JAIST Repository
https://dspace.jaist.ac.jp/

Title	Energy-Efficient Real-Time Pitch Correction System via FPGA
Author(s)	唐, 博文
Citation	
Issue Date	2023-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/18742
Rights	
Description	Supervisor:田中 清史, 先端科学技術研究科, 修士(情報科学)

JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Master's Thesis

Energy-Efficient Real-Time Pitch Correction System via FPGA

TANG, Bowen

Supervisor TANAKA, Kiyofumi, Ph.D.

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

August 2023

Abstract

The pitch, or in technical terms, the fundamental frequency (the lowest oscillating prominent frequency component of a periodic signal) of a signal, determines how "high" or "low" an audio signal would sound. One can change the pitch of an audio recording by shifting the fundamental frequency of the audio signal, to make sure that the final audio will always stay in tune even if the music performer makes mistakes during the recording process. Such a manner is often referred to as "pitch correction" as a post-processing technique in audio engineering.

Pitch correction, in an offline context, will not be a too challenging task. With the development of computing power, audio analysis and modification seem to become more and more trivial. However, what if one wants to perform pitch correction in real time?

A real-time environment sets harsh timing constraints on the computation. If an algorithm could not be executed efficiently enough, the timing requirement would then not be met. What's more, a more limited computation time usually calls for a more powerful computing system. Indeed, computing power is becoming more and more abundant as the technology evolves. Yet, in certain circumstances where a powerful computing system cannot be deployed, an energy-efficient solution becomes vital.

Back to the previous question. If one needs to perform pitch correction, a pitch detection operation needs to be performed first, followed by a pitch shift operation. The pitch detection part tells us where is the current pitch and how far it is away from the designated target, and the pitch shift part modifies the pitch to the desired level. There exist numerous studies with different approaches aimed to perform the prior two tasks. Nonetheless, little research has been conducted to look for an energy-efficient real-time solution that solves both questions.

Thus, this thesis study aims to develop a standalone power-efficient system that can perform pitch detection and pitch shift operation simultaneously in real-time. The processing needs to be fast enough to catch up with the audio sampling frequency, meanwhile utilizing as few hardware resources as possible. In this thesis study, we proposed and developed two novel mechanisms, one being the auto-cross correlation that performs pitch detection, and another being a ring buffer, with jump-based-on-pitch pointers, that performs pitch shift operation. We then verified our approaches via experiments, making sure that they could yield an as-expected result.

After the verification process, we implemented our design on an FPGA platform. After the implementation, we recorded the output generated by our new design and verified that the new design is able to satisfy the requirements set by this study.
Keywords: Pitch detection, pitch shift, power-efficient, real-time, FPGA

Acknowledgment

I would like to express my sincerest gratitude to:

- Dr. Tanaka for creating a thriving and relaxing research environment, as well as all the support and advice along my way.

- Dr. Asano for taking the time to guide me through my minor research.

- Chen Yan for helping me get familiar with Nomi-city, all the technical support, and the great anecdotes shared.

- Wang Chiye for all the great times spent in the BBQ restaurant and Karaoke together.

- Liu Runyu for patiently teaching me about robotics.

- Zhou Yi for cheering and backing me up whenever I feel down.

- My mum and dad for everything you do.

Thank you all.

List of Abbreviations

ACF	Autocorrelation function
ADC	Analog to digital converter
ATC	Auto-cross correlation function
BRAM	Block random-access memory
DAC	Digital to analog converter
DFT	Discrete Fourier transform
DSP	Digital signal processing
FD	Frequency domain
FFT	Fast (discrete) Fourier transform
FIFO	First-in-first-out
FLOP	Floating-point operation
FPGA	Field programmable gate array
FSM	Finite-state machine
LSB	Least significant bit
LUT	Look-up table
MSB	Most significant bit
SDF	Square difference function
SOLA	Synchronous overlap and add
STFT	Short-time Fourier transform
TD	Time domain

List of Symbols

$x(t)$	a continuous signal x
$x[t]$	a discrete signal x
$\mathrm{ATC}_x[\tau]$	auto-cross correlation function of a discrete signal $x[t]$
$R_x[\tau]$	autocorrelation function of a discrete signal $x[t]$
$\log x$	base 2 logarithm value of x
$\mathscr{F}\{x(t)\}$	Fourier transform of a signal $x(t)$
ω	frequency index of a signal
ω_0	fundamental frequency of a signal
$\mathrm{SDF}_x[\tau]$	squared difference function of a discrete signal $x[t]$
t	time index of a signal
$O(\bullet)$	time complexity of a function, where $\bullet$ states the complexity

List of Figures

2.1 A discrete signal $x[t]$ and its DFT $X[\omega]$ 4
2.2 A piece of vocal signal and its DFT 5
2.3 A piece of viola signal and its DFT 6
2.4 DFT of the viola recording in logarithmic scale 6
2.5 Signal $x[t]$ and its ACF and SDF 9
2.6 Different-size DFTs with different frequency resolution 11
2.7 ACF and SDF of the viola signal 12
2.8 ACF and SDF fail to provide any meaningful information . . . 13
2.9 ACF and SDF of an adjusted $x[t]$ 14
2.10 ATC result with a better estimation 17
2.11 ATC of the viola signal 17
2.12 A sine wave and a square wave with the same frequency and amplitude 18
2.13 One-bit ATC of a viola signal 19
2.14 One-bit ATC results, with and without noise 20
2.15 More one-bit ATC results, with and without noise 21
2.16 Normalized square difference function struggling to estimate the fundamental frequency 22

3.1 DFTs of a piano signal and its SOLA stretched signal 30
3.2 Stretching an array using nearest the neighbor interpolation method 30
3.3 Comparison between two interpolation methods 31
3.4 Further comparison between two interpolation methods 31
3.5 Compromised continuity by nearest-neighbor interpolation . . 32
3.6 Appropriately jumping the output pointer 35
3.7 DFT analysis on the stretched signal 35

4.1 Top-level schematics of the proposed system 38
4.2 Schematics of the pitch detection module 39
4.3 An overview of the pitch shift module 42
4.4 Real-time pitch correction targeting at note D 45
4.5 Real-time pitch correction targeting at note C 46

4.6 Real-time pitch correction targeting at note E 46
4.7 Real-time pitch correction targeting at note F# 47
4.8 Real-time pitch shift targeting at one octave higher 47

List of Tables

4.1 Hardware Resource – Component Level 43
4.2 Hardware Resource – Top Level 43
4.3 Power Consumption of Our Design 44

Contents

Abstract **I**

Acknowledgment **III**

List of Abbreviations **V**

List of Symbols **VII**

List of Figures **IX**

List of Tables **XI**

Contents **XIII**

Chapter 1 Introduction **1**

Chapter 2 Pitch Detection **3**

2.1 Background 3
2.1.1 Fundamental Frequency 3
2.1.2 Musical Notation 6
2.2 Frequency Domain and Time Domain 7
2.2.1 Frequency Domain Approaches 7
2.2.2 Time Domain Approaches 8
2.3 Merits and Drawbacks 9
2.3.1 Merits of FD Approaches 9
2.3.2 Drawbacks of FD Approaches 10
2.3.3 Merits of TD Approaches 10
2.3.4 Drawbacks of TD Approaches 11
2.3.5 Conclusions and Thoughts 13
2.4 A New Approach in the Time Domain 14
2.4.1 Current Workarounds 14
2.4.2 Auto-cross Correlation 15
2.4.3 One-bit correlation 17

2.4.4 Results 19
2.4.5 Select Parameters for Hardware 22
2.5 Chapter Summary 23

Chapter 3 Pitch Shift **25**
3.1 Background 25
3.2 Frequency Domain and Time Domain 26
3.2.1 Frequency Domain Approaches 26
3.2.2 Time Domain Approaches 26
3.3 Limitations of the Time Domain Methods 27
3.3.1 Simple Repeat-or-Discard 27
3.3.2 SOLA Operations 28
3.3.3 Resampling 29
3.3.4 Conclusions and Thoughts 32
3.4 Ring Buffer Approach 32
3.5 Chapter Summary 36

Chapter 4 Hardware Design and Implementation **37**
4.1 Motivations 37
4.2 Schematic Details 38
4.2.1 Top Level Overview 38
4.2.2 Pitch Detection Module 39
4.2.3 Pitch Shift Module 41
4.3 Results 43
4.3.1 Hardware Resource and Power Consumption 43
4.3.2 Simulation Results 45
4.4 Chapter Summary 48

Chapter 5 Conclusions **49**
5.1 What Did We Achieve 49
5.2 What Could be Done 49

Appendices **51**

Appendix A Scource code of the FSM designed for pitch detection **51**

Appendix B Scource code of the FSM designed for pitch shift **55**

References **59**

Chapter 1

Introduction

Sound is an oscillated wave that travels through a physical medium. The faster the oscillation, the higher the sound. The height of a sound is usually referred to as the pitch in musicology, and the pitch of an audio signal is determined by the fundamental frequency of the oscillation. Different musical instruments may bring different harmonic frequencies to their oscillation, resulting in different timbres. However, as long as the fundamental frequencies are kept the same, the pitch is kept the same.

Nowadays, people can record audio signals into digital forms using an audio analog to digital converter (ADC). Once the signal's digital data is preserved, people can then use different approaches to analyze the signal or to modify the signal via digital signal processing (DSP). In audio engineering, one of the most important post-recording processes is pitch correction. If a recorded signal is analyzed to have a pitch that is not in the desired range, an audio engineer can use some DSP techniques to bring the pitch of the signal back to the desired range rather than asking the performer to record a second time.

The pitch correction process is mainly made up of two parts: pitch detection and pitch shift. The pitch detection part tries to estimate the fundamental frequency of the audio and calculates the distance between the current pitch and the desired pitch. Once the distance is calculated, the pitch shift part can then shift the signal by the calculated distance.

The above two processes will not be too difficult to realize in an offline situation where timing constraints are not harsh – digital audio processing can be traced way back to the last century when computing power was no way near today's level. However, performing pitch correction in a real-time environment is a different story. Granted, computing power nowadays is becoming not only more abundant but also cheaper compared to computing power even ten years ago. Yet, there exists little research on real-time pitch correction, let alone research on power-efficient real-time pitch correction. Such a fact raised the interest of this research, which aims to develop a standalone field programmable gate array (FPGA)-based system that can perform pitch correction for a monophonic input (with a main focus on the

vocal signals), with two major requirements: real-time capability, and power efficiency.

The flow of this thesis will be as follows. Pitch detection will be discussed in Chapter 2, followed by pitch shift discussed in Chapter 3. The hardware design and integration of this research will be laid out in Chapter 4, and the whole thesis will be wrapped up by conclusions stated in Chapter 5.

Chapter 2

Pitch Detection

This chapter mainly discusses pitch detection. First, some background of pitch detection and basic music theory will be provided in section 2.1. Two general ways of pitch detection will be introduced in section 2.2 along with the technical details behind them. Then in section 2.3, the merits and drawbacks of both approaches will be discussed, and the approach selected by this research will be revealed. In addition, a novel approach will be proposed alongside its experimental results in 2.4. Finally, a chapter summary will be provided in section 2.5.

2.1 Background

2.1.1 Fundamental Frequency

As stated previously, the fundamental frequency (ω_0) of a particular signal determines the pitch of that signal. An audio signal may contain various harmonic frequencies, which are the frequency components accommodated at a higher frequency location than the ω_0. Besides, harmonic frequencies are positive-integer multiples of the ω_0. Nonetheless, only the ω_0 determines the pitch of that signal.

To get a better understanding of the frequency components of a signal, the Fourier transform needs to be introduced.

$$X(\omega) = \mathscr{F}\{x(t)\} = \int_{-\infty}^{\infty} x(t)\, e^{-2j\pi\omega\, t}\, dt \tag{2.1}$$

Equation (2.1) defines the Fourier transform of a continuous signal $x(t)$, where t is the time unit of the signal x and ω is the frequency unit of $x(t)$'s Fourier transform, $X(\omega)$. The Fourier transform decomposes the signal into an infinite amount of sinusoidal components, and the amplitude of X at ω determines how prominent is the sinusoidal component at phase ω. In other words, Fourier transform brings a signal from time domain (TD) to frequency domain (FD).

However, capturing an ideally continuous signal, which has a time interval of infinity, is simply not feasible. Besides, audio engineers nowadays mostly use an ADC device to convert and preserve analog signals in digital form. On that, the discrete Fourier transform (DFT) of a digital (discrete) signal $x[t]$ is defined in equation (2.2):

$$X[\omega] = \mathscr{F}\{x[t]\} = \sum_{t=t_0}^{t_0+T-1} x[t]\, e^{-2j\pi\omega\, t} \tag{2.2}$$

where t_0 and T are the starting time and the time interval of the digital signal $x[t]$, respectively. It's worth noting that for a real discrete signal that has no imaginary component, its DFT will be symmetric at $\omega = \text{length}(X)/2$ since it has no negative frequency component.

We can now define the fundamental frequency (ω_0) of a digital signal $x[t]$. Ideally, the ω_0 of the signal $x[t]$, if it has any frequency components, is the minimal non-zero ω that yields a local maximum $|X[\omega]|$ (the "sinusoidal" component at $\omega = 0$ is simply not oscillating, thus is not considered as a frequency component). For a real signal, ω_0 would be positive since there is no complex component. Also note that if the higher frequencies are not positive-integer multiples of the ω_0, then the signal will not be monophonic. Figure 2.1 shows the plot of a discrete signal, $x[t] = 0.1\,\sin(40\pi\, t) + 0.5\,\sin(80\pi\, t)$ sampled at 1000 Hz, and its DFT's magnitude:

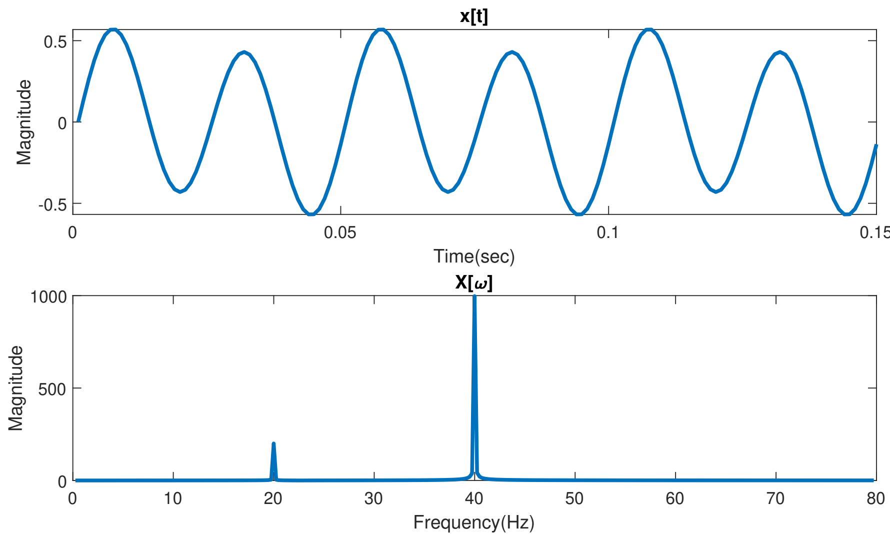

Figure 2.1: A discrete signal $x[t]$ and its DFT $X[\omega]$

The DFT plot shows that there are two local maxima: one occurs at 20

Hz, and another occurs at 40 Hz, which are also the frequency of $\sin(40\pi t)$ and $\sin(80\pi t)$, respectively. In this case, although the 40 Hz component is more prominent due to its higher coefficient, the ω_0 is still considered as 20 Hz. That said, the ω_0 does not always yield the greatest local maximum in the FD.

However, real-world recordings of any type of signal are affected by various sources of noise. Such a factor makes this ideal definition of ω_0 not practical. Consider another plot of a piece of the vocal audio signal recorded via a microphone and its DFT, shown in Figure 2.2:

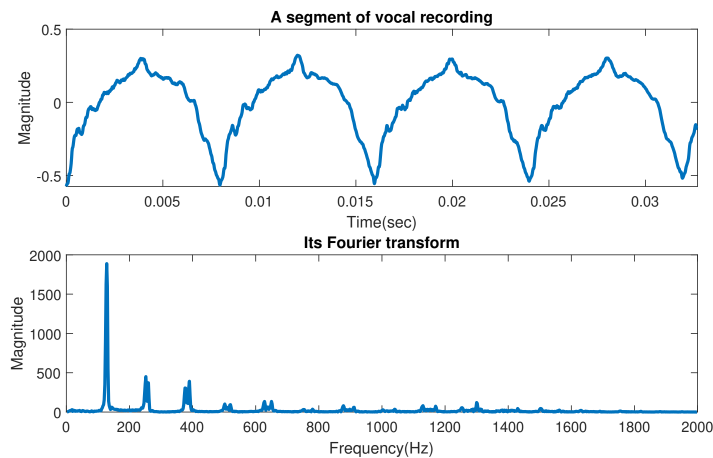

Figure 2.2: A piece of vocal signal and its DFT

Although the major spikes are visible enough, many fluctuations with many local maxima occur throughout the entire span of the signal's FD. Such a fact makes the ideal definition on the ω_0 not practical in real-world usages.

Thus, the definition of ω_0 is very vague in this context. The ω_0 of a digital signal should yield the first "major" local maximum in its FD, but there is no clear definition of how prominent a spike is considered "major". Many musical instruments have stronger harmonic components than their fundamental-frequency component. Such a fact makes the determination of ω_0 much harder. Figure 2.3 plots the recording of a viola playing, sampled at 48 kHz.

One may arbitrarily estimate that the fundamental frequency of the viola sample is at around 262 Hz according to the Fourier transform plot. However, if we plot the result in a logarithmic scale like in Figure 2.4, one may observe that the first major peak occurs at around 131 Hz. Moreover, there also exists a minor spike at around 59 Hz, yet that frequency component is not

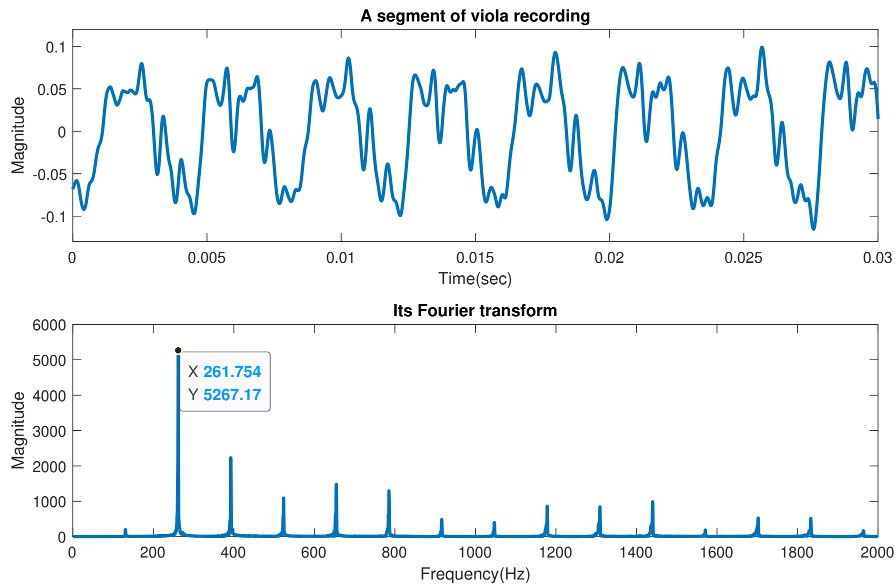

Figure 2.3: A piece of viola signal and its DFT

considered the fundamental frequency due to its low energy level. Thus, it can be concluded that a standard on how to locate the fundamental frequency can be hardly set up.

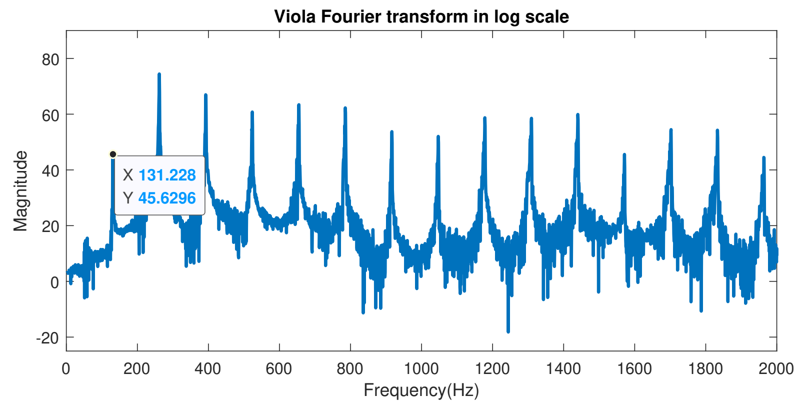

Figure 2.4: DFT of the viola recording in logarithmic scale

2.1.2 Musical Notation

With the ω_0 explained, let us proceed to musical notation. In musicology, there are a total of 12 different notes in an octave. For simplicity, we will

use the numbered musical notation, meaning that each octave is composed of note 1, note 2, ..., and note 12. The note that is higher than note 12 is note 1 but in a higher octave set. Likewise, the note that is lower than note 1 is note 12, but in a lower octave set. Besides, a note will have double the fundamental frequency of the same note in its adjacently lower octave.

The explanation may sound confusing, so let us take an example. Denote note x_i as a note x in octave i, then note x_{i+1}, which is one octave higher than x_i, will have double the ω_0 of x_i, and vice versa. Besides, adjacent notes' frequency will have a multiple of $2^{1/12}$. E.G. assume note 10_4 has a ω_0 of 440 Hz, then note 10_5 will have a ω_0 of 880 Hz, and note 10_3 will have a ω_0 of 220 Hz. Besides, note 11_4, which is adjacently higher than note 10_4, will have a ω_0 of $440 \times 2^{1/12} \approx 466.16$ Hz. Likewise, note 9_4, which is adjacently lower than note 10_4, will have a ω_0 of $440 \div 2^{1/12} \approx 415.31$ Hz. The note higher than note 12_4 is note 1_5, and the note lower than note 1_4 is note 12_3.

In the following section, we will discuss some of the existing approaches to estimating the ω_0 of a piece of digital signal.

2.2 Frequency Domain and Time Domain

In general, there exist two general approaches to obtaining the fundamental frequency of a signal: TD approaches and FD approaches. In fact, the frequency domain approach has just been brought out in section 2.1. We will look further into both techniques in this section.

2.2.1 Frequency Domain Approaches

As previously stated, one can calculate the DFT of an audio signal to obtain its frequency characteristics, and the first major spike shown in the result will be the ω_0. Since the analysis is conducted in the FD, any use of DFT or its variants is considered an FD technique.

When trying to capture the ω_0 of a signal, performing a DFT that covers its entire time span may be impractical. Doing so not only takes more time on waiting for the signal to finish but also suffers from a low resolution in the TD. Imagine a digital signal lasting from t_0 to t_1 changes its frequency characteristics within its lifetime. Then, the DFT of the signal will fuse all the frequency characteristics at different time steps together, resulting in a difficult-to-analyze output. Besides, a longer DFT requires a longer calculation as well, further extending the waiting time.

Thus, it is more practical to perform several short-time Fourier transforms (STFTs), each covering a smaller time interval, rather than a single long

DFT. An STFT is simply a DFT performed in a time window with limited size. Thus, performing STFTs means a more frequent output, yielding an increase in the TD resolution. As a use-case example of the STFT technique, a very early study on harmonics analysis achieves its goal by calculating the STFTs of signals [1].

However, shortening the time frame of STFT will lower the resolution in the FD. This resolution trade-off situation will be discussed in more detail in section 2.3.

2.2.2 Time Domain Approaches

On the contrary to the FD approach, the TD approach "does not" rely on the calculation of DFT nor STFT (more detail on that in section 2.3). TD approaches estimate the fundamental frequency of a periodic or quasi-periodic signal by calculating the similarity between the target signal and its time-delayed self. There are multiple ways to calculate the level of similarity, and two of the most popular choices are the uses of autocorrelation function (ACF) and square difference function (SDF).

$$R_x[\tau] = \sum_{t=t_0}^{t_0+T-1} x[t]\, x[t+\tau] \tag{2.3}$$

$$\mathrm{SDF}_x[\tau] = \sum_{t=t_0}^{t_0+T-1} (x[t] - x[t+\tau])^2 \tag{2.4}$$

Equations (2.3) and (2.4) define the ACF and SDF of discrete signal $x[t]$, correspondingly. In both functions, t_0 and T are the starting time and the time interval of $x[t]$, respectively. It's noticeable that if the discrete signal $x[t]$ starts at t_0 and has a time interval of T, then $x[t+\tau] = 0$ when $t+\tau < t_0$ or $t + \tau > t_0 + T - 1$. Both functions calculate the similarity level between a signal and its time-delayed self. For a periodic or quasi-periodic signal, its period could be determined by the local maxima/minima of its ACF/SDF, correspondingly. Several researchers estimate the fundamental frequency of monophonic signals in TD based on the use of equation (2.3) and/or equation (2.4) [2,3].

Figure 2.5 plots the signal $x[t] = 0.1\,\sin(4\pi\, t) + 0.5\,\sin(8\pi\, t)$ sampled at 100 Hz, along with its $R_x[\tau]$ and $\mathrm{SDF}_x[\tau]$. As one may observe, the $R_x[\tau]$ reaches its absolute maximum, and $\mathrm{SDF}_x[\tau]$ reaches its absolute minimum at $\tau = 0$, when the signal is compared to its identical self. One may also observe that both functions have a tapering effect, this is because $x[t+\tau] = 0$

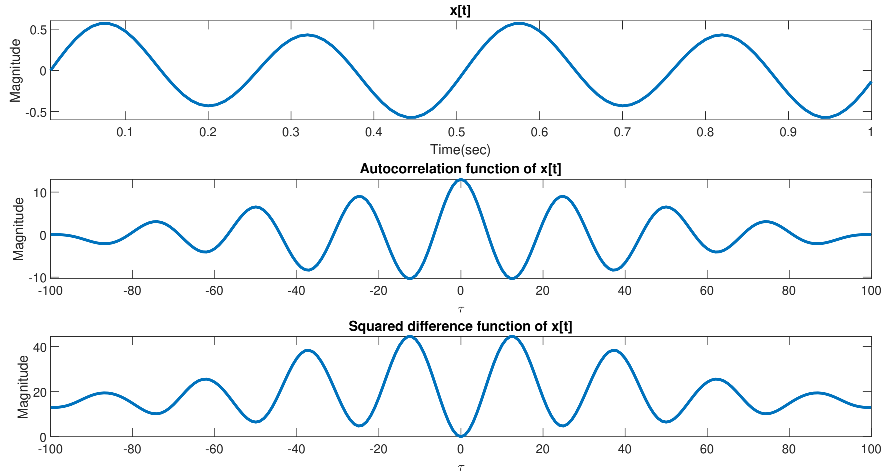

Figure 2.5: Signal $x[t]$ and its ACF and SDF

when $t + \tau < t_0$ or $t + \tau > t_0 + T - 1$. Furthermore, both the ACF and the SDF are symmetric at $\tau = 0$ because $x[t]$ is real.

2.3 Merits and Drawbacks

In this section, we will see some of the merits and drawbacks of both general approaches.

2.3.1 Merits of FD Approaches

One of the major merits of using an FD approach is that it offers very rich information on both the fundamental frequency and the harmonic frequency [1]. Though not necessarily meaningful to the detection of the pitch, the rich information could help perform other audio processing tasks. Past research such as [4,5] perform pitch detection using the methods based on the STFT.

According to equation (2.2), the naive calculation of DFT involving n data points has a time complexity of $O(n^2)$. Yet, an efficient calculation of the DFT named the fast (discrete) Fourier transform (FFT), does exist [6]. Such an algorithm brings down the time complexity of calculating the DFT from $O(n^2)$ to $O(n \log(n))$. For a data point of $N = 2^m$ where m is a positive integer, research [6] calls for about $5N \log(N)$ amount of floating-point operations (FLOPs) to perform the FFT. A latter design further brings down the required amount of FLOPs to $4N \log(N) - 6N + 8$ [7]. Keep in

mind that these numbers all represent the amount of FLOPs, which require much more resource to perform than integer operations.

2.3.2 Drawbacks of FD Approaches

Even though efficient calculations of DFT exist, calculations of both the DFT and the STFT are still considered complicated. Though the time complexities of FFT and its latter variants are $O(n \log(n))$, the calculation of FFT still requires not only FLOPs but also the handle of complex numbers. Both factors severely slow down the calculation process and require a significant amount of hardware resources.

Another major drawback of using a FD approach is that it suffers from spectral leakages [1]. The spectral leakages happen because of the windowing operation used in the STFT, making different frequency components clash together. Think of it as the trade-off of increasing the resolution in the TD – while a smaller-window STFT updates the frequency information more frequently, each window contains less frequency information. For a real signal sampled at SR Hz, its frequency grid, $\mathrm{FG}[t]$, of its N-point DFT, is described in equation (2.5). Keep in mind that a larger grid means a lower resolution.

$$\mathrm{FG}[t] = \frac{\mathrm{SR} \div 2}{N - 1} \tag{2.5}$$

Figure 2.6 demonstrates the effect of a limited frequency resolution. An audio signal produced by a piano is recorded and sampled at 48 kHz, and DFTs are calculated using 800 data points and 22050 data points. When the data points are kept to 800, each bin (I.E. each ω point) only has a frequency resolution of $24000/(800 - 1) \approx 30.04$ Hz, whereas the frequency resolution of using a window size of 22050 is $24000/(22050 - 1) \approx 1.088$ Hz. Thus, even though the true fundamental frequency of the signal is at around 262 Hz, the 800-point DFTcannot locate the frequency component at this frequency, showing a peak at around 240.6 Hz. Moreover, this issue cannot be easily fixed by enlarging the calculation window. Recall from section 2.2.1 that a too-large window will fuse the frequency characteristics at all times together, alongside a decrease in time resolution.

2.3.3 Merits of TD Approaches

A major advantage of performing pitch detection in the TD is that it does not mandate FLOP nor complex-number calculation. Such a fact makes the TD approaches both less time-consuming and less resource-intensive compared to FD approaches that calculate the DFT or STFT naively.

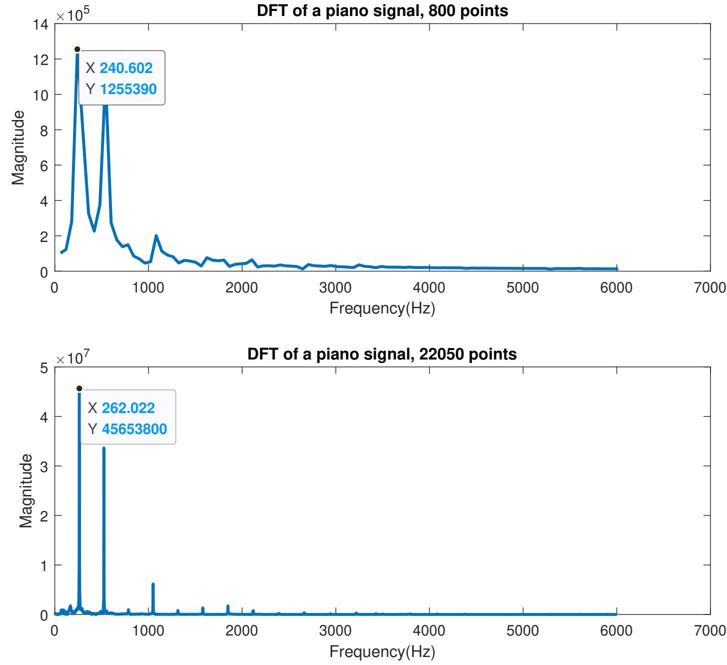

Figure 2.6: Different-size DFTs with different frequency resolution

Besides, since the TD methods perform in the TD only, the trade-off between the TD resolution and the FD resolution is not proportional. For a real signal sampled at SR Hz, the frequency resolution of its N-point window depends on the time index t, and the frequency grid at time FG[t] is given by:

$$\mathrm{FG}[t] = \frac{d}{dt}\frac{\mathrm{SR}}{t} \tag{2.6}$$

As one may observe, FG[t] becomes smaller (which indicates a higher resolution) as t increases due to the inverse-proportion relationship between FG[t] and t. This makes sense since frequency ω is the reciprocal of period t.

2.3.4 Drawbacks of TD Approaches

One of the major drawbacks of TD approaches is the fact that the calculation of an n-sized ACF and SDF has a time complexity of $O(n^2)$ according to equation (2.3) and (2.4). A workaround on calculating the ACF using FFT does exit, which brings the time complexity down to $O(n\ \log(n))$ [8]. However, doing so simply means that the advantage of not involving FLOPs

and complex calculations is compromised. In addition, since FFT is involved, the whole process can then hardly be categorized as a pure TD approach.

In addition, TD approaches can be significantly affected by a signal's harmonic frequencies. Once again, consider the viola sample mentioned at Figure 2.3. Figure 2.7 shows its ACF and SDF. As one may observe, the ACF suggests that the time lag at where the signal finds its most similar self is at $\tau = 185$. Since the sampling frequency is 48 kHz, the suggested frequency is then $48000 \div 185 \approx 260$ Hz, whereas its true fundamental frequency is at around 130 Hz. The SDF also suggests a ω_0 at around 260 Hz. Thus, contrary to FD approaches, the result generated via TD approaches can be much more easily deteriorated by the harmonic frequencies. On that, TD methods also provide little information on multi-phonic inputs, making it not capable of detecting multiple fundamental frequencies at the same time.

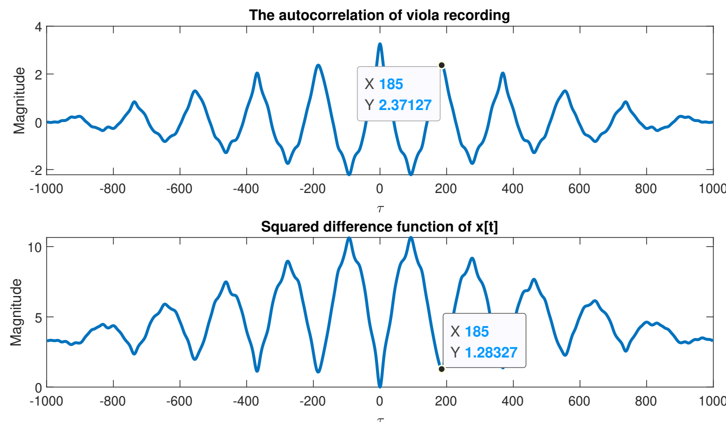

Figure 2.7: ACF and SDF of the viola signal

What's more, the window size of either the ACF or the SDF needs to contain at least 2 periods of the periodic or quasi-periodic signal in order to generate a local maximum or a local minimum, respectively [2]. The reason is that if the window contains less than 2 periods of signal, both the ACF and SDF cannot time delay the original signal to the most similar point, resulting in ambiguous results. Figure 2.8 describe such a circumstance, where the signal $x[t] = \sin(\pi t) + 0.5 \sin(1.5\pi t)$ is sampled at 100 Hz, and the calculation window only contains 200 samples. One may observe that after $\tau > 0$, there is no new maximum in the ACF and no new minimum in the SDF.

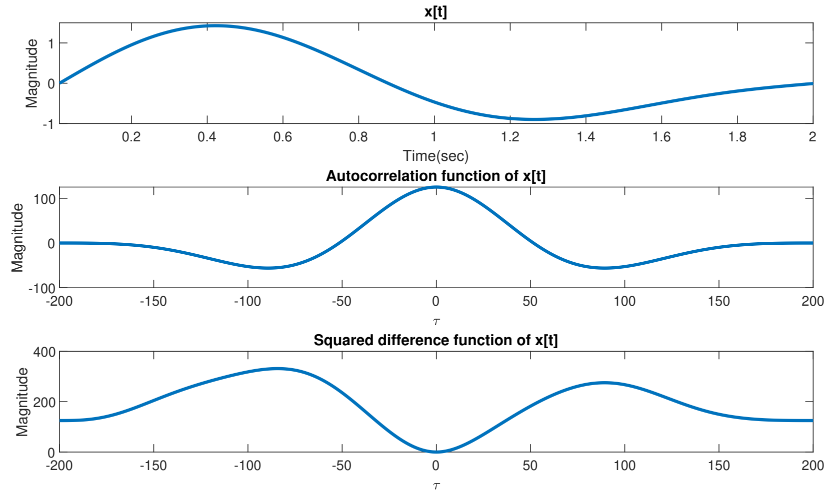

Figure 2.8: ACF and SDF fail to provide any meaningful information

2.3.5 Conclusions and Thoughts

In conclusion, both the FD and the TD methods have their own merits and drawbacks. Many of the drawbacks can be overcome using different techniques, but some hard obstacles cannot be easily avoided. For the FD approaches, it is the involvement of FLOPs and complex calculation. For the TD approaches, it is the efficient calculation that yields a time complexity of $O(n\ \log(n))$ without the use of FFT.

Back to the purpose of this thesis. One of the main requirements of this research is to use as few hardware resources as possible in order to decrease power consumption. The FFT and its latter variants indeed have a time complexity of $O(n\ \log(n))$, yet the achievements are based on the usage of more hardware resources. Modern computing powers are becoming cheaper, but in many circumstances, abundant computing resources cannot be guaranteed, restricted by circumstances that limit the deployment of computing hardware and/or power supplies. Implementing any of the variants of FFT involves complex arithmetic, where a complex addition takes twice the amount of resources as a real addition, and a complex multiplication takes three times the amount of resources as a real multiplication, which itself is a more computing intensive task than real addition. What's more, all of the arithmetic is carried out using FLOPs, resulting in additional multiples of resource consumption than integer-only operation. Since power consumption is one of the major concerns of this research whereas multi-phonic monitoring and processing is not in the scope of this research, we decided to perform all

the operations in the TD only.

The time complexity of calculating either the ACF or the SDF is $O(n^2)$. At first glance, one may think that such inefficient algorithms cannot obey the timing constraint set by any real-time applications. Yet, the actual situation, at least in the context of real-time pitch correction, is not as harsh as one may imagine. In the following section, we will discuss some of the current techniques on how to avoid the pitfalls of TD algorithms mentioned earlier and an all-new design that aims to take the least amount of hardware resources.

2.4 A New Approach in the Time Domain

2.4.1 Current Workarounds

One of the drawbacks of both the ACF and SDF, as mentioned earlier at Figure 2.5, is that they both have a tapering effect. Such an effect may not affect the estimation too much when the fundamental frequency is more dominant than the harmonics but suffers greatly from the situation when the fundamental frequency is less prominent than the harmonics.

Take a look back at Figure 2.5. Both functions suggest that the closest similarity point occurs at $\tau = 25$, which translates to a frequency of $100 \div 25 = 4$ Hz. However, 4 Hz is the frequency of $\sin(8\pi\, t)$, which is not the fundamental frequency. Let us tweak the signal to $x[t] = \sin(4\pi\, t) + 0.5 \sin(8\pi\, t)$ so the fundamental frequency is more dominant. Figure 2.9 shows the adjusted signal with its new ACF and SDF.

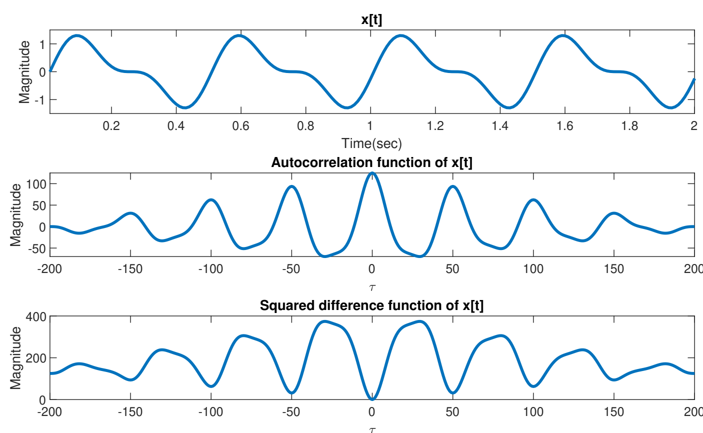

Figure 2.9: ACF and SDF of an adjusted $x[t]$

One may now observe that the ACF and the SDF are yielding the correct result, with the most similar time lag happening at $\tau = 50$, corresponding to the period of $\sin(4\pi\, t)$. Thus, one can conclude that the tapering natural of both the original ACF and SDF have a significantly negative impact on the estimation result. Thus, finding a solution that avoids the tapering effect is crucial.

Different methods of avoiding the tapering effect are discussed in [2, 3] and many other literature. In research [3], the author introduces the concept of "cumulative mean normalization operation", which divides the result at each time-lag point by a limited size of moving average. The moving average decreases along with the decrease of the SDF, resulting in wiping out the tapering effect. However, the choice of the length of the running-average window is crucial, and an inappropriate choice may result in an even worse estimation [9]. In research [2], the concept of "normalized square difference function", which divides the $2\times$ACF value of the signal by the SDF+$2\times$ACF value of the signal, was introduced. One can derive from equation (2.3) and (2.4) that the expansion of the $\text{SDF}_x[\tau]$ contains a term of $-2R_x[\tau]$ in it, concluding that the normalized square difference function will always be in the range of $[-1, 1]$. Since both the ACF and SDF have a tapering effect, the division will cancel out the tapering effect.

However, the task of multiplication itself is already computing-intensive, let alone the introduction of division, or even worse, floating-point division. General-purpose computers nowadays usually have dedicated multiplication hardware, but such a luxury is not available on smaller platforms like a microcontroller unit. Besides, even with the aid of dedicated multipliers in hardware, divisions still take more time to perform than multiplications because divisions can hardly be executed in parallel. Thus, the former solutions for solving the tapering effect, though proven to be effective, are a burden rather than an alleviation to this research.

In the following section, we propose a new calculating method that not only does not suffer from the tapering effect but also helps ease the computing stress of calculating both the ACF and SDF.

2.4.2 Auto-cross Correlation

As stated previously, if the discrete signal $x[t]$ starts at t_0 and has a time interval of T, then $x[t+\tau] = 0$ when $t+\tau < t_0$ or $t+\tau > t_0+T-1$. Such a fact causes the tapering effect: in the context of the ACF, the time-delayed signal is multiplied by zeros when $t+\tau < t_0$ or $t+\tau > t_0+T-1$; in the context of the SDF, the absolute difference is identical to the time-delayed signal's value when $t+\tau < t_0$ or $t+\tau > t_0+T-1$. Research [2, 3] suggest

that one should modify the correlation structure to the description of (2.7), where $\circ$ denotes any type of operation. However, this structure simply skips the terms at when $x[t+\tau] = 0$, which does not necessarily cancel out the tapering effect.

$$\sum_{t=t_0}^{t_0+T-\tau-1} x[t] \circ x[t+\tau] \tag{2.7}$$

We would like to propose a new structure called the "auto-cross correlation function (ATC)" structure. Consider a discrete signal $x[t]$ starting at t_0 and has a time interval of T. The ATC of signal $x[t]$ is defined in equation (2.8).

$$\mathrm{ATC}_x[\tau] = \sum_{t=t_0}^{t_0+\frac{T}{2}-1} x[t]\, x[t+\tau],\ 0 \le \tau \le \frac{T}{2} \tag{2.8}$$

Take a look at the inequality constraint first. $\tau \ge 0$ is set since a negative τ provides information on a negative period. For digital audio signals, which are always real, there is no negative frequency component. Thus, finding the ATC at a negative τ would not provide any useful information since the negative half would just be symmetric to the positive half. The other constraint, $\tau \le \frac{T}{2}$, makes sure that $t+\tau \le t_0+T-1$ holds in τ's entire range.

Taking a further look at the summation, one may realize that the ATC only goes through half of the window size. Doing so not only reduces the number of multiplications to a quarter compared to a traditional ACF operation but also makes sure that the calculation of the ACF never operates at the outside of its window, preventing the involvement of zeros that are out of the bound.

Figure 2.10 shows the ATC result of the same signal used in Figure 2.5, which is $x[t] = 0.1\sin(4\pi t) + 0.5\sin(8\pi t)$ sampled at 100 Hz. In Figure 2.5, both ACF and SDF methods suggest that the highest similarity point shows up at $\tau = 25$, representing a frequency of $100 \div 25 = 4$ Hz. This is the frequency of $0.5\sin(8\pi t)$, and thus is not the fundamental frequency. However, Figure 2.10 shows a higher local maximum occurring at $\tau = 50$, which represents a frequency of $100 \div 50 = 2$ Hz. That said, although the energy level of the $\sin(4\pi t)$ is much less than that of the $\sin(8\pi t)$, the ATC is still able to capture that lower frequency point, which is also the ω_0 in this case. Thus, with the aid of the ATC, we are not only able to alleviate the computing stress by three quarters but also avoid the tapering effect with a better estimation than following the traditional ACF and SDF methods.

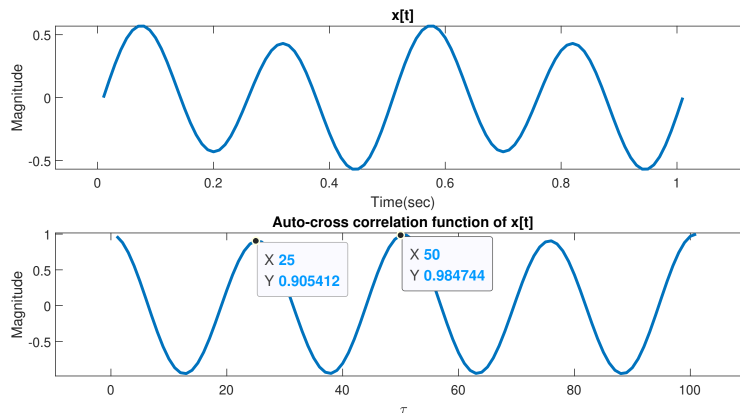

Figure 2.10: ATC result with a better estimation

Another example is shown in Figure 2.11, where the ATC once again is able to estimate the fundamental frequency of the viola signal that both the ACF and the SDF failed to provide an accurate estimate.

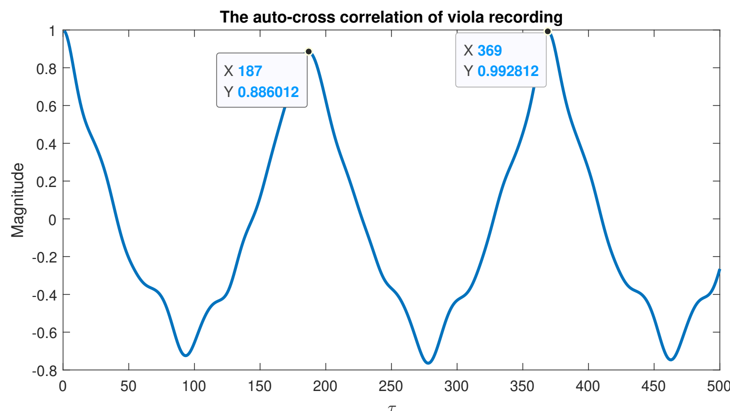

Figure 2.11: ATC of the viola signal

2.4.3 One-bit correlation

Although the ATC can provide a more accurate result with less requirement on the computing power, the calculation is still very inefficient. Most digital audio nowadays is recorded with a bit depth of at least 16 bits. Performing 16-bit multiplication in an $O(n^2)$ manner takes a long time, especially on embedded systems that do not have a dedicated hardware multiplier. Thus, a more aggressive optimization is needed.

To fulfill the requirement of low-energy computing, we would like to propose the operation of one-bit correlation. Let us first look at the property of a square wave. A square wave is a periodic signal that is not sinusoidal – its value jumps between its maximum and minimum. Figure 2.12 shows a square wave and a sine wave both having a frequency of 4 Hz.

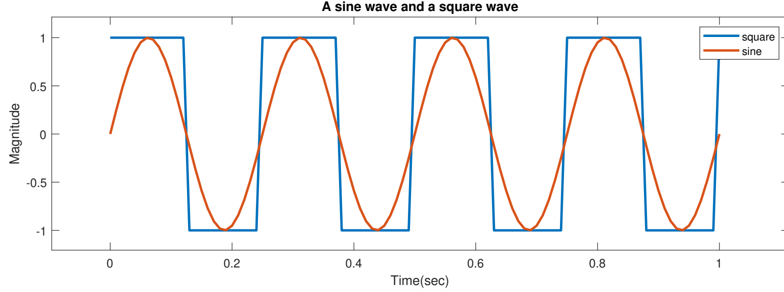

Figure 2.12: A sine wave and a square wave with the same frequency and amplitude

The Fourier expansion of a square wave with a period of $2L$ is described in equation (2.9) [10].

$$\text{square}(t) = \frac{4}{\pi} \sum_{n=2k-1}^{\infty} \frac{1}{n} \sin(\frac{n\pi t}{L}),\ k \in \mathbb{Z}^{+} \tag{2.9}$$

One may observe that a square wave is composed of an infinite amount of sine waves, with the most prominent sine wave staying at the same frequency as the square wave. I.E., the Fourier expansion of a square wave states that there does not exist any sinusoidal component that has a lower frequency than the square wave itself. With that being said, substituting all the sinusoidal components in a signal with a square wave does not introduce any low-frequency component, hence will not change the signal's fundamental frequency. That is, the signal $x_s(t)$ listed in equation (2.10) will have the same fundamental frequency as $x(t)$.

$$x_s(t) = \begin{cases} 1, & \text{if } x(t) >= 0 \\ -1, & \text{if } x(t) < 0 \end{cases} \tag{2.10}$$

Such a fact greatly eases the computation stress of estimating the ω_0 via ATC. Moreover, let us modify equation (2.10) to (2.11). It is obvious that $x_{si}(t) = -0.5x_s(t) + 0.5$. Due to the linearity property of the Fourier transform, each frequency component of $x_{si}(t)$ will have a quarter of the energy as the original frequency component at $x_s(t)$. Nonetheless, since every

frequency band is scaled down by the same factor, the prominence of each frequency component will mostly remain unchanged.

$$x_{si}(t) = \begin{cases} 0, & \text{if } x(t) \geq 0 \\ 1, & \text{if } x(t) < 0 \end{cases} \tag{2.11}$$

Such facts greatly help ease the process of calculating the ATC since every value can now be represented using just ones and zeros. Moreover, in hardware binary computation, the most significant bit (MSB) of any signed integer indicates if a signed integer is negative or not. If the MSB of an integer z is 1, then z is a negative number; if the MSB of an integer z is 0, then z is a non-negative number. Hence, instead of calculating the ATC using the raw data, one can simply calculate the ATC using each sample's MSB, I.E. performing a one-bit correlation. A one-bit multiplication can simply be achieved using a one-bit AND gate, which outputs one only when both inputs are ones and outputs zero otherwise. This little requirement on hardware resources significantly compensates for the fact that the whole process has a time complexity of $O(n^2)$, and is much more appealing than using an FFT to reduce the time complexity.

2.4.4 Results

To verify the accuracy of the one-bit ATC, let us once again use the same viola recording as a test case. Figure 2.13 plots the one-bit ATC of the same viola recording used in Figure 2.11. As one may observe, the one-bit ATC is still able to yield an accurate result just as the normal ATC does.

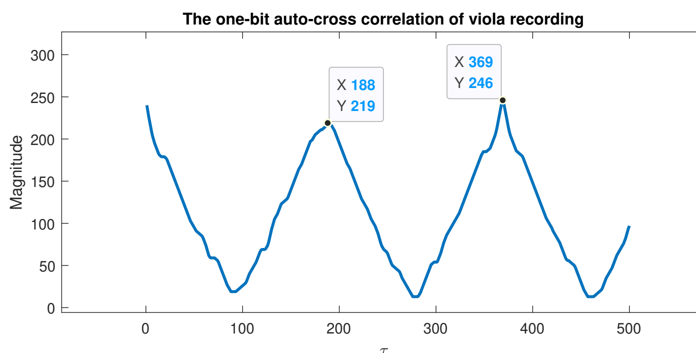

Figure 2.13: One-bit ATC of a viola signal

The one-bit ATC does not suffer from the tapering effect and requires the least amount of hardware resources among all of the existing approaches. All

it needs is a one-bit AND gate to perform the “multiplication”. Since the sum of the multiplications will always be not greater than the value of half the window size (I.E. a 512-sample window will yield a maximum result of 256), the detection module just needs an additional adder that can handle $\lceil \log(\text{window size}) - 1 \rceil$ bits of data. Since the one-bit “multiplication” is never negative, the adder does not even require a sign bit.

In addition, the one-bit ATC can also provide a clarity measure. When a signal is noisy, its autocorrelation tends to hold a very low value. This value is not absolute since the sum of the multiplications depends on the amplitude of the signal, thus a normalization method is needed [2]. However, one-bit ATC does not require normalization for its clarity measure. The reason is that the amplitude of signals has already been standardized at the stage of extracting just one bit from each sample. As previously mentioned, the summation of multiplications will never exceed the value of half the window size. Hence, one only needs to find the difference between the one-bit ATC value at ω_0 and the absolute minimum to determine the level of the noise. Figure 2.13 shows the ATC when a generated white noise is injected into the original viola sample. Although the pattern is still quite similar, the vertical distance from the greatest local maximum to the absolute minimum declined significantly.

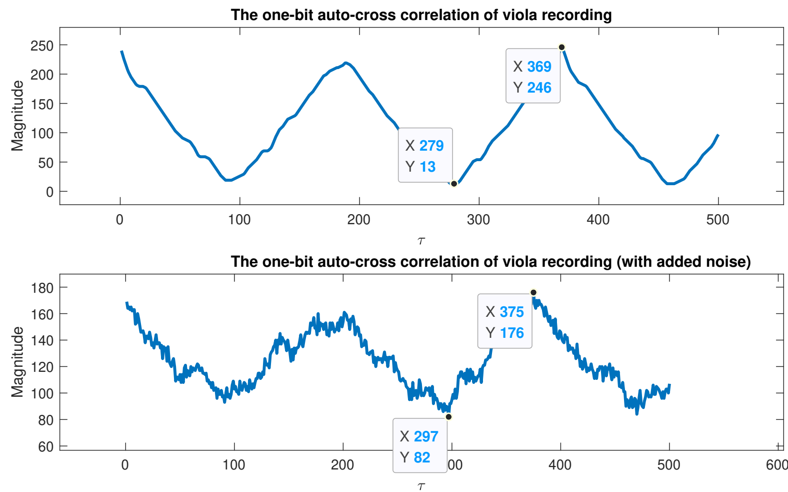

Figure 2.14: One-bit ATC results, with and without noise

It should be pointed out that the greatest local maximum does not always yield the correct estimation. Consider a piece of piano-playing recording, sampled at 48 kHz, and its noise-infused version shown in Figure 2.15. The ω_0 is at around 522 Hz, which is about 92 in terms of sampling period. As

one may notice, the one-bit ATC can barely distinguish the fundamental frequency from the harmonic frequency of even the cleaner signal and fails to provide an accurate estimate using the value of the greatest local maximum. Some musical instruments by their nature have a harder-to-detect frequency characteristic, with a few of them intentionally designed to sound more multi-phonic. The multi-phonic characteristic of these types of instruments can compromise the performances of TD approaches. Figure 2.16 is the result of the previous noisy piano signal processed using the normalized square difference function proposed in [2]. As one may observe, naively choosing the greatest local maximum will still yield an incorrect estimation. Thus, related research calls for a threshold mechanism that would allow certain local maxima to be chosen as the fundamental period, even if they were not the greatest local maxima. The mechanism we propose is as follows: once the ATC detects a local maximum, it will set the temporary estimation to its value's 17/16. Doing so allows this particular local maximum to be chosen as the fundamental period even if a greater local maximum occurs at a later time. We choose 17/16 because dividing an integer by 16 can be easily achieved: one only needs to shift the binary representation to the right by 4 bits (with the least significant bit (LSB) at the right-most position) to obtain the quotient. Then, after the bit-shift operation, we add the quotient back to the temporary value, getting an approximation of 17/16. The number of 17/16 is very close to the reciprocal of the constant factor, 0.93, proposed in [2], and is later proved in [9] to have the most accurate estimation among a variety of instruments. Still, this value cannot guarantee a never-incorrect estimation, because different instruments have different characteristics.

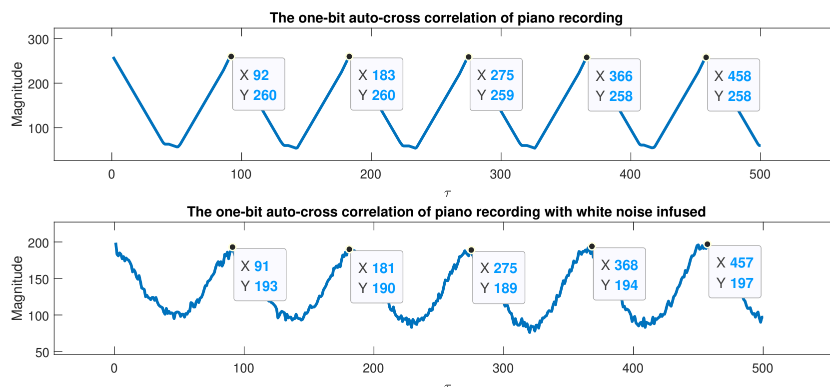

Figure 2.15: More one-bit ATC results, with and without noise

Further experiments were conducted using both self-recorded samples and pre-recorded samples provided by Ableton. The total length of the audio

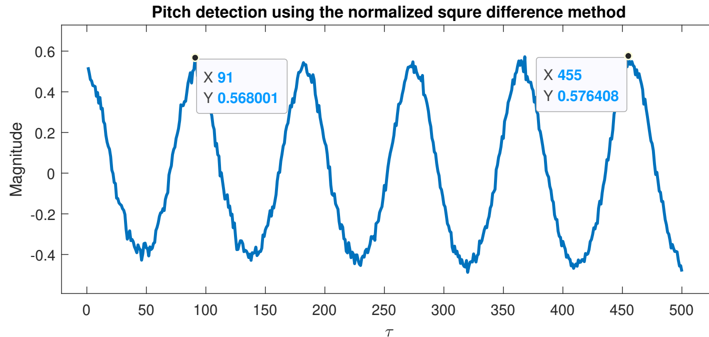

Figure 2.16: Normalized square difference function struggling to estimate the fundamental frequency

signal exceeds 1,440,000 samples, with their fundamental frequency varying from 130 Hz to 1760 Hz. The experimental results show that with the selected multiplier of 17/16, the fundamental frequency calculated by purposed one-bit ATC and never deviates more than 4% of the supposed fundamental frequency, which is less than a deviation of a note. Since the samples are also not guaranteed to be produced at an exact frequency, the deviation is considered as acceptable. To test the clarity measure, we also conduct experiments on an additional 144,000 samples that consist of various kinds of noise, such as the hitting of a keyboard or the plucking of a string. These introduced noises tend to make the one-bit ATC more vulnerable to making an octave error, which outputs an estimation that is twice the supposed fundamental frequency. However, recall from section 2.1.2 that the scale factor between any two notes, no matter their octaves, is the same. Thus, an octave error would not compromise the performance of performing pitch correction.

2.4.5 Select Parameters for Hardware

We will discuss some of the parameter choices of the one-bit ATC to be implemented in FPGA in this section.

The first would be the window size. As previously stated in section 2.3.4, the window size of the calculation needs to contain at least 2 periods of the signal. In order to detect a low fundamental frequency, one needs to enlarge the window size accordingly. However, a larger window size will result in a longer delay due to the wait for arriving samples. After consideration, we decided to choose a window size of 1600 when operating in a 48-kHz-sampling-rate environment. A window size of 1600 will be able to detect a frequency as low as $48000 \div 800 = 60$ Hz, while the waiting time of 1600

samples would be about $1600 \div 48000 \approx 33.33$ milliseconds, a tolerable amount of delay. A shorter delay can be achieved by shortening the window size, which will also raise the lower bound of the frequency detection range. Nonetheless, this delay is purely dependent on the characteristics of the sampling rate, since the one-bit ATC will hardly be the bottleneck when processing signals sampled at 48 kHz.

We then set the clarity threshold at 200. That said, the one-bit ATC module will report an invalid status when the difference between the selected local maximum and the absolute minimum is less than 200. A perfect periodic signal centered at magnitude $= 0$ will have this difference equal to 400, which is a quarter of the window size. Thus, we consider a threshold of 200 as a reasonable value.

In addition, we set a minimal period detection to 18. As stated in 2.3.3, the frequency resolution drops at a more significant rate when the period gets closer to zero. In order to keep a frequency resolution high enough so that it can distinguish two notes that are adjacent to each other in the time domain, we need to make sure that the inequality (2.12) holds. Solving the inequality yields a period of $x \geq 18$, thus we choose a minimal period detection of 18. This determines that the highest frequency it can detect is at around $48000 \div 18 \approx 2666.67$ Hz, which is much higher than the standard upper bound of a female soprano.

Last but not least, we set the output period of the one-bit ATC module to $1600 \div 2 = 800$ samples. Recall that the ATC method only calculates the correlation up to the point at $\tau = T/2$, meaning that the calculation of the ATC terminates at $1600/2 = 800$ sample. Thus, the pitch information can be updated every 800 samples.

$$\frac{\frac{48000}{x}}{\frac{48000}{x-1}} \geq \frac{1}{2^{\frac{1}{12}}},\ x \in \mathbb{Z}^{+} \tag{2.12}$$

2.5 Chapter Summary

In this chapter, we took an overview of the two general pitch detection approaches: the FD and the TD approach. We conclude that one of the major drawbacks of FD approaches, the harsh requirement of hardware resources, cannot be easily avoided, and thus we choose to perform operations in TD only. Then, we checked out some of the existing methods and analyzed what could be done to compensate for the deficiencies. We proposed a novel concept of one-bit ATC, and it is proven to counteract some known flaws of typical TD approaches such as the tapering effect and the harsh requirement

of computing resources. Lastly, we lay down the parameters to be used in the actual hardware design.

Chapter 3

Pitch Shift

This chapter mainly discusses pitch shift. First, a brief introduction to pitch shift will be provided in section 3.1. Two general ways of pitch shift will be briefly introduced in section 3.2. In section 3.3, we will look at some major deficiencies of the existing designs. In addition, a novel approach will be proposed alongside its experimental results in section 3.4. Finally, a chapter summary will be provided in section 3.5.

3.1 Background

Pitch shift can be achieved in a really straightforward manner. To change the pitch by a factor of α where α is a non-negative constant, one simply needs to set the playback speed at a rate of α times the original speed. Doing so will make the original signal $x[t]$ turn into $x[\alpha\, t]$, shrinking the signal when $\alpha > 1$ and expanding the signal when $\alpha < 1$.

However, since the digital signal has a finite length, doing so will inevitably also change the time span of the signal. Depending on the circumstance, a variation in time may or may not be desired. In the context of pitch correction, one would likely prefer not to modify the time interval of the signal, since doing so introduces significant discontinuity at the time when the edit is performed. The goal of this research is also to perform real-time pitch shifts without modifying the time interval.

Similar to pitch detection, pitch shift can also be achieved in either the FD or the TD. However, since it has been determined in 2.3.5 that this research aims to perform all operations in the TD only, we will just briefly introduce some of the FD and the TD pitch shift methods in section 3.2 and then focus on the TD methods.

3.2 Frequency Domain and Time Domain

3.2.1 Frequency Domain Approaches

Just like the FD approaches applied to detect the pitch, the FD approaches aiming to shift pitch also heavily rely on the use of DFT and STFT. As previously mentioned, the Fourier transform offers very rich information in the frequency domain. In the context of pitch detection, richer information is not necessarily helpful since one only needs the frequency information at the fundamental frequency. However, the abundant information can be greatly helpful in the context of pitch shift. Since the DFT also reveals the relationship between harmonic frequencies, one can use these pieces of information to either preserve the harmonic characteristics when performing a frequency shift or to totally change the harmonic characteristics, generating more acoustic effects.

The development of one of the most famous pitch-shifting mechanisms, the phase vocoder, can be traced way back to 1966 [11]. In general, a phase vocoder first performs STFT on segments of a signal, then repositions the area with a significant amplitude, which implies a prominent frequency component, to the desired position. After the repositioning, along with some techniques that improve the sound quality after modification, the vocoder then performs an inverse STFT to transform the signal back to the TD [11].

Other FD methods, though treating the STFTs of signals in different manners, all follow the general flow of performing STFTs, followed by using the obtained frequency information to re-synthesize a new set of frequency pattern, and finished with performing inverse STFTs. A more detailed overview can be found in generative research [12].

Since the scope of this research does not include pitch shift techniques in the FD, we will jump into the analysis of TD methods.

3.2.2 Time Domain Approaches

The technique stated at the very beginning of section 3.1 is a classic way of performing pitch shift in the TD. Although it modifies the time interval of the signal, it performs its operations in the TD only. In order to maintain the time interval of the output at the same level as the input, some segments of the audio signal need to be played back in repetition when shifting up the pitch or be skipped when shifting down the pitch. It is noticeable that if such a device's repetition or skipping pattern is set arbitrarily, it will introduce many discontinuities to the signal, resulting in a low-quality audio

output [13]. However, such an approach does not involve any complicated computation, and thus is very easy to implement even in an analog context.

Besides this straightforward implementation is the famous synchronous overlap and add (SOLA) method. When performing the SOLA, the input audio signal is chopped into frames, each has a small section that overlays on its adjacent frame. When the signal needs to be made shorter, one needs to move these frames closer to each other, and vice versa when the signal needs to be made longer. After the replacement of frames, one then needs to fuse the shifted segments back together. Since each frame will carry a certain amount of the original signal, some of the frequency characteristics will be preserved, resulting in a time-varied signal that has a similar frequency characteristic to the original input. Many TD approaches, such as [14, 15], are based on the SOLA method.

One may have already noticed that these methods by themselves do not, or are not intended to, perform pitch shifts. Since they all try to preserve as much original information as possible, SOLA methods by their nature mainly vary the time interval of a signal. Nevertheless, since the time interval has been lengthened or shortened, one can then change the playback pace of these modified signals, which will yield a pitch-shifted signal with the same time interval as the original input. Thus, to perform an upward frequency shift, one needs to first use any of the SOLA methods to stretch the length of the signal, and then play the stretched signal at a faster rate, resulting in a pitch-shifted-up signal with an identical final length to the original signal. It is worth noting that the order of performing a SOLA operation and changing the playback speed can be swapped, I.E. one can choose to either perform the SOLA operation first or change the playback speed first.

In short, the TD methods can be separated into two main categories, with the first being the simple repeat-or-discard, and the second being the SOLA based operation.

3.3 Limitations of the Time Domain Methods

3.3.1 Simple Repeat-or-Discard

As previously mentioned, this approach is straightforward to implement due to its simple nature. It does not perform any analysis on the current input, nor takes any types of feedback control. Due to the lack of a preprocessing stage, discontinuities will inevitably occur at a repetition or a discarding point, resulting in unpleasant crackling noises fused into the output.

Despite its unsatisfactory performance, the simple repeat-or-discard

methods require almost zero computing resources. All it needs is a memory space that can store some of the inputs in a short amount of time and an "if-else" mechanism that tells it to perform how many times of repetitions or dumps per cycle. As one may realize, a fine-tuned "if-else" mechanism may help improve the quality of its output. Thus, it becomes crucial to implement an additional analysis stage to improve the simple repeat-or-discard approach.

3.3.2 SOLA Operations

The SOLA method will yield a better result compared to that of a simple repeat-or-discard method due to the preservation of a certain amount of original data. However, this operation does not vary too much from the single repeat-or-discard: the only difference is that instead of repeating or discarding a certain frame of the audio, it plays back a longer or a shorter overlaying area of the signal.

Simply stretching or shrinking the overlaying area between adjacent frames can still create many discontinuities. One approach to solve this problem is to apply a smoothing window when extracting time frames [16]. Since the edge of each frame is smoothed out, it will greatly reduce the amount of harsh discontinuities when trying to add separated or gathered frames back together.

However, the smoothing-windowing techniques come at a cost. Extracting frames using a smoothing window, such as a Hamming window or a Kaiser window, means that FLOPs are involved. Besides, trigonometry calculation, the key element of generating a smoothing window, can be very computing intensive. Thus, one either needs to spend more computing resources on obtaining the desired trigonometry components or use a dedicated memory space to store pre-calculated values of the needed trigonometry components. In either case, floating-point multiplications need to be carried out in order to correctly apply the smoothing window. Such a fact puts more burden on the computing unit and does not align with the requirement of using as few hardware resources as possible.

Another technique that helps reduce the number of discontinuities when adding back frames is to find the similarity point within the overlaid area. Like the techniques used in performing pitch detection, calculations like cross-correlation or square difference function can help determine if a similar point exists [17]. If the module is able to find a similar point between two adjacent frames, then it can overlay both frames at that particular point, so that the discrepancy is kept at a low level. However, overlaying frames according to a variable standard makes the final outputs become more inconsistent in time

intervals. Because signals with different frequency components will yield different positions of similar points, the time interval of the final output is not guaranteed to be a fixed value even if the time scaling factors are set to be the same.

Another major drawback of using a SOLA method is that an inappropriate length of frames will deteriorate the result significantly. Ideally, the non-overlaying part of each frame should be kept equal to the length of the signal's period. However, such a condition can hardly be fulfilled in real-time. It was pointed out in [18] that the development of a reliable frame-size determination algorithm is challenging. Moreover, even if we have a reliable algorithm that is optimized to calculate the frame size, the variation of the frame size, along with the variation of the overlaying area and the variation of the correlation-finding area, makes the whole process much harder to be adopted.

We conducted an experiment using one of the SOLA methods mentioned in [18]. Recall that in section 2.4.5, the output period of the pitch estimation is kept at 800 samples. That said, to catch up with the pitch detection pace, the pitch shift module also needs to operate within a window size of 800 samples. However, such a size is too small for the SOLA method to operate, because it simply does not provide enough space to perform meaningful analysis. Figure 3.1 shows the DFT of both a clean piano signal and its time-stretched signal via the SOLA provided in [18]. The window size is set to 800, and the analysis frame size is set to 200. The time scale factor is set to $\alpha = 1.2$. As one may observe, the modified signal no longer maintains a similar harmonic characteristic, and there are many noises occurring around the location of the fundamental frequency, resulting in a multi-phonic output.

Thus, it is shown that the SOLA method is not suitable for real-time processing which operates in a tight window size. Even if it is, it still requires the aid of multiple tools in order to yield a decent result.

3.3.3 Resampling

Up to this point, we describe the task of changing the audio playback speed as the most trivial task in the operation of pitch shifts. However, it turns out that even the most trivial task can be challenging.

When modifying the playback speed, the sampling rate of the original input is modified as well. In an analog context, it would not be a huge issue since the sampling rate is infinite and signals are continuous. However, digital audio is discrete, and it provides no information between each sample. Thus, one would need to guess the data value at a time position in between each sample. Such a process is called interpolation and is a crucial step when

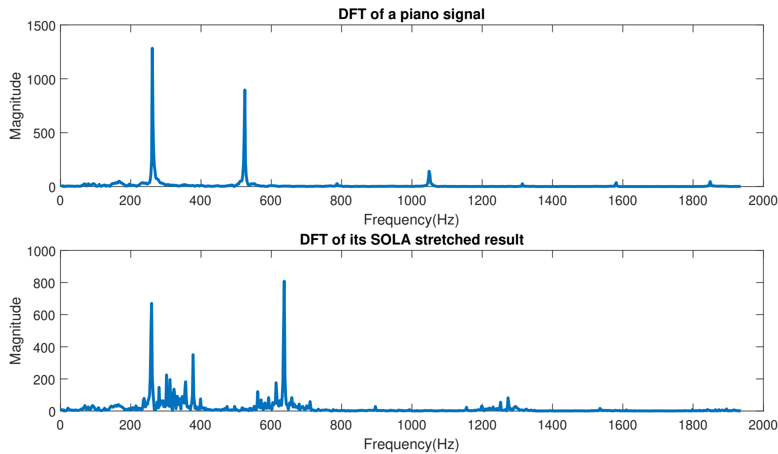

Figure 3.1: DFTs of a piano signal and its SOLA stretched signal

trying to increase the sampling rate. There exist many ways of interpolation, but such a task, usually involving linear filtering, can require a significant amount of computing resources to perform [19]. To maintain a low power consumption, one would have to use the interpolation methods that require the least amount of calculation.

Among all the interpolation methods, the nearest-neighbor interpolation is probably the most straightforward one. When samples need to be inserted at a certain position, nearest-neighbor interpolation simply asserts a value that is equal to its nearest neighbor. Figure 3.2 illustrates the process when trying to stretch a piece of data to its 7/5 using the nearest neighbor interpolation method. It simply asserts two repeating values to the original array at every five-point interval.

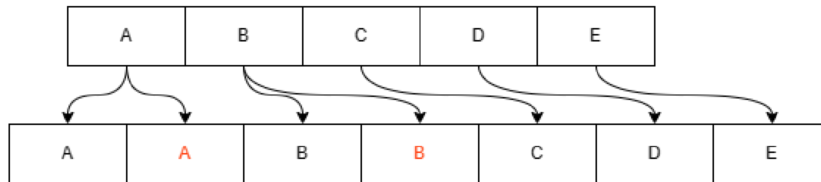

Figure 3.2: Stretching an array using nearest the neighbor interpolation method

This simple interpolation method requires nearly zero computation since it just makes copies of data. It is simple, yet effective. Figure 3.3 shows a comparison between this simple approach and the resampling method provided by MATLAB, which applies a finite-impulse-response anti-aliasing

low-pass filter to help improve the quality of the output [20]. We perform a stretch of 16/15 to a piece of viola signal, resulting in a scale of approximately 15/16 in its frequency. As shown in the plots, both methods successfully shift the pitch down by about 15/16.

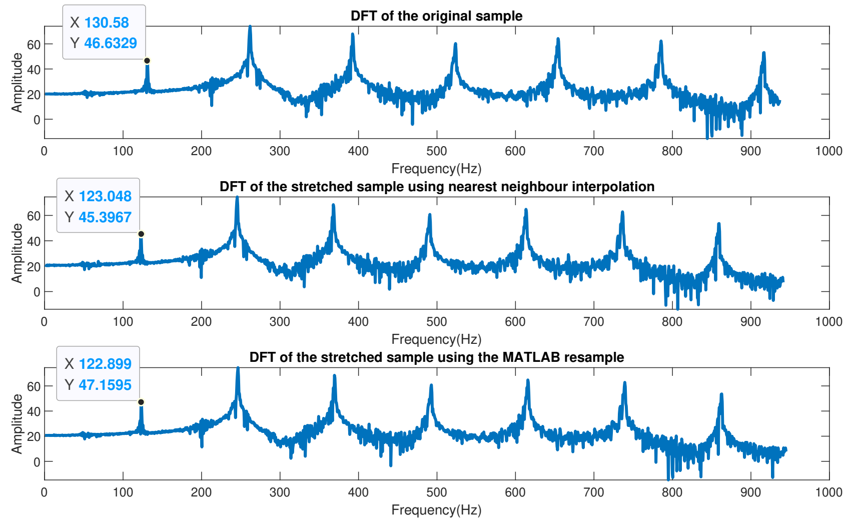

Figure 3.3: Comparison between two interpolation methods

However, taking a further look, one may notice that the nearest-neighbor method introduces a lot more fluctuations at higher frequency bands as shown in Figure 3.4. Such a phenomenon happens due to the fact that the signal's continuity, though not fully destroyed, is compromised as shown in Figure 3.5, with some of the discontinuities highlighted by rectangles.

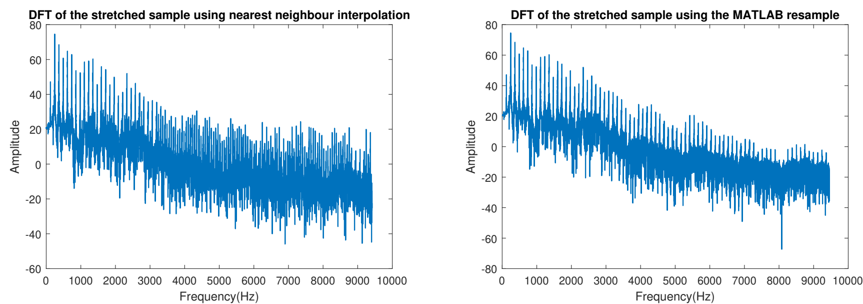

Figure 3.4: Further comparison between two interpolation methods

Nonetheless, these fluctuations all have a smaller energy level and do not interfere with the fundamental frequency. If the top priority is low power

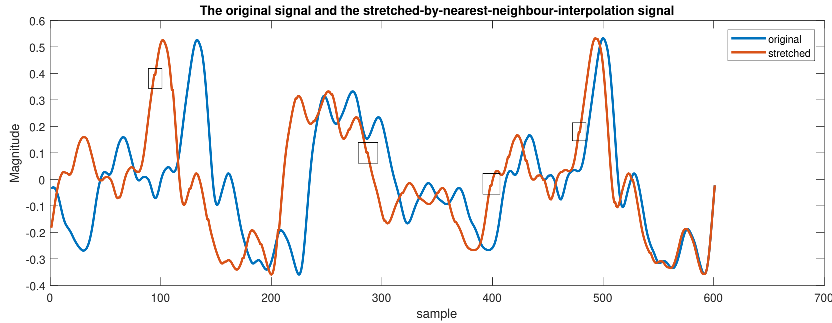

Figure 3.5: Compromised continuity by nearest-neighbor interpolation

consumption, a trade-off in sound quality is acceptable depending on the requirement.

So far, we have been talking about the resampling method of an up-sampling situation. The down-sampling situation is almost identical to the up-sampling circumstance, except that we discard samples instead of inserting samples. Thus, corresponding to the nearest-neighbor interpolation method, one can simply discard an n_1 amount of samples for every n_2 samples to realize a down-sampling rate of $(n_2 - n_1)/n_2$, where $n_1 < n_2$.

3.3.4 Conclusions and Thoughts

We have discovered that the SOLA method, at least with little modification, is not suitable for real-time implementation. There are too many variables to take care of, and it needs extensive aid from all sorts of processing techniques. Thus, we decide to not use any variants of the SOLA method. Instead, we will find a way to improve the simple repeat-or-discard method.

Besides, since we need to implement a resampling process that requires very little amount of hardware resources, we decide to choose the most straightforward approach of using the nearest-neighbor interpolation.

The question now turns into "how to improve the simple repeat-or-discard method". In the following section, we will purpose a workaround to compensate for the natural deficiencies of the simple repeat-or-discard method.

3.4 Ring Buffer Approach

A historical method of performing a time expansion or compression to an audio recording, back in time when audio signals are mainly recorded in

analog forms, is to set up a two-reel tape-playing system like the one designed in [21]. These two reels will be wrapped with tape that contains audio information. By rotating the reel, the replay head can read the signal imprinted on the tape continuously. On that, one of the reels will rotate at a speed that is the same as the rotational speed used at the recording stage, while the other reel's rotational speed is controllable. Thus, by rotating the controllable reel at different rates, alongside the use of different types of replay heads, a time expansion or compression can be achieved.

A modern replica of the two-reel mechanism can be built using a ring buffer. A ring buffer is a memory unit that has a similar structure to a first-in-first-out (FIFO) memory structure, except that the new data will overwrite the most dated data when the ring buffer is at its full capacity. This type of memory unit has a very similar characteristic to a reel-based audio playback device, except that a reel operates with analog data while a ring buffer operates with digital data. A ring buffer, like all memory structures, has a pointer that helps identify which memory address to access. The memory pointer, in the context of audio-signal reading, acts similarly as the replay head of a tape-playing device. The main difference between an analog and a digital ring-storage structure is that an analog system would require two reels to perform writing and reading simultaneously, while a single digital ring buffer can be capable of performing simultaneous writing and reading – as long as it supports dual-port operation, it only needs an extra pointer to distinguish the writing and the reading position. To speed up/slow down the playback speed, one only needs to increase/decrease the moving speed of the output pointer. Therefore, we will implement a ring buffer with an input and an output pointer, with the input pointer moving at a constant speed and the output pointer moving at a variable speed, to achieve pitch-shifting operations.

With the memory structure and playback method decided, it is now time to focus on the discontinuity issue. As previously mentioned, performing repeat-or-discard actions trivially would generate an enormous amount of discontinuities. The same concept applies to the ring buffer structure, too. Since a ring buffer has a finite amount of memory space, if the output pointer moves too slowly, it will be over-passed by the input pointer by a cycle. Since the input pointer will overwrite the old data, the output pointer will read the new data in an abrupt manner. Likewise, if the output pointer moves so fast that it passes the input pointer, it will start outputting data that is not yet updated, making it read the dated data in an abrupt manner. Thus, it is crucial to design a mechanism that keeps the two pointers from over-passing each other, and the output pointer should be kept behind the input pointers at all times.

Nevertheless, since both pointers operate at different speeds when performing pitch shift operations, it is inevitable to jump the output pointer to a safe location. A trivial jumping mechanism will have the same defects as not performing the jump, since an arbitrary jump will still generate a discontinuous point, resulting in a low-quality output. To counteract this situation, we design a simple jumping mechanism that can help keep the discontinuity at a minimal level.

As stated in Chapter 1, a sound is generated by an oscillating wave, and its fundamental oscillating period can be detected using methods introduced in Chapter 2. That said, the safest position for jumping the output pointer will be either a fundamental period before or after its current position. Yet, worth noting that most audio signals are considered quasi-periodic signals, because the amplitude of any sound may vary over time. Moreover, an audio signal's harmonic characteristic may also vary over time, depending on the vibrating material that generates the oscillating wave. Thus, a jumping distance of a whole fundamental period does not guarantee 100% integrity. Nonetheless, it is still the safest distance to perform a jump, since it offers the maximal similarity among the operated signal.

We can now lay out our design of the pitch shift module. As the system starts working, the input pointer, moving at a constant speed, will start recording the incoming samples into the ring buffer, whereas the output pointer will stay at the initial location. The output pointer will start moving after there is a certain amount of data available in the ring buffer. Once there is enough data, the output pointer will then start outputting the previously stored samples at a rate according to the scale of pitch shift. If the distance between the input and output pointer exceeds a preset maximal distance, the output pointer will jump forward by a distance of the current fundamental period of the signal. Likewise, if the distance is shorter than a preset minimal distance, the output pointer will jump backward by a distance of the current fundamental period of the signal. Hence, the output pointer is never ahead of nor over-passed by the input signal, maintaining a similar time signature as the input while keeping the discontinuity level at a minimum. Figure 3.6 shows an example circumstance when the output pointer is about to take over the input pointer.

To verify this concept, we performed experiments using different samples. Alongside the nearest-neighbor interpolation, we are able to get a result output that keeps the same time-interval signature as the input, while shifting its pitch. Figure 3.7 shows the plot of performing the purposed pitch shift operation. We shifted the pitch downward by a factor of 5/6. As one may observe, with an almost identical sample size, the period of the signal is lengthened. Furthermore, a DFT analysis shows that the frequency

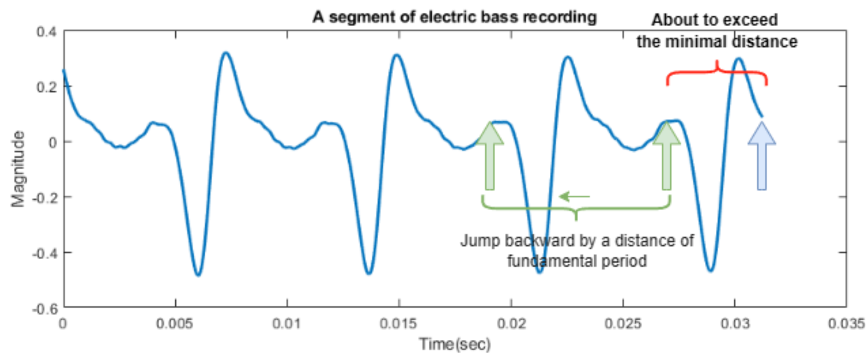

Figure 3.6: Appropriately jumping the output pointer

characteristics are kept at a close-to-original level, which is far superior to the SOLA operation's output shown at Figure 3.1.

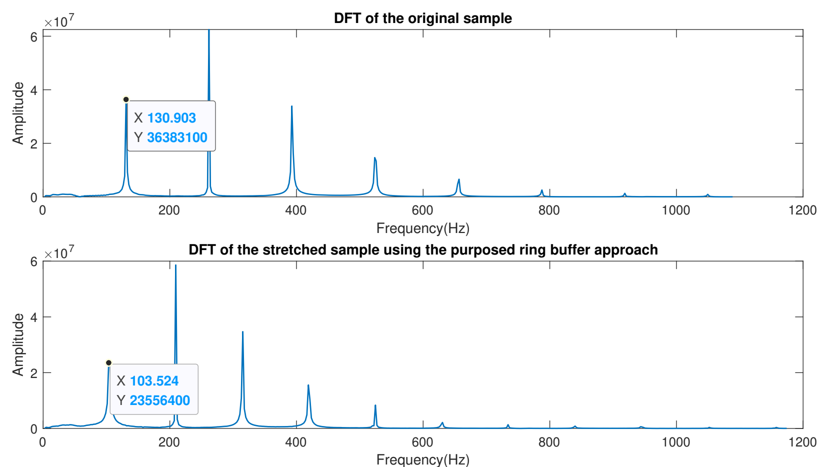

Figure 3.7: DFT analysis on the stretched signal

Since the module itself does not have any ability to estimate the fundamental period of the signal, it will need external help from other modules. Fortunately, a pitch correction system mandates a pitch detection module, which has been introduced with details in section 2.4.2 of Chapter 2. With that being said, the pitch shift module does not need to perform a redundant pitch detection, which further lowers the overall computation cost. All we need to do is to calculate the factor of a desired pitch shift, apply that factor to the output pointer of our ring buffer design, and jump the output pointer in appropriate circumstances.

Our pitch detection module starts working after the arrival of the 1600th

sample and update period information every 800 samples. To keep the pitch shift module synchronized, the output pointer should also wait for 1600 samples before starting to move. Besides, the shortest distance between the input and the output pointer should not be less than 800 samples, and the longest counterpart should not be more than 1600 samples. Those are all the parameters needed to be set when performing the pitch shift operations in our system.

3.5 Chapter Summary

In this chapter, we discussed some existing methods designed to perform pitch shift operations. We analyzed the merits and drawbacks of different types of pitch shift solutions, alongside considerations of interpolation methods, and we concluded that a novel ring-buffer-based design helps us satisfy the requirement. We conducted experiments to verify our concept, and the experiment results showed that the ring-buffer design indeed has superior performance than the SOLA method in a real-time context. Next, we revealed the fact that our pitch shift module does not need to perform any redundant calculations, since a pitch detection module has already been implemented. In the end, we discussed the parameters to be used in the actual hardware design so that the pitch shift module can work with the pitch detection module at the same pace.

Chapter 4

Hardware Design and Implementation

We've already discussed the theories behind our selected approach. In this chapter, we will focus on the design of the actual hardware. We will first briefly discuss the reason to implement our design using FPGA in section 4.1. Then, we will lay out our schematics at each level in section 4.2. We will present the experiment results in section 4.3, and wrap up the chapter with conclusions stated in section 4.4.

4.1 Motivations

As stated in Chapter 1, the main goal of our research is to design a real-time pitch correction system that is power efficient. Such a system not only requires two modules – pitch detection and pitch shift modules – to run simultaneously but also asks for the least amount of hardware-resource usage. Thus, in order to maintain a high configurability and perform parallel computing while also achieving low power consumption, an FPGA platform would be the ideal foundation for our design and development.

Prior to this study, numerous studies were conducted on using FPGAs to alleviate certain computing pressure or increase the speed of certain computations, such as those conducted in [22, 23]. Besides, many studies also focus on the development of FFT cores using FPGAs, which are closely related to this research [24, 25]. However, there exists little to no research that aims to implement a standalone real-time pitch correction system. All these factors motivate the proposal of this research, which will be revealed in detail in the following sections.

4.2 Schematic Details

The theory behind our purposed system has already been discussed with details in Chapter 2 and Chapter 3. Thus, we will only focus on the actual design and implementation of our proposed system.

4.2.1 Top Level Overview

The top level of our design is very straightforward. Figure 4.1 shows the top level of our system

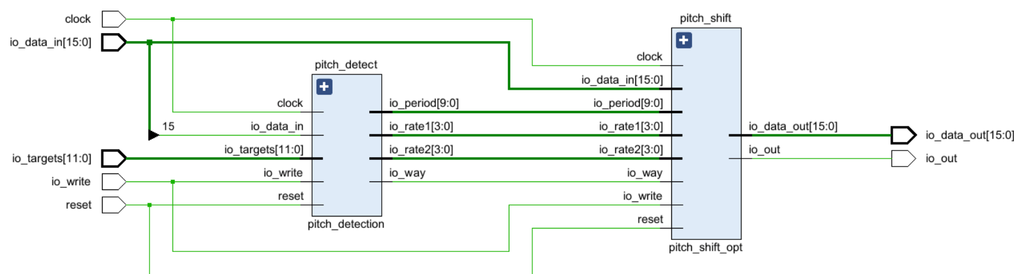

Figure 4.1: Top-level schematics of the proposed system

The top level of our design has 5 inputs and 2 outputs. Our design has sequential components, thus a clock input is needed. The io_data_in represents a parallel input of an audio sample. Since a bit depth of 16 bits is the most commonly used digital audio bit depth, io_data_in is also a 16-bit wide. The io_write signal is the input that indicates whether the current io_data_in is valid or not. If it is at a low level, the whole system will not proceed to calculate. The io_targets input indicates the current desired target notes of its user's preference. Recall in section 2.1.2 that an octave contains 12 different notes. E.G., if one does not want note 3 to appear in the final output, one can switch off the input corresponding to note 3, telling the system to shift every detected note 3 away. The last input, reset, simply resets all the sequential components in the system.

Only two outputs are generated in our design: the io_data_out, which is a 16-bit wide output that carries the output generated by the pitch shift module. There is also a io_out signal that indicates the output io_data_out is valid.

Inside the top level is the two core modules that run in parallel while communicating with each other. As their name suggests, the pitch_detect module calculates the fundamental pitch, and the pitch_shift module performs the pitch shift operation.

4.2.2 Pitch Detection Module

We will now look at the pitch detection module. Figure 4.2 shows the schematics of the pitch detection module. It is made up of four components as follows:

1. The `get_period` component, which is the core of the pitch detection module that calculates the estimated fundamental period and feeds it to the pitch shift module;
2. The `period_to_note` component, which translates the fundamental period to its corresponding musical note;
3. The `get_distance` component that finds the distance between the current note to the next closest target;
4. The `distance_to_rate` component that translates the distance to a pitch-shift scale factor for the outer `pitch_shift` module.

4.2.2.1 FSM that calculates the pitch

The `get_period` component is the core element of not only the pitch detection module but also the entire system. Without its proper functioning, pitch correction cannot be carried out. This module is mainly made up of an finite-state machine (FSM) and a true dual-port memory array that has 2048 addresses for one-bit wide words. We implement a true dual-port memory because we need to check two different locations, with one accessing the location of $x[t + (T/2)]$ and another accessing the location of $x[t + \tau]$.

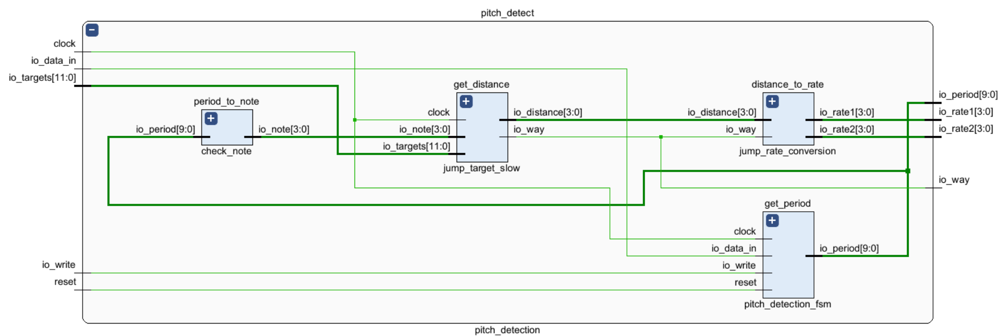

Figure 4.2: Schematics of the pitch detection module

Since this component needs to access two memory addresses simultaneously, it needs two memory pointers as well. Besides, it needs several registers to store the pitch information as well as miscellaneous registers to determine the current state.

Here are some additional technical details:

1. Once there is enough data stored in the memory address, the calculation of the $\text{ATC}_x[\tau]$ starts right after the arrival of a new input. Recall that for each τ, $t + \tau$ is always less than or equal to $t + (T/2)$. Thus, the calculation of $\text{ATC}_x[\tau = 1]$ only requires 801 data to be available when $T = 1600$;
2. This FSM actually requires three pointers: one pointer for writing the data and two pointers for reading the data. However, since the reading process should not start until the writing action is finished anyway, a multiplexer is enough to share the port between the writing pointer and one of the reading pointers;
3. There is no way to check if $\text{ATC}_x[\tau]$ reaches a local maximum right after the calculation of $\text{ATC}_x[\tau]$. Whether $\text{ATC}_x[\tau]$ is a local maximum or not can only be determined at $\tau = \tau + 1$;
4. Synchronous-read memory costs an additional clock cycle to read the data, thus the FSM has to prepare an extra cycle for the calculation;
5. Each $\text{ATC}_x[\tau]$ requires about 810 cycles to be settled down, with 800 of them being the calculation processes. Thus, the frequency of `io_data_out` should not exceed $F/810$, where F is the running frequency of the FPGA. The frequency of the `io_data_out` is usually determined by the audio sampling frequency, depending on the actual use case. That said, for a sampling frequency of 48 kHz, the FPGA's frequency should be higher than $810 \times 48000 \approx 38.88$ MHz.

It's worth noting that since the core of the ATC calculation only requires one AND gate, one can easily put several AND gates in parallel so that the operating frequency of the FPGA can be lowered if there is such a desire.

The source code that describes the above FSM will be provided in Appendix A.

4.2.2.2 Translation from period to note

The entire `period_to_note` component is just a look-up table that helps reference the correct note value. The conversion between note and period/frequency is rather complicated since it involves the term $2^{1/12}$. Because a 1600-sample window will be able to estimate a maximal fundamental period of 800 samples long, the look-up table is simply a block of memory that is 800 words wide, with each word having a length of 4 bits (since the maximal number is 12).

4.2.2.3 Determine the distance to target

The get_distance component calculates the distance between the current note and the closest active target. Worth noting that the distance between note 12 and note 1 is 1 rather than 11, thus this part also involves signed addition/subtraction. The overall logic can be summarized as follows:

1. Calculate the distance between the current note and note 1;
2. If note 1 is active, output the actual distance to the final comparison module. Otherwise, output a distance of 12 to that module;
3. Doing so for all 12 notes in parallel;
4. The comparison module will select the minimal value as its output.

4.2.2.4 Translation from distance to a pitch scaling factor

The distance_to_rate component is again just a look-up table that helps reference the correct pitch scaling value. As introduced in section 2.1.2, the pitch scaling factor of a fixed distance is also fixed, I.E. a distance from note 3 to note 5 will yield the same pitch scaling factor as a distance from note 4 to note 6, since they both have a distance of 2. In addition, worth noting that the maximal distance between any two notes is 6.

One may observe from the schematics that this module has three outputs labeled as rate1, rate2, and way. The purpose of setting up these signals will be explained in section 4.2.3.

4.2.3 Pitch Shift Module

We will now look at the pitch_shift module, which has a very simple structure. An overview of this module is shown in Figure 4.3.

The pitch_shift takes the audio sample as one of its inputs. Besides, it also takes the information on fundamental frequency and the pitch scale factor from the pitch_detection module. The pitch_shift module, just like the pitch_detection module, also operates at a pace decided by the io_write signal that comes from the external of the FPGA unit.

The pitch_shift module itself is basically an FSM that contains a ring buffer. The size of the ring buffer is 2048 words wide, with each word having a length of 16 bits, which is kept the same as the io_data_in. It was previously mentioned in section 3.4 that the maximal distance between the output pointer and the input pointer is 1600 samples. Thus, a ring buffer size larger than 1600 would suffice this condition. Another reason for choosing a memory width of 2048 is that it helps skip the process of resetting the pointer's value since $2047 + 1 = 0$. Likewise, we do not need to worry about

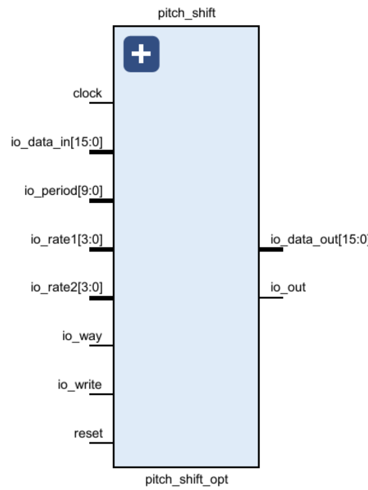

Figure 4.3: An overview of the pitch shift module

taking an absolute value when trying to figure out the distance between the input and the output pointer since $A - B = 2048 + A - B$ when performing unsigned subtraction in a bit width of 11 bits.

We will now explain the logic behind the `rate1`, `rate2`, and `way`. First, `way == true` means the required pitch shift is an upward pitch shift, and vice versa for a downward pitch shift. Next, the `rate1` and `rate2` work as follows: "for every `rate1` amount of samples, stay at/skip `rate2` amount of samples if performing a downward/upward pitch shift". E.G., if one wants to raise the pitch of note 2 to note 3, one can set `rate1=16; rate2=1; way=true`. In that case, the `pitch_shift` module will skip 1 sample for every 16 samples, resulting in a playback speed of $17/16 \approx 2^{1/12}$. The same logic applies to every distance of pitch shift, except that the `rate1`, `rate2` and `way` are automatically controlled by the `pitch_detection` module. Besides, if the `pitch_detection` module determines a signal is too noisy to be detected, it will set `rate1` and `rate2` to zero to prevent the `pitch_shift` module from varying the movement speed of its output pointer.

The source code of the `pitch_shift` module will be provided in Appendix B.

4.3 Results

4.3.1 Hardware Resource and Power Consumption

We will report the required amount of hardware resources and the estimated power consumption of our design in this section. The design was synthesized and implemented via Vivado 2022.1, on a Basys 3 FPGA board that carries the Xilinx Artix-7 XC7A35T-ICPG236C FPGA chip. The board is set to operate at its native clock frequency, which is 100 MHz.

4.3.1.1 Hardware resource

Table 4.1 lists the hardware resources required at a component level. Note that `get_period` is a part of `pitch_detection`. The "LUT" and "BRAM" listed in the table mean "look-up table" and "block random-access memory", correspondingly.

Component	LUT	Register	BRAM
`get_period`	131	101	0.5
`pitch_detection`	410	111	0.5
`pitch_shift`	62	41	1

Table 4.1: Hardware Resource – Component Level

Table 4.2 lists the hardware resources required at the top level. We constructed a total of 3 configurations. The "WNS" listed in the table means the worst negative slack.

Top Level	LUT	Register	BRAM	F7Muxes	F8Muxes	DSP	WNS
Config. 1	585	217	1.5	67	13	0	1.868 ns
Config. 2	518	210	1.5	54	13	0	1.172 ns
Config. 3	474	205	1.5	3	0	0	0.085 ns

Table 4.2: Hardware Resource – Top Level

4.3.1.2 Power consumption

The power consumption reported by Vivado's default post-implementation power report is listed in Table 4.3.

Components	Power (milliwatt)
Clock	3
Signals	2
Logic	1
BRAM	5
Inputs & Outputs	30
Device Static Power	72
Total	113

Table 4.3: Power Consumption of Our Design

4.3.1.3 An informal comparison

We are not able to find any formal research that shares the same goal as ours. In addition, commercial solutions that function in the same way as ours are hard to find. Most commercial hardware that performs real-time pitch correction also integrates several modules, such as ADCs and digital to analog converters (DACs), to make the device plug-and-play. In contrast, our design only focuses on signal processing. Thus, a comparative study is difficult to conduct.

Nonetheless, there exist several recent studies on the development of FPGA FFT cores. Those studies can be a somewhat good reference that reflects the importance of avoiding the FD approaches.

Research [24] purposed an FPGA intellectual property that can calculate 1024-point 16-bit real-time FFT using 466 look-up tables (LUTs), 32,768 bits of random-access memory, 16,384 bits of read-only memory and 4 hardware multipliers. The LUT and random-access memory usages are very close to our design, yet it requires more read-only memory and dedicated hardware multipliers. The number of registers used in this FFT architecture was not revealed.

Research [25] purposed an FPGA architecture that uses 6043 LUTs, 5264 registers, 8 block random-access memorys (BRAMs), and 24 hardware DSP units to perform a 24-bit fixed-point FFT with a window size of 256 points. This approach requires much more hardware resources compared to the previous one. However, this architecture can not only operate at a much higher frequency but also takes fewer cycles to perform the calculation.

In short, one may conclude that the implementation of FFT cores is not trivial. Moreover, FFT itself does not perform any modification in the FD, meaning that one needs to spend additional hardware resources on a modification mechanism. Therefore, we choose to avoid FD approaches since

our goal does not require one.

4.3.2 Simulation Results

We performed several post-implementation timing simulations, and the simulation results are as expected. First, we recorded a piece of a vocalist humming the note C (1), D (3), and E (5). Then, the sample was fed to our design. We recorded the real-time generated output and performed DFT analysis on the output. The experiment results are shown as follows.

In our first experiment, we set the target note to a single D, meaning that all the notes that are detected not as a D will be shifted to the location of D. We then analyzed the result by applying the DFT across the entire time span of the modified signal. The analysis is shown in Figure 4.4.

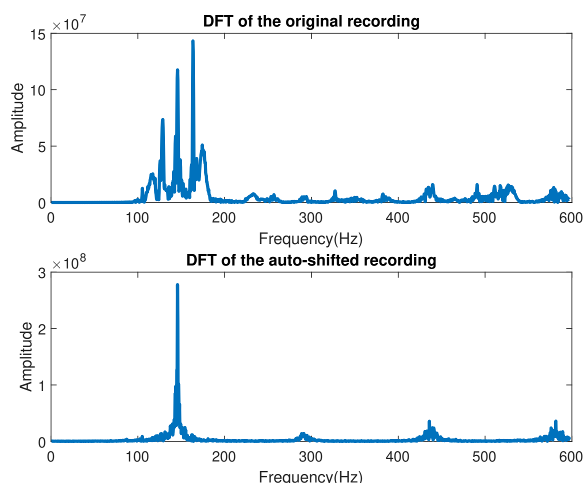

Figure 4.4: Real-time pitch correction targeting at note D

As one may observe, the DFT of the original record shows that there are 3 major spikes centered at around 128 Hz (close to C), 146 Hz (close to D), and 163 Hz (close to E). There are also two minor spikes that occur at around 115 Hz and 174 Hz. Both spikes occur because the unprofessional vocal performer (that's me) was not able to find the right pitch at all times, and the DFT is able to capture these frequency characteristics. Nonetheless, taking a look at the DFT analysis of the auto-shifted recording, we may observe that most spikes are either shifted to around 147 Hz, or to around 294 Hz, which is the note D in an octave higher. We then ran multiple experiments, each targeting at note C, E, and F# (note 7), with their DFT

analysis plotted in Figure 4.5, Figure 4.6 and Figure 4.7 accordingly. The results speak for themselves.

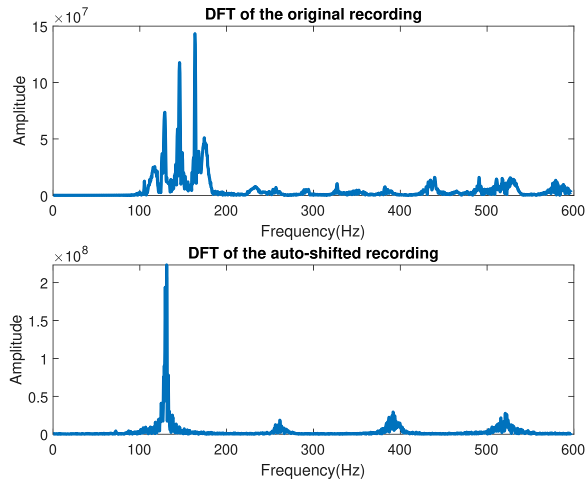

Figure 4.5: Real-time pitch correction targeting at note C

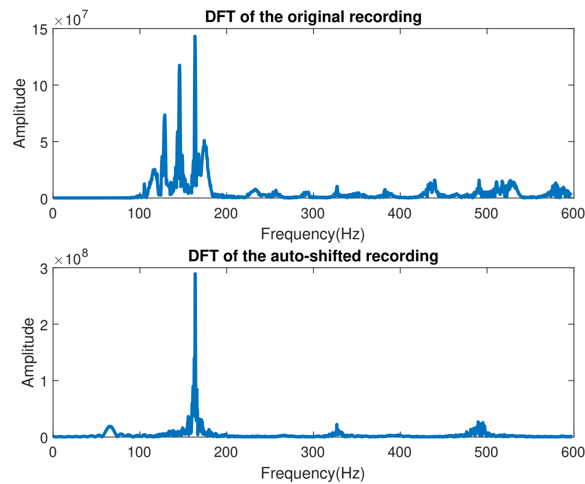

Figure 4.6: Real-time pitch correction targeting at note E

At last, we deactivate all the targets, making the system force shift every detected note to an octave higher. The DFT analysis is plotted in Figure 4.8

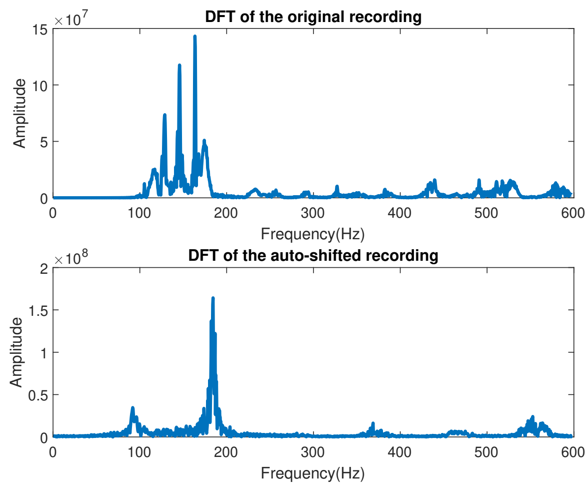

Figure 4.7: Real-time pitch correction targeting at note F#

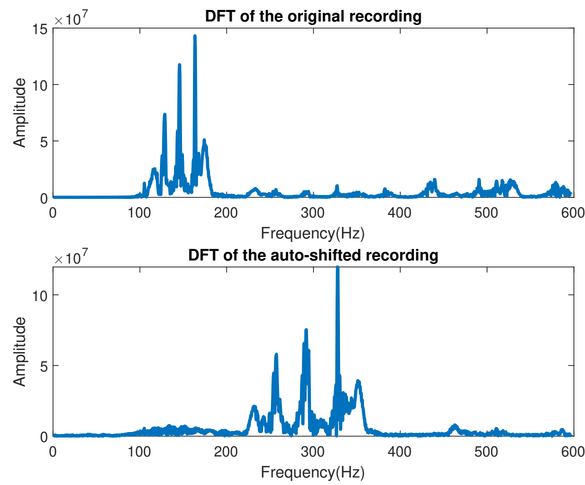

Figure 4.8: Real-time pitch shift targeting at one octave higher

We also performed an informal listening test and found the auto-shifted results all carried subtle crackling sounds. Moreover, when the recording enters a transition between notes, the system, depending on its target, may make a noticeable clipping noise at the transition.

4.4 Chapter Summary

We first stated the motivation for using an FPGA platform to realize our design. We then presented the technical details of our hardware design. After that, we listed the DFT analysis of the post-implementation timing simulations and showed that the system outputs were as expected.

Chapter 5

Conclusions

5.1 What Did We Achieve

We successfully designed and implemented a novel approach to designing a real-time pitch correction system via FPGA. We first looked back at the existing solutions and found the merits and drawbacks of them. We analyzed these solutions in detail and concluded that some of the obstacles can be bypassed while others can hardly be avoided. Studying from the past, we proposed two innovative designs, with one being the one-bit auto-cross correlation for pitch detection, and another being the ring-buffer pointer-jumping techniques for pitch shift. We then verified our concept by performing various experiments, which proved the feasibility of our proposed approaches. Based on that, we then designed and integrated an FPGA based system using the proposed approach. In the end, we tested the system and the results confirmed our concepts. Thus, we concluded that the one-bit auto-cross correlation is an efficient pitch detection method that can significantly alleviate the computation pressure without compromising accuracy, and the jump-based-on-pitch method for controlling the pointers in a ring buffer is efficient and effective in a real-time context.

5.2 What Could be Done

We pointed out that any TD approach for pitch detection will suffer from a lack of frequency resolution as the pitch goes higher. In our design, we simply set a minimal detectable fundamental period instead of solving the issue. Can we use any techniques that help compensate for this deficiency so we can expand the detectable region?

We also implemented a rather naive approach to interpolate a signal when resampling is required. Such an approach cannot generate a result on-par with other more complicated approaches. We didn't use the complicated approaches since an efficient computing method with the least amount of power consumption is our first priority. Nonetheless, does there exist

an interpolation method that can be both energy efficient but also more accurate?

During the informal listening test, we found that our system, although had been given several preventive measures, can still produce clipping noises in special circumstances. How should we further improve our design in order to make our design more robust?

These questions, though remain unsolved for now, is what we aim to break through in the future.

Appendix A

Scource code of the FSM designed for pitch detection

```
1  import chisel3._
2  import chisel3.util._
3
4  object pitch_detection_fsm_states {
5    object state extends ChiselEnum {
6      val idle_state,
7          calculate_state,
8          period_check,
9          period_log = Value
10   }
11 }
12
13 class pitch_detection_fsm(space_n: Int, search_size: Int,
     min_period: Int, threshold: Int) extends Module {
14     import pitch_detection_fsm_states.state
15     import pitch_detection_fsm_states.state._
16
17     // Parameters
18     val ring_size       = math.pow(2, space_n).intValue // ring
           buffer size, not really used
19     val search_size_bit = log2Ceil(search_size)
20     val min_init        = math.pow(2, search_size_bit).intValue
          - 1
21     val start_location  = search_size // in matlab this is
           search_size + 1
22     ///////////////////////////////////////////////////////////
23
24     val io = IO(new Bundle {
25         val data_in  = Input(UInt(1.W)) // 1-bit correlation
26         val write    = Input(Bool())
27         val period   = Output(UInt(search_size_bit.W))
28     })
29
30     // Registers
31     val state_reg          = RegInit(idle_state)
```

```
32      val input_ptr          = RegInit(0.U(space_n.W)) // 0 ~ 2^ring_size-1
33      val read_ptr           = RegInit(0.U(space_n.W)) // 0 ~ 2^ring_size-1
34      val calc_cnt           = RegInit(0.U(space_n.W)) // 0 ~ search_size, for correlation calc
35      val read_cnt           = RegInit(0.U(space_n.W)) // 0 ~ search_size, for window size check
36      val fund_period        = RegInit(0.U(search_size_bit.W)) // 0 ~ search_size
37      val fund_period_final  = RegInit(0.U(search_size_bit.W)) // 0 ~ search_size
38      val current_max        = RegInit(0.U(search_size_bit.W)) // 0 ~ search_size
39      val current_min        = RegInit(min_init.U(search_size_bit.W)) // 0 ~ search_size
40      val prev_result        = RegInit(0.U(search_size_bit.W)) // 0 ~ search_size
41      val prev_increment     = RegInit(false.B) // false = descending, true = incrementing
42      val ready              = RegInit(false.B) // enough inputs in the ram
43      val temp_corr          = RegInit(0.U(search_size_bit.W)) // 0 ~ search_size
44      ////////////////////////////////////////////////////////
45
46      // ram related components & io
47      val curr_win_read_pos = read_ptr + calc_cnt; // corr poistion of the current window
48      val next_win_read_pos = input_ptr - search_size.U + calc_cnt // corr position of the next window
49      val shared_port_pos   = Mux(state_reg === idle_state, input_ptr, curr_win_read_pos) // the r&w port addr
50      val ring_pitch_detect = Module(new single_write_dual_read(space_n, 1)) // true dual-port ram
51      ring_pitch_detect.io.clock  := clock
52      ring_pitch_detect.io.we     := io.write
53      ring_pitch_detect.io.addr_a := shared_port_pos
54      ring_pitch_detect.io.addr_b := next_win_read_pos
55      ring_pitch_detect.io.din    := io.data_in
56      val curr_win_value          = ring_pitch_detect.io.dout_a
57      val next_win_value          = ring_pitch_detect.io.dout_b
58
59      ////////////////////////////////////////////////////////
60
61      // Wires (combinational logic)
62      val pitch_period      = read_cnt
63      val crossed_line      = pitch_period > min_period.U
64      val current_increment = temp_corr > prev_result
```

```
65
66      // fsm
67      switch (state_reg) {
68          is (idle_state) {
69              temp_corr := 0.U
70              ready := ready || (input_ptr === start_location.U)
71              when (io.write) {
72                  input_ptr := input_ptr + 1.U
73                  when (ready) {
74                      calc_cnt := 0.U
75                      state_reg := calculate_state
76                  } .otherwise {
77                      state_reg := idle_state
78                  }
79              } .otherwise {
80                  state_reg := idle_state
81              }
82          }
83
84          is (calculate_state) {
85              when (calc_cnt === 0.U) {
86                  calc_cnt := calc_cnt + 1.U
87                  state_reg := calculate_state
88              } .elsewhen (calc_cnt <= search_size.U) { // it
                    needs an extra cycle since SyncReadMem is one
                    cycle slower
89                  temp_corr := temp_corr + (curr_win_value &
                        next_win_value)
90                  calc_cnt := calc_cnt + 1.U
91                  state_reg := calculate_state
92              } .otherwise {
93                  calc_cnt := 0.U
94                  state_reg := period_check
95              }
96          }
97
98          is (period_check) {
99              when (pitch_period === 0.U) {
100                 prev_result := temp_corr
101             } .elsewhen(pitch_period === 1.U) {
102                 prev_increment := current_increment
103                 prev_result := temp_corr
104             } .otherwise {
105                 when ((!current_increment) && prev_increment) {
106                     when ((prev_result > current_max) &&
                            crossed_line) {
107                         current_max := prev_result + (0.U(4.W)
                                ## prev_result(9, 4))
108                         fund_period := pitch_period
```

```
109                     }
110                 }
111                 prev_increment := current_increment
112                 prev_result := temp_corr
113             }
114 
115             when (temp_corr < current_min) {
116                 current_min := temp_corr
117             }
118 
119             when (read_cnt < (search_size -1).U) {
120                 read_cnt := read_cnt + 1.U
121                 state_reg := idle_state
122             } .otherwise {
123                 read_cnt := 0.U
124                 state_reg := period_log
125             }
126         }
127 
128         is (period_log) {
129             read_ptr := read_ptr + search_size.U
130             when ((current_max - current_min) < threshold.U) {
131                 fund_period_final := 0.U
132             } .otherwise {
133                 fund_period_final := fund_period
134             }
135             current_max := 0.U
136             current_min := search_size.U
137             fund_period := 0.U
138             state_reg   := idle_state
139         }
140 
141     }
142 
143     io.period := fund_period_final
144 }
```

Appendix B

Scource code of the FSM designed for pitch shift

```
1  import chisel3._
2  import chisel3.util._
3
4  object pitch_shift_opt_states {
5    object state extends ChiselEnum {
6      val idle_state,
7          calculate_state,
8          distance_check = Value
9    }
10 }
11
12 class pitch_shift_opt(space_n: Int, search_size: Int) extends
       Module {
13     import pitch_shift_opt_states.state
14     import pitch_shift_opt_states.state._
15
16     // Parameters
17     val ring_size       = math.pow(2, space_n).intValue // ring
           buffer size, not really used
18     val search_size_bit = log2Ceil(search_size)
19     val start_location  = 2*search_size // in matlab this is 2*
           search_size + 1
20     ////////////////////////////////////////////////////////
21
22     val io = IO(new Bundle {
23         val data_in  = Input(UInt(16.W))  // audio data
24         val write    = Input(Bool())      // an incoming data is
               valid
25         val period   = Input(UInt(search_size_bit.W))
26         val way      = Input(Bool())     // true = upward jump,
             false = downward jump
27         val rate1    = Input(UInt(4.W))
28         val rate2    = Input(UInt(4.W))
29         val data_out = Output(UInt(16.W)) // output data
30         val out      = Output(Bool())     // output data is
             ready
```

```
31      })
32
33      // Registers
34      val state_reg   = RegInit(idle_state)
35      val input_ptr   = RegInit(0.U(space_n.W)) // 0 ~ 2^ring_size
            -1
36      val output_ptr  = RegInit(0.U(space_n.W)) // 0 ~ 2^ring_size
            -1
37      val work_rate   = RegInit(0.U(4.W))
38      val ready       = RegInit(false.B) // enough inputs in the
            ram
39      val out_reg     = RegInit(false.B) // drives io.out
40      val distance    = RegInit(0.U(space_n.W))
41      ////////////////////////////////////////////////////////
42
43      // ram related components & io
44      val ring_pitch_shift = Module(new single_write_single_read(
            space_n, 16)) // true dual-port ram
45      ring_pitch_shift.io.clock  := clock
46      ring_pitch_shift.io.we     := io.write
47      ring_pitch_shift.io.addr_w := input_ptr
48      ring_pitch_shift.io.addr_r := output_ptr
49      ring_pitch_shift.io.din    := io.data_in
50      io.data_out                := ring_pitch_shift.io.dout
51
52      ////////////////////////////////////////////////////////
53
54      // fsm
55      switch (state_reg) {
56          is (idle_state) {
57              ready := ready || (input_ptr === start_location.U)
58              when (io.write) {
59                  input_ptr := input_ptr + 1.U
60                  when (ready) {
61                      out_reg := true.B // output the data at
                            output_ptr
62                      state_reg := calculate_state
63                  } .otherwise {
64                      state_reg := idle_state
65                  }
66              } .otherwise {
67                  state_reg := idle_state
68              }
69          }
70
71          is (calculate_state) {
72              out_reg := false.B
73              when (((work_rate <= io.rate2) && (io.rate1 =/= 0.U)
                    )) {
```

```
74                output_ptr := output_ptr + (io.way ## false.B)
75            } .otherwise {
76                output_ptr := output_ptr + 1.U
77            }
78            when (work_rate <= 1.U) {
79                work_rate := io.rate1
80            } .otherwise {
81                work_rate := work_rate - 1.U
82            }
83            distance := input_ptr - output_ptr
84            state_reg := distance_check
85        }
86
87        is (distance_check) {
88            when (distance < search_size.U) { // pointers are
                  too close
89                output_ptr := output_ptr - io.period
90            } .elsewhen (distance > start_location.U) { //
                  pointers are too far
91                output_ptr := output_ptr + io.period
92            }
93            state_reg := idle_state
94        }
95
96    }
97
98    io.out := out_reg
99 }
```

References

[1] F. J. Harris, "On the use of windows for harmonic analysis with the discrete fourier transform," *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51–83, 1978.

[2] P. McLeod and G. Wyvill, "A smarter way to find pitch," in *ICMC*, vol. 5, 2005, pp. 138–141.

[3] A. De Cheveigné and H. Kawahara, "Yin, a fundamental frequency estimator for speech and music," *The Journal of the Acoustical Society of America*, vol. 111, no. 4, pp. 1917–1930, 2002.

[4] A. M. Noll, "Cepstrum pitch determination," *The journal of the acoustical society of America*, vol. 41, no. 2, pp. 293–309, 1967.

[5] ——, "Pitch determination of human speech by the harmonic product spectrum, the harmonic surn spectrum, and a maximum likelihood estimate," in *Symposium on Computer Processing in Communication, ed.*, vol. 19. University of Broodlyn Press, New York, 1970, pp. 779–797.

[6] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[7] R. Yavne, "An economical method for calculating the discrete fourier transform," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, 1968, pp. 115–125.

[8] L. Rabiner and R. Schafer, *Digital Processing of Speech Signals.* Englewood Cliffs: Prentice Hall, 1978.

[9] P. McLeod, "Fast, accurate pitch detection tools for music analysis," *Unpublished PhD thesis. Department of Computer Science, University of Otago*, 2008.

[10] E. W. Weisstein. Fourier series–square wave. [Online]. Available: https://mathworld.wolfram.com/FourierSeriesSquareWave.html

[11] J. L. Flanagan and R. M. Golden, "Phase vocoder," *Bell System Technical Journal*, vol. 45, no. 9, pp. 1493–1509, 1966.

[12] J. Laroche, "Time and pitch scale modification of audio signals," in *Applications of digital signal processing to audio and acoustics*. Springer, 2002, pp. 279–309.

[13] D. Rossum, "An analysis of pitch-shifting algorithms," in *Audio Engineering Society Convention 87*. Audio Engineering Society, 1989.

[14] F. Charpentier and E. Moulines, "Pitch-synchronous waveform processing techniques for text-to-speech synthesis using diphones." in *Eurospeech*, vol. 89, 1989, pp. 13–19.

[15] W. Verhelst and M. Roelands, "An overlap-add technique based on waveform similarity (wsola) for high quality time-scale modification of speech," in *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 2. IEEE, 1993, pp. 554–557.

[16] K. Lent, "An efficient method for pitch shifting digitally sampled sounds," *Computer Music Journal*, vol. 13, no. 4, pp. 65–71, 1989.

[17] S. Roucos and A. Wilgus, "High quality time-scale modification for speech," in *ICASSP'85. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 10. IEEE, 1985, pp. 493–496.

[18] U. Zölzer, X. Amatriain, D. Arfib, J. Bonada, G. De Poli, P. Dutilleux, G. Evangelista, F. Keiler, A. Loscos, D. Rocchesso *et al.*, *DAFX-Digital audio effects*. John Wiley & Sons, 2002.

[19] R. W. Schafer and L. R. Rabiner, "A digital signal processing approach to interpolation," *Proceedings of the IEEE*, vol. 61, no. 6, pp. 692–702, 1973.

[20] MathWorks. Resample uniform or nonuniform data to new fixed rate - matlab. [Online]. Available: https://www.mathworks.com/help/signal/ref/resample.html

[21] G. Fairbanks, W. Everitt, and R. Jaeger, "Method for time of frequency compression-expansion of speech," *Transactions of the IRE Professional Group on Audio*, vol. AU-2, no. 1, pp. 7–12, 1954.

[22] Y. Tu, S. Sadiq, Y. Tao, M.-L. Shyu, and S.-C. Chen, "A power efficient neural network implementation on heterogeneous fpga and gpu devices," in *2019 IEEE 20th international conference on information reuse and integration for data science (IRI)*. IEEE, 2019, pp. 193–199.

[23] C. Bao, T. Xie, W. Feng, L. Chang, and C. Yu, “A power-efficient optimizing framework fpga accelerator based on winograd for yolo,” *Ieee Access*, vol. 8, pp. 94 307–94 317, 2020.

[24] L. Morales-Velazquez and E. Guillén-García, “Fpga real-time fft portable core, design and implementation,” in *2018 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)*. IEEE, 2018, pp. 1–7.

[25] C. Yang and H. Chen, “A efficient design of a real-time fft architecture based on fpga,” 2013.