University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences
Computer Science Division

CS 150
Spring 2002

J. Wawrzynek
Project Info.

Final Project Specification
MIDI Sound Synthesizer
Version 0.5

1 Introduction

For the final project you are required to use an FPGA board to build a "box" that takes a MIDI signal as input and generates a audio waveform as output. Figure 1 shows a high level view of the synthesizer that you will build. MIDI is an acronym for Musical Instrument Digital Interface, and a MIDI signal is a bit-serial stream of bytes. The audio waveform is a mono (as opposed to stereo) signal. It will be strong enough to drive headphones or small speakers. The MIDI synthesizer is monophonic, meaning that it will translate MIDI signals into sound not more than one note at a time, and single channel, meaning that it will produce the voice of only one instrument. The audio waveforms are generated using a technique called *waveform synthesis*; waveforms from actual musical instruments are stored in ROM and used to generate sound in response to MIDI commands. The sound waveforms are stored and played-back using 16-bit data samples. Output sampling rate is 31.25KHz.

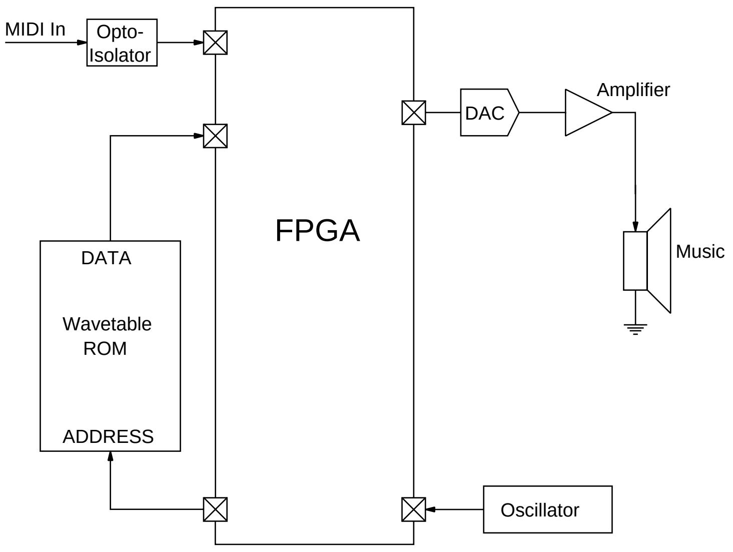

Figure 1: High-level view of the MIDI Synthesizer.

Our project is a simplified version of commercial synthesizers. For comparison, a medium grade commercial sound box has stereo output, is polyphonic, and is capable of generating 30 notes simultaneously. Note pitch can be varied continuously, and the voices include numerous digital filters for adding special effects to the sound output. Such systems cost around several hundred dollars.

1.1 Sound and Music Theory

Sound is air vibrating at an audible frequency, typically 20Hz – 20KHz for an adult human. The amount of displacement can be sampled and recorded as a sequence of magnitudes over time, and reproduced by speakers or headphones.

Our hearing is quite complex in the way we perceive musical tones. Two of the most important characteristics of a musical tone is its loudness and its pitch. To a first approximation human hearing is logarithmic in perceiving both loudness and pitch. In the case of loudness, this means that a we perceive the loudness as being proportional to the logarithm of the sound wave amplitude.

In the case of pitch, the human auditory system is very keen at detecting the logarithmic relationships between frequency, and “musical” pitch intervals. The simplest interval to detect is the octave, where the higher pitch has double the frequency of the lower pitch. Given those two frequencies, it is possible to subdivide the interval (which is the multiple 2) into twelve parts, producing eleven *semi-tones* in between. These twelve semi-tones form the chromatic scale of the traditional western *twelve-tone scale*.

These twelve intervals form a geometric series. As we move from one up to the next we can determine its frequency by multiplying by $\sqrt[12]{2}$. After 12 such multiplications we will have doubled the frequency and reached the octave. Most people can detect a pitch difference much smaller than these 12 divisions, in fact as small as a few hundredths of a semi-tone. In musical terms a *cent* is 1/100 of a *semi-tone*. To increase the pitch of a note by one cent, multiply its frequency by $\sqrt[1200]{2}$.

The note called middle C has a frequency of 261.63Hz. The MIDI encoding for that note is key 60. The note called high C, which is an octave higher, has a double the frequency—523.25Hz, and it has a MIDI encoding of 72. Others tones can be produced by multiplying and dividing the frequency by factors of $\sqrt[12]{2}$. For example, MIDI note number 61 has a frequency of 261.63kHz$\times \sqrt[12]{2}$.

There is one more important aspect to a musical sound, called *timbre*. Whereas pitch is indication of how often a sound wave repeats, and loudness is the way we perceive the amplitude of a sound, timbre is a perception of the *shape* of the sound wave. Sometimes the timbre of a musical tone is called its tone *quality* or *color*. Different musical instruments have different, and readily identifiable, timbres. Like our perception of other characteristics of a musical tone, timbre is complex, but it is deeply related to the way a musical note starts, the shape the waveform takes as it repeats itself, and how it dies out.

For many musical instruments, a simple model can be used to describe the shape of the waveform. The waveform can be split into three parts: an attack, a sustain, and a release. The attack is the most interesting part of a musical sound, corresponding to when a note is first played. It typically lasts for no more than 300ms, and during this time, the waveform may be very irregular. Most of the rest of the musical sound is very repetitive, and it is called the sustain. In the sustain section, not much would be missing to the ear if just one cycle were played over and over again for the duration of the rest of the note. At the end of the note, there may also be irregularities in the waveform, and the release consists of the dying away of the sound. The release is typically shorter than 100ms.

This simple model matches some musical instrument better than others. Notes from flutes and other woodwind instruments and bowed sounds such as from the violin family, fit nicely into this model. However, sounds from instruments that are struck, such as a marimba or even a piano don't really have a sustain part to their notes. In these cases we could simply omit the sustain part of the model, or probably more simply represent the entire note as an attack only.

1.2 Sampling and Storing

A 512KB EPROM will be used to store the sound information. At a sample rate of 31.25KHz, and at 2 bytes per sample, the EPROM can store roughly 8 seconds of raw sound. However, it must be capable of reproducing notes of maybe 100 different pitches at varying durations.

The simplest way of getting arbitrary duration is to store an attack, a short section of the sustain, and a release in a *note template*. Then, to generate a note, the synthesizer will play the attack and continue playing into the sustain. While the note is held, it will continue to play the sustain part by *looping*. When the key is released, to signal the end of the note, the synthesizer will continue playing the sustain to the end of the current loop iteration and then play the release portion of the stored note. This process for playing a note is illustrated in figure 2.

One way of generating an arbitrary pitch is to store just one note template of an instrument at a known pitch, and varying the playback frequency. Early inexpensive synthesizers varied the play back frequency of a note by adjusting the sampling frequency. This creates problems in the system design and will not be used here. Instead we will output all notes with a constant sampling frequency and will vary the *stride* with which we step through the samples in the template. For example, if we stored the note middle C (MIDI key 60) sampled at a rate of 31.25KHz, playing the note back by taking every sample from the template would result in another middle C. However, if we take every other sample from the template (stride = 2), the frequency, and thus the pitch, of the note would double, resulting in high C (MIDI key 72). Generating notes with a frequency in between these two requires the use of non-integer stride and thus a non-integer index into the template. In our design, we will ignore the fractional part of the index when looking up samples stored in the template. However, the index must be maintained as a non-integer number internal to the synthesizer logic. Truncating the index when looking up samples in the template, results in a slight distortion of the waveform, but greatly simplifies the logic. You may choose to earn extra credit by modifying your design to account for the fractional part of the index. As described later in this document, linear interpolation performed on the closest two samples stored in the template results in a more accurate waveform.

An interesting property of musical instruments is that the timbre of the sound varies from note to note over the range of the instrument. Notes played in the high register of a

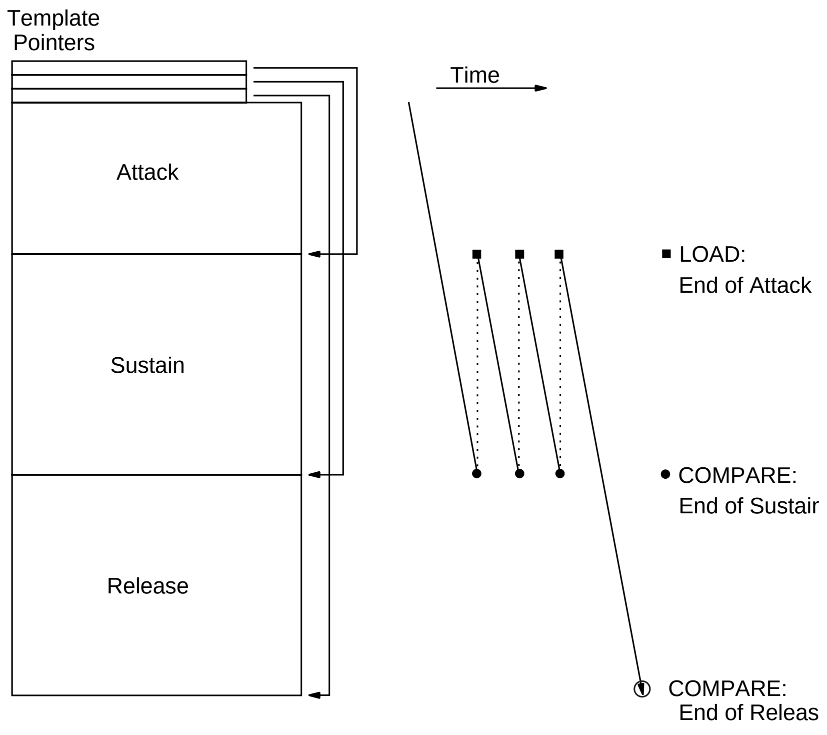

Figure 2: Note Playing Process.

saxophone, while still characteristically saxophone like, have a different quality than the low notes.[1] The consequence of this in our synthesizer is that one note template will not be enough to reproduce all the notes of a particular instrument. Instead we will store a set of note templates, usually two per octave, and then choose one from the set when it it time to generate a note. This mechanism will be implemented in our synthesizers using a template *directory*, as illustrated in figure 3. For each MIDI key number there will be a *directory entry* stored, indicating which note template to use and what step-size to use when stepping through the template.

Note that this directory mechanism relieves the synthesizer hardware from having to perform the costly multiplications by the $\sqrt[12]{2}$. These ratios are all precomputed and encoded as the step-sizes in the directory.

Generating notes with arbitrary loudness is simply a matter of scaling the amplitude of the sample values as they come out of the template.

Figure 3 shows the layout of the EPROM used to store the note information. Directory entries are stored consecutively starting from address zero. They are indexed by MIDI key numbers, ranging from 0 to 127. As shown in figure 4 each 4-byte entry contains a 20-bit pointer and a 12-bit step-size.

The 12-bit step-size is interpreted as a *fixed point* number, with the decimal point fixed after the highest two bits. Its value is therefore equal to its integer interpretation divided by 2^{10}.

[1]It is also the case that the timbre of a note varies with the intensity that it is played.

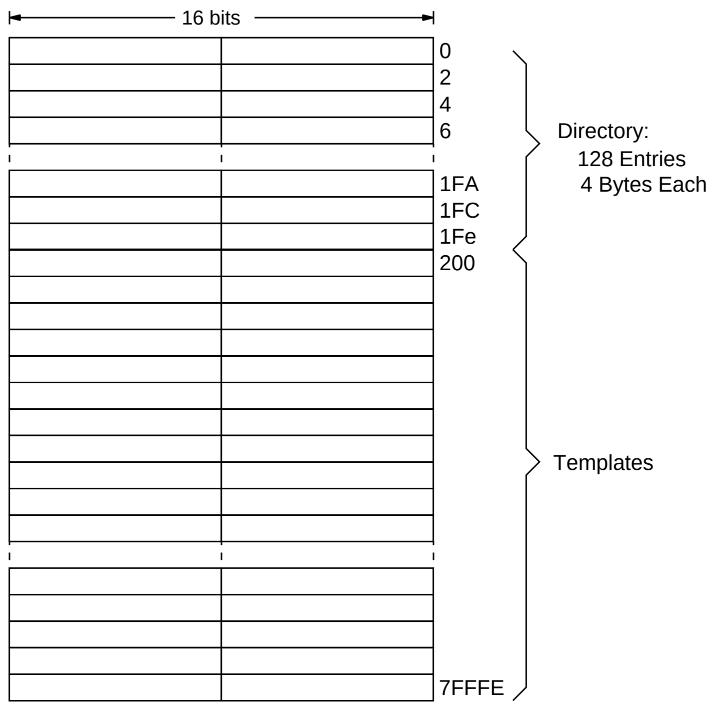

Figure 3: EPROM Layout.

After the directory, the remaining space stores the note templates. Figure 5 shows the layout of a single template. Templates are variable length, and need not be at any particular location or in any particular order. However, they must start at a two-byte boundary. The nine bytes store three 20-bit addresses that point to the last sample of the attack, the last sample of the sustain, and the last sample of the decay. After that, on the next even byte boundary, is stored samples from the attack, sustain, and release sections of the note. Each sample is 16-bit two's complement number.

We will provide you with sample template files and a program for converting these to EPROM format. The format of the template files is straight forward. If you are interested, and you have access to sound samples, you are encouraged to generate your own template files.

1.3 MIDI

MIDI enables music synthesizers, sequencers, home computers, etc. to be interconnected through a standard interface. MIDI signals are sent to a device, such as a synthesizer, via an opto-isolator and a UART. The opto-isolator uses an LED and a photo-detector to isolate the electronics of the receiving device from that of the sender. It is usually the case that the two devices are physically separated by some distance and using the opto-isolator allows the two systems to have independent (and maybe) different ground potentials. The result is a a more reliable system.

MIDI uses a current loop to send information. To signal a logic 0 a 5 mA current is sent from sender to receiver and back in a loop of wire. The loop is part of the cable that

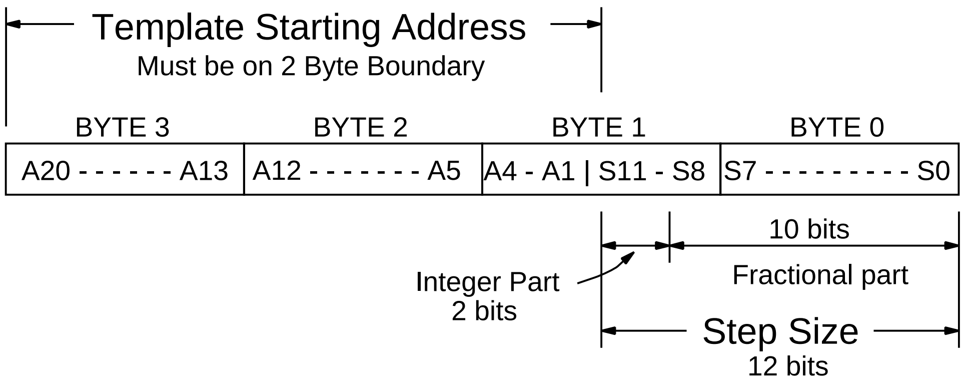

Figure 4: Directory Entry.

makes up the physical connection between the two devices. No current flowing in the loop corresponds to a logic 1. A MIDI connection is half duplex (one-way), asynchronous, and operates at 31.25±1% Kbaud. Each transmitted character is 10 bits long with a start bit, eight data bits (LSB first), and a stop bit.

The MIDI cable connects to a 5 pin DIN female receptacle. One of these has been mounted for you on your FPGA board. The serial signal is wired to an opto-isolator, which performs electrical isolation and translates the current levels on the MIDI cable to the 5V and 0V needed by the FPGA. Details of the electrical connection between the MIDI cable and the opto-isolator are shown in figure 6.

The signal then is received by the UART and processed as MIDI protocol. MIDI protocol is rich and fairly complete, but this project only requires the knowledge of a very small subset. All transmitted bytes are either a *status* byte or a *data* byte. MIDI messages consist of a status byte followed by one, two or possibly more data bytes. A status byte always has a MSB of 1, and a data byte always has a MSB of 0.

There are only two messages that the MIDI synthesizer needs to read. All other MIDI messages must be ignored. The pertinent ones are the *note-on* message and the *note-off* message. Each has two data bytes:

byte 1	byte 2	byte 3
Note On:Channel #	key #	velocity
`1001nnnn`	`0kkkkkkk`	`0vvvvvvv`

byte 1	byte 2	byte 3
Note Off:Channel #	key #	velocity
`1000nnnn`	`0kkkkkkk`	`0vvvvvvv`

Velocity is the way within MIDI to represent the strength or intensity of the note. In MIDI keyboards this value comes from a measurement of the speed at which a key is pressed or released. In MIDI sound synthesizers, velocity is usually used to control the loudness of the resulting note.

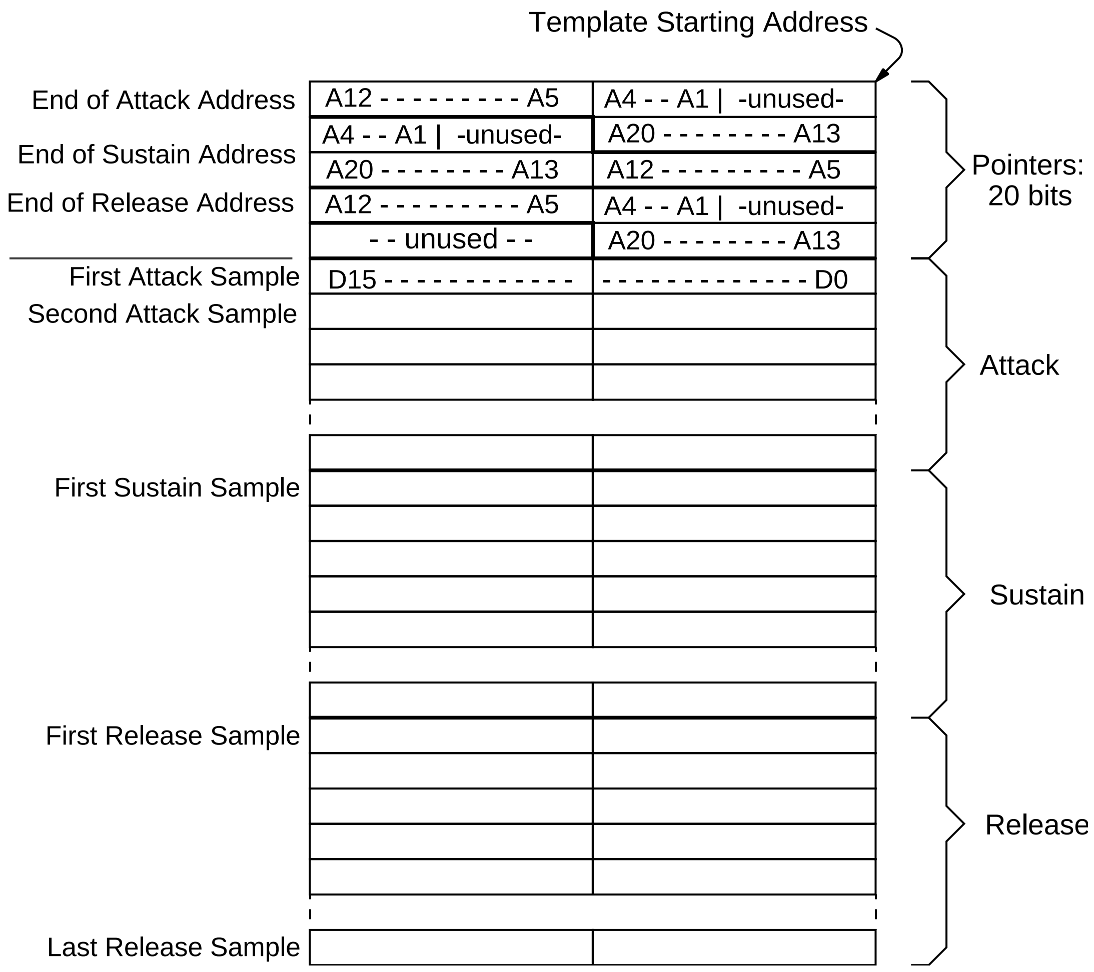

Figure 5: Template Layout.

The channel number allows the message to be directed to 16 different voices or machines. Our MIDI synthesizer will disregard it and take messages from all channels. Note-on causes the MIDI synthesizer to begin playing a note of a certain pitch as specified by the key number, at a particular loudness as specified by the velocity. Note-off causes the MIDI synthesizer to play the release sequence of the note as soon as it can, i.e. after it finishes playing the current iteration of the sustain. In typical commercial MIDI synthesizers, there can be many notes playing at once, hence the key field in the note-off message. Although our MIDI synthesizer can only play one note at a time, you must still respect the convention by only turning a note off if the key number matches that of the current note playing. Ignore the velocity field in the note-off message.

By convention, a note-on message with zero velocity has the same meaning as a note-off message. Therefore your synthesizer will have to recognize two different ways of sending note-off events, either as a note-off message or as a note-on message with velocity=0.

The MIDI standard also supports a convention called “running status.” This convention allows a transmitter to compress the data stream by dropping status bytes. A command without a status byte implicitly uses whatever status byte was most recently sent. Therefore a keyboard can send a sequence of note-on and note-off commands only the first

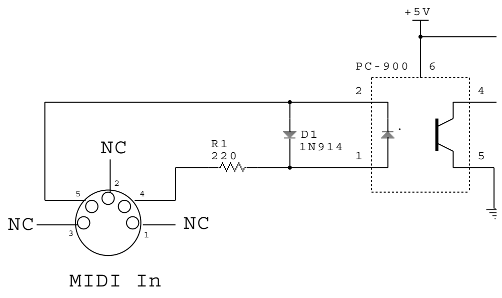

Figure 6: MIDI Physical Connection.

of which having a status byte. Your synthesizer must conform to the running status convention.

2 Loudness Control

We will use the velocity information from the MIDI command to control the *loudness* of the sound. We can do this by using the velocity to scale the amplitude of the waveform. To scale the amplitude of the waveform we simply scale each individual sample before sending it to the DAC.

The relationship between signal amplitude and the perceived loudness of a sound is complex. But fortunately, the ear is far less sensitive to amplitude changes than to other musical qualities of a sound, such as pitch. Because of this insensitivity, we can use several approximate methods for controlling the loudness of the sound without detrimental effects. While, to first order, loudness is proportional to signal amplitude, it is not a direct proportional. Specifically, in order to increase the loudness of a sound by a factor of 2 we need to increase its amplitude by a factor of 2.8. Likewise for decreasing the loudness of a sound; to half its loudness we must decrease its amplitude by a factor of 2.8. We can express this relationship with the following formula:

$$AMP = 2.8^{log_2 LOUDNESS}.$$

Therefore, if we had a way to compute the above function, and given a desired loudness, say the velocity value, we would then have a scaling factor for the amplitude. Then we could multiply the output samples by the desired amplitude value, AMP.

The stored templates in the EPROM files are all at maximum amplitude. Therefore all sample values on their way out to the DAC will be scaled by a number which is *at most* equal to 1.0.

Described below are two schemes for scaling the amplitude of the waveform that will give satisfactory results. Consider these schemes and implement one of them as part of your synthesizer.

Scheme 1 Approximate the amplitude function with a table lookup operation. The velocity information is a 7-bit number. We could use the 7 bits as an index into a table, of 128 amplitude scale numbers. A table of this size is probably overkill for this application. Ignoring some of the low bits of the velocity information will lead to a smaller table size, at the cost of lower accuracy. A table of 16 entries will give satisfactory results. The *width* of the table determines the accuracy of the approximation of the the above function. Once again, 4-bits is sufficient. In this scheme you will need a 16x4 ROM to store the table, and will need a multiplier circuit that takes a 4-bit multiplier (positive only) and a 16-bit multiplicand (2's complement). This multiplier circuit is slightly different from the one presented in class, because all the values stored in the table are less than 1. The binary point is assumed to be just to the left of the MSB. This modification is simple. Try a few cases by hand, and you will see.

Scheme 2 We can make a further approximation to the function by using a simple shifter instead of a multiplier circuit. In this case we will use a small memory in the FPGA to store a table of shift amounts. The shift is a *right shift* to decrease the amplitude.

Shown below is a set of table values (all 4-bit numbers) that you can use for the above schemes. If you don't like the results obtained, using the following numbers, you may change them as you like.

ROM addr	Scheme 1 (multiply)	Scheme 2 (shift)
0	1	6
1	1	4
2	1	3
3	2	3
4	3	2
5	3	2
6	4	1
7	5	1
8	6	1
9	7	1
10	9	0
11	10	0
12	11	0
13	12	0
14	14	0
15	15	0

3 Project Requirements Summary

By now you should have a pretty good idea of what is required of you for the final project. You will design and implement a synthesizer on your FPGA board that will accept MIDI note-on and note-off commands and generate an audio signal suitable for driving headphones. Your synthesizer must respond correctly to note-on and note-off commands, as follows: When a note-on command is received, begin a note with the frequency specified by the key number information and with an amplitude specified by the velocity information. The amplitude control need not be exact. It is acceptable to use a simple shift operation in place of multiplication. Until the note-off command is received (i.e., while the key on the MIDI keyboard is depressed) continue generating sound by looping the sustain portion of the note. Once a note-off command is received, if that key number matches that of the note currently being played, then finish the current iteration of the loop and play out the release section. If while playing a note, a note-on command is received, regardless of the key number, immediately discontinue playing the current note (without finishing the current loop iteration or playing out the release section) and begin the new note.

A particularly challenging part of the design is the method of linear interpolation used for computing output sample values. As discuss above, frequency shifting is performed by stepping though the note templates using a non-integer step size. In your design you must support a step-size with two integer bits and 10 fractional bits. This will allow a maximum frequency shift of just under 2 octaves and frequency shift accuracy to within a couple of cents. For non-integer step sizes you may use linear interpolation between adjacent sample values to generate a sample value. This method is illustrated in figure 7. The X values are used to represent indices into the wave template. X_s is the running index into the template. Each sample value is calculated by first fetching two sample values from the template, corresponding to adjacent values located at X_0 and X_1. The two sample values are used to calculate the local slope of the waveform and the desired sample value Y_s as shown in the figure.

The minimum requirement for the project is a version of the synthesizer without using interpolation. Once you have that working, and if you have the time, you can add interpolation for extra credit. For the non-interpolation version, you can simply output a stored sample value close to the desired sample values by truncating the running template index.

If you don't own headphones (most people have headphones from their portable compact disk players) we will have some that you can checkout. We will have a few MIDI keyboards and MIDI interfaces in the PCs for generating MIDI commands. These will be spread around in 204B and 123 Cory.

4 Project Components

You can get a good idea of the components needed to implement your synthesizer by examining figure 1. We will provide you with all of these and any other miscellaneous small components. If you decide to do something extra, you will have to buy your own

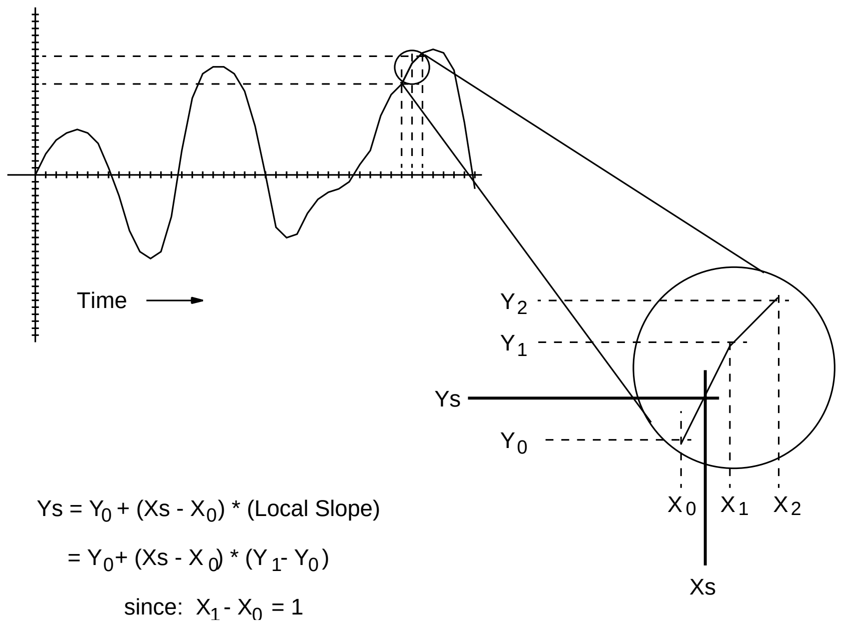
Time
Y2
Y1
Ys
Y0
X0
X1
X2
Xs
Ys = Y0 + (Xs - X0) * (Local Slope)
= Y0+ (Xs - X0) * (Y1- Y0)
since: X1 - X0 = 1

Figure 7: Linear Interpolation Method.

parts. Before you do that, however, check with us—we might have some spare parts in stock.

Refer to the online data sheets for detailed information on the major external components. Review this information early so that you can plan the board layout.

Following is a brief description of the major components:

FPGA All the logic associated with the synthesizer design must be implemented within the FPGA device (Xilinx 4010).

Opto-isolator This part provides electrical isolation from the MIDI input cable and converts the current loop to voltages for input to the FPGA. See figure 6 for details and additional components.

Digital-to-analog converter (DAC) The DAC is used to convert the 16-bit output from the sound generator into an analog waveform. The DAC should be operated with a sampling rate of 31.25KHz. The digital data input to the DAC is bit-serial; see the data sheet for details.

Audio Amplifier The analog output of the DAC does not have sufficient power to drive headphones. Also, the DAC is an expensive device and we which to protect its output from short circuits and other potentially hazardous conditions. Therefore we will use an Audio Amp as a interface between the analog output of the DAC and the headphone output on the board. Be careful with your headphones! Don't put them on until you are certain that your design doesn't have a bug that could hurt your ears.

EPROM A 4-Mbit EPROM, ST 27X040, will be used to store the directories and note templates.

5 Recommended High Level Organization

The internal organization of the logic within the FPGA is up to you, as long as you meet the functional requirements. However, you should consider the organization shown in figure 8.

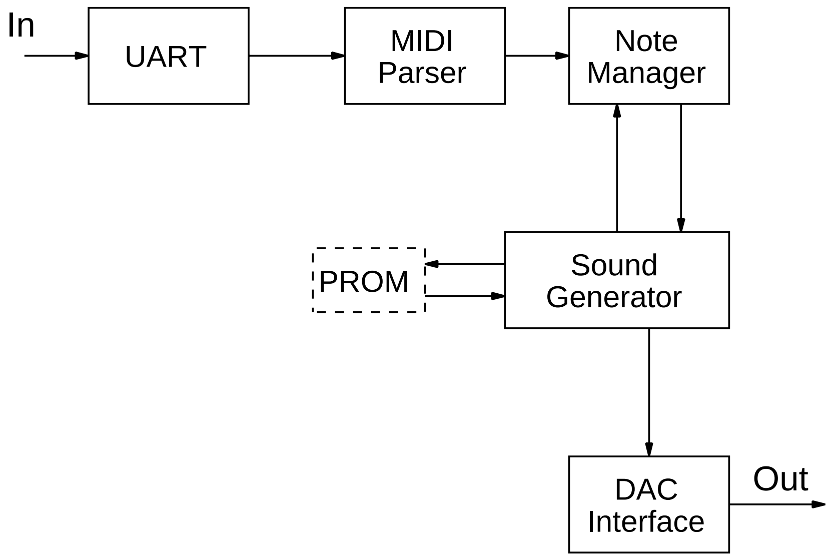

Figure 8: Synthesizer Block Diagram.

Each block in the block diagram has its own FSM controller and a datapath. The **UART** converts the serial bit stream from the MIDI input into parallel bytes. It signals the availability of a byte with a data ready signal. The UART runs with a clock frequency of 8X the MIDI clock rate. With the exception of the sample rate clock, 31.25KHz, the rest of the system should run using a 4 or 8MHz clock.

The **MIDI Parser** is used to "recognize" MIDI commands. Its output is a pair of signals, NOTE-ON and NOTE-OFF, and the contents of two registers, KEY and VELOCITY.

The **Note Manager** is used to control the actions taken when a MIDI key is pressed or released. It receives input from the MIDI Parser and a signal from the Sound Generator indicating the end of playing the release section of a note.

The heart of the synthesizer is the **Sound Generator**. This part does all the calculation needed to produce a new sample every 32us. It receives some control information from the Note Manager, interfaces directly with the EPROM, and produces samples sent to the DAC interface. The main control loop for this block is triggered once per sample period. It does the work to generate a new sample, passes it to the DAC interface, then waits for the next sample clock.

The DAC interface has a buffer to hold a sample ready to send to the DAC. The sample clock triggers the bit-serial transfer of its buffer contents to the DAC.

6 Checkpoints

This section describes a list of checkpoints. Each week you are required to demonstrate to your TA a part of your design. Part of your final project grade depends successful and timely completion of these checks. Of course, you are free (an encouraged) to work ahead. These checkpoints are the minimum amount that you need to have completed each week. By following our plan, you will complete your project with a week to spare at the end. This will give you time to relax, or fix-up some parts of your design, or do some extra credit work. Extra-credit will only be given if you first have a working design.

Project Checkpoints:

1. **3/11, Checkpoint 1: UART Design and Test.**
2. **3/18, Checkpoint 2: Wire-wrap and ROM Interfacing.**
3. **3/25, Recess.**
4. **4/1, Checkpoint 3: MIDI Interface.**

   Wire up opto-isolator, demonstrate working connection between MIDI controller and MIDI UART, demonstrate working MIDI command parser displaying MIDI information on 7-segment display.
5. **4/1, Checkpoint 4: Audio Stage.**

   Wire up the DAC and Audio Amp. Write test program to put out a ramp, using a counter as a waveform generator. Display the analog waveform on the scope and listen using headphones.
6. **4/8, Checkpoint 5: Monotone Notes.**
7. **4/15, Checkpoint 6: Notes of Arbitrary Frequency.**
8. **4/22, Checkpoint 7: Velocity Sensitivity.**
9. **4/29, Spare week.**

   Cleanup and optimize the design. If time, do extra credit work.
10. **5/6, Final Checkoff**

7 Extra Credit

A number of enhancements to the synthesizer for extra credit are possible. However, you will not be given any extra credit if you cannot demonstrate a working project. Your first priority is to satisfy the above design specification.

We may add extra credit ideas as the semester progresses.

Here is a short list for now:

- **Early Final Checkoff.** You can receive extra credit for checking off the complete project one or more weeks in advance.
- **Low CLB Count.** Some groups will qualify for extra credit by achieving low CLB count. We will quantify “low” later.
- **Interpolation.** Add the linear interpolation for sample lookup.
- **Polyphony.** Add the ability to play multiple keys at once.
- **Velocity Sensitive Template Lookup.** Add the ability to index templates not only on key number but also velocity.