Revenge of the Synth: Polyphonic Audio Synthesis using a Field-Programmable Gate Array

Nicolas Reid[†], Callum Tilbury[‡], and Justin Wylie[§]
EEE4120F Class of 2020
University of Cape Town
South Africa
[†]RDXNIC008 [‡]TLBCAL002 [§]WYLJUS002

***Abstract*—This paper details the implementation of a classic direct digital synthesis (DDS) algorithm on a Xilinx Artix-7 field-programmable gate array (FPGA), specifically looking at its performance in synthesizing polyphonic audio. After a brief history of these topics is presented, a more robust description of the relevant signal theory is discussed. Thereafter, the implementation details using Verilog—a hardware description language—are unpacked in depth, eventually leading to a simple prototype of a digital synthesizer, using solely the FPGA board. A so-called 'golden measure' is also developed, which runs on a conventional microprocessor, the Teensy 3.6. By contrasting these two approaches, a thorough analysis of the advantages and drawbacks of hardware acceleration in this context is explored—looking at a range of factors, including the polyphonic 'voice'-count capabilities, the achievable output frequencies, power consumption, efficiency, and cost. Through these analyses, it is concluded that the FPGA offers a attractive—perhaps commercially untapped—solution to modern audio synthesis, boasting benefits of scalability, reconfigurability, and high parallelization.**

I. Introduction

Methods for synthesizing waveforms using digital circuits have been around for many decades. Tierney et al. proposed a so-called 'Digital Frequency Synthesizer' as early as 1971 [1], which is fascinating considering the nascency of the 'digital' era at that time. In the decades that followed this seminal work, much progress has been made—specifically in the methods of direct digital synthesis (DDS), as well as across the whole technological landscape. Much of the work done in improving the methods of DDS has been focused on optimizing performance, correcting for discretization effects, etc., and has been largely successful.

Another interesting development in the digital era has been that of reconfiguarble computing, specifically the advent of the Field-Programmable Gate Array (FPGA). This class of devices has been crucial in the strides made in *hardware acceleration*—that is, using specific digital hardware to perform tasks more efficiently than possible with software running on a general-purpose processor. Applications of this range from deep learning [2] to DNA sequence mapping [3].

This project aimed to explore the intersection of these two advancements, looking at the implementation of a DDS algorithm on an FPGA board. Specifically, by leveraging the parallel capabilities of a hardware-based solution, *polyphonic* audio synthesis was explored—where multiple tones of various frequencies could be played simultaneously, with minimal effects on audio latency. Such an approach acted as a proof-of-concept for more advanced parallel implementations of DDS, and broadly served as a testament to the power of both hardware acceleration and reconfigurable computing within this context. For appropriate comparison, a *golden measure* was also created—essentially the same polyphonic synthesis algorithm, but running on a simple microcontroller.

This report starts by outlining a brief background to digital audio synthesis, and discusses the motivating factors in specifically using an FPGA for such a problem. It moves on to describe in detail the structure and proposed design of a polyphonic DDS system, showing a high-level systems view via block-diagrams, and later diving into the salient code snippets which reveal key functionality. Some discussion is also mentioned surrounding the possible commercial opportunities that this system could leverage, and what would need to be improved to create a competitive product. Finally, the system is put to the test. The experimental set-up is described, along with a brief description of the golden measure implementation. Some theoretical remarks and calculations are made, including some observed unavoidable limitations for each approach, followed by a set of practical, 'real-world' comparisons. This leads naturally into a collection of detailed results, in the form of oscilloscope photos and frequency spectrum plots. Analysis is made into the achieved outputs, and a wide range of quantitative factors are considered, including achievable frequency ranges and polyphony capabilities, as well as power, cost, and efficiency. Finally, conclusions are made, and various avenues of recommended future work are mentioned.

II. Background

Many audiophiles to this day will still hold analogue synthesizers in high regard for their sound quality and organic timbre characteristics, but digital implementations will take the cake in a number of regards. The motivation for a digital implementation is rooted in the following four key points:

(1) larger frequency ranges, (2) easier reconfigurability of envelopes or filtering, (3) greater immunity to environmental interference (such as temperature, dust and volatile dielectrics) and (4) a generally smaller form-factor [4], [5].

The first point is critical for modern day communications signals. This report, however, focuses on basic *audio* signal generation. But, this is not to say that the investigation is completely unrelated; the underlying principles are still the same.

The other advantage factors will be common across most digital implementation schemes. For example, most microprocessors will come with built-in capabilities and software libraries for handling impressively wide ranges of digital signal processing. Why then, is it necessary to move to the far more complicated realm of HDL and FPGAs?

The answer lies in the parallel computing capabilities and direct clock access provided by an FPGA. Many operations in the field of digital synthesis and DSP are highly parallelizable and as such, exploits in the field (such as DDS) can benefit immensely through the parallel digital acceleration achieved in an FPGA implementation. The benefits are realised in two main categories: higher frequency generation [6], and maximum simultaneous output channels/'voices'.

Furthermore, it is expected that additional bonuses may present themselves along the way. Investigations will be carried out accordingly – looking into power consumption, resource usage and scalability.

The downside of an FPGA implementation is that they tend to be enormously expensive when compared to their sequential counterparts. This is an important factor which must be kept in mind for any future product proposals.

III. Methodology

A fundamental philosophy in the design of this project was the introduction of 'levels of abstraction'. This entailed progressively building layers of complexity, and incrementally achieving higher levels of functionality, while simultaneously forgetting about the minute details of lower levels. Such an approach is easier for development, debugging, and overall reliability. Furthermore, it helped massively with scalability, thus allowing for easy inclusion of 'stretch' goals—plans for, if time permitted, more advanced functionality built upon the interfaces of the core modules.

A. Core Modules

In essence, direct digital synthesis is a method for generating periodic, discrete-time waveforms, using digital processes [7]. There are several core elements to this algorithm, starting with generating sinusoidal values (or whatever values are relevant for the waveform), and then iterating through these values at a specific rate (which is based on the target frequency), in order to create a changing output signal of the appropriate waveform. Polyphony takes this one step further, and sums several unique changing waveforms together, thus creating a single output signal containing multiple frequency components. High level descriptions of these algorithms follow.

1) Sinusoidal Waveform Generation: Fundamental to this project is the calculation of a single sinusoidal waveform's values. Indeed, other waveforms were created by the FPGA—square, sawtooth, triangle—but doing so was trivial in comparison, as they relied on simple arithmetic. The sinusoid, on the other hand, presents an interesting design decision. One can implement, using some clever mathematics, trignometric functions that are synthesizable in Verilog (as is discussed in [8], for example). While doing this provides flexibility in the calculations, it is terribly inefficient for the DDS use-case, as calculations would be needlessly repeated many thousands of times every second. Instead, a look-up table approach was taken—where predefined sinusoid values were stored on the FPGA in BRAM, for a range of phase values. When a sinusoid value was to be used, it could simply be read from this memory block.

Fortunately, the look-up table can be optimized, by recognizing the symmetry of the sinusoid waveform. Consider one full period, $\omega \in [0, 2\pi)$, of $y(\omega) = \sin(\omega)$ in figure 1.

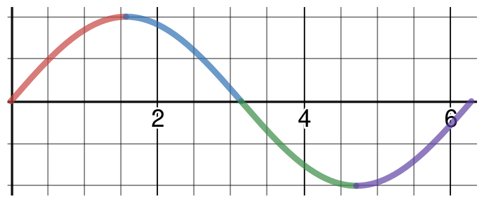


Fig. 1: Graphic showing one full period of $\sin(\omega)$

Notice that the graph can naturally be split into four segments—as is shown via the colouring—and that the latter three segments are all simple translations of the first segment. That is, suppose:

$$y_{\frac{1}{4}}(\omega) = \{\sin(\omega) | \omega \in [0, \frac{\pi}{2})\} \tag{1}$$

It can easily be seen that:

$$y(\omega) = y_{\frac{1}{4}}(\omega) + y_{\frac{1}{4}}(-\omega+\pi) - y_{\frac{1}{4}}(\omega-\pi) - y_{\frac{1}{4}}(-\omega+2\pi) \tag{2}$$

This result implies that instead of storing the entire period of samples for the sinusoidal wave, one need only store the first quarter. Thereafter, simple translations can be used to calculate the other three quarters. With this in mind, consider the block diagram for the sinusoid module, called `fullsine_256`, shown in figure 2. This module requires an 8-bit sample number (between 0 and 255), and returns a 11-bit value (between 0 and 1024), representing the relevant sine value.

Importantly, this diagram depicts the first layer of abstraction. The memory block holding the first quarter of the sine wave values is interfaced inside the `fullsine_256` module, and accepts a 6-bit index—i.e. between 0 and 63. The relevant transformations for indices between 64 and 255 are calculated in this module too. Externally, however, these details are extraneous, as the module simply requires a sample index and returns the appropriate sine value. After this module was coded and working, the specifics of the implementation could be ignored.

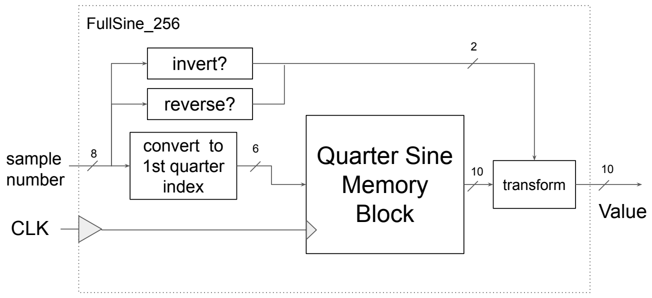


Fig. 2: Block diagram for `fullsine_256` module

2) Function Generator: The next module to be discussed was arguably the bedrock for the direct digital synthesis algorithm, and it enabled *monophonic* audio synthesis. With the `fullsine_256` module defined, the next challenge was to create *changing* waveforms—that is, a way of iterating through the sinusoid look-up table values, at some rate, such that a sinusoid signal was generated on the output. Moreover, it was important that the frequency of this waveform could be specified.

Consider the general case of a look-up table with n samples, which defines a single period of a particular waveform. Suppose, then, this waveform should be played on the output with a frequency of f_0, which is a period of T_0. Notice that the address of the look-up table elements must be incremented every $\frac{T_0}{n}$ seconds—call this the 'tick period', T_t. Figure 3 shows a diagram of this, in the sinusoidal waveform case.

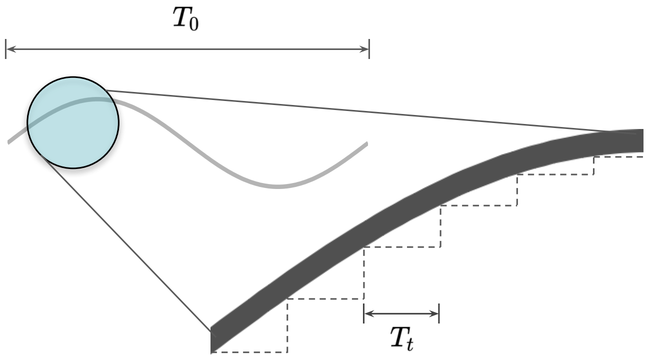


Fig. 3: Graphic showing the relationship between the waveform period, T_0, and the tick period, T_t

The 'tick *frequency*' is clearly equal to:

$$f_t = \frac{n}{T_0} = nf_0 \tag{3}$$

In order to achieve this frequency, one can scale down the 100MHz clock by a factor, α, which is a function of the desired waveform frequency, f_0:

$$\alpha(f_0) = \frac{100 \times 10^6 \text{ Hz}}{f_t} = \frac{100 \times 10^6}{nf_0} \tag{4}$$

Scaling the main clock frequency is a fairly simple process. Essentially, on both the rising and falling edges of the main clock, a counter should be incremented; when this counter reaches $(\alpha - 1)$, the *scaled* clock should be toggled on the next main clock edge.

The `index_generator` module performed these discussed tasks. It accepted a given tick period, T_t, which was used to scale the input clock frequency of 100MHz. The module's output was then the current index of the desired waveform, at that moment in time (hence, *index* generator). In the literature, this block is often called the *phase accumulator* [7].

The current index, however, was not the end goal for the function generation—as it still needed to be converted to an actual waveform magnitude. Thus, the output of the index generator was tied to the input of a `sample_generator` module—also known as the *phase-to-waveform converter* [7].

This module accepted the phase index, as well as a 'waveform definition'—with enumerated options of sinusoid, sawtooth, square, and triangle. This module then, based on the chosen waveform definition, outputted the corresponding magnitude value for the given phase index. For the sinusoid wave, the module contained a `fullsine_256` module, whereas for the other waveforms, simple arithmetic could be used: the index generator's output signal was in fact already a sawtooth wave, and thus needed only to be scaled appropriately; the square and triangle waves were similarly trivial.

The `function_generator` module was the simple composition of these two sub-modules. The modules are all shown together in figure 5.

3) Mixer: The function generator module was able to generate *monophonic* audio, but nevertheless lacked the ability to play two or more notes simultaneously. Instead, upon yet another layer of abstraction, the `mixer` module was defined. This module was actually purely combinational—it did not require a clock signal—and simply received the outputs from multiple function generators, and combined them. The essence of this requirement is shown in the block diagram of figure 4, with N function generators, each with a unique tick period.

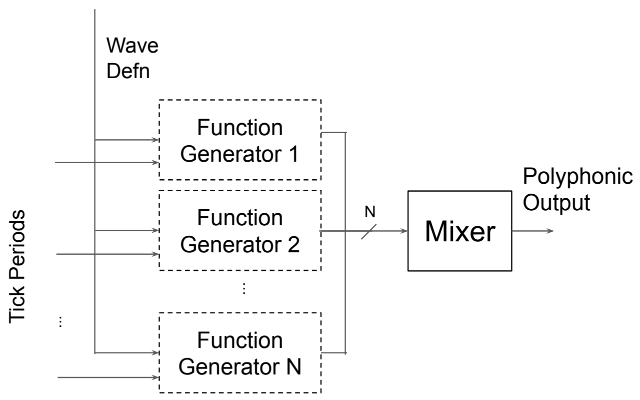


Fig. 4: Block diagram showing the use-case of the `mixer` module

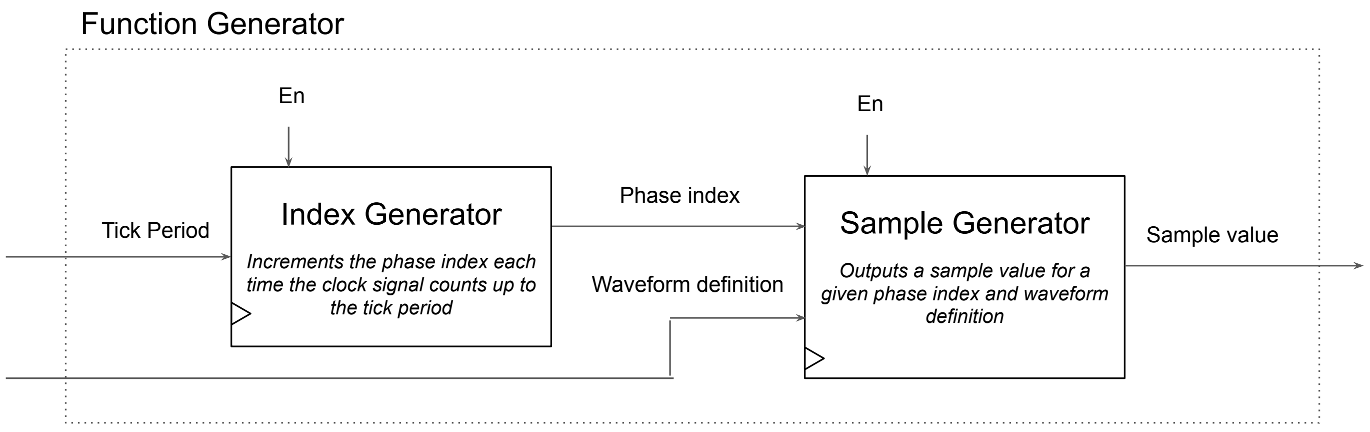

Function Generator
En
En
Tick Period
Index Generator
Increments the phase index each time the clock signal counts up to the tick period
Phase index
Waveform definition
Sample Generator
Outputs a sample value for a given phase index and waveform definition
Sample value

Fig. 5: Block diagram for the `function_generator` module (Note that in the literature, the `index_generator` module is referred to as the *phase accumulator*, and the `sample_generator` as the *phase-to-waveform converter* [7])

B. Interfacing Modules

The modules discussed so far have been part of the core DDS functionality—generating waveforms, and mixing them together. While these components formed the crucial elements of polyphonic synthesis, it was also important to be able to *interact* with the system. When packaged together as an 'audio synthesizer', the FPGA required modules for interacting with buttons, displaying values on the seven-segment displays, as well as actually writing the polyphonic signal to an audio output via pulse-width modulation. Some of the modules responsible for these functions included the `debounced_button`, `bcd_decoder`, `decimal_to_BCD`, `ssdriver`, and `pwmor`.

Note, however, that despite these modules being important, their implementation is regarded to be generally straightforward. Since they did not contribute to the core of the DDS, they are not described in detail in this paper.

C. Signal Shaping

Some early investigation was done into the shaping of the audio output signals. Specifically, an ADSR (attack, delay, sustain, release) envelope—as called in the audio synthesis community—was created. This allowed a shaping of the amplitude of each note, during and after it was pressed on the synthesizer. For example, the parameter of attack indicates how quickly the note rises to its full amplitude. These effects are a key component in a decent synthesizer, and they allow for great versatility in the production of electronic music.

D. Future Modules

Since the time for this project was fairly restricted, much of the desired additional functionality could not be included. These things would arguably improve the overall system and its performance as an audio synthesizer. Examples include the addition of analog input controls (using the XADC tools on the Xilinx) for things like pitch bending and filter control, the reading and recording of MIDI files (using the microSD card slot, or some of the external pin connections), and so on. Unfortunately the investigation and implementation of these modules were out of scope. Nevertheless, the project is highly scalable, and such features would likely be easy to integrate into the existing setup.

IV. Design

The system is set out with the hope that it will eventually operate as a stand-alone product. This means that, as an end product (further theorised in Section V.), the system would not depend on a host. Peripherals might be added (larger keyboards, analog inputs, communication links for reconfiguration, etc.), but the core system would remain functional on its own: an FPGA/sized-down version, an array of keyboard buttons, some settings buttons and an output speaker. The rough idea is laid out in figure 6 below.

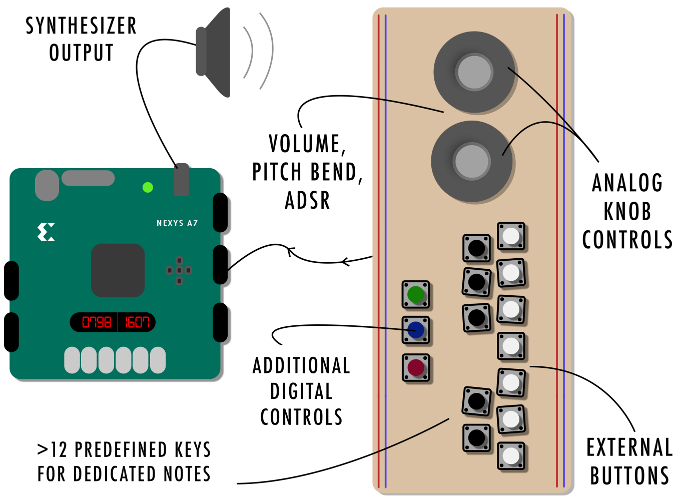


Fig. 6: A simplified concept sketch showing the FPGA as the core element interfacing with various peripherals

The FPGA serves to receive user inputs, process data and output formatted results. It should produce repeatable, consistent results, unless (as mentioned in Section V.) the user decides to alter/customize specific code sections. The following design sections detail each of the aforementioned Verilog modules, which come together to form 'The Revenge of the Synth'.

A. `fullsine_256`

To save on memory space, a smaller LUT, holding quarter sine wave samples, is used to extract the appropriate sample value when requested. This requires knowledge of the current sample number, and subsequently inverts or reverses the value of the quarter sine wave value if appropriate. Listing 1 presents an outline of the methods used.

Listing 1: Invert and reverse logic from `fullsine_256` for a quarter sine wave LUT.

```
//Determine the states of 'reverse' and 'invert' flags
if (&sample_num[7:6] && |sample_num[5:0]) begin
    //Code here to deal with 193-255 (63 samples)
    //Fourth Quart (1:63)
    reverse = 1;
    invert = 1;
    address = sample_num[5:0] - 1;

...

end else begin
    //Code here to deal with 0-64 (65 samples)
    //First Quart (1:64) + First Quart (64)
    reverse = 0;
    invert = 0;

    if(sample_num[6])
        address = 6'b111111;
    else
        address = sample_num[5:0];
end

//Perforom reverse addressing if required
if (reverse)
    addra = 6'd63 - address;
else
    addra = address;
```

In short, the module takes in a clock signal (`CLK100MHZ`) and the current index number (`sample_num`), and outputs the corresponding next sample value (`value`). It will be called upon by the `sample_generator`.

B. `index_generator`

The sine wave is one example of the waveforms used in this system. As alluded to above, the discrete sample values are indexed from 0 to 255. The `index_generator` module dictates the time taken to increment this index value, depending on a desired frequency. It relies on the clock signal as a base frequency and a pre-calculated tick period (`ticT`) as follows in listing 2.

Listing 2: Index incrementation performed in the `index_generator` module.

```
always @(posedge clk) begin
    // Increment tick counter
    ticCount <= ticCount + 1;

    // Check if cap has been reached
    if (ticCount >= ticT) begin
        // The next sample value is needed -> increment phase
            ↪ index
        phIndex <= phIndex + 1;
        // Reset tick counter
        ticCount <= 1;
    end
end
```

C. `sample_generator`

The sample generator follows on from the index generator, using the phase index value (0-255) to access/calculate the sample value, depending on the requested waveform. Waveforms are identified in code using the following enumerated parameter constants.

Listing 3: Setup of waveform identifier constants.

```
parameter [1:0] SINE = 2'b00;
parameter [1:0] SQUARE = 2'b01;
parameter [1:0] TRIANGLE = 2'b10;
parameter [1:0] SAW = 2'b11;
```

The sine wave samples are simply read from the `fullsine_256` module. Sawtooth, triangle, and square wave samples are generated as follows in listing 4.

Listing 4: Code to return the specified wave form's sample value at the current phase index.

```
//...
else if(waveform == SQUARE) begin
  // Set square wave sample value
  if(phIndex < 8'd128) begin    // First half
    sampVal <= 11'd2047;
  end
  else begin                    // Second half
    sampVal <= 11'd0;
  end
end

else if(waveform == SAW) begin
  // Set sawtooth value (propto phase index)
  sampVal <= (phIndex<<3);
end

else if(waveform == TRIANGLE) begin
    // Set triangle wave sample value
  if(phIndex < 8'd128) begin            // First half
    sampVal <= (phIndex<<4);
  end
  else begin                        // Second half
    sampVal <= 2047-(phIndex<<4);
  end
end
// ...
```

D. `function_generator`

The Function Generator simply encapsulates the Index Generator and Sample Generator modules; offering a sense of abstraction and ease of access for modules higher up.

E. `mixer`

The mixer first adds all signals which are currently being requested. Each signal can be seen as a separate voice channel. A `enabled` bitmask is used to enable/disable signals on demand.

Listing 5: Code to add signals that are being requested.

```
wire [15:0] signal_mixed_prescale =
    enabled[0] * signal_a +
    enabled[1] * signal_b +
    // ...
```

The next step is to normalise the overall output signal, i.e. the result of combining/'mixing' the channels. This is done as follows:

Listing 6: Code to normalise signals that are being requested.

```
assign signal_mixed = (signal_mixed_prescale >> 5);
```

Note that the prescaled sum value is stored as a 16-bit value, and thus must be bit shifted by 5, back to 11-bits. This ensures that the output signal will not saturate and cause unwanted audio distortion (i.e. ensures that `signal_mixed < 2047`).

F. Background & Ported Modules

The following modules are excluded from the full design discussion because they were either derived from code provided by the EEE4120F department, or remain non-crucial to the core of this project.

- `ssdriver`: Used to write to the seven segment display.
- `debounced_button`: Stabilises button states on transition edges (pressing/releasing).
- `pwmor`: Converts samples to a PWM signal for the audio jack.
- `decimal_to_bcd`: Converts an integer value to binary-coded-decimal format.
- `bcd_decoder`: Converts a single digit (0-9) into a seven bit code which will be used to display the digit on the seven-segment LED display.

V. Proposed Development Strategy

The discussion so far has shown how the methods of direct digital synthesis can be applied on an FPGA for polyphonic audio creation, taking advantage of the highly 'parallelizable' nature of such a device. Further advantages include reconfigurability, scalability, and power consumption. Importantly, the use of an FPGA in this field presents a great commercial opportunity.

In order for this to happen, though, further development is required. The exploration done in this project resulted only in a prototypical audio synthesizer device, and was a fairly simple iteration thereof. Audio synthesizers have been around for many decades, and are highly common in the music industry for a broad range of genres. As a consequence of this, the technology is well developed, and there are a host of standard features which one would expect in such a product. In order to compete in the market, these standard features are required, at least.

For example, modern synthesizer devices almost *always* have a MIDI interface, due to the ubiquity of MIDI in digital recording workstations (DAWs) and contemporary music production. Moreover, synthesizers usually come with a massive array of built-in sounds, effects, filtering options, and other tools. Finally, user interaction with the devices is usually simple, intuitive, and versatile.

This is not to say that the FPGA cannot compete in the market. With further time (and money), the aforementioned features are all physically realizable, and some may even operate faster or more efficiently due to being implemented in hardware. The device is particularly scalable, where the addition of polyphonic voices, functional blocks, and other audio tools should not noticeably affect the latency of the output signal. With sufficient effort, it is likely that the FPGA could be on par with the conventional synthesizers on the market.

Crucially, though, this development process must be taken one step further. One should not only aim to emulate the existing devices on the market with the FPGA solution. Rather, one should leverage the characteristics of the hardware, and thus create a competitive *advantage*. It has already been pointed out that the FPGA can be highly scalable, and this offers great benefits to its functionality. One can think even more creatively, though.

For example, imagine a product which promotes the idea of 'build-your-own-synth' or 'create-your-own-effect'. Such a device could be built upon the foundations from this project, where direct digital synthesis is implemented on an FPGA. Furthermore, a 'drag-and-drop'-styled graphical application could be developed, where a user can design and implement a wide variety of sounds, filters, effects, and more, and link them into a full signal chain. This design could be compiled into synthesizable Verilog code, which could be further tweaked, if necessary. Finally, the code could be uploaded to the board, thus creating a user-defined custom synthesizer device.

Key to this idea is the reconfigurability of the FPGA platform, together with its parallelizability, creating something which is simply unattainable on a conventional serial processor. This way of thinking—though here just manifested as an initial, simple product idea—is vital to commercial competitiveness of an audio synthesizer based on an FPGA platform. That is, one should innovate and rethink conventional processes, taking advantage of the power of hardware.

Other opportunities exist for FPGA-based digital synthesis, ranging from telecommunications to equipment testing. However, since this project focused solely on audio synthesis, these alternative applications are not discussed further.

VI. Experimentation

A. Golden Measure Overview

It is critical to note that this project was not simply to design the FPGA implementation of a DDS algorithm, nor was it purely to create a polyphonic synthesizer. Instead, these achievements were pursued in order to understand the benefits and/or drawbacks of using the FPGA in such applications, as opposed to a conventional method like using a microcontroller. It was vital then to develop simultaneously a so-called 'serial implementation', to which the FPGA solution can be compared—a *golden measure*.

It was decided that the PJRC Teensy 3.6 development board would be used to implement this alternative approach. The board has a 180MHz ARM Cortex-M4 processor—which felt like a good candidate for fair comparison with the Nexys A7. Interestingly, the Teensy has great audio support from a large open-source community, and its 'Audio' library is truly impressive. However, to compare the Teensy with the FPGA implementation accurately, it was imperative for similar (ideally identical) DDS algorithms to be running on each platform—that is, it would be unfair to compare the sophisticated and well-developed polyphonic audio synthesis libraries running on the Teensy, whereas on the FPGA, the solution has been designed from scratch.

The heart of the golden measure implementation lay with the Teensy's IntervalTimer library[1], which wraps functionality around the board's Periodic Interrupt Timers (PIT). Essentially, a timer object is created with a specified microsecond interval value, and set-up with a interrupt routine. This routine is then periodically called, in intervals of the microsecond value. An example of setting up a timer is shown in listing 7.

Listing 7: IntervalTimer basic set-up

```
IntervalTimer timer;
timer.begin(/*Interrupt Routine*/,/*Microsecond Interval*/);
```

For each timer, a separate counter variable was created. In the interrupt routine, which was written as a lambda expression for both simplicity and speed, the counter was increased by 1. This had the effect of incrementing the counter at a specified frequency, determined by the microsecond interval. This process emulated the FPGA's `index_generator` functionality. On the output audio, then, the counter was used as an index for the sinusoid look-up table. Note that, as done in the FPGA solution, only a quarter-period of the sinusoidal wave was stored. Pseudocode showing the process for one voice is given in listing 8.

Listing 8: Key code snippets from the Teensy implementation

```
IntervalTimer timerA;
unsigned char counterA = 0;
const int wave[] = {/* 64 values for first quarter period */};
float freqA = 261.63; // Example of a Middle-C note

void setup() {
    // Start the IntervalTimer
    timerA.begin([](){++counterA;},(1.0e6)/(freqA*256.0));
}

void loop() {
    // Output audio
    analogWrite(audioPin, wave[quarterToFull(counterA)]);
}
```

For each voice in the arrangement, a separate timer was created, and that voice's interrupt incremented a unique counter. These counters ranged from 0-255 (stored as `unsigned char` types, so they overflowed at 256 automatically), and referred to the current index in the *wave* array, from which the current *sample* was read. The samples from each of the voices were summed and scaled, and played on the audio pin via a standard PWM library. Since the different timers had different microsecond intervals, the resulting audio was polyphonic.

As a side note, such an algorithm is not in fact completely 'serial'. There are four PITs on the Teensy, and all can be used essentially independently—which is, in a sense, a parallel implementation. This does not mean that it fails to serve as a comparative tool, though. Nevertheless, a stricter 'serial' approach could also considered: where only *one* of the timers is used (though, technically, this is still a somewhat parallel approach).

[1]https://www.pjrc.com/teensy/td_timing_IntervalTimer.html

B. Theoretical Considerations

1) Voice Count: Because the golden measure solution uses a hardware timer for each voice in the polyphony, it is known that the Teensy is limited to four simultaneous tones (as it has four Periodic-Interval timers). It is difficult to compare this metric upfront with the Nexys A7 board, however, because the number of look-up tables required for the FPGA solution must be determined empirically from the synthesized bitstream information. Having said that, the board has over 15 000 logic slices, each with several look-up tables and flip flops. On the surface, then, it seems that the FPGA will be able to accommodate many simultaneous voices—far more than four.

2) Upper Frequency Limit: Both approaches to the audio synthesis problem faced an interesting theoretical constraint: an upper frequency limit. Firstly, consider the case of the Teensy, which uses an interrupt routine to increment the counter. Listing 9 shows the member function for 'starting' the IntervalTimer object.

Listing 9: *begin()* member function in IntervalTimer library

```
bool begin(void (*funct)(), float microsec) {
  if (microsec <= 0 || microsec > MAX_PERIOD) return false;
  uint32_t cycles = (float)(F_BUS / 1000000) * microsec - 0.5;
  if (cycles < 36) return false;
  return beginCycles(funct, cycles);
}
```

Notice the calculation of *cycles*:

$$\text{cycles} = \frac{\text{F\_BUS}}{1 \times 10^6} \times \text{microseconds} - 0.5 \tag{5}$$

where `F_BUS` is the peripheral bus clock frequency. 'Cycles' defines the number of ticks of the `F_BUS` clock between successive calls of the interrupt function. Crucially, the Cortex-M4 requires at least 24 cycles for entering and exiting the interrupt routine [9]. Additionally, the interrupt functionality itself requires a certain number of clock cycles, and there are also possibilities of cache misses, etc. To be safe, the minimum cycle count allowed is defined to be **36**, as already seen in listing 9.

The Teensy 3.6 has a standard `F_BUS` of 60MHz, which means—after rearranging equation (5)—the minimum possible microsecond interval allowed is:

$$\text{minimum}(\mu s) = \frac{(1 \times 10^6)(\text{cycles} + 0.5)}{\text{F\_BUS}} = 0.608\mu s \tag{6}$$

Importantly, this is the period between *index* increments. One full period of the output sine wave is 256 samples, and thus requires 256 increments. As a consequence, the minimum output *period* is $256 \times 0.608\mu s$. This means a *maximum* frequency of:

$$\text{maximum}(f_{\text{out}})_{\text{Teensy}} = \frac{1}{256 \times 0.608\mu s} = 6424.8\text{Hz} \tag{7}$$

Note that this frequency falls far below the maximum audible frequency (for humans) of around 20kHz, and does not even cover all the notes on a conventional piano.

It is possible to overclock the Teensy, and doing so does improve the situation slightly—though it is not necessarily a sustainable or reliable approach. For the 3.6 model, the primary clock can be pushed to 256MHz, and the peripheral bus clock to 128MHz. Following the same logic as above, the maximum output frequency then increases to 13.70kHz—which still fails to cover the full audible range.

The FPGA, on the other hand, is able to use the *raw* clock signal of 100MHz, and create in *hardware* a tick counter—as opposed to regularly calling an interrupt routine. Using similar calculations, its maximum theoretical frequency is thus:

$$\text{maximum}(f_{\text{out}})_{\text{FPGA}} = \frac{1}{256 \times \frac{1}{100\times 10^6}} = 390.6\text{kHz} \tag{8}$$

This upper limit is well outside the audible range, which is a great sign for the FPGA, in stark contrast to the poor result from the Teensy 3.6.

3) Discretization Effects: Notice that regardless of the board used, the method of direct digital synthesis increments an *integer* counter based on some tick frequency. Naturally, then, not every frequency can be achieved, even if it falls below the previously discussed maximum frequency—there is essentially a discretization error.

Say, for example, one wanted to generate a 3520Hz sine wave—an A7 note in twelve-tone equal temperament tuning. For a waveform definition spanning 256 samples, according to equation (3), the look-up table index should be incremented every $T_t = 1.101\mu$s. For the FPGA, using its 100MHz clock directly, this requires a scaling factor, $\alpha = 110.97$. Of course, this scaling factor must be a whole number, which means $\alpha \approx 111$. Theoretically, the effect of rounding off this value is that the actual frequency played on the output is 3519.14Hz—an error of only 0.86Hz.

Compare this to the Teensy, which has to use periodic interrupt calls to scale its peripheral bus clock of 60MHz. Moreover, recall that each interrupt must take at least 36 cycles. Following through with the calculations, this limitation theoretically restricts the Teensy to playing 3255.21Hz for the A7 note, which is an error of 264.79Hz. Such a large error is certainly noticeable, even to the untrained ear.

Hence, even though both boards suffer from discretization problems, the FPGA is theoretically in a far stronger position in this regard.

C. Planned Practical Considerations

Moving past the theoretical considerations, it was important to test the two solutions practically, and evaluate their respective performances. To do this, a handful of test procedures were defined, and these are specified briefly in the following paragraphs. Note that due to unforeseen stay-at-home lockdown circumstances (due to the ongoing COVID-19 pandemic), access to a modern digital oscilloscope was limited. Instead, an old analog oscilloscope was used, and physical photographs were taken of the CRT screen for this report. While detailed measurements were sacrificed in this approach, the shapes of the output waveforms were nevertheless shown. Where more specific frequency measurements were required, a simple spectrum analyzer phone application was used.

Firstly, the simple monophonic case was tested, across the various waveform definitions. Recall that four definitions were designed: sinusoidal, triangle, sawtooth, and square. Each of these was tested on both the Teensy (golden measure) and FPGA boards. Verification was done via inspection on the oscilloscope, and the spectrum analyzer.

Next, the polyphony was checked. This was initially done with just two voices—the addition of two sinusoid waves, as well as the addition of two triangle waves. These results were checked on the oscilloscope, for both the Teensy and FPGA implementations. Thereafter, a 12-note polyphonic test was run, which could only be done on the FPGA (since the Teensy is limited to 4 separate voices). Because this signal would be difficult to visualize on an oscilloscope, the spectrum analyzer application was used here instead—with the goal being 12 distinct frequency peaks in the spectrum.

It was then important to test the theoretical claims of frequency accuracy and range discussed in section VI-B2. Both boards were tested at a host of values, and the errors were considered. Also, the Teensy was tested below- and above its theoretical maximum of around 6400Hz—confirming whether this was indeed the case. This step was not necessary for the FPGA, as its output covered the entire audible frequency range. Instead, it was simply confirmed that the board could play high frequencies within hearing.

Though not discussed in detail in this report, the ADSR envelope was then applied and tested. Unfortunately, the rudimentary analog oscilloscope prevented decent measurements of such tests. Instead, a testbench written and run for the ADSR module, simulating the salient functionality.

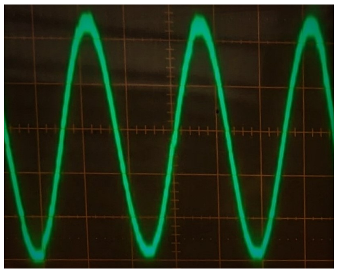
Finally, the power usage of both implementations was investigated briefly. Doing so for the Teensy was limited to a handful of current and voltage measurements, whereas on the FPGA, a more detailed breakdown was given via the Xilinx Vivado suite of tools.

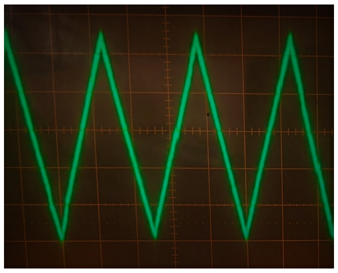
VII. Results

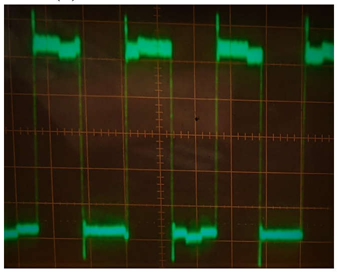
A. Waveform Definitions

The first important requirement was to have the ability to synthesize multiple waveform definitions, and be able to switch between them on-the-go. Figure 7 shows each of the four implemented shapes being generated at ‘middle C’ frequency, 261.6Hz.

Notice the thick, generally smooth plotting—this is due to the slightly coarse nature of the analogue oscilloscope which was used. The overshooting present at the discontinuities in the square and sawtooth waves is a natural effect of rapid changes in the output value, as well as some oscilloscope inaccuracies.


(a) Sine wave: 261.6Hz


(b) Triangle wave: 261.6Hz


(c) Square wave: 261.6Hz

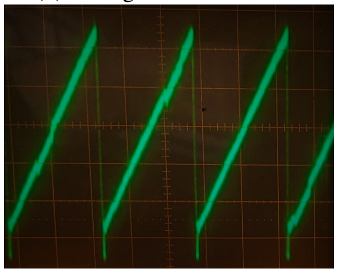

(d) Sawtooth wave: 261.6Hz

Fig. 7: Examples of various waveform definitions, all at middle-C

B. *Polyphony*

In these tests, two or more notes were played at the same time. Three specific cases are considered: mixing two sinusoidal waves, mixing two triangle waves, and mixing 12 sinusoidal waves. The former two—seeing as they are fairly simple—are shown on the oscilloscope screen, whereas the latter is shown on a spectrum analyzer.

1) Mixing Two Sine Waves: Figure 8 shows the result of mixing two sinusoidal waves, the notes C4 and F4.

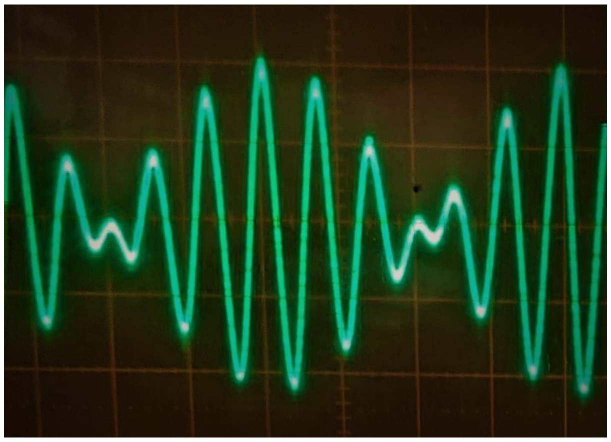


Fig. 8: Successful polyphony: mixing two sine waves.

Note the interesting polyphonic shape. This waveform was confirmed to be correct by comparison with computer plotting software.

2) Mixing Two Triangle Waves: Figure 9 shows the result of mixing two triangle waves, again with the notes C4 and F4.

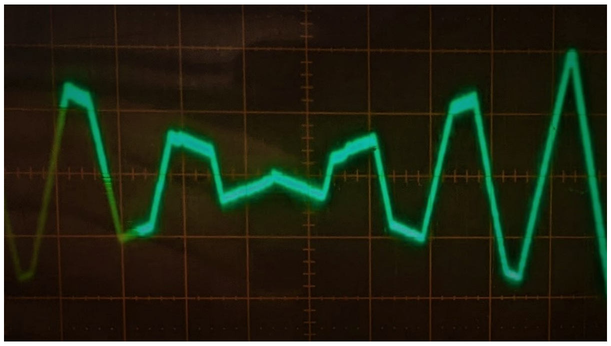


Fig. 9: Successful polyphony: mixing two triangle waves.

Similar to the sine wave version, the triangle waves can be seen to overlay each other, where the higher frequency oscillates within the envelope of the lower.

3) Playing 12 Channels Simultaneously: A cellphone-based spectrum analyser was used to inspect the result of playing 12 separate frequencies at once. Figure 10 below clearly shows 12 distinct peaks.

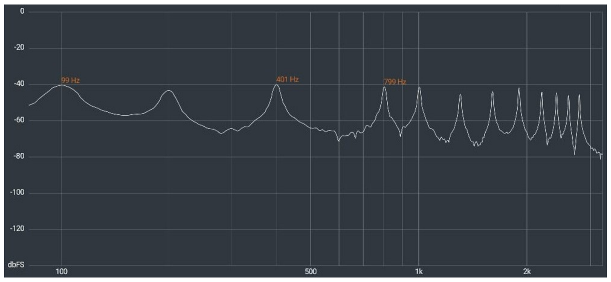


Fig. 10: Twelve channel polyphony. 100Hz – 2.8kHz.

Note that the frequency axis is logarithmically scaled, and this is why the lower frequency peaks appear to be wider and more rounded.

It is supposed that the slight differences in magnitude (despite being equal amplitude sinusoids) may be due to ambient noise interference, or perhaps the frequency characteristics of the output speaker through which the audio is playing.

C. *Frequency Accuracy*

The note ‘middle C’ is positioned precisely at 261.63Hz. The discretization error, calculated here, theorises that the FPGA implemetation should only be inaccurate by 0.01Hz. According to the (admittedly, rather crude) measurements in figure 11, the error appears to be slightly larger—instead missing the mark by at least 0.14Hz.

The reason for this is uncertain. Nevertheless, the difference is audibly imperceptible and it is supposed that the the basic phone application used for this analysis might be partially to blame.

The Teensy implementation was theorised to be over 1.2Hz more inaccurate at the middle C frequency. However, test results were inconclusive due to the lack of precise instrumentation available.

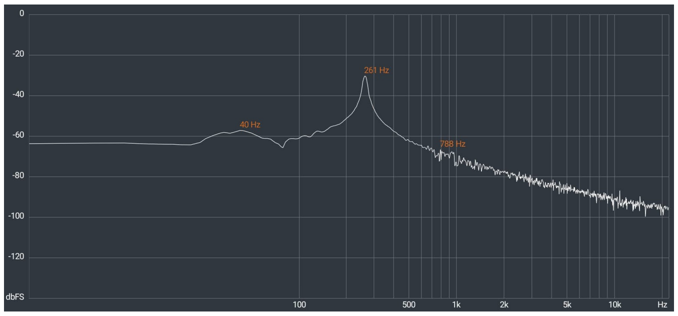
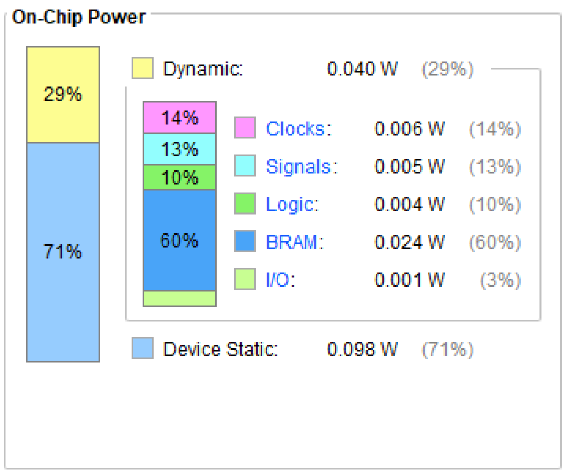


Fig. 11: Middle C sine wave frequency spectrum analysis.

D. *Power Consumption*

The power usage of such a relatively simple (computationally speaking) set of process should generally not be too much of a concern. However, it is still an important parameter to discuss. If, for example, future product developments might want to be made portable, the power consumption would then become a critical factor. Figure 12 shows a summary of the on-chip power usage of each section for the FPGA solution.



On-Chip Power

🟨 Dynamic:	0.040 W	(29%)
🟪 Clocks:	0.006 W	(14%)
🟦 Signals:	0.005 W	(13%)
🟩 Logic:	0.004 W	(10%)
🟦 BRAM:	0.024 W	(60%)
🟩 I/O:	0.001 W	(3%)
🟦 Device Static:	0.098 W	(71%)

Fig. 12: Graphic showing the FPGA power usage characteristics.

As might be expected, the static power usage dominates the overall consumption: $P_{total} = 0.138W$. Block-RAM takes up most of the dynamic power; this is assumed to be a result of the sinewave sample storage required by the system.

Notably, the I/O power usage (buttons and speaker output) remains relatively low. This implies that the addition of a full keyboard set of buttons and potentially more speakers would lead to minimal extra power costs.

For comparison, some simple current and voltage measurements were taken on the Teensy 3.6 while it was running the DDS algorithm, at the recommended 180MHz clock frequency (i.e. not overclocked). At maximum capacity with four independent voices, the average current drawn was around 87mA, at a voltage of around 5.01V. Thus, the average power consumption for the golden measure solution was around $0.435W$.

Interestingly, based on these preliminary measurements, the power consumption of the microcontroller board was more than three times greater than that of the FPGA. However, more detailed tests and measurements would need to be done in order to make reliable conclusions about this.

E. *Resource Usage*

Of the total 63400 LUTs available in the Artix A7 FPGA, only 732 (1.15%) were used in the the final design of this system. The `function_generator` module is understandably the main culprit for this utilization factor, due to its heavy calculation requirements. This low portion of resource usage suggests that the system might be substantially expanded in future developments.

F. *ADSR Envelope*

The ADSR envelope functionality was tested using a simple testbench module, where an input button's state was 'pressed' twice for different lengths of time. A square wave input signal was used for simulation, and arbitrary ADSR parameters were set. The results of this test are shown in figure 13.

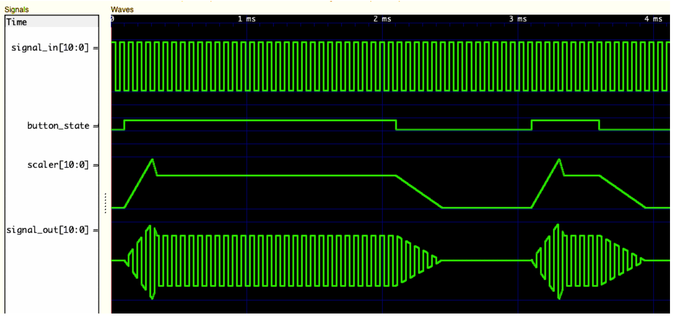


Fig. 13: Sample testbench output for ADSR envelope

Notice that the `scaler` parameter is synchronized with the `button_state`: when the button goes high, the attack section begins, once the scaler reaches max value the decay section begins, and eventually the scaler settles at the sustain value; when the button is released, the scaler is slowly released to zero. Importantly, the actual signal output follows the *envelope* of the scaler signal, but is centered around 1024 (as this represents an output value of 'zero').

Through this simple simulation, together with an audible verification test actually running on the FPGA board, it was confirmed that the ADSR functionality was working. Unfortunately, implementing this envelope feature on the Teensy board would require much more development, and even then, may not be reliable. It was thus not included for comparison in this test.

G. *Cost Analysis*

The rough pricing for the two implementations is presented in Table I.

The difference is significant, however, it is difficult to tell at this stage whether or not the advantages of the FPGA warrant its greater expense for this system. The decision ultimately

	Full package	Just the chip
Nexys A-7	R4616	R2187
Teensy 3.6	R642	R306

TABLE I: Costs of the FPGA board compared to the golden measure solution

lies with the end-product manufacturer. It may be the case that the DDS system detailed in this report forms part of a massive line of infrastructure, consisting of tens of thousands of copies. In such a scenario (particularly if the advantages discussed are valuable for the purpose in question), it might be worth constructing an ASIC implementation, which would further lower the cost.

The Nexys A-7 is also a very feature-rich board, largely unnecessary for this project. There are cheaper boards on the market. And there is also the option of investing in a more customised approach, using just the chip—which comes at around half the price.

VIII. Conclusion

This investigation comes to a close on a tunefully successful note. The final design is able to run as a stand-alone system, consisting of the Nexys-A7 FPGA board, a speaker and peripheral “keyboard” buttons. There are 12 external buttons, each was tied to a note within a full (pre-assigned) octave scale. On-board buttons may be used to shift up and down by degrees of one octave scale at a time. Remaining on-board buttons were left to switch between a variety of four waveforms: square, sinusoid, sawtooth and triangle.

A core objective outlined in the at the onset of this report was to achieve polyphonic audio synthesis. As shown in section VII-B of the results, it was proven possible to play as many as twelve notes simultaneously.

The board’s built-in seven-segment display was used to show the current waveform and octave selections, each as 2-bit binary codes. This resembled, to some extent, a live display panel. In future expansions this may be extended to include the frequency/s being played.

Another fundamental aim of the project was to compare the FPGA’s performance with that of a sequential microprocessor. The Teensy 3.6 was used as this *golden measure* for a sequential implementation. The basic operation was kept as similar as possible between the two platforms, i.e. each was made to directly compute sample values on-the-fly. Naturally, the Teensy implementation might have taken advantage of optimized audio synthesis libraries, but this would not have been a fair comparison.

The extent of polyphony capability (number of simultaneous channels) was a major draw-card for the parallel FPGA implementation. The Teensy was limited to only four simultaneous tones, while the FPGA was proven to easily cope with twelve (and should, in theory be able to handle far more).

As discussed in Section VI-B2, the Teensy is limited to a maximum output frequency of 6424.8Hz. Furthermore, it suffers major losses in accuracy as the frequency demand increases. The Nexys A-7, on the other hand, reaches its theoretical maximum at an impressive 390.6kHz. It too suffers from discretization errors; however, these would be imperceptible to the human ear.

Cost was was an important factor to consider. Particularly because of the great disparity in pricing between the Nexys A-7 and Teensy 3.6. The Teensy comes off the shelf at almost one tenth the price of the the FPGA. Even so, a fair comparison is difficult, not knowing what the ultimate use case might be for this system.

All-in-all, through the ‘Revenge of the Synth’ investigation, the advantages of the FPGA’s parallelism were made obviously apparent when compared to a conventional microprocessor’s sequential implementation. The results are satisfying to behold and one can all but become excited for the future of parallelism and direct digital synthesis. There is certainly scope for future investigation, building upon the research done here.

Appendix

Please find all Verilog code used for this project here: https://github.com/wylieza/revenge_of_the_synth

References

[1] J. Tierney, C. Rader, and B. Gold, “A digital frequency synthesizer,” *IEEE Transactions on Audio and Electroacoustics*, vol. 19, no. 1, pp. 48–57, 1971.

[2] G. Lacey, G. W. Taylor, and S. Areibi, “Deep learning on fpgas: Past, present, and future,” *arXiv preprint arXiv:1602.04283*, 2016.

[3] E. B. Fernandez, W. A. Najjar, S. Lonardi, and J. Villarreal, “Multithreaded fpga acceleration of dna sequence mapping,” in *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–6.

[4] K. Bhagat, “Tutorial on designing and implementing a direct digital synthesizer (dds) on a field programmable gate array (fpga),” Master’s thesis, University of Illinois at Urbana-Champaign, 2012.

[5] L. Cordesses, “Direct digital synthesis: A tool for periodic wave generation,” *IEEE SIGNAL PROCESSING MAGAZINE*, 2004.

[6] Leitner, Stefan, Wang, Haibo and Tragoudas, Spyros. ”Design Techniques for Direct Digital Synthesis Circuits with ImprovedFrequency Accuracy over Wide Frequency Ranges.” Journal of Circuits, Systems and Computers 26, No. 2 (Feb 2017). doi:10.1142/S0218126617500359.

[7] L. Cordesses, “Direct digital synthesis: A tool for periodic wave generation (part 1),” *IEEE Signal processing magazine*, vol. 21, no. 4, pp. 50–54, 2004.

[8] K. A. Davis, “Computing sin cos in hardware with synthesisable verilog.” [Online]. Available: https://kierdavis.com/cordic.html

[9] P. Stoffregen, “Intervaltimer tuning.” [Online]. Available: https://forum.pjrc.com/threads/33089-IntervalTimer-tuning?p=96627&viewfull=1#post96627