Sound Synthesis Using Programmable System-On-Chip Devices

Larry Fitzgerald MSEE, M.Sc.

Department of Computer Science

National University of Ireland, Maynooth, Co. Kildare, Ireland

Maynooth University
National University of Ireland Maynooth

A Thesis submitted in fulfilment of the requirements for the degree of

Master of Science

Supervisor: Dr. Joseph Timoney

October 2019

Declaration

I confirm this is my own work and the use of all material from other sources has been properly cited and fully acknowledged.

Signature: [illegible] Date: October 2019

Acknowledgements

Many thanks to Dr. Joseph Timoney, NUIM for his support, encouragement and advice, and to my wife, Helen, for her forbearance and support.

Glossary

Additive Synthesis	Additive synthesis generates sound by summing the output of multiple sine wave generators, with each generator representing a partial of the tone to be generated.
ADSR envelope	An envelope comprising four phases: attack, decay, sustain, and release.
Attack time	The time taken for the initial transient of the ADSR Envelope. The initial transient begins when a keyboard key is pressed, starting from a zero level and ends when the Envelope reaches its peak value
Control voltage	The term CV refers to the control voltages used in the analogue instruments used to drive the oscillator pitch and alter the synthesis parameter values.
Decay time	The time taken for the second transient of the ADSR Envelope. The second transient begins at the Envelope's peak level, and ends when the envelope reaches a level which is sustained for as long as the key remains pressed.
Envelope	In the context of sound and music, an envelope describes how a sound changes over time
Flash memory	An electronic non-volatile computer storage medium that can be electrically erased and reprogrammed.
Flow Control	A concept from the world of packet-switched networks: it controls the traffic from a particular sender to a receiver and prevents the receiver from being overwhelmed by the data.
Frequency Modulation Synthesis	Frequency modulation involves modulating the frequency of an oscillator in accordance with the amplitude of a modulating signal. For synthesizing harmonic sounds, the modulating signal must have a harmonic relationship to the original carrier signal. For synthesizing sounds with inharmonic bell-like and percussive spectra, modulators with frequencies that are non-integer multiples of the carrier signal are used.
Gate-trigger	The gate-trigger turns filter and amplifier ADSR envelopes on and enacts the release when the keyboard key is raised, and the note is finished.
Glissando	A smooth glide from one pitch to another
Harmonic Partial	A harmonic is any member the set of frequencies that are positive integer multiples of a common fundamental frequency. A harmonic partial is any real partial component of a complex tone that matches (or nearly matches) an ideal harmonic.
Inharmonic Partial	An inharmonic partial is any partial whose frequency is not a positive integer multiple of a common fundamental frequency.
Microcontroller	A microprocessor that, in addition to the CPU, also includes RAM, ROM/FLASH memory, digital I/O, analogue to digital converters, and comparators on a single die/package.
MIDI Controller	a hardware or software device that generates and transmits MIDI data to MIDI-enabled devices, typically to trigger sounds and control parameters of an electronic music performance.
MIDI Keyboard	a piano-style electronic musical keyboard, often with other buttons, wheels and sliders, used for sending MIDI signals or commands to MIDI-enabled devices.

MIDI Message	A MIDI message comprises one of 128 MIDI Commands, followed by 0, 1 or 2 8-bit parameters. It is part of the MIDI protocol which transfers musical information between keyboards and synthesiser.
Mutex	A mutex is locking mechanism used to synchronize access to a shared resource. Only one task at a time can acquire the mutex, and hence the resource. This prevents corrupting the resource's data
Overtone	Any frequency greater than the fundamental frequency of a sound.
Partial	A partial is any of the sine waves of which a complex tone is composed, not necessarily with an integer multiple of the lowest harmonic.
Ping-pong buffer	A ping-pong buffer is a double-buffering technique that uses two buffers to feed a single endpoint: while one buffer is being read, the other is being filled.
Pitch	Pitch is an auditory sensation in which a listener assigns musical tones to relative positions on a musical scale based primarily on their perception of the frequency of vibration.
Pragma	A computer language construct that specifies how a compiler should process its input.
Release time	The time taken for the final transient of the ADSR Envelope. The final transient comprises an Envelope decay from sustain level to zero. It begins when the key is released, and ends when the Envelope has decayed to zero.
Sustain level	The constant level which is maintained from the end of the Decay Time until the key is released.
Subtractive Synthesis	Subtractive synthesis is a method of sound synthesis in which partials of an audio signal are attenuated by a filter to alter the timbre of the sound.
Switching Fabric	A network topology in which network nodes interconnect via one or more network switches
Sub-oscillator	An oscillator derived from a synthesiser's main oscillator by dividing its frequency by 2 or 4.
Synthesiser	An electronic musical instrument that generates audio signals. These signals may be shaped and modulated by components such as filters, envelopes, and low-frequency oscillators.
Timbre	The timbre of musical instruments can be considered in the light of Fourier theory to consist of multiple harmonic or inharmonic partials or overtones. Each partial is a sine wave of different frequency and amplitude that swells and decays over time due to modulation from an ADSR envelope or low frequency oscillator.
Tremolo	A modulation effect that changes the amplitude of a waveform in a periodic manner
Vibrato	A modulation effect that changes the frequency, and hence the pitch of a waveform in a periodic manner
Voltage control	An analogue method of controlling a module. For example, a voltage-controlled filter (VCF) is an electronic filter whose operating characteristics (primarily cut-off frequency) can be set by an input control voltage
Wavetable Synthesis	Wavetable synthesis is based on periodic reproduction of an arbitrary, single-cycle waveform.

List of Abbreviations

AAF	Anti-aliasing filter
ADC	Analogue to digital converter
ADSR	Attack, decay, sustain, release
ADSRV	Attack, decay, sustain, release, volume
AM	Amplitude modulation
AMBA	ARM advanced microcontroller bus architecture
API	Application program interface
ARM	Acorn RISC machine
AWS	Additive waveform synthesis
AXI	Advanced extensible interface
BASIC	Beginners' all-purpose symbolic instruction code
CORDIC	Coordinate rotation digital computer
CPU	Central processor unit
CSV	Comma-separated value
CV	Control voltage
DAC	Digital to analogue converter
DC	Direct current
DDR	DDR SDRAM is a double data rate synchronous dynamic random-access memory
DDS	Direct digital synthesis
DFB	Digital filter processor
DIY	Do-it-yourself
DMA	Direct memory access
DRAM	Dynamic random-access memory
DSP	Digital signal processing
FBD	Functional block diagram
FET	Field-effect transistor
FIFO	First-in, first-out memory
FM	Frequency modulation
FOH	First-order hold
FPAA	Field programmable analogue array
FPGA	Field-programmable gate array
GDS	Global data structure
GUI	Graphical user interface
HLS	High-level synthesis
IC	Integrated circuit
IDE	Integrated design environment
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite impulse response
IO	Input-output
IP	Input
ISR	Interrupt service routine
JSON	Javascript object notation
LFO	Low-frequency oscillator
LPF	Low-pass filter
LUT	Look-up table
MIDI	(Musical Instrument Digital Interface) a technical standard that describes a communications protocol, digital interface, and electrical connectors that connect a wide variety of electronic musical instruments, computers, and related audio devices for playing, editing and recording music.
MIT	Massachusetts Institute of Technology

MSEE	Master of Science in Electrical Engineering
NCO	Numerically-controlled oscillator
NPN	One of the two types of bipolar transistors
NUIM	National University of Ireland, Maynooth
OTA	Operational transconductance amplifier
PBASIC	Microcontroller-based version of BASIC
PC	Personal computer
PCB	Printed circuit board
PIC	Family of Harvard architecture microcontrollers
PL	Programmable logic
PRU	Programmable real-time unit
PS	Processing system
PSOC	(Programmable System-on-Chip): a family of microcontrollers
PWM	Pulse-width modulation
RAM	Random access memory
RCA	Radio Corporation of America
RISC	Reduced instruction-set computer
ROM	Read-only memory
RTL	Register transfer language
SAR	Successive approximation register
SD	Secure Digital, a proprietary non-volatile memory card format
SDRAM	Synchronous dynamic random access memory
SNR	Signal to noise ratio
SPI	Serial peripheral interface
SPICE	SPICE is a general-purpose, open-source analogue electronic circuit simulator
SRAM	Static random access memory
TDM	Time-division multiplexing
UART	Universal asynchronous receiver/transmitter
USB	Universal serial bus
VCA	Voltage-controlled amplifier
VCF	Voltage-controlled filter
VCO	Voltage-controlled oscillator
ZOH	Zero-order hold

Abstract

The last 20 years has witnessed a resurgence of interest in analogue synthesisers[1]. Manufacturers, such as Moog and Sequential Circuits, that had disappeared from the commercial marketplace by the end of the 1980's, have reappeared with an impressive line of products. Other established companies such as Korg and Roland, as well as entrants that had made their name with digital technology, such as Novation and Arturia, have released analogue instruments. Although the feature set of digital synthesisers is extensive and with a falling comparative cost, the analogue market has continued to grow with more and more devices coming available. They are perceived to be of superior sound quality by users, but their primary drawback is price, as numerous discrete components or specialist integrated circuits are required.

This thesis introduces two novel low-cost approaches to building analogue-type synthesisers. Such a low-cost instrument could have applications in an educational laboratory environment for synthesisers. The first approach is to exploit a new mixed-signal technology called the Programmable System-on-Chip (PSoC), which includes a CPU core and mixed-signal arrays of configurable integrated analogue and digital peripherals. The second exploits a System on Chip (SoC) comprising an ARM-based (Acorn RISC Machine) processor and a Field-Programmable Gate Array (FPGA).

Two synthesisers were built and were evaluated for difficulty of implementation and assessed for their sound quality. The design and testing process was recorded and documented in detail. The mixed-signal approach was found to be cheaper than the FPGA-approach both in terms of component costs and development time compared to the FPGA-based approach. Actually, the FPGA-approach was determined to be prohibitively expensive in terms of the development time incurred. The sound quality analysis demonstrated that both instruments were perceived by users to be of high quality, achieving a noticeable analogue sound. Future work would be to repackage the PSoC system and modules into rack-mounted form for use in an educational synthesiser laboratory environment.

Keywords: synthesiser, system on chip, subtractive synthesis, additive synthesis

[1] A syntheziser is an electronic musical instrument that generates audio signals. These signals may be shaped and modulated by components such as filters, envelopes, and low-frequency oscillators.

Table of Contents

1. Introduction and Background Work

Two synthesisers were built and evaluated for sound quality and difficulty of implementation:

- A PSoC-based subtractive synthesiser[2] using mixed-signal arrays to implement some functions and to control other off-board analogue functions. Apart from using a Numerically Controlled Oscillator as part of a note generator, the signal path is entirely analogue. The PSoC development board costs approximately €12.

- A Zynq-7000 SoC additive synthesiser[3] using its FPGA to implement the signal-processing functions. Apart from using an analogue antialiasing filter, the signal path is entirely digital. The Zynq-7000 development board costs approximately €200.

1.1 Project Goals

There were several goals for the project:

- Produce a design for a cheap modular synthesiser suitable for experimentation, which emulates the sounds of the classic synthesisers of the eighties.
- Compare and contrast the two approaches from two points of view:
  - Performance/cost
  - Difficulty of implementation

Other solutions exist, for example Field Programmable Analogue Arrays[26], but are either incomplete or else based on technology that is not publicly available. The emphasis here is to produce a low-cost, tractable solution which can readily adapted/extended by others.

1.1.1 Criteria for success

The following criteria will be used to assess success in meeting the above objectives:

- inexpensive – within the means of the amateur experimenter.
- modular – modules can be used as building blocks for a synthesiser design
- easy to implement - within the capability of the amateur experimenter
- 'analogue sound' – high-quality sound, reminiscent of the classic analogue synthesisers of the 1980's, as judged by a listening panel.

[2] Subtractive synthesis is a method of sound synthesis in which partials of an audio signal are attenuated by a filter to alter the timbre of the sound.
[3] Additive synthesis generates sound by summing the output of multiple sine wave generators, with each generator representing a partial of the tone to be generated.

1.2 Music Synthesisers

1.2.1 Synthesiser

A synthesiser is a musical instrument that uses electronic circuits or software to facilitate the generation of a variety of timbres[4] according to a synthesis algorithm around which the instrument is designed. Unlike traditional acoustic instruments whose sound is defined by their shape, dimensions, materials or playing techniques, these instruments are much more flexible in that the range of possible sounds is not limited by its physical (or virtual) footprint but rather the complexity of the synthesis algorithm that drives them. The 20th century saw the development of synthesisers from single purpose sound generators, to expansive modular synthesis systems, to fully software-based instruments. There are many techniques for synthesis with some being very well-known through commercial products whilst others are solely experimental.

The first well-known technique was subtractive synthesis which was implemented in the analogue synthesisers of the 60's, 70's and 80's. Its popularity then shrank as it was replaced by new techniques such as FM synthesis that appeared in the early digital synths of the 80's, but it returned with the surging popularity of electronic dance music in the 90's. This saw the introduction of hardware instruments that used Digital Signal Processing (DSP) to perform the synthesis. Around the same time software instruments first appeared that eventually could be integrated into the music production software packages. An analogue revival happened in the early 2000's as a market had grown around aficionados seeking out vintage instruments and many of the old manufacturers were revitalized. Today, musicians have a wide range of choices regarding subtractive synthesisers, but in general, digital instruments are much cheaper than their analogue counterparts.

A complimentary technique is additive synthesis whose principles are the opposite of the subtractive synthesiser. These instruments have a much different history commercially. The primary difficulty is that these synthesisers create a timbre by combining many simpler sounds together. This means that they require many sound generators to make a sophisticated timbre. As a result, a physically large instrument is required to contain all the necessary oscillators. The first programmable synthesisers were the RCA Mark I and II [70] which were built in the mid-1950s. This type offered analogue additive synthesis and a form of subtractive synthesis. However, the instrument was the size of a whole room, though this was really because these first instruments were prototypes. Two issues held back additive synthesisers. Firstly, many oscillator circuits were required to generate interesting sounds and secondly, a large interface was needed to interact with the set of parameters for each oscillator. It was not until the late 70's early 80's that digital additive synthesis was possible but no successful commercial product ensued. By the 90's Kawai [71] had released a couple of instruments, but the lack of a suitable interface was limiting. As the personal computer (PC) platforms became more powerful the possibilities for extensive software instruments were realized and products such as Razor [72] (released by Native Instruments) have appeared.

This thesis is concerned with two implementations of synthesiser, each using low-cost off-the-shelf platforms:

- one uses a microcontroller with a set of analogue signal-processing modules; this uses additive synthesis.

[4] The timbre of musical instruments can be considered in the light of Fourier theory to consist of multiple harmonic or inharmonic partials or overtones. Each partial is a sine wave of different frequency and amplitude that swells and decays over time due to modulation from an ADSR envelope or low frequency oscillator.

- the second uses a microcontroller with an on-board FPGA for signal processing; this uses Subtractive Synthesis

First of all, the principles behind both synthesis techniques will be introduced.

1.2.2 Principles of Subtractive Synthesis

The basic idea behind subtractive synthesis is to first mix the outputs of multiple oscillators with a rich harmonic content that are in tune, or detuned, in the same octave, or different octaves. Typically these are sawtooth waves, square wave and triangle waves. This combined waveform is then passed through a filter, often a low pass filter, that will remove, or enhance certain components in the sound depending on its cut-off frequency and resonance peak. Furthermore, this cut-off can be altered dynamically using an attack, decay, sustain, release (ADSR) envelope. The attack is triggered by pressing a key on the keyboard, the decay and sustain happen automatically, and the release is triggered by releasing the key. This filter output is then put through an amplification element that can also alter the volume of the sound dynamically using another ADSR envelope. Other modifiers can be introduced such as low frequency oscillators to create periodic modulations of any parameters such as the volume, the oscillator pitch[5], or the filter cut-off. Additionally, extra oscillators such as a sub-oscillator[6] or a noise generator[7] can be included to extend the range of timbres which can be produced. Figure 1 shows the basic blocks and signal flow of the subtractive synthesiser. The term CV refers to the control voltages used in the analogue instruments used to drive the oscillator pitch and alter the synthesis parameter values. The gate-trigger turns filter and amplifier ADSR envelopes on and enacts the release when the keyboard key is raised, and the note is finished.

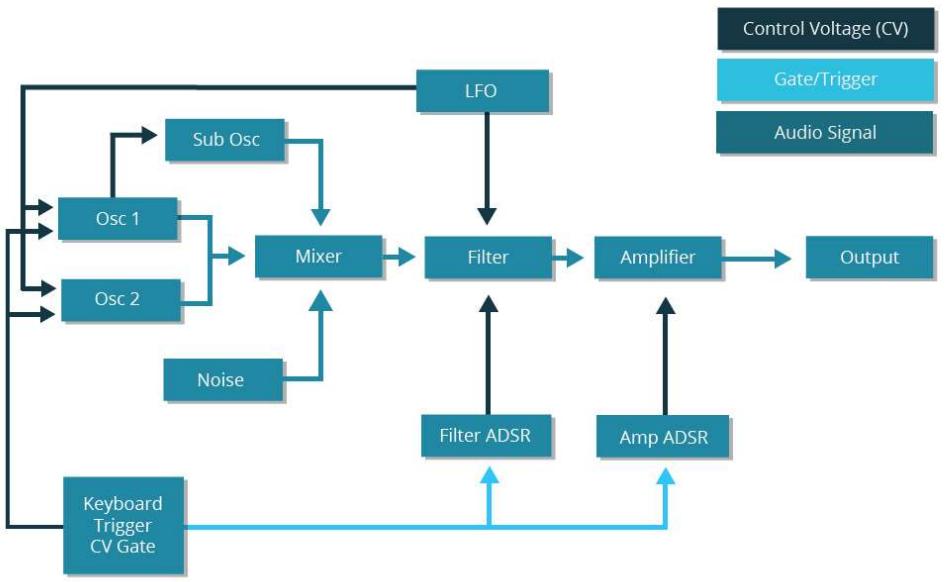


Figure 1 – Subtractive synthesis signal flow [38]

[5] Pitch is an auditory sensation in which a listener assigns musical tones to relative positions on a musical scale based primarily on their perception of the frequency of vibration.
[6] An oscillator derived from a synthesizer's main oscillator by dividing its frequency by 2 or 4.
[7] A noise generator generates white or coloured noise which can be used to emulate breathing effects when synthesising a wind instrument.

1.2.3 Principles of Additive Synthesis

Additive synthesis uses combinations of simple sinewaves to create more complicated tone colours or 'timbres'. This is because a sinewave is the purest waveform, in that it only contains the fundamental, and thus it is used as the basic building block of additive synthesis. A block diagram is given in Figure 2. Here, the synthesiser is represented as a bank of N sinewave oscillators. Each oscillator receives a value for amplitude and frequency. They are then simply added to produce the complex wave output. Further elaborations can be made to this system, but the basic principle holds.

For harmonic, pitched sounds the relationship between the sinewaves is that the frequency of the second is double the frequency of the first, the third three times that of the fundamental, and so on. For in-harmonic[8] sounds, the relationship between the sinewaves is non-integer.

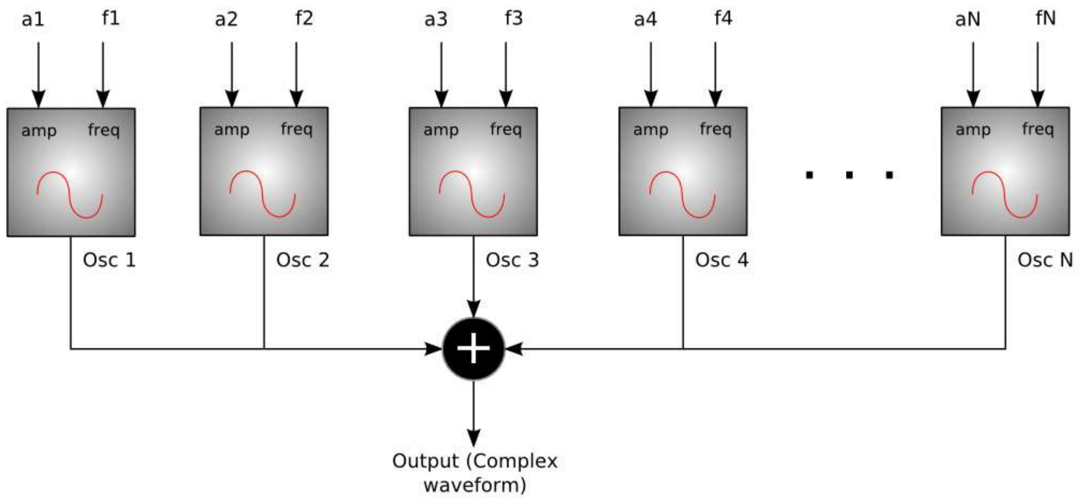


Figure 2 - Additive Synthesis Block [40]

1.3 ELECTRONICS, SYNTHESISERS, AND DIY

The history of electronic instruments is linked with a do-it-yourself electronics subculture that grew up alongside it [35]. This was driven by the availability of cheap components which were surplus to requirements following the end of World War II, and by the fact that electronic equipment for composers and musicians was prohibitively expensive. The first locales with facilities for electronic musicians were associated with film, television and radio studios; these were not accessible by the general public. However, you could make your own electronic music without having access to a special studio as long as you had technological know-how. During the 50's and 60's, companies appeared in the US that sold kits to hi-fi and amateur radio enthusiasts that wanted to build their own equipment, e.g. Heathkit [41] and Dynaco [42], and in the UK, specialist magazines such as Wireless World and Practical Electronics that addressed the same market were published. Wireless World emphasised audio projects, such as those from John Linsley Hood [43].

By the early 70's affordable commercially made electronic instruments arrived, such as the Minimoog synthesiser, which led to their popularity spreading quickly among non-technical musicians. Also, hobbyist magazines were a popular source for circuit designs, such as Wireless World's Electronic Sound Synthesiser [73], and Practical Electronics' Minisonic [74]. The end of the 1970's was a turning point in synthesiser designs as the first instrument with full microprocessor control appeared: the Sequential Circuits Prophet 5. Within a few years Sequential Circuits had introduced the MIDI protocol for communication with electronic instruments that then became, and still is, an industry standard. This meant that interfacing between a PC and an instrument was achievable, and therefore that software could be written to provide desired musical interactions.

Kit building began to go into decline though from that time for a number of reasons: the popularity of the PC, the falling price of consumer electronic devices, the shift to surface mount components which are difficult to hand-solder, and popularity of Application Specific Integrated Circuits (ASICs), which meant that fewer designs were based solely on standard off-the-shelf components that could be obtained by amateurs.

During the 90's, commercial companies introduced synthesisers which used Digital Signal Processing techniques. However, a nostalgia appeared for older the analogue equipment as many asserted that the new machines just did not sound as good, hence the revival of interest in analogue subtractive synthesisers using analogue voicing circuits.

1.3.1 Analogue and Digital Hardware

The hardware revival has not solely been concerned with implementing new versions of the older analogue technology; there has been a number of digital instruments and effects as well. A well-made digital instrument can lead to as much musical creativity and timbral satisfaction as an analogue one.

1.3.2 Microcontroller technologies

In the 80's microcontrollers could only be programmed using assembly language or C. Using a microcontroller in a project necessitated the purchase the bare ICs and building a lot of supporting circuitry. The introduction of Flash-based microcontrollers made the devices convenient to program, and the introduction of low-cost development boards made them available to hobbyists. A description of some of these follows:

1.3.2.1 BASIC STAMP and PIC Microcontrollers

The first proper microcontroller board for hobbyists came in the 1990's with the introduction of the Beginners' All-purpose Symbolic Instruction Code (BASIC) STAMP[44], a microcontroller with a small, specialized BASIC interpreter (PBASIC) built into its Read-Only Memory (ROM). The Peripheral Interface Controller[45] (PIC) microcontroller family was also released in the early 1990's This has a reprogrammable flash ROM can be programmed in assembly language. The BASIC Stamp and the PIC have been used by a number of hobbyists to build MIDI controllers[9], and the latter has been used to build MIDI-to-CV converters to control analogue synthesiser circuits. Neither the Basic STAMP nor the medium range PICs (such as the PIC16F84) were particularly suited to simultaneous MIDI reception and transmission as they do not support buffered serial communications.

[9] A MIDI Controller is a hardware or software device that generates and transmits MIDI data to MIDI-enabled devices, typically to trigger sounds and control parameters of an electronic music performance.

1.3.2.2 Arduino Development Board

Arduino[46] is an open-source hardware and software company that designs and manufactures single-board microcontrollers which are very popular with hobbyists.

The Arduino board designs use a variety of microcontrollers and are equipped with sets of digital and analogue input/output (I/O) pins that may be interfaced to a variety of interchangeable add-on modules known as shields. The boards feature Universal Serial Bus (USB) interfaces, which are also used for loading programs from personal computers. The microcontrollers can be programmed using a high-level programming language which is easier to use for novices than C. The Arduino project provides an integrated development environment (IDE) which is easy for novices to use.

The Arduino has also been a popular choice for building MIDI controllers, particularly gesture-oriented devices. Controllers such as a laser harp, keytar, glove interface, percussion devices, and step sequencer have been developed. MIDI processing on the Arduino itself is quite easy. However, the Arduino's lack of processing power does limit what can be achieved with it for more sophisticated devices,

1.3.2.3 Raspberry PI

The Raspberry PI is a low-cost credit card-sized computer running Linux. and is suited to more complex tasks than other microcontrollers. It has a processor that is a 32-bit, 700MHz system on a chip which is built on the ARM architecture. It has a composite video out, an analogue audio out, an external USB port, Ethernet port, HDMI connector, and Secure Digital (SD) card slot. The PI has been used on several synthesiser projects, including a Raspberry Pi Mellotron Emulator [47], a Raspberry Pi Looper[48] and a Raspberry Pi FM Touch Synth [49].

Zynthian is an open platform for sound synthesis. Based on Raspberry Pi and Linux, its hardware specification is public, and software is open Source.

1.3.2.4 BeagleBone Black

The BeagleBone Black[50] is a low-cost credit-card-sized development platform running Linux, and includes an onboard micro HDMI port, 512MB of double data-rate dynamic random access memory (DDR3L DRAM), 4GB onboard flash memory, an AM3358 processor at 1GHz, and 2 Programmable Real-time Units (PRU). The latter are programmable micro controllers on the chip that are optimized for real time tasks that would normally require very frequent interrupts.

The BeagleBone Black is compatible with a number of I/O expansion cards (Capes), one of which is the Bela Cape[51], an audio expansion card that features stereo audio I/O, 8 channels of 16-bit analogue I/O, 16 digital I/O and on-board speaker amps.

Pyo[52] is a Python module written in C to help DSP script creation and contains classes for a wide variety of audio signal processing, including music synthesis. Pyo has been ported on the Bela platform to provide very low-latency real-time audio DSP on the BeagleBone Black.

1.3.3 Current opportunities for making synthesisers

The advent of low-cost programmable systems-on-chip incorporating microprocessors and mixed-signal arrays, and also systems-on-chip containing multiple ARM processors and large FPGAs, together with the free availability of sophisticated suites of software tools for them, makes it feasible to produce a design for a cheap modular synthesiser suitable for experimentation. The motivation for the work in this thesis arose from this.

1.4 Related Work

The literature for sound synthesis and music technology was sparse for a long period. Although there were many commercial innovations, the technical academic research was minimal in the 1960s and 1970s. Hal Chamberlin's book [22], "Musical Applications of Microprocessors", is a very early reference on 80's analogue signal processing, covering music synthesis principles, sound modification methods, and direct computer synthesis methods. Other important texts and papers related to the area of computer music appeared since then, but it was really from the late 1990s that the volume of published work significantly increased. The increased interest in digital audio effects and synthesis led this, partly motivated by the improved hardware and software capabilities of devices. Some analogue signal processing papers also appeared but in a much smaller quantity.

Beat Frei [12] presents advanced techniques for digital sound generation in electronic musical instruments which focuses on popular building blocks. He also covers filters for sound synthesis including practical examples of oversampling, amplitude compression, and efficient parameter update schemes. Pete Symons [13] describes techniques for arbitrary waveform generation based upon direct digital synthesis principles. The difficulty involved in band-limited waveform generation in order to avoid aliasing effects, motivates one to seek alternative waveform-generation methods.

Douglas Self [23] discusses the design of high-quality circuitry for preamplifiers, mixing consoles, and other audio signal-processing devices with an emphasis on performance in the areas of noise, distortion, crosstalk etc. Stinchcombe [24] presents a very useful analysis of the classic Moog transistor ladder filter.

Nease *et al.* described a transistor ladder voltage-controlled filter implemented on a field programmable analogue array, however FPAAs are not available commercially.

Vesa Välimäki [27] gave an overview of virtual analogue synthesis in his keynote speech at DAFX-13. Ireland, 2013, which covered many of the important ideas in the field.

Lane, *et al,* [29], present DSP algorithms that duplicate the characteristics of subtractive synthesis using IIR filters. Välimäki *et al.* [30] reviewed oscillator algorithms for digital subtractive synthesis, which reproduce classic signal waveforms in the digital domain without incurring audible aliasing. Kleimola *et al.* [31] investigated polynomial and geometric phase shapers which produce classic and novel oscillator effect algorithms. The wave-forms are generally discontinuous, leading to aliasing artefacts. A polynomial bandlimited step function is used to reduce aliasing. Välimäki *et al.* [32] introduce new methods of generating digital versions of classical analogue waveforms with reduced aliasing. They also propose modifications to the digital nonlinear model of the Moog ladder filter. The proposed methods allow the synthesis of retro sounds with a modern computer.

Lazzarini et al. [33] propose new nonlinear distortion synthesis algorithms for classic analogue-waveform generation, which allow for tonal control without the need for separate filtering. Ochiai *et al.* [34] discuss CORDIC methods on FPGAs, covering Voltage-Controlled Oscillators (VCO) and Voltage-Controlled Amplifiers (VCA) but not Voltage-Controlled Filters (VCF).

Nolan Lem [53] describes a Custom Analogue Modular Synthesiser comprising a keyboard controller, voltage controlled oscillator (VCO), low frequency oscillator (LFO), attack-decay-sustain-release (ADSR), voltage controlled filter (VCF) and voltage controlled amplifier (VCA).

Briggs & Veilleux [28] reported on an FPGA digital music synthesiser which is close to one of the two approached explored in this paper. Their digital waveform generation is however, not band-limited.

1.5 Chapter Summary

A brief statement of the work involved was given, followed by a statement of the project's goals. A statement of what makes up a synthesiser was then given followed by the principles of subtractive and additive synthesis. A review of the present state of the art was then followed by a discussion on related work.

2. Programmable System-on-Chip Approach

PSoC (programmable system-on-chip) is a family of microcontroller integrated circuits by Cypress Semiconductor [54]. These chips include a CPU core and mixed-signal arrays of configurable integrated analogue and digital peripherals.

PSoC 5LP is a programmable embedded system-on-chip, integrating configurable analogue and digital peripherals, memory, and a microcontroller on a single chip. The PSoC 5LP includes:

- 32-bit Arm Cortex-M3 core plus DMA controller at up to 80 MHz
- Programmable digital and analogue peripherals
- Routing of any analogue or digital peripheral function to any pin
- A single PSoC device can integrate as many as 100 digital and analogue peripheral functions.

Cypress [55] provides an inexpensive PSoC5 LP development PCB. The original intention was to implement most of the functions of a modular synthesiser on the PSoC5's internal peripherals. 12 bits are required to encode an audio signal to an acceptable signal to noise ratio (SNR) of 74 dB, and 11 bits to encode the range of MIDI frequencies. The PSoC's on-board DAC is limited to 8 bits, despite early indications to the contrary, and the internal analogue peripherals have a dynamic range and linearity commensurate with this. Using the internal analogue peripherals, therefore, was not possible, so a set of external analogue functional blocks was developed.

The output waveforms were produced by subtractive synthesis, the source waveforms being produced by a Numerically Controlled Oscillator (NCO) driving values from a table of differentials of the desired waveform through an analogue integrator. This method is particularly suited to producing band-limited versions of the classic analogue synthesiser waveforms.

The rest of this chapter is organised as follows:

- A comment on subtractive synthesis is followed by a discussion on First Order Holds and a description of the PSoC5 LP.
- A high-level description of the functional blocks is followed by a detailed description of the blocks' implementation. An outline of the control firmware and PC software is given.
- Finally, a summary of the results is presented.

2.1 Theoretical Background

2.1.1 Subtractive Synthesis

In subtractive synthesis, partials of a waveform which is rich in harmonics are attenuated by a filter in order to alter the timbre of the sound. This technique is used in early analogue synthesisers, in which the harmonics of simple waveforms such as sawtooth, pulse or square waves are attenuated by a voltage-controlled resonant low-pass filter.

We use a Programmable System-on-chip device (Cypress PSoC5 LP) to parse MIDI commands and implement a Numerically-Controlled Oscillator (NCO) which drives the waveform generator, as well as outputting control voltages for the external Analogue Blocks which implement the rest of the analogue synthesiser[56].

2.1.2 First-order Hold

Given a number of sampled data points which represent the values of a function for a limited number of values of the independent variable, we are required to estimate the value of that function for an intermediate value of the independent variable. The simplest interpolation method is to locate the nearest data value and assign that value. The mechanism for implementing this is called a *Zero-Order Hold*. The next simplest method is to use the straight line between the two known points as the interpolant. The mechanism for implementing this is called a *First-Order Hold*.

A First-Order Hold (FOH) is used to generate waveforms from tables of the differential of the waveform to be generated, (differential wave tables) for two reasons:

- the approach results in small wave tables; it's easy to interpolate between the values in a table
- The FOH is particularly suited to the generation of the classic analogue synthesis waveforms, as the differential of these waveforms is, in general, piecewise constant.

The zero-order hold[57] (ZOH) is a mathematical model of the practical signal reconstruction done by a conventional digital-to-analogue converter (DAC). That is, it describes the effect of converting a discrete-time signal to a continuous-time signal by holding each sample value for one sample interval, as in Figure 3.

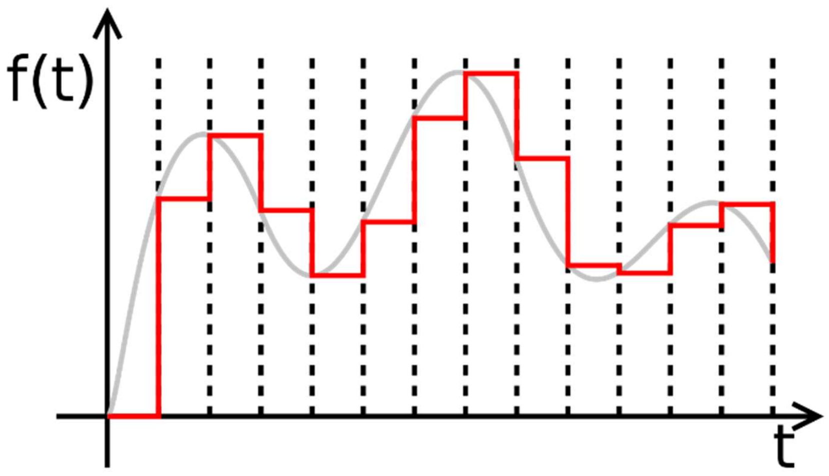


Figure 3 ZOH - Piecewise-constant signal

A better model is provided by linear interpolation, which is particularly suited to reproducing some of the classic analogue synthesis waveforms, since in general, these waveforms are piecewise linear. A First-Order Hold [58] (FOH) is a mathematical model of the practical reconstruction of sampled signals that could be carried out by a conventional digital-to-analogue converter (DAC) and an analogue integrator. For FOH, the signal is reconstructed as a piecewise linear approximation to the original signal that was sampled, as in Figure 4.

The output of the PSoC NCO, at a rate of 32 times the frequency of the note being played, drives piecewise-constant segments of a waveform which is the differential of the waveform we want to reproduce, into a linear integrator, which then outputs the desired waveform.

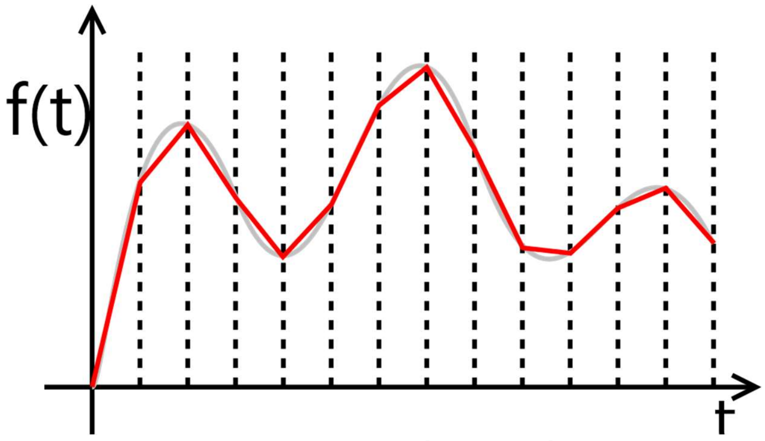


Figure 4 FOH - Piecewise linear signal

We can thus reproduce an arbitrary waveform, and in particular can reproduce the classic analogue synthesis waveforms (Rectangle, Square, Saw, Triangle, Sin, etc.). Note that although the reproduced waveform contains overtones[10], this does not lead directly to aliasing problems; this is not a waveform which is sampled at a fixed rate, but at a variable sample rate which is 32 times the fundamental frequency of the note being played [59]. Thus, it will sound better than a critically-sampled system that is much more vulnerable to aliasing. Figure 5, shows how a quasi-rectangular wave is constructed from its differential waveform.

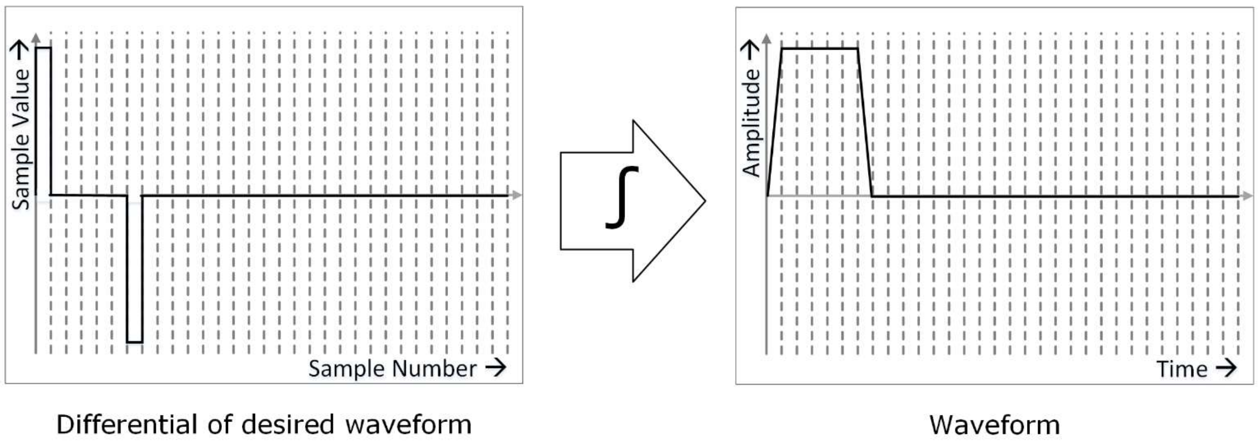


Figure 5 Waveform Construction

2.2 PSoC 5LP (Programmable System on a Chip)

The PSOC 5LP is a is a programmable embedded system-on-chip, integrating configurable analogue and digital peripherals, memory, and a microcontroller on a single chip. The architecture for this chip is shown in Figure 6 below:

[10] Any frequency greater than the fundamental frequency of a sound.

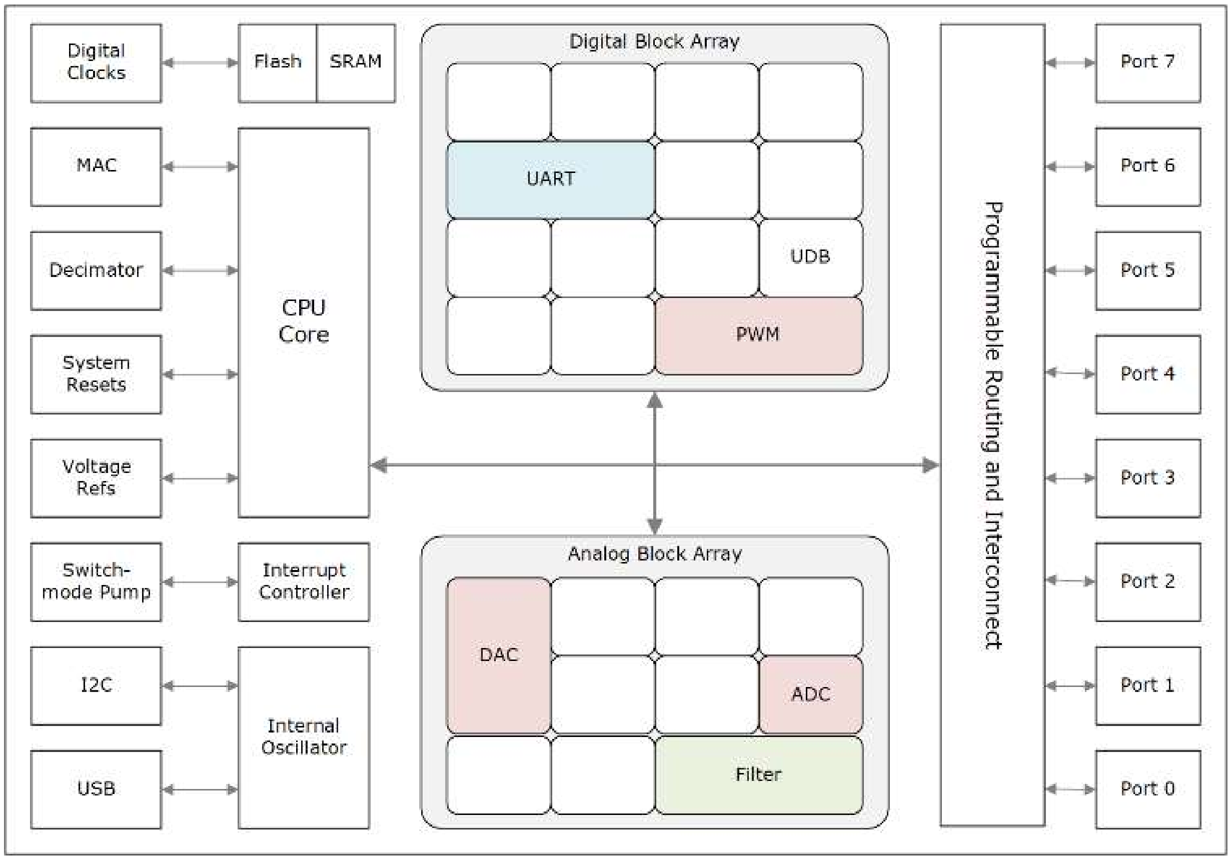


Figure 6 PSoC 5 Architecture

It features:

- 32-bit Arm Cortex-M3 CPU, 32 interrupt inputs
- 24-channel direct memory access (DMA) controller with data transfer between both peripherals and memory
- 24-bit fixed-point digital filter processor (DFB)
- 20+ Universal Building Blocks and Precise Analogue Peripherals
- Programmable Op Amps, 12-bit Successive Approximation Register (SAR) Analogue to Digital Converter (ADC) and 8-bit Digital to Analogue (DAC)

The PSoC platform provides analogue and digital blocks which are configured either by using pre-built library functions or by creating them in Verilog.

2.2.1 IDE and tool-chain

Cypress provides PSoC Creator, a Windows-based Integrated Design Environment (IDE), which facilitates concurrent hardware and firmware design of SoC 5LP based systems. Using PSoC Creator, one can select and place components, write C and/or Assembly source, and debug and program the project/part. Cypress also provides a cheap prototyping kit with an onboard programmer and debugger, which allows one to test the project in a hardware environment while viewing and debugging device activity in a software environment. For each Cypress-provided component that is inserted into the design, driver code is included. The toolchain works well, and the learning curve is not too steep.

Firmware

The firmware is organised as a preamble which configures the hardware, followed by an infinite loop which makes blocking calls to the universal asynchronous receiver/transmitter (UART) and parses the resultant MIDI commands to control the hardware.

Two timer interrupts are provided, one at 400 Hz to run an ADSR state machine, and another at 100 Hz to control vibrato[11]. DMA transfers samples of the piecewise constant waveforms to the output signal DAC.

2.3 Hardware Organisation

The hardware was organised into functional blocks which are plugged into a solderless plug-in breadboard which supplies power and permanent interconnect. Jumper cables are then used for signal routing in the same manner as patch cords on an analogue synth.

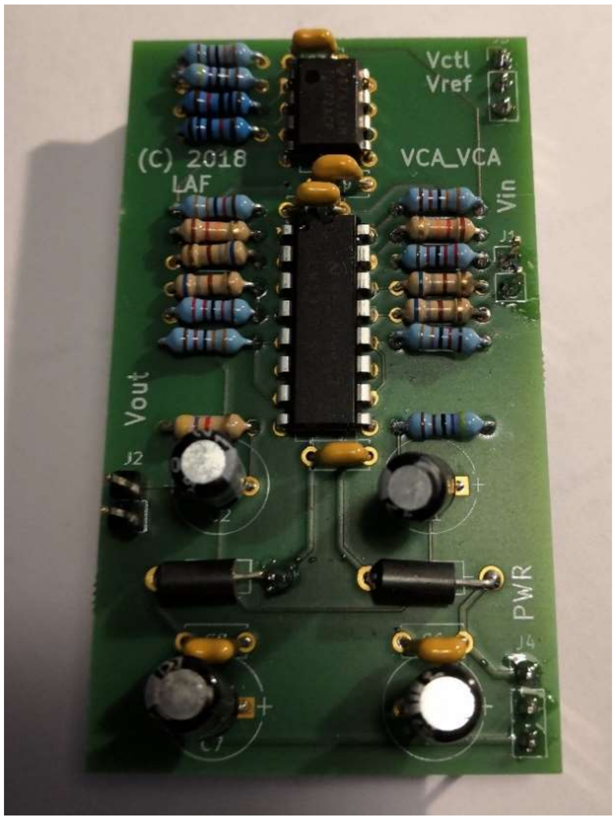


Figure 7 Typical Hardware Module

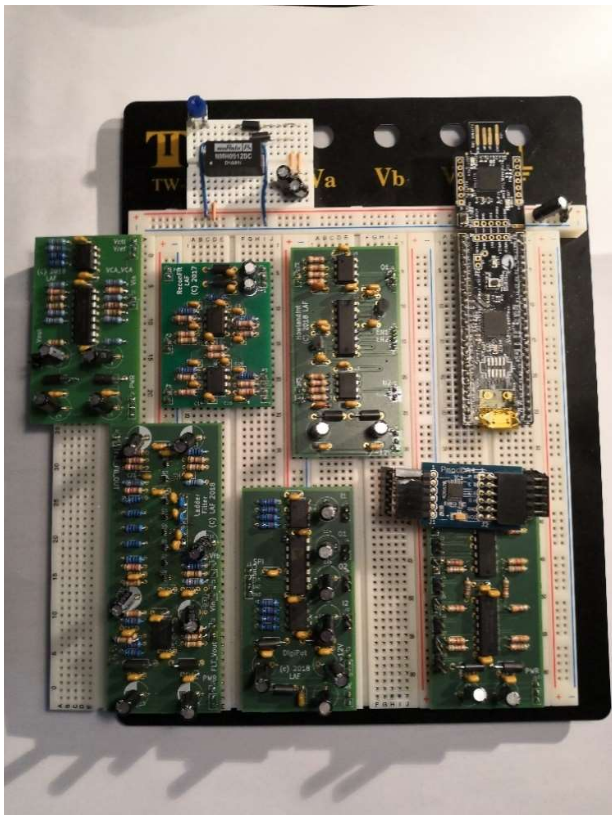


Figure 8 Hardware System

[11] A modulation effect that changes the frequency, and hence the pitch of a waveform in a periodic manner

Figure 7 shows a functional block, mechanised as a hardware module. These are plugged into a solderless breadboard as in Figure 8, which carries power lines and control signals. Data signals are routed using patch cords, making what is in effect, a modular synthesiser.

2.3.1 Functional Blocks

The PSoC hardware functions are divided into a digital section, depicted in Figure 9, and an analogue section, depicted in Figure 10. The digital section is embedded in the processor functional blocks and the analogue section is off-board. The outputs from the digital section are labelled on the right-hand side of Figure 7 and the corresponding inputs to the analogue board are given on the left-hand side of Figure 8. The additional Serial Peripheral Interface (SPI) is from the processor.

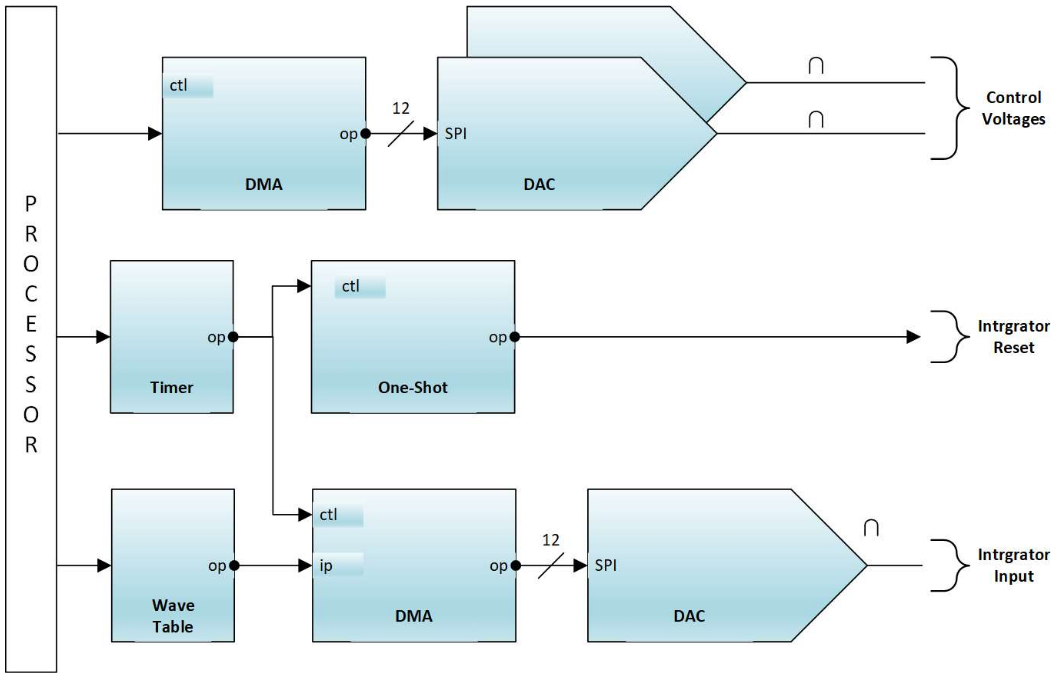


Figure 9 PSoC FBD – Digital

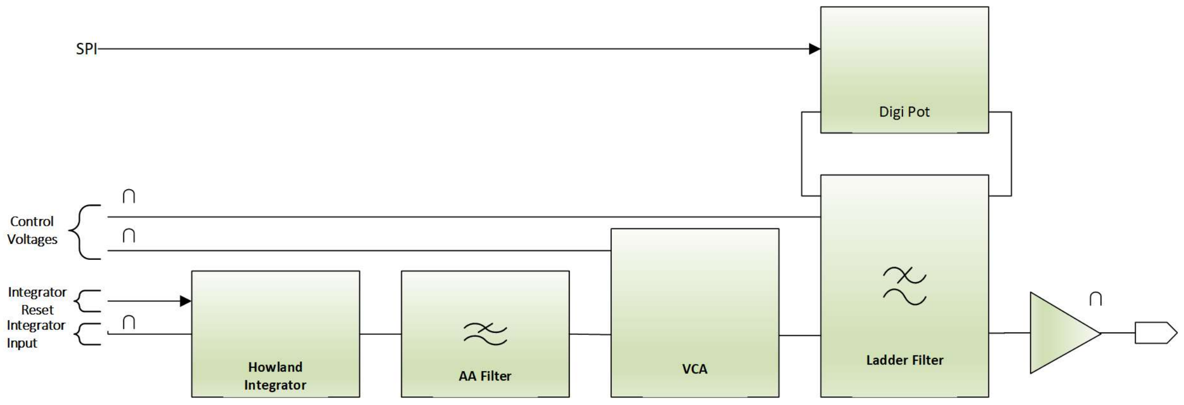


Figure 10 PSoC FBD - Analogue

The following subsections provide the details of the functional blocks that are defined in the list below:

• Numerically-Controlled oscillator (NCO)	Determines the frequency of the waveform to be produced.
• Control DACs	Voltage control for the analogue functions
• Howland Integrator	Used in conjunction with the differential wave form tables to define the timbre of the note being played.
• Anti-aliasing filter	Used for the benefit of any downstream data acquisition systems or digital amplifiers which sample the output of this synthesiser
• Voltage-Controlled Amplifier (VCA)	Controlled dynamically in order to determine the envelope of the note being played.
• Ladder Filter	Classic 24 dB/Octave peaking LPF, used for dynamic modification of the Timbre of the note being played.
• Digital Potentiometer	Controls the feedback factor for the Ladder Filter.

In Section 4.4 the details surrounding the implementation of these blocks are given. The PSOC-based functional block diagram is given in Figure 94 in Appendix A.

2.3.1.1 NCO

The Numerically Controlled Oscillator (NCO) is based on a 24-bit PSoC timer-counter clocked at 24 MHz which inputs a 20-bit *period* parameter corresponding to the note being played. A block diagram is given in Figure 11. The NCO determines the rate at which data are read from the differential wave table (labelled as LUT in the figure), and hence the frequency of the note being played. The *period* parameter can be modulated in real-time in order to produce vibrato and glissando[12] effects.

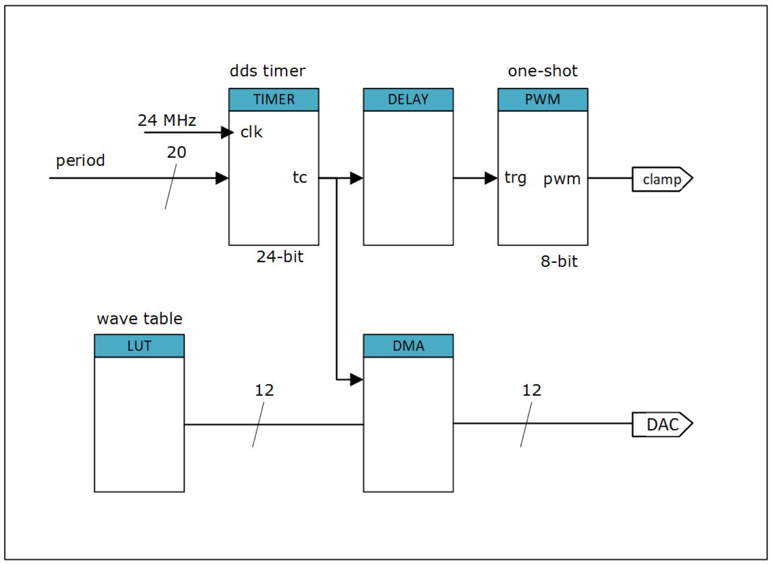


Figure 11 NCO

[12] A smooth glide from one pitch to another.

The terminal count[13] (*tc*) of the DDS timer triggers a DMA cycle which transfers a differential wave table value to an off-board DAC for presentation to the analogue integrator. In the top half of Figure 11, it shows how a delayed version of the *tc* triggers a one-shot which is connected to the *clamp* input of the off-board analogue integrator, so that the latter clamps the output waveform to zero volts at the waveform's zero-crossing, in order to avoid drift. A more detailed view is presented in Figure 95. Refer particularly to the block labelled `ddsTimer`.

2.3.1.2 Control DACs

Control voltages are generated by an octal DAC (*AD5628-1*) mounted on a Digilent PmodDA4 [60] and buffered and scaled on the DacBuf module shown in Figure 12. A full circuit diagram is provided in Figure 97.

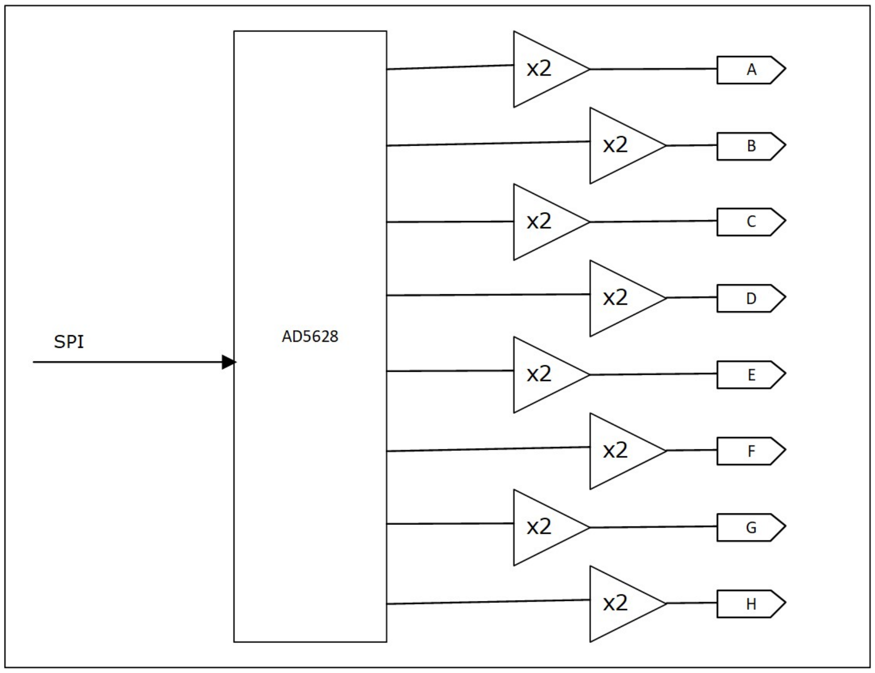


Figure 12 DACs, Buffers

2.3.1.3 Howland Integrator[14]

The piece-wise constant differential waveform signal from the DAC is fed to a Howland Integrator [25], in order to produce a piece-wise linear signal, as described above in Section 2.1.2. Figure 13, gives an outline circuit diagram of the Howland system developed. A full circuit diagram with component values

[13] An output from a digital timer that indicates the end of the time period being timed.

[14] The Howland Current Pump uses the positive and negative inputs of an operational amplifier to make a high-impedance current source using a conventional operational amplifier. One of the obscure applications for the Howland Current pump is the "Howland Integrator", formed by using a capacitor as the load. The integrator is reset with a single FET or switch to ground.

is provided in Figure 98, Appendix B. The *clamp* output from the NCO block is connected to the *clamp* input of the integrator.

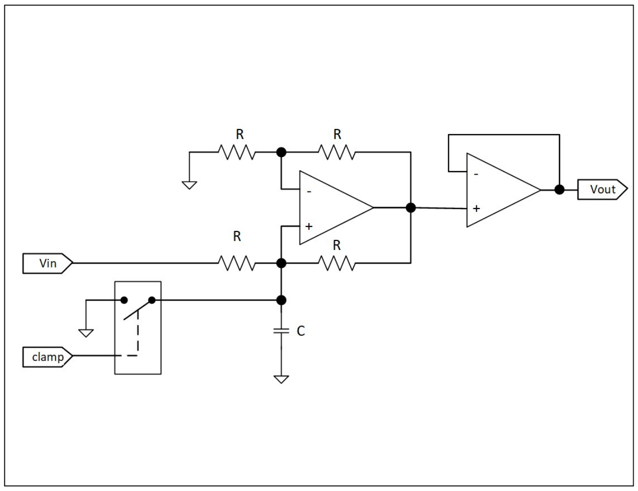


Figure 13 Howland Integrator

2.3.1.4 Anti-aliasing filter

A conventional 4-pole anti-aliasing filter is placed at the output of the Howland integrator. As mentioned above in Section 2.3.1, this is for the benefit of any downstream sampled-data systems[15] connected to the synthesiser output. The circuit diagram for this well-known filter is given in Figure 14. A full circuit diagram with component values is provided in Figure 105, Appendix B.

[15] Typically, desktop amplifiers used with PCs and low-end solid-state guitar amplifiers operate in Class D. these sound horrible when driven by a signal with overtones. Hence the need to band-limit the signal.

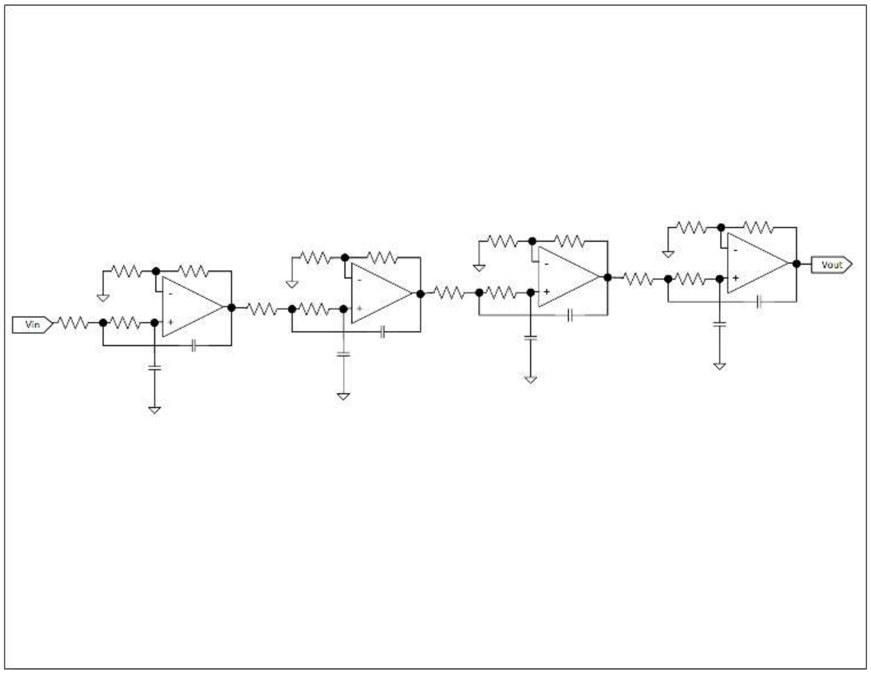


Figure 14 - Anti-aliasing filter

2.3.1.5 VCA

The VCA uses a Texas Instruments current-controlled Operational Transconductance Amplifier (OTA) as a VCA. Because the off-isolation[16] isn't all that it should be, there's a small amount of audio feed-through when the `Note` is supposed to be `off`. To suppress this the simple expedient of using two VCAs in series was applied[17]. These are shown in Figure 15. A full circuit diagram with component values is provided in Figure 99, Appendix B.

[16] in analogue switching, OFF Isolation is a measurement of OFF-state switch impedance. Here, the term is used as a measure of the OTA's ability to block an AC signal in its OFF-state.

[17] This is a little unconventional, but since the LM13700 contains two OTAs, and having one VCA per hardware module, nothing is lost.

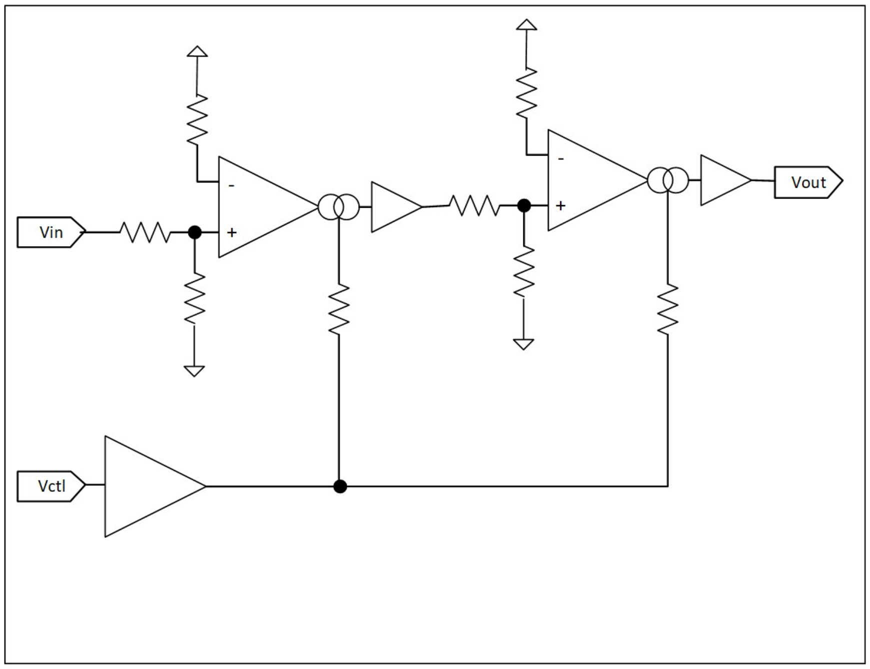


Figure 15 Voltage-Controlled Amplifier

2.3.1.6 Ladder Filter

The ladder filter [24] is a fourth-order resonant low-pass filter using a ladder circuit topology with embedded nonlinear elements. These nonlinearities produce a distortion that adds a specific and musically-pleasing "warmth" to the output sound. This is a conventional design derived from the Moog patented design using matched transistor pairs as the active element. Figure 16, shows the diagram with the audio and cut-off control inputs listed. The two transistors on the bottom act as an exponential converter used to convert from a MIDI-aligned linear control voltage, to an exponentially-derived control current, which matches the human ear's pitch-frequency curve. The four poles of the filter are implemented using the NPN transistor-pairs whose emitters are connected with a capacitor.

A full circuit diagram with component values is provided in Figure 100, Appendix B.

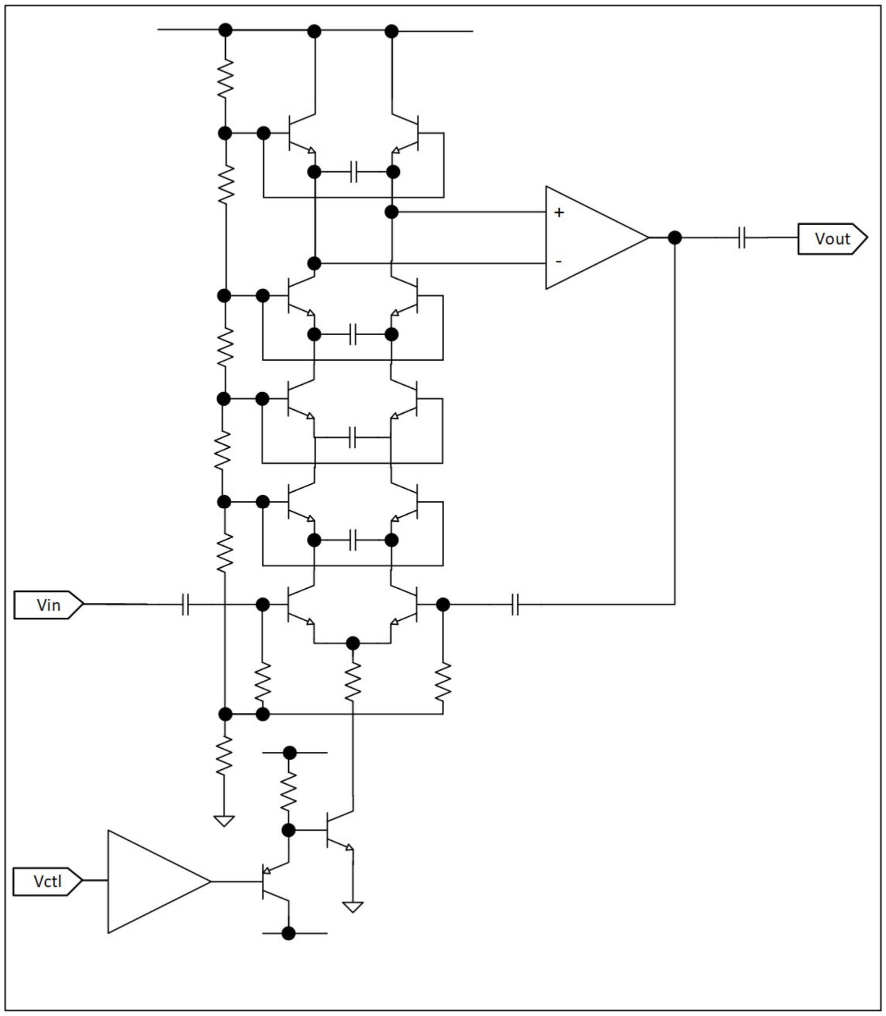


Figure 16 Ladder Filter

2.3.1.7 Digital Potentiometer

This uses an Analogue Devices' AD8402 dual Digital Potentiometer device for ladder filter feedback control (Resonance) and for output level control. Since each of these is adjusted before playing a note, the problem of zipper noise [37] was ignored. Figure 17 shows the potentiometer circuit for one channel only. The SPI controls the resistance value. Because of the limited voltage range of the digital potentiometer (0-5V), the op-amps maintain the DC level at both ends of the potentiometer at half the power-supply voltage.

A full circuit diagram with component values is provided in Figure 101, Appendix B.

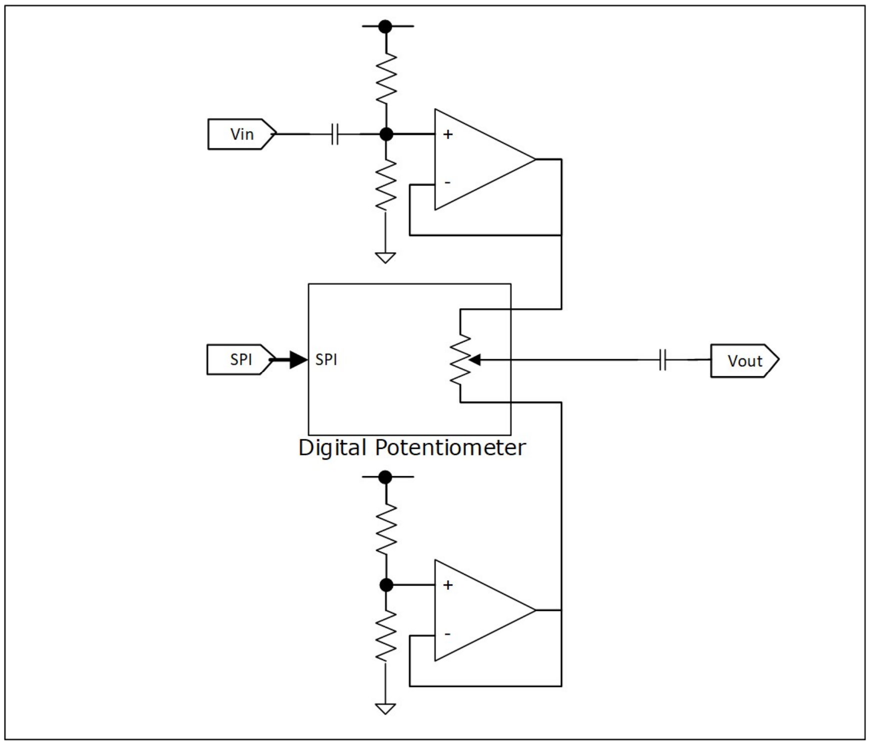


Figure 17 Digital Potentiometer

2.4 IMPLEMENTATION

The section provides the actual implementation details of the blocks explained in the previous section. These would be useful for anyone wishing to reproduce the work.

2.4.1.1 NCO

The NCO function was implemented on the PSoC 5LP's FPGA using Cypress library components. These are detailed in in Figure 18. Additionally, some block descriptions are given below to elaborate further.

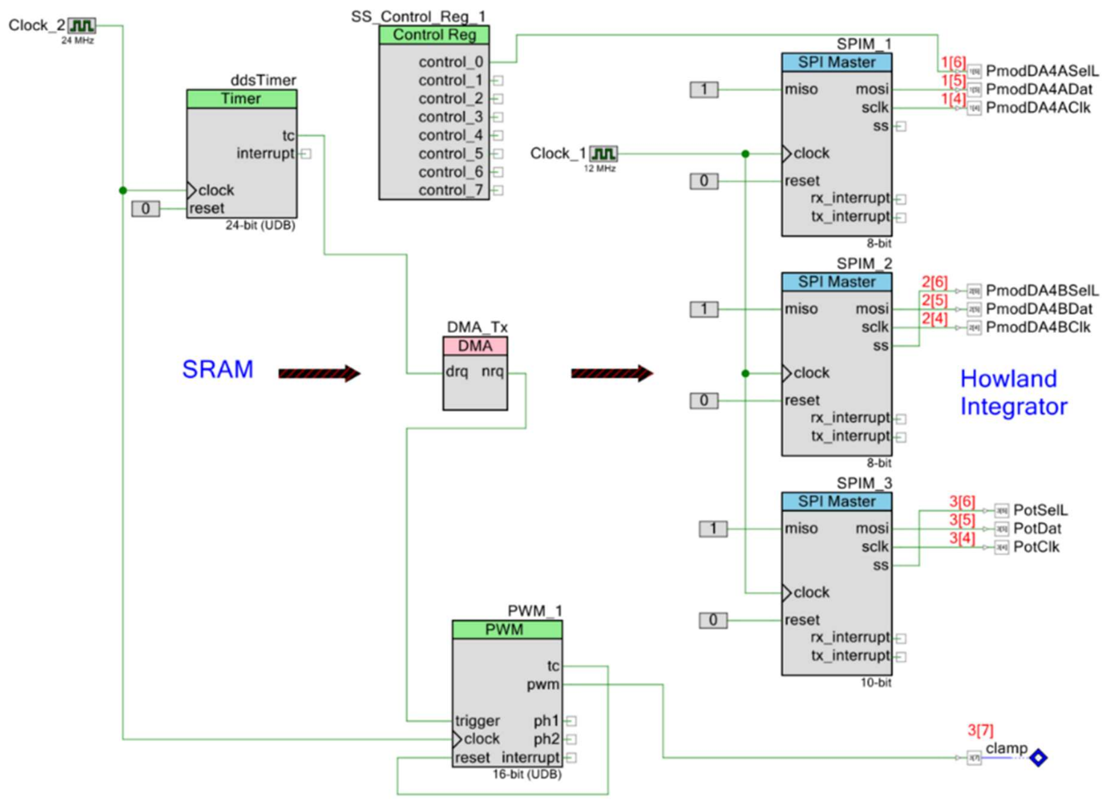


Figure 18 PSoC NCO Function

ddsTimer

The 24-bit PSoC timer-counter clocked at 24 MHz inputs a 20-bit period parameter corresponding to the frequency of the note being played. This parameter can be varied on-the-fly for glissando and vibrato effects.

DMA

The terminal count (tc) of the DDS timer triggers a DMA cycle which transfers a wave table value from static random access memory (SRAM) to an off-board DAC via the SPIM_2 SPI Master block for presentation to the analogue integrator.

DAC

The external 12-bit DAC is provided by a Digilent PmodDA4, which is separate from that used for general level-setting, for ease of design. This is buffered and scaled by one of the buffers in the DAC Buffers, shown in Figure 97.

Clamp

A pulse-width modulation (PWM) block triggered by the DMA outputs a clamp pulse to the Howland Integrator in order to prevent drift.

Level-Setting

Level setting is provided by a second external 12-bit DAC, provided by a Digilent PmodDA4, via SPIM_1 SPI Master block. The DAC outputs are buffered and scaled by the DAC Buffer. The SPIM_3 SPI Master block interfaces with external digital potentiometers.

Timers

Two Timer blocks, shown in Figure 96, provide timed interrupts for Interrupt Service Routines.

UART

A UART block is used for serial over USB communications with the PC.

2.4.2 WaveTables

Wave Tables, containing the differential of the waveform to be generated, were developed on an ad-hoc basis for sine, rectangular, square, triangle, sawtooth and impulse waveforms. The principle on which these work was described in Section 2.1.2 above. The following Figure 19 to Figure 24, show the contents of the wavetable for each waveform type. Each table has 32, 16-bit entries.

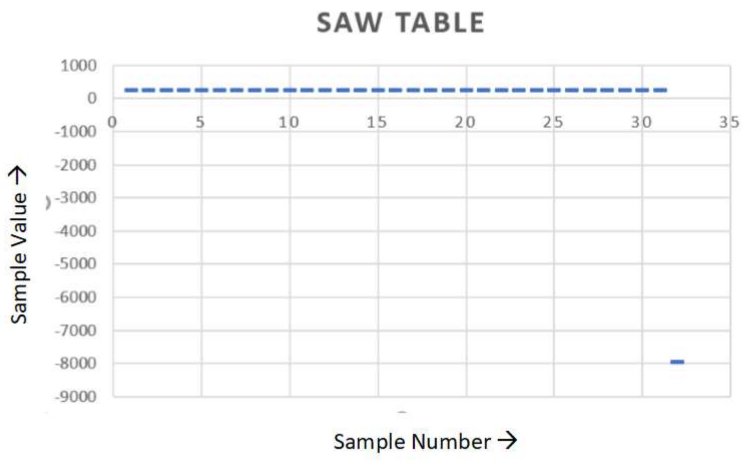


Figure 19 - SAW wave generation table

Inputting the values of the samples in Figure 19 to an integrator produced in a positive linear ramp for 31 sample times, followed by a large negative ramp for 1 sample time, resulting in a classic ramp waveform.

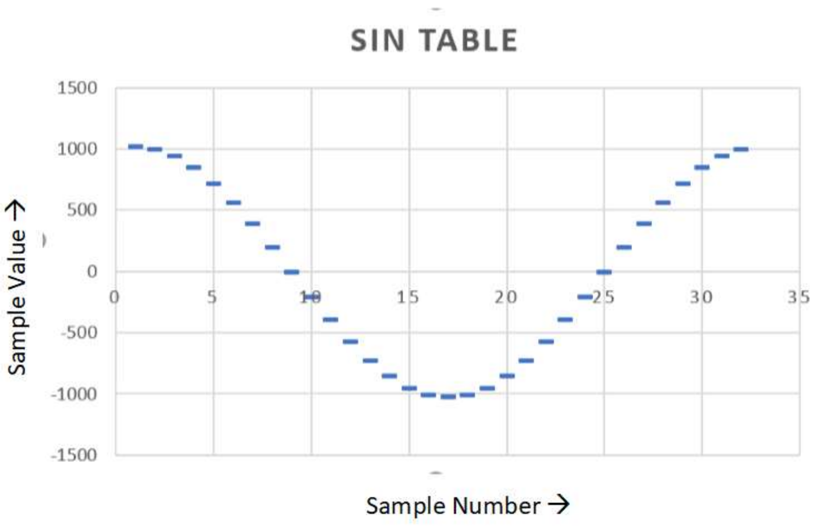


Figure 20- Sine wave generation table

Inputting the values of the samples in Figure 20 to an integrator will resulted in a sine wave for 32 sample times, given that integrating the tables piecewise constant cosine values, produced a piecewise linear approximation of a sine wave.

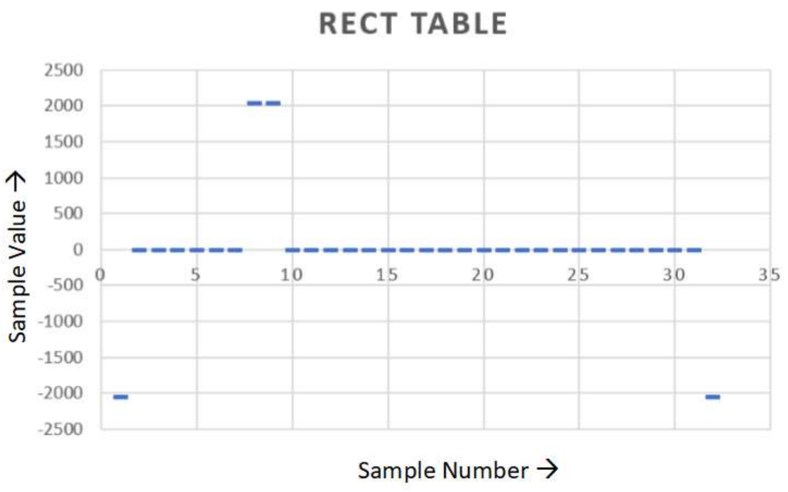


Figure 21 - Rectangular wave generation table

Inputting the values of the samples in Figure 21 to an integrator resulted in a negative ramp for one sample time, followed by a constant (negative) value for 6 sample times, a positive ramp for 2 sample times, followed by a constant (positive) value for 22 sample times, and finally a negative ramp for 1 sample time. This produced a trapezoid approximation of a rectangular wave.

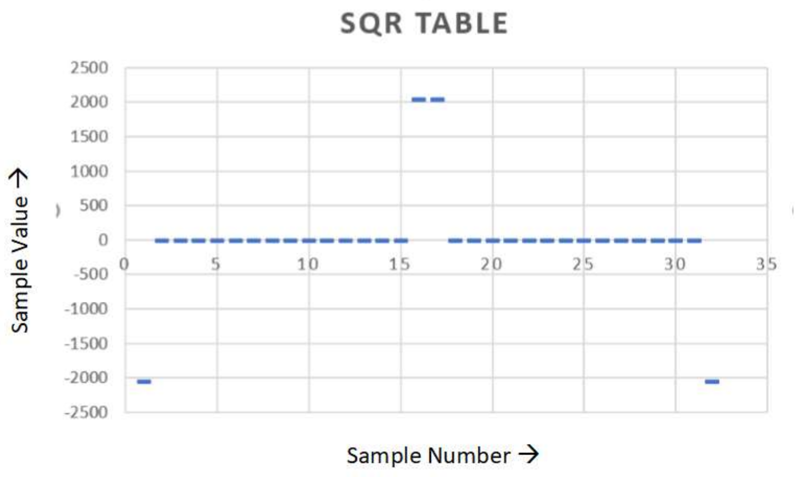


Figure 22 - Square wave generation table

Inputting the values of the samples in Figure 22 to an integrator resulted in a negative ramp for one sample time, followed by a constant (negative) value for 14 sample times, a positive ramp for 2 sample times, followed by a constant (positive) value for 14 sample times, and finally a negative ramp for 1 sample time. This produced a trapezoid approximation of a square wave.

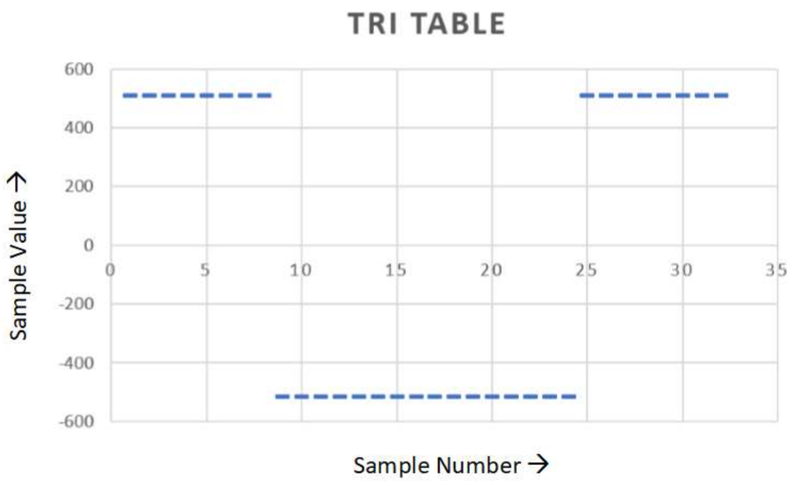


Figure 23 - Triangular wave generation table

Inputting the values of the samples in Figure 23 to an integrator resulted in a positive ramp for 8 sample times, followed by a negative ramp for 16 sample times, and then a positive ramp for 8 sample times. This produced a triangular wave.

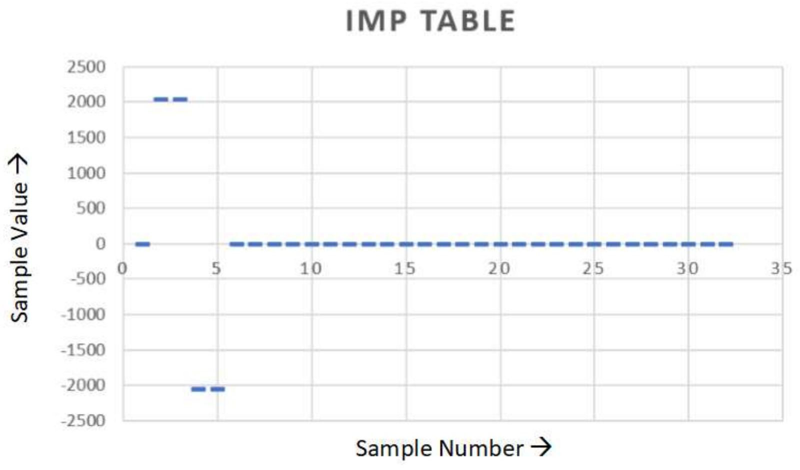


Figure 24 - Impulse wave generation table

In order to produce waveforms of constant amplitude over the range of notes produced by the MIDI keyboard[18], the amplitudes of the above tables needed to be scaled by a quantity which is inversely proportional to frequency.

2.4.3 Howland Integrator

The Howland integrator, as described in Section 2.3.1.3, and detailed in Figure 98 is one of the applications of a Howland current pump which uses a conventional operational amplifier to make a high-impedance current source. The integrator was periodically reset to ground using a FET Switch.

2.4.4 AAF

The Antialiasing Filter (AAF) as described in 2.3.1.4, Figure 105 is a conventional four-pole design.

2.4.5 VCA

The Voltage-Controlled Amplifier (VCA) as described in 2.3.1.5 and detailed in Figure 99, used a pair of LM13700 amplifiers in series for good off-isolation. The LM13700 comprises two current-controlled transconductance amplifiers. The control current for these was provided by a large resistance driven by an op-amp.

2.4.6 Ladder Filter

The ladder filter as described in 2.3.1.6 and detailed in Figure 100 was a conventional design, using PMP4201Y NPN/NPN matched double transistors as the active elements. The control current was provided by a BC847BPN NPN/PNP general-purpose complementary transistor pair with a closely matched current gain, which formed a temperature-compensated exponentially-controlled current source.

[18] a piano-style electronic musical keyboard, often with other buttons, wheels and sliders, used for sending MIDI signals or commands to MIDI-enabled devices.

2.4.7 DigiPot

The digital potentiometer as described in 2.3.1.7 and detailed in Figure 101 used an Analogue Devices' AD8402 digital potentiometer for ladder filter feedback control and for output level control.

2.5 SOFTWARE

The next sections describe the software written in C for controlling the hardware that resides on the microprocessor and software written in Python that runs on the PC.

2.5.1 Control Firmware

The control firmware, running on PSoC CPU, inputs MIDI messages[19] from the PC via a serial channel . It either updates tables in SRAM or else outputs control voltages.

The firmware comprises a foreground section, which initialises and updates elements of a set of Global Data Structures (GDS), and an Interrupt Service Routine, which updates the parameters of an ADSR state machine. The foreground section initialises the Cypress-generated Hardware Drivers and the GDS before entering a loop which:

- gets MIDI Messages from the UART and
- parses the MIDI Messages and updates the Global Data Structures

2.5.2 PC Software

The PC Software receives input MIDI messages from a MIDI Keyboard or a MIDI Controller (Figure 102), and sends them to the PSoC CPU via a Serial_Over_USB interface in order to control the Synth. The Control firmware on the PSoC CPU parses the MIDI messages and controls the hardware either by updating tables in SRAM which are read by the interrupt service routine or by directly outputting control voltages. In this way notes are played, and timbres can be modified.

2.6 RESULTS

The sections below give results obtained for the PSoC synthesis system. The first subsection is used to illustrate the correct working of the NCO only by demonstrating that it can produce sinewaves of the correct frequency. The next subsections will demonstrate the various waveshapes that can be generated.

2.6.1 NCO Operation

The system produced useful wave shapes over a wide range of frequencies as illustrated by the sinewaves given in Figure 25 - Figure 27. The sine waves were generated from the table in Figure 20. The frequency of the sinewaves in each of the figures was stable showing that the synthesiser stayed in tune. The fidelity of the sinewave output is examined in Figure 28 using a spectral plot for a sinewave of 220Hz. It can be seen that fundamental component at 220Hz was at least 40dB greater than any other harmonics produced as an artefact of the generation process.

[19] A MIDI message comprises one of 128 MIDI Commands, followed by 0, 1 or 2 8-bit parameters. It is part of the MIDI protocol which transfers musical information between keyboards and synthesizer.

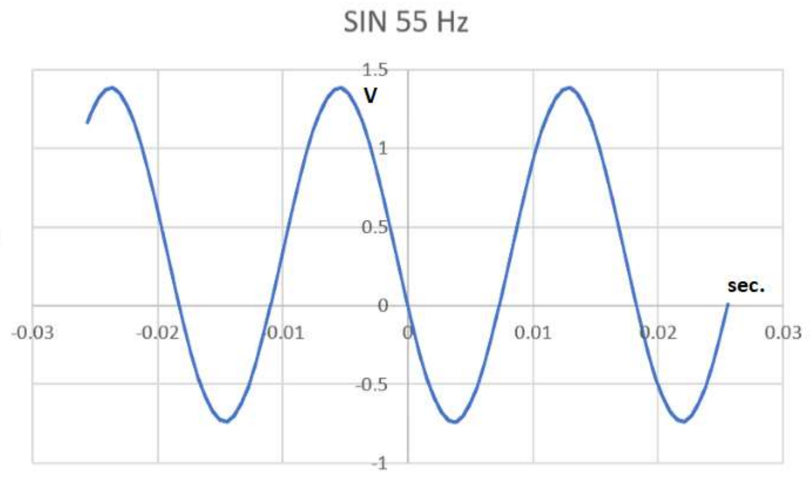


Figure 25 –Observed 55 Hz Sine wave

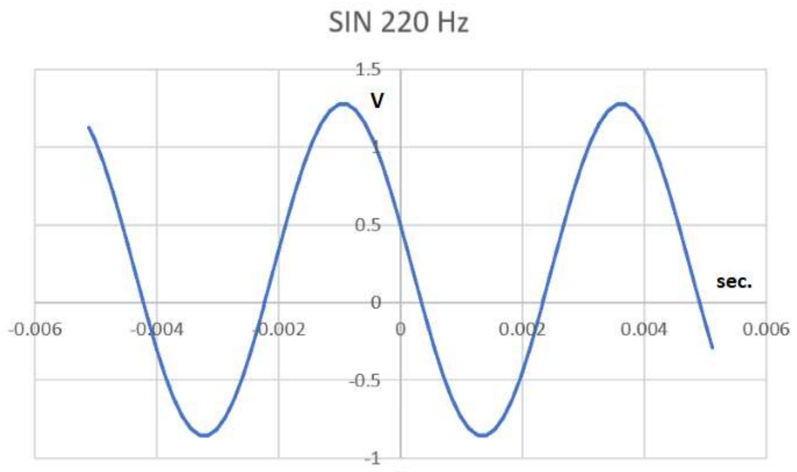


Figure 26 - Observed 220 Hz Sine wave

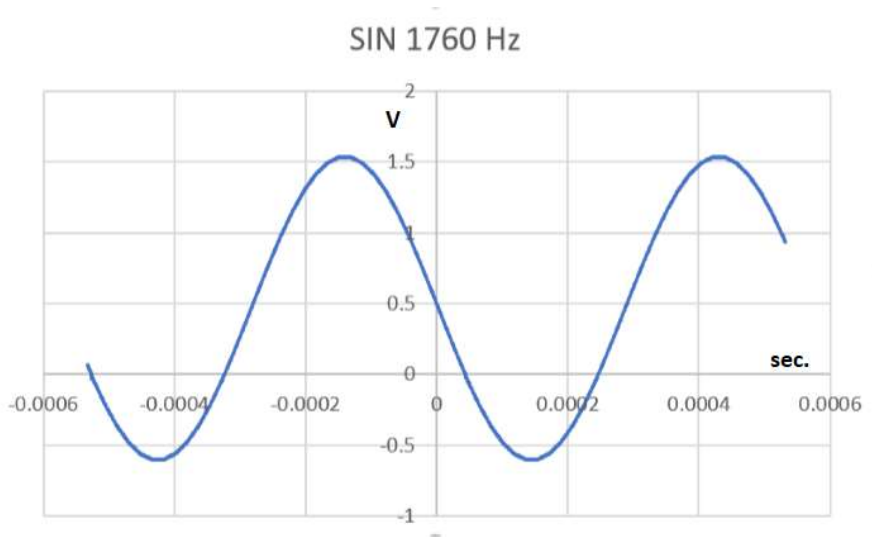


Figure 27 - Observed 1760 Hz Sine wave

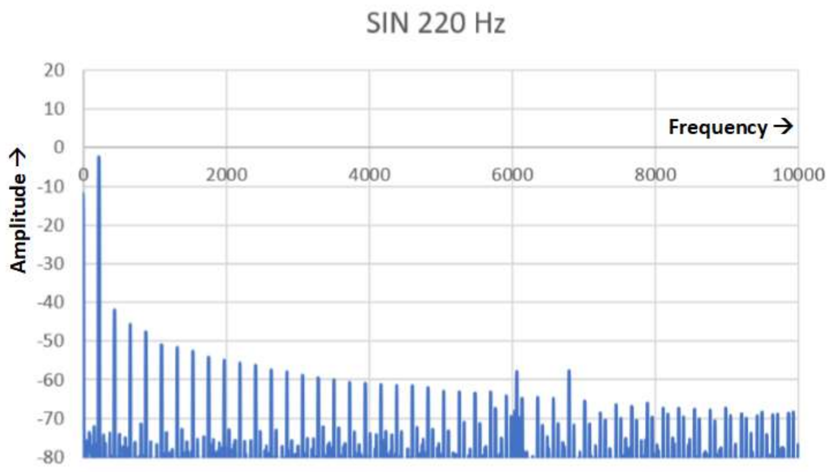


Figure 28 - Observed 220 Hz Sine Spectrum

2.6.2 Wave Generation

As well as generating sinewaves, the system generated the classic waves of subtractive synthesis and others: rectangular, square, triangle, sawtooth and impulse.

- The 220Hz rectangular waveform in Figure 29 was generated from the table in Figure 21.
- The square waveform in Figure 30 was generated from the table in Figure 22.
- The triangle waveform in Figure 31 was generated from the table in Figure 23
- The sawtooth waveform in Figure 32 was generated from the table in Figure 19.
- The impulse waveform in Figure 33 - Observed 220 Hz impulse wave was generated from the table in Figure 24.

In all cases it can be observed from the figures that that the classic analogue synth waveforms are reproduced faithfully.

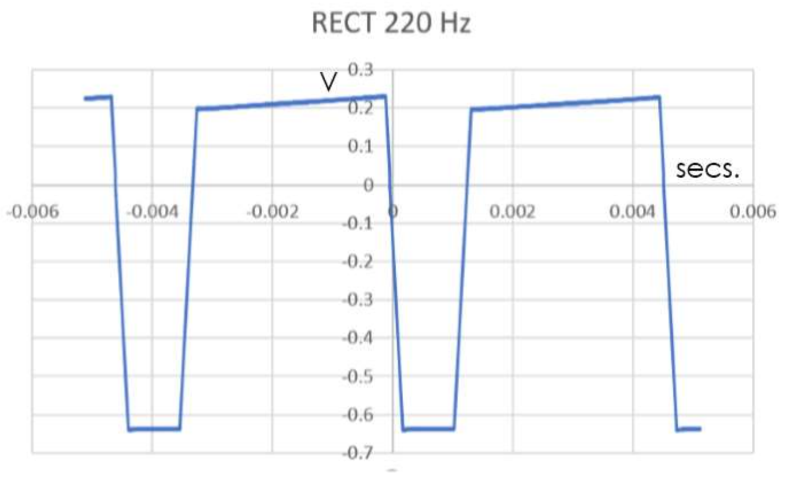


Figure 29 - Observed 220 Hz Rectangular wave

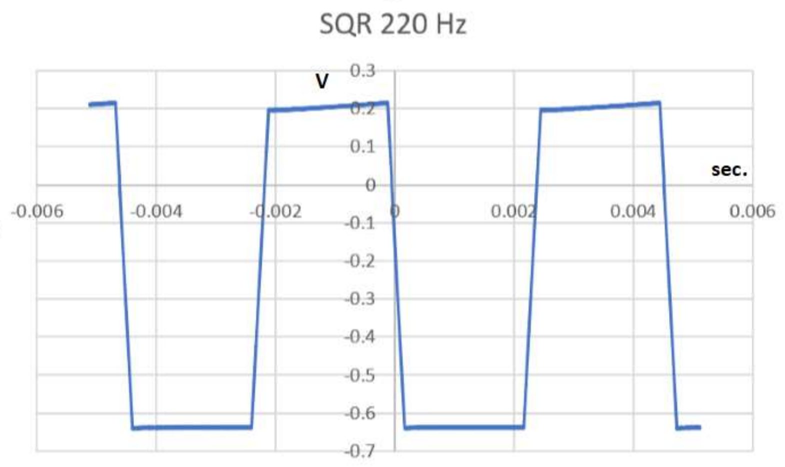


Figure 30 - Observed 220 Hz Square wave

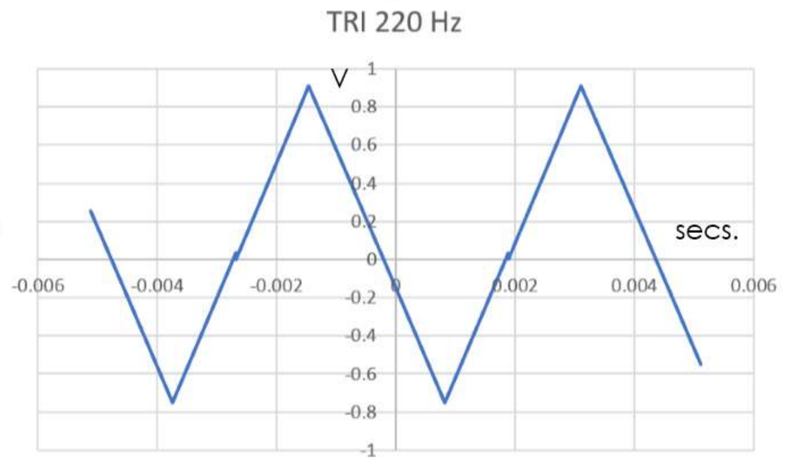


Figure 31 - Observed 220 Hz Triangular wave

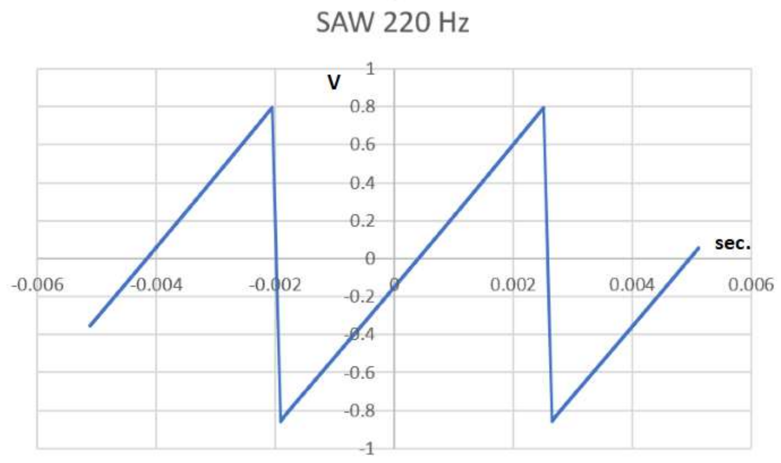


Figure 32 - Observed 220 Hz Sawtooth wave

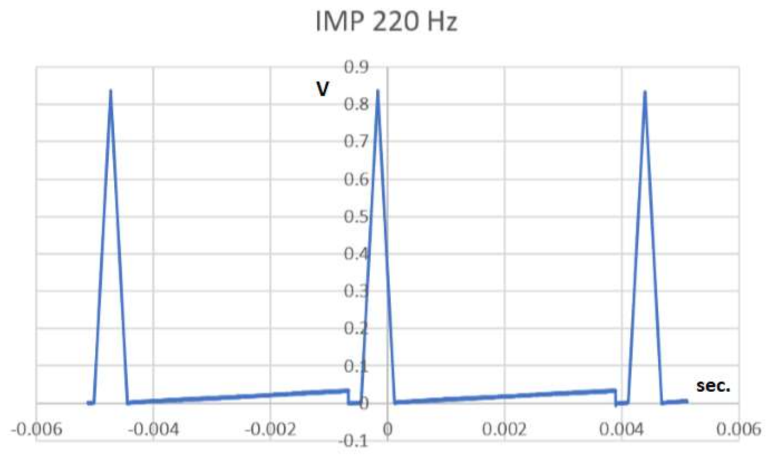


Figure 33 - Observed 220 Hz Impulse wave

2.6.3 Ladder Filter

The ladder filter produced a 4-pole low-pass voltage-controlled filter, with corner frequencies related to the exponential of the control voltage:

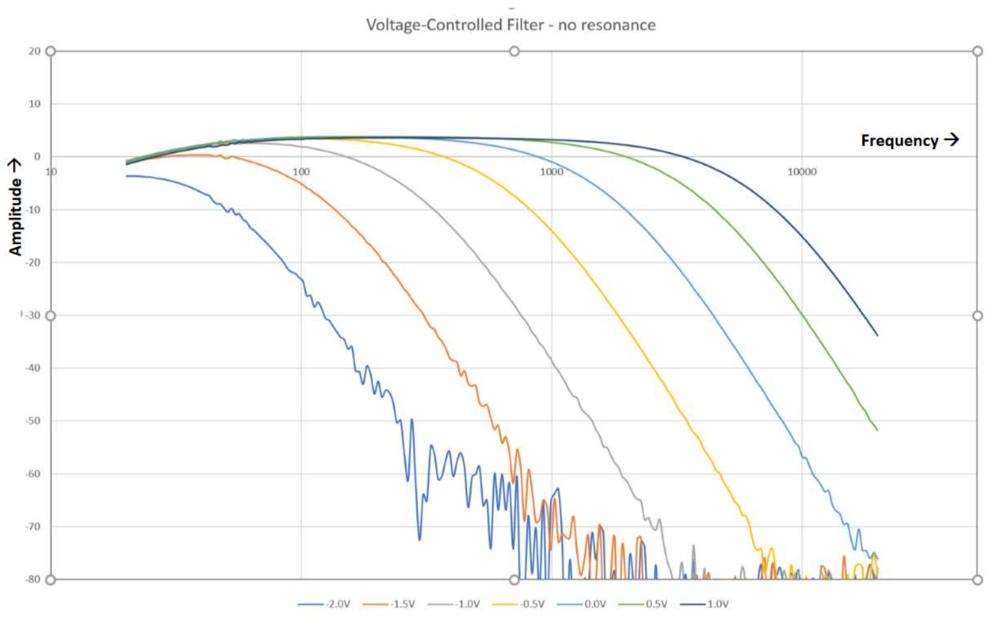


Figure 34 - Observed 4-pole low-pass VCF

Note that the horizontal axis in Figure 34 above is logarithmic: the plots correspond to control voltages starting at -2 V and stepped at intervals of 0.5 V. The corner frequencies tracked the exponential of the control voltage. Increasing the feedback gain resulted in a peaked frequency response shown in Figure 35 below. Again, the corner frequencies were related to the exponential of the control voltage. The ratio of peak response to low-pass response was consistent:

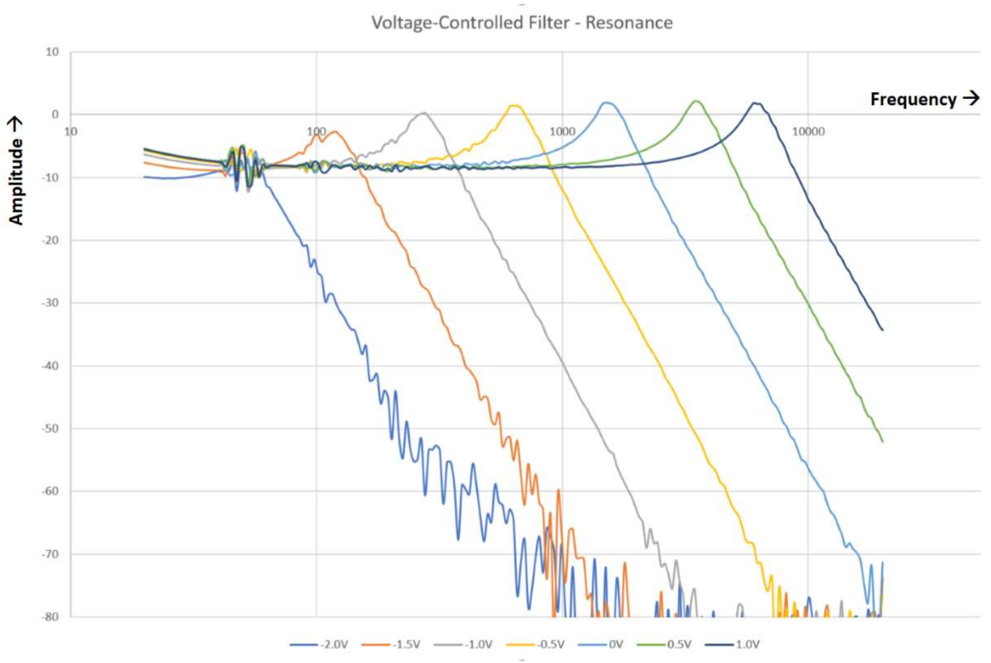


Figure 35 - Observed 4-pole low-pass peaked VCF

Each trace in the above plots was logged in a spectrum analyser, then exported to a .CSV file and finally the resultant ensemble imported to Excel for plotting.

2.7 PRACTICAL CONSIDERATIONS:

Hand-soldering surface-mount packages presents a significant challenge: the ladder filter module used SOT-363 – packaged transistor pairs. Figure 36 shows the completed synthesiser. Its modular character is shown by the various PCBs with their own functionality. The PSoC is evident in the top left-hand corner of the photo. The wiring interconnections were designed to be as uncluttered as possible. All of the power grid was hidden underneath the circuitry.

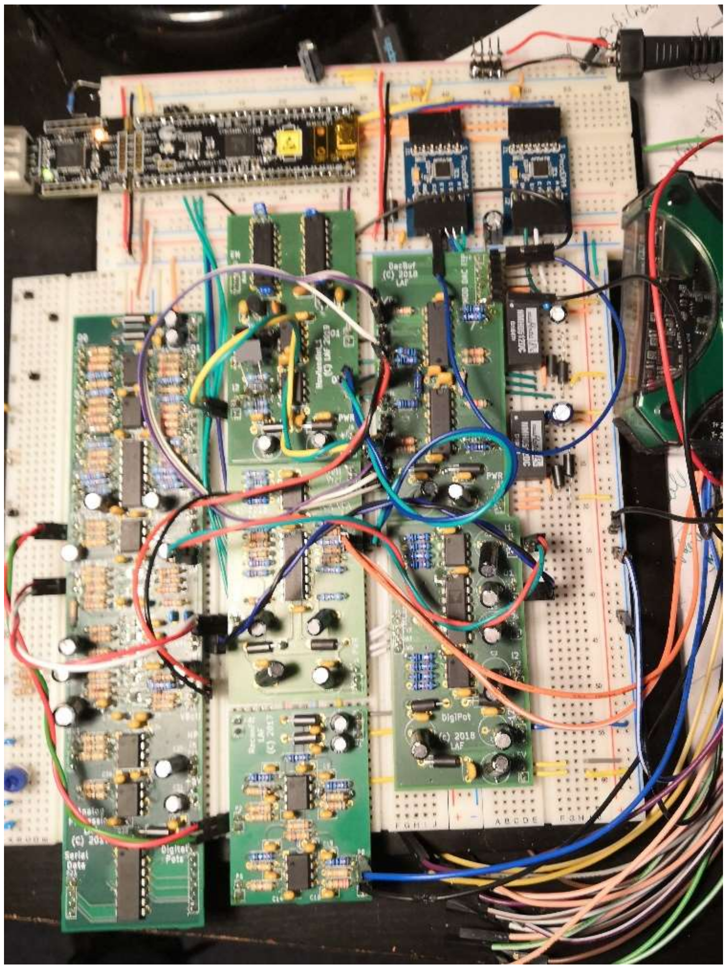


Figure 36 - Hardware System, post-debug

2.8 USER EVALUATION

A selection of waveforms and effects was recorded on Audacity [59] at a 48 KHz sampling rate and a 24-bit resolution from a USB ADC. These were played back over an analogue keyboard amplifier to a 5-person panel whose scores and comments are summarised below. The survey questionnaires are shown in Appendix – Survey Questionnaires. The overall sound quality was perceived to be good.

PSoC-Based Subtractive Synth	Avg. score	Comments
Sine	4	Buzz, slight harmonics
Square	4.6	
Rect	4.6	
Triangle	4	
Sawtooth	4.6	
Range	3.6	Bass not in tune[20], aliasing in high note
ADSR	4.2	Attack sounds 'awkward'
Wheel	5	
Filter	4.2	A little distortion in the filter. Self-oscillator distorted.

Table 1 User Evaluation

2.9 PSOC Approach Summary

The PSOC approach yielded a system that was, with the exception of the surface-mount packaging mentioned above, easy to build, given reasonable hardware and software development skills. The modular hardware used meant that, in effect, a cheap modular synthesiser, suitable for experimentation was produced

2.9.1 Achievements

A PSoC-based subtractive synthesiser using mixed-signal arrays to implement some functions and to control other off-board analogue functions was built. Apart from using a NCO as part of a note generator, the signal path was entirely analogue.

2.9.2 Obstacles

The bit resolution of the PSoC's DACs and the poor dynamic range of the on-board analogue components meant that it wasn't possible to use the PSoC's analogue resources for audio processing. Instead, off-board analogue blocks were controlled from external DACs.

Cypress provides considerable resources to help design Datapath functions. This seemed quite suitable for designing the NCO. However, it turned out that designing datapath functions was difficult. The realisation that one could repurpose a Timer-Counter from Cypress' Component Library as an NCO helped overcome this difficulty.

2.9.3 Meeting the objectives

The PSoC 5LP board used cost less than €20, putting it within the reach of the amateur enthusiast. The off-board analogue peripherals were designed to be highly modular, and with the exception of the transistors used in the resonant filter, through-hole components were used, which ensured that their construction was within the capabilities of the enthusiast.

[20] The 'Bass not in tune' comment relates to a 27.5 Hz sawtooth wave. When a number of wave shapes at 27.5 Hz, were played back using an online additive synthesis waveform generator[68] , the pitch this author perceived varied by up to a semitone as the wave's shape was changed. At 55 Hz, the perceived pitch was constant. Other listener's experiences may be different.

The sound was found to be high-quality; the comment from one listener regarding ‘distortion in the filter’ was especially gratifying.

3. FPGA-based Approach

The intent here is to implement a complete additive synthesiser on the device's Field Programmable Gate Array (FPGA) and to control it from the processor. Xilinx provides an expensive development FPGA PCB which is compatible with a 16-bit analogue I/O card. The FPGA approach combines a dual-core ARM processor with a and uses external 16-bit analogue I/O to implement a complete additive synthesiser. The ARM processor is capable of running a full operating system, and the FPGA can support many digital processing blocks.

The rest of this Chapter is organised as follows:

- a comment on additive synthesis is followed by an introduction to the concept of *backpressure*.
- a high-level description of the functional blocks.
- the Zynq-7000 system and its tool-set is described, followed by a discussion of inter-module communications.
- a more detailed review of backpressure is followed by a description of hardware interfaces used and their implementation
- a detailed description of the functional blocks is followed by a description of their implementation[21].
- an outline of the control firmware and PC software is given.
- results are presented, followed by Conclusions

3.1 Theoretical Background

3.1.1 Additive Synthesis

The timbre of musical instruments can be considered in the light of Fourier theory to consist of multiple harmonic or inharmonic partials[22] or overtones. Each partial is a sine wave of different frequency and amplitude that swells and decays over time due to modulation from an ADSR envelope or a low frequency oscillator. Additive synthesis [60],[61] is a sound synthesis technique that creates timbres by adding sinewaves of different frequencies and amplitude together as described in [1]. In Figure 37 the combination of odd-harmonics to create a square wave is shown using the waveform spectrum on the left-hand side and the resulting wave on the right-hand side. Alternative implementations are possible using pre-computed wavetables or the inverse fast Fourier transform, however, the direct additive synthesis approach allows one to alter timbres in real-time, which permits an amount of expressiveness in playing the instrument.

[21] It would be helpful, in following this discussion, to print out Appendix - FPGA-based Functional Block Diagram in Appendix - D as a reference.

[22] An inharmonic partial is any partial whose frequency is not a positive integer multiple of a common fundamental frequency.

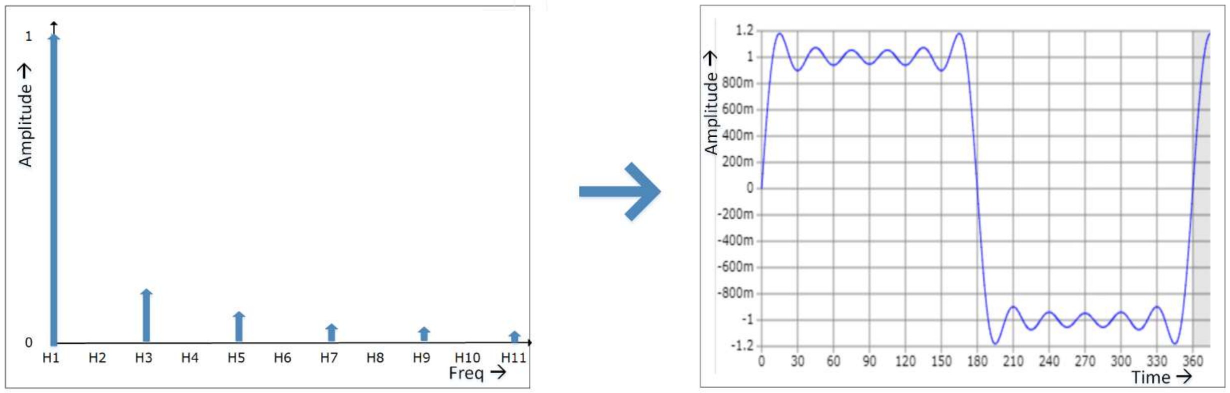


Figure 37 Additive Synthesis

Here, a bank of 16 numerically controlled oscillators (NCO) which outputs vectors of 16 sinewaves, sampled at 100 K samples per second is used. The vector of sinewaves comprises a harmonic series of sine waves at f, 2f, 3f, …. where f is the fundamental frequency. The amplitudes of the vector's components are modulated in real time using one or both of the following two mechanisms:

- All components are multiplied by the same slowly-varying factor, e.g. an ADSR envelope, to vary the volume of the note being played
- Components are multiplied by different slowly-varying factors, e.g. samples from an LPF filter response curve, to vary the spectrum of the note being played.

3.1.2 Backpressure

An original contribution of this work is the development of timing control using backpressure. This can be elaborated as follows:

The custom logic in the FPGA is implemented by compiling a C-based high-level language description into logic blocks, with the inputs and outputs of the blocks comprising state machines, the details of which are inferred. These details, and their timings, as well as the number of clock cycles used by each logic block can change with changes in the compiler and compiler coefficients used. This lack of certainty makes the design of a single-clocked systolic data processing system extremely challenging. Two key ideas overcome this difficulty:

i. The logic blocks in an FPGA are clocked at a rate that is several orders of magnitude greater than the sample rate of the data being output. Provided the chains of logic blocks can be persuaded to output their data in the correct sequence, and have their data ready when asked for, an output DAC clocked at a constant sampling rate can convert this sequence to an audio signal.

ii. We borrow the idea of flow control[23] via backpressure [36] from the world of packet-switched networks. In packet switching, backpressure allows a stage in a multistage switching fabric to prevent the previous stage from sending it any more data. If an output is blocked, backpressure signals quickly propagate

[23] Flow control controls the traffic from a particular sender to a receiver and prevents the receiver from being overwhelmed by the data.

back to the input. Here, backpressure allows a logic block in a tree of blocks to prevent preceding block(s) from sending any more samples, until the output DAC is clocked at the audio sampling rate and requests a sample from its upstream logic block. Backpressure signals quickly propagate back to the data sources.

3.2 Functions

The FPGA synth comprises the high-level blocks shown in Figure 38. The processor controls amplitude and frequency values that are propagated to the block that determines the oscillators outputs, ultimately driving the DAC in order to produce a sound. The function of each block and the use of the backpressure approach within each is described in the text below the figure.

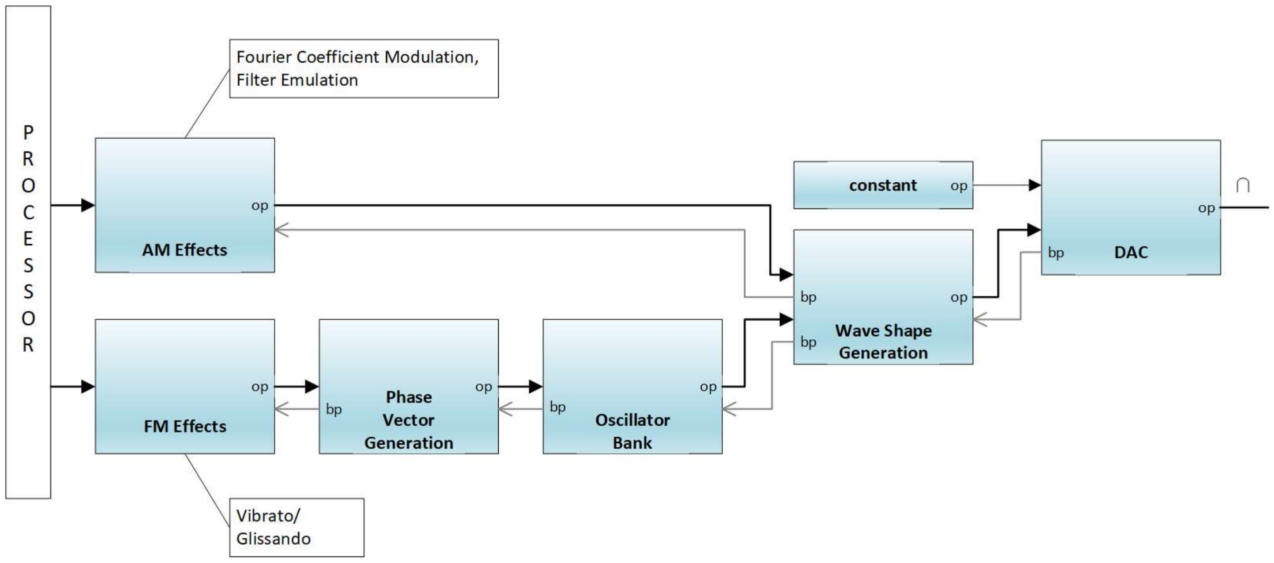


Figure 38 - Functional Blocks

FM Effects: Vibrato/Glissando	This comprises a Vibrato and a Glissando block, the outputs of which were sent to Phase Vector Generation block, to be added to the frequency associated with the note being played. The Vibrato block frequency modulates the Note's frequency, while the Glissando block slew-rate-limits the frequency changes. The block inputs a backpressure signal from Phase Vector Generation in order to regulate the rate of sample generation.
Phase Vector Generation	This comprises a scalar MIDI Note to frequency converter, a scalar frequency to phase increment converter and a phase vector generator. This latter inputs a scalar phase increment and outputs a vector containing phase increments at 16 multiples of the input frequency. The block inputs a backpressure signal from the Oscillator Bank in order to regulate the rate of sample generation, and outputs backpressure signals to the Vibrato and Glissando blocks in order to back-propagate the backpressure signal.
Oscillator Bank	This comprises a block of 16 numerically-controlled oscillators which inputs a vector of 16 phase increments from the Phase Vector Generation block and

	outputs a vector of 16 sine waves to the Wave Shape Generation block. The block inputs a backpressure signal from the Wave Shape Generation block in order to regulate the rate of sample generation, and outputs a backpressure signals to the Phase Vector Generation in order to back-propagate the backpressure signal.
Amplitude Modulation (AM) Effects: Fourier Coefficient Modulation, Filter Emulation	This block inputs vectors of 16 Waveformer coefficients from the CPU. These can vary dynamically, under the control of the CPU, for timbre modification. (This is how filters are emulated). The block inputs a scalar amplitude multiplier from the ADSR, which is multiplied by each element of the input vector. The block outputs vectors of 16 Fourier Coefficients to Wave Shape Generation. The block inputs a backpressure signal from Wave Shape Generation in order to regulate the rate of sample generation.
Wave Shape Generation	This block inputs a vector of 16 modulation coefficients and a vector of 16 sin waves, performs a sum-of-products operation and outputs the resultant additive synthesis waveform to the DAC block. The block inputs a backpressure signal from the DAC block in order to regulate the rate of sample generation, and outputs backpressure signals to the AM Effects and Oscillator Banks in order to back-propagate the backpressure signal.
DAC Block	This block comprises a SPI driver for the DAC, a 16-bit DAC and an anti-aliasing filter. The block inputs a constant and a stream of 16-bit scalars from the Wave Shape Generator. The constant is used to divide down the system clock in order to form a 100 KHz clock. The block outputs a backpressure signal to throttle the input data stream to 100 K samples/second.

3.3 HARDWARE SUPPORT FRAMEWORK

3.3.1 Zynq-7000

The Zynq-7000, shown in Figure 39, comprises an ARM based processing system (PS) and some programmable logic (PL). The PS includes two Cortex-A9 processor cores, a dedicated DDR memory controller, and IO peripherals. The PL is based on the Xilinx 7- Series FPGA fabric. The two areas of the device, PS and PL, are linked by a set of industry-standard Advanced Extensible Interface-4 (AXI4) interfaces. These interfaces allow the designer to implement custom logic in the PL which can be connected to the PS and are mapped to the processor's memory map. A functional block diagram of the FPGA is given in Figure 103.

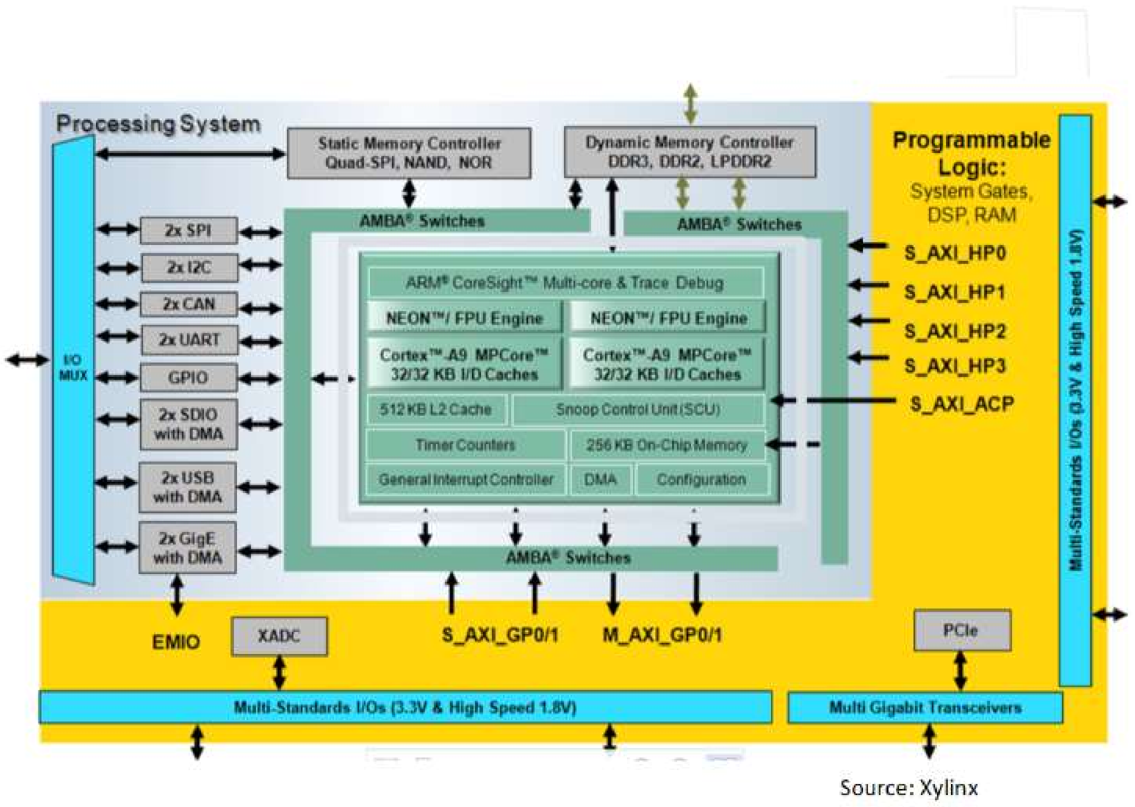

Processing System
Static Memory Controller Quad-SPI, NAND, NOR
Dynamic Memory Controller DDR3, DDR2, LPDDR2
Programmable Logic: System Gates, DSP, RAM
AMBA® Switches
AMBA® Switches
I/O MUX
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
ARM® CoreSight™ Multi-core & Trace Debug
NEON™/ FPU Engine
NEON™/ FPU Engine
Cortex™-A9 MPCore™ 32/32 KB I/D Caches
Cortex™-A9 MPCore™ 32/32 KB I/D Caches
512 KB L2 Cache
Snoop Control Unit (SCU)
Timer Counters
256 KB On-Chip Memory
General Interrupt Controller
DMA
Configuration
AMBA® Switches
S_AXI_HP0
S_AXI_HP1
S_AXI_HP2
S_AXI_HP3
S_AXI_ACP
Multi-Standards I/Os (3.3V & High Speed 1.8V)
EMIO
XADC
S_AXI_GP0/1
M_AXI_GP0/1
PCIe
Multi-Standards I/Os (3.3V & High Speed 1.8V)
Multi Gigabit Transceivers
Source: Xylinx

Figure 39 Zynq IC Architecture

One creates a block of custom logic in the PL, and then adds control and status monitoring capabilities by using memory-mapped registers which the processors can access via the AXI4 interconnect.

3.3.2 Tools

Xilinx provides the Vivado design suite IDE. Its High-Level Synthesis (HLS) tool transforms a C specification into a register transfer level (RTL) implementation that is then instantiated on an FPGA.

Thus, algorithms are developed in C++, at a high level of abstraction, and the functional correctness of the design is validated at the same level of abstraction. Optimization directives are used to control the synthesis in order to create a specific hardware implementation. However, mastering this involves a steep learning curve. Considerable hardware design experience is needed in order to succeed. Compiling an FPGA design takes considerable time, so one needs to ensure that the C-based IP block is comprehensively tested before use.

3.4 Inter-Module Communications

Communications between modules in the Zynq's Programmable Logic section use the following interfaces:

- AXI4-Lite: Processing System to Programmable Logic communication
- AXI4_Stream: data flow between Programmable Logic modules

The next subsection describes the details underlying the AXI protocol taking in both interfaces

3.4.1 AXI Protocol

The ARM Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, bus-based on-chip interconnect specification for the connection of functional blocks in system-on-a-chip (SoC) designs. The AMBA AXI protocol is a high-speed extension of AMBA which adds, amongst other things, point-to-point communications.

Basic AXI Signalling involves five Channels

- Read Address Channel
- Read Data Channel
- Write Address Channel
- Write Data Channel
- Write Response Channel

ARM [2] states "the AXI protocol is burst-based. Every transaction has address and control information on the address channel that describes the nature of the data to be transferred as shown in Figure 40.

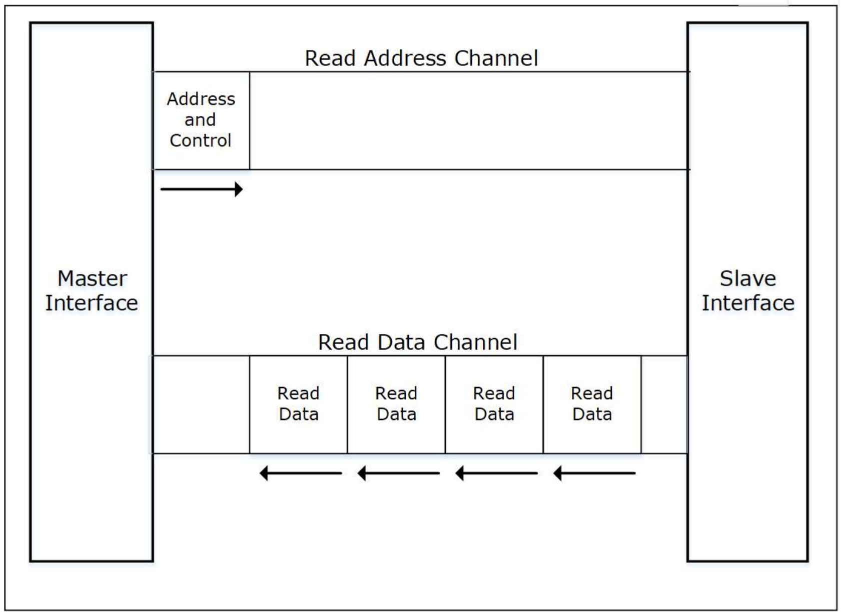


Figure 40 AXI Read Channel

The data are transferred between master and slave using a write data channel to the slave or a read data channel to the master. In write transactions, in which all the data flow from the master to the slave, the AXI protocol has an additional write response channel to allow the slave to signal to the master the completion of the write transaction, Figure 41.

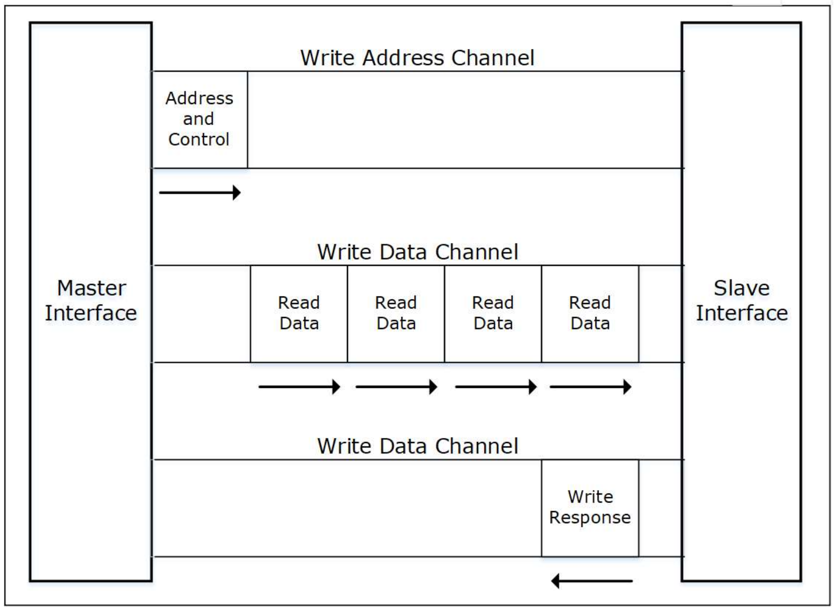


Figure 41 AXI Write Channel

3.4.1.1 Handshake process

ARM [2] states: "all five channels use the same VALID/READY handshake to transfer data and control information. This two-way flow control mechanism enables both the master and slave to control the rate at which the data and control information moves. The source generates the VALID signal to indicate when the data or control information is available. The destination generates the READY signal to indicate that it accepts the data or control information. Transfer occurs only when both the VALID and READY signals are HIGH"

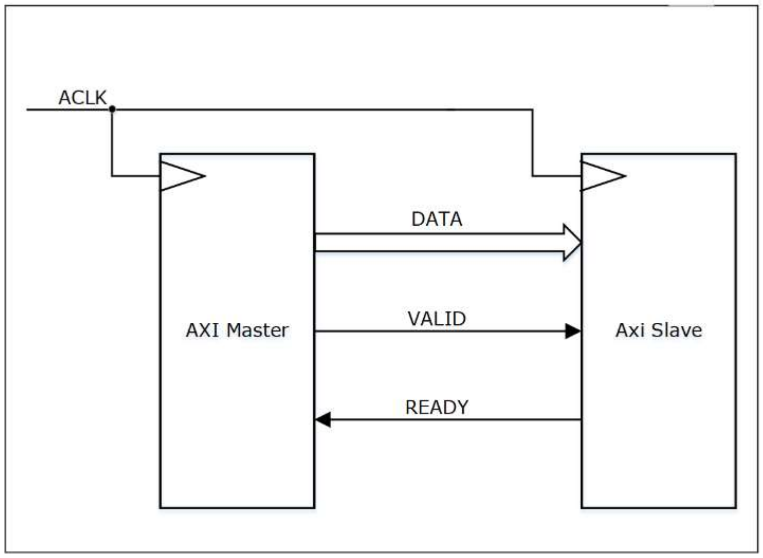


Figure 42 Basic Handshake

Note: Timing Diagrams

Figure 43 Timing Diagram , shows the style of timing diagram used in this document to illustrate the operation of Finite State Machines [39].

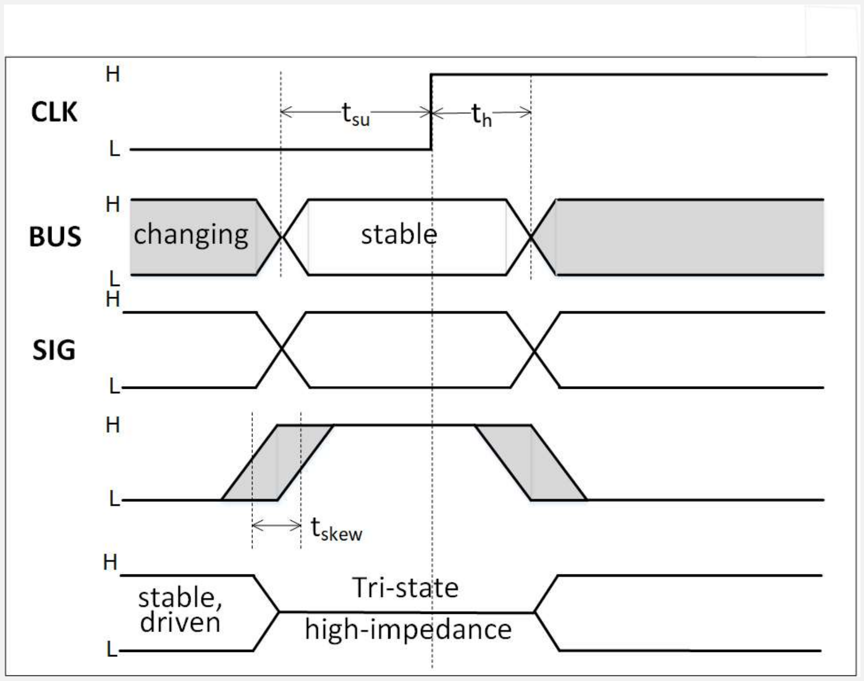


Figure 43 Timing Diagram

The digital systems discussed are assumed to be driven by the positive edge of a clock. Transitions on the inputs to Finite State Machines (FSM) are shown on the immediate left of the clock: These inputs are assumed to be stable from a setup time before the clock edge until a hold time after the clock edge. Transitions resulting from the action of a clock edge are shown to the immediate left of the succeeding clock edge. Shaded areas either represent changing signals on buses, or clock timing skew on signal lines. The usual conventions for driven and high-impedance signals are followed.

The sequence of transitions in the diagrams from left to right, indicate the sequence of the FSM's input and output transitions only. Relative timing should not be inferred.

The following three paragraphs show how inter-module data transfer works over various sequences of assertion of **`valid`** and **`ready`** signals

In Figure 44, the source presents the data or control information and drives the VALID signal HIGH. The data or control information from the source remains stable until the destination drives the READY signal HIGH, indicating that it accepts the data or control information. The arrow shows when the transfer occurs."

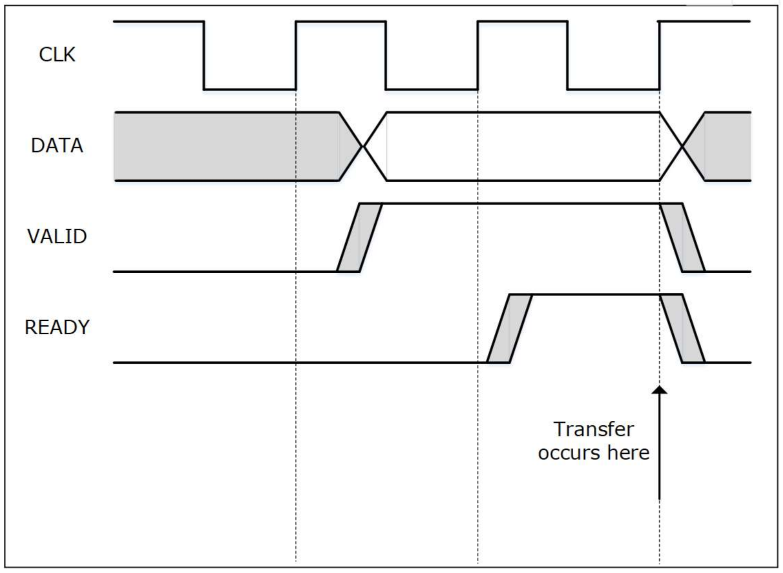


Figure 44 Valid before Ready Handshake

In Figure 45 the destination drives **READY** HIGH before the data or control information is valid. This indicates that the destination can accept the data or control information in a single cycle as soon as it becomes valid."

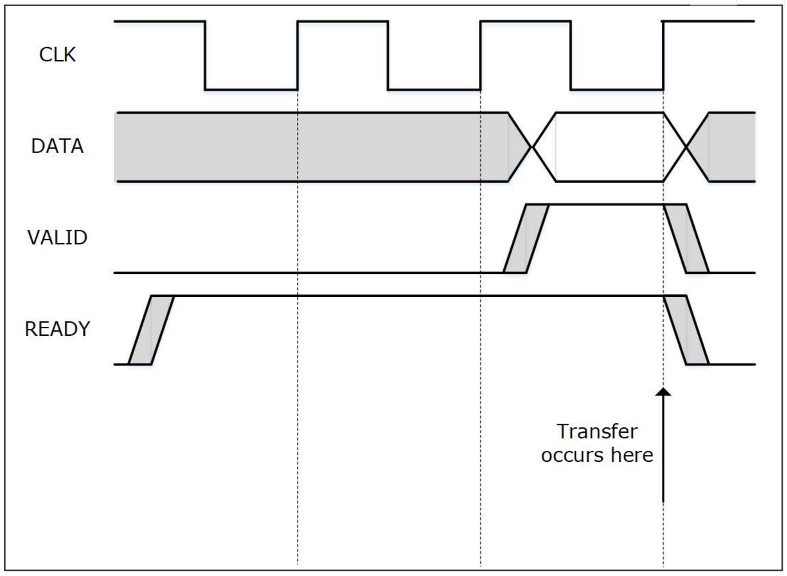


Figure 45 Ready before Valid Handshake

In Figure 46, both the source and destination happen to indicate in the same cycle that they can transfer the data or control information. In this case the transfer occurs immediately.

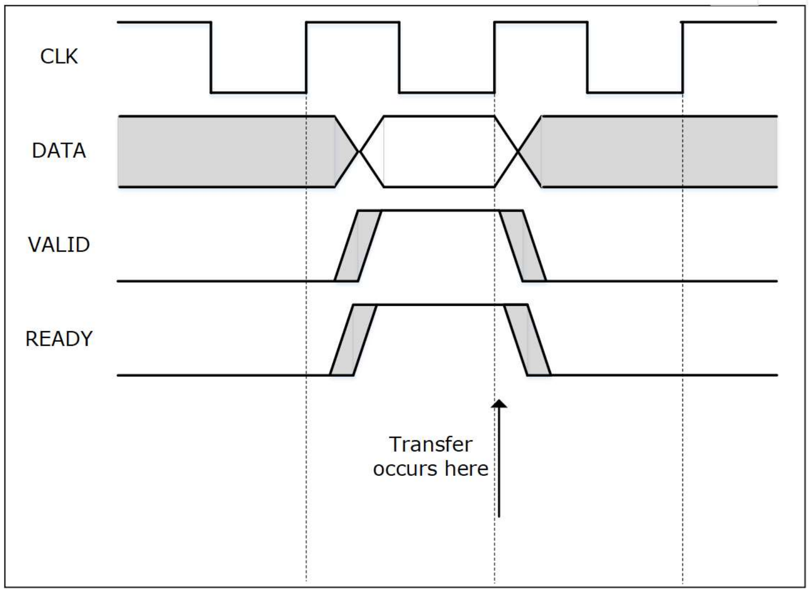


Figure 46 Valid with Ready Handshake

3.4.2 AXI Interfaces

The AXI specifications describe an interface between a single AXI master and a single AXI slave, representing IP cores that exchange information with each other. Memory mapped AXI masters and slaves can be connected together using a structure called an Interconnect block, as detailed below;

3.4.2.1 Full AXI

AXI4 provides separate data and address connections for reads and writes, which allows simultaneous, bidirectional data transfer. AXI4 requires a single address and then bursts up to 256 words of data. Data Width is parameterizable to 32, 64, 128, 256, 512, or 1024 bits.

3.4.2.2 AXI Interconnect

The AXI Interconnect core IP connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices.

The AXI-Lite protocol described below is used for communicating from the CPU to the processing blocks. The AXI Stream protocol is used for communications between the processing blocks.

3.4.2.3 AXI-Lite

AXI4-Lite is similar to AXI4 with some exceptions:

- bursting is not supported.
- data width is restricted to 32 or 64 bits. (Xilinx IP only supports 32-bits)
- Very small footprint
- Bridging to AXI4 is handled automatically by AXI_Interconnect.

AXI4-Lite, together with AXI Interconnect is used in this design to connect the Processing System, which encompasses the CPU, to the Programmable Logic system, which contains all the processing blocks.

AXI_Lite data in this design can carry scalar data, representing control parameters, or vectorised sets of control parameters, used to operate on the sets of frequencies associated with additive synthesis.

3.4.2.4 AXI-Stream

AXI-Stream has no address channel, no read and write, and always transfers data from master to slave. It's effectively an AXI4 "write data" channel. It has unlimited burst length, and virtually same signalling as AXI Data Channels. The protocol allows merging, packing, and width conversion. AXI-Stream is used in this design for data flow between processing blocks. The design uses a subset of the available AXI_Stream signals as shown in Figure 47, and described in Table 1.

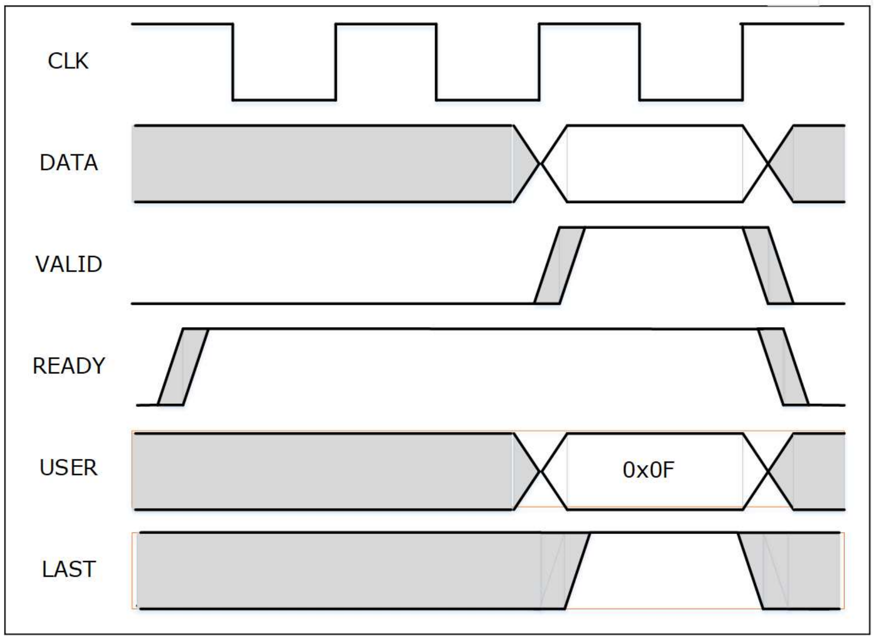


Figure 47 AXI Streaming signals

Signal	Source	Description
ACLK	Clock source	The global clock signal.
ARESETn	Reset source	The global reset signal. Active-LOW.
TVALID	Master	Indicates that the master is driving a valid transfer. A transfer takes place when both TVALID and TREADY are asserted.
TREADY	Slave	Indicates that the slave can accept a transfer in the current cycle.
TDATA [15:0]	Master	Primary payload across the interface. Width is an integer number of bytes.
TLAST	Master	Indicates the boundary of a packet.
TUSER [(3:0]	Master	User defined sideband information: in this design, it's used as an index into vector data.

Table 2 Relevant AXI-Stream Signals

AXI_Stream data in this design can carry scalar data**,** representing audio samples, or vector data, representing the frequencies associated with additive synthesis, or else vectorised parameters.

3.4.3 BackPressure

The AXI signals Tvalid and Tready are used for flow control. If the transmitter is not ready to send more data, the Tvalid signal is not asserted as in Figure 45. Similarly, when the receive side of an AXI stream is not ready to receive more data, it de-asserts the Tready signal Figure 44.

The output Digital to Analogue Converter (DAC) in Figure 48 is clocked at a constant rate of 100 KHz, in order to avoid jitter on the analogue output signal. Upstream processing blocks are clocked at a rate high enough to ensure that they can produce sample data at a rate that is higher than that which can be consumed by the DAC.

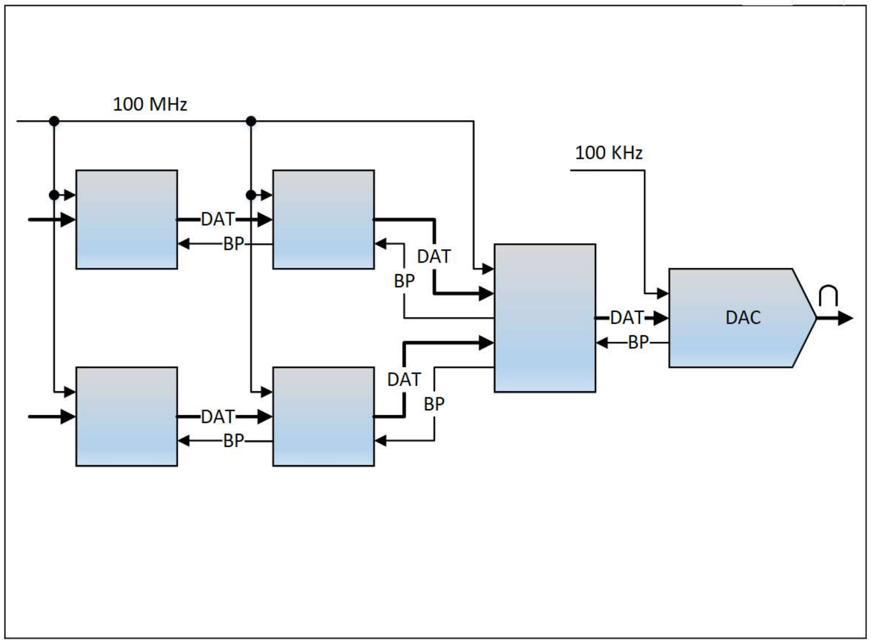


Figure 48 Backpressure System

The DAC asserts backpressure until it requires a new sample to output, whereupon it briefly deasserts backpressure, allowing its upstream processing block to output a sample.

Some of the processing blocks operate on vectors of samples and produce their output data in bursts. First-in, first-out memories (FIFOs) are used to smooth out these bursts of data and are shown in Figure 49. A functional block which is upstream of a FIFO must interrogate the FIFO to ensure that there's room for a burst of data, before transmitting the burst. A functional block which is downstream of a FIFO uses the Valid/Ready handshake in the normal way to acquire one sample at a time. The length of a FIFO should be short, in order to avoid any noticeable latency.

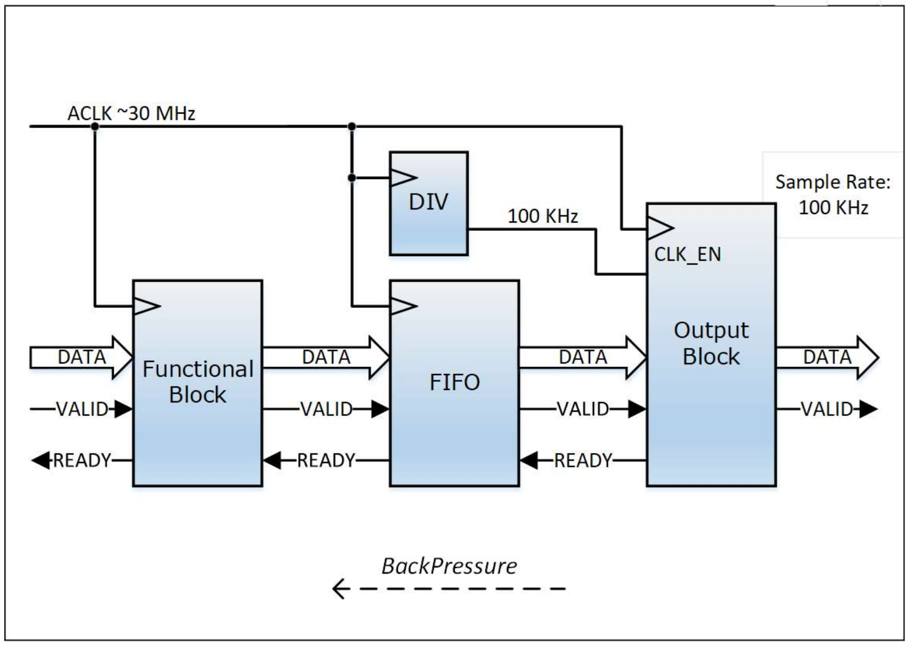


Figure 49 Backpressure, detail

The normal state of a processing block is to be stalled, with a data sample in its output register, waiting for READY to be asserted by its downstream neighbour. Backpressure ensures that the processing blocks operate in lock-step. The actual time at which the blocks output samples isn't relevant, provided:

1. The output DAC never gets starved of data
2. The total number of samples in the pipeline is less than that which would equate to a noticeable latency, when the sequence of samples is translated to a timed stream of samples at the DAC.

3.4.4 Block Parameters

Most functional blocks are controlled in real-time by a set of parameters transmitted by the CPU over the AXI4-Lite interface described above. This happens at a rate determined by a timed interrupt service routine (ISR) , i.e. 200 Hz. Because the timed ISR is itself subject to interruption by higher priority ISRs, the parameters are transmitted asynchronously to the clocks on the functional blocks.

Ping-pong buffers[24] and a mutex[25] are used to ensure that block parameters are not corrupted by race conditions.

[24] A ping-pong buffer is a double-buffering technique that uses two buffers to feed a single endpoint: while one buffer is being read, the other is being filled.
[25] A mutex is a locking mechanism used to synchronize access to a shared resource. Only one task at a time can acquire the mutex, and hence the resource. This prevents corruption of the resource's data.

3.5 Hardware Interfaces

3.5.1 High-Level Synthesis

The Xilinx High-Level Synthesis (HLS) tool transforms a C specification into a register transfer level (RTL) implementation that is then synthesized into a field programmable gate array (FPGA) (See [5]).

Algorithms are developed in C++, at a high level of abstraction, and the functional correctness of the design is validated at the same level of abstraction. Optimization directives are used to control the synthesis in order to create a specific hardware implementation.

3.5.2 C code

C code is made up of functions which represent the design hierarchy. The arguments of the top-level function are transformed into hardware interfaces with an I/O Protocol. Consider the following code fragment from [5]:

```
#include "sum_io.h"

dout_t sum_io(din_t in1, din_t in2, dio_t *sum) {
 dout_t temp;
 *sum = in1 + in2 + *sum;
 temp = in1 + in2;
 return temp;
}
```

This example includes:

- Two pass-by-value inputs in1 and in2.
- A pointer sum that is both read from and written to.
- A function return, the value of temp.

With the default interface synthesis settings, the design is synthesized into an RTL block with ports as shown in the Figure 50:

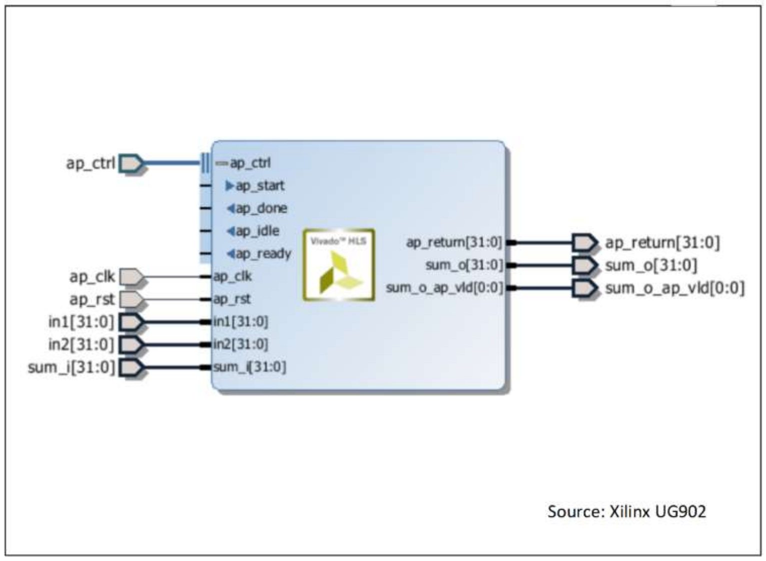


Figure 50 RTL Ports After Default Interface Synthesis

Three types of ports, the functions of which are explained immediately below, are created:

3.5.2.1 Clock and Reset ports:

ap_clk and ap_rst.

3.5.2.2 Block-Level interface protocol

These signals control the block, independently of any port-level I/O protocols:

ap_start	block can start processing data
ap_done	block is ready to accept new inputs
ap_ready	block is ready to accept new inputs
ap_idle	block is idle

3.5.2.3 Port Level interface protocols.

These are created for each argument in the top-level function and the function return. In this example, these ports are:

in1 in2	input pass-by-value arguments implemented as simple wire ports with no associated handshaking signal.
sum_i sum_o	arguments which are both read from and writes to are split into separate input and output ports, sum_i and sum_o
sum_o_ap_vld	output valid port which indicates when the data on the port are valid.
ap_return	Implemented if the function has a return value.

3.5.3 Port interfaces used in this design

The following Port interfaces are used in this design:

- AXI4_Stream: data flow between processing modules
- AXI4-Lite: control flow from CPU.

3.5.3.1 AXI4_Stream Port

The templates in the following code fragment are used to define the data structures used.

```
template<int D, int U, int TI, int TD>
 struct ap_axis{
 ap_int<D> data; // arbitrary precision type
 ap_uint<(D+7)/8> keep; //
 ap_uint<(D+7)/8> strb; //
 ap_uint<U> user; //
 ap_uint<1> last; //
 ap_uint<TI> id; //
 ap_uint<TD> dest; //
 };
```

The following fields in the structure are used:

data	16-bit, 24-bit or 32-bit data to be transferred
user	4-bit field used for the index of a vector quantity to be transferred.
last	1-bit used to indicate the last word in a vector

A typical declaration using the template would be as in the following code fragment, indicating 16 data bits and 4 user bits:

```
// I/O Streams
typedef ap_axis <16,4,1,1> AXI_T;
typedef hls::stream<AXI_T> STREAM_T;
```

The C++ template class hls::stream<> is used for modelling streaming data structures. The streams have the following attributes:

- In the C code, a hls::stream<> behaves like a FIFO of infinite depth.
- They are read from and written to sequentially.

- An hls::stream<> on the top-level interface is by default implemented with an ap_FIFO interface. However, in this design pragmas [26](see below) are used to direct the RTL synthesiser to implement them as an AXI-Stream interface (axis).

The basic accesses to an hls::stream<> object are blocking reads and writes using class methods. These methods stall (block) execution if a read is attempted on an empty stream FIFO, a write is attempted to a full stream FIFO[27]. It is important to use blocking reads and writes if the backpressure system is to work.

The hls::stream<> object also provides an empty() method, which returns true if the hls::stream<> is empty. This is useful for those processing modules which input streams different rates. (Typically, control streams are produced at a 200 Hz rate, while data streams are produced at 100 KHz.) A module can test a lower-rate input with the stream's empty() method in order to avoid using a blocking read() on the lower rate stream.

The following code fragment illustrates the use of a top-level hls::stream<> interface:

```
void ampl_mod( STREAM_T &strm_out_v,
               STREAM_T &strm_in_v,
               MOD_STREAM_T &mod_in){
// axis: Implements all ports as AXI4-Stream interface.
#pragma HLS INTERFACE axis port=strm_out_v
#pragma HLS INTERFACE axis port=strm_in_v
#pragma HLS INTERFACE axis port=mod_in
// No control interface - auto-start as soon as
// there's an input frame
 #pragma HLS INTERFACE ap_ctrl_none port=return
 …
}
```

Here, &strm_out_v is a reference to a hls::stream of AXI_T types, with the latter specifying an AXI4-Stream Interface with Side-Channels. With the following definition:

typedef ap_axis <16,4,1,1> AXI_T;

16 data bits and 4 user bits are specified, and the port in Figure 51 is generated[28], as specified by the pragma directive:

```
#pragma HLS INTERFACE axis port=strm_out_v
```

[26] A `#pragma` directive is used to instruct the compiler to use an implementation-dependent features.

[27] An `axis` interface can be a FIFO with a depth of one.

[28] The signals keep, strb, id and dest are not used in this design, but are included so as to avoid possible compatibility issues with other IP blocks.

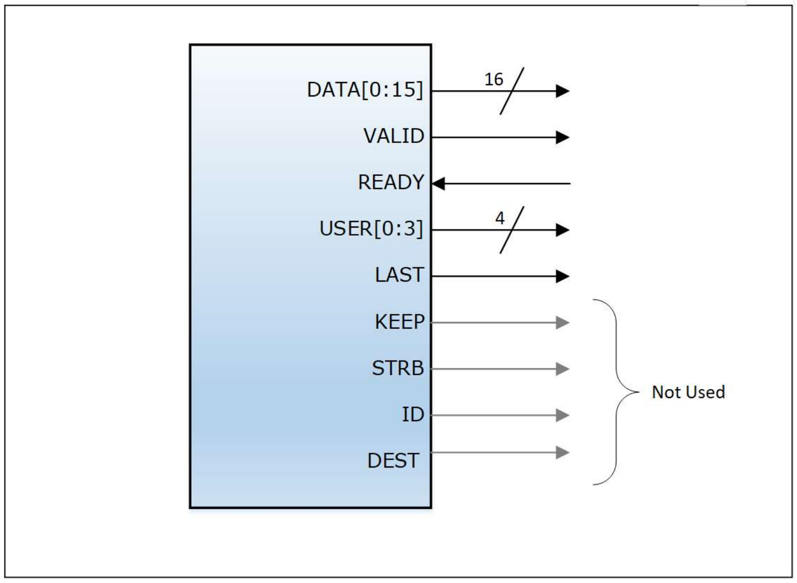


Figure 51 AXIS O/P Port

The valid and ready signals are mandatory signals in an AXI4-Stream and will always be implemented.

&strm_in_v and &mod_in are similarly generated as AXI Stream I/P ports. The function's return is synthesized into an output port called ap_return, but even if the function has no return value, we can disable Block-level handshakes with the following pragma:

```
#pragma HLS INTERFACE ap_ctrl_none port=return
```

3.5.3.2 AXI4_Lite Port

An AXI4-Lite interface is used to allow a processing module to be controlled by the CPU. The following code fragment illustrates the use of a top-level s_axilite interface:

```
void glide(STREAM_T &NOTE_OUT,
        STREAM_T &NOTE_IN,
        int32_t slew ){
// s_axilite: Implements all ports as an AXI4-Lite I/F
// -bundle: Groups function arguments into AXI ports
#pragma HLS INTERFACE s_axilite port=slew bundle=ctrl
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl

// axis: Implements all ports as an AXI4-Stream interface.
#pragma HLS INTERFACE axis port=NOTE_OUT
#pragma HLS INTERFACE axis port=NOTE_IN

static uint32_t note = 69 * 1024;
#pragma HLS RESET variable=note
 …
}
```

Here, &NOTE_OUT is a reference to a hls::stream of AXI_T types, with the latter specifying an AXI4-Stream with 32 data bits and 1 user bit:

typedef ap_axis <32,1,1,1> AXI_T;

The pragma:

```
#pragma HLS INTERFACE axis port=NOTE_OUT
```

generates an AXI4-Stream Interface.

The `slew` argument, directed on by the pragma:

```
#pragma HLS INTERFACE s_axilite port=slew bundle=ctrl ,
```

generates an `s_axilite` interface. The notation, `bundle=ctrl` bundles all the `s_axilite` interfaces so marked, into one interface. If the function return is also specified as an AXI4-Lite interface (s_axilite) all the ports in the block-level interface are grouped into the AXI4-Lite interface.

Vivado HLS generates C driver files for use with the code running on a processor. This provides a set of C application program interface (API) functions, which allows one to easily control the hardware from the software.

3.5.4 PC to Zynq communications

The PC manages the Graphic User Interface (GUI) and reads the MIDI Keyboard, sending JSON[62] messages to the Zynq CPU via a Serial_Over_USB interface in order to control the Synth. The following code fragment shows an Attack-Decay-Sustain-Release (ADSR) message:

```
{
 "dst":{
 "adsrv":0
 },
 "dat":{
 "a":4096,
 "d":200,
 "s":32768,
 "r":100,
 "v":16384
 }
}
```

and the following code fragment shows a Note-on message:

```
{
 "non":{
 "chn":"1",
 "num":"60",
 "vel":"100"
 }
}
```

The messages are self-explanatory, and while more verbose than, say, MIDI messages, they are easy to parse and debug.

3.6 Functional Description

. What follows expands on the functional blocks outlined in Section 3.2 Functions

3.6.1 Additive Waveform Synthesis[29] (AWS)

AWS is performed by two functional blocks: the Direct Digital Synthesis (DDS) block and the waveformer block.

3.6.1.1 DDS

The DDS [14] block is a bank of 16 NCOs which input a vector of 16 phase increments, at a rate of 200 vectors per second, and outputs vectors of 16 sin waves, sampled at 100 K samples per second. The input phase increments specify a harmonic series of sin waves at *f*, 2*f*, 3*f*, …. where *f* is the fundamental frequency. The fundamental can vary, for e.g. vibrato or glissando, while the relationship between the harmonics remains constant.

3.6.1.2 Waveformer

The Waveformer inputs vectors of 16 sin waves, and vectors of 16 modulation coefficients, both sampled at 100 K samples per second. It outputs scalars sampled at 100 KHz. The block multiplies each input sin wave by its associated input coefficient, accumulating and normalising the results for output.

The coefficients can vary in real time for e.g. timbre modification or simulation of filters.

3.6.2 DAC[30]

The DAC Driver block Inputs 16-bit samples at 100 KHz, manages the 16-bit DAC, and communicates with it over its SPI interface.

The DAC Driver block is clocked at a steady rate by a hardware timer, and asserts backpressure on its upstream sources in order to throttle the input sample rate.

Finally the DAC analogue output is filtered by a 4-pole analogue reconstruction filter with a bandwidth of 5 KHz, before being output by a buffer amplifier.

3.6.3 Waveformer Coefficient Generation[31]

3.6.3.1 Amplitude Modulator

The Amplitude Modulator block inputs vectors of 16 Waveformer coefficients from the CPU, delivered at 200 vectors per second. These can vary dynamically, for timbre modification. It also inputs a scalar amplitude multiplier from the ADSR. Each element of the input vector is multiplied by the amplitude

[29] Please refer to the sections labelled: “OSCILLATOR BANK” and “WAVE SHAPE GEN” in Figure 103 - FPGA-based Functional Block Diagram
[30] Please refer to the section labelled: “DAC, ANTI-ALIAS” in Figure 103 - FPGA-based Functional Block Diagram
[31] Please refer to the section labelled: “WAVEFORMER COEFFS” in Figure 103 - FPGA-based Functional Block Diagram

multiplier and normalised before outputting vectors of 16 sin waves sampled at 100 K samples per second.

3.6.3.2 ADSR

The ADSR block inputs the following signals from the CPU, delivered at 200 samples per second:

- attack time
- decayRate
- releaseRate
- sustainLevel
- Volume
- NoteOnOff

The block responds to the NoteOnOff signal, producing an ADSR envelope whose characteristics are defined by the other input parameters, and outputs a scalar amplitude multiplier at a rate of 200 samples per second.

3.6.4 Frequency Vector Generator[32]

3.6.4.1 MIDI Note to Freq Block

The MIDI Note to Freq Block inputs an 18-bit MIDI Note parameter (MIDI Freq * 1024) and a 16-bit frequency modulation parameter. The latter is converted to a signed number before being added to the note value. Then a note-to-frequency operation is performed before outputting modulated 24-bit frequency samples to the Frequency to Phase block.

Inputs to the block are linear values that correspond to the way human beings think about musical notes, and outputs are non-linear, corresponding to the way we hear music. This makes the Vibrato and Glissando functions easier to design.

3.6.4.2 Frequency to Phase block

The Frequency to Phase block inputs 24-bit frequency samples, multiplies them by a constant and then scales the values before outputting 24-bit phase increments at a rate of 200 scalars per second to the Phase Vector Generator block.

3.6.4.3 Phase Vector Generator block

The Phase Vector Generator block inputs a phase increment scalers and outputs a stream of vectors containing the instantaneous phases of a set of harmonically related sin waves to the DDS at a rate of 200 vectors per second.

3.6.5 Vibrato and Glissando[33]

3.6.5.1 Vibrato

The Low Frequency Oscillator (LFO) block inputs scalar phase increments from the CPU at 200 scalars per second, and outputs a 24-bit sin wave to an amplitude modulator block. The latter inputs multiplier values

[32] Please refer to the section labelled: "FREQ VECTOR GEN" in Figure 103 - FPGA-based Functional Block Diagram

[33] Please refer to the sections labelled: "VIBRATO" and "GLISSANDO" in Figure 103 - FPGA-based Functional Block Diagram

from the CPU and outputs the modulated sin wave to the frequency modulation input of the MIDI Note to Freq Block.

Vibrato FM is specified in the linear domain.

3.6.5.2 Glissando

The Glide Block inputs an enable signal, an 18-bit slew rate parameter in increments per 10 micro-seconds and an 18-bit MIDI Note parameter. It slew-rate limits the transition between notes and outputs an 18-bit MIDI Note to the MIDI Note to Freq Block. Glissando is specified in the linear domain.

3.7 IMPLEMENTATION

Please note that the code in the following blocks, while it can and should be run on a CPU for unit testing purposes, is really the specification for functions to be implemented in hardware, to be generated by the Vivado tool suite. The hardware manifestations may therefore function in a slightly different manner from that which one might expect.

Note in particular that the compiler pragmas have a significant effect on the way that the interfaces are generated. An attempt was made to capture this effect by following the description of each set of pragmas included in a module by a schematic of the interface so generated.

It might be useful to refer the Vivado Block Diagram in Appendix E, (Figure 104 - Vivado Block Design) as you follow the rest of this section.

3.7.1 Additive Waveform Generation

This comprises three blocks:

- Phase Vector Generation
- Multiple Channel Direct Digital Synthesis
- Additive Waveform Synthesis

3.7.1.1 Phase Vector Generation

This inputs a phase scaler and outputs a repeating stream of vectors containing the instantaneous phases of a set of harmonically related sin waves. On detection of the leading edge of the Note_on signal, the RESYNC bit of the output data is asserted, in order to instruct the downstream block to reset the accumulated phase of the channel in question.

This block has the C prototype:

```
// DDS PHASE Channel TDATA Structure:
//      bit 24 RESYNC
//      bit [23:0]     PINC

typedef ap_axis <32,1,1,1> AXI_PHASE_IN_T;
typedef hls::stream<AXI_PHASE_IN_T>
 PHASE_IN_STREAM_T;
typedef ap_axis <32,4,1,1> AXI_PHASE_OUT_T;
typedef hls::stream<AXI_PHASE_OUT_T>
 PHASE_OUT_STREAM_T;
void phase_vector_gen(
 PHASE_OUT_STREAM_T &PHASE_VEC,
 PHASE_IN_STREAM_T &phase_in,
 bool note_on);
```

phase_in	stream of 32-bit scalar phase increment values
note_on	MIDI Note on signal
PHASE_VEC	Output stream of 16 * 32-bit phase increment vectors. Each element comprises a 24-bit phase increment value and a boolean resync value for the DDS

Using the pragmas:

```
#pragma HLS INTERFACE s_axilite port=note_on bundle=ctrl
#pragma HLS INTERFACE axis port=phase_in
#pragma HLS INTERFACE axis port=PHASE_VEC
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl
```

generates the following hardware interface:

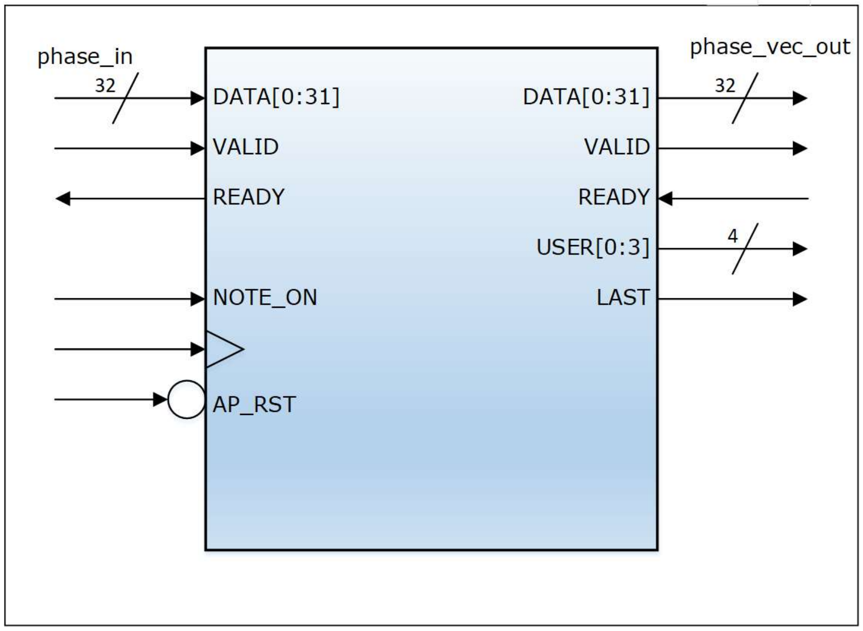


Figure 52 phase vector gen ports

The following code fragment[34] generates the RTL:

[34] The '>>' operator is overloaded to allow use in a manner similar to the stream extraction operator for a C++ stream

```
AXI_PHASE_IN_T phase
AXI_PHASE_OUT_T axis_val;
uint32_t data;

phase_in >> phase;
phase_inc = phase.data & PHASE_MSK;

// detect leading edge of note_on
resync = note_on && !old_note_on;
old_note_on = note_on;
//   for   each   input   phase   scalar,   output   a   vector   of   phase   values   at
 //   frequencies,   f,   2*f,   3*f,   etc.,   where   f   is   the   frequency   of   the
 // input phase value
for(int i = 0; i < NUM_CHANS; i++){
 axis_val.data += phase_inc;
 if(resync){
 data = axis_val.data.to_int();
 data |= RESYNC_BIT;
 axis_val.data = data;
 }
 // load an index value into the output vector element’s USER field
 axis_val.user = i;
 //   if   this   is   the   last   element   of   the   output   vector,   assert
 // the output vector element’s LAST field
 if(i == NUM_CHANS-1){
 axis_val.last = 1;
 resync = false;
 } else {
 axis_val.last = 0;
 }
 // output a vector element
 PHASE_VEC << axis_val;
}
```

For each phase increment input, representing a frequency *f*, the function outputs a vector of 16 elements representing the frequencies: *f*, 2**f*, 3**f*, etc.

3.7.1.2 Multiple Channel Direct Digital Synthesis

This block uses the Xilinx DDS IP described in [15]. It comprises a Phase Generator and Sine LUT.

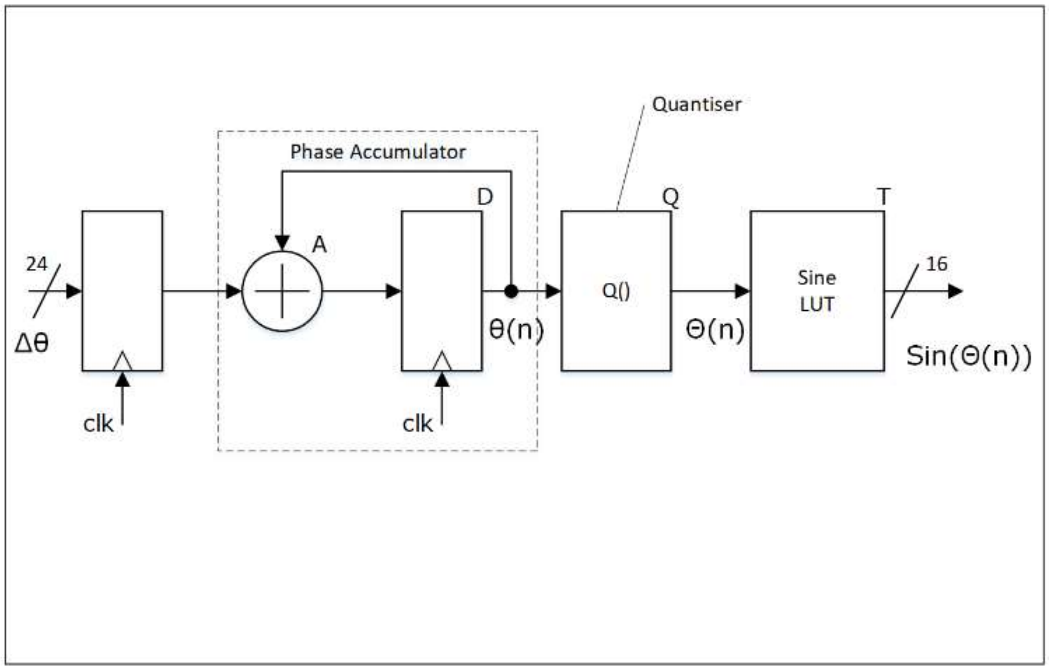


Figure 53 DDS (simplified)

The Phase Accumulator (components D and A in Figure 53) computes a phase slope that is mapped to a sinusoid by the lookup table T. The quantizer Q inputs a phase angle and generates a lower precision representation, which addresses a lookup table that performs the mapping from phase-space to time.

We use the DDS's time-division (TDM) multichannel capability, to support 16 channels as shown in Figure 54.

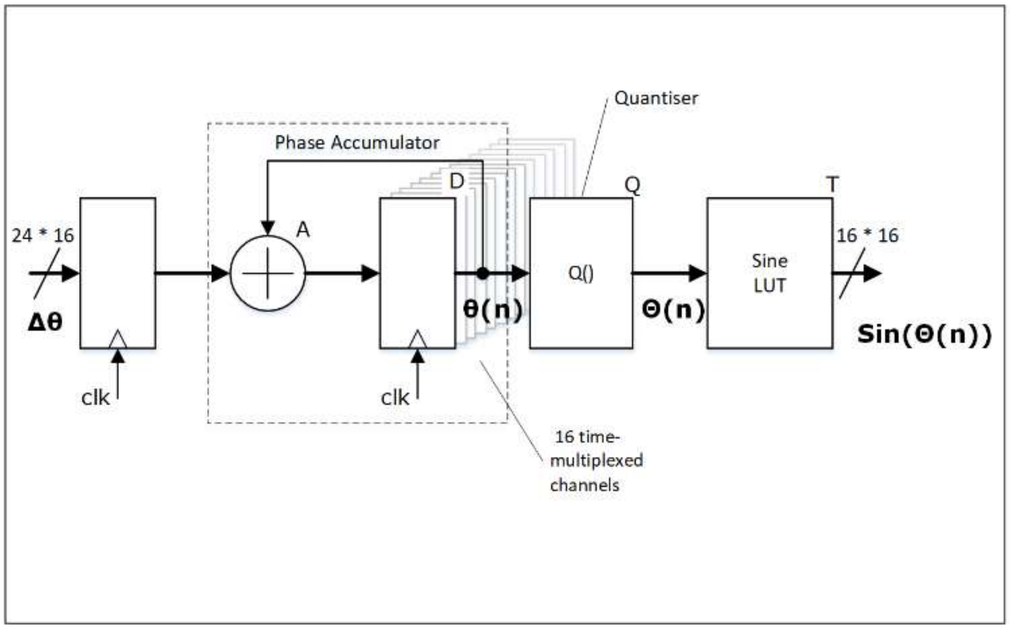


Figure 54 - Multiplexed DDS

Thus the DDS inputs a vector of 16 phase increments at a sample rate of 100 KHz, (throttled by backpressure), and outputs a vector of samples of 16 sinusoids as depicted as a spectrum in the frequency domain in Figure 55 below:

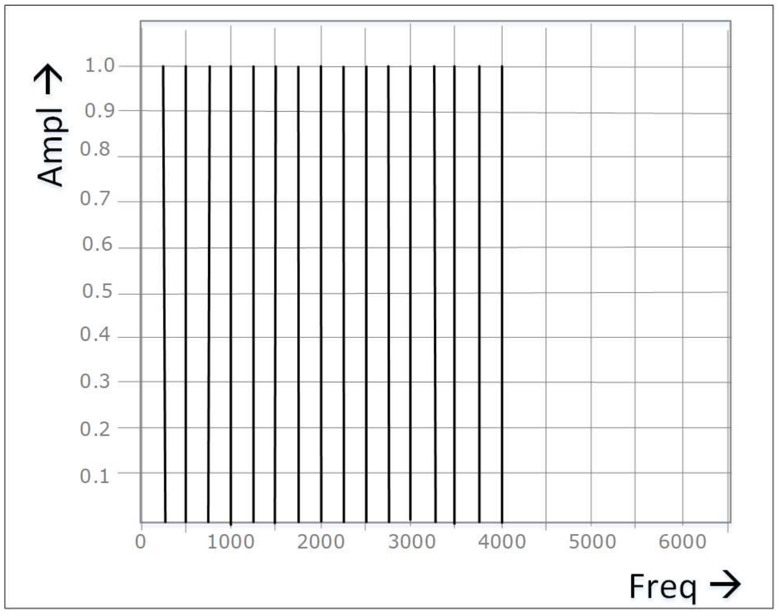


Figure 55 DDS O/P frequencies

3.7.1.3 Additive Waveform Synthesis (Waveformer)

This block performs additive waveform synthesis, inputting a stream of vectors containing samples of harmonically related sin waves.

This block has the C prototype:

```
typedef ap_axis <16,4,1,1> AXI_T;
typedef hls::stream<AXI_T> STREAM_T;

void Waveformer(      STREAM_T &strm_out,
              STREAM_T &strm_in_v,
              STREAM_T &cfg_in_v);
```

strm_in_v	stream of 16 * 16-bit harmonically related sinusoids.		
	side-channel data:	**last** indicates the last element of a vector.	
cfg_in_v	stream of 16 * 16-bit multiplier coefficients		
	cfg_in vectors arrive at a slower rate than strm_in vectors		
	side-channel data:	**last**:	indicates the last element of a vector.
		user:	index of the vector's element
PHASE_VEC	stream of 16-bit scalar waveform samples		

Using the pragmas:

```
#pragma HLS INTERFACE axis port=strm_out
#pragma HLS INTERFACE axis port=strm_in_v
#pragma HLS INTERFACE axis port=cfg_in_v
#pragma HLS INTERFACE ap_ctrl_none port=return
```

generates the following hardware interface:

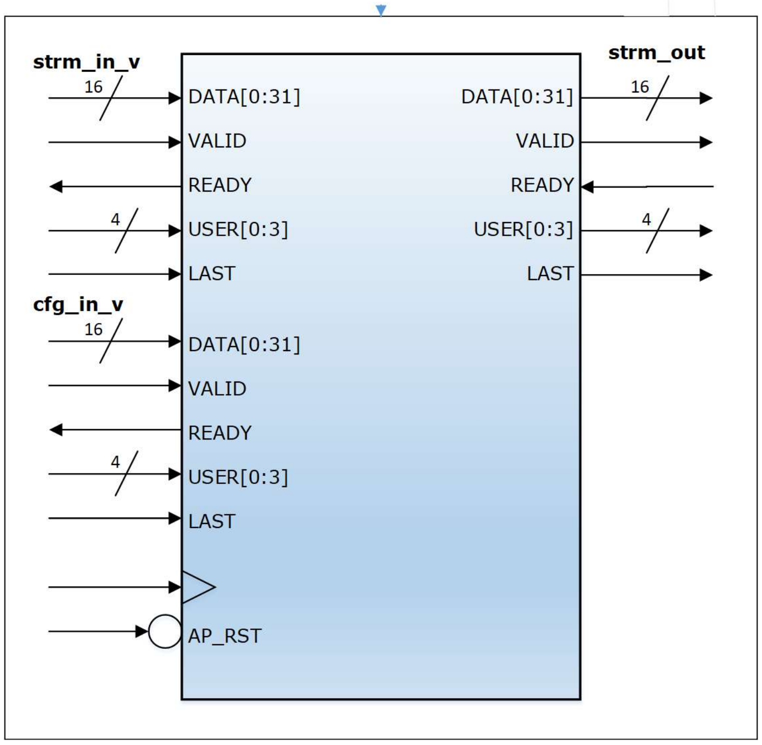


Figure 56 Waveformer ports

The following code fragment generates the RTL:

```
AXI_T cfg_val;        // element of the config vector
bool ping;                   // whether to store the cfg_val in the ping buffer
bool configured;      // true when the entire cfg_vector has been stored
unsigned short smpl_cnt; // state of the machine controlling Mult/Accum

// does the cfg_in_v FIFO have data?
if(!cfg_in_v.empty()){
 cfg_in_v >> cfg_val;
 // cfg_val.user gives the index to store at
 if(ping){
 pong_buf [cfg_val.user.to_int()] = cfg_val.data.to_int();
 } else {
 ping_buf [cfg_val.user.to_int()] = cfg_val.data.to_int();
 }
 // is this the last element of the cfg_in_v vector?
 if(cfg_val.last.to_int()){
 configured = true;
 }
} else {
 configured = false;
}
AXI_T axis_val;
// always input a value from strm_in_v
strm_in_v >> axis_val;

// synch
if(axis_val.last == 1){
 smpl_cnt = NUM_CHANS - 1;
}
// if zeroeth sample, reset the accumulator
if (smpl_cnt == 0){
 if(ping){
 acc = axis_val.data.to_int() * ping_buf[smpl_cnt];
 } else {
 acc = axis_val.data.to_int() * pong_buf[smpl_cnt];
 }
 smpl_cnt++;
} else if(smpl_cnt == NUM_CHANS - 1){ // last element of strm_in_v?
 if(ping){
 acc = acc + axis_val.data.to_int() * ping_buf[smpl_cnt];
 } else {
 acc = acc + axis_val.data.to_int() * pong_buf[smpl_cnt];
 }
 axis_val.data = (ap_int<16>) (acc >> (16));
 axis_val.last = 1;
 smpl_cnt = 0;
 // output a sample.
 strm_out << axis_val;
} else {
 // multiply/accumulate
 if(ping){
 acc = acc + axis_val.data.to_int() * ping_buf[smpl_cnt];
 } else {
 acc = acc + axis_val.data.to_int() * pong_buf[smpl_cnt];
 }
 smpl_cnt++;
}
// if configured, swap the ping-pong buffer pointers
if(configured){
 ping = !ping;
 configured = false;
}
```

Thus, using a ping-pong buffer and a mutex to ensure that block parameters are not corrupted by race conditions, the Waveformer inputs two streams of vectors, forms their dot product and outputs the resulting scalar stream of waveform samples at a rate, limited by backpressure, to 100 K samples/second. A typical output e.g. for a square wave output with 8 harmonics, is depicted in Figure 57 below

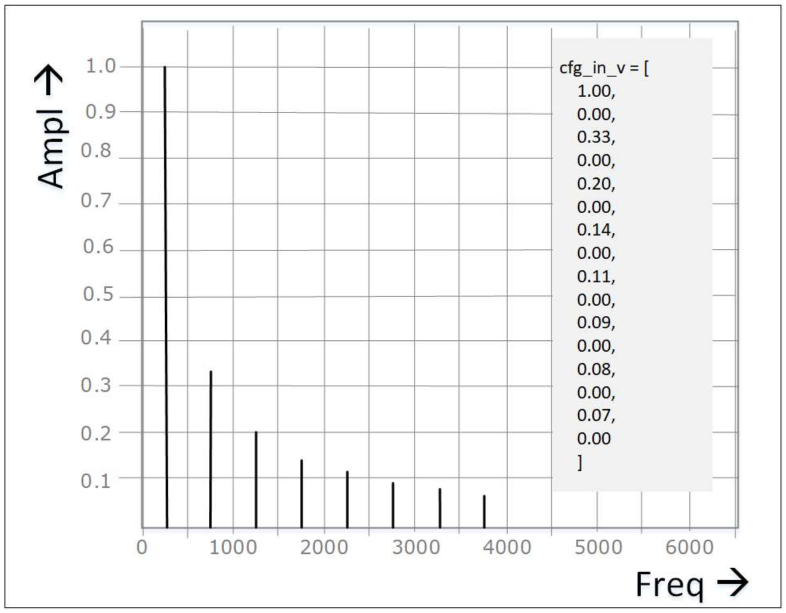


Figure 57 Example Waveformer O/P Spectrum after the application of a given configuration vector.

3.7.2 Analogue Output

This comprises four blocks:

- Antialiasing Filter
- DAC
- SPI_Tx_AXIS
- DAC Driver/Timer

3.7.2.1 Antialiasing Filter

The DAC analogue output is filtered by conventional a 4-pole analogue reconstruction filter with a bandwidth of 5 KHz, before being output by a buffer amplifier whose schematic is shown in Figure 105 Antialiasing filter. Figure 58 - Antialiasing Filter – SPICE plot shows the output of a SPICE[35] simulation of the filter.

[35] SPICE is a general-purpose, open-source analogue electronic circuit simulator

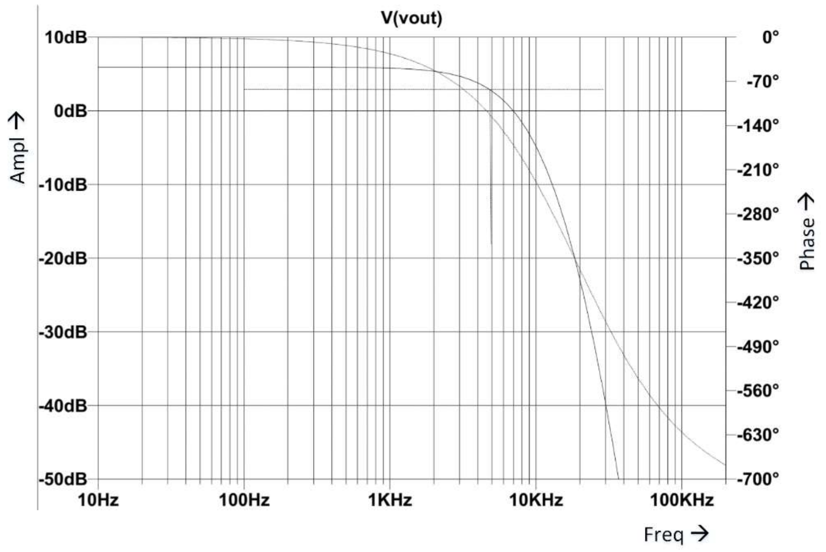


Figure 58 - Antialiasing Filter – SPICE plot

3.7.2.2 DAC

The DAC function uses the 4 Channel 16 bit 100ks/s DAC DAC8564, from Digilent's Analogue Shield Card[63], which uses a SPI interface. The board provides buffering for the DAC inputs, and a power supply for a plug-in board; in this case the anti-aliasing filter.

3.7.2.3 SPI_Tx_AXIS

The SPI_Tx_AXIS inputs 24-bit words embedded in a 32-bit AXI Stream and outputs SPI data to each DAC. It has the following hardware interface:

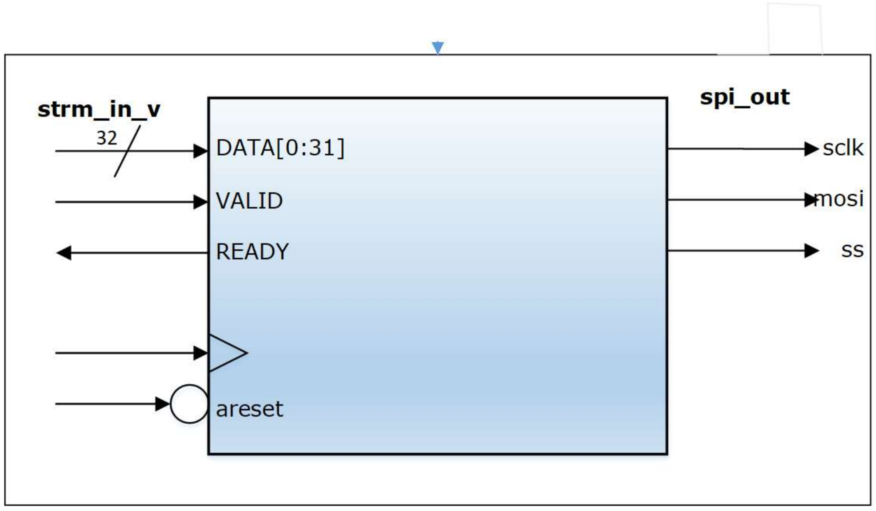


Figure 59 - SPI_Tx_AXIS

This Verilog-coded design is from Harald Rosenfeldt[64]

3.7.2.4 DAC Driver/Timer

The DAC Driver and Timer functions are integrated into a single processing block, the dac8564_drv:

- initialises dac8564 via SPI_Tx_AXIS_v1.1
- times and formats I/P data for onward transmission
- throttles streaming data rate to 100 k_samples/sec, asserting backpressure to upstream components.

This block has the C prototype:

```
typedef int16_t DDS_T;
typedef uint32_t AXI_T;
typedef hls::stream<DDS_T> IP_STREAM_T;
typedef hls::stream<AXI_T> OP_STREAM_T;

void dac8564_drv(
       OP_STREAM_T &op_stream,
       IP_STREAM_T &ip_stream,
       uint16_t term_count,
);
```

ip_stream	stream of 16-bit scalar waveform values
term_count	terminal count. Determines transmission rate.
op_stream	Output stream of 32-bit values containing a 24-bit control/data word for the DAC8564

Using the pragmas:

```
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl
#pragma HLS INTERFACE axis port=op_stream
#pragma HLS INTERFACE axis port=ip_stream
#pragma HLS INTERFACE ap_ctrl_hs port=return
```

Generates the following hardware interface:

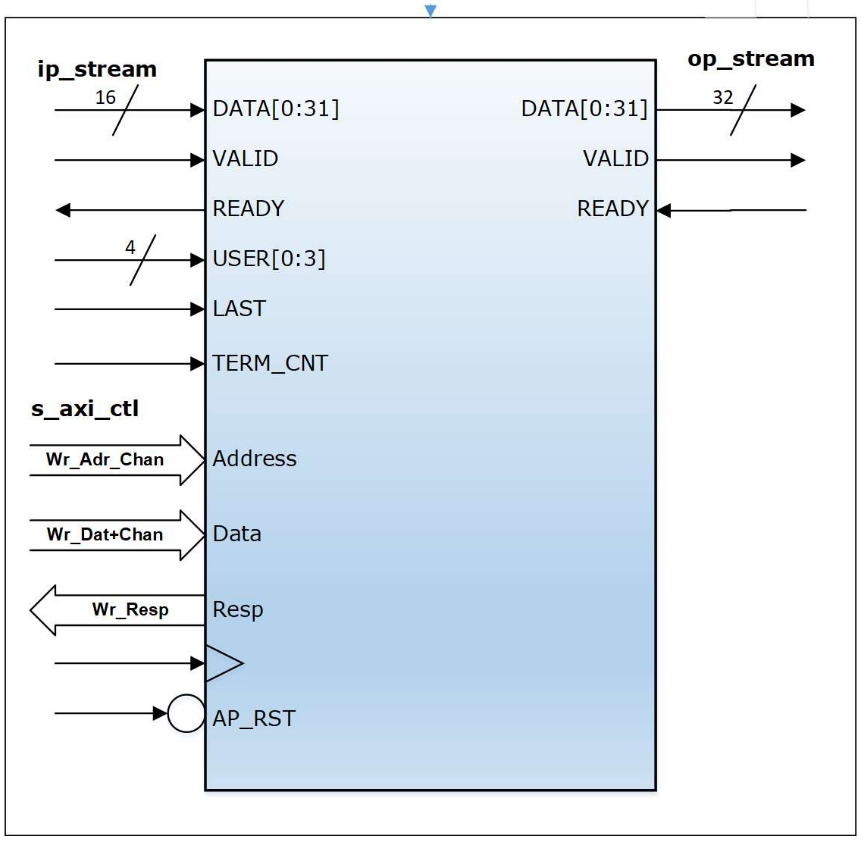


Figure 60 DAC Driver Ports

The following code fragment generates the RTL:

```
static uint16_t counter = 0; // count down system clock to 100 KHz

switch (state)
{
case 0: // DAC8564 RefON
 // set the DAC’s internal reference default mode
 ip_stream >> ip;
 op = ( REF_UP_AWS << REF_OFF);
 state++;
 op_stream << op;
 break;
case 1: // DAC8564 Power-up, normal operation
 // set the DACs for normal operation
 ip_stream >> ip;
 op = (AD0 << AD_OFF) |
 (SNG_CHN_UPD << LD_OFF) |
 (AZ << AZ_OFF) |
 ((CHN_A & 0x03) << CS_OFF) |
 (PWR_UP << PD_OFF) |
 (0x8000);
 op_stream << op;
 state++;
 break;
default:
 // count down system clock to 100 KHz
 if(counter < term_count - 1){
 counter++;
 } else {
 // We input data from ip_stream once per term_count clock cycles,
 // the compiler-generated interface will de-assert READY
 // for the other term_count - 1 clock cycles
 ip_stream >> ip;
 // 2’s complement offset
 ip += 0x8000;
 op = (AD0 << AD_OFF) |
 (SNG_CHN_UPD << LD_OFF) |
 (AZ << AZ_OFF) |
 ((CHN_A & 0x03) << CS_OFF) |
 (PWR_UP << PD_OFF) |
 (ip & 0xffff);
 op_stream << op;
 counter = 0;
 }
break;
}
```

The function outputs two control words, one to set up the DAC’s internal reference and the other to set the DAC to normal operation, before entering an infinite loop which inputs a 16-bit waveform values and formats them for output to the DAC. The 24-bit control/data words for the DAC8564 are specified in [16].

3.7.3 Waveformer Coefficient Generation

This set of functions, which defined the envelope of the waveform being played, comprises the following blocks[36]:

- Coefficient Amplitude Modulator
- AXIS Subset Converter
- AXI Stream FIFO
- ADSR

3.7.3.1 Coefficient Amplitude Modulator

The Coefficient Amplitude Modulator block inputs vectors of 16 Waveformer coefficients from the CPU and a scalar amplitude multiplier from the ADSR. The block multiplies each of the Waveformer coefficients by the scalar. The block outputs coefficient vectors at a rate of 100 K vectors per second (rate is capped by backpressure).

The ratio of Waveformer coefficients[1:15) to Waveformer coefficient[0] determines the spectrum of the waveform being generated. The value of Waveformer coefficient[0] (and hence the other coefficients) determines its amplitude.

This block has the C prototype:

```
typedef ap_axis <16,4,1,1> AXI_T;
typedef hls::stream<AXI_T> STREAM_T;
typedef ap_axiu <16,1,1,1> AXI_MOD_T;
typedef hls::stream<AXI_MOD_T> MOD_STREAM_T;

void ampl_mod( STREAM_T &strm_out_v,
               STREAM_T &strm_in_v,
               MOD_STREAM_T &mod_in)
```

strm_in_v	16-bit AXI Stream of Fourier coefficient vectors		
	side-channel data:	last:	indicates the last element of a vector.
		user:	index of the vector's element
mod_in	16-bit AXI Stream of amplitude multiplier scalars		
strm_out_v	16-bit AXI Stream of Fourier coefficient vectors		
	side-channel data:	last:	indicates the last element of a vector.
		user:	index of the vector's element

Using the pragmas:

[36] See the section "Waveformer Coefficients" in D - Appendix - FPGA-based Functional Block Diagram

```
#pragma HLS INTERFACE axis port=strm_out_v
#pragma HLS INTERFACE axis port=strm_in_v
#pragma HLS INTERFACE axis port=mod_in
#pragma HLS INTERFACE ap_ctrl_none port=return
```

Generates the following hardware interface:

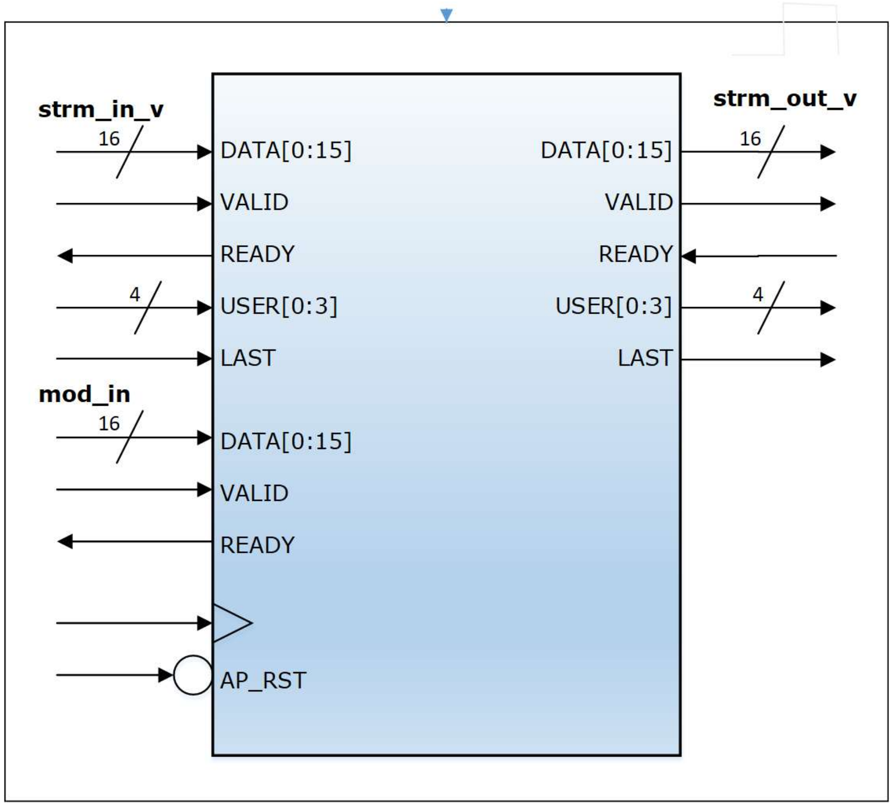


Figure 61 Ampl Mod ports

The following code fragment generates the RTL:

```
unsigned short sample_count; // state of the machine
bool ping; // whether to store the strm_in vector in the ping buffer
bool configured; // true when the entire strm_in vector has been stored

// does the strm_in_v FIFO have data?
if(!strm_in_v.empty()){
 strm_in_v >> axis_in;
 if(ping){
 pong_buf [axis_in.user.to_int()] = axis_in.data.to_int();
 } else {
 ping_buf [axis_in.user.to_int()] = axis_in.data.to_int();
 }
 // is this the last element of the axis_in vector?
 if(axis_in.last.to_int()){
 configured = true;
 }
} else {
 configured = false;
}

// We need one mod_in scalar per strm_in_v vector
if(smpl_cnt == 0){
 mod_in >> mod_val;
}
if(ping){
 op_buf = mod_val.data.to_int() * ping_buf[smpl_cnt];
} else {
 op_buf = mod_val.data.to_int() * pong_buf[smpl_cnt];
}
// assert last if this is the last element of the strm_out vector
if(smpl_cnt == 15){
 axis_out.last = 1;
} else {
 axis_out.last = 0;
}
axis_out.data = (ap_int<16>) (op_buf >> (16));
// write an index
axis_out.user = (ap_int<4>) smpl_cnt;
// output a sample.
strm_out_v << axis_out;

if (smpl_cnt < NUM_CHANS - 1){
 smpl_cnt++;
} else {
 smpl_cnt = 0;
}
// if configured, swap the ping-pong buffer pointers
if(configured){
 ping = !ping;
 configured = false;
}
```

Thus, using a ping-pong buffer and a mutex to ensure that block parameters are not corrupted by race conditions, each element of the input coefficient vector is multiplied by the amplitude multiplier and normalised before outputting 16-bit coefficient vectors.

3.7.3.2 AXI Stream FIFO

The AXI4-Stream FIFO[19] converts a memory mapped AXI4-lite interface to an AXI4-Stream interface, and includes a FIFO with store-and forward capability. This allows a set of coefficients to be stored in the FIFO via the AXI-lite interface. The coefficients are then forwarded as an indivisible vector via the AXI4-Stream interface. a single direction of data transmission is used in this design.

The coefficient set is received from the CPU and forwarded on to the AXIS Subset Converter.

3.7.3.3 AXIS Subset Converter

The Xilinx AXI4-Stream Subset Converter[20] provides a solution for connecting slightly incompatible AXI4-Stream signal sets together. The IP has configurable AXI4-Stream signals for each interface that allows one to convert one signal set to another in consistent manner.

It's used here to convert the 32-bit stream from the AXI Stream FIFO to the 16-bit stream required by the ADSR

3.7.3.4 ADSR

The ADSR block inputs the following inputs from the CPU: NoteOnOff, Attack, Decay, Sustain, Release, and Volume. It defines the envelope of the waveform being played, and also its volume.

This block has the C prototype:

```
typedef ap_axiu <16,1,1,1> AXI_T;
typedef hls::stream<AXI_T> STREAM_T;

void adsr(     STREAM_T &ctrl_out,
               ap_uint<18> atc_tim,
               ap_uint<16> atc_inv,
               ap_uint<16> gammaD,
               ap_uint<16> gammaR,
               ap_uint<16> sus_lev,
               ap_uint<16> vol,
               bool note);
```

atc_tim	18-bit AXI-lite attack time in msec
atc_inv	16-bit AXI-lite increment required to teach attack target in attack time, normalised to 16384
gammaD	16-bit AXI-lite. (1 - Decay-rate parameter)
gammaR	16-bit AXI-lite. (1 - Release-rate parameter)
sus_lev	16-bit AXI-lite. Sustain level, normalised to 16384
vol	16-bit AXI-lite. volume, normalised to 16384
note	bool AXI-lite. Note on/off
ctrl_out	16-bit AXI Stream. Scalar amplitude multiplier

Using the pragmas:

```
#pragma HLS INTERFACE axis port=ctrl_out
#pragma HLS INTERFACE s_axilite port=atc_tim bundle=ctrl
#pragma HLS INTERFACE s_axilite port=atc_inv bundle=ctrl
#pragma HLS INTERFACE s_axilite port=gammaD bundle=ctrl
#pragma HLS INTERFACE s_axilite port=gammaR bundle=ctrl
#pragma HLS INTERFACE s_axilite port=sus_lev bundle=ctrl
#pragma HLS INTERFACE s_axilite port=vol bundle=ctrl
#pragma HLS INTERFACE s_axilite port=note bundle=ctrl
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl
```

Generates the following hardware interface:

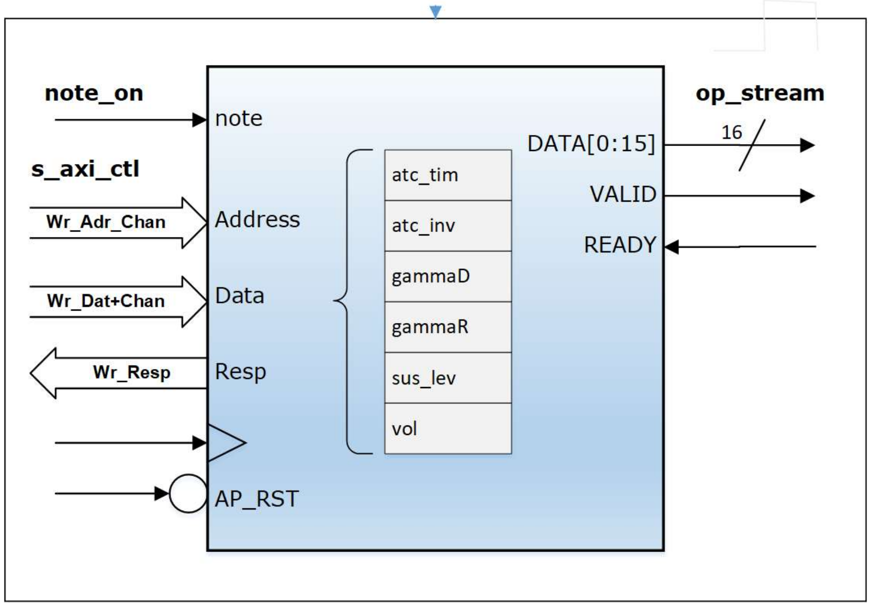


Figure 62 - ADSR

The following code fragment (adapted from De Leon[21]) generates the RTL:

```
// Note on/off: leading edges
note_on = note && !old_note;
note_off = !note && old_note;
old_note = note;

*note_out = note;

// Attack phase
if(note_on){
 val = 0;
 n = 0;
 release = false;
}
if(note_off){
 release = true;
}

if(!release){
 if(n < atc_tim){
 val = (n * atc_inv);
 } else {
 // Sustain phase
 val = ((sus_lev * gammaD) >> 16)
 + (((MAX_VAL - gammaD) * old_val) >> 16);
 }
} else {
// Release phase
 val = ((MAX_VAL - gammaR) * old_val) >> 16;
}
if(val.to_int() > MAX_VAL){
 val = MAX_VAL;
}
old_val = val;
n++;

// Volume
axis_val.data = (val*vol) >> 16;
axis_val.last = 0;

ctrl_out << axis_val;
```

The block responds to the NoteOnOff signal, producing an ADSR envelope whose characteristics are defined by the other input parameters, and outputs a scalar amplitude multiplier for the amplitude modulator.

3.7.4 Frequency Vector Generation

This comprises three blocks:

- Frequency to Phase Converter
- Phase Vector Generation
- AXI4 Stream FIFO

3.7.4.1 Frequency to Phase Converter

The Frequency to Phase block inputs 24-bit frequency samples. It inverts frequency to phase increment by multiplying by a constant, where:

```
*        PINC = F_IN * 2**24/100E3
*         = F_IN * 167.77216
*        INT(167.77216 * 1024) = 171799
```

This block has the C prototype:

```
typedef ap_axis <32,1,1,1> AXI_FREQ_T;
typedef hls::stream<AXI_FREQ_T> FREQ_STREAM_T;

typedef ap_axis <32,1,1,1> AXI_PHASE_T;
typedef hls::stream<AXI_PHASE_T> PHASE_STREAM_T;

void freq_to_phase(
       PHASE_STREAM_T &PHASE_INC,
       FREQ_STREAM_T &FREQ);
```

FREQ	32-bit AXI Stream of frequency samples
PHASE_INC	24-bit AXI Stream of phase increments

Using the pragmas:

```
#pragma HLS INTERFACE axis port=PHASE_INC
#pragma HLS INTERFACE axis port=FREQ
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl
```

Generates the following hardware interface:

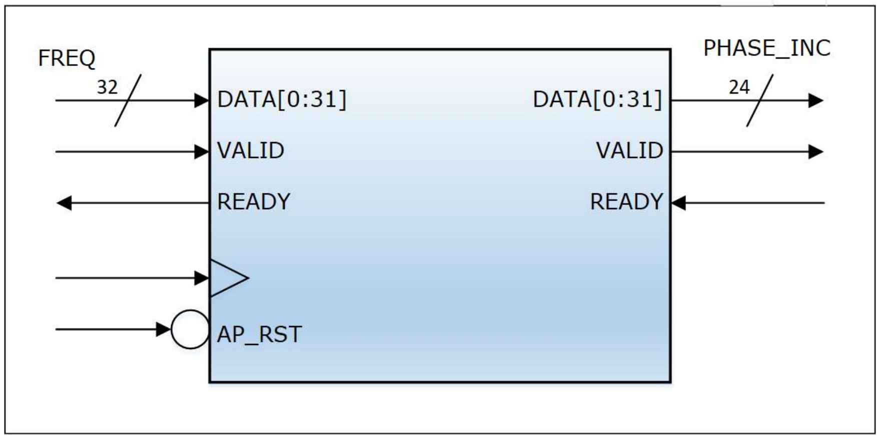


Figure 63 Freq to Phase ports

The following code fragment generates the RTL:

```
// does the strm_in_v FIFO have data?
if(!FREQ.empty()){
       FREQ >> axis_freq;
}

tmp = axis_freq.data;
tmp = tmp * 171799UL;
tmp = tmp >> 20;

axis_phase_inc.data = tmp.to_int();
axis_phase_inc.keep = 0xf;
axis_phase_inc.strb = 0xf;
axis_phase_inc.id = 0;
axis_phase_inc.dest = 0;

PHASE_INC << axis_phase_inc;
```

The function simply multiplies the inputs by a constant and then scales the values before outputting 24-bit phase increments to the Phase Vector Generator block.

3.7.4.2 Phase Vector Generation

The Phase Vector Generator block inputs a series of phase increment scalers and outputs a stream of vectors containing the instantaneous phases of a set of harmonically related sin waves to the DDS. For each phase increment input, representing a frequency *f*, the function outputs a vector of 16 elements representing the frequencies: *f*, 2**f*, 3**f*, etc.

This block has the C prototype:

```
typedef ap_axis <32,1,1,1> AXI_PHASE_IN_T;
typedef hls::stream<AXI_PHASE_IN_T> PHASE_IN_STREAM_T;
typedef ap_axis <32,4,1,1> AXI_PHASE_OUT_T;
typedef hls::stream<AXI_PHASE_OUT_T>       PHASE_OUT_STREAM_T;

void phase_vector_gen(
       PHASE_OUT_STREAM_T &PHASE_VEC,
       PHASE_IN_STREAM_T &phase_in,
       bool note_on);
```

phase_in	32-bit AXI-lite I/F phase scaler. (phase of the fundamental)
note_on	16-bit AXI_lite I/F Note on signal

PHASE_VEC	32-bit AXI Stream of phase vectors[37] Structure: bit 24 RESYNC bit [23:0] PINC

Using the pragmas:

```
#pragma HLS INTERFACE s_axilite port=note_on bundle=ctrl
#pragma HLS INTERFACE axis port=phase_in
#pragma HLS INTERFACE axis port=PHASE_VEC
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl
```

Generates the following hardware interface:

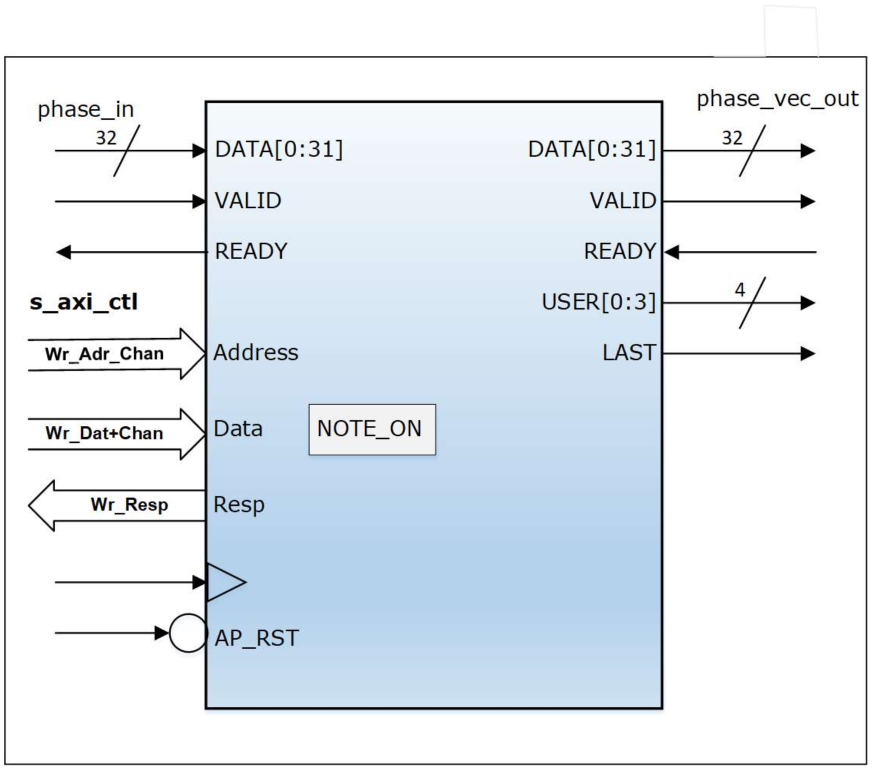


Figure 64 Phase Vector Gen Ports

The following code fragment generates the RTL:

[37] RESYNC, input to the DDS, resets the accumulated phase of the channel in question

```
phase_in >> phase;
phase_inc = phase.data & PHASE_MSK;

// detect leading edge of note_on
resync = note_on && !old_note_on;
old_note_on = note_on;

for(int i = 0; i < NUM_CHANS; i++){
 axis_val.data += phase_inc;
 if(resync){
 data = axis_val.data.to_int();
 data |= RESYNC_BIT;
 axis_val.data = data;
 }
 // output an index
 axis_val.user = i;
 // is this the last element of the output vector?
 if(i == NUM_CHANS-1){
 axis_val.last = 1;
 resync = false;
 }else{
 axis_val.last = 0;
 }
 // output an element of the phase vector
 PHASE_VEC << axis_val;
}
```

Thus, for each input phase increment corresponding to a fundamental frequency f, the function outputs a vector of phase increments corresponding to harmonic series of sin waves at f, 2f, 3f, ...

3.7.4.3 AXI4 Stream FIFO

A Xilinx-provided AXI4-Stream Data FIFO is placed between the Phase Vector Generator and the DDS.

3.7.5 Vibrato and Glissando

These functions provide the Vibrato and Glissando functions, and comprise the following blocks:

- MIDI Note to Frequency Converter
- AXIS Scaler Amplitude Modulator
- DDS Compiler/LFO
    - AXI-Stream FIFO I
    - AXI Stream Subset Converter
- Glide
    - AXI-Stream FIFO II

Because the MIDI Note to Frequency Converter outputs data from the frequency domain and inputs data from the MIDI Notes domain, the Vibrato and Glissando functions can operate in the latter human-friendly domain.

3.7.5.1 MIDI Note to Frequency Converter

The MIDI Note to Freq Block inputs a MIDI Note parameter from the Glide Block and a modulation parameter from the Scalar Amplitude Modulator. The MIDI Note to Freq Block has the C prototype:

```
typedef ap_axis <32,1,1,1> AXI_T;
typedef hls::stream<AXI_T> STREAM_T;

typedef ap_axiu <16,1,1,1> AXI_MOD_T;
typedef hls::stream<AXI_MOD_T> STREAM_MOD_T;

void mod_note_to_freq(
 STREAM_T &FREQ,
 STREAM_T &NOTE_IN,
 STREAM_MOD_T &MOD_IN);
```

NOTE_IN	32-bit AXI-lite I/F: bit [17:0] MIDI NOTE * 1024
MOD_IN	16-bit AXI Stream of modulation values
FREQ	32-bit AXIS I/F: bit [23:0] Stream of frequency values

Using the pragmas:

```
#pragma HLS INTERFACE axis port=FREQ
#pragma HLS INTERFACE axis port=NOTE_IN
#pragma HLS INTERFACE axis port=MOD_IN
#pragma HLS INTERFACE ap_ctrl_none port=return bundle=ctrl
```

Generates the following hardware interface:

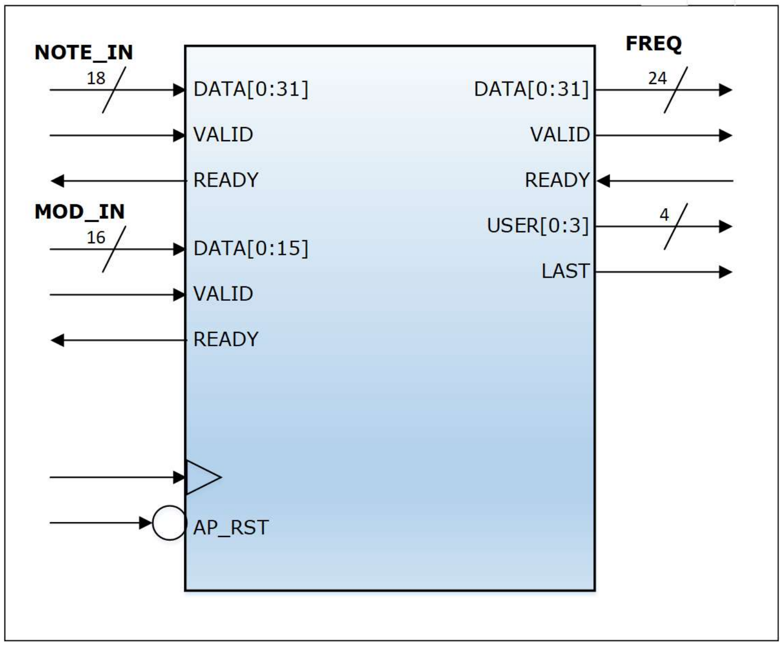


Figure 65 Note to Freq Ports

The following code fragment generates the RTL:

```
// always input a note value
NOTE_IN >> note_in;
// is there data in the MOD_IN FIFO?
if(!MOD_IN.empty()){
 MOD_IN >> mod_in;
}

int tmp;
uint32_t      note;
// modulate the note value
tmp = note_in.data + mod_in.data - 0x8000;
// clip the note value
if(tmp < 0){ tmp = 0; }
if(tmp > 130048){ tmp = 130048; }    // 130048 = 127 * 1024
note = (uint32_t) tmp;
// lookup the corresponding frequency
// freq = 440.0 * pow( 2.0, ((double)data - 69.0) / 12.0);
float freq = interp(note);
// output a frequency value
FREQ << freq_out;
```

The frequency modulation parameter is converted to a signed number before being added to the note value. Then a note-to-frequency operation is performed before outputting modulated 24-bit frequency samples to the Frequency to Phase block.

The interp(note**)** function in the code is implemented as a linearly-interpolated 128-entry table.

3.7.5.2 AXIS Scaler Amplitude Modulator

The Scaler Amplitude Modulator Block inputs a stream of sin values form the DDS/LFO and a stream of multiplier coefficients from the CPU. It multiplies them and outputs the result.

It has the C prototype:

```
typedef ap_axis <16,1,1,1> AXI_T;
typedef hls::stream<AXI_T> STREAM_T;
typedef ap_axiu <16,1,1,1> AXI_U_T;
typedef hls::stream<AXI_U_T> STREAM_U_T;
typedef ap_axiu <16,1,1,1> AXI_MOD_T;
typedef hls::stream<AXI_MOD_T> MOD_STREAM_T;

void scl_ampl_mod(STREAM_U_T &strm_out,
                       STREAM_T      &strm_in,
                       int32_t mod_in);
```

strm_in,	16-bit AXI Stream of sin values
mod_in,	32-bit AXI-lite I/F: 16-bit modulation coefficients
strm_out	16-bit AXI Stream of sin values

Using the pragmas:

```
#pragma HLS INTERFACE s_axilite port=mod_in bundle=ctrl
#pragma HLS INTERFACE axis port=strm_out
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl
```

Generates the following hardware interface:

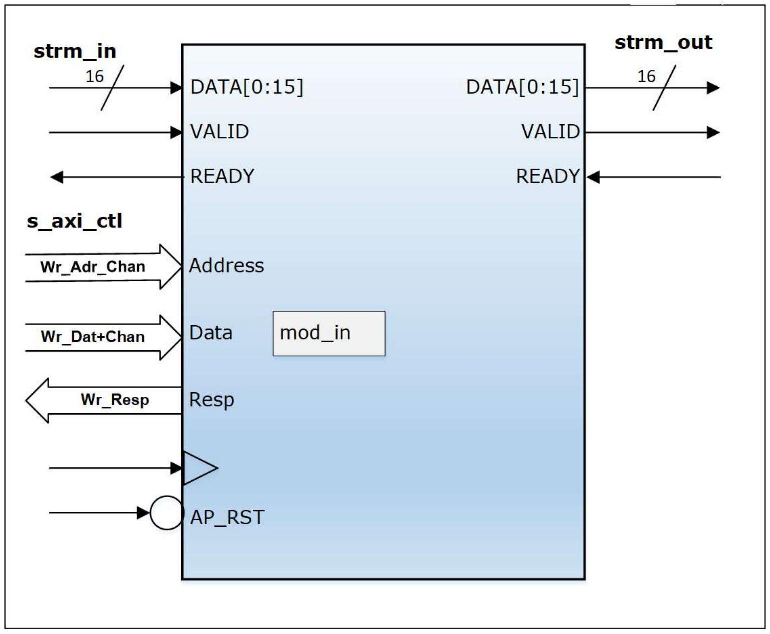


Figure 66 Scalar Ampl Mod Ports

The following code fragment generates the RTL:

```
mod_val = mod_in;

strm_in >> axis_in;
axis_in_data = axis_in.data.to_int();
axis_out_data = mod_val * axis_in_data;
axis_out.data = (uint16_t) ((axis_out_data >> (16)) + 0x8000U);
strm_out << axis_out;
```

The function outputs the modulated sine wave to the frequency modulation input of the MIDI Note to Freq Block.

3.7.5.3 DDS /LFO

The Low Frequency Oscillator (LFO) block inputs scalar phase increments from the CPU via an AXI-Stream FIFO and AXI Stream Subset Converter and outputs a sin wave to the scalar amplitude modulator block. It is implemented using a Xilinx DDS .

The Xilinx DDS is a Numerically Controlled Oscillator (NCO) which is configured to input 16-bit phase increments, at a rate of 200 vectors per second, and output 16-bit sin waves, sampled at 100 K samples per second.

· AXI-Stream FIFO I

AXI4-Stream FIFO[19] converts a memory mapped AXI4-lite interface from the CPU to an AXI4-Stream interface

· AXI Stream Subset Converter

The Xilinx AXI4-Stream Subset Converter[20] converts the 32-bit stream from the AXI Stream FIFO to the 16-bit stream required by the DDS/LFO

3.7.5.4 Glide

The Glide Block inputs an enable signal, an 18-bit slew rate parameter in increments per 10 micro-seconds and an 18-bit MIDI Note parameter. It has the following C prototype:

```
typedef ap_axis <32,1,1,1> AXI_T;
typedef hls::stream<AXI_T> STREAM_T;
void glide( STREAM_T &NOTE_OUT,
 STREAM_T &NOTE_IN,
 int32_t slew );
```

NOTE_IN	32-bit AXI Stream: 18-bit MIDI Note * 1024		
slew	32-bit AXI Lite structure.		
		18-bit	slew_rate: increments per 10 µsec
		bool	slew_en
NOTE_OUT	32-bit AXI Stream: 18-bit MIDI Note * 1024		

Using the pragmas:

```
#pragma HLS INTERFACE s_axilite port=slew bundle=ctrl
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl
#pragma HLS INTERFACE axis port=NOTE_OUT
#pragma HLS INTERFACE axis port=NOTE_IN
```

Generates the following hardware interface:

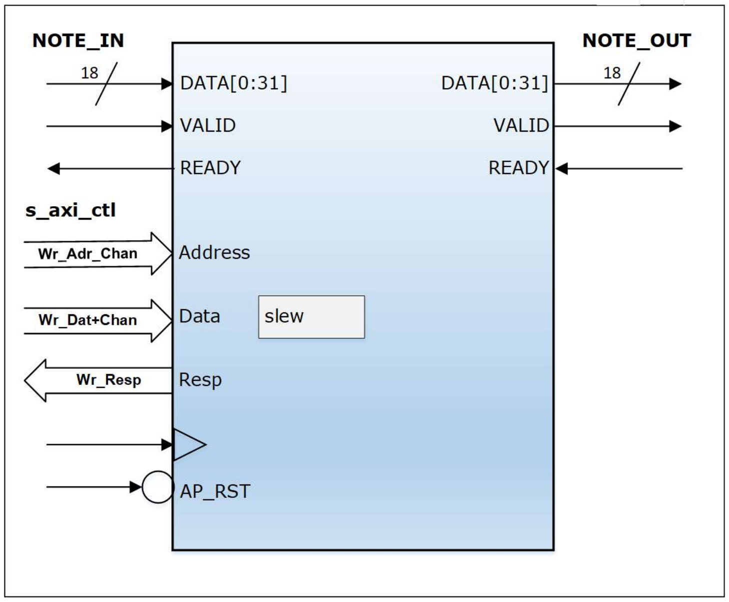


Figure 67 Glide Ports

The following code fragment generates the RTL:

```
NOTE_IN >> note_in;
slew_rate = slew & SLEW_MSK;
int32_t tmp = slew & ENA_MSK;
if(tmp != 0){ slew_en = true; } else { slew_en = false; }

if(slew_en){
 if(note_in.data > note){
 note += slew_rate;
 if(note > note_in.data){
 note = note_in.data;
 }
 } else {
 if(note_in.data < note){
 note -= slew_rate;
 if(note < note_in.data){
 note = note_in.data;
 }
 }
 }
 note_out.data = note;
 NOTE_OUT << note_out;
} else {
 NOTE_OUT << note_in;
}
```

It slew-rate limits the transition between notes and outputs an 18-bit MIDI Note to the MIDI Note to Freq Block.

· AXI-Stream FIFO II

AXI4-Stream FIFO [19] converts a memory mapped AXI4-lite interface from the CPU to an AXI4-Stream interface for the Glide Block and provides a FIFO buffer.

3.8 Software

The PC Software sends JSON[38] messages to the Zynq CPU via a Serial_Over_USB interface in order to control the Synth. The Control Firmware on the Zynq CPU parses the JSON messages[39] and controls the hardware

3.8.1 Control firmware

The control firmware, running of the Zynq-700's ARM, inputs JSON data over a serial channel from the PC, and outputs configuration data to the functional blocks over AXI4-lite interfaces.

The firmware (see Figure 68 Firmware Functional Block Diagram) comprises a foreground section, which initialises and updates elements of a set of Global Data Structures (GDS), and an Interrupt Service Routine, which uses the contents of the GDS (Figure 69) to transmit sets of coefficients to the hardware.

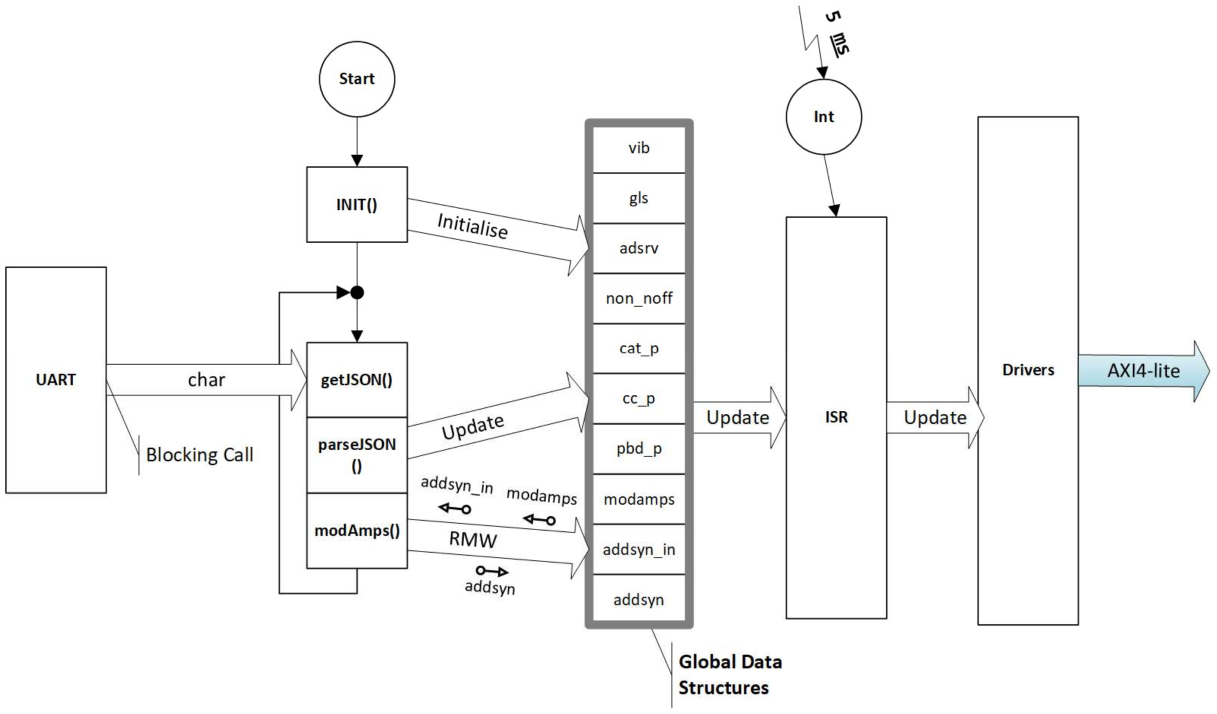


Figure 68 Firmware Functional Block Diagram

3.8.1.1 Foreground Section

The foreground section initialises the Xilinix-generated Hardware Drivers and the GDS before entering a loop which:

[38] JavaScript Object Notation (JSON) is an open standard data interchange format
[39] A message encoded in JSON

- gets JSON Messages from the UART,
- parses the JSON Messages and updates the Global Data Structure and
- modulates the coefficients contained in the addsyn_in structure and copies them to the addsyn structure.

The function getJSON() makes blocking calls to the UART, waiting for text characters from the PC and delivers complete JSON messages to the parseJSON() function. This latter function parses JSON and updates the coefficients in the GDS as the messages are received from the PC. The modAmps() function modulates the amplitudes of the coefficients in the addsyn structure, in order to implement Additive Synthesis

The JSON messages are transmitted from the PC with a latency that's much shorter than human response times, so that the elements of the GDS are effectively updated in real time.

3.8.1.2 Interrupt Service Routine

The Interrupt Service Routine (ISR) is triggered at 200 Hz and transmits the contents of the various elements of the GDS over an AXI4-Lite interface to the hardware on the FPGA. Where these data are in vector form, the ISR manages the Store-and-Forward action of the associated FIFOs, in order to avoid transmitting incomplete vectors.

vib	Vibrato uint32_t depth uint32_t rate
gls	Glissando bool slew_en uint32_t slew_rate
adsrv	ADSR & Vol uint32_t atc_tim **Attack time**. uint32_t atc_inv **Increment to reach full scale**. uint32_t gammaD **Decay time**. uint32_t gammaR **Release time**. uint32_t sus_lev **Sustain level**. uint32_t vol **Volume level**.
addsyn	Additive Synthesis – modulated uint32_t h[16] **amplitude** uint32_t p[16] **phase**
addsyn_in	Additive Synthesis – source uint32_t h[16] **amplitude** uint32_t p[16] **phase**
modamps	Modulation Coeffs for Additive Synth uint32_t m[16] **multiplier**
non_nof	Note on/off bool sem **Semaphore** int non **number of keys pressed** int last_non **Note on** uint8_t chn **Chan** uint8_t num **Number** uint8_t vel **Velocity**
cat_p	Channel Aftertouch uint8_t chn **Chan** uint8_t prs **pressure**
cc_p	Control Channel uint8_t chn **Chan** uint8_t cnm **Controller ID (1 ➔ Mod Wheel)** uint8_t val **Value**
pbd_p	Pitch Bend uint8_t chn **Chan** int16_t val **value**

Figure 69 Global Data Structures

3.8.2 PC Software

The PC Software is a Python script comprising two threads:

- Main wxPytnon Thread presents a Graphical User Interface (GUI) to the user, presenting controls which on being manipulated, result in sending JSON messages to the Zynq-7000.
- MIDI_IP Thread which inputs and parses MIDI messages from a MIDI Keyboard and sends JSON messages to the Zynq-7000.

The MIDI Keyboard used is a velocity-sensitive device with aftertouch. The latter is patched to the vibrato control. The following MIDI Messages are handled:

- NoteOff
- NoteOn
- PolyphonicAftertouch
- ControlChange
- ProgramChange
- ChannelAftertouch
- PitchBend

Screen Grabs of the GUI presented to the user are shown in Appendix – PC Controller Software – Screen Grabs. They are not intended to be an example of good user interface design, but rather as a Test Harness.

3.8.2.1 ADSRV

The ADSRV (attack, decay, sustain, release, volume) Tab (Figure 106) presents sliders for Attack and Decay times, Sustain Level and Release rate, as well as overall Volume. It presents buttons for the selection of various useful presents.

3.8.2.2 ADDSYN

The ADDSYN Tab (Figure 107) Presents Sliders for 16 harmonics of the fundamental sine wave, and radioboxes to allow the selection of one of four phases for each harmonic. It presents buttons for the selection of various useful presets:

- Sine
- Square
- Trapezium
- Triangle
- Sawtooth
- Impulse
- All

The latter button ("All") Sets all the harmonics to the same level, which is useful for demonstrating the action of the Pseudo-Filter components mentioned below.

3.8.2.3 VIBRATO

The VIBRATO Tab (Figure 108) presents sliders for Depth and Rate.

3.8.2.4 GLISSANDO

The GLISSANDO Tab (Figure 109) presents a slider for Glissando rate, and an Enable/Disable switch.

3.8.2.5 FILTER

The FILTER Tab (Figure 110) Presents a slider for Frequency, and three buttons for flat, Peaked Low-Pass Filter (LPF) and LPF responses.

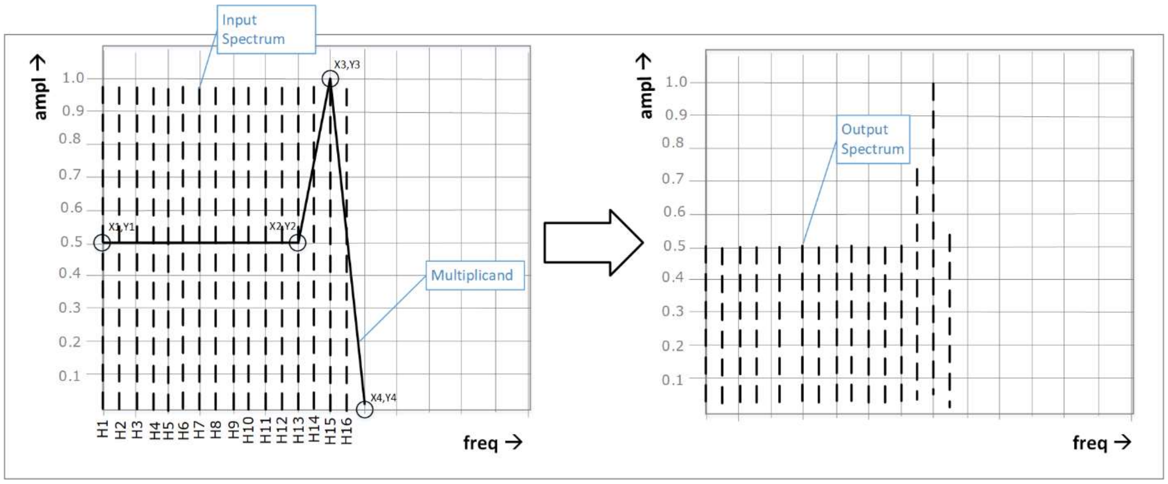


Figure 70 Filter Points

An interesting fact to note is that this is not, in fact, a filter, but a set of four points, with a graph defined by linearly interpolating between the points. A set of dashed lines, representing the 16 harmonics of the waveform being generated is superimposed on the graph, and the vertical ordinate of the intersection between each vertical line and the graph is used to define the multiplier coefficient for the associated harmonic. Now, all that is required to simulate a swept filter, is to sweep the X4 coordinate up and down, with the other X points being swept proportionally.

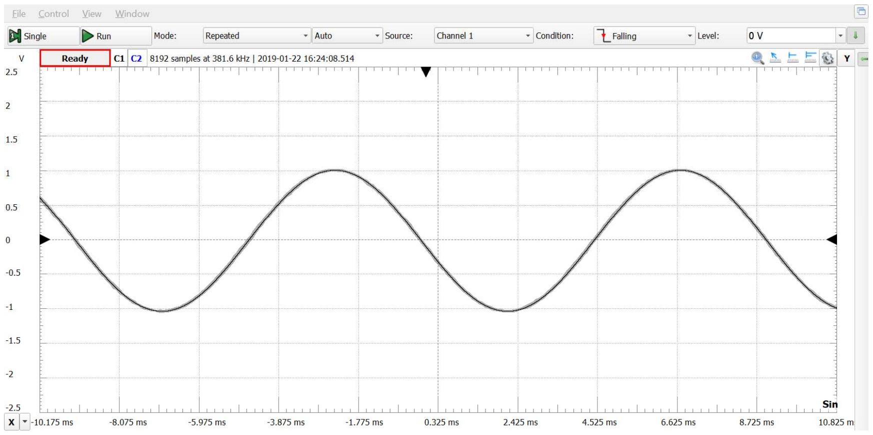
3.9 Results

3.9.1 Generated waveforms

A set of waveforms emulating those of the classic analogue synthesiser was generated. The following plots were taken from the output of the reconstruction filter:

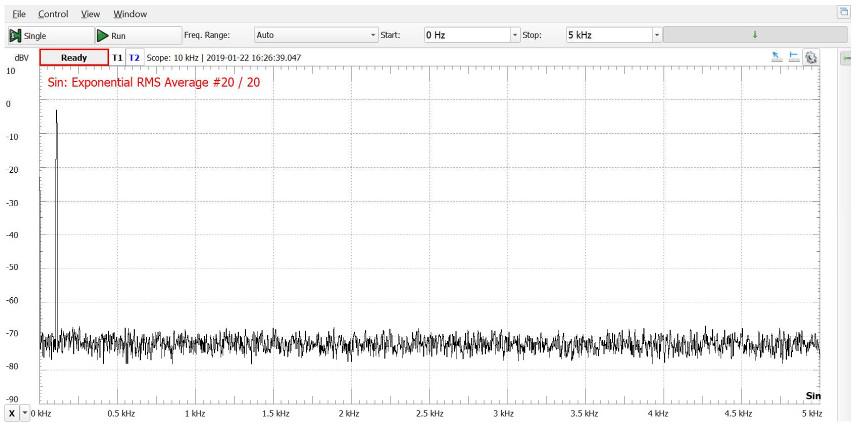
3.9.1.1 Sine

The sine wave generated is free from harmonics, and noise components are 100dB below the amplitude of the sinewave.


Single Run Mode: Repeated Auto Source: Channel 1 Condition: Falling Level: 0 V

Ready C1 C2 8192 samples at 381.6 kHz | 2019-01-22 16:24:08.514

Figure 71 Observed Sine – Amplitude


Single Run Freq. Range: Auto Start: 0 Hz Stop: 5 kHz

Ready T1 T2 Scope: 10 kHz | 2019-01-22 16:26:39.047

Sin: Exponential RMS Average #20 / 20

Figure 72 Observed Sine Spectrum

3.9.1.2 Square

The generated square wave shows Gibbs Phenomenon ringing, as would be expected from a band-limited square wave. Note the “sharp cut-off” at high frequencies: there are no harmonics higher than the 16th. The eight even harmonics have zero amplitude, hence only the eight odd harmonics are evident in the spectrum plot.

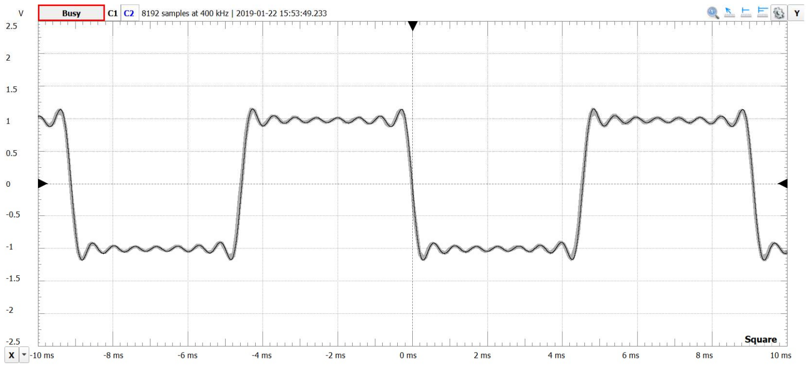


Figure 73 Observed Square Amplitude

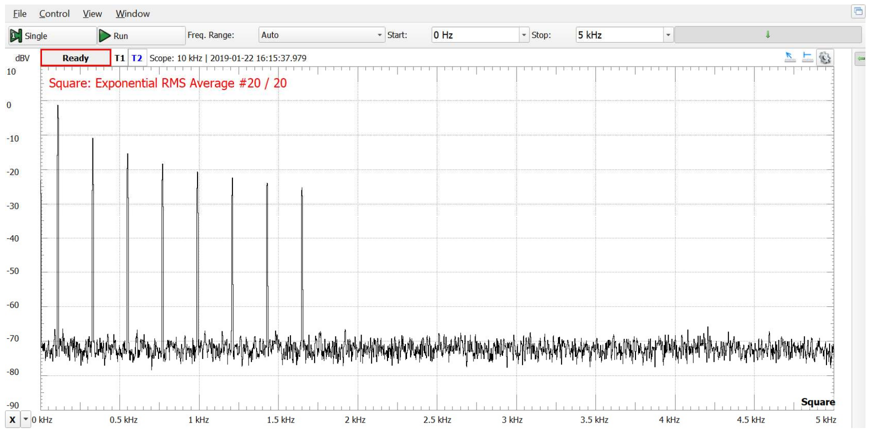


Figure 74 Observed Square Spectrum

3.9.1.3 Trapezium

The trapezium waveform generated shows no ringing.

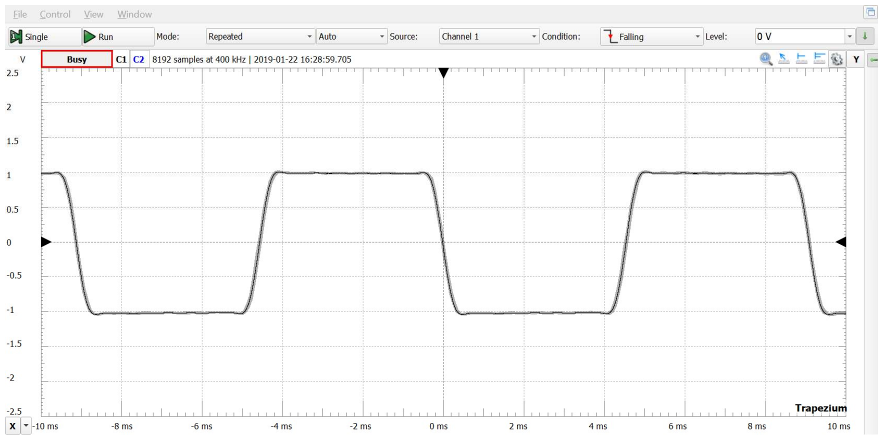


Figure 75 Observed Trapezium Amplitude

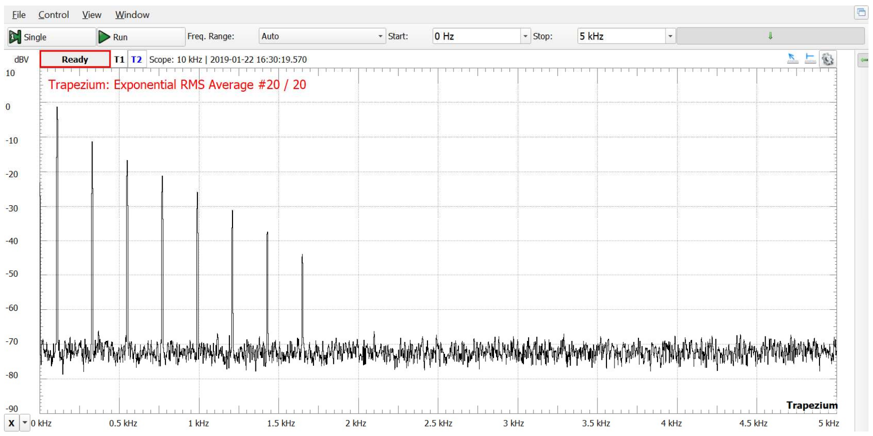


Figure 76 Observed Trapezium Spectrum

Sawtooth

The sawtooth waveform looks like a band-limited saw-tooth as expected. Note that there are 16 harmonics here, as both the odd and even harmonics have non-zero coefficients.

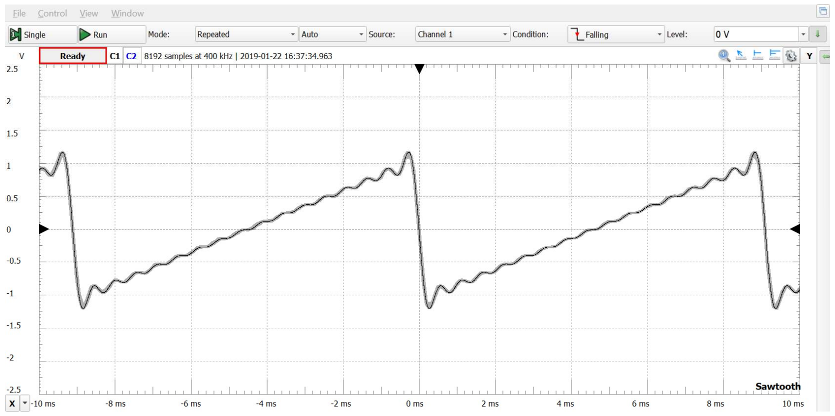


Figure 77 Observed Sawtooth Amplitude

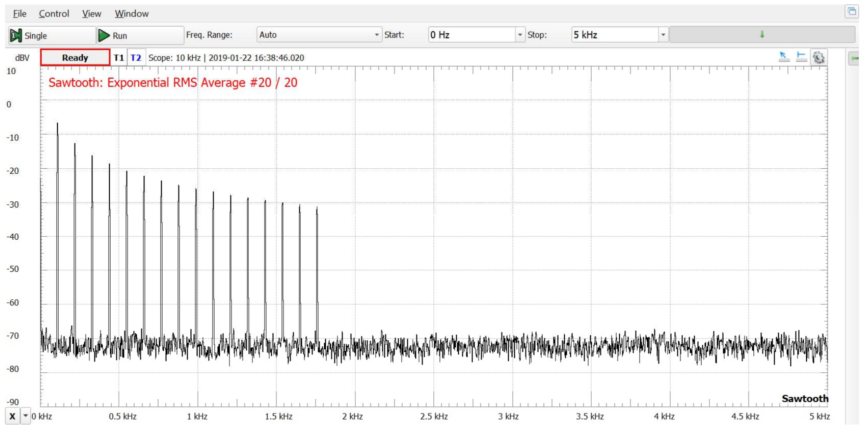
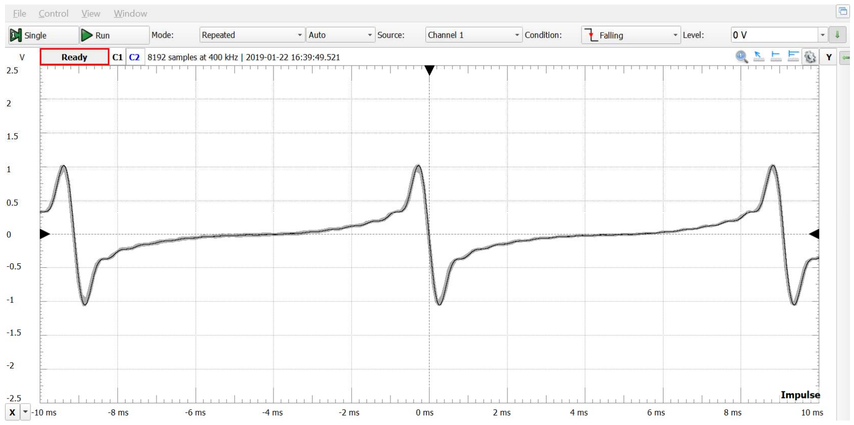


Figure 78 Observed Sawtooth Spectrum

3.9.1.4 Impulse

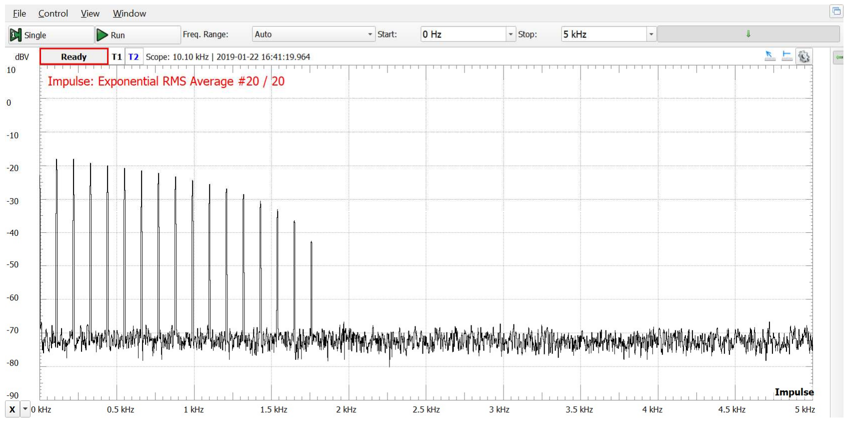
This waveform is as close to an impulse as we can generate.


File Control View Window

Single Run Mode: Repeated Auto Source: Channel 1 Condition: Falling Level: 0 V

Ready C1 C2 8192 samples at 400 kHz | 2019-01-22 16:39:49.521

Figure 79 Observed Impulse Amplitude


File Control View Window

Single Run Freq. Range: Auto Start: 0 Hz Stop: 5 kHz

Ready T1 T2 Scope: 10.10 kHz | 2019-01-22 16:41:19.964

Impulse: Exponential RMS Average #20 / 20

Figure 80 Observed Impulse Spectrum

3.9.1.5 All

This waveform was generated in order to test the filter simulation below. All amplitudes are the same, allowing one to discern the shape of the filter to which this waveform is applied.

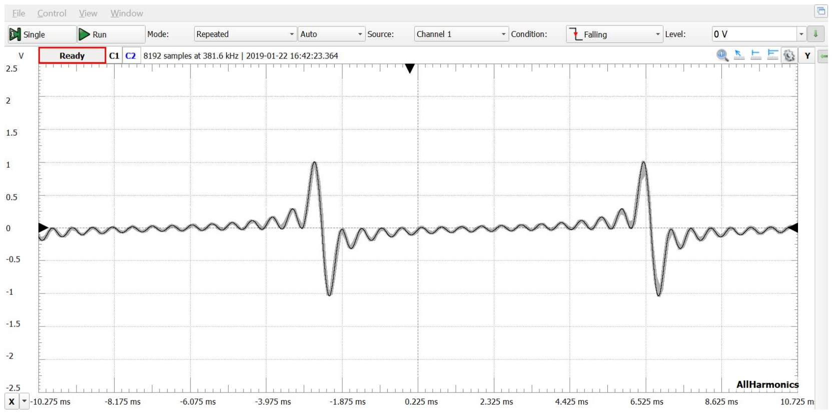

File Control View Window
Single Run Mode: Repeated Auto Source: Channel 1 Condition: Falling Level: 0 V
Ready C1 C2 8192 samples at 381.6 kHz | 2019-01-22 16:42:23.364
AllHarmonics

Figure 81 Observed 'All' Amplitude

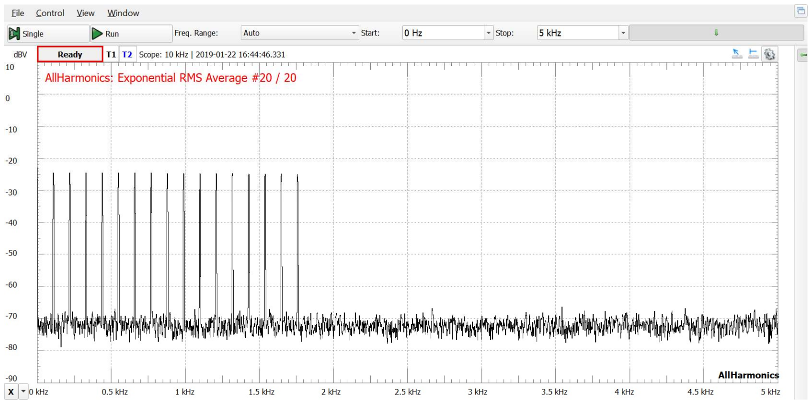

File Control View Window
Single Run Freq. Range: Auto Start: 0 Hz Stop: 5 kHz
Ready T1 T2 Scope: 10 kHz | 2019-01-22 16:44:46.331
AllHarmonics: Exponential RMS Average #20 / 20
AllHarmonics

Figure 82 Observed 'All' Spectrum

3.9.2 Vibrato

A sine waveform shows a signal without vibrato:

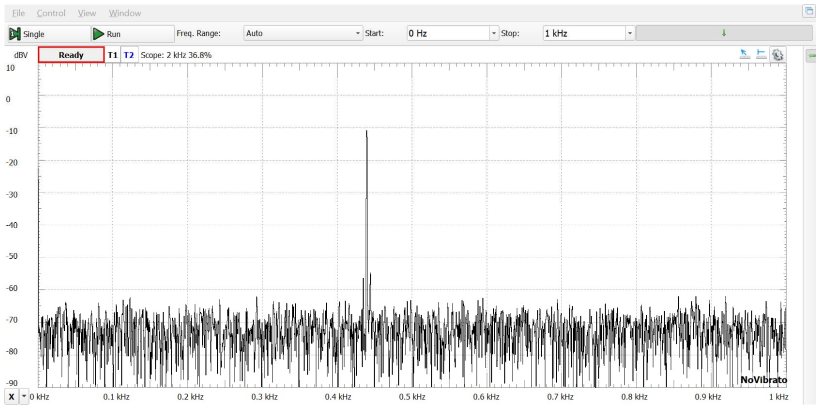

File Control View Window
Single Run Freq. Range: Auto Start: 0 Hz Stop: 1 kHz
Ready T1 T2 Scope: 2 kHz 36.8%
NoVibrato

Figure 83 No Observed Vibrato

The sine wave, modulated by the vibrato signal, has a more complex spectrum:

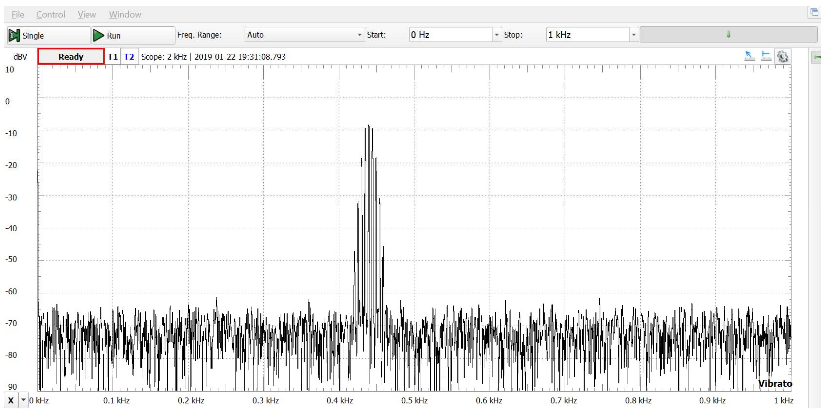


Figure 84 With Vibrato, Observed

3.9.3 Filter Simulation

The following set of figures show the All Spectrum waveform in Figure 82, modulated by various tracking filter emulations. Plots were grabbed as the corner frequency was swept from low to mid to high frequency:

3.9.3.1 Low-pass

The low-pass filter emulation plots look conventional, i.e. what one would expect if one were to pass the All spectrum waveform through a low-pass filter..

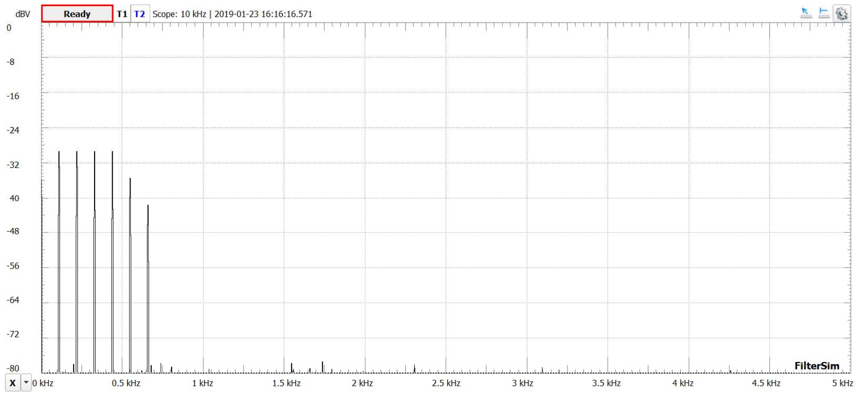


Figure 85 Observed 'Low-Pass Filter' - low corner frequency

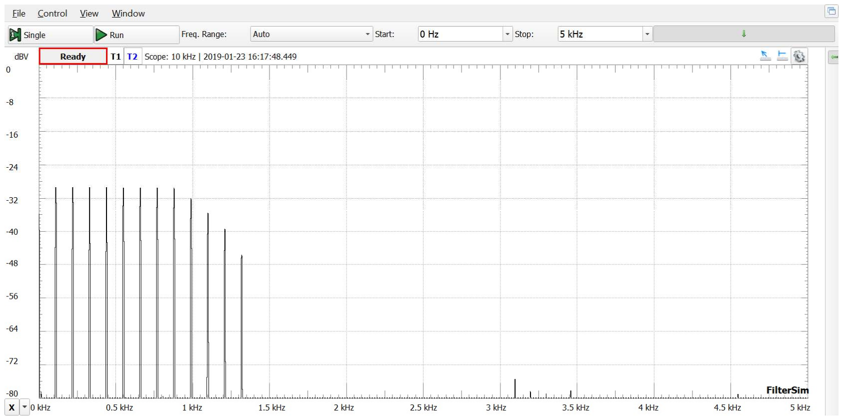


Figure 86 Observed 'Low-Pass Filter' - medium corner frequency

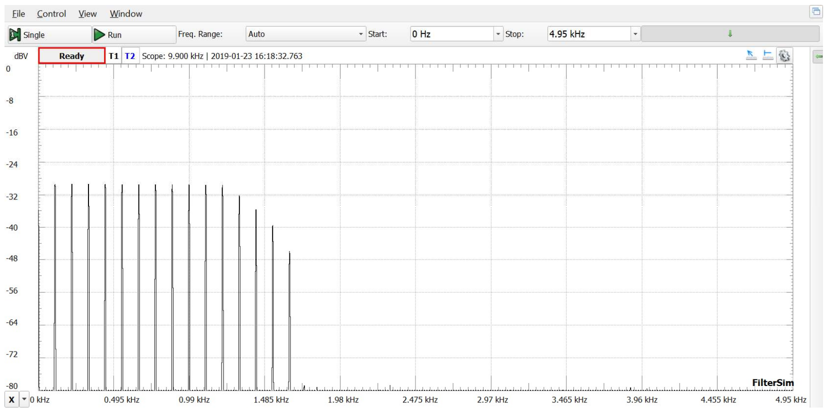
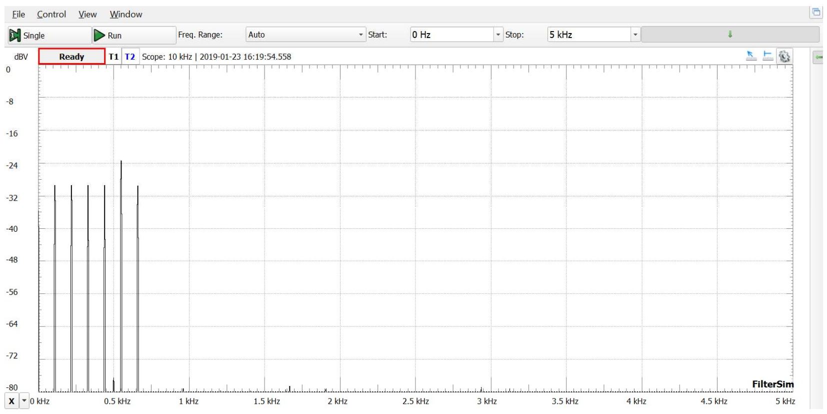


Figure 87 Observed 'Low-Pass Filter' - high corner frequency

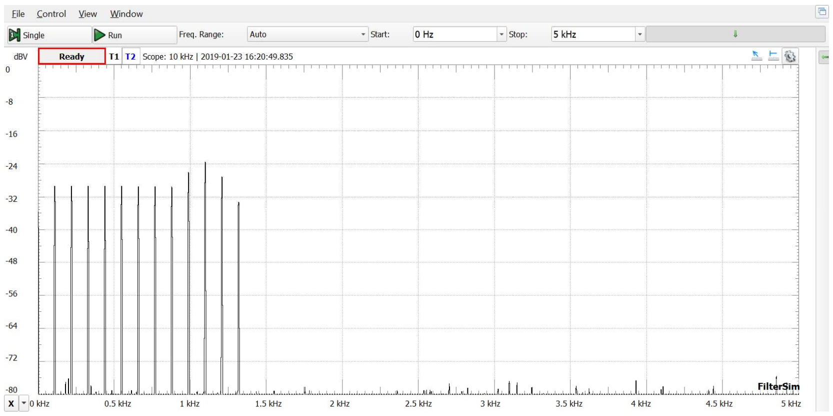
3.9.3.2 Resonant low-pass

The resonant low-pass filter emulation plots show a resonant peak, as expected


Freq. Range: Auto Start: 0 Hz Stop: 5 kHz

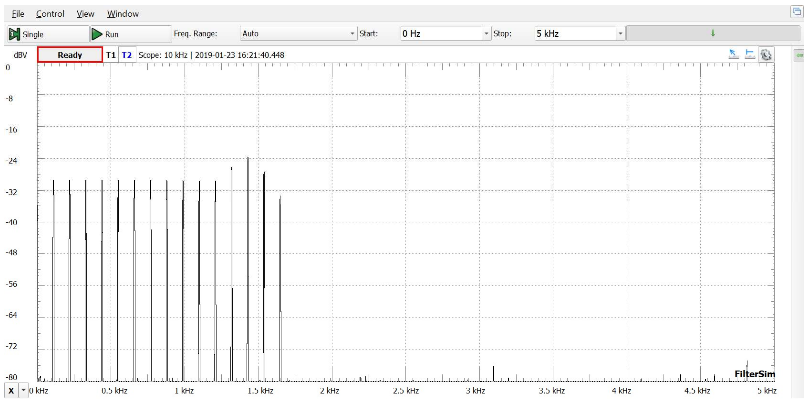
Scope: 10 kHz | 2019-01-23 16:19:54.558

Figure 88 Observed 'Resonant Low-Pass Filter' - low corner frequency


Freq. Range: Auto Start: 0 Hz Stop: 5 kHz

Scope: 10 kHz | 2019-01-23 16:20:49.835

Figure 89 Resonant Low-Pass Filter - medium corner frequency

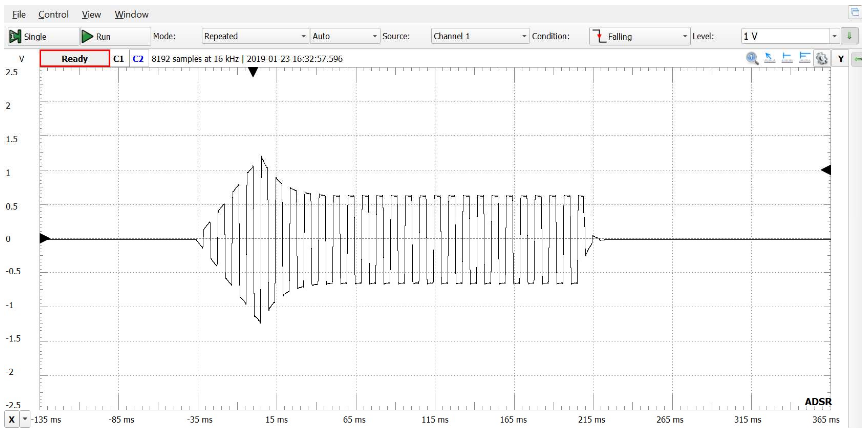

Freq. Range: Auto Start: 0 Hz Stop: 5 kHz

Scope: 10 kHz | 2019-01-23 16:21:40.448

Figure 90 Observed 'Resonant Low-Pass Filter' - high corner frequency

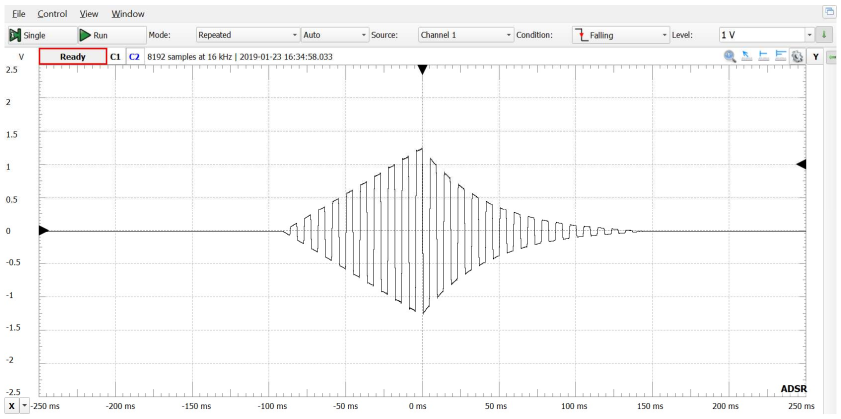
3.9.4 ADSR

The plots below show: an envelope with a long sustain and short decay, a ‘plucked String’-type envelope and an envelope with lower sustain level and longer decay.


Mode: Repeated | Auto | Source: Channel 1 | Condition: Falling | Level: 1 V

8192 samples at 16 kHz | 2019-01-23 16:32:57.596

Figure 91 Observed ADSR - Sustain


Mode: Repeated | Auto | Source: Channel 1 | Condition: Falling | Level: 1 V

8192 samples at 16 kHz | 2019-01-23 16:34:58.033

Figure 92 Observed ADSR 'Plucked String'

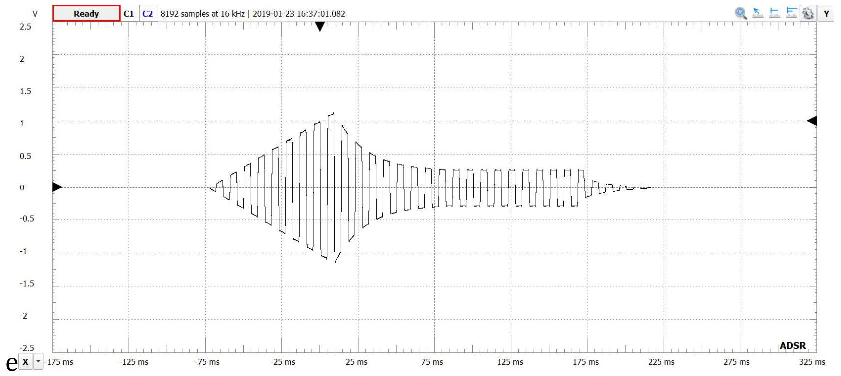

8192 samples at 16 kHz | 2019-01-23 16:37:01.082

Figure 93 Observed ADSR Sustain Level

3.10 User Evaluation

A selection of waveforms and effects was recorded on Audacity [65] at a 48 KHz sampling rate and a 24-bit resolution from a USB ADC. These were played back over an analogue keyboard amplifier to a 5-person panel whose scores and comments are summarised below:

Zynq-based Additive Synth	**Avg. score**	**Comments**
Sin	5	Smoother
Square	4.6	A little distorted
Trapezium	5	
Triangle	4.2	Stronger harmonics than PSoC
Sawtooth	4.6	Bright, buzzy
Impulse	4.4	Same as sawtooth
Range	3.6	Aliasing in the high notes
ADSR	4.6	
Vibrato	5	
Glissando	4.6	
Filter - LPF	4.2	Doesn’t sound as good as an analogue filter
Filter - Peaked	4.0	Doesn’t sound ‘analogue’h

The Survey Questionnaires are shown in Appendix – Survey Questionnaires

Connecting the keyboard aftertouch signal to the vibrato amplitude and the keyboard modulation wheel signal to the filter corner frequency allowed for expressiveness in playing. The overall sound quality was good.

3.11 FPGA Build Summary

The FPGA-based approach yielded a system that was difficult to realise:

- the set of three software tools used for development is not easy to learn and use[40].
- The gamut of Xilinx's documentation is of necessity, vast; hence finding the answer to any particular question is not quick, and
- the silicon compilation time on an Intel dual-core i5 was excessive, at about 30 minutes. To get this time down to something reasonable requires the use of a Workstation-class computer; not something readily available to a typical student.

One could not describe the IP used in this project, in the FPGA, as a modular synthesiser. It was, instead, a monolithic block which is difficult and time-consuming to modify.

The NCO-driven additive synthesis produces a waveform which within limits, has an arbitrary shape and is inherently band-limited, with minimal anti-aliasing artefacts. The frequency is stable, so that the synthesiser stays in tune.

The filter simulation produces low-pass and lowpass resonant 'filters'. The placement of the four (x, y) pairs describing the shape of the filter has a marked positive effect on the timbre of the sound.

3.11.1 Meeting the objectives

The Zynq-7000 development board costs around €200, putting it outside the means of most amateur enthusiasts. While the hardware design was modular, the long silicon compilation time makes it difficult to experiment with the modules' parameters and configurations; one has to regard the system as being monolithic. The time it takes to learn how to use the development tools, and the gamut of professional-level engineering skills necessary to complete the project puts it outside the capabilities of the amateur enthusiast.

The sound got high scores for quality during user evaluation, but was described as 'not analogue'.

[40] Xilinx states that it takes about six months to become proficient in learning the development tools.

4. Conclusions and Future Work

Each of the approaches above produced a synthesiser of good sound quality and playability. However, the analogue integrator was subject to drift, and needs improvement; this is probably best achieved by moving the function to the digital domain.. The mixed-signal approach presented in this work was cheaper in both component cost and development time. The FPGA-based approach was more expensive in component cost and expensive in development time. The mixed-signal approach is to be preferred.

4.1 Comparison

For both the Programmable System-on-Chip (PSoC)-based and FPGA-based approaches, the overall sound quality was perceived to be good. In both cases, the note generator was based on a numerically-controlled oscillator which was inherently free of frequency drift: both instruments stayed in tune.

The software tools used for PSoC were moderately difficult to learn and use: designing datapath elements was difficult, but re-use of existing datapath designs was easier.

The software tools used for FPGA development require time and practice to learn: the manufacturer states that there is a six-month learning curve involved in becoming proficient in their use. This was borne out in practice. Silicon compilation time was excessive, at around 30 minutes per run.

The PSoC approach, in conjunction with analogue off-board peripherals led to a system which was easy to build and genuinely modular, thus meeting one of the original goals of the project. The analogue signal path produced a sound that was reminiscent of classical 80's synthesis, thus meeting another.

The FPGA approach led to a monolithic block which was difficult and time-consuming to modify, thus failing to meet one of the project's goals. The digital signal path produced a high-quality sound in a reproducible manner.

The PSoC approach made it easy to reproduce the saws, triangles and ramps of classical 80's synthesis, but new filters[43], however pleasing to the ear, need a new hardware redesign for each one.

The FPGA approach was capable of reproducing an arbitrary waveform which is inherently band-limited, thus eliminating aliasing problems. The production of new emulated filters[44], using this approach, reduces simply to their specification in the frequency domain, making it easy to experiment with new filter types.

4.1.1 Goals

The PSoC approach produced a design for a cheap modular synthesiser suitable for experimentation and which emulates the sounds of the classic synthesisers of the eighties. The FPGA approachwas in this work, more expensive and required more effort to realize the synthesiser in hardware. The performance of the PSoC approach was adequate and low-cost. The performance of the FPGA approach, in this work,was good but more expensive in both material and development time.

[43] One of the listeners described the analogue 4-pole filter as "having a little distortion"; exactly the effect that was required.
[44] A listener described a low-pass resonant filter as "not sounding analogue". Perhaps a little distortion was needed.

4.2 Future Work

Future work for the PSoC approach involves replacing the analogue integrator with an on-chip digital version and including a second oscillator so as to be able to demonstrate phasing effects. In addition, the PSoC system and modules can be repackaged into a narrower form with a better arrangement of the pin connectors. This would allow the use of a solderless breadboard as a base for the modules, permitting the construction of an inexpensive modular synthesiser for use in an educational synthesiser laboratory environment.

Future work for the FPGA approach involves a GUI which allows one to draw a waveform and produce a set of band-limited Fourier coefficients which yield a closest match synthesised waveform. A similar facility for the filter shape would be convenient. It should also be possible to move some of the functionality from FPGA to code, yielding a more flexible and robust system design.

The present design uses about 12% of the resources of the Zynq FPGA used in this work, and as such an 8-note polyphony should be feasible.

References

[1] ARM. *AMBA 4 AXI4-Stream Protocol Specification.* **ARM** 2010
[2] ARM. *AMBA AXI Protocol Specification.* **ARM** 2004
[3] Xilinx. *Zynq Architecture, 14.2.* **Xilinx** 2012
[4] Sisterna, Cristian. *Introduction to AXI - Custom IP*. In **ICTP** Advanced School on Programmable System-on-Chip for Scientific Instrumentation. Triest, Italy 2017
[5] Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*. **Xilinx** 2017
[6] Xilinx. *Vivado Design Suite AXI Reference Guide (UG1037)*. **Xilinx** 2017
[7] Xilinx. *High-Level Synthesis Flow on Zynq using Vivado (Xilinx University Program >> Vivado >> Vivado-Based Workshops >> High-Level Synthesis Flow on Zynq using Vivado >> 2017.x Workshop Material).* **Xilinx** 2017
[8] Xilinx. *Vivado Design Suite Tutorial, High-Level Synthesis (UG871*). **Xilinx** 2016
[9] Xilinx. *Vivado Design Suite Tutorial, Embedded Processor Hardware Design (UG940),* **Xilinx** 2017
[10] Christopher Stillo, Duncan Mackay, Mike Mitchell. *Methods for Integrating AXI4-based IP Using Vivado IP Integrator (XAPP1204).* **Xilinx** 2014
[11] Kimon Karras, James Hrica*. Designing Protocol Processing Systems with Vivado High-Level Synthesis (XAPP1209)*. **Xilinx** 2014
[12] Beat Frei. *Digital Sound Generation*. Institute for Computer Music and Sound Technology (**ICST**), Zurich University of the Arts. 2010[66]
[13] Pete Symons. *Digital Waveform Generation*. **Cambridge University Press**. 2014
[14] Brendan Cronin. *DDS Devices Generate HighQuality Waveforms Simply, Efficiently, and Flexibly*. **Analogue Dialogue 46-01**, January 2012
[15] Xilinx. *DDS Compiler v6.0 LogiCORE IP Product Guide (PG141).* **Xilinx** 2017
[16] Texas Instruments. *16-Bit, Quad Chan, Ultra-Low Glitch, Voltage Output DAC w/2.5V, 2ppm/C Int. Ref. datasheet (Rev. D)*. **Texas Instruments**. 2011
[17] Harald Rosenfeldt. *Harald's Embedded Electronics: Zedboard Tutorials.* **Harald Rosenfeldt[67]**. 2018
[18] Crockett, Elliot, Enderwitz and Stewart.*The Zynq Book : Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC.* **Strathclyde Academic Media.** 2014
[19] Xilinx. *AXI4-Stream FIFO v4.1 LogiCORE IP Product Guide (PG080).* **Xilinx** 2016
[20] Xilinx. *AXI4-Stream Infrastructure IP Suite v2.2 LogiCORE IP Product Guide (PG085).* **Xilinx** 2018
[21] De Leon, Phillip. *Computer Music in Undergraduate Digital Signal Processing*. **American Society for Engineering Education**, **Gulf-Southwest Section**. 2000
[22] Chamberlin, Hal. Musical Applications of Microprocessors. **Sams**. 1985
[23] Self, Douglas. Small Signal Audio Design. **Elsevier** 2010
[24] Stinchombe, T. E.. *"Analysis of the moog transistor ladder and derivative filters."* **Stinchcombe[68]**. 2008
[25] Texas Instruments. *16AN-1515 A Comprehensive Study of the Howland Current Pump.* **Texas Instruments**. 2013
[26] Nease, Lanterman & Hasler. *A Transistor Ladder Voltage-Controlled Filter Implemented on a Field Programmable Analogue Array*. **Journal of the Audio Engineering Society**. 2014
[27] Vesa Välimäki. *Virtual Analogue Audio Effects and Synthesis*. **DAFX-13**. Ireland, 2013
[28] Briggs & Veilleux, *FPGA Digital Music Synthesiser*, **Worchester Polytechnic Institute**. 2015
[29] Lane, Hoory, Martinez and Wang, *Modeling Analogue Synthesis with DSPs*, **Computer Music Journal, Vol. 21, No. 4, MIT Press,** 1997
[30] Välimäki and Huovilainen, *Antialiasing Oscillators in Subtractive Synthesis,* **IEEE SIGNAL PROCESSING MAGAZINE**, 2007
[31] Kleimola, Lazzarini, Timoney, and Välimäki, *Phaseshaping Oscillator Algorithms for Musical Sound Synthesis*, **7th Sound and Music Computing Conference**, 2010
[32] Välimäki and Huovilainen, *Oscillator and Filter Algorithms for Virtual Analogue Synthesis*, **Computer Music Journal**, 2006
[33] Lazzarini and Timoney, *New Perspectives on Distortion Synthesis for Virtual Analogue Oscillators*, **Computer Music Journal**, 2010
[34] Ochiai, Yamaguchi and Kodama, *The Flexible Sound Synthesiser on an FPGA*, **First International Symposium on Computing and Networking**, 2013
[35] Joseph Timoney and Victor Lazzarini, *DIY instruments and effects: the potential for ubiquitious hardware platforms,* **IV Workshop on Ubiquitous Music** 2013
[36] S. Keshav, *An Engineering Approach to Computer Networking*, **Addison-Wesley**, 1997
[37] AN-1291, Digital Potentiometers: Frequently Asked Questions, Analogue Devices, Inc., 2015
[38] https://www.bhphotovideo.com/explora/pro-audio/tips-and-solutions/a-guide-to-analogue-subtractive-synthesis-with-the-moog-phatty-keyboard, B&H, retrieved 05/10/2019

[39] E Börger, *Abstract State Machines: A unifying view of models of computation and of system design frameworks* , Annals of Pure and Applied Logic, 2005 - **Elsevier**
[40] https://en.wikibooks.org/wiki/Sound_Synthesis_Theory/Additive_Synthesis#/media/File:SSadditiveblock.png, Sound Synthesis Theory, Wikibooks, retrieved 05/10/2019
[41] https://www.heathkit.com/, retrieved 05/10/2019
[42] https://dynaco.com/, retrieved 05/10/2019
[43] https://en.wikipedia.org/wiki/John_Linsley_Hood, John Linsley Hood, retrieved 05/10/2019
[44] https://www.parallax.com/product/bs2-ic
[45] https://www.microchip.com/design-centers/microcontrollers
[46] https://www.arduino.cc/
[47] http://andrewdotni.ch/blog/2015/02/28/MIDI-synth-with-raspberry-p/
[48] https://github.com/otem/Raspberry-Pi-Looper-synth-drum-thing
[49] https://www.youtube.com/watch?v=Gf4wISY6VHs&t=669s
[50] https://beagleboard.org/black
[51] https://bela.io/products
[52] http://ajaxsoundstudio.com/software/pyo/
[53] https://www.nolanlem.com/pdfs/synth.pdf. Retrieved 07/10/2019
[54] https://en.wikipedia.org/wiki/Programmable_system-on-chip. Retreived 07/10/2019
[55] https://www.cypress.com/. Retreived 07/10/2019
[56] https://en.wikipedia.org/wiki/Zero-order_hold Wikipedia. Retrieved 15/01/2019
[57] https://en.wikipedia.org/wiki/First-order_hold Wikipedia. Retrieved 15/01/2019
[58] https://reference.digilentinc.com/reference/pmod/pmodda4/reference-manual, retrieved 07/10/2019
[59] https://www.audacityteam.org, Audacity is an easy-to-use, multi-track audio editor and recorder, retrieved 20/09/2019.
[60] http://ccrma.stanford.edu/~jos/pasp/ , Smith, J.O. Physical Audio Signal Processing. Retrieved 21/08/2019.
[61] https://en.wikipedia.org/wiki/Additive_synthesis Additive Synthesis, Wikipedia, Retrieved 21/08/2019.
[62] https://www.json.org/ Douglas Crockford. Retrieved 05/12/2018
[63] http://analogueshield.com/user-manual.html William J. Esposito. 2014. Retrieved 09/12/2018
[64] https://www.harald-rosenfeldt.de/2017/12/28/zynq-spi-transmitter-using-an-axi-stream-interface/ Harald Rosenfeldt. ZYNQ: SPI Transmitter Using an AXI Stream Interface. 2017. Retrieved 10/12/2018
[65] https://www.audacityteam.org, Audacity is an easy-to-use, multi-track audio editor and recorder, retrieved 20/09/2019.
[66] https://www.zhdk.ch/en/researchproject/426386. Online Book. Retrieved 06/12/2018
[67] https://www.harald-rosenfeldt.de/category/zedboard-tutorials/. Retrieved 10/12/2018
[68] http://www.timstinchcombe.co.uk Online paper. Retrieved 14/01/2019
[69] https://meettechniek.info/additional/additive-synthesis.html Retrieved 27/03/2019
[70] https://en.wikipedia.org/wiki/RCA_Mark_II_Sound_Synthesiser Retrieved 27/03/2019
[71] https://en.wikipedia.org/wiki/Kawai_Musical_Instruments Retrieved 27/03/2019
[72] https://www.native-instruments.com/en/products/komplete/synths/razor/new-in-version-1.5/ Retrieved 27/03/2019
[73] Orr, T. and Thomas, D.W., "Electronic Sound Synthesizer" **Wireless World**, August 1973
[74] Shaw, G.D., "PE MinSonic" **Practical Electronics**, November 1974

A. Appendix - Programmable System-on-Chip-based Functional Block Diagram

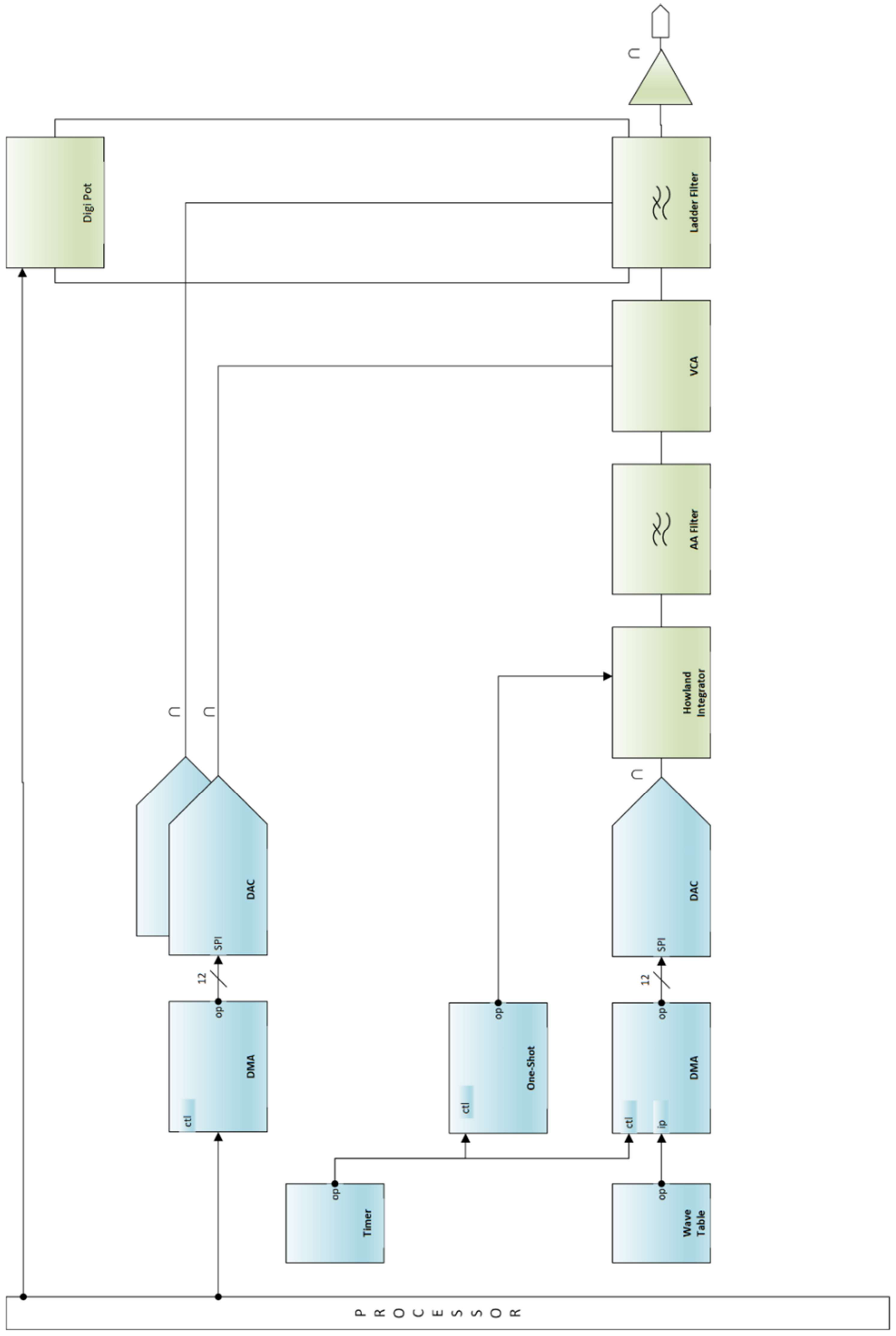


Figure 94 - Programmable System-on-Chip-based Functional Block Diagram

B. Appendix - Programmable System-on-Chip-based Schematics

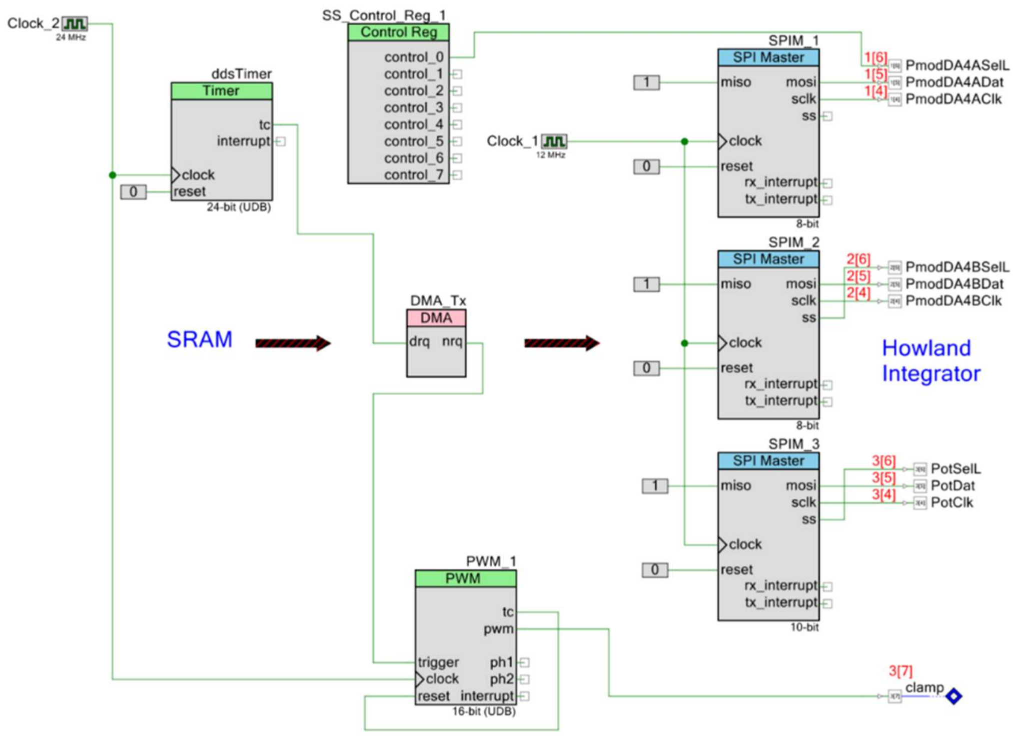


Figure 95 - PSoC Top Design 1

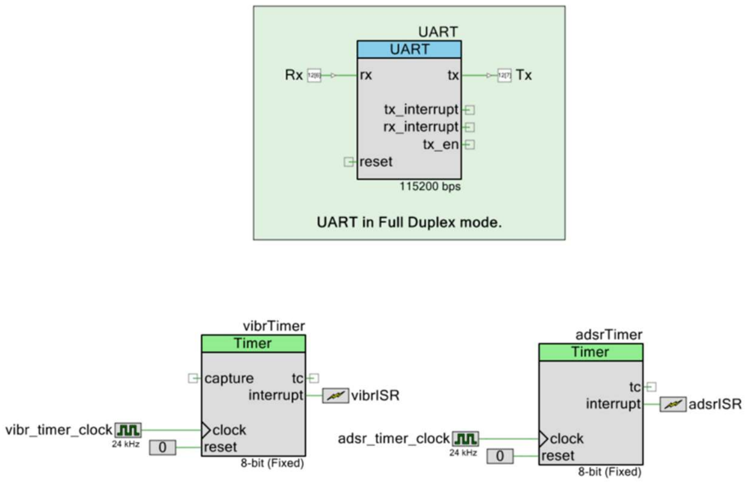


Figure 96 PSoC Top Design 2

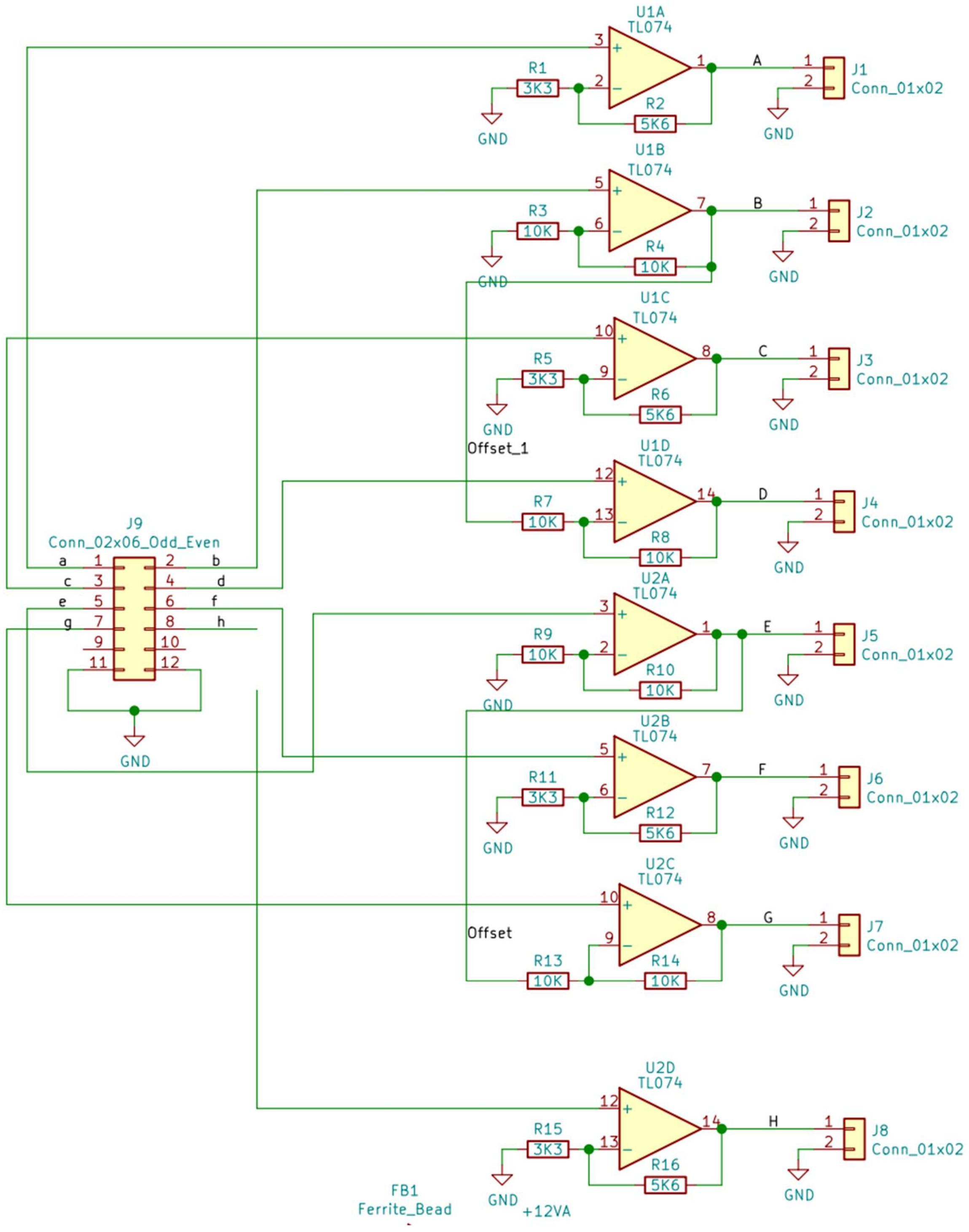


Figure 97 - DAC Buffers

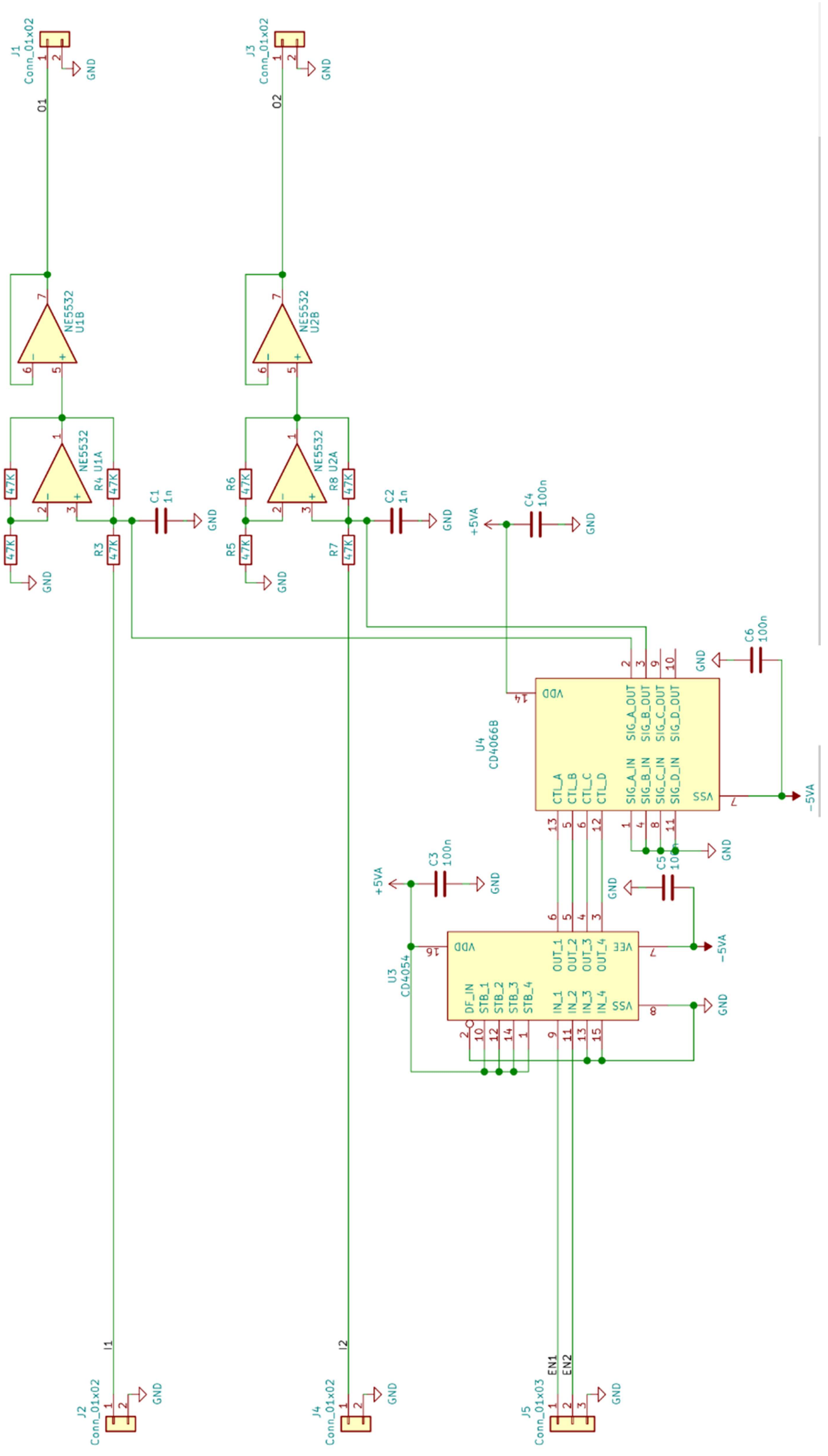


Figure 98 - Howland Integrator

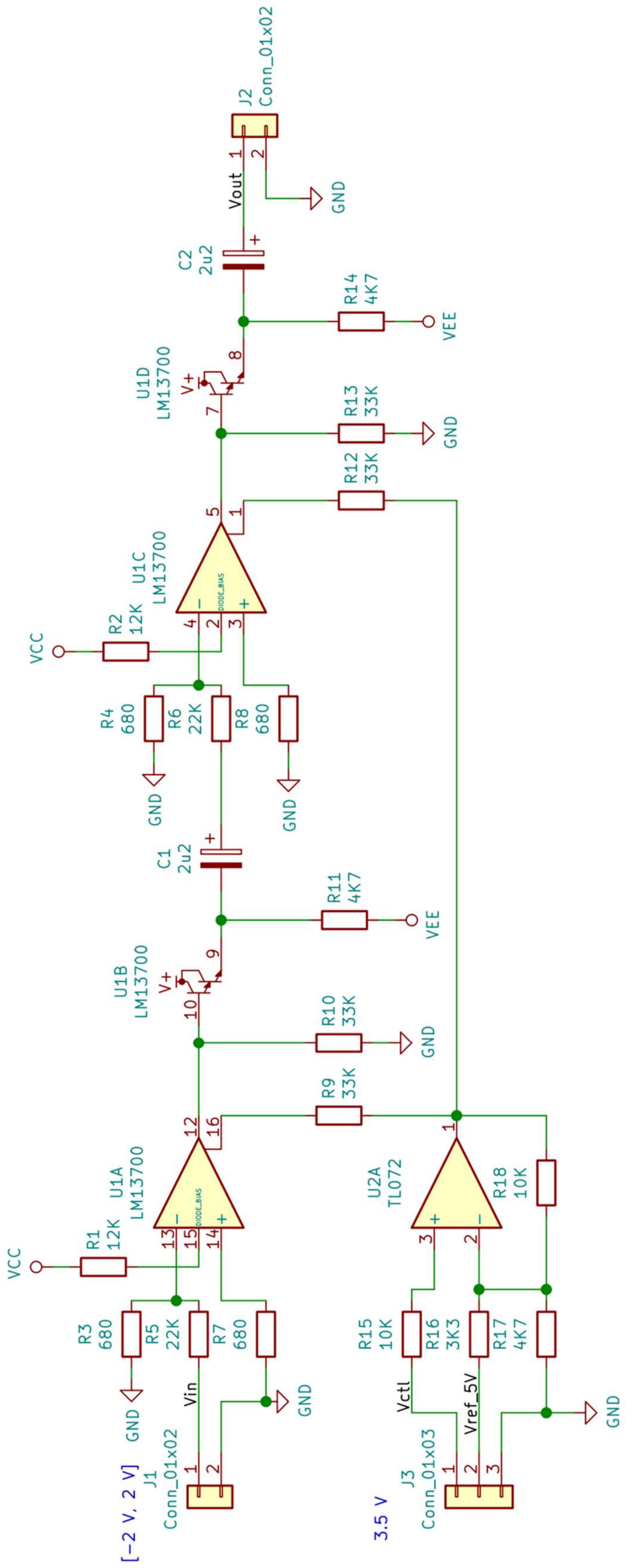


Figure 99 - Voltage-Controlled Amplifier

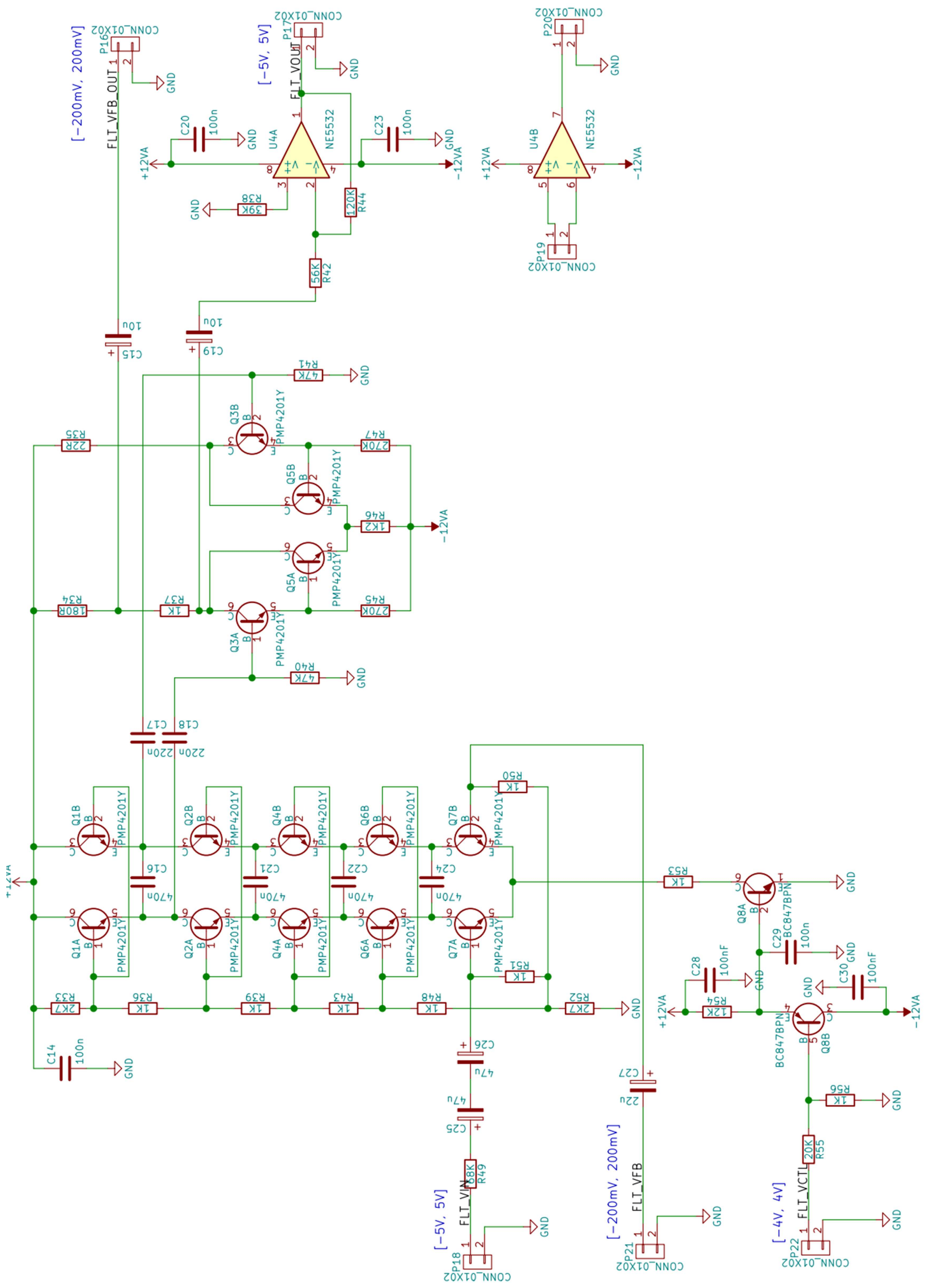


Figure 100 - Ladder Filter

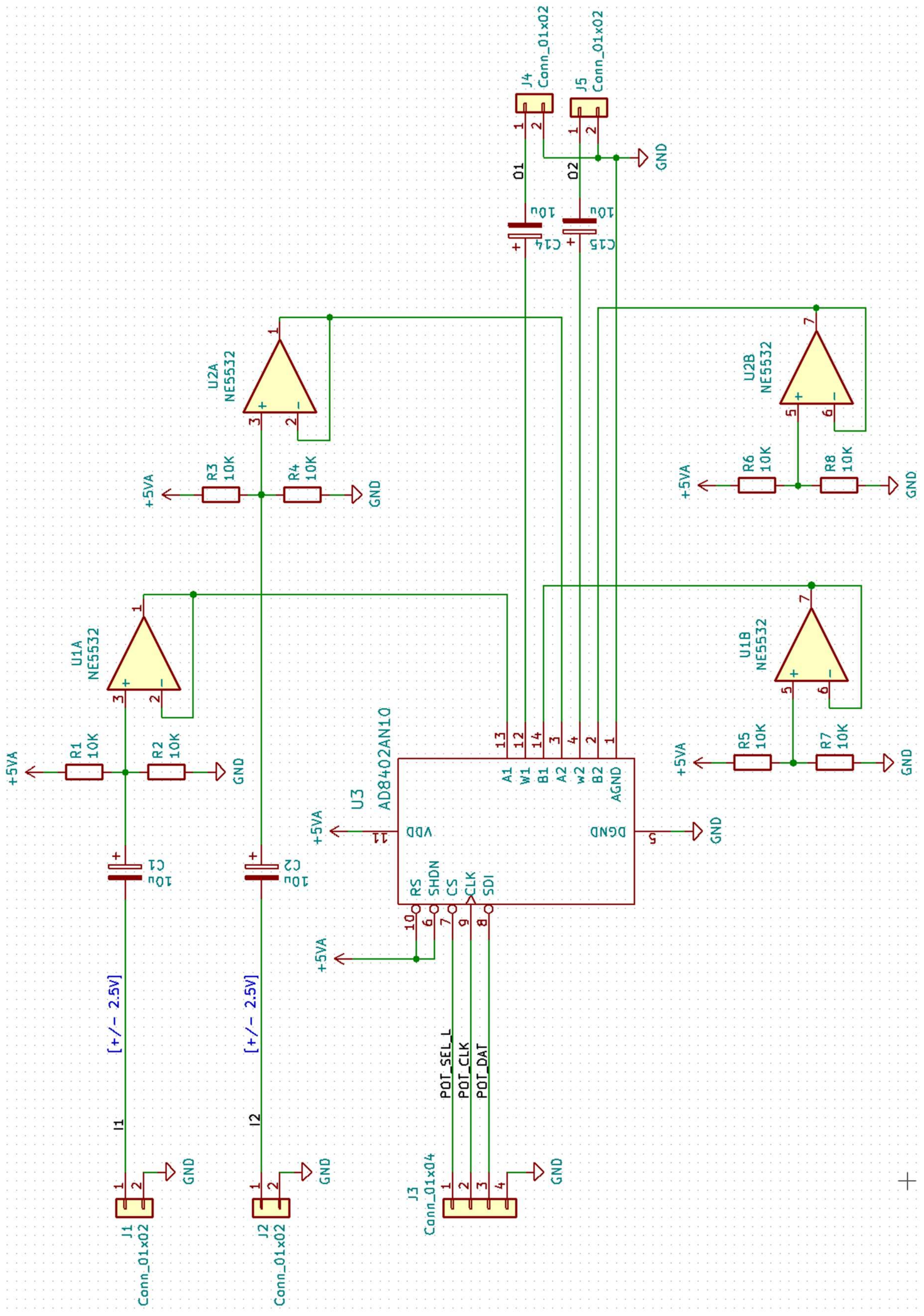


Figure 101 - Digital Potentiometer

C. Appendix – MIDI Controller

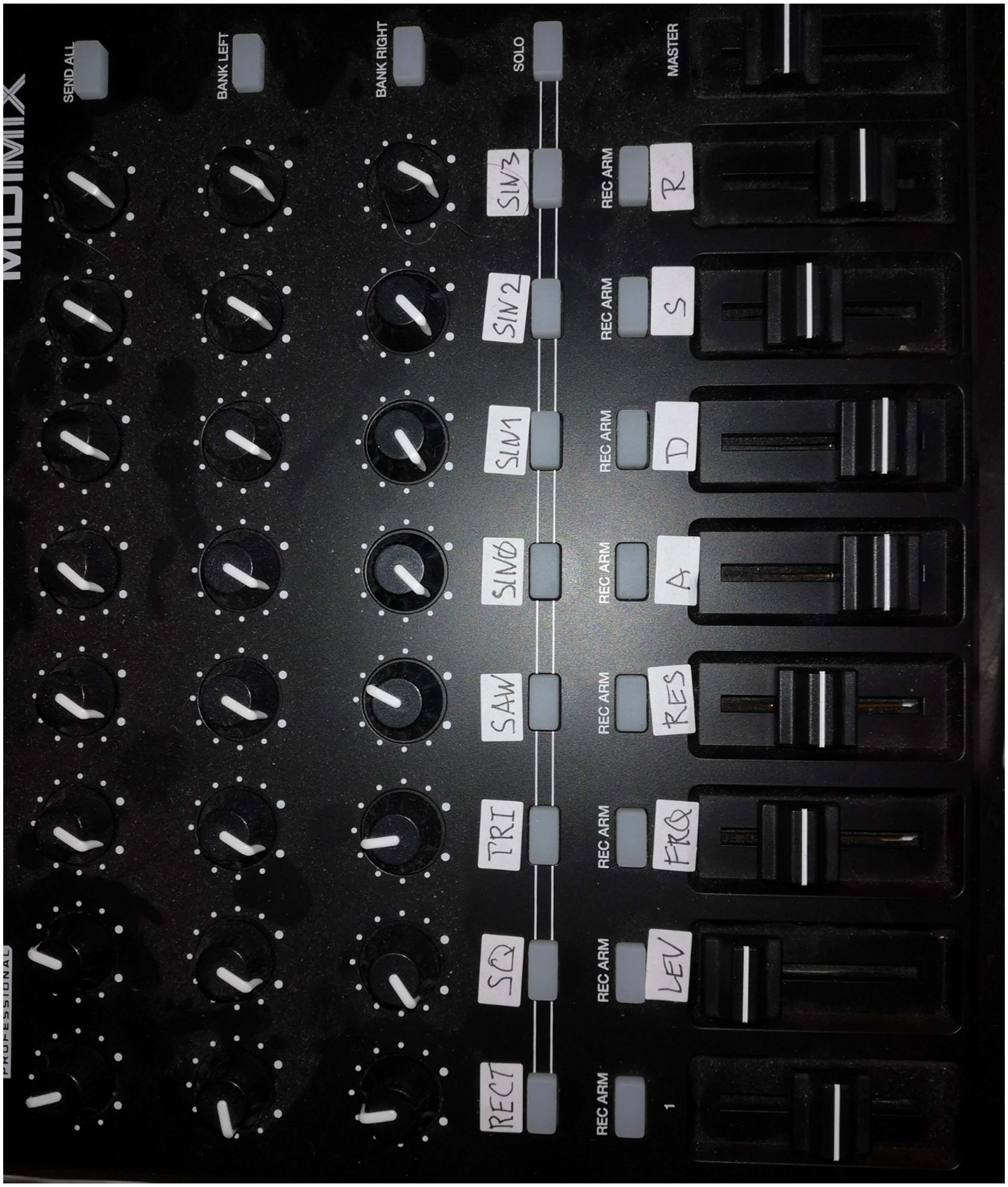


Figure 102 - MIDI Mix Controller

D. Appendix - FPGA-based Functional Block Diagram

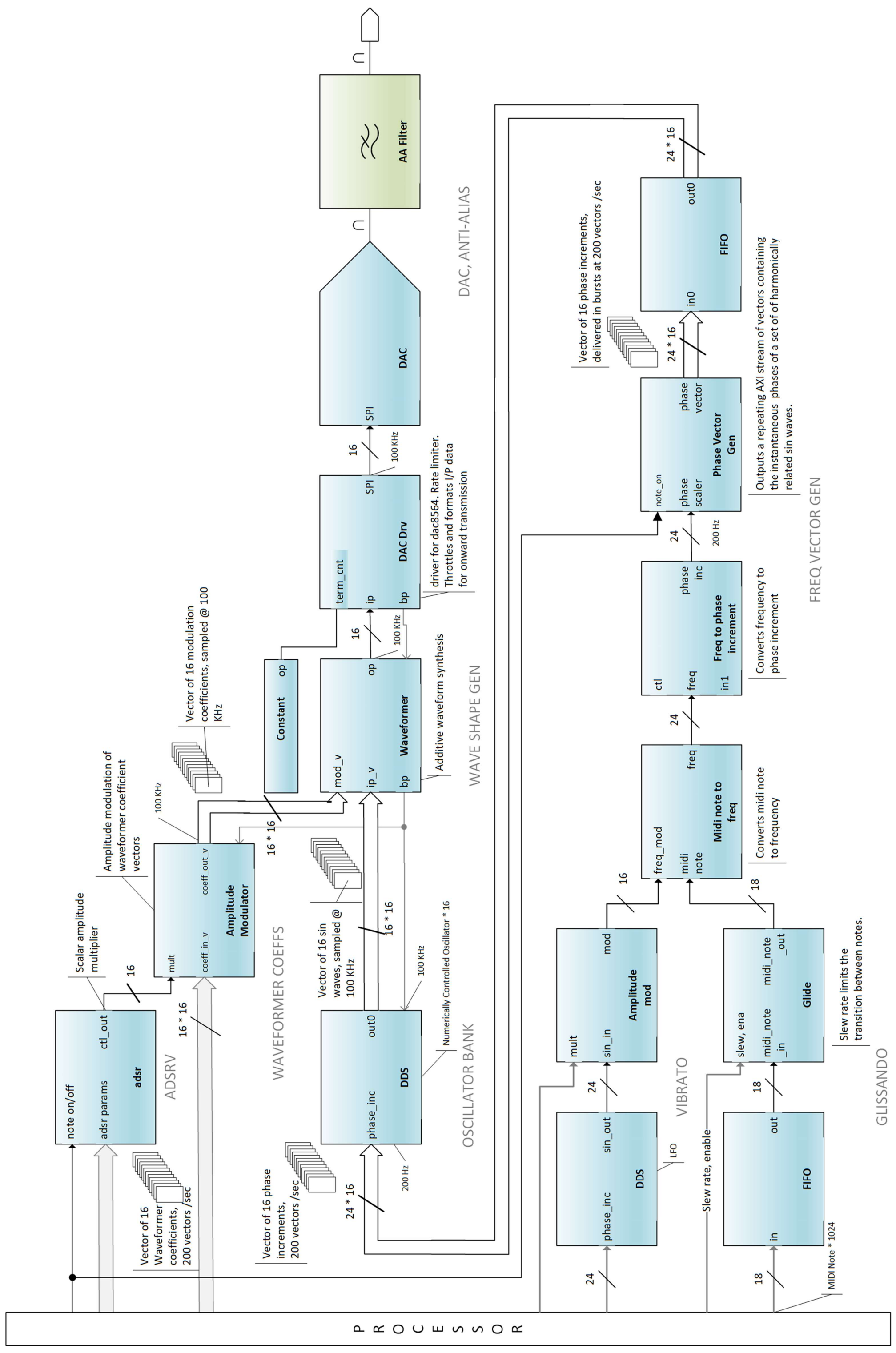

AA Filter
DAC
SPI
16
100 KHz
DAC, ANTI-ALIAS
term_cnt
ip
bp
SPI
DAC Drv
driver for dac8564. Rate limiter. Throttles and formats I/P data for onward transmission
16
100 KHz
Constant
op
mod_v
ip_v
bp
op
Waveformer
Additive waveform synthesis
WAVE SHAPE GEN
Vector of 16 modulation coefficients, sampled @ 100 KHz
Amplitude modulation of waveformer coefficient vectors
100 KHz
16 * 16
mult
coeff_in_v
coeff_out_v
Amplitude Modulator
Scalar amplitude multiplier
16
note on/off
adsr params
ctl_out
adsr
ADSRV
16 * 16
Vector of 16 Waveformer coefficients, 200 vectors /sec
WAVEFORMER COEFFS
Vector of 16 sin waves, sampled @ 100 KHz
16 * 16
100 KHz
phase_inc
out0
DDS
Numerically Controlled Oscillator * 16
OSCILLATOR BANK
200 Hz
24 * 16
Vector of 16 phase increments, 200 vectors /sec
24 * 16
in0
out0
FIFO
Vector of 16 phase increments, delivered in bursts at 200 vectors /sec
24 * 16
note_on
phase scaler
phase vector
Phase Vector Gen
Outputs a repeating AXI stream of vectors containing the instantaneous phases of a set of harmonically related sin waves.
FREQ VECTOR GEN
24
200 Hz
ctl
freq
in1
phase inc
Freq to phase increment
Converts frequency to phase increment
24
freq_mod
midi note
freq
Midi note to freq
Converts midi note to frequency
16
18
mult
sin_in
mod
Amplitude mod
VIBRATO
24
phase_inc
sin_out
DDS
LFO
24
slew, ena
midi_note _in
midi_note _out
Glide
Slew rate limits the transition between notes.
GLISSANDO
Slew rate, enable
18
in
out
FIFO
18
MIDI Note * 1024
P R O C E S S O R

Figure 103 - FPGA-based Functional Block Diagram

E. Appendix - Vivado Block Design

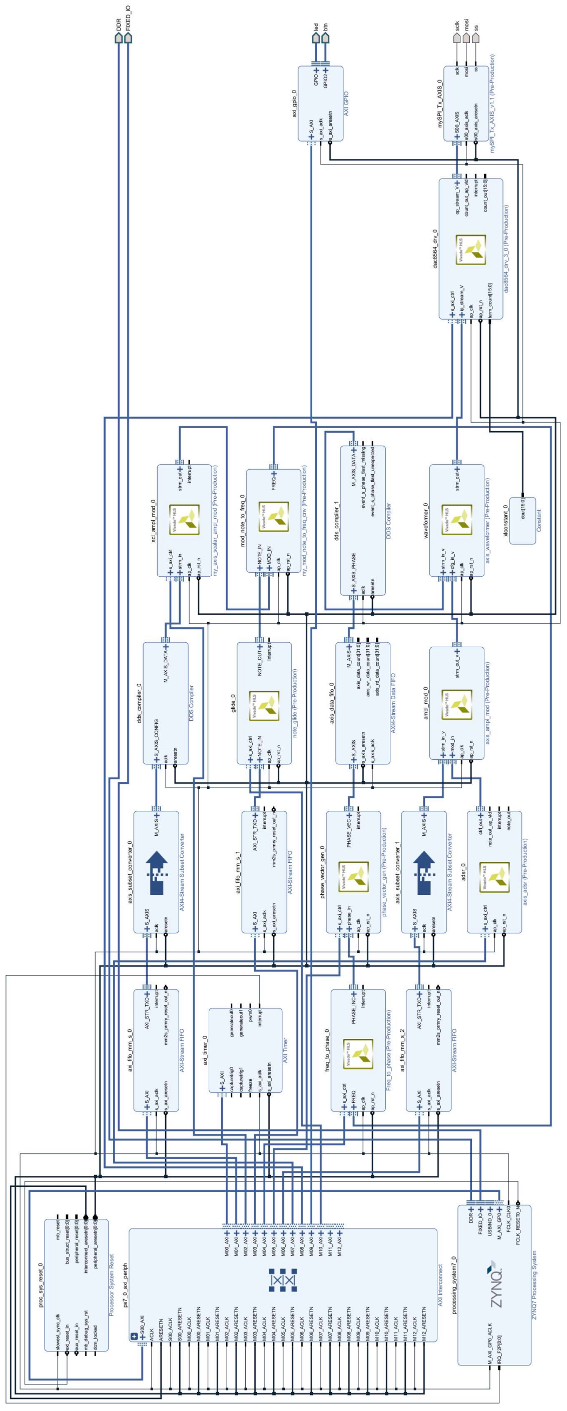


Figure 104 - Vivado Block Design

―

F. Appendix – Antialiasing Filter

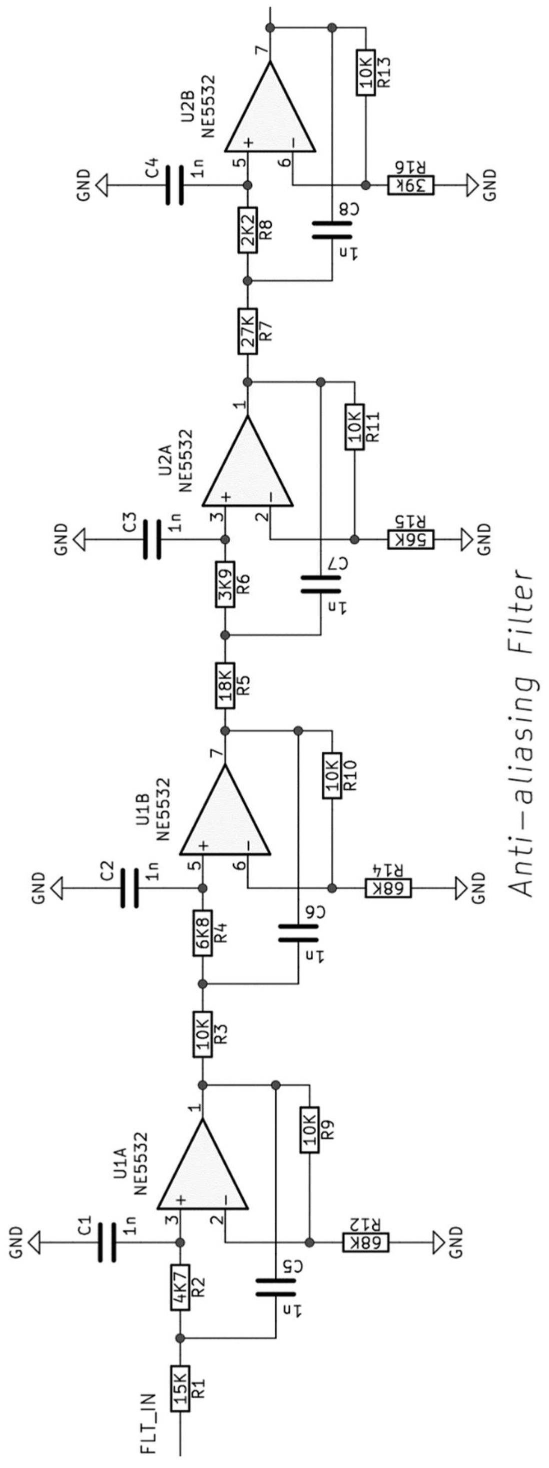
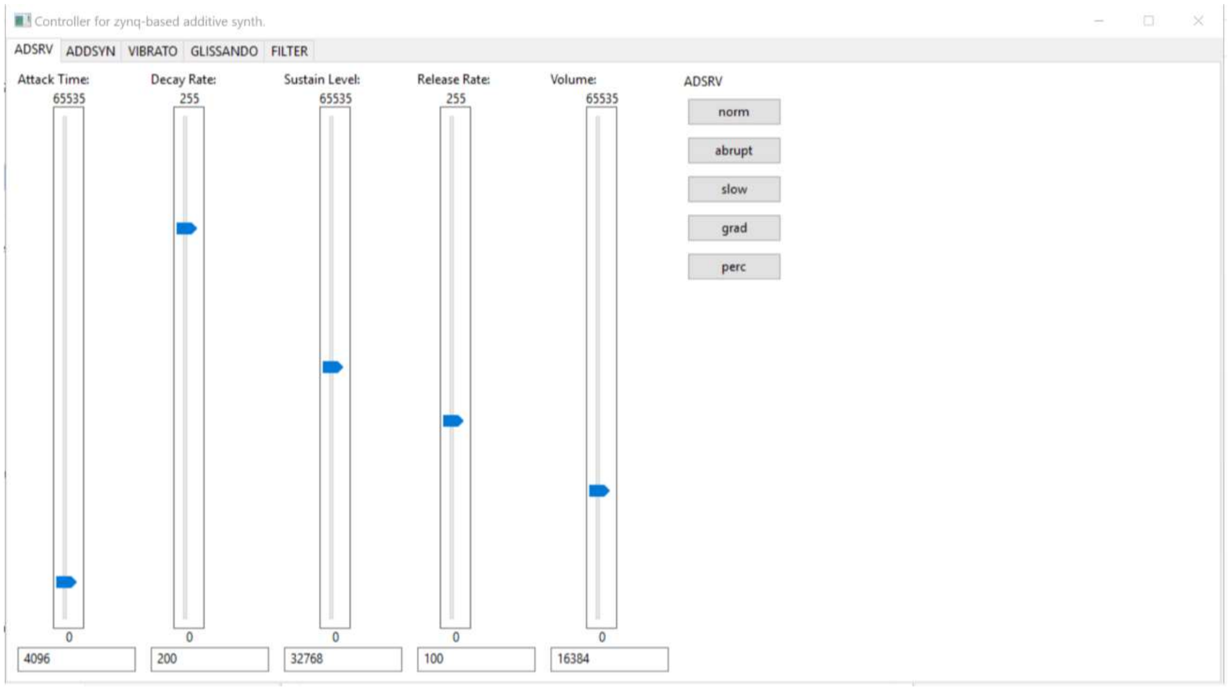


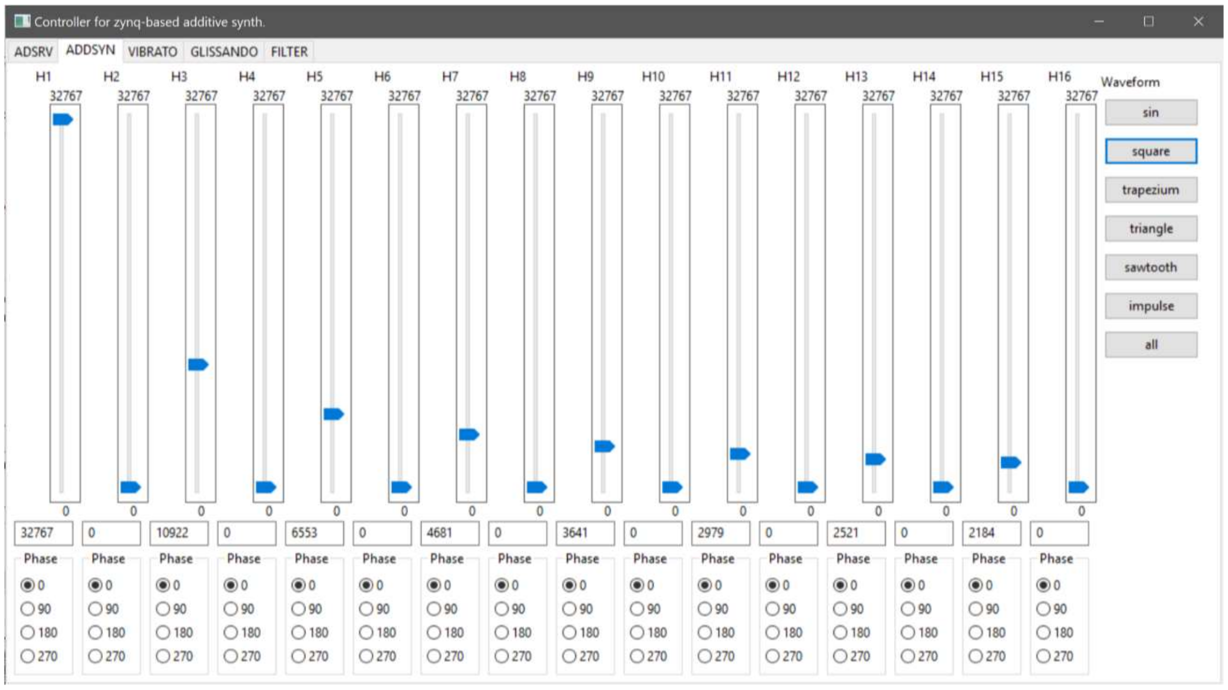
Figure 105 Antialiasing filter

G. Appendix – PC Controller Software – Screen Grabs



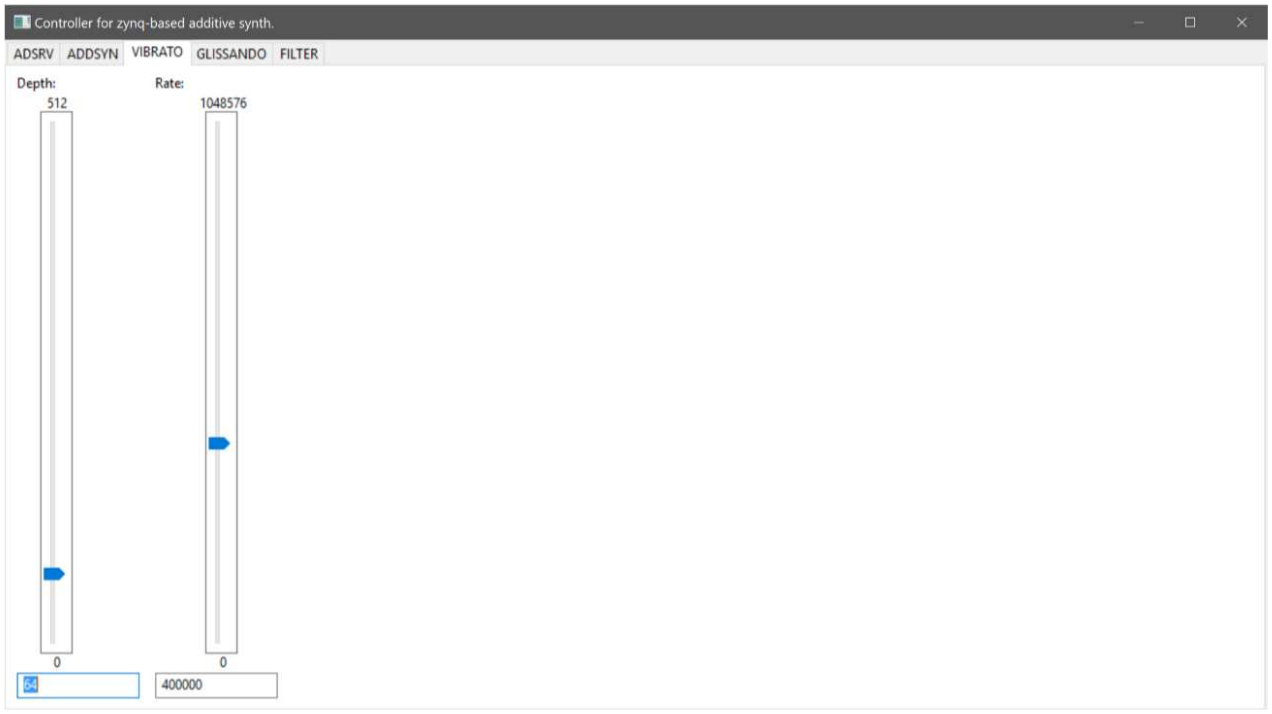
Attack Time:	Decay Rate:	Sustain Level:	Release Rate:	Volume:
65535	255	65535	255	65535
0	0	0	0	0
4096	200	32768	100	16384

Figure 106 ADSRV Screen Grab



	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	H11	H12	H13	H14	H15	H16
	32767	32767	32767	32767	32767	32767	32767	32767	32767	32767	32767	32767	32767	32767	32767	32767
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	32767	0	10922	0	6553	0	4681	0	3641	0	2979	0	2521	0	2184	0
Phase	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270	[✓ 0] 90 180 270

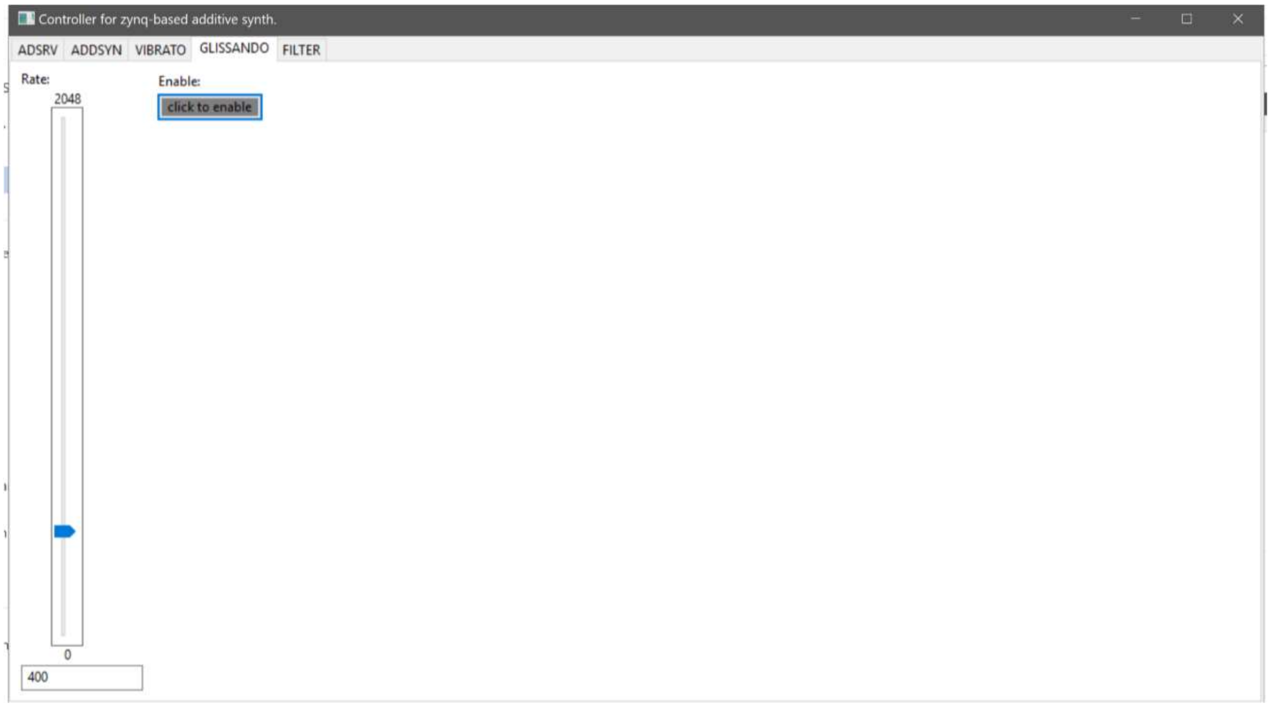
Figure 107 ADDSYN Screen Grab


Controller for zynq-based additive synth.

Depth: 512 0 64

Rate: 1048576 0 400000

Figure 108 VIBRATO Screen Grab


Controller for zynq-based additive synth.

Rate: 2048 0 400

Enable:

Figure 109 GLISSANDO Screen Grab

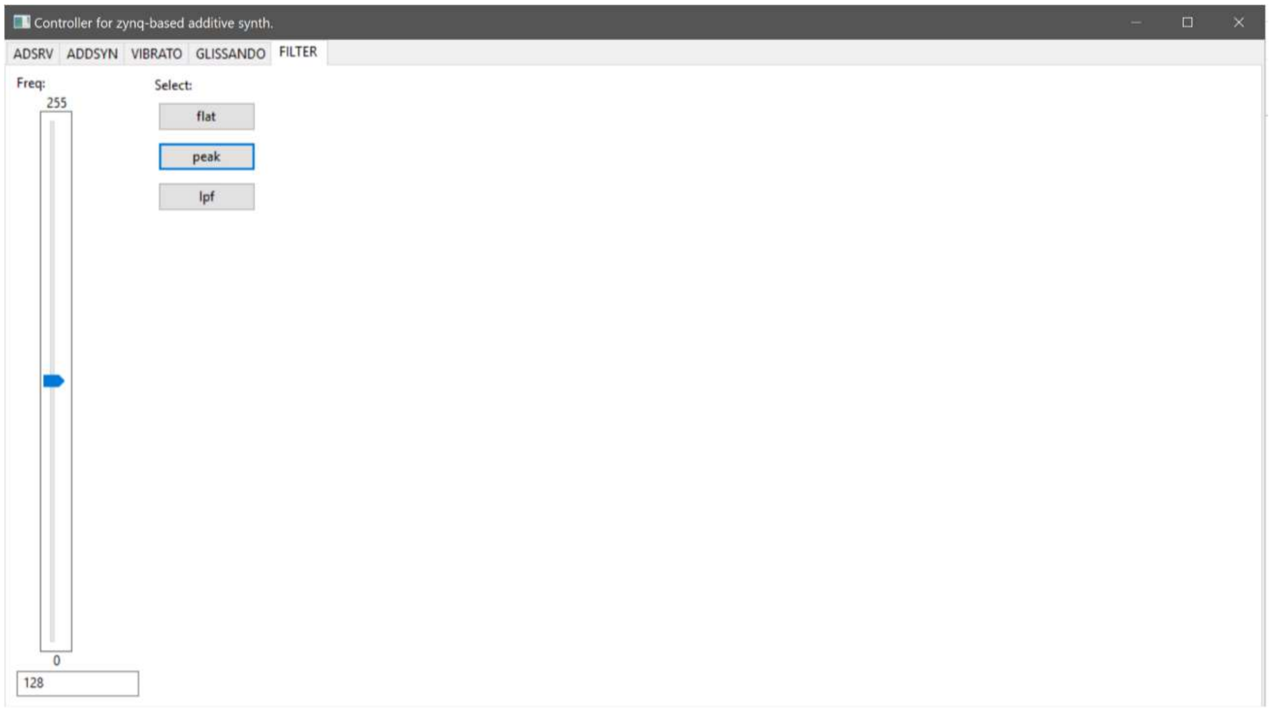


Figure 110 FILTER Screen Grab

PSoC-Based Subtractive Synth

Sin

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Square

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Rect

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Triangle

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Sawtooth

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Range

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Wheel

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Filter

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Zynq-based Additive Synth

Sin

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Square

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Trapezium

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Triangle

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Sawtooth

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Impulse

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

Range

○ 1	○ 2	○ 3	○ 4	○ 5	Remark

ADSR

○ 1	○ 2	○ 3	○ 4	○ 5	Remark