The Fourier Awakens

Direct Digital Synthesis

Robbie Katz[†] Benji Lewis[‡] and Cairin Michie[§]
Group 9
EEE4084F Class of 2016
University of Cape Town
South Africa
[†]KTZROB006 [‡]LWSBEN002 [§]MCHCAI001

***Abstract*—This paper details the conceptual and prototype design of a function generator using the Fourier Series estimation through the use of Direct Digital Synthesis techniques. Various architectures are discussed to determine what is optimal for this design.**

I. Introduction

Direct Digital Synthesis (DDS) is a method of producing periodic analog signals using digital techniques [1]. In this project, DDS will be used to generate the Fourier Series expansion of a user defined waveform.

DDS allows for periodic signals to be synthesized by constantly referencing a 'template' waveform, which is stored in memory [2]. The frequency of the new waveform is determined by how often values are read from memory. The phase is determined by the point at which reading from memory starts. The analog signal is produced by outputting the digital waveform through a Digital-to-Analog converter (DAC).

DDS was first proposed in 1971 by Tierney, et. al [3]. This technique is commonly used in signal synthesis, frequency hopping, medical imaging systems and radio receivers [4]. DDS is beneficial in its ability to generate signals with extreme frequency precision and its ability to change frequency and phase rapidly [5].

The Fourier Series allows for periodic signals to be represented as a sum of sinusoids [6]. Equation 1 [7] shows how this is implemented for an arbitrary waveform $f(t)$, with a period of $2L$.

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left(a_n\cos(\frac{\pi n t}{L}) + b_n\sin(\frac{\pi n t}{L})\right) \quad (1)$$

The individual sinusoids are either cosine or sine, and are characterized by their coefficients, a_n and b_n, and their natural frequency. These individual sinusoids can be paired in terms of their frequency, and can both be generated by a DDS module that is sent this frequency as a control word. This represents a single iteration of the summation in Equation 1.

This project will be implemented on the Digilent Nexys4™ [8]. FPGA-based acceleration is well suited to this application as it allows any number of DDS modules to be implemented in parallel. The number of DDS modules used will be determined by the number of coefficients used in the summation, which in-turn determines the resolution of the final waveform.

A. DDS Overview

Figure 1 shows the block diagram of a DDS module. The single-cycle 'template' waveform is stored in the Phase Register, and is addressed according to the value in the Phase Accumulator. The address of the next sample to be extracted is determined by the sum of the Phase Accumulator and the value of the frequency word, M.

The frequency word is used to set the frequency of the desired output waveform, f_o, and is based on the clock frequency, f_c, and the number of bits in the address space, n. Equation 2 [9] shows the relationship between M and the output frequency.

$$f_o = \frac{M \cdot f_c}{2^n} \quad (2)$$

The size of the address space, n, describes the resolution of the 'template' waveform. On each clock cycle the Phase Accumulator is updated. In the case of $M = 1$, the Phase Accumulator will take 2^n clock cycles before it overflows and restarts the cycle. When $M = 2$, the Phase Accumulator will take twice as fast to 'roll over', thus doubling the frequency of the output waveform.

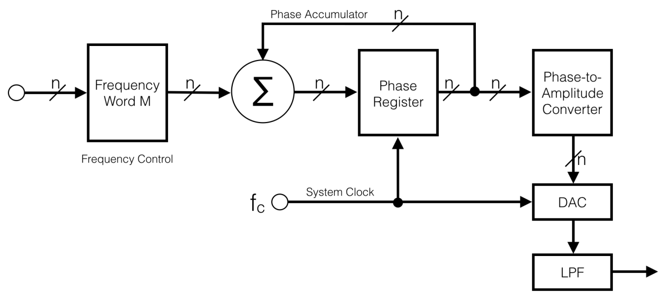


Fig. 1: DDS Block Diagram [9]

II. Overview

A. Implementation

Figure 2 provides an overview of the various modules that make up the complete system. An overview of how these modules will be implemented is detailed below.

The Fourier coefficients and frequencies corresponding to the users waveform will be calculated in Matlab. The Fourier coefficients will be stored in RAM. The user defined parameters will be passed into the system via a UART port as serial data.

A high resolution sine wave will be stored in ROM. The wave will have a resolution of 24 bits for the amplitude and 32 bits for the period. This will allow for the output waveform to have a maximum period of 2^{32} data points. This results in an output frequency resolution of 0.023 Hz. The DDS modules will step through the stored sine wave according to the tuning word, M. As M is increased, the synthesized waveform's frequency increases. Each DDS module will receive a multiple of M, allowing for Fourier series estimation to take place.

The DDS modules will shift the sine wave by $\frac{\pi}{2}$ to form a cosine when this is required in the Fourier calculation. Each DDS module will synthesize a single pair of sine and cosine functions based on their frequency by extracting the point from the look-up table, after which these waveforms will be amplified according to their corresponding coefficients. The outputs of the individual DDS modules will then be summed to produce final output waveform . This waveform will pass through the DAC so that it can be represented by the interface.

The design will implement 45 000 DDS modules. This number was decided on so that the output would safely span through the human audible frequency range of 20 Hz to 20 kHz [10], and satisfy the Nyquist sampling criterion. The 45 000 DDS modules give sufficient bandwidth for the human audible range but is not restricted to any specific frequency range.

The design will be implemented on an ASIC system, to take advantage of the parallelism of the design.

B. Interface

The user defines a waveform they wish to be synthesized on the keyboard interface as seen in the system overview in Figure 2. The user has control over certain parameters of the waveform. Namely, the type of waveform, amplitude, frequency, and the number of DDS modules (Fourier coefficient pairs) used in generating the output. These parameters will need to be within reasonable constraints, as restricted by the implementation. These constraints will be enforced through the keyboard interface.

The synthesized waveform will then be output on the built in oscilloscope, and over the audio jack. This was done so that the waveform can be observed both visually and audibly.

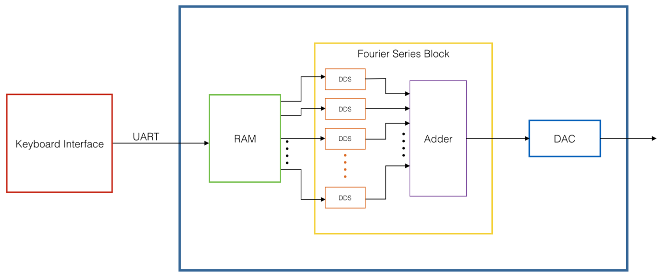


Fig. 2: General System Block Diagram

III. Detailed Design

In this section, detailed descriptions of each of the modules will be elaborated upon.

A. UART

To receive commands from a keyboard, UART communication is implemented. The UART communication protocol is used to send data specifying the Fourier coefficients and required tuning word to the DDS modules. The baud rate of the UART communication effects how fast the design can change between waveforms. With consideration to this fact, a 100 Bd baud rate is implemented. The protocol implemented uses the standard UART protocol for receiving data.

B. ROM

For the design it was decided that using ROM instead of RAM would be sufficient. This is because the values in the look-up table are never changed. The ROM, as discussed for the DDS module, needs to be dual access. This is so that a DDS module can access a sine and cosine wave simultaneously. The waveform stored in the look-up table is a sine wave. To use the look-up table for cosine waves, a $\frac{\pi}{2}$ shift must be applied. The number of values stored in the ROM is 2^{32} with each value having 24 bits. This means that the amplitude resolution of the waveform is 24 bits, to ensure a realistic output for the DAC is obtained, and the resolution of the period is 32 bits.

C. DDS

The classic DDS implementation will be used with a slight twist. The actual DDS module will be implemented using the traditional look up table method, however each DDS module will produce two waveforms simultaneously. A sine and cosine waveform will be produced with the same frequency but different amplitudes in each DDS module. As shown in Equation 1, when approximating a waveform, a sine and cosine waveform of each natural frequency is needed. To be more efficient with the available resources it is logical to generate both the sine and cosine waveform in each DDS module. This is done for two reasons, firstly the waveforms need to be generated at the same frequency and secondly the ROM blocks can be dual access, meaning the number of look-up tables needed can be limited.

With consideration to the generation of the two waveforms, there were two options available. The first was to use a look-up table. This is a very fast method which requires only one fetch cycle to retrieve values, however it requires vast amounts of spacial resources to implement. The second option was to calculate waveform values on the fly. This uses no spacial resources, however, this requires more clock cycles to calculate. To calculate the waveform values on the fly the calculations laid out in Equation 3 [11] would have to be implemented.

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \dots \tag{3}$$

There are techniques that can be used to make the calculation faster, such as factoring equation 3 [11] to ensure it is predominantly an unsigned summation. But using this technique will always limit performance. Specifically the number of clock cycles required to calculate a waveform value will limit the maximum frequency that can be output.

Considering all the details specified above, the final DDS module was designed as shown in Figure 3.

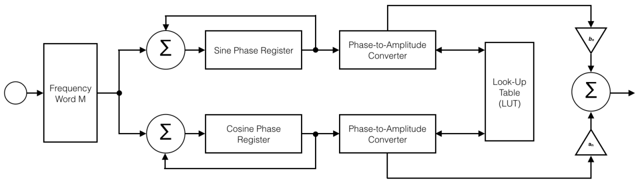


Fig. 3: DDS module

The module receives three values, the tuning word M and the amplitude words a_n and b_n. The tuning word specifies the frequency of the sine and cosine waves generated as shown in Equation 2.

By defining a clock speed and a period length of 2^n bits long, the smallest frequency that can be output is defined. By increasing M, frequency is increased. The amplitude words a_n and b_n define how the sine and cosine amplitude values are scaled, these values correspond to those specified in Equation 1. The sine phase register stores the current phase point that needs to be output. The cosine phase register acts identically to the sine phase register but has a $\frac{\pi}{2}$ phase shift. The phase to amplitude converters check the look-up table for the corresponding value, and output a scaled version corresponding to the coefficients. Finally, the two values are added together, to be used in producing the final output waveform.

D. *Adder and DAC*

The Adder takes all the output values from the DDS modules, adds them together and outputs the final value. Finally, the value on the output of the Adder is passed into the DAC. The DAC creates an analogue signal for output.

IV. Methodology

A. *Proof-of-Concept Design*

A prototype of our system was designed and implemented in order to prove the theoretical operations of our conceptual design.

Figure 4 shows a block diagram of this prototype, which differs from the proposed conceptual design due to the following constraints.

1) Architectural Implementation: The prototype was designed and implemented on the Nexys4™ [8] FPGA as opposed to the proposed ASIC implementation. The key purpose for this decision is that prototyping on an ASIC system is known to be more expensive in terms of time, as well as money, when compared to FPGAs [12].

As a result of using an FPGA, the design had to be limited in terms of spatial resources so that it could fit on the FPGA.

2) DDS Modules: Due to the spatial limitations of the FPGA, instead of implementing 45 000 DDS modules, the prototype only makes use of 32. Whilst this will not provide the frequency resolution desired, it is sufficient to prove the concept of our DDS implementation.

3) User Interface: The conceptual design proposes that the user will be able to interact with our system using a custom-built interface that communicates via UART. For practical reasons, the prototype interface comprises of a PC connection over UART, along with a basic 'menu' implemented using the Nexys4™'s [8] four push buttons.

The PC connection is used to allow the frequency of the output waveform to be changed through the computers keyboard, thus acting much like a 'musical' keyboard.

The 'menu' is used to allow various properties of the system to be tested. The properties that were tested are:

- **Waveform:** Change between the six waveforms implemented (Sine, Square, Sawtooth, Triangle, Guitar and Clarinet).
- **Modules:** Change the number of DDS modules that are used to generate the respective waveforms. This could be anything in the range between 1 and 32.
- **Amplitude:** Control the amplitude of the output signal (effectively the volume).
- **Frequency:** Change the octave of the output by doubling or halving the output signal's frequency.

4) Audio Output: Another key difference of the prototype is the use of PWM and a Low Pass Filter (LPF), rather than using a DAC. The reason for using PWM is because the Nexys4™ [8] does not have any DAC output pins.

The PWM module interprets the value output by the adder module as a duty cycle. The duty cycle is defined by Equation 4.

$$DutyCycle = \frac{AdderOutput}{2^{32}} \tag{4}$$

The PWM module outputs a PWM signal with the calculated duty cycle through a filter that converts PWM to an analogue output.

This is significant due to the effects of the LPF. The Nexys4™ [8] uses a Butterworth filter, the bode plot found in the reference manual [13], which starts filtering out frequencies at approximately 10 kHz. This results in an upper bound for the output frequency at 10 kHz. The filter will affect all frequency components with frequencies higher than 10 kHz.

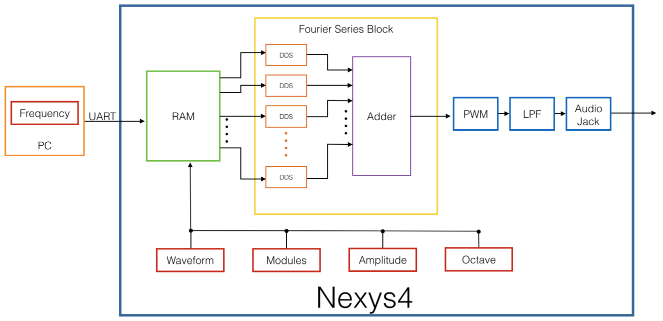


Fig. 4: Prototype Block Diagram

```
for i=0:4095
    x(i+1) = floor(511*sin(i*2*pi/4096)+512);
    fprintf( f,'%d%c\n',x(i+1),',');
end
```

Listing 1. Sine Wave Matlab code

```
module PWM(
        input CLK,
        input [10:0]PWM,
        output reg AUD_out //output to mono jack
    );
        reg [10:0]PWMcount;
        always @(posedge CLK)begin
                PWMcount<=PWMcount + 1'b1;
                if(PWM<PWMcount)
                        AUD_out<=1'b1;
                else
                        AUD_out<='b0;
        end
endmodule
```

Listing 2. PWM module

B. Implementation

The prototype was implemented using several Verilog modules. In the beginning it was recognized that being able to receive feedback from the FPGA was very important, so the first modules that were implemented were such that the output could be observed from the mono audio jack on the Nexys4™ [8]. This meant that the first modules that needed implementing were the Dual Port ROM as well as the PWM module.

1) Look-up Table: To implement the look-up table as dual port ROM, a waveform was needed. To do this a Matlab script was used to produce a single period sine waveform with the appropriate amplitude. In the prototype, DDS modules were implemented that contained sine wave templates with an address space of 12 bits and an amplitude resolution of 10 bits. To produce this sine wave the code shown in Listing 1 was used.

The code in Listing 1 produces a sine wave with 4096 values and an amplitude of 1024. However, care was taken to shift the sine wave up by 512 so that it would only contain positive values.

2) PWM Module: A basic PWM module was implemented, as shown in Listing 2. It receives a value that represents the phase value of the waveform required at the output. The module assumes a maximum output of 2^{11}, this is because in the final prototype different sine waveforms are added together resulting in values that could be above the 2^{10} value. The code counts up to 2^{11}. When the count is below the input value it outputs high, when it is above it outputs low. The module runs at the full clock speed, 100 MHz, with the Butterworth filter at the output converting this into the required analog signal.

With both these modules implemented it was easy to confirm the progress made.

3) DDS Module: The DDS module is an implementation of the DDS block diagram shown in Figure 3. As discussed before, each DDS module accesses the ROM block that contains the template sine waveform. Two addresses are stored, one for the sine and the other for the cosine. On each clock cycle these addresses are shifted by M, the frequency word, and the data received is added together and sent to the output. Importantly for this module, the clock frequency was slowed down significantly. The reason for this is that the output frequency would be too high to be audible, and would be filtered out by the LPF. When considering Equation 2, it is clear with a clock frequency of 100 MHz and an address space of 12 bits, the minimum output frequency is about 25 kHz. To counter this, the output frequency was reduced by 2^7. This meant that the minimum frequency that could be output is 191 Hz. This is good enough for the prototype to test the functionality, but for the conceptual implementation higher resolution is required.

To implement the coefficients a_n and b_n the value 131072 was defined to be equal to one. This was done so that only whole numbers needed to be dealt with in the implementation. When scaling waveforms, each was multiplied by their coefficients and then divided by the value 131072. The number was chosen because it is a power of two and division by powers of two amount to shifting when synthesized. Shifting is a much easier implementation than division and will run faster and use less space on the FPGA.

To implement 32 DDS modules, a generate loop was used to automatically generate the code. This also enabled varying the number of DDS modules generated.

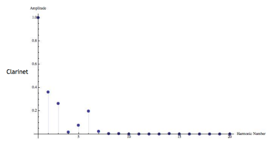
4) Waveforms: To implement the various waveforms, a Matlab script was written to determine the Fourier coefficients for the Square, Triangle and Sawtooth waves. The code for the square wave is shown in Listing 3. This code was adapted to generate the Triangle and Sawtooth waves.

To generate musical instrument sounds, spectrum analyses for two instruments, Guitar and Clarinet, was used. The instruments were generated using only sine waves, this is because the amplitude of the frequency components were important but the phase shift is not. Humans cannot discern phase shifts with audio so it is not so important to implement. Figure 5a and Figure 5b were used as guidelines for the spectrums of each waveform.

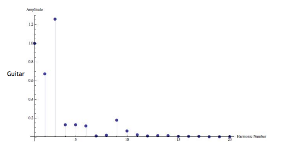
5) UART: The UART code was generated referencing a YouTube video [15]. The Ubuntu terminal was used to send UART commands to the FPGA. These commands specified the value of the frequency word M. This meant that it could be used as a rudimentary musical keyboard.

```
%==============Square Wave==============
 a0=0;
 for i=1:n
     sq = @(x) (Amp*square(x)).*sin(i*x);
     b(i)=(1/pi)*(integral(sq, -pi,pi));
end
```

Listing 3. Square Wave Fourier coefficients Matlab code

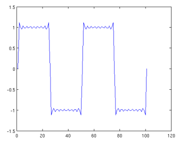


(a) Clarinet Spectrum

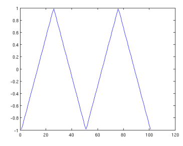


(b) Guitar Spectrum

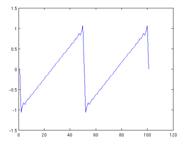
Fig. 5: Instrument Spectrums [14]



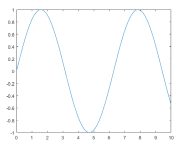
(a) Square Wave



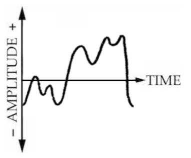
(b) Triangle Wave



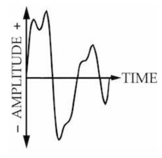
(c) Sawtooth Wave



(d) Sine Wave



(e) Clarinet Wave [16]



(f) Guitar Wave [17]

Fig. 6: Expected Output Waveforms

6) Buttons: The final module that was created was a button module. This module checked for button presses and changed various aspects of the output waveform according to the 'menu' discussed in section IV-A3. The property that the up/down buttons control is shown on the seven segment display.

C. Experimentation

In order to ensure that the prototype operates in the correct manner, a number of experiments were performed. This was done by observing the output produced by the system using an oscilloscope and comparing these to the expected results. The aspects of our system that were tested are further discussed below.

1) Output Waveforms: The key purpose of this project is to produce accurate waveforms (according to the Fourier Series) based on the various input parameters. In order to ensure the output waveforms are exactly as expected, six test waveforms were used, namely: Square, Sawtooth, Triangle, Sine, Clarinet and Guitar.

The expected results were calculated using a MatLab script, part of which is seen in Listing 3, written to produce the Fourier Series approximation of the waveforms using exactly 32 iterations of the Fourier Series, as seen in Equation 1. This will be the number of DDS modules being implemented during the confirmation of the waveforms.

Figure 6 shows the expected waveforms to be seen during experimentation. It should be noted that the clarinet waveform in Figure 6e and the guitar waveform in Figure 6f were not generated in Matlab due to their unique waveforms. Thus the expected results show the exact waveform and not the Fourier approximation. In these cases, the expected results should strongly resemble these waveforms.

2) Number of DDS Modules: In order to investigate the effect of varying the number of DDS modules, the outputs produced when changing the number of DDS modules needed to be examined.

The number of DDS modules used in generating the output waveform is the number of iterations of Equation 1 used to approximate the desired waveform. Thus the accuracy of the output waveform is proportional to the number of modules.

It is expected that when the number of DDS modules used is increased, the output waveform will increasingly approximate the intended waveform, ultimately approaching those seen in Figure 6.

3) Limitations: The boundary cases of the prototype needed to be experimented in order to determine the limitations of the design. The following list details the limitations investigated:

- **Frequency:** The maximum frequency that can be output. It is hypothesized that this will be 10 kHz, in accordance to the cut-off frequency of the on-board LPF. In order to test this, the frequency of the output waveform will be doubled until irregularities are observed.
- **Number of Modules:** The maximum number of DDS modules that can be implemented on the FPGA and if this number can be expanded if no spatial limitations exist. This will be tested by observing the amount of space utilized on the FPGA.

V. Results

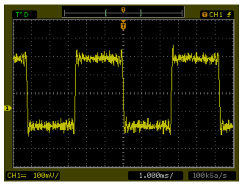
A. Output Waveforms

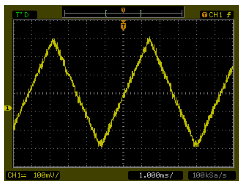
Figure 7 shows the output waveforms observed when testing the six waveforms, with expected waveforms in Figure 6, using 32 DDS modules. It is seen that the waveforms produced correspond to the expected results, with slight irregularities in the case of the Guitar in Figure 7f. This is due to the detailed harmonic representation of this waveform, which led to the loss of certain harmonics in the prototype implementation.

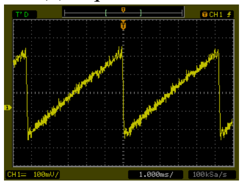
B. Number of DDS Modules

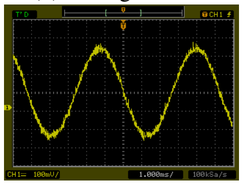
Figure 8 shows the output waveforms observed when varying the number of DDS modules used to estimate the square wave in Figure 6a. It can be seen in Figure 8 that, as the number of modules is increased, the output waveform gets increasingly similar to the expected result.

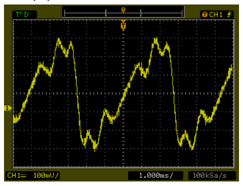
As the number of DDS modules was increased, it was seen that no computational expense was incurred. Thus, the number of modules can be increased as specified in the conceptual design. In the case of using more DDS modules, the output waveform will be a more precise approximation of the desired one.


(a) Square Wave


(b) Triangle Wave


(c) Sawtooth Wave


(d) Sine Wave


(e) Clarinet Wave

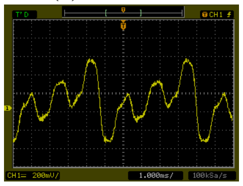
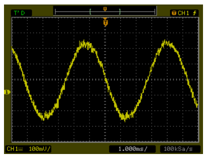
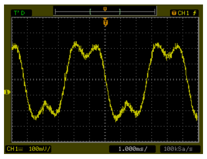
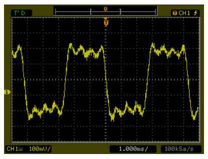
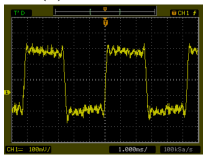

(f) Guitar Wave

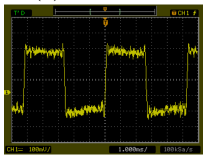
Fig. 7: Output Waveforms


(a) 1 Module


(b) 4 Modules


(c) 8 Modules


(d) 16 Modules


(e) 24 Modules

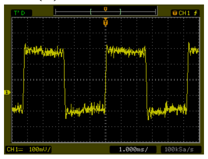

(f) 32 Modules

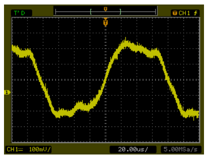
Fig. 8: Results for Varying Number of DDS Modules

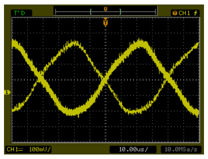
C. Prototype Limitations

1) Frequency: Figure 9 shows the result of increasing the output signals frequency, and the effect of the LPF, in the case of a square wave being the desired output.

At around 6 250 Hz the effects become visible, with the entire signal being cut-out by the time it reaches 50 000 Hz. In the case of 12 500 Hz it can be seen that the high frequency components of the signal get cut-out, thus losing the desired output waveforms shape. As expected, this is a result of the Nexys4™'s [8] on-board Butterworth filter.

2) Spatial Resources: When observing the maximum number of DDS modules that could be implemented on the FPGA, it was seen that 32 modules occupied almost all of the available resources on the Nexys4™ [8]. This proves that the functionality of our implementation is limited to the architecture that it is implemented on.


(a) 6 250 Hz


(b) 12 500 Hz

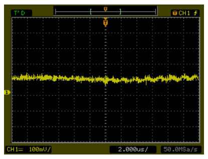

(c) 50 000 Hz

Fig. 9: Effect of LPF with Square Wave input

VI. Conclusion

In conclusion, it has been successfully shown that the proposed conceptual design is implementable. The prototype of the design was implemented with success. It was shown that implementing multiple DDS modules to implement the Fourier Series Expansion is realizable and, importantly, expandable. With the addition of more DDS modules there is a negligible effect on processing time for the system.

Whilst an FPGA implementation is useful for prototyping, it is clear that an ASIC design is essential to realize the conceptual design due to limited spatial resources on FPGAs.

References

[1] M. Christiano, "Everything You Need to Know About Direct Digital Synthesis," http://www.allaboutcircuits.com/technical-articles/direct-digital-synthesis/, November 2015.

[2] N. Instruments, "DDS Waveform Generation Reference Design for LabVIEW FPGA," http://www.ni.com/example/31066/en/, January 2016.

[3] B. G. Joseph Tierney, Charles M. Rader, "A Digital Frequency Synthesizer," *IEEE Transactions on Audio and Electroacoustics*, vol. 19, no. 1, pp. 48 – 57, March 1971.

[4] Lancaster Hunt, "Direct Digital Synthesis (DDS) For Idiots (like me)," http://lancasterhunt.co.uk/direct-digital-synthesis-dds-for-idiots-like-me/, November 2009.

[5] N. Instruments, "Understanding Direct Digital Synthesis (DDS)," http://www.ni.com/white-paper/5516/en/, May 2015.

[6] E. W. Weisstein, "Fourier Series," http://mathworld.wolfram.com/FourierSeries.html.

[7] "Fourier Series," https://www3.nd.edu/nancy/Math30650/Matlab/Demos/fourier_series/fourier_series.html.

[8] Digilent, "Nexys4 FPGA Board Reference Manual," http://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPNexys4/documenatation/Nexys4_RM_VB1_Final_3.pdf, September 2013.

[9] A. Devices, "Fundamentals of Direct Digital Synthesis (DDS)," http://www.analog.com/media/en/training-seminars/tutorials/MT-085.pdf, 2009.

[10] S. W. Smith, "Human Hearing," http://www.dspguide.com/ch22/1.htm.

[11] Dotan Cohen, "Taylor series for sine or sinus," http://dotancohen.com/eng/taylor-sine.php/.

[12] Xilinx, "FPGA Vs. ASIC," http://www.xilinx.com/fpga/asic.htm.

[13] Digilent, "Nexys4 FPGA Board Reference Manual," p. 28, November 2013.

[14] Maria Bell, "Fourier Analysis in Music," https://www.projectrhea.org/rhea/index.php/Fourier_analysis_in_Music.

[15] Toni T800, "FPGA Tutorial 3. UART in VHDL on Altera DE1 Board," https://www.youtube.com/watch?v=fMmcSpgOtJ4&ab_channel=ToniT800, August 2013.

[16] John L. Odhne, "The Correspondences of Musical Instruments," http://highermeaning.org/Authors/JLO/Music.shtml.

[17] Jon Chappel, "What is Sound?" http://www.psneurope.com/what-is-sound/, March 2015.