5.75-7.14 GHz All-Digital Phase-Locked Loop MATLAB Model with Novel Filter to DCO Frequency Decoder

by

Juan David Heredia

A Thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfilment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Carleton University
Ottawa, ON

© 2019, Juan David Heredia

Abstract

The work presented in this thesis provides a straightforward method for simulating an All-Digital Phase-Locked Loop. Previous research has been conducted using MATLAB's Simulink environment, however it is shown to be considerably slower than MATLAB's script environment. For this reason, an ADPLL MATLAB Model is presented and shown to provide comparable closed loop results with decreased simulation times. The low simulation times are crucial in initial circuit testing and debugging.

Through TIE (Time Interval Error) and modulation peak analysis it is shown that the ADPLL MATLAB model handles modulations of the reference and presents a correct lowpass-filter behavior. Most importantly, it can lock to all output frequencies allowed by the DCO with the exceptions incurred by the DCO decoder.

Time analysis was conducted after the model was observed to be slower than the Verilog behavioral model. The source of the ADPLL model's high simulation time stem from the attempt to model synthesizable-based code and the use of computationally taxing functions in the DCO decoder.

Acknowledgments

I would first like to express my deepest gratitude to Microsemi Co. in Kanata for their immense support in my work. On top of providing all the necessary design tools, Microsemi had excellent work environment and culture.

This thesis would not have been possible without the on-going support from my supervisor John Rogers. Not only was he available for technical and theoretical discussion regarding the thesis, but also helped immensely with personal matters that were not in his job description. Through his teaching and counseling, he taught me that good is not good enough and striving for excellence should be done in all aspects of life.

Out of the numerous people at Microsemi that played a role in this thesis, from IT help to phase noise discussions, Krste Mitric was pivotal in ensuring the success of this thesis. I would like to express my appreciation for his continuous help since day one. Even with all the responsibilities that come with being a senior engineer, Krste Mitric spared some of his valuable time to have theoretical and design discussion that were key in the completion of the thesis. His positive and professional attitude in the face of adversity is something to be noted.

Y las personas más importantes, mi familia. No hay palabras para decir que tanto me ayudaron a llegar a este punto de mi vida. Especialmente mi papá, Juan, and mamá, Yadira. Nineteen years ago, they left their lives in Colombia to move to Canada. They gifted me the opportunity to complete this thesis to satisfy the requirement for a MASc graduate degree in Electrical Engineering at Carleton University.

Abbreviations

ADPLL	All-Digital Phase-Locked Loop
APLL	Analog Phase-Locked Loop
CFN	Center Frequency Number
CMOS	Complementary Meta-Oxide Semiconductor
CPU	Central Processing Unit
DC	Direct Current
DCO	Digital Controlled Oscillator
FM	Frequency Modulation
GSM	Global System for Mobile Communication
IC	Integrated Circuit
IDE	Integrated Development Environment
IIR	Infinite Impulse Response
MWP	Multi-Wafer Process
OPLL	Optical Phase-Locked Loop
OTW	Oscillating Tuning Word
PFD	Phase-Frequency Detector
PIF	Proportional-Integral Filter
PLL	Phase-Locked Loop
QPSK	Quadrature Phase Shift Keying
RAM	Random Access Memory
RF	Radio Frequency
RLC	Resistor-Inductor-Capacitor
RRD	Reference Resample DCO Clock
RTC	Real Time Clock
Rx	Received Data
TDC	Time-to-Digital Converter
TIE	Time Interval Error
TSMC	Taiwan Semiconductor Manufacturing Company
Tx	Transmitted Data
VCO	Voltage Controlled Oscillator
XO	Crystal Oscillator

Table of Contents

Abstract	ii
Acknowledgements	iii
Abbreviations	iv
Table of Contents	v
List of Tables	viii
List of Figures	ix
List of Appendices	xii
1. Chapter 1: Introduction	1
1.1 Purpose.....	1
1.2 Phase	2
1.2.1 Definition	2
1.2.2 Phase-Locked Loop Behavior	4
1.3 Applications of PLLs	5
1.3.1 FM Demodulation.....	5
1.3.2 Clock Recovery.....	7
1.3.3 Summary of Thesis	10
2 Chapter 2: Analog vs. Digital Block Implementation.....	11
2.1 Phase Detector	11
2.1.1 Analog Implementation.....	11
2.1.2 Digital Implementation	13
2.2 Controlled Oscillator.....	16
2.2.1 Analog Implementation.....	17
2.2.2 Digital Implementation	19
2.3 Loop Filter	22
2.3.1 Analog Implementation.....	22
2.3.2 Digital Implementation	25
3 Chapter 3: Literature Review	28
3.1 Introduction.....	28
3.2 Review	28
3.2.1 Basic Simulation Models of Phase Tracking Devices Using MATLAB [18]	28

3.2.2	Time-Domain Modeling of an RF All-Digital PLL [19]	30
3.2.3	Nonlinear analysis of BPSK Optical Phase-Locked loops using MATLAB and Simulink [20]	31
3.3	Literature Conclusion.....	33
4	Chapter 4: All-Digital Phase-Locked Loop MATLAB Model.....	34
4.1	Introduction.....	34
4.2	Purpose and Motivation	34
4.2.1	MATLAB Code Advantage	34
4.2.2	Resolution and Speed of Simulation.....	35
4.3	Systematic Design Approach to MATLAB ADPLL Design	38
4.3.1	Introduction.....	38
4.3.2	ADPLL Verilog Base Model	39
4.3.3	Verilog Versus MATLAB Design Environments.....	46
4.3.4	ADPLL Block Implementation	53
4.4	ADPLL Model Simulation Results	67
4.4.1	Varying DCO Output Frequencies.....	68
4.4.2	Modulation Measurement Methodology.....	70
4.4.3	Varying Input Modulation Frequency for Static Corner Frequency	73
4.4.4	Varying pShift and its Effects on Modulation	78
5	Chapter 5: Simulation Time Analysis	84
5.1	Underutilizing MATLAB's Mathematical Prowess	84
5.2	Filter-to-DCO Decoder Logic.....	89
5.3	Conclusion.....	92
6	Chapter 6: Conclusion	93
6.1	Introduction.....	93
6.2	PLL Applications	93
6.3	Analog vs Digital Block Implementations	94
6.4	Literature Review.....	95
6.5	All-Digital Phase-Locked Loop.....	96
6.6	Simulation Time Analysis.....	98
6.7	Contributions.....	99
6.8	Future Work	100
6.8.1	Introduction.....	100

6.8.2	Model Improvements	100
6.8.3	Testing Improvements.....	107
7	Chapter 7: Executive Conclusion	109
8	Appendix.....	111
8.1	A: Simulink and MATLAB Simulation Speed Test Configurations	111
8.2	B: Corner frequency vs. Desired Frequency Behaviors.....	112
8.3	C: Two's Complement Binary to Decimal MATLAB Functions.....	114
9	Bibliography	116

List of Tables

Table 1: Phase Offset of Peak Pairs between.....	4
Table 2: DCO Capacitor Bank Description	20
Table 3: Verilog vs MATLAB bit-operation comparison.....	50
Table 4: MATLAB vs Verilog comparison of always block implementation.....	54
Table 5: MATLAB vs Verilog TDC Clock Implementation	55
Table 6: TDC Enable Counting Condition	56
Table 7: pValue Scale Enable Range	59
Table 8: DCO MATLAB Behavioral Model	61
Table 9: DCO TIE MATLAB Calculation	72
Table 10: Loop Corner Frequency and pShift relationship based on Verilog Simulation	73
Table 11: Lost Frequency Due to Flat Decoder Response.....	103

List of Figures

Figure 1: Negatively Phased Offset Sine Wave.....	2
Figure 2: Increasing Phase Offset between 5 MHz and 5.5 MHz Sinusoids	4
Figure 3: FM Modulation Example	5
Figure 4: FM Demodulation with PLL	6
Figure 5: Clock Recovery Behavior Example	7
Figure 6: Edge Detection Behavior.....	8
Figure 7: PLL Based Clock Recovery System.....	8
Figure 8: Phase Detector Schematic [9].....	12
Figure 9: Phase Detector Behavior where $f_{ref} > f_{osc}$	13
Figure 10: Phase Detector Sensitivity Behavior	13
Figure 11: TDC Basic Circuit	14
Figure 12: TDC output with DCO and Reference in Sync	15
Figure 13: TDC output with DCO early	15
Figure 14: TDC output with DCO late.....	16
Figure 15: Transfer Function Model of Oscillator.....	18
Figure 16: VCO Voltage-Frequency Relation	19
Figure 17: DCO Frequency Control Capacitor Bank.....	20
Figure 18: Percentage Error in DCO Output	21
Figure 19: Analog Loop Filter Configurations using a series resistor a) and a parallel capacitor b).....	23
Figure 20: Magnitude Bode Plots of RC (a) R-Series RC (b) and C-Parallel (c) loop filters.....	24
Figure 21 : Step response of RC Low Pass Filter	25
Figure 22: Measured and Simulated DCO Frequency Behavior	30
Figure 23: Simulink Model of OPLL.....	31
Figure 24: Simulink Environment and Simulation Time Results	37
Figure 25: Simulation-time analysis of 4000 MATLAB simulations (b) and Histogram (a) of results (0.02 sec bin widths)	38
Figure 26: Generating DCO Count from DCO Output.....	39
Figure 27: TDC refClkPulse and rrdClkPulse Generation Block Diagram	40
Figure 28: TDC Counting Period.....	40
Figure 29: TDC Count Generation.....	41

Figure 30: Loop Filter Verilog Implementation Schematic.....	42
Figure 31: Generating diffPhase in first three pipeline stages	42
Figure 32: FinalPhase Word Generation.....	43
Figure 33: Filter Integrator Model	44
Figure 34: DCO Divide by Two Implementation	45
Figure 35: Sequential (left) vs Parallel(right) execution example	47
Figure 36: Data transmission in a non-sequential system.....	47
Figure 37: Data transmission in sequential system.....	47
Figure 38: DCO and TDC Counting Behavior	56
Figure 39: Output DCO Count.....	57
Figure 40: Output TDC Count	57
Figure 41: DCO Count End Behavior.....	57
Figure 42: Filter to DCO Decoder Algorithm.....	61
Figure 43: DCO Decoder Output for all possible input values	64
Figure 44: Final Frequency value of full ADPLL Simulation of 6GHz DCO	65
Figure 45: Flat DCO Decoder Response.....	65
Figure 46: Second Instance of a Flat Decoder Response	66
Figure 47: ADPLL DCO Output Frequency at 5.825 GHz, 6 GHz, 6.5 GHz, 7 GHz.....	68
Figure 48: TIE rising edge behavior example.....	71
Figure 49: TIE of 600kHz Modulated DCO Clock.....	71
Figure 50: 600 kHz Modulation Clock	71
Figure 51: PSD Plot of DCO TIE with 50kHz Modulation.....	72
Figure 52: pShift 11 TIE Attenuation	75
Figure 53: pShift 14 TIE Attenuation	76
Figure 54: DCO Output between low 50 kHz (left) and 4 MHz (right) Modulation Frequencies.....	77
Figure 55: 5kHz Modulation TIE Attenuation Verilog Result	78
Figure 56: 50kHz Modulation TIE Attenuation Verilog Result	79
Figure 57: ADPLL TIE Attenuation for 50 kHz (top) 200 kHz (middle) and 600 kHz(bottom)	80
Figure 58: DCO (left) and REF (right) PSD with 600kHz modulation and pShift 11	81
Figure 59: Modulation Peak in DCO Output for Varying pShift Values.....	81
Figure 60: ADPLL DCO Peak Attenuation for 50 kHz (top) 200 kHz (middle) and 600 kHz(bottom)	82

Figure 61: Profiler Report of Three DCO Implementations for a 1us total time, 0.4ps resolution simulation.....	86
Figure 62: Simulation time comparison for 0.1 us, 1us, and 10 us simulations (semilog view)	87
Figure 63:Simulation time comparison for 0.1 us, 1us, and 10 us simulations (loglog view).....	87
Figure 64: Profiler Report for ADPLL 0.4 ps resolution, 1 us Simulation Time.....	89
Figure 65: Five most computationally heavy processes in Decoder.....	89
Figure 66: Decoder Simulation with round() functions enabled.....	91
Figure 67: Decoder Simulation with round() functions disabled.....	91
Figure 68: Second Order Loop Filter.....	101
Figure 69: 1st Order (left) vs 2nd Order (right) DCO Outputs	102
Figure 70: Five Largest Flat Decoder Responses	103
Figure 71: DCO Phase Noise Profile with Jitter Limits (12kHz-20MHz)	105
Figure 72: MATLAB Memory Error While Running Jitter Calculation	105
Figure 73: Phase Noise Profile of Successful Jitter Calculation.....	106
Figure 74: DCO Simulation with total time of 5us (left) and 0.02us (right).....	106
Figure 75: Cutoff Frequency with respect to changing desired frequency	112
Figure 76:Cutoff Frequency with respect to changing pShift.....	113
Figure 77: Cutoff Frequency with respect to changing tdc resolution.....	113

List of Appendices

A: Simulink and MATLAB Simulation Speed Test Configurations	111
B: Corner frequency vs. Desired Frequency Behaviors	112
C: Two's Complement Binary to Decimal MATLAB Functions	114

1. Chapter 1: Introduction

1.1 Purpose

The objective of this thesis is to present a MATLAB Behavioral Model for an ADPLL (All-Digital Phase-Locked Loop) comprised of an 0.8 ps resolution TDC (Time to Digital Converter), proportional integrating digital loop filter, 5.7-7.14GHz capacitor bank based DCO (Digitally Controlled Oscillator), and a decoder to interpret the filter output into usable data the DCO can utilize. The goal is to have this model run faster simulations than its Verilog counterpart.

The above mentioned ADPLL was designed and implemented using 40nm TSMC technology by Microsemi Co. The company saw value in the design and its potential in future research and development. This thesis stemmed from the exploration of that potential. The value of a model of the product is immense in that it can provide fast and accurate simulations. The goal is to have the model run simulations faster than the Verilog standalone behavioral models combined.

Models provide value through accurate prediction of behaviors without the physical implementation of the product or measurement in a lab setting. For instance, with Multi-Wafer Production (MPW) [1] the cost of masks required for production exceeds all other production steps immensely, reducing the cost effectiveness of implementation and experimentation. Specifically, at the beginning of the century, mask sets for IC (Integrated Circuit) designs were being reported to cost anywhere from \$1 million to \$2 million USD for 130-nm and 90-nm processes [2], respectively. And prices only increased as technology got better and better. In 2015 it was reported that the average IC design cost for a mid-range 14-nm process was \$80 million [3].

A MATLAB model will assist with observation of circuit behavior in its current state and be able to predict behavior when certain stresses are applied, such as input modulation or increased noise within components. It is possible to probe the behavior of the system once it is fabricated, but the specific probes need to be accounted for and designed. That process is common only during initial test chips. Once the final product is to be delivered there needs to be as little, if any, probes in the final design.

1.2 Phase

1.2.1 Definition

Phase is the displacement between two waves having the same frequency. This also extends to the relationship between a wave's original and shifted counterpart.

Figure 1: Negatively Phased Offset Sine Wave

Figure 1 demonstrates a $-\pi/2$ offset imposed on an original 5 MHz wave, resulting in the orange curve. The shift to the right is due to the negative phase offset.

$$y_1 = \text{sine}(2\pi ft) \quad (1.1)$$

$$y_2 = \text{sine}(2\pi ft + \phi) \quad (1.2)$$

When phase shift ϕ is imposed, the terms inside the bracket in Equation 1.2 react to maintain the original value before the shift seen in Equation 1.1. Three of the four other variables present, other than the shift, are constants leaving time as the variable that will compensate for the shift. Given a negative shift, the value inside the brackets will have a net negative value if all else remains constant. This means that time must become more positive to compensate, resulting in its occurrence later in time.

Another definition of phase is an “offset value of a waveform from another independent reference waveform”. This is akin to the previous definition if the phase offset orange curve in Figure 1 is a separate waveform. This is the phase definition that is taken advantage of in Phase-Locked Loops to lock one waveform to another.

1.2.2 Phase-Locked Loop Behavior

In general, the purpose of a phase-locked loop is to keep the phase difference of two signals constant, i.e. have one waveform's phase-locked or follow another waveform's phase. It is important to make the distinction that it is not attempting to make the phases the same, but rather, make the change in phase zero. Two waveforms can exist with a phase difference and if there are no other changes, the loop is fulfilling its duties.

However, when the phase change is not constant the loop must work to make the change zero.

Table 1: Phase Offset of Peak Pairs between 5 MHz and 5.5 MHz Sinusoids

Peak Pair	Phase Offset [rad]
1	$\frac{\pi}{22}$
2	$\frac{5\pi}{22}$
3	$\frac{9\pi}{22}$
4	$\frac{13\pi}{22}$

Figure 2: Increasing Phase Offset between 5 MHz and 5.5 MHz Sinusoids

Table 1 presents the phase difference between sequential peaks of two sinusoids, and the corresponding phase difference. Figure 2 illustrates the increasing phase offsets of the two signals. After the offset reaches π it immediately jumps to $-\pi$ and increases to a value to 2π .

The potential of the phase-locked loop comes from its modification of one waveform, which it generates as an output and loops back, to adjust the phase change down to zero. In doing so, it will have generated its output to be of the same frequency or a multiple of the input frequency.

For this thesis, we will be working with digital signals and the concept of phase change and difference. Instead of taking the phase difference at the peak or zero-crossing, it considers the phase difference of rising or falling edges, depending on the components used.

1.3 Applications of PLLs

The previous section discussed one of the more prevalent uses of a Phase-Locked Loop, which is frequency generation and stabilization. Below, two more possible uses for a basic Phase-Locked Loop Architecture will be shown.

1.3.1 FM Demodulation

Phase-Locked Loops can be utilized as a tool for FM demodulation. A FM modulated signal is information that is transformed into a higher frequency for transmission.

Figure 3: FM Modulation Example

Figure 4: FM Demodulation with PLL

As Figure 3 demonstrates, the information (black) can be modulated with the carrier signal (blue), resulting in a FM signal (red), that will be transmitted. At the receiver end of the transmission system is where the PLL can be utilized. The receiver needs the ability to extract the original message signal from the FM modulated signal, and this is possible through a direct output from the Loop Filter in the PLL system, as shown in Figure 4. This extraction is known as demodulation.

One advantage of using a PLL as a demodulator is that the output has a high degree of linearity [4]. The input variations in the FM signal are relatively small, which allows the PLL to respond linearly. The PLL can stabilize to the initial frequency in the FM modulated signal, and as the incoming signal changes, the PLL will detect the change and adjust the VCO to the new appropriate frequency. Afterwards, it will stabilize at the new frequency until another change

occurs in the input due to the modulation, and the process repeats. The input signal's demodulated counterpart can then be extracted as the Loop Filter adjusts the VCO signal to match the input.

PLLs as FM demodulators can be seen in a wide variety of applications. For instance, it can be used to measure the optical path difference in an interferometric experiment [5]; to serve as a high efficiency demodulator in combination with a RF to DC rectifier for the usage in biomedical implants [6]; and also, to demodulate the output of a piezo-resistive cantilever system in order to detect changes in humidity [7].

1.3.2 Clock Recovery

A PLL is a vital tool in communication systems to re-synchronize data after it has been transmitted. Often, data is created via a clock with the data transmitted independently. However, to use the data at the receiving end, it can be advantageous to recover the clock that was used to create the data originally.

Figure 5: Clock Recovery Behavior Example

To retrieve the clock, two circuit components are used; one of which is the PLL [8], and the other is an Edge Detector which will output a pulse each time a rising or falling edge is seen.

Figure 5 illustrates how the recovered clock would look like if the received data was put through a clock recovery circuit.

Figure 6: Edge Detection Behavior

If the received data in Figure 6 (Rx Data) is assumed to have some bitrate, say T_b , then the frequency of the incoming data is $f_{data} = \frac{1}{T_b}$ Hz. The Edge Detector does not differentiate between a rising or falling edge, indicating that the periodic frequency of the Edge detector will be $f_{ED} = \frac{2}{T_b}$ Hz, and that is the frequency the PLL will lock to, as show in Figure 7:

Figure 7: PLL Based Clock Recovery System

Because the clock that the VCO will generate is twice that of the recovered one from the Rx signal, a divider can be implemented to ensure the recovered signal is at the expected frequency.

One consideration is that it is possible for the incoming Rx signal to be a long sequence of '1's or '0's, in which case the Edge detector will detect no edges. This causes the VCO to drift from the desired $\frac{1}{T_b}$ Hz. To minimize this VCO frequency drift due to the absence of transitions, the Loop

Filters time constant τ_{LPF} can be designed to be much larger than the allowable interval between two consecutive transition [8], which allows the retention of information during longer periods of no information changes

1.3.3 Summary of Thesis

The thesis will begin by discussing how each PLL block is implemented in an analog and digital manner in Chapter 2. Chapter 3 consists of a literature review looking at current PLL research in order to draw inspiration for this thesis. Chapter 4 then discusses how the ADPLL Model was designed, differences between Verilog and MATLAB concerning this research are pointed out. The next chapter delves into how the environmental differences were applied in the implementation of standalone APLL blocks. The models are connected and by modulating the reference clock and changing the desired DCO frequency, the ADPL is shown to follow its Verilog predecessor but with slower simulation time. The lack of speed in the MATLAB simulation is investigated. Finally, limitations and future work are discussed before concluding the thesis.

2 Chapter 2: Analog vs. Digital Block Implementation

This section examines the architecture and functionality of an All-Digital Phase-Locked Loop (ADPLL) as it is the focus of the model for this thesis.

The PLL has three basic components; Phase Detector, Loop Filter, and Voltage Controlled Oscillator.

2.1 Phase Detector

The purpose of a phase detector is to provide the system with an error signal proportional to the phase error between the input reference signal and the feedback signal from the controlled oscillator.

$$v_e(s) = K_p \left(\theta_{ref}(s) - \theta_{out}(s) \right) \quad (2.3)$$

K_p is the constant that determines how fast or slow the error signal changes, this changed is cause by the phase difference between the reference, $\theta_{ref}(s)$, and the output, $\theta_{out}(s)$. For simple analog implementations of a PFD (Phase-Frequency Detector) this constant is said to be $\frac{I}{2\pi}$ [9], where I is the current that is being utilized in the charge pump.

2.1.1 Analog Implementation

A simple implementation of an analog phase detector utilizes the phase-frequency detector in combination with a charge pump. In this analog implementation, the PFD (Phase-Frequency

Detector) creates a signal that is proportional to the phase difference between the two signals and the two current sources are responsible for generating the error current signal.

Figure 8: Phase Detector Schematic [9]

As Figure 8 illustrates, the PFD itself is comprised of two flip flops that are reset by the signal multiplication (AND gate) of the UP and DN signal. The Reference is responsible for activating UP, and Oscillator is responsible for DN. If the Reference signal leads the Oscillator Output by δ seconds, then the UP will facilitate the current flow out of the charge pump for δ seconds, at which point the Oscillator edge arrives and all signals reset.

In the case where the reference signal is at a higher frequency than the oscillators output, it is expected that the oscillator will increase its frequency to match the reference. Thus, the Phase Detector would need to provide some positive current that is proportional to the error.

Figure 9: Phase Detector Behavior where $f_{ref} > f_{osc}$

The dotted blue line in Figure 9 represents how the error signal changes with respect to the phase error, and when plotted, the slope of the line would be $\frac{I}{2\pi}$ which is the phase detector sensitivity, K_p . Figure 10 below illustrates a detailed plot of how the error current increases and decreases with changes in phase.

Figure 10: Phase Detector Sensitivity Behavior

2.1.2 Digital Implementation

For digital versions of the phase detector, the component still needs to provide some information on the phase difference between the input reference and the oscillator output. A circuit known as a Time to Digital Converter (TDC) accomplishes this task. The difference between the input and

the oscillator output is defined by a certain set of bits as opposed to a current level in the analog case. Figure 11 is a simple TDC circuit for demonstrating the behavior:

Figure 11: TDC Basic Circuit

The delay due to a buffer, δ , determines how many stages the TDC requires. The total delay in all stages must be equal to, or more than, the period of the DCO (Digitally Controlled Oscillator), where N is the number of stages.

$$T_{DCO} > N\delta \quad (2.4)$$

For example, for an 8-bit TDC, where the oscillator and the reference are perfectly in sync, the TDC Output will be 00001111. The first cycle of the DCO signal will travel through the buffers and fill the outputs of the last four flip flops just as the reference signal will enable the flip flops.

Figure 12: TDC output with DCO and Reference in Sync

Therefore, ideally DCO and REF signals are in sync and that leads to the loop being locked. If the loop were not locked, the DCO will either be faster or slower (early or late), in comparison to REF. In such cases, the TDC output will reflect this discrepancy in its output with a different combination of bits. In the case where the DCO arrives early with respect to the REF, the TDC output will be 1000111 when clocking the flip flops, as shown in Figure 13. Similarly, the expected TDC output when the DCO is late or faster than REF, is 00011110, as shown in Figure 14.

Figure 13: TDC output with DCO early

Figure 14: TDC output with DCO late

This series of bits describes the phase relationship between the DCO and the REF the same way I_d describes the same relationship in the analog case. The circuit seen in Figure 11, is the simplest case and can be improved. For example, the TDC in Figure 13 and Figure 14 has a resolution of δ , which means that if DCO and REF are closer to each other than δ , then the TDC will not be able to see the difference within this time frame. One solution is to make δ smaller. Another more robust approach is to use a Vernier TDC, which utilizes a secondary delay line, changing the REF. The resolution of the TDC now becomes the difference of the two delay lines and can be made miniscule. This results in improved performance of the loop overall. [10].

2.2 Controlled Oscillator

In this project, the digital implementation of the controlled oscillator (DCO) was unique and presented interesting challenges in its integration with the Phase-Locked Loop System. For this reason, the DCO portion will be analyzed first, since the design of the next component, the loop filter, depended heavily on the input requirements of the DCO.

2.2.1 Analog Implementation

The VCO is a vital part of the Phase-Locked Loop system. It is the final element in the chain before feeding back and is the element that provides the output. In the analog case, the control voltage, or control current, is delivered from the loop filter into the VCO. With the example of a RLC Resonator, the oscillators behavior can be analyzed.

The RLC Resonator is comprised of an inductor, capacitor, and resistor. Given an input impulse, the system is seen to have a time domain response described by the equation [11] below:

$$v_{out}(t) = \frac{\sqrt{2}I_{input}e^{-\frac{t}{2RC}}}{C} \cos\left(\sqrt{\frac{1}{LC} - \frac{1}{4R^2C^2}} t\right) \quad (2.5)$$

As the oscillator approaches steady state, the final nominal operating frequency is seen to be:

$$\omega_{nom} = \sqrt{\frac{L}{C}} \quad (2.6)$$

However, with only a capacitor, inductor, and resistor, the effect of dampening will be too strong to maintain a stable oscillation. The loss of the actual components will drive the initial

oscillations down to zero. Thus, to maintain the oscillation, a “boost” at the input is required.

This boost can be simply taken from the output of the initial oscillation itself, creating a feedback path as seen below:

Figure 15: Transfer Function Model of Oscillator

$H_1(s)$ represents the transfer function of the previous RLC components, and it is known to converge to 0 if left unattended. Thus, the $H_2(s)$ transfer function is used to boost and maintain oscillation, specifically, this transfer function represents an amplifier that is fed back. For the system to have stable oscillations, it must abide by the Barkhausen criteria [12] which state that, given the system seen in Figure 15 above, it must follow the three criteria;

$$\frac{V_o(j\omega)}{V_i(j\omega)} = \frac{H_1(j\omega)}{1 - H_1(j\omega) H_2(j\omega)} \quad (2.7)$$

The following expression must be met:

$$|H_1(j\omega)| |H_2(j\omega)| = 1 \quad (2.8)$$

$$\angle H_1(j\omega) H_2(j\omega) = 2n\pi \quad (2.9)$$

Now that the nominal frequency has been established, the control signal from the loop filter will force the oscillator to deviate from ω_{nom} based on the value of the control signal. The amount by which it changes depends on the oscillator’s sensitivity, K_{vco} , which is dependent on the chosen design.

Figure 16: VCO Voltage-Frequency Relation

Ideally, K_{vco} would result in a straight upward slope (unlike Figure 16 above), and there would be no concerns about the variance of K_{vco} . However, actual circuits tend to behave similarly to Figure 16. For example, in a LC Tank CMOS VCO [13], the K_{vco} is neither negligible nor linear, and is dependent on more than one control factor as illustrated below:

$$K_{vco} = -\frac{1}{4\pi\sqrt{L}(C_v + C_{bank})^{1.5}} \cdot \frac{\partial C_v}{\partial V_{tune}} \quad (2.10)$$

Where C_{bank} is capacitance that coarsely controls the oscillation frequency, C_v tunes the frequency on a finer scale based on the voltage value of V_{tune} .

2.2.2 Digital Implementation

For a digitally implemented VCO, the digital aspect stems from how the oscillating output is controlled. In the previous section, it was shown that an analog control voltage drives the change in the frequency of the VCO. Equation 2.10 described how K_{vco} is not a linear function in actual circuits. In that case, the control variables that change the frequency of oscillation are C_{bank} , which is the total capacitance of a capacitor bank, and C_v , which is the capacitance of a varactor [13].

The capacitors in the capacitor bank must be switched on and off to determine the DCO's output. This is where a binary signal can be used to control the capacitors state. The DCO for the system in this thesis was designed by Shakeeb Abdullah [14], another Masters student, and has the following behavior:

Figure 17: DCO Frequency Control Capacitor Bank

The above DCO utilizes 6 capacitor banks consisting of 16 capacitors each (20 capacitors for the 5th bank as seen in Figure 17 above) to tune the output frequency. Each capacitor in each bank decreases the DCO frequency by a certain step size from its free running frequency.

Table 2: DCO Capacitor Bank Description

Bank	Frequency Step per Capacitor [Hz]	Bits	Bank Offset [Hz]
1	77.5 MHz	16	1.24 GHz
2	7.27 MHz	16	116.32 MHz
3	1.359375 MHz	16	21.75 MHz
4	351.25 kHz	16	5.62 MHz
5	21 kHz	20	420 kHz
6	6.875 kHz	16	110 kHz
		Max Offset [Hz]	1.38422 GHz

The DCO's frequency resolution is equal to the smallest step up or down it can take. Bank 6 is the last, and smallest frequency step bank, resulting in the DCO's resolution of 6.875 kHz.

Figure 18: Percentage Error in DCO Output

Determining the maximum offset the capacitor banks can achieve, 1.38422 GHz, and determining the DCO's free running frequency to be close to 7 GHz, the ideal percentage error can be modelled from the output of the DCO. From Figure 18, it is seen that the DCO's output varies from 9.8e-5 % to ~1.23e-4 %.

Because of the unique input parameters of the DCO, a decoder is needed to transform the binary loop filter output into something the DCO can interpret. This will be detailed in the following section.

2.3 Loop Filter

The main purpose of the loop filter, in this case a Low-Pass Filter, is to attenuate any high frequency noise or unwanted signals produced by the Phase Detector; and provide a clean translated current-to-voltage signal to the VCO [15]. The analog and digital implementation will vary significantly, as they did for the Phase Detector- TDC case. The digital loop filter design will be explored further in this section. In addition, the details of the decoder will be provided here and explain how the output of the filter is translated into the information the DCO banks need to set the output system frequency.

2.3.1 Analog Implementation

The simplest form of the analog loop is a passive RC Filter [16], with the transfer function:

$$F(j\omega) = \frac{1}{1 + j\omega\tau_1} \text{ where } \tau_1 = RC \quad (2.11)$$

But this provides limited performance since the only value that can be manipulated is the time constant, τ_1 . Creating an active filter, would result in adding an operational amplifier to provide gain. Without an active filter, two more options are available to improve the performance of the analog loop filter. Option one is to add a resistor in series [16], and option two is to add a second capacitor in parallel with the already existing RC combination.

Figure 19: Analog Loop Filter Configurations using a series resistor a) and a parallel capacitor b)

By adding a resistor in series (R_2), a second-time constant, τ_2 , is now present that allows for a more flexible performance in the filter, improving the PLL system.

$$F(j\omega) = \frac{1 + j\omega\tau_2}{1 + j\omega\tau_1} \quad (2.12)$$

where $\tau_1 = (R_1 + R_2)C$ $\tau_2 = R_2C$

This remains a first order transfer function. To further improve the performance of the filter, the parallel capacitor method can be implemented. Through analysis of the circuit, the resulting transfer function for the second option results in:

$$(j\omega) = \frac{j\omega RC_1 + 1}{j\omega(C_1 + C_2) - \omega^2 RC_1 C_2} \quad (2.13)$$

For comparison, the bode plots of equations (2.11,2.12,2.13) are shown below

a)

b)

c)

Figure 20: Magnitude Bode Plots of RC (a) R-Series RC (b) and C-Parallel (c) loop filters

As stated before, the simple RC-filter shown in Figure 20.a provides a simplistic low pass behavior with limited flexibility. The circuit containing the resistor in series can offer a different transfer function behavior if that is preferable. However, the attenuation is severe when compared to the other configurations. For this reason, a simple gain stage between the filter and VCO would be required [16], as seen in Figure 20.b. Figure 20.c shows the result of adding an additional capacitor in parallel to the conventional RC branch. An additional storage element pushes the order of the transfer function to be two, allowing for a second corner frequency. This

enables even more flexibility in adjusting the transfer response to accommodate the desired or optimal behavior when integrated in a PLL system.

2.3.2 Digital Implementation

The filter's principle roles for the loop are:

- To translate phase difference information into usable data for the oscillator
- To ensure the translated data is a consistent stream of information for the oscillator

On the analog side, the filter was responsible for translating the error current due to the phase difference, into a corresponding voltage the VCO could use. Due to the transfer characteristics of the low pass filter, any sudden increase or decrease in current were compensated by the filter. The filter protects the input of the VCO by smoothing out the output of the filter, even when the input was an impulse.

Figure 21 on the right shows the response of the RC Low Pass filter, described in Equation 2.11, to a step input. The response here mimics the projected VCO input - in this case, a

Figure 21 : Step response of RC Low Pass Filter

smooth upward change in magnitude instead of a step input itself, which would be problematic in the VCO. With this simple filter, the VCO would see the resulting value set by the step input close to two milliseconds after the initial change [17].

The digital filter then must provide a consistent response to fulfill its role in the loop. One key challenge when working with the digital domain is acknowledgement that discrete time is used rather than continuous.

To achieve a discrete time function in the z-domain, from the frequency domain, an intermediary step of going back into continuous time is ideal for the analysis. For example, in the frequency domain, a simple RC branch can be described as:

$$F(j\omega) = R + \frac{1}{j\omega C} \quad (2.14)$$

The R will remain constant in the translation from the s-domain to the z-domain, leaving variables of $\frac{1}{j\omega C}$ or $\frac{1}{sC}$. The definition of the z-transform of any continuous finite length signal is given by equation 2.15:

$$F(z) = \sum_{n=-\infty}^{\infty} f[n]z^{-n} \quad (2.15)$$

Before using the equation above, $\frac{1}{sC}$ needs to be in the continuous domain, which can be achieved by the inverse Laplace transform resulting in the step function, $u(t)$:

$$f(t) = \mathcal{L}^{-1} \left\{ \frac{1}{sC} \right\} = \frac{1}{C} u(t) \quad (2.16)$$

With this continuous function, equation 2.15 and equation 2.16 can be combined, and z-domain equivalent of $\frac{1}{sC}$ can be solved:

$$F(z) = \sum_{n=-\infty}^{\infty} \frac{1}{C} u[n] z^{-n}$$

$$\begin{aligned}
F(z) &= \frac{1}{C} \sum_{n=-\infty}^{\infty} u[n] z^{-n} \\
F(z) &= \frac{1}{C} \sum_{n=0}^{\infty} z^{-n} \\
F(z) &= \frac{1}{C} (1 + z^{-1} + z^{-2} + z^{-3} \dots z^{-n})
\end{aligned} \tag{2.17}$$

Equation 2.17 demonstrates a geometric progression, and can be simplified using the follow relationship:

$$S_{\infty} = \frac{a}{1 - r} \tag{2.18}$$

Where a is the first term in the series and r is the common ratio or term in the infinity series, S_{∞} :

$$\begin{aligned}
a &= 1 \\
r &= z^{-1}
\end{aligned}$$

Resulting in the final z-domain transform of $\frac{1}{sC}$:

$$F(z) = \frac{1}{C} \frac{1}{1 - z^{-1}} \tag{2.19}$$

Combining equation 2.14 with equation 2.19 and substituting $K_{prop} = R$ and $K_{int} = \frac{T_{ref}}{C}$ [11], the z-domain equivalent of the RC filter expression can be seen below:

$$F(z) = K_{prop} + \frac{K_{int}}{1 - z^{-1}} \tag{2.20}$$

3 Chapter 3: Literature Review

3.1 Introduction

Up until now, we have covered the general idea of what a phase-locked loop is and how it is expected to behave. Furthermore, two applications of how a phase-locked loop and its properties could be taken advantage of for more than simply locking onto a reference signal were discussed.

This chapter will investigate and inquire into the current research being conducted in the field of All-Digital Phase-Locked Loops.

3.2 Review

This section will investigate three papers concerning PLL models implemented in MATLAB. The reason is to gain a better understanding at what is being done in the field and apply it to this project.

3.2.1 Basic Simulation Models of Phase Tracking Devices Using MATLAB [18]

First it is explained how the PLL simulation is divided into three portions; the preprocessor, simulator, and postprocessor. The advantage for following this method is to make the MATLAB models more easily maintained and understood. The preprocessor is the first phase in their simulation hierarchy (followed by the simulator and the postprocessor), its purpose is to setup all simulation and system parameters. Simulation length, time resolution, and sampling frequency would be among the variables that are instantiated in the preprocessor. In addition, variables that might be used in the postprocessor can be defined. It is mentioned that the preprocessor is usually *interactive* speaking to the variable that pertain to the specific simulation being executed.

The simulator executes the behavioral code that is written. Taking the values that were instantiated in the preprocessor it generates the necessary data for analysis in the postprocessor. Postprocessing simply means taking the data generated by the simulator and creating graphical illustrations or subsequent intensive calculations.

A key advantage in splitting up the simulation into three portions is reusability. It is very common to have different simulation models of the same circuit, each with different adjustments. The preprocessing and post processing for all the different simulations could be the same in this case. Therefore, instead of instantiating the loop parameters ten different times for ten different simulations, they are instated once and then used by the ten different simulations. The same can be said for the postprocessor. Another example of how this can be useful is during verification. Perhaps instead of trying to debug or complete a model for one circuit, you might want to compare the results of several different circuits all using the same initial variables and postprocessing analysis.

The remainder of the chapter explores how the above-mentioned method is applied in a simple PLL, and a QPSK (Quadrature Phase Shift Keying) Loop.

3.2.2 Time-Domain Modeling of an RF All-Digital PLL [19]

The authors of this paper propose time-domain modeling and simulation techniques that are suitable for high-level languages such as MATLAB. Their model is based off an ADPLL that has the expected components; TDC, Phase Detector, Filter and DCO. It operates using an 8-40 MHz reference signal and has DCO frequency outputs within 1.6-2 GHz to abide by GSM standards.

The difference in frequency between the reference and the DCO raises the issue of the simulation resolution. In other words, the model is considering the ADPLL system as a single-rate system when that is not the case. This creates unavoidable frequency aliasing because the DCO signal is now being expressed with respect to the reference rate.

A method used to tackle this was to create clock timestamps for both the reference clock and the DCO clock. Afterwards, the model operates on an event-driven ideology using the timestamp data. This way the ADPLL model does not suffer from having to choose one value for the simulation time resolution. Figure 22 illustrates the ADPLL simulation settling in accordance with the measured results, proving the models efficacy.

Figure 22: Measured and Simulated DCO Frequency Behavior

3.2.3 Nonlinear analysis of BPSK Optical Phase-Locked loops using MATLAB and Simulink [20]

This team from China focused on building a MATLAB-Simulink model for an OPLL (Optical Phase-Locked Loop) and showing its BPSK demodulation capabilities. The paper discusses both an ordinary PLL system and a Costas Loop, for the purpose of this thesis the Costas loop will be ignored since it is not touched upon anywhere in the thesis. Figure 23 below shows the proposed Simulink model:

Figure 23: Simulink Model of OPLL

Much like any type of phase-locked loop, one of the key components is the information that describes the difference in phase between the reference and the systems output that is looped back. Here, the model is attempting to model the output of the photodetector using a 180 ° optical hybrid mixer. The output should mimic the following equation:

$$I_{PD} = |E_R|^2 + |E_L|^2 + 2|E_R||E_L|\cos(\Delta\theta(t)) \quad (3.21)$$

Not much detail is given on the error-current equation in the paper but reading the associated reference [21] does explain where some of the terms are come from.

Below is a summary from [22].

$$\Delta\theta(t) = 2\pi\Delta f t + \Delta\theta_0 \quad (3.22)$$

$$\Delta f = f_R - f_L \quad (3.23)$$

$$\Delta\theta_0 = \theta_R - \theta_L \quad (3.24)$$

As stated in the paper, the loop settles to a final output frequency of $\sim 100\text{GHz}$ with a local laser frequency of 200GHz (crystal oscillator equivalent). Frequencies of this range are not high considering it is an optical system.

Finally, the first order loop filter is modeled to have a cut off frequency of 66.314kHz and is proven in Figure 5 of the reviewed paper. The loop is seen to settle in 100ns based on plots shown. The OPLL model is not described to have any control system or programmability but it is assumed that the final output, divide by two, frequency can be changed by altering the frequency of the reference laser.

3.3 Literature Conclusion

The first review discusses how a MATLAB simulation of a PLL can be split up into three separate sections; preprocessing, simulating, and postprocessing. This was considered during the implementation of the ADPLL MATLAB model. Ultimately it was decided not to follow their method, all three phases (preprocessor, simulator, postprocessors) were lumped into one stage. Although it would make it more reader-friendly, there would need to be some sort of top program that would run the preprocessor, simulator, and post processor. This would be advantageous if I was working with multiple simulation models, but since this thesis is only concerning one model it was decided that it would not have considerable benefits. Running the top program would be the same as running one m-file containing all three parts.

Next the review explored a method that addressed the issue of having multiple frequencies in the ADPLL system and avoid aliasing while ensuring fast simulation times. Timestamping the reference clock and the DCO clock is a clever way at tackling this issue but still does not deal with the TDC's low time resolution of 15 ps. Only a quantized TDC time difference is obtained. Both the phase noise profile and the frequency settling behavior are promising and report a fast simulation time of 50 reference clock cycles (2 μ s) per 0.5 seconds of CPU time.

Ideally, the simulation time reported would have been compared against a Verilog counterpart on top of the SPICE and Specter RF simulators.

Finally, a Simulink OPLL model was analyzed for the purpose of seeing what could be achieved with Simulink. As expected, the model operates on a higher level than the previous two

which forgoes specificity for simplicity. This brought up the question of whether Simulink or MATLAB should be utilized for this project.

4 Chapter 4: All-Digital Phase-Locked Loop MATLAB Model

4.1 Introduction

This thesis presents an All-Digital Phase-Locked Loop, non-Simulink, MATLAB Model. Previous work [20], [21], and [22] have shown working DPLL models in a Simulink environment but none in a purely MATLAB IDE. Using MATLAB code proved to be imperative in modeling the very specific ADPLL that was fabricated. Even though the ADPLL has all three major loop components; TDC, Loop-Filter, and DCO, it is not easily-realizable using simple Simulink blocks. Designing the components from scratch allows for increased visibility into the model for minor tweaks, optimization, and debugging.

The goal of the MATLAB ADPLL model is to prove that it models the actual system as accurately as possible. To prove this, a modulation will be introduced into the reference clock and it is expected to be represented correctly in the DCO's output. Both location of spurs due to modulation frequency and amplitude of spurs due to filter characteristics will show the system model is sound.

4.2 Purpose and Motivation

4.2.1 MATLAB Code Advantage

The main reason as to why this Thesis project is being approached using a MATLAB simulation environment as opposed to Simulink is because of the complexity and intricacy of the components

being modeled. The loop filter is not just a simple IIR (Infinite Impulse Response) filter, and both the TDC and DCO are modified from their common digital implementations.

Because of this, the original model of the loop was created in Verilog. It would have been too time intensive to create a Simulink model at the detail and resolution that the Verilog simulations could achieve. The caveat was that Verilog simulations took a long time, this is where converting the model into MATLAB was advantageous. The simulation times are a lot shorter which translates to faster re-designing and testing of the ADPLL system.

4.2.2 Resolution and Speed of Simulation

In a later section discussing the corner frequency and resolution limitations it will be shown that we will need to have a frequency resolution in the kilohertz range to verify the loops corner frequency. This means that our simulation must be able to support and show resolutions of that scale.

A frequency resolution of 2kHz was initially decided upon. It was seen to be an adequate compromise between resolution, simulation time, and available memory. (Quick derivation based on [23]) A derivation is shown below of the simulation time based on the equations seen in [23] and a set time resolution (sampling rate) of 10ps. With a sampling rate of $\frac{1}{10ps} = 0.1 THz$, according to the Nyquist theorem, the signal can contain a maximum frequency of 50 GHz.

$$\begin{aligned} \frac{2kHz}{bin} &= \frac{50GHz}{X FFT Bins} \\ X FFT Bins &= \frac{50GHz}{2kHz/bin} \end{aligned} \tag{4.25}$$

$$\begin{aligned} X FFT Bins &= 25\,000\,000 \\ numSamples &= 2 \cdot 25M FFT Bins \\ numSamples &= 50 M \end{aligned} \tag{4.26}$$

$$simulation\ time = \frac{numSamples}{sampling\ rate}$$

$$simulation\ time = \frac{50\ M}{0.1\ THz}$$

$$simulation\ time = 500\ \mu s$$

Knowing that a total simulation time of 500 μs is required to achieve the desired frequency resolution, a comparison test was done. This simple test was conducted to see the speed of simulating a simulation space of 500 μs , containing 50M samples. Two simulation spaces were created, one built with Simulink and the other in a MATLAB environment. In both tests, a sinewave was instantiated, and graphed using the respective plotting tools. The purpose of the test was to see which simulation environment provided the fastest simulation time. Creating a sinewave, the simplest, most trivial task in the ADPLL model was chosen because the complexity of computation will only increase and the one that can complete this task the fastest will be the most suitable for the rest of the model.

Figure 24: Simulink Environment and Simulation Time Results

and verified by the *Diagnostic Viewer* available.

As mentioned previously, the same test is now replicated in a MATLAB simulation environment for comparison. We want to see if it is in fact faster to build the model in MATLAB over Simulink.

From Figure 24, Simulink is seen to give a high simulation time with an average simulation time of **4438.253** seconds. This is where we start to see the flaws of using Simulink with a time intensive simulation. A simulation surpassing one hour is not unheard off, but when the simulation is to simply plot a sinewave, that is not acceptable.

The data was gathered from the *Simulink Profile Report* after the simulation is finished

Figure 25: Simulation-time analysis of 4000 MATLAB simulations (b) and Histogram (a) of results (0.02 sec bin widths)

The code snippet of the simulation set up, excluding the data analysis plots, can be found in the Appendix A. It is setup so it performs the exact same actions as the Simulink simulation; set up the time environment, generate a sinewave, and plot that sinewave.

After running 4000 simulations, compared to the mere 10 of Simulink, the resulting above analysis shows that MATABL is far superior for time intensive simulations. With a calculated average simulation run time of only **2.077 seconds** and maximum time of 2.0832 seconds it was chosen as the desired IDE for this Thesis.

4.3 Systematic Design Approach to MATLAB ADPLL Design

4.3.1 Introduction

This section of the Thesis will begin by discussing the original Verilog model and how the ADPLL block functions were represented. Then the approach taken in moving the Verilog model into the MATABL environment will be discussed. Afterwards, simulations of the individual MATLAB blocks will be discussed ending with the Decoder that translates the loop filter data into usable data for the DCO.

4.3.2 ADPLL Verilog Base Model

4.3.2.1 TDC

The base behavioral model was built by Tony Wang [24] a PhD student and Krste Mitric a senior engineer at Microsemi. It used the advantages of Verilog to test the system and continue with the chip design for layout and fabrication. The behavior of the TDC is described in Figure 26, Figure 27, and Figure 29 below. The reason for creating the block diagrams from the Verilog models is to help in the understanding of the blocks, as I was not the original author, and to aid on the conversion to the MATLAB environment.

Figure 26: Generating DCO Count from DCO Output

The TDC behavioral block outputs two different values to the rest of the system, *dcoCnt*, and *tdcCnt*. Both values combined attempt to describe how many DCO cycles are needed to describe one reference cycle using an integer portion and a fractional portion. Figure 26 above shows how the integer portion is generated and how the output, *dcoCnt*, is designed to follow the equation below:

$$dcoCnt = \frac{T_{ref}}{T_{dcoDiv2}} = \frac{f_{dcoDiv2}}{f_{ref}} \quad (4.27)$$

The following parts of the TDC behavioral model rely on an internal highspeed TDC clock which will be used to generate the fractional value of the DCO cycles. An 0.8ps clock is realized in Verilog which means that the MATLAB Code must provide a 1.25 THz clock. Resulting in a $tdcCnt$ equation of:

$$tdcCnt = \frac{T_{dcoDiv2}}{tdc_{Res}} = \frac{f_{tdc}}{f_{dcoDiv2}} \quad (4.28)$$

Figure 27: TDC $refClkPulse$ and $rrdClkPulse$ Generation Block Diagram

Next part of the Verilog model is tasked with generating a pulse with width of 0.8ps whenever a rising edge in $refClk$ and $rrdClk$ (*reference-resample dco clock*) is detected. These pulses are the boundaries for which the TDC will count using its fine resolution; recall that $rrdClk$ and $refClk$ are only separated by a divided-by-two DCO signal. The reason for

Figure 28: TDC Counting Period

using a divided DCO signal is so that we can use that same signal to gauge the time of one period and count throughout its entirety. Figure 28

helps show the counting range over which the TDC would be enabled and count.

The final part ties all previous components together, it utilizes the pulses measured to signal the start and stop counting range for one DCO period. Figure 29 below reflects how the Verilog model takes the two boundaries and counts the entire DCO period in 0.8 ps step sizes.

Figure 29: TDC Count Generation

4.3.2.2 Loop Filter

For this thesis, not all the options available in Verilog were translated into MATLAB. For instance, most notably, the loop filter is only in 1st order mode. On top of that, the holdover modes, and direct DCO control modes were not implemented as they were not pertinent in showing the base behavior of the filter and are more geared towards the goal of fabrication.

The filter takes in both outputs from the TDC and generates a *FinalFrequency* word that is decoded and used to drive the DCO.

Figure 30: Loop Filter Verilog Implementation Schematic

The filter implementation although not the focus of the thesis, is important to understand to be able to convert it into MATLAB. The filter Verilog implementation relies on functionalities that MATLAB simply does not support, it is important to understand that in its entirety in order to duplicate the behavior in MATLAB.

The filter is implemented using pipeline signals to ensure that certain actions occur before others and the system operates as shown in Figure 30. The following figures show how the TDC inputs are converted to a value suitable for DCO operation.

Figure 31: Generating diffPhase in first three pipeline stages

The result of Figure 31, *diffPhase*, is the difference between the reference signal and the DCO signal all converted to the TDC counting domain. Now both the reference and the *dcoCount* are in terms of the 0.8ps resolution TDC clock.

The next stages take this phase difference information and passes it through. This is where the second order filter behavior is implemented, but as mentioned previously, only 1st order operation will be considered and *diffPhase* is simply passed through.

Figure 32: FinalPhase Word Generation

Up until now, all the variables have been strings of signed bits. Verilog has the advantage of easily working with signed bit-information, later this will be discussed as one of the major differences between MATLAB and Verilog. The *finalDiffPhase* information is encoded in a 24-bit signed number, which then is shifted by *pShift*. This value is what sets the corner frequency of the loop according to the following equations.

$$f_c = 2^{pShift} \cdot \frac{f_{desired} - f_{dcoMin}}{f_{dcoMax} - f_{dcoMin}} \cdot \frac{2^{outFreqW}}{tdc_{res}} \cdot \frac{10^{-12}}{2\pi} \text{ Hz} \quad (4.29)$$

Where:

- $6 \leq pShift \leq 30$
- $dcoMax = 7.14 \text{ GHz}$
- $dcoMin = 5.75 \text{ GHz}$
- $dcoMin \leq dcoDesired \leq dcoMax$
- $outFreqW = 31$

The reason the 10^{-12} constant is included was because the tdc_{res} is assumed to be in pico-seconds, to get the final cut-off frequency in hertz it needs to be converted into seconds. Appendix B shows how the cut-off frequency reacts with a varying pShift, desired frequency, and TDC resolution. However, for this Thesis, the TDC resolution will be kept at 0.8 ps and the desired frequency was chosen to be 6GHz.

Because the allotted max shift value is 32 bits, *finalDiffPhase* is first sign extended by the maximum shift value and then the shift is performed.

Figure 33: Filter Integrator Model

This stage of the pipeline integrates the *pValue*. Recall that integration is simply the summation of a value during a certain range.

Since time is running automatically in the simulation, to integrate, a simple feedback summing loop can be used. The pValue is added to the sum to perform the $\frac{1}{s}$ operation. The running count is divided by 2^Δ where:

$$\Delta = INT_{FREQW} + IT_{SHIFT} - 2 \cdot pShift \quad (4.30)$$

The new divided value, iValue, is combined with the pValue to be processed by the decoder and used by the DCO.

4.3.2.3 DCO

The DCO's main purpose is to generate a time domain signal. A similar approach was taken as when the internal TDC clock was generated. The main difference is that the frequency of oscillation is based upon how many banks are enabled. Looking back at the “Digital Implementation of the DCO” section, the equation that described the DCO frequency is shown below:

$$dco_{freq} = freq_{maxOffset} - \begin{bmatrix} capsInBank1 \\ \vdots \\ capsInBank6 \end{bmatrix}^T \begin{bmatrix} Bank1Freq \\ \vdots \\ Bank6Freq \end{bmatrix} + dco_{minFreq} \quad (4.31)$$

When all capacitors are turned on it will negate $freq_{maxOffset}$, where the maximum offset is set to 1.38422 GHz, and the resulting DCO frequency will be the minimum frequency of 5.75 GHz. On the other hand, if no capacitors are enabled the resulting DCO frequency will be ~7.14 GHz. On top of generating the DCO clock, it uses the DCO clock to create a divided by two version of itself using the inverting-feedback method shown in Figure 34:

Figure 34: DCO Divide by Two Implementation

4.3.3 Verilog Versus MATLAB Design Environments

In this section the differences between designing a system in Verilog and MATLAB will be discussed and how these challenges were tackled. There are three major differences that were encountered when going from Verilog to MATLAB:

- Verilog's ability for parallel execution
- Verilog's time environment
- Verilog's power of defining and working with binary data

4.3.3.1 *Sequential vs Parallel Programing*

When it comes to the sequence of all execution possibilities, Verilog holds the upper hand. Two executions can occur one after the other or at the same time. The advantage of being able to executive simultaneously is that it allows the ability to better reflect real-life behavior. Take three flip-flops in parallel, each output connected to the next one's input (all being clocked by the same reference). This circuit will have all the inputs transferred to the outputs at each clock edge (falling or rising). The behavior expected would be that the input signal propagates along the chain of flip flops:

Figure 35: Sequential (left) vs Parallel(right) execution example

To properly model this behavior, we must consider the origin of each variable value.

Figure 37: Data transmission in sequential system

Figure 36: Data transmission in a non-sequential system

In the sequential system, **A** will look at the value in the input at the clock edge and change to that value. Since **B** is waiting for **A** to change then perform its own execution it will take on the **same** value that **A** just acquired. In MATLAB this was modeled as follows (left):

<pre>% Sequential Chain a(ii) = input(ii); b(ii) = a(ii) ;</pre>	<pre>% Non-Sequential Chain a(ii) = input(ii-1); b(ii) = a(ii-1) ;</pre>
---	--

On the other hand, for the non-sequential behavior (right), at each clock edge the next value took on the value of the input right before the edge. This results in the desired behavior for the simple flip-flop system. To finish the implementation, non-clock edge times must be considered. Since they are flip-flops, it holds the value until the next clock edge:

```
if(~refPosEdge)
    a(ii) = a(ii-1);
    b(ii) = b(ii-1);
end
```

The above MATLAB code is using indexed variables to perform the task of modeling sequential flip-flops as opposed to how Verilog would have implemented it:

% Sequential Chain	% Non-Sequential Chain
A = input;	A <= input;
B = A ;	B <= A ;

And in Verilog we do not need to take care of saving the value because the variable can be defined as a *reg* and it will hold its value until a new one is given. Notice that Verilog does not need to index the variables like in MATLAB, this is because of the way the time environment differs.

4.3.3.2 Time Environment

One of Verilog's considerable advantage over other programming languages is its ability to define a timescale for the entire simulation. This is not to be confused with the ability to get the time through functions which C, Python, and MATLAB can accomplish as well. The timescale in Verilog is set by stating what the *reference time* and the *time resolution* will be:

```
`timescale 1us/1ns
```

Each component in the simulation will be stepping with the same time resolution and each specified time interval, using the # notation, will step by the same reference time. Afterwards, there's no need to worry about time, in terms on syntax.

One of the powers of MATLAB is how versatile it can be. It can perform tasks which Verilog cannot; frequency domain analysis, control systems ...etc. but to perform some tasks, because of its broad-spectrum of capabilities, it takes more work and attention to detail.

Simulating logic circuits is no exception, and to be able to model Verilog's time behavior one needs to really understand what is going on in a time simulation. Then it can be described with logic.

To model time in MATLAB, one needs to create time. Take the example of a 1 μ s simulation that steps at a time resolution of 1ps. In Verilog, the only action needed is to create a time scale:

```
`timescale 1ns/1ps
```

Then run a simulation with a total run time of 1 μ s. With MATLAB, the creation of time involves creating an array that will count from zero to the total run time in time steps defined by some resolution. To create the same time environment as the above Verilog code, the MATLAB code required is:

```
tstep    = 1*ps      ;  
tstop    = 1*us-tstep ;  
time     = 0:tstep:tstop ;
```

Unlike Verilog, everything that occurs in time must be referencing the time variable. How the use of time in the ADPLL model will be discussed in a further section.

4.3.3.3 Handling Verilog Bitwise Operations in MATLAB

4.3.3.3.1 Most Significant Bit

The first aspect that needed to be addressed was where the most significant bit is in Verilog and how to translate that in MATLAB. In Verilog, the most significant bit is always the left most bit and naturally in MATLAB it would be preferred to keep the same convention. A 16-bit binary value, for example, in Verilog would be instantiated as:

```
reg [7:0] num;
```

In MATLAB that same variable would be instantiated as:

```
num = zeros(1,8));
```

MATLAB has no knowledge of which bit is assumed to be the most significant bit or least significant bit in an array of values, because it only sees it as an array. Thus, for MATLAB we need to agree upon which bit will be the MSB and define the arrays knowing that MATLAB arrays start at an index of 1.

Suppose we want to now create two numbers and perform some arithmetic, or Boolean operations. Let's assume we want to add the number 3 and 4, then see if the summation is greater than the one of the individual values. If not handled correctly in MATLAB it will result in an incorrect value.

Table 3: Verilog vs MATLAB bit-operation comparison

Verilog Implementation	MATLAB Translation
reg [2:0] num1; reg [2:0] num2;	num1 = zeros(1,3)); num2 = zeros(1,3));

<pre> num1 = 3'b001; num2 = 3'b011; sum = num1 + num2; greater = sum > num1 ; </pre>	<pre> rsb = 'right-msb'; num1 = [1 0 0]; num2 = [1 1 0]; sum = bi2de(num1,rsb) + bi2de(num2,rsb); greater = bi2de(sum) > bi2de(num1) ; sum = dec2bi(sum,rsb); </pre>
--	---

Verilog is intelligent enough to take the binary bits and perform both the addition and comparison without much thought in the programming. On the other hand, in MATLAB it was decided that to model the same behavior the most significant bit would be the right most bit, 100 would translate to a decimal value of 1, for example, not 4. The bits would be converted into their decimal counter parts. The reason the convention of right most significant bit was chosen was so that the bit number would only be one numerical value away from the index that is describing it. If the left most significant bit would have been chosen, the third bit would be indexed by '1' and the discrepancy would get worse for larger array sizes.

As the reason the bit-array was converted to decimal was because of the way MATLAB would see these bit-arrays. MATLAB **only** sees them as arrays of integer values and performing

the same operations on the arrays as was done for the bits in Verilog would result in unwanted and unusable results:

$$[1\ 0\ 0] + [1\ 1\ 0] = [2\ 1\ 0]$$

$$[2\ 1\ 0] > [1\ 0\ 0] = [1\ 1\ 0]$$

When the desired series of operations was:

$$[1\ 0\ 0] + [1\ 1\ 0] = [0\ 0\ 1]$$

$$[2\ 1\ 0] > [1\ 0\ 0] = 1 \text{ or } true$$

4.3.3.3.2 Signed Binary Numbers

Because of the nature of the ADPLL system, it is possible for the phase error data to be negative. As such, the binary numbers need to be able to reflect this. Two's complement method is used in both Verilog and MATLAB to deal with this, with the caveat that in MATLAB a separate function was needed to convert negative numbers into two's complement and vice versa. The functions can be found in Appendix C and do take advantage of the MATLAB built-in decimal to binary functions.

To take an example from the ADPLL system. This Verilog bit shift and addition of binary values is done in one line:

```
deltaFreq <= (pValue>>(P_VALUE_W - DELTA_FREQ_W - MAX_P_SHIFT + pShift)) +
iValue;
```

Whereas in MATLAB the operation is done in the decimal domain. Then converted into a binary value, but since the value can be negative the two's complement function is needed to start using bits again.

```
deltaFreqDec(current) = iValueDec(past) + (bitsra(pValueDec(past), P_VALUE_W -
DELTA_FREQ_W - MAX_P_SHIFT + pShift));
    ⋮
deltaFreq(current,:) = twosDe2Bi(deltaFreqDec(current),rsb,DELTA_FREQ_W);
```

4.3.4 ADPLL Block Implementation

With respect to the previously discussed differences between Verilog and MATLAB, this section will touch upon considering those differences and implementing the Verilog behavior in MATLAB. This will not go in depth, as one would in a code review, but major components of the blocks will be discussed.

Overall, the goal was to create a model as close to Verilog as possible. This means that certain coding methods were kept when they might not have been needed. One example is modeling the *Always Block* in MATLAB, this meant declaring and instantiating variables that kept track of the rising edges of the variables in the sensitivity list. Below is the MATLAB vs Verilog implementation of an always block triggering on the rising edge of the master clock:

Table 4: MATLAB vs Verilog comparison of always block implementation

MATLAB	Verilog
<pre> current = ii; past = ii-1; mClkPosEdge = mClk(current) > mClk(past); : if(mClkPosEdge) mClkCnt(current) = mClkCnt(past) + 1; else mClkCnt(current) = mClkCnt(past); end </pre>	<pre> always @(posedge mclk) if (~rst_b) mclkCnt <= {MCLK_CNT_W{1'b0}}; else mclkCnt <= mclkCnt + 'd1; </pre>

4.3.4.1 TDC

Two outputs of the TDC are crucial in the ADPLL system operation; the DCO Count and the TDC Count. The first is a count of how many DCO periods are in one period of the reference-resampled clock, and the second counts how many TDC clock periods are in one DCO period.

Firstly, the highspeed TDC clock needs to be created and is where one of the largest MATLAB vs Verilog differences are encountered. To create a clock in Verilog, the signal needs to be manually set high and set low at the appropriate times. This is implemented using a *forever* loop in conjunction with the period of the signal. The same Verilog loop structure can be simplified to a single line in MATLAB because of MATLAB's mathematical superiority. With the use of

two simple functions, first *sin* to create a periodic waveform, and second *sign* to square off the signal, the digital clock is created.

On top of code simplicity, MATLAB requires less computing power. To use the clock in the Verilog simulation, it requires the section of code responsible for generation to be constantly running. Whereas in MATLAB, once the clock is created it does not require any runtime resources unless it is called.

Table 5: MATLAB vs Verilog TDC Clock Implementation

MATLAB	Verilog
<code>tdc_Clk = A*sign(sin(2*pi*(1/tdcRes)*time)) + DC ;</code>	<pre> Initial begin tdcClk <= 0; #1_000_000 forever begin #(TDC_RES_PER_FS/2); tdcClk <= 0; if (TDC_RES_PER_FS/2 + TDC_RES_PER_FS/2 < TDC_RES_PER_FS) #(TDC_RES_PER_FS/2+1); else #(TDC_RES_PER_FS/2); tdcClk <= 1; end end </pre>

To count, the model needs to be able to identify a range over where counting is valid. For *dcoCnt*, that range is every period or every rising clock edge and for *tdcCnt* it is the interval between the rising edges of *refClk* and *rrdClk* with *refClk* edge happening before *rrdClk*.

Figure 38: DCO and TDC Counting Behavior

As shown in Figure 38, both DCO counter and TDC counter are counting within the expected region. Resulting in the expected counts of 60 and 416, respectively. Notice the counting in the simulation only occurs while the enable is high. This is due to the conditions set for the enable signal seen in Table 6. It ensures that enable is high if and only if a *refClk* edge is seen before the *rrdClk* edge and counts until the latter occurs.

Table 6: TDC Enable Counting Condition

MATLAB Model
<pre> if(tdcClkPosEdge) if(refClkPulse(ii) && rrdClkPulse(ii)) countEn(current) = 0; elseif(refClkPulse(ii)) countEn(current) = 1; elseif(rrdClkPulse(ii)) countEn(current) = 0; else countEn(current) = countEn(past); end else countEn(current) = countEn(past); end </pre>

The counting is stored in a temporary variable as to not make the output of the TDC have a sawtooth like behavior. The output will be updated only after the count finishes. This translates in the model as the TDC output will be passed from the temporary count once the *rrdClk* rising edge occurs.

When the TDC has a 50MHz reference clock signal and a 6GHz DCO as its inputs, according to equations 4.27 and 4.28 the TDC module is expected to output a value of 416 and 60 for *tdcCnt* and *dcoCnt*, respectively.

Figure 39: Output DCO Count

Figure 40: Output TDC Count

Figure 41: DCO Count End Behavior

Figure 40 and Figure 39 illustrate that the TDC module outputs desired values given the input frequencies. After the 5 μ s point in the simulation, both outputs are seen to stabilize. Note that both outputs stabilize at a rounded down value of the expected value. With the *dcoCnt* it only reaches 59 most of the time, because once the 60th

cycle is reached, the previous value is stored and instead of counting to 60, the value is reset.

4.3.4.2 Filter

In MATLAB the loop filter was the component that could be the most simplified yet still retain the behavior seen in Verilog. As a physical component, whether analog or digitally implemented, the filter is a sequential component, in the sense that one action must happen before the other.

In Verilog, sequencing was handled through the use of pipeline signals. This was needed because the filter blocks were all being synchronized to the same master clock but needed to execute in a sequential manner. The MATLAB environment allows for the same sequential behavior merely by placing one execution after the other.

Firstly, the *dcoCnt* and the 50MHz clock reference were converted into *tdcClk* counts in order for the filter to provide the most accurate phase difference information as possible. The filter begins by performing the task of a conventional phase detector by calculating the following equation:

$$diffPhaseIn = refPerInTdcRes - dcoCntInTdcRes + tdcCnt \quad (4.32)$$

Phase difference can be obtained from this equation because the variance in the feedback DCO signal is being represented by *tdcCnt*. *refPerInTdcRes* is the baseline value, and for a reference of 50MHz and 0.8ps resolution that value results in 25 000. Then the *dcoCnt* must be converted in terms of TDC counts. This is accomplished by multiplying the *dcoCnt* with the number of TDC periods that are in one DCO period, assuming an ideal 6GHz DCO, this value equates to (rounded down) $\left(\frac{2}{6GHz} * \frac{1}{0.8ps}\right) = 416$. It should be noted that the value used for the DCO frequency in the calculation is the desired frequency which is a static variable, **not** the DCO frequency of the running simulation.

With the *dcoCnt* at an ideal 60, *dcoCntInTdcRes* is the multiplication of both those variables which results close to *refPerInTdcRes* at 24 960. The *tdcCnt* is then added to provide the phase difference information, the count reflects how many TDC periods are in one DCO period of the running DCO, unlike the set desired value used previously.

For the 1st order filter case, *diffPhaseIn* is passed through to the input of the only filtering section. It proceeds to enter the two paths of the filter, the proportional and the integral paths.

The Verilog model had limiting factors for both sets of paths; the proportional path would only be scaled if it is not at the minimum or maximum values, and the integrator would only be enabled if the value in that path was within a certain range.

Table 7: pValue Scale Enable Range

MATLAB Model
<pre> if(mClkPosEdge) % check logic again if((1 < finalDiffPhase) (finalDiffPhase < finalDiffPhaseMax)) pValue = finalDiffPhase * 2^pShift; else pValue = finalDiffPhase; end end end </pre>

finalDiffPhase was a 21 bit unsigned binary number meaning that it ranges from 0 to 2^{21} , anywhere between those extremes is where *finalDiffPhase* will be scaled up by 2^{pShift} and set as the proportional path's value.

In the integral path *finalDiffPhase* is summed up into an *integrator* variable every master clock rising edge for the duration of the simulation. Afterwards, the resulting *integrator* value is scaled down with respect to *pShift*:

$$iValue = \frac{integrator}{2^{33-2 \cdot pShift}} \quad (4.33)$$

The Verilog model had limits on whether to enable or disable the integrator due to the limitations imposed by the bit-length of each variable. The MATLAB model abided by these limitations as well to preserve as much of the behavior as possible. But MATLAB did not need to have this limitation. For example, a variable can have a value of four or four million without changing the state of the variable, whereas a three-bit variable in Verilog containing the value of four cannot represent four million without further attention.

Finally, the two values describing the frequency variation are combined and added to a Center Frequency Number (CFN). The CFN is a carefully chosen value based on the following equation:

$$CFN = desired_{freq} - MIN_{Freq} \quad (4.34)$$

How the decoder converts the filters output (CFN, pValue and iValue) into a DCO frequency between the two limits is explained in the next section.

4.3.4.3 Decoder and DCO

The Decoder and DCO are an integral part in the model. In the end, the ultimate goal is to measure the desired DCO frequency from the DCO output. For the DCO MATLAB model, the same approach was taken as was with the TDC clock. No need to generate delays and set the clock low and high, the following line described the DCO in its entirety:

Table 8: DCO MATLAB Behavioral Model

MATLAB Model
$dco_Clk(current) = A * \sin(2 * \pi * dcoFreq * time(current)) + DC ;$

The challenge is getting the correct *dcoFreq* every iteration. The decoder preforms this task with the following algorithm:

Figure 42: Filter to DCO Decoder Algorithm

Initially, $finalFreq$ is subtracted from MAX_FREQ to insure the frequency is pushed in the correct direction. This is because of the DCO behavior, as capacitors are turned on, the DCO frequency will be reduced. Starting at a nominal frequency of 7.14 GHz, it can be reduced to $7.14 \text{ GHz} - MAX_FREQ$. Thus, $finalFreq$ has the information of how much to reduce the reduction by to get the 7.14GHz down to the desired output frequency.

The difference shows how much frequency the proceeding bank needs to handle, in this case, Bank 1 will always see the entire frequency push. Since Bank 1 will not be able to give the exact desired frequency due to the bank resolution, the next step is to see how much of the desired frequency the bank can accurately handle. To do so, the frequency into the bank is converted into a number with respect to the resolution of the bank:

$$inFreqShiftBank\Delta = \log_2(Bank\Delta_{res}) \quad (4.35)$$

$$relvntInFreqBank\Delta = \frac{inFreqBank\Delta}{2^{inFreqShiftBank\Delta}} \quad (4.36)$$

$$inFreqBank\Delta_{\delta} = \frac{\delta(Bank\Delta_{res})}{2^{inFreqShiftBank\Delta}} \quad (4.37)$$

Where Δ is the DCO banks available, ranging from one to seven. And δ is how many capacitors are enabled in the bank.

The two numbers were divided down by the corresponding shift value and compared. Starting with the highest number of capacitors enabled, as soon as the value going into the bank is

larger than the amount the capacitors can supply, that number of capacitors is chosen. Now, it is known how many capacitors are enabled in the bank and the amount it will push down the frequency ($freqBank\Delta$).

Although, there will still be some frequency that needs to be considered. As the resolution of each bank will not allow for an exact match. The most accurate match will result in an error equaling the resolution of the smaller bank. To calculate how much frequency needs to be considered in the subsequent bank:

$$inFreqBank\Delta = inFreqBank(\Delta - 1) - freqBank(\Delta - 1) \quad (4.38)$$

The difference from equation 4.38 results in how much frequency the previous bank missed due to its limiting resolution. This process is repeated until all banks have provided the necessary frequency to achieve the closest DCO frequency possible with the given final frequency input number.

Figure 43: DCO Decoder Output for all possible input values

Illustrated above is a standalone simulation of the DCO Decoder, all possible input values (zero to 1.38422 GHz) were put through the algorithm in 100 Hz steps and the resulting behavior is what was expected. As final frequency increases the DCO is pushed to its maximum frequency of ~7.14 GHz.

The majority of the ADPLL model testing was done with a desired DCO frequency of 6 GHz. This frequency was chosen because it fell within the DCO's operating range and was not affect by a Decoder issue that will be explained later. During the testing it was seen (Figure 44) that the final frequency number would have very minute fluctuations.

Figure 44: Final Frequency value of full ADPLL Simulation of 6GHz DCO

Seen in Figure 44, the Final Frequency number was seen to decrease by 100kHz throughout a 1 μ s simulation. This however, was not reflected in the DCO output frequency. Throughout the same simulation, the DCO frequency was seen to stay at a constant 6GHz.

To discover where this inconsistency was originating from, a closer look at the decoder behavior around the desired frequency was taken.

Figure 45: Flat DCO Decoder Response

Upon closer inspection, Figure 45 explains where the DCO behavior is coming from and shows that in fact it spans 200 kHz of frequency. This is not the only flat response seen in the

decoder simulation, close to the 2.44225×10^8 Final Frequency value (Figure 45) there is an even smaller flat response. This inconsistency pattern is repeated and seen throughout the simulation.

Figure 46: Second Instance of a Flat Decoder Response

able to settle to frequencies in between 6.907 GHz and 6.922 GHz but because of the Decoder behavior this is not possible.

The most pronounced instance of the flat decoder response is shown in Figure 46. Previously, the flat response spanned 200 kHz but now it was seen to span 14 MHz. This is control that the filter loses and a gap in DCO frequency is created. In other words, the DCO should be

4.4 ADPLL Model Simulation Results

After connecting all three components, the ADPLL Loop was realized in MATLAB. To verify the loop functionality, three different experiments were conducted, and the resulting loop analysis illustrates the effect of each of said experiments, those include:

- Changing the desired output frequency (from 5.74 GHz to 7.14 GHz)
- Enabling and changing the input modulation set on the reference clock
- Changing pShift, which sets the cutoff frequency of the loop filter.

4.4.1 Varying DCO Output Frequencies

The model was able to adapt and produce the desired frequency set at the beginning of the simulation. Figure 47 shows the result of changing the desired DCO frequency to 5.825 GHz, 6 GHz, 6.5 GHz, and 7 GHz. These frequencies were chosen as test points because they resided in the operation range of the DCO and did not happen to be affected by the decoder issue.

Figure 47: ADPLL DCO Output Frequency at 5.825 GHz, 6 GHz, 6.5 GHz, 7 GHz

This is achieved by changing the CFN described in Equation 4.34. To summarize how this is possible, the following analysis will begin with the origin of the DCO frequency and work

backwards to show how the CFN alters the frequency. Ultimately the DCO frequency is defined by how much it is pushed away from its maximum frequency of 7.14 GHz. That is the case when no capacitors are enabled by the decoder:

$$DCO_{freq} = 7.14 \text{ GHz} - inFreqBank1 \quad (4.39)$$

inFreqBank1 is how much the DCO will be pushed way from the maximum, this value is defined in the decoder (Figure 42):

$$inFreqBank1 = MAX_{Freq} - finalFrequency \quad (4.40)$$

finalFrequency is comprised of CFN and Delta Frequency (coming from the filter). After settling, delta frequency is going to be negligible in comparison to the CFN, thus it can be assumed that:

$$finalFrequency = CFN \text{ (steady state)} \quad (4.41)$$

Combining the last three equations with the equation describing CFN, it can be shown that the DCO frequency will result in the desired set frequency:

$$DCO_{freq} = 7.14 \text{ GHz} - (MAX_{Freq} - finalFrequency)$$

$$DCO_{freq} = 7.14 \text{ GHz} - (MAX_{Freq} - CFN)$$

$$DCO_{freq} = 7.14 \text{ GHz} - (Max_{Freq} - \{desired_{freq} - Min_{freq}\})$$

$$DCO_{freq} = 7.14 \text{ GHz} - (Max_{Freq} + Min_{freq}) + desired_{freq}$$

Note that *Max_{Freq}* described the maximum amount the DCO can be pushed away from its maximum value of 7.14 GHz, and *Min_{freq}* is the lowest possible DCO frequency if the maximum

push is exerted. Meaning the addition of these values is 7.14 GHz. Resulting in a DCO frequency of:

$$DCO_{freq} = desired_{freq} \quad (4.42)$$

4.4.2 Modulation Measurement Methodology

To compare the effects of changing the modulation frequency and pShift, the modulation peak in the DCO PSD was analyzed. Another metric used in the comparison was the TIE (Time Interval Error) of the modulation. In order to measure the TIE of the modulation, a pure DCO signal with no modulation was measured and stored. Then, only enabling the modulation, the same process of running the ADPLL and saving the DCO signal was conducted.

A separate script took these two datasets and created TIE data. To attain the TIE data, the script ran through time stopping at each rising edge of the pure signal. The time of the rising edge was stored. The same procedure was done in parallel for the signal that was being affected by the modulation.

The difference of these two times was stored. This was the time interval error data that was being sought after.

Figure 48: TIE rising edge behavior example

Illustrated in Figure 48 is a possible snapshot of the pure DCO clock and the modulated clock. The rising edges of the modulated DCO clock will shift between being ahead of the pure edge and behind the pure edge. The way the rising edge shifts about the respective edge over time will be sinusoidal and this very sinusoidal behavior is the original modulation.

Figure 49: TIE of 600kHz Modulated DCO Clock

Figure 50: 600 kHz Modulation Clock

Notice the period in Figure 49 is the same as the period of the original modulating clock seen in Figure 50. In addition, the peaks of the TIE data are in their expected location. When the

modulating clock is at its respective peak, it will result in the modulated signal to lag behind the unmodulated signal. Thus, when the TIE is calculated it will result in a negative value because the pure clock rising edge is being subtracted from the modulated rising edge.

Table 9: DCO TIE MATLAB Calculation

DCO TIE Calculation	
if	(pure_dcoNegEdge)
	dco_TIE(n) = mod_dco_time - pure_dco_time;
	else
	dco_TIE(n) = dco_TIE(n-1);
end	

The interest when comparing varying modulation frequencies and pShifts is the frequency of modulation and amplitude, therefore, the TIE is converted into the frequency domain through an FFT operation. The PSD plots can now be analyzed, and the data is stored for further analysis. Figure 51 below shows how the modulation frequency and its amplitude can be easily gathered from the PSD plot of the DCO TIE.

Figure 51: PSD Plot of DCO TIE with 50kHz Modulation

The same was done for the reference signal, and TIE PSD plots for each corresponding test were taken. Ultimately, the peak in the Reference TIE PSD is compared to the peak of a divided down DCO TIE PSD to see how the modulation was affected by the loop. Because of amplitude comparison, the DCO was divided down to the same frequency as the reference.

4.4.3 Varying Input Modulation Frequency for Static Corner Frequency

The following two tests show how the ADPLL Model reacts to input modulations alterations and changes in pShift. As reference, the following table illustrates how the corner frequency is expected to change with respect to pShift.

Table 10: Loop Corner Frequency and pShift relationship based on Verilog Simulation

pShift	Corner Frequency [Hz]
0	33.59375
1	67.1875
2	134.375
3	268.75
4	537.5
5	1075
6	2150
7	4300
8	8600
9	17200
10	34400
11	68800
12	137600
13	275200
14	550400
15	1100800
16	2201600

The table was obtained from the designing engineer (Krstec Mitric) and the values are used to ensure that this model behaved like its Verilog counterpart.

The expected behavior from ADPLL is that a -3dB drop off occurs in accordance to Table 10. For a pShift fixed value, the peak representing the modulation in the TIE PSDs should remain constant until the corner frequency reached. After the modulations surpass the corner frequency, the amplitudes should decrease or its attenuation with respect to the reference TIE should increase.

First, the pShift value was set to 11, which translates to a corner frequency of 68.8kHz. Illustrated in Figure 52, the attenuation is seen to rapidly increase past the corner frequency and settle around 3 dB for the dataset analyzed.

Figure 52: pShift 11 TIE Attenuation

We cannot see much before a 50kHz modulation frequency due to simulation limitations. But it appears to be trending down to an attenuation of 0 dB which is what is expected before the corner frequency. After the corner, again, more data must be collected in order to draw a true conclusion on the expected 20dB attenuation slope.

The key behavior that was expected was to see a significant attenuation change through the corner frequency, which Figure 52 does illustrate.

A second test was conducted, now increasing the pShift to 14. It did not yield promising results.

Figure 53: pShift 14 TIE Attenuation

The expected increase in attenuation is seen across the corner frequency, but it was expected to increase from zero to 3 dB and continue at a 20-dB rate. Both of which are not represented in Figure 53.

Figure 54: DCO Output between low 50 kHz (left) and 4 MHz (right) Modulation Frequencies

A possible reason for this inaccurate result might be that the modulation frequency is affecting the DCO too much. This makes the measurement of the TIE an inaccurate representation of the true peak amplitude. Figure 54 presents two DCO PSD plots, one for a 50 kHz modulation and another for a 4 MHz modulation. The modulation is having a significant effect on the DCO's PSD.

4.4.4 Varying pShift and its Effects on Modulation

For a fixed modulation frequency, the peak representing the modulation in the TIE PSDs should increase until the corner frequency is reached. After, the modulation is in the passband of the filter as pShift has moved the corner past the fixed modulation. In terms of attenuation, it should decrease until the corner and remain at an ideal value of zero. Figure 55 below was created using the results from the Verilog simulation the designing engineer provided. As one can see, the attenuation reaches 3 dB in the middle-left of the F_c range as 5 kHz is closer to a corner frequency of 4.5 kHz than 8.6 kHz.

Figure 55: 5kHz Modulation TIE Attenuation Verilog Result

Another plot, this time with a modulation frequency of 50 kHz was created. Seen below, it follows the expected behavior for varying pShifts.

Figure 56: 50kHz Modulation TIE Attenuation Verilog Result

For the ADPLL Model, three different modulation frequencies were swept with pShift values (1 -16). The results obtained were less conclusive than the Verilog attenuation plots (Figure 55, Figure 56) in showing the correct filter behavior. First, 50k kHz modulation was tested at all pShift values.

Figure 57: ADPLL TIE Attenuation for 50 kHz (top) 200 kHz (middle) and 600 kHz (bottom)

The behavior seen in the three plots on the left are the opposite of what is expected. Beginning with zero attenuation for low pShift values, this indicates that the modulation is in the passband of the filter. After the corner frequency the attenuation begins to increase, most notably in Figure 57 top and middle plots.

Promisingly the behavior/attenuation change does occur within the acceptable range. Attenuation behavior of the three plots is more indicative of the loop filter being a high pass filter instead of a low pass filter.

In that case, the modulation would indeed begin in the passband of the filter. And as it approached the corner, attenuations would begin to be noticed and then drastically increase as the corner was passed. Which is what is seen in the figures to the left.

When these results were obtained, they brought up a question of whether the filter was designed as a lowpass filter. In the previous section, the results did show lowpass behavior, therefore more testing was conducted. For the

same set of simulations, instead of taking the difference of the reference TIE and the DCO TIE to get the resulting attenuation, the modulation peak in the reference PSD and the Divided-DCO PSD were compared and an attenuation from these two values was collected.

Figure 58 below is a PSD plot of the divided down DCO signal that had a 600kHz modulation and pShift of 11.

Figure 58: DCO (left) and REF (right) PSD with 600kHz modulation and pShift 11

The data-points in each of the plots show the frequency and amplitude of the modulation.

From these the attenuation was calculated. Even before the attenuation calculation was done for the three sets of simulations, the modulation peaks in the DCO PSD were already showing lowpass behavior for varying pShift values. Figure 59 illustrates the expected lowpass modulation peak behavior from the same simulations as the TIE plots before.

Figure 59: Modulation Peak in DCO Output for Varying pShift Values

Figure 60: ADPLL DCO Peak Attenuation for 50 kHz (top) 200 kHz (middle) and 600 kHz(bottom)

Figure 60 on the left gives a good representation of lowpass behavior. With the exception of the 50-kHz plot, the attenuation reaches close to the desired 3 dB value. A modulation of 600kHz was the most accurate in this regard. An attenuation value of 4.145 dB was found for a pShift of 14 which translates to 550.4 kHz corner frequency. Because this is slightly before the modulation frequency of 600kHz it can be assumed, based on the trend, that 600 kHz would be closer to a 3-dB value. On the other hand, although the 200-kHz case does also report an attenuation value of 4.395 dB, this is for a pShift value past the modulation frequency of 200 kHz. Meaning that, based on the plot, 200 kHz would show slightly higher attenuations above 4.395 dB and in the opposite direction of the ideal 3 dB attenuation.

It is interesting that the TIE measurements did not show a lowpass behavior when the pShift value was swept for static modulation frequencies. But the same ADPLL MATLAB Model did show lowpass behavior for static pShift values looking at the TIE data and for static modulations but only looking at the non-TIE data.

Perhaps in the testing of sweeping the pShift values and gathering the TIE data some aspect of the experiment might have been faulty. All other experiments give a good representation of lowpass behavior except this one.

5 Chapter 5: Simulation Time Analysis

The main objective of the thesis was to produce an ADPLL model in MATLAB with faster simulation times over the Verilog behavioral model. This was not achieved with the initial version of the model. This section will discuss two possible reasons why this was not accomplished and proposes future work to improve simulation time.

5.1 Underutilizing MATLAB's Mathematical Prowess

MATLAB is known for its high processing capabilities [25][26]. Therefore, it was a surprise that the MATLAB ADPLL model did not perform as expected. After discussing the issue with Krste Mitric from Microsemi, it was concluded that the model was not employing all of MATLAB's mathematical strengths.

The Verilog behavioral model was designed with synthesis in mind. Meaning, that simulation efficiency was not at the forefront of the design ideology. Thus, when a translation into MATLAB was decided, the code meant for synthesis was translated it into a language that excels in mathematical operations but did not utilizing them.

Both the TDC and the Filter did have reductions in logic, this included replacing bit shifts and 2's complement with multiplication, division, and the use of integer numbers. Even so, it was evident that the overall design of the blocks had synthesis as their original motivation. On the other hand, the DCO's implementation was reduced to a single mathematical expression from its Verilog synthesis implementation.

Table 11: MATLAB DCO Mathematical and Synthesis Implementations

MATLAB Mathematical Model	MATLAB Synthesis Model
<pre> % Loop Implementation for ii =1:length(time) dcoClk_math(ii) = A*sign(sin(2*pi*dcoFreq*time(ii))) + DC ; end % Single Execution Implementation dcoClk_math = A*sign(sin(2*pi*dcoFreq*time)) + DC ; </pre>	<pre> for ii =2:length(time) if (counter1 < delayHalfDcoPer) % waiting ... counter1 = counter1 + 1; dcoClk(ii) = dcoClk(ii-1); else % dcoClk(ii) = 0; temp = delayHalfDcoPer; if(counter2 < temp) % waiting ... counter2 = counter2 + 1; else counter1 = 0; counter2 = 0; dcoClk(ii) = 1; end end end end </pre>

Table 11 above illustrates the how the DCO was implemented using one single mathematical statement (left-top) vs. a more direct Verilog translation (right). A third model (left-bottom) was also implemented. The third implementation is completed in one execution as opposed to using a for-loop.

The time difference between the three implementations was gauged with the *Profiler* tool in MATLAB.

Figure 61: Profiler Report of Three DCO Implementations for a 1us total time, 0.4ps resolution simulation

The tool (Figure 61) reports the number of times each line of code is executed (blue), as well how much total CPU time was spent on that line (red).

Three separate simulations were conducted with simulation lengths of 0.1 μs , 10 μs , and 100 μs , each with a time resolution of 0.4 ps. Figure 62 and Figure 63 below illustrate the result of these simulations.

Figure 62: Simulation time comparison for 0.1 μs , 1 μs , and 10 μs simulations (semilog view)

Figure 63: Simulation time comparison for 0.1 μs , 1 μs , and 10 μs simulations (loglog view)

As expected, because of MATLAB's mathematical power, the Single-Line DCO implementation proved to be the most time efficient when compare to the Verilog-Synthesis and For-Loop implementations. Table 12 below is a numerical report of the previous results seen in Figure 62 and Figure 63.

Table 12: Simulation Time Results of DCO Verilog-Synthesis, For-Loop, and Single-Line Implementations

Total Simulation Time (μs)	Verilog-Synthesis Method (seconds)	For-Loop DCO Method (seconds)	Single-Line Method (seconds)
0.1	0.065	0.025	0.007
1	0.471	0.155	0.064
10	3.808	1.91	0.653
100	48.026	46.356	37.477

Looking at Table 13 below, the For-Loop implementation initially had only a 10.46 percentage point difference with the Single-Line implementation. Large simulation times resulted in a rapid decrease in effectiveness of the For-Loop implementation, reporting only a 3.48% decrease in simulation time. The Single-Line implementation still proved to be effective with large simulation times. A 15.67 percentage point increase is reported in simulation time decrease from the 100 μ s Verilog-Synthesis to For-Loop simulations.

Table 13: Percentage Decrease in Simulation Time of DCO Implementations

Total Simulation Time (μs)	Verilog-Synthesis to For-Loop Method Simulation Time Decrease (%)	For-Loop to Single-Line Method Simulation Time Decrease (%)
0.1	61.54	72
1	67.1	58.71
10	49.82	65.81
100	3.48	19.15

5.2 Filter-to-DCO Decoder Logic

The Filter-to-DCO Decoder was also found to hinder simulation time. Using the MATLAB *Profiler* tool, it was discovered that most of the simulation time was spent in the Decoder.

Profile Summary

Generated 20-Sep-2019 11:40:32 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
ADPLL_v0_0_9_top	1	162.399 s	23.640 s	
ADPLL_v0_0_5_decoder	2499999	137.876 s	137.876 s	
phaseNoisePlot	1	0.888 s	0.266 s	
fft_pwr	1	0.841 s	0.776 s	

Figure 64: Profiler Report for ADPLL 0.4 ps resolution, 1 us Simulation Time

Figure 64 reports that 84.89% of the total simulation time was spent processing the Decoder. Further investigation into the Decoder found that the *round()* MATLAB function is utilizing most of the computational resources. There are seven instances, one for each capacitor bank.

Lines where the most time was spent

Line Number	Code	Calls	Total Time	% Time	Time Plot
440	end	2499999	8.024 s	5.8%	
8	relevantInFreqBank1 = round(in...	2499999	6.555 s	4.8%	
300	relevantInFreqBank6 = round(in...	2499999	4.339 s	3.1%	
232	relevantInFreqBank5 = round(in...	2499999	4.275 s	3.1%	
356	relevantInFreqBank7 = round(in...	2499999	4.146 s	3.0%	
All other lines			110.538 s	80.2%	
Totals			137.876 s	100%	

Figure 65: Five most computationally heavy processes in Decoder

The *end* command at the end of the Decoder function took up most of the computational time. At the end of a function, MATLAB requires time to clear the workspace and exit the function properly, this time is what is reported beside the *end* command in the *Profiler* tool [27].

The next logical step was to see how much time was spent in the Decoder when the *round()* functions were removed. Using the same *Profiler* tool, removing the *round()* function reported a significant improvement in simulation time. Below is a summary of the results:

Table 14: Summary of Decoder Analysis with and without round() function

Bank Number	Simulation Time on <i>round()</i> Operation (seconds)	Simulation Time with <i>round()</i> Removed (seconds)	Time Decrease by Removing Round (seconds)	Percentage Decrease (%)
1	20.92	7.508	13.412	64.11089866
2	15.139	5.511	9.628	63.5973314
3	15.021	5.39	9.631	64.116903
4	15.102	5.584	9.518	63.02476493
5	15.539	5.347	10.192	65.58980629
6	14.51	5.451	9.059	62.43280496
7	15.399	5.363	10.036	65.17306319

By removing the *round()* functions, a decrease of 62-65% in simulation time was reported.

Figure 66: Decoder Simulation with `round()` functions enabled

Figure 67: Decoder Simulation with `round()` functions disabled

Looking at Figure 66 and Figure 67 it is clear that excluding the `round()` function affects the Decoder's ability to produce an accurate frequency number the DCO can use.

5.3 Conclusion

The MATLAB ADPLL Model was not able to compete against the Verilog simulation in terms of total simulation time. There are two major aspects of the ADPLL Model that hinder the simulation speed. First, the model is attempting to implement a Verilog model that was designed with synthesis in mind. Second, the use of the *round()* function within the Decoder was found to be much more computationally taxing than initially anticipated.

To improve simulation time in the future, the ADPLL needs to be redesigned with a heavy focus on a mathematical model over a synthesis-based model. This means using mathematical expression to model the TDC and the Filter instead of attempting to model the paths binary values would take.

Finally, the Decoder implementation needs to be improved. Research must be done such that the *round()* function can be removed without sacrificing the integrity of the Decoder's output.

6 Chapter 6: Conclusion

6.1 Introduction

The first chapter began with stating the objective of the thesis, to produce a MATLAB model for the fabricated ADPLL with faster simulation times over a Verilog behavioral model. Verilog was able to produce simulation results but still needed MATLAB to analyze the results, which is why it was advantageous to implement an entire model in MATLAB.

Next was a brief discussion on what phase is and the basic PLL operation. It was important to have a clear idea of what phase was at the beginning because all forthcoming analysis assumed this understanding. The main underlying idea behind phase locking in PLLs is that the system must be able to detect a change in phase and reduce this change to zero. This does not mean that a phase offset cannot exist, it can if the change in phase is non-existent.

6.2 PLL Applications

This chapter discussed two other possible applications PLLs have on top of the more common frequency multiplication and synchronization applications.

The essence of FM demodulation stems from the idea that the error signal of the loop filter describes the modulation imposed on the input signal. Instead of focusing on the output of the DCO, the output of the loop filter is converted to the system's primary output.

Clock recovery was the last application that was analyzed. It is common for data to be sent for transmission and not the clock used to generate the data. If we want to ensure that the data is processed correctly, the original clock must be extracted and the reason why clock recovery is useful. An edge detector circuit, either independent or part of the phase detector, is implemented.

The PLL now locks onto the detected edges and produces an output signal. It must be noted that the output of the DCO is twice the clock frequency because both rising and falling edges are detected, resulting in the need for a divider.

6.3 Analog vs Digital Block Implementations

To realize an ADPLL Model from the Verilog behavioral model, it was important to have a good understanding of the origin of the digital PLL blocks. Adjusting between the two design behaviors was an important aspect in this thesis and knowing the underlying origin of the PLL blocks was essential.

In the analog domain, the phase detector is comprised of a PFD which outputs two signals. These outputs either drain or feed current to the filter depending on the phase difference. In the digital domain, the current level describing the phase difference previously is now a binary value extracted from a TDC circuit.

The next circuit that was analyzed was the oscillator. The most important difference between the VCO and the DCO pertaining to this thesis was the frequency push information. The VCO reads the input voltage and adjusts its frequency push accordingly, whereas the DCO adjusts its frequency push based on how many capacitors are enabled in the capacitor banks. To translate the filter output to capacitors enabled, a decoder was implemented to enable the correct number of capacitors based on the filter output.

The filter was the final PLL component that was analyzed. Traditionally, it is implemented using a combination of resistors and capacitors. Their values decide the location of the corner frequency. In addition, the loop filter helps protect the oscillator from step and impulses seen at the input. Digitally, the filter differs by having two paths, a proportional and an integral path. The

integral path has a $\frac{1}{s}$ integration operation applied to the value in the path, then the value recombines with an untouched version of the input to produce an output suitable for the next stage.

6.4 Literature Review

The literature review aided to providing a scope into what was possible with MATLAB in terms of simulating a PLL. Most models were implemented in Simulink, one of which was an OPLL model which was explored in the review. The other two reviews were more focused on PLL simulation techniques in MATLAB.

From the three pieces of literature, the following was learned. First, it may be advantageous to split up one MATLAB loop simulation in preprocessing, simulator, and postprocessing stages. After considering their approach, it does seem efficient for situations where one has multiple simulations that all use the same initial parameters and output the same results for subsequent analysis. No real advantage was seen for this thesis, but it might be useful in future work with variations of the MATLAB model.

Second was the idea of timestamping the reference and the DCO signals then proceed with the simulation using an event-driven. The reasoning behind this was because in a normal simulation, one simulation resolution needs to be chosen, and clocks abide by this resolution. This resulted in frequency aliasing of the higher clock frequencies. On top of reducing aliasing, timestamping allowed for faster simulation times when compared to SPICE and Specter-RF simulation.

Finally, an OPLL Simulink model was analyzed with the intention of investigating the complexity and what was possible with Simulink. As expected, the Simulink model operates at a higher level which decreases the amount of customization the model has when compare to what is possible with MATLAB. This gave rise to the question as to whether MATLAB or Simulink would be better suited for this thesis. In the end, after a simulation speed test it was decided that MATLAB would be the better choice.

6.5 All-Digital Phase-Locked Loop

The goal of the All-Digital Phase-Locked Loop Model is to have a working MATLAB model that would provide faster results when compared to the Verilog behavioral model. One main reason that MATLAB was chosen over Verilog was due to the Verilog model's complexity. The TDC, Filter, and DCO varied from the common component design and it was simpler to adjust for these variances in MATLAB over Simulink.

The subsequent sections analyzed the Verilog behavioral models in detail. The purpose of this analysis was to gain a deep understanding of the Verilog model to properly port it over to MATLAB. Analyzing the initial Verilog model proved to be beneficial as the logic was reduced in MATLAB, this required extensive knowledge in the differences of the two environments. Between all the differences, three stood out as the most important. One, Verilog's ability in choosing between parallel and sequential executions. Two, the differences in how Verilog and MATLAB handle time. And three, Verilog's ability at working with binary values.

Utilizing the three main differences between the two environments, the individual blocks were implemented as standalone simulation then combined.

A reference frequency of 50 MHz and DCO output frequency of 6 GHz were chosen for the simulation and testing of the ADPLL MATLAB Model. The 50 MHz reference frequency was chosen because the fabricated ADPLL was design with this reference frequency in mind. As for the DCO output frequency, any value in the DCO operating range would have been a valid number taking into consideration the flat Decoder responses.

For the aforementioned reference and DCO frequencies, the TDC was able to produce the expected counts of 60 and 416 for *dcoCnt* and *tdcCnt*, respectively. To implement the TDC, a single mathematical line was used instead of the time-delay method that Verilog employed.

The filter was found to be the component that could most be simplified when going from Verilog to MATLAB. One reason was that MATLAB's environment is sequential in nature. When an execution is written after another in the code, it will wait for the first to complete then begin its execution. Verilog can have multiple operations executing at the same time. The filter's Verilog model needed the use of pipeline signals to make executions sequential, these were removed in the MATLAB model.

Taking the output from the filter and translating the value into usable data for the DCO was the function of the Decoder. It was able to correctly set the number of capacitors to be enabled in order to achieve the desired output frequency, except the instances where it encountered a flat region. Because of these regions, the DCO lost the ability to lock onto a total of 57 MHz spanning its operating range.

6.6 Simulation Time Analysis

This section investigated the source of the unexpectedly high simulation time. Further analysis using MATLAB's *Profiler* tool resulted in the following two conclusions as to why the simulation was not as efficient as anticipated:

1. The ADPLL MATLAB Model is based on a Verilog model that had its primary objective of being synthesizable. This resulted in not using MATLAB's full capacity as a mathematically based programming language.
2. The Decoder was found to be much more computationally taxing than initially expected. The root cause was found to be the use of the *round()* function. Unfortunately, removing the function is not an option as it compromises the Decoders functionality.

Future work must be done to overhaul the ADPLL Model into a more mathematically based design and improvements in the Decoder are required to speed up the simulation time.

6.7 Contributions

In the end the ADPLL MATLAB Model was able to successfully run behavioral simulations. They were not faster than the Verilog simulations and the reasons were presented in *Chapter 5: Simulation Time Analysis*.

The contributions of this thesis and the work conducted were:

- Designed and implemented a MATLAB environment and model where ADPLL simulations can take place.
- Implemented and tested standalone ADPLL block simulations to ensure each blocks functionality.
- Tested novel decoder algorithm to ensure its functionality. Discovered a flat response decoder issue that impacted DCO performance.
- Simplified logic of initial Verilog-to-MATLAB model (modeling time delays, pipelines, and bit shifts in MATLAB) to a more time and memory efficient model.
- Verified ADPLL functionality through TIE frequency domain analysis.
- Analyzed run-time of ADPLL MATLAB simulation and investigated reasons why simulation times were not competitive against Verilog simulations.
- Discovered that MATLAB's attempt at modeling a synthesizable-based Verilog model, on top of the Decoders computational inefficiency, were the sources of why the MATLAB simulations were not able to improve on the speeds reported by the Verilog simulations.

6.8 Future Work

6.8.1 Introduction

This section will discuss what could be improved and/or added to the ADPLL MATLAB model to improve performance and reliability. The future work on the simulation time issue was discussed in the conclusion of “*Chapter 5: Simulation Time Analysis*” and is not repeated below.

6.8.2 Model Improvements

6.8.2.1 2nd Order Filter

One aspect of the model that was intentionally omitted was the second order filtering capability. In the design process of the MATLAB model it was omitted to ensure each component was in its simplest state in order to expedite the finishing of a close loop model. This way, loop testing could begin earlier than if time was taken to translate and test the second order loop filter capabilities.

After the ADPLL model was finished and initial testing began, the second order portion of the loop filter was added resulting in the new schematic for the loop filter.

Figure 68: Second Order Loop Filter

Through testing and verifying of the ADPLL Model (TIE experiments) the *ordersel* was set to pass the data through the first filtering stage in Figure 68.

A quick test was conducted to show that the loop still functions in a closed loop manner and results in the desired DCO frequency with modulation applied to the reference. The test used a 600-kHz modulated clock with a *pShift* value of 11 (68.8kHz corner frequency) and was compared to a simulation with the same parameters but only first order loop capability.

Figure 69: 1st Order (left) vs 2nd Order (right) DCO Outputs

Illustrated in Figure 69 above the effect of the first filtering stage is definitely seen and affecting the output DCO PSD. Whether the modulation is behaving correctly about the corner frequency and dropping off at twice that of the first order model requires extensive further testing.

6.8.2.2 Flat DCO Decoder Response

It was discussed in the Decoder and DCO section on page 61 that the decoder was outputting flat regions. By flat, it is meant that for a range of input values, the decoder is outputting the same output. This is not particularly noticeable when the input is of lower magnitude or not one of the five big flat regions. But as it increases and hits a flat reign, the flat portion is scaled accordingly.

Ideally, the decoder output would be a linearly increasing staircase with steps only as big as the smallest bank resolution. For the most part this is what was seen, but to have a fully reliable

Figure 70: Five Largest Flat Decoder Responses

ADPLL system the output issue needs to be addressed.

Table 15: Lost Frequency Due to Flat Decoder Response

Initial FF [Hz]	End FF [Hz]	Frequency Lost [Hz]
1.441e8	1.595e8	15.4 MHz
4.552e8	4.615e8	6.3 MHz
6.868e8	6.964e8	9.6 MHz
9.3193e8	9.312e8	11.9 MHz
1.152e9	1.166e9	14 MHz

In these five instances alone, 57.2 MHz of frequency range was lost due to the decoders inability to provide the correct output. It appears that the biggest errors occur closest to the edge cases.

6.8.2.3 *TIE Inconsistency*

During the analysis of the TIE data to see how the loop was attenuating the modulation, it was found that for the case of varying the pShift over a static modulation the TIE analysis gave the opposite behavior of what was expected. Further analysis was done using the actual modulation peak from the divided down DCO signal and was found that it was in fact behaving in a lowpass manner. This information combined with the correct behavior observed using TIE analysis for static pShift suggests that the TIE analysis done for static modulations was incorrect.

This first step in correcting this issue would be to replicate the result. First the simulations used in the TIE analysis would need to be re-run then the TIE analysis itself should be repeated. For a sanity test, the TIE analysis for static pShift would be re-done as well to ensure the scripts are functioning correctly.

Trying the above mentioned steps, the next iteration of simulations would be done with an increased time span for more accurate frequency resolutions. More on this challenge will be discussed in the *Testing Improvements* section.

6.8.2.4 Jitter Calculation

Figure 71: DCO Phase Noise Profile with Jitter Limits (12kHz-20MHz)

The ADPLL Model was equipped with the capability of calculating the RMS Jitter following equation 5.56 below:

$$J_{rms} = \frac{\sqrt{2^{N_{12kHz-20MHz}/10}}}{2\pi f_o} \quad (6.43)$$

The main issue encountered while calculating the Jitter was integrating over the jitter range. This was done by multiplying the noise value by the frequency bin it was in. The bin sizes were calculated by taking the difference of each consecutive frequency point. Upon running the equation in MATLAB, it encountered a memory limit issue with a 1 μ s simulation of 0.4ps resolution

```
Out of memory. Type HELP MEMORY for your options.
Error in phaseNoisePlot (line 126)
pn_power = 2*sum(rsbsPSDdBc(low_index:high_index)' .* fd(low_index:high_index-1));
```

Figure 72: MATLAB Memory Error While Running Jitter Calculation

Further investigation shows that MATLAB has 2.0294 GB available for array allocation. To successively run a jitter calculation, the simulation environment needed to be brought down to 0.02 μ s with a 0.4ps resolution. The result of the simulation gave a jitter value of **0.19431 ns**.

Figure 73: Phase Noise Profile of Successful Jitter Calculation

The main issue with this result is the frequency resolution of the PSD and phase noise profiles is too high.

Figure 74: DCO Simulation with total time of 5us (left) and 0.02us (right)

Figure 74 illustrates how decreasing the simulation time increases the frequency resolution, and in doing so, it loses the details in the final noise spectrum. Resulting in an unreliable jitter calculation.

One of the main obstacles for this thesis, was simply how much computing power was available.

6.8.3 Testing Improvements

This section discusses improvement that can be done as future work to improve the testing environment. Upgrades to the testing improves the robustness and reliability of the ADPLL MATLAB Model.

6.8.3.1 Memory Limitation

The amount of memory available was the biggest limiting factor in the ADPLL MATAB Model. One of the first oversights in the beginning of the project was not paying attention to the time resolution. Originally, it was adjusted to better suit the current simulation being run. But this value needed to be a specific value, due to the operation of the TDC and the 0.8 ps clock being used the time resolution needed to be 0.4ps or less. After lowering the resolution is when the TDC began to output the expected counts, but in doing so it revealed how much memory the ADPLL System truly needed to run.

Related to the time resolution is the limiting factor of the frequency resolution in the PSDs. The frequency resolution of an FFT operation is known to be:

$$Freq_{res} = \frac{F_s}{N} \quad (6.44)$$

Where N is the total number of data points defined by:

$$N = \frac{t_{total}}{t_{res}} \quad (6.45)$$

Combining the above two equations yields the following relationship between the frequency resolution and the total simulation time:

$$Freq_{res} = F_s \cdot \frac{t_{res}}{t_{total}} = \frac{1}{t_{total}} \quad (6.46)$$

From this it is realized that to have the most accurate phase noise profiles possible, the total simulation time needs to be as high as possible. Creating a very crucial trade-off in simulating the ADPLL Model, either have low frequency resolutions or low simulation run times.

For this reason, only larger pShift values were able to be verified. A lower pShift translated to corner frequencies requiring low enough resolutions that the working machine simply could not support. Likewise with testing low modulation frequencies and varying pShift.

Future work on this ADPLL Model would have to be conducted on a much more powerful machine to attain high frequency resolutions and not compromise simulation time or be limited by how much memory is allocated to MATLAB for storage and computation.

6.8.3.2 *Comprehensive Corner Frequency and Modulation Frequency Testing*

Assuming a more powerful machine is utilized, each corner frequency would ideally be tested using both the TIE analysis method and simple modulation peak attenuation analysis. This Thesis covered only five test cases, two static pShift values and three static modulation frequencies. Ideally, 32 total tests (pShift =1:16) should be conducted to cover all possible combinations to prove the corner frequencies are in the correct location under closed loop operation.

7 Chapter 7: Executive Conclusion

This thesis presents a MATLAB Behavioral Model for an ADPLL comprised of a 0.8 ps resolution TDC, proportional integrating digital loop filter, and a 5.7-7.14GHz capacitor bank based DCO. The model presents a solution to modeling an ADPLL in MATLAB without requiring the building of a Simulink Model as it would have over complicated the design process and slowed down the simulation times.

Beginning with explaining the general idea of what a basic PLL is, the thesis dove into what phase was, considering it is the backbone of its operation. Then two major applications aside from clock synchronization are discussed; FM demodulation and clock recovery.

After, the thesis begins to discuss the different building blocks in a PLL, both analog and digital implementations are analyzed to get a good understanding of what is going to be designed in the MATLAB model. Said MATLAB model is based off a Verilog ADPLL model (used in fabrication), the reason for the MATABL conversion is to utilize MATLAB's mathematical prowess and to simplify the loop design. A discussion of the difference between the two design environments was done in order to explain certain code design decisions and limitations.

Through modulating the reference clock, moving the corner frequency (via pShift) and adjusting the modulation frequency resulted in the desired output behavior of the ADPLL MATLAB system. Albeit the results were not ideal values, they were enough to show the loop was locking to the correct desired frequency and that the loop filter was acting in a lowpass manner.

The model's inability to compete with the Verilog's simulation time was investigated. It was found that attempting to design the ADPLL too similarly to the Verilog behavioral model and the use of the *round()* function in the decoder caused the simulation to have unexpected high simulation times.

An important limitation encountered in this thesis was computing power, memory limitations resulted in a frequency resolution limit which hindered the testing of all possible corner frequencies. Other issues were found as well, the TIE analysis for a set of simulations provided the opposite result, in addition to not having a reliable jitter calculation. The latter two issues possibly stem from the frequency resolution and memory limitation issues.

In the end, the ADPLL MATLAB model can predict the behavior of the fabricated ADPLL with a higher simulation time than the Verilog behavior model. With the discussed improvements in *Chapter 5: Simulation Time Analysis* the MATLAB ADPLL can surpass the Verilog model and provide quick and accurate behavioral results.

8 Appendix

8.1 A: Simulink and MATLAB Simulation Speed Test Configurations

MATLAB Simulation Speed Test Code

```
for n = 1:runs
    %%%%%%%%%
    tic
    %%%%%%%%%
    s = 1e-3;
    us = 1e-6;
    ns = 1e-9;
    ps = 1e-12;
    fs = 1e-15;

    Hz = 1e0;
    kHz = 1e3;
    MHz = 1e6;
    GHz = 1e9;

    tstep      = 10*ps          ;
    tstop      = 500*us-tstep   ;
    time       = 0:tstep:tstop  ;

    clkFreq = 50*MHz;

    waveTest1 = sin(2*pi*clkFreq*time);
```

```

figure
plot(time,waveTest1)
xlabel('Time (sec)')
ylabel('Amplitude (V)')
title('50 MHz Sinewave ')
grid on

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
matlabSimTimes(n) = toc;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

(n/runs)*100

pause(2)
clear_figures
end

```

8.2 B: Corner frequency vs. Desired Frequency Behaviors

Figure 75: Cutoff Frequency with respect to changing desired frequency

Figure 76: Cutoff Frequency with respect to changing pShift

Figure 77: Cutoff Frequency with respect to changing tdc resolution

8.3 C: Two's Complement Binary to Decimal MATLAB Functions

```
function [decOut] = twosBi2De(bin,rsb,bits)
%UNTITLED5 Summary of this function goes here
% Detailed explanation goes here

% assuming rsb for checking...
if(bin(bits) == 0)
    % Positive 2s num
    decOut = bi2de(bin);
else
    % Negative 2s num
    binInv = ~bin;
    binInv = double(binInv);
    binInvDec = bi2de(binInv,rsb);
    binInvDecP1 = binInvDec + 1;
    decOut = (-1)*binInvDecP1;
end

end

function [binOut] = twosDe2Bi(num,rsb,bits)
%UNTITLED Summary of this function goes here
% Detailed explanation goes here
% check number is within bit range

% binOut = zeros(1,bits);
% numPosBin = zeros(1,bits);

highRange = 2^(bits-1) - 1;
lowRange = -2^(bits-1);

if((highRange >= num) && (num >= lowRange))
    if(num >= 0)
        %If number is positive then f(x) works as is
        binOut(:) = de2bi(round(num,0),rsb,bits);
    else
        % Here the number is negative
        %1) Convert the number into positive
        numPos = (-1)*(num);
        numPosBin(:) = de2bi(round(numPos,0),rsb,bits);
        % double() is needed to convert back to numerical array and not
        % logical array
        numPosBinInv = ~numPosBin;
        numPosBinInv = double(numPosBinInv);
    end
end
```

```

    numPosDecInv = bi2de(numPosBinInv);
    numPosDecInvP1 = numPosDecInv + 1;
    numNegBin = de2bi(round(numPosDecInvP1,0),rsb,bits);

    binOut = numNegBin;

end
else % was getting an error whole running sim Jan 16th 2019

% error('Decimal does not agree with set number of bits')
if(num>highRange)
    binOut(:) = de2bi(round(highRange,0),rsb,bits);
else
    % Here the number is negative
    %1) Convert the number into positive
    numPos = (-1)*(lowRange);
    numPosBin(:) = de2bi(round(numPos,0),rsb,bits);
    % double() is needed to convert back to numerical array and not
    % logical array
    numPosBinInv = ~numPosBin;
    numPosBinInv = double(numPosBinInv);
    numPosDecInv = bi2de(numPosBinInv);
    numPosDecInvP1 = numPosDecInv + 1;
    numNegBin = de2bi(round(numPosDecInvP1,0),rsb,bits);

    binOut = numNegBin;
end
end
end

```

9 Bibliography

- [1]"BaySand to Initiate a Unique Low Cost Multi Project Wafer (MPW) Program", *BaySand Inc*, 2016. [Online]. Available: <https://www.baysand.com/press-release/baysand-initiate-unique-low-cost-mpw/>. [Accessed: 03- Jun- 2019].
- [2]R. Wilson, "Chip industry tackles escalating mask costs", *EETimes*, 2002. [Online]. Available: https://www.eetimes.com/document.asp?doc_id=1177359#. [Accessed: 04- Jun- 2019].
- [3]M. Lapedus, "Semiconductor Engineering - FinFET Rollout Slower Than Expected", *Semiconductor Engineering*, 2015. [Online]. Available: <https://semiengineering.com/finfet-rollout-slower-than-expected/>. [Accessed: 04- Jun- 2019].
- [4]E. Notes, "PLL FM Demodulator | Phase-Locked Loop Detector | Electronics Notes", *Electronics-notes.com*, 2019. [Online]. Available: <https://www.electronics-notes.com/articles/radio/modulation/fm-frequency-demodulation-phase-locked-loop-pll-detector-demodulator.php>. [Accessed: 19- Mar- 2019].
- [5]E. Fischer, E. Dalhoff, S. Heim, U. Hofbauer and H. Tiziani, "Absolute interferometric distance measurement using a FM-demodulation technique", *Applied Optics*, vol. 34, no. 25, p. 5589, 1995. Available: 10.1364/ao.34.005589.
- [6]Y. Hwang, B. Hwang, H. Lin and J. Chen, "PLL-Based Contactless Energy Transfer Analog FSK Demodulator Using High-Efficiency Rectifier", *IEEE Transactions on Industrial Electronics*, vol. 60, no. 1, pp. 280-290, 2013. Available: 10.1109/tie.2011.2181135.
- [7]H. Sone, H. Okano and S. Hosaka, "Picogram Mass Sensor Using Piezoresistive Cantilever for Biosensor", *Japanese Journal of Applied Physics*, vol. 43, no. 7, pp. 4663-4666, 2004. Available: 10.1143/jjap.43.4663.
- [8]P. Allen, "Lecture 200 – Clock and Data Recovery Circuits", Georgia Institute of Technology, 2003.

- [9]J. Rogers, C. Plett and I. Marsland, *Radio frequency system architecture and design*. Boston: Artech House, 2013, pp. 207-261.
- [10]K. Cheng, C. Hu, J. Liu and H. Huang, "A Time-to-Digital Converter Using Multi-Phase-Sampling and Time Amplifier for All Digital Phase-Locked Loop", *Department of Electrical Engineering*, 2010.
- [11]J. Rogers and C. Plett, *Radio Frequency Integrated Circuit Design*. Norwood: Artech House, 2010.
- [12]V. Singh, "Discussion on Barkhausen and Nyquist stability criteria", *Analog Integrated Circuits and Signal Processing*, vol. 62, no. 3, pp. 327-332, 2009. Available: 10.1007/s10470-009-9360-4.
- [13]Y. Moon, Y. Roh, C. Jeong and C. Yoo, "A 4.39–5.26 GHz LC-Tank CMOS Voltage-Controlled Oscillator With Small VCO-Gain Variation", *IEEE Microwave and Wireless*
- [14] S. Abdullah, "Design of Linear LC Digitally Controlled Oscillator Using Topographical Field Maps", Masters, Carleton University, 2019.
- [15]G. Hsieh and J. Hung, "Phase-Locked Loop Techniques - A Survey", *IEEE Transactions on Industrial Electronics*, vol. 43, no. 6, 1996.
- [16]A. Blanchard, *Phase-Locked Loops Application to Coherent Receiver Design*, 1st ed. Wiley-Interscience, 1976, pp. 36-44.
- [17]"Time Response of Second-Order Control System | Electrical4U", *Electrical4U*, 2019. [Online]. Available: <https://www.electrical4u.com/time-response-of-second-order-control-system/>. [Accessed: 28- May- 2019].
- [18] W. Tranter, T. Bose and R. Thamvichai, *Basic simulation models of phase tracking devices using MATLAB*. San Rafael, Calif. (1537 Fourth Street, San Rafael, CA 94901 USA): Morgan & Claypool, 2010.

- [19] I. Syllaos, R. Staszewski and P. Balsara, "Time-Domain Modeling of an RF All-Digital PLL", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 55, no. 6, pp. 601-605, 2008. Available: 10.1109/tcsii.2007.916845.
- [20] Z. Chen, Y. Hu and L. Liu, "Nonlinear analysis of BPSK Optical Phase-Locked loops using MATLAB and Simulink", *Proceedings of the 2nd Information Technology and Mechatronics Engineering Conference (ITOEC 2016)*, 2016. Available: 10.2991/itoec-16.2016.1.
- [21] M. Blagov, N. Kuznetsov, G. Leonov, M. Yuldashev and R. Yuldashev, "Simulation of PLL with impulse signals in MATLAB: Limitations, hidden oscillations, and pull-in range", *2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, 2015. Available: 10.1109/icumt.2015.7382410.
- [22] G. Bianchi, N. Kuznetsov, G. Leonov, M. Yuldashev and R. Yuldashev, "Limitations of PLL simulation: Hidden oscillations in MatLab and SPICE", *2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, 2015. Available: 10.1109/icumt.2015.7382409.
- [23] "What is the relation between FFT length and frequency resolution?", *Electrical Engineering Stack Exchange*, 2011. [Online]. Available: <https://electronics.stackexchange.com/questions/12407/what-is-the-relation-between-fft-length-and-frequency-resolution>. [Accessed: 02- Jul- 2019].
- [24] T. Wang, "A Phase-scaled Vernier Time-to-Digital Converter Architecture with Switchable Coarse/Fine Resolutions, Wide Range and Ultra Low Power Consumption", Masters, Carleton University, 2016.
- [25] R. Du, J. Liu and Y. Li, "Research of DSP Code Generation Based on Matlab-DSP Integrated Design", *Applied Mechanics and Materials*, vol. 551, pp. 244-247, 2014. Available: 10.4028/www.scientific.net/amm.551.244.

[26] K. Bagal, C. Kadu, B. Parvat and P. Vikhe, "PLC Based Real Time Process Control Using SCADA and MATLAB", *IEEE*, 2019. Available: 10.1109/ICCUBEA.2018.

[27] R. Brown, "End of function slow - Matlab Profiler - MATLAB Answers - MATLAB Central", *Mathworks.com*, 2019. [Online]. Available: <https://www.mathworks.com/matlabcentral/answers/83077-end-of-function-slow-matlab-profiler>. [Accessed: 20- Sep- 2019].