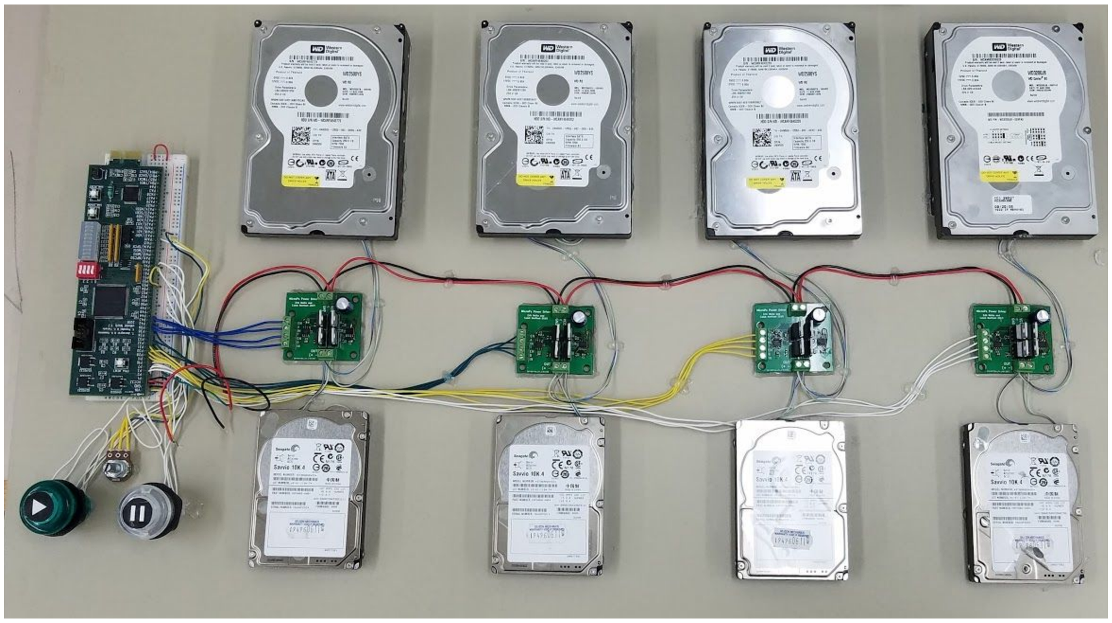
Hard Drive Rock

E155 Final Project Report

December 13th, 2019

Erik Meike and Caleb Norfleet



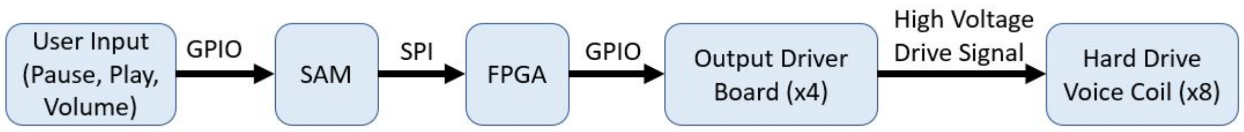
Abstract:
Unconventional methods for music generation have recently become more popular and have been featured in a number of viral YouTube videos such as Wintergartan's "Marble Machine". For our project, we proposed an unconventional method of creating music involving driving the actuators in hard drives to create musical tones. We created a four-track music playing system that can be controlled via play and pause buttons and a volume knob. We also created an output driver board PCBA for delivering high-power signals to the hard drives. We have successfully demonstrated the ability to play several multi-track songs using our music generator system and also characterized the output stage of our music generator as 97.5±1.0% efficient.

I. INTRODUCTION

Alternative music generation techniques have recently become more popular as microcontrollers have become more available to the general public. In 2016, Wintergatan released the single “Marble Machine”[1] which used unconventional mechanical techniques triggered by marbles to create a song. Their YouTube video became quite popular and has over 117 million views. Another YouTube video features a machine called the “floppotron” which used 64 floppy disks and scanner motors to create music and is more similar to the method we propose[2].

For our project, we proposed an unconventional method of creating music involving driving the actuators in hard drives to create musical tones. The user interacts with our music generator via a simple user interface which enables them to play or pause the song and set the volume. The ATSAM determines when to play notes based on this input and communicates to the FPGA over SPI which notes should be played at what volume. The FPGA then generates a note signal based on this information. These note signals are delivered to output driver boards which provide a high voltage and high current signal to the voice coil in our hard drives, causing them to move at the desired audio frequency.

Figure 1: High-Level Block Diagram of Music Player System


User Input (Pause, Play, Volume)
GPIO
SAM
SPI
FPGA
GPIO
Output Driver Board (x4)
High Voltage Drive Signal
Hard Drive Voice Coil (x8)

II. MICROCONTROLLER

The user interacts with our music player system via a pair of buttons which allow them to play or pause the music. In addition, a knob (potentiometer) allows for the user to set the volume level. The ATSAM reads in these inputs using GPIOs and the ADC peripheral.

Songs are hardcoded in the form of arrays of notes with a frequency and a duration. One such array is used for each track within the song, and up to four tracks are supported by our system. Only one song is stored in the ATSAM at a time due to memory constraints, but different songs can be easily `#include`-ed at the top of the source code in order to play different music. A workflow for easily converting MIDI music files into note and duration arrays which can be parsed by our code was developed based around an open source program created to accomplish this task for the Arduboy game system[3]. We’ve successfully used this workflow to convert MIDI files for around a dozen songs to play on our system.

[1] https://www.youtube.com/watch?v=IvUU8joBb1Q
[2] https://www.youtube.com/watch?v=Oym7B7YidKs
[3] https://github.com/MLXXXp/midi2tones

At startup, the ATSAM runs a number of initialization functions to set up the GPIO, SPI, ADC, and timer/counter peripherals. It also initializes the song tracks and sets its current status to paused. Progression through the song for each track is recorded in variables which indicate the index of the current note and the amount of time remaining in that note. Another variable keeps track of the amount of time until the next note change on any of the tracks.

After initialization, the ATSAM enters its main loop. In this loop it checks the values of the input signals and uses these readings to update the current volume (which is applied to all the tracks) and the song status (paused or not paused). If the music player is not paused, the ATSAM uses the timer/counter to iterate through the song based on the minimum amount of time until the next note change on any of the tracks. If this amount of time has passed, the ATSAM updates all the tracks appropriately. The end of each track is indicated by a note with a duration of negative one (individual tracks are allowed to end at different times), and rests are indicated by notes with a frequency of zero. Once all tracks in the song have ended, all of the tracks are reset and the status is set to paused such that the user can press the play button to hear the song again.

Every time that the notes being played or the current volume changes, the ATSAM communicates the new note and volume status to the FPGA. For each note, the microcontroller computes an associated 16-bit "tune word" (explained further in Section III). It also determines an associated 8-bit volume level. Finally, it communicates the tune word and volume level to the FPGA using SPI. In order to make it easier to implement the SPI slave module on the FPGA, we decided on a specific methodology for using SPI. First, the ATSAM sets the clock select pin to high (we decided to use an alternative pin for our clock select in order to control it more easily). Next, the ATSAM sends three 8-bit packets of data for each track (the most significant byte of the tune word, then the least significant byte of the tune word, and then the volume level byte) using the "spiSendRecieve" command from the ATSAM libraries (with a polarity of zero and a phase of one). Finally, the ATSAM sets the clock select pin to low to finish the transaction. Any SPI transactions which do not involve exactly 24 bits per track (96 bits for four tracks) between when clock select is set high and low will be disregarded by the FPGA. We decided on this methodology in order to make it easy to avoid issues with dropped packets leading to data being misinterpreted as the wrong type of information (tune word vs. volume level) without adding the overhead of directly communicating which sort of information was being sent in each packet.

III. FPGA

Figure 2A: Block Diagram of SystemVerilog Deployed to FPGA

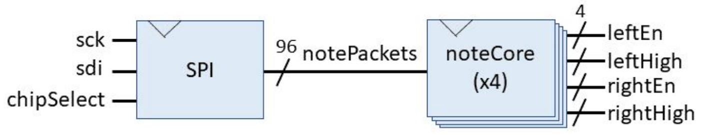
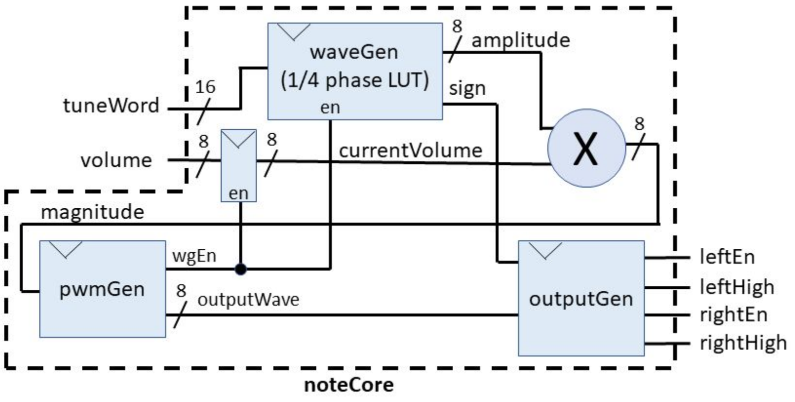


Figure 2B: Block Diagram of "noteCore" Waveform Generator for One of the Four Tracks



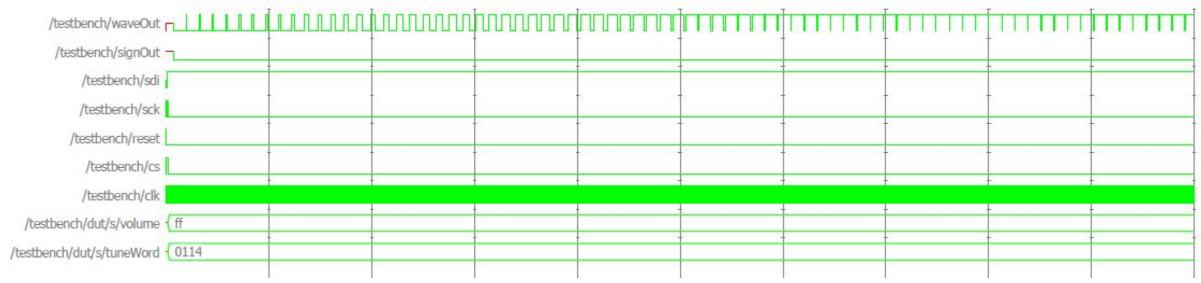
The first element of our FPGA block diagram is an SPI block which handles receiving data from the SAM. This block is particularly interesting because it contains an asynchronous interface between two clock domains: the FPGA's 40MHz clock (clk) and the SPI clock (sck). In order to avoid issues with this interface, we constrained the crossing between those two domains to a single interaction in which the data read in from the SPI (in the sck domain) is copied to a register in the clk domain. This interaction is constrained to only occur if chipSelect is low, and all of the registers in the sck domain are constrained to only update if chipSelect is high. In addition, from the SAM we ensure that a small amount of time passes in between when chipSelect changes value and when sck is active. Thus, we can be confident that none of the registers in the sck should be enabled at any point where they are being read from the clk domain.

The signal generation stage is designed to be as general purpose as possible. It is capable of resampling and outputting any repeating arbitrary waveform at nearly any repetition rate. The core is a look-up table (LUT) which is used to source the appropriate output amplitude for any given time. The LUT is currently configured to represent the first quadrant of the output waveform. The remaining quadrants are produced by sampling the LUT backwards and/or inverting the sign of the output. Samples are generated whenever the PWM generator creates an interrupt asking for a new value for the next PWM cycle (which occurs every 2^8 clock cycles). The output wave signal is eight bits plus a sign bit.

The PWM generator produces a PWM output based off of the result from the waveform generator. The core is an eight bit counter. This counter is allowed to run continuously and overflows every 2^8 input cycles. The output pin is asserted whenever the current timer value is less than or equal to the signal generator's output amplitude. Every time the counter overflows, an interrupt is triggered and sent to the signal generator to request an amplitude for the next cycle. Finally, the PWM output and the sign of the output from the waveform generator is used to determine the correct output signals to the output driver boards for each track (see section IV).

A testbench was also created to verify the functionality of the SystemVerilog design for the FPGA. Waveforms were generated using ModelSim and it was found that the output waveform and sign bits matched expected values.

Figure 3: Testing the FPGA Design Using ModelSim

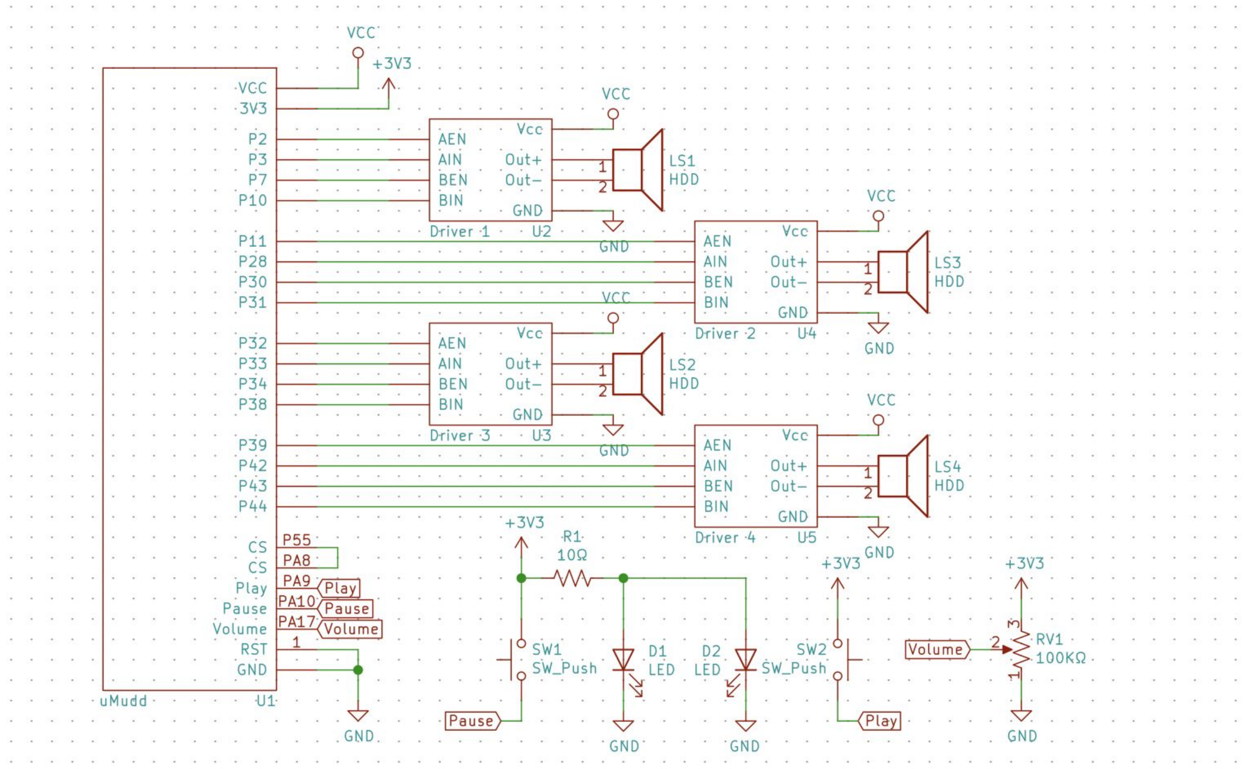


IV. NEW HARDWARE

Several HDDs were obtained and disassembled to get direct access to the read/write head coil. The internal connections were traced such that all connections could be made with the externally available pins. The remaining HDDs were connected by removing the controller and connecting to the appropriate outer chassis contacts. An H-bridge was chosen as the output stage due to the increased power handling capabilities relative to our proposed alternative solutions. The H-bridge was created out of N-channel MOSFETS due to their superior current carrying capabilities. This means the high-side FET requires a voltage above V_{in} to turn on. An integrated FET driver with charge pump was used to produce the higher voltages. Designing an H-bridge also comes with a new set of issues to resolve such as ensuring that both FETs on either side of the push-pull driver are not on at the same time. The fet driver was chosen to have the required dead time to ensure that both FETs are never on at the same time.

A PCB was created to ease the assembly of the final product. The driver array was separated into a single driver per PCB to allow for modularity in the final product. A single power supply was chosen to keep the wiring as simple as possible since each PCB requires 4 control wires. The traces were designed to handle 4 amps continuously without overheating to ensure the PCB can not overload and has a large safety factor when driving an 8Ω load off of 5 volts.

Figure 4: Schematics of Circuitry



V. RESULTS

The final system is capable of playing back a pre-programmed song hardcoded in note and duration format. This format can be easily generated from a MIDI file by selecting the tracks desired and using an open-source program[4]. The final UI has play, pause, and volume input. One notable difference between the initial proposal and final project was that we decided to only have one song longer song with four tracks loaded onto the system at a time (instead of several shorter songs) due to memory constraints on the ATSAM.

The final PCB driver was tested to measure the efficiency of the output drivers. This was not listed as a requirement in the initial proposal, however it seemed to provide a significant impact on the scalability of the project, so it was considered for the final design. To measure the efficiency of the PCB component, it was isolated from the remainder of the project. The output was connected to an 8Ω resistor to make output power calculations easier and match the typical audio output efficiency testing standard. The voltage across the resistor was then measured with

[4] https://github.com/MLXXXp/midi2tones

an oscilloscope, and later a high speed benchtop multimeter to calculate power. The resistance of the resistor was measured to be 8.357Ω by using a 4 wire resistance measurement since this has a large effect on the calculated final output power. The final resulting efficiency of the output driver was 97.5%±1% at an output power of 1.69 watts. The quiescent current was 7.29mA when the output was disabled.

Appendix A: Bill of Materials

Part	**Quantity**	**Link**	**Cost**
0.33uF 25v 0805 capacitor	16	https://www.digikey.com/product-detail/en/samsung-electro-mechanics/CL21B334KAFNNNE/1276-1808-1-ND/3889894	$0.17
100uF 25v capacitor	4	HMC Makerspace	n/a
SD0805S040S0R5 schottky diode	24	https://www.digikey.com/product-detail/en/avx-corporation/SD0805S040S0R5/478-7802-1-ND/3749494	$0.44
4-pin screw terminal	4	https://www.digikey.com/product-detail/en/te-connectivity-amp-connectors/282837-4/A124423-ND/2187975	$1.58
2-pin screw terminal	8	https://www.digikey.com/product-detail/en/te-connectivity-amp-connectors/282837-2/A113320-ND/2187973	$1.04
AOT2618L N channel MOSFET	16	https://www.digikey.com/product-detail/en/alpha-omega-semiconductor-inc/AOT2618L/785-1438-5-ND/3603378	$0.797
LM27222 MOSFET driver	8	https://www.digikey.com/product-detail/en/texas-instruments/LM27222MX-NOPB/296-35268-1-ND/3738976	$2.402
PCB	4	jlcpcb.com	$19.11 (for ten)
3.5” HDD	4	Engineering server graveyard	n/a
2.5” HDD	4	Engineering server graveyard	n/a
Total cost:			$79.00

Appendix B: C Code for Microcontroller

```c
// finalProject.c
// cnorfleet@hmc.edu, emeike@hmc.edu
// 15 November 2019
//
// Sends song notes and volumes to FPGA over SPI
// Note: store song pitch in Hz, dur in ms

#include <stdio.h>
#include <math.h>
#include "SAM4S4B_libraries/SAM4S4B.h"
#include "ArduboyTonesPitches.h"
#define TONES_END -1

// Song to play:
#include "Songs/4channel/onTopOfTheWorld.c"
#define NUM_TRACKS 4
const int* tracks[NUM_TRACKS] = { &(score1[0]), &(score2[0]), &(score3[0]), &(score4[0]) };

#define CHIP_SELECT_PIN PIO_PA8 // connected to Pin 55 on FPGA
// SPCK: PA14 -> P113
// MOSI: PA13 -> P112
// MISO: PA12 -> P111
// NPCS0 (not used): PA11 -> P110
#define PAUSE_PIN PIO_PA10
#define PLAY_PIN PIO_PA9

#define LED0 PIO_PA0
#define LED1 PIO_PA1
#define LED2 PIO_PA2
#define LED3 PIO_PA29
#define LED4 PIO_PA30
#define LED5 PIO_PA5
#define LED6 PIO_PA6
#define LED7 PIO_PA7

#define VOLUME_CH ADC_CH0 // volume selected with ADC CH 0 (PIN PA17)

#define CH_ID  TC_CH0_ID
#define CLK_ID    TC_CLK5_ID
#define CLK_SPEED TC_CLK5_SPEED

unsigned int idx[NUM_TRACKS];
uint16_t currentTuneWord[NUM_TRACKS];
uint8_t  currentVolume = 0b11111111;
int remainingDur[NUM_TRACKS]; // time in ms until next note change per track
int currentDur = 0;           // minimum time in ms until next note change
char bytes[NUM_TRACKS*3];     // byte data to send to FPGA over SPI
char paused = 1;              // indicates whether the song is currently paused

uint16_t getTuneWord(int pitch);
int getMinDur(void);
void updateTrackArray(int track);
void initTrackArrays(void);
void restartSongTracks(void);
char isAllRests(void);
char isStillPlaying(void);
void progressNotes(int timePassed);
void updateVolume(void);
void updateBytes(int track);
void updateAllBytes(void);
void updateAllBytesForPaused(void);
void sendNotes(void);
```

```
int main(void) {
	// Initialize:
	samInit();
	pioInit();
	adcInit(ADC_MR_LOWRES_BITS_10);
	adcChannelInit(VOLUME_CH, ADC_CGR_GAIN_X1, ADC_COR_OFFSET_OFF);
	spiInit(MCK_FREQ/244000, 0, 1);
	// ^ "clock divide" = master clock frequency / desired baud rate
	// the phase for the SPI clock is 0 and the polarity is 0
	tcDelayInit();
	pioPinMode(CHIP_SELECT_PIN, PIO_OUTPUT);
	pioPinMode(PAUSE_PIN, PIO_INPUT);
	pioPinResistor(PAUSE_PIN, PIO_PULL_DOWN);
	pioPinMode(PLAY_PIN, PIO_INPUT);
	pioPinResistor(PLAY_PIN, PIO_PULL_DOWN);

	pioPinMode(LED0, PIO_OUTPUT);
	pioPinMode(LED1, PIO_OUTPUT);
	pioPinMode(LED2, PIO_OUTPUT);
	pioPinMode(LED3, PIO_OUTPUT);
	pioPinMode(LED4, PIO_OUTPUT);
	pioPinMode(LED5, PIO_OUTPUT);
	pioPinMode(LED6, PIO_OUTPUT);
	pioPinMode(LED7, PIO_OUTPUT);

	// Get ready to play song:
	tcDelay(1); // allow for stuff to start up
	restartSongTracks();

	// Play song:
	while (1) {
		if(!isStillPlaying()) { // stop playing at end of song
				paused = 1;
				for(int i = 0; i < TONES_END; i++) {
					currentTuneWord[i] = 0;
					remainingDur[i] = -1;
					updateBytes(i);
				}
				sendNotes();
				restartSongTracks();
		}
		updateVolume(); // display current volume on LEDs even if paused
		if(!paused) {
				tcDelay(currentDur);
				progressNotes(currentDur);
				sendNotes();
		}
		if(paused && pioDigitalRead(PLAY_PIN)) // resume playing
				paused = 0;
		else if(!paused && pioDigitalRead(PAUSE_PIN)) { // pause
				updateAllBytesForPaused();
				sendNotes();
				paused = 1;
		}
	}
}

uint16_t getTuneWord(int pitch) {
	// note: tuneWord of 1 corresponds to 2.384 Hz = ((40MHz)/2^8)/2^16
	uint16_t tuneWord = pitch / 2.38418579;
	return tuneWord;
}
```

```c
int getMinDur(void) {
        int minDur = tracks[0][2*idx[0]+1];
        for(int i = 1; i < NUM_TRACKS; i++) {
                 if((minDur == -1) || ((remainingDur[i] != -1) && (remainingDur[i] < minDur))){
                                minDur = remainingDur[i];
                }
        }
        return minDur;
}

void updateTrackArray(int track) {
        remainingDur[track] = tracks[track][2*idx[track]+1];
        currentTuneWord[track] = getTuneWord(tracks[track][2*idx[track]]);
}

void initTrackArrays(void) {
        for(int i = 0; i < NUM_TRACKS; i++) {
                idx[i] = 0;
                updateTrackArray(i);
        }
        currentDur = getMinDur();
}

void restartSongTracks(void) {
        initTrackArrays();
        while(isAllRests()) {
                progressNotes(getMinDur()); // skip rests at start
        }
}

char isAllRests(void) {
        for(int i = 0; i < NUM_TRACKS; i++) {
                if(currentTuneWord[i] != 0) {
                                return 0;
                }
        }
        return 1;
}

char isStillPlaying(void) {
        return (currentDur != -1);
}

void updateBytes(int track) {
        uint8_t tune_word_byte_1 = currentTuneWord[track] >> 8;
        uint8_t tune_word_byte_2 = currentTuneWord[track];
        uint8_t volume_byte = (currentTuneWord[track] == 0)
                                          ? 0b00000000 : currentVolume; // pitch 0 is rest

        bytes[3*track]   = tune_word_byte_1;
        bytes[3*track+1] = tune_word_byte_2;
        bytes[3*track+2] = volume_byte;
}

void updateAllBytes(void) {
        for(int i = 0; i < NUM_TRACKS; i++)
                updateBytes(i);
}

void updateAllBytesForPaused(void) {
        for(int i = 0; i < NUM_TRACKS*3; i++)
                bytes[i] = 0b00000000;
}
```

```
void progressNotes(int timePassed) {
	// update tracks after timePassed (in ms)
	for(int i = 0; i < NUM_TRACKS; i++) {
		if(remainingDur[i] == -1) {
				currentTuneWord[i] = 0;
				remainingDur[i] = -1;
				updateBytes(i);
				continue;
		}
		remainingDur[i] = remainingDur[i] - timePassed;
		if(remainingDur[i] <= 0) { // continue to next note
				int lastRemainingDur = remainingDur[i];
				idx[i] = idx[i] + 1;
				if(tracks[i][2*idx[i]] == -1) { // at the end of this track
					currentTuneWord[i] = 0;
					remainingDur[i] = -1;
				} else {
					updateTrackArray(i);
					remainingDur[i] = remainingDur[i] + lastRemainingDur;
					// ^ if we've gone too far, subtract from next
				}
				updateBytes(i);
		}
	}
	currentDur = getMinDur();
}

void updateVolume(void) {
	// measure voltage from pin and convert to value between 0 and 1
	float voltage = adcRead(VOLUME_CH);
	double volumeScale = voltage / 3.3;
	volumeScale = (volumeScale > 1) ? 1 : volumeScale;
	volumeScale = (volumeScale < 0) ? 0 : volumeScale;

	uint8_t newVolume = (int) (round(volumeScale * 0b11111111));
	if(newVolume != currentVolume) {
		currentVolume = newVolume;
		updateAllBytes();
	}
	pioDigitalWrite(LED0, (currentVolume)      & 1);
	pioDigitalWrite(LED1, (currentVolume >> 1) & 1);
	pioDigitalWrite(LED2, (currentVolume >> 2) & 1);
	pioDigitalWrite(LED3, (currentVolume >> 3) & 1);
	pioDigitalWrite(LED4, (currentVolume >> 4) & 1);
	pioDigitalWrite(LED5, (currentVolume >> 5) & 1);
	pioDigitalWrite(LED6, (currentVolume >> 6) & 1);
	pioDigitalWrite(LED7, (currentVolume >> 7) & 1);
}

void sendNotes(void) {
	// assert chipSelect
	// for each track:
	//   shift in frequency in two bytes
	//   shift in volume in one byte
	// deassert chipSelect
	pioDigitalWrite(CHIP_SELECT_PIN, 1);
	for(int i = 0; i < NUM_TRACKS*3; i++) {
		spiSendReceive(bytes[i]);
	}
	pioDigitalWrite(CHIP_SELECT_PIN, 0);
}
```

Appendix C: SystemVerilog for FPGA

```
// finalProject.sv
// Erik Meike and Caleb Norfleet
// FPGA stuff for uPs final project

`define NUM_TRACKS 4   // number of tracks (and tone generators) used
`define PACKET_SIZE 24 // bits of data per track in each packet
typedef logic[`PACKET_SIZE-1:0] packetType;

module top(input  logic                  clk, reset,
           input  logic                  chipSelect, sck, sdi,
           output logic[`NUM_TRACKS-1:0] leftHigh, leftEn, rightHigh, rightEn);

       packetType[`NUM_TRACKS-1:0] notePackets;

       spi s(clk, reset, chipSelect, sck, sdi, notePackets);

       noteCore nc[`NUM_TRACKS-1:0](
        .clk          ( clk ),       // single bit replicated across instance array
        .reset        ( reset ),
        .notePacket ( notePackets ), // connected logic wider than port so split across
                                                                    instances
        .leftHigh   ( leftHigh ),
        .leftEn       ( leftEn ),
        .rightHigh  ( rightHigh ),
        .rightEn      ( rightEn )
       );

endmodule

module noteCore(input  logic      clk, reset,
                input  packetType notePacket,
                output logic      leftHigh, leftEn, rightHigh, rightEn);
       // tone generator for one track

       logic[15:0] tuneWord;   // frequency of note signal
       logic[7:0]  volume;     // unsigned volume of output
       logic          sign;    // note signal sign
       logic[7:0]  amplitude;  // note signal amplitude
       logic[7:0]  currentVol; // volume only updated after every 2^8 clock cycles
       logic[7:0]  magnitude;  // amplitude of wave after multiplying with volume
       logic          waveOut; // output signal, PWM at 40MHz to get amplitude at 156.25 kHz
       logic          wgEn;    // interrupt to request next amplitude from waveGen

       assign tuneWord = notePacket[23:8];
       assign volume   = notePacket[ 7:0];

       waveGen wg(clk, reset, wgEn, tuneWord, sign, amplitude);

       always_ff @(posedge clk) begin
               if (reset)      currentVol <= 8'b0;
               else if(wgEn) currentVol <= volume;
       end

       logic[15:0] mult;
       assign mult = ({8'b0, amplitude} * {8'b0, currentVol});
       assign magnitude = (mult[7] & ~&mult[15:8]) ? (mult[15:8] + 8'b1) : (mult[15:8]);
       // ^ note: rounding with saturation

       pwmGen pg(clk, reset, magnitude, wgEn, waveOut);

       outputGen og(clk, reset, waveOut, sign, leftHigh, leftEn, rightHigh, rightEn);

endmodule
```

```
module waveGen(input  logic       clk, reset, wgEn,
               input  logic[15:0] tuneWord,
               output logic       sign,
               output logic[7:0]  amplitude);
        // generates sinusoid based on tuneWord
        // only changes frequency at end of wave (every other zero crossing)

        logic[15:0] phaseAcc;          // phase accumulator
        logic[7:0]  LUTsine[(2**10-1):0]; // look up table
        logic[15:0] currentTuneWord;

        logic nextSign;
        logic[9:0] nextPhase;
        assign nextSign = phaseAcc[15]; // neg in second half
        assign nextPhase = (phaseAcc[14]) ? (10'b0 - phaseAcc[13:4]) : (phaseAcc[13:4]);
        // ^ note that phase is adjusted since we're using a 1/4 phase LUT

        always_ff @(posedge clk) begin
                if(reset) begin
                        phaseAcc       <= 16'b0;
                        currentTuneWord <= 16'b0;
                        amplitude      <= 8'b0;
                        sign           <= 1'b0;
                end
                else if(wgEn) begin
                        if((tuneWord != currentTuneWord) & ((~sign & nextSign) |
                                                          (currentTuneWord == 16'b0))) begin
                                currentTuneWord <= tuneWord;
                                phaseAcc       <= 16'b0;
                                amplitude      <= 8'b0;
                                sign           <= 1'b0;
                end else begin
                                phaseAcc  <= phaseAcc + currentTuneWord;
                                amplitude <= LUTsine[nextPhase];
                                sign    <= nextSign;
                        end
                end
        end

        initial begin
                $readmemb("LUTsine.txt", LUTsine);
        end

endmodule

module pwmGen(input  logic      clk, reset,
              input  logic[7:0] magnitude,
              output logic      wgEn,
              output logic      waveOut);
        // modulates carrier signal based on sine wave

        // wave gen runs at 156.25 kHz = 40MHz / 256 (aka 2^8)
        logic[7:0] waveCounter;
        always_ff @(posedge clk) begin
                if(reset)  waveCounter <= 8'b10000000;
                else    waveCounter <= waveCounter + 8'b1;
        end
        assign wgEn = (waveCounter == 8'b0);

        always_ff @(posedge clk) begin
                waveOut <= (~reset & (waveCounter < magnitude));
                // PWM carrier by magnitude
        end
endmodule
```

```
module outputGen(input  logic clk, reset,
                 input  logic waveOut, sign,
                 output logic leftHigh, leftEn, rightHigh, rightEn);
        // generates FET driver signals based on sign and output wave

        always_ff @(posedge clk) begin
                if(reset) begin
                        leftHigh  <= 1'b0;
                        leftEn  <= 1'b0;
                        rightHigh <= 1'b0;
                        rightEn   <= 1'b0;
                end else begin
                        leftEn    <= 1;
                        rightEn   <= 1;
                        leftHigh  <= ( sign)&waveOut; //  sign^waveOut
                        rightHigh <= (~sign)&waveOut; //~(sign^waveOut)
                end
        end

endmodule

module spi(input  logic clk, reset,
           input  logic chipSelect, sck, sdi,
           output packetType[`NUM_TRACKS-1:0] notePackets);
        // Accepts frequency and volume input over SPI from ATSAM
        // Internal freq and volume only updated after full packet received
        // Note: contains ~3.4 second watchdog timer (turns off music)

        // SPI interface protocol:
        //   assert chipSelect
        //   for each track (in order):
        //      shift in frequency in two bytes (MSB first)
        //      shift in volume in one byte
        //   deassert chipSelect

        logic[31:0] dataCount     = 32'b0; // amt of data in SPI packet so far
        logic       dataValid     =  1'b0; // indicates whether readData is good
        logic       dataValidCopy =  1'b0; // copied into clk domain
        logic[25:0] watchdogCounter;       // 2^27/40MHz = ~3.36 seconds %25:0
        logic       watchdogTriggered;

        logic[(`PACKET_SIZE*`NUM_TRACKS)-1:0] readData;     // data received over SPI
        logic[(`PACKET_SIZE*`NUM_TRACKS)-1:0] readDataCopy; // copied into clk domain
        logic[(`PACKET_SIZE*`NUM_TRACKS)-1:0] lastReadData; // memory for feeding watchdog

        always_ff @(posedge sck or negedge chipSelect) begin
                if(~chipSelect) begin
                         dataCount     <= 32'b0;
                end
                else begin
                         readData <= {readData[(`PACKET_SIZE*`NUM_TRACKS)-2:0], sdi};
                         dataCount <= dataCount + 32'b1;
                         if((dataCount + 32'b1) == (`PACKET_SIZE*`NUM_TRACKS))
                               dataValid <= 1'b1;
                         else  dataValid <= 1'b0;
                end
        end
        always_ff @(posedge clk) begin
                if(~chipSelect) begin // copy over from sck domain if cs is low
                         readDataCopy  <= readData;
                         dataValidCopy <= dataValid;
                end

                if(reset) begin
                         notePackets          <= {`NUM_TRACKS*`PACKET_SIZE{1'b0}};
                         watchdogCounter   <= 26'b0;
                         watchdogTriggered <=  1'b0;
                end else begin
                        if(&watchdogCounter & (readDataCopy == lastReadData)) begin
                                  watchdogTriggered <=  1'b1; // stop playing if watchdog
                                                                  counter at max val
                        end else begin
                        watchdogCounter <= watchdogCounter + 26'b1;
                        end
                        if(dataValidCopy) begin // if the packet is valid, update tracks
                               if(watchdogTriggered) begin // if triggered, don't play
                               notePackets <= {`NUM_TRACKS*`PACKET_SIZE{1'b0}};
                               end else begin // otherwise update tracks with current note
                               notePackets <= readDataCopy;
                               end
                               lastReadData <= readDataCopy;
                               if(~(readDataCopy == lastReadData)) begin
                               // if we've received a new packet, feed watchdog
                                       watchdogCounter   <= 26'b0;
                                       watchdogTriggered <= 1'b0;
                               end
                        end
                end
        end
endmodule
```

Appendix D: SystemVerilog Testbench

```
// testbench.sv
// Erik Meike and Caleb Norfleet
// Testbench for uPs final project

`define NUM_TRACKS 4   // number of tracks (and tone generators) used
`define PACKET_SIZE 24 // bits of data per track in each packet

module testbench();
        logic clk, reset, cs, sck, sdi;
        logic[`NUM_TRACKS-1:0] A, B, C, D;
        logic[(`PACKET_SIZE*`NUM_TRACKS)-1:0] packet;
        integer i;

        // device under test
        top dut(clk, reset, cs, sck, sdi, A, B, C, D);

        // test case
        initial begin
       if(`NUM_TRACKS == 1)
                packet <= 24'h0114ff;
       else
                packet <= 96'h0114ff0217ff0114ff0217ff;
        reset <= 1'b1; #22; reset <= 1'b0;
        end

        // generate clock signal
        Initial forever begin
        clk = 1'b0; #5;
        clk = 1'b1; #5;
        end

        initial begin
                i = 0; sck = 0;
                cs<=1'b1; #1; cs <= 1'b0; #23; cs <= 1'b1;
        end

        // shift in test vectors over SPI
        always @(posedge clk) begin
        if(~reset) begin
                if (i == 24*`NUM_TRACKS) cs = 1'b0;
                        if (i<24*`NUM_TRACKS) begin
                        #1; sdi = packet[(24*`NUM_TRACKS)-1-i];
                        #1; sck = 1; #5; sck = 0;
                        end
                        i = i + 1;
                end
         end
endmodule
```

Appendix E: Schematics