Multi-timbral Sound Module

6.111 Final Project

Valerie Gordeski
Susan Hwang
Chris Sheehan

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

December 9th, 2004

Abstract

The multi-timbral sound module is a music synthesizer that is able to generate three different instrument sounds. A user can choose to play a song from a keyboard using different voices (just by toggling a switch), to record a song from the keyboard entry and then to play it back, or to listen to two prerecording songs that use multiple instruments. Susan Hwang was responsible for the front end: the keyboard entry, the keyboard write/read to and from the RAM, and for the decoding unit that produces the appropriate pitch and play signals to the instrument. Chris Sheehan was responsible for creating the appropriate pitches using the inputs from Susan's module through interpolation, and for the integration of the entire overall system. Valerie Gordeski was responsible for taking the interpolated waveform and for shaping it to sound like the piano, violin and the flute, and to create a back-end mixer and volume control that would mix the digital waveforms and output the analog version of the result to the speaker. The hardest part of the project was the integration of the entire system - when the project takes up several FPGAs it has to be tested separately in little pieces, and when it is put together and little things go wrong it is extremely hard to know right away what is not working.

Table of Contents

Table of Figures

1. Introduction

The purpose of the multi-timbral sound module is to be able to play three different voices in a variety number of ways. In particular, this project includes a violin, piano and flute. The project allows for two different modes of direction from the user: real play from the user on the keyboard and play from a bank of songs. The keyboard real play only allows for playing from one instrument but the user may choose to play from 3 different instrument sounds. The songs contain instructions for all 3 instruments and can create a polyphonic sound. Table 1 is the overall diagram that shows the different modules that the sound module is composed of.

There are three main components to the creation of the project: the front end control, pitch modulation, envelope generation with volume control. The first FPGA contains the overall control of the instruments complete with the keyboard, button and switch interfacing. The front end module then passes the play and pitch signals to the interpolator. The interpolator takes these values and samples the instrument wave stored in the ROM according to the pitch value. This output is then passed on to the envelope generator which will shape the instrument waveform according to the instrument. This will be passed to the speaker to be
heard.

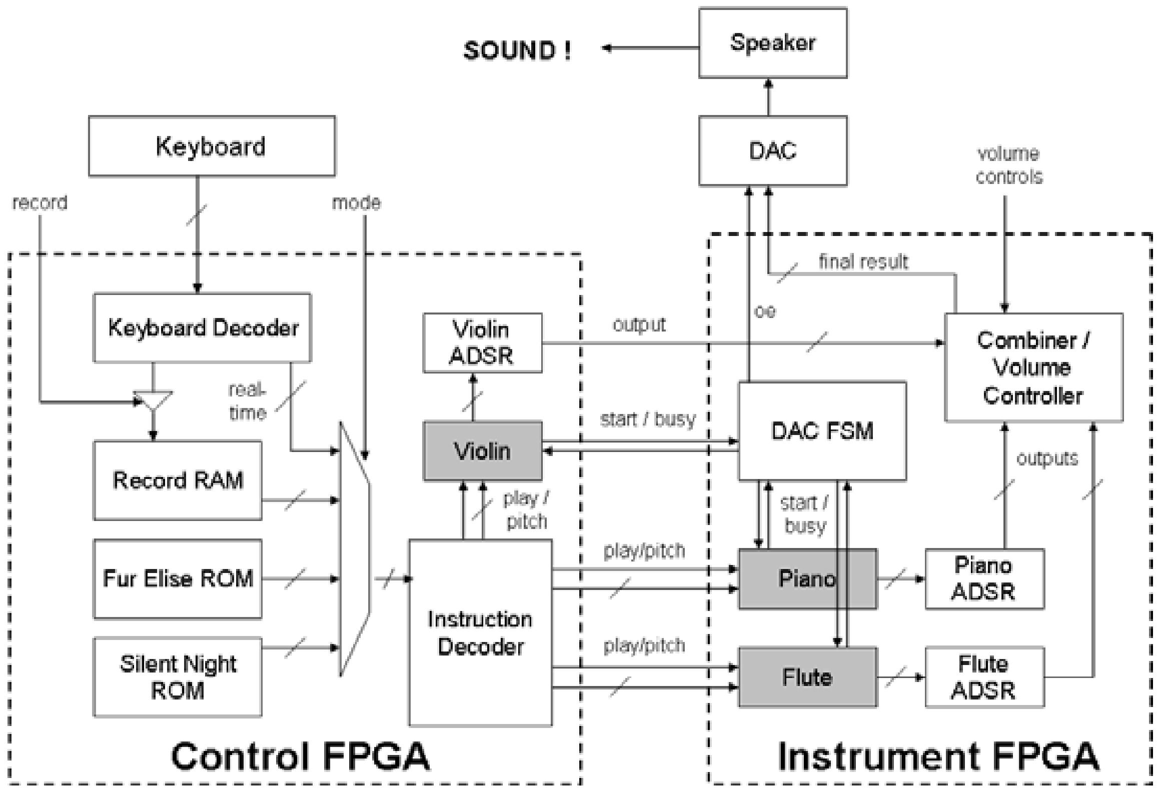


Figure 1: Overall Block Diagram

2. Font-End Input Control and Decoder(*by Susan Hwang*)

The purpose of this front-end module is to allow the user to control how the three instruments are played and act as a director. The user can choose from two songs, play from the keyboard, record from the record and playback. The front-end includes interfacing between external keyboard, switches, and buttons and the instruments. It allows recording of keyboard presses into a ram, blanking of the RAM, real-time playing of the keyboard, and playing from stored song ROMs.

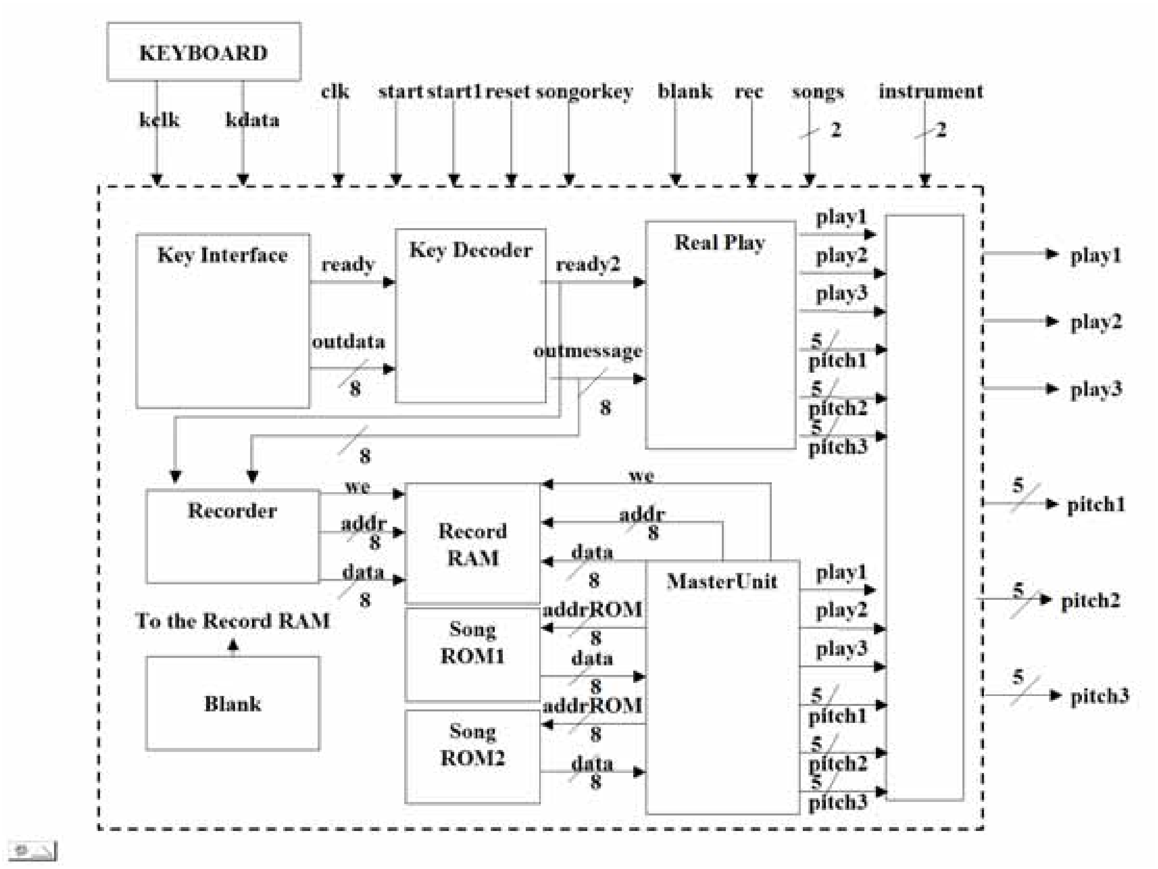


Figure 1: Overall Front-end Diagram

The clock comes from the 1.8 mHz crystal clock and the reset, start (which starts the keyboard interface), start1 (which starts the song play), and blank are button presses. Songorkey (which selects whether user wants to play from a song or a key), record, songs and instrument are all switches. Each instrument receives a play (1 bit) and pitch (5 bits) values from the output of the frontend module. This will control the playing of each of the instruments.

2.1 Decoder

2.11 Keyinterface

The purpose of the keyboard interface is to change the serial output of the keyboard into an 8-bit key code signal. The keyboard interface module takes in the keyboard clock, keyboard data, a reset, start, and outputs the corresponding 8-bit keyboard code and a ready signal that pulses high when all 8-bits of the keyboard code have been received.

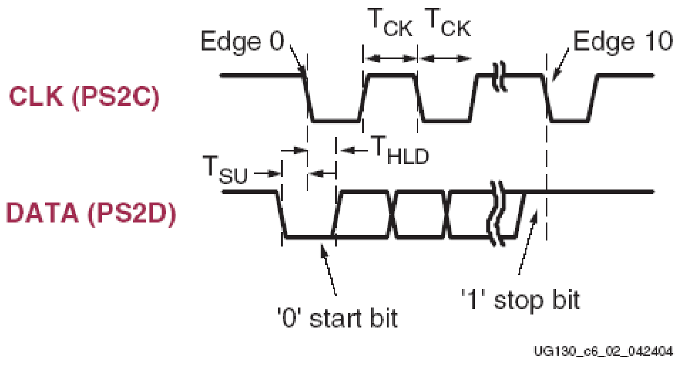


Figure 2: Keyboard PS/2 Bus Timing Waveforms

The keyinterface also has an additional embedded module called the lev2pulse module. Due to the nature of the kclk signal, the level to pulse module pulses on the negative edge of the kclk since the data is valid when the kclk is low. Figure 2 shows the keyboard clock and data outputs and their relation to each other. Therefore, the sample signal of the lev2pulse module will trigger the keyinterface to extract the data from the kdata signal. Figure 3 shows the collecting of 8 data bits from the keyboard. *Ready* is pulsed when all 8 bits have been collected.

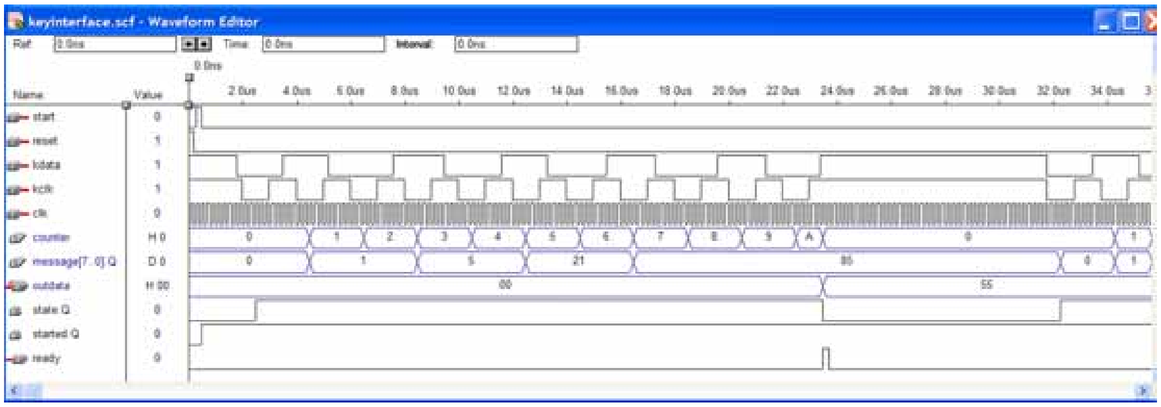


Figure 3: Simulation of Key Interface

The keyboard interface uses a simple two state finite state machine (FSM). It transitions to S_INITIAL with the reset button press. The S_INITIAL will transition to S_1 if sample goes high and the extracting of the data has started. Table 1 shows the keyboard data sequence. By the time it gets to S_1, the first bit has already been ignored. It will stay in S_1, until it has reached the count of 10. Meanwhile, the bits coming in will be stored to the corresponding 8-bit memory register. It only stores bit 1-9 into the message register and ignores the first and last 2 bits. When the count 10 has been reached, S_1 transitions to S_INITIAL to wait for the next sample and outputs the updated message register to the keydecoder as *outmessage* and pulses *ready*.

1	2	3	4	5	6	7	8	9	10	11
start	DATA								parity	Stop

Table 1: Keyboard Data Sequence

2.12 Keydecoder

The keydecoder module takes the key message from the keyinterface and decodes it into the music format that the other modules can recognize. The keydecoder also takes a 2-bit instrument signal which determines which instrument the messages are created for. There are two types of codes that are important to our music player: make code and break code. Each key has a make code which is usually 8-bits long. This indicates a that a key has been pressed. The break code ‘F0’ followed by a key’s break code indicated that the key has been released. Accurately detecting the make and break of a key is important because it determines the length a note is played.

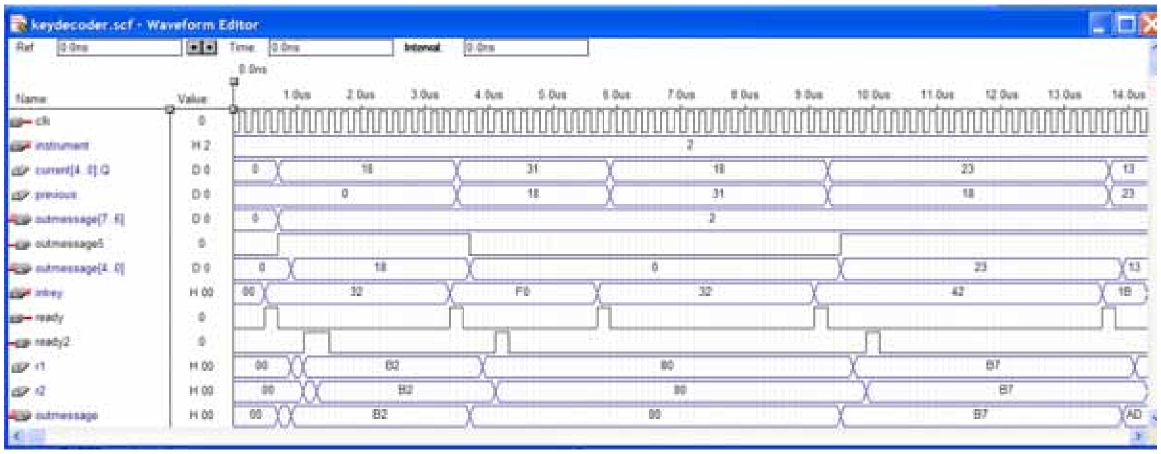


Figure 4: Simulation of Keydecoder

The module has two essential registers: previous and current. When the *outdata* signal is received and when *ready* is pulsed high, *outdata* is decoded into pitch values which are stored in the current register. Then current’s value is latched to previous register. The decoded messages are outputted in two cases. When the previous value is ‘F0’ in which case an *outmessage* of instrument turn off will be sent along with a pulsed *ready2* and when the previous value is not ‘F0,’ an *outmessage* will be set to the instruction “play instrument with pitch set on current register” with a pulsed *ready2* signal. Figure 4 shows a decoding of the *outmessage* taken from the output of the keyinterface. Table 2 shows how the key codes maps to the pitch codes and actual note values.

KEY CODE	PITCH	NOTE
15	0	LOW A
1E	1	A#
1D	2	B
24	3	C
25	4	C#
2D	5	D
2E	6	D#
2C	7	E
35	8	F
3D	9	F#
3C	10	G
3E	11	G#
1A	12	A HIGH
1B	13	A#
22	14	B
21	15	C
2B	16	C#
2A	17	D
34	18	D#
32	19	E
31	20	F
3B	21	F#
3A	22	G
42	23	G#
F0	31	BREAK

Table 2: The key-pitch-note values

2.13 Realplay

The realplay module takes the *outmessage* from the keydecoder module and parses the message into simple play and pitch values for the instruments. The first top 2 bits are taken out first to determine which instrument is playing. After it has determined which instrument is playing, the appropriate play1, play2, play3 are assigned from the 5th bit of the *outmessage* and the corresponding pitches are assigned from the 0-4 bits of *outmessage*. Figure shows how the realplayer takes the messages and translates it into play and pitch values for the instruments.

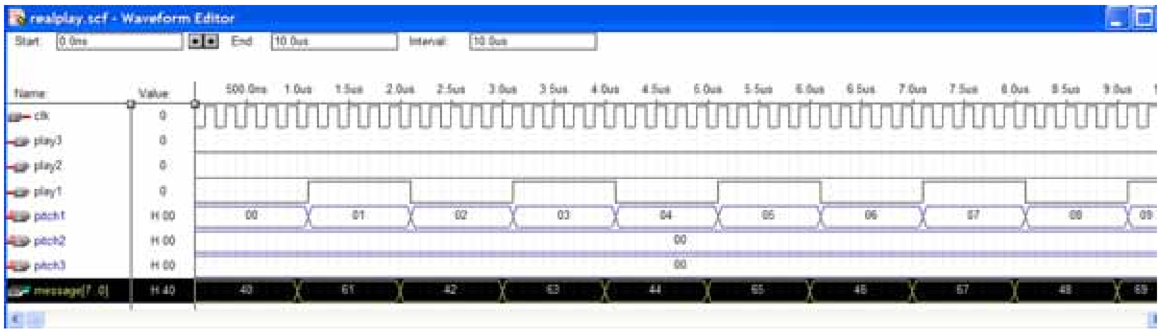


Figure 5: Real play simulation

2.14. Record

The purpose of the record module is to record the keyboard presses into the record RAM with proper musical encoding. The record module takes the *outmessage* of the keydecoder and stores that into the recordram. In addition to just storing the messages of the keycoder, it determines the length between each message and stores that as a special pass message into the RAM. The finished RAM is essentially identical to the form of the song ROMS complete with instrument instructions and pass length messages.

The record module also included an imbedded recordcounter module. The purpose of the recordcounter is to count how many "time-units" have passed between each message. This module takes in a *clk, startcounter, stopcounter* instructions from the record module and outputs a count. The record module in itself contains another embedded module called msdivider. This msdivider divides the 1.8mHz clock into 1/8 of a second. Therefore each count (bit increment) from the recordcounter means an increase of 1/8 of a second.

The record module includes a six state FSM. It transitions to S_INITIAL from a *reset* button press where the register values are initialized. It will transition out of S_INITIAL to S_1 when the *ready2* signal from keydecoder pulses high, indicating a new message is ready. S_1 sets data and address of the recordram to the appropriate values while keeping we low. This is setting up the ram for writing. It transitions to S_2 at the next clock edge. In S_2, we goes high and stores the data in the ram address location. S_2 also triggers the recordcounter to start counting by asserting *startcounter* high. Then S_2 transitions to S_3 at the next clock cycle. S_3 is essentially a wait state for the next message to come. When *ready2* pulses high, S_3 can transition to S_4 where the record module gets ready to store the special pass time message. S_4, the module stops the counter by asserting the *stopcounter* message. The RAM data register is set to the count value and the address is set to the next address value. At the next clock edge, S_4 transitions to S_5 and the pass message is stored into the RAM. At the next clock edge, S_4 transitions to S_6. S_6 is essentially a buffer state that keeps the data and address valid to make sure the correct values are stored into the RAM. At the next clock edge, S_6 transitions back to S_1 to store the next message. Figure 6 shows the recording of various musicdata inputs from the keydecoder and how the recorder interacts with the recordram.

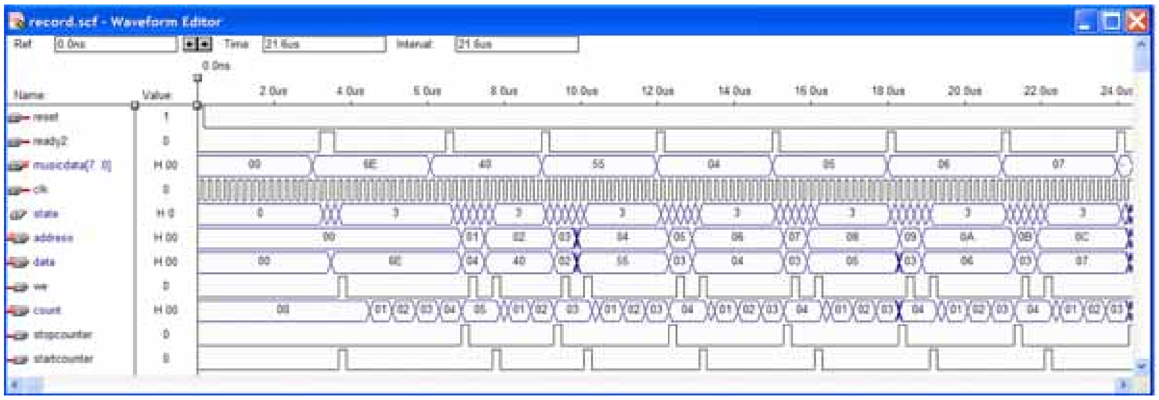


Figure 6: Record Simulation

2.15. Blank

The blank module blanks the recordram in case the user wants to reuse the ram to record something else. The blank module takes an external button press to begin writing the RAM.

The blank module has 4 states. When the external blank button press is first asserted, the state is set to 0 and the *busy* signal immediately goes high, signaling to the top module that the blanker is now accessing the RAM. S_0 is a buffer state to make sure the data and address have been steady for awhile. S_1, the value of 0 is written to the RAM, S_2 is another buffer state and in S_3, the address in incremented. This cycle continues until the address has reached the end of the RAM in which case *busy* signal goes low.

2.16. Masterunit

The masterunit is in charge of reading from the memory units and decoding the messages to control the three instruments. The masterunit takes in a *clock*, *reset*, *start*, and *songs* as external controls. The *start* tells the FSM to begin playing a song and the *songs* signal which is 2 bits long selects which memory unit to read from. In our implementation, song0 is the recordRAM, song1 is Fur Elise and song2 is Silent Night.

The masterunit has an embedded counter module that counts for a variable amount of time and signals to the masterunit when it has done counting with a pulsed *change* signal. The purpose of the counter is to tell when the masterunit can read from the next address in the memory units. This could determine note lengths of pause lengths between notes.

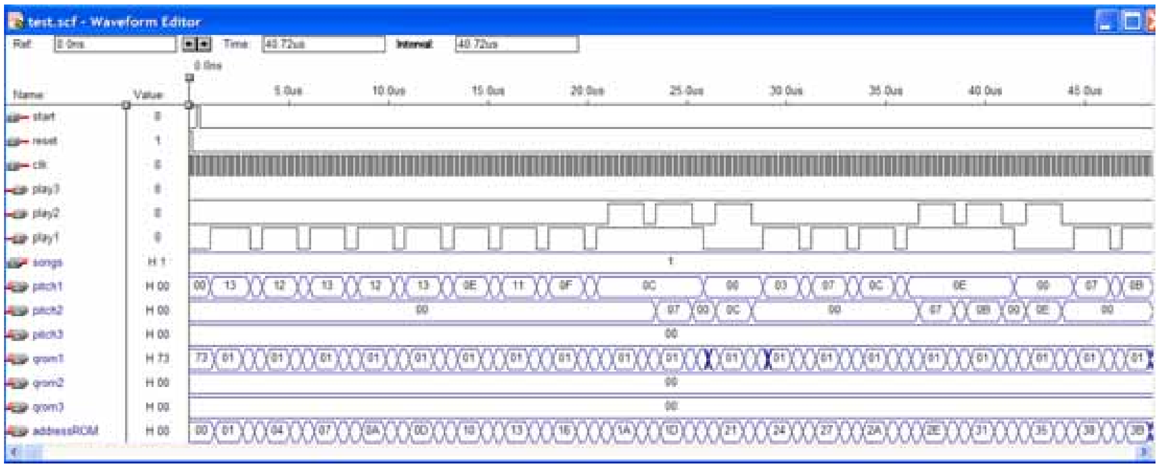


Figure 7: Simulation of Masterunit and memory units

The masterunit has 5 states. It first transitions to S_INITIAL with the *reset* press which initializes all the outputs and registers to 0. In S_INITIAL, if an external *start* button is pressed, then the masterunit transitions to S_1 to begin reading the song. S_1, the address for the memory unit is set and S_2 is the buffer state to hold the address steady. S_3 is when the message from the memory unit is decoded. A case statement is used to determine which type of message or instrument is being described. Any of '01,10,11' indicated that an instrument is being turned on or off so the corresponding play and pitch values should be correctly set. All of the instrument instructions will automatically transition back to S_1. A '00' which is a time pass message means that the next 6 bits is the count value and the counter must be started. Then the masterunit transitions to a special S_4. In S_4, the masterunit waits for the *change* signal from count, indicating that the message in the next address can be read. Figure 7 shows the simulation of playing from songrom1 which has the stored song Fur Elise.

The following is the message encoding that the masterunit uses to interpret the messages stored in the memory units. Each message is currently set at 8 bits long. The 2 MSB encodes the type of instruction, the next 1 bit encodes play(on/off), the next 5 encodes the pitch. However, a special instruction includes the pass message. The pass message, instead of having play, pitch information has the time value information in 6 bits describing how long to wait before reading the next message.

MSB (2)	00	01	10	11
Type	Pass	Voice 1	Voice 2	Voice 3

Table 4: Message Encoding

There are two types of Messages: Pass message and Instrument message.

00	Time Value (6 bits)

Table 5: Pass Message Encoding

Type/Instrument(2)	Note On/OFF(1)	Pitch Value (5bits)
01 (voice 1)	1 (on)	0 (lower C)
01	0 (off)	Nothing (ignored)

Table 6: Instrument Message Encoding

2.17. Frontend

The front-end module is a collection of the instances of the other modules. It interfaces between the external controls and the other modules. It also a contains a mux that controls which module controls the instrument play and pitch outputs. Depending on the value of *songorkey* the frontendcontrol selects either play and pitch signal from the masterunit or the realplay module. It also chooses which module controls the recordram. Depending on the *busy* signal from the blank module, *rec* and *songorkey*, it chooses which we, address and data will access the RAM.

2.2 Testing and Debugging

The testing and debugging was difficult for this project due to its complexity and the abundance of modules. However, there was a systemic method to testing the modules. After I finished coding each module, I compiled and caught the little syntax errors. I then simulated to make sure the module was doing what it was suppose to. I tested things very incrementally. After finishing two modules, I would connect the modules together and simulate it to make sure they were compatible and the chain would continue. An example of the steps I took was I first tested whether the keyboard worked. I then implemented the keyinterface module and checked whether the HEX LEDs would change on the kit when there is a button press. I then implemented keydecoder and connected and simulated keyinterface. After I made sure this worked, I added on realplay, recorder and so on. What was good about this testing is that I could make sure these parts worked incrementally. Since Chris had an instrument that played a square wave early on, I could hear when my part was working and when it wasn't working.

Debugging has very difficult for this project. It was that sometimes when instrument wasn't playing, it was hard to tell if it was the frontend or the instrument that had bugs. It is also hard to interface with other people's projects. For example, if the instruments are not configured correcting, the notes will sound off or it will not play altogether. The debugging also got very complicated when all the modules were put together and there are a million signal to look at and debug. Something that was also difficult was testing all three instruments. Since only two instruments fit on one FPGA, we had to put an instrument with the front end control and bus the input and output signals. Something very difficult is dealing with the keyboard. The keyboard is old and often the connections are not so good so sometimes the keyboard will fail and it is hard to distinguish whether it is a logic failure or hardware failure.

3. Instruments and Interpolation (Chris Sheehan)

The overall project was to create a sound module that can play songs from various inputs by sounding one or more unique instruments. My role in the project was to create the actual digital instruments that create the sound. While Susan worked on the front end of the system and Valerie worked on the backend digital signal processing, I worked on the "middle." My goal was to create a set of digital instruments that would take instructions from Susan's module and output digital signals to Valerie's module.

The instruments were to be digital in nature, so they could not use any natural analog waveforms. An obvious instrument design was to play a series of digital values stored in a ROM to recreate a waveform. Hooking these signals to a digital-to-analog converter (DAC) would allow the sound to be played over an analog speaker.

Each instrument had to be able to play many notes so that a variety of songs could be played. There were a few possibilities of how to create instruments with several pitches. One would be to have separate ROMs storing sampled waveforms of the specific instruments playing specific notes. This would unfortunately require a huge number of rather large ROMs, a memory requirement that is not conducive to FPGAs. Another possibility would be to have fewer ROMs and achieve different pitches by skipping through the values in the ROMs. If playing every value at a certain rate sounded an "A," then playing every other value at the same rate would sound an "A" an octave higher. To allow pitches between octaves to be played, the ROMs should be "oversampled," meaning that there would be great detail in the ROM values. By choosing every fifth or every sixth value in the ROM, for example, we could change the pitch slightly.

I determined that both of these proposed possibilities would not work well with the 6.111 lab kits because of the large memory requirement. To solve this problem, I use a technique called *interpolation* to shift the pitch of the single sample to produce other pitches.

3.1 Interpolation

Interpolation shifts the pitch of the sample by accessing the ROM data at different rates. If the stored sample is a "C," accessing it at twice the rate would produce a "C" an octave above, while accessing it at half the rate would produce a "C" an octave below. The normal "C" would result from reading from the ROM and incrementing the address once every sample period. The higher "C" would result form reading from the ROM and incrementing the address twice every sample period. Thus, we can change the pitch by incrementing the ROM address at different rates, even though the sample period is constant.

Changing the frequency by integer multiples is very easy, but this produces a range in notes that is too great for a conventional song. For example, a "D" is two half-steps above a "C." A given half-step is one-twelfth of an octave, or 1.059 times faster than the note below it. A "D" is (1.059)^2 times a "C" in frequency. If the stored sample in ROM is a "C," we need to increment the address 1.059^2 times every sample period.

The ROM clearly only allows integer address values, so the challenge is to interpret an address value like 1.059. One method is to simply ignore the fractional part of the address value, but this produces a signal that is barely intelligible. Another method, as seen in the top part of the figure below, is to choose only the nearest ROM address value. If the "virtual" ROM address is 4.589, the program will choose address 5.

The third method is to use linear interpolation. This method calculates the ROM value by taking a weighted average of the neighboring ROM values. For example, if the virtual address is 3.4, the program will take 0.6 of the value at address 3, and 0.4 of the value at address 4. See the bottom part of the figure for a pictorial representation of this.

If ROM[*x*] is the value of the ROM at address *x*, *F* is the fractional part of the virtual ROM address, and *I* is the integer part, the output of interpolation is:

$$ROM[I+F]=(1-I)ROM[I]+(F)ROM[I+1]$$

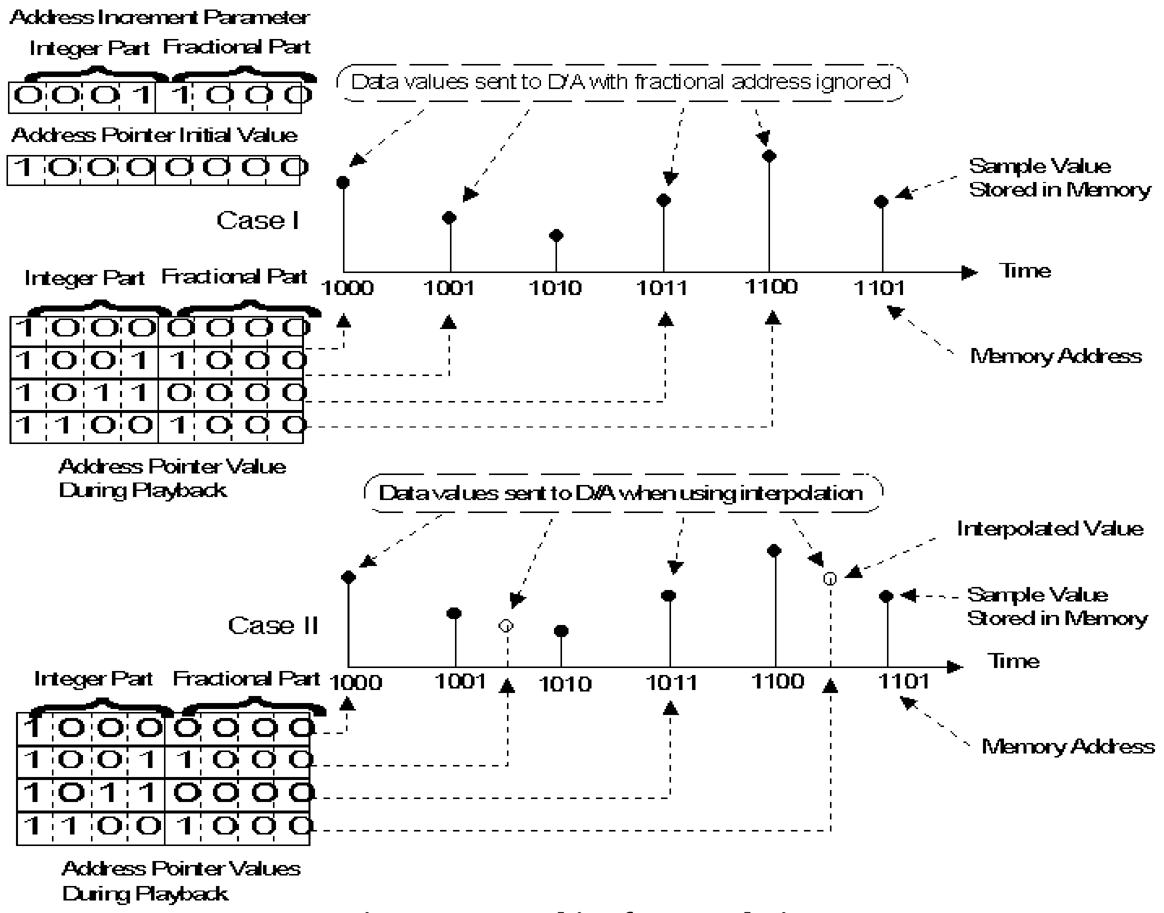

Address Increment Parameter
Integer Part
Fractional Part
0 0 0 1 1 0 0 0
Address Pointer Initial Value
1 0 0 0 0 0 0 0
Data values sent to D/A with fractional address ignored
Sample Value Stored in Memory
Case I
Time
1000
1001
1010
1011
1100
1101
Memory Address
1 0 0 0 0 0 0 0
1 0 0 1 1 0 0 0
1 0 1 1 0 0 0 0
1 1 0 0 1 0 0 0
Address Pointer Value During Playback
Data values sent to D/A when using interpolation
Interpolated Value
Sample Value Stored in Memory
Case II
Address Pointer Values During Playback

Figure 8 - Graphic of Interpolation

3.2 Instrument Module Implementations

3.21 Instrument

The instrument module is the core module of my part of the project. It takes a play signal and note value signal, and produces a digital signal result. The note value can be between 0 and 24, with 0 representing an "A," 24 representing an "A" two octaves higher, and the other numbers representing every half-note in between. The module produces an 8-bit result.

The instrument module is timed by an external source, giving it a start signal. The start signal tells the instrument to calculate the next value of the digital signal for the instrument at the desired pitch. When the instrument is simulated at the correct sampling rate, the stream of instrument results will produce the desired sound when run through the DAC. The figure below depicts an overall block diagram of the instrument module.

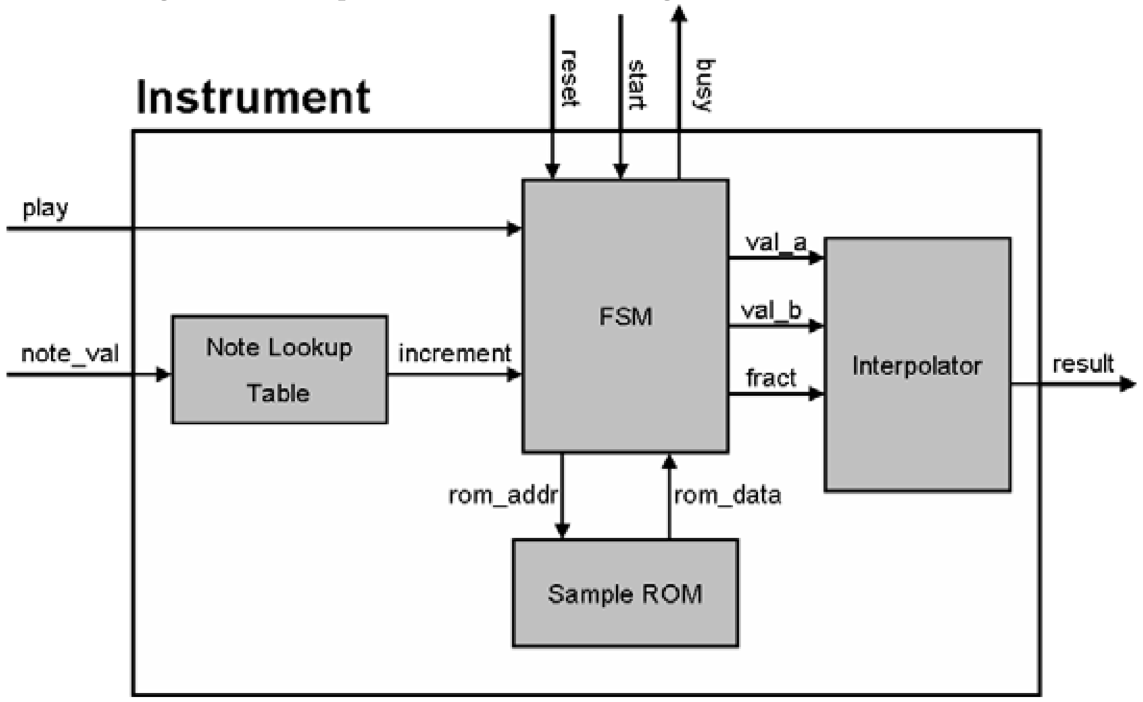


Figure 9 - Overall Block Diagram of Instrument Module

As each start is asserted, the instrument should produce a new result value based on the values in the sample ROM and the note value. Figure 10 shows the note value changing in simulation.

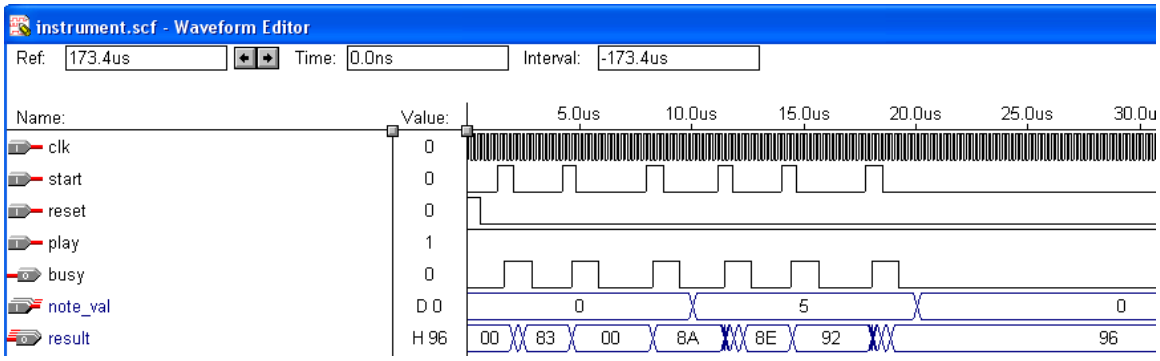


Figure 10 - Instrument Simulation Graphic

3.22 Note Lookup Table

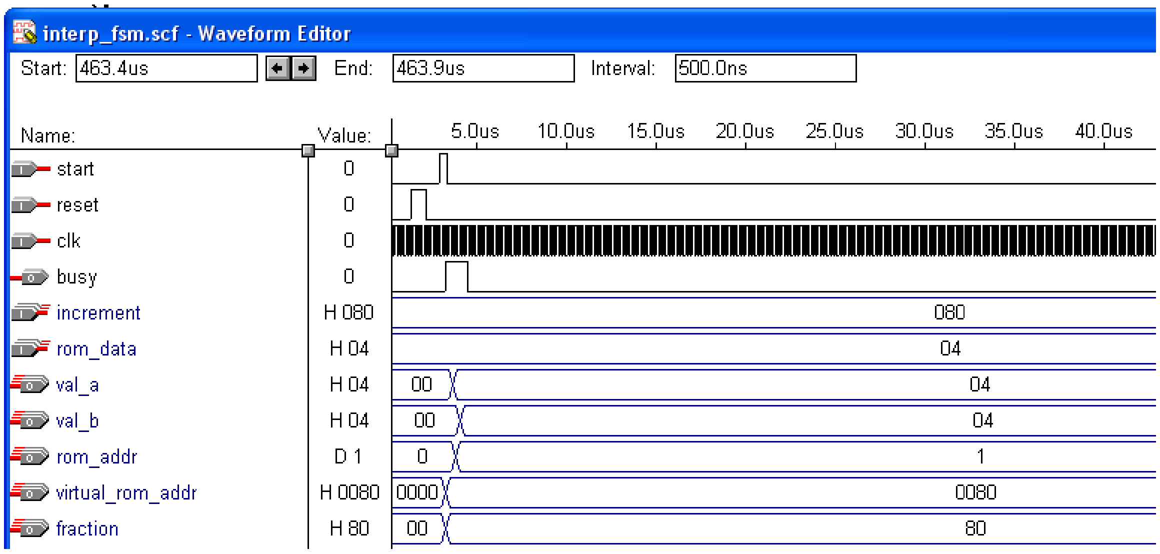
The note value is converted to an *increment parameter*, which represents the value by which the low "A" frequency must be multiplied to produce the desired pitch. If the note value is "1," or a A#, the increment parameter will be 1.059, or one plus the twelfth root of two. This number is represented as a two's complement value with an integer and a fractional part.

The left side of Figure 1 shows how the increment can be represented with an integer and a fractional part. In my case, the increment has two integer bits and eight fractional bits.

3.23 Interpolator FSM

The instrument's finite state machine (FSM) controls the reading from the sample ROM and feed values to the unit that actually interpolates. It keeps track of a virtual address pointer that represents the address of the sample in ROM that we would like to interpolate the value of (e.g. ROM[1.34]).

Every time the instrument acts, it increments the virtual address by the increment parameter. It then retrieves the values in the sample ROM at integer addresses immediately above and below the virtual address (ROM[1] and ROM[2] in this case). It passes these values to the interpolator unit, as well as the fractional part of the address (0.34 here). This number is used to calculated the weighted average of the two ROM addresses to produce the interpolated value.


interp_fsm.scf - Waveform Editor

Start: 463.4us End: 463.9us Interval: 500.0ns

Name:	Value:
start	0
reset	0
clk	0
busy	0
increment	H 080
rom_data	H 04
val_a	H 04
val_b	H 04
rom_addr	D 1
virtual_rom_addr	H 0080
fraction	H 80

Figure 11 - Interpolator FSM Simulation Graphic

In Figure 11, the interpolator FSM is simulated. It fetches two data values from the ROM, val_a and val_b. Here these are both the same value, 04. Val_a comes from ROM[0] and val_b comes from ROM[1]. This is consistent with the increment value, 080, which represents one-half. Because the first virtual address will be 0 + ½ = ½, the first two address should be the one above (1) and the one below (0) the virtual address. The fractional part of the virtual address (80) is an output that will be past on to the interpolation calculator.

3.24 Sample ROM

The Sample ROMs themselves (*.mif files) were made by Valerie. There are three ROMs, one each for the violin, piano, and flute. The ROMs store the values of a complex waveform for each instrument. The values are 8 bits wide, and the ROM has a length of 256. Each ROM stores two full periods of the instrument, and when played at the fundamental sampling rate with no interpolation, the ROMs all store a 440 Hz “A” note.

3.25 Interpolator Calculator

The interpolator calculator (or just “interpolator”) performs the actual linear interpolation. It takes the values from the lower and higher ROM address, as well as the fractional part of the virtual ROM address. Again, if ROM[*x*] is the value of the ROM at address *x*, *F* is the fractional part of the virtual ROM address, and *I* is the integer part, the output of interpolation is:

$$ROM[I+F]=(1-I)ROM[I]+(F)ROM[I+1]$$

The interpolator unit uses two simple multipliers to quickly compute this value.

In the simulation in Figure 12, the fraction is kept constant at one-half (H80). The interpolator unit is simulated at various low_val and high_val outputs, and it shows that the result is indeed the arithmetic mean of the two inputs.

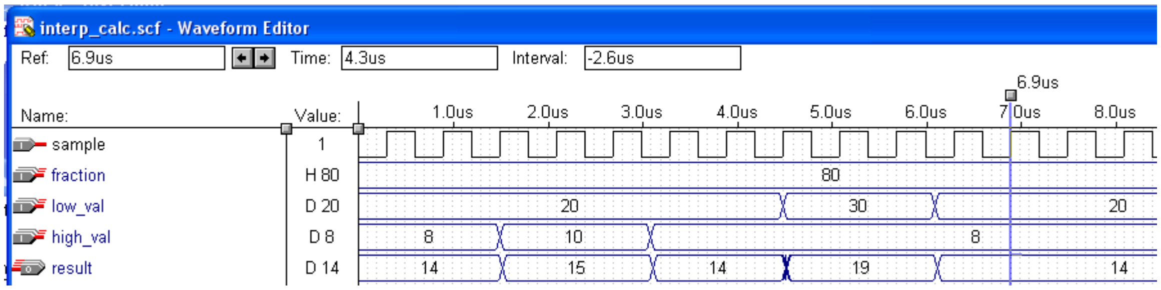


Figure 12 - Interpolator Simulation Graphic

3.26 Divider

The divider module is external to the instruments, but it controls the timing of the instrument system. As I mentioned before, each instrument's sample ROM contains two waves in 256 samples. This must be sampled at a rate that creates a 440 Hz "A." Because the system. It turns out that we need 558 samples per 1/100 second to create an "A" from the stored sample. Because we are using a 1.8432 MHz system clock, the divider unit divides the clock by 33 to create the correct sampling signal.

3.27 DAC FSM

The DAC FSM (digital-to-analog converter finite state machine) effectively synchronizes the instruments in the system. Each sampling period, it tells every instrument to fetch new interpolation values. When the signals from each instrument are all ready, the DAC FSM enables the result of the system to be output to the DAC. This unit deals with the start and busy signals from each instrument, and waits for each instrument to be ready with its values before changing the output to the DAC.

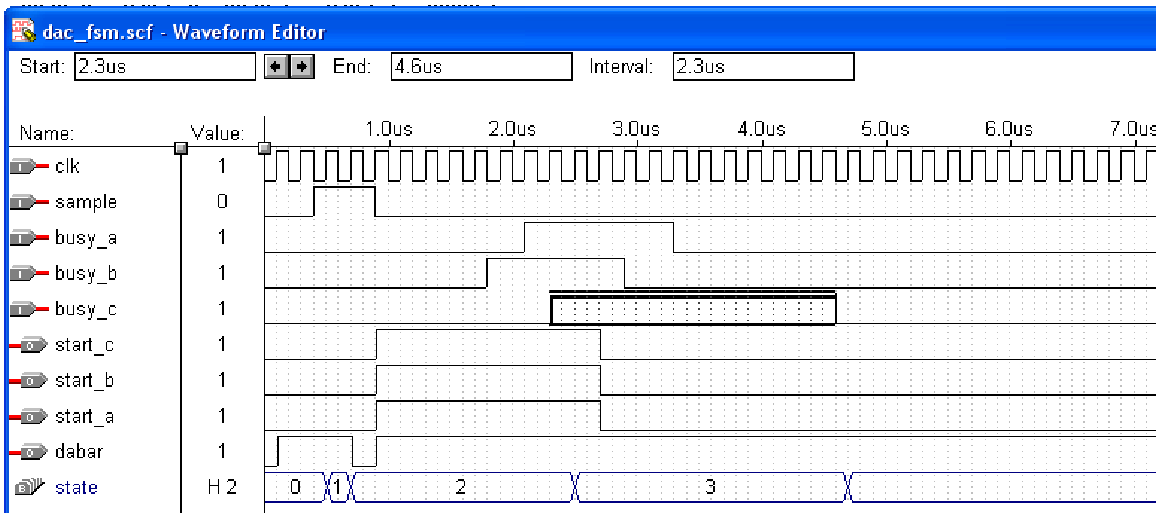


Figure 13 - DAC FSM Simulation Graphic

In the simulation in Figure 6 it is clear that the DAC FSM waits for all three instrument busy signals to go low before continuing to state 0, where it will output the result to the DAC (bring dabar low).

3.3 Other Implementations

In addition to instrument-related modules, I implemented a few other parts of the project. One was the top-level module for my FPGA. The other was a couple of song ROMs to drive part of Susan's control module.

3.31 Midi Synth

This top level module integrates the instruments, ADSRs (Valerie's back end modules), and Valerie's volume control unit. It includes a divider and DAC FSM to control the timing and synchronization of instruments.

Originally, we wanted to divide the project onto two FPGAs: one to have all of the control signals and user inputs, another to house the instruments and back end processing. Unfortunately, the many ROMs in our project did not allow all three instruments and ADSRs to be stored on one FPGA. Thus, this top level module included two instruments and ADSRs internally, and had one instrument and ADSR external.

The module receives control signals from Susan's controller FPGA and gives play and pitch signals to the two internal instruments. It also receives the digital signal output from the external instrument, which is stored on Susan's FPGA. This instrument's pitch and play signals are all controlled within Susan's FPGA, but the instrument interacts with the DAC FSM on my ROM through the start, busy, and sample signals.

All of the signals between the FPGAs are bussed through a 50-pin cable connecting the boards of each unit.

3.32 Songs

One of the features of the sound module is to play pre-stored songs from internal ROMs. Although it was Susan's part of the project to read through the ROMs and interpret the instructions, I programmed the songs into the *.mif format.

The two songs are *Fur Elise* and *Silent Night* ('tis the season!). I found sheet music for the songs online, at http://www.8notes.com.

For *Fur Elise* (see Figure 14), I programmed nine measures of the song into a ROM. The treble clef was programmed as the violin, and the bass clef was the piano. For *Silent Night* (see Figure 15), I programmed the entire twelve-measure song. The violin part was programmed as the violin, and the piano part was split into a two-part harmony between the piano and the flute. I had to transpose some of the notes differently then they were written because of the limited number of notes on the sound module. There were also too many notes on the sheet music to program using only three separate instruments.

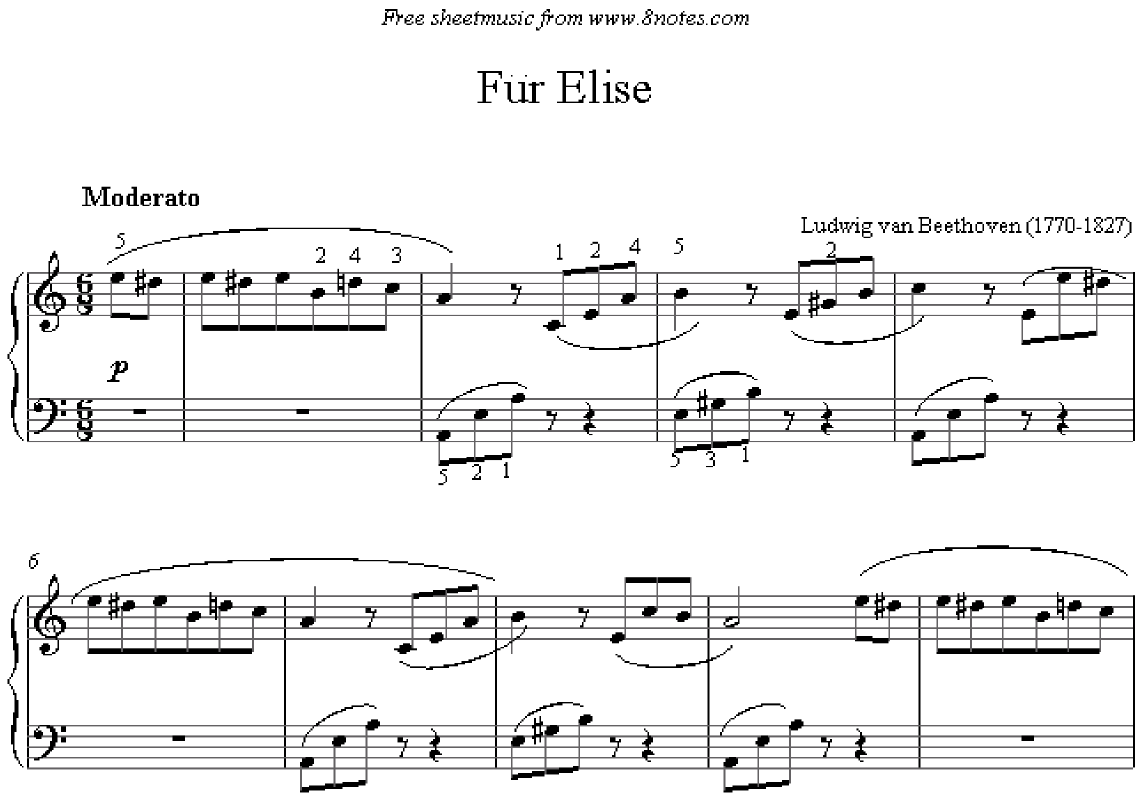

Free sheetmusic from www.8notes.com
Fur Elise
Moderato
Ludwig van Beethoven (1770-1827)
p

Figure 14 - Fur Elise Sheet Music

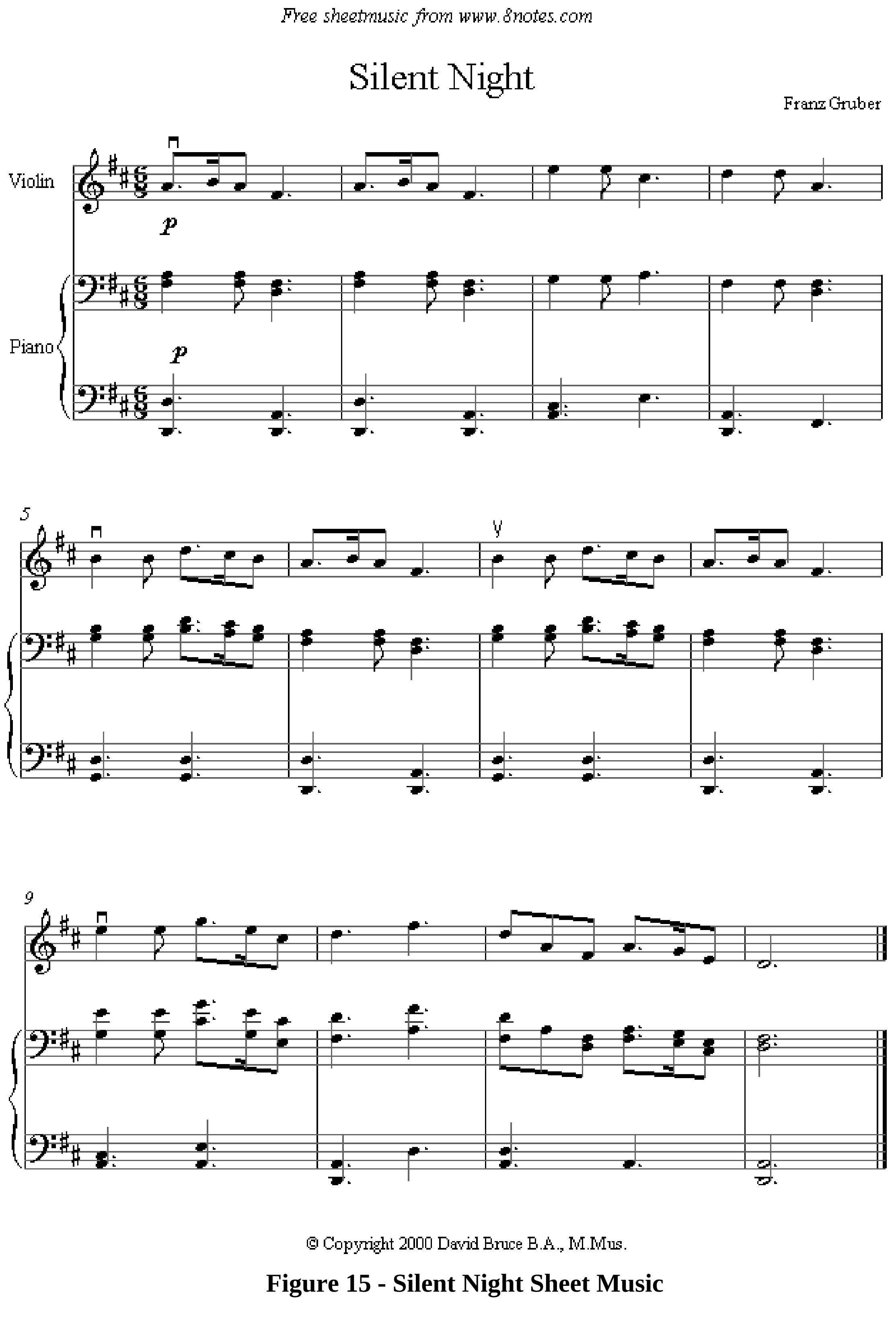

Free sheetmusic from www.8notes.com
Silent Night
Franz Gruber
Violin
Piano
p
p
5
9
© Copyright 2000 David Bruce B.A., M.Mus.

Figure 15 - Silent Night Sheet Music

In the interest of time, I first converted the sheet music into human-readable pseudo-code. The first measure of *Fur Elise*, for example, looked like:

```
---1
p1
1o19
p1
off1
1o18
p1
off1
```

The ---1 is simply a comment indicating the beginning of measure 1. This was useful for finding mistakes in the code. The first instruction “p1” is a pause for 1 period. “1o19” means turn instrument 1 (the violin) on and play pitch 19. Pitch 19 corresponds to a high E, as one can see from the sheet music. Then there is another pause, and “off1” tells the sound module to turn off the instrument. The entire piece continues like this until the end, and other instruments can be added with messages like “2o19” or “off2.”

The next step was to convert the songs to a *.mif file that could be read in Verilog to initialize a ROM with the proper songs. All of the song ROM files were 256 addresses in length (*Silent Night* barely fit). To convert this pseudo-code into actual instructions, I wrote a Python script to process the text files. See the Appendix for the code. After run through the script, the first measure of *Fur Elise* looks like:

```
0    :    1;
1    :    115;
2    :    1;
3    :    64;
4    :    114;
5    :    1;
6    :    64;
```

These instructions can be read by Susan’s control unit to drive the instruments and produce the songs. Because the keyboard does not allow multiple instruments to play at the same time, the song ROMs enable creation of polyphonic sounds.

3.4 Testing and Debugging

Our group was careful to design the project so that it could be implemented in modular, incremental stages. When each module or each incremental implementation was finished, we would test thoroughly.

I found that testing by simulation was often not enough to ensure successful functionality. At different design stages, I would program the FPGA with a module that would allow me to test my code in hardware. This reduced the number of surprises later when I assembled the project and programmed the FPGA.

I also designed as many modules as possible so they could be extended, much as Java has classes and inheritance. For example, instead of creating three separate piano, violin, and flute modules, I designed a single, generic "instrument" module. I tested this module thoroughly with a simple sine wave ROM before adding in the more complicated sample ROMs.

My technique for debugging the overall system and assembling the modules was to incrementally add functionality and test the system. I would gradually convert dummy modules into real functional ones and catch errors along the way. If a module did not work correctly, I would create a new top level module whose only purpose was to test this module. I actually had three or four top level modules that I could separately program to test individual features.

Once in hardware, I would use the LEDs liberally to indicate correct inputs and outputs. I would also be extra careful when wiring to not introduce time-consuming stupid mistakes.

Debugging our project was challenging because many of the aspects of the system could not be easily tested in software simulation. The sounds of the instruments consisted of hundreds of digital values being output from the DAC at a very fast rate. MAX+Plus II could not simulate the actual output waveform. Tweaking the parameters of the instruments and the ADSRs required often recompilation and reprogramming, which took a lot of time.

4. Shaping and Mixing of Sounds (by Valerie Gordeski)

4.1 Different Harmonics

There is an infinite complexity to the way different instrument produce sounds. Physics have long studied why a Stradivarius violin sounds like it does - and tried to reproduce the exact warm and richness of sound on the violin copies. Every note that is played by any instrument has a certain quality depending not only on its fundamental frequency (440 Hz of an A for example), but also on the combinations of the different harmonics that it has. A harmonic is a higher frequency wave that is a multiple integer amount times the fundamental frequency of the pitch (for example, 440*2 is 880, therefore 880 Hz is a harmonic of A). All high quality instruments have multitude of higher harmonics that do not only differ in frequency, but also in their phase from the fundamental. By combining the fundamental frequency and its harmonics, it is possible to obtain a sound that sounds somewhat like an instrument you are looking for.

To figure out the harmonic content for the three instruments that our team has chosen to replicate - the violin, the piano, and the flute - I have spent a lot of time on the internet trying to find papers and sites relating to this subject. I was able to find the relation between the fundamental frequency and the 5 harmonics for a piano, including the phase difference between them. However, for the flute and the violin I was only able to find the coefficients of their harmonic, therefore when I created the waveforms they did not have any phase information in them. These waveforms are stored in a ROM, with two periods and 256 coefficients (128 coefficients per cycle. Please refer to the memory initialization files given in the Appendix C for the exact values of the coefficients.

4.2 ADSR - Attack, Decay, Sustain, Release

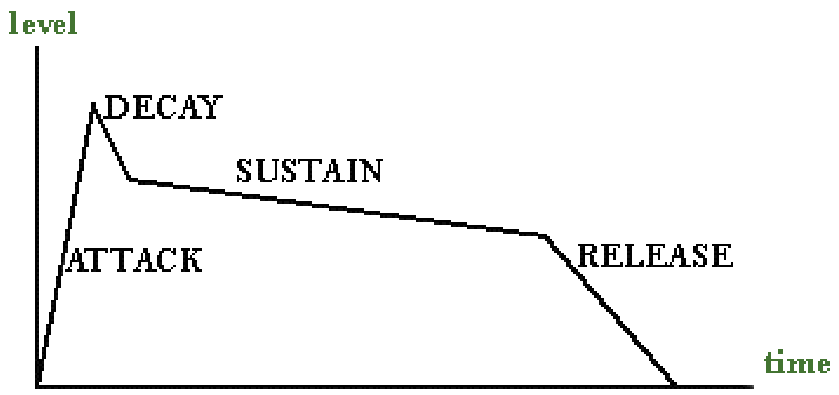


Fig. 16 The typical envelope shape of a piano (from http://www.teachnet.ie/amhiggins/lesson6.html#)

The most important characteristic and distinguishes the hammer-striking sound of a piano and a warm blowing of a flute is the shape of the note envelope. Aside from vibrating at a certain frequency, the amplitude of the note changes over time to give it the sound we are used to hearing. The manner in which every envelope is described is called ADSR - or Advance, Decay, Sustain, and Release. These terms describe very well the basic shape of a piano envelope (see Fig. 16)

The piano has a really sharp ‘attack’ stage, with the peak value corresponding to the impact of a little hammer in the piano striking the string. After the loud initial vibration, the string quickly decays (therefore, the decay stage) to a fairly level value, after which the sound stays somewhat level (hence the name ‘sustain’). During the release part, the sound quickly decays to zero. The flute, being a wind instrument without striking or plucking, has a more rounded envelope shape: it has a gradual increase until a certain peak value, and after that a decrease to zero, without a sharp attack/decay boundary. If the flute sound is sustained, the sound oscillates in volume and its envelope looks like a sinusoid. By researching different instruments, and figuring out how their envelopes differ, our group came up with three instruments that we can shape: the piano, the violin and the flute. These instruments have enough differences in their envelopes that even with the imperfect lab tools the listener will be able to distinguish between different instrument sounds.

4.21 ADSR Piano

The first module that I have created is the ADSRpiano module. This serves as a template for all the other instruments, although each instrument differs from the next because of the different envelopes. With the current encoding scheme, the ADSR is only given the information of when to start and to stop a signal, a clock, a reset button, an enable signal (that serves as a counter) and a waveform. It then modifies the waveform to produce a sound wave with amplitude emulating an actual piano sound (please refer to Fig.17).

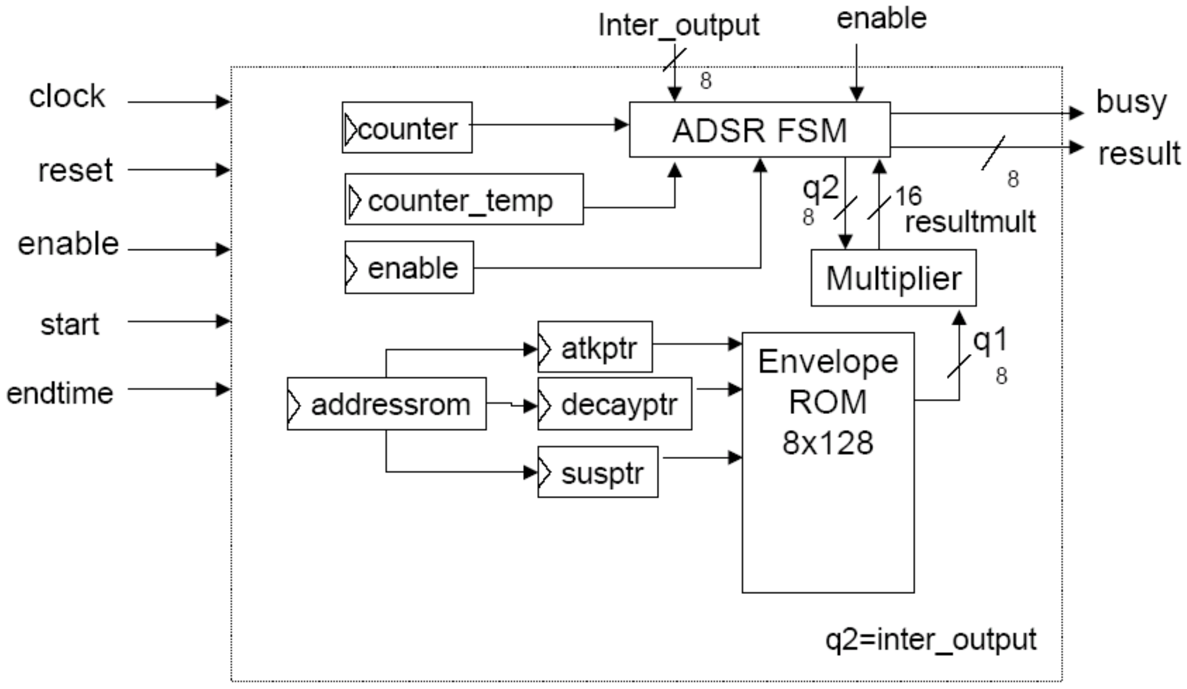


Fig. 17 The block diagram for a Piano ADSR

The way this modification is achieved is through a larger FSM, that controls a multiplier and a ROM with all the ADSR coefficients. The coefficients were first created in Matlab, and then converted into a .mif file. There are three pointers that keep track of the ROM address - atkptr (attack pointer), decayptr (decay pointer) and the susptr (sustain pointer). Originally, the ADSR was configured for a dynamic envelope shaping - given the shape of the note, it would have calculated the duration of the attack, decay, and sustain periods, and if the note was shorter than expected, it would have jumped to the respective pointers. However, our simple encoding scheme only allowed to receive a start and an endtime. Therefore, the piano stayed in the sustained region by moving the susptr backwards until the 'endtime' signal was given.

The FSM that controls the ADSR has four states: attack, decay, sustain, and idle (please see Figure 18 for the state transition diagram). With a reset signal, the FSM enters the idle state. The counter is set to zero, and the program is waiting for a start signal. With the start signal, the FSM moves into an ‘attack’ phase, by setting the rom address to be the atkptr. After that, the counter_temp counts every 109 enables and increments the pointer when it is done, while counter counts to 2618 to know when to reset the count to zero and to move to the next state. In all states, if ‘endtime’ signal is pressed, the FSM enters the idle state, and returns addressrom (pointer that keeps track of the ROM address) to 0 and all the pointers to their appropriate values.

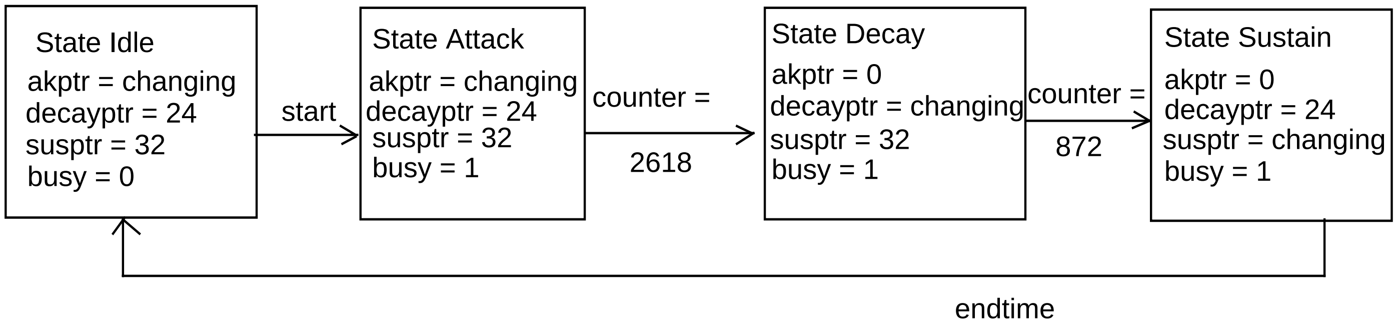


Figure 18. The state transition diagram for the piano module.

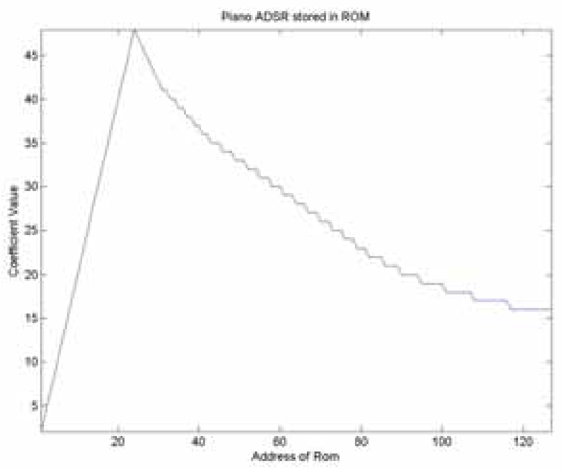


Figure 19. ADSR coefficients stored in the ROM

The shape of the envelope that is stored in the ROM is illustrated in Figure 19. As stated previously, the release part of the waveform is missing because the instrument never knows when to stop until it actually receives the stop signal. Therefore, it just keeps looping in the sustain region in order to keep the sound going.

4.22 ADSR Flute

Unlike the piano, the flute has a very slow attack, and a very gradual decay. It doesn’t really have a sustain mode - when the flute is sustained, you hear a sinusoid increase and decrease in aplitude, creating a wavering sound. Figure 20 illustrates the shape of the flute envelope.

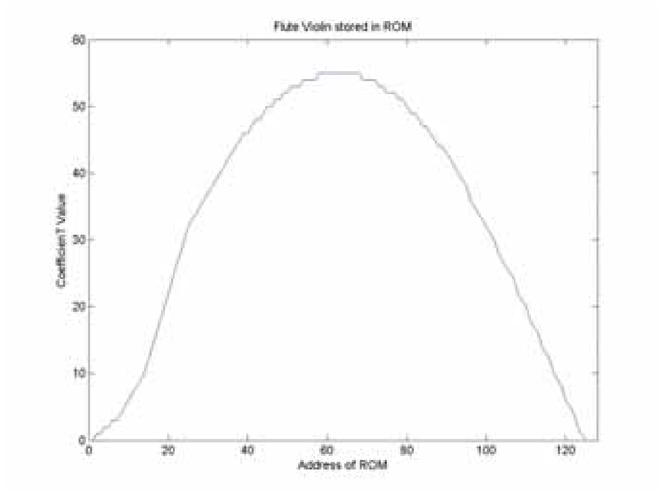


Figure 20. ADSR coefficients for the flute

The block diagram and the state transition diagram of the flute is very similar to the block diagram of the ADSRpiano module. It takes in the same inputs: the clock, the reset, the start, the end, the product of the interpolator, and produces the busy signal (for debugging) and an 8-bit result. Inside you will find a similar envelope ROM that stores the coefficients, and the multiplier. The multiplier takes in the coefficients from the ROM via the addressptr, and multiplies them by the waveform supplied by the interpolator.

Since there is no sustain, the ADSR flute has only three states: an attack state, a decay state, and an idle state. At any time it can leave those states if an end signal has been received - it then returns all the points to their default values and waits for another start signal. If, however, the end signal is not reached even after all the envelope ROM coefficients have been used, the decayptr will keep looping around in the high symmetric part of the coefficients (approximately from 20 to 109), to create a sinusoid-like wavering sound one might hear when an actual flute is sustained.

4.23 ADSR violin

The violin sounds different from both the flute and the piano. The key difference between these instrument is the way they sustain their notes : in the piano, after the hammer strikes the string the sound vibrations die away, therefore there is a gradual decay in the envelope. With the flute, there is the sinusoid fluttering that was described in the above section. With the violin, for as long as the violinist holds the bow to the string, the violin will play without any change in volume (unless of course the violinist wants it to change the volume). Therefore, after the sharp attack phase, in the envelope of the violin there is a long sustain period. I thought that the best way to represent this is just to keep the coefficients in the ADSR ROM constant.

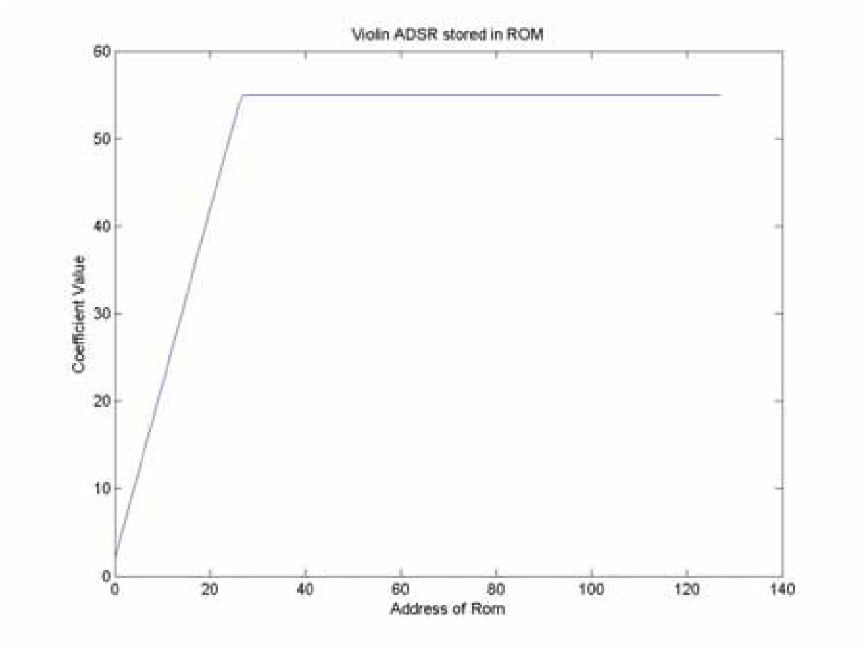


Figure 21. The ADSR envelope shape of the violin

The block diagram and the state transition diagram is again very similar to the piano (please refer to Figure 17). The inputs are the same, and the ROM is the same length. Perhaps a way to optimizer the storage would be just to have one address with the constant coefficient through which I would loop, but I kept all the ROMs having the same format. There are three states in this FSM: the attack, sustain, and idle. The default state is the idle state: this is where the module waits for the input. As soon as it receives the start signal, it goes to the attack phase, and begins its multiplication. As before, counter_temp and counter registers keep track of when to increment the address pointer and when to change state, respectively. These registers are incremented at every ‘enable’ signal.

4.3 The Mixer and the Sound Control

After the outputs come out of the ADSR, it is necessary to add them appropriately. Originally, the waveform that went into ADSR was all positive, and so the multiplication done in the ADSR was unsigned. With addition of thee instruments at the same time, a different strategy has to be used.

4.31 The Mixer

Since all the coefficients for the actual instrument waveforms with all of their harmonics range from 0-255 (all positive), they have to be converted to the range from -128 to 128; and be put into the 2’s complement format. Then, the type of multiplication in the ADSR modules was changed to signed multiplication. Now, when the digital waveform comes out from the ADSR it is centered appropriately around 0. The waveforms can then be

added through a simple addition; or they can be passed through the sound controller first and then be added.
The code for this very simple mixer resides in the top level module midi_synth.v, found with Chris Sheehan's code.

4.31 The Volume Control

The volume control module takes in clock, reset, volup, voldown, and an input from the ADSR. It produces a 12 bit output, which is a result of the appropriate scaling of the sound.

The block diagram for the sound control can be found below. In order to multiply the sound by a certain fraction, you need to first mutiply it and then to divide it. This module doesn't have an FSM, it uses a mix of blocking and nonblocking assignments to produce the appropriate output. The counter in this module keeps track of whether we are increasing or decreasing in volume. The default value for the counter is 2 - this means that the sound is in the 'normal' state. There are two volume down settings - when the user presses the voldown button once, the sound goes down, and then down more. After that, the number of presses doesn't change, and the counter remains at 0. Now if the user presses the volume up button, the counter will increment, and the sound will increase accordingly. This simple scheme for the sound control has been replicated for all thee instruments.

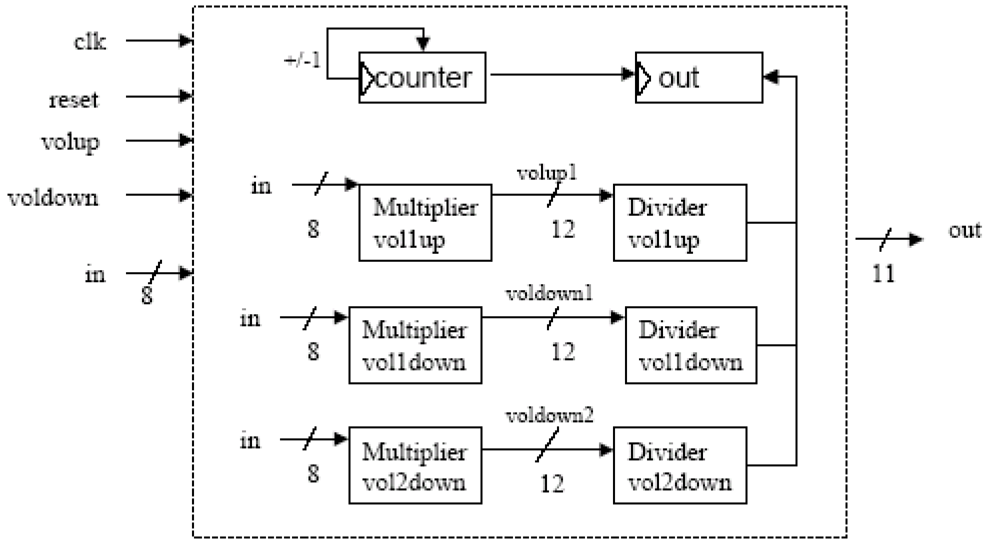


Figure 22. The Volume Control Module.

When this module was integrated with the ADSRs, there was also a volume switch implemented. With the volume switch on, the FPGA sends the outputs through the volume control first, and then reads them out and mixes them. With the volume switch off, it takes the outputs directly from the ADSR, and then mixes them.

4.4 Testing and Debugging

Majority of my testing and debugging happened through simulations. In order to see if the ADSR was working properly and whether the states were changing the way I wanted them to, I changed the counter and counter_temp values to something much smaller, where I could see over several clock cycles that the address pointer was incrementing properly. A sample time diagram from the violinADSR is shown below in Figure 23. In this diagram, you can see the address pointers incrementing when they are supposed to, and the output incrementing at first during the attack phase (note that the attack pointer is changing value at this time) and the sustain pointer changing during the sustain state. You can also see the result incrementing in during the attack phase, and then staying constant through the sustain part of the violin. When I knew that all of my modules behaved in a logical fashion, we have tried to integrate them all. Much to my surprise, all of the ADSRs worked as expected on the first try.

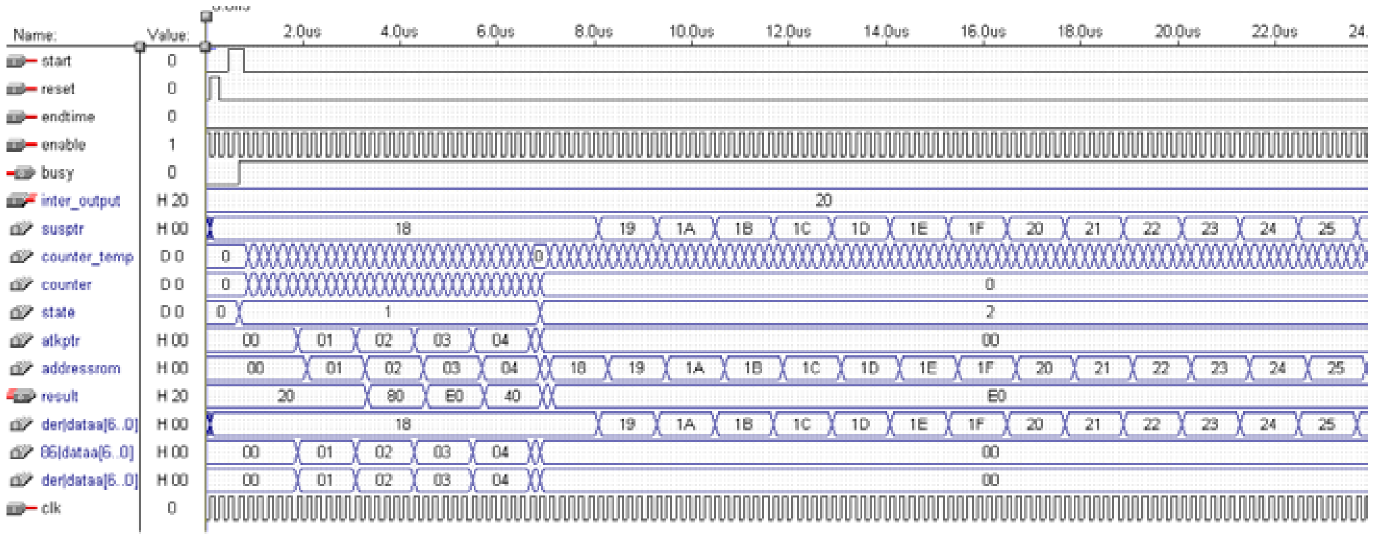


Figure 23. The timing diagram for violinADSR module

The second thing I had to debug was the volume control module. To do that, I first tested the sound module in simulation (see below). As you can see, the output changes appropriately with each volume up and volume down button press. On reset, the state counter switches to 2, which is the default value. With volumeup and volumedown presses, the output becomes scaled accordingly.

The biggest issue I had with the volume control was during integration. At first, I could not hear any changes in volume. Because of the limitation of the 8-bit DAC, and the truncation of the output that had to happen, the range has to be chosen carefully to hear the changes. Also, with the instrument such as the piano, only the attack part can be heard since after it decays past the certain value, the bits where the sound is happening are outside of our 8-bit DAC selection range (the sound control has a 12 bit output, and typically we select the highest 8 bits). This issue could have been resolved by using a 12-

bit DAC, but since we have already integrated the entire system, every control would have to have been changed, and at that point it was too late to do anything about it.
I have also created my own top level module called adsroutputs.v. in order to see if the sound module and the ADSRs can be put together without any compile errors. After that, I put the sound control onto Chris's FPGA and tested it there, and the final code of the adsroutputs became integrated into Chris's midi_synth.v top level module.

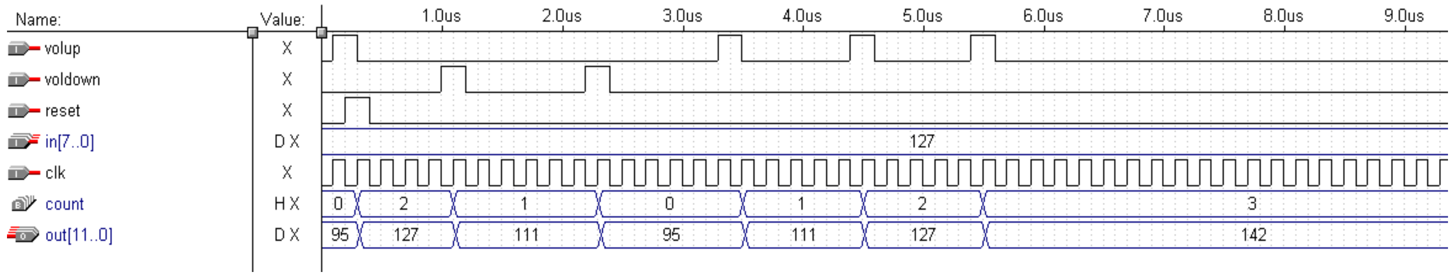


Figure 24. The timing simulation for the sound module.

5. Conclusion

Our group had many lessons to take away from this final project.

First modularity is the key to success in a large project. When there are ten or twenty or more separate modules, the system is too complex to effectively test and debug once all assembled. By splitting the project into three main mini-projects (one for each person), and designing ways for each of the parts to be individually tested, we greatly simplified the design and implementation process. Each group member was then able to further split up his or her part of the project into modules that were small enough to manage and test by themselves.

We also learned that in a project with multiple team members, defining and keeping track of specification is critical. Although each person was implementing the part on their own, it was important that each person understood the inputs and outputs the others expected. This insured that the modules would work well together once assembled. For example, Val was originally going to design her ADSR so it would take in a "begin note" and "duration" signal. But Susan and I determined that note duration signals would be difficult to implement in our song encoding, so Val changed her specification so that only a "start note" and "end note" signal were required.

It was also important but difficult for us to keep track of code versions and file hierarchies. The Verilog code did not support the equivalence of inheritance in object-oriented programming, but there still existed the concept of modules "having" other modules. Our file structure could not really reflect this because everything was in the same folder. Also, compilation produced many extra files that cluttered the file space. In hindsight, we probably should have used a versioning system such as CVS.

In all, we learned to work together as a team. Our victories and failures were ours together. We learned to compromise on issues we disagreed on, and we helped each other through good and bad times during the life of the project.

Given enough time, and enough effort, and enough FPGAs, *anything* is possible.

Appendix A : Susan Hwang's Implementation

```
//FRONTEND CONTROL:  adds all the modules together and is the main
//interface between the keyboard, clk, buttons and switches. this
//module also acts as a controller that regulates the signals going
//into the modules

module frontendcontrol(clk,start,start1,reset,songorkey,rec,songs,
instrument,kclk,kdata,play1,play2,pitch1,pitch2,outdata,blank,busy);

input clk,reset,songorkey,rec,start,kclk,kdata,start1,blank;
input [1:0] songs;
input [1:0] instrument;

output play1,play2,busy;
output [4:0] pitch1,pitch2;
output [7:0] outdata;

wire play3;
wire [4:0] pitch3;
wire [1:0] songs;
wire [7:0] outdata;
wire [7:0] outmessage;
wire play1r,play2r,play3r,play1m,play2m,play3m;
wire [4:0] pitch1r,pitch2r,pitch3r,pitch1m,pitch2m,pitch3m;
wire [7:0] qrom1,qrom2,qrom3,qram,addressROM,addressRAM,addressRAMm,
addressRAMr,dataram, recdataram, ram_addr, ram_data;
wire weram, ready, ready2, LED, weramr, weramm, we, busy;

songrom1 SongROM1(addressROM,qrom1);
songrom2 SongROM2(addressROM,qrom2);
songrom3 SongROM3(addressROM,qrom3);
recordram recordram(addressRAM,weram,dataram,qram);

//blanks the ROM with blank press
blanktherom blanktherom(clk, blank, busy, ram_addr, ram_data, we);

keyinterface keyinterface(reset,start,kclk,kdata,clk,outdata,ready);
keydecoder keydecoder(clk,instrument,outdata,ready,outmessage,ready2);

realplay realplay(clk,outmessage,play1r,play2r,play3r,pitch1r,pitch2r,
pitch3r);

record record(clk,reset,outmessage,ready2,LED,addressRAMr,weramr,
recdataram,count);

masterunit masterunit(clk,reset,start1,songs,qrom1,qrom2,qrom3,qram,
play1m,play2m,play3m,pitch1m,pitch2m,pitch3m,addressROM,addressRAMm,
weramm);

//if you are blanking, use blank's we else if you are using keyboard
//and recording, let record access we
assign weram = busy? we: (!songorkey&&rec)?weramr:weramm;


//if you are blanking, use blank's ram_addr,else if you are
//added mux to choose between which data to use
assign addressRAM =
busy? ram_addr : (!songorkey&&rec)?addressRAMr:addressRAMm;

//using keyboard and recording, let record access address
assign dataram = busy? ram_data : recdataram;

//these choose which play and pitch to use depending on if we are
//in realplay or song mode
assign play1 = songorkey?play1m:play1r;
assign play2 = songorkey?play2m:play2r;
assign play3 = songorkey?play3m:play3r;
assign pitch1 = songorkey?pitch1m:pitch1r;
assign pitch2 = songorkey?pitch2m:pitch2r;
assign pitch3 = songorkey?pitch3m:pitch3r;

endmodule
```

```
//This keydecoder takes the output of the 8 bit key code of
//the keyboard and uses that to find the corresponding musicalnote

module keydecoder(clk,instrument,inkey,ready,outmessage,ready2);
input clk,ready;
input [1:0] instrument;
input [7:0] inkey;
output [7:0] outmessage;
output ready2;

reg ready2;
reg [4:0] current;
reg [4:0] previous;
reg [7:0] outmessage,r1,r2;

always @ (posedge clk) begin

r1 <= outmessage; //to level2pulse the outmessage
r2 <= r1;

//if it is a break, off note then play off message
//when there is a new outmessage, ready2 goes high
if (current == 5'd31) begin
            outmessage <= {instrument,1'b0,5'b00000};
            ready2 <= (r1 != r2) ? 1 : 0;
      end

//if it is NOT a repeat, then play on message
//when there is a new outmessage, ready2 goes high
else if (previous != 5'd31 && inkey != 8'd0 && current != 5'd30) begin
      outmessage <= {instrument,1'b1,current};
      ready2 <= (r1 != r2) ? 1 : 0;
      end

//these are the corresponding pitch codes to the keyboard codes

if(ready) begin
case(inkey)
8'h15:      current<= 5'd0; //LOW A
8'h1E:      current<= 5'd1; //A#
8'h1D:      current<= 5'd2; //B
8'h24:      current<= 5'd3; //C
8'h25:      current<= 5'd4; //C#
8'h2D:      current<= 5'd5; //D
8'h2E:      current<= 5'd6; //D#
8'h2C:      current<= 5'd7; //E
8'h35:      current<= 5'd8; //F
8'h3D:      current<= 5'd9; //F#
8'h3C:      current<= 5'd10;//G
8'h3E:      current<= 5'd11;//G#
8'h1A:      current<= 5'd12; //MIDDLE A
8'h1B:      current<= 5'd13; //A#
8'h22:      current<= 5'd14; //B
8'h21:      current<= 5'd15; //C
8'h2B:      current<= 5'd16; //C#
8'h2A:      current<= 5'd17; //D
8'h34:      current<= 5'd19; //D# //i switched 34, with 32
```

```
8'h32:      current<= 5'd18; //E
8'h31:      current<= 5'd20; //F
8'h3B:      current<= 5'd21; //F#
8'h3A:      current<= 5'd22; //G
8'h42:      current<= 5'd23; //G#
8'hF0:      current<= 5'd31; //BREAK
default: current<= 5'd30;
endcase
previous <= current;
end
end

endmodule
```

```
//The KeyInterface module handles the interface between the PS/2
//Keyboard and the FPGA. It takes in the the clk and data from the
//keyboard and samples the data at low of kclk

module keyinterface(reset,start,kclk,kdata,clk,outdata,ready);

input clk,kclk,kdata,reset,start;
output [7:0] outdata;
output ready;

//must level-to-pulse the kclk, the samples determine
//when the interface samples the kdata from keyboard
lev2pulse Level2Pulse(clk,kclk,sample);


parameter S_INITIAL = 0;
parameter S_1 = 1;

reg [7:0] message,outdata;
reg state,started,ready;
reg [3:0] counter;

always @ (posedge clk) begin
if (reset) begin              //at reset, state at initial,
      message <= 0;           //nothing is started and keypress is 0;
      state <= S_INITIAL;
      started <= 0;
      end
if (start)                    //once the start button is pressed,
      started <= 1;           //nothing is started and keypress is 0;
else begin
      case(state)       //if kdata (start bit) is low and module
      S_INITIAL:  begin //started, counter begins and counting and
                        //nothing is started and keypress is 0;
                        if (~kdata && sample && started) begin
                              state <= S_1;
                              message <= 0;
                              end
                              ready <= 0;
                        end
      S_1:  begin
                  if (counter == 4'b1010 && kdata==1)
                        begin       //when counter reaches 10,

                        state <= S_INITIAL;//message has been parsed
                        outdata <= message;
                        ready <= 1;
                        counter <= 0;
                        end
                  else if (sample) begin  //each bit is assigned
                        case(counter)
                        0: message[0] <= kdata;
                        1: message[1] <= kdata;
                        2: message[2] <= kdata;
                        3: message[3] <= kdata;
                        4: message[4] <= kdata;
                        5: message[5] <= kdata;
                        6: message[6] <= kdata;
                        7: message[7] <= kdata;
                        endcase
                        counter <= counter + 1;
                        end
                  end
      endcase
      end
end
endmodule
```

```
//LEV2PULSE: is used by the keyinterface to level to pulse the
//negative clock edge of the kclk

module lev2pulse(clk,in,out);

      input clk;
      input in;
      output out;

      reg r1,r2,r3;
      always @ (posedge clk)
      begin
            r1 <= in;
            r2 <= r1;
            r3 <= r2;
      end

      // level to pulse converter: taking high to low rise
      assign out = r3 & ~r2;
endmodule
```

```
//MASTERUNIT: Interfaces with the 4 memory units to take the messages
//stored in the memory units and translating the messages to control
//the 3 instruments.  It has an external control that determines which
song it will play.

module masterunit(clk,reset,start,songs,ROM1data,ROM2data,
ROM3data,RAMdata,play1,play2,play3,pitch1,pitch2,pitch3,addressROM,
addressRAM,we);

input reset,start,clk;
input [1:0] songs;      //songs determines roms to read from
input [7:0] ROM1data,ROM2data,ROM3data,RAMdata;

output play1,play2,play3,we;
output [4:0] pitch1, pitch2, pitch3;
output [7:0] addressROM,addressRAM;

parameter S_INITIAL = 0;
parameter S_1 = 1;
parameter S_2 = 2;
parameter S_3 = 3;
parameter S_4 = 4;

reg startedcount,startedROM,play1,play2,play3;
reg [2:0] state;
reg [5:0] count;
reg [4:0] pitch1,pitch2,pitch3;
reg [7:0] address,data;

//wire change;
counter THEcounter(clk,count,startedcount,change);

always @ (posedge clk) begin

case(songs)                    //songs selects which ROMdata to use
2'b00:      data <= RAMdata;  //this is from recordram
2'b01:      data <= ROM1data; //this plays "fur elise"
2'b10:      data <= ROM2data; //this play "silent night"
2'b11:      data <= ROM3data; //this is empty
endcase

if(reset) begin                      //everything is initialized
      state <= S_INITIAL;
      startedcount <= 0;
      play1 <= 0;
      play2 <= 0;
      play3 <= 0;
      pitch1 <= 0;
      pitch2 <= 0;
      pitch3 <= 0;
      startedROM <= 0;
      end
else begin
      case(state)
      S_INITIAL: begin
                  if (start)  //waits for start to start playing
                        state <= S_1;
                  else
                        address <= 0;
                  end
      S_1:  begin             //This reads from the memory unit
            startedROM <= 1;
            state <= S_2;
            if (startedROM)
            address <= address + 1;
            end
      S_2:  state <= S_3;           //buffer state
      S_3:  begin
            case(data[7:6])         //parses the message
            2'b00: begin            //This case is the special "pass"
                  startedcount <= 1; //start counting
                  count <= data[5:0];
                  state <= S_4;     //go to S_4 to wait for count
                  end
            2'b01: begin            //This case is the instrument 1
                  play1 <= data[5];
                  pitch1 <= data[4:0];
                  state<= S_1;      //go to S_1 to access next address
                  end
            2'b10: begin            //This case is the instrument 2
                  play2 <= data[5];
                  pitch2 <= data[4:0];
                  state<= S_1;
                  end
            2'b11: begin            //This case is the instrument 3
                  play3 <= data[5];
                  pitch3 <= data[4:0];
                  state<= S_1;
                  end
            endcase
      end
      S_4:  if (change)             //if change, itis time to access
                  begin             //the next address, S_1
                  state <= S_1;
                  startedcount <= 0;
                  end
      endcase
end
end


assign addressROM = address;
assign addressRAM = address;
assign we = 0;    // we is always 0 because it is always reading

endmodule
```

```
//This module msdivider divides the clock into
//eighth-notes.  This is how we determine
//how fast to play the stored songs and
//how fast to record the key-input songs
//1-bit equals (1/8 of a second)

module msdivider(clk,enable);
      input clk;
      output enable;

      reg [19:0] counter;
      reg temp_enable;
      always @ (posedge clk)
      begin

if (counter == 20'd460800)
                  counter <= 0;
            else
                  counter <= counter+1;
    if (counter == 0)
                  temp_enable <= 1;
            else
                  temp_enable <= 0;
      end

      assign enable = temp_enable;

endmodule
```

```
//REALPLAY: takes in a message from the keydecoder and translates it
//into control signals for ONE instrument while the other instruments
//are silent.

module realplay(clk,message,play1,play2,play3,pitch1,pitch2,pitch3);

input clk;
input [7:0] message;

output play1,play2,play3;
output [4:0] pitch1,pitch2,pitch3;

reg play1,play2,play3;
reg [4:0] pitch1,pitch2,pitch3;

always @ (posedge clk) begin
case(message[7:6])            //selects which instrument is playing
2'b01: begin
      play1 <= message[5];
      pitch1 <= message[4:0];
      end
2'b10:      begin
      play2 <= message[5];
      pitch2 <= message[4:0];
      end
2'b11: begin
      play3 <= message[5];
      pitch3 <= message[4:0];
      end
default : ;
endcase
end
endmodule
```

```
//The Record Module in enabled from external control button called rec
//and stores entries from keyboard input while counting the time that
//passes between each subsequent message and adds in pass messages.

module record(clk,reset,musicdata,ready2,LED,address,we,data,count,
startcounter,stopcounter);

input clk,ready2,reset;
input [7:0] musicdata;

output [7:0] address,data;
output LED,we,startcounter,stopcounter;
output [5:0] count;

parameter S_INITIAL = 0;
parameter S_1 = 1;
parameter S_2 = 2;
parameter S_3 = 3;
parameter S_4 = 4;
parameter S_5 = 5;
parameter S_6 = 6;

reg we,LED,started;
reg [3:0] state;
reg startcounter,stopcounter;
reg [7:0] data,address;

//recordcounter takes care of counting the length of keypresses
wire [5:0] count;
recordcounter recordcounter(clk,startcounter,stopcounter,count);

always @ (posedge clk) begin

if (reset)
      state <= S_INITIAL;
else
      case(state)
      S_INITIAL: begin        //initial state, initializes variables
                        address <= 0;
                        we <= 0;
                        data <= 0;
                        started <= 0;
                        if (ready2) begin       // new message
                              state <= S_1;
                              end
                        end
      S_1:  begin       //get the data and address prepared for write
                        started <= 1;
                        we <= 0;
                        data <= musicdata;
                        if(started)
                        address <= address + 1;
                        state <= S_2;
                  end
      S_2:  begin       //the message gets written to the recordram
                  we <= 1;    //while the counter begins to count
                  state <= S_3;
                  end
      S_3:  begin                  // buffer state, stopping we while
                        we <= 0;
                        if(ready2)  begin       //new message
                              state <= S_4;     //to be stored
                              end
                  end
      S_4:  begin       //the counter is stopped and stores time
                        we <= 0;
                        data <= {2'b0,count};
                        address <= address + 1;
                        state <= S_5;
                  end
      S_5:  begin             //the time message is now stored
                  we <= 1;
                  state <= S_6;
                  end
      S_6:  begin                  //this is the buffer state
                        we <= 0;
                        state <= S_1;
                  end

      endcase

startcounter <= (state==S_2)?1:0;   //the start of the time recording
stopcounter <= (state==S_4)?1:0;    //the stop of the time recording
LED <= (address == 8'hFF)? 1:0;     //indicates the end of the song
end
endmodule
```

```
//RECORDCOUNTER: Is in charge of recording the length of the
//keypresses from the keyboard which is stored in the variable count.
//This length message in turn is stored into the recordram.
//The record module determines when the recordcounter starts and
//stops counting.

module recordcounter(clk,startcounter,stopcounter,count);

input clk,startcounter,stopcounter;
output [5:0] count;

msdivider THEdivider(clk,enable);

reg [5:0] count;
reg started;

always @ (posedge clk) begin
if (startcounter) begin
      started <= 1;
      count <= 0;
      end
else if (stopcounter)
      started <= 0;
else if (started&&enable)
      count <= count + 1;
end
endmodule
```

```
//BLANKTHEROM:  blanks the RECORD RAM
module blanktherom(clk, start, busy, rom_addr, rom_data, we);

      // Blanks a RAM (misnamed) by writing zeros to all of its
      // addresses

      input clk, start;
      output busy, we;
      output [7:0] rom_addr, rom_data;

      // always write a zero
      assign rom_data = 0;

      reg [2:0] state;
      reg busy;
      reg [7:0] rom_addr;
      reg we;

      // human-controlled start button must be synchronized
      wire start_sync;
      button butt(clk, start, start_sync);

      always @ (posedge clk) begin
            if (start_sync) begin
                  busy <= 1;
                  rom_addr <= 0;
                  state <= 0;
            end
            if (busy) begin
                  case (state)
                  0:    begin       // wait state
                        state <= 1;
                        end
                  1:    begin       // write a zero
                        we <= 1;
                        state <= 2;
                        end
                  2:    begin       // stop writing
                        we <= 0;
                        state <= 3;
                        end
                  3:    begin       // increment the rom address
                        rom_addr <= rom_addr + 1;
                        state <= 0;
                        end
                  endcase
                  if (rom_addr == 8'hFF) begin        // reset the
address
                        rom_addr <= 0;
                        busy<= 0;
                  end
            end
      end
endmodule
```

Appendix B : Chris Sheehan's Implementation with .v, .mif, and .py files

Midi_synth.v

```
module midi_synth(clk, reset, result, dabar, pitchA, pitchB, play1,
play2,
      start_c, busy_c, sample, violin_sound, volup, voldown,
vol_control);



      // Top level module that takes control signals (play, pitch) for
three instruments,
      // as well as volume information, and outputs the combined signal
of the three finished
      // instruments for a DAC.
      //
      // Because of space limitations, two of the instruments (piano and
flute) are loaded into
      // this top module, and the other instrument is loaded onto the
control FPGA.

      input volup, voldown, vol_control;


      // The signals controlling the external instrument
      input busy_c;
      output start_c, sample;
      input [7:0] violin_sound;

// Control signals for internal instruments

      input [4:0] pitchA, pitchB;
      wire [4:0] pitchA, pitchB;
      input play1, play2;
      wire play1, play2;

      parameter ROM_data_length = 8;      // how many data bits
      parameter ROM_addr_length = 8;      // how many address bits

      input clk, reset;


      // Final result, with dabar to time the DAC
      output [ROM_data_length:0] result;
      output dabar;


      wire sample;
      divider diver(clk, reset, sample);  // divide the clock into the
sampling rate


      // DAC FSM asks each of the instrument modules for the next sample
      wire busy_a, start_a, busy_b, start_b, busy_c, start_c;
      dac_fsm dacfsm(clk, sample, busy_a, start_a, busy_b, start_b,
busy_c, start_c, dabar);
```

```
      // Intermediate data wires

      wire [ROM_data_length-1:0] piano_out;
      wire [ROM_data_length-1:0] flute_out;
      wire [ROM_data_length-1:0] violin_out;

      wire [ROM_data_length-1:0] piano_temp;
      wire [ROM_data_length-1:0] flute_temp;

      // Convert the instrument outputs to

      wire [10:0] inter_output_a, inter_output_b;

      assign inter_output_a = {~piano_temp[7], {~piano_temp[7],
piano_temp[7:1] - 7'b1000000 } };

      assign inter_output_b = {~flute_temp[7], {~flute_temp[7],
flute_temp[7:1] - 7'b1000000 } };

      wire busypiano, busyflute;


      // The final outputs from the flute and piano

      wire [7:0] result_a;
      wire [7:0] result_b;


      // The internal instruments

      piano instr1(clk, play2, pitchB, reset, start_a, busy_a,
piano_temp);

      flute instr2(clk, play1, pitchA, reset, start_b, busy_b,
flute_temp);


      // ADSR units post-process the tones from the raw instruments

      adsrpiano mypiano(clk, reset, inter_output_a, ~play2, play2,
busypiano, sample, result_a);

      adsrflute myflute(clk, reset, inter_output_b, ~play1, play1,
busyflute, sample, result_b);


      // Synchronize the human volume control buttons
      wire volup_sync, voldown_sync;


      button butter(clk, volup, volup_sync);
      button bitter(clk, voldown, voldown_sync);

      wire[11:0] flute_result;
      wire[11:0] piano_result;
      wire[11:0] violin_result;



      soundcontrol_flute controllerflute(clk, reset, volup_sync,
voldown_sync, result_b, flute_result);
```

```
      soundcontrol_violin controllerviolin(clk, reset, volup_sync,
voldown_sync, violin_sound, violin_result);


      wire [10:0] result_temp;


      // Select volume controlled instruments, or not

      assign result_temp = (vol_control) ?  piano_result[9:2]
+flute_result[10:3]+ violin_result[10:3] :
                                                              result_b
+result_a+ violin_sound;


      // convert back to straight binary
      assign result = {~result_temp[8], result_temp[7:1]};


endmodule
```

Button.v

```
module button(clk, in, out);

      // Module synchronizes a slow, asynchronous human input
      // Converts the level input to a one-clk pulse

      input clk;
      input in;
      output out;

      // Avoid a metastable state with cascaded registers

      reg r1,r2,r3;

      always @ (posedge clk)
      begin
            r1 <= in;
            r2 <= r1;
            r3 <= r2;
      end


      // level to pulse
      assign out = ~r3 & r2;

endmodule
```

Combiner.v

```
module combiner(in_a, in_b, in_c, out);

      // take three 8 bit inputs in straight binary format
      // output an 8 bit number that is the sum of all three, taken as
      // one-fourth of each input, summed.

      // this was not used in the final project, but it was very useful
for testing
      // it has now been partially replaced by val's volume controller


      input [7:0] in_a, in_b, in_c;
      output [7:0] out;

      wire [8:0] result1, result2, result3, result_temp;
      wire [15:0] b_scaled;
      wire [7:0] final_b;

      multer mult(3, in_b, b_scaled);
      //division divider(2, b_multiplied, b_scaled);

      assign final_b = vol_up ? in_b : b_scaled[15:8];

      // convert to two's complement and shift
      assign result1 = {~in_a[7], {~in_a[7], in_a[7:1] - 7'b1000000 } };
      assign result2 = {~in_b[7], {~in_b[7], in_b[7:1] - 7'b1000000 } };
      assign result3 = {~in_c[7], {~in_c[7], in_c[7:1] - 7'b1000000 } };

      // sum them and convert back to straight binary
      assign result_temp = result1 + result2 + result3;

      assign out = {~result_temp[8], result_temp[7:1]};

endmodule
```

Dac_fsm.v

```
module dac_fsm(clk, sample, busy_a, start_a,
      busy_b, start_b, busy_c, start_c, dabar);
      // controls the instrument sampling and output to DAC
      // tells each instrument to interpolate once per sample period

      input clk, sample;
      input busy_a, busy_b, busy_c;       // instrument A is busy

      output start_a, start_b, start_c;         // start instrument A
      output dabar;           // CS and CE bar of the DAC

      reg [1:0] state;
      reg dabar, start_a, start_b, start_c;

      always @ (posedge clk)
      begin

            case (state)
                  0:    begin                         // wait for sample
                        state <= sample ? 1 : state;
                        dabar <= 1;
                        end
                  1:    begin                         // output to DAC
                        state <= 2;
                        dabar <= 0;
                        end
                  2:    begin             // start interpolation(s)
                        state <= (busy_a && busy_b && busy_c) ? 3 :
state;                  //(busy_a && busy_b) ? 3 : state;
                        start_a <= 1;
                        start_b <= 1;
                        start_c <= 1;
                        dabar <= 1;
                        end
                  3:    begin             // wait for interpolation(s)
                        state <= ~(busy_a || busy_b || busy_c) ? 0 :
state;      //~(busy_a || busy_b) ? 0 : state;
                        start_a <= 0;
                        start_b <= 0;
                        start_c <= 0;
                        end
                  default:    state <= 0; // go to wait state
            endcase
      end

endmodule
```

Divider.v

```
module divider(clk, reset_sync, enable);

      // controls the sampling rate for this project
      // the stored waveform samples have 256 datapoints for
      // two complete periods.  the signal should produce a
      // 440 Hz "A" note, so this works out to 558 samples per
      // 1/100 sec, if we are using the 1.8432 MHz clock

      input clk, reset_sync;
      output enable;

      reg [0:10] count;
      reg enable;

      always @ (posedge clk)
      begin
            if (count == 10'd33) // 558 enables per 1/100 sec
            begin
                  enable <= 1;
                  count <= 0;
            end

            else begin

                  enable <= 0;

                  if (reset_sync)
                  begin
                  count <= 0;
                  end

                  else begin
                        count <= count + 1;
                  end
            end
      end

endmodule
```

Instrument.v

```
module instrument(clk, play, note_val, reset, start, busy, result);

      // a generic instrument file.  each instrument will have it's own
ROM
      // the instrument has a note_lookup table, an interp_fsm to
control it
      // and process the start and reset signals, and an interp_calc to
actually
      // calculate the interpolation result.

      parameter ROM_data_length = 8;      // how many data bits
      parameter ROM_addr_length = 8;      // how many address bits
      input clk;
      input play;                   // signal to start playing from the
song fsm
      input reset, start;           // overall control signals
      input [4:0] note_val;   // numerical note / pitch value ("C" "D"
or "E"...)

      output busy;                        // tells dac_fsm it is still
performing interpolation
      output [ROM_data_length-1:0] result;      // the result of this
sample

      wire [9:0] increment_val;     // amount to increase frequency of
stored ROM sample

      wire [4:0] note_val;
      note_lookup_rom notelookup(note_val, increment_val);  // note
value lookup table

      wire [ROM_data_length-1:0] rom_data;
      wire [ROM_addr_length-1:0] rom_addr;
      wire [ROM_data_length-1:0] val_a;
      wire [ROM_data_length-1:0] val_b;
      wire [7:0] fraction;
      wire start_internal;                //FSM doesn't start if play
isn't high
      wire [ROM_data_length-1:0] interp_result; // result is zero if
play isn't high


      assign start_internal = start;


      interp_fsm interpfsm(clk, increment_val, rom_data, reset,
start_internal, busy, val_a, val_b,

                        rom_addr, fraction);    // interpolation fsm

      // this may be replaced with flutewaverom, pianowaverom, or
violinwaverom
      // to produce a flute, piano, or violin instruement.  doing this
produces
      // three separate verilog files

      sinerom romrom(rom_addr, rom_data);

      interp_calc interpcalc(fraction, val_a, val_b, interp_result);
      // interpolation calculator

      reg [7:0] result_internal;

      // because the stored samples produce data values that are
      // straight binary, the "zero" value produced when nothing is
being
      // played should be the average value of the signal, or 128 here

      assign result = play ? interp_result: 128;


endmodule
```

Interp_calc.v

```
module interp_calc(fraction, low_val, high_val, result);//,
high_result);

//    perform linear interpolation given two values and fraction


      parameter ROM_data_length = 8;      // how many data bits
      parameter ROM_addr_length = 8;      // how many address bits


      input [7:0] fraction;   // fractional part of the virtual ROM
address

      input [ROM_data_length-1:0] low_val;      // value in ROM before
the virtual address
      input [ROM_data_length-1:0] high_val;     // value in ROM after
the virtual address

      output [ROM_data_length-1:0] result;      // result of
interpolation (integer)


      // find the fraction complement to multiply the low value by
      wire [7:0] comple_fract = 1 - fraction;


      // Get the weighted part of the high value
      wire [15:0] high_result;
      multer multhigh(high_val, fraction, high_result);

      // Get the weighted part of the low value
      wire [15:0] low_result;
      multer multlow(low_val, comple_fract, low_result);

      // Add them together and output the correct range
      wire [15:0] result_temp = high_result + low_result;

      assign result = result_temp[15:8];


endmodule
```

Interp.fsm.v

```
module interp_fsm(clk, increment, rom_data, reset, start,
            busy, val_a, val_b, rom_addr,
            virtual_rom_addr,
            fraction
            );


// keeps track of a virtual address pointer representing the address of
the sample in ROM
// that we would like to interpolated the value of (i.e. ROM[1.34]).
the module increments
// this value every time it is invoked by the input increment, or it
resets the address if
// reset is asserted.  the module passes the values in the nearest ROM
addresses (1 and 2 if
// the virtual address is 1.34), as well as the fractional part of the
```

```
virtual address (0.34),
// to an interpolator calculation unit.


      parameter ROM_data_length = 8;      // how many data bits
      parameter ROM_addr_length = 8;      // how many address bits

      input clk;
      input start;      // tells the FSM to begin interpolating
      input reset;

      input [9:0] increment;  // amount to increment virtual address to
ROM
                              // amount to increase the frequency of the
sampled data in ROM
                              // top 2 bits are integer part, bottom 8
are fractional part

      input [ROM_data_length-1:0] rom_data;     // sampled data


      output busy;      // indicates that interpolation has not finished
      output [ROM_data_length-1:0] val_a; // first sampled data value
      output [ROM_data_length-1:0] val_b; // second sampled data value
                                          // these values go into the
interpolation computer module,
                                          // which will take a weighted
average of them

      output [ROM_addr_length-1:0] rom_addr;    // real ROM address
      output [ROM_addr_length + 7:0] virtual_rom_addr;

      output [7:0] fraction;

      reg [3:0] state;
      reg [ROM_addr_length + 7:0] virtual_rom_addr;
                                          // the address that we will
interpolate the value of
                                          // top bits are integer part,
bottom 8 are the fractional part

      assign fraction = virtual_rom_addr[7:0];  // fractional part of
virtual rom address

      reg [ROM_addr_length-1:0] rom_addr;
      reg [ROM_data_length-1:0] val_a;
      reg [ROM_data_length-1:0] val_b;


      wire [ROM_addr_length-1:0] low_rom_addr;  // floor{virtual
address}
      wire [ROM_addr_length-1:0] high_rom_addr; // ceiling{virtual
address}

      assign low_rom_addr = {virtual_rom_addr[(ROM_addr_length + 7) :
8]};
      // the lower ROM address is the integer value of the virtual one

      assign high_rom_addr = low_rom_addr + 1;

      // the higher ROM address is one more than the lower one
      reg busy;
      reg [12:0] counter;

      always @ (posedge clk)
      begin
            if (reset)
                  state <= 0;

            else begin

            case (state)
                  0:    begin
                        state <= 1;                               //
reset the virtual address
                        virtual_rom_addr <= 0;
                        busy<=0;
                        end

                  1:    begin state <= start ? 2 : state;         //
wait for start signal
                        busy <=0; end

                  2:    begin
                        state <= 3;                               //
increment the virtual address
                        virtual_rom_addr <= virtual_rom_addr +
increment;
                        busy <= 1;
                        end

                  3:    begin
                        state <= 4;                               //
assert the lower sample address
                        rom_addr <= low_rom_addr[ROM_addr_length-1:0];
                        end

                  4:    begin
                        state <= 5;                               //
read the lower sample data
                        val_a <= rom_data;
                        end

                  5:    begin
                        state <= 6;                               //
assert ths higher sample address
                        rom_addr <= high_rom_addr;
                        end

                  6:    begin
                        state <= 7;                               //
read the lower sample data
                        val_b <= rom_data;
                        end

                  7:    begin
                        state <= 1;
                        end


                  default: state <= 0;

            endcase
            end
      end

endmodule
```

Manual_play.v

```
module manual_play(clk, reset, play1, play2, pitch, result, dabar);

      //    play notes on instruments through manual inputs on FPGA
      //    for testing only

      parameter ROM_data_length = 8;       // how many data bits
      parameter ROM_addr_length = 8;       // how many address bits

      input clk, reset, play1, play2;
      output [ROM_data_length-1:0] result;
      output dabar;

      input [4:0] pitch;
      wire [4:0] pitch;

      wire sample;
      divider diver(clk, reset, sample);

      wire busy_a, start_a;
      dac_fsm dacfsm(clk, sample, busy_a, start_a, busy_b, start_b,
busy_c, start_c, dabar);

      wire [ROM_data_length-1:0] result_temp1;
      wire [ROM_data_length-1:0] result_temp2;

      wire [8:0] result1;

      // add up half the first result plus a fourth of the second result

      assign result1 = {~result_temp1[7], result_temp1 - 8'b10000000 } +
{~result_temp2[7],{~result_temp2[7], result_temp2[7:1] - 7'b1000000 } };

      assign result = {~result1[8], result1[7:1]};

      wire play1, play2;

      // two regular instruments

      instrument instr1(clk, play1, pitch, reset, start_a, busy_a,
result_temp1);

      instrument instr2(clk, play2, 5'b11100, reset, start_b, busy_b,
result_temp2);

endmodule
```

Note_lookup.v

```
module note_lookup(note_val, increment_val);
      // DUMMY note lookup table for testing

      // looks up a note value (between 0 and 24) in a ROM, and converts
it to an
      // increment value.  this is the value by which we must increase
the
      // frequency of the sample in ROM in order to produce the desire
note, like
```

```
      // a "C."

      input [4:0] note_val;   // integer from 0 - 24
                  // 0 is the low C, 12 is middle C, 24 is high C
      output [9:0] increment_val;
                                    // lower 8 bits are fractional
part, higher bits are integer

      assign increment_val = 10'h100;           // output constant for
now

endmodule
```

Song MIF Files

FurElise.mif

```
WIDTH = 8;
DEPTH = 256;


ADDRESS_RADIX = DEC;
DATA_RADIX = DEC;

CONTENT BEGIN
      0     :     1;
      1     :     115;
      2     :     1;
      3     :     64;
      4     :     114;
      5     :     1;
      6     :     64;
      7     :     115;
      8     :     1;
      9     :     64;
      10    :     114;
      11    :     1;
      12    :     64;
      13    :     115;
      14    :     1;
      15    :     64;
      16    :     110;
      17    :     1;
      18    :     64;
      19    :     113;
      20    :     1;
      21    :     64;
      22    :     111;
      23    :     1;
      24    :     64;
      25    :     108;
      26    :     160;
      27    :     1;
      28    :     128;
      29    :     167;
      30    :     1;
      31    :     128;
      32    :     64;
      33    :     172;
      34    :     1;
      35    :     128;
      36    :     99;
      37    :     1;
      38    :     64;
      39    :     103;
```

```
40    :      1;
41    :      64;
42    :      108;
43    :      1;
44    :      64;
45    :      110;
46    :      167;
47    :      1;
48    :      128;
49    :      171;
50    :      1;
51    :      128;
52    :      64;
53    :      174;
54    :      1;
55    :      128;
56    :      103;
57    :      1;
58    :      64;
59    :      107;
60    :      1;
61    :      64;
62    :      110;
63    :      1;
64    :      64;
65    :      111;
66    :      160;
67    :      1;
68    :      128;
69    :      167;
70    :      1;
71    :      128;
72    :      64;
73    :      172;
74    :      1;
75    :      128;
76    :      103;
77    :      1;
78    :      64;
79    :      115;
80    :      1;
81    :      64;
82    :      114;
83    :      1;
84    :      64;
85    :      115;
86    :      1;
87    :      64;
88    :      114;
89    :      1;
90    :      64;
91    :      115;
92    :      1;
93    :      64;
94    :      110;
95    :      1;
96    :      64;
97    :      113;
98    :      1;
99    :      64;
100   :      111;
101   :      1;
102   :      64;
103   :      108;
```

```
      104   :     160;
      105   :     1;
      106   :     128;
      107   :     167;
      108   :     1;
      109   :     128;
      110   :     64;
      111   :     172;
      112   :     1;
      113   :     128;
      114   :     99;
      115   :     1;
      116   :     64;
      117   :     103;
      118   :     1;
      119   :     64;
      120   :     108;
      121   :     1;
      122   :     64;
      123   :     110;
      124   :     167;
      125   :     1;
      126   :     128;
      127   :     171;
      128   :     1;
      129   :     128;
      130   :     64;
      131   :     174;
      132   :     1;
      133   :     128;
      134   :     103;
      135   :     1;
      136   :     64;
      137   :     107;
      138   :     1;
      139   :     64;
      140   :     110;
      141   :     1;
      142   :     64;
      143   :     108;
      144   :     160;
      145   :     1;
      146   :     128;
      147   :     167;
      148   :     1;
      149   :     128;
      150   :     172;
      151   :     1;
      152   :     128;
      153   :     1;
      154   :     64;
      155   :     4;
      156   :     0;
      157   :     0;
            :
            :
            :
      % continued….%

END;
```

SilentNight.mif

```
WIDTH = 8;
DEPTH = 256;
```

```
ADDRESS_RADIX = DEC;
DATA_RADIX = DEC;

CONTENT BEGIN
      0     :     108;
      1     :     165;
      2     :     233;
      3     :     3;
      4     :     64;
      5     :     110;
      6     :     1;
      7     :     64;
      8     :     108;
      9     :     2;
      10    :     64;
      11    :     128;
      12    :     192;
      13    :     105;
      14    :     160;
      15    :     229;
      16    :     6;
      17    :     64;
      18    :     128;
      19    :     192;
      20    :     108;
      21    :     165;
      22    :     233;
      23    :     3;
      24    :     64;
      25    :     110;
      26    :     1;
      27    :     64;
      28    :     108;
      29    :     2;
      30    :     64;
      31    :     128;
      32    :     192;
      33    :     105;
      34    :     160;
      35    :     229;
      36    :     6;
      37    :     64;
      38    :     128;
      39    :     192;
      40    :     115;
      41    :     160;
      42    :     240;
      43    :     3;
      44    :     64;
      45    :     128;
      46    :     192;
      47    :     1;
      48    :     115;
      49    :     160;
      50    :     240;
      51    :     2;
      52    :     64;
      53    :     128;
      54    :     192;
      55    :     112;
      56    :     172;
      57    :     231;
      58    :     5;
```

```
59    :     64;
60    :     128;
61    :     192;
62    :     1;
63    :     113;
64    :     172;
65    :     243;
66    :     3;
67    :     64;
68    :     128;
69    :     192;
70    :     1;
71    :     113;
72    :     172;
73    :     243;
74    :     2;
75    :     64;
76    :     128;
77    :     192;
78    :     108;
79    :     165;
80    :     233;
81    :     6;
82    :     64;
83    :     128;
84    :     192;
85    :     110;
86    :     165;
87    :     234;
88    :     3;
89    :     64;
90    :     128;
91    :     192;
92    :     1;
93    :     110;
94    :     165;
95    :     234;
96    :     2;
97    :     64;
98    :     128;
99    :     192;
100   :     113;
101   :     234;
102   :     165;
103   :     3;
104   :     64;
105   :     112;
106   :     1;
107   :     64;
108   :     110;
109   :     2;
110   :     64;
111   :     128;
112   :     192;
113   :     108;
114   :     165;
115   :     233;
116   :     3;
117   :     64;
118   :     110;
119   :     1;
120   :     64;
121   :     108;
122   :     2;
```

```
123   :     64;
124   :     128;
125   :     192;
126   :     105;
127   :     165;
128   :     6;
129   :     64;
130   :     128;
131   :     110;
132   :     165;
133   :     234;
134   :     3;
135   :     64;
136   :     128;
137   :     192;
138   :     1;
139   :     110;
140   :     165;
141   :     234;
142   :     2;
143   :     64;
144   :     128;
145   :     192;
146   :     113;
147   :     234;
148   :     165;
149   :     3;
150   :     64;
151   :     112;
152   :     1;
153   :     64;
154   :     110;
155   :     2;
156   :     64;
157   :     128;
158   :     192;
159   :     108;
160   :     165;
161   :     233;
162   :     3;
163   :     64;
164   :     110;
165   :     1;
166   :     64;
167   :     108;
168   :     2;
169   :     64;
170   :     128;
171   :     192;
172   :     105;
173   :     165;
174   :     6;
175   :     64;
176   :     128;
177   :     115;
178   :     240;
179   :     160;
180   :     3;
181   :     64;
182   :     192;
183   :     1;
184   :     115;
185   :     240;
186   :     2;
```

```
187   :     64;
188   :     128;
189   :     192;
190   :     118;
191   :     160;
192   :     231;
193   :     3;
194   :     64;
195   :     115;
196   :     1;
197   :     64;
198   :     112;
199   :     2;
200   :     64;
201   :     128;
202   :     192;
203   :     113;
204   :     165;
205   :     236;
206   :     5;
207   :     64;
208   :     128;
209   :     192;
210   :     1;
211   :     117;
212   :     165;
213   :     236;
214   :     5;
215   :     64;
216   :     128;
217   :     192;
218   :     1;
219   :     113;
220   :     165;
221   :     224;
222   :     2;
223   :     64;
224   :     108;
225   :     2;
226   :     64;
227   :     105;
228   :     2;
229   :     64;
230   :     128;
231   :     192;
232   :     160;
233   :     108;
234   :     3;
235   :     64;
236   :     106;
237   :     1;
238   :     64;
239   :     103;
240   :     2;
241   :     64;
242   :     128;
243   :     113;
244   :     172;
245   :     12;
246   :     64;
247   :     128;
248   :     12;
249   :     0;
250   :     0;
```

```
      251   :     0;
      252   :     0;
      253   :     0;
      254   :     0;
      255   :     0;

END;
```

Python Code

ROM_generator.py

```
# Makes a ROM of specified length and width containing
# coefficients for a number of periods of a sine wave.

import math

depth = 256      # number of lines in the ROM
width = 8        # width of ROM (in bits)
periods = 2     # number of periods per ROM

highest = int(math.pow(2, width))

romfile = open('c://sineReg.mif','w')

romfile.write('WIDTH = ' + str(highest) + ';\n')
romfile.write('DEPTH = ' + str(depth) + ';\n')
romfile.write('\n\n')
romfile.write('ADDRESS_RADIX = DEC;\n')
romfile.write('DATA_RADIX = DEC;\n\n')
romfile.write('CONTENT BEGIN\n')

for address in range(depth):
      num = address * (periods*2*math.pi)/depth
      result = math.sin(num)
      data = int(round(highest/2 * result + highest/2))
      line = '\t' + str(address) + '\t:\t' + str(data) + ';\n'
        romfile.write(line)
      print line


romfile.write('\nEND;\n')

romfile.close()
```

Songwriter.py

```
# creates a .mif file from pseudo code in a text file
# ROM length is always 256

import string

fundamental = 1 # clicks per pause
length = 256  # length of ROM

# convert a line of pseudocode to a .mif line
def convert_line(line):
    end = string.find(line, '\n')
    print line

    # off signal
    if (line[0:3] == 'off'):
        instrument = int(line[3:end])
        message = get_instrument(instrument)

    # pause signal
    elif (line[0] == 'p'):
        message = int(line[1:end])

    # on signal
    else:
        split = string.find(line, 'o')
        instrument = int(line[0:split])
        note = int(line[split+1:end])

        message = 32 + note + get_instrument(instrument)

    return str(message)

# based on instrument, get the decimal value for the instruction
def get_instrument(instr):
    if (instr==1):
        return 64
    elif (instr==2):
        return 128
    else:
        return 192


# input file name
filein = open('c:\\python23\\work\\silentnight.txt','r')
lines = filein.readlines()


filein.close

# output filename
fileout = open('c:\\python23\\work\\silentnight.mif','w')

# beginning of ROM file
fileout.write('WIDTH = 8;\n')
fileout.write('DEPTH = ' + str(length) + ';\n')
fileout.write('\n\n')
fileout.write('ADDRESS_RADIX = DEC;\n')
fileout.write('DATA_RADIX = DEC;\n\n')
fileout.write('CONTENT BEGIN\n')

outputlines = []

for i in range(length):
    outputlines.append('0')

counter = 0

# step through all the lines in the file
for line in lines:
    # skip comment lines beginning with -----
    if (line[0] != '-'):
        outputlines[counter] = convert_line(line)
        counter = counter + 1

print outputlines

# create the .mif file
for address in range(length):
    data = outputlines[address]
    theline = '\t' + str(address) + '\t:\t' + str(data) + ';\n'
    fileout.write(theline)

fileout.write('\nEND;\n')

fileout.close()
```

Appendix C: Valerie Gordeski's Implementation with .mif files

ADSRPIANO.V

```
// This module sets the envelope shape for the piano
module ADSRpiano(clk, reset, inter_output, endtime, start, busy,
enable, result);

input start, enable, reset, clk, endtime;
input[7:0] inter_output;

output busy;
output[7:0] result;

reg[6:0] atkptr, decayptr, susptr; // pointers used to keep track of
//where we are in memory
reg[14:0] counter;
reg[8:0] counter_temp;
reg[1:0] state;
reg[6:0] addressrom;
reg busy;
wire[7:0] q1;
wire[7:0] q2;
wire[15:0] resultmult;

parameter attack=1;
parameter decay=2;
parameter sustain=3;
parameter idle=0;

assign q2 = inter_output; // this is what goes into the multiplier
assign result = resultmult[15:8]; // we select highest bits of the
//result as our output

// this rom stores the envelope coefficients
pianoROM mypianorom (addressrom, q1); // q1 is unsigned


// this multiplier multiplies the coefficients by the interpolator
//output
pianomult mymult (q1, q2, resultmult);

always @ (posedge clk)
begin
```

```
if (reset)
        begin
                state<=idle;
        end
else case (state)
attack:
        begin
        addressrom<=atkptr; // keep track of address
        if (enable)
        begin
        if (endtime)
                state<=idle;
        else if (counter == 2618) // change state
        //else if (counter == 30)
                begin
                        counter<=0;
                        counter_temp<=0;
                        atkptr<=0;
                        state<=decay;
                end
        else if (counter_temp == 109) // increment pointer
        //else if (counter_temp == 5)
                begin
                        counter_temp<=0;
                        counter<=counter+1;
                        atkptr <= atkptr+1;
                        state <= state;
                end
        else
                begin
                        counter_temp<=counter_temp+1;
                        counter<=counter+1;
                        state<=state;
                end
        end
        end
decay:
        begin
        if (enable)
        begin
                addressrom<=decayptr;
        if (endtime)
                state<=idle;
        else if (counter == 872)
        //      else if (counter == 10)
                        begin
                                counter<=0;
                                counter_temp<=0;
                                state<=sustain;
                                decayptr<=24;
                        end
                else if (counter_temp == 109)
                //else if (counter_temp == 5)
                        begin
                        counter_temp<=0;
                        counter<=counter+1;
                        decayptr<= decayptr+1;
                        state<=state;
                        end
                else
                        begin
                        counter_temp<=counter_temp+1;
                        counter<=counter+1;
                        state<=state;
                        end
                end
        end
sustain:
        begin
        addressrom<=susptr;
                if (endtime)
                        begin
                                counter<=0;
                                counter_temp <=0;
                                state <= idle;
                        end
        else if (enable)
                begin
                        if (susptr==127)
                                begin
                                        susptr<=119;
                                        counter<=0;
                                        counter_temp<=0;
                                        state<=state;
                                end
                        else if (counter_temp == 109)
                        //else if (counter_temp == 5)
                                begin
                                        counter_temp<=0;
                                        susptr<=susptr+1;
                                end
                        else
                                begin

counter_temp<=counter_temp+1;
                                        state<=state;
                                end
                end // elseif enable
        end // sustain
idle:
        begin
                atkptr<=0;
                decayptr<=24;
                susptr<=32;
                busy<=0;
                counter<=0;
                counter_temp<=0;
                if (start)
                        begin
                                busy<=1;
                                state<=attack;
                        end
                else state<=state;
        end
        default: state<= idle;
endcase

end// always

endmodule
```

ADSRFLUTE.V

```
// sets the evelope shape for the flute
module ADSRflute(clk, reset, inter_output, endtime, start, busy,
enable, result);

input start, enable, reset, clk, endtime;
input[7:0] inter_output;

output busy;
output[7:0] result;

reg[6:0] atkptr, decayptr;
reg[14:0] counter;
reg[8:0] counter_temp;
reg[2:0] state;
reg[6:0] addressrom;
reg busy;
wire[7:0] q1;
wire[7:0] q2;
wire[15:0] resultmult;

parameter attack=1;
parameter decay=2;
parameter idle=0;

assign q2 = inter_output;
assign result = resultmult[15:8]; // we select highest bits of the
//result after multiplication

fluteROM myfluterom (addressrom, q1); // multiplication is signed
flutemult myflutemult (q1, q2, resultmult);

always @ (posedge clk)
        begin
                if (reset)
                        begin
                                state<=idle;
                        end
                else case (state)
                attack:
                        begin
                        addressrom<=atkptr;
                        if (endtime)
                                begin
                                        busy<=0;
                                        state<=idle;
                                end
```

```
                              if (enable)
                                    begin
                                    if (counter == 6981)
                                    //if (counter == 30)
                                          begin
                                                counter<=0;
                                                counter_temp<=0;
                                                atkptr<=0;
                                                state<=decay;
                                          end
                                    else if (counter_temp == 109)
                                    //else if (counter_temp == 5)
                                          begin
                                                counter_temp<=0;
                                                counter<=counter+1;
                                                atkptr <= atkptr+1;
                                                state <= state;
                                          end
                                    else
                                          begin

                  counter_temp<=counter_temp+1;
                                                counter<=counter+1;
                                                state<=state;
                                          end
                                    end
                              end
                        decay:
                              begin
                              addressrom<=decayptr;
                              if (endtime)
                                    begin
                                          counter<=0;
                                          counter_temp <=0;
                                          state <= idle;
                                    end
                              else if (enable)
                                    begin
                                          if (decayptr==92) // decayptr
            //keeps looping, giving the flute its fluttering sound
                                                begin
                                                      decayptr<=35;
                                                      state<=state;
                                                end
                                          else if (counter_temp == 109)
                                          //else if (counter_temp == 5)
                                                begin
                                                decayptr<=decayptr+1;
                                                      counter_temp<=0;
                                                end
                                          else
                                                begin
                                          counter_temp<=counter_temp+1;
                                          state<=state;
                                                end
                                    end // elseif enable
                              end // begin
```

```
                idle:
                        begin
                                atkptr<=0;
                                decayptr<=64;
                                busy<=0;
                                counter<=0;
                                counter_temp<=0;
                                if (start)  // at the start signal, begin
                                        begin
                                                busy<=1;
                                                state<=attack;
                                        end
                                else state<=state;
                        end
        endcase
        end// always

endmodule
```

ADSRVIOLIN.V

```
// this module follows the model of the piano ADSR
module ADSRviolin(clk, reset, inter_output, endtime, start, busy,
enable, result);

input start, enable, reset, clk, endtime;
input[7:0] inter_output;

output busy;
output[7:0] result;

reg[6:0] atkptr, susptr;
reg[14:0] counter;
reg[8:0] counter_temp;
reg[1:0] state;
reg[6:0] addressrom;
reg busy;
wire[7:0] q1;
wire[7:0] q2;
wire[15:0] resultmult;

parameter attack=1;
parameter sustain=2;
parameter idle=0;

assign q2 = inter_output;
assign result = resultmult[15:8];

violinROM myviolinrom (addressrom, q1); // q1 is signed
violinmult myviolinmult (q1, q2, resultmult);

always @ (posedge clk)
begin
        if (reset)
                begin
```

```
                        state<=idle;
                end
        else case (state)
        attack:
                begin
                addressrom<=atkptr;
                if (endtime) // if finished, then go to idle state
                        begin
                                counter<=0;
                                counter_temp <=0;
                                state <= idle;
                        end
                else if (enable)
                        begin
                        if (counter == 2618) // set counter value to
switch states
                        //if (counter == 30) debugging
                                begin
                                        counter<=0;
                                        counter_temp<=0;
                                        atkptr<=0;
                                        state<=sustain;
                                end
                        else if (counter_temp == 109)//increments ptrs
                        // else if (counter_temp == 5)
                                begin
                                        counter_temp<=0;
                                        counter<=counter+1;
                                        atkptr <= atkptr+1;
                                        state <= state;
                                end
                        else
                                begin
                                        counter_temp<=counter_temp+1;
                                        counter<=counter+1;
                                        state<=state;
                                end
                        end // enable
                end // attack
                sustain:
                        begin
                        addressrom<=susptr;
                        if (endtime)
                                begin
                                        counter<=0;
                                        counter_temp <=0;
                                        state <= idle;
                                end
                        else if (enable)
                                begin
                                        if (susptr==127) // keeps looping
//in the sustain region until entime signal is obtained
                                                begin
                                                        susptr<=24;
                                                        counter<=0;
                                                        counter_temp<=0;
                                                        state<=state;
                                                end
                                        //else if (counter_temp == 109)
                                        else if (counter_temp == 5)
                                                begin
                                                        counter_temp<=0;
                                                        susptr<=susptr+1;
                                                end
                                        else
                                                begin

        counter_temp<=counter_temp+1;
                                                        state<=state;
                                                end
                                end //elseif enable
                        end // sustain
        idle:
                begin
                        atkptr<=0;
                        susptr<=24;
                        busy<=0;
                        counter<=0;
                        counter_temp<=0;
                        if (start)
                                begin
                                        busy<=1;
                                        state<=attack;
                                end
                        else state<=state;
                end
        default: state<= idle;
endcase

end// always

endmodule

// This module controls the volume for the flute. The other two modules
// controlling the volume for the piano and the violin remain
//uncommented because they follow exact the same patter
module soundcontrol_flute (clk, reset, volup, voldown, in, out);

input clk, reset, volup, voldown;
input[7:0] in;

output[11:0] out;
wire[11:0] volumeup1, volumeup2, volumedown1, volumedown2;
reg     [2:0] remainder_a, remainder_b, remainder_c, remainder_d;
reg[2:0] count;


wire[11:0] out_temp;
wire[12:0] product1, product3, product4;

// these dividers and multipliers provide the right fraction by which
to multiply
soundcontrol_divider volup1flute (product1, 7, volumeup1, remainder_a);
soundmult multvolup1flute (in, 8, product1);
```

```
soundcontrol_divider voldown1flute (product3, 4, volumedown1,
remainder_c);
soundmult multvoldown1flute (in, 3, product3);


soundcontrol_divider voldown2flute (product4, 4, volumedown2,
remainder_d);
soundmult multvoldown2flute (in, 2, product4);


// this out temp checks which count value has been obtained, therefore
//choosing the right output of the multiplication for its output
assign out_temp = (count==2) ? in :
                       ((count==3) ? volumeup1 :
                       ((count==1) ? volumedown1 :
                       ((count==0) ? volumedown2 : out_temp)));

assign out = out_temp;

// this clocked block keeps track of the number of button presses, and
//assigns the count value which determines whether the volume is going
//up or down.
always @ (posedge clk)
begin
 if (reset)
        count<=2;
 else if (volup)
        if (count == 3)
               count<=count;
        else count<=count+1;
 else if (voldown)
        begin
               if (count == 0)
                      count<=count;
               else
               count<=count-1;
        end
else
        count<=count;
end

endmodule


module soundcontrol_piano (clk, reset, volup, voldown, in, out);

input clk, reset, volup, voldown;
input[7:0] in;

output[11:0] out;
wire[11:0] volumeup1, volumeup2, volumedown1, volumedown2;
reg     [2:0] remainder_a, remainder_b, remainder_c, remainder_d;
reg[2:0] count;
```

```
wire[11:0] out_temp;
wire[12:0] product1, product3, product4;


soundcontrol_divider volup1 (product1, 7, volumeup1, remainder_a);
soundmult multvolup1 (in, 8, product1);


soundcontrol_divider voldown1 (product3, 4, volumedown1, remainder_c);
soundmult multvoldown3 (in, 3, product3);


soundcontrol_divider voldown2 (product4, 4, volumedown2, remainder_d);
soundmult mult (in, 2, product4);

assign out_temp = (count==2) ? in :
                                       ((count==3) ? volumeup1 :
                                                       ((count==1) ?
volumedown1 :

        ((count==0) ? volumedown2 : out_temp)));

assign out = out_temp;

always @ (posedge clk)
begin
 if (reset)
        count<=2;
 else if (volup)
        if (count == 3)
                count<=count;
        else count<=count+1;
 else if (voldown)
        begin
                if (count == 0)
                        count<=count;
                else
                count<=count-1;
        end
else
        count<=count;
end

endmodule


module soundcontrol_violin (clk, reset, volup, voldown, in, out);

input clk, reset, volup, voldown;
input[7:0] in;

output[11:0] out;
wire[11:0] volumeup1, volumedown1, volumedown2;
reg     [2:0] remainder_a, remainder_b, remainder_c, remainder_d;
reg[2:0] count;

wire[11:0] out_temp;
```

```
wire[12:0] product1, product3, product4;


soundcontrol_divider volup1viol (product1, 7, volumeup1, remainder_a);
soundmult multvolup1viol (in, 8, product1);


soundcontrol_divider voldown1viol (product3, 4, volumedown1,
remainder_c);
soundmult multvoldown1viol (in, 3, product3);


soundcontrol_divider voldown2viol (product4, 4, volumedown2,
remainder_d);
soundmult multvoldown2viol (in, 2, product4);

assign out_temp = (count==2) ? in :
                                      ((count==3) ? volumeup1 :
                                                       ((count==1) ?
volumedown1 :

        ((count==0) ? volumedown2 : out_temp)));

assign out = out_temp;

always @ (posedge clk)
begin
 if (reset)
        count<=2;
 else if (volup)
        if (count == 3)
                count<=count;
        else count<=count+1;
 else if (voldown)
        begin
                if (count == 0)
                        count<=count;
                else
                count<=count-1;
        end
else
        count<=count;
end

endmodule

// This was just a test module for integration.  An approximate copy of
//this module appreas in midi_synth.v

module ADSRoutputs(clk, reset, busypiano, busyflute, busyviolin,
in_piano, in_flute, in_violin,
enablea, enableb, enablec, endtimea, endtimeb, endtimec, starta,
startb, startc, out, volcontrol, volup, voldown);

input clk, reset, enablea, enableb, enablec, endtimea, endtimeb,
endtimec, starta, startb, startc;
input[7:0] in_piano, in_flute, in_flute;
```

```
output busypiano, busyflute, busyviolin;
output[7:0] out;
wire inter_output_a, inter_output_b, inter_outer_c;
reg[8:0] result_temp;

// this part changes the output levels from 0-255 to -128-128. converts
//to 2’s complement
assign inter_output_a = {~in_piano[7], {~in_piano[7], in_piano[7:1] -
7'b1000000 } };
assign inter_output_b = {~in_flute[7], {~in_piano[7], in_piano[7:1] -
7'b1000000 } };
assign inter_output_c = {~in_violin[7], {~in_violin[7], in_violin[7:1]
- 7'b1000000 } };

// this part passes everything through the adsr
adsrpiano mypiano(clk, reset, inter_output_a, endtimea, starta,
busypiano, enablea, result_a);
adsrflute myflute(clk, reset, inter_output_b, endtimeb, startb,
busyflute, enableb, result_b);
adsrviolin myviolin(clk, reset, inter_output_c, endtimec, startc,
busyviolin, enablec, result_c);

wire[11:0] pianovol, flutevol, violinvol;
input[1:0] volcontrol;

input volup, voldown;
button volumeup (clk, volup, volup_synch);
button volumedown (clk, voldown, voldown_synch);

// this part takes the adsr outputs and puts them through volume
//control
soundcontrol_piano spiano (clk, reset, volup_synch, voldown_synch,
result_a, pianovol);
soundcontrol_violin sviolin (clk, reset, volup_synch, voldown_synch,
result_c, violinvol);
soundcontrol_flute sflute (clk, reset, volup_synch, voldown_synch,
result_b, flutevol);


// if the volume control switch is up, then the controlled signals are
// obtained. If not, then the regular signals are obtained

assign result_temp = (volcontrol) ?
                       pianovol[10:3]+ violinvol[11:4]+flutevol[11:4] :
                                      result_a + result_b + result_c;

assign out = {~result_temp[8], result_temp[7:1]};


endmodule
```

Waveform .mif files:

```
// This .mif file is for violin only. In order to save space, I
//(Valerie) did not include the other .mif files. If this is really
//crucial, I can attach them separately.  Thank you.

WIDTH=8;
DEPTH=256;

ADDRESS_RADIX = HEX;  % Address and data radixes are optional, default
is hex %
DATA_RADIX = DEC;

CONTENT BEGIN
00: 128;                 20:184;                  40:122;
01:147;                  21:178;                  41:115;
02:165;                  22:172;                  42:109;
03:183;                  23:165;                  43:103;
04:199;                  24:158;                  44:99;
05:213;                  25:151;                  45:96;
06:226;                  26:144;                  46:94;
07:236;                  27:138;                  47:93;
08:245;                  28:133;                  48:95;
09:250;                  29:129;                  49:97;
0A:254;                  2A:126;                  4A:100;
0B:255;                  2B:124;                  4B:105;
0C:255;                  2C:124;                  4C:109;
0D:254;                  2D:125;                  4D:114;
0E:251;                  2E:127;                  4E:119;
0F:247;                  2F:131;                  4F:124;
10:238;                  30:135;                  50:127;
11:233;                  31:140;                  51:130;
12:229;                  32:145;                  52:132;
13:225;                  33:150;                  53:132;
14:221;                  34:154;                  54:131;
15:218;                  35:158;                  55:129;
16:215;                  36:161;                  56:125;
17:213;                  37:162;                  57:120;
18:210;                  38:163;                  58:114;
19:209;                  39:161;                  59:108;
1A:206;                  3A:159;                  5A:101;
1B:204;                  3B:155;                  5B:94;
1C:202;                  3C:150;                  5C:87;
1D:198;                  3D:144;                  5D:80;
1E:194;                  3E:137;                  5E:74;
1F:190;                  3F:129;                  5F:68;
60:63;                   6D:25;                   7A:37;
61:59;                   6E:20;                   7B:51;
62:56;                   6F:15;                   7C:67;
63:53;                   70:11;                   7D:84;
64:50;                   71:6;                    7E:102;
65:48;                   72:3;                    7F:120;
66:46;                   73:1;                    80:139;
67:44;                   74:0;                    81:158;
68:42;                   75:1;                    82:176;
69:40;                   76:4;                    83:193;
6A:37;                   77:9;                    84:208;
6B:33;                   78:16;                   85:221;
6C:29;                   79:26;                   86:232;
```

```
87:242;
88:248;
89:253;
8A:255;
8B:255;
8C:255;
8D:252;
8F:249;
90:245;
91:240;
92:235;
93:231;
94:226;
95:222;
96:219;
97:216;
98:213;
99:211;
9A:209;
9B:207;
9C:205;
9D:203;
9E:200;
9F:196;
A0:192;
A1:187;
A2:181;
A3:174;
A4:168;
A5:161;
A6:154;
A7:147;
A8:141;
A9:135;
AA:130;
AB:127;
AC:125;
AD:124;
AE:125;
AF:126;
B0:129;
B1:133;
B2:138;
B3:143;
B4:148;
B5:152;
B6:156;
B7:160;
B8:162;
B9:163;
BA:162;
BB:160;
BC:157;
BD:152;
BE:146;
BF:140;
C0:132;
C1:125;
C2:118;
C3:111;
C4:105;
C5:100;
C6:97;
C7:94;
C8:93;
C9:94;
CA:96;
CB:99;
CC:103;
CD:107;
CE:112;
CF:117;
D0:122;
D1:126;
D2:129;
D3:131;
D4:132;
D5:131;
D6:130;
D7:126;
D8:122;
D9:117;
DA:111;
DB:104;
DC:97;
DD:90;
DE:83;
DF:77;
E0:71;
E1:65;
E2:61;
E3:57;
E4:54;
E5:     51;
E6: 49;
E7:     47;
E8:     45;
E9:     43;
EA:     41;
EB:     38;
EC:     35;
ED: 31;
EE:     26;
EF:     22;
F0:     17;
F1:     12;
F2:     8;
F3:     4;
F4:     2;
F5:     0;
F6:     0;
F7:     2;
F8:     7;
F9:     13;
FA:     21;
FB:     32;
FC:     45;
FD:     60;
FE:     77;
FF:     94;

end;
```