FPGA Digital Audio Workstation

Julian Espada
Nathan Ramesh

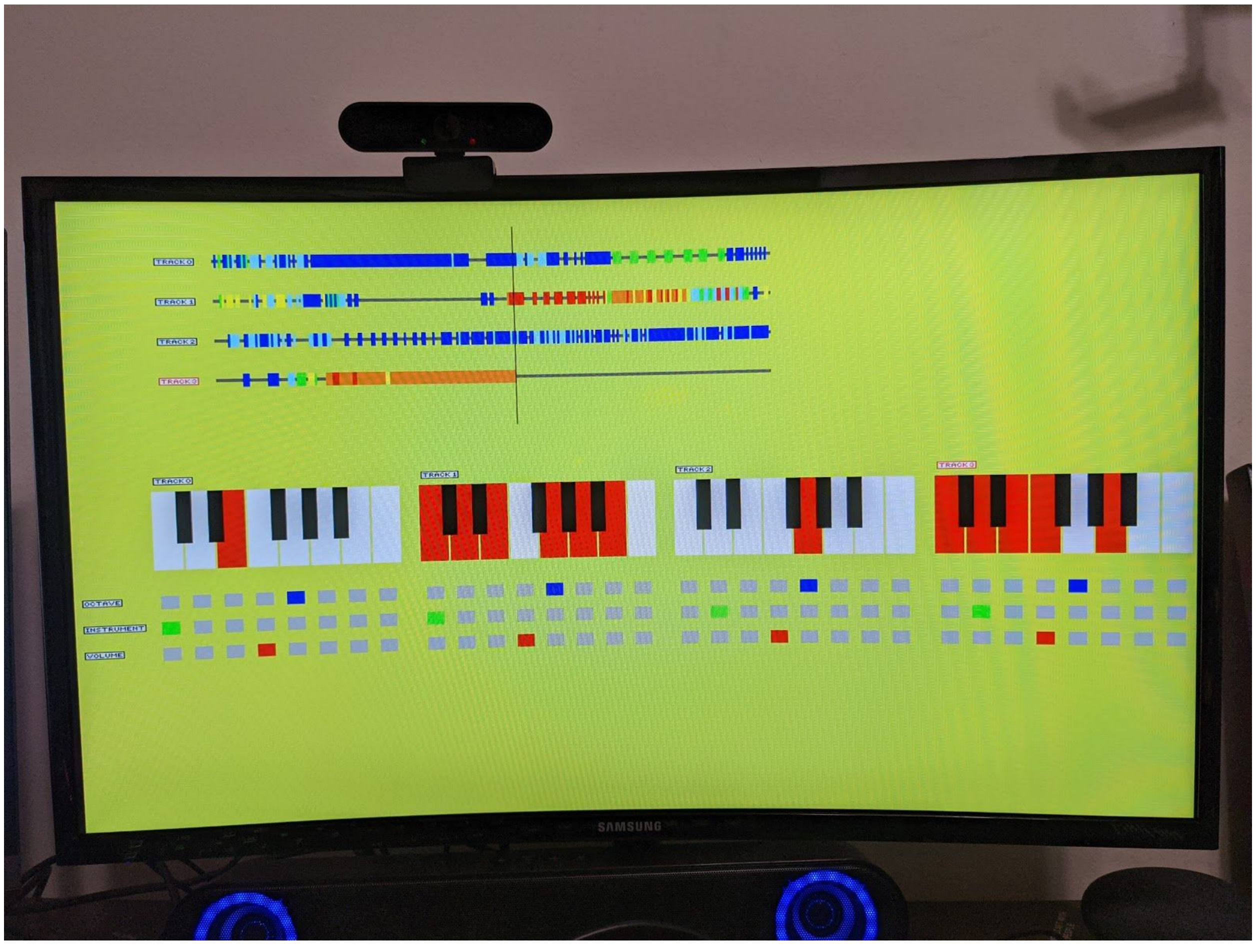
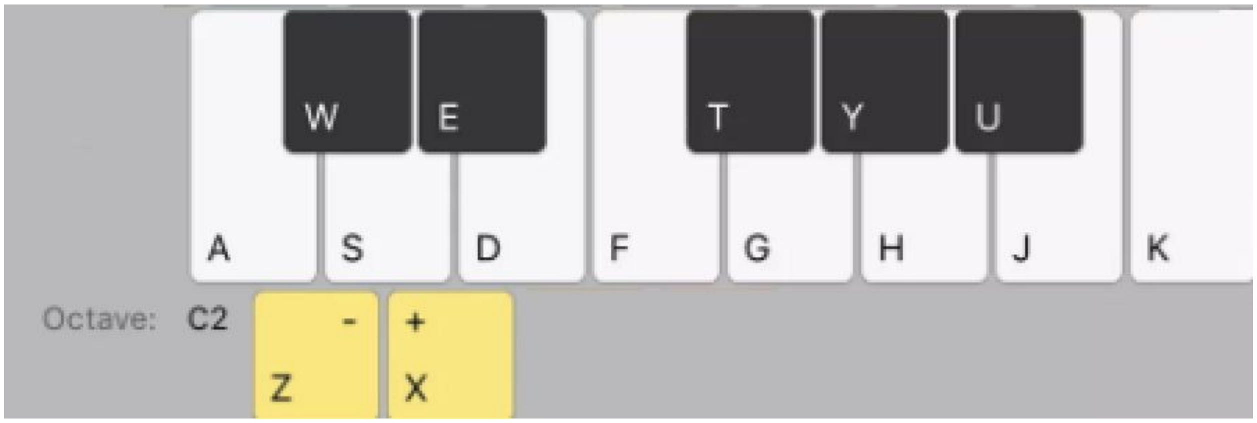


Table of Contents

Overview

Our project is an implementation of a digital audio workstation (DAW), such as Garageband, Logic, Ableton, FL Studio, etc. This software enables producers to record, edit, and mix audio. DAWs typically include digital instruments, the ability to record multiple tracks over each other, and loopable playback. Our simplified design has four different recordable tracks, each with a synthesizer instrument with 8 different sounds spanning 8 octaves, and is played using a standard USB computer keyboard using the keys shown below.



Each of these tracks has three requirements to be played:

- 13 bits to indicate which notes in the octave are being played
- 3 bits to indicate which octave is being played
- 3 bits to indicate which instrument is being played

The record function saves the state of these 19 bits 4 times a second for 64 seconds for each instrument (four 19x256 BRAMs, which are actually implemented as 20x256 to provide slack for extra functionality). In converting these bits to 8 bit 48 kHz audio, we use 8 bit look up tables of 8 predefined waveforms (sine, saw, square, triangle, experimental random coefficients, and some combinations).

The entire system runs on a 65 MHz clock that drives the XVGA display at 60 Hz and allows us a little bit more slack with critical path delay.

Our motivation for this project comes from both of our passion for music and a desire to learn about interfacing with HID devices and digital signal processing.

Milestones

Commitment (all accomplished)

Live synthesizer playing: Input is read from a USB keyboard, parsed into audio waveforms of a certain instrument, and output by the headphone jack.
Choose waveform of synth (sine, square, saw, etc.): The user can select the waveform for the audio to be played through, given a variety of “instrument” options.
Display instrument view: The system outputs VGA to a monitor to display a piano with the played notes denoted, along with some indicator of the waveform being selected.

Goal (all accomplished)

Recording: Input can be saved to a recording or played live, this is managed by the track module.
Multi-track support: Multiple tracks exist and the input can be routed into whichever is desired.
Mixing: The multiple tracks can be mixed together at varying volumes to produce the output.
Display track view: The current track has a display featuring the waveform recorded/played live.

Stretch (did not have time to complete, noted in future improvements section)

Effects: Tracks can be routed through effects modules where the user may select effects to apply to them.
Stereo mixing (panning): The mixer features panning of tracks left/right.
Velocity control: The input module can read note velocity (effective volume) and route that through to output.
Automation: The mixer might automatically adjust dynamics over time.
Display effects view: Selected effects are displayed on the monitor.

Modules

Input Handler

This module serves to turn the keyboard serial data into a useful format for the DAW to process. The format we chose for both space constraints and consideration of practicality is a 13-bit binary number encoding of whether each individual key in a given octave (plus the octave of the root) is currently being pressed or not, and a 3-bit binary number encoding the octave chosen. This saves on space compared to the 88-bit format we proposed, and allows for the same functionality given the keyboard only allows for one octave to be played at a given time anyway.

This module has a nested keyboard_handler module containing the serial receiver and outputting a 32-bit shift buffer of the four most-recent 8-bit keyboard scan codes. The keyboard input comes in through the ps2 data emulator built-in to the FPGA, and an HID-supported keyboard is plugged directly into the USB host port on the FPGA. The keyboard_handler's sequential element clocked on the ps2 data clock, which is clocked by the keyboard at some frequency much less than the FPGA clock only when information is being sent. The ps2_data and ps2_clk lines are debounced for 10 cycles at 65MHz to deal with any noise and synchronize the different clock domains.

A scan code preceded with the “F0” scancode indicates a key release, and a lack thereof indicates a key press, so we use a simple state machine with state transitions on updates in the scancode buffer to determine how to update the notes and octave in response to a new scan code.

Debounce

This module serves to eliminate any noise caused by “bouncing”, a mechanical phenomenon where a button or transducer takes some stabilization time in the transition between states before it’s output can be guaranteed to be what is expected. The buttons on the FPGA are susceptible to bouncing, as are the ps2 data and clk lines (due to the fact that these are relying on quartz clocking inside the keyboard).

Additionally, debouncers are useful as modules which synchronize external signals operating under a different clock domain to our system’s 65Mhz clock, as they are effectively sampling the data at our preferred system clock speed.

Recorder

This is the module that takes in the output from the input module and writes it to recording RAM if necessary, saving the state of user input every quarter second.

Each of the tracks has one BRAM with each having width 20 and depth 256. The 20 bits is to store the set of notes being played, the octave, and the instrument (there is one extra bit for flexibility with adding features). The 256 bits means the BRAM can store 256 quarter seconds of data, i.e. 64 seconds. When in recording mode, this module writes the current state of the inputs to the BRAM corresponding to the currently selected track (for all other tracks, it will simply read the BRAM).

This module is also responsible for storing the latest valid index in each BRAM, which effectively gives the user the ability to reset a given track. When the user wants to clear a track, it resets this latest valid index to 0. The index increments when recording past it. These indices are an output for use by the display module.

This module also generates signals when the beat counter increments for the metronome module.

Metronome

This small module creates the audio for the metronome. To do this, it uses signals generated from the recorder counter (which keeps track of the current beat), to determine when to output sound and when to output silence. It uses tone generators just like octave in order to generate the audio.

Octave

This is the module that takes the output from the recorder module (i.e. the data that says what notes, octaves, and instruments should be currently playing) and outputs an 8 bit audio_out for each track. Accordingly, there are four instances of this module in top_level, one for each track.

By design, all notes currently being played will be in the same octave, as opposed to having a different octave for each note. To evaluate the audio_out signal, there are 13 tone generators, one for each note. The tone generator, described below, takes in an octave and instrument and produces one 8 bit audio output.

The relevant thirteen tone generator outputs are then added together to produce an octave output. Here, relevant means that the input to the octave module indicates that particular note should be playing. The one complication here is that because the audio is limited to 8 bits, simply adding together will cause overflow and not work as intended. To fix this issue, the audio is bit shifted before adding. To prevent it from being too quiet when only one note is playing, the bit shift amount depends on the number of notes currently being played. To prevent clipping after adding the mixer and metronome, everything is bit shifted one more than it needs to be. For example, if two notes are meant to be played at the same time, they could each be bit shifted by 1 (i.e. divided by 2) before adding so that when they are added their sum fits in 8 bits. Because we add an extra bit of padding, our shifts are as follows:

- 1 note: shift by 1
- 2 notes: shift by 2
- 3-4 notes: shift by 3
- 5-8 notes: shift by 4
- 9-13 notes: shift by 5

The sum of these shifted values is then output as the audio of the octave (i.e. the track).

Note that shifting by 4 does lose a considerable amount of data about the waveform, since a 4 bit sine value is less accurate than 8. However, since 4 notes are playing at the same time, it is very hard to detect this difference.

One effect of doing different amounts of bit shifts based on the number of notes is that the volume of each individual note isn’t fixed-a note will sound louder if it is the only thing being played on its track than if multiple notes are sharing the track with it. Effectively, instead of each note being the same volume so that the output gets louder with more notes, we keep the octave/track volume constant and vary the relative volumes of the notes to achieve that.

Tone Generator

The tone generator module takes in an octave and instrument, both 3 bits, as well as a parameter that helps determine the frequency of the resulting wave. Using an internal counter, along with the parameter, it cycles through LUT values at a speed that produces the appropriate frequency. Because changing octaves multiplies/divides the frequency by 2, these can be encoded as simple bit shifts.

Each instance of this module contains 8 LUT modules, returning the output of the one corresponding to the instrument input.

LUTs

The 8-bit lookup tables store one period of several common waveforms such as sine, triangle, saw, and square. There is also one with just random values, and a couple combinations of sine waves of different frequencies (adding overtones).

Certain waveforms (e.g. square wave) that do not have to be implemented with an 8 bit lookup table like this are still implemented this way to show that the design will work for an arbitrary set of waveforms.

One disadvantage to using lookup tables like this is that to do the lookups, all lookups need to happen even though only one of them ends up being used. Thus, after performing the lookups there has to be some sort of layered MUX to choose between them. As the number of waveforms increases, so does the propagation delay of this MUX.

However, because all the lookups are performed anyway, this opens up the option to mix the waveforms relatively easily. That is, the current implementation makes it easy to add a feature that would allow a user to use a waveform that is 50% sine and 50% saw. An alternate RAM implementation of these look up tables that wouldn’t involve looking up all of the values would not have this flexibility as easily, since it would only be able to be read once per clock cycle.

Waveform_select & track_select

These modules take in a clocked 1-bit signal (associated with debounced btnd and btnu for waveform and track selection, respectively) and perform simple state transitions between the 8 available waveforms and 4 available tracks. The difference between the modules is in the use-case and the bit depth of the switched output.

Mixer

This module takes in information about the four tracks and the metronome “track” and adds them together, using volume information from the FPGA switches to determine how much weight should be associated with each track. Track 0’s volume is determined by sw[15:13], track 1 by sw[12:10], track 2 by sw[9:7], track 3 by sw[6:4], and the metronome by sw[3:1]. Firstly, all tracks were bit shifted by 7 - volume, where volume is a 3-bit number corresponding to the track’s respective volume level. Afterwards, we bit shift each track right by 2 and sum them to yield the input, as to prevent overflow when adding them. Note that, unlike the octave module, *everything is always shifted by 2*. This makes it so that there are no surprises in volume changes. The mixer doesn’t care if three tracks are playing or one track is playing; track 1 at volume 5 will still sound the same. This approach is somewhat rudimentary as we lose precision due to the shifting and the effect on the perceived volume is very vast as perceived through the audio jack, but it fulfills the purpose of controlling volume levels of tracks independently for mixing.

Seven_seg_controller

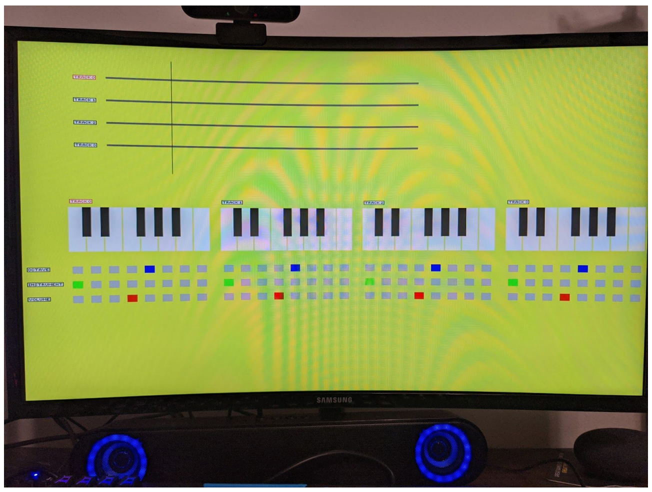
This module serves primarily as a debugging feature and doesn’t have a primary feature, in the course of the project we have used it to verify scan codes, counters, notes, and volumes. Currently it just displays the latest valid beat of the currently selected track and the current beat (0 to 255).

Xvga

This module serves to generate the signals necessary to adhere to the 1024x768 @60Hz XVGA protocol we designed our FPGA to support. Entirely stateful, gives the top_level an hcount, vcount, vsync, hsync, and blank signal to be turned into the signal at the VGA port. hcount and vcount are used in the top_level and display to represent the position of the current pixel being rendered. hsync, vsync, and blank are signals which serve to control the timing of VGA events such as row/column switching and blanking of the screen.

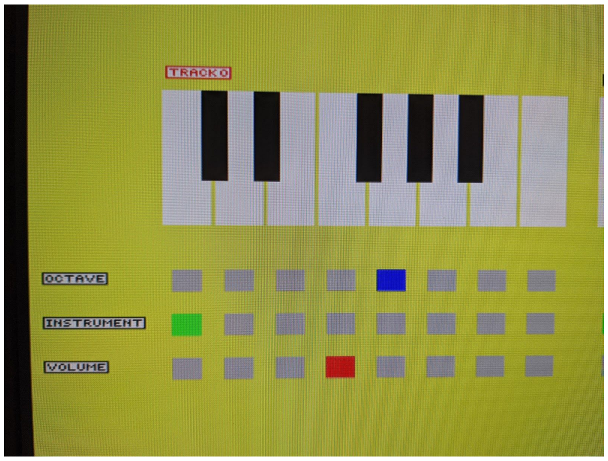
Display

This module generates all of the graphics to be displayed on the screen by outputting a 12-bit color pixel which is routed to VGA in top_level. This module is broken down into a number of sections dedicated to controlling related aspects of the screen, namely the keyboard, track config, waveform heatmap, and text icons. All of these are then tied together with the pixel selection multiplexer.



Within the 65MHz, our system has plenty of time to handle the designated graphics with minimal pipelining, so we resorted to delegating pipelining to the image ROMs and otherwise handling propagation delay by multiplexing between the different sections of the screen.

All portions of the display can be broken down largely into rectangle_blobs and selectable_blobs, both of which are effectively wrapped bounding-boxes. A pixel is input into one of these modules, bounds checking is performed, and if the pixel falls inside the box, it's color is changed to that of the box; otherwise, the pixel input is assigned to the output. The selectable_blobs, in addition to the feature mentioned previously, take in a 1-bit selection signal which determines what color the inside of the bounding box should be. The waveform_heatmap uses more complex pixel-selection by internally storing a circular buffer of the number of notes being played at any instance in a recording per track. The horizontal position of the considered pixel is linearly transformed into an index in this buffer, which is routed to a colormap, which is routed to the output if it passes a boundary check.



The keyboard display shows one octave of a keyboard containing all the white and black keys between two adjacent octaves of the note C. Signaled by keypresses on valid keys, the notes on the keyboard will fill with red (if white) or blue (if black) to indicate that the respective key is being pressed. This module consists solely of in-series rectangle_blobs as mentioned before.

The track config shows an array of rectangles arranged in a way to convey to the user what volume level, octave, and instrument they currently have selected. A row in this array corresponds to a label (volume, octave, instrument) and a column corresponds to a level (e.g. the highest octave/volume/instrument ID is the rightmost column). This module also consists solely of in-series rectangle_blobs as mentioned before.

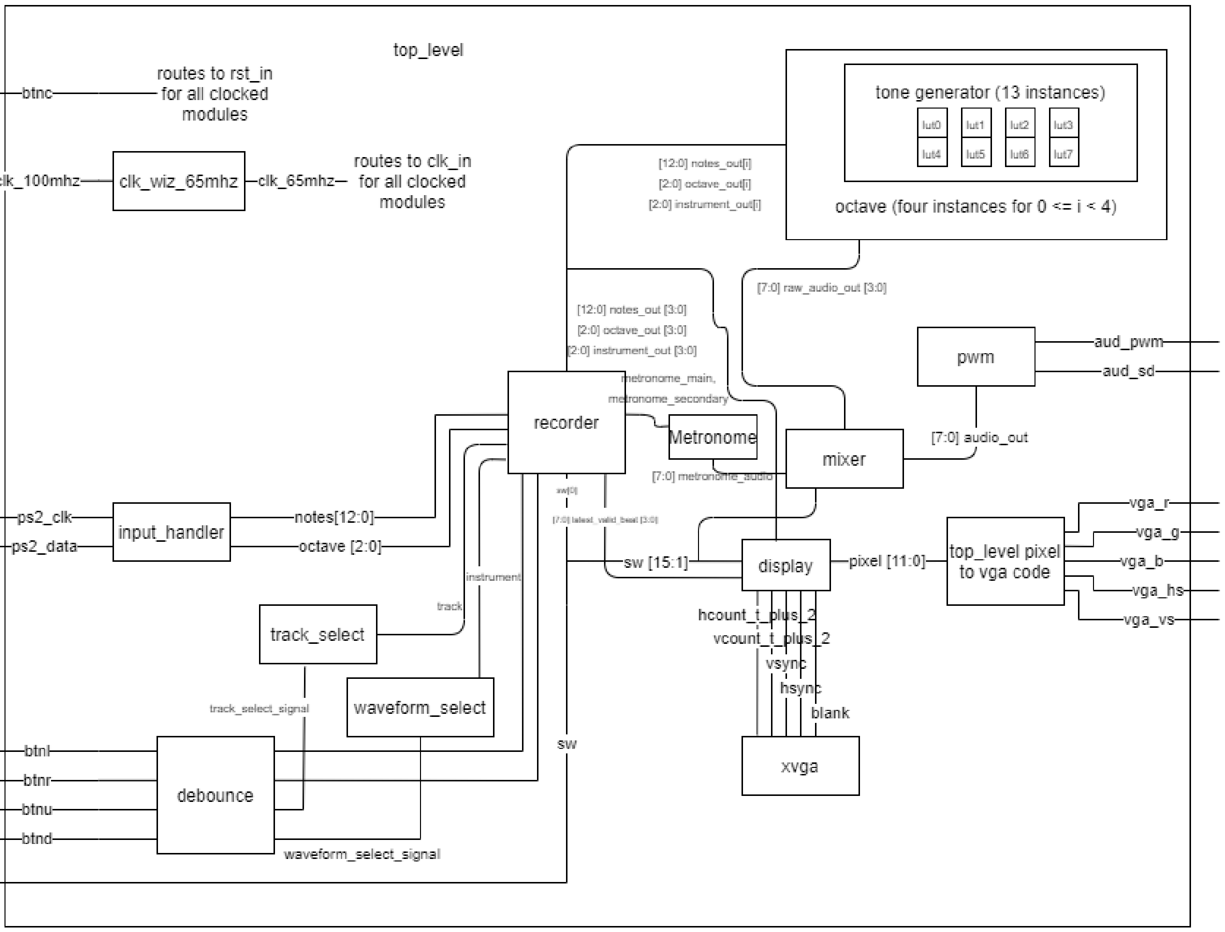
The waveform heatmap shows four "bar lines" corresponding to the four tracks, and a beat-counter bar which moves along the tracks to indicate where the current beat lies. As the recording of a track becomes populated with notes, the respective track's bar line will become populated with colors indicating the number of notes being played at a given time. Deep blue indicates one note, cyan indicates two, green indicates three, according to the last 3-bits of the number of notes being pressed at a given time.

While the rest of the audio architecture is structured to handle more than 7 notes being played simultaneously on one track, the waveform heatmap was chosen to support this limit as many keyboards do not work well with large numbers of simultaneous keypresses, the volume mixing is performed such that large numbers of notes become fairly quiet due to the number of bit shifts needed to support 8-bit audio integrity, and the impracticality of actually recording more than 7 notes on one octave for a piece of music.

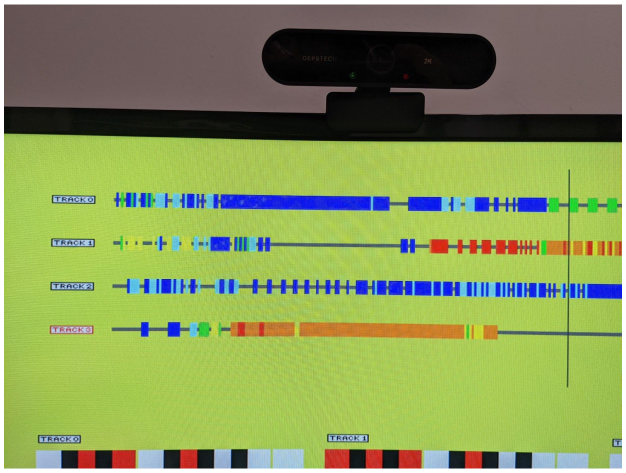
The text icons just denote labels for the octave, volume, and instrument selectors on the track config and the selected tracks on the keyboard and waveform displays. The text icons corresponding to the track config are static, but the icons indicating selections of tracks are highlighted in red to indicate a selected track and black otherwise.

The pixel select multiplexer effectively shortens the path that any pixel has to take by parallelizing mutually-exclusive graphics on the screen and choosing whichever output is unequal to it’s input (the background color), or the background color if none match. This eliminates the need for introducing pipelining and increasing the overall signal delay in pixels, which would introduce new issues with the VGA interface.

Block Diagram



Challenges



- Waveform Heatmap: The heatmap of notes in each track proved to be a more challenging feat than was initially expected. Our initial idea was to display the exact waveform stored in the BRAM, but doing this proved challenging as it would mean introducing way more inputs to route each BRAM from the respective track (octave) to the display. Our workaround was to display a similar amount of information by displaying note counts per timeslot, which is much easier to route properly. The synchronization of the BRAM and display contents proved somewhat challenging as well, but this was mitigated by designating the track reset behavior as resetting a “latest_valid_beat” for which anything after this beat in the buffer is ignored.

- 8 bit audio clipping: As mentioned earlier, to prevent audio from clipping when adding multiple raw waveforms together, it is critical to make sure that they don’t exceed the maximum 8 bits we allow. To do this, we do simple divisions with bit shifts to try to equalize the volume of the instrument no matter how many notes are being played. This works great when 1, 2, or 4 notes are being played, but 3 notes and 5+ notes sound a little bit quieter than one might expect. This is hard to notice, but is it noticeable. One way to deal with this is to use a better approximation for 1/3 than 1/4. For example, we could have used 5/16 or even 21/64. To keep it simple, and because it is hard to notice anyway, we stuck with the easy ratios.
- Display division by 3: For desired proportions on the display, we wanted to give notes in the heatmap a width of 2 pixels and a spacing 1 pixel, summing to an effective 3 pixels per note. However, this introduced much more complicated math via higher propagation delay into the conversion from hcount to circular buffer index, and required us to risk losing a substantial amount of slack with 10-bit multiplies or equivalent sums of bit-shifts. This challenge was resolved by simply changing the proportions such that the note spacing was 0 but note width remained 2. This made the required linear transformation able to be reduced into a simple bit shift and subtraction by a constant, which is much faster computationally. Also, this leaves more space for any other modules we might decide to add.

Future Improvements

- Transpose button: Right now, the synthesizers can only be played in the default key, C major. We would like to add a feature to transpose. We initially had the idea of changing the sampling rate, but this would change the fidelity of the audio which feels more like a hack than a true solution.

- Instrument waveform display: The ability to see a period of the waveform of the instrument currently selected would be a nice immersive step which could help users better identify the instruments they are using.
- Expansion of keyboard options: Our project currently relies on the FPGA buttons and switches for a lot of its functionality. Ideally, the keyboard would be able to control everything, so integrating these features into keys could be a potential improvement.
- Testbenching: Our current project relies very little on testbenching. In the future, this would save a lot of time in implementing new features.
- Smarter keyboard sampling: Notes are currently sampled at the metronome beat, so if the player has inconsistent rhythm or is playing faster than the metronome supports, these notes are not picked up by the recording.
- Stereo support: The FPGA currently supports mono output. Mixers tend to allow for panning of tracks. It could be a neat improvement to figure out how to convert a signal from a mono port into a stereo-interpretable format.
- Digital effects: The ability to apply various convolutional filters (distortion, envelope, low-pass) over the audio on multiple tracks.
- Automation: The ability to program dynamics in mixing over-time (fade-ins, etc).
- Velocity control: The ability to control per-note effective “volume” by interpreting keyboard commands as controlling a measure of how hard a note is being pressed.
- Audio Pipelining: Right now, all the audio calculations are done in a single cycle because they can be. As more instruments and tracks get added, this will no longer be the case. Luckily, the 48 kHz output (vs. the 65 MHz clock) means there are over a thousand clock cycles to produce output. Taking multiple cycles could also allow switching to BRAM look up tables.

Lessons Learned / Advice for Future Projects

Julian:

From this project, I learned that version control in Vivado projects is a nightmare and a good .gitignore is absolutely essential to get version-controlled work done effectively. I also learned firsthand why testbenching is so essential, as I'd dismissed the main module I spent my time working on (display) as too difficult to testbench, but probably ended up wasting many hours in synthesis that I could have saved if I took the time to build and work with a proper testbench.

Additionally, I learned the importance of synchronizing inputs and clock domain crossing, as unsynchronized keyboard inputs initially led to very unpredictable keyboard behavior. Clock domain crossing between our system's 100MHz clock and the clock wizard's 65MHz clock resulted in our WNS(worst negative slack) being off-puttingly bad, as where these signals intersected it was seemingly expecting processing at around 6.5GHz (the least common multiple frequency between the two). This clock domain issue could have been fixed by synchronization, but we chose to clock everything at 65MHz as we didn't need the 100MHz throughput and a higher tolerance for slack was desired for our calculations.

As far as insight for future projects goes, I would strongly advise prioritizing testbenching and maybe even test-first programming as the amount of time and stress saved in the long-run by having functional tests is very notable. I would also stress the importance of thoroughly analyzing any external sources used or studied, as I had originally used Digilent-provided code for USB keyboard input and discovered that there was actually a major flaw in it, where repeated keypresses could not be detected due to a misplaced if statement. My assumption that the external source was correct led to my dismissal of the issue as pertaining to something I wrote and tacked many hours of needless debugging onto my original time budget.

Nathan:

+1 on Julian's version control comment.

I think the main thing I learned is organizational. There were dozens of ways to implement the audio pipeline. The way it is currently set up, the recorder module has to know about all 4 tracks instead of each track having its own recorder module. The reason it works this way is because it prevents additional logic being required in top_level to be able to decide which tracks to use the recorded values from and which tracks to just pass the input values through (when currently recording). Of course, it could really work either way. Before I realized it would be easier to set the recorder up like this though, I had all different bit depths of values to control the modules. It wasn’t a huge deal to change them, but it was a little annoying. So I would say my advice here is to just have a very good picture of how everything will fit together before starting (i.e. exactly which modules will be responsible for holding exactly what).

We did originally have a whole block diagram for exactly that. But the design changed slightly each time we realized better ways to do things, and we didn’t stop to update the diagram. For example, for generating audio, originally I thought the best way to do it would be to store a period of each waveform of each note and to read them all at the same speed. While perhaps easier, it would take up far more space than the way we ended up implementing it, which is to just store one period of each waveform and to read it at an appropriate pace to achieve the desired note. There are disadvantages (like doing it for all notes, even ones you are not playing), but it is far easier to implement and gets the job done.

Source Files

The source code for our project can be found at the following link: https://github.mit.edu/jesp1999/FPGA-DAW.

Appendix

References

[1]
https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-4-ddr-keyboard-demo/start
[2]
https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual

Acknowledgments

We would like to thank Joe and Gim for a wonderful semester. We would especially like to thank Joe for all the help and patience in office hours and for being a fun and entertaining lecturer and Gim for mentoring our project and helping us get up and running with the USB keyboard input. Also a big thank you to the TAs and all the LAs that helped us along the way.