DIGITAL AUDIO WORKSTATION

Final Project Group Report

ENSC 452 Spring 2021
Group 7
April 16th, 2021

Nitish Mallavarapu & Kiel Henkelman
nmallava@sfu.ca, khenkelm@sfu.ca

Contents

Introduction

In this class, groups were tasked with building a firmware project on the Xilinx Zedboard. The project had to use two of the board's processors, but the idea and proposal were otherwise essentially left open as group choice. Groups were to divide the project into twelve milestones, six for each group member, to be completed by regular intervals over the period of the course.

Our group decided on creating a digital audio workstation (DAW), which is a tool for composing or transcribing audio, such as music. Specifically, it was decided that we would make a DAW strongly inspired by the "Music mode" of the 1992 Super Nintendo Entertainment System game *Mario Paint*, known as *Mario Paint Composer*. This selection was made for several reasons:

- It included audio and video components, which could be handled by separate cores.
- It had a high number of relatively small features, allowing for it to be more easily broken down into milestones.
- It was expected to require low enough computational resources to be able to run smoothly on the Xilinx Zedboard.
- It was judged sufficiently difficult for a course of this level.
- It was an idea that our group felt would be enjoyable to design and program.

The original Mario Paint Composer includes enough features to compose and play a simple chiptune, and that is the lead our project aimed to follow. At the beginning of the project, our group tabulated all the relevant features from a copy of the game, which we then used to construct our milestones. All together, our goals for the project were to allow the user to:

- Construct a song out of notes from scratch on a scale.
- Play back said song.
- Add notes to the song across at least two octaves.
- Easily remove notes from the song.
- Switch between natural and sharp notes.
- Use at least two different waveforms as instruments simultaneously.
- Modify the tempo of the song.
- Toggle looping for the custom song, wherein the song will automatically repeat at the end.
- Create a lengthy song using multiple pages of notes.
- Alter volume for a given beat.

In this report, the process and results of this project are explored.

Background

Our project required the development of two primary components – audio and video. The output itself of these components was relatively simple as audio data can be sent directly out of the Zedboard via an audio jack, and video data can be sent directly out of the Zedboard via a VGA cable. The more challenging part of development was the production of the data being sent.

For this project, it is integral to understand the relationship between waveforms and sound. At a physical level, sound manifests as a wave travelling through air. The frequency, magnitude, shape, and interval of the wave govern the pitch, volume, timbre, length, and location in time of the sound, respectively. For a DAW, all these qualities of sound are extremely relevant, as they form a complete description of a note, and notes are the fundamental building blocks of music. Some terms can be elaborated upon:

- Pitch is what affects how "high" or "low" a note is. It is a function of the frequency of a wave.
- Timbre is the quality of sound that humans use to tell two instruments apart, even when they are at the same volume and pitch. It is a function of the shape of a wave.

Audio is interpreted at a hardware level as a steam of signed 32-bit values. Each individual value can be interpreted as the level of air inflow our outflow from the speaker at that time, with numbers above zero representing outflow and numbers below zero representing inflow. The result of this is that sound data can be generated by a function, manipulated to account for qualities of the sound, then streamed to the speakers.

It is necessary to explain how "pitch" works more thoroughly. In Mario Paint Composer (and Western sheet music, more generally), the full infinite range of possible pitches is divided into discrete units, separated by intervals. Musicians are not expected to use the frequency space within these intervals. Instead, they use one of about one hundred common pitches, which range from extremely low to extremely high[1]. The gap between two adjacent pitches is known as a "semitone". Each "doubling" of frequency spans exactly thirteen semitones. Any given span of thirteen semitones is called an "octave". The table of the relative frequencies of any given octave is shown below:

[1] One Hundred is an estimate. There is no official set of "common pitches", but one hundred pitches span the approximate range of bearable human hearing, and one hundred and twenty would exceed the maximum range of human hearing

	Relative Frequency
12 Semitones Up	0.5
11 Semitones Up	0.5297
10 Semitones Up	0.5612
9 Semitones Up	0.5946
8 Semitones Up	0.6300
7 Semitones Up	0.6674
6 Semitones Up	0.7071
5 Semitones Up	0.7492
4 Semitones Up	0.7937
3 Semitones Up	0.8409
2 Semitones Up	0.8909
1 Semitone Up	0.9439
Original Pitch	1

This table is defined by the formula:

$$p = 69 + 12\log_2\left(\frac{f}{440\text{ Hz}}\right)$$

where *f* is the frequency and *p* is the indexed pitch value.

With about two octaves, a user can transcribe most songs and find they sound good, even if the original song uses more than two octaves. This is because pitches separated by exactly one octave have similar qualities of resonance with other notes and can be used as rough "sub-ins". This is the rationale for the two-octave minimum in this project.

Traditionally, when visualizing music on a scale, an octave is divided into only eight discrete pitches, rather than thirteen. Then, if the remaining five pitches are needed, a "sharp" (♯) symbol is affixed, which raises the pitch by one semitone. The reasoning for this system is beyond the scope of this report, but it can be most easily visualized by looking at a full octave on a piano. The white keys are the eight pitches visible on the scale, and the black keys are accessible by applying a "sharp" symbol to the key immediately to the left.[2] Note that the non-sharp equivalent of a note is called a "natural" note.

[2] In musical notation, flat (♭) symbols also exist, which lower the pitch by one semitone. While this is useful, sharps alone are sufficient to cover all necessary pitches.

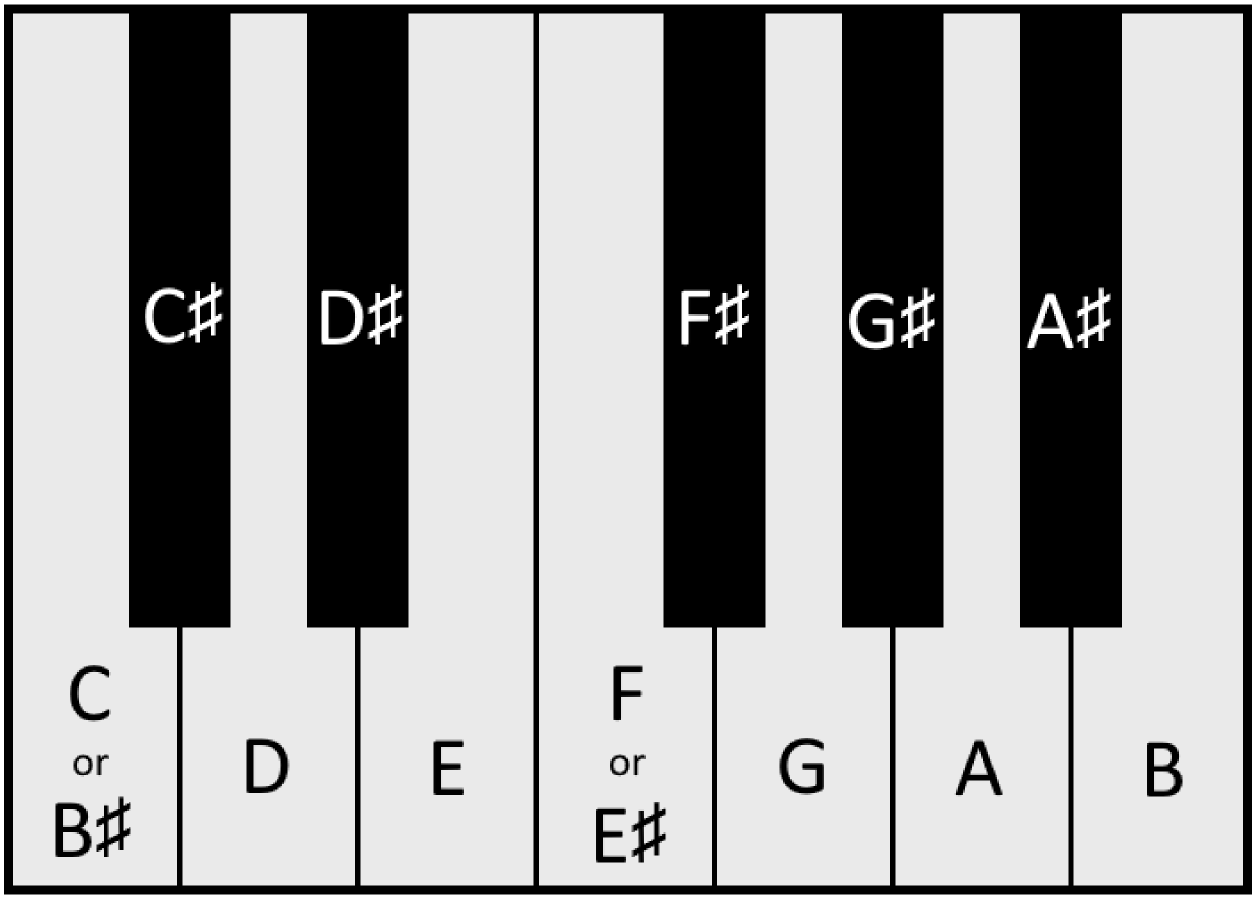


Figure 1: Octave on Piano

An additional term worth describing is "tempo". Tempo is the speed of the song, typically measured in "beats per minute". Note that the tempo of a song cannot be adjusted by applying horizontal stretching or squeezing, as this would also change the pitch of the song. Instead, the length of the beats themselves must be reduced by cropping their intervals.

Some might observe the absence of explanations for some additional properties of notes: attacks, decays, sustains, and releases. While a full explanation or implementation of these terms is beyond the scope of this project, it suffices to say that modifying these factors is not part of the original Mario Paint Composer DAW, nor was it implemented as part of our Mario Paint Composer-inspired DAW. In our project, attacks and releases are always instantaneous, and there is no decay. Notes also sustain for the full length of the beat, meaning that two identical notes adjacent in time naturally adjoin into one note.

System Overview

System Block Diagram

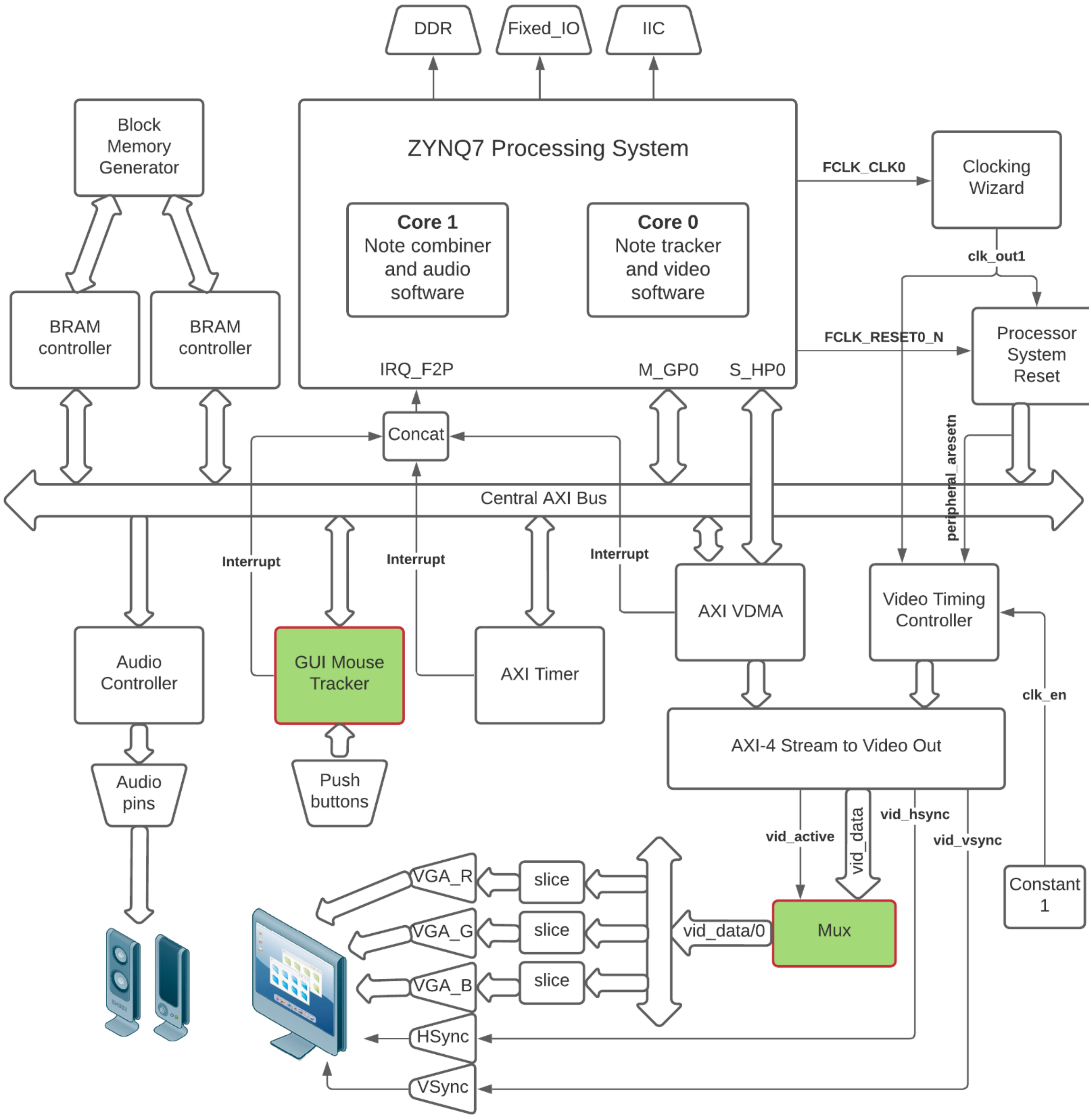


Figure 2: System block diagram (highlighted blocks are our custom designed IP cores).

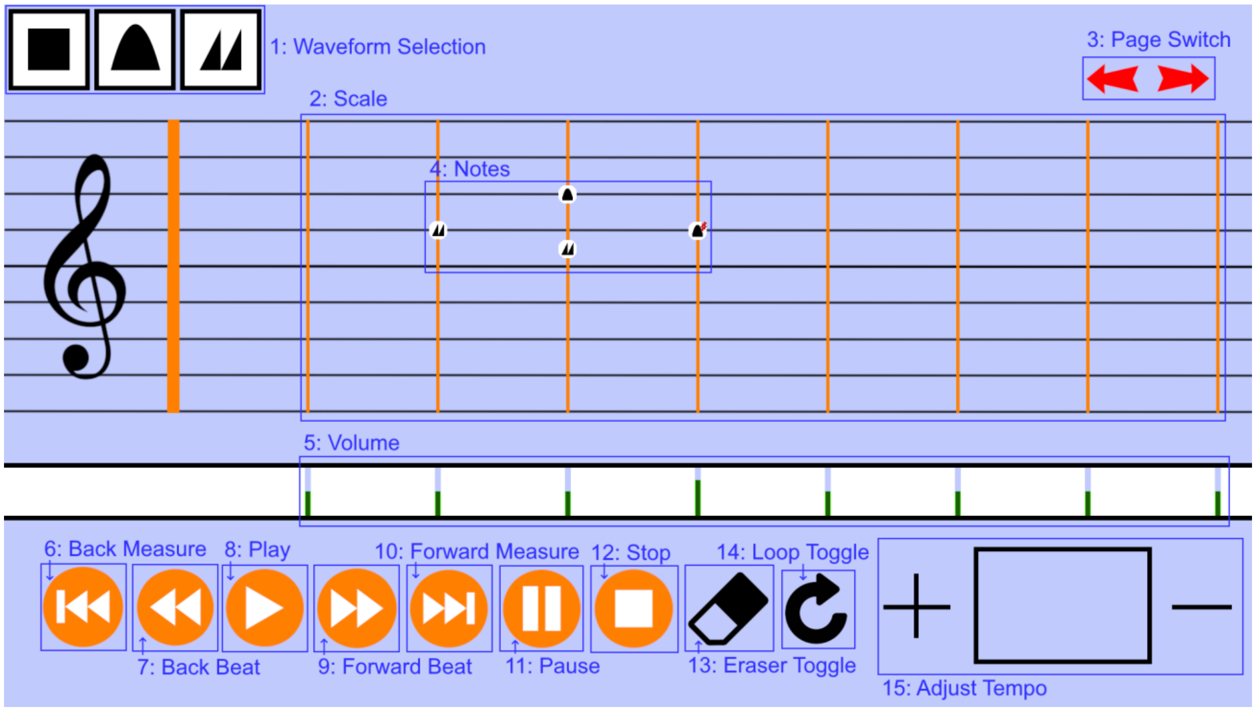


Figure 3: Interface Labels

<u>General</u>

While running, a cursor will be visible. The position of this cursor can be adjusted by the directional buttons on the Zedboard, and the cursor can be made to interact with the screen by pressing the centre button. Also visible while running would be a number for the tempo, a number for the page, and a pink current beat tracking "scroller" bar that sits over the presently selected beat.

<u>1: Waveform Selection</u>

By selecting one of these buttons, the cursor can be used to place notes that use waveforms of the specific type. From left to right, the waveforms are square, sine, and sawtooth. The currently selected waveform is encircled in red.

<u>2: Scale</u>

This is the region of the screen in which notes are placeable and is the primary workspace for building songs. Clicking between or on top of any rung on any of the beats, which are represented by orange bars, places a note.

3: Page Switch

Pressing the forward or backward arrows here allow the user to jump between pages, each of which is played in succession when the song is running.

4: Notes

Notes are the building blocks of songs. By clicking a note that is already placed, it can be toggled on-and-off between sharp and natural, which is indicated by a red sharp symbol or the lack thereof. The pitch of a note is visible in its height on the scale, and its waveform is shown as a symbol on the note.

5: Volume

The volume of a given beat can be adjusted between 5 values by repeatedly clicking the volume level below the beat. Like in Mario Paint Composer, the volumes of individual notes cannot be altered, only the beat itself.

6: Back Measure

Moves the scroller bar back by four beats.

7: Back Beat

Moves the scroller bar back by one beat.

8: Play

Plays the song from the scroller bar.

9: Forward Beat

Moves the scroller bar forwards by one beat.

10: Forward Measure

Moves the scroller bar forwards by four beats.

11: Pause

Stops the song from playing, keeping the scroller where it left off.

12: Stop

Stops the song from playing if it is currently playing and returns the scroller bar to the beginning of the song.

13: Eraser Toggle

Clicking this button toggles the eraser on/off. When toggled on, the eraser is encircled in red, and clicking on notes removes them.

14: Loop Toggle

Clicking this button toggles looping on/off. When this option is toggled on and the song is playing, the song will repeat itself immediately after the last note is finished, continuing indefinitely until the song is stopped, paused, or the loop is toggled off.

15: Adjust Tempo

Clicking the plus or minus buttons on either side of the tempo raises or lowers the toggle. The maximum and minimum values are 240 and 60 beats per minute, respectively.

Brief Description of IP Cores

ZYNQ7 Processing System

- Provides a wrapper for the processing system.
- Comes from the Xilinx library .

Block Memory Generator

- Automates the creation of optimized block memories by leveraging built in knowledge about the architectural features of the device.
- Comes from the Xilinx library.

AXI BRAM Controller

- Allows for system master devices to communicate with BRAM.
- Is optimized for performance.
- Comes from the Xilinx library.

AXI VDMA

- Provides direct memory access between memory and AXI4-Stream type video target peripherals.
- Learned how to use this core through a YouTube tutorial.
- Comes from the Xilinx library.

AXI4-Stream to Video Out

- Converts AXI4-Stream interface signals to a standard video output interface with timing signals.
- Learned how to use this core through a YouTube tutorial.
- Comes from the Xilinx library.

Video Timing Controller

- Automatically generates video timing signals based on display resolution that is selected within the IP configuration menu.
- Learned how to use this core through a YouTube tutorial.
- Comes from the Xilinx library.

AXI Timer

- 32-bit timer/counter that can generate interrupts based on user specification.
- Comes from the Xilinx library.

GUI Mouse

- Always keeps track of the mouse's position and prevents the mouse from leaving the edge of the screen.
- Generates either mouse move interrupts or mouse press interrupts.
- Custom programmed with Verilog and using the AXI-4 Lite interface.

Processor System Reset

- Allows the user to customize their application by setting certain parameters to enable/disable features.
- Comes from the Xilinx library.

Clocking Wizard

- Configures a clock circuit to user requirements.
- Comes from the Xilinx library.

Audio Controller

- AXI interface to control data sent to the onboard audio DAC.
- Comes from "the ZYNQ Book tutorials".

Mux

- 24-bit multiplexer that either sends input data or all zeroes based on control signal.
- Custom programmed with Verilog.

Constant

- Supplies a constant value.
- Comes from the Xilinx library.

Concat

- Concatenates bits together.
- Comes from the Xilinx library.

AXI Interconnect

- Connects one or more AXI memory-mapped masters to one or more memory-mapped slaves.
- Comes from the Xilinx library.

Slice

- Used to split bits from a bus.
- Comes from the Xilinx library.

Outcome

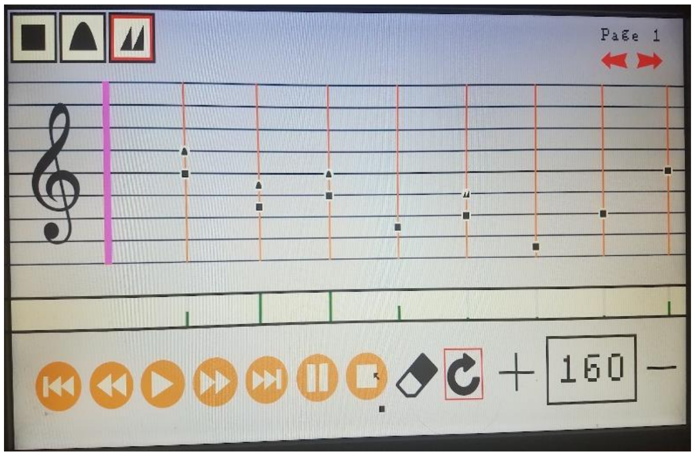


Figure 4: Screen capture of the finished product

We were able to successfully run a demonstration that achieved all our goals that we laid out in our milestones, although there were some issues with our graphics processing, some of which affected the audio playback since there is a direct correlation.

The GUI interface works almost exactly as planned, the only problem is that when the pink scroller is being updated, the mouse freezes for a short period of time while the graphics for the scroller are erased and redrawn. This freezing occurs even more rapidly at higher tempos. The root of the problem comes from the fact that the AXI Timer interrupt (which controls when to update the pink scroller) takes priority over the mouse interrupts. It would not have been a good idea to have it the other way around because if the user continuously moves the mouse, then the pink scroller wont update at all. A suggestion to get rid of the periodic freezing would have been to reroute the mouse movement interrupt to a different core from the timer interrupt, that way both updating the scroller and moving the mouse could have been handled at the same time.

Another minor issue is that the background always has a tiny black box[3] that is left undrawn. I am assuming that this is caused by a memory collision between the buffer that is being drawn and some other piece of data. Some further debugging is necessary to get rid of this eye sore, but due to time constraints we decided to ignore it.

By far the biggest issue we had was with how fast a new page could be drawn. In our demo video, you can see when a page changes, all the notes are drawn 1 by 1; it does not happen instantaneously. Because of this, the scroller hangs on the last note of the previous page, and you can notice that the song becomes off beat. I have come up with two possible suggestions: the first one sacrifices image quality but saves on memory, and the second one saves image quality but sacrifices memory.

[3] You can notice the black square underneath the stop button in Figure 4.

1) *Lower the resolution that we chose.*

   We decided on a 1080p resolution which meant that a buffer of 1920*1080*3 bytes was needed. We could have easily downgraded to a 720p or even 480p resolution which would have decreased the buffer size by 62% and 85% respectively. This would have made traversing the buffer and sending the pixel data to hardware much faster.

2) *Have 2 extra buffers with both the next page and the previous page data pre-written to them*.

   We only used 1 buffer which meant that we had to first erase the old pixels from the buffer and then draw the new pixels which was the main cause of the delay. With the extra buffers, we could have just pointed to which buffer to use based on what action just happened. This would have also required some overhead to keep track of which buffer was being used and which buffer could be redrawn to.

If we had to choose one of the two suggestions, we would select the latter. Since our application is quite small, memory constraints were not a real issue. One buffer was ~6mb, so 2 additional buffers would have been an additional ~12mb. Considering we have 512mb of DDR3 memory on the Zedboard, we would rather sacrifice memory over image quality.

Another problem we had was with audio quality after combining multiple notes. When more than one note was played at the same time (a "chord"), it seemed that a significant degradation in the quality of the audio took place. To the ear, this degradation sounded like the notes were played on top of very coarse "buzzing". It was suggested on project completion that this was likely a result of the natural "jaggedness" (i.e., non-differentiability) of square waves and sawtooth waves, which do not smoothly change between values all the time. Then, this effect would be a natural consequence of the waveforms we chose and was not a bug as we originally assumed.

Description of Blocks

Hardware Modules

ZYNQ7 Processing System

- v5.5 used from the Xilinx library
- Reconfigured settings
    - Enabled S_AXI_HP0 interface
    - Enabled I2C0 IO peripheral
    - Set FCL_CLK1 to 100MHz
    - Enabled FCLK_CLK1 and request 10MHz for the master clock of the audio codec
    - Enabled PL-PS interrupts

Block Memory Generator

- v8.4 used from the Xilinx library
- Reconfigured settings
    - Changed to dual port RAM
    - Turned off safety circuit

AXI BRAM Controller

- v4.1 used from the Xilinx library
- Used 2 different BRAM controllers, 1 for each ARM core
- Reconfigured settings
    - Had to change each from dual port to single port

AXI VDMA

- v6.3 used from the Xilinx library
- Followed this youtube tutorial: https://youtu.be/cADto95fV7Q
- Reconfigured settings
    - Set the stream data width to 24 bits
    - Disabled the write channel since we preloaded all sprite channel and no on the fly writing was needed
    - Had to allow for unaligned data transfers

AXI4-Stream to Video Out

- v4.0 used from the Xilinx library
- Followed this youtube tutorial: https://youtu.be/a6OXozk1Ix8

Video Timing Controller

- v6.2 used from the Xilinx library
- Followed this youtube tutorial: https://youtu.be/_o4FbVFLbuw
- Reconfigured settings:
    - Chose 1080p option to auto generate the proper timing signals

AXI Timer

- v2.0 used from the Xilinx library
- Followed Exercise 2D from "The Zynq Book Tutorials" to learn how to use the timer IP core

GUI Mouse

- Architecture of GUI Mouse is all custom programmed in Verilog
- Used the following youtube videos to learn about creating custom user IP cores with the AXI-4 Lite interface:
    - https://youtu.be/I0eu_Y3pMmM
    - https://youtu.be/meQcwzC4Vtk
- Inputs:
    - Slave AXI interface
    - 5 push buttons
- Outputs:
    - Interrupt signal
- Keeps track of the mouse position, prevents the mouse from scrolling off the screen, and generates interrupts
- 32-bit position register
    - [31:16] → y position
    - [15:0] → x position
- 2 interrupts can be generated
    - Mouse press
    - Mouse move
- Architecture includes 2 different versions of debounce logic
    - Single pulse
    - Continuously high

Processor System Reset

- v5.0 used from the Xilinx library

<u>Clocking Wizard</u>

- v6.0 used from the Xilinx library
- Reconfigured settings:
  - Set output clock to 148.5MHz which was required for the pixel clock
- This is the operating clock of the whole system

<u>Audio Controller</u>

- Used IP repository supplied in Exercise 5A from "The Zynq Book Tutorials"
- Followed the rest of Exercise 5 to learn how to use drivers for the core

<u>Mux</u>

- Architecture of Mux is custom programmed in Verilog
- Inputs:
  - 24-bit value
  - control
- Outputs:
  - 24-bit input value or all 0's depending on control signal

<u>Constant</u>

- v1.1 used from the Xilinx library

<u>Concat</u>

- v2.1 used from the Xilinx library

<u>AXI Interconnect</u>

- v2.1 used from the Xilinx library

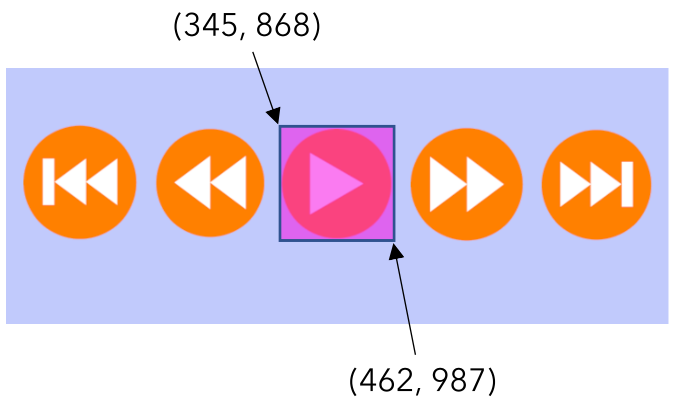
<u>Slice</u>

- v1.0 used from the Xilinx library

Software Modules

GUI

The user interface was programmed with a simple "hitbox" method. Within the mouse press interrupt handler, which only gets called when the center button is pressed, we first retrieve the x-y coordinates of the mouse[4]. Then using a series of if statements we check if the mouse was clicked at a position on the screen that has a predefined action. As an example, here is a short code snippet of how the play button works:



```
if((yPos > 868) && (yPos < 987)) {
        if((xPos > 345) && (xPos < 462) {
                //action for play button
        }

}
```

Figure 5: Example of how the play button is programmed using a hitbox

The same method was followed for all other controls as well as the actual placing of notes.

Note Tracker

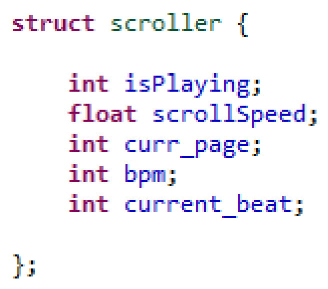
```
struct note {

    int isUsed;
    enum base_pitch m_tone;
    enum accidental m_accidental;
    enum timbre m_timbre;

};
```

Figure 6: Code snippet of note struct.

This module keeps track of all the notes that the user has placed throughout all the pages. It consists of a note struct, a 3D array of note structs, as well as functions to manipulate the 3D array. When a grid position on the staff is clicked on by the mouse, the object at the corresponding position in the 3D array is retrieved and updated based on whether the current action is placing a note or erasing a note. To successfully accomplish this, the note tracker must also convert pixel positions to beat positions for storing the notes at the appropriate position in the 3D array.

[4] Top left of the screen is (0,0) and bottom right of the screen is (1919, 1079)

Note Scroller



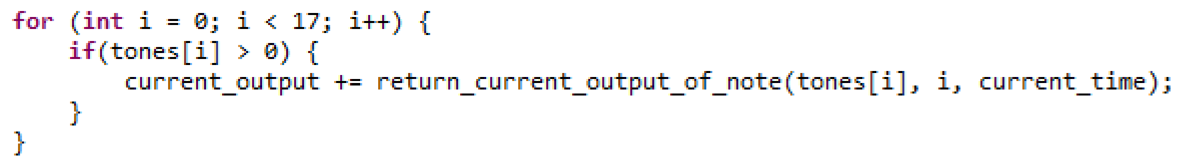
```
struct scroller {

    int isPlaying;
    float scrollSpeed;
    int curr_page;
    int bpm;
    int current_beat;

};
```

Figure 7: Code snippet of scroller struct.

This module is responsible for 2 things: The redrawing of the pink scrollbar that you can see in Figure 2 and sending the notes that need to be played to Core 1 where the actual note combination and audio output happen.

After initialization, the bpm member variable is used to calculate what value to load into the AXI Timer hardware module. The AXI timer is then configured to expire every time the counter rolls over and reset back to the initial load value. This causes the generation of an interrupt every time the scroller needs to move forward one beat and the graphics are subsequently updated. The notes at the new beat are then written to memory for further use by the note combiner module in Core 1.

Note Combiner



```
for (int i = 0; i < 17; i++) {
    if(tones[i] > 0) {
        current_output += return_current_output_of_note(tones[i], i, current_time);
    }
}
```

Figure 8: Code snippet of notes being merged

This module is responsible for delegating the processing of notes being played and combining them in real-time. It checks which notes are placed in each of the seventeen beats that are currently being played, and retrieves different information depending on the type of notes. The note combiner is also responsible for handling volume, because volume is applied to an entire beat rather than individual notes.

The Note Combiner is possibly implemented differently than a person would expect. For other software, the length and the position in time of a note would be included in the note's metadata. While the position in time of notes *is* kept track of for this program as well, it is not necessary for the note combiner to understand. In fact, the note combiner simply starts playing notes when they are received, and stops playing them when a new note is received (including "empty" notes), or the song is paused. This method was able to implement what was intentional from beginning – that successive identical notes blend into each other without stopping. This allows notes longer than a beat to be played, and cuts down on overhead.

Graphics

The base of the graphics module was found at the following opensource GitHub repository found here. It was designed for a grid-structured game called "Connect 6", and because our musical staff also had grid properties, I decided to use it, but some modifications of the source code was still necessary to fit our needs.

The first thing we added was erasing pixels. Technically speaking, we are not actually erasing pixels but rather redrawing the default background pixels. The first version of this erase function would erase the whole screen, which required static parts of the GUI to have to be redrawn. So, we updated the function to allow us to only erase a specific portion of the screen. This allowed us to leave the static parts of the GUI the way they were, speeding up the overall drawing. We then also had to add a function that could draw images positioned from the center of the image rather than the top left corner. The last thing we added was a helper function to draw all the notes on a page. Finally, we ended up removing a lot of the base code since they were meant for the "Connect 6" game.

All of the software that setup the AXI VDMA, AXI4-Stream to Video Out, and Video Timing Controller was also found from a GitHub repository found here.

Audio

The foundation of the audio portion of the project is the audio.c file from the audio tutorial, provided for this ENSC 452 course. The parts of this module that interface with the board or provide setup have generally been left unaltered, as no changes were needed.

The adventures_with_ip.c file from the audio tutorial remains in name only, but no code originally included in the tutorial is still included in this file, as the audio configuration functions were moved to main.c. Instead, adventures_with_ip.c includes the unique functions of each individual waveform, which return results in time for their respective waves, including grabbing a modifier for pitch from a lookup table. When playing sine waves, a sin() function is not used – instead, the current value is calculated using a more simple linear function, pulled from a table of linear functions that approximate a sine wave.

Description of Design Tree

Zipped Folders

vga_practice

This folder includes the primary Vivado project file, and all necessary components of the Vivado part of the project.

vga_software_practice_system

This folder includes the information linking the two cores of the system together. It is the "system" folder of the Vitis project.

vga_practice_hardware

This folder includes the platform data for the Vitis project.

vga_software_practice

This folder includes the Core 0 data in its src folder, as described in "Vitis Files" below. It is one of the two "application" folders.

note combiner

This folder includes the Core 1 data in its src folder, as described in "Vitis Files" below. It is one of the two "application" folders.

Vitis Files

main (Core 0)

This is the first file loaded. It includes the setup, the feedback loop and interrupt handlers, the handling of cursor movement and clicking, and the delegation of all other unincluded tasks. It is also the file that links and communicates with the second core.

- note_tracker

  This file includes helper functions for adding and removing notes, retrieving note positions, and modifying tempo, as well as all of the scrolling action code

- video

  This file includes the low-level setup of the video portions of the project.

- platform

  This file, provided by Xilinx, is used for board initialization.

  - platform_config

    This file contains some definitions provided by and used by Xilinx.

- bgData

  Image data of the background

- scroller

  Image data of the scroller bar

- selected_eraser_pic

  Image data of the eraser toggle when selected

- selected_repeat

  Image data of the loop toggle when selected

- selected_square

  Image data of the square waveform toggle when selected

- selected_sine

  Image data of the sine waveform toggle when selected

- selected_sawtooth

  Image data of the sawtooth waveform icon when selected

- graphics

  This file includes many helper functions for drawing elements of the screen, including partial drawings and text elements. It also includes specialized functions for drawing notes, volume and the cursor.

  - font

    This file, from the Connect 6 project, includes the data for the tempo and page number text displays.

  - natural_noise

    Image data of a natural noise note (unimplemented)

  - natural_sawtooth

Image data of a natural sawtooth note

- natural_sharp_sawtooth

  Image data of natural and sharp sawtooth notes pair (unimplemented)

- natural_sharp_sine

  Image data of natural and sharp sine notes pair (unimplemented)

- natural_sharp_square

  Image data of natural and sharp square notes pair (unimplemented)

- natural_sine

  Image data of a natural sine note

- natural_square

  Image data of a natural square note

- sharp_sawtooth

  Image data of a sharp sawtooth note

- sharp_sine

  Image data of a sharp sine note

- sharp_square

  Image data of a sharp square note

- volume_high

  Image data of a volume bar at fourth level

- volume_low

  Image data of a volume bar at second level

- volume_medium

  Image data of a volume bar at third level

- volume_very_high

  Image data of a volume bar at fifth level

- volume_very_low

  Image data of a volume bar at first level main (Core 1)

This file is responsible for receiving information from core 1 and the primary loop of playing audio. It also includes some of the setup for the audio portion of the project.

- audio

  This file, provided as part of the audio tutorial, includes the remainder of the setup of the audio portion of the project.

- platform

  This file, provided by Xilinx, is used for board initialization.

  - platform_config

    This file contains some definitions provided by and used by Xilinx.

- adventures_with_ip

  This file includes helper functions for playing specific waveforms, and interpreting data received from Core 0. The name is a remnant from the tutorial, but no elements from the tutorial are included in this file.

  - sine_storage

    Once contained data for sine wave approximation (unimplemented, instead included as part of adventures_with_ip)