FPDJ

Elena Byun, Angus MacMullen, Baltazar Ortiz

FPDJ is an all-in-one electronic music production station. Our project aims to provide users with a stand-alone hardware interface for composing musical patterns, play back audio clips from an SD card, and synthesize other sounds in real-time. Many alternatives exist in the form of computer software, but dedicated hardware can offer a more tactile experience, more conducive to creativity and fun.

As proposed, FPDJ was to contain three main modules. A sequencer would allow for live editing of a rhythmic and melodic sequence, allowing for triggers for different sounds and notes to be placed in time in a looping pattern, and allow a user to trigger their own sounds outside of the programmed sequence. The sequencer would be controlled with an interface of buttons, LEDs, and encoders. A sampler module would be able to load sound samples from an SD card and enable the sequencer to trigger them on command. It would support the playback of multiple samples simultaneously for complex, multi-layered tracks. A synthesizer module would generate sound on the fly, based on the principles of conventional analog music synthesizers, where oscillators are modulated and passed through filters to shape their timbre. The synthesizer's parameters would be tweakable in real-time on an interface of knobs and buttons.

We were unable to reach this target functionality. In the end, we built three individual modules that each on their own achieved at least baseline functionality; we fell short of integrating them fully, so FPDJ is, for now, incomplete.

Nonetheless, in this report, we aim to provide a detailed description of what we intended to do, where the project currently stands, and how we look to continue progress on FPDJ in the future.

Contents

SEQUENCER (Elena)

The Sequencer module handles the composition of rhythmic and melodic patterns. Using pushbuttons, switches, and a quadrature encoder, the user sets musical samples and synthesized notes to trigger at different points in a sixteen-step loop. In addition, the user can also trigger non-scheduled samples and notes, adding a "live play" experience.

The Sequencer has the following inputs: a 200MHz system clock, 16 multi-function buttons, play/pause button, write on/off button, commit button, revert button, track select switches, and a rotary encoder.

The Sequencer has the following outputs: a 7 bit MIDI standard pitch command to the Synthesizer, a 4 bit sample command to the Sampler, 8 seven-segment displays on the Nexys4, 16 LEDs for multi-function visualization, as well as 4 more LEDs to display the status of play/pause, write on/off, commit, and revert buttons.

The large number of IO required for this module necessitated the use of shift registers. Two 74HC595 and two 74HC165 were incorporated for the operation of the 16 multi-function buttons and 16 LEDs. This allowed the button readings and LED visualization to only take up 3 wires each, one of which was shared (serial clock input). Overall, the entire Sequencer module only needed to utilize two PMOD ports in total.

The submodules of the Sequencer are as follows: step clock divider, sample clock divider, serial clock divider, piso, sipo, decoder, input handler, output handler, composer, and conductor. The debounce and display_8hex submodules from the 6.111 course materials was also utilized in this module. The submodules are explained below.

Step Clock Divider

The step clock divider takes the 200MHz system clock and divides it into a slower clock to be used to define the speed at which scheduled samples and notes are triggered. Since 140 bpm was decided as a target speed for our system, the step clock was decided to output a 9.41Hz pulse, such that it would be hold a high value at the beginning of every sixteenth note.

Sample Clock Divider

The sample clock divider takes the 200MHz system clock and divides it into a 44khz clock to be used by the Sampler module.

Serial Clock Divider

The sample clock divider takes the 200MHz system clock and divides it into a 1Mhz clock. This clock signal is used to operate the shift registers for reading buttons and sending outputs to LEDs.

PISO (Parallel in, Serial out)

The PISO receives a 16 bit LED command from the output handler and outputs it on a single MOSI serial wire. When all 16 bits have been shifted into the 74HC595 (a serial in, parallel out shift register), it activates the latch, allowing the command to be loaded and visualized. Every time the LED command changes, the "change_LED" flag signals the PISO submodule to trigger this process.

SIPO (Serial in, Parallel out)

The SIPO interprets 16 button inputs by continuously commanding a load from the 74HC165 (a parallel in, serial out shift register) and reading from a single MISO serial wire. These updated button inputs, after being debounced, are sent to the input handler.

Rotary Encoder Decoder

The Decoder interprets quadrature signals from the rotary encoder. By counting the transitions of the two input signals from the encoder, the encoder angle change, as well as the direction of the angle change, can be found. These values are passed on to the input handler.

Input Handler

The Input Handler interprets all user inputs to determine what the Composer and the Conductor submodules do or don't. If the play/pause button is pressed, the "play" state is toggled. Similarly, a press of the write on/off toggles the "write" state. All other inputs are interpreted differently depending on "write" state, as well as the track select.
If write mode is on, the 16 button inputs correspond to a selection of a time step that the user would like to compose to. Once a time step is selected, the user can manipulate the encoder to select the sample or pitch to load to that time step. The Sequencer module can support four tracks in total: 2 sampler tracks, and 2 synthesizer tracks – if a sampler track is selected (track 0 and 1), then turning the encoder scrolls through 16 options, where 0 means "play no sample", and the rest correspond to sample numbers. If a synthesizer track

is selected (track 2 and 3), then turning the encoder scrolls through almost all MIDI standard pitches from C#0 to G10, as well as an option to not play at all. In addition, any revert and commit button inputs are accepted. All of these inputs are then passed on to the Composer to be saved or to affect saved memory.

If write mode is not on, the 16 button inputs correspond to samples and pitches to be "live played". If a sampler track is selected, buttons 1-15 trigger samples 1-15. If a synthesizer track is selected, buttons 0-15 trigger pitches from C2 to D#3. In addition, manipulating the encoder allows the user to change the octave of the triggered pitches. Revert and commit signals are ignored. All of these inputs are then passed on to the Conductor, bypassing the Composer, to be immediately played.

Output Handler

The Output Handler manages the visualization of module state that is useful to the user. This includes the input signal to the PISO module to control 16 LEDS that indicate what button press is being read, as well as the 7 segment displays on the Nexys4. The eight on board 7 segment displays show the following:

- 1 digit current read button (from 16 multi-function buttons; this is redundant with the LEDs.)

- 1 digit current step (for when play mode is on)

- 2 digits currently selected pitch

- 1 digit currently selected synth parameter (unused; was a reach goal)

- 1 digit currently selected sample

- 1 digit currently selected time step (for when write mode is on)

- 1 digit currently selected track number

Composer

The composer receives the following inputs from the input handler: currently selected pitch, synth parameter, sample number, track number, revert, commit, and the write state. The composer also receives the following input from the conductor: current step.

Whenever new information is received in write mode, the composer saves the 16 bit pitch (8), synth parameter (4), and sample number (4) information to a 2D array (4 tracks x 16 steps). Any work before a “commit” is saved to a WIP (work in progress) array. When a “commit” occurs, the WIP array is copied to a COM (committed) array, and the previous WIP array is copied to a PRV (previous) array. When a “revert” occurs, the PRV array is copied to the COM array, and the COM array is copied to the WIP array. Since music is only played back from the COM array, this allows the user to freely modify and compose without always having to affect the playback, allowing for careful planning and control.

At every time step, the current step input to the Composer changes; the Composer outputs four 16 bit wires for what each track needs to be playing at that time step to be used by the conductor.

Conductor

The conductor has the following inputs: step clock, four 16 bit wires for each track’s pitch, syn param, and sample number info for scheduled play, user selected pitch, synth param, and sample number info for nonscheduled “live” play, play state, and write state.

The conductor has the following outputs: 4 bit current step, 16 bit sampler command, 8 bit synthesizer pitch command, and 4 bit synth parameter command(unused).

The conductor updates the current step every rising edge of the step clock. When play mode is on, every time there is a new current step, it uses the new to-be-played information from the Composer module and triggers each on the sampler and synthesizer commands at each rising edge of the system clock. When it is done iterating through each track, if write mode is on, it rests for the rest of the time step. If write mode is on, then it passes on user selected pitch, synth param, and sample information directly to the outputs, such that “live play” is enabled.

The Sampler handles the loading and playback of recorded audio clips. It retrieves audio files from the SD card on the Nexys 4 board and stores them in the board’s DDR RAM. The Sequencer sends the Sampler a trigger number from 0 to 15, with 0 corresponding to silence and the other numbers corresponding to loaded samples. When a trigger number greater than zero is received, the Sampler reads the corresponding sample from the DDR RAM and and sends the data to the mixer at a sample rate of 44.1kHz. The Sampler supports 16 bit, mono, 44.1kHZ WAV files, which are slightly processed by a Python script, as described below.

Sequencer Conclusion

The Sequencer module, in many ways, reached most goals that I intended to achieve in my proposal. Track muting and track clearing functions were not completed due to lack of time, but instead, multiple track support was prioritized.

Off-board hardware was well-utilized, though the breadboarded user interface did leave much to be desired for future iterations. The 16 LED display, time allowing, could have been improved to show more than just the current button pressed, such as performing a light show based on what samples and pitches are currently playing.

The initial plan to make a mode manager, operating via an FSM, quickly proved to be inelegant; after several more design iterations, the current system of submodules was devised. With more time spent thinking about the system architecture beforehand, the time spent on this module could have been greatly reduced.

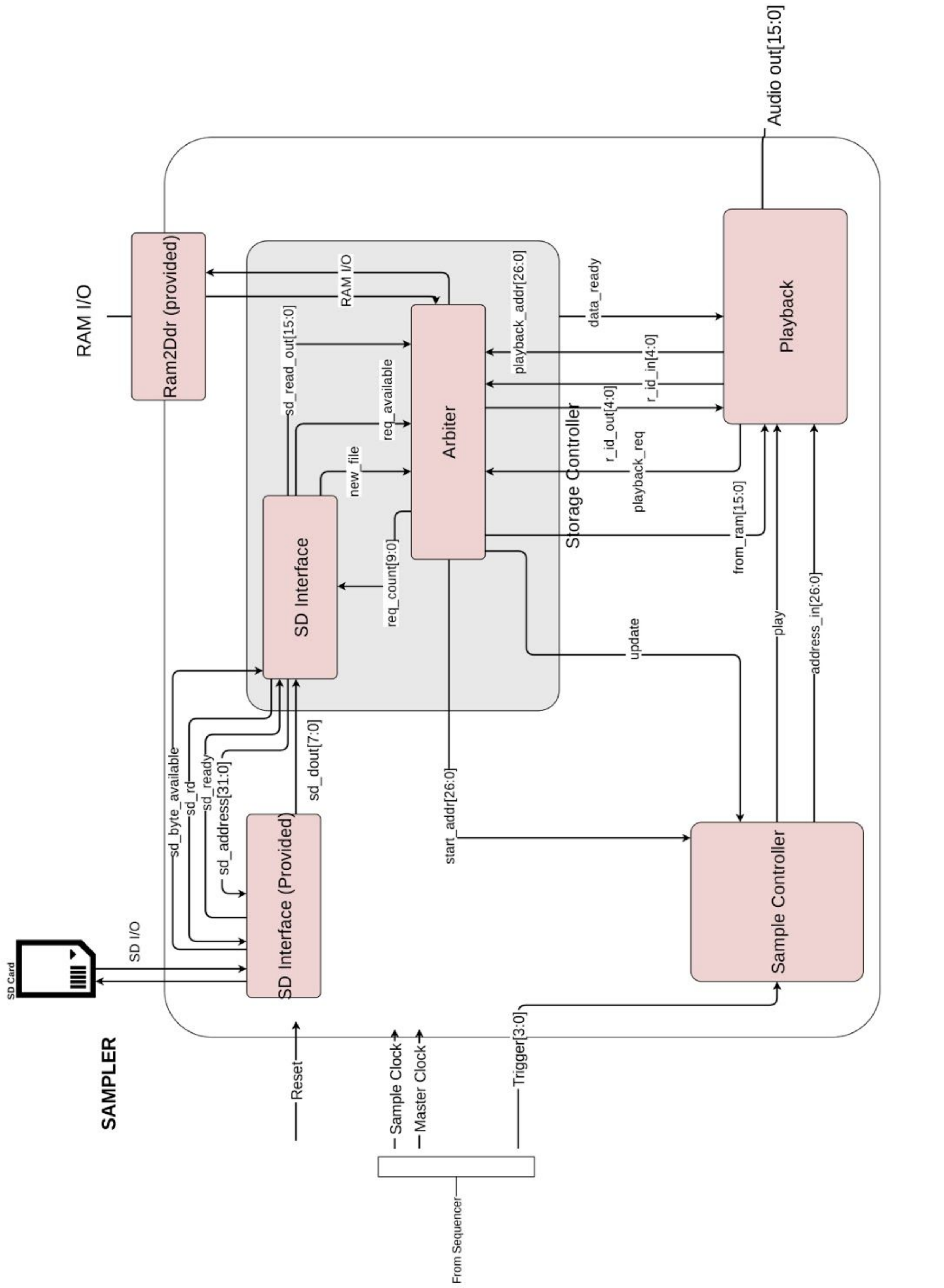
Unfortunately, the as the integration goal was not reached, the Sequencer module's full performance was neither obvious nor easily testable.

SAMPLER (Baltazar)

The Sampler consists of a Sample Controller, a Storage Controller, and a Playback module. The Sample Controller and Storage Controller were fully functional by the end of the project, but the Playback module suffered from a few bugs that prevented correct audio output. However, these should be easily correctable to achieve full functionality.

The Sampler nominally runs at 100MHz. However, the SD card runs at 25MHz, and the DDR RAM runs at 200MHz. Theoretically, the Sampler's core modules should be fine running at 200MHz as well, but this was not tested.

The block diagram on the following page omits a few top level wires that were used for debugging, and has explicit widths for some busses that were parameterized, showing the values used in the final version of the design.


SAMPLER
SD Card
SD I/O
RAM I/O
Ram2Ddr (provided)
RAM I/O
Storage Controller
SD Interface
Arbiter
SD Interface (Provided)
Playback
Sample Controller
Audio out[15:0]
sd_read_out[15:0]
req_available
new_file
req_count[9:0]
playback_addr[26:0]
data_ready
r_id_in[4:0]
r_id_out[4:0]
playback_req
from_ram[15:0]
update
play
address_in[26:0]
start_addr[26:0]
sd_byte_available
sd_rd
sd_ready
sd_address[31:0]
sd_dout[7:0]
Reset
Sample Clock
Master Clock
Trigger[3:0]
From Sequencer

Storage Controller

The storage controller provides an interface between the SD card and the RAM. The storage controller turned out to be the most complex part of the Sampler, and was split into two sub-modules to organize its functionality.

SD Interface

The first module that makes up the storage controller is the SD Interface. It interfaces with the Nexys 4 SD interface provided on the course website, and adds a FSM that allows the FPGA to read SD cards that have been prepared with a Python script. The Python script extracts the samples from the WAV files (skipping the headers for simplicity) and inserts "magic words" (0xDEADBEEF, 0xCAFED00D, and 0xFEE1DEAD) as separators corresponding to the beginning and end of each audio file as well as end of the data loaded on the SD.

The SD Interface FSM starts a read once the FPGA is powered on and the SD card is ready, then continues to start 512 byte reads until the end of the written data is reached. The SD Interface receives the number of requests currently stored in the Arbiter, the second module of the Storage Controller, and if there is not space for another full 512 byte read, the FSM waits until some requests have been processed before starting another read. The version of the FSM used for the final checkoff of the project stays in a "done" state after loading and never needed to write to the SD card, but the FSM can easily be altered to go back to the ready state to trigger more than one load in future versions. In addition, the FSM could be easily expanded to support writing data back to the SD card.

The SD Interface buffers the bytes it receives and passes them to the Arbiter in 16 bit chunks. In addition, it signals when the first bytes of a new file are being sent to the Arbiter.

The largest challenge that the SD Interface provided was that it runs at 25 MHz, while the rest of the FPDJ system runs at 100-200MHz. This was mainly an issue because it was a detail that was forgotten until hardware testing started, but it was fairly easy to handle.

Arbiter

The Arbiter acts as an interface to the DDR RAM. In the current design, it is connected to the SD Interface and the Playback module, but it could be easily generalized into a port system that would allow it to be quickly connected to a recording module or other modules that need RAM access.

The Arbiter consists of a FIFO IP core and two FSMs. The FIFO is configured as a First Word Fall Through FIFO for performance, but the design works with a standard FIFO as well. In addition, the FIFO was configured with a depth of 1024 just in case the SD card outputs two full 512 byte reads without any of the incoming data being written to the RAM, but in practice, because the SD card is only running at 25MHz while the Arbiter runs at 100MHz,

The Arbiter receives "requests" from the SD Interface and Playback modules, which need to write to and read from the RAM, respectively. Requests are 33 bits wide and have the following structure:

- bits 31:30 - READ or WRITE (10 or 11)
  - Read requests:
    - Bits 30:27 - request ID number (described in Playback module section)
    - Bits 26:0 - RAM address
  - Write requests:
    - Bit 30 - new file (1 or 0)
    - Bits 29:16 - unused
    - Bits 15:0 - data to write

The first FSM in the Arbiter is the Request FSM. When the SD Interface or Playback module assert their "incoming request" wire, the Request FSM adds the request to the FIFO. If both modules assert a request at once, it adds them one at a time. The Request FSM also contains internal registers to ensure that it only adds one register per assertion of the incoming request wires. This prevents duplicate requests being added due to the requesting modules having different clock speeds than the Arbiter (for example, the SD Interface, which runs at 25MHz, while the Arbiter runs at 100MHz) and therefore asserting a "one cycle" signal for much longer than one clock cycle inside the Arbiter. This design strategy turned out to be very useful in ensuring that actions that were only supposed to happen once did in fact only occur once, as unexpected signal hold times can occur from FSM transition time in addition to different clock domains.

The second FSM is the RAM FSM. It idles until the FIFO contains at least one request, then reads a request and either reads from or writes to the RAM based on the request type. To do this, the FSM sends signals to the Ram2Ddr component provided by Digilent, which creates a standard SRAM interface for communicating with the DDR. The RAM FSM holds the RAM control wires at the correct values for long enough to meet the timing specifications of the DDR, then moves on to the next request. Based on the type of request, the RAM FSM also outputs whether it is writing a new file along with the address it has just written to or the request ID number of the sample it has just read from RAM, and these pieces of information are sent to the Sample Controller and Playback Module, respectively. Because the Nexys 4 has more than enough space to store 15 audio files, the RAM FSM simply starts at RAM address zero and increments upwards. However, this would be straightforward to change if more complex memory organization was needed.

The Arbiter took a lot of planning to design. The two FSMs were initially combined, but this resulted in incoming requests being missed if the Arbiter was currently holding the RAM control wires for a read or write. After the FSMs were separated, care had to be taken to ensure that the Request FSM was the only one adding data to the FIFO and the RAM FSM was the only one reading data out of the FIFO so that multiple driver errors did not occur. In addition, once it was noted that the SD card runs at 25MHz, the registers had to be added to the Request FSM to prevent duplicate requests being added.
Simulating and hardware testing both the SD Interface and Arbiter was also difficult. Simulating required figuring out how to read data files into the testbench, which was not very clearly documented. Hardware testing was a challenge even with the ILA because of the number of signals that needed to be tracked, which significantly increased implementation time, and because the RAM does not provide any sort of acknowledgement of a successful write, so testing did not really occur until the Playback module was also implemented. In retrospect, it would have been a much better idea to create simple top level to verify each part of the storage controller in the hardware before the entire Sampler was complete.

Sample Controller

The sample controller translates input from the Sequencer into memory addresses that are sent to the playback module.

As described above, when the SD card detects it is reading a new file, it sends a signal to the Arbiter, which then signals and outputs the RAM address when it actually writes the beginning of the new file. The Sample Controller stores the incoming addresses in an array of 15 registers and maintains a pointer for which register to update next. The pointer wraps around from the last to the first register so that registers will be overwritten if more than 15 samples are loaded. Similarly to the Arbiter, the Sample Controller uses an “already updated” flag to prevent duplicate addresses being stored to mitigate issues with longer than expected input signals.

When the Sequencer sends a nonzero trigger number to the Sample Controller, it sends the starting address stored in the register with the trigger number to the Playback module.

The Sample Controller was the simplest module to implement. Due to its simplicity, the Sample Controller can be easily parameterized in the future to support more than 15 samples at once.

Playback

The Playback module handles reading samples from the RAM and mixing their sound data together to create the final audio output for the Sampler. While the Playback module was not completely functional for the project checkoff, the architecture is fully in place to play multiple audio files at once after a few bugs are fixed.

The Playback module is organized into a number of playback "slots." The current design was implemented with 30 playback slots, though the theoretical maximum is somewhere around 100 depending on how timing actually works out on the hardware and between modules (23 microseconds in between each 44.1kHz sample clock pulse divided by 210 nanoseconds minimum DDR read time according to the Digilent Ram2Ddr spec).

Each slot consists of a RAM address, a data buffer, and two flags to track whether the slot has generated a RAM request in the current sample clock cycle and whether it has received data back from the RAM in the current sample clock cycle. In addition, the Playback module has three registers to prevent issues arising from longer than expected input signal lengths, just as the Arbiter and Sample Controller do.

When the Playback module gets a starting address from the Sample Controller, it loads the address plus two (to avoid the first two start of file magic word bytes) into the next available slot using a pointer that wraps around so that the oldest slots are replaced first. A potential improvement to this mechanism would be tracking which slots are empty so that old but still playing slots are skipped in favor of jumping to a newer, empty slot.

At each pulse of the 44.1kHz sample clock, the Playback module begins to look at one slot per clock cycle. If the current slot is at the end of the file, it is cleared. Otherwise, if the slot is not empty and the current slot has not yet received data during this sample clock cycle, the Playback module generates a playback request which is sent to the Arbiter. As described in the request structure, each playback request includes an ID which corresponds to the number of the slot generating the request. When a request is generated, the address stored in the slot is updated for the next sample clock cycle.

When the Arbiter signals that there is data ready from the RAM, it also provides the ID that was in the request that retrieved the available data. Using the ID, the Playback module updates the corresponding slot's flags and shifts the new data into the corresponding slot's buffer. While shifting the data into the buffer, the Playback module switches the order of the two bytes so that they are read in the correct order for actual audio output, as WAV files store their samples in little endian format.

As data is shifted into the corresponding slot's buffer, the data that is being shifted out is added to a register that stores the overall audio mix for the current sample clock cycle. The audio being added to the mix register is shifted to prevent clipping. At each pulse of the sample clock, the contents of the mix register are sent to the overall audio output of the playback module, and the cycle begins again for the next set of audio samples.

The Playback module turned out to be significantly more complicated than expected, and was difficult to debug in hardware as it also relied on the full functionality of the Sample Controller and Storage Controller. Similarly, simulating it either required a slow simulation including both of the other modules, or faking the input data, which does not always provide an accurate model of timing between the modules.

While the architecture for the Playback module appeared to work in simulation, there seemed to be some bugs that were not debugged in time for checkoff. The main issues of unintelligible playback despite seemingly correct ILA output that appeared during checkoff were caused by forgetting to switch the byte order to account for the little-endianness of WAV files. In addition, the wrong bits were not passed to the PWM and the PWM was not biased correctly, but that would not have fixed the byte ordering issue. Even when the PWM and byte ordering were fixed after checkoff, the audio output, while clear, occured at half the expected speed (with a corresponding lowering in pitch). This is most likely due to a miswiring somewhere higher up, but nothing obvious was found. There also appears to be a bug when triggering multiple samples in a row on hardware, although this bug does not appear at all in simulation. Finally, shifting the data being added to the mix was not implemented until after checkoff, but as the overall audio output was not functional then, trying to mix samples without shifting was not fully tested on the hardware. Despite these issues, the Playback module seems to be close to functional, and once fixed in the future, will allow the Sampler to be fully functional.

The Playback module would also benefit from future restructuring, as some of the flags it uses may be redundant, and it contains nested if statements that should be optimized. In addition, Vivado seemed to be unable to implement the slot registers as BRAM as expected, and used separate registers for each slot instead. The reason for this was not particularly clear, but other fixes may resolve this as a side effect.

Sampler Conclusion

Although the audio output was not fully functional by the time of checkoff, designing the Sampler was a very rewarding process. The most interesting parts of the design were working with components in multiple clock domains and creating an inter-module communication system that can be generalized to other designs using the DDR RAM in the future. Making sure that signed values for audio data were correctly propogated through the design also required some extra attention to detail. In addition, learning how to write testbenches that read datafile and looked at internal uut signals was a good learning experience, and figuring out how to use Verilog header files was also useful.

The design process would probably have been smoother if each component had been tested on the hardware after it was simulated, rather than simulating each component until the entire Sampler was complete, then trying to debug the entire thing on the FPGA. This would have involved effectively writing “hardware testbenches” for each module, but the end result would have probably made debugging much easier. In addition, small but important details such as the SD card running at a slow clock speed and WAV data being stored in little-endian format should have been noted early and handled during the initial design process, not realized during the debug process and fixed at the end.

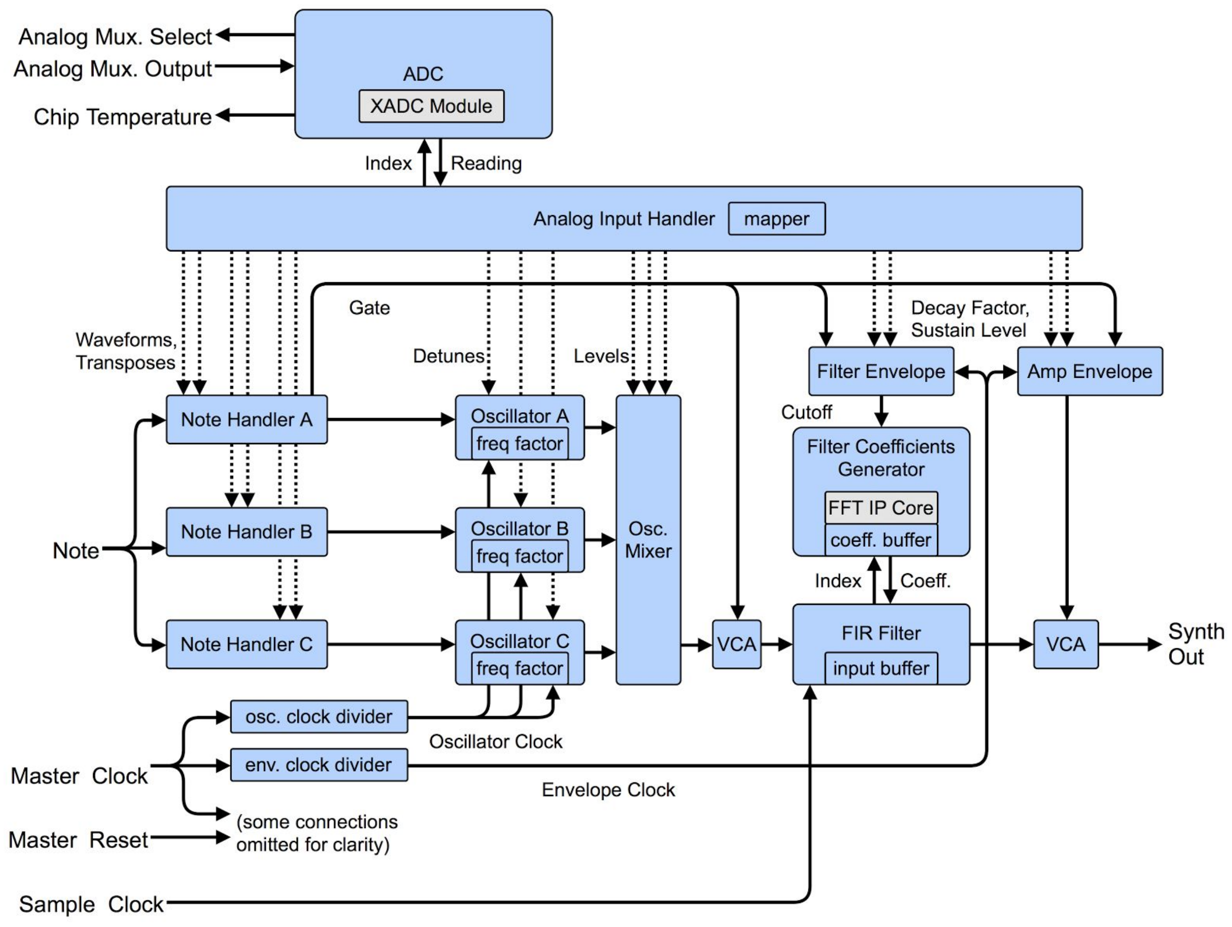
Finally, while the Sampler is relatively independent from the Sequencer and Synthesizer integration-wise, a different version of the Ram2Ddr component had to be used for integration since the Synthesizer uses the XADC. When testing the Sampler on its own, the version of the Ram2Ddr component that automatically instantiates the XADC was used. This meant that an additional wire needed to be exposed to get the board temperature, as required by the Ram2Ddr component, and the non-XADC version of the component seemed to cause a few more hiccups when implementing than the version that instantiates the XADC itself.

(This document is just for Angus to compose his parts of the final report — so there’s no annoying conflict of multiple people writing in the same document. Don’t edit this if you’re not Angus!)

Synthesizer (Angus)

Overview

The synthesizer module is meant to be digital emulation of a typical analog subtractive musical synth. Generally, in subtractive synthesis, the start of the signal chain is one or more oscillators, producing harmonic-rich waveforms, which are passed through filters to shape the harmonic content. Oscillators and filter parameters can be modulated over time to produce many different sorts of sounds. The architecture of the FPDJ synth itself is shown in the following block diagram:



Note Handlers

The primary input to the synthesizer module is a number from the sequencer, representing the note the synthesizer is to play. The mapping of numbers to notes is according to the MIDI standard, where middle C (261.6Hz) is assigned the number 60, and all other notes are in chromatic order above and below that.

The single_note_handler module translates note numbers to frequencies. It also takes into account a transpose parameter, which shifts the outputted frequency by a specified number of semitones up or down. This allows, for example, the oscillators to be tuned to different octaves for a thicker, layered sound. The output of the note handler is sixteen times the selected note frequency in Hz, for higher frequency precision. (at low frequencies, an error of 1Hz can make a pitch noticeably out of tune)

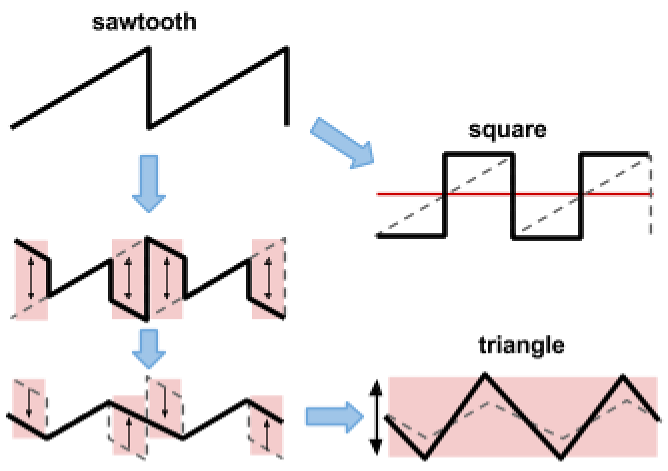
The single_note_handler module has an additional gate output, which is asserted high as long as the note handler receives a nonzero note value. This output is used to trigger subsequent modules along the signal path.

Oscillators

Each oscillator receives a frequency value (16x the frequency in Hz), and outputs a periodic waveform at the specified frequency. Internally, each oscillator has an accumulator which is periodically incremented by a particular value. The size of the increment controls the rate at which the accumulator overflows, which determines the oscillator frequency. To reduce the size of accumulator required, it is incremented at a rate much slower than the system master clock. This oscillator clock was defined to be 1MHz.

The oscillator frequency is directly proportional to the increment value. The proportionality factors to convert frequencies to increments are precomputed, and stored in BRAM in the f2i_factor module. To handle oscillator detuning, the f2i_factor module actually contains a precomputed table of frequency-to-increment factors, one for each detune amount in cents. (Otherwise, a costly exponential computation would be required; an interval of n cents is a factor of $2^{n/1200}$ difference in frequency). Detuned oscillators create a “beating” effect that adds interesting color to a synthesized sound.

The oscillator output waveforms, depicted below, are derived from the accumulator value. The accumulator alone follows a sawtooth shape, counting up and instantly resetting when it reaches the maximum value, so the high bits of the accumulator are used directly for the sawtooth wave. The square wave is generated by comparing the accumulator to a middle threshold, and outputting a high or low value accordingly. The triangle wave is made by inverting and shifting certain parts of the sawtooth wave, and scaling the result back to full amplitude.



Oscillator Mixer

The oscillator mixer performs a weighted sum of the outputs of the three oscillators. In addition to the oscillator signals themselves, the mixer receives input values corresponding to the level of each oscillator in the mix. Blending different amounts of different waveforms at different pitches offers a wide range of synthesized sounds.

FIR Filter

The fir_4095 module performs a convolution of a finite-length impulse response with an impulse signal. For reasonable frequency response performance across the audio range (on the order of 20Hz to 20kHz), the filter operates at a sample rate of 44.1kHz, with an impulse response length of 4095 points. Assuming each point of the impulse response is processed sequentially (minimizing logic area), the whole convolution requires 44100*4095 ≈ 180 computational steps. With a 200MHz master clock, then, the FIR module must then process one sample every cycle, to have the filter output ready for each 44.1kHz sample period. To meet timing requirements, the FIR convolution operation is fully pipelined.

Internally, the fir_4095 module has a BRAM buffer to store the last 4096 samples from the oscillator mixer. The module acquires the impulse response coefficients from the filter coefficients generator module, by specifying a particular index every cycle. To make the filter causal, the impulse response is shifted back 2048 samples, introducing a time delay of 2048/44.1kHz ≈ 50ms, which is a noticeable but still acceptable latency.

As the 44.1kHz sample rate filter is sampling oscillators updated at 1MHz, aliasing may be a concern, as the oscillators would have frequency components well above the 22kHz Nyquist rate. To solve this, a fixed FIR filter with sample rate 1MHz and cutoff frequency around 20kHz could be inserted between the oscillator mixer and adjustable filter. However, in testing, this aliasing was determined not to be a significant issue (only somewhat audible in upper-range notes).

Filter Coefficients Generator

The filter_coeff_gen module produces the time-domain impulse response coefficients used by the FIR filter to process the oscillator signals. Internally, it uses a Xilinx FFT module to compute the inverse DFT of a particular frequency response, and buffers the resulting time-domain data for use in the filter. The module passes a simple lowpass response into the IFFT computation: maximum gain in the passband, with a sharp cutoff to zero in the stopband. The input to the IFFT module is made to be symmetric, so that the time-domain output is real. (in practice, rounding error produces non-zero imaginary components, but these can be neglected)

The primary input to the module is a value representing the cutoff frequency. This value is the threshold above which frequency bin coefficients into the IFFT computation are set to 0. So a value of half the IFFT size (4096 points) corresponds to the maximum cutoff frequency, around 22kHz (half the sampling frequency). A cutoff of 1kHz, for example, would be designated with the value 1kHz/22kHz * 2048 ≈ 93.

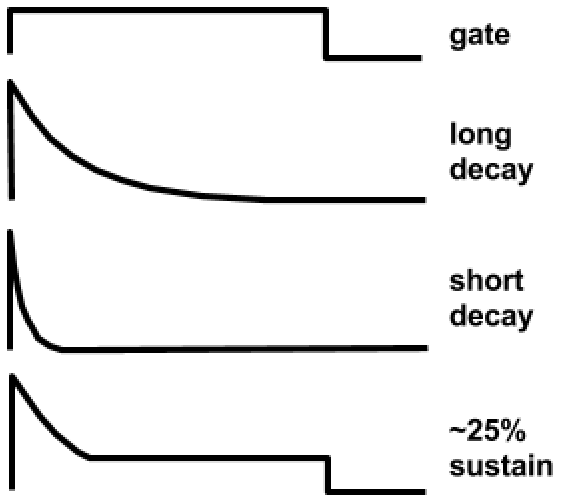
While the oscillator section creates varied timbres additively with harmonic-rich waveforms, the filter works subtractively: applying a lowpass filter removes high-frequency components of the signal, smoothing out the waveform and producing a less bright, harsh tone.

Envelope Generators

The range of possible sounds from the synthesizer can be further expanded if the parameters are modulated over time. For example, the sound of a plucked physical string has two main characteristics: an initial burst of high-frequency harmonics from the pluck, followed by a slower decay of harmonics closer to the string's fundamental frequency as the vibrations slowly die out. This could be emulated in synthesis with a quickly decaying lowpass filter, and a more slowly decaying output amplitude.

The ds_envelope_geom module is a source of this sort of decaying shape that can be used to control other synth parameters. It uses 2 parameters, decay factor and sustain level, to control the shape of the control signal, or envelope. The decay factor determines the speed at which the output falls down from the maximum value, until it reaches the sustain level. The envelope is triggered with a gate input; a rising edge initiates the decay shape, and the output is held at the sustain level until the gate input falls.

The envelope generators are updated at a lower frequency than the audio signals, operating with a separate subdivided envelope clock pulse. They produce a geometric decay, the output being scaled by a constant factor each envelope clock cycle. We perceive frequency and volume on a logarithmic scale, so a geometric decay envelope is better suited to modulating the filter cutoff and output amplitude than a simpler linear envelope, because the geometric decay is closer to an exponential curve.

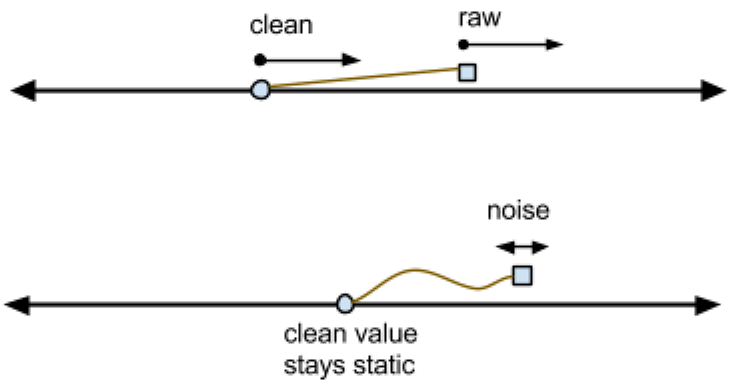


Analog Inputs

The analog_inputs module configures the FPGA's XADC module to take readings from potentiometers through an analog multiplexer. With the analog multiplexer, only one ADC input pin on the FPGA is required to read 8 different potentiometer outputs.

Because the Artix-7's analog inputs can only accept up to 1V, the potentiometers are connected across a ~0.9V supply, generated by an op-amp voltage follower fed by a resistor voltage divider on the 3.3V rail. The wiper of each pot is wired into a 74LV4051 8:1 analog multiplexer, whose common output is connected to an auxiliary analog input on the FPGA.

The XADC module is configured to automatically take readings from each input of the multiplexer sequentially. To clean up noise in the potentiometer inputs, the analog_inputs module applies a sort of "hysteresis," where the clean potentiometer value is "dragged" behind the raw, noisy readings.



Another module can access these cleaned potentiometer readings by selecting one with the pot_select input of the analog_inputs module, after which the module outputs the corresponding reading.

The analog_inputs module also outputs an on-chip temperature reading from the XADC. This value is to be passed to the sampler module, because the DDR memory IP core requires it.

Analog Input Handler

The analog_input_handler module maps the values of the potentiometer inputs to the range of other synth modules' parameters. When programming the FPGA, the 8 potentiometers connected to the board can be mapped to any 8 synth parameters, such as oscillator waveforms or tuning, filter cutoff frequency, or envelope decay rates.

The analog_input_handler module cycles through all of the specified parameters, taking the corresponding potentiometer reading, scaling and offsetting the value appropriately, and outputting the result to other synth modules. For certain parameters, like oscillator detuning, the user shouldn't have to struggle to exactly dial in the default 0 value. Therefore, the input handler module can create a "deadzone" to the potentiometer reading; that is, a window in the center of the potentiometer's travel that maps to a constant output value.

Miscellaneous Submodules

Two internal modules divide the master clock to produce the oscillator and envelope clock pulses.

The last module in the signal path (vca) simply scales the output of the filter by the output of an envelope generator for transient shape to the synth's amplitude. The term "VCA" stands for "voltage controlled amplifier;" in a real analog synth, the module that serves this function would be controlled by an analog voltage.

An additional simple VCA (gate_vca) cuts off the output of the oscillators from the filter when the synth is inactive, so that the filter buffer is cleared out. Otherwise, there is a short burst at the previous frequency when a new note value is sent to the synthesizer module.

Synthesizer Conclusion

In implementing the synthesizer, I realized that my proposed design goals were too ambitious. Tasks like interfacing with the complex XADC and FFT cores took significantly more time than expected, and the many steps in the synth signal chain created many opportunities for mistakes and bugs in the code. I did meet or exceed all of my minimum goals (which were: single synth channel, one tunable oscillator, fixed FIR filter, one knob per parameter), but I didn't fulfill all of my main goals. Still, I feel I made good progress toward them, as the code I did write was designed to be modular and easily expandable.

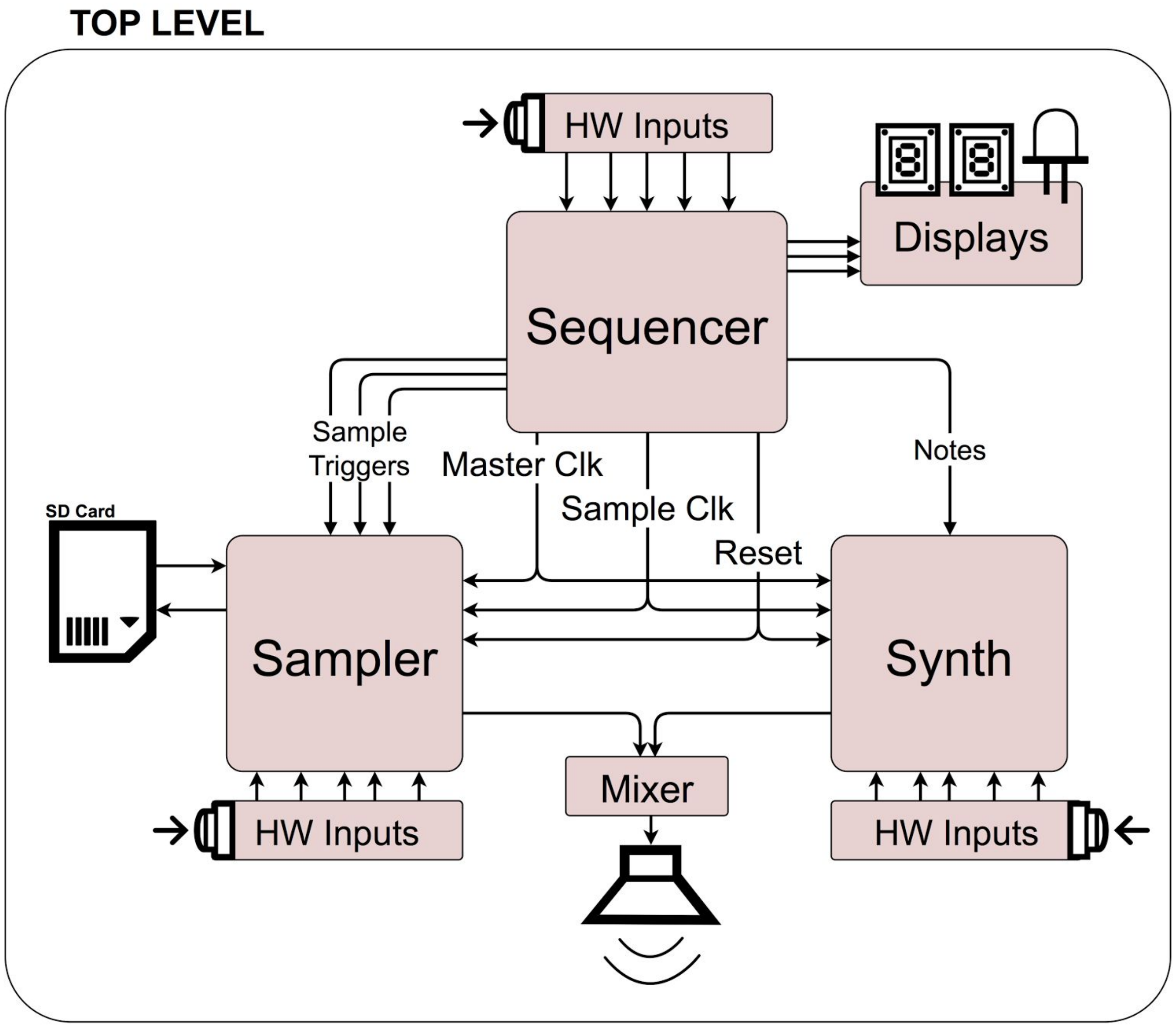
I proposed implementing more than one synthesis channel. With the modular structure of the synthesizer, this would mostly just involve instantiating another set of each synth module. The one exception is the FFT module, which occupies too much area to be duplicated. However, in the filter_coeff_gen module, I already buffer the output of the FFT in BRAM. Since this impulse response data is buffered, one FIR filter module could continue to access the coefficients while the FFT is initiated on a different computation. In this way, the single FFT module could easily be shared between multiple synth channels to generate multiple sets of frequency response filter coefficients.

I also proposed emulating a typical analog lowpass filter (adjustable resonant peak at the cutoff frequency, finite roll-off slope in the stopband), while my final design modeled an ideal window-type frequency response. Again, I don't think this goal is far off from where I left the code. Other filter response characteristics would still use the same interface to the FFT module, and the same FIR filter module to apply them to a signal, so most of the framework is already in place to implement different filter types.

My analog_input_handler module was also supposed to implement multi-purpose knobs (e.g. the waveform of all three oscillators controlled by a single potentiometer). Like in the filter modules, the existing input handler code is built in a way that can accommodate further expansion. The potentiometer mapping operation is parameterized, so extended input functionality should only require some additional states and transitions in the input handler FSM.

Top-Level Integration

The top-level structure of FPDJ would be very similar to our original proposal.



The sequencer would interface to the sampler via a bus of sample trigger wires. The synth would be controlled by a note value bus. The sampler and synth would update their outputs on a common 44.1kHz sample clock, and their signals would be mixed in a master mixer before being outputted to the on-board PWM output filtering stage.

Conclusion

The biggest shortcoming of this project was the incomplete final integration stage. Steps to begin integrating the three modules with each other should have happened much earlier. Time was likely the only real obstacle in putting together a complete, functional FPDJ device as proposed.

Each module demonstrated functionality meeting almost all minimum commitments, and many central goals. The sequencer offered an interface for live manipulation of a multi-track musical sequence. The sampler provided support for loading arbitrary sound clips from an SD card, and playing back samples on command. The synthesizer presents a subtractive synthesis architecture tweakable in real-time to produce a wide range of interesting timbres. These components were all built with the proper interfaces to be interconnected. Additionally, the three modules combined would not end up near limitations of FPGA area or I/O, based on individual implementation results.

Yet, in the end, we failed to bring it all together into one flexible, fun device. But we all enjoyed the challenges this project presented to us, and we intend to continue work on the FPDJ system on our own time to bring it up to our original objectives.

Appendix

The Verilog code for this project, including top-level modules for each central module (Sequencer, Samper, Synth) individually, can be found on the 6.111 course website.