The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

AN INVESTIGATION INTO THE THEORY AND IMPLMENTATION OF SOFTWARE DEFINED RADIO SYSTEMS

A Thesis in

Computer Science and Engineering

by

Steven M. Brown

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 2009

The thesis of Steven M. Brown was reviewed and approved* by the following:

Kyusun Choi
Assistant Professor of Computer Science and Engineering
Thesis Advisor

Lee Coraor
Associate Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School

ABSTRACT

This thesis describes the implementation and integration of software defined radio algorithms on a low cost platform. With an eye toward efficient, cost-conscious design, a system is shown to be capable of direct transmission and reception of radio broadcasts. This system, like all software defined radios, was constructed to minimize the amount of analog signal conditioning. A variety of decisions and investigations are described to show the flexibility of this system. Simulations are also described showing the feasibility of alternative modulation techniques.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Dr. Kyusun Choi for his guidance and inspiration during the course of the work described in this thesis. I would also like to thank Dr. Sven Bilén for his recommendations when our work ground to a stop. Most of all, I would like to thank Loay Naji and Justin Ford without whom this work would not have been possible.

This research was partly funded by the eSensIF MURI project of the IST Division of the Applied Research Laboratory which is supported by the Army Research Laboratory and the Army Research Office under award W911NF-07-1-0376. Any opinions, findings and conclusions, or recommendations expressed in this thesis are those of the authors and do not reflect the views of the sponsors.

Chapter 1

Introduction

This thesis describes an investigation into the implementation of a software defined radio system [1], [2]. This investigation concentrated on software defined radio technology and techniques. The result was the construction of a software defined radio, a radio whose frequency tuning and bandwidth adjustment were performed in software rather than hardware; after a software update, the capabilities and modulation technique of the platform were also modifiable.

What is software defined radio? According to the Software Defined Radio Forum [3], software defined radios are “radios that provide software control of a variety of modulation techniques, wide-band or narrow-band operation, communications security functions (such as hopping), and waveform requirements of current and evolving standards over a broad frequency range.” This is the broad definition for a software defined radio, however. The key to a true software defined radio, according to the Software Defined Radio Forum is its capability to store a “large number of waveforms or air interfaces” [3] and its ability to add new waveforms. This contrasts with the definition for software controlled radios; “software controlled radios have control functionality implemented in software, but do not have the ability to change attributes, such as modulation and frequency band, without changing hardware.” [3]

The need for software defined radio comes from several factors. Software is faster to develop than hardware. Using software rather than hardware speeds up the time-tomarket drive for new products. Software is cheaper to develop than hardware. This lowers development costs. Since software is inherently more modular than analog hardware, intellectual property can be leveraged. Since software defined radios have generic analog subsystems, the software can implement multiple standards of communication while going through various regulatory processes only once (at least for Underwriters Laboratories; the Federal Communications Commission would demand re-approval for modified operational bands). Also due to the generic nature of the software defined radio system, a single standardized radio system could be individually customized by the Originating Equipment Manufacturer from a mass produced radio. This allows both product differentiation as well as the economy of scale.

Software defined radios are a promising technology. Though software defined radios currently become economically viable when three or more radios must be used (three hardware radios), digital systems' speed is increasing all the time while their costs are decreasing. This technology has already made inroads into military communications and some base-station cellular products. Before this technology can become truly ubiquitous, the minimum hardware requirements must be understood.

This thesis describes two related investigations. The first was the construction of a small software defined radio (small in terms of hardware footprint) to perform Amplitude Modulation demodulation and data communication using Binary Frequency Shift Keying. The second investigation probed the design of a Phased Locked Loop (PLL) so that Frequency Modulation and Phased Shift Keying could be explored.

Chapter 2

System Architecture

Any design requires trade-offs. Sometimes this trade-off is an exchange of cost and performance. In other cases, the trade-off is reliability and cost.

This chapter will discuss the design decisions that are required in a software defined radio and the design of the radio that is the primary focus of this thesis. The software defined radio to be described required a decision on where to place the analog to digital split. The radio's software also required some processing element. The last portion of this chapter will be devoted to the specific components considered for the design of the software defined radio.

2.1 The analog/digital split and processing components

The analog/digital split is the most significant decision in the design of any software defined radio. This decision not only determines the scope of the analog subsystem, but also the processing requirements placed on the software subsystem [4].

Though the analog system must contend with the potential system bandwidth (a software defined radio's bandwidth may vary as communications standards change), the analog system may need additional features in order to support the digital system. Features that the analog front-end may need to include are bandwidth limiting filtering and frequency down-conversion. Though these tasks can both be done in the digital domain, the extent to which these tasks need to be done needs to be determined. At this time, bandwidth limiting is required in all systems that do not use a frequency selective antenna that acts as a bandwidth limiting filter. This filtering is required to eliminate the aliases that occur during digital sampling of the analog signal. Frequency down-conversion may also be required, depending on the sampling rate of the digital system and the target frequency band. In the analog back-end, frequency up-conversion and noise shaping are required when the output frequency of the system is beyond the capabilities of the digital-to-analog converter (DAC).

Given the potentially heavy computational burden placed on the digital subsystem of a software defined radio, the nature of the processor in the radio needs to be determined. The nature of the system can be either a sequential system, such as a microprocessor, or a parallel system, such as a Field Programmable Gate Array (FPGA). The class of processing device would need to be determined at the time of the design of the analog/digital split. The processing capabilities must also support user visible features such as encryption and audio decoding in addition to the low-level signal processing. These tasks can also be split among several processing elements, such as digital signal processors (DSPs), microcontrollers, and FPGAs.

2.2 Prototype system

The system consisted of four components. The components were an analog front-end, an ADC evaluation board, an FPGA evaluation board, a DAC, and an analog back-end prototyping board (on which the DAC was mounted). The hardware structure of the system is shown in Figure 2-1.

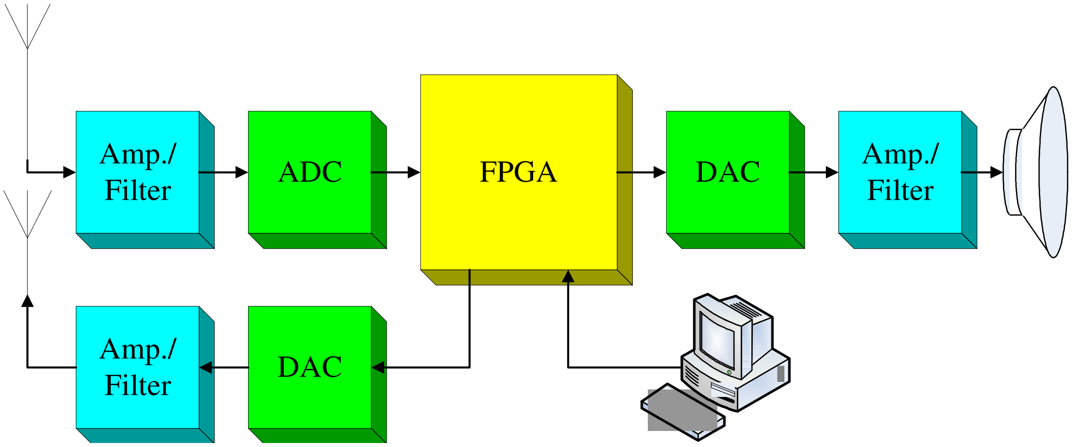


Figure 2-1: Hardware architecture

2.2.1 Front-ends

The analog front-end consisted of two primary components. The most important component of the front-end was the antenna. The second component was a filter network.

The antenna in this system was an inductor based antenna. Though any antenna will suffice for the purpose of receiving radio waves, a frequency selective antenna was much more useful for demonstration purposes. This antenna was removed from a commercial radio in order to eliminate the concern for its functionality (it worked). The antenna design was optimized for AM reception. The most common type of AM antenna has been the inductor based antenna (the inductor may be either air-core or ferrite-core; this antenna was a ferrite core). In addition to the inductor, the antenna was accompanied by a variable capacitor. Though the inductor of the antenna and the capacitance range of the capacitor were not known, this fact was also irrelevant (it worked).

The second component in the system was a filter network. The filter network in a system of this nature is tuned to the antenna. The original filter network for this system was design by Justin Ford [5]. The antenna used was a simple long-wire antenna. Unfortunately, an antenna of this variety is susceptible to electronic noise (long-wire antennas are highly receptive to E-field energy). In addition to the excessive noise received from the laboratory equipment, the filter network itself was troublesome to tune for the antenna. The inductor-based antenna had a much lower amplitude output, making a high-gain but unselective filter network necessary. This second filter network design was implemented by Michael Debole. AM band communication appears to be best accomplished through the use of inductor based antennas (in spite of the difficulty of creating a high gain amplifier network). In addition to their band-selectivity, the inductor based antenna's preference for M-field energy seemed to have helped with the noise problem that was initially encountered with the long-wire antenna.

2.2.2 ADC evaluation boards

Several ADC boards were tested in the radio system. The first board was a 200 Msps, 10-bit combined ADC and DAC board. The second board was a 50 Msps, 8-bit ADC board. The third board was a 20 Msps, 10-bit ADC board.

The first board tested was an Analog Devices AD9410 evaluation board. The full capabilities of this board were never explored. Though the best speed and resolution of the boards selected, the connection between the ADC board and the FPGA evaluation board was unreliable. Due to the poor design of the ADC board, the included DAC would not be accessed without board modification. This fact combined with the complex wiring and timing required to make this board functional made it a poor choice for additional development. The part number for the DAC on this board was AD9751.

The second board tested was a National Semiconductor ADC08100 evaluation board. Though this board’s resolution was not as good as the other boards, its sampling rate did mitigate this fact. In terms of radio communication, this resolution limitation is of paramount concern. Resolution in radio communication is equivalent to dynamic range. This is to say that if two competing signals are present in the received bandwidth of the ADC, the resolution of the ADC will be the ultimate determining factor of whether the weaker signal can be received in spite of the stronger signal. Surprisingly, this evaluation board seemed to receive the AM voice data the best of the three ADC boards (an objective test was never performed to verify this opinion). Unfortunately, one of the two matched boards failed during the course of testing.

The third board tested was a National Semiconductor ADC10321 evaluation board. This board had the slowest sample rate of the boards tested. As is described in chapter 4, this was not a problem for the FPGA used to implement the software defined radio. Though not extensively tested, a comparison of the ADC08100 and ADC10321 was compared to determine the relative ability to transfer data. Though not exhaustively tested, observations suggested that the ADC10321 performed better for data transmission than the ADC08100.

2.2.3 FPGA evaluation boards

Two FPGA evaluation boards were tested in this project. The first board tested was the Digilent D2E evaluation board with DIO1 add-on board. The second board tested was the Digilent S3BOARD evaluation board. Both FPGA boards operated at 50 MHz. The Spartan®-3 based S3BOARD operated with lower latency compared to the Spartan®-2 based D2E board for the same VHDL code (as reported by the Xilinx synthesizer).

2.2.4 DAC chip

Unlike the other components of this system, only one DAC was investigated. The device chosen was the TI TLC7524. This DAC was an 8-bit low speed device (no more than 6 Msps, though no sample rate was listed). The DAC could be described as a digitally controlled resistor network. Due to its weak current-mode output, a voltage buffer (unity gain amplifier) was required to provide current drive to a filter network. Though inaccurate (8-bit), low speed, and possessing anemic drive capability, the DAC had the bandwidth to support both audio reproduction and AM band transmission (voltage buffer provided).

2.2.5 Analog back-end

The fourth component of the software defined radio system was the analog back-end. Two back-ends were used for testing. The first back-end was an audio range filter network with powered PC speakers. This filter eliminated the digital noise that was generated by the DAC. The second back-end was an amplifier network capable of driving a long-wire antenna. Though no filtering was employed for this amplifier, its short transmission distance did not necessitate such a filter.

2.3 PLL structure

The PLL design simulated during this investigation was based on a presentation by Fred Harris [6]. This PLL contained Complex (real and imaginary valued) multiplication of the input data and generated frequencies, a trigonometric function, a filter, and a Digitally Controlled Oscillator (DCO). Figure 2-2 shows the original structure of the PLL. The DCO in this system consisted of a phase accumulator and a complex sine/cosine lookup.

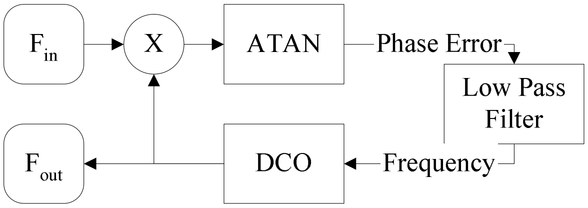


Figure 2-2: Structure of Harris' PLL

The complex multiplication of the input and output frequencies can be seen in Eq. 2.1. As can be seen here, the arctangent of the imaginary and real parts of the product of the input and output will directly generate the phase error of the PLL. Since the input to the DCO must be the frequency, the low pass filter in this system must be tracking the frequency of the input.

$$e^{j2\pi\Phi} \times e^{-j2\pi\Theta} \equiv \cos(\Phi - \Theta) + j \times \sin(\Phi - \Theta) \qquad \textbf{2.1}$$

Though this basic structure was retained, a number of enhancements were applied to make the system more amenable to implementation. The arctangent function was the most offensive part of this system and was eliminated. This function could be approximated by simply disregarding the real component of the multiplication and dividing the imaginary result by four. This approximation dramatically reduced the effort involved in the complex multiplication while completely eliminating the arctangent function.

With this PLL, both frequency demodulation and phase shift keying can be performed. Frequency demodulation can be trivially derived from the input value to the DCO by subtracting the carrier frequency, a constant. Phase shift keying requires slightly more logic since it requires a reference PLL to lock onto the phase of the transmission and a tracking PLL to follow the transmission after the transmission preamble.

Phase shift keying does not require two complete PLLs. In fact, the reference and tracking PLL share everything but the DCO input value and DCO phase accumulator. In effect, these two PLLs can be implemented with one PLL and two phase accumulators. During preamble locking, both accumulators can operate to lock onto the preamble. After this preamble lock, the reference phase accumulator holds its frequency input constant while the tracking phase accumulator tracks the subsequent phase shifts that contain the data transmission. By comparing the phase output of the two accumulators, the phase difference encoding the data immediately results.

Chapter 3

Integration Issues

The components in the software defined radio were designed for different systems and needed to communicate with one-another; problems arose. Often these components were not designed to interoperate. Not only did the components in this heterogeneous system have mechanical difficulty, but they also had logical difficulty communicating among themselves (“logical” referring both to logic levels, voltages, and protocols). When these components were connected, performance issues also arose.

3.1 Digital subsystem integration

Though the digital components could be treated as mere components to be placed on a printed circuit board (PCB), inherent difficulties in achieving this interoperability existed: physical interconnection, logic timing, register initialization and data extraction, and high-level protocol support. Physical interconnection concerns pins, cables, and connectors. Logic timing refers to the rate and level at which signals change on those pins. Register initialization and data extraction refer to communication between two devices. High level protocol support refers to communication between systems. Combined, these form a communication hierarchy.

3.1.1 Physical interconnection

At the most primitive level of interconnection, physical factors were dealt with. The system construction for the National ADC boards and the TI DAC went smoothly, as can be seen in Figure 3-1 and Figure 3-2. Unfortunately, the Analog Devices ADC and DAC caused significant problems, as can be seen in Figure 3-3.

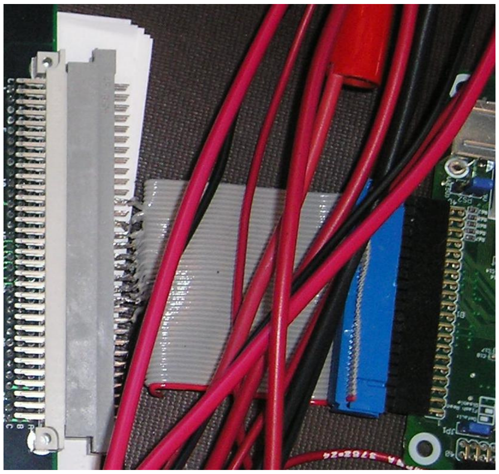


Figure 3-1: National ADC board cable

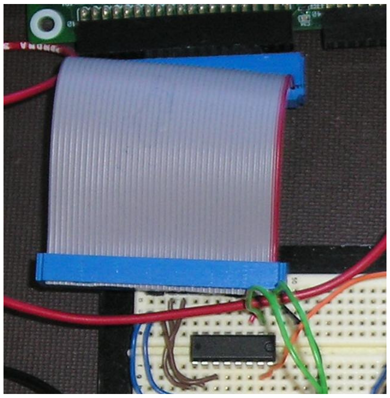


Figure 3-2: TI DAC cable

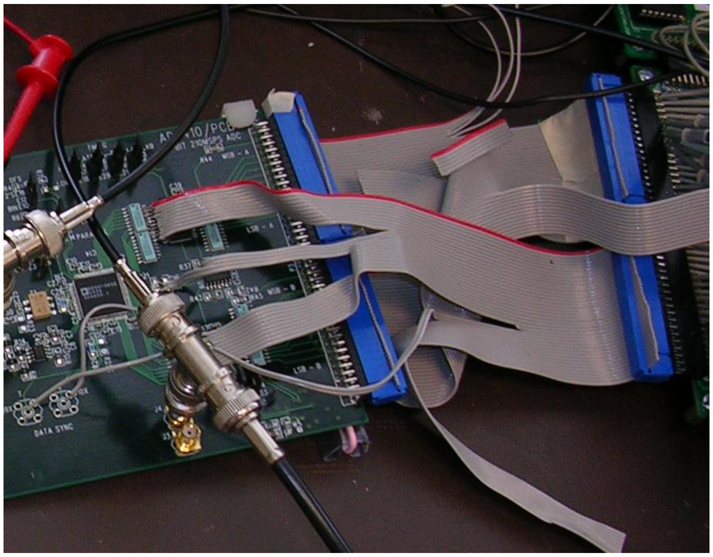


Figure 3-3: Analog Devices ADC and DAC cabling

The cause of the difficulty in cable construction for the Analog Devices board was that the board was never designed to operate as two separate devices. Though the DAC could be driven directly with minor board alterations, as described in {next section}, extracting the ADC data from the board required more invasive tactics, highlighted in Figure 3-4 (the ribbon cable was soldered directly to resistor packs on the evaluation board).

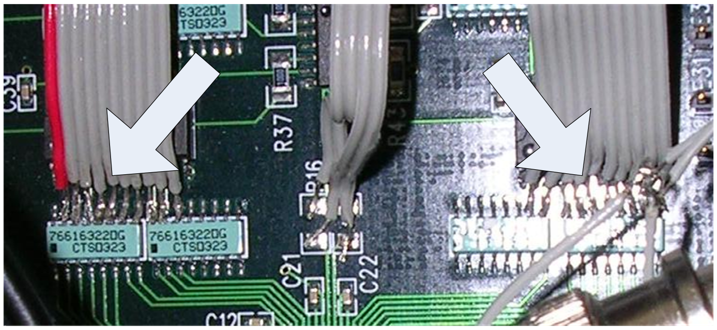


Figure 3-4: ADC data extraction

3.1.2 Logic timing

Once these devices were physically connected, they could interact electronically. Though they could interact, the goal was to achieve meaningful interaction. This required compatible voltage levels and appropriate signal timing.

Though both logic timing and logic levels are important, ensuring the proper logic level for interconnection is of paramount importance. The FPGAs in the system had 3.3V logic inputs and outputs. The National ADCs and TI DAC used in the system were all 5V parts. The ADC outputs had voltage levels capable of operating the FPGA inputs; though, operating at 5V was causing stress to the FPGA's inputs (this could reduce the reliability of the system, since the FPGA's inputs were not 5V tolerant). The DAC's inputs were also operating at 5V; however, since the FPGA's outputs swung wide enough to reach the switching threshold of the DAC, there was no problem with DAC operation. DAC speed was not significantly affected by the logic level mismatch due to the sluggish speed of the DAC. The Analog Devices ADC and DAC were both 3.3V parts, avoiding these concerns.

Logic timing in the system was a concern with the ADCs, but less of a concern with the DACs. The DACs used in the system supplied simple latch-style interfaces, requiring simple clock division to slow the FPGA signals down to the rate at which the DACs operated. The ADCs in the system supplied three different logical interfaces. The ADC10321 supplied a simple latch-based interface, just like the DACs. The ADC08100 was isolated from the FPGA by a FIFO on the evaluation board. These posed an interesting problem. Though they supplied a latch-style interface to the FPGA, the status flags telling the FPGA whether the FIFO was full or empty were not supplied to the FPGA. The result of this was that the FPGA was operated at a rate comparable to the rate of the ADC; this was obviously non-optimal (in the case of under-flow, the FIFO would doubtless supply garbage). The final logic interface was that of the AD9410. Unlike the other devices that were controlled by the FPGA, the AD9410 –supplied– the latch style interface to the FPGA (the FPGA had to act as the latch in the system). Rather than be concerned with two separate clock domains, the ADCs control signals were used as the sole clock in the system (a FIFO would have had to be created within the FPGA to separate the two clock domains).

3.1.3 Register Initialization

A memory based interface is one in which an address and a data bus (that may be multiplexed) are presented to the controller of the slave device. A latch based interface is an interface in which there is only a data bus and data-present signal. The memory based interface requires both a data-present signal and an address-present signal. The data bus may also operate bi-directionally, requiring a read-or-write signal. Generating the control signals is only slightly more complex between the two interface styles.

All the devices connected to the FPGA in the software defined radio system either supplied or expected a latch-based interface. Memory based interfaces typically supply a number of internal register locations (internal to the ADC or DAC) that would allow the system implementer to set internal voltage references. The difficulty with these devices is not in the generation of the control signals, but rather in the register initialization itself.

The initialization of the registers requires a state machine operating at a level above the interface timing level. This initialization process may also require a specific sequence of operations in order to set the device in the proper operating mode. Due to the opaque nature of the system, debugging this initialization process can be quite difficult.

3.1.4 High level protocols

Spanning the breadth of the communications stack, high level protocols are the most powerful and abstract. These communication formats tunnel through the other tiers of communication to transmit the information that is needed for system level operation. Though the lower tiers of communication have an affect on performance, the high level protocol will have the most profound impact on performance.

In the software defined radio, there was one high level protocol employed. The protocol was used to upload tuning coefficients the radio while testing AM reception. These coefficients were the coefficients for the filter. This protocol was a simple transmit and acknowledge format with fixed sized messages. Though no timing for communication was specified, the acknowledgement was expected to be generated within a microsecond of message reception (the application written reported an error when no acknowledgement was received within approximately 300 milliseconds). The format of the message sent to the radio is shown in Table 3-1. In order to upload a full set of filter coefficients (or filter selection, basically a radio station preset button like on a car radio), three messages and acknowledgements were required to upload the three coefficients. Since this protocol did not have high performance requirements, the overhead incurred by the acknowledgements was not an issue. The protocol simply had to send the data at a fast rate from a mere human perspective (were the perspective that of a computer, the 25% overhead from the one-byte acknowledgements may have been excessive). No error checking was implemented in this protocol and the acknowledgement byte was not inspected after receipt.

Table 3-1: Software Defined Radio Coefficient Upload Protocol

<table>
<tr><td colspan="8">Bits 0-7: exponent</td></tr>
<tr><td colspan="8">Bits 8-15: least significant 8 bits of fraction</td></tr>
<tr><td colspan="2">Bits 16-17: most significant 2 bits of fraction</td><td>Bit 18: sign of fraction</td><td colspan="2">Bits 19-20: selected coefficient (0 = A1, 1 = A2, 2 = G)</td><td colspan="2">Bits 21-22: filter selection (0 is pass-through)</td><td>Bit 23: unused</td></tr>
</table>

3.2 Performance Optimization

System performance during the design of the software defined radio was a concern. In addition to the central filtering algorithm, the support code's performance was also an issue. The performance of these algorithms was of a concern of both space and speed of the resulting implementations.

3.2.1 Pipelining Applicability

Pipelining is a way of time-division multiplexing a single set of hardware to perform multiple tasks [7]. This is accomplished by dividing a single linear hardware process into multiple independent pieces. Each of these pieces is itself a linear hardware process. These pieces are separated from one another by registers. These registers hold the intermediate inputs and outputs from the independently executing hardware processes. These independently executing processes form a parallel computer. The key to applying pipelining is being able to divide the computational task into multiple independently executing segments. The sizing of these segments is typically determined by the target operating frequency. Pipelining necessarily adds latency to the system. The goal of pipelining is to increase through-put.

Pipelining was employed in two tasks in the software defined radio. The first task was converting the integer numbers from the ADC into the internal floating point format within the FPGA. The second task was converting the floating point format back to integer numbers for the DAC. In the case of integer to floating point conversion, the operation is zero counting combined with shifting. Converting floating point numbers back to integers is a shifting operation. The FPGA used contains registers on the outputs from each of the combinational elements, minimizing the size impact of pipelining. The increased latency added to the processing of the received waveforms was insignificant compared to the execution speed of the filter algorithm. A total of three pipeline stages running at 50 MHz were used while the filter processed one sample at 1/3 of the 50 MHz clock rate. So, the pipelined conversion blocks could provide samples at three times consumption rate of the filter algorithm.

3.2.2 IIR Pipelining

During the early phases of the development of the software defined radio, pipelining was explored as an option to speed up the processing of the filter algorithm. Since pipelining is a scheme used to speed up computation, this attempt was not entirely flawed. Unfortunately, an IIR algorithm is based on a feedback path. This feedback path results in a system whose inputs depend on its outputs. This path renders all computation along the path self-dependent. Since the filter algorithm is entirely a feedback path, this results in a system that cannot be pipelined. This fact was observed by a significant increase in processing speed once the pipelining was removed from the floating point operations.

Chapter 4

Algorithm Implementation

This chapter concerns the structure and use of four algorithms. The first section shows how the ADC data was converted between the floating point format and the data formats supported by the ADC and DAC. The second section covers the AM modulator and demodulator. The third section describes the Binary Frequency Shift Key (BFSK) modulator and demodulator, based on the AM modulator and demodulator. The fourth section explores the PLL design.

4.1 Floating point conversion

The first algorithm to be implemented after the ADC and DAC driver was the floating point conversion modules. There were two modules: the floating point to fixed point module and the fixed point to floating point module. These modules were initially prototyped in “C”, along with initial implementations of floating point routines for addition, subtraction, and multiplication. Though the floating point routines for addition, subtraction, and multiplication were later updated to reflect the operation of the VHDL implementations, the conversion routines in “C” were left mostly unchanged. Since the conversion modules added phase shift but were not on the critical timing path in the FPGA, they were both pipelined with a maximum delay matched to the speed of the critical path (the critical path in this implementation was the band-pass filter).

Due to the light-weight floating point format used in this design, described in appendix **A.1** and influenced by [8], the conversion from an integer format to the floating point format consists primarily of counting leading zeros. A simple shift-register approach was employed in the "C" conversion routines shown in appendix **A.2**. This trivial approach ran with speed linearly related to the number of bits in the integer format. A method that operated with speed logarithmically related to the number of bits in the integer format was easily created, however. This logarithmic speed counter was used in the FPGA implementations and was carried over into the floating point adder for the purposes of format normalization.

The FPGA portions of this research needed to convert unsigned integers to and from the reduced precision floating point format. Though the "C" routines in **A.2** were tolerable when running on a desktop computer, they were not quite the proper implementation for an FPGA. As mentioned, the "C" integer conversion routines used single bit-shifts to determine the number of leading zeros in the incoming integer. This results in a linear running time. Logarithmic zero counting allows simultaneous exponent calculation and format normalization. Table **4-1** shows the pseudo-code for this algorithm for an unsigned 16-bit integer. When converting from an unsigned integer from 9 through 16 bits, the logic remains unchanged with the exception of line 6 which changes to "exponent = (number of input bits)-exponent".

Table 4-1: Pseudo-code For Logarithmic Conversion From Unsigned Integer

```
(1) exponent = 0, mantissa = input
(2) if leading 8 bits of mantissa are 0, left-shift mantissa 8 bits and OR 8 into
    exponent
(3) if leading 4 bits of mantissa are 0, left-shift mantissa 4 bits and OR 4 into
    exponent
(4) if leading 2 bits of mantissa are 0, left-shift mantissa 2 bits and OR 2 into
    exponent
(5) if leading bit of mantissa is 0, left-shift mantissa 1 bit and OR 1 into exponent
(6) if mantissa is non-zero, then exponent = 16-exponent
(7) else, exponent = 0
```

The conversion from the reduced precision floating point format to an unsigned integer has a few special cases and is somewhat more difficult to follow than conversion to the reduced precision format. Since the reduced precision floating point format can represent negative numbers and unsigned integers cannot, the FPGA implementation replaced the exponent with a value that resulted in the entire contents of the mantissa being shifted into oblivion. This replacement was also performed when the exponent was negative since unsigned integers cannot represent values between one and zero (purely fractional values are insignificant). In addition to these cases when the exponent was large enough that the most significant digits of the mantissa would be lost, the unsigned integer output was saturated. While these checks were performed, the number of right-shifts was calculated. The last steps in this conversion were logarithmic right-shifts. Table 4-2 shows the pseudo-code for the algorithm working with 16-bit integers. Since the FPGA implementation used 10-bit mantissas, the table's step 4 was not necessary. To convert this pseudo-code into a format from 9 through 15 bits, replace the 16's on lines 1 through 3 with the desired number of bits and omit line 4.

Table 4-2: Pseudo-code For Logarithmic Conversion To Unsigned Integer

```
(1) if sign is negative or exponent is negative, then amount_to_shift = 16
(2) else if exponent is > 16, then amount_to_shift = 0, mantissa = all 1's
(3) else, amount_to_shift = 16-exponent
(4) if amount_to_shift bit 4 is set, right-shift 16 bits
(5) if amount_to_shift bit 3 is set, right-shift 8 bits
(6) if amount_to_shift bit 2 is set, right-shift 4 bits
(7) if amount_to_shift bit 1 is set, right-shift 2 bits
(8) if amount_to_shift bit 0 is set, right-shift 1 bit
```

4.2 AM modulation and demodulation

The AM modulator's heart is a frequency synthesizer. This synthesizer was implemented by Michael Debole using the floating point conversion functions in Appendix A. The design of the synthesizer consists of two parts: a phase accumulator, a program counter, and a sine lookup table. Several frequencies were hard-coded into the transceiver test system and selected via switches. These frequencies provided different clock divisors, different program counter increments, to the synthesizer in order to generate the desired output waveform. The clock divisors were "0.16" representations of the fraction of the input frequency needed to generate the desired output frequency, that is "output_freq/input_freq*65536" (for a 50 MHz input frequency, 1966 would generate a 1.5 MHz frequency). Since the synthesizer's output range was negative one through one, these data needed to be scaled and offset in order to be output through the tested DAC (whose range was 0 through 255). Since both the input and output ranges were bounded by powers of two, simple addition to the floating point exponent could be used to correct the scaling (this scaling resulted in a range of -128 through 128, which could not be directly output to the DAC even after adding 128; the saturation of the floating point to integer module prevented troublesome roll-over, however). The offset calculation required general purpose addition, unfortunately.

The AM receiver consisted of two parts: the band-pass filter and demodulator created using a trough detector as shown in Figure 4-1 The AM receiver was initially written by Justin Ford and later rewritten by myself. The band-pass filter equations are shown in Eq. 4.1 ('Wc' is the center frequency for the filter and 'BW' is the bandwidth). The center frequency for the filter was calculated using Eq. 4.2 (decimation is the clock division required to allow the filter to execute, derived from the maximal latency through the filter and the clock rate). For a given waveform, a trough detector outputs the minimum samples over each period of the waveform. This is determined by noting a change in the sign of the slope of the input waveform. Table 4-3 shows the pseudo-code for a trough detector.

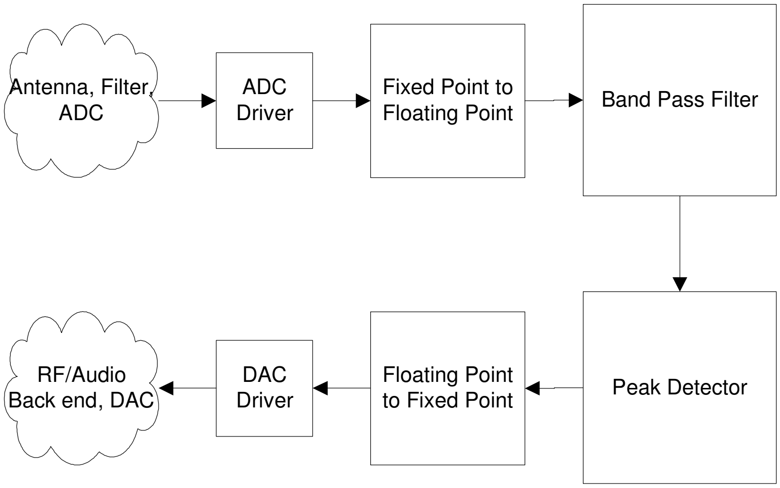


Figure 4-1: AM demodulation testing software

$$a_1 = (BW + 1) * \cos(Wc)$$
$$a_2 = -1 * BW \qquad \textbf{4.1}$$
$$g = 2 * \sqrt{\left|1 - \cos^2(Wc) * (1 + BW)^2 + BW^2 + 2 * BW * \cos(2 * Wc)\right|}$$

$$Wc = 2 * \pi * frequency * decimation / clock\_rate \qquad \textbf{4.2}$$

Table **4-3**: Pseudo-code for a Trough Detector

```
(1) comp_exp = x_exp(n)-x_exp(n-1)
(2) comp_fract = x_fract(n)-x_fract(n-1)
(3) greater_than(n) = NOT((comp_exp < 0) OR ((comp_exp == 0) AND (comp_fract
    < 0))
(4) if NOT(greater_than(n)) && greater_than(n-1) && x(n-3) < 0, then output = x(n-
    3)
```

4.3 Binary Frequency Shift Key modulation and demodulation

BFSK communication, as it was implemented in these investigations, was built upon the foundation of the AM modulator and demodulator. The BFSK modulator was an AM transmitter with few modifications. The BFSK demodulator was a pair of AM demodulators with some added logic to determine what frequency was being transmitted on. The structure of the software used for testing the BFSK modulation and demodulation and AM modulation (used by the BFSK modulator) is shown in Figure **4-2**.

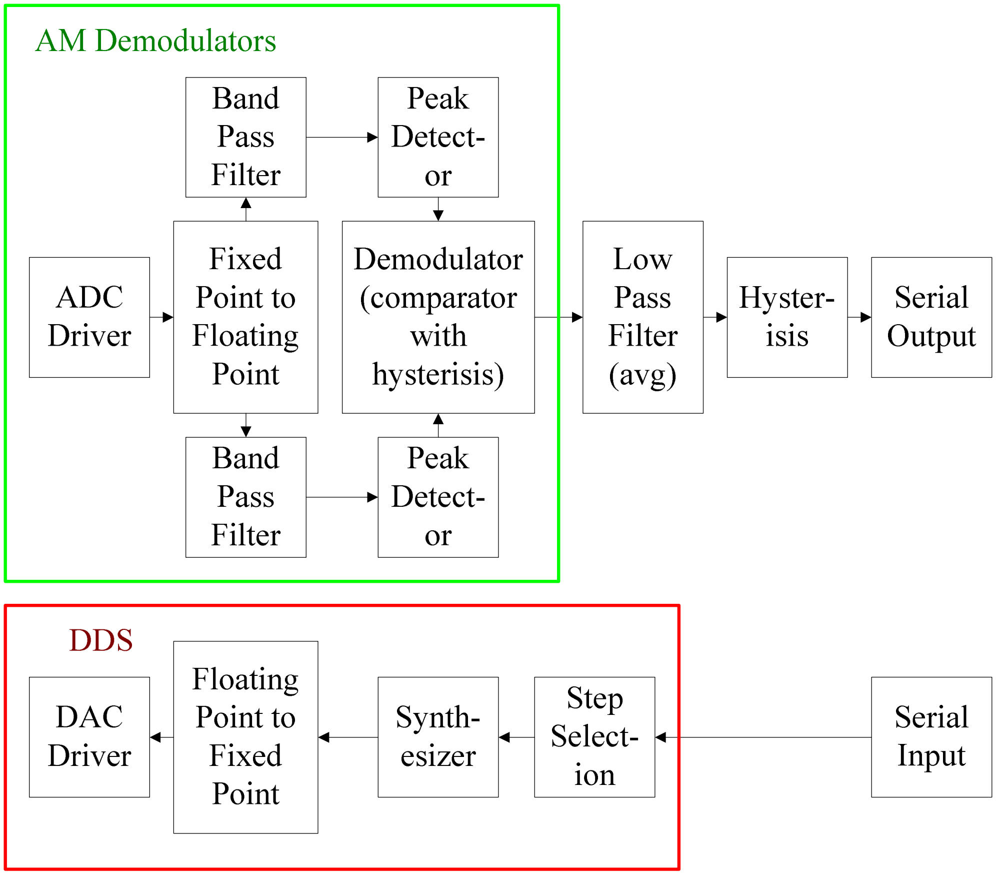


Figure 4-2: BFSK modulation and demodulation testing software

The BFSK transmitter was an AM transmitter. At any one time, the transmitter was either sending on frequency “A” or frequency “B”. So when transmitting a “1”, frequency “A” was generated. Likewise, a “0” corresponded to frequency “B”. Since the input to the AM transmitter is the frequency to generate, this is a simple two-to-one multiplexer connected to the input of the transmitter (a switch that controls what value is provided to the transmitter).

The BFSK receiver required more logic than the transmitter in order to determine what value was being transmitted. Though the transmitter could accomplish data transmission using a single AM transmitter whose transmission frequency varied, the receiver required two full AM receivers. One of the receivers was tuned for frequency "A" and the other was tuned to frequency "B". Each of the receivers generated an output level; hence, these output levels needed to be combined in order to determine whether the value being transmitted was a "0" or a "1". This comparison logic is shown in Table 4-4 ("A" and "B" are the AM receiver output values in floating point). Note that this logic implicitly possesses hysteresis, yet uses relative magnitudes.

Table 4-4: BFSK Demodulator

(1) magnitude_difference = A.exp – B.exp
(2) next_state = A.exp > B.exp
(3) if magnitude_difference is greater than 1 or less than -1, then current_state = next_state
(4) else, current_state remains unchanged

4.4 Phase Locked Loop design

In its initial form, the PLL contained Complex (real and imaginary valued) multiplication, a trigonometric function, a filter, and a Digitally Controlled Oscillator (DCO). This system also relied on Complex input values. In order to make this system more amenable to implementation, two problems needed to be overcome: reducing the complexity of the PLL itself and deriving Complex input from a Real signal (reading a Complex input from an antenna would require two ADCs and additional circuitry). Figure 2-2 shows the original structure of the PLL.

The original design was retained in large part. The complex multiplication was retained. The ultimate goal of the upper half of above diagram is to generate a phase error. The inputs and outputs of the multiplication are shown in Eq. 4.3. If the input and the output of the PLL are not complex (including both real and imaginary components), the cross product terms to not cancel out and the resulting equation is difficult to solve. The Low Pass Filter in Figure 4-3 is used as an integrator to accumulate the phase error and generate the output frequency, rather than to eliminate any high frequency noise. The DCO also includes an integrator. This integrator accumulates the frequency to generate a phase that is used in conjunction with a lookup table to generate the output frequency. Note that the phase error is not the output from the DCO's integrator, as the phase error is zero when the PLL is locked while the DCO's integrator will output a cyclic phase when the PLL is locked.

$$e^{j2\pi\Phi} \times e^{-j2\pi\Theta} \equiv \cos(\Phi - \Theta) + j \times \sin(\Phi - \Theta) \qquad \mathbf{4.3}$$

The arctangent function was not retained in my implementation. Though the arctangent function can be implemented using a lookup table, a better solution would be to not implement the PLL using the arctangent function at all. After all, the best implementation of a complex function is to never implement it. Rather than use an arctangent function, a function that served the same purpose, but was far easier to implement, was used. The goal of the arctangent function is to generate an error value that decreases to zero when the PLL achieves phase synchronization. In addition to this, the function should have a positive output when the phase difference is positive and a negative output when the phase difference is negative. The function that replaced arctangent was Matlab's "imag()" function (whose output was then divided by four). This function extracts the imaginary coefficient of the imaginary component of a complex number. Figure 4-3 shows the comparison of my chosen function to Harris'. Note that my chosen function has an unstable point of equilibrium when the actual phase difference is positive or negative π. This would not cause problems in practice due to noise on the input signal (numerical noise within the system would most likely dislodge it from this point of equilibrium). Since this error function does not require the real component of the complex multiplication mentioned in Eq. 4.3, the "complex" multiplication requires two multipliers and one adder, rather than four multipliers and two adders that are needed to calculate both the real and imaginary components.

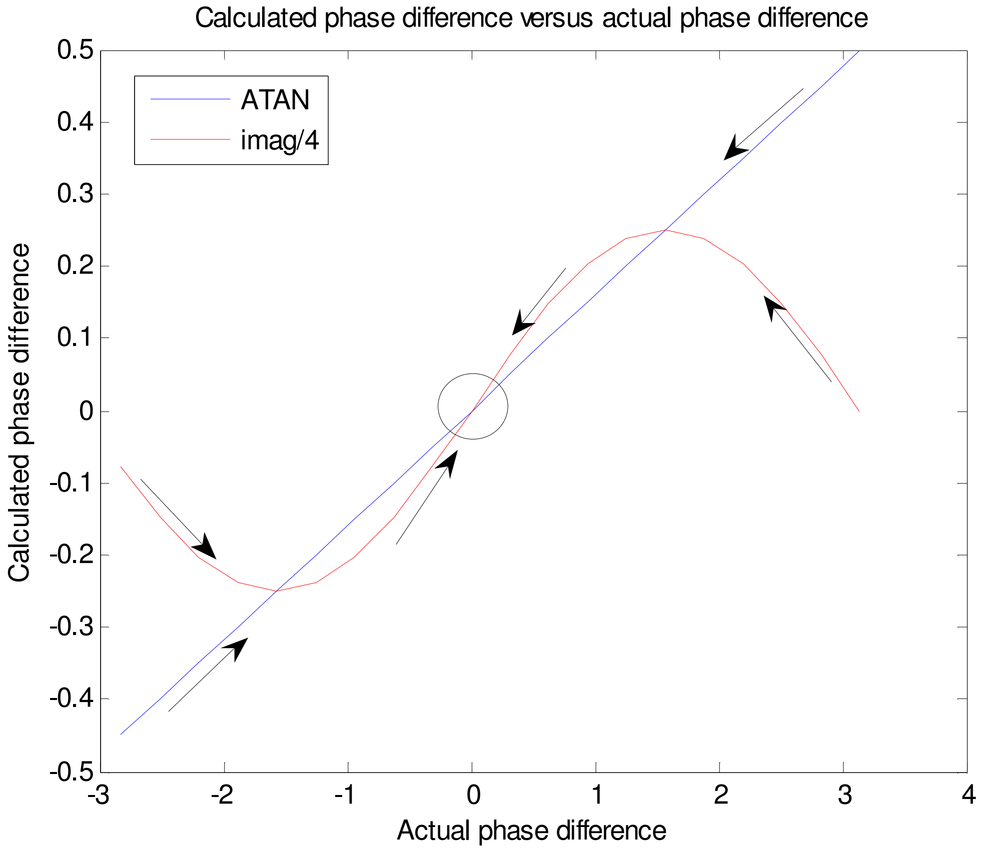


Figure 4-3: Comparison between arctangent and imag()/4

The elimination of arctangent was nice; however, the generation of the complex component (known as the analytic component) of the input was of critical importance. The most commonly suggested way of creating the complex component is the Hilbert transform [9]. This transform has two major drawbacks: the first drawback is that it is a FIR filter, requiring many multipliers, and the second drawback is that it cannot be implemented convincingly [10] (shown in Johansson's figure 4.1). Two observations led to a phase shifter that worked well: the sine of the arccosine of the input yielded perfect output magnitude, though the polarity was untrustworthy, (note that the input range must be normalized, ranging from negative one to positive one) and the derivative of the input has the correct phase, its polarity was correct. As can be seen in Figure 4-4, this first attempt at a non-Hilbert transform based phase shifter nearly worked. The discontinuities in the shifted output result from a zero or nearly zero derivative of the input at the peak and trough of the input, the derivative (calculated in discrete time) lagged behind. Using a linear extrapolation of the calculated shifted phase could have yielded a usable phase estimate; a simpler approach to determine the output polarity was chosen: use the current PLL output. Using the PLL's phase output only when the derivative was close to zero yielded good results, though this would result in an unstable system if used as the sole polarity determination tool. Note that when the derivative is close to zero, the PLL's phase output unambiguously dictates the output polarity for the phase shifter. Table 4-5 shows the pseudo-code for the phase shifter.

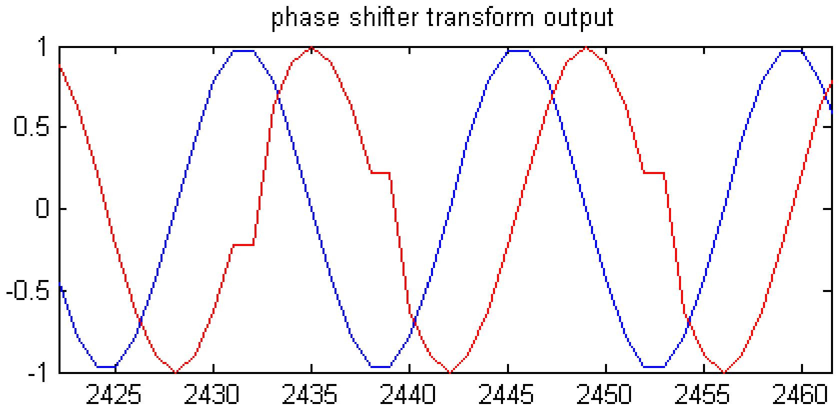


Figure 4-4: Phase shifter, first attempt

Table 4-5: Pseudo-code for the working phase shifter

```
(1) if abs(derive(input)) < 0.25, then
(2) if PLL phase is between 0 and $\pi$, then shifted_input = sin(acos(input))
(3) else shifted_input = -sin(acos(input))
(4) else shifted_input = sin(acos(input)*-sign(derive(input))
```

Though the basic structure of the PLL was retained, their implementations were modified significantly. The unadulterated components of the base design were the integrator and DCO. The modified implementation of the PLL can be seen in Figure 4-5.

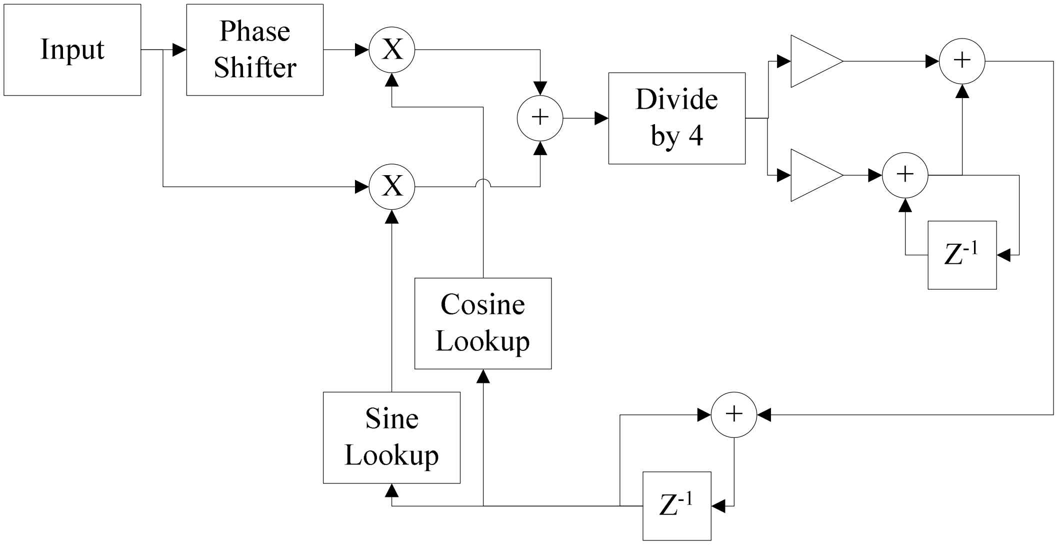


Figure **4-5**: Modified PLL

Though this PLL worked perfectly well in a Matlab simulation, working in double-precision floating point was not considered to be desirable. As described previously, floating point (whether reduced, single, or double precision) is an expensive proposition. If a PLL could be implemented using fixed point, it should. The first and most important step in the conversion of this system to fixed point was to determine the dynamic range of all the component signals. Figure **4-6** shows my analysis of the system signals.

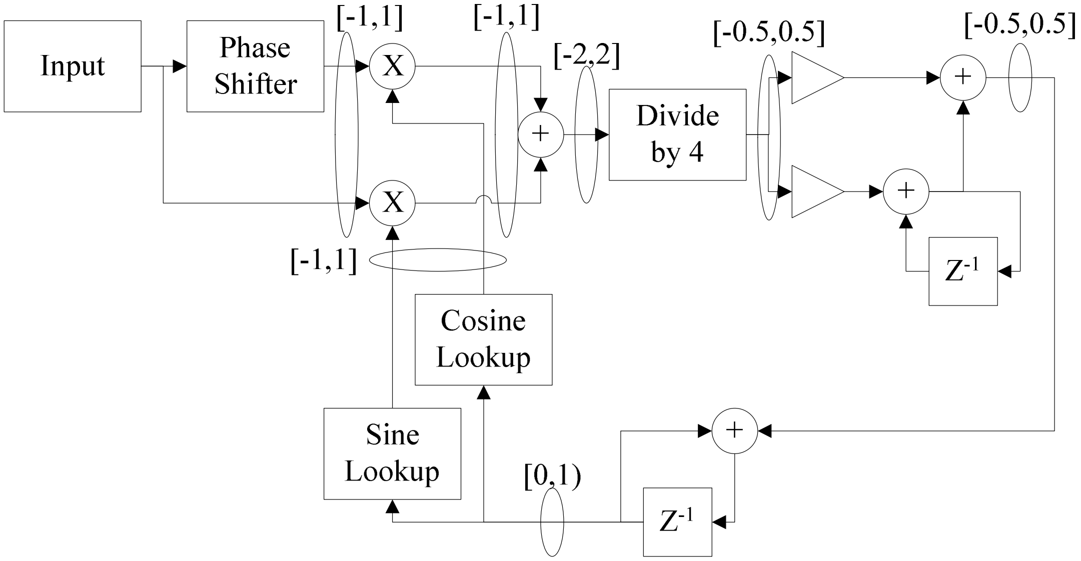


Figure 4-6: Dynamic range analysis of the PLL

Implementing these ranges in fixed point requires a bit of imagination. The divide by 4 required by the arctangent approximation is, in fact, no operation at all. The location of the decimal point in fixed point is nothing more than a matter of interpretation. I, therefore, choose the data format prior to the division to be “2.13” signed fixed-point prior to the “division” and choose the data format to be “0.15” signed fixed-point after the “division”. The next data ranges to be distorted are the ranges feeding into and flowing out of the DCO (the integrator that feeds the sine and cosine lookup tables). First of all, the output from the integrator is a normalized phase in the range of [0, 1). The original model’s integrator also had a range of [0, 1), which was scaled to [0, 2*π) when calculating the sine and cosine values. This range lends itself to a fixed point representation of “0.16”, unsigned. In order to determine a realistic range for the signal feeding the integrator, some understanding of the meaning of the signal is required. This signal is the frequency feeding the DCO. Where this system implemented in continuous time and infinite precision, this signal has the range [-∞, +∞]. The model that this work was based upon was a discrete time model. Since this model was a discrete time model, the frequency request was in the normalized frequency range of [-0.5, 0.5]. This range suggests a “0.15” signed fixed-point representation.

Chapter 5

Experimental Results

A number of tests and experiments were performed on the software defined radio components in order to determine whether they worked or not and to what degree. The floating point conversion was tested in an ad hoc manner. The AM demodulator was tested using a short range transmitter. The BFSK modulator and demodulator were tested by assembling two complete systems and transmitting data between them. The PLL was tested in a series of simulations.

5.1 Floating point conversion testing

Testing on the floating point conversion module was implemented prior to the implementation of the floating point addition module and was tested at that time. As mentioned in 5.1, this module was initially prototyped in "C". At that time, only ad hoc testing was performed on the "C" code implementation. It was only after the FPGA interfaces to an ADC and DAC that the real testing began.

After developing the communication modules for the AD9410 evaluation board mentioned in 2.2.2 (and discovering the design flaws in that evaluation board), testing began shortly on the conversion modules. A series of loop-back tests were performed using a function generator. Initially, the interface to the evaluation board was tested by connecting the input module to the output module. This is to say that the ADC data was read into the FPGA and written back out to the DAC. By observing the DAC output using an oscilloscope, the timing errors could be corrected. Extending this test to include the floating point conversion code enabled the conversion from an integer format to floating point and back. After a way of viewing the results was implemented using the DAC, this test was able to be performed satisfactorily. After all, simulations may give you an impression that code should work in practice, but you can only be sure once it is integrated into a working system (timing errors and all). Due to the structure of this conversion code, it can be pipelined to a high degree. Though a strict maximum speed was never investigated, this conversion code could only add phase delay to the communication systems implemented using this conversion code. The code was pipelined with three internal pipeline stages and registered output. In this configuration, the implementation was able to keep up with the 200 MHz sampling rate of the AD9410 evaluation board. A four stage pipelined implementation with registered output was written, but was consumed more logic and was deemed unnecessary.

5.2 AM demodulator testing

Two experiments were performed on the AM demodulator, after testing the logic to ensure its functionality: frequency selectivity testing and AM reception. In order to test the frequency selectivity of the band-pass filter, a frequency sweep was performed using a frequency generator after selecting a specific frequency and bandwidth for the filter. The AM reception testing was performed using a Heath Kit™ AM transmitter powered by a 9V battery.

The frequency sweep used a frequency generator connected directly to the ADC input of the AD9410 evaluation board and the board's DAC output was measured by an oscilloscope. The FPGA evaluation board connected to the ADC evaluation board was the Xilinx Spartan®-2 evaluation board from Digilent. The frequency selected was 600 KHz with a selected bandwidth of 0.9995. The input voltage to the ADC was 740 mV, peak to peak; and, the output voltage from the DAC was 220 mV peak to peak at its maximum. Measurements were taken every 100 Hz from 598 KHz through 602 KHz. Both a single filter and two cascaded filters were tested, experimentally verifying the theoretical results described in [11]. The results are shown in Figure 5-1.

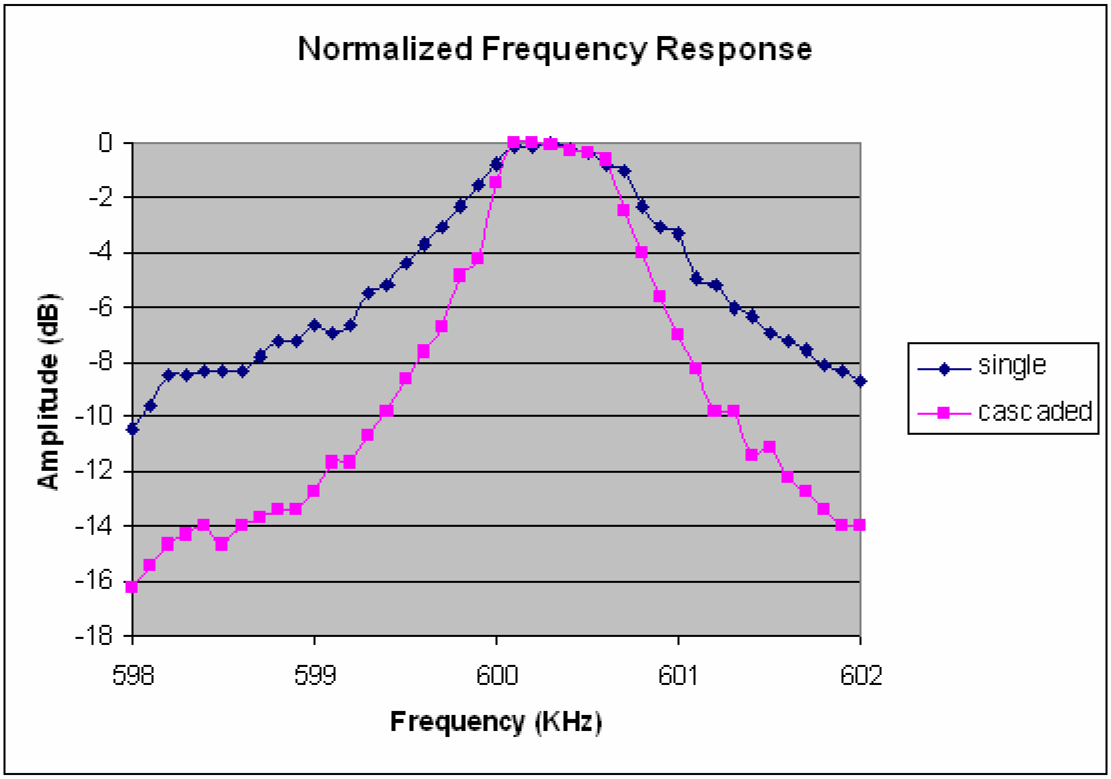
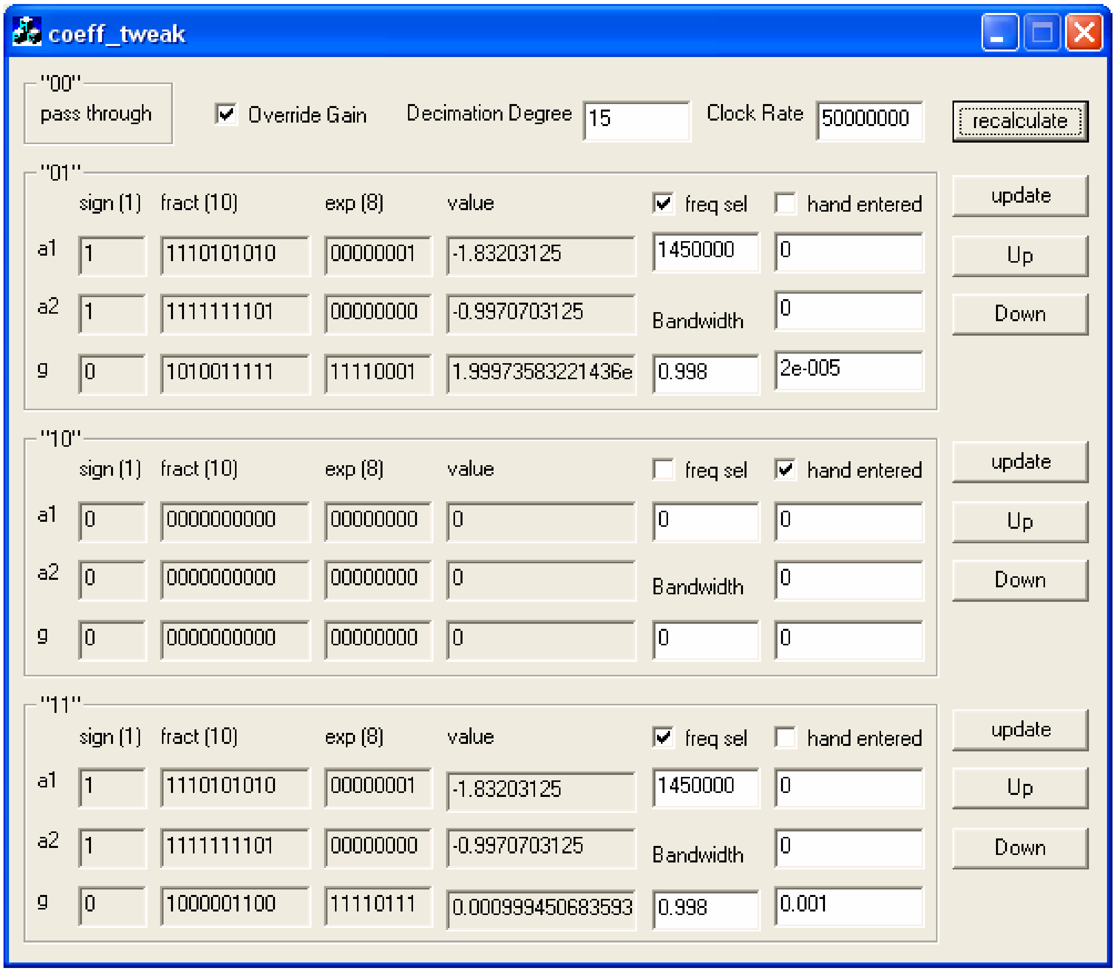


Figure 5-1: AM demodulator filter testing

The second test on the AM demodulator involved using an AM transmitter to test the trough detector combined with the filter to form a complete AM receiver. Each combination of the ADC boards and DAC chips tested had a version of the AM demodulator created for it. Though some have speculated that a live radio station was able to be heard using the AM demodulator, the signal to noise ratio was sufficient to describe those claims as “speculative”. The AM transmitter was sufficiently strong to overcome nearly any system noise level. With this AM transmitter, the radio could be demonstrated to be tuned and properly receive the transmitted signal. Testing was performed using tone generation software and an Internet radio station (http://www.sky.fm/classical). The best results were obtained using the inductor based antenna mentioned in **2.2.1**. An analog radio was able to significantly out-perform the software defined radio, unfortunately. The analog radio was a dollar store variety of the same brand and model that the inductor based antenna was harvested from. The analog radio had better reception of the transmission at approximately twice the distance as the software radio.

In these tests, the software define radio’s tuning parameters were set using a configuration application. This application communicated with the radio via a simple serial communication protocol. The protocol consisted of three bytes: the exponent (eight bits), the least significant eight bits of the fractional component, and the last byte contained the most significant two bits of the fractional component as well as the sign bit and four bits to select what coefficient was to be configured. The four selection bits were broken down into two fields: the coefficient bits (“00” for “a1”, “01” for “a2”, and “10” for “g”) and the coefficient group. The toggle switches on the board selected which coefficient group was to be used for demodulation. These could be considered to be the "presets" on a car stereo. The configuration application can be seen in Figure 5-2



"00"
pass through

☑ Override Gain

Decimation Degree 15

Clock Rate 50000000

"01"

	sign (1)	fract (10)	exp (8)	value	☑ freq sel	☐ hand entered
a1	1	1110101010	00000001	-1.83203125	1450000	0
a2	1	1111111101	00000000	-0.9970703125	Bandwidth	0
g	0	1010011111	11110001	1.99973583221436e	0.998	2e-005

"10"

	sign (1)	fract (10)	exp (8)	value	☐ freq sel	☑ hand entered
a1	0	0000000000	00000000	0	0	0
a2	0	0000000000	00000000	0	Bandwidth	0
g	0	0000000000	00000000	0	0	0

"11"

	sign (1)	fract (10)	exp (8)	value	☑ freq sel	☐ hand entered
a1	1	1110101010	00000001	-1.83203125	1450000	0
a2	1	1111111101	00000000	-0.9970703125	Bandwidth	0
g	0	1000001100	11110111	0.000999450683593	0.998	0.001

Figure 5-2: AM demodulator configuration application

5.3 BFSK modulator and demodulator testing

Unlike the AM receiver which received its frequency and tuning parameters via a serial port configuration application, the BFSK modulator and demodulator was configured via switches. Two switches on the evaluation board were used to determine the transmitting frequencies and two switches for the receiving frequencies. The frequencies used were 1500 KHz/1530 KHz, 1530 KHz/1560 KHz, 1560 KHz/1590 KHz, and 1590 KHz/1620 KHz (lower frequency/higher frequency). There was a maximum deviation of 0.3 KHz in the frequency generator (due to precision of the counter used to generate the frequencies). Since the frequency pairs for transmitting and receiving could be set independently from one another, the initial testing on the bench was a simple loop-back test using a single board. Though this demonstrated that the system could work, this test did not say anything about how fast the system could transmit. This test would not even require a transmitting or receiving antenna since a significant portion of the transmission was passing through the ground plane of the system and other hard-wired connections.

True wireless communication was achieved. The hardware used for each of the two systems tested was the following: a Spartan®-3 evaluation board, a PC, an analog front-end using an inductor based antenna, an ADC evaluation board (ADC10321), a DAC (TLC7524), and analog back-end also using an inductor based antenna. With these two systems, a maximum data rate of 4800 baud was observed with a system separation of no less than three feet. At this data rate approximately one kilobyte of data was able to be sent with no apparent loss of information. At higher data rates, UART desynchronization rendered a precise measurement of bit error rate difficult if not impossible; once a UART looses track of where bytes start and stop, the transmission is meaningless until it resynchronizes (this could take anywhere from one byte to dozens).

5.4 PLL simulation

Two types of Phase Locked Loops were simulated: an infinite (floating point) precision PLL and a fixed point precision PLL. The simulations included tests of frequency locking and tracking and phase tracking. The simulations could also introduce White Gaussian Noise into the input. The infinite precision PLL is diagramed in Figure 5-7.The finite precision PLL is based on the analysis in that figure. All ranges were scaled up to 16-bit fixed point with multiplications being carried out in floating point prior to being truncated back to 16-bit fixed point. The simulations were run in Matlab.

The infinite precision simulations demonstrate that the PLL works. As can be seen in Figure 5-3, the PLL can lock and track input frequencies within a fairly broad range of input frequencies. Note that in the first step the PLL locks on to a negative frequency, rather than the correct, positive, frequency. Though annoying, this ambiguity does not affect the quality of the output since detecting and correcting a negative frequency is trivial. Though the simulation is not shown, attempting to simply use the absolute value of the frequency did result in an unstable PLL which suggests that the PLL really did want to lock on to the negative frequency.

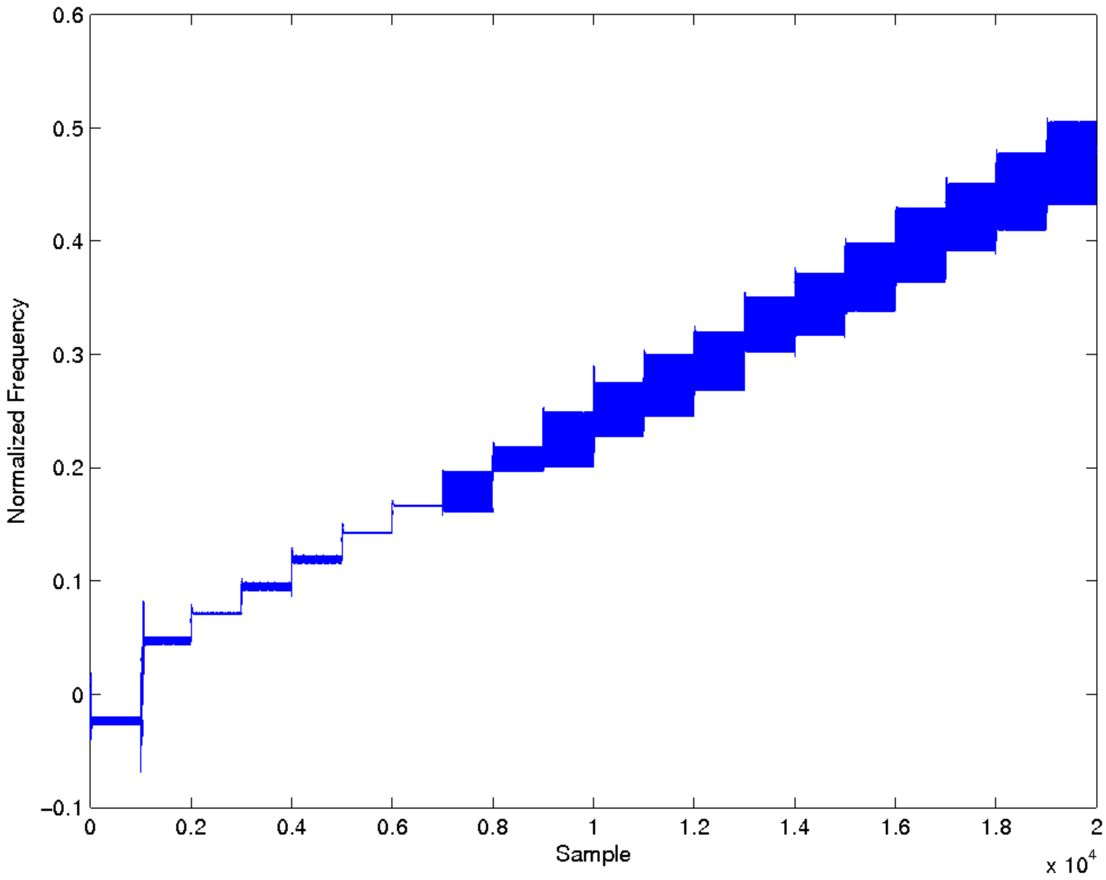


Figure 5-3: Frequency locking and tracking

Though the PLL can track a frequency the speed of the lock directly affects the transmission rate data using the PLL (though the quality of the lock has a greater affect on the data rate, the infinite precision simulation cannot effectively address that metric). Shown in Figure 5-4 is a single step from one frequency to another. As shown in this figure, the lock is achieved in approximately 50 samples.

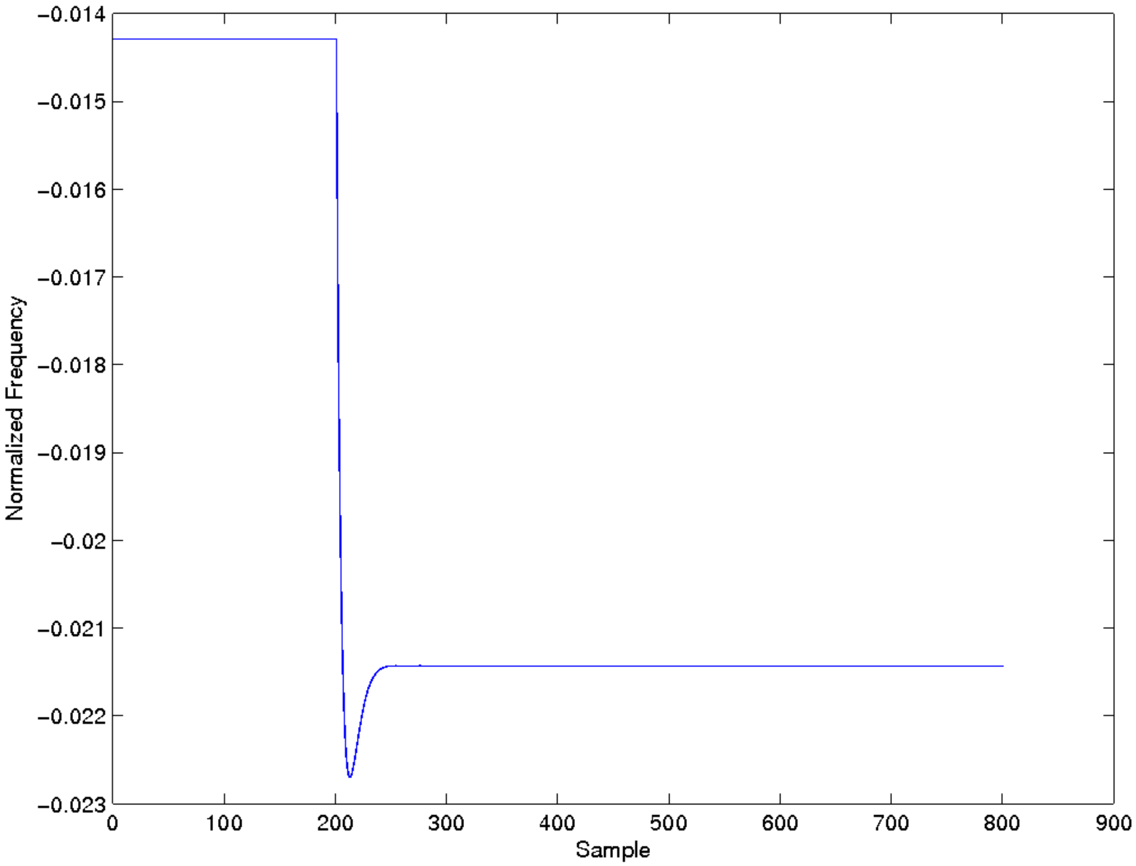


Figure 5-4: Locking speed when stepping from one frequency to another

While frequency tracking is one of the desirable characteristics of the PLL, the ability of a PLL to track phase allows it to not only implement Frequency Shift Keying but also to implement Phase Shift Keying. Figure 5-5 shows a short simulated input sequence provided to the infinite precision PLL with a very drastic phase shift of 180 degrees. Figure 5-6 demonstrates that when the PLL encounters the phase shift the PLL does lock on to the input's new phase.

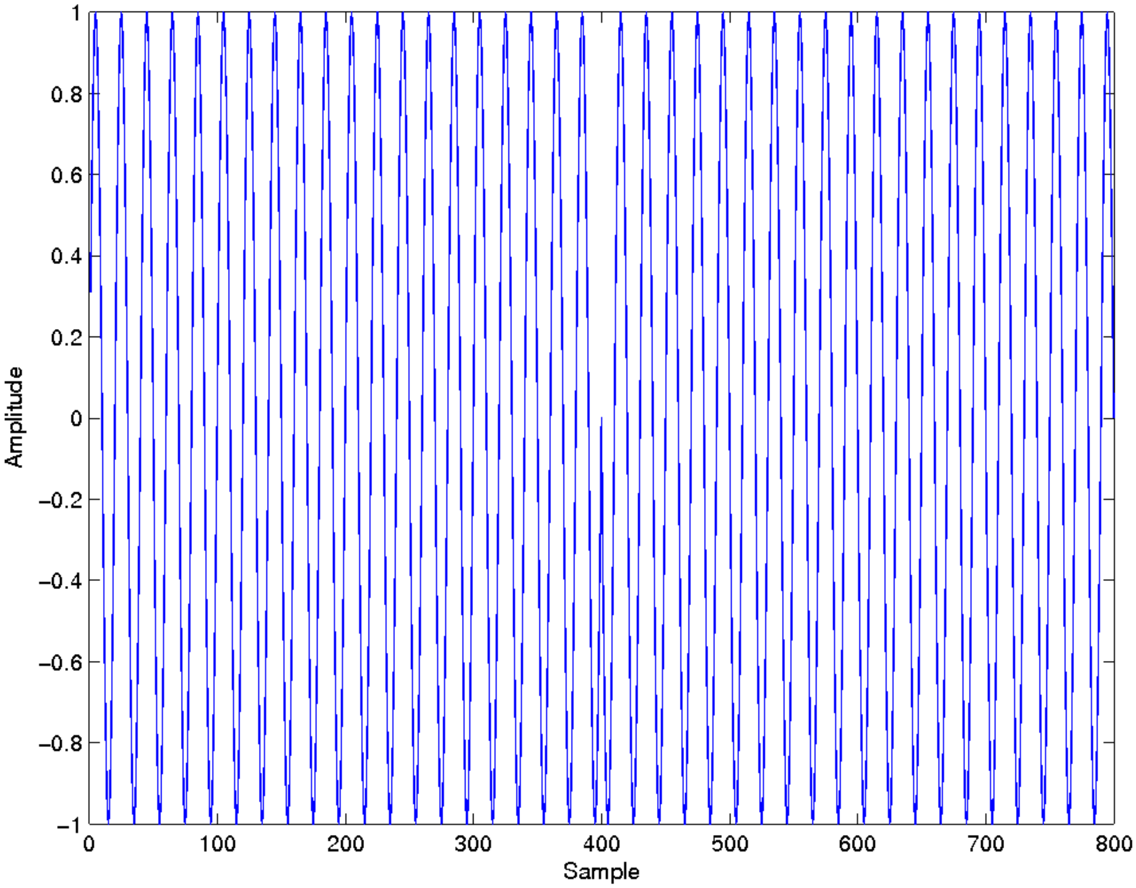


Figure 5-5: Phase shifted simulation input

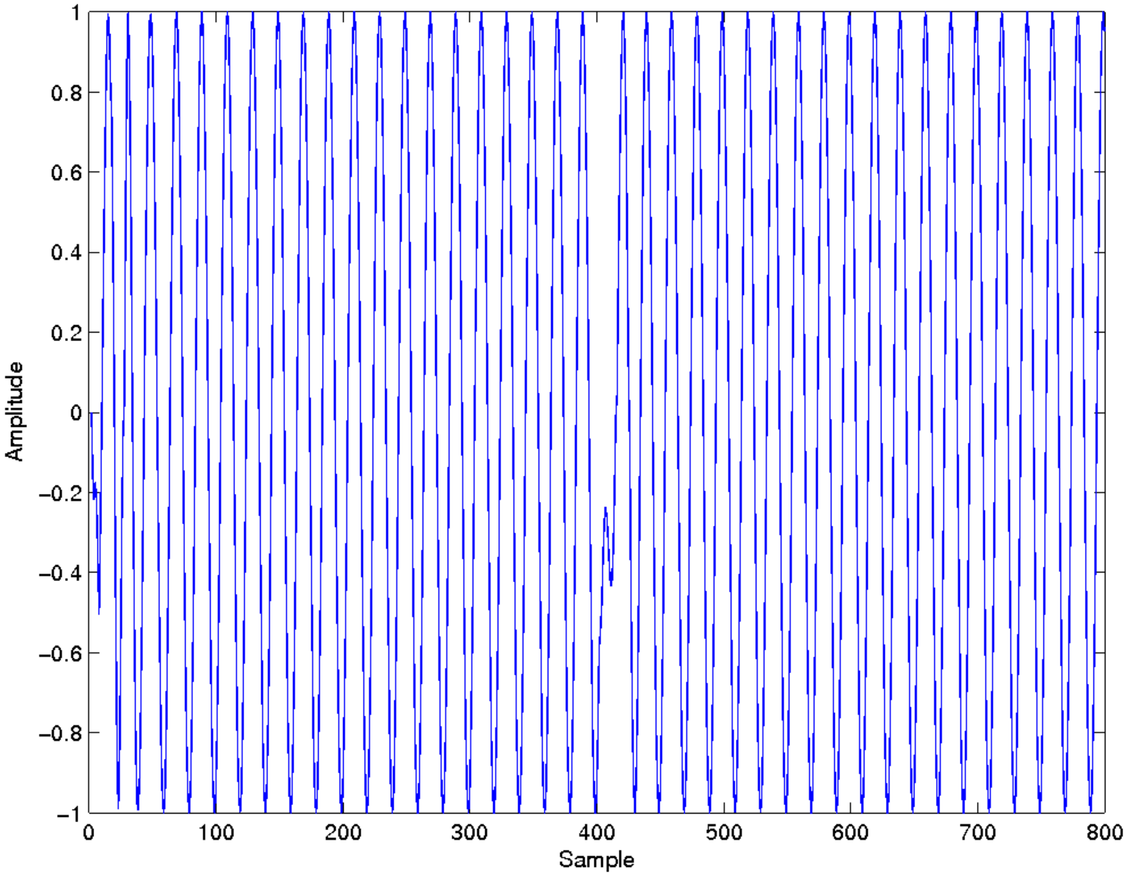


Figure 5-6: PLL tracking of the phase

Both finite and infinite precision simulations were performed to verify that the algorithm can be implemented. This was done to ensure that a fast and relatively small PLL could be made using this algorithm. Assuming that the quality of the PLL could be maintained, using fixed point arithmetic would increase the bandwidth and decrease the size of the PLL. Figure 5-7 shows the result of a frequency sweep of a finite precision PLL based on the analysis in Figure 5-7. A phase step simulation was also performed, as shown in Figure 5-8.

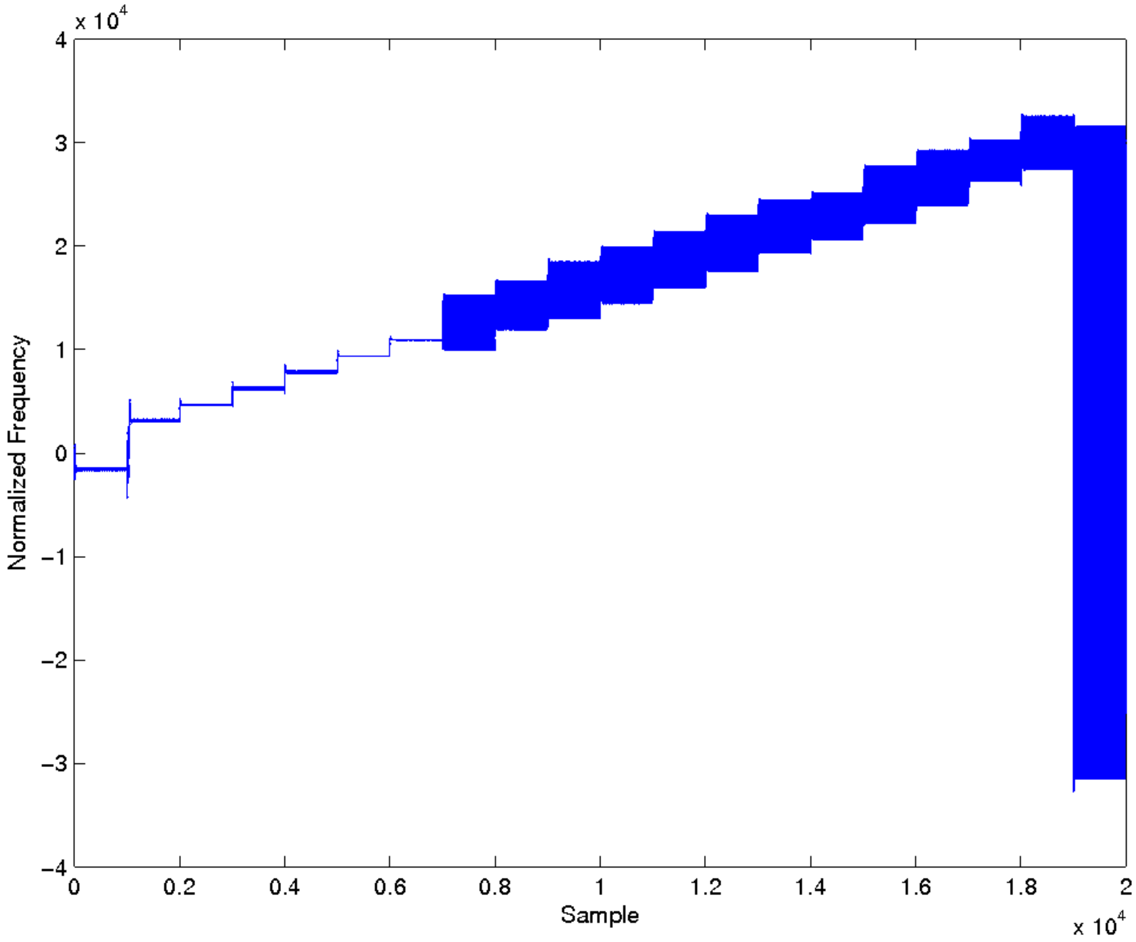


Figure 5-7: Finite precision PLL frequency sweep

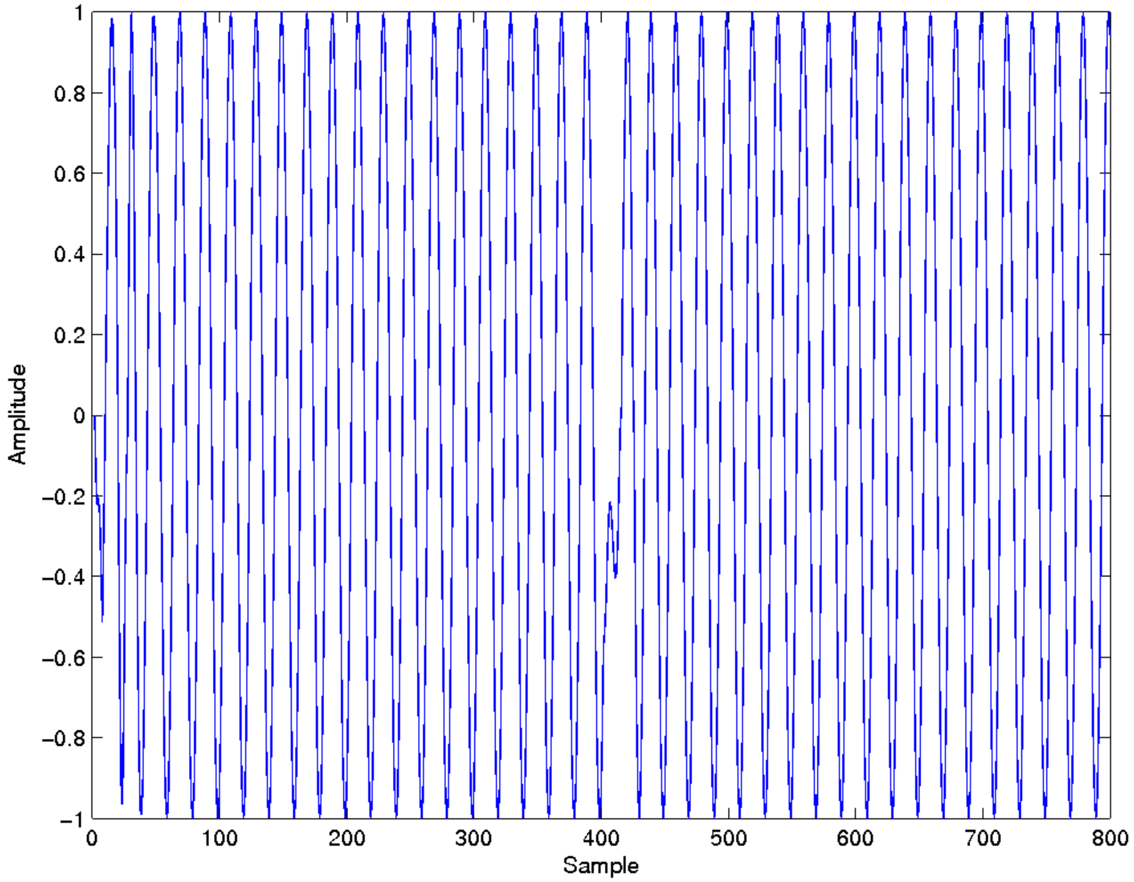


Figure **5-8**: Finite precision PLL phase step

While neither the simulation in Figure **5-7** nor the simulation in Figure **5-3** is stable above a certain frequency, some information can be obtained in that region. By applying a 64 sample averaging filter to the frequency output (basic noise reduction), the output can be rendered much more stable. Note, however, that this simple filter can not be placed inline in the PLL due to the filter's associated phase shift. This phase shift would dramatically reduce the stability of the filter. The results of applying this simple filter to the finite precision simulation frequency output are shown in Figure **5-9**

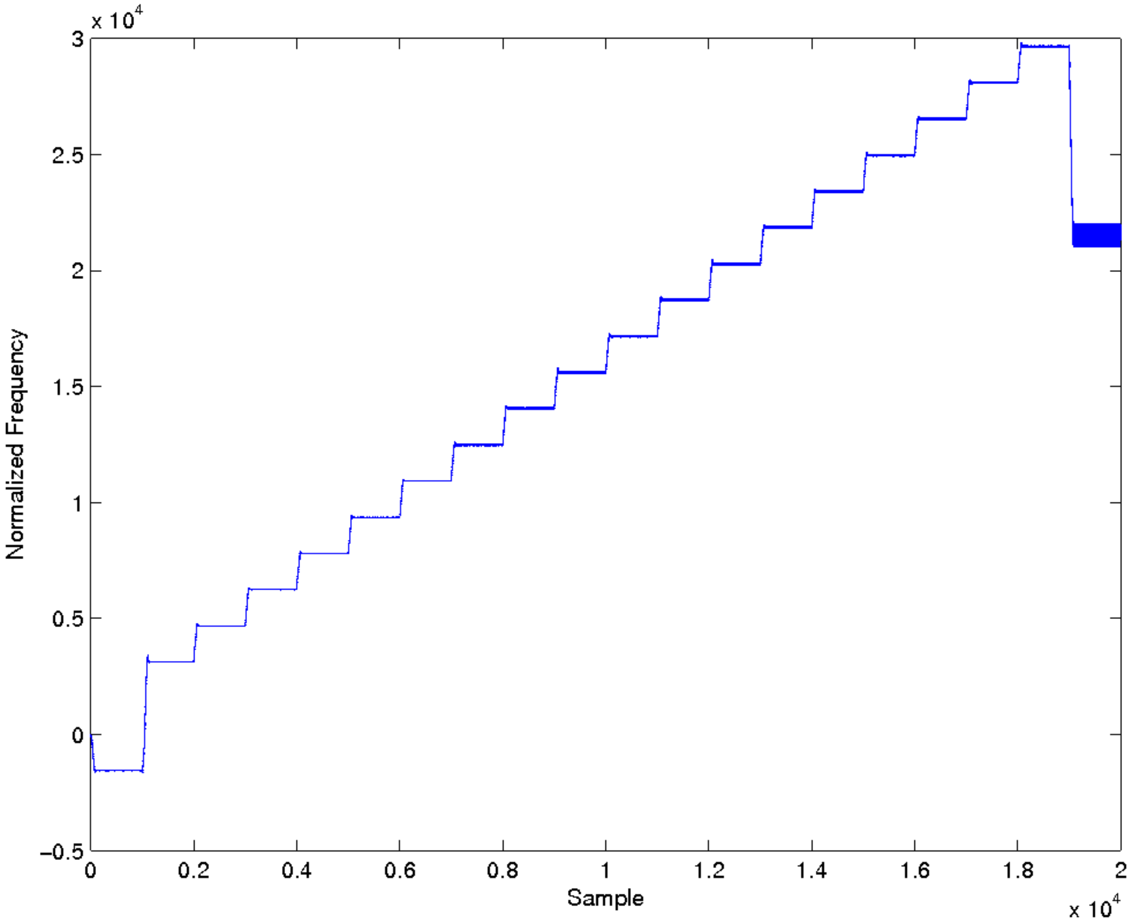


Figure **5-9**: Averaging filter applied to the frequency output of the finite precision PLL

As mentioned previously, the PLL directly encodes the frequency of the received signal. This signal says nothing about the phase of the received signal other than that the input signal has a constant phase change over time (this is the definition of frequency, after all). This is not to say, however, that the PLL cannot be used to track the phase. A simple subtraction of the tracking phase from the reference phase will directly yield the phase-encoded data of the transmission. Figure **5-10** shows the result of a reference and tracking PLL. Prior to the phase shift, the difference between the two phases is insignificant. After the phase shift of 0.5 (normalized phase), the distance between the two is 0.5. Note that while there appears to be a point where the two phases are equal (at 0.5), this is merely an artifact of the simulation plot. This intersection occurs at the discontinuity when one of the phase accumulators rolls over and is, therefore, not an intersection at all (the plot would more accurately be shown as a series of ramps, rather than a series of saw-teeth).

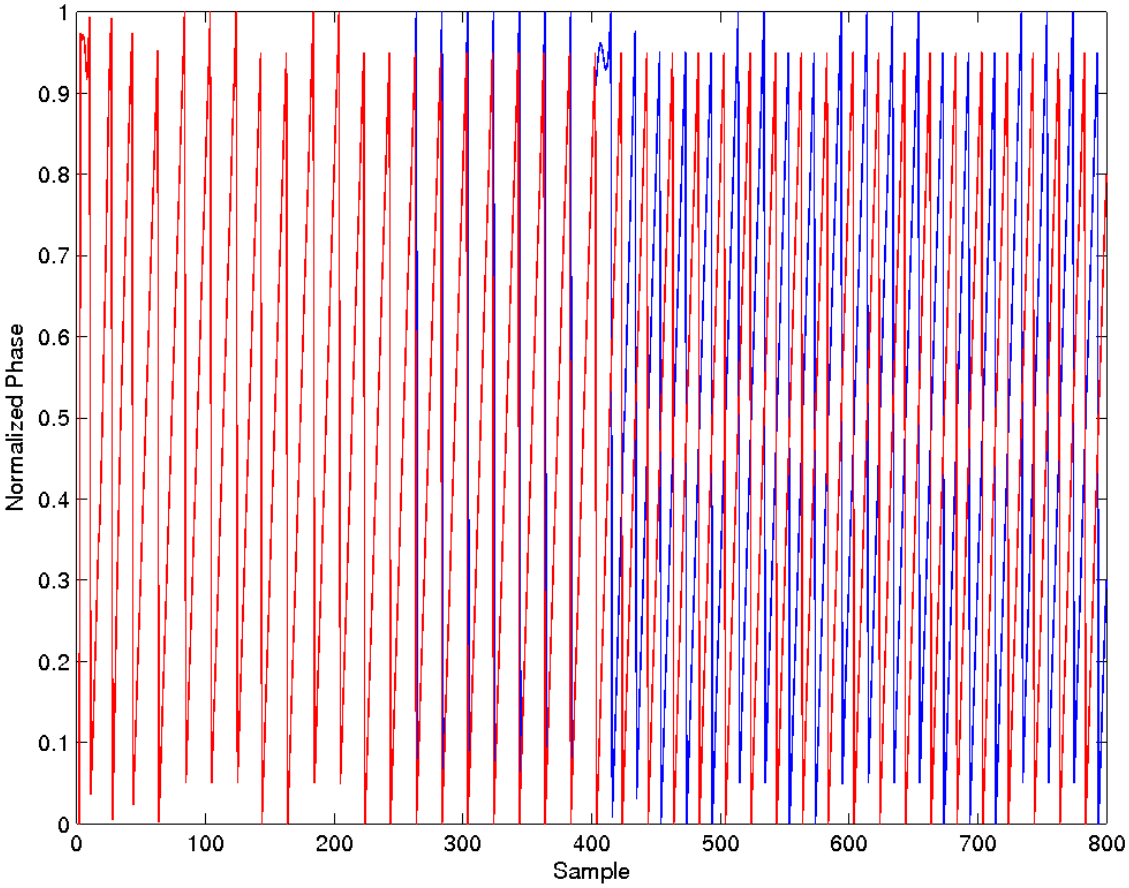


Figure **5-10**: Reference and tracking PLL simulation

When a phase change is encountered by the PLL, there is a discontinuity in the received frequency. Though this discontinuity does not show what the phase is, it does indicate that the phase has changed. An instantaneous change in phase is, of course, a drastic frequency change. Even a small change in phase will have some affect on the frequency. Figure 5-11 shows the disturbance in the infinite precision PLL simulation.

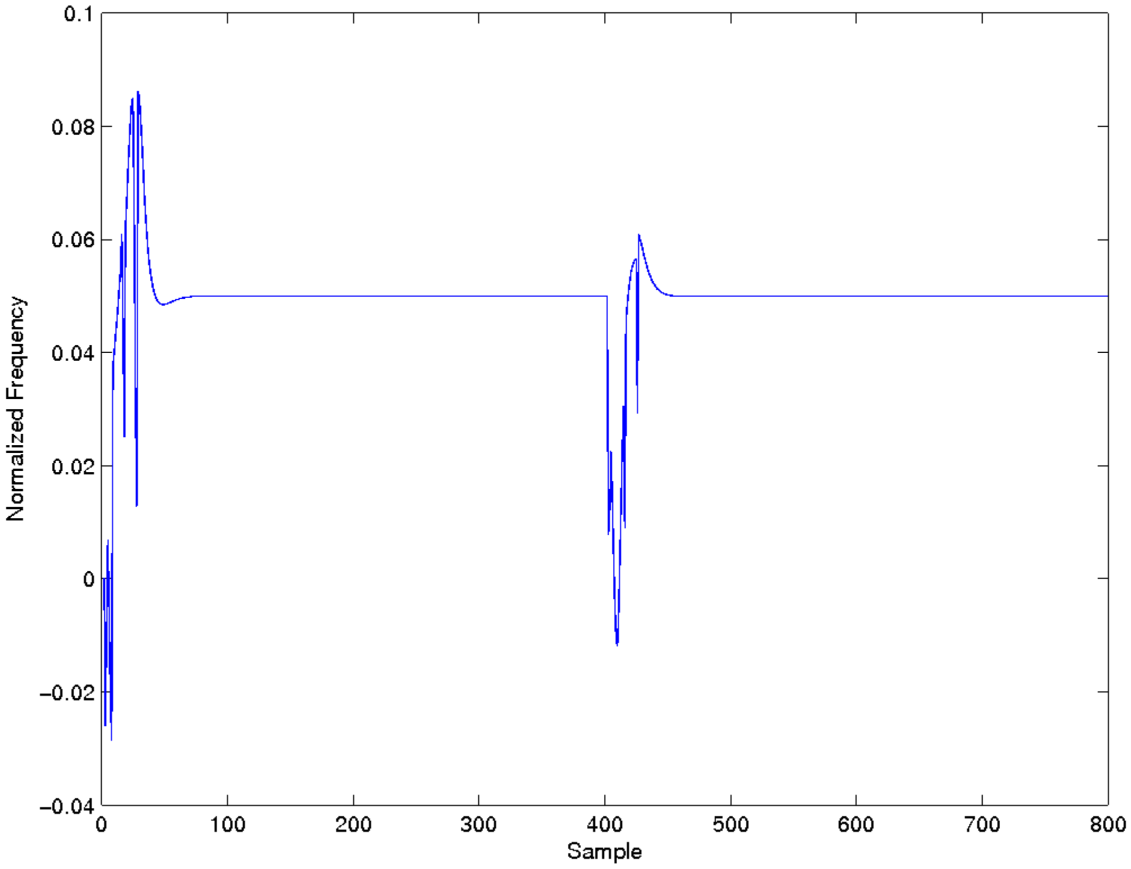


Figure **5-11**: Infinite precision PLL simulation frequency response during a phase step

A simulation of the finite precision PLL shows the disturbing result in Figure 5-12. The disturbing nature of this plot stems from the instability seen after the frequency has been locked. This noise is not apparent in the infinite precision simulation due to its greater precision and dynamic range. The frequency used in this simulation cannot be perfectly represented in binary: 0.1 normalized frequency. Since the number system used to represent this frequency is simple fixed point, the dynamic range limitation of this representation will have problems with this particular frequency (it is not making efficient use of the bits that it has to work with). The resulting instability is sufficient to ruin this particular implementation as a reference PLL.

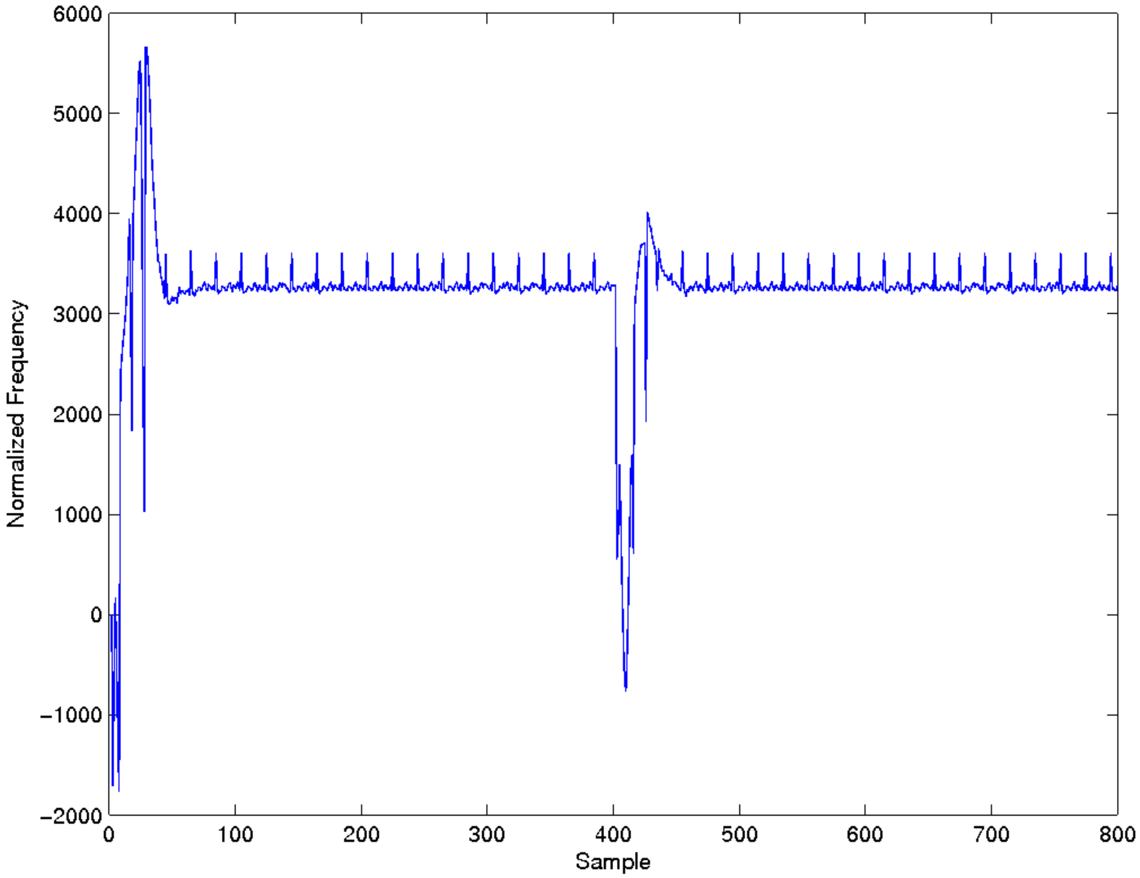


Figure **5-12**: Finite precision PLL simulation frequency response during a phase step

Chapter 6

Conclusions

From the experiments performed, several conclusions can be made about the software defined radio created and the PLL that was simulated. The signal to noise ratio of the software defined radio can be improved. The PLL's stability is insufficient for use in phase shift keyed communication but can be improved. There are also several avenues of research that need to be explored.

Noise is the bane of the software defined radio. The analog front-end was particularly prone to noise. Receiving commercial broadcast radio seemed to be difficult if not impossible with the original analog front-end. Further research is needed to determine if the analog front-end is capable of receiving a commercial broadcast or if it has been lost in the background. The first step would seem to be moving the analog front-end off the breadboard and onto a real circuit board.

The instability shown in Figure **5-12** prevents the PLL from being used in phase shift keying (as the phase reference), but improvement of the finite precision PLL can be accomplished in a number of ways. The first would be to use more floating point arithmetic so that the reference PLL is more accurate. By leaving the tracking PLL as a fixed point system, the over-all system complexity can be controlled but possibly at the cost of performance. Increasing the reference PLL's precision by extending it from a 16-bit to 24-bit accumulator should help and has the added benefit of retaining a complete fixed-point system. If the range of the PLL is limited to a lower frequency band, the fixed point number system can be more precisely tuned for operation in that band. This would limit the generality of the system while improving its performance in that less-than-general frequency band.

Much like the PLL, automatic gain control (AGC) is a vital piece of many communication systems but is frequently overlooked in academic analyses. Note that the input waveform to the PLL simulations was assumed to be normalized. This assumption stems from the logic described in Table 4-5. The offending portion of that logic is the sine of the arccosine of the input signal. This is only possible if the input signal is normalized to [-1, 1]. This AGC would doubtless be a control loop similar in some ways to the PLL. It would need to adapt to the recent characteristics of the input waveform and be responsive without excessive distortion. Note that the PLL can function with input data that does not swing fully, but its output would no doubt suffer.

Though AM modulation and demodulation were explored in this thesis, other modulation techniques should be explored. Among these other modulation techniques, Frequency Modulation (FM) was of interest. With a working PLL, FM should be trivial to demodulate (modulation requires no PLL, but is quite simple). In-phase and Quadrature modulation and demodulation techniques should also be possible using the PLL [12].

Bibliography

1. S. M. Brown, J. R. Ford, L. M. Naji, and K. Choi, “A Proof-of-Concept Software Defined Radio Receiver Implementation in Xilinx Sparan-IIE FPGA.” The Pennsylvania State University, 2005.

2. S. M. Brown, J. R. Ford, and K. Choi, “A Single Semester Software Defined Radio Transceiver Implementation in a Xilinx Spartan-3 FPGA,” Software Defined Radio Forum 2005, Proceedings of, Nov 2005

3. The Software Defined Radio Forum, *SDR forum YEARBOOK 2005*, Denver, CO, Software Defined Radio Forum 2005.

4. Patel, M. and Lane, P., Comparison of Downconversion Techniques for Software Radio, *London Communications Symposium*, Sept 2000.

5. J. R. Ford and D. J. Miller, “A Radio Frequency Digital Signal Processor Using Software Defined Radio Techniques to Receive Amplitude Modulation at Carrier Frequency.” Master’s paper, The Pennsylvania State University, 2005.

6. F. Harris, “DSP Based Carrier and Timing Synchronization in Digital Modems.” Software Defined Radio Forum 2005, Tutorial Presentations, Nov 2005.

7. J. P. Shen and M. H. Lipasti, *Modern Processor Design, Fundamentals of Superscalar Processors*, New York, NY: McGraw-Hill, 2005.

8. L. Louca, T. A. Cook, and W. H. Johnson, “Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs,” *FPGAs for Cusom Computing Machiines, 1996, Proceedings, IEEE Symposium on,* pp107-116, 1996.

9. S. W. Smith, *Digital Signal Processing, A Practical Guide for Engineers and Scientists*, St. Louis, MO: Newnes, 2003.

10. M. Johansson and B. Nilsson, “The Hilbert transform.” Master’s paper, Växjö University, 1999.

11. L. M. Naji, “Comparison of Finite Precision Bandpass IIR Filter Structures for Software Radio Systems.” Master’s paper, The Pennsylvania State University, 2004.

12. D. L. Jones, et al, “Digital Receivers, Symbol-Timing Recovery for QPSK,” http://cnx.org/content/m10485/latest/, 2005.

Appendix A

Floating Point in "C"

This appendix contains the "C" source code for performing the reduced precision floating point used in the research described in this thesis as well as the "C" source used to run simulations in Matlab with the reduced precision format. This appendix includes the structure definition for the floating point format, conversion functions to and from the floating point format (for integer and floating point), addition operator, subtraction operator, multiplication operator, and Matlab MEX function to import the format into Matlab.

A.1 Structure Definition

The following table is the "C" definition for the reduced precision floating point format (the author of this structure had no shame).

Table A-1: Reduced Precision Floating Point Format Definition

```
/* make sure that fract and exp are less than 30 bits wide (each) */
struct steves_float
{
   unsigned int    sign:1;     /* 1 bit of sign */
   unsigned int    fract:10;   /* fractional bits, unsigned */
   int             exp:8;      /* exponent bits, signed 2's complement */
};
```

A.2 Format Conversion

This section includes a number of conversion functions. These functions convert standard floating point to the reduced precision format, the reduced precision format to standard floating point, signed integer to the reduced precision format, and the reduced precision format to signed integer. A brief description of unsigned integer conversion is also included in the signed integer conversion descriptions.

The following table showing the function that was used to convert standard floating point to the reduced precision format consists of several steps. The first step is the derivation of the sign. The second step of this function is the calculation of “exp_adjustment_count”; this variable is used to both normalize the floating point number for the fixed point reduced precision mantissa as well as to calculate the exponent of the reduced precision representation. The two “while” loops are used to calculate the exponent and normalize the floating point number. The next step is to assign the exponent. The final step is to truncate the floating point value and assign it to the mantissa.

Table **A-2**: Conversion From Standard Floating Point to Reduced Precision Floating Point

```
void conv_float_to_steves_float( float a, struct steves_float * out )
{
   unsigned int    one_too_big_exp;
   unsigned int    exp_adjustment_count;
   if (a < 0)
   {
      out->sign = 1;
      a = -1*a;
   }
   else
   {
      out->sign = 0;
   }
   exp_adjustment_count = 0;
   out->fract = -1;
   one_too_big_exp = out->fract + 1;
   while (one_too_big_exp != 1)
   {
      exp_adjustment_count++;
      one_too_big_exp = one_too_big_exp >> 1;
   }
   out->exp = 0;
   if (a != 0)
   {
      while (a < (float)(1u<<exp_adjustment_count))
      {
         a = a*2;
         out->exp--;
      }
      while (a >= (float)(1u<<exp_adjustment_count))
      {
         a = a/2;
         out->exp++;
      }
      out->exp = exp_adjustment_count+(out->exp);
   }
   out->fract = (unsigned int)a;
   return;
}
```

The following table showing the function that was used to convert the reduced precision to standard floating point format consists of several steps. The first step in the conversion is the calculation of “exp_adjustment_count”, the variable used to convert the fixed point mantissa back to standard floating point. The second step is to calculate the sign of the standard floating point number. The third step is to convert the fixed point number back to standard floating point. This third step consists of two while loops that deal with left and right shifts of the mantissa.

Table A-3: Conversion From Reduced Precision Floating Point to Standard Floating Point

```
void conv_steves_float_to_float( struct steves_float a, float * out )
{
   struct steves_float temp;
   unsigned int      one_too_big_exp;
   unsigned int      exp_adjustment_count;
   int               sign;
   int               exponent;

   exp_adjustment_count = 0;
   temp.fract = -1;
   one_too_big_exp = temp.fract + 1;
   while (one_too_big_exp != 1)
   {
      exp_adjustment_count++;
      one_too_big_exp = one_too_big_exp >> 1;
   }

   sign = (1-2*(a.sign));
   *out = ((float)sign)*(a.fract);

   exponent = a.exp;
   while ((exponent-(int)exp_adjustment_count) < 0)
   {
      *out = (*out)/2;

      exponent++;
   }

   while ((exponent-(int)exp_adjustment_count) > 0)
   {
      *out = (*out)*2;

      exponent--;
   }

   return;
}
```

The following table showing the function that was used to convert signed integers to the reduced precision format consists of several steps. The parts of this function are very similar to the steps needed for conversion from standard floating point to the reduced precision format. The first step determines the sign. Note that this step uses a scratch variable rather than modifying the original parameter variable. This was necessary in order to avoid sign extension during the normalization step. The second step of this function is the calculation of “exp_adjustment_count”; this variable is used to both normalize the integer for the fixed point reduced precision mantissa as well as to calculate the exponent of the reduced precision representation. The two “while” loops are used to calculate the exponent and normalize the mantissa (using bit-wise shifts rather than floating point multiplication, as in the standard floating point conversion function). The next step is to assign the exponent. The final step is to assign the mantissa. In this function, the numerical truncation occurs in the bitwise shifts used in the “while” loops.

Table A-4: Conversion From Signed Integer to Reduced Precision Floating Point

```
void conv_int_to_steves_float( int a, struct steves_float * out )
{
   unsigned int    scratch;
   unsigned int    one_too_big_exp;
   unsigned int    exp_adjustment_count;

   if (a < 0)
   {        out->sign = 1;        scratch = 0 - a;     }
   else
   {        out->sign = 0;        scratch = a;     }

   exp_adjustment_count = 0;
   out->fract = -1;
   one_too_big_exp = out->fract + 1;
   while (one_too_big_exp != 1)
   {
      exp_adjustment_count++;
      one_too_big_exp = one_too_big_exp >> 1;
   }

   out->exp = 0;
   if (scratch != 0)
   {
      while (scratch < (1u<<exp_adjustment_count))
      {
         scratch = scratch << 1;
         out->exp--;
      }

      while (scratch >= (1u<<exp_adjustment_count))
      {
         scratch = scratch >> 1;
         out->exp++;
      }

      out->exp = exp_adjustment_count + out->exp;
   }

   out->fract = scratch;

   return;
}
```

The following table showing the function that was used to convert the reduced precision format to signed integers consists of several steps. The parts of this function are very similar to the steps needed for conversion from the reduced precision format to standard floating point. The first step in the conversion is the calculation of “exp_adjustment_count”, the variable used to convert the fixed point mantissa back to standard floating point. The loops that were needed for in Table **A-3** were replaced by bit-shifts in this routine resulting in a single “if” statement. This conditional statement consists of two parts. The first deals with the possibility that a left-shift is required. This suggests that the reduced precision format may not have had enough precision to hold the value that it was tasked with storing; a second interpretation of this portion of the statement is that a large number was stored in the format. The second half of the conditional statement, containing a right-shift, deals with the possibility that the integer output of this function cannot accurately represent the value stored in the reduced precision format; this is to say that the reduced precision format contains a fractional component.

Table A-5: Conversion From Reduced Precision Floating Point to Signed Integer

```
void conv_steves_float_to_int( struct steves_float a, int * out )
{
   struct steves_float temp;
   unsigned int       one_too_big_exp;
   unsigned int       exp_adjustment_count;

   exp_adjustment_count = 0;
   temp.fract = -1;
   one_too_big_exp = temp.fract + 1;
   while (one_too_big_exp != 1)
   {
      exp_adjustment_count++;
      one_too_big_exp = one_too_big_exp >> 1;
   }

   if (a.exp > (int)exp_adjustment_count)
   {
      *out = (int)(a.fract<<(a.exp-exp_adjustment_count))*(1-2*a.sign);
   }
   else
   {
      *out = (int)(a.fract>>(exp_adjustment_count-a.exp))*(1-2*a.sign);
   }

   return;
}
```

Though the code for unsigned integer conversion is not included here, it is very similar to signed integer conversion. In the routine to convert a signed integer to the reduced precision format, the sign was extracted from the signed integer input and the, now positive, integer was assigned to an unsigned integer as the first step in the routine. Since this is not necessary in the unsigned integer to reduced precision format conversion routine, a simple assignment replaces this first step (no conditional was required). The routine to convert the reduced precision format to unsigned integer is barely more complex than the routine to convert the reduced precision format to signed integer. The final conditional statement in that routine was enclosed within a check to ensure that the reduced precision format number was positive. If the reduced precision format value was negative, the output from the reduced precision format to unsigned integer conversion function was zero.

A.3 Addition Operator

There are several major obstacles to the implementation of floating point addition. The first of these obstacles is the decimal point alignment. This step is made more difficult by the fact that the maximum precision of the results should be maintained by this shift. By shifting the smaller of the two parameters to this operator, this goal can be achieved. This was implemented by comparing the two inputs and swapping them so that the second parameter was always the smaller of the two prior to aligning the decimal points (note that the exponents of the parameters is equal when their decimal points are aligned). The actual step of addition is quite simple; if the signs of the parameters are equal, add the fractions, otherwise, subtract the parameters. There are three possible outcomes from the addition/subtraction step: zero, overflow (right-shift required), and normalization (zero or more left-shifts required). If the result is zero, the entire contents of the output must be zero. If the result has overflowed (only possible when the signs are equal), a single right-shift is required and the exponent must be incremented by one. If normalization is required, a loop was used to left-shift the output and decrement the exponent until the output was normalized. These steps are illustrated in the following table.

Table A-6: Addition Operator for Reduced Precision Floating Point

```
void add_steves_float(struct steves_float a, struct steves_float b, struct steves_float * out)
{
   struct steves_float temp_out;
   unsigned int       result, one_too_big;

   if ((a.fract == 0) || (a.exp < b.exp) || ((a.exp == b.exp) && (a.fract < b.fract)))
   {       temp_out = a;       a = b;       b = temp_out;    }

   b.fract = b.fract>>(a.exp-b.exp);

   if (a.sign == b.sign)
   {       result = a.fract + b.fract;    }
   else
   {       result = a.fract - b.fract;    }

   if (result != 0)
   {
      temp_out.fract = -1;       one_too_big = temp_out.fract;       one_too_big++;
      temp_out = a;

      if (result >= one_too_big)
      {          result = result>>1;          temp_out.exp++;       }
      else
      {
         while (result < one_too_big>>1)
         {
            result = result<<1;
            temp_out.exp--;
         }
      }

      temp_out.fract = result;
   }
   else
   {       temp_out.sign = 0;       temp_out.fract = 0;       temp_out.exp = 0;    }

   *out = temp_out;

   return;
}
```

A.4 Subtraction Operator

The following table illustrates the simplicity of the subtraction operator. The addition operator is applied after toggling the second parameter's sign.

Table A-7: Subtraction Operator for Reduced Precision Floating Point

```
void sub_steves_float(struct steves_float a, struct steves_float b, struct steves_float * out)
{
   b.sign ^= 1;

   add_steves_float( a, b, out );

   return;
}
```

A.5 Multiplication Operator

Though the implementation of integer multiplication is more complex than the implementation of integer addition, the implementation of floating point multiplication is little more than the implementation of integer multiplication. The first step in implementing the multiplication was the calculation of the sign (exclusive-OR the signs). The second step was multiplying the fractions; this required an integer multiplication, a time-consuming operation (though still faster than the many steps in floating point addition). The third step was calculating the exponents (add the exponents). The last step in the multiplication operator was the only conceptually difficult part of the implementation, normalizing the result. There were three possible outcomes from the multiplication: zero, no normalization, or a single right-shift. If the result was zero, the proper zero representation was generated. If the result required no normalization, the product was right-shifted; this apparent normalization was due to the shift applied during the multiplication (the product of two normalized numbers results in either 2*n-1 bits or 2*n bits, the shift applied during the multiplication "under shifts" in the case of 2*n bits in order to preserve precision). In the single right-shift case, the exponent must be decremented and the result shifted (though this has already been performed in the "under shifted" multiplication). This routine is shown below.

Table **A-8**: Multiplication Operator for Reduced Precision Floating Point

```
void mult_steves_float(struct steves_float a, struct steves_float b, struct steves_float *
out)
{
   struct steves_float temp_out;
   unsigned __int64    temp_fract;
   unsigned int        one_too_big_exp, exp_adjustment_count;

   exp_adjustment_count = 0;
   temp_out.fract = -1;
   one_too_big_exp = temp_out.fract + 1;
   while (one_too_big_exp != 1)
   {       exp_adjustment_count++;        one_too_big_exp = one_too_big_exp >> 1;    }

   temp_out.sign   = a.sign ^ b.sign;
   temp_fract      = ((unsigned __int64)a.fract * (unsigned __int64)b.fract)>>(unsigned
__int64)(exp_adjustment_count-1);
   temp_out.exp    = a.exp + b.exp;

   if ((temp_fract&(1<<exp_adjustment_count)) == 0)
   {
      temp_out.fract = (unsigned int)temp_fract;
      temp_out.exp--;
   }
   else
   {       temp_out.fract = (unsigned int)(temp_fract>>1);    }

   if (temp_out.fract == 0)
   {       temp_out.sign = 0;       temp_out.exp = 0;    }

   *out = temp_out;

   return;
}
```

A.6 Matlab MEX Function

The MEX functions were each implemented in two parts: the Matlab entry function and the function that performed the arithmetic operation (appropriately named “do_stuff”). The entry function ensures that exactly two values are passed to the function, checks that both values are non-Complex, and allocates the return value. The function that performed the arithmetic operation converted the IEEE double-precision floating point numbers to the reduced precision format, executed the arithmetic function using the reduced precision values, and converted the result back into IEEE double-precision floating point. The following two tables are these two functions; the addition arithmetic function is shown as an example.

Table **A-9**: Matlab Entry Function

```
void mexFunction(int nlhs, mxArray *plhs[],int nrhs, const mxArray * prhs[])
{
   double *    a, b, x;
   int         result_rows[2], result_columns[2];

   if (nrhs == 2)
   {
      result_rows[0]       = mxGetM(prhs[0]);
      result_columns[0]    = mxGetN(prhs[0]);
      result_rows[1]       = mxGetM(prhs[1]);
      result_columns[1]    = mxGetN(prhs[1]);

      if ((mxIsDouble(prhs[0]) && !mxIsComplex(prhs[0])) &&
         (mxIsDouble(prhs[1]) && !mxIsComplex(prhs[1])) &&
         ((result_rows[0] == 1) && (result_columns[0] == 1)) &&
         ((result_rows[1] == 1) && (result_columns[1] == 1)))
      {
         plhs[0] = mxCreateDoubleMatrix( 1, 1, mxREAL );

         a = mxGetPr(prhs[0]);
         b = mxGetPr(prhs[1]);
         x = mxGetPr(plhs[0]);

         *x = do_stuff( *a, *b );
      }
      else
      {          mexErrMsgTxt( "Inputs must be noncomplex scalar doubles." );       }
   }
   else
   {       mexErrMsgTxt("Two inputs required.");    }

   return;
}
```

Table A-10: Addition Arithmetic Function

```
double do_stuff( double a, double b )
{
   struct steves_float x;
   struct steves_float y;
   struct steves_float z;
   float  intermediate_result;
   double result;

   conv_float_to_steves_float( (float)a, &x );
   conv_float_to_steves_float( (float)b, &y );

   add_steves_float( x, y, &z );

   conv_steves_float_to_float( z, &intermediate_result );

   result = (double)intermediate_result;

   return result;
}
```

Appendix B

PLL Implementation in Matlab

The four PLL models and the simulation used to characterize these models are described in this appendix; these models and simulation were used to determine the suitability of the PLL design for implementation as described in **4.1**. The models are the following: the infinite precision PLL, the finite precision PLL, the infinite precision PLL with reference locking, and the finite precision PLL with reference locking. Comments and white-space that were in the original files have been removed for the sake of formatting.

B.1 Infinite Precision PLL

The infinite precision PLL model consists of seven parts shown in Table **B-1**: loop filter initialization, storage pre-allocation, phase shifter calculation, phase error estimation, loop filter application, phase accumulation, and data collection. In the loop filter initialization, the loop filter coefficients are calculated as in Harris' original simulation. In storage pre-allocation, the output and intermediate storage locations are allocated so that the simulation runs in a timely manner. In the phase shifter calculation, the phase shifted input is calculated. In the phase error estimation step, the calculation of the input sine wave and the PLL output sine wave are combined using the arctangent approximation described in **4.1.4** to produce the phase error. The loop filter application filters the phase error to produce a phase step, a frequency. The phase accumulation step accumulates the phase steps to produce an estimated phase. The data collection step bundles the data generated in the model and returns it to the simulation framework.

Table **B-1**: Infinite Precision PLL

```
function [results] = dpll_mfile11( input )
theta_0=2*pi/100;                                            % filter initialization
eta=sqrt(2)/2;
k_i=(4*theta_0*theta_0)/(1+2*eta*theta_0+theta_0*theta_0);
k_p=(4*eta*theta_0)/(1+2*eta*theta_0+theta_0*theta_0);
input_derivative=0;                                          % storage pre-allocation
phase_shift_input=0;
phase_shift_output=zeros(1,length(input));
phase_error=zeros(1,length(input));
phase_step=zeros(1,length(input));
current_phase_angle=zeros(1,length(input));
int_l=0;
for i=3;length(input)
  input_derivative=input(i)-phase_shift_input;               % phase shift calculation
  phase_shift_input=input(i);
  if (~(abs(input_derivative)<0.25) & …
    (sign(input_derivative)==1)) | …
    ((abs(input_derivative)<0.25) & …
    (current_phase_angle(i-1)>=0.5))
    phase_shift_output(i)=-1*sin(acos(input(i)));
  else
    phase_shift_output(i)=sin(acos(input(i)));
  end
  tpcpa=-2*pi*current_phase_angle(i-1);                      % phase error estimation
  phase_error(i)=(input(i)*sin(tpcpa)+phase_shift_output*cos(tpa))/4;
  int_l=int_l+k_i*phase_error(i);                            % loop filter application
  phase_step(i)=k_p*phase_error(i)+int_l;
  current_phase_angle=mod(current_phase_angle(i-1)+phase_step(i),1);% phase accum.
end
results.output = cos(current_phase_angle);                   % data collection
results.input = input;
results.phase_error = phase_error;
results.phase_step = phase_step;
results.phase_angle = phase_angle;
results.phase_shift_output = phase_shift_output;
```

B.2 Finite Precision PLL

The finite precision PLL is structurally identical to the infinite precision PLL. Unlike the infinite precision PLL however, the finite precision PLL was forced to use range limited variable types that also suffered from rollover effects. Table B-2 shows the Matlab release 13 compatible code for the finite precision PLL. Note that more recent versions of Matlab have more seamless support for fixed point calculations and would not require such verbose code.

In the code below, a subtlety of the calculation can be seen in the code comments (preserved from my original Matlab code). In the prior analysis and the infinite precision model, there is a divide by four (a part of the arctangent approximation). In the code below, no such division exists outside of the comment. In fixed point calculations, a shift in the decimal place is equivalent to a binary shift. Since the division is by four, this is equivalent to a change in the location of the binary representation's decimal place. Effectively this means that the representation has changed from a "2.13" representation to a "0.15" representation. Though this is a conceptual change, no shifts or other modifications are needed to accomplish this in fixed point. This is to say that this "change" was nothing more than a footnote in the design documentation to be taken into consideration in subsequent calculations. As the comment so succinctly puts it, "and *POOF* the phase error is divided by 4".

Table **B-2**: Finite Precision PLL

```
function [results] = dpll_mfile12( input )
new_input = int8(input*128);
theta_0=2*pi/100;                                                    % filter initialization
eta=sqrt(2)/2;
k_i=intmax(‘int16’)*((4*theta_0*theta_0)/(1+2*eta*theta_0+theta_0*theta_0));
k_p=intmax(‘int16’)*((4*eta*theta_0)/(1+2*eta*theta_0+theta_0*theta_0));
input_derivative=int8(0);                                            % storage pre-allocation
phase_shift_input=int8(0);
phase_shift_output=int8(zeros(1,length(input)));
phase_error=int16(zeros(1,length(input)));
phase_step=int16(zeros(1,length(input)));
current_phase_angle=uint16(zeros(1,length(input)));
int_l=int16(0);
for i=3;length(new_input)
    input_derivative=new_input(i)-phase_shift_input;                % phase shift calculation
    phase_shift_input=new_input(i);
    if (~(abs(input_derivative)<0.25*intmax(‘int8’)) & …
        (sign(input_derivative)==1)) | …
        ((abs(input_derivative)<0.25* intmax(‘int8’)) & …
        (current_phase_angle(i-1)>=0.5* intmax(‘uint16’)))
        phase_shift_output(i)= int8(double(bitcmp( …
        uint16(sin(acos(double(new_input(i))/(double(intmax('int8'))+1)))*127),8))-256);
    else
        phase_shift_output(i)=int8(sin(acos(double(new_input(i))/ …
        (double(intmax('int8'))+1)))*127);
    end
    tpcpa=-2*pi*double(current_phase_angle(i-1)/256)/256;
    phase_error(i)= int16( ...
                (double(new_input(i))*double(int8(sin(tpcpa)*128)) ...
                 + ...
                 double(phase_shift_output(i))*double(int8(cos(tpcpa)*128))) ...
                 );
    % I wave my magic wand and *POOF* phase error is divided by 4
    % loop filter application on the following three lines
    int_l=int_1+int16(double(k_i)/double(intmax('int16'))*double(phase_error(i)));
    phase_step(i)=int16(double(k_p)/double(intmax('int16'))* …
    double(phase_error(i)))+int_1;
    % phase accumulation on the following two lines
    current_phase_angle=uint16(mod(double(current_phase_angle(i-1))+ …
    double(phase_step(i)),intmax('uint16')+1));
end  % data collection as in the Infinite Precision PLL (excluded for brevity)
```

B.3 Infinite Precision PLL with Reference Locking

Rather than duplicating Table B-1, this section will describe the three changes to that code that created the infinite precision reference locking PLL model. These changes affected the initialization, the main loop (consisting of “phase shift”, “phase error” and “loop filter” calculations), and the data collection portions of the model.

The initialization section was modified to include the code in Table B-3. The “reference_stop” variable holds the angular velocity of the reference PLL (the frequency). The “reference_phase_angle” variable holds the current phase of the reference PLL. The “reference_locked” variable flags whether the PLL has locked onto the reference frequency or is still searching.

Table B-3: Initialization Additions to the Infinite Precision PLL

```
reference_step = zeros(1,length(input));
reference_phase_angle = zeros(1,length(input));
reference_locked = false;
```

The main loop additions to the infinite precision PLL are shown in Table B-4. Though the initialization additions need not be performed in any particular order, the additions shown in Table B-4 must be after the “phase accumulation” step.

The calculation of the reference PLL occurs in two steps, as shown. The first step determines whether the PLL is locked or not (maximum phase error less than 0.001 for 150 samples, determined by experimentation rather than calculation). The second step updates the reference phase step and current angle (when the PLL is not locked, the reference and tracking PLL are equivalent).

Table B-4: Main Loop Additions to the Infinite Precision PLL

```
% reference dss
% Once the target has been locked (based on the phase error),
% disconnect the reference PLL and let the tracking PLL continue.
% Phase lock for my testing appears to be abs(phase error) < 0.001 for a
% "little while" <- that being ~100 samples.
if ~reference_locked && (i > 150) && (max(abs(phase_error((i-150):i))) < 0.001)
   reference_locked = true;
end
if ~reference_locked
   reference_step(i) = phase_step(i);
   reference_phase_angle(i) = mod(reference_phase_angle(i-1)+phase_step(i),1);
else
   reference_step(i) = reference_step(i-1);
   reference_phase_angle(i) = mod(reference_phase_angle(i-1)+reference_step(i),1);
end
```

The third addition to the infinite precision PLL is a single line addition to the data collection phase. By adding the line “results.reference_phase_angle = reference_phase_angle;” to the model, the reference phase angle is reported to the simulation environment for further experimentation and analysis.

B.4 Finite Precision PLL with Reference Locking

Rather than duplicating Table B-2, this section will describe the three changes to that code that created the finite precision reference locking PLL model. These changes affected the initialization, the main loop (consisting of “phase shift”, “phase error” and “loop filter” calculations), and the data collection portions of the model.

The initialization section was modified to include the code in Table B-5. The “reference_stop” variable holds the angular velocity of the reference PLL (the frequency). The “reference_phase_angle” variable holds the current phase of the reference PLL. The “reference_locked” variable flags whether the PLL has locked onto the reference frequency or is still searching.

Table **B-5**: Initialization Additions to the Finite Precision PLL

```
reference_step = int16(zeros(1,length(input)));
reference_phase_angle = uint16(zeros(1,length(input)));
reference_locked = false;
```

The main loop additions to the finite precision PLL are shown in Table **B-6**. Though the initialization additions need not be performed in any particular order, the additions shown in Table **B-6** must be after the “phase accumulation” step.

Table **B-6**: Main Loop Additions to the Finite Precision PLL

```
if ~reference_locked && (i > 150) && (max(abs(phase_error((i-150):i))) < 2500)
   reference_locked = true;
end
if ~reference_locked
   if (i > 64)
      reference_step(i) = uint16(sum(double(phase_step((i-64):i)))/64);
   else
      reference_step(i) = phase_step(i);
   end
   reference_phase_angle(i) = uint16(mod(double(reference_phase_angle(i-1))+ …
      double(phase_step(i)),intmax('uint16')+1));
else
   reference_step(i) = reference_step(i-1);
   reference_phase_angle(i) = uint16(mod(double(reference_phase_angle(i-1))+ …
      double(reference_step(i)),intmax('uint16')+1));
end
```

The third addition to the finite precision PLL is a single line addition to the data collection phase. By adding the line “results.reference_phase_angle = reference_phase_angle;” to the model, the reference phase angle is reported to the simulation environment for further experimentation and analysis.

B.5 Simulation Environment

The simulation environment was able to generate waveform input to the various PLLs. This input could be injected with white noise by the simulation environment at various signal-to-noise ratios. In the absence of white noise, the waveform generated by the simulation environment was continuous. In the absence of white noise, the waveform generated by the simulation environment also had continuous phase (no phase jumps when transitioning from one frequency to another).

The simulation environment can be divided into two sections: test specification and test generation. The abbreviated code for the test specification portion is shown in Table **B-7**. The code for the test generation is shown in Table **B-8**.

Table B-7: Simulation Environment Test Specification

```
clear
test_selection = 2;

% define the input intervals and their frequencies
if test_selection == 1
   % test various frequencies and phases
   % omitted for brevity (the frequencies were arbitrary for this test)
elseif test_selection == 2
   % this is convenient for determining the operational range,
   % performs a frequency sweep
   for i=1:20
      interval(i).frequency = pi*i/21;      interval(i).amplitude = 1;
      interval(i).offset    = 0;      interval(i).duration  = 1000;
   end
elseif test_selection == 3
   % a small phase shift test
   interval(1).frequency   = pi/10;   interval(1).amplitude   = 1;
   interval(1).offset      = 0;   interval(1).duration    = 400;
   interval(2).frequency   = pi/10;   interval(2).amplitude   = 1;
   interval(2).offset      = pi;   interval(2).duration    = 400;
elseif test_selection == 4
   % a sweep over a restricted frequency range with higher resolution than
   % test_selection 2 (within that range)
   for i=1:20
      interval(i).frequency = pi*i/21*6/20;      interval(i).amplitude = 1;
      interval(i).offset    = 0;      interval(i).duration  = 1000;
   end
elseif test_selection == 5
   % test amplitude variation in the input sequence
   interval(1).frequency   = pi/10;   interval(1).amplitude   = 1;
   interval(1).offset      = 0;   interval(1).duration    = 200;
   interval(2).frequency   = pi/10;   interval(2).amplitude   = .75;
   interval(2).offset      = 0;   interval(2).duration    = 200;
else
   % test various phase steps, useful for testing phase shift sensitivity
   for i=1:20
      interval(i).frequency = pi*3/21;      interval(i).amplitude = 1;
      interval(i).offset    = pi*(i-1)/20;      interval(i).duration  = 1000;
   end
end
```

Table **B-8**: Simulation Environment Test Generation

```
% determine the length of the total input sequence
samples = 0;
for i = 1:length(interval)
   samples = samples + interval(i).duration;
end

% noise component in the input sample stream (uncomment the “SNR” and “noise” lines)
% SNR=20*log10(Asignal/Anoise)
% Asignal = 1 (by definition)
% Anoise=1/(10^(SNR/20))
%SNR = 30;
%noise = 2*(rand(1,samples)-1/2)/(10^(SNR/20));
noise = zeros(1,samples);

%-- create the input waveform
current_phase = 0;
current_start_sample = 1;
input = noise;
for i = 1:length(interval)
   current_phase = interval(i).offset + current_phase;
   input(current_start_sample:(current_start_sample+interval(i).duration-1)) = ...
      input(current_start_sample:(current_start_sample+interval(i).duration-1)) + ...
      sin(interval(i).frequency*[1:interval(i).duration] +
current_phase)*interval(i).amplitude;
   current_phase = mod(interval(i).frequency*interval(i).duration + current_phase -
interval(i).offset, 2*pi);
   current_start_sample = current_start_sample + interval(i).duration;
end

% normalize the input to [-1,1]
input = input./max(abs(input));

% run the DPLL given the input waveform
results = dpll_mfile11(input);
```