Final Project Report

Music Synthesizer with playback on FPGA

Minseob Shin & Kevin Peercy

Introduction

Functionality

Our project intends to act as a piano synthesizer and playback application. We have 3 tracks to enable playing notes on different levels and so that the user can add and remove portions of their piece seamlessly. The piano is reflected on the keyboard, with the natural white notes using the middle row of keys and the black sharps/flats being placed accordingly above. 2 Octaves from C3 to C5 are available in our system, with a designated octave switching key to enable this function. Playback can be limited in time to anything less than or equal to 32 seconds of recording. Lastly there is both an edit and playback specific mode which allows you to add/remove notes, and play only what is stored already respectively.

How the design builds on top of the basic 6.2 Lab code

The previously completed basic 6.2 Lab code has provided us with a solid foundation for the keycode fetching mechanism and the basic drawing functionality. We were able to modify such pre-implemented code to account for several more keys, adding corresponding functionalities to them. We also added BRAM to properly store the enabled notes at the given time up to 30 seconds. We entirely re-did the display in order to create a user-friendly interface that resembles the physical piano.

Written Description of the system

Written description of entire system

Taking the completed Lab 6.2 code, the I/O (keyboard input) of our design uses the GPIO concepts to communicate with the MAX chip that is automatically receiving USB input. The GPIO allows our MicroBlaze to now send SPI signals, and this component connection is making use of the USB pins to take in the keycodes to our MicroBlaze system. Through such memory-mapped I/O, we first receive the user pressed keycodes. Then we use those received keycodes to simultaneously display the notes as being played on the display, store the enabled keycodes in BRAM, and enable notes to output the correct audio in real time. Such was achieved by having multiple clocks with differential frequencies and by putting them together accounting for the latencies, we were able to make the system seamless. Extraneous functionality and graphics components were more additional and for-fun additions with octave changes/tracks and little ROM designs to represent them.

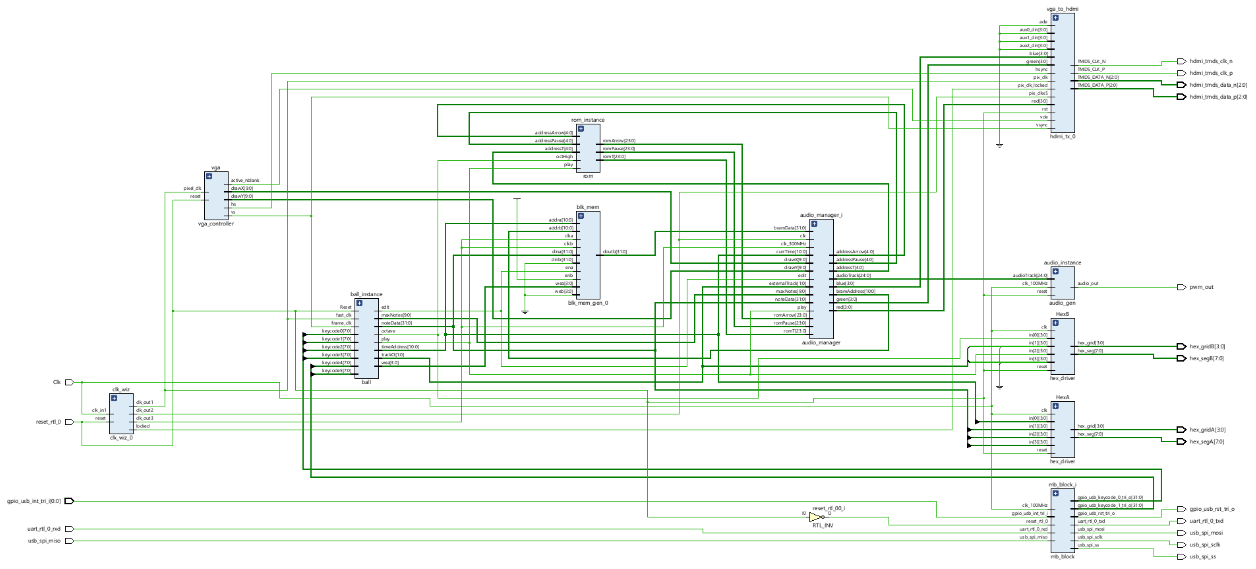
Describe the algorithm used to draw/store/play the notes

Storing the notes: Each note played was represented by 2 bits, an octave and an on/off bit for that specific time frame. We store those 2 bits for every possible note we can play, so 26 bits right there every single 20th of a second, which represents our "polling frame". BRAM was an obvious choice for this, even though accessing that data is highly restricted, due to the fact that we could store all 640 polled cycles that we decided on, as drawing that would be one pixel width per period. This also allowed for 32 seconds of music which was a solid amount by our determinations. 3 tracks meant that we had 3 times the registers that we would use for one full collection of notes, which meant 1920 total addresses with 26 bits each. Tracks 2 and 3 just start 640 addresses after the one prior.

Drawing the notes: Using the drawX and drawY signals, we just check the BRAM for which drawX address we are on. If we are on the 3rd horizontal pixel, the data for that note (whether it is on or off) will be in the 3rd address of BRAM. The most challenging part of this project, graphics wise, was that we had to check every track for every single pixel we are drawing, because we have to draw the specific track color. This was tough, as BRAM only has one line of access, and the HDMI output must maintain 60Hz, which meant in every pixel clock, we had to READ the BRAM 3 separate times, which as we know from lab 7 has a 2 clk cycle delay. We were forced to optimize the clk that was being sent to BRAM up to around 200MHz, which when compared to the 25MHz pixel clk would allow us to just make those limits. At that point, we just draw the colors associated with the track and this allows the visual tracking of everything being played.

Playing the notes: We decide when to generate the note by following the currently playing note, represented by the vertical bar on the screen. Each frame we store the enable bit for each of the 25 tones that should be generated for that 20th of a second. Each note is generated by note_gen module, which uses a phase accumulator and sine table values to generate a digital sine wave at a frequency determined by a phase increment value. The amplitude of each note is supposed to be dynamically shaped by an adsr module to model the real piano keys being played and released. In the top module of audio gen, we instantiate note_gen module for all 25 notes in parallel. Then the outputs of all active notes are digitally summed in awareness of possible overflow and distortion. Once the digital signal is mixed, we use such a value to generate a PWM signal to output the audio.

Block Diagram



Module Descriptions

Module: mb_usb_hdmi_top.sv
Inputs: Clk, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd

Outputs: gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0]hdmi_tmds_data_n, [2:0]hdmi_tmds_data_p, [7:0] hex_segA, [3:0] hex_gridA, [7:0] hex_segB, [3:0] hex_gridB, pwm_out
Description: This is a top module of the processor. It overarches the entire logic of the synthesizer
Purpose: This module is used to instantiate submodules and IPs and help their communications through local logic variables and rightful port connections

Module: hex_driver.sv
Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: This module converts the values of inputs to hex values that will be displayed on the board
Purpose: This module is used to correctly display the desired value within the cpu

Module: VGA_controller.sv
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: This module contains a positive edge clock triggered flip-flop that updates the display pixel information
Purpose: This module is used to correctly update and store VGA

Module: ball.sv
Inputs: Reset, frame_clk, fast_clk, [7:0] keycode0, [7:0] keycode1, [7:0] keycode2, [7:0] keycode3, [7:0] keycode4, [7:0] keycode5
Outputs: [10:0] timeAddress, [31:0] noteData, [9:0] maxNotes, play, [1:0] track0, octave, [3:0] wea, edit
Description: This module checks for the key presses every 20th of a second and determines the internal logic state such as octave, play, enabled notes, and the current timeframe and corresponding address.
Purpose: This module is used to make the system responsive to the user action through controlling the internal system logic

Module: audio_manager.sv
Inputs: [31:0] noteData, [31:0] bramData, [10:0] currTime, clk, clk_300MHz, [9:0] drawX, [9:0] drawY, [9:0] maxNotes, play, [1:0] externalTrack, edit
Outputs: [10:0] bramAddress, [3:0] red, [3:0] green, [3:0] blue, [24:0] audioTrack
Description: This module handles the display of the system by correctly handling the timing issues utilizing the buffers and sets the value of drawX, drawY, rgb, and the actual audio data that combines the notes across multiple tracks
Purpose: This module is used to properly display the synthesizer system with the user-friendly interface that illustrates the current track, play/edit, notes that are assigned to the given timeframe, and the notes being played.

Module: audio_gen.sv
Inputs: clk_100MHz, reset, [24:0] audioTrack
Outputs: audio_out
Description: This module operates on 100MHz clock to continuously output the PWM audio according to

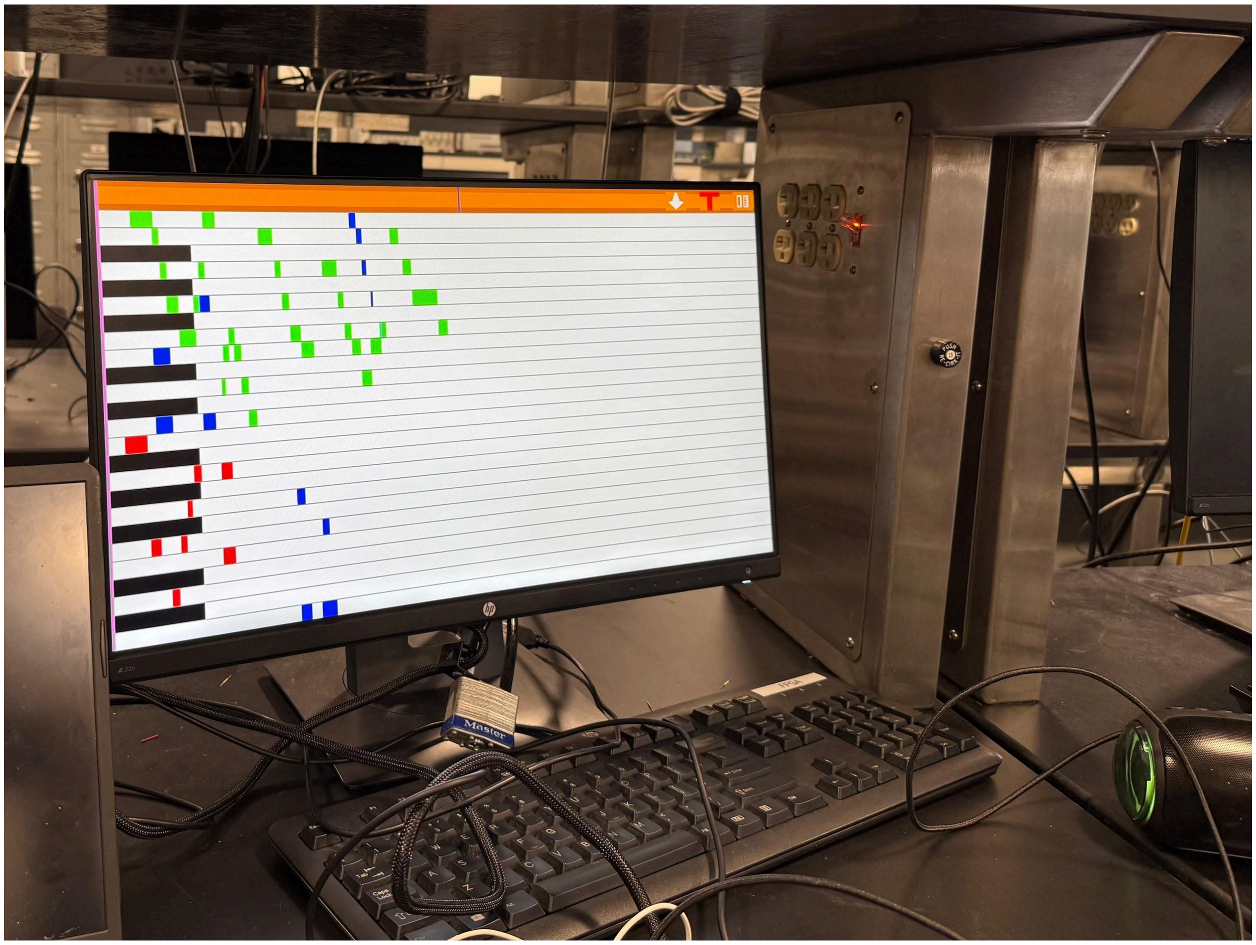
the audioTrack
Purpose: This module is used to instantiate and digitally mix the individual notes to output the overall audio

Module: note_gen.sv
Inputs: clk_100MHz, reset, note_on, note_off, phase_inc_fund
Outputs: [7:0] duty_cycle
Description: This module operates on 100MHz clock to continuously generate sine wave according to the phase_inc_fund
Purpose: This module is used to create an individual sine wave that corresponds to each note and manage them based on the note_on and note_off.

Module: asdr.sv
Inputs: clk, reset, note_on, note_off
Outputs: [7:0] envelope
Description: This module operates on 100MHz clock to manage the amplitude of the given note, by having the states with different meaning (attack, release, etc), it simulates the actual piano sound
Purpose: This module is used to manage the amplitude of each notes according to their states

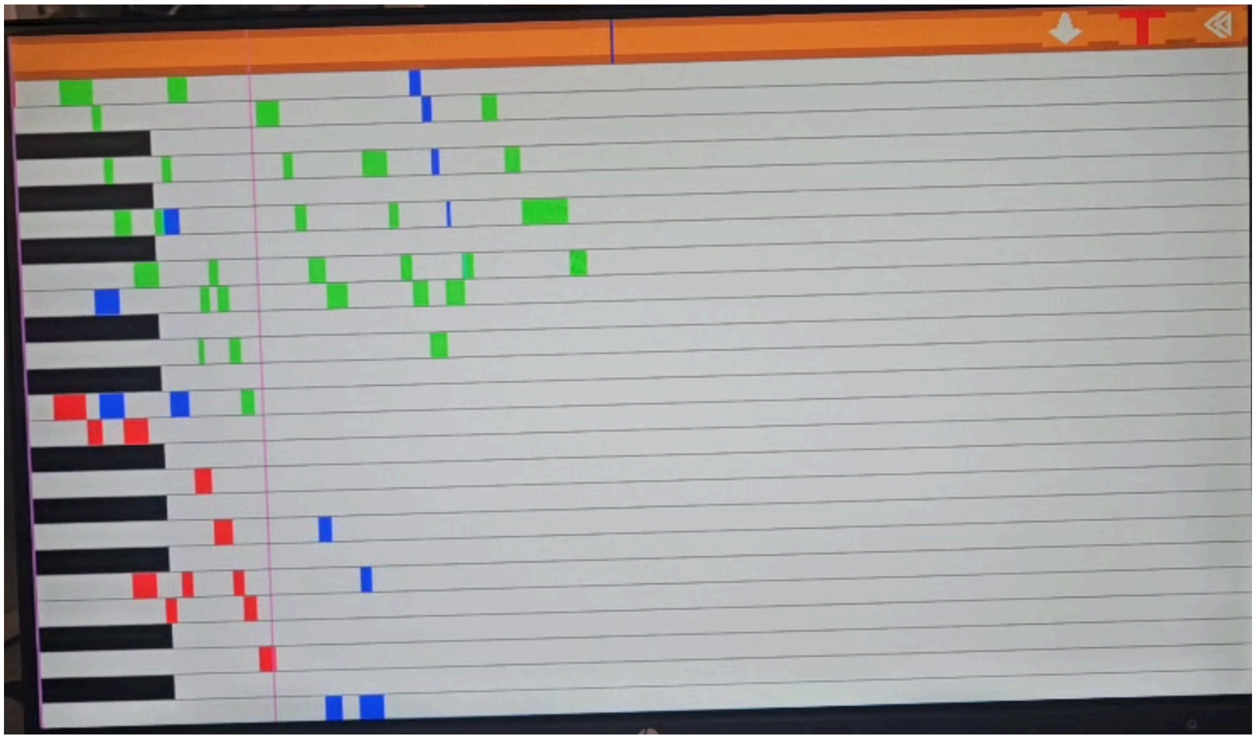
Module: rom.sv
Inputs: [4:0] addressPause, [4:0] addressT, [4:0] addressArrow, octHigh, play
Outputs: [23:0] romPause, [23:0] romT, [23:0] romArrow
Description: This module stores the bit data of the display components
Purpose: This module is used to store the pixel demonstration of track, play/pause, and arrow to assist in the drawing.

Simulation waveforms and Images

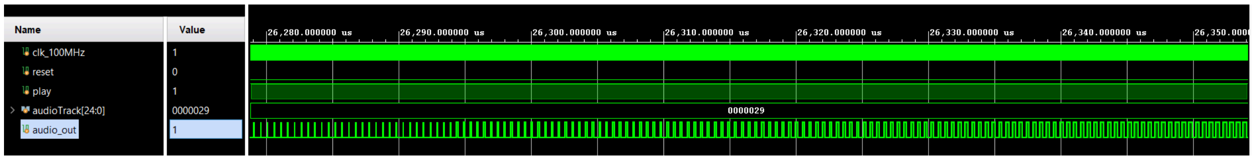


<u>Final version of the synthesizer in work</u>

The tracks can be seen by the different RGB sections, with the piano represented on the side showing what we are playing at that moment.



Mid-Playback - Pink bar showing the currently playing note, and blue bar at the top for the end of our playback. Down arrow reflects the current octave being low, and red T is the red track we are currently playing on, which we can see is being added to in real time on the screen. The play button is for us currently playing music.



Name	Value
clk_100MHz	1
reset	0
play	1
audioTrack[24:0]	0000029
audio_out	1

Changing duration of the PWM high could be observed. This complicated form encapsulates sin wave emulations and pulse phase tracking to make sure notes do not overlap.

Design Resources and Statistics

LUT	13786
DSP	53
Memory (BRAM)	22.5
Flip-Flop	4704
Frequency (MHz)	0.059

Static Power (W)	0.077
Dynamic Power (W)	0.630
Total Power (W)	0.707

Conclusion

Things that didn’t work

In the end, our piano note sounded fairly clean, with a moderate attempt at recreating the timbre and feeling of the piano. The attack aspect of piano notes, where the initial hit is louder, was also attempted in the implementation but did not come across as strongly as we might have wanted.

Things you learned

The level of freedom we had with what we felt like including and what looked good on the UI allowed some super creative progressions throughout the project. Due to the setbacks we faced with our original idea, this idea was more rushed than other projects might have been, but the motivation of having a super fun and super intuitive project was more than enough for us to make it solid in the end. We were on the same page throughout the whole process of assembling this application, and each test we had was exciting and demonstrated what we have learned in the class as well as sheer enjoyment.

Last thoughts?

This project was lovely, and an amazing ending to this course. I like the emphasis towards games and impactful demos with this project, and hope I can see other people’s projects somewhere as well. Thank you for the time reading all these and assisting us throughout the whole process, it has been an enjoyable experience.