ECE241 Project Report

DE1-SoC Digital Synthesizer

Danja Papajani, KC Tremblay, Shakiba Tonekaboni

4 December 2017

Introduction

Inspired by the mathematics of various harmonics in music, we set out for a project that deviated from the norm of a hardware video game, and decided to design and describe an digital synthesizer on the DE1-SoC board. Our design calls for a functionality such that the user inputs three sequences of notes, and upon their request, loop these sequences over one another, similar to a "launchpad" that many music producers use. We made use of the DE1-SoC elements as follows:

SWITCHES	ASSIGNMENT
[0]	MIDDLE C NOTE
[1]	D NOTE
[2]	E NOTE
[3]	F NOTE
[4]	G NOTE
[5]	A NOTE
[6]	B NOTE
[7]	1.HIGH C NOTE 2.FIRST SEQUENCE OF NOTES
[8]	SECOND SEQUENCE OF NOTES
[9]	1.newclk2 2. THIRD SEQUENCE OF NOTES

Table 1

HEX	ASSIGNMENT
[0]	RAMSLOTR1
[1]	RAMSLOTR2
[2]	RAMSLOTR3

Table 2

KEYS	ASSIGNMENT
[0]	RESET
[1]	N/A
[2]	STORE NOTE
[3]	GO SIGNAL

Table 3

As the user inputs a sequence of notes into it's respective RAM instantiation, HEX0, HEX1, and HEX2 will output the address at which the user is storing the note, while the VGA will output the note that is being stored. For example, if Middle C Note was inserted into address one of the third RAM (i.e. third sequence of notes), HEX2 will show a "1" and the computer screen will display the letter "C". The user will input up to five notes into each RAM instantiation, by pressing KEY[2] to store the note and KEY[3] to advance to the next address. Once the user has finished storing all their notes, they will reach a state in which all the RAM addresses are reset to "1"; this can be seen through our HEX [2:0]. When the user is ready, they can use SW [9:7] to output their respective sequence of notes, which will play in a loop and overtop one another. Throughout the development process, and in our final presentation, we assigned LEDR [9:0] to our FSM's current state.

The Design

The final design of our project was a user-dependant, 5-note audio playback mixer. The design of the project linearly flows from user input to synthesis to audio output. Internally, the project stores and retrieves data decided by the user and then creates a square wave based on this data. This square wave is then sent to the audio output to be heard as sound at different frequencies.

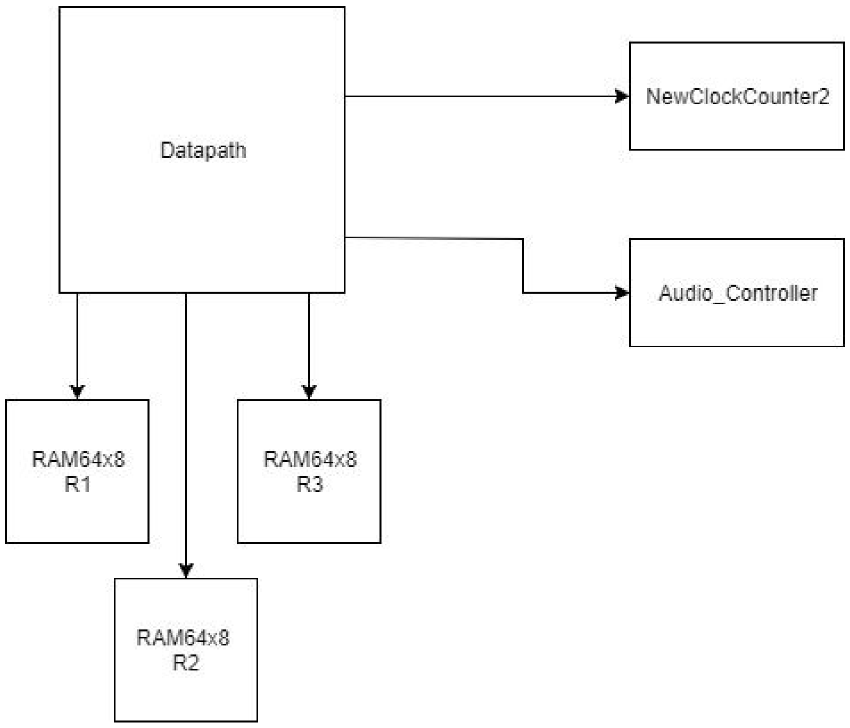

Figure 1

From the top-level module (**DJVerilog**), the datapath takes the speed determined by the user via SW[9]. The speed value is saved in a wire called **speed2** which triggers the module **NewClockCounter2** on the positive edge of KEY[3]. Next, the FSM traverses through 11 states for storing data, where each sub-state for waiting increments the RAM address by one bit. There are three instantiations of this sequence, for the three separate RAM modules. Once the speed and note values are stored, the datapath enables a play-back state where the new clock speed (**NewClockCounter2**) is enabled, reading out RAM data is enabled and a square wave generator block is enabled. The play_back state for the RAMs are controlled by the user via SW[9:7], where the user decides what RAM sequence to output first. All of this data is then sent to the **Audio_Controller** module which uses the FPGA board's audio decoder to output sound to the speakers via the "line-out" audio jack. Each square wave output was named soundR1, soundR2, and soundR3 and added to the "line-out" audio jack, alloweing the waves to be played over one another.

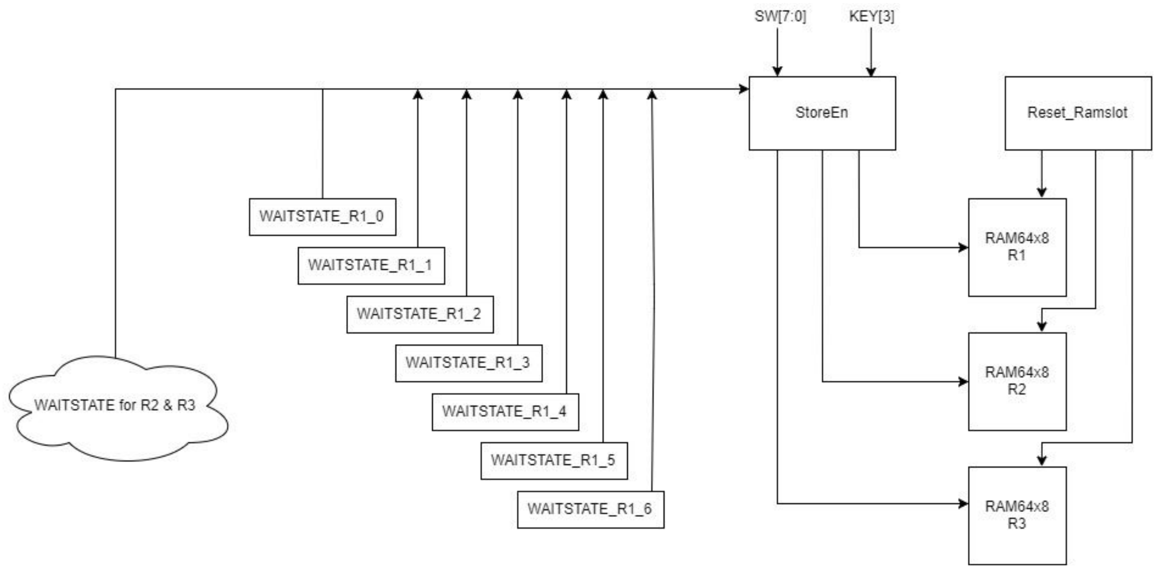

Figure 2

Storing the play-back speed and 5 notes happens through 5 states and their respective sub-wait-states (**WaitState0-6)**. This is shown in Figure 2. Each wait state increments the address of RAM memory by a 1 bit left shift. The **PlayEn** state takes the switch configuration of switches 0 to 7 and stores it into **RAM64x8** on the positive edge of KEY[3]. This is done 5 times for 5 note inputs, 15 notes in total. At the Beginning and end of storing the speed and 5 note values for each RAM instantiation, **Reset_Ramslot** is enabled, which sets the RAM addresses back to the first address (5'b1).

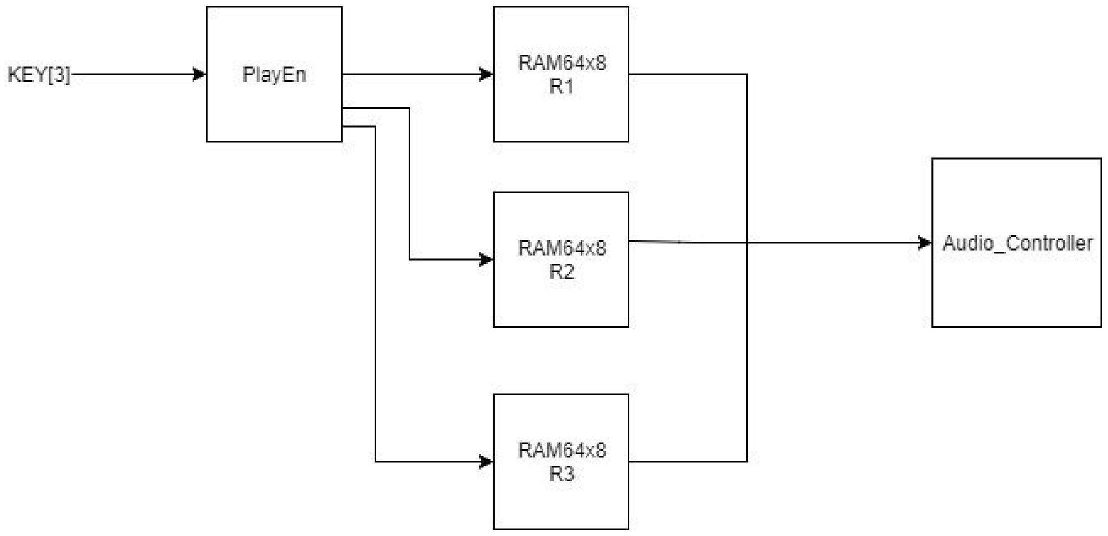

Figure 3

Once the speed and notes are inputted and stored, **PlayEn** is enabled, which allows the stored note values to be read out of RAM memory at the selected speed. This data is then connected to a state table that enables one of 8 possible delay values. These delay values each create a different square waveform, as seen in Figure 4. This waveform is connected to the **Audio_Contoller** input, which converts it from digital to analog signals via the audio coder/decoder. This is then outputted as audio sound. Figure 3 shows this logic.

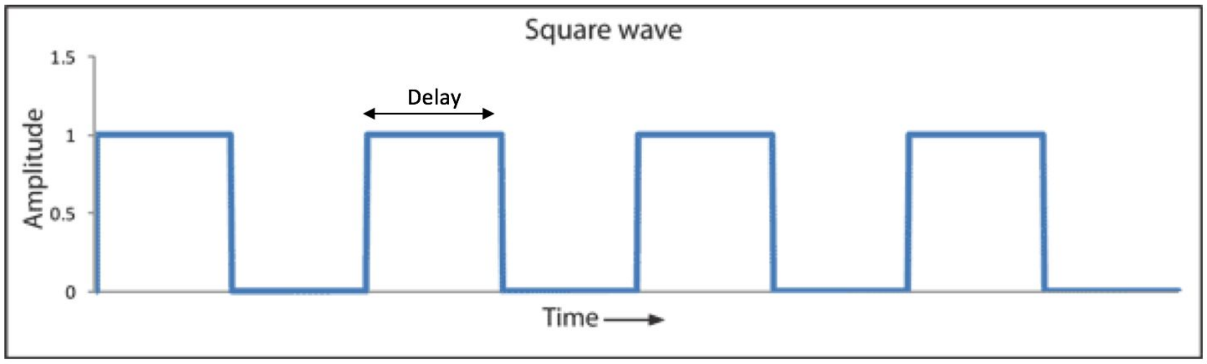

Figure 4

The **Audio_Contoller** module was provided by Professor J. Rose of the University of Toronto ECE department. Although it was not written by us for this project, the essential function that it performed for our design was taking serial, binary data and sending it to the FPGA board's audio decoder. This decoder is built into the board and it is able to convert binary data (digital) into audio output (analog). This involves taking the square wave signal from the time domain into the frequency domain, where it can be represented by a sinusoidal function. Being able to control and send the correct data to this module and decoder was the primary focus of this project.

For the VGA module, we had a counter for both the **x_plot** and **y_plot** signal that in turn also increments the address of the RAM. Each note had their own respective image.mif file, which was called using a **case_statement** depending on which note (SW[7:0]) the user inputted. Each .mif file had their own RAM instantiation. The module uploads the image onto the screen using "left to right" logic- we first increment the **x_count**, and once it reaches it's maximum value, we increment the **y_count** by one. Due to the limited memory on an FPGA chip, we were limited to using 3 colour bits per channel.

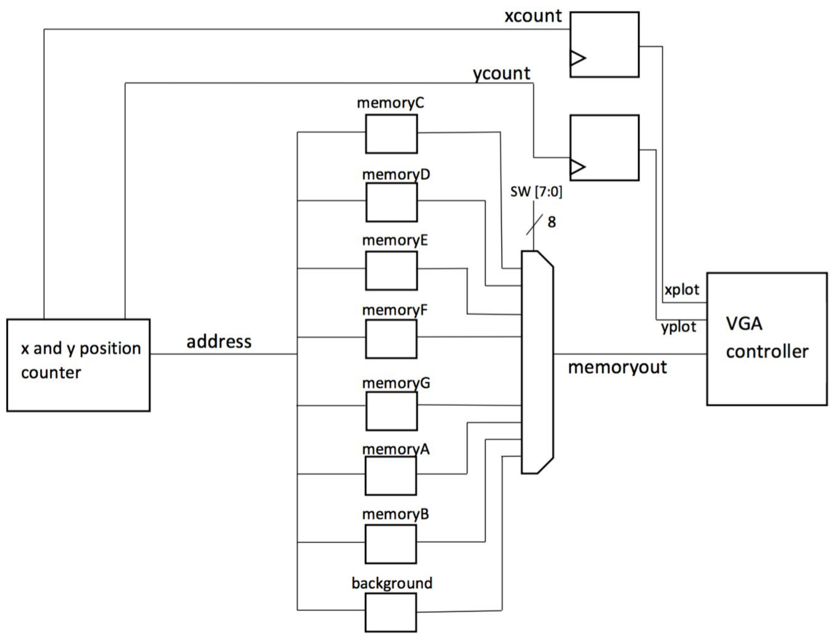

Figure 4

Report on Issues

The only issue encountered throughout the design process of "DJ Verilog", other than minor bugs, was the team's initial lack of experience with the VGA. The team had minimal exposure to the VGA from Lab 7 and had to spend more time to close this knowledge gap. As a team we conceptually understood the logic behind the VGA functionality but experienced issues with how to write the Verilog implementation of our schematic. We were unsure how to connect the multiple components of our schematic, but after consulting with our TA we decided the easiest approach would be to create one large module that contained all counters, registers, and RAMs.

After we had established a structure, we had written non-functional code describing our position counter. This was likely due to our incomplete understanding of the VGA and our minimal experience with Verilog. Initially, the counter incremented the x position and y position concurrently, resulting in the outputted pixels of the selected RAM to be "drawn" diagonally across the VGA monitor. We realized to properly display each 160x120 pixel image, the code should increment the x position from 0 to 159, then increment the y position by 1 pixel, and then repeat this cycle 119 times. Also, we had initially planned on using an image resolution of 320x240 with 3 bits per channel (512 colour combinations), but then ran out of available space on the DE1-SoC. We had to compromise by decreasing the resolution to ensure sufficient space for each of the 8 RAM instantiations. As a whole, the team experienced very few issues with only a few minor setbacks regarding the VGA.

What Would you do Differently

Milestone Indications

While creating the milestones, the team underestimated the VGA component of the project. For the first milestone, the VGA should have been able to display the correct image based on the switch selected, however we were only able to display the background image by the first lab. This resulted in the whole VGA component of the project to be delayed. We should have started planning for the VGA and created our schematic sooner, so we could have met the first VGA milestone and later on could have added an extra element of difficulty to increase the complexity of the project.

Appendix A

This URL directs to the page from which we obtained the audio controller code:
http://www.eecg.toronto.edu/~jayar/ece241_08F/AudioVideoCores/audio/audio.html
http://www.eecg.toronto.edu/~jayar/ece241_08F/AudioVideoCores/audio/src/Audio_Demo.zip

Appendix B

```
//****************************************
//*                                      *
//*          TOP LEVEL MODULE            *
//*                                      *
//****************************************
module DJVerilogWorking (
        // Inputs
        CLOCK_50,
        KEY,
        LEDR,
        SW,
                HEX0,
                HEX1,
                HEX2,
        AUD_ADCDAT, // (DECODEC)
        // Bidirectionals (from DECODEC)
        AUD_BCLK,
        AUD_ADCLRCK,
        AUD_DACLRCK,
        FPGA_I2C_SDAT,
        // Outputs (DECODEC)
        AUD_XCK,
        AUD_DACDAT,
        FPGA_I2C_SCLK,
                VGA_CLK,                                    //      VGA Clock
                VGA_HS,                                            //      VGA H_SYNC
                VGA_VS,                                            //      VGA V_SYNC
                VGA_BLANK_N,                                //      VGA BLANK
                VGA_SYNC_N,                                 //      VGA SYNC
                VGA_R,                                      //      VGA Red[9:0]
                VGA_G,                                             //      VGA Green[9:0]
                VGA_B

);
//***************************************
//*                                     *
//*         PORT DECLERATIONS           *
//*                                     *
//***************************************
       input CLOCK_50;
       input [3:0] KEY;
       input [9:0] SW;
       input AUD_ADCDAT;
       // Bidirectionals
       inout AUD_BCLK;
       inout AUD_ADCLRCK;
       inout AUD_DACLRCK;
       inout FPGA_I2C_SDAT;
       // Outputs
       output AUD_XCK;
       output AUD_DACDAT;
       output FPGA_I2C_SCLK;
       output [9:0] LEDR;
       output [6:0] HEX0, HEX1, HEX2;
       output VGA_CLK;                         //      VGA Clock
       output VGA_HS;                          //      VGA H_SYNC
       output VGA_VS;                          //      VGA V_SYNC
       output VGA_BLANK_N;                     //      VGA BLANK
       output VGA_SYNC_N;                      //      VGA SYNC
       output [9:0] VGA_R;                            //      VGA Red[9:0]
       output [9:0] VGA_G;                            //      VGA Green[9:0]
       output [9:0] VGA_B;                            //      VGA Blue[9:0]


//***************************************
//*                                     *
//*     INTERNAL WIRES AND REGS         *
//*                                     *
//***************************************
```

```
// Internal Wires
	wire audio_in_available;
	wire [31:0] left_channel_audio_in;
	wire [31:0] right_channel_audio_in;
	wire read_audio_in;
	wire audio_out_allowed;
	wire [31:0] left_channel_audio_out;
	wire [31:0] right_channel_audio_out;
	wire write_audio_out;
	wire reset, go, clk;
	wire [7:0] ramOutputR1;
	wire [7:0] ramOutputR2;
	wire [7:0] ramOutputR3;

	reg [18:0] delay_cntR1;
	reg [18:0] delayR1;
	reg [18:0] delay_cntR2;
	reg [18:0] delayR2;
	reg [18:0] delay_cntR3;
	reg [18:0] delayR3;
	reg sndR1;
	reg sndR2;
	reg sndR3;
	reg wren;
	reg [7:0] playNoteR1;
	reg [7:0] playNoteR2;
	reg [7:0] playNoteR3;
	reg [6:0] ramSlotR1;
	reg [6:0] ramSlotR2;
	reg [6:0] ramSlotR3;
	reg [7:0] note;
	reg [7:0] ramDataR1;
	reg [7:0] ramDataR2;
	reg [7:0] ramDataR3;

// Assignment Statements
	assign reset =  ~KEY[0];
	assign go    =  ~KEY[3];
	assign clk   =   CLOCK_50;

//VGA
	wire [14:0] address;
	reg [7:0] xcount;
	reg [6:0] ycount;
	assign resetn = KEY[0];
	assign plot = 1'b1;
	reg [7:0] x_plot;
	reg [6:0] y_plot;
	reg  [1:0] current_state2, next_state2;
	reg [8:0] memoryout;
```

```
//**************************************
//*                                    *
//*       SQUARE WAVE GENERATOR 1      *
//*                                    *
//**************************************

       always @(posedge clk)
                   if(delay_cntR1 == delayR1) begin
                                    delay_cntR1 <= 0;
                                    sndR1 <= !sndR1;
                   end else delay_cntR1 <= delay_cntR1 + 1;

       always @(*) begin

                   case(playNoteR1)
                                    8'b00000001: delayR1 = 19'd97304;
                                    8'b00000010: delayR1 = 19'd88804;
                                    8'b00000100: delayR1 = 19'd82304;
                                    8'b00001000: delayR1 = 19'd78554;
                                    8'b00010000: delayR1 = 19'd71054;
                                    8'b00100000: delayR1 = 19'd63554;
                                    8'b01000000: delayR1 = 19'd56054;
                                    8'b10000000: delayR1 = 19'd52304;
                                    default: delayR1 = 19'd0;
                   endcase
       end
       // Assign generated sound wave to output signal on Audio Controller
       wire [31:0] soundR1 = (playNoteR1 == 8'd0) ? 0 : sndR1 ? 32'd10000000 : -32'd10000000;

//**************************************
//*                                    *
//*       SQUARE WAVE GENERATOR  2     *
//*                                    *
//**************************************

       always @(posedge clk)
                   if(delay_cntR2 == delayR2) begin
                                    delay_cntR2 <= 0;
                                    sndR2 <= !sndR2;
                   end else delay_cntR2 <= delay_cntR2 + 1;

       always @(*) begin

                   case(playNoteR2)
                                    8'b00000001: delayR2 = 19'd97304;
                                    8'b00000010: delayR2 = 19'd88804;
                                    8'b00000100: delayR2 = 19'd82304;
                                    8'b00001000: delayR2 = 19'd78554;
                                    8'b00010000: delayR2 = 19'd71054;
                                    8'b00100000: delayR2 = 19'd63554;
                                    8'b01000000: delayR2 = 19'd56054;
                                    8'b10000000: delayR2 = 19'd52304;
                                    default: delayR2 = 19'd0;
                   endcase
```

```
        end
        // Assign generated sound wave to output signal on Audio Controller
        wire [31:0] soundR2 = (playNoteR2 == 8'd0) ? 0 : sndR2 ? 32'd10000000 : -32'd10000000;

//*************************************
//*                                   *
//*      SQUARE WAVE GENERATOR  3     *
//*                                   *
//*************************************

        always @(posedge clk)
                        if(delay_cntR3 == delayR3) begin
                                    delay_cntR3 <= 0;
                                    sndR3 <= !sndR3;
                        end else delay_cntR3 <= delay_cntR3 + 1;

        always @(*) begin

                        case(playNoteR3)
                                    8'b00000001: delayR3 = 19'd97304;
                                    8'b00000010: delayR3 = 19'd88804;
                                    8'b00000100: delayR3 = 19'd82304;
                                    8'b00001000: delayR3 = 19'd78554;
                                    8'b00010000: delayR3 = 19'd71054;
                                    8'b00100000: delayR3 = 19'd63554;
                                    8'b01000000: delayR3 = 19'd56054;
                                    8'b10000000: delayR3 = 19'd52304;
                                    default: delayR3 = 19'd0;
                        endcase
        end
        // Assign generated sound wave to output signal on Audio Controller
        wire [31:0] soundR3 = (playNoteR3 == 8'd0) ? 0 : sndR3 ? 32'd10000000 : -32'd10000000;



//*************************************
//*                                   *
//*         AUDIO CONTROLLER          *
//*                                   *
//*************************************

        assign read_audio_in                    = audio_in_available & audio_out_allowed;
        assign left_channel_audio_out           =
left_channel_audio_in+soundR1+soundR2+soundR3; //add sounds here
        assign right_channel_audio_out          =
right_channel_audio_in+soundR1+soundR2+soundR3;
        assign write_audio_out                  = audio_in_available & audio_out_allowed;

// Instantiation of Audio Controller
Audio_Controller Audio_Controller (
        // Inputs
        .CLOCK_50                                           (CLOCK_50),
        .reset                                          (reset),
        .clear_audio_in_memory                (),
        .read_audio_in                                (read_audio_in),

        .clear_audio_out_memory                (),
        .left_channel_audio_out                (left_channel_audio_out),
        .right_channel_audio_out        (right_channel_audio_out),
        .write_audio_out                        (write_audio_out),
        .AUD_ADCDAT                                        (AUD_ADCDAT),
        // Bidirectionals
        .AUD_BCLK                                        (AUD_BCLK),
        .AUD_ADCLRCK                                (AUD_ADCLRCK),
        .AUD_DACLRCK                                (AUD_DACLRCK),
        // Outputs
        .audio_in_available                        (audio_in_available),
        .left_channel_audio_in                (left_channel_audio_in),
        .right_channel_audio_in                (right_channel_audio_in),
        .audio_out_allowed                        (audio_out_allowed),
        .AUD_XCK                                        (AUD_XCK),
        .AUD_DACDAT                                        (AUD_DACDAT)
);
// Instanatiation of Microphone input for Audio Controller
avconf #(.USE_MIC_INPUT(1)) avc (
        .FPGA_I2C_SCLK                                        (FPGA_I2C_SCLK),
        .FPGA_I2C_SDAT                                        (FPGA_I2C_SDAT),
        .CLOCK_50                                                (CLOCK_50),
        .reset                                                        (reset)
);

//**************************************
//*                                    *
//*            STATE TABLE             *
//*                                    *
//**************************************

       reg  [7:0] current_state, next_state;

       localparam                           RESET               = 8'd0,

                                       WAIT_STATE_R1_0    = 8'd1,

                                       STORE_SPEED        = 8'd2,

                                       WAIT_STATE_R1_1    = 8'd3,

                                       STORE_NOTE_R1_1    = 8'd4,

                                       WAIT_STATE_R1_2    = 8'd5,

                                       STORE_NOTE_R1_2    = 8'd6,

                                       WAIT_STATE_R1_3    = 8'd7,

                                       STORE_NOTE_R1_3    = 8'd8,
                                        WAIT_STATE_R1_4   = 8'd9,
                                        STORE_NOTE_R1_4   = 8'd10,
                                        WAIT_STATE_R1_5   = 8'd11,
                                        STORE_NOTE_R1_5   = 8'd12,
                                        WAIT_STATE_R1_6   = 8'd13,
                                        STORE_NOTE_R2_1   = 8'd14,
WAIT_STATE_R2_1   = 8'd15,
STORE_NOTE_R2_2   = 8'd16,
WAIT_STATE_R2_2   = 8'd17,
STORE_NOTE_R2_3   = 8'd18,
WAIT_STATE_R2_3   = 8'd19,
STORE_NOTE_R2_4   = 8'd20,
WAIT_STATE_R2_4   = 8'd21,
STORE_NOTE_R2_5   = 8'd22,
WAIT_STATE_R2_5   = 8'd23,
STORE_NOTE_R3_1   = 8'd24,
WAIT_STATE_R3_1   = 8'd25,
STORE_NOTE_R3_2   = 8'd26,
WAIT_STATE_R3_2   = 8'd27,
STORE_NOTE_R3_3   = 8'd28,
WAIT_STATE_R3_3    = 8'd29,

STORE_NOTE_R3_4    = 8'd30,

WAIT_STATE_R3_4    = 8'd31,

STORE_NOTE_R3_5    = 8'd32,

WAIT_STATE_R3_5    = 8'd33,
                                        RESET_RAMSLOT     = 8'd34,

                                        WAIT_FINAL        = 8'd35,

                                        PLAY_BACK         = 8'd36;



        always @(*)

        begin: state_table

                        case(current_state)

                                RESET: next_state = go ? WAIT_STATE_R1_0 : RESET;

                                WAIT_STATE_R1_0: next_state = go ? WAIT_STATE_R1_0 :
STORE_SPEED;

                                STORE_SPEED: next_state = go? WAIT_STATE_R1_1 : STORE_SPEED;

                                WAIT_STATE_R1_1: next_state = go ? WAIT_STATE_R1_1 :
STORE_NOTE_R1_1;

                                                                            STORE_NOTE_R1_1:
next_state = go ? WAIT_STATE_R1_2 : STORE_NOTE_R1_1;

                                WAIT_STATE_R1_2: next_state = go ? WAIT_STATE_R1_2 :
STORE_NOTE_R1_2;

                                STORE_NOTE_R1_2: next_state = go ? WAIT_STATE_R1_3 :
STORE_NOTE_R1_2;

                                WAIT_STATE_R1_3: next_state = go ? WAIT_STATE_R1_3 :
STORE_NOTE_R1_3;

                                STORE_NOTE_R1_3: next_state = go ? WAIT_STATE_R1_4:
STORE_NOTE_R1_3;
```

```
                                WAIT_STATE_R1_4: next_state = go ? WAIT_STATE_R1_4 :
STORE_NOTE_R1_4;
                                STORE_NOTE_R1_4: next_state = go ? WAIT_STATE_R1_5 :
STORE_NOTE_R1_4;
                                WAIT_STATE_R1_5: next_state = go ? WAIT_STATE_R1_5 :
STORE_NOTE_R1_5;
                                STORE_NOTE_R1_5: next_state = go ? WAIT_STATE_R1_6 :
STORE_NOTE_R1_5;
                                WAIT_STATE_R1_6: next_state = go ? WAIT_STATE_R1_6 :
STORE_NOTE_R2_1;
                                STORE_NOTE_R2_1: next_state = go ? WAIT_STATE_R2_1 :
STORE_NOTE_R2_1;
                                WAIT_STATE_R2_1: next_state = go ? WAIT_STATE_R2_1 :
STORE_NOTE_R2_2;
                                STORE_NOTE_R2_2: next_state = go ? WAIT_STATE_R2_2 :
STORE_NOTE_R2_2;
                                WAIT_STATE_R2_2: next_state = go ? WAIT_STATE_R2_2 :
STORE_NOTE_R2_3;
                                                                           STORE_NOTE_R2_3:
next_state = go ? WAIT_STATE_R2_3 : STORE_NOTE_R2_3;
                                                                           WAIT_STATE_R2_3:
next_state = go ? WAIT_STATE_R2_3 : STORE_NOTE_R2_4;
                                                                           STORE_NOTE_R2_4:
next_state = go ? WAIT_STATE_R2_4 : STORE_NOTE_R2_4;
                                                                           WAIT_STATE_R2_4:
next_state = go ? WAIT_STATE_R2_4 : STORE_NOTE_R2_5;
                                                                           STORE_NOTE_R2_5:
next_state = go ? WAIT_STATE_R2_5 : STORE_NOTE_R2_5;
                                                                           WAIT_STATE_R2_5:
next_state = go ? WAIT_STATE_R2_5 : STORE_NOTE_R3_1;
                                STORE_NOTE_R3_1: next_state = go ? WAIT_STATE_R3_1 :
STORE_NOTE_R3_1;
                                WAIT_STATE_R3_1: next_state = go ? WAIT_STATE_R3_1 :
STORE_NOTE_R3_2;
                                STORE_NOTE_R3_2: next_state = go ? WAIT_STATE_R3_2 :
STORE_NOTE_R3_2;
```

```
                                WAIT_STATE_R3_2: next_state = go ? WAIT_STATE_R3_2 :
STORE_NOTE_R3_3;
                                                                             STORE_NOTE_R3_3:
next_state = go ? WAIT_STATE_R3_3 : STORE_NOTE_R3_3;
                                                                             WAIT_STATE_R3_3:
next_state = go ? WAIT_STATE_R3_3 : STORE_NOTE_R3_4;
                                                                             STORE_NOTE_R3_4:
next_state = go ? WAIT_STATE_R3_4 : STORE_NOTE_R3_4;
                                                                             WAIT_STATE_R3_4:
next_state = go ? WAIT_STATE_R3_4 : STORE_NOTE_R3_5;
                                                                             STORE_NOTE_R3_5:
next_state = go ? WAIT_STATE_R3_5 : STORE_NOTE_R3_5;
                                                                             WAIT_STATE_R3_5:
next_state = go ? WAIT_STATE_R3_5 : RESET_RAMSLOT;

                                RESET_RAMSLOT: next_state = go ? WAIT_FINAL : RESET_RAMSLOT;

                                WAIT_FINAL: next_state = go ? WAIT_FINAL : PLAY_BACK;

                                PLAY_BACK: next_state = go ? RESET : PLAY_BACK;

                        default: next_state = RESET;

                endcase

        end


//**************************************
//*                                    *
//*          CONTROLPATH               *
//*                                    *
//**************************************

        always @(*)

                begin

                                 reset_ramSlotR1 = 1'b0;
                                     reset_ramSlotR2 = 1'b0;
                                     reset_ramSlotR3 = 1'b0;
                                     storeEnR1 = 1'b0;
                                     storeEnR2 = 1'b0;
                                     storeEnR3 = 1'b0;
                                     playEn = 1'b0;
```

```
                                                   waitStateR1_0 = 1'b0;
                                                   waitStateR1_1 = 1'b0;
                                                   waitStateR1_2 = 1'b0;
                                                   waitStateR1_3 = 1'b0;
                                                   waitStateR1_4 = 1'b0;
                                                   waitStateR1_5 = 1'b0;
                                                   waitStateR1_6 = 1'b0;
                                                   waitStateR2_1 = 1'b0;
                                                   waitStateR2_2 = 1'b0;
                                                   waitStateR2_3 = 1'b0;
                                                   waitStateR2_4 = 1'b0;
                                                   waitStateR2_5 = 1'b0;
                                                   waitStateR3_1 = 1'b0;
                                                   waitStateR3_2 = 1'b0;
                                                   waitStateR3_3 = 1'b0;
                                                   waitStateR3_4 = 1'b0;
                                                   waitStateR3_5 = 1'b0;
                                                   waitFinal = 1'b0;
                                                   //xander = 1'b0;


                            case(current_state)

                                    RESET:                          begin                                   reset_ramSlotR1 <=
1'b1;


reset_ramSlotR2 <= 1'b1;


reset_ramSlotR3 <= 1'b1;


                              end





                                    WAIT_STATE_R1_0:                begin


                                                                                                                    waitStateR1_0
<= 1'b1;
                                                                    end


                                    STORE_SPEED:                    begin

                                                                                                            speedEn <= 1'b1;


                                                                    end
```

```
                    WAIT_STATE_R1_1:            begin
                                                              waitStateR1_1 <= 1'b1;

                                                end

                    STORE_NOTE_R1_1:            begin
                                                              storeEnR1 <= 1'b1;


                       end


                    WAIT_STATE_R1_2:            begin
                                                              waitStateR1_2 <= 1'b1;
                                                end

                    STORE_NOTE_R1_2:            begin
                                                              storeEnR1 <= 1'b1;
                                                end

                    WAIT_STATE_R1_3:            begin


                                                                  waitStateR1_3
<= 1'b1;


                       end



                    STORE_NOTE_R1_3:            begin


                                                                  storeEnR1 <=
1'b1;


                       end
                    WAIT_STATE_R1_4:        begin
                                                                waitStateR1_4
<= 1'b1;
                                            end

                    STORE_NOTE_R1_4:        begin
                                                        storeEnR1 <= 1'b1;

                      end

                    WAIT_STATE_R1_5:        begin
                                                        waitStateR1_5 <= 1'b1;

                        end

                    STORE_NOTE_R1_5:        begin
                                                        storeEnR1 <= 1'b1;

                        end

                    WAIT_STATE_R1_6:        begin
                                                        waitStateR1_6 <= 1'b1;

                        end
                    STORE_NOTE_R2_1:        begin
                                                        storeEnR2 <= 1'b1;

                        end

                    WAIT_STATE_R2_1:        begin
                                                        waitStateR2_1 <= 1'b1;
                                            end
                        STORE_NOTE_R2_2:            begin
                                                                        storeEnR2 <= 1'b1;
                                                    end

                        WAIT_STATE_R2_2:            begin


                                                                                waitStateR2_2
<= 1'b1;


                              end



                        STORE_NOTE_R2_3:            begin


                                                                                storeEnR2 <=
1'b1;


                              end






                        WAIT_STATE_R2_3:            begin
                                                                                waitStateR2_3
<= 1'b1;
                                                    end


                        STORE_NOTE_R2_4:            begin

                                                                        storeEnR2 <= 1'b1;


                          end


                        WAIT_STATE_R2_4:            begin

                                                                        waitStateR2_4 <= 1'b1;
```

```
                    end

                STORE_NOTE_R2_5:            begin
                                                      storeEnR2 <= 1'b1;
                    end

                                        WAIT_STATE_R2_5:            begin
                                                      waitStateR2_5 <= 1'b1;

                    end

                                      STORE_NOTE_R3_1:            begin
                                                      storeEnR3 <= 1'b1;

                    end

                WAIT_STATE_R3_1:            begin
                                                      waitStateR3_1 <= 1'b1;
                                            end

                STORE_NOTE_R3_2:            begin
                                                      storeEnR3 <= 1'b1;
                                            end

                WAIT_STATE_R3_2:            begin

                                                          waitStateR3_2
<= 1'b1;

                    end

                STORE_NOTE_R3_3:            begin

                                                          storeEnR3 <=
1'b1;

                              end

                          WAIT_STATE_R3_3:              begin
                                                                                  waitStateR3_3
<= 1'b1;
                                                  end

                          STORE_NOTE_R3_4:              begin
                                                                          storeEnR3 <= 1'b1;

                           end

                          WAIT_STATE_R3_4:              begin
                                                                          waitStateR3_4 <= 1'b1;

                              end

                          STORE_NOTE_R3_5:              begin
                                                                          storeEnR3 <= 1'b1;
                              end
                                                        WAIT_STATE_R3_5:            begin
                                                                          waitStateR3_5 <= 1'b1;

                              end

                          RESET_RAMSLOT:                begin
```

```
                                                                            reset_ramSlotR1 <=
1'b1;

reset_ramSlotR2 <= 1'b1;

reset_ramSlotR3 <= 1'b1;
                                                      end

                                                        WAIT_FINAL:               begin

                                                                                   waitFinal <=
1'b0;

                                   end



                             PLAY_BACK:                begin
                                                                            playEn <= 1'b1;

                                   end




                    endcase
            end

            // Changes the state

            always @(posedge clk)
            begin: state_FFs
                    if(!reset)
                            current_state <= next_state;
                    else
                            current_state <= RESET;
            end

            // Internal RAM, 64 words by 8 bits
            ram64x8 R1(
                    .data(ramDataR1),
                    .wren (wren),
                    .address(ramSlotR1),
                    .clock(clk),
                    .q(ramOutputR1));
```

```
      ram64x8 R2(
.data(ramDataR2),
.wren (wren),
.address(ramSlotR2),
.clock(clk),
.q(ramOutputR2));

 ram64x8 R3(
.data(ramDataR3),
.wren (wren),
.address(ramSlotR3),
.clock(clk),
.q(ramOutputR3));


	wire newclk2;
	reg reset_ramSlotR1, reset_ramSlotR2, reset_ramSlotR3;
	reg storeEnR1, storeEnR2, storeEnR3;
	reg playEn;
	reg waitStateR1_0, waitStateR1_1, waitStateR1_2, waitStateR1_3, waitStateR1_4,
waitStateR1_5, waitStateR1_6;
	reg waitStateR2_1, waitStateR2_2, waitStateR2_3, waitStateR2_4, waitStateR2_5;
	reg waitStateR3_1, waitStateR3_2, waitStateR3_3, waitStateR3_4, waitStateR3_5,
waitFinal, xander;
	reg speed2;
	reg speedEn;


//**************************************
//*                                    *
//*             DATAPATH               *
//*                                    *
//**************************************
	 always @(posedge clk)

		begin

						//add state that resets FSM counter


		 // Chooses the speed of the notes based on switch inputs
		 if (speedEn)

		begin


			if (SW[9:8] == 2'b10)

				begin
					speed2 <= 1'b1;
					//speed1 <= 1'b0;
				end
		end
```

```
                                if (reset_ramSlotR1)

                                        begin

                                                ramSlotR1 <= 7'd1;
                                                playNoteR1 <= 8'b00000000;

                                        end
                                                                  if (reset_ramSlotR2)

                                        begin

                                                ramSlotR2 <= 7'd1;
                                                playNoteR2 <= 8'b00000000;

                                        end

                                                                                  if
(reset_ramSlotR3)

                                        begin

                                                ramSlotR3 <= 7'd1;
                                                playNoteR3 <= 8'b00000000;

                                        end


                               if (waitStateR1_0)

                         begin

                                 ramSlotR1 <=5'd1;

                         end


                   if (waitStateR1_1)

                         begin

                                  ramSlotR1 <= 5'd1;

                         end

                if (waitStateR1_2)

                         begin

                                  ramSlotR1 <= 5'd2;
```

```
     end


if (waitStateR1_3)

    begin
            ramSlotR1 <= 5'd3;

        end


if (waitStateR1_4)

                             begin

             ramSlotR1 <= 5'd4;

         end


if (waitStateR1_5)

        begin
                ramSlotR1 <= 5'd5;
        end


if (waitStateR1_6)

        begin
                ramSlotR2 <= 5'd1;
        end


if (waitStateR2_1)

        begin
                ramSlotR2 <= 5'd2;
        end

         if (waitStateR2_2)

        begin
                ramSlotR2 <= 5'd3;
        end

               if (waitStateR2_3)

        begin
                ramSlotR2 <= 5'd4;
        end

               if (waitStateR2_4)
        begin
                ramSlotR2 <= 5'd5;
        end

               if (waitStateR2_5)

        begin
                ramSlotR3 <= 5'd1;
        end

               if (waitStateR3_1)

        begin
                ramSlotR3 <= 5'd2;
        end

         if (waitStateR3_2)

        begin
                ramSlotR3 <= 5'd3;
        end

               if (waitStateR3_3)

        begin
                ramSlotR3 <= 5'd4;
        end

               if (waitStateR3_4)

        begin
                ramSlotR3 <= 5'd5;
        end

               if (waitStateR3_5) //MIGHT BE A PROBLEM PLS CHECK

        begin
                wren <= 1'b0;
        end



if (storeEnR1) begin

          wren <= 1'b1;

        begin


                if(~KEY[2])
                        begin

                                ramDataR1 <= SW[7:0];
```

```
                    end

            end
                            end

                   if (storeEnR2) begin

             wren <= 1'b1;

            begin

                    if(~KEY[2])
                            begin

                                    ramDataR2 <= SW[7:0];

                            end

            end
                            end

                  if (storeEnR3) begin

             wren <= 1'b1;

            begin

                    if(~KEY[2])
                            begin

                                    ramDataR3 <= SW[7:0];

                            end

            end
                            end

    // Play the notes back by reading them out of RAM at the selected speed

            //end

    if (playEn)

            begin

                    wren <= 1'b0;
```

```
                                               if (speed2)
                                                     begin


              if(SW[7] == 1'b1)

              begin

              wren <= 1'b0;


                                                      playNoteR1 <= ramOutputR1;

                                                             if (newclk2 && ramSlotR1 != 5'd5)
                                                                     ramSlotR1 <= ramSlotR1 + 1'd1;

                                    else if (newclk2 && ramSlotR1 == 5'd5)




                                                  ramSlotR1 <= 5'd1;
                                                      end

              // end


                                             if (SW[8] == 1'b1)

                                                      begin

              wren <= 1'b0;


                                                      playNoteR2 <= ramOutputR2;

                                                             if (newclk2 && ramSlotR2 != 5'd5)
                                                                     ramSlotR2 <= ramSlotR2 + 1'd1;

                                    else if (newclk2 && ramSlotR2 == 5'd5)




                                                  ramSlotR2 <= 5'd1;
                                                      end
                                                                                       if (SW[9] == 1'b1)

                                                      begin

              wren <= 1'b0;
```

```
                                        playNoteR3 <= ramOutputR3;

                                               if (newclk2 && ramSlotR3 != 5'd5)
                                                       ramSlotR3 <= ramSlotR3 + 1'd1;

                      else if (newclk2 && ramSlotR3 == 5'd5)




                                    ramSlotR3 <= 5'd1;
                                        end


                emd
```

```
                                    end



        end // End of Datapath


assign LEDR [9:0] = current_state;
//assign LEDR [9:4] = waitState9;

/*music xander1( .clk(clk),
.speaker(speaker));*/

newClockCounter2 C1(
.key3                (~KEY[3]),
.CLOCK_50        (clk),
.newclk2         (newclk2)
);

hex_decoder H0(
        .hex_digit(ramSlotR1),
        .segments(HEX0)
        );

hex_decoder H1(
        .hex_digit(ramSlotR2),
        .segments(HEX1)
        );

hex_decoder H2(
        .hex_digit(ramSlotR3),
        .segments(HEX2)
        );



//**************************************
//*                                    *
//*                VGA                    *
//*                                       *
//***************************************

       vga_adapter VGA(
                      .resetn(resetn),
                      .clock(CLOCK_50),
                      .colour(memoryout),
                      .x(x_plot),
                      .y(y_plot),
                      .plot(plot),
                      /* Signals for the DAC to drive the monitor. */
                      .VGA_R(VGA_R),
                      .VGA_G(VGA_G),
                      .VGA_B(VGA_B),
                      .VGA_HS(VGA_HS),
                      .VGA_VS(VGA_VS),
                      .VGA_BLANK(VGA_BLANK_N),
                      .VGA_SYNC(VGA_SYNC_N),
                      .VGA_CLK(VGA_CLK));
              defparam VGA.RESOLUTION = "160x120";
              defparam VGA.MONOCHROME = "FALSE";
              defparam VGA.BITS_PER_COLOUR_CHANNEL = 3;
              defparam VGA.BACKGROUND_IMAGE = "djbackground.mif";


       localparam WAIT = 1'b0,
                  NOTE_DISPLAY = 1'b1;

       //vga display has a resolution of 320x240
       parameter maxX = 8'd159,
                             maxY = 7'd119;


       //x and y counter
       always @( posedge CLOCK_50) begin
                      if(current_state2 == WAIT)begin
                             xcount <= 8'b0;
                             ycount <= 7'b0;
                             end
                      else if ((current_state2 == NOTE_DISPLAY) && (xcount<= maxX) )
                             xcount <= xcount +1'b1;
                      else if (current_state2 == NOTE_DISPLAY && (xcount > maxX)) begin
                             xcount <= 8'b0;
                             ycount <= ycount + 7'b1;

                             if (ycount > maxY) begin
                                           current_state2 = WAIT;
                             end

                      end

                      current_state2 <= next_state2;
```

```
        end


        //register to deal with delay
        always@(posedge CLOCK_50) begin
                x_plot <= xcount;
                y_plot <= ycount;
        end


        //output the address
        assign address = xcount + (ycount*160);


        //state table
        always@(*)
        begin: state_table2
                case(current_state2)
                        WAIT: next_state2 = (~KEY[3]) ? NOTE_DISPLAY : WAIT; //stay in wait
state until note is selected
                        NOTE_DISPLAY: next_state2 = (address == 15'd19199) ? WAIT :
NOTE_DISPLAY; //keep plotting screen until counting is done
                        default: next_state2 = WAIT;
                endcase
        end

        ramC m1(
        .address(address),
        .clock(CLOCK_50),
        .data(9'b000000000),
        .wren(1'b0),
        .q(memoryC));

        ramB m2 (
        .address(address),
        .clock(CLOCK_50),
        .data(9'b000000000),
        .wren(1'b0),
        .q(memoryB));

        ramD m3 (
        .address(address),
        .clock(CLOCK_50),
        .data(9'b000000000),
        .wren(1'b0),
        .q(memoryD));

        ramA m4 (
        .address(address),
        .clock(CLOCK_50),
        .data(9'b000000000),
        .wren(1'b0),
        .q(memoryA));
```

```
ramE m5 (
.address(address),
.clock(CLOCK_50),
.data(9'b000000000),
.wren(1'b0),
.q(memoryE));

ramF m6 (
.address(address),
.clock(CLOCK_50),
.data(9'b000000000),
.wren(1'b0),
.q(memoryF));

ramG m7 (
.address(address),
.clock(CLOCK_50),
.data(9'b000000000),
.wren(1'b0),
.q(memoryG));


rambackground m8 (
.address(address),
.clock(CLOCK_50),
.data(9'b000000000),
.wren(1'b0),
.q(background));


//wires to transfer colour data from memory (notes)
wire [8:0] memoryC;
wire [8:0] memoryD;
wire [8:0] memoryE;
wire [8:0] memoryF;
wire [8:0] memoryG;
wire [8:0] memoryA;
wire [8:0] memoryB;

//wire to transfer colour data about background
wire [8:0] background;


always@(*)
       begin
              case(SW[7:0])
                     8'b00000001: memoryout = memoryC;
                     8'b00000010: memoryout = memoryD;
                     8'b00000100: memoryout = memoryE;
                     8'b00001000: memoryout = memoryF;
                     8'b00010000: memoryout = memoryG;
                     8'b00100000: memoryout = memoryA;
                     8'b01000000: memoryout = memoryB;
```

```
                              8'b10000000: memoryout = memoryC;
                              default: memoryout = background;
                         endcase
                    end



endmodule // End of top level module


//***************************************
//*                                     *
//*          NEWCLOCK COUNTER2          *
//*                                     *
//***************************************
module newClockCounter2 (CLOCK_50, newclk2, key3); //top module
input key3;
input CLOCK_50; //clock 50Mhz
output reg newclk2; //new speed
reg [26:0] counter; //counter to 50 000 000 for 1 second rate
always @(posedge CLOCK_50) // triggered every time clock rises
        begin
                //if the clock gets full
                if(counter == 27'd50000000 || key3)

                        begin
                                newclk2 <= 1'b1;
                                counter <= 27'd0;
                        end

         else //just increment the counter

                        begin
                                newclk2 <= 1'b0;
                                counter <= counter + 1'b1;
                        end
        end

endmodule


//***************************************
//*                                     *
//*         HEX DECODER MODULE          *
//*                                     *
//***************************************
module hex_decoder(hex_digit, segments);

    input [3:0] hex_digit;

    output reg [6:0] segments;
    always @(*)

        case (hex_digit)

            4'h0: segments = 7'b100_0000;
            4'h1: segments = 7'b111_1001;
            4'h2: segments = 7'b010_0100;
            4'h3: segments = 7'b011_0000;
            4'h4: segments = 7'b001_1001;
            4'h5: segments = 7'b001_0010;
            4'h6: segments = 7'b000_0010;
            4'h7: segments = 7'b111_1000;
            4'h8: segments = 7'b000_0000;
            4'h9: segments = 7'b001_1000;
            4'hA: segments = 7'b000_1000;
            4'hB: segments = 7'b000_0011;
            4'hC: segments = 7'b100_0110;
            4'hD: segments = 7'b010_0001;
            4'hE: segments = 7'b000_0110;
            4'hF: segments = 7'b000_1110;

            default: segments = 7'h7f;

        endcase


endmodule
```