AUDIO SPECTRUM

Módulo de visualización

https://github.com/Felipe2395/Matrix_RGB

Escobar P. Andres Felipe, Guerrero C. Anderson Danilo, Martinez P. Herzon Andres

Grupo 9

INTRODUCCIÓN

Este módulo se encarga de recibir arreglos que describen amplitudes de 32 bandas de frecuencia, con el objetivo de graficarlas en una matriz led RGB, en un rango de 0 a 7 y ordenadas de menor a mayor frecuencia. Dando como resultado el espectro del audio en función de la frecuencia.

DESCRIPCIÓN

- Recibir el arreglo de datos, que describe la amplitud de cada frecuencia.
- Dividir el arreglo en 32 partes, con el fin de separar cada frecuencia.
- Codificar los 3bits en 8 bits de cada frecuencia.
- Formar una matriz de 8x32 apilando las frecuencias de todo el arreglo.
- Graficar la matriz de 8x32 en la matriz led RGB.

REQUERIMIENTOS OPERACIONALES

MATRIX LED RGB

Este panel tiene 512 LEDs RGB de media potencia, los cuales consumen un máximo de2.5A a una frecuencia de reloj de 50MHz.

Se requieren 12 pines digitales (6 bit data, 6 bit control) y una fuente de 5V DC a 3A por panel.

Esta matriz tiene una gama de 512 colores, incluyendo el blanco y el negro.

DIAGRAMA DE CAJA NEGRA

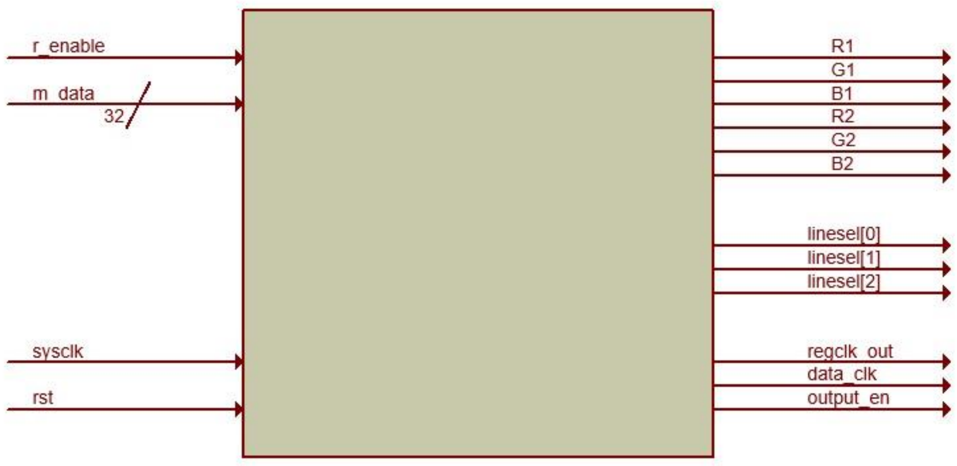
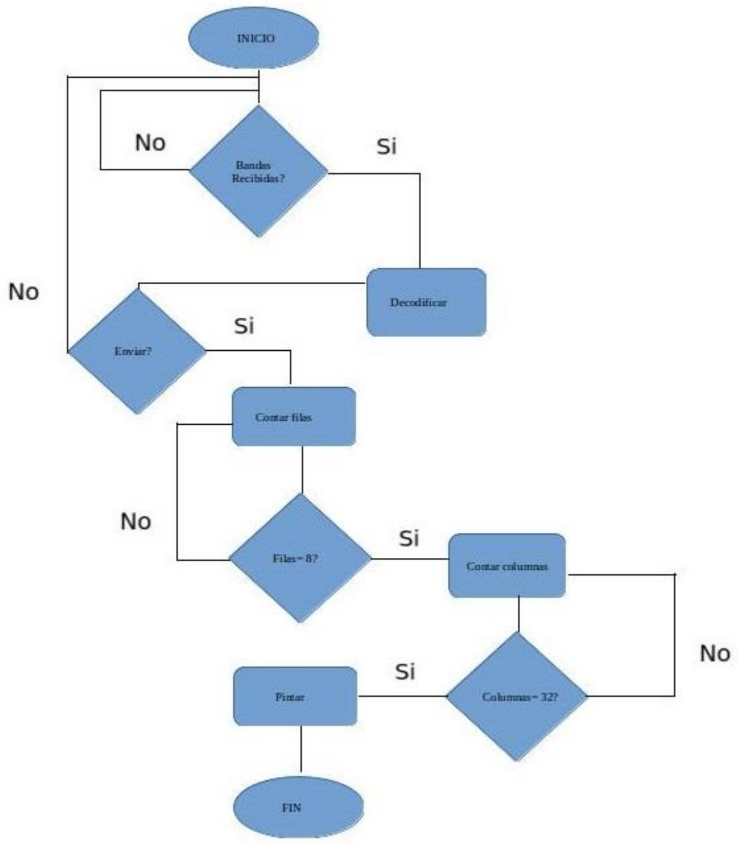
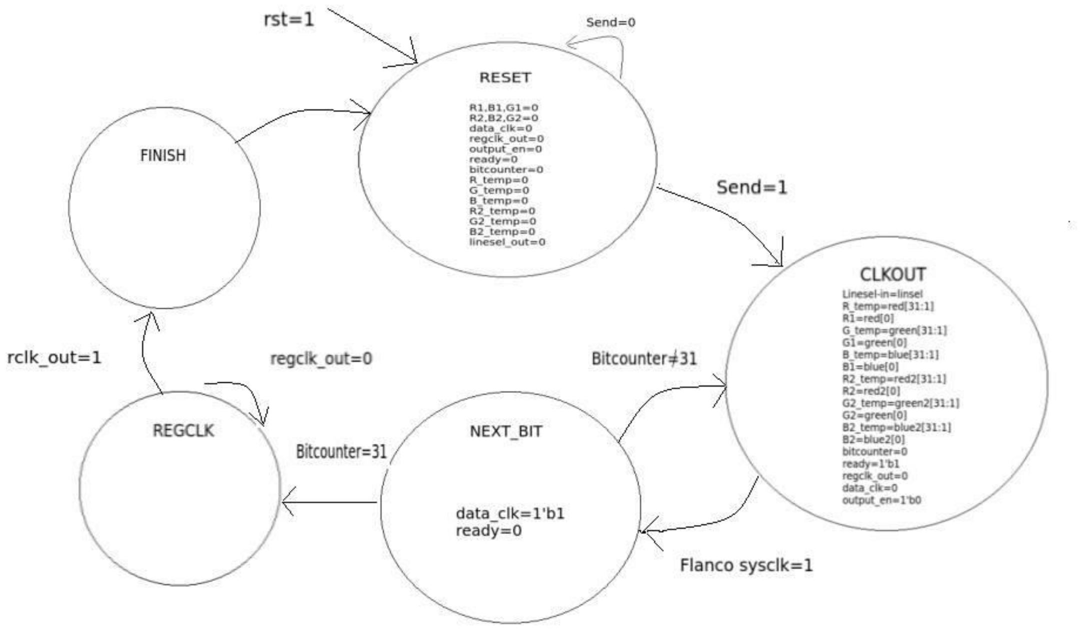


DIAGRAMA DE FLUJO



MÁQUINA DE ESTADOS



CÓDIGO DEL SUB-MÓDULO DE DECODIFICACIÓN

```
`timescale 1ns / 1ps

module Data_codec(
        input r_enable,
        input [0:95] m_data,
        input [0:2] linesel,

        output reg linesel_en,
        output reg [0:31]red,
        output reg [0:31]red2,
        output send,
        output enable
   );



        integer i, u_i=0, u_f=2, a;
        reg[0:7]matrix_R[31:0];
        reg[0:7]data_c;
        reg [0:95] data_desp;

initial begin
data_desp<= m_data;

        while(r_enable==1) begin
linesel_en<=0;
for (i=0;i<=31;i=i+1) begin

a<=data_desp[0:2];
data_desp<= data_desp<<3;
        //u_i<=u_i+2'd3;
        //u_f<=u_f+2'd3;
//      a<=m_data[u_i];
//      u_i<=u_i+2'd3;
//      u_f<=u_f+2'd3;

        case(a)
                0: data_c=8'd128;
                1: data_c=8'd192;
                2: data_c=8'd224;
                3: data_c=8'd240;
                4: data_c=8'd248;
                5: data_c=8'd252;
                6: data_c=8'd254;
                7: data_c=8'd255;
                default: data_c=8'd255;
        endcase

        matrix_R[i]=data_c;
```

```
	end //for
	end //while

	linesel_en<=1;

	while(linesel==0) begin
		red=matrix_R[0][0:31];
		red2=matrix_R[7][0:31];
	end
	while(linesel==1) begin
		red=matrix_R[1][0:31];
		red2=matrix_R[6][0:31];
	end
	while(linesel==2) begin
		red=matrix_R[2][0:31];
		red2=matrix_R[5][0:31];
	end
	while(linesel==3) begin
		red=matrix_R[3][0:31];
		red2=matrix_R[4][0:31];
	end
	while(linesel==4) begin
		red=matrix_R[4][0:31];
		red2=matrix_R[3][0:31];
	end
	while(linesel==5) begin
		red=matrix_R[5][0:31];
		red2=matrix_R[2][0:31];
	end
	while(linesel==6) begin
		red=matrix_R[6][0:31];
		red2=matrix_R[1][0:31];
	end
	while(linesel==7) begin
		red=matrix_R[7][0:31];
		red2=matrix_R[0][0:31];
	end
end

endmodule
```

Código del sub-módulo linesel

```
`timescale 1ns / 1ps

module linesel_count(

	input linesel_en,
	output reg linesel
    );
integer j;
initial begin
	while(linesel_en==1)begin
		linesel=0;
		for(j=0;j<=7;j=j+1) begin
			linesel=j;
		end
		linesel=0;
	end
end
endmodule
```

CÓDIGO DEL DRIVER DE LA MATRIZ

```
timescale 1ns / 1ps
//////////////////
module DriverRGB(input sysclk,
   input [31:0] red,
            input [31:0] green,
            input [31:0] blue,
            input [31:0] red2,
            input [31:0] green2,
            input [31:0] blue2,
            input [2:0] linesel,
            input rst,
            input send,
            input enable,
            output reg R1, G1, B1, R2, G2, B2,
            output reg [2:0] linesel_out,
            output reg data_clk,
            output reg regclk_out,
            output reg output_en,
   output reg ready

   );

            reg [4:0] bitcounter  = 0;
            reg [30:0] R_temp;
            reg [30:0] G_temp;
            reg [30:0] B_temp;
            reg [30:0] R2_temp;
            reg [30:0] G2_temp;
            reg [30:0] B2_temp;
            reg [2:0] linesel_in;

//analyze analyze_instance_name(
//          r_enable, m_data, a );
//
//codec codec_codec_name(
//          a, data_codec );
//
//matrix_draw(
//          linesel_in, red, green, blue, red2, green2,
blue2 );
//
//count count_count_name(
//          clk, rst, linesel_in);

/*Data_codec instance_name (
   .r_enable(r_enable),
   .m_data(m_data),
   .linesel(linesel),
   .linesel_en(linesel_en),
   .red(red),
   .red2(red2),
   .send(1'b1),
   .enable(enable)
   );
*/
            parameter st_RESET =   5'b00001;
  parameter st_CLKOUT =  5'b00010;
           parameter st_NEXTBIT = 5'b00100;
  parameter st_REGCLK =  5'b01000;
  parameter st_FINISH =  5'b10000;

           (* FSM_ENCODING="ONE-HOT",
SAFE_IMPLEMENTATION="NO" *) reg [4:0]
state = st_RESET;
always@(posedge sysclk)
     if (rst) begin
       state <= st_RESET;
       R1 <= 0;
                            G1 <= 0;
                            B1 <= 0;
                            R2 <= 0;
                            G2 <= 0;
                            B2 <= 0;
                            data_clk <= 0;
                            regclk_out <= 0;
                            output_en <= 1; //Active
low signal - display off during reset.
                            ready <= 1;
                            bitcounter <= 0;
                            R_temp <= 0;
                            G_temp <= 0;
                            B_temp <= 0;
                            R2_temp <= 0;
                            G2_temp <= 0;
                            B2_temp <= 0;
                            linesel_out <= 0;
     end
     else
       (* FULL_CASE, PARALLEL_CASE *) case
(state)
          st_RESET : begin

                                                 if
(!send)
             state <= st_RESET;                  //No
data for us to send yet.
           else
             state <= st_CLKOUT;                 //
Change state, now transmitting.
```

```
            linesel_in <= linesel;                    //
Register the line selects.
                                                R_temp
<= red[31:1];                        // Register the 31
lsb's of the longs we're sending.
                                                R1 <=
red[0];                                         // First
bit to send is MSB.
                                                G_temp
<= green[31:1];              // Register the 31 lsb's of
the longs we're sending.
                                                G1 <=
green[0];                                       // First
bit to send is MSB.
                                                B_temp
<= blue[31:1];               // Register the 31 lsb's of
the longs we're sending.
                                                B1 <=
blue[0];                                        // First
bit to send is MSB.
                                                R2_temp
<= red2[31:1];               // Register the 31 lsb's of
the longs we're sending.
                                                R2 <=
red2[0];                                        // First
bit to send is MSB.

            G2_temp <= green2[31:1];            //
Register the 31 lsb's of the longs we're sending.
                                                G2 <=
green2[0];                                      // First
bit to send is MSB.
                                                B2_temp
<= blue2[31:1];              // Register the 31 lsb's of
the longs we're sending.
                                                B2 <=
blue2[0];                                       // First
bit to send is MSB.


            bitcounter <= 0;
// We'll be sending the first bit.
                                                ready <=
1'b1;
            // Still waiting for input.

            regclk_out <= 0;
                                                data_clk
<= 0;
```

```
            output_en <= 1'b0;
                                     // Turn output back on.
          end
st_CLKOUT : begin
             state <= st_NEXTBIT;

                                                data_clk
<= 1'b1;
                                                ready <=
0;
            // Signal that we're not accepting input right
now.
          end

                                        st_NEXTBIT :
begin
             if (&bitcounter)                 // We're done
after this bit AND we need to send a regclk out.
                state <= st_REGCLK;
             else
                state <= st_CLKOUT;     // We're not
done.  Send another clock bit next cycle.

                                                data_clk
<= 1'b0;
            // Lower the data clock.
                                                if
(!(&bitcounter)) begin
            // Only do these next things if we're not
done yet.
                                                  R1 <=
R_temp[0];
            // Send the next bit.
                                                  R_temp
<= {1'b0,R_temp[30:1]};              // Shift remaining
bits right.
                                                  G1 <=
G_temp[0];
            // Send the next bit.

G_temp <= {1'b0,G_temp[30:1]};              // Shift
remaining bits left.
                                                  B1 <=
B_temp[0];
            // Send the next bit.
                                                  B_temp
<= {1'b0,B_temp[30:1]};              // Shift remaining
bits left.
                                                  R2 <=
R2_temp[0];
            // Send the next bit.
```

```
R2_temp <= {1'b0,R2_temp[30:1]};
            // Shift remaining bits left.
                                                              G2 <=
G2_temp[0];
            // Send the next bit.

G2_temp <= {1'b0,G2_temp[30:1]};
            // Shift remaining bits left.
                                                              B2 <=
B2_temp[0];
            // Send the next bit.

B2_temp <= {1'b0,B2_temp[30:1]};
            // Shift remaining bits left.

bitcounter <= bitcounter + 1'b1;                  //
Increment bit count.
                                                            end
                                                            else

            output_en <= 1'b1;           // Turn off output
while we change the latch and the linesel.
          end

                                                st_REGCLK :
begin
             if (!regclk_out)
                                        // Pulse the register clock
once when we get here.
```

```
               state <= st_REGCLK;
             else
               state <= st_FINISH;
                        // Then move on to tell the caller
we're ready for a new byte.


             regclk_out <= ~regclk_out;

                                                      if
(!regclk_out)

linesel_out <= linesel_in;

           end

                                          st_FINISH :
begin
             state <= st_RESET;
                              // Go back to the
beginning.
                                                      ready <=
1'b1;
                                                      end
        endcase



endmodule
```