

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA
DIGILENT CONTEST • FALL 2006

DigitalSynth

DIGITAL DESIGN REPORT

AUTHOR LUCIAN CHEȚAN
Technical University of Cluj-Napoca
Computer Science Department

ADVISOR PROF. DR. ENG. MIRCEA DĂBĂCAN
Technical University of Cluj-Napoca
Applied Electronics Department

Contents

| | |
|--------------------------------------|-----------|
| 1. Introduction | 4 |
| 1.1 Overview | 4 |
| 1.2 Summary of the design project | 4 |
| 1.3 Hardware requirements | 5 |
| 1.4 Software requirements | 5 |
| 2. Background | 6 |
| 2.1 Analysis of existing designs | 6 |
| 2.2 Digital sound synthesis | 6 |
| 2.2.1 Amplitude modulation synthesis | 8 |
| 2.3 Audio effects | 9 |
| 2.3.1 The circular buffer | 9 |
| 2.3.2 Delay | 10 |
| 2.3.3 Echo | 11 |
| 2.3.4 Vibrato | 11 |
| 2.3.5 Flange | 11 |
| 2.3.6 Reverb | 12 |
| 2.4 MIDI files | 12 |
| 2.4.1 The header chunk | 13 |
| 2.4.2 The track chunk | 13 |
| 2.4.3 MIDI events | 14 |
| 2.4.4 Meta events | 14 |
| 2.4.5 Example | 15 |
| 3. Hardware design | 17 |
| 3.1 Design overview | 17 |
| 3.2 The top level block diagram | 19 |
| 3.3 Components | 20 |
| 3.3.1 The top component | 20 |
| 3.3.2 KeyboardUnit | 21 |
| 3.3.2.1 KeyboardScanner | 21 |
| 3.3.2.2 KeyboardStatusGenerator | 23 |
| 3.3.2.3 PolyphonyArbitration | 24 |
| 3.3.3 UsbUnit | 26 |
| 3.3.4 NoteSelector | 30 |
| 3.3.5 OscillatorsUnit | 31 |
| 3.3.5.1 NoteMemory | 31 |
| 3.3.5.2 OscillatorGroup | 32 |
| 3.3.5.2.1 Oscillator | 33 |
| 3.3.5.2.2 Adder | 37 |
| 3.3.5.3 Sampler | 38 |

| | | |
|-----------|--|-----------|
| 3.3.6 | EffectsUnit | 38 |
| 3.3.6.1 | CircularBuffer | 39 |
| 3.3.6.2 | DelayEcho | 41 |
| 3.3.6.3 | VibratoFlanger | 41 |
| 3.3.6.4 | Reverb | 43 |
| 3.3.6.5 | Feedback | 44 |
| 3.3.6.6 | Mix | 44 |
| 3.3.7 | SpiUnit | 45 |
| 4. | Software design | 47 |
| 4.1 | Design overview | 47 |
| 4.2 | Classes | 47 |
| 4.3 | UML diagrams | 48 |
| 4.3.1 | Use case diagram | 48 |
| 4.3.2 | Sequence diagrams | 49 |
| 4.3.4 | Class diagram | 52 |
| 4.4 | USB communication | 52 |
| 4.4.1 | Decoding and interpreting MIDI files | 53 |
| 4.4.2 | Playing a MIDI file | 53 |
| 4.4.3 | Recording a MIDI file | 54 |
| 5. | Discussion | 55 |
| 5.1 | Design issues | 55 |
| 5.1.1 | Computing sine values | 55 |
| 5.1.2 | Generating basic waveforms with variable frequency | 55 |
| 5.1.3 | Creating audio filters | 56 |
| 5.1.4 | The quality of the sound | 56 |
| 5.1.5 | The music sheets | 56 |
| 5.2 | Design report | 57 |
| 6. | References | 58 |

1. Introduction

1.1 Overview

This report presents the design of a digital keyboard synthesizer. The upper two key rows of a PS/2 keyboard connected to the FPGA board represent the keys of a piano. Pressing a key generates the corresponding note. The digital instrument allows pressing at most four keys, thus allowing a four-note polyphony. The design uses the AM (amplitude modulation) synthesis method. The resulting signal is output to a digital-to-analog converter and then to a speaker.

With the help of a companion software application, the synthesizer is capable of applying audio effects (such as reverb, delay, echo, vibrato and flange) on the generated waves, and of playing and recording MIDI files. The software application also displays the music sheet of the MIDI files.

The hardware project has been developed using Xilinx ISE WebPACK 8.1i and the designs have been described in VHDL. Simulations on certain parts have been done using ModelSim XE III.

The software project has been developed using Microsoft Visual C# 2005 Express Edition. The MIDI files have been created using Geniesoft Overture 3, and audio comparisons and tests have been done using Adobe Audition 2.0.

1.2 Summary of the design project

The design is complete and meets its objectives. It has been verified through simulation and physical implementation.

The design has a modular approach and every component has a specific function. There are components that could be successfully used in other designs, although some modifications might be necessary.

Although not intended for commercial use, this design explores the idea of implementing DSP (*Digital Signal Processing*) functions on FPGA circuits and the possibility of creating FPGA-based digital instruments.

1.3 Hardware requirements

- Digilent Nexys FPGA Board
- Digilent PMOD DA1
- Digilent SPKR1
- Digilent PS/2 Module
- PS/2 keyboard
- USB cable

1.4 Software requirements

- Microsoft Windows XP
- Digilent Adept ExPort
- Microsoft .NET Framework 2.0

2. Background

2.1 Analysis of existing designs

Synthesizers were invented in the 19th century and developed intensively during the 20th century. The earliest analog synthesizers were created using vibrating electromagnetic circuits. Then followed synthesizers based on electronic circuits, but their complexity and cost weren't matched by their capabilities. In the '60s and the '70s they became very popular, and started to "invade" the music of that time.

The era of digital synthesizers started in the '80s. Ever since, they have never stopped evolving. Using the newest and increasingly cheaper microprocessors, memories and other integrated circuits, they became an interesting alternative to the bulky and expensive traditional instruments.

Nowadays, FPGA (*Field Programmable Gate Array*) technology is following the same trend of decreasing costs and increasing performance. Yet this technology is not used on a large scale in the digital instruments domain. ASICs (*Application-Specific Integrated Circuits*) are still preferred to FPGAs.

ASICs represent a very powerful architecture that may include processors, memory blocks (ROM, RAM, Flash) and other important elements. It integrates many functional blocks, so it can be called "system on a chip". The downside is the cost of production which is quite large. This renders ASICs suitable only for mass production. FPGA architecture can offer the same functionality at lower production costs.

Existing designs of synthesizers and digital instruments are built using powerful ASICs, large memory blocks, they support connection interfaces such as MIDI, USB and even Internet connectivity.

This project is a study of the FPGA alternative and tries to demonstrate that at least the most basic features of modern synthesizers can be implemented on an FPGA circuit.

2.2 Digital sound synthesis

Physically, the sound is the wave produced by the mechanical vibration of a solid or liquid medium. The human ear is sensitive to sounds with frequencies of 20 Hz to 20 kHz, with maximum sensitivity around 3500 Hz.

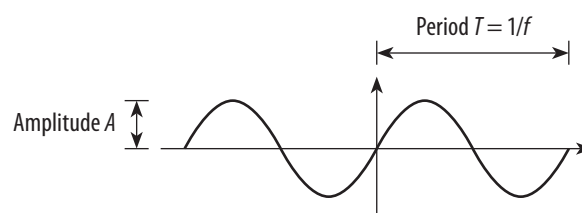


Figure 2.1. The sine waveform.

A sound can be represented as a mathematical function which can be written as an infinite sum of trigonometric functions (*sine* and *cosine*). This sum is called *Fourier series*, and its equation is:

$$f(x) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n \cos(nx) + \sum_{n=1}^{\infty} b_n \sin(nx)$$

According to this equation, that sound can be represented as a sum of elementary sounds such as sine waves, each with a different amplitude.

Some of the sounds that are used in sound synthesis are presented in the following table.

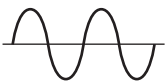
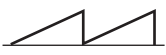

| Function | Waveform | Fourier series | Observations |
|----------|---|---|--|
| Sine |  | $f(x) = \sin(x)$ | The sound associated with the <i>sine</i> function has the fundamental frequency f . |
| Sawtooth |  | $f(x) = \frac{1}{2} - \frac{1}{\pi} \sum_{n=1}^{\infty} \frac{1}{n} \sin\left(\frac{2n\pi x}{T}\right)$ | The sound associated with the <i>sawtooth</i> function contains all multiples of the fundamental frequency $f + 2f + 3f + \dots$ ($2f, 3f, \dots$ are the <i>harmonics</i>) |
| Square |  | $f(x) = \frac{4}{\pi} \sum_{n=1,3,\dots}^{\infty} \frac{1}{n} \sin\left(\frac{2n\pi x}{T}\right)$ | The sound associated with the <i>square</i> function contains odd multiples of the fundamental frequency $f + 3f + 5f + \dots$ ($3f, 5f, \dots$ are the <i>harmonics</i>) |

Table 2.1. Sounds that are most commonly used in sound synthesis.

The sine waveform is both the purest and the “poorest” sound of the three. The “richest” sound is the sawtooth waveform, because it has more elementary components.

Some of the techniques of sound synthesis are:

- *additive synthesis* — creating complex sounds by combining elementary sounds (such as sine waves) with various amplitudes determined after a Fourier analysis;
- *subtractive synthesis* — creating complex sounds by filtering other complex sounds (such as sawtooth waves);
- *AM synthesis* — modulating a wave’s amplitude according to another wave’s amplitude;
- *FM synthesis* — modulating a wave’s frequency according to another wave’s amplitude;
- *sample-based synthesis* — creating complex sounds using sounds recorded from real instruments.

Sounds generated by real instruments can’t be easily replicated using synthesis techniques because of their complexity. For instance, the piano sound is defined by the following components: the fundamental frequency (basically, the note identified when hearing the sound), the harmonics (multiples of the fundamental frequency), other sounds triggered by the resonance of the harmonics, reverberations of the instrument, noises caused by moving elements such as pedals, keys etc. Rendering these components requires the use of a combination of the techniques described above.

In order to be able to reproduce realistic sounds, a good synthesizer would need many *oscillators*, units that generate the desired waveforms. This involves large computing power and complex programming. A general scheme of a synthesizer is presented in the next figure.

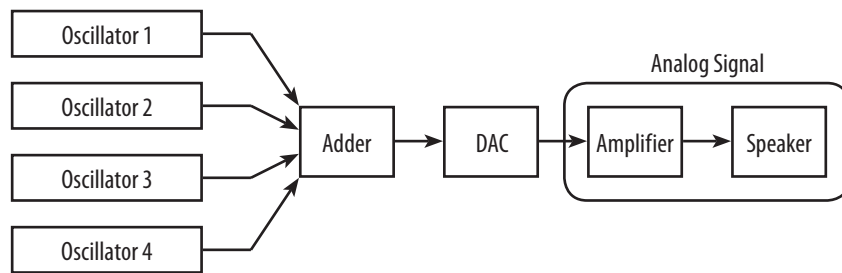


Figure 2.2. A basic digital synthesizer.

These waveforms can be added in order to form a single resulting sound, or used separately.

2.2.1 Amplitude modulation synthesis

AM synthesis is the process of modifying the amplitude of a signal (*the source*) according to the amplitude of another signal (*the modulator*). The resulting sound is obtained by multiplying these signals.

The modulation signal can be an envelope which, applied to the source signal, determines its amplitude over time. One of the most used envelope models is *ADSR* — *attack, decay, sustain, release*.

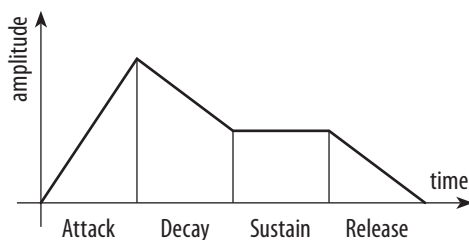


Figure 2.3. The ADSR amplitude envelope model.

During the attack phase, the amplitude of the sound is increased from zero to its maximum value. During decay, the amplitude is decreased from its maximum value to the value at which it will remain constant during the sustain phase. During release, the amplitude of the sound is decreased from the constant value to zero.

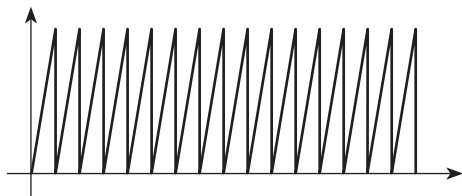


Figure 2.4. The sawtooth wave before enveloping.

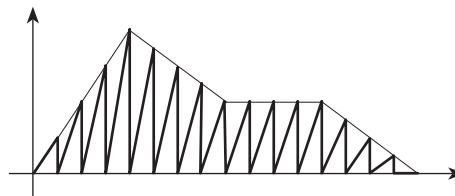


Figure 2.5. The sawtooth wave after enveloping.

ADSR alters the amplitude of the signal but not the frequency. This means that the volume of the sound changes while the pitch remains the same. Figures 2.4 and 2.5 show the sawtooth wave enveloped using the ADSR model. If the modulation signal were periodical, there would be some notable frequency changes (additional frequencies called *sidebands* would appear).

2.3 Audio effects

Audio effects are used to enhance the synthetic sounds generated by the oscillators for better reproducing the sounds of real instruments, or for creating new sounds.

The structure of an effect unit can be represented as a system.

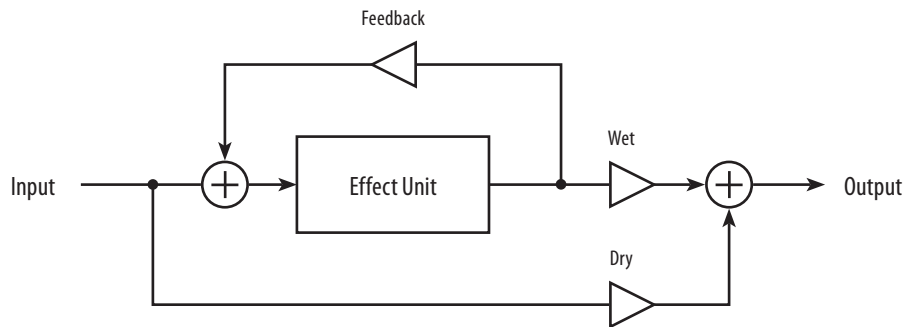


Figure 2.6. The system that describes an effect unit.

The original audio data is passed through the effect unit. The resulting data is fed back into the process, which gives more depth to the effect. The *wet* or processed data is mixed with the *dry* or original data to form the output of the system. The feedback, wet and dry coefficients represent adjustable values between 0 and 100%.

The reverb, delay, echo, vibrato and flange effects and the additional structures needed are explained in the following chapters.

2.3.1 The circular buffer

The effects mentioned above are all delay-based effects. This means that they use previous values of the audio data. For this purpose they use structures that store the current value and the previous values of the sound.

In the past, the most common analog “tool” used for delay-based effects was *tape*. The following figure presents the principle of a tape delay unit.

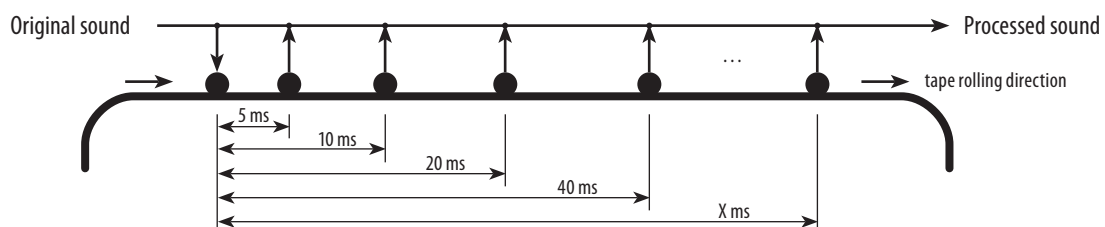


Figure 2.7. The tape delay unit.

The audio information is recorded on the tape and read at a number of different positions on the tape. The distance between the read heads and the speed of the tape give the delay between sound copies. The original sound and the delayed copies are mixed and the result is the processed sound.

In the digital era, delay effects use *delay lines* and *circular buffers*. A delay line is presented in the next figure.

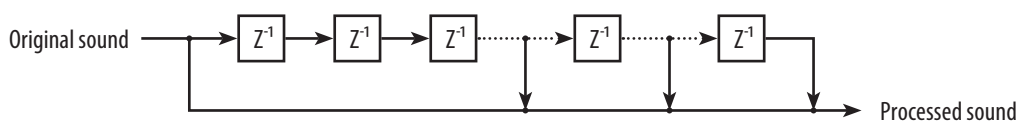


Figure 2.8. The delay line.

The delay line uses an array of registers. Audio data is stored in the first register and then is passed from one register to the next. Data is read from certain registers so that it corresponds to certain delays.

For example, at a sample rate of 48 kHz, the delay line would need 48,000 registers for a maximum delay of one second. Reading data delayed by 0.5 seconds requires reading the contents of the 24,000th register.

The disadvantage of delay lines is the large number of registers needed for larger delays. As a register structure is harder to access and as registers are more expensive than memories, a better option is to use a memory as a circular buffer. Data is written at a certain index that is incremented for every new sample. Reading is done at certain offsets in relation to the write location.

For example, at a sample rate of 48 kHz, a memory with 48,000 locations is needed for a maximum delay of one second. The first sample is written at index 0, the second sample is written at index 1 and the 48,000th sample is written at index 47,999. The 48,001st sample is written at index 0 and this is what gives the memory its circular appearance. Reading data delayed by 0.5 seconds is done at an address equal to the write address minus 24,000. The principle of a circular buffer is presented in the following figure.

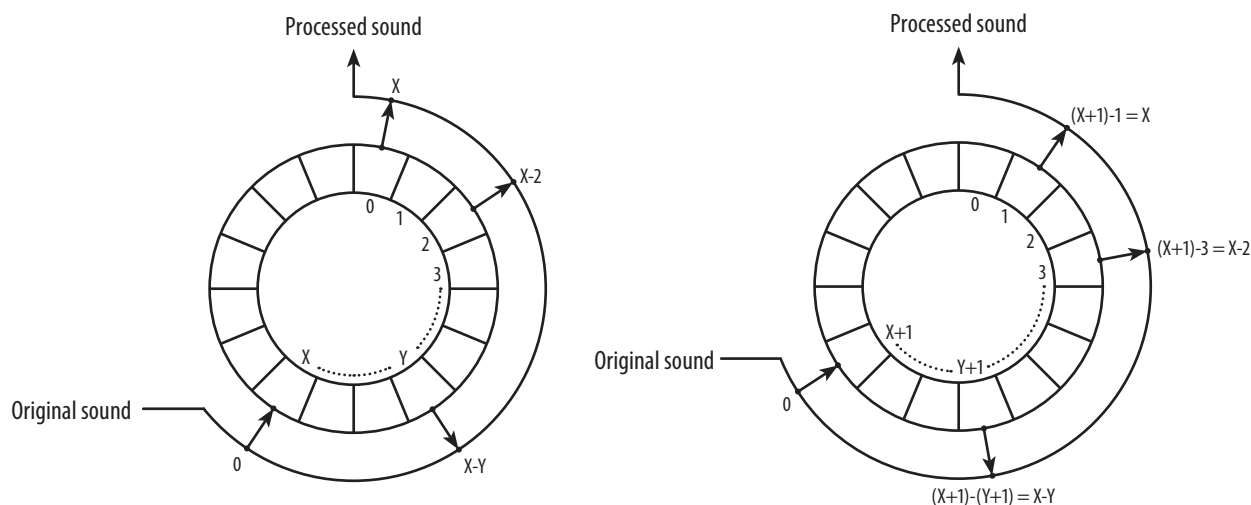


Figure 2.9. The circular buffer principle.

2.3.2 Delay

The delay is one of the simplest audio effects. It consists of playing the audio data after a specific period of time. Using a circular buffer, this involves reading data at a single offset in relation to the current write address. The delay time is an adjustable parameter.

2.3.3 Echo

The echo is the reflected sound that arrives to the listener after the direct sound. The listener perceives the echo as a distinct copy of the original sound.

The echo effect is based on the delay effect, the difference is that the processed data is fed back into the echo unit. Using a circular buffer, data is read at a single offset in relation to the current write address, and the delay time is an adjustable parameter, just like in the case of the delay effect.

Basically, the delay and echo effects could share the same hardware or software resources. In the case of the delay effect, the feedback coefficient is set to 0%, while in the case of the echo effect it has a value between 0 and 100%.

2.3.4 Vibrato

The vibrato effect consists of quick periodical changes in the pitch of a sound. The effect is naturally present in the human voice and is used in the case of some instruments for more expression. For instance, the violin player slightly changes the position of the finger on the chord, thus slightly changing the pitch of the played note.

A non-periodical pitch deviation occurs when a car or a plane passes by, which is due to the Doppler effect. This pitch variation is due to the fact that the distance between the car and the listener changes over time. Varying the distance is equivalent to varying the delay. Periodically varying the delay creates a periodical pitch variation which is the vibrato effect.

The delay oscillates between 0 and a given number of milliseconds. The shape of the oscillation can be sinusoidal, triangular etc.

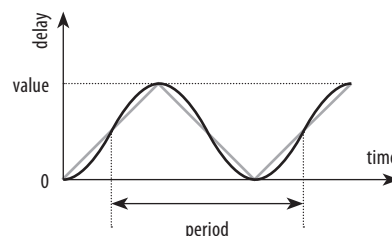


Figure 2.10. The shape of the delay parameter variation.

Using a circular buffer, data is written at the current address. Data is read at an offset that changes periodically so that it targets the desired delay. For instance, at a sample rate of 48 kHz and a maximum delay of 10 milliseconds, the offset of the read address varies periodically between 0 and 480.

The system of the process has the dry coefficient set to 0%, which means that only the processed data is played. The feedback coefficient is also set to 0%.

2.3.5 Flange

The flange effect is based on the vibrato effect, which means that they too can share the same resources. The difference between them consists in the fact that in the case of the flange effect the dry coefficient is

larger than 0%, which means that the original data is mixed with the processed one. The value of the feedback coefficient can be anywhere between 0 and 100%.

2.3.6 Reverb

The reverb effect is the result of the reflections of a sound that occur in an enclosed space.

The sound is emitted by a source and what the listener hears are the original sound and its reflections off the walls.

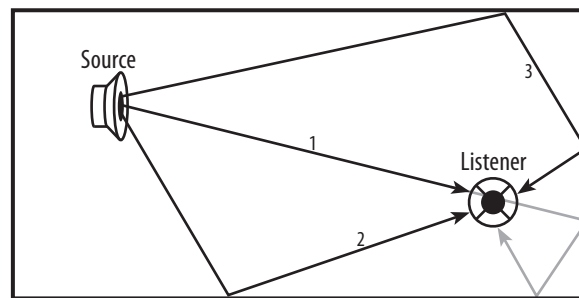


Figure 2.11. Reverberations inside of an enclosed space.

The first wave is the direct wave which arrives in the shortest time. The second and the third are reflected waves and they arrive in a longer time. In reality, there is an infinite number of reflected waves because each reflected wave generates other ones and so on. Because the delays between all these waves are relatively small, the listener can't hear individual sounds. At the moment of reflection, the intensity of the reflected wave decreases because of the properties of the wall and, eventually, it becomes equal to zero.

The reverb effect used to be realized using a circular buffer. Data is written at the current address. Data is read at constant offsets in relation to the write address. For a more realistic reverb, reading should be done at as many addresses as possible. Also, these values can be multiplied with the coefficients that define the amount of decay introduced by the reflective surfaces (walls, floor, ceiling, other objects).

The system of the process can have a feedback value between 0 and 100%.

2.4 MIDI files

MIDI or *Musical Instrument Digital Interface* is an industry-standard electronic communication protocol. Standard MIDI files are used by musical software and hardware devices to store song information (title, track names, instruments) and musical events (notes, instrument control).

MIDI files are made up of one header chunk and several track chunks. Each chunk starts with four characters that define the type of the chunk and a 32-bit word that defines the size of the chunk.

There are three types of events: *MIDI events* (related to notes and instrument control), *meta events* (related to other settings in the file) and *SysEx events* (related to the MIDI hardware and software control).

2.4.1 The header chunk

The *header chunk* is located at the beginning of the file and has the following structure:

| Element | Size | Values | Description |
|--------------------|----------------------|------------|--|
| <chunk ID> | 4 characters (bytes) | "MThd" | The ASCII correspondent of the "chunk ID" is "MThd". |
| <chunk size> | 4 bytes | 6 | The size of the rest of the header chunk is always 6 bytes. |
| <file format> | 2 bytes | 0 – 2 | A format 0 MIDI file has only one track that contains all of the musical events of the song. A format 1 MIDI file has two or more synchronous tracks. The first one contains song information such as the title, the time signature etc. and the next tracks contain note events and instrument control. A format 2 MIDI file contains multiple asynchronous tracks (they can start at different times). |
| <number of tracks> | 2 bytes | 1 – 65,535 | The number of track chunks. |
| <time division> | 2 bytes | | There can be two representations of time: ticks per beat (the number of clock ticks that a quarter note lasts for) or samples per frame (the number of samples in the number of frames in a second). |

2.4.2 The track chunk

The *track chunk* has the following structure:

| Element | Size | Values | Description |
|--------------|----------------------|--------|--|
| <chunk ID> | 4 characters (bytes) | "MTrk" | The ASCII correspondent of the "chunk ID" is "MTrk". |
| <chunk size> | 4 bytes | N | The size of the rest of the track chunk. |
| <data> | N bytes | — | Track event data (notes, instruments etc.). |

Events describe the musical content of the file, including the title, the tempo, the time signature, every note etc. An event is formed of the delta-time (the time between the previous event and the current event), the event type and the event parameters.

Delta-time represents the time that passes between the previous event and the current event. Time is represented according to the setting in the header chunk. Delta-times are represented as variable-length values that use the lower seven bits of a byte for data and the eighth bit to signal a following byte. If the eighth bit is '1', then another byte follows. Otherwise, that is the last byte. The following example illustrates the concept.

| Value | | Variable-length value | |
|--------|-----------------------|-----------------------|---|
| 64h | 0110 0100 | 64h | 0 110 0100 |
| 128h | 1 0010 1000 | 8128h | 1 000 0010 0 010 1000 |
| 12345h | 1 0010 0011 0100 0101 | 81A345h | 1 000 0100 1 000 0110 1 100 0110 0 100 0101 |

Two events separated by a delta-time equal to 0 are simultaneous.

2.4.3 MIDI events

MIDI events are composed of a delta-time and a number of bytes (two or three) that define the event type, the MIDI channel it affects and the parameters.

| Delta-time | Event type | MIDI channel | Parameter 1 | Parameter 2 |
|-----------------------|------------|--------------|-------------|-------------|
| variable-length value | 4 bits | 4 bits | 1 byte | 1 byte |

Some of the most common MIDI events are presented in the following table.

| MIDI event type | Parameter 1 | Parameter 2 | Description |
|----------------------|------------------------------|-----------------------------|--|
| Note Off 8h | note number 0 – 127 | velocity 0 – 127 | This event is used to signal when one of the 128 MIDI note stops. |
| Note On 9h | note number 0 – 127 | velocity 0 – 127 | This event is used to signal when one of the 128 MIDI note starts. The second parameter determines the volume of the note. |
| Controller Bh | controller number 0 – 127 | controller value 0 – 127 | This event controls different attributes of a MIDI channel, such as volume, pan, modulation and other effects. |
| Program change Ch | program number 0 – 127 | not used | This event controls the instrument used on a MIDI channel. For instance, program number 0 corresponds to the grand piano and 40 to the violin. |

Table 2.2. MIDI events.

2.4.4 Meta events

Meta events are composed of a delta-time, a byte that always has the value FFh and a number of bytes that define the event type and the parameters.

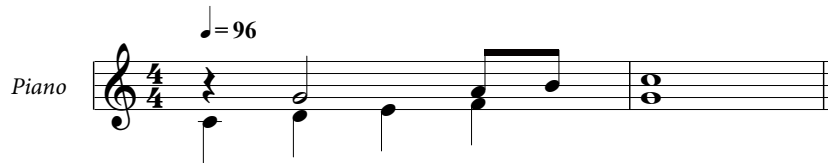
Some of the most common meta events are presented in the following table.

| Meta event type | Parameters | | | | | Description |
|----------------------------|--|--|--|--|--|---|
| Track name FFh 03h | NN the length of the text | TT ... the ASCII text | | | | This event defines the name of a track. |
| Instrument name FFh 04h | NN the length of the text | TT ... the ASCII text | | | | This event defines the name of the instrument. |
| End of track FFh 2Fh | NN the number of bytes to follow (always 00h) | | | | | This event marks the end of a track. |
| Tempo FFh 51h | NN the number of bytes to follow (always 03h) | TT TT TT microseconds per quarter note | | | | This events sets the tempo of the song. |
| Time signature FFh 58h | NN the number of bytes to follow (always 04h) | NM numerator 0 – 255 | DN denominator 0 – 255 (0 = whole note 1 = half note, 2 = quarter etc.) | CC the number of metronome ticks in a quarter note | TH the number of 32nd notes in a quarter note (always 8) | This events sets the time signature of the track. |

Table 2.3. Meta events.

2.4.5 Example

Music



The music sheet above has a tempo value of 96, no sharps or flats and two four-quarter measures. The structure of the corresponding format 1 MIDI file is presented in the following figures.

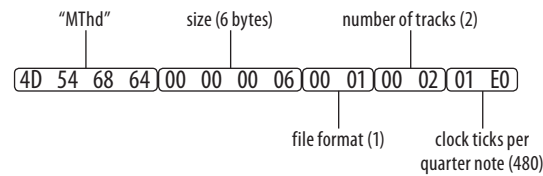


Figure 2.12. The header chunk.

A quarter note lasts for 480 clock ticks. A full note (and a measure, in this case) lasts for $480 \times 4 = 1920$ clock ticks. An eighth note lasts for $480 / 2 = 240$ clock ticks.

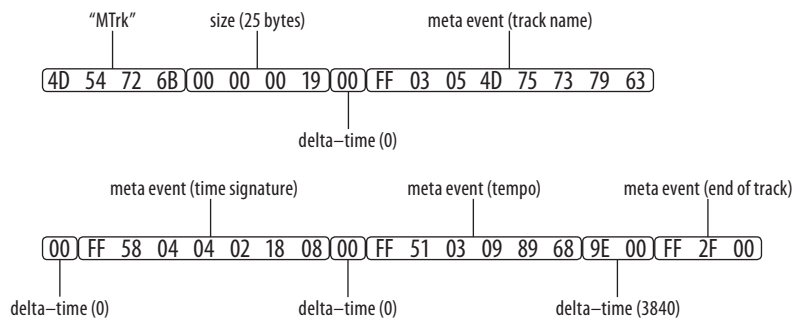


Figure 2.13. The first track chunk.

The track name meta event (FFh 03h) defines a string of 5 characters (“Music”). The time signature meta event (FFh 58h) sets the numerator to 4, the denominator to 2 (quarters), the number of metronome clicks in a quarter note to 24. The number of 32nd notes in a quarter note is 8. The tempo meta event (FFh 51h) sets the number of microseconds for a quarter note to 625,000. A minute has 60,000,000 microseconds, which means that the tempo is equal to $60,000,000 / 625,000 = 96$ beats per minute (BPM). The end of track meta event (FFh 2Fh) is at the end of the track, and it has a delta-time value of $3840 = 1920 \times 2$, which means that the song has two measures.

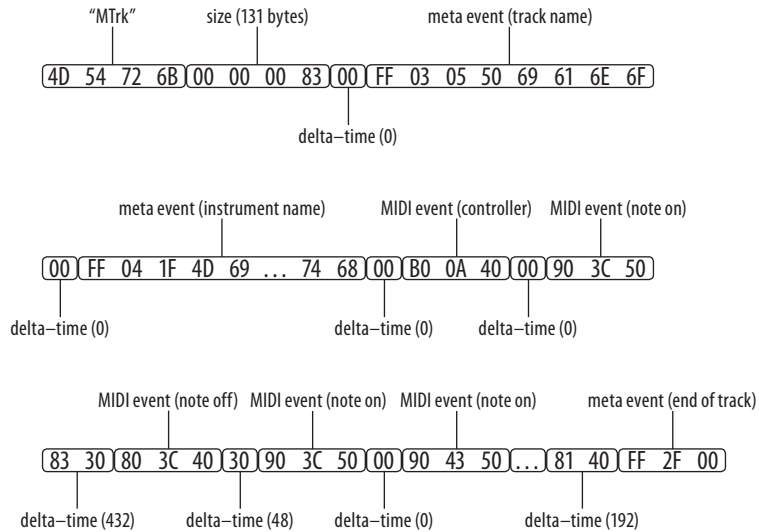


Figure 2.13. The second track chunk.

The track name meta event (FFh 03h) defines a string of 5 characters (“Piano”). The instrument name meta event (FFh 04h) defines a string of 31 characters (“Microsoft GS Wavetable SW Synth”). The first MIDI event sets the pan of the song to 64, which means that notes will be heard equally on both audio channels of the stereo output. The following MIDI events are related to notes that start (90h) and end (80h) at different moments in time. The last event is the end of track meta event (FFh 2Fh).

The program used to generate the MIDI file according to the sheet above sets the length of any note to 90% of its theoretical size. A quarter note should last for 480 clock ticks, as mentioned in the header chunk. In the second track, the first note, a quarter note, starts at moment 0 and ends after 432 clock ticks, which represents 90% of 480. The second note starts 48 clock ticks after the end of the previous note, which represents 10% of 480. The end of the track appears 192 clock ticks after the end of a full note, which represents 10% of the length of the full note ($480 \times 4 = 1920$).

3. Hardware design

3.1 Design overview

The design uses the PS/2 keyboard as a piano keyboard. Although there are some limitations regarding the number of keys that can be simultaneously pressed, four keys are, in most cases, supported.

The keys used for playing are found on the upper two rows of the keyboard. They correspond to the white and black keys of a piano. The design is capable of playing 31 notes that cover almost 2 ½ octaves (F3 to B5), but only 17 notes are accessible to the keyboard. The lowest note is C4 and the highest note is E5. The keys and their corresponding notes are presented in the following table and figure.

| | | | | | | | | | | | | | | | | | | | | | | | | | |
|---------------------|----------------------|---------------------|----------------------|---------------------|----------------------|----------------------|---------------------|----------------------|---------------------|----------------------|---------------------|---------------------|----------------------|---------------------|----------------------|---------------------|---------------------|----------------------|---------------------|----------------------|---------------------|----------------------|---------------------|--|--|
| F3 175 Hz | F#3 185 Hz | G3 196 Hz | G#3 208 Hz | A3 220 Hz | A#3 233 Hz | B3 245 Hz | | | | | | | | | | | | | | | | | | | |
| USB only | | | | | | | | | | | | | | | | | | | | | | | | | |
| C4 262 Hz | C#4 277 Hz | D4 294 Hz | D#4 311 Hz | E4 330 Hz | F4 349 Hz | F#4 370 Hz | G4 392 Hz | G#4 415 Hz | A4 440 Hz | A#4 466 Hz | B4 494 Hz | C5 523 Hz | C#5 554 Hz | D5 587 Hz | D#5 622 Hz | E5 659 Hz | | | | | | | | | |
| Q | 2 | W | 3 | E | R | 5 | T | 6 | Y | 7 | U | I | 9 | 0 | 0 | P | | | | | | | | | |
| | | | | | | | | | | | | | | | | | F5 698 Hz | F#5 740 Hz | G5 784 Hz | G#5 831 Hz | A5 880 Hz | A#5 932 Hz | B5 988 Hz | | |
| | | | | | | | | | | | | | | | | | USB only | | | | | | | | |

Table 3.1. The 31 notes.

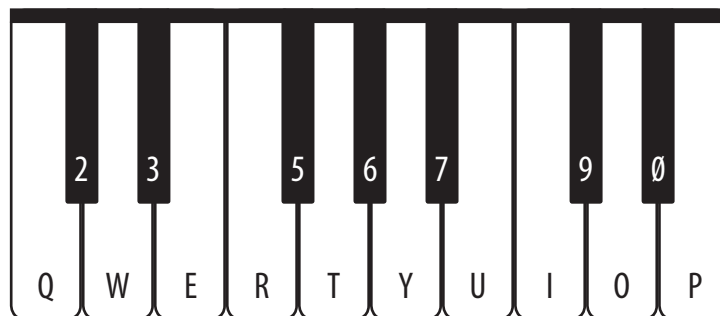


Figure 3.1. The key layout.

Pressing one of the keys generates a wave. Pressing more than one key generates an equal number of individual waves. The design supports a four-note polyphony, that is a maximum number of four keys can be simultaneously pressed. The keys can be pressed at any time, and can be released at any time, one independently of the other. The four-note polyphony uses four wave generators. Because there are 17 keys and only 4 generators, a polyphony arbitration rule is needed.

The concept of polyphony arbitration stands for the need of control over a large number of keys and a limited number of polyphony resources. The arbitration model implemented in this design allows any number of notes to be played as long as there are no more than four keys pressed at a time. If there are four keys pressed and then a fifth key is pressed, the fifth note is ignored. If there are four keys pressed and then one of them is released, the recently freed slot can be assigned to eventual future notes. The polyphony arbitration unit controls the status of the four polyphony slots according to the status of the keys (pressed or released). Each polyphony slot is linked to a wave generator or oscillator.

There are four oscillators that generate sawtooth or square waves, according to the note periods. Each of the waves is enveloped using the ADSR envelope model. The resulting amplitude modulated waves are added together to form the output signal which is then resampled to a specific sample rate.

The oscillators generate sounds at different rates. They generate a constant number of 256 samples per period. The lowest note, F3, with the frequency 175 Hz requires $256 \times 175 = 44,800$ samples per second. This corresponds to a 44.8 kHz sample rate. The highest note, B5, has the frequency 988 Hz and it requires $256 \times 988 = 252,928$ samples per second, equivalent to a 253 kHz sample rate. According to the sampling theorem, sounds with sample rate s reproduce correctly frequencies between 0 and $s/2$. The design reduces the various sample rates to a constant 48 kHz which can handle correctly frequencies between 0 and 24 kHz, thus covering the human hearing range. The resampling process slightly reduces the quality of the sound, as frequencies above 24 kHz are not properly handled.

The sampled sound is sent to the audio effects unit where it can be enhanced with effects such as reverb, echo, flange etc. or it can pass unmodified. The effects that the design supports are called “delay effects” because they process samples generated at various moments in the past. They all use a buffer that stores the newly generated samples and that allows access to samples generated in the past. Different effects read samples generated at different times. For example, the reverb effect reads samples that are 0, 5, 10, 20, 30, 45, 60, 75 and 100 milliseconds old. The vibrato effect reads samples generated between 0 and 5 milliseconds ago.

The digital-to-analog converter works by processing a serialized 16-bit frame, composed of the 8-bit wide sample data and the control byte that instructs the converter how to function. After the 16-bit frame a synchronization impulse is then sent, which instructs the converter to process the data it has received. The converter transforms the 8-bit digital information into analog information (voltage values). For the 256 possible digital values there are 256 possible voltage values, ranging from 0 to 3.3 volts.

The analog information is sent to the speaker.

The design uses the USB interface supported by the FPGA board to communicate with the computer. Certain aspects of the design can be changed via USB, such as the status and the note values of the oscillators, and the effect parameters. Also, the computer can receive information generated by the board such as the notes that the oscillators are currently playing.

The USB connection allows customizing the effects, playing decoded MIDI files on the FPGA board and creating MIDI files based on the information that the board generates. The MIDI events that are sent or received are the most basic ones: “note on” and “note off”. This involves the control of the status of the oscillators (note on – start playing the note, note off – stop playing the note) and of the note value they are using. Because the design doesn’t follow all the MIDI standard specifications and because it needs a companion software application, it presents an adapted implementation of the MIDI standard.

The form of the MIDI messages is adapted to the EPP communication protocol of the onboard USB controller. A byte of information contains all the necessary elements such as the message type and the note involved. The standard implementation would require also the volume of the affected note, but the design does not support this feature. In the rest of this document the “MIDI message” concept will be used with the above stated meaning.

3.2 The top level block diagram

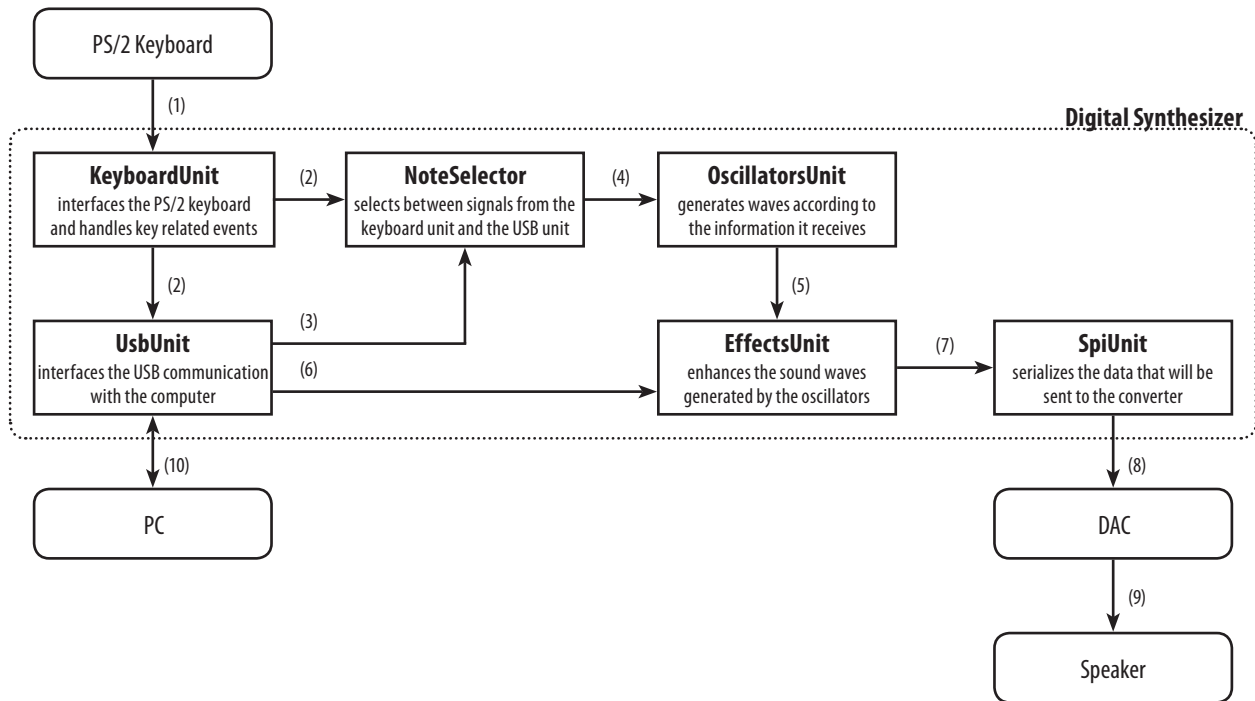


Figure 3.2. The top level block diagram.

The description of the signals that link the components are presented in the following table.

| Number | Description |
|--------|--|
| 1 | The <i>PS/2 keyboard</i> sends the scan codes of the keys that are pressed or released. This information is processed in the <i>KeyboardUnit</i> . |
| 2 | The <i>KeyboardUnit</i> generates the status and the note values of the four polyphony slots. This information is sent to the <i>NoteSelector</i> (which decides if it passes it to the oscillators) and to the <i>UsbUnit</i> (which sends it via USB to the software application). |
| 3 | The <i>UsbUnit</i> receives from the <i>PC</i> note information that is sent to the <i>NoteSelector</i> . |
| 4 | The <i>NoteSelector</i> sends the status and the note values of the polyphony slot to the oscillators. The selection is done using a signal coming from the <i>UsbUnit</i> . When the user sends a MIDI file to be played in the FPGA board, the source for the oscillators is the <i>UsbUnit</i> . When the user plays on the keyboard, the source for the oscillators is the <i>KeyboardUnit</i> . |
| 5 | The waves generated by the <i>OscillatorsUnit</i> are sent to the <i>EffectsUnit</i> . |
| 6 | The <i>UsbUnit</i> receives from the <i>PC</i> the parameters that customize the <i>EffectsUnit</i> . |
| 7 | The enhanced waves are sent to the <i>SpiUnit</i> for serialization. |
| 8 | The serial information sent by the <i>SpiUnit</i> is output to the <i>DAC</i> (digital-to-analog converter). |
| 9 | The analog signal from the converter is sent to the <i>speaker</i> . |
| 10 | The <i>PC</i> sends and receives from the <i>UsbUnit</i> information such as effect parameters, notes to be played on the board and notes to be recorded on the <i>PC</i> . |

Table 3.2. Description of the top level block signals.

3.3 Components

The design has six major components: *KeyboardUnit*, *UsbUnit*, *NoteSelector*, *OscillatorsUnit*, *EffectsUnit* and *SpiUnit*, and the top component that interconnects them.

3.3.1 The top component

The purpose of this component is to interconnect all the major components.

The component schematics and signals are presented in the following figure and table.

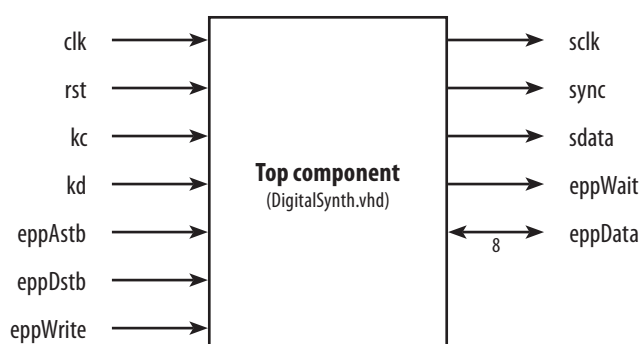


Figure 3.3. The top component.

| Name | Width | Direction | Description |
|----------|-------|---------------|---|
| clk | 1 | Input | Master clock (50 MHz) |
| rst | 1 | Input | Reset button (used for synchronous reset) |
| kc | 1 | Input | Keyboard clock signal |
| kd | 1 | Input | Keyboard data signal |
| eppAstb | 1 | Input | EPP protocol address strobe |
| eppDstb | 1 | Input | EPP protocol data strobe |
| eppWrite | 1 | Input | EPP protocol write strobe |
| sclk | 1 | Output | DAC clock (25 MHz) |
| sync | 1 | Output | DAC synchronization signal |
| sdata | 1 | Output | DAC serial data |
| eppWait | 1 | Output | EPP protocol wait strobe |
| eppData | 8 | Bidirectional | EPP protocol data bus |

Table 3.3. Signal description of the top component.

3.3.2 KeyboardUnit

This component groups the following submodules: *KeyboardScanner*, *KeyboardStatusGenerator* and *PolyphonyArbitration*, which read the scan codes sent by the PS/2 keyboard, determine the status of the keys and control the four polyphony slots.

The component schematics, structure and signals are presented in the following figures and table.

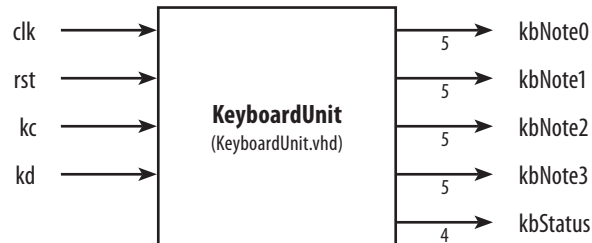


Figure 3.4. The “KeyboardUnit” component.

| Name | Width | Direction | Description |
|----------|-------|-----------|---|
| clk | 1 | Input | Master clock (50 MHz) |
| rst | 1 | Input | Reset button (used for synchronous reset) |
| kc | 1 | Input | Keyboard clock signal |
| kd | 1 | Input | Keyboard data signal |
| kbNote0 | 5 | Output | The note of the first polyphony slot |
| kbNote1 | 5 | Output | The note of the second polyphony slot |
| kbNote2 | 5 | Output | The note of the third polyphony slot |
| kbNote3 | 5 | Output | The note of the fourth polyphony slot |
| kbStatus | 4 | Output | The status of the four polyphony slots (identical to the status of the four keys they handle) |

Table 3.4. Signal description of the “KeyboardUnit” component.

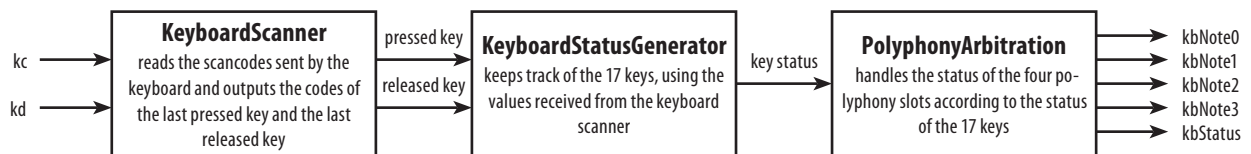


Figure 3.5. The internal structure of the “KeyboardUnit” component.

3.3.2.1 KeyboardScanner

The “KeyboardScanner” component reads the scan codes sent by the keyboard and stores in two registers the scan code of the most recently pressed key and the scan code of the most recently released key.

The component schematics and signals are presented in the following figure and table.

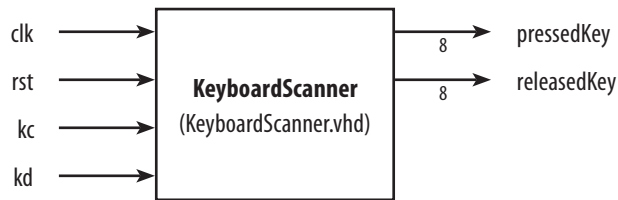


Figure 3.6. The “KeyboardScanner” component.

| Name | Width | Direction | Description |
|-------------|-------|-----------|--|
| clk | 1 | Input | Master clock (50 MHz) |
| rst | 1 | Input | Reset button (used for synchronous reset) |
| kc | 1 | Input | Keyboard clock signal |
| kd | 1 | Input | Keyboard data signal |
| pressedKey | 8 | Output | The code of the most recently pressed key |
| releasedKey | 8 | Output | The code of the most recently released key |

Table 3.5. Signal description of the “KeyboardScanner” component.

The module uses a 25 MHz clock which is obtained by dividing the frequency of the master clock. On the “kc” line the modules receives the clock generated by the PS/2 keyboard. This clock needs filtering because there may be a bad physical connection between the PS/2 connector and the board which can cause incorrect data reading. The filtering is realized by reading the values of the “kc” signal. When the signal becomes stable (it has a constant value), then that value is considered.

In idle mode (when no keys are pressed), the keyboard outputs high logic on both lines. Before sending a scan code the keyboard outputs ‘0’ on the data line and ‘1’ on the clock line, and this is what triggers the reading process. The next figure presents the “kc” and “kd” values while a keyboard is sending a scan code.

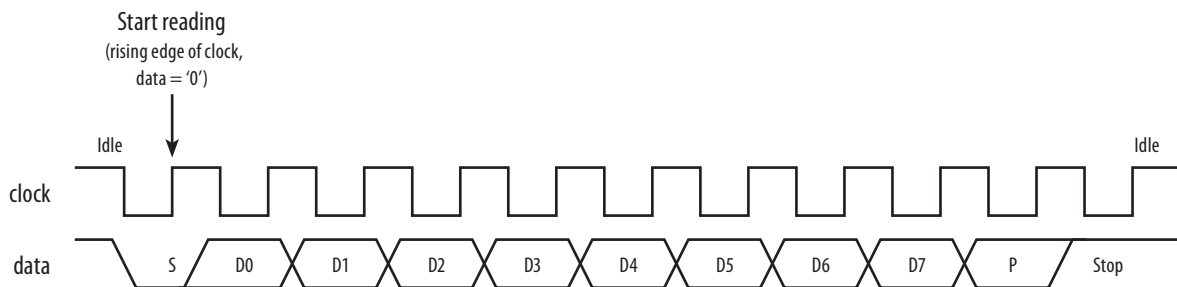


Figure 3.7. Keyboard operation.

The rising edge of “kc” when “kd” is ‘0’ triggers the start of reading. On every rising edge the value read on the “kd” signal is stored into a shift register. After eight readings the scan code is complete. The parity bit and the stop bit are ignored by this design.

The keyboard is permanently scanning the key matrix. When a key is pressed a *make code* is generated. Most keyboards generate a *repeat code* while a key is being pressed. Releasing a key generates a *break code*. The make code requires one read sequence. The break code requires two read sequences, because it’s made of two bytes: the first one is F0h and the second one is the code of that key, identical to the make code. Using the Alt, Shift, Ctrl and other special keys require sending more bytes.

Because the keys can be pressed and released at any time, the keyboard generates many codes that make harder to determine the status of the 17 keys. Working only with make and break codes is easier.

For instance, when pressing the Q key, then the W key, then releasing the Q key and then the W key, the keyboard issues make, repeat and break codes in a certain sequence displayed in the following figure. De-

pending on their type (make or break), these codes are stored in the two registers that help determining the status of the 17 keys.

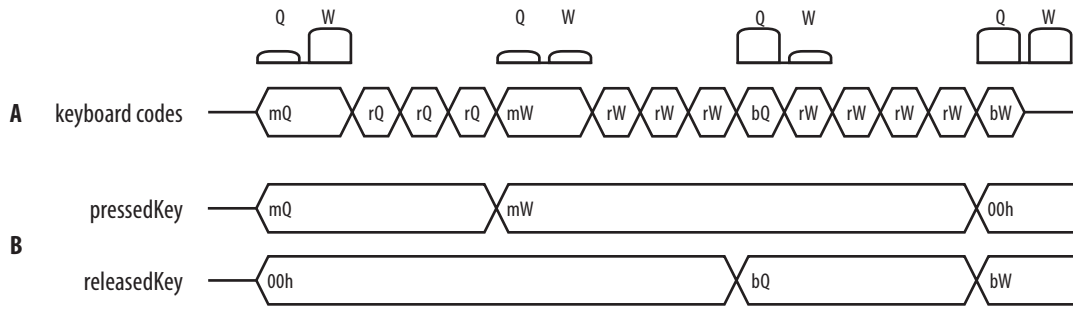


Figure 3.8. (A) The scan codes generated by the keyboard (m = make, r = repeat, b = break). (B) The contents of the registers that help determining the status of the 17 keys.

The design uses a state machine with two states for writing in the two registers.

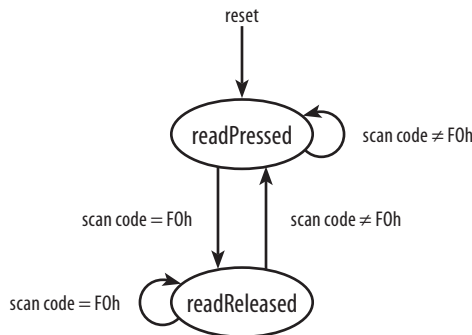


Figure 3.9. The state machine used for writing in the “pressedKey” and “releasedKey” registers.

| State | Description |
|---------------|---|
| read pressed | The state machine stays in this state as long as the keyboard doesn't generate the first byte of the break code. This means that the keyboard issues either make code or repeat codes of the last pressed key. Those bytes are identical and are stored in the “pressedKey” register. If the keyboard sends the F0h byte, then the state machine goes into the next state, because it expects the rest of a break code. |
| read released | The state machine stays in this state until the keyboard sends the rest of the break code, which will be stores in the “releasedKey” register. After that, the state machine goes into the first state, waiting for another make or repeat code. |

Table 3.6. State machine description.

The contents of the two registers (“pressedKey” and “releasedKey”) can't be identical because a key can't be pressed and released at the same time. If a key is pressed and then released the content of the “pressedKey” register is cleared. If the same key is pressed again, the content of the “releasedKey” register is cleared.

3.3.2.2 KeyboardStatusGenerator

The “KeyboardStatusGenerator” component keeps track of the 17 keys using a 17-bit register. Each bit corresponds to one key. If one bit has the value is ‘1’ then the corresponding key is pressed. Otherwise, it is not pressed.

For instance, when there are no keys pressed the “keyStatus” register contains only zeros “0...00”. If the Q key is pressed, the value of the register is “0...01”; if the W key is pressed afterwards, the value of the

register is “0...11”. If Q is released, the value of the register is “0...10”. If W is released too, the value of the “keyStatus” register is “0...00”.

The component schematics and signals are presented in the following figure and table.

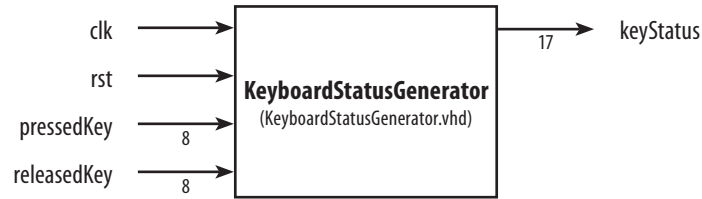


Figure 3.10. The “KeyboardStatusGenerator” component.

| Name | Width | Direction | Description |
|-------------|-------|-----------|---|
| clk | 1 | Input | Master clock (50 MHz) |
| rst | 1 | Input | Reset button (used for synchronous reset) |
| pressedKey | 8 | Input | The code of the most recently pressed key |
| releasedKey | 8 | Input | The code of the most recently released key |
| keyStatus | 17 | Output | The status of the keys ('0' = not pressed, '1' = pressed) |

Table 3.7. Signal description of the “KeyboardStatusGenerator” component.

The contents of “pressedKey” and “releasedKey” registers are compared with the codes of the 17 keys. If there is a match between “pressedKey” and one of the keys, then the corresponding bit is set to ‘1’. If there is a match between “releasedKey” and one of the keys, then the corresponding bit is set to ‘0’.

The “keyStatus” register is used for determining the notes that the oscillators will be playing, according to the polyphony arbitration rule.

3.3.2.3 PolyphonyArbitration

The “PolyphonyArbitration” component manages the four slots of polyphony according to the status of the 17 keys. If there is a key pressed, the corresponding note will be assigned to the first polyphony slot. If there are four keys pressed and the keyboard supports this key combination, all four slots will be used. If a fifth key is pressed, as there are no slots free, the note won’t be assigned to any slots. If one of the initial four keys is released, another key is pressed and the resulting combination is supported by the keyboard, the freed slot will be used by the that new key. The component outputs the addresses of the notes which will be used for the note memory that contains their respective periods, and the status of the four slots.

The component schematics and signals are presented in the following figure and table.

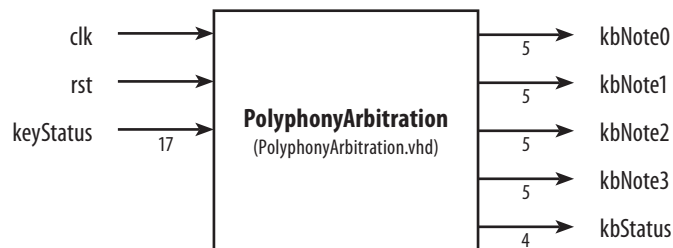
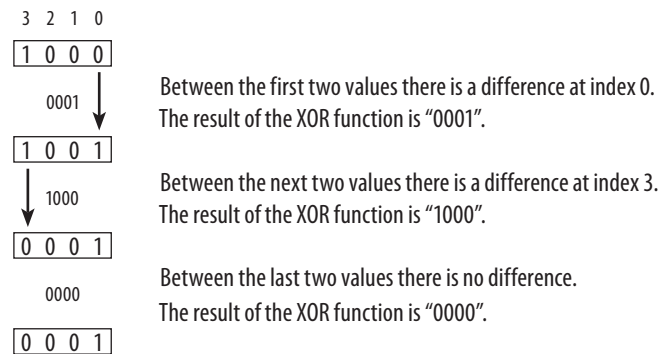


Figure 3.11. The “PolyphonyArbitration” component.

| Name | Width | Direction | Description |
|-----------|-------|-----------|---|
| clk | 1 | Input | Master clock (50 MHz) |
| rst | 1 | Input | Reset button (used for synchronous reset) |
| keyStatus | 17 | Input | The status of the keys ('0' = not pressed, '1' = pressed) |
| kbNote0 | 5 | Output | The address of the first slot note |
| kbNote0 | 5 | Output | The address of the second slot note |
| kbNote0 | 5 | Output | The address of the third slot note |
| kbNote0 | 5 | Output | The address of the fourth slot note |
| kbStatus | 4 | Output | The status of the four slots (identical to the status of the four keys) |

Table 3.8. Signal description of the “PolyphonyArbitration” component.

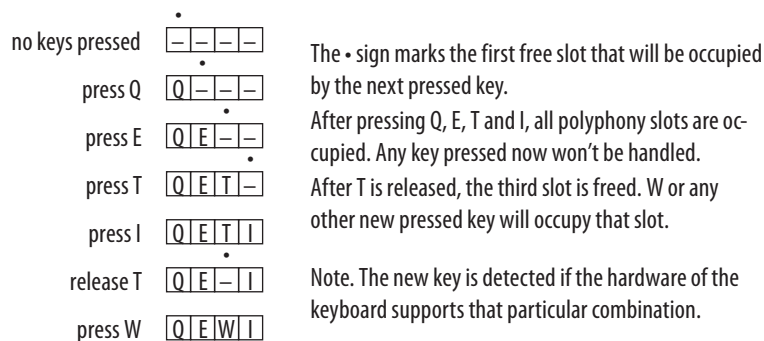
The “PolyphonyArbitration” component determines changes in the status of the keys by analyzing consecutive values of the “keyStatus” input. It determines the differences by applying logical XOR (exclusive OR) between the current value and the previous one. The following example illustrates the concept.



Logical XOR doesn’t detect the type of change — ‘0’ into ‘1’ for a pressed key, or ‘1’ into ‘0’ for a released key. Because the clock speed is very high, between two consecutive values there can be at most one change. The user can’t press or release keys at a speed high enough to cause two changes at the same time.

The design also determines the key that generated the event and the slot currently handling it. If the event that occurred is “press”, then one of the free slots will handle that key. If it is a “release” event, then the slot handling that key will be freed.

The design handles events according to their type and, in order to determine it, the component tests the status of the key that generated the event. If an event occurred on a key whose status is now ‘1’, then the key has been pressed. The first free slot — the first slot whose status is set to ‘0’ — will be occupied by this key. If an event occurred on a key whose status is now ‘0’, then the key has been released. The slot currently handling the key will be freed. The following example illustrates the component operation.



The component outputs the status of the four polyphony slots which controls the operation of the oscillators. It outputs also the values of the four notes that will be rendered by the oscillators.

3.3.3 UsbUnit

The “UsbUnit” component represents the hardware interface to the software application. It sends and receives data as MIDI messages and controls the “EffectsUnit”.

The component schematics and signals are presented in the following figure and table.

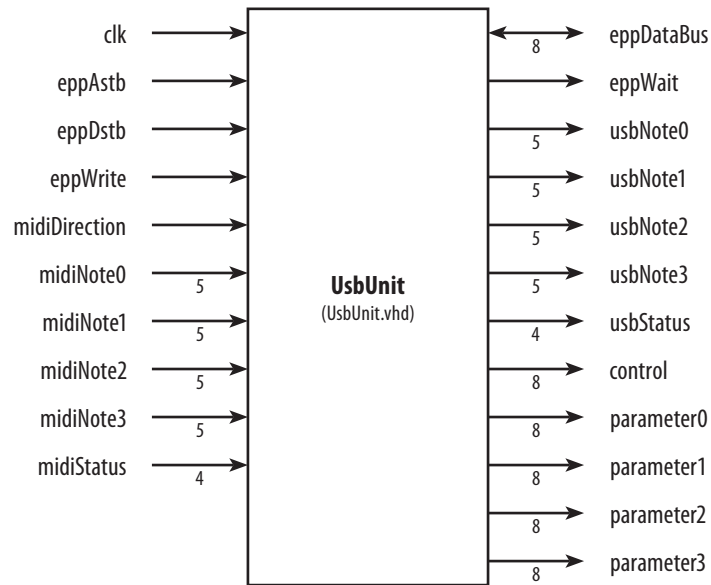


Figure 3.12. The “UsbUnit” component.

| Name | Width | Direction | Description |
|---------------|-------|---------------|--|
| clk | 1 | Input | Master clock (50 MHz) |
| eppAstb | 1 | Input | Address strobe of the EPP protocol |
| eppDstb | 1 | Input | Data strobe of the EPP protocol |
| eppWrite | 1 | Input | Write signal of the EPP protocol |
| midiDirection | 1 | Input | Signal that selects the direction of MIDI messages '0' = board to board/PC, '1' = PC to board |
| midiNote0 | 5 | Input | The address of the note played by the first oscillator |
| midiNote1 | 5 | Input | The address of the note played by the second oscillator |
| midiNote2 | 5 | Input | The address of the note played by the third oscillator |
| midiNote3 | 5 | Input | The address of the note played by the fourth oscillator |
| midiStatus | 4 | Input | The status of the four oscillators |
| eppDataBus | 8 | Bidirectional | Bidirectional data bus of the EPP protocol |
| eppWait | 1 | Output | Wait signal of the EPP protocol |
| usbNote0 | 5 | Output | The address of the note that the first oscillator plays when MIDI direction is from PC to board |
| usbNote1 | 5 | Output | The address of the note that the second oscillator plays when MIDI direction is from PC to board |
| usbNote2 | 5 | Output | The address of the note that the third oscillator plays when MIDI direction is from PC to board |
| usbNote3 | 5 | Output | The address of the note that the fourth oscillator plays when MIDI direction is from PC to board |
| usbStatus | 4 | Output | The status of the oscillators when MIDI direction is from PC to board |
| control | 8 | Output | Control register used in the rest of the design |

| | | | |
|------------|---|--------|---|
| parameter0 | 8 | Output | Parameter that controls the effects |
| parameter1 | 8 | Output | Parameter that controls the feedback amount |
| parameter2 | 8 | Output | Parameter that controls the wet amount |
| parameter3 | 8 | Output | Parameter that controls the dry amount |

Table 3.9. Signal description of the “UsbUnit” component.

The FPGA board communicates with the PC via USB. The USB controller installed on the board handles the USB communication and emulates the standard EPP protocol — an 8-bit bidirectional parallel data bus with handshake lines to control the data transfer.

The design uses an 8-bit address register and nine 8-bit data registers. The address contained by the address register selects the corresponding data register. The addresses of the registers are presented in the following table.

| Address | Register | Register usage |
|---------|------------|--|
| 00 | regControl | Selects the wave type generated by the oscillators, the MIDI messages direction and the current audio effect |
| 01 | regNote0 | Stores the note of the first oscillator |
| 02 | regNote1 | Stores the note of the second oscillator |
| 03 | regNote2 | Stores the note of the third oscillator |
| 04 | regNote3 | Stores the note of the fourth oscillator |
| 80 | regParam0 | Controls the effects unit |
| 81 | regParam1 | Stores the feedback coefficient |
| 82 | regParam2 | Stores the wet coefficient |
| 83 | regParam3 | Stores the dry coefficient |

The most significant bit of the control register selects the wave type that the oscillators generate. The next four bits are not used. The next two bits are used for selecting the audio effect. The last bit determines the direction of the MIDI messages.

Registers “regNote0”, “regNote1”, “regNote2” and “regNote3” store notes generated by the keyboard unit when “midiDirection” input signal is ‘0’ and they are sent to the PC for MIDI file recording. When “midiDirection” input signal is ‘1’, they store notes sent by the software application in order to be played by the oscillators.

The most significant bit of the note register controls the status of its corresponding oscillator. The next two bits are not used. The last five bits contain the address of the note. The PC application can access all 31 notes, unlike the keyboard which uses only 17 keys and so, 17 notes.

The “regParam0” register controls the parameters of the audio effects. The byte is internally split in the “EffectsUnit” component. Registers “regParam1”, “regParam2” and “regParam3” serve as input signals for the “Feedback” and “Mix” components as they contain the feedback, wet and dry coefficients.

Transfers between the PC and the board are initiated only by the PC, while the board only responds. In order to read or write one of the nine registers, the PC must send the address first. The selected register will either place its value on the data bus, or will store the data found on the data bus.

Transfers between the board and the PC are realized using four bus cycles:

- address write,
- address read,
- data write,
- data read.

The “UsbUnit” component uses a state machine with nine states in order to handle these four bus cycles.

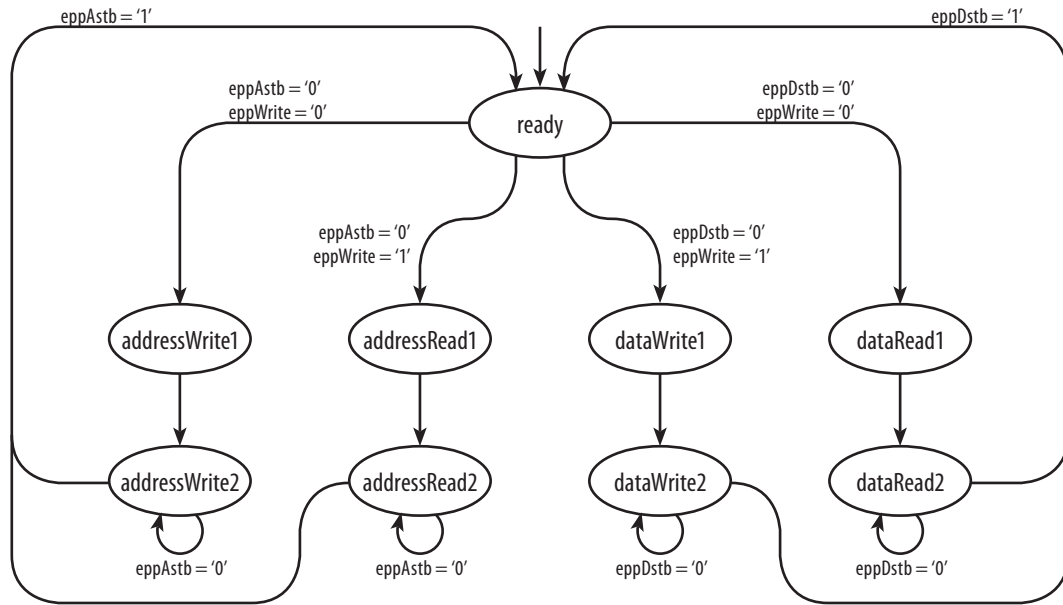


Figure 3.13. The state machine of the EPP protocol.

| State | Description |
|---------------|--|
| ready | The controller is ready for address or data operation. Both the address and data strobes are '1'. |
| addressWrite1 | The address strobe is '0' and the write strobe is also '0', which represents the start of an address write bus cycle. During this phase the address register is written with the value from the data bus. |
| addressWrite2 | The "eppWait" signal is set to '1' which means that the peripheral has completed the address write cycle. The state machine now waits for the host to complete the cycle (by setting the address strobe to '1'). |
| addressRead1 | The address strobe is '0' and the write strobe is '1', which represents the start of an address read bus cycle. During this phase the value of the address register is written on the data bus. |
| addressRead2 | The "eppWait" signal is set to '1' which means that the peripheral has completed the address read cycle. The state machine now waits for the host to complete the cycle (by setting the address strobe to '1'). |
| dataWriteA | The data strobe is '0' and the write strobe is also '0', which represents the start of a data write bus cycle. During this phase the selected data register is written with the value from the data bus. |
| dataWriteB | The "eppWait" signal is set to '1' which means that the peripheral has completed the data write cycle. The state machine now waits for the host to complete the cycle (by setting the address strobe to '1'). |
| dataReadA | The data strobe is '0' and the write strobe is '1', which represents the start of a data read bus cycle. During this phase the value of the selected data register is written on the data bus. |
| dataReadB | The "eppWait" signal is set to '1' which means that the peripheral has completed the data read cycle. The state machine now waits for the host to complete the cycle (by setting the address strobe to '1'). |

Table 3.10. State machine description.

The waveforms of the EPP signals during the four cycles are presented in the following figures.

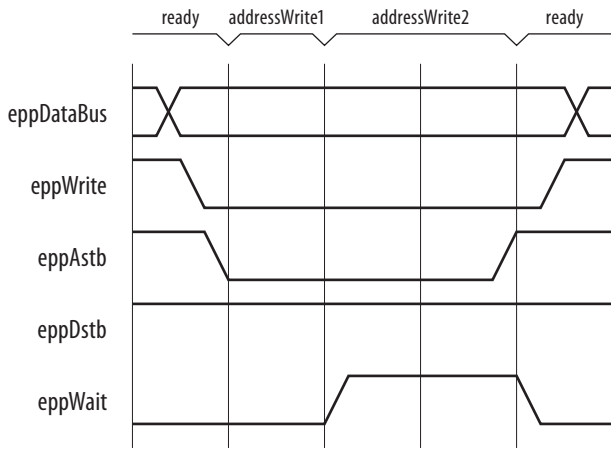


Figure 3.14. Address write cycle.

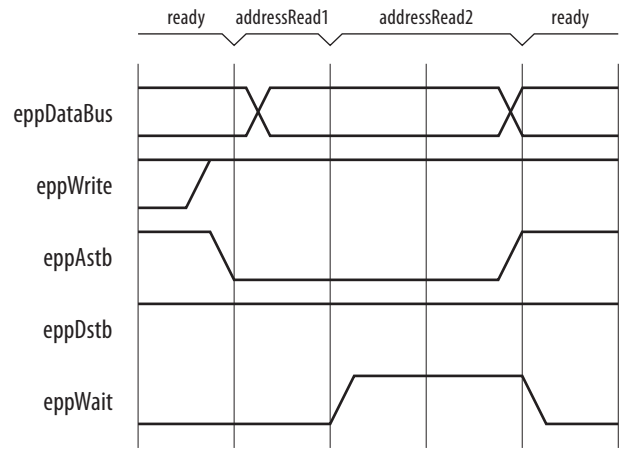


Figure 3.15. Address read cycle.

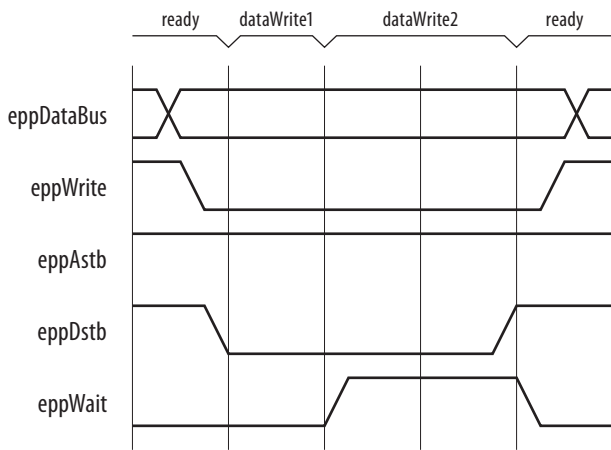


Figure 3.16. Data write cycle.

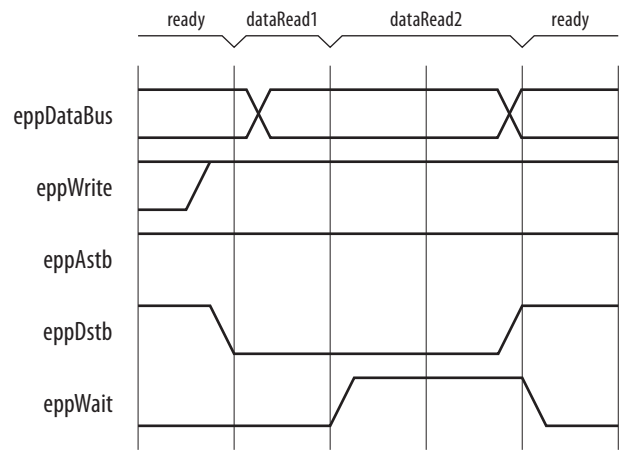


Figure 3.17. Data read cycle.

3.3.4 NoteSelector

The “NoteSelector” component selects between the notes and status coming from the “KeyboardUnit” and the notes and status coming from the “UsbUnit”. The chosen notes and status are sent to the oscillators.

The component schematics and signals are presented in the following figure and table.

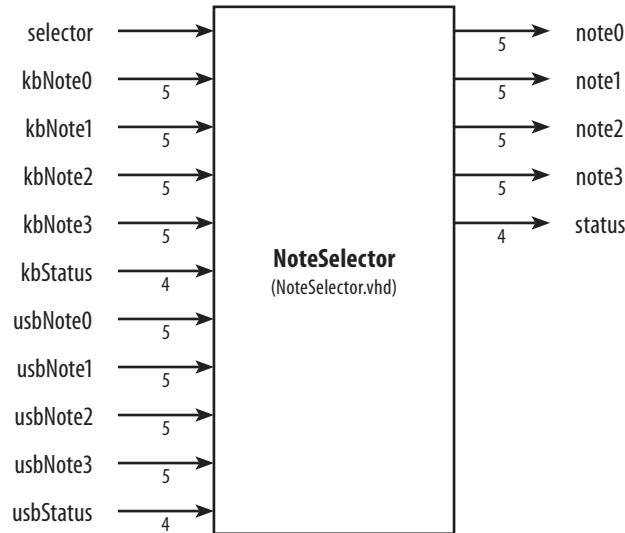


Figure 3.18. The “NoteSelector” component.

| Name | Width | Direction | Description |
|-----------|-------|-----------|--|
| selector | 1 | Input | Signal that selects between keyboard notes ('0') and USB notes ('1') |
| kbNote0 | 5 | Input | The first note address generated by the keyboard |
| kbNote1 | 5 | Input | The second note address generated by the keyboard |
| kbNote2 | 5 | Input | The third note address generated by the keyboard |
| kbNote3 | 5 | Input | The fourth note address generated by the keyboard |
| kbStatus | 4 | Input | The status of the oscillators generated by the keyboard |
| usbNote0 | 5 | Input | The first note address received by the USB unit |
| usbNote1 | 5 | Input | The second note address received by the USB unit |
| usbNote2 | 5 | Input | The third note address received by the USB unit |
| usbNote3 | 5 | Input | The fourth note address received by the USB unit |
| usbStatus | 4 | Input | The status of the oscillators received by the USB unit |
| note0 | 5 | Output | The note address of the first oscillator |
| note1 | 5 | Output | The note address of the second oscillator |
| note2 | 5 | Output | The note address of the third oscillator |
| note3 | 5 | Output | The note address of the fourth oscillator |
| status | 4 | Output | The status of the oscillators |

Table 3.11. Signal description of the “NoteSelector” component.

The “selector” input signal comes from the “UsbUnit” and decides who controls the oscillators. If “selector” = ‘0’, then the notes generated by the keyboard are played. If “selector” = ‘1’, then the notes received by the “UsbUnit” from the PC are sent to the oscillators in order to be played.

3.3.5 OscillatorsUnit

This component groups the following submodules: *NoteMemory*, *OscillatorGroup* and *Sampler*, which generate the period of the notes, render the notes and resamples the data to a constant 48 kHz.

The component schematics, structure and signals are presented in the following figures and table.

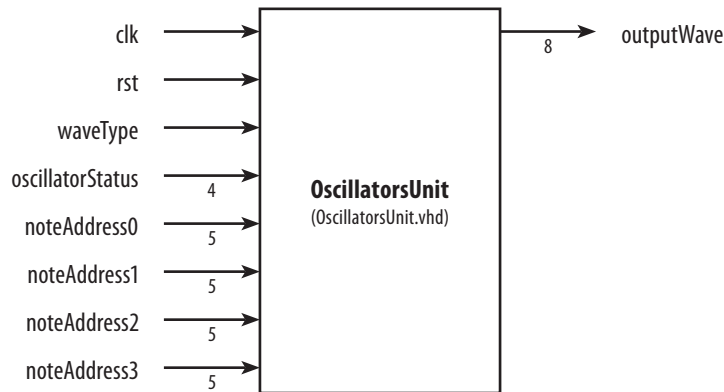


Figure 3.19. The “OscillatorsUnit” component.

| Name | Width | Direction | Description |
|------------------|-------|-----------|--|
| clk | 1 | Input | Master clock (50 MHz) |
| rst | 1 | Input | Reset button (used for synchronous reset) |
| waveType | 1 | Input | Signal that allows selecting between sawtooth ('0') and square ('1') waves |
| oscillatorStatus | 4 | Input | The status of the four oscillators controlled by the status of the four keys |
| noteAddress0 | 5 | Input | The note of the first oscillator |
| noteAddress1 | 5 | Input | The note of the second oscillator |
| noteAddress2 | 5 | Input | The note of the third oscillator |
| noteAddress3 | 5 | Input | The note of the fourth oscillator |
| outputWave | 8 | Output | The wave generated by the four oscillators |

Table 3.12. Signal description of the “OscillatorsUnit” component.

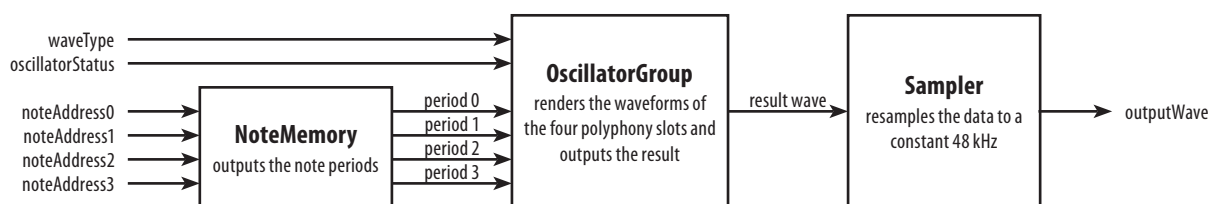


Figure 3.20. The internal structure of the “OscillatorsUnit” component.

3.3.5.1 NoteMemory

The “NoteMemory” component contains the periods of the 31 notes, where a period is represented by the number of clock ticks of the 50 MHz master clock.

The component schematics and signals are presented in the following figure and table.

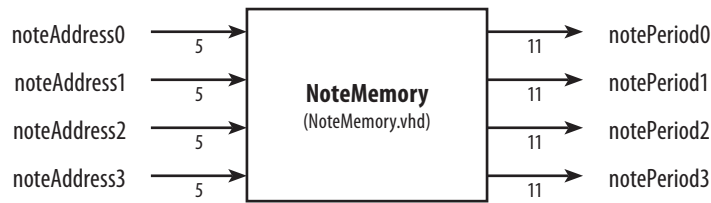


Figure 3.21. The “NoteMemory” component.

| Name | Width | Direction | Description |
|--------------|-------|-----------|--------------------------------|
| noteAddress0 | 5 | Input | The address of the first note |
| noteAddress1 | 5 | Input | The address of the second note |
| noteAddress2 | 5 | Input | The address of the third note |
| noteAddress3 | 5 | Input | The address of the fourth note |
| notePeriod0 | 11 | Output | The period of the first note |
| notePeriod1 | 11 | Output | The period of the second note |
| notePeriod2 | 11 | Output | The period of the third note |
| notePeriod3 | 11 | Output | The period of the fourth note |

Table 3.13. Signal description of the “NoteMemory” component.

The component is a 32×11-bit memory. The “PolyphonyArbitration” component is able to access locations only between 7 and 23 (17 locations for 17 keys), while the software application is able to access the entire memory, except for location 31 which is reserved for internal operation (it is the default value for unused polyphony slots).

The frequency of note F3 is 175 Hz. The frequency of the master clock is 50 MHz. Therefore the note takes $50,000,000 / 175 = 285,714$ clock ticks in a period. The note B5 takes $50,000,000 / 988 = 50,607$ clock ticks. Because the oscillators generate 256 samples per period, the resulting note period used by this design is 256 times smaller. F3 needs $285,714 / 256 = 1,116$ clock ticks and B5 needs 197 clock ticks for each sample. The note, as a whole, will have the right period.

3.3.5.2 OscillatorGroup

The “OscillatorGroup” is composed of the following submodules: *Oscillator* and *Adder*. There are four “Oscillator” instances that generate waves which are added together to form the output.

The component schematics, structure and signals are presented in the following figures and table.

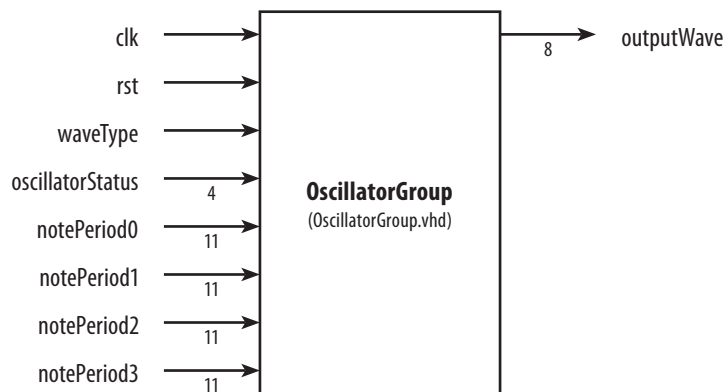


Figure 3.22. The “OscillatorGroup” component.

| Name | Width | Direction | Description |
|------------------|-------|-----------|--|
| clk | 1 | Input | Master clock (50 MHz) |
| rst | 1 | Input | Reset button (used for synchronous reset) |
| waveType | 1 | Input | Signal that allows selecting between sawtooth ('0') and square ('1') waves |
| oscillatorStatus | 4 | Input | The status of the four oscillators controlled by the status of the four keys |
| notePeriod0 | 11 | Input | The note period of the first oscillator |
| notePeriod1 | 11 | Input | The note period of the second oscillator |
| notePeriod2 | 11 | Input | The note period of the third oscillator |
| notePeriod3 | 11 | Input | The note period of the fourth oscillator |
| outputWave | 8 | Output | The wave generated by the four oscillators |

Table 3.14. Signal description of the “OscillatorGroup” component.

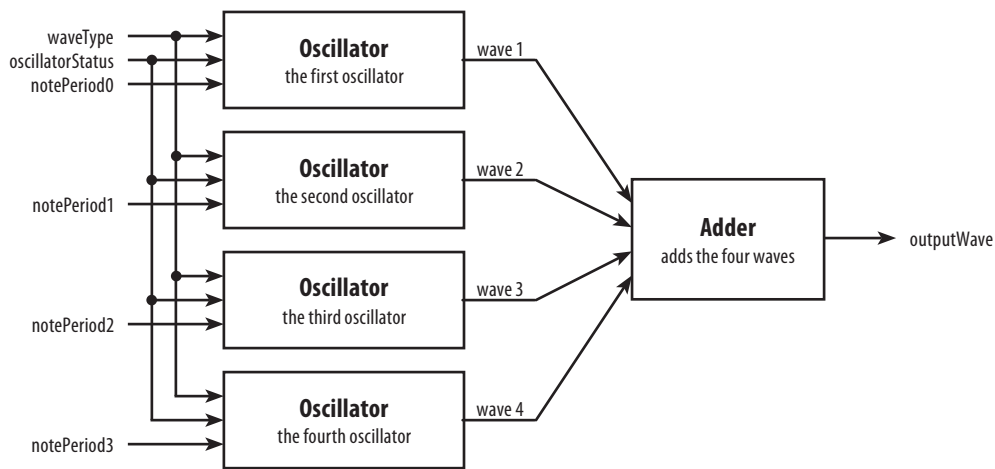


Figure 3.23. The internal structure of the “OscillatorGroup” component.

3.3.5.2.1 Oscillator

The “Oscillator” component generates the wave according to the note period. It can generate a sawtooth or square wave, depending on user’s choice. The wave is then enveloped using the ADSR model.

The component schematics, structure and signals are presented in the following figures and table.

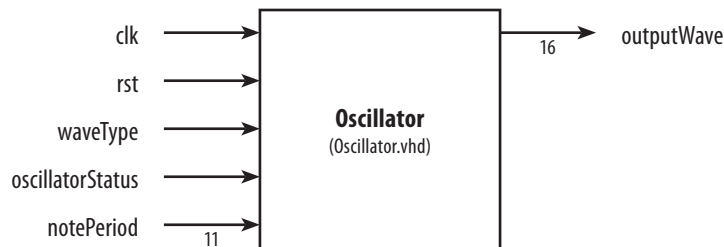


Figure 3.17. The “Oscillator” component.

| Name | Width | Direction | Description |
|----------|-------|-----------|--|
| clk | 1 | Input | Master clock (50 MHz) |
| rst | 1 | Input | Reset button (used for synchronous reset) |
| waveType | 1 | Input | Signal that allows selecting between sawtooth ('0') and square ('1') waves |

| | | | |
|------------------|----|--------|--|
| oscillatorStatus | 1 | Input | The status of the oscillator controlled by the status of the corresponding key |
| notePeriod | 11 | Input | The note period of the oscillator |
| outputWave | 16 | Output | The wave generated by the oscillator |

Table 3.15. Signal description of the “Oscillator” component.

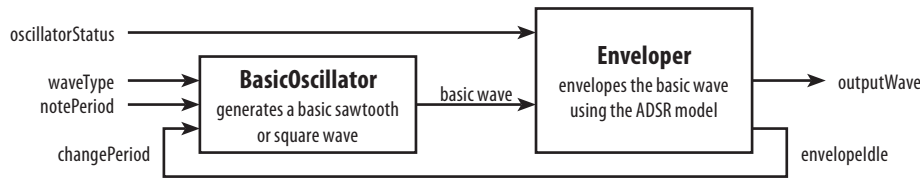


Figure 3.25. The internal structure of the “Oscillator” component.

“Oscillator” is composed of the following submodules: *BasicOscillator* and *Enveloper*.

The “BasicOscillator” component generates a basic waveform — sawtooth or square, according to the user’s choice.

The component schematics and signals are presented in the following figure and table.

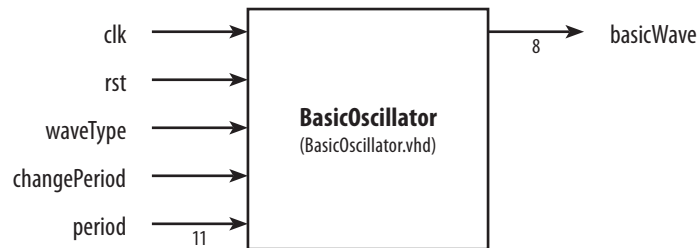


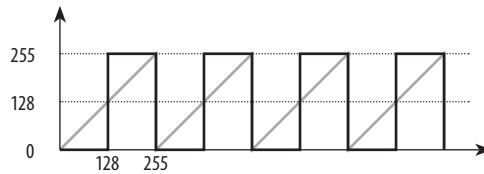
Figure 3.26. The “BasicOscillator” component.

| Name | Width | Direction | Description |
|--------------|-------|-----------|---|
| clk | 1 | Input | Master clock (50 MHz) |
| rst | 1 | Input | Reset button (used for synchronous reset) |
| waveType | 1 | Input | Signal that allows selecting between sawtooth ('0') and square ('1') waves |
| changePeriod | 1 | Input | Signal that enables the period to change only when the ADSR process is finished |
| period | 11 | Input | The note period of the basic oscillator |
| basicWave | 8 | Output | The wave generated by the basic oscillator |

Table 3.16. Signal description of the “BasicOscillator” component.

The period that the “BasicOscillator” uses must not change during the ADSR process. The moment a key is released, the corresponding slot is marked as unused and the corresponding period is set to the default value (location 31 in the note memory). This change of period must not affect the basic oscillator, because although the key was released, the note rendering continues with the release phase of the ADSR envelope model. The basic oscillator reads the new note period only before the enveloper starts the attack phase of a new note.

The component generates the sawtooth wave by counting from 0 to 255. It generates the square wave by comparing the sawtooth with the threshold value of 128 — if lower, the square value is 0; otherwise, it’s 255.



The “Envelope” modulates the amplitude of the waves generated by the basic oscillator, using the ADSR envelope model.

The component schematics and signals are presented in the following figure and table.



Figure 3.27. The “Envelope” component.

| Name | Width | Direction | Description |
|------------------|-------|-----------|--|
| clk | 1 | Input | Master clock (50 MHz) |
| rst | 1 | Input | Reset button (used for synchronous reset) |
| oscillatorStatus | 1 | Input | The status of the oscillator |
| basicWave | 8 | Input | The wave generated by the basic oscillator |
| envelopIdle | 1 | Output | Flag which signals that the envelope process is idle |
| modulatedWave | 16 | Output | The wave modulated in amplitude |

Table 3.17. Signal description of the “Envelope” component.

The basic waves that comes from the basic oscillator has values that range from 0 to 255, so it has an amplitude value of 256. The ADSR envelope model reduces this amplitude by different amounts during different phases. Amplitude modulation consists of multiplying the original signal by coefficients with values between 0 and 1. In this design the coefficients have values between 0 and 255 and the result of the multiplication is divided by 256 (or right-shifted by 8 positions, equivalent to ignoring the eight least significant bits).

| Phase | Characteristics |
|---------|--|
| attack | The phase starts when the key is pressed. The modulation coefficient (modulator) is gradually increased from 0 to 32. The phase lasts for 0.2 seconds. |
| decay | The modulation coefficient is gradually decreased from 32 to 16. The phase lasts for 0.4 seconds. |
| sustain | The modulation coefficient has a constant value of 16. The phase lasts for as long as the key is pressed. |
| release | The phase starts when the key is released. The modulation coefficient is gradually decreased from 16 to 0. The phase lasts for approximately 0.6 seconds. |

Table 3.18. ADSR envelope model phases.

The enveloper uses a counter to generate the modulator and a multiplier to obtain the modulated wave.

The “Envelope” component uses a state machine with five states for ADSR amplitude modulation.

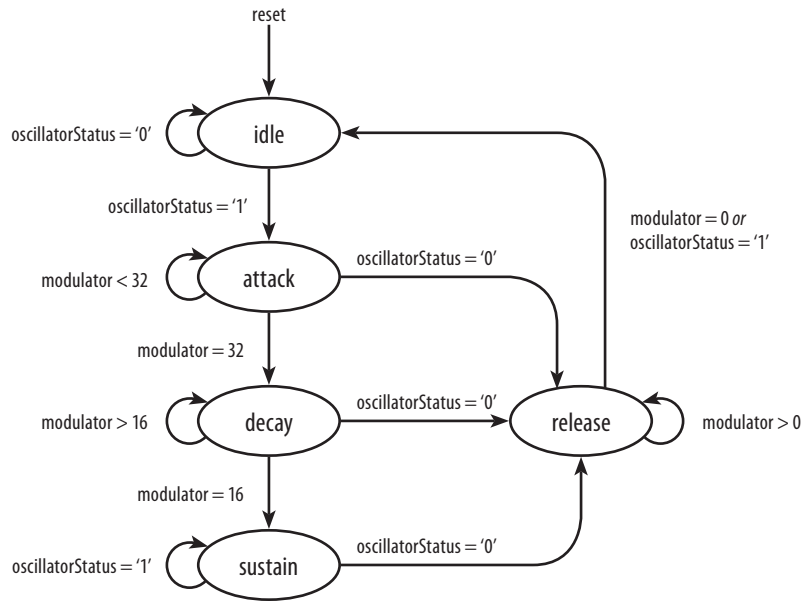


Figure 3.28. The state machine used for ADSR amplitude modulation.

| State | Description |
|---------|--|
| idle | The state machine stays in this state as long as the “oscillatorStatus” input is ‘0’ which means that the oscillator doesn’t render any notes. The moment a key is pressed and is assigned to the oscillator, the state machine goes into the next state (attack). In this state, the value of the modulator is 0. |
| attack | This state starts when the key is pressed. In this state, the value of the modulator increases from 0 to 32. While the value is below 32, the machine stays in this state. When the value is 32, the state machine goes into the next state (decay). If the key is released during this state (“oscillatorStatus” becomes ‘0’), then the state machine goes into the release state with the current value of the modulator (which may be lower than 32). |
| decay | In this state, the value of the modulator decreases from 32 to 16. While the value is above 16, the machine stays in this state. When the value is 16, the state machine goes into the next state (release). If the key is released during this state (“oscillatorStatus” becomes ‘0’), then the state machine goes into the release state with the current value of the modulator (which may be greater than 16). |
| sustain | In this state, the value of the modulator is 16. The machine stays in this state as long as the “oscillatorStatus” input is ‘1’, that is while the key is pressed. When it becomes ‘0’, the state machine goes into the release state with the value of the modulator equal to 16. |
| release | This state starts when the key is released. In this state, the value of the modulator decreases from 16 to 0. While the value is above 0, the machine stays in this state. When the value is 0, the state machine goes into the idle state. If at any time another key is pressed, the state machine goes into the idle state to start a new envelope process. The release phase of the previous note might not be completed. |

Table 3.19. State machine description.

The counter used by the modulation is controlled so that during the attack phase it counts up and during decay and release it counts down. At any time the counter can be interrupted in order to continue with the count down of the release phase. Also, at any time the release phase can be interrupted by the attack phase of a new key, which will start with the value of the modulator at which the release phase ended. This allows faster notes to be heard. Otherwise, these notes would have been very silent as they never use a larger modulator value.

The multiplier simply multiplies the 8-bit basic wave by the 8-bit modulator; the result is the 16-bit modulated wave. Each of the four 16-bit modulated waves are added by the next component and the result is divided by 256.

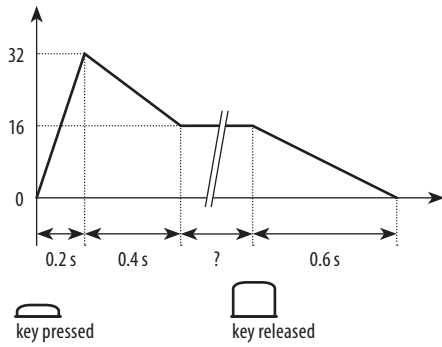


Figure 3.29. The ADSR characteristics.

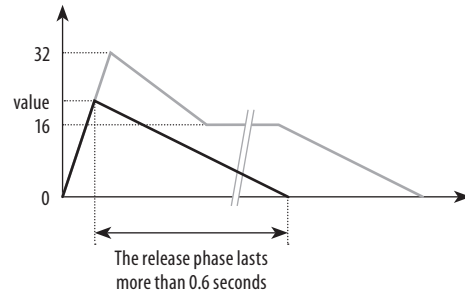


Figure 3.30. Attack interrupted by release.

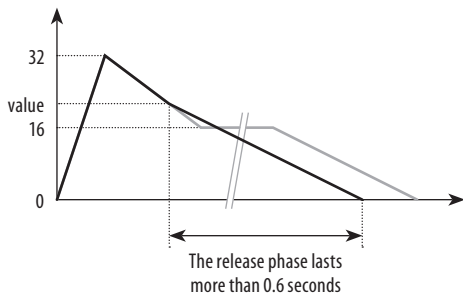


Figure 3.31. Decay interrupted by release.

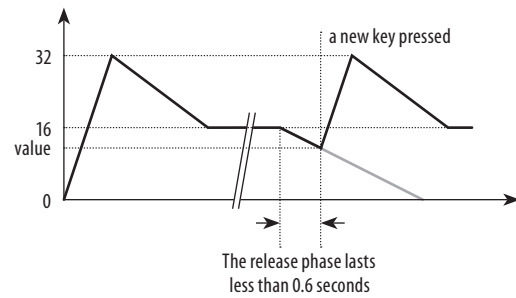


Figure 3.32. Release interrupted by attack.

3.3.5.2.2 Adder

The “Adder” component adds the waves generated by the four oscillators.

The component schematics and signals are presented in the following figure and table.

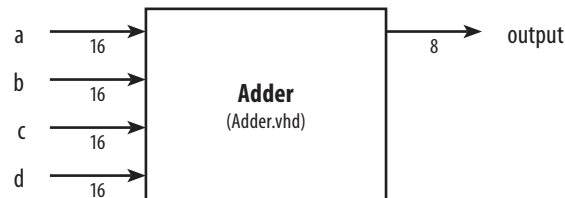


Figure 3.33. The “Adder” component.

| Name | Width | Direction | Description |
|--------|-------|-----------|--------------------|
| a | 16 | Input | The first wave |
| b | 16 | Input | The second wave |
| c | 16 | Input | The third wave |
| d | 16 | Input | The fourth wave |
| output | 8 | Output | The resulting wave |

Table 3.20. Signal description of the “Adder” component.

The input signals of the “Adder” component are the waves that the four oscillators generate. These values are 16-bit wide as a result of the multiplication of two 8-bit values. The sum is divided by 256 which is equivalent to ignoring the eight least significant bits. Although the adder could have added four previously divided values, it performs only one division because it leads to a smaller bit error (after one division one bit can be lost, after four divisions — four).

3.3.5.3 Sampler

The “Sampler” component adjusts the number of samples per second of the wave output by the adder to a constant value of 48 kHz.

The component schematics and signals are presented in the following figure and table.

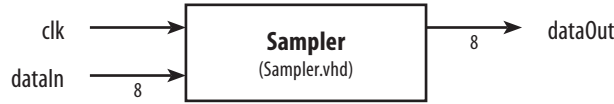


Figure 3.34. The “Sampler” component.

| Name | Width | Direction | Description |
|---------|-------|-----------|-----------------------|
| clk | 1 | Input | Master clock (50 MHz) |
| dataIn | 8 | Input | Data to be sampled |
| dataOut | 8 | Output | Sampled data (48 kHz) |

Table 3.21. Signal description of the “Sampler” component.

Various notes are rendered at different sample rates. For instance, the lowest note, F3, with a frequency of 175 Hz generates 44,800 samples per second. The highest note, B5, with a frequency of 988 Hz generates 252,928 samples per second. The “Sampler” outputs the incoming data at a frequency of 48 kHz. The resampling process slightly reduces the quality of the sound, but a constant value is more easily handled by the effects unit.

3.3.6 EffectsUnit

This component groups the following submodules: *DelayEcho*, *VibratoFlanger*, *Reverb*, *CircularBuffer*, *Feedback* and *Mix*, which apply various audio effects (delay, echo, reverb, vibrato and flange) on the sounds generated by the oscillators.

The component schematics, structure and signals are presented in the following figures and table.

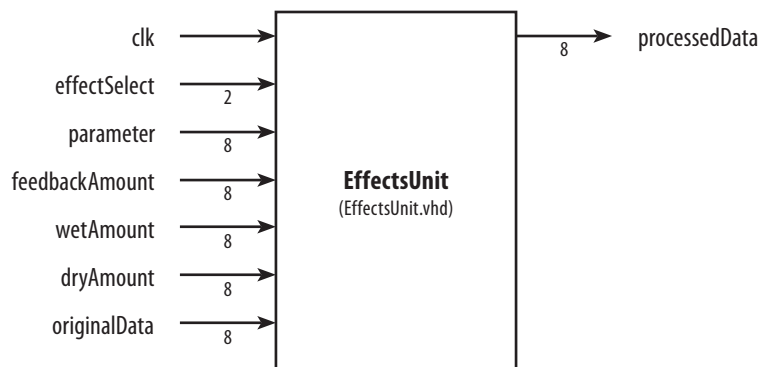


Figure 3.35. The “EffectsUnit” component.

| Name | Width | Direction | Description |
|------|-------|-----------|-----------------------|
| clk | 1 | Input | Master clock (50 MHz) |

| | | | |
|----------------|---|--------|---|
| effectSelect | 2 | Input | Signal that selects the audio effect: "00" – no effect, "01" – delay or echo, "10" – vibrato or flange, "11" – reverb. (The signal is controlled by the software application via USB) |
| parameter | 8 | Input | Signal that controls certain parameters of the effects: parameter(7:6) – "DelayEcho" module: delayTime parameter(5:1) – "VibratoFlange" module: effectType, maximumDelay, modulationRate parameter(0) – "Reverb" module: decayType (The signal is controlled by the software application via USB) |
| feedbackAmount | 8 | Input | The amount of processed data reinserted in the effect unit (The signal is controlled by the software application via USB) |
| wetAmount | 8 | Input | The amount of processed data to be sent for output (The signal is controlled by the software application via USB) |
| dryAmount | 8 | Input | The amount of original (not processed) data to be sent for output (The signal is controlled by the software application via USB) |
| originalData | 8 | Input | Data coming from the oscillators |
| processedData | 8 | Output | Processed data sent for output |

Table 3.22. Signal description of the "EffectsUnit" component.

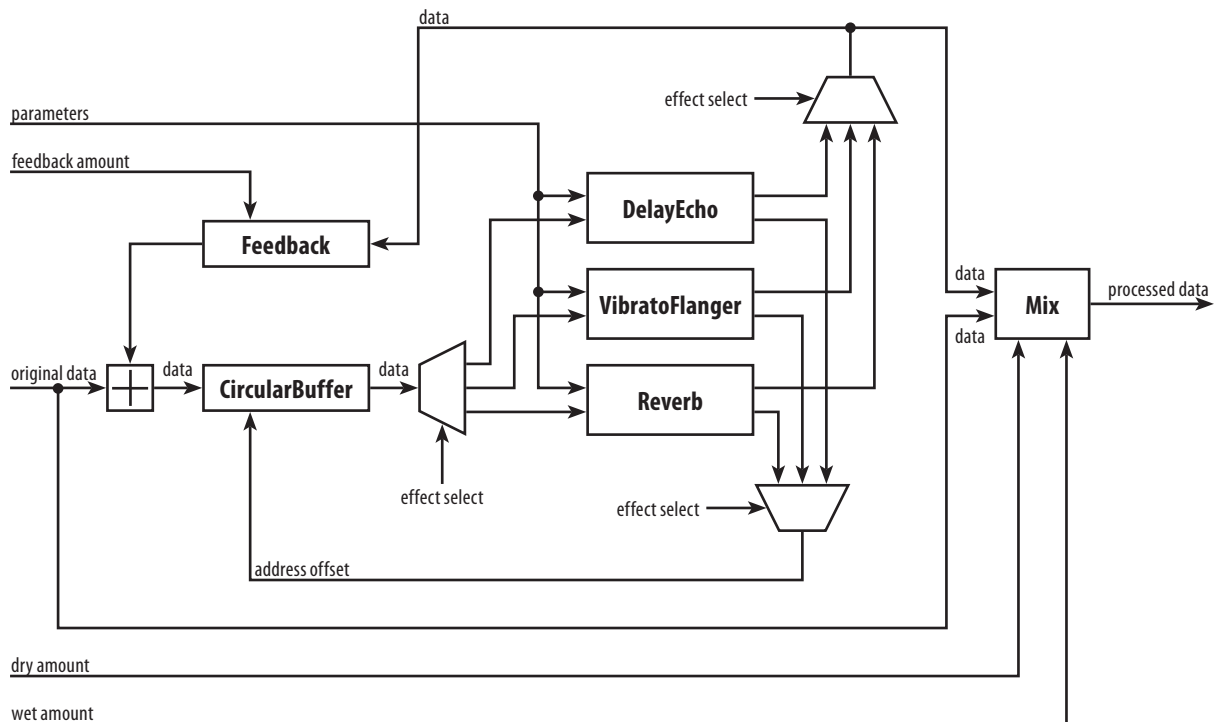


Figure 3.36. The internal structure of the "EffectsUnit" component.

3.3.6.1 CircularBuffer

The "CircularBuffer" component is a memory that stores the audio data. It allows reading values generated at various moments in time that are used by the effect units.

The component schematics and signals are presented in the following figure and table.

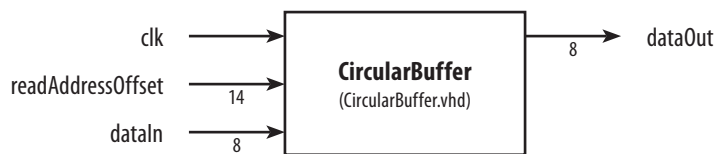


Figure 3.37. The “CircularBuffer” component.

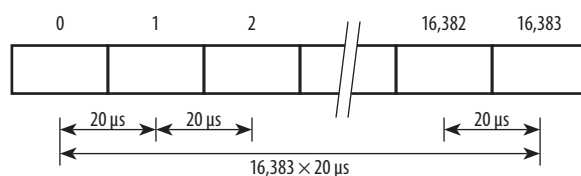
| Name | Width | Direction | Description |
|-------------------|-------|-----------|--|
| clk | 1 | Input | Master clock (50 MHz) |
| readAddressOffset | 14 | Input | The offset which helps generating the read address |
| dataIn | 8 | Input | Data to be written in the buffer |
| dataOut | 8 | Output | Data read from the buffer |

Table 3.23. Signal description of the “CircularBuffer” component.

The “CircularBuffer” is composed of a memory with 14-bit wide addresses and $2^{14} = 16,384$ locations. At a frequency of 48 kHz, equal to that of the “Sampler” component, the memory increments the address at which it stores the audio data and enables writing. This means consecutive samples are stored in consecutive locations.

Writing is allowed only when a new sample is generated. In the rest of the time the memory is enabled only for reading. The reading address is obtained from the “readAddressOffset” input signal generated by the effect units, which is subtracted from the current write address.

Between two consecutive samples there a time difference of $\frac{1}{48,000}$ of a second (almost 20 microseconds, due to the sample rate of 48,000 Hz). The time difference between the most recent and the least recent sample is almost a third of a second ($0.3413 \text{ seconds} = 16,383 \times 20 \mu\text{s}$). This allows effects such as delay or echo to be properly perceived. For larger time differences, using a larger memory or working at lower sample rates is required.



The component uses an internal counter that helps reducing the 50 MHz frequency of the master clock to 48 kHz. For this purpose, it counts from 0 to 1040 ($50,000,000 / 48,000 \approx 1041$). If its value is equal to 1040, writing in the memory is allowed. If its value is lower than 1040, the memory is enabled only for reading. The following figure shows how the circular buffer operates.

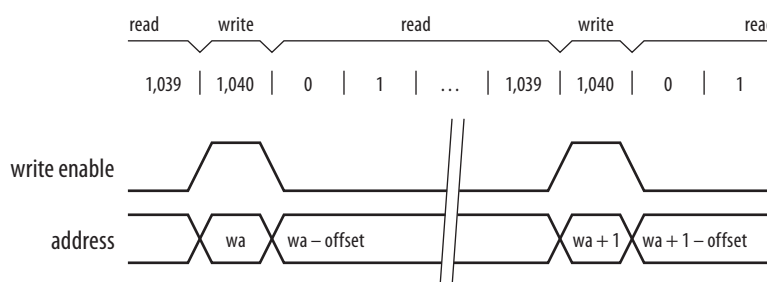


Figure 3.38. The circular buffer operation ($wa = \text{write address}$).

3.3.6.2 DelayEcho

The “DelayEcho” component generates both the delay and the echo effects.

The component schematics and signals are presented in the following figure and table.



Figure 3.39. The “DelayEcho” component.

| Name | Width | Direction | Description |
|-------------------|-------|-----------|--|
| delayTime | 2 | Input | Signal that allows choosing between four delay preset values |
| dataIn | 8 | Input | Data received from the circular buffer |
| readAddressOffset | 14 | Output | The offset which helps generating the read address |
| dataOutWet | 8 | Output | Processed audio data |

Table 3.24. Signal description of the “DelayEcho” component.

The component generates an address offset that is sent to the circular buffer. The offset is generated according to the “delayTime” input signal. There are four preset values of the address offset. The time difference between two consecutive addresses is approximately 20 microseconds ($\frac{1}{48,000}$ of a second).

| delayTime | readAddressOffset | Decimal value | Time offset |
|-----------|-------------------|---------------|---|
| “00” | “00011111010000” | 2,000 | $2,000 \times 20 \mu\text{s} = 40 \text{ ms}$ |
| “01” | “00111111010000” | 4,000 | $4,000 \times 20 \mu\text{s} = 80 \text{ ms}$ |
| “10” | “01111110100000” | 8,000 | $8,000 \times 20 \mu\text{s} = 160 \text{ ms}$ |
| “11” | “11111010000000” | 16,000 | $16,000 \times 20 \mu\text{s} = 320 \text{ ms}$ |

The data that the circular buffer returns is simply output by the “DelayEcho” component.

The difference between the two effects is in the amount of feedback. Delay has no feedback (the amount is 0%), while echo has a feedback between 0 and 100%. The “Feedback” component is described after the rest of the effect units.

3.3.6.3 VibratoFlanger

The “VibratoFlanger” component generates the vibrato and flange effects.

The component schematics and signals are presented in the following figure and table.

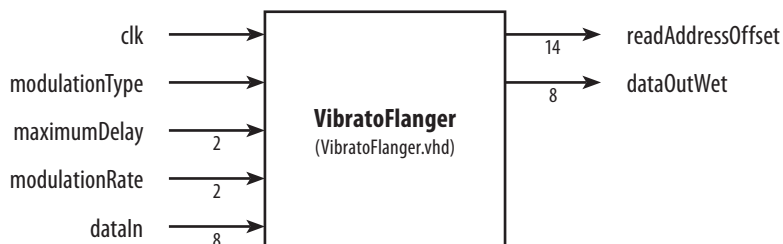


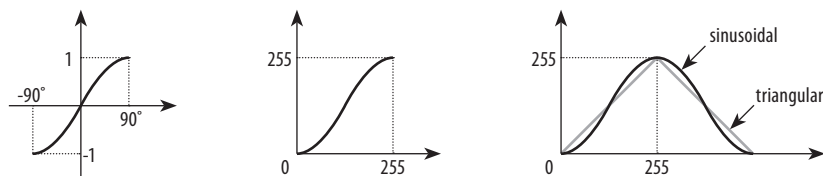
Figure 3.40. The “VibratoFlanger” component.

| Name | Width | Direction | Description |
|-------------------|-------|-----------|---|
| clk | 1 | Input | Master clock (50 MHz) |
| modulationType | 1 | Input | Signal that allows choosing between sinusoidal and triangular modulation of the delay |
| maximumDelay | 2 | Input | Signal that allows choosing between three delay preset values |
| modulationRate | 2 | Input | Signal that allows choosing between four modulation rate preset values |
| dataIn | 8 | Input | Data received from the circular buffer |
| readAddressOffset | 14 | Output | The offset which helps generating the read address |
| dataOutWet | 8 | Output | Processed audio data |

Table 3.25. Signal description of the “VibratoFlanger” component.

The component generates an address offset that is sent to the circular buffer. The offset is modulated according to the “modulationType” and “modulationRate” input signals. If “modulationType” = ‘0’ the modulation is sinusoidal; otherwise, it’s triangular.

For the sinusoidal modulation, the component uses a memory that stores 256 sine values.



Sine values for angles between -90° and 90° have been normalized and stored in the “SineMemory”. For a complete sine shape, the memory needs to be read from location 0 to location 255 and back to 0. The design uses a bidirectional counter that counts up from 0 to 255 and then counts down to 0. The generated values are indexes or addresses for the sine memory.

The “modulationType” selects between the sine memory values which generate a sinusoidal modulation and the indexes which generate a triangular modulation.

| modulationType | |
|----------------|------------|
| ‘0’ | sinusoidal |
| ‘1’ | triangular |

The “modulationRate” input signal allows selecting between four preset values that divide the master clock to the desired frequency at which the addresses for the sine memory are generated.

| modulationRate | Modulation frequency |
|----------------|----------------------|
| “00” | 1 Hz |
| “01” | 4 Hz |
| “10” | 8 Hz |
| “00” | 16 Hz |

The modulation outputs the address offsets which help generating the read address in the circular buffer. The time difference between two consecutive addresses is approximately 20 microseconds and, for an offset between 0 and 255, the circular buffer outputs a sample between 0 and $255 \times 20 \mu\text{s} \approx 5$ milliseconds old. In order to read a sample 10 milliseconds old, the offset is left-shifted once. In order to read a sample 20 milliseconds old, the offset is left-shifted twice.

| maximumDelay | Delay value | Address offset |
|--------------|-----------------|---------------------------|
| “00” | 5 milliseconds | 0 – 255 |
| “01” | 10 milliseconds | 0 – 510 = 255×2 |
| “10” | 20 milliseconds | 0 – 1020 = 255×4 |
| “11” | — | — |

The “VibratoFlanger” component outputs the data returned by the circular buffer.

The vibrato effect outputs only the processed data, which means that the dry coefficient is set to 0%. The flange effect mixes the dry and the wet data, which means that the dry coefficient has a value between 0 and 100%. Vibrato has no feedback, but flange has a feedback between 0 and 100%.

3.3.6.4 Reverb

The “Reverb” component generates the reverb effect.

The component schematics and signals are presented in the following figure and table.

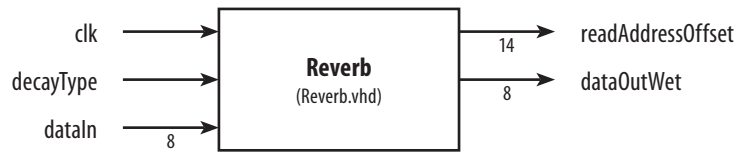


Figure 3.41. The “Reverb” component.

| Name | Width | Direction | Description |
|-------------------|-------|-----------|--|
| clk | 1 | Input | Master clock (50 MHz) |
| decayType | 1 | Input | Signal that allows choosing between no decay and exponential decay of the delayed data |
| dataIn | 8 | Input | Data received from the circular buffer |
| readAddressOffset | 14 | Output | The offset which helps generating the read address |
| dataOutWet | 8 | Output | Processed audio data |

Table 3.26. Signal description of the “Reverb” component.

The “Reverb” reads samples with delays of 5, 10, 20, 30, 45, 60, 75 and 100 milliseconds that are stored in eight corresponding registers. The design uses a counter that counts from 0 to 7. These values serve as indexes in the memory that stores the address offsets.

| Delay | readAddressOffset | Decimal value | Time offset | Decay coefficient |
|-------|-------------------|---------------|--|-------------------|
| 5 | “00000011111010” | 250 | $250 \times 20 \mu\text{s} = 5 \text{ ms}$ | 230 |
| 10 | “000001111110100” | 500 | $500 \times 20 \mu\text{s} = 10 \text{ ms}$ | 210 |
| 20 | “000011111101000” | 1,000 | $1,000 \times 20 \mu\text{s} = 20 \text{ ms}$ | 175 |
| 30 | “000101111011100” | 1,500 | $1,500 \times 20 \mu\text{s} = 30 \text{ ms}$ | 150 |
| 45 | “00100011001010” | 2,250 | $2,250 \times 20 \mu\text{s} = 45 \text{ ms}$ | 110 |
| 60 | “00101110111000” | 3,000 | $3,000 \times 20 \mu\text{s} = 60 \text{ ms}$ | 95 |
| 75 | “00111010100110” | 3,750 | $3,750 \times 20 \mu\text{s} = 75 \text{ ms}$ | 90 |
| 100 | “01001110001000” | 5,000 | $5,000 \times 20 \mu\text{s} = 100 \text{ ms}$ | 85 |

Data read from the circular buffer at the address offset of the 5 milliseconds delay is stored in the first register, data read at the address offset of the 10 milliseconds delay is stored in the second register and so on. The values stored in the eight registers are added together and divided by four (not eight which would be the right value), which gives more depth to the effect.

The amplitude of sound waves decreases after reflection. Most of the times, the decay is exponential. This means that the older waves are reduced by a greater amount. The “decayType” input signal allows choosing between no decay or exponential decay. If “decayType” is ‘0’, the registers store the value read from the buffer. If the value is ‘1’, then the registers store data read from the sample multiplied by a specific coeffi-

cient (255 is the maximum value and allows the sample to be stored as read from the buffer; 128 allows the sample to be stored at half value).

The “Reverb” component outputs the value returned by the adder. The feedback coefficient can have values between 0 and 100% for more depth of the effect. Large values of feedback can produce unexpected results (distorted sounds), as the system becomes instable.

3.3.6.5 Feedback

The “Feedback” component generates the processed data that is reinserted into the circular buffer.

The component schematics and signals are presented in the following figure and table.



Figure 3.42. The “Feedback” component.

| Name | Width | Direction | Description |
|----------------|-------|-----------|---|
| feedbackAmount | 8 | Input | The feedback coefficient that corresponds to the amount of processed data mixed with the original data that will be stored in the circular buffer |
| input | 8 | Input | The processed data |
| output | 8 | Output | The data that will be mixed with the original data in order to form the input of the circular buffer |

Table 3.27. Signal description of the “Feedback” component.

The “Feedback” component determines how much of the processed data will be mixed with the original data in order to form the input of the circular buffer. If the feedback coefficient is equal to 0%, the value of the “feedbackAmount” input signal is equal to 0 and if the feedback is equal to 100% then the value of “feedbackAmount” is 255.

The “Feedback” component uses a multiplier that multiplies two 8-bit values and the 16-bit result is internally divided by 256. The output of the multiplier is an 8-bit value.

3.3.6.6 Mix

The “Mix” component adds the original data multiplied by the dry coefficient to the processed data multiplied by the wet coefficient.

The component schematics and signals are presented in the following figure and table.



Figure 3.43. The “Mix” component.

| Name | Width | Direction | Description |
|-----------|-------|-----------|-------------------------------------|
| wetData | 8 | Input | The processed data |
| dryData | 8 | Input | The original data |
| wetAmount | 8 | Input | The amount of wet data to be output |
| dryAmount | 8 | Input | The amount of dry data to be output |
| mixedData | 8 | Output | The mixed data |

Table 3.28. Signal description of the “Mix” component.

The “Feedback” component uses two multipliers that multiply the original data by the dry coefficient and the processed data by the wet coefficient. The two results are added and the mixed data is output.

Each of the two multipliers has two 8-bit values as inputs; the 16-bit result is internally divided by 256, thus the output of the multiplier is an 8-bit value.

3.3.7 SpiUnit

The “SpiUnit” receives the audio samples and sends them bit by bit to the digital-to-analog converter connected to the FPGA board.

The component schematics and signals are presented in the following figure and table.

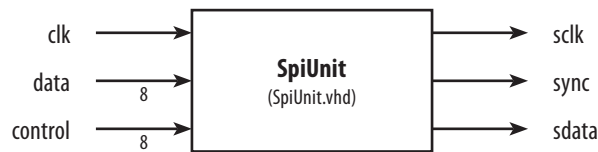


Figure 3.44. The “SpiUnit” component.

| Name | Width | Direction | Description |
|---------|-------|-----------|--|
| clk | 1 | Input | Master clock (50 MHz) |
| data | 8 | Input | The audio samples that will be sent to the converter |
| control | 8 | Input | The control byte needed by the converter (10h) |
| sclk | 1 | Output | Serial clock (25 MHz) |
| sync | 1 | Output | Synchronize signal |
| sdata | 1 | Output | Serial data |

Table 3.29. Signal description of the “SpiUnit” component.

The “SpiUnit” communicates with the digital-to-analog converter (DAC) through an SPI compatible interface. Data to be sent to the DAC is 16-bit wide. The first byte is the control byte and is used for selecting certain operation modes of the DAC. The second byte is the digital data that will be output as an analog voltage value.

The DAC operates at a frequency lower than 30 MHz. For the “sclk” output signal a value of 25 MHz has been chosen.

The control byte in this design is 10h, which means using only one of the two converters inside the chip (as there is only one stream of audio data) and using an internal voltage reference.

The “SpiUnit” component receives the control byte and the 8-bit sample and creates a 16-bit word. After sending each of the 16 bits, a synchronization impulse follows on the “sync” line. This instructs the converter to process the data it has received. During this impulse, the new data to be sent is assembled. This doesn’t allow data to change while being transmitted. The waveform of a send cycle is presented in the following figure.

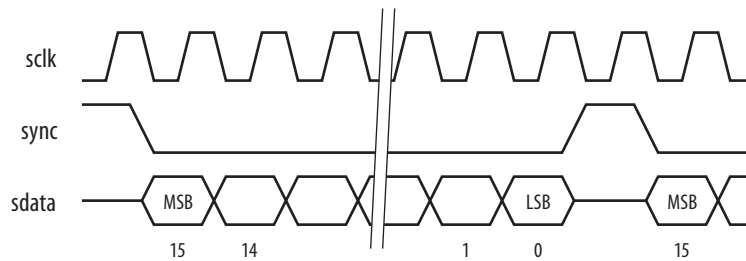


Figure 3.45. SPI data send cycle.

4. Software design

4.1 Design overview

The software application has been developed using Microsoft Visual C# 2005 Express Edition. The design has a modular approach, where each class has its own functionality. This document presents the classes used by the design and the structure of the application using use case, sequence and class diagrams.

4.2 Classes

The following table presents the classes designed for the software application.

| Class name | Description |
|--------------------------------|---|
| AboutForm | Displays basic information about the software application. |
| ApplicationForm | Defines the main window and handles events related to window elements such as buttons, menus etc. It also displays the music sheet of MIDI files. |
| AudioConsoleForm | Defines the audio console window and communicates with the FPGA device. |
| CONVERTER (static class) | Converts byte structures to integer values. |
| DelayEffectForm | Defines the delay effect window and sends the parameters of the effect to the FPGA device. |
| DisplayElements | Creates the structures that hold musical elements such as notes, rests and bars. |
| DPCUTIL (static class) | Defines the methods contained in the "dpcutil.dll" dynamic library. |
| EchoEffectForm | Defines the echo effect window and sends the parameters of the effect to the FPGA device. |
| FlangeEffectForm | Defines the flange effect window and sends the parameters of the effect to the FPGA device. |
| HelpForm | Defines the help window. |
| IDrawingElement (interface) | Defines the methods that all musical elements share. |
| MeasureBar | Defines the visual representation of the vertical bar between measures. |
| MidiDecoder | Decodes MIDI files from the PC. |
| MidiEncoder | Encodes MIDI files recorded from the FPGA device and assembles new MIDI files stored on the PC. |
| MidiInterpreter | Interprets the decoded MIDI file and generates the structure of the music sheet. |
| NewFileForm | Defines the new file window that controls the parameters of the file to be recorded from the FPGA device. |
| Note | Defines the visual representation of the musical note. |
| PlayBar | Defines the visual representation of the sign that shows the currently played measure. |

| | |
|----------------------|--|
| PlayMidi | Sends song data to the FPGA device. |
| RecordMidi | Records song data to the FPGA device. |
| Rest | Defines the visual representation of the musical rest. |
| ReverbEffectForm | It defines the reverb effect window and sends the parameters of the effect to the FPGA device. |
| Staff | Defines the visual representation of the horizontal lines of the staff. |
| UsbCommunicationForm | Defines the USB communication window that allows initializing the desired FPGA device. |
| VibratoEffectForm | Defines the vibrato effect window and sends the parameters of the effect to the FPGA device. |

Table 4.1. Classes of the software application.

4.3 UML diagrams

4.3.1 Use case diagram

The use case diagram is presented in the following figure and table.

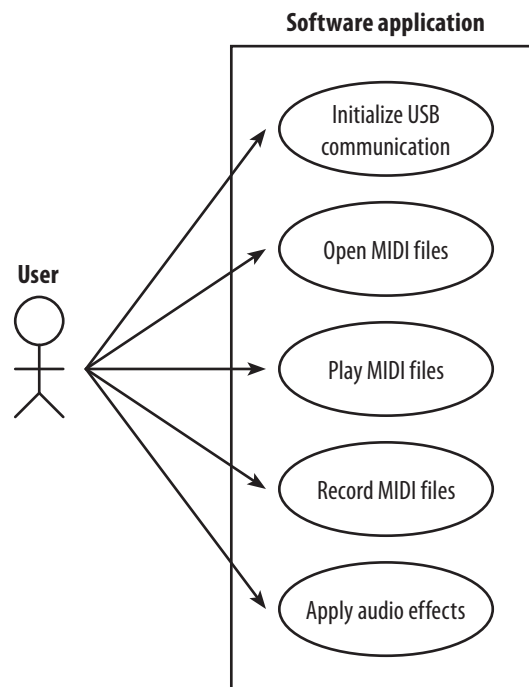


Figure 4.1. The use case diagram.

| Use case | Details |
|------------------------------|--|
| Initialize USB communication | The user selects “USB Communication...” in the menu of the application and initializes the USB communication with the desired FPGA device. |
| Open MIDI files | The user selects “Open...” in the menu of the application and selects in the “Open file” dialog box a MIDI file to open. The application displays the music sheet of the file. The user browses the sheet by pressing the “Previous” and “Next” buttons from the toolbar of the application. |
| Play MIDI files | The user presses the “Play” button from the toolbar of the application. The application sends adapted MIDI messages to the FPGA board. |
| Record MIDI files | The user presses the “Record” button from the toolbar of the application. The application receives adapted MIDI messages from the FPGA board. |
| Apply audio effects | The user selects “Audio Console...” in the menu of the application and selects a voice that the FPGA will use for playing or applies an audio effect on the sounds generated by the device. |

Table 4.2. Use case description.

4.3.2 Sequence diagrams

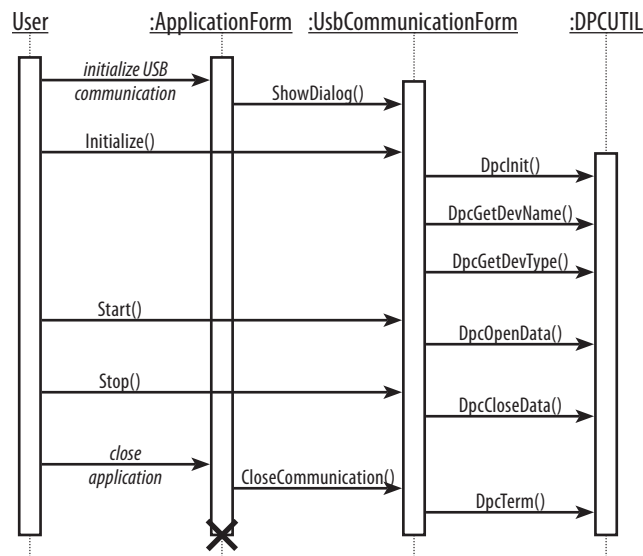


Figure 4.2. The sequence diagram for the “Initialize USB communication” use case.

When the user presses the “USB Communication...” in the main menu, a window appears that allows selecting an FPGA device. When the user presses the “Initialize” button of that window, the application calls the necessary methods of the “dpcutil.dll” and creates a list with all the available FPGA devices. The user selects the desired device in that list and presses the “Start” button which triggers the start sequence. The “Start” button becomes the “Stop” button, which, if pressed, triggers the stop sequence. When the user closes the main window, the application automatically calls the method that closes the communication with the FPGA device.

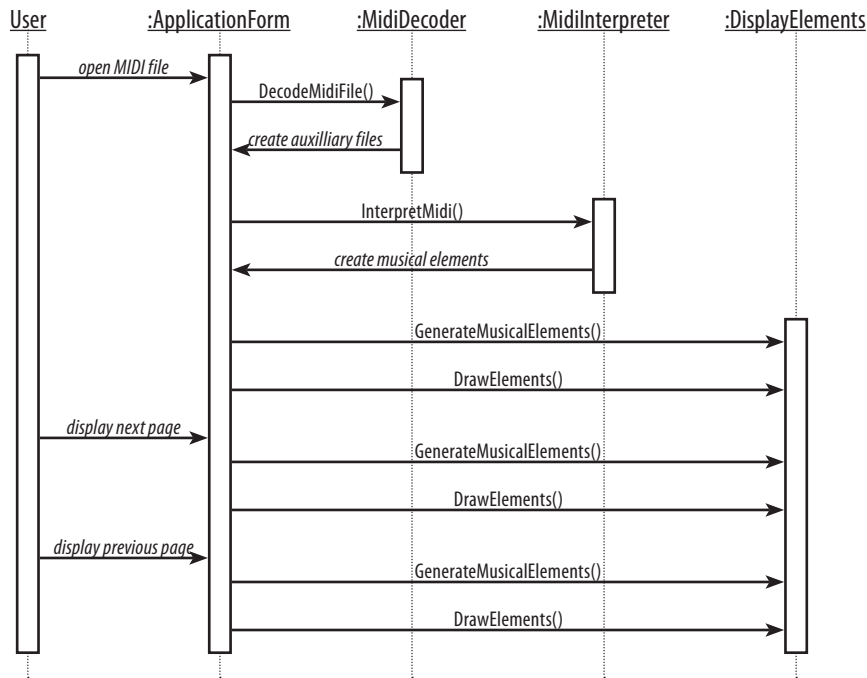


Figure 4.3. The sequence diagram for the “Open MIDI files” use case.

When the user selects “Open...” in the main menu, the application displays a dialog box that allows selecting the desired MIDI file. Afterwards, it creates a “MidiDecoder” object that reads the file and generates another file that contains certain events (only “note on” and “note off”). This file is interpreted by the “MidiInterpreter” object that generates one array list for each of the four tracks that correspond to the four-note polyphony. The application then generates the musical elements (note objects, rest objects etc.) corresponding to the first two measures of the song and displays them in the main window. If the user presses the “Next” button in the main toolbar, the application generates the musical elements corresponding to the next two measures of the song and displays them in the main window.

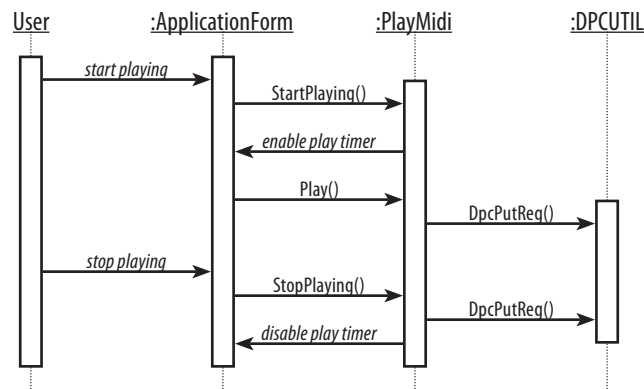


Figure 4.4. The sequence diagram for the “Play MIDI files” use case.

When the user presses the “Play” button of the toolbar, the application calls the “StartPlaying()” method of the “PlayMidi” object. The board is set to use the current selected voice and effect and the application can now start the timer according to which the adapted MIDI messages are sent. The “Play” button becomes the “Stop” button, which, if pressed, determines the application to call the “StopPlaying()” method of the “PlayMidi” object. The timer is disabled and the board stops playing notes.

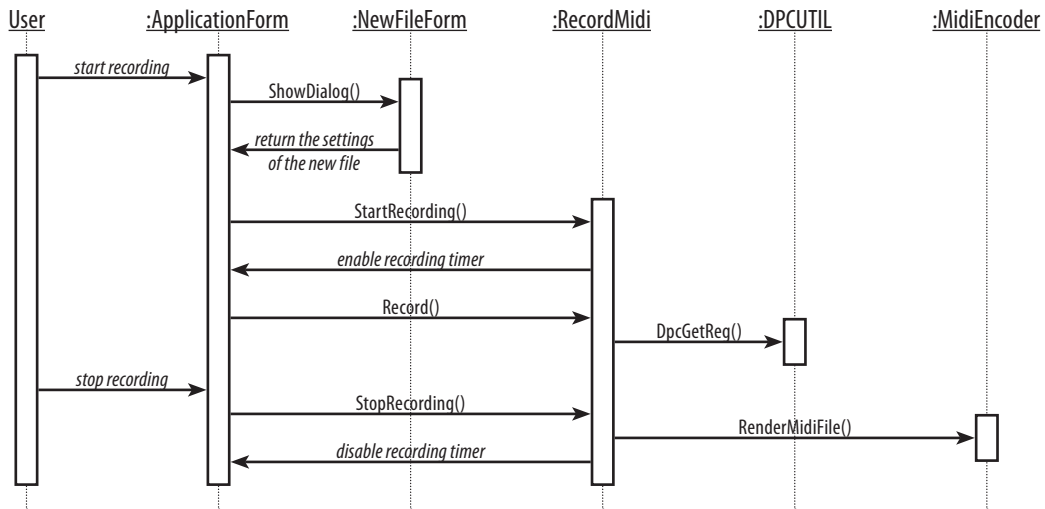


Figure 4.5. The sequence diagram for the “Record MIDI files” use case.

When the user presses the “Record” button of the toolbar, the application calls the “StartRecording()” method of the “RecordMidi” object. The application can now start the timer according to which the adapted MIDI messages are read. The “Record” button becomes the “Stop recording” button, which, if pressed, determines the application to call the “StopRecording()” method of the “RecordMidi” object. The timer is disabled and the board stops sending notes. The application renders the MIDI file and the displays the sheet in the main window.

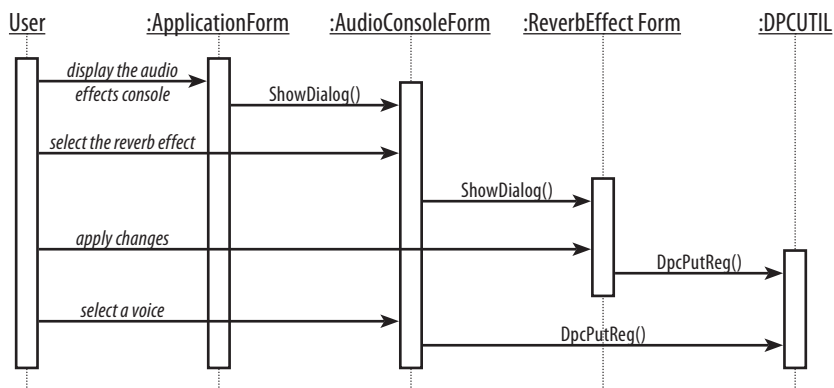


Figure 4.6. The sequence diagram for the “Apply audio effects” use case.

When the user selects “Audio Console...” in the main menu, the application displays a dialog box that allows selecting the desired voice or audio effect. If the user wants to apply a reverb effect, the application displays the corresponding window. The user applies the effect by writing certain information to the FPGA board. If the user wants to change the voice that the board is currently using, he or she presses the desired button and the application communicates the information to the FPGA device.

4.3.4 Class diagram

The following class diagram presents a simplified view of the classes and the relations between them.

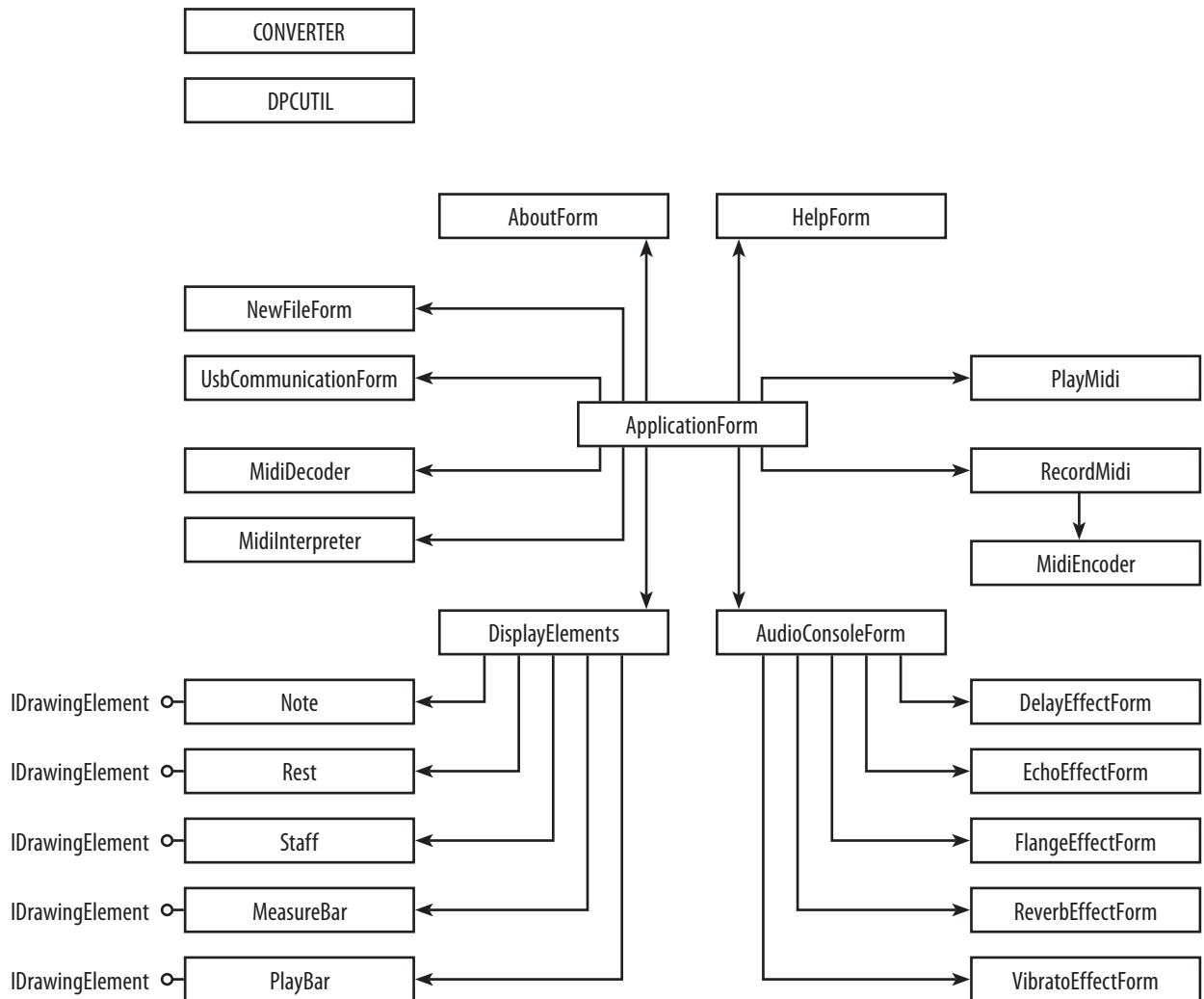


Figure 4.7. The simplified class diagram.

4.4 USB communication

The software application uses Digilent’s “dpcutil.dll” that contains the methods used for USB communication. These methods have been declared in the “DPCUTIL.cs” class. The software application calls certain methods of the dynamic library which communicates with the USB port of the computer. Data is sent through the cable to the connected FPGA board. The controller installed on the board handles the USB communication and offers the user a basic parallel interface.

The communication diagram is presented in the following figure.

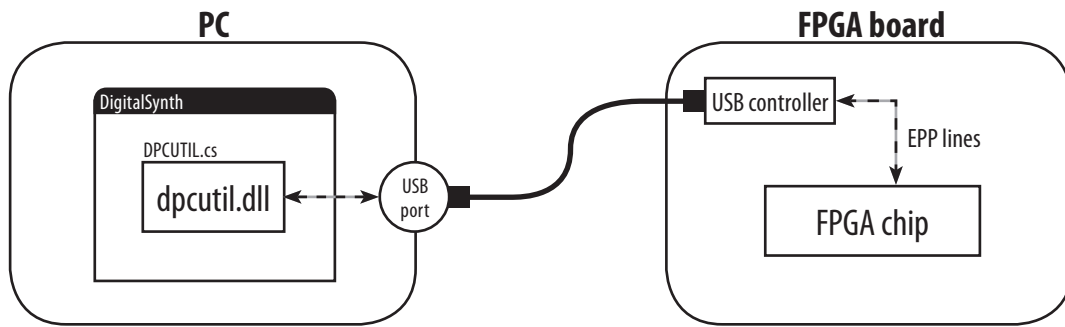


Figure 4.8. The USB communication.

4.4.1 Decoding and interpreting MIDI files

When the user opens a MIDI file, the software application reads the bytes of the file, determines the meta and MIDI events and creates a temporary file that will be used to determine which oscillators render which notes.

The application uses an array in which it stores the notes (the note value, the start time and the end time) and an array in which it stores the number of the slot (or oscillator) corresponding to the notes in the previous array.

In order to draw the music sheet, the application separates the notes in the note arrays according to the slot value from the second array. Each of the new note array will have its own staff (a set of horizontal lines) to be drawn on. The application determines the notes and the rests necessary for each track and then generates the graphical elements.

4.4.2 Playing a MIDI file

When the user plays a MIDI file on the FPGA board, the application uses a timer with an interval set to a quarter of the duration of the shortest note (sixteenth). This ensures that the file will be played more accurately. Also, the application uses a variable in which it stores the current play time (the position in the note array).

On every tick of the timer, the application increments the time and determines which notes from the note array have their start value equal to the time variable. Those notes are sent to the FPGA board at the corresponding registers, according to the oscillator number stored in the slot array. Also, these notes are stored in another array, and the application always checks to see which of these notes have their end time equal to the time variable. Rendering these notes will be stopped in case of equality.

4.4.3 Recording a MIDI file

When the user record a MIDI file from the FPGA board, the application uses a timer with an interval set to a quarter of the duration of the shortest note (sixteenth). This ensures that most of the events (note on, note off) are acknowledged. The application uses a time variable that determines the time at which the event occurred.

On every tick of the timer, the application reads the four registers that correspond to the four oscillators, determines the event type (note on, note off) and stores that event and the corresponding time. When the user stops recording, the application processes the arrays that contain the events and the times.

First, the arrays are “trimmed”, that is “note off “ events are deleted from the event array until the first “note on” event is found. Then the application determines the time difference between events (the delta time) and writes the resulting bytes in the new MIDI file.

After rendering, the application opens the new MIDI file and displays its music sheet. Due to the way it has been created, the music sheet will not be too accurate like in the case of MIDI files created in dedicated programs. The MIDI file can also be played in a program such as Winamp.

5. Discussion

5.1 Design issues

5.1.1 Computing sine values

The design of a component that generates sine values presents two options:

- using the CORDIC algorithm to compute the sine values,
- using a memory that stores the sine values.

The CORDIC algorithm can be implemented using either a serial or parallel architecture. Choosing one of the possibilities involves having to decide whether to sacrifice space (the parallel design requires a large number of registers, adders and a larger look-up table and resources demand increases for larger order algorithms) or time (the serial design requires using a small number of FPGA resources for a given number of iterations before obtaining the desired results).

A memory is capable of storing many values if the FPGA device provides that amount of space. The access time is very small, allowing the design to work well at 50 MHz.

Both of these options were considered, but the one that got to be implemented in the final version of the design is the second one. The reason is that the sine shape computed by the CORDIC algorithm is not close enough to the mathematical sine shape.

The values that the memory contains were computed using a C# program, thus obtaining a reasonable sine shape.

5.1.2 Generating basic waveforms with variable frequency

The component that generates basic waves is the “BasicOscillator” which allows the frequency of the generated sound to change only when the ADSR envelope process is idle. If the frequency changed during rendering, the generated sound would be the result of a different synthesis technique, called *frequency modulation*. The design would require a new component to control the frequency change (it could use a component similar to the one that realizes the ADSR enveloping). The resulting wave could still be ADSR enveloped.

5.1.3 Creating audio filters

The reverb effect unit uses audio samples that were generated at various moments in time and adds them together in order to form the reverb effect. If exponential decay is enabled, the samples are multiplied with certain coefficients to replicate the physical phenomenon.

The same component could be used as a FIR filter. The order of the filter (or the number of taps) could be set at a convenient value very easily. The reverb effect requires reading at a number of eight addresses in the circular buffer. Because samples are generated at a low frequency such as 48 kHz and the design operates at 50 MHz, a very large number of readings can be done between two writings. The read addresses could be consecutive (and generated by a counter) or however the user desires (and stored in a look-up table). Also, the decay coefficients could be used as the coefficients of the filter and the design would still use only one multiplier.

5.1.4 The quality of the sound

The design outputs 8-bit samples which correspond to a very low sound resolution. Also, the quality of the sound depends on the speaker used for rendering. If the speaker has built-in filters, the sound is greatly improved, but still doesn't match CD quality.

Although samples are generated at a large sample rate (48 kHz), the solution for a better sound would be using samples with a larger number of bits.

5.1.5 The music sheets

Musical notation is respected, except for tied notes, which are shown in the following figure.



Figure 5.1. (A) Standard musical notation for tied notes.
(B) Musical notation for tied notes used by the “DigitalSynth” application.

The music sheets generated for MIDI files created in dedicated programs (such as Geniesoft Overture) are accurate because the files are very strict (the events are synchronized, they start and end at precise times).

The music sheets generated for the MIDI files recorded from the FPGA board are not as accurate, because the events are not perfectly synchronized (the user presses and releases the keys at random times). Therefore, the “DigitalSynth” application is not capable of rendering perfect sheets in this case. Also, playing the same files on the board is not identical to what the user had recorded. The application decodes and interprets MIDI files and the result is used both for the musical sheet and for board playing.

If the user plays recorded MIDI files in a dedicated player such as Winamp, the result is identical to what the user had recorded, because the player does not interpret the files for sheet rendering; instead, it plays them as they are.

5.2 Design report

The hardware design has been developed on a Spartan3 FPGA chip. The synthesis report is presented in the following table.

| Resource | Available | Used | Utilization |
|-------------|-----------|------|-------------|
| Slices | 1,920 | 838 | 43% |
| Block RAMs | 12 | 9 | 75% |
| Multipliers | 12 | 8 | 66% |

Table 5.1. Device utilization.

The design process has been done on a Windows XP computer. The hardware project has been developed using Xilinx ISE WebPACK 8.1i and tested using ModelSim XE III, and the software project has been developed using Microsoft Visual C# 2005 Express Edition.

6. References

Digilent documents

- Digilent Adept Users Manual
[http://www.digilentinc.com/Data/Software/Adept/Adept Users Manual.pdf](http://www.digilentinc.com/Data/Software/Adept/Adept%20Users%20Manual.pdf)
- Digilent Port Communications Programmers Reference Manual
[http://www.digilentinc.com/Data/Software/Adept/DPCUTIL Programmers Reference Manual.pdf](http://www.digilentinc.com/Data/Software/Adept/DPCUTIL%20Programmers%20Reference%20Manual.pdf)
- Digilent Parallel Interface Model Reference Manual
[http://www.digilentinc.com/Data/Software/Adept/DpimRef programmers manual.pdf](http://www.digilentinc.com/Data/Software/Adept/DpimRef%20programmers%20manual.pdf)

Audio effects documents

- Harmony Central
<http://www.harmony-central.com/Effects/effects-explained.html>
- Wikipedia
http://en.wikipedia.org/wiki/Audio_effect

MIDI file format documents

- The Sonic Spot
<http://www.sonicspot.com/guide/midifiles.html>

Other documents

- Digital-to-analog converter
http://www.analog.com/UploadedFiles/Data_Sheets/AD7303.pdf
- Using Embedded Multipliers in Spartan-3 FPGAs
www.xilinx.com/bvdocs/appnotes/xapp467.pdf
- XST User Guide
toolbox.xilinx.com/docsan/xilinx7/books/docs/xst/xst.pdf



THE DILIGENT DESIGN CONTEST
Cluj-Napoca, October 7th, 2006