# VAD:
# VHDL AUDIO DEVICE

*by*

Ronen Agranat

A dissertation submitted to the Department of Electrical Engineering
in fulfilment of the requirements for the degree of

BSC (ENG) ELECTRICAL AND COMPUTER ENGINEERING

at the

UNIVERSITY OF CAPE TOWN

Supervisor
Simon Winberg




19 October 2010

# Declaration

I declare that this project is my own unaided work; all sources that I have used have been referenced and are listed in the Bibliography. It is being submitted for the degree of Bachelor of Science in Electrical and Computer Engineering at the University of Cape Town. This work has not been submitted before for any other degree or examination in any other university.

Signature of Author: .....

Cape Town

19 October 2010

# ABSTRACT

This fourth-year electrical and computer engineering thesis project details the design and implementation of a general purpose audio device on a Digilent Nexys2 evaluation board, supporting 44.1 KHz 12-bit audio monitoring and playback with 53 seconds of recording time. The emphasis of the project is on digital logic, VHDL and FPGA's.

Five track MIDI files can be played, with two polyphonic tracks and a drum track. The musical notes are synthesised by waveform generators implemented in the hardware domain. These oscillators are controlled by a software application running on the Microblaze soft core. The device makes use of external ADC and DAC chips to interact with analogue audio signals. The project is implemented using Xilinx Embedded Development Kit (EDK) 12.2.

The functional and performance characteristics of the device are investigated with a series of experiments which verify the prototype's functionality. These results are used to validate the thesis and prove or disprove the hypotheses.

# ACKNOWLEDGEMENTS

I would like to express my gratitude to **Professor Simon Winberg** for his guidance, supervision and assistance during the course of this thesis.

# NOMENCLATURE

| Acronym | Meaning |
|---|---|
| ADC | Analogue to Digital Converter |
| ASIC | Application-Specific Integrated Circuit |
| BSB | Base System Builder |
| CLB | Complex Logic Block |
| DAC | Digital to Analogue Converter |
| DSP | Digital Signal Processing |
| EDK | Embedded Development Kit |
| ELF | Executable Linkable File |
| FPGA | Field Programmable Gate Array |
| FSL | Fast Simplex Link |
| GCC | GNU Compiler Collection |
| GPIO | General Purpose Input/Output |
| IC | Integrated Circuit |
| IOB | Input/Output Buffer |
| IP | Intellectual Property |
| LUT | Look-Up Table |
| MIDI | Musical Instrument Digital Interface |
| OPB | On-chip Peripheral Bus |
| PLB | Processor Local Bus |
| PWM | Pulse Width Modulation |
| RISC | Reduced Instruction Set Computer |
| RTL | Register Level Transfer |
| SOC | System on a Chip |
| SOPC | System on a Programmable Chip |
| SPI | Serial/Parallel Interface |
| VHDL | VHSIC Hardware Descriptive Language |
| VHSIC | Very High Speed Integrated Circuit |
| VLSI | Very Large Scale Integrated |

Table 1: Nomenclature

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

CHAPTER 1

# OVERVIEW

## 1.1 INTRODUCTION

The aim of this thesis is to create a customisable audio device using the Digilent Nexys2-500 FPGA platform. The reconfigurable FPGA is exciting in an audio context, because it can naturally model the way in which different digital hardware and software audio devices co-exist in a typical audio electronics environment. All these devices and their interconnections could be integrated on a single FPGA System on a Programmable Chip with a soft core Microblaze processor. The resulting system could be customised to the specifications of a variety of target applications and could form the basis of a modular audio prototyping framework on the Digilent Nexys2 evaluation board.

## 1.2 HYPOTHESES

- **H0:** The 50 Mhz Digilent Nexys2-500 FPGA platform can perform 12-bit audio recording and playback at 44.1 KHz.
- **H1:** It can synthesise musical frequencies from 16.35 Hz to 4798.03 Hz to within a reasonable degree of error.
- **H2:** It interfaces effectively with 12-bit ADC's and DAC's using a 20 MHz SPI interface.
- **H3:** It can play MIDI files of four tracks with polyphony, meaning multiple notes can play at once.
- **H4:** It can perform these tasks in parallel, due to the concurrent nature of the underlying hardware platform.
- **H5:** It can be rapidly and effectively reconfigured for a wide variety of end audio applications.
- **H6:** It can also be used as a digital waveform generator producing sine, square, triangle and sawtooth waveforms at up to 30 kHz, with adjustable phase.

## 1.3 OBJECTIVES

***Requirements***

- **Audio file playback**

  VAD should be able to play audio data at a variety of bit-depths and sample rates. The device should have a variety of audio outputs, including built-in speakers, line outputs and headphones. There should be a volume control.

- **Audio recording**

  It should be possible to capture audio data onto the device through a variety of inputs, such as a built-in microphone and RCA line inputs. It should be possible to enable or disable inputs with built-in switches. It should be possible to play and loop recordings.

- **Audio sampling**

  It should be possible to load sound samples onto the device, which can be played through buttons built onto the device.

- **Audio synthesis**

  The device should feature oscillators which generate musical tones. These should have the performance and configuration parameters one expects of digital signal generators.

- **MIDI file playback**

  It should be possible to load MIDI files onto the device, which will be played through oscillators and audio samplers. MIDI file playback should be controlled with built-in pushbuttons.

- The device should be low-cost and so should use as few external components as possible.

- The device should be able to perform these audio functions concurrently; for example, it should be possible to play audio while recording.

- The device should be highly reconfigurable for different audio applications. For example, some applications may require stereo input and output while others may require no input but 5.1 channel output. Each system should be application-specific, and so represents a customisation, not a compromise, of the VAD platform.

## 1.4 SCOPE AND LIMITATIONS

It is assumed the reader is familiar with the fundamental concepts of electrical and computer engineering, and Digital Signal Processing (DSP) theory.

This document assumes the reader is familiar with FPGA's and VHDL. A brief review of these technologies is provided where relevant to the design.

The emphasis of this project is on digital logic.

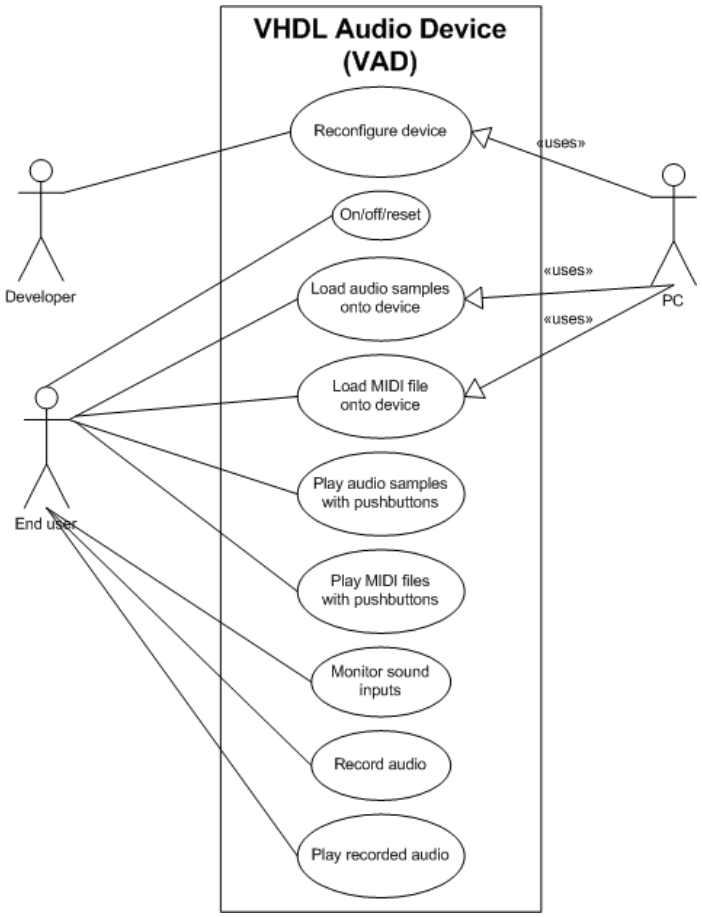


Figure 1.1: These use cases specify the ultimate functionality of VAD, illustrating how it will be utilised by various parties.

The VAD prototype which will be developed is intended to demonstrate the design and to prove it meets the requirements of this thesis. It is not itself intended to be an end application.

## 1.5 DOCUMENT OUTLINE

This document continues with a thorough review of the technologies used to implement this project, VHDL and FPGA's, then examines the specific evaluation board and development environment employed. A series of experiments will be designed which will test the hypotheses laid out in this chapter. Corresponding system reconfigurations will be designed, implemented and tested. The results of these tests will be analysed and used to draw conclusions.

CHAPTER 2

# LITERATURE REVIEW

This chapter establishes the platform on which the VAD prototype will be created and is crucial for producing a successful design. It establishes the theoretical basis for this project.

## 2.1 VERY HIGH SPEED INTEGRATED CIRCUIT HARDWARE DESCRIPTIVE LANGUAGE (VHDL)

VHDL was first a specification language for documenting the digital logic inventory of the United States Department of Defence. Simulators were then developed which could test VHDL designs without requiring fabrication, followed by synthesisers which could map functionality described in VHDL to the physical hardware resources of an FPGA, or used to stage and fabricate ASIC's based on VHDL specifications.

Thus, VHDL is a language for digital logic specification, simulation and synthesis. It combines documentation, debugging and design. It is an integrated language for integrated circuits.

Understanding the needs which VHDL itself was intended to fulfil will help when designing the prototype's hardware/software partition. A brief review of VHDL's history will be illustrative towards its usage.

### 2.1.1 Historical review of VHDL

The evolution of Integrated Circuits (IC) technology was guided through the 1970's by free market forces. The United States Department of Defence (DoD) had adopted a passive stance to the burgeoning industry, minimising state spending on IC research. In accordance with Moore's law, the number of logic gates on a single chip continued to rise meteorically, resulting in the advent of Very Large Scale Integrated circuits: exponentially larger, and more complex, than their predecessors. As the decade ended, the DoD realised it had lost much influence over the IC industry. At that stage, fewer than ten percent of the IC market share was attributable to the DoD [2, p.133]. The market had become saturated with components no longer satisfying the stringent quality requirements of the military; standards which were the mandate of the DoD.

In 1979, a United States government programme was launched which was to reverse prior policy and propel the Department of Defence to the forefront of Integrated Circuit research. The Very

High Speed Integrated Circuits programme was a tri-service collaboration among the United States army, navy and air force. VHSIC primarily targeted Application Specific Integrated Circuits (ASIC's), though no electrical technology was excluded [2, p.134]

The primary goals of the VHSIC programme were the following:

- **Reduce the operational costs incurred throughout the life-cycle of electronic equipment.** Some of this expenditure lay in reconciling documentation with physical devices, in porting low-level devices across platforms and in ongoing maintenance and modifications. Since electronic equipment purchased by the DoD may have a design lifetime of twenty years or more, an additional initial investment to reduce annual cost was worthwhile [1].

- **Reduce the turn-around time associated with deploying new technologies in the field.** This involved expediting all phases of the engineering process, from project inception to hand-over. It constituted nothing less than an overhaul of the digital systems development paradigm.

- **Improve the quality of such equipment through increased testability**. The development and standardisation of testing procedures was a goal from the outset. Testability leads to improved quality, since quality is quantified through testing. Built-In Self Test (BIST's) became common-place towards this end.

- **Increase the functionality of electronic systems.** Developer productivity could be improved by providing familiar, convenient language constructs. A language providing abstraction would allow designs to be specified at a higher level.

[2, p. 133]

The key to achieving these goals was identified as the development of a new Hardware Description Language (HDL). In 1981, the Department of Defence held a workshop in Woods Hole, Massachusetts, where government analysts studied existing Hardware Descriptive Languages, deducing requirements for the VHSIC HDL. These findings were released in 1983 as the Department of Defence's formal requirements for a language called VHDL.

The Department of Defence had been working on a high-level programming language called Ada since 1975. VHDL is syntactically similar to Ada, whose language specification was released in the same year [16].

In 1985, US government restrictions on the VHSIC programme were lifted and custodianship of the standard was transferred to the IEEE, a move which encouraged involvement from the IC industry as a whole. The VHDL 1076-1987 standard – VHDL87 – was ratified with the release of IEEE's VHDL Language Reference Manual in 1987. VHDL was the first industry-standard Hardware Descriptive Language. The Department of Defence spent more than twenty million dollars developing VHDL in that decade alone.

A specialist subcommittee of the IEEE continued to improve VHDL through the late eighties and into the nineties, culminating in the release of the VHDL93 standard and the associated Language Reference Manual in 1993. Work on the standard continued until 1994 [18]. This time period coincided with a surge in the popularity of Field Programmable Gate Arrays (FPGA's).

The Department of Defence required that contractees provide three VHDL deliverables for procured designs: a structural architecture, a behavioural architecture and a test bench. Such procurements had a value of three billion dollars in the eighties [1, p. 1-3].

The industry-wide adoption of VHDL and vigorous research led to an expanding role for VHDL in the IC industry. It is these roles and needs which have defined VHDL.

***Digital logic as Intellectual Property (IP)***

The high-level, abstract representation of digital logic offered by VHDL disembodies digital hardware from a physical manifestation. The VHDL specification itself becomes a valued commodity, protected by standard copyright treaties in the case where designs are published for sale, and possibly also by patents where applicable.

This leads to the term *Intellectual Property (IP)* being commonly used to describe products or designs which are available for purchase in the form of VHDL. This is the same legal protection enjoyed by software source code. [16, p. 233-238]

### 2.1.2 Technical review of VHDL

***Standard VHDL libraries***

VHDL supports libraries, since they facilitate modular development which expedites development time [18, p.27].

The two standard VHDL libraries on which most VHDL designs depend are *std_logic_arith* and *numeric_std*. The former is a de-facto standard originally developed by Synopsis; the latter is an IEEE standard. Both typically ship with VHDL93 toolkits. The developer chooses which library to use: only one should be included in a given source file. *numeric_std* is considered more modern and correct. [13, p. 5]

The standard libraries provide language constructs through the definitions of datatypes, functions and operator overloads.

***VHDL datatypes***

Below is a brief review of the VHDL datatypes which will be used to implement the VAD prototype.

- **std_logic** is the elemental type which succeeds `bit` and allows for the many states of line-level logic to be represented, such as logic-high, -low, high-impedance, floating, and so on.

- **std_logic_vector** is an array of `std_logic` bits and the low-level representation of digital values. It has no intrinsic sign. Bit width is explicitly stated, and the individual bits are accessed through Ada index notation, introducing the concept of endianess.

```
-- big_endian(0) is MSB
signal big_endian : std_logic_vector(0 to N-1);

-- little_endian(N-1) is MSB
signal little_endian : std_logic_vector(0 downto N-1);
```

- The **unsigned** type is a `std_logic_vector` which is explicitly non-negative, and the **signed** type is a `std_logic_vector` which is represented in two's-complement. These represent an intermediate-level form where arithmetic is possible.

- **integer** is a high-level numeric type. It has a concept of sign and indirectly of bit width. The number of bits to be used for an integer is inferred from the given `range`. If the range is [0,N-1], the number of bits is `ceiling(log2N)`. If no range is specified, XST assumes the maximum bit width for integers of 32 bits. Negative ranges can be specified. Values will be truncated to the smallest range of the integers involved in a mathematical operation.

```
-- 32 bits
variable an_integer : integer := 0;
-- 8 bits
variable another_integer : integer range (0 to 255) := 0;
```

  Integer maths is best performed using this datatype and `numeric_std`. This project will use a Microblaze with no floating point unit, so the integer subtype will correspond closely with the rest of the System on a Programmable Chip.

- An **integer subtype** is an `integer` with a defined range and a name. The range can be defined in terms of **constant**s. These make it convenient to specify the bit widths for the various digital values in the system in terms of their respective ranges and purposes. A Register-Transfer Level (RTL) structure is inferred.

```
constant FOO_MAX_VALUE : integer := 63;
subtype foo_type is integer range (0 to FOO_MAX_VALUE);
...
variable foo : foo_type := 0;
```

  The use of integer subtypes is highly recommended, since they use optimal logic resources and provide a convenient programming construct [3].

***Strong typing***

VHDL is a strongly-typed language, meaning that the synthesiser cannot automatically infer when data must be converted from one internal representation to another. These casts must be performed manually. This enforces a clear, formal style of source code, with little room for guess work.

A VHDL datatype has the following properties which contribute to its strong type:

- A bit width, dictating the range of values the datum can assume and amount of resources required to represent the value.
- Whether the datum is signed and can be either positive or negative.
- Whether the datum is big-endian or little-endian. The Microblaze processor is big-endian.
- Whether the value has some other internal format, such as fixed-point or floating-point. This is an issue when interfacing with entities created with tools such as Xilinx's *AccelDSP* or Matlab's *Simulink*.

VHDL will not synthesise correctly when faced with types varying in one or more of the above factors. Compared to the average strongly-typed language, then, VHDL is *super*-strongly typed. This contributes to its learning curve.

***Mathematics with VHDL***

Consider two primary types of mathematical operation: arithmetic and logical.

Bit-wise logic operations, such as AND, OR, shifts, rotates, concatenations and so on, are best implemented directly with Ada index notation and the concatenation operator (&) with `std_logic_vector`.

Arithmetic operations are best implemented using `integer` types and subtypes. `numeric_std` provides the arithmetic operators. Multiplication (*) and division (/) implicitly map to native macrocells in the FPGA.

Boolean and evaluative expressions are also available. The equality operator '=' is less resource-intensive than the comparator operator '>='.

***VHDL Datatype Casting***

It is often desirable to convert data from one datatype to another, for performing arithmetic, bit-wise logic or for input/output constraints. The following is the recommended technique for achieving this :

```
-- std_logic_vector to integer cast.
an_integer := to_integer(unsigned(a_std_logic_vector));
```

```
an_integer := to_integer(signed(a_std_logic_vector));

-- integer to std_logic_vector cast.
now_a_vector <= std_logic_vector(to_unsigned(an_integer,32));
now_a_vector <= std_logic_vector(to_signed(an_integer,32));
```

[13, p. 17]

### *VHDL Design Patterns*

The two methods of specifying designs in VHDL are structurally and behaviourally. This project will employ the behavioural method, where an entity is specified with ports and internal processes.

Each process is triggered from ports specified in its sensitivity list. Synchronous processes are those triggered from a common clock line.

The processes are essentially concurrent within themselves and with one another. VHDL does provide sequential language constructs such as `if`, `then`, `else` and so on, but these are for convenience and are synthesised as equivalent Register-Level Transfers (RTL's) [18, p.27].

State machines can be implemented using one process per state, or monolithically. A monolithic design has one primary synchronous process. This process is sensitive to a primary clock line and also to an asynchronous reset line. It acts only on the rising or falling edge of the clock.

### *Signals versus Variables*

Signals in VHDL are low-level constructs which emulate electronic connections and can also be used as registers. They are commonly found in structural features such as entity port and architecture declarations.

Variables are a higher-level programming construct which correspond more closely to the computer science variable, and are restricted to processes. Conceptually, variables update immediately whereas signals have a propogation delay. This means that variables can be updated several times within a process, but signals only once, making variables appropriate for the specifying sequential logic [18, p.27].

A distinction between variables and signals used to be that signals could be shared among all processes within an entity, while variables could not. With VHDL93, *s*hared variables can be be declared which are somewhat available to all processes in an entity.

### *Generics*

Generics allow VHDL entities to be customised with external parameters [18, p.27], and in this way are similar to the constructor of computer science. They are specified within VHDL similarly to port maps. EDK features the ability to generate a GUI configuration dialog box for custom IP cores, allowing the various generics to be specified.

## 2.2 FIELD PROGRAMMABLE GATE ARRAYS (FPGA'S)

This project is implemented on a Xilinx FPGA. It is important to understand FPGA's, their history and how they work, in order to understand this project. Someone who does not understand the function of FPGA's will not understand the purpose of this project.

### 2.2.1 Historical Review of FPGA's

In 1978, Monolithic Memories, Inc. recognised the need for a chip providing customisable digital logic. This chip, the Programmable Array Logic (PAL), could be programmed by the developer to perform a customised digital logic function in a single 20-pin Dual Inline Package (DIP): a viable alternative to connecting many discrete logic gates together during assembly.

The Programmable Array Logic (PAL) was a customisable 'sea of logic gates'. The input pins on the chip and their complements are fed, for example, into AND gates which cascade into OR gates. Unprogrammed, every possible connection between input pin and logic gate could be closed. The PAL's function is defined by destroying fuses at undesired connection points (or anti-fuses, where explicit connections between gates must be made). This structure coincides closely with the normal minterm form of Boolean functions [15, p. 49], making it widely applicable. The fuse mechanism was provided by PROM, meaning these early devices could be programmed only once.

The Programmable Array Logic chip, which simplified assembly, decreased cost and increased efficiency, spawned a new industry: Programmable Logic Devices (PLD's).

The invention of the FPGA in 1984 was contrary to market trends at the time. The market overvalued transistors, tending to minimise the number of transistors on each chip. Xilinx co-founder Ross Freeman recognised that, in accordance with Moore's Law, the price of transistors should fall rapidly with increasing transistor density. While other vendors were developing equipment with as few logic gates as possible, Xilinx created a device with more logic gates than could be utilised: the Field Programmable Gate Array (FPGA). "It was a radical concept that required lots of transistors at a time when transistors were considered extremely precious," recalled colleague Bill Carter [23].

FPGA's used Look-Up Tables as the fundamental logic element, rather than logic gates. The SRAM could be programmed with a Look-Up Table corresponding to the truth table of a desired Boolean function. The input to the logic block then corresponds to some address in SRAM, while the output is based on a Boolean combination of the data stored there, the input address and other inputs, configuration and digital logic.

The static nature of SRAM means the FPGA needs to be configured only once, at start-up. The FPGA SRAM is typically loaded from an Electrically Erasable PROM (EEPROM) by dedicated circuitry using dedicated pins. The use of SRAM and EEPROM over the PROM of PAL and Flash of CPLD's became a historical distincation between FPGA's and other PLD's, though this distinction is becoming blurred as Flash memory improves.

The realisation that memory could also be used to control the *routing* of data within the chip

led to a major breakthrough in the configurability of Programmable Logic Devices, since it was now possible to develop fully-customised datapaths. These new devices were termed 'reconfigurable' and heralded the era of reconfigurable computing.

In 2009, Ross Freeman was inducted into the American National Inventors Hall of Fame, credited with inventing the FPGA as US patent number 4 870 302: "Configurable electrical circuit having configurable logic elements and configurable interconnects".

***FPGA's vs. ASIC's***

FPGA's began to converge with ASIC's as a modeling, simulation and development environment which required no fabrication for testing, fulfiling an important objective of VHDL. FPGA's are often used to design, develop and prototype systems which could ultimately be fabricated as ASIC's, if so desired. Historically, FPGA's have been used in low volume prototyping or production environments, while ASIC's are produced in very large quantities at a foundry. Recently, FPGA's are found in relatively high-volume productions [12, p.1].

Of course, the FPGA itself is a chip, fabricated in a process similar to that of ASIC's. The FPGA, however, is considerably larger than any ASIC it could be used to prototype, resulting in the FPGA's higher per-unit cost. The reconfigurability of the FPGA, and it's ability to manifest multiple designs, is a feature ASIC's simply do not have, by their application-specific nature.

The higher per-unit cost of an FPGA versus an ASIC is worthwhile if the FPGA will host multiple designs, since a distinct ASIC would have been manufactured for each. The increased cost is spread across the wide variety of applications which could be housed in a single FPGA [16]. It also leads to a distinction between *System on a Chip* and *System on a Programmable Chip*.

***FPGA's versus Complex Programmable Logic Devices (CPLD's)***

FPGA's differ fundamentally from CPLD's in their internal architecture. The CPLD is an extension of the PAL concept. Consider multiple cascaded PAL's. This could be extended to two or more dimensions. The CPLD still fundamentally uses the 'sea of logic gates' approach.

FPGA's are larger, more powerful and more expensive than CPLD's. CPLD's are typically used to implement 'glue' logic, while FPGA's are used to implement complete Systems on a Programmable Chip.

### 2.2.2 Technical Review of FPGA's

There are three primary components in an FPGA: the Configurable Logic Block (CLB), the interconnections between the CLB's and the Input/Output Buffers (IOB's). These elements are programmable and reconfigurable. A brief review of these three mechanisms which produce the FPGA's functionality follows.

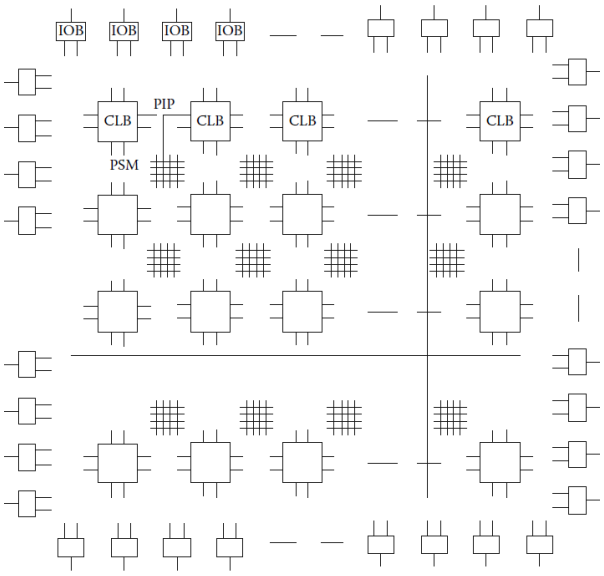


Figure 2.1: Overall structure of an FPGA [16, p.219]

### *Configurable Logic Block (CLB)*

Each CLB is an atomic digital subsystem which receives an input and produces an output. Configurable Logic Block's are responsible for implementing the functionality of the application.

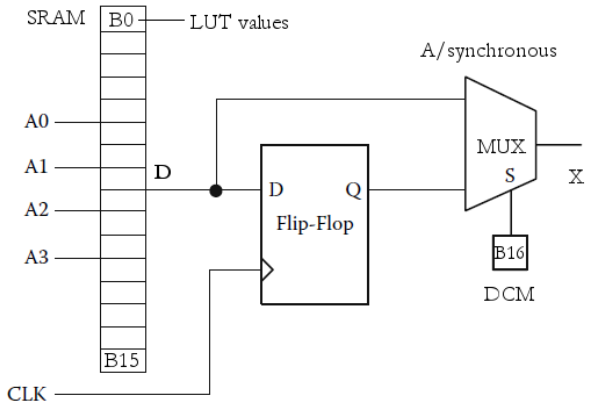


Figure 2.2: Basic structure of a CLB [16, p.220]

The CLB has the following components:

- **Static Random Access Memory (SRAM)** A Look-Up Table (LUT) representing a truth table defining some Boolean function is loaded into the SRAM from the bitstream. A simple LUT could have a 1-bit output, while a realistic one could have a multi-bit digital value.
- A **D Flip-Flop** is used to latch and synchronise the SRAM data output with a global clock signal. This would be a register in the multi-bit case; a register being an array of flip-flops

[15]. Flip-Flops and registers are crucial FPGA resources, allocated separately to LUT's. They can be used to implement BRAM: memory for the Microblaze soft processor.

- **Multiplexer** The multiplexer provides reconfigurable routing. An essential multiplexer function is to allow a synchronous or asynchronous output to be obtained from the FPGA.
- **Distributed configuration memory (DCM)** A smaller portion of total SRAM on an FPGA is used for purposes such as controlling multiplexers, and thereby customising the internal datapath and function of the CLB. A trivial CLB could have 16 bits of Look-Up Table values and 1 bit of routing DCM.
- **Standard logic gates** are also used within the CLB structure. A device such as the Xilinx Virtex has significantly more complex CLB's.

[16, p.220]

The CLB's are loaded with bits from the bitstream synthesised from VHDL.

***Interconnection fabric***

A highly configurable internal communication fabric is characteristic of the FPGA.

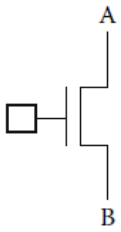


Figure 2.3: Programmable Interconnection Point (PIP)

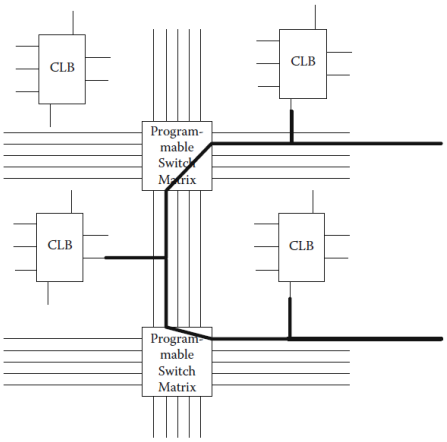


Figure 2.4: Programmable Switch Matrix (PSM)

- A **Programmable Interconnection Point (PIP)** is a reconfigurable analogue to the make-or-break fuse of the early PAL devices.
- The **Programmable Switch Matrix (PSM)** governs the intersections of communication rows and columns.

Each CLB has direct connections to its neighbours. There are also double-length lines which visit every second CLB, quadruple-length which visit every fourth and carefully designed global bus, clock and reset lines which span the whole FPGA. Each node visited along a communication line reduces the speed of communication. It is impossible to effectively use every single logic element of an FPGA, because the communication resources will deplete first [16].

***Input/Output Buffers (IOB's)***

Input/Output Buffers are the FPGA's interface to the outside world. They can function as inputs or outputs, or bidirectionally. This corresponds closely with the `in`, `out` and `inout` port specifications of VHDL.

## 2.3 DIGILENT

Digilent is the Original Equipment Manufacturer for the Nexys2 evaluation board. They produce and distribute a variety of Programmable Logic Devices.

### 2.3.1 Digilent Nexys2-500

The Nexys2 evaluation board has the following features relevant to this project:

- The **Xilinx Spartan-3E FPGA** is the central component on the evaluation board and is used to implement the VAD System on a Programmable Chip.

  The FPGA includes a Flash ROM connected to a JTAG header and a USB2 interface, supporting both cable types for FPGA programming. Bitstreams can be loaded directly into SRAM, or the platform Flash can be configured with an MCS file to load on start-up.
- The **50Mhz quartz oscillator** is available to custom IP cores through the PLB signal `Bus2IP_Clk`. Real-time applications are difficult to implement on a PC platform, since they are removed from low-level hardware by the operating system.
- **16 MB Micron RAM**, tightly integrated into the Microblaze memory map.
- **16 MB Flash memory**, allowing persistent software data storage on the FPGA.
- An **RS-232 port** allows PC connectivity for monitoring and data transfer, and provides the possibility of a MIDI port.
- Four pushbuttons, eight DIP switches, eight LED's and a seven-segment LED display, which can all be controlled using General Purpose Input/Output (GPIO).

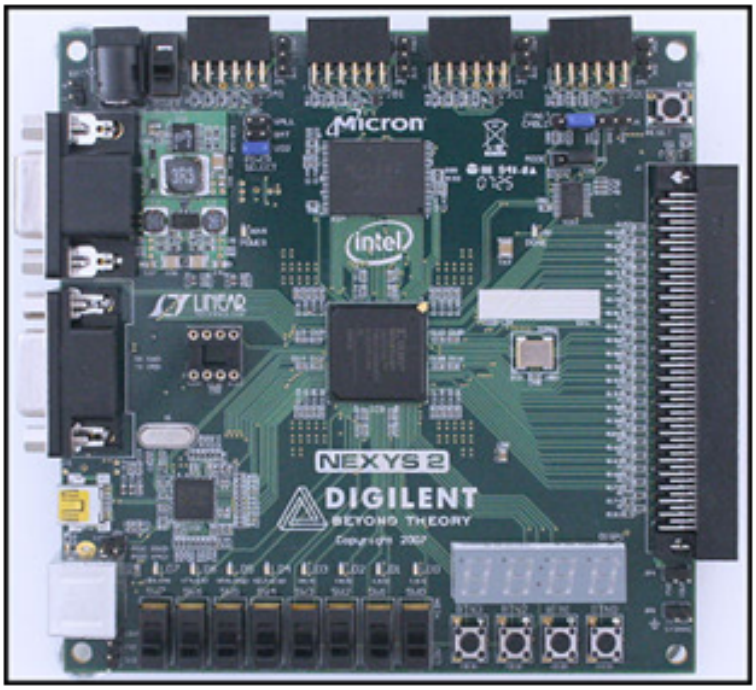


Figure 2.5: Photograph of Nexys2-500 evaluation board.

- Eight 'pmod' extension ports which expose FPGA pins to the outside world and are used to connect external components.
- The Nexys2 can be powered from a USB socket, a wall transformer or a battery.

[6]

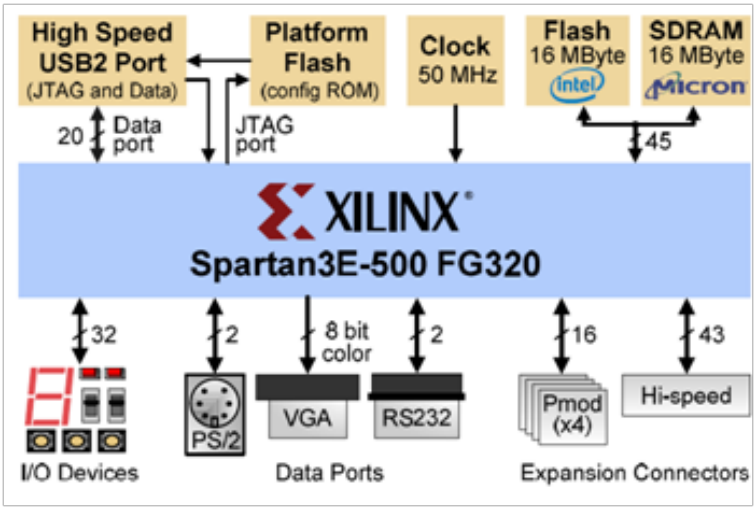


Figure 2.6: Functional block diagram of Nexys2-500 evaluation board [6, p.1].

***Xilinx Board Description (XBD) file***

The XBD file used to configure the Nexys2 evaluation board is available from Digilent's website. This greatly expedites Microblaze system configuration. The provided XBD file has the 16MB Micron RAM configured, but not the 16MB Flash memory [8].

***Digilent Adept***

Digilent provides an inexpensive USB cable which can be used to program the Spartan FPGA, removing the need for an expensive Xilinx Platform Cable. However, the USB cable is incompatible with the programmers in the Xilinx suite, and requires a special tool called *Digilent Adept* to program the FPGA. Adept is available from Digilent's website [7]. An additional driver is available which supposedly improves USB cable compatibility with Xilinx tools [9], but was found not to work with the development PC.

***Digilent Pmods***

Digilent manufacture a range of extension boards for their evaluation boards. These are surface-mount Printed Circuit Boards (PCB's). Each has a datasheet and a reference design for ISE, which must be re-written for EDK. Pmods are inexpensive and purchasable directly from Digilent's website.

Each hardware peripheral typically has an associated custom IP core which is a digital controller housed in the FPGA. This controller connects the physical peripheral to the PLB bus and mediates between the device and the Microblaze processor.

## 2.4 XILINX

Xilinx manufactures the FPGA and development tools that will be used for this project. UCT is engaged in the Xilinx University Programme, which provides kits at academic prices.

Formed in 1984, it is the electronics company credited with inventing the FPGA. Today, they produce a wide range of Programmable Logic Devices (PLD's), development tools, Intellectual Property (IP) and evaluation boards. They claim more than a fifty percent share in the PLD market, with more than 2000 patents [23].

The company produces many different development tools, some with overlapping functionality. Deciding which tools are appropriate for a given device and application is a learning curve in itself. A judicious choice of tools results in an efficient development environment.

Altera is a modern-day competitor of Xilinx in terms of both market share and mind share. Comparisons will be made between Xilinx and Altera products where illustrative, since students must often choose between the two.

### 2.4.1 The ISE Webpack Development Environment

The FPGA development suite is distributed via an internet download called ISE Webpack, containing a licensing system. The download size is around 2GB for Windows or Linux and 3GB

for both.

The Webpack includes free tools, as well as tools which require licensing. ISE Webpack features the ability to select which tools should be installed, as well as providing a brief licensing overview of the selection.

Xilinx's development tools are rich, and though at different levels of maturity, comprise an extensive development environment with different possible workflows and combinations of tools.

ISE Webpack is available for both 32-bit and 64-bit PC architectures. This project uses the 64-bit architecture version, which would be considered more experimental, but potentially improves synthesis wall time. The 12.2 version is discussed here.

It is helpful to know which development tools one requires before commencing installation. This is best achieved with a brief description of the relevant tools.

***ISE***

ISE is the primary VHDL Computer Aided Design tool in the Xilinx development suite. Entities are specified in HDL, then placed on a schematic and graphically connected. Large Systems on a Programmable Chip may be assembled this way. ISE also allows the developer to examine the internals of synthesised entities with Register-Transfer Level (RTL) diagrams. The development, workflow and simulation features of ISE are the most mature of the suite. [28]

ISE is analogous to Altera's Quartus2 software. Both are free.

Many tools in the suite can interface with ISE. To this end, ISE allows artefacts from different tools in the suite to co-exist in a single project, if desired. Questions related to importing these heterogenous file types into ISE are common.

The company is recently promoting a workflow where ISE is the top-level project in embedded designs [25, p.29]. This was found to be unnecessary and in fact undesirable for the project at hand, since EDK can perform these tasks in a single, integrated environment.

***Embedded Development Kit (EDK)***

This project uses only the Embedded Development Kit, a licensed tool which is available through the Xilinx University Programme. An evaluation license is included with ISE Webpack.

EDK is the development tool for the Microblaze soft core processor. It is synonymous with Xilinx Platform Studio (XPS). EDK is an integrated platform for defining the Microblaze system, assembling the various peripherals and busses, configuring the memory map and connecting custom ports.

EDK features VHDL peripheral development, C/C++ software development, Base System Builder SOPC generation, System Assembly View SOPC configuration and bitstream generation.

[25]

The current version of EDK has limited simulation support for Spartan3E FPGA. Some components of the EDK warn of pending deprecation, but were found to work well, despite limitations.

***Microblaze Soft Processor Core***

The Microblaze is a 32-bit soft core processor for Xilinx FPGA's. It is a Reduced Instruction Set Computer (RISC) with a Harvard architecture, meaning separate memory is used for instructions and data. The FPGA and the processor manifested within it are clocked at the same frequency.

A soft core processor is useful for I/O and control purposes. It exists entirely within the logic elements of the FPGA itself. It can interface with the outside world through the IOB pins of the FPGA using a GPIO or through other pcore peripherals.[16, p.223]

The recommended way to create a Microblaze system is with Base System Builder.[25, p.11]

***Base System Builder (BSB)***

Base System Builder allows rapid development of Microblaze systems. The evaulation board in use is configured using the supplied Xilinx Board Description (XBD) file. The XBD file is installed to the `$XILINX_EDK/board` subdirectory. The board then appears for selection in BSB, expediting configuration [25, p.16].

The following decisions can be made regarding the Microblaze system. These configure the underlying hardware platform, and represent design decisions which will have to be made while designing the VAD prototype.

- **Amount of BRAM dedicated to data and instruction memory.** The instructions constituting the software to be run on the Microblaze are stored in BRAM on the FPGA. The BRAM is instantiated as a Xilinx BRAM IP core, which is dual-ported, meaning only one IP is required for both data and instruction memory. Two memory controllers are created, one per port.

  There is a fairly strict constraint on the amount of BRAM available for this purpose, and consequently on the size of software applications which can be stored entirely in BRAM. This is around 16 KB for the Nexys2-500. 8 KB is the default.

- **Configuration of the RAM controller IP.** In the case of the Nexys2 evaluation board, the Micron RAM is an external microchip which must be interfaced with through the Input/Output Buffers of the FPGA. The use of an appropriate Xilinx Board Definition file should map the external ports of the memory controller IP to the pins of the FPGA which control the RAM.

- **Configuration of UART IP.** The Universal Asynchronous Receiver/Transmitter (UART) is a digital logic controller for RS-232 communication. It maintains the baud rate and signal format associated with the protocol.

EDK features two UART IP's: Uart Lite and Uart 1640/1650. The Uart Lite requires no software configuration, but only 'standard' baud rates are available and the baud rate is synthesised directly into the IP core. The 1640/1650 is based on a popular, eponymous UART chip. It features software configurability and a FIFO for reliable asynchronous operation. The VAD prototype uses the 1650.

Again, the XBD file specifies an entry in the UCF file connecting the UART to the external MAX232-equivalent chip via the appropriate pins of the FPGA. Additionally, an entry is created in `xparameters.h` specifying the serial port as POSIX-style `stdin` and `stdout`. This is the base address of the UART in the memory map.

- **Instruction and/or data cache** Caching data in native structures on the FPGA greatly improves software application performance, at the expense of logic elements. Accessing BRAM is slow, as is accessing the Micron RAM, and both data and instructions are stored in memory. A system using cache is more likely to meet synchronicity requirements. Cache is recommended and selected by default.
- A **floating-point unit** can be included, if desired. This uses a large number of logic elements and constrains the number of peripherals comprising the system. This project does not use a floating-point unit.
- A **dual-core SOPC** can be created, if desired.

[25, p.11-16]

BSB presents a summary of the system before creating the files which define the SOPC. The LED's, switches, pushbuttons and seven segment display will also be configured.

Base System Builder is analogous to Altera's SOPC Builder. SOPC builder is free, while BSB is a part of EDK, which is licensed.

***System Assembly View***

IP cores are essentially a high-level VHDL entities conforming to Microblaze bus specifications. They augment system functionality.

System Assembly View allows the developer to reconfigure the SOPC by adding, removing and configuring peripheral IP cores and their connections. The IP cores are added from the *IP catalogue*, and are accompanied by PDF datasheets, which are accessible from the GUI. A Javadoc-style documentation for the peripheral drivers is also available.

Three important items must be configured for each new IP: its bus connections, memory map and port map.

**Bus connections**

Most peripherals need to be connected to the global Microblaze bus lines. This is usually done by connecting the Slave PLB (SPLB) bus port of a new peripheral to the Master PLB (MPLB) controller of the Microblaze by clicking on the bus diagram.

The two immediate bus types are Local Memory Bus (LMB) for the BRAM and controllers and Processor Local Bus (PLB) for peripherals. A graphical legend is presented towards the bottom of System Assembly View. A warning is issued in the console view if a peripheral does not appear to be connected to the Microblaze via a bus.

**Memory map**

The PLB bus is available to the Microblaze software through the system's memory map. A memory size and base address must be specified for new, unmapped peripherals. Otherwise, a warning will be issued.

The memory map can be automatically generated by clicking the 'Generate' button, but it is useful to manually group different instances of the same peripheral in the memory map.

Memory allocations of different peripherals cannot overlap, or a warning will be issued.

**Port map**

The peripheral interconnections corresponding to VHDL ports of the peripherals in the system can be configured from the port map tab. It is important that signals between peripherals are consistently named in this view. Otherwise, the system will not function correctly.

There are two types of port connection: connections amongst peripherals and external connections to the IOB pins of the FPGA.

[25, p.21], [14]

### *User Constraints File (UCF)*

External ports are mapped to IOB pins of the FPGA in the User Constraints File (UCF), which specifies constraints to the netlist. `system.ucf` is available from the 'Project' view and resides in the `data` project subdirectory.

A typical UCF entry looks as follows:

```
...
Net fpga_0_LEDs_8Bit_GPIO_IO_O_pin<0> LOC=R4  |  IOSTANDARD = LVCMOS33;
...
```

[5]

ISE also has a concept of a UCF, and in fact the same UCF can be used in ISE. This workflow is not used here.

### *Platform Generator (Platgen)*

Platform Generator essentially produces the VHDL source code which represents the SOPC designed with BSB and System Assembly View [25, p.28].

### *Xilinx Synthesis Toolkit (XST)*

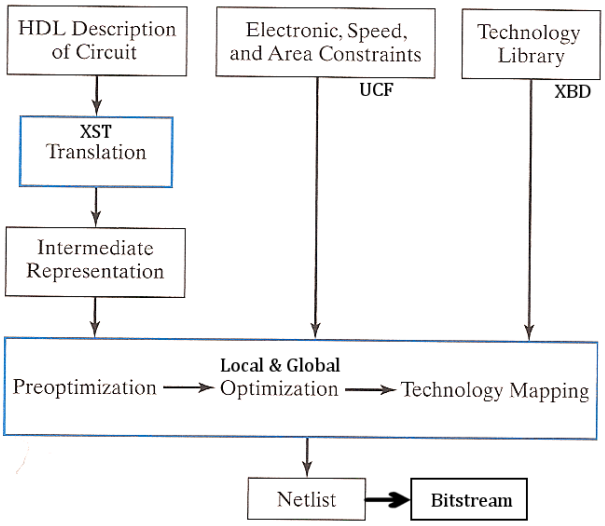


Figure 2.7: VHDL synthesis process [15, p.175]

XST the VHDL synthesiser for ISE Webpack. It generates intermediate-form structural descriptions from the VHDL source.

### *Place And Route (PAR)*

Place and route performs several iterations of local and global optimisation. The output is a netlist file which contains the FPGA technology mapping.

### *Bitstream Generator (Bitgen)*

Bitgen produces a `system.bit` FPGA hardware configuration file from the netlist, which is ultimately loaded into FPGA SRAM. This corresponds to the 'Generate Bitstream' command in the Hardware menu. At this stage, a hardware-only system can be loaded onto the FPGA. `system.bit` is located in the `implementation` project subdirectory. This file is loaded onto the FPGA with Digilent Adept.

### *Application view*

The application view is used to develop software which can be loaded into the instruction BRAM of the Microblaze. Sample BRAM and RAM test applications are provided. They produce output to the RS232 port at 9600-N-8 by default.

The BRAM test application, `TestApp_Memory_microblaze_0`, is the software template for this project. Various applications may be developed, each occupying its own eponymous

subdirectory. The application to be loaded into BRAM is indicated by right-clicking the application and selecting 'Init BRAM'. [14]

***Compiler and Linker configuration***

The compiler dialog allows the specification of either `mb-gcc` or `mb-g++` as the software compiler, providing both the C and C++ languages for software development. `ld` is the linker tool.

The linker dialog allows the allocation of various types of variables to either BRAM or RAM. The relevant ones are

- `.text` Instruction memory. This must be BRAM; otherwise, a boot-loader is required.
- `.bss` Free, global variables. This can be allocated to Micron RAM to allow large data buffers.
- `.stack` Automatic variables created at the beginning of functions.
- `.heap` Pointer variables allocated using `malloc()`. These are not used in this project, because the library occupies too much instruction memory.

These tools are based on the GNU Compiler Collection (GCC) and support cross-platform compilation. The C language is used here, because it offers just enough abstraction without leading to software bloat. There is a limited amount of instruction BRAM available.

These dialogs warn of depracation to the Eclipse-based Software Development Kit (SDK). However, SDK was found to be unstable on the development PC, and unnecessary for this project.

***Library Generator (Libgen)***

Libgen effectively generates the software platform for the SOPC, based on the given configuration. The standalone operating system and drivers are generated for the IP cores in use. Driver header files must be manually included into the software application. Libgen also generates the standard libraries supported by Microblaze. [26, p.99-107]

The output of the software compilation process is an Executable/Linkable File (ELF).

***ELF to Data tool***

The final step in the software compilation process is to occupy the BRAM of the Microblaze SOPC with the software ELF file. A tool called Data to Mem merges the software ELF with the `system.bit` bitstream to produce `download.bit`, the ultimate synthesis artefact, containing both software and hardware reconfiguration [26, p.203-204]. `download.bit` is located in the `implementation` project subdirectory. This file is loaded onto the FPGA with Adept.

***Impact and MCS file generation***

The Impact tool can produce an FLASH configuration file from `download.bit` which will program the FPGA SRAM on start-up. This file is loaded onto the FPGA with Adept. [10]

***Create/Import Peripheral (CIP) Wizard***

The CIP wizard is used to develop custom VHDL peripherals for the Microblaze SOPC. These peripherals connect to the Microblaze soft core processor either through the Processor Local Bus (PLB) or with Fast Simplex Link (FSL). PLB is slow compared to FSL, but more versatile, as it allows bidirectional communication and a very useful slave register model. FSL would be used to develop a dedicated coprocessor. This project uses PLB version 4.6.

CIP Wizard is a tool for generating PLB template peripherals. The 32-bit slave registers are accessible through the memory map and through C macro's defined in a generated driver file. The name of the pcore and the number of slave registers can be selected. Two slave registers are created by default. They operate independently in terms of input and output. There are other templates available, such as a direct memory-map model, but these were not used here.

Two VHDL files are generated for each custom peripheral: `peripheral_name.vhd` and `user_logic.vhd`. The former is a wrapper used to define the PLB bus interface and custom external ports and generics, while the latter is used to define the functionality of the custom peripheral. These are available in the `pcores` project subdirectory and will have to be opened manually.

Additionally, a C driver header file is generated in the `drivers` subdirectory. In this file, the obsolete `xio_in_32()` macro must be replaced with the newer `XIo_In32()`, and `xio_out_32()` replaced with `XIo_Out32()`. These are the basic memory functions. The header file must then be included into the software application by double-clicking 'Headers' in the application view and adding the file, and using the `#include "peripheral_name.h"` macro.

Custom external ports and generics are tightly integrated with System Assembly View, facilitating rapid reconfiguration. It is possible to include code from third-party libraries, using the `work.` base library. CIP wizard has an obscure workflow in these situations. First, the peripheral should be created, using the 'New peripheral' option. Then the structural adjustments regarding external ports, generics and libraries should be made. Finally, the peripheral should be re-imported using the 'Import existing peripheral...' tool. This is done by locating the Peripheral Order Analysis (POA) file for the pcore, located in the `data` subdirectory. Only then will the correct Microblaze Peripheral Description (MPD) file be generated, allowing the peripheral to be properly configured. When re-importing, the previous peripheral version will be backed-up and overridden. [27, p.53-64]

Here is an example of adding a custom port. In `peripheral_name.vhd`:

```
...

-- ADD USER PORTS BELOW THIS LINE ------------------
```

```
--USER ports added here
 AUXILIARY_PORT : out std_logic_vector(0 to 9);
-- ADD USER PORTS ABOVE THIS LINE ------------------

...

-- MAP USER PORTS BELOW THIS LINE ------------------
--USER ports mapped here
AUXILIARY_PORT => AUXILIARY_PORT,
-- MAP USER PORTS ABOVE THIS LINE ------------------

...
```

This exposes the port to the outside world and maps it internally to a port to be defined in `user_logic.vhd`:

```
...

--USER ports added here
AUXILIARY_PORT : out std_logic_vector(0 to 9);
-- ADD USER PORTS ABOVE THIS LINE ------------------

...
```

Here is an example of using `numeric_std` and a custom library in a peripheral. The library must be in the same directory as the user logic file. The CIP wizard should detect the library dependency during re-importation. [24]

```
-- DO NOT EDIT BELOW THIS LINE --------------------
library ieee;
use ieee.std_logic_1164.all;
--use ieee.std_logic_arith.all;
--use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library proc_common_v3_00_a;
use proc_common_v3_00_a.proc_common_pkg.all;

-- custom library
library work;
use work.library_name.all;
```

-- DO NOT EDIT ABOVE THIS LINE ----------

CHAPTER 3

# METHODOLOGY

## 3.1 STEPS TO BE TAKEN

The following process will be used to complete this thesis.

- Design the VAD prototype system, consisting of hardware and software components and their interfaces.
- Identify and explain the essential performance characteristics and parameters of the system and design decisions.
- Find the system clock counts corresponding to the various frequencies in the system.
- Design tests to be conducted in support of the stated hypotheses. These verify the functionality of the system.
- Design the various reconfigurations of the system which will be used in support of the tests.
- Conduct the experiments and record the results.
- Analyse results and observations of the experiments.
- Prove or disprove the hypotheses based on the conclusions of the analyses. These validate the functionality of the VAD system.
- Illustrate how the VAD prototype system can be expanded to a fully-fledged audio framework for the Nexys2 evaluation board.

## 3.2 EXPERIMENT DESIGN

### 3.2.1 Apparatus

The following equipment is required to conduct the experiments.

- Nexys2 VAD prototype, as described in Prototype Design. This is the system under test.
- Tektronix TDS 210 digital oscilloscope. It has two 60 MHz channels with trigger function and frequency detection.
- Matlab 2008. MATLAB is used to format data for the system and to analyse results.
- Cubase SX 2. This is an audio editing application.
- M-Audio Delta 44 sound box. This 'breakout' box provides a high-quality interface to the PC.
- A PC is necessary to perform these experiments.
- Miscellaneous cables: RS-232, USB and RCA cables and adapters.
- An iPod sound source with test waveforms.

### 3.2.2 Experiments

***Experiment 0: DAC SPI verification***

First the DAC and ADC functionality must be verified. The SPI signals are inspected on the digital oscilloscope and recorded. The results are compared with the given timing diagrams. A sample waveform is generated. Further analysis is conducted.

This experiment is in support of Hypothesis 2.

***Experiment 1: Straight-through monitoring***

A VAD system which connects the line input directly to the output is created. An iPod sound source connects directly to the M-Audio sound box, and also through the VAD system to the M-Audio box. These recordings need not happen simultaneously. It is assumed the affect of the recording device is negligible and that the iPod plays the sample identically each time.

The DC shift is removed from the VAD signal by the sound box. The samples are aligned on a sample-by-sample basis in time and cropped to the same length.

The input and output waveforms are then opened in Matlab. The first twenty seconds are compared in the time and frequency domains, using the discrete and continuous Fourier transforms. The system is characterised as a $H(e^{j\omega})$ linear system using fundamental Digital Signal Processing (DSP) theory.

***Experiment 2: Line-in recording and playback***

The device is used to record an audio signal. The signal is then played back and recorded onto the PC. This signal is compared with the original in the time and frequency domains. The VAD recording subsystem is characterised as a $H(e^{j\omega})$ linear system using DSP theory.

This experiment is in support of Hypothesis 0.

***Experiment 3: Monophonic synthesis***

The minimum and maximum frequencies which the sine oscillator produces are found using a digital oscilloscope. The various waveforms are inspected.

This is in support of Hypothesis 1.

***Experiment 4: Polyphonic synthesis***

Four sine oscillators are used to generate discrete frequencies. The output from the device is recorded onto the PC. The data is analysed with Matlab.

This is in support of Hypotheses 3 and 4.

***Experiment 5: Prototype system demonstration***

This final experiment will demonstrate how the prototype can perform all these functions concurrently, due to the nature of its design.

An ultimate prototype system is presented, of which the other reconfigurations are subsets. The success in reconfiguring the system for these various experiments will determine whether the VAD system is indeed highly reconfigurable.

The system will play the synthesisers using a MIDI file while monitoring the line and microphone inputs to the speaker, headphone and line output. At the same time, it will play samples from the sampler which will be configured as a drum machine. Finally, the inputs will be recorded, and when recording is complete, the recording will be played back and looped.

The MIDI song will play with four sine oscillators, two saw oscillators, a triangle oscillator, a square oscillator and a drum machine. The song should be easily recognisable when demonstrated. It should be straight-forward and easy to convert any MIDI file to be playable on the device.

The MIDI files to be used for this experiment are the Mario soundtrack, The Simpsons theme, The Flintstones Theme and Star Wars Theme.

This experiment is in support of Hypotheses 3, 4 and 5.

CHAPTER 4

# PROTOTYPE DESIGN

## 4.1 BILL OF PARTS

Digilent provide a range of pmod extension boards for the Nexys2. These surface-mounted PCB's are compact and convenient. The reference designs which Digilent provides are for Xilinx ISE and so not applicable, except for learning purposes. The DAC used is from the UCT GT16 kit.

| Component | Description | Estimated Cost | Supplier |
|---|---|---|---|
| Nexys2-500 | FGPA evaluation board | R800 | Digilent |
| PModAmp1 | Stero amplifier extension board | R160 | ” |
| PModMic | Microphone and ADC extension board | R120 | ” |
| PModAD1 | Two 12-bit ADC extension board | R200 | ” |
| MCP4921 | 12-bit DAC with SPI interface | R35 | Mantech |
| LM324 | 4x Non-inverting op-amp with single supply | R1 | Microsource |

Table 4.1: Parts used to construct VAD prototype

Total estimated prototype cost: **R1 316**

## 4.2 INTENDED END APPLICATIONS

The VAD system should be reconfigurable to realise the following user applications.

- **Dictaphone** The user records audio on the device and plays it back later through a built-in speaker.
- **Portable music player** The user loads songs onto the device and plays them back through stereo headphones
- **iPod docking station** The iPod connects to the line input with a headphone jack. The audio is heard through a built-in speaker.

- **Kareoke machine** The user loads MIDI or audio files onto the device. The user plays the MIDI file while singing through a built-in microphone. The user's singing is also audible through the output, which is an RCA line-out. The performance can be recorded.
- **Sampler/Drum Machine** The user loads short audio clips onto the device and plays them back with buttons.
- **Mixer** An M-input, N-output audio system.
- **Embedded audio** A third-party embeds VAD in their device as the audio subsystem. This could be used for alarm systems, telephones, doorbells, etc.

## 4.3 HARDWARE/SOFTWARE PARTITION

The hardware/software partition is the division of functionality between hardware and software components.

The hardware components of the prototype are implemented as VHDL custom IP cores in EDK. The software components are implemented as C and executed on the Microblaze soft core. The Processor Local Bus (PLB) version 4.6 forms the hardware/software interface.

Data transfer between the processor and peripherals is initiated by the processor, creating a master/slave relationship between the hardware and software. The software can write to peripheral slave registers using driver functions generated by CIP wizard. Writing to the PLB bus is relatively slow, therefore the slave registers are used primarily for peripheral control purposes.

By building the VAD peripherals in hardware, the Chuck audio programming language concept of 'strong-timing' is satisfied: individual audio signals will be processed on a sample-by-sample basis, in lock step with all other audio samples in the system. Where audio samples must cross the hardware/software partition, an asynchronous buffer is used.

The above factors designate the software as the control unit and the hardware as the datapath, to draw an analogy with general-purpose processors. Interrupts on the Microblaze are disabled, since polling was found to be adequate, so the hardware cannot invoke action on behalf of the software.

## 4.4 AUDIO PERIPHERAL DESIGN

### 4.4.1 Analysis of PLB Peripherals

PLB peripherals, as defined in `user_logic.vhd` by CIP wizard, are largely monolithic. A primary synchronous process is responsible for managing slave register writes. There is a smaller asynchronous process for reading from the slave registers.

Both read and write logic are included for each slave register, though both are not necessarily needed and either or both may be commented out, conserving logic resources on the FPGA.

The slave register template is a flexible starting point for most Microblaze peripherals.

### 4.4.2 Inserting User Logic

Most functionality is implemented in the synchronous slave register write process, due to the fact that the slave registers should only be written to in a single process; otherwise, synchronicity issues will result. User logic will usually require writing to the slave registers at some point.

### 4.4.3 PLB Peripheral Functionality

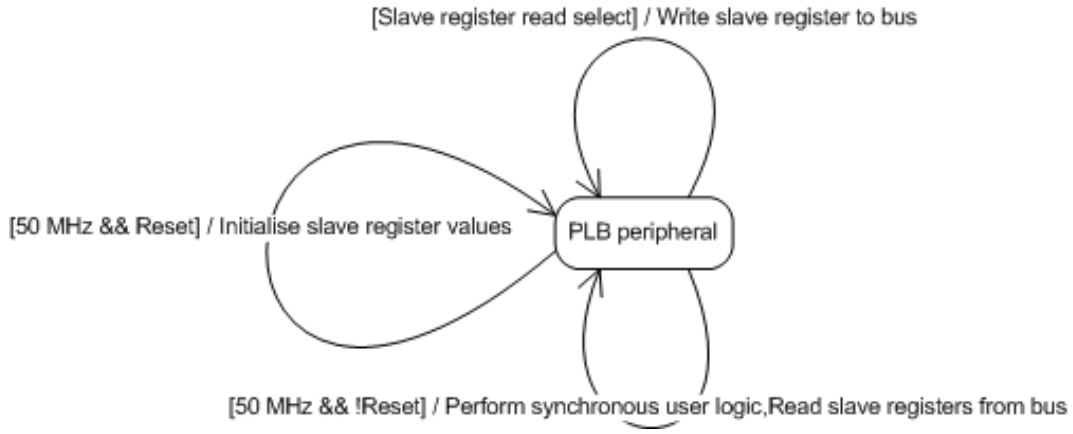


Figure 4.1: Statechart of PLB Peripheral Template with Slave Registers. The peripheral is stateless and transitive by default. It has a synchronous reset.

### 4.4.4 Multiple Timing Constraints

In the VAD system, a single peripheral is required to perform a variety of tasks at different frequency intervals. These actions can only occur synchronously within a monolithic process. This is achieved by dividing the main synchronous register write process into subprocesses .

The main register write process is clocked from the 50 MHz `Bus2IP_Clk` signal. The 'subprocesses' count system clock cycles to operate at a given frequency in real-time. Each count represents an additional delay of $(50MHz)^{-1} = 20\ ns$.

When the frequency interval is constant, a `mod` operation can be used to achieve this clock division. When the frequency is adjustable, as in the case of the oscillators, a slightly different construct is used.

## 4.5 AUDIO PIPELINE

### 4.5.1 Function of audio pipeline

The audio pipeline is responsible for transferring data from one audio peripheral in the VAD SOPC to another.

Dedicated audio communication lines free the soft processor from having to continually move data around the system, and allow multiple audio samples to be transferred in a single clock cycle. Once the pipeline is full, an output sample is produced every cycle [15, p. 544 - 553].

### 4.5.2 Properties of audio pipeline

***Bus width***

The bit-width for the pipeline must be selected. Since 12-bit DAC's and ADC's are employed, a 12-bit audio pipeline will be used. This gives a numerical range from 0 to 4096. The median value of 2048 should be output when a device is idle. Audio data placed on the pipeline must be confined to this range. Values larger than 4096 will be truncated and digital *clipping* will occur.

***Bus reconfiguration***

The audio pipeline is highly reconfigurable through System Assembly View. Devices may be connected arbitarily, from input to output. One output to many input connections are possible. New devices which interface with the audio pipeline can easily be created.

***Clock speed***

Pipeline speed is typically constrained by the slowest computational element in the pipeline. In this case, time must be allowed for audio samples to be serialised to the DAC's and ADC's in the system. Most other audio peripheral operations take a single cycle.

This $SCLK$ frequency affects the maximum sampling rate for audio in the system and the frequency of the fastest waveform which can be generated by the oscillators.

The 50 MHz FPGA clock is divided by the empirical value of 40 to produce the audio pipeline clock frequency.

### 4.5.3 Pipeline implementation

A peripheral connecting to the audio bus is called an audio peripheral and has a 12-bit `AUDIO_IN` input and/or a 12-bit `AUDIO_OUT` output ports. The synchronicity of these ports is illustrated below.

***Custom ports***

Similarly to Fast Simplex Link (FSL), the audio pipeline is unidirectional. Each device may have two audio ports: an audio input and an audio output. Sound sources, such as oscillators, ADC's and sound output buffers, have only an AUDIO_OUT. Sound sinks, such as DAC's and

sound input buffers, have only an AUDIO_IN. Other devices in the pipeline have both ports. These are external custom IP ports and are visible in the port map of System Assembly View.

In this way, the hybrid design of PLB peripherals using a custom pipeline provides the best of both the PLB and FSL bus technologies.

***Timing mechanism***

The audio pipeline is implemented as an audio subprocess. The following VHDL code demonstrates how the PLB bus clock is divided by 40 to achieve an audio pipeline data rate of 1.25 [MHz]. A modulo divider is used.

```
-- AUDIO subprocess
audio_clk := (audio_clk+1) mod 40;
if audio_clk = 0 then
    -- write to AUDIO_OUT
elsif audio_clk = 1 then
    -- read from AUDIO_IN
end if;
```

Each device writes a new value to its output, then reads a new value to its input. This two-phase mechanism ensures that signals are not contested by input and output devices and is analogous to a device operating on both the falling edge and rising edge of a clock.

Both branches are not needed if the device is input- or output-only. This again would preserve logic registers.

The data rate of a single audio path through the system is then

$$1.25MHz \times 12bits = 15Mbps$$

This datarate results from the parallel bandwidth offered by the 12-bit pipeline.

Thus each peripheral maintains its own independent count of when the next audio transfer should occur. In a software situation, this could be problematic, but in the hardware domain, synchronicity is given through a common clock line.

It is impractical to introduce additional clock lines to PLB peripherals, because these processes will be difficult to synchronise with the slave register write process.

### 4.5.4 Future work

More bits could be used for the audio pipeline to increase internal numerical accuracy. The larger the range of values, the smaller percentage error an integer rounding produces. For example, 24-bit audio samples could be used rather than 8-bit. The samples would be scaled down to 12-bits at the output using a bit shift by a ‘bitcrusher’ entity.

The bit width is commonly referred to as 'head room', since the greater the bit width, the larger the sample values which can be recorded without producing clipping. Larger sample values correspond to louder sounds, since they indicate extreme position of the speaker or microphone membranes.

The audio pipeline speed could be specified through generics. All peripherals would require the same speed to function.

## 4.6 DIGITAL TO ANALOGUE CONVERTER (DAC)

### 4.6.1 Audio output subsystem

***Function of output subsystem***

The audio output subsystem produces acoustic energy by vibrating air with the membrane of a speaker. This is perceived as sound.

A digital value is output from the Nexys2 into an external DAC component. The DAC output voltage is fed into an amplifier. The amplifier produces a corresponding current which magnetises ('drives') the coils of a speaker, causing the speaker membrane to move to a position which is approximately proportional to the input voltage, and hence the initial digital value.

In this way, values from the system are used to create sound.

***Components***

The analogue output subsystem comprises the following components:

- The **DAC controller IP** loads the DAC with samples from the audio pipeline via an SPI interface.
- The **DAC** itself is connected to the DAC controller through a pmod connector.
- The Digilent **PModAmp amplifier** is a stereo extension board with a pmod interface. It has parallel speaker and headphone outputs and a volume control [4].
- A small **speaker** mounted in a plastic enclosure connects to the speaker output of the PmodAmp.
- Consumer **headphones** can be connected via the headphone jack.

### 4.6.2 DAC

### 4.6.3 Function of DAC

The DAC transposes data from the digital electronics domain into the analogue electronics domain. A digital value originating from a computer system is translated to a corresponding voltage level.

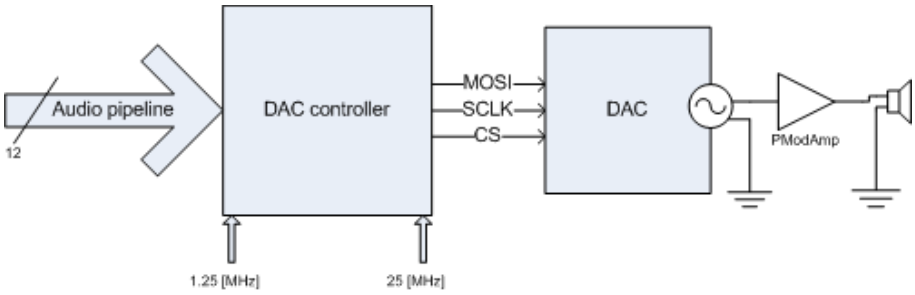


Figure 4.2: VAD audio output subsystem

For the 3V Nexys2, the DAC output ranges from 0 to 3.3V.

$$V_{OUT} = \frac{V_{REF} G D_N}{2^n}$$

Figure 4.3: DAC ideal transfer function equation [17, p. 16]

$V_{ref}$ is a parameter specifying the maximum output voltage, which is $3.3V$. $n$ is the number of bits of the DAC. $\frac{V_{ref}}{n}$ gives the voltage step size of the DAC. $G$ is the gain, which is one. $D_N$ is the digital input value.

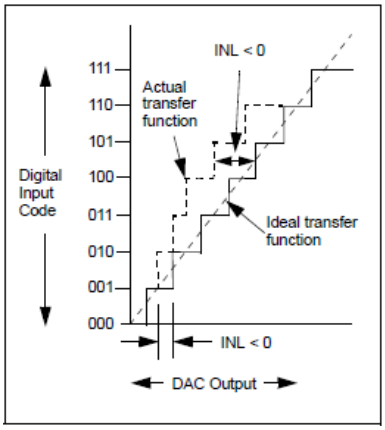


Figure 4.4: DAC transfer function graph [17, p. 16]

### *DAC selection*

The Nexys2 has no built-in DAC. Since the DAC is an analogue device, it must be provided as an external component, connected to the FPGA via the pmod interface.

Digilent provides a pmod DAC called PModDA2. For economic reasons, the MCP4921 DIP from UCT's GT16 kit will be used instead.

### 4.6.4 Serial/Parallel Interface (SPI)

SPI is the communication interface characteristic of DAC's and ADC's. Rather than transferring digital values to the external chips in parallel, requiring 12 data lines, data is transferred using only three lines: clock, chip select and data. This reduction in data lines necessitates a more complex controller.

The MCP4921 DIP in particular has the following SPI pins.

- **Master Out Slave In (MOSI)**, also referred to as data input or $V_{DD}$ in the pinout diagram below, is the serialised 12-bit value to be placed on the output line, preceded by 4 bits of control information.
- **Serial clock (SCLK)**, SCK, is used to latch bits into the SPI register on the MCP.
- **Chip Select** ($\overline{CS}$), also known as $\overline{SYNC}$, is used to indicate to the chip that a new value is being sent.
- $\overline{LDAC}$ is used to synchronise the outputs of multiple DAC's. It can be wired low for a single DAC. This pin is wired to ground in the prototype. [17, p.20].
- The voltage output range is 0 to $V_{ref}$. $V_{ref}$ is tied to 3.3V [17, p.15].

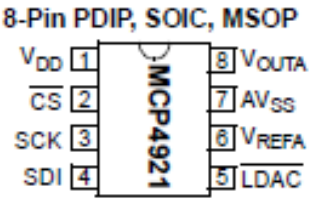


Figure 4.5: MCP4921 pinout diagram [17, p.1]

***Synchronicity***

The MCP4921 is designed for a maximum serial clock frequency of 20 MHz. However, this is not tested during production. It is clocked at 25 MHz in the prototype. The results of this will be monitored during the experiments.

***DAC value format***

The 12-bit DAC accepts 16 bits of input, the most significant four of those being a control nybble. The control nybble is concatenated to the current audio pipeline value as a hardware constant. The control nybble in this case is '0011'.

### 4.6.5 DAC controller IP

***Selection of custom IP core over XPS SPI IP core***

EDK offers a peripheral for communicating with SPI peripherals, using a software driver. [11]. There are a few disadvantages to this approach.

- The XPS SPI relies on software control and polling or interrupts. This is an unnecessary burden on the soft processor which should be moved to the hardware domain.

- The software and hardware components of this pcore are large.

For these reasons, a custom SPI controller is used instead.

***Function of DAC controller IP***

The following timing diagram illustrates how values are loaded into the DAC.

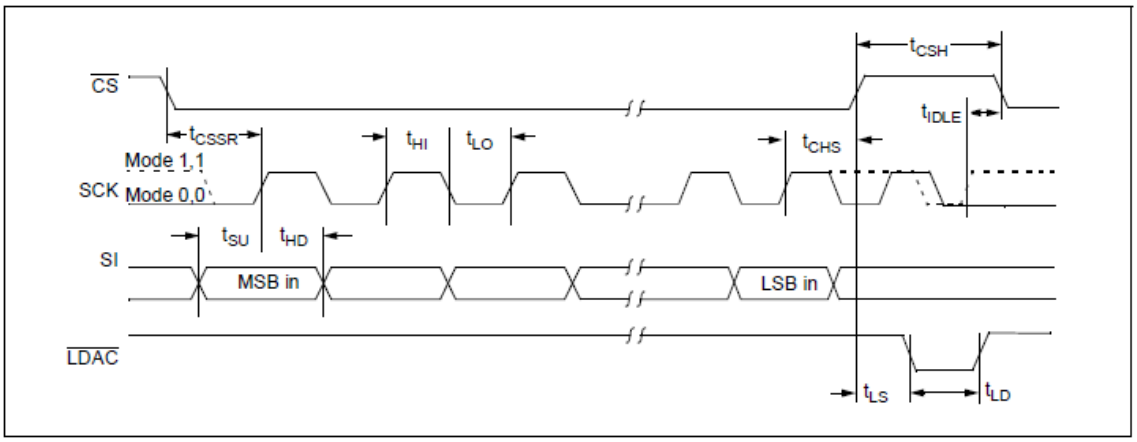


Figure 4.6: MCP4921 SPI Timing Diagram [17, p.6]

***SPI rotation register***

The Most Significant Bit (MSB) of the register containing the 16-bit value to be serialised is connected to the MOSI output. The register is then repeatedly rotated in synchronicity with the SCLK signal. This effectively serialises the register value. [5]

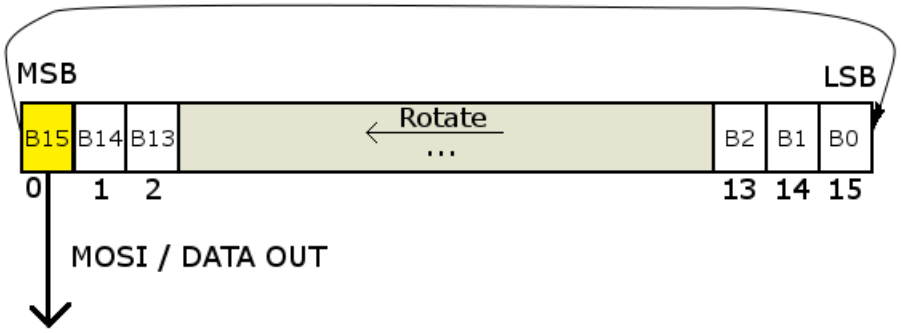


Figure 4.7: Conceptual Diagram of 16-bit Big-Endian SPI Rotation Register. The DAC literature specifies the MSB as B15, and it would be at index 0 in a big-endian VHDL register.

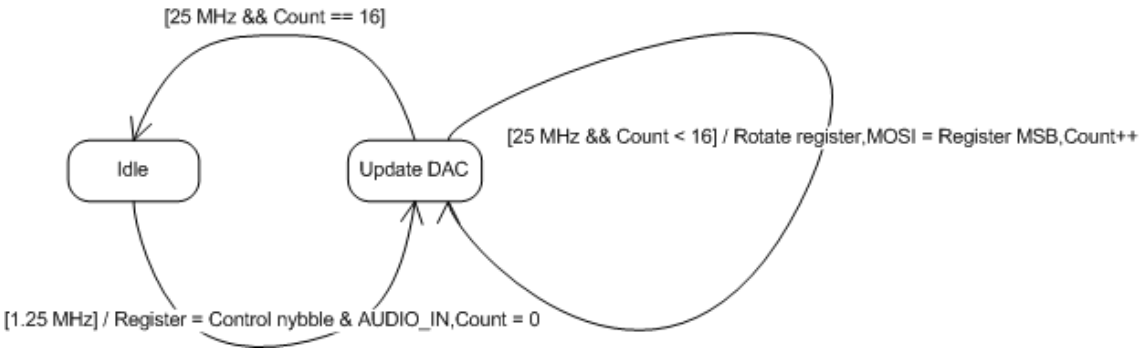


Figure 4.8: Statechart of DAC controller IP

### 4.6.6 Implementation of DAC controller IP

***Synchronicity***

The DAC controller performs two synchronous functions at different frequency intervals. The first function is to latch values from the audio pipeline. This occurs at the frequency of the audio pipeline, which is 1.25 MHz. This value must then be serialised to the DAC, which is clocked at 25 MHz.

The DAC must be clocked fast enough to fully serialise the current audio pipeline value before the next value arrives. A quick calculation verifies this is the case.

$$\frac{1}{25MHz} \times 16\ bits = 0.64\mu s < \frac{1}{1.25MHz} = 0.8\mu s$$

The DAC controller does not require slave register I/O.

### 4.6.7 Future work

The DAC serial clock rate could be made adjustable through a generic.

***Controllers for Multiple DAC's***

For the prototype system, one DAC IP must be created for each DAC chip.

A single DAC controller could control multiple DAC's. For example, a DAC controller could have two data channels, one for each DAC, and common $\overline{CS}$ and $SCLK$ lines [4]. A single 32-bit slave register could be used for both 16-bit values.

## 4.7 Analogue to Digital Converter (ADC) Controller IP

### 4.7.1 Audio Input Subsystem

The input subsystem is responsible for capturing analogue audio data onto the device. The components of the subsystem are listed below.

- The **PModMic** extension board features a microphone, compressor and ADC.

  The microphone is a vibrating membrane and is nearly identical to a speaker, but transforming kinetic energy into electrical energy instead of vice-versa.

  The compressor, or dynamics controller, is used to reduce the dynamism in a signal. Soft parts are made louder, and loud parts softer. This typically improves the audibility of the signal.

  The **ADCS7476** analogue-to-digital converter chip is the principle component in the subsystem and converts the audio signal from the analogue domain to the digital domain by means of successive approximation [19].

- The **PModAD1** extension board features two ADCS7476 ADC's. A **PModCon4** connector provides the physical RCA interface; this could be replaced with an ordinary RCA connector for economic reasons

  The single-supply non-inverting **LM324** operational amplifier is used to create a level shifter for the line-level input.

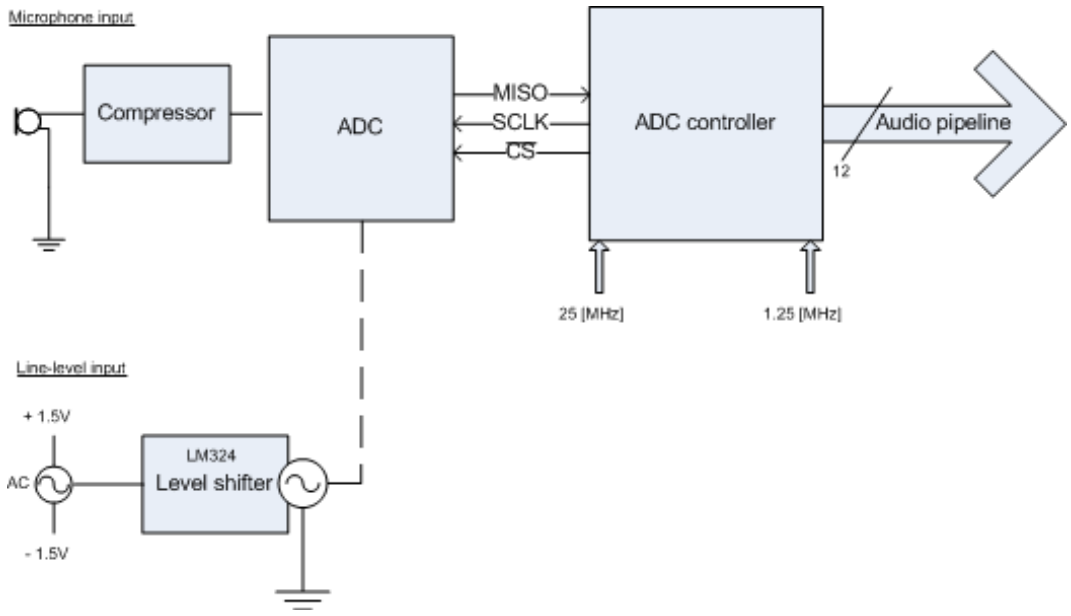


Figure 4.9: Audio input subsystem functional block diagram

### 4.7.2 **ADCS7476**

#### *SPI Interface*

The ADC's in the system have similar pinouts to the DAC's, with the MOSI pin being replaced with MISO (Master In Slave Out), and referred to as SDATA in the pinout diagram below.

#### *Synchronicity*

The ADC, like the DAC, is rated for a 20 MHz serial clock. It is clocked at 25 MHz in the prototype. The results of this will be monitored.

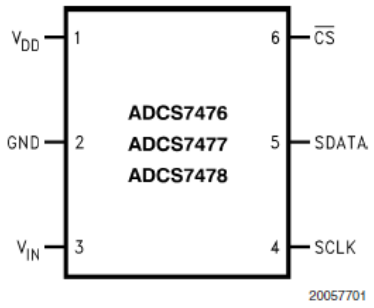


Figure 4.10: Pinout diagram of ADCS7476 [19, p.1]

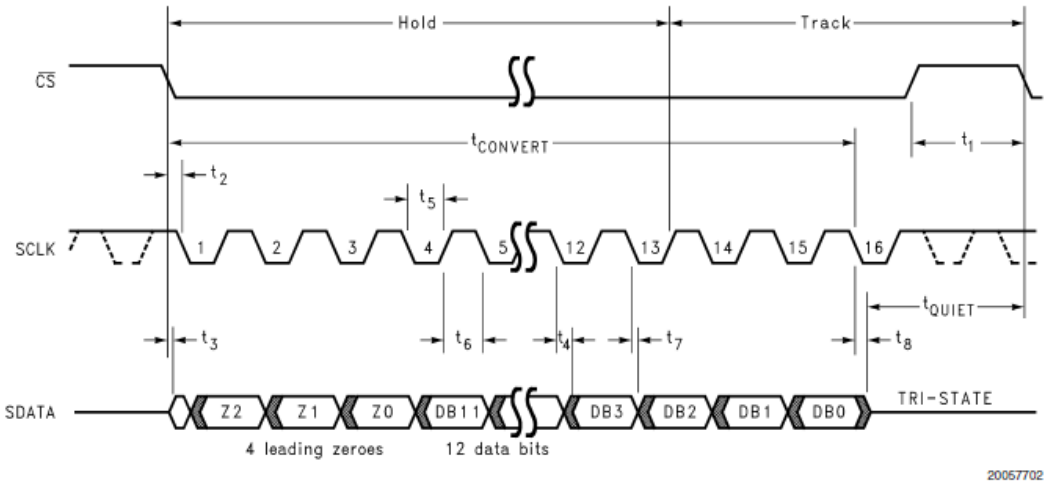


Figure 4.11: Timing diagram of ADCS7476 SPI interface[19, p.11]

### *ADC Input Voltage Level Shifter*

The Nexys2 board has a single 3.3 V power supply. However, the typical analogue audio signal is an AC signal with a mean of 0. In order to avoid placing a negative voltage on the ADC input, a level shifter is used to shift the signal to a mean of 1.6 V.

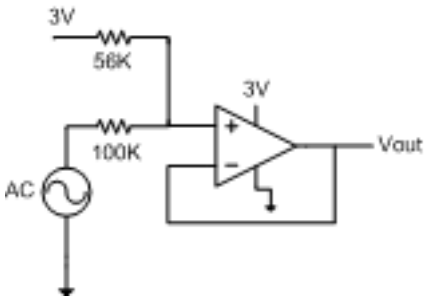


Figure 4.12: Circuit Diagram of 1.5 V Level Shifter with Non-Inverting Single Supply Operational Amplifier

### 4.7.3 Functionality of ADC Controller IP

The ADC controller operates much like the DAC controller, but receives data from the external component instead of sending data out. Accordingly, the line used is called Master In Slave Out (MISO), and not Master Out Slave In (MOSI).

The same general design of the DAC controller is used. Instead of a rotation register, the controller uses timing logic to load each serial bit in the appropriate bit of the slave register. This would be much like the timing logic on the DAC serial input, except since the IP is the SPI master, it initiates transfer with the ADC using the $\overline{CS}$ line.

Once the data has been read from the ADC, it is placed on the `AUDIO_OUT` port.

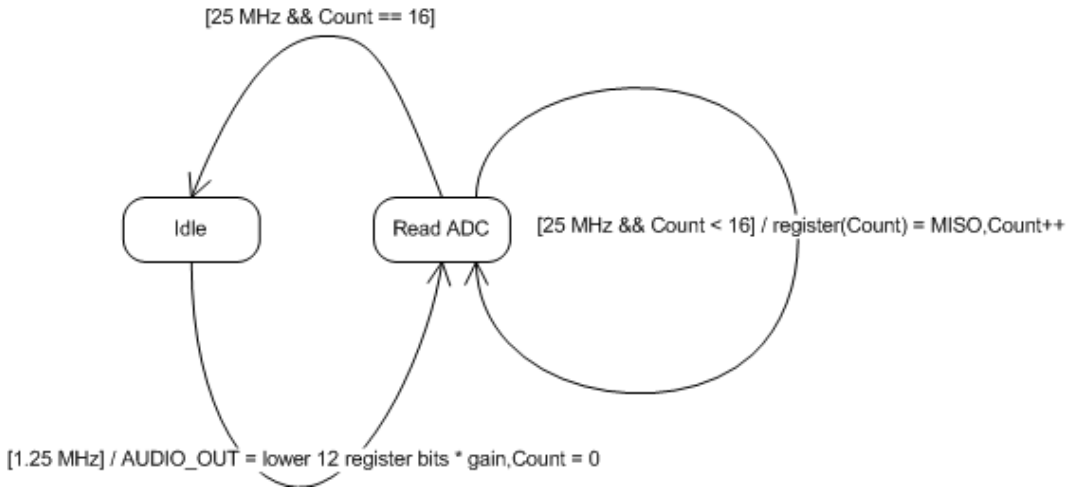


Figure 4.13: Statechart for ADC controller IP

The ADC value begins with the MSB, corresponding to register index 0 in the big-endian system.

### 4.7.4 Implementation of ADC controller

An input clock counter is used to write the MISO value to the appropriate Ada index of an internal register at the rate of 25 MHz. The lower 12 bits of the register are placed on the `AUDIO_OUT` port at the pipeline rate of 1.25 MHz.

A generic is used to specify a gain, which is used to multiply the input value. This is useful because the compressor produces an output signal with limited range. An effective value for this gain differs between the PmodMic and PModAD1 extension boards, with 4 being typical for the former and 6 for the latter, empirically.

The ADC controller IP does not require slave register I/O or a software driver.

## 4.8 DESIGN 0: MICROPHONE/LINE-IN MONITORING

The most basic VAD reconfiguration connects the microphone or line-in directly to the speaker. Sound entering the microphone will exit the speaker.

These systems are created in System Assembly View.

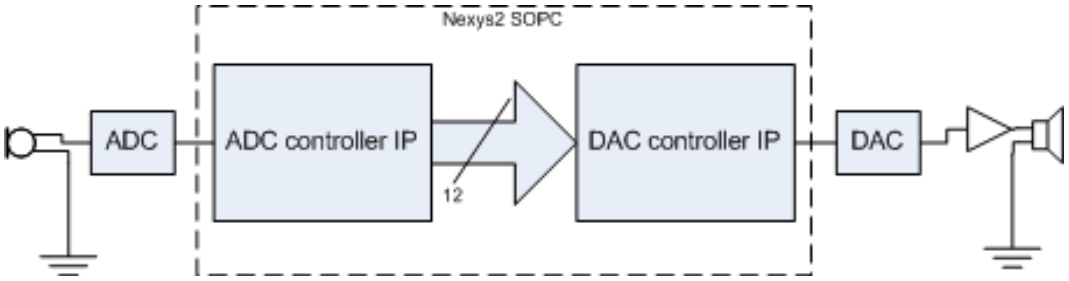


Figure 4.14: VAD Reconfiguration for Microphone/Line-In to Speaker 'monitor' connection

This reconfiguration is composed purely of hardware peripherals, therefore `system.bit` can be used to program the device and no Microblaze software is required.

## 4.9 INPUT AND OUTPUT SOUND BUFFERS IP CORES

### 4.9.1 Functionality of Sound Buffer IP's

The sound buffer IP cores exist at the boundary of the hardware/software partition. They provide a means for the Microblaze software to place audio samples in the audio pipeline. These samples are stored in RAM buffers and are transferred onto the Microblaze from a PC using the RS-232 port.

In order for these samples to reach the DAC, the Microblaze must transfer them from RAM to the audio pipeline. It is assumed the software will not execute at a fixed rate, and thus it is the responsibility of the sound buffer IP's to ensure that samples enter or exit the pipeline at their given sampling rate, which is 44.1 KHz in this case.

Once the samples enter the pipeline, they are clocked along systolically at the rate of the audio bus, 1.25 MHz. Thus, the function of the audio buffer is to enforce a real-time sampling rate constraint on the software, and to negotiate between this sampling rate and the processing rate of the audio pipeline.

### 4.9.2 Implementation of Sound Buffers IP's

A buffer allows samples to be queued as they enter or exit the pipeline, providing leeway for the software if occassionally it cannot process samples quickly enough.

The simplest buffer to implement in VHDL for this purpose is a ring buffer. If the buffer becomes full, no further data can be added. If it becomes empty, no data can be removed. The buffer size is exposed to software through a slave register, and in this way, the software may poll the buffer regularly to determine if data should be placed onto or removed from the buffer.

The following variables are used to implement a ring buffer.

- **Hardware buffer** 256 bits of logic resources are set aside for the sound buffer. 12 bit samples are stored at intervals of 16 bits, so that in the future the sound buffer may process 16-bit audio.

  $\frac{256}{16} = 16$ samples can be stored in this buffer.

  At a sampling rate of 44.1 KHz, this provides $(44100)^{-1} \times 16 = 362.8\mu s$ of leeway to the software program, assuming the buffer was full, for output, or empty, for input.

- **Buffer size** This is available through slave register 0. For the sound output buffer, a size of less than 256 means that more samples should be loaded onto the buffer. For the input buffer, a size of more than 0 means that more samples should be read from the buffer.

- **Read index** This is the sample index of the next sample 'read' operation. The circular nature of the buffer arises when the read index reaches 256: it then rolls over to 0 using a modulo operation.

- **Write index** The sample index of the next 'write' operation. This too is circular and rolls over.

  The use of the buffer size constraints means that the read index will never surpass the write index, in the case of an input buffer, and the write index will never surpass the read index, in the case of an output buffer.

### *Sound Output Buffer Software Interface*

The software design pattern for interacting with the sound buffers is similar in the input and output cases. The buffer size is polled in the main loop, and action taken accordingly. This is an efficient way to control multiple buffers.

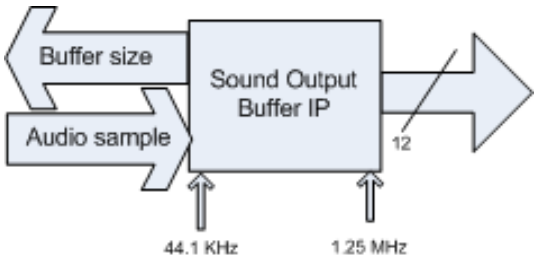


Figure 4.15: Sound Output IP Functional Diagram

```
// get buffer size
value32 = SOUND_BUFFER_OUTPUT_mReadSlaveReg0(BASEADDR, 0);
// is there space on the buffer?
if (value32 < 256) {
    // write current sample to buffer
```

```
    SOUND_BUFFER_OUTPUT_mWriteSlaveReg0(BASEADDR, 0, i);
    sample = (sample + 1) % SAMPLER_SAMPLE_SIZE;
    // calculate NEXT sample value; store in i.
    if (recordingPlay == 1) {
        // add recording track sample to i, normalise
    }
}
```

### *Sound Input Buffer Software Interface*

The sound input buffer has an additional complication. Because reading occurs in an asynchronous process, the act of reading the next sample value alone cannot cause the sound input buffer IP to update the buffer variables. The IP must be synchronously notified that the value has indeed been read by writing any value to slave register 0.

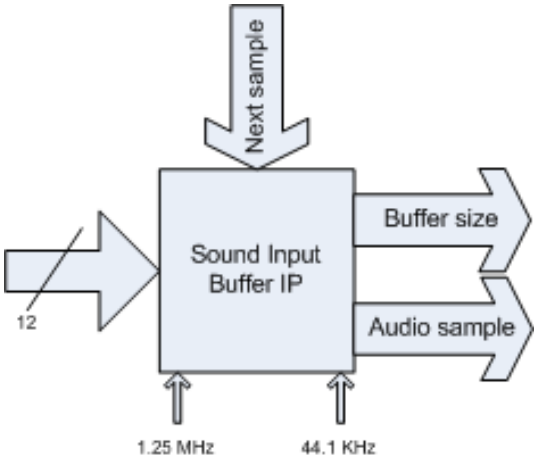


Figure 4.16: Sound Input Buffer IP Functional Diagram. The additional slave register input triggers the next sample read.

```
if (recordIndex == SAMPLE_SIZE) {
    // done recording.
} else {
    // get buffer size
    value32 = SOUND_BUFFER_INPUT_mReadSlaveReg1(BASEADDR, 0);
    // are there samples for us to read?
    if (value32 > 0) {
        // trigger that we are reading the next sample
        SOUND_BUFFER_INPUT_mWriteSlaveReg0(BASEADDR, 0, 0);
        // read sample
        sample1[recordIndex++]
            = (Xuint16)SOUND_BUFFER_INPUT_mReadSlaveReg0(BASEADDR, 0);
    }
}
```

***Software Sound Buffers***

The following audio buffers exist in the VAD software. The number of samples and their durations are limited by the amount of RAM, and the sampling rate and bit depth of the audio data.

1. Samples for the sampler/drum machine. The VAD prototype has one sampler with 8 samples. The sampler is 8-bit, for ease of implementing the RS-232 interface, which transfers 8 data bits per frame. This buffer is called `RecvBuffer`, after the sample RS-232 application, and is implemented as a free variable.

$$1s \times \frac{44100}{1\ s} \times 8\ samples \times \frac{1\ byte}{1\ sample} = 352808\ bytes$$

   of 8-bit audio are allocated. In this case, the term 'sample' is used in the audio studio sense and refers to a sample set comprising a brief sound.

2. Tracks which have been recorded by the user and can be looped. The VAD prototype has one 16-bit track called `sample1`. Four bits are wasted, since the value from the ADC IP is 12-bits.

$$\frac{2400000\ samples}{44100\ samples/s} = 54.422\ s \text{ of audio recording time.}$$

### 4.9.3 Future work

16-bit audio sample buffers could be supported by using software shift operations to recreate the sample values from the RS-232 input. This technique is demonstrated in the `vlf()` MIDI function. The rest of the VAD system would be modified accordingly.

Direct Memory Access (DMA) could be used to load the sound buffers directly from RAM. This would mean that no logic resources would be needed for a sound buffer. However, resources would be needed for the DMA controller IP.

## 4.10 DESIGN 1: MICROPHONE RECORDING AND PLAYBACK

The following system reconfiguration uses sound buffer IP's to record and play back audio recorded from a microphone or line input. Arbitary audio data and samples from the sampler can also be played using software.

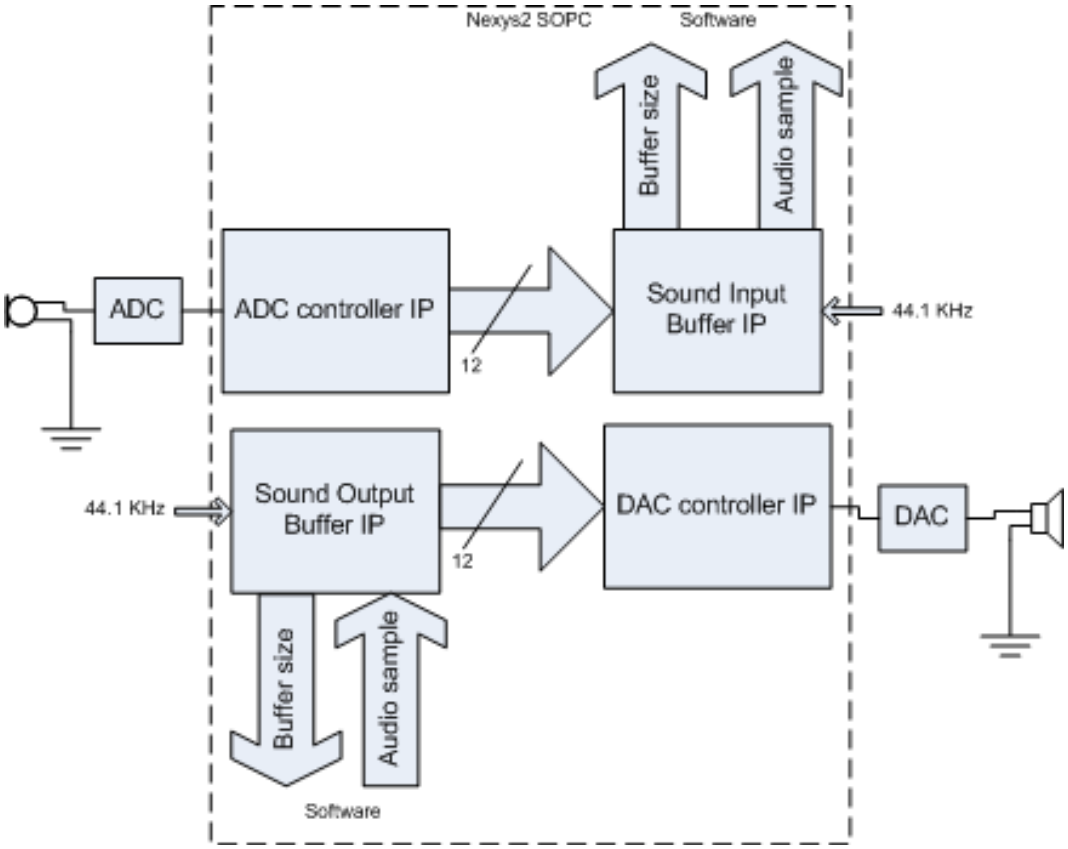


Figure 4.17: Reconfiguration for microphone recording and playback

## 4.11 OSCILLATOR IP CORES

### 4.11.1 Function of oscillators

VAD uses hardware oscillators to generate waveforms corresponding to musical notes. Four types of oscillators are provided. These oscillators have adjustable frequencies. The sine oscillator has adjustable phase. They produce waveforms with a range from 0 to 4096, to occupy all bits of the audio pipeline.

The essential function of the oscillator is to produce a waveform at a given frequency.

### 4.11.2 Types of oscillators

- Sine oscillator. The sine wave is a pure frequency, or tone. It is the least encumbered sound and has the simplest possible Fourier representation.
- Square oscillator with adjustable duty cycle. The square oscillator is immediately recognisable from Nintendo games and other lo-fi audio applications, since the square wave is historically one of the easier waveforms to generate, both in the analogue and digital domains.
- Triangle oscillator. The triangle is a subtle tone, being continuous.
- Saw tooth oscillator. The saw tooth is one of the more complex waveforms and has a very colourful Fourier representation due to its discontinuity. These result in what would commonly be referred to as 'rich harmonics'.

### 4.11.3 Properties of oscillators

Each oscillator 'contains' one period of a waveform. Empirically, there are 64 samples in one period.

Each oscillator has an `index`, indicating which is the current sample of the 64. The index is incremented at a given frequency, specified by the number of 50 MHz clock cycles that should be counted between incrementations.

The following equation is used to calculate the number of cycles to count for a given frequency. There is a maximum error of 20ns, which is the duration of a single clock cycle.

$$count\_until \approx \frac{50MHz}{target\_freq \times 64},\ count\_until \in \mathbb{Z}$$

This relationship leads to the term 'frequency divider'.

#### *Generation of Count Values*

The oscillators must be able to produce the frequencies corresponding to the notes of the equally-tempered musical scale [21]. A short Matlab script was written to convert the table of musical frequencies into a Look-Up Table relating MIDI note numbers to count values.

```
counts = (50000000)./(freqs * 64);
counts = floor(counts);
```

The LUT is implemented as a static C array in software. It is the responsibility of software to translate from MIDI note numbers to count values.

### 4.11.4 Implementation of Oscillators

The oscillators are implemented as Microblaze peripherals with `AUDIO_OUT` ports. A Microblaze software application controls the oscillators by specifying parameters through slave registers.

- Slave register 0 is used to indicate phase or duty cycle. Not all oscillators in the system support these parameters.
- Slave register 1 is used to indicate frequency. A frequency of 0 means that no waveform should be generated and the median value of 2048 should be output instead.

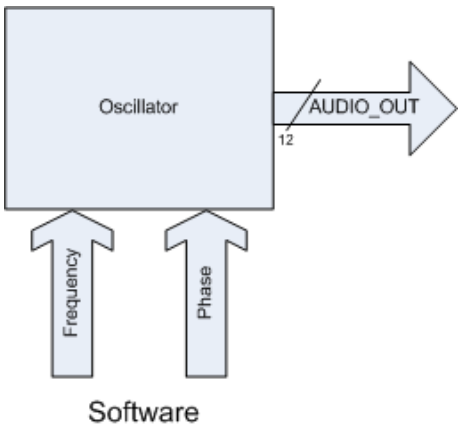


Figure 4.18: Functional Diagram of Typical Oscillator IP

These registers are loaded using driver functions generated by CIP wizard. It is not necessary to read values from the slave registers, so the slave read logic may be commented out.

***Synchronicity***

The oscillator increments the index at some given frequency. The sample value indicated by the current index location is placed on the audio pipeline at a frequency of 1.25 MHz. This occurs regardless of the frequency of the oscillator. Theoretically a frequency of up to half that should be synthesisable, using a trivial square wave.

The following construct is used to provide arbitrary frequency specification in the main register write synchronous process.

```
if counter >= countUntil then
    counter := 0;
    -- move to the next sample
    index := (index + phase + 1) mod 64;
    -- additional logic to create sample from index.
end if;
```

### *Waveforms based on arithmetic*

Oscillators which generate waveforms corresponding to arithmetical equations can be implemented directly as a function of the current index. For example, the square wave produces its duty cycle as follows.

```
if (index > pulse_width) then
    output := 0;
else
    output := 4095;
end if;
```

### *Waveforms based on Look-Up Tables*

The common way to implement a sine wave oscillator is to use a Look-Up Table (LUT). Only the first quadrant of the sine wave needs to be quantified; then the other three quadrants can be derived.

$$\frac{64}{4} = 16 \text{ samples in sine wave LUT}$$

The LUT is a third-party VHDL package which is created by the Doulos VHDL sine wave LUT generator script located at

http://www.doulos.com/knowhow/vhdl_designers_guide/models/sine_wave_generator/

The script is used only to generate the LUT, which is called `sine_package`.

The LUT package uses a `when`-style design, which XST should synthesise using FPGA LUT's. The data and address bit widths must be specified. For 16 samples, the address width is $log_2 16 = 4$ bits. The first quadrant ranges from 0 to 2048, giving a data width of $log_2 2048 = 11$ bits. $16 \times 11\ bits = 176\ bits = 22\ bytes$ of LUT resources are used.

Care must be taken to modify the table value integer subtype in `sine_package` so that the full range of 0 to 4096 can be represented.

The sine oscillator performs a number of gymnastics to synthesise a waveform using only a LUT of the first quadrant. This makes the sine oscillator the biggest and most complex of the oscillators.

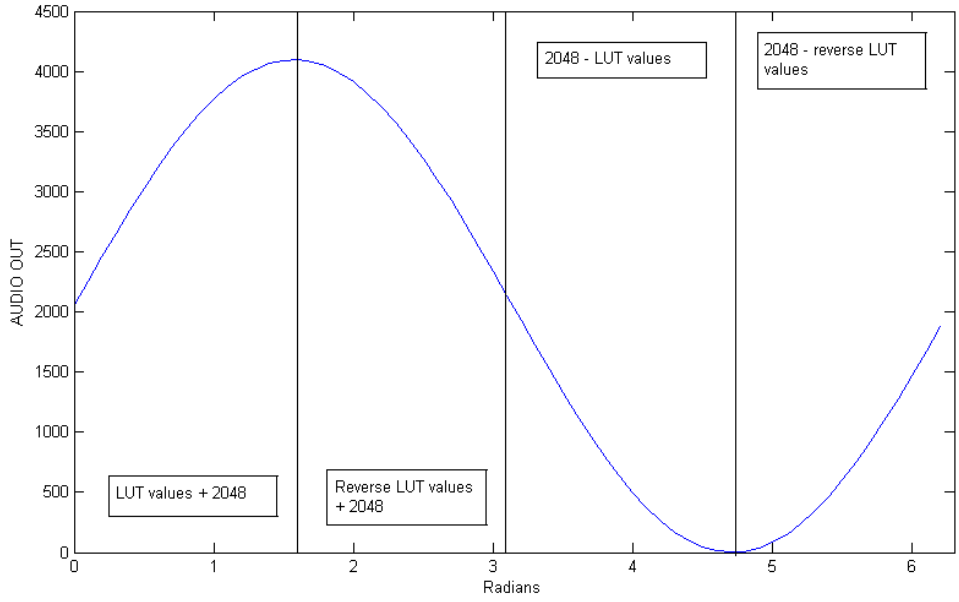


Figure 4.19: Generating sine wave from first quadrant LUT

### 4.11.5 **Polyphony**

'Polyphony' refers to the capability of the device to synthesise multiple musical notes at once. This is achieved by including multiple oscillator IP's in the VAD design. Each oscillator is entirely indepedent and oscillates in the hardware domain.

By grouping oscillators of the same waveform in the memory map, software macro's can be written which use any one of a number of fungible oscillators to produce a particular musical note.

### 4.11.6 **Future work**

All oscillators should support the phase parameter.

It would be interesting to create an oscillator with a software-configurable LUT.

## 4.12 DESIGN 2: MONOPHONIC MUSIC SYNTHESIS

An oscillator can be wired directly to a DAC controller to create a waveform generator or monophonic music synthesiser.

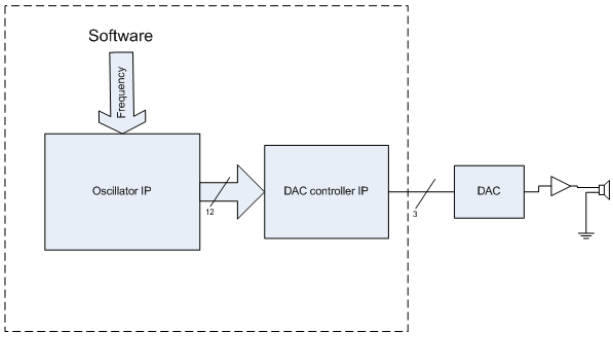


Figure 4.20: Functional Block Diagram of Single Oscillator VAD Reconfiguration

## 4.13 MIXERS

### 4.13.1 Function of Mixer IP

The mixer is a simple but crucial element in the VAD system. It has four inputs and one output. This refers to the audio studio definition of a mixer.

The mixer IP is responsible for the point-wise addition of audio waveforms, such that the listener perceives that both waveforms are playing simultaneously. This is because the linear nature of the Fourier transform means that all frequency components of both signals are present in their sum, and it is these frequency components which are detected by the human ear. Sometimes the waves will reinforce one other, and sometimes they will counteract.

***Normalisation***

The obvious problem with adding multiple waveforms together is that two 12-bit samples have a 13-bit sum. Clipping will occur when the sum value of the input signals requires more than 12 bits to represent.

The easiest way to prevent this is to ensure the signal is always normalised to a range of 0 to 4096 by taking the average of many signals as opposed to the sum. The affect is that the individual waveforms have less amplitude in the resulting signal, their frequency components having been scaled.

### 4.13.2 Implementation of Mixer IP

The mixer latches its four inputs in the `AUDIO_IN` read cycle. These are summed, shifted by two to divide by $4 = 2^2$ and placed on the output in the `AUDIO_OUT` write cycle.

The mixer does not require slave register I/O and exists entirely in the hardware domain. No software driver is required.

***Number of inputs***

Communication resources are the constraint on an FPGA platform. Caution must be taken not to allocate too many input 12-bit audio input bus inputs to the mixer IP's, since it is anticipated

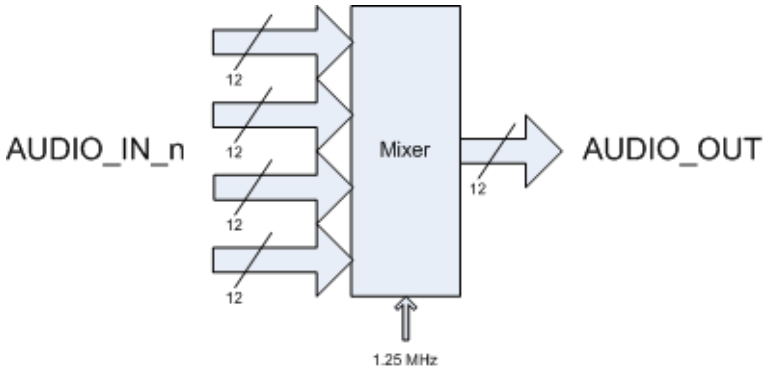


Figure 4.21: Mixer functional block diagram

there will be several of these IP's in the system. A compromise must be made between the input line resources and the number of mixer IP's required. Additionally, the shift operates in powers of two. Finally, care must be taken not to divide by too large a value and so attenuate the signal. Four is then a reasonable value.

## 4.14 DESIGN 3: POLYPHONIC SYNTHESIS

This reconfiguration uses multiple hardware oscillators to generate four musical tones simultaneously.

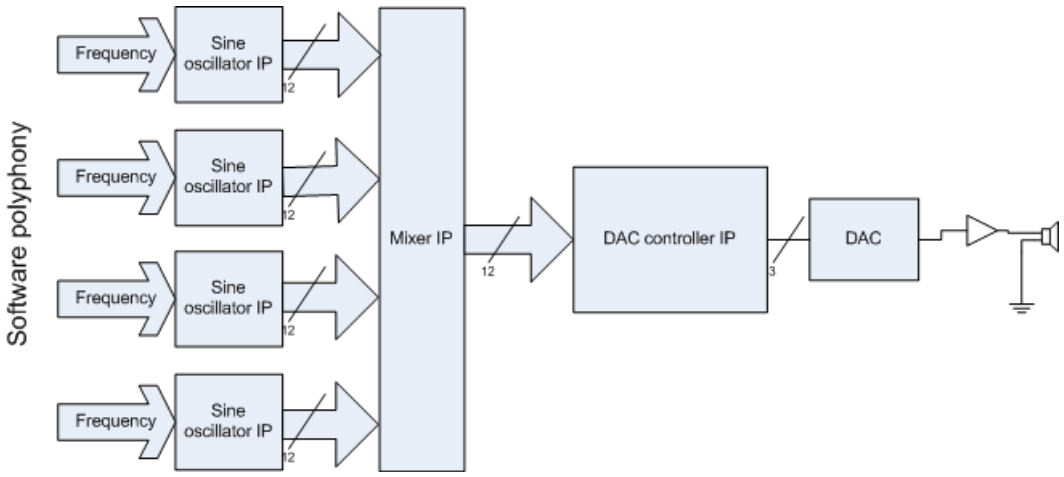


Figure 4.22: Functional Block Diagram of Four Sine Oscillator Polyphonic VAD Reconfiguration

## 4.15 DESIGN 4: RECORDING WITH LIVE MONITORING

This reconfiguration uses two mixers to connect the audio inputs to the speaker and is referred to as 'live monitoring', since the input channels are audible. This connection is made purely in hardware and indicates the hardware system is working.

Audio can be recorded from all system inputs and played back simultaneously. Other samples could also be played while recording.

The software samples are 8-bit in the prototype while the ADC outputs and recording track are 12 bits. This disparity in bit-depth is allowed for since the samples tend to be louder than audio arriving at the inputs.

System Assembly View is quite content to connect the output of one peripheral to the inputs of multiple other peripherals.

Unfortunately, the mixers will divide the input by four, instead of two, which would be optimal. This suggests that a future mixer IP should have the ability to select the number of inputs in use with a generic.

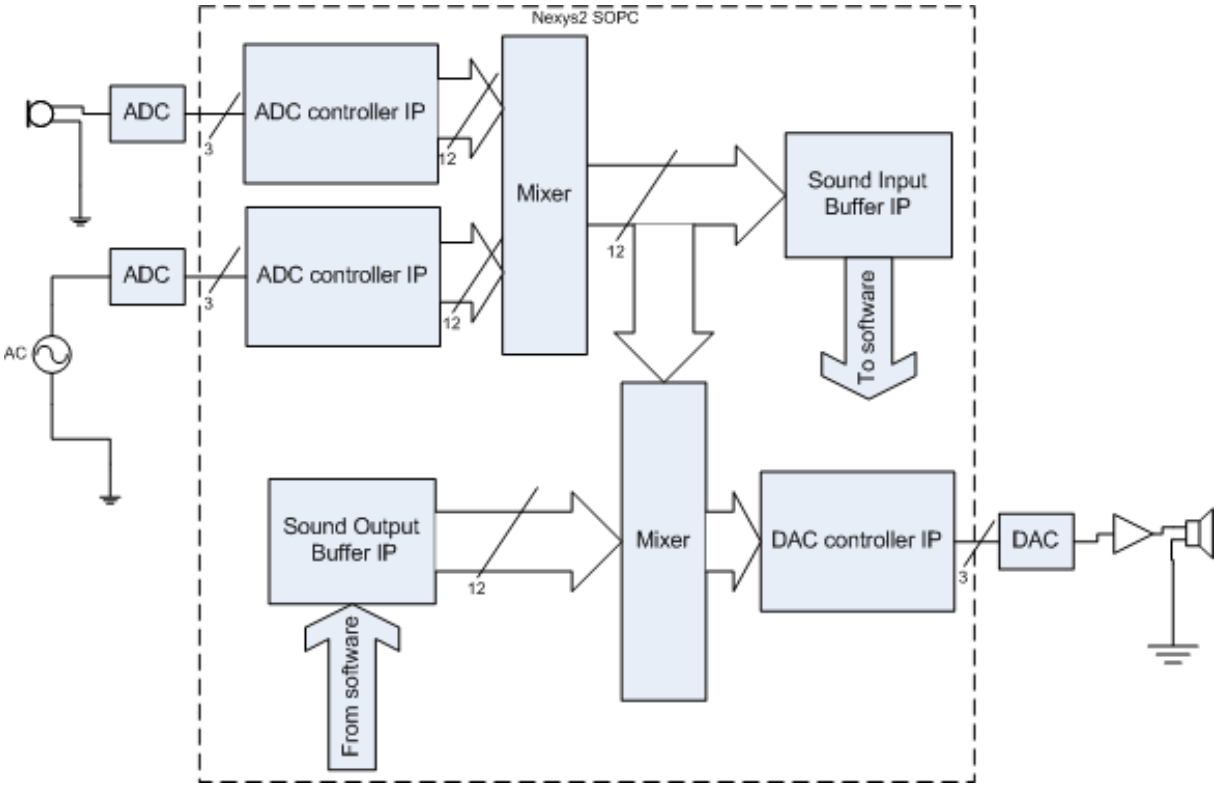


Figure 4.23: Functional Block Diagram of Multichannel Input Recording, Monitoring and Playback VAD Reconfiguration

## 4.16 MIDI FILE PLAYBACK

### 4.16.1 Musical Instrument Digital Interface (MIDI)

Musical Instrument Digital Interface (MIDI) is an old but widely popular digital musical format. It is an 8-bit RS-232 protocol operating at 31.25 KHz, using current signals and optocouplers for ground isolation.

### 4.16.2 MIDI file format

MIDI is most useful in the form of MIDI files. No external sequencer or instrument is required; just the commands stored in the files. These commands are the same as in the serial protocol.

Format 0 MIDI files are used. There is one header track which begins `MThd` and contains the basic timing information for the track. Even though this is the simplest MIDI file format, it is still rather cumbersome and will be covered briefly [20].

#### *MIDI timing*

Since a MIDI file is essentially a static version of the MIDI protocol, timing information is inserted to allow the sequencer to reproduce the commands at the correct intervals. These times precede each command and are referred to as ‘delta times’. The number of clock ticks

corresponding to each delta count is derived from the MIDI tempo meta-event and the clock precision indicated in the MIDI header.

### 4.16.3 MIDI Command and Data bytes

MIDI command bytes are signified with an MSB of 1. Data bytes have an MSB of 0. Thus, the largest data value which can be represented in a byte is 127.

Command bytes consist of two 4-bit nybbles. The upper nybble is the command. The prototype is only concerned with one MIDI command, Note On, with an upper nybble of 0x9.

The lower nybble indicates the track number to which the command applies. The system uses this information to assign musical notes to oscillators. Thus, MIDI has support for $2^4 = 16$ tracks. MIDI Track 1 has a nybble of 0x0, and so on.

The command byte is followed by a data byte to indicate the note value. This is used as the index to the musical note frequency LUT. A second data byte follows which indicates the 'velocity', or sound volume, of the note. This is used to multiply the waveform sample values.

***Variable Length Format (VLF) values***

Variable Length Format values are used to represent MIDI values greater than 127. Where a VLF value is indicated, up to four bytes may appear. The MSB of each byte indicates whether the VLF value continues to the next byte. The other seven bits are data. The bytes are big-endian and together represent a value of up to $7 \times 4 = 28$ bits.

***Meta Commands***

Meta commands exist only in MIDI files and are indicated by a command byte of 255. The next byte indicates the command, and the next, a VLF value indicating the length of the command. The most important meta-command is the 'tempo' command, 0x51, which relates the system clock to the command delta times.

***Implementation of vlf()***

Software shift operations are used to recreate the original value. This is the most difficult task when reading MIDI.

```
u32 vlf(int *offset) {
  u8 deltaBuffer[4];
  u32 deltaTime = 0;
  int variableLength;
  int j;
  for (j = 0; j<4; j++) {
  deltaBuffer[j] = midiBuffer[*offset+j];
      if ((deltaBuffer[j] & 0x80) == 0x0) {
        // if MSB is 0, we're done
            break;
```

```
        }
    }
    // length of variable length delta value
    variableLength = j;
    // calculate delta value
    for (j = 0; j <= variableLength; j++) {
        deltaTime += ((Xuint32)deltaBuffer[j] & 0x7F)
            << (7*(variableLength-j));
    }

    //  update the user's offset
    *offset += variableLength + 1;
    return deltaTime;
}
```

### *VLF values in the MIDI File*

The following VLF values exist in the MIDI file:

- Delta times. These times precede each command and must be processed correctly to read a MIDI file.
- Meta command lengths. These commands have no fixed length and so their lengths are specified as VLF values.

### *Lazy commands*

The most confusing aspect of MIDI files are lazy commands. First, Note On with a velocity byte of 0 is the same as Note Off. The Note Off command is no longer needed. Secondly, when a data byte is received instead of a command byte, it is assumed that the previous command for that track remains in affect. Thus, the Note On command is inserted to specify track changes.

### *Future work*

MIDI input and output ports could be added to the Vad system using a second Uart 1650 IP, a connection to the pmod interface, a MAX232 and an optocoupler.

An input would allow the device to be played like a musical instrument, while an output would allow the device to control other MIDI instruments.

## 4.17 FINAL DESIGN

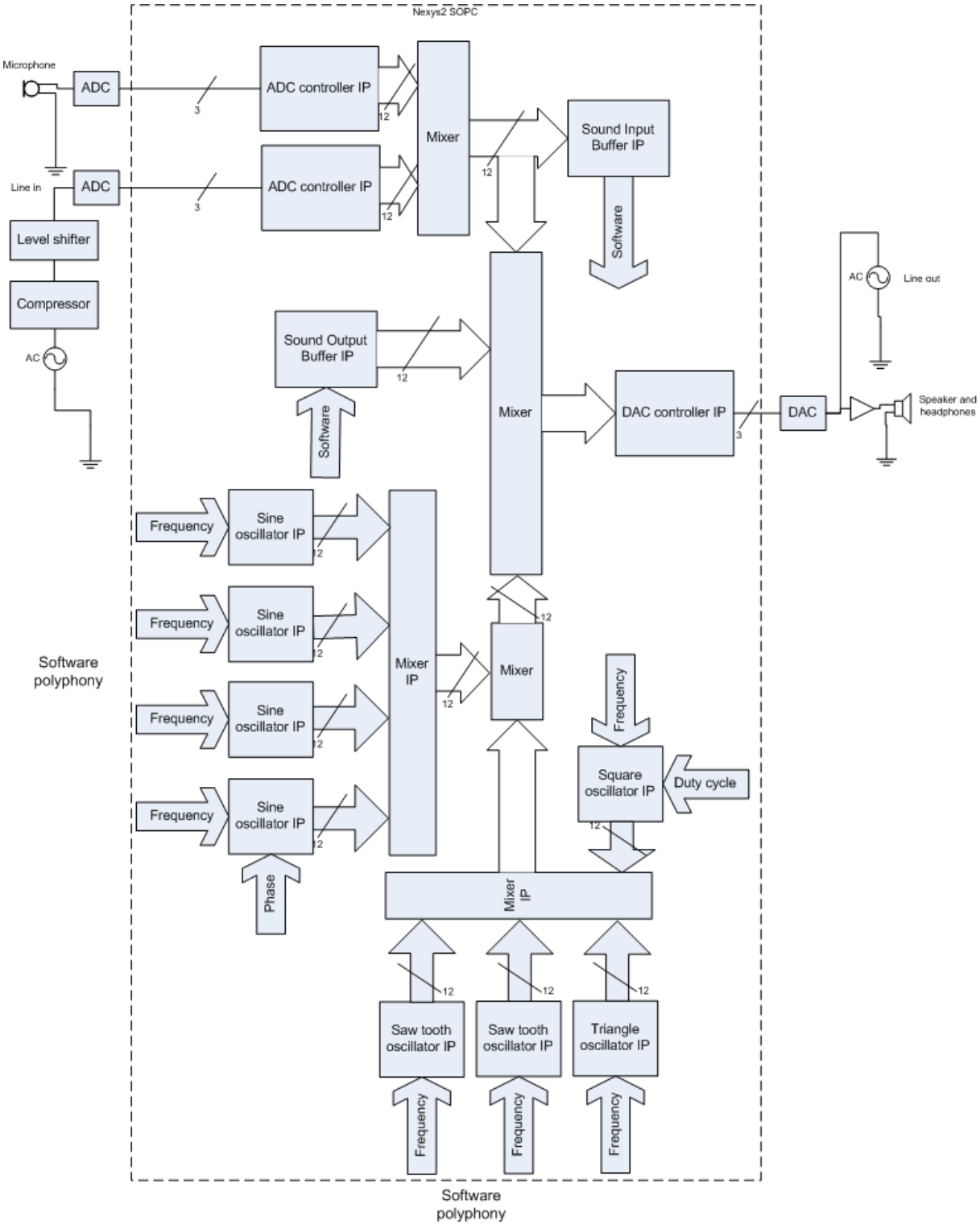


Figure 4.24: Functional Block Diagram of Final Prototype VAD System Reconfiguration

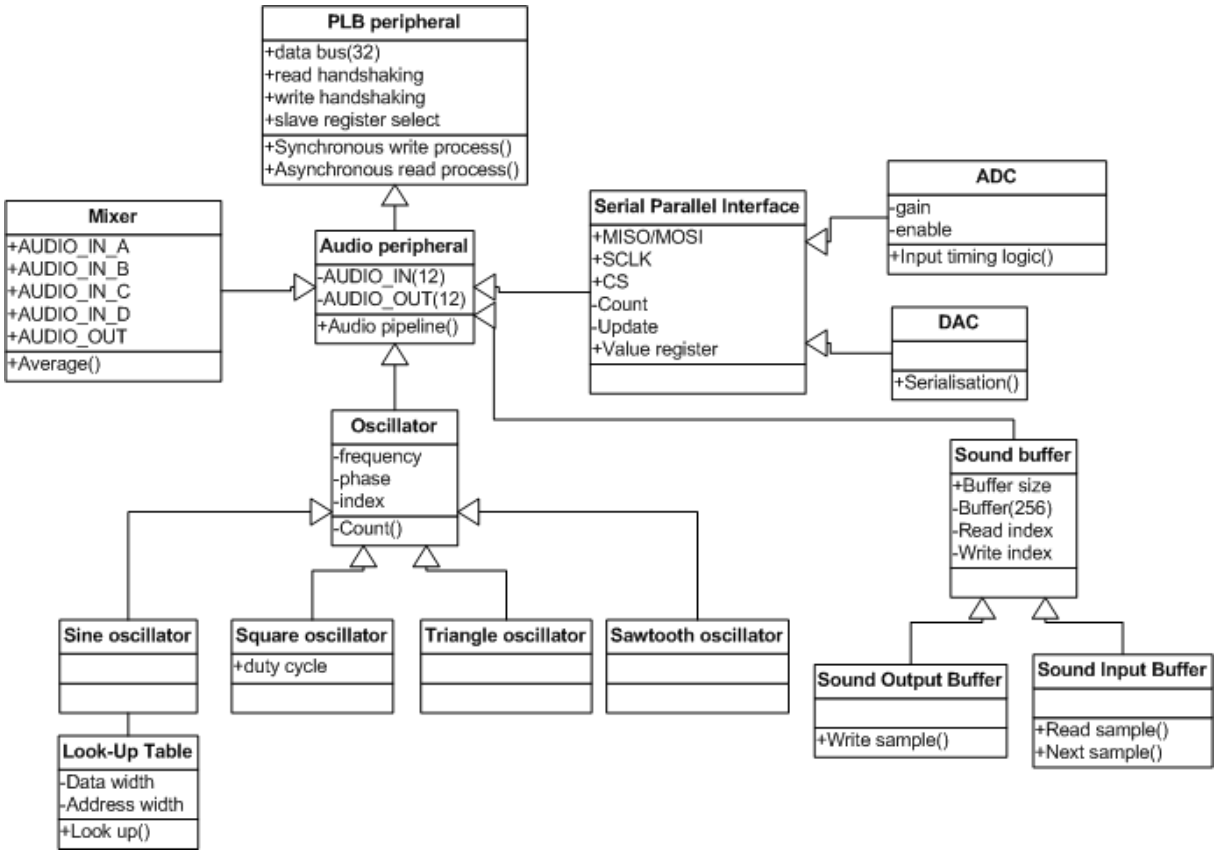


Figure 4.25: UML Conceptual Structure Diagram of VAD Custom Microblaze Peripherals. VHDL does not have a literal inheritance construct; the generalisations indicate common high-level attributes and methods.

## 4.18 RESULTS AND OBSERVATIONS

### 4.18.1 Experiment 0: SPI interfaces

***DAC SPI***

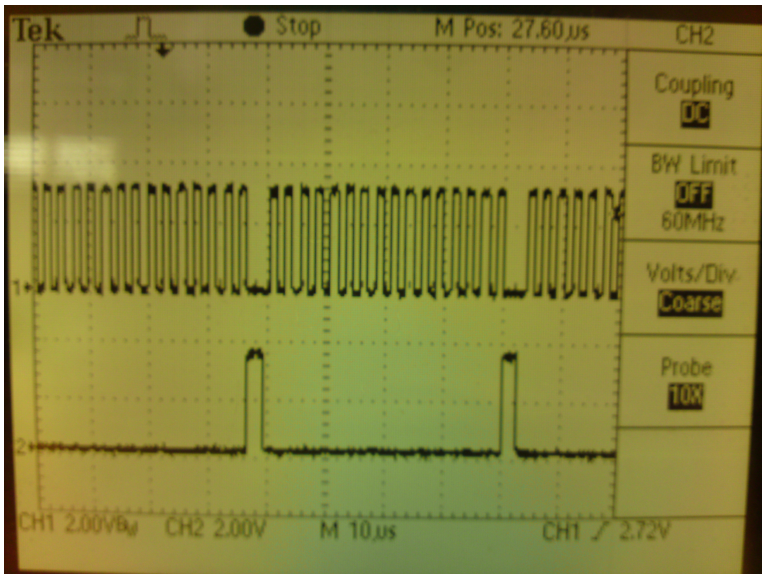


Figure 4.26: Oscilloscope display for DAC $SCLK$ (top) and $\overline{CS}$ (bottom). The clock does not run while the DAC is idle. There are 16 clock cycles.

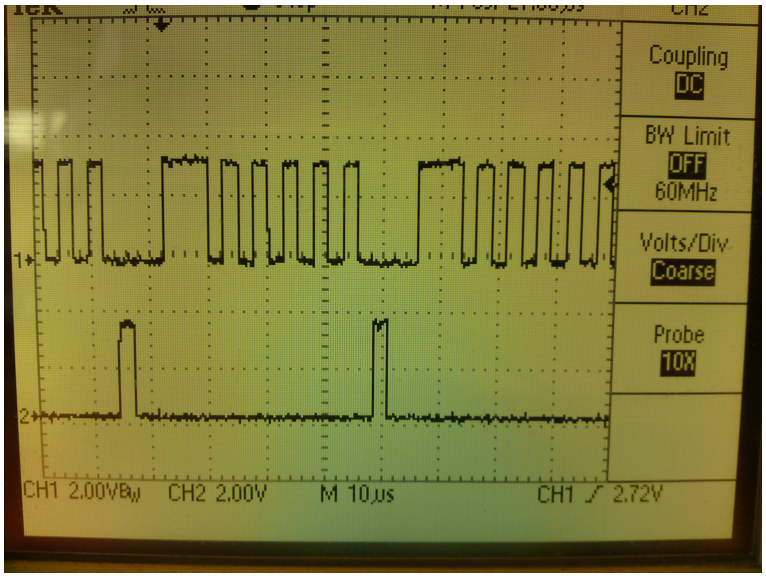


Figure 4.27: Oscilloscope display for DAC $MOSI$ (top) and $\overline{CS}$ (bottom). The first rise in the $MOSI$ signal is the DAC control nybble.

Both the DAC and ADC are rated for 20 MHz, but were found to work adequatelyl at 25 MHz. 25 MHz is 50 MHz divided by two; the SCLK signal alternates with every 50 MHz cycle.

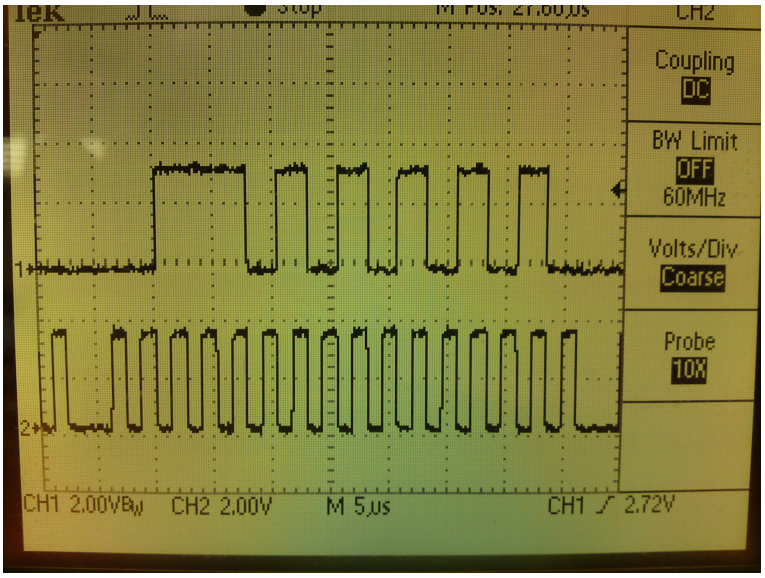


Figure 4.28: Oscilloscope display for DAC $MOSI$ (top) and $SCLK$ (bottom). Data is clocked into the DAC on the falling edge. 16 bits are clocked in.

Dividing by 3, the alternative, would operate at only 16.67 MHz, which would constrain the speed of the audio pipeline.

### *ADC SPI*

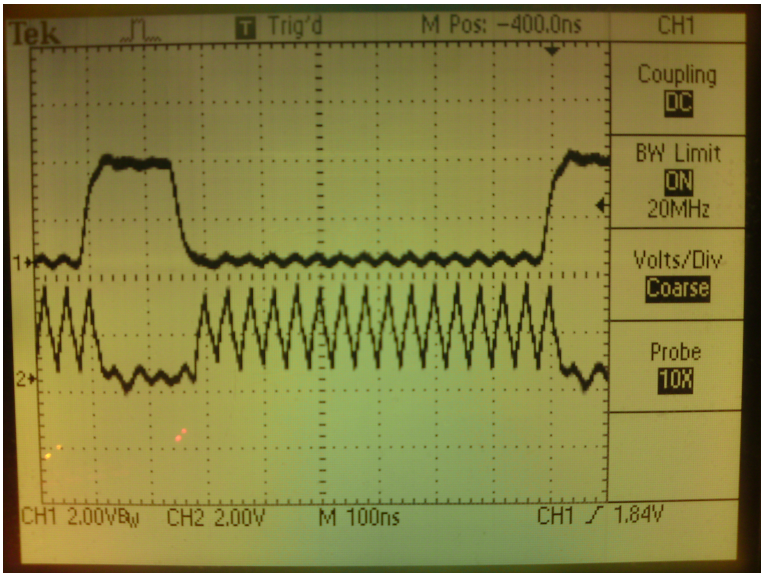


Figure 4.29: Oscilloscope display for ADC $\overline{CS}$ (top) and $SCLK$ (bottom). The clock signal appears pointy, as if it is just managing to charge. There are 16 clock cycles. The clock does not cycle when the chip is not selected.

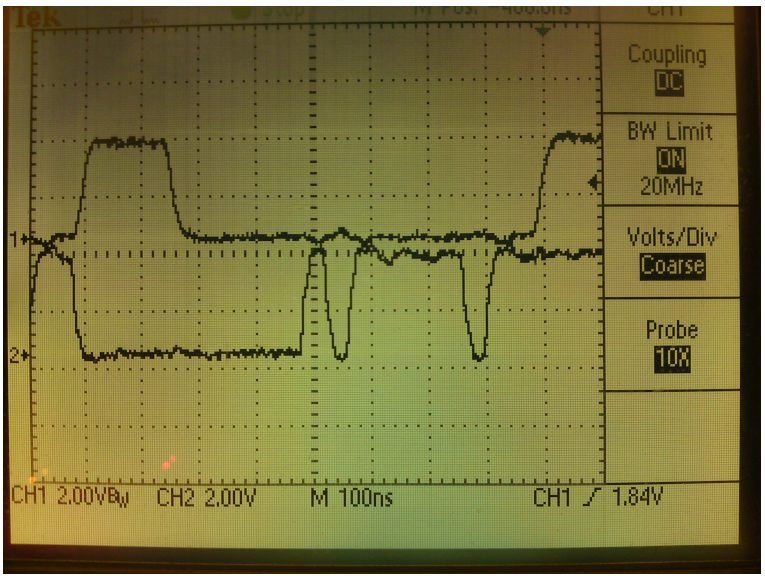


Figure 4.30: Oscilloscope display 1 for ADC $\overline{CS}$ (top) and $MISO(bottom)$ with arbitrary audio data. The first few MISO bits are always 0.

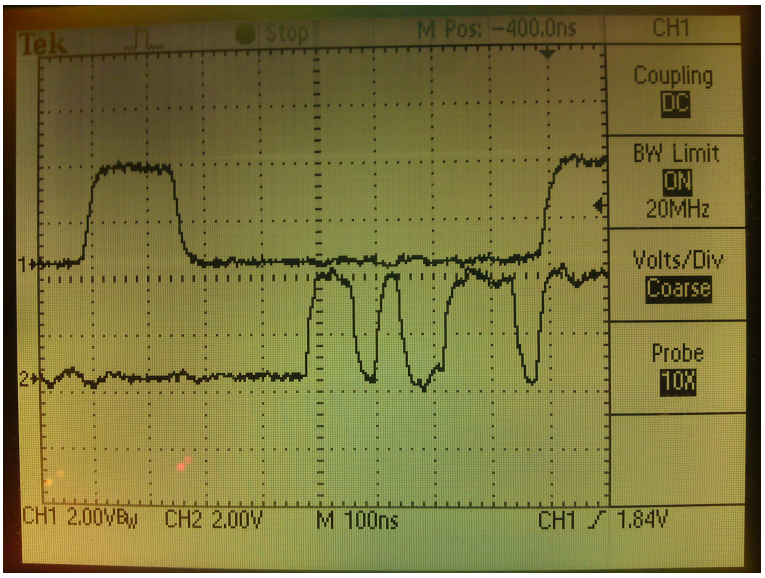


Figure 4.31: Oscilloscope display 2 for ADC $\overline{CS}$ (top) and $MISO(bottom)$ with arbitrary audio data.

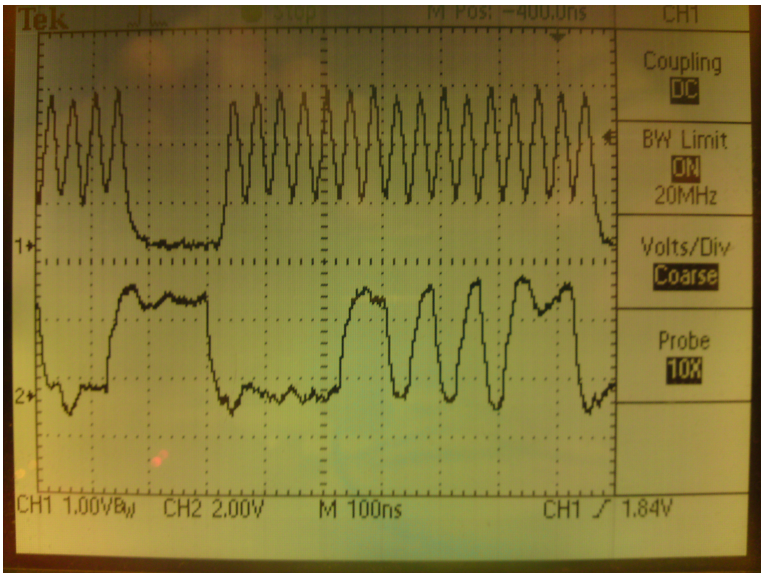


Figure 4.32: Oscilloscope display for ADC $SCLK$ (top) and $MISO$ (bottom) with arbitrary audio data.

### 4.18.2 **Experiment 1: Straight-through system**

Twenty seconds of each song are analysed in MATLAB.

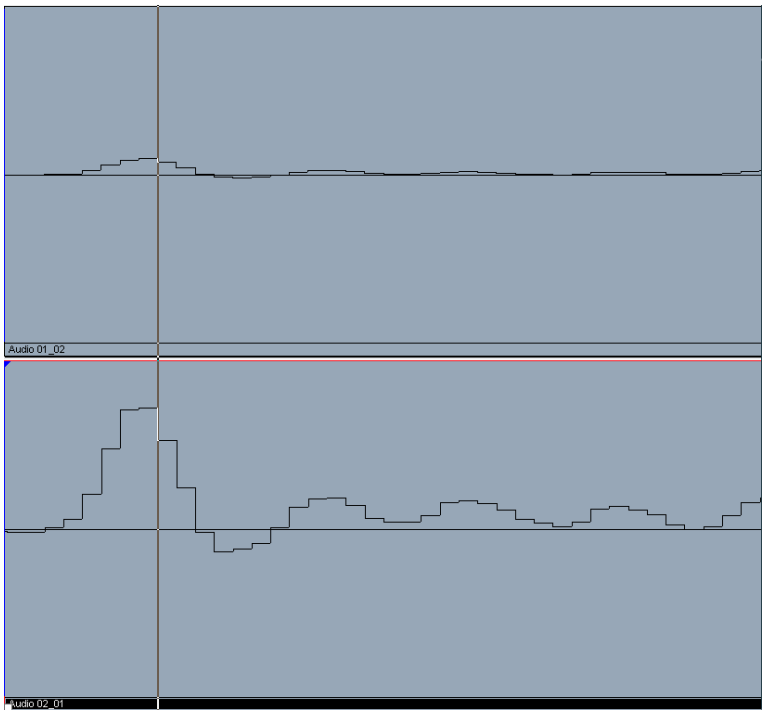


Figure 4.33: Aligning Input and Output Waveforms in Cubase

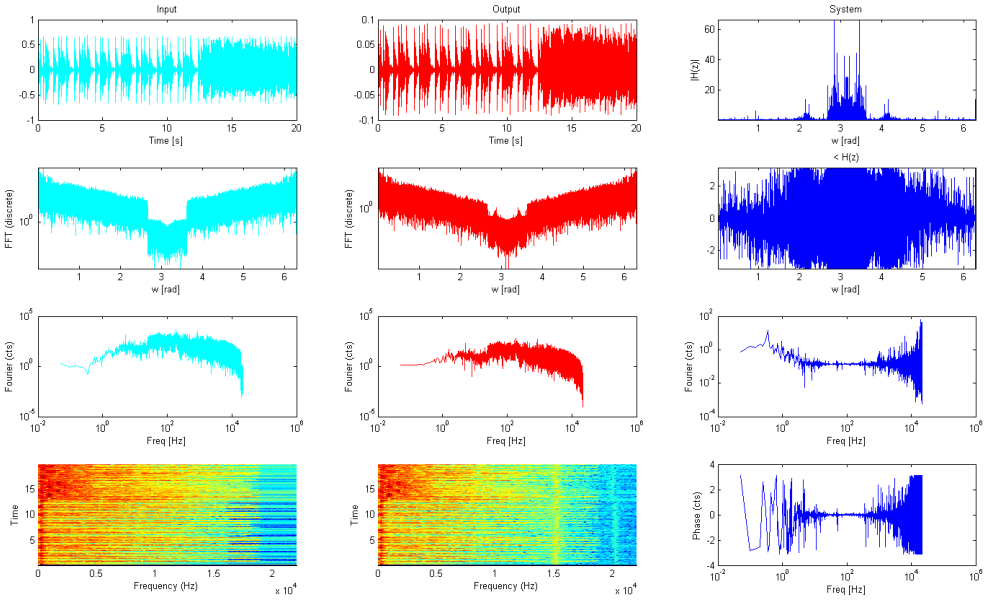


Figure 4.34: MATLAB Results for Test Song 1 in straight-through VAD reconfiguration Design 0.

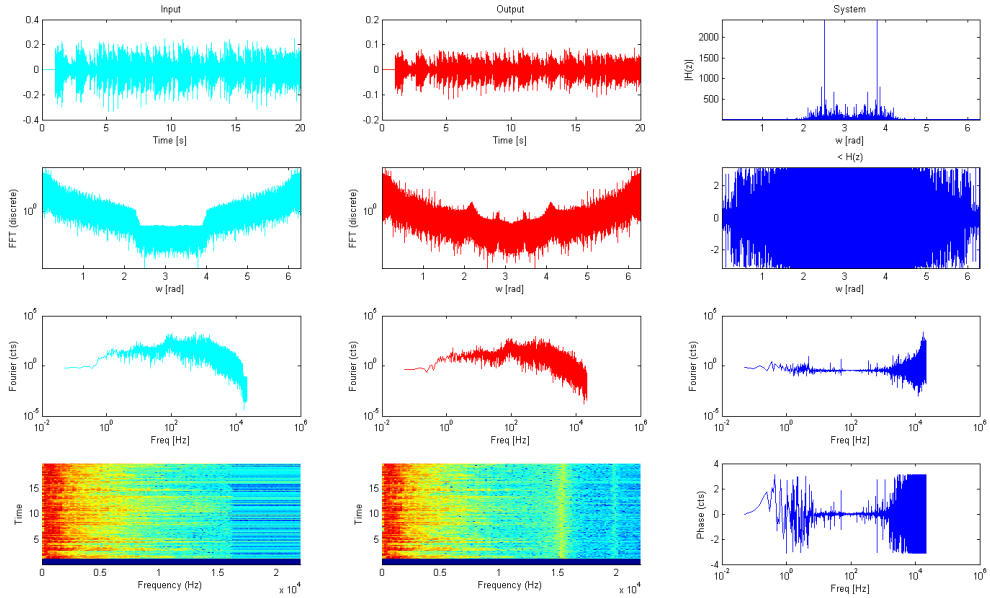


Figure 4.35: MATLAB Results for Test Song 2 in straight-through VAD reconfiguration Design 0.

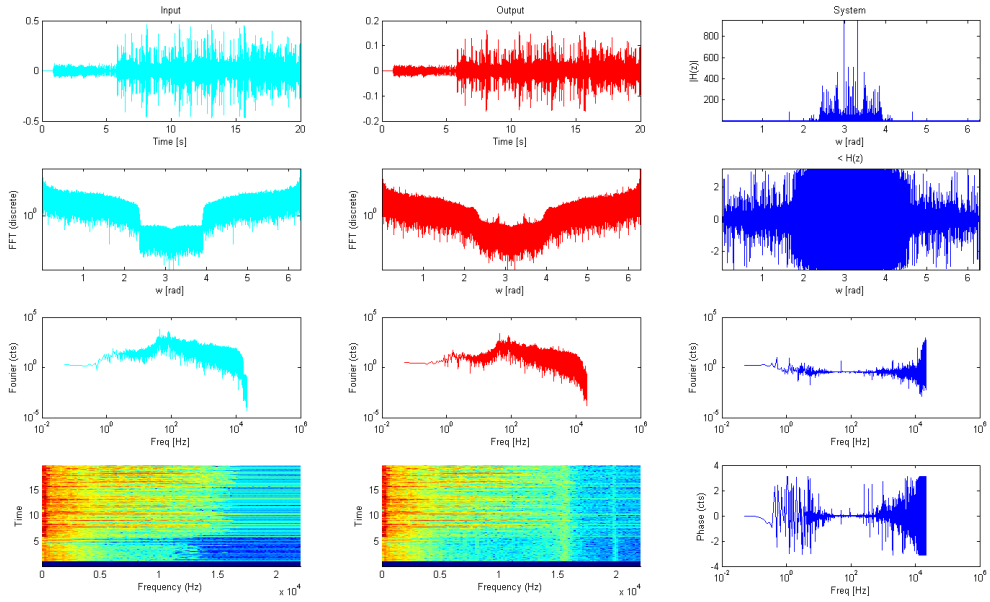


Figure 4.36: MATLAB Results for Test Song 3 in straight-through VAD reconfiguration Design 0.

Noise is the form of a high-frequency tone was audible during recording.

There are two primary frequency artefacts introduced by the VAD system, at 15 KHz and 20

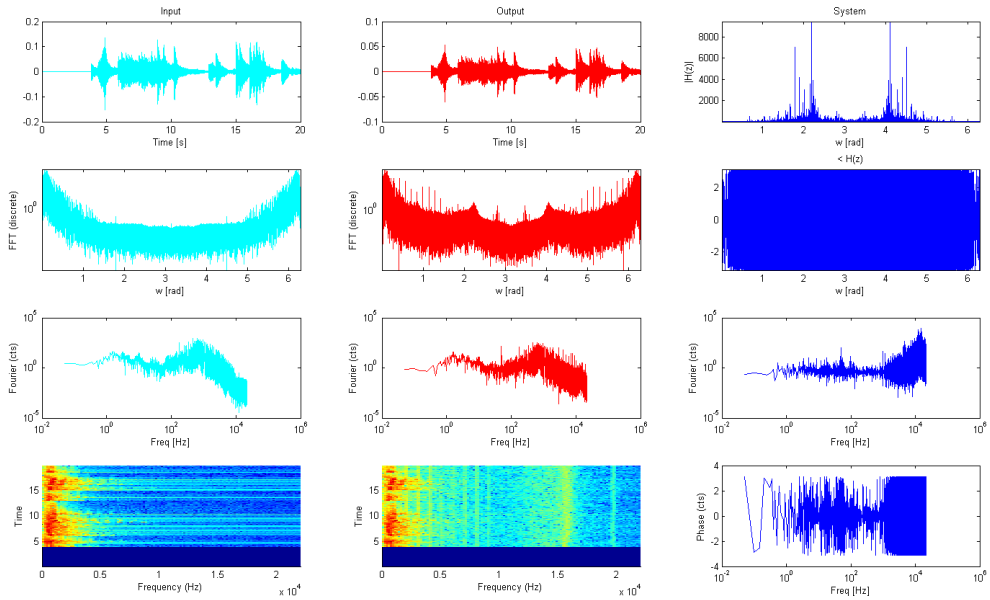


Figure 4.37: MATLAB Results for Test Song 4 in straight-through VAD reconfiguration Design 0.

KHz. The 15 KHz artefact is more pronounced, as indicated by the yellow colour on the spectrogram. These are visible on all four spectrograms.

All songs were recorded and played through the VAD system at an apparently high audio quality through a high fidelity amplifier system.

The system has a significant phase shift characteristic, which is not desirable for audio systems. This is especially noticable on the system characterisation derived using the Jazz genre test song. The softer song shows most distortion.

The VAD system characterisation appears consistent across the four test songs.

### 4.18.3 **Experiment 2: Audio recording and playback**

The original signal is compared to a version recorded into the VAD system's RAM and then played back and recorded onto the PC.

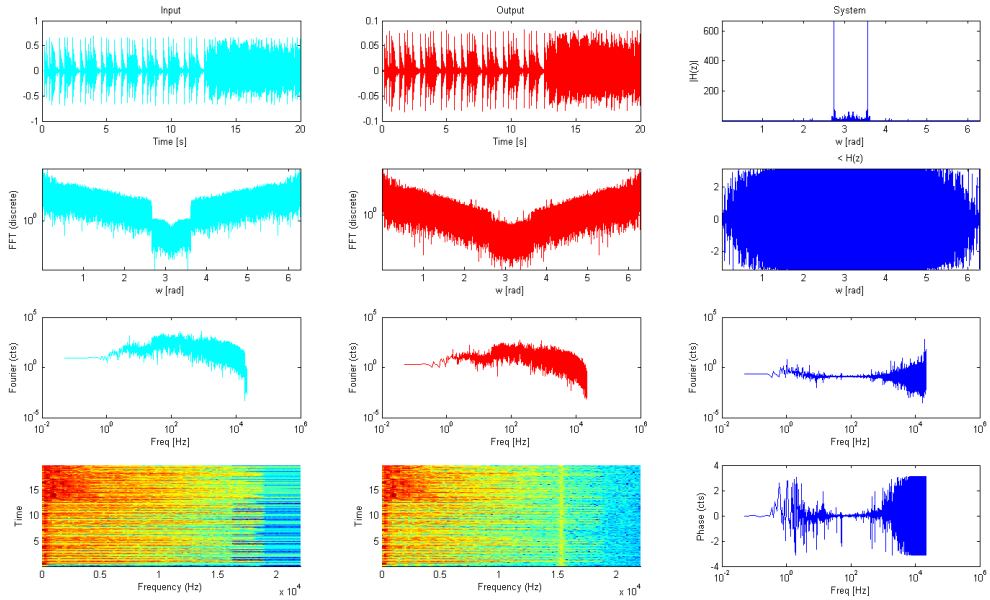


Figure 4.38: MATLAB Results for Test Song 1 in recording/playback VAD reconfiguration Design 1. The system magnitude appears quite well-behaved.

The song with the most amplitude underwent the least distortion, and the song with the least amplitude, the most.

Recording took around 54 seconds.

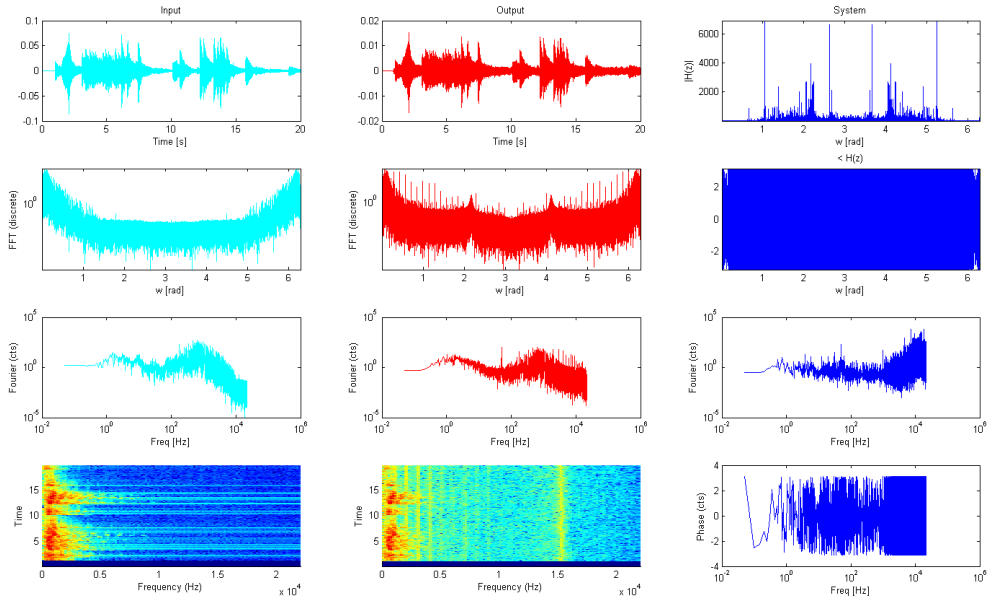


Figure 4.39: MATLAB Results for Test Song 4 in recording/playback VAD reconfiguration Design 1. This song has the least signal amplitude and undergoes significant phase distortion.

### 4.18.4 **Experiment 3: Monophonic oscillator**

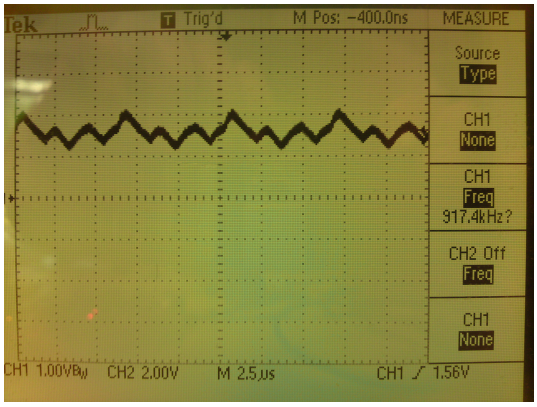


Figure 4.40: Oscilloscope Display of Sine Oscillator with Count of 1. The signal is destroyed due to aliasing. The question mark in the frequency readout indicates the frequency is not accurate.

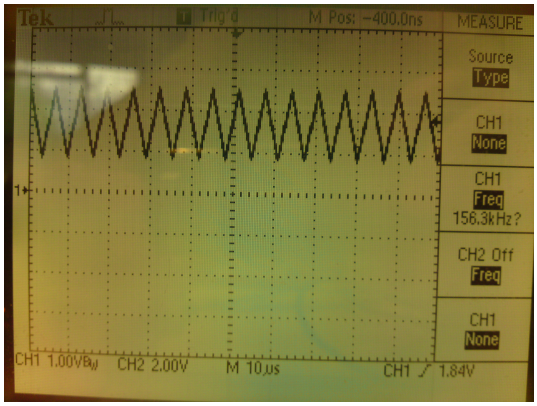


Figure 4.41: Oscilloscope Display of Sine Oscillator with Count of 5. The sine wave cannot reach the rails.

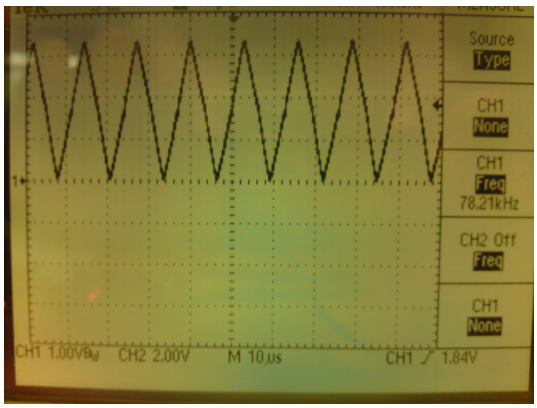


Figure 4.42: Oscilloscope Display of Sine Oscillator with Count of 10. The sine wave reaches the rails but the shape of the waveform is lost.

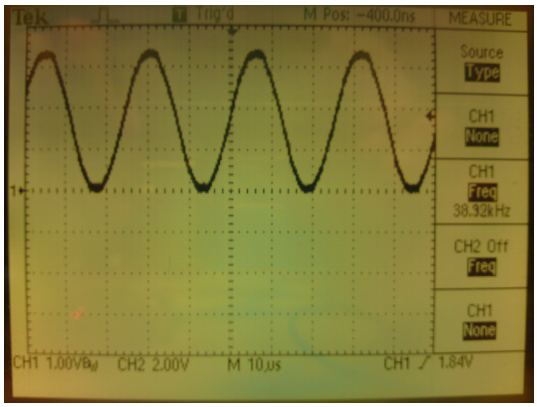


Figure 4.43: Oscilloscope Display of Sine Oscillator with Count of 20. A well-formed sine wave at a frequency of 38.9 KHz. The absence of a question mark in frequency readout indicates an accurate frequency.

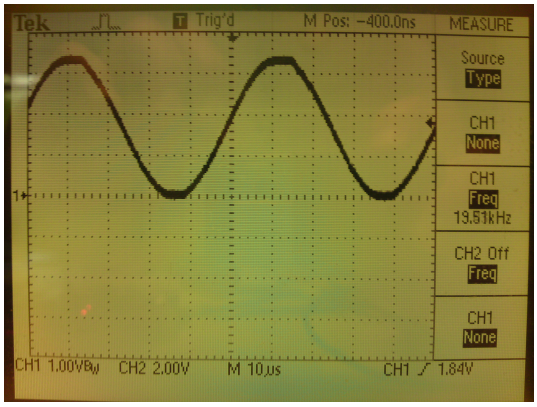


Figure 4.44: Oscilloscope Display of Sine Oscillator with Count of 40, frequency reading 19.51 KHz. This is the highest frequency in an audio environment.

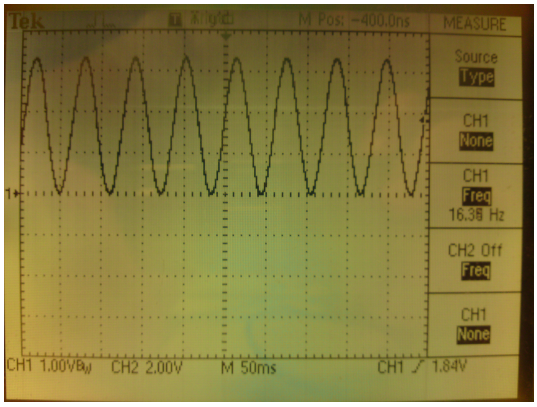


Figure 4.45: Oscilloscope Display of Sine Oscillator synthesising $C_0$, the lowest musical note [21], frequency reading 16.38 KHz.

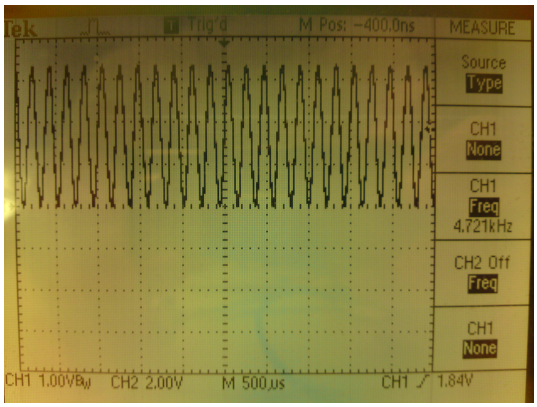


Figure 4.46: Oscilloscope Display of Sine Oscillator synthesising $D_8$, the second highest musical note [21], frequency reading 5 KHz.

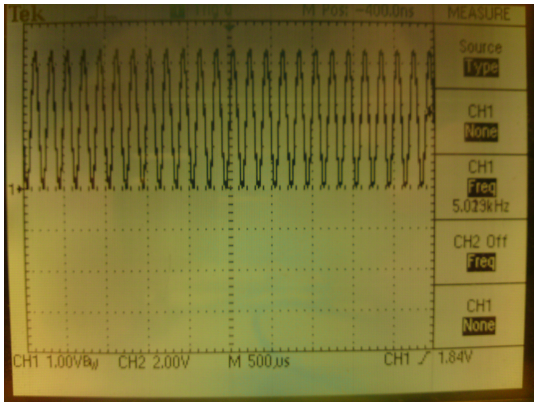


Figure 4.47: Oscilloscope Display of Sine Oscillator synthesising $D\#_8$, the highest musical note [21], frequency reading 5 KHz.

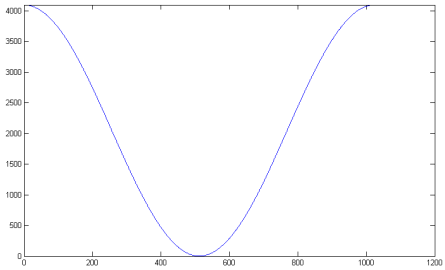


Figure 4.48: Graph of Sine Oscillator Phase Shifted to Become a Cosine. This was produced with a prototype oscillator which had 1024 samples instead of 64 and was clocked by the software. It was useful for transferring the precise waveform produced to the PC, but was poor as an accurate high-frequency oscillator.

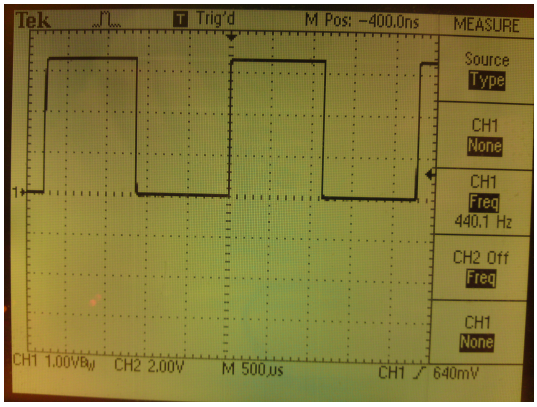


Figure 4.49: Oscilloscope Display of Square Oscillator synthesising $A_4$, frequency reading 440.1 Hz.

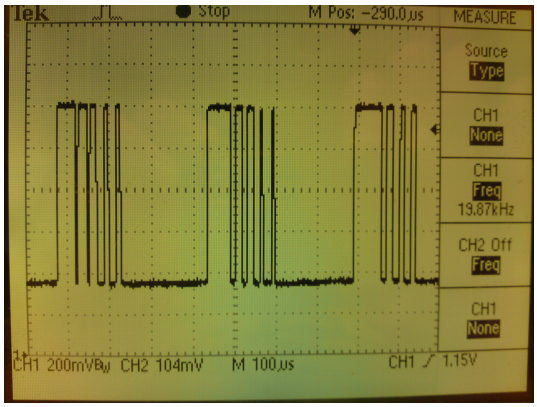


Figure 4.50: Oscilloscope Display of Square Oscillator Pulse Width Modulation using Low Frequency Saw Tooth implemented as Software Busy Loop.

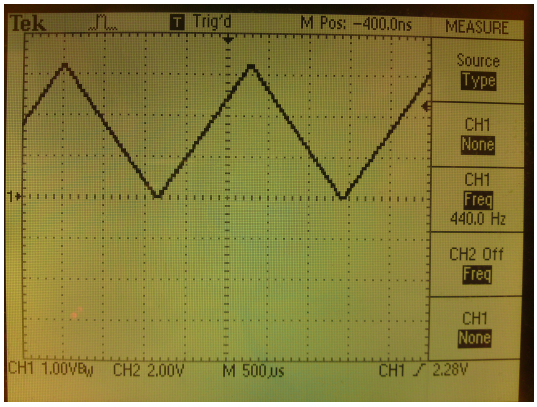


Figure 4.51: Oscilloscope Display of Triangle Oscillator synthesising $A_4$, frequency reading 440.0 Hz.

### 4.18.5 **Experiment 4: Polyphonic Synthesis**

Four sine oscillators are connected to a mixer, with the mixer connected to the DAC controller. Each sine oscillator is given an arbitrary frequency count. The output is recorded to a PC and analysed in MATLAB.

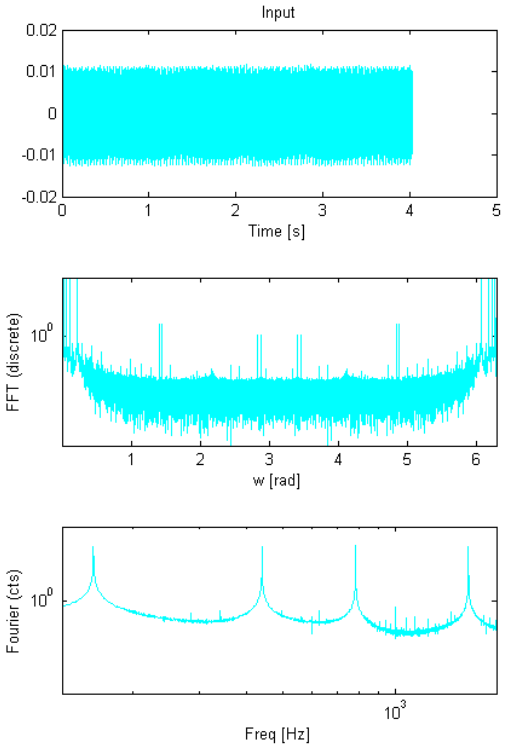


Figure 4.52: Oscilloscope Display of Triangle Oscillator synthesising $A_4$, frequency reading 440.0 Hz.

### 4.18.6 **Experiment 5: Prototype System Demonstration**

The VAD prototype has been demonstrated to perform the stated functionality.

It is difficult to quantify the accuracy of playing a MIDI song, since each MIDI device will synthesise a song differently. Similarly, it is difficult to quantatively prove that the prototype does perform all these tasks simultaneously. This is shown to be true through the presented results and prototype design, which illustrate how these functions are performed simultaneously.

CHAPTER 5

# CONCLUSIONS

## 5.1 DAC AND ADC SPI INTERFACES

### 5.1.1 DAC

Comparing results to the given timing diagram, the DAC SPI signals were well formed and functioned as expected, even at the fast serial clock rate of 25 MHz. The performance and decreased cost of the MCP4921 versus the DAC Pmod suggest that the MCP4921 is a good selection of DAC.

### 5.1.2 ADC

Comparing results to the given timing diagram, the ADC SPI signals appeared just to be working at 25 MHz, by the triangular nature of the clock pulse. It would be inadvisable to clock the ADC any faster. The system characterisation indicates that the ADC is working satisfactorily.

The adjustable gain generic of the ADC controller was found useful to produce an input signal which utilised the available bit width.

### 5.1.3 Hypothesis

The Nexys2 evaluation board is capable of interfacing effectively with multiple DAC's. However, since 20 MHz is not an integer factor of 50 MHz, the chips must be clocked at either 25 MHz or 16.67 MHz. Neither is ideal, though the MCP4921 DAC handles the faster clocking rate with greater success than the ADCS7476 ADC. Hypothesis H2 is mostly proven.

This increased serialisation/de-serialisation rate allowed the audio pipeline to be clocked at a faster speed of 1.25 MHz.

### 5.1.4 Future work

Future research into clocking the DAC and ADC at 20 MHz using the Digital Clock Manager IP and natives could be conducted.

A future system could have the DAC/ADC $SCLK$ rates specified by generics.

## 5.2 STRAIGHT-THROUGH SYSTEM CHARACTERISATION

### 5.2.1 Analysis of System

The straight-through system characteristic has been analysed using four songs. The songs with most amplitude tend to experience least distortion, which is expected in an electronic system, since the Signal to Noise Ratio decreases as the signal amplitude decreases, and so the effect of the noise is greater. The system then appears to introduce more distortion and thus to have a less desirable characteristic.

The presence of noise indicates that VAD is not a linear system, since its characeristic depends on the input amplitude, though it has been modelled as a linear system in this experiment. It is approximately a linear system, as can be seen by the more-or-less consistent system results. Its general effect is to amplify and phase-distort higher frequency components. In some cases, the entire signal is phase-distorted. These are not desirable characteristics for an audio system: a good audio filter has a continuous phase characteristic.

### 5.2.2 Hypothesis

This experiment partially proves hypothesis H0. The prototype has been shown to process 12-bit audio data at a rate of approximately 44.1 KHz. However, this frequency is not exactly generated since 44.1 KHz is not an integer factor of 50 MHz. The near frequency is generated in the sound buffer IP's, using a count of 1134 to produce a sampling rate of $\frac{50\ MHz}{1134} = 44.091KHz$, an error of $\frac{44100-44091}{44100} = 0.02\%$. Most importantly, the Nyquist sampling constraint for signals with frequency content of up to 20 KHz is still satisifed.

Though this suggets the prototype will gain time when used with an external device, in practice multiple audio devices must share a clock for synchronicity in any case.

The 15 KHz tone may have been due to noise or interference arising from the connection between the VAD device and the recording PC.

The system amplitude characteristic should be trusted over the phase characteristic, as the amplitude does not vary with shifts in time between the input and output signals, while the phase had to be manually aligned in Cubase. A future experiment should record both the input and output tracks simultaneously, adding credibility to the phase characteristic.

### 5.2.3 Future work

Research could be conducted into improving the phase response of the system.

### 5.2.4 Possible sources of error

The 15 KHz tone may have been due to noise or interference arising from the connection between the VAD device and the recording PC.

A future experiment should record both the input and output tracks simultaneously, removing the need for manual signal alignment in Cubase and adding credibility to the phase characteristic

results.

## 5.3 RECORDING SYSTEM CHARACTERISTIC

### 5.3.1 System Analysis

The recording system introduces an additional stage into the playback process. Since the values are digital, similar characterisations are expected of the recording system as were found in the straight-through system. This does appear to be the case. The recording appears to be reasonably faithful to both the original signal and the straight-through output version of the signal.

The phase characteristics indicate phase distortion, which again is undesirable for audio applications.

### 5.3.2 Hypothesis

This experiment proves hypothesis H0. The VAD system can be used to record and play back 12-bit audio at 44.1 KHz with an error of 0.02 %.

## 5.4 MONOPHONIC SYNTHESIS

### 5.4.1 Oscillator Frequency

The oscillator should not be clocked at a rate faster than $\frac{1.25\ MHz}{64\ samples} = 19.53\ KHz$, to avoid samples arriving too quickly to be loaded by the audio pipeline and thus producing aliasing. However, the sine oscillator is shown to produce a well-formed signal from a frequency of around 38 KHz. These results apply to the other oscillators, since the sine oscillator is the most complex.

***Error in Frequency Generation***

The oscillators are shown to produce accurate waveforms, as indicated by the absence of the question mark in the oscilloscope frequency readout. These are typically accurate to within a few Hertz. The accuracy decreases as the number of cycles to count decreases and the frequency rises. Therefore, the system synthesises low frequencies more accurately than high ones, which is to be expected.

The maximum error in frequency generation is 20 ns, since that is the duration of a system clock cycle.

### 5.4.2 Hypotheses

The musical frequency range of 16 Hz to 38 Hz of the system has been demonstrated, fulfilling hypothesis H1. The results show that the system's oscillators are correctly formed and may have parameters usually found in waveform generators, such as phase shift and adjustable duty cycle, fulfilling hypothesis H6.

## 5.5 POLYPHONIC SYNTHESIS

### 5.5.1 Analysis of Polyphonic Signal

The mixer IP is shown to work and combine functionality of diverse audio peripherals in a single reconfiguration. The generated sine waves are clearly visible in the Fourier analysis of the signal. Frequencies are specified as in the monophonic case, and it can be seen that the frequencies are generated as expected.

### 5.5.2 Hypothesis

The system has been shown to produce multiple musical notes simultaneously, with effective software control. This fulfils hypothesis H3.

## 5.6 FINAL PROTOTYPE RECONFIGURATION

The prototype is capable of playing samples and MIDI files and performing all stated functionality concurrently, fulfilling hypothesis H4. At this stage, the system has been reconfigured numerous times, proving that the system is highly reconfigurable and fulfilling hypothesis H5.

## 5.7 CONCLUSION

### 5.7.1 Thesis Validation

By thoroughly investigating the initial hypotheses and functionality of the system, the work performed in this thesis project has been validated.

A general-purpose audio device for the Nexys2 evaluation board has been designed, implemented and tested. The VAD system has been shown to meet the requirements as outlined in Chapter 1.

### 5.7.2 Future Work

It is my sincere hope that this thesis will form an instructive basis and audio prototyping framework for the Nexys2 evaluation board.

A future system could feature a wide variety of audio peripherals and be software-configurable, using an audio programming language such as Chuck.

A commercial synthesiser such as Reason Subtractor could be modelled on the VAD system.

***Future Features***

- Low-Pass Filter LPF and other DSP peripherals.
- Portamento and other note frequency based effects.
- Gain and volume control peripherals, such as an ADSR envelope.

- Bit-depth manipulation peripherals, such as bit crushers and expanders.

APPENDIX A

# INCLUDED CD

The CD included with this thesis has the following content:

- Electronic copy of thesis.
- EDK project directory for final system design.
- Bitstreams for VAD reconfigurations.
- Cubase audio files.
- Matlab spectrograms.
- Ancillary source code.
- Assorted electronic documentation.

APPENDIX B

# LOADING DATA ONTO THE PROTOTYPE

The wave files are converted to a single text file with numerical data. A Python script then packages the data into a binary file containing the raw audio data in 8-bit PCM big-endian format. This is the sampler file.

A 5-track MIDI file is prepared using Cubase. MIDI tracks 1 and 3 have four and two notes of polyphony, respectively. Track 10 is the drum track.

The sampler file and MIDI track are joined using the Windows command prompt.

Matlab script

```
function [fid] = toFile(fid,x)
    range = 2^8;
    x = range .* x;
    x = range/2 + x;
    x = floor(x);
    N = 44101;
    for i = 1:N
        fprintf(fid, '%d\n', x(i));
    end

function [x,Fs] = sampler()
    [bass,Fs]= wavread('bass1.wav');
    [bells,Fs] = wavread('bells.wav');
    [snare1,Fs] = wavread('snare1.wav');
    [snare2,Fs] = wavread('snare2.wav');
    [hihat1,Fs] = wavread('hihat1.wav');
    [hihat2,Fs] = wavread('hihat2.wav');
    [crash,Fs] = wavread('symbol.wav');
    [clap,Fs] = wavread('clap.wav');
```

```
    fid = fopen('sample.txt', 'w');
    toFile(fid,bass);
    toFile(fid,snare1);
    toFile(fid,snare2);
    toFile(fid,hihat1);
    toFile(fid,hihat2);
    toFile(fid,bells);
    toFile(fid,crash);
    toFile(fid,clap);

    fclose(fid);
```

Python script

```
import struct

textfile = open("sample.txt", 'r')
binfile = open('sampler.dat', 'wb')
i = 0
for line in textfile.readlines():
    i += 1
    if i < 25:
        print int(line)
    # unsigned char
    data = struct.pack('>B', int(line))
    binfile.write(data)
textfile.close()
binfile.close()
```

Windows Command Prompt

```
copy \b sampler.bin+song.mid song.vad
```

Each song may have its own samples associated with it. This is similar to the MOD file format which was popular on the Amiga.

The resulting file is loaded onto the prototype using a serial port terminal application such as *RealTerm*.

APPENDIX C

# MATLAB ANALYSIS SCRIPT

The Matlab script using fundamental principles of Digital Signal Processing (DSP) theory to analyse the affect of the VAD system on an input signal based on a 'clean' and a 'dirty' signal.

```
function [R,Fs] = characterise(file1, file2)
    % read file
    [x,Fs] = wavread(file1);
    [y,Fs] = wavread(file2);

    N = size(x, 1); % length of vector
    % find time axis
    n = (1:N);
    T = n./Fs;

    % plot signals in time domain
    subplot(4,3,1), plot(T, x, 'c'), xlabel('Time [s]'), title('Input')
    subplot(4,3,2), plot(T, y, 'r'), xlabel('Time [s]'), title('Output')
    % N-point DTFT
    X = fft(x);
    Y = fft(y);
    % scale horizontal axis to find relevant transforms from fft
    % find little omega axis (2pi periodicity)
    waxis = (n./N)*(2*pi);

    piEquiv = N/2;
    n = (1:piEquiv);
    Waxis = (n./piEquiv)*(Fs/2);

    % only need until highest freq (pi) to describe in cts time
    Xcts = X(1:piEquiv);
    Ycts = Y(1:piEquiv);
    % plot Fourier
    subplot(4,3,4),semilogy(waxis,abs(X), 'c'), xlabel('w [rad]'),
```

```
   ylabel('FFT (discrete)'),axis([-Inf 2*pi -Inf Inf])
subplot(4,3,5),semilogy(waxis,abs(Y), 'r'), xlabel('w [rad]'),
   ylabel('FFT (discrete)'), axis([-Inf 2*pi -Inf Inf])
% plot Fourier in terms of Hertz (continuous-time Fourier)
subplot(4,3,7), loglog(Waxis, abs(Xcts), 'c'), xlabel('Freq [Hz]'),
   ylabel('Fourier (cts)')
subplot(4,3,8), loglog(Waxis, abs(Ycts), 'r'), xlabel('Freq [Hz]'),
   ylabel('Fourier (cts)')
% find frequency response
H = Y./X;
% plot magnitude/phase
subplot(4,3,3),plot(waxis, abs(H)), title('System'),
   ylabel('|H(z)|'), xlabel('w [rad]'), axis([-Inf 2*pi -Inf Inf])
subplot(4,3,6),plot(waxis, angle(H)), title('< H(z)'),
   xlabel('w [rad]'), axis([-Inf 2*pi -Inf Inf])

Hcts = H(1:piEquiv);

subplot(4,3,9), loglog(Waxis, abs(Hcts)),
  xlabel('Freq [Hz]'), ylabel('Fourier (cts)')
subplot(4,3,12), semilogx(Waxis, angle(Hcts)),
  xlabel('Freq [Hz]'), ylabel('Phase (cts)')

subplot(4,3,10),spectrogram(x,256,250,256,Fs);
subplot(4,3,11),spectrogram(y,256,250,256,Fs);
```

APPENDIX D

# SOFTWARE POLYPHONY

These software macro's are responsible for assigning musical notes to oscillators, when more than one oscillator is available. These oscillators are contiguous in the memory map.

```
#define SINE_OSCILLATOR_SET_COUNTS(osc, counts) \
SINE_OSCILLATOR_mWriteSlaveReg1(BASEADDR + 0x100000*osc, 0, counts)

#define SAW_OSCILLATOR_SET_COUNTS(osc, counts) \
SAW_OSCILLATOR_mWriteSlaveReg1(BASEADDR + 0x100000*osc, 0, counts)
```

APPENDIX E

# SOFTWARE MIXING

These software mixes together the playing samples using point-wise addition.

```
// calculate NEXT sample value
i = 0;
N = 0;
for (j = 0; j < N_DRUMS; j++) {
     if (drums[j] != 0) {
        i += ((Xuint32)RecvBuffer[j*SAMPLER_SAMPLE_SIZE+drumIndex[j]])
            *drums[j]/64;
        N++;
        // move to next sample in this drum track
        drumIndex[j]++;
        if (drumIndex[j] == SAMPLER_SAMPLE_SIZE) {
            // switch note off
            drums[j] = 0;
        }
    }
}
if (N != 0) {
    i /= N;
}

if (recordingPlay == 1) {
    i += recordSample;
    i /= 2;
    // iterate to the next sample
    recordIndex = (recordIndex + 1) % SAMPLE_SIZE;
    // load the next sample
    recordSample = sample1[recordIndex];
}
```

APPENDIX F

# RS-232 INTERFACE

The Uart 1650 RS-232 interface code is taken from a sample Microblaze application.

```
int uart_init() {
    int Status;
    u16 Options;
    Status = XUartNs550_Initialize(&UartNs550, UART_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    Status = XUartNs550_SelfTest(&UartNs550);
    if (Status != XST_SUCCESS) {
         return XST_FAILURE;
    }
    Options = XUN_OPTION_FIFOS_ENABLE;
    XUartNs550_SetOptions(&UartNs550, Options);

    return XST_SUCCESS;
}

void uart_receive() {
    unsigned int ReceivedCount = 0;
    while (1) {
        ReceivedCount += XUartNs550_Recv(&UartNs550,
       RecvBuffer + ReceivedCount,
       TEST_BUFFER_SIZE - ReceivedCount);
        if (ReceivedCount == TEST_BUFFER_SIZE)
        {
            break;
        }
    }
}
```

```
// in main...
status = uart_init();
if (status != XST_SUCCESS) {
    return XST_FAILURE;
}

// 115200-8-N
XUartNs550_SetBaud(BASEADDR, XPAR_XUARTNS550_CLOCK_HZ, 115200);
XUartNs550_SetLineControlReg(BASEADDR, XUN_LCR_8_DATA_BITS);
```

APPENDIX G

# PULSE WIDTH MODULATION

The following code will modulate the square wave duty cycle to produce pulse width modulation about a given frequency. This snippet must be called at regular intervals.

```
pwm = ((pwm+1) % 30)+4;
SQUARE_OSCILLATOR_mWriteSlaveReg0(BASEADDR, 0, pwm);
```

# BIBLIOGRAPHY


[1] COELHO, D. R. *The VHDL handbook*. Kluwer Academic Publishers, 1989.

[2] CREASEY, D. J. *Advanced signal processing*. Peter Peregrinus Ltd., 1985.

[3] DECALUWE, J. These ints are made for countin'. Tech. rep., Jan Decaluwe, 2009.

[4] DIGILENT. Pmodamp1 schematic. Tech. rep., Digilent, 2007.

[5] DIGILENT. Pmodda2 reference ise vhdl component. Tech. rep., Digilent, 2007.

[6] DIGILENT. Nexys2-500 reference manual. Tech. rep., Digilent, 2008.

[7] DIGILENT. Adept application users manual. Tech. rep., Digilent, 2009.

[8] DIGILENT. Nexys2 edk board support files release notes. Tech. rep., Digilent, 2009.

[9] DIGILENT. Digilent plug–in for xiliinx 12.x tools user manual. Tech. rep., Digilent, 2010.

[10] DIGILENT. Mcs file creation with xilinx ise tutorial. Tech. rep., Digilent, 2010.

[11] HICKOK, T. Cpe 329: Programmable logic and microprocessor-based system design laboratory 4. Tech. rep., California Polytechnic State University, 2007.

[12] LACH, J., MANGIONE-SMITH, W. H., AND POTKONJAK, M. Low overhead fault-tolerant fpga systems.

[13] LEWIS, J. Vhdl math tricks of the trade. Tech. rep., SynthWorks, 2003.

[14] LIDDICOAT, A. A., AND SLIVOVSKY, L. A. Xilinx embedded development kit (edk) 9.1i nexys/nexys 2 tutorial. Tech. rep., California Polytechnic State University, 2008.

[15] MANO, M. M., AND KIME, C. R. *Logic and Computer Design Fundamentals*. Pearson Education, Inc., 2008.

[16] MARTINEZ, D. R., BOND, R. A., AND VAI, M. M. *High Performance Embedded Computing Handbook, a Systems Perspective*. CRC Press, 2008.

[17] MICROCHIP. Mcp4921 datasheet. Tech. rep., Microchip Technology Inc., 2004.

[18] NAVABI, Z. *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill, 1998.

[19] SEMICONDUCTOR, N. Adcs7476. Tech. rep., National Semiconductor, 2003.

[20] SPOT, T. S. Midi file format. Tech. rep., The Sonic Spot, 2007.

[21] SUITS, B. H. Physics of music - notes. frequencies for equal-tempered scale. Tech. rep., Michigan technological university, 1998.

[22] SWAN, R., WYATT, A., CANT, R., AND LANGENSIEPEN, C. Reconfigurable computing. *Crossroads 5 (3es)*.

[23] WASHINGTON, L. Xilinx co-founder ross freeman honored as 2009 national inventors hall of fame inductee for invention of fpga. Tech. rep., Xilinx, 2009.

[24] XILINX. Xup lab 3 - powerpc processor adding custom ip to an embedded system lab. Tech. rep., Xilinx.

[25] XILINX. Edk concepts, tools and techniques (ctt). Tech. rep., Xilinx, 2009.

[26] XILINX. Embedded system tools reference guide edk 11.3.1. Tech. rep., Xilinx, 2009.

[27] XILINX. Microblaze v7.20 faq microblaze soft processor v7.20 frequently asked questions. Tech. rep., Xilinx, 2009.

[28] XILINX. Xilinx ise webpack vhdl tutorial. Tech. rep., Xilinx, 2010.