GHOST synth

Implementing flexible sound card functionality for SoCs

April 30, 2019

Austin Liolli
Marco Merlini
Mat Hildebrand
Sriharsha H S

Table of Contents

1. Introduction

This report details the design of an SoC-based synthesizer. The processor system (PS) is required to transmit "note on" and "note off" commands, as well as to configure audio voices and sound effects. Figure 1.1 shows an overview of the design.

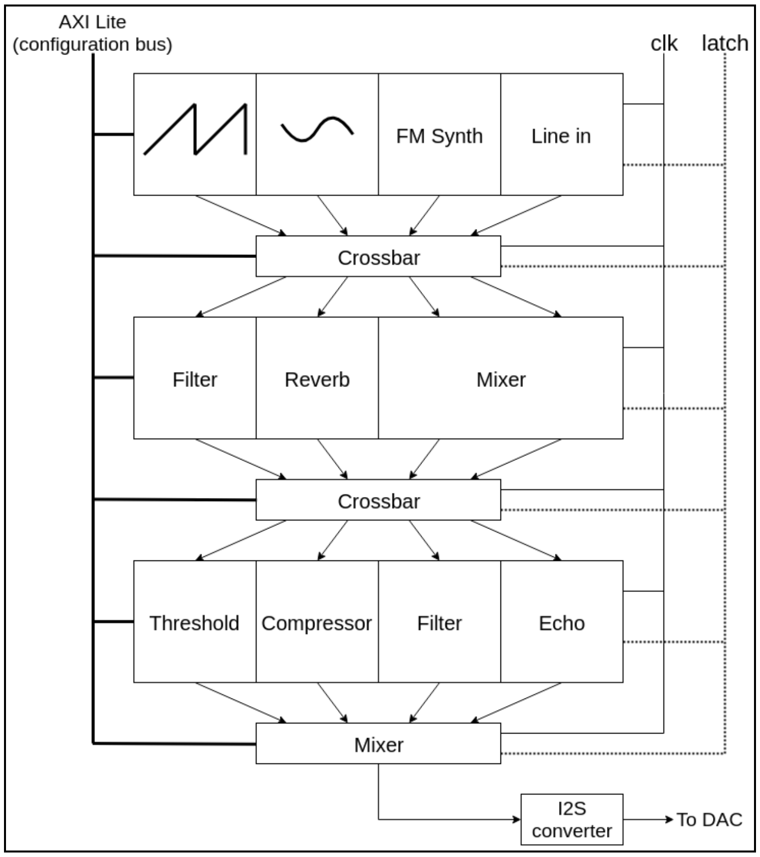


Figure 1.1:. Overview of proposed FPGA synthesizer design

The synthesizer is designed in layers of *audio modules*. Contained in the top layer are all the basic sound generation modules; these accept commands that turn notes on or off, as well as commands to select pitch and timbre. Between each layer there is a crossbar, to allow the PS to connect modules to arbitrary effects. Each effect may also expose configuration registers (e.g. the echo effect could have a configurable delay time).

2. Current Status

The current status of the design has met all requirements established at the outset of the project. The design is currently targeting a Zynq Mini-ITX hardware platform containing a Xilinx Kintex-7 SoC. The project currently uses Vivado 2017.2 for its FPGA development platform, with a petalinux BSP sourced from Vivado 2016.2. The currently implemented high-level synthesis (HLS) audio synthesis modules are as follows:

- Sawtooth tone generator
- FM synthesizer
- White noise generator
- Digital filter
- Echo/Reverb
- Mixer
- Envelope generator
- Tremolo
- Vibrato
- Sample latcher

With the chosen architecture these modules can be configured at runtime via memory-mapped registers and combined in (almost) any order with the use of AXI Stream crossbars. The current system in block diagram form can be seen in Figures 4.1, 4.2, and 4.3.

To showcase the system, a program with the ability to play arbitrary MIDI files has been included. This program is capable of intelligently mapping polyphonic sounds to four saw wave voices, to demonstrate a subset of the features within the audio synthesizer.

Future improvements to this design could include:

- More sophisticated MIDI playback
    - The standard MIDI file format includes special codes for configuring the voice of the playback instrument. Currently, these are ignored.
    - The MIDI playback engine could be modified to accept a configuration file with information on audio modules and their memory-mapped registers. This would allow new modules to be added to the FPGA without modifying the MIDI playback source code.
- Audio module improvements
    - The audio modules are clocked several hundred times faster than the audio sampling rate; as a result, they are often idle. They could be expanded to produce multiple channels of audio, thus increasing the number of simultaneous voices/effects possible in the synthesizer

    - We could also simply add more modules. For example, each effects layer could be duplicated. Additionally, we could support operations such as ducking, chorus, distortion, etc.
- FPGA shell improvements
    - The hardware allows for a microphone or line in input, this could be leveraged fairly easily with some additional code in the I2S core as well as adding a data path for the i2s block to act as another source alongside the sawtooth and fm synthesizer.
    - The AXI Stream Switch IPs concatenate all of the output streams together, resulting in an ultra-wide bus. Either implementing a custom configuration for independent small width data buses, or reconfiguring the crossbar would reduce timing congestion.

3. Initial Architectural Design

3.1 Design Specification

The initial goal of this project was an FPGA-based digital synthesizer that can accept simple run-time commands from a processor by exposing memory-mapped configuration registers. These run-time commands would then control both audio source and effect modules, as well as which components the audio flows through. The synthesizer would then generate a stream of digital samples at a 48 kHz rate, to be output through an audio CODEC. The rest of this section details the initial design requirements of the audio modules, interconnect and data flow between the modules, data flow to the external CODEC and run-time configurability of the audio cores.

System Specifications

- Audio samples must be produced by the FPGA at a 48kHz sampling rate, with Time Domain Multiplexed (TDM) left justified mode communicating to the audio CODEC (Appendix C).
- The core clock of the device must run at a frequency to match the Zynq processor interconnect to avoid additional clock crossing in the design.
- The user is able to manipulate the configuration of audio modules through the use of scripts or command line arguments to modify the audio stream.
- There will be multiple audio sources including an FM synthesizer, wave generators and noise generator.
    - The wave generators must have a register interface that allows the user to set the pitch, while the FM synth will also have a modulation frequency.
- There will be multiple audio effect modules which will modify the stream of audio samples. These will be located downstream from the audio sources and be interconnected with simple NxN crossbars.
    - These effects modules must include frequency filters, echo/reverb, tremolo and audio mixers.
- The final output must pass through an envelope and compressor module to ensure a non clipping and user controllable audio level.
- The design must include at least an FM synthesizer, digital filter, echo, and enough other generators/effects (decided on the fly by the development team) to total at least 10 unique modules

Note that the choice of modules in the final design was not specified at the outset of the project. As development went on, team members would have ideas for new modules to add to the design. Often, choices were made based on the difficulty of implementing a particular module versus the amount of time remaining. The FM synthesizer is one of the more complex modules in the project, and was thus started very early.

3.2 Clocking

The first two specifications indicated the need for two independent clocks in the FPGA design. One to match the Zynq processing system for the memory mapped interconnect system (no clock crossing required) and another to drive data out to the audio CODEC at the rate specified by its datasheet [1].

The ADAU1761 accepts a wide variety of clocking rates, but the master clock (MCK) rate that provides the fewest number of configuration changes on the audio CODEC is 12.288 MHz. However, 12.288 MHz is not the clock that drives data or any control signals into the audio CODEC, in fact it requires two more clocks. These clocks are known as the bit clock (BCK) and the word clock (LRCK) and they were determined to be 3.072 MHz and 48 kHz respectively. They work in tandem with the MCK to properly drive the audio CODEC. These clocks as well as their decided frequencies are discussed more in Appendix C.

3.3 Audio Module Interconnect and Dataflow

To offer a wide range of possible sounds, it was decided to allow a configurable connection between modules. In order to get $O(n^2)$ interesting combinations from only $O(n)$ modules, an initial design using a crossbar was proposed (shown in Figure 3.3.1).

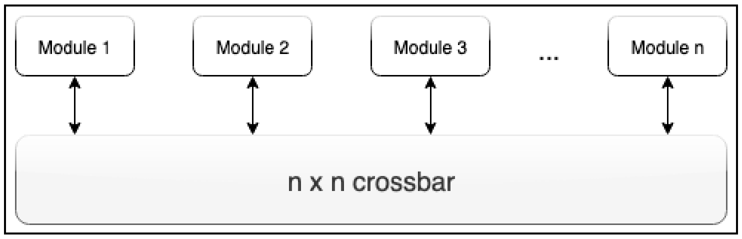


Figure 3.3.1: Original module interconnect design

Additionally, it was necessary to synchronize audio sample generation to a slower clock, such as 48 kHz. Originally, each sound operator i would be a simple state machine with n_i states, where the first $n_i - 1$ states would perform the necessary computations, and the n_i'th state would simply wait for a synchronization signal before transitioning back to state 1, effectively acting as a blocking synchronization state.

3.4 Data flow to external codec

The Integrated Inter-IC Sound Bus (I2S) core is a fundamental component of the design itself. Without it, no audio can be sent to the audio CODEC. As discussed in section 3.1, three clocks were required to drive the audio CODEC. First, a PLL was added to the design to generate the 12.288 MHz MCK. Then, to generate the BCK and LRCK, a module called an audio timing generator was developed in verilog. This module simply takes in the MCK and provides a rising edge aligned BCK and LRCK as derivatives of the input MCK. This module could not be produced using HLS due to a clock-cycle accurate requirement as it generates new clocks to be used elsewhere in the system.

Furthermore, after reading through the ADAU1761 datasheet, it was determined that the audio sample quality would be 24 bits, rather than 32, 16 or 8. This is because 32 bits is not supported with the codec and 16 or 8 bits does not give a good quality audio source. More information on the sample quality can be found in Appendix C.

Initial design had specified the use of a pre-existing I2S core, such as the one developed by Lattice Semiconductor [3]. This was a core designed in Verilog, but with a wishbone interface for both samples and register configuration that would need to be ported to Axi stream and Axi-lite protocols for data and configuration respectively.

As discussed in the clocking section above, the audio CODEC consumes data by the bit clock (3.072 MHz), thus the I2S core can also run at this frequency when producing audio samples. However, the control registers and incoming audio samples are clocked at the rate defined by the processor system. In order to handle this, the design called for a clock crossing module for the register map, as well as a clock crossing FIFO for the audio samples. It is for these reasons as well as the complicated components within the latticesemi I2S core design that the core was re-written from the ground up for the final design.

3.5 Initial Architecture Drawing

From the sections above, the individual requirements were met, but to ensure the cohesive system met all requirements, an initial architecture drawing was sketched up to validate the entire design. As seen from Figure 3.5.1, the ARM processor is connected via a command interpreter to all of the audio modules, along with a single large crossbar that supports looping audio module output back through to other audio modules to their input. The final output is then transferred to an I2S core and then to the DAC.

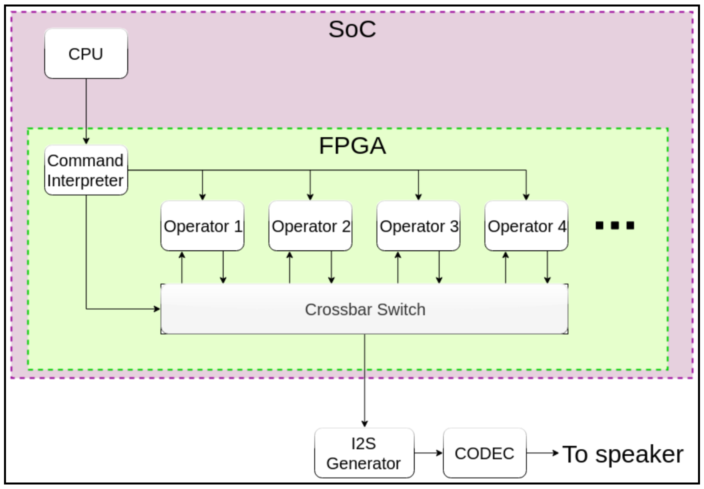


Figure 3.5.1: Initial Architecture Drawing Diagram

4. Final Architectural Design

The final architecture followed the spirit of the initial design, but with fine grain detail changes (such as the AXI stream crossbar setup). This caused subtle differences in functionality and architecture, yet no changes occurred in the projects initial requirements. Figure 4.1 illustrates a high-level overview of the final design. The I2S *and Timing* block is expanded upon in Figure 4.2, and the *Audio Modules* block is expanded upon in Figure 4.3.

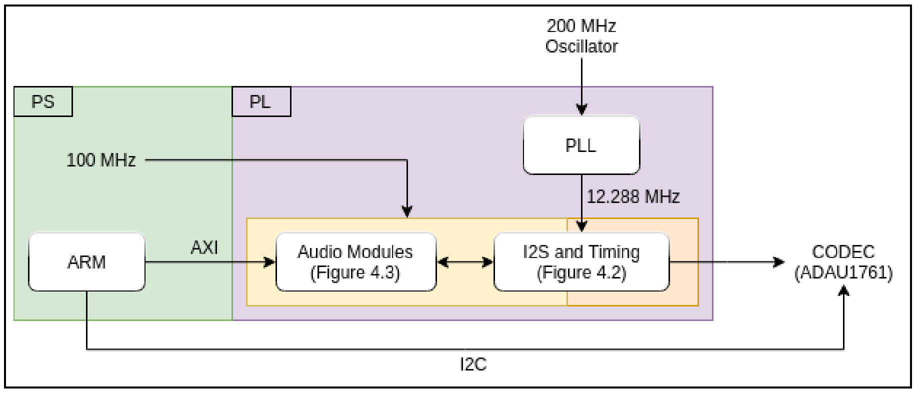


Figure 4.1: High-level overview of final design. The ARM processor (running Linux) sends commands to the audio modules over an AXI bus, and configures the CODEC over an I2C bus. The audio modules reside in a 100 MHz clock domain (indicated with a yellow background) and communicate to each other via an AXI stream protocol (not shown). The output audio sample stream is converted to an I2S protocol and sent to the audio CODEC. The generation of I2S signals is handled in a 12.288 MHz clock comain (indicated with an orange background).

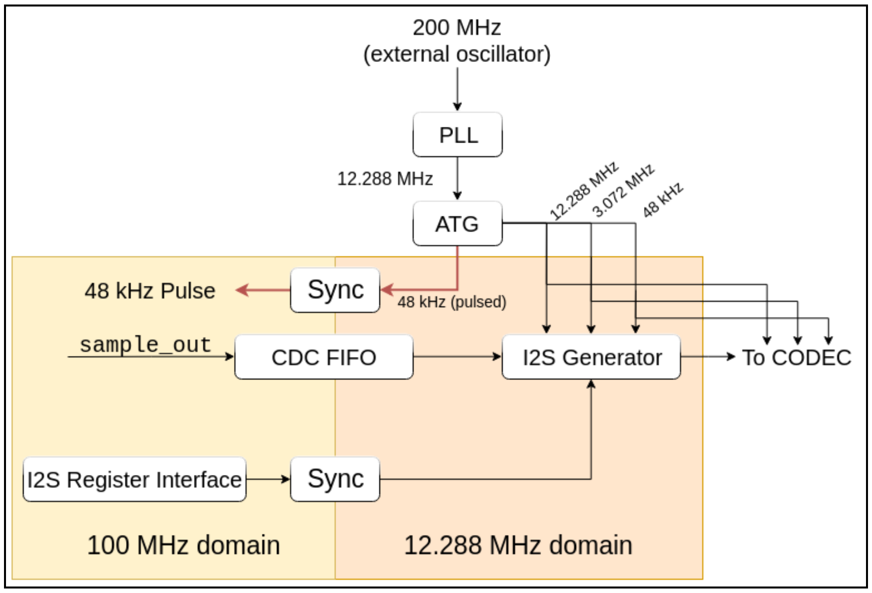


Figure 4.2: "I2S and Timing" portion of Figure 4.1, illustrating the generation of a 48 kHz pulse and conversion to I2S. An external 200 MHz is fed into a PLL which generates a 12.288 MHz clock. This clock feeds an Audio Timing Generator (section 4.1), which produces 3.072 MHz and 48 kHz clocks, as well as a 48 kHz pulse. The 3.072 MHz is the frequency of the I2S protocol, and the 48 kHz is the audio sampling frequency.

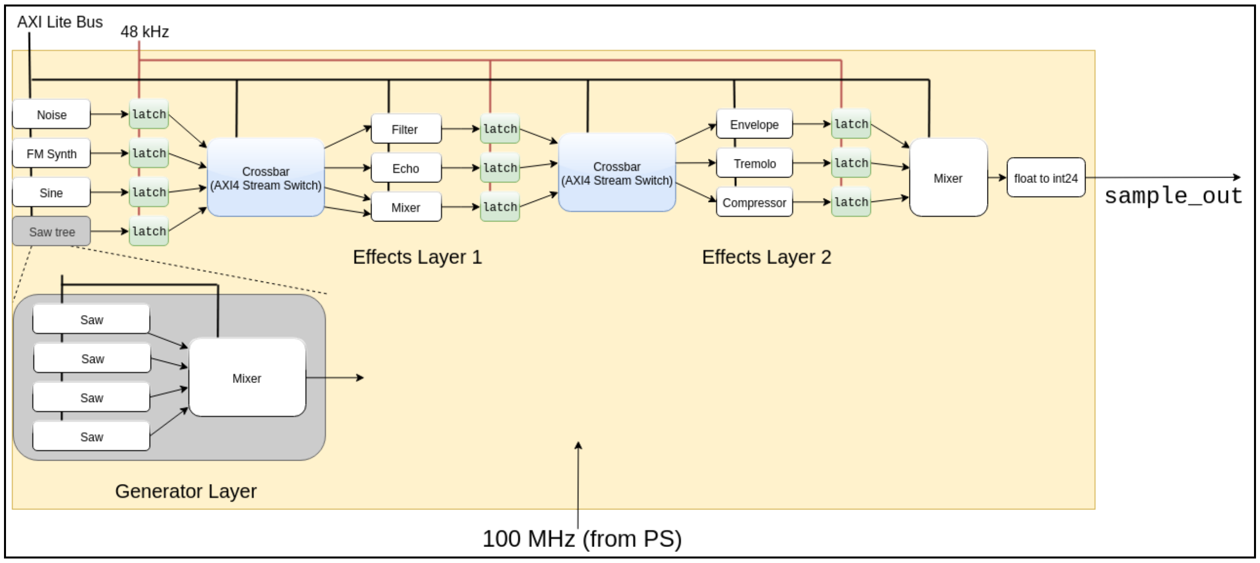


Figure 4.3: Audio Modules portion of Figure 4.1. Each left-to-right arrow represents an AXI stream protocol. Each audio module is followed by a **`latch_wait`** *module to synchronize audio sample generation (discussed in section 4.5). Additionally, a more flexible interconnect is implemented with the use of crossbars (discussed in section 4.4).*

All components of the above figures are discussed in detail below. We first describe the overall structure and clocking decisions as seen in Figure 4.1. Following the overall structure, the I2S core is discussed, as well as the modules required for it to properly drive the audio CODEC (Figure 4.2). Then, we briefly examine the design of the audio module interconnect and synchronization (Figure 4.3).

Once all of the required infrastructure is discussed, the rest of this section is used to present the HLS audio modules chosen for development as well as the demo to drive them. The final design required only a *variety* of modules, as opposed to a specific set. Thus, the team members were free to pursue modules they found interesting (provided they could be finished within a reasonable amount of time).

N.B. - Except for I2S and timing generation (created using Verilog), as well as the AXI Stream Switch and the Zynq processor IP (provided by Xilinx), all IP cores were created using Vivado HLS.

4.1 Structure and Clocking

As seen from Figure 4.1, the design is broken down into two clock domains as specified in the initial design. The core clock used for the final design is 100 MHz, it drives all register interfaces for the design, as well as driving the data flow in and out of all of the HLS audio modules. This 100 MHz clock is provided by the Zynq processing system (PS) and also drives the processor system axi interconnect. As the main FPGA reset is also provided by the PS, a specialized reset synchronizer block is used to provide ***`RESETN`*** signals to all of the modules. This reset synchronizer block is driven from control signals provided by the Zynq PS.

The second clock domain (12.288 MHz) drives all of the I2S core related modules. The clock is sourced from a mixed-mode clock manager (MMCM) PLL block provided by Xilinx, while its associated ***RESETN*** is provided by the locked output pin from the PLL. The MMCM PLL was chosen over a basic PLL implementation as it provides a much more fine grain clock divider capability to synthesize non-integer multiple clocks as outputs [4]. The second important detail about this clock domain is its PLL is driven by an external oscillator, running at 200 MHz. After investigating the Mini-ITX schematic(See Appendix C), the external oscillator input into the chip was traced to an external pin and a new clock constraint set was added to the design using said pin.

At the outset of the design phase, the PLL was to be driven from the same 100 MHz clock that is used for the first clock domain. However, as the design progressed, some issues were raised about sourcing a PLL with another PLL output (from the Zynq) so the 200 MHz clock was selected to guarantee a stable clock with minimal jitter.

4.2 Processing System Configuration

After investigating the audio CODEC provided with the hardware (Analog Devices ADAU1761), it was determined that configuration of the chip was required over an I2C bus. As seen from Figure 4.2.1, the ADAU1761 provides a very complicated digital-to-analog (DAC) audio path [1]. For this reason, I2C capability was added from the Zynq PS to directly control the I2C bus, rather than the FPGA. Furthermore, PJTAG was added to the MIO pin set in order to facilitate chipscope when debugging issues.

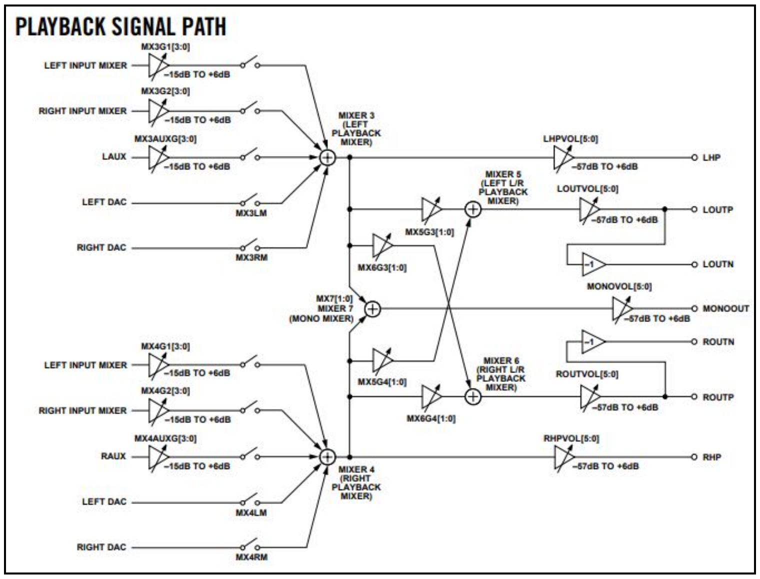


Figure 4.2.1: ADAU1761 register-configurable DAC path. [1]

4.3 FPGA Shell and I2S core

As an AXI4 peripheral interconnect block was already provided by the golden reference design, the register interfaces for all of the HLS and verilog modules could be connected fairly easily. Also, with the addition of the MMCM PLL mentioned above, both clock domains were properly set up. However, modules such as an audio timing generator (ATG), clock crossing synchronizers and an I2S core were still required to finish the FPGA shell design.

Initially, the ATG was required to act as a simple clock divider. The original requirements called for a 12.288MHz clock domain, with 3.072MHz and 48kHz clocks being derived from it. This is simple as the clocks required are exactly 1/4th and 1/256th of the clock domain they are in. However, one added requirement was a 48kHz single-cycle pulse that must be generated by the ATG to drive all of the HLS audio modules (Section 4.5). As all of the HLS audio modules were clocked at 100MHz, custom clock crossing modules were added to the ATG to synchronize the pulse to the 100MHz domain.

As mentioned in section 2, the final implementation does not use a pre-existing I2S core. Instead, a new I2S core was designed to better fit the clock crossing and AXI compatibility requirements set out by the initial design specification. The core itself was a straightforward design, following the I2S specification seen in Appendix C. However, the register configuration functionality used a third party register file generator called AirHDL [5]. This register file would be a submodule within the I2S core, run at 100MHz and allow the Zynq processing system to configure the I2S core at run-time. These register configuration values would then be sent over a clock crossing interface to be used by the I2S core.

Finally, for the audio sample interface, a clock crossing FIFO was implemented on the input to the I2S core. The FIFO depth was arbitrarily set to 64 words, but in reality the FIFO was designed to only use one or two spaces at any given time. The final shell implementation can be seen in Figure 4.3.1, where the grey module is the actual synthesizer application designed by the rest of the team.

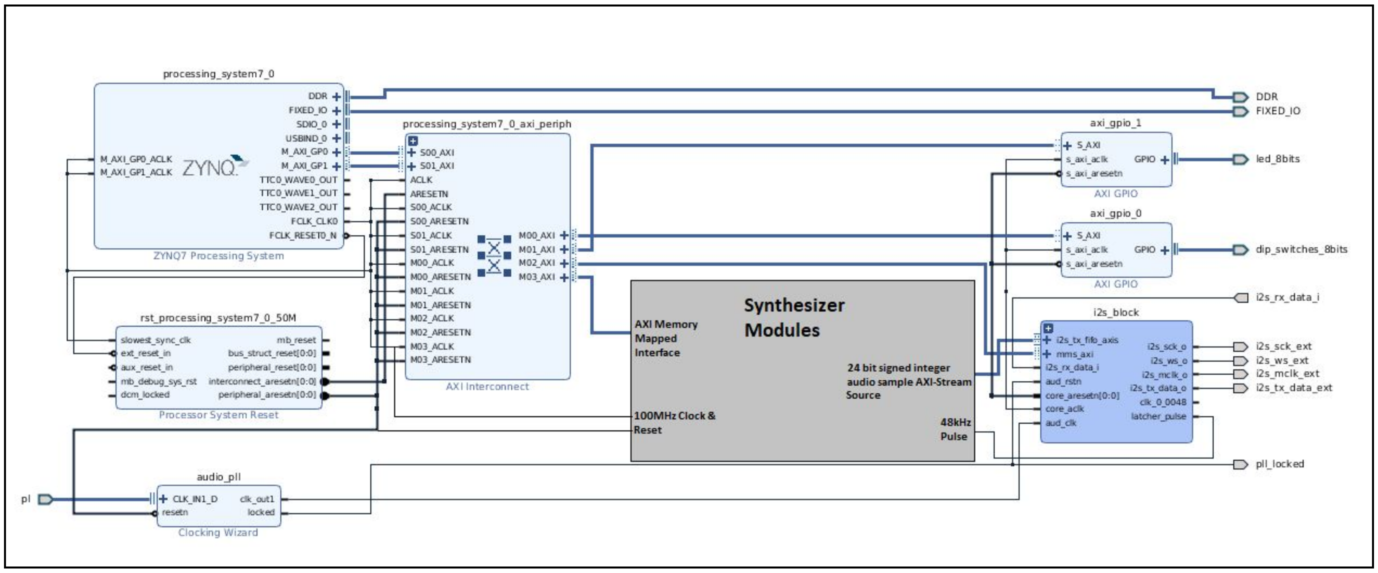


Figure 4.3.1: FPGA shell

4.4 Audio Module Interconnect

Our early design for audio module interconnect was a single $n \times n$ crossbar connecting all n modules (Figure 2.3.1). While this offers a fully configurable interconnect, it is generally expensive to place large crossbars into an FPGA due to its large muxing and bus width found later on in the design phase. The improved design is shown in Figure 4.3, where the modules are split into smaller groups and full crossbars are only placed between these groups.

In general, this design splits the audio modules into layers of related operations. For example, basic generation modules are in the first layer, and audio effects appear in later layers. In particular, effects such as tremolo (section 4.13) and compression (section 4.14) are placed in the final layer, since they are usually applied at the end of a chain of effects.

Each point-to-point connection (such as the link from a crossbar to an audio module, Figure 4.3) is implemented with the AXI Stream protocol. This has a number of important benefits, which translate into an easily implemented back-pressure mechanism inside an HLS design. This backpressure mechanism would prove to be paramount to clocking audio samples out at the correct 48kHz rate.

4.5 Audio Module Standard Interface

A critical aspect of the design was a standardized audio module interface (see Appendix A). In order to successfully parallelize the development of separate modules, several early decisions were made and strictly followed until the end of the project; often times these decisions were made to simplify the use of HLS, rather than to increase performance of the final design.

We selected single-precision floating point as the numerical representation for the audio samples. This has the benefit of simplifying program logic, since it is usually not necessary to worry about overflow or underflow. Additionally, it is quite easy to write floating point code in HLS. However, as the specifications stated, the I2S core would need to output audio samples in a 24 bit signed integer format. Shown on the right-hand side of Figure 4.3, the stream of floating point numbers is converted to a stream of signed 24 bit integers via another HLS core, solving the incompatible numerical format issue.

Additionally, the final 48kHz synchronization mechanism for sample generation changed from the initial design. It was initially specified to be a blocking state within each HLS module, but it was *removed* from the modules to make writing audio modules easier as well as speeding up the HLS compiler. Instead, a ***`latch_wait`*** core is added in series with each audio module in the design (Figure 4.5.1). Its purpose is to apply back-pressure (through its AXI Stream connection to the preceding module) until it receives a pulse on its ***`latch`*** input (Figure 4.5.2). This latch signal was provided from the ATG module as discussed in section 4.3.

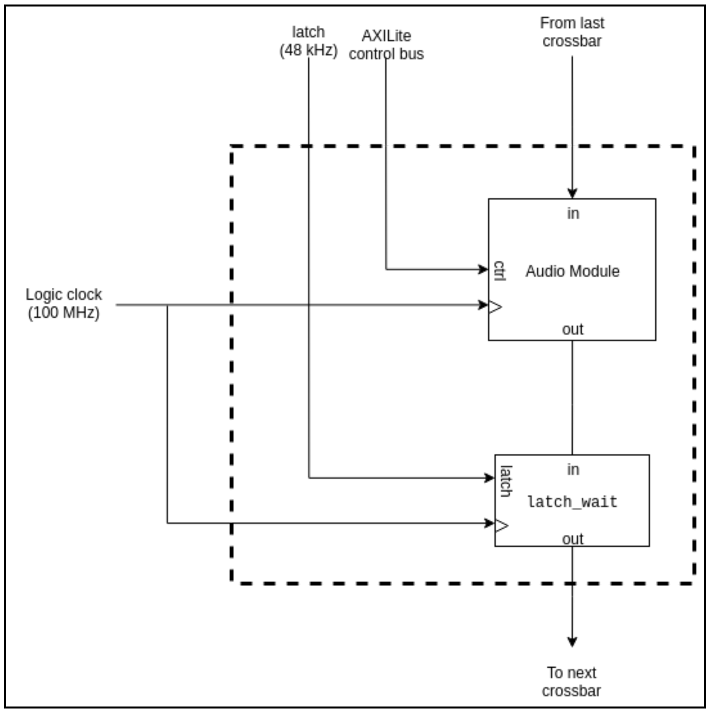


Figure 4.5.1: Splitting synchronization into separate IP core

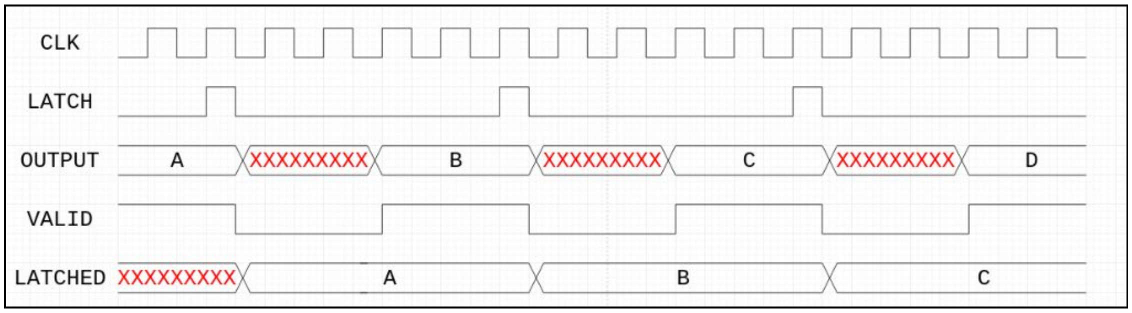


Figure 4.5.2. Waveform displaying how the latcher works

4.6 Saw Wave Generator

Figure 4.6.1 displays a block diagram of the saw wave generator. The centerpiece is a 24 bit counter, whose value is treated as a signed integer. Every sample, this counter is incremented by the correct value in order to generate a particular frequency. In this fashion, overflow of the 24 bit counter is handled in a very natural way; it is exactly the desired calculation! That is to say, no extra logic is needed to deal with overflow.

The frequency is selected as a floating point number and passed to the module via an AXI Lite interface. It must be converted to the correct increment for the counter, which is shown as a multiplication by K in the diagram. The calculation is provided below:

$$K = 2^{24}\frac{f}{f_s}$$

where f is the desired frequency in Hz, and $f_s = 48\ \mathrm{kHz}$ is the audio sampling frequency.

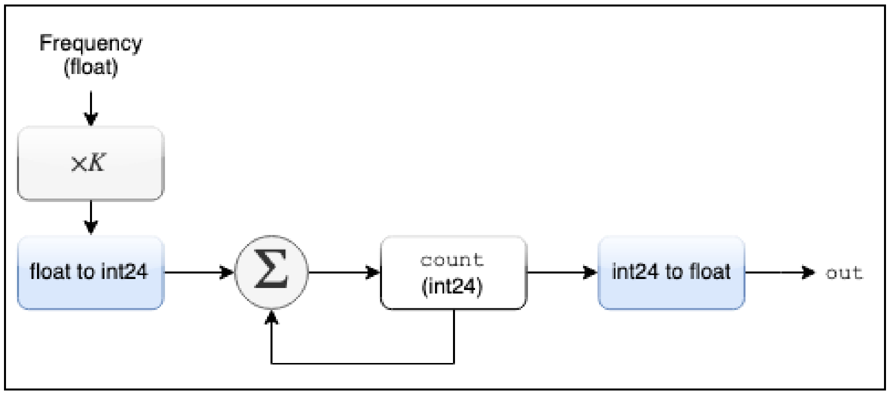


Figure 4.6.1: Block diagram of saw generator

4.7 FM Synthesizer

Figure 4.7.1 shows a block diagram of the FM Synthesizer, one of the audio source cores used in the project. The amplitude values of one period of the lowest sampled octave are stored in registers so no on-the-fly calculations of the sinewave are required. They are stored such that if they are repeatedly sampled they would produce a pure 48 kHz wave of that tone.

The flow of this module can be broken down into three stages:

- First the modulator wave is located and the instance is read with respect to time, octave and phase shift of the modulator
- The result is fed back to the carrier wave read pointer, where the phase shift is calculated as the sum of the modulator wave amplitude and carrier input phase shift
- The value is recorded and sent to the bus for later modules to place effects on it

Also found in the diagram is the wave calculation for FM Synthesizers with single modulator wave. Please refer to Appendix F for the background theory and operation of the FM Synth.

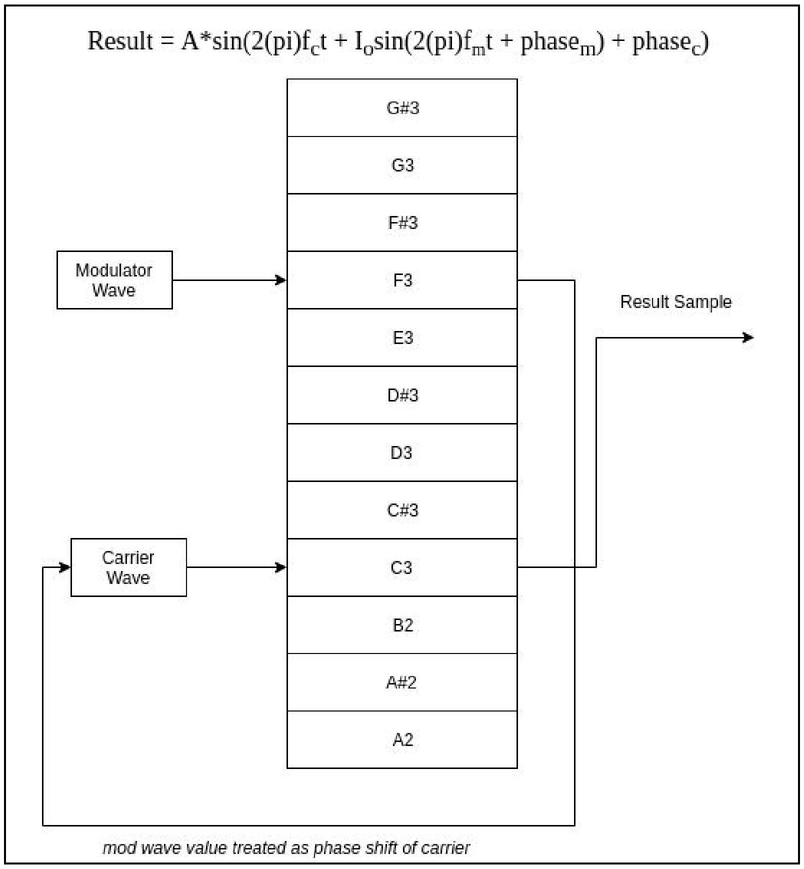

Result = A*sin(2(pi)f_c t + I_o sin(2(pi)f_m t + phase_m) + phase_c)
G#3
G3
F#3
Modulator Wave
F3
Result Sample
E3
D#3
D3
C#3
Carrier Wave
C3
B2
A#2
A2
mod wave value treated as phase shift of carrier

Figure 4.7.1: FM Synthesizer Block Diagram

4.8 White Noise Generator

The last audio source generator discussed is the white noise generator. It is a standard 16 bit linear feedback shift register (LFSR) with polynomial $x^{16} + x^{12} + x^{3} + x + 1$. At each sample, 16 bits are queried from the LFSR. This 16 bit value, treated as a signed integer, is converted to a float and normalized to lie between -1 and 1.

4.9 Digital Filter

Instead of implementing a multitude of different filters, it was decided to implement a generic yet configurable filter. One straightforward way to achieve this is with a standard biquad filter, depicted in Figure 4.9.1.

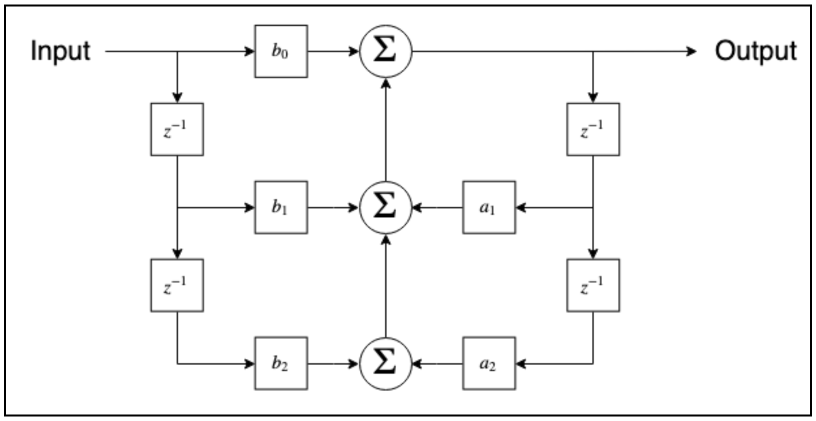


Figure 4.9.1: Diagram of biquad filter. z^{-1} represents the delay operator (i.e. a shift register)

By setting appropriate gains for b_0, b_1, b_2, a_1, a_2, this filter can become a low-pass, high-pass, or bandpass filter, among many others. The online BiQuadDesigner tool [6] was used to generate coefficients.

4.10 Echo/Reverb

The echo/reverb module is used to create repeated signal that decays in amplitude as time passes. This decrease is determined by the user in the scale factor variable. A value is read in from the former modules and summed with the current value stored at the read pointer multiplied by the scale factor. The result is stored at the write address which is *delay* addresses ahead of the read pointer. By multiplying the scale factor by each read, the scale reduction will compound with each read creating the hollow effect.

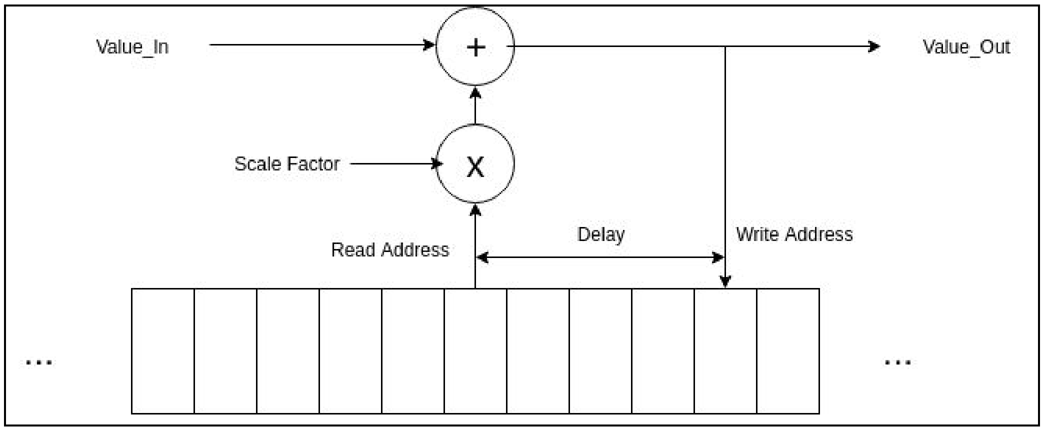


Figure 4.10.1: Echo/Reverb Block Diagram

4.11 Mixer

The chosen mixer implementation is a simple element-wise multiply with sum reduction operation (Figure 4.11.1). The inputs are gathered from the module's AXI Stream inputs, and the volume of each is specified by the user via the AXI Lite configuration bus. As an implementation detail, the module will not perform a (blocking) read on the corresponding input if the volume is below a particular threshold. This prevents the module from getting blocked if some of its inputs are unused. Without this, the datapath could easily lock up if not carefully configured.

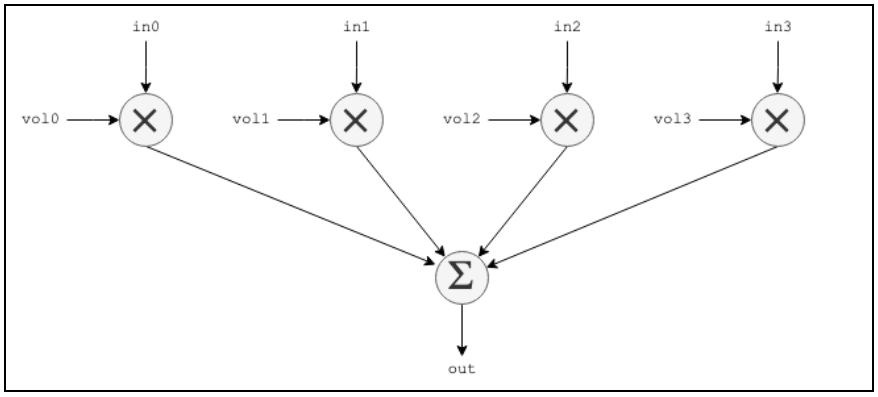


Figure 4.11.1: Block diagram of 4-input mixer

4.12 Envelope Generator

The envelope generator operates by simulating the pressing of a note by the input audio module. The constant wave data is passed to the envelope generator and the *press* signal determines the start of the function. The 4 regions of the generator are: *Attack, Decay, Sustain* and *Release*.

- The *attack* is the initial spike in amplitude that allows the instrument to have character upon the initial press. This allows a player to emphasize notes with harder and more impactful hits.
- The *decay* is the portion of time between the maximum attack amplitude and the amplitude the wave will output with as long as the key is being pressed. This is a negative amplitude slope usually as the attack is generally larger than the sustain amplitude.
- The *sustain* region is maintained as long as the key is being pressed and remains at a constant amplitude.
- When the key is released, the envelope enters the *release* region, where the amplitude steadily decreases for the amount of samples specified by the user.

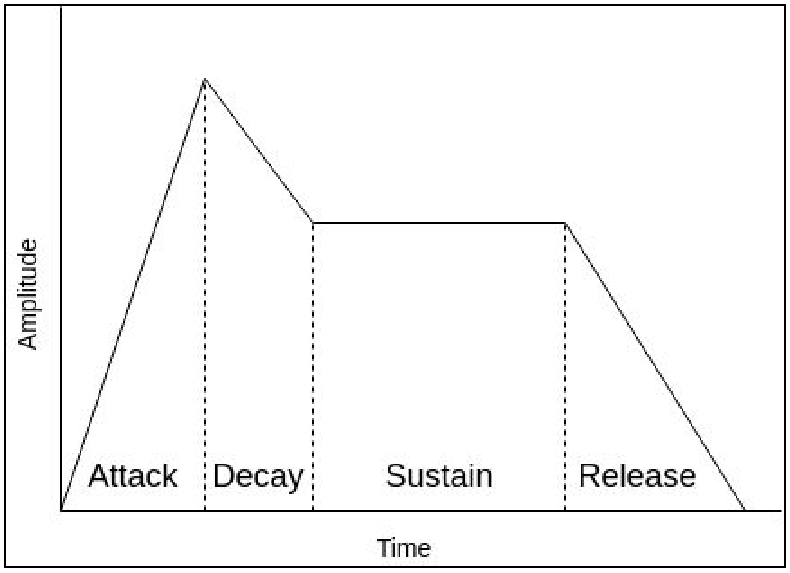


Figure 4.12.1: Envelope Amplitude Profile

4.13 Tremolo

This is one the volume altering effects achieved by performing amplitude modulation uniformly over the signal. The input audio signal is streamed in and multiplied element-wise with a low frequency triangle wave generated inside the module.

The internal settings (like depth and the LFO frequency) required for the tremolo are defined inside the module. These variables are initialized when the FPGA comes out of reset.

The real time performance of the system was satisfying and engaging. Figure 4.13.1 gives the block level view of Tremolo effect module.

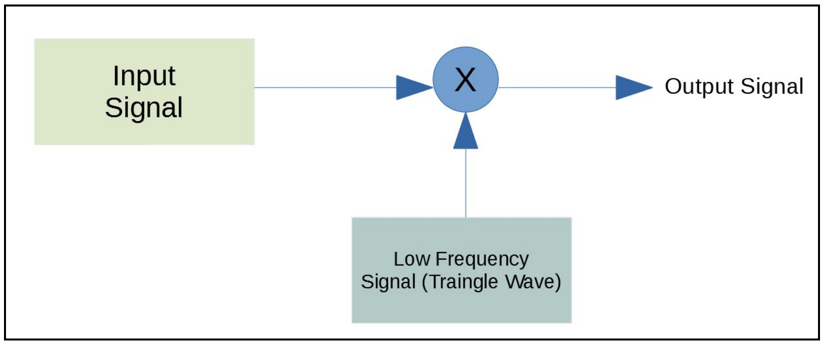


Figure 4.13.1: Block diagram of tremolo effect

4.14 Compressor

The purpose of an audio compressor is to reduce an signal's dynamic range with minimal distortion. For example, if you have a signal with many spikes (causing clipping), passing it through a compressor can reduce the amount of overall distortion.

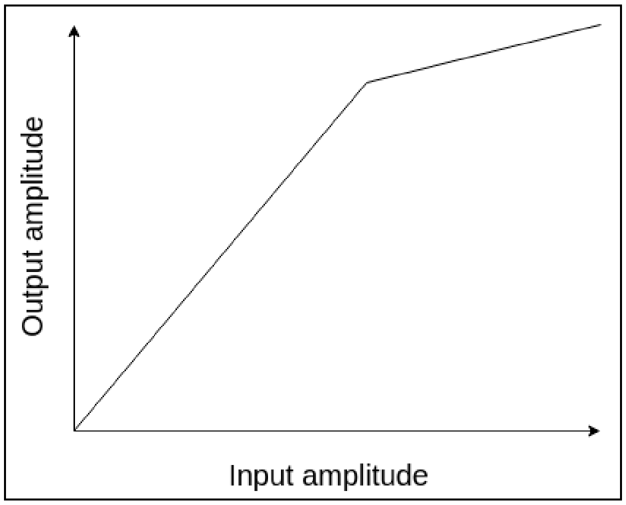


Figure 4.14.1: Output amplitude vs. input amplitude for simple audio compressor. The slope and location of the "knee" are configurable.

4.15 Vibrato

Vibrato modulates the frequency of the input signal with a low frequency source. This source (a sine wave of 6 to 10 Hz) is generated within the module. A circular buffer is created to hold a number of recent samples; the write pointer is used to add the input samples to the buffer, and the read pointer is placed as a function of the low frequency source. If the read pointer should land between two (discrete) samples, linear interpolation is used to generate the output signal. Figure 4.15.1 shows the different blocks that are used to achieve this functionality.

The effect is applied on a stream of input samples and the delay and width parameters are configured by the host application using AXI Lite Control bus. The default delay time is 0.4 ms and the LFO (Sine wave Generator) is configured by default to produce a signal of around 10 Hz.

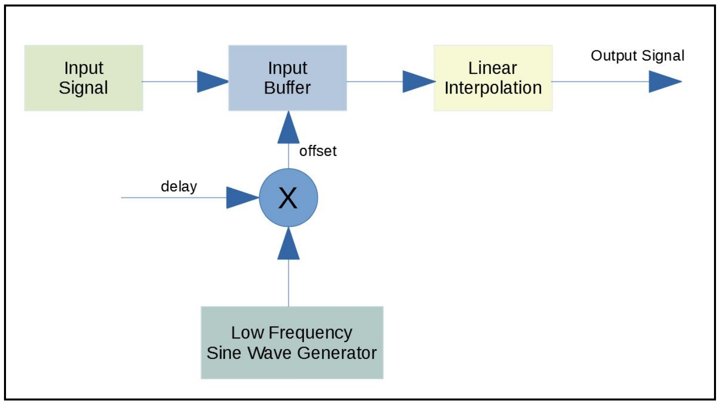


Figure 4.15.1: Vibrato block diagram

4.16 Demo Program

The demo program was written in C and cross-compiled using the Vivado SDK. It consists of a simple parser which extracts events from standard MIDI files. It uses the Linux real-time library to play events at the correct time.

There is no limit on the number of simultaneous voices that can be specified by a MIDI file. However, the demo program only makes use of the saw tree (Figure 4.1), which only supports 4 simultaneous voices. The program uses a simple least-recently-used replacement algorithm to select which voice to use for new notes. This means that certain MIDI files do not sound very good, but most come out as relatively faithful reproductions of the original song.

In addition to the MIDI file player, a number of shell scripts are provided, which simplify the task of playing the other sound generators and adding/removing effects.

4.17 Block Diagram

Included here are screenshots of the final design in the Vivado IP integrator. This design was created hierarchically, and is presented here in a "top-down" fashion.

Figure 4.17.1 shows the top-level block diagram. Seen here is the FPGA shell, including the IPs for configuring the PLL and Zynq processor, as well as the I2S generator block. Additionally, the converter block near the middle is used to convert floating point numbers to the 24 signed integers required by the CODEC.

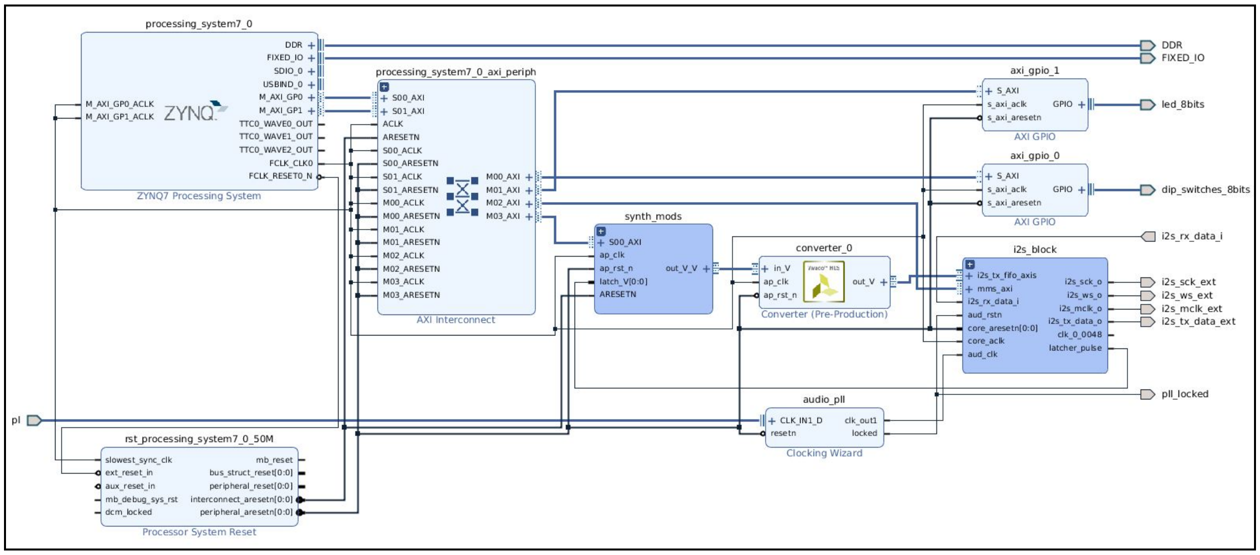


Figure 4.17.1: Top-level block diagram of final design

Figure 4.17.2 shows the contents of the I2S block in Figure 4.17.1 above. It contains three submodules which are discussed in Section 4.3. The entire core is driven by aud_clk (12.288MHz) and core_aclk (100MHz) and supplies five external signals that externally connect to the audio CODEC. As can be seen from the figure, there are no HLS cores in this submodule, as Verilog RTL currently appears to be the only (or only easy) way to achieve cycle-accurate RTL cores.

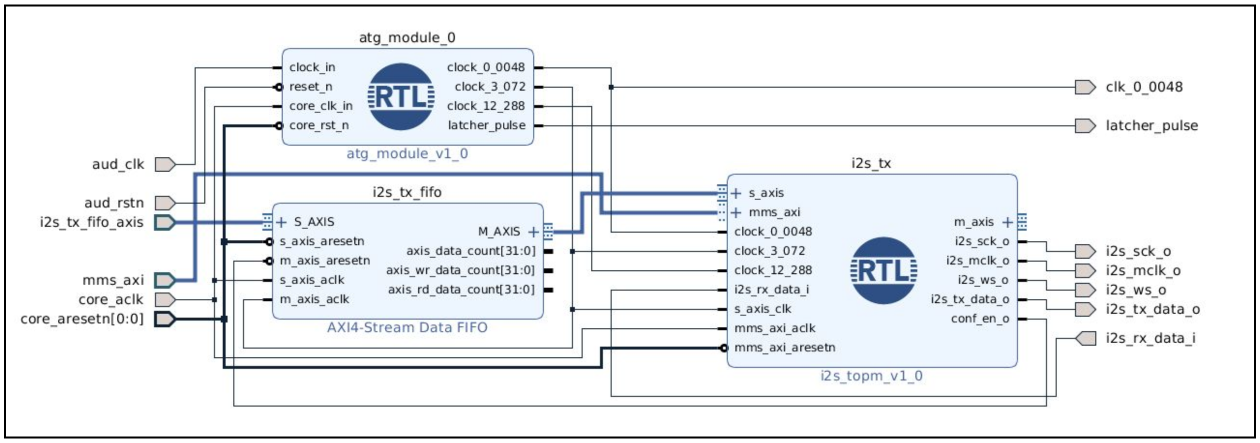


Figure 4.17.2: I2S and timing generation

Figure 4.17.3 represents the contents of the ***synth_mods*** block of the top level block diagram (Figure 4.17.1). The diagram has been cut in half to fit better on the page; note that the ***fx2_xbar*** module at the top right is also shown on the bottom left. The top half shows the generators and effect layers, which are expanded out in Figures 4.17.4, 4.17.5, and 4.17.6. The bottom half shows the output stages where voices are combined into a single output. Note the use of a compressor and a filter just before the main output, allowing the user to easily tailor the synth to a new set of speakers (which may have a different frequency response).

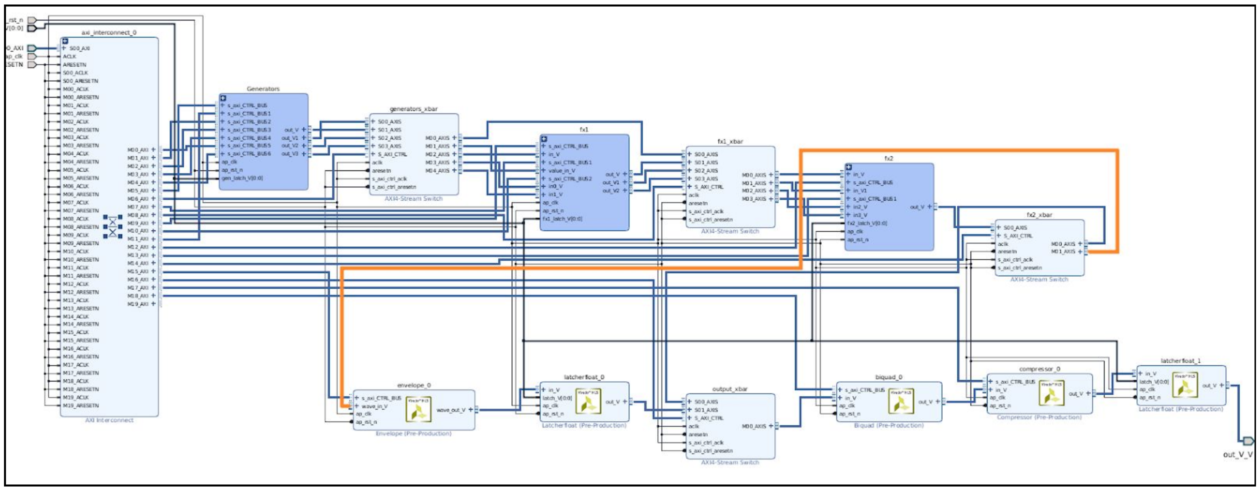


Figure 4.17.3: Audio modules (note layers and crossbars). To better fit on the page, the right half of the "chain" was moved underneath the left half. The highlighted wire shows the audio stream.

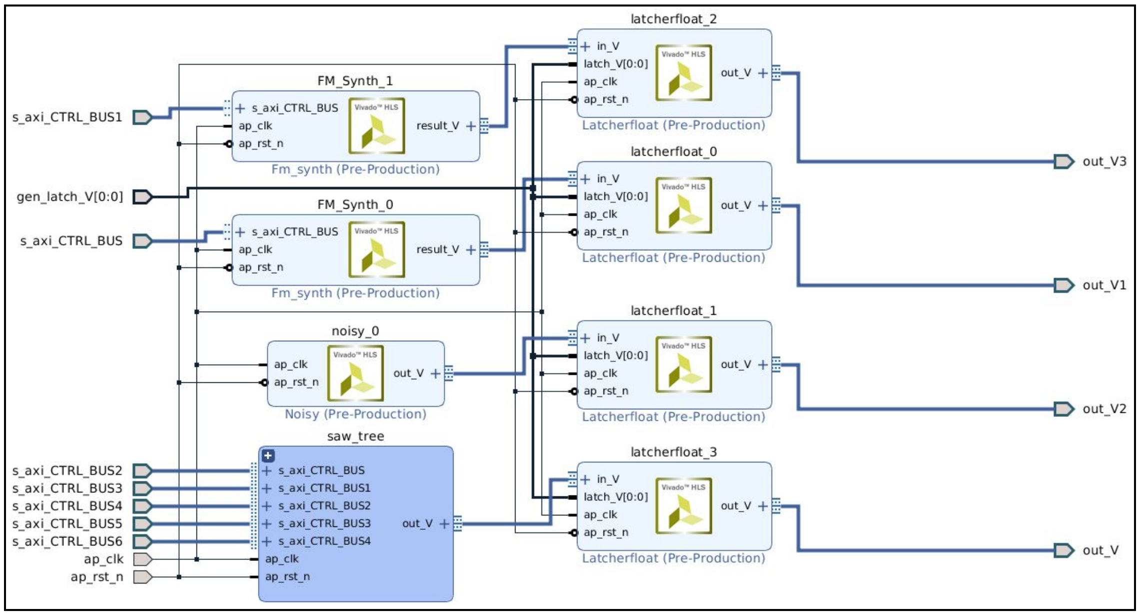


Figure 4.17.4: Generators block of 4.17.3. Note the use of data latchers

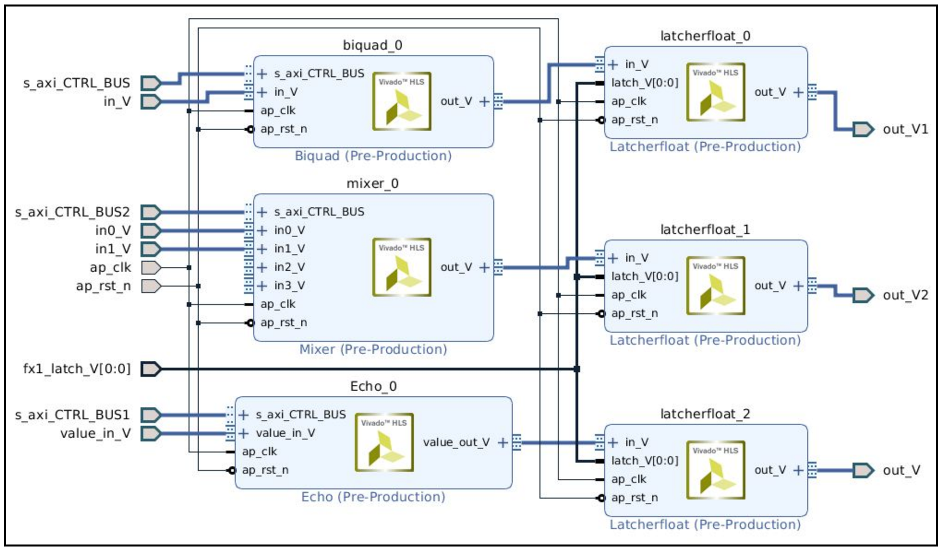


Figure 4.17.5: FX1 block of 4.17.3

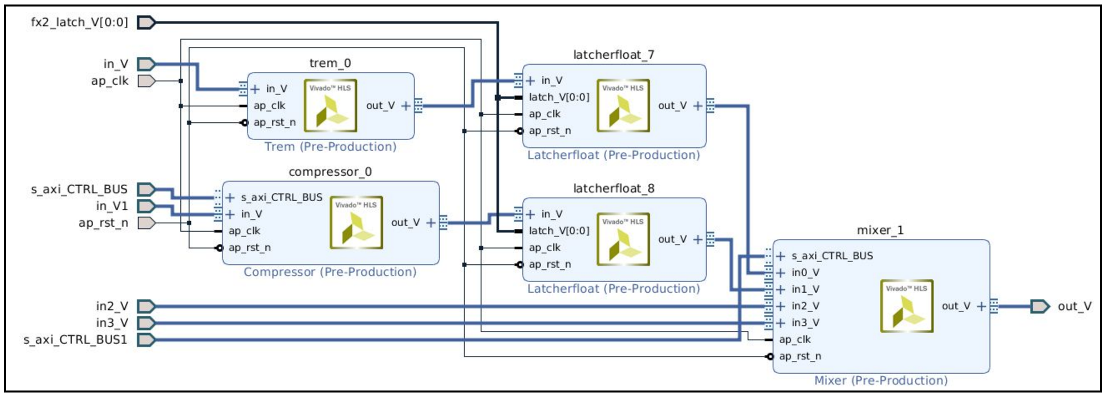


Figure 4.17.6: FX2 block of 4.17.3.

5. Methodology

The project methodology followed a milestone based flow. In broad terms, there were four stages of the design, each with their own requirements that needed fulfilling before the next milestone could begin. Some parallelization occurred during the development, but no final tests of a milestone could be completed before the prior stage testing was considered complete.

5.1 Design Environment

As part of the first milestone, or even as a precursor to it, the hardware platform needed to be chosen. The AVNET Zynq Mini ITX development platform, containing a Xilinx Kintex-7 SoC [Fig 5.1] was selected for the basis to our audio synthesizer project [2]. It was selected due to the board having a hardened ARM processor in it, as well as the full featured Texas Instruments ADAU1761 codec [1].

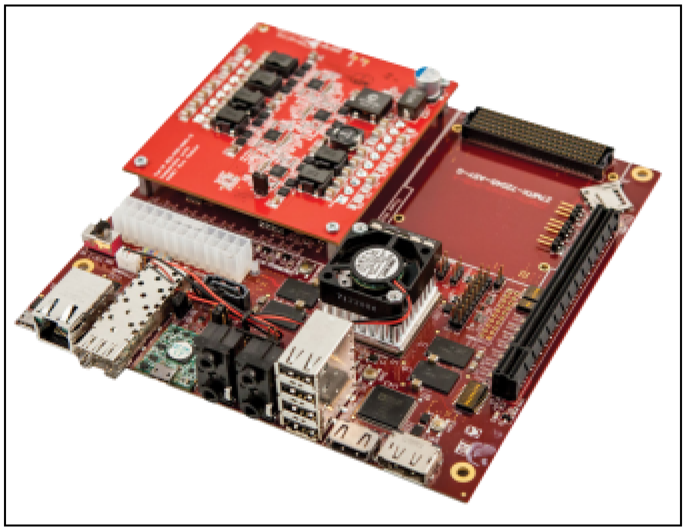


Figure 5.1: AVNET Zynq Mini ITX development platform [2]

The GHOST Synth project was built using a design flow that included Xilinx Vivado 2017.2 for the FPGA shell, Xilinx Vivado HLS 2017.2 for the synthesizer modules, Vivado SDK 2017.2 for the runtime demo scripts and Petalinux toolchain 2016.2 for the on-chip linux environment. Petalinux was chosen as AVNET provided a complete petalinux board support package (BSP) and a golden hardware reference design (GHRD) to bootstrap the development.

From Figure 5.2 it can be seen that the basic golden reference design contains a Zynq Processor System, some external GPIO interfaces and an AXI memory mapped interconnect module. Starting a project from a golden reference design is the preferred method when considering risk mitigation on board bringup and platform design.

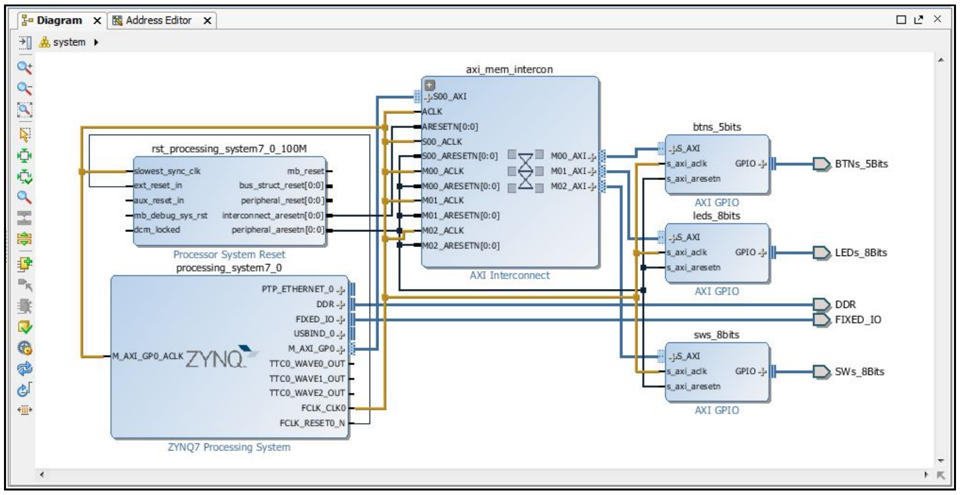


Figure 5.2: Golden Hardware Reference Design (GHRD)

Lastly, Git was chosen to be the revision control software used for the development flow of the entire project. All of the HLS cores were treated as submodules with their own git repositories to create a modular development flow that allows for truly parallel development between teammates. Please see Appendix B for the GitHub repository link.

5.2 Design Partitioning & Verification/Testing

As mentioned at the start of this section, the design was broken down into multiple milestones. Each one represents a complete hardware verification and git commit before continuing with the next milestone. In a sense this also partitions the design at a functional level, as each milestone focuses on a different independent element of the design.

The following sections explain what their respective milestone achieved, how it was completed and what tests were performed to ensure that the project was working in the scope of each milestone.

5.2.1 Milestone 1: Board Bringup and Tool Flow

The board bringup and toolflow, while not an actual design component is an essential milestone to the project development. These processes set up the rest of the project for either a smooth development cycle or a difficult and buggy experience.

For a baseline image, a Petalinux BSP and GHRD provided by AVNET was used as a starting point. The first goal was to run through a fresh build of the FPGA bitstream as well as the linux image without changing any features. That image was then used to verify that the existing GPIO memory mapped interfaces were still accessible. This in itself is a long and complicated process when doing full SoC builds from the ground-up and many issues were found along the way (Section 8).

However, once the image was being properly generated, the onboard LED's connected via memory mapped GPIO pins were used to verify that the image was indeed working. Another test performed was probing the I2C bus to verify that devices (such as the audio CODEC) are detected from the PS. There are many minute details and subtleties that make board bringup a larger task than expected in most projects. A full progress log can be found in Appendix D.

5.2.2 Milestone 2: FPGA Shell, CODEC configuration, and Sawtooth Generator

Once the tool-flow milestone was completed, modifications to the FPGA image itself, as well as the linux image were required to verify that sound could be produced by the FPGA with our architecture. This began with developing the I2S core, PLL and ATG modules. Once these were developed, debugged and connected in Vivado IP Integrator, a small test bench was required to verify clocking and resets to the shell modules, as well as the output clocks provided by the FPGA to the audio CODEC.

To create a simple test bench, access to the Xilinx Virtual IP simulation core was required. This core facilitates writes and reads to modules connected to the AXI interconnect. The Vivado version initially intended for the project (2016.2) had to be updated to a more modern version (2017.2) in order to gain access to the Xilinx Virtual IP simulation core, rather than the old unlicensed version (Appendix D).

Once the clocking was verified, a simple HLS sawtooth generator was added to the block diagram in Vivado, along with an accompanying latcher module for backpressure and synchronization to the 48 kHz sampling rate. Again, the simulation needed to be updated to control the new blocks, but once debugged a full data flow from an HLS core to the external I2S pins could be seen (Figure 5.2.1).

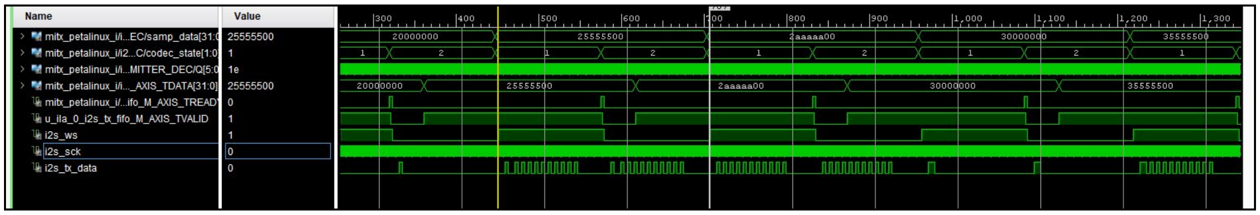

Name
Value
mitx_petalinux_i/i...EC/samp_data[31:0] 25555500
mitx_petalinux_i/i2...C/codec_state[1:0] 1
mitx_petalinux_i/i...MITTER_DEC/Q[5:0] 1e
mitx_petalinux_i/i..._AXIS_TDATA[31:0] 25555500
mitx_petalinux_i/...ifo_M_AXIS_TREADY 0
u_ila_0_i2s_tx_fifo_M_AXIS_TVALID 1
i2s_ws 1
i2s_sck 0
i2s_tx_data 0
300
400
500
600
700
800
900
1,000
1,100
1,200
1,300
20000000
25555500
2aaaaa00
30000000
35555500

Figure 5.2.1: External facing I2S waveform from the SAW wave generator. This is what the audio CODEC would see

Next, hardware verification could begin, along with determining exactly which combination of I2C writes were required to access and program the ADAU1761 audio CODEC. There were a few hurdles to overcome with this, namely the 16 bit data addressing that the audio CODEC runs on and a complicated register configuration for the audio CODEC. However, using the SigmaStudio software provided by Analog devices, the correct set of I2C writes to the audio CODEC were determined (Appendix C).

The final test for this milestone was to verify that an oscilloscope could detect a 1 kHz sawtooth wave coming out of the DAC [Fig 5.2.2]. Once this was verified the shell could be considered complete.

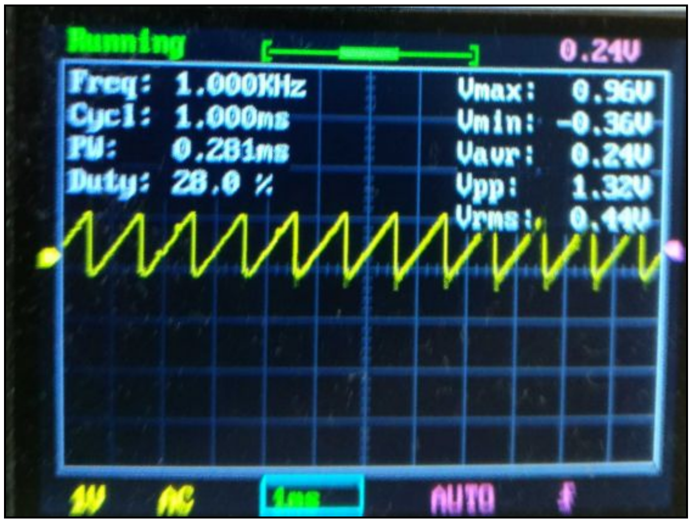


Fig 5.2.2: 1kHz sawtooth wave.

5.2.3 Milestone 3: Verify Inter-module Dataflow and MIDI Demo

With the FPGA shell in place (and with confirmation that the latching approach was working correctly), it became possible to verify the audio module interconnect. First, a subset of all the audio modules (consisting of a saw generator, noise generator, and mixer) were verified in hardware. For each of these modules, a Vivado simulation was performed on just that module inside the shell, and if this simulation showed good results, a bitstream was tested in hardware. This allowed us to ensure that each module was producing the correct type of sound or effect.

Configuring modules and crossbars from within the Linux OS is done via memory-mapped registers. A crude way to set the desired values is the use the ***`devmem`*** program provided in busybox. This quickly becomes tiresome, as it requires the user to manually type out the hex constants for each individual register. Instead, a number of shell scripts were created to facilitate the testing of modules.

Once this minimal subset of modules was verified, they were then used to test the crossbar approach shown in Figure 4.3. Originally, a crossbar IP was hand-written in HLS. It was discovered that there was a large number of boundary cases not correctly handled by the original crossbar. For example, sometimes a crossbar would be in an intermediate state (between two valid states) causing some modules to get stuck on a (blocking) AXI stream read/write. To remedy this issue, we discovered the AXI Stream Switch IP produced by Xilinx which appropriately handles all these difficult edge cases.

Once the basic functionality of modules and crossbars was in place, it became possible to begin writing a demo program while development on the remaining modules was completed. We chose to start writing this program early, since it could be done in parallel with remaining work, and would also be useful for testing new bitstreams.

5.2.4 Milestone 4: Final Implementation

The requirement of the final implementation was that a demo program, a shell script that parses through testing all implemented modules, is run. Luckily, the incremental approach allowed for rapid debug of the minor hardware issues not found in simulation, as one module could be quickly isolated as the issue when the others have already been verified. However, there was still a major issue in generating the appropriate scripts to test and present various portions of the project.

The scripts have to handle setting the proper crossbar addresses for each module to be activated as well as writing the appropriate values to each offset to configure each module for operation. Various demo programs are developed to show the use of the midi program, a series of midi files read and played by the saw wave generators. A script is also used to turn on the FM Synth and change the modulator and carrier waves.

Using these audio modules, the program will step through turning on the echo module, tremolo module and using both effects together. This displays that by layering the effects between crossbars, we are able to bypass the effect or wire through their outputs. The biquad filter is used in the *bandpass* mode on the saw waves. Using the FM Synth, we are able to verify the correctness of the envelope generator, as notes are played and their amplitudes are given character based on a key-press demo script. The noise generator is used to show the other properties of the biquad filter. The biquad is turned from both highpass to lowpass filters and the output is verified.

6. Contributions

Austin Liolli

Austin was responsible for developing audio and effects modules used throughout the final design as well as assisting in the debug of the final system. Austin worked closely with Marco to generate the modules such that they streamed ideally throughout the flow of the project and the generated samples that would be handled correctly by other modules. Austin was also responsible for generating the final presentation for the group. The audio and effects modules contributed are:

- FM Synthesizer (section 4.7 and Appendix F)
- Echo/Reverb (section 4.10)
- Envelope Generator (section 4.12)

Marco Merlini

Marco supervised the portion of the project dealing with audio modules. Over discussions with other team members, and with significant HLS advice from Mr. Juan-Camilo Vega, he developed a standard audio module interface (see Appendix A). He was also responsible for implementing the following audio modules:

- Latcher (synchronization primitive, section 4.5)
- Saw generator (section 4.6)
- White noise generator (section 4.8)
- Digital filter (section 4.9)
- Mixer (section 4.11)
- Compressor (section 4.14)

In addition to supervising the design of the module interface, Marco (in tandem with Mat and Austin) contributed to the final integration effort, including any and all debugging.

Lastly, Marco was responsible for the creation of the demo program (section 4.16).

Mat Hildebrand

Mat Hildebrand was responsible for the overall system architecture and tool-flow for the project. Most of the work he completed was within the first two milestones (5.2.1, 5.2.2). His responsibilities for the project can be broken down as follows:

- Clocking architecture (4.1)
- Processing System configuration and linux BSP development (4.2)

- FPGA Shell development (4.3)
- I2S core development (4.3)
- Design Environment and Tool Flow (5.1)
- Initial board bringup (5.2.1)
- Audio CODEC configuration (5.2.2)
- Ongoing System Debug

SRIHARSHA H S

Sriharsha was responsible for generating some of the audio effects on FPGA using Vivado HLS. Also, he assisted with the ATG (Audio Timing Generator) module development. The modules he contributed are:

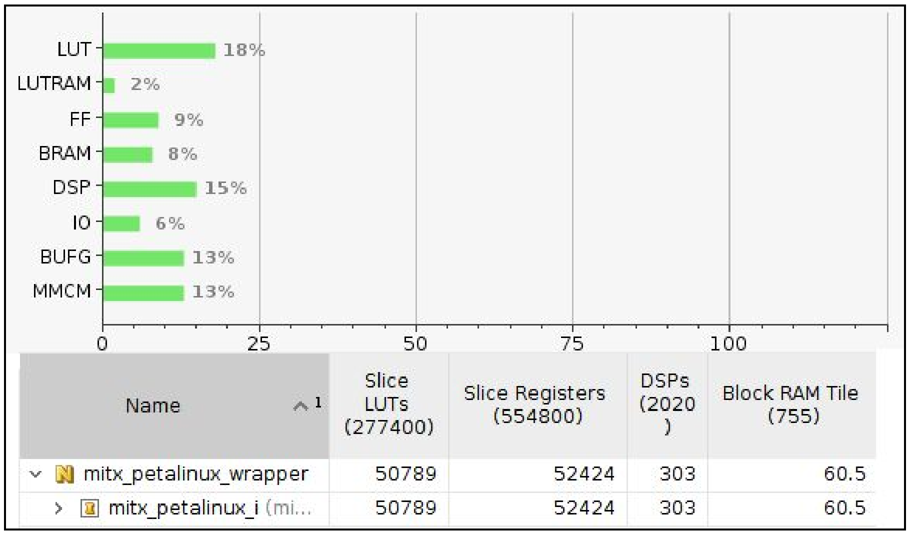
- Vibrato (Section 4.15)
- Tremolo (Section 4.13)

7. Design Characteristics

This section details the utilization reports of the final design (as shown in section 4.17). Additionally, we discuss the utilization of time by the development team.

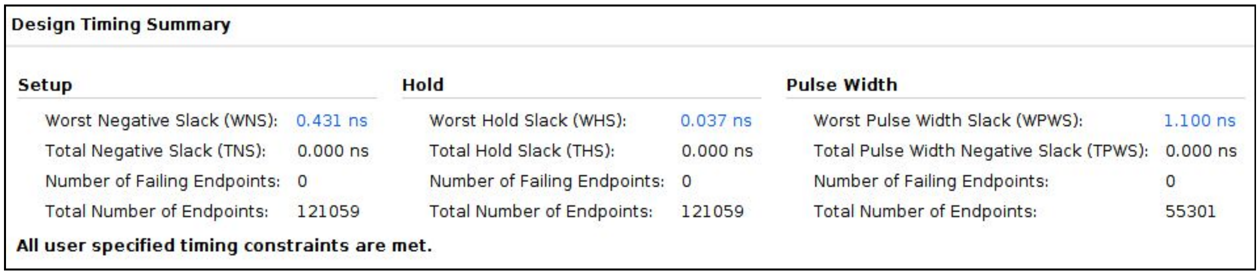
7.1 Resource Utilization

Due to the audio synthesizer not being a demanding application for a large FPGA, no issues with fitting or intra-clock timing arose. The final utilization numbers can be visualized below in Figure 7.1.1, with the clean timing results seen in Figure 7.1.2.



Name	Slice LUTs (277400)	Slice Registers (554800)	DSPs (2020)	Block RAM Tile (755)
mitx_petalinux_wrapper	50789	52424	303	60.5
mitx_petalinux_i (mi...	50789	52424	303	60.5

Figure 7.1.1: a) top, Visualization of Utilization b) bottom, Overall Utilization Numbers



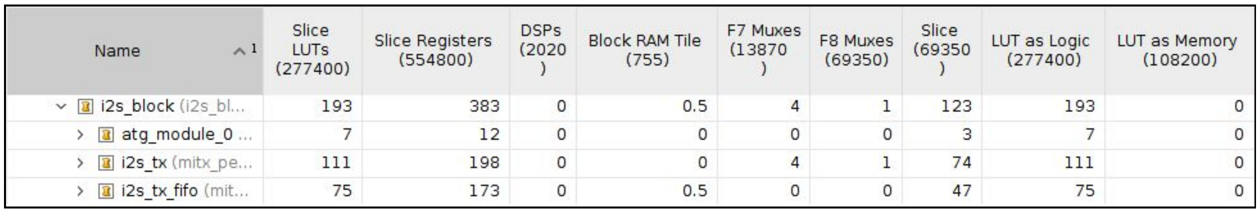
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.431 ns	Worst Hold Slack (WHS):	0.037 ns	Worst Pulse Width Slack (WPWS):	1.100 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	121059	Total Number of Endpoints:	121059	Total Number of Endpoints:	55301

All user specified timing constraints are met.

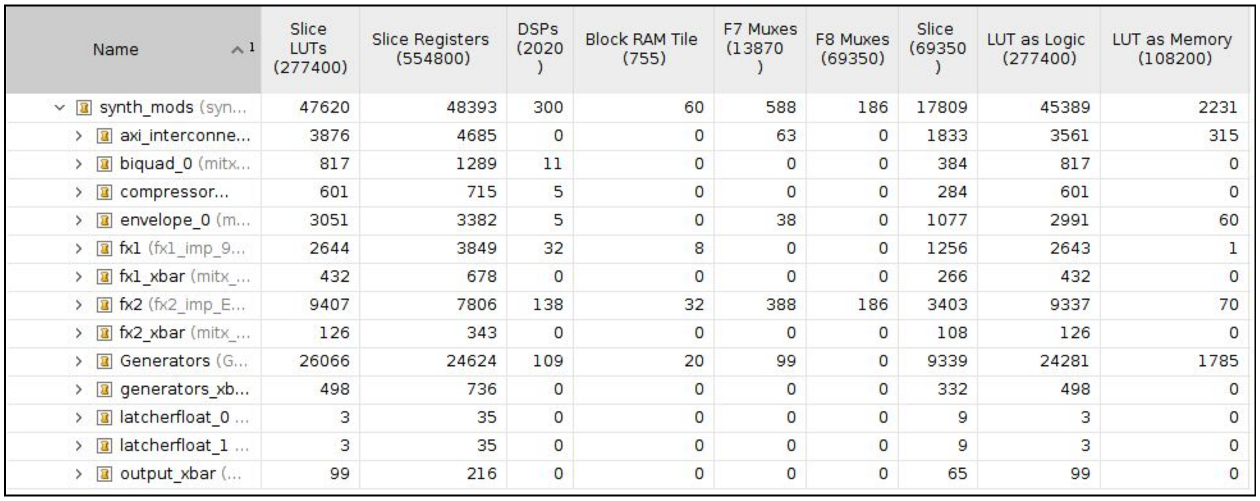
Figure 7.1.2 Timing results from final design

Figure 7.1.3 shows the utilization breakdown per module of the I2S block, while Figure 7.1.4 shows the utilization breakdown per module of the audio synthesizers core. These numbers are meant to show a general trend of where the various FPGA resources were allocated. In this case, IO, BUFG and MMCM resources aren't allocated as they are more or less fixed with the design architecture (e.g. one MMCM due to the audio PLL).



Name	Slice LUTs (277400)	Slice Registers (554800)	DSPs (2020)	Block RAM Tile (755)	F7 Muxes (13870)	F8 Muxes (69350)	Slice (69350)	LUT as Logic (277400)	LUT as Memory (108200)
i2s_block (i2s_bl...	193	383	0	0.5	4	1	123	193	0
atg_module_0 ...	7	12	0	0	0	0	3	7	0
i2s_tx (mitx_pe...	111	198	0	0	4	1	74	111	0
i2s_tx_fifo (mit...	75	173	0	0.5	0	0	47	75	0

Figure 7.1.3 Utilization breakdown for I2S Block



Name	Slice LUTs (277400)	Slice Registers (554800)	DSPs (2020)	Block RAM Tile (755)	F7 Muxes (13870)	F8 Muxes (69350)	Slice (69350)	LUT as Logic (277400)	LUT as Memory (108200)
synth_mods (syn...	47620	48393	300	60	588	186	17809	45389	2231
axi_interconne...	3876	4685	0	0	63	0	1833	3561	315
biquad_0 (mitx...	817	1289	11	0	0	0	384	817	0
compressor...	601	715	5	0	0	0	284	601	0
envelope_0 (m...	3051	3382	5	0	38	0	1077	2991	60
fx1 (fx1_imp_9...	2644	3849	32	8	0	0	1256	2643	1
fx1_xbar (mitx_...	432	678	0	0	0	0	266	432	0
fx2 (fx2_imp_E...	9407	7806	138	32	388	186	3403	9337	70
fx2_xbar (mitx_...	126	343	0	0	0	0	108	126	0
Generators (G...	26066	24624	109	20	99	0	9339	24281	1785
generators_xb...	498	736	0	0	0	0	332	498	0
latcherfloat_0 ...	3	35	0	0	0	0	9	3	0
latcherfloat_1 ...	3	35	0	0	0	0	9	3	0
output_xbar (...	99	216	0	0	0	0	65	99	0

Figure 7.1.4 Utilization breakdown for synthesizer modules

7.2 Time Utilization

As a group, we are of the opinion that time was utilized well. Parallelization of tasks was used when possible and collaboration otherwise. We worked together to develop a platform with clear goals and set reasonable milestone deadlines that could be met with earnest effort. Some things to improve upon would include:

- Choosing a better flow for the SDK and linux image, or choosing a more modern board with premade BSP's that can be used in a plug-and-play method.
- Communication on how to debug each others work as well as what to look for. As an example the FM synthesizer was being debugged for a significant portion of time, but it was actually not the root cause of the issue. With more knowledge sharing this could be avoided.
- Better environment setup so all team members can develop on the platform without needing a large docker image.

All goals were met in a timely fashion and no major project hiccups were encountered that lasted more than three days.

8. Problems

The problems faced while developing this project can be divided into three categories: issues due to the tools/environment, issues with the hardware platform and issues with the application development itself.

8.1 Issues Relating to Tools/Environment

For the entire duration of a project, the tool flow/development environment are on the critical path. For this reason, any issue here was treated with a high priority status and dealt with immediately. One exception to this was the Xilinx license server provided by the ECE department. Occasionally this would go down and the downtime would be out of the hands of the group members. However, it would always come back up within a day or so, limiting its negative impact on the project.

The most critical development environment issue faced was that of version compatibility between the provided BSP by AVNET (version 2016.2) and the required version of Vivado for simulation of the design(version 2017.2). Generally speaking, it is good to have matching versions of the software tool chain (for generating a linux image) and the hardware, but in the case of petalinux and Vivado, it is required [7]. The BSP that was provided was generated for Vivado version 2016.2 and would require significant effort to port to 2017.2, as many changes occurred between the two versions. On the other hand, any Vivado version prior to 2017.x used an old method/module of simulating the Zynq processor system or any AXI bus. These old simulation models were no longer licensed or supported and thus simulation could not occur.

The time saving solution was to perform all changes to the linux image, including processing system configuration in Vivado 2016.2. On the other hand, the actual FPGA development was completed in Vivado 2017.2. This could be done as the bitstream would have to be the same regardless of version, as the bitfile interpreter does not change.

Finally, we faced an issue related to Vivado HLS. When using the dataflow directive, HLS creates a Verilog module called ***`Block_proc_U0`*** inside the final IP. If multiple dataflow-based HLS IPs are used in the same IP integrator diagram, this results in a name collision. One team member spent about 5 hours tracking down this issue, since Vivado's error messages were not very helpful. Once the problem was discovered, the solution was to add a ***`config_rtl`*** option in the solution's compilation settings (in Vivado HLS) in order to add a prefix to all generated RTL code.

8.2 Hardware Platform Problems

As in any project, the issues will not necessarily be with the application or the development environment. Due to the complexity of hardware platforms, digesting and understanding every caveat can be time consuming. This is exacerbated by the fact that no two applications are the same and thus no user guide can cover every situation.

In our case, we had two misunderstandings of the hardware that required combing through the schematics. First, an I2C jumper on the board was in the wrong location for our project. It was (by default) set to have the FPGA I2C bus pins driving the entire I2C bus instead of the hardened processor system ones. This became an issue when trying to use the onboard i2c bus tools to talk to the audio CODEC and the jumper had to be found and moved.

The second issue revolved around a complex register configuration required by the audio CODEC (discussed in section 4.2). However, on top of this and on a more hardware related note, the ADAU1761 uses a 16-bit data-addressing mode and is hidden behind an unnecessary I2C mux between the FPGA and the audio CODEC. The solution was tricky to find but it involved bit-banging i2c writes together from the linux shell [8].

Finally, two actual hardware problems were found. The first being the SD card location; it was placed in a very awkward location and was difficult to access. Secondly, the probe points for the audio CODEC (in order to verify all of the clocks going to it) were on the bottom of the board, thus the entire case needed to be disassembled to perform this step.

8.3 Application Bugs

For a period of about three days, two team members were working together exclusively to fix a bug where the FM synthesizer was not producing sound. Originally, it was believed that since the saw generators were producing sound, the problem had to lie within the FM synthesizer. However, eventually it was discovered that the other modules had not yet been modified to produce/accept samples normalized between -1 and +1. That is, the saw generators were producing large numbers, and the float-to-int24 converter was converting the samples directly. The FM synthesizer, however, was (correctly) producing values between -1 and +1, which were present but completely inaudible. This situation is depicted in Figure 8.3.1. Ultimately. the solution was to normalize the saw generators, and include a large gain in the converter.

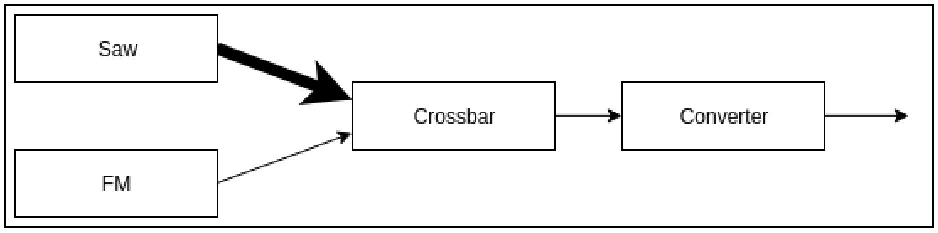


Figure 8.3.1: Simplified diagram of normalization bug. The saw generator was producing disproportionately large values, and as a consequence, the converter was not applying a large gain. However, the small values from the FM synth were not loud enough to be audible.

9. Retrospective, Conclusions, etc.

9.1 Experience with HLS

It's important to understand the strengths and weaknesses of HLS. It is *particularly* weak at converting program logic and calculations written in C into efficient hardware[1]. However, it is *exceptionally* strong at allowing the designer to easily manage interface protocols on blocks and ports. To this end, the most salient design feature of our audio modules is their interface as described by HLS (see Appendix A).

The very first thing to mention is the ***`ap_ctrl_none`*** interface for blocks. This causes HLS to run your function from its opening curly brace to its closing curly brace in perpetuity. Aside from a clock and reset input, there is no other way to directly control whether or not the module is running. This interface was used in all HLS modules created by the team.

More importantly however, is the ***`axis`*** interface for ports. Behind the scenes, using the somewhat mystical ***`hls::stream`*** class template, HLS will not only generate and manage the appropriate extra wires to implement an AXI stream interface, but it also allows you to use blocking reads and writes. That is, even if you have fancy pipelining or dataflow optimizations enabled, the HLS compiler will automatically enable your module to be "paused" in a natural way when performing I/O on any port. In our design, we specifically use this functionality to allow our modules to compute at full speed, yet easily be synchronized to the audio sampling rate of the CODEC.

This project did not have stringent performance requirements. Thus, it is likely that the design would have taken about twice as much effort to complete using an HDL. However, if performance *had* been a concern, the amount of extra time that would have been needed *in HLS* would have likely caused the two approaches to require roughly equal amounts of effort. Perhaps there is an "inline Verilog" option in HLS which could expedite the process in these cases.

The authors would like to thank Mr. Juan-Camilo Vega for his invaluable advice regarding HLS; without it, it is likely we would have wasted a lot of time trying to make things work with a suboptimal approach.

[1] We got around this problem in our project because the audio sampling rate (48 kHz) is much slower than the 100 MHz clock for the audio modules. This meant each module had a budget of about 2000 clock cycles per sample, which was more than enough. The longest initial interval in the project was less than 50.

9.2 Other

The overall flow of the project was facilitated by the individual group members which each contributed in their area of expertise. The group organized meetings where objectives were laid out in completable tasks with a specified completion date.

The groups decision to debug at each step of module insertion allowed for a low risk design flow and new components to be added with a high level of confidence in the rest of the system. By working out what was needed at each stage of the design the group was able to attack each stage effectively. We are all quite pleased with the result of the project as its functionality displays all features originally discussed in the presentation and conceptual design.

To enhance the design of the synthesizer, a user interface could be generated either through push buttons or GUI. This would allow the general user to more freely access all modules of the design while the background runs the appropriate scripts to set the crossbars, writes to memory addresses etc. Another optimization would be to use a real MIDI keyboard as the input to the audio modules, where the user would be able to select the notes interactively. Depending on the audio module the amount of voices are limited. In the case of FM Synth, where there are carrier and modulator waves both based on keyboard notes, an appropriate selection of the modulator wave would have to be determined. For filtering options as well as modules with scaleable parameters, a GUI with simulated knobs would give the user an accurate feel for using each modules features.

The course being divided into the labs and project felt disconnected from the class, where topics covered in the lecture related to some timing principles and the labs were more geared towards HLS specific optimizations. The TA's were very helpful with HLS tips and tricks through both the assignments and project, but each of the members would have benefitted from a tutorial of navigating Vivado, general HLS practices and a realistic use case of each pragma.

References

[1] Analog Devices, "SigmaDSP Stereo, Low Power, 96 kHz, 24-Bit Audio Codec with Integrated PLL," ADAU1761 Datasheet, Jan. 2009 [Revised Oct. 2018]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/ADAU1761.pdf

[2] Avnet Engineering Services, "Mini ITX Board," Product Brief. Available: http://zedboard.org/product/mini-itx-board

[3] Lattice Semiconductor, "I2S Controller with WISHBONE Interface," Product Brief, March 2014. Available: https://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/ReferenceDesigns/ReferenceDesigns02/I2SControllerwithWISHBONEInterface

[4] J. Tatsukawa, "MMCM and PLL Dynamic Reconfiguration," Xilinx Application Note 888, Jan. 2012 [Revised Apr. 2017], Available: https://www.xilinx.com/support/documentation/application_notes/xapp888_7Series_DynamicRecon.pdf

[5] airhdl, "airhdl User Guide," 2019. Available: https://airhdl.com/help/UserGuide.html

[6] P. Lutus. "BiQuadDesigner," 2017. Available: https://arachnoid.com/BiQuadDesigner/

[7] Xilinx Product Forums, "petalinux & vivado version compatibility," Forum Discussion, Jul. 2017. Available: https://forums.xilinx.com/t5/Embedded-Linux/petalinux-amp-vivado-version-compatibility/td-p/782552

[8] Raspberry Pi Forums, "I2C 16-bit Memory Addressing?" Forum Discussion, Oct. 2012. Available: https://www.raspberrypi.org/forums/viewtopic.php?t=17590#p190833

Appendix A - Info For Writing Audio Modules

Here's a quick example of a module which multiplies its input by a configurable scalar (i.e. a volume control module). Explanations follow below.

```
#include <hls_stream.h>
void volume_ctl(
        hls::stream<float>& in, //source of input samples
        hls::stream<float>& out, //source of output samples
        float volume_adjust
        ) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=volume_adjust bundle=CTRL_BUS
#pragma HLS INTERFACE axis register both port=out
#pragma HLS INTERFACE axis register both port=in

        //At the beginning of your function: save the values from the axilite
        float vol_cached = volume_adjust;

        float tmp;
        in >> tmp; //Read input from input stream
        out << tmp*vol_cached; //Output modified value
}
```

A1. Setting up Your Function's Ports

When writing a module, there are a few requirements involving the ports (see highlighted section in the code):

- The first argument should be an input stream using an ***`HLS INTERFACE axis`*** pragma
  - See below for information on what data type you *must* use
- The second argument should be an output stream using an ***`HLS INTERFACE axis`*** pragma
- Any parameters you wish to add to your module must use an ***`HLS INTERFACE s_axilite`*** pragma and should all be in the same bundle
  - Note: by default, HLS places all ports listed as AXILite in the same bundle
- The function itself (i.e. the "return port") should have an ***`HLS INTERFACE ap_ctrl_none`*** pragma. This will cause the module to run continuously.

```
#include <hls_stream.h>
void volume_ctl(
        hls::stream<float>& in, //source of input samples
        hls::stream<float>& out, //source of output samples
        float volume_adjust
        ) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=volume_adjust bundle=CTRL_BUS
#pragma HLS INTERFACE axis register both port=out
#pragma HLS INTERFACE axis register both port=in

        //At the beginning of your function: save the values from the axilite
        float vol_cached = volume_adjust;

        float tmp;
        in >> tmp; //Read input from input stream
        out << tmp*vol_cached; //Output modified value
}
```

A2. Data Types for the Stream Variables

Our modules use axi stream to send and receive data between each other. In the following discussion, ***`TYPE`*** is the data type we're using for audio calculations. We're planning to use ***`float`***, but it would be possible to do all our calculations on 24 bit integers.

- You must use an ***`hls::stream<TYPE>&`***.
    - The reference is not optional; Camilo tells me that HLS will choke if you don't do this.
    - It's actually up to us what we want to use for ***`TYPE`***, but it seems like we're planning to use ***`float`***.
    - These are defined in ***`hls_stream.h`***.

- You must set the ***`INTERFACE`*** pragma for ***`in`*** and ***`out`*** to ***`axis`***

- You must use the blocking read and write, which I've shown in the above code (using the ***<<*** and ***>>*** operators)

```
#include <hls_stream.h>
void volume_ctl(
        hls::stream<float>& in, //source of input samples
        hls::stream<float>& out, //source of output samples
        float volume_adjust
        ) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=volume_adjust bundle=CTRL_BUS
#pragma HLS INTERFACE axis register both port=out
#pragma HLS INTERFACE axis register both port=in
```

```
        //At the beginning of your function: save the values from the axilite
        float vol_cached = volume_adjust;

        float tmp;
        in >> tmp; //Read input from input stream
        out << tmp*vol_cached; //Output modified value
}
```

A3. Using the AXILite bus

Any of your module's ports defined with an AXILite interface (and you can have as many as you want) are implemented with registers. Every time you write to that parameter, its value overwrites the old saved value.

This means that the port's value can change *as your module is running*. So, for this reason, it's better to read the ports once at the beginning of your function and cache them in local variables. This way, the user can change the parameters to your module on the fly.

```
#include <hls_stream.h>
void volume_ctl(
        hls::stream<float>& in, //source of input samples
        hls::stream<float>& out, //source of output samples
        float volume_adjust
        ) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=volume_adjust bundle=CTRL_BUS
#pragma HLS INTERFACE axis register both port=out
#pragma HLS INTERFACE axis register both port=in

        //At the beginning of your function: save the values from the axilite
        float vol_cached = volume_adjust;

        float tmp;
        in >> tmp; //Read input from input stream
        out << tmp*vol_cached; //Output modified value
}
```

A3.1 A Word About Synchronizing Multiple Parameters

Say you have like twelve parameters for your module, and you need to make sure that all the parameters are read together. In other words, you don't want to only have half of them updated by the time the module caches them. There is a technique you can use for this:

```
#include <hls_stream.h>
void volume_ctl(
        hls::stream<float>& in, //source of input samples
        hls::stream<float>& out, //source of output samples
        float volume_adjust,
        int thing,
        char otherthing,
        double yetanotherthing,
        int etc,
        //and so on...
        int sync
        ) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=volume_adjust bundle=CTRL_BUS
#pragma HLS INTERFACE axis register both port=out
#pragma HLS INTERFACE axis register both port=in

        //Keep restarting this module until the user has signalled that they
        //are done writing to all the parameters
        if (sync == 0) return;

        //Now cache all the parameters
        int thing_cached = thing;
        char otherthing_cached = otherting;
        double yetanotherthing_cached = yetanotherthing;
        int etc_cached = etc;
        //and so on...

        float tmp;
        in >> tmp; //Read input from input stream
        out << tmp*vol_cached; //Output modified value
}
```

Quite simply, have an extra variable (called ***sync*** in the example code above) and immediately return at the beginning of your function if it is equal to 0.

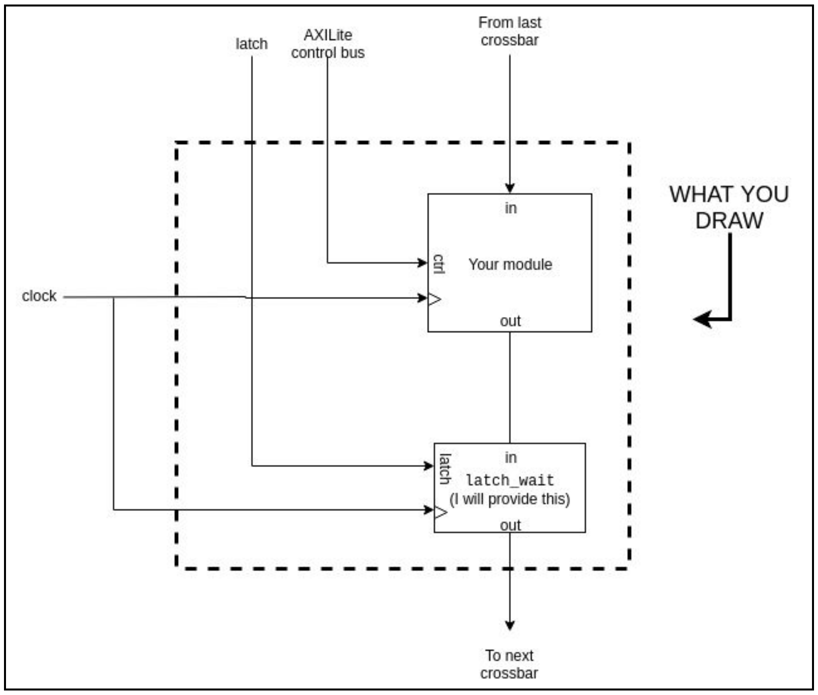
To write parameters to this function, the user will do:

- Set ***sync*** to 0
- Write to ***thing***
- Write to ***anotherthing***
- Write to ***yetanotherthing***
- etc…
- Set ***sync*** to 1

and the module will not continue until that last "set ***sync*** to 1" is executed.

A4. Adding Your Module to the Block Diagram

When you're ready to add your module to the board, you'll have to edit the top level block diagram:



Inside the dashed box is what you have to draw. Add your module as well as a ***latch_wait*** IP (which I have already created for you). Make sure to hook up the clocks and the 48 KHz pulse (i.e. the latch signal) as shown.

Appendix B - Documentation

This section provides documentation for the organization and meaning of file, as well as the register summaries of the audio modules and API for the MIDI library.

B1. Git Repository and File Structure

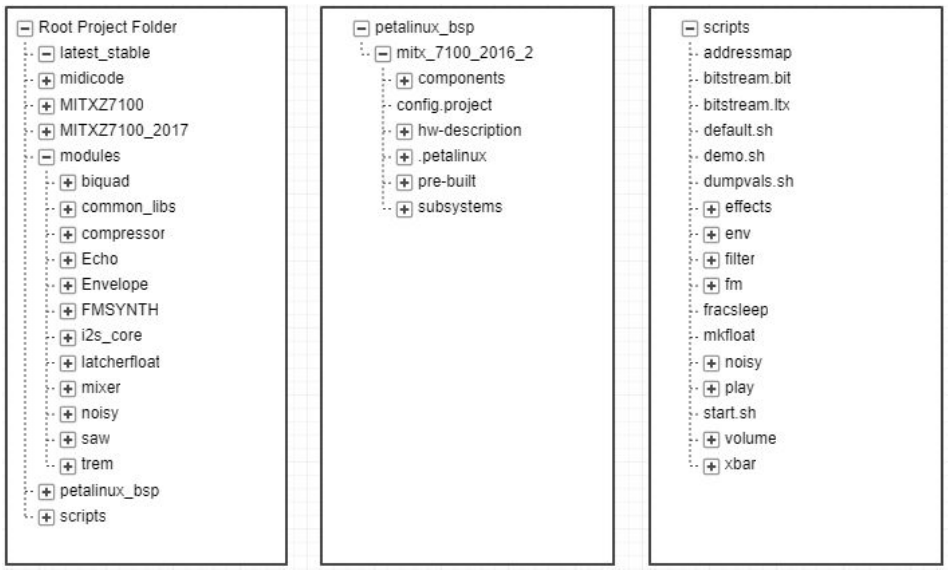
The main git repository can be found on github[2].It is currently open . All of the audio submodules are contained within their own repository and are added as independent submodules in the project.

The folder structure within the repository can be seen below, where the left-most image is the root folder and the other two images are further expanded folders from within the root project.

The important folders are:

- Latest_stable: *Stable SD card images and bitstream files.*
- Midicode: *Contains C sources and Makefile to build MIDI file player.*
- MITXZ7100: *Vivado project folder that contains a stable 2016.2 image for use with the BSP generation.*
- MITXZ7100_2017: *Vivado project of the latest FPGA image, containing the final project*
- Modules: *All submodules within the project, including all HLS cores as well as the i2s core..*
- Petalinux_bsp: *Project folder for petalinux 2016.2, containing the device tree structure from MITXZ7100.*
- Scripts: *Collection of scripts to run the demo, including a compiled binary of the MIDI file player.*

[2] https://github.com/mathild7/ECE1373_GhostSynth.git



```
⊟ Root Project Folder
  ⊟ latest_stable
  ⊞ midicode
  ⊞ MITXZ7100
  ⊞ MITXZ7100_2017
  ⊟ modules
    ⊞ biquad
    ⊞ common_libs
    ⊞ compressor
    ⊞ Echo
    ⊞ Envelope
    ⊞ FMSYNTH
    ⊞ i2s_core
    ⊞ latcherfloat
    ⊞ mixer
    ⊞ noisy
    ⊞ saw
    ⊞ trem
  ⊞ petalinux_bsp
  ⊞ scripts
```

```
⊟ petalinux_bsp
  ⊟ mitx_7100_2016_2
    ⊞ components
    config.project
    ⊞ hw-description
    ⊞ .petalinux
    ⊞ pre-built
    ⊞ subsystems
```

```
⊟ scripts
  addressmap
  bitstream.bit
  bitstream.ltx
  default.sh
  demo.sh
  dumpvals.sh
  ⊞ effects
  ⊞ env
  ⊞ filter
  ⊞ fm
  fracsleep
  mkfloat
  ⊞ noisy
  ⊞ play
  start.sh
  ⊞ volume
  ⊞ xbar
```

B2. Audio Module Register Summaries

This section presents the register interface of each module. The register locations are given as offsets past the (configurable) base address of the module. Except for the AXI Stream Switch, all modules were produced by a team member. Note that certain modules (the latcher, white noise generator, and tremolo) do not expose any configuration registers.

B2.1 I2S Core

Offset	Bit Index	Name	Type	Meaning
0x0	***[0:0]***	***En***	***bitfield***	1 - Enable I2S core, 0 - Soft reset
0x0	***[1:1]***	***Int En***	***bitfield***	Interrupt Enable - DEPRECATED
0x0	***[2:2]***	***Ch Swp***	***bitfield***	Swap L and R channels
0x0	***[4:3]***	***BuffSize***	***bitfield***	Fifo watermark for interrupt - DEPRECATED
0x0	***[10:5]***	***Samp Res***	***bitfield***	Sample resolution (32 bit to 8 bit)
0x0	***[15:11]***	***Freq Ratio***	***bitfield***	Ratio of LRCLK to MCLK - DEPRECATED

B2.2 AXI Stream Switch

Offset	Name	Type	Meaning
0x0	***Control***	***bitfield***	Writing a 1 to bit 1 causes the switch to update its internal connections
0x40 + 4n	***MI_MUX[n]***	***int***	Selects source for output ***n***. Set to ***0x80000000*** to disable that particular output.

B2.3 Saw Generator

Offset	Name	Type	Meaning
0x10	***freq***	***float***	Desired frequency in Hertz
0x18	***vol***	***float***	Desired volume, a number between 0 and 1.
0x20	***wait***	***int***	Setting this to a nonzero value causes the generator to enter a waiting state. This allows the user to ensure all parameters are updated before they are read by the module

B2.4 FM Synthesizer

Offset	Name	Type	Meaning
0x10	***Press***	***int***	Active signal to show when a note is being currently played by a user
0x18	***Modulator_Wave***	***int***	The integer offset which determines the note to read modulator wave data from
0x20	***Modulator_Phase***	***float***	Radians to determine the phase shift of the modulator wave, converted to samples based on the modulator wave
0x28	***Scale_Factor***	***float***	Scale is multiplied by the modulator amplitude either increasing or decreasing its effect on the carrier phase shift

0x30	***Carrier_Wave***	***int***	The integer offset which determines the note to read carrier data from
0x38	***Carrier_Phase***	***float***	Radians to determine the phase shift of the carrier wave, converted to samples based on the carrier wave
0x40	***User_Writing***	***int***	This value blocks the operation of blocks while registers are being updated with new waves to read and values associated with those indicies

B2.5 Digital Filter

(See Figure 4.9.1)

Offset	Name	Type	Meaning
0x10	***a1***	***float***	a_1 parameter in standard biquadratic filter
0x18	***a2***	***float***	a_2 parameter in standard biquadratic filter
0x20	***b0***	***float***	b_0 parameter in standard biquadratic filter
0x28	***b1***	***float***	b_1 parameter in standard biquadratic filter
0x30	***b2***	***float***	b_2 parameter in standard biquadratic filter
0x38	***wait***	***int***	Setting this to a nonzero value causes the filter to enter a waiting state. This allows the user to ensure all parameters are updated before they are read by the module

B2.6 Echo/Reverb

Offset	Name	Type	Meaning
0x10	***Delay***	***int***	Amount of samples between the instance it is first heard and the scaled echo of the signal
0x18	***Scale***	***float***	How much feedback the echo will provide, or the relative amplitude of the echoed signal

B2.7 Mixer

Offset	Name	Type	Meaning
0x10	***vol0***	***float***	Volume setting for input 0
0x18	***vol1***	***float***	Volume setting for input 1
0x20	***vol2***	***float***	Volume setting for input 2
0x28	***vol3***	***float***	Volume setting for input 3

B2.8 Envelope Generator

Offset	Name	Type	Meaning
0x10	***Press***	***int***	Active signal to show when a note is being currently played by a user
0x18	***Attack_Duration***	***int***	Amount of samples the attack volume of a note will increase in amplitude for
0x20	***Decay_Duration***	***int***	Amount of samples between the attack duration and the balanced sustain amplitude
0x28	***Sustain_Amplitude***	***float***	Size of the maximum amplitude the sustain region, active as long as press is active
0x30	***Release_Duration***	***int***	Amount of samples the note decreases in amplitude after the user has stopped pressing the key

B2.9 Compressor

Offset	Name	Type	Meaning
0x10	***thresh***	***float***	Amplitude at which compressor takes effect
0x18	***slope***	***float***	Slope of out/in past threshold value

B2.10 Vibrato

Offset	Name	Type	Meaning
0x10	*delay*	*float*	Different pitch effect is achieved through the use of this modulated delay variable that is configured from the host via AXI Lite configuration bus.
0x18	*depth*	*float*	The depth is used to change the intensity of Vibrato effect and is configured via AXI Lite bus.

B3. MIDI Library API

B3.1 MIDI Library Objects

The basic objects here are a MIDI event and a MIDI context struct. The definitions are as follows (this is just copy-pasted from the ***MIDI.h*** file):

```
typedef enum {
        NOTE_ON,
        NOTE_OFF,
        PROGRAM_CHANGE,
        CONTROL_CHANGE,
        SYSEX,
        TEMPO_CHANGE,
        AFTERTOUCH,
        PITCH_WHEEL
} MIDI_ev_type;

typedef struct {
        MIDI_ev_type type;
        unsigned char channel;
        unsigned char data[5]; //Note: may need to make this bigger if we want to use sysex
messages to configure the voices
} MIDI_ev;

typedef struct {
        unsigned format;
        unsigned divisions; //ticks per quarter note
        unsigned tracks;

        //...
    //A bunch of stuff used internally. For example, there are things like the
    //state variables for each track being played
} MIDI;
```

B3.2 Reading MIDI Files and Extracting Events

When writing a program to read a MIDI file, you first initialize a MIDI context struct by pointing it at a file (this syntax is inspired by the file I/O functions in the standard C library):

```
#include "midi.h"
MIDI *m = midi_open("songs/cotb.mid");
//your code here...
midi_close(m);
```

The MIDI context struct keeps a list of pending MIDI events. You can access them one by one using the getEvent function:

```
MIDI *m = midi_open("something");

//...
step_ticks(m, 1); //Explained below
//...

MIDI_ev *ev;
while(getEvent(m, &ev) != 0) {
        //use ev->type, ev->channel, and ev->data to do things...
}

//...

midi_close(m);
```

When you first initialize a MIDI context struct, it has no events stored in its queue. In order to get it to read events from the file, you need to tell it to step time. The ***step_ticks*** function does two things:

1. Discards all unread events in the queue
2. Fills the queue with any events that are scanned during the next stretch of time

For example, say you have a MIDI file with two ***NOTE_ON*** events at time 10, a ***NOTE_OFF*** event at time 20, and another ***NOTE_OFF*** event at time 30. Calling ***step_ticks(m,25)*** causes the two ***NOTE_ON*** events and the first ***NOTE_OFF*** event to be stored in m's internal event queue. If you only call ***getEvent(m, &ev)*** twice, there will still be one event left over in ***m***'s internal queue. If you then call ***step_ticks(m,10)***, that event is discarded, and the second ***NOTE_OFF*** event is added to the internal queue.

B3.3 Other details

The ***step_ticks*** function returns an integer:

- If it is negative, an error occurred, and it is probably time to abort
- If it is zero, that means we have finished reading the file
- If it is positive, there was no error, but there are still more events left

Because of what I'm doing with this code, I added a "feature" where, within the event queue, ***NOTE_OFF*** events are always placed at the start.

SYSEX events are completely ignored. In addition, all meta-events (except the tempo change event) are also ignored.

The data field of a MIDI event is filled in the same order that the bytes appear in the MIDI file

Appendix C - Schematics and Other Details

I2S or the Integrated Inter-IC Sound Bus protocol is simply a serial interface in which audio samples are sent to a CODEC chip. The I2S protocol is actually a derivative of the time-domain-multiplexing TDM audio protocol, which in left justified mode can be seen below. In our specific implementation the data clock, or bit clock (BCK) runs at 1/4th of the master clock. Furthermore there is a type of ‘start of packet’ signal, called the word clock, or LRCK which defines the start of each audio ‘word’ and runs at 1/256 of the master clock in our project.

It can also be seen that data padding occurs if the samples are less than 32 bits, in this case the 24 bit audio sample is shifted up 8 bits, but in our project the sample is shifted up and padded with zeros on the LSB.

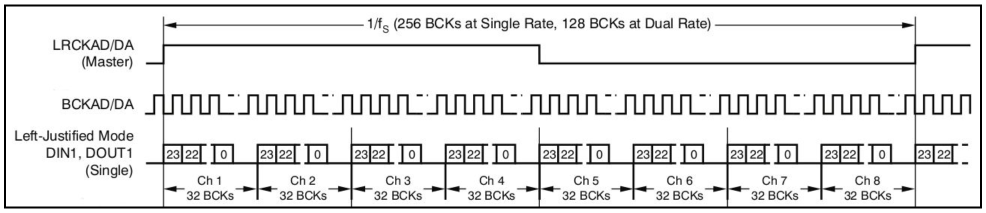
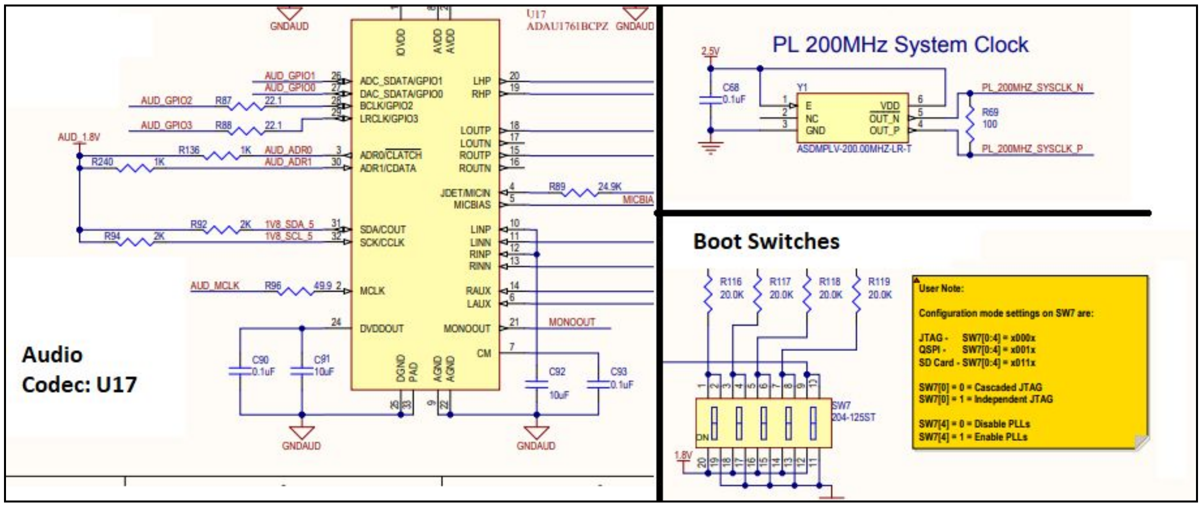


Figure C1

A few important parts of the hardware platform schematic can be seen below. These include the items most investigated during the hardware schematic review. The PL 200 MHz system clock is a differential pair clock input that directly drives the audio PLL, the boot switches define where the boot media is (in our case the SD card) and the audio CODEC itself had many features and pin connections required for the design.



User Note:

Configuration mode settings on SW7 are:

JTAG - SW7[0:4] = x000x
QSPI - SW7[0:4] = x001x
SD Card - SW7[0:4] = x011x

SW7[0] = 0 = Cascaded JTAG
SW7[0] = 1 = Independent JTAG

SW7[4] = 0 = Disable PLLs
SW7[4] = 1 = Enable PLLs

Figure C2

The startup script for the board is below, with all of the i2c writes required to program the Analog Devices ADAU1761 Audio CODEC.

```
cat bitstream.bit > /dev/xdevcfg

#Enable FPGA i2s Core
devmem 0x80002000 32 0x402
devmem 0x80002000 32 0x403

#Route I2C mux to codec
i2cset -y 0 0x70 0x0 0x20

#Codec Config
i2cset -y 0 0x3b 0x40 0x00 0x01 i
i2cset -y 0 0x3b 0x40 0x02 0x00 0xFD 0x00 0x0C 0x20 0x01 i
i2cset -y 0 0x3b 0x40 0x08 0x00 i
i2cset -y 0 0x3b 0x40 0x09 0x00 i
i2cset -y 0 0x3b 0x40 0x0A 0x01 i
i2cset -y 0 0x3b 0x40 0x0B 0x05 i
i2cset -y 0 0x3b 0x40 0x0C 0x01 i
i2cset -y 0 0x3b 0x40 0x0D 0x05 i
i2cset -y 0 0x3b 0x40 0x0E 0x00 i
i2cset -y 0 0x3b 0x40 0x0F 0x00 i
i2cset -y 0 0x3b 0x40 0x10 0x00 i
i2cset -y 0 0x3b 0x40 0x11 0x00 i
i2cset -y 0 0x3b 0x40 0x12 0x00 i
i2cset -y 0 0x3b 0x40 0x13 0x00 i
i2cset -y 0 0x3b 0x40 0x14 0x00 i
i2cset -y 0 0x3b 0x40 0x15 0x08 i
i2cset -y 0 0x3b 0x40 0x16 0x01 i
i2cset -y 0 0x3b 0x40 0x17 0x18 i
i2cset -y 0 0x3b 0x40 0x18 0x00 i
i2cset -y 0 0x3b 0x40 0x19 0x13 i
i2cset -y 0 0x3b 0x40 0x1A 0x00 i
i2cset -y 0 0x3b 0x40 0x1B 0x00 i
i2cset -y 0 0x3b 0x40 0x1C 0x21 i
i2cset -y 0 0x3b 0x40 0x1D 0x00 i
i2cset -y 0 0x3b 0x40 0x1E 0x41 i
i2cset -y 0 0x3b 0x40 0x1F 0x00 i
i2cset -y 0 0x3b 0x40 0x20 0x00 i
i2cset -y 0 0x3b 0x40 0x21 0x00 i
i2cset -y 0 0x3b 0x40 0x22 0x01 i
i2cset -y 0 0x3b 0x40 0x23 0xE7 i
i2cset -y 0 0x3b 0x40 0x24 0xE7 i
i2cset -y 0 0x3b 0x40 0x25 0x00 i
i2cset -y 0 0x3b 0x40 0x26 0x00 i
i2cset -y 0 0x3b 0x40 0x27 0xE4 i
i2cset -y 0 0x3b 0x40 0x28 0x08 i
i2cset -y 0 0x3b 0x40 0x29 0x03 i
i2cset -y 0 0x3b 0x40 0x2A 0x03 i
i2cset -y 0 0x3b 0x40 0x2B 0x00 i
i2cset -y 0 0x3b 0x40 0x2C 0x00 i
i2cset -y 0 0x3b 0x40 0x2D 0xAA i
i2cset -y 0 0x3b 0x40 0x2F 0xAA i
i2cset -y 0 0x3b 0x40 0x30 0x00 i
i2cset -y 0 0x3b 0x40 0x31 0x08 i
i2cset -y 0 0x3b 0x40 0x36 0x03 i
i2cset -y 0 0x3b 0x40 0xC0 0x00 i
i2cset -y 0 0x3b 0x40 0xC1 0x00 i
i2cset -y 0 0x3b 0x40 0xC2 0x00 i
i2cset -y 0 0x3b 0x40 0xC3 0x00 i
i2cset -y 0 0x3b 0x40 0xC4 0x00 i
i2cset -y 0 0x3b 0x40 0xC6 0x00 i
```

```
i2cset -y 0 0x3b 0x40 0xC7 0x00 i
i2cset -y 0 0x3b 0x40 0xC8 0x00 i
i2cset -y 0 0x3b 0x40 0xC9 0x00 i
i2cset -y 0 0x3b 0x40 0xD0 0x00 i
i2cset -y 0 0x3b 0x40 0xD1 0x00 i
i2cset -y 0 0x3b 0x40 0xD2 0x00 i
i2cset -y 0 0x3b 0x40 0xD3 0x00 i
i2cset -y 0 0x3b 0x40 0xD4 0x00 i
i2cset -y 0 0x3b 0x40 0xEB 0x01 i
i2cset -y 0 0x3b 0x40 0xF2 0x01 i
i2cset -y 0 0x3b 0x40 0xF3 0x00 i
i2cset -y 0 0x3b 0x40 0xF4 0x00 i
i2cset -y 0 0x3b 0x40 0xF5 0x00 i
i2cset -y 0 0x3b 0x40 0xF6 0x00 i
i2cset -y 0 0x3b 0x40 0xF7 0x00 i
i2cset -y 0 0x3b 0x40 0xF8 0x00 i
i2cset -y 0 0x3b 0x40 0xF9 0x7F i
i2cset -y 0 0x3b 0x40 0xFA 0x03 i
```

Appendix D - Shell Creation Progress Log

Milestone 1.0 a - Tool bringup & Flow generation

1. Understanding overall tool flow and Vivado operation
2. Understanding petalinux flow
3. Installing petalinux and libraries
4. Figuring out version matching between petalinux and vivado requirement along with pregenerated bsp.
5. Generating a baseline load, are the basic gpio pins working? Is I2C working?
6. I2C wasn't working, needed to set jumper pins
7. Can I make a small change? Does the flow still work?

Milestone 2.0 a - Coding new I2S core

1. Initially use semilattice premade driver: https://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/ReferenceDesigns/ReferenceDesigns02/I2SControllerwithWISHBONEInterface
2. This is finally a good candidate, its in verilog, but its wishbone
3. Turns out its just easier to write from scratch, using a 3 component core, register interface, parallel to serial interface, and clock crossing fifo into core. Still use some legacy registers from the lattice semi design.

Milestone 2.0 b - Full simulation

1. Understanding how to program registers with axi
2. Found out AXI BFM isnt licensed needed to do simulations in 2017.2. Concern for compatibility w/ BSP
3. Find workaround for compatibility. Dual load 2016.2 and 2017.2.
4. Build basic test bench to check audio, add sawtooth in HLS from Marco
5. Debug code to get valid sim

Milestone 2.0 c - Programming the ADAU1761

1. Reading through datasheet to understand how registers are configured
2. Understanding that the device is behind a MUX
3. Understanding that the device uses 16 bit data-addressing. Need to bitbang together i2cset commands

Milestone 2.0 d - Constraints & Hardware debug

1. Added pin constraints for all PL items, such as i2s core, LED pins, etc.
2. First build, is the i2s regmap writeable? Only 16 bits?
3. Obviously first build doesn't work. Tried to figure out ILA core, couldn't. Went to take board out and verify frequencies
4. Frequencies were off, namely 24kHz instead of 48kHz.
5. Figured out there was a petalinux BSP change required when changing the PL clk output. Rebuild BSP. Frequency looks good now
6. No output from the audio core still.. I see digital data coming to the correct pin to the codec.
7. Assume that something is wrong with the frequencies.. PLL doesn't seem to lock.
8. Move entire system over to 200MHz ext clk from clk gen on pcb. Find lots of clock crossing issues and proceed to fix them.
9. Add LED output to verify PLL locks. Firm indication without ILA needed
10. Sim looks way better now too. But still no audio out of codec
11. Finally figure out ILA debug after reading thru entire datasheet. Confirm that digital data looks good. Also needed to change JTAG clock.
12. Must be I2C programming. Finding sigmastudio was great, could generate premade I2C writes. Fiddle with that a little more and audio works!

Appendix E - Hints for the Next Time

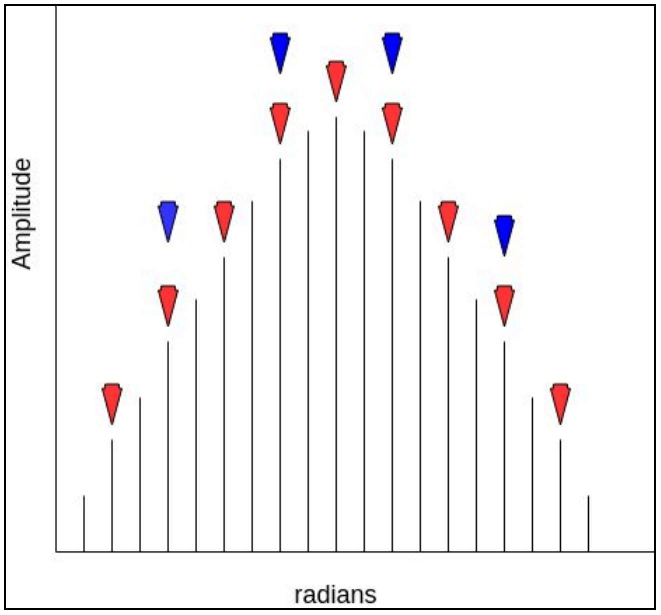
Here are a few brief hints that we have benefited from:

- Write HLS modules without loops.
    - Think of each module as a state machine, and use static variables that get updated every time the module is triggered (instead of index variables in long for loops).
- Favour a multitude of simple HLS modules connected together as opposed to a single monolithic module
- Learn how to write testbenches!
    - This is the single biggest thing that would have helped for assignments 1 and 2, and it was used to great effect for the project. It is not complicated, and is very helpful. Asic-world's "The Art of Writing Testbenches" is a really great tutorial for this.
- Read the sections in the HLS User Guide on port and block interface signals.
    - HLS has difficulty with program logic, but the thing it's really good at is managing interfaces. In fact, it's so good at managing interfaces that this virtue alone makes HLS worth the extra trouble.
- It's 100% worth it to know some basics about embedded linux.
    - Understanding how to mmap ***`/dev/mem`*** and how to use it for controlling IP cores via memory-mapped registers
    - Knowing the basics of which files are needed and when

Appendix F - FM Synthesizer Theory

The FM Synthesizer is a sine wave based audio module. There are two waveforms used to generate the tone heard by the user, the modulator and the carrier. The special case is the modulator wave is not used to generate any audio tone, the modulator wave is used to phase shift the carrier wave with respect to the cyclic nature of the modulator wave. The main functionality of an FM Synthesizer is that the carrier, the wave heard, is phase shifted in real time with respect to the modulator wave. The amount of shift is relative to the amplitude of the modulator wave at the same time instance of the carrier.

In digital signal audio, a sample is produced at the period of the sampling frequency, to ensure that the lowest possible frequency stored could be reproduced the lowest octave used is sampled at 48kHz. One period of each note in the lowest sampled octave is stored in registers. In music, when a note is increased by an octave the frequency of that note is doubled. Using the digital samples, each increase in octave results in reading each $2^{octave_increase}$th sample. In the figure below, the shown samples are the lowest octave sample. Notes annotated with red arrows are the samples read for the next octave, notes annotated with blue arrows are the samples read for 2 octaves above the sampled frequency.



Octave Note Sampling in FM Synth

Because the FM synth operates by phase shifting the carrier wave with respect to the modulator wave in time, the circuitry required to generate the sample at each clock instance would be larger and require more clock cycles per computation. Because of this reading property each modulation is a shift in memory read instance. Each note that is sampled to 48kHz has a different number of samples per period, thus the distance between two samples (in radians) varies with each note. The amplitude produced is converted to an amount of samples based on the radians per sample of the carrier wave. This phase shift calculation is added to the final read of the carrier amplitude for each instance.

Both the carrier and modulator waves have an offset of samples relative to the amount of radians each sample is responsible for. In a regular sine calculation each phase added is treated as a radian value. Whereas in the digital samples, each radian addition must relate to a total amount of samples shifted. Therefore, the conversion factor associated with each note determines the amount of samples that must be shifted at each read to produce the correct amplitude result.

The steps associated with phase shifting the carrier wave read is:

1. The phase shift value is read in by radians, converted to samples based on Modulator_Wave_Conversion

2. Modulator wave amplitude determined by reading at location [time + total_phase_shift]

3. This modulator wave amplitude is multiplied by the carrier wave conversion factor, to determine the amount of samples in the carrier registers that must be shifted for the read

4. This is summed with the Carrier_Phase*Carrier_Wave_Conversion to produce the total_Carrier_Wave_Shift

5. The final amplitude of the carrier wave at that sample instance is carrierwave[time + total_Carrier_Wave_Shift]

By following this flow, rather than calculating samples on the fly, the ability to pipeline the system for optimizations was present. All wave data was already recorded which reduced the run time constraint and reduced it down to multiplications and memory reads. The trade off is the register space required, which although not large, in a constrained design may not be feasible.