EE/CSE 371 Final Project

Adam Friesz (2263330) & Tawsif Ahmed (2223861)
EE/CSE371
Jun 5, 2024
Final Lab Report

Design Procedure

The following lab gave the students to implement and create anything over the DE1_SoC that utilizes three key functions: A VGA display, some form of significant memory storage, and some form of user input. With the required steps, we implement a music editor program with a graphical display similar to a music score and enable audio playback. The user has the ability to switch between drafting and editing modes, which allows the user to quickly transcribe and hear their original creations.

We decided to represent a range of notes, corresponding to sine wave frequencies, going from a low C in the 2nd octave (C2) to a D sharp in the 7th octave (D#7). This gave us a range of 64 notes which we encoded using unsigned numbers 0-63, which conveniently fits into 6 bits of information. This range was chosen as C2 corresponds to the lowest note on the cello and D#7 corresponds to the highest note regularly played on the violin.

User Input

The user input or user_io module is responsible for allowing the user to compose and edit music sheets by switching between drafting, editing, and playing with the SW[1:0]. In drafting mode, the KEYs are used to select between note tones as well as its octave. The current note is always visually displayed on the HEX to let the user know before confirming. Once a note is selected, the user can confirm to display the note on the VGA and save it in their composition. The saving mechanism is accomplished through the use of a RAM block called note_RAM and a pointer called i_note that corresponds to the number of saved notes within the RAM. Inside the editing mode, the user has to option to reset, move forward/backward through the composition, and delete notes. Using editing and drafting modes in tandem, the user has the ability to modify notes in the middle of their composition - simplifying the creation process. The user can select to play the current composition of music, whenever they want by turning on SW[1].

.

State	Button Configuration	Action
ANY	SW[1] = 1	Play music
Draft	1000	Note up
Draft	0010	Note down
Draft	0001	Octave up
Draft	0100	Confirm/Place
Edit	1000	Move forward
Edit	0010	Move backward
Edit	0001	Reset
Edit	0100	Delete

Button configuration is ordered: {KEY0, KEY1, KEY2, KEY3}
State = SW[0]? Edit: Draft;

Figure 1: The chart for the User input module

Figure 1 shows the chart used to create the module and will begin in the drafting mode with the next state being based on SW[1:0]. Inside the drafting state, the user can press KEY[0] to cycle through the octave from 2 to 7. Pressing KEY[1] and KEY[3] allows the user to increase or decrease the note respectively. The user can confirm the note and its octave by turning KEY[2], which would store it in the note_RAM. Signals from the keys are filtered through an additional module called io_pulse_gen before being sent to the user_io module. The filtering module accepts the key signals as inputs, filters them for metastability, ignores short oscillations produced by the physical mechanism of the keys, and outputs a pulse that is sent to our user_io module. A code snippet of the module is shown below in **Figure 2.**

The user is allowed to be in the editing state by turning on SW[0]. Inside the module, the user is given the same amount of options as in the drafting state. KEY[0] allows the user to completely reset the composition, which would also erase the notes stored in the note_RAM. The user can press KEY[1] and KEY[3] to move between notes that have been composed to edit the note or delete the note by pressing KEY[2], which would erase the note from the RAM as well. The editing can be down by going back to the draft mode to select a new note and confirming, which would replace the current note with a new note. Once the user turns on SW[1], the module goes into the play state where the user input is disabled until the switch is turned off. Inside the play state, the module gives the enable signal to the top module to allow the audio to be played.

```
`define LEN 1
module io_pulse_gen (clk, buttons, button_pulse);
   input logic clk;
   input logic [3:0] buttons;
   output logic [3:0] button_pulse;

   logic [3:0] b1, b2;
   // filter for metastability
   always_ff @(posedge clk) begin
      b1 <= buttons;
      b2 <= b1;
   end //always_ff

   logic [12:0] counter0, counter1, counter2, counter3;
   logic wasOn0, wasOn1, wasOn2, wasOn3;

   always_ff@(posedge clk) begin
      case(b2[0])
         0: begin counter0 <= 0; wasOn0 <= 0; end
         1: counter0 <= counter0 + 1;
      endcase

      case(b2[1])
         0: begin counter1 <= 0; wasOn1 <= 0; end
         1: counter1 <= counter1 + 1;
      endcase

      case(b2[2])
         0: begin counter2 <= 0; wasOn2 <= 0; end
         1: counter2 <= counter2 + 1;
      endcase

      case(b2[3])
         0: begin counter3 <= 0; wasOn3 <= 0; end
         1: counter3 <= counter3 + 1;
      endcase

      if (counter0 ==`LEN) begin
         wasOn0 <= 1;
      end
      if (counter1 ==`LEN) begin
         wasOn1 <= 1;
      end
      if (counter2 ==`LEN) begin
         wasOn2 <= 1;
      end
      if (counter3 ==`LEN) begin
         wasOn3 <= 1;
      end
   end // always_ff

   assign button_pulse = {~wasOn3 & counter3==`LEN, ~wasOn2 & counter2==`LEN, ~wasOn1 & counter1==`LEN, ~wasOn0 & counter0==`LEN};
endmodule // pulse_gen
```

Figure 2: The code snippet of IO_pulse_gen module for the User input module

Graphical Interface

The node_decoder is our graphical interface module that is responsible for managing the display of the musical notes on colored VGA given to us. It takes in status signals from the user input module to place, delete, and replace notes visually to generate a colorful version of sheet music. The display of the musical notes is updated in real-time.

The modules use a simple FSM-based **Figure 3** to do operations in order to indicate what to display on the screen. The module has two helper modules to operate, graph module and note_color. The graph module creates the graphical outline with an x-axis and y-axis like in **Figure 4.**

Figure 3: Visual of the graph helper module

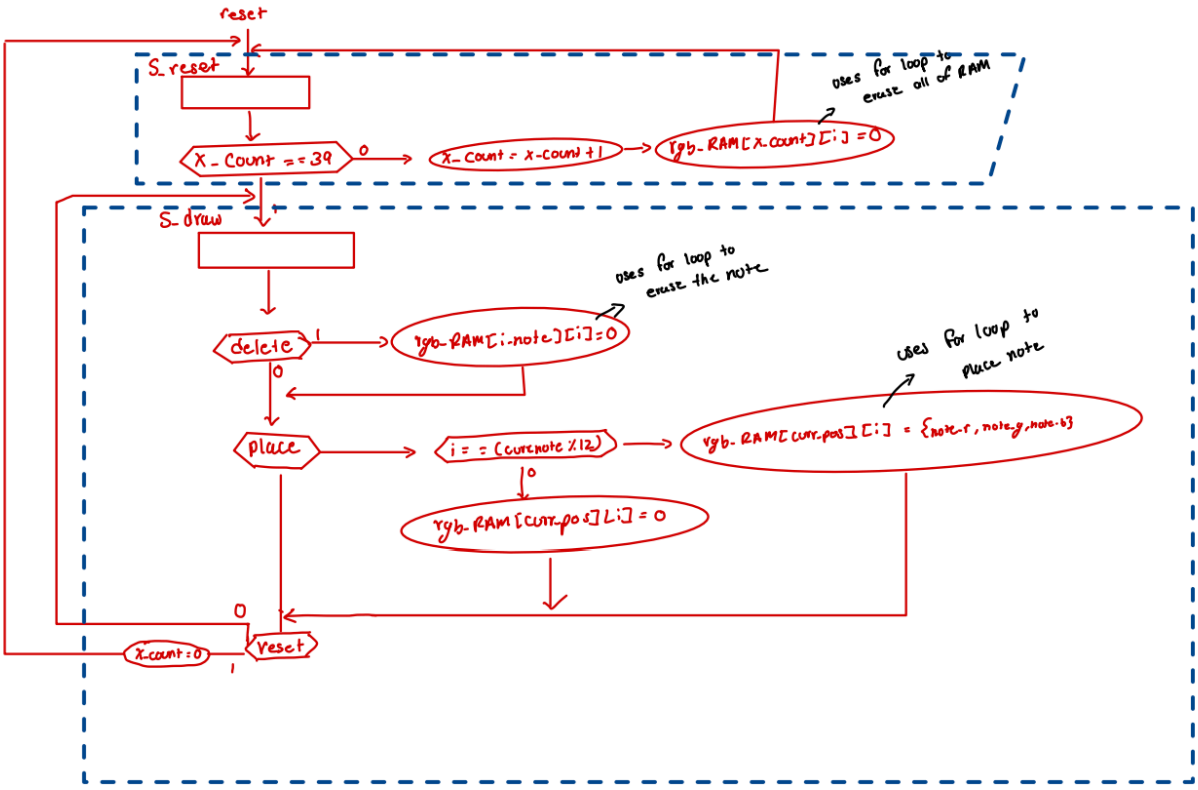

Figure 4: The ASM chart for the note_decoder (Graphical Interface) module

The y-axis determines the note while the x-axis represents the time. Since the colored-VGA iterates through x and y constantly every clock cycle, the labels and axes are hardcoded by giving the corresponding color to the notes based on the ranges of x and y. **Figure 5** shows the code snippet of the hard-coding label G# inside the graph module.

```
// G#
else if ( (x > 29 && x < 54) && ( y > 309 && y < 345)) begin
   // Left part
   if ( (x > 29 && x < 32) && ( y > 320 && y < 334)) begin
      r = 0; g = 89; b = 233;
   end

   // Top part
   else if ( (x > 31 && x < 38) && ( y > 316 && y < 320)) begin
      r = 0; g = 89; b = 233;
   end

   // Bottom part
   else if ( (x > 31 && x < 38) && ( y > 334 && y < 338)) begin
      r = 0; g = 89; b = 233;
   end

   // Right part
   else if ( (x > 36 && x < 39) && ( y > 329 && y < 334)) begin
      r = 0; g = 89; b = 233;
   end

   else if ( (x > 33 && x < 39) && ( y > 327 && y < 330)) begin
      r = 0; g = 89; b = 233;
   end

   // #
   else if ( (x > 40 && x < 51) && ( y > 320 && y < 324)) begin
      r = 0; g = 89; b = 233;
   end

   else if ( (x > 40 && x < 51) && ( y > 330 && y < 334)) begin
      r = 0; g = 89; b = 233;
   end

   else if ( (x > 42 && x < 45) && ( y > 316 && y < 338)) begin
      r = 0; g = 89; b = 233;
   end

   else if ( (x > 46 && x < 49) && ( y > 316 && y < 338)) begin
      r = 0; g = 89; b = 233;
   end

   else begin
      r = 0; g = 0; b = 0;
   end
end
```

Figure 5: The code snippet of a label from graph module

The note_color module takes in a note that is 6-bit to represent the 0th to 63th note. C2 being 0th to D#7 being the 63th note. Based on the note and octave, it represents the color and the shade of the color. The lower octave is indicated by a darker shade of the note color, while the higher octave is represented by a lighter shade. **Figure 6** shows the code snippet of the module.

```
module note_color (note, r, g, b);
   input logic [5:0] note;
   output logic [7:0] r, g, b;

   // Chooses a color based on the notes and the octave
   always_comb begin
      case (note)
         // Octave 2
         0: begin r = 66; g = 200; b = 0; end
         1: begin r = 195; g = 199; b = 0;end
         2: begin r = 195; g = 73; b = 0; end
         3: begin r = 144; g = 0; b = 0; end
         4: begin r = 210; g = 0; b = 65; end
         5: begin r = 211; g = 0; b = 202;end
         6: begin r = 89; g = 0; b = 207; end
         7: begin r = 0; g = 0; b = 172; end
         8: begin r = 0; g = 66; b = 210; end
         9: begin r = 0; g = 203; b = 211;end
         10: begin r = 0; g = 206; b = 86;end
         11: begin r = 0; g = 166; b = 0; end

         // Octave 3
         12: begin r = 78; g = 211; b = 0; end
         13: begin r = 207; g = 210; b = 0;end
         14: begin r = 207; g = 84; b = 0;end
         15: begin r = 167; g = 0; b = 0; end
         16: begin r = 221; g = 0; b = 77;end
         17: begin r = 222; g = 0; b = 214;end
         18: begin r = 100; g = 0; b = 219;end
         19: begin r = 0; g = 0; b = 195; end
         20: begin r = 0; g = 78; b = 221;end
         21: begin r = 0; g = 215; b = 222;end
         22: begin r = 0; g = 218; b = 97; end
         23: begin r = 0; g = 190; b = 0; end

         // Octave 4
         24: begin r = 90; g = 222; b = 0; end
         25: begin r = 219; g = 221; b = 0; end
         26: begin r = 219; g = 95; b = 0; end
         27: begin r = 188; g = 0; b = 0; end
         28: begin r = 233; g = 0; b = 88; end
         29: begin r = 233; g = 0; b = 226; end
         30: begin r = 111; g = 0; b = 231; end
         31: begin r = 0; g = 0; b = 216; end
         32: begin r = 0; g = 89; b = 233; end
         33: begin r = 0; g = 227; b = 233; end
         34: begin r = 0; g = 230; b = 108; end
         35: begin r = 0; g = 211; b = 0; end

         // Octave 5
         36: begin r = 102; g = 233; b = 0; end
         37: begin r = 231; g = 232; b = 0; end
```

Figure 6: The code snippet of note_color module

A large block of 2 dimensional RAM is used in our graphical display to store color values for each block of pixels. We initially had a 600x400 block of memory that stored the RGB values at each pixel but quickly realized this would overload the machine storage and would not be the most efficient way to store the information. We conceived of dividing the display area into blocks that represent individual half second notes, the smallest feature we display. By utilizing this “block” approach we reduced the amount of memory we needed by a factor of 500.

The note_decoder module puts together the graph and note_color helper modules to accurately display the notes on the VGA and edit the rgb_RAM. The module uses two states to determine whether to draw or reset the VGA. The drawing state uses combinational logic, which looks at the x & y to determine the color of the VGA by reading from the color RAM or taking RGB from the graph module. If x & y are within a box given by us, the module outputs RGB from the RAM.

While in the draw state, if the user_io module asserts the confirm signal, then it will start writing RGB to color RAM addresses based on the note indicated and the position of the x & y. If the delete signal is asserted, then the module will replace the address values based on the x, to be 0. These can be seen in the code snippet of the module from **Figure 7.** Reset is given a separate state to minimize the synthesized circuity by accomplishing the command sequentially. A counter increments through each column in the playing area, corresponding to the 1st dimension in our rgb_RAM, and the column is reset to black.

```
always_ff @(posedge clk) begin
    if (delete) begin
        for (i = 0; i < 12; i = i+1)
            rgb_RAM[i_note-1][i] <= 0;
    end

    if (place) begin
        for (i = 0; i < 12; i = i+1)
            rgb_RAM[curr_pos][i] <= (i==(curr_note%12))? {note_r,note_g,note_b}: 0;
    end
```

Figure 7: The code snippet of the note_decoder module

Note Playback

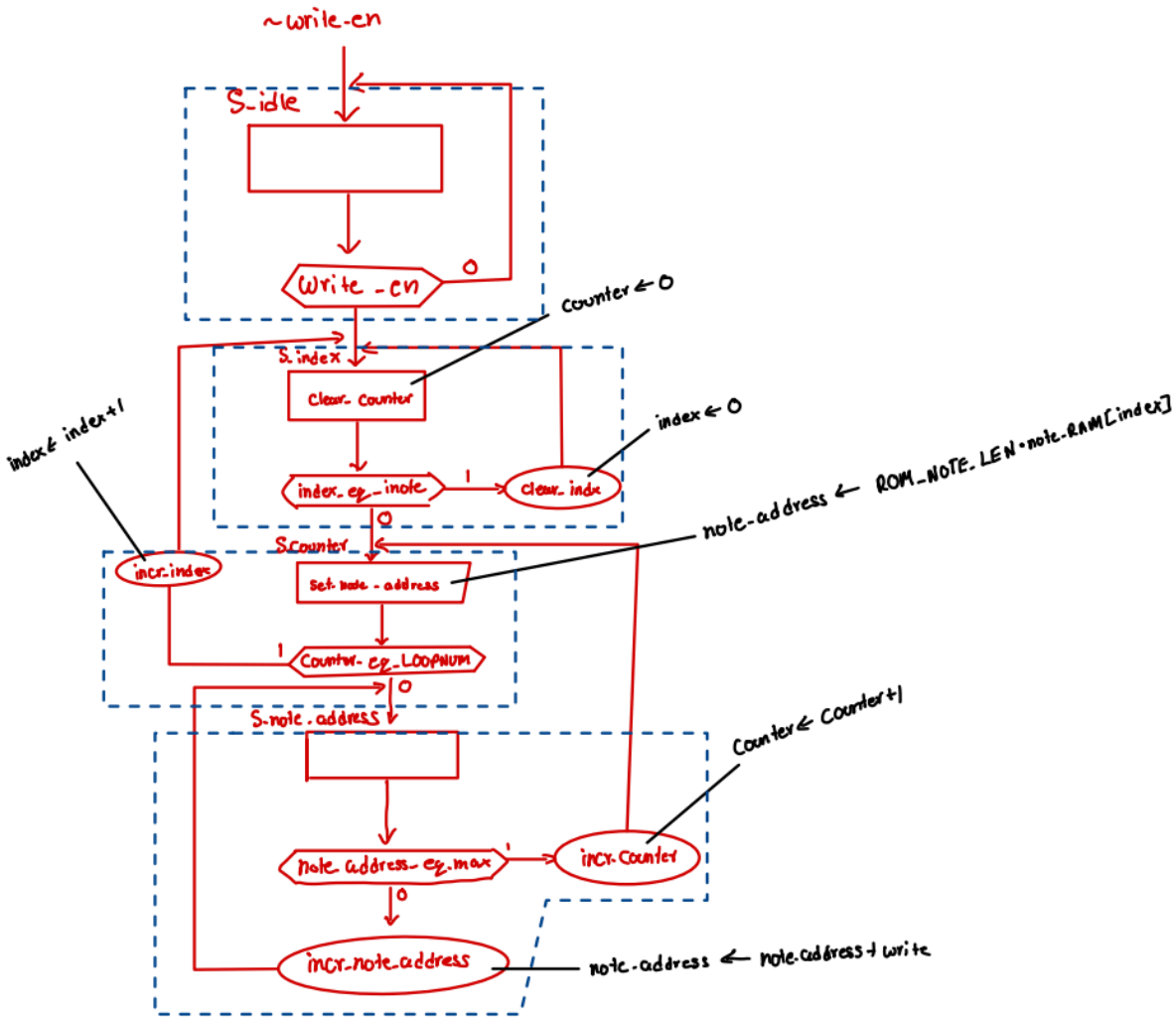

Figure 8: The ASMD chart for the Note Playback (decoder_audio_playback) module

The note playback or the decoder_audio_playback module handles the playback of the musical composition by converting the note data stored in the note_RAM to audio amplitude data that is sent to the top module. Audio amplitude data is stored in the combined mif file, which is read by the combined_notes ROM. We modified the note generation Python script from lab 3 to combine the mif data for all 64 notes into the same file. The generated mif file combined 375 entries from each note to save 24000 total entries with C2 starting at address 0 and C#2 starting at 375.

The module uses the ASMD Chart from **Figure 8** to constantly loop to play the composition until the SW[1] is off. Instead of creating two separate modules to build the controller and datapath, we implemented both of them in one singular module. The module stays in the idle state until the user asserts the write-enable signal. Once it is asserted, the module moves to the index state. In the index state, it checks whether the index equals i_note. If index_eq_inote is asserted, the controller clears the index otherwise it clears the counter.

After index_eq_inote is de-asserted, the module moves to the counter state, where it checks to see if counter_eq_LOOPNUM is true or not. If the status signal is true, it reverts back to the index state and increments the index. If not, it will set the note address based on the current index and note in note_RAM and move to the address state. Insider the address state, the module will check if note_address_eq_max is asserted or not. If note_address_eq_max is true inside the state, then it will increment the counter and loop from the counter state to the address state until note_address_eq_max is de-asserted. Once it is, it will increase the note address and stay in the address state. Based on the note address, it will read from the combined ROM and output the data to the top module to play notes from the sheet music.

Overall System

Figure 9 shows the block diagram of our top-level module (DE1_SoC). Our system uses CLOCK_50 as the main clock for all modules. The HEXs are connected to show users corresponding notes that they are confirming/editing.

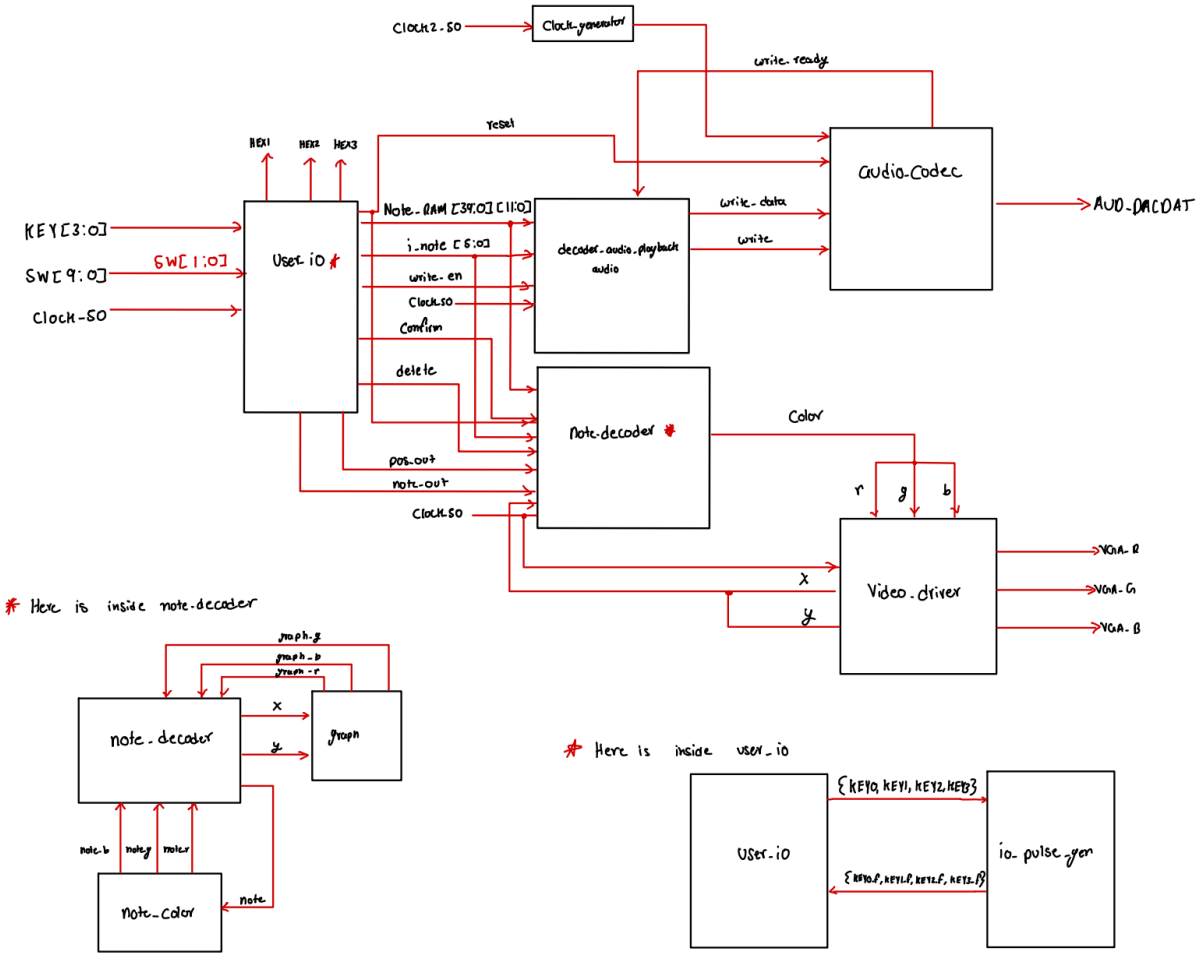

Figure 9: Block Diagram for Overall System

Results

User Input

Our final design for user input differed from our proposal because we decided to have the user specify the notes they wanted via board switches and buttons instead of playing audio to transcribe into the microphone. We decided to make this switch because we were having a lot of difficulty with implementing digital signal processing techniques to enable us to perform a Fast Fourier Transform. Allowing the user to specify notes themselves also transforms the project from music copying to music creation. We tested the user input portion of the design by creating testbenches for the io_pulse_gen module and the user_io module.

Figure 10 shows the results of simulating the io_pulse_gen module. The module correctly ignores input less than the specified LEN macro and turns long strings of high input signal. This is in our simulation as the output button_pulse[0] is only high for one clock cycle at 75ps despite the input signals buttons[0] being high for over 300ps.

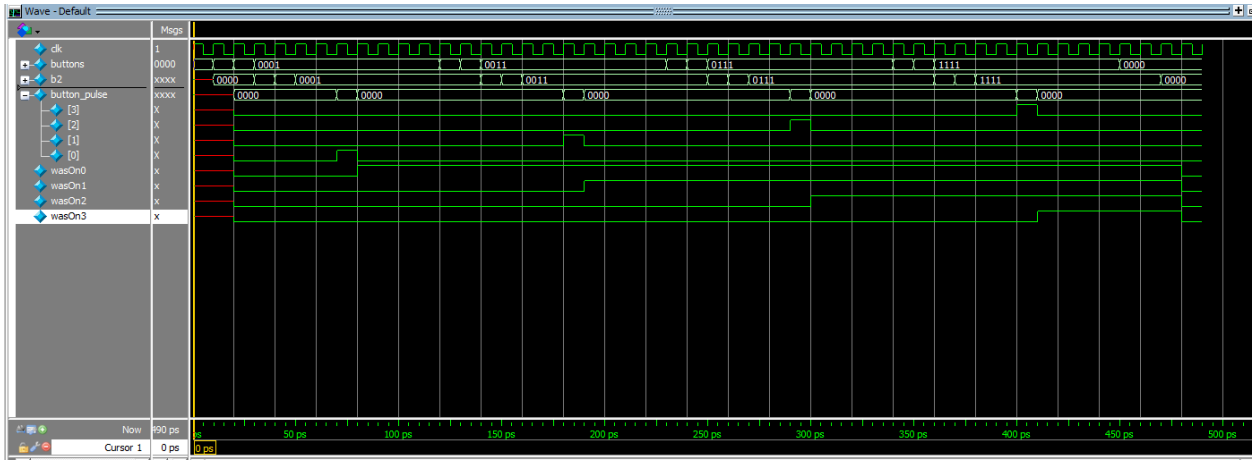

Figure 10: Simulation of io_pulse_gen module.

We tested the user_io module by comprehensively checking each behavior in the editing and drafting state. The simulation for confirm is shown below in **Figure 11**. From 186ps to 260ps we choose which note we have selected via the note up and note down signals. Then at 275ps the confirm signal is asserted and the values in note_RAM accurately change along with the output signals pos_out and note_out which are sent to the note_decoder. Correct behavior of incrementing i_note and curr_pos also occur and the additional confirm signals at 375 and 440ps verify that the curr_pos is correctly determining which address of note_RAM to write to.

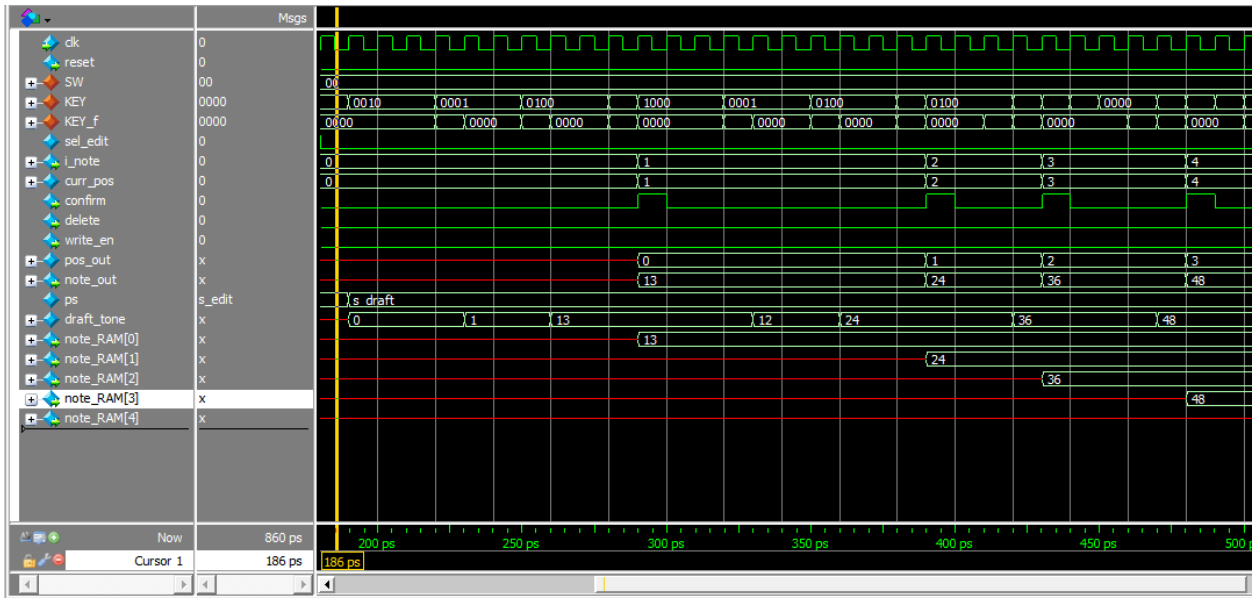

Figure 11: Simulation of user_io module showing accurate confirm behavior.

Figure 12 shows the verification of the edit behavior which is a combination of navigating forward and backward while in edit mode and changing the value at the current position while in drafting mode. At 680ps we decrement curr_pos from 5 to 4 with i_note correctly staying at 5. The confirm signal is then asserted at 720ps and the value stored in note_RAM[4] is correctly changed from its previous value to the current value of draft_tone.

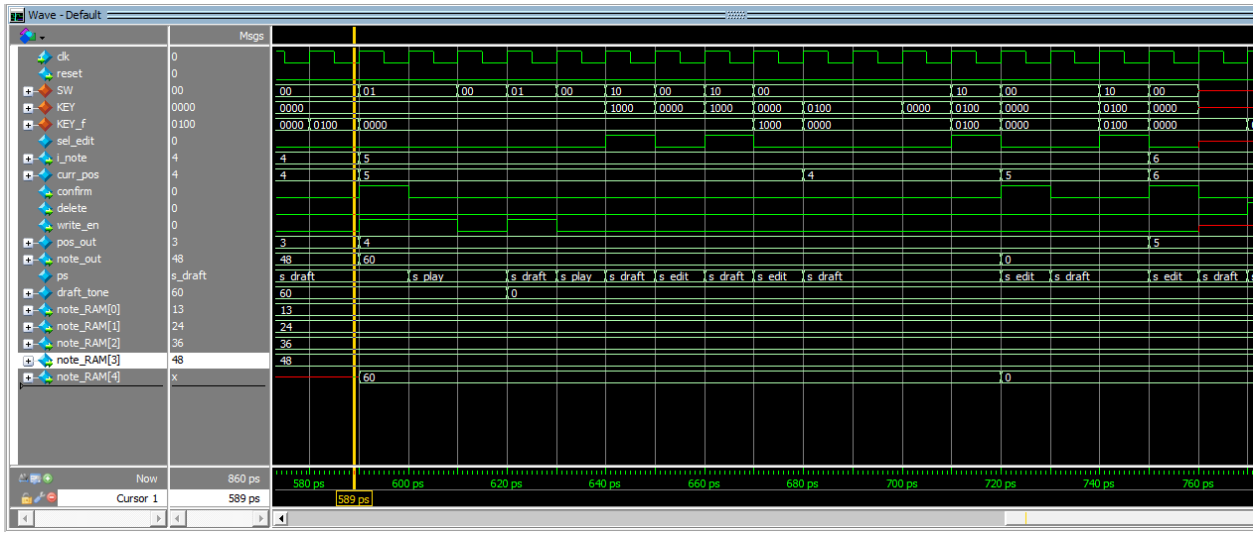

Figure 12: Simulation of user_io module showing accurate edit behavior.

Figure 13 shows the overall simulation of user_io. This view allows us to verify the acceptance of consecutive signals and navigation between s_draft and s_edit. The correct order of adding to addresses is also clear as the 0th address of note_RAM is first confirmed followed by 1, 2, 3, and 4.

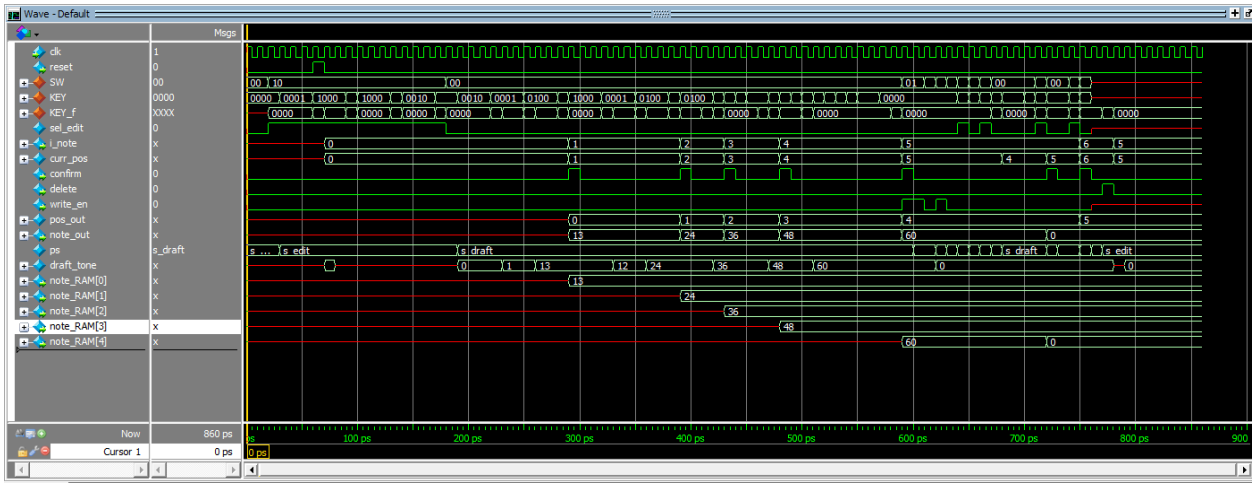

Figure 13: Overall simulation of user_io module.

Graphical Interface

Due to the long streams of data and many clock cycles needed to simulate VGA output we decided to test the graphical interface portion of our project with a mix of testbenching as well as checking visually by synthesizing onto LabsLand. To accomplish this visual testing we created testing code that allowed us to predictably change the inputs to the VGA which simulated our input from the note_decoder module.

Additionally, we thoroughly simulated our note_decoder design to verify place, delete, and reset signals were correctly responded to. This allowed us to uncover timing issues with

when these signals would be high compared to when the curr_note and curr_pos signals were valid.

To test the functionality of our helper module, which would just be the static visual of portion of our module. We tested the graph indenpentdetly, which was seen in **Figure 3**. Then, we tested our ability to change notes and the octaves with the graph helper module and the result in seen **Figure 14.**

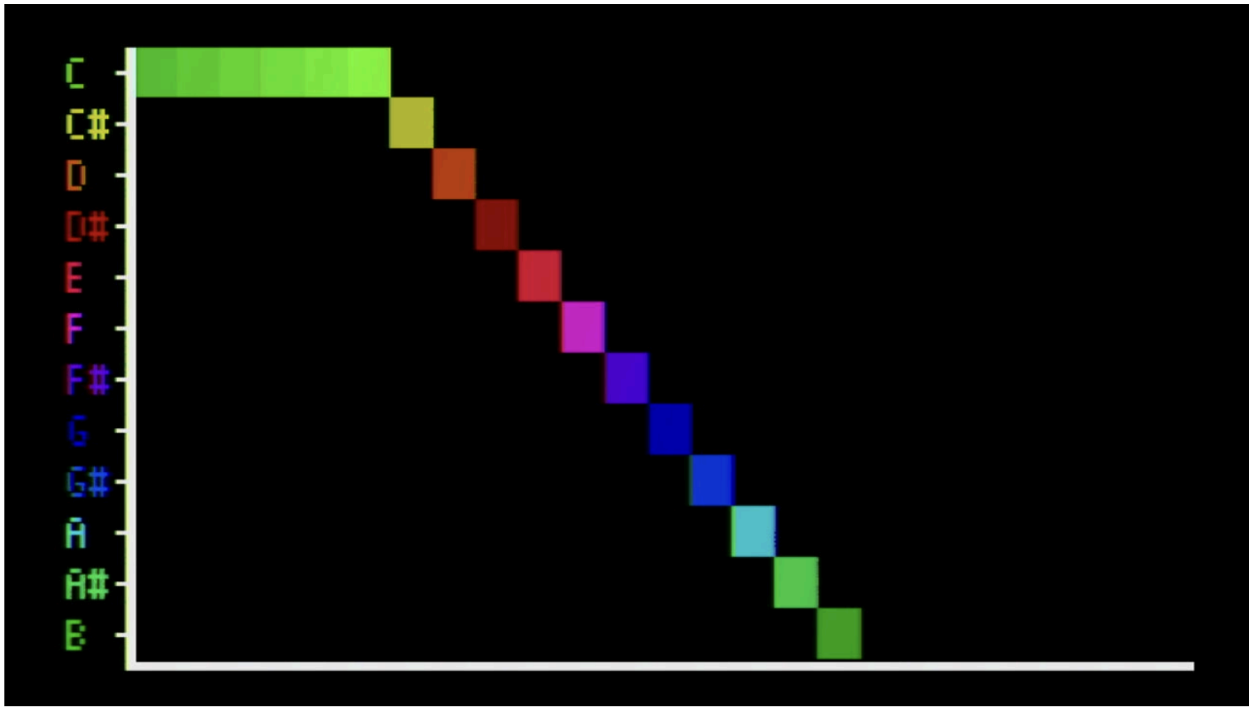

Figure 14: Visual simulation of the note_color helper module

Figure 15 shows rgb_RAM correctly saving new values in response to the place signal which comes from user_io. Each value being stored in the entry of rgb_RAM is different from the previous which is expected because each note has a distinct color value and we increment the value of curr_note between each place.

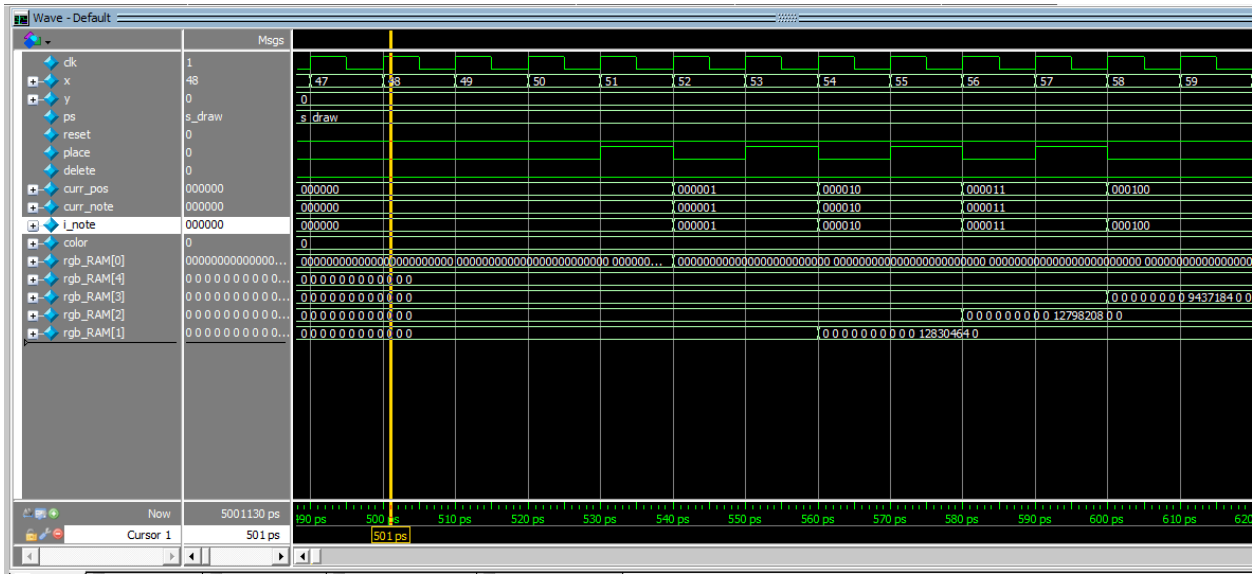

Figure 15: Simulation showing simulation of rgb_RAM correctly updating in response to place control signal.

Figure 16 shows the overall simulation of note_decoder and is primarily useful to verify that the value of color changes as we cycle between x and y values. We can also see the modified content of the rgb_RAMs. Visual verification via previous figures x and x is more useful validation for this piece of the design.

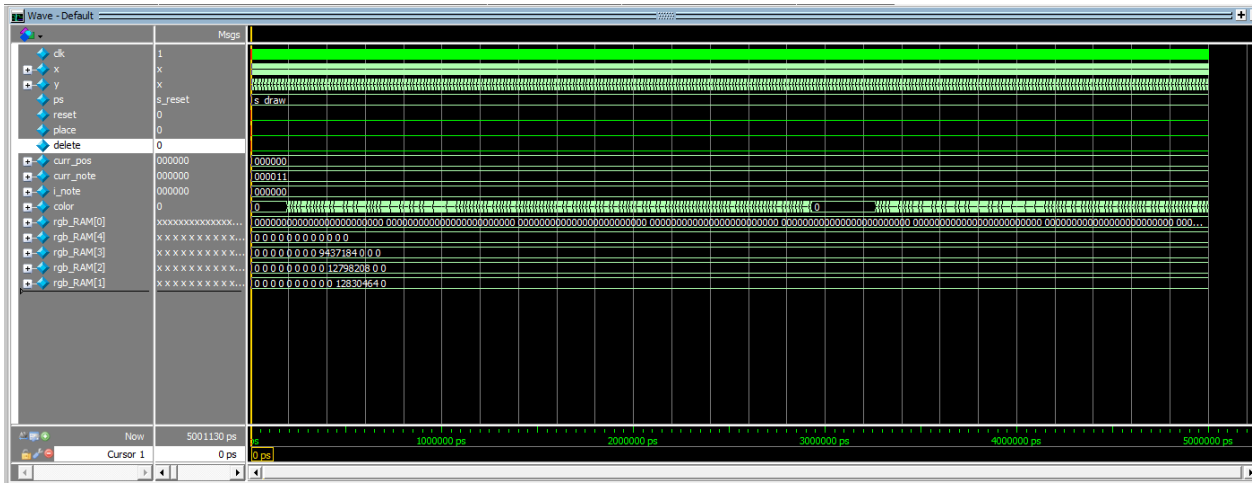

Figure 16: Full simulation of note_decoder, showing color updating in response to x, y.

Note Playback

To test our note playback module we engaged in lots of testbenching because the long synthesis time and the lack of audio capabilities on many of the LabsLand DE1s made audial testing inefficient. We simulated our design for over 100,000 clock cycles to watch it correctly cycle through the address of our ROM corresponding to each note correctly.

In **Figure 17** we can see the write signal correctly lines up with the write_ready signal as in lab 3. Additionally we can verify that note_address is correctly incrementing and resetting when its value is equal to 375 greater than its initial value at which point counter increments.

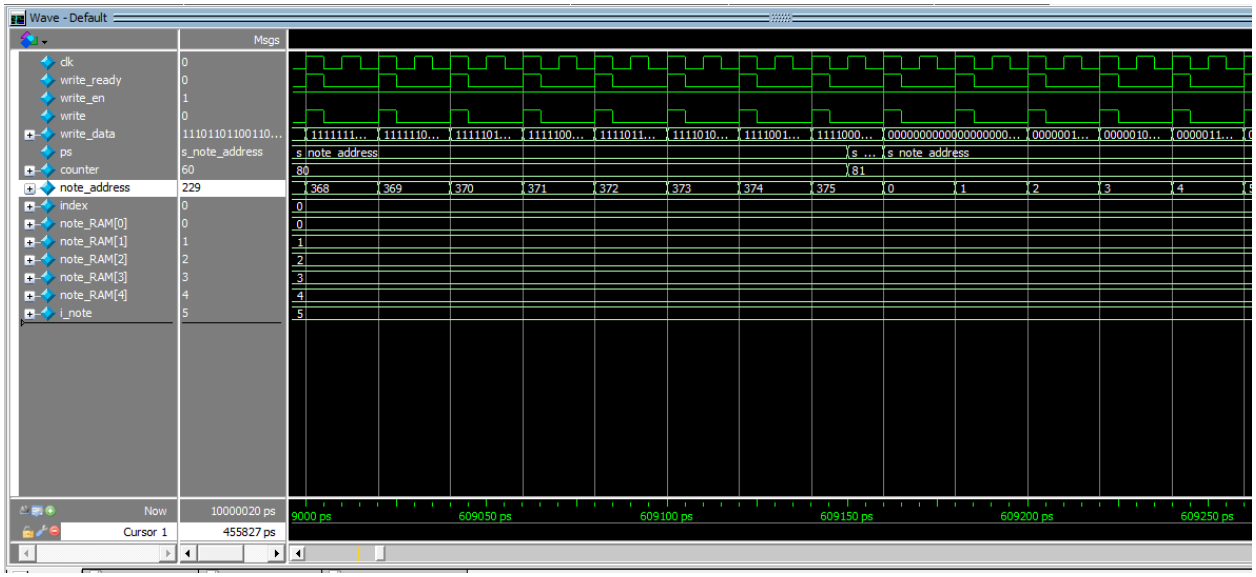

Figure 17: Simulation showing accurate incrementation of decoder_audio_playback note_address.

Figure 18 shows a more zoomed out view of the audio playback simulation. This view allows us to verify the proper incrementation of counter. When counter reaches 127 it resets to 0 and the value of index is incremented. This ensures that the correct section of audio is looped over for each period of the music stored in note_RAM.

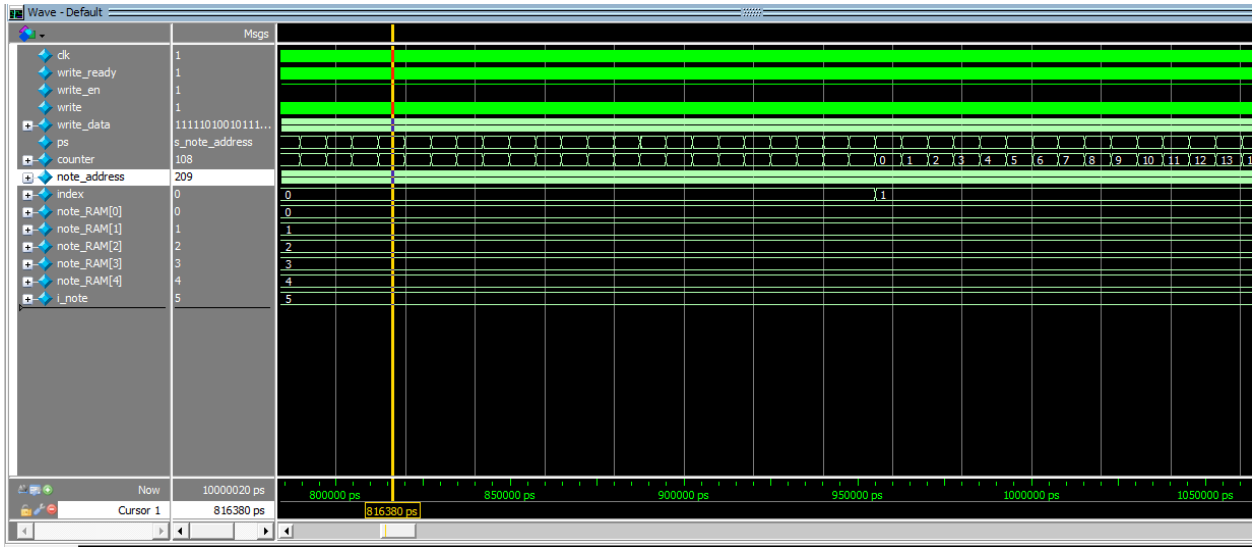

Figure 18: Simulation showing accurate incrementation of decoder_audio_playback counter.

Figure 19 shows the most zoomed out view of the audio playback module. This view allows us to verify that the audio correctly loops while write_en is still active. This is shown at around 500,000ps when index resets to 0 after reaching 4 which is equal to i_note minus 1. The initial frame of simulation at 0ps also shows us that write is disabled when write_en is deasserted.

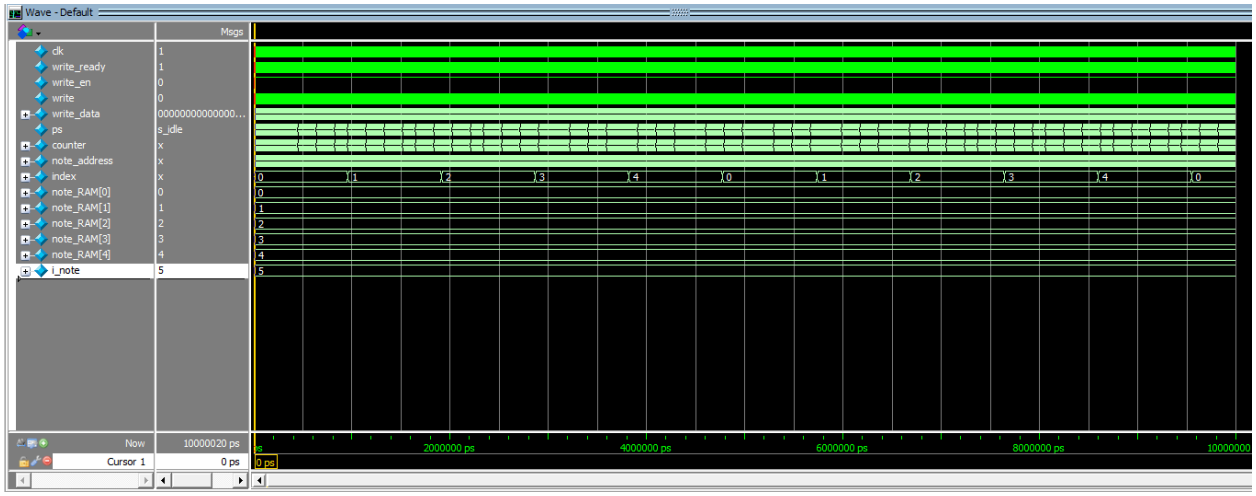

Figure 19: Full simulation of decoder_audio_playback, showing accurate incrementation of index.

DE1_SoC

Simulation of our overall design was difficult because of the large number of clock cycles needed to verify the behavior of audio playback and VGA output. It is also unnecessary because there is no significant logic in the top-level module and it merely serves to connect our individual pieces. Considering we thoroughly simulated our modules individually there is no loss of verification in not having a testbench for the DE1-SoC.sv file.

Flow Summary

Flow Summary

<<Filter>>

Flow Status	Successful - Wed Jun 05 16:02:45 2024
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	N/A
Total registers	12192
Total pins	84
Total virtual pins	0
Total block memory bits	582,144
Total DSP Blocks	2
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	2
Total DLLs	0

Figure 20: Flow summary for our final design.la

Experience Report

This lab was challenging for us partly because of the broad project description and the freedom we had in choosing our own topic. We initially set out to implement the Fast Fourier Transform to perform note recognition of a microphone input. We spent about 15 hours working on this idea but were ultimately unsuccessful primarily due to our lack of requisite knowledge in the field of digital signal processing. This is still a goal that we would like to work towards in the future and many of the structures we built in this project, such as the audio playback and graphical interface, would aid in its completion.

Working with the new video driver module to communicate with the VGA was another point of difficulty. Our initial approach to this interface was to use a series of hard-coded if statements to divide the playing area into blocks of different colors. This approach is shown in the graph module. As we became more familiar with the new driver we discovered more efficient algorithmic approaches to interface with the VGA through the use of a RAM storing the pixel color for each x and y coordinate.

This lab took approximately 45 hours to complete with time being spent as follows:

- Planning and development of FFT-based approach - 15 hours
- Graphics development - 11 hours
- Audio playback - 4 hours
- User input - 7 hours
- Testbenching - 6 hours
- Lab report - 6 hours