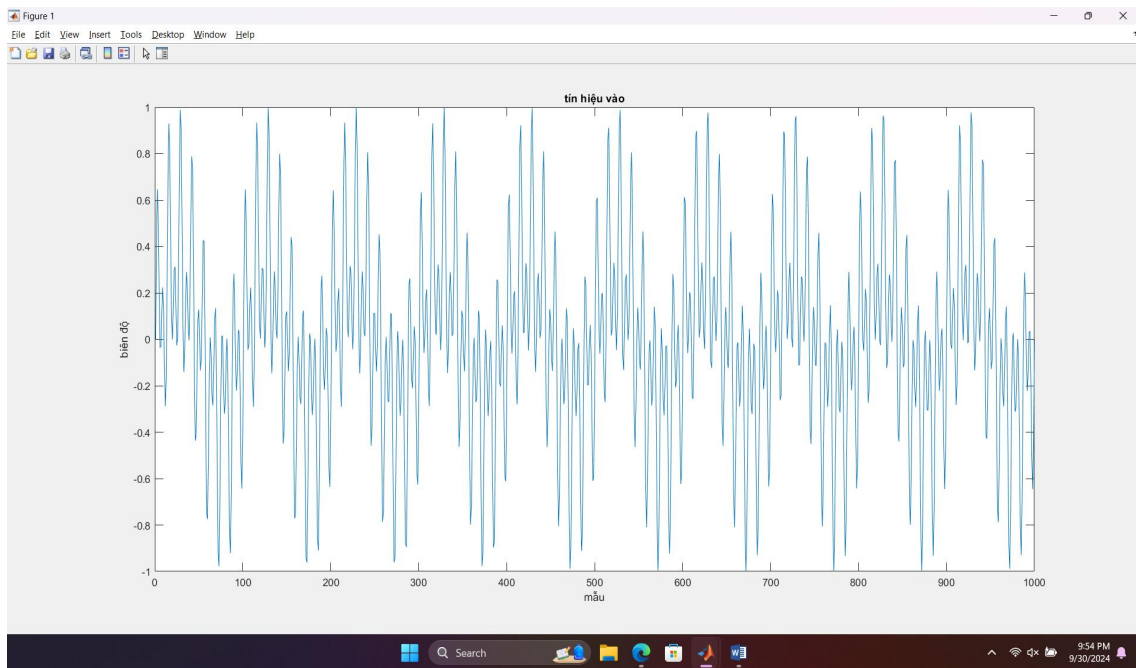
- Dưới đây là phần MATLAB để tìm hệ số của bộ lọc:

- Đoạn code này dùng để đọc file .hex ở đường dẫn, ở đây cụ thể là sine.hex

```
>> fileID = fopen('D:\DOWNLOAD\sine.hex','r');  
>> hexData = textscan(fileID,'%s');  
>> fclose(fileID);
```

- Tiếp theo cần biểu diễn ngõ vào của tín hiệu

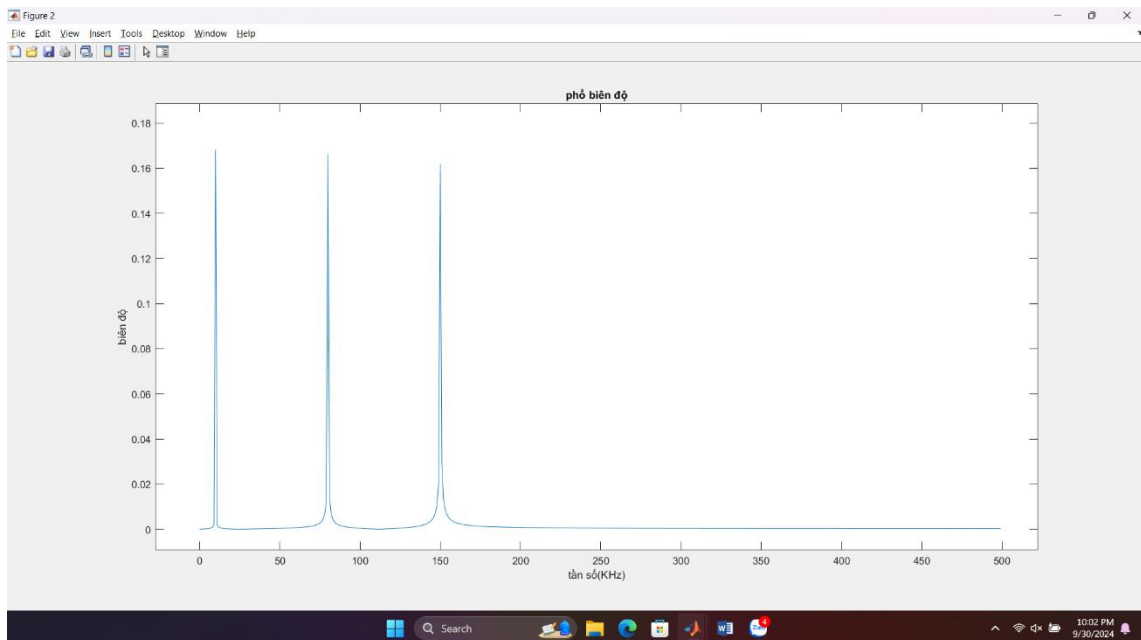
```
>> decimalData = zeros(length(hexData{1}),1);  
>> nBits = 24;  
>>  
>> for i = 1:length(hexData{1})  
    hexValue = hexData{1}{i};  
    decimalValue = hex2dec(hexValue);  
    if bitand(decimalValue, 2^(nBits-1))  
        decimalValue = decimalValue - 2^nBits;  
    end  
    decimalData(i) = decimalValue;  
end  
>> realData = decimalData / 2^(nBits-1);  
>> figure;  
>> plot(realData);
```



Hình 2. Tín hiệu đầu vào $\sin(x)$ trên MATLAB

- Ta cần phân tích tín hiệu và biểu diễn phổ của tín hiệu

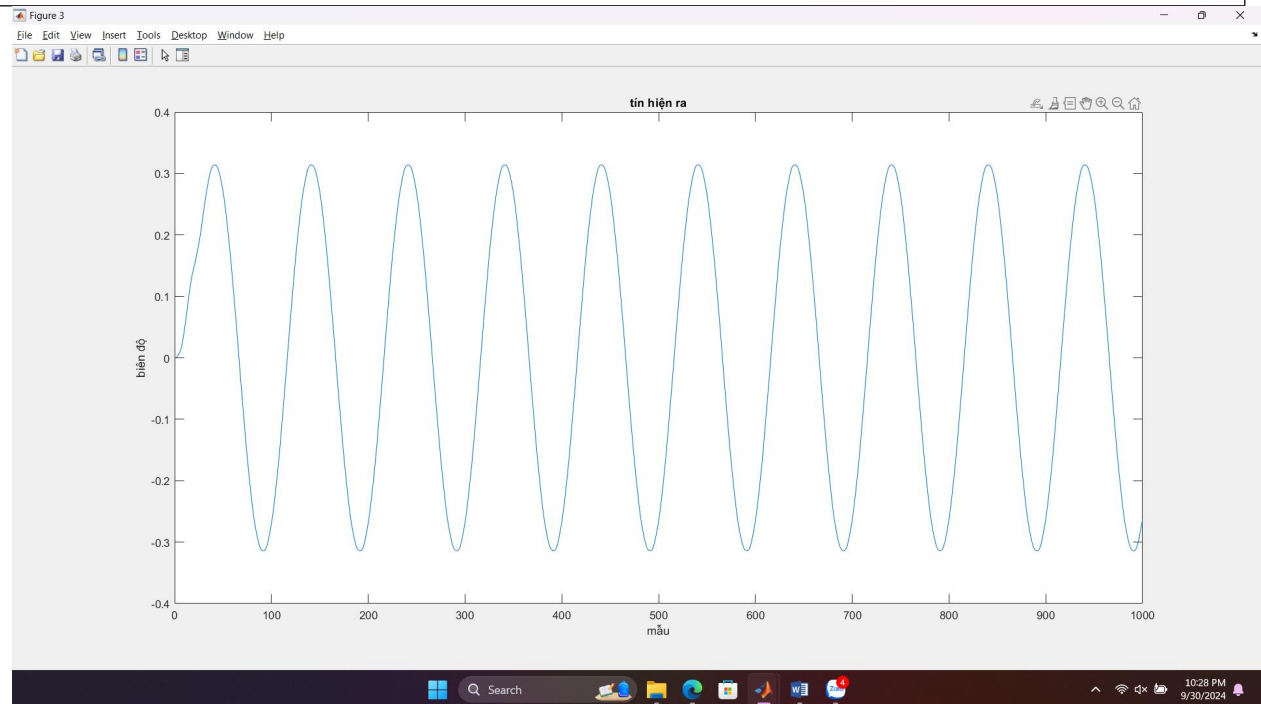
```
>> Fs = 1000;
>> X = fft(realData);
>> N = length(X);
>> f = (0:N-1)*(Fs/N);
>> X_magnitude = abs(X)/N;
>> figure;
>> plot(f(1:N/2), X_magnitude(1:N/2));
```



Hình 3. Phổ biên độ của tín hiệu đầu vào

- Thiết kế bộ lọc và hiển thị đầu ra

```
>> cutoff_freq = 20;  
>> normalized_cutoff = cutoff_freq/(Fs/2);  
>> filter_order = 31;  
>> b = fir1(filter_order, normalized_cutoff);  
>> filtered_data = filter(b, 1, realData);  
>> plot(filtered_data);
```



Hình 4. Tín hiệu đầu ra trên MATLAB

- Chuyển đổi từ số thực sang số hex của các hệ số bộ lọc

```
>> b_scaled = round(b * 2^23);  
>> b_hex_signed = arrayfun(@(x) dec2hex(typecast(int32(x), 'uint32'), 6),  
b_scaled, 'UniformOutput', false);  
>> disp(b_hex_signed);  
Columns 1 through 7  
  
{'005254'} {'006678'} {'0093DC'} {'00DEF4'} {'014A28'}  
{'01D580'} {'027E6F'}
```


Columns 8 through 14

{'033FDF'} {'041269'} {'04ECC8'} {'05C46E'} {'068E43'}
{'073F64'} {'07CDF8'}

Columns 15 through 21

{'0831DF'} {'086552'} {'086552'} {'0831DF'} {'07CDF8'}
{'073F64'} {'068E43'}

Columns 22 through 28

{'05C46E'} {'04ECC8'} {'041269'} {'033FDF'} {'027E6F'}
{'01D580'} {'014A28'}

Columns 29 through 32

{'00DEF4'} {'0093DC'} {'006678'} {'005254'}

- Như vậy ta đã sử dụng MATLAB để tìm ra hệ số của bộ lọc, tiếp đến là phần code của bộ lọc

II. Code bộ lọc FIR

- Khai báo các tín hiệu input và output cho bộ lọc FIR:

```
module fir_filter(  
    input clk,  
    input reset,  
    input signed [23:0] fir_input, // 24-bits input  
    input fir_ready,  
    output reg signed [23:0] fir_output // 24-bits output  
);
```

- clk: tín hiệu đồng hồ để đồng bộ hóa hoạt động của mô-đun.
- reset: tín hiệu khởi tạo lại mô-đun về trạng thái ban đầu.
- fir_input: Dữ liệu đầu vào của bộ lọc FIR, định dạng 24 bit.
- fir_ready: Tín hiệu xác nhận rằng mô-đun có thể nhận thêm dữ liệu đầu ra.
- fir_output: Dữ liệu đầu ra 24 bit.

```
// 32-tap FIR Filter  
parameter TAP_COUNT = 32;  
reg enable_fir, enable_buff;  
reg [4:0] buff_cnt;  
reg signed [23:0] in_sample;  
  
// buffer array to hold delayed self  
reg signed [23:0] buffer [0:TAP_COUNT-1];  
  
// tap coefficients obtain via matlab  
wire signed [23:0] taps [0:TAP_COUNT-1];  
  
// accumulator reg  
reg signed [47:0] acc [0:TAP_COUNT-1]; // 48-bit accumulation for precision  
reg signed [47:0] sum;
```

- enable_fir và enable_buff: các biến dùng để kiểm soát khi nào bộ lọc FIR và bộ đệm có thể hoạt động
- buff_cnt: bộ đếm 5-bit dùng để theo dõi vị trí hiện tại trong bộ đệm
- in_sample: mẫu đầu vào hiện tại cho bộ lọc FIR
- buffer0 đến buffer31: các thanh ghi lưu trữ 32 mẫu đầu vào gần nhất
- tap0 đến tap31: các dây nối tương tự ứng với mỗi mẫu trong bộ đệm
- acc0 đến acc31 : các thanh ghi lưu trữ kết quả tích lũy của mỗi mẫu sau khi áp dụng hệ số của bộ lọc FIR.

- Ở đây, bộ lọc FIR được chọn là bộ lọc thông thấp, tốc độ lấy mẫu là 1MS/s, passband frequency là 200kHz và stopband frequency là 355kHz

```
assign taps[0] = 24'h0051EB; // 0.0025 * 2^23
assign taps[1] = 24'h006594; // 0.0031 * 2^23
assign taps[2] = 24'h009374; // 0.0045 * 2^23
assign taps[3] = 24'h00DED2; // 0.0068 * 2^23
assign taps[4] = 24'h014AF4; // 0.0101 * 2^23
assign taps[5] = 24'h01D495; // 0.0143 * 2^23
assign taps[6] = 24'h027EF9; // 0.0195 * 2^23
assign taps[7] = 24'h03404E; // 0.0254 * 2^23
assign taps[8] = 24'h041205; // 0.0318 * 2^23
assign taps[9] = 24'h04ED91; // 0.0385 * 2^23
assign taps[10] = 24'h05C5D6; // 0.0451 * 2^23
assign taps[11] = 24'h068DB8; // 0.0512 * 2^23
assign taps[12] = 24'h073EAB; // 0.0566 * 2^23
assign taps[13] = 24'h07CED9; // 0.061 * 2^23
assign taps[14] = 24'h083126; // 0.064 * 2^23
assign taps[15] = 24'h086594; // 0.0656 * 2^23
assign taps[16] = 24'h086594; // 0.0656 * 2^23
assign taps[17] = 24'h083126; // 0.064 * 2^23
assign taps[18] = 24'h07CED9; // 0.061 * 2^23
assign taps[19] = 24'h073EAB; // 0.0566 * 2^23
assign taps[20] = 24'h068DB8; // 0.0512 * 2^23
assign taps[21] = 24'h05C5D6; // 0.0451 * 2^23
assign taps[22] = 24'h04ED91; // 0.0385 * 2^23
assign taps[23] = 24'h041205; // 0.0318 * 2^23
assign taps[24] = 24'h03404E; // 0.0254 * 2^23
assign taps[25] = 24'h027EF9; // 0.0195 * 2^23
assign taps[26] = 24'h01D495; // 0.0143 * 2^23
assign taps[27] = 24'h014AF4; // 0.0101 * 2^23
assign taps[28] = 24'h00DED2; // 0.0068 * 2^23
assign taps[29] = 24'h009374; // 0.0045 * 2^23
assign taps[30] = 24'h006594; // 0.0031 * 2^23
assign taps[31] = 24'h0051EB; // 0.0025 * 2^23
```

- Các hằng số được lấy từ code trong Matlab, và được gán giá trị vào các tap từ tap0 đến tap31

- Khi có cạnh lên của clk hoặc cạnh xuống của reset:

```
// circular buffer control and input sample management
always @(posedge clk or negedge reset)
begin
    if (reset == 1'b0) begin
        buff_cnt <= 5'd0;
        enable_fir <= 1'b0;
        in_sample <= 24'd0;
    end else if (buff_cnt == TAP_COUNT-1) begin
        buff_cnt <= 5'd0;
        enable_fir <= 1'b1;
        in_sample <= fir_input;
    end else begin
        buff_cnt <= buff_cnt + 1;
        in_sample <= fir_input;
    end
end
end
```

- Nếu tín hiệu reset có cạnh xuống (reset == 1'b0), các biến buff_cnt, enable_fir, và in_sample sẽ được khởi tạo lại.
- Nếu buff_cnt đạt giá trị tối đa (== TAP_COUNT-1), nghĩa là bộ đệm đã đầy, buff_cnt sẽ được đặt lại về 0, enable_fir sẽ bật, và in_sample sẽ được gán giá trị mới từ fir_input.
- Với tất cả các trường hợp khác, buff_cnt sẽ tăng thêm 1 đơn vị và in_sample sẽ được cập nhật giá trị mới từ fir_input.
- - Khi có cạnh lên của clk:

```
// buffer update
always @(posedge clk or negedge reset)
begin
    if (reset == 1'b0)
        enable_buff <= 1'b0;
    else if (fir_ready == 1'b0)
        enable_buff <= 1'b0;
    else
        enable_buff <= 1'b1;
end
end
```

- Nếu có cạnh xuống của reset hoặc tín hiệu xác nhận fir_ready không hợp lệ (reset == 1'b0 và fir_ready == 1'b0) thì tín hiệu enable_buff được gán là 1'b0, nghĩa là lúc đó nó không sẵn sàng nhận hoặc gửi dữ liệu, và bộ đệm không được kích hoạt.
 - Trong trường hợp ngược lại, các tín hiệu trên đều hợp lệ, thì tín hiệu enable_buff được gán giá trị 1'b1, nghĩa là đã sẵn sàng nhận và gửi dữ liệu, và bộ đệm được kích hoạt.
- Sau khi khai báo và đặt điều kiện cần thiết để thực hiện bộ lọc, ta bắt đầu với việc update bộ đệm khi có input vào:

```
always @(posedge clk)
begin
integer i;
    if (enable_buff == 1'b1) begin
        buffer[0] <= in_sample;
        for (i = 1; i < TAP_COUNT; i = i + 1) begin
            buffer[i] <= buffer[i-1]; // Shift buffer values
        end
    end
end
```

Sau khi update bộ đệm, ta sẽ thực hiện phép nhân với các coeff đã được khai báo bên trên:

```
always @(posedge clk)
begin
integer i;
    if (enable_fir == 1'b1) begin
        for (i = 0; i < TAP_COUNT; i = i + 1) begin
            acc[i] <= taps[i] * buffer[i]; // Perform multiplication
        end
    end
end
```

- Với $acc_0, acc_1, \dots, acc_{14}$ là kết quả của các bộ nhân trước đó, ta thực hiện bộ cộng cuối cùng để tìm ra giá trị của tín hiệu sau khi đã qua bộ lọc:

```
always @(posedge clk)
begin
integer i;
|   if (enable_fir == 1'b1) begin
|       sum <= 48'd0; // Reset sum
|       for ( i = 0; i < TAP_COUNT; i = i + 1) begin
|           sum <= sum + acc[i]; // Accumulate all products
|       end
|   end
end
end
```

- Cuối cùng, ta rút gọn (làm tròn) tín hiệu sau khi lọc bằng cách lấy 24bit cao ([31:8]) bao gồm bit dấu của tín hiệu ngõ ra để trở thành `data_out` với độ dài phù hợp với cài đặt đã cho trước.

```
always @(posedge clk)
begin
|   fir_output <= sum[31:8]; // truncate and round the result to fit into 24 bits
|   end
endmodule
```

- Bộ lọc FIR về cơ bản đã được hoàn thành, tiếp theo ta sẽ chạy mô phỏng để kiểm tra kết quả.

III. Testbench mô phỏng

1. Testbench

- FIR_tb là module testbench dùng để mô phỏng với localparam là hằng số cục bộ được giữ nguyên trong quá trình hoạt động, cụ thể như sau:

```
module FIR_tb ();
    localparam FILE_PATH = "sine.hex";
    localparam OUT_PATH  = "output.hex";
    localparam FREQ = 100_000_000;

    localparam WD_IN      = 24                ; // Data width
    localparam WD_OUT     = 24                ;
    localparam PERIOD     = 1_000_000_000/FREQ;
    localparam HALF_PERIOD = PERIOD/2        ;
```

- FILE_PATH và OUT_PATH là đường dẫn của tệp sine.hex và output.hex, lần lượt là các mẫu dữ liệu đầu vào và dữ liệu đầu ra sau khi được xử lý bởi bộ lọc FIR mô phỏng.
- FREQ là **tần số xung nhịp** mô phỏng, ở đây được đặt là 100MHz.
- WD_IN và WD_OUT là **chiều rộng** của dữ liệu đầu vào và dữ liệu đầu ra, ở đây là 24 bits.
- PERIOD là chu kỳ xung nhịp tính bằng nano giây, được tính bằng công thức 1 giây chia cho tần số. HALF_PERIOD là nửa chu kỳ xung nhịp bằng $\frac{1}{2}$ PERIOD.

- Tiếp theo, ta khai báo các chân và các tín hiệu được sử dụng trong module cùng các biến số nguyên dùng để thao tác với file, sau đó là chuyển đổi giữa các kiểu dữ liệu (số thực, số có dấu) để dễ dàng tính toán hoặc so sánh:

```
// Testbench signals
reg          clk, reset_n;
reg [WD_IN-1:0] data_in ;
wire [WD_OUT-1:0] data_out;
reg fir_ready;
integer file, status, outfile;

real analog_in, analog_out;
assign analog_in  = $itor($signed(data_in));
assign analog_out = $itor($signed(data_out));
```

- Tiếp theo tạo tín hiệu clock xung vuông bằng cách đảo được tín hiệu clk mỗi nửa chu kỳ.

```
// Clock generation
always #HALF_PERIOD clk = ~clk;
```

- Khởi tạo các tín hiệu đầu vào (reset về 0) và đặt tín hiệu reset về 1 mỗi một chu kỳ, tín hiệu fir_ready được đặt về 1 thể hiện bộ lọc sẵn sàng nhận dữ liệu.

```
// Test procedure
initial begin
    // Initialize inputs
    clk      = 0;
    reset_n  = 0;
    data_in  = 0;

    // Apply reset
    #PERIOD reset_n = 1; // Deassert reset after a period
    fir_ready = 1;
```

- Điều kiện dừng mô phỏng sau 100 đơn thời gian và đóng file, sau đó sử dụng tín hiệu monitor để in ra sự thay đổi của các tín hiệu reset, data_in và data_out tại từng thời điểm. Điều này để có thể debug dễ dàng.

```
// Wait for a while to observe output
#100 $finish; // Stop simulation after 100 time units
$fclose(file);
$fclose(outfile);
end

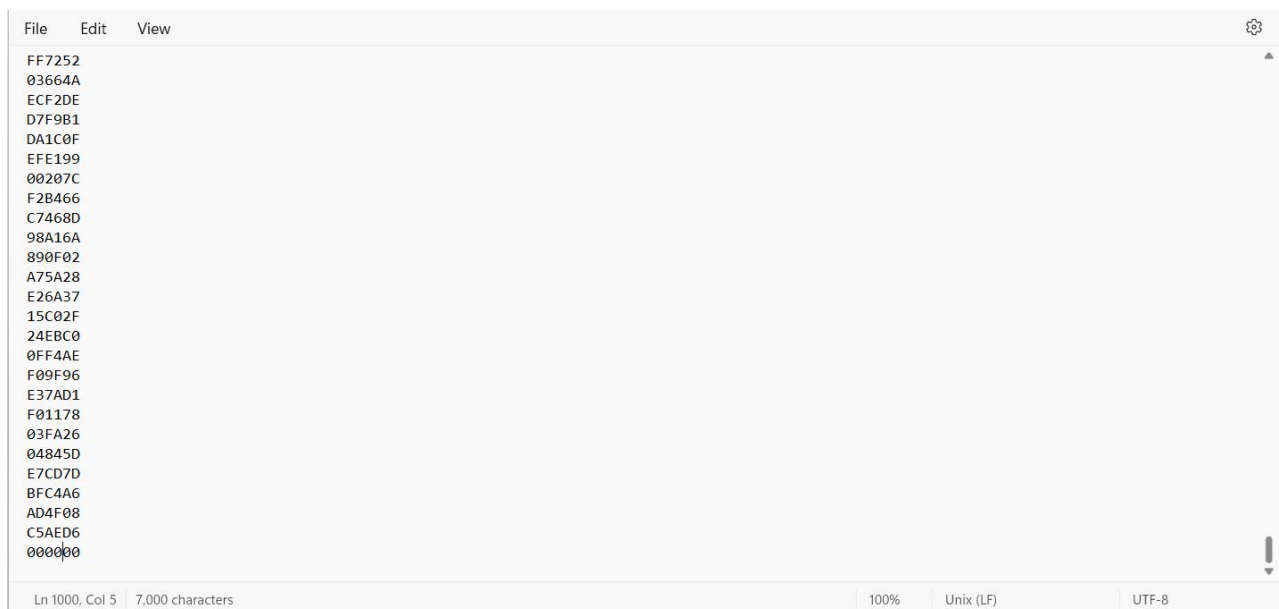
// Monitor signals for debugging
initial begin
    $monitor("Time = %0t | Reset = %b | Data In = %h | Data Out = %h",
    | $time, reset_n, data_in, data_out);
end

endmodule
```


- Cuối cùng ta gọi module `fir_filter` để tiến hành quá trình lọc nhiễu

```
// Instantiate the FIR filter module
fir_filter dut (
    .clk      (clk      ),
    .reset    (reset_n ),
    .fir_ready(fir_ready),
    .fir_input(data_in ),
    .fir_output(data_out)
);
```

- Ở phía trên coi như đã làm xong việc khai báo và gọi module bộ lọc, tiếp theo ta sẽ đưa tín hiệu sóng sin vào để mô phỏng lọc nhiễu và kiểm tra kết quả:



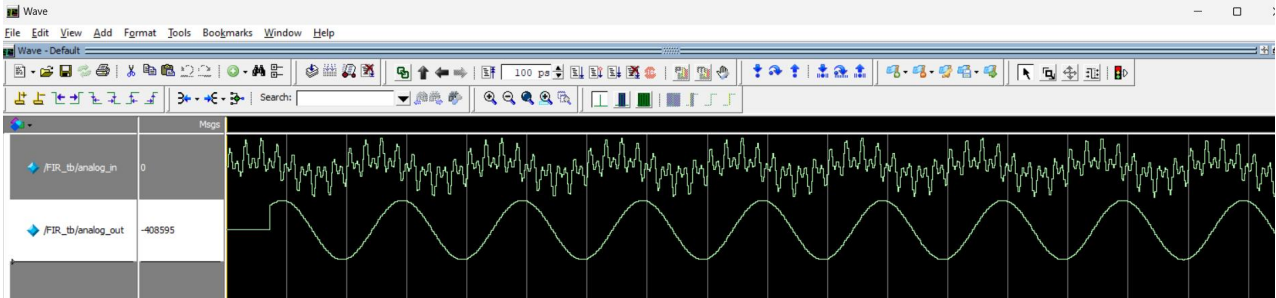
- Đọc file `sine.hex` trên và khởi tạo file `.hex` đầu ra, báo lỗi nếu không thể mở file. Sử dụng vòng lặp để đọc dữ liệu đầu vào và ghi các dữ liệu đầu ra.

```
// Read hex file
file = $fopen(FILE_PATH,"r");
outfile = $fopen(OUT_PATH, "w");
if (file == 0)    $error("Hex file not opened");
if (outfile == 0) $error("Output file not opened");
do begin
    status = $fscanf(file, "%h", data_in);
    @(posedge clk);
    $fdisplay(outfile, "%h", data_out);
end while (status != -1);
```

2. Kết quả mô phỏng

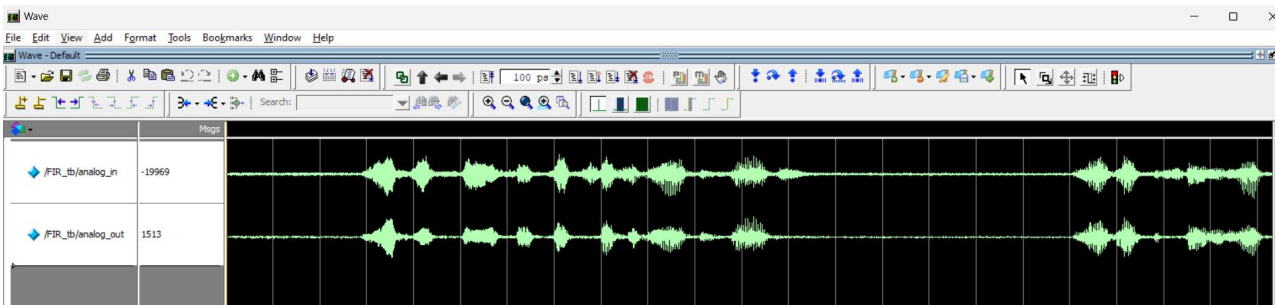
- Dưới đây là kết quả dạng sóng mô phỏng bằng ModelSim:

- Mô phỏng với mẫu sine.hex



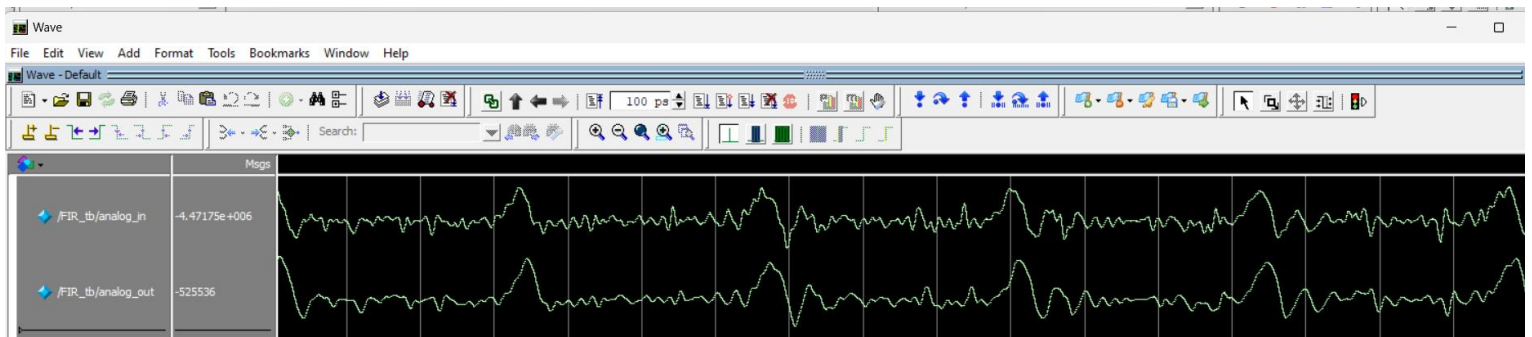
Hình 5. Dạng sóng input (phía trên, còn nhiễu) và dạng sóng output (phía dưới, đã lọc bớt nhiễu) của mẫu sine.hex

- Mô phỏng với mẫu audio.wav (audio.hex)



Hình 6. Dạng sóng input và dạng sóng output của mẫu audio.wav

- Mô phỏng với mẫu ecg.hex



Hình 7. Dạng sóng input và dạng sóng output của mẫu ecg.hex

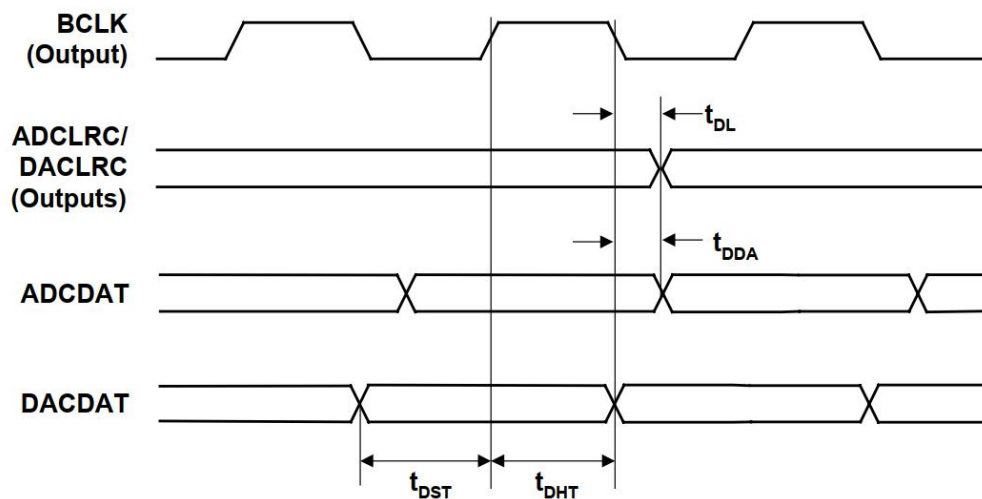
- Ta có thể thấy dạng sóng output khi đi qua bộ lọc FIR đã được lọc nhiễu và mịn hơn rất nhiều, chứng minh bộ lọc có thể lọc tín hiệu đầu vào đúng như mục tiêu.

IV. Kết nối phần cứng (Dự định thực hiện trên kit FPGA)

1. Serial To Parallel

a. Cơ sở lý thuyết

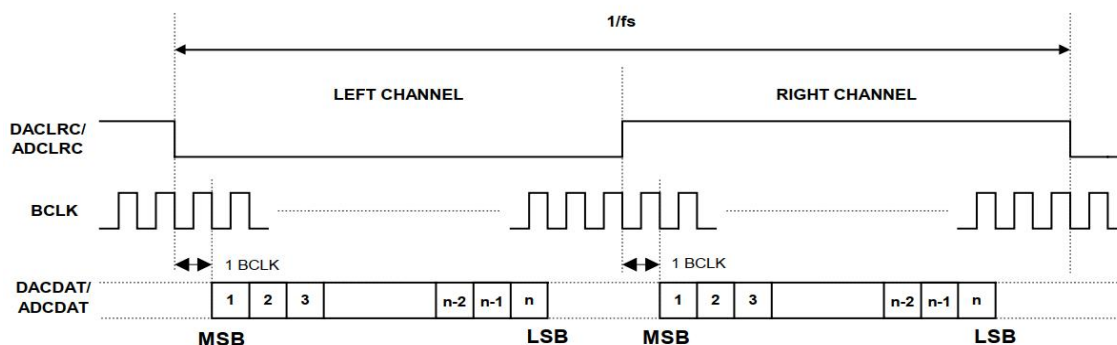
- Với tín hiệu âm thanh ngõ vào, để lấy 24bit data ta phải thực hiện lấy 24 mẫu 1bit từ chân AUD_ADCDAT:



Hình 8. BCLK trong FPGA

- Có thể thấy khi BCLK có cạnh xuống, các tín hiệu ADCDAT sẽ thực hiện lấy 1 bit từ tín hiệu ngõ vào, nên ta sử dụng BCLK và một thanh ghi 24bit để nhận lấy tín hiệu âm thanh, sau đó sử dụng bộ lọc FIR để lọc âm thanh đầu vào đó.

- Việc lấy mẫu sẽ được bắt đầu thực hiện dựa vào giá trị xung clock LRCLK được sử dụng cho bộ lọc FIR:



Hình 9. LRCLK trong FPGA

- Có thể thấy, mỗi khi LRCLK thay đổi giá trị logic, sau 1 chu kỳ BCLK thì tín hiệu âm thanh sẽ bắt đầu được lấy mẫu, vậy ta sẽ bắt đầu thực hiện việc lấy mẫu sau mỗi khi tín hiệu LRCLK thay đổi sau thời gian 1 xung BCLK.

2. Parallel To Serial

a. Cơ sở lý thuyết

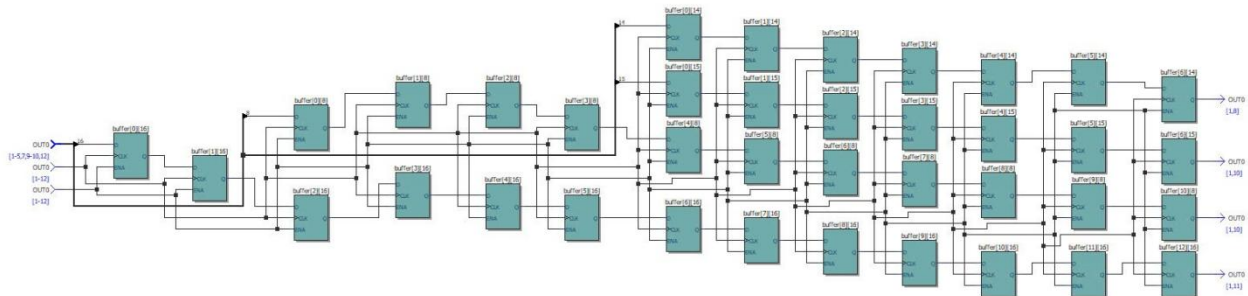
Ở đây ta cũng thực hiện tương tự với bộ Serial to Parallel, với input là 24bit đã được lọc từ FIR và output là từng bit tín hiệu âm thanh để đưa đến AUD_DACDAT.

3. Wrapper

a. Cơ sở lý thuyết

Từ file.bdf đã được cung cấp, ta thực hiện module wrapper khai báo các chân cho trước, kết nối các chân theo kí hiệu để có thể thực hiện giao tiếp với FPGA. Ở đây ta cần bổ sung bộ lọc FIR bằng cách gán chân AUD_ADCDAT vào bộ Serial to Parallel, lấy giá trị ngõ ra đưa vào bộ lọc FIR. Sau đó thực hiện Parallel to Serial để đưa tín hiệu đã được lọc vào chân AUD_DACDAT.

Toàn bộ quá trình được thực hiện bởi Quartus, sẽ cho ra được RTL viewer như sau:



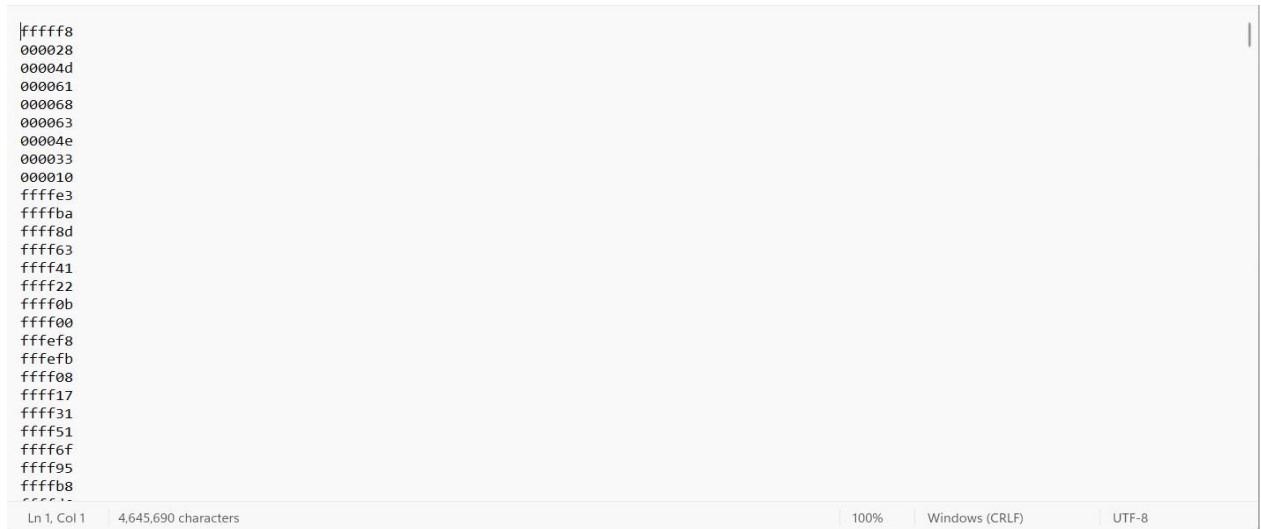
Hình 10. RTL Viewer của wrapper

V. Kết quả

- Dưới đây là kết quả của nhóm sau khi chạy code và dùng ModelSim mô phỏng của riêng mẫu **audio.wav** (audio.hex):

- Đưa file output.hex vào code Python hex2audio.py để chuyển từ file định dạng .hex thành file âm thanh định dạng .wav có thể nghe

Lưu ý: Cần xoá bớt khoảng đợi của tín hiệu



```
fffff8
000028
00004d
000061
000068
000063
00004e
000033
000010
ffffe3
ffffba
ffff8d
ffff63
ffff41
ffff22
ffff0b
ffff00
fffe8
fffeb
fff08
fff17
fff31
fff51
fff6f
fff95
fffb8
```

Hình 11. File tín hiệu đầu ra dưới dạng .hex

```

E: > Quartus > hex2audio.py > ...
1 import numpy as np
2 import wave
3
4 def hex_to_pcm(hex_file_path, bits=24):
5     with open(hex_file_path, 'r') as file:
6         hex_lines = file.readlines()
7
8     pcm_data = []
9     for line in hex_lines:
10        hex_value = line.strip()
11        if hex_value:
12            int_value = int(hex_value, 16)
13            if int_value >= (1 << bits-1):
14                int_value -= (1 << bits)
15            pcm_data.append(int_value)
16
17    return np.array(pcm_data, dtype=np.int32)
18
19 def pcm_to_wav(pcm_data, output_wav_path, sample_rate=44100):
20     # Normalize PCM data to fit in 16-bit range
21     max_val = np.max(np.abs(pcm_data))
22     pcm_data = np.int16(pcm_data * (32767 / max_val)) # Scale to 16-bit PCM
23
24     with wave.open(output_wav_path, 'w') as wav_file:
25         num_channels = 1
26         sampwidth = 2 # 16 bits
27         nframes = len(pcm_data)
28         comptype = "NONE"
29         compname = "not compressed"
30
31         wav_file.setparams((num_channels, sampwidth, sample_rate, nframes, comptype, compname))
32         wav_file.writeframes(pcm_data.tobytes())
33
34 hex_file_path = 'output2.hex'
35 output_wav_path = 'audio_filtered.wav'
36
37 # Convert hex file to PCM data

```

Hình 12. Code Python chuyển từ .hex về .wav

- Cuối cùng là link drive chứa file âm thanh trước và sau khi lọc của nhóm

<https://drive.google.com/drive/folders/15ar-LhFDwbIx9F-gm4k3hLP59TTdpbS3?usp=sharing>

KẾT LUẬN

Trong đề tài này, nhóm đã trình bày cơ sở lý thuyết về bộ lọc FIR truyền thống và lý do và mục tiêu khi sử dụng bộ lọc này trong việc lọc nhiễu. Qua đó, nhóm đã phát triển thành công chương trình sử dụng bộ lọc FIR để xử lý nhiễu âm thanh, đồng thời xây dựng chương trình wrapper kết nối với chương trình mô phỏng, sau đó tiến hành test trên kit DE2. Nhóm đã mô phỏng tín hiệu một cách hiệu quả bằng cách chạy chương trình bộ lọc FIR trên phần mềm Matlab, điều chỉnh các tham số của bộ lọc và áp dụng các kỹ thuật thiết kế tối ưu. Kết quả là, nhóm đã lọc được nhiễu và cho ra tín hiệu mịn hơn ban đầu rất nhiều, đáp ứng được việc cải thiện chất lượng âm thanh đầu ra. Tuy nhiên, bên cạnh đó vẫn còn một số nhiễu trong tín hiệu đầu ra nhóm chưa hoàn toàn loại bỏ được, vì vậy dự định của nhóm để phát triển bộ lọc FIR này là: **Tăng số tap hoặc tăng tần số lấy mẫu để lọc chính xác hơn nữa.**

TÀI LIỆU THAM KHẢO

<https://doelab.github.io/posts/DSPonFPGALab1/>

https://github.com/doelab/EE3041_DSPonFPGA

<https://vhdlwhiz.com/part-2-finite-impulse-response-fir-filters/>

<https://de.mathworks.com/help/dsphdl/ug/hdl-filter-architectures.html>