ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
BỘ MÔN ĐIỆN TỬ

BÀI TẬP LỚN MÔN XỬ LÝ TÍN HIỆU SỐ VỚI FPGA

LAB 1:

THỰC HIỆN TẠO MÁY PHÁT SÓNG BẰNG NGÔN NGỮ MÔ TẢ PHẦN CỨNG VÀ KIT FPGA DE10 STANDARD

LỚP L01 --- NHÓM 05 --- HK242

NGÀY NỘP 09/03/2025

Giảng viên hướng dẫn: Nguyễn Tuấn Hùng

Sinh viên thực hiện	Mã số sinh viên	Điểm số
Phương Hiển Long	2211898	
Dương Tiến Dũng	2012858	
Huỳnh Kiến Hào	2210857	

Thành phố Hồ Chí Minh – 2025

LỜI NÓI ĐẦU

Trong thời đại công nghệ số, FPGA ngày càng khẳng định vai trò quan trọng trong việc xử lý tín hiệu nhờ khả năng linh hoạt và hiệu suất cao. Một trong những ứng dụng thực tế của FPGA là tạo và xử lý tín hiệu âm thanh, phục vụ nghiên cứu và phát triển trong nhiều lĩnh vực như điện tử, viễn thông và tự động hóa.

Báo cáo này trình bày quá trình thiết kế và triển khai máy tạo sóng trên KIT FPGA DE10, trong đó FPGA sẽ tạo ra các dạng sóng như sin, tam giác, vuông,... trong đó có thể chèn nhiễu vào tính hiệu theo ý muốn và xuất tín hiệu ra ngoài thông qua bộ chuyển đổi WM8731 bằng giao tiếp I2C. Dự án này không chỉ giúp hiểu sâu hơn về lập trình phần cứng với Verilog, mà còn là cơ hội để áp dụng các giao thức giao tiếp quan trọng trong hệ thống nhúng.

Trong quá trình thực hiện, chúng em đã gặp không ít thách thức, từ việc thiết kế bộ tạo sóng, cấu hình WM8731 cho đến kiểm tra tín hiệu trên dao động ký. Tuy nhiên, chính những khó khăn này đã giúp chúng tôi có thêm nhiều kinh nghiệm thực tế, đồng thời củng cố kiến thức về FPGA và xử lý tín hiệu số.

Hy vọng rằng báo cáo này không chỉ giúp tổng kết lại quá trình nghiên cứu mà còn có thể trở thành tài liệu tham khảo hữu ích cho những ai quan tâm đến chủ đề này. Xin gửi lời cảm ơn chân thành đến các thầy và bạn bè đã hỗ trợ trong suốt quá trình thực hiện.

MỤC LỤC

MỤC TIÊU BÀI TẬP LỚN

Trong bài tập lớn này, chúng em sẽ thiết kế một bộ tạo dạng sóng có khả năng tạo ra nhiều dạng sóng khác nhau như sin, vuông, tam giác, răng cưa, ECG và nhiễu. Bộ tạo sóng sẽ cho phép điều chỉnh các thông số quan trọng bao gồm: tần số, biên độ và duty_cycle (nếu có tùy theo dạng sóng).

Ngoài ra, hệ thống cũng cần có khả năng chèn nhiễu vào các dạng sóng được tạo ra. Nói về nhiễu, bộ tạo nhiễu cũng phải có khả năng cho người sử dụng điều chỉnh các thông số về biên độ và tần số.

Máy tạo sóng này phải được viết bằng các ngôn ngữ mô tả phần cứng như Verilog, SystemVerilog hay VHDL.

Máy tạo sóng này phải được mô phỏng hoàn chỉnh trên các phần mềm mô phỏng như Modelsim hay Questasim.

Và cuối cùng, sẽ được thực hiện trong điều kiện thực tế bằng KIT FPGA DE10 Standard và hiển thị dạng sóng trên máy hiện sóng (Oscilloscope).

CÔNG CỤ SỬ DỤNG

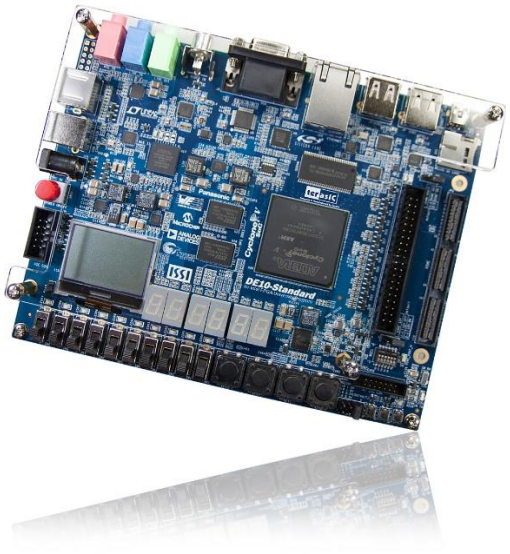
Công cụ sử dụng để thực hiện bài tập lớn này bao gồm phần mềm viết code mô tả phần cứng Quartus. Đây là một phần mềm thiết kế mạch số do Intel (trước đây là Altera) phát triển, được sử dụng rộng rãi trong lĩnh vực lập trình FPGA (Field Programmable Gate Array). Quartus cung cấp một môi trường tích hợp (IDE) mạnh mẽ để thiết kế, mô phỏng, tổng hợp và triển khai các thiết kế số trên FPGA. Quartus có nhiều chức năng quan trọng hỗ trợ kỹ sư và nhà phát triển trong quá trình thiết kế mạch số:

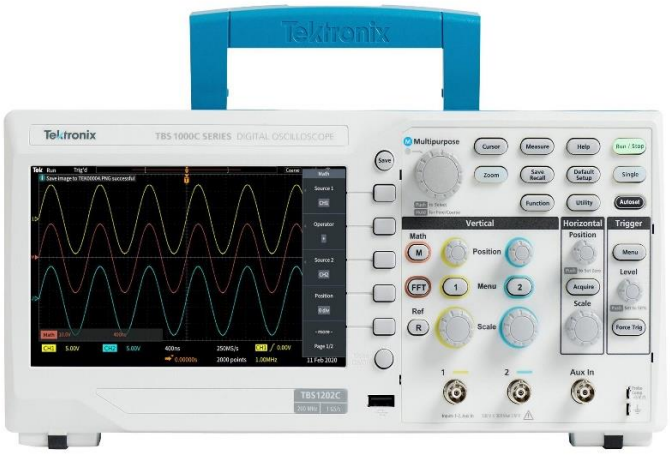
- Tổng hợp (Synthesis): Chuyển đổi mã RTL (Verilog/VHDL) thành sơ đồ cổng logic tối ưu để triển khai trên FPGA.
- Mô phỏng và kiểm tra (Simulation & Verification): Hỗ trợ mô phỏng hành vi của thiết kế để kiểm tra tính chính xác trước khi tải lên FPGA.
- Bố trí và định tuyến (Placement & Routing): Tối ưu hóa vị trí các phần tử logic trên FPGA để đạt hiệu suất cao nhất.
- Tích hợp phân tích thời gian (Timing Analysis): Giúp đảm bảo thiết kế đáp ứng các yêu cầu về tốc độ và độ trễ.
- Hỗ trợ Debugging: Cung cấp các công cụ như SignalTap Logic Analyzer để kiểm tra tín hiệu bên trong FPGA trong thời gian thực.
- Hỗ trợ lập trình và cấu hình FPGA: Cho phép nạp bitstream trực tiếp lên thiết bị FPGA thông qua JTAG hoặc phương thức cấu hình khác.

Quartus là một công cụ mạnh mẽ và toàn diện cho việc thiết kế FPGA, cung cấp đầy đủ các chức năng từ lập trình, mô phỏng đến kiểm tra và triển khai hệ thống. Với giao diện thân thiện, khả năng tối ưu hóa cao và sự hỗ trợ từ cộng đồng,

Thứ hai là các phần mềm mô phỏng kết quả như Modelsim và Questasim được sử dụng rộng rãi trong công nghiệp, nghiên cứu và giáo dục, đặc biệt trong thiết kế vi mạch, hệ thống nhúng và FPGA. Với khả năng mô phỏng chính xác và công cụ kiểm tra mạnh mẽ, đây là hai phần mềm không thể thiếu đối với các kỹ sư phần cứng. Trong bài tập lớn này chúng em sẽ viết các testbench mô phỏng thử các kết quả của các module, và hiển thị kết quả của cúng thông qua Modelsim và Questasim.

Cuối cùng là KIT FPGA DE10 STANDARD và máy hiện sóng (Oscilloscope) để thực hiện tạo sóng và hiện dạng sóng từ chương trình đã code trong điều kiện thực tế.

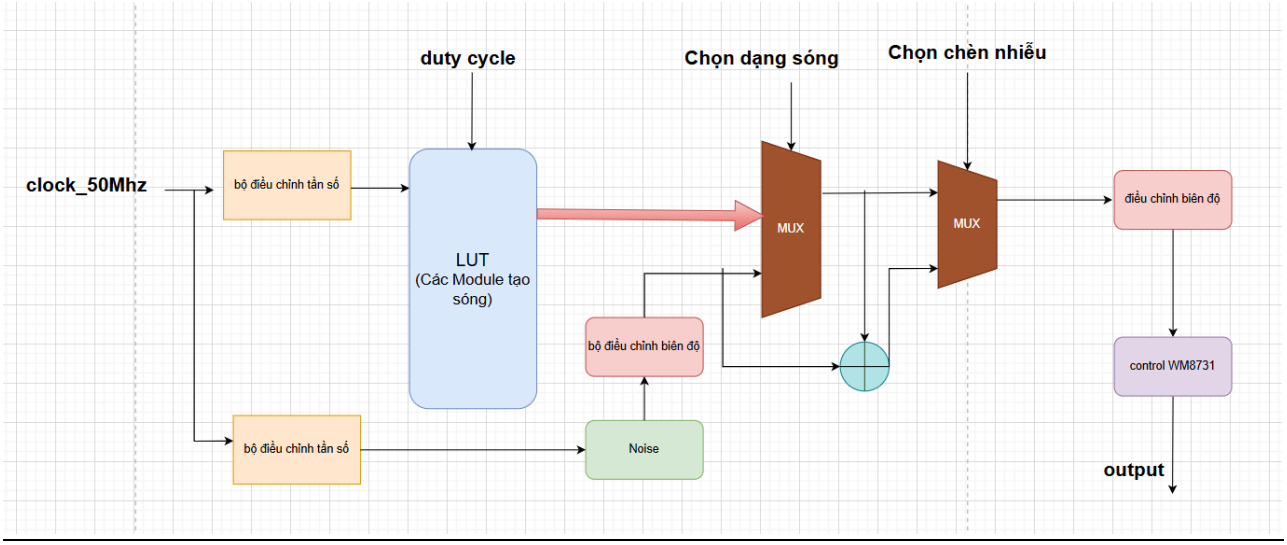




NỘI DUNG BÁO CÁO

I. Sơ đồ tổng quát của máy phát sóng

1. Nguyên lý hoạt động của mạch:



Như sơ đồ ở trên ta có thể thấy rằng mạch máy phát sóng là sự kết hợp của các module con tạo sóng, các bộ mux để lựa chọn các trường hợp, bộ cộng với nhiệm vị chèn nhiễu vào các sóng được tạo, các bộ đếm để chia tần số và chọn chế độ cho các bộ chia tần số và nhân biên độ của các sóng được tạo ra và nhiễu. Bộ dịch bit làm nhiệm vụ tăng tần só và DAC trong trường hợp này cụ thể là WM8731 để chuyển các tính hiệu số thành các dạng sóng analog liên tục để hiển thị trên máy hiện sóng.

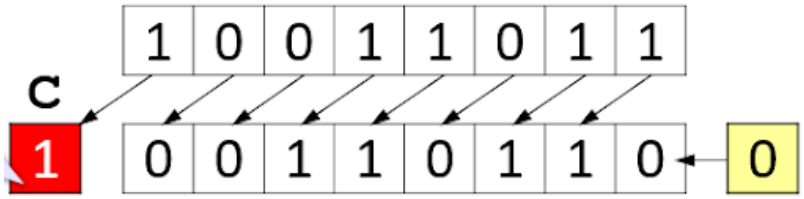
Như sơ đồ trên ta có thể gán các ngoại vi trên KIT DE10 với các chức năng của máy tạo sóng cụ thể như sau:

Ngoại vi	Chức năng
Button 0	Điều chỉnh biên độ sóng
Button 1	Điều chỉnh tần số sóng
Button 2	Điều chỉnh biên độ nhiễu
Button 3	Điều chỉnh tần số nhiễu
Switch 0	Reset
Switch 1,2,3	Chọn dạng sóng
Switch 4	Chọn chèn nhiễu
Switch 5,6,7,8	Điều chỉnh Duty_Cycle

2. Nguyên lý thay đổi biên độ, tần số, độ rộng xung:

2.1. Biên độ

Bộ khuếch đại trong thiết kế máy tạo sóng trên FPGA hoạt động dựa trên nguyên lý dịch bit để thay đổi biên độ tín hiệu. Cụ thể, tín hiệu đầu vào được biểu diễn dưới dạng số nhị phân và có thể được khuếch đại bằng cách dịch các bit sang trái. Khi dịch trái 1 bit, giá trị số học của tín hiệu sẽ tăng gấp đôi (tương đương nhân 2), và khi dịch trái 2 bit, giá trị sẽ tăng gấp bốn lần (tương đương nhân 4). Phương pháp này giúp thực hiện phép nhân một cách nhanh chóng và hiệu quả mà không cần sử dụng bộ nhân số học phức tạp, từ đó tiết kiệm tài nguyên phần cứng và cải thiện hiệu suất hệ thống. Việc điều chỉnh biên độ theo cách này giúp duy trì độ chính xác cao và tránh lỗi làm tròn, đồng thời đảm bảo tín hiệu đầu ra có biên độ phù hợp với yêu cầu ứng dụng.

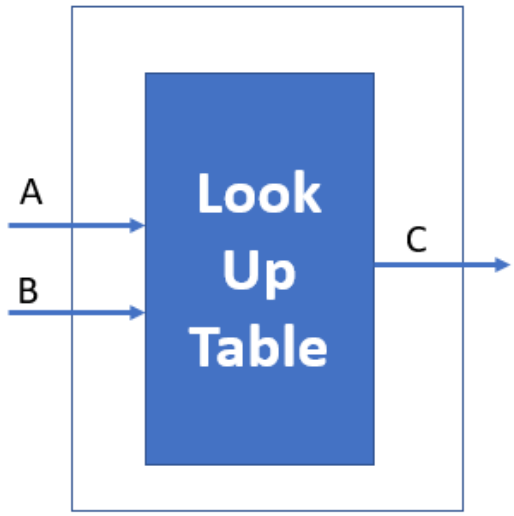


2.2. Tần số

Bộ chỉnh tần số trong máy tạo sóng trên FPGA hoạt động dựa trên nguyên tắc thay đổi bước nhảy địa chỉ khi đọc dữ liệu từ bảng tra (lookup table - LUT). Trong thiết kế này, tín hiệu sóng được lưu dưới dạng 1024 mẫu dữ liệu số, mỗi chu kỳ sóng hoàn chỉnh sẽ được tạo ra bằng cách quét tuần tự qua toàn bộ 1024 địa chỉ.

Mặc định, nếu mỗi chu kỳ tín hiệu tăng địa chỉ theo bước nhảy +1, FPGA sẽ đọc toàn bộ 1024 mẫu, tạo ra tần số cơ bản của tín hiệu đầu ra. Để tăng tần số của sóng, ta tăng bước nhảy địa chỉ lên +2 hoặc +4. Khi đó, số lượng mẫu thực sự được sử dụng trong một chu kỳ sóng giảm xuống còn 512 hoặc 256, dẫn đến việc hoàn thành một chu kỳ nhanh hơn, từ đó làm tăng tần số của tín hiệu đầu ra.

Phương pháp này có ưu điểm là đơn giản, hiệu quả và tận dụng tối đa tài nguyên FPGA mà không cần sử dụng phép chia phức tạp. Tuy nhiên, khi tăng bước nhảy, số lượng mẫu trên một chu kỳ giảm, có thể ảnh hưởng đến độ mịn của sóng, đặc biệt với tín hiệu sin, do số lượng điểm lấy mẫu bị giới hạn. Điều này có thể khắc phục bằng cách sử dụng nội suy hoặc tăng độ phân giải của bảng tra



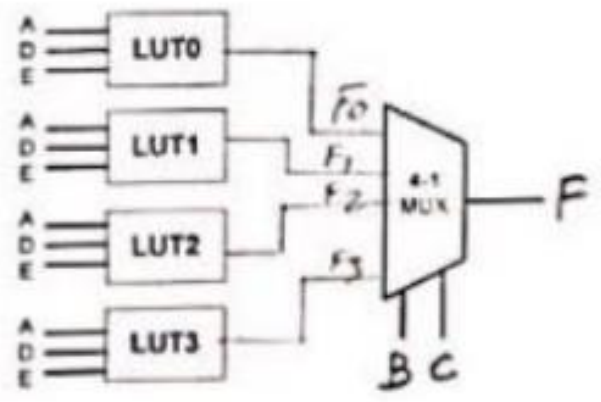
2.3 Độ rộng xung

Bộ điều chỉnh duty cycle trong máy tạo sóng trên FPGA được thiết kế bằng cách sử dụng 11 bảng tra (lookup table - LUT) tương ứng với 11 mức giá trị duty cycle: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% và 100%. Mỗi bảng chứa dữ liệu được tiền tính để tạo ra tín hiệu có tỷ lệ thời gian mức cao (HIGH) và mức thấp (LOW) phù hợp với giá trị duty cycle mong muốn.

Khi cần thay đổi duty cycle, hệ thống sẽ chọn bảng tra tương ứng với giá trị yêu cầu và sử dụng dữ liệu từ bảng đó để tạo tín hiệu đầu ra. Ví dụ:

- Với duty cycle 50%, tín hiệu xung vuông có thời gian mức cao và mức thấp bằng nhau.
- Với duty cycle 30%, tín hiệu ở mức cao trong 30% chu kỳ và mức thấp trong 70% chu kỳ.
- Với duty cycle 90%, tín hiệu ở mức cao trong 90% chu kỳ và mức thấp chỉ 10%.

Phương pháp này có ưu điểm là đơn giản, không yêu cầu tính toán phức tạp trong thời gian thực, giúp tiết kiệm tài nguyên phần cứng và đảm bảo tốc độ xử lý nhanh. Việc sử dụng các bảng tra cũng giúp tín hiệu đầu ra ổn định và chính xác, tránh lỗi làm tròn hoặc nhiễu không mong muốn. Bộ điều chỉnh duty cycle này có thể áp dụng cho các dạng sóng như xung vuông, tam giác và sawtooth, giúp máy tạo sóng linh hoạt trong việc tạo tín hiệu theo yêu cầu của ứng dụng.



II. Các module con

1. Module tạo sóng sine

Module tạo sóng sine của chúng em được tạo nên theo thủ thuật dùng Look-Up Table (LUT) tức là dùng một bảng có những giá trị có sẵn. Những giá trị có sẵn này được tạo ra theo một quy luật nhất định mà trong trường hợp này những điểm giá trị trên khi được hiển thị trên đồ thị sẽ tạo thành hình dạng sóng sine. Dựa theo ý tưởng này tụi em đã code Module tạo ra dạng sóng sine như bên dưới:

```
Module SineLUT (
Input logic [9:0]i_addr,
output logic [23:0] o_data );
parameter MEM_SIZE = 1024;
logic [23:0] LUT_sine [MEM_SIZE-1:0];
always_comb begin :
next_pc_ff o_data <= LUT_sine[i_addr[9:0]];
end : next_pc_ff
initial begin
integer i;
for (i=0; i < MEM_SIZE; i++) begin
LUT_sine[i] = 24'b0;
```

end

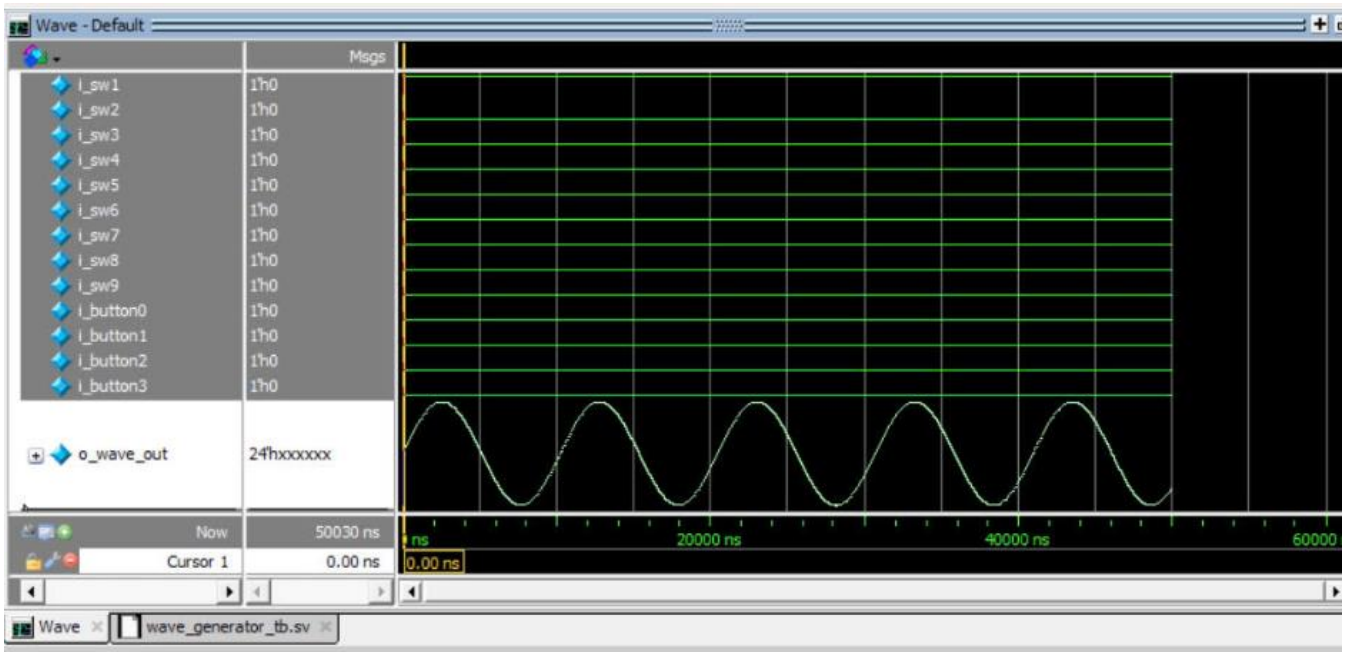
$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_sine.dump",LUT_sine);

end

endmodule

Theo như đoạn code trên thì Module tạo sóng sine này sẽ cho dạng sóng đầu ra có dạng sóng là 24 bit, số mẫu (số giá trị để tạo nên hình dạng sóng) có số lượng là 1024 mẫu để đảm rằng dạng sóng tạo nên sẽ đúng hình dạng, mượt mà và không bị bậc thang. Trong đó i_addr là 10-bit, dùng để chỉ định vị trí của giá trị trong LUT. LUT hoạt động như một **bộ nhớ chỉ đọc**, trả về giá trị o_data tương ứng với i_addr. Kết hợp với vòng lặp **for** có giới hạn là 1024 và lệnh **$readmemh** giúp giá trị được đọc từ file.dump bên ngoài đồng thời trả về ngõ ra một cách liên tục và lặp lại theo chu kỳ giúp tạo nên hình dạng sóng sine theo ý muốn.

Kết quả mô phỏng Module:



2. Module tạo sóng vuông

Đối với Module tạo dạng sóng vuông chúng em cũng áp dụng ý tưởng tương tự như sóng sine để tạo ra hình dạng sóng vuông. Nhìn thoáng qua sóng vuông tưởng chừng như sẽ dễ dàng đẻ tạo ra dạng sóng hơn khi bản chất của nó chỉ có vỏ vẹn 2 mức giá trị là mức cao và mức thấp. Tuy nhiên do sóng vuông là một sóng có bản chất như những xung nhịp nên có yếu tố về độ rộng xung (Duty_cycle) trong đó. Cho nên code Module sóng vuông của chúng em có tích hợp thêm cả chế độ điều chỉnh được độ rộng xung ngay trong chức năng tạo sóng của Module. Với việc này việc tạo nên Module tạo sóng vuông sẽ khó khăn hơn khi bây giờ chương trình tạo sóng phải có nhiều LUT tương ứng với những thông số về độ rộng xung khác nhau. Dưới đây là phần code của Module này:

```
module Square_waveLUT (
  input  logic [9:0] i_addr,
  input  logic [3:0] i_sel,
  output logic [23:0] o_data
);
  parameter MEM_SIZE = 1024;
  logic [23:0] LUT_square10  [MEM_SIZE-1:0];
  logic [23:0] LUT_square20  [MEM_SIZE-1:0];
  logic [23:0] LUT_square30  [MEM_SIZE-1:0];
  logic [23:0] LUT_square40  [MEM_SIZE-1:0];
  logic [23:0] LUT_square50  [MEM_SIZE-1:0];
  logic [23:0] LUT_square60  [MEM_SIZE-1:0];
  logic [23:0] LUT_square70  [MEM_SIZE-1:0];
  logic [23:0] LUT_square80  [MEM_SIZE-1:0];
  logic [23:0] LUT_square90  [MEM_SIZE-1:0];
  logic [23:0] LUT_square100 [MEM_SIZE-1:0];

  always_comb begin : next_pc
    case(i_sel)
      4'b0000: o_data = 24'b0;//duty_cycle 0%
          4'b0001: o_data = LUT_square10[i_addr[9:0]]; //duty_cycle 10%
          4'b0010: o_data = LUT_square20[i_addr[9:0]]; //duty_cycle 20%
          4'b0011: o_data = LUT_square30[i_addr[9:0]]; //duty_cycle 30%
```

```
            4'b0100: o_data = LUT_square40[i_addr[9:0]]; //duty_cycle 40%
            4'b0101: o_data = LUT_square50[i_addr[9:0]]; //duty_cycle 50%
            4'b0110: o_data = LUT_square60[i_addr[9:0]]; //duty_cycle 60%
            4'b0111: o_data = LUT_square70[i_addr[9:0]]; //duty_cycle 70%
            4'b1000: o_data = LUT_square80[i_addr[9:0]]; //duty_cycle 80%
            4'b1001: o_data = LUT_square90[i_addr[9:0]]; //duty_cycle 90%
            4'b1010: o_data = LUT_square100[i_addr[9:0]];//duty_cycle 100%
     default: o_data = 24'b0;
   endcase
 end : next_pc
 initial begin
   integer i;
    for (i=0; i < MEM_SIZE; i++) begin
      LUT_square10[i]  = 24'b0;
      LUT_square20[i]  = 24'b0;
            LUT_square30[i]  = 24'b0;
            LUT_square40[i]  = 24'b0;
            LUT_square50[i]  = 24'b0;
            LUT_square60[i]  = 24'b0;
            LUT_square70[i]  = 24'b0;
            LUT_square80[i]  = 24'b0;
            LUT_square90[i]  = 24'b0;
            LUT_square100[i] = 24'b0;
    end

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_square10.dump",LUT_square10);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_square20.dump",LUT_square20);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_square30.dump",LUT_square30);
```

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_square40.dump",LUT_square40);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_square50.dump",LUT_square50);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_square60.dump",LUT_square60);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_square70.dump",LUT_square70);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_square80.dump",LUT_square80);

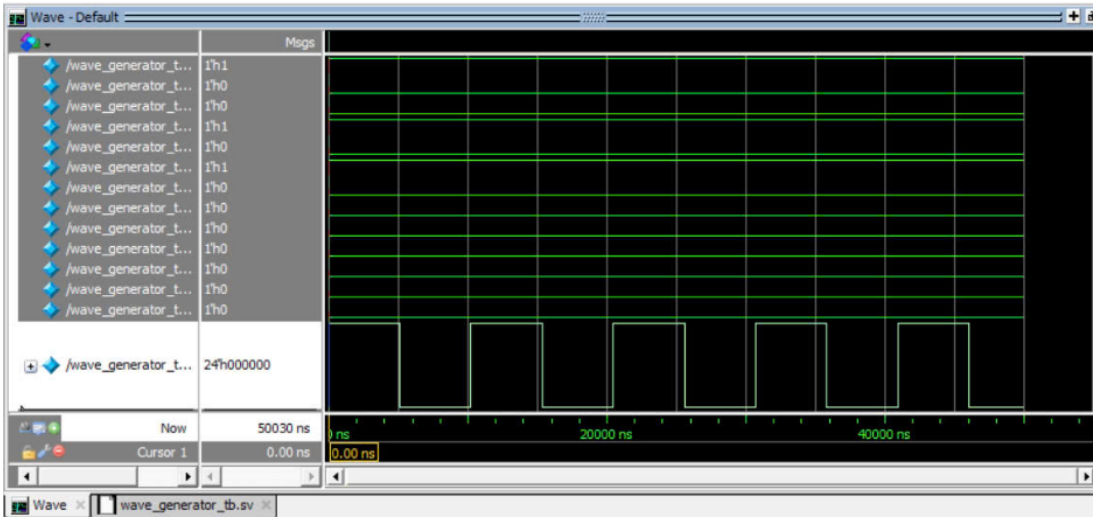
$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_square90.dump",LUT_square90);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_square100.dump",LUT_square100);
end

endmodule

Như ta thấy Module trên có cấu trúc gần giống như module tạo sóng sine nhưng lại có thêm **i_sel** để chọn dạng sóng vuông được xuất ra theo điều chỉnh độ rộng xung. Đồng thời lệnh đọc file **$readmemh** được dùng nhiều lần để lấy giá trị từ các file.dump khác nhau.

Kết quả mô phỏng Module:



3. Module tạo sóng tam giác, răng cưa

Đối với Module sóng tam giác và răng cưa thì ý tưởng hoạt động giống như hai Module trên tức cũng sử dụng LUT để tạo ra hình dạng sống thích hợp và tích hợp các Switch gạt để điều chỉnh độ rộng xung như sóng vuông.Điều duy nhất khác biết là nằm ở các giá trị được lưu trong LUT của các dạng sóng. Trong khi các giá trị LUT của sóng tam giác sẽ được tạo ra và có mối quan hệ tuyến tính tăng dần với nhau đến một giá trị đạt đỉnh nào đó rồi sẽ giảm dần tuyến tính y hệt như thế để tạo nên hai cạnh bên tam giác. Thì các giá trị LUT của sóng răng cưa sẽ tăng tuyến tính đến giá trị max rồi lặp tức về 0 ở giá trị tiếp theo để tạo nên hình dạng răng cưa. Vòng lặp **for** cũng sẽ được dùng trong 2 module này để tạo ra các chu kỳ sóng nối tiếp nhau theo yêu cầu.

Code Module tạo sóng tam giác:

```
module Triangle_waveLUT (
  input  logic [9:0] i_addr,
  input  logic [3:0] i_sel,
  output logic [23:0] o_data
);
  parameter MEM_SIZE = 1024;
  logic [23:0] LUT_triangle10  [MEM_SIZE-1:0];
  logic [23:0] LUT_triangle20  [MEM_SIZE-1:0];
  logic [23:0] LUT_triangle30  [MEM_SIZE-1:0];
  logic [23:0] LUT_triangle40  [MEM_SIZE-1:0];
```

```
logic [23:0] LUT_triangle50  [MEM_SIZE-1:0];
logic [23:0] LUT_triangle60  [MEM_SIZE-1:0];
logic [23:0] LUT_triangle70  [MEM_SIZE-1:0];
logic [23:0] LUT_triangle80  [MEM_SIZE-1:0];
logic [23:0] LUT_triangle90  [MEM_SIZE-1:0];
logic [23:0] LUT_triangle100 [MEM_SIZE-1:0];

always_comb begin : next_pc
  case(i_sel)
    4'b0000: o_data = 24'b0;//duty_cycle 0%
          4'b0001: o_data = LUT_triangle10[i_addr[9:0]]; //duty_cycle 10%
          4'b0010: o_data = LUT_triangle20[i_addr[9:0]]; //duty_cycle 20%
          4'b0011: o_data = LUT_triangle30[i_addr[9:0]]; //duty_cycle 30%
          4'b0100: o_data = LUT_triangle40[i_addr[9:0]]; //duty_cycle 40%
          4'b0101: o_data = LUT_triangle50[i_addr[9:0]]; //duty_cycle 50%
          4'b0110: o_data = LUT_triangle60[i_addr[9:0]]; //duty_cycle 60%
          4'b0111: o_data = LUT_triangle70[i_addr[9:0]]; //duty_cycle 70%
          4'b1000: o_data = LUT_triangle80[i_addr[9:0]]; //duty_cycle 80%
          4'b1001: o_data = LUT_triangle90[i_addr[9:0]]; //duty_cycle 90%
          4'b1010: o_data = LUT_triangle100[i_addr[9:0]];//duty_cycle 100%
    default: o_data = 24'b0;
  endcase
end : next_pc
initial begin
  integer i;
   for (i=0; i < MEM_SIZE; i++) begin
     LUT_triangle10[i]  = 24'b0;
     LUT_triangle20[i]  = 24'b0;
          LUT_triangle30[i]  = 24'b0;
          LUT_triangle40[i]  = 24'b0;
          LUT_triangle50[i]  = 24'b0;
          LUT_triangle60[i]  = 24'b0;
          LUT_triangle70[i]  = 24'b0;
          LUT_triangle80[i]  = 24'b0;
          LUT_triangle90[i]  = 24'b0;
```

```
            LUT_triangle100[i] = 24'b0;
        end

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_triangle10.dump",LUT_triangle10);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_triangle20.dump",LUT_triangle20);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_triangle30.dump",LUT_triangle30);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_triangle40.dump",LUT_triangle40);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_triangle50.dump",LUT_triangle50);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_triangle60.dump",LUT_triangle60);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_triangle70.dump",LUT_triangle70);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_triangle80.dump",LUT_triangle80);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_triangle90.dump",LUT_triangle90);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_triangle100.dump",LUT_triangle100);
  end

endmodule
```

Code Module tạo sóng răng cưa:

```
module Sawtooth_waveLUT (
  input  logic [9:0] i_addr,
  input  logic [3:0] i_sel,
  output logic [23:0] o_data
);
  parameter MEM_SIZE = 1024;
  logic [23:0] LUT_sawtooth10  [MEM_SIZE-1:0];
  logic [23:0] LUT_sawtooth20  [MEM_SIZE-1:0];
  logic [23:0] LUT_sawtooth30  [MEM_SIZE-1:0];
  logic [23:0] LUT_sawtooth40  [MEM_SIZE-1:0];
  logic [23:0] LUT_sawtooth50  [MEM_SIZE-1:0];
  logic [23:0] LUT_sawtooth60  [MEM_SIZE-1:0];
  logic [23:0] LUT_sawtooth70  [MEM_SIZE-1:0];
  logic [23:0] LUT_sawtooth80  [MEM_SIZE-1:0];
  logic [23:0] LUT_sawtooth90  [MEM_SIZE-1:0];
  logic [23:0] LUT_sawtooth100 [MEM_SIZE-1:0];

  always_comb begin : next_pc
    case(i_sel)
      4'b0000: o_data = 24'b0;//duty_cycle 0%
           4'b0001: o_data = LUT_sawtooth10[i_addr[9:0]]; //duty_cycle 10%
           4'b0010: o_data = LUT_sawtooth20[i_addr[9:0]]; //duty_cycle 20%
           4'b0011: o_data = LUT_sawtooth30[i_addr[9:0]]; //duty_cycle 30%
           4'b0100: o_data = LUT_sawtooth40[i_addr[9:0]]; //duty_cycle 40%
           4'b0101: o_data = LUT_sawtooth50[i_addr[9:0]]; //duty_cycle 50%
           4'b0110: o_data = LUT_sawtooth60[i_addr[9:0]]; //duty_cycle 60%
           4'b0111: o_data = LUT_sawtooth70[i_addr[9:0]]; //duty_cycle 70%
           4'b1000: o_data = LUT_sawtooth80[i_addr[9:0]]; //duty_cycle 80%
           4'b1001: o_data = LUT_sawtooth90[i_addr[9:0]]; //duty_cycle 90%
           4'b1010: o_data = LUT_sawtooth100[i_addr[9:0]];//duty_cycle 100%
     default: o_data = 24'b0;
    endcase
  end : next_pc
```

```
 initial begin
  integer i;
   for (i=0; i < MEM_SIZE; i++) begin
     LUT_sawtooth10[i]  = 24'b0;
     LUT_sawtooth20[i]  = 24'b0;
          LUT_sawtooth30[i]  = 24'b0;
          LUT_sawtooth40[i]  = 24'b0;
          LUT_sawtooth50[i]  = 24'b0;
          LUT_sawtooth60[i]  = 24'b0;
          LUT_sawtooth70[i]  = 24'b0;
          LUT_sawtooth80[i]  = 24'b0;
          LUT_sawtooth90[i]  = 24'b0;
          LUT_sawtooth100[i] = 24'b0;
   end
```

```
$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_sawtooth10.dump",LUT_sawtooth10);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_sawtooth20.dump",LUT_sawtooth20);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_sawtooth30.dump",LUT_sawtooth30);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_sawtooth40.dump",LUT_sawtooth40);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_sawtooth50.dump",LUT_sawtooth50);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(
new)/wave_generator/LUT_sawtooth60.dump",LUT_sawtooth60);
```

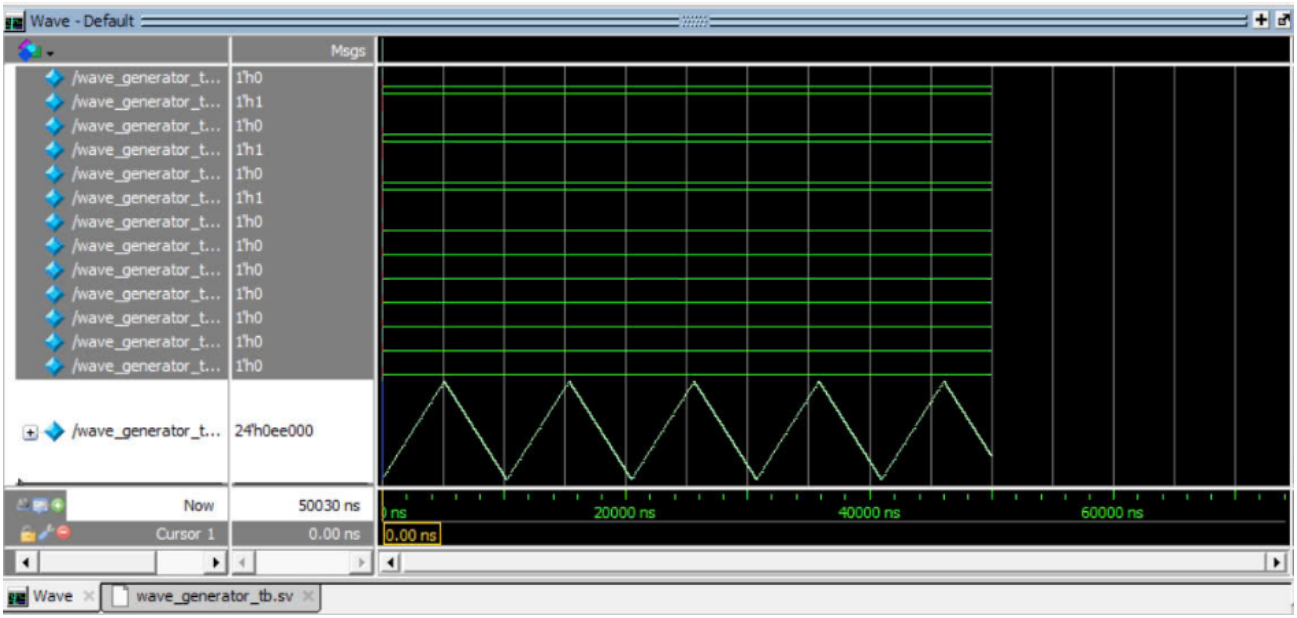
$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_sawtooth70.dump",LUT_sawtooth70);

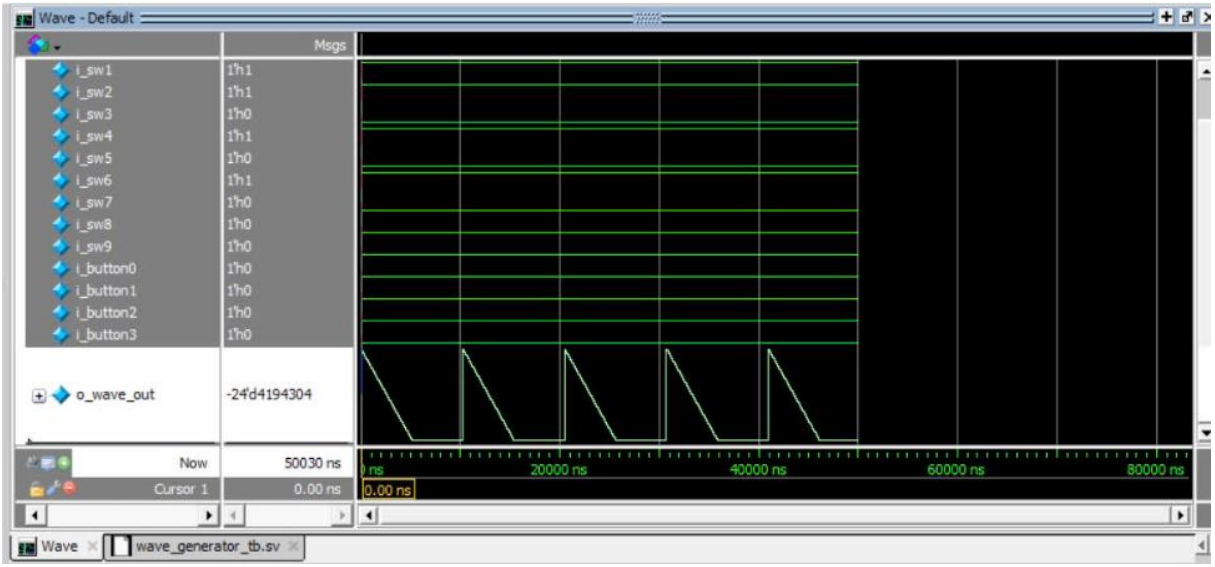
$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_sawtooth80.dump",LUT_sawtooth80);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_sawtooth90.dump",LUT_sawtooth90);

$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_sawtooth100.dump",LUT_sawtooth100);
end
endmodule

Kết quả mô phỏng Module:





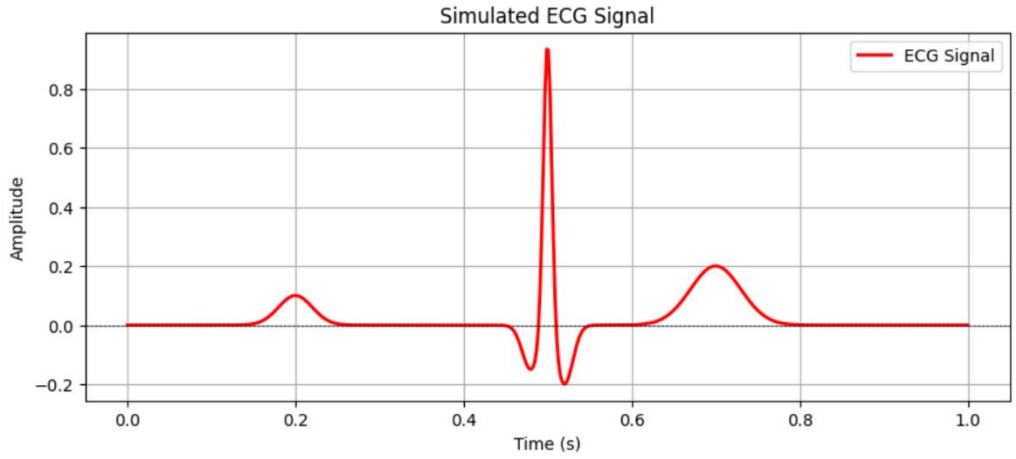
4. Module tạo sóng ECG

Chúng em sử dụng code python để giả lập lại sóng ECG như sau:

```
import numpy as np
import matplotlib.pyplot as plt

# Tạo trục thời gian
fs = 500  # Tần số lấy mẫu (Hz)
t = np.linspace(0, 1, fs)  # 1 giây dữ liệu ECG

# Hàm tạo sóng Gaussian
def gauss_wave(t, center, width, height):
return height * np.exp(-((t - center) ** 2) / (2 * width ** 2))

# Hàm tạo QRS complex
def qrs_complex(t, center, q_amp, r_amp, s_amp, q_width, r_width, s_width):
q_wave = gauss_wave(t, center - 0.02, q_width, -q_amp)
r_wave = gauss_wave(t, center, r_width, r_amp)
s_wave = gauss_wave(t, center + 0.02, s_width, -s_amp)
return q_wave + r_wave + s_wave

# Tạo tín hiệu ECG
ecg_signal = (
gauss_wave(t, 0.2, 0.02, 0.1) +  # P wave
qrs_complex(t, 0.5, 0.15, 1, 0.2, 0.01, 0.005, 0.01) +  # QRS complex
```

```
gauss_wave(t, 0.7, 0.03, 0.2)  # T wave
)

# Vẽ tín hiệu ECG
plt.figure(figsize=(10, 4))
plt.plot(t, ecg_signal, label="ECG Signal", color="r", linewidth=2)
plt.axhline(0, color='k', linestyle='--', linewidth=0.5)  # Đường baseline
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Simulated ECG Signal")
plt.legend()
plt.grid()
plt.show()
```

Kết quả của sóng ECG giả lập:



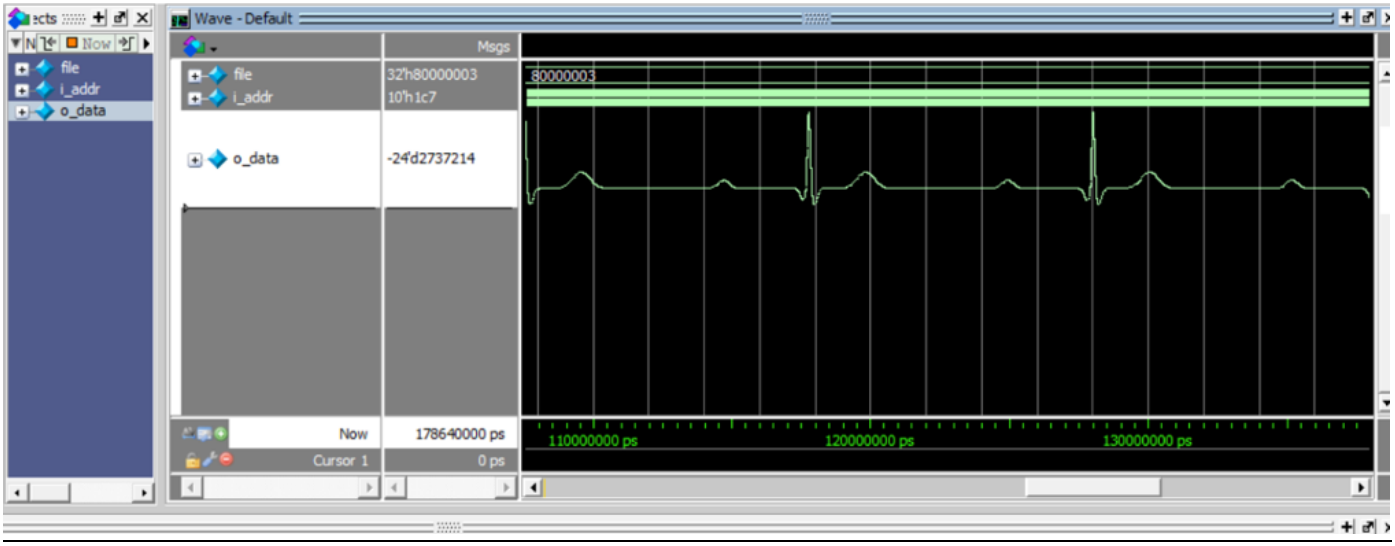
Code Module tạo sóng ECG:

```
module Hamecg (
input [9:0] i_addr,
output reg [23:0] o_data
);
parameter MEM_SIZE = 1024;
reg [23:0] ecg [0:MEM_SIZE-1];

always @(*) begin
o_data = ecg[i_addr]; // Đọc dữ liệu từ LUT
end
```

initial *begin*
$readmemh("E:/Quartus_Prime/Hamecg/ecg.hex", ecg);
end

endmodule

Kết quả mô phỏng bằng testbench:



5. Module tạo nhiễu

Bộ nhiễu trong máy tạo sóng trên FPGA được thiết kế để thêm nhiễu vào tín hiệu gốc, giúp mô phỏng các điều kiện thực tế hoặc kiểm tra hiệu suất của hệ thống trong môi trường có nhiễu. Bộ nhiễu này có hai thành phần điều chỉnh chính: bộ điều chỉnh tần số và bộ điều chỉnh biên độ, giúp kiểm soát nhiễu linh hoạt hơn.

❖ Bộ Điều Chỉnh Tần Số Nhiễu

Bộ điều chỉnh tần số của nhiễu hoạt động tương tự như bộ điều chỉnh tần số của tín hiệu gốc, bằng cách thay đổi bước nhảy địa chỉ khi đọc dữ liệu từ bảng tra nhiễu (LUT chứa 4096 giá trị ngẫu nhiên). Bộ này có bốn mức điều chỉnh:

- Gấp 4 lần tần số gốc: Bước nhảy địa chỉ tăng nhanh, nhiễu thay đổi nhanh hơn, tạo ra nhiễu có tần số cao.
- Gấp 2 lần tần số gốc: Nhiễu thay đổi nhanh hơn mức chuẩn, nhưng không quá cao.
- Bằng tần số gốc: Mức nhiễu tiêu chuẩn, không thay đổi bước nhảy.
- Bằng 1/2 lần tần số gốc: Nhiễu thay đổi chậm hơn, tạo ra nhiễu có tần số thấp.

Cách tiếp cận này giúp kiểm soát mức độ dao động của nhiễu theo nhu cầu thực tế, phù hợp với các ứng dụng mô phỏng khác nhau.

❖ Bộ Điều Chỉnh Biên Độ Nhiễu

Bộ điều chỉnh biên độ của nhiễu hoạt động theo nguyên tắc giống như bộ khuếch đại biên độ của tín hiệu gốc. Biên độ nhiễu có thể được nhân theo hệ số cố định bằng cách sử dụng phép dịch bit để nhân 2 hoặc nhân 4, giúp tăng hoặc giảm ảnh hưởng của nhiễu đối với tín hiệu đầu ra.

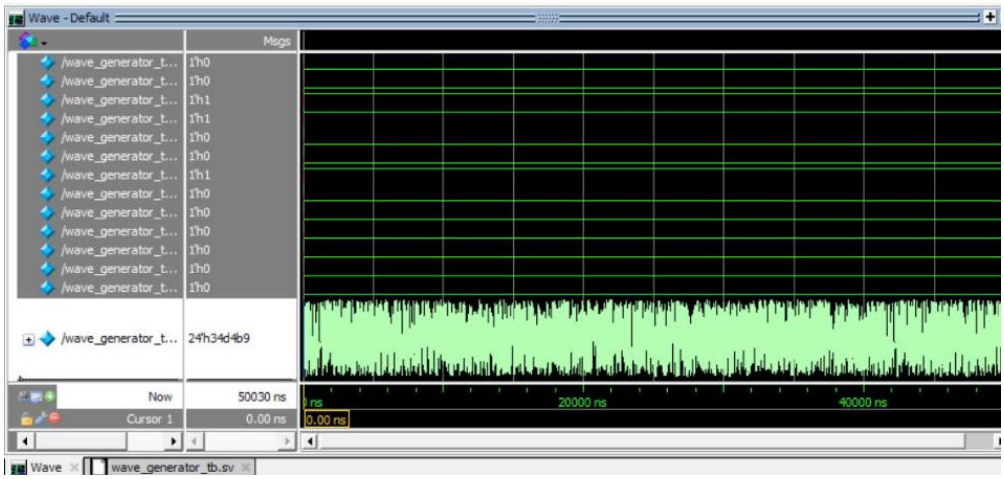
❖ Cách Hoạt Động Tổng Thể

Tín hiệu nhiễu sau khi được điều chỉnh tần số và biên độ sẽ được cộng trực tiếp vào tín hiệu gốc trước khi xuất ra bộ DAC hoặc bộ xử lý tiếp theo. Điều này giúp tạo ra tín hiệu đầu ra có đặc tính nhiễu linh hoạt, phù hợp để kiểm tra hệ thống lọc tín hiệu, nghiên cứu ảnh hưởng của nhiễu, hoặc mô phỏng điều kiện thực tế trong các ứng dụng xử lý tín hiệu số (DSP).

Sau đây là code Module nhiễu:

```
module lfsr_noise (
 input logic [11:0] i_addr,
output logic [23:0] o_data );
parameter MEM_SIZE = 4096;
logic [23:0] LUT_noise [MEM_SIZE-1:0];
 always_comb begin :
next_pc_ff o_data <= LUT_noise[i_addr[11:0]];
 end : next_pc_ff
initial begin
integer i;
 for (i=0; i < MEM_SIZE; i++)
begin LUT_noise[i] = 24'b0;
 end
$readmemh("C:/Users/TUAN/tailieuhocbk/nam4ki1/DSPonFPGA/HK242/lab1(new)/wave_generator/LUT_noise.dump",LUT_noise);
end
endmodule
```

Kết quả mô phỏng Module:



III. Module máy tạo sóng

Máy tạo sóng trên FPGA được thiết kế nhằm tạo ra các dạng sóng như sin, tam giác, xung vuông và sawtooth, với khả năng điều chỉnh tần số, biên độ, duty cycle và thêm nhiễu. Hệ thống được chia thành nhiều module, mỗi module đảm nhận một nhiệm vụ cụ thể nhằm đảm bảo tính linh hoạt và hiệu suất cao.

1. Các module chính

1.1. Bộ điều chỉnh tần số

Bộ điều chỉnh tần số kiểm soát tốc độ truy xuất dữ liệu từ bảng tra LUT để thay đổi tần số tín hiệu đầu ra. Nguyên tắc hoạt động dựa trên bước nhảy địa chỉ:

- Mỗi chu kỳ, bộ đếm địa chỉ sẽ tăng một giá trị cố định để lấy mẫu từ LUT.
- Nếu bước nhảy địa chỉ là +1, toàn bộ 1024 mẫu sẽ được quét trong một chu kỳ, tạo ra tần số cơ bản.
- Nếu bước nhảy là +2, số mẫu sử dụng sẽ giảm một nửa (512 mẫu), làm tần số tăng gấp đôi.
- Nếu bước nhảy là +4, số mẫu sử dụng giảm còn 256 mẫu, làm tần số tăng gấp bốn lần.

Nhờ cách điều chỉnh này, tần số đầu ra có thể thay đổi linh hoạt mà không cần tăng kích thước LUT hoặc thực hiện phép chia phức tạp.

1.2. Bảng tra LUT

Bảng tra LUT lưu trữ sẵn các mẫu tín hiệu số của các dạng sóng khác nhau. LUT này có 1024 mẫu trên một chu kỳ, giúp đảm bảo độ mượt của tín hiệu khi chuyển đổi sang tương tự.

Bộ điều chỉnh tần số sẽ quyết định địa chỉ đọc dữ liệu từ LUT, sau đó gửi giá trị này sang bộ điều chỉnh biên độ trước khi xuất ra DAC.

1.3. Bộ nhiễu

Bộ nhiễu có chức năng thêm nhiễu vào tín hiệu gốc để mô phỏng điều kiện thực tế hoặc kiểm tra hiệu suất của hệ thống trong môi trường bị ảnh hưởng bởi nhiễu.

- Cấu trúc bộ nhiễu

LUT chứa 4096 giá trị ngẫu nhiên, giúp đảm bảo tính đa dạng của nhiễu, tránh sự lặp lại dễ nhận thấy.

Bộ điều chỉnh tần số nhiễu giúp điều chỉnh tốc độ thay đổi nhiễu với 4 mức:

- Gấp 4 lần tần số gốc → nhiễu thay đổi nhanh, phổ rộng.
- Gấp 2 lần tần số gốc → nhiễu thay đổi nhanh, nhưng ít hơn mức gấp 4.
- Bằng tần số gốc → nhiễu đồng bộ với tín hiệu.
- 1/2 hoặc 1/4 lần tần số gốc → nhiễu thay đổi chậm, tạo hiệu ứng trôi nhẹ.

Bộ điều chỉnh biên độ nhiễu giúp kiểm soát mức độ ảnh hưởng của nhiễu bằng cách khuếch đại hoặc giảm biên độ, tương tự như bộ khuếch đại tín hiệu gốc.

- Nguyên lý hoạt động

Sau khi điều chỉnh tần số và biên độ, tín hiệu nhiễu sẽ được cộng trực tiếp vào tín hiệu gốc trước khi gửi đến DAC. Điều này giúp tạo ra tín hiệu có mức độ nhiễu mong muốn mà không làm thay đổi quá trình tạo sóng chính.

1.4. Bộ điều chỉnh biên độ

Bộ điều chỉnh biên độ có nhiệm vụ thay đổi biên độ tín hiệu đầu ra mà không làm thay đổi dạng sóng.

a) Nguyên lý hoạt động

- Để thay đổi biên độ một cách đơn giản và hiệu quả, hệ thống sử dụng dịch bit thay vì phép nhân thông thường.
- Nhân biên độ gấp 2 lần bằng cách dịch trái 1 bit.
- Nhân biên độ gấp 4 lần bằng cách dịch trái 2 bit.

Việc sử dụng dịch bit giúp tiết kiệm tài nguyên phần cứng so với phép nhân thông thường.

1.5. Bộ điều khiển giao tiếp DAC WM8731

Bộ điều khiển này đảm nhận nhiệm vụ truyền tín hiệu số từ FPGA đến DAC Wolfson WM8731, đảm bảo dữ liệu đầu ra đúng định dạng để chuyển đổi sang tín hiệu tương tự.

- Chuẩn hóa dữ liệu: Điều chỉnh độ rộng bit phù hợp với độ phân giải 24-bit của WM8731.
- Truyền dữ liệu qua giao thức I2S hoặc SPI để đảm bảo đồng bộ với DAC.
- Điều khiển xung clock để đảm bảo tốc độ truyền dữ liệu ổn định.

Sau khi dữ liệu được gửi đến DAC, nó sẽ được chuyển đổi thành tín hiệu tương tự và có thể quan sát trên oscilloscope.

2. Quá trình hoạt động tổng thể

- Bộ điều chỉnh tần số xác định tốc độ đọc mẫu từ LUT.
- LUT xuất ra giá trị sóng tại địa chỉ tương ứng.
- Bộ nhiễu tạo ra tín hiệu nhiễu, điều chỉnh tần số và biên độ trước khi cộng vào tín hiệu gốc.
- Bộ điều chỉnh biên độ thay đổi mức độ biên độ theo yêu cầu.
- Bộ điều khiển giao tiếp DAC đảm bảo dữ liệu đầu ra được gửi đúng định dạng đến DAC WM8731.
- DAC chuyển đổi tín hiệu số thành tín hiệu tương tự, có thể quan sát trên oscilloscope hoặc sử dụng làm đầu vào cho các hệ thống khác.

Code Module máy phát sóng:

```
module wave_generator #(
  parameter PHASE_WORD_WIDTH = 32
)(
  input  logic       i_clk, i_rst,                  // Clock và Reset
  input  logic i_sw1, i_sw2, i_sw3, i_sw4, i_sw5, i_sw6, i_sw7, i_sw8,i_sw9,
  input  logic i_button0, i_button1, i_button2, i_button3,
  output logic [23:0] o_wave_out
);
  logic [PHASE_WORD_WIDTH-1:0] current_phase; // Pha hiện tại (32-bit)
  logic [9:0] wave_addr; // Địa chỉ LUT (10-bit)
  logic [23:0] sine_wave, square_wave, triangle_wave, sawtooth_wave,
noise_wave;
  logic [23:0] noise_plus_wave;
```

```
logic [23:0] noiseless_wave;
//pin
logic noise_enable;
logic [3:0]  duty_cycle;
logic [1:0] frequency;
logic [1:0] amplitude_sel;
logic [2:0]  wave_sel;
logic [31:0] phase_step_temp;
logic [23:0] wave_out;
logic button_noise_freq, button_noise_ampl;
logic [1:0] noise_freq_sel, noise_ampl_sel;
logic [31:0] current_phase_noise;
logic [31:0] phase_step_noise_temp;
logic [11:0] noise_addr;
logic [23:0] noise_wave_ampl;
//button_counter
button_counter counter_frequency_sel(
  .i_clk(i_clk),
        .i_rst(i_rst),
  .i_button(i_button0),
  .o_counter_value(frequency)
);
```

```
button_counter counter_amplitude_sel(
  .i_clk(i_clk),
        .i_rst(i_rst),
  .i_button(i_button1),
  .o_counter_value(amplitude_sel)
);

button_counter counter_noisefreq_sel(
  .i_clk(i_clk),
        .i_rst(i_rst),
  .i_button(button_noise_freq),
  .o_counter_value(noise_freq_sel)
);

button_counter counter_noiseampl_sel(
  .i_clk(i_clk),
        .i_rst(i_rst),
  .i_button(button_noise_ampl),
  .o_counter_value(noise_ampl_sel)
);
frequency_wave_sel change_frequency(
```

```
  .i_sel(frequency),
  .o_phase_step(phase_step_temp));

 //switch
 assign noise_enable = i_sw8;
 assign duty_cycle = {i_sw7,i_sw6,i_sw5,i_sw4};
 assign wave_sel = {i_sw3,i_sw2,i_sw1};
 assign button_noise_freq = (i_sw9) ?      1'b0 : i_button3 ;
 assign button_noise_ampl = (i_sw9) ? i_button3 : 1'b0      ;
phase_accumulator #(
  .PHASE_WORD_WIDTH(PHASE_WORD_WIDTH)
 ) phase_acc (
  .i_clk(i_clk),
  .i_rst(i_rst),
  .i_PhaseStep(phase_step_temp),
  .o_CurrentPhase(current_phase)
 );

 // Trích xuất 10-bit MSB từ pha 32-bit làm địa chỉ LUT
 assign wave_addr = current_phase[31:22];
 SineLUT sine_lut (
  .i_addr(wave_addr),
```

```
    .o_data(sine_wave)
  );

  Square_waveLUT square_lut (
    .i_addr(wave_addr),
    .i_sel(duty_cycle),
    .o_data(square_wave)
  );

  Triangle_waveLUT triangle_lut (
    .i_addr(wave_addr),
    .i_sel(duty_cycle),
    .o_data(triangle_wave)
  );

  Sawtooth_waveLUT sawtooth_lut (
    .i_addr(wave_addr),
    .i_sel(duty_cycle),
    .o_data(sawtooth_wave)
  );
// noise
  always_comb begin
```

```
   case(noise_freq_sel)

         2'b00: phase_step_noise_temp = 32'b10000000000000000000; //4096
sample

              2'b01: phase_step_noise_temp = 32'b10000000000000000000;
//2048 sample

              2'b10: phase_step_noise_temp = 32'b10000000000000000000000;
//512 sample

              2'b11: phase_step_noise_temp =
32'b10000000000000000000000;//256  sample

              default: phase_step_noise_temp =
32'b10000000000000000000;//4096 sample

        endcase

  end

  phase_accumulator #(

    .PHASE_WORD_WIDTH(PHASE_WORD_WIDTH)

  ) phase_noise (

    .i_clk(i_clk),

    .i_rst(i_rst),

    .i_PhaseStep(phase_step_noise_temp),

    .o_CurrentPhase(current_phase_noise)

  );

  assign noise_addr = current_phase_noise[31:20];


  lfsr_noise noise_signal (
```

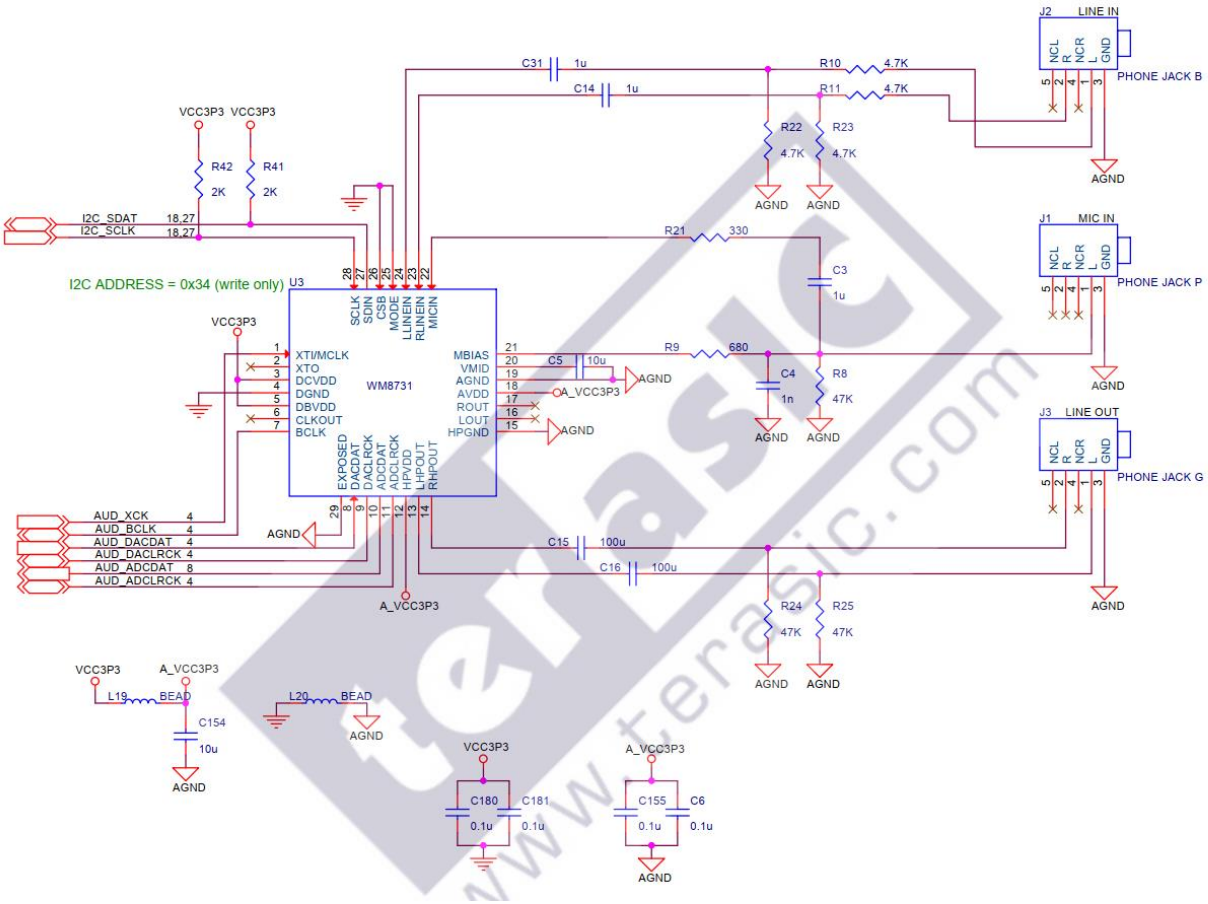
```
  .i_addr(noise_addr),
  .o_data(noise_wave)
);

amplitude_wave_sel ampli_noise(
  .i_wave(noise_wave),
  .i_sel(noise_ampl_sel),
  .o_wave_ampl(noise_wave_ampl));
//
/*
test2_wm8731(
  .clock_50(),
       .key(),
       .sw(),
  .AUD_ADCDAT(),
       .AUD_BCLK(),
       .AUD_XCK(),
       .AUD_ADCLRCK(),
       .AUD_DACLRCK(),
       .AUD_DACDAT(),
       .ledr(),
       .FPGA_I2C_SCLK(),
```

```
.FPGA_I2C_SDAT(),
);
*/
// combination
always_comb begin
  case(wave_sel)
        3'b000:  noiseless_wave = sine_wave;   //sine wave
            3'b001:  noiseless_wave = square_wave; //square wave
            3'b010:  noiseless_wave = triangle_wave;//triangle wave
            3'b011:  noiseless_wave = sawtooth_wave;//sawtooth wave
            3'b100:  noiseless_wave = noise_wave_ampl;
            default: noiseless_wave = 24'b0;
      endcase
end
assign noise_plus_wave = noiseless_wave + noise_wave_ampl;
//output combination
assign wave_out = (noise_enable) ? noise_plus_wave : noiseless_wave;
amplitude_wave_sel ampli_wave(
  .i_wave(wave_out),
  .i_sel(amplitude_sel),
  .o_wave_ampl(o_wave_out));
endmodule
```

IV. Module giao tiếp với WM8731

Để giao tiếp với WM8731, nhóm chúng em sử dụng giao thức I2C để thiết lập các thanh ghi của WM8731. Đồng thời sử dụng mode DSP/PCM để gửi dữ liệu đến đầu vào của bộ DAC của WM8731.

Schematic của WM8731:



1. Module I2C

Theo schematic của DE10 standard với WM8731, CSB được nối đất, do vậy có địa chỉ I2C theo datasheet là: 0b0011010.

Để hiện được sóng lên oscilloscope, chúng em thiết lập các thanh ghi sau:

Thanh ghi	Giá trị	Mô tả chức năng
2 (0000010)	101111001	Mức tín hiệu đầu ra bằng mức tín hiệu đầu vào, không có khuếch đại hay giảm đi.
4 (0000100)	000010010	Không sử dụng MIC làm ngõ vào. Chọn DAC làm nguồn phát tín hiệu (DACSEL = 1).

5 (0000101)	000000000	DAC không bị tắt tiếng, có thể phát âm thanh bình thường.
6 (0000110)	000000111	Codec đang hoạt động (PowerOff = 0). Tắt ADC.
7 (0000111)	000010011	Codec hoạt động ở chế độ Slave (MS = 00), nghĩa là FPGA/MCU phải cấp clock cho nó. Truyền MSB đi ở cạnh lên của BCLK thứ 2 sau cạnh lên của DACLRC. Chọn chế độ ngõ vào DAC là 16 bit.
8 (0001000)	000000001	Chọn chế độ USB. Chọn tốc độ quá lấy mẫu là 250 f_s. Chọn tần số lấy mẫu là 48kHz. Chọn tần số hoạt động của WM8731 là MCLK.
9 (0001001)	000000001	Codec đang hoạt động (ACTIVE = 1).
15 (0001111)	000000000	Reset WM8731

Source code của module I2C:

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


entity i2c is
port(
i2c_busy: out std_logic;
i2c_scl: out std_logic;
i2c_send_flag: in std_logic;
i2c_sda: inout std_logic;
i2c_addr: in std_logic_vector(7 downto 0);
i2c_done: out std_logic;
i2c_data: in std_logic_vector(15 downto 0);
i2c_clock_50: in std_logic
);
end i2c;


architecture main of i2c is

signal i2c_clk_en: std_logic:='0';
signal clk_prs: integer range 0 to 300:=0;
```

```
signal clk_en: std_logic:='0';
signal ack_en: std_logic:='0';
signal clk_i2c: std_logic:='0';
signal get_ack: std_logic:='0';
signal data_index: integer range 0 to 15:=0;
type fsm is (st0,st1,st2,st3,st4,st5,st6,st7,st8);
signal i2c_fsm:fsm:=st0;
begin


------generate two clocks for i2c and data transitions
process(i2c_clock_50)
begin

if rising_edge(i2c_clock_50) then

if(clk_prs<250)then
clk_prs<=clk_prs+1;
else
clk_prs<=0;
end if;

if(clk_prs<125)then ---50 % duty cylce clock for i2c
clk_i2c<='1';
else
clk_i2c<='0';
end if;

---- clock for ack  on SCL=HIGH
if(clk_prs=62)then
ack_en<='1';
else
ack_en<='0';
end if;


---- clock for data on SCL=LOW
if(clk_prs=187)then
clk_en<='1';
else
```

```
clk_en<='0';
end if;

end if;

if rising_edge(i2c_clock_50) then


if(i2c_clk_en='1')then
i2c_scl<=clk_i2c;
else
i2c_scl<='1';
end if;


----ack on SCL=HIGH
if(ack_en='1')then
case i2c_fsm is
when st3=> ---- get ack

if(i2c_sda='0')then
i2c_fsm<=st4;---ack
data_index<=15;
else
i2c_clk_en<='0';
i2c_fsm<=st0;---nack
end if;

when st5=> --- get ack

if(i2c_sda='0')then
i2c_fsm<=st6;---ack
data_index<=7;

else
i2c_fsm<=st0;---nack
i2c_clk_en<='0';
end if;

when st7 => ----get ack
```

```
if(i2c_sda='0')then
i2c_fsm<=st8;---ack
else
i2c_fsm<=st0;---nack
i2c_clk_en<='0';
end if;

when others=>NULL;
end case;
end if;



-----data tranfer on SCL=LOW
if(clk_en='1')then
case i2c_fsm is
when st0=> ----------stand by
i2c_sda<='1';
i2c_busy<='0';
i2c_done<='0';
if(i2c_send_flag='1')then
i2c_fsm<=st1;
i2c_busy<='1';
end if;

when st1=> -------start condition
i2c_sda<='0';
i2c_fsm<=st2;
data_index<=7;
when st2=> -------send addr
i2c_clk_en<='1';---start clocking i2c_scl

if(data_index>0) then
data_index<=data_index-1;
i2c_sda<=i2c_addr(data_index);
else
i2c_sda<=i2c_addr(data_index);
get_ack<='1';
end if;
```

```
if(get_ack='1')then
get_ack<='0';
i2c_fsm<=st3;
i2c_sda<='Z';
end if;

when st4=> ---- send 1st 8 bit

if(data_index>8) then
data_index<=data_index-1;
i2c_sda<=i2c_data(data_index);
else
i2c_sda<=i2c_data(data_index);
get_ack<='1';
end if;

if(get_ack='1')then
get_ack<='0';
i2c_fsm<=st5;
i2c_sda<='Z';
end if;

when st6 => ---send 2nd 8 bit

if(data_index>0) then
data_index<=data_index-1;
i2c_sda<=i2c_data(data_index);
else
i2c_sda<=i2c_data(data_index);
get_ack<='1';
end if;

if(get_ack='1')then
get_ack<='0';
i2c_fsm<=st7;
i2c_sda<='Z';
end if;

when st8 => --stop condition
```

```
i2c_clk_en<='0';
i2c_sda<='0';
i2c_fsm<=st0;
i2c_done<='1';
when others=>NULL;
end case;
end if;

end if;
end process;
end main;
```

Giải thích:

- Input của module:

* i2c_send_flag: cờ cho phép gửi dữ liệu từ FPGA

* i2c_addr: địa chỉ I2C của WM8731: 0b0011010

* i2c_data: dữ liệu gửi bằng I2C, gồm 16 bit trong đó có 7 bit cho biết thanh ghi nào và 9 bit dữ liệu truyền vào thanh ghi đó

* i2c_clock_50: clock 50Mhz từ FPGA

- Output của module:

* i2c_scl: clock 200Khz tạo bởi module i2c

* i2c_busy: cờ bận được tạo bởi module i2c

* i2c_done: cờ gửi hoàn tất tạo bởi module i2c

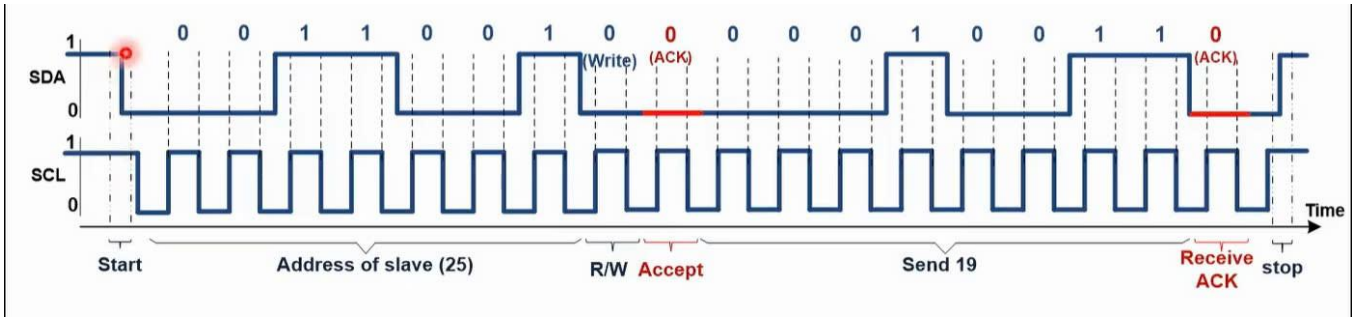
- Inout của module:

* i2c_sda: dữ liệu của I2C được cấu hình là inout

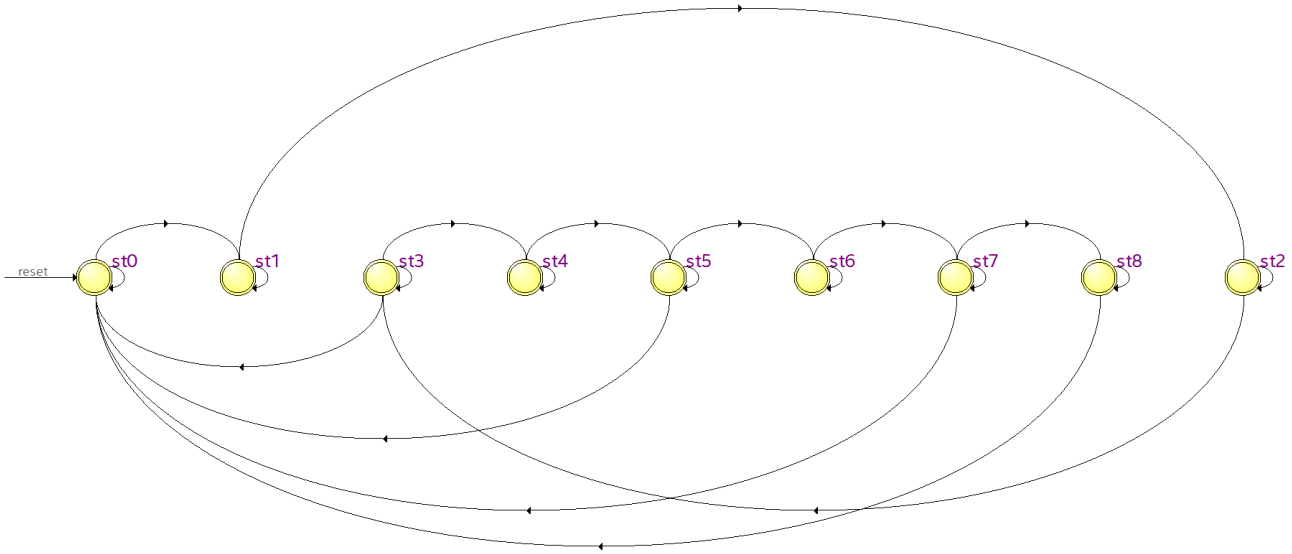
- Đầu tiên từ clock 50Mhz của FPGA, tạo bộ đếm tới 250 để tạo tần số 200Khz làm xung clock cho I2C. Xung SCLK này là xung vuông có duty là 50% ứng với hai giá trị của bộ đếm.

- Dùng bộ đếm để xác định mức 1 và mức 0 của SCLK nhằm bật các tín hiệu cho phép ACK và cho phép gửi dữ liệu

- Lập trình dựa trên phương thức truyền dữ liệu của I2C theo giản đồ trạng thái:



Giản đồ trạng thái:



Các điều kiện chuyển trạng thái, lấy từ state machine viewer trong quartus.

Trạng thái hiện tại		Trạng thái tiếp theo	Điều kiện chuyển trạng thái
1	st0	st1	(i2c_send_flag).(clk_en)
2	st0	st0	(!i2c_send_flag) + (i2c_send_flag).(!clk_en)
3	st1	st1	(!clk_en)
4	st1	st2	(clk_en)
5	st2	st2	(!get_ack) + (get_ack).(!clk_en)
6	st2	st3	(get_ack).(clk_en)
7	st3	st0	(i2c_sda~direct).(ack_en)

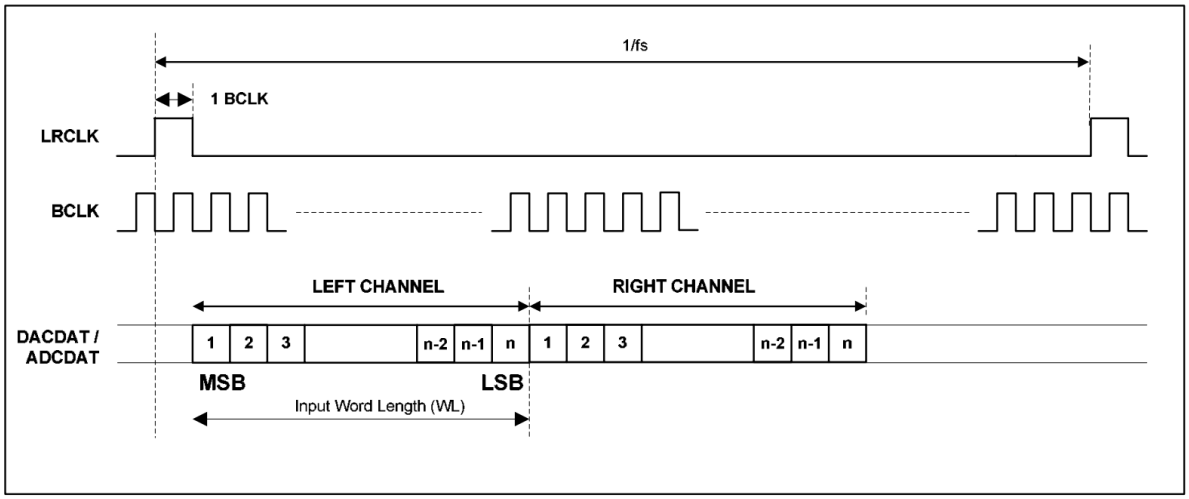
8	st3	st4	(!i2c_sda~direct).(ack_en)
9	st3	st3	(!ack_en)
10	st4	st5	(get_ack).(clk_en)
11	st4	st4	(!get_ack) + (get_ack).(!clk_en)
12	st5	st0	(i2c_sda~direct).(ack_en)
13	st5	st6	(!i2c_sda~direct).(ack_en)
14	st5	st5	(!ack_en)
15	st6	st6	(!get_ack) + (get_ack).(!clk_en)
16	st6	st7	(get_ack).(clk_en)
17	st7	st0	(i2c_sda~direct).(ack_en)
18	st7	st8	(!i2c_sda~direct).(ack_en)
19	st7	st7	(!ack_en)
20	st8	st0	(clk_en)
21	st8	st8	(!clk_en)

- Từ việc mô phỏng lại quá trình truyền dữ liệu của I2C, chúng em có thể dùng module này như một module con trong module lớn, khi cần truyền dữ liệu vào thanh ghi, truyền 7 bit chỉ thanh ghi đó là thanh ghi nào trước, sau đó lại truyền 9 bit dữ liệu vào trong thanh ghi đó.

2. Module aud_gen

Trong bộ audio codec WM8731 có 4 chế độ gửi dữ liệu vào DAC, đó là chế độ căn chỉnh trái, căn chỉnh phải, I2S, DSP/PCM.

Trong lab1 này, chúng em chọn chế độ DSP/PCM để gửi dữ liệu tới DAC. Phương thức truyền của chế độ này dựa theo hình sau:



Với mỗi chu kỳ của LRCLK, một mẫu dữ liệu 2n bit được gửi đến DAC, trong đó bao gồm n bit kênh trái và n bit kênh phải.

Clock BCLK nhằm mục đích truyền từng bit trong mẫu đi trong một chu kỳ của LRCLK.

Do vậy nên clock của BCLK phải nhanh hơn nhiều so với clock LRCLK.

Tần số lấy mẫu của bộ DAC được chọn dựa theo hai chế độ là Normal mode và USB mode. Trong đó chế độ USB mode cố định tần số MCLK, là nguồn cung cấp clock chính của bộ codec được cung cấp từ fpga. Do vậy nên chúng em chọn mode USB để tạo ra tần số lấy mẫu cố định là 48Khz và MCLK là 12Mhz. Chế độ USB tạo tần số lấy mẫu được cho dưới hình sau đây:

SAMPLING RATE		MCLK FREQUENCY	SAMPLE RATE REGISTER SETTINGS					DIGITAL FILTER TYPE
ADC	DAC							
kHz	kHz	MHz	BOSR	SR3	SR2	SR1	SR0	
48	48	12.000	0	0	0	0	0	0
44.1 (Note 2)	44.1 (Note 2)	12.000	1	1	0	0	0	1
48	8	12.000	0	0	0	0	1	0
44.1 (Note 2)	8 (Note 1)	12.000	1	1	0	0	1	1
8	48	12.000	0	0	0	1	0	0
8 (Note 1)	44.1 (Note 2)	12.000	1	1	0	1	0	1
8	8	12.000	0	0	0	1	1	0
8 (Note 1)	8 (Note 1)	12.000	1	1	0	1	1	1
32	32	12.000	0	0	1	1	0	0
96	96	12.000	0	0	1	1	1	3
88.2 (Note 3)	88.2 (Note 3)	12.000	1	1	1	1	1	2

Lý do để nhóm chúng em chọn mode DSP/PCM để truyền dữ liệu:

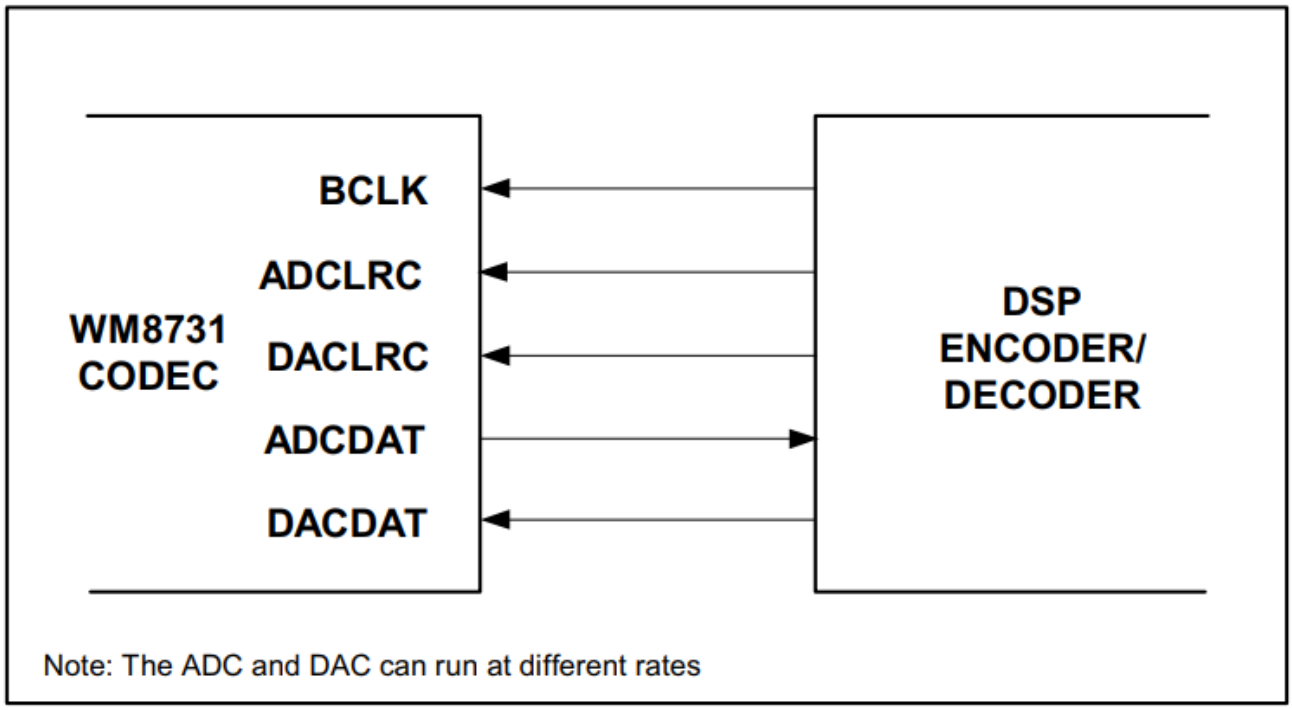
- Truyền dữ liệu 2 kênh trái phải được truyền liên tiếp nên trong một chu kỳ của LRCLK không nhất thiết phải có duty 50% như 3 chế độ còn lại. Do vậy nên có thể tạo được clock LRCLK một cách dễ dàng hơn từ MCLK.

- BCLK không cần phải liên tục trong mỗi chu kỳ của LRCLK, do chỉ cần truyền hết 2n bit nên còn một khoảng còn lại clock không dùng.

DAC trong WM8731 xử lý dữ liệu dạng 24 bit có dấu, nhưng có rất nhiều trường hợp số bit khác nhau, khi đó có thể xử lý để đưa về 24 bit. Các tùy chọn này được cho trong bảng thanh ghi sau:

REGISTER ADDRESS	BIT	LABEL	DEFAULT	DESCRIPTION
0000111 Digital Audio Interface Format	1:0	FORMAT[1:0]	10	Audio Data Format Select 11 = DSP Mode, frame sync + 2 data packed words 10 = I^2S Format, MSB-First left-1 justified 01 = MSB-First, left justified 00 = MSB-First, right justified
	3:2	IWL[1:0]	10	Input Audio Data Bit Length Select 11 = 32 bits 10 = 24 bits 01 = 20 bits 00 = 16 bits
	4	LRP	0	DACLRC phase control (in left, right or I^2S modes) 1 = Right Channel DAC data when DACLRC high 0 = Right Channel DAC data when DACLRC low (opposite phasing in I^2S mode) or DSP mode A/B select (in DSP mode only) 1 = MSB is available on 2nd BCLK rising edge after DACLRC rising edge 0 = MSB is available on 1st BCLK rising edge after DACLRC rising edge
	5	LRSWAP	0	DAC Left Right Clock Swap 1 = Right Channel DAC Data Left 0 = Right Channel DAC Data Right
	6	MS	0	Master Slave Mode Control 1 = Enable Master Mode 0 = Enable Slave Mode
	7	BCLKINV	0	Bit Clock Invert 1 = Invert BCLK 0 = Don't invert BCLK

Trong module này, chúng em dựa trên kết nối giữa WM8731 và fpga như sau ở chế độ slave mode:



Source code của module aud_gen:

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


entity  aud_gen is
port (
aud_clock_12: in std_logic;
aud_bk: out std_logic;
aud_dalr: out std_logic;
aud_dadat: out std_logic;
aud_data_in: in std_logic_vector(31 downto 0)

);
end aud_gen;

architecture main of aud_gen is
```

```
signal sample_flag: std_logic:='0';
signal data_index: integer range 0 to 31:=0;
signal da_data :std_logic_vector(15 downto 0):=(others=>'0');
signal da_data_out: std_logic_vector(31 downto 0):=(others=>'0');
signal aud_prscl: integer range 0 to 300:=0;
signal clk_en: std_logic:='0';
begin

aud_bk<=aud_clock_12;

process(aud_clock_12)
begin

if falling_edge(aud_clock_12) then

aud_dalr<=clk_en;

if(aud_prscl<250)then------48k sample rate
aud_prscl<=aud_prscl+1;
clk_en<='0';
else
aud_prscl<=0;
da_data_out<=aud_data_in;--get sample
clk_en<='1';
end if;



if(clk_en='1')then-------send new sample
sample_flag<='1';
data_index<=31;
end if;

if(sample_flag='1')then

if(data_index>0)then
aud_dadat<=da_data_out(data_index);
data_index<=data_index-1;
else
aud_dadat<=da_data_out(data_index);
```

```
sample_flag<='0';
end if;

end if;
end if;

end process;
end main;
```

Giải thích:

- Input của module:

* aud_clock_12: clock 12Mhz tạo bởi fpga

* aud_data_in: dữ liệu 32 bit gởi từ fpga

- Output của module:

* aud_bk: clock BCLK

* aud_dalr: clock LRCLK

* aud_dadat: đại diện cho mỗi bit của dữ liệu để gửi bởi mỗi clock của BCLK

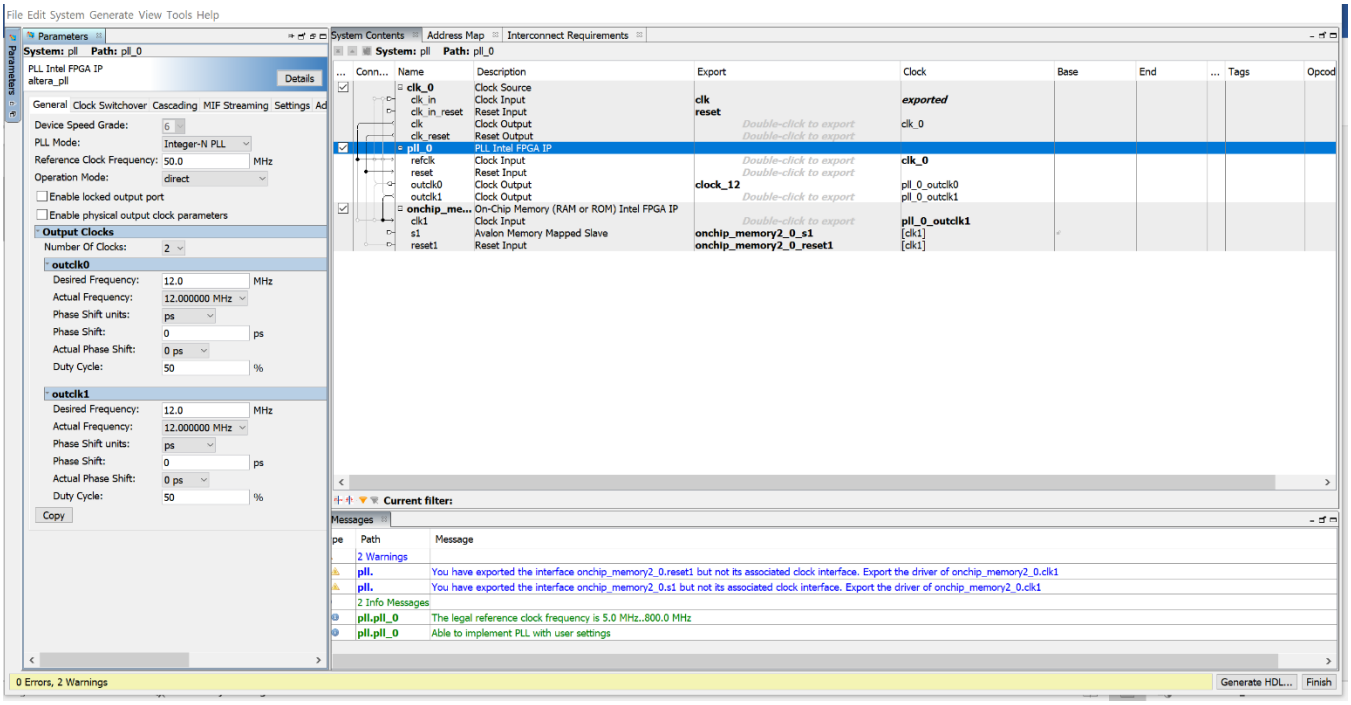
- Từ clock 12Mhz tạo bởi bộ PLL, gán cho MCLK và BCLK, chia 250 lần để tạo ra tần số lấy mẫu 48Khz.

- Và dựa trên mỗi clock của BCLK để gửi từng bit của dữ liệu.

3. Module audio_codec

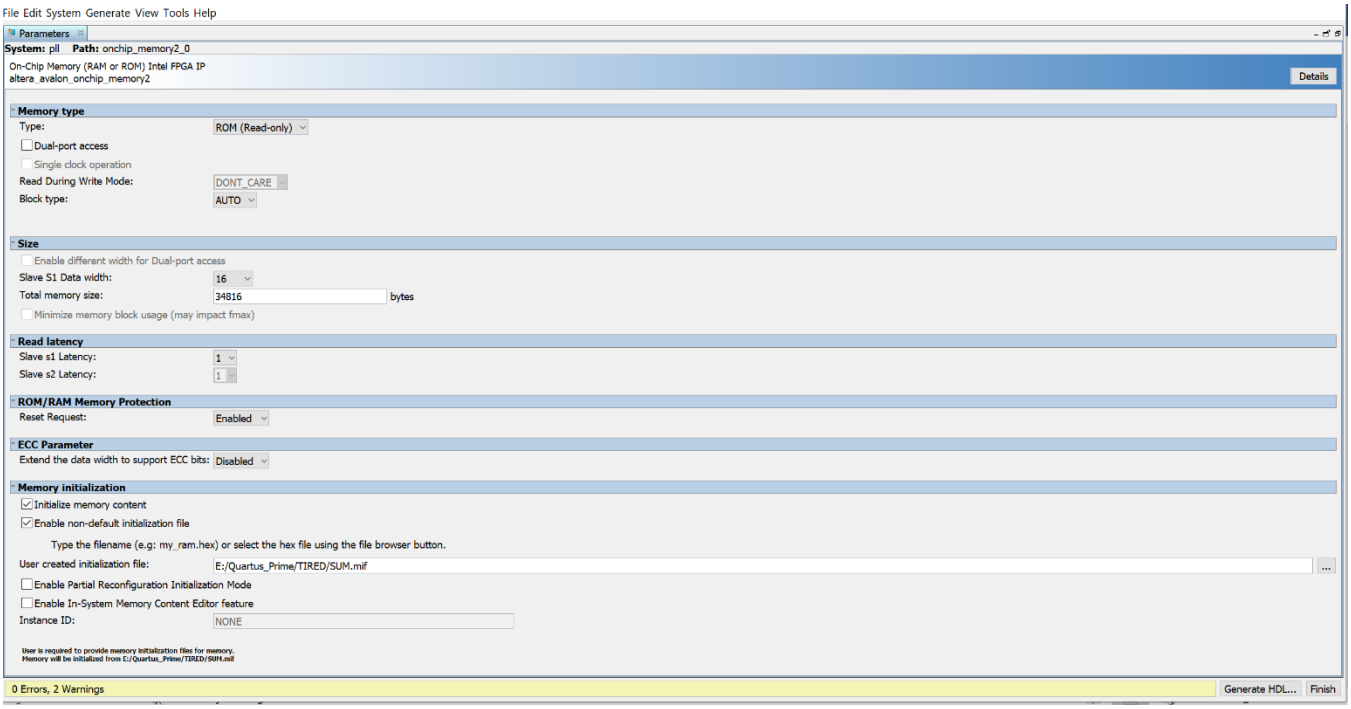
- Đây là module tổng hợp 2 module trên để hiện sóng lên oscilloscope.

- Để tạo ra được clock 12Mhz, em sử dụng IP PLL có sẵn trong platform designer của quartus như hình:



- Trong này, em tạo ra 12Mhz từ clock 50Mhz của fpga, phân ra làm 2 nguồn clock. Và kết nối nó với clk_0.

- Nhóm em tạo 1 IP onchip_memory2_0 từ platform designer này để lưu file MIF (memory initialization file) trong ROM, file này chứa giá trị các mẫu của sóng giống như file LUT ở phần thực hiện ở máy tính.



Với file SUM.mif là file chứa 5120 mẫu của 5 dạng sóng, mỗi sóng bao gồm 1024 mẫu, mỗi mẫu là dữ liệu 16 bit có dấu.

Sau đó generate để tạo ra file pll.qip và add vào file của project

Source code của module audio_codec:

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


entity  audio_codec is
port (


----------WM8731 pins-----
AUD_BCLK: out std_logic;
AUD_XCK: out std_logic;
AUD_ADCLRCK:  out std_logic;
AUD_ADCDAT: in std_logic;
AUD_DACLRCK: out std_logic;
AUD_DACDAT: out std_logic;

---------FPGA pins-----

clock_50: in std_logic;
key: in std_logic_vector(3 downto 0);
ledr: out std_logic_vector(9 downto 0);
sw: in std_logic_vector(9 downto 0);
FPGA_I2C_SCLK: out std_logic;
FPGA_I2C_SDAT: inout std_logic

);

end audio_codec;


architecture main of audio_codec is

signal aud_mono: std_logic_vector(31 downto 0):=(others=>'0');
signal read_addr: integer range 0 to 5220:=0;
signal ROM_ADDR: std_logic_vector(12 downto 0);
signal ROM_OUT: std_logic_vector(15 downto 0);
signal clock_12pll: std_logic;
signal WM_i2c_busy: std_logic;
signal WM_i2c_done: std_logic;
signal WM_i2c_send_flag: std_logic;
signal WM_i2c_data: std_logic_vector(15 downto 0);
signal DA_CLR: std_logic:='0';

component pll is
port (
clk_clk                         : in  std_logic                     := 'X';             -- clk
clock_12_clk                    : out std_logic;                                        -- clk
reset_reset_n                   : in  std_logic                     := 'X';             --
reset_n
```

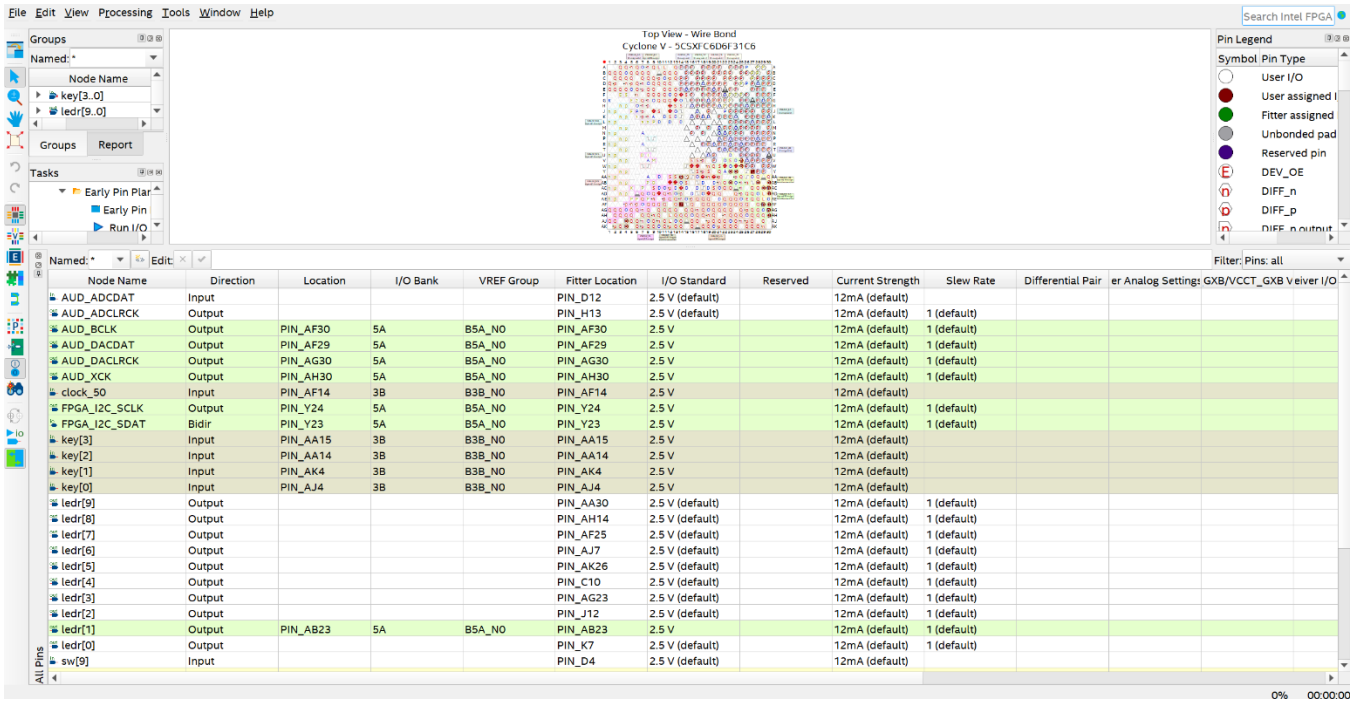
```
onchip_memory2_0_s1_address     : in  std_logic_vector(12 downto 0) := (others => 'X'); --
address
onchip_memory2_0_s1_debugaccess : in  std_logic                     := 'X';             --
debugaccess
onchip_memory2_0_s1_clken       : in  std_logic                     := 'X';             -- clken
onchip_memory2_0_s1_chipselect  : in  std_logic                     := 'X';             --
chipselect
onchip_memory2_0_s1_write       : in  std_logic                     := 'X';             -- write
onchip_memory2_0_s1_readdata    : out std_logic_vector(15 downto 0);                    --
readdata
onchip_memory2_0_s1_writedata   : in  std_logic_vector(15 downto 0) := (others => 'X'); --
writedata
onchip_memory2_0_s1_byteenable  : in  std_logic_vector(1 downto 0)  := (others => 'X'); --
byteenable
onchip_memory2_0_reset1_reset   : in  std_logic                     := 'X'              -- reset
);
end component pll;

component aud_gen is
port (
aud_clock_12: in std_logic;
aud_bk: out std_logic;
aud_dalr: out std_logic;
aud_dadat: out std_logic;
aud_data_in: in std_logic_vector(31 downto 0)
);
end component aud_gen;




component i2c is
port(
i2c_busy: out std_logic;
i2c_scl: out std_logic;
i2c_send_flag: in std_logic;
i2c_sda: inout std_logic;
i2c_addr: in std_logic_vector(7 downto 0);
i2c_done: out std_logic;
i2c_data: in std_logic_vector(15 downto 0);
i2c_clock_50: in std_logic
);

end component i2c;


begin


u0 : component pll
port map (
clk_clk       => clock_50,                                      -- clk.clk
reset_reset_n => '1',                                           -- reset.reset_n
clock_12_clk  => clock_12pll,
                 -- clock_12.clk
onchip_memory2_0_s1_address     => ROM_ADDR,
onchip_memory2_0_s1_debugaccess =>'0',                          -- debugaccess
```

```
onchip_memory2_0_s1_clken       =>'1',                                  -- clken
onchip_memory2_0_s1_chipselect  =>'1',                                  -- chipselect
onchip_memory2_0_s1_write      =>'0',                                   -- write
onchip_memory2_0_s1_readdata   =>ROM_OUT,                               -- readdata
onchip_memory2_0_s1_writedata  =>(others=>'0'),
onchip_memory2_0_s1_byteenable  =>"11",
onchip_memory2_0_reset1_reset=>'0'
);



sound: component aud_gen
port map(
aud_clock_12=>clock_12pll,
aud_bk=>AUD_BCLK,
aud_dalr=>DA_CLR,
aud_dadat=>AUD_DACDAT,
aud_data_in=>aud_mono

);

WM8731: component i2c
port map(
i2c_busy=>WM_i2c_busy,
i2c_scl=>FPGA_I2C_SCLK,
i2c_send_flag=>WM_i2c_send_flag,
i2c_sda=>FPGA_I2C_SDAT,
i2c_addr=>"00110100",
i2c_done=>WM_i2c_done,
i2c_data=>WM_i2c_data,
i2c_clock_50=>clock_50
);

AUD_XCK<=clock_12pll;
AUD_DACLRCK<=DA_CLR;

ROM_ADDR<=std_logic_vector(to_unsigned(read_addr,13));

process (clock_12pll)
begin

if rising_edge(clock_12pll) then
if (SW(8) = '1') then -- Reset
aud_mono <= (others => '0');
else
LEDR(1) <= SW(7);
aud_mono(15 downto 0) <= ROM_OUT; -- Mono sound
aud_mono(31 downto 16) <= ROM_OUT;

if (DA_CLR = '1') then
if (SW(1) = '1') then
read_addr <= 0;
if (read_addr < 1023) then
read_addr <= read_addr + 1;
else
read_addr <= 0;
end if;
```

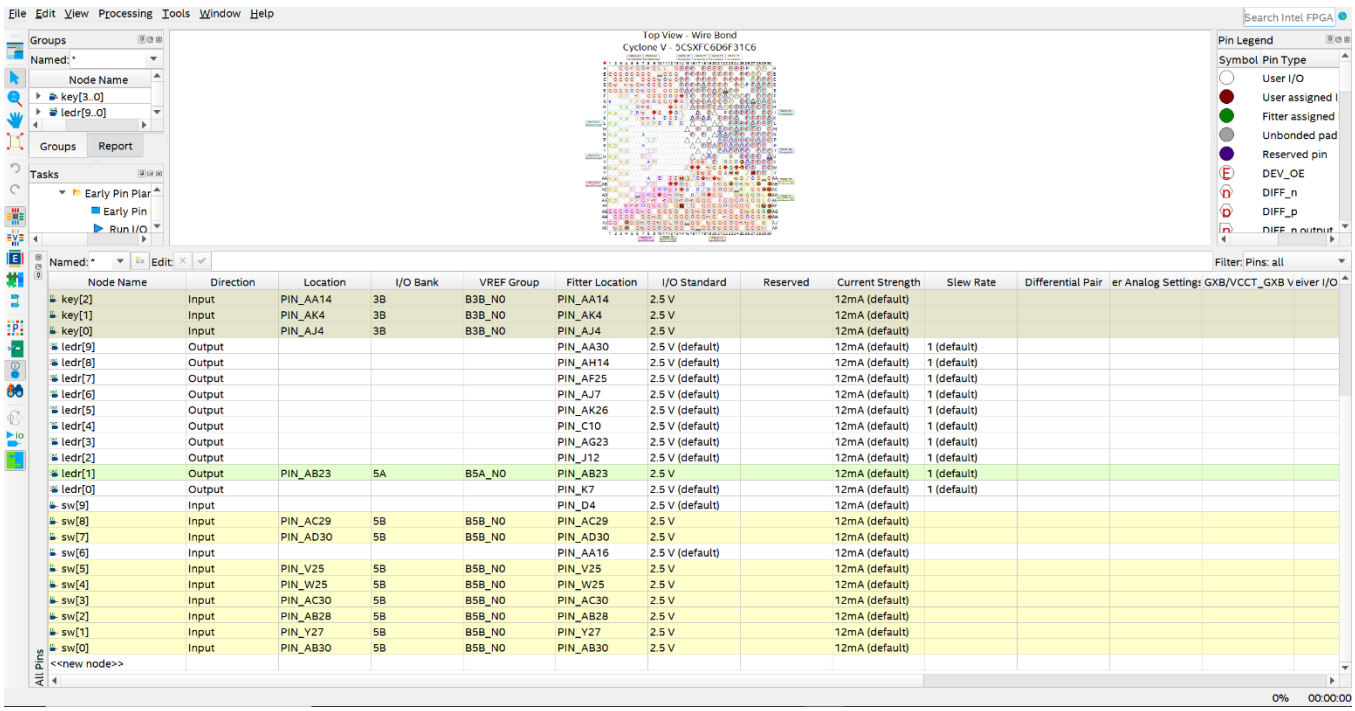
```
elsif (SW(2) = '1') then
read_addr <= 1024;
if (read_addr < 2047) then
read_addr <= read_addr + 1;
else
read_addr <= 1024;
end if;

elsif (SW(3) = '1') then
read_addr <= 2048;
if (read_addr < 3071) then
read_addr <= read_addr + 1;
else
read_addr <= 2048;
end if;

elsif (SW(4) = '1') then
read_addr <= 3072;
if (read_addr < 4095) then
read_addr <= read_addr + 1;
else
read_addr <= 3072;
end if;

elsif (SW(5) = '1') then
read_addr <= 4096;
if (read_addr < 5119) then
read_addr <= read_addr + 1;
else
read_addr <= 4096;
end if;
end if;
end if;
end if;
end if;




end process;

process (clock_50)
begin

if rising_edge (clock_50)then
if(KEY="1111")then
WM_i2c_send_flag<='0';
end if;
end if;
if rising_edge(clock_50) and WM_i2c_busy='0' then


if (KEY(0)='0') then ----Digital Interface: DSP, 16 bit, slave mode
WM_i2c_data(15 downto 9)<="0000111";
WM_i2c_data(8 downto 0)<="000010011";
WM_i2c_send_flag<='1';
```

```
elsif (KEY(0)='0'AND SW(0)='1' ) then---HEADPHONE VOLUME
WM_i2c_data(15 downto 9)<="0000010";
WM_i2c_data(8 downto 0)<="101111001";
WM_i2c_send_flag<='1';

elsif (KEY(1)='0'AND SW(0)='0' ) then---ADC of, DAC on, Linout ON, Power ON
WM_i2c_data(15 downto 9)<="0000110";
WM_i2c_data(8 downto 0)<="000000111";

WM_i2c_send_flag<='1';
elsif (KEY(1)='0'AND SW(0)='1' ) then---USB mode
WM_i2c_data(15 downto 9)<="0001000";
WM_i2c_data(8 downto 0)<="000000001";

WM_i2c_send_flag<='1';
elsif (KEY(2)='0'AND SW(0)='0') then---activ interface
WM_i2c_data(15 downto 9)<="0001001";
WM_i2c_data(8 downto 0)<="111111111";

WM_i2c_send_flag<='1';
elsif (KEY(2)='0'AND SW(0)='1') then---Enable DAC to LINOUT
WM_i2c_data(15 downto 9)<="0000100";
WM_i2c_data(8 downto 0)<="000010010";

WM_i2c_send_flag<='1';
elsif (KEY(3)='0' AND SW(0)='0') then---remove mute DAC
WM_i2c_data(15 downto 9)<="0000101";
WM_i2c_data(8 downto 0)<="000000000";

WM_i2c_send_flag<='1';
elsif (KEY(3)='0' AND SW(0)='1') then---reset
WM_i2c_data(15 downto 9)<="0001111";
WM_i2c_data(8 downto 0)<="000000000";

WM_i2c_send_flag<='1';
end if;

end if;
end process;
end main;
```

Gán chân bằng Pin Planner:



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	er Analog Setting:	GXB/VCCT_GXB	eiver I/O
AUD_ADCDAT	Input				PIN_D12	2.5 V (default)		12mA (default)					
AUD_ADCLRCK	Output				PIN_H13	2.5 V (default)		12mA (default)	1 (default)				
AUD_BCLK	Output	PIN_AF30	5A	B5A_N0	PIN_AF30	2.5 V		12mA (default)	1 (default)				
AUD_DACDAT	Output	PIN_AF29	5A	B5A_N0	PIN_AF29	2.5 V		12mA (default)	1 (default)				
AUD_DACLRCK	Output	PIN_AG30	5A	B5A_N0	PIN_AG30	2.5 V		12mA (default)	1 (default)				
AUD_XCK	Output	PIN_AH30	5A	B5A_N0	PIN_AH30	2.5 V		12mA (default)	1 (default)				
clock_50	Input	PIN_AF14	3B	B3B_N0	PIN_AF14	2.5 V		12mA (default)					
FPGA_I2C_SCLK	Output	PIN_Y24	5A	B5A_N0	PIN_Y24	2.5 V		12mA (default)	1 (default)				
FPGA_I2C_SDAT	Bidir	PIN_Y23	5A	B5A_N0	PIN_Y23	2.5 V		12mA (default)	1 (default)				
key[3]	Input	PIN_AA15	3B	B3B_N0	PIN_AA15	2.5 V		12mA (default)					
key[2]	Input	PIN_AA14	3B	B3B_N0	PIN_AA14	2.5 V		12mA (default)					
key[1]	Input	PIN_AK4	3B	B3B_N0	PIN_AK4	2.5 V		12mA (default)					
key[0]	Input	PIN_AJ4	3B	B3B_N0	PIN_AJ4	2.5 V		12mA (default)					
ledr[9]	Output				PIN_AA30	2.5 V (default)		12mA (default)	1 (default)				
ledr[8]	Output				PIN_AH14	2.5 V (default)		12mA (default)	1 (default)				
ledr[7]	Output				PIN_AF25	2.5 V (default)		12mA (default)	1 (default)				
ledr[6]	Output				PIN_AJ7	2.5 V (default)		12mA (default)	1 (default)				
ledr[5]	Output				PIN_AK26	2.5 V (default)		12mA (default)	1 (default)				
ledr[4]	Output				PIN_C10	2.5 V (default)		12mA (default)	1 (default)				
ledr[3]	Output				PIN_AG23	2.5 V (default)		12mA (default)	1 (default)				
ledr[2]	Output				PIN_J12	2.5 V (default)		12mA (default)	1 (default)				
ledr[1]	Output	PIN_AB23	5A	B5A_N0	PIN_AB23	2.5 V		12mA (default)	1 (default)				
ledr[0]	Output				PIN_K7	2.5 V (default)		12mA (default)	1 (default)				
sw[9]	Input				PIN_D4	2.5 V (default)		12mA (default)					



Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	er Analog Setting:	GXB/VCCT_GXB	eiver I/O
key[2]	Input	PIN_AA14	3B	B3B_N0	PIN_AA14	2.5 V		12mA (default)					
key[1]	Input	PIN_AK4	3B	B3B_N0	PIN_AK4	2.5 V		12mA (default)					
key[0]	Input	PIN_AJ4	3B	B3B_N0	PIN_AJ4	2.5 V		12mA (default)					
ledr[9]	Output				PIN_AA30	2.5 V (default)		12mA (default)	1 (default)				
ledr[8]	Output				PIN_AH14	2.5 V (default)		12mA (default)	1 (default)				
ledr[7]	Output				PIN_AF25	2.5 V (default)		12mA (default)	1 (default)				
ledr[6]	Output				PIN_AJ7	2.5 V (default)		12mA (default)	1 (default)				
ledr[5]	Output				PIN_AK26	2.5 V (default)		12mA (default)	1 (default)				
ledr[4]	Output				PIN_C10	2.5 V (default)		12mA (default)	1 (default)				
ledr[3]	Output				PIN_AG23	2.5 V (default)		12mA (default)	1 (default)				
ledr[2]	Output				PIN_J12	2.5 V (default)		12mA (default)	1 (default)				
ledr[1]	Output	PIN_AB23	5A	B5A_N0	PIN_AB23	2.5 V		12mA (default)	1 (default)				
ledr[0]	Output				PIN_K7	2.5 V (default)		12mA (default)	1 (default)				
sw[9]	Input				PIN_D4	2.5 V (default)		12mA (default)					
sw[8]	Input	PIN_AC29	5B	B5B_N0	PIN_AC29	2.5 V		12mA (default)					
sw[7]	Input	PIN_AD30	5B	B5B_N0	PIN_AD30	2.5 V		12mA (default)					
sw[6]	Input				PIN_AA16	2.5 V (default)		12mA (default)					
sw[5]	Input	PIN_V25	5B	B5B_N0	PIN_V25	2.5 V		12mA (default)					
sw[4]	Input	PIN_W25	5B	B5B_N0	PIN_W25	2.5 V		12mA (default)					
sw[3]	Input	PIN_AC30	5B	B5B_N0	PIN_AC30	2.5 V		12mA (default)					
sw[2]	Input	PIN_AB28	5B	B5B_N0	PIN_AB28	2.5 V		12mA (default)					
sw[1]	Input	PIN_Y27	5B	B5B_N0	PIN_Y27	2.5 V		12mA (default)					
sw[0]	Input	PIN_AB30	5B	B5B_N0	PIN_AB30	2.5 V		12mA (default)					
<<new node>>													

Giải thích:

- Khai báo lại các input và output cho các pin của FPGA và WM8731, khai báo các tín hiệu trung gian để gán vào các module con, lấy clock_12pll từ module pll của qsys, đưa địa chỉ từ module audio_codec vào ROM để đọc ra được dữ liệu trong file MIF với tần số 12MHZ, sau đó đưa các thông số vào module i2c để dùng gửi dữ liệu.

- Chúng em sẽ đưa từng mẫu vào module aud_gen để gửi tới DAC với tốc độ bằng với tốc độ lấy mẫu của bộ DAC luôn và bằng 48Khz, tức là ở dòng if (DA_CLR = '1') then …

- Khi đổ xuống KIT, trước đó, cần cấu hình I2C bằng cách nhấn các nút để có thời gian để fpga gửi các dữ liệu cho các thanh ghi khác nhau, quy trình gửi bằng các nút sau:

* Gạt SW8 lên để reset

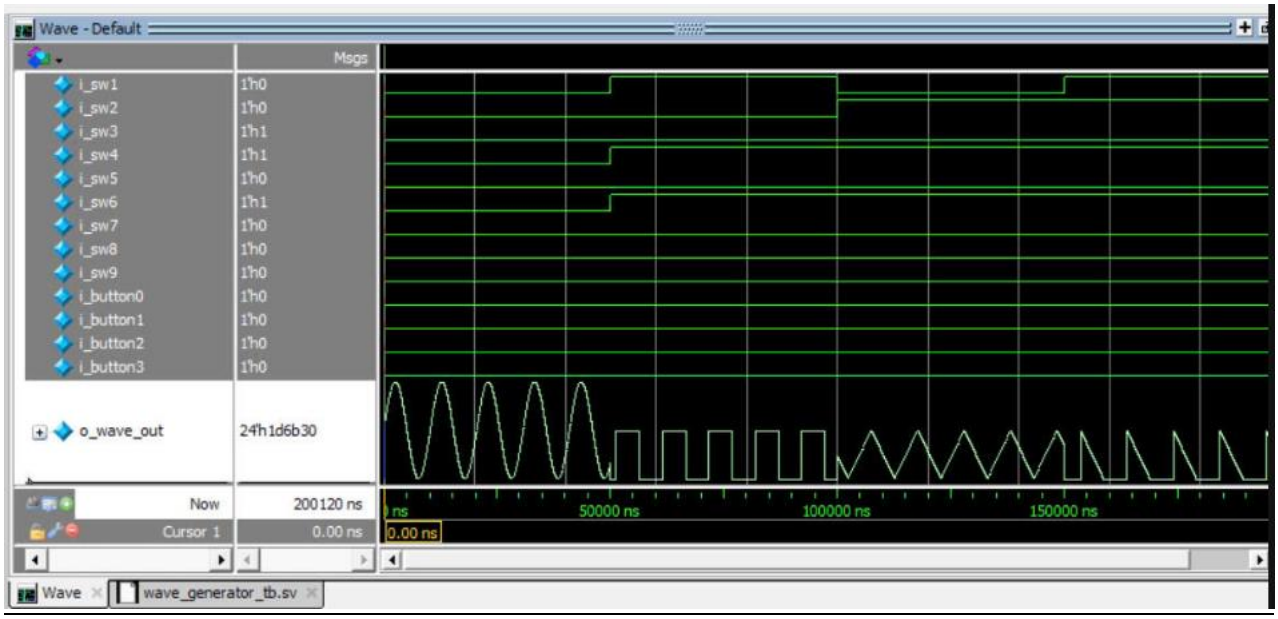
* Gạt SW0 xuống, nhấn KEY3, KEY2, KEY1.

* Gạt SW0 lên, nhấn KEY0, KEY1, KEY2.

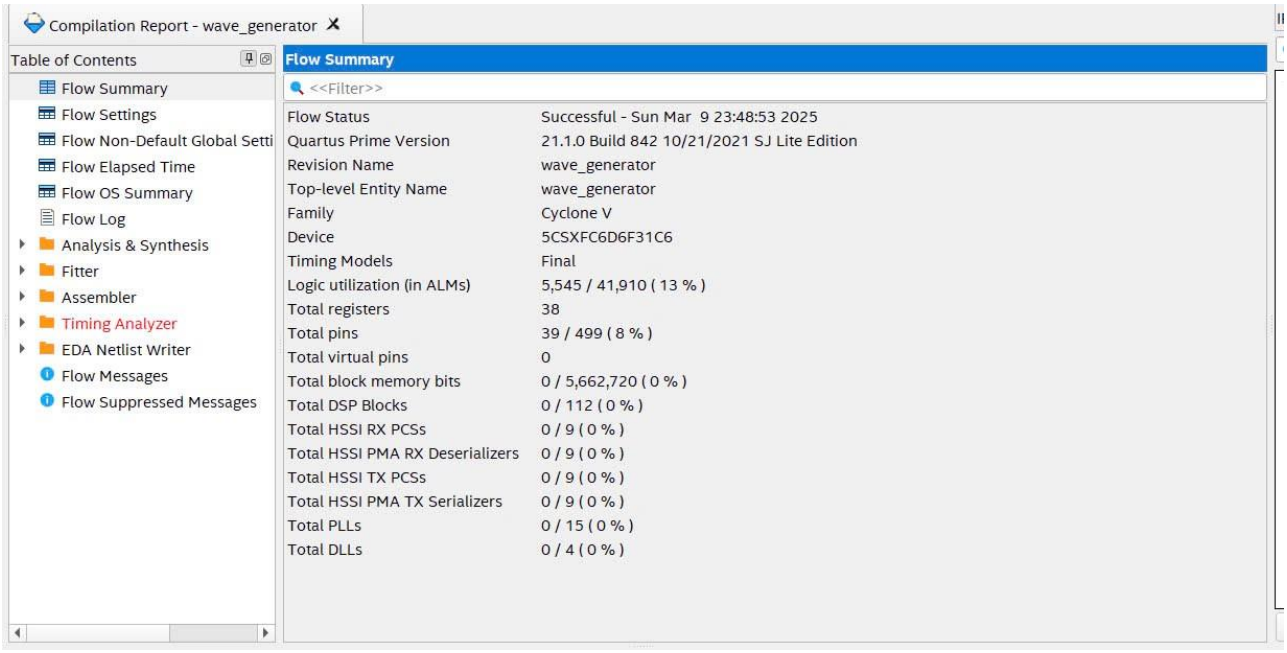
- Sau khi đã cấu hình xong I2C, để gửi dữ liệu lên bằng cách đọc các địa chỉ đã lưu sẵn của các mẫu, bằng cách nhấn các nút để thay đổi địa chỉ cần đọc ra.

V. Kết quả

1. Kết quả mô phỏng trên máy tính



2. Kết quả tổng hợp trên QUARTUS



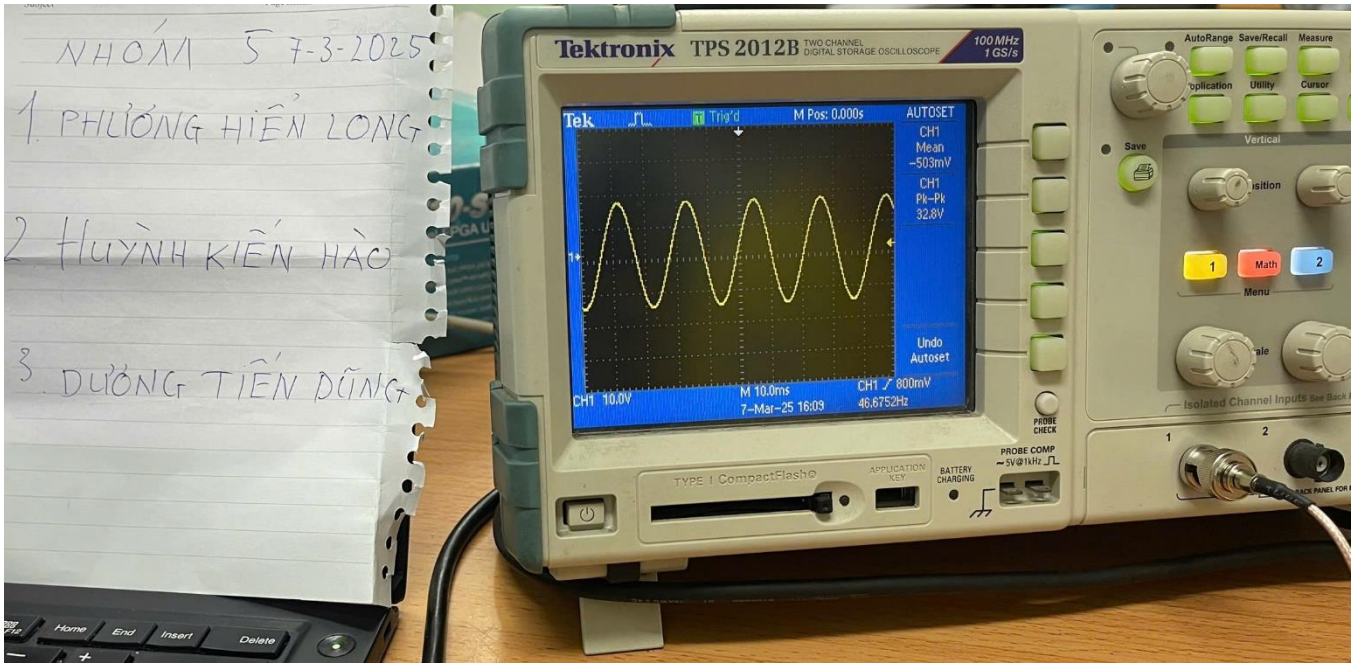
3. <u>Kết quả tạo sóng bằng DE10 Standard KIT</u>

Hiện tại nhóm chúng em đã hiện được các dạng sóng khác nhau, dựa trên các switch.

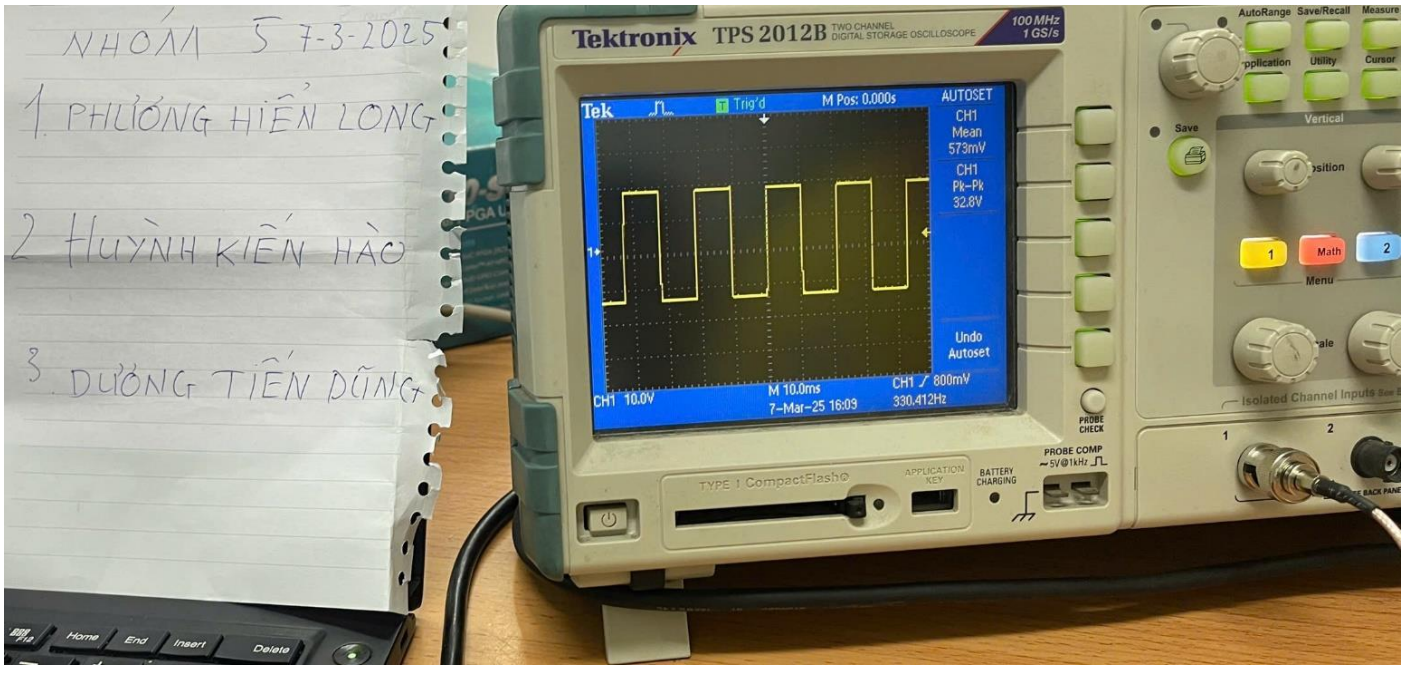
Tổ hợp phím để lựa chọn sóng:

Switch 8: gạt lên là reset, gạt xuống là hiện sóng.

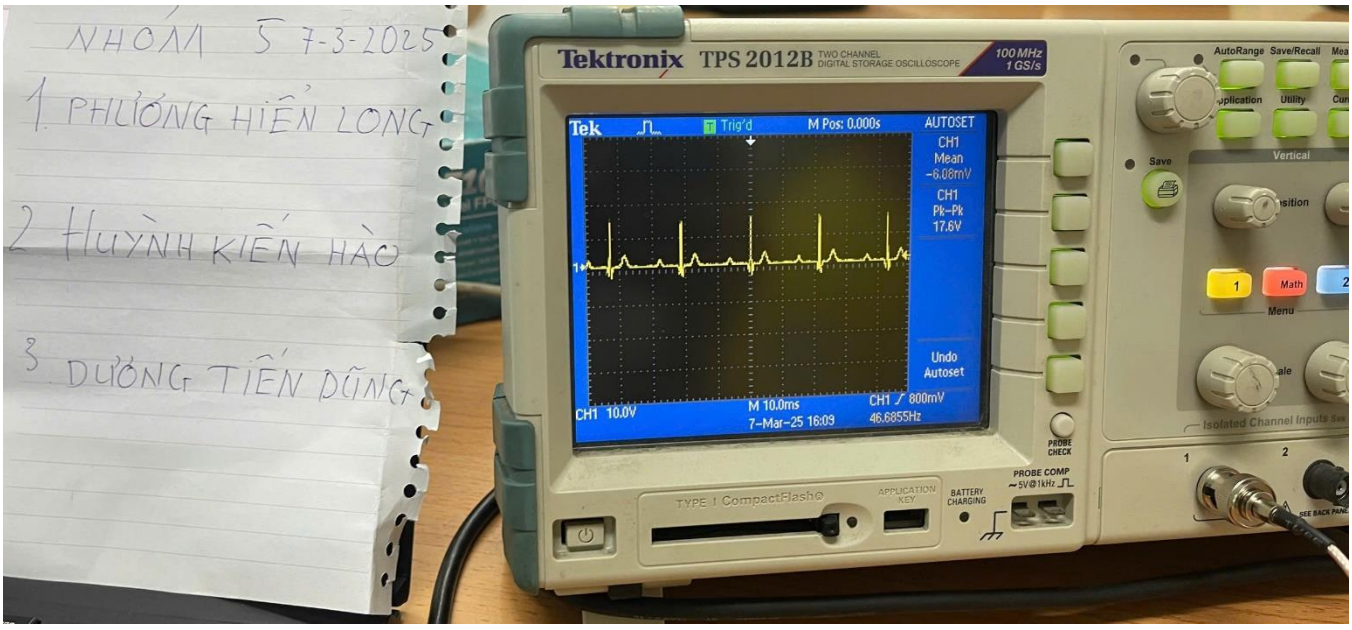
Switch 1: gạt lên là hiện sóng sin



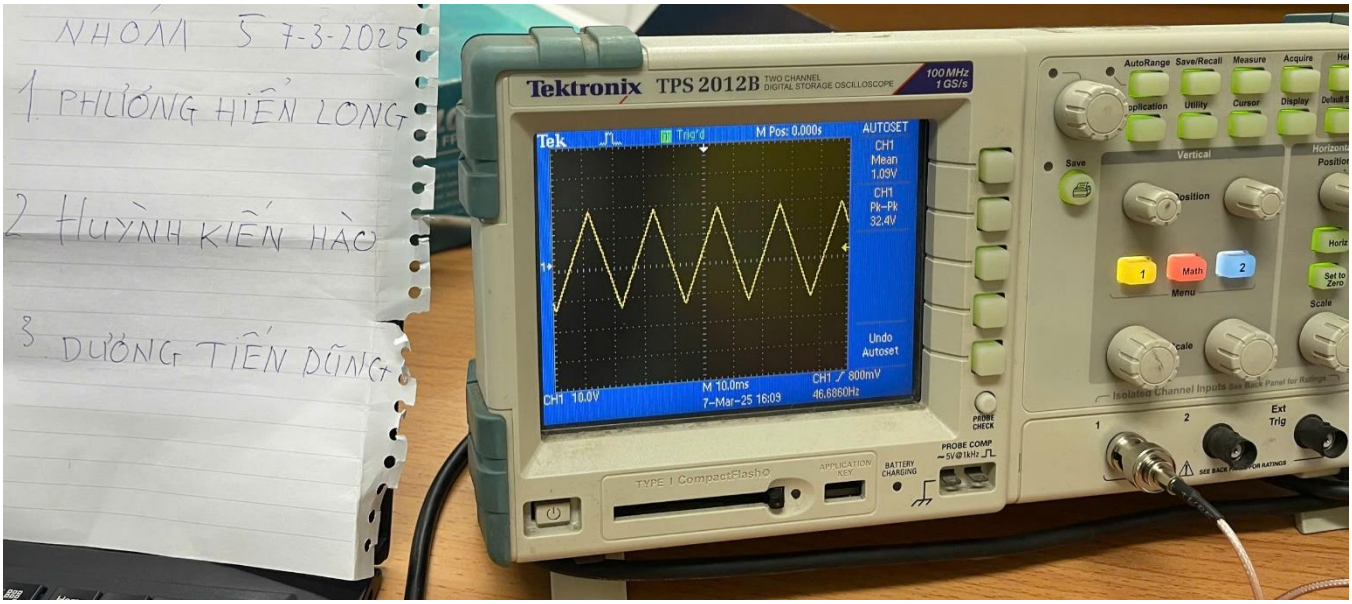
Switch 2: gạt lên là hiện sóng vuông



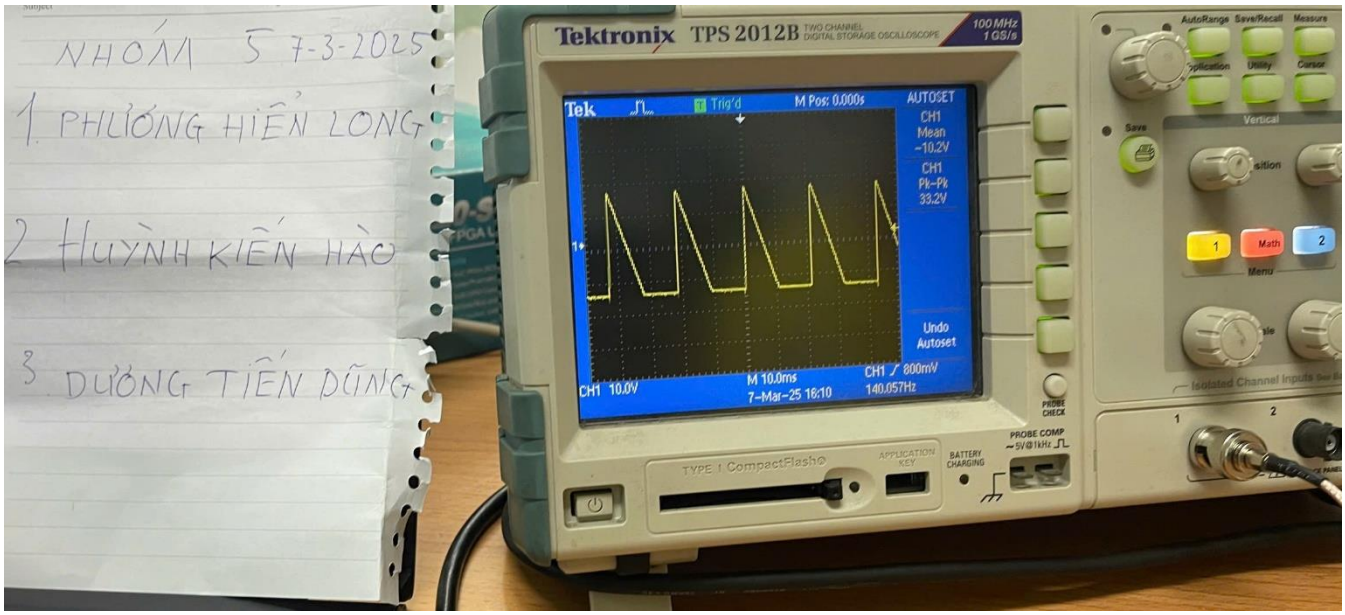
Switch 5: gạt lên là hiện sóng ECG



Switch 3: gạt lên là hiện sóng tam giác



Switch 4: gạt lên là hiện sóng răng cưa



TỔNG KẾT BÀI TẬP LỚN

Sau khi hoàn thành bài tập lớn nêu trên nhóm đã có thể code ra được một chương máy tạo sóng hoàn chỉnh khi có thể tạo ra được các dạng sóng khác nhau và mô phỏng chương trình thành công trên các phần mềm mô phỏng.

Nhóm cũng đã được rèn luyện khả năng tư duy lập trình với các ngôn ngữ mô tả phần cứng như Verilog, SystemVerilog và VHDL. Đồng thời nhóm chúng em đã có thêm kinh nghiệm trong việc sử dụng thành công các dạng giao thức (protocol) mà cụ thể trong bài tập lớn này là I2C cũng như các thiết bị thực tế là KIT DE10 Standard và dao động ký.

Tuy nhiên vẫn có những hạn chế mà nhóm hiện tại vẫn chưa thể khắc phục được. Trong đó quan trọng nhất là việc chưa tích hợp hoàn chỉnh chương trình máy phát sóng với các module giao tiếp với DAC WM8731 để có thể hiện dạng sóng và điều chỉnh chúng bằng KIT thực tế, hiện tại nhóm chỉ có thể hiển thị và chọn các sóng bằng KIT bằng một source code VHDL có sẵn.

Thứ hai là việc các module tạo sóng của nhóm chỉ dùng LUT để ghi nhớ giá trị và tạo ra dạng sóng, điều này vô tình làm số lượng bộ nhớ tiêu hao của chương trình chiếm số lượng lớn hơn các thủ thuật tạo sóng khác.

Hy vọng trong tương lai nhóm sẽ có thể nâng cao hơn kỹ năng của mình đồng thời có thể hoàn thiện bài tập lớn này một cách tốt hơn.