Warsaw University of Technology

FACULTY OF
ELECTRONICS AND INFORMATION TECHNOLOGY

Institute of Informatics

Bachelor's diploma thesis

in the field of study Computer Science
and specialisation Computer Systems and Networks

FPGA based hardware accelerator for musical synthesis for Linux system

Jakub Duchniewicz
student record book number 277132

thesis supervisor
dr hab. inż. Wojciech M. Zabołotny

WARSAW 2020

FPGA based hardware accelerator for musical synthesis for Linux system

Abstract. Work focuses on realizing audio synthesizer in a System on Chip, utilizing FPGA hardware resources. The resulting sound can be polyphonic and can be played directly by an analog connection and is returned to the Hard Processor System running Linux OS. It covers aspects of sound synthesis in hardware and writing Linux Device Drivers for communicating with the FPGA utilizing DMA. An optimal approach to synthesis is researched and assessed and LUT-based interpolation is asserted as the best choice for this project. A novel State Variable IIR Filter is implemented in Verilog and utilized. Four waveforms are synthesized: sine, square, sawtooth and triangle, and their switching can be done instantaneously. A sample mixer capable of spreading the overflowing amplitudes in phase is implemented. Linux Device Driver conforming to the ALSA standard is written and utilized as a soundcard capable of generating the sound of 24 bits precision at 96kHz sampling speed in real time. The system is extended with a simple GPIO analog sound output through 1 pin Sigma-Delta DAC.

Keywords: FPGA, Sound Synthesis, SoC, DMA, SVF

Sprzętowy syntezator muzyczny wykorzystujący FPGA dla systemu Linux

Streszczenie. Celem pracy jest realizacja syntezatora muzycznego na platformie SoC (System on Chip) z wykorzystaniem zasobów sprzętowych FPGA. Wymagana jest generacja polifonicznego dźwięku, który można odtworzyć w czasie rzeczywistym za pomocą przetwornika cyfrowo analogowego. Wygenerowany dźwięk jest także przekazywany do systemu Linux, pracującego na procesorze systemu SoC. W pracy poruszone są zagadnienia dotyczące sprzętowej syntezy muzycznej, tworzenia sterownika urządzenia dla systemu Linux oraz komunikacji z FPGA poprzez DMA. Po zbadaniu szeregu sposobów syntezy, zostaje wybrany system z interpolacją przy użyciu LUT. Zaprojektowany został nowatorski filtr zmiennych stanu o nieskończonej odpowiedzi impulsowej, który został zaimplementowany w języku Verilog a następnie użyty jako filtr dolnoprzepustowy. Syntezowane są cztery przebiegi akustyczne: sinusoidalny, kwadratowy, piłowy oraz trójkątny. Dla wzbogacenia efektów muzycznych możliwe jest ich błyskawiczne przełączanie. W celu zminimalizowania zniekształcenia dźwięku przy chwilowym przekroczeniu zakresu podczas sumowania próbek, zrealizowano eksperymentalny akumulator próbek, który rozkłada w czasie próbki o wartości wykraczającej poza zakres reprezentacji wyjściowej. Dzięki temu sterownikowi urządzenie jest widoczne w systemie Linux jako karta dźwiękowa, która jest w pełni zgodna ze standardem ALSA, oraz generuje dźwięk o rozdzielczości 24 bitów z częstotliwością próbkowania 96kHz. System jest rozszerzony za pomocą przetwornika cyfrowo-analogowego zaimplementowanego w układzie Sigma-Delta na jednym bicie GPIO.

Słowa kluczowe: FPGA, Synteza dźwięku, SoC, DMA, SVF

załącznik nr 10 do zarządzenia
nr 46 /2016 Rektora PW

Politechnika Warszawska
Warsaw University of Technology

…….....
miejscowość i data
place and date

……..
imię i nazwisko studenta
name and surname of the student
……..
numer albumu
student record book number
…….…….
kierunek studiów
field of study

OŚWIADCZENIE

DECLARATION

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.
Under the penalty of perjury, I hereby certify that I wrote my diploma thesis on my own, under the guidance of the thesis supervisor.

Jednocześnie oświadczam, że:
I also declare that:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- *this diploma thesis does not constitute infringement of copyright following the act of 4 February 1994 on copyright and related rights (Journal of Acts of 2006 no. 90, item 631 with further amendments) or personal rights protected under the civil law,*

- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- *the diploma thesis does not contain data or information acquired in an illegal way,*

- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- *the diploma thesis has never been the basis of any other official proceedings leading to the award of diplomas or professional degrees,*

- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- *all information included in the diploma thesis, derived from printed and electronic sources, has been documented with relevant references in the literature section,*

- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.
- *I am aware of the regulations at Warsaw University of Technology on management of copyright and related rights, industrial property rights and commercialisation.*

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.
I certify that the content of the printed version of the diploma thesis, the content of the electronic version of the diploma thesis (on a CD) and the content of the diploma thesis in the Archive of Diploma Theses (APD module) of the USOS system are identical.

.....
czytelny podpis studenta
legible signature of the student

Contents

1. Introduction

Since the popularization of a modular one by Robert Moog in the mid-60s, synthesizers have changed the way humans produce music tremendously. While at first analog, they soon became digital and dominated the music industry. Nevertheless, analog ones are still sought for their unique sound and because they can produce signals less deterministic than digital ones.

Because a synthesizer can imitate real instruments it is often used by professional and amateur musicians alike. Such a synthesizer often can be implemented solely in software, but when on-stage and performing live, an artist requires it to be at least partly embedded into the hardware. When realized in hardware, it can be composed of special Digital Signal Processing units, which are designed especially for this. However, for amateur realizations or scientific experiments such a solution is often too inflexible and in such cases implementing this in Programmable Logic is more appropriate. Field Programmable Gate Arrays are the most popular types of such devices, and they allow for very rapid prototyping of a hardware circuit written in a Hardware Description Language such as Verilog, SystemVerilog or VHDL.

The world of FPGA's is constantly evolving and the available materials soon become obsolete and the developer has to face a plethora of time-consuming challenges. Apart from them, they have to possess knowledge ranging from electronic circuit design, through digital and analog filtering knowledge, to Linux Device Drivers and Linux internals understanding. For this project especially, understanding signal processing was crucial because it utilizes a Numerically Controlled Oscillator for the waveform generation. Moreover, some degree of proficiency with high-speed digital circuits was required to implement PL pipelining.

1.1. Scope of Thesis

This project is meant to utilize System on a Chip that comprises FPGA to generate signals corresponding to a MIDI note read from the user, and process them in Hardware, before passing them to Software running a Linux Operating System. The combination of HPS and FPGA is desired because it allows for utilizing best of both worlds, parrallelism of the FPGA paired with the potential of the Linux on top of the HPS. Minimal requirement is a polyphonic sine wave oscillator capable of generating all MIDI notes. While various steps of the sound synthesis and processing may be offloaded to the FPGA to utilize its inherent parallelism, at least a minimal synthesizer should be implemented entirely in PL and then feed that sound to the Linux System via Direct Memory Access. After successful implementation of a sine wave synthesizer different waveforms can be added to even further push the boundaries of this SoC and provide more control to the user.

All the basic processing tasks needed to generate a sound successfully should be

offloaded to the FPGA, while still allowing for the reception of the sound on the Software side and further processing of received signal samples. Hence, generation, filtering and all of the other calculations which can be parallelized should be implemented in the PL while allowing the user to provide their DSP subsystems on the Linux side via the JACK audio system. The data received by the Linux side can be processed in real time or saved to a file. Saving to a file should be without any glitches or unwanted audible effects.

1.2. Thesis Layout

The next chapters of this thesis focus on first introducing concepts and history of sound synthesis, then briefly presenting the MIDI protocol. Next, overall system design is discussed, hardware choice and capabilities are explained, then the functional requirements of the system are stated. Last but not least, both the hardware and the software part are discussed in detail. Communication, synthesis, and effects generation is explained; some details and encountered problems are examined. Finally, testing and results are presented along with an overview of things that were enhanced after thorough testing of the system. The thesis is surmounted with a conclusion and acknowledgments.

1.3. Overview of existing solutions

The topic of creating a synthesizer using Programmable Logic is not one of much widespread eminence. Although various amateur projects are focusing on creating a working synthesizer, almost none of them try to make use of both the hardware and software parts to generate and process the sound. Three main components are usually discussed in scientific papers: synthesis techniques using the FPGA, DMA techniques, and writing an ALSA driver. On the day of writing this thesis, several projects are focusing on utilizing FPGA's for accelerating sound processing. However, none of them implements a working live stand-alone synthesizer which can be further enhanced on the software side.

At the time of writing this article, there seems to be just one work resembling mine. Two students of Worcester Polytechnic implemented synthesizer which utilized FPGA as the main synthesis and effects generator unit while offering Python GUI to the user.[1] The FPGA used there is much more powerful than the one I used in my project, however, was not pushed to the limit. The FPGA used in the cited project was also realized in a SoC system but utilizing the system just for reading the MIDI commands from the external keyboard. It is worth mentioning the extent to which their work implemented various musical effects, most of which are too expensive in terms of resources utilized for my choice of hardware.

1.3.1. Synthesis techniques

This topic is by far the most explored, as it is quite mature and the demand for efficient synthesis techniques is ever-growing. Numerous works focus solely on the ways the

FPGA may be used to accelerate the synthesis process, or where it is used as a standalone synthesizer from the bottom-up. A work by Jacek Borko et al.[2] focuses on the parametric additive synthesis of audio signals while maintaining a low power consumption. Other works solely utilize the FPGA for generating audio effects[3] or make use of the parallelism of the FPGA for converting mono sound on the input to a stereo on output.[4] There are also some course works focusing on teaching some aspects of the FPGA utilization and programming which present interesting approaches to the synthesis. One of such works is a project specification for a project at University of California at Berkeley, where wavetables with precalculated waveforms are sampled using the FPGA to generate sound.[5]

1.3.2. DMA techniques

Utilizing Direct Memory Access is ubiquitous is every high-speed data acquisition or processing system. Because of that numerous works are written on this topic, and it is incessantly developed. One such work presents several distinct techniques for such communication, among which one utilizes DDR as temporary buffers.[6] Because currently, effective utilization of Graphics Processing Units is a popular topic, many works are written regarding FPGA-GPU communication. Thankfully, many complex IP cores have been already written and tested, in turn allowing for robust and quick development of such a solution on the FPGA. One example of such IP core is the Altera mSGDMA IP Core which is documented in the manual[7] and additionally in a guide written by one of the users.[8]

1.3.3. Writing ALSA driver

This last subject is most poorly documented, offering outdated technical guides and almost no scientific work regarding it. Usually, ALSA device drivers are written by specialists, and judging by the scarce amount of such drivers in the Linux kernel git repository, virtually nobody needs to share this knowledge as every device has its specification. Nevertheless, there were some approaches for making it accessible[9], and detailed as this article is, it still leaves out a lot of crucial information. Some of it is complemented with the pages from the ALSA-project wiki page[10] and some of it has to be garnered by trial and error, or reading ALSA source code. Fortunately, there is a project which tried to even further formalize the steps required for writing a successful ALSA driver, but it again failed to convey the most crucial step - data transfer constraints when communicating with real devices.[11][12] Still, it remains the most up-to-date and relevant document regarding ALSA driver implementation and deserves more praise.

2. Background

Since the dawning of the human race, music was an indispensable part of our lives. Much can be said about the effects it has on human beings, ranging from motivating and energizing to relaxing and unwinding. The virtue of synthesizers is that they allow generating almost every conceivable sound a human being is capable of hearing, or even beyond that. Because of that, they are nowadays ubiquitous, being embedded in our mobile phones or even in smartwatches. Nevertheless, industry-grade synthesizers are still expensive enough to discourage their compulsive purchase and are a mix of both analog and digital domains.

Because their usage often does not require much specialized domain knowledge beyond some prior musical education, they found their way to almost every household in the form of a musical keyboard. Probably most of the young people were in touch with them at some point in their lives, and some of them use it daily. Synthesizers are an indispensable tool for amateur musicians and music producers because they allow for a broad range of instruments and effects for a low cost. Pairing them with a Digital Audio Workstation permits for enhancing their capabilities and even altering their primary functionality by reprogramming them. Such options are enabled in this project's Novation LaunchKey MINI, where every knob and touchpad can be reprogrammed on a whim of the potential user. Often buying a synthesizer is a much better choice for an amateur musician or a band than buying a specific physical instrument, especially when their theme music is related to electronic or techno. Their current fidelity is such high that sometimes one cannot tell the difference whether the instrument currently being played is generated by means of a synthesizer or a true musical instrument. Nowadays, there is a trend for returning to the origins of synthesis and the craftsmanship of analog synthesizers is abound. This project is somewhat a movement in this direction since it aims to show that sound synthesis is possible in yet another way on yet another hardware and software.

It leaves no doubt that synthesizers changed and improved the ways people create music, facilitating this process as a whole and most importantly allowing for very quick iterations resembling prototyping. Having an abundance of effects and sound banks, one can quickly find a sound that suits them without the need for re-tuning the instrument or even worse ordering a new one to be crafted by a specialist maker. With synthesizers, it takes just one knob turn or one keypress to change the whole sound domain in which the musician currently operates, which makes them much sought after item.

2.1. Origins of synthesis

It is transparent that with the passage of time people would become gradually interested in making the process of sound creation easier and more accessible in both financial terms and certain required skills. The popularization of electricity even further influenced

their rapid development. First in form of Theremin[13], Mellotron[14], and then Hammond's Organ[15], synthesizers soon became very popular throughout avant-garde music communities. The de-facto first synthesizer was conceived by Harry Olson and Herber Belar at the RCA laboratories in Princeton in 1957 - RCA Mark II. Because transistors were just being started to be mass-produced, it utilized 750 vacuum tubes instead of them. The next milestone in their development was the creation of a modular synthesizer by Robert Moog, who was an engineer and a physicist. It is worth noting that these synthesizers were all monophonic, and even though they offered various effects to the user, they could be just applied to the single sound being played. Only in 1978, with the release of Prophet-5 from Sequential Circuits, polyphony found its way to the synthesizer world. Moreover, it allowed for storing the sound for future replay, instead of tediously recreating it each time it was desired. Miniaturization of synthesizers was also a popular topic during these times

Figure 2.1. A Minimoog analog synthesizer.

- Minimoog was one of such creations and it is visible in figure 2.1.

By the 1980s, with the rise of MIDI and the conception of first digital synthesizers, they conquered the music market. Pop artists started using them for their performances which greatly influenced their reach. The most popular synthesizer models, being sold even today were developed during this time. Such creations as Yamaha DX7, Roland D-50, or the famous Korg M1 remain prominent examples of precursor digital synthesizers. In the subsequent years, synthesizers became miniaturized and even embedded in special DSP chips that we can find in almost every electronic device. With the passage of time, analog synthesizers became again sought after due to their vintage sound and feel, and probably in search of sounds different from these popular nowadays.

It cannot be settled whether it was the synthesizer that made bands like Pink Floyd or musicians such as David Bowie or Jean Michel Jarre popular, or it was the other way round. Nevertheless, they soon became indispensable mediums for progressive rock bands and after a few decades, the groundwork for electronic and techno music. Ranging from classical music, through pop and rock music, to electronic and hard techno, synthesizers have found their way to the contemporary musician and are now an indispensable tool of the trade. If not for them, some greatest musical pieces would not have been composed due to not having such types of sounds to choose from.

2.2. Synthesis approaches

While there are multiple ways to manage sound synthesis, the most prominent of them are a subtractive and additive synthesis. These two are historically the most important ways of achieving the goal of synthesis - generating an audible effect. Some other types include Frequency Modulation synthesis, Sample-Based synthesis, and Granular Synthesis. All of these do not stem from the idea of harmonics and periodicity, but instead, employ different techniques to achieve the same result.

When implementing signal synthesis in the digital domain, one has to take into account that the digital representation is inherently finite when compared to its analog form. Because of that, some special techniques have to be undertaken in order to preserve the highest accuracy and quality of the output signal. The most prominent of which is the concept of Direct Digital Synthesis, which is used in this project extensively.[16]

2.2.1. Subtractive synthesis

This is the type of synthesis that was most popular in analog synthesizers because it relies on just two components - harmonic-rich wave generator and a voltage-controlled LPF, the result of these can be then amplified and put through a mixer to achieve polyphony. The most popular wave shapes that are utilized in such synthesize are square, sawtooth, or triangle wave. These of course can be mixed to create even more unique and peculiar results at the final stage of synthesis. The lowpass filter may be a regular one or equipped with a resonance near the cutoff frequency to enhance the resultant sound. Of course, different types of filters may be also applied to filter out for example some special bands or notches in the spectrum. Filters used are usually 2-pole or 4-pole, meaning they have an attenuation of respectively 12 dB or 24 dB per octave. The signal leaving the filter can now be shaped using a special envelope or a simple amplifier. The most popular technique of shaping the envelope is called ADSR which stands for Attack Decay Sustain Release. With these parameters, one can control how long the sound rises, falls, is stable, and finally how long it takes until it disappears after the note is released. Finally, if one desires so they may add a Low Frequency Oscillator to create a vibrato or a sweep effect at the output of the synthesizer. In the end, the signal may be mixed with other signals for polyphonic

output. Wielding all these components, the synthesizer is powerful enough to meet the requirements of even the most demanding musicians.

2.2.2. Additive synthesis

Additive synthesis techniques can be traced back as far as the conception of harmonic wave analysis by Joseph Fourier in 1822. It states that every waveform can be composed of simple harmonic waveforms of different frequencies and is the groundwork behind some of today's most prominent technological advancements such as MRI or space exploration. It can be seen in the figure 2.2. The earliest synthesizers relied on this technique to gener-

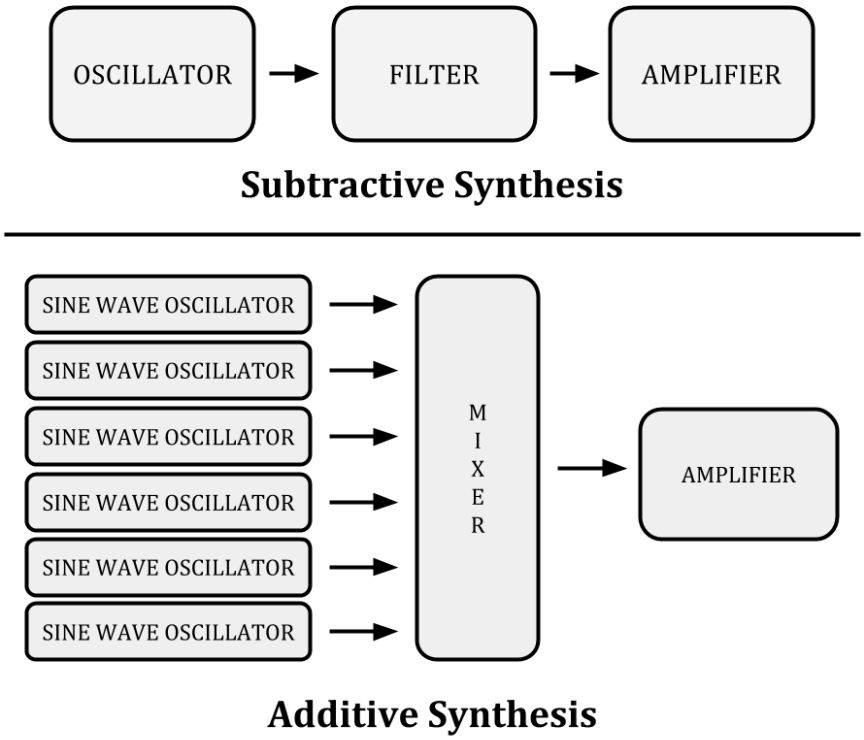

Figure 2.2. The difference between subtractive and additive synthesis techniques.

ate their unique sounds, such as the Hammond's Organ. In principle, the sinewaves of differing frequencies are mixed to achieve a waveform of much different timbre and shape. Such waveforms as square, triangle, or sawtooth are composed of multiple harmonics of a sinewave, hence they themselves are already results of additive synthesis. Similarly to the subtractive synthesizer, this one also makes use of contour generators and amplifiers to create its myriad sounds. A mixer is used here for every single monophonic sound in order to stack the harmonic waveforms on top of each other. One crucial component of the additive synthesizer is the noise generator, which makes some sounds (especially orchestral ones) sound more real. It is also a building block for a technique called resynthesis, which takes an input sound and tries to recreate the harmonic structure of it.

2.2.3. Other approaches

Apart from the two most popular types of synthesis, some additional are quite popular in the music industry. Frequency Modulation synthesis is one such technique, where via the usage of a modulator oscillator, the frequency of the original wave is changed to produce yet another one. The most prominent example of a synthesizer employing it was Yamaha DX7, which shaped the sound of the 1980s. This concept is fairly similar to the ordinary Frequency Modulation known from radio communication, except here the carrier modulation is in the audible range and allows for interesting results.

On the other hand, sample-based synthesis or wavetable synthesis is based on storing short parts of a periodic signal and running through them with either a slower or a faster pace to generate sounds of differing frequencies. They can be overlapped and combined in any desired way to create novel sound forms. This project utilizes a similar approach to generate the sound, with a quarter-sine wavetable and different tuning words that regulate the output sound frequency.

Granular synthesizers are an upgraded version of the sample-based ones. Their samples are even shorter, around 10-50ms of duration and they are incessantly combined in order to create a fine-grained waveform. They require quite a lot of processing power because of this and with the development of more advanced software, they were adapted in form of programming languages. Starting from mostly console-style ones like CSound, Chuck or Faust through more interactive sorts of Pure Data or SuperCollider, numerous languages have off-sprung from that synthesis technique. Most of them do not require much programming knowledge, and allow for achieving quick effects without real synthesizer hardware. Instead they utilize mostly the software resources of one's PC in conjunction with their soundcard.

2.3. MIDI protocol

Musical Instrument Digital Interface is a protocol used by all modern digital music devices, Digital Audio Workstations, and regular computers. It is important to understand that MIDI is not a musical file format, it is just a sequence of commands that MIDI-compliant devices can interpret and react to. Since 1983 it is an industry-standard protocol for all Musical devices. In 2020 MIDI 2.0 was introduced and brought some new capabilities like profile configuration, property exchange, and protocol negotiation. Because this protocol is still in its infancy, this project utilizes the older version of the protocol. There is also an alternative protocol called Open Sound Control, which is similar to MIDI but allegedly allows for more data types and utilizing symbolic paths instead of numerical addresses of the devices.

MIDI, as a set of commands sent from the generating device to the MIDI-compliant playback device, is utilizing commands encoded in a binary format. Each command is composed of three bytes, the first one being the control byte and the remaining two being

the parameter bytes. All bytes except the first one have 0 as MSB, effectively allowing up to 128 values to be set. The first byte informs the receiver what type of message is being sent and to which channel. It is worth noting that MIDI allows up to 16 channels - simultaneous instrument tracks. The two following bytes have a different meaning depending on the context in which they are sent. For note messages, they are consequently: the MIDI value of the note being turned on or off, and the velocity with which it should be played - loudness. For control messages: the controller number and the value assigned to the controller. There is also a pitch/bend sequence of bytes which is used for varying the pitch of currently pressed (or echoing) sound.[17]

3. Hardware overview and System Design

Choosing hardware and planning the overall system design is often a demanding task, especially for a project rooted in several distinct fields of study as this. One cannot underline the importance of proper platform choice which often heavily influences the pace at which the project progresses.

3.1. Hardware overview

For the project discussed, I needed a device that would offer both an FPGA and an embedded Linux system with preferably shared memory to make DMA transfers easier. Although there are various devices that offer both FPGA and a HPS capable of running Linux, I was constrained by budget and did not want to spend too much on a board which I would use just once if it turned out not suiting my liking. The obvious choice of the manufacturer arose: Xilinx or Intel. Both offered similar devices at a similar price: De0-Nano-SoC from Intel and slightly more expensive PYNQ-Z2 from Xilinx. The choice was not straightforward, as from a perspective PYNQ offered much more DSP slices which are crucial for my application and the Vivado suite is much better documented than Quartus II. Though the opinions vary, the Quartus suite is mostly error-free and allows for the rapid development of prototypes thanks to the broad library of University Programme IP Cores. The lack of rapid-recompile option in the free version of the program is most vexing because in this project compilation times took about 15 minutes, which when debugging a minor issue occurring only on hardware is quite much.

However, after reading numerous reviews and browsing through projects utilizing both of these boards I decided to opt for De0-Nano-SoC. Despite multiple issues with setting up the Quartus IDE, and missing or outdated quickstart's for SoC applications, I was able to quite quickly complete a mock project and blink a LED on it. The board chosen for this project is visible in figure 3.1.

The board features an Altera Cyclone V FPGA with over 20 000 logic elements, 4 Phase Locked Loops and 3 clocks. Moreover it provides the user with a HPS based on ARM

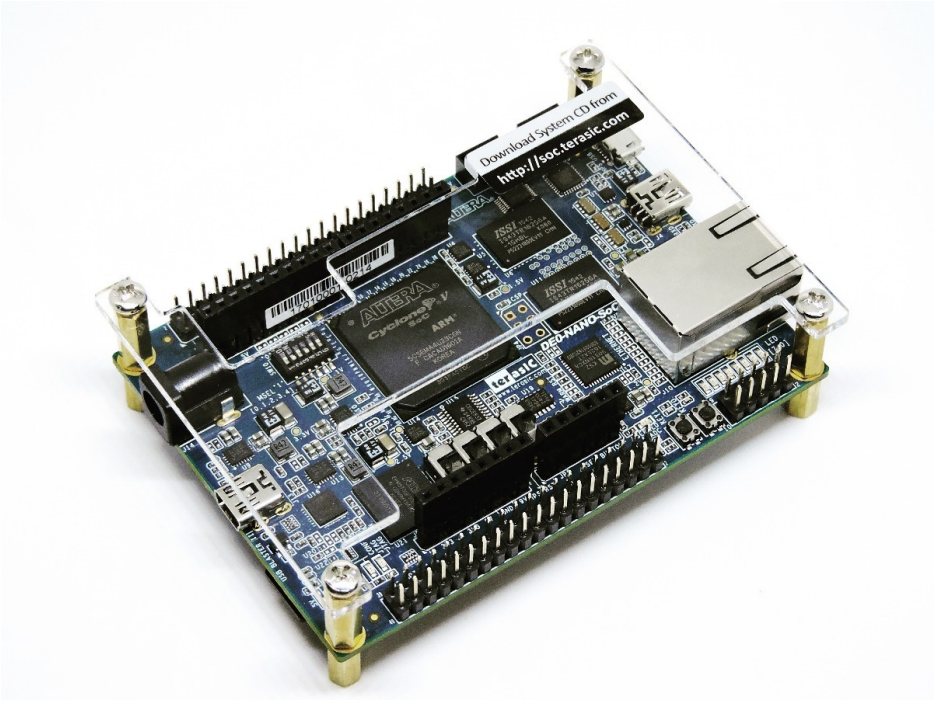

Figure 3.1. De0-Nano-SoC development board

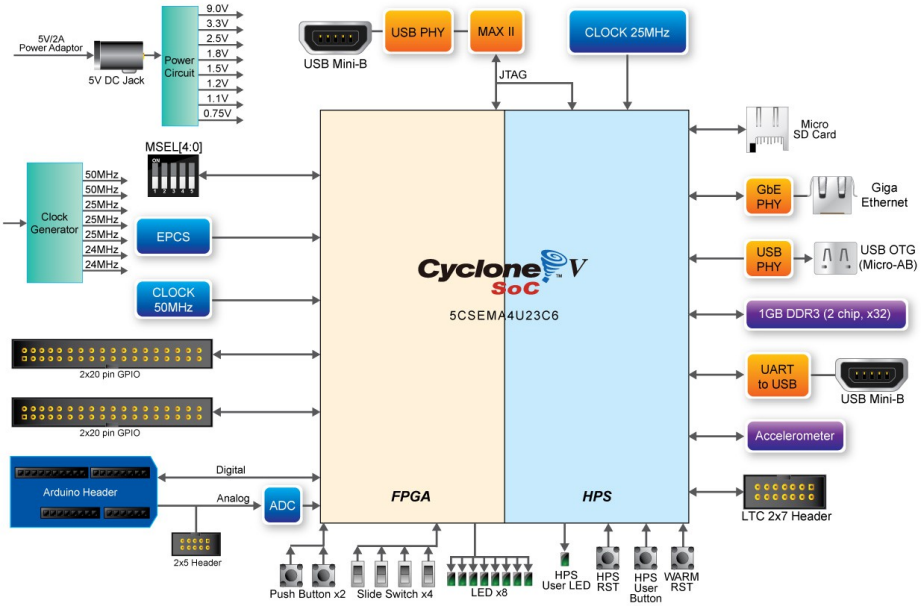

Figure 3.2. Schematic of the De0-Nano-SoC development board

Cortex-A9 Dual-Core processor working at 925MHz and 1GB DDR3 SDRAM which can be shared with the FPGA. It also provides some basic connections: GPIO's, Ethernet, USB connections and push-buttons. It can be visible in figure 3.2.

De0-Nano-SoC can be reconfigured by special settings of MSEL pins to utilize only FPGA, HPS, or both of them simultaneously. The on-board processor is capable of running any embedded Linux distribution, be it Yocto or custom Buildroot setup. It can also

support other operating systems, such as FreeRTOS. Overall, the board faced the challenge intrepidly and with some struggling all necessary components could fit into the restricted LE pool, which was much smaller than one in similar projects.

Figure 3.3. Novation Launchkey MINI keyboard

Moreover, for the MIDI compliant device, I have chosen the Launchkey MINI from Novation. It features 25-keys and allows tuning a few octaves up and down, effectively covering the whole MIDI range. It also supports note velocity detection allowing for taking this into account when synthesizing the requested sound. Moreover, it features up to 16 MIDI channels, hence it can act as multiple MIDI instruments at once. Several programmable knobs are also available, for example for pitch-bend control. It can be seen in Figure 3.3

3.2. Requirements Analysis

The system was designed in a particular task in mind: synthesizing a musical signal with the help of the FPGA and outputting the result to the Linux system running on the target. The user is presumably a musician or a hobbyist who would like to have a handy basic synthesizer capable of generating a requested musical note. Once the system is designed and flashed it would require minimal (if any) interaction from the layman user. The system can be extended with additional effects the user would like to implement themselves both in hardware and in software.

Apart from being simply generated, the signal has to be properly filtered by the use of a Low-Pass Filter. The device should allow for a polyphonic playback of requested notes, up to 10 simultaneously. Key presses and releases should be registered without any delay. The resulting sound can be recorded or routed to other applications on the user side in the Linux operating system. Additionally switching between different waveforms should be

possible and accessible for the user - preferably in a form of keyboard control push-button or knob.

There should be no audible delays between the keypress and its reception by the human ear. The system should strive for an as pure signal as possible, taking into account distortions incurred by imperfect passive components of the RC circuit.

3.3. System Design

From an overview, the system is composed of just a single board that comprises various distinct components. One such component is the FPGA which is responsible for signal generation, filtering, accumulating polyphonic samples, and passing them via DMA to the Linux OS. Because one of the project requirements is to allow for polyphonic sound generation, an important design decision had to be undertaken - how to generate multiple samples simultaneously. Obviously generating a separate synthesis lane for each of the 10 samples would be too costly, therefore a pipelined approach was conceived. Thus, a multi-cycle penalty is incurred, but since the sampling speed is much lower than the generating speed it is not a problem at all. The hardware side generates and stores the samples for the DMA transfers to the device driver implemented in software and at the same time outputs this generated sound to the delta-sigma DAC [18]. Generated Pulse Width Modulated signal leaves the board via 1 GPIO pin to a second order RC circuit. This allows us to immediately test the signal and observe any issues on an oscilloscope.

On the other hand, the Software part is tasked with reading the MIDI command from the user keyboard device, decoding it and sending an appropriate instruction to the FPGA. Moreover, the software part is responsible for communicating with the mSGDMA device implemented in the FPGA and receiving the buffers. A special ALSA sound-card is emulated by a device driver that receives the buffers via the DMA and feeds them to the ALSA system at appropriate intervals. This device can be used by, for instance, the JACK audio system[19] to even further extend the sound generation and processing pipeline.

A high-level overview of the system is visible in the figure 3.4.

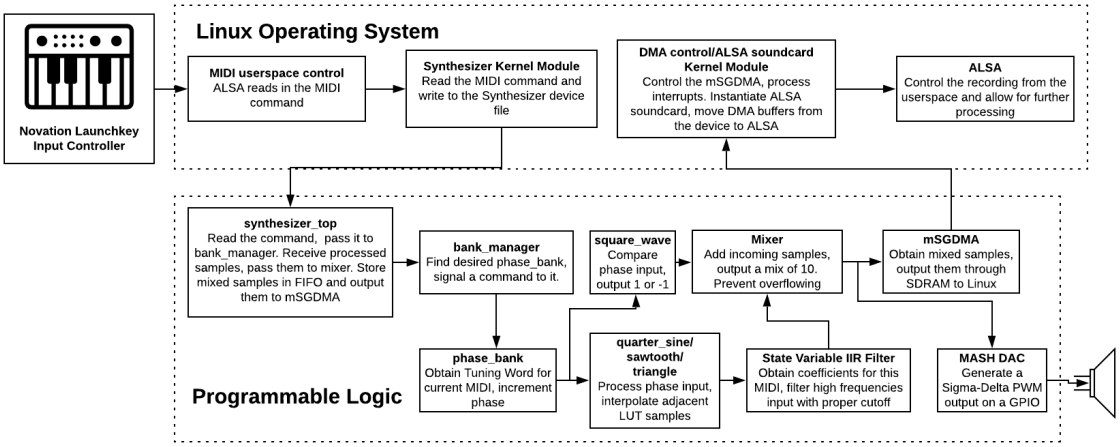

Figure 3.4. High-level overview of the system.

4. Hardware Design

The crucial part of the system - the signal generator is implemented in PL on the FPGA. Since DDS requires LUTs, FPGA was an ideal choice of architecture for this project. Moreover, because all DSP blocks usually run in a pipelined fashion, they share resources that otherwise would have to be copied for every additional DDS lane. This project allows for parallel processing of 10 sounds, but this requirement is purely based on the fact that humans usually do not play more than this number of notes simultaneously. The hardware clock runs at 100MHz, while the outputting sampling speed is only 96kHz, and usually no greater is needed for even high-quality sound processing. This allows for many-cycle delays between the sound is generated and actually played back. It is a convenience that is often utilized in projects such as this. Where the clock domains need to be crossed, a double-clocked FIFO IP core block is used, thus eliminating any glitches due to metastability issues. Unfortunately, the slow clock could not be generated using a DLL because there are not enough resources left to do so - its frequency is 96 kHz which would require fractional pipelined DLL's, and these are costly. Hence, the slow clock was implemented using a regular counter and a clock-enable signal which lasts just for a cycle of the fast clock. Because of this approach, Clock Domain Crossing was avoided, which is a common malady when designing logic systems and requires for example double latching and other synchronization mechanisms which deal with Metastability.

From a high-level overview, the system is an NCO with a preprocessing step, an IIR filter, an accumulator, and a sampling module. When the control is issued by the Linux kernel module, which inputs a control message to either play or stop a given note with a velocity corresponding to the force with which it was stricken, the 'synthesizer_top' module receives it via the Avalon-MM interface and signals the 'bank_manager' module which then handles its contents. The Bank Manager is tasked with marking which modules

are currently in use and which are free, additionally, it passes all intermediate results along the synthesis chain, such as sine LUT output to the IIR filter. When the signal is finally generated and filtered, it is the Bank Manager, that outputs this result. Handling this data by the Avalon-ST source is done in the top module of the Synthesizer IP Core, which in turn can be either output to a delta-sigma DAC or to the Linux OS running on the target. For outputting the signal to the OS, an mSGDMA IP core is used, which is then controlled on the software side by a Linux Device Driver.

The wave-changing logic is also implemented in the 'bank_manager' module, where it changes the currently output waveform instantly. There may be a few cycle residue in the SVF because of that but it is unnoticeable by the human ear.

	phase_bank	quarter_sine				state_variable_filter		
cycle	TW LUT	Negation + Symmetry	Sine LUT	Interpolation multiplication	Interpolation	Coefficients LUT	Second cycle	Third cycle
1	T	T - 1						
2	T + 1	T	T - 1					
3		T + 1	T	T - 1				
4			T + 1	T	T - 1			
5				T + 1	T	T - 1		
6					T + 1	T	T - 1	
7						T + 1	T	T - 1
8							T + 1	T

Figure 4.1. A view of pipelining.

The pipelining approach can be seen in the figure 4.1, T is the current sample, T-1 and T+1 are previous and next samples respectively. It can be observed that every pipeline element is constantly working even if the data is not valid in order to maintain pipelining efficiency.

4.1. Sine wave generation

The core of every synthesis - generator is implemented in a way facilitating both resource usage and ease of modification. This generation scheme is visible in 4.2. The

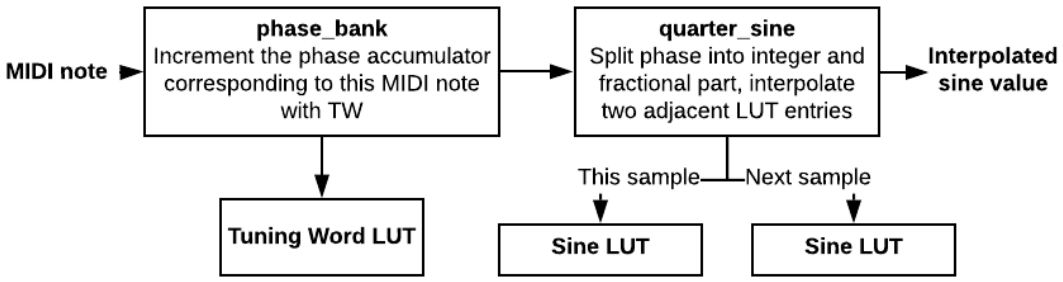

Figure 4.2. Sine generation overview.

groundwork behind such a solution is a phase accumulator and a phase step - the Tuning Word. This tuning word is also obtained from a LUT covering phase increments corre-

sponding to the desired MIDI note. The equation describing the mapping of MIDI note to the frequency is [20]

$$f_{note} = 2^{\frac{(m-69)}{12}} 440 \tag{1}$$

. These Tuning Words are pregenerated with a python script which takes into account the necessary output width, the sampling speed, and the desired frequency of the output signal. Again, the relation adheres to a certain mathematical relation

$$tw = \frac{f_{midi} * 2^{tw_{bits}}}{f_s} \tag{2}$$

. This allows for a flexible stepping through the sine wave LUT for each of the MIDI notes at a low memory cost of just 128 - 24-bit values in case of my solution. Since there are ten phase banks that are stepped through 1-by-1, the effective generator frequency for each of them is 10MHz instead of 100MHz, i.e. they are incremented one in ten cycles of the fast clock. It is important to note that the note sampling frequency has to be specified in the tuning word calculation - 96kHz in this case. The phase counter is an unsigned register of 24-bits of width, therefore it naturally overflows which is the desired effect when implementing a DDS system. The resulting phase is then passed to the sine module, which does the rest of the generation utilizing a sine LUT. It is of course possible to have much greater precision of Tuning Words but this requires a bigger phase register and some changes in the sine generation module, therefore it was unfeasible to further increase its size once 24-bits proved to be enough.

The sine LUTs are of the size of the first quarter of a full sine period because it is a twofold symmetric function and can be easily generated with just a quarter of a period of values at hand. This does not mean such a trade-off is without its consequences - the whole system is delayed by three cycles only due to the sine wave generation and interpretation of the input phase. The diagram explaining the above symmetry is visible in

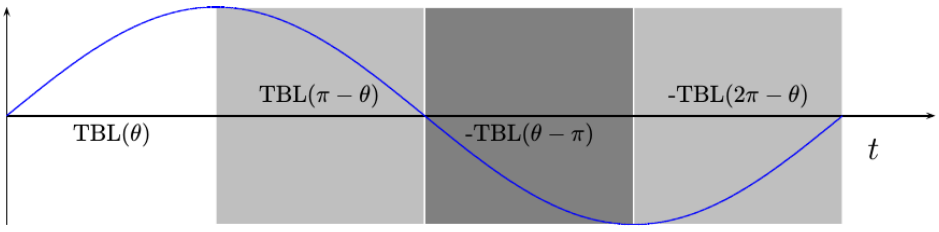

Figure 4.3. Sine wave symmetry.

4.3. The first symmetry can be observed along with the MSB - the negative symmetry. The second one is the 2nd MSB, which is the rise-fall symmetry, which even further reduces the size of the required LUT. [21] In order to prevent minor glitches in the sine wave size, it is shifted left by 1 sample. In effect, two bits of the incoming phase are utilized just for the symmetry resolution which allows for the generation of a smaller LUT by an order of 4.

Such a solution is quite common in both simple and advanced synthesis systems, because utilizing floating point mathematical functions is often quite costly and in some cases impossible to implement on hardware. If one wished for even greater performance, they could use CORDIC [22], which even further reduces resource utilization on a system like the FPGA.

Initially, such a solution was enough to generate a high-quality, smooth sine wave on output. However, the resource requirement for a high-quality audio wave is huge, assuming just the 16 bits of phase input to this LUT. Obviously, storing a LUT of the size of 2^{24} is unfeasible, as it would occupy much precious LE's which can be used for much more robust tasks. Because of that, a less resource-heavy way of generating sine wave samples was sought and implemented - linear interpolation of the accumulated phase between two adjacent sine wave LUT entries. It comprises two sine LUTs which are then interpolated at a step of 8192 units to attain the 24-bit precision of the whole system. The interpolation is linear, hence previous and next samples are multiplied by the distances from the current phase value. The results of this multiplication are added and they are the resultant sine value. Though this step requires an additional block of multipliers and a cycle is lost to compute the result, it is much more resource-efficient than the approach without interpolation.

Measuring these two distinct generation ways yielded different Signal to Noise Ratios, hinting at the interpolation's superiority. Because the digital quality of the generated sound is more important than the analog one, moreover the analog DAC is not meant to produce the highest possible quality, the SNR is defined for the digital fixed-point integers as follows

$$SNR_{dB} = 20 * \log_{10} \frac{signal_{lvl}}{noise_{lvl}} \tag{3}$$

The non-interpolated signal achieved the SNR of slightly less than 60dB and the interpolated over 75dB which is a not bad result for taking into account the non-ideality of the interpolation technique. Hence, the interpolation technique in this case proved to be superior to the other in terms of both resultant signal and resource utilization.

4.2. Square wave generation

The square wave is generated without the help of any LUT as it would be non-efficient in terms of resource utilization. Instead, it compares the input phase against a constant representing half of the phase and outputs either a logical -1 or 1 represented by the 2's complement 24-bit value of the corresponding magnitude. Because it is a square-wave and is composed of many periodic signals, the SVF exploited some unexpected behavior with overflows and this signal bypasses the filtering stage. The output signal is arithmetically shifted right by one to equal its magnitude to other signals, like sine or sawtooth.

4.3. Sawtooth wave generation

Similarly to the sine and triangle waves, this waveform uses a LUT for its sample generation. Instead of a quarter-wave, it utilizes a half period LUT which is shifted up for the second part of its period. Instead of 512 LUT entries, twice as that is used to obtain the exact same precision as for other waveforms. The usual 4-cycle latency is also maintained here.

4.4. Triangle wave generation

This waveform is implemented in exactly the same way as the sine wave, except for the LUT which is a rising slope. Again it is only a quarter of wave which is then used twice for interpolation and is outputted after 4 cycles.

4.5. State Variable Filter

As much as important as the generator, this module is especially vital in an audio signal generation because of high sampling speed and quite low frequencies of the generated signal. The LPF prevents aliasing and harmonic occurrences of the generated signal, therefore making it an ideal candidate for this type of solution. Moreover, this particular type of filter boasts its variable cutoff frequency, which makes it an even better candidate for filtering out all unwanted frequency components depending on the MIDI note stricken. If this was an analog synthesizer, such filters as an ordinary transistor-ladder in a Moog style or a Sallen-Key filter would be a perfect choice. But since this is a digital synthesis project, a different solution is necessary, and thankfully DSP and Audio companies incessantly develop new types and variations of digital filters. What is more, since such filters usually are used in embedded solutions and special DSP processors, they are highly optimized and come in many types.

Initially, such a filter was conceived by Hal Chamberlin and was named a state-variable filter [23], and even though it was an excellent filter implementation back then, it had one galling trait - limited frequency control range. Hopefully, recently Andrew Simper from Cytomic developed a filter based on it which used trapezoidal integration to overcome the aforementioned issue [24]. Implementing this filter in Verilog proved to be challenging for me, as this was the first time I developed a digital filter in this language and for an FPGA.

The filter coefficients were precalculated with a python script as usual and placed in a LUT in a Q2.37 notation because of high precision requirements. This extended precision is maintained throughout whole filter calculations and is truncated only for filter output in order to maintain as much stability and accuracy as possible. This LUT is also not very large in size, and although it could be replaced with an in-system multiplication and division, it would incur unwanted delays and probably a precision loss. For this module, four multipliers are needed to be synthesized and they run in a pipelined fashion like the rest of this project. In just three cycles, the input signal is filtered and is available for

output, which is much faster than it would be possible with the fastest FIR filter. In my implementation, I used the abbreviations in step with these of Andrew's to make it more accessible to a potential reader or implementer.

It is important to note that when numbers are expressed in the Q format, they require special scaling for bringing them back to the ordinary binary notation. For the multiplication result, this scaling has to be performed twice - once for each multiplicand. Forgetting to do so will result in a totally jumbled output signal and a faulty filter. When truncating the output signal, special care needs to be taken in maintaining the sign of this result. Therefore, the MSB and bits from 36 to 14 inclusive are taken as the output signal. This way two bits which are necessary for a precise calculation of fixed-point numbers are skipped and the result is correct.

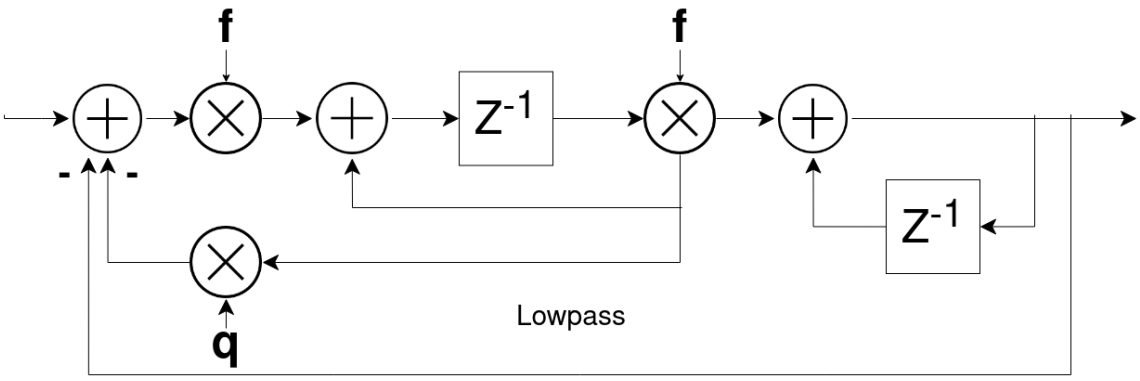

Figure 4.4. A Lowpass State Variable Filter block diagram.

A diagram showing this particular SVF operation is visible in figure 4.4.

4.6. Sample accumulator

When the sample is finally generated and filtered, it is sent to the sample accumulator which is tasked with mixing the incoming samples from up to 10 synthesis lanes and outputting the result to the FIFO. In order to prevent overflowing of the value, several measures are undertaken and implemented. First of all, the incoming value is scaled down by half by a single arithmetic shift right. Next, when the incoming values are added together, they are compared against MIN and MAX value possible to achieve with signed 24 bits signals, and if after summing they are greater, the overflow buffer is filled with the overflowed amount. The overflow buffer is appropriately sized, to prevent even further overflows and it is unloaded whenever possible. If in one cycle the value starts to overflow and in the next, it is decreasing, the overflow buffer is gradually discharged, until it is empty and the output value is again equal to the actual sum of the inputs. This solution incurs a slight phase delay but prevents loss of precision which is crucial in high-resolution systems like this.

Because it works on a 10-cycle basis, the last cycle has to be treated slightly differently and the code for this is therefore lengthy. To achieve slightly less precision with a simpler implementation, one could ignore more than one cycle of phase delay and just output as much of the overflow buffer in the sample succeeding the overflowed one, ignoring the rest and clearing the overflow register immediately.

5. Software Design

The software part of the project comprises most importantly the Linux OS which was configured to support MIDI, ALSA, and JACK. Additionally, two main parts can be distinguished: MIDI decoder userspace application and a special Linux kernel driver to support receiving DMA buffers and feeding them to the ALSA system. Configuration of the system plays a major role in making this possible and is also described in much detail. Since the FPGA is a real device, it has to be recognized by the operating system and registered with the Linux platform system by a special driver. Even though there is much common knowledge regarding writing such a driver, fine-tuning communication parameters, and designing an efficient communication scheme proved to be demanding.

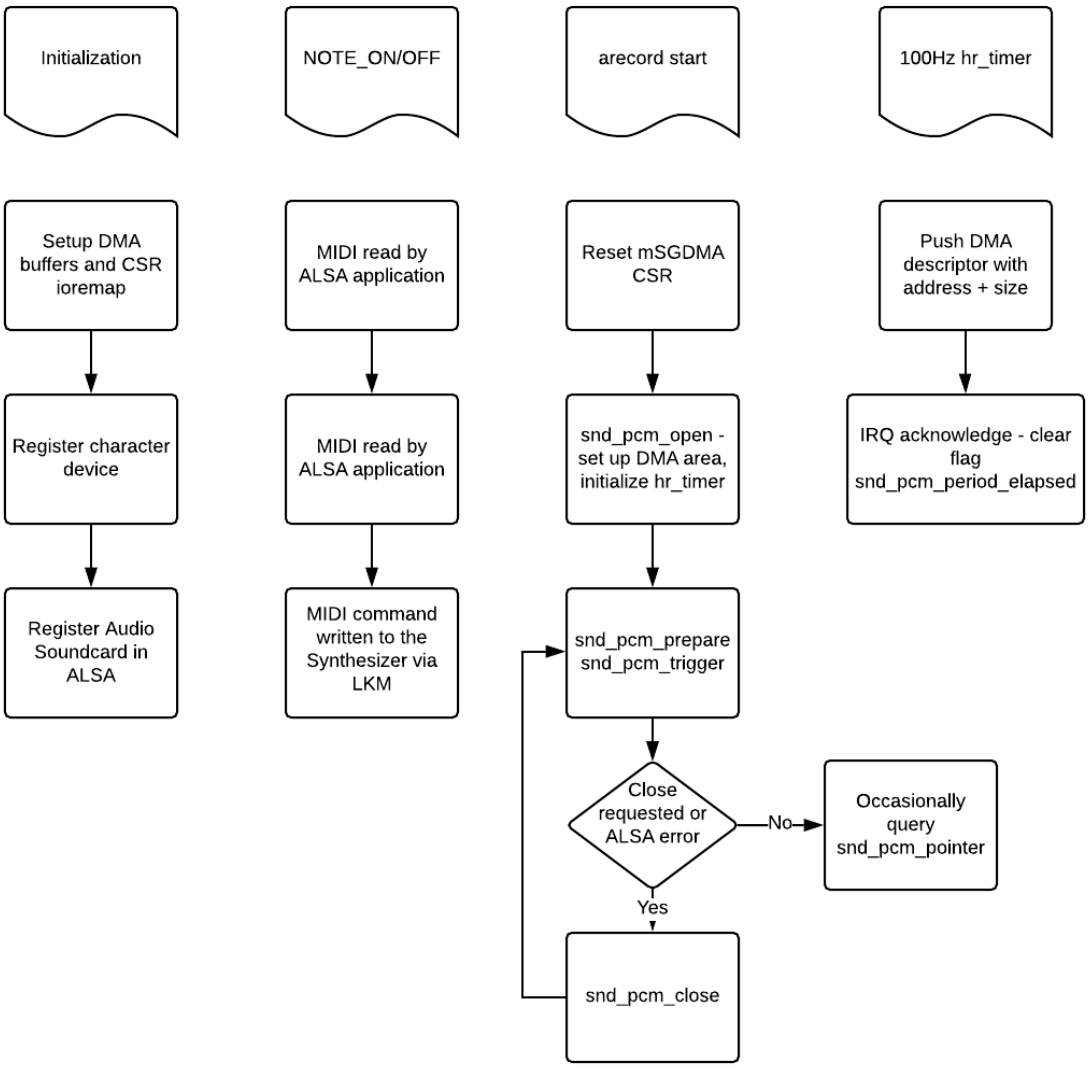

Figure 5.1. A software operation flowchart.

The logic flow of the software is visible in figure 5.1. Three major operating events

are visible: striking a MIDI note, commencing recording, and the high-resolution timer routine. It is worth mentioning that the recording logic flow is commenced by the arecord process and is regulated by ALSA, hence some function calls are subject to race conditions and in such places, a mutex is used to synchronize the concurrent access.

5.1. Board Setup

First and foremost, since a special hardware device was created and had to be visible in the operating system, a new entry in the Device Tree Structure had to be populated. When the hardware design is completed, Quartus scripts can be utilized to generate the new DTS, which can be then provided to the bootloader and kernel. Next, the uBoot bootloader utilizes this DTS for itself and SPL generation so that proper hardware components are initialized at boot-time before Linux is loaded. Because some desired parameters and DTS entries are not properly exported by Quartus, they have to be moved manually - this is particularly relevant to custom clocks synthesized with the use of PLL's. Moreover, when designing custom IP cores, one has to export several parameters manually to a _hw.tcl file, in order to add them to the generated DTS. Most of the commands and descriptions of this process in more detail can be found at dedicated wiki page[25] and thanks to quite dated but still valuable guide[26]. During the process of finding the proper and up-to-date way of setting up the board, I created a document summing up all these steps.[27]

Intel provides their own fork of Linux kernel - linux-socfpga[28], which is slightly modified to accommodate for some of their architectural differences and is lagging behind the development of the primary repository. Linux kernel version 4.14-130ltsi was used for this project as it was the latest version marked as long-term supported at the conception of this project. Apart from enabling ALSA support, ext4 filesystem, Loadable Kernel Modules, enabling FPGA Bridge and Manager, and setting the system type to Altera SOCFPGA Family no other changes are necessary in order to have kernel capable of working with this project.

The root filesystem used for this project was build using the Buildroot build system and needed only installing desired ALSA-lib applications such as aseqdump or arecord. In order to have a smooth project flow, I set up the Ethernet connection and transferred all necessary data via scp. There are few additional steps required for setting it up successfully, such as setting up sshd and generating enough entropy (which is scarce on embedded systems) with haveged. The scripts which perform these steps are available in the Appendices section with all other scripts.

When any Loadable Kernel Modules are built, they have to be installed to the target kernel manually by specifying it in the make command. Moreover, LKM have to be enabled in the target kernel in order to support them. With these steps completed, the initial SD card with the system image has to be flashed via a special utility script designed by Intel

and available for download. The card has to be only reflashed when a change to the kernel or bootloader is required, any other change made to the rootfs can be made via Ethernet.

5.2. MIDI receiving application

This is the first component that starts the processing chain, here the MIDI command is read from the ALSA system where it is provided by built-in ALSA kernel MIDI driver which supports the Launchpad keyboard. The command is decoded and reformatted to suit the internal format of messages for the FPGA synthesizer. Once formatted, it is written to the character device file of the memory-mapped Avalon Slave and received on the Hardware side. It supports the following events: NOTE_ON, NOTE_OFF, WAVE_CHANGE, and a special command for turning off all the notes at once. The application supports reading on different MIDI ports, thus can be easily extended to support additional devices if the hardware allows for more USB connections or virtual MIDI generators.

The feeder application is paired with a kernel module whose only responsibility is the registration of the device file in sysfs and allowing writing into it, in turn transmitting the command to the FPGA. It writes on a word basis every time the userspace requests a write and the device is successfully registered.

Table 5.1. MIDI userspace command

Bits	32 - 16	15	14-8	7-0
Function	Unused	ON/OFF	MIDI note	velocity

The API used in this case is a simple *write* syscall which writes a single 32-bit word with a command composed of a truncated MIDI command visible in the table 5.1.

5.3. DMA - ALSA synthesizer driver

Once the data is synthesized at the FPGA side, it is then sampled with a clock running at 96 kHz and submitted to the mSGDMA for transfer to the Linux driver. It is responsible for registering a soundcard within the ALSA system and performing DMA transactions for the data from the FPGA. When the application uses ALSA API for recording this data it uses a ring buffer, hence the driver has to follow the same fashion and allow for enough buffers so that buffer overruns do not occur. Each time a transaction is completed, an interrupt is fired from the DMA controller and the driver responds by informing ALSA that a recording period has elapsed. Because ALSA mostly uses a ‘frame‘ terminology, which stands for one sample size taking into account the number of channels and data width, conversions from bytes are required. The format used is S32_LE which is a 32-bit value, as the recorded file memory sizes are not of much relevance and this way the communication is faster. Even though the signal is 24-bit Little Endian it is stored as a 32-bit value, because of that, it can be played by most of the common audio players like VLC or Audacity.

At the time of writing this paper, the driver performs interrupts every 10ms thus requiring a high-resolution Linux clock, otherwise, jitter and stalling occur heavily disturbing the driver output. The device is controlled with special registers that are memory-mapped and controlled with control words.

5.4. Obtaining the data

The userspace can make use of the data that the soundcard receives and for example use a program like arecord or audacity to capture its output. Currently, just a single format and sampling frequency is supported, which is caused by these parameters being fixed at the Hardware side. The data is also monophonic but multiple channels can be added as an extension, this would require support from the hardware side to either output the data in an interleaved format or in any other. Because the ext4 system in conjunction with the SD card is incapable of real-time audio processing, the recorded file is first saved to RAM in /tmp/ and then moved to the regular SD card.

6. Testing

Testing a system, especially a complex one, requires particular attention to detail throughout the whole design process. Starting from a simple Verilog module or a C language function, through independent testing of PL and the Linux kernel driver, to holistic system tests covering the whole project. While hardware can be tested with the help of standardized Verilog test benches and then plotted in ModelSIM, some parts have to be tested in ingenious ways. One such part is the DMA communication and observing the actual output from the 1 pin GPIO DAC. In this chapter are presented various methods used throughout the project to assess whether it meets the requirements.

6.1. Verilog Testbenches

The test-bench setup tested only the internals of the system, not the Avalon-MM and Avalon-ST external communication protocols because simulating them by hand would be very time-consuming. The ModelSIM simulation software was used to display the signals changing in time and to properly observe, understand, and then fix the faulty logic. Because two distinct clock domains are crossed in this project, debugging it proved to be quite challenging, especially with the SignalTap software, which requires onboard memory for storing real samples of current-cycle logic.

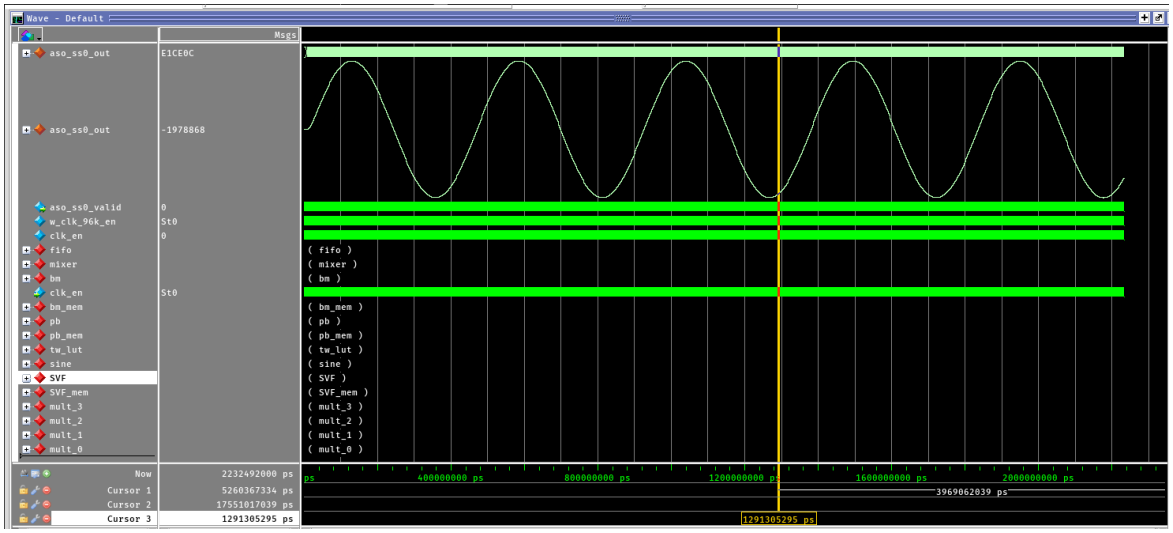

Figure 6.1. ModelSIM simulation view of E4 note.

A view of ModelSIM simulation waveforms can be found in figure 6.1.

Each component was tested using a test-bench setup containing all the necessary components, and short-circuiting the pipeline whenever one of them was not being tested. This in turn allowed for testing more complex scenarios, like turning notes on 1 by 1, or displaying all pipelined values simultaneously. This allowed for quick elimination of some rudimentary issues and testing whether the logic was proper. Unfortunately, ModelSIM

cannot behave equally to the real hardware, hence some errors slipped through this phase of testing. Most prominent of them are uninitialized variables, which ModelSIM initializes on its own and timing issues, especially with the double-clocking.

After testing in this fashion the device was programmed using SignalTap and proof checked whether simulated values occur on the device. Long compilation times and much more complex user controls did not appeal to me. Hence, I tested in this fashion only to observe whether something is occurring and debugged the logic back again in the simulator. SignalTap allows for quick reflashing of the debugging FPGA code using a JTAG connection. Once the PL behaved as predicted it could be flashed as .rbf into the SD card image.

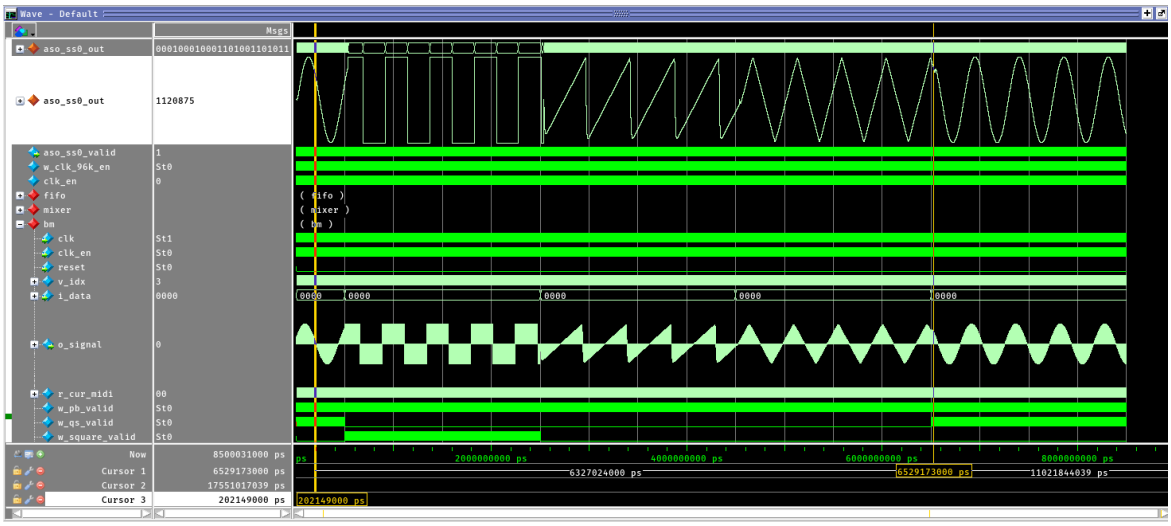

Figure 6.2. ModelSIM simulation of all 4 waveforms.

Some additional results of the ModelSIM displaying various waveforms are visible in 6.2.

6.2. Oscilloscope testing

The 1 pin GPIO connection was utilized to output the signal to an RC circuit which then filtered out most high-frequency components that may appear in the spectrum. Even though the signal is perfectly audible in the headphones connected by the audiojack, and the RC components were chosen to provide cutoff frequency of 15,9kHz, not all noise was filtered out. After spending much time trying to achieve as noise-free signal as possible, I deduced what might be the cause of these noises - a DC-DC step-down converter(LTC3612), which operates at 4MHz. The periodic noise and its amplitude is visible in figure 6.3. Moving the RC circuit closer and further from the DC-DC converter aligns with my speculations, when closer the amplitude of the noise is much larger, when further it remains around 140mV. The headphone connection allowed for checking whether the sound being generated actually matches the MIDI note. Testing this way allowed

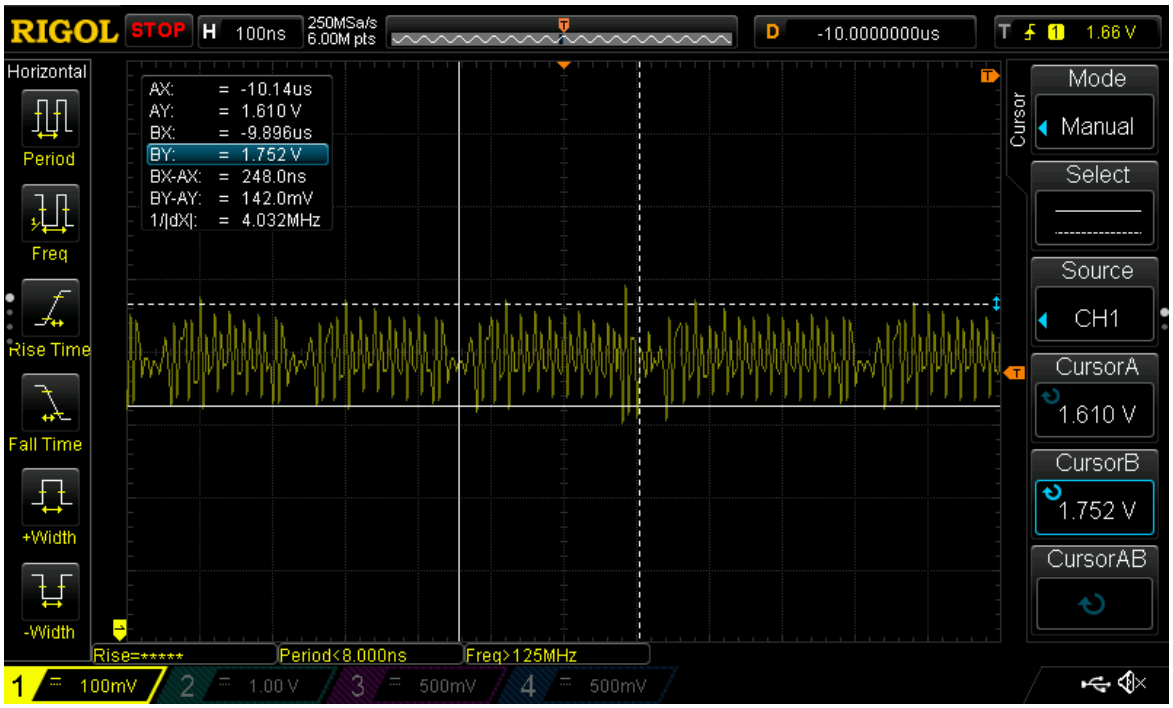

Figure 6.3. Periodic noise of 4MHz frequency.

to fine-tune the Tuning Word table and discover previously undiscovered errors in the project, effectively shifting MIDI notes back to their proper octaves.

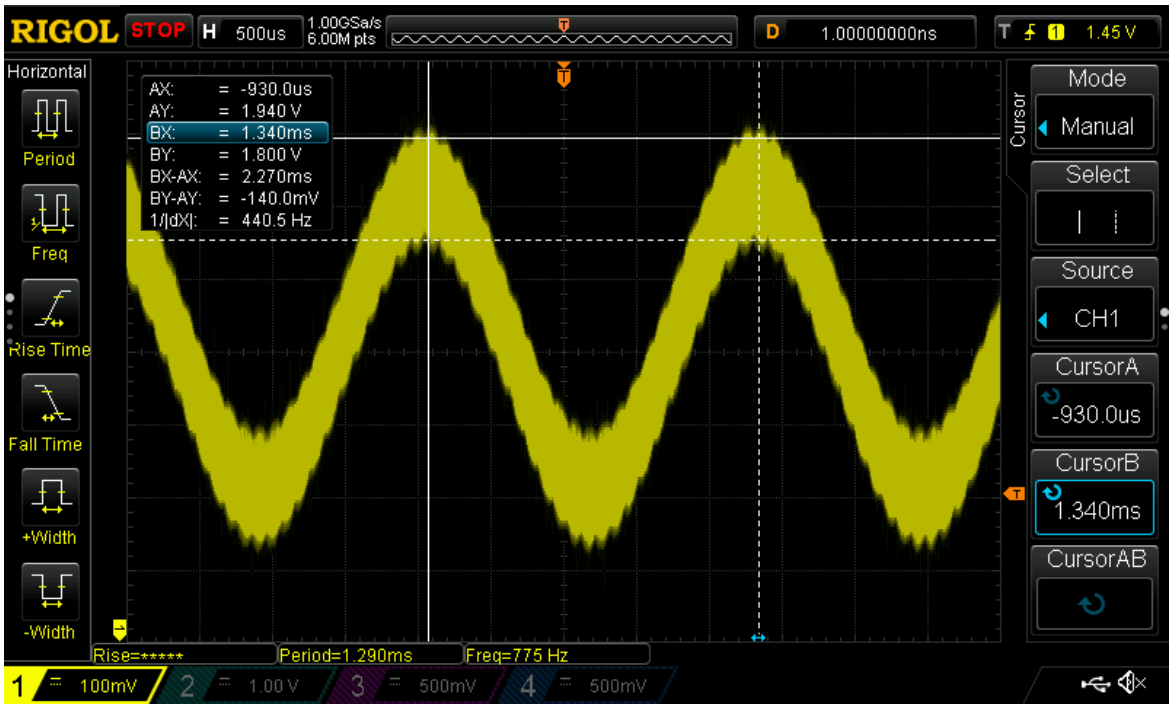

Figure 6.4. Sine wave DDS waveform for 440Hz

Figures 6.4 - 6.7 show sine, square, sawtooth and triangle waveforms the note A4 - MIDI 69 - 440Hz. As mentioned above, it is not filtered out completely, nevertheless, the

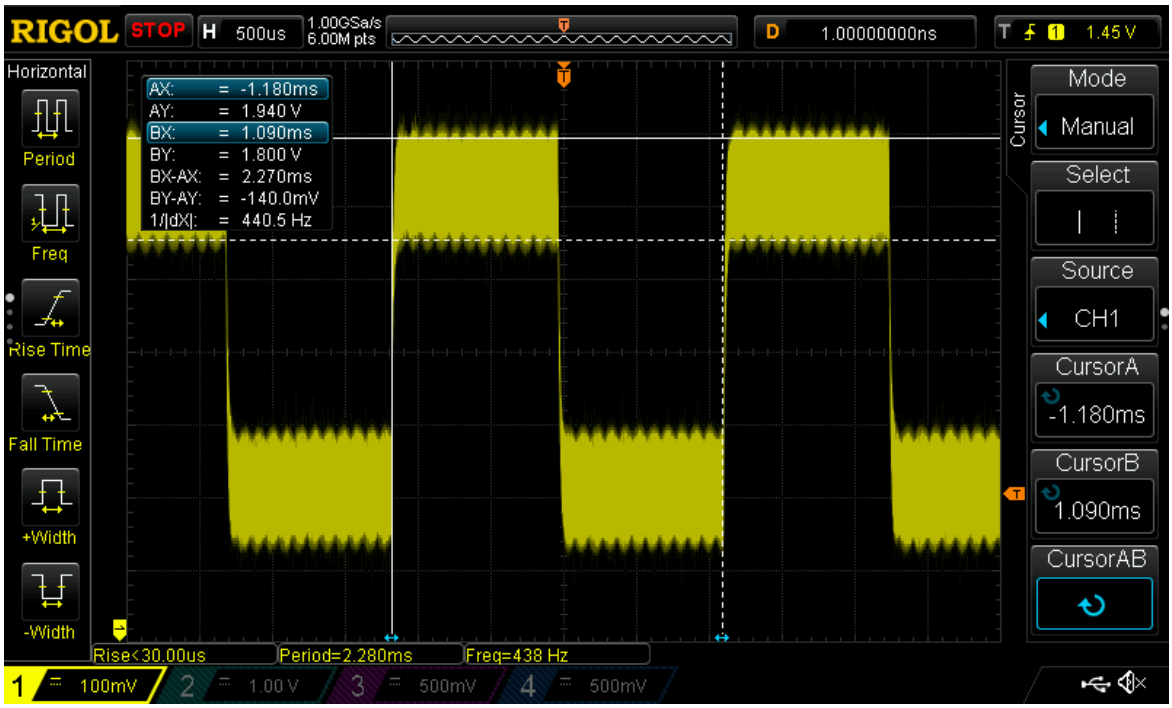

Figure 6.5. Square wave DDS waveform for 440Hz

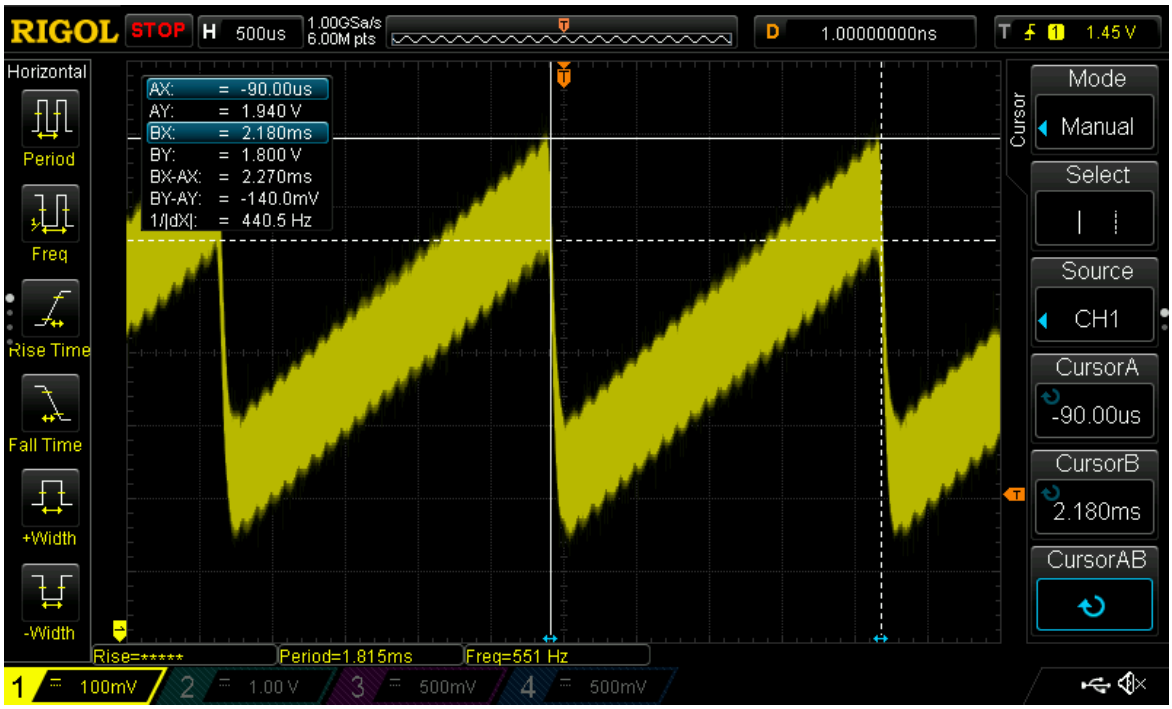

Figure 6.6. Sawtooth wave DDS waveform for 440Hz

shape is visible. All the notes have corresponding recordings available in the .wav format for the reception. Although the DAC worked with the clock 100MHz, and the signal was measured at the output of Lowpass RC circuit, the output was still not free of noise. This

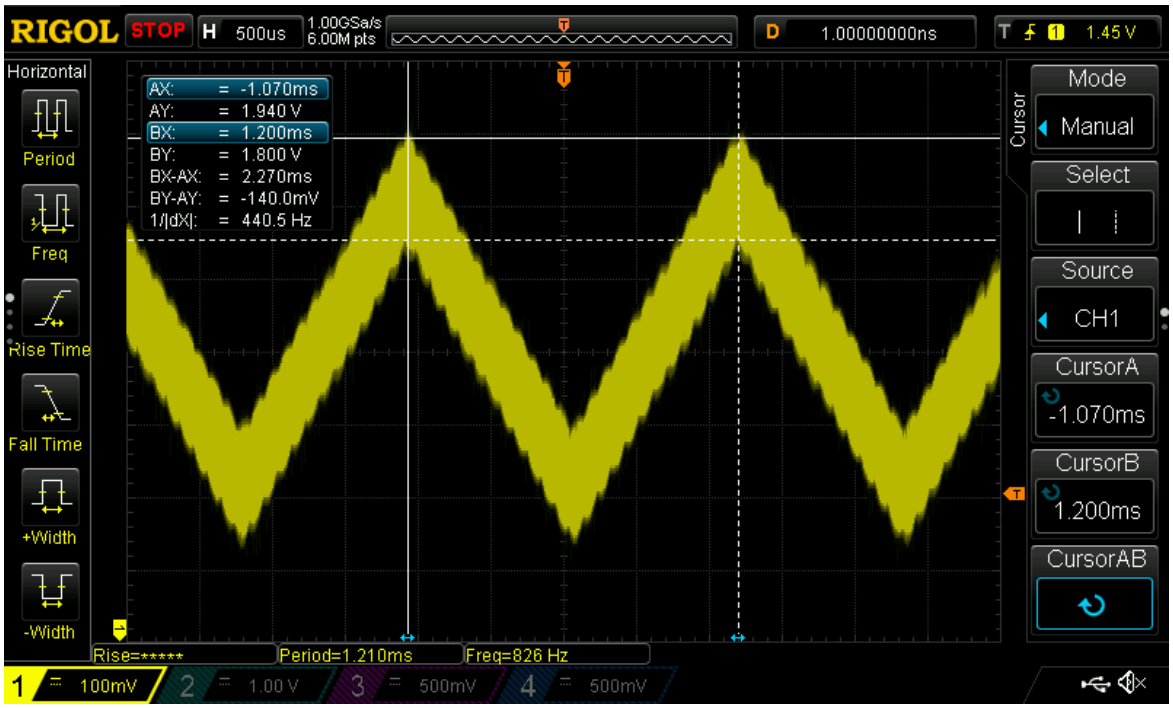

Figure 6.7. Triangle wave DDS waveform for 440Hz

quasi-periodic oscillation not filtered by the RC circuit is of about 10MHz periodicity which does not hint at any reasonable noise source.

6.3. System Testing

Testing software was performed in a left-to-right approach - following the input signal, first testing the MIDI receiver, then the control kernel driver, finally testing the receiving kernel driver, and assessing whether the obtained result is as expected. Testing the ALSA MIDI application was actually quite simple, pressing different MIDI notes and observing what the output is. Some care had to be taken when composing the control word for the kernel driver, as not all MIDI fields were used. The control kernel driver, being as simple as possible was tested by observing whether the proper values appeared on the SignalTap output.

The most difficult part of the testing was the combined DMA driver and ALSA sound-card because of a lack of proper documentation and a tedious testing procedure. Testing was again performed in conjunction with SignalTap software to observe whether any values were being fed to the mSGDMA FIFO and then observing these values on the receiving side. The procedure was as follows:

1. Start recording with the arecord command on Linux side
2. Press Launchkey keys to generate NOTE_ON's
3. Terminate the recording and copy the .wav file using scp to the host

4. Play the recording with aplay to observe whether it matches the one heard by the headphones connected to the DAC
5. Perform a hex dump of the .wav file with xxd
6. Plot the hex data using a python script and observe the waveform (performing endian conversion)

Choosing the proper parameters for ALSA soundcard and DMA buffer sizes was probably the most tedious task and in order to check whether the DMA communication was proceeding without any hiccups a special debug counter was input to the mSGDMA. Plotting of the values received by such deterministic testing rapidly pinpointed any problems. Choosing a proper ALSA encoding was also cumbersome, as with S24_LE the signal appeared to sound almost proper, but when multiple notes were pressed only noise was audible. Changing the format to full 32-bits S32_LE solved the problem and moreover allowed to playback this .wav file in every major music player.

Moreover, as discussed in the previous section, a problem I could not debug until the very end was ALSA overruns which were due to too slow ext4 filesystem. Changing the filesystem to tmpfs alleviated this instantly.

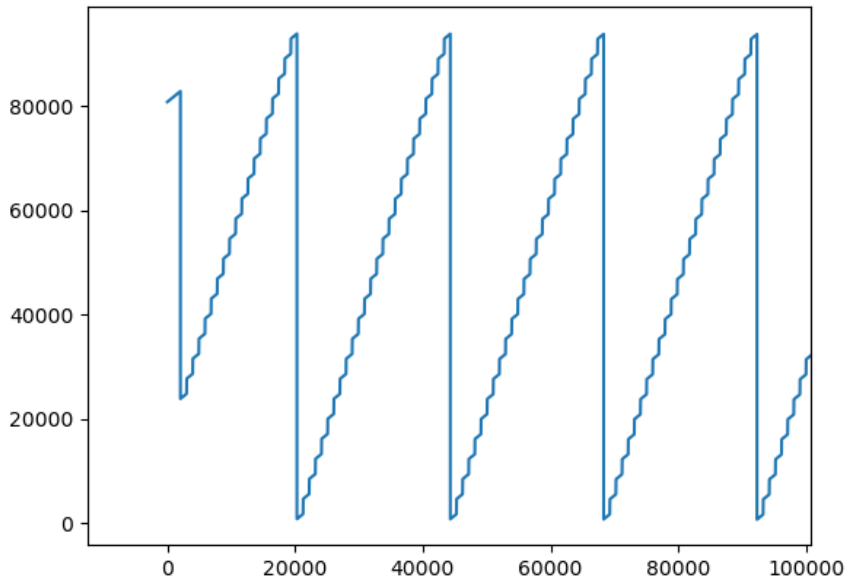

Figure 6.8. Visible jumps on the rising slope of the sawtooth. Caused by wrong DMA transfers window (ALSA period) size.

Two figures showing how DMA period size influences the continuity of the waveform are visible in 6.8 and 6.9.

With proper logs set up and mSGDMA Control Status Register being printed out, all issues with communication with the device were found out and solved. For the first testing

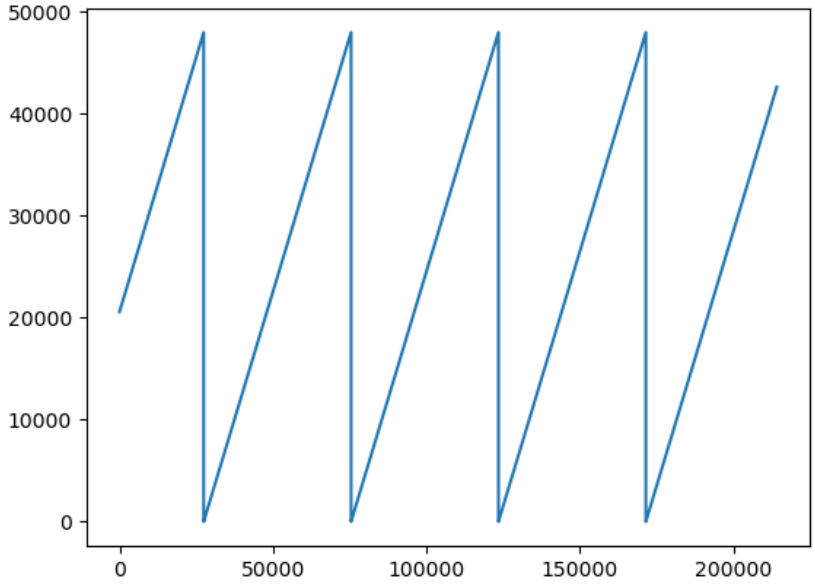

Figure 6.9. Proper continuous sawtooth.

of this communication a special userspace program was written, which would interface the mSGDMA as a character file and read from it in chunks of data.

6.3.1. Examples

Apart from the examples of every waveform being played at a frequency of 440Hz additional examples are necessary to prove that the system behaves as properly. In the file progression.wav one octave of progression may be heard starting from note C4 to C5 and one octave lower and higher in their respective recordings. Additionally, several tracks displaying the polyphonic abilities of the system are presented. Because the samples are mixed, their amplitude has to be reduced (by effectively dividing the sample in half or more), because of that notes played together sound louder than the single notes. Some additional examples are provided, such as very low frequency notes and very high frequency notes playback for a chosen wave-form.

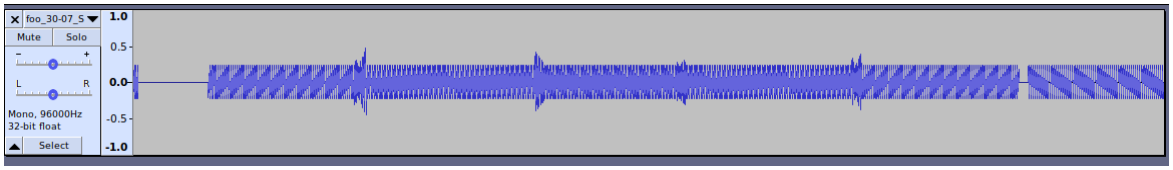

Figure 6.10. Audacity output of an audio recording.

In figure 6.10 an Audacity output of the recorded file may be viewed. Distinct notes being pressed are visible, as well as periods of silence.

6.4. Debugging

Debugging such a complex system requires assessing what the key data is and how much of it should we monitor. Therefore, in PL special counter variables were introduced and plotted in the resulting ModelSIM simulation in order to assess which sample the module is currently processing. Without such facilitation debugging a pipelined architecture would be a hefty task.

Analyzing kernel Oopses was also insightful to debug undocumented ALSA code which misbehaved. Proper logging and logging levels were indispensable in debugging all issues in the kernel driver. Without both ALSA and ALSA-lib (arecord and aplay) source code, it would have been almost impossible to orient oneself in the error messages these libraries outputted.

6.5. Timing and Delays

Since this project is meant to be highly accurate and with minimal delays, proper choices of clocks and timing domains were essential. There are multiple choke-points that incur delays and these are discussed in more detail in this chapter. Most notable of which are: MIDI command input reading and dispatching, generating the DDS sample, and DMA data transfer.

6.5.1. Software delays

The delays in the software are minuscule and can only be alleviated with the usage of RT patched kernel or tweaking some scheduler related settings. Nevertheless, browsing through audio forums I decided it is not worth changing the default settings. Delays that I directly measured and modified were in the DMA driver, and these had a major impact on the quality of the recorded audio file. Starting from normal low-resolution Linux kernel timers, I could observe various delays and non-deterministic interrupts being generated due to the low accuracy of these timers. Replacing them with high-resolution timers reduced the issue, and as mentioned in subsectionSystem Testing the last glitches and delays were removed by the usage of tmpfs instead of ext4.

6.5.2. Hardware delays

Contrary to software delays, hardware delays were almost wholly reducible by accurate programming and careful control of enabling signals. In the table 6.1. Because some signals may be generated in fewer number of cycles than other, the delay measured on the oscilloscope corresponds to the sinewave generation. It is visible that the delays are marginal and due to efficient filter choice and pipelining approach these delays will not grow very significantly if one decides to support more simultaneous samples generation. In figure 6.11 the measured delay between NOTE_ON signal and start of waveform generation is measured. It can be observed that the delay is about 200 microseconds, which given a

Table 6.1. Synthesis delays

Step	Delay (cycles)	Delay (ns)
Acquire	2	20
Phase increment	1	10
Sine calculation	4	40
Filter	3	30
Mixer	10	100
Total delay:	20	200

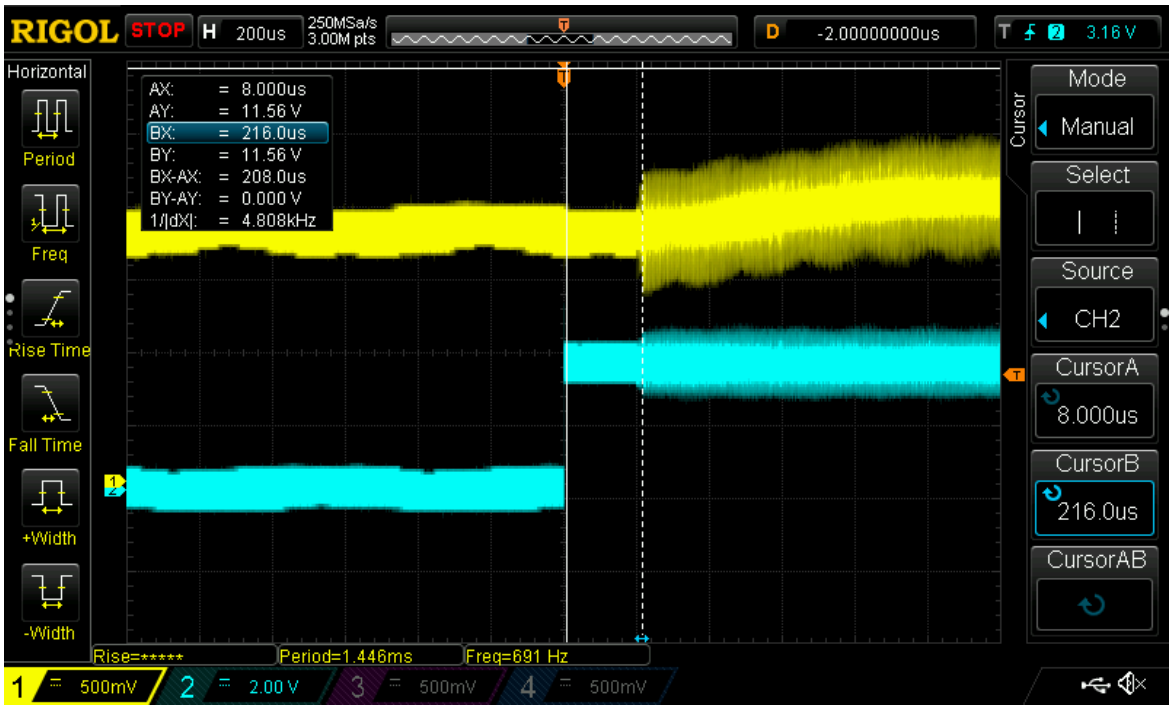

Figure 6.11. Delay between NOTE_ON and the waveform being generated.

96kHz sampling rate amounts to about 20 samples delay, which at this sampling rate is minuscule. Because the sampling rate is much lower than the internal FPGA generation rate, these delays are mostly due to the internal sampling clock speed and not due to the generation process.

7. Results

High-quality sound output was achieved, slightly deteriorated on the analog side because of the usage of a single-bit Sigma-Delta DAC. However, the signal outputted via the ALSA driver meets the project requirements. As shown in the preceding paragraph, the delays are marginal and induced mostly by the software which does not work in real-time and is subject to the OS scheduler and priority degradation. In accordance with the project requirements, this delay is not audible and therefore acceptable. The possibility to playback up to 10 polyphonic sounds is possible, though the amplitudes of all the sounds have to be scaled in order to prevent overflows on the hardware side (which are of a too big magnitude to be in step with the mixer's amplitude-in-phase spreading possibilities). The system is thus portable and with a simple RC filter of even first order, the audio quality is decent and noiseless.

The sound can be recorded from the command line and then further processed if one wishes so. Moreover, it can be processed in real-time if the user wishes for it by means of connection to the ALSA soundcard this device emulates. Different waveform switching is also simple and non-intrusive to the sound being played. Because of this, fluent switching of the waveforms in a round-robin fashion is possible.

7.1. Resource Utilization

Although the De0-Nano-SoC is not a resource-rich platform, it provided even more than enough resources for a high-quality audio synthesis and filtering. Clever usage of LUTs and pipelining techniques allowed for hosting an efficient and effective system onboard. Because this was my first big SoC project, as mentioned before, I based my solution on the Grand Hardware Reference Design from Intel, which itself comprises some LE-heavy blocks. Some essential components, such as HPS-FPGA interconnections take up almost 25% of the board Logic Elements. With the SVF taking up almost half of the resources used by the synthesizer IP core, the rest is used mostly for the LUTs and registers required for pipelining. All the multipliers are offloaded to special DSP blocks which proved to be more than enough contrary to the initial doubts.

On the software side, the necessary components fit in a 512 MB SD card image and can be reduced even more if one desires so. The binaries are very small and the only limiting factor is the rootfs and kernel image size. However, because De0-Nano-SoC is quite formidable in terms of resources, I would consider even expanding the capabilities of kernel and rootfs with some onboard DSP or even adding an LCD and GUI to the synthesizer.

The resource utilization for this project is visible in 7.1. As mentioned above, the total usage of LE's is less than 50%, which contradicts initial doubts.

Revision Name	soc_system
Top-level Entity Name	ghrd
Family	Cyclone V
Device	5CSEMA4U23C6
Timing Models	Final
Logic utilization (in ALMs)	7,226 / 15,880 (46 %)
Total registers	9668
Total pins	230 / 314 (73 %)
Total virtual pins	0
Total block memory bits	604,032 / 2,764,800 (22 %)
Total RAM Blocks	78 / 270 (29 %)
Total DSP Blocks	19 / 84 (23 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1 / 5 (20 %)
Total DLLs	1 / 4 (25 %)

Figure 7.1. Resource utilization for the project.

8. Conclusions

The project goals were successfully achieved, some were even surpassed and because of the way the project was structured, further, expansion is possible and uncomplicated. During the development of this project, I have learned about a variety of subjects, starting from resolving and debugging hardware issues, through digital signal processing topics to finish on Linux device drivers and ALSA architecture understanding. The knowledge gained in these areas will surely allow me to expand upon this project and approach even more complex systems with more ease.

Of course, no project is without its hardships and wrong decisions, and this one also suffered from some of them. Notably, first of all, the wrong approach to pipelining was assumed, instead of sharing resources and incurring delays, there were 10 same component lanes that had to process samples simultaneously. This of course was too costly in terms of resources and was quickly replaced with proper pipelining.

Next, the first implementation of sine LUT utilized 32k entries instead of 512 because it did not utilize any sort of interpolation. This again was too costly and was quickly replaced. The greatest annoyance in writing the software was with debugging the DMA communication and falling in step with undocumented ALSA requirements. It required a significant amount of trial and error to finally fall into place.

Apart from that, some major irritations were encountered with using the Quartus software. When creating the .sof file which was the FPGA programming code, sometimes the board froze when programmed with it during runtime. Due to mismatch in the checksum, the board panicked and rebooted. The error occurs from time to time and is

fixed with reloading the Quartus project (in turn forcing a new checksum) and rescanning the Device Chain in JTAG programmer.

There was also a long-unnoticed issue of having wrong sampling clock speed which initially led to an unsound choice of DMA communication parameters, which when noticed shed more light on this choice and allowed for a proper understanding of this communication.

Wrapping up, apart from being a source of valuable information for me, reignited the dormant musician inside of me and helped me understand some physical and mathematical concepts that eluded me when I studied music as a child. Since, during my studies I had an only basic introduction to DSP, FPGA's, digital and analog electronics, and no courses on Linux drivers, I wanted this project to be a curriculum-filler for me. After successfully completing this project I believe these topics are far more familiar to me and I will be able to apply this knowledge elsewhere.

8.1. Future Work

Naturally, this project can be further expanded, both on the hardware and software side. Starting from adding basic audio effects in hardware, to capturing a voice and acting as a vocoder, the device may be significantly expanded. Because currently, there is no amplitude control and only basic waveshaping possibility, these could be added in the form of ADSR or mixing the waveforms together. Effects like echo or pitch-bend control could also be implemented in hardware. Moreover, the addition of LFO would make this synthesizer an even more complete project. It is also worth mentioning that splitting the sound to more channels would be an interesting exercise, maybe not only just to two channels, but the cinematic standards 5.1 and 7.1.

Since the analog signal being currently outputted is not of the greatest quality, if one wishes to connect this synthesizer directly to an analog system, it would require much more powerful DAC. Although there exists a CODEC on the board which could be used for outputting the signal to the analog external peripherals, it has too low bit resolution (14 bits) compared to the 24 bits generated in the FPGA. Hence, if in the future an alternative way of outputting signal directly from the FPGA wants to be considered, it would probably require investing in an external DAC.

On the software side, the possibilities are also plentiful. Starting from handling different MIDI signals and channels, and even software looping of some of them to create a feeling of a Digital Audio Workstation. The DMA connection is very powerful and currently, it is barely utilized, nevertheless extremely fast, could be used to transfer some additional data when the number of channels would be extended. Current throughput of this connection is: $96'000[1/s] * 4[B] = 384'000[B/s] = 384[kB/s]$ which would be doubled for each channel added. As mentioned earlier, with such a powerful board, one could connect an LCD

display to add GUI capabilities to this system instead of pure command-line control. This would of course increase the size of the solution, but only slightly.

Some real-time processing of the audio could be added on the software side, but this would require adding RT patches to the Linux kernel because without them there would be delays due to non-RT scheduler. JACK audio system would be of much help in this case, as it was designed with professional audio in mind, and is built on top of ALSA.

8.2. Alternative solutions

Some of the solutions proposed in this project could have been implemented differently and will be briefly discussed here. First such major change would be changing from the LUT based synthesis to for example wavetable synthesis as in [5]. This way synthesizer can have many complex pre-mixed waveforms that allow the user to have more different sounds for their use. Implementing it would be nevertheless costly in terms of resource utilization, but with some clever offloading to the SDRAM, it could be achieved in a reasonable time.

The filter choice was based on the requirement of variable cutoff frequency to limit aliasing as much as possible. Instead of such a complicated SVF filter, I could have chosen for example a cascade of simpler IIR filters or even add an FIR filter for even more filtering control.

The audio samples transfer from the FPGA to Linux could have been done with the use of Altera Audio/Video IP Core instead of mSGDMA. However, at the time of starting this implementation, I did not even know about the existence of such a module, because it is available in Altera University Program and not documented in the main manual.

9. Acknowledgements

In the first place, I would like to thank dr hab. inż. Wojciech M. Zabołotny for his invaluable guidelines and materials on the subjects of the thesis. They greatly contributed to my understating of the topic at hand.

Additionally, I would like to thank inż. Adrian Lewczuk and inż. Maciej Urda for inspiring me for choosing such a topic for the thesis and their advocacy on electronic subjects.

Lastly, many thanks to the www.reddit.com communities and their users who helped me garner more knowledge on these topics and understand difficult concepts, especially: r/FPGA, r/DSP, and r/AskElectronics.

References

[1] E. Briggs and S. Veilleux, "FPGA Digital Music Synthesizer", Master's thesis, Worcester Polytechnic Institute, 2015.

[2] J. Borko, G. Dulnik, A. Grzelka, A. Łuczak, and A. Paszkowski, "A parametric synthesizer of audio signals on FPGA", *Measurement Automation Monitoring*, vol. 61, no. 07, pp. 367–369, July 2015.

[3] S. R. Chhetri, B. Poudel, S. Ghimire, S. Shresthamali, and D. K. Sharma, "Implementation of Audio Effect Generator in FPGA", *Nepal Journal of Science and Technology*, vol. 15, no. 1, pp. 89–98, 2014.

[4] S. Gangopadhyay, R. Biswas, and M. Acharya, "Design and implementation of stereo sound enhancement on FPGA", *International Journal of Electrical, Electronics and Data Communication*, vol. 1, no. 3, pp. 2320–2084, May 2013.

[5] J. Wawrzynek, *Final Project Specification - MIDI Sound Synthesizer*, `https://www-inst.eecs.berkeley.edu//~cs150/sp02/lectures/spec.pdf`, 2002.

[6] W. M. Zabołotny, "DMA implementations for FPGA-based data acquisition systems", in *Proceedings Volume 10445*, SPIE, 2017.

[7] *Embedded Peripherals IP User Guide*, 2020.

[8] S. Hauke, *Using the mSGDMA IP: An introduction*, `http://blog.reds.ch/?p=835`.

[9] T. Iwai, *Writing an ALSA Driver*, `https://dri.freedesktop.org/docs/drm/sound/kernel-api/writing-an-alsa-driver.html`.

[10] C. J. Dutton, *FramesPeriods*, `https://www.alsa-project.org/main/index.php/FramesPeriods`.

[11] S. Dimitrov and S. Serafin, "Minivosc - a minimal virtual oscillator driver for ALSA(Advanced Linux Sound Architecture", in *Proceedings of the Linux Audio Conference 2012*, CCRMA, Stanford University, 2012.

[12] ——, *Minivosc*, `https://www.alsa-project.org/wiki/Minivosc`.

[13] C. Rockmore, *Method for Theremin*, `http://www.electrotheremin.com/claramethod.html`, 1998.

[14] N. Awde, *Mellotron: The Machines and the Musicians that Revolutionised Rock*. Bennet & Bloom, 2008.

[15] M. Vail, *The Hammond Organ: Beauty in the B*. Backbeat Books, 2002.

[16] A. Devices, *Fundamentals of Direct Digital Synthesis (DDS)*, `https://www.analog.com/media/en/training-seminars/tutorials/MT-085.pdf`.

[17] D. Vandenneucker, *MIDI tutorial for programmers*, `http://www.music-software-development.com/midi-tutorial.html`.

[18] W. M. Zabołotny, *2nd Order Sigma-Delta DAC*, `https://opencores.org/projects/sigma_delta_dac_dual_loop`, 2012.

[19] JACK, *JACK Audio Connection Kit*, `https://jackaudio.org/`.

[20] W. Joe, *Note names, MIDI numbers and frequencies*, `https://newt.phys.unsw.edu.au/jw/notes.html`.

[21] G. Dan, *Building a quarter sine-wave lookup table*, https://newt.phys.unsw.edu.au/jw/notes.html.

[22] J. E. Volder, "The CORDIC Trigonometric Computing Technique", *IRE Transactions on Electronic Computers*, September 1959.

[23] C. Hal, *Musical applications of microprocessors*. Hasbrouck Heights, N.J.: Hayden Book Co., 1985.

[24] S. Andrew, "Solving the continuous SVF equations using trapeizodal integration and equivalent currents", Cytomic, Tech. Rep., 2013.

[25] G. Peacock, *Building latest bootloaders for soc fpga devices*, https://rocketboards.org/foswiki/Documentation/BuildingBootloader.

[26] Rocketboards, *Embedded linux beginners guide*, https://rocketboards.org/foswiki/Documentation/EmbeddedLinuxBeginnerSGuide.

[27] J. Duchniewicz, *How to set up linux environment on de0-nano soc*, https://docs.google.com/document/d/1INlbTnX7lRPMpKtmacp4uEmOT8S_Sv-j8qElCnHfyQI/edit?usp=sharing.

[28] Intel, *Linux kernel repository*, https://github.com/altera-opensource/linux-socfpga.

List of Symbols and Abbreviations

ADC – Analog to Digital Converter
ALSA – Advanced Linux Sound Architecture
ASIC – Application Specific Integrated Circuit
DAC – Digital to Analog Converter
DDS – Direct Digital Synthesis
DLL – Delay Locked Loop
DMA – Direct Memory Access
DSP – Digital Signal Processing
DTS – Device Tree Structure
EiTI – Wydział Elektroniki i Technik Informacyjnych
FIFO – First-in First-out
FPGA – Field Programmable Gate Array
GPIO – General Purpose IO
HDL – Hardware Description Language
IDE – Integrated Development Environment
JACK – JACK Audio Connection Kit
JTAG – Joint Test Action Group
LE – Logic Elements
LKM – Loadable Kernel Module
MIDI – Musical Instrument Digital Interface
NCO – Numerically Controlled Oscillator
PL – Programmable Logic
PLL – Phase Locked Loop
PW – Politechnika Warszawska
SoC – System on a Chip

List of Figures

List of Appendices

Appendix 1. Verilog Code

Listing 1. *synthesizer_top.v*

```
1   // Top level module for Synthesizer project
2
3   module synthesizer_top_p(input clk,
4           input reset,
5           input avs_s0_write,
6           input avs_s0_read,
7           input [31:0] avs_s0_writedata,
8           output [31:0] avs_s0_readdata,
9           output o_dac_out,
10          output reg [31:0] aso_ss0_data,
11          output reg aso_ss0_valid);
12
13    reg [15:0]  r_oneshot_data;
14    reg clk_en;
15    wire signed[23:0] w_fifo_out;
16    reg signed[23:0] r_fifo_in;
17    reg r_wr_req, r_rd_req;
18    wire w_full, w_empty;
19
20    // DAC connections
21    reg signed[23:0] r_dac_in;
22
23    wire signed[23:0] w_osignal;
24    wire w_rdy;
25    wire signed[23:0] w_mixed_sample;
26
27    wire w_clk_96k_en;
28
29    slow_clk_en #(100_000_000, 96_000) clk_96_en(.clk(clk),
30                                                 .rst(reset),
31                                                 .clk_en(w_clk_96k_en));
32
33    bank_manager_p bm(.clk(clk),
34                .clk_en(clk_en),
35                .reset(reset),
36                .i_data(r_oneshot_data),
37                .o_signal(w_osignal));
38
39    fifo mixed_samples_fifo(.data(r_fifo_in),
40                            .rdclk(clk),
41                            .rdreq(r_rd_req),
42                            .wrclk(clk),
43                            .wrreq(r_wr_req),
44                            .q(w_fifo_out),
45                            .rdempty(w_empty),
46                            .wrfull(w_full));
47
48    mixer mix(.clk(clk),
49              .clk_en(clk_en),
50              .rst(reset),
51              .i_data(w_osignal >>> 1),
52              .o_mixed(w_mixed_sample),
53              .o_rdy(w_rdy));
54
55    dac_dsm2_top dac(.din(r_dac_in),
56                     .dout(o_dac_out),
57                     .clk(clk),
58                     .n_rst(~reset));
59
60    initial begin
61      r_oneshot_data = 16'b0;
62      clk_en = 1'b1;
63      r_fifo_in = 24'b0;
64      r_wr_req = 1'b0;
65      r_rd_req = 1'b0;
66      r_dac_in = 24'b0;
67      aso_ss0_data = 32'b0;
68      aso_ss0_valid = 1'b0;
69    end
70
71    // generator and system clock
72    always @ (posedge clk or posedge reset) begin
73      if (reset) begin
74        r_oneshot_data <= 16'b0;
75        r_fifo_in <= 24'b0;
76        r_wr_req <= 1'b0;
77        r_rd_req <= 1'b0;
78        clk_en <= 1'b0;
79        aso_ss0_data <= 32'b0;
```

```
80          aso_ss0_valid <= 1'b0;
81        end else begin
82        // written enough samples, wait until free slot available
83          if (w_full) begin
84            clk_en <= 1'b0;
85            r_wr_req <= 1'b0;
86          end else begin
87            // if got a full 10 batch
88            if (w_rdy && !r_wr_req) begin
89              r_fifo_in <= w_mixed_sample;
90              r_wr_req <= 1'b1;
91            // signal wr_req just for a one cycle
92            end else begin
93              r_wr_req <= 1'b0;
94            end
95            clk_en <= 1'b1;
96          end
97
98    // reading logic -> passing it
99    // clock_en for DAC sampling and outputing samples to mSGDMA
100         if (w_clk_96k_en) begin
101           if (!w_empty) begin
102             r_rd_req <= 1'b1;
103             aso_ss0_valid <= 1'b1;
104             r_dac_in <= w_fifo_out;
105             aso_ss0_data <= {{8{1'b0}}, w_fifo_out[7:0], w_fifo_out[15:8],
106                              w_fifo_out[23:16]};
107           end
108         end else begin
109           r_rd_req <= 1'b0;
110           aso_ss0_valid <= 1'b0;
111         end
112
113         // Avalon communication logic
114         if (avs_s0_write) begin
115           r_oneshot_data <= avs_s0_writedata[15:0];
116         end else  begin
117           // keep the input value to BM
118           r_oneshot_data <= 16'b0;
119         end
120       end
121     end
122
123   endmodule
```

Listing 2. *bank_manager.v*

```
1   // Pipelined bank manager
2
3   module bank_manager_p(input clk,
4                         input clk_en,
5                         input reset,
6                         input[15:0] i_data,
7                         output reg signed[23:0] o_signal);
8
9     parameter NBANKS = 10;
10    localparam SINE = 2'b00, SQUARE = 2'b01, SAWTOOTH = 2'b10, TRIANGLE = 2'b11;
11
12    reg[6:0]          midi_vals[NBANKS-1:0];
13    reg[6:0]          r_cur_midi;
14
15    reg[1:0]        waveform;
16    reg             r_sine_en, r_square_en, r_sawtooth_en, r_triangle_en;
17
18    wire              w_pb_valid, w_wave_valid, w_qs_valid, w_square_valid,
19                      w_sawtooth_valid, w_triangle_valid, w_svf_valid;
20    wire signed[23:0] w_wave_out, w_square_out, w_sawtooth_out,
21                      w_triangle_out, w_qs_out, w_svf_out;
22    wire [23:0]       w_pb_out;
23    wire[6:0]         w_pb_o_midi, w_wave_o_midi, w_qs_o_midi, w_square_o_midi,
24                      w_sawtooth_o_midi, w_triangle_o_midi, w_svf_o_midi;
25
26    wire              w_cmd;
27    wire[6:0]         w_midi;
28
29    assign w_cmd = i_data[15];
30    assign w_midi = i_data[14:8];
31    assign w_velocity = i_data[7:0];
32
33    phase_bank_p pb(.clk(clk),
34                    .clk_en(clk_en),
35                    .rst(reset),
36                    .i_midi(r_cur_midi),
37                    .o_midi(w_pb_o_midi),
```

```
38                      .o_valid(w_pb_valid),
39                      .o_phase(w_pb_out));
40
41      quarter_sine_p sine(.clk(clk),
42                          .clk_en(clk_en),
43                          .wav_en(r_sine_en),
44                          .rst(reset),
45                          .i_midi(w_pb_o_midi),
46                          .o_midi(w_qs_o_midi),
47                          .i_phase(w_pb_out),
48                          .i_valid(w_pb_valid),
49                          .o_valid(w_qs_valid),
50                          .o_sine(w_qs_out));
51
52      square_wave square(.clk(clk),
53                         .clk_en(clk_en),
54                         .wav_en(r_square_en),
55                         .rst(reset),
56                         .i_midi(w_pb_o_midi),
57                         .o_midi(w_square_o_midi),
58                         .i_phase(w_pb_out),
59                         .i_valid(w_pb_valid),
60                         .o_valid(w_square_valid),
61                         .o_square(w_square_out));
62
63      sawtooth_wave sawtooth(.clk(clk),
64                             .clk_en(clk_en),
65                             .wav_en(r_sawtooth_en),
66                             .rst(reset),
67                             .i_midi(w_pb_o_midi),
68                             .o_midi(w_sawtooth_o_midi),
69                             .i_phase(w_pb_out),
70                             .i_valid(w_pb_valid),
71                             .o_valid(w_sawtooth_valid),
72                             .o_sawtooth(w_sawtooth_out));
73
74      triangle_wave triangle(.clk(clk),
75                             .clk_en(clk_en),
76                             .wav_en(r_triangle_en),
77                             .rst(reset),
78                             .i_midi(w_pb_o_midi),
79                             .o_midi(w_triangle_o_midi),
80                             .i_phase(w_pb_out),
81                             .i_valid(w_pb_valid),
82                             .o_valid(w_triangle_valid),
83                             .o_triangle(w_triangle_out));
84
85      state_variable_filter_iir_p SVF(.clk(clk),
86                                      .clk_en(clk_en),
87                                      .rst(reset),
88                                      .i_midi(w_wave_o_midi),
89                                      .o_midi(w_svf_o_midi),
90                                      .i_data(w_wave_out),
91                                      .i_valid(w_wave_valid),
92                                      .o_valid(w_svf_valid),
93                                      .o_filtered(w_svf_out));
94
95
96      assign w_wave_out = w_qs_out | w_sawtooth_out | w_triangle_out;
97      assign w_wave_o_midi = w_qs_o_midi | w_sawtooth_o_midi | w_triangle_o_midi;
98      assign w_wave_valid = w_qs_valid | w_sawtooth_valid | w_triangle_valid;
99
100     integer v_idx;
101     integer i;
102
103     initial begin
104       for (i = 0; i < NBANKS; i = i + 1) begin
105         midi_vals[i] = 7'h0;
106       end
107       r_cur_midi = 7'b0;
108       waveform = 2'b0;
109       r_sine_en = 1'b1;
110       r_square_en = 1'b0;
111       r_sawtooth_en = 1'b0;
112       r_triangle_en = 1'b0;
113       o_signal = 24'b0;
114       v_idx = 0;
115     end
116
117     always @(posedge clk or posedge reset) begin
118       if (reset) begin
119         for (i = 0; i < NBANKS; i = i + 1) begin
120           midi_vals[i] <= 7'h0;
121         end
122         // and more...
```

```
123         r_cur_midi <= 7'b0;
124         waveform <= 2'b0;
125         r_sine_en <= 1'b1;
126         r_square_en <= 1'b0;
127         r_sawtooth_en <= 1'b0;
128         r_triangle_en <= 1'b0;
129         o_signal <= 24'b0;
130         v_idx <= 0;
131       // handle commands
132       end else begin
133         if (w_cmd == 1) begin
134           if (w_midi == 7'b0 && w_velocity == 8'b0) begin // CHANGE_WAVE
135             if (waveform == TRIANGLE) begin // turn on sine
136               waveform <= SINE;
137               r_sine_en <= 1'b1;
138               r_square_en <= 1'b0;
139               r_sawtooth_en <= 1'b0;
140               r_triangle_en <= 1'b0;
141             end else if (waveform == SINE) begin // turn on square
142               waveform <= SQUARE;
143               r_sine_en <= 1'b0;
144               r_square_en <= 1'b1;
145               r_sawtooth_en <= 1'b0;
146               r_triangle_en <= 1'b0;
147             end else if (waveform == SQUARE) begin // turn on saw
148               waveform <= SAWTOOTH;
149               r_sine_en <= 1'b0;
150               r_square_en <= 1'b0;
151               r_sawtooth_en <= 1'b1;
152               r_triangle_en <= 1'b0;
153             end else if (waveform == SAWTOOTH) begin // turn on triangle
154               waveform <= TRIANGLE;
155               r_sine_en <= 1'b0;
156               r_square_en <= 1'b0;
157               r_sawtooth_en <= 1'b0;
158               r_triangle_en <= 1'b1;
159             end
160           end else if (midi_vals[0] == 7'h0) begin
161             midi_vals[0] <= w_midi;
162           end else if (midi_vals[1] == 7'h0) begin
163             midi_vals[1] <= w_midi;
164           end else if (midi_vals[2] == 7'h0) begin
165             midi_vals[2] <= w_midi;
166           end else if (midi_vals[3] == 7'h0) begin
167             midi_vals[3] <= w_midi;
168           end else if (midi_vals[4] == 7'h0) begin
169             midi_vals[4] <= w_midi;
170           end else if (midi_vals[5] == 7'h0) begin
171             midi_vals[5] <= w_midi;
172           end else if (midi_vals[6] == 7'h0) begin
173             midi_vals[6] <= w_midi;
174           end else if (midi_vals[7] == 7'h0) begin
175             midi_vals[7] <= w_midi;
176           end else if (midi_vals[8] == 7'h0) begin
177             midi_vals[8] <= w_midi;
178           end else if (midi_vals[9] == 7'h0) begin
179             midi_vals[9] <= w_midi;
180           end // failure to playback yet another sound should be signalled to user?
181         end else if (w_cmd == 0) begin
182           if (w_midi == 7'h7f) begin // STOP_ALL
183             for (i = 0; i < NBANKS; i = i + 1) begin
184               midi_vals[i] <= 7'h0;
185             end
186           end else if (midi_vals[0] == w_midi) begin
187             midi_vals[0] <= 7'h0; // MIDI 0 is equal to turn off
188           end else if (midi_vals[1] == w_midi) begin
189             midi_vals[1] <= 7'h0;
190           end else if (midi_vals[2] == w_midi) begin
191             midi_vals[2] <= 7'h0;
192           end else if (midi_vals[3] == w_midi) begin
193             midi_vals[3] <= 7'h0;
194           end else if (midi_vals[4] == w_midi) begin
195             midi_vals[4] <= 7'h0;
196           end else if (midi_vals[5] == w_midi) begin
197             midi_vals[5] <= 7'h0;
198           end else if (midi_vals[6] == w_midi) begin
199             midi_vals[6] <= 7'h0;
200           end else if (midi_vals[7] == w_midi) begin
201             midi_vals[7] <= 7'h0;
202           end else if (midi_vals[8] == w_midi) begin
203             midi_vals[8] <= 7'h0;
204           end else if (midi_vals[9] == w_midi) begin
205             midi_vals[9] <= 7'h0;
206           end
207         end
```

```
208
209         // handle dispatching midi with valid info
210         if (clk_en) begin
211           r_cur_midi <= midi_vals[v_idx]; // if 7'h0 then invalid
212
213           if (w_svf_valid) begin
214             o_signal <= w_svf_out;
215           end else if (w_square_valid) begin // Bypass the SVF for square
216             o_signal <= w_square_out >>> 1;
217           end else begin
218             o_signal <= 24'b0;
219           end
220
221           if (v_idx == NBANKS - 1)
222             v_idx <= 0;
223           else
224             v_idx <= v_idx + 1;
225         end
226       end
227     end
228   endmodule
```

Listing 3. *quarter_sine.v*

```
1   // Quarter-wave sine logic Pipelined
2
3   module quarter_sine_p(input clk,
4         input clk_en,
5         input wav_en,
6         input rst,
7         input[6:0] i_midi,
8         output reg[6:0] o_midi,
9         input[23:0] i_phase,
10        input i_valid,
11        output reg o_valid,
12        output reg signed[23:0] o_sine);
13      parameter NBANKS = 10;
14
15      reg r_negate_1[1:0][NBANKS-1:0];
16      reg r_negate_2[1:0][NBANKS-1:0];
17      reg signed[15:0] r_lut_sine_1[NBANKS-1:0];
18      reg signed[15:0] r_lut_sine_2[NBANKS-1:0];
19
20      // for buffering values
21      reg valid[2:0];
22      reg[6:0] midi[2:0];
23
24      reg[8:0]    r_cur_index_1, r_cur_index_2;
25      wire[15:0] w_sine_out_1, w_sine_out_2;
26
27      // next step phase
28      wire[10:0] i_phase_next;
29      reg [12:0] r_phase_frac[1:0][NBANKS-1:0];
30
31      reg signed[15:0] r_mult_1a, r_mult_2a;
32      reg [13:0]       r_mult_1b, r_mult_2b;
33      wire signed[29:0] w_result_1, w_result_2;
34      wire signed[29:0] w_result_added;
35
36      // LUTs of size 512 outputting 16 bit sine values to be interpolated
37      quarter_sine_lut slut_1(.i_phase(r_cur_index_1),
38                              .o_val(w_sine_out_1));
39                              // difference between them is 2**13
40                              // so interpolation distance is 8192
41      quarter_sine_lut slut_2(.i_phase(r_cur_index_2),
42                              .o_val(w_sine_out_2));
43
44      // mult
45      sine_mult mult_1(.dataa(r_mult_1a),
46                       .datab(r_mult_1b),
47                       .result(w_result_1));
48      sine_mult mult_2(.dataa(r_mult_2a),
49                       .datab(r_mult_2b),
50                       .result(w_result_2));
51
52      integer v_idx;
53      integer i, j;
54
55      assign i_phase_next = i_phase[23:13] + 11'b1;
56      // this is the NEXT sample, our LUT has entries
57      // from 1 to 2 ** 11 indexed with 11 bits
58      assign w_result_added = w_result_1 + w_result_2;
59
60      initial begin
```

```
61          o_sine = 24'b0;
62          v_idx = 8; // delay of 2 computation blocks
63          r_cur_index_1 = 9'b0;
64          r_cur_index_2 = 9'b0;
65          r_mult_1a = 16'b0;
66          r_mult_2a = 16'b0;
67          r_mult_1b = 14'b0;
68          r_mult_2b = 14'b0;
69          for (i = 0; i < NBANKS; i = i + 1) begin
70            for (j = 0; j < 2; j = j + 1) begin
71              r_negate_1[j][i] = 1'b0;
72              r_negate_2[j][i] = 1'b0;
73              r_phase_frac[j][i] = 13'b0;
74            end
75            r_lut_sine_1[i] = 16'b0;
76            r_lut_sine_2[i] = 16'b0;
77          end
78          for (i = 0; i < 3; i = i + 1) begin
79            valid[i] = 1'b0;
80            midi[i] = 7'b0;
81          end
82          o_midi = 7'b0;
83        end
84
85        always @(posedge clk or posedge rst) begin
86          if (rst) begin
87            o_sine <= 16'b0;
88            v_idx <= 8; // delay of 2 computation blocks
89            r_cur_index_1 <= 9'b0;
90            r_cur_index_2 <= 9'b0;
91            r_mult_1a <= 16'b0;
92            r_mult_2a <= 16'b0;
93            r_mult_1b <= 14'b0;
94            r_mult_2b <= 14'b0;
95            for (i = 0; i < NBANKS; i = i + 1) begin
96              for (j = 0; j < 2; j = j + 1) begin
97                r_negate_1[j][i] <= 1'b0;
98                r_negate_2[j][i] <= 1'b0;
99                r_phase_frac[j][i] <= 13'b0;
100             end
101             r_lut_sine_1[i] <= 16'b0;
102             r_lut_sine_2[i] <= 16'b0;
103           end
104           for (i = 0; i < 3; i = i + 1) begin
105             valid[i] <= 1'b0;
106             midi[i] <= 7'b0;
107           end
108           o_midi <= 7'b0;
109         end else if(clk_en && wav_en) begin
110           // clock one
111           if (i_valid) begin
112             r_negate_1[0][v_idx] <= i_phase[23]; // negate or not
113             r_negate_2[0][v_idx] <= i_phase_next[10];
114             r_cur_index_1 <= i_phase[22] ?
115                              ~i_phase[21:13] :
116                              i_phase[21:13];
117                              // invert index if 2nd MSB is set
118             r_cur_index_2 <= i_phase_next[9] ?
119                              ~i_phase_next[8:0] :
120                              i_phase_next[8:0];
121                              // take just the lower part
122             // store current phase fraction
123             r_phase_frac[0][v_idx] <= i_phase[12:0];
124           end
125
126           // clock two
127           if (valid[0]) begin
128             if (v_idx == 0) begin
129               r_lut_sine_1[NBANKS - 1] <= w_sine_out_1;
130               r_lut_sine_2[NBANKS - 1] <= w_sine_out_2;
131                //to avoid overwriting
132               r_negate_1[1][NBANKS - 1] <= r_negate_1[0][NBANKS - 1];
133               r_negate_2[1][NBANKS - 1] <= r_negate_2[0][NBANKS - 1];
134               r_phase_frac[1][NBANKS - 1] <= r_phase_frac[0][NBANKS - 1];
135             end else begin
136               r_lut_sine_1[v_idx - 1] <= w_sine_out_1;
137               r_lut_sine_2[v_idx - 1] <= w_sine_out_2;
138               r_negate_1[1][v_idx - 1] <= r_negate_1[0][v_idx - 1];
139               r_negate_2[1][v_idx - 1] <= r_negate_2[0][v_idx - 1];
140               r_phase_frac[1][v_idx - 1] <= r_phase_frac[0][v_idx - 1];
141             end
142           end
143
144           //clock three
145           if (valid[1]) begin // output only valid values
```

```
146             if (v_idx == 0) begin
147               if (r_negate_1[1][NBANKS - 2])
148                 r_mult_1a <= -r_lut_sine_1[NBANKS - 2];
149               else
150                 r_mult_1a <= r_lut_sine_1[NBANKS - 2];
151
152               if (r_negate_2[1][NBANKS - 2])
153                 r_mult_2a <= -r_lut_sine_2[NBANKS - 2];
154               else
155                 r_mult_2a <= r_lut_sine_2[NBANKS - 2];
156
157               r_mult_1b <= 14'h2000 - r_phase_frac[1][NBANKS - 2];
158               r_mult_2b <= r_phase_frac[1][NBANKS - 2];
159             end else if (v_idx == 1) begin
160               if (r_negate_1[1][NBANKS - 1])
161                 r_mult_1a <= -r_lut_sine_1[NBANKS - 1];
162               else
163                 r_mult_1a <= r_lut_sine_1[NBANKS - 1];
164
165               if (r_negate_2[1][NBANKS - 1])
166                 r_mult_2a <= -r_lut_sine_2[NBANKS - 1];
167               else
168                 r_mult_2a <= r_lut_sine_2[NBANKS - 1];
169
170               r_mult_1b <= 14'h2000 - r_phase_frac[1][NBANKS - 1];
171               r_mult_2b <= r_phase_frac[1][NBANKS - 1];
172             end else begin
173               if (r_negate_1[1][v_idx - 2])
174                 r_mult_1a <= -r_lut_sine_1[v_idx - 2];
175               else
176                 r_mult_1a <= r_lut_sine_1[v_idx - 2];
177
178               if (r_negate_2[1][v_idx - 2])
179                 r_mult_2a <= -r_lut_sine_2[v_idx - 2];
180               else
181                 r_mult_2a <= r_lut_sine_2[v_idx - 2];
182
183               r_mult_1b <= 14'h2000 - r_phase_frac[1][v_idx - 2];
184               r_mult_2b <= r_phase_frac[1][v_idx - 2];
185             end
186           end
187
188           // clock four -> multiply result
189           if (valid[2]) begin
190             o_sine <= w_result_added[29:6];
191           end else begin
192             o_sine <= 24'b0;
193           end
194
195           // move valid value
196           valid[0] <= i_valid;
197           valid[1] <= valid[0];
198           valid[2] <= valid[1];
199           o_valid <= valid[2];
200
201           // move midi value
202           midi[0] <= i_midi;
203           midi[1] <= midi[0];
204           midi[2] <= midi[1];
205           o_midi <= midi[2];
206
207           if (v_idx == NBANKS - 1)
208             v_idx <= 0;
209           else
210             v_idx <= v_idx + 1;
211         end else if(!wav_en) begin
212           o_sine <= 24'b0;
213           o_midi <= 7'b0;
214           o_valid <= 1'b0;
215         end
216       end
217   endmodule
```

Listing 4. *square_wave.v*

```
1   // A square wave module in 1 cycle
2
3   module square_wave(input clk,
4                      input clk_en,
5                      input wav_en,
6                      input rst,
7                      input[6:0] i_midi,
8                      output reg[6:0] o_midi,
9                      input[23:0] i_phase,
```

```
10                  input i_valid,
11                  output reg o_valid,
12                  output reg signed[23:0] o_square);
13
14      reg signed[23:0] MAX_SIGNED = {1'sb0, {23{1'sb1}}};
15      reg signed[23:0] MIN_SIGNED = {1'sb1, {23{1'sb0}}};
16
17      initial begin
18        o_square = 24'b0;
19        o_midi = 7'b0;
20        o_valid = 1'b0;
21      end
22
23      always @(posedge clk or posedge rst) begin
24        if (rst) begin
25          o_square <= 24'b0;
26          o_midi <= 7'b0;
27          o_valid <= 1'b0;
28        end else if(clk_en && wav_en) begin
29          // greater than half, output -1
30          if (i_phase >= MIN_SIGNED && i_valid) begin
31            o_square <= MIN_SIGNED;
32          // less then half, output 1
33          end else if(i_valid) begin
34            o_square <= MAX_SIGNED;
35          end else begin
36            o_square <= 24'b0;
37          end
38          o_midi <= i_midi;
39          o_valid <= i_valid;
40        end else if(!wav_en) begin
41          o_square <= 24'b0;
42          o_midi <= 7'b0;
43          o_valid <= 1'b0;
44        end
45      end
46
47  endmodule
```

Listing 5. *state_variable_filter.v*

```
1   // Pipelined State variable filter based on Andrew Simper's SVF whitepaper
2   // https://cytomic.com/files/dsp/SvfLinearTrapOptimised2.pdf
3
4   module state_variable_filter_iir_p(input clk,
5                                      input clk_en,
6                                      input rst,
7                                      input[6:0] i_midi,
8                                      output reg[6:0] o_midi,
9                                      input signed[23:0] i_data,
10                                     input i_valid,
11                                     output reg o_valid,
12                                     output reg signed[23:0] o_filtered // == v2
13                                     );
14    parameter NBANKS = 10;
15
16    reg signed[39:0] v1[NBANKS-1:0];
17    reg signed[39:0] v2[NBANKS-1:0];
18    reg signed[39:0] v3[NBANKS-1:0];
19    reg signed[39:0] ic1eq[NBANKS-1:0];
20    reg signed[39:0] ic2eq[NBANKS-1:0];
21
22    wire signed[39:0] w_a1, w_a2, w_a3;
23    wire signed[79:0] mR_0, mR_1, mR_2, mR_3;
24
25    reg signed[39:0] r_m_a1, r_m_a2, r_m_a3, r_m_ic1eq, r_m_v3;
26
27    // for buffering values
28    reg valid[1:0];
29    reg[6:0] midi[1:0];
30
31    wire signed[39:0] w_extended_i_data;
32
33    coefficients_lut lut(.i_midi(i_midi),
34                         .o_a1(w_a1),
35                         .o_a2(w_a2),
36                         .o_a3(w_a3));
37
38    lpm_multiplier
39              mult0(.dataa(r_m_a1),
40                    .datab(r_m_ic1eq),
41                    .result(mR_0)),
42              mult1(.dataa(r_m_a2),
43                    .datab(r_m_v3),
```

```
44                      .result(mR_1)),
45              mult2(.dataa(r_m_a2),
46                      .datab(r_m_ic1eq),
47                      .result(mR_2)),
48              mult3(.dataa(r_m_a3),
49                      .datab(r_m_v3),
50                      .result(mR_3));
51
52      integer v_idx;
53      integer i;
54
55      assign w_extended_i_data = {{3{i_data[23]}}, {i_data[22:0]}, 14'b0};
56
57      initial begin
58        for (i = 0; i < NBANKS; i = i + 1) begin
59          v1[i] = 40'b0;
60          v2[i] = 40'b0;
61          v3[i] = 40'b0;
62          ic1eq[i] = 40'b0;
63          ic2eq[i] = 40'b0;
64        end
65        for (i = 0; i < 2; i = i + 1) begin
66          valid[i] = 1'b0;
67          midi[i] = 7'b0;
68        end
69        r_m_a1 = 40'b0;
70        r_m_a2 = 40'b0;
71        r_m_a3 = 40'b0;
72        r_m_ic1eq = 40'b0;
73        r_m_v3 = 40'b0;
74
75        v_idx = 5;
76      end
77
78      always @(posedge clk or posedge rst) begin
79        if (rst) begin
80          for (i = 0; i < NBANKS; i = i + 1) begin
81            v1[i] <= 40'b0;
82            v2[i] <= 40'b0;
83            v3[i] <= 40'b0;
84            ic1eq[i] <= 40'b0;
85            ic2eq[i] <= 40'b0;
86          end
87          for (i = 0; i < 2; i = i + 1) begin
88            valid[i] <= 1'b0;
89            midi[i] <= 7'b0;
90          end
91          r_m_a1 <= 40'b0;
92          r_m_a2 <= 40'b0;
93          r_m_a3 <= 40'b0;
94          r_m_ic1eq <= 40'b0;
95          r_m_v3 <= 40'b0;
96
97          v_idx <= 5;
98        end else if (clk_en) begin
99          // three cycles of delay - sine-like solution
100         // first cycle calc coefficients -> clear registers if midi 00
101         if (i_midi == 7'b0) begin
102           v1[v_idx] <= 40'b0;
103           v2[v_idx] <= 40'b0;
104           v3[v_idx] <= 40'b0;
105           ic1eq[v_idx] <= 40'b0;
106           ic2eq[v_idx] <= 40'b0;
107           // fill the lpm_multiplier
108           r_m_a1 <= 40'b0;
109           r_m_a2 <= 40'b0;
110           r_m_a3 <= 40'b0;
111           r_m_ic1eq <= 40'b0;
112           r_m_v3 <= 40'b0;
113         end else begin
114           // fill the lpm_multiplier
115           r_m_a1 <= w_a1;
116           r_m_a2 <= w_a2;
117           r_m_a3 <= w_a3;
118           r_m_ic1eq <= ic1eq[v_idx];
119           r_m_v3 <= w_extended_i_data - ic2eq[v_idx]; // to prevent 1 cycle delay
120           // filter step
121           v3[v_idx] <= w_extended_i_data - ic2eq[v_idx];
122         end
123
124         // second cycle -> obtain multiplication results
125         if (v_idx == 0) begin
126         // remove scaling factor due to multiplication
127           v1[NBANKS - 1] <= (mR_0 >>> 37) + (mR_1 >>> 37);
128           v2[NBANKS - 1] <= ic2eq[NBANKS - 1] + (mR_2 >>> 37) + (mR_3 >>> 37);
```

```
129         end else begin
130           v1[v_idx - 1] <= (mR_0 >>> 37) + (mR_1 >>> 37);
131           v2[v_idx - 1] <= ic2eq[v_idx - 1] + (mR_2 >>> 37) + (mR_3 >>> 37);
132         end
133
134         // third cycle -> last steps and output result
135         if (v_idx == 0) begin
136           ic1eq[NBANKS - 2] <= (v1[NBANKS - 2] <<< 1) - ic1eq[NBANKS - 2];
137           ic2eq[NBANKS - 2] <= (v2[NBANKS - 2] <<< 1) - ic2eq[NBANKS - 2];
138           o_filtered <= {v2[NBANKS - 2][39], v2[NBANKS - 2][36:14]};
139         end else if (v_idx == 1) begin
140           ic1eq[NBANKS - 1] <= (v1[NBANKS - 1] <<< 1) - ic1eq[NBANKS - 1];
141           ic2eq[NBANKS - 1] <= (v2[NBANKS - 1] <<< 1) - ic2eq[NBANKS - 1];
142           o_filtered <= {v2[NBANKS - 1][39], v2[NBANKS - 1][36:14]};
143         end else begin
144           ic1eq[v_idx - 2] <= (v1[v_idx - 2] <<< 1) - ic1eq[v_idx - 2];
145           ic2eq[v_idx - 2] <= (v2[v_idx - 2] <<< 1) - ic2eq[v_idx - 2];
146           o_filtered <= {v2[v_idx - 2][39], v2[v_idx - 2][36:14]};
147         end
148
149         // move valid value
150         valid[0] <= i_valid;
151         valid[1] <= valid[0];
152         o_valid <= valid[1];
153
154         // move midi value
155         midi[0] <= i_midi;
156         midi[1] <= midi[0];
157         o_midi <= midi[1];
158
159         if (v_idx == NBANKS - 1)
160           v_idx <= 0;
161         else
162           v_idx <= v_idx + 1;
163       end
164     end
165
166 endmodule
```

Listing 6. *mixer.v*

```
1   // Digital mixer of 10 values - counts to 10 and outputs stored value
2
3   module mixer(input clk,
4             input clk_en,
5             input rst,
6             input signed[22:0] i_data,
7             output reg signed[23:0] o_mixed,
8             output reg o_rdy);
9
10    // Because Verilog does not like signed numbers,
11    // these have to be explicite signed regs
12    reg signed[23:0] MAX_SIGNED = {1'sb0, {23{1'sb1}}};
13    reg signed[23:0] MIN_SIGNED = {1'sb1, {23{1'sb0}}};
14
15    reg signed[27:0] r_mixed, r_overflow;
16    integer v_idx;
17
18    initial begin
19      r_mixed = 28'b0;
20      r_overflow = 28'b0;
21      o_mixed = 24'b0;
22      o_rdy = 1'b0;
23      v_idx = 0;
24    end
25
26    always @ (posedge clk or posedge rst) begin
27      if (rst) begin
28        r_mixed <= 28'b0;
29        r_overflow <= 28'b0;
30        o_mixed <= 24'b0;
31        o_rdy <= 1'b0;
32        v_idx <= 0;
33      end
34
35      else if (clk_en) begin
36        // handle overflow/underflow
37        // because in-sync with BM, do everything in 10 cycles,
38        // reset r_mixed and output o_mixed + i_data in range
39        if (v_idx === 9) begin
40        // OF, output max and fill new reg with i_data + as much
41        // of overflow as possible
42          if (r_mixed + i_data > MAX_SIGNED) begin
43            o_mixed <= MAX_SIGNED; // output maximum
```

```
44              // r_mixed + i_data - MAX_SIGNED <- how much
45              // current sample overflows the buffer
46              // OF remainder + curr_overflow too big,
47              // reduce OF but add remainder
48              if ((r_mixed + i_data - MAX_SIGNED) + r_overflow
49                  > MAX_SIGNED) begin
50                r_mixed <= MAX_SIGNED;
51                r_overflow <= r_overflow +
52                            (r_mixed + i_data - MAX_SIGNED) - MAX_SIGNED;
53              end else begin
54                r_mixed <= r_overflow + (r_mixed + i_data - MAX_SIGNED);
55                r_overflow <= 28'b0; // all of it went to mixed;
56              end
57            end else if (r_mixed + i_data < MIN_SIGNED) begin // UF
58              o_mixed <= MIN_SIGNED; // output maximum
59              // r_mixed + i_data - MIN_SIGNED <- how much current sample
60              // underflows the buffer
61                        // UF remainder + curr_overflow too small,
62                        // reduce UF but add remainder
63              if ((r_mixed + i_data - MIN_SIGNED) + r_overflow
64                  < MIN_SIGNED) begin
65                r_mixed <= MIN_SIGNED;
66                r_overflow <= r_overflow +
67                            (r_mixed + i_data - MIN_SIGNED) - MIN_SIGNED;
68              end else begin
69                r_mixed <= r_overflow + (r_mixed + i_data - MIN_SIGNED);
70                r_overflow <= 28'b0; // all of it went to mixed;
71              end
72            // in range - try offloading the overflow
73            end else begin
74                // OF still too big
75              if (r_mixed + i_data + r_overflow > MAX_SIGNED) begin
76                o_mixed <= MAX_SIGNED;
77                r_mixed <= (r_mixed + i_data + r_overflow) - MAX_SIGNED;
78              // UF still too small
79              end else if (r_mixed + i_data + r_overflow < MIN_SIGNED) begin
80                o_mixed <= MIN_SIGNED;
81                r_mixed <= (r_mixed + i_data + r_overflow) - MIN_SIGNED;
82              // can get rid off of the OF/UF or equal to 0
83              end else begin
84                o_mixed <= r_mixed + i_data + r_overflow;
85                r_mixed <= 28'b0;
86              end
87              r_overflow <= 28'b0; // OF is now empty, all in reg
88            end
89          end else begin
90            if (r_mixed + i_data > MAX_SIGNED) begin // OF
91              r_mixed <= MAX_SIGNED;
92              r_overflow <= r_overflow + (r_mixed + i_data - MAX_SIGNED);
93            end else if (r_mixed + i_data < MIN_SIGNED) begin // UF
94              r_mixed <= MIN_SIGNED;
95              r_overflow <= r_overflow + (r_mixed + i_data - MIN_SIGNED);
96            // in range - try offloading the overflow
97            end else begin
98                  // OF still too big
99              if (r_mixed + i_data + r_overflow > MAX_SIGNED) begin
100               r_mixed <= MAX_SIGNED;
101               r_overflow <= r_mixed + i_data + r_overflow - MAX_SIGNED;
102             // UF still too small
103             end else if (r_mixed + i_data + r_overflow < MIN_SIGNED) begin
104               r_mixed <= MIN_SIGNED;
105               r_overflow <= r_mixed + i_data + r_overflow - MIN_SIGNED;
106             // can get rid off of the OF/UF or equal to 0
107             end else begin
108               r_mixed <= r_mixed + i_data + r_overflow;
109               r_overflow <= 28'b0;
110             end
111           end
112           o_mixed <= 24'b0;
113         end
114
115         o_rdy <= (v_idx === 9 ? 1'b1 : 1'b0);
116         v_idx <= (v_idx === 9 ? 0 : v_idx + 1);
117       end else begin // counter not active - retain old values
118         v_idx <= v_idx;
119         r_mixed <= r_mixed;
120         r_overflow <= r_overflow;
121         o_mixed <= o_mixed;
122         o_rdy <= 1'b0;
123       end
124     end
125
126 endmodule
```

Appendix 2. C and Python code

Listing 7. *midi_control.c*

```
#include <stdio.h>
#include <signal.h>
#include <poll.h>
#include <alloca.h>
#include <alsa/asoundlib.h>

static snd_seq_t* seq;
static snd_seq_addr_t* port;
static volatile sig_atomic_t stop = 0;
static int fpga_fd = 0;

static void fatal(const char* msg, ...)
{
    va_list ap;
    va_start(ap, msg);
    vfprintf(stderr, msg, ap);
    va_end(ap);
    fputc('\n', stderr);
    exit(EXIT_FAILURE);
}

static void sighandler(int sig)
{
    stop = 1;
}

static void check_snd(const char* operation, int err)
{
    if (err < 0)
        fatal("Cannot␣%s␣-␣%s", operation, snd_strerror(err));
}

static void mem_check(void* p)
{
    if(!p)
        fatal("Out␣of␣memory");
}

static void init_seq(void)
{
    int err;
    // open sequencer for reading
    err = snd_seq_open(&seq, "default", SND_SEQ_OPEN_DUPLEX, 0);
    check_snd("open␣sequencer", err);

    err = snd_seq_set_client_name(seq, "keyboard");
    check_snd("set␣client␣name", err);
}

static void create_port(void)
{
    int err;
    err = snd_seq_create_simple_port(seq, "keyboard",
                                     SND_SEQ_PORT_CAP_WRITE |
                                     SND_SEQ_PORT_CAP_SUBS_WRITE,
                                     SND_SEQ_PORT_TYPE_MIDI_GENERIC |
                                     SND_SEQ_PORT_TYPE_APPLICATION);
    check_snd("create␣port", err);
}

static void connect_port(void)
{
    int err;
    // 0 because we do not specify outgoing port
    err = snd_seq_connect_from(seq, 0, port->client, port->port);
    if (err < 0)
        fatal("Cannot␣connect␣from␣port␣%d:%d␣-␣%s",
              port->client, port->port, snd_strerror(err));
}

static unsigned int create_fpga_command(int on,
                                        unsigned char note,
                                        unsigned char velocity)
{
    unsigned int command = 0x0;
    command |= note;
    command <<= 8;
    command |= velocity;
    command |= (on << 15); // the order is important because of the types
```

```
80      //printf("Command is: %x\n", command);
81      return command;
82  }
83
84  static void write_device(unsigned int* command)
85  {
86      printf("Writing a command %x to the /dev/synthesizer\n", *command);
87      if (write(fpga_fd, command, sizeof(unsigned int)) < sizeof(unsigned int))
88          fatal("Could not write command %x to the /dev/synthesizer\n",
89              *command);
90  }
91
92  static void handle_event(snd_seq_event_t* ev)
93  {
94      unsigned int command = 0x0;
95      printf("%3d:%-3d ", ev->source.client, ev->source.port);
96      switch (ev->type) {
97          case SND_SEQ_EVENT_NOTEON:
98              if (ev->data.note.velocity)
99              {
100                 printf("Note on            %2d, note %d, velocity %d\n",
101                         ev->data.note.channel,
102                         ev->data.note.note,
103                         ev->data.note.velocity);
104                 // write to FPGA
105                 command = create_fpga_command(1,
106                                 ev->data.note.note,
107                                 ev->data.note.velocity);
108                 write_device(&command);
109             }
110             else // never triggers?
111             {
112                 printf("Note off           %2d, note %d",
113                         ev->data.note.channel,
114                         ev->data.note.note);
115                 command = create_fpga_command(0,
116                                 ev->data.note.note,
117                                 0x0);
118                 write_device(&command);
119             }
120             break;
121         case SND_SEQ_EVENT_NOTEOFF:
122         {
123             printf("Note off           %2d, note %d, velocity %d\n",
124                     ev->data.note.channel,
125                     ev->data.note.note,
126                     ev->data.note.velocity);
127             command = create_fpga_command(0,
128                             ev->data.note.note,
129                             ev->data.note.velocity);
130             write_device(&command);
131             break;
132         }
133         case SND_SEQ_EVENT_CONTROLLER:
134         {
135             printf("Controller         %2d, param %d, value %d\n",
136                     ev->data.control.channel,
137                     ev->data.control.param,
138                     ev->data.control.value);
139             // Hardcoded param == 108 and value equal 127
140             if (ev->data.control.param == 108 && ev->data.control.value == 127)
141             {
142                 command = create_fpga_command(1, 0, 0);
143                 write_device(&command);
144             }
145             break;
146         }
147         default:
148             printf("Event type: %d\n", ev->type);
149     }
150 }
151
152 int main(int argc, char* argv[])
153 {
154     int err;
155     struct pollfd* pfds;
156     int npfds;
157     if (argc < 2) // argv[1] is the port num to listen on
158         fatal("Please pass a port to listen on!\n");
159
160     // open FPGA device file
161     fpga_fd = open("/dev/synthesizer", O_WRONLY);
162     if (fpga_fd < 0)
163         fatal("Could not open the FPGA char file!\n");
164
```

```
165         // initialize the sequencer object
166         init_seq();
167
168         // choose appropriate port for reading from
169         // firstly allocating buffer for returned addr
170         port = realloc(port, sizeof(snd_seq_addr_t));
171         mem_check(port);
172
173         err = snd_seq_parse_address(seq, port, argv[1]);
174         if (err < 0)
175             fatal("Invalid port %s - %s", argv[1], snd_strerror(err));
176
177         // create the port object
178         create_port();
179
180         // connect the port object to the sequencer
181         // in this case we just connect to a dummy output port
182         connect_port();
183         // set the non-block mode so that the client won't go to sleep
184         // once it fills the queue of sequencer with events
185         err = snd_seq_nonblock(seq, 1);
186         check_snd("set nonblock mode", err);
187
188         printf("Waiting for data at port %d:0.", snd_seq_client_id(seq));
189         printf(" Press Ctrl+C to end.\n");
190         printf("Source Event                Ch  Data\n");
191
192         signal(SIGINT, sighandler);
193         signal(SIGSTOP, sighandler);
194
195         npfds = snd_seq_poll_descriptors_count(seq, POLLIN);
196         pfds = alloca(sizeof(*pfds) * npfds);
197
198         // sequencer obtains event from fd's associated with it, we must allocate
199         // space in userspace for them and then obtain data which is then handled
200         // loop terminates on any error or interrupt signal
201         for (;;)
202         {
203             snd_seq_poll_descriptors(seq, pfds, npfds, POLLIN);
204             if (poll(pfds, npfds, -1) < 0)
205                 break;
206             do {
207                 snd_seq_event_t* event;
208                 err = snd_seq_event_input(seq, &event);
209                 if (err < 0)
210                     break;
211                 if (event)
212                     handle_event(event);
213             } while (err > 0);
214             fflush(stdout);
215             if (stop)
216                 break;
217         }
218
219         close(fpga_fd);
220         snd_seq_close(seq);
221         return 0;
222     }
```

Listing 8. *dma_snd.h*

```
1   #include <linux/kernel.h>
2   #include <linux/module.h>
3   #include <linux/fs.h>
4   #include <linux/dma-mapping.h>
5   #include <linux/interrupt.h>
6   #include <linux/slab.h>
7   #include <linux/cdev.h>
8   #include <linux/platform_device.h>
9   #include <linux/of_address.h>
10  #include <linux/uaccess.h>
11  #include <linux/interrupt.h>
12  #include <linux/wait.h>
13  #include <linux/jiffies.h>
14  #include <linux/hrtimer.h>
15  #include <sound/core.h>
16  #include <sound/control.h>
17  #include <sound/pcm.h>
18  #include <sound/initval.h>
19  #include <asm/io.h>
20
21  #define DEV_NAME                "dma_snd"
22
23  #define MSGDMA_MAP_SIZE         0x30
```

```
24  #define DMA_BUF_SIZE          (1 << 20)
25  // 1 MB -> how big the buffer allocated by the driver is
26  // (max onetime buffer fill without reading)
27
28  /* ALSA constraints for efficient communication
29   *
30   * PCM interrupt interval -> ex: 10ms
31   *  Period -> how many frames per one PCM interrupt
32   * Frame -> 1 sample from all channels, here:
33   * 1 channel * 1 sample in bytes = 1 * 4 = 4 B
34   * Buffer -> holds some periods in ring-like fashion, PCM reads from it
35   */
36
37  #define DMA_TX_PERIOD_MS      10 // 10ms
38
39  // assuming IRQ every 10 ms i.e. 100 in a second
40  #define PERIOD_SAMPLES        960
41  #define PERIOD_SIZE_BYTES     4 * PERIOD_SAMPLES
42  #define MAX_PERIODS_IN_BUF    100
43  #define MIN_PERIODS_IN_BUF    MAX_PERIODS_IN_BUF
44  // The size of buffer in kernel, has to be smaller than DMA_BUF_SIZE
45
46  static int debug = 0;
47  #undef dbg
48  #define dbg(format, arg...) do { if (debug) \
49  pr_info(": " format "\n", ##arg); } while (0)
50  #define dbg_info(format, arg...) do { if (debug == 2) \
51  pr_info(": " format "\n", ##arg); } while (0)
52  #define dbg_timer(format, arg...) do { if (debug == 3) \
53  pr_info(": " format "\n", ##arg); } while (0)
54
55  typedef u32 volatile reg_t;
56
57  #pragma pack(1)
58  struct msgdma_reg {
59      /* CSR port registers */
60      reg_t csr_status;
61      reg_t csr_ctrl;
62      reg_t csr_fill_lvl;
63      reg_t csr_resp_fill_lvl;
64      reg_t csr_seq_num;
65      reg_t csr_comp_conf1;
66      reg_t csr_comp_conf2;
67      reg_t csr_comp_info;
68
69      /* Descriptor port registers */
70      reg_t desc_read_addr;
71      reg_t desc_write_addr;
72      reg_t desc_len;
73      reg_t desc_ctrl;
74
75      /* Response port registers */
76      reg_t resp_bytes_transferred;
77      reg_t resp_term_err;
78  };
79  #pragma pack()
80
81  /* MSGDMA Register bit fields */
82  enum STATUS {
83      IRQ                   = (1 << 9),
84      STOPPED_EARLY_TERM    = (1 << 8),
85      STOPPED_ON_ERR        = (1 << 7),
86      RESETTING             = (1 << 6),
87      STOPPED               = (1 << 5),
88      RESP_BUF_FULL         = (1 << 4),
89      RESP_BUF_EMPTY        = (1 << 3),
90      DESC_BUF_FULL         = (1 << 2),
91      DESC_BUF_EMPTY        = (1 << 1),
92      BUSY                  = (1 << 0),
93  };
94
95  #define STATUS_RESET_ALL    GENMASK(9, 0)
96
97  enum CONTROL {
98      STOP_DESC             = (1 << 5),
99      GLOBAL_IRQ_EN         = (1 << 4),
100     STOP_ON_EARLY_TERM    = (1 << 3),
101     STOP_ON_ERR           = (1 << 2),
102     RESET_DISP            = (1 << 1),
103     STOP_DISP             = (1 << 0),
104 };
105
106 enum DESC_CTRL {
107     GO                    = (1 << 31),
108     WAIT_WRITE_RESP       = (1 << 25),
```

```
109         EARLY_DONE_EN       = (1 << 24),
110         TX_ERR_IRQ_EN       = (1 << 23),
111         EARLY_TERM_IRQ_EN   = (1 << 15),
112         TX_COMPL_IRQ_EN     = (1 << 14),
113         END_ON_EOP          = (1 << 12),
114         PARK_WRITES         = (1 << 11),
115         PARK_READS          = (1 << 10),
116         GEN_EOP             = (1 << 9),
117         GEN_SOP             = (1 << 8),
118         TX_CHANNEL          = (1 << 7),
119     };
120
121     /* Driver private data */
122     struct msgdma_data {
123         dev_t dev_id;
124         struct cdev cdev;
125
126         struct msgdma_reg* msgdma0_reg;
127         int msgdma0_irq;
128         void* dma_buf_rd;
129         dma_addr_t dma_buf_rd_handle;
130
131         // to be removed?
132         struct class *cl;
133
134         struct snd_card* card;
135         struct snd_pcm* pcm;
136         const struct dma_snd_pcm_ops* timer_ops;
137         /* just one substream so keep all data in this struct */
138         struct mutex cable_lock;
139         /* flags */
140         unsigned int running;
141         /* timer stuff */
142         struct hrtimer hr_timer;
143
144         struct snd_pcm_substream* substream;
145         unsigned int buf_pos; /* position in buffer in bytes */
146     };
147
148     /* SND MINIVOSC Data */
149     static struct snd_pcm_hardware dma_snd_pcm_hw = {
150         .info = (SNDRV_PCM_INFO_MMAP |
151         SNDRV_PCM_INFO_INTERLEAVED |
152         SNDRV_PCM_INFO_BLOCK_TRANSFER |
153         SNDRV_PCM_INFO_MMAP_VALID),
154         .formats            = SNDRV_PCM_FMTBIT_S32_LE,
155         .rates              = SNDRV_PCM_RATE_96000,
156         .rate_min           = 96000,
157         .rate_max           = 96000,
158         .channels_min       = 1,
159         .channels_max       = 1,
160         .buffer_bytes_max   = DMA_BUF_SIZE,
161         .period_bytes_min   = PERIOD_SIZE_BYTES,
162         .period_bytes_max   = PERIOD_SIZE_BYTES,
163         .periods_min        = MIN_PERIODS_IN_BUF,
164         .periods_max        = MAX_PERIODS_IN_BUF,
165     };
166
167     /* Function declarations */
168     static int dma_snd_open(struct inode* node, struct file* f);
169     static int dma_snd_release(struct inode* node, struct file* f);
170     //static ssize_t dma_snd_read(struct file* f, char __user* ubuf,
171                                 size_t len, loff_t* off);
172
173     static int dma_snd_probe(struct platform_device* pdev);
174     static int dma_snd_remove(struct platform_device* pdev);
175
176     /* ALSA functions */
177     static int dma_snd_pcm_open(struct snd_pcm_substream* ss);
178     static int dma_snd_pcm_close(struct snd_pcm_substream* ss);
179     static int dma_snd_hw_params(struct snd_pcm_substream* ss,
180                                  struct snd_pcm_hw_params* hw_params);
181     static int dma_snd_prepare(struct snd_pcm_substream* ss);
182     static int dma_snd_pcm_trigger(struct snd_pcm_substream* ss, int cmd);
183     static int dma_snd_pcm_dev_free(struct snd_device* device);
184     static int dma_snd_pcm_free(struct msgdma_data* chip);
185     static snd_pcm_uframes_t dma_snd_pcm_pointer(struct snd_pcm_substream* ss);
186
187     /* timer functions */
188     static void dma_snd_timer_start(struct msgdma_data* dma_snd_dev);
189     static void dma_snd_timer_stop(struct msgdma_data* dma_snd_dev);
190     static void dma_snd_fillbuf(struct msgdma_data* dma_snd_dev);
191     static enum hrtimer_restart dma_snd_timer_handler(struct hrtimer* timer);
192
193     static struct snd_pcm_ops dma_snd_pcm_ops = {
```

```
194         .open         = dma_snd_pcm_open,
195         .close        = dma_snd_pcm_close,
196         .ioctl        = snd_pcm_lib_ioctl,
197         .hw_params    = dma_snd_hw_params,
198         .prepare      = dma_snd_prepare,
199         .trigger      = dma_snd_pcm_trigger,
200         .pointer      = dma_snd_pcm_pointer,
201     };
202
203     /* specifies what function is called at
204        snd_card_free - used in snd_device_new */
205     static struct snd_device_ops snd_dev_ops = {
206         .dev_free     = dma_snd_pcm_dev_free,
207     };
208
209     static const struct file_operations dma_snd_fops = {
210         .owner        = THIS_MODULE,
211         .open         = dma_snd_open,
212         .release      = dma_snd_release,
213        // .read         = dma_snd_read,
214     };
215
216
217     static const struct of_device_id dma_snd_of_match [] = {
218         {.compatible = "altr,msgdma-19.1" },
219         {}
220     };
221
222     static struct platform_driver dma_snd_driver = {
223         .probe        = dma_snd_probe,
224         .remove       = dma_snd_remove,
225         .driver       = {
226             .name = DEV_NAME,
227             .of_match_table = dma_snd_of_match,
228         },
229     };
```

Listing 9. *dma_snd.c*

```
1   #include "dma_snd.h"
2
3   /* Utility functions */
4   static void setbit_reg32(volatile void __iomem* reg, u32 mask)
5   {
6       u32 val = ioread32(reg);
7       iowrite32(val | mask, reg);
8   }
9
10  /*
11  static void clearbit_reg32(volatile void __iomem* reg, u32 mask)
12  {
13      u32 val = ioread32(reg);
14      iowrite32((val & (~mask)), reg);
15  }
16  */
17
18  static void dma_snd_reset(struct msgdma_reg* reg)
19  {
20      dbg("%s", __func__);
21      /* Clear all existing status bits */
22      setbit_reg32(&reg->csr_status, STATUS_RESET_ALL);
23
24      /* Set the resetting bit, wait until deasserted by the device */
25      setbit_reg32(&reg->csr_ctrl, RESET_DISP);
26      while (ioread32(&reg->csr_status) & RESETTING);
27
28      dbg("%s␣done␣resetting|␣status:␣%d␣control␣%d",
29          __func__, reg->csr_status, reg->csr_ctrl);
30      /* Set up transfer descriptors */
31      setbit_reg32(&reg->csr_ctrl,
32          STOP_ON_EARLY_TERM | STOP_ON_ERR | GLOBAL_IRQ_EN);
33
34      dbg("%s␣done␣setting␣control␣bits|␣status:␣%d␣control␣%d",
35          __func__, reg->csr_status, reg->csr_ctrl);
36  }
37
38  static void dma_snd_push_descr(
39      struct msgdma_reg* reg,
40      dma_addr_t read_addr,
41      dma_addr_t write_addr,
42      u32 len,
43      u32 ctrl)
44  {
45      iowrite32(read_addr, &reg->desc_read_addr);
```

```
46      iowrite32(write_addr, &reg->desc_write_addr);
47      iowrite32(len, &reg->desc_len);
48      iowrite32(ctrl | GO, &reg->desc_ctrl);
49  }
50
51  /* ALSA functions */
52  static int dma_snd_pcm_open(struct snd_pcm_substream* substr)
53  {
54      struct msgdma_data* dma_snd_dev = substr->private_data;
55      struct timespec time;
56      dbg("%s", __func__);
57
58      dma_snd_reset(dma_snd_dev->msgdma0_reg);
59      mutex_lock(&dma_snd_dev->cable_lock);
60
61      /* set runtime DMA buffer information */
62      substr->runtime->dma_area = dma_snd_dev->dma_buf_rd;
63      substr->runtime->dma_bytes = DMA_BUF_SIZE;
64      substr->runtime->dma_addr = dma_snd_dev->dma_buf_rd_handle;
65
66      substr->runtime->hw = dma_snd_pcm_hw;
67
68      dma_snd_dev->substream = substr;
69      substr->runtime->private_data = dma_snd_dev;
70
71
72      /* SETUP timer */
73      hrtimer_init(&dma_snd_dev->hr_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
74      dma_snd_dev->hr_timer.function = &dma_snd_timer_handler;
75      dbg("Resolution: %u secs and %u nsecs", time.tv_sec, time.tv_nsec);
76      mutex_unlock(&dma_snd_dev->cable_lock);
77      return 0;
78  }
79
80  static int dma_snd_pcm_close(struct snd_pcm_substream* substr)
81  {
82      struct msgdma_data* dma_snd_dev = substr->private_data;
83      dbg("%s", __func__);
84
85      // even though the mutex will be set to null already, lock it
86      mutex_lock(&dma_snd_dev->cable_lock);
87      substr->private_data = NULL;
88      mutex_unlock(&dma_snd_dev->cable_lock);
89      return 0;
90  }
91
92  static int dma_snd_hw_params(struct snd_pcm_substream* substr,
93                               struct snd_pcm_hw_params* hw_params)
94  {
95      dbg("%s", __func__);
96      return 0;
97  }
98
99  static int dma_snd_prepare(struct snd_pcm_substream* substr)
100 {
101     struct msgdma_data* dma_snd_dev = substr->private_data;
102     dbg("%s", __func__);
103
104     dma_snd_dev->buf_pos = 0;
105     return 0;
106 }
107
108 static int dma_snd_pcm_trigger(struct snd_pcm_substream* substr, int cmd)
109 {
110     int ret = 0;
111     struct msgdma_data* dma_snd_dev = substr->private_data;
112     dbg("%s - trigger %d", __func__, cmd);
113
114     switch (cmd)
115     {
116         case SNDRV_PCM_TRIGGER_START:
117             /* start the hw capture */
118             if (!dma_snd_dev->running)
119             {
120                 /* START the timer */
121                 dma_snd_timer_start(dma_snd_dev);
122             }
123             // add a bitmask for each stream that is running (in our case just one)
124             dma_snd_dev->running |= 1 << substr->stream;
125             break;
126         case SNDRV_PCM_TRIGGER_STOP:
127             /* stop the hw capture */
128             dma_snd_dev->running &= ~(1 << substr->stream);
129             if (!dma_snd_dev->running)
130                 /* STOP the timer */
```

```
131                     dma_snd_timer_stop(dma_snd_dev);
132                 break;
133             default:
134                 ret = -EINVAL;
135         }
136         return ret;
137     }
138
139     // These functions would do any special freeing on snd_card_free,
140     // however no need to do anything since no special allocations made
141     static int dma_snd_pcm_dev_free(struct snd_device* device)
142     {
143         dbg("%s", __func__);
144         return dma_snd_pcm_free(device->device_data);
145     }
146
147     static int dma_snd_pcm_free(struct msgdma_data* chip)
148     {
149         dbg("%s", __func__);
150         return 0;
151     }
152
153     static snd_pcm_uframes_t dma_snd_pcm_pointer(struct snd_pcm_substream* substr)
154     {
155         struct msgdma_data* dma_snd_dev = substr->private_data;
156         snd_pcm_uframes_t pos = 0;
157         dbg_timer("%s␣jiffies␣%d␣buf_pos␣%d", __func__,
158                     jiffies_to_msecs(jiffies), dma_snd_dev->buf_pos);
159         pos = bytes_to_frames(substr->runtime,
160                               dma_snd_dev->buf_pos * PERIOD_SIZE_BYTES);
161         dbg_info("␣␣␣dma_snd_pcm_pointer␣%ld", pos);
162         return pos;
163     }
164
165     /* timer functions */
166     static void dma_snd_timer_start(struct msgdma_data* dma_snd_dev)
167     {
168         dbg_timer("%s", __func__);
169
170         hrtimer_start(&dma_snd_dev->hr_timer,
171                       ms_to_ktime(DMA_TX_PERIOD_MS), HRTIMER_MODE_REL);
172     }
173
174     static void dma_snd_timer_stop(struct msgdma_data* dma_snd_dev)
175     {
176         dbg_timer("%s", __func__);
177         hrtimer_cancel(&dma_snd_dev->hr_timer);
178     }
179
180     enum hrtimer_restart dma_snd_timer_handler(struct hrtimer* timer)
181     {
182         struct msgdma_data* dma_snd_dev = container_of(timer,
183                                                        struct msgdma_data, hr_timer);
184         dbg_timer("%s␣jiffies␣%d␣buf_pos␣%d",
185                     __func__, jiffies_to_msecs(jiffies), dma_snd_dev->buf_pos);
186
187         /* update every DMA_TX_FREQ */
188         hrtimer_forward_now(timer, ms_to_ktime(DMA_TX_PERIOD_MS));
189
190         dma_snd_fillbuf(dma_snd_dev);
191         return HRTIMER_RESTART;
192     }
193
194     static void dma_snd_fillbuf(struct msgdma_data* dma_snd_dev)
195     {
196         struct snd_pcm_runtime* runtime = dma_snd_dev->substream->runtime;
197         dma_addr_t read_addr = runtime->dma_addr
198                                + dma_snd_dev->buf_pos * PERIOD_SIZE_BYTES;
199
200         if (!dma_snd_dev->running)
201             return;
202
203         dbg_info("␣␣␣dma_snd_fillbuf␣buf_pos␣%d␣read_addr␣%x",
204                  dma_snd_dev->buf_pos, read_addr);
205
206         dma_snd_push_descr(
207             dma_snd_dev->msgdma0_reg,
208             0,
209             read_addr,
210             PERIOD_SIZE_BYTES, // write one whole buffer of 4B samples
211             TX_COMPL_IRQ_EN);
212
213         ++dma_snd_dev->buf_pos;
214         if (dma_snd_dev->buf_pos >= MAX_PERIODS_IN_BUF)
215             dma_snd_dev->buf_pos = 0;
```

```
216 }
217
218 /* Character file functions */
219 static int dma_snd_open(struct inode* node, struct file* f)
220 {
221     struct msgdma_data* data;
222     data = container_of(node->i_cdev, struct msgdma_data, cdev);
223     f->private_data = data;
224     return 0;
225 }
226
227 static int dma_snd_release(struct inode* node, struct file* f)
228 {
229     return 0;
230 }
231
232 static irqreturn_t dma_snd_irq_handler(int irq, void* dev_id)
233 {
234     struct msgdma_data* data = (struct msgdma_data*)dev_id;
235     struct msgdma_reg* msgdma0_reg = data->msgdma0_reg;
236
237     dbg_timer("␣␣jiffies␣%u␣dma␣device␣status␣interrupt␣%x␣buf_pos␣%d",
238                 jiffies_to_msecs(jiffies),
239                 msgdma0_reg->csr_status,
240                 data->buf_pos);
241     /* acknowledge corresponding dma and wake up whoever is waiting */
242     if (ioread32(&msgdma0_reg->csr_status) & irq)
243     {
244         setbit_reg32(&msgdma0_reg->csr_status, irq);
245         if (!data->running)
246             goto __eexit;
247
248         snd_pcm_period_elapsed(data->substream);
249     }
250
251 __eexit:
252     return irq_handled;
253 }
254
255 static int dma_snd_register_chrdev(struct msgdma_data* data)
256 {
257     int ret = 0;
258     struct device* dev;
259
260     ret = alloc_chrdev_region(&data->dev_id, 0, 1, dev_name);
261     if (ret < 0)
262     {
263         pr_err("character␣device␣region␣allocation␣failed");
264         goto __error;
265     }
266     // create a class in sysfs to be mounted by udev
267     if (is_err(data->cl = class_create(this_module, "chrdev")))
268     {
269         pr_err("character␣class␣creation␣failed");
270         goto __chrdev_add_err;
271     }
272     if (is_err(dev = device_create(data->cl, null, data->dev_id,
273                 null, "dma_snd")))
274     {
275         pr_err("character␣device␣creation␣failed");
276         class_destroy(data->cl);
277         goto __chrdev_add_err;
278     }
279     // actual registering of the device
280     cdev_init(&data->cdev, &dma_snd_fops);
281     ret = cdev_add(&data->cdev, data->dev_id, 1);
282     if (ret < 0)
283     {
284         pr_err("character␣device␣initialization␣failed");
285         device_destroy(data->cl, data->dev_id);
286         class_destroy(data->cl);
287         goto __chrdev_add_err;
288     }
289
290     return 0;
291 __chrdev_add_err:
292     unregister_chrdev_region(data->dev_id, 1);
293 __error:
294     return ret;
295 }
296
297 static void dma_snd_unregister_chrdev(struct msgdma_data* data)
298 {
299     cdev_del(&data->cdev);
300     device_destroy(data->cl, data->dev_id);
```

```
301         class_destroy(data->cl);
302         unregister_chrdev_region(data->dev_id, 1);
303     }
304
305     /* main functions */
306     static int dma_snd_probe(struct platform_device* pdev)
307     {
308         struct msgdma_data* data;
309         struct resource* res;
310         struct resource* region;
311         struct device* dev;
312
313         struct snd_card* card;
314         int nr_subdevs = 1; // how many capture substreams (by default just 1)
315         struct snd_pcm* pcm;
316         int ret = 0;
317
318         dbg("dma_snd_probe entered");
319
320         dev = &pdev->dev;
321
322         /* alsa part */
323         ret = snd_card_new(dev, 3, "fpga_synth",
324                            this_module,
325                            sizeof(struct msgdma_data), &card);
326         if (ret < 0)
327             goto __nodev;
328
329         data = card->private_data;
330         data->card = card;
331         // must have mutex_init here, else crash on mutex_lock
332         mutex_init(&data->cable_lock);
333
334         dbg("dma_snd_data %p dev_id %d", data, pdev->id);
335
336         strcpy(card->driver, "dma_snd_driver");
337         sprintf(card->shortname, "fpga_synthesizer_%s", dev_name);
338         strcpy(card->longname, card->shortname);
339
340         dbg("dma_snd_card_names_copying_success");
341         ret = snd_device_new(card, sndrv_dev_lowlevel, data, &snd_dev_ops);
342         if (ret < 0)
343             goto __nodev;
344         /* 0 playback, 1 capture substreams  */
345         ret = snd_pcm_new(card, card->driver, 0, 0, nr_subdevs, &pcm);
346         if (ret < 0)
347             goto __nodev;
348
349         snd_pcm_set_ops(pcm, sndrv_pcm_stream_capture, &dma_snd_pcm_ops);
350         // it should be the dev/card struct (the one containing snd_card* card)
351         // -> this will not end up in substream->private_data
352         pcm->private_data = data;
353         pcm->info_flags = 0;
354         strcpy(pcm->name, card->shortname);
355         dbg("dma_snd_snd_pcm_set_ops_success");
356
357         ret = snd_card_register(card);
358         if (ret < 0)
359             goto __nodev;
360
361         /* dma part */
362
363         platform_set_drvdata(pdev, (void*)data);
364
365         data->dma_buf_rd = dma_alloc_coherent(
366             dev,
367             dma_buf_size,
368             &data->dma_buf_rd_handle,
369             gfp_kernel);
370
371         if (data->dma_buf_rd == null)
372         {
373             ret = -enomem;
374             goto __fail;
375         }
376
377         /* remap io region of the device */
378         /* obtain the resource structure containing
379             start, end and io memory size */
380         res = platform_get_resource(pdev, ioresource_mem, 0);
381         if (res == null)
382         {
383             dev_err(&pdev->dev, "io_region_resource_not_defined");
384             return -enodev;
385         }
```

```
386
387         /* request the region from the memory to guarantee exclusiveness */
388         region = devm_request_mem_region(
389             dev,
390             res->start,
391             resource_size(res),
392             dev_name(dev));
393         if (region == null)
394         {
395             dev_err(dev, "mem region not requested");
396             return -ebusy;
397         }
398
399         /* map the region to memory */
400         data->msgdma0_reg = devm_ioremap_nocache(dev,
401                                     region->start, msgdma_map_size);
402         if (data->msgdma0_reg <= 0)
403         {
404             dev_err(dev, "could not remap io region");
405             return -efault;
406         }
407
408         /* initialize the device itself */
409         dma_snd_reset(data->msgdma0_reg);
410
411         /* get device's irq number(s) */
412         data->msgdma0_irq = platform_get_irq(pdev, 0);
413         if (data->msgdma0_irq < 0)
414         {
415             pr_err("could not get irq number");
416             return -enxio;
417         }
418
419         ret = devm_request_irq(dev, data->msgdma0_irq,
420                                dma_snd_irq_handler, irqf_shared,
421                                "msgdma0", data);
422         if (ret < 0)
423         {
424             dev_err(dev, "could not request irq %d", data->msgdma0_irq);
425             return ret;
426         }
427
428         ret = dma_snd_register_chrdev(data);
429         if (ret < 0)
430             return ret;
431
432         dbg("dma probe exit");
433         return 0;
434
435
436     __nodev:
437         dbg("__nodev reached!!");
438         snd_card_free(card); // this will call .dev_free registered func
439
440     __fail:
441         dbg("__fail reached!!");
442         dma_snd_remove(pdev);
443         return ret;
444     }
445
446     static int dma_snd_remove(struct platform_device* pdev)
447     {
448         struct msgdma_data* data = (struct msgdma_data*)platform_get_drvdata(pdev);
449
450         snd_card_free(data->card);
451
452         dma_snd_unregister_chrdev(data);
453         dma_free_coherent(
454             &pdev->dev,
455             dma_buf_size,
456             data->dma_buf_rd,
457             data->dma_buf_rd_handle);
458         return 0;
459     }
460
461     static int __init dma_snd_init(void)
462     {
463         return platform_driver_register(&dma_snd_driver);
464     }
465
466     static void __exit dma_snd_exit(void)
467     {
468         platform_driver_unregister(&dma_snd_driver);
469     }
470
```

```
471 subsys_initcall(dma_snd_init);
472 module_exit(dma_snd_exit);
473
474 MODULE_LICENSE("GPL");
475 MODULE_AUTHOR("Jakub Duchniewicz, j.duchniewicz@gmail.com");
476 MODULE_DESCRIPTION("DMA receiver driver acting as ALSA Hardware Source");
477 MODULE_VERSION("1.0");
```

Listing 10. *generate_svf_coeff.py*

```
1  import argparse
2  import math
3
4  class Generator:
5      def __init__(self, args):
6          self.args = args
7          self.sampling_speed = int(self.args.s)
8          self.k = 2
9
10
11     def generate_coeff(self):
12         out = open(self.args.d, 'w')
13         for i in range(128):
14             print(i)
15             freq = 2**((i - 69)/12) * 440
16             print(freq)
17             curve_val = ((math.exp((2.5 * i)/127) - 1)/(math.exp(2.5) - 1))
18             print(curve_val)
19             cutoff_freq = curve_val * 32*10**3 # rescale to 1-32KHz
20             print(cutoff_freq)
21             g = math.tan((cutoff_freq/(self.sampling_speed * 10**3)) * math.pi)
22             print(g)
23             m = 1 + g*(g+self.k)
24             a1 = 1/m
25             a2 = g*a1
26             a3 = g*a2
27             print("m:{}, a1:{}, a2:{}, a3:{}".format(m,a1,a2,a3))
28             # now convert to Q2.37 format for precision
29             a1_q = int(a1 * 2**37)
30             a2_q = int(a2 * 2**37)
31             a3_q = int(a3 * 2**37)
32
33             if self.args.q is not False:
34                 midi_hex = '{0:0{1}x}'.format(i, 2)
35                 a1_bin = '{0:0{1}b}'.format(a1_q, 40)
36                 a2_bin = '{0:0{1}b}'.format(a2_q, 40)
37                 a3_bin = '{0:0{1}b}'.format(a3_q, 40)
38
39                 line = "7'h{} \t: \t begin \n".format(midi_hex)
40                 line += "\to_a1 <= 40'b{};\n".format(a1_bin)
41                 line += "\to_a2 <= 40'b{};\n".format(a2_bin)
42                 line += "\to_a3 <= 40'b{};\n".format(a3_bin) # later add spacing between n and m values '_'
43                 line += "\tend\n"
44                 print(line)
45                 out.write(line)
46             else:
47                 print("in_Q2.32 a1:{}, a2:{}, a3:{}".format(a1_q,a2_q,a3_q))
48
49
50 def main():
51     parser = argparse.ArgumentParser()
52     parser.add_argument('-s', help='sampling speed in kHz', required=True)
53     parser.add_argument('-q', help='generate verilog ready lines of sequential logic', action="store_true")
54     parser.add_argument('-d', help='output file', required=True)
55     args = parser.parse_args()
56     gen = Generator(args)
57     gen.generate_coeff()
58
59 if __name__ == "__main__":
60     main()
```

Listing 11. *generate_wave.py*

```
1  import argparse
2  import math
3
4  # dependencies:
5  # number of samples cause the output index width
6  # output bits width cause the sine wave values
7  class Generator:
8      def __init__(self, args):
9          self.args = args
```

```
10          self.output_width = int(self.args.r)
11          self.output_max_size = 2 ** int(self.args.r)
12          self.output_width_hex_len = int(self.args.r) // 4 # 4 bits is 1 hex value
13          self.table_size = int(self.args.n)
14          self.input_width = int(math.log2(self.table_size))
15          self.table_size_hex_len = int(math.ceil(math.log2(self.table_size) / 4))
16
17      def generate(self):
18          if 'sine' in self.args.t:
19              self.generate_sine()
20          elif 'triangle' or 'sawtooth' in self.args.t:
21              self.generate_ramp()
22
23      def generate_sine(self):
24          out = open(self.args.d, 'w')
25          if self.args.q is not False:
26              self.generate_qnotation_sine(out)
27          else:
28              self.generate_quarter_sine(out)
29
30      # generate a quarter of a triangle OR
31      # generate the positive half of the sawtooth
32      def generate_ramp(self):
33          out = open(self.args.d, 'w')
34          self.output_width -= 1
35          for sample in range(self.table_size):
36              val = int(sample / self.table_size * 2**self.output_width)
37              val_bin = '{0:0{1}b}'.format(val, self.output_width) # format specifier
38
39              if self.args.g is not False:
40                  idx_hex = '{0:0{1}x}'.format(sample, self.table_size_hex_len)
41
42                  lhs = "{0}'h{1}".format(self.input_width, idx_hex)
43                  rhs = "␣\t:\to_val␣<=␣{0}'b0{1};\n".format(self.output_width + 1, val_bin) #leading zero because it is positive only
44
45                  lhs += rhs
46                  out.write(lhs)
47              else:
48                  out.write(str(val) + '\n')
49
50      # always generate quarter of a wave for Q format
51      def generate_qnotation_sine(self, out):
52          self.output_width -= 1
53          for sample in range(self.table_size):
54              rad = ((2 * sample + 1) / (2 * self.table_size * 4)) * 2 * math.pi # according to zipCPU, take quarter of full wave
55              sine = math.sin(rad)
56              qnotation_sine = int(2**(self.output_width) * sine)
57              sine_bin = '{0:0{1}b}'.format(qnotation_sine, self.output_width) # format specifier
58
59              if self.args.g is not False:
60                  idx_hex = '{0:0{1}x}'.format(sample, self.table_size_hex_len)
61
62                  lhs = "{0}'h{1}".format(self.input_width, idx_hex)
63                  rhs = "␣\t:\to_val␣<=␣{0}'b0{1};\n".format(self.output_width + 1, sine_bin) #leading zero because it is just a quarter
64
65                  lhs += rhs
66                  out.write(lhs)
67              else:
68                  out.write(str(sine) + '\n')
69
70
71      def generate_quarter_sine(self, out):
72          self.table_size >>= 2
73          self.output_width -= 1
74          for sample in range(self.table_size):
75              rad = ((2 * sample + 1) / (2 * self.table_size * 4)) * 2 * math.pi # according to zipCPU, take quarter of full wave
76              sine = math.sin(rad)
77              sine_bin = '{0:0{1}b}'.format(int(self.output_max_size * sine), self.output_width) # format specifier
78
79              if self.args.g is not False:
80                  idx_hex = '{0:0{1}x}'.format(sample, self.table_size_hex_len)
81
82                  lhs = "{0}'h{1}".format(self.input_width, idx_hex)
83                  rhs = "␣\t:\to_val␣<=␣{0}'b0{1};\n".format(self.output_width + 1, sine_hex)
84
85                  lhs += rhs
86                  out.write(lhs)
87              else:
88                  out.write(str(sine) + '\n')
89
90  def main():
91      parser = argparse.ArgumentParser()
92      parser.add_argument('-t', choices=['sine', 'triangle', 'sawtooth'], help='type␣of␣wave␣to␣generate', required=True)
93      parser.add_argument('-n', choices=['512', '1024', '2048', '4096', '8192', '16384', '32768'], help='number␣of␣samples␣to␣be␣generated', required=True)
94      parser.add_argument('-g', help='generate␣verilog␣ready␣lines␣of␣sequential␣logic', action="store_true")
```

```
95      parser.add_argument('-q', help='generate table in Q0.<n> notation', action="store_true")
96      parser.add_argument('-d', help='output file', required=True)
97      parser.add_argument('-r', default='16', help='output bits width')
98      args = parser.parse_args()
99      gen = Generator(args)
100     gen.generate()
101
102 if __name__ == "__main__":
103     main()
```

Listing 12. *generate_tuning_word.py*

```
1   import argparse
2   import math
3
4   class Generator:
5       def __init__(self, args):
6           self.args = args
7           self.tw_bits = int(args.n)
8           self.sampling_speed = float(self.args.s)
9
10      def generate_tw(self):
11          out = open(self.args.d, 'w')
12          for i in range(128):
13              print("number " + str(i))
14              freq = 2**((i - 69)/12) * 440
15              print("freq " + str(freq))
16              tw = int(freq * 2 ** self.tw_bits / (self.sampling_speed * 10 ** 3))
17              print("tw " + str(tw))
18
19              if self.args.q is not False:
20                  midi_hex = '{0:0{1}x}'.format(i, 2)
21                  tw_bin = '{0:0{1}b}'.format(tw, self.tw_bits)
22                  line = "7'h{0} \t:\to_tw <= {1}'b{2};\n".format(midi_hex, self.tw_bits, tw_bin)
23                  out.write(line)
24              else:
25                  out.write(str(tw))
26
27  def main():
28      parser = argparse.ArgumentParser()
29      parser.add_argument('-n', help='number of bits used for sine calculation', required=True)
30      parser.add_argument('-s', help='sampling speed in kHz', required=True)
31      parser.add_argument('-q', help='generate verilog ready lines of sequential logic', action="store_true")
32      parser.add_argument('-d', help='output file', required=True)
33      args = parser.parse_args()
34      gen = Generator(args)
35      gen.generate_tw()
36
37  if __name__ == "__main__":
38      main()
```