**ECE 385**

Fall 2019

Final Project

# **FPGA Audio Synthesizer and Sampler**

Marco Palella and Zach Wong

Section AB5, Thursday at 11 AM

Xinying Wang, Gene Shiue

***Introduction:***

For our final project design, we wanted to construct a sawtooth wave synthesizer and have a keyboard visual display to go along with it. Additionally, we wanted to have our display also highlight the note being played by the user as well as display the corresponding note. We chose to build a synthesizer because we both have an interest in digital signal processing and audio concepts. Additionally, we believed incorporating digital signal processing would give us a unique challenge as none of the previous labs throughout the semester utilized the concepts/ideas. We not only were able to construct and implement our final project design idea, but we had enough time to add more functionality and build on our original idea. Specifically, we added a piano and drum sampler as well as the ability to play multiple notes at once. Users will have the ability to toggle between which mode they want to select. The graphic display will also highlight the specified keys the user is playing to add a nice visual appeal. Overall, this project helped us gain more knowledge and insight on the concepts taught throughout the course. For our project, we had to expand on the ideas taught in experiment 8 in regards to the VGA interface. Specifically, we drew more elaborate designs like a keyboard as well as incorporate sprites to show the corresponding notes. To implement the piano and drum sampler we decided to use SRAM to store the data. Therefore, we had to use our knowledge of experiment 6’s concepts in regards to fetching memory addresses and storing/extracting data from SRAM. For the synthesizer, we used wavetables to store our samples. We also had to work with the audio driver on the DE2 board and communicate with that chip in order to correctly play audio from the audio jack on the board. Overall, we enjoyed working on our project and are happy with the functionality we were able to add onto our original project idea.

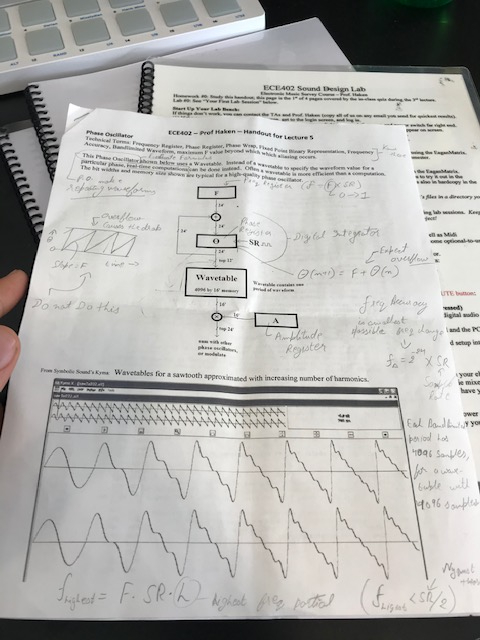
***Written Description of Final Project System:***

Our final project system is comprised of three primary components of hardware (SystemVerilog) and one software component (C code). For the software component, we expanded on Lab 8’s concepts and ideas of polling USB keyboard data using the Cypress chip and communicating that data back to the NIOS II. The hardware component is comprised of the synthesizer, piano/drum sampler, and graphics. The synthesizer portion was formed by using wavetables to function as look-up tables in order to extract our sampling data. The data would then be fed to the audio driver on the DE2 board and be outputted to the user in form of audio. The piano/drum sampler component was constructed differently than the synthesizer. Instead of using wavetables, we decided to take in .wav files each containing different notes and combining them into one .wav file. From there, we store the .wav file data into SRAM and would fetch the data accordingly. Once fetched, the data would be sent to be outputted as audio. The graphics portion relied on using the VGA controller module to keep track of where the “beam” on the VGA interface is currently drawing the pixel on screen. Once known, we simply used a combination of rectangles to draw the piano keyboard. We additionally needed to use the pixel coordinates to implement our sprites as well as highlight certain keys on the keyboard when pressed by the user. Combining all these components makes our final project. A project that combines both a synthesizer and sampler into one machine and users can toggle to which mode they will like to select as well as either playing one key or multiple keys.

***Written Description and Diagrams of the Sampler/Synth (Hardware):***

***Written description of the synthesizer:***

Our implementation of the synthesizer utilizes the concept of implementing wavetables to function as look-up tables when grabbing sampling data. The data stored in the wavetables were pre-computed values we calculated in order to correctly and precisely synthesize up to seven harmonics of a sawtooth wave together. Wavetable synthesis is a practical synthesizing technique that is used by many when creating synthesized sound for a variety of products. In order to fetch our stored frequency data held within the wavetables, we mapped out twelve different keyboard keys to correspond with specific notes. Each key mapped would be given a specific frequency from the following equation , with the frequency sample rate being 48kHz. Once each key has its own specific, unique frequency, when a key is pressed the frequency signal would be sent to certain registers for holding and slowly be passed into our phase register. This phase register will then send the top twelve bits to the look-up tables and have that data be sent to the audio driver to be outputted to the user. To add in our harmonics for the synthesizer, we simply adjusted our phase register module. Specifically, we had to add in the phase based off of the frequency from the key press like the original phase register, but then we need to add that frequency to the phase. To produce the second harmonic, we simply need to add the frequency twice and for the third harmonic add the frequency three times and so forth. When sending the data to the audio driver we need to ensure it was being sent on the sampling clock. However, since the sampling clock was a square wave, it was being high for too long. Therefore, we implemented a separate component called pulse that simply captures one cycle of when the sampling clock is high. We used this counter to base off when we should be sending data to the audio driver. The image below is brief visual display and description of the wavetable we used for our synthesizer.



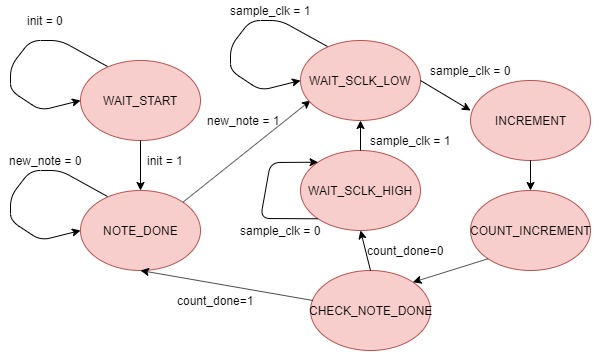
***Written description of the Audio Sampler:***

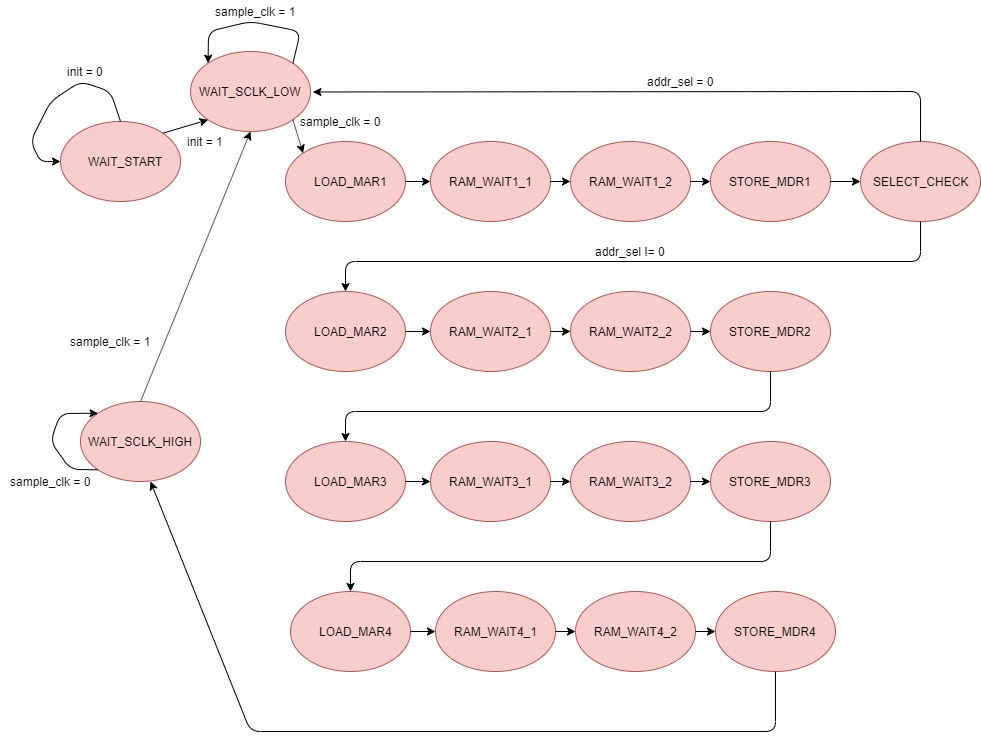
For our audio sampler we implemented both a piano and a drum sampler. Unlike the synthesizer, we used SRAM as our storage device holding the specific data needed for each instrument note. We found each sample of the specific instrument’s notes by gathering .wav file samples online. Specifically, for the piano sampler we gathered twelve different samples representing each piano key stored in a .wav file. After we collected all of the piano samples we combined the twelve .wav files into one .wav file. We followed the same procedure for gathering and combining the drum sampler data. Once we had both piano and drum samples combined in their respective .wav files, we combined it all together into a single .wav file and stored in into SRAM. In order to fetch the right memory addresses of our data in SRAM, we used similar techniques we implemented for Lab 6 when we built a simplified version of LC-3. Specifically, we incorporated techniques used to produce the MAR, MDR, and tri-state buffer. We mapped each keyboard key press input to correspond to a memory address of a sample of both the piano and drum sampler. We then built a state machine that will take in the keyboard key press input, if any, read off the corresponding memory address and fetch it within SRAM. The memory address is held in certain registers as well as the data. Once the address is found in SRAM, the data contained in that memory address will be moved into its own register and then be sent to the audio driver to output the corresponding sound. To handle multiple keyboard key presses, we simply created more instances of our registers holding the respective memory and data. Each memory address register would put in its corresponding data into its own specific data register. To output and produce the sound of multiple piano keys or drum samples played out, we add up the data in each of the four data registers representing the four possible keyboard inputs and outputs that data to the sound driver. Therefore, if there was only keyboard key being pressed the other three data registers would contain nothing and the data outputted to the sound driver would be the data of the single sample. We also separated the piano and drum sampler into two different functional modes by splitting it up using a push-button signal on the DE2 board. Users will have a choice of function mode by pressing this corresponding push-button to toggle between outputting a piano sampler and a drum sampler as well as the synthesizer too.

***Written description of the Graphics:***

For the graphic component of our project, we relied on using the DrawX and DrawY counters from the VGA controller module in order to keep track of where the “beam” was currently drawing in the pixel. For our display, we wanted to show a one octave piano keyboard as well as highlight both single and multiple keys that are being pressed by the user. Finally, we wanted to display sprites on the screen to show users what notes they are playing. To construct the piano keyboard, we simply constructed rectangles based off the interface display size and colored them either white or black depending on the placement of the rectangle. After drawing the rectangles, we add black lines to outline and separate each key in order to make it more visually appealing. Essentially, we had each piano keyboard key be mapped to a certain area/boundary of the display. We used a mapper to map each keyboard press to a specific piano keyboard key on the display. When a key is pressed, we simply highlight the area/boundary of that piano key on the display and make it a different color other than its usual white or black. In our implementation we chose green. This behavior highlights the piano key on the display showing the user which piano key is being represented by the keyboard key. We used this same technique to be able to highlight multiple notes as well. For the sprites, we wanted to display them to show to the user which piano note is being played based off of keyboard press. We used a sprite table to hold the data for our eight sprites. Originally, these sprites were too small, so we enlarged them to make them visible enough to the user. We essentially mapped each sprite to a certain key press similar to how we highlighted the piano keys. Whenever, a key press was sent that corresponding sprite will be displayed. We reserved two spaces on the screen for the sprites. We needed to reserve a second space to represent the sharp piano notes. When a sharp piano key is pressed, it will display the corresponding letter sprite but also the sharp symbol sprite to correctly display the right note. We colored the sprites green to go along with the highlighting of the piano keys.

***State Diagram of SRAM sampler control:***

******

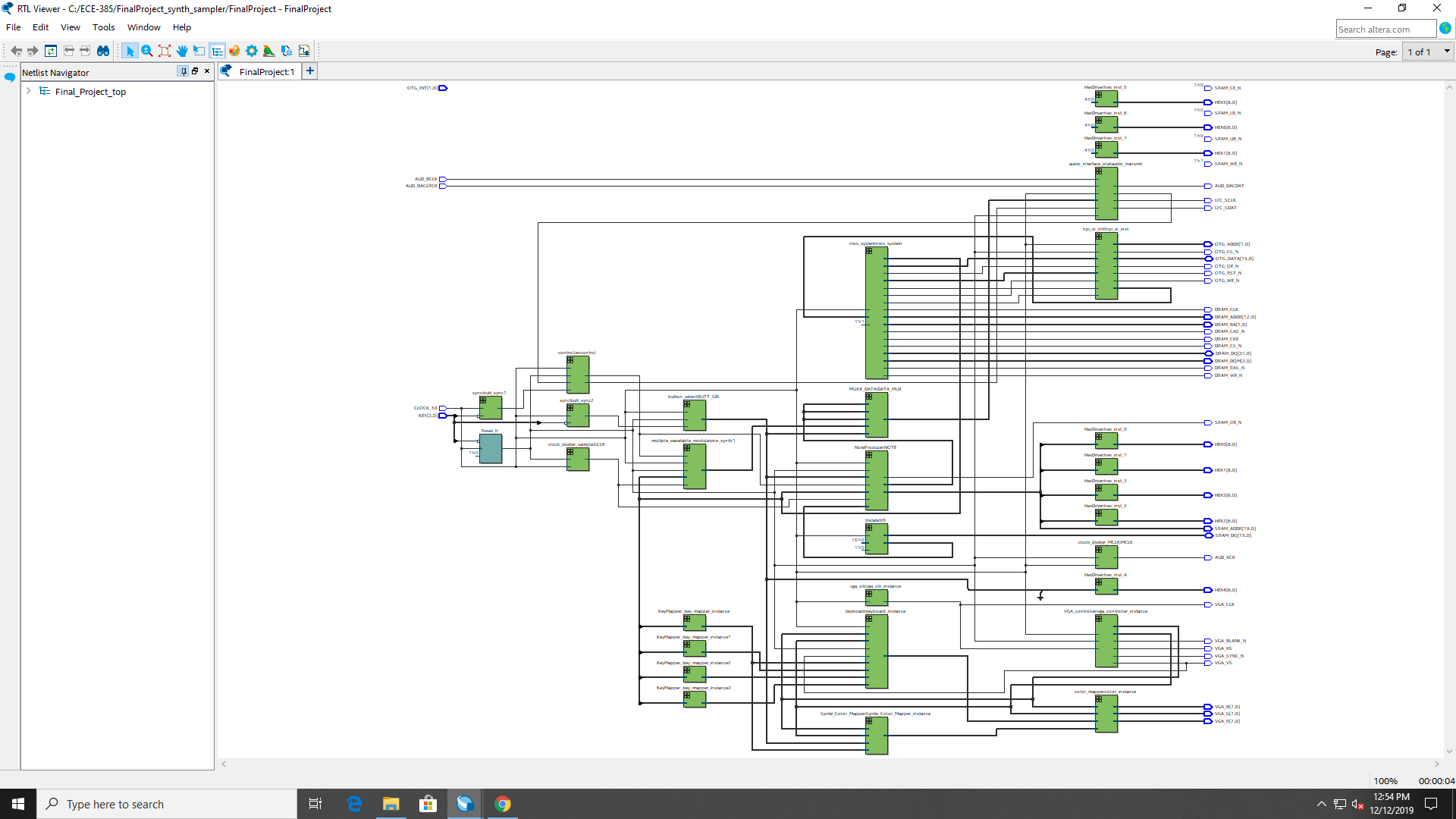
***State Diagram of sampler note address control:***

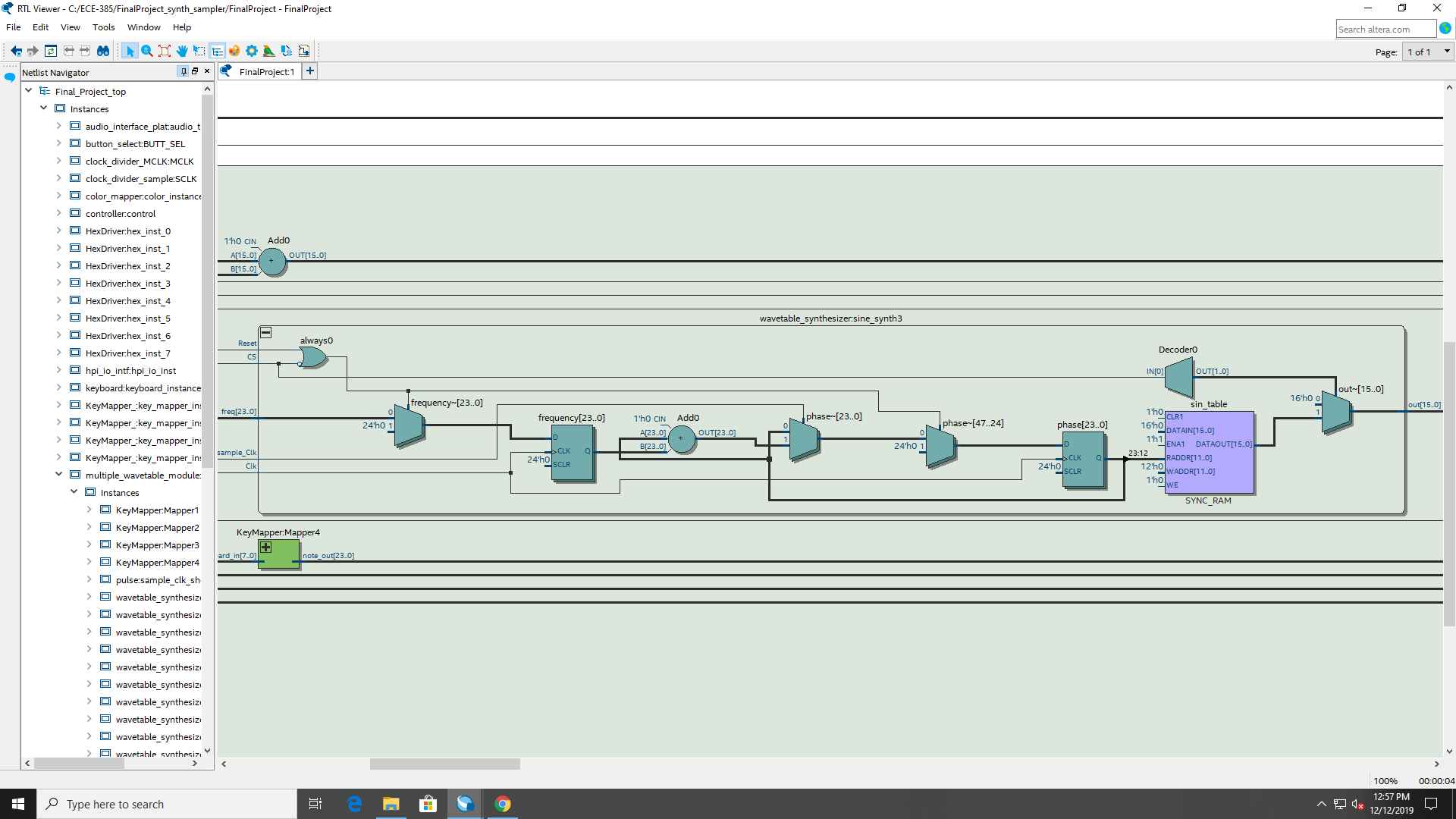
***Written Description Software Components:***

Our final project did not contain a lot of software components. Specifically, we only had one software component which was reading USB keyboard data. We originally implemented reading the USB keyboard data code from Lab 8 when we had to use the keyboard input to move the white ball around the screen. We took that same USB keyboard data code that establishes communication from the USB port and the Cypress chip to the NIOS II. We expanded on the code to read in four keyboard inputs instead of the original implementation of just reading in a single keyboard input. Essentially, we originally were reading in an eight bit keycode to represent a keyboard press. We changed it to reading a thirty-two bit keycode to represent four keyboard presses. In order to read in four keyboard presses we had to change the code. Specifically, the original implementation was reading from an address containing the first two keycodes (sixteen bits). To read in the last two keycodes, we simply had to read from address that was sixteen bits away from the original address (0x051e) which was 0x0520. We then had to adjust our keyboard definition from a volatile char to a volatile int because we were reading a thirty-two bit number rather than an eight bit number. Those were all the software changes we needed to make to ensure we were reading multiple keyboard presses. The only other thing we needed to change to was in our platform designer and modifying the keyboard PIO block to thirty-two bits rather than eight bits. The action of reading from the USB keyboard data being sent from the USB port was the only software component we implemented for our project. The code was similar to the implementation need for Lab 8, but we expanded on it to read in multiple keyboard presses.

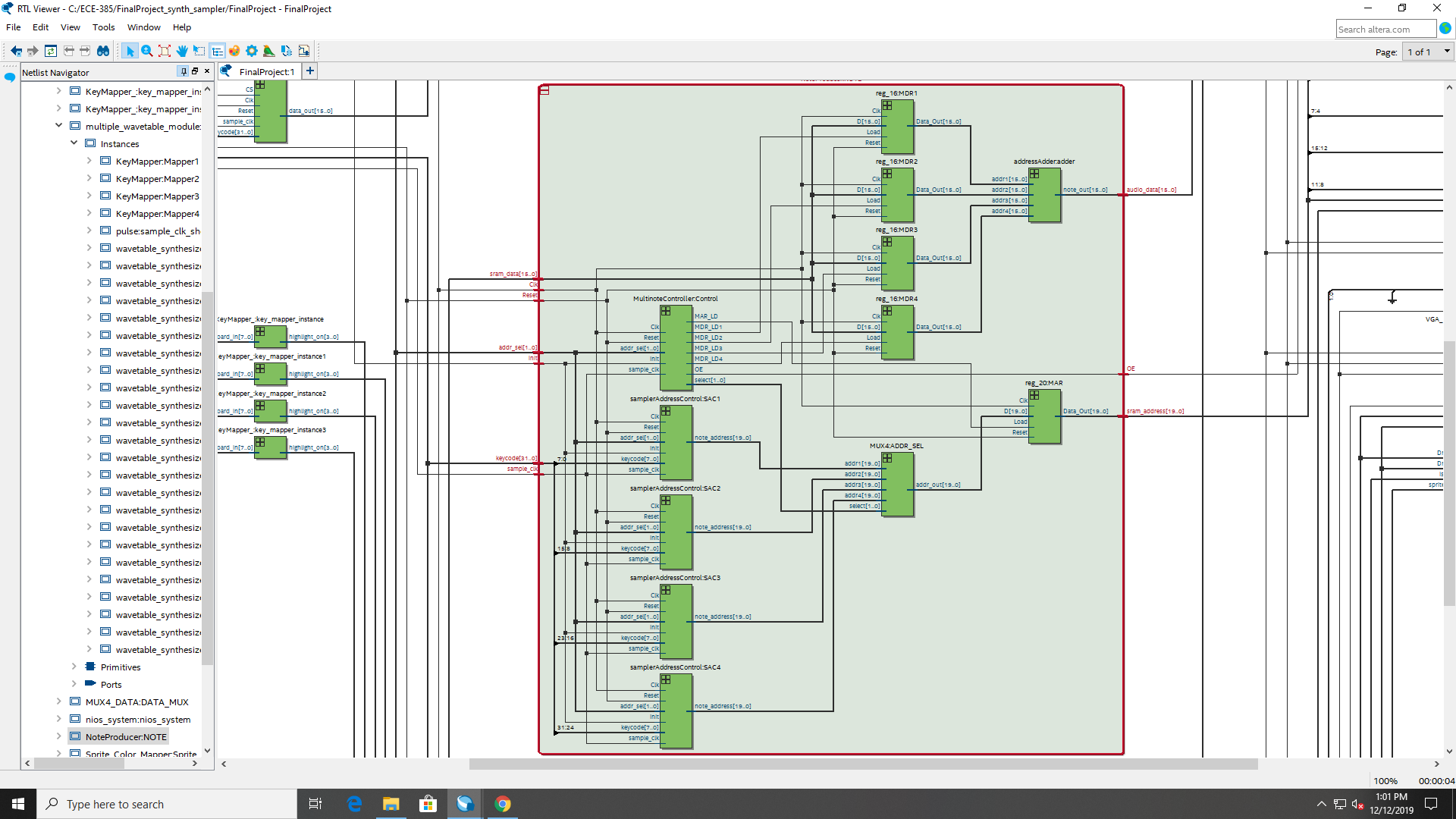
***Block Diagrams:***

**Top Level of Project:**

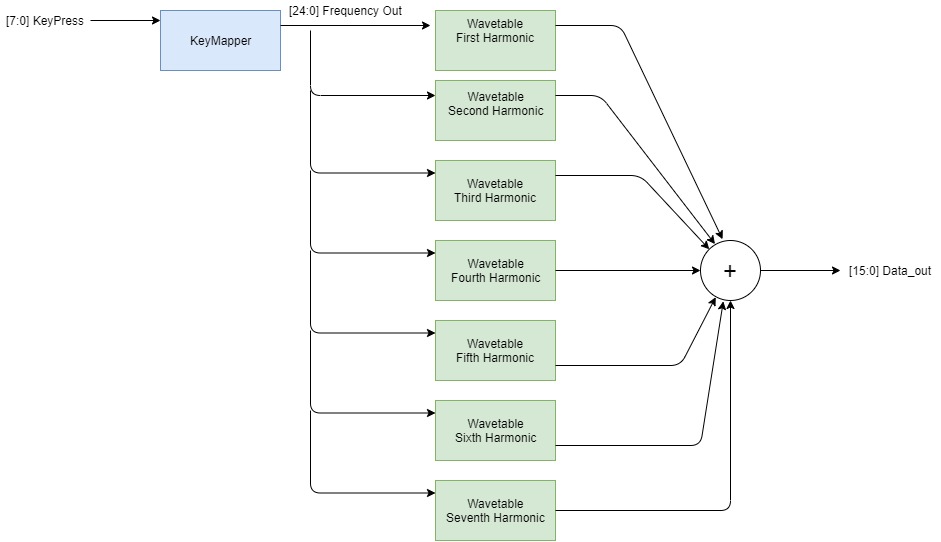
******

**Wavetable Synthesizer:**

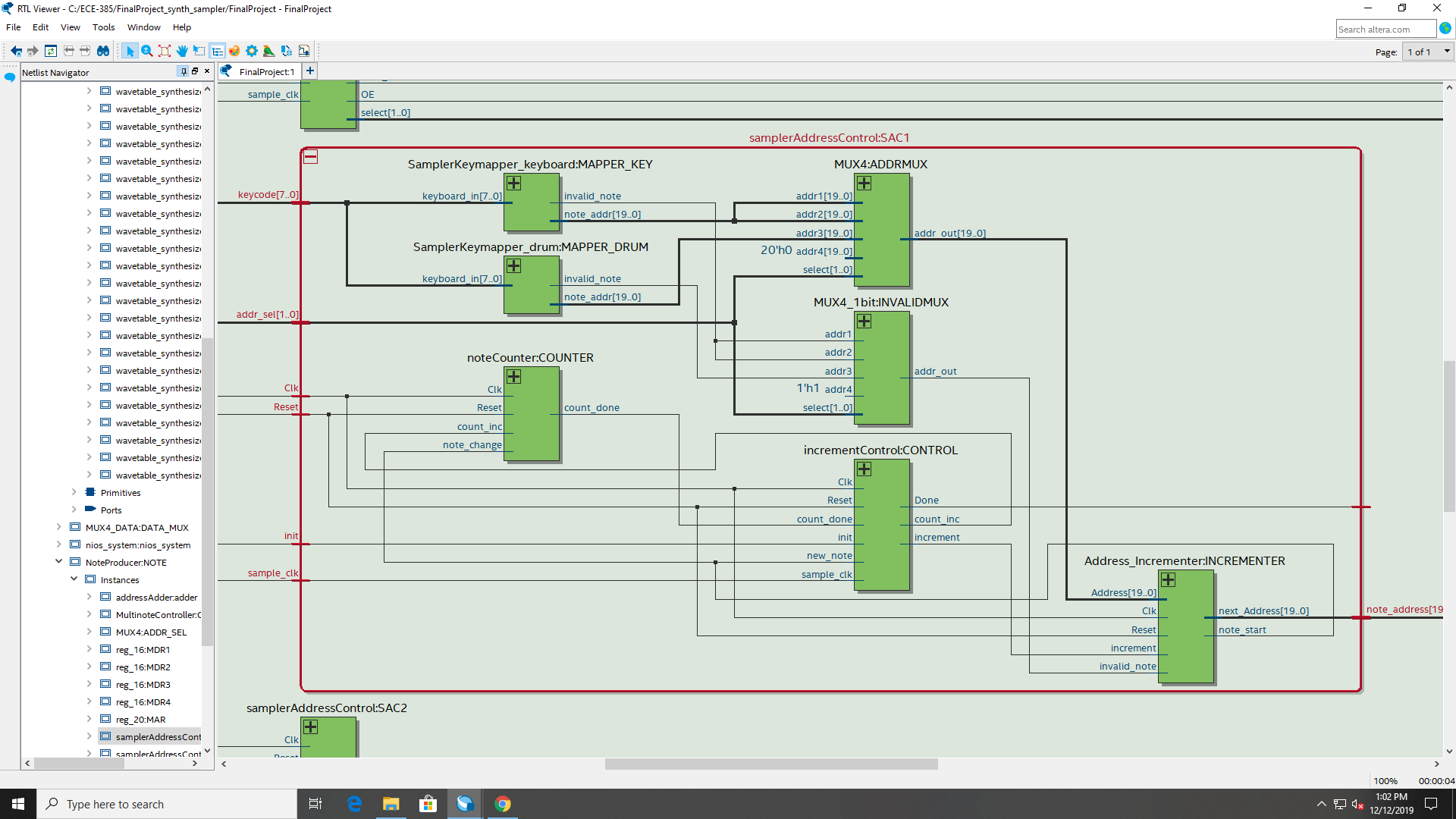
**NoteProducer:**

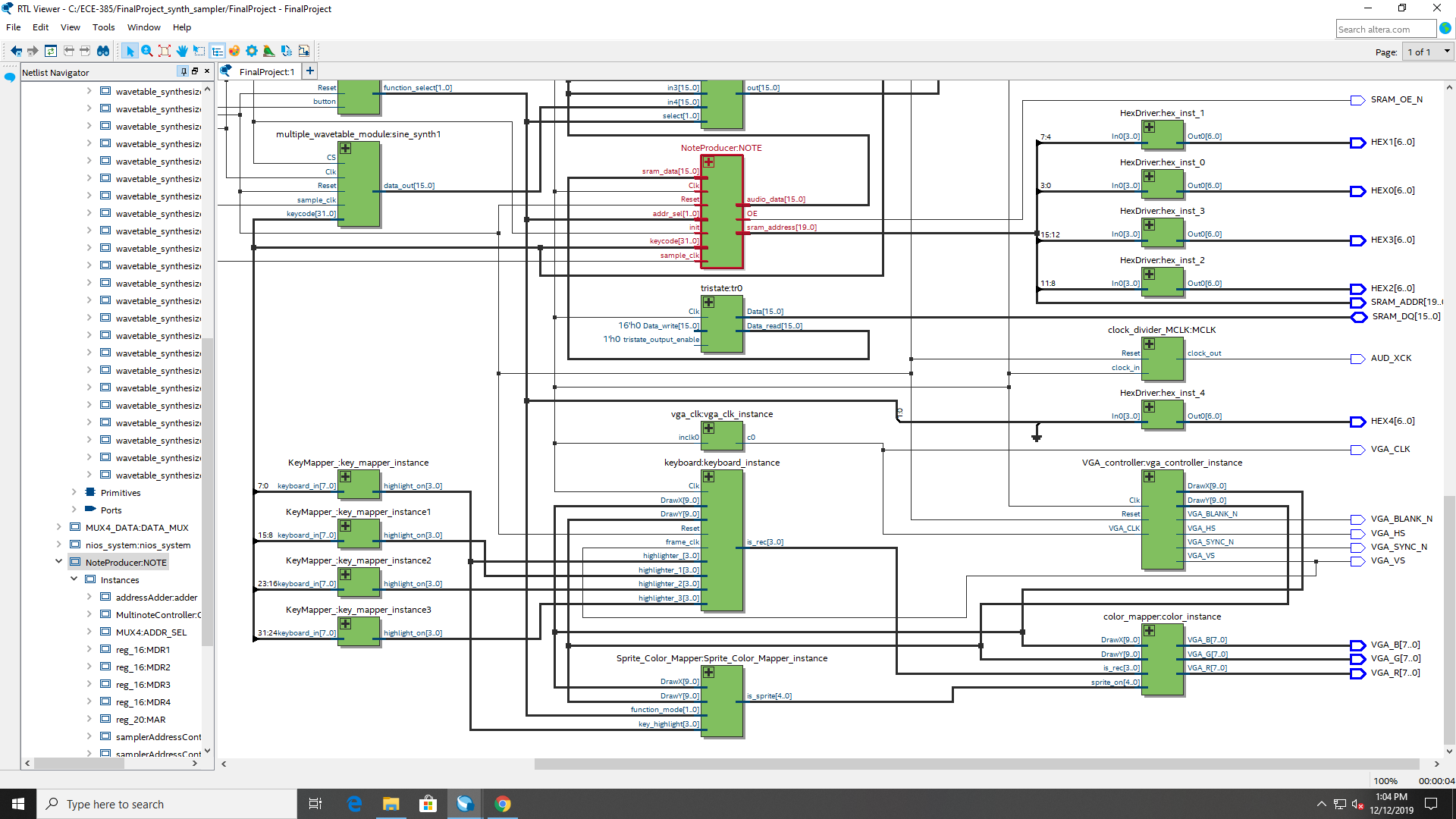
****

**Multiple Wavetable Synth:**

****

**Sampler Address Control Unit for Fundamental harmonic:**

**Graphics Block:**



***Written Description of .SV Modules:***

***Final\_Project\_top.sv***

**Module:** *Final\_Project\_top*

**Inputs:** CLOCK\_50, AUD\_BCLK, AUD\_DACLRCK

[1:0] OTG\_INT

[3:0] KEY

**Outputs:** VGA\_CLK, VGA\_SYNC\_N, VGA\_BLANK\_N, VGA\_VS, VGA\_HS, OTG\_CS\_N, OTG\_OE\_N,

OTG\_WE\_N, OTG\_RST\_N, DRAM\_RAS\_N, DRAM\_CAS\_N, DRAM\_CKE, DRAM\_WE\_N,

DRAM\_CS\_N, DRAM\_CLK, AUD\_XCK, AUD\_DACDAT, I2C\_SCLK, I2C\_SDAT, SRAM\_CE\_N,

SRAM\_UB\_N, SRAM\_LB\_N, SRAM\_OE\_N, SRAM\_WE\_N

[1:0] DRAM\_BA

[3:0] DRAM\_DQM,

[6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7

[7:0] VGA\_R, VGA\_G, VGA\_B, VGA\_CLK

[12:0] DRAM\_ADDR

[19:0] SRAM\_ADDR

**In/Out:** [15:0] OTG\_DATA, SRAM\_DQ

**Description:** this is the top level module that contains all other modules, and has I/O for all external chips on DE-115 board

**Purpose:** this modules purpose is hold all modules that will be contained inside FPGA, and create a peripheral for external chips

***multiple\_wavetable\_module.sv***

**Module:** *multiple\_wavetable\_moduler*

**Inputs:** Clk, Reset, CS, sample\_clk

[31:0] keycode

**Outputs:** [15:0] data\_out

**Description:** This module contains 28 wavetables and 4 keymappers, with 7 wavetables dedicated to a single keymapper. Each of these wavetables is one of the first 7 harmonics of a sawtooth, and all of the wavetable outputs are added together and outputted as data\_out.

**Purpose:** This module is the toplevel of a 7-harmonic polyphonic sawtooth wave, which is produced using wavetables and a keymapper. The module is capable of outputting four notes at a time.

***wavetable\_synthesizer.sv***

**Module:** *wavetable\_synthesizer*

**Inputs:** Clk, Reset, CS, sample\_Clk

[23:0] freq

**Outputs:** [15:0] out

**Description:** This module takes in a frequency from the keymapper, which corresponds to the frequency that a sine wave will be played at. The desired frequency to be played is stored in a register every clock cycle. There is an additional phase register, which the frequency is added to every time sample\_clk is set high. The 12 most significant bits of the phase register is used as an address in a lookup table which contains 4096 values of a sine wave with max amplitude of 2048.

**Purpose:** This module is the baseline that allows for the data of a sine wave with a specific frequency to be played at any given sampling rate. This data will be sent to the audio driver at a rate of 48 kHz to play a sine out of the line out jack.

***wavetable\_synthesizer\_harm\_(n).sv***

**Module:** *wavetable\_synthesizer\_harm\_(n)*

**Inputs:** Clk, Reset, CS, sample\_Clk

[23:0] freq

**Outputs:** [15:0] out

**Description:** This module works the same way as wavetable\_synthesizer, using a lookup table to send data for a sine wave that has a frequency of n\*freq and a maximum amplitude of (2048/n), n representing whicher harmonic the verilog file represents.

**Purpose:** This module is needed to give the synthesizer more aesthetics, and make it sound closer to a sawtooth

***pulse.sv***

**Module:** *pulse*

**Inputs:** Clk, Reset, data

**Outputs:** out

**Description:** This module is a simple state machine that sends a clock high pulse every time a signal raises high and then waits for the signal to drop low before sending another pulse

**Purpose:** This module ensures that the wavetable is changing values only once every sample clock cycle.

***KeyMapper.sv***

**Module:** *KeyMapper*

**Inputs:** [7:0] keyboard\_in

**Outputs:** [23:0] note\_out

**Description:** this module mapped 12 different values of keys on the USB keyboard to precalculated frequency values. Each key is to play a note that coincides with a scale on the keyboard. These notes each have a certain fundamental frequency affiliated with it. For example, an A4 has a fundamental frequency of 440 Hz. The calculated frequency that each key is calculated with the following equation: , with being 48 kHz.

**Purpose:** this modules purpose is to be able to convert key presses to a frequency that can be sent to a wavetable and have it play at that frequency

***controller.sv***

**Module:** *controller*

**Inputs:** Clk, Reset, init\_finish, start

**Outputs:** init, CS

**Description:** this module is a state machine that has two inputs, start and init\_finish, which are being sent from the audio driver. There are two outputs: init, which is sent to the audio driver to start the initialization process when the start button is pressed, and CS, which turns on all the other modules that are creating sound data

**Purpose:** this state machine is necessary to prevent audio data to start being sent to the audio codec until all I2C initialization information has been sent to the codec by the audio driver

***tristate.sv***

**Module:** *tristate*

**Inputs:** Clk, tristate\_output\_enable

[15:0] Data\_write

[15:0] Data

**Outputs:** [15:0] Data, Data\_read

**Description:** This module is a tristate buffer is a register, which stores data coming in and out of SRAM

**Purpose:** This module is necessary to accommodate for the fact that SRAM is asynchronous, and our device is synchronous

***clock\_divider\_MCLK.sv***

**Module:** *clock\_divider\_MCLK*

**Inputs:** clock\_in, Reset

**Outputs:** clock\_out

**Description:** This module uses a counter to make the FPGAs 50 MHz clock slower

**Purpose:** This module is necessary to create a master clock that can be used by the audio codec to analyze data

***clock\_divider\_SCLK.sv***

**Module:** *clock\_divider\_SCLK*

**Inputs:** clock\_in, Reset

**Outputs:** clock\_out

**Description:** This module uses a counter to make the FPGAs 50 MHz clock slower

**Purpose:** This module is necessary to create a sample clock that will be used to send data at the correct rate to the audio codec.

***Synchronizers.sv***

**Module:** *sync*

**Inputs:** Clk, d

**Outputs:** q

**Description:** this device ensures that data is only changed from the input every clock cycle

**Purpose:** This module is used to ensure that debouncing doesn't impact inputs into the FPGA

***HexDriver.sv***

**Module:** *HexDriver*

**Inputs:** [3:0] In0

**Outputs:** [6:0] Out0

**Description:** this is a simple mapper that converts corresponding signals with the correct values for the seven digit display

**Purpose:** Used to show hex values on the 7 segment displays on the FPGA

***hpi\_io\_intf.sv***

**Module:** *hpi\_io\_intf*

**Inputs:** Clk, Reset, from\_sw\_r, from\_sw\_w, from\_sw\_cs, from\_sw\_reset

[1:0] from\_sw\_address

[15:0] from\_sw\_data\_out

**Outputs:** OTG\_RD\_N, OTG\_WR\_N, OTG\_CS\_N, OTG\_RST\_N

[1:0] OTG\_ADDR

[15:0] from\_sw\_data\_in

**In/Out:** [15:0] OTG\_DATA

**Description:** this is a driver that communicates the EZ-OTG chip. It takes in information given by the NIOS processor and ensures that it is being sent correctly to the device

**Purpose:** this modules purpose is to act as a driver between our program and the EZ-OTG chip

***audio\_interface\_plat.sv***

**Module:** *audio\_interface\_plat*

**Inputs:** Clk, Reset, INIT, AUD\_DACLRCK, AUD\_BCLK

[15:0] DATA

**Outputs:** INIT\_FINISH, AUD\_MCLK, AUD\_DACDAT, I2C\_SDAT, I2C\_SCLK, data\_over

**Description:** this module is a driver for the Audio Codec that is on the DE2-115 board. This driver contains the logic to send initialization info to the codec, which is done via I2C. The driver also takes in a parallel 16-bit audio data word, and converts it into a serial data stream, which is sent to the audio codec via I2S formatting. The initialization process begins when init is raised high, and when the initialization process is over, init\_finish is raised high.

**Purpose:** This codec has a DAC on it, which will be used to convert the digital data being made by the synthesizer and sampler modules into an analog signal.

***button\_select.sv***

**Module:** *button\_select*

**Inputs:** Clk, Reset, button

**Outputs:** [1:0] function\_select

**Description:** this is a state machine used to set the FPGA into four different modes. It changes states every time button is held high and goes to Mode 0 when the button is pressed on mode 4

**Purpose:** This module controls whether single note piano, drum-pad, polyphonic piano, or the wavetable synth mode is set. The function select controls several muxes that change which audio data is being sent to the audio driver, along with controlling several datastreams inside of the noteProducer module.

***NoteProducer.sv***

**Module:** NoteProducer

**Inputs:** Clk, Reset, sample\_clk, init

[1:0] addr\_sel

[15:0] sram\_data

[31:0] keycode

**Outputs:** [15:0] audio\_data

[19:0] sram\_address

**Description:** this module contains four Sampler Address Control units, which are used to communicate to the SRAM what address should be read. This address is stored inside of an MAR, which is connected to the SRAM tristate outside of this module. There are also four MDRs in this module that store the data values for each of four addresses being sent by the Sampler Control Units. The outputs of the MDRs are all added together in an adder and outputted to the audio driver. There is a control unit that determines which Sampler Address Control unit is sending an address value to the MAR, and which MDR is being loaded. Addr\_sel toggles the input address values from two different mappers, which maps the starting address to each audio sample in SRAM.

**Purpose:** this is a toplevel of a module that communicates with SRAM to contain audio data for the piano and drum samples stored in SRAM. The four Sampler Address Control Units allow for polyphonic capabilities for up to 4 notes pressed at the same time.

***reg\_16.sv***

**Module:** *reg\_16*

**Inputs:** Clk, Reset, Load

[15:0] D

**Outputs:** [15:0] Data\_Out

**Description:** this is a 16 bit register that outputs the input when load is set to high

**Purpose:** this register will be used as MDRs from SRAM. Each MDR corresponds to one of the Sampler Address Control modules, which corresponds to one of the four possible keycode presses

***reg\_20.sv***

**Module:** *reg\_20*

**Inputs:** Clk, Reset, Load

[19:0] D

**Outputs:** [19:0] Data\_Out

**Description:** this is a 20 bit register that outputs the input when load is set to high

**Purpose:** this register will be used as MDRs from SRAM. Each MDR corresponds to one of the Sampler Address Control modules, which corresponds to one of the four possible keycode presses

***MultinoteController.sv***

**Module:** *MultinoteController*

**Inputs:** Clk, Reset, sample\_clk, init

[1:0] addr\_sel

**Outputs:** MAR\_LD, MDR\_LD1, MDR\_LD2, MDR\_LD3, MDR\_LD4, OE

[1:0] select

**Description:** this is a state machine that is in charge of synchronizing data reception from SRAM. This device controls when the MDR registers and MAR register stores data. This controller is also in charge of the select on a mux inside of the NoteProducer module. This Mux is used to choose which addresses from the four Sampler Address Control units will be sent to the MAR. The state machine repeatedly sends each of the four addresses coming from each Sampler Control Unit to SRAM and stores the data sent back from SRAM into each MDR (one MDR corresponds to the data from an address being sent from one of the Sampler Control Units). The controller waits until the falling edge of the clock to repeat this process to ensure that the data is being outputted at the sampling frequency of the audio driver. If addr\_sel is set to single note mode, then it will only grab/receive data for the first Sampler Control Unit.

**Purpose:** The purpose of this module is to control the process of communicating data to SRAM for the piano and drum sampler and outputting it to the audio driver. The purpose for using four Sample Control Units is to allow for polyphonic playing.

***samplerAddressControl.sv***

**Module:** *Address\_Incrementer*

**Inputs:** invalid\_note, increment, Clk, Reset

[19:0] Address

**Outputs:** note\_start

[19:0] next\_Address

**Description:** this is the top level for a sampler address control unit. The Sampler Address Control unit contains a control unit, an incrementer unit, a counter, and an AddressKeymapper.

**Purpose:** This unit serves the purpose of determining which address needs to be accessed in SRAM based off of the ASCII value of an individual keyboard press

***Address\_Incrementer.sv***

**Module:** *Address\_Incrementer*

**Inputs:** invalid\_note, increment, Clk, Reset

[19:0] Address

**Outputs:** note\_start

[19:0] next\_Address

**Description:** this module serves two purposes. Its first purpose is to take in the address value coming from an Address Mapper. If the value of the mapper changes to a valid note, then the output address of the incrementer is set to whatever address is being inputted. Its other purpose is to increment whatever the current address is by one every time the increment input is set to high.

**Purpose:** this register serves as the only module inside the Sampler Address Control Unit that manipulates the output address of the Sampler Address Control Unit

***SamplerKeymapper\_keyboard.sv***

**Module:** *SamplerKeymapper\_keyboard*

**Inputs:** [7:0] keyboard\_in

**Outputs:**  invalid\_note

[19:0] note\_addr

**Description:** this module is a mapper that takes in one of the four values coming from the keyboard and maps it to an address that is sent to the address incrementer. The module also determines whether the note is valid or not so that the address incrementer knows whether to set the address at the start of a new note.

**Purpose:** the purpose of this module is to convert the keyboard values into something that the incrementer can work with

***noteCounter.sv***

**Module:** *noteCounter*

**Inputs:** Clk, Reset, count\_inc, note\_change

**Outputs:**  count\_done

**Description:** this is a counter module that counts up to 30,870 (the size of one note in SRAM) then sends a high signal via count\_done for a clock cycle.

**Purpose:** the purpose of this module is to let the controller on the Sampler Address Control Unit know when to stop telling the address incrementer to shift addresses.

***incrementControl.sv***

**Module:** *incrementControl*

**Inputs:** Clk, Reset, new\_note, count\_done, sample\_clk, init

**Outputs:**  Done, count\_inc, increment

**Description:** this is a state machine that controls the actions of the Sampler Address Control Unit. Every sample clock cycle, the control unit increments the address on the incrementer by one and tells the note counter to move up by one. When count\_done is set to high, the controller waits until a new note is pressed, and repeats the process.

**Purpose:** This modules purpose is to act as the controller for the Sampler Address Control Unit.

***Sprite\_Color\_Mapper.sv***

**Module:** *Sprite\_Color\_Mapper*

**Inputs:** logic [3:0] key\_highlight, logic [1:0] function\_mode, input logic [9:0] DrawX, DrawY

**Outputs:**  logic [4:0] is\_sprite

**Description:** This module determines what keyboard presses are coming from user and routes each press to the specific sprite that represents that musical note. The specified note based on the key press is displayed on the VGA interface.

**Purpose:** This module allows for users to see what note is being displayed based on the key presses from the user.

***KeyMapper\_.sv***

**Module:** *KeyMapper\_*

**Inputs:** logic [7:0] keyboard\_in

**Outputs:**  logic [3:0] highlight\_on

**Description:** This module determines what keys are being pressed by the user and based on the key presses sends a signal to highlight the specified keys on the VGA display. Specifically, the signal sent tells which boundaries of the screen should be a different color in order to highlight the corresponding key based off keyboard input.

**Purpose:** This module allows for the ability for multiple keyboard notes to be highlighted based on user keyboard input and display it on the VGA interface.

***keyboard.sv***

**Module:** *keyboard*

**Inputs:** logic Clk, Reset, frame\_clk, logic [3:0] highlighter\_, highlighter\_1, highlighter\_2, highlighter\_3, logic [9:0] DrawX, DrawY

**Outputs:**  logic [3:0] is\_rec

**Description:** This module maps out the boundaries and areas of the VGA display in which we want both the keyboard and sprites to be displayed. Specifically, this module maps out the areas in which keys on the keyboard should be drawn in addition to the sprites on the VGA interface.

**Purpose:** This module draws and maps out the boundaries on the VGA interface in which the keys on the keyboard should be displayed as well as the sprites. Additionally, this module also sends what color should be drawn for each key and sprite as well as change the color if the key is highlighted.

***color\_mapper.sv***

**Module:** *color\_mapper*

**Inputs:** logic [3:0] is\_rec, logic [4:0] sprite\_on, logic [9:0] DrawX, DrawY

**Outputs:**  logic [7:0] VGA\_R, VGA\_G, VGA\_B

**Description:** This module determines which colors need to be drawn on the VGA interface. Specifically, what areas of the display should be drawn one color and other areas drawn a different color.

**Purpose:** This module colors the VGA interface with the colors we want on our display. Specifically, this module colors the keyboard keys black and white respectively as well as color the keys green whenever a key is pressed by the user along with the sprites.

***font\_rom.sv***

**Module:** *font\_rom*

**Inputs:** logic [9:0] addr

**Outputs:**  logic [63:0] data

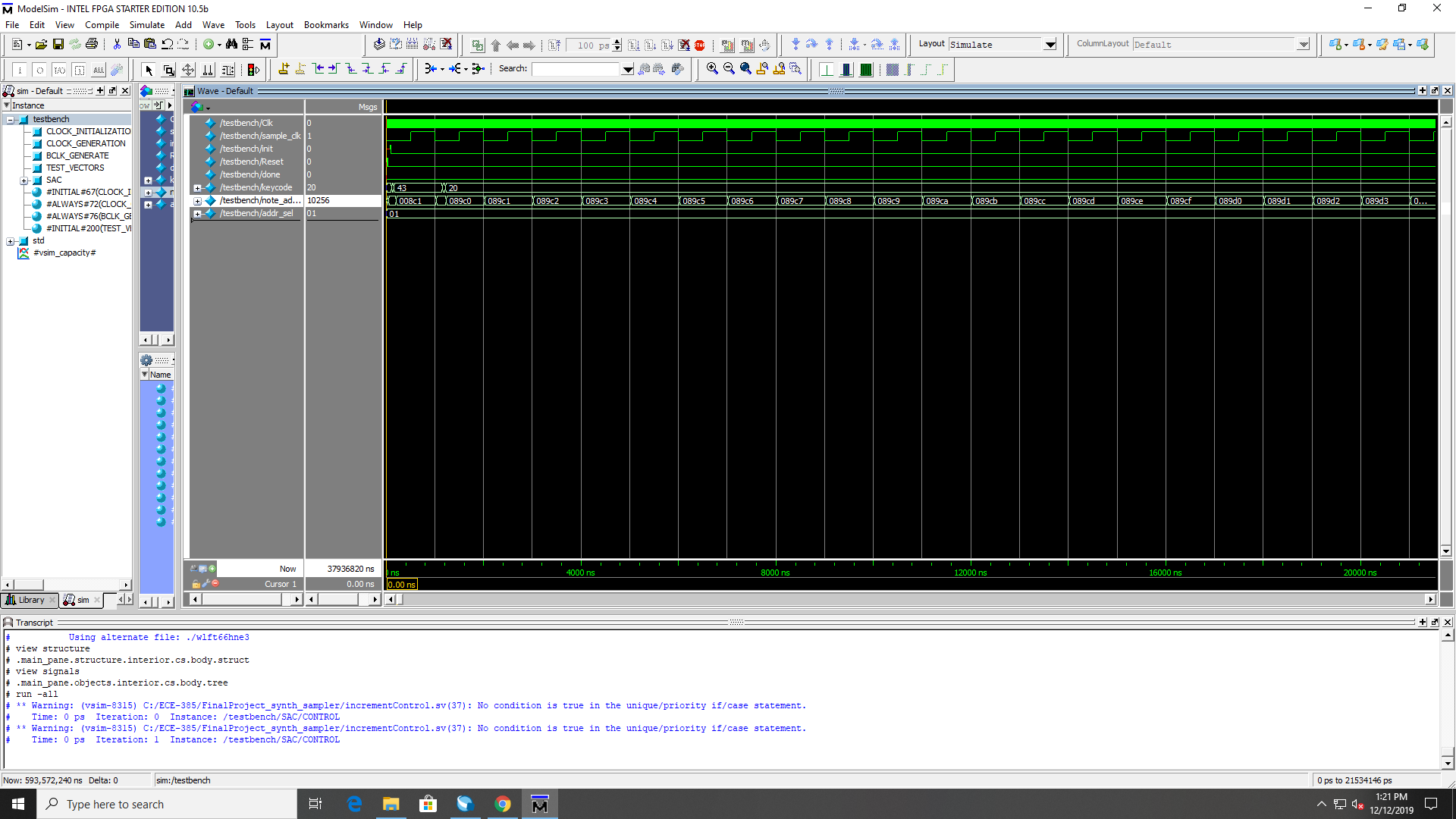
**Description:** This module contains the sprite table with all the sprites we will be using for our VGA display. Specifically, this sprite table contains the notes corresponding to one octave on the piano. We enlarged the sprites in order for the size to be adequate for the user. Based on the keyboard input, we select the correct sprite from the table and display it on screen.

**Purpose:** This module contains the sprites used for our final project as well as the data needed in order to draw them to represent the corresponding piano key notes.

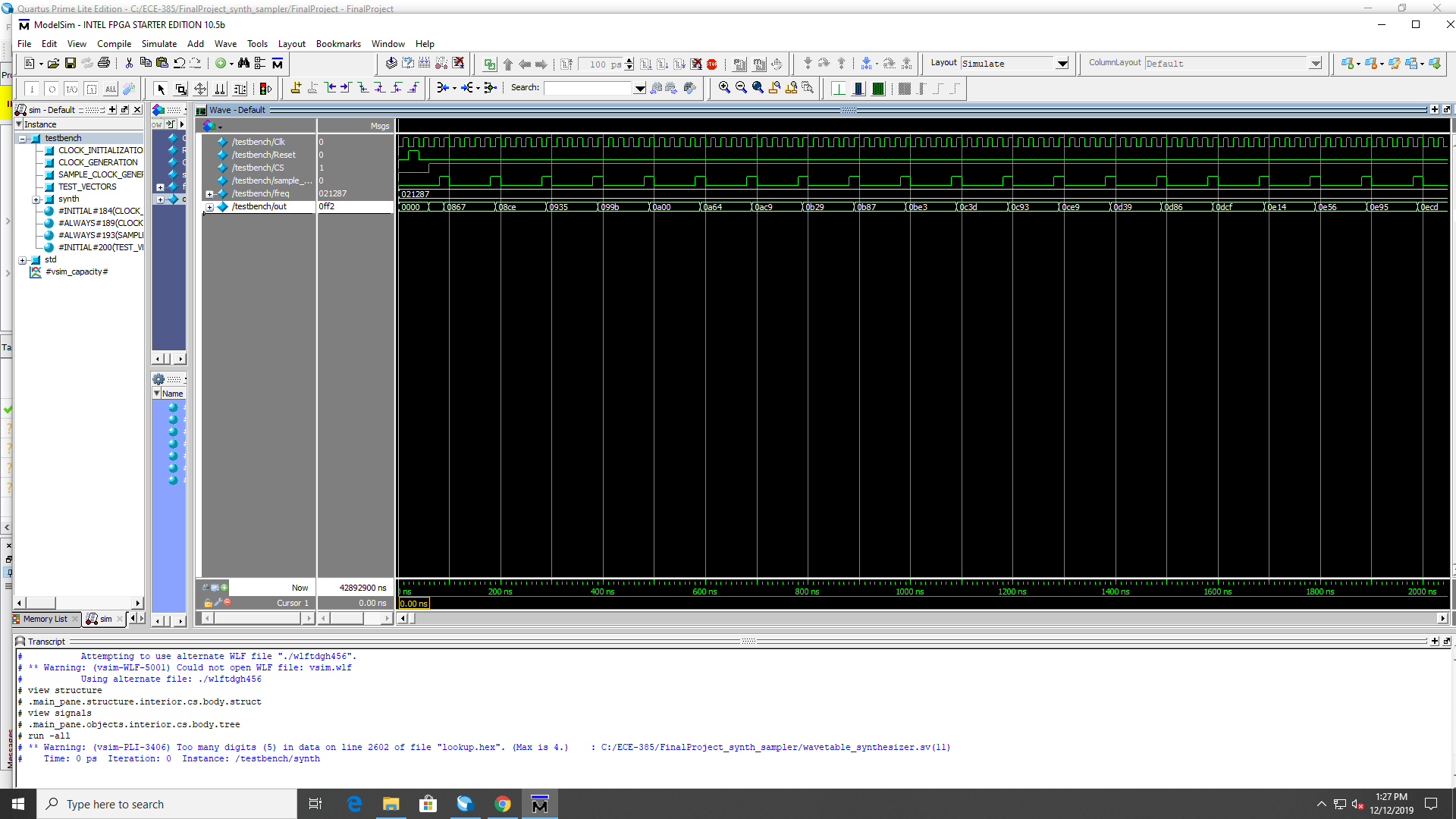
***Simulations:***

***\*note:*** all clock signals inside of simulations are not the same as the actual clock speeds being used. They are long enough to prove that the modules will work as expected.

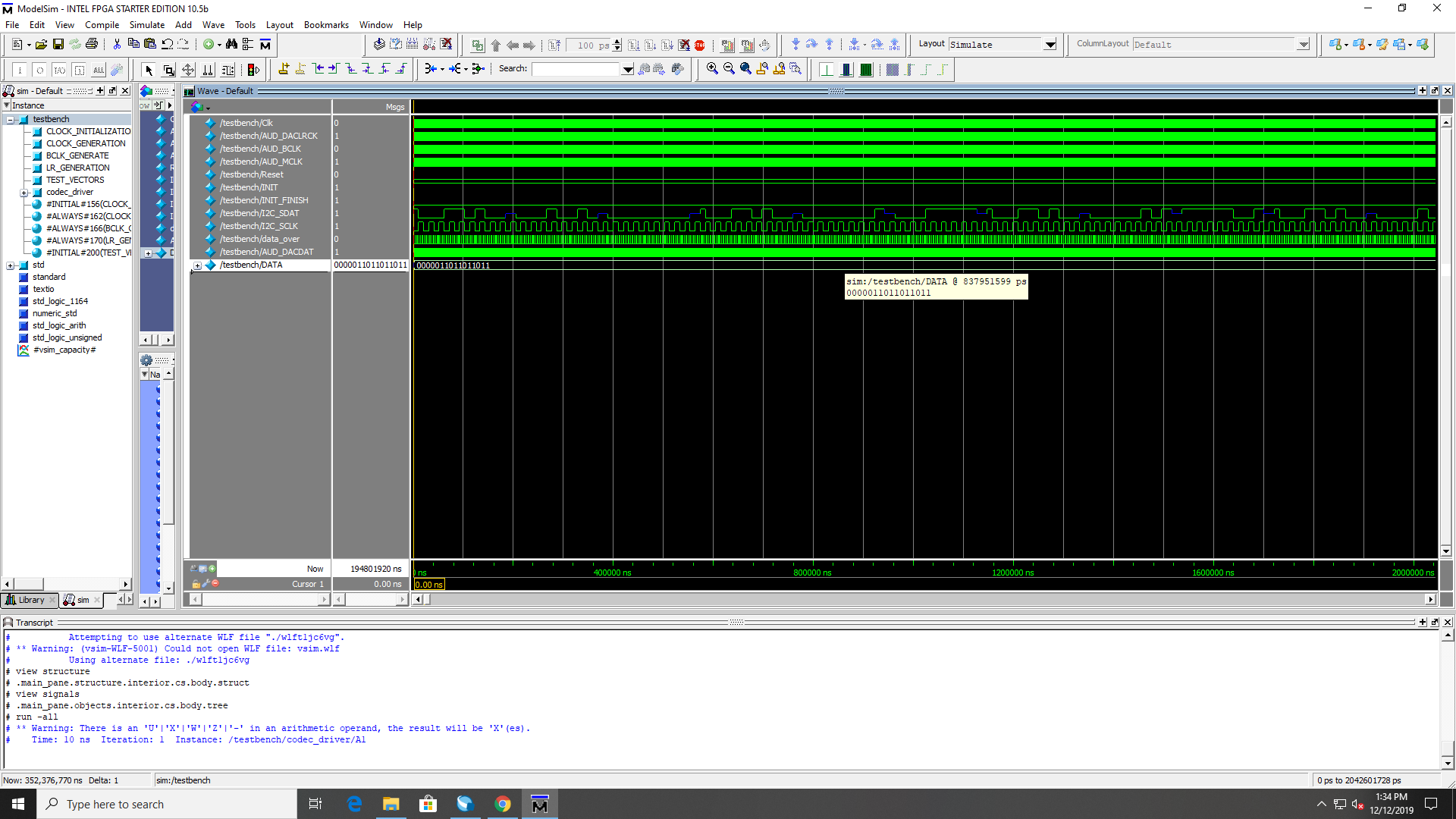
**Sampler Address Control Unit:**

*This simulation shows the Sampler Address Control unit being set to 0x08c0 by a keycode input, then incrementing addresses every sample clock cycle.*

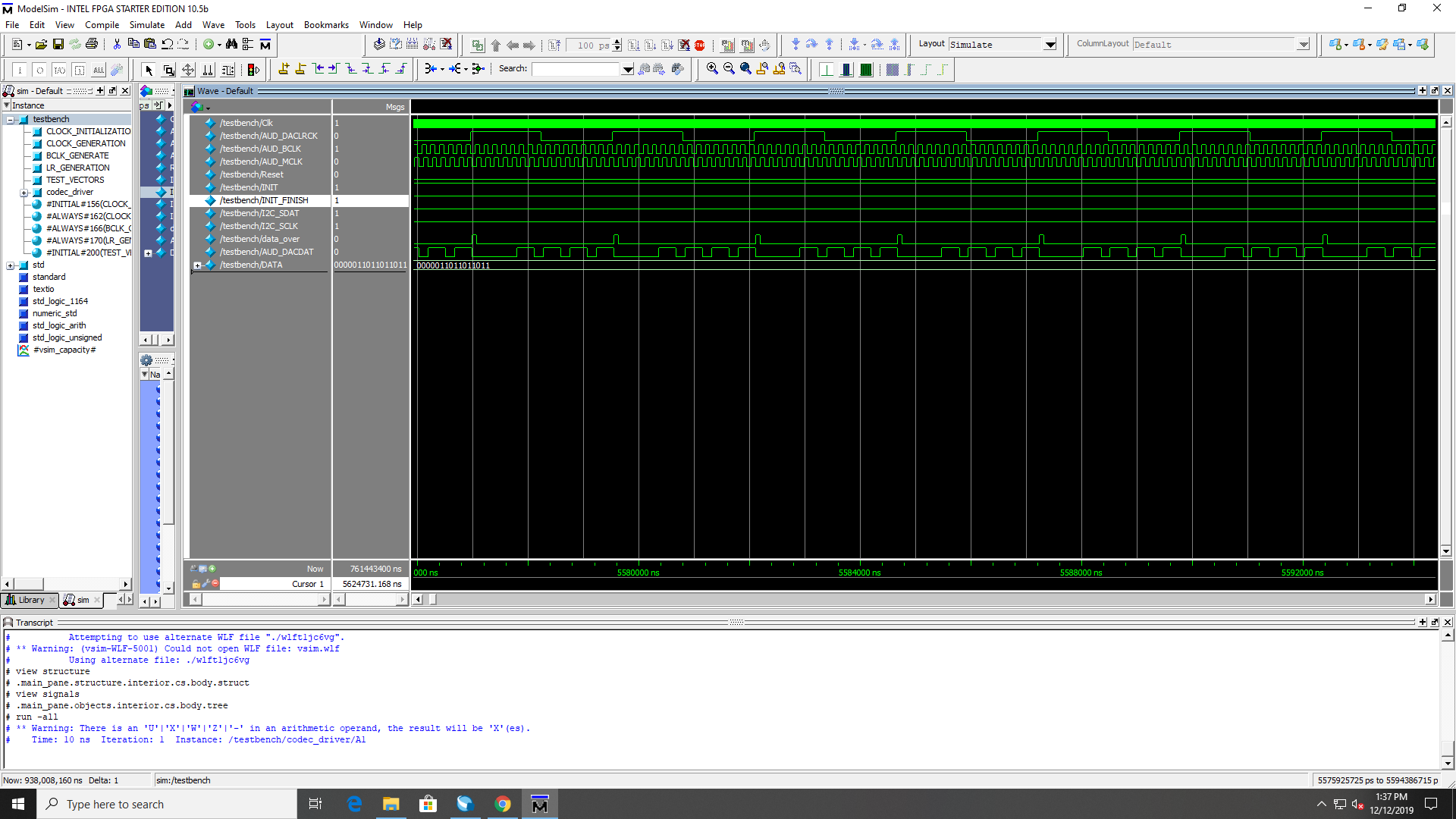
**Wavetable Synthesizer:**

*This is a simulation of the wavetable synthesizer being given the frequency required to play a C. The output should change every clock cycle.*

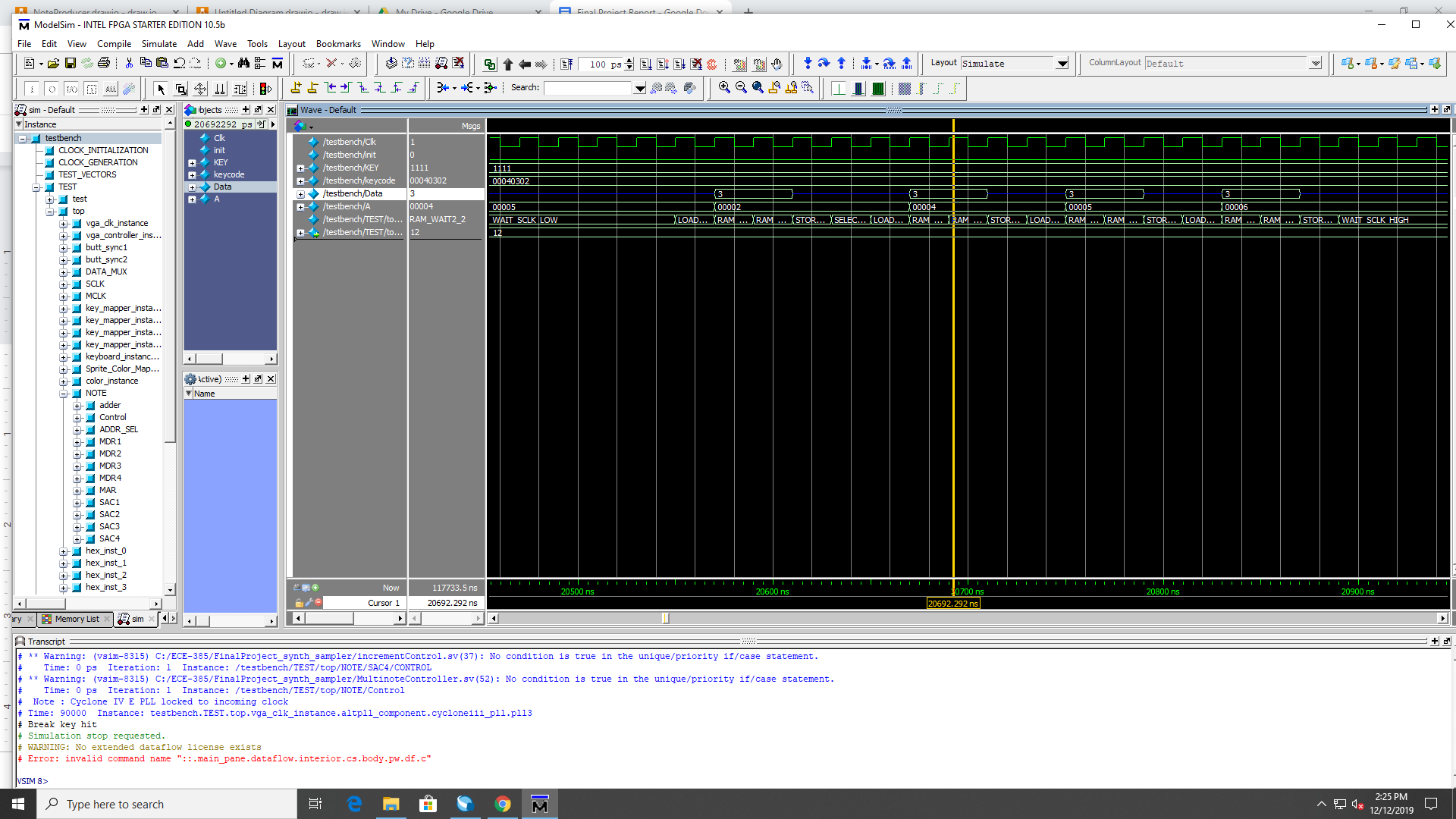
**Audio Driver Initialization Phase:**

*This simulation shows the functionality of the Audio Driver. The audio driver is initialized by setting INIT to 1, then several I2C serial data commands are sent serially via the I2C\_SDAT pin.*

**Audio Driver Sending Serial Data:**

*This simulation shows the functionality of the Audio Driver after initialization is done. The BCLK and AUD\_DACLRCK clocks are being mimicked in simulation since these are typically sent by the audio codec. The data value 0000011011011011 is sent serially, with the bit changing everytime the BCLK goes high.*

**Top Level SRAM communication:**

*This simulation shows the NoteProducer module, which sends and receives data from the on board SRAM. The data can be seen being transmitted. For the sake of testing, dummy address locations were made, and all the data values in these addresses are set to 3. You can see that all three values are added together and made to be twelve as intended.*

***Design Resources and Statistics Table:***

|  |  |
| --- | --- |
| LUT | 50,760 |
| DSP | 0 |
| Memory (BRAM) | 11,392 |
| Flip Flop | 3,613 |
| Frequency | 50 MHz |
| Static Power | 109.72 mW |
| Dynamic Power | 176.55 mW |
| Total Power | 456.82 mW |

***Future Improvements:***

Future improvements we would make to our project is to expand on the notes being played out of the sound driver as well as on the VGA display. Our project contains only one octave of a piano and we would like to be able to expand it to more than one octave, specifically having a full keyboard of sounds outputted. In order to fulfill this requirement, we would have to gather more samples covering notes of a full piano keyboard and store them into memory and access its data accordingly. We would also update the display by drawing more piano keyboard keys and expanding on the available sprites to represent a full piano keyboard worth of notes. Another thing we want to improve is that some of the piano samples and drum samples are slightly distorted. Specifically, the B-note on the piano sampler and the crash, ride, and the high tom drum samples sound distorted when outputted from the sound driver. The reason why they are distorted is because these samples are too loud that it messes up the SRAM process of extracting the data and sending it out to the sound driver. To fix this issue, we simply need to adjust the volume of the .wav file containing all the samples and re-uploading it into the SRAM. Another improvement we also want to make was that we did not have enough time implementing multiple notes being displayed when multiple keys were being pressed by the user. To implement this action, we need to allocate more space on the display to display up to four notes based on the four possible keyboard key presses. We would also need to add more sprites in the sprite table to represent all the notes, especially if we implement a full piano keyboard. The last improvement we want to make is to fix an issue of occasional glitching occuring in the sound when two keyboard key presses are simultaneously pressed and you release one key but still hold down the other. The reason why this glitching is occurring is because when keys are pressed simultaneously pressed and one is released the keycodes representing the keys being still pressed are all moved down a spot when the entire thirty-two bit keycode data is being sent. Our hardware architecture reads from the specific keycode slots, so there is a brief glitch happening when the key presses are moved down a slot. To fix this behavior, we believe we can implement a software component (C code) and write code that holds a dummy value to avoid having the keycodes moving down a slot in the keycode data. Overall, these are some of the improvements we would like to make to our final project to enhance and clean its functionality. Outside of these improvements, we could also add more instruments and adjust the VGA display to show drum pads for the drum sampler or a guitar for a guitar sampler. There are many improvements we can make and we are hopeful we can continue expanding on our project’s functionality for years to come and possibly opening it up to other people in the process.

***Conclusion:***

Overall, our final project was a success. We not only were able to complete our original project design idea from our proposal but also expand and build more functionality on it. Our original design idea was comprised of constructing a sawtooth wave synthesizer as well as adding a keyboard visual. Upon pressing certain keyboard keys, the DE2 board will output specific audio tones and the display will highlight the note being played as well as display the note. We expanded on this idea immensely by adding a piano and drum sampler as well as making all the notes polyphonic. Through this implementation, users will be able to select what type of audio they want to be outputted as well as have a nice visual that will highlight the keys being played. During our time making the project we did come across several challenges and hurdles. Specifically, the first major hurdle was communicating with the audio driver on the DE2 board and being able to output audio. Another challenge was constructing a state machine that would be able to fetch and extract data from specific memory locations in SRAM. These memory locations would hold the corresponding .wav file data used in our piano and drum sampler. For the synthesizer, we needed to make sure we were polling enough samples and storing them in a wave-table based on the correct clock. For the graphic interface, we crossed various hurdles as far as properly drawing the keyboard and highlighting the specific keys we want pressed, especially when there is more than one being key pressed. Our journey in building this project definitely gave us more insight and knowledge of the DE2 board and SystemVerilog itself. Overall, we had a lot of fun making this project as the idea we chose was something both of us found fun and intriguing. The project challenged us in new and unique ways as most of our labs did not involve any digital signal processing. We are very proud of our final project.